JS

1. 闭包和作用域

> 闭包是作用域应用的特殊场景。 js中常见的作用域包括全局作用域、函数作用域、块级作用域。要知道 **Js中自由变量的查找是在函数定义的地方，向上级作用域查找，不是在执行的地方**。 常见的闭包使用有两种场景：一种是函数作为参数被传递；一种是函数作为返回值被返回。

2. 函数中的arguments

- 只有函数才有arguments
- 每个函数都有一个内置好了的argument，不用手动去创建
- arguments具有length属性
- arguments按索引方式存储数据
- arguments不具有数组的push/pop等操作

3. 一秒理解call，apply，bind

作用：重定义this指向

1. bind返回一个函数方法

```
var name='小王',age=17;
var obj={
  name:'小张'
  objAge:this.age,
  myFun:function(){
    console.log(this.name+"年龄"+this.age);
  }
}
var db={
  name:'德玛',
  age:99,
}

obj.objAge;  // 17
obj.myFun()  // 小张年龄 undefined
obj.myFun.call();        // 指向windows
obj.myFun.call(db);        // 德玛年龄 99
obj.myFun.apply(db);       // 德玛年龄 99
obj.myFun.bind(db)();      // 德玛年龄 99
// bind 返回的是一个新的函数，你必须调用它才会被执行
```

2. `call` 、 `bind` 、 `apply` 这三个函数的第一个参数都是 `this` 的指向对象，第二个参数差别就来了：

1. `call` 的参数是直接放进去的，第二第三第 n 个参数全都用逗号分隔，直接放到后面 `obj.myFun.call(db,'成都', ... ,'string' )` 。
2. `apply` 的所有参数都必须放在一个数组里面传进去 `obj.myFun.apply(db,['成都', ..., 'string' ])` 。
3. `bind` 除了返回是函数以外，它的参数和 `call` 一样。
4. 三者的参数不限定是 string 类型，允许是各种类型，包括函数 、 object 等等

```
var name='小王',age=17;
var obj={
  name:'小张'
  objAge:this.age,
  myFun:function(){
    console.log(this.name+"年龄"+this.age,"来自"+fm+"去往"+t);
  }
}
var db={
  name:'德玛',
  age:99,
}

obj.myFun.call(db,'成都','上海');//德玛年龄99来自成都去往上海
obj.myFun.apply(db,['成都','上海']);//德玛年龄99来自成都去往上海
obj.myFun.bind(db,'成都','上海')();// 德玛年龄99来自成都去往上海
obj.myFun.bind(db,['成都','上海'])();//德玛年龄99来自成都, 上海去往 undefined
```

4. 一秒理解this

> 函数运行时，在函数体内部自动生成的一个对象，只能在函数体内部使用

1. 函数调用
   函数的最通常用法，属于全局性调用，因此this就代表全局对象

```
var x = 1;
function test() {
  console.log(this.x);
}
test();  // 1
```

2. 作为对象方法的调用
   作为某个对象的方法调用，这时this就指这个 上级对象 。

```
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;

obj.m(); // 1
```

3. 作为构造函数调用

   通过这个函数，可以生成一个新对象。这时，this就指这个新对象

```
function test() {
  this.x = 1;
}

var obj = new test();
obj.x // 1
```

4. apply 调用

   apply()是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时this指的就是这第一个参数。

```
var x = 0;
function test() {
  console.log(this.x);
}

var obj = {};
obj.x = 1;
obj.m = test;
obj.m.apply() // 0
obj.m.apply(obj); //1
```

apply()的参数为空时，默认调用全局对象。因此，这时的运行结果为0，证明this指的是全局对象。

5. WeakMap

> - 原生的 WeakMap 持有的是每个键对象的“弱引用”，这意味着在没有其他引用存在时垃圾回收能正确进行。原生 WeakMap 的结构是特殊且有效的，其用于映射的 `key` _只有_在其没有被回收时才是有效的。
> - 由于这样的弱引用， `WeakMap` 的 `key` 是**不可枚举的**(没有方法能给出所有的 `key`)。如果 `key` 是可枚举的话，其列表将会受垃圾回收机制的影响，从而得到不确定的结果。因此，如

> 果你想要这种类型对象的 key 值的列表，你应该使用 Map

TS

1. type和interface的区别

> interface可以重复声明，type不行，继承方式不一样，type使用交叉类型方式，interface使用extends实现。在对象扩展的情况下，使用接口继承要比交叉类型的性能更好。建议使用interface来描述对象对外暴露的借口，使用type将一组类型重命名（或对类型进行复杂编程）

```
interface iMan {
  name: string;
  age: number;
}
// 接口可以进行声明合并
interface iMan {
  hobby: string;
}

type tMan = {
  name: string;
  age: number;
};
// type不能重复定义
// type tMan = {}

// 继承方式不同,接口继承使用extends
interface iManPlus extends iMan {
  height: string;
}
// type继承使用&，又称交叉类型
type tManPlus = { height: string } & tMan;

const aMan: iManPlus = {
  name: "aa",
  age: 15,
  height: "175cm",
  hobby: "eat",
};

const bMan: tManPlus = {
  name: "bb",
  age: 15,
  height: "150cm",
};
```

1.