变量、作用域与内存

原始值与引用值

ES的变量可以包含两种不同类型的数据：原始值与引用值
原始值：最简单的数据
引用值：多个值构成的对象
保存原始值的变量是按值（by value）访问的，操作的就是实际值

引用值是保存在内存中的对象：

- JS不允许直接访问内存地址，不能直接操作对象所在的内存空间
- 操作对象，实际上操作的是 对该对象的引用，并非对象本身
- 保存引用值的变量是按引用（by reference）访问的

动态属性

引用值可以随时添加、修改和删除属性和方法
原始值没有属性，添加属性不会报错，但是无用

```
let name = "Nicholas";
name.age = 27;
console.log(name.age); // undefined
```

复制值

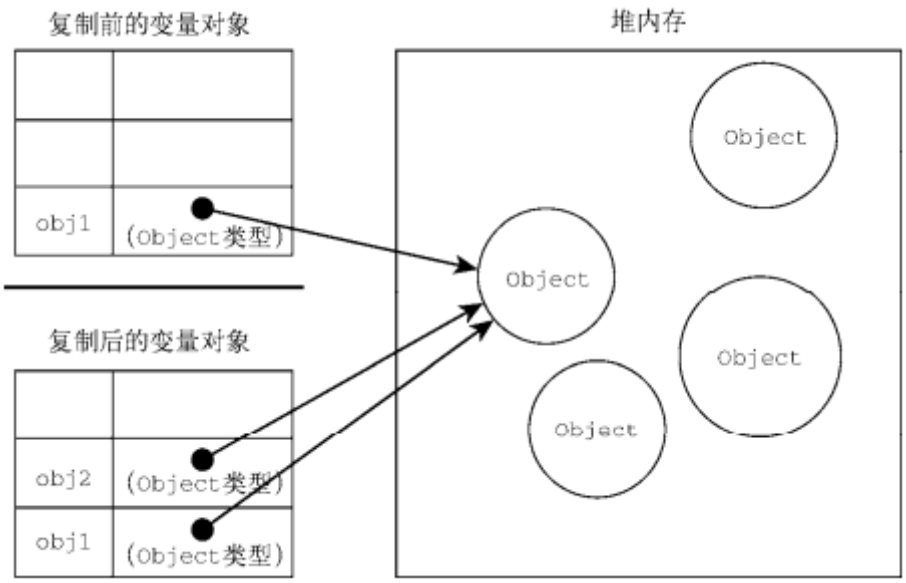
原始值复制，复制的是副本，两个变量可以独立使用，互不干扰

```
let num1 = 5;
let num2 = num1
```

图 4-1

引用值复制，复制的是指针，指向存储在堆内存中的对象

```
let obj1 = new Object();
let obj2 = obj1;
obj1.name = "Nicholas";
console.log(obj2.name); // "Nicholas"
```



传参

ES中所有函数的参数都是按值传递

个人理解：

原始值：直接复制

引用值：复制指针

```
function setName(obj) {
 obj.name = "Nicholas";
}
```

```
let person = new Object();
setName(person);
console.log(person.name); // "Nicholas"

function setName(obj) {
 obj.name = "Nicholas";
 obj = new Object();
 obj.name = "Greg";
}
let person = new Object();
setName(person);
console.log(person.name); // "Nicholas"
```

ES中函数的参数就是局部变量!

确定类型

由于typeof 无法区分对象的类型，提供instance of操作符
result = variable instanceof constructor

```
console.log(person instanceof Object); // 变量 person 是 Object 吗?
console.log(colors instanceof Array); // 变量 colors 是 Array 吗?
console.log(pattern instanceof RegExp); // 变量 pattern 是 RegExp 吗
```

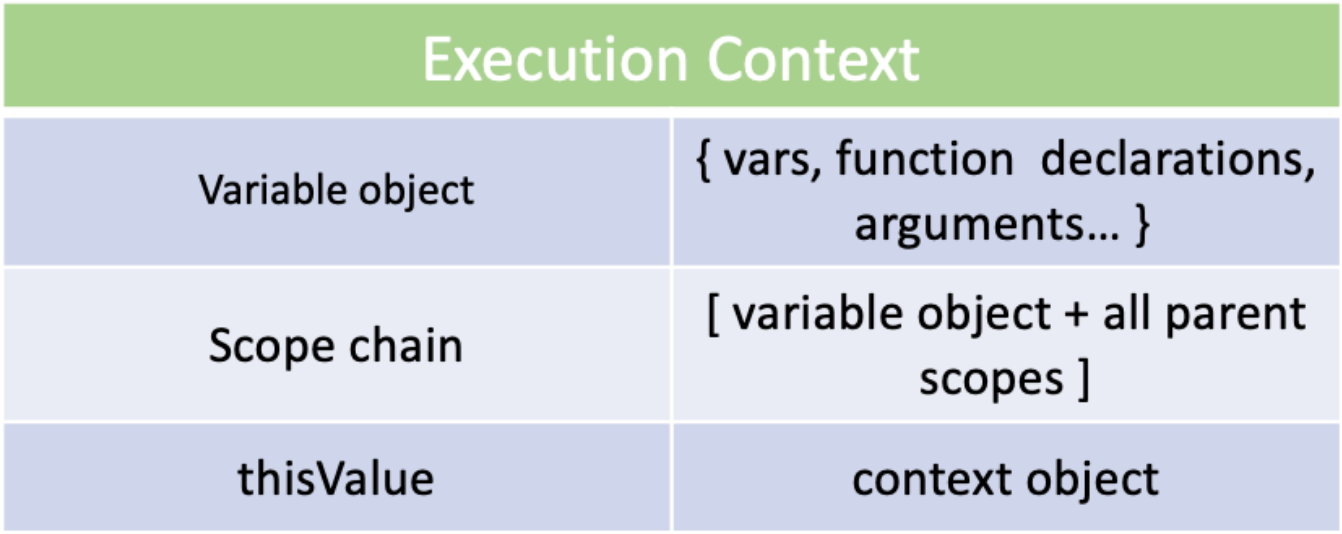
如果变量是给定的引用类型（由其原型链决定）的实例，则instanceof返回true

所有引用值都是Object的实例

执行上下文与作用域 (很绕，其实一直都知道)

在 JavaScript 代码运行时，解释执行全局代码、调用函数或使用 eval 函数执行一个字符串表达式都会创建并进入一个新的执行环境，而这个执行环境被称之为执行上下文。因此执行上下文有三类：全局执行上下文、函数执行上下文、eval 函数执行上下文。

- 变量或函数的执行上下文决定了他们可以访问哪些数据，他们的行为
- 每个执行上下文都有一个关联的变量对象
- 执行上下文中的所有变量和函数都存在这个对象上
- 代码无法访问变量对象，但是后台处理数据会用到
- 执行上下文在所有代码执行完毕后会被销毁掉，包括变量和函数
- 全局上下文在应用程序退出前才会被销毁，如关闭网页，退出浏览器



Variable object：变量对象，用于存储被定义在执行上下文中的变量 (variables) 和函数声明 (function declarations)

Scope chain：作用域链，是一个对象列表 (list of objects)，用以检索上下文代码中出现的标识符 (identifiers)

thisValue：this指针，是一个与执行上下文相关的特殊对象，也被称之为上下文对象

浏览器中全局上下文就是window对象

执行上下文代码执行时，创建变量对象的一个作用域链，决定了各级上下文的代码在访问变量和函数时的顺序。

执行上下文是函数：活动对象用对变量对象，最初只有一个定义变量：arguments

作用域链中的下一个变量对象来自包含上下文，再下一个对象来自再下一个包含上下文。以此类推直至全局上下文；全局上下文的变量对象始终是作用域链的最后一个变量对象。

```
var color = "blue";
function changeColor() {
 let anotherColor = "red";
 function swapColors() {
 let tempColor = anotherColor;
 anotherColor = color;
 color = tempColor;
 // 这里可以访问 color、anotherColor 和 tempColor
 }
 // 这里可以访问 color 和 anotherColor，但访问不到 tempColor
 swapColors();
}
// 这里只能访问 color
changeColor();
```

作用域链增强

某些语句会导致在作用域链前端临时添加一个上下文，在代码执行后被删除。

- try/catch 的catch

- with

with，作用域链前端添加指定的对象；
catch，创建一个全新的变量对象，包含要抛出的错误对象的声明
IE8之前catch捕捉的错误添加到执行上下文的变量对象中，导致catch外可以访问到错误。IE9修正了这个问题

变量声明

- var的函数作用域声明
  使用var声明变量时，变量会自动添加到最接近的上下文
  如果未被声明就被初始化了，就自动添加到全局上下文中（全局变量） 严格模式报错 var变量提升
- let的块级作用域声明
  同一作用域内，let反复声明报错，var重复声明被忽略
- const的常量声明
  与let相同，但是必须初始化，且后面不能改
  对象不能改引用值，但可以改键
- 标识符查找
  找最近的执行上下文，找不到往上找
  标识符查找有代价，但是JS做了优化，差异微乎其微

```
var color = 'blue';
function getColor() {
 let color = 'red';
 {
 let color = 'green';
 return color;
 }
}
console.log(getColor()); // 'green'
```

垃圾回收

垃圾回收：执行环境在代码执行时管理内存。
通过自动内存管理实现内存分配和闲置资源回收
思路：确定哪个变量不会再用，释放内存。

垃圾回收执行是周期性的

并不完美，存在不可判定的情况

标记清理

JS最常用的垃圾回收策略
给变量加标记的方式有很多

举例：
进入上下文时添加，存在于上下文 标记
离开上下文时添加，离开上下文 标记

过程：

1. 垃圾回收程序运行时，标记内存中存储的所有变量
2. 将所有上下文中的变量，以及在被上下文中变量引用的变量标记去掉
3. 内存清理：销毁带标记的所有制并回收内存

引用计数

并不常用
对每个值记录被引用测次数。

- 被引用+1
- 引用者将其覆盖 -1
- 引用值为0时，回收

有bug，循环引用等等

```
function problem() {
 let objectA = new Object();
 let objectB = new Object();
 objectA.someOtherObject = objectB;
 objectB.anotherObject = objectA;
}
```

通过属性值循环引用，离开上下文，引用数都是2

IE8以及更早版本的IE，BOM/DOM对象都是C+++实现的，叫COM（component object model）
COM使用引用计数实现垃圾回收
所以这些IE只要涉及到COM就会有bug

```
let element = document.getElementById("some_element");
let myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

如何避免循环引用

```
myObject.element = null;
element.someObject = null;
```

性能

垃圾回收程序周期性运行，时间调度很重要

V8:堆增长策略
在一次完整的垃圾回收之后，根据活跃对象的数量外加一些余量来确定何时再次垃圾回收

性能：取决于 时间调度 策略

内存管理

桌面软件内存>浏览器内存>移动浏览器内存

优化内存占用最佳手段：只保存必要数据，不再必要设置为null，释放引用，解除引用
解除引用适合全局变量和全局对象属性，局部变量在离开作用域后自动解除

方案：

1. 通过const和let声明提升性能

2. 隐藏类和删除操作
   V8解释js代码，会把创建的对象与隐藏类关联，跟踪属性特征，如果能够共享隐藏类，就共享，节约内存
   隐藏类的弊端与解决方案：

```
function Article() {
 this.title = 'Inauguration Ceremony Features Kazoo Band';
}
let a1 = new Article();
let a2 = new Article();
a2.author = 'Jake';

// a1,a2无法共享创建了两个隐藏类，反而造成了负担

function Article(opt_author) {
 this.title = 'Inauguration Ceremony Features Kazoo Band';
 this.author = opt_author;
}
let a1 = new Article();
let a2 = new Article('Jake');
```

   使用delete也能导致生成相同隐藏类片段

```
function Article() {
 this.title = 'Inauguration Ceremony Features Kazoo Band';
 this.author = 'Jake';
}
let a1 = new Article();
let a2 = new Article();
delete a1.author;

// 即使用了同一个构造函数，也不再共享一个隐藏类
// 最好把不要的属性设置为null
// 隐藏类共享+引用计数垃圾回收
```

3. 内存泄露

   - 意外声明的全局变量

```
function setName() {
 name = 'Jake';
}
```

- 定时器造成内存泄漏

```
let name = 'Jake';
setInterval(() => {
 console.log(name);
}, 100);
// 用了闭包，name一直占用内存
```

- 闭包很容易造成内存泄漏
  如果闭包引用内容很大，问题就很大了！！

```
let outer = function() {
 let name = 'Jake';
 return function() {
 return name;
 };
};
```

4. 静态分配和对象池
压榨浏览器，避免多余的垃圾回收

```
function addVector(a, b) {
 let resultant = new Vector();
 resultant.x = a.x + b.x;
 resultant.y = a.y + b.y;
 return resultant;
}
```

频繁的替换会让垃圾回收调度加快，从而影响性能
解决方案:

```
function addVector(a, b, resultant) {
 resultant.x = a.x + b.x;
 resultant.y = a.y + b.y;
 return resultant;
}
```

外部依然需要实例化，但是函数的行为没有改变，依然会触发垃圾回收。
策略：对象池
在初始化的某一时刻，可以创建一个对象池，用来管理一组可回收的对象。应用程序可以向这个对象池请求一个对象、设置其属性、使用它，然后在操作完成后再把它还给对象池。由于没发生对象初始化，垃圾回收探测就不会发现有对象更替，因此垃圾回收程序就不会那么频繁地运行
伪实现

```
// vectorPool 是已有的对象池
let v1 = vectorPool.allocate();
let v2 = vectorPool.allocate();
let v3 = vectorPool.allocate();
v1.x = 10;
v1.y = 5;
v2.x = -3;
```

```
v2.y = -6;
addVector(v1, v2, v3);
console.log([v3.x, v3.y]); // [7, -1]
vectorPool.free(v1);
vectorPool.free(v2);
vectorPool.free(v3);
// 如果对象有属性引用了其他对象
// 则这里也需要把这些属性设置为 null
v1 = null;
v2 = null;
v3 = null;
```

对象池一般用数组实现。且创建时要想好大小多大，不然超出大小会删除原大小，创建更大的数组，这个动作会引来垃圾回收!

静态分配 典型的内存换性能。除非被垃圾回收严重影响性能，否则不考虑

总结

原始值和引用值的特点:

- 原始值大小固定，因此保存在栈内存上。
- 从一个变量到另一个变量复制原始值会创建该值的第二个副本。
- 引用值是对象，存储在堆内存上。
- 包含引用值的变量实际上只包含指向相应对象的一个指针，而不是对象本身。
- 从一个变量到另一个变量复制引用值只会复制指针，因此结果是两个变量都指向同一个对象。
- typeof 操作符可以确定值的原始类型，而 instanceof 操作符用于确保值的引用类型。

执行上下文:

- 执行上下文分全局上下文、函数上下文和块级上下文。
- 代码执行流每进入一个新上下文，都会创建一个作用域链，用于搜索变量和函数。
- 函数或块的局部上下文不仅可以访问自己作用域内的变量，而且也可以访问任何包含上下文乃至全局上下文中的变量。
- 全局上下文只能访问全局上下文中的变量和函数，不能直接访问局部上下文中的任何数据。
- 变量的执行上下文用于确定什么时候释放内存。

垃圾回收:

- 离开作用域的值会被自动标记为可回收，然后在垃圾回收期间被删除。
- 主流的垃圾回收算法是标记清理，即先给当前不使用的值加上标记，再回来回收它们的内存。
- 引用计数是另一种垃圾回收策略，需要记录值被引用了多少次。JavaScript 引擎不再使用这种算法，但某些旧版本的 IE 仍然会受这种算法的影响，原因是 JavaScript 会访问非原生 JavaScript 对象（如 DOM 元素，实际是COM的原因）。
- 引用计数在代码中存在循环引用时会出现问题。
- 解除变量的引用不仅可以消除循环引用，而且对垃圾回收也有帮助。为促进内存回收，全局对象、全局对象的属性和循环引用都应该在不需要时解除引用。（设置为null）