夏老师的返场内容

- 实现一个Cli工具 - vue自动路由配置
- 实现零编码Restful后台
- 设计模式
  - 观察者模式 - 数据绑定
  - 策略模式 - 数据有效性检查

一、 工具链

```
mkdir vue-auto-router-cli
cd vue-auto-router-cli
npm init -y
npm i commander download-git-repo ora handlebars -s
```

```
# bin/kkb
#!/usr/bin/env node
console.log('cli.....')

# package.json
"bin": {
  "kkb": "./bin/kkb"
},
```

```
npm link

# 删除的情况
ls /usr/local/bin/
rm /usr/local/bin/kkb
```

// 引入commander

kkb文件

```
#!/usr/bin/env node
const program = require('commander')
program.version(require('../package').version, '-v', '--version')
    .command('init <name>', 'init project')
    .command('refresh','refresh routers...')
program.parse(process.argv)
```

kkb-init

```
#!/usr/bin/env node
const program = require('commander')
program
    .action(name => {
            console.log('init ' + name)
    })
program.parse(process.argv)
```

/lib/download.js

```
const {promisify} = require('util')
module.exports.clone = async function(repo,desc) {
    const download = promisify(require('download-git-repo'))
    const ora = require('ora')
    const process = ora(`下载.....${repo}`)
    process.start()
    await download(repo, desc)
    process.succeed()
}
```

kkb-init

```
const {clone} = require('../lib/download')
            console.log('🚀创建项目: ' + name)
            await clone('github:su37josephxia/vue-template',name)
```

kkb-refresh

```
#!/usr/bin/env node

const program = require('commander')
const symbols = require('log-symbols')
const chalk = require('chalk')
// console.log(process.argv)
```

```
program
  .action(() => {
    console.log('refresh .... ')
  })
program.parse(process.argv)

const fs = require('fs')
const handlebars = require('handlebars')

const list =
  fs.readdirSync('./src/views')
  .filter(v => v !== 'Home.vue')
  .map(v => ({
    name: v.replace('.vue', '').toLowerCase(),
    file: v
  }))

compile({
  list
}, './src/router.js', './template/router.js.hbs')

compile({
  list
}, './src/App.vue', './template/App.vue.hbs')

function compile(meta, filePath, templatePath) {
  if (fs.existsSync(templatePath)) {
    const content = fs.readFileSync(templatePath).toString();
    const result = handlebars.compile(content)(meta);
    fs.writeFileSync(filePath, result);
  }
  console.log(symbols.success, chalk.green(`🚀${filePath} 创建成功`))
}
```

- 发布npm

```
#!/usr/bin/env bash
npm config get registry # 检查仓库镜像库
npm config set registry=http://registry.npmjs.org
echo '请进行登录相关操作: '
npm login # 登陆
echo "-------publishing-------"
npm publish # 发布
npm config set registry=https://registry.npm.taobao.org # 设置为淘宝镜像
echo "发布完成"
exit
```

二、零代码生成Restful接口

- conf.js

```
module.exports = {
    db: {
        url:"mongodb://localhost:27017/test",
        options: { useNewUrlParser: true }
    }
}
```

- model/user.js

```
module.exports = {
    schema: {
        mobile: { type: String, required: true },
        realName: { type: String, required: true },
    }
}
```

- index.js

```
const Koa = require('koa')
const app = new Koa()

const port = 3000
app.listen(port,() => {
    console.log(`app started at port ${port}...`)
})
```

- // framework/loader.js

```
const fs = require('fs')
const path = require('path')
const mongoose = require('mongoose')

function load(dir, cb) {
    // 获取绝对路径
    const url = path.resolve(__dirname, dir)
    const files = fs.readdirSync(url)
    files.forEach(filename => {
        // 去掉后缀名
        filename = filename.replace('.js', '')
        // 导入文件
        const file = require(url + '/' + filename)
        // 处理逻辑
```

```
            cb(filename, file)
        })
    }

    const loadModel = config => app => {
        mongoose.connect(config.db.url, config.db.options);
        const conn = mongoose.connection
        conn.on("error", () => console.error("连接数据库失败"))
        app.$model = {}
        load('../model', (filename, { schema }) => {
            console.log('load model: ' + filename, schema)
            app.$model[filename] = mongoose.model(filename, schema)
        })
    }

    module.exports = {
        loadModel
    }
```

- 通过loader加载数据模型

```
// index.js
// 初始化数据库
const config = require('./config')
const {loadModel} = require('./framework/loader.js')
loadModel(config)(app)
```

/framework/router.js

```
const router = require('koa-router')()
const {
    init, get, create, update, del,
} = require('./api')

router.get('/api/:list', init, get)
router.post('/api/:list', init,create)
router.put('/api/:list/:id', init, update)
router.delete('/api/:list/:id', init, del)

module.exports = router.routes()
```

/framework/api.js

```
module.exports = {
    async init(ctx, next) {
        console.log(ctx.params)
        const model = ctx.app.$model[ctx.params.list]
        if (model) {
            ctx.list = model
            await next()
```

```
        } else {
            ctx.body = 'no this model'
        }
    },

    async get(ctx) {
        ctx.body = await ctx.list.find({})

    },
    async create(ctx) {
        const res = await ctx.list.create(ctx.request.body)
        ctx.body = res
    },
    async update(ctx) {
        const res = await ctx.list.updateOne({ _id: ctx.params.id }, ctx.request.body)
        ctx.body = res
    },
    async del(ctx) {
        const res = await ctx.list.deleteOne({ _id: ctx.params.id })
        ctx.body = res
    },
    async page(ctx) {
        console.log('page...', ctx.params.page)
        ctx.body = await ctx.list.find({})/*  */
    },
}
```

- 加载路由

```
const bodyParser = require('koa-bodyparser')
app.use(bodyParser())
app.use(require('koa-static')(__dirname + '/'))
app.use(restful)
```

三、设计模式

> http://c.biancheng.net/view/1383.html

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。 1995 年，GoF（Gang of Four，四人组/四人帮）合作出版了《设计模式：可复用面向对象软件的基础》一书，共收录了 23 种设计模式，从此树立了软件设计模式领域的里程碑，人称「GoF设计模式」。 这 23 种设计模式的本质是面向对象设计原则的实际运用，是对类的封装性、继承性和多态性，以及类的关联关系和组合关系的充分理解。

1. 策略模式

策略（Strategy）模式的定义：该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

1. 多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。
2. 策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。
3. 策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。
4. 策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。
5. 策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

思考一下这个怎么搞

```
validation.add(document.getElementById('pba'), 'isNoEmpty', '用户名不能为空');
        validation.add(document.getElementById('pba'), 'minLength:6', '长度不能小于6个字符');
```

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

<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>

<body>
    <div>
        <label>用户名称: <input type="text" id="pba" placeholder="请输入用户名称" onkeydown="valid()" /></label>

        <br /><br />
        <label id='msg'></label>
    </div>

    <script type="text/javascript">
        // 策略对象
        var VerifiPolicy = {
            // 判断是否为空
            isNoEmpty: function (value, errorMsg) {
                if (value == '') {
                    return errorMsg;
                }
            },
            // 判断最小长度
            minLength: function (value, length, errorMsg) {
                console.log('a', value, length)
                if (value.length < length) {
                    return errorMsg;
```

```
                }
            },
            // 判断是否为手机号
            isMobile: function (value, errorMsg) {
                if (!/(^1[3|5|8][0-9]{9}$)/.test(value)) {
                    return errorMsg;
                }
            }
            // 其他
        }
        // 构造函数
        var Formvalidation = function (VerifiPolicy) {
            // 保存策略对象
            this.strategies = VerifiPolicy;
            // 验证缓存
            this.validationFns = [];
        }
        // add 方法
        Formvalidation.prototype.add = function (dom, rule, errorMsg) {
            var ary = rule.split(':');
            var arg = [];
            var self = this;
            this.validationFns.push(function () {
                arg = []; // 重置参数
                var ruleName = ary[0]; // 策略对象方法名
                // 组装参数
                arg.push(dom.value);
                if (ary[1]) {
                    arg.push(ary[1]);
                }
                arg.push(errorMsg);
                // 调用策略函数
                return self.strategies[ruleName].apply(dom, arg);
            });
        }
        // 开始验证
        Formvalidation.prototype.start = function () {
            for (var i = 0; i < this.validationFns.length; i++) {
                var msg = this.validationFns[i]();
                if (msg) {
                    return msg;
                }
            }
        }
        var validation = new Formvalidation(VerifiPolicy);
        console.log(document.getElementById('pba'))
        validation.add(document.getElementById('pba'), 'isNoEmpty', '用户名不能为空');
        validation.add(document.getElementById('pba'), 'minLength:6', '长度不能小于6个字
符');
        function valid() {
            var msg = validation.start()
            document.getElementById('msg').innerHTML = msg
        }
```

```
    </script>
</body>

</html>
```

2. 观察者模式

观察者（Observer）模式的定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

1. 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
2. 目标与观察者之间建立了一套触发机制。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <div>
        <label>用户名称: <input type="text" id="pba" placeholder="请输入用户名称" />
</label><br /><br />
        <label>生成邮箱: <input type="text" id="oba" readonly /></label>
        <label>生成ID: <input type="text" id="obb" readonly /></label>
    </div>

    <script type="text/javascript">
        //发布者
        function Publisher(obj){
            this.observers = [];
            var state = obj.value;      //让该内容不能直接访问
            //新增两个对于state的操作 获取/更新
            this.getState=function(){
                return state;
            }
            this.setState=function(value){
                state = value;
                this.notice();
            }
            this.obj = obj;
        }
        Publisher.prototype.addOb=function(observer){
            var flag = false;
            for (var i = this.observers.length - 1; i >= 0; i--) {
                if(this.observers[i]===observer){
```

```
                    flag=true;
                }
            };
            if(!flag){
                this.observers.push(observer);
            }
            return this;
        }
        Publisher.prototype.removeOb=function(observer){
            var observers = this.observers;
            for (var i = 0; i < observers.length; i++) {
                if(observers[i]===observer){
                    observers.splice(i,1);
                }
            };
            return this;
        }
        Publisher.prototype.notice=function(){
            var observers = this.observers;
            for (var i = 0; i < observers.length; i++) {
                    observers[i].update(this.getState());
            };
        }
        //订阅者
        function Subscribe(obj){
            this.obj = obj;
            this.update = function(data){
                this.obj.value = data;
            };
        }
        //实际应用
        var oba = new Subscribe(document.querySelector("#oba")),
            obb = new Subscribe(document.querySelector("#obb"));
        var pba = new Publisher(document.querySelector("#pba"));
        pba.addOb(oba);
        pba.addOb(obb);
        oba.update = function(state){
            this.obj.value = state+"@w3c.com";
        }
        obb.update = function(state){
            this.obj.value = "ID-"+state;
        }
        pba.obj.addEventListener('keyup',function(){
            pba.setState(this.value);
        });

    </script>
</body>
</html>
```