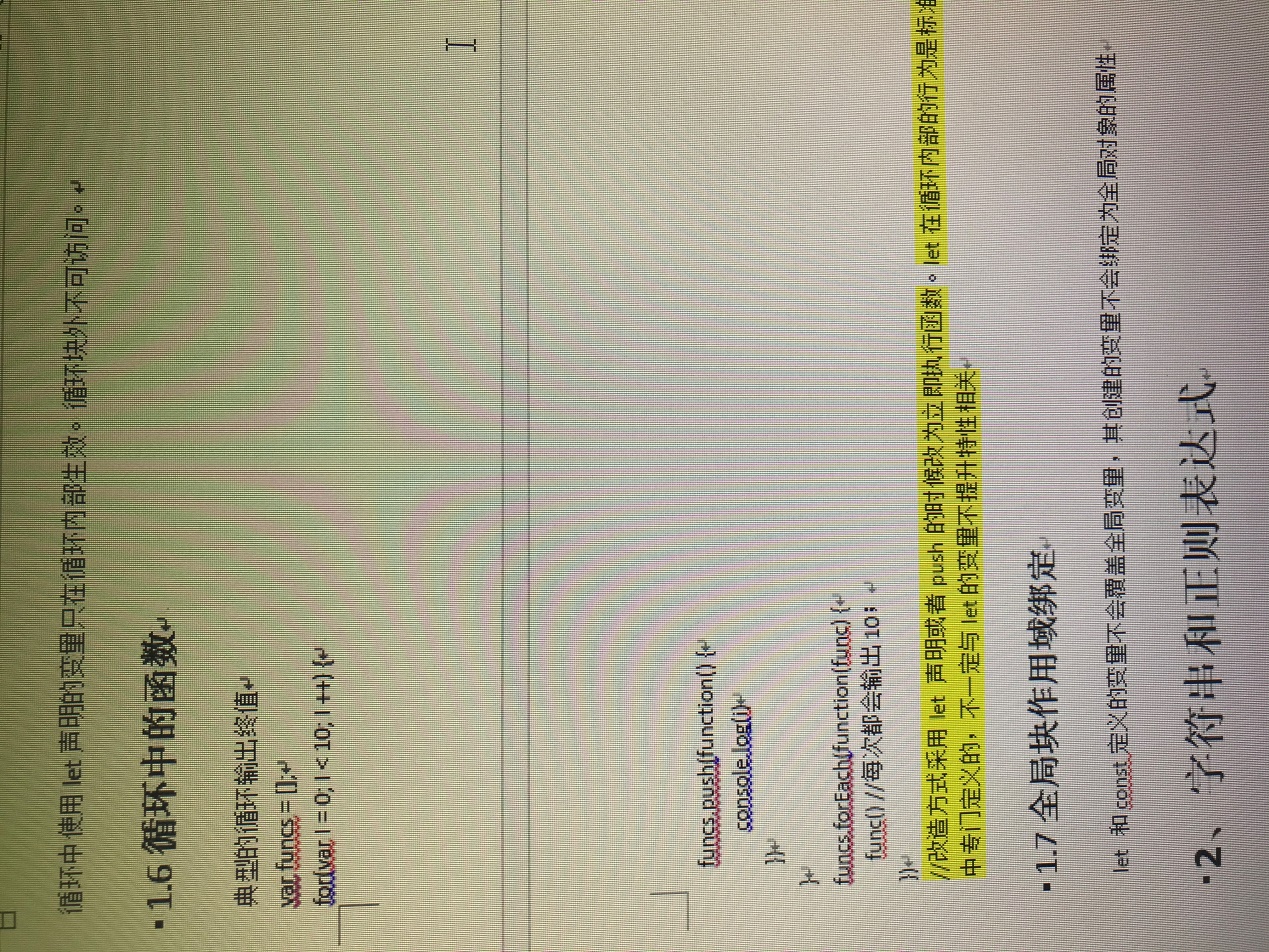
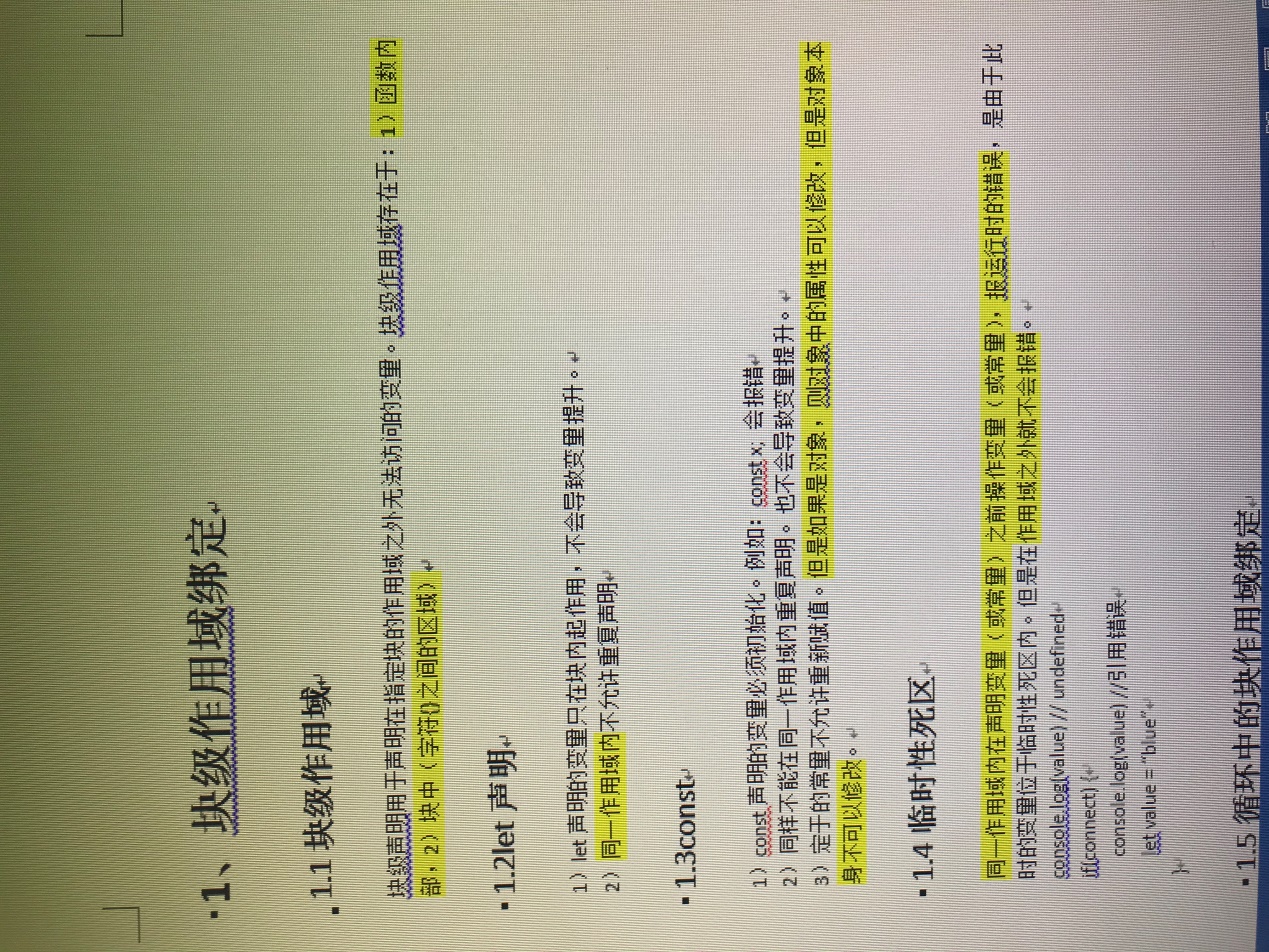
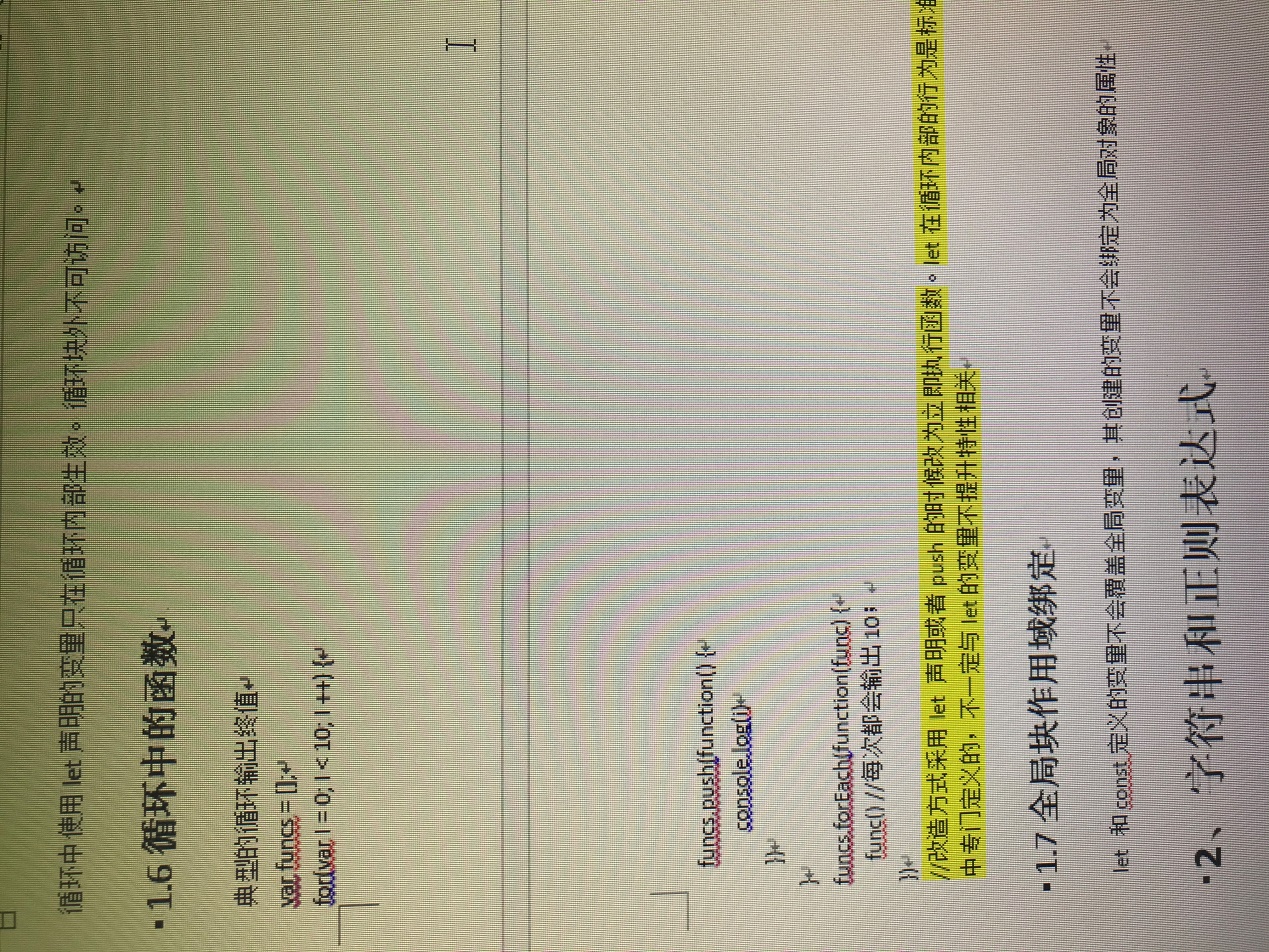
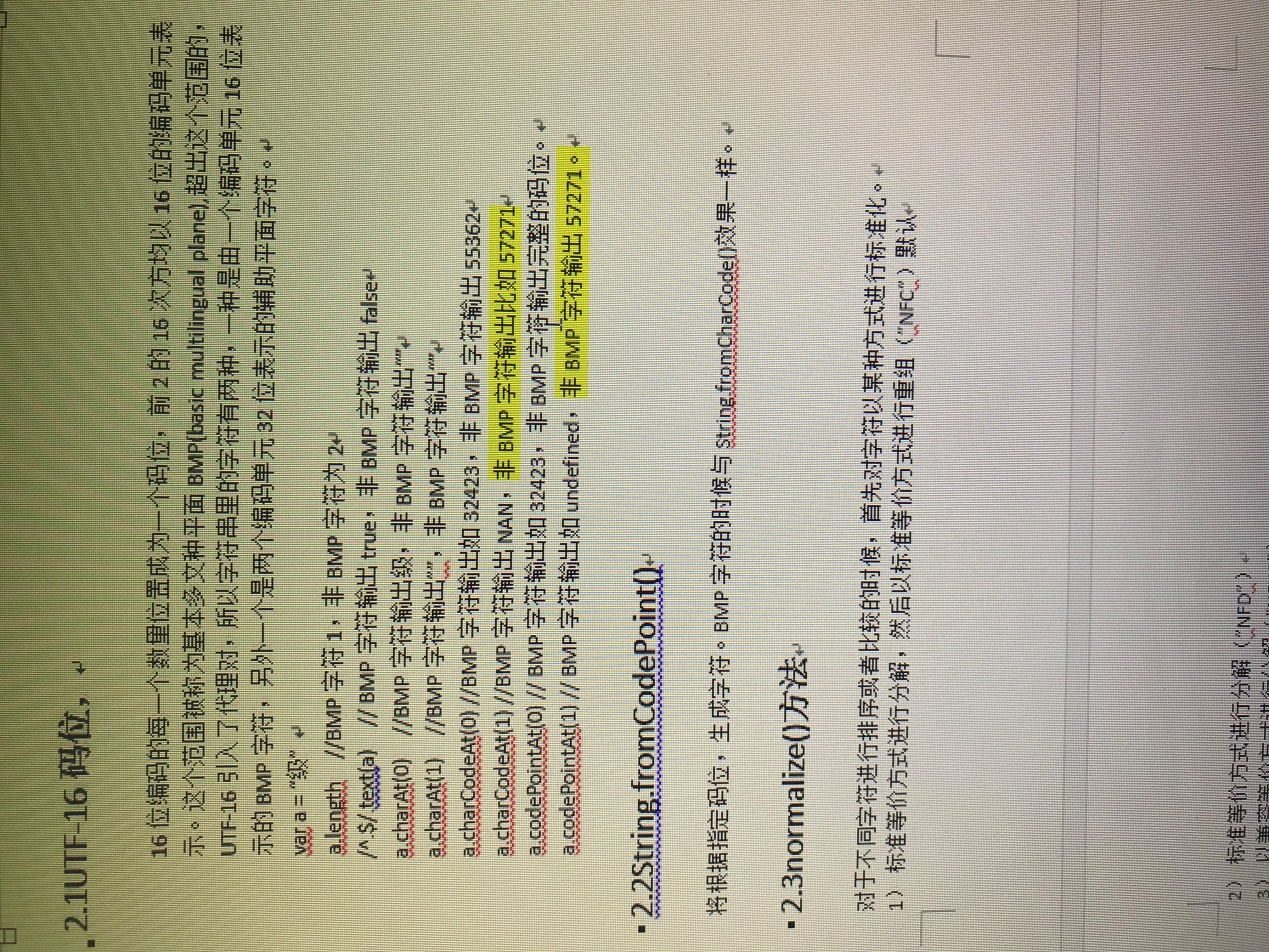
Es6笔记

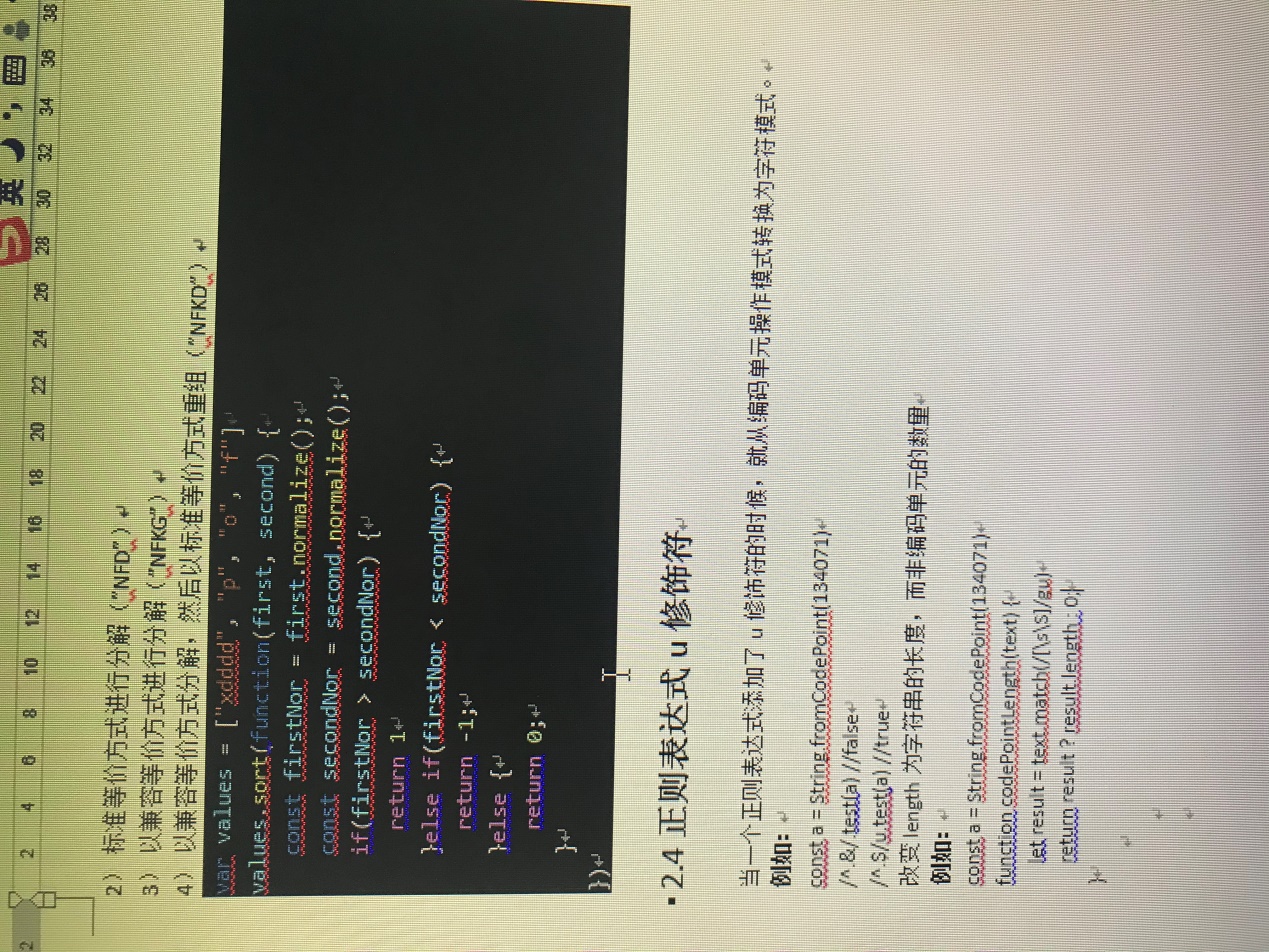
# 1、块级作用域绑定





# 2、字符串和正则表达式





**2.5其他字符串变更**

1）str.includes(childstr,startIndex) //返回true或者false

2）startWith(childStr, startIndex) //true或者false

3）endWith(childStr,index) //true或者false 如果传第二个参数，则从index-childStr.length的位置开始正向匹配

4）lastIndexOf(childStr)和indexOf(chilsStr)能够传入正则表达式作为参数，1）2）3）则不可以。

5）str.repeat(number) //返回字符串重复number次以后的字符串

**2.6正则表达式y修饰符**

1）

let text = "hello1 hello2 hello3";

let parttern = /hello\d\s?/;

let result = parttern.exec(text);

console.log(parttern.lastIndex) //此时parttern.lastIndex为0；

parttern.lastIndex = 1; //修改lastIndex为1，parttern会忽略；

result = parttern.exec(text);

console.log(parttern.lastIndex) //为0

console.log(result[0]) //parttern会忽略，所以返回还是"hello1 "

let globalPattern = /hello\d\s?/g;

let globalResult = globalPattern.exec(text);

console.log(globalPattern.lastIndex) //此时globalPattern的lastIndex为7；

globalPattern.lastIndex = 1; //修改lastIndex为1，globalPattern不会忽略，而是更改为1；

globalResult = globalPattern.exec(text); //从字母e开始匹配，往后找到"hello2 "匹配成功

console.log(globalPattern.lastIndex) //此时为14；

console.log(globalResult[0]) //"hello2 "

let stickyPattern = /hello\d\s?/y;

let stickyResult = stickyPattern.exec(text);

console.log(stickyPattern.lastIndex) //此时stickyPattern的lastIndex为7；

stickyPattern.lastIndex = 1; //修改lastIndex为1，stickyPattern不会忽略，而是更改为1；

stickyResult = stickyPattern.exec(text) //从字母e开始匹配，没有匹配到，返回null,如果上一步lastIndex不做修改，则此时lastIndex为14；

console.log(stickyPattern.lastIndex); //此时为0

console.log(stickyResult) //为null

注：y修饰符只对正则表达式的方法exec()，test()方法生效，对字符串的方法match()不起作用,对于粘制表达式（含有y修饰符）而言，如果正则中使用^来匹配，则只会从字符串的起始位置或者多行的首行开始匹配。

2）检测y修饰符是否存在

let pattern = /hello\d/y;

console.log(pattern.sticky)如果支持粘制修饰符，则sticky为true,此属性为只读属性。

function hasRegExp() { //判断js引擎是否支持y修饰符。

try{

new RegExp(“.”, y)

return true;

}catch(err) {return false}

}

**2.7复制正则表达式**

es6中

let reg1 = /ab/i;

let reg2 = new RegExp(reg1, “g”) //可以更改修饰符，不更改的话和reg1保持一样。

2.8 flags和source

let reg1 = /ab/i reg1.source //”ab” reg1.flags //i

**2.8模板字面量``**

1）实现多行字符串：var messge = `Hello

world`

2）占位符：`${任意的js表达式}`

3）模板标签:标签例如下面的tag就是

const fs = require("fs");

const path = require("path")

const writeFile = path.resolve(\_\_dirname, "./text.txt");

const name ="zhang"

const age = 10;

function tag(iterals, ...subArr) { //iterals数组有三个参数：a第一个`后面第一个占位符前面的””

//b第一个占位符后面和第二个占位符之间的字符串

//c第二个占位符后面的

//当然如果占位符有多个，数组元素个数也随之更改

//iteral有个等价iteral.raw数组，包含它的原生字符串信息。

// subArr有两个元素，分别对应占位符解析后的值

let str = "";

for(let i = 0; i < subArr.length; i++) {

str+=iterals[i]

str+=subArr[i]

}

str+= iterals[iterals.length-1];

fs.writeFile(writeFile, str, {encoding: "utf-8"}, function(err, data) {

console.log(err)

console.log(data)

})

// return str;

}

tag `My name is ${name}, my age is ${age}` //注意这里的调用方式，不要加“()”

3）String.raw() 字面量等价的字符串的原生字符串，多用于输出含有代码的字符串，代码中含有转义字符的时候，不需要在执行的时候转义。

如：String.raw `Multiline\nstrimg` //” Multiline\nstrimg”

# 3、函数

**3.1、 es6的默认参数：**

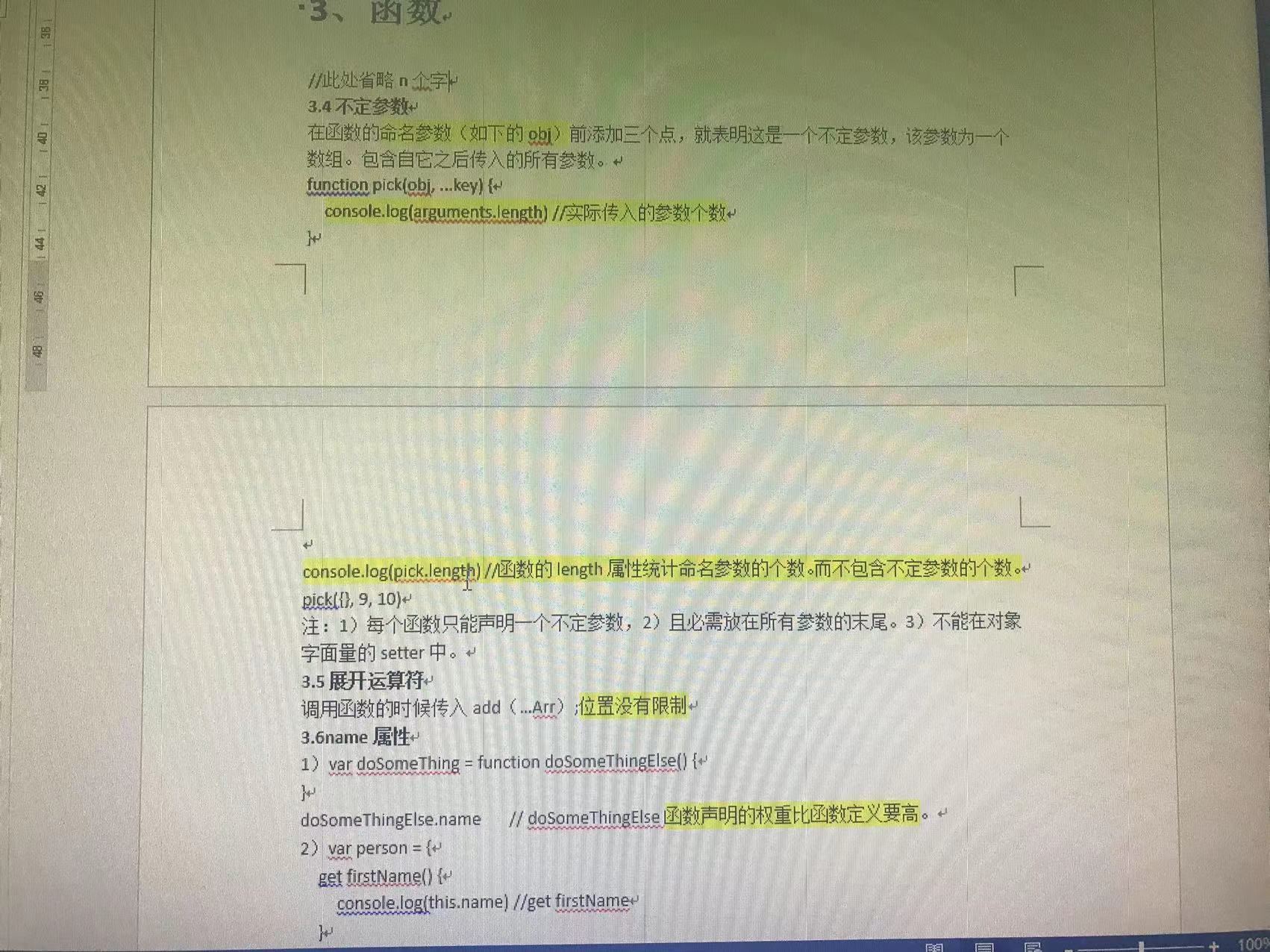
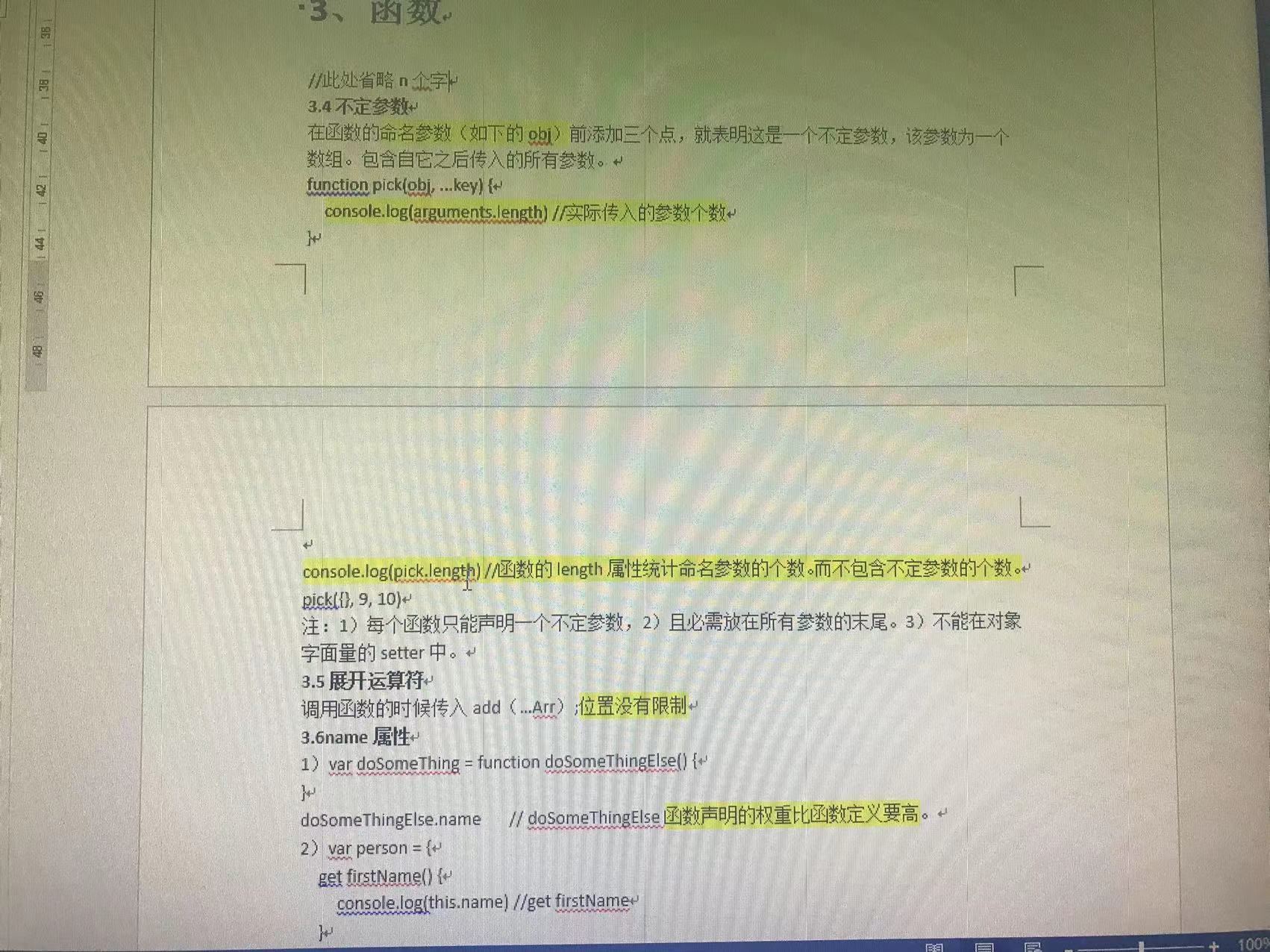
默认参数可以是任意位置元素。并且null不会使用默认参数

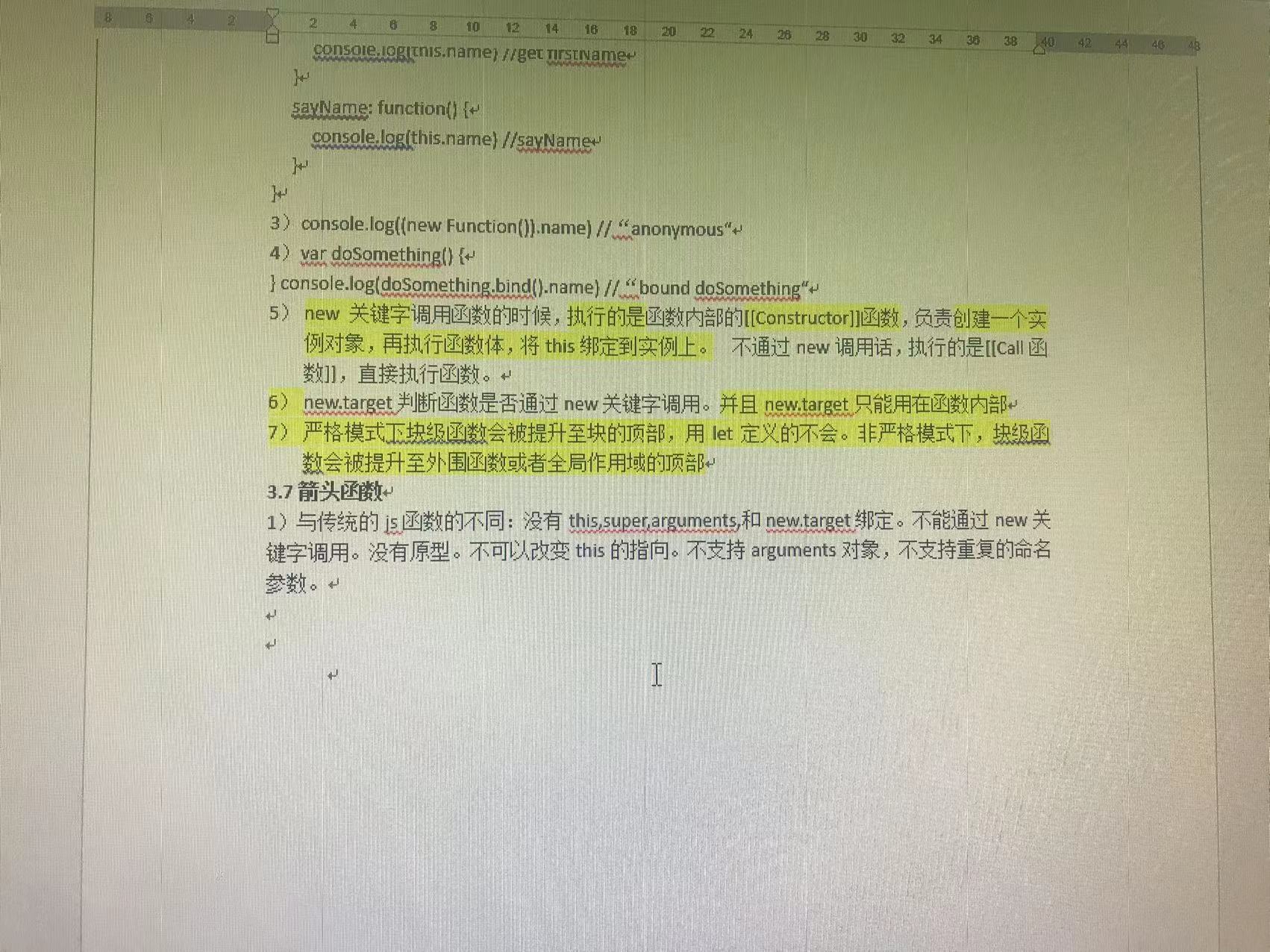
3.2、默认值参数对arguments对象的影响：

ES5非严格模式下，在函数内部修改参数的值，arguments中的元素会随之修改。

ES5严格模式解决了这一问题。也就是arguments中的元素值和传入的实参保持一致。Es6中如果传入了默认参数，则也和严格模式一样。其值也为实际传入的实参。

3.3默认参数可以是函数调用。后面的参数的默认值可以是引用前一个参数，反之则不可以。





2）function createArrow() {

return ()=> arguments[0] //指的是外围函数的参数。

}

4）箭头函数仍然可以使用call() apply() bind()的方法。

5）尾调用优化：在函数内部，尾调用是最后一条语句。尾调用的结果作为函数值返回。

尾调用函数不是一个闭包。

# 4、扩展对象的功能性。

**4.1属性初始值的简写**

function createPerson(name, age) { //这里的参数必须和属性名同名才可以简写

return {

name,

age,

}

}

**4.2对象方法的简写方法。**

var person = {

sayName() {

return this.name; // sayName

}

}

**4.3可计算属性名**

let lastName = “last name”;

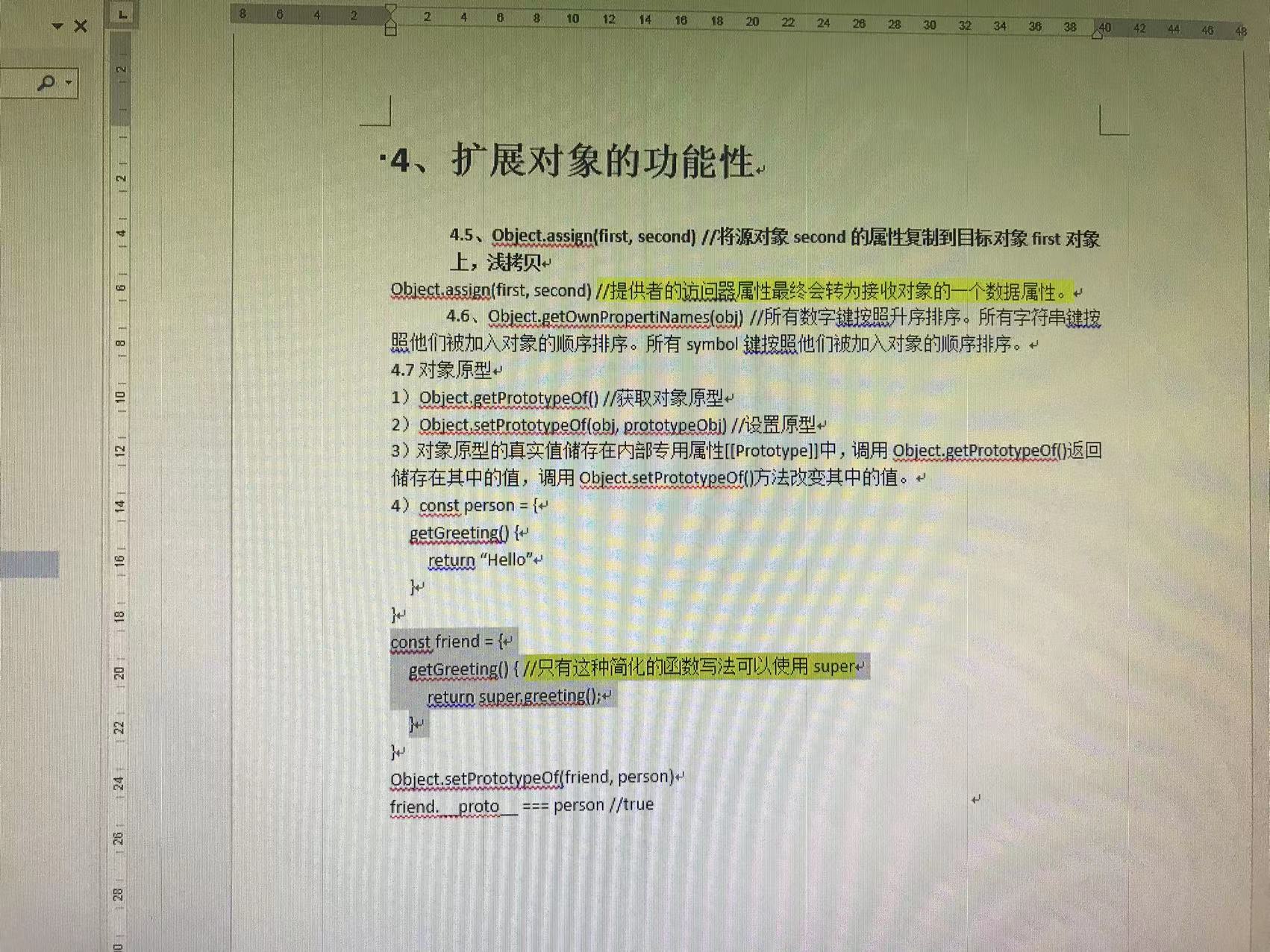
let person = {

[lastName + “One”]: “Nicholas”

}

**4.4新增方法**

Object.is() 类似于 === 但有两个区别：Object.is(NaN, NaN);//true Object.is(+0, -0) //false



5)super:super引用是固定的，比如在4）的基础上增加代码

let relative = Object.create(friend); console.log(relative.getGreeting()) //“Hello”

# 5、解构：使数据访问更便捷

**5.1对象解构**

1)let node = {

name: “foo”,

type: “Function”

}

let {name, type, value} = node; //右侧必须有值，并且不能是undefined,null;

//value的值为undefiend;

//可以let {name, type, value = true} = node;

**2)非同名对局部对象解构**

let {type:localType, name: localName} = node

//语法含义：读取名为type的属性，并将其值存储在变量localType中。

**5.2数组解构**

**1）let colors = [“red”, “green”, “blue”];**

let [firstColor] = colors;

let [ , , thirdColor] = colors;

**2）解构赋值**：不需要和对象那样，需要()

let firstColor = “yellow”;

[firstColor] = colors;

**3）默认值**

let colors = [“red”]

let [firstColor, secondColor=”green”] = colors;

**4）不定元素:必须为最后一个元素**

let [firstColor, ...restColors] = [“red”, “green”, “blue”];

console.log(restColors.length) //2

**5）数组赋值**

let [...clonedColors] = [“red”, “green”] //console.log(clonedColors)

**5.3解构参数**

# 6、Symbol和Symbol属性

**6.1使用：**

1）let firstName = Symbol(“first name”)

let person = {};

person[firstName] = ‘Nicholas’

2）let firstName = Symbol(“first name”)

let person = {

[firstName]: “Nicholas”

}

Object.defineProperty(person, firstName, {writeable: false})

3）let lastName = Symbol(“last name”);

Object.defineProperties(person, {

[lastname]: {

value: “Zakas”,

writeable: false

}

})

**6.2Symbol共享**

1）let uid1 = Symbol.for(“uid”); //没有就直接全局注册

let uid2 = Symbol.for(“uid”); //存在就直接读取

let obj = {[uid1]: “12345”}

console.log(obj[uid2]) //”12345”

console.log(Object.is(uid2, uid1)) //true

console.log(uid2) //”Symbol(uid)”

2）Symbol.keyFor(uid1) //”uid” //检索与Symbol有关的键

Symbol.keyFor(uid2) //”uid”

let uid3 = Symbol(“uid”)

Symbol.keyFor(uid3) // undefined

3)let uid = Symbol.for(“uid”);

let desc = String(uid); //实际上调用了uid.toString()方法。

console.log(desc) //”Symbol(uid)”

//必须手动将Symbol引用转换成对象，否则报错，例如let desc = uid + “”

4）Object.keys() //返回对象的可枚举属性

Object.getOwnPropetyNames() //不考虑属性的可枚举性一律返回。

Object.getOwnPropertSymbols() //返回包含所有Symbol自有属性的数组。

**6.3通过well-known Symbol暴露内部操作**

**1） Symbol.hasInstance方法**，可以获取和修改

function MyObject() { //空函数

}

Object.defineProperty(MyObject, Symbol.hasInstance, { //不可写属性只有通////过Objectd.defineProperty属性进行修改

value: function(v) {

return false;

}

})

console.log(new MyObject() instanceof MyObject) //false

//上面这句相当于console.log(MyObject[Symbol.hasInstance](new MyObject))

**2） Symbol.isConcatSpreadable属性**

let obj = {

"0": "hello",

"1": "world",

length:2,

[Symbol.isConcatSpreadable]: true //如果为true表示对象有length属性和数字键，其数值型属性值应该被独立添加到concat()调用的结果中。

}

let message = ["hi"].concat(obj)

console.log(message)

**3）Symbol.match(),Symbol.replace(regex, replacement),Symbol.search() 和Symbol.split()属性**。表示match(),replace(),search(),split()方法的第一个参数应该调用的正则表达式的方法。

//实际上等于/^.{10}$/

let hasLengthOf10 = {

[Symbol.match]: function(val) {

return val.length === 10 ? [val] : null;

},

[Symbol.replace]: function (val, replacement) {

return val.length === 10 ? replacement : val;

},

[Symbol.search]: function(val) {

return val.length === 10 ? 0 : -1;

},

[Symbol.split]: function(val) {

return val.length === 10 ? [, ] : [val]

}

}

let message1 = "Hello world";

let message2 = "Hello John";

console.log(message1.match(hasLengthOf10)) //null

console.log(message2.match(hasLengthOf10)) //["Hello John"]

console.log(message1.replace(hasLengthOf10)) //Hello world

console.log(message2.replace(hasLengthOf10, message1)) //Hello world

console.log(message1.search(hasLengthOf10)) //-1

console.log(message2.search(hasLengthOf10)) //0

console.log(message1.split(hasLengthOf10)) //["Hello world"]

console.log(message2.split(hasLengthOf10)) //["", ""]

**4）Symbol.toPrimitive()方法,转换为原始值**

function Temperature(degrees) {

this.degrees = degrees;

}

Temperature.prototype[Symbol.toPrimitive] = function(val) {

switch(val) { //String(freezing)时触发

case "string":

return this.degrees + "\u00b0"; //degrees symbol;

case "number": // /触发

return this.degrees;

case "default": // + 触发

return this.degrees + "degrees"

}

}

const freezing = new Temperature(30);

console.log(freezing + "!") //30 degrees

console.log(freezing / 2); //15

console.log(String(freezing)) //30▫

**5）Symbol.toStringTag属性**

function Person(name) {

this.name = name;

}

Person.prototype[Symbol.toStringTag] = "Person";

var me = new Person("Nichols");

console.log(me.toString()) //"[object Person]"

console.log(Object.prototype.toString.call(me)) //"[object Person]"

//在Person.prototype上定义了一个Symbol.toStringTag属性作为创建字符串表示名称时的默认值，因为Person.prototype继承了Object.prototype.toString()方法，所以调用me.toString()方法时也使用了Symbol.toStringTag的返回值。