Broadview®
www.broadview.com.cn

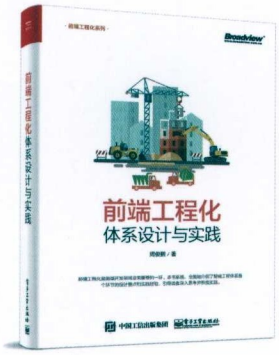
本书并没有把所有的前端知识按部就班地罗列出来，而是努力地向读者传递一种行之有效的学习方法，总结出了一套适合大多数人的学习方式：围绕核心，渐进增强。

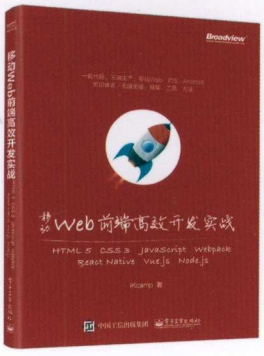
JavaScript
核心技术开发解密

阳波 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

好书分享


前端工程化体系设计与实践
Broadview
前端工程化
体系设计与实践


移动Web前端高效开发实战
Broadview
移动Web前端高效开发实战
HTML 5 CSS 3 Javascript Webpack
React Native Vue.js Node.js
iKcamp 著

JavaScript
核心技术开发解密

阳波 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书针对 JavaScript 中的核心技术，结合前沿开发实践，对 JavaScript 的内存、函数、执行上下文、闭包、面向对象、模块等重点知识，进行系统全面的讲解与分析。每一个知识点都以实际应用为依托，帮助读者更加直观地吸收知识点，为学习目前行业里的流行框架打下坚实基础。本书适合 JavaScript 初学者，有一定开发经验但是对于 JavaScript 了解不够的读者，以及开发经验丰富但没有形成自己知识体系的前端从业者。

图书在版编目（CIP）数据

JavaScript 核心技术开发解密 / 阳波编著. – 北京: 电子工业出版社, 2018.3

ISBN 978-7-121-33696-6

I. ①J…II. ①阳…III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2018）第 029430 号

策划编辑：安　娜
责任编辑：安　娜
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：14.5　字数：276 千字
版　　次：2018 年 3 月第 1 版
印　　次：2018 年 3 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

在阅读这本书之前，不知道大家有没有思考过一个问题：

前端学习到底有没有捷径？

在我看来，学习的捷径并不是指不用付出多少努力就能够获得成功，而是在我们付出努力之后，能够感觉到自己的努力没有白费，能够学到更多的知识，能够真正做到一分耕耘，一分收获。

所以学习有没有捷径？我的答案是：一定有。

其实大多数人并不是不想付出努力，而是不知道如何去努力，不知道如何有效地努力。我们想要学好一个知识，想要掌握一门技术，但是往往不知从何下手。

前端的学习也是如此。也许上手简单的 HTML/CSS 知识，会给刚开始学习的读者一个掌握起来很容易的印象。但是整个前端知识体系繁杂而庞大，导致大多数人在掌握了一些知识之后，仍然觉得自己并没有真正入门，特别是近几年，前端行业的从业人员所要掌握的知识越来越庞杂，入门的门槛也越来越高，甚至进阶道路也变成了一场马拉松。

也许在几年以前，我们只需会用 jQuery 就可以说自己是一名合格的前端开发者，但是现在的 JavaScript 语言已经不再是几年前那样，只需处理一些简单的逻辑就足够了。随着前后端分离的方式被越来越多的团队运用于实践，用户对 UI 的要求越来越高，对性能的要求也越来越高，JavaScript 承载了更多的任务。虽然前端行业大热，但是我们的学习压力也随之倍增。

所以我一直在思考，在这样的大环境背景下，对于新人朋友来说，什么样的学习方式能让我们的学习效率更高？或者说，一本什么样的前端书籍才算是好书？

是将所有的前端知识按部就班地罗列出来告诉大家吗？肯定不是。

很多书籍，以及各类官方文档其实都在做这件事。但是对于多数读者来说，把所有知识罗列出来摆在眼前，并不是一个能够掌握它们的有效方式。这种学习方法不仅效率低下，而且学完之后，也并不知道在实践中到底如何使用它们，我们其实是迷茫的。

所以，如果有一本书，它在努力地向读者传递一种行之有效的学习方法，那么对于适合这种学习方法的读者来说，就一定是一本好书。

这就是我们这本书努力的方向。

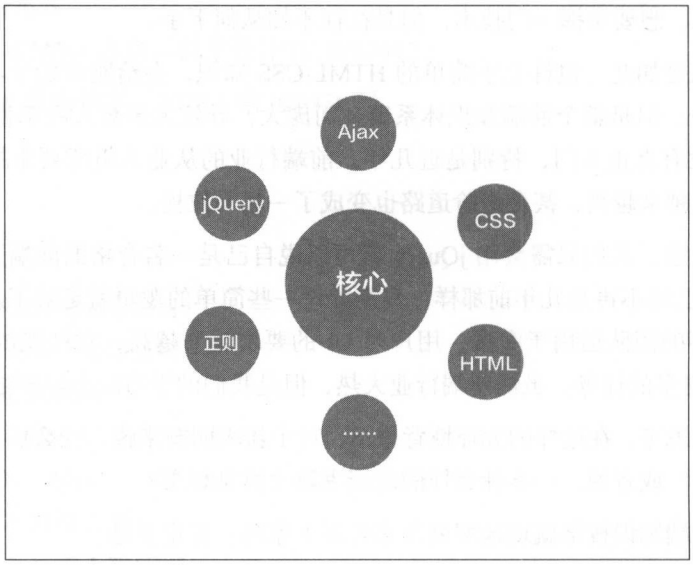
凭借多年的工作经验，在长期写博客并与读者互动的过程中，我总结出了一套适合大多数人的学习方式，那就是：

围绕核心，渐进增强。

本书将整个 JavaScript 相关的知识点简单粗暴地划分为核心知识与周边知识。

周边知识的特点就是相对独立，不用非得学会了其他的知识点之后才能掌握它，也不用掌握了它之后才能学习其他的知识。例如 Ajax，如果仅仅只是想要使用它，那么用别人封装好的方法，通过官方文档或者搜索引擎，只需要两分钟你就知道怎么使用。周边知识不会成为我们学习的瓶颈。

而核心知识不一样，核心知识是整个前端知识体系的骨架所在。它们前后依赖，环环相扣。例如，在核心知识链中，如果你搞不清楚内存空间管理，你可能就不会真正明白闭包的原理，就没办法完全理解原型链，这是一个知识的递进过程。我们在学习过程中遇到的瓶颈，往往都是由于某一个环节的核心知识没有完整掌握造成的。而核心知识的另一个重要性就在于，它们能够帮助我们更加轻松地掌握其余的周边知识。



所以，如果读者知道这条核心知识链到底是什么，并且彻底地掌握它们，那么你就已经具备了成为一名优秀前端程序员的能力。这样的能力能够让你在学习其他知识点的时候方向明确，并且充满底气。

所以这本书的主要目的就在于帮助读者拥有这样的进阶能力。

基于这个思路，这本书的呈现方式必定与其他图书不同。本书不会按部就班地告诉你如何声

明变量、如何声明函数，不会罗列出所有的基础知识，对于基础知识的传授，《JavaScript 高级编程》已经做得足够好，因此没有必要重复做同样的事情。我会一步一步与大家分享这条完整的核心链。我的期望是，当大家学完这本书中的知识后，能够对前端开发的现状有一个大致的了解，知道什么知识是最有用的，什么知识是工作中需要的，拥有进一步学习流行前端框架的能力，拥有在前端方向自主学习、自主进步的知识基础与能力。

最后希望在这本书的陪伴下，大家能有一个愉快的、充实的学习历程。

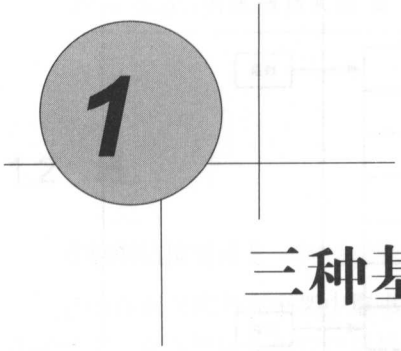
读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- ◎ **下载资源：**本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- ◎ **提交勘误：**您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- ◎ **交流互动：**在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：`http://www.broadview.com.cn/33696`

目录



三种基础数据结构

在 JavaScript 中，有三种常用的数据结构是我们必须了解的，它们分别是栈（stack）、堆（heap）、队列（queue）。它们是理解整个核心的基础，在 JavaScript 中分别有不同的应用场景，因此先来介绍它们。

1.1 栈

当我们在学习中遇到栈这个名词时，可能面临的是不同的含义。如果没有理清不同的应用场景，就会给我们的理解带来困惑。

场景 1：栈是一种数据结构，它表达的是数据的一种存取方式，这是一种理论基础。

场景 2：栈可用来规定代码的执行顺序，在 JavaScript 中叫作函数调用栈（call stack），它是根据栈数据结构理论而实现的一种实践。理解函数调用栈的概念非常重要，我们会在后续的章节里详细说明。

场景 3：栈表达的是一种数据在内存中的存储区域，通常叫作栈区。但是 JavaScript 作为一门高级程序语言，并没有同其他语言那样区分栈区或堆区，因此这里不做扩展。我们可以简单粗暴地认为在 JavaScript 中，所有的数据都是存放在堆内存空间中的。

这里需要重点掌握栈这种数据结构的原理与特点，学习它的最终目的是掌握函数调用栈的运行方式。下面可以通过乒乓球盒子这个案例来简单理解栈的存取方式，如图1-1所示。

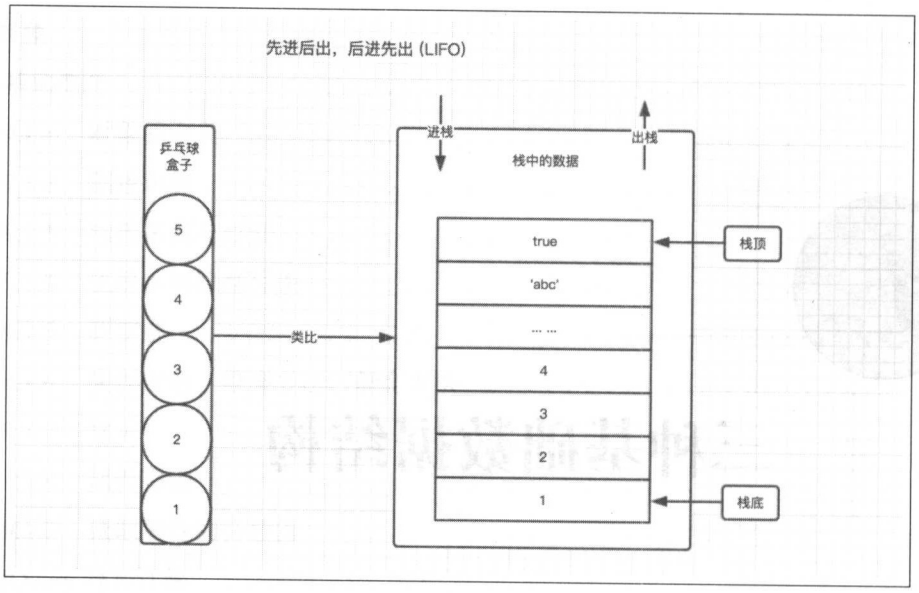


图 1-1　乒乓球盒子

往图1-1所示的乒乓球盒子中依次放入乒乓球，当想要取出来使用的时候，处于盒子顶层的乒乓球 5，它一定是最后被放进去并且最先被取出来的。而要想使用底层的乒乓球 1，则必须先将上面的所有乒乓球取出来之后才能取出。但乒乓球 1 是最先放入盒子的。

这种乒乓球的存取方式与栈数据结构如出一辙。这种存取方式的特点可总结为**先进后出，后进先出**（LIFO，Last In，First Out）。如图1-1右侧所示，处于栈顶的数据 true，最后进栈，最先出栈。处于栈底的数据 1，最先进栈，最后出栈。

在 JavaScript 中，数组（Array）提供了两个栈方法来对应栈的这种存取方式，它们在实践中十分常用。

push：向数组末尾添加元素（进栈方法）。

push 方法可以接收任意参数，并把它们逐个添加到数组末尾，并返回数组修改后的长度。

```
var a = [];
a.push(1);          // a: [1]
a.push(2, 4, 6);    // a: [1, 2, 4, 6]
var l = a.push(5); // a: [1, 2, 4, 6, 5] l: 5
```

pop：弹出数据最末尾的一个元素（出栈方法）。

pop 方法会删除数组最末尾的一个元素，并返回。

```
var a = [1, 2, 3];
a.pop(); // a: [1, 2]

// a.pop()的返回结果为 3
```

1.2 堆

堆数据结构通常是一种树状结构。

它的存取方式与在书架中取书的方式非常相似。书虽然整齐地摆放在书架上，但是只要知道书的名字，在书架中找到它之后就可以很方便地取出，我们甚至不用关心书的存放顺序，即不用像从乒乓球盒子中取乒乓球那样，必须将一些乒乓球拿出来之后才能取到中间的某一个乒乓球。

图 1-2是 testHeap 示意图。

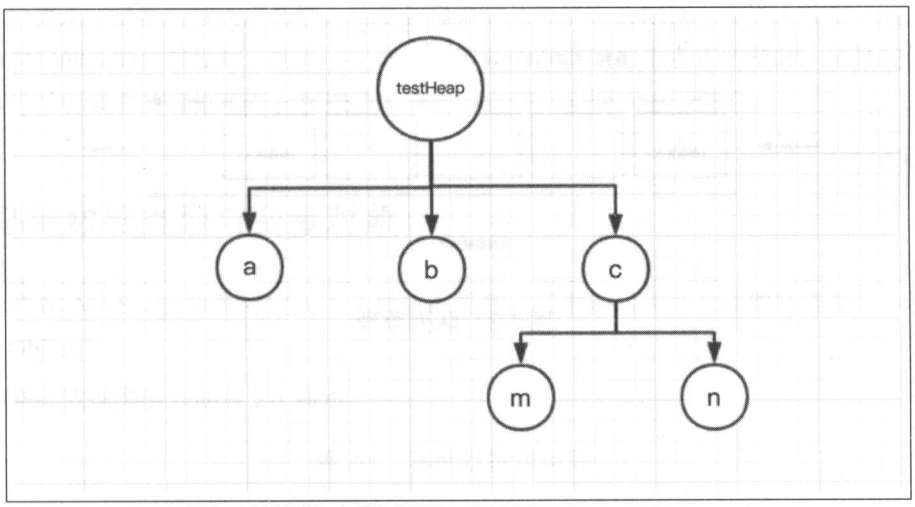


图 1-2 testHeap 示意图

该示意图可以用字面量对象的形式体现出来。

```
var testHeap = {
    a: 10,
    b: 20,
```

```
    c: {
        m: 100,
        n: 110
    }
}
```

当我们想要访问 a 时，只需通过 testHeap.a 来访问即可，而不用关心 a、b、c 的具体顺序。

1.3 队列

在 JavaScript 中，理解队列数据结构的目的是为了搞清楚事件循环（Event Loop）机制到底是怎么回事。在后续的章节中会详细分析事件循环机制。

队列（queue）是一种先进先出（FIFO）的数据结构。正如排队过安检一样，排在队伍前面的人一定是最先过安检的人。队列原理如图 1-3所示。

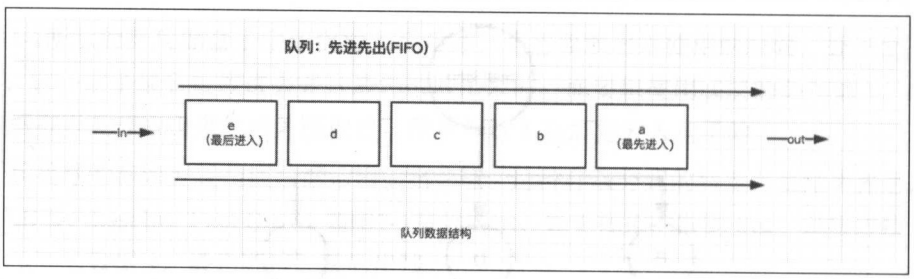


图 1-3 队列原理

内存空间

因为 JavaScript 有垃圾自动回收机制，所以对于前端开发人员来说，内存空间并不是一个经常被提及的概念，很容易被大家忽视。特别是很多非计算机专业的读者在进入前端行业之后，通常对内存空间的认知比较模糊，甚至一无所知。但是内存空间却是真正的基础，这是我们进一步理解闭包等重要概念的理论基石，所以非常有必要花费一点时间去了解它。

2.1 基础数据类型与变量对象

最新的 ECMAScript 标准号定义了 7 种数据类型，其中包括六种基础数据类型与一种引用数据类型（Object）。

其中基础数据类型表如表 2-1 所示。

表 2-1 基础数据类型表

类 型	值
Boolean	只有两个值：true 与 false
Null	只有一个值：null
Undefined	只有一个值：undefined
Number	所有的数字
String	所有的字符串
Symbol	符号类型 var sym = Symbol('testsymbol')

由于目前常用的浏览器版本还不支持 Symbol，而且通过 babel 编译之后的代码量过大，因此在实践中建议暂时不要使用 Symbol。

下面来探讨一个问题，有一个很简单的例子如下所示。

```
function fn() {
  var a1 = 10;
  var a2 = 'hello';
  var a3 = null;
}
```

现在需要思考的是，当运行函数 fn 时，它其中的变量 a1、a2、a3 都保存在什么地方？

函数运行时，会创建一个执行环境，这个执行环境叫作执行上下文（Execution Context，我们会在后续的章节详细介绍它）。在执行上下文中，会创建一个叫作变量对象（VO，后续章节详细学习）的特殊对象。基础数据类型往往都保存在变量对象中，如图 2-1所示。

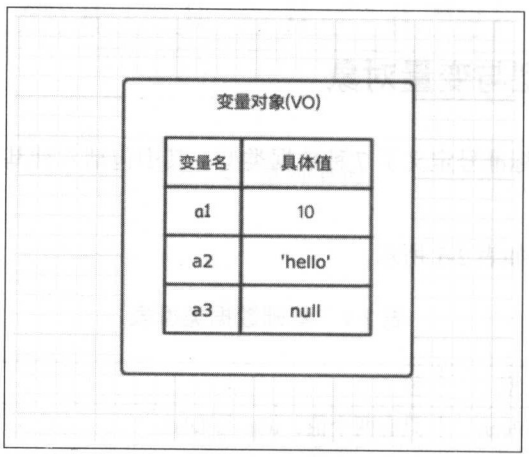


图 2-1　变量对象

变量对象也存在于堆内存中，但是由于变量对象有特殊职能，因此在理解时，建议仍然将其与堆内存空间区分开来。

2.2 引用数据类型与堆内存空间

引用数据类型（Object）的值是保存在堆内存空间中的对象。在 JavaScript 中，不允许直接访问堆内存空间中的数据，因此不能直接操作对象的堆内存空间。在操作对象时，实际上是在操作对象的引用而不是实际的对象。因此，引用数据类型都是按引用访问的。这里的引用，可以理解为保存在变量对象中的一个地址，该地址与堆内存中的对象相关联。

为了更好地理解变量对象与堆内存，下面用一个例子与图解配合讲解。

```
function foo() {
    var a1 = 10;
    var a2 = 'hello';
    var a3 = null;
    var b = { m: 20 };
    var c = [1, 2, 3];
}
```

如图 2-2所示，当我们想要访问堆内存空间中的引用数据类型时，实际上是通过一个引用（地址指针）来访问的。

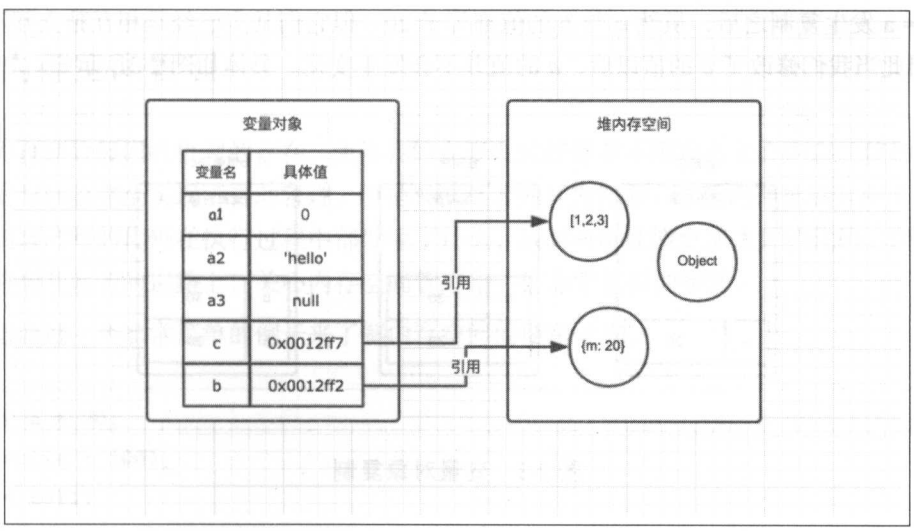


图 2-2 变量对象地址指针

在前端面试题中，我们常常会遇到这样一个类似的题目。

```
// demo01.js
var a = 20;
var b = a;
b = 30;

// 这时a的值是多少
```

```
// demo02.js
var m = { a: 10, b: 20 }
var n = m;
n.a = 15;

// 这时m.a的值是多少
```

在 demo01 中，基础数据类型发生了一次复制行为。在 demo02 中，引用数据类型发生了一次复制行为。

当变量对象中的数据发生复制行为时，新的变量会被分配到一个新的值。在 demo01 中，通过 var b = a 发生复制之后，虽然 *a* 与 *b* 的值都等于 20，但它们其实已经是相互独立互不影响的值了。因此当我们修改了 *b* 的值以后，*a* 的值并不会发生变化。具体如图 2-3所示。

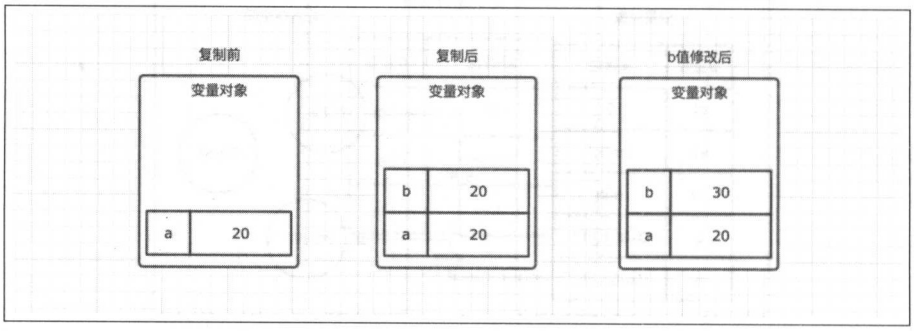


图 2-3　变量对象复制

在 demo02 中，通过 var n = m 发生了一次复制行为。引用类型的复制同样会为新的变量自动分配一个新的值并保存在变量对象中。但不同的是，这个新的值，仅仅只是引用类型的一个地址指针。当地址指针相同时，尽管它们相互独立，但是它们指向的具体对象实际上是同一个。

因此，当修改 n 时，m 也会发生变化，这就是引用类型的特性。具体如图 2-4所示。

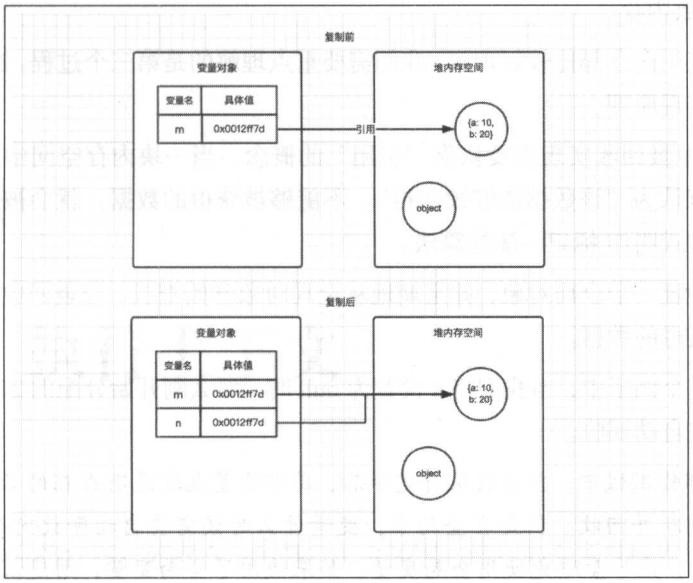


图 2-4 变量对象引用复制

2.3 内存空间管理

因为自动垃圾回收机制的存在，使得我们在开发时好像并不用那么关心内存的使用问题，内存的分配与回收完全实现了自动管理。但是根据笔者的开发经验，了解内存机制有助于自己清晰地认知到自己写的代码在执行过程中都发生了什么，从而写出性能更加优秀的代码。因此在成为更好的前端开发者的道路上，关心内存空间管理是一件非常重要的事情。

下面通过一个非常简单的例子来了解内存空间的使用过程。

```
var a = 20;
alert(a + 100);
a = null;
```

上面的三条语句，分别对应如下三个过程。

◎ 分配内存；

◎ 使用分配到的内存；

◎ 不需要时释放内存。

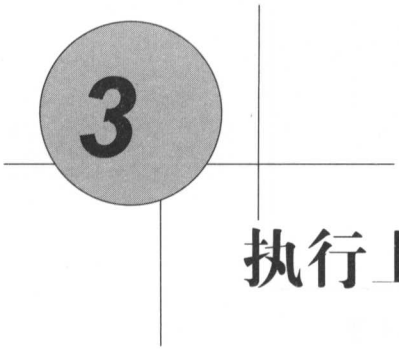
分配内存与使用内存都比较好理解，我们需要重点理解的是第三个过程。这里涉及 JavaScript 垃圾回收机制的实现原理。

JavaScript 的垃圾回收实现主要依靠“引用”的概念。当一块内存空间中的数据能够被访问时，垃圾回收器就认为“该数据能够被获得”。不能够被获得的数据，就会被打上标记，并回收内存空间。这种方式叫作 **标记—清除算法**。

这个算法会设置一个全局对象，并定期地从全局对象开始查找，垃圾回收器会找到所有可以获得与不能够被获得的数据。

因此在上面这个例子中，当我们将 a 设置为 null 时，那么刚开始分配的 20，就无法被访问到了，而是很快会被自动回收。

注意：在局部作用域中，当函数执行完毕后，局部变量也就没有存在的必要了，因此垃圾收集器很容易做出判断并回收。但是在全局中，变量什么时候需要自动释放内存空间则很难判断，因此我们在开发时，应尽量避免使用全局变量。如果使用了全局变量，则建议不再使用它时，通过 a = null 这样的方式释放引用，以确保能够及时回收内存空间。



执行上下文

JavaScript 代码在执行时，会进入一个执行上下文中。执行上下文可以理解为当前代码的运行环境。

JavaScript 中的运行环境主要包括以下三种情况。

◎ 全局环境：代码运行起来后会首先进入全局环境。

◎ 函数环境：当函数被调用执行时，会进入当前函数中执行代码。

◎ eval 环境：不建议使用，这里不做介绍。

因此可以预见的是，在一个 JavaScript 程序中，必定会出现多个执行上下文。

JavaScript 引擎会以栈的方式来处理它们，这个栈，就是前面多次提到的函数调用栈。函数调用栈规定了 JavaScript 代码的执行顺序。栈底永远都是全局上下文，栈顶则是当前正在执行的上下文。

当代码在执行过程中遇到以上几种情况时，都会生成一个执行上下文并放入函数调用栈中，处于栈顶的上下文执行完毕之后，会自动出栈。

为了更加清晰地理解整个过程，我们可以通过几个实例来了解函数调用栈的执行规则。

3.1 实例 1

```
// demo01.js
var color = 'blue';
```

```
function changeColor() {
    var anotherColor = 'red';

    function swapColors() {
        var tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;
    }

    swapColors();
}

changeColor();
```

我们用 ECStack 来表示处理执行上下文的堆栈。

第一步全局上下文入栈，并一直存于栈底，如图 3-1所示。

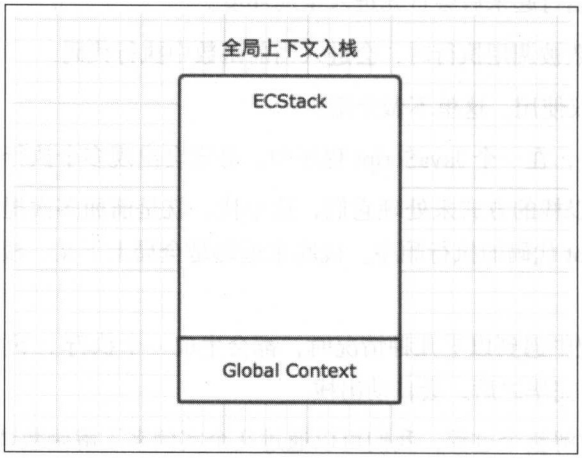


图 3-1　全局上下文入栈

第二步，全局上下文入栈之后，从可执行代码开始执行，直到遇到 changeColor()，这句代码激活了函数 changeColor，从而创建 changeColor 自己的执行上下文，因而此时是 changeColor EC 的上下文入栈，如图 3-2所示。

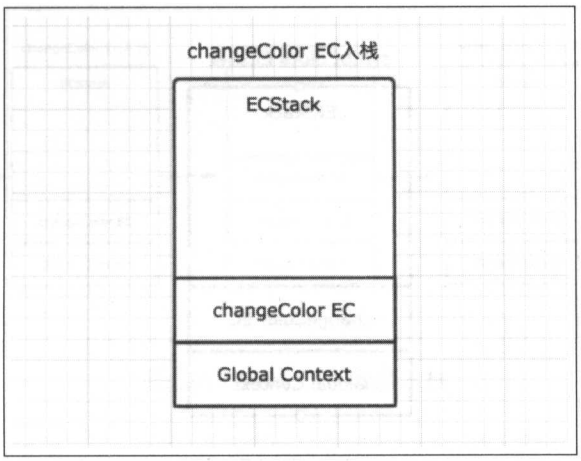


图 3-2 changeColor EC 的上下文入栈

第三步，changeColor EC 的上下文入栈之后，开始执行其中的可执行代码，并在遇到 swapColors() 这句代码之后又激活了 swapColors 的执行上下文。因此第三步就是 swapColors EC 的上下文入栈，如图 3-3所示。

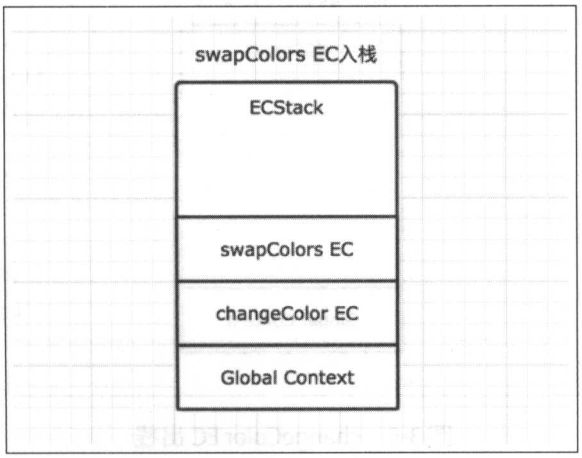


图 3-3 swapColors EC 的上下文入栈

第四步，在 swapColors 的可执行代码中，没有其他能生成执行上下文的情况，因此这段代码顺利执行完毕，swapColors 的上下文从栈中弹出，如图 3-4所示。

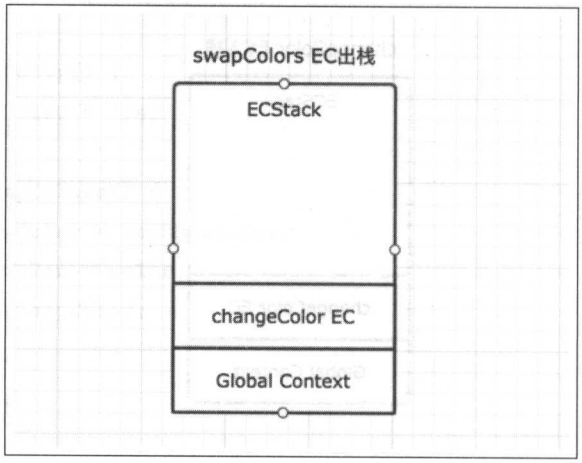


图 3-4 swapColors EC 出栈

第五步，swapColors 的执行上下文弹出之后，继续执行 changeColor 的可执行代码，没有再遇到其他执行上下文，顺利执行完毕后弹出。这样，ECStack 中就只剩下全局上下文了，如图 3-5所示。

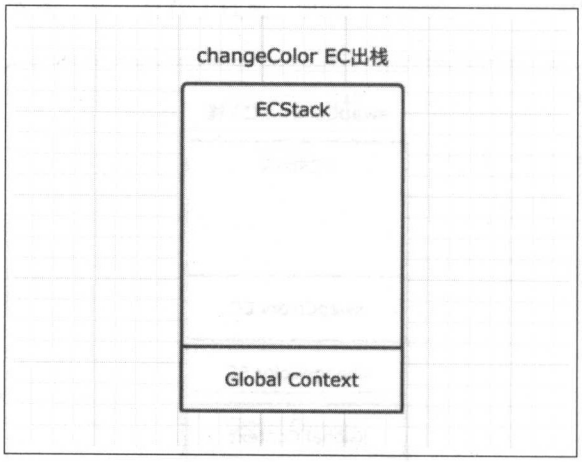


图 3-5 changeColor EC 出栈

最后，全局上下文在浏览器窗口关闭后出栈。

注意：函数执行过程中遇到 return 能直接终止可执行代码的执行，因此会直接将当前上下文弹出栈。

整个过程如图 3-6所示。

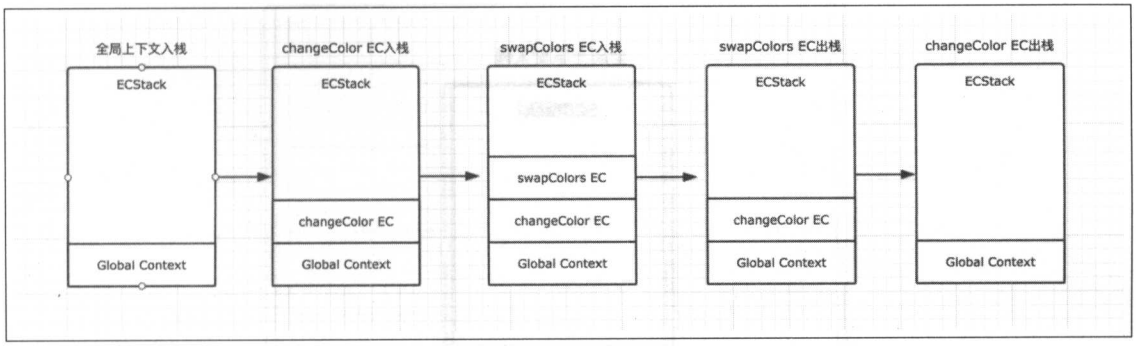


图 3-6 全局上下文出入栈

3.2 实例 2

```
// demo02.js
function f1(){
    var n=999;
    function f2(){
        alert(n);
    }
    return f2;
}
var result=f1();
result(); // 999
```

这是一个简单的闭包例子，整个例子具有一定的迷惑性。

但是我们只需要根据“函数执行时才会创建执行上下文”这一原则来理解，那么这段代码执行时的函数调用栈顺序就会比较清晰了。

第一步，仍然是全局上下文先入栈，如图 3-7所示。

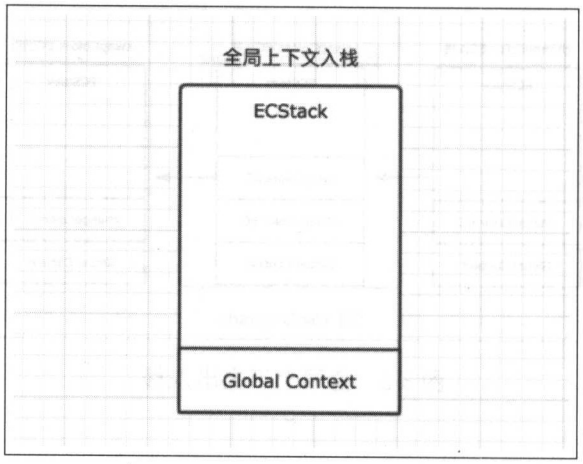


图 3-7 全局上下文入栈

第二步，全局代码在执行过程中，遇到了 f1() 函数，执行 `var result=f1();`，因此 f1 会创建对应的执行上下文并入栈，如图 3-8所示。

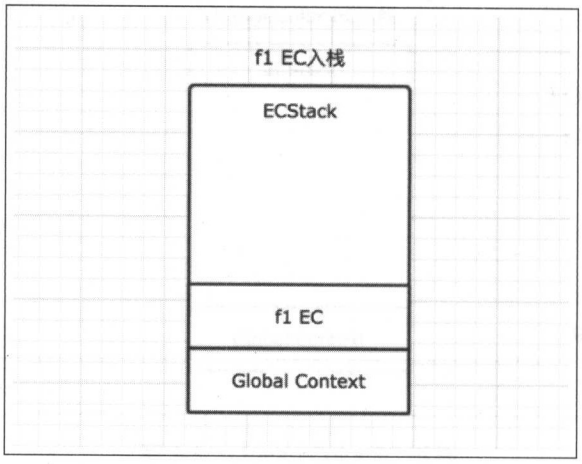


图 3-8 f1 EC 入栈

第三步，在 f1 的可执行代码中，虽然声明了一个函数 f2，但是并没有执行任何函数，因此也就不会产生别的上下文，代码执行结束后，f1 自然会出栈，如图 3-9所示。

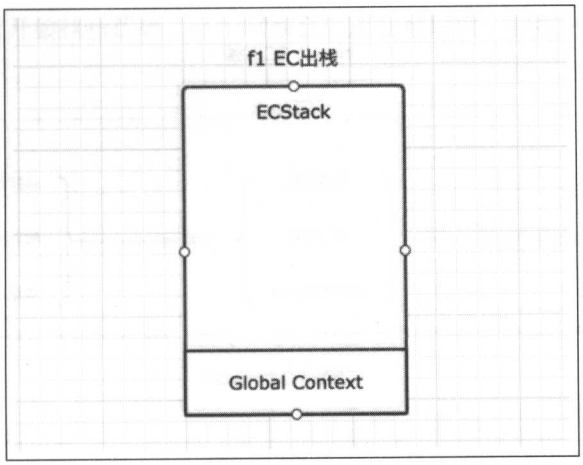


图 3-9 f1 EC 出栈

第四步，f1 出栈之后，继续执行全局上下文的代码，这个时候遇到了 result()，result() 会创建一个新的上下文，因此这个时候 result 的上下文入栈，如图 3-10所示。

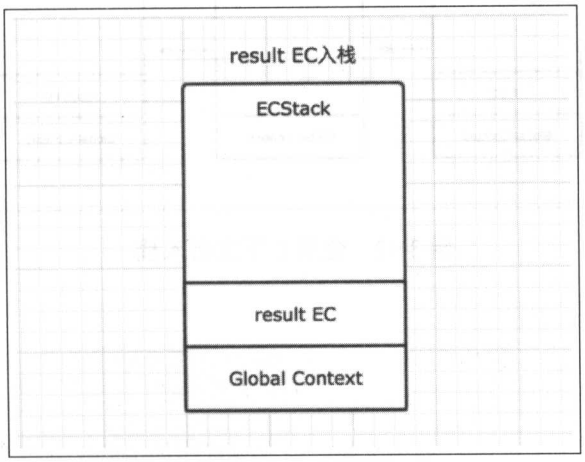


图 3-10 result EC 入栈

第五步，这个 result() 其实就是在 f1 中声明的函数 f2，因此这个时候会执行 f2 中的代码。由于在 f2 中没有产生新的上下文，因此执行完毕后直接出栈，如图 3-11所示。

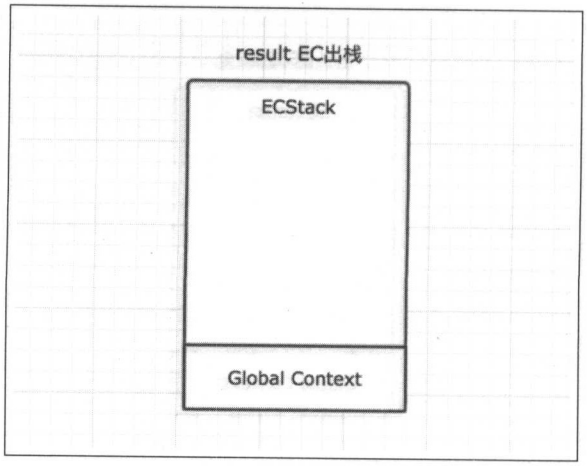


图 3-11　result EC 出栈

完整过程如图 3-12所示。

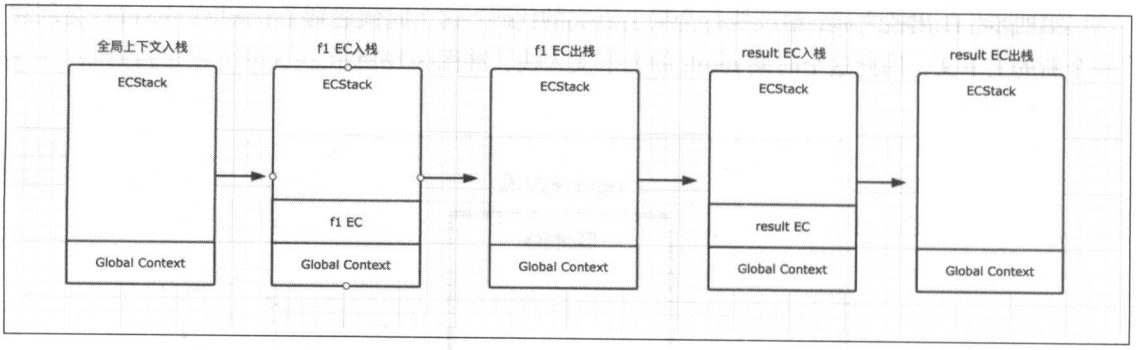


图 3-12　全局上下文出入栈

3.3　生命周期

我们知道，当一个函数调用时，一个新的执行上下文就会被创建。一个执行上下文的生命周期大致可以分为两个阶段：创建阶段和执行阶段。

创建阶段

在这个阶段，执行上下文会分别创建变量对象，确认作用域链，以及确定 this 的指向。

执行阶段

创建阶段之后，就开始执行代码，这个时候会完成变量赋值、函数引用，以及执行其他可执行代码，如图 3-13所示。

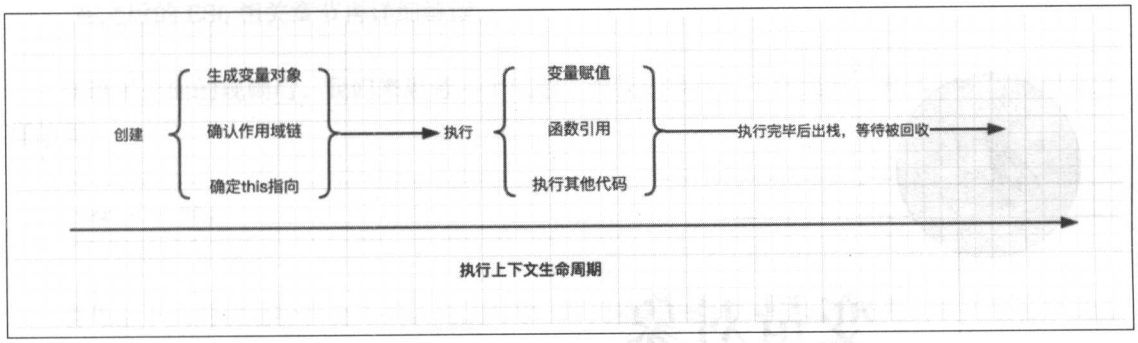
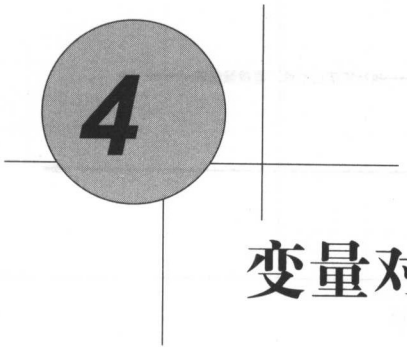


图 3-13 执行上下文生命周期

从执行上下文的生命周期可以看到它的重要性，其中涉及了变量对象、作用域链、this 等许多重要但并不那么容易搞清楚的概念，这些概念有助于我们真正理解 JavaScript 代码的运行机制。



变量对象

在第 2 章内存空间中曾提到过变量对象（Variable Object），我们在 JavaScript 代码中声明的所有变量都保存在变量对象中，除此之外，变量对象中还可能包含以下内容。

- ◎ 函数的所有参数（Firefox 中为参数对象 arguments）。
- ◎ 当前上下文中的所有函数声明（通过 function 声明的函数）。
- ◎ 当前上下文中的所有变量声明（通过 var 声明的变量）。

4.1 创建过程

变量对象的创建，依次经历了以下几个过程。

1. 在 Chrome 浏览器中，变量对象会首先获得函数的参数变量及其值；在 Firefox 浏览器中，是直接将参数对象 arguments 保存在变量对象中。
2. 依次获取当前上下文中所有的函数声明，也就是使用 function 关键字声明的函数。在变量对象中会以函数名建立一个属性，属性值为指向该函数所在的内存地址引用。如果函数名的属性已经存在，那么该属性的值会被新的引用覆盖。
3. 依次获取当前上下文中的变量声明，也就是使用 var 关键字声明的变量。每找到一个变量声明，就在变量对象中就以变量名建立一个属性，属性值为 undefined。如果该变量名的属性已经存在，为了防止同名的函数被修改为 undefined，则会直接跳过，原属性值不会被修改。

ES6 支持新的变量声明方式 let/const，规则与 var 完全不同，它们是在上下文的执行阶段开始执行的，避免了变量提升带来的一系列问题，因此这里暂时先不详细介绍，在之后的 ES6 相关章节再详细解读。

知道了上面的规则后，我们来思考一个问题，当我们执行以下代码时，具体的执行过程是怎样的呢？

```
var a = 30;
```

首先上下文的创建阶段会先确认变量对象，而变量对象的创建过程则是先获取变量名并赋值为 undefined，因此第一步是：

```
var a = undefined;
```

上下文的创建阶段完毕后，开始进入执行阶段，在执行阶段需要完成变量赋值的工作，因此第二步是：

```
a = 30;
```

需要注意的是，这两步分别是在上下文的创建阶段与执行阶段完成的。因此 var a = undefined 这一步其实是提前到了比较早的地方去执行，下面通过一个简单的例子来证明。

```
console.log(a);     // a 这个时候输出结果为undefined
var a = 30;
```

结合之前的理解，这个例子的实际执行顺序为：

```
// 创建阶段
var a = undefined;

// 执行阶段
console.log(a);
a = 30;
```

这种现象我们称之为变量提升（Hoisting）。

从上面的规则中我们还可以看出，在变量对象的创建过程中，函数声明的执行优先级会比变量声明的优先级更高一点，而且同名的函数会覆盖函数与变量，但是同名的变量并不会覆盖函数。

但是在上下文的执行阶段，同名的函数会被变量重新赋值。

```
var a = 20;
function fn() { console.log('fn') };
function fn() { console.log('cover fn.') };
function a() { console.log('cover a.') };

console.log(a);
fn();

var fn = 'I want cover function named fn.';
console.log(fn);

// 20
// cover fn.
// I want cover function named fn.
```

上面例子的执行顺序其实为：

```
// 创建阶段
function fn() { console.log('fn') };
function fn() { console.log('cover fn.') };
function a() { console.log('cover a.') };
var a = undefined;
var fn = undefined;

// 执行阶段
a = 20;
console.log(a);
fn();
fn = 'I want cover function named fn.';
console.log(fn);
```

根据输出结果可以证明，在创建阶段，后创建的函数 fn 覆盖了前面创建的函数 fn，但是变量 fn 并没有覆盖函数 fn。而在执行阶段，a 与 fn 的重新赋值导致它们发生了变化。

4.2 实例分析

为了更加深刻地理解变量对象，下面结合一些简单的例子来进行探讨。

```
// demo01
function test() {
    console.log(a);
    console.log(foo());

    var a = 1;
    function foo() {
        return 2;
    }
}

test();
```

运行 test 函数时，对应的上下文开始创建，可以用如下形式来表达这个过程。

```
// 创建过程
testEC = {
    VO: {},          // 变量对象
    scopeChain: [], // 作用域链
    this: {}
}

// 这里暂时不深入分析作用域链与this，后续章节会详细讲解

// VO为Variable Object的缩写，即变量对象
VO = {
    arguments: {...},
    foo: <foo reference>,
```

```
    a: undefined
}
```

在函数调用栈中，如果当前执行上下文处于函数调用栈的栈顶，则意味着当前上下文处于激活状态，此时变量对象称之为活动对象（AO，Activation Object）。活动对象中包含变量对象的所有属性，并且此时所有的属性都已经完成了赋值，除此之外，活动对象还包含了 this 的指向。

```
// 执行阶段
VO -> AO
AO = {
    arguments: {},
    foo: <foo reference>,
    a: 1,
    this: Window
}
```

我们可以通过断点调试的方式在 Chrome 开发者工具中查看这一过程。如何使用断点调试会在后续章节详细讲解。

因此上面的例子实际执行顺序如下。

```
function test() {
    function foo() {
        return 2;
    }
    var a = undefined;
    console.log(a);
    console.log(foo());
    a = 1;
}

test();
```

下面再来看一个例子，巩固一下。

```
// demo2
function test() {
    console.log(foo);
    console.log(bar);

    var foo = 'Hello';
    console.log(foo);
    var bar = function () {
        return 'world';
    }

    function foo() {
        return 'hello';
    }
}

test();
```

```
// 创建阶段
VO = {
    arguments: {...},
    foo: <foo reference>,
    bar: undefined
}
// 这里有一个需要注意的地方，即var声明的变量在遇到同名的属性时，会跳过而不是覆盖

// 执行阶段
VO -> AO
VO = {
    arguments: {...},
    foo: 'Hello',
    bar: <bar reference>,
    this: Window
}
```

需要结合上面的知识点，仔细对比这个例子中变量对象的变化过程。如果你已经理解了，则说明变量对象相关的内容你已经掌握了。

最后留下一个简单的思考题：下面的例子中，函数 bar 的变量对象中是否包含了变量 a?

```
function foo() {
    var a = 20;

    function bar() {
        a = 30;
        console.log(a);
    }

    bar();
}

foo();
```

4.3 全局上下文的变量对象

以浏览器为例，全局对象为 window 对象。

全局上下文的变量对象有一个特殊的地方，即它的变量对象就是 window 对象，而且全局上下文的变量对象不能变成活动对象。

```
// 以浏览器为例，全局对象为window
// 全局上下文
windowEC = {
    VO: window,
    scopeChain: {},
    this: window
}
```

除此之外，全局上下文的生命周期与程序的生命周期一致，只要程序运行不结束（比如关掉浏览器窗口），全局上下文就会一直存在，其他所有的上下文环境都能直接访问全局上下文的属性。

5 作用域与作用域链

在 JavaScript 中，作用域是用来规定变量与函数可访问范围的一套规则。

5.1 作用域

最常见的作用域有两种，分别是全局作用域与函数作用域。

5.1.1 全局作用域

全局作用域中声明的变量与函数可以在代码的任何地方被访问。

一般来说，以下三种情形拥有全局作用域。

1. 全局对象下拥有的属性与方法。

```
window.name
window.location
window.top
...
```

2. 在最外层声明的变量与方法。

我们知道，全局上下文中的变量对象实际上就是 window 对象本身，因此在全局上下文中声明的变量与方法其实就变成了 window 的属性与方法，所以它们也拥有全局作用域。

```
var foo = function() {}
var str = 'out variable';
var arr = [1, 2, 3];
function bar() {}

...
```

3. 在非严格模式下，函数作用域中未定义但直接复制的变量与方法。

   在非严格模式下，这样的变量与方法会自动变成全局对象 window 的属性，因此它们也有用全局作用域。

```
function foo() {
  bar = 20;
}

function fn() {
  foo();
  return bar + 30;
}

fn(); // 50
```

在实践中，无论是从避免多人协作带来的冲突的角度考虑，还是从性能优化的角度考虑，我们都要尽可能少地自定义全局变量与方法。

5.1.2 函数作用域

函数作用域中声明的变量与方法，只能被下层子作用域访问，而不能被其他不相干的作用域访问。

```
function foo() {
    var a = 20;
    var b = 30;
}
foo();
```

```
function bar() {
    return a + b;
}

bar(); // 因为作用域的限制，bar中无法访问到变量a和b，因此执行报错
```

```
function foo() {
    var a = 20;
    var b = 30;

    function bar() {
        return a + b;
    }
    return bar();
}
foo();  // 50  bar中的作用域为foo的自作用域，因此能访问到变量a和b
```

在 ES6 以前，ECMAScript 没有块级作用域，因此使用时需要特别注意，一定是在函数环境中才能生成新的作用域，下面的情况则不会有作用域的限制。

```
var arr = [1, 2, 3, 4, 5];

for(var i = 0; i < arr.length; i++) {
    console.log('do something by ', i);
}

console.log(i); // i == 5
```

因为没有块级作用域，因此单独的‘{}’并不会产生新的作用域。这个时候 i 的值会被保留下来，在 for 循环结束后仍然能够访问。

5.1.3 模拟块级作用域

如果没有块级作用域则会给我们的开发带来一些困扰。例如，上面 for 循环的例子中，i 值在作用域中仍然可以被访问，那么这个值就会对作用域中其他同名的变量造成干扰，因此我们需要

想办法模拟块级作用域。一个函数可以生成一个作用域，因此这个时候，可以利用函数来达到我们的目的。

```
var arr = [1, 2, 3, 4, 5];

(function() {
    for(var i = 0; i < arr.length; i++) {
        console.log('do something by ', i);
    }
})();


console.log(i); // i is not defined
```

这种方式叫作**函数自执行**。

通过这种方式，我们就可以限定变量 i 值仅仅只在 for 循环中生效，而不会对其他代码造成干扰。

函数自执行时声明一个匿名函数并且立即执行，这种方式大体有如下几种写法。

```
(function() {
    ...
})(); // 最常用

+function() {
    ...
}();

-function() {
    ...
}();

!function() {
    ...
}();
```

当我们使用 ECMAScript5 时，往往通过函数自执行的方式来实现模块化。而模块化是实际开发中需要重点掌握的开发思维，在后续的章节中会介绍相关的思路。

5.2 作用域链

作用域链（Scope Chain）是由当前执行环境与上层执行环境的一系列变量对象组成的，它保证了当前执行环境对符合访问权限的变量和函数的有序访问。

如果是第一次接触作用域链的概念，则可能对上面这句话的理解有一点困难，下面就结合一个例子，以及对应的图示来说明。

```
var a = 20;

function test() {
    var b = a + 10;

    function innerTest() {
        var c = 10;
        return b + c;
    }

    return innerTest();
}

test();
```

在上面的例子中，先后创建了全局函数 test 和函数 innerTest 的执行上下文。假设它们的变量对象分别为 VO(global)、VO(test) 和 VO(innerTest)，那么 innerTest 的作用域链则同时包含了这三个变量对象。

所以 innerTest 的执行上下文可表示如下。

```
innerTestEC = {
    VO: {...},  // 变量对象
    scopeChain: [VO(innerTest), VO(test), VO(global)], // 作用域链
    this: {}
```

```
}
```

可以用一个数组来表示作用域链的有序性。数组的第一项 scopeChain[0] 为作用域链的最前端，而数组的最后一项则为作用域链的最末端。所有作用域链的最末端都是全局变量对象。

很对人会用父子关系或者包含关系来理解当前作用域与上层作用域之间的关系，但其实这种说法并不准确，以当前上下文的变量对象为起点，以全局变量对象为终点的单方向通道这样描述会更加贴切，如图 5-1所示。

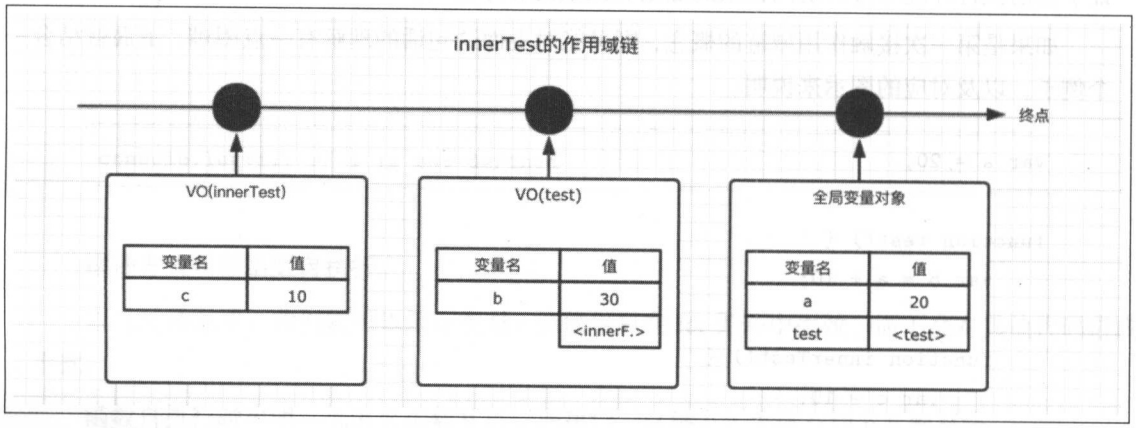
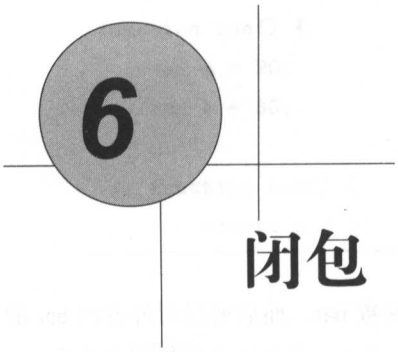


图 5-1　innerTest 的作用域链

理解作用域链至关重要，但是更多的知识需要结合闭包来理解，因此，在学习闭包的章节中会扩展更多的内容。



闭包

闭包（Closures）是学习过程中的一个瓶颈。大家在初学的时候对于闭包都是望而生畏的，面试必问，可理解起来又十分费劲，就连许多有多年 JavaScript 开发经验的人也不一定对闭包真的理解透彻，更别说向别人讲解出来了。

但是大家不要怕，本章将会抽丝剥茧，深入浅出，帮助大家彻底理解闭包。

6.1 概念

闭包是一种特殊的对象。

它由两部分组成–执行上下文（代号 A），以及在该执行上下文中创建的函数（代号 B）。

当 B 执行时，如果访问了 A 中变量对象中的值，那么闭包就会产生。

许多书籍、文章里都以函数 B 的名字代指这里生成的闭包。而在 Chrome 中，则以执行上下文 A 的函数名代指闭包。

我们只需知道，一个闭包对象，由 A、B 共同组成，在以后的篇幅中，都将以 Chrome 的标准来称呼。

```
// demo01
function foo() {
    var a = 20;
    var b = 30;
```

```
    function bar() {
        return a + b;
    }

    return bar;
}

var bar = foo();
bar();
```

在上面的例子中，首先执行上下文 foo，在 foo 中定义了函数 bar，而后通过对外返回 bar 的方式让 bar 得以执行。当 bar 执行时，访问了 foo 内部的变量 a 和 b。因此这个时候闭包产生。

在 Chrome 中通过断点调试的方式可以逐步分析该过程，从图 6-1中可以看出，此时闭包产生，用 foo 代指。

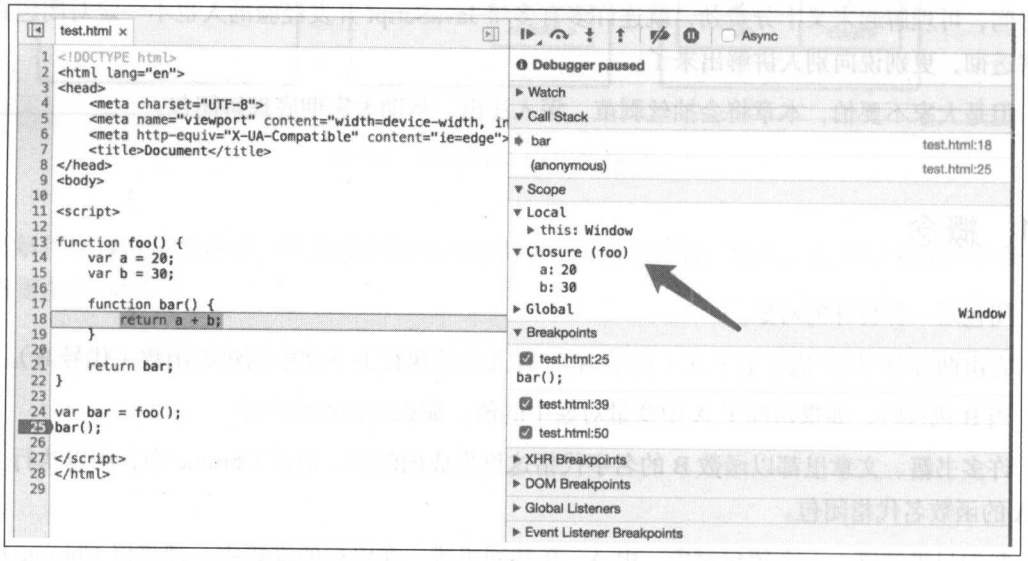


图 6-1 Chrome 断点调试

在图6-1中，箭头所指的正是闭包。其中 Call Stack 为当前的函数调用栈，Scope 为当前正在被执行函数的作用域链，Local 为当前活动对象。

断点调试的方式在第 7 章中讲解。

在学习了闭包的基础概念后，下面就来验证一下你是否真的理解了。现在思考一个小问题，把 demo01 的代码稍作调整后，是否形成了闭包呢？

```
// demo02
function foo() {
    var a = 20;
    var b = 30;

    function bar() {
        return a + b;
    }

    bar();
}

foo();
```

仍然是 foo 中定义的 bar 函数在执行时访问了 foo 中的变量，因此这个时候仍然会形成闭包，如图 6-2所示。

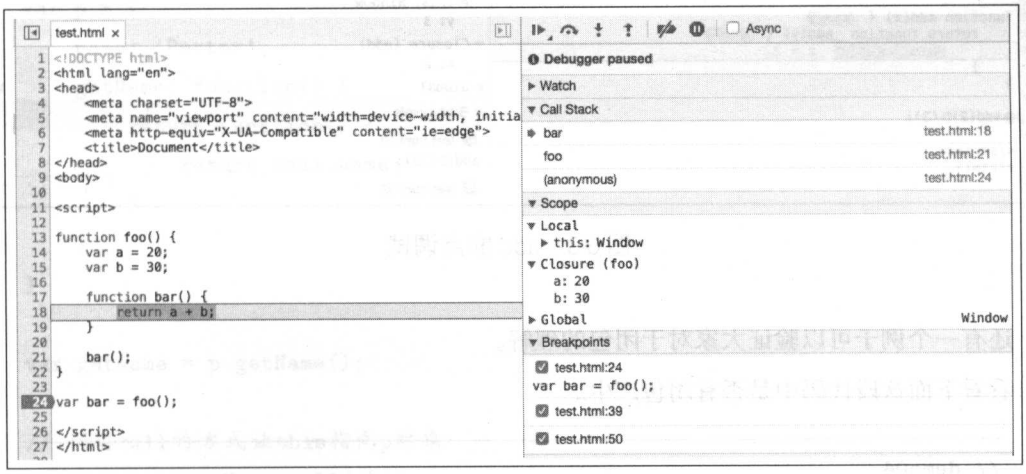


图 6-2 foo 断点调试

再来看一个非常有意思的例子。

```
// demo03
function add(x) {
    return function _add(y) {
        return x + y;
    }
}

add(2)(3); // 5
```

这个例子有闭包产生吗？

当然有。当内部函数_add 被调用执行时，访问了 add 函数变量对象中的 x，这个时候，闭包就会产生，如图 6-3所示。一定要记住，函数参数中的变量传递给函数之后也会加到变量对象中。

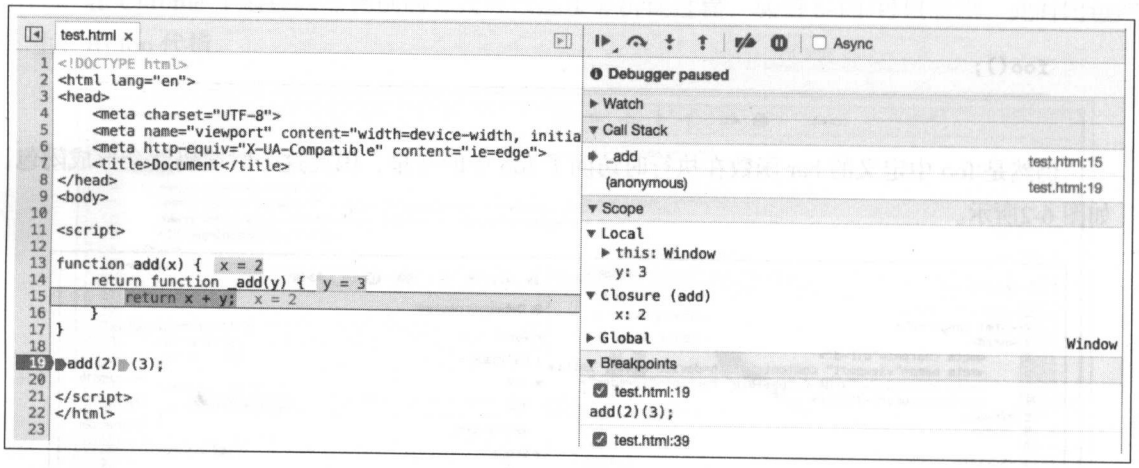


图 6-3　add 断点调试

还有一个例子可以验证大家对于闭包的理解。

看看下面这段代码中是否有闭包产生。

```
// demo04
var name = "window";

var p = {
    name: 'Perter',
```

```
    getName: function() {
        return function() {
            return this.name;
        }
    }
}

var getName = p.getName();
var _name = getName();
console.log(_name);
```

getName 在执行时，它的 this 其实指向的是 window 对象，而这个时候并没有形成闭包的环境，因此这个例子没有闭包。

那么如果按照下面的方式进行改动呢?

```
// 改动一
// demo05
var name = "window";

var p = {
    name: 'Perter',
    getName: function() {
        return function() {
            return this.name;
        }
    }
}

var getName = p.getName();

// 利用call的方式让this指向p对象
var _name = getName.call(p);
console.log(_name);
```

```
// 改动二
```

```
// demo06
var name = "window";

var p = {
    name: 'Perter',
    getName: function() {

        // 利用变量保存的方式保证其访问的是p对象
        var self = this;
        return function() {
            return self.name;
        }
    }
}

var getName = p.getName();
var _name = getName();
console.log(_name);
```

分别利用 call 与变量保存的方式保证 this 指向的都为 p 对象，那么它们哪一个产生了闭包，哪一个没有产生闭包呢？这里就留一个思考题，大家可以根据上面的概念进行分析。为了验证自己的分析是否正确，大家可以在学习后续章节后在 Chrome 开发者工具中查看闭包。

后续章节会详细讲解 this 的指向问题。

6.2 闭包与垃圾回收机制

前面我们学习了垃圾回收机制，知道当一个值失去引用之后就会被标记，然后被垃圾回收机制回收并释放空间。

我们知道，当一个函数的执行上下文运行完毕之后，内部的所有内容都会失去引用而被垃圾回收机制回收。

我们还知道，闭包的本质就是在函数外部保持了内部变量的引用，因此闭包会阻止垃圾回收机制进行回收。

下面就用一个例子来证明这一点。

```
function f1() {
    var n = 999;
    nAdd = function() {
        n += 1;
    }

    return function f2() {
        console.log(n);
    }
}

var result = f1();
result(); // 999
nAdd();
result(); // 1000
```

从上面的例子可以看出，因为 nAdd、f2 都访问了 f1 中的 n，因此它们都与 f1 形成了闭包。这个时候变量 *n* 的引用被保留了下来。因为 f2(result) 与 nAdd 执行时都访问了 n，而 nAdd 每运行一次就会将 n 加 1。所以上例的执行结果非常符合我们的认知。

认识到闭包中保存的内容不会被释放之后，我们在使用闭包时就要保持足够的警惕性。如果滥用闭包，很可能会因为内存的原因导致程序性能过差。

6.3 闭包与作用域链

先花几秒钟时间来思考一个小问题，闭包会导致函数的作用域链发生变化吗？

我们结合下面的例子来分析一下。

```
var fn = null;
function foo() {
    var a = 2;
    function innnerFoo() {
        console.log(a);
    }
    fn = innnerFoo; // 将innnerFoo的引用赋值给全局变量中的fn
```

```
}

function bar() {
    fn(); // 此处保留innerFoo的引用
}

foo();
bar(); // 2
```

在这个例子中，foo 内部的 innerFoo 访问了 foo 的变量 a。因此当 innerFoo 执行时会有闭包产生，这是一个比较简单的闭包例子。不一样的地方在于全局变量 fn。fn 在 foo 内部获取了 innerFoo 的引用，并在 bar 中执行。

那么 innerFoo 的作用域链会是怎样的呢？我们先利用断点调试的方式在浏览器中观察一下，结果如下图 6-4所示。

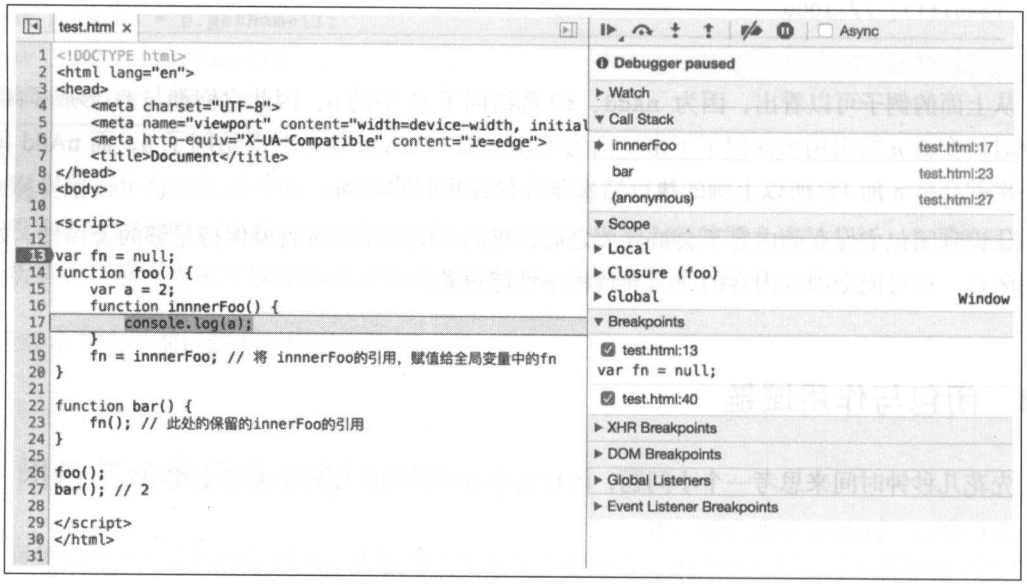


图 6-4　innerFoo 断点调试

在这里需要特别注意的地方是函数调用栈（图6-4右侧的 Call Stack）与作用域链（Scope）的区别。

因为函数调用栈其实是在代码执行时才确定的，而作用域规则在代码编译阶段就已经确定，虽然作用域链是在代码执行时才生成的，但是它的规则并不会在执行时发生改变。

所以这里闭包的存在并不会导致作用域链发生变化。

6.4 在 Chrome 开发者工具中观察函数调用栈、作用域链与闭包

前端发展迅猛，知识也会过时。

也许前面我们学的某个知识点，在未来的某个节点就会悄然发生变化，因此我们需要掌握一些技能，以确保知识发生变化时我们能够快速地应对，确保自己所学到的知识没有发生偏差。

而这里将要学习的一个非常重要的技能就叫作**断点调试**。

在 Chrome 开发者工具中，通过断点调试，我们可以非常方便地一步步观察 JavaScript 代码在执行过程中的细节变化。我们能够直观地感知函数调用栈、变量对象、作用域链、闭包、this 等关键信息的变化过程。因此断点调试对于快速定位代码错误，了解代码执行过程有着非常重要的作用，这也是前端开发中必不可少的一个高级技能。

1. 基础概念回顾

函数在被调用执行时，会创建一个当前函数的执行上下文。在该执行上下文的创建阶段，变量对象、作用域链、闭包、this 等会分别确认。而一个程序中一般来说会有多个函数执行，因此执行引擎会使用函数调用栈来管理这些函数的执行顺序。函数调用栈的执行顺序与栈数据结构一致。

2. 认识断点调试工具

尽量在最新版本的 Chrome 浏览器中（不保证老版本的 Chrome 浏览器与笔者的一致），调出 Chrome 浏览器的开发者工具，并根据如下顺序打开界面。

单击浏览器右上角竖着的三点 → 更多工具 → 开发者工具 → Sources。

如图 6-5所示，为了让大家清晰地明白我们需要重点关注的地方在哪里，这里用箭头把它们标注了出来。

图 6-5　断点调试工具

左侧这个箭头指向的区域表示代码的行数，当我们单击某一行时，就可以在该行设置一个断点。

右侧第一个箭头指向的区域有一排图标。我们可以通过这排图标来控制函数的执行进程。从左到右它们依次是：

◎ **resume/pause script execution**
恢复/暂停脚本执行

◎ **step over next function call**
跨过。实际表示是在未遇到函数时，执行下一步。遇到函数时，不进入函数直接执行下一步。

◎ step into next function call
跨入。实际表示的是未遇到函数时，执行下一步。遇到函数时，进入函数执行上下文。

◎ step out of current function
跳出当前函数。

◎ deactivate breakpoints
停用断点。

◎ don't pause on exceptions
不暂停异常捕获

其中跨过、跨入、跳出是调试过程中用得最多的三个操作。

图6-5中右侧第二个箭头所指的区域为 Call Stack（当前所处于的执行上下文的函数调用栈）。

图6-5中右侧第三个箭头所指向的区域为 Scope，即当前函数的作用域链。其中，Local 表示当前正在执行的活动对象，Closure 表示闭包。

因此我们可以借助此处作用域链的展示，观察到谁才是闭包，对于闭包的深入了解有非常大的帮助作用。

在显示代码行数的区域单击，即可在代码对应行数所在的地方设置一个断点。设置断点后刷新页面，代码会执行到断点位置处暂停，这时我们就可以通过上面介绍过的几个图标操作一步步调试我们的代码了。

Chrome 中，在单独的变量声明（没有赋值操作）与函数声明的那一行，无法设置断点。

3. 实例

接下来为了进一步掌握断点调试，我们借助一些实例，通过断点调试来观察这些代码的执行过程。这里主要以闭包的例子来展示。

```
// demo01
var fn;
function foo() {
    var a = 2;
    function baz() {
        console.log( a );
    }
    fn = baz;
}
function bar() {
    fn();
}

foo();
bar(); // 2
```

可以先思考这个例子中闭包是如何产生的，然后通过调试来验证自己的想法是否正确。

很显然，fn 是对 foo 或者 foo 内部函数 baz 的引用。因此当 fn 执行时，其实就是 baz 在执行。而 baz 在执行时访问了 foo 中的变量，因此闭包产生。在 Chrome 中，用 foo 来指代生成的闭包。

第一步，设置断点，然后刷新页面，如图 6-6所示。

图 6-6　设置断点

经过简单地分析可以看出，代码是从 foo() 这一行开始执行的，因此在这里设置一个断点。

第二步，单击图6-6箭头所指的图标（step into），该按钮的作用是根据代码的执行顺序，一步步向下执行，当遇到函数时，跳入函数执行。多次点击，直到 baz 函数执行，如图 6-7所示。

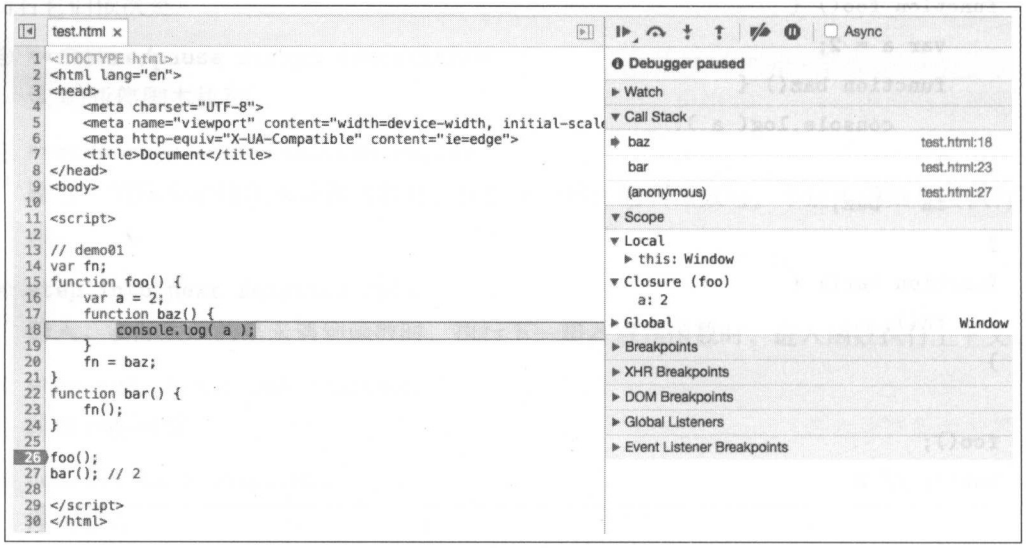


图 6-7　断点顺序执行

我们应该关注代码执行过程中各个变量的变化情况，以及 Call Stack 与 Scope 的变化。当执行到 baz 函数时，Call Stack 与 Scope 如图6-7所示。如果大家对前面的知识有足够地理解，就应

该明白函数调用栈的不同就应该如此，而作用域链则不会因为闭包发生变化。

而我们还需要关注的一个点在于 Chrome 对作用域链所做的一个优化。

在《JavaScript 高级编程 3》中对于闭包有这样一段描述，“闭包所保存的是整个变量对象，而不是某个特殊的变量”。

从上面的例子中我们可以明确地知道，在函数 foo 的变量对象中，应该保存了一个变量 a 与一个函数 baz。但是从图6-7中可看出，Closure(foo) 并没有函数 baz，仅仅只有变量 a。

其实这是 Chrome 新版本对闭包与作用域链所做的一个优化，它仅仅只保留了会被访问到的变量。我们可以用下面的例子来进一步证明这一点。

```
// demo02
function foo() {
    var x = 20;
    var y = 10;
    function child() {
        var m = 5;

        return function add() {
            var z = 'this is add';
            return x + y;
        }
    }

    return child();
}

foo()();
```

在上面这个例子中，我们通过前面所学到的知识来思考一下函数 add 在执行时它的作用域链应该是怎样的？

代码如下所示。

```
addEC = {
    scopeChain: [AO(add), VO(child), VO(foo), VO(Global)]
}
```

可是 Chrome 中的表现是怎样的呢？一起来看一下，如图 6-8所示。

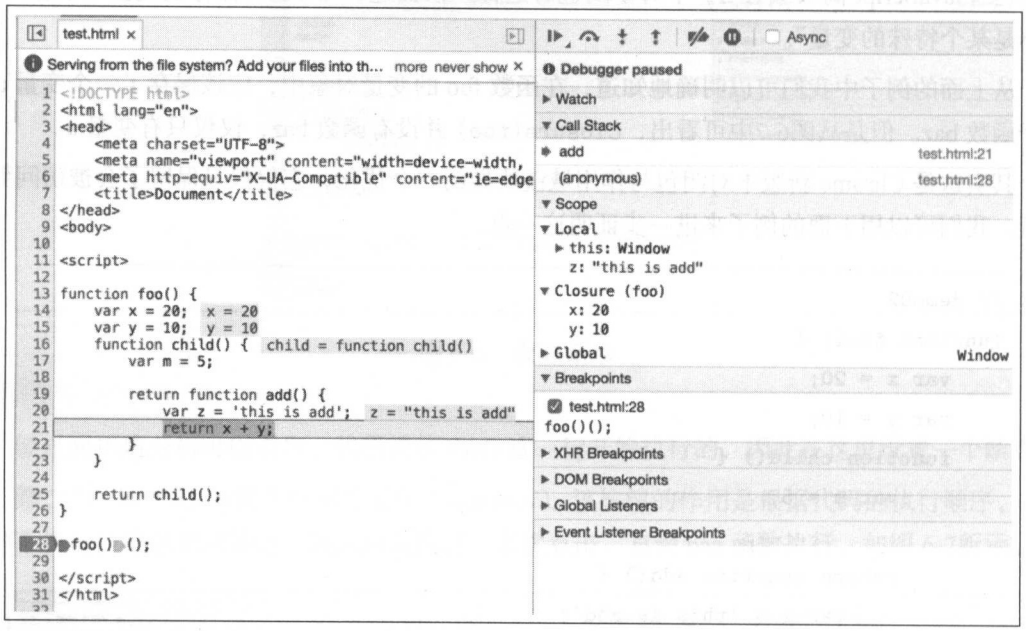


图 6-8　Chrome 断点调试

咦！中间的 VO(child) 直接被省略了，这正是 Chrome 的优化。因为 child 函数中的变量 m 与函数 add 在 add 的执行上下文中并没有被访问，因此就没有保留在内存中的必要。

同样的例子在 Firefox 中调试，它已经明确指出了那些已经被优化掉的变量与函数，如图6-9所示。但是通过对比可以发现，Chrome 的优化其实更进一步。

在 Firefox 中，仅仅只是对没有被访问到的变量对象进行了优化，而访问到的变量对象中，未被访问到的变量并没有被优化。而 Chrome 中，对未被访问到的变量同样进行了优化，如图6-9所示。

图 6-9 Firefox 断点调试

这里需要注意的是 arguments 对象，在 Chrome 中，是将参数展开显示在变量对象中的，而在 Firefox 中，则直接保存的是一个 arguments 对象。具体的可以通过之前的例子与图示区分。

下面再来看一个特殊的例子。

```
// demo03
function foo() {
    var a = 10;

    function fn1() {
```

```
        console.log(a);
    }

    function fn2() {
        var b = 10;
        console.log(b);
    }

    fn2();
}

foo();
```

花几秒钟思考一下，当函数 fn2 执行时，这个例子中有没有闭包产生？如图 6-10所示。

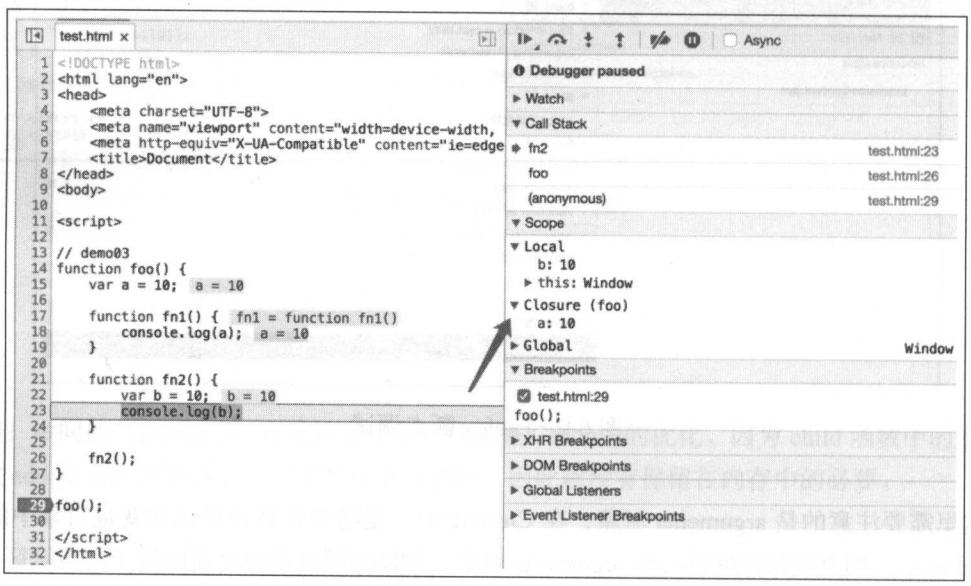


图 6-10　fn2 产生闭包

从图6-10中可以看出，确实产生了闭包。

在最新的 MDN 中，对闭包是这样定义的（结合上例）：“闭包是指这样的作用域（foo），它包含了一个函数（fn1），这个函数（fn1）可以调用被这个作用域所封闭的变量（a）、函数或者闭包等内容。通常我们通过闭包所对应的函数来获得对闭包的访问。”

这里的定义其实包含了这种特殊情况，fn1 与 foo 共同产生了闭包，但是 fn1 并没有任何地方可以调用它。在实际开发中我们几乎不会这样使用，如果在面试中被问及，知道如何回答即可。

6.5 应用闭包

通过前面对闭包知识的学习，现在我们已经能够比较准确地识别是否产生了闭包。接下来需要培养的能力，就是如何利用闭包解决实际问题。与前面的知识不同，运用闭包并不是一蹴而就的事情，我们需要给自己更多的时间，通过实践把闭包知识掌握得更好。

下面就通过三个案例，运用闭包来解决实际问题。

6.5.1 循环、setTimeout 与闭包

在面试题中常常会遇到一个与循环、闭包有关的问题，如下所示。

```
// 利用闭包的知识，修改这段代码，让代码的执行结果为隔秒输出1，2，3，4，5
for (var i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log(i);
    }, i*1000 );
}
```

首先来分析一下如果直接运行这个例子会输出什么结果。

前面我们已经知道，for 循环的大括号并不会形成自己的作用域，因此这个时候肯定是没有闭包产生的，而 i 值作为全局的一个变量，会随着循环的过程递增。因此循环结束之后，i 变成 6。

而每一个循环中，setTimeout 的第二个参数访问的都是当前的 i 值，因此第二个 i 值分别是 1，2，3，4，5。而第一个参数 timer 函数中虽然访问的是同一个 i 值，但是由于延迟的原因，当 timer 函数被 setTimeout 运行时，循环已经结束，即 i 已经变成了 6。

因此这段代码直接运行的结果是隔秒输出 6。

而我们想要的是隔秒输出 1，2，3，4，5，因此需要借助闭包的特性，将每一个 i 值都用一个闭包保存起来。每一轮循环，都把当前的 i 值保存在一个闭包中，当 setTimeout 中定义的操作执行时，访问对应的闭包即可。

这个时候我们回想一下闭包形成的条件，简单来说，就是一个函数里定义一个子函数，子函数内访问函数的变量对象。因此我们只需创建出这样的环境即可。

修改代码如下：

```
for (var i=1; i<=5; i++) {
    (function(i) {
        setTimeout( function timer() {
            console.log(i);
        }, i*1000 );
    })(i);
}
```

定义一个匿名函数，称作 A，并将其当作闭包的环境。而 timer 函数则作为 A 的内部函数，当 A 执行时，只需访问 A 的变量对象即可。因此将 i 值作为参数传入，这样也就满足了闭包的条件，并将 i 值保存在了 A 中。

同样的道理，也可以在 timer 函数这里做文章，如下：

```
for (var i=1; i<=5; i++) {
    setTimeout((function(i) {
        return function timer() {
            console.log(i);
        }
    })(i), i*1000 );
}
```

6.5.2 单例模式与闭包

在 JavaScript 中有许多解决特定问题的编码思维（设计模式），例如工厂模式、订阅通知模式、装饰模式、单例模式等。其中，单例模式是实践中最常用的模式之一，而它的实现，与闭包息息相关。

所谓单例模式，就是只有一个实例。

1. 最简单的单例模式

对象字面量的方法就是最简单的单例模式，我们可以将属性与方法依次放在字面量里。

```
var per = {
    name: 'Jake',
    age: 20
    getName: function() {
        return this.name;
    },
    getAge: function() {
        return this.age;
    }
}
```

但是这样的单例模式有一个严重的问题，即它的属性可以被外部修改。因此在许多场景中，这样的写法并不符合我们的需求，我们期望对象能够有自己的私有方法与属性。

2. 有私有方法/属性的单例模式

通过前面所学的知识我们很容易就能想到，想要一个对象拥有私有的方法属性，那么只需创建一个单独的作用域将对象与外界隔离起来就行了。这里我们借助匿名函数自执行的方式即可。

```
var per = (function() {
    var name = 'Jake';
    var age = 20;

    return {
        getName: function() {
            return name;
        },

        getAge: function() {
            return age;
        }
    }
```

```
})();

// 访问私有变量
per.getName();
```

私有变量的好处在于，外界对于私有变量能够进行什么样的操作是可以控制的。我们可以提供一个 getName 方法让外界可以访问名字，也可以额外提供一个 setName 方法，来修改它的名字。对外提供什么样的能力，完全由我们自己决定。

不知不觉中，我们已经利用闭包来解决问题了。匿名函数（称为 A）中，当 name 被 getName 访问时，闭包就已经产生，因此 A 中的两个属性都会被保留下来。

这个时候离我们最终的模块化思维已经比较接近了，在模块化的开发中，每一个模块都是一个与此类似的单例模式。如果在后面学习模块化开发时遇到了困难，不妨回到这里来理顺一下思路。

3. 调用时才初始化的单例模式

有的时候（使用频次较少）我们希望自己的实例仅仅只是在调用的时候才被初始化，而不是如上面两个例子那样，即使没有调用 per，per 的实例在函数自执行的时候就返回了。

那么我们就需要在上面例子的基础上做一个简单的判断。

```
var per = (function() {
    // 定义一个变量，用来保存实例
    var instance = null;
    var name = 'Jake';
    var age = 20;

    // 初始化方法
    function initial() {
        return {
            getName: function() { return name; },
            getAge: function() { return name; }
        }
    }

    return {
```

```
        getInstance: function() {
            if (!instance) {
                instance = initial();
            }
            return instance;
        }
    }
})();

// 只在使用时获得实例
var p1 = per.getInstance();
var p2 = per.getInstance();

console.log(p1 === p2);  // true
```

在这个例子中，我们在匿名函数中定义了一个 instance 变量用来保存实例。在 getInstance 方法中判断了是否对它进行重新赋值。由于这个判断的存在，因此变量 instance 仅仅只在第一次调用 getInstance 方法时赋值了。所以这种写法完美符合了单例模式的思路。

这个例子中利用闭包的思路其实与上个例子一样，因此不多做分析，留给大家自己思考以进一步熟悉闭包。

6.5.3 模块化与闭包

如果想在所有的地方都能访问同一个变量，那么应该怎么办呢？在实践中这种场景很多，比如全局的状态管理。

但前面我们介绍过，在实际开发中，不要轻易使用全局变量，那又该怎么办呢？模块化的思维能够帮助我们解决这个问题。

模块化开发是目前最流行，也是必须要掌握的一种开发思路。而模块化其实是建立在单例模式基础之上的，因此模块化开发和闭包息息相关。

目前流行的模块化开发思路，无论是 require，还是 ES6 的 modules，虽然实现方式不同，但是核心的思路一样。因此为了方便大家理解模块化的思维，这里就以建立在函数自执行基础之上的单例模式为例，一起来感受一下模块化开发的魅力。

第一，请记住：每一个单例就是一个模块。

在未来，你可能会被告知，每一个文件，就是一个模块。而这里把每一个单例模式假想成一个单独的文件即可。定义一个模块，而变量名就是模块名。

```
var module_test = (function() {

})();
```

第二，每一个模块要想与其他模块交互，则必须有获取其他模块的能力，例如 requirejs 中的 `require` 与 ES6 modules 中的 `import`。

```
// require
var $ = require('jquery');

// es6 modules
import $ from 'jquery';
```

当然，因为这里在定义时模块名就已经是全局变量，所以就省略了这一步。

第三，每一个模块都应该有对外的接口，以保证与其他模块交互的能力。这里直接使用 return 返回一个字面量对象的方式来对外提供接口。

```
var module_test = (function() {
    ...

    return {
        testfn1: function() {},
        testfn2: function() {}
    }
})();
```

现在我们结合一个简单的案例来走一遍模块化开发的流程。这个案例想要实现的功能是每隔一秒，body 的背景色就随着一个数字的递增在固定的三种颜色之间切换。

（1）首先创建一个专门用来管理全局状态的模块。这个模块中有一个私有变量保存了所有的状态值，并对外提供了访问与设置这个私有变量的方法，代码如下。

```
var module_status = (function() {
    var status = {
        number: 0,
        color: null
    }

    var get = function(prop) {
        return status[prop];
    }

    var set = function(prop, value) {
        status[prop] = value;
    }

    return {
        get: get,
        set: set
    }
})();
```

（2）再来创建一个模块，这个模块专门负责 body 背景颜色的改变。

```
var module_color = (function() {

    // 假装用这种方式执行第二步引入模块
    // 类似于 import state from 'module_status';
    var state = module_status;
    var colors = ['orange', '#ccc', 'pink'];

    function render() {
        var color = colors[state.get('number') % 3];
        document.body.style.backgroundColor = color;
    }

    return {
        render: render
```

```
    }

})();
```

在这个模块中，引入了管理状态的模块，并且将颜色的管理与改变方式都定义在该模块中，因此在使用时我们只需调用 render 方法就可以了。

接下来我们还要创建另外一个模块来负责显示当前的 number 值，用于参考与对比。

```
var module_context = (function() {
    var state = module_status;

    function render() {
       document.body.innerHTML = 'this Number is ' + state.get('number');
    }

    return {
       render: render
    }
})();
```

这些功能模块都创建完毕之后，最后我们只需创建一个主模块即可。这个主模块的目的就是借助功能模块，来实现我们想要的效果。

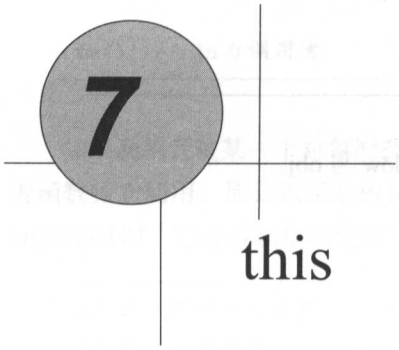
```
var module_main = (function() {
    var state = module_status;
    var color = module_color;
    var context = module_context;

    setInterval(function() {
        var newNumber = state.get('number') + 1;
        state.set('number', newNumber);

        color.render();
        context.render();
    }, 1000);
})();
```

如果你有过模块化开发的经验，那么结合前面对于闭包的理解，这段代码就再简单不过了。如果你是初次正式学习模块化的概念，那么这个例子也是非得值得细细品味的。把这段代码摘抄到一个 HTML 文件的 script 标签中即可看到展示效果，完整代码如下。

```
<script>
var module_status = (function() {
    var status = {
        number: 0,
        color: null
    }
    var get = function(prop) {
        return status[prop];
    }

    var set = function(prop, value) {
        status[prop] = value;
    }

    return {
        get: get,
        set: set
    }
})();
var module_color = (function() {

    // 假装用这种方式执行第二步引入模块
    // 类似于 import state from 'module_status';
    var state = module_status;
    var colors = ['orange', '#ccc', 'pink'];

    function render() {
        var color = colors[state.get('number') % 3];
        document.body.style.backgroundColor = color;
    }

    return {
        render: render
```

```
    }
})();
var module_context = (function() {
    var state = module_status;

    function render() {
       document.body.innerHTML = 'this Number is ' + state.get('number');
    }

    return {
        render: render
    }
})();

var module_main = (function() {
    var state = module_status;
    var color = module_color;
    var context = module_context;

    setInterval(function() {
        var newNumber = state.get('number') + 1;
        state.set('number', newNumber);

        color.render();
        context.render();
    }, 1000);
})();

</script>
```

这里就先以这个简单的例子让大家初步感受一下模块化的开发思维，等后续的章节学习了更加规范的模块化语法之后，我们再结合涉及面更广的实际案例来学习。当然，这里介绍的模块化思维与开发方式可以在简单一点的页面中运用（或者你的项目还没用到构建工具），相信用这样的思路来开发代码，肯定会更加合理。



this

在笔者 JavaScript 基础还不是很扎实的时候，去面试时最怕被问及的两个知识点，一个是闭包，另外一个就是 this。

而恰恰这两个知识点又都非常的重要，且不容易理解。如果能够把它们掌握好，我可以大胆地说，你的基础知识已经比前端从业者的一半甚至更多的人扎实了。

因此我们有必要调整一下略显浮躁的心态，认真地花一点时间把 this 这个知识点掌握好。

即便你已在网上学习过 this，这里也建议你认真地把本章内容看一遍，因为网上的文章存在不少的误导性，可能会导致你学到的知识不是那么的准确。

前面我们已经知道了，当函数被调用执行时，变量对象会生成，这个时候，this 的指向会确定。因此首先要牢记一个非常重要的结论，**当前函数的 this 是在函数被调用执行的时候才确定的**。如果当前的执行上下文处于函数调用栈的栈顶，那么这个时候变量对象会变成活动对象，同时 this 的指向确认。

正是由于这个原因，才导致一个函数内部的 this 到底指向谁是非常灵活且不确定的，这也是 this 难以被真正理解的原因所在。例如，下面的例子，同一个函数由于调用的方式不同，它内部的 this 指向了不同的对象。

```
var a = 10;
var obj = {
    a: 20
}
```

```
function fn () {
    console.log(this.a);
}

fn(); // 10
fn.call(obj); // 20
```

通过 a 值的不同表现，我们可以知道 this 分别指向了 window 与 obj。

接下来，我们一步步来分析 this 中的具体表现。

1. 全局对象中的 this

在之前变量对象的学习中曾提到过，全局对象的变量对象是一个比较特殊的存在。在全局对象中，this 指向它本身，因此相对简单，没有那么多复杂的情况需要考虑。

```
// 通过this绑定到全局对象
this.a2 = 20;

// 通过声明绑定到变量对象，但在全局环境中，变量对象就是它本身
var a1 = 10;

// 仅仅只有赋值操作，标识符会隐式绑定到全局对象
a3 = 30;

// 输出结果全部符合预期
console.log(a1);
console.log(a2);
console.log(a3);
```

2. 函数中的 this

在上面的例子中，同一个函数中的 this 由于调用方式不同导致 this 指向不同，因此，this 最终指向谁，与调用该函数的方式息息相关。

在一个函数的执行上下文中，this 由该函数的调用者提供，由调用函数的方式来决定其指向。下面的例子展示了谁是调用者。

```
function fn() {
    console.log(this);
}

fn(); // fn为调用者
```

如果调用者被某一个对象所拥有，那么在调用该函数时，内部的 this 指向该对象。如果调用者函数独立调用，那么该函数内部的 this 则指向 undefined。但是在非严格模式中，当 this 指向 undefined 时，它会自动指向全局对象。

```
// 为了能够准确判断，我们在函数内部使用严格模式
// 因为非严格模式会自动指向全局
function fn() {
    'use strict';
    console.log(this);
}

fn();  // fn是调用者，独立调用，this为undefined
window.fn();  // fn是调用者，被window所拥有，this为window对象
```

函数是独立调用，还是被某个对象拥有，是非常容易辨认的。综合上面的结论，下面结合一些简单的例子来分析。

```
// demo01
var a = 20;

var obj = {
    a: 40
}
function fn() {
    console.log('fn this: ', this);

    function foo() {
        console.log(this.a);
    }
```

```
    foo();
}
fn.call(obj);
fn();
```

这个例子中 fn 最终的调用方式不同，因此在 fn 的环境中，this 会有所变化。但是无论 fn 如何调用，在 fn 执行时，foo 始终都是独立调用。因此 foo 内部的 this 都是指向 undefined 的，但是由于这是非严格模式，因此自动转向 window，所以上面例子的输出结果如下。

```
fn this: Object { a: 40 }
20

fn this: Window {}
20
```

```
// demo02
'use strict';

var a = 20;
function foo () {
    var a = 1;
    var obj = {
        a: 10,
        c: this.a + 20
    }
    return obj.c;

}

console.log(window.foo());  // 20
console.log(foo());  // 报错 TypeError
```

对象字面量的写法并不会产生自己的作用域，因此 demo02 中的 obj.c 上的 this 属性并不会指向 obj，而是与 foo 函数内部的 this 一样。

因此当使用 `window.foo()` 调用时，foo 内部的 this 指向 window 对象，这个时候 `this.a` 则能访问到全局的 a 变量。但是当 `foo()` 独立调用时，foo 内部的 this 指向 undefined，由于这个例子是在严格模式中，因此并不会转向 window 对象，此时执行代码会报错，`Uncaught TypeError: Cannot read property 'a' of undefined`。

```
// demo03
var a = 20;
var foo = {
    a: 10,
    getA: function () {
        return this.a;
    }
}
console.log(foo.getA()); // 10

var test = foo.getA;
console.log(test());  // 20
```

这是一个非常容易理解错误的例子，但只要我们牢牢记住调用者的独立调用与被某个对象所拥有的区别，就不怕任何迷魂阵。

在 demo03 的 `foo.getA()` 中，getA 为调用者，被 foo 所拥有，当 getA 执行时，this 指向 foo，因此执行结果返回 10。

而 `test()` 在执行时，test 为调用者，它是独立调用。虽然 test 与 `foo.getA` 的引用指向同一个函数，但是调用方式不同，因此，getA 内部的 this 指向了 undefined，自动转向 window，所以结果返回 20。

看了这些例子后，是不是对 this 的指向一清二楚了呢？下面就留几道思考题，请读者自行分析 this 的指向。如果拿不定结果，可通过断点调试的方式在 Chrome 中查看。

```
// demo04
function foo() {
    console.log(this.a)
}

function active(fn) {
    fn();
```

```
}

var a = 20;
var obj = {
    a: 10,
    getA: foo,
    active: active
}

active(obj.getA);  // 输出的值是多少?
obj.active(obj.getA); // 输出的值是多少?
```

```
// demo05
var n = 'window';
var object = {
    n: 'object',
    getN: function() {
        return function() {
            return this.n;
        }
    }
}

console.log(object.getN()());  // 输出的结果是多少?
```

3.call/apply/bind 显式指定 this

JavaScript 内部提供了一种可以手动设置函数内部 this 指向的方式，它们就是 call/apply/bind。所有的函数都能调用这三个方法。在最初学习它们的时候可能会有一些理解上的困惑，但是通过接下来的分析，相信你一定能完全掌握它们。

假设有如下一个例子。

```
var a = 20;
var object = {
    a: 40
```

```
}

function fn() {
    console.log(this.a);
}
```

如果正常调用函数 fn，那么很容易想到，fn 为独立调用，因此 this 最终会指向 window，所以函数的输出结果为 20。

```
fn(); // 20
```

还可以通过如下方式，当 fn 运行时，显式指定函数内部的 this。

```
fn.call(object); // 40
fn.apply(object); // 40
```

当函数调用 call/apply 时，则表示会执行该函数，并且函数内部的 this 指向 call/apply 的第一个参数。

而 call/apply 的不同之处在于参数的传递形式，例如：

```
function fn(num1, num2) {
    return this.a + num1 + num2;
}

var a = 20;
var object = { a: 40 }
```

call 的第一个参数是为函数内部指定 this 指向，后续的参数则是函数执行时所需要的参数，一个个传递。

apply 的第一个参数与 call 相同，为函数内部 this 指向，而函数的参数，则以数组的形式传递，作为 apply 的第二个参数。

```
// 正常执行
fn(10, 10); // 40
```

```
// 通过call改变this指向
fn.call(object, 10, 10); // 60

// 通过apply改变this指向
fn.apply(object, [10, 10]); // 60
```

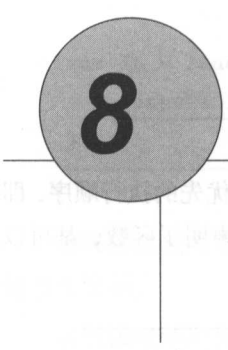
bind 方法也能指定函数内部的 this 指向，但是它与 call/apply 有所不同。

当函数调用 call/appy 时，函数的内部 this 被显式指定，并且函数会立即执行。而当函数调用 bind 时，函数并不会立即执行，而是返回一个新的函数，这个新的函数与原函数有共同的函数体，但它并非原函数，并且新函数的参数与 this 指向都已经被绑定，参数为 bind 的后续参数。

通过一个例子来理解。

```
function fn(num1, num2) {
    return this.a + num1 + num2;
}

var a = 20;
var object = { a: 40 }

var _fn = fn.bind(object, 1, 2);

console.log(_fn === fn); // false
_fn(); // 43
_fn(1, 4); // 43  因为参数被绑定，因此重新传入参数是无效的
```

call/apply/bind 的特性让 JavaScript 变得十分灵活，它们的应用场景十分广泛，例如，将类数组转化为数据，实现继承，实现函数柯里化等。这里先记住它们的基础知识与基本特性，在后续章节中，还会遇到与 this 相关的知识。



函数与函数式编程

函数是 JavaScript 的基础语法之一，当然，也是最重要的，必须要掌握好的知识点之一。

前面所学的执行上下文、作用域、变量对象、闭包、this 等知识点，其实都是围绕函数在展开。

前面的学习可以说是在进一步认识函数，而接下来的学习，则是在认识了函数之后，来使用函数。

在学习了那么多函数本质的知识点之后，是时候来做个总结了。

◎ 在实际开发中，需要用函数来做些什么？

◎ 可以用函数的这些特性可以玩哪些高级的东西？

◎ 要怎样来使用函数才能让代码更加清晰、直观与合理？

当然，有的应用在前面已经涉及了一些，而本章，我们将会学到更多。如果你迫不及待地想要自己对函数的使用往大神级靠近，那么从现在开始，一步步把函数的更多的东西都掌握起来吧。学习完本章内容之后，相信你一定会清晰地感知到自己对于函数运用的进步。

8.1 函数

在实际开发中，经常能遇到的函数形式大概有四种，分别是函数声明、函数表达式、匿名函数与自执行函数。

可能大家对这几种常见的函数形式已不再陌生，但这里还是要专门总结一下关于这些函数的基础知识，以确保每一位读者都有足够的理论基础以进行下一步的学习。

1. 函数声明

函数声明是指利用关键字 function 来声明一个函数。

```
function fn() {
    console.log('function');
}
```

在变量对象的创建过程中，function 声明的函数比 var 声明的变量有更加优先的执行顺序，即我们常常提到的函数声明提前。因此在同一执行上下文中，无论在什么地方声明了函数，都可以直接使用。

```
function fn() {
    var a = 20;
    function bar() {
        var b = 10;
        return a + b;
    }

    return bar();
}

fn(); // 30
```

2. 函数表达式

函数表达式其实是将一个函数体赋值给一个变量的过程。

```
var fn = function() {}
```

因此，我们理解函数表达式时，可以与变量共同理解。上面代码的执行步骤如下。

```
var fn = undefined;
fn = function() {}
```

其中，var fn = undefined 会因为变量对象的原因而提前执行，这就是常常被提到的变量提升。因此当我们使用函数表达式时，必须要考虑代码的先后顺序，这是与函数声明不同的地方。

```
fn(); // TypeError: fn is not a function

var fn = function() {
    console.log('function');
}
```

如果你比较喜欢使用函数表达式，那么一定要有一个良好的编码习惯，以规避变量提升带来的负面影响。

函数体赋值的操作在其他很多场景中都能够遇到，如下所示。

```
function Person(name) {
    this.name = name;
    this.age = age;
    // 在构造函数内部添加方法
    this.getAge = function() {
        return this.age;
    }
    this.
}

// 给原型添加方法
Person.prototype.getName = function() {
    return this.name;
}

// 在对象中添加方法
var a = {
    m: 20,
    getM: function() {
        return this.m;
    }
}
```

3. 匿名函数

顾名思义，匿名函数就是没有名字的函数，一般会作为一个参数或者作为一个返回值来使用，通常不使用变量来保存它的引用。常见的场景如下。

（1）setTimeout 中的参数。

```
var timer = setTimeout(function() {
    console.log('延迟1000ms执行该匿名函数');
}, 1000);
```

（2）数组方法中的参数。

```
var arr = [1, 2, 3];

arr.map(function(item) {
    return item + 1;
})

arr.forEach(function(item) {
    // do something
})
```

（3）匿名函数作为一个返回值。

```
function add() {
    var a = 10;
    return function() {
        return a + 20
    }
}

add()();
```

在实际开发中，当匿名函数作为一个返回值时，为了方便调试，常常会为匿名函数添加一个名字，这样在 Chrome 开发者工具中就能知道是它。

```
function add() {
    var a = 10;
    return function bar() {
        return a + 20
    }
}

add()();
```

注意：许多人会分不清匿名函数和闭包。相信在学完前面几章的内容之后，你也会觉得这是一个很奇怪的现象。但是为了防止万一，这里还是稍微解释一下。匿名函数就当成函数来理解即可，而闭包的形成条件，仅仅只是有的时候会和匿名函数有关而已。

4. 自执行函数

自执行函数是匿名函数一个非常重要的应用场景。因为函数会产生独立的作用域，因此我们常常利用自执行函数来模拟块级作用域，并进一步在此基础上实现模块化的运用。

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

模块化的重要性需要反复强调，在学习这些基础知识的过程中，应慢慢养成对于模块化思维的一个认知与习惯。因此，虽然前面在讲解闭包时对模块已经做了非常详细的分析，但这里仍然要借助自执行函数来了解模块化。

一个模块可以包括：私有变量、私有方法、公有变量、公有方法。

当我们使用自执行函数创建一个模块时，也就意味着，外界已经无法访问该模块内部的任何属性与方法了。好在有闭包，作为模块之间能相互交流的桥梁，让模块能够在我们的控制之下，选择性地对外开发属性与方法。

```
(function() {
    // 私有变量
    var age = 20;
    var name = 'Tom';
```

```
    // 私有方法
    function getName() {
        return `your name is ` + name;
    }


    // 公有方法
    function getAge() {
        return age;
    }

    // 将引用保存在外部执行环境的变量中，这是一种简单的对外开发方法的方式
    window.getAge = getAge;
})();
```

在前面的章节中我们已经创建过一个简单的状态管理模块，这里我们将扩展它，以便应对更加复杂的场景。

```
// 自执行创建模块
(function() {
    // states 结构预览
    // states = {
    //     a: 1,
    //     b: 2,
    //     m: 30,
    //     o: {}
    // }
    var states = {};  // 私有变量，用来存储状态与数据

    // 判断数据类型
    function type(elem) {
        if(elem == null) {
            return elem + '';
        }
```

```
    return toString.call(elem).replace(/[\[\]]/g, '').split(' ')[1].
    toLowerCase();
}


/**
 * @Param name 属性名
 * @Description 通过属性名获取保存在states中的值
*/
function get(name) {
    return states[name] ? states[name] : '';
}

function getStates() {
    return states;
}

/*
* @param options {object} 键值对
* @param target {object} 属性值为对象的属性，只在函数实现，递归调用时
传入
* @desc 通过传入键值对的方式修改state树，使用方式与小程序的data或者
react中的setStates类似
*/
function set(options, target) {
    var keys = Object.keys(options);
    var o = target ? target : states;

    keys.map(function(item) {
        if(typeof o[item] == 'undefined') {
            o[item] = options[item];
        }
        else {
            type(o[item]) == 'object' ? set(options[item], o[item]) :
             o[item] = options[item];
        }
```

```
            return item;
        })
    }

    // 对外提供接口
    window.get = get;
    window.set = set;
    window.getStates = getStates;
})()

// 具体使用方式如下

set({ a: 20 });      // 保存属性a
set({ b: 100 });     // 保存属性b
set({ c: 10 });      // 保存属性c

// 保存属性o，它的值为一个对象
set({
    o: {
        m: 10,
        n: 20
    }
})

// 修改对象o 的m值
set({
    o: {
        m: 1000
    }
})

// 给对象o中增加一个c属性
set({
    o: {
        c: 100
    }
```

```
})
console.log(getStates())
```

8.2 函数式编程

当我们想要使用一个函数时，其实就是想将一些功能、逻辑等封装起来以便使用。相信有一些编码基础的读者对于封装这个概念并不陌生，我们经常使用函数封装来做一些想要做的事情。

例如，若想计算任意三个数的和，就可以将这三个数作为参数封装成一个简单的函数。

```
function add(a, b, c) {
    return a + b + c;
}
```

当再次需要计算三个数的和时，直接调用该函数即可。

```
add(1, 2, 3);
```

一般来说，当想要做的事情比较简单时，可能还看不出封装成函数之后带来的便利。如果想要做的事情稍微复杂一点呢，例如想要计算一个数组中所有子项的和。

```
function mergeArr(arr) {
    var result = 0;
    for(var i = 0; i ++; i < arr.length) {
        result += arr[i];
    }

    return result;
}
```

如果不通过封装成函数的方式，而是每次都用 for 循环去计算数组中所有子项的和，那么代码量肯定就会偏多。封装之后，当再次做这件事情的时候，只需用一句代码即可。

```
mergeArr([1, 2, 3, 4, 5]);
```

函数封装的意义现在已非常明确，但面临的问题是，当想要去封装一个函数时，怎么做才是最好的呢？

如果没有认真想过这个问题，那么你封装的函数可能会非常的糟糕。也许使用起来并不是那么好用，甚至可能会导致你的程序里出现无法预知的 bug。因此实践中在做函数封装的时候，我们有必要学习一些优秀的封装习惯，来让自己的代码看上去更加的专业与可靠，而不是连自己都没有底气封装。

需要提醒大家的是，我们需要忘记前面所有 demo 中函数的编写风格。因为那仅仅只是为了方便大家理解闭包、this 等，只是列出来的简单例子，如果将这样的编写风格放在实践中，那么一定是一场灾难。

这里将要学习的函数封装思维叫作**函数式编程**。

与函数式编程对应的叫作命令式编程。这也是我们在初学代码时，不由自主地会使用的编码方式。

下面用一个简单的例子来区分这两种不同的思维。

我们在实践中常常需要处理各种不同的数据，假设这个时候有一个数组，我们需要找出该数组中所有类型为 number 的子项。

当使用命令式编程时，写出的代码如下所示。

```
var array = [1, 3, 'h', 5, 'm', '4'];
var res = [];
for(var i = 0; i < array.length; i ++) {
    if (typeof array[i] === 'number') {
        res.push(array[i]);
    }
}
```

在这种实现方式中，平铺直叙地达到了我们想要的目的。这样做的问题在于，当在另外一个时刻/场景，出现了同样的需求，或者需要将另外一个数组中的 number 子项也找出来，那么用这种方式的后果就是不得不把实现逻辑再重写一遍，因此这个时候代码就变得非常冗余且难以维护。

而函数式编程的思维则是当遇到这种场景时，把逻辑封装起来。

```
function getNumbers(array) {
    var res = [];
```

```
    array.forEach(function(item) {
        if (typeof item === 'number') {
            res.push(item);
        }
    })
    return res;
}

// 以上是封装，以下是功能实现
var array = [1, 3, 'h', 5, 'm', '4'];
var res = getNumbers(array);
```

在函数式编程实践中，我们封装了一个工具方法，专门用来找出一个数组中的所有 number。而我们真正需要维护的代码则仅仅只有两行。我们只需知道 `getNumbers` 这个工具方法能做什么即可，而不关心它的内部如何实现。这样代码会简洁许多，维护起来也非常简单。

在现实生活中这种思维场景也非常多见，例如点外卖。只需要在 App 里下单，然后等待外卖小哥把食物送到我们手里就行了。我们不用关注是如何做出来的，也不用关注外卖小哥是如何送过来的。这种更加关心结果的思维方式，就是函数式编程。

通过一些简单的例子了解了函数式编程之后，第 9 章我们将更加详细地学习函数式编程的思维。

8.2.1 函数是一等公民

所谓的“一等公民”，其实就是普通公民。也就是说，函数其实没有什么特殊的，我们可以像对待任何其他数据类型一样对待函数。

◎ 可以把函数赋值给一个变量。

```
var fn = function() {}
```

◎ 也可以把函数存在数组里。

```
function fn(callback) {
    var a = 20;
    return callback(20, 30) + a;
```

```
}

function add(a, b) {
    return a + b;
}

fn(add); // 70
```

◎ 还可以把函数作为另一个函数运行的返回值。

```
function add(x) {
    var y = 20;
    return function() {
        return x + y;
    }
}

var _add = add(100);
_add();  // 120
```

这些都是 JavaScript 的基本概念，但是很多人都无视这些概念。下面用一个简单的例子来验证一下。

首先自定义如下这样一个函数，要求在 5000ms 之后执行该函数，我们应该怎么做？建议思考 10s 再往下看。

```
function delay() {
    console.log('5000ms之后执行该函数。');
}
```

有的人可能会这样写。

```
var timer = setTimeout(function() {
    delay();
}, 5000);
```

很显然，这样做能够达到我们的目的，但这也正是我们忽视了上面的概念写出来的糟糕代码。

函数既然能够作为一个参数传入另外一个函数，那么是不是可以直接将 delay 函数传入，而不用在固有的思维上额外再封装一层多余的 function 呢？

```
var timer = setTimeout(delay, 5000);
```

当然，如果你已经提前想到这样做了，那么恭喜你。

第一种写代码的方式在未来还会遇到很多次，如果你没有特别关注，那么你可能仍然会写出糟糕的代码。为了验证大家确实理解了，现在需要思考一下如何优化下面的例子。

```
function getUser(path, callback) {
    return $.get(path, function(info) {
        return callback(info);
    })
}

getUser('/api/user', function(resp) {
    // resp为成功请求之后返回的数据
    console.log(resp);
})
```

在这个例子中，我们期望封装一个获取用户信息的函数，并期望在请求成功之后把需要处理的事情放在回调函数 callback 中来做。

下面一起来分析一下，先看看 getUser 这个方法内部的实现。

```
$.get(path, function(info) {
    return callback(info);
})
```

看这一段代码，是不是和上面 setTimeout 的例子一模一样？callback 函数被额外包裹了一层没有意义的函数，因此第一步就是对其进行简化。

```
$.get(path, callback);
```

于是最初的例子其实等同于如下代码。

```
function getUser(path, callback) {
    return $.get(path, callback);
}
```

但是再仔细观察，是不是又发现了同样的问题，$.get 方法也同样被包裹了一层没有意义的函数。因此再优化，则得到如下结果。

```
// $.get是jquery自带的工具方法
var getUser = $.get;
```

是不是很神奇。

当然，可能会有一部分人对于参数的处理有一些疑问，下面就通过一个例子来进行简单的类比。

```
function add(a, b) {
    return a + b;
}

var other = add;
other(10, 20); // 30
```

在未来的编程中，我们还会遇到非常多类似的情形，如果能够意识到，自己正在正确地使用函数一等公民的身份，那么恭喜你，你对函数的理解已经先人一步。

8.2.2 纯函数

相同的输入总会得到相同的输出，并且不会产生副作用的函数，就是纯函数。

我们可以通过一个是否会改变原始数据的两个同样功能的方法来区别纯函数与非纯函数之间的不同。

我们期望封装一个函数，能够获取到传入数组的最后一项。那么可以通过以下两种方式来实现。

```
function getLast(arr) {
    return arr[arr.length];
}

function getLast_(arr) {
    return arr.pop();
}

var source = [1, 2, 3, 4];

var last = getLast(source);  // 返回结果4，原数组不变
var last_ = getLast_(source); // 返回结果4，原数组最后一项被删除
```

getLast 与 getLast_虽然都能够获得数组的最后一项值，但是 getLast_改变了原数组。而当原始数组被改变，我们再次调用该方法时，得到的结果就会变得不一样。这种不可预测的封装方式是非常糟糕的，它会把我们的数据搞得非常混乱。在 JavaScript 原生支持的数据方法中，也有许多不纯的方法，我们在使用时要多加警惕，要清晰地知道原始数据的改变是否会留下隐患。

```
var source = [1, 2, 3, 4, 5];

source.slice(1, 3);  // 纯函数返回[2, 3]，source不变
source.splice(1, 3); // 不纯的函数返回[2, 3, 4]，source被改变

source.pop(); // 不纯的
source.push(6); // 不纯的
source.shift(); // 不纯的
source.unshift(1); // 不纯的
source.reverse(); // 不纯的

// 不能短时间知道现在source被改变成什么样子了，干脆重新约定一下
source = [1, 2, 3, 4, 5];

source.concat([6, 7]);  // 纯函数返回[1, 2, 3, 4, 5, 6, 7]，source不变
source.join('-'); // 纯函数返回1-2-3-4-5，source不变
```

与这种会改变原始数据的函数相比，纯函数明显更加可靠。很显然我们都不希望自己的数据在经过几次调用之后就变得一团糟。

纯函数还有一个重要的特点，那就是除传入的参数外，不依赖任何外界的信息与状态。例如下面这个例子。

```
var name = 'Jake';

function sayHello() {
    return 'Hello, ' + name;
}

sayHello();  // Hello, Jake

// 当我们有其他需求时需要改变name的值
name = 'Tom';
sayHello(); // Hello, Tom
```

同样的调用，但是由于 sayHello 函数依赖于外界的 name 变量，因此当外界变量发生变化时，函数的运行结果就变得不一样。很显然这并不是我们封装函数时所希望看到的状况，因为这样的变化无法预测。因此，对于上面的例子，我们应该把 name 当作一个参数传入，这样就能够直观地看到该函数执行时会输出的结果了。

```
function sayHello(name) {
    return 'Hello, ' + name;
}
```

1. 纯函数的可移植性

在封装一个函数、一个库或一个组件时，其实都期望一次封装，多处使用，而纯函数刚好具备这样的特性。

纯函数不依赖参数之外的值，因此纯函数的依赖非常明确。也正是如此，我们才能够把一些常用的功能封装成一个公共方法，这样以后遇到类似的场景时就不用再重新封装了。

我们知道一个页面的URL里常常会在“?”后面带有参数，例如 `https://www.baidu.com/s?tn=baidu&wd=javascript&rsv_sug=1`。很多时候我们需要从这段 URL 中，获取某些参数对应的

值。例如，这个例子中的“wd”的值为 javascript。那么要想封装这样一个纯函数，应该怎么做呢？代码如下所示。

```
function getParams(url, param) {
    if (!/\?/.test(url)) {
        return null;
    }

    var search = url.split('?')[1];
    var array = search.split('&');

    for(var i = 0; i < array.length; i++) {
        var tmp = array[i].split('=');
        if (tmp[0] === param) {
            return decodeURIComponent(tmp[1]);
        }
    }

    return null;
}

var url = 'https://www.baidu.com/s?tn=baidu&wd=javascript&rsv_sug=1';
getParams(url, 'wd'); // javascript
```

虽然 getParams 并非完全健壮，但是已经足以体现纯函数可移植的特点。我们可以在任何需要从 URL 中取得参数对应值的地方调用该方法。

2. 纯函数的可缓存性

在实践中我们可能会处理大量的数据，例如根据日期，得到当日相关的数据，并处理成前端能够使用的数据。假设我们封装了一个 process 方法来处理每天的数据，而这个处理过程会很复杂。如果不缓存处理结果，那么每次想要得到当天的数据时，就不得不从原始数据再转换一次。当数据的处理足够复杂时，那么很可能不是性能最优的解决方案。而纯函数的特点是，相同的输入总能得到相同的输出，因此如果将处理过的每一天的数据都缓存起来，那么当第二次或者更多次的想要得到当天的数据时，就不用经历复杂的处理过程了。

```
// 传入日期，获取当天的数据
function process(date) {
    var result = '';

    // 略掉中间复杂的处理过程

    return result;
}

function withProcess(base) {
    var cache = {}

    return function() {
        var date = arguments[0];
        if (cache[date]) {
            return cache[date];
        }
        return base.apply(base, arguments);
    }
}

var _process = withProcess(process);

//经过上面一句代码处理之后，就可以使用_process来获取我们想要的数据了。
//如果数据存在，就返回缓存中的数据；如果不存在，则调用process方法重新获取
_process('2017-06-03');
_process('2017-06-04');
_process('2017-06-05');
```

上面例子中利用了闭包的特性，将处理过的数据都缓存在了 cache 中。这种方式算是高阶函数的一种运用。我们将在 8.2.3 节介绍高阶函数。也正是因为纯函数的可靠性，才能够确保缓存的数据一定就是我们想要的正确结果。

至此，什么是纯函数，纯函数有什么特点，以及为什么要尽量使用纯函数已解释得非常清楚了。虽然在实践中并不是所有的场景都能够使用纯函数，但还是应尽量在合适的场景使用它。

8.2.3 高阶函数

大家对于 JavaScript 面向对象相关的知识可能都有所涉猎（如果还没有接触过，可以在后面的章节中学习），有一个问题可能困扰过很多人，那就是在构造函数中，如果使用了 this，那么这个 this 指向的是谁？如果在定义的原型方法中使用了 this，那么这个 this 又指向谁？是构造函数、原型，还是实例？

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.getName = function() {
    return this.name;
}

var p1 = new Person('Jake', 18);
p1.getName();
```

构造函数其实就是普通的函数，而 this 是在函数运行时才确认的。那么是什么导致了构造函数变得特别？

答案与 new 关键字有关。

如果我们自定义一个 New 方法，来模拟关键字 new 的能力，那么会有如下实现。

```
// 将构造函数以参数形式传入
function New(func) {

    // 声明一个中间对象，该对象为最终返回的实例
    var res = {};
    if (func.prototype !== null) {

        // 将实例的原型指向构造函数的原型
        res.__proto__ = func.prototype;
    }

    // ret为构造函数执行的结果，这里通过apply，
```

```
    // 将构造函数内部的this指向修改为指向res，即为实例对象

    var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));

    //当在构造函数中明确指定了返回对象时，那么new的执行结果就是该返回对象
    if ((typeof ret === "object" || typeof ret === "function") && ret !==
     null) {
        return ret;
    }

    // 如果没有明确指定返回对象，则默认返回res，这个res就是实例对象
    return res;
}
```

为了方便大家理解，我们在例子中做了详细的注解。通过 New 方法的实现可以看出，当 New 执行时，利用 apply 设定了传入的构造函数的 this 指向，因此当使用 New 方法创建实例时，构造函数中的 this 就指向了被创建的实例。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.getName = function() {
    return this.name;
}

// 使用上例中封装的New方法来创建实例
var p1 = New(Person, 'Jake', 18);
var p2 = New(Person, 'Tom', 20);
p1.getName(); // Jake
p2.getName(); // Tom
```

如果把当前函数看成基础函数，那么高阶函数，就是让当前函数获得额外能力的函数。如果把构造函数看成基础函数，那么 New 方法，就是构造函数的高阶函数。构造函数本就和普通函数一样，没有什么区别。但是因为 New 的存在，它获得了额外的能力。New 方法每次执行都会

创建一个新的中间对象，并将中间对象的原型指向构造函数的原型，将构造函数的 this 指向该中间对象。这样统一逻辑的封装，就是高阶函数的运用。

当然，如果简单粗暴一点来理解，则凡是接收一个函数作为参数的函数，就是高阶函数。但是如果这样理解，那么我们可能并不能很好地利用高阶函数的特性来让代码变得更加优雅。因为高阶函数其实是一个高度封装的过程，理解它需要一点想象力。接下来，就借助几个例子，来理解高阶函数的封装。

1. 数组 map 方法封装的思考过程

我们知道数组有一个 map 方法，它对数组中的每一项运行给定的函数，返回每次函数调用的结果并组成数组。简单来说，就是遍历数组的每一项，并且在 map 的第一个参数中进行运算处理后返回计算结果，最终返回一个由所有计算结果组成的新数组。

```
// 声明一个被遍历的数据array
var array = [1, 2, 3, 4];

// map方法第一个参数为一个回调函数，该函数拥有三个参数
// 第一个参数表示array数组中的每一项
// 第二个参数表示当前遍历的索引值
// 第三个参数表示数组本身
// 该函数中的this指向map方法的第二个参数，若该参数不存在，则this指向丢失
var newArray = array.map(function(item, i, array) {
    console.log(item, i, array, this); // 可运行查看每一项参数的具体值
    return item + 1;
}, { a: 1})

// newArray为一个新数组，由map遍历的结果组成
console.log(newArray); // [2, 3, 4, 5]
```

在上面的例子中，我们详细分析了 map 的所有细节。现在需要思考的是，如果要我们自己来封装这样一个方法，应该怎么办？

因为所有的数组遍历方法，其实都是在 for 循环的基础之上封装的，因此我们可以从 for 循环开始考虑。

当然，一个 for 循环的过程其实很好封装，其难点在于，for 循环里面对数组每一子项所做的事情很难用一个固定的模式把它封装起来，在不同的场景下，for 循环对数据的处理肯定是不一

样的，那么应该怎么办呢？

在封装函数时，对于一个不确定的变量，我们可以用往函数中传入参数的方式来指定它，例如：

```
function add(a) {
    return a + 10;
}
```

同样的道理，对于一个不确定的处理过程，我们可以用往函数中传入另外一个函数的方式来自定义这个处理过程。因此，基于这个思路，我们可以按照如下方式来封装 map 方法。

```
Array.prototype._map = function(fn, context) {
    // 首先定义一个数组来保存每一项的运算结果，最后返回
    var temp = [];
    if(typeof fn == 'function') {
        var k = 0;
        var len = this.length;
        // 封装for循环过程
        for(; k < len; k++) {
            // 将每一项的运算操作丢进fn里，
            // 利用call方法指定fn的this指向与具体参数
            temp.push(fn.call(context, this[k], k, this))
        }
    } else {
        console.error('TypeError: '+ fn +' is not a function.');
    }

    // 返回每一项运算结果组成的新数组
    return temp;
}

var newArr = [1, 2, 3, 4]._map(function(item) {
    return item + 1;
})
// [2, 3, 4, 5]
```

回过头反思 map 方法的封装过程可以发现，其实我们封装的是一个数组的 for 循环过程。每一个数组在使用 for 循环遍历时，虽然无法确认在 for 循环中到底会做什么事情，但是可以确定的是，它们一定会使用 for 循环。

因此我们把“都会使用 for 循环”这个公共逻辑封装起来，而具体要做什么事，则以一个函数作为参数的形式，来让使用者自定义。这个被作为参数传入的函数，就可以称之为基础函数。而我们封装的 map 方法，就可以称之为高阶函数。

高阶函数的使用思路正在于此，它其实是一个封装公共逻辑的过程。

在实践中，高阶函数的用途十分广泛，下面就通过另外一个例子来再次感受高阶函数的魅力。

假设我们正在做一个音乐社区的项目。

很显然，在进入这个项目的每一个页面时，都必须判断当前用户是否已经登录。因为登录与未登录所展示的页面肯定是有很多差别的。不仅如此，在确认用户登录之后，还需得到用户的具体信息，如昵称、姓名、VIP 等级、权限范围等。

因此用户状态的判断逻辑，是每个页面都必须要做的一个公共逻辑，那么在学习了高阶函数之后，我们就可以用高阶函数来做这件事情。

为了强化模块化思维，我们继续使用模块化的方式来完成这个 demo。根据现有的知识，我们可以利用自执行函数来划分模块。

首先需要一个高阶函数来专门处理获取用户状态的逻辑，因此可以单独将这个高阶函数封装为一个独立的模块。

```
// 高阶函数withLogin，用来判断当前用户状态
(function() {

    // 用随机数的方式来模拟一个获取用户信息的方法
    var getLogin = function() {
        var a = parseInt(Math.random() * 10).toFixed(0));
        if (a % 2 == 0) {
            return { login: false }
        }

        return {
            login: true,
            userinfo: {
                nickname: 'jake',
                vip: 11,
```

```
                userid: '666666'
            }
        }
    }

    var withLogin = function(basicFn) {
        var loginInfo = getLogin();

        // 将loginInfo以参数的形式传入基础函数中
        return basicFn.bind(null, loginInfo);
    }

    window.withLogin = withLogin;
})();
```

假设我们要展示主页，则可以通过 renderIndex 的方法来渲染。当然，渲染主页仍然是一个单独的模块。

```
(function() {
    var withLogin = window.withLogin;

    var renderIndex = function(loginInfo) {
        // 这里处理index页面的逻辑

        if (loginInfo.login) {
            // 处理已经登录之后的逻辑
        } else {
            // 这里处理未登录的逻辑
        }
    }

    // 对外暴露接口时，使用高阶函数包一层，来判断当前页面的登录状态
    window.renderIndex = withLogin(renderIndex);
})();
```

同样的道理，当我们想要展示其他页面，例如个人主页时，则可以使用 renderPersonal 方法，

如下所示。

```
(function() {
    var withLogin = window.withLogin;
    var renderPersonal = function(loginInfo) {
        if (loginInfo.login) {
            // do something
        } else {
            // do other something
        }
    }
    window.renderPersonal = withLogin(renderPersonal);
})();
```

当我们使用高阶函数封装每个页面的公共逻辑之后，会发现我们的代码逻辑变得非常清晰，而且更加统一。当再写新的页面逻辑时，就在此基础之上完成即可，而不用再去考虑已经封装过的逻辑。

最后，在合适的时机使用这些渲染函数即可。

```
(function() {
    window.renderIndex();
})();
```

相信我，在你的项目中使用高阶函数之后，你的代码会变得更加优雅。

8.2.4 柯里化

通过对前面章节的学习我们知道，所有以函数作为参数的函数，都可以叫作高阶函数。我们常常利用高阶函数来封装一些公共的逻辑。

本节我们要学习的柯里化，其实就是高阶函数的一种特殊用法。

柯里化是指这样一个函数（假设叫作 createCurry），它接收函数 A 作为参数，运行后能够返回一个新的函数，并且这个新的函数能够处理函数 A 的剩余参数。

这样的定义可能不太好理解，下面通过例子来配合理解。

假设有一个接收三个参数的函数 A。

```
function A(a, b, c) {
    // do something
}
```

又假如我们有一个已经封装好了的柯里化通用函数 createCurry。它接收 bar 作为参数，能够将 A 转化为柯里化函数，返回结果就是这个被转化之后的函数。

```
var _A = createCurry(A);
```

那么_A 作为 createCurry 运行的返回函数，能够处理 A 的剩余参数。因此下面的运行结果都是等价的。

```
_A(1, 2, 3);
_A(1, 2)(3);
_A(1)(2, 3);
_A(1)(2)(3);
A(1, 2, 3);
```

函数 A 被 createCurry 转化之后得到柯里化函数_A，_A 能够处理 A 的所有剩余参数。因此柯里化也被称为部分求值。

在简单的场景下，我们可以不借助柯里化通用式来转化得到柯里化函数，仅凭借眼力自己封装。

例如，有一个简单的加法函数，它能够将自身的三个参数加起来并返回计算结果。

```
function add(a, b, c) {
    return a + b + c;
}
```

那么 add 函数的柯里化函数_add 则可以写成：

```
function _add(a) {
    return function(b) {
        return function(c) {
            return a + b + c;
```

```
        }
    }
}
```

因此下面的运算方式是等价的。

```
add(1, 2, 3);
_add(1)(2)(3);
```

当然，柯里化通用式具备更加强大的能力，仅靠眼力自己封装的柯里化函数则自由度偏低，因此我们需要知道如何封装这样一个柯里化的通用式。

首先通过_add可以看出，柯里化函数的运行过程其实是一个参数的收集过程，我们将每一次传入的参数收集起来，并在最里层进行处理。因此在实现createCurry时，可以借助这个思路来进行封装。

代码如下。

```
// arity 用来标记剩余参数的个数
// args 用来收集参数
function createCurry(func, arity, args) {
    //第一次执行时，并不会传入arity，而是直接获取func参数的个数 func.
    length
    var arity = arity || func.length;

    // 第一次执行也不会传入args，而是默认为空数组
    var args = args || [];

    var wrapper = function() {

        // 将wrapper中的参数收集到args中
        var _args = [].slice.call(arguments);
        [].push.apply(args, _args);

        // 如果参数个数小于最初的func.length，则递归调用，继续收集参数
        if (_args.length < arity) {
            arity -= _args.length;
```

```
            return createCurry(func, arity, args);
        }

        // 参数收集完毕，执行func
        return func.apply(func, args);
    }

    return wrapper;
}
```

尽管已经做了详细的注解，但是仍不太容易理解，因此建议大家多阅读几遍。这个 createCurry 函数的封装其实借助了闭包与递归，实现了一个参数收集，并在收集完毕之后执行所有参数。

有些读者可能已经发现，函数经过 createCurry 转化为一个柯里化函数后，最后执行的结果，不是正相当于执行函数自身吗？柯里化是不是把简单的问题复杂化了？

如果能够提出这样的问题，说明你对柯里化已经有了一定的了解。柯里化确实是把简单的问题复杂化了，但在复杂化的同时，我们在使用函数时拥有了更多的自由度。对于函数参数的自由处理，正是柯里化的核心所在。

下面举一个常见的例子。

如果想要验证一串数字是否是正确的手机号，那么按照普通的思路来做，可能会这样封装，代码如下：

```
function checkPhone(phoneNumber) {
    return /^1[34578]\d{9}$/.test(phoneNumber);
}
```

而如果想要验证是否是邮箱呢？很可能会这么封装：

```
function checkEmail(email) {
    return /^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/.test(email);
}
```

当然，还可能会遇到验证身份证号、验证密码等各种验证信息，因此在实践中，为了统一逻辑，我们会封装一个更为通用的函数，把待验证的正则表达式与将要被验证的字符串作为参数传入。

```
function check(reg, targetString) {
    return reg.test(targetString);
}
```

但是这样封装之后，在使用时又会遇到问题，因为总是需要输入一串正则表达式，这样就导致了使用时的效率低下。

```
check(/^1[34578]\d{9}$/, '14900000088');
check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@163.com');
```

这个时候，就可以借助柯里化，在 check 的基础上再做一层封装，以简化使用。

```
var _check = createCurry(check);

var checkPhone = _check(/^1[34578]\d{9}$/);
var checkEmail = _check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/);
```

最后在使用的时候就会变得更加简洁与直观了。

```
checkPhone('183888888');
checkEmail('xxxxx@test.com');
```

在这个过程中可以发现，柯里化能够应对更加复杂的逻辑封装。当情况变得多变时，柯里化依然能够应付自如。

虽然柯里化在一定程度上将问题复杂化了，也让代码变得更加不容易理解，但是柯里化在面对复杂情况时的灵活性却让我们不得不爱。

当然这个案例本身情况还算简单，所以不能特别明显地凸显柯里化的优势，我们的主要目的是借助这个案例帮助大家了解柯里化在实践中的用途。

下面继续来思考一个例子，这个例子与 map 有关。在 8.2.3 节，我们分析了封装 map 方法的思考过程。由于没有办法确认一个数组在遍历时会执行什么操作，因此只能将调用 for 循环的这个统一逻辑封装起来，而具体的操作则通过参数传入的形式让使用者自定义，这就是 map 函数。

但是，这是针对了所有的情况我们才会这样想。

在实践中常常会发现，在我们的某个项目中，针对于某一个数组的操作其实是固定的，也就是说，同样的操作，可能会在项目的不同地方调用很多次。

于是，这个时候，我们就可以在 map 函数的基础上，进行二次封装，以简化在项目中的使用。假设这个在项目中会调用多次的操作是将数组的每一项都转化为百分比：1 ~ 100%。

普通思维下可以这样来封装。

```
function getNewArray(array) {
    return array.map(function(item) {
        return item * 100 + '%'
    })
}
getNewArray([1, 2, 3, 0.12]);   // ['100%', '200%', '300%', '12%'];
```

而如果借助柯里化来二次封装这样的逻辑，则会有如下代码：

```
function _map(func, array) {
    return array.map(func);
}

var _getNewArray = createCurry(_map);

var getNewArray = _getNewArray(function(item) {
    return item * 100 + '%'
})

getNewArray([1, 2, 3, 0.12]);   // ['100%', '200%', '300%', '12%'];
getNewArray([0.01, 1]); // ['1%', '100%']
```

如果项目中的固定操作是希望对数组进行一个过滤，找出数组中的所有 Number 类型的数据，则借助柯里化思维还可以这样做。

```
function _filter(func, array) {
    return array.filter(func);
}
```

```
var _find = createCurry(_filter);

var findNumber = _find(function(item) {
    if (typeof item == 'number') {
        return item;
    }
})

findNumber([1, 2, 3, '2', '3', 4]); // [1, 2, 3, 4]

// 当我们继续封装另外的过滤操作时就会变得非常简单
// 找出数字为20的子项
var find20 = _find(function(item, i) {
    if (typeof item === 20) {
        return i;
    }
})
find20([1, 2, 3, 30, 20, 100]);  // 4

// 找出数组中大于100的所有数据
var findGreater100 = _find(function(item) {
    if (item > 100) {
        return item;
    }
})
findGreater100([1, 2, 101, 300, 2, 122]); // [101, 300, 122]
```

这里采用了与 check 例子不一样的思维方向来向大家展示我在使用柯里化时的想法，目的是想告诉大家，柯里化能够帮助我们应对更多更复杂的场景。

当然不得不承认，这些例子都太简单了，简单到如果使用柯里化的思维来处理它们会显得有一点多此一举，而且变得难以理解。因此你可能很难从这些例子中感受到柯里化的魅力。不过，如果能够通过这些例子掌握到柯里化的思维，那就是最好的结果了。在未来的实践中，如果发现用普通的思维封装一些逻辑慢慢变得困难，不妨想一想在这里学到的柯里化思维，应用起来，柯里化足够强大的自由度一定能给你一个惊喜。

当然，也并不建议在任何情况下以炫技为目的地去使用柯里化，柯里化虽然具有了更多的自

由度，但同时柯里化通用式里调用了 arguments 对象，使用了递归与闭包，因此柯里化的自由度是以牺牲了一定的性能为代价换来的。只有在情况变得复杂时，才是柯里化大显身手的时候。

额外知识补充：无限参数的柯里化

在前端面试中，可能会遇到这样一个涉及柯里化的题目。

```
// 实现一个add方法，使计算结果能够满足如下预期：
add(1)(2)(3) = 6;
add(1, 2, 3)(4) = 10;
add(1)(2)(3)(4)(5) = 15;
```

这个题目的目的是想让 add 执行之后返回一个函数能够继续执行，最终运算的结果是所有出现过的参数之和。而这个题目的难点在于参数的不固定。我们不知道函数会执行几次，因此不能使用前面封装的 createCurry 通用公式来转换一个柯里化函数，只能自己封装，该怎样操作呢？在此之前，补充两个非常重要的知识点。

一个是 ES6 函数的不定参数。假如我们有一个数组，希望把这个数组中所有的子项展开传递给一个函数作为参数，那么应该怎么做呢？

```
// 大家可以思考一下，如何将args数组的子项展开作为add的参数传入
function add(a, b, c, d) {
    return a + b + c + d;
}
var args = [1, 3, 100, 1];
```

在 ES5 中，我们可以借助之前学过的 apply 来达到这个目的。

```
add.apply(null, args);  // 105
```

而在 ES6 中，提供了一种新的语法来解决这个问题，那就是不定参，写法如下：

```
add(...args);  // 105
```

这两种写法是等效的。在接下来的实现中，我们会用到不定参的特性。

第二个要补充的知识点是函数的隐式转换。当函数直接参与其他计算时，函数会默认调用 toString 方法，直接将函数体转换为字符串参与计算。

```
function fn() { return 20 }
console.log(fn + 10);     // 输出结果 function fn() { return 20 }10
```

但是我们可以重写函数的 toString 方法，让函数参与计算时，输出我们想要的结果。

```
function fn() { return 20; }
fn.toString = function() { return 30 }

console.log(fn + 10); // 40
```

除此之外，当我们重写函数的 valueOf 方法时也能改变函数的隐式转换结果。

```
function fn() { return 20; }
fn.valueOf = function() { return 60 }

console.log(fn + 10); // 70
```

当同时重写函数的 toString 方法与 valueOf 方法时，最终的结果会取 valueOf 方法的返回结果。

```
function fn() { return 20; }
fn.valueOf = function() { return 50 }
fn.toString = function() { return 30 }

console.log(fn + 10); // 60
```

补充了这两个知识点之后，就可以来尝试完成之前的题目了。add 方法的实现仍然是一个参数的收集过程。当 add 函数执行到最后时，返回的仍然是一个函数，但是我们可以通过定义 toString/valueOf 的方式，让这个函数可以直接参与计算，并且转换的结果是我们想要的，而且它本身也仍然可以继续接收新的参数，实现方式如下。

```
function add() {
    // 第一次执行时，定义一个数组专门用来存储所有的参数
    var _args = [].slice.call(arguments);
```

```
    // 在内部声明一个函数，利用闭包的特性保存_args并收集所有的参数值
    var adder = function () {
        var _adder = function() {
            // [].push.apply(_args, [].slice.call(arguments));
            _args.push(...arguments);
            return _adder;
        };

        // 利用隐式转换的特性，当最后执行时隐式转换，计算最终的值并返回
        _adder.toString = function () {
            return _args.reduce(function (a, b) {
                return a + b;
            });
        }

        return _adder;
    }
    // return adder.apply(null, _args);
    return adder(..._args);
}

var a = add(1)(2)(3)(4);    // f 10
var b = add(1, 2, 3, 4);    // f 10
var c = add(1, 2)(3, 4);    // f 10
var d = add(1, 2, 3)(4);    // f 10

// 可以利用隐式转换的特性参与计算
console.log(a + 10); // 20
console.log(b + 20); // 30
console.log(c + 30); // 40
console.log(d + 40); // 50

// 也可以继续传入参数，得到的结果再次利用隐式转换参与计算
console.log(a(10) + 100);  // 120
console.log(b(10) + 100);  // 120
console.log(c(10) + 100);  // 120
```

```
console.log(d(10) + 100);  // 120
```

8.2.5 代码组合

在学习代码组合之前，我们需要回顾一下高阶函数的应用。

在学习高阶函数的时候，曾探讨过一个实践中的案例。每一个页面都会判断用户的登录状态，因此我们封装了一个 withLogin 的高阶函数来处理这个统一的逻辑。而每一个页面的渲染函数，则作为基础函数，通过下面的方式得到高阶函数 withLogin 赋予的新能力。这个新能力就是直接从参数中得到用户的登录状态。

```
indow.renderIndex = withLogin(renderIndex);
```

但是如果这个时候，又新增一个需求，即不仅需要判断用户的登录状态，还需要判断用户打开当前页面所处的具体环境：是在某一个 App 中打开，还是在移动端打开，或者是在 PC 端的某一个浏览器中打开。因为在不同的打开环境需要做不同的处理。

根据高阶函数的用法，还需要封装一个新的高阶函数 withEnvironment 来处理这个统一的环境判断逻辑。

```
(function() {
    var env = {
        isMobile: false,
        isAndroid: false,
        isIOS: false
    }
    var ua = navigator.userAgent;
    env.isMobile = 'ontouchstart' in document;
    env.isAndroid = !!ua.match(/android/);
    env.isIOS = !!ua.match(/iphone/);

    var withEnvironment = function(basicFn) {
        return basicFn.bind(null, env);
    }

    window.withEnvironment = withEnvironment;
```

```
})();
```

正常情况下，在使用这个高阶函数时，一般会这样做。

```
window.renderIndex = withEnvironment(renderIndex);
```

但现在的问题是，我们这里已经有两个高阶函数想要给基础函数 renderIndex 传递新能力了。因为在高阶函数的实现中使用了 bind 方法，因此 withEnvironment(renderIndex) 与 renderIndex 其实是拥有共同的函数体的，所以当遇到多个高阶函数时，也可以这样来使用。

```
window.renderIndex = withLogin(withEnvironment(renderIndex));
```

之后，就能够在 renderIndex 中接收到两个高阶函数带来的新能力了。但是这又出现了多层嵌套的使用问题，为了避免这个问题，我们可以使用代码组合的方式来解决。

我们期望有一个组合方法 compose，可以这样来使用。参数从右至左，将第一个参数 renderIndex 作为第二个参数 withEnvironment 的参数，并将运行结果作为第三个参数 withLogin 的参数，依次递推，最终返回一个新的函数。这个新函数，是在基础函数 renderIndex 之上，得到了所有高阶函数的新能力。

```
window.renderIndex = compose(withLogin, withEnvironment, renderIndex);
```

这样做之后，代码变得更加清晰直观，也不用担心更多的高阶组件进来增加嵌套。那么应该如何实现这样一个 compose 函数呢？

```
// ...args 为ES6语法中的不定参数，args表示一个由所有参数组成的数组，
// 最新的Chrome浏览器已经支持该语法
function compose(...args) {
    var arity = args.length - 1;
    var tag = false;
    if (typeof args[arity] === 'function') {
        tag = true;
    }

    if (arity > 1) {
```

```
        var param = args.pop(args[arity]);
        arity--;
        var newParam = args[arity].call(args[arity], param);
        args.pop(args[arity]);

        // newParam 是上一个参数的运行结果，我们可以打印出来查看它的值
        args.push(newParam);
        console.log(newParam);

        return compose(...args);
    } else if (arity == 1) {
        // 将操作目标放在最后一个参数，目标可能是一个函数，
        // 也可能是一个值，因此可针对不同的情况做不同的处理
        if (!tag) {
            return args[0].bind(null, args[1]);
        } else {
            return args[0].call(null, args[1]);
        }
    }
}
```

下面就来验证一下封装的这个 compose 函数是否可靠。

```
var fn1 = function(a) { return a + 100 }
var fn2 = function(a) { return a + 10 }
var fn3 = function(a) { return a + 20 }

var bar = compose(fn1, fn2, fn3, 10);
console.log(bar());

// 输出结果
// 30
// 40
// 140
```

```
var base = function() {
    return arguments[0] + arguments[1];
}

var foo1 = function(fn) {
    return fn.bind(null, 20);
}
var foo2 = function(fn) {
    return fn.bind(null, 30);
}

var res = compose(foo1, foo2, base);
console.log(res());

// 输出结果
// f() {}
// 50
```

通过这两个验证的例子，可以确定封装的这个组合函数还是比较可靠的，因此可以直接放心使用。

当然，组合函数还可以借助柯里化封装变得更加灵活。

```
window.renderIndex = compose(withLogin, withEnvironment, renderIndex);

// 还可以这样
window.renderIndex = compose(withLogin, withEnvironment)(renderIndex);
```

这里不再继续深入探讨具体的封装方法，我们可以在使用时借助工具库 lodash.js 中的 flowRight 来实现这种灵活的效果。

```
// ES6 模块化语法，引入flowRight函数
import flowRight from 'lodash/flowRight';

// ...
```

```
// ES6模块化语法，对外暴露接口
export default flowRight(withLogin, withEnvironment)(renderIndex);
```

9 面向对象

如果要我总结一下学习前端以来曾遇到过哪些瓶颈，那么面向对象一定是第一个毫不犹豫想到的。

为了帮助大家更加直观地学习和了解面向对象，我会用尽量简单易懂的描述来展示面向对象的相关知识，并且也准备了一些实用的例子帮助大家更加快速地掌握面向对象的真谛。

9.1 基础概念

9.1.1 对象的定义

在 ECMAScript-262 中，对象被定义为 **“无序属性的集合，其属性可以包含基本值、对象或者函数”**。

也就是说，对象是由一系列无序的 key-value 对组成。其中 value 可以是基本数据类型、对象、数组、函数等。

```
// 这里的person就是一个对象
var person = {
    name: 'Tom',
    age: 18,
```

```
    // value 为函数
    getName: function() {
        return this.name;
    },

    // value 为对象
    parent: {}
}
```

9.1.2 创建对象

可以通过关键字 new 来创建一个对象。

```
var obj = new Object();
```

也可以通过对象字面量的形式创建一个对象。

```
var obj = {};
```

当我们想要给创建的对象添加属性与方法时，方法如下。

```
// 可以这样
var person = {};
person.name = 'TOM';
person.getName = function() {
    return this.name;
}

// 也可以这样
var person = {
    name: 'TOM',
    getName: function() {
        return this.name;
    }
}
```

访问对象的属性与方法

假设我们有一个简单的对象如下：

```
var person = {
    name: 'TOM',
    age: 20,
    getName: function() {
        return this.name;
    }
}
```

当我们想要访问它的 name 属性时，可以使用如下方式：

```
person.name

// or
person['name'];

//or
var _name = 'name';
person[_name];
```

当我们想要访问的属性名是一个变量时，可以使用中括号的方式，方法如下：

```
['name', 'age'].forEach(function(item) {
    console.log(person[item]);
})
```

9.1.3 构造函数与原型

在学习函数式时曾提过，封装函数其实是封装一些公共的逻辑与功能，通过传入参数的形式达到自定义的效果。当面对具有共同特征的一类事物时，就可以结合构造函数与原型的方式将这类事物封装成对象。

例如，我们将“人”这一类事物封装成一个对象，那么可以这样做。

```
// 构造函数
var Person = function(name, age) {
    this.name = name;
    this.age = age;
}
// Person.prototype 为Person的原型，这里在原型上添加了一个方法
Person.prototype.getName = function() {
    return this.name;
}
```

这样，我们就利用构造函数与原型封装好了一个 Person 对象。

具体某一个人的特定属性，通常放在构造函数中。例如此处的 name、age，它们的值不是所有人的共同属性，而且仅仅属于某一个人。因为每个人的名字与年龄可能都是不同的。

所有人公共的方法与属性，通常会放在原型对象中。例如此处的 getName，它表示一个共同的动作，访问当前这个人的姓名。

当我们想要使用 Person 对象创建一个具体的“人”时，我们称这个被创建的“人”为一个**实例**。

```
var p1 = new Person('Jake', 20);
var p2 = new Person('Tom', 22);

p1.getName();  // Jake
p2.getName();  // Tom
```

p1 和 p2 都是根据对象 Person 创建的实例。我们可以通过原型的方法访问该实例的属性，例如上例中的 getName 调用。

构造函数其实与普通函数并无区别，首字母大写是一种约定，用来表示这是一个构造函数。但是 new 关键字的存在，让构造函数变得与众不同。在学习高阶函数时，我们探讨了为什么构造函数中的 this 指向的是当前的实例。为了加深印象，我们再次温习一下。

构造函数中的 this，与原型方法中的 this（实例调用该方法时确认），指向的都是当前的实例。这是面试中常常会被问及的问题，其原因是什么呢？我们可以模拟构造 new 关键字的能力，实现一个 New 方法，来观察一下 new 关键字到底做了什么事情，代码如下：

```
// 将构造函数以参数形式传入
function New(func) {

    // 声明一个中间对象，该对象为最终返回的实例
    var res = {};
    if (func.prototype !== null) {

        // 将实例的原型指向构造函数的原型
        res.__proto__ = func.prototype;
    }

    //ret为构造函数的执行结果，这里通过apply，
    //将构造函数内部的this指向修改为指向res，即实例对象
    var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));

    //当我们在构造函数中明确指定了返回对象时，
    //那么new的执行结果就是该返回对象
    if ((typeof ret === "object" || typeof ret === "function") && ret !==
     null) {
        return ret;
    }

    // 如果没有明确指定返回对象，则默认返回res，这个res就是实例对象
    return res;
}
```

我们同样可以使用自己封装好的 New 方法来创建对象。

```
var Person = function(name) {
    this.name = name;
}

Person.prototype.getName = function() {
    return this.name;
}
```

```
var p1 = New(Person, 'Jake');
var p2 = New(Person, 'Tom');

p1.getName(); // Jake
p2.getName(); // Tom
```

使用 New 方法声明的实例与 new 关键字声明的实例拥有同样的能力与特性。

因此，通过对 New 方法的封装，我们知道 new 关键字在创建实例时经历了如下过程：

- ◎ 先创建一个新的、空的实例对象；
- ◎ 将实例对象的原型，指向构造函数的原型；
- ◎ 将构造函数内部的 this，修改为指向实例；
- ◎ 最后返回该实例对象。

如果使用 PPrototype 指代原型对象，那么构造函数、原型、实例之间有如下关系。

```
// -> 表示指向
Person.prototype -> PPrototype;
p1.__proto__ -> PPrototype;
p2.__proto__ -> PPrototype;
PPrototype.constructor -> Person
```

如图 9-1所示。

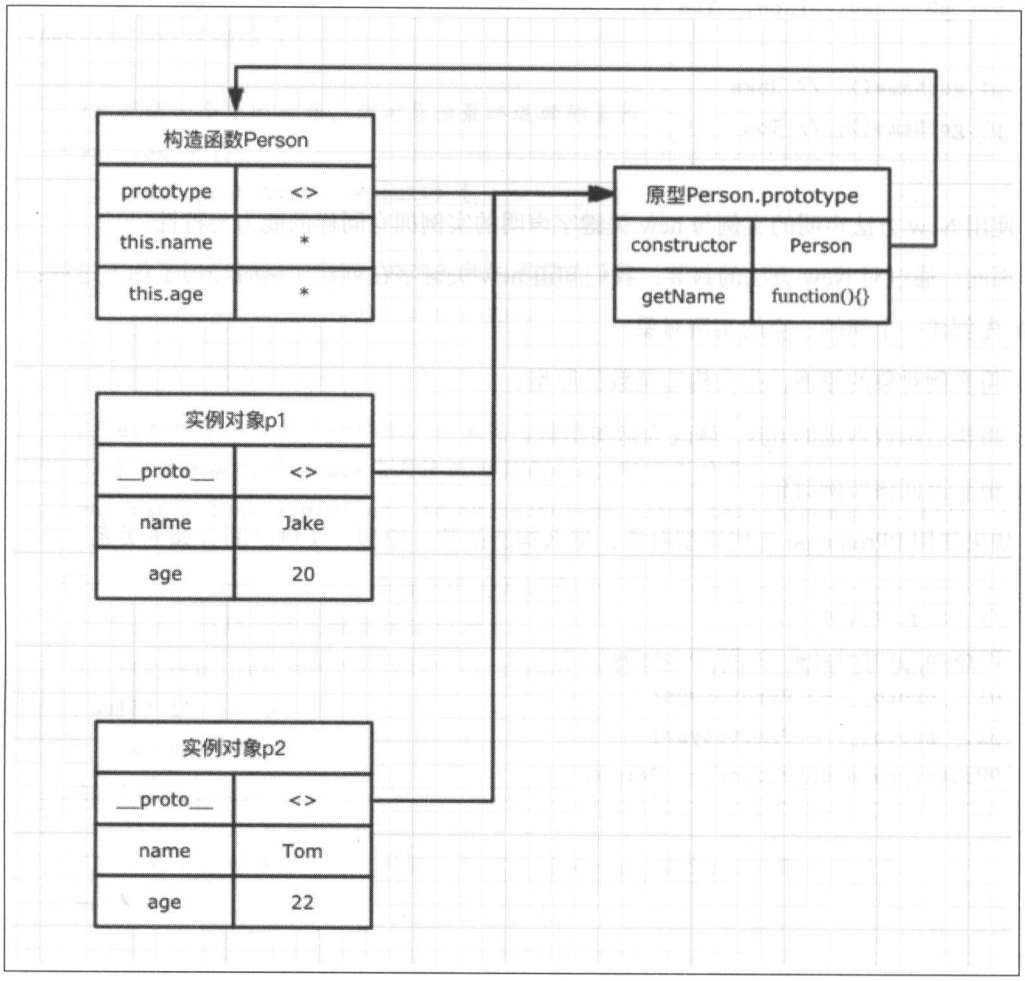


图 9-1 构造函数、原型、实例之间的关系

从前面的分析可以看出，构造函数的 `prototype` 与所有实例的`__proto__`都指向原型对象。而原型对象的 `constructor` 则指向构造函数。

因为在构造函数中声明的变量与方法只属于当前实例，因此我们可以将构造函数中声明的属性与方法称为该实例的私有属性与方法，它们只能被当前实例访问。

而原型中的方法与属性能够被所有的实例访问，因此我们将原型中声明的属性与方法称为公有属性与方法。

与在原型中添加一个方法不同，当在构造函数中声明一个方法时，每创建一个实例，该方法

都会被重新创建一次。而原型中的方法仅仅只会被创建一次（这也是我们称其为私有方法的原因之一）。

因此在构造函数中，声明私有方法会消耗更多的内存空间。

如果构造函数中声明的私有方法/属性与原型中的公有方法/属性重名，那么会优先访问私有方法/属性，如下例所示。

```
function Person(name) {
    this.name = name;
    this.getName = function() {
        return this.name + '，你正在访问私有方法。'
    }
}

Person.prototype.getName = function() {
    return this.name;
}

var p1 = new Person('Tom', 20);
p1.getName();  // Tom，你正在访问私有方法
```

在这个例子中，同时在构造函数与原型中都声明了一个同名方法 getName。运行的结果显示，原型中的方法并没有被访问。

可以通过 in 来判断一个对象是否拥有某一个方法/属性，无论该方法/属性是否公有。

```
// 接上例中创建的p1实例
console.log('name' in p1);    // true
console.log('getName' in p1); // true
console.log('gender' in p1);  // false
```

我们常常使用 in 的这种特性来判断当前页面所处的环境是否在移动端。

```
// 接上例中创建的p1实例
// 特性检测，只有移动端环境才支持touchstart事件
var isMobile = 'ontouchstart' in document;
```

9.1.4 更简单的原型写法

要想在原型上添加很多的方法与属性，则可以这样写。

```
function Person() {}
Person.prototype.getName = function() {}
Person.prototype.getAge = function() {}
Person.prototype.sayHello = function() {}
```

除此之外，还可以使用更为简洁的对象字面量的写法来添加原型方法。

```
Person.prototype = {
    constructor: Person,
    getName: function() {},
    getAge: function() {},
    sayHello: function() {}
}
```

使用对象字面量能够简化写法，但同时有一个需要特别注意的地方。当我们使用 Person.prototype = {} 时，其实是将 Person 的原型指向了一个新的对象 {}。如果不做特殊处理，那么将会导致原型对象丢失。因此在这个新的对象中，需要将它的 constructor 属性指向构造函数 Person，这样就重新建立了正确的对应关系，然后就可以放心大胆地使用了。

9.1.5 原型链

原型对象其实也是普通对象。

几乎所有的对象都可以是原型对象，也可以是实例对象，还可以是构造函数，甚至可以身兼多职。

当一个对象身兼多职时，它就可以被看作原型链中的一个节点。因此理解了原型后，再来理解原型链的概念就简单多了。

当一个对象 A 作为原型时，它有一个 constructor 属性指向它的构造函数，即 A.constructor。当一个对象 B 作为构造函数时，它有一个 prototype 属性指向它的原型，即 B.prototype。当一个对象 C 作为实例时，它有一个__proto__属性指向它的原型，即 C.__proto__。当想要判断

一个对象 foo 是否是构造函数 Foo 的实例时，可以使用 instanceof 关键字，返回一个 boolean 值。

```
foo instanceof Foo   // true: foo是Foo的实例。false: 不是。
```

当创建一个对象时，可以使用 new Object() 来创建。因此 Object 其实是一个构造函数，而其对应的原型 Object.prototype 则是原型链的终点。

```
foo instanceof Foo
// true: foo是Foo的实例。false: 不是Object.prototype.__proto__ === null

// 所有的函数与对象都有一个toString与valueOf方法，
// 就是来自于Object.prototype
Object.prototype.toString = function() {}
Object.prototype.valueOf = function() {}
```

当创建函数时，除可以使用 function 关键字外，还可以使用 Function 对象。

```
var add = new Function("a", "b", "return a + b");


// 等价于
var add = function(a, b) {
    return a + b
}
```

因此这里创建的 add 方法是一个实例，它对应的构造函数是 Function，它的原型是 Function.prototype。

```
add.__proto__ === Function.prototype // true
```

这里还有一个非常特殊的地方，Function 同时是 Function.prototype 的构造函数与实例。

```
add.__proto__ === Function.prototype  // true
Function.prototype.constructor === Function  // true
Function.prototype === Function.prototype // true
Function.__proto__ === Function.prototype  // true
add instanceof Function   // true  判断add是否是构造函数Function的实例
```

而与此同时，Function.prototype 不仅是 Function 的原型，还是 Object.prototype 的实例。

```
Function.prototype.__proto__ === Object.prototype
Function instanceof Function  // true
```

所有的函数都是构造函数 Function 的实例，所以有如下关系：

```
Object.__proto__ === Function.prototype   // true
Object instanceof Function   // true
```

因此，当我们随便创建一个函数 add 时，与它相关的原型链可以用图 9-2表示。

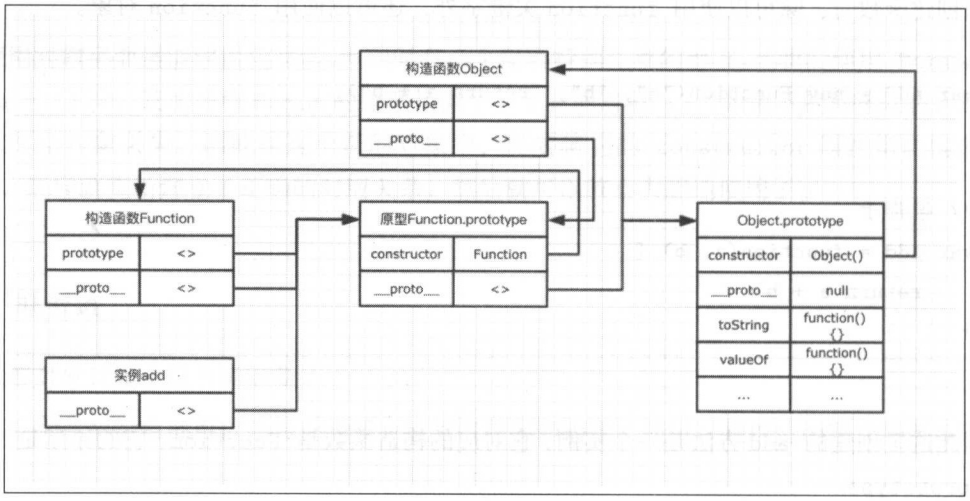


图 9-2　add 函数相关的原型链

对原型链上方法与属性的访问，与作用域链的访问类似，也是一个单向的查找过程。虽然 add 方法与 Object 并没有直接的关系，但是它们同处于一条原型链上，因此 add 可以根据原型链的特点访问 Object 上的方法。

```
function add() {}

add.toString === Object.toString  // true
```

需要注意的是，当构造函数与原型拥有同名的方法/属性时，如果用创建的实例访问该方法/属性，则优先访问构造函数的方法/属性。

```
function Person(name) {
    this.name = name;
    this.getName = function() {
        return 'name in Person.'
    }
}
Person.prototype.getName = function() {
    return 'name in Person.prototype.'
}

var p1 = new Person('alex');

// 原型上的方法被覆盖
p1.getName();   // name in Person
```

9.1.6 实例方法、原型方法、静态方法

在最开始分析 New 函数的实现时曾介绍过，构造函数中的 this 指向的是新创建的实例。因此在往 this 上添加方法与属性时，其实是在往新创建的实例上添加属性与方法，所以构造函数中的方法可称之为 **实例方法**。

而通过 prototype 添加的方法，将会挂载到原型对象上，因此称之为 **原型方法**。

那么什么是静态方法呢？我们在使用 jQuery 的时候，往往会使用一些构造函数直接调用，而非通过实例调用的方法。例如 $.each, $.ajax, $.extend, $.isArray 等，这些方法被直接挂载在构造函数上，我们称之为 **静态方法**。如果能够非常准确地区分实例、构造函数与原型，那么就应该能够想到，静态方法不能通过实例访问，只能通过构造函数来访问。

```
function Foo() {
    this.bar = function() {
        return 'bar in Foo';  // 实例方法
    }
}

Foo.bar = function() {
    return 'bar in static'   // 静态方法
}

Foo.prototype.bar = function() {
    return 'bar in prototype'  // 原型方法
}
```

静态方法又称为工具方法，常用来实现一些常用的，与具体实例无关的功能。例如遍历方法each。

9.1.7 继承

假设原型链的终点 Object.prototype 为原型链的 E（end）端，原型链的起点为 S（start）端。通过前面原型链的学习我们知道，处于 S 端的对象，可以通过 S→E 的单向查找，访问原型链上的所有方法与属性。这给继承提供了理论基础，我们只需在 S 端添加新的对象，新的对象就能够通过原型链访问到父级的方法与属性。因此要想实现继承，则是一件非常简单的事情。

因为封装一个对象是由构造函数与原型共同组成的，所以继承也被分为有构造函数的继承与原型的继承两种。

假设已经封装好了一个父类对象 Person，如下所示。

```
var Person = function(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.getName = function() {
    return this.name;
```

```
}

Person.prototype.getAge = function() {
    return this.age;
}
```

构造函数的继承比较简单，可以借助 call/apply 来实现。假设想要通过继承封装一个 Student 的子类对象，那么构造函数的实现如下所示。

```
var Student = function(name, age, grade) {
    // 通过call方法还原Person构造函数中的所有处理逻辑
    Student.call(Person, name, age);
    this.grade = grade;
}

// 等价于
var Student = function(name, age, grade) {
    this.name = name;
    this.age = age;
    this.grade = grade;
}
```

原型的继承则需要一点思考。首先应该考虑，如何将子类对象的原型加到原型链中？其实只需让子类对象的原型成为父类对象的一个实例，然后通过__proto__访问父类对象的原型，这样就继承了父类原型中的方法与属性了。

可以先封装一个方法，该方法会根据父类对象的原型创建一个实例，该实例即为子类对象的原型。

```
function create(proto, options) {
    // 创建一个空对象
    var tmp = {};

    // 让这个新的空对象成为父类对象的实例
    tmp.__proto__ = proto;
```

```
    // 传入的方法都挂载到新对象上，新对象将作为子类对象的原型
    Object.defineProperties(tmp, options);
    return tmp;
}
```

在简单封装了 create 对象之后，就可以使用该方法来实现原型的继承了。

```
Student.prototype = create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
    getGrade: {
        value: function() {
            return this.grade
        }
    }
})
```

下面来验证这里实现的继承是否正确。

```
var s1 = new Student('ming', 22, 5);

console.log(s1.getName());  // ming
console.log(s1.getAge());   // 22
console.log(s1.getGrade()); // 5
```

全部都能正常访问，在 ECMAScript5 中直接提供了一个 Object.create 方法来完成上面封装的 create 的功能，因此可以直接使用 Object.create。

```
Student.prototype = create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
    getGrade: {
```

```
        value: function() {
            return this.grade
        }
    }
})
```

完整代码如下。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.getName = function() {
    return this.name
}
Person.prototype.getAge = function() {
    return this.age;
}

function Student(name, age, grade) {
    // 构造函数继承
    Person.call(this, name, age);
    this.grade = grade;
}

// 原型继承
Student.prototype = Object.create(Person.prototype, {
    // 不要忘了重新指定构造函数
    constructor: {
        value: Student
    }
    getGrade: {
        value: function() {
            return this.grade
        }
    }
```

```
})

var s1 = new Student('ming', 22, 5);

console.log(s1.getName());  // ming
console.log(s1.getAge());   // 22
console.log(s1.getGrade()); // 5
```

9.1.8 属性类型

在上面的继承实现中，使用了一个大家可能不太熟悉的方法：defineProperties，并且在定义 getGrade 时使用了一个很奇怪的方式。

```
getGrade: {
    value: function() {
        return this.grade
    }
}
```

这其实是对象中的属性类型。通常给对象添加一个属性时，直接使用 object.param 的方式就可以了，或者直接在对象中挂载。

```
var person = {
    name: 'TOM'
}
```

在 ECMAScript5 中，对每个属性都添加了几个属性类型，用来描述这些属性的特点，具体如下。

- **configurable**：表示该属性是否能被 delete 删除。当其值为 false 时，其他的特性也不能被改变。默认值为 true。
- **enumerable**：是否能枚举。即是否能被 for-in 遍历。默认值为 true。
- **writable**：是否能修改值，默认为 true。

◎ **value**：该属性的具体值是多少，默认为 undefined。

◎ **get**：当通过 person.name 访问 name 的值时，get 将被调用。该方法可以自定义返回的具体值是多少，get 的默认值为 undefined。

◎ **set**：当通过 person.name = 'Jake' 设置 name 的值时，set 方法将被调用。该方法可以自定义设置值的具体方式。set 的默认值为 undefined。

需要注意的是，不能同时设置 value、writable 与 get、set 的值。

可以通过 Object.defineProperty 方法来修改这些属性类型。

下面就通过几个简单的例子来演示这些属性类型的具体表现。

configurable

```
// 用普通的方式给person对象添加一个name属性，值为TOM
var person = {
    name: 'TOM'
}

// 使用delete删除该属性
delete person.name;  // 返回true，表示删除成功

// 通过Object.defineProperty重新添加name属性
// 设置name的属性类型的configurable为false，表示不能再用delete删除
Object.defineProperty(person, 'name', {
    configurable: false,
    value: 'Jake'  // 设置name属性的值
})

// 再次delete，已经不能删除了
delete person.name   // false

console.log(person.name)    // 值为Jake

// 试图改变value
person.name = "alex";
console.log(person.name) // Jake 改变失败
```

enumerable

```
var person = {
    name: 'TOM',
    age: 20
}

// 使用for-in枚举person的属性
var params = [];

for(var key in person) {
    params.push(key);
}

// 查看枚举结果
console.log(params);  // ['name', 'age']

// 重新设置name属性的类型，让其不可被枚举
Object.defineProperty(person, 'name', {
    enumerable: false
})

var params_ = [];
for(var key in person) {
    params_.push(key)
}

// 再次查看枚举结果
console.log(params_); // ['age']
```

writable

```
var person = {
    name: 'TOM'
}

// 修改name的值
person.name = 'Jake';
```

```
// 查看修改结果
console.log(person.name); // Jake 修改成功

// 设置name的值不能被修改
Object.defineProperty(person, 'name', {
    writable: false
})

// 再次试图修改name的值
person.name = 'alex';

console.log(person.name); // Jake 修改失败
```

value

```
var person = {}

// 添加一个name属性
Object.defineProperty(person, 'name', {
    value: 'TOM'
})

console.log(person.name)  // TOM
```

get/set

```
var person = {}

// 通过get与set自定义访问、设置name属性
Object.defineProperty(person, 'name', {
    get: function() {
        // 一直返回TOM
        return 'TOM'
    },
    set: function(value) {
```

```
        // 设置name属性时，返回该字符串，value为新值
        console.log(value + ' in set');
    }
})

// 第一次访问name，调用get
console.log(person.name)    // TOM

// 尝试修改name值，此时set方法被调用
person.name = 'alex'    // alex in set

// 第二次访问name，还是调用get
console.log(person.name) // TOM
```

注意：请尽量同时设置 get 和 set。如果、只设置了 get，那么将无法设置该属性值。如果只设置了 set，那么将无法读取该属性的值。

Object.defineProperty 只能设置一个属性的属性特性，当想要同时设置多个属性的特性时，需要使用之前介绍过的 Object.defineProperties。

```
var person = {}

Object.defineProperties(person, {
    name: {
        value: 'Jake',
        configurable: true
    },
    age: {
        get: function() {
            return this.value || 22
        },
        set: function(value) {
            this.value = value
        }
    }
})
```

```
person.name   // Jake
person.age    // 22
```

9.1.9 读取属性的特性值

可以使用 Object.getOwnPropertyDescriptor 方法读取某一个属性的特性值。

```
var person = {}

Object.defineProperty(person, 'name', {
    value: 'alex',
    writable: false,
    configurable: false
})

var descripter = Object.getOwnPropertyDescriptor(person, 'name');

console.log(descripter);  // 返回结果如下

descripter = {
    configurable: false,
    enumerable: false,
    value: 'alex',
    writable: false
}
```

9.2 jQuery 封装详解

前几年学习前端时，大家都非常热衷于研究 jQuery 源码。然而随着前端的发展，以及另外几种前端框架的崛起，大家对于 jQuery 的热情降低了很多，但是许多从 jQuery 中学到的技巧在实践中仍然非常好用，简单地了解 jQuery 有助于我们更加深入地理解 JavaScript。

这里就把 jQuery 的实现作为一个学习案例，帮助我们进一步掌握面向对象的使用，也为进一步学习 jQuery 源码做一个铺垫，算是抛砖引玉吧。

使用 jQuery 时，我们通常会这样写：

```
// 声明一个jQuery实例
$('.target')

// 获取元素的css属性
$('.target').css('width')

// 获取元素的微信信息
$('target').offset()
```

是不是与普通的对象实例不太一样，new 关键字去哪里了，$ 符号又是什么？带着这些疑问，一起来实现一个简化版的 jQuery 库吧。

一个库就是一个单独的模块，因此应使用自执行函数的方式模拟一个模块。

```
(function() {
    // do something
});
```

既然能够在全局直接调用 jQuery，则说明 jQuery 被挂载在了全局对象上。因此当我们在模块中对外提供接口时，可以采取 window.jQuery 的方式。

```
var jQuery = function() {}

// ...

window.jQuery = jQuery
```

我们在使用过程中，并没有使用 jQuery，而是使用了 $，其实只是多加了一个赋值操作。

```
window.$ = window.jQuery = jQuery
```

在使用过程中直接使用 $，其实相当于直接调用构造函数 jQuery 创建了一个实例，而没有使用 new。但是创建一个实例时，new 关键字是必不可少的，由此说明 new 的操作被放在了 jQuery 方法中来实现，而 jQuery 并不是真正的构造函数。

通过前面的学习我们知道，每一个函数都可能是任何角色，jQuery 内部的实现正是利用了这一点，在具体实现时，改变了内部某些函数的 prototype 指向。下面先来看看实现代码，再来具体分析。

```
(function(ROOT) {

    // 构造函数
    var jQuery = function(selector) {
        // 在该方法中直接返回new创建的实例，
        // 因此这里的init才是真正的构造函数
        return new jQuery.fn.init(selector);
    }

    jQuery.fn = jQuery.prototype = {
        constructor: jQuery,
        version: '1.0.0',
        init: function(selector) {
            var elem, selector;
            elem = document.querySelector(selector);
            this[0] = elem;

            // 在jQuery中返回的是一个由所有原型属性方法组成的数组，
            // 这里做了简化，直接返回this即可
            return this;
        },

        // 在原型上添加一堆方法
        toArray: function() {},
        get: function() {},
        each: function() {},
        ready: function() {},
        first: function() {},
        slice: function() {}
        // ... more
    }

    // 让init方法的原型指向jQuery的原型
```

```
    jQuery.fn.init.prototype = jQuery.fn;

    ROOT.jQuery = ROOT.$ = jQuery;
})(window);
```

在上面的实现中，首先在jQuery构造函数中声明了一个fn属性，并将其指向了原型 jQuery.prototype。随后在原型对象中添加了 init 方法。

```
jQuery.fn = jQuery.prototype = {
    init: function() {}
}
```

之后又将 init 的原型指向了 jQuery.prototype。

```
jQuery.fn.init.prototype = jQuery.fn;
```

而在构造函数 jQuery 中，则返回了 init 的实例对象。

```
var jQuery = function(selector) {
    return new jQuery.fn.init(selector);
}
```

最后对外暴露接口时，将字符 $ 与方法 jQuery 对等起来。

```
ROOT.jQuery = ROOT.$ = jQuery;
```

因此当使用 $('#test') 创建一个 jQuery 实例时，实际上调用的是 jQuery('#test') 创建的一个 init 实例。这里正在构造函数的是原型中的 init 方法。

构造过程中的逻辑变化如图 9-3所示。

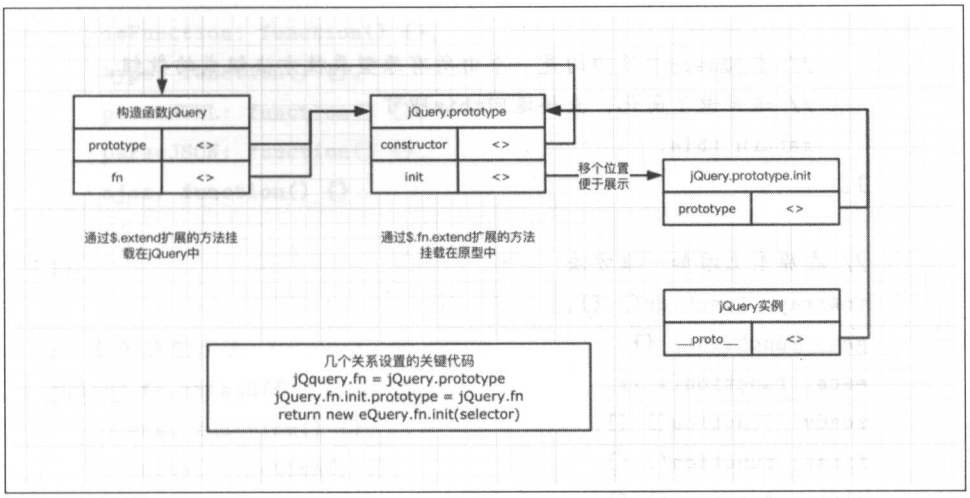


图 9-3 构造过程中的逻辑变化

扩展方法

jQuery 提供了两个扩展接口来帮助自定义 jQuery 的方法，通常称自定义的 jQuery 方法为 jQuery 插件。那么这两个扩展方法是如何实现的呢？在上面的基础上我们继续添加代码，如下所示。

```
(function(ROOT) {

    // 构造函数
    var jQuery = function(selector) {
        // 在该方法中直接返回new创建的实例，
        // 因此这里的init才是真正的构造函数
        return new jQuery.fn.init(selector);
    }

    jQuery.fn = jQuery.prototype = {
        constructor: jQuery,
        version: '1.0.0',
        init: function(selector) {
            var elem, selector;
            elem = document.querySelector(selector);
            this[0] = elem;
```

```
        // 在jQuery中返回的是一个由所有原型属性方法组成的数组，
        // 这里做了简化，直接返回this即可
        return this;
    },

    // 在原型上添加一堆方法
    toArray: function() {},
    get: function() {},
    each: function() {},
    ready: function() {},
    first: function() {},
    slice: function() {}
    // ... more
}

// 让init方法的原型指向jQuery的原型
jQuery.fn.init.prototype = jQuery.fn;

// 实现jQuery的两种扩展方法
jQuery.extend = jQuery.fn.extend = function(options) {
    // 在jQuery源码中根据参数不同进行不同的判断，这里假设只有一种方式
    var target = this;
    var copy;

    for(name in options) {
        copy = options[name];
        target[name] = copy;
    }
    return target;
}

// jQuery利用上面实现的扩展机制，添加了许多方法

// 添加静态扩展方法，即工具方法
jQuery.extend({
```

```
        isFunction: function() {},
        type: function() {},
        parseHTML: function() {},
        parseJSON: function() {},
        ajax: function() {}
        // ...
    })

    // 添加原型方法
    jQuery.fn.extend({
        queue: function() {},
        promise: function() {},
        attr: function() {},
        prop: function() {},
        addClass: function() {},
        removeClass: function() {},
        val: function() {},
        css: function() {}
        // ...
    })

    ROOT.jQuery = ROOT.$ = jQuery;
})(window);
```

在上面的代码中，我们通过下面的方式简单实现了两个扩展方法。

```
jQuery.extend = jQuery.fn.extend = function(options) {

    // 在jQuery源码中，会根据参数的不同进行很多判断，
    // 而这里直接用了一种方式，所以就不用判断了
    var target = this;
    var copy;

    for(name in options) {
        copy = options[name];
        target[name] = copy;
```

```
    }
    return target;
}
```

要理解它的实现，首先要明确知道内部 this 的指向。相信学习过前面的内容后对于 this 的指向已经非常明确了。传入的参数 options 对象是一个 key-value 模式的对象。我们可以通过 for in 遍历 options，将 key 作为新的属性，value 作为该属性对应的新方法，分别添加到 jQuery 与 jQuery.fn 中。

也就是说，当通过 $.extend 扩展 jQuery 时，方法被添加到了静态方法中；而通过 $.fn.extend 扩展 jQuery 时，方法被添加到了原型对象中。

静态方法可以直接调用，因此也被称为工具方法，代码如下。

```
$.ajax()
$.isFunction()
$.each()
//...
```

原型方法必须通过声明的实例才能调用。

```
$('#test').css();
$('#test').attr();

// ...
```

9.3 封装一个拖曳对象

基础概念的掌握并不是很难，但是要想真正地转化为自己的知识，则需要通过大量的实践才行。

1. 如何让一个 DOM 元素动起来

拖曳的本质就是让 DOM 元素能够跟着鼠标运动起来。

设想一下，在页面中仅有一个 class 名为 autumn 的 div 标签，它的基本样式如下：

```
<div class="autumn"></div>
```

```
.autumn {
    width: 20px;
    height: 20px;
    background-color: orange;
}
```

思考一下，让.autumn 运动（即让它的位置发生变化）可以通过哪些途径？

相信熟悉 CSS 的读者很快就能想到不少方法，例如，改变.autumn 的 margin 值；或者设置.autumn 的定位方式为 relative，修改其 left/top 属性；又或者直接修改它的 translate 值。

但是通常情况下我们不会采用修改 margin 值的方式让元素的位置发生改变，以避免对其他元素造成影响。

这里以修改 left 为例，一起来实现元素每被点击一次就右移 5 个像素的效果，代码如下。

```
<div class="autumn"></div>
```

```
var autumn = document.querySelector('.autumn');
autumn.style.position = 'relative';

autumn.addEventListener('click', function() {
    this.style.left = (this.offsetLeft + 5) + 'px';
}, false);
```

但是当页面所处的环境支持 CSS3 属性 translate 时，修改 left/top 会导致频繁的重排与回流，因此建议在处理元素运动时修改 translate 的值。

```
.autumn {
    transform: translateX(0px);
}
```

但是在使用 translate 时，就不得不面临一个兼容性的问题，不同浏览器的兼容写法有以下几种。

```
['transform', 'webkitTransform', 'MozTransform', 'msTransform', '
OTransform']
```

因此首先需要判断当前浏览器环境支持的 transform 属性是哪一种，方法如下。

```
// 获取当前浏览器支持的transform兼容写法
function getTransform() {
    var transform = '',
        divStyle = document.createElement('div').style,
        _transforms = ['transform', 'webkitTransform', 'MozTransform', '
        msTransform', 'OTransform'],
        i = 0,
        len = _transforms.length;

    for(; i < len; i++)  {
        if(_transforms[i] in divStyle) {
            // 找到之后立即返回，结束函数
            return transform = _transforms[i];
        }
    }

    // 如果没有找到，就直接返回空字符串
    return transform;
}
```

该方法用于获取当前浏览器支持的 transform 属性。如果返回空字符串，则表示该浏览器不支持 transform，这个时候就需要退而求其次，选择 left/top。

2. 如何获取元素的初始位置

为了获取元素的初始位置，需要声明一个专门用来获取元素样式的功能函数。获取元素样式的方法在 IE 中与其他浏览器中有所不同，所以需要一个兼容性的写法，代码如下。

```
function getStyle(elem, property) {
    // IE通过currentStyle来获取元素的样式，
    // 其他浏览器通过getComputedStyle来获取
    return document.defaultView.getComputedStyle ? document.defaultView.
    getComputedStyle(elem, false)[property] : elem.currentStyle[property
    ];
}
```

有了这个方法之后，我们就可以动手来实现一个获取元素位置的方法了，代码如下。

```
function getTargetPos(elem) {
    var pos = {x: 0, y: 0};
    var transform = getTransform();
    if(transform) {
        var transformValue = getStyle(elem, transform);
        if(transformValue == 'none') {
            elem.style[transform] = 'translate(0, 0)';
            return pos;
        } else {
            var temp = transformValue.match(/-?\d+/g);
            return pos = {
                x: parseInt(temp[4].trim()),
                y: parseInt(temp[5].trim())
            }
        }
    } else {
        if(getStyle(elem, 'position') == 'static') {
            elem.style.position = 'relative';
            return pos;
        } else {
            var x = parseInt(getStyle(elem, 'left') ? getStyle(elem, '
            left') : 0);
            var y = parseInt(getStyle(elem, 'top') ? getStyle(elem, 'top
            ') : 0);
            return pos = {
                x: x,
```

```
                y: y
            }
        }
    }
}
```

在拖曳过程中，需要不停地设置目标元素的位置，这样它才能够移动起来，因此还需要声明一个设置元素位置的方法。

```
// pos = { x: 200, y: 100 }
function setTargetPos(elem, pos) {
    var transform = getTransform();
    if(transform) {
        elem.style[transform] = 'translate('+ pos.x +'px, '+ pos.y +'px)
        ';
    } else {
        elem.style.left = pos.x + 'px';
        elem.style.top = pos.y + 'px';
    }
    return elem;
}
```

有了这几个工具方法后，就可以使用更为完善的方式来实现上述要求的效果了，代码如下。

```
var autumn = document.querySelector('.autumn');

autumn.addEventListener('click', function() {
    var curPos = getTargetPos(this);
    setTargetPos(this, {
        x: curPos.x + 5,
        y: curPos.y
    });
}, false);
```

(1) 拖曳的原理

可以结合 mousedown、 mousemove、 mouseup 这三个事件来实现拖曳。

◎ mousedown：鼠标按下时触发。

◎ mousemove：鼠标移动时触发。

◎ mouseup：鼠标松开时触发。

在这些事件触发的回调函数中得到了一个事件对象，通过事件对象即可获取当前鼠标所处的精确位置。鼠标位置信息是实现拖曳的关键。

当鼠标按下（mousedown 触发）时，需要记住鼠标的初始位置与目标元素的初始位置。当鼠标移动时，目标元素也跟着移动，因此鼠标与目标元素的位置有如下关系：

移动后鼠标位置 – 鼠标初始位置 = 移动后目标元素位置 – 目标元素初始位置

如果鼠标位置的差值用变量 dis 来表示，那么目标元素的位置就等于：

移动后目标元素位置 =dis + 目标元素的初始位置

通过事件对象中提供的鼠标精确位置信息，在鼠标移动时可以轻易地计算出鼠标移动位置的差值，然后根据上面的关系，计算出目标元素的当前位置，这样拖曳就能够实现了。

(2) 代码实现

part1：准备工作。

```
// 获取目标元素对象
var autumn = document.querySelector('.autumn');

// 声明2个变量用来保存鼠标初始位置的x，y坐标
var startX = 0;
var startY = 0;

// 声明2个变量用来保存目标元素初始位置的x，y坐标
var sourceX = 0;
var sourceY = 0;
```

part2：功能函数。

由于前面已经给过代码，因此这里不再重复。

```
// 获取当前浏览器支持的transform兼容写法
function getTransform() {}

// 获取元素属性
function getStyle(elem, property) {}

// 获取元素的初始位置
function getTargetPos(elem) {}

// 设置元素的初始位置
function setTargetPos(elem, potions) {}
```

part3：声明三个事件的回调。

```
autumn.addEventListener('mousedown', start, false);

// 绑定在mousedown上的回调，event为传入的事件对象
function start(event) {
    // 获取鼠标初始位置
    startX = event.pageX;
    startY = event.pageY;

    // 获取元素初始位置
    var pos = getTargetPos(autumn);

    sourceX = pos.x;
    sourceY = pos.y;

    // 绑定
    document.addEventListener('mousemove', move, false);
    document.addEventListener('mouseup', end, false);
}

function move(event) {
    // 获取鼠标当前位置
    var currentX = event.pageX;
```

```
    var currentY = event.pageY;

    // 计算差值
    var distanceX = currentX - startX;
    var distanceY = currentY - startY;

    // 计算并设置元素当前位置
    setTargetPos(autumn, {
        x: (sourceX + distanceX).toFixed(),
        y: (sourceY + distanceY).toFixed()
    })
}

function end(event) {
    document.removeEventListener('mousemove', move);
    document.removeEventListener('mouseup', end);
    // do other things
}
```

至此，一个简单的拖曳就实现了。

(3) 使用面向对象进行封装

我们前面学习了面向对象的基础知识，下面就基于这些基础知识将上面实现的拖曳封装为一个拖曳对象。我们的目标是，只要声明一个拖曳实例，那么传入的目标元素就将自动具备可以被拖曳的功能。

在实际开发中，常常将一个对象放在一个 js 文件中，这个 js 文件将单独作为一个模块，利用各种模块的方式组织起来使用。当然，这里并没有复杂的模块交互，因为这个例子只需要一个模块即可。

为了避免变量污染，我们需要将模块放置在一个函数自执行方式模拟的块级作用域中。

```
(function() {
    ...
})();
```

在普通的模块组织中，只是将许多 js 文件单纯地压缩成一个 js 文件，此处的第一个分号是为了防止上一个模块的结尾不用分号而导致报错，因此必不可少。当然，使用 require 或者 ES6 模块等方式时不会出现这种情况。

我们知道，在封装一个对象的时候，可以将属性与方法放置在构造函数或者原型中，而在增加了自执行函数之后，又可以将属性和方法放置在模块的内部作用域。这是闭包的知识。

而我们面临的挑战是，如何合理地处理属性与方法的位置。

当然，每一个对象的情况都不一样，不能一概而论，因此我们需要清楚地知道这三种位置的特性，这样才能做出最适合的决定。

- ◎ 构造函数中：属性与方法为当前实例所单独拥有，只能被当前实例访问，并且每声明一个实例，其中的方法都会被重新创建一次。
- ◎ 原型中：属性与方法为所有实例共同拥有，可以被所有实例访问，新声明的实例不会重复创建方法。
- ◎ 模块作用域中：属性和方法不能被任何实例访问，但是能被内部方法访问，新声明的实例不会重复创建相同的方法。

对于方法的判断则比较简单，因为构造函数中的方法总是在声明一个新的实例时被重复创建，因此声明方法时应尽量避免出现在构造函数中。

如果你的方法中需要用到构造函数中的变量，或者想要公开，那么就需要放在原型中。

如果方法需要私有不被外界访问，那么就放置在模块作用域中。

对于属性应放置在什么位置很多时候很难做出正确的判断，因此很难给出一个准确的定义告诉你什么属性一定要放在什么位置，这需要在实际开发中不断地总结经验。但是总体来说，仍然要结合这三个位置的特性来做出最适合的判断。

- ◎ 如果属性值只能被实例单独拥有，比如 person 对象的 name，那只能属于某一个 person 实例；
- ◎ 又比如拖曳对象中，某一个元素的初始位置仅仅是这个元素的当前位置，那么这个属性适合放在构造函数中。
- ◎ 如果一个属性仅仅供内部方法访问，那么这个属性就适合放在模块作用域中。

关于面向对象，上面的几点必须认真思考。如果在封装时没有思考清楚，则很可能会遇到很多意想不到的 bug，所以建议大家结合自己的开发经验，多多思考，总结出自己的观点。

根据这些思考，大家可以自己尝试封装一下。在下面例子的注释中，笔者将自己的想法表达出来了。

```
(function() {
    // 这是一个私有属性，不需要被实例访问
    var transform = getTransform();

    function Drag(selector) {
        // 放在构造函数中的属性，被每一个实例所单独拥有
        this.elem = typeof selector == 'Object' ? selector : document.
        getElementById(selector);
        this.startX = 0;
        this.startY = 0;
        this.sourceX = 0;
        this.sourceY = 0;

        this.init();
    }


    // 原型
    Drag.prototype = {
        constructor: Drag,

        init: function() {
            // 初始时需要做哪些事情
            this.setDrag();
        },

        // 稍作改造，仅用于获取当前元素的属性，类似于getName
        getStyle: function(property) {
            return document.defaultView.getComputedStyle ? document.
            defaultView.getComputedStyle(this.elem, false)[property] :
            this.elem.currentStyle[property];
        },

        // 用来获取当前元素的位置信息，注意与之前的不同之处
        getPosition: function() {
            var pos = {x: 0, y: 0};
```

```
        if(transform) {
            var transformValue = this.getStyle(transform);
            if(transformValue == 'none') {
                this.elem.style[transform] = 'translate(0, 0)';
            } else {
                var temp = transformValue.match(/-?\d+/g);
                pos = {
                    x: parseInt(temp[4].trim()),
                    y: parseInt(temp[5].trim())
                }
            }
        } else {
            if(this.getStyle('position') == 'static') {
                this.elem.style.position = 'relative';
            } else {
                pos = {
                    x: parseInt(this.getStyle('left') ? this.getStyle
                    ('left') : 0),
                    y: parseInt(this.getStyle('top') ? this.getStyle
                    ('top') : 0)
                }
            }
        }

        return pos;
    },

    // 用来设置当前元素的位置
    setPostion: function(pos) {
        if(transform) {
            this.elem.style[transform] = 'translate('+ pos.x +'px, '+
             pos.y +'px)';
        } else {
            this.elem.style.left = pos.x + 'px';
            this.elem.style.top = pos.y + 'px';
        }
```

```
},

// 该方法用来绑定事件
setDrag: function() {
    var self = this;
    this.elem.addEventListener('mousedown', start, false);
    function start(event) {
        self.startX = event.pageX;
        self.startY = event.pageY;

        var pos = self.getPosition();

        self.sourceX = pos.x;
        self.sourceY = pos.y;

        document.addEventListener('mousemove', move, false);
        document.addEventListener('mouseup', end, false);
    }

    function move(event) {
        var currentX = event.pageX;
        var currentY = event.pageY;

        var distanceX = currentX - self.startX;
        var distanceY = currentY - self.startY;

        self.setPostion({
            x: (self.sourceX + distanceX).toFixed(),
            y: (self.sourceY + distanceY).toFixed()
        })
    }

    function end(event) {
        document.removeEventListener('mousemove', move);
        document.removeEventListener('mouseup', end);
        // do other things
```

```
            }
        }
    }

    // 私有方法，仅仅用来获取transform的兼容写法
    function getTransform() {
        var transform = '',
            divStyle = document.createElement('div').style,
            transformArr = ['transform', 'webkitTransform', 'MozTransform
            ', 'msTransform', 'OTransform'],

            i = 0,
            len = transformArr.length;

        for(; i < len; i++)  {
            if(transformArr[i] in divStyle) {
                return transform = transformArr[i];
            }
        }

        return transform;
    }

    // 一种对外暴露的方式
    window.Drag = Drag;
})();

// 使用：声明2个拖曳实例
new Drag('target');
new Drag('target2');
```

这样一个拖曳对象就封装完成了。

建议读者根据本书提供的思维方式，多多尝试封装一些组件，比如封装一个弹窗或封装一个循环轮播等。练得多了，面向对象就不再是问题了。这种思维方式在未来任何时候都能够用到。

4. 将拖曳对象扩展为一个 jQuery 插件

在前面的学习中我们知道，可以使用 $.extend 扩展 jQuery 工具方法，来使用 $.fn.extend 扩展原型方法。当然，这里的拖曳插件扩展为原型方法是最合适的。

```
// 通过扩展方法将拖曳扩展为jQuery的一个实例方法
(function ($) {
  $.fn.extend({
    canDrag: function () {
      new Drag(this[0]);
      return this;
      // 注意：为了保证jQuery所有的方法都能够链式访问，
      // 每一个方法的最后都需要返回this，即返回jQuery实例
    }
  })
})(jQuery);
```

这样就能够很轻松地让目标 DOM 元素具备拖曳能力了。

```
$('target').canDrag();
```

9.4 封装一个选项卡

选项卡在 Web 中的应用可以说是无处不在，因此必须要掌握它的实现。

通常情况下，选项卡由两部分组成。一部分是头部，它包含一堆按钮，每一个按钮对应不同的页面，按钮包括选中与无法选中两种状态。另一部分则由一些具体的页面组成，当我们单击按钮时，就切换到对应的页面。

注意：如果每个页面中包含的是根据动态加载的数据渲染出来的界面，那么通常只会有一个页面，单击按钮时重新加载数据并重新渲染页面。

首先在 HTML 中将这两部分代码写出来。

```
<div class="box" id="tab_wrap">
    <ul class="tab_options">
```

```
        <li data-index="0" class="item active">tab1</li>
        <li data-index="1" class="item">tab2</li>
        <li data-index="2" class="item">tab3</li>
        <li data-index="3" class="item">tab4</li>
    </ul>
    <div class="tab_content">
        <div class="item_box active">tab box 1</div>
        <div class="item_box">tab box 2</div>
        <div class="item_box">tab box 3</div>
        <div class="item_box">tab box 4</div>
    </div>
</div>
```

并简单写一些 CSS 代码。

```
body {
    margin: 0;
}

ul, li {
    list-style: none;
    padding: 0;
}

.box {
    max-width: 400px;
    margin: 10px auto;
    background: #efefef;
}

.box .tab_options {
    height: 40px;
    display: flex;
    justify-content: space-around;
    border-bottom: 1px solid #ccc;
}
```

```
.box .tab_options li {
    line-height: 40px;
    cursor: pointer;
}
.box .tab_options li.active {
    color: red;
    border-bottom: 1px solid red;
}

.box .tab_content {
    min-height: 400px;
    position: relative;
}

.box .tab_content .item_box {
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    display: none;
    text-align: center;
}
.box .tab_content .item_box.active {
    display: block;
}
```

选项卡的实现原理非常简单，在HTML代码中，每一个头部按钮都保存了一个 data-index 属性，这个属性告诉我们这是第几个按钮，这个值同时也对应第几页。因此只需声明一个 index 变量来保存当前页的序列，并在单击时把当前页的值修改为 data-index 的值就可以了。与此同时，把当前按钮修改为选中状态，其他按钮修改为未选中状态，让当前页显示，其他页隐藏即可。JavaScript 代码如下。

```
var tabHeader = document.querySelector('.tab_options');
var items = tabHeader.children;
var tabContent = document.querySelector('.tab_content');
var itemboxes = tabContent.children;
```

```
var index = 0;

tabHeader.addEventListener('click', function(e) {
    var a = [].slice.call(e.target.classList).indexOf('item');
    if (a > -1 && index != e.target.dataset.index) {
        items[index].classList.remove('active');
        itemboxes[index].classList.remove('active');
        index = e.target.dataset.index;
        items[index].classList.add('active');
        itemboxes[index].classList.add('active');
    }
}, false);
```

此时假设要新增一个功能，即在 HTML 中新增两个按钮，单击它们就可以分别切换到上一页或下一页，该如何操作呢？

```
<button class="next">Next</button>
<button class="prev">Prev</button>
```

为了更直观地实现这个功能，下面尝试将选项卡封装为一个对象，代码如下。

```
!function(ROOT) {
    var index = 0;
    function Tab(elem) {
        this.tabHeader = elem.firstElementChild;
        this.items = this.tabHeader.children;
        this.tabContent = elem.lastElementChild;
        this.itemboxes = this.tabContent.children;
        this.max = this.items.length - 1;

        this.init();
    }

    Tab.prototype = {
        constructor: Tab,
```

```
init: function() {
    this.tabHeader.addEventListener('click', this.clickHandler.
    bind(this), false);
},
clickHandler: function(e) {
    var a = [].slice.call(e.target.classList).indexOf('item');
    if (a > -1) {
        this.switchTo(e.target.dataset.index);
    }
},
switchTo: function(i) {
    if (i == index) {
        return;
    }
    this.items[index].classList.remove('active');
    this.itemboxes[index].classList.remove('active');
    index = i;
    this.items[index].classList.add('active');
    this.itemboxes[index].classList.add('active');
},
next: function() {
    var tgIndex = 0;
    if (index >= this.max) {
        tgIndex = 0;
    } else {
        tgIndex = index + 1;
    }
    this.switchTo(tgIndex);
},
pre: function() {
    var tgIndex = 0;
    if (index == 0) {
        tgIndex = this.max;
    } else {
        tgIndex = index - 1;
    }
```

```
            this.switchTo(tgIndex);
        },
        getIndex: function() {
            return index;
        }
    }

    ROOT.Tab = Tab;
}(window);
```

在上面的代码中，将切换功能封装成了基础的 switchTo 方法，它接收一个表示页面序列的参数，只要我们调用这个方法，就能够切换到对应的页面。

因此基于这个基础方法，就能够很容易地扩展出下一页 next 与上一页 pre 的方法。

再次使用时，就很简单了，代码如下。

```
var tab = new Tab(document.querySelector('#tab_wrap'));

document.querySelector('.next').addEventListener('click', function() {
    tab.next();
    console.log(tab.getIndex());
}, false);

document.querySelector('.prev').addEventListener('click', function() {
    tab.pre();
    console.log(tab.getIndex());
}, false);
```

这样，一个简单的选项卡功能就完成了，并且拥有了一些简单的扩展功能。从使用中可以看到，封装好之后，想要实现什么功能，只需调用对应的接口即可。面向对象封装之后，代码变得非常简单直观。

需要注意的是，为了考验大家对于闭包的理解，在上面的代码中，故意留了一个与变量 index 相关的坑，大家只要在页面中创建两个选项卡，就能够发现问题。建议结合闭包思考一下为什么会出现这样的情况，并做正确的调整。

9.5 封装无缝滚动

无缝滚动指的是几个元素循环滚动，视觉效果就像是有无穷无尽的元素一样。

实现的原理也很简单，首先对容器元素进行滚动操作。子元素在容器元素中依次排列，并且将子元素复制一份，放在同一个容器元素中，这样就实现了首尾相接。当最后一个子元素滚动过临界点的时候，将容器元素的位置拉回初始位置，然后重复滚动操作即可。

首先写好 HTML 代码与 CSS 代码。

```
<div id="scroll_area">
    <div class="scroll_body">
        <div class="item">1</div>
        <div class="item">2</div>
        <div class="item">3</div>
        <div class="item">4</div>
        <div class="item">5</div>
    </div>
</div>

<div class="buttons">
    <button class="left">move Left</button>
    <button class="right">move right</button>
    <button class="stop">stop</button>
</div>
```

```
body {
    margin: 0;
}

#scroll_area {
    width: 400px;
    height: 100px;
    border-top: 1px solid #ccc;
    border-bottom: 1px solid #ccc;
    margin: 20px auto;
    overflow: hidden;
```

```
}

.scroll_body {
    display: flex;
    font-size: 30px;
    height: 100%;
    position: relative;
    left: 0;
}

.scroll_body .item {
    flex: 1;
    display: flex;
    align-items: center;
    justify-content: center;
}

.buttons {
    width: 400px;
    margin: 20px auto;
}
```

从 HTML 代码中可以看出，滚动区域元素 scroll_area 固定，容器元素 scroll_body 在滚动区域中滚动，滚动效果就是通过定时器每秒钟移动 1 个像素来实现的。

代码实现如下。

```
var lastTime = 0,
    nextFrame = window.requestAnimationFrame       ||
                window.webkitRequestAnimationFrame ||
                window.mozRequestAnimationFrame    ||
                window.msRequestAnimationFrame     ||
                function(callback) {
                    var currTime = + new Date,
                        delay = Math.max(1000/60, 1000/60 - (currTime -
                        lastTime));
                    lastTime = currTime + delay;
```

```
                return setTimeout(callback, delay);
            },
    cancelFrame = window.cancelAnimationFrame              ||
                  window.webkitCancelAnimationFrame        ||
                  window.webkitCancelRequestAnimationFrame ||
                  window.mozCancelRequestAnimationFrame    ||
                  window.msCancelRequestAnimationFrame     ||
                  clearTimeout;

var area = document.querySelector('#scroll_area');
var areaWidth =  area.offsetWidth;

var scrollBody = area.querySelector('.scroll_body');
var itemWidth = areaWidth/(scrollBody.children.length);

scrollBody.style.width = areaWidth * 2 + 'px';
scrollBody.innerHTML = scrollBody.innerHTML + scrollBody.innerHTML;

var targetPos = areaWidth;
var scrollX = 0;
var timer = null;

function ani() {
    cancelFrame(timer);
    timer = nextFrame(function() {
        scrollX -= 1;

        if (-scrollX >= targetPos) {
            scrollX = 0;
        }

        scrollBody.style.left = scrollX + 'px';
        ani();
    })
}
```

```
ani();
```

对于初学者来说，一个比较难以理解的地方就是 nextFrame 与 cancelFrame 的声明，这是一个类似于定时器的 setTimeout 的兼容性写法。requestAnimationFrame 是一个在 HTML5 中用于实现动画效果的 API。

另一个则是函数 ani 的递归调用，通过递归调用的方式来代替 setInterval 是一个非常棒的解决方案。

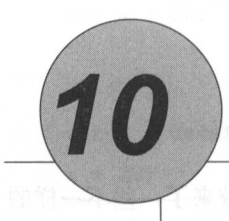
当然，也可以通过面向对象的方式来扩展控制滚动方向、停止滚动等接口，代码如下。

```
(function(ROOT) {
    var lastTime = 0,
        nextFrame = window.requestAnimationFrame       ||
                    window.webkitRequestAnimationFrame ||
                    window.mozRequestAnimationFrame    ||
                    window.msRequestAnimationFrame     ||
                    function(callback) {
                        var currTime = + new Date,
                            delay = Math.max(1000/60, 1000/60 - (currTime
                             - lastTime));
                        lastTime = currTime + delay;
                        return setTimeout(callback, delay);
                    },
        cancelFrame = window.cancelAnimationFrame              ||
                      window.webkitCancelAnimationFrame        ||
                      window.webkitCancelRequestAnimationFrame ||
                      window.mozCancelRequestAnimationFrame    ||
                      window.msCancelRequestAnimationFrame     ||
                      clearTimeout;


    var timer = null;

    function Scroll(elem) {
        this.elem = elem;
        this.areaWidth = elem.offsetWidth;
```

```
    this.scrollBody = elem.querySelector('.scroll_body');
    this.itemWidth = this.areaWidth/this.scrollBody.children.length;
    this.scrollX = 0;
    this.targetPos = this.areaWidth;
    this.init();
}

Scroll.prototype = {
    constructor: Scroll,
    init: function() {
        this.scrollBody.style.width = this.areaWidth * 2 + 'px';
        this.scrollBody.innerHTML = this.scrollBody.innerHTML + this.
        scrollBody.innerHTML;
        this.moveRight();
    },
    moveLeft: function() {
        var self = this;
        cancelFrame(timer);
        timer = nextFrame(function() {
            self.scrollX -= 1;
            if (-self.scrollX >= self.targetPos) {
                self.scrollX = 0;
            }
            self.scrollBody.style.left = self.scrollX + "px";
            self.moveLeft();
        })
    },
    moveRight: function() {
        var self = this;
        cancelFrame(timer);
        timer = nextFrame(function() {
            self.scrollX += 1;
            if (self.scrollX >= 0) {
                self.scrollX = -self.targetPos;
            }
            self.scrollBody.style.left = self.scrollX + "px";
```

```
                self.moveRight();
            })
        },
        stop: function() {
            cancelFrame(timer);
        }
    }

    ROOT.Scroll = Scroll;
})(window);

var scroll = new Scroll(document.querySelector('#scroll_area'));

var left_btn = document.querySelector('.left');
var right_btn = document.querySelector('.right');
var stop_btn = document.querySelector('.stop');

left_btn.onclick = function() {
    scroll.moveLeft();
}
right_btn.onclick = function() {
    scroll.moveRight();
}
stop_btn.onclick = function() {
    scroll.stop();
}
```

实例的主要目的就是通过源码学习相关的知识点，所以建议大家一定要自己动手尝试，仔细思考相关知识点在实例中的运用。



ES6 与模块化

ES6 是 ECMAScript 6 的简称，是 ECMAScript 的新标准。

看过《JavaScript 高级编程》的读者应该知道，JavaScript 实际上是由 DOM、BOM 和 ECMAScript 三部分组成的。在此前的学习中，用到的都是 ES5 的语法标准，也是我们最常见的标准。

之所以把 ES6 单独拿出来学习，是因为与 ES5 相比，它在很大程度上改变了我们的编程习惯。虽然从某种程度上来说，ES6 的出现增加了我们的学习成本，但是对于前端开发带来的改变却是非常令人惊喜的。因此在实践中，大多数前端团队已经开始全面使用 ES6 进行开发了。

ES6 是学习前端必须要掌握的技能。网络上越来越多的优质资源也开始使用 ES6 进行知识分享。虽然目前来说，并不是所有的浏览器都能够直接支持所有的 ES6 特性，但是在开发中，借助 babel 提供的编译工具，即可将 ES6 转化为 ES5，这也极大地推动了前端团队对于 ES6 的接受。

ES6 于 2015 年 6 月正式发布，因此又被称为 ES2015，并在 2016 年进行了修改。在未来会每年命名一个版本，如 2017 年发布的版本，会称为 ES7，或者 ES2017，依次类推。

对于大多数常用的 ES6 基础知识，目前最新版本的 Chrome 都已经全部支持，因此我们只需将 Chrome 更新到最新版本，就可以直接学习了。不过对于部分知识，例如模块化 modules，则需要通过构建工具才能够学习。

10.1 常用语法知识

可以使用 babel 官网提供的在线编译工具，将 ES6 编译为对应的 ES5 代码。仔细观察两者之间的不同，有助于我们更加了解 ES6 的知识。

http://babeljs.io/repl/

1. 新的变量声明方式 let/const

在 ES5 中，使用 var 来声明一个变量。在 ES6 中，新的变量声明方式带来了一些不一样的特性，其中最重要的就是具备了块级作用域并且不再有变量提升。

下面通过两个简单的例子来说明。

```
{
    let a = 20;
}

console.log(a);  // a is not defined
```

这段代码会被编译为：

```
{
    let _a = 20;
}

console.log(a);  // a is not defined
```

```
// demo02
// ES5
console.log(a);   // undefined
var a = 20;

// ES6
console.log(a); // a is not defined
let a = 20;
```

通过 demo02 的例子可以看出，变量提升好像确实不存在了，那么这是什么原因导致的呢？

我们知道，通过 var 声明的变量，会分别经历两个步骤，首先给变量赋值一个 undefined，并将整个操作提升到作用域的前面，因此会有：

```
var a = 20;

// 等同于
var a = undefined;  // 该操作提升
a = 20;
```

但是当通过 let/const 声明变量时，其实提升的操作仍然存在，但是并不会给这个变量赋值为 undefined。也就是说，虽然声明提前了，但是该变量并没有任何引用，所以当进行如下操作时，会报 referenceError，即引用错误。

```
console.log(a);   // ReferenceError: a is not defined
let a = 20;
```

由于不会默认赋值为 undefined，加上 let/const 存在自己的作用域，因此会出现一个叫作**暂时性死区**的现象，例如：

```
var a = 20;
if (true) {
    console.log(a);  // ReferenceError: a is not defined
    let a = 30;
}
```

此处虽然 a 已经声明过了，但是由于存在暂时性死区，因此仍然无法正常访问。我们在自己的代码中，要注意这些异常，尽量将声明主动放置在代码的前面。

要想使用 ES6，就需要全面使用 let/const 来替换之前非常常用的 var，那么什么时候用 let，什么时候用 const 呢？

一般来说，声明一个引用可以被改变的变量时用 let，声明一个引用不能被改变的变量时用 const。

例如，使用 let 声明一个值总是会变的变量。

```
let a = 20;
a = 30;

a = 40;
console.log(a);
```

这里需要注意的是，a 值改变的原因，是引用改变了。

可以使用 const 来声明一个常量。

```
const PI = 3.1415;
const MAX_LENGTH = 100;

// 试图改变引用
PI = 3; // Untaught TypeError: Assignment to constant variable
```

除此之外，当声明一个引用类型的数据时，也会使用 const。尽管可能会改变该数据的值，但是必须保持它的引用不变。

```
const a = [];
a.push(1);
console.log(a); // [1]
```

```
const b = {}
b.max = 20;
b.min = 0;
console.log(b); // { max: 20, min: 0 }
```

观察下面这个例子，请思考一下能不能这样用？思考之后，再在浏览器中运行试试看吧。

```
const arr = [1, 2, 3, 4];
arr.forEach(function(item) {
    const temp = item + 1;
    console.log(temp);
})
```

2. 箭头函数（arrowfunction）

与 function 相比，箭头函数是一个用起来更加舒服的语法。下面一起来看看箭头函数的基本使用规则。

```
// es5
var fn = function(a, b) {
    return a + b;
}

// ES6 箭头函数写法，当函数直接被return时，可以省略函数体的括号
const fn = (a, b) => a + b;

// es5
var foo = function() {
    var a = 20;
    var b = 30;
    return a + b;
}

// es6
const foo = () => {
   const a = 20;
   const b = 30;
   return a + b;
}
```

需要注意的是，箭头函数只能替换函数表达式，即使用 var/let/const 声明的函数。而直接使用 function 声明的函数是不能使用箭头函数替换的。

除写法不同外，箭头函数还有一个非常重要的特性必须掌握。

前面学习 this 时我们知道，函数内部的 this 指向，与它的调用者有关，或者使用 call/apply/bind 也可以修改内部的 this 指向。

下面通过一个例子来简单复习一下。

```
var name = 'TOM';
```

```
var getName = function() {
    console.log(this.name);
}

var person = {
    name: 'Alex',
    getName: getName
}

var other = {
    name: 'Jone'
}

getName();  // 独立调用，this指向undefined，并自动转向window
person.getName(); // 被person调用，this指向person
getName.call(other); // call 修改this，指向other
}
```

明白了 this 的指向后，可以很容易地知道这几个不同的方法调用时会输出什么结果。但是当我们将最初声明的 getName 方法修改为箭头函数的形式后，输出结果会发生什么变化呢？一起来看一下。

```
var name = 'TOM';

// 更改为箭头函数的写法
var getName = () => {
    console.log(this.name);
}

var person = {
    name: 'Alex',
    getName: getName
}

var other = {
    name: 'Jone'
```

```
}

getName();
person.getName();
getName.call(other);
```

通过运行可以发现，三次调用都输出了 TOM。这也正是要跟大家分享的箭头函数的不同。**箭头函数中的 this，就是声明函数时所处上下文中的 this，它不会被其他方式所改变。**

在这个例子中，getName 是在全局上下文中声明的，因此 this 指向的是 window 对象，所以输出的结果全是 TOM。

在实践中，常常会遇到 this 在传递过程中发生改变，因此带来许多困扰，例如：

```
var Mot = function(name) {
    this.name = name;
}
Mot.prototype = {
    constructor: Mot,
    do: function() {
        console.log(this.name);
        document.onclick = function() {
            console.log(this.name);
        }
    }
}

new Mot('Alex').do();
```

在这个例子中，当调用 do 方法时，我们期望单击 document 时仍然输出‘Alex’。但是很遗憾，在 onclick 的回调函数中，this 的指向其实已经发生了变化，它指向了 document，因此肯定得不到我们想要的结果。通常的解决方案是使用箭头函数。

```
var Mot = function(name) {
    this.name = name;
}
Mot.prototype = {
```

```
    constructor: Mot,
    do: function() {
        console.log(this.name);
        // 修改为箭头函数即可
        document.onclick = () => {
            console.log(this.name);
        }
    }
}

new Mot('Alex').do();
```

除此之外，arguments 还有一个需要大家注意的地方，即在箭头函数中，没有 arguments 对象。

```
var add = function(a, b) {
    console.log(arguments);
    return a + b;
}

add(1, 2);

// 结果输出一个类数组对象
/*
[
    0: 1,
    1: 2,
    length: 2,
    callee: f (a, b),
    Symbol(Symbol.iterator): f values()
]
*/
```

```
var add = (a, b) => {
    console.log(arguments);
    return a + b;
}
```

```
add(1, 2);    // arguments is not defined
```

10.2 模板字符串

模板字符串是为了解决传统字符串拼接不便利而出现的。先通过一个简单的例子来观察一下模板字符串与传统字符串的差别。

```
// ES5
var a = 20;
var b = 30;
var string = a + "+" + b + "=" + (a + b);

// ES6
const a = 20;
const b = 30;
const string = `${a}+${b}=${a+b}`;
```

模板字符串使用反引号“`”将整个字符串包裹起来，变量或者表达式则使用 `${}` 来包裹。

除了能够在字符串中嵌入变量，还可以用来定义多行字符串，其中所有的空格、缩进、换行都会被保留下来。

```
var elemString = `<div>
    <p>So you crossed the line, how'd you get that far?</p>
    <p>${word} you give it a try.</p>
</div>`
```

如果是用传统的“+”来拼接这段字符串，则会非常麻烦。

${} 中可以放入一个变量、表达式，甚至一个函数。

```
// 变量
const hello = 'hello';
let message = `${hello}, world!`;
```

```
// 表达式
const a = 40;
const b = 50;
let result = `the result is: ${a + b}`;

// 函数
let fn = () => {
    const result = 'you are the best.';
    return result;
}
let str = `he said: ${fn()}`
```

10.3 解析结构

解析结构是一种从对象或者数组中取得值的一种全新的写法，只需通过一个简单的例子就能立刻明白是怎么回事了。

首先假设有这样一个 JSON 数据：

```
var tom = {
    name: 'TOM',
    age: 20,
    gender: 1,
    job: 'studend'
}
```

这里用对象的方式保存了 TOM 的一些基本信息，在传统方式中，若想取得其中的某些信息，通常会这样做：

```
var name = tom.name;
var age = tom.age;
var gender = tom.gender;
var job = tom.job;
```

当使用解析结构时，可以这样做：

```
const { name, age, gender, job } = tom;

// 上面实际上是下面这种方式的简写，但在实践中我们并不会这么使用
// const {name: name, age: age, gender: gender, job: job } = tom;
```

是不是简单了很多。

我们还可以给变量指定默认值：

```
// 如果数据中能找到name，则变量的值与数据中相等；若找不到，则使用默认值
const { name = 'Jake', stature = '170' } = tom;
```

或者给变量重新命名：

```
const { gender: t, job } = tom;
```

重命名之后，gender 将无法访问，而只能通过新的变量名 t 来访问对应的数据。

我们还可以利用解析结构获取嵌套数据中的值。

```
const peoples = {
    counts: 100,
    detail: {
        tom: {
            name: 'tom',
            age: 20,
            gender: 1,
            job: 'student'
        }
    }
}

// 获取tom
const { detail: { tom } } = peoples;

// 直接获取tom的age与gender
const { detail: { tom: { age, gender }}} = peoples;
```

如果看不懂这样的嵌套结构，在使用时也可以拆解来看。

```
const { detail } = peoples;
const { tom } = detail;
const { age, gender } = tom;
```

除此之外，数组也有自己的解析结构。当然，写法上与对象的解析结构略有不同，下面通过一个简单的例子来观察。

```
const arr = [1, 2, 3];
const [a, b, c] = arr;

// 等价于
const a = arr[0];
const b = arr[1];
const c = arr[2];
```

注意：与对象不同的是，数组中变量和值的关系与序列号是一一对应的，这是一个有序的对应关系。而对象则根据属性名一一对应，这是一个无序的对应关系。因此在实践中，对象的解析结构使用得更加频繁与便利。

在使用时，无论是对象还是数组，建议将解析结构的运用重点放在取值上，而不是试图利用解析结构来声明初始变量。

运用重点放在取值上的意思是，当想要从一个已有的数据中获取想要的信息时应使用解析结构，而不是为了声明几个初始变量，自己拼凑一个对象或者数组出来。

在一个函数中，当传入的参数是数组或者对象时，也可以使用解析结构来简化我们的代码。下面是解析结构在不同场景下的取值。

```
const fn = ({name, age}) => {
    return `${name} is age is ${age}`;
}

fn({ name: 'TOM', age: 20 });
```

最后，总结一个关于默认值应用场景的小知识点。

```
// 对象解析结构中的默认值
const { name, age = 20 } = tom;

// 数组解析结构中的默认值
const [a, b = 20] = [1]

// 函数参数中的默认值
const fn = (x = 20, y = 30) => x + y;
fn();  // 50
```

10.4 展开运算符

在ES6中，使用...来表示展开运算符，它可以展开数组/对象。

下面通过一个简单的例子来了解一下展开运算符的作用。

```
// 首先声明一个数组
const arr1 = [1, 2, 3];

// 其次声明另一个数组，我们期望新数组中包含数组arr1的所有元素，
// 因此可以利用展开运算符
const arr2 = [...arr1, 4, 5, 6];

// 那么arr2就变成了 [1, 2, 3, 4, 5, 6]
```

当然，展开对象也可以得到类似的结果。

```
const object1 = {
    a: 1,
    b: 2,
    c: 3
}

const object2 = {
```

```
    ...object1,
    d: 4,
    e: 5,
    f: 6
}

// object2的结果等价于
object2 = {
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: 5,
    f: 6
}
```

在解析结构中，也常常使用展开运算符。

```
const tom = {
    name: 'TOM',
    age: 20,
    gender: 1,
    job: 'student'
}

const { name, ...others } = tom;

// others = {age: 20, gender: 1, job: "student"}
```

利用这样的方式，可以将 tom 对象中的剩余参数全部丢在 others 中，而不用一个一个地去列举所有的属性。

在 react 组件中，常常使用展开运算符来传递数据。

```
const props = {
  size: 1,
  src: 'xxxx',
```

```
  mode: 'si'
}

const { size, ...others } = props;

// 利用展开运算符传递数据
<button {...others} size={size} />
```

展开运算符还可以运用在函数参数中，放置于函数参数的最后一个参数（且只能放置于最后），表示不定参。

```
// 求所有参数之和
const add = (a, b, ...more) => {
    return more.reduce((m, n) => m + n) + a + b
}

console.log(add(1, 23, 1, 2, 3, 4, 5)) // 39
```

展开运算符可以大大提高代码效率，因此记住这些常见的应用场景非常重要。

10.5 Promise 详解

10.5.1 异步与同步

什么是异步？什么是同步？在代码的执行过程中，经常会涉及两个不同的概念，它们就是同步与异步。

同步是指当发起一个请求时，如果未得到请求结果，代码将会等待，直到结果出来，才会执行后面的代码。异步是指当发起一个请求时，不会等待请求结果，而是直接继续执行后面的代码。

我们可以用一个双人问答的场景来比喻异步与同步。A 向 B 问了一个问题之后，然后就笑呵呵地等着 B 回答，B 回答了之后 A 才会接着问下一个问题，这是同步。A 向 B 问了一个问题之后，不等待 B 的回答，接着问下一个问题，这是异步。

在 JavaScript 中，也可以用代码来表示它们之间的区别。

首先使用 Promise 模拟一个发起请求的函数，该函数执行后，会在 1s 之后返回数值 30。

```
function fn() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve(30);
        }, 1000);
    })
}
```

在该函数的基础上，可以使用 async/await 语法来模拟同步的效果。

```
var foo = async function() {
    var t = await fn();
    console.log(t);
    console.log('next code');
}

foo();
```

输出结果为：

```
// 1s之后依次输出
30
next code
```

而异步效果则会有不同的输出结果：

```
var foo = function() {
    fn().then(function(resp) {
        console.log(resp);
    });
    console.log('next code');
}
```

输出结果：

```
next code
// 停顿1s后继续输出
30
```

下面分别学习 Promise 与 async/await 的具体作用与使用方法。

10.5.2 Promise

1.Ajax

Ajax 是网页与服务端进行数据交互的一种技术。我们可以通过服务端提供的接口，用 Ajax 向服务端请求我们需要的数据。整个过程的简单实现如下所示。

```
// 简单的Ajax原生实现

// 由服务端提供的接口
var url = 'https://hq.tigerbrokers.com/fundamental/finance_calendar/
getType/2017-02-26/2017-06-10';
var result;

var XHR = new XMLHttpRequest();
XHR.open('GET', url, true);
XHR.send();

XHR.onreadystatechange = function() {
    if (XHR.readyState == 4 && XHR.status == 200) {
        result = XHR.response;
        console.log(result);
    }
}
```

在 Ajax 的原生实现中，利用了 onreadystatechange 事件，只有当该事件触发并且符合一定条件时，才能拿到我们想要的数据，之后才能开始处理数据。

这样做看上去并没有什么，但是如果这个时候，还需要做另外一个 Ajax 请求，那么这个新的 Ajax 请求中的一个参数，则必须从上一个 Ajax 请求中获取，这个时候我们就不得不这样做：

```
var url = 'https://hq.tigerbrokers.com/fundamental/finance_calendar/
getType/2017-02-26/2017-06-10';
var result;

var XHR = new XMLHttpRequest();
XHR.open('GET', url, true);
XHR.send();

XHR.onreadystatechange = function() {
    if (XHR.readyState == 4 && XHR.status == 200) {
        result = XHR.response;
        console.log(result);

        // 伪代码
        var url2 = 'http:xxx.yyy.com/zzz?ddd=' + result.someParams;
        var XHR2 = new XMLHttpRequest();
        XHR2.open('GET', url, true);
        XHR2.send();
        XHR2.onreadystatechange = function() {
            ...
        }
    }
}
```

当第三个 Ajax（甚至更多）仍然依赖上一个请求的时候，此时的代码就变成了一场灾难。我们需要不停地嵌套回调函数，以确保下一个接口所需要的参数的正确性。这样的灾难，我们称之为 **回调地狱**。

这时，我们就需要一个叫作 Promise 的语法来解决这样的问题。

当然，除回调地狱外，还有一个非常重要的需求：为了使代码更具可读性和可维护性，我们需要将数据请求与数据处理明确地区分开来。上面的写法，是完全没有区分开，当数据处理变得复杂时，也许我们自己都无法轻松维护自己的代码了。这也是模块化过程中，必须要掌握的一个重要技能，请一定重视。

通过前面的学习我们知道，当想要确保某代码在某某之后执行时，可以利用函数调用栈，将想要执行的代码放入回调函数中。

```
// 一个简单的封装
function want() {
    console.log('这是你想要执行的代码');
}

function fn(want) {
    console.log('这里表示执行了一大堆各种代码');

    // 其他代码执行完毕后，最后执行回调函数
    want && want();
}

fn(want);
```

利用回调函数封装，是我们在初学 JavaScript 时常常会使用的技能。

除利用函数调用栈的执行顺序外，还可以利用队列机制来确保我们想要的代码压后执行。

```
function want() {
    console.log('这是你想要执行的代码');
}

function fn(want) {
    // 将想要执行的代码放入队列中后，根据事件循环机制，
    // 就不用将它放到最后面了，可以自由选择
    want && setTimeout(want, 0);
    console.log('这里表示执行了一大堆各种代码');
}

fn(want);
```

与 setTimeout 类似，Promise 也可以认为是一种任务分发器，它将任务分配到 Promise 队列中，通常的流程是首先发起一个请求，然后等待（等待时间无法确定）并处理请求结果。

简单的用法如下所示。

```
var tag = true;
var p = new Promise(function(resolve, reject) {
    if (tag) {
        resolve('tag is true');
    } else {
        reject('tag is false');
    }
})

p.then(function(result) {
    console.log(result);
})
.catch(function(err) {
    console.log(err);
})
```

运行这段代码，并通过修改 tag 的值来感受一下运行结果的不同。

接下来介绍 Promise 相关的基础知识。

- new Promise 表示创建一个 Promise 实例对象
- Promise 函数中的第一个参数为一个回调函数，也可以称之为 executor。通常情况下，在这个函数中，会执行发起请求操作，并修改结果的状态值。
- 请求结果有三种状态，分别是 pending（等待中，表示还没有得到结果）、resolved（得到了我们想要的结果，可以继续执行），以及 rejected（得到了错误的，或者不是我们期望的结果，拒绝继续执行）。请求结果的默认状态为 pending。在 executor 函数中，可以分别使用 resolve 与 reject 将状态修改为对应的 resolved 与 rejected。resolve、reject 是 executor 函数的两个参数，它们能够将请求结果的具体数据传递出去。
- Promise 实例拥有的 then 方法，可用来处理当请求结果的状态变成 resolved 时的逻辑。then 的第一个参数为一个回调函数，该函数的参数是 resolve 传递出来的数据。在上面的例子中，result = tag is true。
- Promise 实例拥有的 catch 方法，可用来处理当请求结果的状态变成 rejected 时的逻辑。catch 的第一个参数为一个回调函数，该函数的参数是 reject 传递出来的数据。在上面的例子中，err = tag is false。

下面通过几个简单的例子来感受一下 Promise 的用法。

例子 1：

```
function fn(num) {
    // 创建一个Promise实例
    return new Promise(function(resolve, reject) {
        if (typeof num == 'number') {
            // 修改结果状态值为resolved
            resolve();
        } else {
            // 修改结果状态值为rejected
            reject();
        }
    }).then(function() {
        console.log('参数是一个number值');
    }).catch(function() {
        console.log('参数不是一个number值');
    })
}

// 修改参数的类型，观察输出结果
fn('12');

// 注意观察该语句的执行顺序
console.log('next code');
```

例子 2：

```
function fn(num) {
    return new Promise(function(resolve, reject) {
        // 模拟一个请求行为，2s以后得到结果
        setTimeout(function() {
            if (typeof num == 'number') {
                resolve(num)
            } else {
                var err = num + ' is not a number.'
```

```
                reject(err);
            }
        }, 2000);
    })
    .then(function(resp) {
        console.log(resp);
    })
    .catch(function(err) {
        console.log(err);
    })
}

// 修改传入的参数类型，观察结果变化
fn('abc');

// 注意观察该语句的执行顺序
console.log('next code');
```

因为 fn 函数运行的结果是返回一个 Promise 对象，因此也可以将上面的例子修改为：

```
function fn(num) {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            if (typeof num == 'number') {
                resolve(num)
            } else {
                var err = num + ' is not a number.'
                reject(err);
            }
        }, 2000);
    })
}

fn('abc')
.then(function(resp) {
    console.log(resp);
```

```
})
.catch(function(err) {
    console.log(err);
})

// 注意观察该语句的执行顺序
console.log('next code');
```

then 方法可以接收两个参数，第一个参数用来处理 resolved 状态的逻辑，第二个参数用来处理 rejected 状态的逻辑。

```
// 修改上面例子中的部分代码
fn('abc')
.then(function(resp) {
    console.log(resp);
}, function(err) {
    console.log(err)
})
```

因此，catch 方法其实与下面的写法等价：

```
fn('abc').then(null, function(err) {
    console.log(err)
})
```

then 方法因为返回的仍是一个 Promise 实例对象，因此 then 方法可以嵌套使用。在这个过程中，通过在内部函数末尾 return 的方式，能够将数据持续往后传递。下面的例子中，注意观察数据传递过程中的变化。

```
function fn(num) {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            if (typeof num == 'number') {
                resolve(num)
            } else {
```

```
                var err = num + ' is not a number.'
                reject(err);
            }
        }, 2000);
    })
}

fn(20)
.then(function(result) {
    console.log(result); //20
    return result + 1;
})
.then(function(result) {
    console.log(result); // 21
    return result + 1;
})
.then(function(result) {
    console.log(result); // 22
    return result + 1;
})
.then(function(result) {
    console.log(result); // 23
})
.then(function(result) {
    console.log(result); // undefined
})

// 注意观察该语句的执行顺序
console.log('next code');
```

了解了这些基础知识之后，再回过头来看看最开始提到过的 Ajax 的例子，我们可以进行一个简单的封装，详细代码如下。

```
var url = 'https://hq.tigerbrokers.com/fundamental/finance_calendar/
getType/2017-02-26/2017-06-10';
```

```
// 封装一个get请求的方法
function getJSON(url) {
    return new Promise(function(resolve, reject) {
        // 利用Ajax发送一个请求
        var XHR = new XMLHttpRequest();
        XHR.open('GET', url, true);
        XHR.send();

        // 等待结果
        XHR.onreadystatechange = function() {
            if (XHR.readyState == 4) {
                if (XHR.status == 200) {
                    try {
                        var response = JSON.parse(XHR.responseText);
                        //  得到正确的结果修改状态并将数据传递出去
                        resolve(response);
                    } catch (e) {
                        reject(e);
                    }
                } else {
                    // 得到错误结果并抛出异常
                    reject(new Error(XHR.statusText));
                }
            }
        }
    })
}

// 封装好之后，使用就很简单了
getJSON(url).then(function(resp) {
    console.log(resp);
    // 之后就是处理数据的具体逻辑
});
```

现在几乎所有的库都利用 Promise 对 Ajax 请求进行了封装，当然也包括常用的 jQuery，因此在使用 jQuery 等库中的 Ajax 请求时，可以利用 Promise 来让代码更加的简洁和优雅。

```
$.get(url).then(function(resp) {
    // ... 处理success的结果
})
.catch(function(err) {
    // ...
})
```

2.Promise.all

当有一个 Ajax 请求，它的参数需要另外两个甚至更多个请求都有返回结果之后才能确定时，就需要用到 Promise.all 来帮助我们应对这个场景。

Promise.all 接收一个 Promise 对象组成的数组作为参数，当这个数组中所有的 Promise 对象状态都变成 resolved 或者 rejected 时，它才会去调用 then 方法。

```
var url = 'https://hq.tigerbrokers.com/fundamental/finance_calendar/
getType/2017-02-26/2017-06-10';
var url1 = 'https://hq.tigerbrokers.com/fundamental/finance_calendar/
getType/2017-03-26/2017-06-10';

function renderAll() {
    return Promise.all([getJSON(url), getJSON(url1)]);
}

renderAll().then(function(value) {
    // 建议在浏览器中看看这里的value值
    console.log(value);
})
```

3.Promise.race

与 Promise.all 相似的是，Promise.race 也是以一个 Promise 对象组成的数组作为参数，不同的是，只要当数组中的其中一个 Promsie 状态变成 resolved 或者 rejected 时，就可以调用.then 方法，而传递给 then 方法的值也会有所不同。可以在浏览器中运行下面的例子，然后与上面的例子进行对比。

```
function renderRace() {
    return Promise.race([getJSON(url), getJSON(url1)]);
}

renderRace().then(function(value) {
    console.log(value);
})
```

10.5.3 async/await

异步问题不仅可以使用前面学到的 Promise 来解决，还可以用 async/await 来解决。

async/await 是 ES7 中新增的语法，虽然现在最新的 Chrome 浏览器已经支持了该语法，但在实际使用中，仍然需要在构建工具中配置对该语法的支持才能放心使用。因此，如果你目前的开发经验还没有涉及构建工具的使用，则可以暂时跳过该语法的学习。

注意：接下来的知识会配合 ES6 的语法进行讲解，如果你还没有学习 ES6 的语法，也可以暂时跳过。

在函数声明的前面，加上关键字 async，这就是 async 的具体使用。

```
async function fn() {
    return 30;
}

// 或者
const fn = async () => {
    return 30;
}
```

可以打印出 fn 函数的运行结果。

```
console.log(fn());

// result
Promise = {
```

```
    __proto__: Promise,
    [[PromiseStatus]]: "resolved",
    [[PromiseValue]]: 30
}
```

可以发现 fn 函数运行后返回的是一个标准的 Promise 对象，因此可以猜想到 async 其实是 Promise 的一个语法糖，目的是为了让写法更加简单，因此也可以使用 Promise 的相关语法来处理后续的逻辑。

```
fn().then(res => {
    console.log(res);  // 30
})
```

await 的含义是等待，意思就是代码需要等待 await 后面的函数运行完并且有了返回结果之后，才继续执行下面的代码。这正是同步的效果。

但是需要注意的是，await 关键字只能在 async 函数中使用，并且 await 后面的函数运行后必须返回一个 Promise 对象才能实现同步的效果。

当使用一个变量去接收 await 的返回值时，该返回值为 Promise 中 resolve 传递出来的值，也就是 PromiseValue。

```
// 定义一个返回Promise对象的函数
function fn() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(30);
        }, 1000);
    })
}

// 然后利用async/await来完成代码
const foo = async () => {
    const t = await fn();
    console.log(t);
    console.log('next code');
}
```

```
foo();

// result:
// 30
// next code
```

通过运行这个例子可以看出，在 async 函数中，当运行遇到 await 时，就会等待 await 后面的函数运行完毕，而不会直接执行 next code。

如果直接使用 then 方法，要想达到同样的结果，就不得不把后续的逻辑写在 then 方法中。

```
const foo = () => {
    return fn().then(t => {
        console.log(t);
        console.log('next code');
    })
}

foo();
```

很显然，如果使用 async/await，代码结构会更加简洁，逻辑也更加清晰。

3. 异常处理

在 Promise 中，是通过 catch 的方式来捕获异常的，而当使用 async 时，则通过 try/catch 来捕获异常。

```
function fn() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('some error.');
        }, 1000);
    })
}
```

```
const foo = async () => {
    try {
        await fn();
    } catch (e) {
        console.log(e);  // some error
    }
}

foo();
```

如果有多个 await 函数，那么只返回第一个捕获到的异常。

```
function fn1() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('some error fn1.');
        }, 1000);
    })
}
function fn2() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('some error fn2.');
        }, 1000);
    })
}

const foo = async () => {
    try {
        await fn1();
        await fn2();
    } catch (e) {
        console.log(e);  // some error fn1.
    }
}
```

```
foo();
```

在实践中遇到异步场景最多的就是接口请求，因而这里就以 jQuery 中的 $.get 为例，简单展示一下如何配合 async/await 来解决这个场景。

```
//先定义接口请求的方法，由于jQuery封装的几个请求方法都是返回Promise实例
//因此可以直接使用await函数实现同步
const getUserInfo = () => $.get('xxxx/api/xx');

const clickHandler = async () => {
    try {
        const resp = await getUserInfo();
        // resp为接口返回内容，接下来利用它来处理对应的逻辑
        console.log(resp);

        // do something
    } catch (e) {
        // 处理错误逻辑
    }
}
```

为了保证逻辑的完整性，在实践中 try/catch 必不可少。

10.6 事件循环机制

事件循环机制（Event Loop）是全面了解 JavaScript 代码执行顺序绕不开的一个重要知识点。虽然许多人知道这个知识点非常重要，但是其实很少有人能够真正地理解它。特别是在 ES6 正式支持 Promise 之后，对于新标准中事件循环的理解就变得更加重要。

在学习事件循环机制之前，默认大家已经理解了以下概念，这些知识点在本书前面的章节中都有详细解读过，如果还没有理解，可以回过头去重新学习。

- ◎ 执行上下文（Execution context）
- ◎ 函数调用栈（call stack）
- ◎ 队列数据结构（queue）

◎ Promise

先来看两个简单的例子，看看是否能够得出正确的执行结果。

```
// demo01
setTimeout(function() {
    console.log(1);
}, 0);
console.log(2);

for(var i = 0; i < 5; i++) {
    console.log(3)
}

console.log(4);
```

```
// demo02
console.log(1);

for(var i = 0; i < 5; i++) {
    setTimeout(function() {
        console.log('2-' + i);
    }, 0);
}

console.log(3);
```

很多人在运行之后可能会感到困惑，为什么即使设置了 setTimeout 的延迟时间为 0，它里面的代码仍然是最后执行的？

通常情况下，决定代码执行顺序的是函数调用栈。很显然这里的 setTimeout 中的执行顺序已经不是用函数调用栈能够解释清楚的了，这是为什么呢？

答案是队列，如图 10-1所示。

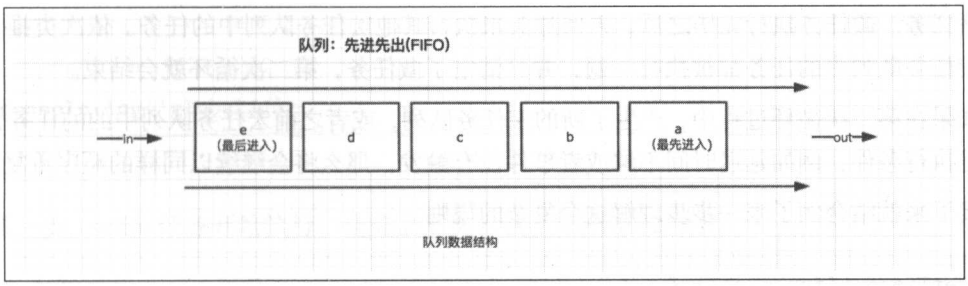


图 10-1 队列

JavaScript 的一个特点就是单线程，但是很多时候我们仍然需要在不同的时间去执行不同的任务，例如给元素添加点击事件，设置一个定时器，或者发起 Ajax 请求。因此需要一个异步机制来达到这样的目的，事件循环机制也因此而来。

每一个 JavaScript 程序都拥有唯一的事件循环，大多数代码的执行顺序是可以根据函数调用栈的规则执行的，而 `setTImeout/setInterval` 或者不同的事件绑定（click、mousedown 等）中的代码，则通过队列来执行。

`setTimeout` 为任务源，或者任务分发器，由它们将不同的任务分发到不同的任务队列中去。每一个任务源都有对应的任务队列。

任务队列又分为宏任务（macro-task）与微任务（micro-task）两种，在浏览器中，包括：

◎ macro-task：script（整体代码）、setTimeout/setInterval、I/O、UI rendering 等。

◎ micro-task：Promise。

注意：在 node.js 中还包括更多的任务队列，此处不做讨论。来自不同任务源的任务会进到不同的任务队列中，其中 setTimeout 与 setInterval 是同源的。

事件循环的顺序，决定了 JavaScript 代码的执行顺序。

它从 macro-task 中的 script 开始第一次循环。此时全局上下文进入函数调用栈，直到调用栈清空（只剩下全局上下文），在这个过程中，如果遇到任务分发器，就会将任务放入对应队列中去。

第一次循环时，macro-task 中其实只有 script，因此函数调用栈清空之后，会直接执行所有的 micro-task。当所有可执行的 micro-task 执行完毕之后，就表示第一次事件循环已经结束。

第二次循环会再次从 macro-task 开始执行。此时 macro-task 中的 script 队列中已经没有任务了，但是可能会有其他的队列任务，而 micro-task 中暂时还没有任务。此时会先选择其中一个宏任务队列，例如 setTimeout，将该队列中的所有任务全部执行完毕，然后再执行此过程中可能产

生的微任务。微任务执行完毕之后，再回过头来执行其他宏任务队列中的任务。依次类推，直到所有宏任务队列中的任务都被执行一遍，并且清空了微任务，第二次循环就会结束。

如果在第二次循环过程中，产生了新的宏任务队列，或者之前宏任务队列中的的任务暂时没有满足执行条件，例如延迟时间不够或者事件没有触发，那么将会继续以同样的顺序重复循环。

接下来就结合例子来一步步理解这个复杂的规则。

```
setTimeout(function() {
    console.log('setTimeout')
}, 0);
console.log('global');
```

首先，macro-task script 任务队列最先执行。执行过程中遇到 setTimeout，setTimeout 会将它的任务分发到 setTimeout 任务队列中去，因此里面的代码是不执行的。代码会接着往下执行，因而这里会首先输出 global。

由于整个过程没有产生 micro-task，因而第一轮循环结束。第二轮循环开始，发现 setTimeout 任务队列中已存在一个任务，所以执行它，这个时候会输出“setTimeout”。

因此这个例子的最终输出结果为：

```
global
setTimeout
```

再来看一个稍微复杂一点的例子。

```
setTimeout(function() {
    console.log('timeout1');
})

new Promise(function(resolve) {
    console.log('promise1');
    for(var i = 0; i < 1000; i++) {
        i == 99 && resolve();
    }
    console.log('promise2');
}).then(function() {
    console.log('then1');
```

```
})

console.log('global1');
```

第一步，script 任务开始执行，全局上下文入栈，如图 10-2所示。

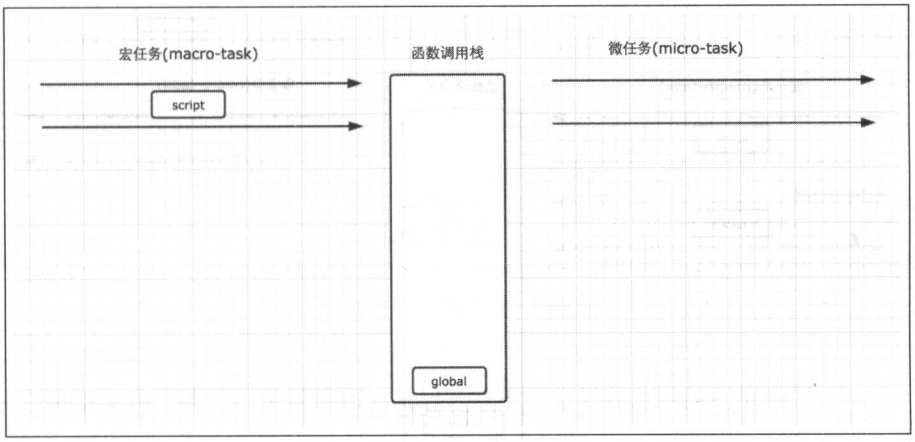


图 10-2 全局上下文入栈

第二步，script 任务执行时首先遇到了 setTimeout，setTimeout 为一个宏任务源，而它的作用就是将任务分发到它对应的队列中去，如图 10-3所示。

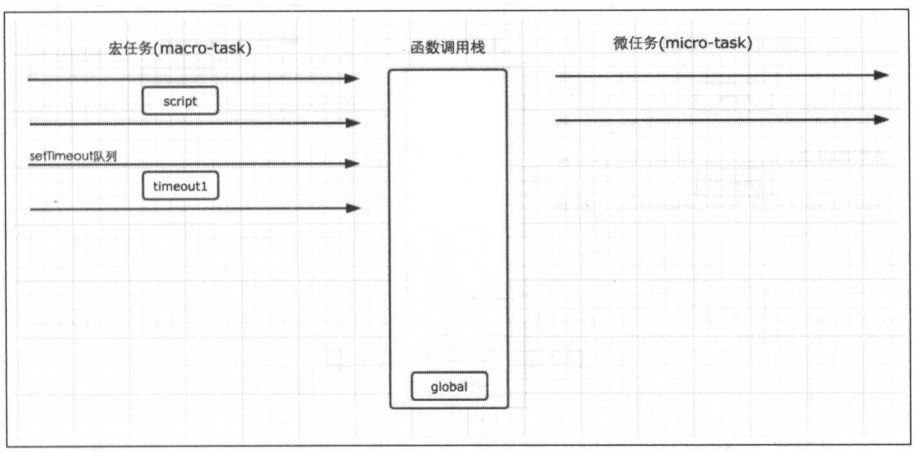


图 10-3 任务分发

第三步，script 执行时遇到 Promise 实例。Promise 构造函数中的第一个参数，是在 new 创建实例的时候执行，因此不会进入任何其他的队列，而是在当前任务直接执行，后续的.then 则会被分发到 micro-task 的 Promise 队列中去。

因此，构造函数执行时，里面的参数进入函数调用栈执行。for 循环不会进入任何队列，因此代码会依次执行，所以这里的 promise1 和 promise2 会依次输出。

promise1 入栈执行，执行后 promise1 被最先输出，如图 10-4所示。

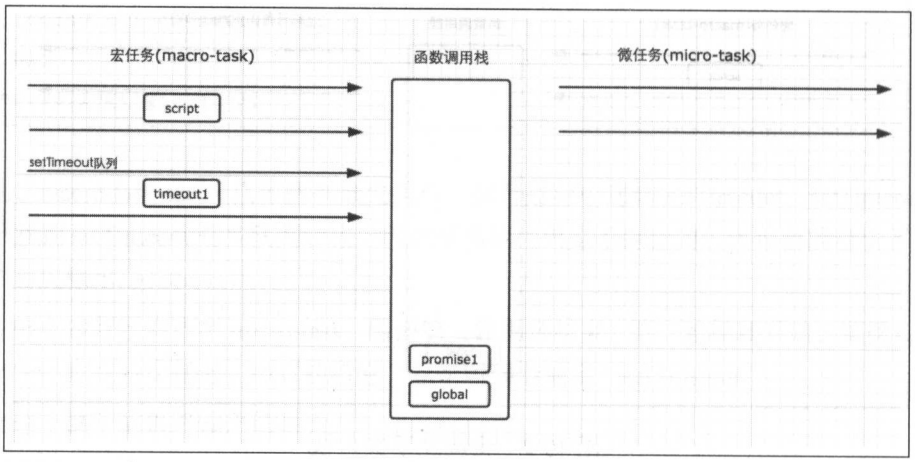


图 10-4　promise1 被最先输出

resolve 在 for 循环中入栈执行，如图 10-5所示。

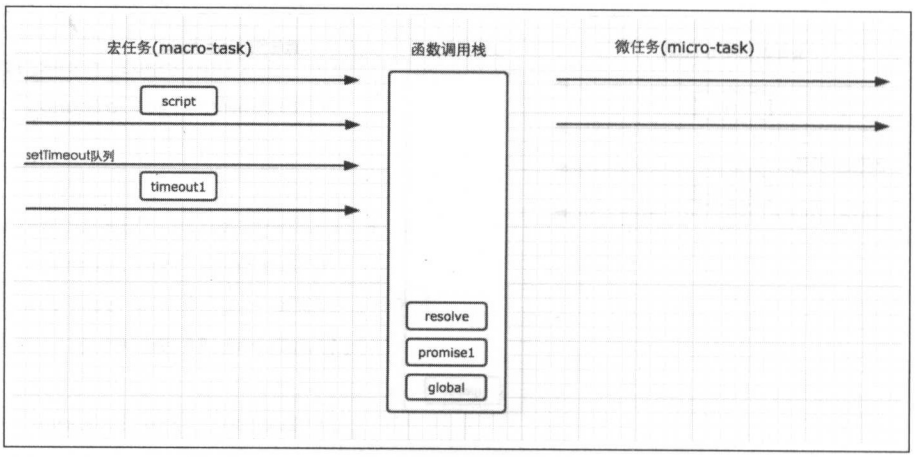


图 10-5　resolve 在 for 循环中入栈

在构造函数执行过程中，resolve 执行完毕出栈，promise2 输出，promise1 也出栈。then 执行时，Promise 任务 then1 进入对应队列，如图 10-6所示。

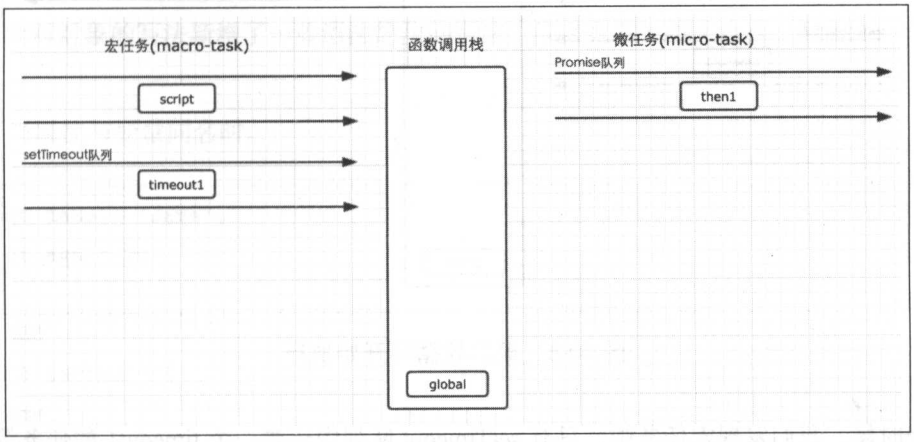


图 10-6 函数入栈并输出

script 任务继续向下执行，最后输出 globa1，至此，全局任务就执行完毕了。

第四步，第一个宏任务 script 执行完毕之后，就开始执行所有可执行的微任务。这个时候，微任务中，只有 Promise 队列中的一个任务 then1。因此直接执行就可以了，执行结果输出 then1，当然，它也是进入函数调用栈中执行的，如图 10-7所示。

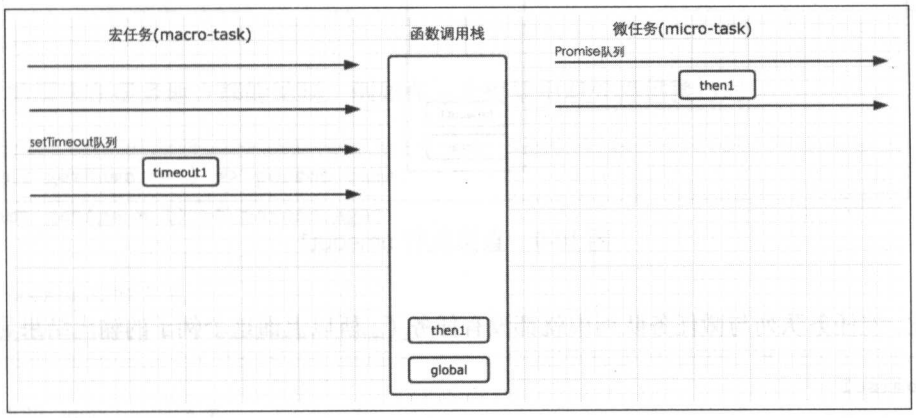


图 10-7 微任务调用栈

第五步，当所有的 micro-tast 执行完毕之后，表示第一轮的循环就结束了开始第二轮的循环。第二轮循环仍然从宏任务 macro-task 开始，如图 10-8所示。

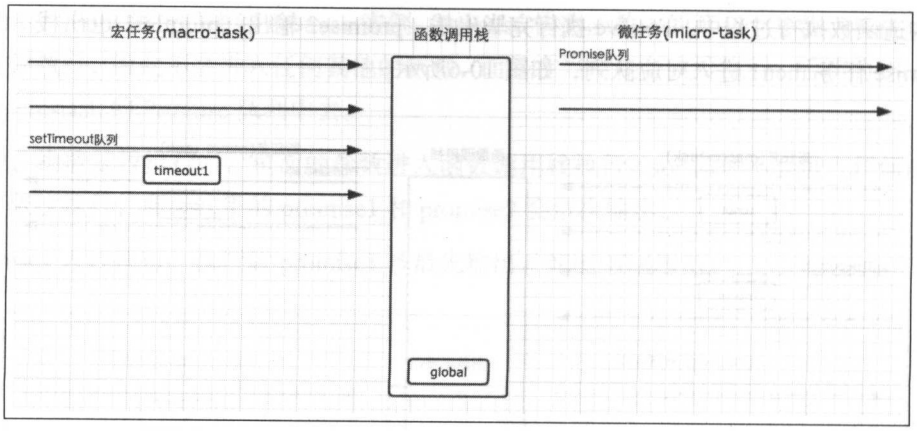


图 10-8　第二轮循环开始执行

这个时候，我们发现宏任务中，只有 setTimeout 队列中还要一个 timeout1 的任务等待执行。因此直接执行即可，如图 10-9所示。

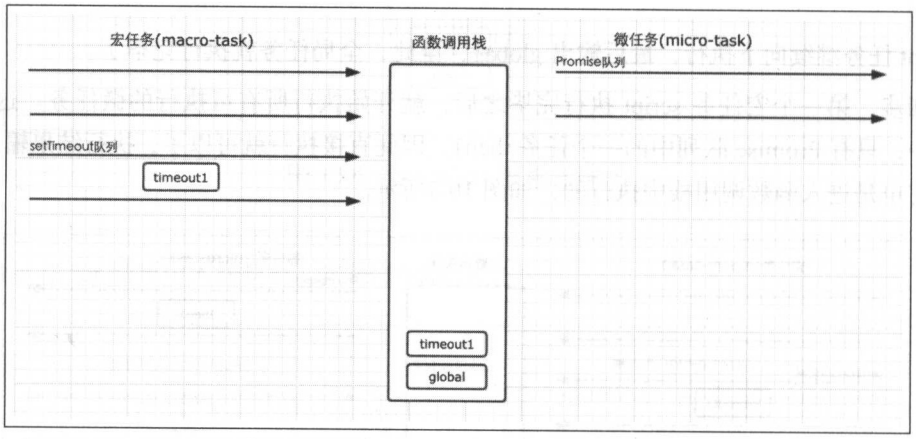


图 10-9　直接执行 timeout1

至此，宏任务队列与微任务队列中就都没有任务了，所以上面这个例子的输出结果显而易见。

```
promise1
promise2
global1
then1
timeout1
```

10.7 对象与 class

ES6 针对对象的写法新增了一些语法简化的写法，没有太多的技术难点，只需一一学习一下就可以了。

（1）当属性与变量同名时。

```
const name = 'Jane';
const age = 20

// ES6
const person = {
  name,
  age
}

// 等价于 ES5
var person = {
  name: name,
  age: age
};
```

这样的写法在很多地方都能见到，例如在一个模块中对外提供接口时。

```
const getName = () => person.name;
const getAge = () => person.age;

// commonJS的方式
module.exports = { getName, getAge }

// ES6 modules的方式
export default { getName, getAge  }
```

（2）对象中方法的简写。

```
// ES6
const person = {
    name,
    age,
    getName() { // 只要不使用箭头函数，this就还是我们熟悉的this
        return this.name
    }
}

// ES5
var person = {
    name: name,
    age: age,
    getName: function getName() {
        return this.name;
    }
};
```

（3）可以使用变量作为对象的属性，只需用中括号 [] 包裹即可。

```
const name = 'Jane';
const age = 20

const person = {
    [name]: true,
    [age]: true
}
```

在 ant-design 的源码中，大量使用了这种方式来拼接当前元素的 className，例如：

```
let alertCls = classNames(prefixCls, {
      [`${prefixCls}-${type}`]: true,
      [`${prefixCls}-close`]: !this.state.closing,
      [`${prefixCls}-with-description`]: !!description,
      [`${prefixCls}-no-icon`]: !showIcon,
```

```
    [`${prefixCls}-banner`]: !!banner,
}, className);
```

1.class

ES6 为创建对象提供了新的语法–class。如果对 ES5 中面向对象的方式非常熟悉，则 class 掌握起来会非常容易，因为除了写法不同，并不会新增其他难以理解的知识点。下面就用一个简单的例子来看看具体是怎样的。

```
// ES5
// 构造函数
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// 原型方法
Person.prototype.getName = function() {
    return this.name
}

// ES6
class Person {
    constructor(name, age) {  // 构造函数
        this.name = name;
        this.age = age;
    }

    getName() {  // 原型方法
        return this.name
    }
}
```

利用 ES6 的 class 关键字，可以很容易地定义一个类。相比较而言，比原型的方式更加简单，也更加容易理解。

除此之外，还需要特别注意在实际使用中的几种不同写法。在下面的例子中，说明了它们分别对应的 ES5 中的含义。

```
class Person {
    constructor(name, age) {  // 构造函数
        this.name = name;
        this.age = age;
    }

    getName() {   // 这种写法表示将方法添加到原型中
        return this.name
    }

    static a = 20;  // 等同于 Person.a = 20

    c = 20;   // 表示在构造函数中添加属性，在构造函数中等同于 this.c = 20

    // 箭头函数的写法表示在构造函数中添加方法，在构造函数中等同于this.
    getAge = function() {}
    getAge = () => this.age

}
```

如果想要运行这段代码，请在构建环境中运行，或者通过 babel 的在线编译运行。

需要注意的是，constructor 方法是一个默认方法，当通过 new 声明实例时，会调用该方法，相当于构造函数。如果没有显式定义，那么将会添加一个空的默认方法。

2. 继承

与 ES5 相比，ES6 的继承要简单许多，直接来看一个例子。

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
```

```
    getName() {
        return this.name
    }
}

// Student类继承Person类
class Student extends Person {
    constructor(name, age, gender, classes) {
        super(name, age);
        this.gender = gender;
        this.classes = classes;
    }

    getGender() {
        return this.gender;
    }
}

const s = new Student('TOM', 20, 1, 3);
s.getName(); // TOM
s.getGender(); // 1
```

我们只需用一个 extends 关键字与 supper 方法，就能够实现继承了。

在子类的构造函数中必须调用super方法，它表示构造函数的继承，与ES5中利用 call/apply 继承构造函数的功能一样。

```
// 构造函数中
// ES6
super(name, age);

// ES5
Person.call(this);
```

关于面向对象的知识点，在之前讲解原型时几乎全部都有涉及，而 ES6 只是新增了语法糖，

并不涉及新的知识点，因此这里不再赘述。在原型的学习中，我们尝试了几个面向对象的例子，大家可以尝试使用 ES6 的语法重新实现一遍，需要改动的地方并不多。

10.8 模块化

在实际开发中，模块化 modules 的运用无处不在。每一个文件都是一个独立的模块。

一个模块需要对外提供接口，主要使用 `export` 命令，一个模块可以引入其他模块。

1. **在线的模块化学习环境**

codepen.io

通过创建一个免费的独立项目来编写我们的学习或者测试代码，支持 ES6 的所有语法。

2. **本地模块化开发环境搭建。**

如果自己学习如何配置本地环境，则需要额外去学习构建工具的使用，这将会花费许多时间，因此这里推荐借助别人已经构建好的环境，下载安装到本地即可。此处介绍的是 `create-react-app` 的使用。

（1）在自己的电脑上安装 node。

通常的安装方式是去 node 官网下载安装包，选择与自己电脑相符合的版本即可。下载地址：http://nodejs.cn/download/。

（2）安装命令行工具。

在 Windows 环境下，系统默认的 cmd 非常难用，因此推荐使用 git.bash 或者 cmder。

git 下载地址：https://git-scm.com/downloads。git 安装好之后，在安装目录下会有一个 bash 工具，双击打开就可以直接使用了。

cmder 下载地址：http://cmder.net/。

在 Mac 环境下就方便很多，可以直接使用系统自带的 terminal 工具，也可以使用一款叫作 iterm2 的工具。iterm2 下载地址：http://www.iterm2.com/downloads.html。

除此之外，还可以自行去网上搜索 iterm2 的相关插件与配置。

（3）安装包管理工具 npm 或者 yarn。

我们在开发代码的时候，经常会用到别人已经写好的库、插件等，因此需要一个包管理工具，以让我们的使用过程更加简单与方便，因此就有了 npm 与 yarn。

npm 在安装 node 时已经默认安装了，因此只需在命令行工具中使用如下指令验证一下是否安装成功即可。

```
// 查看node版本
> node -v

// 查看npm版本
> npm -v
```

yarn 是一个更新的包管理工具，具体的安装过程可到其官方网站中查看，地址如下：https://yarnpkg.com/zh-Hans/docs/install#mac-tab。

（4）安装 create-react-app。

```
npm install -g create-react-app

// 或者
> yarn global add create-react-app
```

安装完毕之后，就可以使用 `create-react-app` 来创建一个项目了。

（5）创建 create-react-app 项目。

创建并进入一个你要存放项目的目录，假设命名为 `develop`，运行以下指令。

```
> create-react-app es6app
```

create-react-app 会自动帮助我们在 develop 目录下创建一个叫作 es6app 的文件夹，这就是我们新创建的项目。

（6）运行项目。

使用如下命令行进入刚才创建的项目 es6app。

```
> cd es6app
```

可以使用创建项目时的提示指令来运行项目。

```
> yarn start

// 或者
> npm start
```

项目启动需要花费一点时间，启动之后，通常浏览器会自动打开一个新的页面运行项目，也可以在浏览器中输入 `http://localhost:3000` 来运行项目。

（7）认识项目。

◎ package.json 与 yarn.lock。

这两个文件是构建工具需要的配置信息与项目依赖包信息。目前暂时不用过多考虑它们的用处，等之后在深入学习构建工具时自然会明白它们的作用。

◎ node_modules。

项目依赖包的安装目录。当我们创建项目时，会根据配置文件中的依赖包信息，把所有需要的插件工具、模块等都安装在该目录下。

◎ public。

用于存放静态文件。主要作用是 HTML 入口文件的存放，也可以存放其他公用的静态资源，如图片、CSS 文件等。其中，index.html 就是项目的入口 HTML 文件。

◎ src。

组件的存放目录。在 create-react-app 创建的项目中，每一个单独的文件都可以被看成一个单独的模块、单独的 image、单独的 css、单独的 js 等，而所有的组件都存放于 src 目录中，其中，index.js 是 js 的入口文件。虽然并没有在 index.html 中使用 script 标签引用它，但是它的作用就是如此。

当然，create-react-app 创建项目的主要目的是帮助大家进行 React 的开发。因此利用它来学习 ES6 已是绰绰有余，并且能够为我们进一步学习 React 奠定一些基础。

10.8.1 基础语法

1.import

首先需要知道的一个常识是，每一个文件都是一个单独的模块。通过 `import` 指令，可以在当前模块中引入其他模块。

观察项目 `index.js` 中的代码。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';
```

```
ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

我们发现文件中大量使用了 import 指令。如果把 index.js 看成是项目的入口文件，那么这些由 import 指令加载的模块，则都是项目所需要的模块。

以 import React from 'react' 为例分析一下具体的使用规则。

◎ import 表示引入/加载一个模块。

◎ react 可以理解为这个模块的名字。它是 react 模块在创建时对外提供的所有接口组合成的对象，因此在代码中可以使用它来访问 react 模块提供的接口，例如 React.Component。正因为此处是一个对象，因此可以使用解析结构的形式来直接获取某个特定的接口，代码如下：

```
import React, { Component } from 'react';
```

大括号中接口的名字，必须与对外提供的接口名字相同，在某些必要的场合，当我们想要给接口换一个名字时，可以通过 as 来搞定。

```
// 通常修改名字是为了避免冲突
import React, { Component as OtherName } from 'react';
```

◎ from 表示模块来自于哪里。

◎ 此处的 react 表示模块的出处。通常情况下应该使用路径的形式表示，例如下面的 ./index.css，这样表示是因为构建工具能够自动识别安装在 node_modules 中的模块，而不用具体显示真实的路径，此处可以理解为一个简写。

观察比较仔细的同学应该已经在 index.js 的代码中发现了一些奇怪的地方，居然有一个 css 文件被当成模块加载进来了！

这正是构建工具（webpack）的强大之处，它可以把所有的文件都当成一个模块，css 可以是一个模块，图片也可以是一个模块。这样的强大机制让我们能够顺利地把模块化开发的思维再往前推进一步，那就是组件化思维。当然，组件化思维不是本书学习的重点，大家可以先行了解，待需要进一步学习 React 时再做深入了解。

除此之外，还有一个比较小的细节，当我们引入 .js 文件时，可以省略文件后缀名。

```
import App from './App.js';
// 通常都会省略
import App from './App';
```

引入的模块都会自行执行一次，因此当我们期望引入的模块仅仅只是执行，但是不必在当前模块使用时，就可以直接简写成：

```
import './test';
import './style.css';
```

2.export

一个模块需要有向外提供接口的能力，在之前的例子中我们还没有学会模块化语法时，简单粗暴地将对外的接口放在了 window 对象上，而在 ES6 的语法中，则是借助 `export` 命令来提供对外接口。

可以在同一个模块中多次使用 `export` 命令对外提供多个接口。

在项目的 src 目录下创建一个 `module01.js` 文件，代码如下：

```
// src/module01.js
export const name1 = 'TOM';
export const name2 = 'Jake';
```

当在其他模块（index.js）引入该模块时，如果仅仅只是引入其中的某一个接口，那么可以这样做：

```
import { name1 } from './module01';
```

但是如果需要引入该模块中所有的对外接口，那么一种方式是在大括号中将所有的名称都列出来，另外一种方式就是使用通配符与 as 配合。

```
import { name1, name2 } from './module01';

// or 利用别名的方式
```

```
import * as module01 from './module01';
// 那么就有
name1 = module01.name1
name2 = module01.name2
```

还可以通过 export default 来对外提供接口，这种情况下，对外接口通常是一个对象。

```
// 修改module01.js
const name1 = 'TOM';
const name2 = 'Jake';

export default {
    name1,
    name2
}
// ES6对象的简写语法
```

因此，在引入模块时代码的写法上需要做一些调整。例如，当模块中有 export default 命令抛出的接口时，那么引入模块就可以直接这样写：

```
import module01 from './module01';
```

此时的 module01 就是 export default 抛出的对象。

我们接着看一个新的例子，在 src 目录下创建一个新的文件 module02.js，代码如下：

```
// src/module02.js
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getName() {
        return this.name
    }
}
```

```
export default Person;
```

在其他模块中使用时：

```
// src/index.js
import Person from './module02';

const p1 = new Person('Tom', 20);
console.log(p1.getName());
```

需要注意的是，一个模块中只允许出现一次 export default 命令，不过可以同时拥有多个 export 与一个 export default。创建一个新模块 module03.js。

```
// src/module03.js
export function fn() {
    console.log('this is a function named fn.');
}

export function bar() {
    console.log('hello everybody.');
}

class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getName() {
        return this.name
    }
}

export default Person;
```

当我们引入这个新模块时，可以这样写。

```
import Person, { fn } from './module03';

// 当然也可以全部引用，实践中通常不会这样做
import * as module03 from './module03';

// module03的内容如下
module03 = {
    fn: fn,
    bar: bar,
    default: Person
}
```

10.8.2 实例

现在来实现一个场景。

首先有一个普通的正方形 div，并且有一堆设置按钮，现在想要通过这些按钮来控制 div 的显示/隐藏、背景颜色、边框颜色、长宽等属性，应该怎么做呢？

在实践中类似的场景很多，例如手机的设置、控制中心，以及每个网页都有的个人中心里的设置等。

当然，如果仅仅只是通过一个按钮来控制一个 div 的单一属性，那么非常简单，但实践中的场景往往更加复杂。第一个难点是我们可能会有更多的属性需要控制，也有更多的目标元素需要控制。第二个难点则是在构建代码之初，目标元素们可能存在于不同的模块中，如何通过单一的变量来控制不同的目标元素？

当项目变得越来越复杂时，需要管理的状态值也会变得越来越多，如果只是用我们初学时在当前作用域中随便定义一个变量的方式来做，那么，肯定是一场无法预料的灾难。我们可能需要用更多的时间去调试、去修复更多的 bug、去面对杂乱无章的代码，还需要克制烦躁的心态，以及无休止的加班。而当需求变动时，还不得不忍受自己都看不下去的代码，再次重复之前的痛苦。相信没有人愿意尝试这种经历，那么应该怎么办呢？

办法肯定是有的，目前非常流行的 Redux 就能够解决这样的问题。当然，Redux 虽然能独立使用，但在实践中往往和 React 搭配使用，且学习成本并不算小，对我们来说，目前强行去掌握它有点跨步太大，因此这里的方案是，针对准备要实现的效果，自己动手实现一个简单的状态管

理模块，这样不仅可以降低学习成本，还能够让我们对于状态管理有更加深刻的认知。下面就跟着我的思路，一起来试试吧。

1. 准备工作

首先找到之前创建的学习项目 es6app，然后删除 src 文件夹中除 index.js 外的所有文件，并将 index.js 的内容清空。现在项目已经被清空了。最后通过 `yarn start` 指令运行项目。

2. 状态管理模块

在 src 目录中新建一个状态管理模块，并命名为 `state.js`。

首先需要创建一个状态树。在整个项目中，状态树是唯一的，我们会把所有的状态名与状态值通过 key-value 的形式保存在状态树中。

```
const store = {}
```

当根据需求往状态树中保存状态时，状态树大概会变成如下形式：

```
store = {
    show: 0,
    backgroundColor: '#cccccc',
    width: '200',
    height: '200'
    // ... more
}
```

在学习闭包的实例中有提到过，每一个模块，都是一个单例模式，也是一个闭包，因此如果想要在其他模块中操作 store，那么就需要对外提供对应的接口。

根据需求，可以先大概列出可能会用到的方法，如果之后需要补充则再另行添加。可能会用到的方法如下。

- ◎ registerState：往状态树中放入新的状态值。
- ◎ getStore：获取整个状态树。
- ◎ getState：获取某一个状态值。
- ◎ setState：修改状态树中某一个状态值。

具体代码如下。

```
// 往store中添加一个状态值
export const registerState = (status, value) => {
    if (store[status]) {
        throw new Error('状态已经存在。')
        return;
    }
    store[status] = value;
    return value;
}

// 获取整个状态树
export const getStore = () => store

// 获取某个状态的值
export const getState = status => store[status]

// 设置某个状态的值
export const setState = (status, value) => {
    store[status] = value;
    dispatch(status, value);
    return value;
}
```

注意：为了简化学习，方法比较简单，没有过多考虑异常情况与健全处理，请勿直接运用于实践，实践可在此基础上根据需要扩展。

当通过交互改变状态值时，其实期待的是界面 UI 能够发生相应的改变。UI 的变动可能会比较简单，也可能会非常复杂，因此为了能够更好地维护 UI 改变，我们将每个 UI 变化用函数封装起来，并与对应的状态值对应。这样，当状态值改变时，调用一下对应的 UI 函数就能够实现界面的实时变动了。

因此，除需要一个 store 来保存状态值外，还需要一个 events 对象来保存 UI 函数。

```
const events = {}
```

状态值与 UI 函数的对应关系如下。

```
store = {
    show: 0,
    backgroundColor: '#cccccc',
    width: '200',
    height: '200'
    // ... more
}

events = {
    show: function() {},
    backgroundColor: function() {},
    width: function() {},
    height: function() {}
    // ... more
}

// 通过相同的状态命名，可以访问对应的状态值与函数
```

同样的道理，我们也需要提供几个能够操作 events 的方法。

```
// UI方法可以理解为一个绑定过程，因此命名为bind，在有的地方也称为订阅
export const bind = (status, eventFn) => {
    events[status] = eventFn;
}

// 移除绑定
export const remove = status => {
    events[status] = null;
    return status;
}

// 需要在状态值改变时触发UI的变化，因此在setState方法中调用了该方法
export const dispatch = (status, value) => {
    if (!events[status]) {
        throw new Error('未绑定任何事件！')
```

```
    }
    events[status](value);
    return value;
}
```

至此，一个简单的状态管理模块就完成了，接下来看看应该如何运用它。

完整代码如下。

```
// src/state.js
const store = {}
const events = {}

// 往store中添加一个状态值，并确保传入一个初始值
export const registerState = (status, value) => {
    if (store[status]) {
        throw new Error('状态已经存在。')
        return;
    }
    store[status] = value;
    return value;
}

// 获取整个状态树
export const getStore = () => store

// 获取某个状态的值
export const getState = status => store[status]

// 设置某个状态的值
export const setState = (status, value) => {
    store[status] = value;
    dispatch(status, value);
    return value;
}

// 将状态值与事件绑定在一起，通过status-events 的形式保存在events对象中
```

```
export const bind = (status, eventFn) => {
    events[status] = eventFn;
}

// 移除绑定
export const remove = status => {
    events[status] = null;
    return status;
}

export const dispatch = (status, value) => {
    if (!events[status]) {
        throw new Error('未绑定任何事件! ')
    }
    events[status](value);
    return value;
}
```

3. 注册状态值模块

我们需要管理的状态很多，当然，可以在每一个使用到这些状态值的模块中各自注册，而笔者更偏向于使用一个单独的模块来注册状态。如果担心自己会忘记状态值的作用，则建议对每一个状态都做好注释。

注册状态的方式就是利用状态管理模块中定义 registerState 的方法来完成。

```
// src/register.js
import { registerState } from './state';

// 控制显示隐藏
registerState('show', 0);

registerState('backgroundColor', '#FFF');

registerState('borderColor', '#000');

registerState('width', 100);
```

```
registerState('height', 100);
//... and more
```

4. 功能函数模块

每一个项目中都有这样一个功能函数模块，我们会把一些封装好的功能性的方法都放到这个模块中去。我们在实践中常常会遇到各种需求，例如，取一个数组中最大的那个值，获取 url 中某个属性对应的具体值，对时间格式按需进行处理等，这时就可以直接将这些操作封装好，存放到工具函数模块中，在使用时引用即可。

当然，这个例子中不会用到特别多的功能函数，因此这里只封装了一个，在下一个例子中我们会封装更多的功能函数。

```
// src/utils.js

// 获取DOM元素属性值
export const getStyle = (obj, key) => {
    return obj.currentStyle ? obj.currentStyle[key] : document.
    defaultView.getComputedStyle( obj, false )[key];
}
```

除此之外，也可以引入 lodash.js 这样的工具库。lodash 是一个具有一致接口、模块化、高性能的工具库，它封装了许多常用的工具函数，在实践开发中非常有用。

5. 目标元素模块

目标元素，也就是可能会涉及 UI 改变的元素。之前在创建状态管理模块时已经提到，我们需要将 UI 改变的动作封装为函数，并保存/绑定到 events 对象中。这个操作就选择在目标元素模块中来完成。

首先在 public/index.html 中写入一个 div 元素。

```
.target {
    width: 100px;
    height: 100px;
    background: #CCC;
```

```
    /* display: none; */
    transition: 0.3s;
}
.target.hide {
    display: none;
}
```

此处目标元素是一个正方形的 div 元素，我们将会通过控制按钮来改变它的显示/隐藏、边框、背景、长宽等属性。因此该模块的主要功能就是根据注册的状态变量，绑定 UI 变化的函数，具体代码如下。

```
// src/box.js
import { bind } from './state';
import { getStyle } from './utils';
import './register';

const div = document.querySelector('.target');

// control show or hide
bind('show', value => {
    if (value === 1) {
        div.classList.add('hide');
    }
    if (value === 0) {
        div.classList.remove('hide');
    }
})

// change background color
bind('backgroundColor', value => {
    div.style.backgroundColor = value;
})

// change border color
bind('borderColor', value => {
    const width = parseInt(getStyle(div, 'borderWidth'));
```

```
    if (width === 0) {
        div.style.border = '2px solid #ccc';
    }
    div.style.borderColor = value;
})

// change width
bind('width', value => {
    div.style.width = `${value}px`;
})

bind('height', value => {
    div.style.height = `${value}px`;
})
```

6. 控制模块

我们可能会通过按钮、input 框，或者滑块等不同的方式来改变状态值，因此控制模块将会是一个比较复杂的模块。为了更好地组织代码，一个可读性和可维护性都很强的方式是将整个控制模块拆分为许多小模块，每一个小模块只完成一个状态值的控制操作。

因此我们需要根据需求，分别创建对应的控制模块。

首先在 public/index.html 中添加相应的 HTML 代码。

```
<div class="control_wrap">
    <div><button class="show">show/hide</button></div>
    <div>
        <input class="bgcolor_input" type="text" placeholder="input
        background color" />
        <button class="bgcolor_btn">sure</button>
    </div>
    <div>
        <input type="text" class="bdcolor_input" placeholder="input
        border color" />
        <button class="bdcolor_btn">sure</button>
    </div>
```

```
    <div>
        <span>width</span>
        <button class="width_reduce">-5</button>
        <button class="width_add">+5</button>
    </div>
    <div>
        <span>height</span>
        <button class="height_reduce">-</button>
        <input type="text" class="height_input" readonly>
        <button class="height_add">+</button>
    </div>
</div>
```

现在在 src 目录下创建一个 controlBtns 文件夹，该文件夹中全部用来存放控制模块，然后依次编写控制模块的代码即可。

◎ 控制目标元素显示/隐藏的模块。

```
// src/controlBtns/showBtn.js
import { getState, setState } from '../state';

const btn = document.querySelector('.show');

btn.addEventListener('click', () => {
    if (getState('show') == 0) {
        setState('show', 1);
    } else {
        setState('show', 0)
    }
}, false);
```

◎ 控制目标元素背景颜色变化的模块。

```
// src/controlBtns/bgColor.js
import { setState } from '../state';
```

```
const input = document.querySelector('.bgcolor_input');
const btn = document.querySelector('.bgcolor_btn');

btn.addEventListener('click', () => {
    if (input.value) {
        setState('backgroundColor', input.value);
    }
}, false);
```

◎ 控制目标元素边框颜色变化的模块。

```
// src/controlBtns/bdColor.js
import { setState } from '../state';

const input = document.querySelector('.bdcolor_input');
const btn = document.querySelector('.bdcolor_btn');

btn.addEventListener('click', () => {
    if (input.value) {
        setState('borderColor', input.value);
    }
}, false);
```

◎ 控制目标元素宽度变化的模块。

```
// src/controlBtns/width.js
import { getState, setState } from '../state';

const red_btn = document.querySelector('.width_reduce');
const add_btn = document.querySelector('.width_add');

red_btn.addEventListener('click', () => {
    const cur = getState('width');
    if (cur > 50) {
        setState('width', cur - 5);
    }
```

```
}, false)

add_btn.addEventListener('click', () => {
    const cur = getState('width');
    if (cur < 400) {
        setState('width', cur + 5);
    }
}, false)
```

◎ 控制目标元素高度变化的模块。

```
// src/controlBtns/height.js
import { getState, setState } from '../state';

const red_btn = document.querySelector('.height_reduce');
const add_btn = document.querySelector('.height_add');
const height_input = document.querySelector('.height_input');

height_input.value = getState('height') || 100;

red_btn.addEventListener('click', () => {
    const cur = getState('height');
    if (cur > 50) {
        setState('height', cur - 5);
        height_input.value = cur - 5;
    }
}, false)

add_btn.addEventListener('click', () => {
    const cur = getState('height');
    if (cur < 400) {
        setState('height', cur + 5);
        height_input.value = cur + 5;
    }
}, false)
```

最后将这些模块整合起来。

```
// src/controlBtns/index.js
import './showBtn';
import './bgColor';
import './bdColor';
import './width';
import './height';
```

在构建工具中，如果引入一个文件夹当作模块，那么相当于默认引入的是该文件下名为 index.js 的模块，因此可以通过在 controlBtns 文件夹下创建 index.js 的方式，来让该文件夹成为一个模块。

也就是说，在引入这个模块时：

```
import './controlBtns';

// 等价于
import './controlBtns/index';
```

细心的读者肯定已经发现了，我们给按钮绑定点击事件时，仅仅对状态值做了改变，而没有考虑对应的 UI 变化，这是为什么呢?

在以前的开发经验中，要想改变一个元素的某个属性，一般来说会有状态值的变化，并且还有对应的 UI 操作，而这里的做法好像不太一样。

其实这里是利用一个例子，带大家尝试一下分层的开发思维。在这个例子中，我们将状态控制设定为控制层，而 UI 变化设定为 view 层。我们只需在目标元素模块中，将 view 层的变化封装好，然后利用状态管理模块中的机制，在控制层只需考虑状态值的变化即可。

这样处理之后，我们的开发重心就从考虑整个界面的变化，转移到了仅仅考虑状态值的变化。这样做的好处是，极大地简化了我们在实现需求过程中所需考虑的问题。在未来的进阶学习中，大家可能还会大量接触到这样的开发思路。

7. 拼合模块

在 src 目录下的 index.js 文件中，可以通过 import 将需要的模块拼合起来。

```
// src/index.js
import './controlBtns';
import './box';

import './index.css';
```

至此，我们需要的项目就已经全部完成了。如果你一直在跟着动手实践，相信现在已经能够看到项目的最终效果了。整个项目的相关目录结构如下所示。

```
+ public
  - index.html
+ src
  - index.js
  - index.css
  - box.js
  - state.js
  - utils.js
  - register.js
  + controlBtns
    - index.js
    - showBtn.js
    - bgColor.js
    - bdColor.js
    - width.js
    - height.js
```

8. 项目小结

模块化的开发思路，实际上是通过视觉元素、功能性等原则，将代码划分为一个个拥有各自独立职能的模块。我们通过 ES6 的 modules 语法，按需将这些模块组合起来，并借助构建工具打包成我们所熟知的 js 文件。

当然，在实践中可能会遇到更复杂的情况。例如，目标元素并非单一元素的改变，而是整个区域发生变化；又或者控制目标元素变化的好几个状态值同时发生变化，带来性能问题等。其实并不用太过担心，这些问题虽然都是新的挑战，但是相信大家在掌握了本书中的知识点后，花点时间去调试是能够克服这些挑战的。这也是从初级往高级进步的必经之路。

大家也可以主动在此例子的基础上去增加复杂度。例如，新增多个目标元素，让目标元素某个属性同时由几个状态值控制等。

投稿联络：安 娜
微信&QQ：80303489
邮箱：anna@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

JavaScript
核心技术开发解密

本书将整个JavaScript 相关的知识点简单地划分为核心知识与周边知识。核心知识是整个前端知识体系的骨架所在。它们前后依赖，环环相扣。如果知道这条核心知识链到底是什么，并且彻底地掌握它们，那么你就已经具备了成为一名优秀前端程序员的能力。这样的能力能够让你在学习其他知识点的时候方向明确，并且充满底气。所以这本书的主要目的就是帮助读者拥有这样的进阶能力。

本书将一步步与大家分享这条完整的核心链。期望当大家学完本书的知识后，能够对前端开发的现状有一个大致的了解，知道什么知识是有用的，什么知识是工作中需要的，拥有进一步学习流行前端框架的能力，拥有在前端方向自主学习、自主进步的知识基础与能力。

上架建议：数据库

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：安　娜
封面设计：李　玲

ISBN 978-7-121-33696-6

定价：69.00元