16 状态模式

状态模式是一种虽然不是简单到一目了然甚至增加代码量的模式，但是一旦你明白了精髓，以后一定会感谢它带给你的好处。

关键是区分事物内部的状态，内部状态的改变往往会带来事物的行为改变

初识状态模式

有这样一个场景，开关控制电灯，然后通过点击开关来切换电灯状态

第一个例子：电灯程序

```
/**
 * 电灯程序
 */
// 一般写法-简单状态机
class Light{
    constructor(){
        this.button = null;
        this.state = 'off'
    }

    init(){
        const button = document.createElement('button');
        const self = this;
        button.innerHTML = '开关';
        this.button = document.body.appendChild(button);
        this.button.onclick = function(){
            self.buttonWasPressed();
        }
    }

    buttonWasPressed(){
        if(this.state === 'off'){
            this.state = 'on';
            console.log('开灯')
        } else if(this.state === 'on') {
            this.state = 'off';
            console.log('关灯');
        }
    }
```

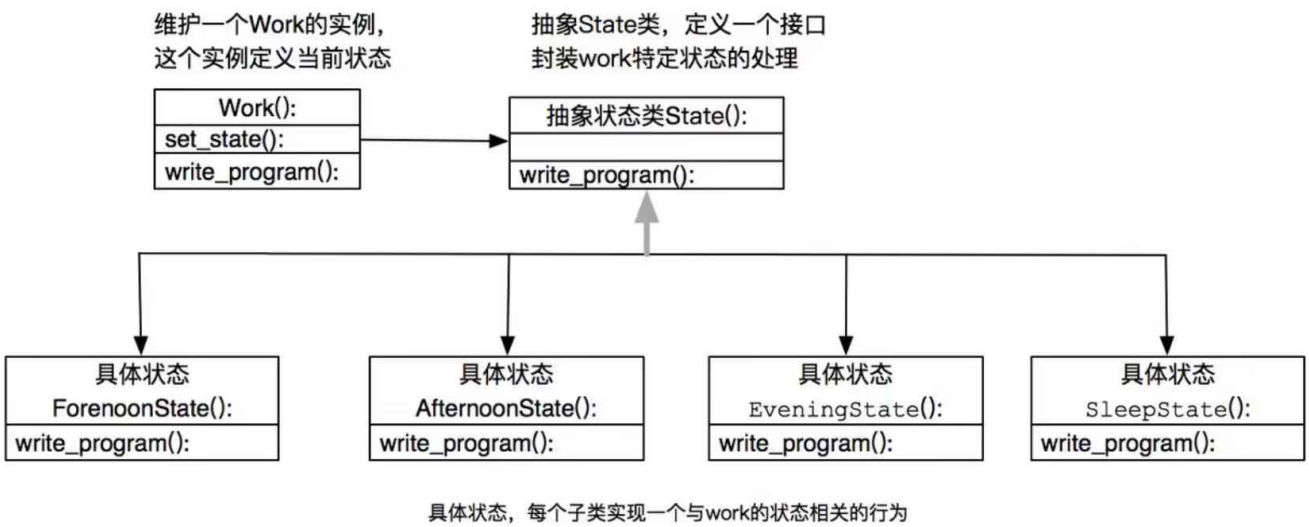
```
30
31 }
```

我们编写了一个状态机，逻辑简单且缜密，但是世界上的电灯并非只有一种，例如有一种调节灯，第一次打开弱光，第二次打开强光，第三次关灯，那我们必须改动buttonWasPressed函数才能实现了。我们总结下上述程序的缺点：

- buttonWasPressed函数违反开闭原则，每次新增light的状态这里都要改动
- buttonWasPressed封装了所有和状态有关的行为，未来可能变得非常巨大，尤其是内部业务逻辑复杂时
- 状态的切换操作不明显，仅仅为对state变量赋值，实际开发中必须读完buttonWasPressed的逻辑代码才能注意
- 状态之间的切换变成了函数中if-else逻辑的维护，难以阅读和维护

状态模式改进程序

一般我们谈到对象封装，都会优先封装对象的行为而非状态，但在状态模式则相反，关键是把状态本身封装为单独的类，跟此状态有关的行为封装到对应的类中，当button被按下时只需要把请求委托给具体的状态类来处理就好，它们负责渲染自己。同时可以把状态的切换规则分布在状态类中，消除大量条件分支语句



具体状态，每个子类实现一个与work的状态相关的行为

我们看一下根据状态模式改造之后的样子：

```
1 // 状态模式改造
2 class OffLightState {
3     constructor(light){
4         this.light;
5     }
```

```
6
7       buttonWasPressed(){
8           // off状态的逻辑
9           console.log('弱光');
10          this.light.setState(this.light.weakLightState);
11      }
12  }
13
14  class WeakLightState {
15      constructor(light){
16          this.light;
17      }
18
19      buttonWasPressed(){
20          console.log('强光');
21          this.light.setState(this.light.strongLightState);
22      }
23  }
24
25  class StrongLightState {
26      constructor(light){
27          this.light;
28      }
29
30      buttonWasPressed(){
31          console.log('关闭');
32          this.light.setState(this.light.offLightState);
33      }
34  }
35
36  // 改写Light类
37  class Light{
38      constructor(){
39          this.button = null;
40          this.offLightState = new OffLightState(this);
41          this.weakLightState = new WeakLightState(this);
42          this.strongLightState = new StrongLightState(this);
43      }
44
45      init(){
46          const button = document.createElement('button');
47          const self = this;
48          button.innerHTML = '开关';
49          // 不再通过字符串来记录state，而是通过具体的状态对象
50          this.currState = this.offLightState;
51          this.button = document.body.appendChild(button);
52          this.button.onclick = function(){
```

```
            // 委托给具体状态对象执行
            self.currState.buttonWasPressed();
        }
    }

    setState(state){
        // 提供状态切换方法，状态切换逻辑在具体状态类中，Context不需要关心
        this.currState = state;
    }

}
```

有状态相关的逻辑变化时我们只需要修改对象的状态类或者新增状态类就可以了，现在每一种状态以及行为之间的关系都分散在各个状态类了，很容易阅读和管理代码。也不需要通过if-else条件来控制状态扭转。

如果需要位light对象新增一种状态也很方便，只需要新增一个superStrongLightState对象，改变light的init以及strongLight的状态切换规则

状态模式的定义

Gof中的定义：允许一个对象在其内部状态改变时改变他的行为，对象看起来似乎修改了它的类

- 将状态封装为独立的类，将请求委托给状态对象
- 客户角度来看，使用的对象会在不同状态下有不同的行为，这个对象看起来是从不同的类中实例化而来的

缺少抽象类的变通方式

我们注意到状态类中会定义诸如buttonWasPressed的共同行为方法，在Java中可以通过抽象类来保证子类都拥有该方法。但是JS不支持抽象类也没有接口（现在可以通过TS来约束了）

这里我们可以在父类中顶buttonWasPressed方法并在其中throw error，这样子类如果不重写那么程序就无法执行：

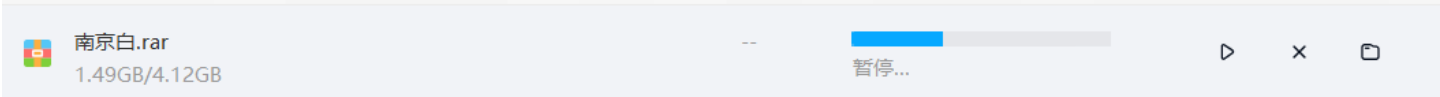
```
// 抽象父类代替
class State {
    buttonWasPressed(){
        throw new Error('父类必须被重写')
    }
}

class SuperLightState extends State {
    buttonWasPressed(){
```

```
10          // TODO
11      }
12  }
```

另一个状态模式示例-文件上传

这是一个真实项目，重构上传模块的经理。实际上无论文件上传，还是音乐视频播放器，都有一些明显状态区分。例如文件上传就有扫描，上传中，暂停，上传成功，上传失败等。此时点击同一个按钮在不同状态下行为表现也是不一样的，包括UI也可能不同（> || ）



更复杂的切换条件

之前的电灯案例其实状态切换逻辑简单，使用if-else也不至于迷失，都是有规律可循。

例如文件上传有两个按钮，分别用于暂停/继续上传 与 删除。

- 扫描状态下，不能暂停也不能删除
- 扫描完成之后根据md5判断文件是否已经在服务器，是则进入上传完成状态。如果文件大小超过允许最大值或损坏，则进入上传失败状态。其他情况进入上传中
- 上传中可以暂停
- 暂停时可以继续
- 暂停，上传完成，上传失败才可以删除

准备工作

微云时利用了浏览器插件来帮助完成文件上传，因为上传是异步过程，他会不停调用window.external.upload这个我们提供了JS函数来通知上传进度。

这里我们通过setTimeout来模拟文件上传进度，upload函数只负责打印简单log，同时我们在页面里面放置一个上传的插件对象：

```
1  /**
2   * 文件上传示例
3   */
4  window.external.upload = (state) => {
```

```
    console.log(state);
}

const plugin = (function () {
    const plugin = document.createElement('embed');
    plugin.style.display = 'none';
    plugin.type = 'application/txftn-webkit';
    plugin.sign = () => { console.log('开始文件扫描') };
    plugin.pause = () => { console.log('暂停文件上传') };
    plugin.uploading = () => { console.log('开始文件上传') };
    plugin.del = () => { console.log('删除文件上传') };
    plugin.done = () => { console.log('文件上传完成') };
    document.body.appendChild(plugin);

    return plugin;
})();
```

开始编写代码 - 初始版

```
// 不使用状态模式
class Upload {
    constructor(fileName) {
        this.plugin = plugin;
        this.fileName = fileName;
        this.button1 = null;
        this.button2 = null;
        this.state = 'sign'; // 初始状态
    }

    // 初始化工作-创建button节点
    init() {
        let that = this;
        this.dom = document.createElement('div');
        this.dom.innerHTML = '<span>文件名称:' + this.fileName + '</span>' + '<bu
        document.body.appendChild(this.dom);
        this.button1 = this.dom.querySelector('[data-action="button1"]');
        this.button2 = this.dom.querySelector('[data-action="button2"]');
        this.bindEvent();
    }

    // 按钮绑定事件 - 处理状态逻辑
    bindEvent() {
        let that = this;
        this.button1.onclick = function () {
            if (self.state === 'sign') {
```

```
27                 console.log('扫描中，点击无效')
28             } else if (self.state === 'uploading') {
29                 self.changeState('pause');
30             } else if (self.state === 'pause') {
31                 self.changeState('uploading');
32             } else if (self.state === 'done') {
33                 console.log('文件以及上传成功，点击无效')
34             } else if (self.state === 'error') {
35                 console.log('文件上传失败，点击无效');
36             }
37         }
38 
39         this.button2.onclick = function () {
40             if (self.state === 'done' || self.state === 'error' || self.state ==
41                 self.changeState('del');
42             } else if (self.state === 'sign') {
43                 console.log('扫描中，不能删除')
44             } else if (self.state === 'uploading') {
45                 console.log('上传中，不能删除')
46             }
47         }
48     }
49 
50     // 切换状态之后的具体行为，与按钮UI逻辑控制，调用插件执行真正的操作
51     changeState(state) {
52         switch (state) {
53             case 'sign':
54                 this.plugin.sign();
55                 this.button1.inner = '扫描中，任何操作无效';
56                 break;
57             case 'uploading':
58                 this.plugin.uploading();
59                 this.button1.inner = '正在上传点击暂停';
60                 break;
61             case 'pause':
62                 this.plugin.pause();
63                 this.button1.inner = '已暂停，点击继续上传';
64                 break;
65             case 'done':
66                 this.plugin.done();
67                 this.button1.inner = '上传完成';
68                 break;
69             case 'error':
70                 this.button1.inner = '上传失败';
71                 break;
72             case 'uploading':
73                 this.plugin.del();
```

```
                this.dom.parentNode.removeChild(this.dom);
                console.log('删除完成')
                break;
        }
    }
}

// 测试下
const uploadObj = new Upload('JS设计模式文件');
uploadObj.init();

window.external.upload = (state){
    // 插件调用JS方法
    uploadObj.changeState(state);
}

window.external.upload('sign');
setTimeout(() => {
    uploadObj.changeState('uploading')
}, 1000)
setTimeout(() => {
    uploadObj.changeState('done')
}, 5000)
```

这里是一个反例，充斥着if-else逻辑，状态和行为耦合严重，很难扩展这个状态机，往后状态复杂了更严重

状态模式重构

第一部分准备的代码没有变化，就是upload和plugin那部分，其他具体看下面代码：

```

/**
 * 状态模式重构文件上传
 */
 class Upload {
    constructor(fileName) {
        this.plugin = plugin;
        this.fileName = fileName;
        this.button1 = null;
        this.button2 = null;

        this.signState = new SignState(this);
        this.uploadingState = new UploadingState(this);
        this.pauseState = new PauseState(this);
```

```
        this.doneState = new DoneState(this);
        this.errorState = new ErrorState(this);
        // 设置初始状态
        this.currState = this.signState;
    }

    // 不用改变，仍然负责创建DOM与绑定事件
    init() {
        let that = this;
        this.dom = document.createElement('div');
        this.dom.innerHTML = '<span>文件名称:' + this.fileName + '</span>' + '<bu
        document.body.appendChild(this.dom);
        this.button1 = this.dom.querySelector('[data-action="button1"]');
        this.button2 = this.dom.querySelector('[data-action="button2"]');
        this.bindEvent();
    }

    // 按钮绑定事件 - 不再自己处理而是委托给状态类
    bindEvent() {
        let self = this;
        this.button1.onclick = function () {
            self.currState.clickHandler1();
        }

        this.button2.onclick = function () {
            self.currState.clickHandler2();
        }
    }

    // 把状态对应的逻辑行为放在这里
    sign(){
        this.plugin.sign();
        this.currState = this.signState;
    }

    uploading(){
        this.plugin.uploading();
        this.button1.inner = '正在上传点击暂停';
        this.currState = this.uploadingState;
    }

    pause(){
        this.plugin.pause();
        this.button1.inner = '已暂停，点击继续上传';
        this.currState = this.pauseState;
    }

```

```
    done(){
        this.plugin.done();
        this.button1.inner = '上传完成';
        this.currState = this.doneState;
    }

    error(){
        this.button1.inner = '上传失败';
        this.currState = this.errorState;
    }

    del(){
        this.plugin.del();
        this.dom.parentNode.removeChild(this.dom);
        console.log('删除完成')
    }
}

// 编写状态类实现

const StateFactory = (function(){
    class State {
        constructor() { }
        clickHandler1() {
            throw new Error('handler1是必要的');
        }
        clickHandler2() {
            throw new Error('handler2是必要的');
        }
    }

    return function(param){
        class F {
            constructor(uploadObj) {
                this.uploadObj = uploadObj;
            }
        }
        F.prototype = new State();
        for(let i in param){
            F.prototype[i] = param[i]
        };
        return F;
    }
})()

const SignState = StateFactory({
    clickHandler1: function(){
```

```
109         console.log('扫描中，点击无效')
110     },
111     clickHandler2: function(){
112         console.log('扫描中，删除无效')
113     }
114 });
115 
116 const UploadingState = StateFactory({
117     clickHandler1: function(){
118         this.uploadObj.pause();
119     },
120     clickHandler2: function(){
121         console.log('上传中，不能删除')
122     }
123 })
124 
125 // ...
```

状态模式优缺点

优点：

- 封装了状态和行为的关系，通过新增状态类很容易扩展新的状态
- 避免Context无限膨胀，状态切换逻辑和独立状态类维护
- Context中的请求动作与状态类中的封装的行为可以独立变化且互不影响

缺点：

- 定义额外的状态类，增加不少对象
- 逻辑分散，状态机逻辑不太容易看出

状态模式性能优化点

- 如果state对象庞大，建议仅当state对象需要时再创建并随后销毁。否则最好一开始就都创建出来，不销毁
- state对象其实各个Context如果逻辑一致是可以共享的，这也是享元模式的应用场景之一

和策略模式的关系

- 他们很像，都内部封装了算法或者行为，都有上下文，并且将请求委托给封装类
- 但是意图上不同，策略模式的各个策略是平等且平行的，没有任何关系，客户必须熟知策略的作用，主动选择算法。而状态模式的扭转是封装好的，“改变行为”发生再状态模式内部，客户不感知这些。

JS版本的状态机

前面两个都是模拟传统面向对象方式的状态模式实现。

状态模式是状态机的实现方式之一，但在JS这种没有类的语言中，其实状态不一定要从类中创建。另外JS使用委托技术很方便，无需事先让一个对象持有另外一个对象。来看看下面的状态机通过委托给字面量对象来执行：

```
/**
 * JS字面量配置状态
 */
class Light {
    constructor() {
        this.button = null;
        this.state = FSM.off;
    }

    init() {
        const button = document.createElement('button');
        const self = this;
        button.innerHTML = '以关灯';
        this.button = document.body.appendChild(button);
        this.button.onclick = function () {
            // 委托给FSM状态机
            self.currState.buttonWasPressed.call(self);
        }
    }
}

const FSM = {
    off: {
        buttonWasPressed: function(){
            console.log('关灯');
            this.button.innerHTML = '下次按我开灯';
            this.currState = FSM.on;
        }
    },
    on: {
        buttonWasPressed: function(){
            console.log('开灯');
            this.button.innerHTML = '下次按我关灯';
            this.currState = FSM.off;
        }
    }
}
```

表驱动的有限状态机

其实还有另外一种基于表驱动的状态机，通过表来定义状态的转移和行为：

状态转移表			
当前状态->条件	状态A	状态B	状态C
条件X	...	...	...
条件Y	...	状态C	...

GitHub上有一个库可以基于表快速创建FSM（https://github.com/jakesgordon/javascript-state-machine）：

```
/**
 * 表驱动
 */
const fsm = StateMachine.create({
    initial: 'off',
    events: [
        {name: 'buttonWasPressed', from: 'off', to: 'on'},
        {name: 'buttonWasPressed', from: 'on', to: 'off'},
    ],
    callbacks: {
        onbuttonWasPressed: function(event, from, to){
            console.log(arguments);
        }
    },
    error: function(event, from, to, errorCode, errorMessage){
        // 状态切换实现不了
        console.log(arguments);
    }
})

button.onclick = function(){
    fsm.buttonWasPressed();
}
```

实际项目中的其他状态机

实际开发中很多场景可以用状态机模拟，例如下拉菜单再hover动作下有显式，悬浮，隐藏等；TCP连接的请求连接，监听，关闭等；游戏人物的移动，攻击，防御等

尤其在游戏开发中也有很多用途，例如AI的逻辑编写，例如街霸游戏Ryu在走动过程中如果变攻击，就会从走动状态切换到跌倒状态。跌倒状态无法进入攻击和防御状态。

```
/**
 * 游戏动作举例
 */
const GameFSM = {
    walk: {
        attack: function () {
            console.log('攻击')
        },
        defense: function () {
            console.log('防御')
        }
    },
    attack: {
        walk: function () {
            console.log('攻击时不能走动')
        },
        defense: function () {
            console.log('攻击时不能防御')
        }
    }
}
```

总结

实际上通过状态模式可以让很多杂乱无章的代码逻辑变清晰。实际应用中也许被大家低估的模式之一