深浅拷贝

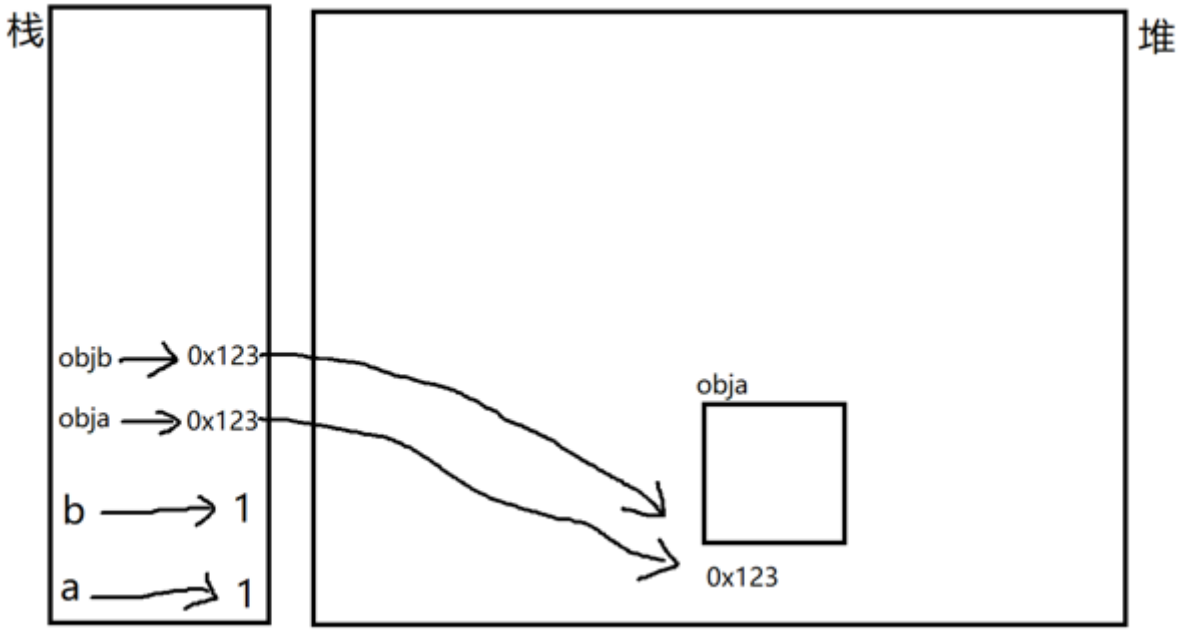
赋值、深拷贝和浅拷贝

赋值：

1. 基本类型赋值：赋值后两个变量互不影响
2. 引用类型赋值：赋值后两个变量具有相同的引用，指向同一个对象，互相影响。

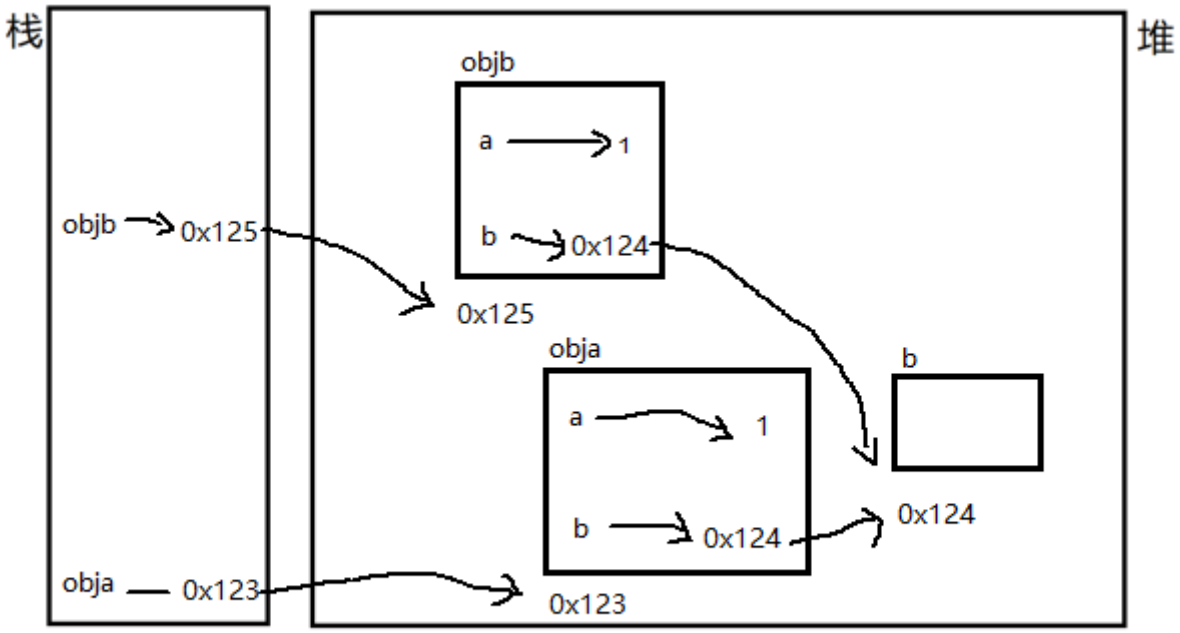
```
var a = 1;
var b = a;

var obja = {}
var objb = obja;
```



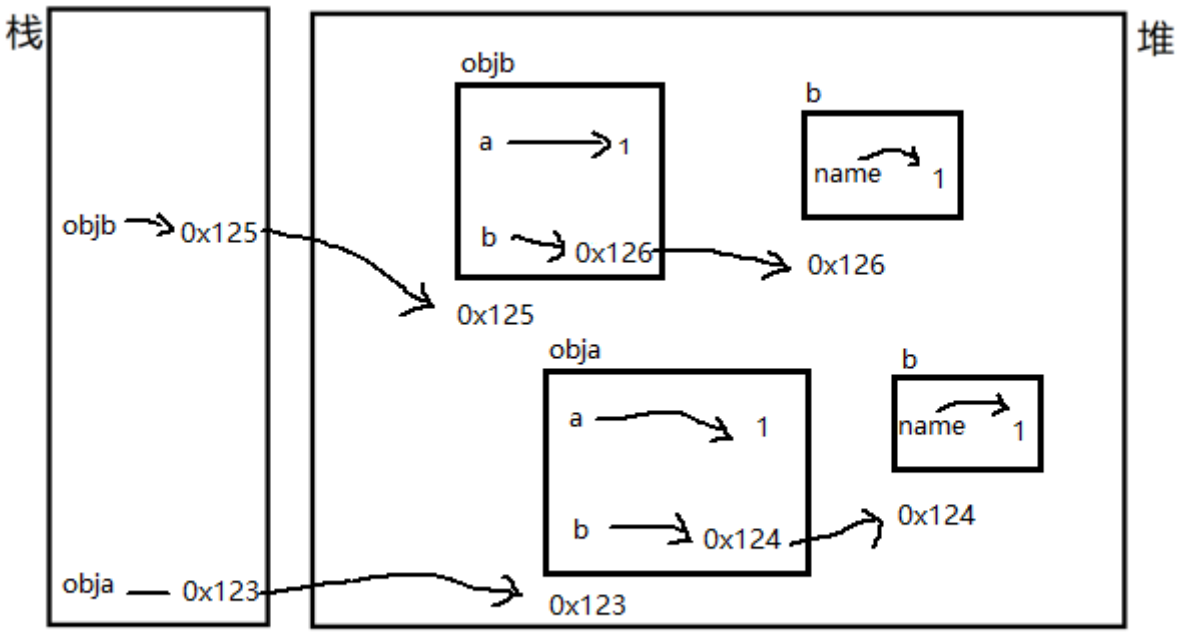
浅拷贝： 浅拷贝基本类型和赋值一样，浅拷贝引用类型的时候创建了一个新的对象，是对原始对象属性值的一份拷贝。如果属性是基本类型就拷贝类型值，如果是引用类型就拷贝地址值。

```
var obja = {
  a: 1
  b: {}
}
//objb浅拷贝obja
```



深拷贝： 深拷贝会拷贝所有的属性，并且拷贝属性指向的动态分配的内存。

```
var obja = {
  a: 1
  b: {
    name: '1'
  }
}
//objb深拷贝obja
```



JSON

> JSON对象中有两个方法，parse()解析JSON，stringify()将对象转换成JSON。

```
const sym = Symbol('sym');
let obj = {
  name: 'names',
  age: 18,
  un: undefined,
  nu: null,
  fn: () => {
    console.log('fn');
  },
  [sym]: 'sym'
}

let obj2 = obj;
let obj3 = JSON.parse(JSON.stringify(obj));

console.log('obj', obj, 'obj2', obj2, 'obj3', obj3);
```

我们通过图片观察到新对象中的un、fn、Symbol('sym')属性丢失。所以不管JSON是不是深拷贝他的缺点就是不会对undefined、Symbol、function拷贝。

然后我们来验证JSON是深拷贝还是浅拷贝。

```
const sym = Symbol('sym');
let obj = {
  name: 'names',
  age: 18,
  un: undefined,
  nu: null,
  fn: () => {
    console.log('fn');
  },
  [sym]: 'sym'
}
```

```
obj.age = 19;

let obj2 = obj;
let obj3 = JSON.parse(JSON.stringify(obj));

console.log('obj', obj, 'obj2', obj2, 'obj3', obj3);
```

我们加了一行`obj.age = 19;`

我们可以看到，原对象和赋值的对象中的age属性都发生了改变，而拷贝的obj3中的age属性没有发生改变，那么我们可以说JSON是深拷贝吗？显然不行，因为浅拷贝是对对象中的属性进行拷贝，如果属性是基本类型就拷贝类型值，如果是引用类型就拷贝地址值。 我们应该给obj添加一个obj属性，值是`{age: 18}`。

```
const sym = Symbol('sym');
let obj = {
  name: 'names',
  age: 18,
  un: undefined,
  nu: null,
  fn: () => {
    console.log('fn');
  },
  [sym]: 'sym',
  obj: {
    age: 18
  }
}

let obj2 = obj;
let obj3 = JSON.parse(JSON.stringify(obj));

obj.obj.age = 19;

console.log('obj', obj, 'obj2', obj2, 'obj3', obj3);
```

我们通过图片可以看到当原对象obj.obj.age参数改变后，obj3中的obj.obj.age没有发生改变，这就说明JSON是深拷贝。

Array.concat()

> concat()方法用于合并两个或多个数组，此方法不会更改现数组，而是返回一个新数组。

```
let arr = [1, '2', NaN, undefined, null, (function () { console.log('fn') }),{
  age: 18
}];

let arr2 = arr;
let arr3 = [].concat(arr);

arr[6].age = 19;

console.log('arr',arr,'arr2',arr2,'arr3',arr3);
```

我们可以看到，当arr[6].age = 19后，arr3[6].age也跟着改变了，所以说，Array.concat()是浅拷贝。

展开运算符

```
const sym = Symbol('sym');
let obj = {
  name: 'names',
  age: 18,
  un: undefined,
  nu: null,
  fn: () => {
    console.log('fn');
  },
  [sym]: 'sym',
  obj: {
    age: 18
  }
}

let obj2 = obj;
let obj3 = {...obj};

obj.obj.age = 19;

console.log('obj', obj, 'obj2', obj2, 'obj3', obj3);
```

我们通过图片可以看到当原对象obj.obj.age参数改变后，obj3中的obj.obj.age发生改变，这就说明扩展运算符是浅拷贝。

Object.assign()

> Object.assign() 方法将所有可枚举的自有属性从一个或多个源对象复制到目标对象，返回修改后的对象。

```
const sym = Symbol('sym');
let obj = {
  name: 'names',
  age: 18,
  un: undefined,
  nu: null,
  fn: () => {
```

```
    console.log('fn');
  },
  [sym]: 'sym',
  obj: {
    age: 18
  }
}

let obj2 = obj;
let obj3 = Object.assign({},obj);

obj.obj.age = 19;

console.log('obj', obj, 'obj2', obj2, 'obj3', obj3);
```

我们通过图片可以看到当原对象obj.obj.age参数改变后，obj3中的obj.obj.age发生改变，这就说明Object.assign()是浅拷贝。

Array.slice()

> slice() 方法返回一个新的数组对象，这一对象是一个由 begin 和 end 决定的原数组的浅拷贝（包括begin，不包括end）。原始数组不会被改变。

```
let arr = [1, '2', NaN, undefined, null, (function () { console.log('fn') }),{
  age: 18
}];

let arr2 = arr;
let arr3 = arr.slice(0,arr.length);

arr[6].age = 19;

console.log('arr',arr,'arr2',arr2,'arr3',arr3);
```

我们可以看到，当arr[6].age = 19后，arr3[6].age也跟着改变了，所以说，Array.slice()是浅拷贝。

Lodash的_.cloneDeep方法

html

手写深拷贝

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <script>
    //判断target是否是对象、数组、函数
    const isObject = (target) => target !== null && (typeof target === 'object' ||
typeof target === 'function');
    //判断target的具体类型
    const getType = (target) => Object.prototype.toString.call(target);

    const mapTag = '[object Map]'
    const setTag = '[object Set]'

    const boolTag = '[object Boolean]'
    const stringTag = '[object String]'
    const numberTag = '[object Number]'
    const dateTag = '[object Date]'
    const funTag = '[object Function]'
    const regexp = '[object RegExp]'
    const errorTag = '[object Error]'

    const canTraverse = {
      '[object Object]': true,
```

```
    '[object Map]': true,
    '[object Set]': true,
    '[object Array]': true,
    '[object Arguments]': true,
  }

  function handleNotTraverse(target, type) {
    const ctor = target.constructor;
    switch (type) {
      case boolTag:
        return new Boolean(Boolean.prototype.valueOf.call(target));
      case numberTag:
        return new Number(Number.prototype.valueOf.call(target));
      case stringTag:
        return new String(String.prototype.valueOf.call(target));
      case dateTag:
      case errorTag:
        return new ctor(target);
      case funTag:
        return handleFun(target);
      case regexp:
        return handleRexExp(target);
      default:
        return new ctor(target);
    }
  }

  function handleFun(target) {
    //箭头函数直接返回自身
    if (!target.prototype) return target;
    target = target.toString();
    const bodyReg = /(?<={)(.|\n)+(?=})/m
    const paramReg = /(?<=\().+(?=\)(\s*)+{)/
    const param = paramReg.exec(target);
    const body = bodyReg.exec(target);
    console.log('fun',target,'param',param);
    if(!body)return null;
    if(param){
      const paramsArr = param[0].split(',');
      return new Function(...paramsArr,body[0]);
    }else{
      return new Function(body[0]);
    }
  }

  function handleRexExp(target) {
    const { source, flags } = target;
    return new target.constructor(source, flags);
  }

  //深拷贝
  function deepClone(target, map = new WeakMap()) {
    let cloneTarget = null;
    //判断是否是引用类型
```

```
    if (!isObject(target)) return target;
    //获取更精确的类型
    let type = getType(target);
    if (!canTraverse[type]) {
      //不可遍历
      return handleNotTraverse(target, type);
    } else {
      //可遍历
      let ctor = target.constructor;
      cloneTarget = new ctor();
    }
    //判断是否循环引用
    if (map.get(target)) return map.get(target);
    map.set(target, cloneTarget);
    if (type === mapTag) {
      //处理Map
      target.forEach((element, key) => {
        cloneTarget.set(deepClone(key, map), deepClone(element, map));
      });
    }
    if (type === setTag) {
      //处理Set
      target.forEach((element, key) => {
        cloneTarget.add(deepClone(item, map));
      })
    }
    //处理数组和对象
    for (let props in target) {
      if (target.hasOwnProperty(props)) {
        cloneTarget[props] = deepClone(target[props], map);
      }
    }
    return cloneTarget;
  }

  function fnc(a, b) {
    console.log(a + b);
  }

  let obj = {
    name: '1',
    age: {
      age: function fnc(a, b){
        console.log(a+b);
      }
    }
  };


  let obj2 = deepClone(obj);

  console.log('obj',obj,'obj2',obj2);
</script>
</body>
```

```
</html>
```