# 开发笔记代码整理记录

1.我们需要成为一个非常具有竞争力的JAVA程序员

2.我们项目上可能会遇到很多困难一定坚持到开窍

3.不停的拓展自己的编程思想和经验值

4.不懂所以从轮子开始，再一步一步分析他的过程

## 1商品代码开发学习：

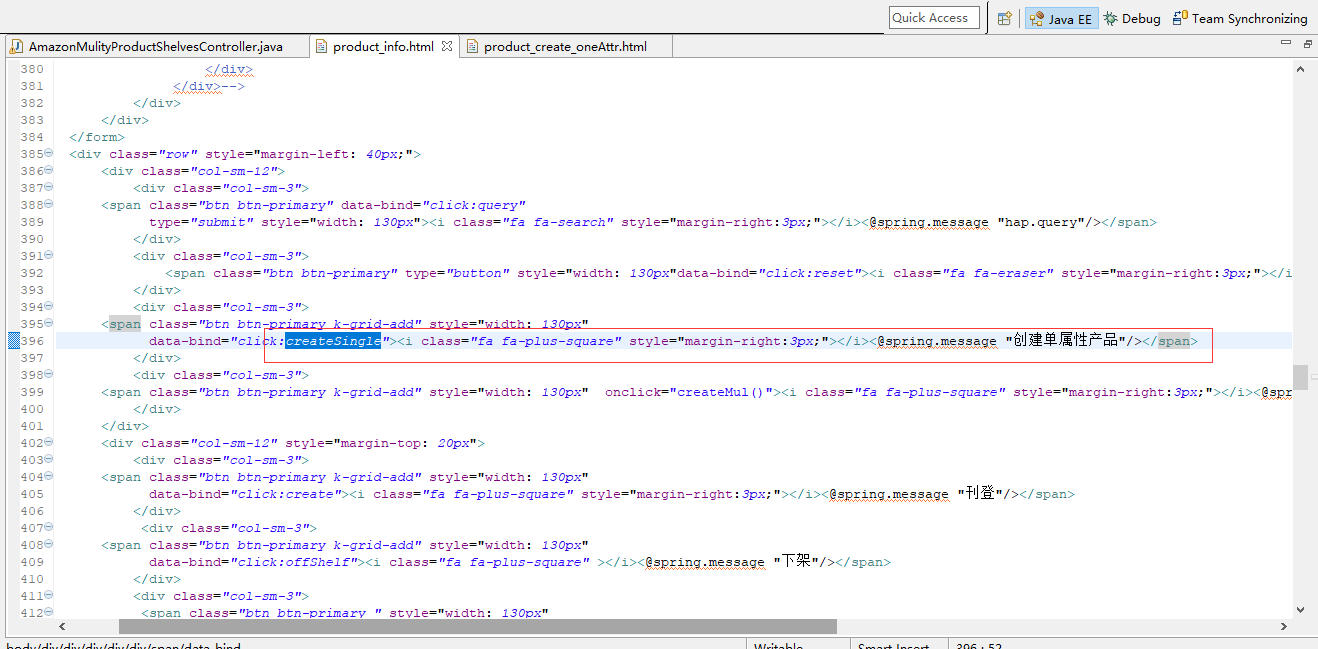
#### 商品管理

##### 我们需要注意的代码逻辑，学会使用此框架

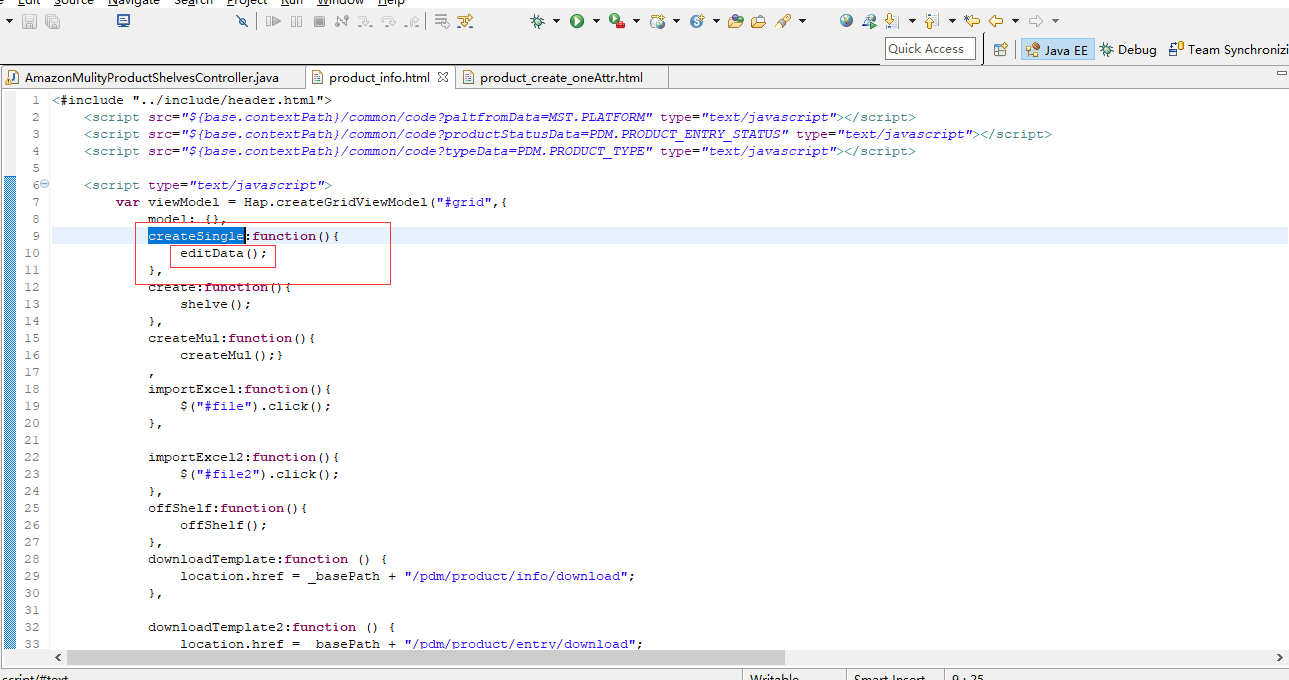




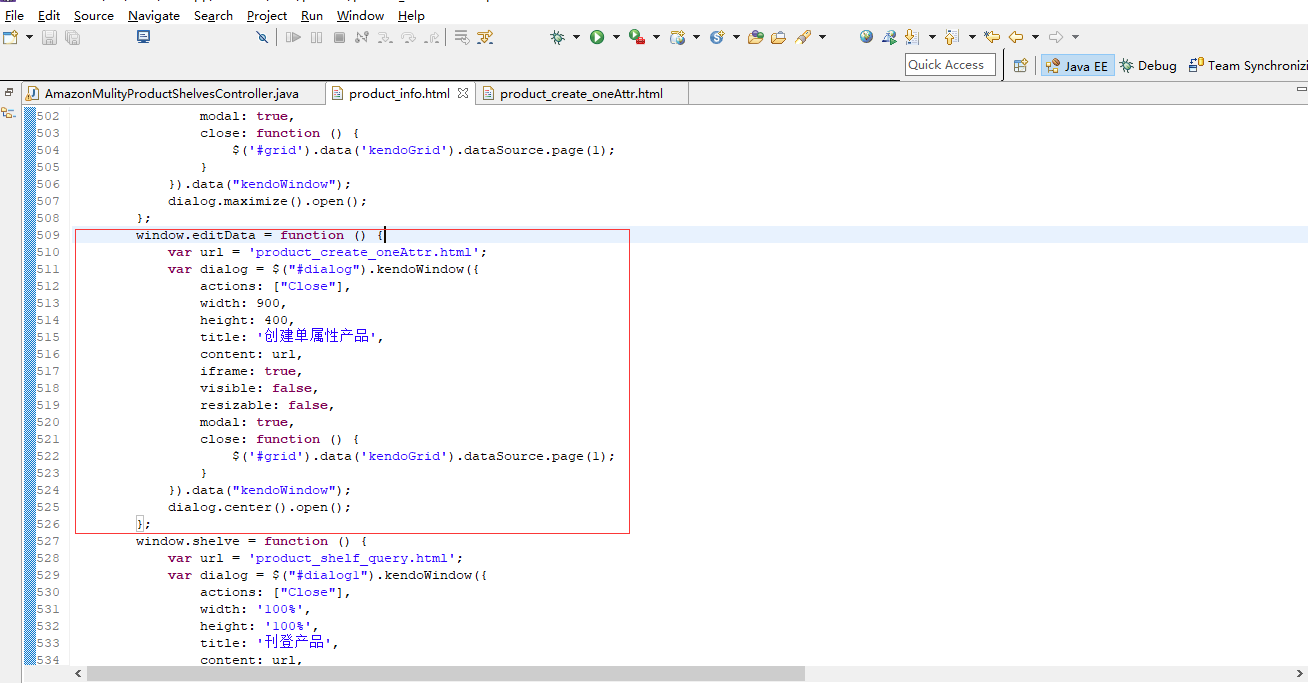
##### product\_info.html



##### 触发事件



##### 我们是通过div给他赋值一个window控件



##### 跳转页面url。

##### $("#dialog") 赋值一个 将一个window赋值给一个 div

window.editData = function () {

var url = 'product\_create\_oneAttr.html';

var dialog = $("#dialog").kendoWindow({

actions: ["Close"],

width: 900,

height: 400,

title: '创建单属性产品',

content: url,

iframe: true,

visible: false,

resizable: false,

modal: true,

close: function () {

$('#grid').data('kendoGrid').dataSource.page(1);

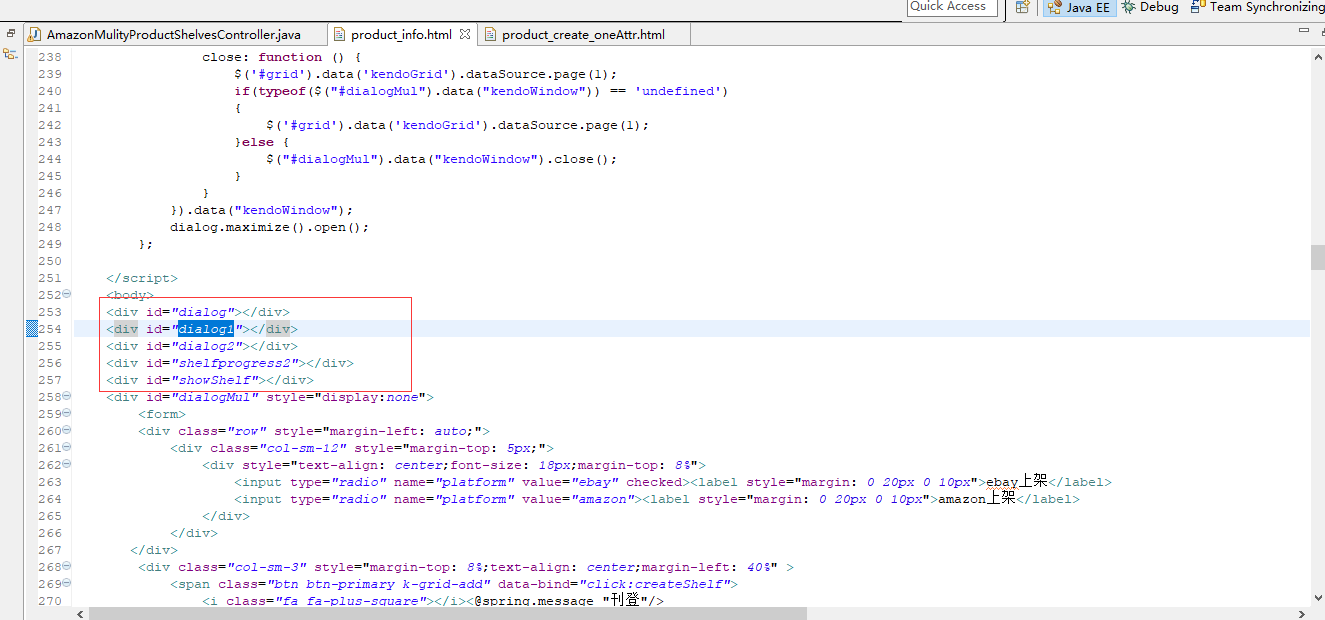
}

}).data("kendoWindow");

dialog.center().open();

};

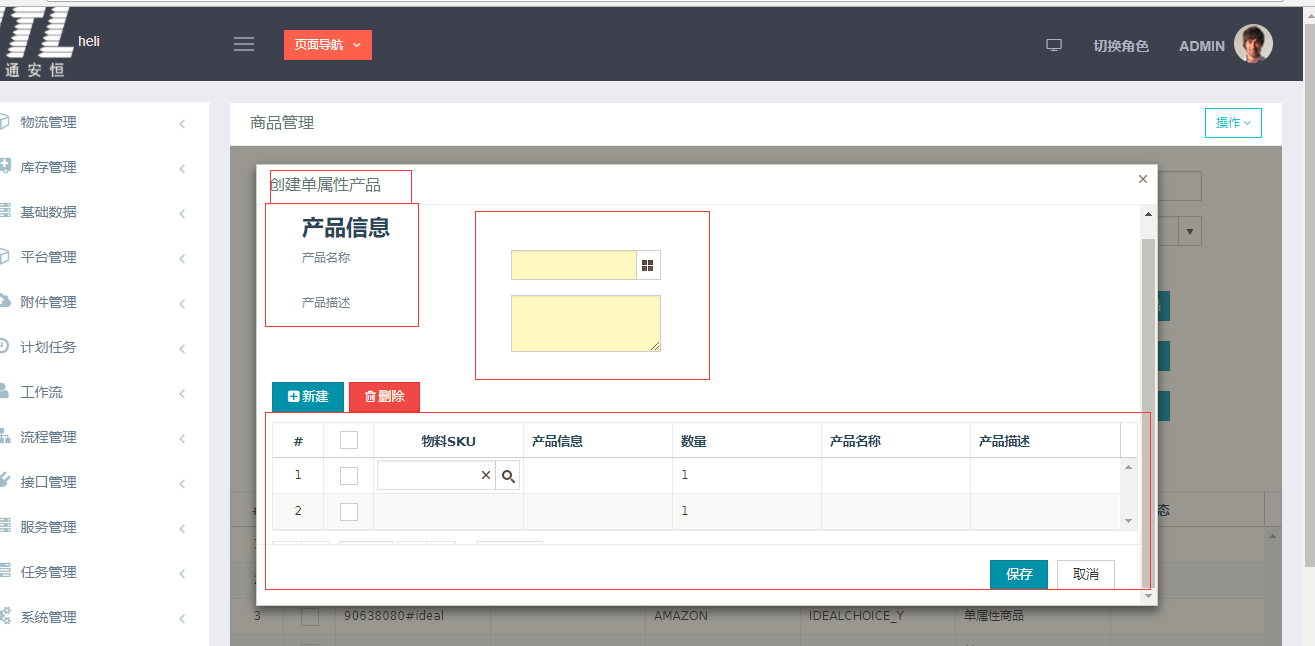
##### $("#dialog") 赋值一个 将一个window赋值给一个 div



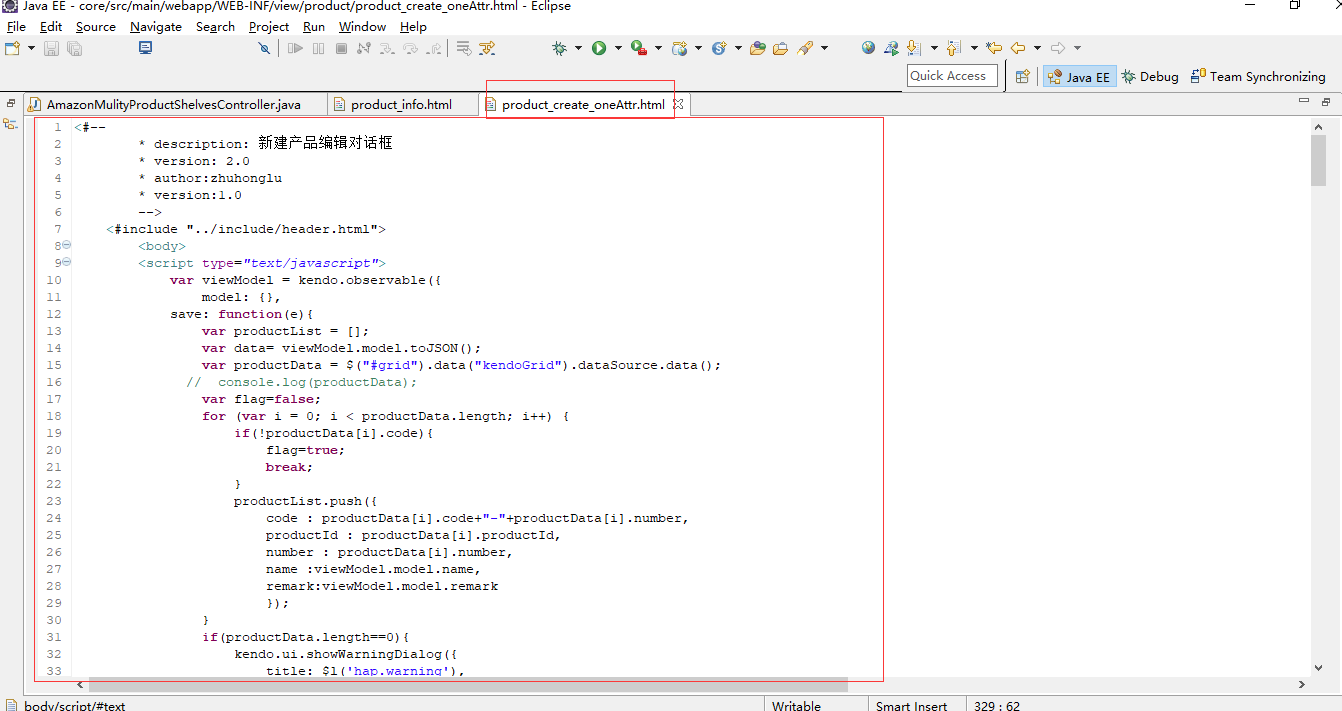
##### 另一个页面跳转

##### 跳转页面url。product\_create\_oneAttr.html

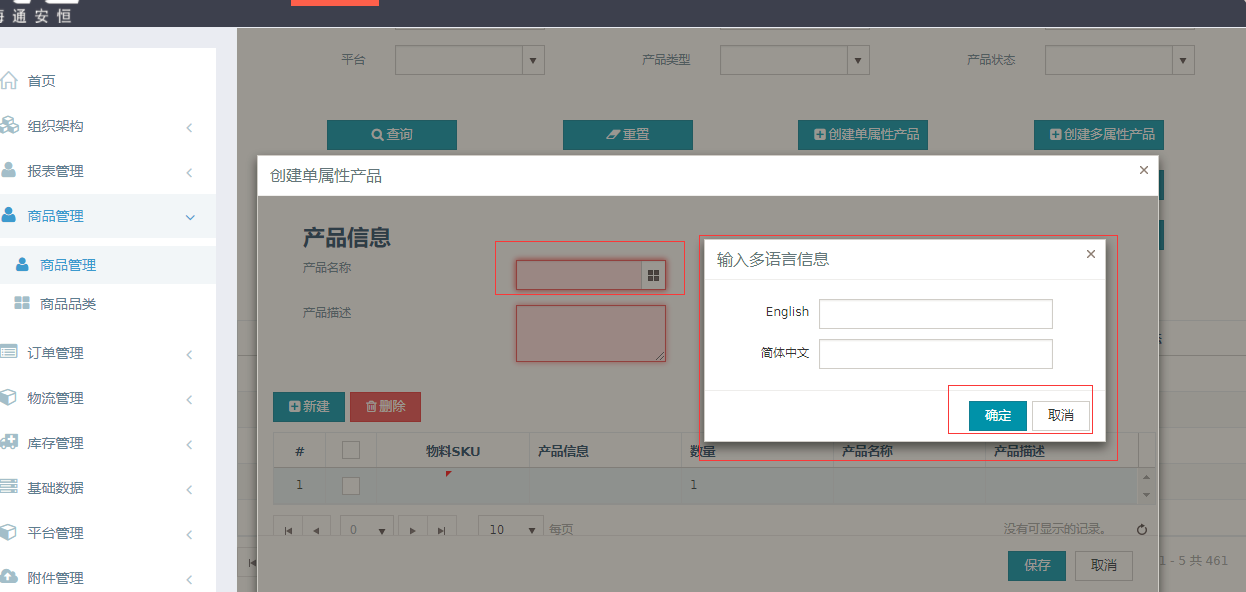
界面



##### 代码和名称 product\_create\_oneAttr.html



##### 小弹框





<div class="col-sm-12" style="margin-bottom: 15px;">

<div class="col-sm-3">

<label ><@spring.message"产品名称"/></label>

</div>

<div class="col-sm-8">

<input id="name" name="name" type="text"

data-bind="value:model.name"required

style="width: 150px;>

<span class="k-invalid-msg" data-for="name"></span>

<script>

$("#name").kendoTLEdit({

idField: 'productId',

field: 'name',

dto: "com.heli.product.dto.ProductInfo",

model: viewModel.model

})

kendo.bind($('#name'), viewModel);

</script>

</div>

</div>

##### 在这个input框中做了非必填限制





<div class="col-sm-12" style="margin-bottom: 15px;">

<div class="col-sm-3">

<label ><@spring.message"产品描述"/></label>

</div>

<div class="col-sm-8">

<textarea rows="3" required cols="20" data-role="maskedtextbox" style="width: 150px;

margin-right:5px;" data-bind="value:model.remark" class="k-textbox">

</textarea>

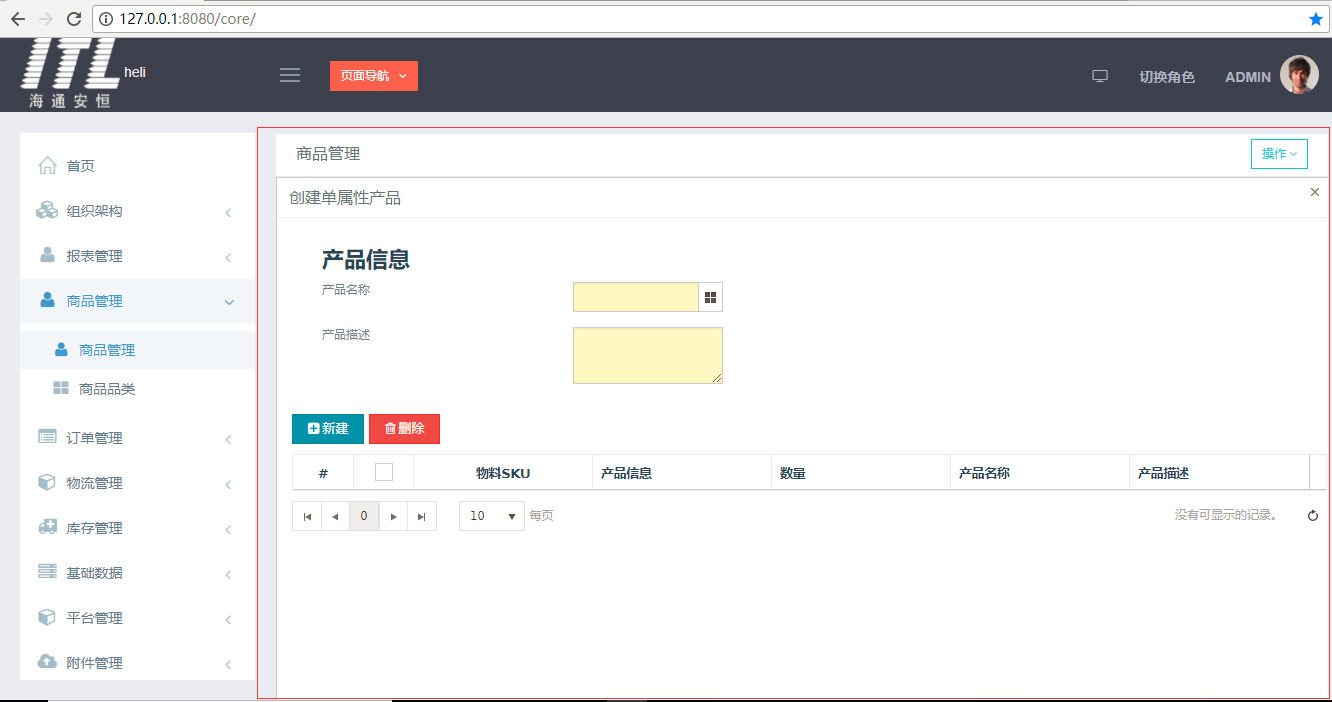
</div>

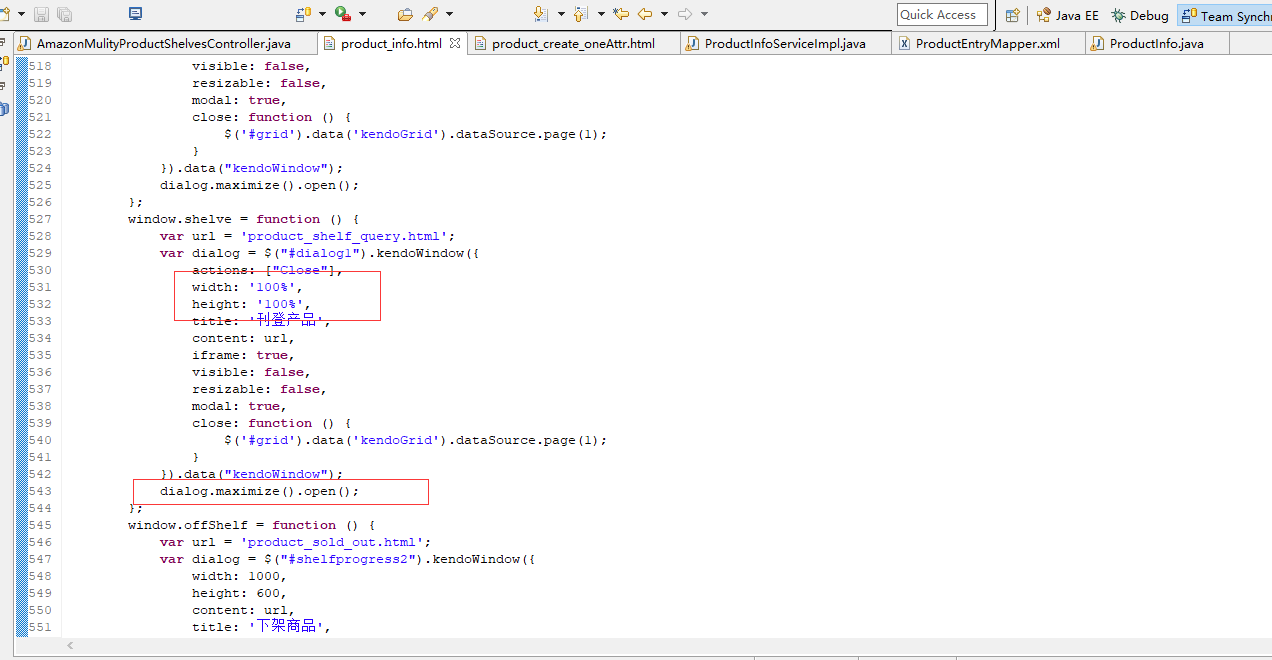
</div>

-----------------------------------------------------------------

##### dialog.maximize().open();

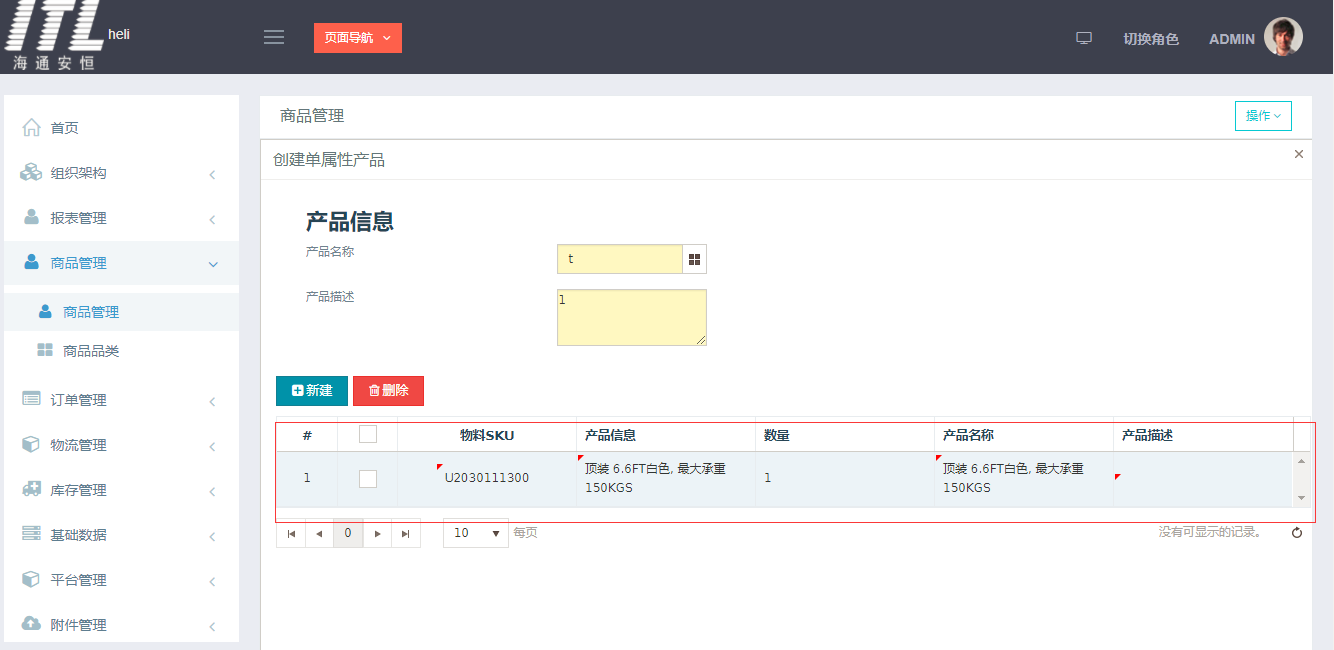
使其全屏显示





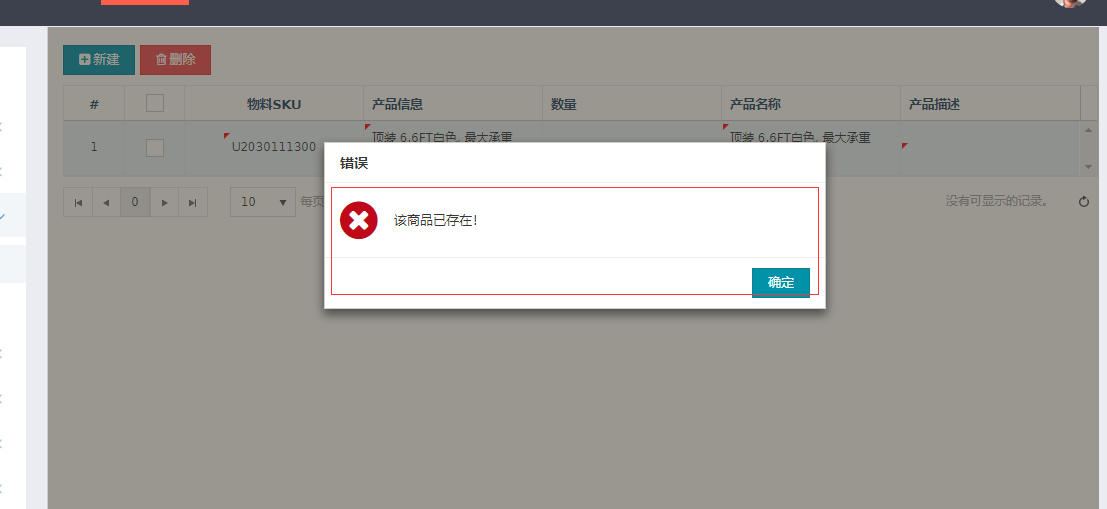
-----------------------------------------------------------------

##### 保存逻辑

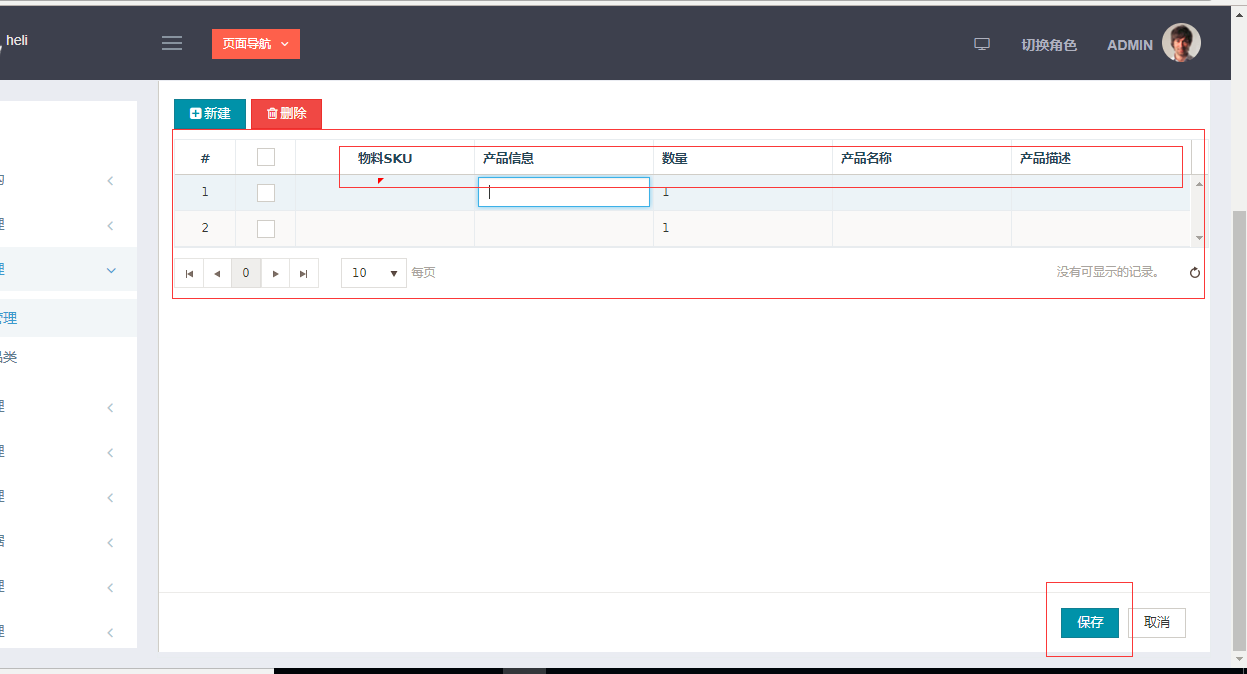


我们使用物料sku的lov选择后，自动带出 sku,产品信息，数量，产品名称，描述。

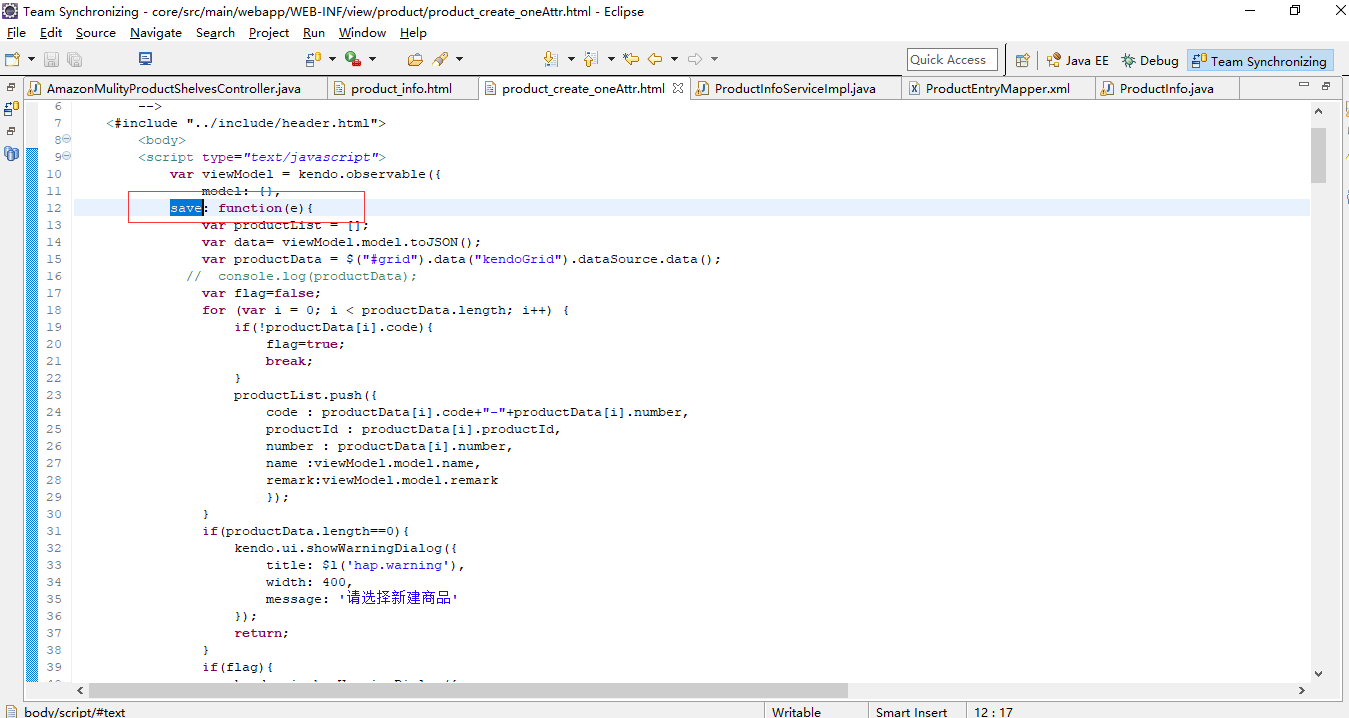
数据自动带出。



相同产品无法保存。

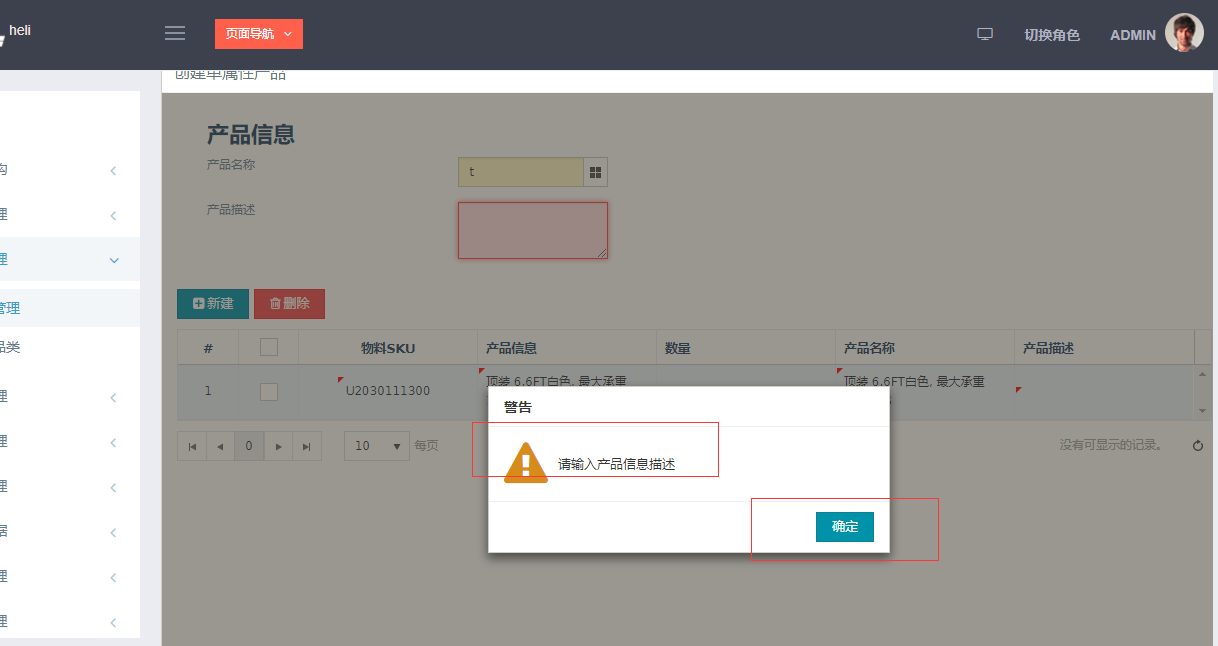






前端参数传递：是通过一个集合

首先对参数检验后存入这个集合中，ajax提交整个集合。



弹出不能为空提交框。



##### 走到这一步，将data里边的状态码 设置为 add

data.\_\_status = "add";

##### 这里将所有参数拿到 XXX.data();

var productData = $("#grid").data("kendoGrid").dataSource.data();

##### 遍历集合 添加进入

for (var i = 0; i < productData.length; i++) {

##### 如果code不存在 就直接跳出

if(!productData[i].code){

flag=true;

break;

}

productList.push({

code : productData[i].code+"-"+productData[i].number,

productId : productData[i].productId,

number : productData[i].number,

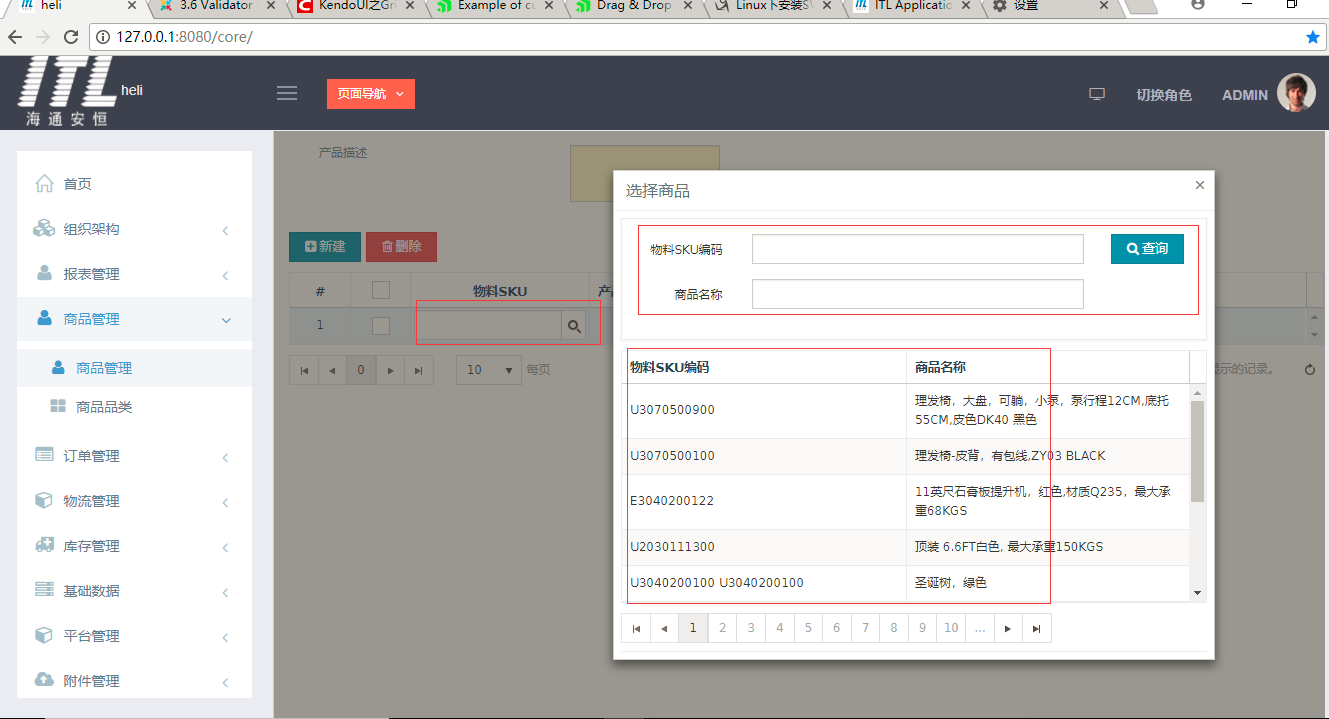
name :viewModel.model.name,

remark:viewModel.model.remark

});

}

##### Lov逻辑



##### 多语言显示 lov

query:function(e){

e.param['locale'] = '${base.locale}';

},

##### 设置小弹框，使其只选择一行



if(grid.selectedDataItems().length == 1){

}else{

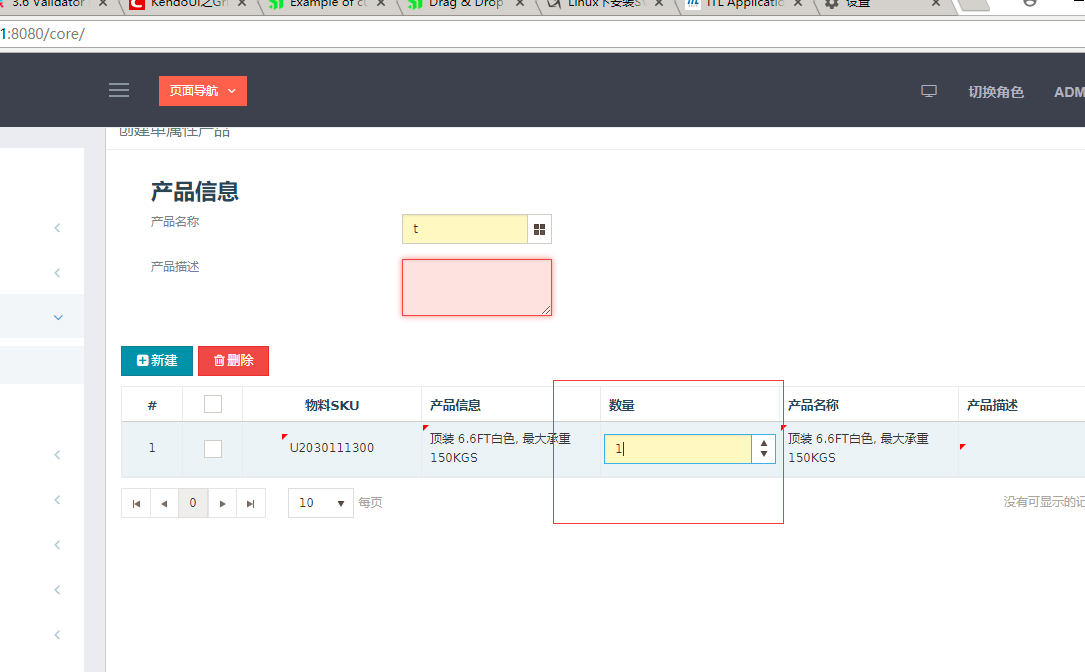
kendo.ui.showInfoDialog({

message : $l('hap.tip.selectrow')

});

}

##### 显示小数点



{

field: "number",

title: '数量',

editor:function(container, options){

$('<input required name="' + options.field + '"/>').appendTo(container).kendoNumericTextBox({

format: "#",

decimals: 1,

min:1,

});

},

width: 120

},

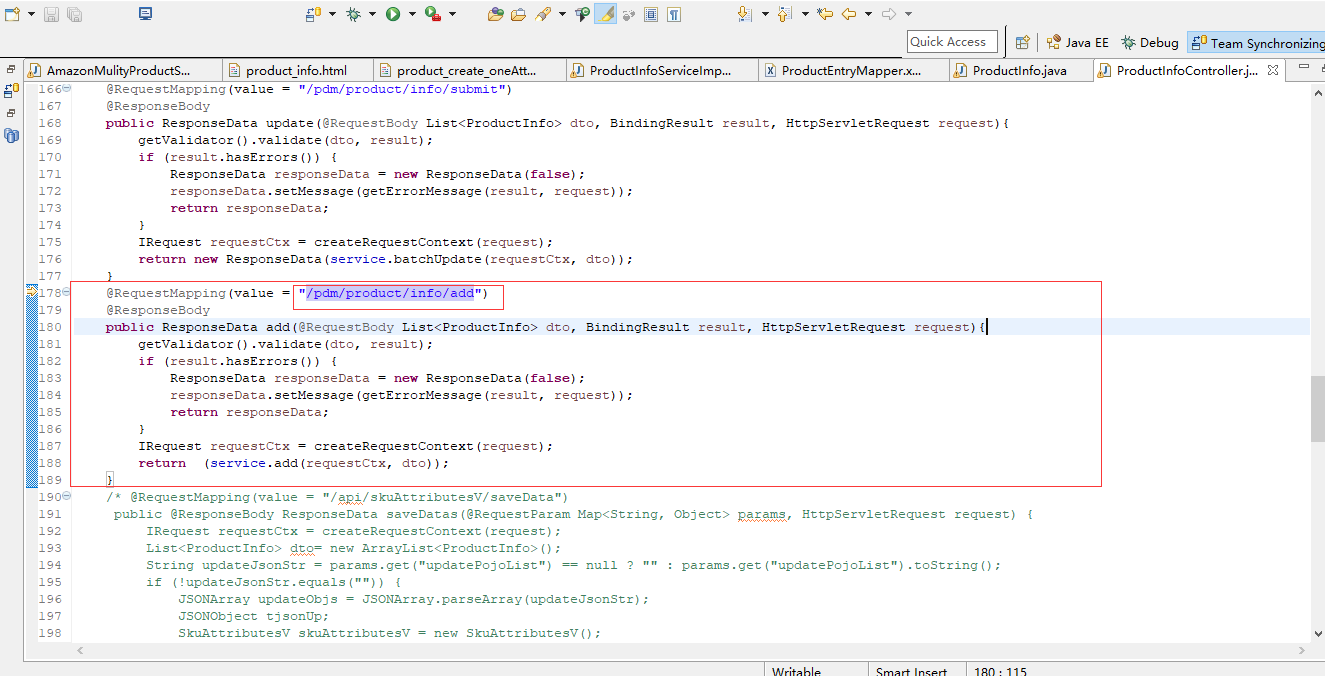
##### 成功回调后，关闭窗口

success: function (data) {

window.parent.$("#dialog").data("kendoWindow").close();

}

##### 这儿ajax访问url到 Controller



##### Service层

-----------------------------------------------------------------

-----------------------------------------------------------------

## 2，代码开发

### 商品管理



### 国际化显示 多语言表

涉及lov显示 info表

Map<String, String> lang = new HashMap<String, String>();

lang.put("zh\_CN", productInfoErp.getMaterialDesc());

lang.put("en\_GB", productInfoErp.getMaterialDescEn());

Map<String, Map<String, String>> tls = new HashMap<String, Map<String, String>>();

tls.put(ProductInfo.FIELD\_NAME, lang);

productInfo.set\_\_tls(tls);

Map<String, String> lang = new HashMap<String, String>();

lang.put("zh\_CN", name);

lang.put("en\_GB", name);

Map<String, Map<String, String>> tls = new HashMap<String, Map<String, String>>();

tls.put(ProductInfo.FIELD\_NAME, lang);

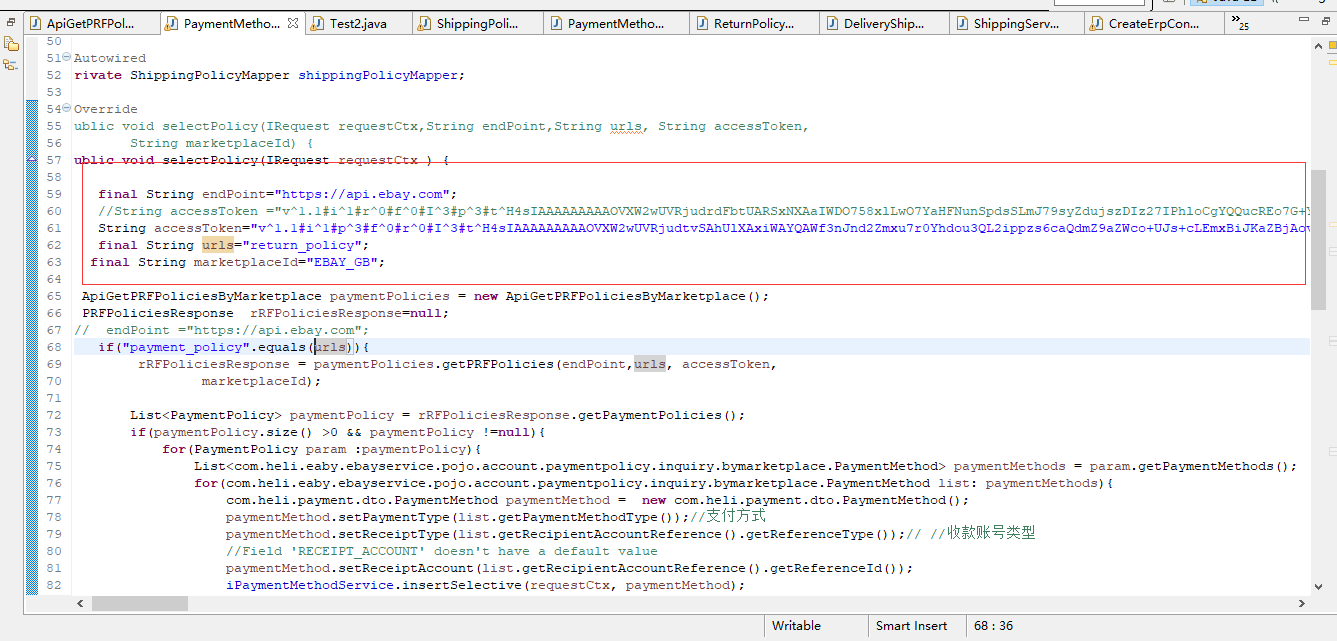
shelvesProductInfo1.set\_\_tls(tls);

## eBay接口开发

测试接口

<http://127.0.0.1:8080/core/api/public/mst/payment/method/selectPolicy>

把控制层形参，service接口全部去掉。直接使用这等参数即可。

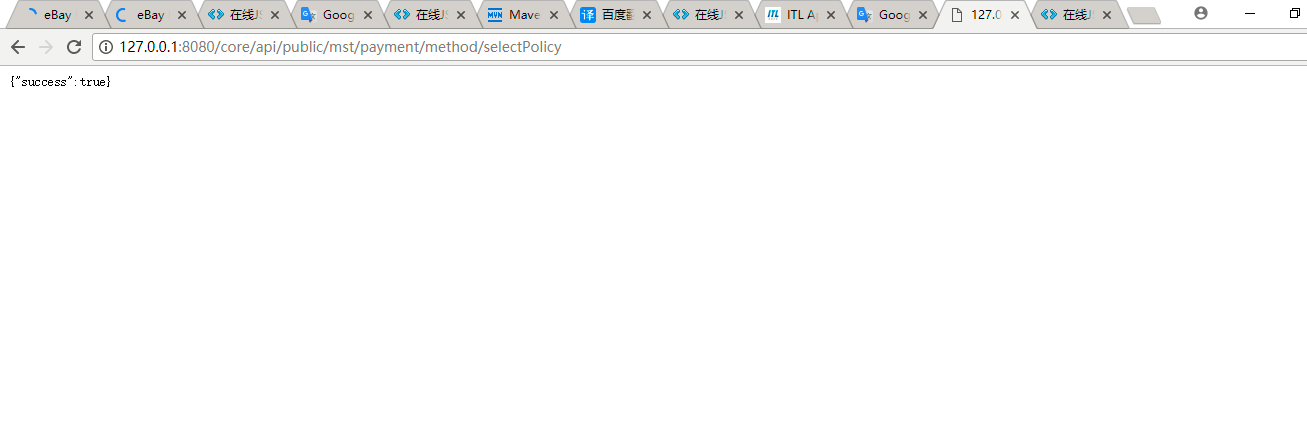


返回成功在此

我的eBay接口账号：

[Shelianglu@outlook.com](mailto:Shelianglu@outlook.com)

123520ily!



## 开发代码：

Dto与model复制以及赋值

## 导入导出

要一份完整的excle数据，

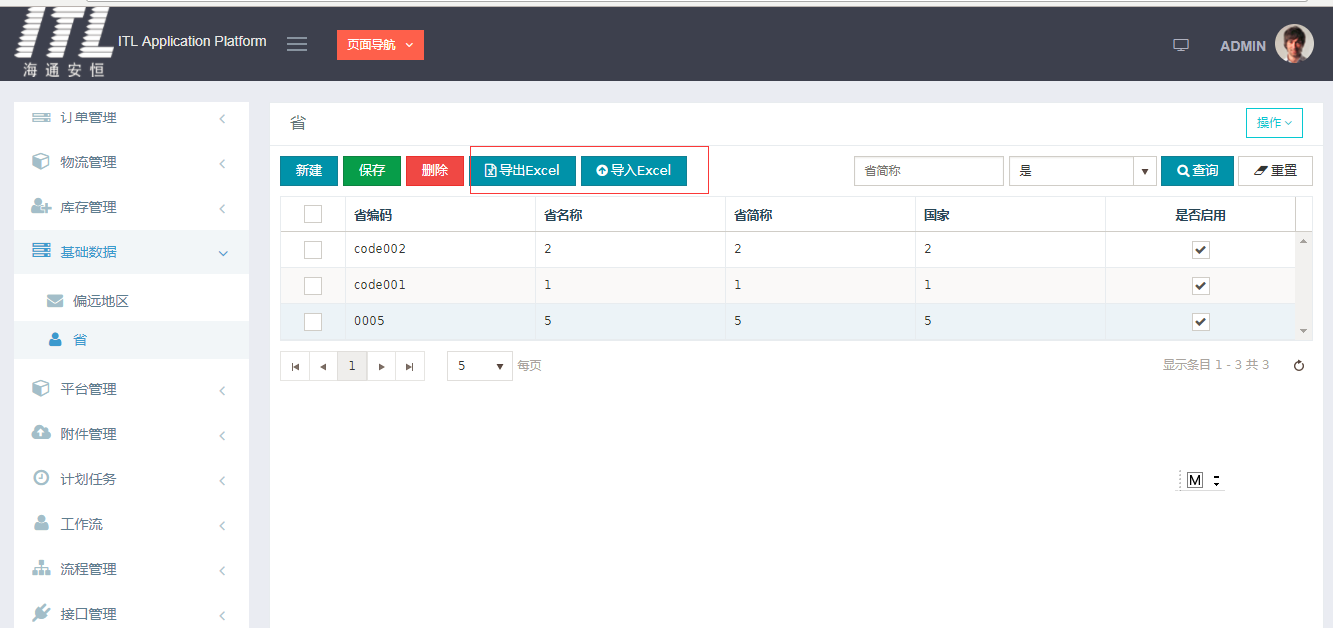
查看数据库字段，约束，长度。

考虑具体的细节，错误等 空指针，类型转换失败，数据库字段不够长 等等 抛出异常，需要显示的信息 。

必填和非必填。

Excle模板名称和dto实体类名称要一致。

需要把所有问题要考虑到



### 前端代码html

<span class=*"btn btn-primary "* style="float:*left*;margin-right:*5px*;" onclick='Hap.exportExcel("MST\_PROVINCE","${base.contextPath}/mst/province/query")'><i class=*"fa fa-file-excel-o"* style="margin-right:*3px*;"></i><@spring.message "hap.exportexcel"/></span>

<span class=*"btn btn-primary k-grid-excel"* style="float:*left*;margin-right:*5px*" onclick='Hap.importExcel("MST\_PROVINCE")' ><i class=*"fa fa-arrow-circle-up"* style="margin-right:*3px*;"></i><@spring.message "excel.import"/></span>

### 后台代码Controller

@RequestMapping(value = "/mst/province/query")

@ResponseBody

@ExcelExport(

table = Province.**class**

)

**public** ResponseData query(Province dto,@RequestParam(defaultValue = "1") **int** page,

@RequestParam(defaultValue = "10") **int** pagesize, HttpServletRequest request,HttpServletResponse response) {

IRequest requestContext = createRequestContext(request);

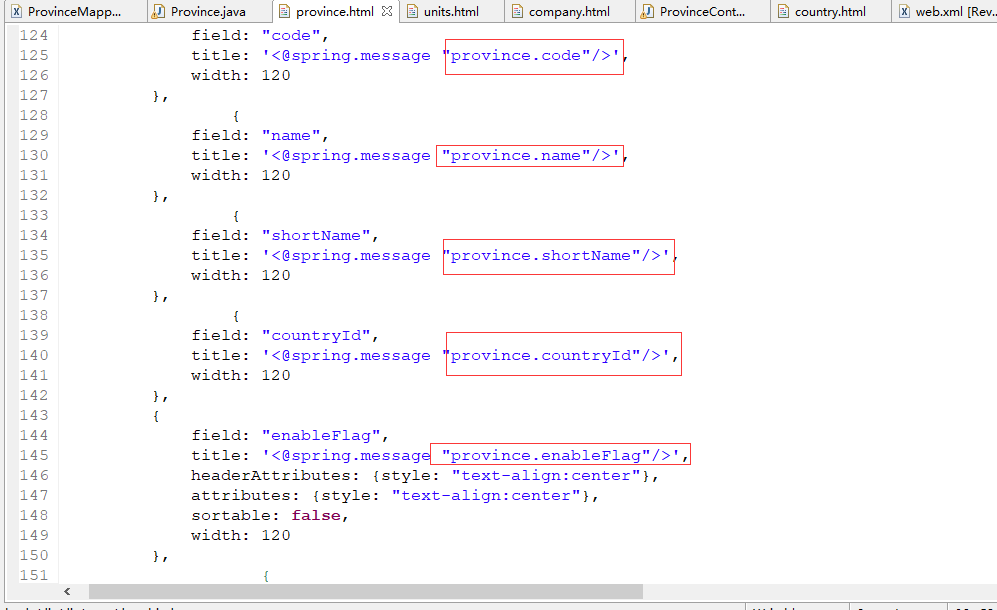
**return** **new** ResponseData(service.selectOptions(requestContext, dto, **null**, Integer.*valueOf*(page), Integer.*valueOf*(pagesize)));

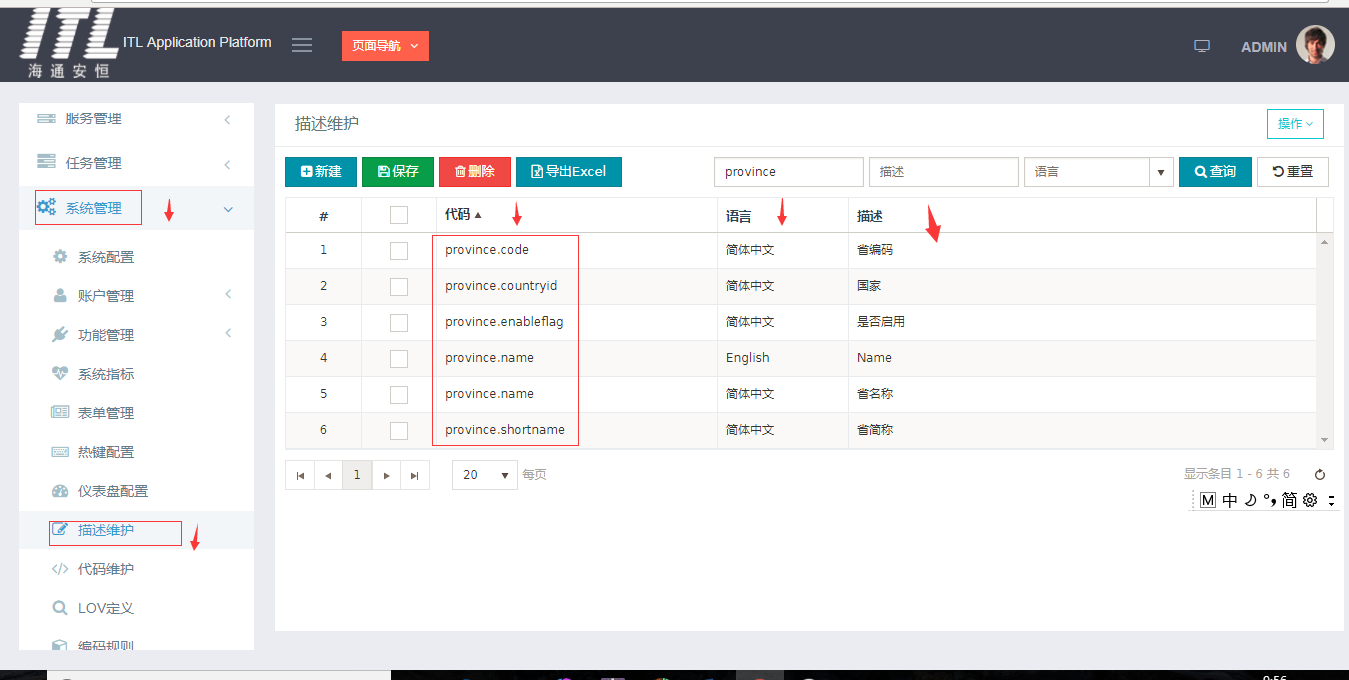
// return new ResponseData(service.select(requestContext,dto,page,pageSize));

}

### 还有一个问题就是我们必须维护中英文。不然导出会出错的。

我们今后模块最好如此。





导入不用框架的

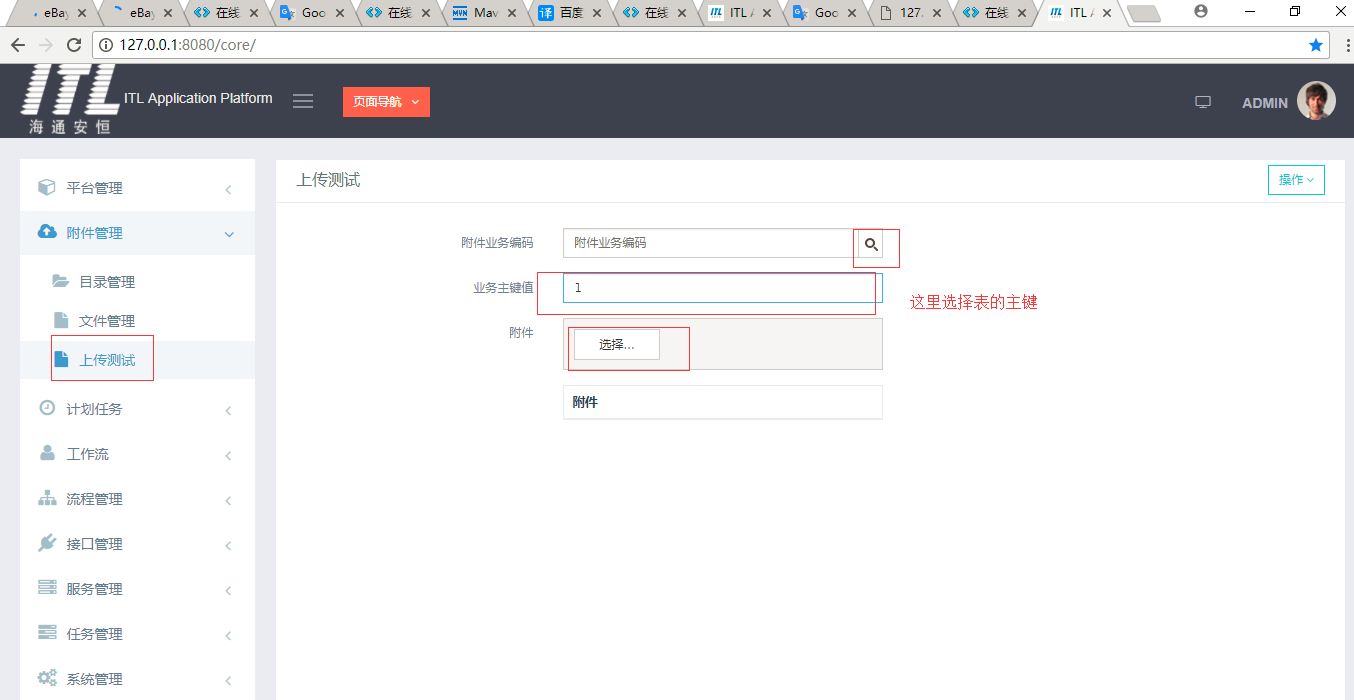
写一个代码模板出来  
用poi导入

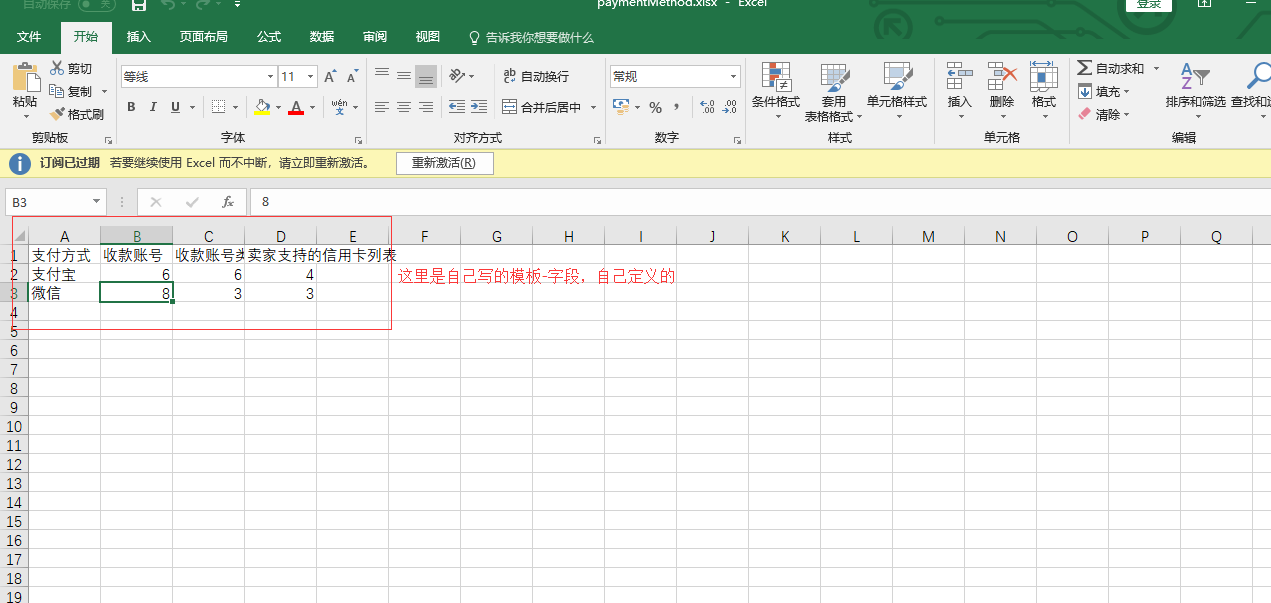
## 新的下载模板（以海龙哥规范）

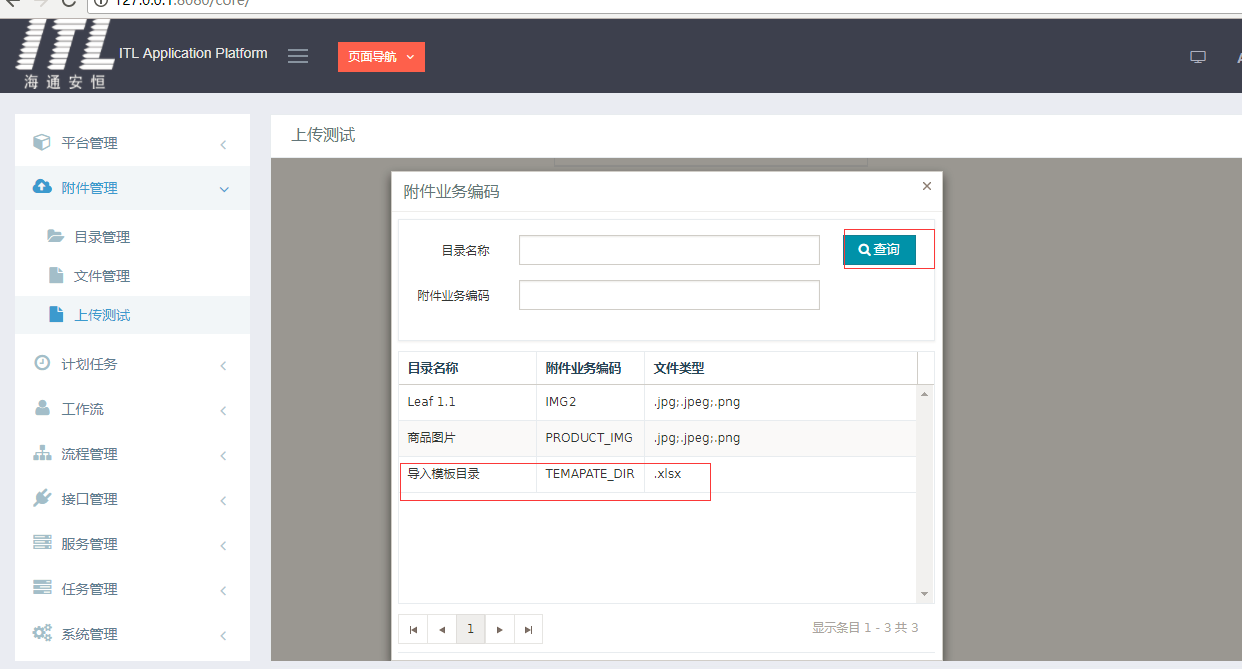
导出数据还需导出



维护主键



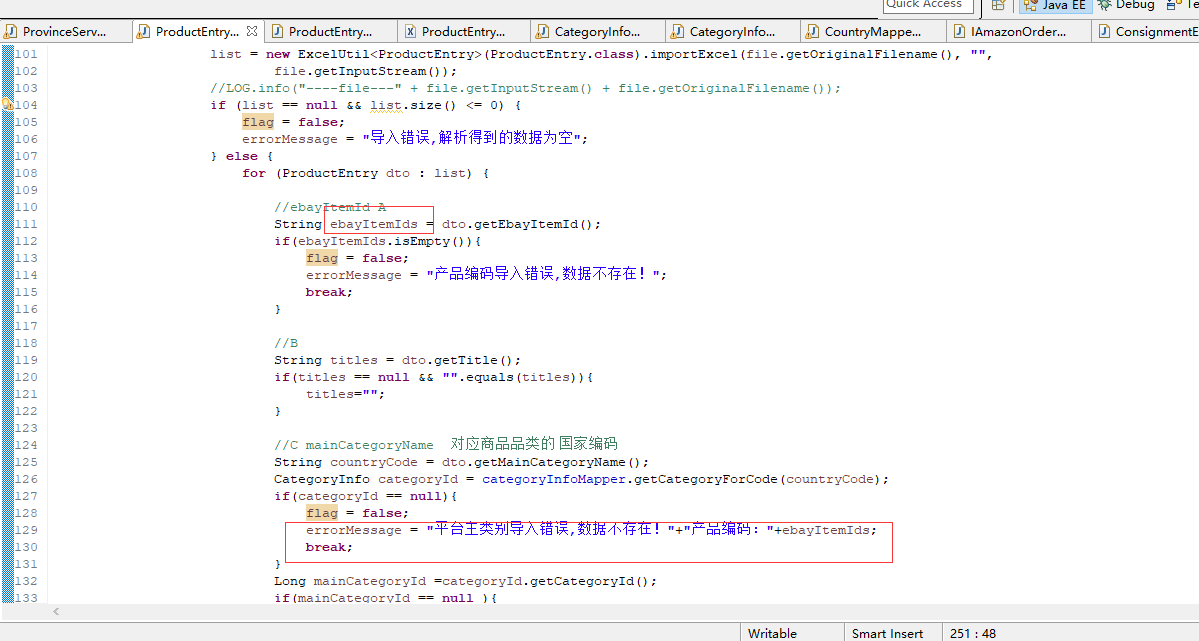


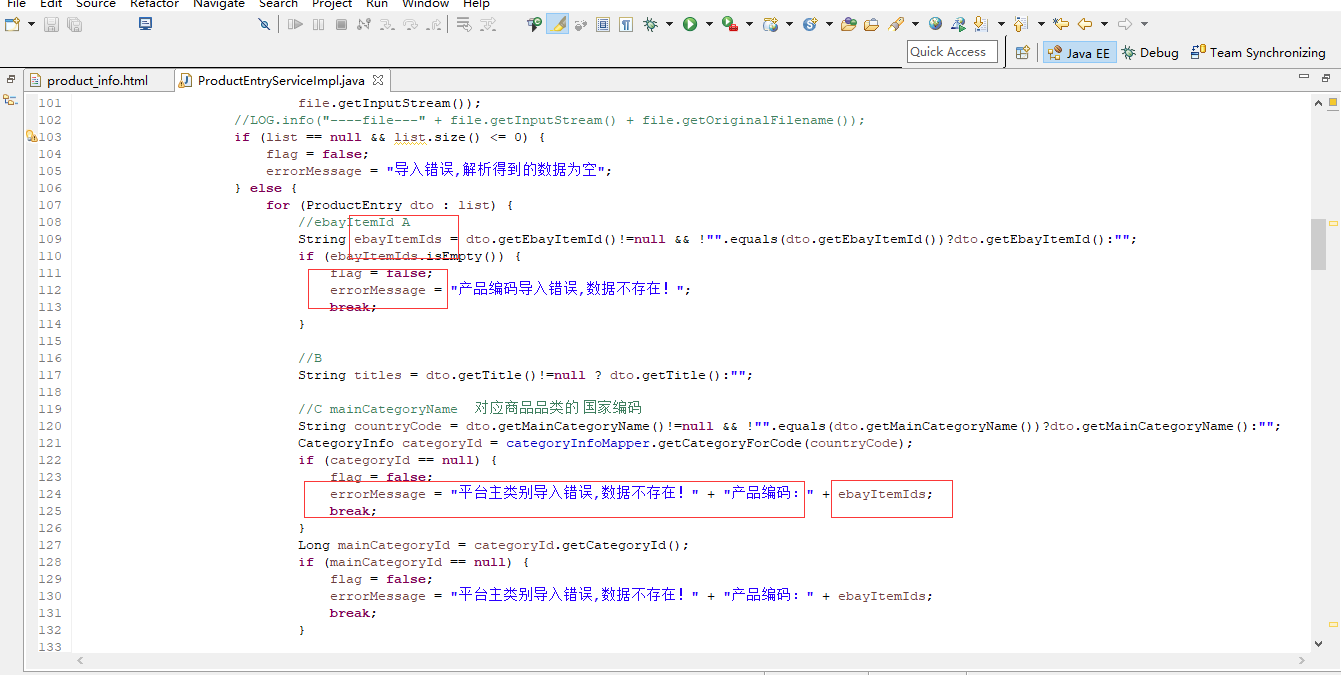


返回错误消息 并告知是那个行出了问题 避免errorMessage消息被覆盖了。

Break. 跳出。

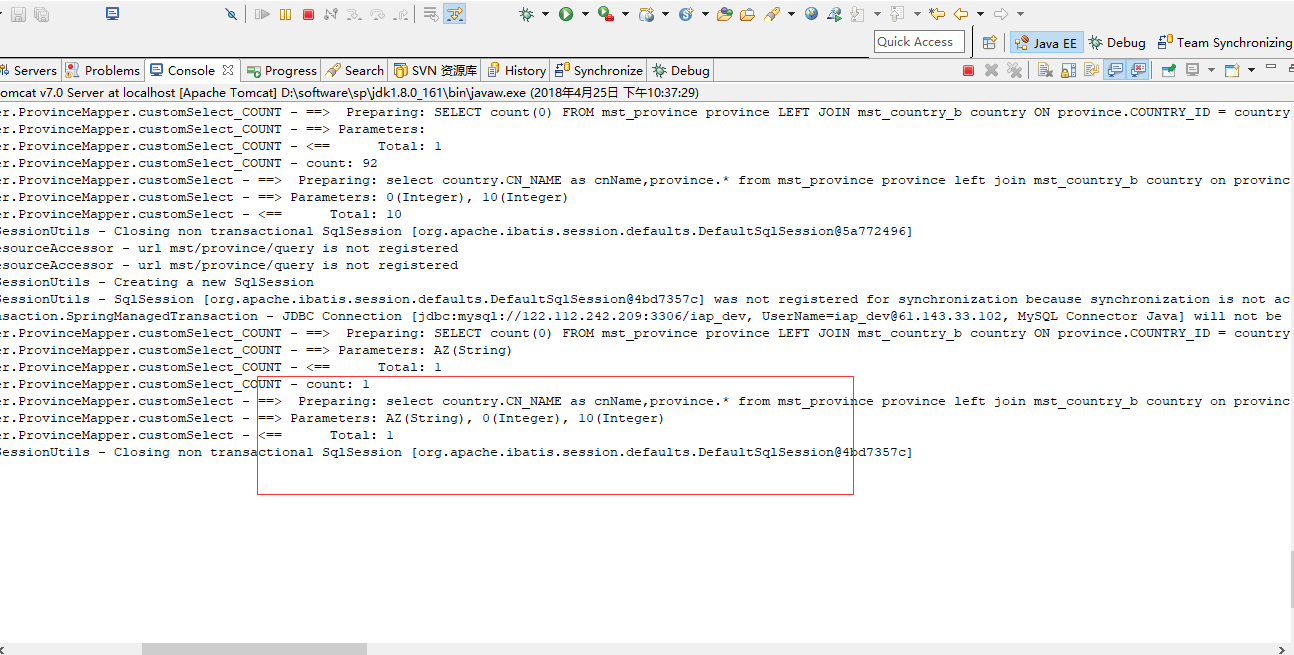
我们还需进行判断为空，这是必不可少的啊。





## 我们的控制台 查看sql执行计划

返回总数

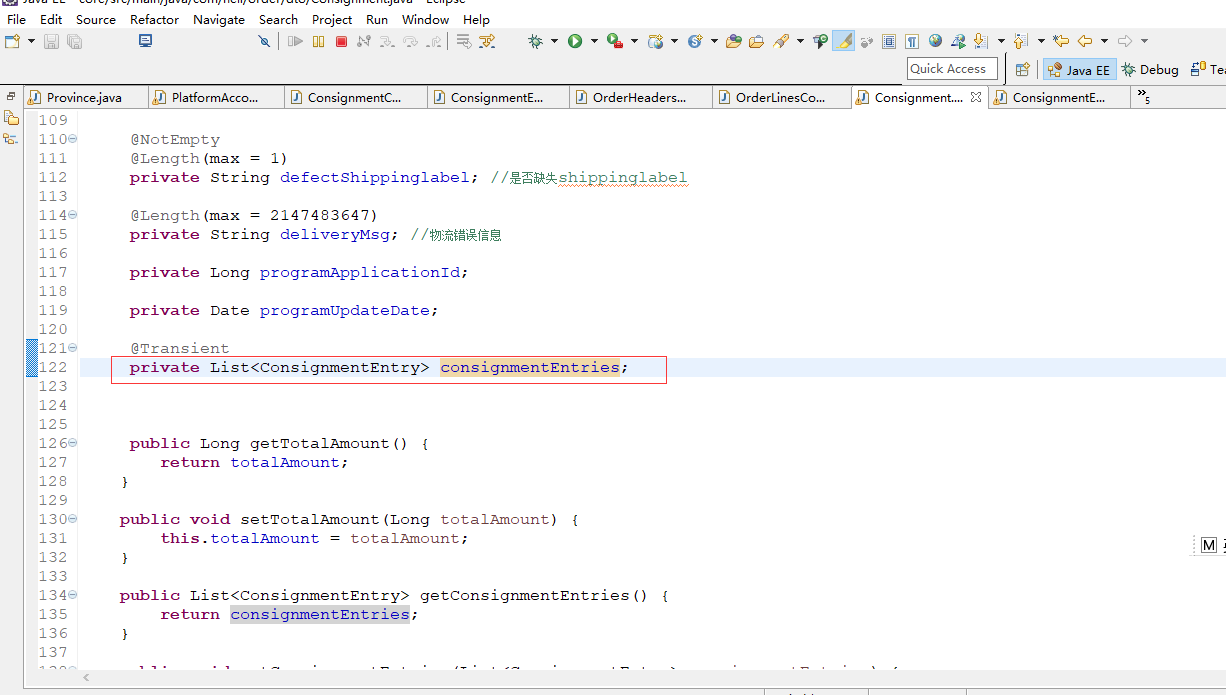


# 框架使用记录

## Dto注解使用

### @Transient

如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient 后面就不要用实体去查询了，在dao类中写一个多表联查。



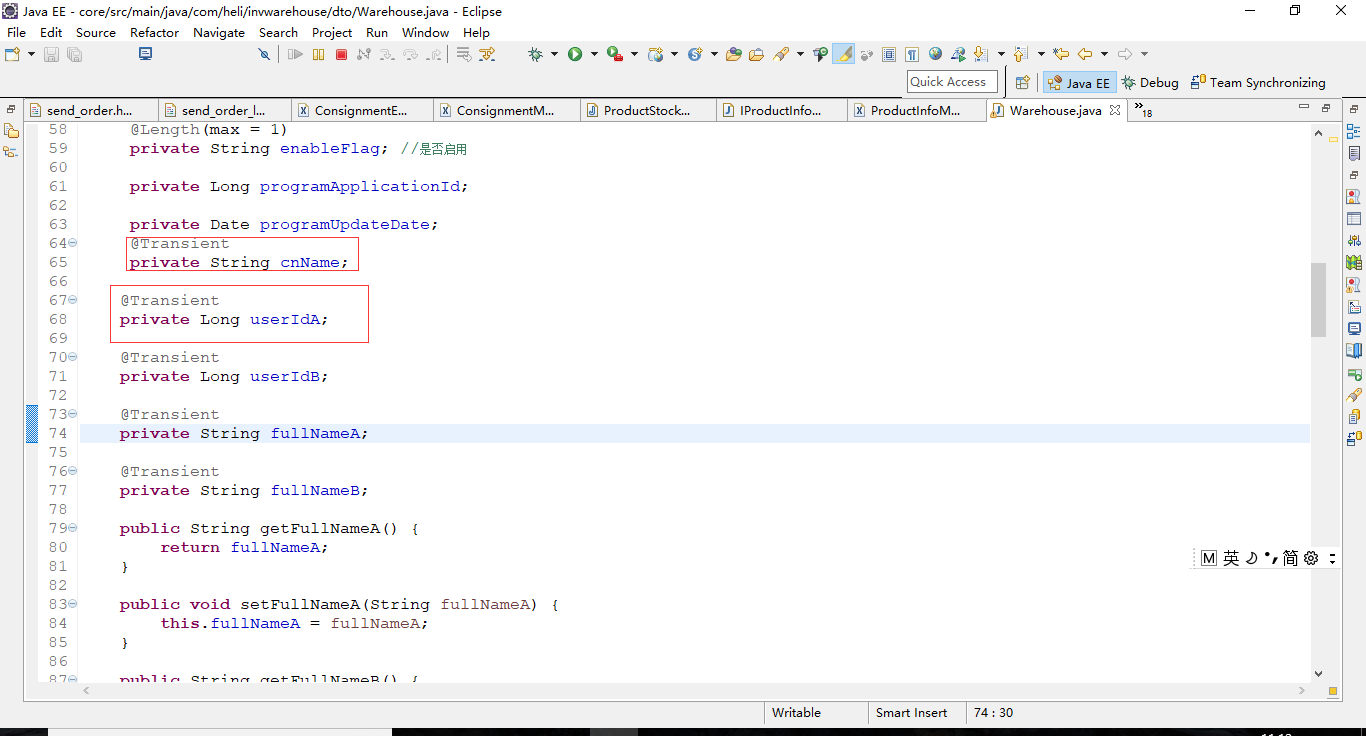
关联映射

### @Length(max = 2147483647)

### @Length(max = 2147483647)

**private** String shippingLabelUrl; //运输标签图片URL

length最大应该是Integer.MAXVALUE，即2147483647



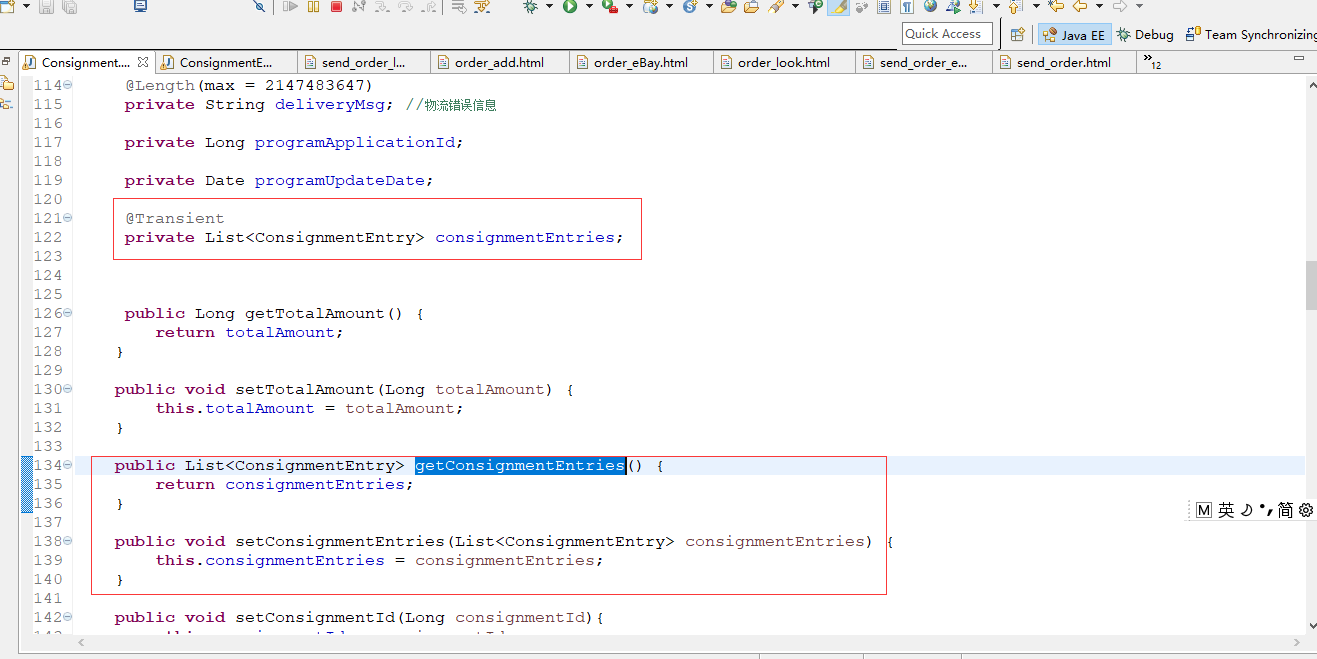
### @Id

@Id

@GeneratedValue

private Long warehouseId; //表ID，主键，供其他表做外键

### dto实体类中的配置一对多



## Mybatis

### \_parameter



### 2.Update

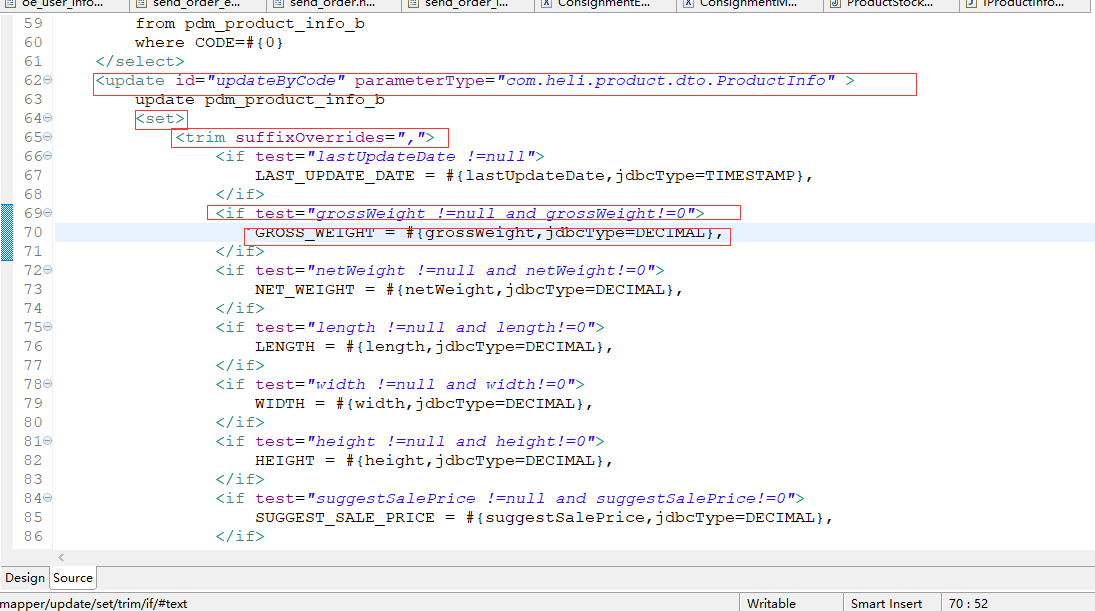
**1.<trim prefix="" suffix="" suffixOverrides="" prefixOverrides=""></trim>**

prefix:在trim标签内sql语句加上前缀。

suffix:在trim标签内sql语句加上后缀。

suffixOverrides:指定去除多余的后缀内容，如：suffixOverrides=","，去除trim标签内sql语句多余的后缀","。

prefixOverrides:指定去除多余的前缀内容



## SpringMVC

### [Http请求中Content-Type讲解以及在Spring MVC中的应用](http://www.cnblogs.com/think-in-java/p/5473772.html)

引言： 在Http请求中，我们每天都在使用Content-type来指定不同格式的请求信息，但是却很少有人去全面了解content-type中允许的值有多少，这里将讲解Content-Type的可用值，以及在[spring](http://lib.csdn.net/base/17) MVC中如何使用它们来映射请求信息。

1.  Content-Type

  MediaType，即是Internet Media Type，互联网媒体类型；也叫做MIME类型，在Http协议消息头中，使用Content-Type来表示具体请求中的媒体类型信息。

#### 博客URL:

http://www.cnblogs.com/think-in-java/p/5473772.html

# 3.前端页面使用

## dataItme["number"] 表示当前td参数值

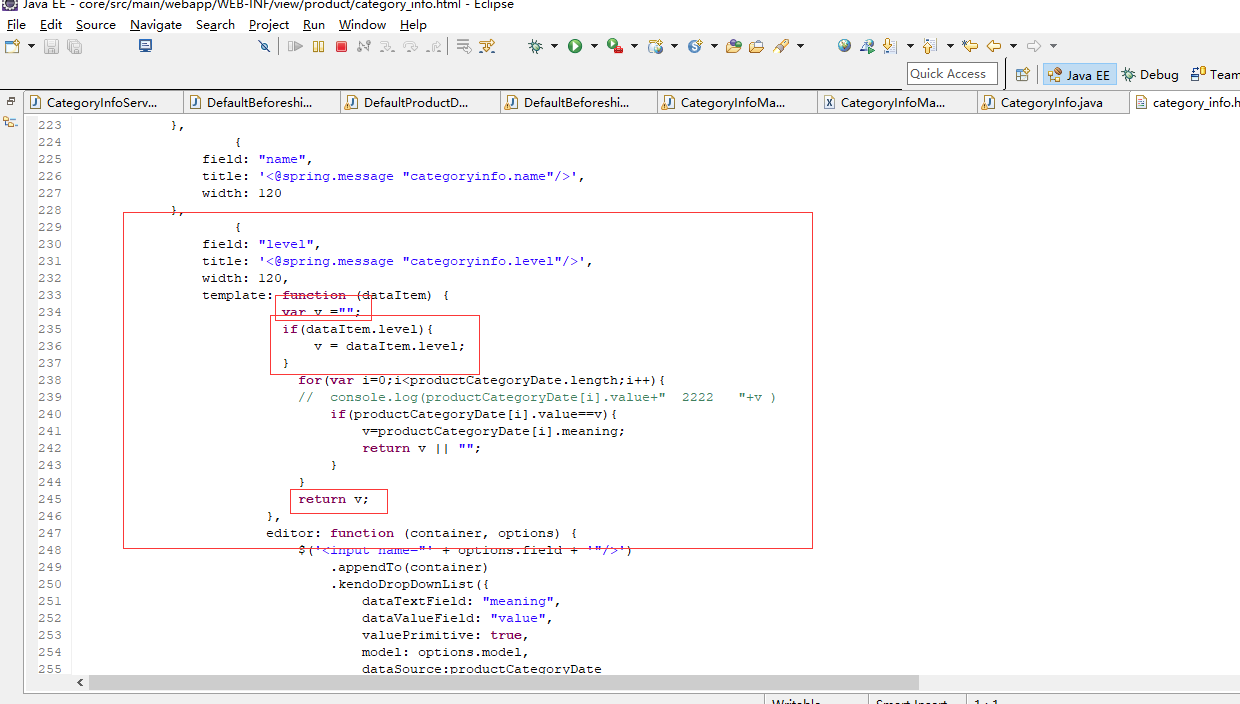
我们需要注意。

template: function (dataItem) {

template模板

editor: function (container, options) {编辑的啊。比如我们拼装我们span.

这里判断为



## 删除不了无法保存的js代码

remove:function(){

var grid = $("#grid").data("kendoGrid");

var checked = grid.selectedDataItems(); //获得选中的数据行

if (checked.length == 0) {

kendo.ui.showInfoDialog({

message: "请至少选择一行"

})

} else {

for (var i = 0; i < checked.length; i++) {

console.log("id"+checked[i].deliveryCompanyId);

if(null==checked[i].deliveryCompanyId || " "==checked[i].deliveryCompanyId){

console.log("id"+checked[i].deliveryCompanyId);

Hap.deleteGridSelection({

grid: $('#grid')

});

}else{

kendo.ui.showInfoDialog({

message: '已保存数据不能删除'

});

}

}

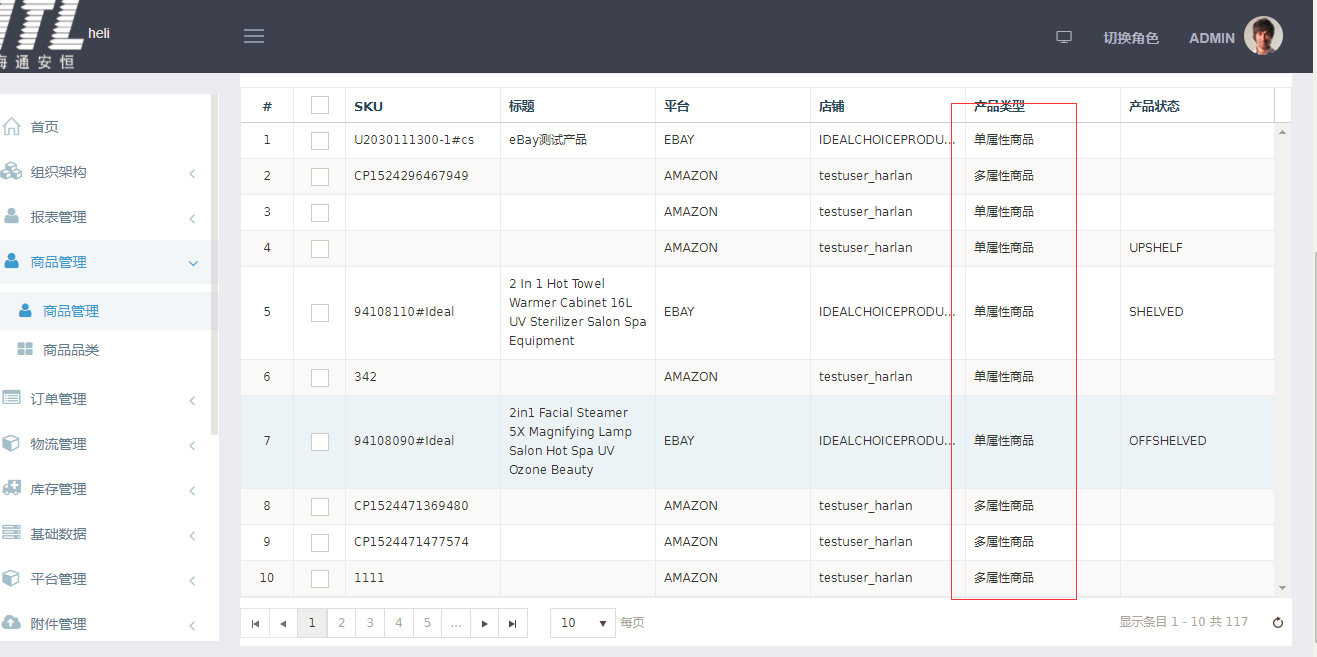
}

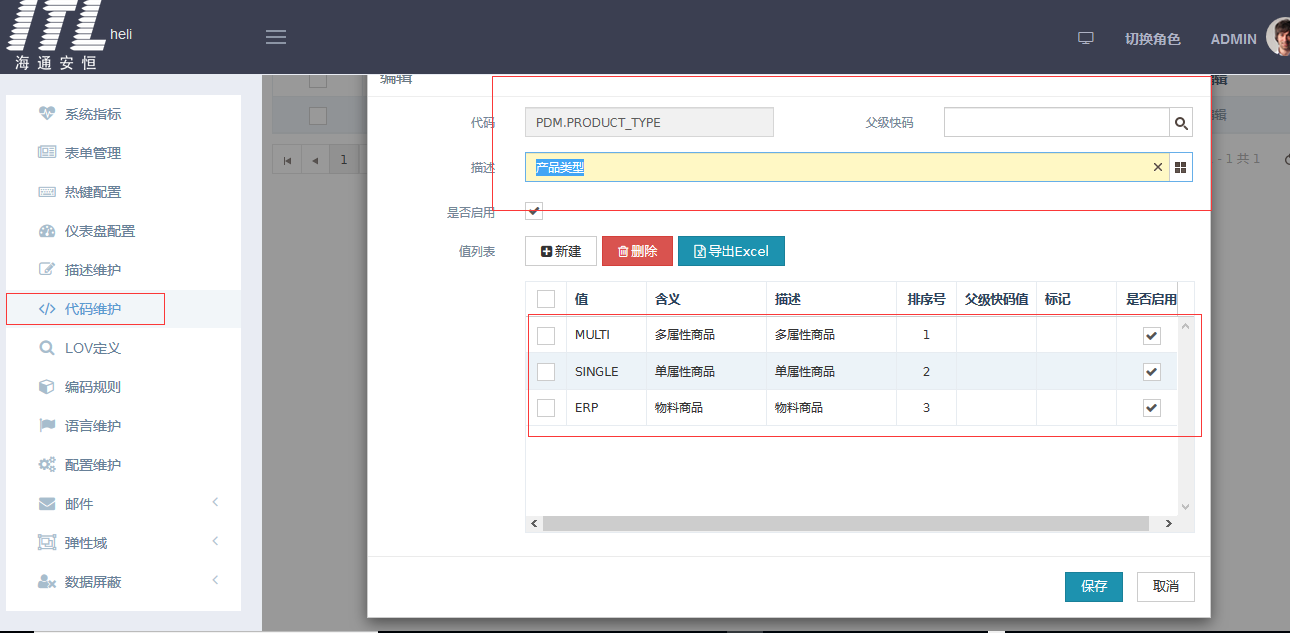
},

## 3，增加页面常量显示，多选，涉及遍历所有值，然后显示

## 这个常量的参数

### 1.图片





### 2.引入

<script src="${base.contextPath}/common/code?typeData=PDM.PRODUCT\_TYPE" type="text/javascript"></script>

### 3，具体代码细节

{

field: "entryType",

title: '产品类型',

width: 120,

template: **function** (dataItem) {

**var** v = dataItem.entryType ? dataItem.entryType : "";

**for**(**var** i=0;i<typeData.length;i++){

**if**(typeData[i].value==v){

v=typeData[i].meaning;

**return** v;

}

}

**return** v;

}

},

# 6.BUG---错误日常记录

## 1.平时操作的注意一些写代码操作

### 小心选择性插入insertSelective

因为他会对一些字段进行insert时，但也会违反约束。

不能为空的字段没被我们操作insert，所以导致报错。

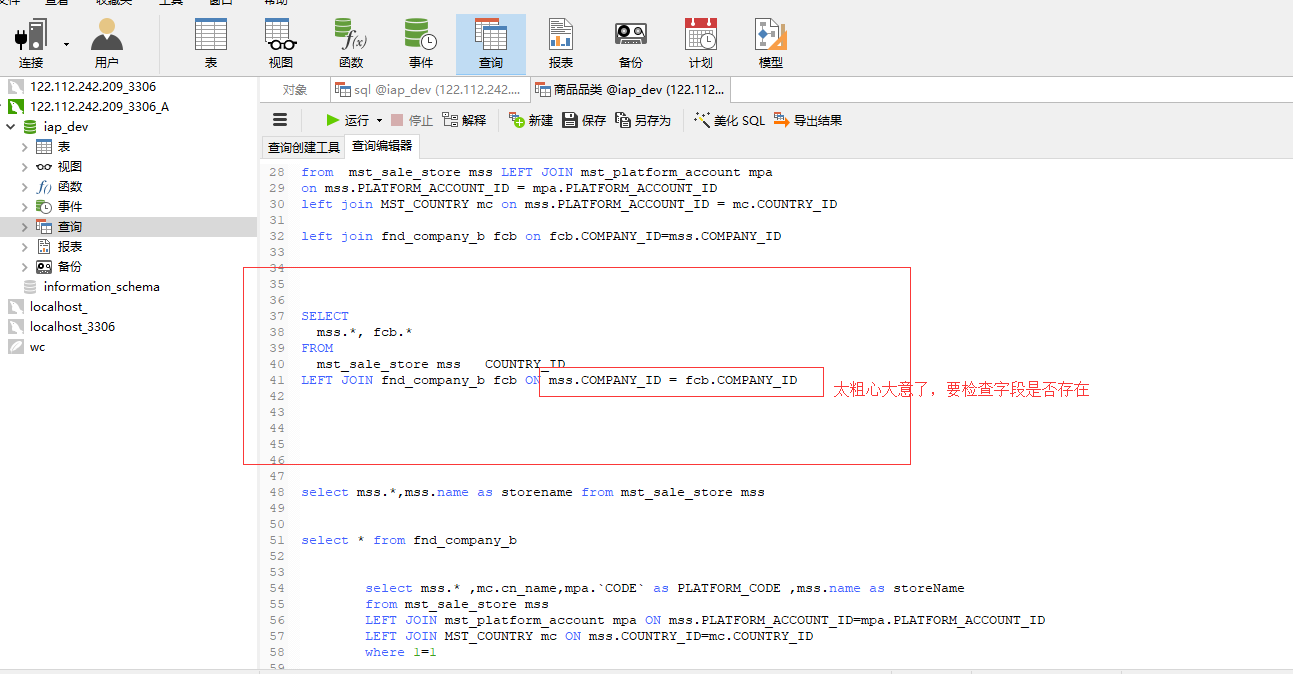
我需要手动设置一下值。

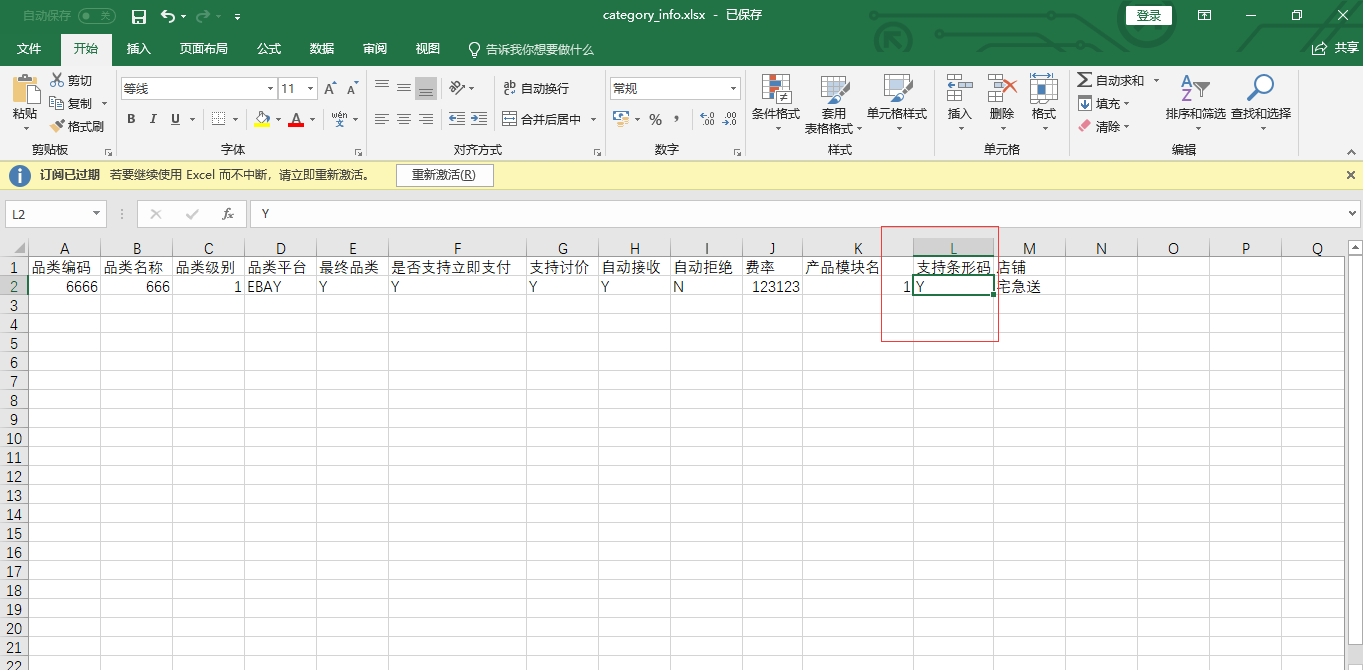
要小心for long……….. 字符超多了。 Boolean值 true或者fales 改成 ‘Y’ OR ‘N’

### console.log(resultXML);//将数据打印

## 2.自己的粗心大意，不该犯的错

### 1.导入问题





Excle文本格式错误了。

要检查清楚这个。下次。

Excle格式要与注解的格式对应。如果缺少就会报异常。

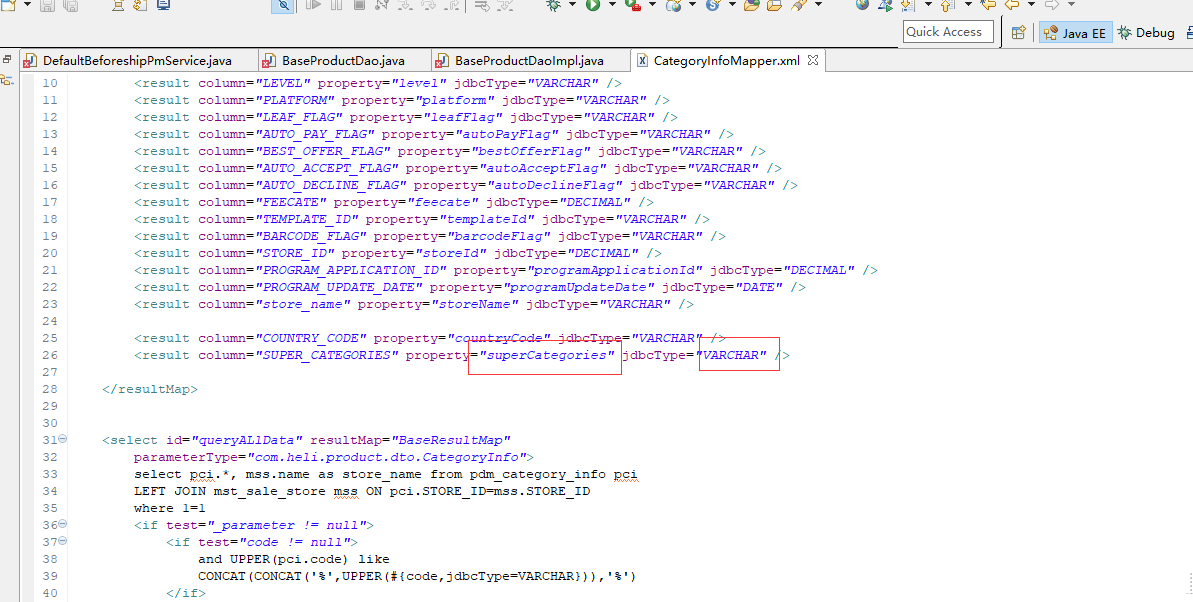
我今天错误就是忘记写L 支持条形码 的row了。

（海龙哥帮找错误-通过断点去查看excle dto注入参数看是否 将13个属性注入进来

结果发现少注入一个属性 导致报错。是我excle格式写错了，将支持条形码写漏了。写漏一个，而dt确有这个注解，就报异常了。 它必须excle对应属性要正确的。）

### 2.mapper漏写属性 或者 没加上 “”

会让后台一直报错，启动不起来。



还有指定的类型也要对上数据库的。

### 3要仔细看是否存在外键啊

粗心 粗心 粗心

有些事情就是自己粗心呢。

要引以为戒。粗心事儿，下次这种事一定要自己先看好。

战哥帮忙找的错误。

对应表没有ID. 自己将国家id看错成公司id。

导致一直没有列。

### 4 LOG.info时会发捕获异常不忘下层走了

我们报错时，如果代码中打了 LOG.INFO的话，就会被捕获到catch中。导致不往下层走了。

需要注意。我们可以将其注释掉。

或者注意断点是否会走，

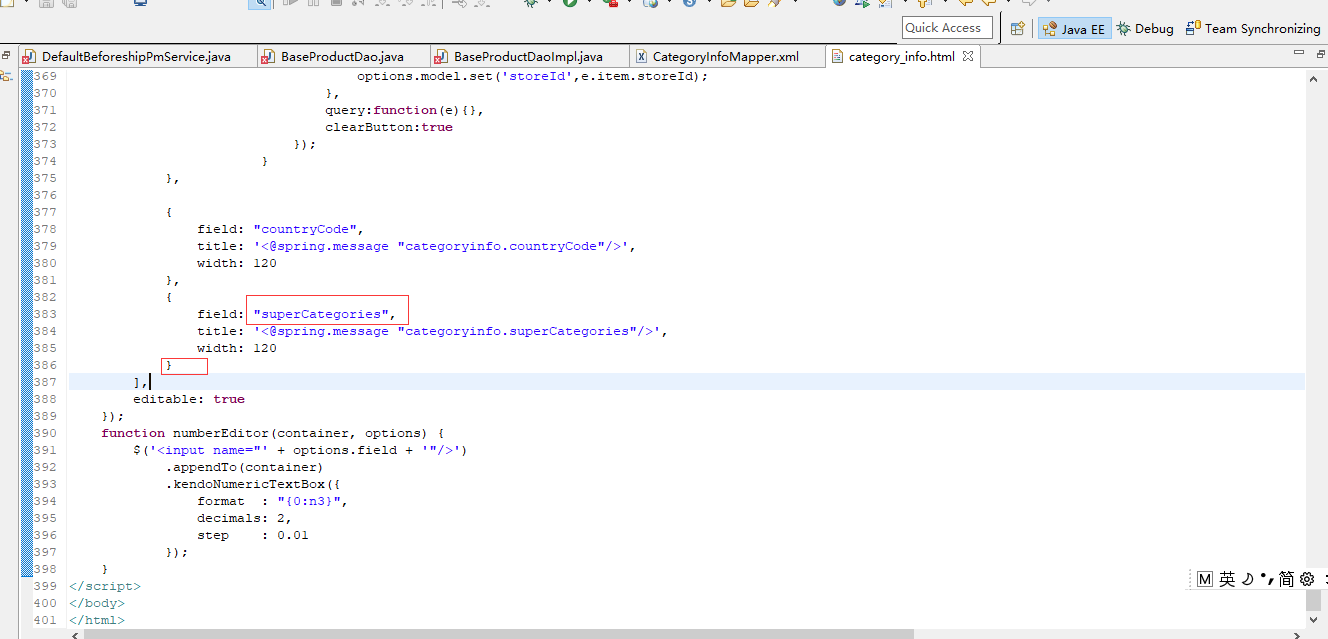
程序报错，碰到LOG.INFO的话，就不往下走了，直接抛出。

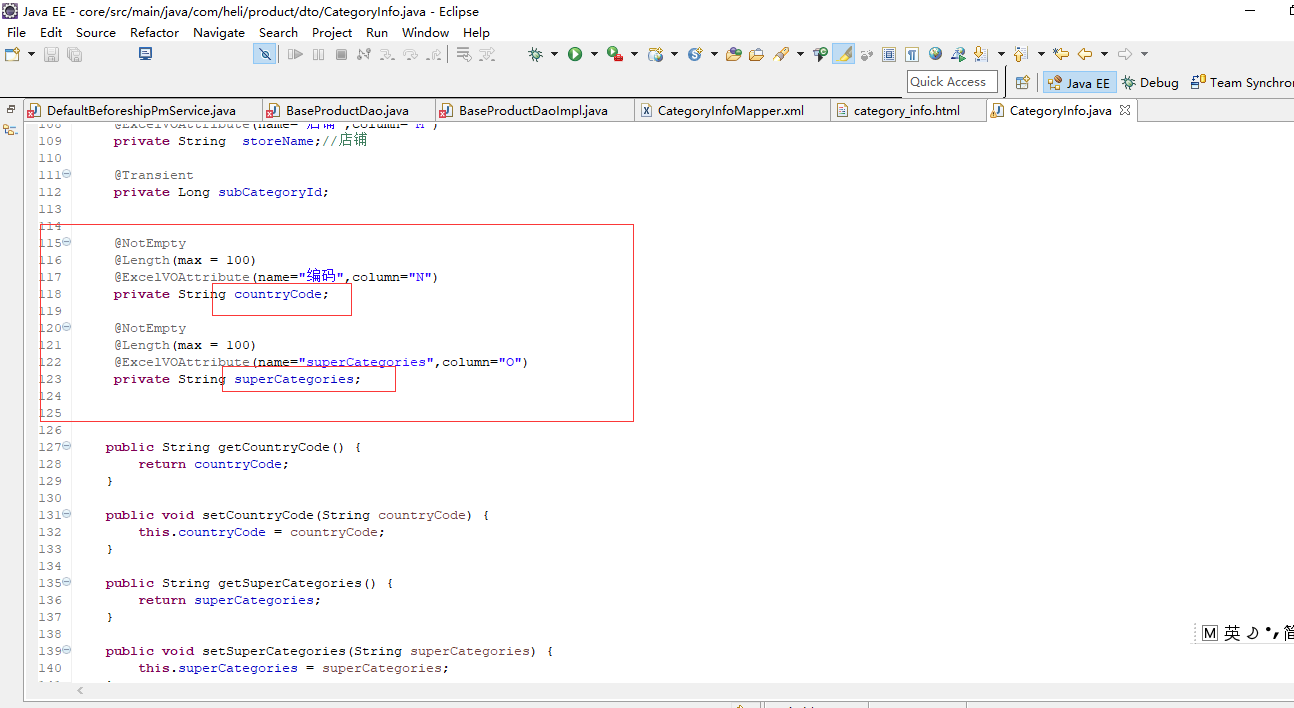
我们下次需要将其注释掉。

（战哥帮忙找的错误 ）

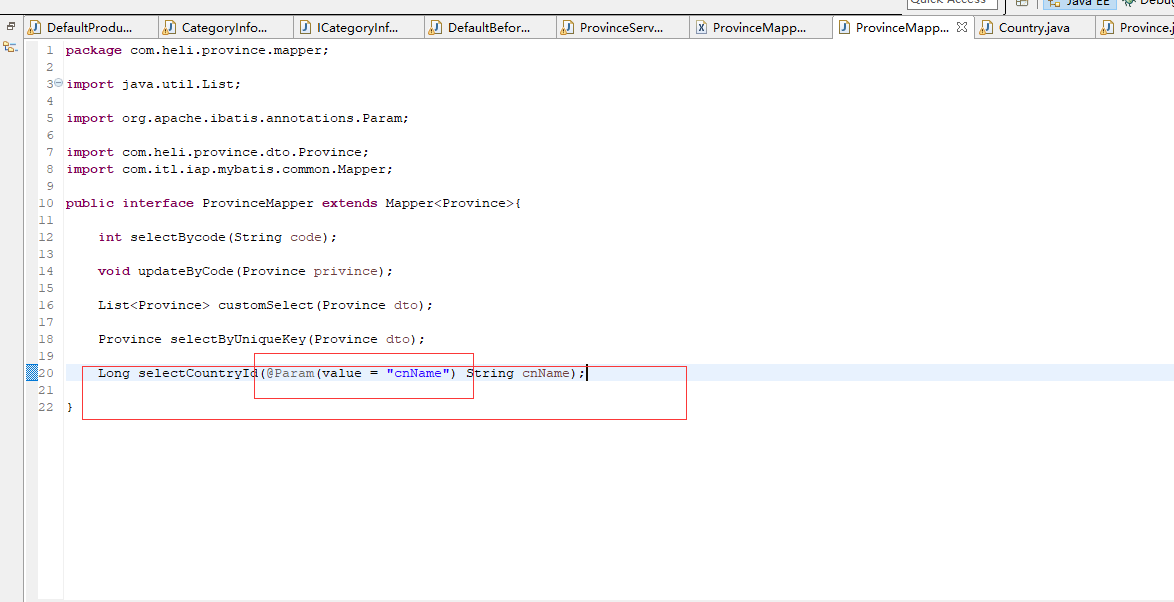
### 5前后台 html的field要和dto的要对应上。

否则会报错字段不存在。





## 3.myBatis配置文件问题



这个注解属性不加会报

要小心 访问接口时 URL 的 / 要注意的

今天访问eBay接口就出现此问题。

## 4，今天注意事项 4/22

1. 提交svn代码犯了错误。

主要是粗心，忘记提交接口mapper接口的参数

还有就是mapper，xml文件中有重复的方法。

这个问题很不应该啊。我们应该去掉重复的方法

还有就是接口名字应该规范。

提交代码一定不能忘记 漏提交的。不然会导致项目启动失败。

今天主要是mapper层不停滴报错呢。

第二 ： 晚上的导出 1.是因为你mapper.xml没有返回id.所以一直导致get不到id

出来。加上即可。下次可不能再犯了。

## 5，在调用接口时，需要注意将 多余 / / 去掉，因为我们开发工具是无法帮我去掉的。

检查好斜杆问题 /

8

# 7.我们项目运用的webservice

我们发送一个请求到一个系统，他会给我们返回来数据。这个就是一个webservice

# 8.定时任务学习

**我们在写**[**Java**](http://lib.csdn.net/base/javase)**程序中经常会写一些定时执行的任务，比如某月的几号去执行一件事，每天的凌晨做些什么，或者每天执行一次什么方法。**

# 9.工作流学习

# 10.前端KendoUI学习

# 11.开发编程基础运用

## [synchronized的使用方法](https://blog.csdn.net/superit401/article/details/51243857)

记下来，很重要。

### [Java](http://lib.csdn.net/base/17)语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

     一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

     二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

     三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

     四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

     五、以上规则对其它对象锁同样适用.

举例说明：    
     一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

package ths;

public class Thread1 implements Runnable {    
     public void run() {    
          synchronized(this) {    
               for (int i = 0; i < 5; i++) {    
                    System.out.println(Thread.currentThread().getName() + " synchronized loop " + i);   
               }    
          }    
     }    
     public static void main(String[] args) {    
          Thread1 t1 = new Thread1();    
          Thread ta = new Thread(t1, "A");    
          Thread tb = new Thread(t1, "B");    
          ta.start();    
          tb.start();    
     }   
}

结果：    
     A synchronized loop 0    
     A synchronized loop 1    
     A synchronized loop 2    
     A synchronized loop 3    
     A synchronized loop 4    
     B synchronized loop 0    
     B synchronized loop 1    
     B synchronized loop 2    
     B synchronized loop 3    
     B synchronized loop 4

     二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

package ths;

public class Thread2 {    
     public void m4t1() {    
          synchronized(this) {    
               int i = 5;    
               while( i-- > 0) {    
                    System.out.println(Thread.currentThread().getName() + " : " + i);   
                    try {    
                         Thread.sleep(500);    
                    } catch (InterruptedException ie) {    
                    }    
               }    
          }    
     }    
     public void m4t2() {    
          int i = 5;    
          while( i-- > 0) {    
               System.out.println(Thread.currentThread().getName() + " : " + i);   
               try {    
                    Thread.sleep(500);    
               } catch (InterruptedException ie) {    
               }    
          }    
     }    
     public static void main(String[] args) {    
          final Thread2 myt2 = new Thread2();    
          Thread t1 = new Thread(  new Runnable() {  public void run() {  myt2.m4t1();  }  }, "t1"  );   
          Thread t2 = new Thread(  new Runnable() {  public void run() { myt2.m4t2();   }  }, "t2"  );   
          t1.start();    
          t2.start();    
     }   
}

结果：    
     t1 : 4    
     t2 : 4    
     t1 : 3    
     t2 : 3    
     t1 : 2    
     t2 : 2    
     t1 : 1    
     t2 : 1    
     t1 : 0    
     t2 : 0

     三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

     //修改Thread2.m4t2()方法：    
     public void m4t2() {    
          synchronized(this) {    
               int i = 5;    
               while( i-- > 0) {    
                    System.out.println(Thread.currentThread().getName() + " : " + i);   
                    try {    
                         Thread.sleep(500);    
                    } catch (InterruptedException ie) {    
                    }    
               }    
          }

     }

结果：

     t1 : 4    
     t1 : 3    
     t1 : 2    
     t1 : 1    
     t1 : 0    
     t2 : 4    
     t2 : 3    
     t2 : 2    
     t2 : 1    
     t2 : 0

     四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

     //修改Thread2.m4t2()方法如下：

     public synchronized void m4t2() {    
          int i = 5;    
          while( i-- > 0) {    
               System.out.println(Thread.currentThread().getName() + " : " + i);   
               try {    
                    Thread.sleep(500);    
               } catch (InterruptedException ie) {    
               }    
          }    
     }

结果：    
     t1 : 4    
     t1 : 3    
     t1 : 2    
     t1 : 1    
     t1 : 0    
     t2 : 4    
     t2 : 3    
     t2 : 2    
     t2 : 1    
     t2 : 0

     五、以上规则对其它对象锁同样适用:

package ths;

public class Thread3 {   
     class Inner {   
          private void m4t1() {   
               int i = 5;   
               while(i-- > 0) {   
                    System.out.println(Thread.currentThread().getName() + " : Inner.m4t1()=" + i);  
                    try {   
                         Thread.sleep(500);   
                    } catch(InterruptedException ie) {   
                    }   
               }   
          }   
          private void m4t2() {   
               int i = 5;   
               while(i-- > 0) {   
                    System.out.println(Thread.currentThread().getName() + " : Inner.m4t2()=" + i);  
                    try {   
                         Thread.sleep(500);   
                    } catch(InterruptedException ie) {   
                    }   
               }   
          }   
     }   
     private void m4t1(Inner inner) {   
          synchronized(inner) { //使用对象锁   
          inner.m4t1();   
     }   
     private void m4t2(Inner inner) {   
          inner.m4t2();   
     }   
     public static void main(String[] args) {   
          final Thread3 myt3 = new Thread3();   
          final Inner inner = myt3.new Inner();   
          Thread t1 = new Thread( new Runnable() {public void run() { myt3.m4t1(inner);} }, "t1");  
     Thread t2 = new Thread( new Runnable() {public void run() { myt3.m4t2(inner);} }, "t2");  
     t1.start();   
     t2.start();   
  }   
}

结果：

尽管线程t1获得了对Inner的对象锁，但由于线程t2访问的是同一个Inner中的非同步部分。所以两个线程互不干扰。

     t1 : Inner.m4t1()=4    
     t2 : Inner.m4t2()=4    
     t1 : Inner.m4t1()=3    
     t2 : Inner.m4t2()=3    
     t1 : Inner.m4t1()=2    
     t2 : Inner.m4t2()=2    
     t1 : Inner.m4t1()=1    
     t2 : Inner.m4t2()=1    
     t1 : Inner.m4t1()=0    
     t2 : Inner.m4t2()=0

现在在Inner.m4t2()前面加上synchronized：

     private synchronized void m4t2() {    
          int i = 5;    
          while(i-- > 0) {    
               System.out.println(Thread.currentThread().getName() + " : Inner.m4t2()=" + i);   
               try {    
                    Thread.sleep(500);    
               } catch(InterruptedException ie) {    
               }    
          }    
     }

结果：

尽管线程t1与t2访问了同一个Inner对象中两个毫不相关的部分,但因为t1先获得了对Inner的对象锁，所以t2对Inner.m4t2()的访问也被阻塞，因为m4t2()是Inner中的一个同步方法。

     t1 : Inner.m4t1()=4    
     t1 : Inner.m4t1()=3    
     t1 : Inner.m4t1()=2    
     t1 : Inner.m4t1()=1    
     t1 : Inner.m4t1()=0    
     t2 : Inner.m4t2()=4    
     t2 : Inner.m4t2()=3    
     t2 : Inner.m4t2()=2    
     t2 : Inner.m4t2()=1    
     t2 : Inner.m4t2()=0

第二篇：

synchronized 关键字，它包括两种用法：synchronized 方法和 synchronized 块。    
1. synchronized 方法：通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。如：    
public synchronized void accessVal(int newVal);    
synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能

执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行

状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有

一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）

。    
在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized ，以控制其对类的静态成

员变量的访问。    
synchronized 方法的缺陷：若将一个大的方法声明为synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为

synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。当然我们可

以通过将访问类成员变量的代码放到专门的方法中，将其声明为 synchronized ，并在主方法中调用来解决这一问题，但是 Java 为我们提供

了更好的解决办法，那就是 synchronized 块。    
2. synchronized 块：通过 synchronized关键字来声明synchronized 块。语法如下：    
synchronized(syncObject) {    
//允许访问控制的代码    
}    
synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机

制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。    
对synchronized(this)的一些理解   
一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线

程必须等待当前线程执行完这个代码块以后才能执行该代码块。    
二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized

(this)同步代码块。    
三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)

同步代码块的访问将被阻塞。    
四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个

object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。    
五、以上规则对其它对象锁同样适用

<http://hi.baidu.com/sunshibing/blog/item/5235b9b731d48ff430add14a.html>  
**java中synchronized用法**

打个比方：一个object就像一个大房子，大门永远打开。房子里有 很多房间（也就是方法）。

这些房间有上锁的（synchronized方法）， 和不上锁之分（普通方法）。房门口放着一把钥匙（key），这把钥匙可以打开所有上锁的房间。

另外我把所有想调用该对象方法的线程比喻成想进入这房子某个 房间的人。所有的东西就这么多了，下面我们看看这些东西之间如何作用的。

在此我们先来明确一下我们的前提条件。该对象至少有一个synchronized方法，否则这个key还有啥意义。当然也就不会有我们的这个主题了。

一个人想进入某间上了锁的房间，他来到房子门口，看见钥匙在那儿（说明暂时还没有其他人要使用上锁的 房间）。于是他走上去拿到了钥匙

，并且按照自己 的计划使用那些房间。注意一点，他每次使用完一次上锁的房间后会马上把钥匙还回去。即使他要连续使用两间上锁的房间，

中间他也要把钥匙还回去，再取回来。

因此，普通情况下钥匙的使用原则是：“随用随借，用完即还。”

这时其他人可以不受限制的使用那些不上锁的房间，一个人用一间可以，两个人用一间也可以，没限制。但是如果当某个人想要进入上锁的房

间，他就要跑到大门口去看看了。有钥匙当然拿了就走，没有的话，就只能等了。

要是很多人在等这把钥匙，等钥匙还回来以后，谁会优先得到钥匙？Not guaranteed。象前面例子里那个想连续使用两个上锁房间的家伙，他

中间还钥匙的时候如果还有其他人在等钥匙，那么没有任何保证这家伙能再次拿到。 （JAVA规范在很多地方都明确说明不保证，象

Thread.sleep()休息后多久会返回运行，相同优先权的线程那个首先被执行，当要访问对象的锁被 释放后处于等待池的多个线程哪个会优先得

到，等等。我想最终的决定权是在JVM，之所以不保证，就是因为JVM在做出上述决定的时候，绝不是简简单单根据 一个条件来做出判断，而是

根据很多条。而由于判断条件太多，如果说出来可能会影响JAVA的推广，也可能是因为知识产权保护的原因吧。SUN给了个不保证 就混过去了

。无可厚非。但我相信这些不确定，并非完全不确定。因为计算机这东西本身就是按指令运行的。即使看起来很随机的现象，其实都是有规律

可寻。学过 计算机的都知道，计算机里随机数的学名是伪随机数，是人运用一定的方法写出来的，看上去随机罢了。另外，或许是因为要想弄

的确定太费事，也没多大意义，所 以不确定就不确定了吧。）

再来看看同步代码块。和同步方法有小小的不同。

1.从尺寸上讲，同步代码块比同步方法小。你可以把同步代码块看成是没上锁房间里的一块用带锁的屏风隔开的空间。

2.同步代码块还可以人为的指定获得某个其它对象的key。就像是指定用哪一把钥匙才能开这个屏风的锁，你可以用本房的钥匙；你也可以指定

用另一个房子的钥匙才能开，这样的话，你要跑到另一栋房子那儿把那个钥匙拿来，并用那个房子的钥匙来打开这个房子的带锁的屏风。

         记住你获得的那另一栋房子的钥匙，并不影响其他人进入那栋房子没有锁的房间。

         为什么要使用同步代码块呢？我想应该是这样的：首先对程序来讲同步的部分很影响运行效率，而一个方法通常是先创建一些局部变

量，再对这些变量做一些 操作，如运算，显示等等；而同步所覆盖的代码越多，对效率的影响就越严重。因此我们通常尽量缩小其影响范围。

如何做？同步代码块。我们只把一个方法中该同 步的地方同步，比如运算。

         另外，同步代码块可以指定钥匙这一特点有个额外的好处，是可以在一定时期内霸占某个对象的key。还记得前面说过普通情况下钥

匙的使用原则吗。现在不是普通情况了。你所取得的那把钥匙不是永远不还，而是在退出同步代码块时才还。

          还用前面那个想连续用两个上锁房间的家伙打比方。怎样才能在用完一间以后，继续使用另一间呢。用同步代码块吧。先创建另外

一个线程，做一个同步代码 块，把那个代码块的锁指向这个房子的钥匙。然后启动那个线程。只要你能在进入那个代码块时抓到这房子的钥匙

，你就可以一直保留到退出那个代码块。也就是说 你甚至可以对本房内所有上锁的房间遍历，甚至再sleep(10\*60\*1000)，而房门口却还有

1000个线程在等这把钥匙呢。很过瘾吧。

          在此对sleep()方法和钥匙的关联性讲一下。一个线程在拿到key后，且没有完成同步的内容时，如果被强制sleep()了，那key还一

直在 它那儿。直到它再次运行，做完所有同步内容，才会归还key。记住，那家伙只是干活干累了，去休息一下，他并没干完他要干的事。为

了避免别人进入那个房间 把里面搞的一团糟，即使在睡觉的时候他也要把那唯一的钥匙戴在身上。

          最后，也许有人会问，为什么要一把钥匙通开，而不是一个钥匙一个门呢？我想这纯粹是因为复杂性问题。一个钥匙一个门当然更

安全，但是会牵扯好多问题。钥匙 的产生，保管，获得，归还等等。其复杂性有可能随同步方法的增加呈几何级数增加，严重影响效率。这也

算是一个权衡的问题吧。为了增加一点点安全性，导致效 率大大降低，是多么不可取啊。

synchronized的一个简单例子

public class TextThread {

public static void main(String[] args) {   
   TxtThread tt = new TxtThread();   
   new Thread(tt).start();   
   new Thread(tt).start();   
   new Thread(tt).start();   
   new Thread(tt).start();   
}   
}

class TxtThread implements Runnable {   
int num = 100;   
String str = new String();

public void run() {   
   synchronized (str) {   
    while (num > 0) {

     try {   
      Thread.sleep(1);   
     } catch (Exception e) {   
      e.getMessage();   
     }   
     System.out.println(Thread.currentThread().getName()   
       + "this is " + num--);   
    }   
   }   
}   
}

上面的例子中为了制造一个时间差,也就是出错的机会,使用了Thread.sleep(10)

Java对多线程的支持与同步机制深受大家的喜爱，似乎看起来使用了synchronized关键字就可以轻松地解决多线程共享数据同步问题。到底如

何？――还得对synchronized关键字的作用进行深入了解才可定论。

总的说来，synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，也就是平时说的同步方法和同步语句块。如果再细的分类，

synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。

在进一步阐述之前，我们需要明确几点：

A．无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁――而且同步方法很可能还会被其

他线程的对象访问。

B．每个对象只有一个锁（lock）与之相关联。

C．实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

接着来讨论synchronized用到不同地方对代码产生的影响：

假设P1、P2是同一个类的不同对象，这个类中定义了以下几种情况的同步块或同步方法，P1、P2就都可以调用它们。

1． 把synchronized当作函数修饰符时，示例代码如下：

Public synchronized void methodAAA()

{

//….

}

这也就是同步方法，那这时synchronized锁定的是哪个对象呢？它锁定的是调用这个同步方法对象。也就是说，当一个对象P1在不同的线程中

执行这个同步方法时，它们之间会形成互斥，达到同步的效果。但是这个对象所属的Class所产生的另一对象P2却可以任意调用这个被加了

synchronized关键字的方法。

上边的示例代码等同于如下代码：

public void methodAAA()

{

synchronized (this)      // (1)

{

       //…..

}

}

(1)处的this指的是什么呢？它指的就是调用这个方法的对象，如P1。可见同步方法实质是将synchronized作用于object reference。――那个

拿到了P1对象锁的线程，才可以调用P1的同步方法，而对P2而言，P1这个锁与它毫不相干，程序也可能在这种情形下摆脱同步机制的控制，造

成数据混乱：（

2．同步块，示例代码如下：

public void method3(SomeObject so)

{

    synchronized(so)

    {   
       //…..   
    }

}

这时，锁就是so这个对象，谁拿到这个锁谁就可以运行它所控制的那段代码。当有一个明确的对象作为锁时，就可以这样写程序，但当没有明

确的对象作为锁，只是想让一段代码同步时，可以创建一个特殊的instance变量（它得是一个对象）来充当锁：

class Foo implements Runnable

{

        private byte[] lock = new byte[0]; // 特殊的instance变量

        Public void methodA()   
        {

           synchronized(lock) { //… }

        }

        //…..

}

注：零长度的byte数组对象创建起来将比任何对象都经济――查看编译后的字节码：生成零长度的byte[]对象只需3条操作码，而Object lock

= new Object()则需要7行操作码。

3．将synchronized作用于static 函数，示例代码如下：

Class Foo   
{

    public synchronized static void methodAAA()   // 同步的static 函数   
    {   
        //….   
    }

    public void methodBBB()   
    {

       synchronized(Foo.class)   // class literal(类名称字面常量)

    }   
}

   代码中的methodBBB()方法是把class literal作为锁的情况，它和同步的static函数产生的效果是一样的，取得的锁很特别，是当前调用这

个方法的对象所属的类（Class，而不再是由这个Class产生的某个具体对象了）。

记得在《Effective Java》一书中看到过将 Foo.class和 P1.getClass()用于作同步锁还不一样，不能用P1.getClass()来达到锁这个Class的

目的。P1指的是由Foo类产生的对象。

可以推断：如果一个类中定义了一个synchronized的static函数A，也定义了一个synchronized 的instance函数B，那么这个类的同一对象Obj

在多线程中分别访问A和B两个方法时，不会构成同步，因为它们的锁都不一样。B方法的锁是Obj这个对象，而A的锁是Obj所属的那个Class。

小结如下：

搞清楚synchronized锁定的是哪个对象，就能帮助我们设计更安全的多线程程序。

还有一些技巧可以让我们对共享资源的同步访问更加安全：

1． 定义private 的instance变量+它的 get方法，而不要定义public/protected的instance变量。如果将变量定义为public，对象在外界可以

绕过同步方法的控制而直接取得它，并改动它。这也是JavaBean的标准实现方式之一。

2． 如果instance变量是一个对象，如数组或ArrayList什么的，那上述方法仍然不安全，因为当外界对象通过get方法拿到这个instance对象

的引用后，又将其指向另一个对象，那么这个private变量也就变了，岂不是很危险。 这个时候就需要将get方法也加上synchronized同步，并

且，只返回这个private对象的clone()――这样，调用端得到的就是对象副本的引用了

## [java与js中的字符串正则表达式应用总结](https://blog.csdn.net/superit401/article/details/79713730)

### 一、java中可使用正则地方一般在String类、Pattern 类、Matcher类，String对象可使用正则的方法matches()、replace()、replaceAll()、replaceFirst()、split()等。

例：

String s="4ruser1beuser2be";

System.out.println(s.matches("(\\w\*)user(\\w\*)"));//true

System.out.println(s.matches("(\\S\*)user(\\S\*)"));//true

System.out.println(s.matches("[A-Za-z0-9]\*user[A-Za-z0-9]\*"));//true

System.out.println(s.matches("([A-Za-z0-9]\*)user[A-Za-z0-9]\*"));//true

System.out.println(s.matches("user"));//false

System.out.println(s.replace("user", "TT"));//4rTT1beTT2be

System.out.println(s.replaceAll("user", "TT"));//4rTT1beTT2be

System.out.println(s.replace("user\\d", "TT"));//4ruser1beuser2be

System.out.println(s.replaceAll("user\\d", "TT"));//4rTTbeTTbe

System.out.println(s.replace("/user\\d/g", "TT"));//4ruser1beuser2be

System.out.println(s.replaceAll("user\\d/g", "TT"));//4ruser1beuser2be

参考：http://www.runoob.com/java/java-regular-expressions.html

二、javascript中RegExp表达式对象的exec()和test()，String对象的match()、replace()、search()、split()。

----------------------------------------RegExp.exec()-----------------------------------

对于表达式对象的exec方法，不加g，只匹配第一个，无论执行多少次匹配都是如此；

加g匹配第一次执行也返回第一个匹配，再执行返回第二个匹配，依次类推。

var regx=new RegExp('user');

var str='user18beeuser2bee';

var rs=regx.exec(str);

var rs2=regx.exec(str);

console.log("rs:"+rs);//rs:user

console.log("rs2:"+rs2);//rs2:user

var regx=/user\d/;

var str='user18beeuser2bee';

var rs=regx.exec(str);

var rs2=regx.exec(str);

console.log("rs:"+rs);//rs:user1

console.log("rs2:"+rs2);//rs2:user1

regx=/user\d/g;

var str='user18beeuser2bee';

var rs=regx.exec(str);

var rs2=regx.exec(str);

console.log("rs:"+rs);//rs:user1

console.log("rs2:"+rs2);//rs2:user2

-----------------------------------------RegExp.test()-----------------------------------

表达式对象的test方法，加入g于不加上g没有什么区别。

var regx=/user\d/;

var str='user18beeuser2bee';

var rs=regx.test(str);

console.log("rs:"+rs);//rs:true

var regx=/user\d/g;

var str='user18beeuser2bee';

var rs=regx.test(str);

console.log("rs:"+rs);//rs:true

-----------------------------------------String.match()-----------------------------------

对于String对象的match方法，不加入g，也只是返回第一个匹配，一直执行match方法也总是返回第一个匹配，加入g，则一次返回所有的匹配（注意这与表达式对象的exec方法不同，对于exec而言，表达式即使加上了g，也不会一次返回所有的匹配）。

var regx=/user\d/;

var str='user1beeuser2bee';

var rs=str.match(regx);

console.log("rs:"+rs);//rs:user1

var regx=/user\d/g;

var str='user1beeuser2bee';

var rs=str.match(regx);

console.log("rs:"+rs);//rs:user1,user2

var regx='user';

var str='user1beeuser2bee';

var rs=str.match(regx);

console.log("rs:"+rs);//rs:user

-----------------------------------------String.replace()-----------------------------------

对于String对象的replace方法，表达式不加入g，则只替换第一个匹配，如果加入g，则替换所有匹配。

var regx='user';

var str='user1beeuser2bee';

var rs=str.replace(regx,'X');

console.log("rs:"+rs);//rs:X1beeuser2bee

var regx=/user\d/;

var str='user1beeuser2bee';

var rs=str.replace(regx,'X');

console.log("rs:"+rs);//rs:Xbeeuser2bee

var regx=/user\d/g;

var str='user1beeuser2bee';

var rs=str.replace(regx,'X');

console.log("rs:"+rs);//rs:XbeeXbee

var regx=/usEr\d/i;

var str='user1beeuser2bee';

var rs=str.replace(regx,'X');

console.log("rs:"+rs);//rs:Xbeeuser2bee

-----------------------------------------String.search()-----------------------------------

对于String对象的search方法，加不加g也是一样的。

var regx=/user\d/;

var str='4ruser1beeuser2bee';

var rs=str.search(regx);

console.log("rs:"+rs);//rs:2

var regx=/user/;

var str='4ruser1beeuser2bee';

var rs=str.search(regx);

console.log("rs:"+rs);//rs:2

var regx=/user/g;

var str='4ruser1beeuser2bee';

var rs=str.search(regx);

console.log("rs:"+rs);//rs:2

var regx=/user\d/g;

var str='4ruser1beeuser2bee';

var rs=str.search(regx);

console.log("rs:"+rs);//rs:2

var regx=/user/i;

var str='4ruser1beeuser2bee';

var rs=str.search(regx);

console.log("rs:"+rs);//rs:2

-----------------------------------------String.split()-----------------------------------

对于String对象的split方法，加上g与不加g是一样的。

var regx=/user\d/;

var str='4ruser1beeuser2bee';

var rs=str.split(regx);

console.log("rs:"+rs);//rs:4r,bee,bee

var regx=/user\d/g;

var str='4ruser1beeuser2bee';

var rs=str.split(regx);

console.log("rs:"+rs);//rs:4r,bee,bee

var regx='user';

var str='4ruser1beeuser2bee';

var rs=str.split(regx);

console.log("rs:"+rs);//rs:4r,1bee,2bee

参考：http://www.runoob.com/js/js-regexp.html

https://blog.csdn.net/huiguixian/article/details/6131048

https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/1700215?fr=aladdin#9\_1

## [接口测试基础](http://www.cnblogs.com/houzhizhe/p/6825457.html)

URL:

一些好的博客网址入口

<http://www.cnblogs.com/houzhizhe/category/1012313.html>

### 什么是接口测试

测试人员通常所说的“接口测试”是针对系统各组件之间接口的一种测试，它属于功能测试。接口能测出普通界面操作难以发现的问题。如，我们都知道系统是由前端后端组成，一些数据在前端做了校验，后端同样也需要校验才能保证安全，界面操作显然只能检查到前端校验这一层，只有直接面对前后端之间的该接口才能检验出后端是否也做了校验。

**接口测试的必要性**

   可以发现很多页面操作发现不了的问题

   检查系统的异常处理能力

   检查系统的安全性、稳定性

   前端随便变，接口测好了，后端不用变

**接口测试的流程**

   需求评审，熟悉业务和需求

   开发提供接口文档

   编写接口测试用例

   用例评审

   提测后开始测试

   提交测试报告

**接口文档**是接口测试的参照，至少包括：

1、接口说明

2、调用url

3、请求方法（get\post ……）

4、请求参数、参数类型、请求参数说明

5、返回参数说明

**接口测试用例设计**

通过性验证：首先保证接口好用，按文档正常传入，查看是否可以返回正确的结果。

参数组合： 按接口文档中对参数的要求进行有目的的组合，比如必填未填是否通过，标志类参数值的切换是否能对应正确的功能等。（这部分很关键）

接口安全：

      1、绕过正常值验证。

      2、绕过身份授权验证。

      3、参数是否加密，加密规则是否容易破解。

      4、密码安全规则，密码的复杂程度校验。

异常验证：不按照接口文档上的要求输入参数，来验证接口对异常情况的反应。

**接口测试用例模板**（可根据项目实际情况设计增减）

1、项目            测试针对哪个项目

2、模块            哪个功能模块

3、用例id

4、接口名称

5、用例标题      测试用途概括

6、请求方式      GET/POST

7、请求url        URL地址

8、请求参数

9、前置条件       执行当前请求依赖的条件，不满足就不能正确执行

10、结果验证     预期结果

11、请求报文     可以不写

12、返回报文 　一定要写，这里应该是你请求返回的真实结果

13、测试结果    通过/失败

14、测试人员

**测试http接口**

请求常见有Get请求和Post请求。Get请求通常用来接收数据，Post请求通常用来发送数据；测Get请求可用浏览器完成，参数都可以写在URL里面，测Post请求需要借助工具如Postman，因为客户端需要提供给服务器的信息较多，你要写body传输大量数据。

接口调用有两种传参方式：key-value形式，Json串传参形式。

key-value形式可以把参数拼接在url的后面由?相连，多个参数之间用&相连，如url?parameter1=key1&parameter2=key2…

Json串传参不能把参数直接连在url中，需要写在请求的body里面，可借助工具Postman，打开请求的body写入Json格式参数（由花括号括起来的‘键：值’对）如

{

 “count”: 1,

 “start”: 0,

 “total”: 1

}

请求发出后，http会返回一个状态码表示请求是否成功，状态码有三位，其中开头一位确定了状态类型：

   2xx： 表示请求发送成功，常见200。

   3xx： 代表重定向，要完成请求必须进行更进一步的操作，或把请求重定向到别的地方了，最常见的是302。

   4xx： 客户端错误，请求有语法错误或请求无法实现。400代表客户端发送的请求有语法错误，不能被服务器所理解；401代表访问的页面没有授权；403服务器收到请求，但是拒绝提供服务，比如没有权限访问这个页面；404请求的资源不存在，比如输入错的URL没有这个页面。

   5xx： 代表服务器有异常，500代表服务器内部异常；503服务器当前不能处理客户端的请求，一段时间后可能恢复正常；504代表服务器端超时，没返回结果。

**测试WebSevice接口**

不需要像测http接口那样拼报文，直接把wsdl地址或wsdl文件（这两个都由开发人员提供）填写或导入到工具SoapUI里面，工具里可显示所有相关接口或报文，直接填入参数发送请求参照接口文档查看结果即可。

**Cookie 和 Session**

Cookie是存在于本地的一个键值对，Session是存在于服务器端的一个键值对，通常保存在数据库或缓存里。Cookie和Session在第一次发送某个请求时成对生成，两端都会记录下生成的时间，超出既定的时限后便会自动删除。当请求在时限内再次发出后，Cookie和Session两者会相互比对，匹配上了便执行某些操作，匹配不上则不允许执行某些操作，以此实现快速处理，它们并不是孤立作用的。

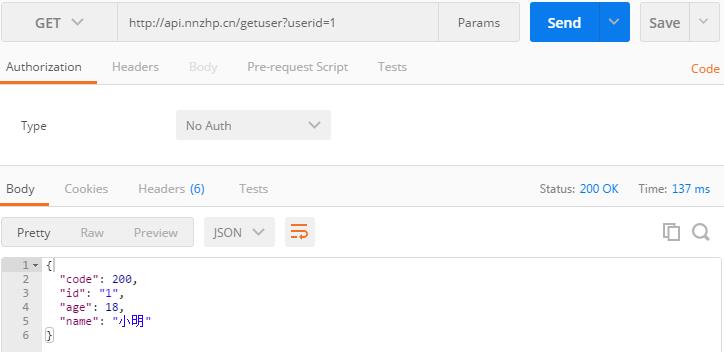
### **Postman做各种类型的http接口测试**

首先，做接口测试前要有明确的接口文档(e.g. <http://test.nnzhp.cn/wiki/index.php?doc-view-59>) ，假设已经在PC上安装好了Postman。

1. 普通的以key-value传参的get请求

e.g. 获取用户信息

Get请求，写入url拼好参数，发送请求，查看结果。



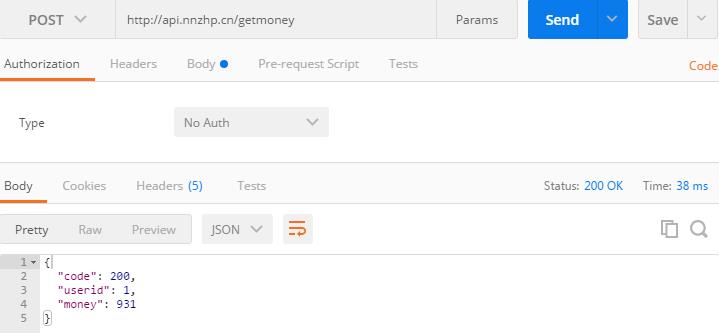
2. 以Json串传参的post请求

e.g. 获取用户余额

Post请求，写入url，点击Body->raw， 写入Json串参数，发送请求



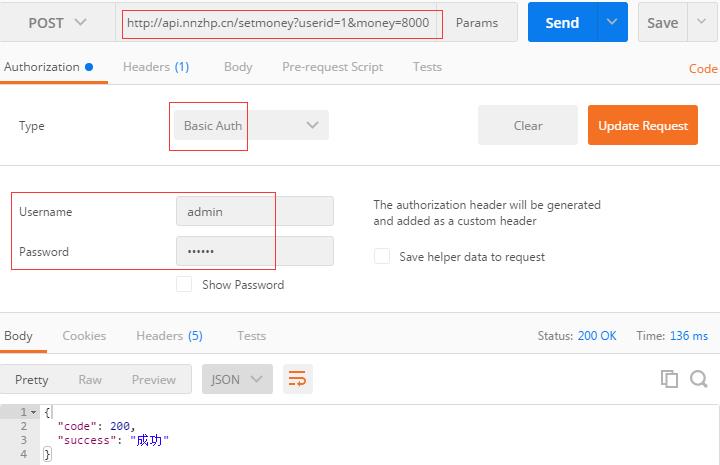
返回Authorization查看结果



3.  请求时需要http权限验证

e.g. 修改用户余额

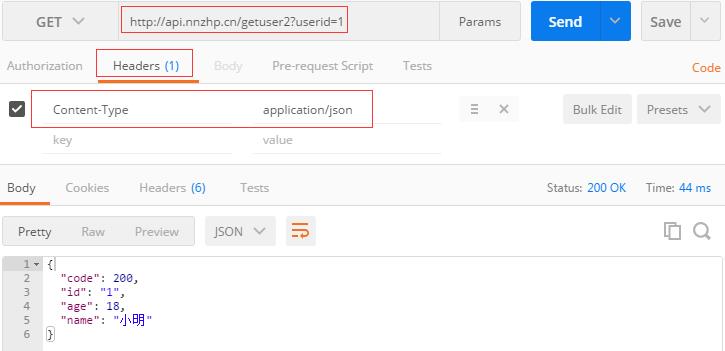
Post请求，写入url和参数，Type选Basic Auth ，填写用户名和密码（如有需要接口文档会给出），发送请求



4.  请求时需要添加请求标头

e.g. 获取用户信息2

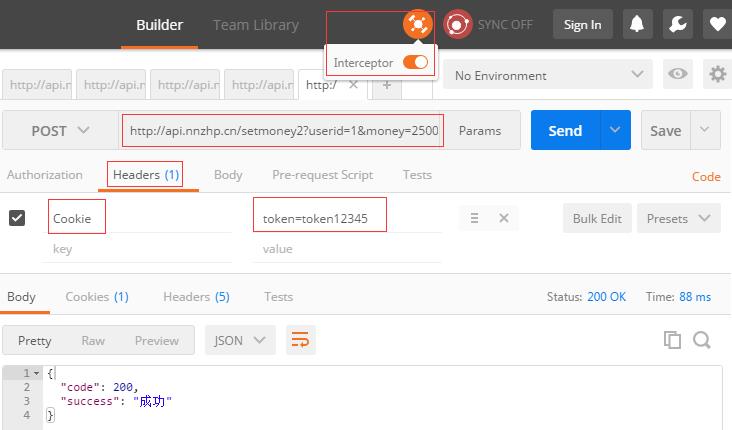
Get请求，写入url和参数，点击Headers并添加header(如有需要接口文档中会给出)，发送请求，查看结果



5.  请求时需要添加Cookie

e.g. 修改用户余额2

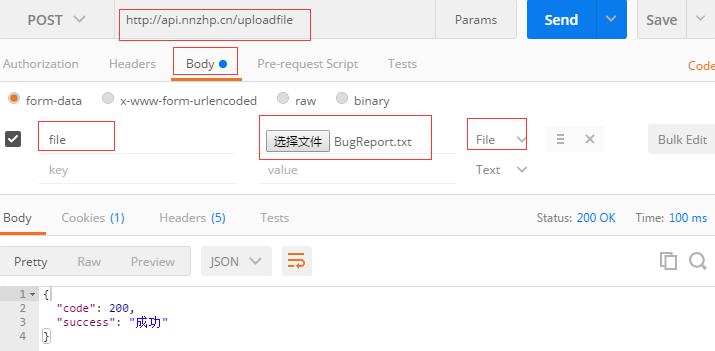
Post请求，写入url和参数，打开插件Interceptor（如果提示安装需要在外网环境下安装，安装一次一劳永逸），在Headers里写入Cookie，值按接口文档所给的填写好，发送请求，查看结果。



6. 请求通过某个接口上传文件

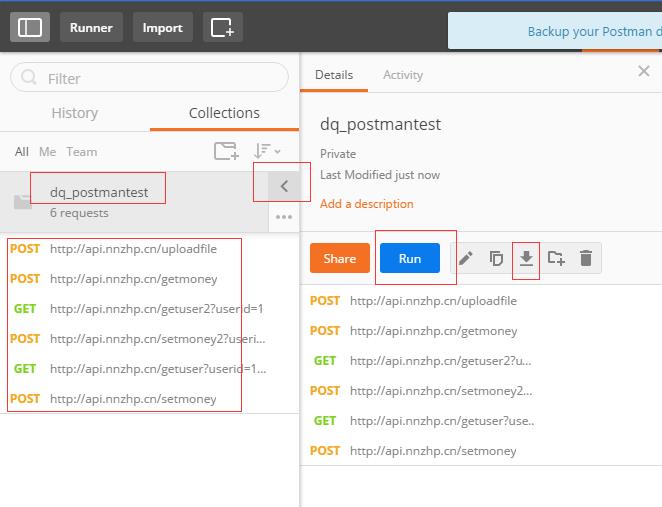
    e.g.上传文件

    Post请求，写入url，在Body中修改text为File，填入参数名，选择本地文件，发送请求，查看结果



**Postman管理请求**

我们可以在Postman 创建Collection（可以按照项目，按照功能模块等看实际情况定）然后把多个请求保存在Collection里，这样就可以实现简单的管理。另外我们可以对Collection做一些简单的操作，比如把Collection导出到本地备份，批量运行Collection中的请求，等等，批量运行的速度好像很慢啊，这可能就是没人用Postman做批量接口测试的原因吧，可以大胆尝试看看，还会有更多的收获。



## 集合运用

### [HashMap的实现原理](http://www.cnblogs.com/think-in-java/p/5169660.html)

**数组**的特点是：寻址容易，插入和删除困难；而**链表**的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的**哈希表**，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法—— 拉链法，我们可以理解为“**链表的数组**” ，

#### 博客URL:

<http://www.cnblogs.com/think-in-java/p/5169660.html>

### Java集合源码剖析——ArrayList源码剖析

#### ****ArrayList简介****

ArrayList是基于数组实现的，是一个动态数组，其容量能自动增长，类似于C语言中的动态申请内存，动态增长内存。

ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能被克隆。

#### ****ArrayList源码剖析****

ArrayList的源码如下（加入了比较详细的注释）：

package java.util;      
    
public class ArrayList<E> extends AbstractList<E>      
       implements List<E>, RandomAccess, Cloneable, java.io.Serializable      
{      
   *// 序列版本号*  
   private static final long serialVersionUID = 8683452581122892189L;      
    
   *// ArrayList基于该数组实现，用该数组保存数据*  
   private transient Object[] elementData;      
    
   *// ArrayList中实际数据的数量*  
   private int size;      
    
   *// ArrayList带容量大小的构造函数。*  
   public ArrayList(int initialCapacity) {      
       super();      
       if (initialCapacity < 0)      
           throw new IllegalArgumentException("Illegal Capacity: "+      
                                              initialCapacity);      
       *// 新建一个数组*  
       this.elementData = new Object[initialCapacity];      
   }      
    
   *// ArrayList无参构造函数。默认容量是10。*  
   public ArrayList() {      
       this(10);      
   }      
    
   *// 创建一个包含collection的ArrayList*  
   public ArrayList(Collection<? extends E> c) {      
       elementData = c.toArray();      
       size = elementData.length;      
       if (elementData.getClass() != Object[].class)      
           elementData = Arrays.copyOf(elementData, size, Object[].class);      
   }      
    
    
   *// 将当前容量值设为实际元素个数*  
   public void trimToSize() {      
       modCount++;      
       int oldCapacity = elementData.length;      
       if (size < oldCapacity) {      
           elementData = Arrays.copyOf(elementData, size);      
       }      
   }      
    
    
   *// 确定ArrarList的容量。*  
   *// 若ArrayList的容量不足以容纳当前的全部元素，设置 新的容量=“(原始容量x3)/2 + 1”*  
   public void ensureCapacity(int minCapacity) {      
       *// 将“修改统计数”+1，该变量主要是用来实现fail-fast机制的*  
       modCount++;      
       int oldCapacity = elementData.length;      
       *// 若当前容量不足以容纳当前的元素个数，设置 新的容量=“(原始容量x3)/2 + 1”*  
       if (minCapacity > oldCapacity) {      
           Object oldData[] = elementData;      
           int newCapacity = (oldCapacity \* 3)/2 + 1;      
           *//如果还不够，则直接将minCapacity设置为当前容量*  
           if (newCapacity < minCapacity)      
               newCapacity = minCapacity;      
           elementData = Arrays.copyOf(elementData, newCapacity);      
       }      
   }      
    
   *// 添加元素e*  
   public boolean add(E e) {      
       *// 确定ArrayList的容量大小*  
       ensureCapacity(size + 1);  *// Increments modCount!!*  
       *// 添加e到ArrayList中*  
       elementData[size++] = e;      
       return true;      
   }      
    
   *// 返回ArrayList的实际大小*  
   public int size() {      
       return size;      
   }      
    
   *// ArrayList是否包含Object(o)*  
   public boolean contains(Object o) {      
       return indexOf(o) >= 0;      
   }      
    
   *//返回ArrayList是否为空*  
   public boolean isEmpty() {      
       return size == 0;      
   }      
    
   *// 正向查找，返回元素的索引值*  
   public int indexOf(Object o) {      
       if (o == null) {      
           for (int i = 0; i < size; i++)      
           if (elementData[i]==null)      
               return i;      
           } else {      
               for (int i = 0; i < size; i++)      
               if (o.equals(elementData[i]))      
                   return i;      
           }      
           return -1;      
       }      
    
       *// 反向查找，返回元素的索引值*  
       public int lastIndexOf(Object o) {      
       if (o == null) {      
           for (int i = size-1; i >= 0; i--)      
           if (elementData[i]==null)      
               return i;      
       } else {      
           for (int i = size-1; i >= 0; i--)      
           if (o.equals(elementData[i]))      
               return i;      
       }      
       return -1;      
   }      
    
   *// 反向查找(从数组末尾向开始查找)，返回元素(o)的索引值*  
   public int lastIndexOf(Object o) {      
       if (o == null) {      
           for (int i = size-1; i >= 0; i--)      
           if (elementData[i]==null)      
               return i;      
       } else {      
           for (int i = size-1; i >= 0; i--)      
           if (o.equals(elementData[i]))      
               return i;      
       }      
       return -1;      
   }      
      
    
   *// 返回ArrayList的Object数组*  
   public Object[] toArray() {      
       return Arrays.copyOf(elementData, size);      
   }      
    
   *// 返回ArrayList元素组成的数组*  
   public <T> T[] toArray(T[] a) {      
       *// 若数组a的大小 < ArrayList的元素个数；*  
       *// 则新建一个T[]数组，数组大小是“ArrayList的元素个数”，并将“ArrayList”全部拷贝到新数组中*  
       if (a.length < size)      
           return (T[]) Arrays.copyOf(elementData, size, a.getClass());      
    
       *// 若数组a的大小 >= ArrayList的元素个数；*  
       *// 则将ArrayList的全部元素都拷贝到数组a中。*  
       System.arraycopy(elementData, 0, a, 0, size);      
       if (a.length > size)      
           a[size] = null;      
       return a;      
   }      
    
   *// 获取index位置的元素值*  
   public E get(int index) {      
       RangeCheck(index);      
    
       return (E) elementData[index];      
   }      
    
   *// 设置index位置的值为element*  
   public E set(int index, E element) {      
       RangeCheck(index);      
    
       E oldValue = (E) elementData[index];      
       elementData[index] = element;      
       return oldValue;      
   }      
    
   *// 将e添加到ArrayList中*  
   public boolean add(E e) {      
       ensureCapacity(size + 1);  *// Increments modCount!!*  
       elementData[size++] = e;      
       return true;      
   }      
    
   *// 将e添加到ArrayList的指定位置*  
   public void add(int index, E element) {      
       if (index > size || index < 0)      
           throw new IndexOutOfBoundsException(      
           "Index: "+index+", Size: "+size);      
    
       ensureCapacity(size+1);  *// Increments modCount!!*  
       System.arraycopy(elementData, index, elementData, index + 1,      
            size - index);      
       elementData[index] = element;      
       size++;      
   }      
    
   *// 删除ArrayList指定位置的元素*  
   public E remove(int index) {      
       RangeCheck(index);      
    
       modCount++;      
       E oldValue = (E) elementData[index];      
    
       int numMoved = size - index - 1;      
       if (numMoved > 0)      
           System.arraycopy(elementData, index+1, elementData, index,      
                numMoved);      
       elementData[--size] = null; *// Let gc do its work*  
    
       return oldValue;      
   }      
    
   *// 删除ArrayList的指定元素*  
   public boolean remove(Object o) {      
       if (o == null) {      
               for (int index = 0; index < size; index++)      
           if (elementData[index] == null) {      
               fastRemove(index);      
               return true;      
           }      
       } else {      
           for (int index = 0; index < size; index++)      
           if (o.equals(elementData[index])) {      
               fastRemove(index);      
               return true;      
           }      
       }      
       return false;      
   }      
    
    
   *// 快速删除第index个元素*  
   private void fastRemove(int index) {      
       modCount++;      
       int numMoved = size - index - 1;      
       *// 从"index+1"开始，用后面的元素替换前面的元素。*  
       if (numMoved > 0)      
           System.arraycopy(elementData, index+1, elementData, index,      
                            numMoved);      
       *// 将最后一个元素设为null*  
       elementData[--size] = null; *// Let gc do its work*  
   }      
    
   *// 删除元素*  
   public boolean remove(Object o) {      
       if (o == null) {      
           for (int index = 0; index < size; index++)      
           if (elementData[index] == null) {      
               fastRemove(index);      
           return true;      
           }      
       } else {      
           *// 便利ArrayList，找到“元素o”，则删除，并返回true。*  
           for (int index = 0; index < size; index++)      
           if (o.equals(elementData[index])) {      
               fastRemove(index);      
           return true;      
           }      
       }      
       return false;      
   }      
    
   *// 清空ArrayList，将全部的元素设为null*  
   public void clear() {      
       modCount++;      
    
       for (int i = 0; i < size; i++)      
           elementData[i] = null;      
    
       size = 0;      
   }      
    
   *// 将集合c追加到ArrayList中*  
   public boolean addAll(Collection<? extends E> c) {      
       Object[] a = c.toArray();      
       int numNew = a.length;      
       ensureCapacity(size + numNew);  *// Increments modCount*  
       System.arraycopy(a, 0, elementData, size, numNew);      
       size += numNew;      
       return numNew != 0;      
   }      
    
   *// 从index位置开始，将集合c添加到ArrayList*  
   public boolean addAll(int index, Collection<? extends E> c) {      
       if (index > size || index < 0)      
           throw new IndexOutOfBoundsException(      
           "Index: " + index + ", Size: " + size);      
    
       Object[] a = c.toArray();      
       int numNew = a.length;      
       ensureCapacity(size + numNew);  *// Increments modCount*  
    
       int numMoved = size - index;      
       if (numMoved > 0)      
           System.arraycopy(elementData, index, elementData, index + numNew,      
                numMoved);      
    
       System.arraycopy(a, 0, elementData, index, numNew);      
       size += numNew;      
       return numNew != 0;      
   }      
    
   *// 删除fromIndex到toIndex之间的全部元素。*  
   protected void removeRange(int fromIndex, int toIndex) {      
   modCount++;      
   int numMoved = size - toIndex;      
       System.arraycopy(elementData, toIndex, elementData, fromIndex,      
                        numMoved);      
    
   *// Let gc do its work*  
   int newSize = size - (toIndex-fromIndex);      
   while (size != newSize)      
       elementData[--size] = null;      
   }      
    
   private void RangeCheck(int index) {      
   if (index >= size)      
       throw new IndexOutOfBoundsException(      
       "Index: "+index+", Size: "+size);      
   }      
    
    
   *// 克隆函数*  
   public Object clone() {      
       try {      
           ArrayList<E> v = (ArrayList<E>) super.clone();      
           *// 将当前ArrayList的全部元素拷贝到v中*  
           v.elementData = Arrays.copyOf(elementData, size);      
           v.modCount = 0;      
           return v;      
       } catch (CloneNotSupportedException e) {      
           *// this shouldn't happen, since we are Cloneable*  
           throw new InternalError();      
       }      
   }      
    
    
   *// java.io.Serializable的写入函数*  
   *// 将ArrayList的“容量，所有的元素值”都写入到输出流中*  
   private void writeObject(java.io.ObjectOutputStream s)      
       throws java.io.IOException{      
   *// Write out element count, and any hidden stuff*  
   int expectedModCount = modCount;      
   s.defaultWriteObject();      
    
       *// 写入“数组的容量”*  
       s.writeInt(elementData.length);      
    
   *// 写入“数组的每一个元素”*  
   for (int i=0; i<size; i++)      
           s.writeObject(elementData[i]);      
    
   if (modCount != expectedModCount) {      
           throw new ConcurrentModificationException();      
       }      
    
   }      
    
    
   *// java.io.Serializable的读取函数：根据写入方式读出*  
   *// 先将ArrayList的“容量”读出，然后将“所有的元素值”读出*  
   private void readObject(java.io.ObjectInputStream s)      
       throws java.io.IOException, ClassNotFoundException {      
       *// Read in size, and any hidden stuff*  
       s.defaultReadObject();      
    
       *// 从输入流中读取ArrayList的“容量”*  
       int arrayLength = s.readInt();      
       Object[] a = elementData = new Object[arrayLength];      
    
       *// 从输入流中将“所有的元素值”读出*  
       for (int i=0; i<size; i++)      
           a[i] = s.readObject();      
   }      
}

#### ****几点总结****

**关于ArrayList的源码，给出几点比较重要的总结：**

1、注意其三个不同的构造方法。无参构造方法构造的ArrayList的容量默认为10，带有Collection参数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData。

2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用Arrays.copyof()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList。

3、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。我们有必要对这两个方法的实现做下深入的了解。

首先来看Arrays.copyof()方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型版本的源码：

public static <T> T[] copyOf(T[] original, int newLength) {    
   return (T[]) copyOf(original, newLength, original.getClass());    
}

很明显调用了另一个copyof方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源码如下：

public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {    
   T[] copy = ((Object)newType == (Object)Object[].class)    
       ? (T[]) new Object[newLength]    
       : (T[]) Array.newInstance(newType.getComponentType(), newLength);    
   System.arraycopy(original, 0, copy, 0,    
                    Math.min(original.length, newLength));    
   return copy;    
}

这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为newlength的数组，调用System.arraycopy()方法，将原来数组中的元素复制到了新的数组中。

下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK中是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

4、注意ArrayList的两个转化为静态数组的toArray方法。

第一个，Object[] toArray()方法。该方法有可能会抛出java.lang.ClassCastException异常，如果直接用向下转型的方法，将整个ArrayList集合转变为指定类型的Array数组，便会抛出该异常，而如果转化为Array数组时不向下转型，而是将每个元素向下转型，则不会抛出该异常，显然对数组中的元素一个个进行向下转型，效率不高，且不太方便。

第二个，<T> T[] toArray(T[] a)方法。该方法可以直接将ArrayList转换得到的Array进行整体向下转型（转型其实是在该方法的源码中实现的），且从该方法的源码中可以看出，参数a的大小不足时，内部会调用Arrays.copyOf方法，该方法内部创建一个新的数组返回，因此对该方法的常用形式如下：

public static Integer[] vectorToArray2(ArrayList<Integer> v) {      
   Integer[] newText = (Integer[])v.toArray(new Integer[0]);      
   return newText;      
}

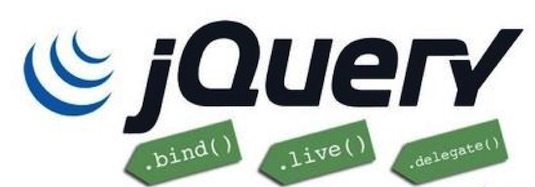
5、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

6、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。

## [十条jQuery代码片段助力Web开发效率提升](https://blog.csdn.net/superit401/article/details/51205066)

### 转载自：http://developer.51cto.com/art/201604/509093.htm

### JQuery是继prototype之后又一个优秀的Javascript库。它是轻量级的js库 ，它兼容CSS3，还兼容各种浏览器（IE 6.0+, FF 1.5+, Safari 2.0+, Opera 9.0+），jQuery2.0及后续版本将不再支持IE6/7/8浏览器。jQuery使用户能更方便地处理HTML（标准通用标记语言下的一个应用）、events、实现动画效果，并且方便地为网站提供AJAX交互。jQuery还有一个比较大的优势是，它的文档说明很全，而且各种应用也说得很详细，同时还有许多成熟的插件可供选择。jQuery能够使用户的html页面保持代码和html内容分离，也就是说，不用再在html里面插入一堆js来调用命令了，只需要定义id即可。以下十项jQuery示例可以帮助大家的Web设计项目顺利实现效率提升。

[](http://s1.51cto.com/wyfs02/M00/7E/F2/wKiom1cNvNqR6-eXAABoU0NzQnM374.jpg-wh_651x-s_547032348.jpg)

**检测IE浏览器**

在进行CSS设计时，IE浏览器对开发者及设计师而言无疑是个麻烦。尽管IE6的黑暗时代已经过去，IE浏览器家族的人气亦在不断下滑，但我们仍然有必要对其进行检测。当然，以下片段亦可用于检测其它浏览器。

1. $(document).ready(function() {
3. **if** (navigator.userAgent.match(/msie/i) ){
5. alert('I am an old fashioned Internet Explorer');
7. }
9. });

来源: [Stack Overflow](http://stackoverflow.com/questions/14365725/easiest-lightest-replacement-for-browser-detection-jquery-1-9)

**平滑滚动至页面顶部**

以下是jQuery最为常见的一种实现效果：点击一条链接以平滑滚动至页面顶部。虽然没什么新鲜感可言，但每位开发者几乎都用得上。

1. $("a[href='#top']").click(function() {
3. $("html, body").animate({ scrollTop: 0 }, "slow");
5. **return** **false**;
7. });

来源: [Stalk Overflow](http://stackoverflow.com/questions/1144805/how-do-i-scroll-to-the-top-of-the-page-with-jquery/1145297#1145297)

**保持始终处于顶部**

以下代码片段允许某一元素始终处于页面顶部。可以想见，其非常适合处理导航菜单、工具栏或者其它重要信息。

1. $(function(){
3. var $win = $(window)
5. var $nav = $('.mytoolbar');
7. var navTop = $('.mytoolbar').length && $('.mytoolbar').offset().top;
9. var isFixed=0;
11. processScroll()
13. $win.on('scroll', processScroll)
15. function processScroll() {
17. var i, scrollTop = $win.scrollTop()
19. **if** (scrollTop >= navTop && !isFixed) {
21. isFixed = 1
23. $nav.addClass('subnav-fixed')
25. } **else** **if** (scrollTop <= navTop && isFixed) {
27. isFixed = 0
29. $nav.removeClass('subnav-fixed')
31. }
33. }

来源: [DesignBump](http://designbump.com/12-handy-jquery-snippets-developers-week-2/)

**替换html标签**

jQuery能够非常轻松地实现html标签替换，而这也将为我们带来更多新的可能。

1. $('li').replaceWith(function(){
3. **return** $("<div />").append($(**this**).contents());
5. });

来源: [Allure Web Solutions](https://allurewebsolutions.com/blog/replace-one-html-tag-with-another-using-jquery)

**检测屏幕宽度**

现在移动设备的人气几乎已经超过了传统计算机，因此对小型屏幕的尺寸进行检测就变得非常重要。幸运的是，我们可以利用jQuery轻松实现这项功能。

1. var responsive\_viewport = $(window).width();
3. /\* if is below 481px \*/
5. **if** (responsive\_viewport < 481) {
7. alert('Viewport is smaller than 481px.');
9. } /\* end smallest screen \*/

来源:[jQuery Rain](http://www.jqueryrain.com/2012/12/jquery-responsive-scripts-that-fire-only-on-specific-window-size/)

**自动修复损坏图片**

如果大家的站点非常庞大而且已经上线数年，那么其中或多或少会出现图片损坏的情况。这项功能可以检测损坏图片并根据我们的选择加以替换。

1. $('img').error(function(){
3. $(**this**).attr('src', 'img/broken.png');
5. });

来源:[WebDesignerDepot](http://www.webdesignerdepot.com/2014/01/10-jquery-snippets-every-designer-should-know/)

**检测复制、粘贴与剪切操作**

利用jQuery，大家可以非常轻松地检测到选定元素的复制、粘贴与剪切操作。

1. $("#textA").bind('copy', function() {
3. $('span').text('copy behaviour detected!')
5. });
7. $("#textA").bind('paste', function() {
9. $('span').text('paste behaviour detected!')
11. });
13. $("#textA").bind('cut', function() {
15. $('span').text('cut behaviour detected!')
17. });

来源: [Snipplr](http://snipplr.com/view/87114/detect-copy-paste-and-cut-behavior-with-jquery/)

**自动为外部链接添加target=“blank”属性**

在链接至外部站点时，大家可能希望使用target="blank"属性以确保在新的选项卡中打开页面。问题在于，target="blank"属性并未经过W3C认证。jQuery能够帮上大忙：以下片段能够检测当前链接是否指向外部，如果是则自动为其添加target="blank"属性。

1. var root = location.protocol + '//' + location.host;
3. $('a').not(':contains(root)').click(function(){
5. **this**.target = "\_blank";
7. });

来源: [jQuery Rain](http://www.jqueryrain.com/2012/09/check-if-link-contains-external-url/)

**悬停时淡入/淡出**

又是另一项“经典”效果，大家可以利用以下片段随时加以运用。

1. $(document).ready(function(){
3. $(".thumbs img").fadeTo("slow", 0.6); // This sets the opacity of the thumbs to fade down to 60% when the page loads
5. $(".thumbs img").hover(function(){
7. $(**this**).fadeTo("slow", 1.0); // This should set the opacity to 100% on hover
9. },function(){
11. $(**this**).fadeTo("slow", 0.6); // This should set the opacity back to 60% on mouseout
13. });
15. });

来源: [Snipplr](http://snipplr.com/view/18606/)

**禁用文本/密码输入中的空格**

无论是电子邮件、用户名还是密码，很多常见字段都不需要使用空格。以下代码能够轻松禁用选定输入内容中的全部空格。

1. $('input.nospace').keydown(function(e) {
3. **if** (e.keyCode == 32) {
5. **return** **false**;
7. }
9. });

原文标题：10 jQuery Snippets for Efficient Web Development

【51CTO译稿，合作站点转载请注明原文译者和出处为51CTO.com】

【编辑推荐】

1. [Web开发人员不容错过的十款最佳HTML5响应式框架](http://developer.51cto.com/art/201601/504117.htm)
2. [10个非常炫酷的jQuery相册动画赏析](http://developer.51cto.com/art/201601/504275.htm)
3. [10个超赞的jQuery图片滑块动画](http://developer.51cto.com/art/201601/504712.htm)
4. [Jooby框架简化Java Web开发流程](http://developer.51cto.com/art/201601/504812.htm)
5. [成为Web开发人员的7个简单步骤](http://developer.51cto.com/art/201603/507101.htm)
6. [Stack Overflow发布开发者年度调查报告 全堆栈Web开发者占28%](http://developer.51cto.com/art/201603/507808.htm)

## 乐观锁和悲观锁

乐观并发控制(乐观锁)和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

乐观锁（给表加一个版本号字段）来防止更新丢失

乐观锁

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

一、悲观锁

要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。set autocommit=0;

悲观并发控制实际上是“先取锁再访问”的保守策略，为数据处理的安全提供了保证。

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

# 14开发工具使用(eclipse,浏览器，svn , git，)（包括使用方法，快捷键等）

## [Google浏览器](https://blog.csdn.net/superit401/article/details/51072074)

### [Google快捷键大全](https://blog.csdn.net/superit401/article/details/51072074)

url: <https://blog.csdn.net/superit401/article/details/51072074>

## [navicat](https://blog.csdn.net/superit401/article/details/78237780)

URL: <https://blog.csdn.net/superit401/article/category/6850102>

## Git



URL: <https://blog.csdn.net/superit401/article/category/6157643>

# 数据库（mysql oracle 语法总结 使用方式 等等）

# 17，[Linux命令](http://www.hollischuang.com/archives/800)

1.查找文件

find / -name filename.txt 根据名称查找/目录下的filename.txt文件。

find . -name "\*.xml" 递归查找所有的xml文件

find . -name "\*.xml" |xargs grep "hello world" 递归查找所有文件内容中包含hello world的xml文件

grep -H 'spring' \*.xml 查找所以有的包含spring的xml文件

find ./ -size 0 | xargs rm -f & 删除文件大小为零的文件

ls -l | grep '.jar' 查找当前目录中的所有jar文件

grep 'test' d\* 显示所有以d开头的文件中包含test的行。

grep 'test' aa bb cc 显示在aa，bb，cc文件中匹配test的行。

grep '[a-z]\{5\}' aa 显示所有包含每个字符串至少有5个连续小写字符的字符串的行。

2.查看一个程序是否运行

ps –ef|grep tomcat 查看所有有关tomcat的进程

ps -ef|grep --color java 高亮要查询的关键字

3.终止线程

kill -9 19979 终止线程号位19979的进程

4.查看文件，包含隐藏文件

ls -al

5.当前工作目录

pwd

6.复制文件

cp source dest 复制文件

cp -r sourceFolder targetFolder 递归复制整个文件夹

scp sourecFile romoteUserName@remoteIp:remoteAddr 远程拷贝

7.创建目录

mkdir newfolder

8.删除目录

rmdir deleteEmptyFolder 删除空目录 rm -rf deleteFile 递归删除目录中所有内容

9.移动文件

mv /temp/movefile /targetFolder

10.重命令

mv oldNameFile newNameFile

11.切换用户

su -username

12.修改文件权限

chmod 777 file.java //file.java的权限-rwxrwxrwx，r表示读、w表示写、x表示可执行

13.压缩文件

tar -czf test.tar.gz /test1 /test2

14.列出压缩文件列表

tar -tzf test.tar.gz

15.解压文件

tar -xvzf test.tar.gz

16.查看文件头10行

head -n 10 example.txt

17.查看文件尾10行

tail -n 10 example.txt

18.查看日志类型文件

tail -f exmaple.log //这个命令会自动显示新增内容，屏幕只显示10行内容的（可设置）。

19.使用超级管理员身份执行命令

sudo rm a.txt 使用管理员身份删除文件

20.查看端口占用情况

netstat -tln | grep 8080 查看端口8080的使用情况

21.查看端口属于哪个程序

lsof -i :8080

22.查看进程

ps aux|grep java 查看java进程

ps aux 查看所有进程

23.以树状图列出目录的内容

tree a

ps:Mac下使用tree命令

24. 文件下载

wget http://file.tgz mac下安装wget命令

curl http://file.tgz

25. 网络检测

ping www.just-ping.com

26.远程登录

ssh userName@ip

27.打印信息

echo $JAVA\_HOME 打印java home环境变量的值

28.java 常用命令

java javac jps ,jstat ,jmap, jstack

29.其他命令

svn git maven

28.linux命令学习网站:

http://explainshell.com/

参考资料：

Linux端口被占用的解决(Error: JBoss port is in use. Please check)

linux 中强大且常用命令：find、grep

Linux命令