1. let

let 关键字用来声明变量，使用 let 声明的变量有几个特点：

1. 不允许重复声明
2. 块儿级作用域
3. 不存在变量提升
4. 不影响作用域链

应用场景：项目编码中使用let代替var

2. const

const 关键字用来声明常量，const 声明有以下特点

1. 声明必须赋初始值
2. 标识符一般为大写
3. 不允许重复声明
4. 值不允许修改
5. 块儿级作用域

3. 解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构赋值。

数组的解构赋值

```
const arr = ['张学友', '刘德华', '黎明', '郭富城'];
let [zhang, liu, li, guo] = arr;
```

对象的解构赋值

```
const lin = {
    name: '林志颖',
    tags: ['车手', '歌手', '小旋风', '演员']
};
let {name, tags} = lin;
```

复杂对象的解构赋值

```
 let wangfei = {
     name: '王菲', age: 18,
     songs: ['红豆', '流年', '暧昧', '传奇'],
     history: [
         {name: '窦唯'},
         {name: '李亚鹏'},
         {name: '谢霆锋'}
     ]
 };
 let {songs: [one, two, three], history: [first, second, third]} = wangfei;
```

4. 模板字符串

模板字符串（template string）是增强版的字符串，用反引号（`）标识，具有以下特点：

- 字符串中可以出现换行
- 可以使用 `${xxx}` 形式输出变量

```
// 定义字符串
let str = `<ul>
            <li>沈腾</li>

            <li>玛丽</li>

            <li>魏翔</li>

            <li>艾伦</li>
           </ul>`;

// 变量拼接
let star = '王宁';

let result = `${star}在前几年离开了开心麻花`;
```

使用场景：当遇到字符串与变量拼接的情况下使用模板字符串。

5. 简化对象写法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
let name = '计算机';

let slogon = 'coding forever';

let improve = function () {
    console.log('改变事件');
}

//属性和方法简写
let department = {
    name,
    slogon,
    improve,
    change() {
        console.log('可以改变你')
    }
};
```

6. 箭头函数

ES6 允许使用「箭头」（=>）定义函数。

```
/**
* 1. 通用写法
*/
let fn = (arg1, arg2, arg3) => {
    return arg1 + arg2 + arg3;
}
```

箭头函数的注意点:

1. 如果形参只有一个，则小括号可以省略
2. 函数体如果只有一条语句，则花括号可以省略，函数的返回值为该条语句的执行结果
3. 箭头函数 this 指向声明时所在作用域下 this 的值
4. 箭头函数不能作为构造函数实例化
5. 不能使用 arguments

```
let school = {
    name: '清华大学',
    getName(){
        let fn5 = () => {
            console.log(this);
        }
        fn5();
    }
};
```

箭头函数中的this仍然为当前上下文的this指针。

7. rest参数

S6 引入 rest 参数，用于获取函数的实参，用来代替 arguments。

```
/**
* 作用与 arguments 类似
*/
function add(...args){
    console.log(args);
}
add(1,2,3,4,5);

/**
* rest 参数必须是最后一个形参
*/
function minus(a,b,...args){
    console.log(a,b,args);
}
minus(100,1,2,3,4,5,19);
```

rest参数适用于不定个数参数函数的场景。

8. spread扩展运算符

spread运算符即 `...`，用户对数据或对象进行解包。

```
/**
* 展开数组
*/

let tfboys = ['德玛西亚之力','德玛西亚之翼','德玛西亚皇子'];

function fn(){
    console.log(arguments);
}

```

```
fn(...tfboys)

/**
* 展开对象
*/
let skillOne = {
    q: '致命打击',
};
let skillTwo = {
    w: '勇气'
};
let skillThree = {
    e: '审判'
};
let skillFour = {
    r: '德玛西亚正义'
};

let gailun = {...skillOne, ...skillTwo,...skillThree,...skillFour};
```

9. Symbol

ES6 引入了一种新的原始数据类型 `Symbol`， 表示独一无二的值。它是JavaScript 语言的第七种数据类型，是一种类似于字符串的数据类型。

9.1. Symbol 特点

1. **Symbol 的值是唯一的，用来解决命名冲突的问题**
2. Symbol 值不能与其他数据进行运算
3. Symbol 定义的对象属性不能使用 for...in 循环遍历， 但是但是可以使用 `Reflect.ownKeys` 来获取对象的所有键名

9.2. 应用实例

```
//创建 Symbol
let s1 = Symbol();
console.log(s1, typeof s1);// Symbol() "symbol"

//添加标识的 Symbol
let s2 = Symbol('wl');

let s2_2 = Symbol('wl');
console.log(s2 === s2_2);// false

//使用 Symbol for 定义
let s3 = Symbol.for('wl');
let s3_2 = Symbol.for('wl');
console.log(s3 === s3_2);// true
```

9.3. symbol内置值

除了定义自己使用的 `Symbol` 值以外，ES6 还提供了11 个内置的Symbol 值，指向语言内部使用的方法。可以称这些方法为魔术方法，因为它们会在特定的场景下自动执行。

方法	含义
Symbol.hasInstance	当其他对象使用 instanceof 运算符，判断是否为该对象的实例时，会调用这个方法
Symbol.isConcatSpreadable	对象的 Symbol.isConcatSpreadable 属性等于的是一个布尔值，表示该对象用于 Array.prototype.concat()时，是否可以展开。
Symbol.species	创建衍生对象时，会使用该属性
Symbol.match	当执行 str.match(myObject) 时，如果该属性存在，会调用它，返回该方法的返回值。
Symbol.replace	当该对象被 str.replace(myObject)方法调用时，会返回 方法的返回值。
Symbol.search	当该对象被 str. search (myObject)方法调用时，会返回该方法的返回值。
Symbol.split	当该对象被 str. split (myObject)方法调用时，会返回该方法的返回值。
Symbol.iterator	对象进行 for...of 循环时，会调用 Symbol.iterator 方法， 返回该对象的默认遍历器
Symbol.toPrimitive	该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。
Symbol. toStringTag	在该对象上面调用 toString 方法时，返回该方法的返回值
Symbol. unscopables	该对象指定了使用 with 关键字时，哪些属性会被 with 环境排除。

10. 迭代器

遍历器（Iterator）就是一种机制。它是一种接口，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。

1. 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
2. 原生具备 iterator 接口的数据(可用 for of 遍历)
    - Array
    - Arguments
    - Set
    - Map
    - String
    - TypedArray
    - NodeList
3. 工作原理
    - 创建一个指针对象，指向当前数据结构的起始位置

- 第一次调用对象的 next 方法，指针自动指向数据结构的第一个成员
- 接下来不断调用 next 方法，指针一直往后移动，直到指向最后一个成员
- 每调用 next 方法返回一个包含 value 和done 属性的对象

11. Promise

`Promise` 是异步编程的一种解决方案，从语法上讲，Promise是一个对象，从它可以获取异步操作的消息；从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。

Promise有三种状态：pending(等待态)，fulfiled(成功态),rejected(失败态)；**状态一旦改变，就不会再变。**创造Promise实例后，它会立即执行。

11.1. 基本使用

- 创建Promise

```
let p = new Promise((resolve, reject) => {
    // 做一些异步操作
    setTimeout(() => {
      console.log('执行完成');
      resolve('成功结果')
    }, 2000);
})
```

- then和reject的用法

```
 new Promise((resolve, reject) => {
     // 请求数据
     cosnt {success, data} = request('/hello');
     if(success) {
         resolve(data)
     } else {
         reject('请求数据出错')
     }
 }).then((data) => {
     console.log("请求成功,数据=" + data);
 }, (errMsg) => {
     console.log(errMsg)
 })
```

- catch的用法

```
new Promise((resolve, reject) => {
    // 请求数据
    cosnt {success, data} = request('/hello');
    if(success) {
        resolve(data)
    } else {
        reject('请求数据出错')
    }
}).then((data) => {
    console.log("请求成功,数据=" + data);
}).catch((errMsg) => {
    console.log(errMsg)
})
```

> 效果与.then的第二个参数一样，也是reject的一个回调。
> 与.then的第二个参数的区别，**在运行时如果js代码报错，也会进入**catch。

11.2. Promise有什么作用

- 解决 **回调地狱问题**
- promise可以支持多个并发的请求，获取并发请求中的数据
- promise可以解决异步的问题，本身不能说promise是异步的

11.3. 扩展用法

- `Promise.all()`

  Promise.all()接收一个Promise数组的参数;

  const p = Promise.all([p1, p2, p3]);只有p1、p2、p3**全部成功，p才成功**,只要有一个失败，则p失败;

  有了all，就可以并行执行多个异步操作，然后在一个回调里处理所有数据。例如初始化一个游戏前可能需要很多资源，那就可以使用此功能做到所有资源都加载完成后才初始化游戏。

```
let promise1 = new Promise((resolve, reject) => {});
let promise2 = new Promise((resolve, reject) => {});
let promise3 = new Promise((resolve, reject) => {});
Promise.all([promise1, promise2, promise3]).then(() => {
    // 三个都成功,则成功
}, () => {
    // 只要有一个失败，则失败
})
```

- `Promise.race()`

  Promise.race()和Promise.all()一样，也是接收一个数组参数;
  const p = Promise.race([p1, p2, p3]);只要p1、p2、p3中**有一个状态率先改变，则P的状态随之改变。**

```
const p = Promise.race([
    fetch('/resource-that-may-take-a-while'),
    new Promise(function (resolve, reject) {
        setTimeout(() => reject(new Error('request timeout')), 5000)
    })
]);
p.then(console.log)
 .catch(console.error);
// 上面代码中，如果 5 秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。
```

12. generator函数

generator是ES6引入的新的数据类型。generator看上去像一个函数，但可以返回多次。

特点：

- function关键字和函数名之间有一个*号
- 函数体内容使用yield语句，定义不同的内部状态

12.1. 基本使用

```
function * gen() {
    yield "a";
    yield "b";
    yield "c";
    yield "ending";
}
```

> gen()函数有4个阶段，分别是“a，b，c，ending”
> gen()返回的并不是函数的执行结果，而是返回一个指向函数内部状态的迭代器对象。
>
> 1、分段执行，可以暂停
> 2、可以控制阶段和每个阶段的返回值
> 3、可以知道是否执行到结尾

12.2. generator函数的作用

普通函数是一次性生成所有的数据返回，若想获取每一个数据，那么需要使用for循环来迭代。**如果数据太多，则有可能造成内存溢出。**
生成函数可以一条一条的生成数据，这样就可以**避免占用更多的内存。**

12.3. 扩展用法

12.3.1. yield表达式

特点：

1. 遇到 `yield` 表达式时，会暂停执行后面的操作，并将紧跟在 `yield` 表达式后面的值作为返回对象的 `value` 属性；
2. `yield` 表达式只能用于生成函数，不能用于普通函数。

```
function* gen() {
  yield  123 + 456;
}
// 会在调用.next()方法后，执行 123 + 456
let g = gen();
g.next(); // {value: 579, done:false}
```

- `yield` 表达式只能用于生成函数，用于普通函数会报错

```
(function (){
  yield 1;
})()
// Error: Unexpected number
```

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a) {
  a.forEach(function (item) {
    if (typeof item !== 'number') {
      yield* flat(item);
    } else {
      yield item;
    }
  });
};

for (var f of flat(arr)){
  console.log(f);
}
```

> 上面代码也会产生句法错误，因为 `forEach` 方法的参数是一个普通函数，但是在里面使用了 `yield` 表达式。一种修改方法是改用 `for` 循环。

- `yield` 表达式如果用在另一个表达式之中，必须放在圆括号里面

```
function* demo() {
  console.log('Hello' + yield); // SyntaxError
  console.log('Hello' + yield 123); // SyntaxError

  console.log('Hello' + (yield)); // OK
  console.log('Hello' + (yield 123)); // OK
}
```

12.3.2. 与Iterator的关系

由于 Generator 函数就是遍历器生成函数，因此可以把 Generator 赋值给对象的 `Symbol.iterator` 属性，从而使得该对象具有 Iterator 接口。

```
var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

[...myIterable] // [1, 2, 3]
```

- 生成器对象的`Symbol.iterator`属性就是其对应的生成器函数

  ```
  function* gen(){
    // some code
  }

  var g = gen();

  g[Symbol.iterator]() === g
  // true
  ```

  上面代码中，`gen`是一个 Generator 函数，调用它会生成一个遍历器对象`g`。它的`Symbol.iterator`属性，也是一个遍历器对象生成函数，执行后返回它自己。

12.3.3. next方法的参数

`yield`表达式本身没有返回值，或者说总是返回`undefined`。`next`方法可以带一个参数，该参数就会被当作上一个`yield`表达式的返回值。

```
function* f() {
  for(var i = 0; true; i++) {
    var reset = yield i;
    if(reset) { i = -1; }
  }
}

var g = f();

g.next() // { value: 0, done: false }
g.next() // { value: 1, done: false }
g.next(true) // { value: 0, done: false }
```

上面代码先定义了一个可以无限运行的 Generator 函数`f`，如果`next`方法没有参数，每次运行到`yield`表达式，变量`reset`的值总是`undefined`。当`next`方法带一个参数`true`时，变量`reset`就被重置为这个参数（即`true`），因此`i`会等于`-1`，下一轮循环就会从`-1`开始递增。

这个功能有很重要的语法意义。Generator 函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过`next`方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，**可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。**

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
```

```
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

> 从语义上讲，第一个 `next` 方法用来启动遍历器对象，所以不用带有参数。

- 如果想要第一次调用 `next` 方法时，就能够输入值，可以在 Generator 函数外面再包一层

```
function wrapper(generatorFunction) {
  return function (...args) {
    let generatorObject = generatorFunction(...args);
    generatorObject.next();
    return generatorObject;
  };
}

const wrapped = wrapper(function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
});

wrapped().next('hello!')
// First input: hello!
```

12.3.4. for ...of循环

`for...of` 循环可以自动遍历Generator函数运行时的 `Iterator` 对象，且此时不再需要调用 `next` 方法。

```
function* foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
  return 6;
}

for (let v of foo()) {
  console.log(v);
}
// 1 2 3 4 5
```

- 实现斐波那契数列

```
function* fibonacci() {
  let [prev, curr] = [0, 1];
  for (;;) {
    yield curr;
    [prev, curr] = [curr, prev + curr];
  }
}

for (let n of fibonacci()) {
  if (n > 1000) break;
  console.log(n);
}
```

12.3.5. Generator.prototype.throw()

Generator 函数返回的遍历器对象，都有一个`throw`方法，可以在函数体外抛出错误，然后在 Generator 函数体内捕获。

```
var g = function* () {
  try {
    yield;
  } catch (e) {
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 内部捕获 a
// 外部捕获 b
```

上面代码中，遍历器对象`i`连续抛出两个错误。第一个错误被 Generator 函数体内的`catch`语句捕获。`i`第二次抛出错误，由于 Generator 函数内部的`catch`语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了 Generator 函数体，被函数体外的`catch`语句捕获。

- `throw`方法抛出的错误要被内部捕获，前提是必须至少执行过一次`next`方法

```
function* gen() {
  try {
    yield 1;
  } catch (e) {
    console.log('内部捕获');
  }
}

var g = gen();
g.throw(1);
// Uncaught 1
```

- `throw` 方法被捕获以后，会附带执行下一条 `yield` 表达式。也就是说，会附带执行一次 `next` 方法

```
var gen = function* gen(){
  try {
    yield console.log('a');
  } catch (e) {
    // ...
  }
  yield console.log('b');
  yield console.log('c');
}

var g = gen();
g.next() // a
g.throw() // b
g.next() // c
```

一旦 Generator 执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。如果此后还调用 `next` 方法，将返回一个 `value` 属性等于 `undefined` 、 `done` 属性等于 `true` 的对象，即 JavaScript 引擎认为这个 Generator 已经运行结束了。

12.3.6. Generator.prototype.return()

Generator 函数返回的遍历器对象，还有一个 `return()` 方法，可以返回给定的值，并且终结遍历 Generator 函数。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()        // { value: 1, done: false }
g.return('foo') // { value: "foo", done: true }
g.next()        // { value: undefined, done: true }
```

如果 Generator 函数内部有 `try...finally` 代码块，且正在执行 `try` 代码块，那么 `return()` 方法会导致立刻进入 `finally` 代码块，执行完以后，整个函数才会结束。

```
function* numbers () {
  yield 1;
  try {
    yield 2;
    yield 3;
  } finally {
    yield 4;
    yield 5;
  }
  yield 6;
}
var g = numbers();
g.next() // { value: 1, done: false }
g.next() // { value: 2, done: false }
g.return(7) // { value: 4, done: false }
```

```
g.next() // { value: 5, done: false }
g.next() // { value: 7, done: true }
```

12.3.7. next()、throw()、return() 的共同点

`next()`、`throw()`、`return()` 这三个方法本质上是同一件事，可以放在一起理解。它们的作用都是让 Generator 函数恢复执行，并且使用不同的语句替换 `yield` 表达式。

- `next()` 是将 `yield` 表达式替换成一个值。

```
const g = function* (x, y) {
  let result = yield x + y;
  return result;
};

const gen = g(1, 2);
gen.next(); // Object {value: 3, done: false}

gen.next(1); // Object {value: 1, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = 1;
```

  上面代码中，第二个 `next(1)` 方法就相当于将 `yield` 表达式替换成一个值 `1`。如果 `next` 方法没有参数，就相当于替换成 `undefined`。

- `throw()` 是将 `yield` 表达式替换成一个 `throw` 语句

```
gen.throw(new Error('出错了')); // Uncaught Error: 出错了
// 相当于将 let result = yield x + y
// 替换成 let result = throw(new Error('出错了'));
```

- `return()` 是将 `yield` 表达式替换成一个 `return` 语句

```
gen.return(2); // Object {value: 2, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = return 2;
```

12.4. yield * 表达式

如果在 Generator 函数内部，调用另一个 Generator 函数。需要在前者的函数体内部，自己手动完成遍历。

```
function* foo() {
  yield 'a';
  yield 'b';
}

function* bar() {
  yield 'x';
  // 手动遍历 foo()
  for (let i of foo()) {
    console.log(i);
  }
  yield 'y';
}

```

```
for (let v of bar()){
  console.log(v);
}
// x
// a
// b
// y
```

上面代码中，`foo`和`bar`都是 Generator 函数，在`bar`里面调用`foo`，就需要手动遍历`foo`。如果有多个 Generator 函数嵌套，写起来就非常麻烦。

ES6 提供了`yield*`表达式，作为解决办法，用来在一个 Generator 函数里面执行另一个 Generator 函数。

```
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  yield 'a';
  yield 'b';
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  for (let v of foo()) {
    yield v;
  }
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// "x"
// "a"
// "b"
// "y"
```

- 如果被代理的 Generator 函数有`return`语句，那么就可以向代理它的 Generator 函数返回数据

```
function* foo() {
  yield 2;
  yield 3;
  return "foo";
}

function* bar() {
  yield 1;
  var v = yield* foo();
  console.log("v: " + v);
  yield 4;
```

```
}

var it = bar();

it.next()
// {value: 1, done: false}
it.next()
// {value: 2, done: false}
it.next()
// {value: 3, done: false}
it.next();
// "v: foo"
// {value: 4, done: false}
it.next()
// {value: undefined, done: true}
```

12.5. 作为对象属性的Generator函数

```
let obj = {
  * myGeneratorMethod() {
    ···
  }
};
// 等价于
let obj = {
  myGeneratorMethod: function* () {
    // ···
  }
};
```

13. Set

ES6 提供了新的数据结构 Set（集合）。它类似于数组，但成员的值都是唯一的，集合实现了 iterator 接口，所以可以使用『扩展运算符』和『for…of…』进行遍历，集合的属性和方法：

1. size 返回集合的元素个数
2. add 增加一个新元素，返回当前集合
3. delete 删除元素，返回 boolean 值
4. has 检测集合中是否包含某个元素，返回 boolean 值
5. clear 清空集合，返回 undefined

```
//创建一个空集合
let s = new Set();
//创建一个非空集合
let s1 = new Set([1,2,3,1,2,3]);

//集合属性与方法
//返回集合的元素个数
console.log(s1.size);
//添加新元素
console.log(s1.add(4));
//删除元素
console.log(s1.delete(1));
//检测是否存在某个值
```

```
console.log(s1.has(2));
//清空集合
console.log(s1.clear());
```

14. Map

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。Map 也实现了iterator 接口，所以可以使用『扩展运算符』和『for…of…』进行遍历。Map 的属性和方法：

1. size 返回 Map 的元素个数
2. set 增加一个新元素，返回当前 Map
3. get 返回键名对象的键值
4. has 检测 Map 中是否包含某个元素，返回 boolean 值
5. clear 清空集合，返回 undefined

```
//创建一个空 map
let m = new Map();
//创建一个非空 map
let m2 = new Map([
    ['name','zhangsan'],
    ['slogon','codeing everywhere']
]);
//属性和方法
//获取映射元素的个数
console.log(m2.size);
//添加映射值
console.log(m2.set('age', 6));
//获取映射值
console.log(m2.get('age'));
//检测是否有该映射
console.log(m2.has('age'));
//清除
console.log(m2.clear());
```

15. class类

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 class 关键字，可以定义类。基本上，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。

知识点：

1. class 声明类
2. constructor 定义构造函数初始化
3. extends 继承父类
4. super 调用父级构造方法
5. static 定义静态方法和属性
6. 父类方法可以重写

15.1. 基本使用

```
//父类
```

```
class Phone {
    //构造方法
    constructor(brand, color, price) {
    this.brand = brand; this.color = color; this.price = price;
    }

    //对象方法
    call() {
        console.log('我可以打电话!!!')
    }
}

//子类
class SmartPhone extends Phone {
    constructor(brand, color, price, screen, pixel) {
        super(brand, color, price);
        this.screen = screen; this.pixel = pixel;
     }

    //子类方法
    photo() {
        console.log('我可以拍照!!');
    }
    playGame() {
        console.log('我可以打游戏!!');
    }

    //方法重写
    call() {
        console.log('我可以视频通话!!');
    }

    // 静态方法
    static run() {
        console.log('开机....!!');
    }
}

//实例化对象
const Nokia = new Phone('诺基亚', '灰色', 230);

const iPhone6s = new SmartPhone('苹果', '白色', 6088, '4.7inch','500w');

//调用子类方法
iPhone6s.playGame();
//调用重写方法
iPhone6s.call();
//调用静态方法
SmartPhone.run();
```

16. 数值扩展

- `Number.isFinite()`

  用来检查一个数值是否为有限的

  如果参数类型不是数值，`Number.isFinite` 一律返回 `false`

- `Number.isNaN()`

  用来检查一个值是否为 NaN

- `Math.trunc`

  用于去除一个数的小数部分，返回整数部分

```
Math.trunc(4.1) // 4
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-4.9) // -4
Math.trunc(-0.1234) // -0
```

```
Math.trunc('123.456') // 123
Math.trunc(true) //1
Math.trunc(false) // 0
Math.trunc(null) // 0
```

17. 对象扩展

ES6 新增了一些 Object 对象的方法

- `Object.is`

  比较两个值是否严格相等，与 `===` 行为基本一致

```
Object.is('foo', 'foo')
// true
Object.is({}, {})
// false
```

- `Object.assign`

  对象的合并，将源对象的所有可枚举属性，复制到目标对象

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

18. 模块化

模块功能主要由两个命令构成：export 和 import。

18.1. export

用于规定模块的对外接口

- 输出变量

```
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;

export { firstName, lastName, year };
```

- 输出变量或class

```
export function multiply(x, y) {
  return x * y;
};
```

- 重命名输出信息

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

18.2. import

用于导入其他模块提供的功能

- 导入

```
import { firstName, lastName, year } from './profile.js';

function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

- 导入后重命名

```
import { lastName as surname } from './profile.js';
```