电商前端架构简介以及支撑服务梳理

图灵：楼兰

这一节课主要是带大家快速梳理电商项目中一些非核心的支撑服务。包括前端的重要流程以及分布式任务调度。课程内容主要是带一些基础比较薄弱的同学快速的熟悉整个电商项目，为后面马上就要开始的微服务内容进行铺垫。

今天课程的关注点是以下几个问题：

- 电商前端重要功能梳理：快速熟悉前端项目的启动方式，路由策略，以及状态保存机制
- 电商登录用户状态同步机制梳理：以这个示例场景熟悉电商项目前后端交互的机制。
- 分布式任务调度系统部署及使用：快速部署并开发一个xxljob定时任务。具体的业务会在后续课程中讲解，这里只介绍xxljob的部署及开发方式。

一、电商前端项目简介

电商项目采用前后端分离的架构，前端有两个应用，mall-admin-web：这个是电商管理系统的前端工程。 tlmall-front：这个是电商项目的前端工程。这两个前端工程都是采用VUE框架开发的前端项目。关于VUE框架的基础知识，比如Nodejs环境搭建、VUE的模版加载、双向绑定等，这里我们就不做介绍了。如果对VUE不熟悉的同学，可以先看下之前徐庶老师提供的单体版本电商项目课程。今天只是帮助大家快速调试前端源码。

1、前端项目启动方式

mall-admin-web项目的开发环境启动指令： npm run dev 。 本地可以通过开发环境直接启动。

tlmall-front项目的开发环境启动指令： npm run serve 。

> 注：需要在本地先安装nodejs，建议版本v14.15.0
>
> 如果第一次启动，需要使用npm install指令下载大量node依赖。所有依赖会下载到nolde_modules目录下。
>
> 如果出现错误Local package.json exists, but node_modules missing, did you mean to install?可以使用cnpm install指令下载。cnpm指令默认访问的是国内阿里的镜像库资源，比npm更稳定。
>
> 指令如果不记得的话，可以查看项目中的package.json文件。里面的scripts部分就对应具体的指令。

开发环境下可以启动本地调试模式。在IDE中修改前端代码，会即时编译并生效。

如果要部署到生产环境，可以使用 npm run build(两个项目配的都一样)。Vue会将整个项目编译成压缩的文件，放到项目的dist目录下。后续如果需要执行的话，需要自行部署nginx。将dist目录下的文件全部拷贝到nginx的工作目录中即可执行。nginx中的参考配置如下：

```
server
    {
        listen 81;

        index index.html index.htm index.php;
```

```
        root  /www/server/nginx;

        #error_page   404   /404.html;
        include enable-php.conf;

        location / {
        root tlmall-admin;
               index index.html index.htm index.php;
        }
        access_log  /www/wwwlogs/access.log;
    }
server
    {
        listen 88;

        index index.html index.htm index.php;
        root  /www/server/nginx;

        #error_page   404   /404.html;
        include enable-php.conf;

        location /{
        index index.html index.htm index.php;
        root tlmall-front;
        }

        location /home/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.214:8887/home/;
                proxy_pass http://192.168.65.214:8888/home/;
            }

        location /pms/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.220:8866/pms/;
                proxy_pass http://192.168.65.214:8888/pms/;
            }

            location /cart/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.165:8855/cart/;
                proxy_pass http://192.168.65.214:8888/cart/;
            }

            location /sso/ {
                proxy_set_header Host $http_host;
```

```
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.152:8877/sso/;
                proxy_pass http://192.168.65.214:8888/sso/;
            }

            location /member/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.152:8877/member/;
                proxy_pass http://192.168.65.214:8888/member/;
            }

            location /order/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                #proxy_pass http://192.168.65.221:8844/order/;
                proxy_pass http://192.168.65.214:8888/order/;
            }

            location /static/qrcode/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_pass http://192.168.65.221:8844/static/qrcode/;
            }

        location /es/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header REMOTE-HOST $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_pass http://192.168.65.152:8054/;

            }

        access_log  /www/wwwlogs/access.log;
    }
```

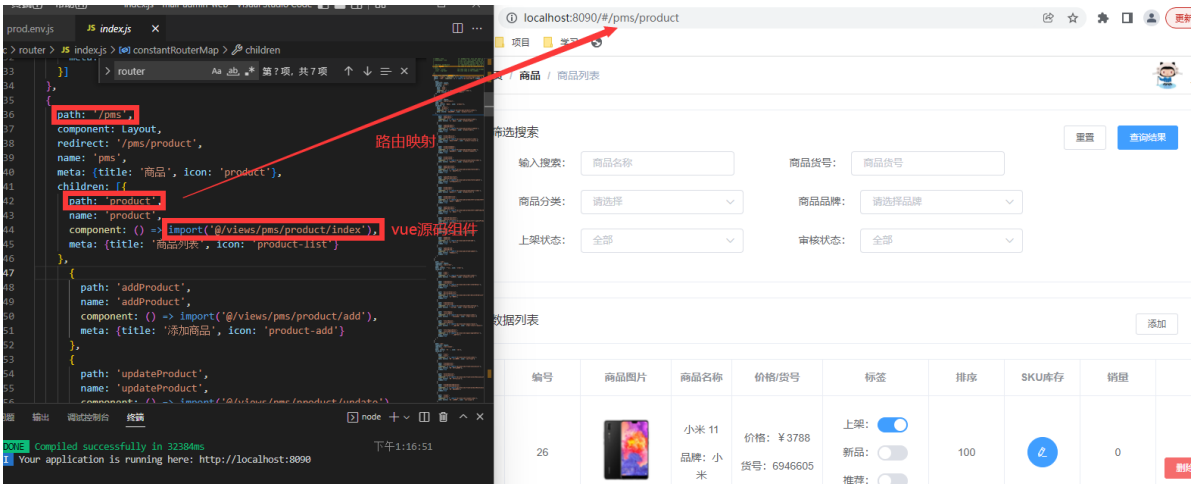
> 88端口为mall-admin-web。81端口为tlmall-front
>
> 192.168.65.214:8888为网关服务的部署地址。后端服务地址根据部署情况自行调整。网关模块会对登录状态进行验证后，再通过微服务进行请求转发。

2、前端页面路由机制

这一部分主要是要能通过前端路径快速定位到vue源码。项目中使用vue-router组件快速进行前端跳转。核心的配置文件是router.js。这个文件会列出vue框架支持的所有URL。

> mall-admin-web中的文件是src/router/index.js

可以通过下图快速进行前端路径定位。



3、后端请求路由机制

vue的前端请求地址都是以/#/开头，后端跳转地址则没有#号。

mall-admin-web的后端地址配置在/config/dev.env.js中。

```
module.exports = merge(prodEnv, {
  NODE_ENV: '"development"',
  BASE_API: '"http://tlmall-admin:8081"'
})
```

通过这个配置，vue就会将对应的后端请求全部转发到BASE_API指定的电商管理系统后台中。

例如在src/pms/brand/index.vue(对应管理后台的品牌管理功能)中，通过下面方法会发起一次后端请求

```
handleUpdate(index, row) {
        this.$router.push({path: '/pms/updateBrand', query: {id: row.id}})
      }
```

这个请求的实际地址就是http://tlmall-admin:8081/pms/updateBrand。

mall-admin-web只对应一个后端地址即可，而在tlmall-front中，后端的对应服务地址就会比较多。这时，会根据请求的第一个部分，例如/pms/转发到不同的后端服务当中。具体的配置参见vue.config.js

```
devServer:{
    host:'localhost',
    port:8080,
    proxy:{
      '/home/*':{
       target:'http://localhost:8887',
        changeOrigin:true,
        // pathRewrite:{
        //   '/home':''
        // }
      },
        '/pms/*':{
            target:'http://localhost:8866',
            changeOrigin:true,
        },
```

```
        '/cart/*':{
            target:'http://localhost:8855',
            changeOrigin:true,
        },
        '/sso/*':{
            target:'http://localhost:8877',
            changeOrigin:true,
        },
        '/member/*':{
            target:'http://localhost:8877',
            changeOrigin:true,
        } ,
        '/order/*':{
            target:'http://localhost:8844',
            changeOrigin:true,
        },
        '/es/*':{
            target:'http://localhost:8054',
            changeOrigin:true,
            pathRewrite:{
              '/es':''
            }
        }
    }
```

> 1、具体调试时，根据自己的服务部署情况进行修改。
>
> 2、这种后端请求地址跳转只在调试模式下生效。生产环境经过build编译之后，这种后端跳转功能就失效了，这时可以通过nginx配置类似的请求转发规则。
>
> 3、host属性配置成localhost则在开发模式下，只能本地访问。如果想要支持远程访问，可以配置成0.0.0.0

4、VUE状态持久化机制

在电商的前端项目中，都通过VUE实现了组件化。有些数据需要在多个组件之间传递或者共享，这部分数据官方说法叫做状态管理。在当前电商项目中，主要的状态数据就是用户的登录信息。

> mall-admin-web项目对于未登录用户的前端请求，直接跳转到登录页。而tlmall-front项目对于未登录用户，则是在页面头部不显示用户登录信息。

电商两个前端项目都是使用vuex组件作为前端的状态管理组件，以下就以tlmall-front项目为例，简单介绍vue的前端状态管理机制。

vuex是一个专门为vue.js应用程序开发的状态管理模式，其中有五种基础的状态管理对象。

- store：存储状态

  主要是用来定义需要保存哪些状态。

例如，在/src/store.index.js中定了需要保存哪些状态。

```
const state = {
  token:'',
  username:'',//登录用0
  cartCount:0,//购物车商品数量
  memberId: 0,
  nickName:''
}
export default new Vuex.Store({
  state,
  mutations,
  actions
});
```

这其中定义的state对象就是需要多个组件共享的状态信息。定义了状态后，在前端vue模版中，就可以用 {{this.$store.state.memberId}}直接访问这些变量。

另外，在项目中也通常会使用computed声明状态，这样一旦状态值有变化，就可以在vue模版中及时感应到。这比直接访问state显然要更合适一点。例如在/etc/components/NavHeader.vue模版中，就会这样访问用户名

```
#页面：
<a href="javascript:;" v-if="username">{{username}}</a>
#js
computed:{
       username(){
         return this.$store.state.username;
       },
       cartCount(){
         return this.$store.state.cartCount;
       },
       ...mapState(['username','cartCount'])
    }
```

这样的好处是发现username状态有变化，就会触发对username的数据绑定，从而拿到最新的状态。

- getters：对变量进行计算

  通常vue官方不建议直接访问状态值，而建议通过getters组件来访问state状态值。

```
const getters = {
  sidebar: state => state.app.sidebar,
  device: state => state.app.device,
  token: state => state.user.token,
  avatar: state => state.user.avatar,
  name: state => state.user.name,
  roles: state => state.user.roles
}
export default getters
```

> getters中也允许对state的值进行计算，但是通常不建议这么做。

在vue模版中，就可以用{{this.$store.getters.name}}来获取对应的状态值。并且，在vue中，这些getter的返回值会根据他的依赖被缓存起来，只要他的依赖值发生了变化，就会触发重新计算。也就是说，getters相当于是对状态进行computed计算。

与mapState相似，vues也提供了mapGetters快速访问这些getters。

```
export default {
  computed: {
    ...mapGetters({
      myname:name  //将store中的name值映射到本地实例的myname变量
    })
  }
}
#如果不 需要对store中的值进行改名，也可以使用更为简单的方式
export default {
  computed: {
    ...mapGetters(['name'])
  }
}
```

- mutations：对store中的值进行修改

  state中的值可以快速获取，但是通常不建议直接修改。如果需要修改，就需要使用mutations。例如在/src/store/mutations.js中声明了商城用到的mutations

```
export default {
  saveToken(state,token){
    state.token=token;
  },
  saveUserName(state, username) {
    state.username = username;
  },
  saveCartCount(state, count) {
    state.cartCount = count;
  },
  saveMemberId(state,memberId){
    state.memberId=memberId;
  },
  saveNickName(state,nickName){
    state.nickName=nickName;
  }
}
```

> 注意，这里export声明的对象，还是需要在/src/store/index.js中引用，放到Vues.store对象中。

声明了mutation后，在前端vue模版中，就可以使用 this.$store.commit()方法调用这些mutation。

- actions：异步提交mutation操作

  使用mutation就可以修改state的值，但是电商项目中并没有这样的用法。实际上，vue官方也不建议使用mutation来修改state的值。这是因为mutation中的函数必须是同步的。而为了处理异步的操作，官方建议使用action来提交一个mutation，以这样的方式来修改state的值。例如，在/stc/store/action.js中，定义了电商项目用到的action。

```
export default {
  saveToken(context,token){
    context.commit('saveToken',token);
```

```
  },
  saveUserName(context,username){
    context.commit('saveUserName', username);
  },
  saveMemberId(context,memberId){
    context.commit('saveMemberId', memberId);
  },
  saveNickName(context,nickName){
    context.commit('saveNickName', nickName);
  },
  saveCartCount(context, count) {
    context.commit('saveCartCount', count);
  },
  saveItemids(context,itemids){
    context.commit('saveitemids',itemids);
  }
}
```

> 这里通过export声明的action对象同样需要在/src/store/index.js中声明到Vuex.store对象中。
>
> 这里通过context.commit方法传入的第一个参数，就是对应的mutation，第二个参数是mutation的参数。

声明了action之后，在vue前端模版中，就可以通过this.$store.dispatch()方法调用action。例如，在/src/pages/login.vue登录页面，在登录完成后，就调用了几个action来保存状态。

```
methods:{
    login(){
      let { username,password,verifycode } = this;

      this.axios.post(
        '/sso/login',
         Qs.stringify({
         username:username,
         password:password,
         verifyCode: verifycode
         }),{headers: {'Content-Type': 'application/x-www-form-
urlencoded'}}).then((res)=>{
            this.$cookie.set('token',res.tokenHead+' '+res.token,
{expires:'1M'});
            this.$cookie.set('memberId',res.memberId,{expires:'1M'});
            this.$cookie.set('nickName',res.nickName,{expires:'1M'});
            this.$store.dispatch('saveToken',res.token);
            this.$store.dispatch('saveMemberId',res.memberId);
            this.$store.dispatch('saveNickName',res.nickName);
    ......
    },
    ...mapActions(['saveUserName']),
    ......
}
```

在action中可以进行一些异步的复杂操作，例如发起http请求，然后在返回值中调动mutation，记录state值。action是vue官方建议的操作方式。

- modules：模块化管理状态

  modules就是将store分隔成不同的模块，每个模块可以有自己的stete、getters、mutation、action、getter等，甚至可以嵌套子模块。这样可以避免在大型应用当中store对象过于庞大的问题。在tlmall-front中并没有使用模块机制，但是在mall-admin-web项目中，就将应用和用户管理分成了两个module。见/src/stroe/index.js。

```
const store = new Vuex.Store({
  modules: {
    app,
    user
  },
  getters
})

export default store
```

> 实际上，如果你仔细体会下，vue的整个状态机制，就相当于是前端的MVC架构。 action就相当于是controller，直接响应vue模版中的请求。state就相当于是entity，负责保存最终的数据。而中间的getters和mutations就相当于是service服务，负责业务逻辑。

二、电商项目用户登录状态同步机制梳理

1、前端往后端发请求时通过请求头携带信息。

在电商项目中，实现了前后端统一的用户登录信息管理机制。也就是说只要完成了登录，那么在电商项目的前端页面和后端请求中都可以拿到用户的登录信息。

以tlmall-front项目为例，按照vuex状态机制去梳理他的登录逻辑，就会发现对于电商项目，在完成登录后，会将登录信息同时保存到cookie以及store中。其中，大部分前端页面都通过cookie来判断用户登录情况。这个没什么特别的。

但是有很多后端请求也需要获取用户信息，他们是如何拿到登录用户的呢？例如在电商的购物车页面，会列出当前用户的所有购物车商品信息。这里显然是需要获取登录用户状态的。

```
    mounted(){
      this.getCartList();
    },
    methods:{
      // 获取购物车列表
      getCartList(){
        this.axios.get('/cart/list').then((res)=>{
          this.renderData(res);
        })
      },
      ......
    }
```

在前端项目中，在使用vue的axios插件发送http请求时，设定了一个拦截器，每次往后端发起http请求时，都会从cookie中获取登录的用户信息，然后放入到http请求的header头部中。在tlmall-front的/src/main.js中，对axios进行了声明。

```
// request
let v_loading=false;
```

```
axios.interceptors.request.use(config => {
  if(config.method=="post")v_loading=Loading.service();
  var token= getCookie("token");
  var memberId=getCookie("memberId");
  var nickName=getCookie("nickName");
if (token !=undefined) {
    config.headers['Authorization'] = token; // 让每个请求携带自定义token 请根据实际情
况自行修改
  }
  if (memberId !=undefined) {
        config.headers['memberId'] =memberId;
    }
  if (nickName !=undefined) {
        config.headers['nickName'] = nickName;
    }
    return config
}, error => {
  // Do something with request error
  Promise.reject(error)
})
```

> 思考一下为什么要用cookie客户端缓存，而不用store中的state呢?
>
> 这是因为js代码都是运行在内存中，state也是存在内存当中的，所以一旦客户按下F5刷新页面，以前申请的内存就会被释放，然后就需要重新加载js脚本计算。而使用cookie则绕过了重新计算的过程，提升客户端运行效率。当然，cookie不是很安全，客户端可以自行清空cookie，甚至修改cookie。如果需要更安全的客户端数据持久化，可以使用localStorage本地存储空间以及sessionStorage会话存储空间。

2、gateway模块对token和memberId进行校验，校验不通过则跳转到登录页重新登录

请求发往gateway模块后，gateway模块会拿到用户的token，进行校验。主要是检测用户的的登录状态是否超时。

```
//AuthorizationFilter
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        String currentUrl = exchange.getRequest().getURI().getPath();
        //1:不需要认证的url
        if(shouldSkip(currentUrl)) {
            //log.info("跳过认证的URL:{}",currentUrl);
            return chain.filter(exchange);
        }
        //log.info("需要认证的URL:{}",currentUrl);
        //第一步:解析出我们Authorization的请求头  value为: “bearer XXXXXXXXXXXXX”
        String authHeader =
exchange.getRequest().getHeaders().getFirst("Authorization");
        //第二步:判断Authorization的请求头是否为空
        if(StringUtils.isEmpty(authHeader)) {
            log.warn("需要认证的url,请求头为空");
            throw new
GateWayException(ResultCode.AUTHORIZATION_HEADER_IS_EMPTY);
        }
        //第三步 校验我们的jwt 若jwt不对或者超时都会抛出异常
```

```
        Claims claims = JwtUtils.validateJwtToken(authHeader,publicKey);
        //第四步 把从jwt中解析出来的 用户登陆信息存储到请求头中
        ServerWebExchange webExchange = wrapHeader(exchange,claims);
        return chain.filter(webExchange);
    }
```

> oauth2.0的校验流程会在后续课程中详细讲解。

如果校验不通过，则会返回给前端项目。前端项目通过axios中的response插件响应不正常的请求，并跳转到登录页。

```
// tlmall-front->main.js
// 接口错误拦截
axios.interceptors.response.use(function(response){
  if(v_loading){v_loading.close();v_loading=false;}
  let res = response.data;
  let path = location.hash;
  if(res.code == 200){
    return res.data;
  }else if(res.code==401 || res.code==403|| res.code==600){  //|| res.code==401
|| res.code==403
    if (path != '#/index'){

      window.location.href = '/#/login';
    }
    return Promise.reject(res);
  }else{
    Message.warning(res.message);
    return Promise.reject(res);
  }
},(error)=>{
  if(v_loading){v_loading.close();v_loading=false;}
  let res = error.response;
  if(res) Message.error(res.data.message);
  else Message.error(error.message);
  return Promise.reject(error);
});
```

3、gateway校验通过，往后端发送请求信息。

gateway校验通过后，会将前端传过来的参数添加到http请求的header中，继续往后端服务发送。

```
//AuthorizationFilter
private ServerWebExchange wrapHeader(ServerWebExchange serverWebExchange,Claims
claims) {
        String loginUserInfo = JSON.toJSONString(claims);
        //log.info("jwt的用户信息:{}",loginUserInfo);
        String memberId =
claims.get("additionalInfo",Map.class).get("memberId").toString();
        String nickName =
claims.get("additionalInfo",Map.class).get("nickName").toString();
        //向headers中放文件，记得build
        ServerHttpRequest request = serverWebExchange.getRequest().mutate()
                .header("username",claims.get("user_name",String.class))
                .header("memberId",memberId)
```

```
                .header("nickName",nickName)
                .build();
        //将现在的request 变成 change对象
        return serverWebExchange.mutate().request(request).build();
    }
```

接下来，在后端业务中，就可以通过@RequestHeader注解获取到请求头中的参数。例如，在购物车模块，是这样获取memberId的：

```
//tulingmall-cart模块com.tuling.tulingmall.controller.CartItemController
    @ApiOperation("获取某个会员的购物车列表")
    @RequestMapping(value = "/list", method = RequestMethod.GET)
    @ResponseBody
    public CommonResult<List<OmsCartItem>> list(@RequestHeader("memberId") Long
memberId) {
        List<OmsCartItem> cartItemList = cartItemService.list(memberId);
        return CommonResult.success(cartItemList);
    }
```

4、后端微服务模块之间，通过拦截器统一传递请求头信息

前端请求发送到后端后，后端微服务之间会进行频繁的服务调用。在这些服务调用之间，依然需要同步用户的登录状态。在当前电商项目中，会采用拦截器统一添加请求头的方式，在后端微服务之间继续传递用户状态。例如在购物车模块tulingmall-cart中，添加了一个拦截器，在需要进行基于feign的微服务RPC调用时， 统一将memberID添加到请求头中，这样目标服务也可以通过请求头拿到登录的用户ID。

```
package com.tuling.tulingmall.feignapi.interceptor;
......
@Slf4j
public class HeaderInterceptor implements RequestInterceptor {
    @Override
    public void apply(RequestTemplate template) {
        ServletRequestAttributes attributes = (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        if(attributes != null){
            HttpServletRequest request = attributes.getRequest();
            log.info("从Request中解析请求头:{}",request.getHeader("memberId"));
            template.header("memberId",request.getHeader("memberId"));
        }
    }
}
```

> 在电商项目的很多个服务模块中，都采用这样的方式来同步memberId。

但是这种基于HTTP请求头传递消息的方式，通常只适合少量比较小的数据，如果请求头内容过长时，很容易抛出Request Header is too large的异常。

三、电商分布式任务调度系统快速上手

在大型项目中，使用分布式任务调度系统来运行一些定时任务，通常是必不可少的一个环节，在当前电商项目中，同样也有执行定时任务的需求。例如在订单模块，需要实现一个定时任务，将mysql中的过时订单信息转存到mongodb中(`具体的业务逻辑会在后面课程中分析`)。因此，电商项目也需要部署一个任务调度系统，来运行定时任务。

定时任务是一个非常常见的需求，因此实现的方式是非常多的，也有非常多现成的工具可以选择。重点是需要根据自己的需求选择合适的工具。

1、contab指令快速实现简单的定时任务

例如，在Linux机器上直接使用crontab -e指令，就可以编辑一个简单的调度任务。

```
* * * * * echo "timer">/root/test.out
```

编辑保存后，就可以在Linux上启动一个定时任务，默认会每分钟执行一次。

> 需要保证crond服务是启动的。 service crond start。

这个指令前面几个星号组成一个crontab表达式，后面则是执行任务的脚本。

crontab表达式是一种非常通用的定时任务表达式，在很多地方都用到。这里简单介绍下规则。

```
*    *    *    *    *  ?
-    -    -    -    -  -
|    |    |    |    |  +-- year 年份(可选)。没有可以不填。
|    |    |    |    +----- week 星期中星期几 (0 - 6) (星期天 为0)
|    |    |    +---------- month 月份 (1 - 12)
|    |    +--------------- day 一个月中的第几天 (1 - 31)
|    +-------------------- hour 小时 (0 - 23)
+------------------------- minute 分钟 (0 - 59)
```

> 从这个配置可以看到，Linux系统默认的contab任务执行的时间精度最低是以分钟为单位。而很多定时任务框架都可以做到秒级。配置方式是在分钟前面，加入一个秒的配置。

在配置每一个部分的表达式时，可以指定具体的数值，比如分钟位设置为0，表示每次第0分钟执行一次。在这些基础数字之外，还允许几个特殊的字符：

- * 表示所有可能的值。
- - 表示指定范围。例如在day位配置 1-10，表示每个月的第1天到第10天
- ，表示枚举。例如minute位配置 3,5 表示第3分钟和第5分钟执行。
- / 表示指定增量。例如在minute位配置0/15表示从0分钟开始，每15分钟执行一次。3/20表示第三分钟开始，每20分钟执行一次。

另外还有几个用得比较少的特殊符号。例如对于day和week位置。为了避免可能的冲突，就可以将其中一个值设为?

2、使用xxljob快速实现轻量级分布式任务调度

实现定时任务的框架有很多，目前在国内用得比较多的是elasticjob和xxljob。这两个项目都有非常多的运用场景。其中elasticjob依赖zookeeper进行分布式任务协调，而xxljob则只需要通过数据库进行协调即可。由于当前电商项目中并没有其他组件用到zookeeper，因此，为了简化部署环境，电商项目就是采用的xxljob。

关于xxljob，上手是非常简单的，在VIP官网中也有关于xxljob的一个专项课程。对于如何部署和搭建xxljob的内容，这里就不再多做分享。

在电商项目中，定时任务已经被封装到了一个http接口中，在xxljob中，只需要定时调用http接口即可。因此，在电商项目中，采用GLUE的方式快速部署了一个任务。任务内容如下：

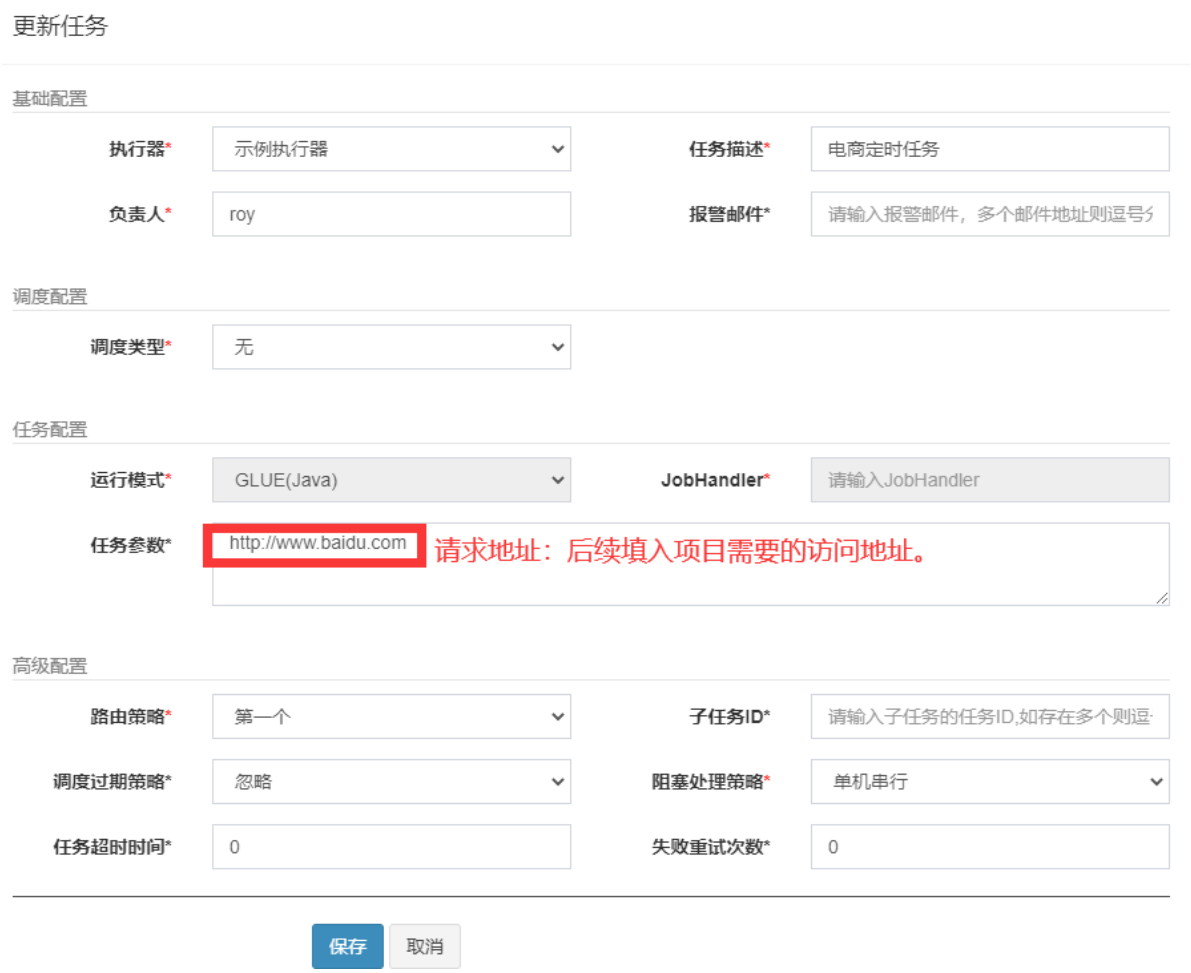
```
package com.xxl.job.service.handler;

import com.xxl.job.core.context.XxlJobHelper;
import com.xxl.job.core.handler.IJobHandler;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpJobHandler extends IJobHandler {
    @Override
    public void execute() throws Exception {
        String target = XxlJobHelper.getJobParam();
        XxlJobHelper.log("received job target:"+target);
        boolean isPostMethod= false;
        // request
        HttpURLConnection connection = null;
        BufferedReader bufferedReader = null;
        try {
            // connection
            URL realUrl = new URL(target);
            connection = (HttpURLConnection) realUrl.openConnection();
            connection.setDoOutput(isPostMethod);
            connection.setDoInput(true);
            connection.setUseCaches(false);
            connection.setReadTimeout(5 * 1000);
            connection.setConnectTimeout(3 * 1000);
            connection.setRequestProperty("connection", "Keep-Alive");
            connection.setRequestProperty("Content-Type",
"application/json;charset=UTF-8");
            connection.setRequestProperty("Accept-Charset",
"application/json;charset=UTF-8");
            // do connection
            connection.connect();
            // valid StatusCode
            int statusCode = connection.getResponseCode();
            if (statusCode != 200) {
                throw new RuntimeException("Http Request StatusCode(" +
statusCode + ") Invalid.");
            }
            // result
            bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream(), "UTF-8"));
            StringBuilder result = new StringBuilder();
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                result.append(line);
            }
            String responseMsg = result.toString();
            XxlJobHelper.log(responseMsg);
            return;
        } catch (Exception e) {
```

```
            XxlJobHelper.log(e);
            XxlJobHelper.handleFail();
            return;
        } finally {
            try {
                if (bufferedReader != null) {
                    bufferedReader.close();
                }
                if (connection != null) {
                    connection.disconnect();
                }
            } catch (Exception e2) {
                XxlJobHelper.log(e2);
            }
        }
    }
}
```

在这个实现中，访问的目标地址target，是作为任务参数传入的。因此，接下来只要在xxljob的任务配置界面将URL以参数形式配置即可。



> 当前项目中，任务参数部分需要填写服务请求地址： http://192.168.65.79:8866/order/migrate/migrateTables。这是在秒杀环节需要进行的一个定时任务。具体业务逻辑会在后续秒杀环节介绍，这里只要能够实现定时请求即可。
>
> 其他参数酌情填写。

3、了解定时任务的核心-时间轮算法

这些框架的使用并不难，但是在上手使用xxljob后，有一个基础设计思想必须要了解。这就是时间轮算法Timing Wheel。这是业界对于定时任务一种非常重要的设计思想。在很多开源项目包括Kafka、Dubbo、Netty、MySQL等产品背后，都有时间轮的影子。

对于时间轮，可以使用Netty提供的HashedWheelTimer来简单体验一下。比如在Maven项目中，只要引入了Netty的依赖。

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.63.Final</version>
</dependency>
```

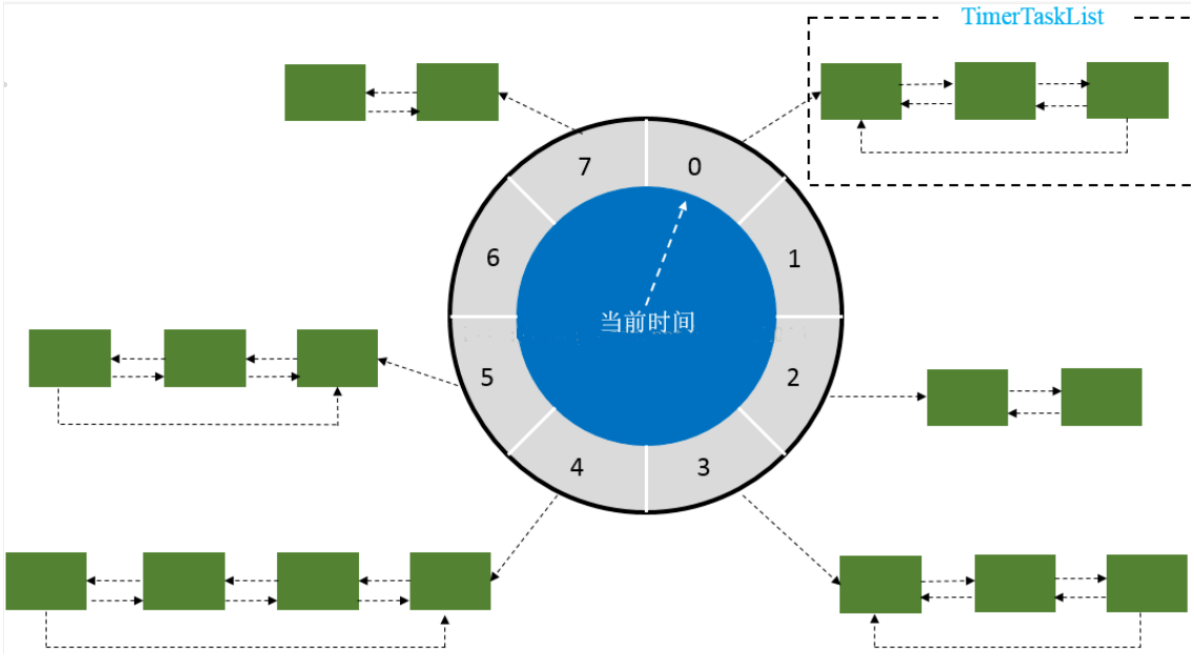
就可以使用下面的示例做个简单的Demo

```
public class TimeWheelDemo {
    public static void main(String[] args) throws InterruptedException {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
        HashedWheelTimer timer = new HashedWheelTimer(r -> new Thread(r,
"HashedWheelTimer " + r.hashCode()), 1, TimeUnit.MILLISECONDS,8);

        TimerTask timerTask = new TimerTask() {
            @Override
            public void run(Timeout timeout) throws Exception {
                System.out.println("hello world " +
LocalDateTime.now().format(formatter));
                //执行完成之后再次加入调度
                timer.newTimeout(this, 4, TimeUnit.SECONDS);
            }
        };
        //将定时任务放入时间轮
        timer.newTimeout(timerTask, 4, TimeUnit.SECONDS);
        Thread.currentThread().join();
    }
}
```

> 执行效果每隔4秒打印出一句话

时间轮是定时任务中最常用的一种算法，大部分的定时任务框架以及很多产品的定时任务实现都或多或少借鉴了这种算法。



时间轮算法可以简单的看成由一个循环数组+双向链表的数据结构实现。循环数组构成一个环形结构，指针每隔tickDuration时间就走一步。而数组中每个节点上挂载了一个双向链表结构的定时任务列表。

> 当然，这只是时间轮的一种标准应用方式。另外还有很多基于标准时间轮的变种实现，例如使用多个时间轮共同组成一个复杂时间轮(模拟时钟实现)等等。

双向链表上的任务有个属性为remainingRounds，表示当前任务剩下的轮次是多少。每当指针走到该任务对应的数据节点上时，remainingRounds就减少1，直到remainingRounds为0时，就触发当前定时任务。从这个原理中可以看到，tickDuration越小，定时任务就越精确，但是响应的，系统的计算负担就越重。