前　言

EXT JS通常简称为EXT，它是一个非常优秀的Ajax框架，可以用来开发具有炫丽外观的富客户端应用。它是一个用JavaScript编写的与后台技术无关的Ajax框架。EXT绚丽多彩的界面吸引了许多程序员的眼球，同时也吸引了众多客户，它似乎一夜之间就迅速流行开来。对于企业应用系统，尤其是MIS类型的系统而言，EXT非常适用。

当我们第一次使用EXT时，就被它深深地吸引住了。对于像我们这样的没有美术功底的程序员来说，EXT为我们解决了一大难题，因为它天生拥有炫丽的外表。同时，有很多用其他技术无法实现或极难实现的功能，却能用EXT轻易实现，比如EXT中的表格、树形、布局等控件能为我们的日常开发工作节约大量的时间和精力，这些都坚定了我们使用EXT的决心。

我们在学习EXT的过程中做了大量笔记，记下了学习过程中的一些心得和体会，同时也写了很多示例程序，但是从未想过会将这些资料付诸出版。EXT的参考资料很缺乏，我们发现身边很多学习EXT的朋友都在黑暗中摸索，尤其是英文不太好的朋友，学习起来非常吃力。EXT的中文资料就更少了，虽然有人把EXT官方的API文档中文化了，但是API文档中只有一些基础理论和简单示例，并不能指导我们快速地去实践。我们是实用主义者，本书的最大特点就是以实例为基础，在实例的基础上讲解EXT的各种用法。这样既便于读者理解，也方便让读者亲自实践，从而迅速地将所学到的知识运用到实际项目中去。

本书适合有一定CSS和HTML基础的开发者阅读，它的主要目的是让开发者能快速学会EXT，并立即付诸实践。本书中的示例代码都是以EXT 2.2为基础的，也包含了即将发布的EXT 3.0中的新特性，对EXT的相关知识进行了深入而全面的阐述。

EXT发布包中有一个examples目录，是专门用来放置各种演示示例的，本书附带的所有示例[①]也可以直接放在这个目录下使用。使用时，请将对应目录放在EXT发布包的examples目录下，可以用浏览器打开示例HTML文件观看效果（见图0-1）。

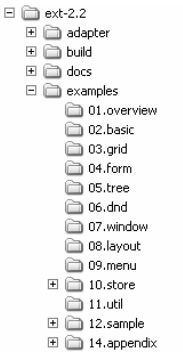

图0-1　examples目录展开图

示例中使用了localXHR.js，无需服务器就可以读取JSON数据，从而可以直接在本地浏览大部分示例。对于那些需要后台JSP提供数据的示例，最简单的方法是将整个EXT发布包复制到Tomcat的

① 这些示例可以从图灵公司网站上与本书对应的页面下载。——编者注

webapps目录下，启动Tomcat后，可通过浏览器访问examples下的示例。

出于对EXT的喜爱，又承蒙广大爱好EXT的朋友的错爱，我们写作了本书。如果有不恰当之处，敬请批评指正。为了便于与读者朋友交流，我们特为本书在我们的网站www.family168.com上开辟了专门的页面[①]。欢迎大家把对本书的意见和建议发到这个页面上来，我们会积极参与讨论，在此深表感谢。

最后，我们要感谢所有在本书的写作期间给予我们帮助和鼓励的朋友们，还有那些志同道合的EXT爱好者们。

徐会生　何启伟　康爱媛

2008年10月20日

① http://www.family168.com/extjs

目　录

第 10 章

数据存储与传输

10

本章内容

- Ext.data简介
- Ext.data.Connection
- Ext.data.Record
- Ext.data.Store
- 常用proxy
- 常用reader
- 高级store
- EXT中的Ajax
- 关于scope和createDelegate()
- DWR与EXT整合

10.1 Ext.data 简介

Ext.data在命名空间中定义了一系列store、reader和proxy。Grid和ComboxBox都是以Ext.data为媒介获取数据的，它包含异步加载、类型转换、分页等功能。Ext.data默认支持Array、JSON、XML等数据格式，可以通过Memory、HTTP、ScriptTag等方式获得这些格式的数据。如果要实现新的协议和新的数据结构，只需要扩展reader和proxy即可。DWRProxy就实现了自身的proxy和reader，让EXT可以直接从DWR获得数据。

10.2 Ext.data.Connection

Ext.data.Connection是对Ext.lib.Ajax的封装，它提供了配置使用Ajax的通用方式，它在内部通过Ext.lib.Ajax实现与后台的异步调用。与底层的Ext.lib.Ajax相比，Ext.data.Connection提供了更简洁的配置方式，使用起来更方便。

Ext.data.Connection主要用于在Ext.data.HttpProxy和Ext.data.ScriptTagProxy中执行与后台交互的任务，它会从指定的URL获得数据，并把后台返回的数据交给HttpProxy或ScriptTagProxy处理，Ext.data.Connection的使用方式如代码清单10-1所示。

代码清单10-1 使用Ext.data.Connection

```
var conn = new Ext.data.Connection({
    autoAbort: false,
    defaultHeaders: {
        referer: 'http://localhost:8080/'
    },
    disableCaching : false,
    extraParams : {
        name: 'name'
    },
    method : 'GET',
    timeout : 300,
    url : '01-01.txt'
});
```

在使用Ext.data.Connection之前，都要像上面这样创建一个新的Ext.Connection实例。我们可以在构造方法里配置对应的参数，比如autoAbort表示链接是否会自动断开、defaultHeaders参数表示请求的默认首部信息、disableCaching参数表示请求是否会禁用缓存、extraParams参数代表请求的额外参数、method参数表示请求方法、timeout参数表示连接的超时时间、url参数表示请求访问的网址等。

在创建了conn之后，可以调用request()函数发送请求，处理返回的结果，如下面的代码所示。

```
conn.request({
    success: function(response) {
        Ext.Msg.alert('info', response.responseText);
    },
    failure: function() {
        Ext.Msg.alert('warn', 'failure');
    }
});
```

Request()函数中可以设置success和failure两个回调函数，分别在请求成功和请求失败时调用。请求成功时，success函数的参数就是后台返回的信息。

我们再来看一下request函数中的其他参数。

- url:String：请求url。
- params:Object/String/Function：请求传递的参数。
- method:String：请求方法，通常为GET或POST。
- callback:Function：请求完成后的回调函数，无论是成功还是失败，都会执行。
- success:Function：请求成功时的回调函数。
- failure:Function：请求失败时的回调函数
- scope:Object：回调函数的作用域。
- form:Object/String：绑定的form表单。
- isUpload:Boolean：是否执行文件上传。

- headers:Object：请求首部信息。
- xmlData:Object：XML文档对象，可以通过URL附加参数的方式发起请求。
- disableCaching:Boolean：是否禁用缓存，默认为禁用。

Ext.data.Connection还提供了abort([Number transactionId])函数，当同时有多个请求发生时，根据指定的事务id放弃其中的某一个请求。如果不指定事务id，就会放弃最后一个请求。isLoading([Number transactionId])函数的用法与abort()类似，可以根据事务id判断对应的请求是否完成。如果未指定事务id，就判断最后一个请求是否完成。

10.3 Ext.data.Record

Ext.data.Record就是一个设定了内部数据类型的对象，它是Ext.data.Store的最基本组成部分。如果把Ext.data.Store看作是一张二维表，那么它的每一行就对应一个Ext.data.Record实例。

Ext.data.Record的主要功能是保存数据，并且在内部数据发生改变时记录修改的状态，它还可以保留修改之前的原始值。

我们使用Ext.data.Record时通常都是由create()函数开始，首先用create()函数创建一个自定义的Record类型，如下面的代码所示。

```
var PersonRecord = Ext.data.Record.create([
    {name: 'name', type: 'string'},
    {name: 'sex', type: 'int'}
]);
```

PersonRecord就是我们定义的新类型，包含字符串类型的name和整数类型的sex两个属性，然后我们使用new关键字创建PersonRecord的实例，如下面的代码所示。

```
var boy = new PersonRecord({
    name: 'boy',
    sex: 0
});
```

创建对象时，可以直接通过构造方法为对象赋予初始值，将'boy'赋值给name，0赋值给sex。

现在，我们得到了PersonRecord的实例boy，如何才能得到它的属性呢？以下三种方式都可以获得boy中name属性的数据，如下面的代码所示。

```
alert(boy.data.name);
alert(boy.data['name']);
alert(boy.get('name'));
```

这里涉及Ext.data.Record的data属性，这是定义在Ext.data.Record中的一个公共属性，用于保存当前record对象的所有数据。它是一个JSON对象，可以直接从它里面获得需要的数据。可以通过Ext.data.Record的get()函数方便地从data属性中获得指定的属性值。

如果我们需要修改boy中的数据，请不要使用以下方式直接操作data，如下面的代码所示。

```
boy.data.name = 'boy name';
boy.data['name'] = 'boy name';
```

而应该使用set()函数，如下面的代码所示。

```
boy.set('name', 'body name');
```

set()函数会判断属性值是否发生了改变，如果改变了，就要将当前对象的dirty属性设置为true，并将修改之前的原始值放入modified对象中，供其他函数使用。如果直接操作data中的值，record就无法记录属性数据的修改情况。

Record的属性数据被修改后，我们可以执行如下几种操作。

- commit()（提交）：这个函数的效果是设置dirty为false，并删除modified中保存的原始数据。
- reject()（撤销）：这个函数的效果是将data中已经修改了的属性值都恢复成modified中保存的原始数据，然后设置dirty为false，并删除保存原始数据的modified对象。
- getChanges()获得修改的部分：这个函数会把data中经过修改的属性和数据放在一个JSON对象里并返回。例如上例中，getChanges()返回的结果是{name:'body name'}。
- 我们还可以调用isModified()判断当前record中的数据是否被修改。

  Ext.data.Record还提供了用于复制record实例的函数copy()。

```
var copyBoy = boy.copy();
```

这样我们就得到了boy的一个副本，它里面包含了boy的data数据，但copy()函数不会复制dirty和modified等额外的属性值。

Ext.data.Record中其他的参数大多与Ext.data.Store有关，请参考与Ext.data.Store相关的讨论。

10.4 Ext.data.Store

Ext.data.Store是EXT中用来进行数据交换和数据交互的标准中间件，无论是Grid还是ComboBox，都是通过它实现数据读取、类型转换、排序分页和搜索等操作的。

Ext.data.Store中有一个Ext.data.Record数组，所有数据都存放在这些Ext.data.Record实例中，为后面的读取和修改操作做准备。

10.4.1 基本应用

在使用之前，首先要创建一个Ext.data.Store的实例，如下面的代码所示。

```
var data = [
    ['boy', 0],
    ['girl', 1]
];

var store = new Ext.data.Store({
    proxy: new Ext.data.MemoryProxy(data),
    reader: new Ext.data.ArrayReader({}, PersonRecord)
});
store.load();
```

每个store最少需要两个组件的支持，分别是proxy和reader，proxy用于从某个途径读取原始数据，reader用于将原始数据转换成Record实例。

这里我们使用的是Ext.data.MemoryProxy和Ext.data.ArrayReader，将data数组中的数据转换成对应的几个PersonRecord实例，然后放入store中。store创建完毕之后，执行store.load()实现这个转换过程。

经过转换之后，store里的数据就可以提供给Grid或ComboBox使用了，这就是Ext.data.Store的最基本用法。

10.4.2　对数据进行排序

Ext.data.Store提供了一系列属性和函数，利用它们对数据进行排序操作。

可以在创建Ext.data.Store时使用sortInfo参数指定排序的字段和排序方式，如下面的代码所示。

```
var store = new Ext.data.Store({
    proxy: new Ext.data.MemoryProxy(data),
    reader: new Ext.data.ArrayReader({}, PersonRecord),
    sortInfo: {field: 'name', direction: 'DESC'}
});
```

这样，在store加载数据之后，就会自动根据name字段进行降序排列。对store使用store.setDefaultSort('name','DESC');也会达到同样效果。

也可以在任何时候调用sort()函数，比如store.sort('name', 'DESC');，对store中的数据进行排序。

如果我们希望获得store的排序信息，可以调用getSortState()函数，返回的是类似{field: "name", direction: " DESC"}的JSON对象。

与排序相关的参数还有remoteSort，这个参数是用来实现后台排序功能的。当设置为remoteSort:true时，store会在向后台请求数据时自动加入sort和dir两个参数，分别对应排序的字段和排序的方式，由后台获取并处理这两个参数，在后台对所需数据进行排序操作。remoteSort:true也会导致每次执行sort()时都要去后台重新加载数据，而不能只对本地数据进行排序。

详细的用法可以参考第2章。

10.4.3　从 store 中获取数据

从store中获取数据有很多种途径，可以依据不同的要求选择不同的函数。最直接的方法是根据record在store中的行号获得对应的record，得到了record就可以使用get()函数获得里面的数据了，如下面的代码所示。

```
store.getAt(0).get('name')
```

通过这种方式，我们可以遍历store中所有的record，依次得到它们的数据，如下面的代码所示。

```
for (var i = 0; i < store.getCount(); i++) {
    var record = store.getAt(i);
    alert(record.get('name'));
}
```

Store.getCount()返回的是store中的所有数据记录，然后使用for循环遍历整个store，从而得到每条记录。

除了使用getCount()的方法外，还可以使用each()函数，如下面的代码所示。

```
store.each(function(record) {
    alert(record.get('name'));
});
```

Each()可以接受一个函数作为参数，遍历内部record，并将每个record作为参数传递给function()处理。如果希望停止遍历，可以让function()返回false。

也可以使用getRange()函数连续获得多个record，只需要指定开始和结束位置的索引值，如下面的代码所示。

```
var records = store.getRange(0, 1);
for (var i = 0; i < records.length; i++) {
    var record = records[i];
    alert(record.get('name'));
}
```

如果确实不知道record的id，也可以根据record本身的id从store中获得对应的record，如下面的代码所示。

```
store.getById(1001).get('name')
```

EXT还提供了函数find()和findBy()，可以利用它们对store中的数据进行搜索，如下面的代码所示。

```
find( String property, String/RegExp value, [Number startIndex], [Boolean anyMatch],
[Boolean caseSensitive] )
```

在这5个参数中，只有前两个是必须的。第一个参数property代表搜索的字段名；第二个参数value是匹配用字符串或正则表达式；第三个参数startIndex表示从第几行开始搜索，第四个参数anyMatch为true时，不必从头开始匹配；第五个参数caseSensitive为true时，会区分大小写。

如下面的代码所示：

```
var index = store.find('name','g');
alert(store.getAt(index).get('name'));
```

与find()函数对应的findBy()函数的定义格式如下：

```
findBy( Function fn, [Object scope], [Number startIndex] ) : Number
```

findBy()函数允许用户使用自定义函数对内部数据进行搜索。fn返回true时，表示查找成功，于是停止遍历并返回行号。fn返回false时，表示查找失败（即未找到），继续遍历，如下

面的代码所示。

```
index = store.findBy(function(record, id) {
    return record.get('name') == 'girl' && record.get('sex') == 1;
});
alert(store.getAt(index).get('name'));
```

通过findBy()函数，我们可以同时判断record中的多个字段，在函数中实现复杂逻辑。

我们还可以使用query和queryBy函数对store中的数据进行查询。与find和findBy不同的是，query和queryBy返回的是一个MixCollection对象，里面包含了搜索得到的数据，如下面的代码所示。

```
alert(store.query('name', 'boy'));
    alert(store.queryBy(function(record) {
        return record.get('name') == 'girl' && record.get('sex') == 1;
    }));
```

10.4.4　更新 store 中的数据

可以使用add(Ext.data.Record[] records)向store末尾添加一个或多个record，使用的参数可以是一个record实例，如下面的代码所示。

```
store.add(new PersonRecord({
    name: 'other',
    sex: 0
}));
```

Add()的也可以添加一个record数组，如下面的代码所示：

```
store.add([new PersonRecord({
    name: 'other1',
    sex: 0
}), new PersonRecord({
    name: 'other2',
    sex: 0
})]);
```

Add()函数每次都会将新数据添加到store的末尾，这就有可能破坏store原有的排序方式。如果希望根据store原来的排序方式将新数据插入到对应的位置，可以使用addSorted()函数。它会在添加新数据之后立即对store进行排序，这样就可以保证store中的数据有序地显示，如下面的代码所示。

```
store.addSorted(new PersonRecord({
    name: 'lili',
    sex: 1
}));
```

store会根据排序信息查找这条record应该插入的索引位置，然后根据得到的索引位置插入数据，从而实现对整体进行排序。这个函数需要预先为store设置本地排序，否则会不起作用。

如果希望自己指定数据插入的索引位置，可以使用insert()函数。它的第一个参数表示插入数据的索引位置，可以使用record实例或record实例的数组作为参数，插入之后，后面的数

据自动后移，如下面的代码所示。

```
store.insert(3, new PersonRecord({
    name: 'other',
    sex: 0
}));

store.insert(3, [new PersonRecord({
    name: 'other1',
    sex: 0
}), new PersonRecord({
    name: 'other2',
    sex: 0
})]);
```

删除操作可以使用remove()和removeAll()函数，它们分别可以删除指定的record和清空整个store中的数据，如下面的代码所示。

```
store.remove(store.getAt(0));
store.removeAll();
```

store中没有专门提供修改某一行record的操作，我们需要先从store中获取一个record。对这个record内部数据的修改会直接反映到store上，如下面的代码所示。

```
store.getAt(0).set('name', 'xxxx');
```

修改record的内部数据之后有两种选择：执行rejectChanges()撤销所有修改，将修改过的record恢复到原来的状态；执行commitChanges()提交数据修改。在执行撤销和提交操作之前，可以使用getModifiedRecords()获得store中修改过的record数组。

与修改数据相关的参数是pruneModifiedRecords，如果将它设置为true，当每次执行删除或reload操作时，都会清空所有修改。这样，在每次执行删除或reload操作之后，getModifiedRecords()返回的就是一个空数组，否则仍然会得到上次修改过的record记录。

10.4.5 加载及显示数据

store创建好后，需要调用load()函数加载数据，加载成功后才能对store中的数据进行操作。load()调用的完整过程如下面的代码所示。

```
store.load({
    params: {start:0,limit:20},
    callback: function(records, options, success){
        Ext.Msg.alert('info', '加载完毕');
    },
    scope: store,
    add: true
});
```

- params是在store加载时发送的附加参数。
- callback是加载完毕时执行的回调函数，它包含3个参数：records参数表示获得的数据，options表示执行load()时传递的参数，success表示是否加载成功。

- Scope用来指定回调函数执行时的作用域。
- Add为true时，load()得到的数据会添加在原来的store数据的末尾，否则会先清除之前的数据，再将得到的数据添加到store中。

一般来说，为了对store中的数据进行初始化，load()函数只需要执行一次。如果用params参数指定了需要使用的参数，以后再次执行reload()重新加载数据时，store会自动使用上次load()中包含的params参数内容。

如果有一些需要固定传递的参数，也可以使用baseParams参数执行，它是一个JSON对象，里面的数据会作为参数发送给后台处理，如下面的代码所示。

```
store.baseParams.start = 0;
store.baseParams.limit = 20;
```

为store加载数据之后，有时不需要把所有数据都显示出来，这时可以使用函数filter和filterBy对store中的数据进行过滤，只显示符合条件的部分，如下面的代码所示。

```
filter( String field, String/RegExp value, [Boolean anyMatch],
[Boolean caseSensitive] ) : void
```

filter()函数的用法与之前谈到的find()相似，如下面的代码所示。

```
store.filter('name', 'boy');
```

对应的filterBy()与findBy()类似，也可以在自定义的函数中实现各种复杂判断，如下面的代码所示。

```
store.filterBy(function(record) {
    return record.get('name') == 'girl' && record.get('sex') == 1;
});
```

如果想取消过滤并显示所有数据，那么可以调用clearFilter()函数，如下面的代码所示。

```
store.clearFilter();
```

如果想知道store上是否设置了过滤器，可以通过isFiltered()函数进行判断。

10.4.6　其他功能

除了上面提到的数据获取、排序、更新、显示等功能外，store还提供了其他一些功能函数。

```
collect( String dataIndex, [Boolean allowNull], [Boolean bypassFilter] ) : Array
```

collect函数获得指定的dataIndex对应的那一列的数据，当allowNull参数为true时，返回的结果中可能会包含null、undefined或空字符串，否则collect函数会自动将这些空数据过滤掉。当bypassFilter参数为true时，collect的结果不会受查询条件的影响，无论查询条件是什么都会忽略掉，返回的信息是所有的数据，如下面的代码所示。

```
alert(store.collect('name'));
```

这样会获得所有name列的值，示例中返回的是包含了'boy'和'girl'的数组。

getTotalCount()用于在翻页时获得后台传递过来的数据总数。如果没有设置翻页，get-

TotalCount()的结果与getCount()相同，都是返回当前的数据总数，如下面的代码所示。

```
alert(store.getTotalCount());
```

indexOf(Ext.data.Record record)和indexOfId(String id)函数根据record或record的id获得record对应的行号，如下面的代码所示。

```
alert(store.indexOf(store.getAt(1)));
alert(store.indexOfId(1001));
```

loadData(object data, [Boolean append])从本地JavaScript变量中读取数据，append为true时，将读取的数据附加到原数据后，否则执行整体更新，如下面的代码所示。

```
store.loadData(data, true);
```

Sum(String property, Number start, Number end):Number用于计算某一个列从start到end的总和，如下面的代码所示。

```
alert(store.sum('sex'));
```

如果省略参数start和end，就计算全部数据的总和。

store还提供了一系列事件（见表10-1），让我们可以为对应操作设定操作函数。

表10-1 store提供的事件

事件名	参 数
add	(Store this, Ext.data.Record[] records, Number index)
beforelaod	(Store this, Object options)
clear	(Store this)
datachanged	(Store this)
load	(Store this, Ext.data.Record[] records, Object options)
loadexception	()
metachange	(Store this, Object meta.)
remove	(Store this, Ext.data.Record record, Number index)
update	(Store this, Ext.data.Record record, String operation)

至此，store和record等组件已经讲解完毕，下面我们主要讨论一下常用的proxy和reader组件。

10.5 常用 proxy

10.5.1 MemoryProxy

MemoryProxy只能从JavaScript对象获得数据，可以直接把数组，或JSON和XML格式的数据交给它处理，如下面的代码所示。

```
var proxy = new Ext.data.MemoryProxy([
    ['id1','name1','descn1'],
    ['id2','name2','descn2']
]);
```

10.5.2 HttpProxy

HttpProxy使用HTTP协议，通过Ajax去后台取数据，构造它时需要设置url:'xxx.jsp'参数。这里的url可以替换成任何一个合法的网址，这样HttpProxy才知道去哪里获取数据，如下面的代码所示。

```
var proxy = new Ext.data.HttpProxy({url:'xxx.jsp'});
```

后台需要返回EXT所需要的JSON格式的数据，下面的内容就是后台使用JSP的一个范例，如下面的代码所示。

```
response.setContentType("application/x-json");
Writer out = response.getWriter();
out.print("[" +
        "['id1','name1','descn1']" +
        "['id2','name2','descn2']" +
    "]");
```

请注意，这里的HttpProxy不支持跨域，它只能从同一域中获得数据。如果想跨域，请参考下面的ScriptTagProxy。

10.5.3 ScriptTagProxy

ScriptTagProxy的用法几乎和HttpProxy一样，如下面的代码所示。

```
var proxy = new Ext.data.ScriptTagProxy({url:'xxx.jsp'});
```

从这里也看不出来它是如何支持跨域的，我们还需要在后台进行相应的处理，如下面的代码所示。

```
String cb = request.getParameter("callback");
response.setContentType("text/javascript");
Writer out = response.getWriter();
out.write(cb + "(");
out.print("[" +
        "['id1','name1','descn1']" +
        "['id2','name2','descn2']" +
    "]");
out.write(");");
```

其中的关键就在于从请求中获得的callback参数，这个参数叫做回调函数。ScriptTagProxy会在当前的HTML页面里添加一个<script type="text/javascript"src="xxx.jsp"></script>标签，然后把后台返回的内容添加到这个标签中，这样就可以解决跨域访问数据的问题。为了让后台返回的内容可以在动态生成的标签中运行，EXT会生成一个名为callback的回调函数，并把回调函数的名称传递给后台，由后台生成callback(data)形式的响应内容，然后返回给前台自动运行。

虽然上述处理过程比较难理解，但是我们只需要了解ScriptTagProxy的用法就足够了。如果还想进一步了解ScriptTagProxy的运行过程，可以使用Firebug查看动态生成的HTML以及响

应的JSON内容。

最后我们来分析一下EXT的API文档中提供的示例，这段后台代码会自动判断请求的类型，返回支持ScriptTagProxy或HttpProxy的数据，如代码清单10-2所示。

代码清单10-2 在后台同时支持HttpProxy和ScriptTagProxy

```
boolean scriptTag = false;
String cb = request.getParameter("callback");
if (cb != null) {
    scriptTag = true;
    response.setContentType("text/javascript");
} else {
    response.setContentType("application/x-json");
}
Writer out = response.getWriter();
if (scriptTag) {
    out.write(cb + "(");
}
out.print(dataBlock.toJsonString());
if (scriptTag) {
    out.write(");");
}
```

代码中通过判断请求中是否包含callback参数来决定返回何种数据类型。如果包含，就返回ScriptTagProxy需要的数据；否则，就当作HttpProxy处理。

10.6 常用 Reader

10.6.1 ArrayReader

从proxy中读取的数据需要进行解析，这些数据转换成Record数组后才能提供给Ext.data.Store使用。

ArrayReader的作用是从二维数组里依次读取数据，然后生成对应的Record。默认情况下是按列顺序读取数组中的数据，不过你也可以考虑用mapping指定record与原始数组对应的列号。ArrayReader的用法很简单，但缺点是不支持分页。使用二维数组的方式如下面的代码所示。

```
var data = [
    ['id1','name1','descn1'],
    ['id2','name2','descn2']
];
```

对应的ArrayReader如下面的代码所示。

```
var reader = new Ext.data.ArrayReader({
    id:1
},[
    {name:'name',mapping:1},
    {name:'descn',mapping:2},
    {name:'id',mapping:0},
```

```
]);
```

我们演示的是字段顺序不一致的情况，如果字段顺序和列顺序一致，就不用额外配置mapping。

10.6.2 JsonReader

在JavaScript中，JSON是一种非常重要的数据格式，key:value的形式比XML那种复杂的标签结构更容易理解，代码量也更小，很多人倾向于使用它作为EXT的数据交换格式。为JsonReader准备的JSON数据如下面的代码所示。

```
var data = {
    id:0,
    totalProperty:2,
    successProperty:true,
    root:[
        {id:'id1',name:'name1',descn:'descn1'},
        {id:'id2',name:'name2',descn:'descn2'}
    ]
};
```

与数组相比，JSON的最大优点就是支持分页，我们可以使用totalProperty参数表示数据的总量。successProperty参数是可选的，可以用它判断当前请求是否执行成功，进而判断是否进行数据加载。在不希望JsonReader处理响应数据时，可以把successProperty设置成false。

现在来讨论一下JsonReader，看看它是如何与上面的JSON数据对应的，如下面的代码所示。

```
var reader = new Ext.data.JsonReader({
    successProperty: "successproperty",
    totalProperty: "totalProperty",
    root: "root",
    id: "id"
}, [
    {name:'id',mapping:'id'},
    {name:'name',mapping:'name'},
    {name:'descn',mapping:'descn'}
]);
```

上例中的对应方式不够简洁，因为name和mapping部分的内容是相同的，其实这里的mapping可以省略，默认会用name参数从JSON中获得对应的数据。如果不想与JSON里的名字一样，也可以使用mapping修改。不过，mapping在这里还有其他用途，如代码清单10-3所示。

代码清单10-3　为JsonReader设置mapping进行数据映射

```
var data = {
    id:0,
    totalProperty:2,
    successProperty:true,
    root:[
        {id:'id1',name:'name1',descn:'descn1',person:{
```

```
            id:1,name:'man',sex:'male'
        }},
        {id:'id2',name:'name2',descn:'descn2',person:{
            id:2,name:'woman',sex:'female'
        }}
    ]
};
var reader = new Ext.data.JsonReader({
    successProperty: "successproperty",
    totalProperty: "totalProperty",
    root: "root",
    id: "id"
}, [
    'id','name','descn',
    {name:'person_name',mapping:'person.name'},
    {name:'person_sex',mapping:'person.sex'}
]);
```

在上面的代码中，我们使用JSON支持更复杂的嵌套结构，其中的person对象自身就拥有id、name和sex等属性。在JsonReader中可以用mapping把这些嵌套的内部属性映射出来，赋予对应的record，而其他字段都不变。

10.6.3 XmlReader

XML是非常通用的数据传输格式，XmlReader使用的XML格式的数据如代码清单10-4所示。

代码清单10-4 XmlReader使用的XML格式的数据

```
<?xml version="1.0" encoding="utf-8"?>
<dataset>
    <id>1</id>
    <totalRecords>2</totalRecords>
    <success>true</success>
    <record>
        <id>1</id>
        <name>name1</name>
        <descn>descn1</descn>
    </record>
    <record>
        <id>2</id>
        <name>name2</name>
        <descn>descn2</descn>
    </record>
</dataset>
```

这里一定要用dataset作为XML根元素。再让我们看一下如何对XmlReader进行配置，从而读取上面示例中的XML数据，如下面的代码所示。

```
var reader = new Ext.data.XmlReader({
   totalRecords: 'totalRecords',
   success: 'success'
   record: 'record',
   id: "id"
```

```
}, ['id','name','descn']);
```

XmlReader使用的参数与之前介绍的JsonReader有些不同，我们可以看到这里用到了totalRecords和record两个参数，其中totalRecords用来指定从'totalRecords'标签里获得后台数据总数，record则表示XML中放在record标签里的数据是我们需要显示的结果数据。其他两个参数success和id的含义和JsonReader中对应的参数相似，分别用来判断操是否成功和这次返回的id。因为XML中的标签和reader里需要的名字是相同的，所以简化了配置，将[{name:'id'},{name:'name'},{name:'descn'}]直接写成了['id','name','descn']。

因为XmlReader不能将JavaScript中的字符串自动解析成XML格式的数据，因此我们需要利用其他方法进行演示。参考localXHR.js中构造XML的方式，我们有了下面的解决方案，如代码清单10-5所示。

代码清单10-5　通过本地字符串构造XML对象

```
var data = "<?xml version='1.0' encoding='utf-8'?>" +
    "<dataset>" +
        "<id>1</id>" +
        "<totalRecords>2</totalRecords>" +
        "<success>true</success>" +
        "<record>" +
            "<id>1</id>" +
            "<name>name1</name>" +
            "<descn>descn1</descn>" +
        "</record>" +
        "<record>" +
            "<id>2</id>" +
            "<name>name2</name>" +
            "<descn>descn2</descn>" +
        "</record>" +
    "</dataset>";

var xdoc;

if(typeof(DOMParser) == 'undefined'){
    xdoc = new ActiveXObject("Microsoft.XMLDOM");
    xdoc.async="false";
    xdoc.loadXML(data);
}else{
    var domParser = new DOMParser();
    xdoc = domParser.parseFromString(data, 'application/xml');
    domParser = null;
}

var proxy = new Ext.data.MemoryProxy(xdoc);

var reader = new Ext.data.XmlReader({
    totalRecords: 'totalRecords',
    success: 'success',
    record: 'record',
```

```
    id: "id"
}, ['id','name','descn']);

var ds = new Ext.data.Store({
    proxy: proxy,
    reader: reader
});
```

10.7 高级 store

实际开发时，并不需要每次都对proxy、reader、store这三个对象进行配置，EXT为我们提供了几种可选择的整合方案。

- SimpleStore = Store + MemoryProxy + ArrayReader

```
var ds = Ext.data.SimpleStore({
    data: [
          ['id1','name1','descn1'],
          ['id2','name2','descn2']
    ],
    fields: ['id','name','descn']
});
```

SimpleStore是专为简化读取本地数组而设计的，设置上MemoryProxy需要的data和ArrayReader需要的fields就可以使用了。

- JsonStore = Store + HttpProxy + JsonReader

```
var ds = Ext.data.JsonStore({
    url: 'xxx.jsp',
    root: 'root',
    fields: ['id','name','descn']
});
```

JsonStore将JsonReader和HttpProxy整合在一起，提供了一种从后台读取JSON信息的简便方法，大多数情况下可以考虑直接使用它从后台读取数据。

- Ext.data.GroupingStore对数据进行分组

Ext.data.GroupingStore继承自Ext.data.Store，它的主要功能是可以对内部的数据进行分组，我们可以在创建Ext.data.GroupingStore时指定根据某个字段进行分组，也可以在创建实例后调用它的groupBy()函数对内部数据重新分组，如下面的代码所示。

```
var ds = new Ext.data.GroupingStore({
    data: [
        ['id1','name1','female','descn1'],
        ['id2','name2','male','descn2'],
        ['id3','name3','female','descn3'],
        ['id4','name4','male','descn4'],
        ['id5','name5','female','descn5']
    ],
    reader: new Ext.data.ArrayReader({
        fields: ['id','name','sex','descn']
```

```
        }),
        groupField: 'sex',
        groupOnSort: true
    });
```

上例中，我们使用groupField作为参数，为Ext.data.Grouping设置了分组字段，另外还设置了groupOnSort参数，这个参数可以保证只有在进行分组时才会对Ext.data.GroupingStore内部的数据进行排序。如果采用默认值，就需要手工指定sortInfo参数，从而指定默认的排序字段和排序方式，否则就会出现错误。

创建Ext.data.GroupingStore的实例之后，我们还可以调用groupBy()函数重新对数据进行分组。因为我们设置了groupOnSort:true，所以在重新分组时，EXT会使用分组的字段对内部数据进行排序。如果不希望对数据进行分组，也可以调用clearGrouping()函数清除分组信息，如下面的代码所示。

```
ds.groupBy('id');
ds.clearGrouping();
```

10.8　EXT 中的 Ajax

EXT与后台交换数据时，很大程度上依赖于底层实现的Ajax。所谓底层实现，就是说很可能就是我们之前提到的 Prototype、jQuery或YUI中提供的Ajax功能。为了统一接口，EXT在它们的基础上进行了封装，让我们可以用同一种写法“游走”于各种不同的底层实现之间。

10.8.1　最容易看到的 Ext.Ajax

Ext.Ajax的基本用法如下所示。

```
Ext.Ajax.request({
    url: '07-01.txt',
    success: function(response) {
        Ext.Msg.alert('成功', response.responseText);
    },
    failure: function(response) {
        Ext.Msg.alert('失败', response.responseText);
    },
    params: { name: 'value' }
});
```

这里调用的是Ext.Ajax的request函数，它的参数是一个JSON对象，具体如下所示。

- url参数表示将要访问的后台网址。
- success参数表示响应成功后的回调函数。

上例中我们直接从response取得返回的字符串，用Ext.Msg.alert显示出来。

- failure参数表示响应失败后的回调函数。

注意，这里的响应失败并不是指数据库操作之类的业务性失败，而是指HTTP返回404或500错误，请不要把HTTP响应错误与业务错误混淆在一起。

- params参数表示请求时发送到后台的参数，既可以使用JSON对象，也可以直接使用"name=value"形式的字符串。

上面的示例可以在10.store/07-01.html中找到。

Ext.Ajax直接继承自Ext.data.Connection，不同的是，它是一个单例，不需要用new创建实例，可以直接使用。在使用Ext.data.Connection前需要先创建实例，因为Ext.data.Connection是为了给Ext.data中的各种proxy提供Ajax功能，分配不同的实例更有利于分别管理。Ext.Ajax为用户提供了一个简易的调用接口，实际使用时，可以根据自己的需要进行选择。

10.8.2 Ext.lib.Ajax 是更底层的封装

其实Ext.Ajax和Ext.data.Connection的内部功能实现都是依靠Ext.lib.Ajax来完成的，在Ext.lib.Ajax下面就是各种底层库的Ajax了。

如果使用Ext.lib.Ajax实现以上的功能，就需要写成下面的形式，如下面的代码所示。

```
Ext.lib.Ajax.request(
    'POST',
    '07-01txt',
    {success: function(response){
        Ext.Msg.alert('成功', response.responseText);
    },failure: function(){
        Ext.Msg.alert('失败', response.responseText);
    }},
    'data=' + encodeURIComponent(Ext.encode({name:'value'}))
);
```

我们可以看到，使用Ext.lib.Ajax时需要传递4个参数，分别为method、url、callback和params。它们的含义与Ext.Ajax中的参数都是一一对应的，唯一没有提到过的method参数表示请求HTTP的方法，它也可以在Ext.Ajax中使用method:'POST'的方式设置。

相对于Ext.Ajax来说，Ext.lib.Ajax有如下几个缺点。

- 参数的顺序被定死了，第一个参数是method，第二个参数是url，第三个参数是回调函数callback，第四个参数是params。这样既不容易记忆，也无法省略其中某个不需要的参数。Ext.Ajax中用JSON对象来定义参数，使用起来更灵活。
- 在params部分，Ext.lib.Ajax必须使用字符串形式，显得有些笨重。Ext.Ajax则可以在JSON对象和字符串之间随意选择，非常灵活。

比与Ext.Ajax相比，Ext.lib.Ajax的唯一优势就是它可以在EXT 1.x中使用。如果你使用的是EXT 2.0或更高的版本，那么就放心大胆地使用Ext.Ajax吧，它会带给你更多的惊喜。

该示例在10.store/07-02.html中。

10.9 关于 scope 和 createDelegate()

关于JavaScript中this的使用，这是一个由来已久的问题了。我们这里就不介绍它的发展历史了，只结合具体的例子，告诉大家可能会遇到什么问题，在遇到这些问题时EXT是如何解决的。在使用EXT时，最常碰到的就是使用Ajax回调函数时出现的问题，如下面的代码所示。

```
<input type="text" name="text" id="text">
<input type="button" name="button" id="button" value="button">
```

现在的HTML 页面中有一个text输入框和一个按钮。我们希望按下这个按钮之后，能用Ajax去后台读取数据，然后把后台响应的数据放到text中，实现过程如代码清单10-6所示。

代码清单10-6　Ajax中使用回调函数

```
function doSuccess(response) {
    text.dom.value = response.responseText;
}

Ext.onReady(function(){
    Ext.get('button').on('click', function(){
        var text = Ext.get('text');
        Ext.lib.Ajax.request(
            'POST',
            '08.txt',
            {success:doSuccess},
            'param=' + encodeURIComponent(text.dom.value)
        );
    });
});
```

在上面的代码中，Ajax已经用Ext.get('text')获得了text，以为后面可以直接使用，没想到回调函数success不会按照你写的顺序去执行。当然，也不会像你所想的那样使用局部变量text。实际上，如果什么都不做，仅仅只是使用回调函数，你不得不再次使用Ext.get('text')重新获得元素，否则浏览器就会报text未定义的错误。

在此使用Ext.get('text')重新获取对象还比较简单，在有些情况下不容易获得需要处理的对象，我们要在发送Ajax请求之前获取回调函数中需要操作的对象，有两种方法可供选择：scope和createDelegate。

- 为Ajax设置scope。

```
function doSuccess(response) {
    this.dom.value = response.responseText;
}
Ext.lib.Ajax.request(
    'POST',
    '08.txt',
    {success:doSuccess,scope:text},
    'param=' + encodeURIComponent(text.dom.value)
);
```

在Ajax的callback参数部分添加一个scope:text，把回调函数的scope指向text，它的作用就是把doSuccess函数里的this指向text对象。然后再把doSuccess里改成this.dom. value，这样就可以了。如果想再次在回调函数里用某个对象，必须配上scope，这样就能在回调函数中使用this对它进行操作了。

- 为success添加createDelegate()。

```
function doSuccess(response) {
    this.dom.value = response.responseText;
}

Ext.lib.Ajax.request(
    'POST',
    '08.txt',
    {success:doSuccess.createDelegate(text)},
    'param=' + encodeURIComponent(text.dom.value)
);
```

createDelegate只能在function上调用，它把函数里的this强行指向我们需要的对象，然后我们就可以在回调函数doSuccess里直接通过this来引用createDelegate()中指定的这个对象了。它可以作为解决this问题的一个备选方案。

如果让我选择，我会尽量选择scope，因为createDelegate是要对原来的函数进行封装，重新生成function对象。简单环境下，scope就够用了，倒是createDelegate还有其他功能，比如修改调用参数等。

示例在10.store/08.html中。

10.10 DWR 与 EXT 整合

据不完全统计，从事Ajax开发的Java程序员有一大半都使用DWR。我们下面来介绍一下如何在EXT中使用DWR与后台交互。

10.10.1 在 EXT 中直接使用 DWR

因为DWR在前台的表现形式和普通的JavaScript完全一样，所以我们不需要特地去做些什么，直接使用EXT调用DWR生成的JavaScript函数即可。以Grid为例，比如现在我们要显示一个通讯录的信息，后台记录的数据有：id、name、sex、email、tel、addTime和descn。编写对应的POJO，代码如下所示。

```
public class Info {
    long id;
    String name;
    int sex;
    String email;
    String tel;
    Date addTime;
    String descn;
}
```

然后编写操作POJO的manager类，代码如下所示。

```
public class InfoManager {
    private List infoList = new ArrayList();

    public List getResult() {
```

```
            return infoList;
        }
    }
```

代码部分有些删减，我们只保留了其中的关键部分，就这样把这两个类配置到dwr.xml中，让前台可以对这些类进行调用。

下面是EXT与DWR交互的关键部分，我们要对JavaScript部分做如下修改，如代码清单10-7所示。

代码清单10-7　使用EXT调用DWR

```
var cm = new Ext.grid.ColumnModel([
    {header:'编号',dataIndex:'id'},
    {header:'名称',dataIndex:'name'},
    {header:'性别',dataIndex:'sex'},
    {header:'邮箱',dataIndex:'email'},
    {header:'电话',dataIndex:'tel'},
    {header:'添加时间',dataIndex:'addTime'},
    {header:'备注',dataIndex:'descn'}
]);

var store = new Ext.data.JsonStore({
    fields: ["id","name","sex",'email','tel','addTime','descn']
});

// 调用DWR取得数据
infoManager.getResult(function(data) {
    store.loadData(data);
});

var grid = new Ext.grid.GridPanel({
    renderTo: 'grid',
    store: store,
    cm: cm
});
```

注意，执行infoManager.getResult()函数时，DWR就会使用Ajax去后台取数据了，操作成功后调用我们定义的匿名回调函数。在这里我们只做一件事，那就是将返回的data直接注入到ds中。

DWR返回的data可以被JsonStore直接读取，我们需要设置对应的fields参数，以告诉JsonReader需要哪些属性。

在这里，EXT和DWR两者之间没有任何关系，将它们任何一方替换掉都可以。实际上它们只是在一起运行，并没有整合。我们给出的这个示例也是说明了一种松耦合的可能性，实际操作中完全可以使用这种方式。

10.10.2　DWRProxy

要结合使用EXT和DWR，不需要对后台程序进行任何修改，可以直接让前后台数据进行交

互。不过还要考虑很多细节，比如Grid分页、刷新、排序、搜索等常见的操作。EXT的官方网站上已经有人放上了DWRProxy，借助它可以让DWR和EXT连接得更加紧密。不过，需要在后台添加DWRProxy所需要的Java类，这可能不是最好的解决方案。但我们相信，通过对它的内在实现的讨论，我们可以有更多的选择和想象空间。

注意 这个DWRProxy.js一定要放在ext-base.js和ext-all.js后面，否则会出错。

我们现在就用DWRProxy来实现一个分页的示例。除了准备好插件DWRProxy.js外，还要在后台准备一个专门用于分页的封装类。因为不仅要告诉前台显示哪些数据，还要告诉前台一共有多少条数据。现在我们来重点看一下ListRange.java，如下面的代码所示。

```
public class ListRange {
    Object[] data;
    int totalSize;
}
```

其实ListRange非常简单，只有两个属性：提供数据的data和提供数据总量的totalSize。再看一下InfoManager.java，为了实现分页，我们专门编写了一个getItems方法，代码如下所示。

```
public ListRange getItems(Map conditions) {
    int start = 0;
    int pageSize = 10;
    int pageNo = (start / pageSize) + 1;

    try {
        start = Integer.parseInt(conditions.get("start").toString());
        pageSize = Integer.parseInt(conditions.get("limit").toString());
        pageNo = (start / pageSize) + 1;
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    List list = infoList.subList(start, start + pageSize);
    return new ListRange(list.toArray(), infoList.size());
}
```

getItems()的参数是Map，我们从中获得需要的参数，比如start和limit。不过HTTP里的参数都是字符串，而我们需要的是数字，所以要对类型进行相应的转换。根据start和limit两个属性从全部数据中截取一部分，放进新建的ListRange中，然后把生成的ListRange返回给前台，于是一切都解决了。

重头戏要上演了，我们就要使用传说中的Ext.data.DWRProxy了，还有Ext.data.ListRangeReader。通过这两个扩展，EXT完全可以支持DWR的数据传输协议。实际上，这正是EXT要把数据和显示分离设计的原因，这样你只需要添加自定义的proxy和reader，不需要修改EXT的其他部分，就可以实现从特定途径获取数据的功能。后台还是DWR，所以至少在Grid部分，我们可以很好地使用它们的结合，主要代码如下所示。

```
var store = new Ext.data.Store({
    proxy: new Ext.data.DWRProxy(infoManager.getItems, true),
    reader: new Ext.data.ListRangeReader({
        totalProperty: 'totalSize',
        root: 'data',
        id: 'id'
    }, info),
    remoteSort: true
});
```

与我们上面说的一样，我们修改了proxy，也修改了reader，其他地方都不需要进行修改，Grid已经可以正常运行了。需要提醒的是DWRProxy的用法，其中包括两个参数：第一个是dwrCall，它把一个DWR函数放进去，它对应的是后台的getItems方法；第二个参数是pagingAndSort，这个参数控制DWR是否需要分页和排序。

ListRangeReader部分与后台的ListRange.java对应。totalProperty表示后台数据总数，我们通过它指定从ListRange中读取totalSize属性的值来作为后台数据总数。还需要指定root参数，以告诉它在ListRange中的数据变量的名称为data，随后DWRProxy会从ListRange中的data属性中获取数据并显示到页面上。如果不想使用我们提供的ListRange.java类，也可以自己创建一个类，只要把totalProperty和data两个属性与之对应即可。

10.10.3 DWRTreeLoader

我们现在来尝试一下让树形也支持DWR。有了前面的基础，整合DWR和tree就更简单了。在后台，我们需要树形节点对应的TreeNode.java。目前，只要id、text和leaf三项就可以了。

```
public class TreeNode {
    String id;
    String text;
    boolean leaf;
}
```

id是节点的唯一标记，知道了id就能知道是在触发哪个节点了。text是显示的标题，leaf比较重要，它用来标记这个节点是不是叶子。

这里还是用异步树，TreeNodeManager.java里的getTree()方法将获得一个节点的id作为参数，然后返回这个节点下的所有子节点。我们这里没有限制生成的树形的深度，你可以根据自己的需要进行设置。TreeNodeManager.java的代码如下所示。

```
public List getTree(String id) {
    List list = new ArrayList();
    String seed1 = id + 1;
    String seed2 = id + 2;
    String seed3 = id + 3;
    list.add(new TreeNode(seed1, "" + seed1, false));
    list.add(new TreeNode(seed2, "" + seed2, false));
    list.add(new TreeNode(seed3, "" + seed3, true));

    return list;
}
```

上面的代码并不复杂，它实现的效果与在Java中使用List或数组是相同的，因为返回给前台的数据都是JSON格式的。前台使用JavaScript处理返回信息的部分更简单，先引入DWRTree-Loader.js，然后把TreeLoader替换成DWRTreeLoder即可，如下面的代码所示。

```
var tree = new Ext.tree.TreePanel('tree', {
    loader: new Ext.tree.DWRTreeLoader({dataUrl: treeNodeManager.getTree})
});
```

参数依然是dataUrl，它的值treeNodeManager.getTree代表的是一个DWR函数，我们不需要对它进行深入研究，它的内部会自动处理数据之间的对应关系。DWR有时真的很方便。

10.10.4 DWRProxy 和 ComboBox

DWRProxy既然可以用在Ext.data.Store中，那么它也可以为ComboBox服务，如代码清单10-8所示。

代码清单10-8 DWRProxy与ComboBox整合

```
var info = Ext.data.Record.create([
    {name: 'id', type: 'int'},
    {name: 'name', type: 'string'}
]);

var store = new Ext.data.Store({
    proxy: new Ext.data.DWRProxy(infoManager.getItems, true),
    reader: new Ext.data.ListRangeReader({
        totalProperty: 'totalSize',
        root: 'data',
        id: 'id'
    }, info)
});

var combo = new Ext.form.ComboBox({
    store: store,
    displayField: 'name',
    valueField: 'id',
    triggerAction: 'all',
    typeAhead: true,
    mode: 'remote',
    emptyText: '请选择',
    selectOnFocus: true
});
combo.render('combo');
```

我们既可以用mode:'remote'和triggerAction:'all'在第一次选择时读取数据，也可以设置mode:'local'，然后手工操作store.load()并读取数据。

DWR要比Json-lib方便得多，而且DWR返回的数据可以直接作为JSON使用，使用Json-lib时还要面对无休无止的循环引用。

这次的示例稍微复杂一些，因为包括依赖jar包、class、XML和JSP，所以示例单独放在

10.store/dwr2/下，请将它们复制到tomcat的webapps下，然后再使用浏览器访问。

10.11　localXHR 支持本地使用 Ajax

Ajax是不能在本地文件系统中使用的，必须把数据放到服务器上。无论是IIS、Apache、Tomcat，还是你熟悉的其他服务器，只要支持HTTP协议，就可以使用EXT中的Ajax。

至于本地为何不能用Ajax，主要是因为Ajax要判断HTTP响应返回的状态，只有status=200时才认为这次请求是成功的。所以，localXHR做的就是强行修改响应状态，让Ajax可以继续下去。

下面我们来分析一下localXHR的源代码。

- 加入了一个forceActiveX属性，默认是false，它用来控制是否强制使用activex，activex是在IE下专用的。
- 修改createXhrObject函数，只是在最开始处加了一条判断语句，如下所示。

```
if(Ext.isIE7 && !!this.forceActiveX){throw("IE7forceActiveX");}
```

- 增加了getHttpStatus函数，这是为了处理HTTP的响应状态，如代码清单10-9所示。

代码清单10-9　处理HTTP响应状态

```
getHttpStatus: function(reqObj){
    var statObj = {
        status:0
        ,statusText:''
        ,isError:false
        ,isLocal:false
        ,isOK:false
    };
    try {
        if(!reqObj)throw('noobj');
        statObj.status = reqObj.status || 0;

        statObj.isLocal = !reqObj.status && location.protocol == "file:" ||
                          Ext.isSafari && reqObj.status == undefined;

        statObj.statusText = reqObj.statusText || '';

        statObj.isOK = (statObj.isLocal ||
                        (statObj.status > 199 && statObj.status < 300) ||
                         statObj.status == 304);

    } catch(e){
        //status may not avail/valid yet.
        statObj.isError = true;
    }

    return statObj;
},
```

它为状态增添了更多语义，status表示状态值，statusText表示状态描述，isError表示是否有错误，isLocal表示是否在本地进行Ajax访问，isOK表示操作是否成功。

判断isLocal是否为本地的有两种方法：reqObj没有status，而且请求协议是file:；浏览器是Safari，而且reqObj.status没有定义。

statObj中的isOK属性用来判断此次请求是否成功。判断请求是否成功的条件很多，例如：isLocal的属性为true、响应状态值在199~300之间、响应状态值是304等。如果处理过程中出现了异常，就会将isError属性设置为true，最后会把配置好的statObj对象返回，等待下一个步骤的处理。

localXHR.js对handleTransactionResponse函数进行了简化。因为增加的getHttpStatus函数很好地封装了与请求相关的各种状态信息，所以在handleTransactionResponse函数中我们不会看到让人头晕目眩的响应状态代码。取而代之的是isError和isOK这些更容易理解的属性，localXHR.js直接使用这些属性来处理响应。

createResponseObject函数被大大强化了。其实前半部分都是一样的，localXHR.js中对isLocal做了大量的处理，响应中的responseText可以从连接中获得。如果需要XML，它就使用ActiveXObject("Microsoft.XMLDOM")或new DOMParser()把responseText解析成XML放到response里，响应状态也是重新计算的，这样就能让Ajax正常调用了。

最后处理的是asyncRequest函数，如果在异步请求时出现异常，就调用handleTransactionResponse返回响应，然后根据各种情况稍微修改header属性。

我们来看看下面这行代码：

```
Ext.lib.Ajax.forceActiveX = (document.location.protocol == 'file:');
```

如果协议是file:，就强制使用activex。

10.12 本章小结

本章系统地讨论了Ext.data包中的各个类的功能和使用方式，还涉及如何将EXT与DWR通过自定义的proxy相结合的示例。我们介绍了如何使用Ext.data.Connection与后台进行数据交互，还专门介绍了它的子类Ext.Ajax，并讨论了EXT中Ajax的应用以及在回调函数中使用scope或createDelegate()解决this的问题。

接着详细介绍了类Ext.data.Record和Ext.data.Store的功能和使用方法，这两个类结合起来形成了Ext.data中的主体数据模型，很多组件（包括Grid和ComboBox）都是建立在它们之上的。除此之外，还讨论了常用的proxy、reader、store：SimpleStore和JsonStore，以及它们的应用场景。

最后我们介绍了扩展插件localXHR.js，它可以解决EXT中Ajax无法访问本地文件的问题。

第 12 章

12 一个完整的EXT应用

本章内容

- 确定整体布局
- 使用HTML和CSS设置静态信息
- 对学生信息进行数据建模
- 在页面中显示学生信息列表
- 添加表单编辑学生信息
- 为表单添加提交事件
- 清空表单信息
- 删除指定的学生信息
- 在Grid和Form之间进行数据交互

在本章中，我们将综合运用前面所学的知识，开发一个简单的学生信息管理系统（如图12-1所示）。该系统的主要功能包括：显示学生信息、添加学生信息、修改学生信息，以及删除学生信息。这些功能的实现非常简单，我们在这里将演示如何在EXT中实现这些常用功能。

图12-1　学生信息管理系统界面

12.1　确定整体布局

在我们动手实现这些功能操作之前，首先应该确定页面的整体布局。在这里，我们用

BorderLayout把页面分隔成4个部分：最上方显示系统的名称，最下方显示版权信息，中间部分左侧显示学生信息列表，中间部分右侧中的表单用来添加或修改学生信息。

实现上述布局效果的代码如代码清单12-1所示。

代码清单12-1　实现学生信息管理系统的布局

```
Ext.onReady(function() {
    // layout start
    var viewport = new Ext.Viewport({
        layout: 'border',
        items: [{
            region: 'north',
            html: 'head'
        }, {
            region: 'center',
            html: 'grid'
        }, {
            region: 'east',
            html: 'form'
        }, {
            region: 'south',
            html: 'foot'
        }]
    });
    // layout end
});
```

我们使用`Ext.Viewport`为整个页面进行了布局设置，其中每个部分都直接用HTML参数做了标记，现在我们就可以看到这个布局的显示效果如图12-2所示。

图12-2　学生信息管理系统布局效果图

如图12-2所示，现在我们看到的只是一个空白的框架，每一部分的具体内容都需要我们进一步去实现。无论该学生信息管理系统的功能多么复杂，都不会超出目前框架中的布局设计。接下来，让我们逐一实现各个部分的功能。

12.2 使用 HTML 和 CSS 设置静态信息

用于显示标题和版权信息的文字都是静态的，我们可以直接调用HTML中设置好的内容。在Ext.Panel中添加静态信息的方式有如下2种。

- HTML参数：它让我们可以直接在JavaScript脚本中写上静态信息的内容。
- contentEl参数：它引用页面中某一个div的id，在显示Panel时将这个div中的内容显示在对应的布局域中。

相对而言，HTML参数更适合简单的内容。如果需要引入复杂格式的静态信息，还是应该使用contentEl参数。在这个示例中，我们就选择了contentEl参数为头部和底部制定静态信息部分的内容，实现方法如代码清单12-2所示。

代码清单12-2 在布局中设置静态内容

```
var viewport = new Ext.Viewport({
    layout: 'border',
    items: [{
        region: 'north',
        contentEl: 'head'
    }, {
        region: 'center',
        html: 'grid'
    }, {
        region: 'east',
        html: 'form'
    }, {
        region: 'south',
        contentEl: 'foot'
    }]
});
```

在上面的代码中，显示在上方的north部分引用的contentEl是'head'，它在页面中的内容如下所示。

```
<div id="head" style="font-weight:bold;font-size:200%;">学生信息管理</div>
```

显示在下方的south部分引用的contentEl是'foot'，它在页面中的内容如下所示。

```
    <div id="foot" style="text-align:right;"> - &copy; 2008 <a
href="http://www.family168.com" target="_blank">www.family168.com</a> - </div>
```

因为这两部分的标签内容都会在EXT进行页面布局时重新提取和设置，所以我们在开始时不用考虑把这两个div写到页面的什么位置，只要把它们写到页面里，EXT就会自动进行布局，把它们放到预设的位置。

设置过contentEl后，整个页面就变成了图12-3的样子。我们这里演示的效果比较简单，结合使用了HTML标签和CSS，为标题设置字体（加粗）和字号（变大），版权信息则是右对齐并设置了超链接。在实际项目中，可以把美工设计出来的页面标签直接复制到对应的div下，刷新页面后就会显示在对应的位置。

图12-3　上下部分加入静态信息的效果

12.3　对学生信息进行数据建模

接下来我们要实现对学生信息进行实际操作的功能。我们用Java编写后台脚本，用Hsqldb数据库作为保存数据的介质。首先要在数据库中创建学生信息数据表，如代码清单12-3所示。

代码清单12-3　创建学生信息表

```
create table student(
    id bigint, -- 主键
    code varchar(50), -- 学号
    name varchar(50), -- 姓名
    sex integer, -- 性别
    age integer, -- 年龄
    political varchar(50), -- 政治面貌
    origin varchar(50), -- 籍贯
    professional varchar(50) -- 所属系
);
alter table student
    add constraint pk_student primary key (id);
alter table student
    alter column id bigint generated by default as identity (start with 1,increment by 1);
```

这张student表中包含8个字段，分别对应学生的各种详细信息。最后两行不是标准的ANSI SQL语句，这是Hsqldb中的特有功能，表示把id字段设置成student表的唯一主键，并由数据库服务器控制自动增长。

我们在示例中将它作为嵌入式内存数据库，只要把hsqldb-1.8.0_7.jar放到WEB-INF/lib目录下就可以使用了，不必再去安装和配置外部服务器。Hsqldb数据库会在JDBC第一次连接时启动，从WEB-INF/classes目录下的test.scripts和test.properties两个文件中加载初始数据，其后的所有操作都会在内存中进行，唯一的缺陷是以这种方式运行的数据都保存在内存里。一旦服务器关闭就会导致数据丢失，下次重新启动数据库时将无法得知上次执行过何种操作，我们看到的数据依然是从test.scripts和test.properties中读取的初始化数据。有关Hsqldb数据库的详细信息可以去它的官方网站www.hsqldb.org查看。

在下面的讨论中，我们将尽量使用标准的ANSI SQL语句，保证可以在不同的数据库上正常

运行。对于不得不使用Hsqldb数据库的特定功能的SQL语句，我们会对它们单独进行讨论。

对应已经建立好的数据表结构，我们编写了一个与数据库表结构对应的JavaBean——Student.java，这个JavaBean中的字段与数据库表中的字段是一一对应的，如代码清单12-4所示。

代码清单12-4 学生信息领域模型

```
package com.family168.student;

public class Student {
    private long id;
    private String code;
    private String name;
    private int sex;
    private int age;
    private String political;
    private String origin;
    private String professional;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
}
// getter and setter
}
```

这是一个简单的JavaBean，它的每个属性对应了student表中的一个字段，我们将数据库中的字段映射为Java对象，在后面的操作里就可以为它们添加特定的逻辑，封装功能，简化操作。

直接操作Student.java的类叫做StudentDao.java，DAO是Data Access Object的简写，它主要负责Student.java和数据库之间的数据转换。比如，pagedQuery()会把从数据库中读取的数据放入Student.java对象中，并按照特定的形式返回。Insert()和update()方法则是将Student.java中的数据保存到数据库中。Remove()方法执行的是删除操作，它会根据指定的id从数据库中删除对应的对象。

这些对数据库的操作实际上都是大同小异的，每次先打开与数据库的连接，然后执行查询或更新操作，最后关闭连接并释放资源。

我们将获得数据库连接的部分代码封装在一个叫做DbUtils.java的工具类中，如代码清单12-5所示。

代码清单12-5 数据库工具类

```
package com.family168.student;

import java.sql.*;

public class DbUtils {
```

```
    static {
        try {
            Class.forName("org.hsqldb.jdbcDriver");
        } catch(Exception ex) {
            System.err.println(ex);
        }
    }

    static Connection getConn() throws Exception {
        return DriverManager.getConnection("jdbc:hsqldb:res:/test", "sa", "");
    }

    static void close(ResultSet rs, Statement state, Connection conn) {
        if (rs != null) {
            try {
                rs.close();
            } catch(SQLException ex) {
                ex.printStackTrace();
            }
            rs = null;
        }
        if (state != null) {
            try {
                state.close();
            } catch(SQLException ex) {
                ex.printStackTrace();
            }
            state = null;
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            conn = null;
        }
    }
}
```

这是一个工具类，不需要使用new关键字实例化就可以直接调用其中的方法，static{}静态初始化时加载Hsqldb的JDBC驱动，其后每次执行getConn()方法就可以得到一个与数据库的连接。Close()方法提供了一种关闭数据库连接并释放资源的简便方法，使用它可以一次性关闭ResultSet、Statement和Connection 3个对象，方法内自动检测对象是否为null，可以放心使用。

在StudentDao.java中统一通过DbUtils.java来获得数据库连接，以此实现对数据库的操作，比如remove()方法中就是先通过DbUtils的getConn()方法获得数据库的连接再进行删除操作的，如下面的代码所示。

```
public void remove(long id) throws Exception {
```

```
        String sql = "delete from student where id=?";

        Connection conn = DbUtils.getConn();
        PreparedStatement state = conn.prepareStatement(sql);
        state.setLong(1, id);

        state.executeUpdate();
        DbUtils.close(null, state, conn);
    }
```

先用DbUtils.getConn()获得数据库链接，然后准备SQL语句对应的Statement。设置指定的id后，调用Statement和executeUpdate()执行删除操作，最后不要忘记用DbUtils.close()关闭与数据库的连接释放资源。

另外两种更新数据库的方法与remove()相似，insert()和update()分别使用的是SQL中的insert和update语句，再将作为参数的Student.java对象设置到Statement中执行更新，实际代码请参考StudentDao.java中的内容，这里就不再赘述了。

pagedQuery()中的查询功能比较复杂，需要详细讨论一下，先看一下其中的代码部分，如代码清单12-6所示。

代码清单12-6 pagedQuery()

```
    public Page pagedQuery(int start, int limit, String sort, String dir) throws
    Exception {

        String sql = "select limit " + start + " " + limit + " * from student";
        if (sort != null && !sort.equals("") && dir != null && !dir.equals("")) {
            sql += " order by " + sort + " " + dir;
        }
        Connection conn = DbUtils.getConn();
        Statement state = conn.createStatement();

        ResultSet rs = state.executeQuery(sql);
        List result = new ArrayList();
        while (rs.next()) {
            Student student = new Student();
            student.setId(rs.getLong("id"));
            student.setCode(rs.getString("code"));
            student.setName(rs.getString("name"));
            student.setSex(rs.getInt("sex"));
            student.setAge(rs.getInt("age"));
            student.setPolitical(rs.getString("political"));
            student.setOrigin(rs.getString("origin"));
            student.setProfessional(rs.getString("professional"));
            result.add(student);
        }
        rs = state.executeQuery("select count(*) from student");
        int totalCount = 0;
        if (rs.next()) {
            totalCount = rs.getInt(1);
        }
```

```
        DbUtils.close(rs, state, conn);

        Page page = new Page(totalCount, result);
        return page;

    }
```

先看pagedQuery()方法的4个参数，分别是start、limit、sort和dir。start和limit用来进行分页查询，start表示从第几条数据进行查询，limit表示最多返回几条查询数据。sort和dir用来对查询的结果进行排序，sort表示对哪个字段进行排序，dir表示排序时使用升序还是降序。

为了实现分页和排序功能，pagedQuery()方法中首先要根据传递过来的参数生成对应功能的SQL语句。

```
String sql = "select limit " + start + " " + limit + " * from student";
```

上面是从student表中进行查询，并依据start和limit的内容进行分页。这里的"select limit " + start + " " + limit的不是标准的ANSI SQL语句，而是Hsqldb专门为分页查询提供的功能。很不幸的是，标准ANSI SQL中没有提供分页查询的功能，基本上每个主流数据库都提供了自己的分页查询方式，这些方式又完全不相同。如果你想把这个示例转换到其他数据库上，必须将这里的分页查询部分修改为对应数据库的SQL语句。

```
if (sort != null && !sort.equals("") && dir != null && !dir.equals("")) {
sql += " order by " + sort + " " + dir;
}
```

生成排序功能的SQL语句比较简单，只需要判断sort和dir是否为空。如果不为空，就可以直接附加到原来的SQL的后面，对获得的查询结果实现排序。

得到了需要的SQL语句之后，我们立刻进行查询，通过Statement和ResultSet，把每一条返回的记录都转换成Student.java对象，放入到ArrayList中。

接下来我们还需要获取数据库中的总记录数，对应的SQL语句为select count(*) from student，这是一条标准的ANSI SQL语句，不用做任何修改就可以在所有的主流数据库上运行。使用这条SQL语句，我们获得了数据库中保存的学生信息的总数。

现在我们可以关闭数据库连接，把学生信息总数totalCount和本页显示的学生信息列表result放入Page.java对象中并返回。

至此，我们完成了数据建模部分代码的编写，对应的源代码放在WEB-INF/src目录下，编译后的代码放在WEB-INF/classes目录下。对应的com.family168.student包下的内容供对应的JSP或其他JAVA类调用，通过这些类，我们可以访问数据库获得需要的查询结果，也可以通过这些类对数据库中的数据进行更新。

12.4　在页面中显示学生信息列表

后台的数据库和Java代码都已经准备妥当，现在来编写显示学生信息列表的Grid部分的代码，如代码清单12-7所示。

代码清单12-7 前台Prid部分的实现代码

```
var sexRenderer = function(value) {
    if (value == 1) {
        return '<span style="color:red;font-weight:bold;">男</span>';
    } else if (value == 2) {
        return '<span style="color:green;font-weight:bold;">女</span>';
    }
};

var StudentRecord = Ext.data.Record.create([
    {name: 'id', type: 'int'},
    {name: 'code', type: 'string'},
    {name: 'name', type: 'string'},
    {name: 'sex', type: 'int'},
    {name: 'age', type: 'int'},
    {name: 'political', type: 'string'},
    {name: 'origin', type: 'string'},
    {name: 'professional', type: 'string'}
]);

var store = new Ext.data.Store({
    proxy: new Ext.data.HttpProxy({url: './jsp/list.jsp'}),
    reader: new Ext.data.JsonReader({
        totalProperty: 'totalCount',
        root: 'result'
    },StudentRecord),
    remoteSort: true
});
store.load({params:{start:0,limit:15}});

var columns = new Ext.grid.ColumnModel([
    {header: '学号', dataIndex: 'code'},
    {header: '姓名', dataIndex: 'name'},
    {header: '性别', dataIndex: 'sex', renderer: sexRenderer},
    {header: '年龄', dataIndex: 'age'},
    {header: '政治面貌', dataIndex: 'political'},
    {header: '籍贯', dataIndex: 'origin'},
    {header: '所属系', dataIndex: 'professional'}
]);
columns.defaultSortable = true;

// grid start
var grid = new Ext.grid.GridPanel({
    title: '学生信息列表',
    region: 'center',
    loadMask: true,
    store: store,
    cm: columns,
    sm: new Ext.grid.RowSelectionModel({singleSelect:true}),
    viewConfig: {
        forceFit: true
```

```
        },
        bbar: new Ext.PagingToolbar({
            pageSize: 15,
            store: store,
            displayInfo: true
        })
    });
    // grid end
```

sexRenderer是一个工具函数，它用来在Grid中显示学生的性别。数据库中sex字段的类型为int，我们用1代表“男”，2代表“女”，sexRenderer中使用if语句判断当前行的sex值。在1的情况下显示红色粗体的“男”，在2的情况下显示绿色粗体的“女”。之后sexRenderer会作为ColumnModel的一部分设置到Grid中，负责显示“性别”这一列的内容。

StudentRecord部分使用Ext.data.Record的create()函数创建了一个类，就像之前在数据建模的过程中我们使用JavaBean对数据库进行映射一样，EXT中也对student的数据进行了封装。这样我们就可以在Grid的store中和form的reader中直接使用这个预定义的类型，避免了重复定义相同的数据类型。

接下来的Ext.data.Store就利用了上面的StudentRecord类型，处理从后台获得的信息。它使用Ext.data.HttpProxy从jsp/list.jsp中获得学生信息列表的信息，返回信息中的totalProperty和root两个参数分别指定了后台数据的记录总数和当前页面显示信息的队列，这些数据最终都会显示在Grid中。

创建好store之后，随即调用store.load({params:{start:0,limit:15}});进行分页查询，这里传递的两个参数start和limit，表示从第一条记录开始查询，最多获得15条记录。这部分与后台的jsp/list.jsp交互的操作我们留在后面再详细讨论。

下面的工作是创建Ext.grid.ColumnModel，将Grid中每列显示的数据与store中的数据相对应。建立好列模型后，再调用columns.defaultSortable = true;将所有列都设置成可排序的。排序功能也可以通过为每一列设置sortable:true来实现，但是像那样逐一设置会显得繁琐，不如统一设置方便。

万事俱备只欠东风，Grid需要的组件部分都准备好了，现在可以创建Grid来显示学生信息列表了。为了不使Grid显得寒酸，除了上面提到的store和columns之外，我们还为Grid设置了标题title。用loadMask:true开启读取数据时的提示功能，这会在每次store去后台读取数据时自动显示等待提示信息。sm: new Ext.grid.RowSelectionModel({singleSelect:true})限制用户每次只能选中一行，这是为了后面与表单form进行互操作所做的准备。viewConfig:{forceFit: true}自动调整每列的宽度，使整个表格更加饱满。最后还为bbar添加了分页工具条，可以用它进行Grid的分页跳转和数据刷新操作。

Region:'center'表示把这个grid放到BorderLayout的中间位置。接下来，我们调整原来Viewport中的配置，把之前放在中间的Panel替换为我们创建好的Grid，如代码清单12-8所示。

代码清单12-8　将Grid加入页面

```
var viewport = new Ext.Viewport({
```

```
    layout: 'border',
    items: [{
        region: 'north',
        contentEl: 'head'
    }, grid, {
        region: 'east',
        html: 'form'
    }, {
        region: 'south',
        contentEl: 'foot'
    }]
});
```

最终的显示效果如图12-4所示。

图12-4 加入Grid后的页面效果

因为我们没有为右侧的form部分设置宽度，所以Viewport自动为它计算了一个最小宽度，结果位于中间的Grid几乎布满了整个页面。从上图中我们可以看到，Grid中显示了每列对应的数据，“性别”这一列也按照sexRenderer中定义的那样显示得错落有致。现在你可以单击Grid下部的分页工具条上的按钮查看分页查询的效果，也可以单击某一列的首部，查看按列排序的功能。

看过了页面的效果，我们再回到后台看看为前台提供数据的list.jsp中的内容，如代码清单12-9所示。

代码清单12-9 list.jsp

```
<%@ page contentType="application/json;charset=utf-8" import="com.family168.student.
*" %><%
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");
    int start = 0;
    try {
        start = Integer.parseInt(request.getParameter("start"));
    } catch(Exception ex) {
```

```
            System.err.println(ex);
        }
        int limit = 15;
        try {
            limit = Integer.parseInt(request.getParameter("limit"));
        } catch(Exception ex) {
            System.err.println(ex);
        }
        String sort = request.getParameter("sort");
        String dir = request.getParameter("dir");

        StudentDao dao = StudentDao.getInstance();
        Page pager = dao.pagedQuery(start, limit, sort, dir);

        out.print(pager.toString());
    %>
```

List.jsp中的代码可以分成三部分，如下所示。

(1) 第一部分，设置JSP使用的contentType和encoding，因为Ajax在访问后台时，默认使用UTF-8作为请求和响应时传递的数据的默认编码，而在Tomcat服务器中，使用的默认编码是ISO-8859-1。这种编码的差异会在传递中文字符时出现乱码，所以我们首先需要把请求和响应的编码都统一设为UTF-8。

(2) 第二部分，处理前台传递的参数，包括start、limit、sort、dir等4个参数，这4个参数在12.3节中已经讨论过。start和limit用来进行分页查询，start表示从第几条数据进行查询，limit表示最多返回几条查询数据。sort和dir用来对查询的结果进行排序，sort表示对哪个字段排序，dir表示排序时使用升序还是降序。

因为使用HTTP协议只能传递字符类型的参数，所以在list.jsp中，我们要对这几个参数进行类型转换。注意，当有可能出现转换失败的情况时，需要为对应的参数设置默认值，我们这里默认设置start为0，limit为15。

(3) 第三部分，将获得的4个参数传递给StudentDao，获得分页结果对象，最后调用Page的toString()方法，将它生成的内容返还给前台处理。

我们将StudentDao设计成一个单例模式（Singleton），这样可以在当前系统中保证只创建一个实例，这个实例被其他类所共享，避免了重复创建对象造成的资源浪费。

关于Page的toString()方法，是为了将分页结果转换成JSON格式，这个功能是通过Student.java和Page.java两个类中的toString()方法来实现的。

Student.java的代码如下所示：

```
public String toString() {
    return "{id:" + id +
        ",code:'" + code +
        "',name:'" + name +
        "',sex:" + sex +
        ",age:" + age +
        ",political:'" + political +
        "',origin:'" + origin +
```

```
            "',professional:'" + professional +
            "'}";
    }
```

Page.java的代码如下所示：

```
public String toString() {
    return "{totalCount:" + totalCount + ",result:" + result + "}";
}
```

通过这两个方法，page.toString()会返回一个符合JSON格式的字符串，并使用之前设置好的UTF-8编码格式发送给前台。前台EXT接收到的内容如代码清单12-10所示。

代码清单12-10 前台EXT获得的JSON数据

```
{
totalCount:16,
result:[{
id:1,
code:'2002015',
name:'张光和',
sex:1,
age:21,
political:'团员',
origin:'湖北省',
professional:'物流工程学院'
}, {
id:2,
code:'2002002',
name:'张值强',
sex:1,
age:22,
political:'党员',
origin:'河南省',
professional:'动力工程系'
}]
}
```

totalCount表示数据库中现有学生信息的总数，result表示当前页显示的信息列表，前台获得这些信息之后，就可以把这些数据显示到Grid中。

12.5 添加表单编辑学生信息

配置好Grid之后，我们再添加一个表单来编辑学生信息，如代码清单12-11所示。

代码清单12-11 编辑学生信息的表单

```
// form start
var form = new Ext.form.FormPanel({
    title: '编辑学生信息',
    region: 'east',
    frame: true,
```

```
width: 300,
autoHeight: true,
labelAlign: 'right',
labelWidth: 60,
defaultType: 'textfield',
defaults: {
    width: 200,
    allowBlank: false
},
items: [{
    xtype: 'hidden',
    name: 'id'
},{
    fieldLabel: '学号',
    name: 'code'
},{
    fieldLabel: '姓名',
    name: 'name'
},{
    fieldLabel: '年龄',
    name: 'age',
    xtype: 'numberfield',
    allowNegative: false
},{
    fieldLabel: '性别',
    name: 'sexText',
    hiddenName: 'sex',
    xtype: 'combo',
    store: new Ext.data.SimpleStore({
        fields: ['value','text'],
        data: [['1','男'],['2','女']]
    }),
    emptyText: '请选择',
    mode: 'local',
    triggerAction: 'all',
    valueField: 'value',
    displayField: 'text',
    readOnly: true
},{
    fieldLabel: '政治面貌',
    name: 'political',
    xtype: 'combo',
    store: new Ext.data.SimpleStore({
        fields: ['text'],
        data: [['群众'],['党员'],['团员']]
    }),
    emptyText: '请选择',
    mode: 'local',
    triggerAction: 'all',
    valueField: 'text',
    displayField: 'text',
    readOnly: true
```

```
        },{
            fieldLabel: '籍贯',
            name: 'origin'
        },{
            fieldLabel: '所属系',
            name: 'professional'
        }],
        buttons: [{
            text: '添加'
        },{
            text: '清空'
        },{
            text: '删除'
        }]
    });
    // form end
```

我们把这个表单的宽度设置为300，让它里面的输入组件对应的文字标签都右对齐，其中的defaults参数很有用，在它里面设置的参数会自动赋予form内部的输入组件，这样内部组件的相同配置只需要配置一次就可以了。我们使用的配置是{ width: 200, allowBlank: false}，每个组件的宽度都设置为80，而且不能提交空值。因为我们使用的输入组件大多是textfield，所以把defaultType设置为'textfield'可以不必为每个items设置xtype。

在对Ext.form.FormPanel进行了这些设置之后，我们可以把学生信息对应的输入控件放到items中了。其中“学号”，“姓名”，“籍贯”，“所属系”都是Textfield类型，因此只需要配置name和fieldLabel。

需要使用xtype:'hidden'将id字段配置为隐藏域。虽然在页面上看不到它，但是也可以通过后面的loadRecord()方法设置id字段的数据。当进行表单提交时，它里边的数据也会一起发送到后台，它的作用和<input type="hidden" name="id">是一致的。

“年龄”输入框限制用户只能输入数字，使用了Ext.form.NumberField数字输入组件，对应使用了xtype:'numberfield'。因为人的年龄不可能是负数，所以我们还将allowNegative设置为false，不允许用户输入负数。

虽然“性别”和“政治面貌”使用的都是Ext.form.ComboBox，但还是有一些不同。

虽然“性别”这个ComboBox选择的是“男”和“女”，但是实际上传递到后台的是对应的1或2两个整数。因此，在配置ComboBox时需要将displayField和valueField分开使用，还要配置上hiddenName，否则发送到后台的依然是显示在页面上的“男”和“女”，而不是与之对应的1或2两个整数。

“政治面貌”这个ComboBox就简单了许多，选择的文字和传递给后台的数据是一致的，只需displayField和valueField的配置相同，不需要额外配置hiddenName参数。

经过了这一系列的配置后，我们得到了一个可以用来编辑学生信息的表单form，效果如图12-5所示。

到此为止，学生信息管理系统的整个界面已经制作完成，但是还不能利用表单对学生信息执行添加、修改和删除等操作。

图12-5　加入表单后的界面效果

下面我们要为表单的按钮设置监听事件，让我们在单击相应按钮时执行对应的操作。

12.6　为表单添加提交事件

在上面的表单中，我们放置了三个按钮，分别是“添加”、“清空”和“删除”。首先我们为“添加”按钮添加事件，为它的handler参数指定一个处理函数，如代码清单12-12所示。

代码清单12-12　表单按钮的单击事件

```
{
            text: '添加',
            handler: function() {
                if (!form.getForm().isValid()) {
                    return;
                }
                if (form.getForm().findField("id").getValue() == "") {
                    // 添加
                    form.getForm().submit({
                        url: './jsp/save.jsp',
                        success: function(f, action) {
                            if (action.result.success) {
                                Ext.Msg.alert('消息', action.result.msg,function(){
                                    grid.getStore().reload();
                                    form.getForm().reset();
                                    form.buttons[0].setText('添加');
                                });
                            }
                        },
                        failure: function() {
                            Ext.Msg.alert('错误', "添加失败");
                        }
```

```
                    });
                } else {
                    // 修改
                    form.getForm().submit({
                        url: './jsp/save.jsp',
                        success: function(f, action) {
                            if (action.result.success) {
                                Ext.Msg.alert('消息',action.result.msg,function(){
                                    grid.getStore().reload();
                                    form.getForm().reset();
                                    form.buttons[0].setText('添加');
                                });
                            }
                        },
                        failure: function() {
                            Ext.Msg.alert('错误', "修改失败");
                        }
                    });
                }
            }
        }
```

在handler处理函数中，首先调用form.getForm().isValid()进行数据校验。如果返回false，说明表单中某些输入组件中的数据还无法通过校验，不应该提交这些错误格式的数据，这时我们应该直接跳出函数，中止提交操作。

如果表单顺利通过数据校验检测，我们便进入下一步，准备向后台提交数据。还记得我们刚才讲过的id对应的隐藏域吗？这里就是通过它的数值来判断本次提交是添加操作还是修改操作。如果form.getForm().findField("id").getValue()为空字符串，就是在进行数据添加；如果它不为空字符串，就是在对一个已有的数据进行修改。

虽然我们对添加和修改操作进行了区分，但是在实际提交代码时并没有太大区别，只是在提交错误时分别提示“添加失败”和“修改失败”而已。让我们通过代码清单12-13来看看这些数据是如何提交给后台的。

代码清单12-13 将数据提交给后台

```
// 添加
form.getForm().submit({
url: './jsp/save.jsp',
success: function(f, action) {
    if (action.result.success) {
        Ext.Msg.alert('消息', action.result.msg, function() {
            grid.getStore().reload();
            form.getForm().reset();
            form.buttons[0].setText('添加');
        });
    }
},
failure: function() {
```

```
Ext.Msg.alert('错误', "添加失败");
}
});
```

这里的form表示我们前面创建的Ext.form.FormPanel，它的getForm()函数返回FormPanel内部对应的Ext.form.BasicForm。现在我们调用BasicForm的submit()函数，将内部items中输入组件的值提交给后台的jsp/save.jsp。

如果后台没有出现异常，而且返回的JSON信息中包含{success:true}，那么就会执行success参数对应的处理函数。在success处理函数中，我们创建一个Ext.Msg.alert()显示响应的JSON信息中的msg部分的内容。在用户关闭alert提示框之后，调用grid.getStore().reload()刷新Grid中的数据，同时调用form.getForm().reset()清空上次提交的数据。

如果后台出现400或500错误，就会触发failure参数对应的处理函数。这里只是弹出一个alert提示框告诉用户“添加失败”，等待用户对刚才提交失败的信息进行修改或做其他处理。

在添加和修改信息时，都是提交给后台的jsp/save.jsp进行处理。save.jsp这个文件同时负责添加和修改两个操作，它里面的内容如代码清单12-14所示。

代码清单12-14　save.jsp

```
<%@ page contentType="application/json;charset=utf-8" import="com.family168.student.
*" %><%
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");

    String id = request.getParameter("id");
    String code = request.getParameter("code");
    String name = request.getParameter("name");
    String sex = request.getParameter("sex");
    String age = request.getParameter("age");
    String political = request.getParameter("political");
    String origin = request.getParameter("origin");
    String professional = request.getParameter("professional");

    Student student = new Student();
    student.setCode(code);
    student.setName(name);
    student.setSex(Integer.parseInt(sex));
    student.setAge(Integer.parseInt(age));
    student.setPolitical(political);
    student.setOrigin(origin);
    student.setProfessional(professional);

    StudentDao dao = StudentDao.getInstance();
    if (id == null || id.equals("")) {
        dao.insert(student);
    } else {
        student.setId(Long.parseLong(id));
        dao.update(student);
```

```
        }
        out.print("{success:true,msg:'保存成功'}");
    %>
```

与之前讨论过的list.jsp类似，它里面的处理过程也大致分为 3 步。

(1) 设置JSP使用的contentType和encoding，把请求和响应的编码都统一为UTF-8。

(2) 处理前台传递的参数，通过request.getParameter()方法从请求中获得刚刚提交过来的学生信息，创建一个Student.java对象，将对应的学生信息添加到对象中。这个过程中要注意对age和sex两个参数进行数据类型转换，因为从HTTP请求中只能获得字符串，而它们都需要在其后转换为整数类型。

为了在后面的操作中区分添加和修改操作，我们没有对参数id进行处理。

(3) 现在我们已经获得了提交数据，并把这些数据都放到了对应的Student.java对象中。现在我们可以判断参数id的值，以此来判断后面要执行的是添加操作还是修改操作。

与前台EXT提交数据代码相似，如果id == null || id.equals("")，就表明应该执行添加操作，执行StudentDao的insert()方法。否则，应该执行修改操作，执行StudentDao的update()方法。

最后，只要未出现异常，我们就认为添加或修改操作成功了，直接向response中写入提示成功信息的JSON字符串。

```
out.print("{success:true,msg:'保存成功'}");
```

12.7 清空表单信息

再来看看“清空”按钮中handler对应的处理函数。

```
{
    text: '清空',
    handler: function() {
       form.getForm().reset();
       form.buttons[0].setText('添加');
    }
}
```

它的作用是调用form.getForm().reset();清空form中的所有数据，然后把form.buttons[0]也就是form的第一个按钮的文字修改为“添加”。

Reset将form恢复到初始状态。如果刚才在表单中添加了一些不必要的数据，又不想逐一去删除它们，那么只需要单击这个按钮就能清除所有输入框中的数据了。

Reset还有一个功能是清空隐藏域id中的数据，因为id在页面上是看不到的，所以无法手工删除它的数据，此时就只好求助于“清空”按钮了。如果想从“修改”状态转换到“添加”状态，也需要它的帮助。

Form.buttons[0]从表单中取出排在第一位的按钮。我们一共设置了3个按钮，第一个是“添加”按钮，因为在修改学生信息时会把它上面的文字改为“修改”，所以“清空”按钮也要在执行reset()之后把它的文字改为“添加”才行。

12.8　删除指定的学生信息

“删除”按钮是表单上的第三个按钮，它的作用是删除当前显示的学生信息。既然是从数据库中删除已有的学生信息，那它就只能在“修改”信息的状态下起作用。因为在没有获得学生信息的id之前，无法执行删除操作。

“删除”按钮对应的handler处理函数如代码清单12-15所示。

代码清单12-15　删除学生信息的处理函数

```
{
text: '删除',
    handler: function() {
        var id = form.getForm().findField('id').getValue();
        if (id == '') {
            Ext.Msg.alert('提示', '请选择需要删除的信息。');
        } else {
            Ext.Ajax.request({
                url: './jsp/remove.jsp',
                success: function(response) {
                    var json = Ext.decode(response.responseText);
                    if (json.success) {
                        Ext.Msg.alert('消息', json.msg, function() {
                            grid.getStore().reload();
                            form.getForm().reset();
                            form.buttons[0].setText('添加');
                        });
                    }
                },
                failure: function() {
                    Ext.Msg.alert('错误', "删除失败");
                },
                params: "id=" + id
            });
        }
    }
}
```

处理函数首先要判断form.getForm().findField('id').getValue()的值是否为空，如果id为空，表示还没有选择需要删除的信息，这时会弹出提示信息，同时中止删除操作。

如果id不为空，表示已经选择了需要删除的信息，这样可以把对应信息的id发送到后台执行删除操作。与“添加”按钮不同的是，向后台发送要删除的信息的id时要通过Ext.Ajax来实现，因为删除操作并不需要将表单中所有输入组件的数据都发送到后台。

使用Ext.Ajax.request()函数发送信息时，url参数表示发送给后台的jsp/remove.jsp进行处理，params表示将学生信息的id作为参数传递给后台。Success对应的处理方法会在响应成功时执行，与form不同的是，这里不再需要在响应的JSON中设置{success:true}，但我们需要使用Ext.decode()函数将响应返回的response.responseText字符串手工转换为JSON对

象，然后用Ext.Msg.alert()显示响应中的提示信息。响应成功后，刷新Grid数据，清空表单内容的操作与之前的添加和修改操作相同。

因为删除操作只向后台的remove.jsp发送需要删除的信息的id，所以remove.jsp中的处理代码也比较简单，如代码清单12-16所示。

代码清单12-16 remove.jsp

```
<%@ page contentType="application/json;charset=utf-8" import="com.family168.student.
*" %><%
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");

    String id = request.getParameter("id");
    StudentDao dao = StudentDao.getInstance();
    dao.remove(Long.parseLong(id));

    out.print("{success:true,msg:'删除成功'}");
%>
```

这里我们还是先要设置请求和响应的编码，然后从请求中获得参数id的值，转换成long长整型，调用StudentDao的remove()方法执行删除操作。如果处理的过程没有出现异常，就说明删除操作成功，最后向响应中写入删除成功的提示信息。

这样我们就完成了删除指定学生信息的操作。

12.9 在 Grid 和 Form 之间进行数据交互

我们在上面已经实现了在Grid中显示学生信息列表，也实现了使用Form对学生信息执行添加、修改和删除等操作。但是，Grid和Form之间的数据还无法交互使用。

一个尚未解决的问题是，如何在Form中显示我们需要修改的学生信息呢？这个问题也适用于删除操作，如果我们不提供选择某一条学生记录的方法，那么修改和删除操作都无法进行。

在这个示例中，我们希望在单击左侧的Grid时同步更新右边Form中的数据。如果用户单击Grid中的某一行，就会把这行对应的学生信息放到Form中显示，于是我们就能对这条信息进行修改和删除操作了。为此，我们要给Grid添加一个事件监听函数，专门处理鼠标点击事件，如下面的代码所示。

```
// 单击修改信息开始
grid.on('rowclick', function(grid, rowIndex, event) {
    var record = grid.getStore().getAt(rowIndex);
    form.getForm().loadRecord(record);
    form.buttons[0].setText('修改');
});
// 单击修改信息结束
```

这里监听的事件名为rowclick，它对应Ext.grid.RowSelectionModel的监听事件，每当用户选中Grid中的一行时，就会触发该事件。事件被触发的同时还会执行我们设置的监听函数。

监听函数预设了3个参数：第一个参数grid表示哪个Grid被点击了；第二个参数rowIndex表

示选中了哪一行；event是EXT内部通用的事件对象，我们在这里没有用到。

在监听函数执行时，首先通过grid.getStore().getAt(rowIndex);获得被选中的这一行对应的record。这个record是保存在store中的数据，Grid上没有显示出来的id也包含在其中。对应的所有学生信息都可以从这个record中获得，但并不需要从record中把学生信息逐一取出来，然后再逐一放到Form中去。Form提供的loadRecord()函数可以一次性将record中的数据赋予Form中的输入组件，只要保证输入组件的name或hiddenName与record中定义的属性一致即可。

在使用loadRecord()将Grid中选择的数据复制到Form中以后，我们再调用form.buttons[0].setText('修改')；将Form中的第一个按钮的文字设置为“修改”。这样用户就知道现在提交Form执行的是对某一条学生信息进行修改的操作。如果要继续添加新的学生信息，可以单击“清空”按钮，它会将刚刚从Grid中复制的信息都清除掉，包括id隐藏域中的数据，还会把第一个按钮上的“修改”设置为“添加”。再次输入数据并单击“提交”按钮，这时执行的就是“添加”操作了。

到此为止，我们用前面学过的知识实现了一个完整的学生信息管理系统。其中涉及BorderLayout的布局应用、Grid的分页显示和数据排序、Form的提交和清空、利用Ajax与后台进行数据交互、通过事件监听实现Grid与Form之间的数据交互等知识。

这个示例虽然简单，但基本上包含了所有的常见操作，可以供大家练习时参考。

12.10　本章小结

本章详细演示了如何实现一个学生信息管理系统。其中简要介绍了如何使用Java访问数据库，以及一些SQL语句的使用方法。重点介绍了如何使EXT与后台进行交互，以实现对数据库信息进行分页显示和后台排序。此外，还讲解了如何在EXT中对数据库信息执行添加、删除、修改和查找等基本操作，并提出了一种Grid与Form之间进行数据交互的方式。