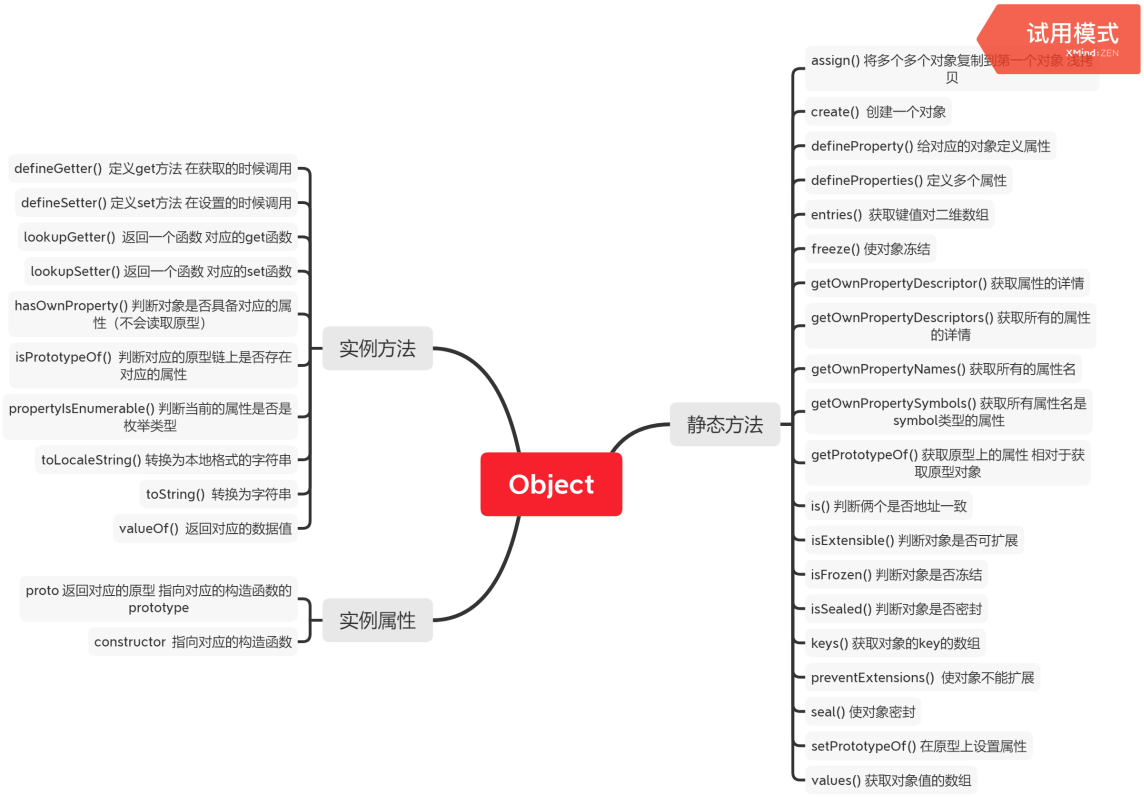
day25 Object的相关方法

概述

Object类是所有类的父类、也就是说所有对象都默认继承Object类，那么对应的Object的方法所有的对象都可以使用。主要学习的**object的相关方法是提供给其他对象使用的。**



Object相关属性及方法

Object的方法主要分为原型方法（实例）和静态方法、相关属性对应的实例拥有的属性（实例属性）

相关属性

- `__proto__` **表示对象的原型指定当前构造函数的原型prototype**

```
console.log({}.__proto__) //指向对应的Object的原型prototype
```

- **constructor 表示对象的构造函数**

```
console.log({}.constructor) //指向Object构造函数
```

原型方法

hasOwnProperty （判断本身是否具备这个属性 不包含原型上的） *

```
//原型添加的属性
Object.prototype.age = 18
```

```
//本身的对象
let obj = {
    name: 'jack'
}
obj.sex = '男'
console.log(obj.age) //18 原型上的属性
console.log(obj.name) //本身的属性
//hasOwnProperty 判断对象本身是否具备属性 不包含原型上的属性的
//传入属性名
console.log(obj.hasOwnProperty('name'))//true
console.log(obj.hasOwnProperty('age')) //false
console.log(obj.hasOwnProperty('sex'))//true
```

isPrototypeof 判断当前对象是否存在原型链上 *

```
class Person{
    constructor(){
        this.name = 'tom'
    }
}
class Child extends Person{
    constructor(){
        super()
        this.age = 19
    }
}
let child = new Child()
let person = new Person()
console.log(child.isPrototypeOf(person))//false  child对象是否存在于person的原型链上
console.log(person.isPrototypeOf(child))//false  person是否存在于child的原型链上
//判断是否存在原型链上 传入的是对应的对象
console.log(Person.prototype.isPrototypeOf(child)) // 对应的Person的原型是否存在于child的原型链上
//将对应的child的原型进行赋值
child.__proto__ = person
//isPrototypeOf 传入的是对应查找原型链的对象
console.log(person.isPrototypeOf(child)) //person对象是否存在于child的原型链上
```

propertyIsEnumerable 判断属性是否枚举（是否可以被for in遍历）*

```
//判断是否可以被for in遍历（是否枚举）propertyIsEnumerable
class Animal{
    constructor(){
        this.name = 'dog'
        this.run = ()=>{}
    }
    //不能被枚举的
    eat(){

    }
}
let animal = new Animal()
animal.age = 18 //默认的赋值对应的属性可以被枚举
for(var key in animal){
    console.log(key) //name run age
}
```

```
//检索是否可以枚举 （for in只能遍历可以枚举的属性）
console.log(animal.propertyIsEnumerable('run')) //true
console.log(animal.propertyIsEnumerable('name')) //true
console.log(animal.propertyIsEnumerable('age')) //true
console.log(animal.propertyIsEnumerable('eat')) //false
```

toString 将对象转为字符串、toLocalString 将对象转为本地格式字符串

valueOf 得到本身的值

废弃的四个方法

- `__defineGetter__` **定义getter函数**
- `__defineSetter__` **定义setter函数**
- `__lookUpGetter__` **返回getter函数**
- `__lookUpSetter__` **返回setter函数**

```
//废弃的四个方法
let obj1 = {
    name:'jack',
    age:18
}
let _obj =  {
    name:'jack',
    age:18
}
//__defineGetter__ __defineSetter__ 定义getter和setter方法
//传递的属性名 和对应的处理函数
obj1.__defineGetter__('name',()=>{
    console.log('执行了')
    return _obj.name //getter不允许调用自身
})
obj1.__defineSetter__('age',(value)=>{
    _obj.age = value
    // obj1.age = value 报错 栈溢出
})
console.log(obj1.name) //访问getter 获取getter中的返回值
obj1.age = 30 //调用setter方法
//__lookupGetter__ __lookupSetter__ 返回对应的getter方法和setter方法
console.log(obj1.__lookupGetter__('name'))
console.log(obj1.__lookupSetter__('age'))
```

静态方法

Object.assign 将传入的对象的内容填入第一个传入的对象内容内 返回的是第一个对象 *

```
//里面可以传入任意对象 将对应的后面的参数的内容填入到第一个对象中 返回第一个对象
let first = {
    name: 'jack',
    likes:['苹果']
}
let object = Object.assign(first, {
```

```
    age: 18
}, {
    sex: '男'
})
console.log(object) //{name:'jack',age:18,sex:'男'}
console.log(first == object) //true
console.log(first)
//object.assign 可以完成对象的浅拷贝（拷贝第一层值 深层拷贝地址 产生新的对象）
let copyObj = Object.assign({},first)
console.log(copyObj == first) //false
//第二层拷贝地址 copyObj的likes对应的地址 和  first的likes的地址是一样的
console.log(copyObj.likes == first.likes)//true
```

Object.create 创建一个对象 *

```
//根据传入的内容来创建对应的内容
let first = {
    name: 'jack'
}
//create它是将对应的传入的对象放入创建对象的原型上
let newObj = Object.create(first) //根据传入的first来创建新的对象
console.log(newObj)
let obj = create(first)
console.log(obj)
//简单实现一下Object的create方法
function create(obj){
    //创建一个新的对象
    let newObj = {}
    //将传入的对象放入新的对象的原型上
    newObj.__proto__ = obj
    //返回这个新的对象
    return newObj
}
```

Object.keys 返回对象所有的key组成的数组

Object.values 返回对象所有的value组成的数组

Object.entries 返回对象所有的键值对组成的数组

```
let obj = {
    name: 'hello',
    age: '17'
}
class Person {
    constructor() {

    }
    //不可枚举的
    eat() {

    }
}
//原型上的属性
Person.prototype.age = 18
//返回都是数组
```

```
console.log(Object.keys(obj)) //所有key的数组
console.log(Object.values(obj)) //value的数组
console.log(Object.entries(obj)) //所有的键值对 组成的二维数组
//Object.keys不包含不可枚举的属性 也不包含原型上的属性
let person = new Person()
console.log(Object.keys(person))
//for in不包含不可枚举的属性 包含原型上的属性
for(var key in person){
    console.log(key)
}
```

Object.is 判断俩个对象是否一致

```
let first = {}
let last = {}
console.log(Object.is(first,last))//false
```

Object.getPrototypeOf 获取原型

Object.setPrototypeOf 设置原型

```
function Person(){}
Person.prototype.age = 19
let person = new Person()
//获取原型 getPrototypeOf
console.log(Object.getPrototypeOf(person) )//获取person的原型对象 相当于获取
__proto__
//设置原型 相当于给__proto__进行赋值 setPrototypeOf
let obj = {name:'jack'}
Object.setPrototypeOf(person,obj) //将obj做为perosn的原型
console.log(obj.isPrototypeOf(person) ) //true
console.log(person)
```

对象相关操作限制的方法 *

- 不可扩展 不能进行内容添加 **preventExtensions** （判断是否可扩展 **isExtensible**）
- 密封 只能查询和修改 其他操作不允许 **seal** （判断是否密封 **isSealed** ）
- 冻结 只能查询 **freeze** （**isFrozen** 判断是否冻结）

```
//查询都能做的 不然的话对应的对象创建就没有意义
let obj = {
    name: 'jack'
}
//不可扩展 返回当前的对象
Object.preventExtensions(obj)
//检测是否可以扩展
console.log(Object.isExtensible(obj)) //false
//添加新的属性
obj.age = 18 //无效
obj.name = 'tom' //改
console.log('name' in obj) //查询
delete obj.name //删除
console.log(obj)
//密封 修改和查询其他操作都不允许
let obj1 = {
```

```
    name: 'jack'
}
//使用对象密封
Object.seal(obj1)
//密封会导致不可扩展
console.log(Object.isExtensible(obj1)) //false
//判断是否密封
console.log(Object.isSealed(obj1)) //true
obj1.age = 18 //无效
obj1.name = 'tom' //改
console.log('name' in obj1) //查询
delete obj1.name //无效
console.log(obj1)
//冻结 只能查询
let obj2 = {
    name: 'jack'
}
//冻结对象 只能执行查询操作
Object.freeze(obj2)
//冻结会导致密封
console.log(Object.isSealed(obj2)) //true
//判断是否冻结
console.log(Object.isFrozen(obj2))//true
obj2.age = 18 //无效
obj2.name = 'tom' //无效
console.log('name' in obj2) //查询
delete obj2.name //无效
console.log(obj2)
```

冻结 必定密封和不可扩展 密封 必定不可扩展

属性相关的获取方法 *

- **Object.getOwnPropertyNames** （获取对象上的所有属性名 （不包含symbol修饰的 不包含原型上的））
- **Object.getOwnPropertySymbols** （获取对象上名字为symbol的属性名 不包含原型上的）
- **Object.getOwnPropertyDescriptor** （获取对应的属性的详情对象 （不包含原型上的））
- **Object.getOwnPropertyDescriptors** （获取所有的属性的详情对象 （不包含原型上的））

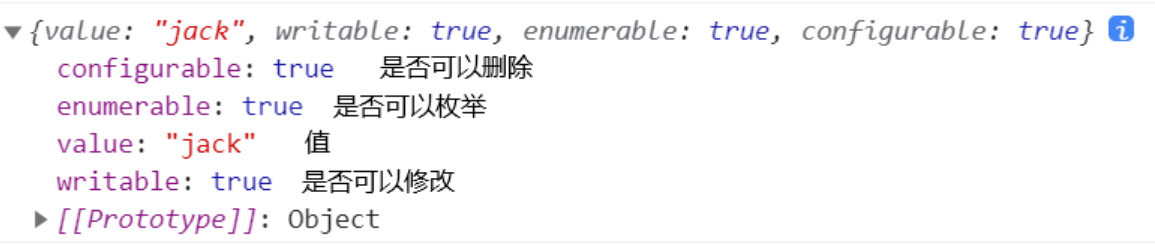
```
function Person(){
    this.name = 'jack'
}
//原型上的属性
Person.prototype.age = 18
Person.prototype[Symbol('原型上的')] = '你好'
let person = new Person()
//symbol修饰的属性
person[Symbol('对象中的')] = 'hello'
person.sex = '男'
//获取所有的属性名 传入对象 返回的是对应的数组
//getOwnPropertyNames 不会获取原型的属性 不会获取symbol为key的属性
let names= Object.getOwnPropertyNames(person)
console.log(names)
//获取所有的symbol修饰的属性 不会获取原型上的
let symbols = Object.getOwnPropertySymbols(person)
console.log(symbols);
//获取属性的详情对象
```

```
//获取某个属性的详情对象 传入对应的对象及属性
let descriptor = Object.getOwnPropertyDescriptor(person,'name')
console.log(descriptor);
//获取所有属性的详情对象（对象） (不包含原型上的属性 包含symbol修饰的属性)
let descriptors = Object.getOwnPropertyDescriptors(person)
console.log(descriptors)
```

属性对象 (descriptor 对象)

属性属性

```
configable 是否可以删除
enumerable 是否可以枚举
value 值
writable 是否可以修改
```



访问器属性

```
configable 是否可以删除
enumerable 是否可以枚举
get getter方法 访问的时候调用
set setter方法 设置的时候调用
```

Object.defineProperty 用于定义对象的属性 (vue2的底层实现) *

实现数据驱动 (双向数据绑定)

```
let obj = {}
//传入对象 传入属性 传入属性对象
Object.defineProperty(obj, 'name', {
    configurable: false,
    enumerable: false,
    value: 'tom',
    writable: false
})
console.log(obj) // {name:tom}
obj.name = 'rose' //无效 writable为false
for (var key in obj) {
    console.log(key) //name也不会打印 enumerable为false
}
delete obj.name //无效 configurable为false
console.log(obj)
//传入访问器属性对象 进行属性定义
let _obj = {} //借助一个新的对象来进行操作
Object.defineProperty(obj,'age',{
    configurable:true,
    enumerable:true,
    get(){
```

```
        console.log('访问了get')
        return _obj.age
        // return obj.age 会造栈溢出
    },
    set(value){
        console.log('访问了set')
        _obj.age = value
    }
})
obj.age = 18 //访问set
console.log(obj.age)//访问get
```

Object.defineProperties 用于定义对象的多个属性 *

```
let obj = {}
//传入对象 及对应的属性详情对象
let _obj = {}
Object.defineProperties(obj,{
    name:{
        configurable:true,
        enumerable:true,
        value:'tom',
        writable:true
    },
    age:{
        configurable:true,
        enumerable:true,
        get(){
            return _obj.age
        },
        set(value){
            _obj.age = value
        }
    }
})
console.log(obj)
```

面试题

查看属性的相关方法

- for in 只能遍历可枚举的属性 （可以遍历原型上的 不可以遍历symbol）
- Object.keys 只能遍历本身的 （不可以遍历原型上的 也不可以遍历不可枚举的 symbol）
- Object.getOwnPropertyNames （可以遍历不可枚举的 不可以遍历原型上的 及 symbol）