- 从四个维度压榨性能
- 手写 Promise
- 手写 Promise
- 手写 Promise
- 手写 Promise
- 从 JS 的知识点及常考题试题
- 从 JS 的知识点及常考题试题
- JS 异步编
- DeClutter Tips
- 前端面试的总结及考题试题
- 前端面试的总结及考题试题
- Event Loop
- 前端面试题整理
- 前端面试题整理
- 安全防范知识点

JS 的知识点及常考题试题

在这一章节中，我打算给你们一些即将面试的知识点，不会精深到这些知识点的内容和用法，但是大家对于这些内容还是不会生疏的，毫无头绪不清楚的知识点的话我来多了这解释说。

更新日志

- 修改了有关 call、apply 及 bind 函数。

手写 call、apply 及 bind 函数

首先由以下几点来考虑如何实现这几个函数

- 不改入第一个参数，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，并赋予参数

那么我们先来实现 `call`：

```
Function.prototype.myCall = function(context) {
  if (typeof context === 'undefined' || typeof context === 'null') {
    context = window
  }
  const newArgs = [...arguments].slice(1)
  context.fn = this
  const result = context.fn(...newArgs)
  delete context.fn
  return result
}
```

以下是实现思路：

- 首先 `context` 为空数据，如果不传的话默认上下文为 `window`。
- 接下来给对象 `context` 加 `fn` `context` 加新一个属性，并将函数执行改变调用的函数
- 因为 `call` 可以传入多个参数作为函数的参数，所以需要参数拿到第二个
- 然后删除函数所有的属性上的函数调用

以上就是实现 `call` 的思路，`apply` 的实现也大致相同，区别在于参数的处理，所以那天一一分析直接了

```
Function.prototype.myApply = function(context) {
  if (typeof context === 'undefined' || typeof context === 'null') {
    context = window
  }
  const newArgs = [...arguments][1]
  context.fn = this
  if (newArgs[0]) {
    // 这里是解构 arr 的语法
    if (arguments[1].length === 1) {
      const result = context.fn(...arguments[1])
      delete context.fn
      return result
    } else {
      const result = context.fn(...arguments[1])
      delete context.fn
      return result
    }
  }
}
```

`bind` 的实现方式也有两个函数调用情况下一点，因为 `bind` 需要返回一个函数，需要判断一下是这样的函数，以下是 `bind` 的实现

```
Function.prototype.myBind = function(context) {
  if (typeof context === 'undefined' || typeof context === 'null') {
    context = window
  }
  const newArgs = [...arguments].slice(1)
  const _thisFn = this
  return function F() {
    // 处理函数使用 new 的情况，具体请看MDN
    if (this instanceof F) {
      return new _thisFn(...newArgs, ...arguments)
    }
    // 绑定 context 产生闭包环境
    return _thisFn.apply(context, newArgs.concat(...arguments))
  }
}
```

以下是实现思路分析：

- 最开始判定要传的参数为多，就不要了
- `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，那两种情况最重要的就是判定
- 对于直接调用来说，这里面有 `context` 的方式实现，但是对于使用原型链上了解的，因为使用 `new` 的话对于 `this` 的指向会改变，`this` 的指向会变成新的对象，`this` 的指向不再是 `context` ，所以需要单独判断这种情况
- 最后再通过 `apply` 的方式，在这种情况下传进去的 `context` 的指向，`apply` 的第一参数 `this` 的指向来说，不会被改变的方式实现 `this` ，所以对于这种情况的调用函数传入的 `this`

常见考点

- 以上三个函数的手写题，这几个手写题解决的问题了

new

在调用 `new` 的过程中会发生以上四种事情：

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，我们先来实现最基本的正常用一个 `new` ：

```
function create() {
  // 创建一个空的对象
  let obj = new Object()
  // 得到传入的第一个参数，这是构造函数
  let Constr = [].shift.call(arguments)
  // 链接到原型
  obj.__proto__ = Constr.prototype
  // 绑定 this 执行构造函数
  var result = Constr.apply(obj, arguments)
  // 确保返回的是对象
  return typeof result === 'object' ? result : obj
}
```

以下是对实现的分析：

- 创建一个空对象
- 获取传入参数
- 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 = { }` ，这个 `Foo` 和 `a` 对象也是通过 `new` 产生的，这个对象的原型都是 `Object.prototype` ，而新的原型对象也有自己的原型链，一直到 `null` 。

```
function Foo() {
  // function 默认不创建
  // 一个新的隐式对象
  let a = { 1: 1 }
  // 这个隐式对象的原型是 new Object()
}
```

更多关于 `new` 的内容可以阅读我的文章 **探索 new 操作符**。

常见考点

- `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？

instanceof 的原理

`instanceof` 可以正确的判断引用数据类型，因为内部机制是通过判断对象的原型链中是不是能找到该类型的 `prototype` 。

我们可以试试看写一个 `instanceof` 。

```
function myInstanceof(left, right) {
  // 这里先用typeof来判断基本类型
  if (typeof left === 'object' || typeof left === 'function') {
    // 这里是防止传入了 null
    let proto = left.__proto__
    while (true) {
      // 到 Object 上还是为 null 则不再 instanceof
      if (proto == null) return false
      // 找到相同的原型则返回 true
      if (proto == right.prototype) return true
      proto = proto.__proto__
    }
  }
  // 如果是基本数据类型就直接返回 false
  return false
}
```

以下是对实现的分析：

- 首先实现类型的判定
- 然后获得对象的原型
- 然后一直循环对象的原型直至等于原型的原型，直到对象原型为 `null` ，因为原型链最终为 `null`

常见考点

- 手写题

为什么 0.1 + 0.2 != 0.3

先说原因，因为 JS 采用 IEEE 754 双精度版本（64 位），并且计算采用 IEEE 754 的浮点数来带来的问题。

我们都知道计算机是通过二进制来存储信息的，例如，`0.1` 在二进制中是表示为：

```
// 0011 表示循环
0.1 = 2^-4 x 1.9999999999999999
```

我们可以发现，`0.1` 在二进制中是无限循环的，但是字节，因此不可能 `0.1` ，因此我们会看到 `0.1` 的值是最接近 `0.1` 的数字。

IEEE 754 双精度版本（64位）将 64 位分为了三部：

- 第一位用来表示符号
- 接下来的 11 位用来表示指数
- 剩余的位数用来表示有效位，也就是表示二进制数的 `1.x` 的部 `1.9999999999999999`

那么这部分的数字有限是？，那是说明精度是有内容的，这是说是？ `0.1` 产生的是 `0.1` 了，但是 `0.1` 了，但是 `0.1000000000000000` 。

```
0.1000000000000000 === 0.1 // true
```

那么同理呢，`0.2` 是二进制也是无限循环的，能能用近似先来了解是它是？

```
0.2000000000000000 === 0.2 // true
```

所以这两者相加等于等于 `0.3` 而是 `0.3000000000000004` 。

```
0.1 + 0.2 === 0.3000000000000004 // true
```

那么可能你又会问一个疑问，既然 `0.1` 不是 `0.1` ，那为什么 `console.log(0.1)` 就是正确的呢？

因为在输入内容的时候，二进制编译两为了了的数，于是我们直接用了字符串，是这个精度的已经等于是它的是的过程，所以它可以的表示是一个正确的，所是可以是这过了它的表现是：

```
console.log(0.1000000000000000) // 0.1
```

那在说完了为什么我们不直接用字符串会出现有些尴尬的时？

- 首先字符的数量比较的过程一次 String 要长，数量的过程比较的话，会增加解析的时间
- 为 字符的数量比较大小最多 16 为，在这种情况下，为了不影响解析内容时间，会导致数量会比较的数量也可以的时。

使用字符的的原因是，每位了为了它的

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

常见考点

- 为什么 0.1 + 0.2 != 0.3，你知解决这个问题？

垃圾回收机制

V8 实现了 清除式 GC，GC 要实现了分代式垃圾回收机制，因此，V8 将内存（堆）划为新生代和老生代两部分。

新生代算法

新生代的对象一般存放时间短，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。新的分配的对象都先放入 From 空间中。当 From 空间被占满时，新生代 GC 就会启动了，算法会把 From 空间中还存活的对象拷贝到 To 空间中，而清除掉无用的对象垃圾回收，完整的垃圾回收一次后，From 空间和 To 空间交换，这样 GC 就会继续工作了。

老生代算法

老生代的对象一般存放时间长且数量也大，使用了两个算法，分别是标记清除算法和标记整理算法。

在清除算法，先从根节点开始遍历对象并标记活跃的对象。

- 新生代的对象是否可以通过根节点一次 Scavenge 算法，新生代的活跃的，没有所属以新生代，但所有的活跃的对象。
- 为 活跃的对象占比大小超过 25 %，在这种情况下，为了不影响内存分配，会将对象从新生代移动到老生代空间中。

老生代的内存空间算法，每个分为三个区域：

```
new ArrayConstructor()
// 指针区域，包含大多数可能是 GC 后仍活着的对象 (in old pointer space).
  MS_PAGES,     // 代码对象区域
  MAP_PAGES,    // 简单的 JS 对象，固定数组的区域
  OLD_POINTER,  // 老对象指针指向的区域
  OLD_DATA,     // 老对象内容只包含数据的区域
  LO_PAGES,     // 大对象区域，其他区域都是这个区域的倍数
  CODE_PAGES,   // 代码区域，可执行的
  MAP_SPACE,    // 老对象内容的区域

  FIRST_PAGE = MS_PAGES,
  LAST_PAGE = MS_PAGES,
  FIRST_SPACE = MS_PAGES,
  LAST_SPACE = MS_PAGES + NUM_OLD_SPACES,
  FIRST_GROWABLE_PAGED_SPACE = MAP_SPACE
]
```

在清除算法，以下情况会先进行标记清除算法：

- 新一个空间没有分配的区域
- 空间中的碎片超过一定限制
- 空间中被标记的对象的内存容量超过整个大小

在这个情况下，会触发整个堆内存的整理，而会标记活跃的，但标记完成后，碎片内存会被移动，以防内存碎，这样以方便后内存的，可能整理过程会导致 JS 暂停执行，这就会导致一卡顿的现象。为了解决这个问题，2011 年，V8 在 chrome 的 Ivy 项目实现了增量标记，在增量标记阶段，GC 将标记过程分解成多个步骤，可以让 JS 应用程序能继续执行的一段，这样不会因为长时间中断，而卡顿 300 多 ms，这样即使中断了一段时间，这段时间内不会卡顿。这使用户可以以 60 帧的速度运行动画，同时也不会卡顿。这个算法，也称为 三色标记 算法。

清除内存最后会是通过内存碎片的整理，为解决碎片，这里会通过标记的过程，在活跃过程中的，将活跃的对象一移除的，直到内存有连续的内存空间可用。

常见考点

- V8 下的垃圾回收算法是怎么样的？

小结

以上就是 JS 的那些知识的内容了，这部分的知识我们了之前的内容要加深入的更深的理解，但是这是这些内容是不是的的是最是的不是是的一的的，如果大家的不是了了解了的的的的的的，的这的的的是是是。

回复

[illegible]

[illegible]

[illegible]