

JS 进阶知识点及常考面试题

在这一章节中，我们将会学习到一些原理相关的知识，不会解释涉及到的知识点的作用及用法，如果大家对于这些内容还不怎么熟悉，推荐先去学习相关的知识点内容再来学习原理知识。

手写 call、apply 及 bind 函数

涉及面试题：call、apply 及 bind 函数内部实现是怎么样的？

首先从以下几点来考虑如何实现这几个函数

- 不传入第一个参数，那么上下文默认为 window
- 改变了 this 指向，让新的对象可以执行该函数，并能接受参数

那么我们先来实现 call

```
Function.prototype.myCall = function(context) {  
  if (typeof this !== 'function') {  
    throw new TypeError('Error')  
  }  
  context = context || window  
  context.fn = this  
  const args = [...arguments].slice(1)  
  const result = context.fn(...args)  
  delete context.fn  
  return result  
}
```

以下是对实现的分析：

- 首先 context 为可选参数，如果不传的话默认上下文为 window
- 接下来给 context 创建一个 fn 属性，并将值设置为需要调用的函数
- 因为 call 可以传入多个参数作为调用函数的参数，所以需要将参数剥离出来
- 然后调用函数并将对象上的函数删除

以上就是实现 call 的思路，apply 的实现也类似，区别在于对参数的处理，所以就不一一分析思路了

```
Function.prototype.myApply = function(context) {  
  if (typeof this !== 'function') {  
    throw new TypeError('Error')  
  }  
  context = context || window  
  context.fn = this  
  let result  
  // 处理参数和 call 有区别  
  if (arguments[1]) {  
    result = context.fn(...arguments[1])  
  } else {  
    result = context.fn()  
  }  
  delete context.fn  
  return result  
}
```

bind 的实现对比其他两个函数略微地复杂了一点，因为 bind 需要返回一个函数，需要判断一些边界问题，以下是 bind 的实现

```

Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  const _this = this
  const args = [...arguments].slice(1)
  // 返回一个函数
  return function F() {
    // 因为返回了一个函数，我们可以 new F(), 所以需要判
    断
    if (this instanceof F) {
      return new _this(...args, ...arguments)
    }
    return _this.apply(context,
args.concat(...arguments))
  }
}

```

以下是对实现的分析：

- 前几步和之前的实现差不多，就不赘述了
- bind 返回了一个函数，对于函数来说有两种方式调用，一种是直接调用，一种是通过 new 的方式，我们先来说直接调用的方式
- 对于直接调用来说，这里选择了 apply 的方式实现，但是对于参数需要注意以下情况：因为 bind 可以实现类似这样的代码 `f.bind(obj, 1)(2)`，所以我们需要将两边的参数拼接起来，于是就有了这样的实现 `args.concat(...arguments)`
- 最后来说通过 new 的方式，在之前的章节中我们学习过如何判断 this，对于 new 的情况来说，不会被任何方式改变 this，所以对于这种情况我们需要忽略传入的 this

new

涉及面试题：new 的原理是什么？通过 new 的方式创建对象和通过字面量创建有什么区别？

在调用 new 的过程中会发生以上四件事情：

1. 新生成了一个对象
2. 链接到原型
3. 绑定 this
4. 返回新对象

根据以上几个过程，我们也可以试着来自己实现一个 new

```
function create() {  
  let obj = {}  
  let Con = [].shift.call(arguments)  
  obj.__proto__ = Con.prototype  
  let result = Con.apply(obj, arguments)  
  return result instanceof Object ? result : obj  
}
```

以下是对实现的分析：

- 创建一个空对象
- 获取构造函数
- 设置空对象的原型
- 绑定 this 并执行构造函数
- 确保返回值为对象

对于对象来说，其实都是通过 new 产生的，无论是 function Foo() 还是 let a = { b : 1 }。

对于创建一个对象来说，更推荐使用字面量的方式创建对象（无论性能上还是可读性）。因为你使用 new Object() 的方式创建对象需要通过作用域链一层层找到 Object，但是你使用字面量的方式就没这个问题。

```
function Foo() {}  
// function 就是个语法糖  
// 内部等同于 new Function()  
let a = { b: 1 }  
// 这个字面量内部也是使用了 new Object()
```

instanceof 的原理

涉及面试题：instanceof 的原理是什么？

instanceof 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype。

我们也可以试着实现一下 instanceof

```
function myInstanceof(left, right) {  
  let prototype = right.prototype  
  left = left.__proto__  
  while (true) {  
    if (left === null || left === undefined)  
      return false  
    if (prototype === left)  
      return true  
    left = left.__proto__  
  }  
}
```

以下是对实现的分析：

- 首先获取类型的原型
- 然后获得对象的原型
- 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 null，因为原型链最终为 null

为什么 $0.1 + 0.2 \neq 0.3$

涉及面试题：为什么 $0.1 + 0.2 \neq 0.3$ ？如何解决这个问题？

先说原因，因为 JS 采用 IEEE 754 双精度版本（64位），并且只要采用 IEEE 754 的语言都有该问题。

我们都知道计算机是通过二进制来存储东西的，那么 0.1 在二进制中会表示为

```
// (0011) 表示循环  
0.1 = 2-4 * 1.10011(0011)
```

我们可以发现， 0.1 在二进制中是无限循环的一些数字，其实不只是 0.1 ，其实很多十进制小数用二进制表示都是无限循环的。这样其实没什么问题，但是 JS 采用的浮点数标准却会裁剪掉我们的数字。

IEEE 754 双精度版本（64位）将 64 位分为了三段

- 第一位用来表示符号
- 接下去的 11 位用来表示指数
- 其他的位数用来表示有效位，也就是用二进制表示 0.1 中的 10011(0011)

那么这些循环的数字被裁剪了，就会出现精度丢失的问题，也就造成了 0.1 不再是 0.1 了，而是变成了 0.100000000000000002

```
0.100000000000000002 === 0.1 // true
```

那么同样的， 0.2 在二进制也是无限循环的，被裁剪后也失去了精度变成了 0.200000000000000002

```
0.200000000000000002 === 0.2 // true
```

所以这两者相加不等于 0.3 而是 0.3000000000000000004

```
0.1 + 0.2 === 0.3000000000000000004 // true
```

那么可能你又会会有一个疑问，既然 0.1 不是 0.1，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制被转换为了十进制，十进制又被转换为了字符串，在这个转换的过程中发生了取近似值的过程，所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
console.log(0.1000000000000000002) // 0.1
```

那么说完了为什么，最后来说说怎么解决这个问题吧。其实解决的办法有很多，这里我们选用原生提供的方式来最简单的解决问题

```
parseFloat((0.1 + 0.2).toFixed(10)) === 0.3 // true
```

垃圾回收机制

涉及面试题：V8 下的垃圾回收机制是怎么样的？

V8 实现了准确式 GC，GC 算法采用了分代式垃圾回收机制。因此，V8 将内存（堆）分为新生代和老生代两部分。

新生代算法

新生代中的对象一般存活时间较短，使用 Scavenge GC 算法。

在新生代空间中，内存空间分为两部分，分别为 From 空间和 To 空间。在这两个空间中，必定有一个空间是使用的，另一个空间是空闲的。新分配的对象会被放入 From 空间中，当 From 空间被占满

时，新生代 GC 就会启动了。算法会检查 From 空间中存活的对象并复制到 To 空间中，如果有失活的对象就会销毁。当复制完成后将 From 空间和 To 空间互换，这样 GC 就结束了。

老生代算法

老生代中的对象一般存活时间较长且数量也多，使用了两个算法，分别是标记清除算法和标记压缩算法。

在讲算法前，先来说下什么情况下对象会出现在老生代空间中：

- 新生代中的对象是否已经经历过一次 Scavenge 算法，如果经历过的话，会将对象从新生代空间移到老生代空间中。
- To 空间的对象占比大小超过 25 %。在这种情况下，为了不影响到内存分配，会将对象从新生代空间移到老生代空间中。

老生代中的空间很复杂，有如下几个空间


```
enum AllocationSpace {
    // TODO(v8:7464): Actually map this space's
    memory as read-only.
    RO_SPACE,      // 不变的对象空间
    NEW_SPACE,     // 新生代用于 GC 复制算法的空间
    OLD_SPACE,     // 老生代常驻对象空间
    CODE_SPACE,    // 老生代代码对象空间
    MAP_SPACE,     // 老生代 map 对象
    LO_SPACE,      // 老生代大空间对象
    NEW_LO_SPACE,  // 新生代大空间对象

    FIRST_SPACE = RO_SPACE,
    LAST_SPACE = NEW_LO_SPACE,
    FIRST_GROWABLE_PAGED_SPACE = OLD_SPACE,
    LAST_GROWABLE_PAGED_SPACE = MAP_SPACE
};
```

在老生代中，以下情况会先启动标记清除算法：

- 某一个空间没有分块的时候
- 空间中被对象超过一定限制
- 空间不能保证新生代中的对象移动到老生代中

在这个阶段中，会遍历堆中所有的对象，然后标记活的对象，在标记完成后，销毁所有没有被标记的对象。在标记大型对内存时，可能需要几百毫秒才能完成一次标记。这就会导致一些性能上的问题。为了解决这个问题，2011 年，V8 从 stop-the-world 标记切换到增量标志。在增量标记期间，GC 将标记工作分解为更小的模块，可以让 JS 应用逻辑在模块间隙执行一会，从而不至于让应用出现停顿情况。但在 2018 年，GC 技术又有了一个重大突破，这项技术名为并发标记。该技术可以让 GC 扫描和标记对象时，同时允许 JS 运行，

你可以点击 [该博客](#)

(<https://v8project.blogspot.com/2018/06/concurrent-marking.html>) 详细阅读。

清除对象后会造成堆内存出现碎片的情况，当碎片超过一定限制后会启动压缩算法。在压缩过程中，将活的对象像一端移动，直到所有对象都移动完成然后清理掉不需要的内存。

小结

以上就是 JS 进阶知识点的内容了，这部分的知识相比于之前的内容更加深入也更加的理论，也是在面试中能够于别的候选者拉开差距的一块内容。如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。