JavaScript忍者秘籍（第2版）读书笔记

第一章 热身

- **主要概念**

  > - 函数是一等公民（一级对象）--- 在JavaScript中，函数与其他对象共存，并且能够像任何其他对象一样地使用。
  > - 函数闭包从根本上例证了函数之于JavaScript的重要性。
  > - 作用域
  > - 基于原型的面向对象

  生成器 -- 一种可以基于一次请求生成多次值的函数，在不同请求之间也能挂起执行。

  Promise -- 让我们更好地控制异步代码。

  代理 -- 让我们控制对特定对象的访问。

  高级数组方法 -- 书写更加优雅的数组处理函数。

  Map -- 用于创建字典集合；*Set* -- 处理仅包含不重复项目的集合。

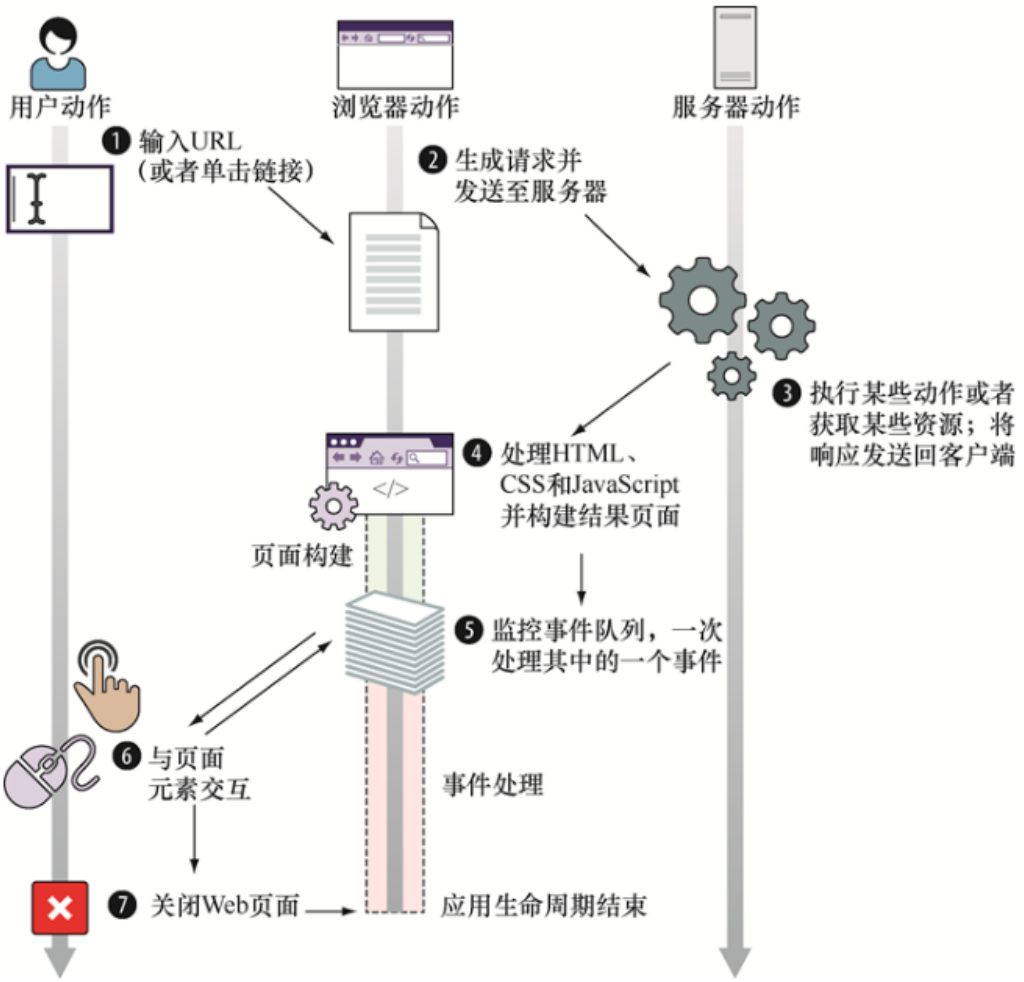
  正则表达式 -- 简化用代码书写起来很复杂的逻辑。

  模块 -- 把代码分为较小的可以自包含的片段，使项目更易于管理。

第二章 运行时的页面构建过程

- **Web应用的生命周期步骤**

  > - 典型客户端Web应用的生命周期从用户在浏览器地址栏输入一串URL，或单击一个链接开始。
  > - 从用户的角度来说，浏览器构建了发送至服务器（序号2）的请求，该服务器处理了请求（序号3）并形成了一个通常由HTML、CSS和JavaScript代码所组成的响应。当浏览器接收了响应（序号4）时，我们的客户端应用开始了它的生命周期。



其由**两个步骤**组成：**页面构建**和**事件处理**

1．页面构建——创建用户界面；
2．事件处理——进入循环（序号5）从而等待事件（序号6）的发生，发生后调用事件处理器。

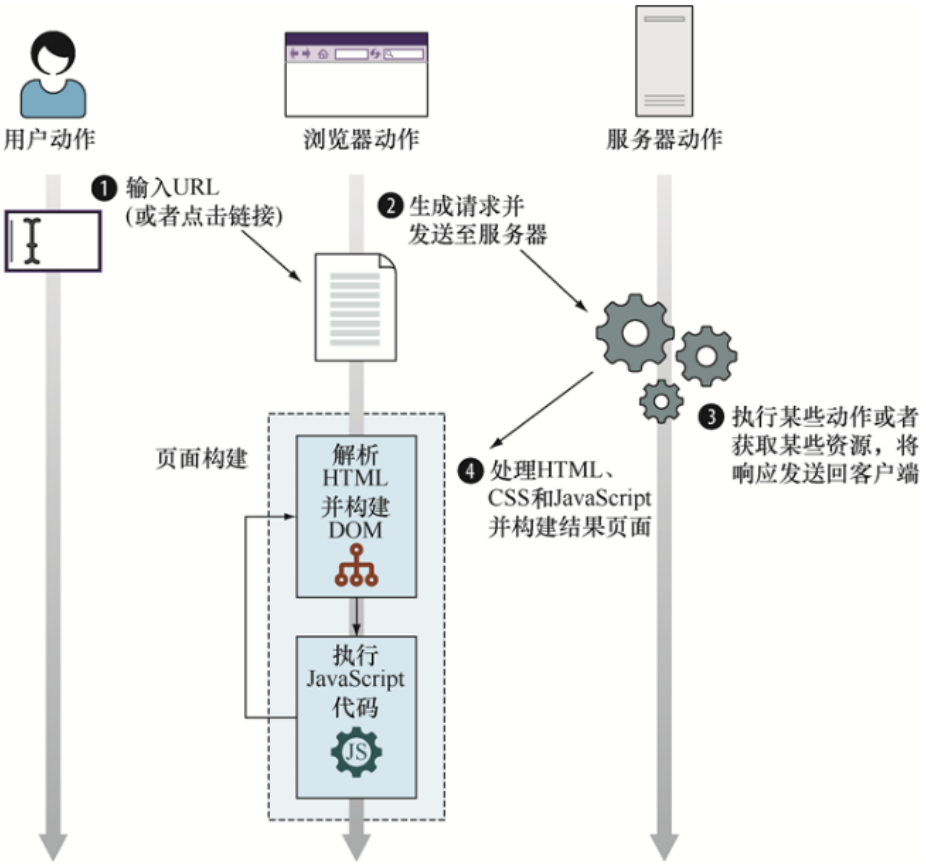


图2.3　页面构建阶段从浏览器接收页面代码开始。其执行分为两个步骤：HTML解析和DOM构建，以及JavaScript代码的执行

- **从HTML代码到Web页面的处理过程**
  - **HTML解析和DOM构建**

    > 页面构建阶段**始于**浏览器接收HTML代码时，该阶段为浏览器构建页面UI的基础。通过解析收到的HTML代码，构建一个个HTML元素，**构建DOM**。在这种对HTML结构化表示的形式中，每个**HTML元素**都被当作一个**节点**。

  - **执行JavaScript代码**
    - 所有包含在脚本元素中的JavaScript代码由浏览器的**JavaScript引擎**执行。

    > 由于代码的主要目的是提供动态页面，故而浏览器**通过全局对象提供**了一个**API** 使JavaScript引擎可以与之**交互并改变页面内容**。

    - **windows对象**：一个全局对象，**代表了一个页面的窗口。**它是获取所有其他全局对象、全局变量（包含用户定义对象）和浏览器API的**访问途径。**

      > 全局window对象最重要的属性是document，它代表了当前页面的DOM。通过使用这个对象，JavaScript代码就能在任何程度上改变DOM，包括修改或移除现存的节点，以及创建和插入新的节点。

    - **JavaScript代码的不同类型**：**JavaScript全局代码和函数代码。** 包含在函数内的代码叫作函数代码，而在所有函数以外的代码叫作全局代码。

      > 全局代码由JavaScript引擎以一种直接的方式自动执行，每当遇到这样的代码就一行接一行地执行。

    在页面构建阶段遇到了脚本节点则会停止HTML到DOM的构建，开始执行JavaScript代码，即全局JavaScript代码以及由全局代码执行中调用的函数代码。

- **JavaScript代码的执行顺序**

  全局代码由JavaScript引擎以一种直接的方式自动执行，每当遇到这样的代码就一行接一行地执行。若想执行函数代码，则必须被其他代码调用：既可以是全局代码，也可以是其他函数，还可以由浏览器调用。

- **与事件交互**

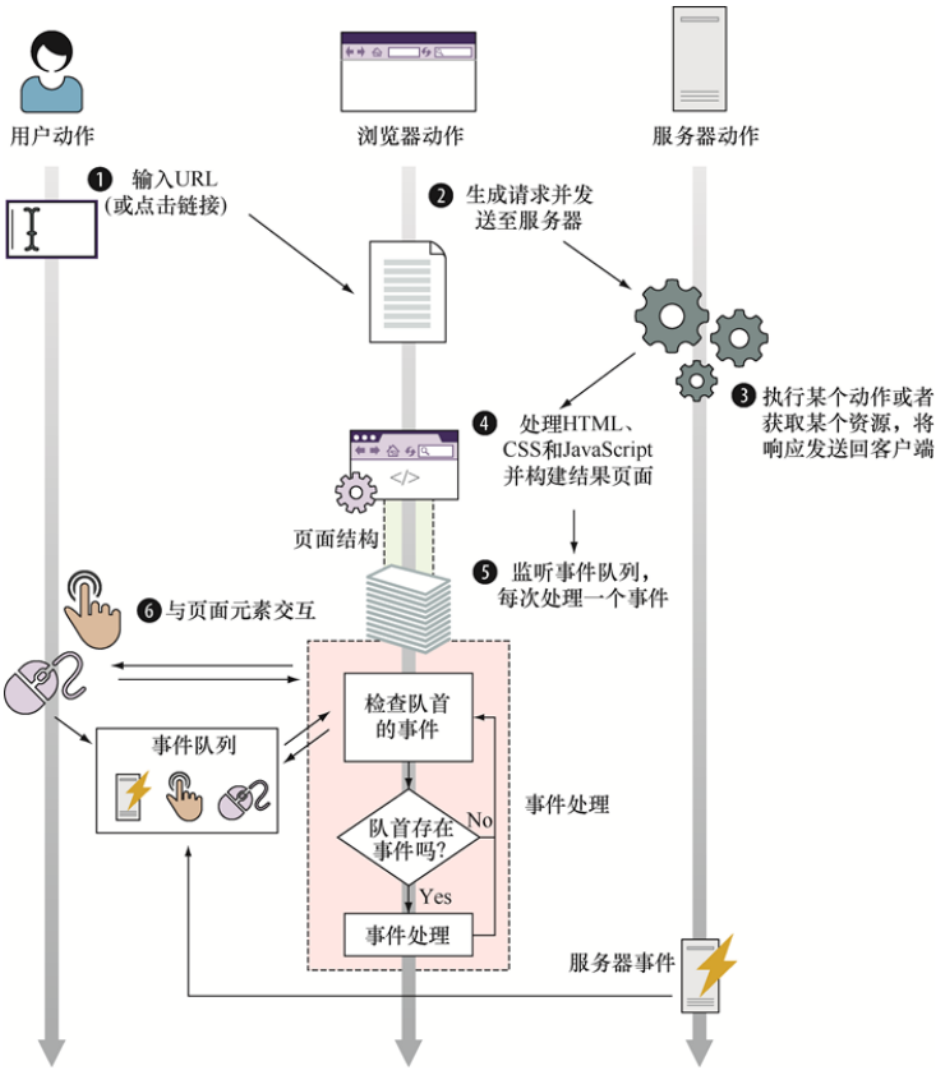
  - **浏览器执行环境的核心思想**基于：同一时刻只能执行一个代码片段，即所谓的**单线程执行模型。**

  > 客户端Web 应用是一种GUI应用，也就是说这种应用会对不同类型的事件作响应，如鼠标移动、单击和键盘按压等。因此，在页面构建阶段执行的JavaScript代码，除了会影响全局应用状态和修改DOM外，还会**注册事件监听器（或处理器）**。这类监听器会在事件发生时，由浏览器调用执行。有了这些事件处理器，我们的应用也就有了**交互能力。**

  - 浏览器通过**事件队列**来跟踪已经发生但尚未处理的事件。

    > 一次只能处理一个事件，所以我们必须格外注意处理所有事件的总时间。执行需要花费大量时间执行的事件处理函数会导致Web应用无响应！

    - 浏览器检查事件队列头；
    - 如果浏览器没有在队列中检测到事件，则继续检查；
    - 如果浏览器在队列头中检测到了事件，则取出该事件并执行相应的事件处理器（如果存在）。在这个过程中，余下的事件在事件队列中耐心等待，直到轮到它们被处理。



  - **事件是异步的**

我们对事件的处理，以及处理函数的调用是异步的。事件处理的概念是Web应用的核心，在事件能被处理之前，代码必须要告知浏览器我们要处理特定事件。

- **注册事件处理器** （两种方法）
  - 把函数赋值给特殊属性是一种简单而直接的注册事件处理器方式。

```
document.body.onclick = function () {};
```

  - addEventListener()方法

```
<script>
   document.body.addEventListener ("mousemove", function ()  {
// 为mousemove事件注册处理器
      var second = document.getElementById ("second") ;
      addMessage (second, "Event: mousemove") ;
   }) ;

   document.body.addEventListener ("click", function () {// 为
click事件注册处理器
      var second = document.getElementById ("second") ;
      addMessage (second, "Event: click") ;
   }) ;
</script>

```

- **处理事件**

事件处理背后的的主要思想是：当事件发生时，浏览器调用相应的事件处理器。

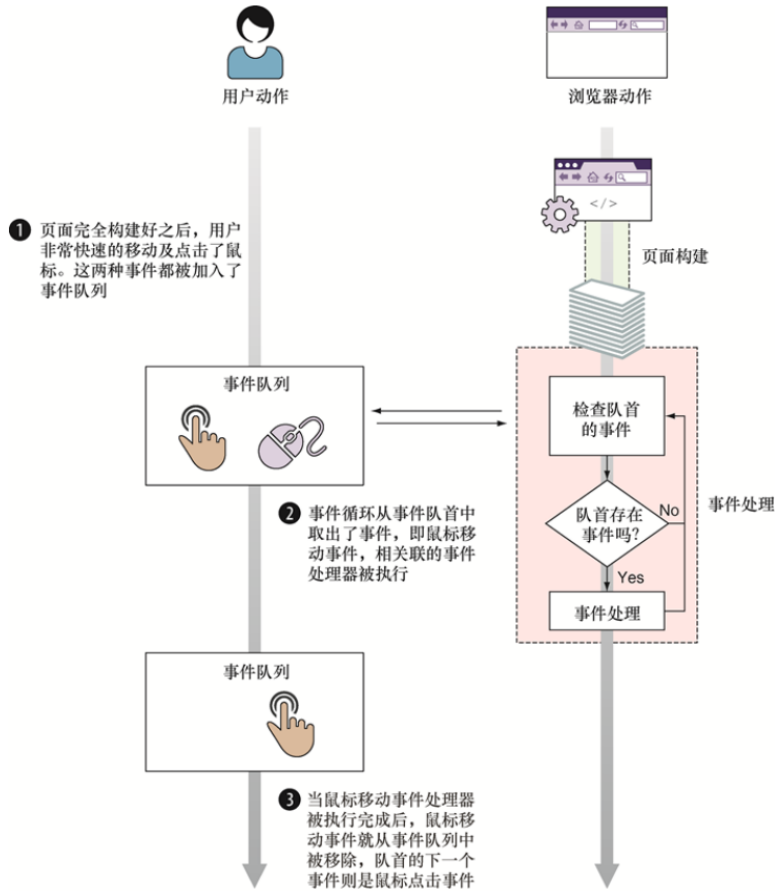


图2.9　两个事件——鼠标移动和单击中的事件处理阶段示例

总结

浏览器接收的HTML代码用作创建DOM的蓝图，它是客户端Web应用结构的内部展示阶段。我们使用JavaScript代码来动态地修改DOM以便给Web应用带来动态行为。

客户端Web应用的执行分为两个阶段。页面构建代码是用于创建DOM的，而全局JavaScript代码是遇到script节点时执行的。在这个执行过程中，JavaScript代码能够以任意程度改变当前的DOM，还能够注册事件处理器——事件处理器是一种函数，当某个特定事件（例如，一次鼠标单击或键盘按压）发生后会被执行。注册事件处理器很容易：使用内置的addEventListener方法。事件处理——在同一时刻，只能处理多个不同事件中的一个，处理顺序是事件生成的顺序。事件处理阶段大量依赖事件队列，所有的事件都以其出现的顺序存储在事件队列中。事件循环会检查实践队列的队头，如果检测到了一个事件，那么相应的事件处理器就会被调用。

第三章 新手的第一堂函数课：定义与参数

- JavaScript中最关键的概念是：函数是第一类对象，或者说它们被称为一等公民。**函数与对象共存，函数也可以被视为其他任意类型的JavaScript对象。**
- **函数的定义**

  JavaScript提供了**4类定义函数的方式**：
  - **函数声明和函数表达式**
    - 函数声明

      > 每个函数声明以**强制性**的function**开头**，其后紧接着**强制性**的**函数名**，以及括号和括号内一列以逗号分隔的可选参数名。**函数体是一列可以为空的表达式，这些表达式必须包含在花括号内。**除了这种形式以外，每个函数声明还必须包含一个条件：作为一个单独的JavaScript语句，函数声明必须独立。

      ```
      //函数声明
      //在全局代码中定义函数
      function word() {
          return "hello world";
      }
      //在函数内部定义一个新的函数
      function word() {
          function printword() {
              return "hello world";
          }
      }
      ```

    - **函数表达式**

      > 一种总是其他表达式的一部分的函数（作为赋值表达式的右值，或者作为其他函数的参数）叫作函数表达式。

      ```
      var a = function() {
          return "a";
      };
      ```

    - 对于函数声明来说，函数名是强制性的，而对于函数表达式来说，函数名则完全是可选的。
    - **立即函数**

> 我们首先创建了一个函数，然后立即调用这个新创建的函数。这种函数叫作**立即调用函数表达式（IIFE）**，或者**简写为立即函数**。这一特性能够模拟JavaScript中的模块化。

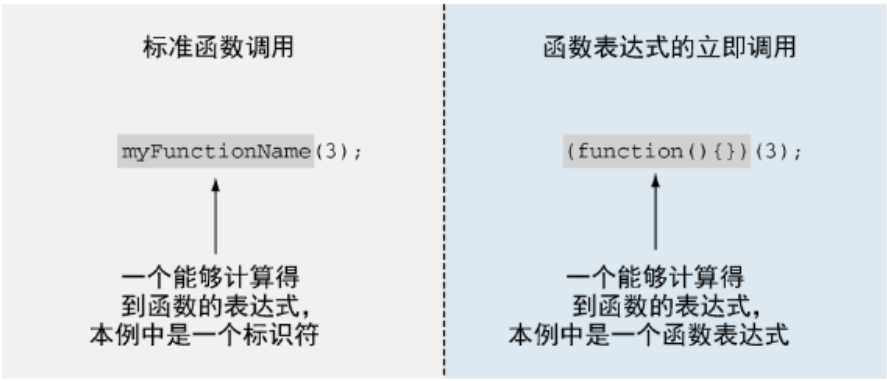


图3.5　标准函数的调用和函数表达式的立即调用的对比

> JavaScript解析器必须能够轻易区分函数声明和函数表达式之间的区别。为了避免错误，函数表达式要放在括号内，为JavaScript解析器指明它正在处理一个函数表达式而不是语句。

`回调函数和立即函数的关系`

回调函数和立即函数是两个不同的概念，它们之间没有直接的关系。下面分别介绍一下这两个概念：

1. 回调函数（Callback Function）：是一种函数，它作为参数传递给另一个函数，并且在另一个函数执行完毕后被调用。回调函数通常用于异步编程，用于在某个异步任务完成后执行一些操作，比如处理返回的数据。

举个例子，在JavaScript中，常用的异步编程方式是通过回调函数来实现。比如，使用jQuery库的ajax方法发送HTTP请求，可以通过回调函数处理返回的数据：

```
code$.ajax({
  url: "example.com/data",
  success: function(data) {
    // 处理返回的数据
  }
});
```

2.立即函数（Immediately Invoked Function Expression，IIFE）：是一种函数表达式，它定义后立即执行。立即函数通常用于创建一个私有作用域，以避免变量污染全局作用域。

举个例子，在JavaScript中，可以使用立即函数来创建一个私有作用域，以避免变量污染全局作用域：

```
code(function() {
  var a = 1; // 在私有作用域中定义变量a
  console.log(a); // 输出1
})(); // 定义后立即执行
```

回调函数和立即函数都是JavaScript中常用的编程方式，它们之间没有直接的关系。但是，在某些情况下，可以使用立即函数作为回调函数来实现一些特定的功能，比如将立即函数作为回调函数传递给setTimeout函数，实现一次性定时器的功能：

```
codesetTimeout(function() {
  (function() {
    console.log("立即函数被调用了");
  })();
}, 1000);
```

在上面的例子中，立即函数被作为回调函数传递给了setTimeout函数，表示在1000毫秒后执行。当定时器触发时，立即函数会被调用，并输出一条信息到控制台。

- **箭头函数**

> 箭头函数的定义以一串可选参数名列表开头，参数名以逗号分隔。如果没有参数或者多余一个参数时，参数列表就必须包裹在括号内。但如果只有一个参数时，括号就不是必须的。参数列表之后必须跟着一个胖箭头符号，胖箭头操作符后面有两种可选方式。如果要创建一个简单函数，那么可以把表达式放在这里（可以是数学运算、其他的函数调用等），则该函数的返回值即为此表达式的返回值。当箭头函数没那么简单从而需要更多代码时，箭头操作符后则可以跟一个代码块。

```
//简单的箭头函数
var greet = name => "Greetings" + name;
```

```
//复杂的箭头函数
var greet = name => {
    var num = 'Greetings';
    return num + name;
}
```

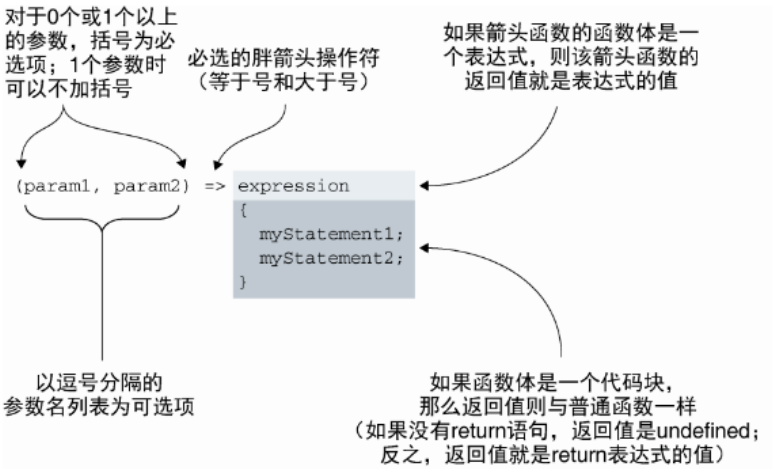


图3.6　箭头函数的语法

- 函数构造函数
- 生成器函数

- **函数的实参和形参**

> 形参是我们**定义函数时**所列举的变量。
> 实参是我们**调用函数时**所传递给函数的值。

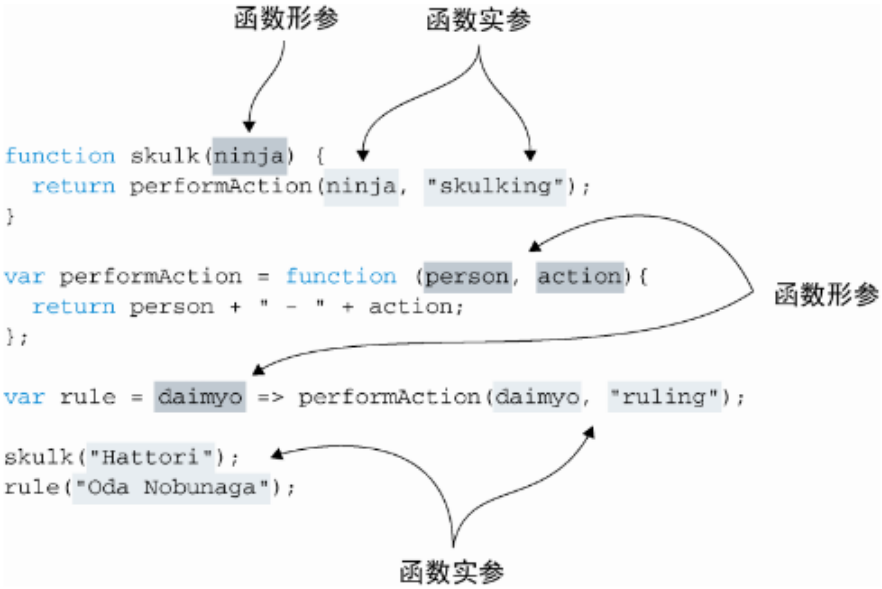


图3.7　函数形参和函数实参的不同点

- 实参以函数形参指定的顺序赋值给函数形参，额外的实参不会赋值给任何形参。
- 如果形参的数量大于实参，那么那些没有对应实参的形参则会被设为undefined。
- **剩余参数和默认参数**

```
//剩余参数以...为前缀
function multMax(first, ...remainNumbers) {
    var sorted = remainNumbers.sort(function(a,b) {
        return b - a;
    })
    return first * sorted[0];
}
```

  - 只有**函数的最后一个参数才能是剩余参数。**试图把省略号放在不是最后一个形参的任意形参之前都会报错，错误以SyntaxError: parameter after rest parameter 的形式展现。
  - JavaScript不支持函数重载(再定义一个名字相同但参数不同的函数); **JavaScript创建默认参数的方式是为函数的形参赋值。**

```
function performAction(ninja, action = "skulking"){ //ES6中可以为函数的形参赋值
   return ninja + " " + action;
}
```

- **总结**
  - JavaScript可以看做是函数式语言，书写复杂代码。
  - 函数作为第一类对象，具有其他对象一样的功能：
    - 通过字面量创建
    - 赋值给变量或者属性
    - 作为函数参数传递
    - 作为函数的结果返回

        - 赋值给属性和方法
    - 回调函数是被代码随后“回来调用”的函数，它是一种常用的函数，特别是在事件处理场景下。
    - 函数具有属性，这些属性能够储存任何信息。
    - 有很多不同类型的函数：函数声明、函数表达式、箭头函数、函数生成器等。
    - 函数声明和函数表达式是两种主要的函数类型。函数声明必须有函数名，在代码中必须作为一个独立的语句存在。函数表达式可以不具有函数名，但此时它必须作为其他语句的一部分。
    - 箭头函数是ES6的新增特性，可以让我们使用更简洁的方式来定义函数。
    - 形参是定义函数时列出的变量，实参是函数调用时传递给函数的值。
    - 函数的形参与实参长度可以不同。
        - 未赋值的形参求值得到undefined
        - 传入的额外实参不会被赋给任何一个命名参数
    - 剩余参数和默认参数是新增特性。
        - 剩余参数 --- 不与任何参名相匹配的额外实参可以通过剩余参数来引用
        - 默认参数 --- 函数调用时，若没有传入参数，默认参数可以给函数提供缺省的参数值

第四章 函数进阶：理解函数调用

- **函数中两个隐含参数arguments和this**，它们会被静默地传递给函数，并且可以向函数体内显式的声明的参数一样被访问。

  > 参数this表示被调用函数的上下文对象，而arguments参数表示函数调用过程中传递的所有参数。

- **隐式函数参数 --- arguments**

  arguments参数是传递给函数的所有参数集合。无论是否有明确定义对应的形参，通过它我们都可以访问到函数的所有参数。**借此可以实现原生JavaScript并不支持的函数重载特性**，而且可以实现接收参数数量可变的可变函数。

    - **隐式函数参数 arguments 是对象，不是数组。**它具有“*length*”属性，可以通过数组下标的方式访问到每一个参数值

      > arguments对象的主要作用是允许我们访问传递给函数的所有参数，即便部分参数没有和函数的形参关联也无妨。

    - arguments对象可以作为函数参数的别名

```
function infiltrate(person) {
  assert(person === 'gardener',
    'The person is a gardener');
  assert(arguments[0] === 'gardener',
    'The first argument is a gardener'); //检查person参数的值等于gardener，并作为第一个参数被传入

  arguments[0] = 'ninja';

  assert(person === 'ninja',
    'The person is a ninja now');
  assert(arguments[0] === 'ninja',
```

```
    'The first argument is a ninja'); //改变argument对象的值也会改变相应的形参

  person = 'gardener';

  assert(person === 'gardener',
    'The person is a gardener once more');
  assert(arguments[0] === 'gardener',
    'The first argument is a gardener again'); //这两种方式下，别名都正常工作了
}

infiltrate("gardener");
```

> 改变了arguments对象的值，同时也会影响对应的函数参数;改了某个参数值，会同时影响参数和arguments对象。

- **使用严格模式避免使用arguments别名**

- **this参数：函数上下文**

> **this参数是面向对象JavaScript编程的一个重要组成部分，代表函数调用相关联的对象。因此，通常称之为函数上下文。**

  - 函数调用（4种方式）
    - 作为函数直接被调用

> 如果一个函数没有作为方法、构造函数或者通过apply和call调用的话，我们就称之为作为函数被直接调用。

```
function ninja() {};
ninja();//函数定义作为函数被调用

var samurai = function(){};
samurai();//函数表达式作为函数被调用
(function(){})()  //会被立即调用的函数表达式，作为函数被调用
//当以这种方式调用时，函数上下文（this关键字的值）有两种可能性：在非严格模式下，它将是全局上下文（window对象），而在严格模式下，它将是undefined。
```

    - 作为方法被调用

> 当一个函数被赋值给一个对象的属性，并且通过对象属性引用的方式调用函数时，函数会作为对象的方法被调用。

```
var ninja = {};
ninja.skulk = function(){};
ninja.skulk();//这种情况下函数被称为方法
```

当函数作为某个对象的方法被调用时，该对象会成为函数的上下文，并且在函数内部可以通过参数访问到。

二者区别： JavaScript中函数作为函数被调用和作为方法被调用的区别**主要在于函数被调用时的上下文对象不同。**

当函数作为函数被调用时，它的上下文对象是全局对象（在浏览器中为window对象）。这种情况下，函数内部的this指向全局对象。

```
function myFunction() {
  console.log(this); // 输出全局对象（在浏览器中为window对象）
}

myFunction(); // 函数作为函数被调用

```

当函数作为对象的方法被调用时，它的上下文对象是该对象。这种情况下，函数内部的this指向该对象。

```
var myObject = {
  myMethod: function() {
    console.log(this); // 输出myObject对象
  }
};

myObject.myMethod(); // 函数作为方法被调用

```

- 作为构造函数调用

> 函数的构造器，它可以通过字符串来构造一个新的函数。

```
//它将创建一个函数，它包含两个形参a和b，函数的返回结果是两者的和。
new Function('a','b','return a+b');
```

> 构造函数，是我们用来创建和初始化对象实例的函数。

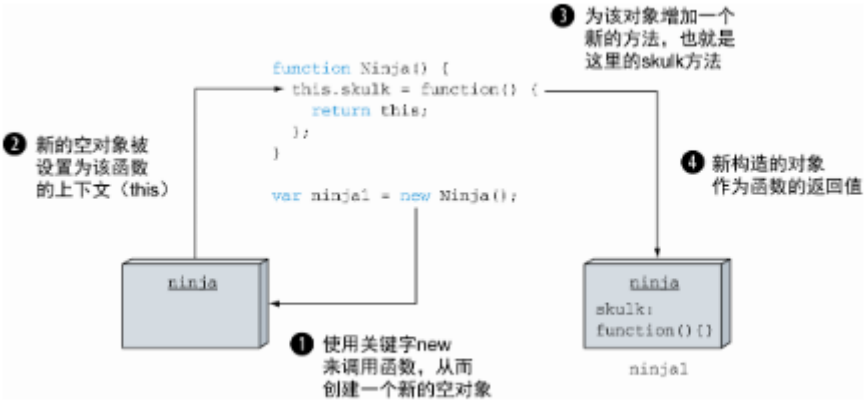
```
*使用构造函数来实现通用对象*
function Ninja() { //构造函数
  this.skulk = function() {
     return this;
  };
}
//构造函数创建一个对象，并在该对象也就是函数上下文上添加一个属性skulk。这个skulk方法再次返回函数上下文，从而能让我们在函数外部检测函数上下文

var ninja1 = new Ninja();
var ninja2 = new Ninja();  // 通过关键字new调用构造函数从而创建两个新对象。
```

构造函数只是一些使用**new操作符**时被调用的函数，使用关键字new调用函数会触发以下几个动作。

`1.创建一个新的空对象。`

`2.该对象作为this参数传递给构造函数，从而成为构造函数的函数上下文。`

`3.新构造的对象作为new运算符的返回值。`



构造函数的返回值

> 如果构造函数返回一个对象，则该对象将作为整个表达式的值返回，而传入构造函数的this将被丢弃。
> 如果构造函数返回的是非对象类型，则忽略返回值，返回新创建的对象。

```
function foo(){
      var obj = {'name': 'beipiao'};
      return obj;
      // return [1,2,3]  // 若是直接返回数组[1,2,3]， result输出[1,2,3]
}
var result = new foo();
result;  //  {'name': 'beipiao'}

function foo(){
      var obj = {'name': 'beipiao'};
      return 10;
}
var result = new foo();
result;  // foo对象
```

- 使用apply和call方法调用

> JavaScript为我们提供了一种调用函数的方式，从而可以**显式地指定任何对象作为函数的上下文**。我们可以使用每个函数上都存在的这两种方法来完成：apply和call。

使用apply方法调用函数需要传递两个参数：作为函数上下文的对象，一个数组形式的参数。

使用call方法调用函数需要传递两个参数：作为2函数上下文的对象，一个列表形式的参数。

```
function sum() {
    var result = 0;
    for(var i = 0; i < arguments.length; i++) {
result += arguments[i];}
    this.result = result;
}

//测试对象 初始值为空
var sum1 = {};
var sum2 = {};

sum.apply(sum1,[1,2,3,4]);//applyf方法调用
sum.call(sum2,5,6,7,8);//call方法调用
```

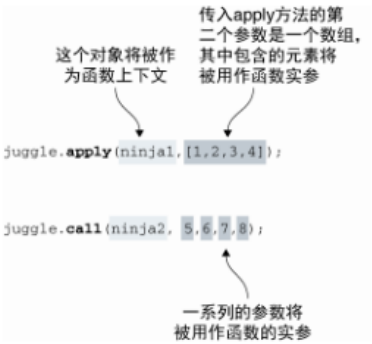


图4.3　传入call 和 apply 方法的第一个参数都会被作为函数上下文，不同处在于后续的参数。apply方法只需要一个额外的参数，也就是一个包含参数值的数组；call方法则需要传入任意数量的参数值，这些参数将用作函数的实参

> call和apply这两个方法对于我们要特殊指定一个函数的上下文对象时特别有用，在执行回调函数时可能会经常用到。

- **解决函数上下文的问题**
  - 使用箭头函数绕过函数上下文

    > 箭头函数自身不含上下文，从定义时的所在函数继承上下文。

```
const obj = {
  name: 'Alice',
  sayHello: function() {
    console.log(`Hello, my name is ${this.name}`);
  },
  sayHelloArrow: () => {
    console.log(`Hello, my name is ${this.name}`);
  }
};

obj.sayHello(); //函数上下文是obj对象本身， 输出 "Hello, my name is Alice"

obj.sayHelloArrow(); // 输出 "Hello, my name is undefined"
//函数上下文是定义 obj 对象的父级作用域，因此它不能访问 obj 对象的属性，导致输出 undefined。
```

- 使用bind方法

  > bind方法创建的新函数与原始函数的函数体相同，新函数被绑定到指定的对象上。 新函数的 `this` 值将永久地被设置为指定的对象。

  ```
  //bind语法   thisArg为指定的对象 如果使用new运算符构造绑定函数，则忽略该值。
  function.bind(thisArg[, arg1[, arg2[, ...]]])
  ```

  例子：

  ```
  const obj = {
    name: 'Alice',
    sayHello: function() {
      console.log(`Hello, my name is ${this.name}`);
    }
  };

  const sayHelloToBob = obj.sayHello.bind({ name: 'Bob' });

  sayHelloToBob(); // 输出 "Hello, my name is Bob"

  ```

  这个例子中我们定义了一个名为*obj*的对象，其中包含了一个名为*sayHello*的方法。然后使用*bind*方法创建了一个名为*sayHelloToBob*的新函数，并将其函数上下文绑定到一个包含*name*属性的新对象上。在我们调用*sayHelloToBob*方法时，输出 "Hello, my name is Bob"， 因为新函数的函数上下文已经被绑定到了 `{ name: 'Bob' }` 对象上。

- **总结**
  - 当调用函数时，除了传入在函数定义中显示声明的参数外，同时还传入两个隐式参数：arguments和this。
    - arguments参数是传入函数的所有参数的集合，具有length属性，表示传入参数的个数，通过arguments参数还可以获得那些与函数形参不匹配的参数。在非严格模式下，arguments对象是函数参数的别名，修改arguments对象会修改函数实参，可以通过严格模式避免修改函数实参。
    - this表示函数上下文，即与函数调用相关联的对象。函数定义方式和调用方式决定了this的取值。
  - 函数的调用方式有4种。
    - 作为函数直接调用：sum();
    - 作为方法调用：obj.sum();
    - 作为构造函数调用：new sum();
    - 通过apply和call方法调用：obj.apply(number)或obj.call(number);
  - 函数的调用影响this的取值。
    - 如果作为函数调用，在非严格模式下，this指向全局window对象；在严格模式下，this指向undefined。
    - 作为方法调用，this通常指向调用的对象。
    - 作为构造函数调用，this指向新创建的对象。
    - 通过apply或call调用，this指向其第一个参数。
  - 箭头函数没有指定的this值，只有在创建时确定。
  - 所有函数都可以使用bind方法，创建新函数，并绑定到bind方法传入的参数上。被绑定函数与原始函数具有一致的行为。

第五章 精通函数：闭包和作用域

这一章节结合了红宝书和一些文章进行理解。

变量

> ECMAScript 变量可以包含**两种不同类型的数据**：**原始值和引用值。**原始值（primitive value）就是
> 最简单的数据，引用值（reference value）则是由多个值构成的对象。

原始值（6种）：Number、String、Symbol、Boolean、Null、Undefined

引用值（用引用类型创建的值）：Arrary、Function、Date等

- 当我们把一个值赋给变量时，JavaScript引擎必须知道这个值是原始值还是引用值：*引用值就是保存在内存中的对象，JavaScript不允许直接访问内存位置，因此不能直接操作对象所在的内存空间。*在操作对象时，实际操作的是该对象的引用，所以**保存引用值的变量是按引用访问的。**

  > 保存**原始值**的变量是**按值访问**的，保存在**栈内存**中。
  >
  > 保存**引用值**的变量是**按引用访问**的，保存在**堆内存**中。

- 属性值

  原始值和引用值的定义方式很类似，都是创建一个变量，然后给它赋一个值。不过，在变量保存了这个值之后，可以对这个值做什么，则大有不同。

  > 对于引用值来说，可以随时添加、删除，修改其属性和方法。

  ```
  //创建对象，添加属性
  let person = new Object();
  person.name = "shenbeipiao";
  console.log(person.name); // "shenbeipiao";
  ```

  > 原始值不能有属性，尽管尝试给原始值添加属性不会报错。

  ```
  let name = "shenbeipiao";
  name.age = 10;
  console.log(name.age);//undefined
  ```

  > 原始类型的初始化可以只使用原始字面量形式。如果使用的是 new 关键字，则 JavaScript 会创建一个 Object 类型的实例，但其行为类似原始值。

  ```
  let name1 = "shenbeipiao";
  let name2 = new String("Jack");
  name1.age = 27;
  name2.age = 26;
  console.log(name1.age); // undefined
  console.log(name2.age); // 26
  console.log(typeof name1); // string
  console.log(typeof name2); // object
  ```

- 复制值

  > 在通过变量把一个原始值赋值到另一个变量时，原始值会被复制到新变量的位置。

```
let num1 = 5;
let num2 = num1;
//这两个变量可以独立使用，互不干扰。
```

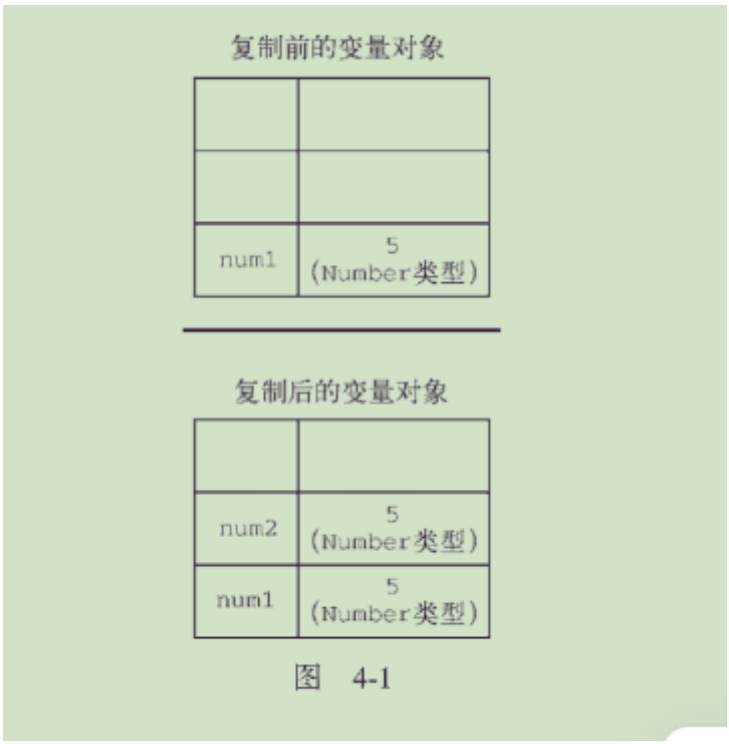
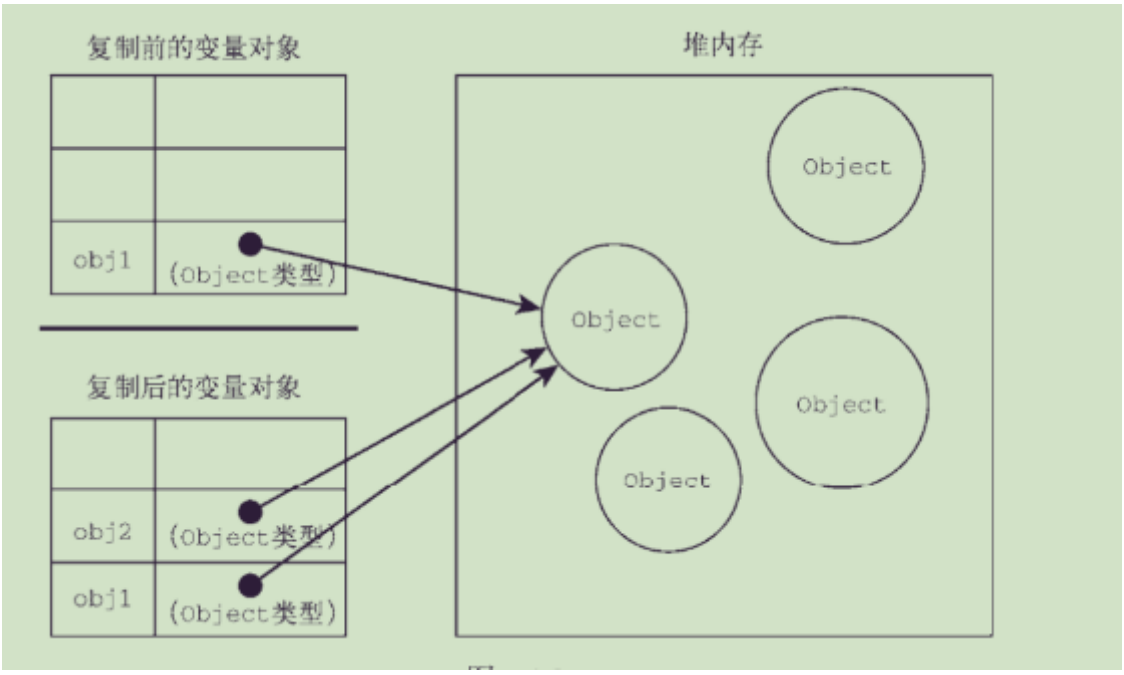


图 4-1

> 在把引用值从一个变量赋给另一个变量时，存储在变量中的值也会被复制到新变量所在的位置。**区别在于，这里复制的值实际上是一个指针，它指向存储在堆内存中的对象。操作完成后，两个变量实际上指向同一个对象，因此一个对象上面的变化会在另一个对象上反映出来。**

```
let Obj1 = new Object();
let Obj2 = Obj1;
Obj1.name = "shenbeipiao";
console.log(Obj2.name);//"shenbeipiao"
```



- 传递参数

> ECMAScript 中**所有函数的参数都是按值传递的。**这意味着函数外的值会被复制到函数内部的参数中，就像从一个变量复制到另一个变量一样。**如果是原始值，那么就跟原始值变量的复制一样，如果是引用值，那么就跟引用值变量的复制一样。**

执行上下文和作用域

> 变量或函数的上下文决定了它们可以访问哪些数据，以及它们的行为。每个上下文都有一个关联的变量对象（variable object），而这个上下文中定义的所有变量和函数都存在于这个对象上。
> 虽然无法通过代码访问变量对象，但后台处理数据会用到它。

- **全局上下文**

  全局上下文指的就是最外层的上下文，它根据宿主环境决定 。在浏览器中，全局上下文就是我们常说的 window 对象，因此所有通过 var 定义的全局变量和函数都会成为 window 对象的属性和方法。

  - 全局上下文在关闭网页或者退出浏览器时销毁
  - 全局上下文会根据不同的宿主环境变化，在浏览器中指的是window对象
  - 使用var定义的全局变量和函数都会出现在window对象上
  - 使用let和const声明的变量与函数不会出现在window对象上
- 函数上下文

  > 每个函数调用都有自己的上下文。当代码执行流进入函数时，函数的上下文被推到一个上下文栈上。
  > 在函数执行完之后，上下文栈会弹出该函数上下文，将控制权返还给之前的执行上下文。ECMAScript
  > 程序的执行流就是通过这个上下文栈进行控制的。

- 作用域链

  上下文中的代码在执行的时候，会创建变量对象的一个作用域链（scope chain）。**这个作用域链决定了各级上下文中的代码在访问变量和函数时的顺序。**

  - **代码正在执行的上下文的变量对象始终位于作用域的最前端。**
  - **如果上下文是函数，其活动对象用作变量对象。**
    - **活动对象最初只有一个默认变量：arguments（全局上下文不存在），作用域链中的下一个变量对象来自包含上下文，再下一个变量对象来自再一个包含上下文。以此类推直至全局上下文。**
  - **全局上下文的变量对象始终是作用域链的最后一个变量对象。**

  作用域链增强：

  > 虽然执行上下文主要有全局上下文和函数上下文两种（eval()调用内部存在第三种上下文），但有其他方式来增强作用域链。**某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在代码执行后会被删除。**

  通常在两种情况下会出现这个现象，即代码执行到下面任意一种情况时：
  □ try/catch 语句的 catch 块
  □ with 语句
  这两种情况下，都会**在作用域链前端添加一个变量对象。**对 with 语句来说，会向作用域链前端添加指定的对象；对 catch 语句而言，则会创建一个新的变量对象，这个变量对象会包含要抛出的错误
  对象的声明。

变量作用域

在JavaScript中声明变量的关键字有：`var`、`let`、`const`，不同关键字声明出来的变量，作用域大不相同。

- 函数作用域

> 使用`var`**声明变量时，变量会被自动添加到最接近的上下文。**在函数中，最接近的上下文就是函数的局部上下文。**如果变量未声明直接初始化，那么它就会自动添加到全局上下文。**

```
function add(num1, num2) {
 var sum = num1 + num2;
 return sum;
}
let result = add(10, 20); // 30
console.log(sum); // 报错：sum 在这里不是有效变量
```

```
function add(num1, num2) {
sum = num1 + num2;
 return sum;
}
let result = add(10, 20); // 30
console.log(sum); // 30
```

> var声明会被拿到函数或全局作用域的顶部，位于作用域中所有代码之前。这个现象叫作“提升”（hoisting）。提升让同一作用域中的代码不必考虑变量是否已经声明就可以直接使用。

```
var name = "Jake";
// 等价于：
name = 'Jake';
var name;
```

- 块级作用域

> 使用`let`关键字声明的变量，会有自己的作用域块，它的作用域是块级的，**块级作用域由最近的一对的花括号`{}`届定。**也就是说，`if`、`while`、`for`、`function`的块内部用let声明的变量，它的作用域都界定在`{}`内部，甚至单独的块，在其内部用let声明变量，它的作用域也是界定在`{}`内部。

```
if (true) {
 let a;
}
console.log(a); // ReferenceError: a 没有定义

while (true) {
 let b;
}
console.log(b); // ReferenceError: b 没有定义

function foo() {
 let c;
}
console.log(c); // ReferenceError: c 没有定义
 // 这没什么可奇怪的
 // var 声明也会导致报错
// 这不是对象字面量，而是一个独立的块
// JavaScript 解释器会根据其中内容识别出它来
{
 let d;
}
```

```
22 console.log(d); // ReferenceError: d 没有定义
```

- 暂时性死区

> ES6规定，let和const会使区块形成封闭的作用域。若在声明前使用变量就会报错。总之，在代码块内使用let声明变量之前该变量都是不可用的。这在语法上称为“暂时性死区”（TDZ）。

```
1 console.log(a); //ReferenceError: Cannot access 'a' before
  initialization
2 let a = 1;
```

- const常量声明

> 使用 const 声明的变量必须同时初始化为某个值。一经声明，在其生命周期的任何时候都不能再重新赋予新值。

```
1 const a; // SyntaxError: 常量声明时没有初始化
2 const b = 3;
3 console.log(b); // 3
4 b = 4; // TypeError: 给常量赋值
```

const 除了要遵循以上规则，其他方面与 let 声明是一样的：

```
1  if (true) {
2   const a = 0;
3  }
4  console.log(a); // ReferenceError: a 没有定义
5
6  while (true) {
7   const b = 1;
8  }
9  console.log(b); // ReferenceError: b 没有定义
10 function foo() {
11  const c = 2;
12 }
13 console.log(c); // ReferenceError: c 没有定义
14 {
15  const d = 3;
16 }
17 console.log(d); // ReferenceError: d 没有定义
```

> const 声明只应用到顶级原语或者对象。换句话说，**赋值为对象的 const 变量不能再被重新赋值为其他引用值，但对象的键则不受限制。**

```
1 const o1 = {};
2 o1 = {}; // TypeError: 给常量赋值
3 const o2 = {};
4 o2.name = 'Jake';
5 console.log(o2.name); // 'Jake'
```

- 变量生存周期
  - 变量如果处在全局上下文中，如果我们不主动销毁，那么它的生存周期则是永久的。
  - 变量如果处在函数上下文中，它会随着函数调用的结束而被销毁。

理解闭包

> 我们知道函数上下文中的变量会随着函数执行结束而销毁，如果我们通过某种方式让函数中的变量不让其随着函数执行结束而销毁，那么这种方式就称之为**闭包** 。 闭包指的是那些引用了另一个函数作用域中变量的函数，通常是在嵌套函数中实现的。

```
var selfAdd = function () {
    var a = 1;
    return function () {
        a++;
        console.log(a);
    }
};

const addFn = selfAdd();
addFn();//2
addFn();//3
addFn();//4
addFn();//5
```

在这个例子中：

- 先声明了一个selfAdd函数，内部定义变量a
- 随后，在函数内部返回了一个匿名函数的引用
- 在这个匿名函数内部，可以访问到selfAdd函数上下文中的变量
- 我们在调用selfAdd函数时，它返回对匿名函数的引用，因为匿名函数在全局上下文中被继续引用，所以它有了不被销毁的理由
- 此时，这里就产生了一个闭包结构，selfAdd函数上下文中的变量生命得到延续

闭包的作用

闭包是通过某种方式让函数中的变量不让其随着函数执行结束而销毁，那么我们可以简单的认为闭包的作用就是延长变量的生命周期。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>学习闭包</title>
</head>
<body>
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<div>5</div>
</body>
<script>
    window.onload = function() {
    const divs = document.getElementsByTagName("div");
    for (var i = 0; i < divs.length; i++) {
        divs[i].onclick = function() {
            alert(i);
```

```
        };
    }
};
</script>
</html>
```

这里面的js部分获取了页面中的所有div标签，循环为每个标签绑定点击事件，由于点击事件是被异步触发的，当事件触发时，for循环早已结束，此时变量`i`的值已经是6，所以在div的点击事件函数中顺着作用域链从内到外查找变量i时，找到的值总是6。 *此处我们可以借助闭包，把每次循环的i值都封闭起来*：

```
window.onload = function() {
    const divs = document.getElementsByTagName("div");
    for (var i = 0; i < divs.length; i++) {
        (function(i) {
            divs[i].onclick = function() {
                alert(i);
            };
        })(i);
    }
};

```

第六章 未来的函数：生成器和promise

什么是生成器

> Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。 它能生成一组值的序列，但每个值的生成是基于每次请求，并不是像标准函数那样立即生成。

```
//Generator 函数
function* sum(x) {
    var y = yield x + 1;
    return y;
}

//简单调用
var i = sum(1);
i.next();//value = 2, done = false;
i.next();//vlaue = 3, done = false;
```

创建一个生成器函数很简单：只需要在关键字function后加上‘ * ’。生成器函数内部可以使用*yield*关键字·，从而生成独立的值。当代码执行到关键字*yield*时，就会生成一个中间结果，然后**返回一个新对象，这个新对象封装了结果值和一个指示完成的指示器。**

> **调用生成器并不会执行生成器函数，相反，它会创建一个叫做迭代器的对象。** 迭代器用于控制生成器的执行，**迭代器对象暴露的最基本的接口是next方法。next方法可以用来向生成器请求一个值**，从而控制生成器。

```

```

```
//通过迭代器对象控制生成器
function* weaponGenerator() {
    yield "x20";
    yield "y39";
}
//定义一个生成器，生成了一个包含两个型号武器数据的序列

//调用生成器,生成迭代器，进而控制生成器的执行
var num = weaponGenerator();

//调用迭代器的next方法向生成器请求一个新值
const num1 = num1.next();
 //断言  done属性的值为false，表明之后还会有值生成
assert(typeof num1 === "object" && num1.value === "y39" && num1.done, "is x20");

const num2 = num1.next();
//断言  done属性的值为true，表明之后不会有值生成
assert(typeof num1 === "object" && num1.value === "y39" && num1.done, "is x20");
```

对迭代器进行迭代

使用while循环进行迭代：

```
function* weaponGenerator() {
    yield "x20";
    yield "y39";
}

const num = weaponGenerator();
//创建变量 保存生成器产生的值
let item;
//当生成器不再生成新值停止迭代
while(!(item = num.next()).done) {
    assert(item !== null, item.value);
}
```

使用yield操作符将执行权交给另一个生成器

```
function* WarriorGenerator(){
    yield "Sun Tzu";
    yield* NinjaGenerator();//yield*将执行权交给了另一个生成器,待其执行完后返回到这一步
    yield "Genghis Khan";
}

function* NinjaGenerator(){
    yield "Hattori";
    yield "Yoshi";
}

for(let warrior of WarriorGenerator()){
    assert(warrior !== null, warrior);
}

```

使用生成器

- 使用生成器生成ID序列

  当我们创建某些对象需要为每个对象赋一个唯一的ID值时，除了直接使用for循环外还·可以使用生成器。

```
function* idGenerator() { //生成器函数
    let id = 0;
    while(true) {
        yield ++id;
    }
}
const id = idGenerator();//向生成器请求新的值
//请求三个id值
const id1 = {id: id.next().value };
const id2 = {id: id.next().value };
const id3 = {id: id.n.value };
```

> 标准函数中一般不应该书写无限循环的代码。但在生成器中没问题！当生成器遇到了一个yield语句，它就会一直挂起执行直到下次调用next方法，所以只有每次调用一次next方法，while循环才会迭代一次并返回下一个ID值。

- 使用生成器遍历DOM树

```
<div id="subTree">
    <form>
        <input tytpe="text"/>
    </form>
    <p>Pragraph</p>
    <span>Span</span>
</div>
<script>
    function* domTraversal(element) {
        yield element;
        element = element.firsElementChild;
        while(element) {
            yield* domTraversal(element); //将其执行权转给另一个函数（和递归效果一样）
            element = element.nextElementSibiling;
        }
    }

    const subTree = document.getElementById(subTree);
    for(let element of domTraversal(subTree)) { //for循环迭代节点
        assert(element !==null, element.Nodename);
    }
</script>
```

> 不同于在下一层递归处理每个访问过的节点子树，我们为每个访问过的节点创建了一个生成器并将执行权交给它，从而使我们能够以迭代的方式书写概念上递归的代码。它的好处在于我们能够不凭借讨厌的回调函数，仅仅以一个简单的for-of循环就能处理生成的节点。

与生成器交互

- 作为生成器函数参数发送值

  向生成器发送值的方法和其他函数一样，调用函数并传入实参

```
function* word(a) {
    const result = yield("hi" + a);//产生新值 通过next方法可以将数据传回生成器
}

const word = word("jack");
const ret = word.next();
```

- 使用next方法向生成器发送值

> next方法为等待中的yield表达式提供了值，所以，如果没有等待中的yield表达式，也就没有什么值能应用的。基于这个原因，我们无法通过第一次调用next方法来向生成器提供该值。 但记住，如果你需要为生成器提供一个初始值，你可以调用生成器自身，就像NinjaGenerator("skulk")。

- 抛出异常

  生成器函数可以手动抛出其他异常。要手动抛出异常，可以在生成器函数中使用 `throw` 语句

```
function* NinjaGenerator() {
  try {
    yield "Hattori";
    fail("The expected exception didn't occur");
  }
  catch(e) {
    assert(e === "Catch this!", "Aha! We caught an exception");
  }
}
const ninjaIterator = NinjaGenerator();
const result1 = ninjaIterator.next();
ninjaIterator.throw("Catch this!");
```

探索生成器内部构成

- 生成器内部状态
  - 挂起开始 --- 创建一个生成器后，它首先以这种状态开始。其中的任何代码都未执行。

- 执行 --- 生成器中的代码已执行。执行要么刚开始了，要么从上次挂起的时候继续的。当生成器对应的迭代调用了next方法，并且当前存在可执行的代码时，生成器都会转到这个状态。
- 挂起让渡 --- 当生成器在执行过程中遇到了一个yield表达式，它会创建一个包含着返回值的新对象，随后再挂起执行。生成器在这个状态暂停并等待继续执行。
- 完成 --- 在生成器执行期间，如果代码执行到return语句或者全部代码执行完毕，那么生成器就进入这个状态。

让我们更进一步补充一些知识，看看生成器是如何跟随执行环境上下文的，如图6.5所示。

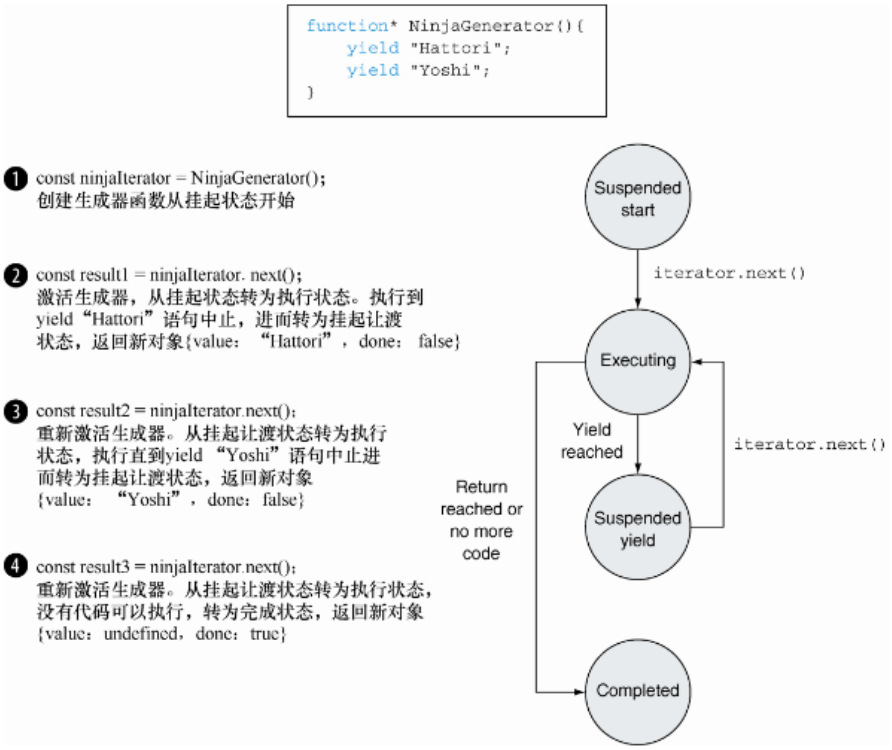


图6.5　在执行过程中，生成器在相对应的生成器调用next函数之间移动状态

使用promise

原理

JavaScript的执行环境是单线程，单线程指的是js引擎中负责解释和执行JavaScript代码的线程只有一个，也就是一次只能完成一项任务，这个任务完成之后才能执行下一个，它会阻塞其他任务。这个任务可以称为主线程。

常见的异步模式有：定时器、接口调用、事件函数。

接口调用的常见方式有：原生Ajax、基于jQuery的Ajax、fetch、**Promise**、axios。

多次异步调用的依赖分析

- 多次异步调用的结果顺序可能不同步
- 异步调用的结果如果存在依赖，则需要嵌套。

在ES5中，当进行**多层嵌套回调**时，会导致代码层次过多，很难进行维护和二次开发，而且会导致**回调地狱**的问题。在ES6中的Promise可以解决这个问题。

- 回调函数带来的问题

  我们使用异步代码的原因在于不希望在执行长时间任务时应用程序的执行被阻塞（影响用户体验）。当我们使用回调函数来解决这个问题时，这个长时间任务可能会发生错误，然后进一步导致其他长期任务的执行，最终导致了互相依赖的一系列异步回调任务错误。

什么是promise

ES6中的Promise是异步编程的一种方案。从语法上讲，**Promise是一个对象，它可以获取异步操作的信息。**Promise对象**可以异步操作以同步的流程表达出来。**

> 使用Promise的好处主要有：可以很好地解决回调地狱的问题（避免了层层嵌套的回调函数）；语法简洁，Promise对象提供了简洁的API，使得控制异步操作更加容易。

promise的基本用法

- 使用new实例化一个Promise对象，Promise的构造函数中传递一个参数，这个参数是一个函数，该函数用于处理异步任务。
- 并且传入两个参数：resolve和reject，分别表示异步执行成功后的回调函数和异步执行失败后的回调函数。
- 通过promise.then()处理返回结果。

```
var promise = new Promise(function(resolve, reject) {
    const x = 1;
    const y = 2;
    if(x === y) {
        reslove();
    } else {
        reject();
    }
});

promise.then(function () {
            console.log('success');})
        .catch(function () {
            console.log('error');
});
```

promise对象的三个状态:

- 初始化状态（pending）：当new Promise()执行后，promise对象的状态会被初始化为pending。

  > new Promise()这行代码，括号里的内容是同步执行的。括号里定义了一个function，它有两个参数：resolve、reject。如果请求成功，则执行resolve()，此时promise的状态会被自动修改为fullfilled；如果请求失败，则执行reject()，此时promise的状态自定修改为rejeced。

- promise.then()方法，括号里有两个参数，分别代表两个函数function1和function2.

  > 如果promise的状态是fullfilled（请求成功），则执行function1里的内容；如果promise的状态是rejected（请求失败），则执行function2里的内容。

promise的常用API：实例方法

- promise.then():获取异步任务的正常结果
- promise.catch():获取异步任务的异常结果
- promise.finaly():异步任务无论成功与否，都会执行

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <script>
            function queryData() {
                return new Promise((resolve, reject) => {
                    setTimeout(function() {
                        var data = {retCode: 0, msg: 'word'};//接口返回数据
                        if(data.ret.Data == 0) {
                            //接口请求成功调用
                            resolve(data);
                        } else {
                            //接口请求失败调用
                            reject({retCode: -1, msg: 'error'});
                        }
                    },100)
                });
            }

            queryDta()
                .then(
                    data => {
                        //resovle获取正常结果
                        console.log('success');
                        console.log(data);
                    })
                .catch(data => {
                   data => {
                        //reject获取异常结果
                        console.log('error');
                        console.log(data);
                    }
                })
                .finally(() =>{
                    console.log('over')
                });
        </script>
    </body>
</html>
```

promise的常用API：对象方法

- Promise.all():并发处理多个异步任务，只有任务都执行成功才能得到结果。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <script type="text/javascript">
            /*
              封装 Promise 接口调用
            */
            function queryData(url) {
                return new Promise((resolve, reject) => {
                    var xhr = new XMLHttpRequest();
                    xhr.onreadystatechange = function(){
                        if(xhr.readyState ! = 4) return;
                        if(xhr.readyState == 4 && xhr.status == 200) {
                            //处理正常结果
                            resolve(xhr.responseText);
                        } else {
                            //处理异常结果
                            reject('error');
                        }
                    };
                    xhr.open('get', url);
                    xhr.send(null);
                });
            }

            var promise1 = queryData('http://localhost:3000/a1');
            var promise2 = queryData('http://localhost:3000/a2');
            var promise3 = queryData('http://localhost:3000/a3');

            Promise.all([promise1, promise2,promise3]).then(result => {
                console.log(result);
            });
         </script>
    </body>
</html>
```

- Promise.race():并发处理多个异步任务，只要有一个任务完成就能得到结果。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <script type="text/javascript">
            /*
```

```
            封装 Promise 接口调用
        */
        function queryData(url) {
            return new Promise((resolve, reject) => {
                var xhr = new XMLHttpRequest();
                xhr.onreadystatechange = function(){
                    if(xhr.readyState ! = 4) return;
                    if(xhr.readyState == 4 && xhr.status == 200) {
                        //处理正常结果
                        resolve(xhr.responseText);
                    } else {
                        //处理异常结果
                        reject('error');
                    }
                };
                xhr.open('get', url);
                xhr.send(null);
            });
        }

        var promise1 = queryData('http://localhost:3000/a1');
        var promise2 = queryData('http://localhost:3000/a2');
        var promise3 = queryData('http://localhost:3000/a3');

        Promise.race([promise1, promise2,promise3]).then(result =>
{
            console.log(result);
        });
    </script>
</body>
</html>
```

面向未来的async函数

> 通过在关键字function之前使用关键字async，可以表明当前的函数依赖一个异步返回的值。在每个调用异步任务的位置上，都要放置一个await关键字，用来告诉JavaScript引擎，请在不阻塞应用执行的情况下在这个位置上等待执行结果。

```
(async function () {
  try {
    const ninjas = await getJSON("data/ninjas.json");
    const missions = await getJSON(missions[0].missionsUrl);

    console.log(missions);
  }
  catch(e){
    console.log("Error: ", e);
  }
})()

```

总结

- 生成器是一种不会在同时输出所有值序列的函数，而是基于每次的请求生成值。

- 不同于标准函数，生成器可以挂起和回复它们的执行状态。当生成器生成了一个值后，它将会在不阻塞主线程的基础上挂起执行，随后静静地等待下次请求。
- 生成器通过在function后面加一个星号（*）来定义。在生成器函数体内，我们可以使用新的关键字yield来生成一个值并挂起生成器的执行。如果我们想让渡到另一个生成器中，可以使用yield操作符。
- 在我们控制生成器的执行过程中，通过使用迭代器的next方法调用一个生成器，它能够创建一个迭代器对象。除此之外，我们还能够通过next函数向生成器中传入值。
- promise是计算结果值的一个占位符，它是对我们最终会得到异步计算结果的一个保证。promise既可以成功也可以失败，一旦设定好了，就不能够有更多改变。
- promise显著地简化了我们处理异步代码的过程。通过使用then方法来生成promise链，我们就能轻易地处理异步时序依赖。并行执行多个异步任务也同样简单：仅使用Promise.all方法即可。通过将生成器和promise相结合我们能够使用同步代码来简化异步任务。

第七章 面向对象与原型

理解原型

在软件开发过程中，为了避免重复造轮子，我们希望可以尽可能地复用代码。继承是代码复用的一种方式，继承有助于合理地组织程序代码，将一个对象的属性扩展到另一个对象上。在JavaScript中，可通过原型实现继承。

> 原型的概念很简单，每个对象上都含有原型的引用，当查找属性时，若对象本身不具有该属性，则会查找原型上是否有该属性。

```
//对象可以通过原型访问其他对象的属性
//创键三个带有属性的对象
const yoshi = { skulk: true};
const hattori = { sneak: true};
const kuma = { creep: true};

//yoshi 只能访问自身的属性
assert("skulk" in yoshi, "Yoshi can skulk");
assert("sneak" in yoshi, "Yoshi can not sneak");
assert("creep" in yoshi, "Yoshi can not creep");

//Object.setPrototypeOf()方法，将一个对象设置为另一个对象的原型
Object.setPrototypeOf(yoshi, hattori);
assert("sneak" in yoshi, "Yoshi can sneak");//可以访问属性
```

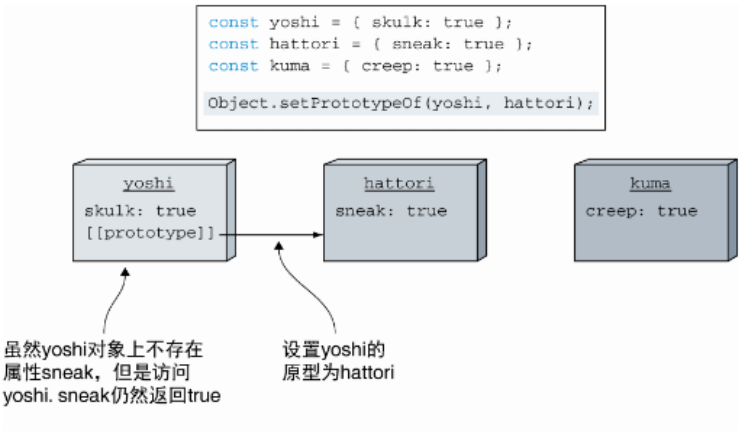


图7.2　当访问对象上不存在的属性时，将查询对象的原型。在这里，我们可以通过对象yoshi访问hattori的属性sneak，因为hattori是yoshi的原型

> 每个对象都可以有一个原型，每个对象的原型也可以拥有一个原型。以此类推，形成了一个原型链，查找特定属性会被委托到整个原型链上。只有当没有更多的原型可以进行查找时，才会停止查找。

对象构造器和原型

> 每个函数都有一个原型对象，该原型对象指向创建对象的函数。

- 通过原型方法创建新的实例

```
//定义一个空函数，无返回值
function Ninja() {};
//每个函数都具有可置的原型对象，我们可以自由更改
Ninja.pototype.swingSword = function() {
    return true;
}

const ninja1 = Ninja();
assert(ninja == undefined, "No");//作为函数调用，验证该函数没有返回值

//作为构造函数调用，验证不仅创建了实例并且该实例还具有原型上的方法
const ninja2 = new Ninja();
assert(ninja2 && ninja2.swingSword && ninja2.swingSword(), "Yes");
```

  - 每个函数都有一个原型对象
  - 每一个函数的原型都具有一个constructor属性，该属性指向函数本身
  - constructor对象的原型设置为新创建的的对象的原型

- 实例属性

  当函数作为构造函数，通过操作符new进行调用时，它的上下文被定义为新的对象实例。通过原型暴露属性，通过构造函数的参数进行初始化。

```
function Ninja() {
    this.swung = false;//创建布尔类型的实例变量，初始化为false
    this.swingSword = function() {
```

```
        return !this.swung; //创建实例方法，返回值为swung取反
    }
}

//定义一个与实例方法同名的原型方法
Ninja.pototype.swingSword = function() {
    return this.swung;
}

const ninja = new Ninja();
assert(ninja.swingSword(), "重写了方法！");//实例会隐藏原型中与实例方法重名的方法
```

> 如果我们需要私有对象，在构造函数内指定方法是唯一的解决方法。

- JavaScript动态特性的副作用

  JavaScript是一门动态语言，可以很容易地添加、删除和修改属性。

  > 对象与函数原型之间的引用是在对象创建时建立的，新创建的对象将引用新的原型对象。

```
//通过原型，一切都可以在运行时修改

//定义构造函数，创建属性num，初始化为布尔值
function test() {
    this.num = true;
}

const test1 = new test();//通过new操作符调用构造函数，创建实例test1

//在实例创建完后，在原型上添加一个方法
test.prototype.numTo = function() {
    return this.num;
}
assert(test.numTo(), "Yes");//方法存在于对象中

//使用字面量对象完全重写test的原型对象，仅有一个pierce方法
test.prototype = {
    pierce: function() {
        return true;
    }
}
assert(test.numTo, "Yes");//仍然保持着对旧的原型的引用

const test2 = new test();//创建实例
assert(test2.pierce(), "Yes");
assert(test2.numTo(), "No");//仅有pierce方法
```

- 通过构造函数实现对象类型

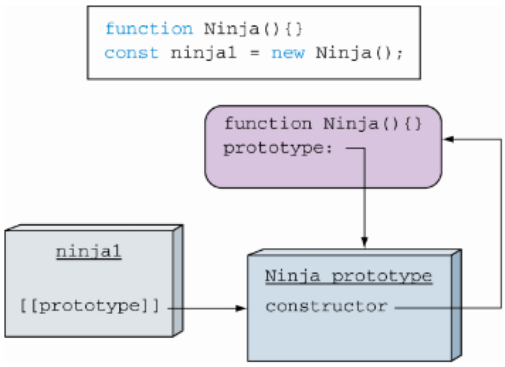


图7.11　每个函数的原型对象都具有一个constructor属性，该属性指向函数本身

- 检查实例的类型与它的constructor

```
function Test() {
    const test = new Test();
    assert(typeof test === "object", "Yes");//检查test类型，是一个对象
    assert(instanceof Test,"Yes");//test是由Test构造而来
    assert(test.constructor === Test, "Yes");//通过constructor引用检测test的类型，得到的结果为其构造函数的引用
}
```

- 使用constructor的引用创建新对象

```
function Test() {};

const test = new Test();
const test1 = new test.constructor();//通过第一个实例化对象的constructor方法创建第二个对象
assert(test1 instanceof Test, "Yes");//说明新创建的对项是Test的实例
assert(test !== test1, "not same");//test和test1不是同一个对象，是两个不同的实例
```

> 虽然对象的constructor属性有可能发生改变，改变constructor属性没有任何直接或明显的建设性目的（可能要考虑极端情况）。constructor属性的存在仅仅是为了说明该对象是从哪儿创建出来的。如果重写了constructor属性，那么原始值就被丢失了。

- 实现继承

  继承是一种在新对象上复用现有对象的属性的形式，这有助于避免重复代码和重复数据。

```
function Person() {};
Person.prototype.dance = function () {};

function Ninja() {};
Ninja.prototype = new Person();// 通过将Ninja的原型赋值为Person的实例，实现Ninja继承Person
```

- 配置对象的属性

  在JavaScript中，对象是通过属性描述（property descriptor）进行描述的，我们可以配置以下关键字。
  configurable —— 如果设为true，则可以修改或删除属性。如果设为false，则不允许修改。
  enumerable —— 如果设为true，则可在for-in循环对象属性时出现（我们很快会介绍for-in循

环）。
value —— 指定属性的值，默认为undefined。
writable —— 如果设为true，则可通过赋值语句修改属性值。
get —— 定义getter函数，当访问属性时发生调用，不能与value与writable同时使用。
set —— 定义setter函数，当对属性赋值时发生调用，也不能与value与writable同时使用。

- instanceof操作符

> 在大部分编程语言中，检测对象是否是类的最直接方法是使用操作符 instanceof。在JavaScript中，操作符instanceof使用在原型链中。

```
ninja instanceof Ninja
//操作符instanceof用于检测Ninja函数是否存在于ninja实例的原型链中。
```

instanceof操作符**检查右边的函数原型是否存在于操作符左边的对象的原型链上。**小心函数的原型可以随时发生改变。

- 在ES6中使用JavaScript的class

ES6引入新的关键字class，它提供了一种更为优雅的创建对象和实现继承的方式，底层仍然是基于原型的实现，它是语法糖。

```
//在ES6中创建类
class Ninja { //使用class创建类
    constructor(name) { //定义一个构造函数，当使用关键字new调用时，会调用这个函数
        this.name = name;
    }

    swingSword() { // 定义一个所有Ninja实例均可访问的方法
        return true;
    }
}
```

  - 静态方法

```
class Ninja {
    constructor(name) {
        this.name = name;
        this.level = level;
    }

    swingSword() {
        return true;
    }

    //使用关键字'static'创建静态方法
    static compare(num1, num2) {
        return num1.level - num2.level;
    }
}
```

  - 在ES6中实现继承

```
class Person {
    constructor(name) {
```

```
        this.name = name;
    }

    dance() {
        return true;
    }
}

//使用关键字extends实现继承
class Ninja extends Person {
    constructor(name, weapon) {
        super(name);//使用关键字super调用基类构造函数
        this.weapon = weapon;
    }

    wieldWeapon() {
        return true;
    }
}

var person = new Person("Bob");
var ninja = new Ninja();
```

总结

- JavaScript对象是属性名与属性值的集合。
- JavaScript使用原型。
- 每个对象上都具有原型的引用，搜索指定的属性时，如果对象本诗不存在该属性则可以代理到原型上进行搜索。对象的原型也可以具有原型，以此类推，形成了原型链。
- 可以通过Object.setPrototype()方法定义对象的原型。
- 原型与构造函数密切相关，每个函数都有原型属性，该函数创建的对象的原型就是函数本身。
- 函数原型对象具有constructor属性，该属性指向函数本身。该函数创建的全部对象均访问该属性，constructor属性还可以用于判断对象是否由指定函数创建。
- 在JavaScript中，几乎所有的内容在运行时都会发生变化，包括对象的原型和函数的原型。
- 如果我们希望一个构造函数的实例可以继承（访问）另一个构造函数的属性，那么将这个构造函数的原型设置为后者类的实例。
- 在JavaScript中，原型具有属性。这些属性可以通过内置的Object.defineProperty()方法进行定义。
- JavaScript ES6中引入关键字class，使得我们可以方便地实现模拟类，但是在底层仍然是使用原型实现的。
- 使用extends可以更优雅地实现继承。

第八章 控制对象的访问

使用getter与setter控制属性访问

> 在JavaScript中，对象是相对简单的属性集合。保持程序状态的主要方法是修改对象的这些属性。

```
function Ninja(level) {
    this.skillLevel = level;
}
const ninja = Ninja(100);
```

这里我们定义了构造函数Ninja，并创建了ninja实例，它仅具有一个属性skillLevel，如果我们想要改变该属性，我们可以通过代码实现： ninja.skillLevel = 10;

虽然这样很方便的实现属性改变，但可能会出现一些问题：

- 我们需要避免意外的错误发生，例如错误赋值。
- 我们需要记录属性的变化。
- 我们需要在网页的UI中显示属性值。我们自然需要显示属性的更新值，但如何轻松的做到?

通过getter和setter方法，可以优雅的实现这一切。

- **使用getter和setter保护私有属性**

```
function Ninja() {
    let SkillLevel; //定义私有变量
    this.getskillLevel = () => skillLevel; //使用getter方法控制对私有变量的访问
    this.setSkillLevel = value => {
        skillLevel = value;
    };
}

const ninja = new Ninja();
ninja.setSkillLevel(100);//通过setter方法为变量赋值
assert(ninja.getSkillLevel() === 100, "Yes");//使用getter方法获取变量值
```

- **定义getter和setter**

  在JavaScript中可以通过两种方式定义getter和setter。

  > - 通过对象字面量定义，或在ES6的class中定义
  > - 通过使用内置的Object.defineProperty方法

  - 在对象字面量中定义getter和setter

    我们通过在属性名前添加关键字get定义getter方法，在属性名前添加关键字set定义setter方法。

```
const ninjaCollection = {
    ninjas: ["Yoshi", "Kuma", "Hattor"],
           //定义一个getter方法，返回列表中的第一个值
           get firstNinja() {
               return this.ninjas[0];
       }
          //定义setter方法，设置列表中的第一个值
           set firstNonja() {
               this.ninjas[0] = value;
           }

}
```

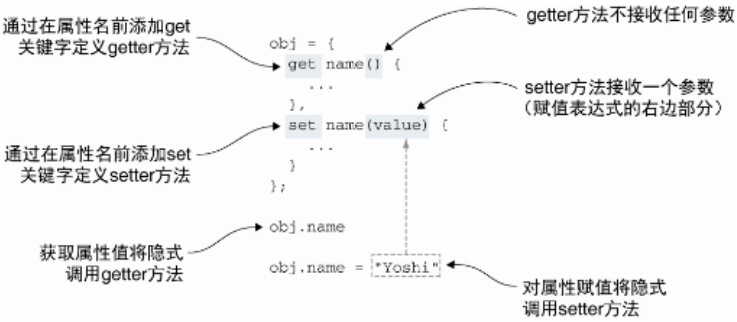


图8.1　定义getter和setter的语法，在属性名之前添加关键字set或get

可以通过原生的getter和setter设置属性，但是这些属性是在访问属性时立即执行的。

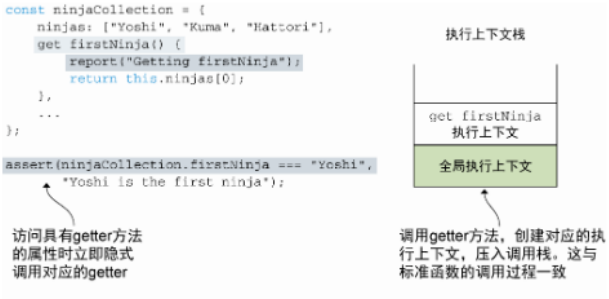


图8.3　访问具有getter方法的属性时隐式调用对应的getter。这个过程看起来与标准方法调用一致，执行了getter方法。通过setter对属性赋值的过程也类似

- **在ES6的class中使用getter和setter**

```
class NinjaCollection {
    constructor() {
        this.ninjas = ["Yoshi", "Kuma", "Hattor"];
    }
    get firstNinja() {
        return this.ninjas[0];
    }
    set firstNinja(value) {
        this.ninjas[0] = value;
    }
}
```

- **通过Object.defineProperty定义getter和setter**

传统上，getter和setter方法用于控制访问私有对象属性，但是JavaScript中并没有私有对象属性，我们可以通过闭包模拟私有对象属性，通过定义变量和指定对象包含这些变量。由于对象字面量和类、getter、setter方法不是在同一个·作用域中定义的，因此那些希望作为私有对象属性的变量是无法实现的。但我们可以通过Object.defineProperty方法实现。

```
function Ninja() { //定义构造函数
    let _skillLevel = 0; //定义私有变量，通过闭包访问该变量
    Object.defineProperty(this,'skillLevel', {
        //使用内置的Object.defineProperty定义属性skillLevel
        get: () => { //访问skillLevel属性时调用get方法
            return _skillLevel;
        }
```

```
        set: value => { //对属性skillLevel赋值时调用set方法
            _skillLevel = value;
        }
    })
}

const ninja = new Ninja(); //创建新的`Ninja`实例

assert(typeof ninja._skillLevel === "undefined",
        "We cannot access a 'private' property");
assert(ninja.skillLevel === 0, "The getter works fine!"); //无法直接访问私有变量，但可以通过`getter`访问

ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");  //对属性`skillLevel`属性赋值时隐式调用`set`方法

```

> 与对象字面量不同和类中的getter和setter不同，通过Object.defineProperty创建的getter和setter方法，与私有skillLevel变量处于相同的作用域中。get和set方法分别创建了含有私有变量的闭包，我们只能通过get和set方法访问私有变量。

- **使用getter和setter校验属性值**

  当对属性赋值时，会立即调用setter方法。我们可以利用这一特性，在代码试图更新属性值时实现一些行为。

```
//通过setter校验属性值
function Ninja() {
    let _skillLevel = 0;
    Object.defineProperty(this, 'skillLevel', {
        get: () => _skollLevel,
        set: value => {
            //校验传入的值是否是整型，不是则抛出异常
            if(!Number.isInteger(value)) {
                throw new TypeError("Skill level should be a number");
            }
            _skillLevel = value;
        }
    });
}

//将整型值赋值变量
const ninja = new Ninja();
ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");

//试图将非整型值赋值给变量，抛出异常
try {
    ninja.skillLevel = "Great";
    alert("Error");
} catch {
    alert("Yes");
}
```

- 使用getter和setter定义如何计算属性值

  除了能够控制指定对象属性的访问外，我们还可以用来定义属性值的计算方法，即每次访问该属性值时都会进行计算属性值。计算属性值不会存储具体的值，它们提供get和set方法，用于直接提取、设置属性。

```
//定义如何计算属性
const shogun = {
    name: "abc",
    clan: "def",
    get fullTitle() { //在对象字面量上定义属性fullTitle的getter方法，该方法将name与clan两个属性值拼接在一起
        return this.name + " " + this.clan;
    },
    set fullTitle(value) { //在对象字面量上定义属性fullTitle的setter方法，该方法将传入的参数通过空格分开，并分别更新标准属性name与clan的值
        const segments = value.split(" ");
        this.name = segments[0];
        this.clan = segments[1];
    }
};



assert(shogun.name === "Yoshiaki", "Our shogun Yoshiaki");
assert(shogun.clan === "Ashikaga", "Of clan Ashikaga");
assert(shogun.fullTitle === "Yoshiaki Ashikaga",
       "The full name is now Yoshiaki Ashikaga");
//name与clan属性均是普通属性，具有直接属性值。而访问fullTitle属性的值时将调用对应的get方法计算属性值

shogun.fullTitle = "Ieyasu Tokugawa";
assert(shogun.name === "Ieyasu", "Our shogun Ieyasu");
assert(shogun.clan === "Tokugawa", "Of clan Tokugawa");
assert(shogun.fullTitle === "Ieyasu Tokugawa",
       "The full name is now Ieyasu Tokugawa");
//对fullTitle属性赋值时将调用对应的set方法，该方法将计算后分别赋值给name与clan属性
```

使用代理控制访问

getter和setter方法对于JavaScript语言是非常有用的，可以处理日志记录，数据验证，属性值变化测验等。但getter和setter方法有时是不够用的，在这种特殊情况下我们使用一种全新的对象类型：代理。

> 代理（proxy）是我们通过代理控制对另一个对象的访问。通过代理可以定义当对象发生交互时可执行的自定义行为---如读取或设置属性值，或调用方法。可以将代理理解为通用化的getter和setter，区别是每个setter与getter仅能控制单个对象的属性，而代理可以用于对象交互的通用处理，包括调用对象的方法。

- **通过Proxy构造器创建代理**

```
const emperor = {name: "abc"};//emperor是目标对象

```

```
//通过Proxy构造器创建代理，传入对象emperor，以及包含get和set方法的对象，用于处理对象属性的读写操作
const representative = new Proxy(emperor, {
    get:(target, key) => {
        return key in target ? target[key] : "Not"
    },
    set: (target, key, value) => {
        target[key] = value;
    }
});

//分别通过目标对象和代理对象访问name属性
assert(emperor.name === "abc", "Yes");
assert(representative.name === "abc","Yes");

//访问不存在的属性将返回undefined
assert(emperor.nickname === undefined, "Not");
//通过代理访问时，检测到属性不存在，返回警告
assert(representative.nickname === "not have", "Not");

//通过代理给对象添加属性，  通过目标对象和代理对象均可访问属性
representative.nickname = "def";
assert(emperor.nickname === "adef", "Yes");
assert(representative.name === "def","Yes");
```

- **使用代理记录日志**

  代理的直接用途是在我们读写属性时使用一种更好、更清洁的方式启用日志记录。

```
function makeLoggable(target) {//定义形参为target的函数，并使得target可以记录日志
    return new Proxy(target, { //针对target对象创建代理
        get: (target, property) => { //通过get方法实现属性读取时记录日志
            return target[property];
        },
        set: (target, property,value) => {//通过set方法实现属性赋值时记录日志
            target[property] = value;
        }
    });
}

//创建新的对象并作为目标对象传入makeLoggable方法，使其可以记录日志
let ninja = {name: "abc"};
ninja = makeLoggable(ninja);
assert(ninja.name === "abc", "Yes");
ninja.weapon = "sword";//对代理对象进行读写操作时，均会通过代理方法记录日志
```

> 这种日志记录方式比使用标准的getter和setter方法更容易、更透明。我们不会把原有代码与日志代码混淆，也不需要为每个对象添加单独的日志。所有的读写属性操作都会进入代理方法。记录日志的代码只需要在一处指定，无论读写属性多少次、无论属性增加多少，都可以记录对应的日志。

- **使用代理检测性能**

```
```

```
function isPrime(number) {
    if(number < 12) {return false;}

    for(let i = 2; i < number; i++) {
        if(number % i === 0) {return false;}
    }
    return true;
}

isPrime = new Proxy(isPrime, { //使用代理包装isPrime方法
    apply: (target, thisArg,args) => { //定义apply方法，当处理对象作为函数被调用时将会触发该apply方法的执行
        console.time("isPrime");//启动计时器，记录isPrime函数执行的起始时间
        const result = target.apply(this.Arg, args); //调用目标函数
        console.timeEnd("isPrime"); //停止计时器的执行并输出结果
        return result;
    }
});

isPrime(1003);
```

> 我们将新创建的代理对象赋值给isPrime标识符，这样，我们无需修改isPrime函数内部代码就可以调用apply方法实现isPrime函数的性能评估。

- **使用代理自动填充属性**

```
function Floder() {
    return new Proxy({}, {
        get: (target, property) => {
            if(!(property in target)) { //如果对象不具有该属性则创建该属性
                target[property] = new Floder();
            }
            return target[property];
        }
    });
}

const rootFloder = new Floder();
try {
  rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";//每当访问属性时，都会执行代理方法，若该属性不存在，则创建该属性

}
catch(e){
   alert("error");
}
```

> 因为使用了代理，所以每次访问属性时代理方法都会被激活。如果访问的属性在文件夹对象存在则直接返回对应的值，不存在则创建新的文件夹并赋值给该属性。

- **使用代理实现负数组索引**

JavaScript中不支持数组负索引，但我们可以使用代理进行模拟。

```
function creatNegativeArrayProxy(array) {
    if(!Array.isArray(array)) {
        throw new TypeError("error"); //传入的参数不是数组则抛出异常
```

```
    }

    return new Proxy(array, { //返回新的代理，该代理使用传入的数组作为代理目标
        get: (target, index) => { //当读取数组元素时调用get方法
            index = +index; //使用一元+操作符将属性名变成数值
            //如果访问的是负向索引则逆向访问数组。是正向索引则正常访问数组
            return target[index < 0 ? target.length + index : index];
        },
        set: (target, index, val) => { //当写入数组元素时调用set方法
            index = +index;
            return target[index < 0 ? target.length + index : index =
val];
        }
    });
}

const ninjas = ["yoshi", "kuma", "hattori"]; //创建标准数组
const proxiedNinjas = creatNegativeArrayProxy(ninjas);//创建代理
assert(proxiedNinjas[-1] === "fattori", "Yes"); //负索引访问
proxiedNinjas[-1] = "hachi";//通过代理，使用负索引设置数组元素
assert(proxiedNinjas[-1] === "hachi", "Yes");
```

- **检查代理的性能限制**

  代理是我们通过代理对象控制对另一个对象的访问。代理可以定义执行特定操作时同时调用的方法。它能够实现许多有用的功能，如日志记录、性能评估等。但它并不是完美的，依然有缺点，我们通过代理添加了一个间接层使我们能够实现这些功能，但同样也引入了大量的额外处理，会影响性能，要谨慎使用代理。

```
//借用上个例子

function measure(otems) {
    const startTime = new Date().getTime();//在执行循环体前获取当前时间
    for(let i = 0; i < 500000; i++) { //在长时间执行的循环中访问集合中的元素
        items[0] = "yoshi";
        itens[1] = "kuma";
        items[2] = "hattori";
    }
    return new Date().getTime() - startTime;//计算时间差
}

const ninjas = ["yoshi", "kuma", "hattori"];
const proxiedNinjas = creatNegativeArrayProxy(ninjas);//创建代理
//比较二者时间差


console.log("Proxies are around",
            Math.round(measure(proxiedNinjas)/ measure(ninjas)),
            "times slower");

```

总结

- 我们可以使用getter、setter和代理监控对象。

- 通过访问器方法（getter和setter），我们可以对对象属性的访问进行控制。
  - 可以通过内置的Object.defineProperty方法定义访问属性，或在对象字面量中使用get和set语法或在ES6的class。
  - 当读取对象属性时会隐式调用get方法，当写入对象属性时隐式调用set方法。
  - 使用getter方法可以定义计算属性，在每次读取对象属性时计算属性值；同理，setter方法可以用于实现数据校验与日志记录。
- 代理是JavaScript ES6引入的，可用于控制对象。
  - 代理可以控制对象交互时的行为（例如，当读取属性或调用方法时）。
  - 所有的交互行为都要通过代理，指定的行为发生时会调用代理方法。
- 使用代理可以优雅地实现这些功能。
  - 日志记录
  - 性能测量
  - 数据校验
  - 自动填充对象属性（以此避免null异常）
  - 数组负索引
- 代理效率不高，所以在需要执行多次的代码中需要谨慎使用，建议进行性能测试。

第九章 处理集合

数组

- 创建数组

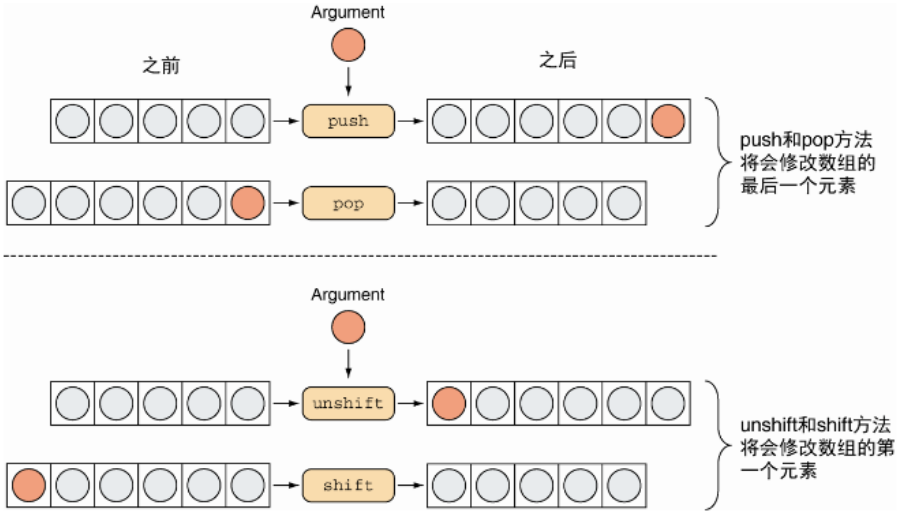
  创建数组有两种基本方式

  > 1.使用内置的Array构造函数
  >
  > 2.使用数组字面量[]

```
//使用数组字面量[]创建数组
const ninja = ["kuma", "hattori", "yagu"];
//使用内置的Array构造函数创建函数
const samura = new Array("oda", "tome");

//数组长度  length属性
assert(ninja.length === 3, "Yes");
assert(samura.length === 2, "Yes");

//通过索引访问数组元素
assert(ninja[0] === "kuma", "Yes");
assert(samura[samura.length - 1] === "tome", "Yes");

//对超出数组边界的索引写入元素将扩充数组
ninja[4] = "abc";
assert(ninja.length === 5, "Yes"); //ninja[3] == undefined

//手动修改数组的length属性为更小值将会删除多余的元素
ninja.length = 2;
assert(ninja[2] === undefined, "no yagu");
```

- 在数组两端添加、删除元素
  - push: 在数组末尾添加元素
  - unshift: 在数组开头添加元素
  - pop: 从数组末尾删除元素
  - shift: 从数组开头删除元素

```
//创建空数组
const ninja = [];

//在数组末尾添加3个元素
ninja.push("num1");
ninja.push("num2");
ninja.push("num3");

//在数组开头添加1个元素
ninja.unshift("num4");

//从数组末尾删除元素
const firstArr = ninja.pop();
//从数组开头删除元素 其他元素自动左移
const secondArr = ninja.shift();
```



> 性能考虑:
>
> pop和push方法只影响数组的最后一个元素 --- pop移除最后一个元素；push在数组末尾添加一个元素。
>
> shift和unshift方法修改第一个元素，之后的每一个元素的索引都发生改变。
>
> 因此，pop和push方法比shift和unshift方法要快很多，非特殊情况不建议使用shift和unshift方法。

- 在数组任意位置添加、删除元素

```
const ninja = ["num1", "num2", "num3", "num4"];
var newArr = ninja.splice(1, 1); //使用内置的splice方法从索引1开始，删除一个元素

//splice方法将返回被删除的元素数组
assert(newArr.length === 1, "Yes");
assert(newArr[0] === "num1", "Yes");

//使用splice方法并添加参数可以在指定位置添加元素
newArr = ninja.splice(1, 2, "num5", "num6", "num7");
```

- **数组常用操作**
  - 遍历数组

```
//使用forEach方法
const ninja = ["num1", "num2", "num3"];
ninja.forEach(ninja => {
    assert(ninja !== null, ninja);
});
```

  - 映射数组

> 基于已有数组的元素创建数组是非常常见的，因此它具有一个特殊的名称：映射数组。
>
> 主要思想是将数组中的每个元素的属性映射到新数组的元素上。

```
const ninja = [
    {name:"tom", weapon: "x1"},
    {name: "jack", weapon: "x2"},
    {name: "sam", weapon: "x3"}
];

//内置的map方法接收回调函数作为参数，并对数组的美国元素执行该函数
const weapons = ninja.map(ninja => ninja.weapon);
assert(weapons[0] === "x1" && weapons[1] === "x2" && weapons[1] === "x3", "Yes" );
```

内置的map方法创建了一个全新的数组，然后遍历输入的数组。对输入数组的每个元素在新建的数组上都会基于回调函数的执行结果创建一个对应的元素。

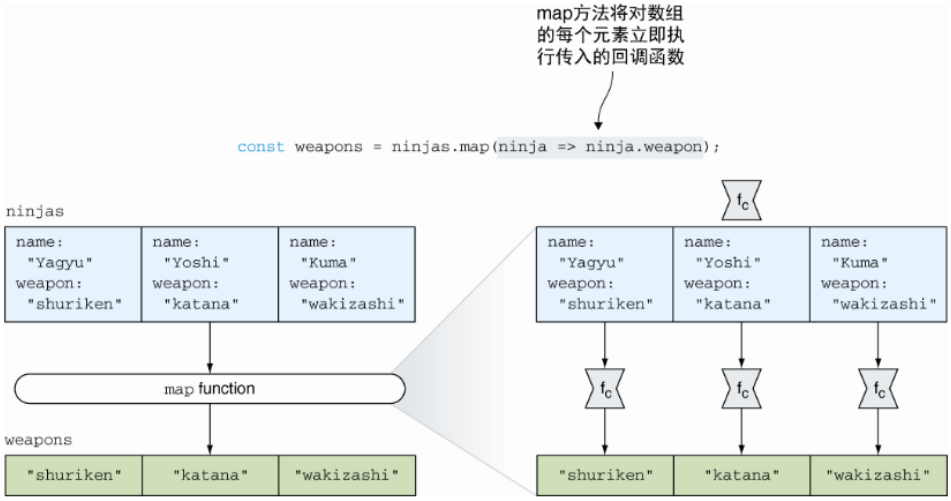


图9.5　map函数对数组的每个元素执行回调函数，使用返回值创建新数组

- 测试数组

> 处理集合的元素时，常常遇到需要知道数组的全部元素或部分元素是否满足某些条件。为了尽可能有效的编写这段代码，JavaScript数组具有内置的every和some方法。

```
//使用every和some方法测试数组
const ninja = [
    {name:"tom", weapon: "x1"},
    {name: "jack", weapon: "x2"},
    {name: "sam", weapon: "x3"}
];

const allNinjaName = ninja.every(ninja => "name" in ninja);
const allNinjaWea = ninja.some(ninja => "weapon" in ninja);
```

此例中every方法接收回调函数，对集合中的每个ninja对象检查是否含有name属性。当且仅当全部的回调函数都返回true时every方法才返回true；some方法从第一项开始执行回调函数，直到回调函数返回true；如果有一项元素执行回调函数时返回了true，some方法返回true，否则some方法返回false。

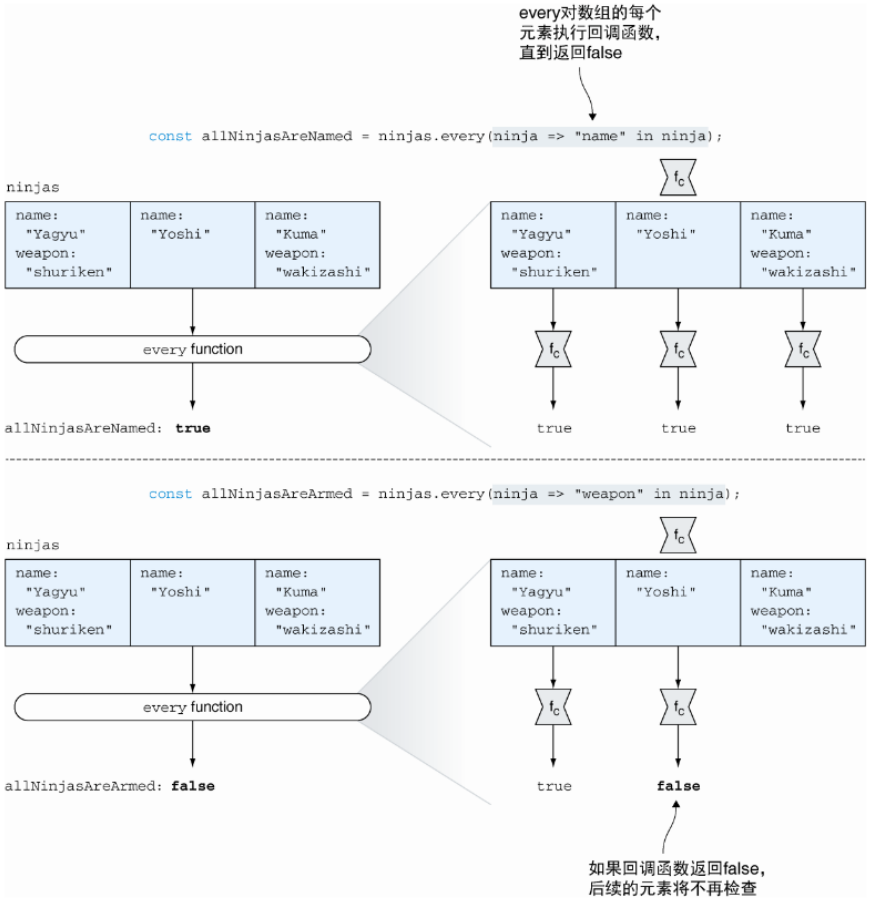


图9.6　every方法通过回调函数测试数组中的所有元素是否满足某个条件

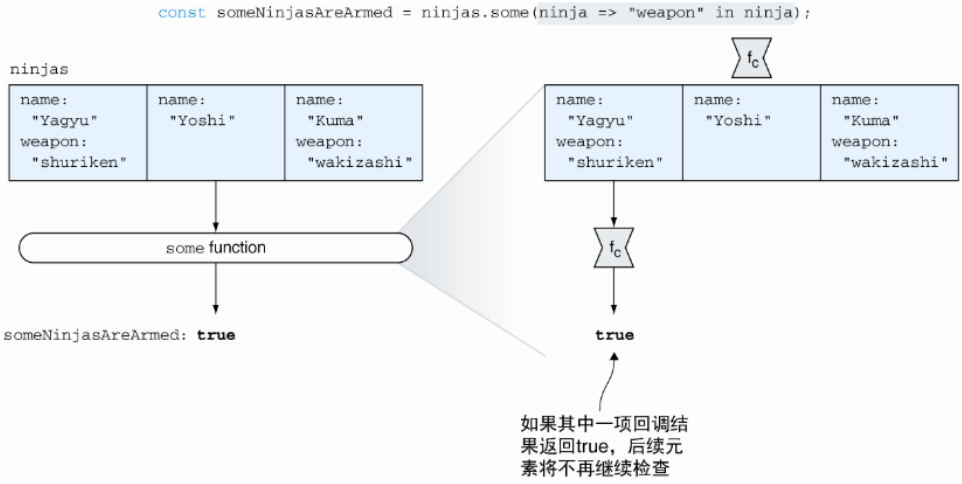


图9.7　some通过回调函数检查数组中是否至少有一项满足回调函数中指定的条件

- 数组查找
  - 查找数组元素

```
const ninja = [
    {name:"tom", weapon: "x1"},
    {name: "jack"},
    {name: "sam", weapon: "x3"}
];

//使用find方法查找满足回调函数中指定条件的第一个元素,未找到则返回
undefined
const ninjaWithX3 = ninja.find(ninja => {
    return ninja.weapon === "x3";
});

//使用filter方法查找满足条件的多个元素
const armedNinja = ninja.filter(ninja => "weapon" in ninja);
```

- 查找数组索引

```
const ninja = ["num1", "num2", "num3", "num1"];

//使用findIndex查找索引, 未找到则返回-1
const isFind = ninja.findIndex(ninja => ninja === "num8");
//使用indexOf方法查找特定元素, 返回第一次出现的索引
const first = ninja.indexOf("num2"); //1
//使用lastIndexOf方法查找某个特定元素最后一次出现的索引
const last = ninja.lastIndexOf("num1");//3
```

- 数组排序

```
const ninja = [{name: "num1"}, {name: "num2"}, {name: "num3"}];
//向内置的sort方法传入回调函数。指定排序顺序
ninja.sort(function(ninja1, ninja2) {
    if(ninja1.name < ninja2.name) {
        return -1;
    }
    if(ninja1.name > ninja2.name) {
        return 1;
    }
});
```

JavaScript具有sort方法:

```
array.sort((a, b) => a - b);
/*
    返回值 a - b < 0 ,升序排序
          a - b > 0 , 降序排序
          a - b = 0 , 两元素处于相同位置（a 和 b 的相对位置不变）
*/
```

- 合计数组元素

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((aggregates, number) => aggregates +
number, 0);
```

reduce方法接收初始值，对数组每个元素执行回调函数，回调函数接收上一次回调结果以及当前数组元素作为参数。

Map

- 创建map

```
//使用构造函数创建map
const ninja = new Map();
//定义三个对象
const ninja1 = {name: "tom"};
const ninja2 = {name: "jack"};
const ninja3 = {name: "sam"};

//使用Map的set方法，建立两个ninja对象的映射关系
ninja.set(ninja1, {homeIs: "honshu"});
ninja.set(ninja2, {homeIs: "kokkaido"});

//使用Map的get方法，获取ninja对象
assert(ninja.get(ninja1).homeIs === "honshu");

//使用has方法验证map中是否存在指定的key
assert(ninja.has(ninja3), "No");

//使用delete方法删除key
ninjaIs.delete(ninja1);
assert(!ninjaIs,has(ninja1) && ninjaIs.size() === 1, "yes");

//使用clear完全清空map
ninjaIs.clear();
assert(ninjaIs.size() === 0, "yes");
```

map是键值对的集合，key可以是任意类型的值，甚至可以是对象。

处理map时的一个基本概念就是确定两个映射的key是否相等。

> 两个不同的对象创建不同的映射。
>
> 在JavaScript中，我们不能重载相等运算符，虽然两个对象的内容相同，但是两个对象仍然不相等。

```
const map = new Map();
//使用内置的location获取当前页面的url
const currentLocation = location.herf;
//创建两个当前页面的链接
const firstLink = new URL(currentLocation);
const secondtLink = new URL(currentLocation);

//分别为两个链接添加映射
map.set(firstLink, {description: "firstLink"});
map.set(secondLink, {description: "secondLink"});

//尽管每个链接指向同一个页面，但仍然具有各自的映射
assert(map.get(firstLink).description === "firstLink", "yes");
assert(map.get(secondLink).description === "secondLink", "yes");
```

- 遍历map

```
const directory = new Map();
//在每个对象中存储电话号码
directory.set("yoshi", "+86 12345");
directory.set("kuma", "+86 67890");

//使用for of 循环遍历， 每个元素具有两个值：key value
//两个方法：keys  values
for(let key of directory.keys()) {
    assert(key !== null, "key" + key);
    assert(directory.get(key) != null, "value" + value);
}
for(var value of directory.values()) {
    assert(value !== null, "value" + value);
}
```

Set

在实际问题中，我们必须处理一种集合，集合中的每个元素都是唯一的（每个元素出现一次），这种集合称为Set。

- 创建Set

```
//set构造函数接收数组进行初始化
const ninja = new Set(["kuma"], ["hattori"], ["yagu"], ["hattori"]);
//丢弃重复项
assert(ninja.has("hattori"), "yes");
assert(ninja.size === 3, "只3个元素");
//添加元素  添加已有元素将不起任何作用
ninja.add("yoshi");
assert(ninja.size === 4, "yes");
//遍历集合
for( let num of ninja) {
    assert(ninja !== null, ninja);
}
```

- 并集

  两个集合的并集指的是创建一个新的集合，同时包含A和B中的所有成员。

```
//创建两个集合，hattori在两个数组中都存在
const ninja = ["kuma", "hattori", "yagu"];
const samurai = ["hattori", "tom"];
//创建两个数组的并集
const warriors = new Set([...ninja], [...samurai]);
```

- 交集

  两个集合的交集指的是创建新集合，该集合中只包含集合A和B中同时出现的元素。

```
const ninja = new Set(["kuma", "hattori", "yagu"]);
const samurai = new Set(["hattori", "tom"]);

//使用延展运算符将集合转换为数组，以便调用数组的filter方法，保留同时存在的元素
const ninjaSam = new Set([...ninja].filter(ninja =>
samurai.has(ninja)));
```

- 差集

  两个集合的差集指的是创建新集合，只包含在与集合A、不包含于集合B中的元素。

```
const ninja = new Set(["kuma", "hattori", "yagu"]);
const samurai = new Set(["hattori", "tom"]);

const newset = new Set([...ninja].filter(ninja => !samurai.has(ninja)));
```

总结

- 数组特殊的对象，具有length属性，原型是Array.prototype。
- 可以使用数组字面量（[]）或者Array构造函数创建数组。
- 通过使用数组对象的方法可以修改数组的内容。
  - push: 在数组末尾添加元素
  - unshift: 在数组开头添加元素
  - pop: 从数组末尾删除元素
  - shift: 从数组开头删除元素
- 数组可以访问很多有用的方法。
  - map方法可以对数组成员调用回调函数，并使用调用结果创建新数组。
  - every和some方法检测全部或部分元素是否匹配某些条件。
  - find和filter方法查找满足某些条件的元素。
  - sort方法对数组排序。
  - reduce方法将数组元素合计为一个值。
- 可以在自定义对象上，显示定义对象方法，使用call或apply方法对数组方法进行复用。
- Map和字典是包含key和value映射关系的对象。
- JavaScript中的对象是糟糕的map，只能使用字符串类型作为key，并且存在访问原型属性的风险。因此，使用内置的Map集合。
- 可以使用for of循环遍历Map集合。
- Set成员的值都是唯一的。

第十章 正则表达式

在JavaScript中，创建正则表达式有两种方式。

- 使用正则表达式字面量。
- 通过创建RegExp对象的实例。

```
//字面量
const reg = /test/
//实例
const reg1 = new RegExp("test");
```

> 当正则表达式在开发环境中是明确的，推荐优先使用字面量语法；当需要在运行时动态创建字符串来构建正则表达式时，则使用构造函数的方式。

- **术语和操作符**
  - 精确匹配

    除了非特殊字符和操作符之外，字符必须准确出现在表达式中。例如正则/test/中的4个字符，必须完全出现在所匹配的字符串中。
  - 匹配字符集

    如果我们需要匹配一组有限的字符集中的字符，我们可以将我们希望匹配的字符集放在**[]**中，指定字符集操作符：[abc]。如果我们需要匹配一组有限的字符集以外的任意字符，则可以使用^: [abc]。限定范围符：[a-m].

    ```
    [abc]//表示匹配a、b、c中的任意一个字符
    [^abc]//表示匹配除了a、b、c以外的任意字符
    [a-m]// -表示按字母顺序从a到m之间的所有字符的集合
    ```
  - 转义

    在正则表达式中，反斜线对其后面的字符进行转义，使其匹配字符本身的含义。

    ```
    const test = /\[/; //匹配[
    ```
  - 起止符号

    起始符号：^ 。 结束符号：$。

    ```
    const num = /^test/; //匹配的是test出现在字符串的开头
    const num1 = /test$/; //$表示字符串的结束
    const num2 = /^test$/; //同时使用^和$表示匹配整个字符串
    ```
  - 重复出现

    正则表达式提供了几种用于指定重复选项的方式：
    - **?** ：指定可选字符（出现0次或1次），在字符后添加?

      ```
      /t?est/   //可以同时匹配test与est
      ```
    - **+**：指定字符出现1次或多次，使用+

      ```
      /t+est/  //可匹配 test、ttest、tttest等
      ```
    - ***** ：指定字符出现0次或1次或多次，使用*

```
/t*est/  //可匹配 test、ttest、tttest等以及est
```

- **{x}** ：指定重复次数，使用{x}，x表示重复的次数

```
/a{4}/   //匹配四个连续的字符a
```

- **{x, }** ：指定开放区间，省略第二个值，保留逗号

```
/a{4,}/  //匹配4个或更多连续的字符a
```

这些运算符都可以是贪婪或非贪婪的，默认是贪婪模式，可以匹配所有可能的字符。在运算符后添加？可以变为非贪婪模式，只进行最小限度的匹配。

```
//字符串 aaa
/a+/  /*会匹配全部字符*/
/a+?/  /*匹配一个字符a*/
```

- 预定义字符集

预定义元字符	匹配的字符集
/t	水平制表符
\b	空格
\r	回车符
\f	换页符
\h	换行符
\cA:\cZ	控制字符
\u0000:\uFFFF	十六进制Unicode码
\x00:\xFF	十六进制ASCII码
.	匹配除换行符（\n、\r、\u2028和\u2029）之外的任意字符
\d	匹配任意十进制数字，等价于[0-9]
\D	匹配除了十进制数字外的任意字符，等价于[^ 0-9]
\w	匹配除了字母、数字和下划线之外的字符，等价于[^A-Za-z0-9]
\s	匹配除空白字符外的任意字符
\b	匹配单词边界
\B	匹配非单词边界（单词内部）

- 分组

如果对一组术语使用操作符，可以使用圆括号进行分组。

```
/(ab)+/  //匹配一个或多个连续的ab
```

- 或操作符

```
/(ab)+|(cd)+/  //匹配一个或多个ab或cd
```

- 修饰符

修饰符	含义
i	ignore，不区分大小写
g	global, 全局匹配
m	multi-line，多行匹配
s	特殊字符圆点.中包含换行符\n
u	支持Unicode转义
y	支持粘连匹配

- 总结
  - 创建正则表达式可以使用正则表达式字面量（/test/）或者正则构造函数（new RegExp("test")）。对于开发环境明确的推荐使用正则字面量，在运行时则推荐使用构造函数。
  - 每个正则都可以使用5个标识符：i---不区分大小写　g---全局匹配　m---支持多行匹配 y---支持粘连匹配 u---支持Unicode转义。在正则后面添加标志位如 /test/ig，或者作为构造函数的第二个参数传入，如new RegExp("test", "i")。
  - 使用section指定一组待匹配的字符。
  - 使用^表示匹配字符串的起始位置，$表示字符串的结束位置。
  - 使用？表示可选项，+ 表示出现1次或多次·，*表示出现0次、1次或多次。
  - 使用.匹配任何字符
  - 使用反斜线()转义特殊字符
  - 使用圆括号()对多个术语分组，使用|表示或
  - 通过反斜线+数字如（\1,\2）可以对匹配的字符串进行反向引用
  - 每个字符串可以使用match函数，match函数的传入参数是正则表达式，返回值是匹配到的全部字符串以及全部捕获。使用replace函数，可以对固定字符串进行替换

第十一章 代码模块化

总结

- 小的、组织良好的代码远比庞大的代码更容易理解和维护。优化程序结构和组织方式的一种方式是将代码拆分成小的、耦合相对松散的片段或模块。
- 模块是比对象或函数稍大的、用于组织代码的单元，通过模块可以将程序进行分类。
- 通常来说，模块·可以降低理解成本，模块易于维护，并可以提高代码的可重用性。

- 在JavaScript ES6以前没有内置的模块，开发者们不得不创造性地发挥JavaScript语言现有的特性实现模块化。最流行的方式之一是通过立即执行函数的闭包实现模块。
    - 使用立即执行函数创建定义模块变量的闭包，从外部作用域无法访问这些变量。
    - 使用闭包可以使模块变量保持活跃。
    - 最流行的是模块模式，通常采用立即执行函数，并返回一个新的对象作为 模块的公共接口。
- 除了模块模式，还有两个流行的的模块标准：AMD, 可以在浏览器端使用； CommonJS，在JavaScript服务器端更流行。
    - AMD可以自动解决依赖，异步加载模块，避免阻塞。
    - CommonJS语法简单，可以同步加载模块（因此在服务器端更流行），通过npm可以获取大量模块。
- ES6结合了AMD和CommonJS的特点。ES6模块受CommonJS的影响，语法简单，并提供了与AMD类似的异步模块加载机制。
    - ES6模块基于文件，一个文件是一个模块。
    - 通过关键字export导出标识符，在其他模块中可引用这些标识符。
    - 在其他模块中通过关键字import导入标识符。
    - 模块可以使用默认导出，通过一个export导出整个模块。
    - export与import都可以通过关键字as使用别名。

在ES6之前的版本中模块化代码

js在ES6之前只有两种作用域：全局作用域和函数作用域。没有介于两者之间的作用域，没有命名空间或模块可以将功能进行分组。为了编写模块化代码，通过js的语法特性进行模块化。

当决定使用哪个功能时，我们需要谨记每个模块系统至少应该能够执行以下操作：

（1）定义模块接口，通过接口可以调用模块的功能。

（2）隐藏模块的内部实现，使模块的使用者无需关注模块内部的实现细节。同时，隐藏模块的内部实现，避免有可能产生的副作用和对bug的不必要修改。

- **使用对象、闭包和立即执行函数实现模块**

> 隐藏模块的内部实现----调用JavaScript函数创建新的作用域，我们可以在该作用域中定义变量，此时定义的变量只在当前函数中可见。因此，隐藏模块内部实现的一个方式是选择使用函数作为模块。采用这种方式，所有的函数变量都成为模块内部变量，模块外部不可见。

> 定义模块接口----使用函数实现模块意味着只能在模块内部访问变量。但是，如果使用其他代码调用该模块，我们必须定义简洁的接口，可以通过接口暴露模块提供的功能。一种实现方式是利用对象和闭包。思路是，通过函数模块返回代表模块公共接口的对象。该对象必须包含模块提供的方法，而这些方法将通过闭包保持模块内部变量，甚至在模块函数执行完成之后仍然保持模块变量。

    - 使用函数作为模块

```
(function countClicks() {
    //定义一个局部变量，储存点击次数
    let numClicks = 0;
    document.addEventListener("click",() => {
        //当用户单击时，计数器增加并返回当前次数
        alert(++numClicks);
    });
})();
```

- 变量numClicks处于函数countClicks内部，在单击事件函数闭包内保持活跃。该变量只能通过事件处理器调用。
- 只有一处调用countClicks函数，因此，与其定义函数再单独编写调用语句，不如使用立即执行函数或使用IIFE，定义并立即执行函数。
- 事件处理器创建的闭包保持局部变量numClicks。
- 浏览器具有单击事件处理器的引用。

- 模块模式：使用函数扩展模块，使用对象实现接口

  模块接口一般包含一组变量和函数。创建接口最简单的方式是使用JavaScript对象。

```
//创建一个全局模块变量，赋值为立即执行函数的执行结果
const MouseCounterModule = function() {
    //创建一个模块私有变量
    let numCLicks = 0;

    const handleClick = () => {
        alert(++numClicks);
    };

    return { //返回一个对象，代表模块的接口。通过闭包，可以访问模块私有变量和方法
        countClicks: () => {
            document.addEventListener("click", handleClick);
        }
    };
}();
```

- 模块扩展

```
//创建一个全局模块变量，赋值为立即执行函数的执行结果
const MouseCounterModule = function() {
    //创建一个模块私有变量
    let numCLicks = 0;

    const handleClick = () => {
        alert(++numClicks);
    };

    return { //返回一个对象，代表模块的接口。通过闭包，可以访问模块私有变量和方法
        countClicks: () => {
            document.addEventListener("click", handleClick);
        }
    };
}();

//立即调用一个函数，该函数接收需要扩展的模块作为参数
(function(moudle) {
    //定义新的私有变量和函数
    let numScrolls = 0;
    const handleScroll = () => {
        alert(++numScrolls);
    };

    //扩展模块接口
    moudle.countScrolls = () => {
```

```
        document.addEventListener("wheel", handleScroll);
    };
})(MouseCounterModule); //将模块传入作为参数


//现在模块公共接口有两个方法，我们可以这样使用
MouseCounterModule.countClicks();//初始接口
MouseCounterModule.countScrolls();//扩展模块新增方法
```

当扩展模块时，我们对其外部接口增加新功能，通常将模块传入立即执行函数。本例中两个函数是在不同的环境中定义的，不可以访问对方的内部变量。

扩展模块无法共享原有模块的内部属性。模块模式不能实现依赖其他模块的功能，所以出现了AMD和CommonJS两个标准。

使用AMD和CommonJS模块化JavaScript应用

AMD和CommonJS是两个互相竞争的标准，除了语法和原理的区别外，两者的主要区别是AMD的设计理念是基于浏览器，而CommonJS的设计是面向通用的JavaScript环境，不局限于浏览器。

- 使用AMD定义模块依赖于jQuery

```
//使用define函数指定模块及其依赖，模块工厂函数会创建对应的模块
define('MouseCounterModule', ['jQuery'], $ => {
    let numClicks = 0;
    const handleClick = () => {
        alert(++numClicks);
    };

    //模块的公共接口
    return {
        countClicks: () => {
            $(document).on("click", handleClick);
        }
    };
});
```

AMD提供的define函数接收三个参数：

  - 新创建模块的ID。使用该ID，可以在系统的其他部分引用该模块。
  - 当前模块依赖的模块ID列表。
  - 初始化模块的工厂函数，该工厂函数接收依赖的模块列表作为参数。

AMD的三个优点：

  - 自动处理依赖，我们无需考虑模块引入的顺序。
  - 异步加载模块，避免阻塞。
  - 在同一个文件中可以定义多个模块。

- 使用CommonJS定义模块

```
//模块文件  MouseCounterModule.js
const $ = require("jQuery");//同步引入jQuery模块
let numClicks = 0;
const handleClick = () => {
    alert(++numClicks);
};

//使用module.exports定义模块公共接口
module.exports = {
    countClicks: () => {
        $(document).on("click", handleClick);
    }
};
```

```
//在另一个文件引用该模块
const MouseCounterModule = require("MouseCounterModule.js");
MouseCounterModule.countClicks();
```

CommonJS具有两个优势：

- 语法简单。只需定义module.exports属性，剩下的模块代码与标准JavaScript无差异。引用模块的方法也很简单，只需要使用require函数。
- CommonJS是Node.js默认的模块格式，所以我们可以使用npm上的包。

ES6模块

ES6模块结合了AMD和CommonJS的优点：

- ES6模块语法相对简单，并且基于文件（每个文件就是一个模块）。
- 与AMD类似，ES6模块支持异步模块加载。

ES6模块的主要思想是必须显示地使用标识符导出模块，才能从外部访问模块。其他标识符甚至是在最顶级作用域中定义的标识符，只能在模块内使用。为了提供这个功能，ES6引入两个关键字：

- export-----从模块外部指定标识符
- import----导入模块标识符

导入和导出功能

- 从Ninja.js模块中导出

```
const ninja = "yoshi"; //在模块中定义一个顶级变量
export const message = "hello";

//使用关键字export分别导出定义的变量和函数
export function sayHiToNinja() {
    return message + " " + ninja; //通过模块公共API访问模块内部变量
}
```

- 在模块的最后一行导出

```
//定义所有的模块标识符
const ninja = "yoshi";
const message = "hello";

function sayHiToNinja() {
    return message + " " + ninja;
}
//将所有模块导出
export { message, sayHiToNinja};
```

- 从Ninja.js模块中导入

```
//使用关键字import从模块中导入标识符
import { message, sayHiToNinja} from "Ninja.js";
```

- 导入在Ninja.js模块中导出的全部标识符

```
import * as ninjaModule from "Ninja.js";//使用*导入所有的标识符
```

- 默认导出

  通常，我们不需要从模块中导出一组相关的标识符，只需要一个标识符来代表整个模块的导出。常见的情况是当模块中包含一个类：

```
//使用export default关键字定义模块的默认导出
export default class Ninja {
    constructure(name) {
        this.name = name;
    }
}

//使用默认导出的同时，我们还可以指定导出的名称
export function compareNinjas(ninja1, ninja2) {
    return ninja1.name = ninja2.name;
}
```

- 导入模块默认导出的内容

```
//导入模块默认导出的内容，不需要使用花括号{}，可以任意指定名称
import ImportedNinja from "Ninja.js";
//导入指定内容
import {compareNinjas} from "Ninja.js";
//等价于 importedNinja, {compareNinjas} from "Ninja.js";

//创建两个实例，并验证存在性
const ninja1 = new ImportedNinja("yoshi");
const nionja2 = new ImportedNinjaf("hattori");
assert(ninja1 !== undefined && ninja2 !== undefined, "yes");
assert(!compareNinjas(ninja1, ninja2), "can comapare");
```

- export与import时使用重命名

  需要时，可以重命名export和import。

```
//文件名  greet.js
//定义函数
function sayHi() {
    return "hello";
}
//  验证我们只能访问'sayHi'函数，而不能通过别名访问
assert(typeof sayHi === "function"
    && typeof sayHello === "undefined",
     "Within the module we can access only sayHi");
//通过关键字as设置别名
export { sayHi as sayHello}
```

```
//文件名 main.js
import {sayHello} from "greet.js";
//只能导入sayHello
assert(typeof sayHi === "function"
    && typeof sayHello === "undefined",
     "Within the module we can access only sayHi");
```

只能在export表达式中进行重命名，不能通过关键字export修改变量前缀或函数声明。**当对重命名的export执行import时只能通过别名导入。**

```
/************* Hello.js *************/
export function greet() {
   return "Hello";
} //  在Hello.js文件中导出名为greet的函数

/************* Salute.js *************/
export function greet() {
   return "Salute";
} //在Salute.js文件中导出名为greet的函数

/************* main.js *************/
import { greet as sayHello} from "Hello.js";
import { greet as salute} from "Salute.js";//使用as关键字重命名import的内容，避免命名冲突

assert(typeof greet === "undefined",
      "We cannot access greet");  //不能通过原始名称访问函数

assert(sayHello() === "Hello" && salute() === "Salute",
      "We can access aliased identifiers!"); //但可以访问别名

```

- **回顾ES6模块语法**

代码	含义
export const ninja = "yoshi"	导出变量
export function com() {}	导出函数
export class Ninja {}	导出类
export default class class Ninja {}	导出默认类
export default function com() {}	导出默认函数
const ninja = "a"; function com() {}; export {ninja, com};	导出存在的变量
export {ninja as com}	使用别名导出变量
import Ninja from "ninja.js"	导入默认导出
import {ninja, Ninja} from "ninja.js"	导入命名导出
import * as Ninja from "ninja.js";	导入模块中声明的全部导出内容
import {ninja as iNinja} from "ninja.js"	通过别名导入模块中声明的全部导出内容

第十二章 DOM操作*

总结

- 将HTML字符串转换为DOM元素包括以下步骤。
  - 确保HTML字符串是有效的HTML代码
  - 将其包装成封闭的标记，符合浏览器规则要求
  - 通过DOM元素的innerHTML属性将HTML插入虚拟DOM元素
  - 将创建的DOM节点提取出来
- 为了快速插入DOM节点，请使用DOM片段，因为可以在单个操作中注入片段，从而减少操作次数。
- DOM元素属性和特性虽然挂钩，但并不总是相同。我们可以通过使用getAttribute和setAttribute方法读取和写入DOM属性，同时也可以使用对象属性符号写入DOM属性。
- 使用属性和特性时，也有必要了解自定义属性。我们在DOM元素上自定义的特性，仅用于自定义信息，不能与元素属性等同看待或者使用。
- 元素style属性是一个对象，它含有与元素标记中指定的样式值相对应的属性。要获得计算后样式，需要同时考虑样式表中设置的样式，请使用getCompustedStyle方法。
- 要获取HTML元素的尺寸，请使用offsetWidth和offsetHeight属性。
- 当代码对DOM进行一系列连续的读取和写入操作时，浏览器每次都会进行强制重新进行计算布局信息，这会引起布局抖动。进而导致Web应用程序运行和响应速度变慢。

- 请批量更新DOM

将HTML字符串转换成DOM

innerHTML属性将字符串转换成DOM，转换步骤：

- 确保HTML字符串是合法有效的
- 将它包裹在任意符合浏览器规则要求的闭合标签内
- 使用innerHTML将这串HTML插入到一个需求DOM中
- 提取该DOM节点

在下面这个骨架HTML中插入元素：

```
<--!骨架HTML--!>
<option>yoshi</option>
<option>kuma</option>
<table/>
```

```
//首先确保自闭合元素被正确解释

//使用正则表达式匹配我们不需要关心的元素名
const tags =
/^(area|base|br|embed|hr|img|input|keygen|link|menuitem|meta|param|source|track|wbr)$/i;

//转换函数，通过使用正则表达式将自闭合标签转为‘正常’形式的标签对
function convert(html) {
    return html.replace(/(<(\w+)[^>]*?)\/>/g,(all,front,tag) => {
        return tags.test(tag)? all : front + "></" + tag + ">";
    });
}
```

执行该函数后，代码变为：

```
<--!骨架HTML--!>
<option>yoshi</option>
<option>kuma</option>
<table><table/>
```

包装HTML：根据HTML语义，一些HTML元素必须包装在某些容器元素中才能被注入。我们有两种方式来进行包装，这两种方式都需要构建问题元素和容器之间的映射。

- **通过innerHTML将该字符串直接注入到它的特定父元素中，该父元素提前使用内置的document.crateElement创建好。**尽管大多数情况下的大部分浏览器都支持这种方式，但仍不能保证完全通用。
- **HTML字符串可以在使用对应父元素包装后，直接注入到任意容器元素中**，这样更保险，也更麻烦。

将元素标签转为一系列DOM节点：

```
function getNodes(htmlString, doc) {
    const map = {
      "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],
      "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],
```

```
        "<tr>":[2,"<table><thead>","</thead></table>"],
        "<option>":[1,"<select multiple>","</select>"],
        "<optgroup>":[1,"<select multiple>","</select>"],
        "<legend>":[1,"<fieldset>","</fieldset>"],
        "<thead>":[1,"<table>","</table>"],
        "<tbody>":[1,"<table>","</table>"],
        "<tfoot>":[1,"<table>","</table>"],
        "<colgroup>":[1,"<table>","</table>"],
        "<caption>":[1,"<table>","</table>"],//需要特殊父级容器的元素映射表。每个条目都包含新节点的深度，以及父元素的HTML头尾片断

        "<col>":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
    };

    const tagName = htmlString.match(/<\w+/);
    let mapEntry = tagName ? map[tagName[0]] : null;//匹配起始标记和标签名
    if (!mapEntry) { mapEntry = [0, " "," " ];} //  如果映射表中有匹配，使用匹配结果：如果没有，则构造空的“父”标记，深度设为0，作为结果
    let div = (doc || document).createElement("div");//创建用来包含新节点的</div>。注意，如果传入了文档(document)对象，使用传入的，否则默认当前document对象
    div.innerHTML = mapEntry[1] + htmlString + mapEntry[2]; //  使用匹配得到的父级容器元素，包装起传入的HTML字符串，并将其注入到新创建的<div>中
    while (mapEntry[0]--) { div = div.lastChild;} //参照映射关系定义的深度，向下遍历刚刚创建的DOM树，最终得到的应该是新创建的2元素。
    return div.childNodes;   //返回新创建的元素
}
assert(getNodes("<td>test</td><td>test2</td>").length === 2,
       "Get two nodes back from the method.");
assert(getNodes("<td>test</td>")[0].nodeName === "TD",
       "Verify that we're getting the right node.");

```

批量DOM读取和写入以避免布局抖动

```
div id="ninja">I’m a ninja</div>
<div id="samurai">I’m a samurai</div>
<div id="ronin">I’m a ronin</div>//  定义一组HTML元素
<script>
   const ninja = document.getElementById("ninja");
   const samurai = document.getElementById("samurai");
   const ronin = document.getElementById("ronin");  //通过DOM获取元素

   const ninjaWidth = ninja.clientWidth;
   ninja.style.width = ninjaWidth/2 + "px";
   const samuraiWidth = samurai.clientWidth;
   samurai.style.width = samuraiWidth/2 + "px";

   const roninWidth = ronin.clientWidth;
   ronin.style.width = roninWidth/2 + "px";  //执行一系列连续的读写操作，修改DOM使得布局失效
</script>

```

第十三章 历久弥新的事件

深入事件循环

> 对于初学者，事件循环不仅仅是包含事件队列，而是具有至少两个队列，除了事件，还要保持浏览器执行的其他操作。这些操作被称为任务，并且分为两类：宏任务（任务）和微任务。

宏任务的例子有很多，包括创建主文档对象、解析HTML、执行主线（全局）JavaScript代码，更改当前URL以及各种事件，如页面加载、输入、网络事件和定时器事件。从浏览器的角度来看，宏任务代表一个个离散的、独立工作单元。运行完任务后，浏览器可以继续其它调度，如重新渲染页面的UI或执行垃圾回收。

微任务是更小的任务。微任务更新应用程序的状态，但必须在浏览器任务继续执行其它任务之前执行，浏览器任务包括重新渲染页面的UI。微任务的案例包括promise回调函数、DOM发生变化等。微任务需要尽可能快地、通过异步方式执行，同时不能产生全新的微任务。微任务使得我们能够在重新渲染UI之前执行指定的行为，避免不必要的UI重绘，UI重绘会使应用程序的状态不连续。

> 事件循环的实现至少应该含有一个用于宏任务的队列和至少一个用于微任务的队列。大部分的实现通常会更多用于不同类型的宏任务和微任务的队列。这使得事件循环能够根据任务类型进行优先处理。

事件循环基于两个基本原则：

- 一次处理一个任务
- 一个任务开始后直到运行完成，不会被其他任务中断

我们来看看图13.1，它描绘了这两个原则。

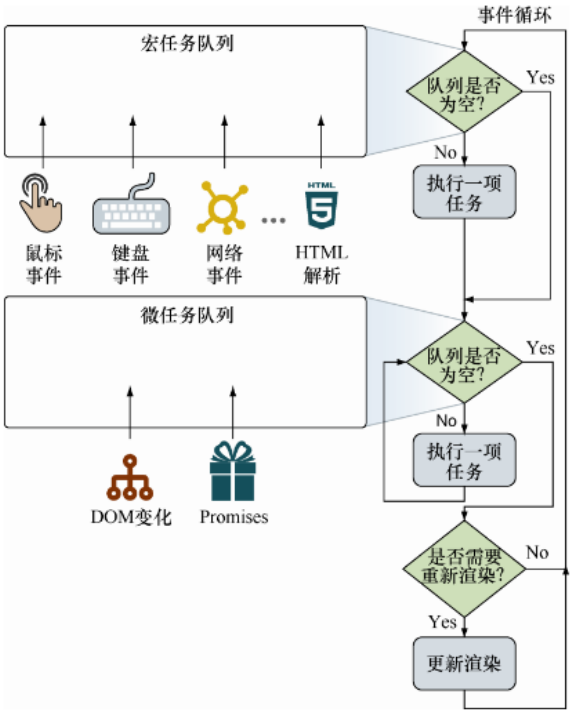


图13.1　事件循环通常至少需要两个任务队列：宏任务队列和微任务队列。两种队列在同一时刻都只执行一个任务

处理宏任务和微任务队列之间的区别：单次循环迭代中，最多处理一个宏任务（其余的在队列中等待），而队列中的所有微任务都会被处理。

仅含宏任务的事件循环案例：

```
//全局JavaScript代码
//两个按钮以及对应的两个单击处理器
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
    const firstButton = document.getElementById("firstButton");
    const secondButton = document.getElementById("secondButton");

    firstButton.addEventListener("click", function firstHandler() {
        //在第一个按钮上注册点击事件处理器
        //8ms
    });
    secondButton.addEventListener("click", function secondHandler() {
        //在第二个按钮上注册点击事件处理器
        //5ms
    });
    //15ms
</script>
```

这段代码：主线程JavaScript代码执行时间需要15ms

第一个单击事件处理器需要运行8ms

第二个单击事件处理器需要运行5ms

假设一个用户在代码执行后5ms时单击第一个按钮，随后12ms时单击第二个按钮，如下图：

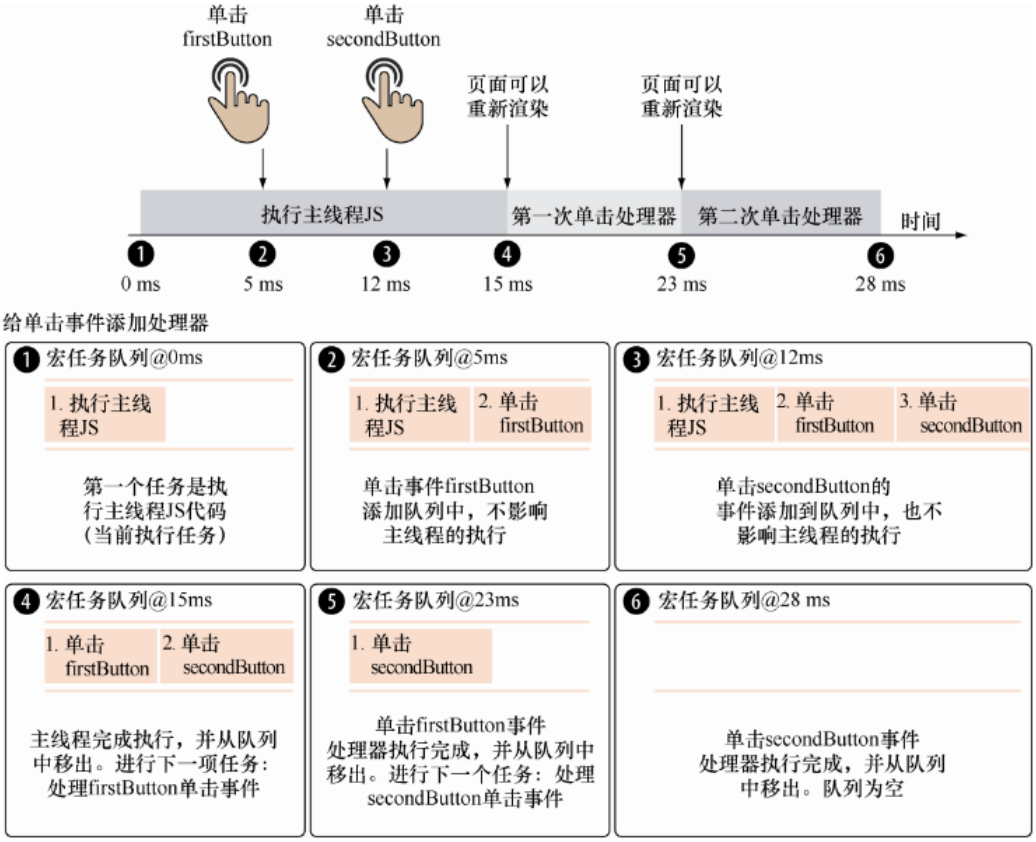


图13.2　时间表显示了当事件发生时任务是如何添加到队列中的。当一个任务执行完成，事件循环将该任务移除队列，并开始执行下一个任务

本示例强调如果其它任务正在执行，那么事件则需要按顺序等待执行。

同时含有宏任务和微任务的示例：

```
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
    const firstButton = document.getElementById("firstButton");
    const secondButton = document.getElementById("secondButton");

    firstButton.addEventListener("click", function firstHandler() {
        //在第一个按钮上注册点击事件处理器
        Promise.resolve().then(() => {
            //4ms
        });
    });
    secondButton.addEventListener("click", function secondHandler() {
        //在第二个按钮上注册点击事件处理器
        //5ms
    });
    //15ms
</script>
```

此时加入promise并需要运行4ms的传入回调函数。

在本例中，我们创建立即兑现的promise。JavaScript引擎本应该立即调用回调函数，因为我们已经知道promise成功兑现，但为了连续性，JavaScript引擎不会这么做，仍然会在firstHandler代码执行完成之后再异步调用回调函数。通过创建微任务，将回调函数放入微任务队列。

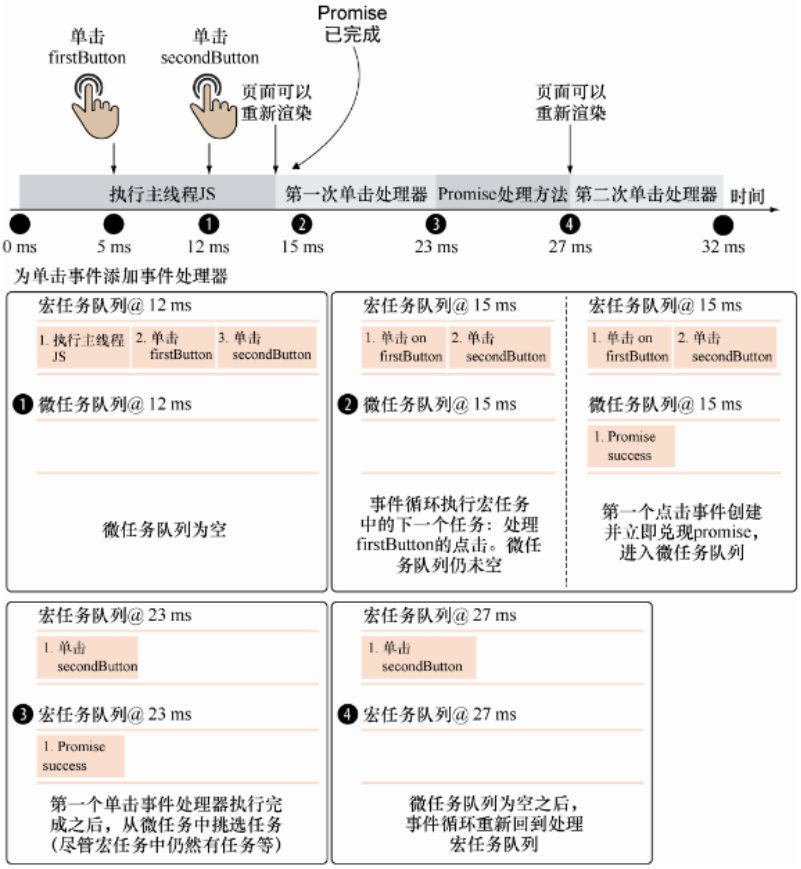


图13.6　如果微任务队列中含有微任务，不论队列中等待的其他任务，微任务都将获得优先执行权。在本例中，promise微任务优先于secondButton单击任务开始执行

计时器：延迟执行和间隔执行

计时器能延迟一段代码的运行，延迟时长**至少**是指定的时长（单位是ms）。我们可以使用计时器将长时运行的任务分解成不阻塞事件循环的小任务，以阻止浏览器渲染，浏览器渲染过程会使得应用程序运行缓慢，没有反应。

- 浏览器提供两个创建计时器的方法：setTimeout和setInterval
- 浏览器也提供了两个清除计时器的方法：clearTimeout和clearInterval

> 计时器提供一种异步延迟执行代码片段的能力，至少要延迟指定的毫秒数。**因为JavaScript单线程的本质，我们只能控制计时器何时被加入队列中，而无法控制何时执行。**

处理事件

注册事件处理器

```
<button id="myButton">Click</button>
<script>
    const myButton = document.getElementById("button");
    myButton.addEventListener("click", function myHandler(event) {
        //使用addEventListener注册事件处理程序
        assert(event.target === myButton, "yes");
    });
</script>
```

处理一个事件有两种方式：

- 捕获：首先被顶部元素捕获，并依次向下传递。
- 冒泡：目标元素被捕获后，事件处理器转向冒泡，从目标元素向顶部元素冒泡。

这两种方式如图13.15所示。

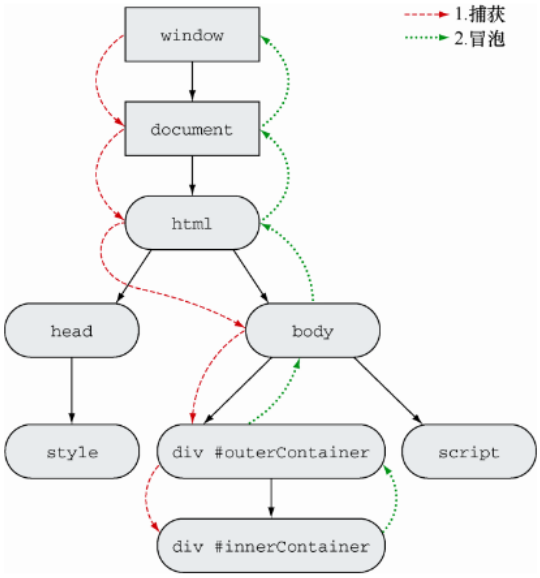


图13.15　通过捕获，事件最终传递到目标元素。通过冒泡，事件从目标元素向上冒泡

自定义事件

为什么要采用自定义事件?

假设以共享模式处理业务，我们想让页面上的代码知道何时达到指定条件。如果我们使用熟练工常用的全局函数的方法，劣势是我们共享的代码为函数，需要定义一个固定的名称，并且使用共享函数的页面的代码均需要使用这样的一个函数。此外，如果指定的条件发生时需要处理多件事件该如何操作？发送多条通知势必会非常麻烦、混乱。这些缺点是**紧耦合的结果**，检测匹配条件的代码需要了解满足条件的代码细节。

松耦合，当代码触发匹配条件时，不需要指定关于条件的细节代码。事件处理器的优点之一是，我们可以创建任意数量的事件处理器，并且事件处理器之间是完全独立的。所以事件处理器是松耦合很好的例子。

创建自定义事件

```
<style>
   #whirlyThing {display: none; }
</style>
<button type="button" id="clickMe">Start</button>//单击按钮，模拟Ajax请求
<img id="whirlyThing" src="whirly-thing.gif" />  //使用旋转的图片表示正在加载

<script>
   function triggerEvent(target, eventType, eventDetail) {
     const event = new CustomEvent(eventType, {  //使用CustomEvent构造器创建一个新事件
       detail: eventDetail//通过detail属性为事件对象传入信息
     });
     target.dispatchEvent(event); //  使用内置的dispathEvent方法解除事件绑定
  }

  function performAjaxOperation() {
     triggerEvent(document, 'ajax-start', {url: 'my-url'});
     setTimeout(() => {
       triggerEvent(document, 'ajax-complete'); //  使用延迟计时器模拟Ajax请求。开始执行时，触发ajax-start事件，一段时间过去之后，激活ajax-complete事件。传入URL作为事件额外信息
     }, 5000);
  }

   const button = document.getElementById('clickMe');
   button.addEventListener('click', () => {
     performAjaxOperation();  //当单击一个按钮时，Ajax操作开始
  });

  document.addEventListener('ajax-start', e => {//  显示旋转图片，处理ajax-start 事件
     document.getElementById('whirlyThing').style.display = 'inline-block';
     assert(e.detail.url === 'my-url', 'We can pass in event data'); //验证我们可以访问附加的事件数据
  });

  document.addEventListener('ajax-complete', e => {//  处理ajax-complete事件，隐藏旋转图片
       document.getElementById('whirlyThing').style.display = 'none';
  });
</script>

```

总结

- 事件循环任务代表浏览器执行的行为。任务分为两类：

    - 宏任务是分散、独立的浏览器事件，如创建主文档对象、处理各种事件、更改URL等。
    - 微任务是应该尽快执行的任务，包括promise回调和DOM突变。
- 由于单线程的执行模型，一次只能处理一个任务，一个任务开始执行后不能被另一个任务所中断。事件循环通常至少有两个事件队列：宏任务队列和微任务队列。
- 异步定时器提供延迟执行一段代码的能力，至少延迟指定的毫秒数。
- 使用setTimeout函数在指定延迟时间后执行回调。
- 使用setInterval函数来启动一个计时器，将尝试在指定的延迟间隔执行回调，直至被清除。
- 两个函数均返回对应的计时器ID，通过clearTimeout和clearInterval函数，我们可以使用计时器ID来取消计时器。
- 使用计时器，将计算开销很高的代码分解成可管理的、不阻塞浏览器的代码块。
- DOM是元素的分层树，发生在一个元素上的事件通常是通过DOM进行代理的，有以下两种机制：
    - 事件捕获模式：事件从顶部元素向下传递到目标元素。
    - 事件冒泡模式：事件从目标元素向上冒泡到顶部元素。
- 当调用事件处理器时，浏览器也会传入一个事件对象。通过该对象的属性可访问发生事件的目标元素。通过处理器，使用this关键字引用在处理器上注册过的元素。
- 通过内置的CustomEvent构造函数和dispatchEvent方法，创建和分发自定义事件，减少应用程序不同部分之间的耦合。

第十四章 跨浏览器开发技巧

总结

- 虽然情况有了较大的改善，但是浏览器不可能没有缺陷，并且通常不支持Web标准。
- 当编写JavaScript应用程序时，选择支持的浏览器和平台是一个重要的考虑因素。
- 由于不可能支持所有组合，因此需要牺牲质量赢取覆盖率。
- 编写可运行于多种浏览器的JavaScript代码，最大的挑战是：缺陷修复、回归、浏览器缺陷、特性缺失以及外部代码。
- 可重用的跨浏览器开发涉及了以下几个因素:
    - 代码体积---保持文件体积尽可能小。
    - 性能开销---以优秀作为最低性能标准。
    - API质量---保证不同浏览器上API一致性。
- 这些因素的权衡没有绝对的计算公式。
- 每位开发人员的个人努力是可用于权衡考虑的因素。
- 通过使用智能技术，如功能检测，当可重用代码受到攻击时，我们可以有效抵御，不需要做任何不必要的牺牲。

> **通俗来讲，JS垫片就是，在低级环境中用高级语法时，在低级环境中手动实现的高级功能，模拟高级环境** 。