ES6 中的基本数据类型是：

Number、String、Null、Undefined、Symbol、Boolean

用typeof可以检测出变量的基本数据类型，但是有个特例，就是null的typeof返回的是object，这是个javascript的历史bug。。

typeof Symbol() //"symbol"

typeof Number() //"number"

typeof String() //"string"

typeof Function() //"function"

typeof Object() //"object"

typeof Boolean() //"boolean"

typeof null //"object"

typeof undefined //"undefined"

1、es6 的新特性有哪些

1、let&&const

2、iterable

3、解构赋值

4、箭头函数

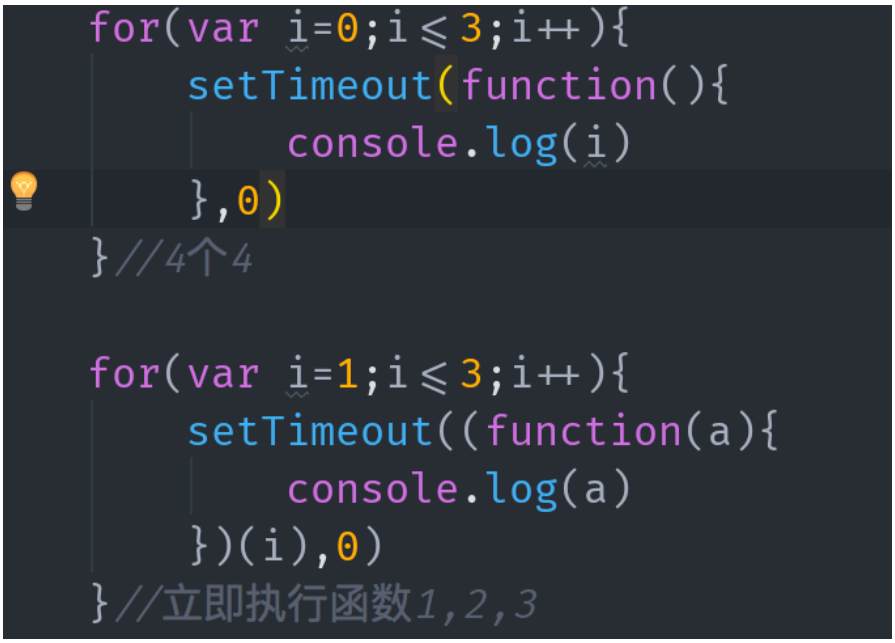
5、模板字符串

6、延展操作符

7、类

Const 只读属性不可修改，但是const 对象内部的key值可以修改 比如说const Test={}不可修改，但是const Test={key1:value1}中key1就可以修改

var全局变量污染的时候解决方法：let、闭包、立即执行函数（let和const）



把以下代码使用两种方法，依次输出0-9

```
var funcs = []
for (var i = 0; i < 10; i++) {
  funcs.push(function () {
    console.log(i)
  })
}
funcs.forEach(function (func) {
  func(); //输出十个10
})
```

方法一：let

```
var funcs = []
for (let i = 0; i < 10; i++) {
  funcs.push(function () {
    console.log(i)
  })
}
funcs.forEach(function (func) {
  func(); //依次输出0-9
})
```

方法二：闭包

```
function show(i) {
  return function () {
    console.log(i)
  }
}
var funcs = []
for (var i = 0; i < 10; i++) {
  funcs.push(show(i))
}
funcs.forEach(function (func) {
  func(); //0 1 2 3 4 5 6 7 8 9
})
```

方法三：立即执行函数

```
var funcs = []
for (var i = 0; i < 10; i++) {
  funcs.push((function (value) {
    return function () {
      console.log(value)
    }
  }(i)))
}
funcs.forEach(function (func) {
  func(); //依次输出0-9
})
```

箭头函数

```
class Animal {
  constructor() {
    this.type = "animal";
  }
  say(val) {
    setTimeout(function () {
      console.log(this); //window
      console.log(this.type + " says " + val);
    }, 1000)
  }
}
var animal = new Animal();
animal.say("hi"); //undefined says hi
```

使用箭头函数之后

```
class Animal {
  constructor() {
    this.type = "animal";
  }
  say(val) {
    setTimeout(() => {
      console.log(this); //Animal
      console.log(this.type + ' says ' + val);
    }, 1000)
  }
}
var animal = new Animal();
animal.say("hi"); //animal says hi
```

【特点】

- 不需要function关键字来创建函数
- 省略return关键字
- 继承当前上下文的 this 关键字

箭头函数需要注意的地方

*当要求动态上下文的时候，就不能够使用箭头函数，也就是this的固定化。

1、在使用=>定义函数的时候，this的指向是定义时所在的对象，而不是使用时所在的对象；

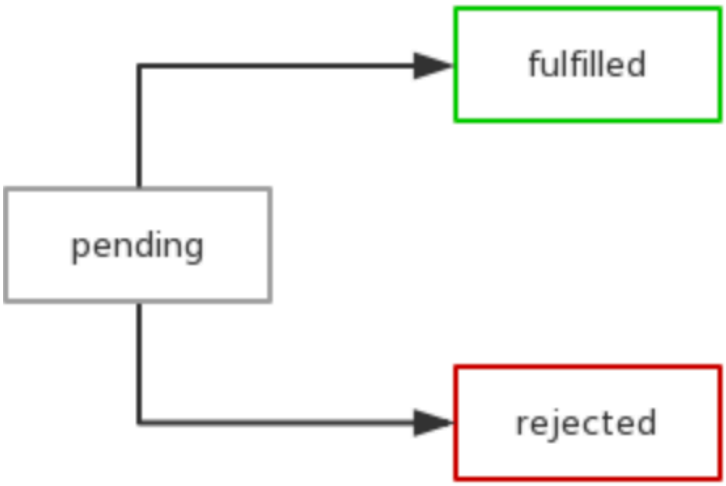
2、不能够用作构造函数，这就是说，不能够使用new命令，否则就会抛出一个错误；
3、不能够使用arguments对象；
4、不能使用yield命令；

Class

在ES6中，子类的构造函数必须含有super函数，super表示的是调用父类的构造函数，虽然是父类的构造函数，但是this指向的却是实例本身

promise的原理以及实现：

1、异步操作成功或者失败之后执行回调函数，回调函数接收一个数组（因为有可能同时有很多个回调）
2、加入延时机制（resolve或reject之后执行then()）就是通过setTimeout机制，将resolve中执行回调的逻辑放置到JS任务队列末尾，以保证在resolve执行时，then方法的回调函数已经注册完成.
3、状态机制（pending,fulfilled,rejected,pending可以转化为fulfilled或rejected并且只能转化一次，也就是说如果pending转化到fulfilled状态，那么就不能再转化到rejected。并且fulfilled和rejected状态只能由pending转化而来，两者之间不能互相转换。）



4、现在回顾下Promise的实现过程，其主要使用了设计模式中的观察者模式：

1. 通过Promise.prototype.then和Promise.prototype.catch方法将观察者方法注册到被观察者Promise对象中，同时返回一个新的Promise对象，以便可以链式调用。
2. 被观察者管理内部pending、fulfilled和rejected的状态转变，同时通过构造函数中传递的resolve和reject方法以主动触发状态转变和通知观察者。

Promise

1、怎么解决回调函数里面回调另一个函数，另一个函数的参数需要依赖这个回调函数。需要被解决的代码如下：

```
$http.get(url).success(function (res) {
  if (success != undefined) {
    success(res);
  }
}).error(function (res) {
  if (error != undefined) {
    error(res);
  }
});

function success(data) {
  if ( data.id != 0)  {
    var url = "getdata/data?id=" + data.id + "";
    $http.get(url).success(function (res) {
      showData(res);
    }).error(function (res) {
      if (error != undefined) {
        error(res);
      }
    });
  }
}
```

解决方案：promise.then()链式调用

```
$http.get(url).then(function(data) {
    if(res.id!=0){
        var newUrl="getdata/res?id=" + res.id + "";
        return $http.get(data);
    }
}).then(function Success(res) {
    showData(res)
    console.log("resolved: ", res);
}, function Error(err){
    console.log("rejected: ", err);
});
```

2、以下代码依次输出的内容是?

```
setTimeout(function () {
  console.log(1)
}, 0);
new Promise(function executor(resolve) {
  console.log(2);
  for (var i = 0; i < 10000; i++) {
    i == 9999 && resolve();
  }
  console.log(3);
}).then(function () {
  console.log(4);
});
console.log(5);
```

23541

首先先碰到一个 setTimeout，于是会先设置一个定时，在定时结束后将传递这个函数放到任务队列里面，因此开始肯定不会输出 1

然后是一个 Promise，里面的函数是直接执行的，因此应该直接输出 2 3 。

然后，Promise 的 then 应当会放到当前 tick 的最后，但是还是在当前 tick 中。

因此，应当先输出 5，然后再输出 4 ， 最后在到下一个 tick，就是 1 。

3、jQuery的ajax返回的是promise对象吗?

jquery的ajax返回的是deferred对象，通过promise的resolve()方法将其转换为promise对象。

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

4、promise只有2个状态，成功和失败，怎么让一个函数无论成功还是失败都能被调用?

使用promise.all()

Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。

Promise.all方法接受一个数组作为参数，数组里的元素都是Promise对象的实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例。）

示例:
var p =Promise.all([p1,p2,p3]);
p的状态由p1、p2、p3决定，分为两种情况。
当该数组里的所有Promise实例都进入Fulfilled状态：Promise.all**返回的实例才会变成Fulfilled状态。并将Promise实例数组的所有返回值组成一个数组，传递给Promise.all返回实例的回调函数**。

当该数组里的某个Promise实例都进入Rejected状态：Promise.all返回的实例会立即变成Rejected状态。并将第一个rejected的实例返回值传递给Promise.all返回实例的回调函数。

5、分析下列程序代码，得出运行结果，解释其原因

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  }, 1000)
})
const promise2 = promise1.then(() => {
  throw new Error('error!!!')
})

console.log('promise1', promise1)
console.log('promise2', promise2)

setTimeout(() => {
  console.log('promise1', promise1)
  console.log('promise2', promise2)
}, 2000)
```

运行结果：

```
promise1 Promise { <pending> }
promise2 Promise { <pending> }
(node:50928) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 1): Error: error!!!
(node:50928) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.
promise1 Promise { 'success' }
promise2 Promise {
  <rejected> Error: error!!!
    at promise.then (...)
    at <anonymous> }
```

原因：

promise 有 3 种状态：pending（进行中）、fulfilled（已完成，又称为Resolved） 或 rejected（已失败）。状态改变只能是 pending->fulfilled 或者 pending->rejected，状态一旦改变则不能再变。上面 promise2 并不是 promise1，而是返回的一个新的 Promise 实例。

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('once')
    resolve('success')
  }, 1000)
})

const start = Date.now()
promise.then((res) => {
  console.log(res, Date.now() - start)
})
promise.then((res) => {
  console.log(res, Date.now() - start)
})
```

运行结果：

```
once
success 1001
success 1001
```

原因：

promise 的 .then 或者 .catch 可以被调用多次，但这里 Promise 构造函数只执行一次。或者说 promise 内部状态一经改变，并且有了一个值，那么后续每次调用 .then 或者 .catch 都会直接拿到该值。

```
const promise = Promise.resolve()
  .then(() => {
    return promise
  })
promise.catch(console.error)
```

运行结果

```
TypeError: Chaining cycle detected for promise #<Promise>
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:188:7)
    at Function.Module.runMain (module.js:667:11)
    at startup (bootstrap_node.js:187:16)
    at bootstrap_node.js:607:3
```

原因

.then 或 .catch 返回的值不能是 promise 本身，否则会造成死循环。

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('once')
    resolve('success')
  }, 1000)
})

const start = Date.now()
promise.then((res) => {
  console.log(res, Date.now() - start)
})
promise.then((res) => {
  console.log(res, Date.now() - start)
})
```

运行结果：

```
once
success 1001
success 1001
```

原因：

promise 的 .then 或者 .catch 可以被调用多次，但这里 Promise 构造函数只执行一次。或者说 promise 内部状态一经改变，并且有了一个值，那么后续每次调用 .then 或者 .catch 都会直接拿到该值。

```
Promise.resolve(1)
  .then(2)
  .then(Promise.resolve(3))
  .then(console.log)
```

运行结果

```
1
```

原因

.then 或者 .catch 的参数期望是函数，传入非函数则会发生值穿透。

```
Promise.resolve()
  .then(function success (res) {
    throw new Error('error')
  }, function fail1 (e) {
    console.error('fail1: ', e)
  })
  .catch(function fail2 (e) {
    console.error('fail2: ', e)
  })
```

运行结果

```
fail2: Error: error
    at success (...)
    at ...
```

原因

.then 可以接收两个参数，第一个是处理成功的函数，第二个是处理错误的函数。.catch 是 .then 第二个参数的简便写法，但是它们用法上有一点需要注意：.then 的第二个处理错误的函数捕获不了第一个处理成功的函数抛出的错误，而后续的 .catch 可以捕获之前的错误。

```
process.nextTick(() => {
  console.log('nextTick')
})
Promise.resolve()
  .then(() => {
    console.log('then')
  })
setImmediate(() => {
  console.log('setImmediate')
})
console.log('end')
```

运行结果

```
end
nextTick
then
setImmediate
```

原因

process.nextTick 和 promise.then 都属于 microtask，而 setImmediate 属于 macrotask，在事件循环的 check 阶段执行。事件循环的每个阶段（macrotask）之间都会执行 microtask，事件循环的开始会先执行一次 microtask。

es6中引入类，为什么都有constructor，还有必须要有一个super函数?

ES6的class可以看作只是一个语法糖，它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。在constructor中必须调用 super方法，子类没有自己的this对象，而是继承父类的

this对象，然后对其进行加工。super代表了父类构造函数。对于你的实例相当于执行Component(props)。但是注意，此处this指向 子类。更严谨的是相当于Component.prototype.constructor.call(this,props)。

Symbol

产生背景：对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了

现在：对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突

1、不可以与其他类型进行运算，会报错

2、是不可遍历的

Set和WeakSet的区别

WeakSet 结构与 Set 类似，也是不重复的值的集合。但是，它与 Set 有两个区别。

首先，WeakSet 的成员只能是对象，而不能是其他类型的值。

其次，WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

因此，WeakSet 适合临时存放一组对象，以及存放跟对象绑定的信息。只要这些对象在外部消失，它在 WeakSet 里面的引用就会自动消失。

ES6 规定 WeakSet 不可遍历。(是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet 的一个用处，是储存 DOM 节点，而不用担心这些节点从文档移除时，会引发内存泄漏)

WeakMap与Map的区别有两点:

首先，WeakMap只接受对象作为键名（null除外），不接受其他类型的值作为键名。

其次，WeakMap的键名所指向的对象，不计入垃圾回收机制。

一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。

(如果你要往对象上添加数据，又不想干扰垃圾回收机制，就可以使用WeakMap)

weakMap案列:

```
Last login: Thu Mar 29 09:11:07 on ttys001
-bash: rt: command not found
zhaishuangMacBook-Pro:~ zhaishuang$ node --expose-gc
> global.gc();
undefined
> process.memoryUsage();
{ rss: 26259456,
  heapTotal: 6635520,
  heapUsed: 4815904,
  external: 12656 }
> let wm=new WeakMap();
undefined
> let key=new Array(5*1024*1024);
undefined
> wm.set(key,1);
WeakMap {}
> global.gc();
undefined
> process.memoryUsage();
{ rss: 69767168,
  heapTotal: 49639424,
  heapUsed: 46894544,
  external: 8667 }
> key=null;
null
> global.gc();
undefined
>
> process.memoryUsage();
{ rss: 27885568,
  heapTotal: 7684096,
  heapUsed: 5006840,
  external: 8690 }
>
```

Map的产生：

ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

Map 本身没有map和filter方法）

WeakMap 与 Map 在 API 上的区别主要是两个，一是没有遍历操作（即没有key()、values()和entries()方法），也没有size属性。二是无法清空，即不支持clear方法。因此，WeakMap只有四个方法可用：get()、set()、has()、delete()。

遍历器Iterator

1、每次调用next()方法，都会返回一个包含value和done两个属性的对象

2、会返回该数组的遍历器对象（即指针对象）it。

3、指针对象的next方法，用来移动指针

4、terator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环，当使用for...of循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。

5、对象obj是可遍历的（iterable），因为具有Symbol.iterator属性。执行这个属性，会返回一个遍历器对象。

数值求最值：

最大值：Math.max.apply(null,arr);

Math.max.apply(null, [14, 3, 77]) es5

Math.max(...[14, 3, 77])es6

最小值：Math.min.apply(null,arr);

Async await的使用返回后是否还要调用promise?

1、首先声明promise和async await诞生的初衷都是为了解决‘回调地狱’问题

Promise改进后：

```
getData()
.then(a ⇒ getMoreData(a))
.then(b ⇒ getMoreData(b))
.then(c ⇒ getMoreData(c))
.then(d ⇒ getMoreData(d))
.then(e ⇒ console.log(e));
```

async/await改进后：

```
(async () => {
    const a = await getData();
    const b = await getMoreData(a);
    const c = await getMoreData(b);
    const d = await getMoreData(c);
    const e = await getMoreData(d);
    console.log(e);
})();
```

2、async/await诞生的初衷为了简化多个Promise的同步操作，就像Promise要解决层层嵌套的回调函数的问题一样

async/await是在Promise的基础上做了改进，await是接收一个Promise对象，而当Promise执行到resolve()或者reject()的时候（fulfilled和rejected），await才会继续往下执行。

所以关键点就是得是返回Promise对象的函数才行，不然await等你后面的函数执行完了，见你没返回Promise对象，那他就继续执行了，不管你了。

async 函数

也就是说async函数会返回一个Promise对象。

- 如果async函数中是return一个值，这个值就是Promise对象中resolve的值；
- 如果async函数中是throw一个值，这个值就是Promise对象中reject的值。

```
async function imAsync(num) {
  if (num > 0) {
    return num // 这里相当于resolve(num)
  } else {
    throw num // 这里相当于reject(num)
  }
}

imAsync(1).then(function (v) {
  console.log(v); // 1
});

// 注意这里是catch
imAsync(0).catch(function (v) {
  console.log(v); // 0
})
```

await函数

await会暂停当前async函数的执行，等待后面的Promise的计算结果返回以后再继续执行当前的async函数。”

所以我们之前如果单纯的 await setTimeout(...) 或者 await exec(...) 是不行滴，await 不是什么都等，它等待的只是Promise，你如果没有给他返回个Promise，那它还是会继续向下执行。

所以 await 等待的不是所有的异步操作，等待的只是Promise。

3、应用场景：大量连续的异步操作

Promise/A+规范：

Promise表示一个异步操作的最终结果。与Promise最主要的交互方法是通过将函数传入它的then方法从而获取得Promise最终的值或Promise最终最拒绝（reject）的原因

1、术语

promise是一个包含了兼容promise规范then方法的对象或函数，

thenable 是一个包含了then方法的对象或函数。

value 是任何Javascript值。（包括 undefined, thenable, promise等).

exception 是由throw表达式抛出来的值。

reason 是一个用于描述Promise被拒绝原因的值。

2. 要求

一个Promise必须处在其中之一的状态：pending, fulfilled 或 rejected.

一个Promise必须提供一个then方法来获取其值或原因。Promise的then方法接受

两个可选参数：promise.then(onFulfilled, onRejected)
Promise解析过程 是以一个promise和一个值做为参数的抽象过程，可表示为[[Resolve]](promise, x). 过程如下
https://segmentfault.com/a/1190000002452115