# OA笔记

## 需求简介

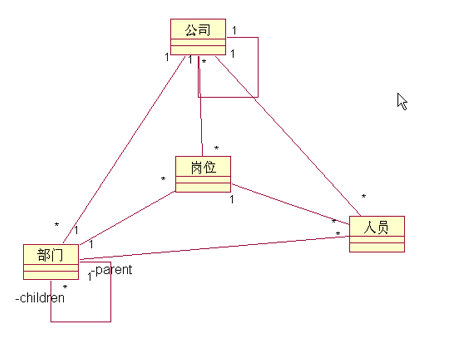
树形结构是资源

菜单项也是资源

把action类看成资源，其中的方法看成操作

## 建立概念模型

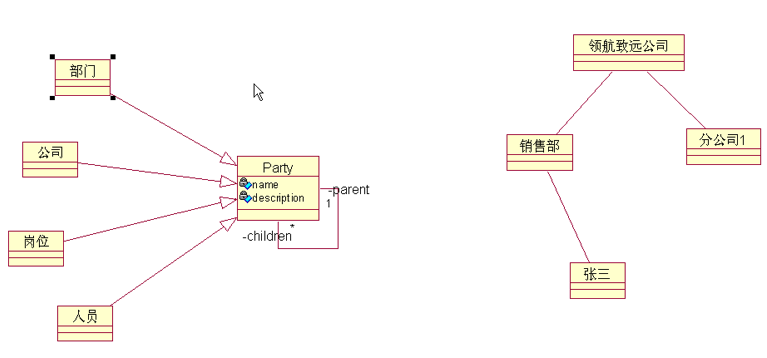
1. 第1步：



1. 第2步：

树形结构

存在问题：如：人员下面不能有公司

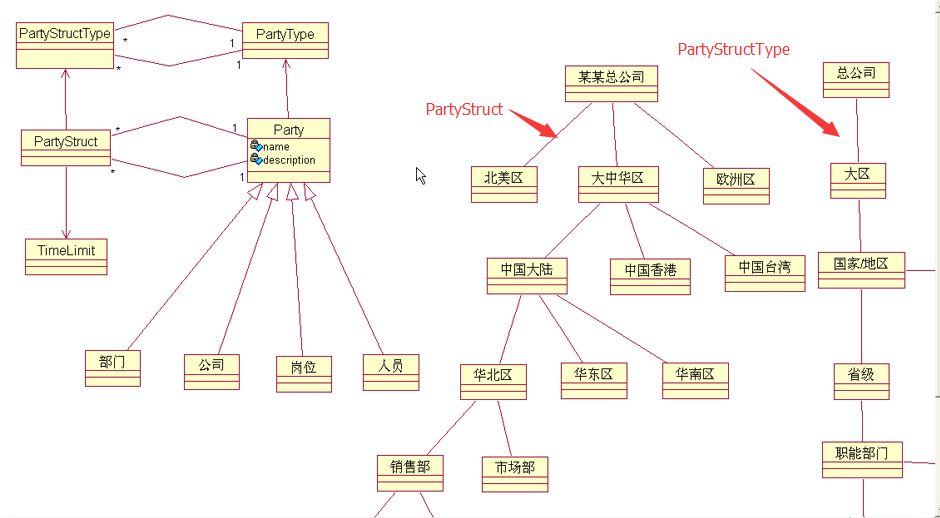


1. 责任模式

PartyStruct： Party和Party之间多对多关系拆分成两个多对一关系，其中多的一端为PartyStruct。

PartyStructType：Party上下级之间的约束规则。如：总公司和大区直接的关系。

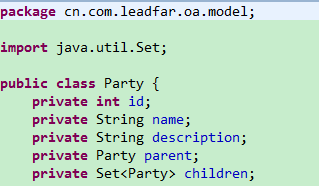
PartyType：Party类型，如：总公司、大区等。



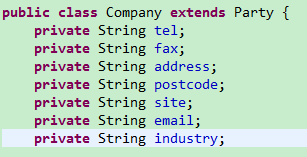
## 建立数据库表

### 建立实体类和Dao层、Service层

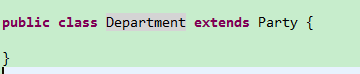
Party类：



company类



Department类，Person类，Position类



hibernate-mapping

<hibernate-mapping>

<class name=*"cn.com.leadfar.oa.model.Party"* table=*"t\_party"* discriminator-value=*"P"*>

<id name=*"id"*>

<generator class=*"native"*/>

</id>

<discriminator column=*"partyType"* type=*"char"*></discriminator>

<property name=*"name"*></property>

<property name=*"description"*></property>

<many-to-one name=*"parent"* column=*"pid"*></many-to-one>

<set name=*"children"* lazy=*"extra"* inverse=*"true"*>

<key column=*"pid"*></key>

<one-to-many class=*"cn.com.leadfar.oa.model.Party"*/>

</set>

<subclass name=*"cn.com.leadfar.oa.model.Company"* discriminator-value=*"C"*>

<property name=*"tel"*></property>

<property name=*"fax"*></property>

<property name=*"address"*></property>

<property name=*"email"*></property>

<property name=*"industry"*></property>

<property name=*"postcode"*></property>

<property name=*"site"*></property>

</subclass>

<subclass name=*"cn.com.leadfar.oa.model.Department"* discriminator-value=*"D"*></subclass>

<subclass name=*"cn.com.leadfar.oa.model.Person"* discriminator-value=*"R"*></subclass>

<subclass name=*"cn.com.leadfar.oa.model.Position"* discriminator-value=*"T"*></subclass>

</class>

</hibernate-mapping>

BaseDao接口定义CRUD方法

**public** **interface** BaseDao {

**public** **void** save(Object entity);

**public** **void** update(Object entity);

**public** **void** del(Object entity);

**public** <T> T findById(Class<T> entityClass,**int** id);

**public** <T> List<T> findAll(Class<T> entityClass);

}

PartyDao接口定义针对Party的CRUD方法，继承BaseDao接口

**public** **interface** PartyDao **extends** BaseDao{

**public** Company findCurrentCompany();

}

BaseDaoImpl类是BaseDao接口的具体实现类

**public** **class** BaseDaoImpl **implements** BaseDao {

@Resource

**private** SessionFactory sessionFactory;

@Override

**public** **void** del(Object entity) {

getSession().delete(entity);

}

@Override

**public** <T> List<T> findAll(Class<T> entityClass) {

**return** getSession().createCriteria(entityClass).list();

}

@Override

**public** <T> T findById(Class<T> entityClass, **int** id) {

**return** (T)getSession().load(entityClass, id);

}

@Override

**public** **void** save(Object entity) {

getSession().save(entity);

}

@Override

**public** **void** update(Object entity) {

getSession().update(entity);

}

**protected** Session getSession(){

**return** sessionFactory.getCurrentSession();

}

}

PartyDaoImpl继承了BaseDaoImpl实现了PartyDao接口

@Repository("partyDao")

**public** **class** PartyDaoImpl **extends** BaseDaoImpl **implements** PartyDao {

@Override

**public** Company findCurrentCompany() {

String hql = "select c from Company c where c.parent is null";

**return** (Company)getSession().createQuery(hql).uniqueResult();

}

}

PartyService定义调用数据持久化层的相关方法

**public** **in**terface PartyService {

public void addParty(Party party);

public void updateParty(Party party);

public Company currentCompany();

}

PartyServiceImpl实现了PartyService接口

@Service("partyService")

**public** **class** PartyServiceImpl **implements** PartyService {

@Resource

**private** PartyDao partyDao;

@Override

**public** **void** addParty(Party party) {

partyDao.save(party);

}

@Override

**public** Company currentCompany() {

**return** partyDao.findCurrentCompany();

}

@Override

**public** **void** updateParty(Party party) {

partyDao.update(party);

}

}

applicationContext-common.xml

<!-- 自动扫描在某个包（包括子包）下面的所有类型，寻找包含特定注解的类，并将它注册到Spring中 -->

<context:component-scan base-package=*"cn.com.leadfar"* />

<!-- 配置数据源 -->

<bean id=*"dataSource"*

class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"*

value=*"jdbc:mysql://localhost/oa"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"leadfar"* />

</bean>

<!-- 将SessionFactory交给Spring来管理 -->

<bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"hibernateProperties"*>

<value> hibernate.dialect=org.hibernate.dialect.MySQLInnoDBDialect

hibernate.show\_sql=true

hibernate.hbm2ddl.auto=update

</value>

</property>

<property name=*"mappingResources"*>

<list>

<value>cn/com/leadfar/oa/model/Party.hbm.xml</value>

</list>

</property>

</bean>

<!-- Hibernate事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"* />

</bean>

<!-- 配置哪些地方需要进行事务管理 -->

<aop:config>

<aop:pointcut id=*"allServiceMethods"*

expression=*"execution(\* cn.com.leadfar.oa.service.\*.\*(..))"* />

<aop:advisor advice-ref=*"txAdvice"*

pointcut-ref=*"allServiceMethods"* />

</aop:config>

<!-- 配置事务特性 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- 所有add开头的方法，都使用事务 -->

<tx:method name=*"add\*"* />

<!-- 所有del开头的方法，都使用事务 -->

<tx:method name=*"del\*"* />

<tx:method name=*"update\*"* />

<!--所有其它方法，都使用事务，而且是只读的，即不允许在这些方法中做CUD操作！ FlushMode - AUTO/MANUAL-->

<tx:method name=*"\*"* read-only=*"true"* />

</tx:attributes>

</tx:advice>

### 利用xml文件初始化数据库数据

begin.xml

<?xml version="1.0" encoding="gbk"?>

<inits package="cn.com.leadfar.oa.model">

<entity class="Company" name="石院" description="石家庄学院" call="partyService.addParty">

<entity class="Department" name="计算机学院">

<entity class="Position" name="院长">

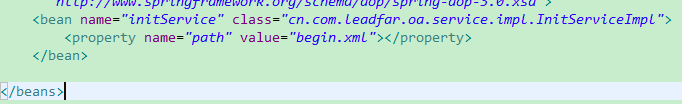
</entity>

</entity>

</entity>

</inits>

applicationContext-service.xml



InitService

**public** **interface** InitService {

**void** addInitDatas();

}

InitServiceImpl

**public** **class** InitServiceImpl **implements** InitService, BeanFactoryAware {

**private** String path; // xml文件的路径，配置注入

**private** BeanFactory factory;

**public** **void** setPath(String path) {

**this**.path = path;

}

// Spring在实例化InitServiceImpl对象的时候，如果InitServiceImpl类实现了BeanFactoryAware接口，

// Spring会自动把自身BeanFactory注入此对象中

@Override

**public** **void** setBeanFactory(BeanFactory factory) **throws** BeansException {

**this**.factory = factory;

}

/\*\*

\* 读取xml文件，获取根节点和根节点的子节点，并添加到数据库

\*/

@Override

**public** **void** addInitDatas() {

**try** {

Document document = **new** SAXReader().read(Thread.*currentThread*()

.getContextClassLoader().getResourceAsStream(path));

// 取得根节点

Element root = document.getRootElement();

// 取得根节点package属性

String pkg = root.attributeValue("package");

// 取得根节点名为entity的子节点

List elements = root.elements("entity");

**for** (Iterator iterator = elements.iterator(); iterator.hasNext();) {

Element element = (Element) iterator.next();

// 添加entity到数据库

addEntity(pkg, element, **null**, **null**);

}

} **catch** (DocumentException e) {

e.printStackTrace();

}

}

/\*\*

\* **@param** pkg包名

\* **@param** element节点

\* **@param** parentEntity上级部门

\* **@param** callString对该entity执行操作要调用的方法

\*/

**public** **void** addEntity(String pkg, Element element, Object parentEntity,

String callString) {

**try** {

String claz = pkg + "." + element.attributeValue("class");

Object entity = Class.*forName*(claz).newInstance();

// 给entity赋值

Iterator iterator = element.attributeIterator();

**while** (iterator.hasNext()) {

Attribute attribute = (Attribute) iterator.next();

String attrName = attribute.getName();

**if** (!attrName.equals("class") && !attrName.equals("call")) {

String attrValue = attribute.getValue();

// 给entity相应的属性值,用commons-beanutils.jar

BeanUtils.*copyProperty*(entity, attrName, attrValue);

}

}

// 判断parentEntity是否为空，如果不为空，则给对象赋值

**if** (parentEntity != **null**) {

Method[] methods = entity.getClass().getMethods();

**for** (Method method : methods) {

// 调用entity的setParent方法，设置上级部门

**if** (method.getName().equals("setParent")) {

method.invoke(entity, parentEntity);

}

}

}

String call = element.attributeValue("call");

**if** (call == **null**) {

// 若节点没有定义call属性，则使用父节点传下来的call属性

call = callString;

}

// .是个特殊字符，需要转义

String[] strs = call.split("\\.");

// 用getBean的方式，因为不能确定要用到哪个service方法

Object service = factory.getBean(strs[0]);

// 在调用Service的方法之前，先检查该类是否存在这个方法

Method[] methods = service.getClass().getMethods();

**for** (Method method : methods) {

**if** (method.getName().equals(strs[1])) {

// 利用反射机制调用方法addEntity，添加entity到数据库

// 第一个参数是方法属于的类，第二个参数为调用该方法传递的参数

method.invoke(service, entity);

}

}

// 判断当前entity下是否含有entity，递归添加子节点

List<Element> subEntityElements = element.elements("entity");

**for** (Iterator iterator2 = subEntityElements.iterator(); iterator2

.hasNext();) {

Element e = (Element) iterator2.next();

addEntity(pkg, e, entity, call);

}

} **catch** (Exception e) {

e.printStackTrace();

}

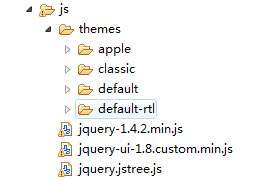
}

}

### Json、Json –tools和Jquery用法见其他笔记

### Js-tree插件的用法

1. 拷贝js库文件



<script type=*"text/javascript"* src=*"js/jquery-1.4.2.min.js"*></script>

<script type=*"text/javascript"* src=*"js/jquery.jstree.js"*></script>

<script type=*"text/javascript"*>

$(**function**() {

$("#menuContainer").jstree({

//配置用哪些插件

plugins : [ "html\_data", "themes", "ui" ],

//下面对所用的插件进行配置

themes : { //对themes这个plugin进行配置

theme : "classic"

}

});

});

</script>

<body>

<div id=*"menuContainer"* style="height:*100%*;">

<ul id=*"navigation"*>

<li><a href=*"#"*>个人办公</a>

<ul>

<li><a href=*"#"*>桌面</a>

<ul>

<li><a href=*"#"*>我的桌面</a>

</li>

<li><a href=*"#"*>桌面配置</a>

</li>

</ul></li>

<li><a href=*"#"*>任务计划</a>

<ul>

<li><a href=*"#"*>创建计划</a>

</li>

<li><a href=*"#"*>查询计划</a>

</li>

<li><a href=*"#"*>计划分类</a>

</li>

</ul></li>

<li><a href=*"#"*>日程安排</a>

<ul>

<li><a href=*"#"*>创建日程</a>

</li>

<li><a href=*"#"*>查询日程</a>

</li>

<li><a href=*"#"*>日程分类</a>

</li>

</ul></li>

<li><a href=*"#"*>日志管理</a>

<ul>

<li><a href=*"#"*>创建日志</a>

</li>

<li><a href=*"#"*>我的日志</a>

</li>

<li><a href=*"#"*>共享日志</a>

</li>

<li><a href=*"#"*>日志分类</a>

</li>

</ul></li>

</ul>

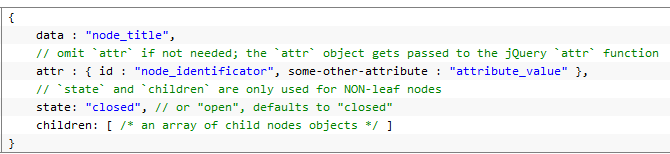
</li>

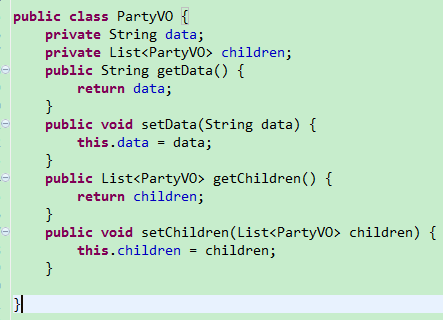
</ul>

</div>

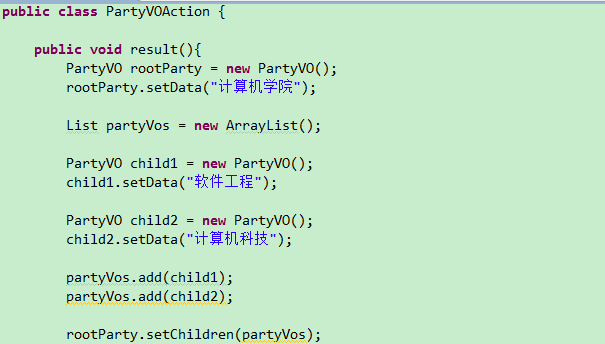
1. 配置themes plugin设置用什么主题，配置html\_data表示把html代码转化成树结构显示，配置json\_data表示把json数据转化成树结构显示，配置UI plugin可以屏蔽掉链接
2. 示例

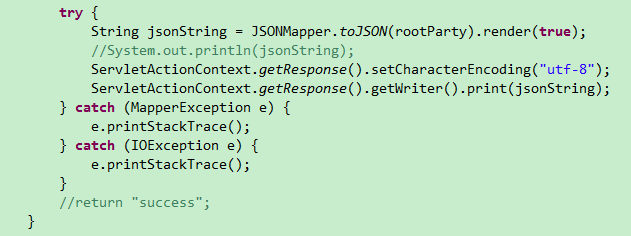
json\_data格式：



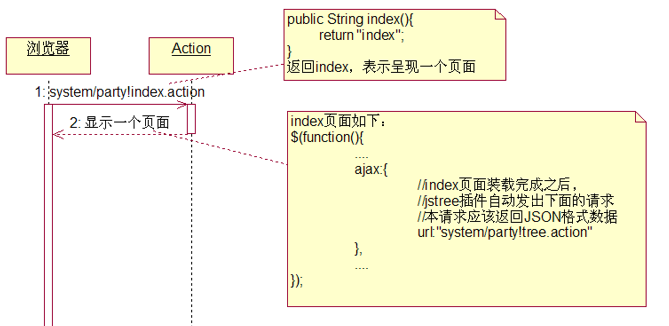


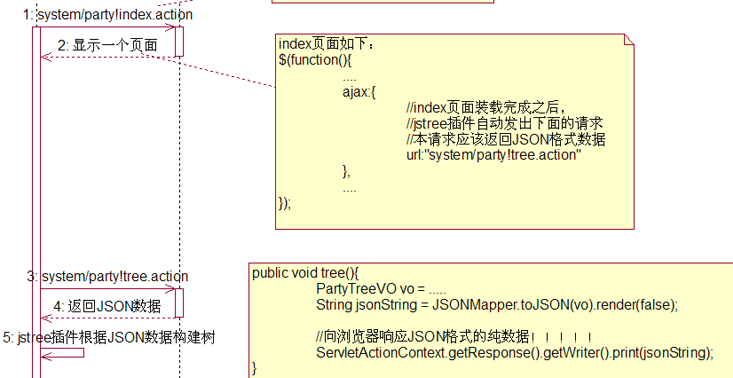






注意：action呈现页面和action返回json数据是两个概念，请求json数据时，方法直接返回json数据，不跳转。





1. Js-tree绑定事件

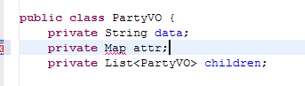
整个树构建完毕之后，执行绑定的事件

$(“#tree”).bind(“事件名”，函数);

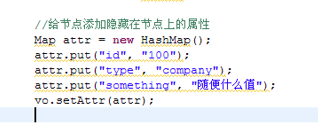
$(“#tree”).jstree(“操作名称”，-1); -1是参数

1. 给节点添加隐藏属性

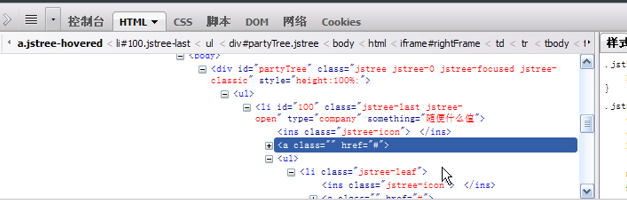
Attr携带节点信息，例如，节点id，类型等。Attr属性定义成Map类型，灵活。



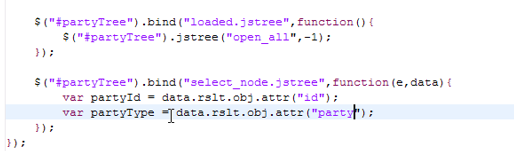
添加属性



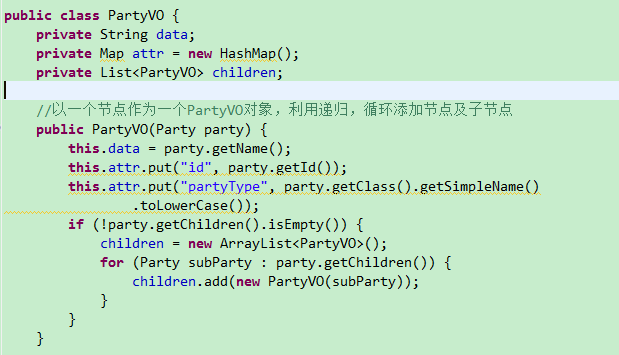
查看Attr携带信息，<li>标签的属性中

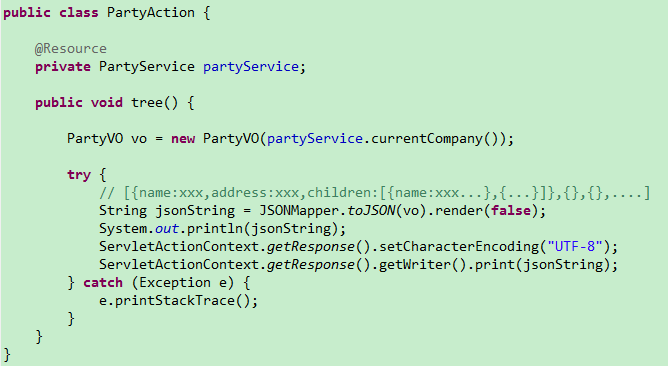


取得attr属性

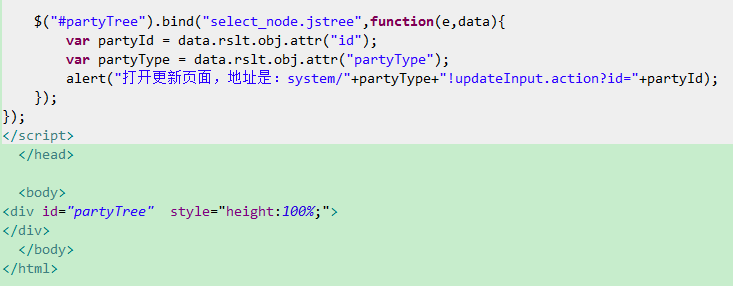


1. 利用PartyTreeVO创建组织机构树



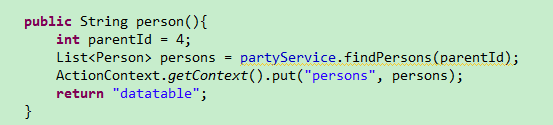


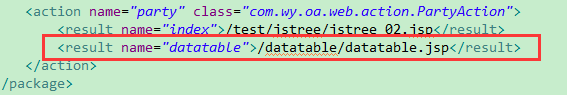


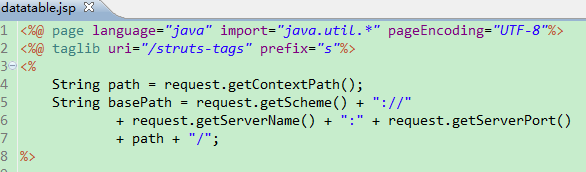


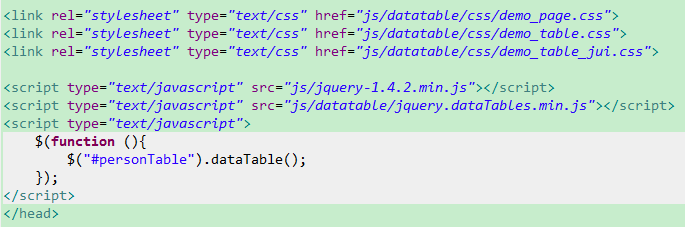
### Datatable插件的用法

1. 前台分页





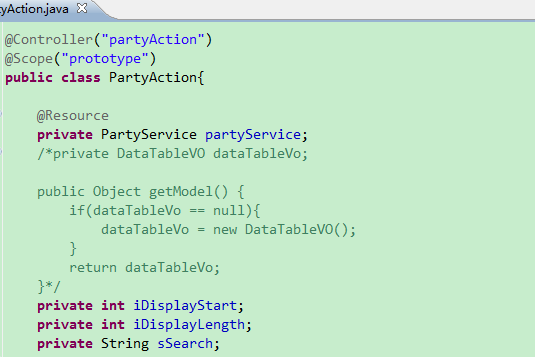


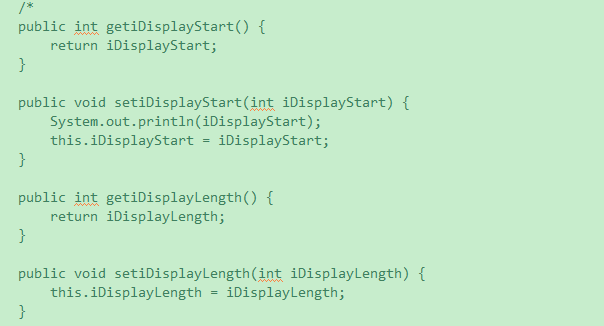


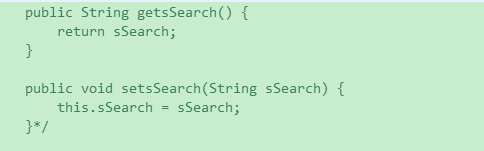


1. 后台分页

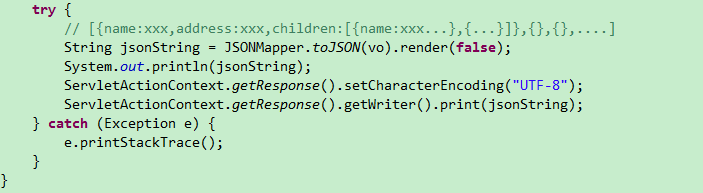
PartyAction



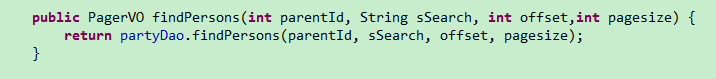




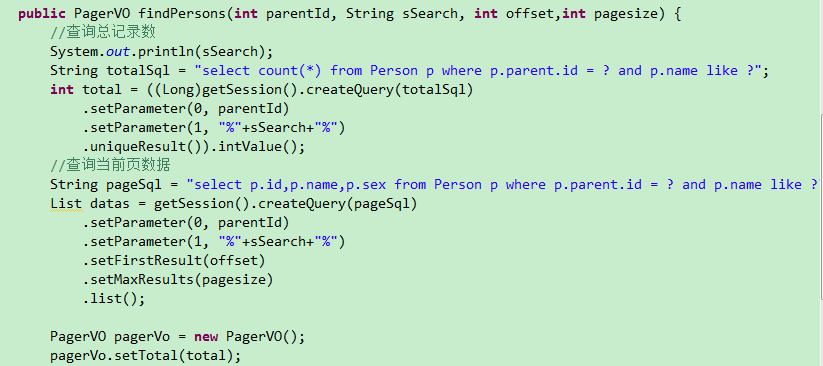




PartyServiceImpl

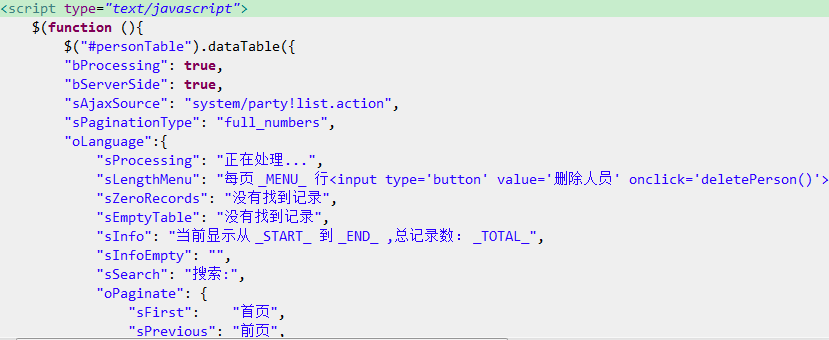


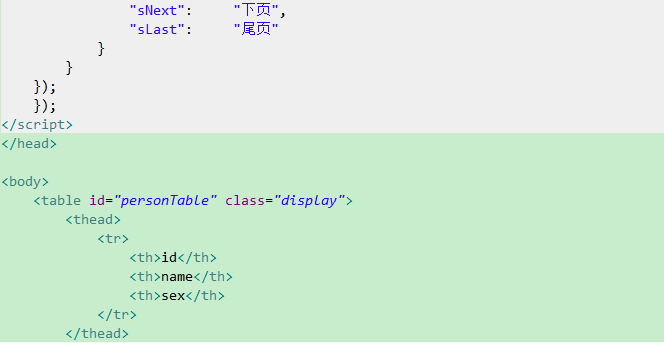
PartyDaoImpl

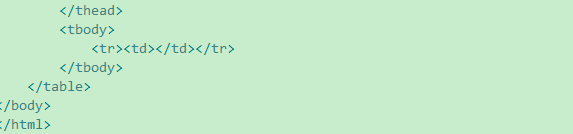




Datatable.jsp



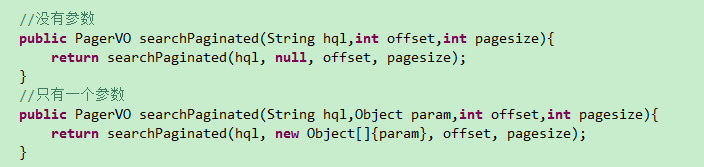


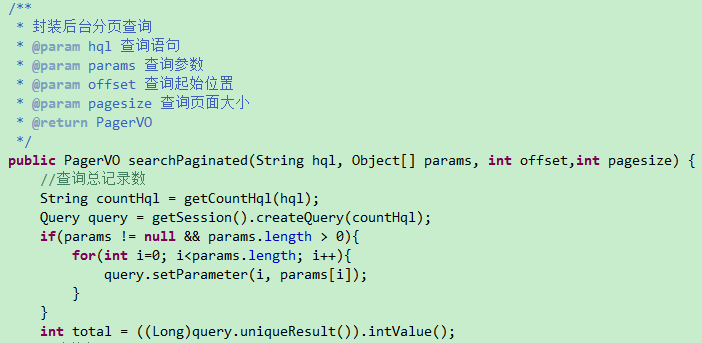


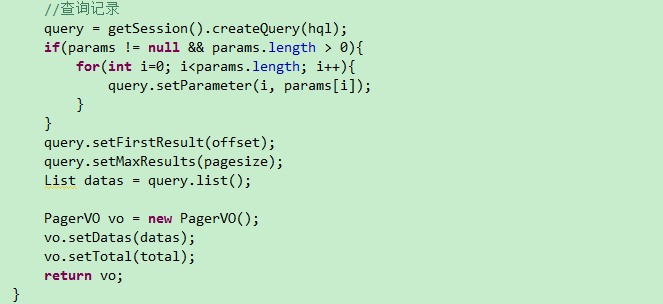
定义的列数跟aadata返回的数组返回的列数一致

### 封装后台分页查询

BaseDaoImpl

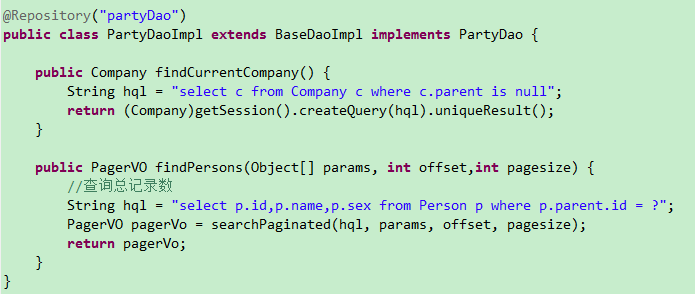




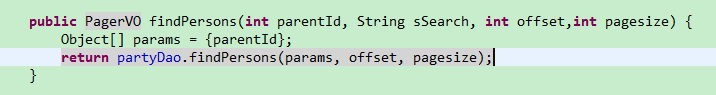




PartyDaoImpl继承BaseDaoImpl



ServiceImpl



### ThreadLocal

#### 概述

ThreadLocal是什么呢？其实ThreadLocal并非是一个线程的本地实现版本，它并不是一个Thread，而是threadlocalvariable(线程局部变量)。也许把它命名为ThreadLocalVar更加合适。线程局部变量(ThreadLocal)其实的功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是Java中一种较为特殊的线程绑定机制，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。

从线程的角度看，每个线程都保持一个对其线程局部变量副本的隐式引用，只要线程是活动的并且 ThreadLocal 实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被垃圾回收（除非存在对这些副本的其他引用）。

通过ThreadLocal存取的数据，总是与当前线程相关，也就是说，JVM 为每个运行的线程，绑定了私有的本地实例存取空间，从而为多线程环境常出现的并发访问问题提供了一种隔离机制。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单，在ThreadLocal类中有一个Map，用于存储每一个线程的变量的副本。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

#### API说明

ThreadLocal()

创建一个线程本地变量。

T get()

返回此线程局部变量的当前线程副本中的值，如果这是线程第一次调用该方法，则创建并初始化此副本。

protected T initialValue()

返回此线程局部变量的当前线程的初始值。最多在每次访问线程来获得每个线程局部变量时调用此方法一次，即线程第一次使用 get() 方法访问变量的时候。如果线程先于 get 方法调用 set(T) 方法，则不会在线程中再调用 initialValue 方法。

若该实现只返回 null；如果程序员希望将线程局部变量初始化为 null 以外的某个值，则必须为 ThreadLocal 创建子类，并重写此方法。通常，将使用匿名内部类。initialValue 的典型实现将调用一个适当的构造方法，并返回新构造的对象。

void remove()

移除此线程局部变量的值。这可能有助于减少线程局部变量的存储需求。如果再次访问此线程局部变量，那么在默认情况下它将拥有其 initialValue。

void set(T value)

将此线程局部变量的当前线程副本中的值设置为指定值。许多应用程序不需要这项功能，它们只依赖于 initialValue() 方法来设置线程局部变量的值。

在程序中一般都重写initialValue方法，以给定一个特定的初始值。

#### 总结

ThreadLocal使用场合主要解决多线程中数据数据因并发产生不一致问题。ThreadLocal为每个线程的中并发访问的数据提供一个副本，通过访问副本来运行业务，这样的结果是耗费了内存，单大大减少了线程同步所带来性能消耗，也减少了线程并发控制的复杂度。

ThreadLocal不能使用原子类型，只能使用Object类型。ThreadLocal的使用比synchronized要简单得多。

ThreadLocal和Synchonized都用于解决多线程并发访问。但是ThreadLocal与synchronized有本质的区 别。synchronized是利用锁的机制，使变量或代码块在某一时该只能被一个线程访问。而ThreadLocal为每一个线程都提供了变量的副本， 使得每个线程在某一时间访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。而Synchronized却正好相反，它用于在多个线程间通 信时能够获得数据共享。

Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。

当然ThreadLocal并不能替代synchronized,它们处理不同的问题域。Synchronized用于实现同步机制，比ThreadLocal更加复杂。

#### ThreadLocal使用的一般步骤

1、在多线程的类（如ThreadDemo类）中，创建一个ThreadLocal对象threadXxx，用来保存线程间需要隔离处理的对象xxx。

2、在ThreadDemo类中，创建一个获取要隔离访问的数据的方法getXxx()，在方法中判断，若ThreadLocal对象为null时候，应该new()一个隔离访问类型的对象，并强制转换为要应用的类型。

3、在ThreadDemo类的run()方法中，通过getXxx()方法获取要操作的数据，这样可以保证每个线程对应一个数据对象，在任何时刻都操作的是这个对象。

#### 定义ThreadLocal

***public class SystemContext {***

***private static ThreadLocal offset = new ThreadLocal();***

***private static ThreadLocal pagesize = new ThreadLocal();***

***public static void setOffset(int \_offset){***

***offset.set(\_offset);***

***}***

***public static int getOffset(){***

***Integer \_offset = (Integer)offset.get();***

***if(\_offset == null){***

***return 0;***

***}***

***return \_offset;***

***}***

***public static void removeOffset(){***

***offset.remove();***

***}***

***public static void setPagesize(int \_pagesize){***

***pagesize.set(\_pagesize);***

***}***

***public static int getPagesize(){***

***Integer \_pagesize = (Integer)pagesize.get();***

***if(\_pagesize == null){***

***return Integer.MAX\_VALUE;***

***}***

***return \_pagesize;***

***}***

***public static void removePagesize(){***

***pagesize.remove();***

***}***

***}***

定义拦截器

***public class PagerInterceptor extends AbstractInterceptor {***

***@Override***

***public String intercept(ActionInvocation invocation) throws Exception { SystemContext.setOffset(getRequestOffset(ServletActionContext.getRequest()));***

***SystemContext.setPagesize(getRequestPagesize(ServletActionContext.getRequest()));***

***try{***

***return invocation.invoke();***

***}finally{***

***//使用完后把参数移除掉***

***SystemContext.removeOffset();***

***SystemContext.removePagesize();***

***}***

***}***

***private int getRequestOffset(HttpServletRequest request){***

***try {***

***System.out.println(Integer.parseInt(request.getParameter("iDisplayStart")));***

***return Integer.parseInt(request.getParameter("iDisplayStart"));***

***} catch (NumberFormatException ignore) {***

***return 0;***

***}***

***}***

***private int getRequestPagesize(HttpServletRequest request){***

***try {***

***return Integer.parseInt(request.getParameter("iDisplayLength"));***

***} catch (NumberFormatException ignore) {***

***return Integer.MAX\_VALUE;***

***}***

***}***

BaseDaoImpl实现从threadLocal取参数

***public PagerVO searchPaginated(String hql){***

***return searchPaginated(hql, null);***

***}***

***public PagerVO searchPaginated(String hql,Object param){***

***return searchPaginated(hql, new Object[]{param});***

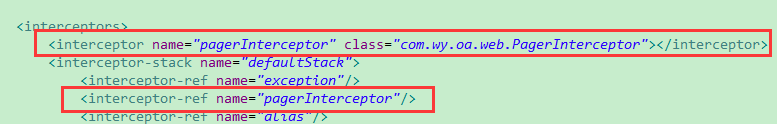
***}***

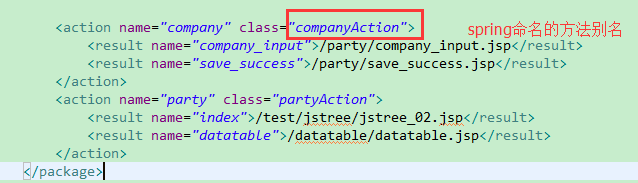
***public PagerVO searchPaginated(String hql,Object[] params){***

***return searchPaginated(hql, params, SystemContext.getOffset(), SystemContext.getPagesize());***

***}***

配置拦截器





默认拦截器不需要配置

## 权限控制

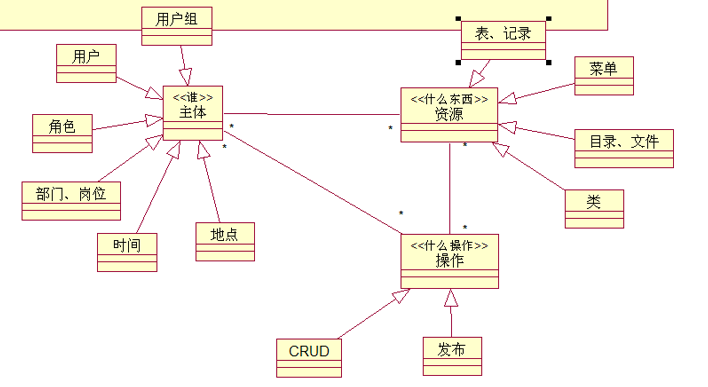
### 权限控制三要素

权限控制：控制主体对资源做什么操作，主体、资源、操作

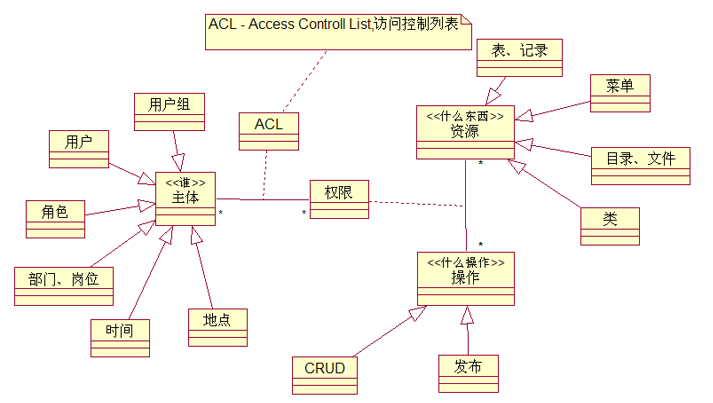
授权：授予主体某权限（建立某主体和某权限之间的关联）

认证：判断某主体是否拥有某权限（查询某主体和某权限之间是否存在关联）

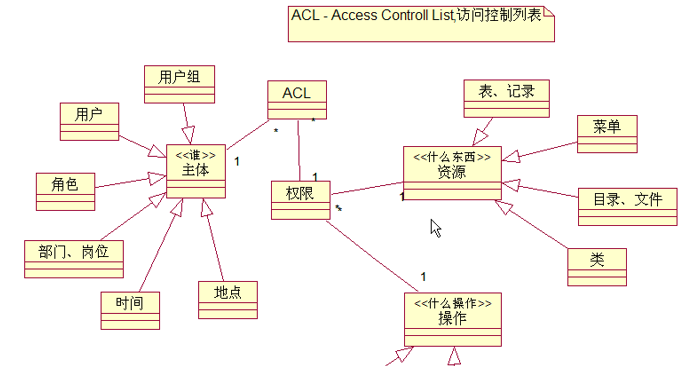
权限：就是对什么东西做什么操作（资源+操作）



权限就是操作和资源之间的多对多关联，授权就是建立主体和权限之间的多对多关联



多对多的关联被拆分成两个多对一，中间为多两边为一



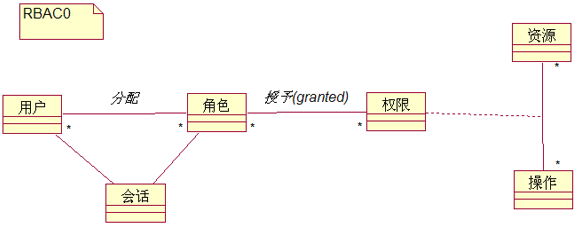
数据库表



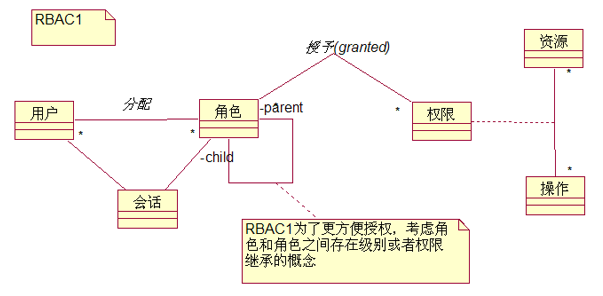
### RBAC

Role-Based Access Control

#### RBAC0

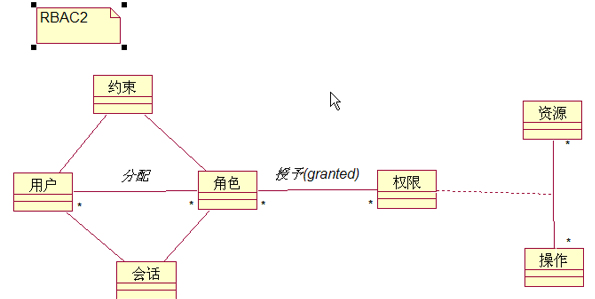


#### RBAC1



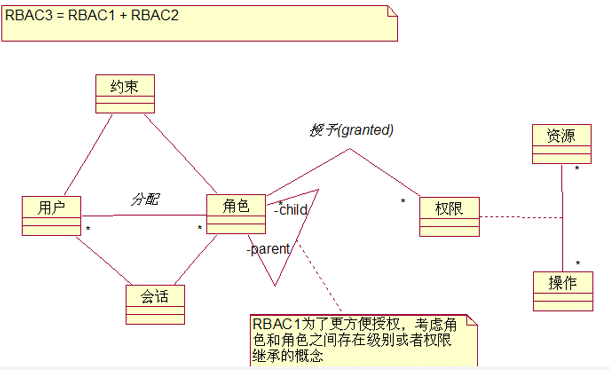
#### RBAC2

当一个用户的多个角色之间有冲突时，建立约束关系，避免冲突

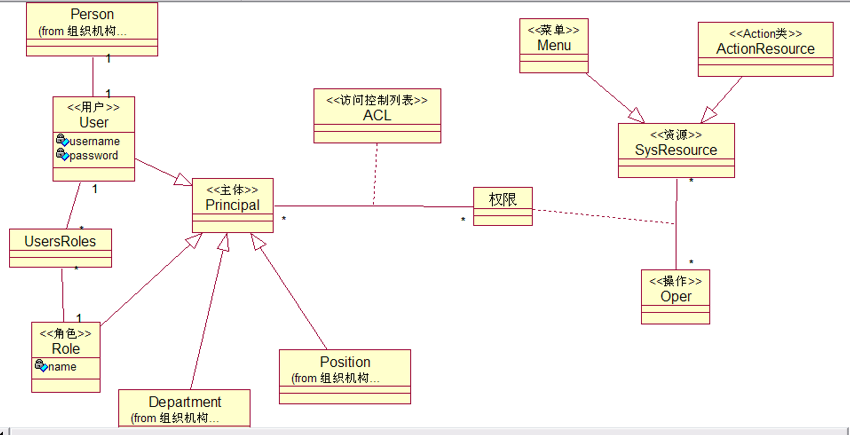


#### RBAC3

RBAC3=RBAC1+RBAC2



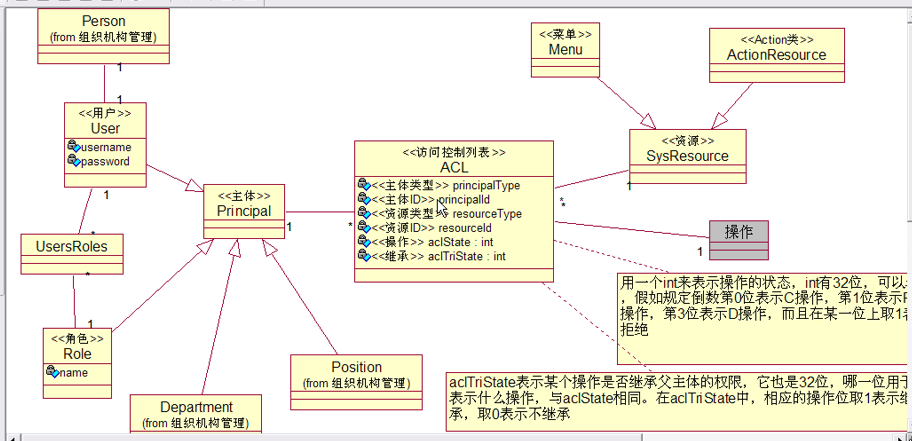
### 权限管理建模



### 权限继承

1. 某一主体可继承父主体（部门、人、岗位）的权限。
2. 也可以直接给用户授权。
3. 父主体之间的授权有冲突，以拒绝为准。
4. 子主体授权可继承或屏蔽父主体的授权。

一个人的多个角色有冲突时，采用“拒绝”设置。



Action类看成资源，