22 代码重构

本书目的不是讲解重构，但其实目前为止都在进行代码级别的优化，通过反例代码讲解设计模式重构之后的代码

模式与重构是与生俱来的关系，设计模式就是为了重构行为提供目标

实际项目中除了设计模式，还有一些容易忽略细节帮助我们达到重构目的，这里我们介绍一部分

提炼函数

JS中，函数体内最好逻辑清晰明了，过长的话看是否可以独立一部分逻辑：

- 避免出现超大函数
- 独立出来的容易复用
- 独立出来的函数有良好的命名，充当注释

合并重复的条件片段

如果一个函数体内有条件分支语句，并且内部散步了重复的代码，我们是否可以进行合并去重工作来优化。例如如下进行currPage边界保护的跳转：

```
const paging = function(currPage){
    if(currPage <= 0){
        currPage = 0;
        jump(currPage)
    } else if(currPage >= totalPage){
        currPage = totalPage;
        jump(currPage)
    } else {
        jump(currPage)
    }
}
```

这里我们完全可以将重复的jump抽离出来：

```
const paging = function(currPage){
    if(currPage <= 0){
        currPage = 0;
```

```
    } else if(currPage >= totalPage){
        currPage = totalPage;
    }
    jump(currPage)
}
```

复杂的分值语句提炼成函数

这里如有一个规则：如果是夏天，所有商品8折出售，这里我们就可以把判断是夏天的逻辑抽离处来，一方面代码意图明显，另一方面代替了注释：

```
const isSummer = function(){
    const date = new Date();
    return date.getMonth() >= 6 *&& date.getMonth() <= 9
}

const getPrice = function(price){
    if(isSummer){
        return price * 0.8
    }
    return price
}
```

提前让函数推出代替嵌套条件分支

许多程序员都有一个观念，函数只有一个入口和出口，但是只有一个出口其实有不同看法，有时候嵌套的if-else复杂且冗长之后之后会影响可读性，这个时候其实可以挑选一些条件分支，例如把外层的if反转来重构，提前通过边缘case退出函数：

```
const del = function(obj){
    if(obj.isReadOnly){
        return;
    }
    if(obj.isFolder){
        return deleteFolder(obj)
    }
}
```

传递对象参数代替过长的参数列表

有时候函数会接受多个参数，而参数数量越多其实就越难理解和使用，还需要注意参数顺序和位置，尤其是在参数之间新增参数就很麻烦。

这里我们可以把参数放到一个对象内：

```
const setUserInfo = function(obj){
    // obj.xx
}

// 只需要关注key就行
setUserInfo({
    id: 1314,
    name: 'sven'
    //...
})
```

尽量减少参数数量

除了上面方法，有时候其实某个入参是可以通过其他入参算出来的，那么我们就可以在函数里面计算之后直接获取而不需要在外面作为参数传进来，当然如果这个计算耗时很久那还是放在外面公用

少用三目运算符

- 三目运算符性能上比if-else几乎完全一样，同一个级别
- 虽然代码量少，但是代码可读性和维护性都有一些损失
- 尤其是是条件分支逻辑非常复杂，还是按部就班使用if-else好，代码更清晰

合理使用链式调用

jQuery的程序员可能熟悉了链式调用方法，JS中实现也很简单，让方法结束后返回对象自身就可以了。

```
new User().setId(1234).setName('sven')
```

但是带来的坏处就是调试不方便，如果链路有错误必须把链条拆开来并加上调试log才可以定位。

如果链路稳定那么没问题，如果不确定建议还是普通调用：

```
const user = new User();
user.setId(1234);
```

```
user.setName('sven');
```

分解大型类

有时候一些类会非常庞大，不仅负责创建对象，来包括对象所有的动作，并且有的动作还非常复杂。其实面向对象设计鼓励将行为分布在粒度合适的小对象之中：

```
class Person {
    constructor(name){
        this.name = name;
        this.attackObj = new Attack(this);
    }

    attack(type){
        this.attackObj.start(type);
    }
}

class Attack {//...
}


```

用return退出多重循环

假设函数体内有一个两重循环语句，我们需要在内层循环判断达到某个临界条件时退出外层的循环。这里有的时候会通过一个flag变量来实现：

```
for(let i = 0; i < 10; i++){
    for(let j = 0; j < 10;j++){
        if(i*j>30){
            flag= true;
            break
        }
    }
    if(flag===true){
        break
    }
}
```

这里我们其实可以通过return直接退出方法：

```
for(let i = 0; i < 10; i++){
    for(let j = 0; j < 10;j++){
        if(i*j>30){
        // do...
            return;
        }
    }
}
```