32.2.3. 小结

有了垃圾回收机制，不代表不用关注内存泄露。那些很占空间的值，一旦不再用到，需要检查是否还存在对它们的引用。如果是的话，就必须手动解除引用

32.3. 常见内存泄露情况

意外的全局变量

JavaScript | 复制代码

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

另一种意外的全局变量可能由 `this` 创建：

JavaScript | 复制代码

```
function foo() {
    this.variable = "potential accidental global";
}
// foo 调用自己，this 指向了全局对象（window）
foo();
```

上述使用严格模式，可以避免意外的全局变量

定时器也常会造成内存泄露

JavaScript | 复制代码

```
var someResource = getData();
setInterval(function() {
    var node = document.getElementById('Node');
    if(node) {
        // 处理 node 和 someResource
        node.innerHTML = JSON.stringify(someResource));
    }
}, 1000);
```

如果 `id` 为Node的元素从 `DOM` 中移除，该定时器仍会存在，同时，因为回调函数中包含对 `someResource` 的引用，定时器外面的 `someResource` 也不会被释放

包括我们之前所说的闭包，维持函数内局部变量，使其得不到释放

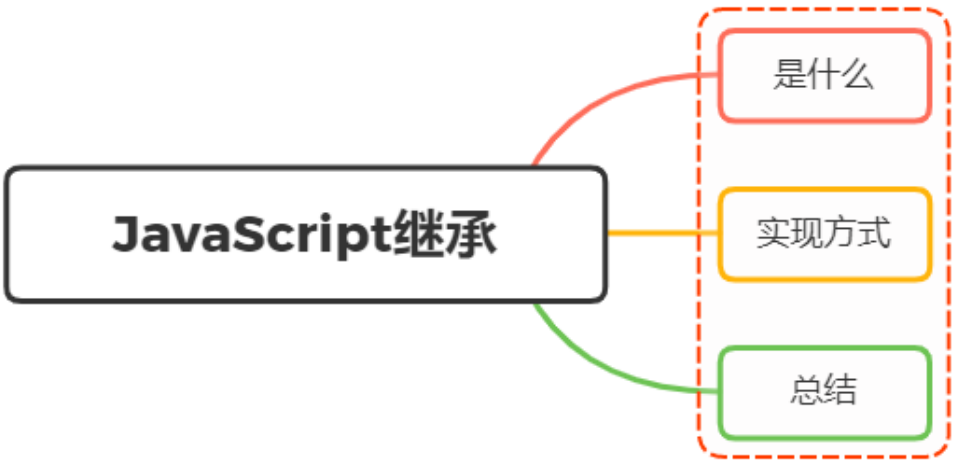
```javascript
function bindEvent() {
  var obj = document.createElement('XXX');
  var unused = function () {
    console.log(obj, '闭包内引用obj obj不会被释放');
  };
  obj = null; // 解决方法
}
```

没有清理对 `DOM` 元素的引用同样造成内存泄露

```javascript
const refA = document.getElementById('refA');
document.body.removeChild(refA); // dom删除了
console.log(refA, 'refA'); // 但是还存在引用能console出整个div 没有被回收
refA = null;
console.log(refA, 'refA'); // 解除引用
```

包括使用事件监听 `addEventListener` 监听的时候，在不监听的情况下使用 `removeEventListener` 取消对事件监听

33. Javascript如何实现继承?



33.1. 是什么

继承（inheritance）是面向对象软件技术当中的一个概念。

如果一个类别B“继承自”另一个类别A，就把这个B称为“A的子类”，而把A称为“B的父类别”也可以称“A是B的超类”

- 继承的优点

继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码

在子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能

虽然 `JavaScript` 并不是真正的面向对象语言，但它天生的灵活性，使应用场景更加丰富

关于继承，我们举个形象的例子：

定义一个类（Class）叫汽车，汽车的属性包括颜色、轮胎、品牌、速度、排气量等

JavaScript | 复制代码

```
class Car{
    constructor(color,speed){
        this.color = color
        this.speed = speed
        // ...
    }
}
```

由汽车这个类可以派生出“轿车”和“货车”两个类，在汽车的基础属性上，为轿车添加一个后备厢、给货车添加一个大货箱

JavaScript | 复制代码

```
// 货车
class Truck extends Car{
    constructor(color,speed){
        super(color,speed)
        this.Container = true // 货箱
    }
}
```

这样轿车和货车就是不一样的，但是二者都属于汽车这个类，汽车、轿车继承了汽车的属性，而不需要再次在“轿车”中定义汽车已经有的属性

在“轿车”继承“汽车”的同时，也可以重新定义汽车的某些属性，并重写或覆盖某些属性和方法，使其获得与“汽车”这个父类不同的属性和方法

```JavaScript
class Truck extends Car{
    constructor(color,speed){
        super(color,speed)
        this.color = "black" //覆盖
        this.Container = true // 货箱
    }
}
```

从这个例子中就能详细说明汽车、轿车以及卡车之间的继承关系

33.2. 实现方式

下面给出 `JavaScripy` 常见的继承方式：

- 原型链继承
- 构造函数继承（借助 call）
- 组合继承
- 原型式继承
- 寄生式继承
- 寄生组合式继承

33.2.1. 原型链继承

原型链继承是比较常见的继承方式之一，其中涉及的构造函数、原型和实例，三者之间存在着一定的关系，即每一个构造函数都有一个原型对象，原型对象又包含一个指向构造函数的指针，而实例则包含一个原型对象的指针

举个例子

```JavaScript
function Parent() {
    this.name = 'parent1';
    this.play = [1, 2, 3]
  }
  function Child() {
    this.type = 'child2';
  }
  Child1.prototype = new Parent();
  console.log(new Child())
```

上面代码看似没问题，实际存在潜在问题

```JavaScript
var s1 = new Child2();
var s2 = new Child2();
s1.play.push(4);
console.log(s1.play, s2.play); // [1,2,3,4]
```

改变 `s1` 的 `play` 属性，会发现 `s2` 也跟着发生变化了，这是因为两个实例使用的是同一个原型对象，内存空间是共享的

33.2.2. 构造函数继承

借助 `call` 调用 `Parent` 函数

```javascript
function Parent(){
    this.name = 'parent1';
}

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

function Child(){
    Parent1.call(this);
    this.type = 'child'
}

let child = new Child();
console.log(child);  // 没问题
console.log(child.getName());  // 会报错
```

可以看到，父类原型对象中一旦存在父类之前自己定义的方法，那么子类将无法继承这些方法

相比第一种原型链继承方式，父类的引用属性不会被共享，优化了第一种继承方式的弊端，但是只能继承父类的实例属性和方法，不能继承原型属性或者方法

33.2.3. 组合继承

前面我们讲到两种继承方式，各有优缺点。组合继承则将前两种方式继承起来

JavaScript | 复制代码

```javascript
function Parent3 () {
    this.name = 'parent3';
    this.play = [1, 2, 3];
}

Parent3.prototype.getName = function () {
    return this.name;
}
function Child3() {
    // 第二次调用 Parent3()
    Parent3.call(this);
    this.type = 'child3';
}

// 第一次调用 Parent3()
Child3.prototype = new Parent3();
// 手动挂上构造器，指向自己的构造函数
Child3.prototype.constructor = Child3;
var s3 = new Child3();
var s4 = new Child3();
s3.play.push(4);
console.log(s3.play, s4.play);  // 不互相影响
console.log(s3.getName()); // 正常输出'parent3'
console.log(s4.getName()); // 正常输出'parent3'
```

这种方式看起来就没什么问题，方式一和方式二的问题都解决了，但是从上面代码我们也可以看到 `Parent3` 执行了两次，造成了多构造一次的性能开销

33.2.4. 原型式继承

这里主要借助 `Object.create` 方法实现普通对象的继承

同样举个例子

JavaScript | 复制代码

```javascript
let parent4 = {
    name: "parent4",
    friends: ["p1", "p2", "p3"],
    getName: function() {
      return this.name;
    }
  };

  let person4 = Object.create(parent4);
  person4.name = "tom";
  person4.friends.push("jerry");

  let person5 = Object.create(parent4);
  person5.friends.push("lucy");

  console.log(person4.name); // tom
  console.log(person4.name === person4.getName()); // true
  console.log(person5.name); // parent4
  console.log(person4.friends); // ["p1", "p2", "p3","jerry","lucy"]
  console.log(person5.friends); // ["p1", "p2", "p3","jerry","lucy"]
```

这种继承方式的缺点也很明显，因为 `Object.create` 方法实现的是浅拷贝，多个实例的引用类型属性指向相同的内存，存在篡改的可能

33.2.5. 寄生式继承

寄生式继承在上面继承基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法

JavaScript | 复制代码

```javascript
let parent5 = {
    name: "parent5",
    friends: ["p1", "p2", "p3"],
    getName: function() {
        return this.name;
    }
};

function clone(original) {
    let clone = Object.create(original);
    clone.getFriends = function() {
        return this.friends;
    };
    return clone;
}

let person5 = clone(parent5);

console.log(person5.getName()); // parent5
console.log(person5.getFriends()); // ["p1", "p2", "p3"]
```

其优缺点也很明显，跟上面讲的原型式继承一样

33.2.6. 寄生组合式继承

寄生组合式继承，借助解决普通对象的继承问题的 `Object.create` 方法，在前面几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对最优的继承方式

JavaScript | 复制代码

```javascript
function clone (parent, child) {
    // 这里改用 Object.create 就可以减少组合继承中多进行一次构造的过程
    child.prototype = Object.create(parent.prototype);
    child.prototype.constructor = child;
}

function Parent6() {
    this.name = 'parent6';
    this.play = [1, 2, 3];
}
Parent6.prototype.getName = function () {
    return this.name;
}
function Child6() {
    Parent6.call(this);
    this.friends = 'child5';
}

clone(Parent6, Child6);

Child6.prototype.getFriends = function () {
    return this.friends;
}

let person6 = new Child6();
console.log(person6); //{friends:"child5",name:"child5",play:[1,2,3],__proto__:Parent6}
console.log(person6.getName()); // parent6
console.log(person6.getFriends()); // child5
```

可以看到 person6 打印出来的结果，属性都得到了继承，方法也没问题

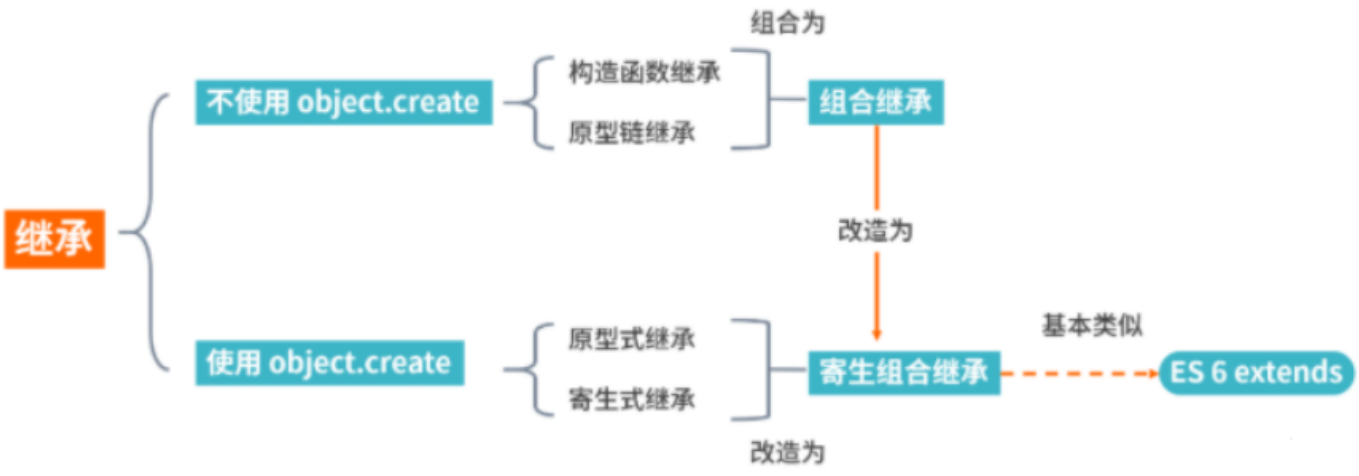
文章一开头，我们是使用 `ES6` 中的 `extends` 关键字直接实现 `JavaScript` 的继承

```javascript
class Person {
  constructor(name) {
    this.name = name
  }
  // 原型方法
  // 即 Person.prototype.getName = function() { }
  // 下面可以简写为 getName() {...}
  getName = function () {
    console.log('Person:', this.name)
  }
}
class Gamer extends Person {
  constructor(name, age) {
    // 子类中存在构造函数，则需要在使用“this”之前首先调用 super()。
    super(name)
    this.age = age
  }
}
const asuna = new Gamer('Asuna', 20)
asuna.getName() // 成功访问到父类的方法
```

利用 `babel` 工具进行转换，我们会发现 `extends` 实际采用的也是寄生组合继承方式，因此也证明了这种方式是较优的解决继承的方式

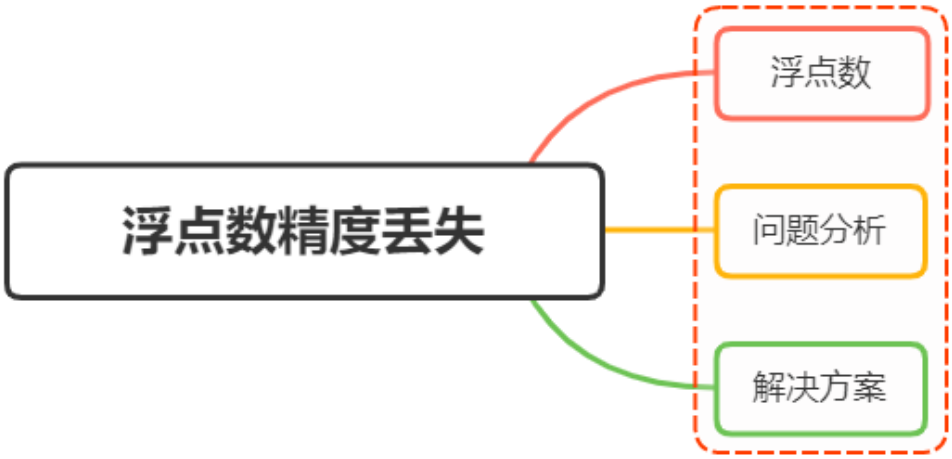
33.3. 总结

下面以一张图作为总结：



通过 `Object.create` 来划分不同的继承方式，最后的寄生式组合继承方式是通过组合继承改造之后的最优继承方式，而 `extends` 的语法糖和寄生组合继承的方式基本类似

34. 说说 Javascript 数字精度丢失的问题，如何解决?



34.1. 场景复现

一个经典的面试题

```
JavaScript | 复制代码
1  0.1 + 0.2 === 0.3 // false
```

为什么是 `false` 呢?

先看下面这个比喻

比如一个数 1÷3=0.33333333.....

3会一直无限循环，数学可以表示，但是计算机要存储，方便下次取出来再使用，但0.333333..... 这个数无限循环，再大的内存它也存不下，所以不能存储一个相对于数学来说的值，只能存储一个近似值，当计算机存储后再取出时就会出现精度丢失问题

34.2. 浮点数

“浮点数”是一种表示数字的标准，整数也可以用浮点数的格式来存储

我们也可以理解成，浮点数就是小数

在 `JavaScript` 中，现在主流的数值类型是 `Number` ，而 `Number` 采用的是 `IEEE754` 规范中64位双精度浮点数编码

这样的存储结构优点是可以归一化处理整数和小数，节省存储空间

对于一个整数，可以很轻易转化成十进制或者二进制。但是对于一个浮点数来说，因为小数点的存在，小数点的位置不是固定的。解决思路就是使用科学计数法，这样小数点位置就固定了

而计算机只能用二进制（0或1）表示，二进制转换为科学记数法的公式如下：

$$X = a * 2^e$$

其中， `a` 的值为0或者1，e为小数点移动的位置

举个例子：

27.0转化成二进制为11011.0 ，科学计数法表示为：

$$1.10110 * 2^4$$

前面讲到， `javaScript` 存储方式是双精度浮点数，其长度为8个字节，即64位比特

64位比特又可分为三个部分：

- 符号位S：第 1 位是正负数符号位（sign），0代表正数，1代表负数
- 指数位E：中间的 11 位存储指数（exponent），用来表示次方数，可以为正负数。在双精度浮点数中，指数的固定偏移量为1023
- 尾数位M：最后的 52 位是尾数（mantissa），超出的部分自动进一舍零

如下图所示：

举个例子：

27.5 转换为二进制11011.1

11011.1转换为科学记数法 $1.10111 * 2^4$

符号位为1(正数)，指数位为4+，1023+4，即1027

因为它是十进制的需要转换为二进制，即 `10000000011`，小数部分为 `10111`，补够52位即： 1011 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000`

所以27.5存储为计算机的二进制标准形式（符号位+指数位+小数部分（阶数）），既下面所示

0+10000000011+011 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000`

34.3. 问题分析

再回到问题上

```JavaScript
0.1 + 0.2 === 0.3 // false
```

通过上面的学习，我们知道，在 `javascript` 语言中，0.1 和 0.2 都转化成二进制后再进行运算

```JavaScript
// 0.1 和 0.2 都转化成二进制后再进行运算
0.0001100110011001100110011001100110011001100110011001100110011010 +
0.0011001100110011001100110011001100110011001100110011001100110011010 =
0.0100110011001100110011001100110011001100110011001100110011001100111

// 转成十进制正好是 0.30000000000000004
```

所以输出 `false`

再来一个问题，那么为什么 `x=0.1` 得到 `0.1`？

主要是存储二进制时小数点的偏移量最大为52位，最多可以表达的位数是 `2^53=9007199254740992`，对应科学计数尾数是 `9.007199254740992`，这也是 JS 最多能表示的精度

它的长度是 16，所以可以使用 `toPrecision(16)` 来做精度运算，超过的精度会自动做凑整处理

```JavaScript
.1000000000000000555.toPrecision(16)
// 返回 0.1000000000000000，去掉末尾的零后正好为 0.1
```

但看到的 `0.1` 实际上并不是 `0.1`。不信你可用更高的精度试试：

```javascript
0.1.toPrecision(21) = 0.100000000000000005551
```

如果整数大于 `9007199254740992` 会出现什么情况呢?

由于指数位最大值是1023，所以最大可以表示的整数是 `2^1024 - 1`，这就是能表示的最大整数。但你并不能这样计算这个数字，因为从 `2^1024` 开始就变成了 `Infinity`

```
> Math.pow(2, 1023)
8.98846567431158e+307

> Math.pow(2, 1024)
Infinity
```

那么对于 `(2^53, 2^63)` 之间的数会出现什么情况呢?

- `(2^53, 2^54)` 之间的数会两个选一个，只能精确表示偶数
- `(2^54, 2^55)` 之间的数会四个选一个，只能精确表示4个倍数
- ... 依次跳过更多2的倍数

要想解决大数的问题你可以引用第三方库 `bignumber.js`，原理是把所有数字当作字符串，重新实现了计算逻辑，缺点是性能比原生差很多

34.3.1. 小结

计算机存储双精度浮点数需要先把十进制数转换为二进制的科学记数法的形式，然后计算机以自己的规则{符号位+(指数位+指数偏移量的二进制)+小数部分}存储二进制的科学记数法

因为存储时有位数限制（64位），并且某些十进制的浮点数在转换为二进制数时会出现无限循环，会造成二进制的舍入操作(0舍1入)，当再转换为十进制时就造成了计算误差

34.4. 解决方案

理论上用有限的空间来存储无限的小数是不可能保证精确的，但我们可以处理一下得到我们期望的结果

当你拿到 `1.4000000000000001` 这样的数据要展示时，建议使用 `toPrecision` 凑整并 `parseFloat` 转成数字后再显示，如下：

```
parseFloat(1.4000000000000001.toPrecision(12)) === 1.4  // True
```

封装成方法就是：

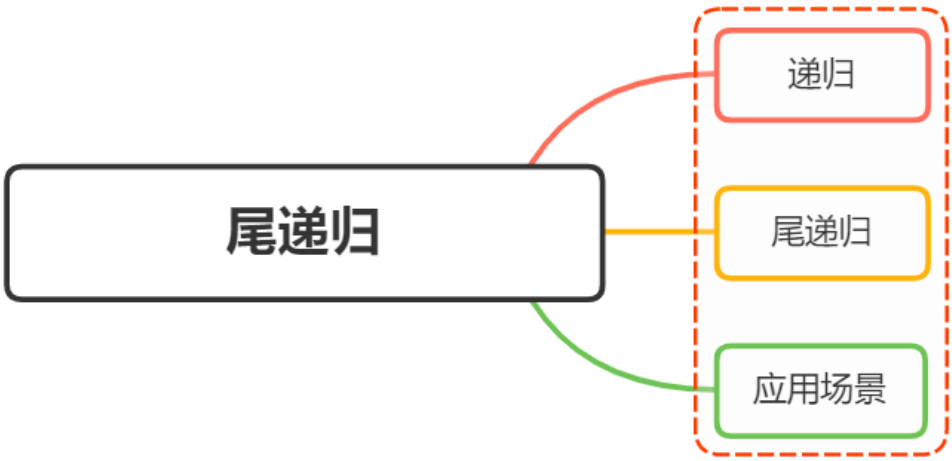
```javascript
function strip(num, precision = 12) {
  return +parseFloat(num.toPrecision(precision));
}
```

对于运算类操作，如 `+-*/` ，就不能使用 `toPrecision` 了。正确的做法是把小数转成整数后再运算。以加法为例：

```javascript
/**
 * 精确加法
 */
function add(num1, num2) {
  const num1Digits = (num1.toString().split('.')[1] || '').length;
  const num2Digits = (num2.toString().split('.')[1] || '').length;
  const baseNum = Math.pow(10, Math.max(num1Digits, num2Digits));
  return (num1 * baseNum + num2 * baseNum) / baseNum;
}
```

最后还可以使用第三方库，如 `Math.js` 、 `BigDecimal.js`

35. 举例说明你对尾递归的理解，有哪些应用场景



35.1. 递归

递归（英语：Recursion）

在数学与计算机科学中，是指在函数的定义中使用函数自身的方法

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数

其核心思想是把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解

一般来说，递归需要有边界条件、递归前进阶段和递归返回阶段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回

下面实现一个函数 `pow(x, n)`，它可以计算 `x` 的 `n` 次方

使用迭代的方式，如下：

JavaScript | 复制代码

```javascript
function pow(x, n) {
  let result = 1;

  // 再循环中，用 x 乘以 result n 次
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

使用递归的方式，如下：

JavaScript | 复制代码

```javascript
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

`pow(x, n)` 被调用时，执行分为两个分支：