1.ES6 都有什么 Iterator 遍历器

答案：Set、Map

1、遍历器（Iterator）是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）

2、Iterator 的作用有三个：

- 一是为各种数据结构，提供一个统一的、简便的访问接口；
- 二是使得数据结构的成员能够按某种次序排列；
- 三是 ES6 创造了一种新的遍历命令 for...of 循环，Iterator 接口主要供 for...of 消费。

3、默认部署了 Iterator 的数据有 Array、Map、Set、String、TypedArray、arguments、NodeList 对象，ES6 中有的是 Set、Map、

2. ES6 中类的定义

```
// 1、类的基本定义
class Parent {
  constructor(name = "小白") {
    this.name = name;
  }
}
// 2、生成一个实例
let g_parent = new Parent();
console.log(g_parent); //{name: "小白"}
let v_parent = new Parent("v"); // 'v'就是构造函数 name 属性 , 覆盖构造函数的 name 属性值
console.log(v_parent); // {name: "v"}
// 3、继承
class Parent {
  //定义一个类
  constructor(name = "小白") {
    this.name = name;
  }
}

class Child extends Parent {}

console.log("继承", new Child()); // 继承 {name: "小白"}
// 4、继承传递参数
class Parent {
  //定义一个类
  constructor(name = "小白") {
    this.name = name;
  }
}

class Child extends Parent {
  constructor(name = "child") {
    // 子类重写 name 属性值
    super(name); // 子类向父类修改 super 一定放第一行
    this.type = "preson";
  }
}
console.log("继承", new Child("hello")); // 带参数覆盖默认值  继承{name: "hello", type: "preson"}
// 5、ES6 重新定义的 ES5 中的访问器属性
class Parent {
  //定义一个类
  constructor(name = "小白") {
    this.name = name;
  }
```

```
  get longName() {
    // 属性
    return "mk" + this.name;
  }

  set longName(value) {
    this.name = value;
  }
}

let v = new Parent();
console.log("getter", v.longName); // getter mk 小白

v.longName = "hello";
console.log("setter", v.longName); // setter mkhello
// 6、类的静态方法
class Parent {
  //定义一个类
  constructor(name = "小白") {
    this.name = name;
  }

  static tell() {
    // 静态方法:通过类去调用，而不是实例
    console.log("tell");
  }
}

Parent.tell(); // tell
// 7、类的静态属性：

class Parent {
  //定义一个类
  constructor(name = "小白") {
    this.name = name;
  }

  static tell() {
    // 静态方法:通过类去调用，而不是实例
    console.log("tell"); // tell
  }
}

Parent.type = "test"; // 定义静态属性

console.log("静态属性", Parent.type); // 静态属性 test

let v_parent = new Parent();
console.log(v_parent); // {name: "小白"}  没有 tell 方法和 type 属性
```

3.谈谈你对 ES6 的理解

答案：es6 是一个新的标准，它包含了许多新的语言特性和库，是 JS 最实质性的一次升级。 比如'箭头函数'、'字符串模板'、'generators(生成器)'、'async/await'、'解构赋值'、'class'等等，还有就是引入 module 模块的概念。

箭头函数可以让 this 指向固定化，这种特性很有利于封装回调函数

- （1）函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象。
- （2）不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。

- （3）不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 Rest 参数代替。
- （4）不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。
- async/await 是写异步代码的新方式，以前的方法有回调函数和 Promise。
- async/await 是基于 Promise 实现的，它不能用于普通的回调函数。async/await 与 Promise 一样，是非阻塞的。
- async/await 使得异步代码看起来像同步代码，这正是它的魔力所在。

4.说说你对 promise 的了解

答案：Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件监听——更合理和更强大。所谓 Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

Promise 对象有以下两个特点:

1. 对象的状态不受外界影响，Promise 对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（已失败）
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。

5.解构赋值及其原理

解构赋值：其实就是分解出一个对象的解构，分成两个步骤：

1. 变量的声明
2. 变量的赋值

原理：ES6 变量的解构赋值本质上是“模式匹配”,只要等号两边的模式相同，左边的变量就会被赋予匹配的右边的值，如果匹配不成功变量的值就等于 undefined

解析：

一、 数组的解构赋值

```
// 对于数组的解构赋值，其实就是获得数组的元素，而我们一般情况下获取数组元素的方法是通过下标获取，例如：
let arr = [1, 2, 3];
let a = arr[0];
let b = arr[1];
let c = arr[2];

// 而数组的解构赋值给我们提供了极其方便的获取方式，如下：
let [a, b, c] = [1, 2, 3];
console.log(a, b, c); //1,2,3
```

模式匹配解构赋值

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
console.log(foo, bar, baz); //1,2,3
```

省略解构赋值

```
let [, , a, , b] = [1, 2, 3, 4, 5];
console.log(a, b); //3,5
```

含剩余参数的解构赋值

```
let [a, ...reset] = [1, 2, 3, 4, 5];
console.log(a, reset); //1,[2,3,4,5]
```

其转成 ES5 的原理如下：

```
var a = 1,
  reset = [2, 3, 4, 5];
console.log(a, reset); //1,[2,3,4,5]
```

注意：如果剩余参数是对应的值为 undefined，则赋值为[]，因为找不到对应值的时候，是通过 slice 截取的，如下：

```
let [a, ...reset] = [1];
console.log(a, reset); //1,[]
```

其转成 ES5 的原理如下：

```
var _ref = [1],
  a = _ref[0],
  reset = _ref.slice(1);
console.log(a, reset); //1,[]
```

非数组解构成数组(重点，难点)

一条原则：要解构成数组的前提：如果等号右边，不是数组(严格地说，不是可遍历的解构)，则直接报错，例如：

```
let [foo] = 1; //报错
let [foo1] = false; //报错
let [foo2] = NaN; //报错
let [foo3] = undefined; //报错
let [foo4] = null; //报错
let [foo5] = {}; //报错
```

为什么？转成 ES5 看下原理就一清二楚了：

```
var _ = 1,
  foo = _[0]; //报错
var _false = false,
  foo1 = _false[0]; //报错
var _NaN = NaN,
  foo2 = _NaN[0]; //报错
var _undefined = undefined,
  foo3 = _undefined[0]; //报错
var _ref = null;
foo4 = _ref[0]; //报错
var _ref2 = {},
  foo5 = _ref2[0]; //报错
```

Set 的解构赋值

先执行 new Set()去重，然后对得到的结果进行解构

```
let [a, b, c] = new Set([1, 2, 2, 3]);
console.log(a, b, c); //1,2,3
```

迭代器解构

```
function* fibs() {
  let a = 0;
  let b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

let [first, second, third, fourth, fifth, sixth] = fibs();
sixth; // 5
```

总结 1：只要某种数据结构具有 Iterator 接口，都可以采用数组形式的解构赋值。

7. 解构赋值的默认值

当变量严格等于 undefined 的时候，会读取默认值，所谓的严格等于，就是“===”

```
----------
let [a,b = 'default'] = [1];
console.log(a,b);//1,'default'
----------
let [c = 'default'] = [undefined];
console.log(c);//'default'
----------
function f() {
  console.log('aaa');
}
let [x = f()] = [1];
console.log(x);//1
----------
function f() {
  console.log('aaa');//'aaa'
}
```

```
let [a,x = f()] = [1];
console.log(a,x);//1,undefined
```
总结 2：如果不使用默认值，则不会执行默认值的函数

二、对象的解构赋值

1. 解构赋值的举例：

```
let p1 = {
  name: "zhuangzhuang",
  age: 25
};
let { name, age } = p1; //注意变量必须为属性名
console.log(name, age); //"zhuangzhuang",25
```
其转成 es5 的原理则为：
```
var _p1 = p1,
  name = _p1.name,
  age = _p1.age;
console.log(name, age); //"zhuangzhuang",25
```
2. 解构赋值的别名

如果使用别名，则不允许再使用原有的解构出来的属性名，看以下举例则会明白：
```
let p1 = {
  name: "zhuangzhuang",
  age: 25
};
let { name: aliasName, age: aliasAge } = p1; //注意变量必须为属性名
console.log(aliasName, aliasAge); //"zhuangzhuang",25
console.log(name, age); //Uncaught ReferenceError: age is not defined
```
为何打印原有的属性名则会报错？让我们看看转成 es5 后的原理是如何实现的：
```
var _p1 = p1,
  aliasName = _p1.name,
  aliasAge = _p1.age;
console.log(aliasName, aliasAge); //"zhuangzhuang",25
console.log(name, age); //所以打印 name 和 age 会报错——“Uncaught ReferenceError: age is not defined”，但是为何只报错 age，不报错 name 呢？
```
只报错 age，不报错 name，这说明其实 name 是存在的，那么根据 js 的解析顺序，当在当前作用域 name 无法找到时，会向上找，直到找到 window 下的 name,而我们打印 window 可以发现，其下面确实有一个 name，值为“”，而其下面并没有属性叫做 age，所以在这里 name 不报错，只报 age 的错。类似 name 的属性还有很多，比如 length 等。

3. 解构赋值的默认值

有些情况下，我们解构出来的值并不存在，所以需要设定一个默认值，例如：
```
let obj = {
  name: "zhuangzhuang"
};
let { name, age } = obj;
console.log(name, age); //"zhuangzhuang",undefined
```
我们可以看到当 age 这个属性并不存在于 obj 的时候，解构出来的值为 undefined，那么为了避免这种尴尬的情况，我们常常会设置该属性的默认值，如下：
```
let obj = {
  name: "zhuangzhuang"
};
let { name, age = 18 } = obj;
console.log(name, age); //"zhuangzhuang",18
```
当我们取出来的值不存在，即为 undefined 的时候，则会取默认值(假设存在默认值)，ES6 的默认值是使用**“变量=默认值”**的方式。

注意：只有当为 undefined 的时候才会取默认值，null 等均不会取默认值
```
let obj = {
  name: "zhuangzhuang",
  age: 27,
```

```
  gender: null, //假设未知使用 null
  isFat: false
};
let { name, age = 18, gender = "man", isFat = true, hobbies = "study" } = obj;
console.log(name, age, gender, isFat, hobbies); //"zhuangzhuang", 27, null, false, "study"
```

4. 解构赋值的省略赋值

当我们并不是需要取出所有的值的时候，其实可以省略一些变量，这就是省略赋值，如下

```
let arr = [1, 2, 3];
let [, , c] = arr;
console.log(c); //3
```

注意：省略赋值并不存在与对象解构，因为对象解构，明确了需要的属性

```
let obj = {
  name: "zhuangzhuang",
  age: 27,
  gender: "man"
};
let { age } = obj;
console.log(age); //27
```

5. 解构赋值的嵌套赋值(易错点，重点，难点)

```
let obj = {},
  arr = [];

({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });
console.log(obj, arr); //{prop:123},[true]
```

注意当解构出来是 undefined 的时候，如果再给子对象的属性，则会报错，如下

```
let {
  foo: { bar }
} = { baz: "baz" };
//报错，原因很简单，看下原理即可，如下：
//原理:
let obj = { baz: "baz" };
let foo = obj.foo; //foo 为 undefined
let bar = foo.bar; //undefined 的 bar，可定报错
```

6. {}是块还是对象？

当我们写解构赋值的时候，很容易犯一个错误——{}的作用是块还是对象混淆，举例如下：

```
//举例一：
let {a} = {a:"a"};
console.loh(a);//'a',这个很简单
//很多人觉得，以下这种写法也是可以的：
let a;
{a} = {a:"a"};//直接报错，因为此时 a 已经声明过了，在语法解析的时候，会将这一行的{}看做块结构，而“块=对象”，显然是语法错误，所以正确的做法是不将大括号写在开头，如下：
let a;
({a} = {a:"a"})
```

7. 空解构

按照之前写的，解构赋值，左边则为解构出来的属性名，当然，在这里，我们也可以不写任何属性名称，也不会又任何的语法错误，即便这样没有任何意义，如下：

```
({} = [true, false]);
({} = "abc");
({} = []);
```

8. 解构成对象的原则

如果解构成对象，右侧不是 null 或者 undefined 即可! 之前说过，要解构成数组，右侧必须是可迭代对象，但是如果解构成对象，右侧不是 null 活着 undefined 即可!

三、字符串的解构赋值

字符串也是可以解构赋值的

```
const [a, b, c, d, e] = "hello";
```

```
console.log(a, b, c, d, e); //'h','e','l','l','o'
```
转成 es5 的原理如下:
```
var _hello = "hello",
  a = _hello[0],
  b = _hello[1],
  c = _hello[2];

console.log(a, b, c);
```
注意：字符串有一个属性 length，也可以被解构出来，但是要注意，解构属性一定是对象解构
```
let { length } = "hello";
console.log(length); //5
```
4. 布尔值和数值的解构

布尔值和数值的解构，其实就是对其包装对象的解构，取的是包装对象的属性
```
{toString:s} = 123;
console.log(s);//s === Number.prototype.toString

{toString:s} = true;
console.log(s);//s === Boolean.prototype.toString
```
总结：解构赋值的规则是：

1. 解构成对象，只要等号右边的值不是对象或数组，就先将其转为对象。由于 undefined 和 null 无法转为对象，所以对它们进行解构赋值，都会报错。
2. 解构成数组，等号右边必须为可迭代对象

6.Array.from() 与 Array.reduce()

答案：

Array.from()方法就是将一个类数组对象或者可遍历对象转换成一个真正的数组 Array.reduce()方法对累加器和数组中的每个元素 (从左到右)应用一个函数，将其减少为单个值。

解析：

Array.from()
```
// 那么什么是类数组对象呢？所谓类数组对象，最基本的要求就是具有 length 属性的对象。
// 1、将类数组对象转换为真正数组：
let arrayLike = {
  0: "tom",
  1: "65",
  2: "男",
  3: ["jane", "john", "Mary"],
  length: 4
};
let arr = Array.from(arrayLike);
console.log(arr); // ['tom','65','男',['jane','john','Mary']]
// 那么，如果将上面代码中 length 属性去掉呢？实践证明，答案会是一个长度为 0 的空数组。
// 这里将代码再改一下，就是具有 length 属性，但是对象的属性名不再是数字类型的，而是其他字符串型的，
代码如下：
let arrayLike = {
  name: "tom",
  age: "65",
  sex: "男",
  friends: ["jane", "john", "Mary"],
  length: 4
};
let arr = Array.from(arrayLike);
console.log(arr); // [ undefined, undefined, undefined, undefined ]
// 会发现结果是长度为 4，元素均为 undefined 的数组
// 由此可见，要将一个类数组对象转换为一个真正的数组，必须具备以下条件：
// 1、该类数组对象必须具有 length 属性，用于指定数组的长度。如果没有 length 属性，那么转换后的数组是
一个空数组。
// 2、该类数组对象的属性名必须为数值型或字符串型的数字
```

```
// ps: 该类数组对象的属性名可以加引号，也可以不加引号
// 2、将 Set 结构的数据转换为真正的数组：
let arr = [12, 45, 97, 9797, 564, 134, 45642];
let set = new Set(arr);
console.log(Array.from(set)); // [ 12, 45, 97, 9797, 564, 134, 45642 ]
//   Array.from 还可以接受第二个参数，作用类似于数组的 map 方法，用来对每个元素进行处理，将处理后的值放入返回的数组。如下：
let arr = [12, 45, 97, 9797, 564, 134, 45642];
let set = new Set(arr);
console.log(Array.from(set, item => item + 1)); // [ 13, 46, 98, 9798, 565, 135, 45643 ]
// 3、将字符串转换为数组：
let str = "hello world!";
console.log(Array.from(str)); // ["h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d", "!"]
// 4、Array.from 参数是一个真正的数组：
console.log(Array.from([12, 45, 47, 56, 213, 4654, 154]));
// 像这种情况，Array.from 会返回一个一模一样的新数组
```

Array.reduce()

语法：

```
array.reduce(function(accumulator, currentValue, currentIndex, array), initialValue);
```

accumulator：累加器，即函数上一次调用的返回值。第一次的时候为 initialValue || arr[0]

currentValue：数组中函数正在处理的的值。第一次的时候 initialValue || arr[1]

currentIndex：数据中正在处理的元素索引，如果提供了 initialValue ，从 0 开始；否则从 1 开始

array： 调用 reduce 的数组

initialValue：可选项，累加器的初始值。没有时，累加器第一次的值为 currentValue；注意：在对没有设置初始值的空数组调用 reduce 方法时会报错。

```
//无初始值
[1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
  return accumulator + currentValue;
}); // 10
```

callback	accumulator	currentValue	currentIndex	array	return value
first call	1 数组第一个元素	2 数组第二个元素	1(无初始值为 1)	[1, 2, 3, 4]	3
second call	3	3	2	[1, 2, 3, 4]	6
third call	6	4	3	[1, 2, 3, 4]	10

```
//有初始值
[1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
  return accumulator + currentValue;
}, 10); // 20
```

callback	accumulator	currentValue	currentIndex	array	return value
first call	10(初始值)	1(数组第一个元素)	0(有初始值为 0)	[1, 2, 3, 4]	11
second call	11	2	1	[1, 2, 3, 4]	13
third call	13	3	2	[1, 2, 3, 4]	16
fourth call	16	4	3	[1, 2, 3, 4]	20

```
//1.数组元素求和
[1, 2, 3, 4].reduce((a, b) => a + b); //10
```

```
//2.二维数组转化为一维数组
[[1, 2], [3, 4], [5, 6]]
  .reduce((a, b) => a.concat(b), []) //[1, 2, 3, 4, 5, 6]

  [
    //3.计算数组中元素出现的次数
    (1, 2, 3, 1, 2, 3, 4)
  ].reduce((items, item) => {
    if (item in items) {
      items[item]++;
    } else {
      items[item] = 1;
    }
    return items;
  }, {}) //{1: 2, 2: 2, 3: 2, 4: 1}

  [
    //数组去重①
    (1, 2, 3, 1, 2, 3, 4, 4, 5)
  ].reduce((init, current) => {
    if (init.length === 0 || init.indexOf(current) === -1) {
      init.push(current);
    }
    return init;
  }, []) //[1, 2, 3, 4, 5]
  [
    //数组去重②
    (1, 2, 3, 1, 2, 3, 4, 4, 5)
  ].sort()
  .reduce((init, current) => {
    if (init.length === 0 || init[init.length - 1] !== current) {
      init.push(current);
    }
    return init;
  }, []); //[1, 2, 3, 4, 5]
```

7.var let 在 for 循环中的区别

8.Set 数据结构

答案：- es6 方法,Set 本身是一个构造函数，它类似于数组，但是成员值都是唯一的。

```
const set = new Set([1, 2, 3, 4, 4]);
console.log([...set]); // [1,2,3,4]
console.log(Array.from(new Set([2, 3, 3, 5, 6]))); //[2,3,5,6]
```

9.Class 的讲解

答案：

- class 语法相对原型、构造函数、继承更接近传统语法，它的写法能够让对象原型的写法更加清晰、面向对象编程的语法更加通俗 这是 class 的具体用法。

10.模板字符串

答案：

- 就是这种形式${varible},在以往的时候我们在连接字符串和变量的时候需要使用这种方式'string' + varible + 'string'但是有了模版语言后我们可以使用 string${varible}string 这种进行连接。基本用途有如下：

1、基本的字符串格式化，将表达式嵌入字符串中进行拼接，用${}来界定。

```
//es5
var name = "lux";
console.log("hello" + name);
//es6
const name = "lux";
```

```
console.log(`hello ${name}`); //hello lux
```
2、在 ES5 时我们通过反斜杠()来做多行字符串或者字符串一行行拼接，ES6 反引号(``)直接搞定。
```
//ES5
var template =
  "hello \
world";
console.log(template); //hello world

//ES6
const template = `hello
world`;
console.log(template); //hello 空行 world
```
11.箭头函数需要注意的地方

箭头函数有几个使用注意点。

（1）函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象。

（2）不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。

（3）不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（4）不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。

上面四点中，第一点尤其值得注意。this 对象的指向是可变的，但是在箭头函数中，它是固定的。
```
function foo() {
  setTimeout(() => {
    console.log("id:", this.id);
  }, 100);
}
var id = 21;
foo.call({ id: 42 });
// id: 42
```
12.ES6 如何动态加载 import
```
import("lodash").then(_ => {
  // Do something with lodash (a.k.a '_')...
});
```
14.谈一谈你对 ECMAScript6 的了解？

答案：ES6 新的语法糖，类，模块化等新特性

15.箭头函数和普通函数有什么区别

答案：

- 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象，用 call apply bind 也不能改变 this 指向
- 不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。
- 不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
- 不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。
- 箭头函数没有原型对象 prototype

16.Promise 构造函数是同步执行还是异步执行，那么 then 方法呢？

17.ES5/ES6 的继承除了写法以外还有什么区别？

18.对 Promise 的理解

19.generator 原理

20.说说箭头函数的特点

21.请介绍 Promise，异常捕获（网易）

22.promise 如何实现 then 处理（宝宝树）

23.Promise.all 并发限制

24.介绍下 Promise.all 使用、原理实现及错误处理

25.设计并实现 Promise.race()