

第一章 起步（1）

在这章里，我们将介绍 ExtJS 的基本功能。如果你对 web 开发熟悉的话，您会惊诧于 ExtJS 框架的优雅！不同其他的 JavaScript 库，ExtJS 为您的开发夯实了基础，只需几行代码，您就可以制作出丰富的用户界面。

本章包括：

1. ExtJS 的功能和您将会喜欢上它的原因；
2. 如何获得 Ext 并在 web 应用中采用它；
3. 采用”适配器(adapters)”使得 Ext 和其他的 JavaScript 库共存；
4. 充分利用 AJAX 技术；
5. 在您的语言中展示 ExtJS 对象；

关于 Ext：

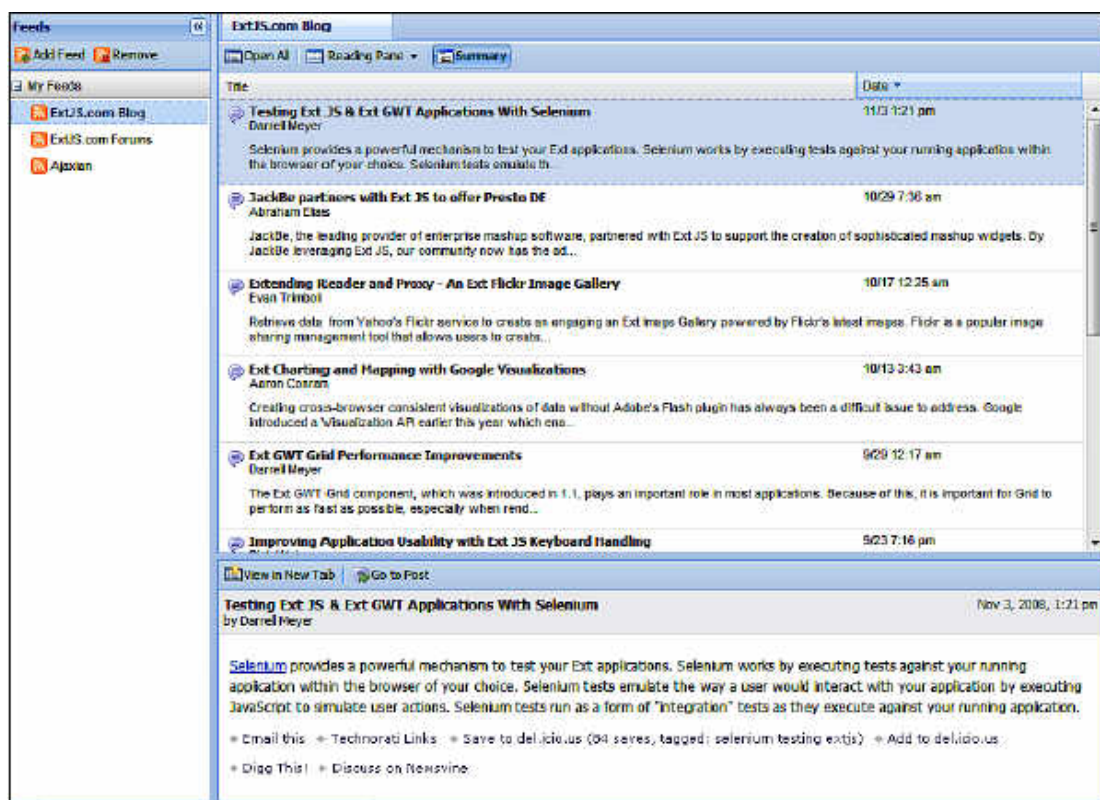
我们采用最新的 Ext 2.x 版本，1.x 版本到 2.x 版本的是一个重组的过程，包括添加新的组件、重命名组件用来重新组织结构等等。这使得 1.x 和 2.x 兼容性很低。3.x 版本则不然，他对 2.x 有很强的兼容性，可以很好的联合本书中所设计的内容。Ext 的开发组决定在日后的版本发布中都做到向前兼容。

Ext 库是对雅虎 YUI 的一个拓展，提供了它所不支持的特性：良好的 API，真实的控件。虽然 YUI 致力于用户界面，但是它却没有提供许多有用的功能。

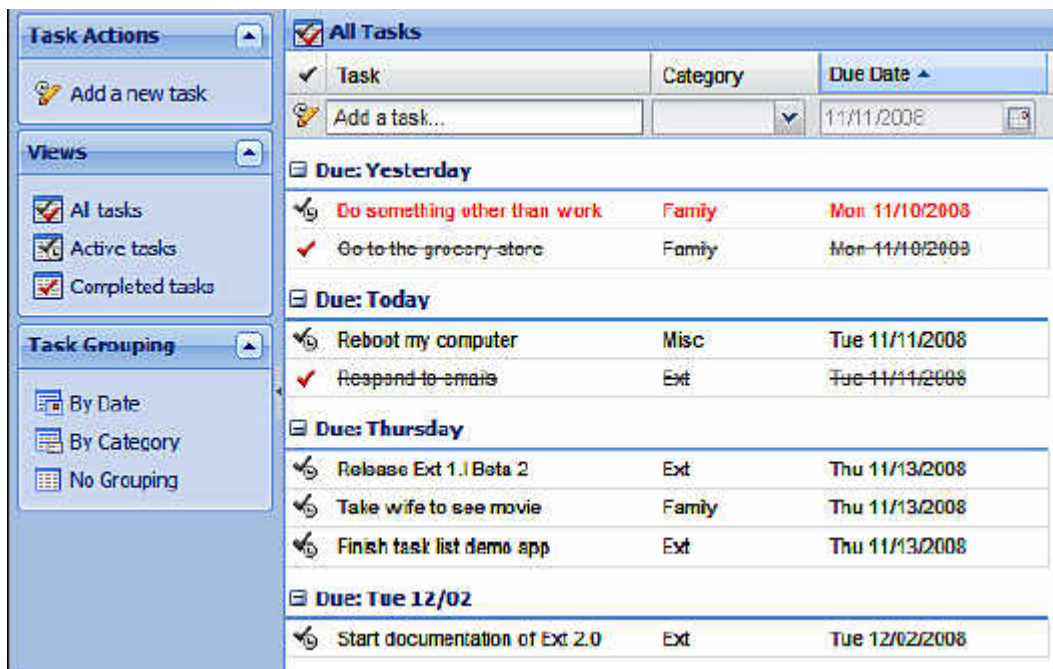
Ext 的产生源自于开发者、开源贡献者们将 YUI 扩展成一个强大的客户端应用程序库的努力。

Ext 提供了一个简单丰富的用户界面，如同桌面程序一般。这使得开发者能够把精力更多的转移到实现应用的功能上。Ext 官网上的示例会让你知道它是如何的不可思议：<http://www.extjs.com/deploy/dev/examples/>。

其中最引人注目的一个例子就是 Feed Viewer，它展示了 Ext.However 提供的多种特性，对于学习来说它可能太复杂了，所以现在只需你感受它带给你的精彩。



另外一个精彩的例子就是 Simple Task 任务跟踪程序，它加载了 Google Gears 的数据库。



Ext: 不仅仅是另一个 JS 库：

Ext 不仅仅是另一个 JS 库，实际上它可以通过适配器（adapter）和其它 JS 库一起工作。我们将在本章的稍后来介绍适配器。

通常来说，我们使用 Ext 的目的是满足高层次的用户交互——要比我们传统概念上的站点交互性更强。一个采用了工作流和任务管理的网站就是一个很好的示例，否则 Ext 只能带给您的上司惊奇的喘息。

Ext 让通过如下的方式来让 web 应用的开发变的十分简单：

- 提供简单的，跨浏览器的控件，如：窗口、表格、表单。这些组件都是能够适应市场上的主流浏览器的。我们不需要做任何改动。
- 用户是通过 **EventManager** 来和浏览器做交互的，相应的事件有：用户的键盘输入，鼠标击打，浏览器监听（窗口改变大小，改变字体）等等；
- 在和用户交互时不需要刷新页面，一切在后台进行。它允许你从服务器通过 **AJAX** 来获取或者提交数据并且在第一时间执行你的反馈。

跨浏览器 DOM：

我确定不需要在解释浏览器兼容带来的问题了。一个有着自定义样式的<div>在不同的浏览器的显示是不同的。但当我们使用 Ext 的控件时，Ext 库会很好地控制这种浏览器的兼容性，所以在不同浏览器中控件的显示是几乎相同的。这些浏览器包括：

Internet Explorer 6+

Firefox 1.5+ (PC, Mac)

Safari 2+

Opera 9+ (PC, Mac)

事件驱动接口：

事件是用来描述某种动作的发生。一个事件可以是用户的动作，例如单击某个元素或者对 **AJAX** 请求的相应。当用户对一个按钮进行操作的时候，会产生很多而不是一个事件，包括：鼠标指向按钮，点击按钮和离开按钮。我们可以添加监听器（**listener**）来监听这些事件，并调用相应代码采取相应。

对事件的监听并不完全依靠用户界面，系统事件随时随刻发生，当你发起 **AJAX** 请求，和 **AJAX** 状态相关的事件也就产生了，它们是：开始，完成和失败等等。

Ext 和 AJAX：

AJAX 用来描述同服务器的异步交互，你可以在同服务器交互的同时来进行其他的任务。一个用户可以在加载一个表格数据的同时填写自己的表单——它们并行不悖，不用等待页面的刷新。

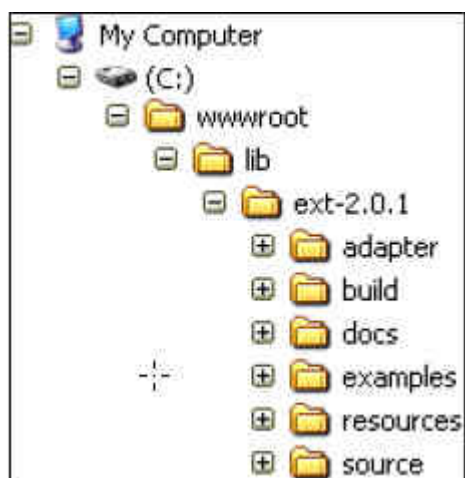
获得 Ext:

所有你所需的东西都可以从Ext官网上获得: <http://www.extjs.com/download>。下载包含大量示例和API的Ext SDK, 它最重要的功能是带给你使Ext正确运行的资源。

如何安置 Ext:

当你获得 SDK 文件, 把它解压到你的硬盘上, 最好放在你自己的文件夹中。我访问文件的方式是按照 linux 规则来的, 为了很好地运行本书示例, 我把所有文件解压到命名为 lib 的文件夹中。

当一切处理好后, 你的目录结构应该如下所示:



为了使 Ext 容易升级到最新版本, 我把解压后的文件夹 `ext-2.0.1` 重新命名为 `extjs`。

SDK 里有一个包含所以 Ext 的 JS 代码的文件, 名字为 `ext-all`。它还有一个可供调试代码的副本, 由于进行了代码的格式化和反混淆等操作, 该文件可以您准确定位到错误的行数和具体位置。开发时您可以使用 `ext-all-debug`, 发布时转换为 `ext-all`, 一切就是这么的简单以及过度自然。

在 SDK 中包含着依赖说明、文档、示例等内容。`adapter` 和 `resources` 是 Ext 工作所必须的, 其他资源则是为开发准备的。

- **adapter:** 包含能够使其他库和 Ext 共用的文件;
- **build:** 可以自定义 `ext-all.js` 的文件;
- **docs:** 文档中心 (必须在服务器上查看);
- **examples:** 大量精彩深刻的示例;
- **resources:** Ext 运行的各种资源 (如 CSS 和图片等);
- **source:** Ext 完全而详细的源代码。

在创建站点主目录后，您需要上传 `adapter` 和 `resources` 文件夹。

在页面中引入：

在我们使用 Ext 之前，需要引入 Ext 库文件。在 HEAD 标签加入 SDK 中提供的几个文件，如下所示：

```
<html>

  <head>

    <title>Getting Started Example</title>

    <link rel="stylesheet" type="text/css"

      href="lib/extjs/resources/css/ext-all.css" />

    <script src="lib/extjs/adapter/ext/ext-base.js"></script>

    <script src="lib/extjs/ext-all-debug.js"></script>

  </head>

  <body>

    <!-- Nothing in the body -->

  </body>

</html>
```

到 Ext 的路径一定要保证正确，和你的 HTML 文件的相对路径关系一定要调整好。这些文件需要按照如下的顺序引入：

- `ext-all.css`：Ext 依赖的全部 CSS；
- 其他的 js 库（参阅本章“适配器”一节）；
- `ext-base.js`：Ext 的“adapter”——我们将在后面学习；
- `ext-all.js` 或者 `ext-all-debug.js`：主要的 Ext 库文件；
- 主题文件（CSS）在此引入，或者在引入玩 Ext 的 CSS 文件后任何位置引入。

第一章 起步（2）

这些文件是用来干什么的：

我们需要添加一下三个文件：

- **ext-all.css**：这是一个样式表文件，用来控制组件的显示。该文件不需要进行修改，任何改动将影响日后的升级。如果真的想改动样式的话，可以在引入该样式文件后再做覆盖。
- **ext-base.js**：这个文件提供了 Ext 的核心功能。如果 Ext 是一部车，它就是车的引擎。这个文件我们可以修改以用来采用其他的库，如 jQuery，让它和 Ext 共存。
- **ext-all-debug.js/ext-all.js**：所有的组件都蕴含在这个文件里面。前者用来调试开发，后者在发布的时候采用。

当这些文件都准备好后，我们就可以真正开始享受 Ext 带给我们的快乐。

N：如果你在同时使用后台语言（如 PHP、ASP.NET 等），你可以选择动态的加载这些行。对于本书中的大部分示例来讲，我们假定其运行在静态 HTML 页面中。

使用 Ext 库：

现在我们在页面中加入了 Ext 库，我们可以在页面中添加 Ext 代码了。在第一个例子中，我们使用 Ext 来显示消息对话框。可能满足不了您的胃口，但是这只是个开始。

动作的时间：

我们可以通过在文件的头部添加 script 段来加入 Ext 代码，当然得添加到 Ext 文件引入之后。我们的例子会弹出一个 Ext 风格的警告对话框：

```
<html>

  <head>

    <title>Getting Started Example</title>

    <link rel="stylesheet" type="text/css"

      href="lib/extjs/resources/css/ext-all.css" />

    <script src="lib/extjs/adaptor/ext/ext-base.js"></script>

    <script src="lib/extjs/ext-all-debug.js"></script>

    <script>
```

```

Ext.onReady(function(){

    Ext.Msg.alert('Hello', 'World');

});

</script>

</head>

<body>

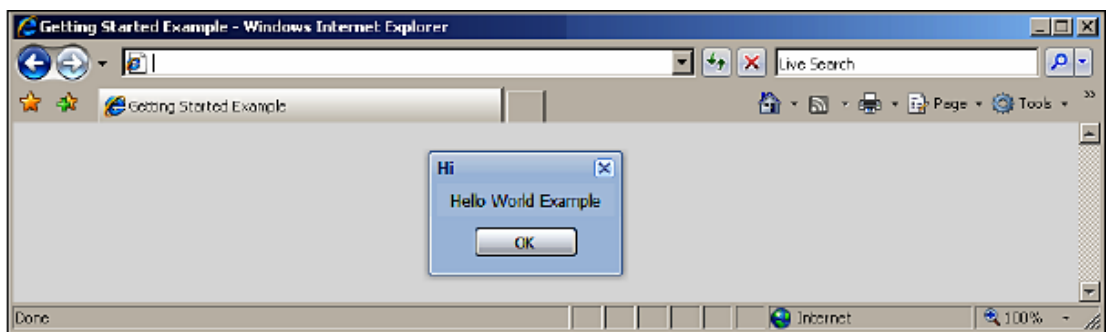
    <!-- Nothing in the body -->

</body>

</html>

```

我们现在先不详细解释脚本。首先，我们要确定 Ext 库被成功的引用。当我们打开页面，我们应该能看到这样的警告消息：



如同一个真的对话框一样，你可以任意拖拽它，但只局限有浏览器中，因为它并不是一个真正的对话框，只是如同罢了。它是有很多<div>堆砌而成，你可以看到 **Close** 和 **Ok** 按钮在鼠标移近时高亮显示——一切只不过是一句话的事情。Ext 为您搞定了一切，我们来看看如何能做的更多。

N: 你可能已经发现我们的文档的 **body** 中没有任何元素。Ext 实现其功能不需要事先存在的标记，它只是按照需要来产生一切内容。

这个示例：

我们来研究下代码。Ext 组件需要以 “Ext” 开头并且经常要包含 onReady 函数，这个我们将在下章做详细介绍。

```

Ext.onReady(function(){

    Ext.Msg.alert('Hello', 'World');

});

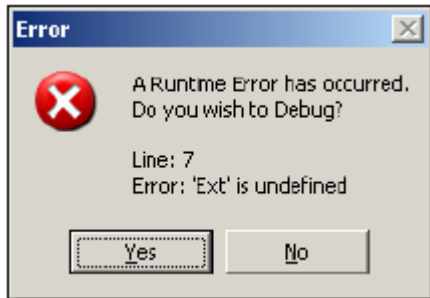
```

Ext 包含很多易懂的接口，大部分只是一句话，当准备好 Ext 后，它可以展示一个警告窗口，让 Hello 作为标题，让 World 作为主体。

我们的消息窗口源自于 Ext.Msg，它是所有消息窗体的前缀，是“MessageBox”的简称。alert 部分告诉我们消息窗口的类型。

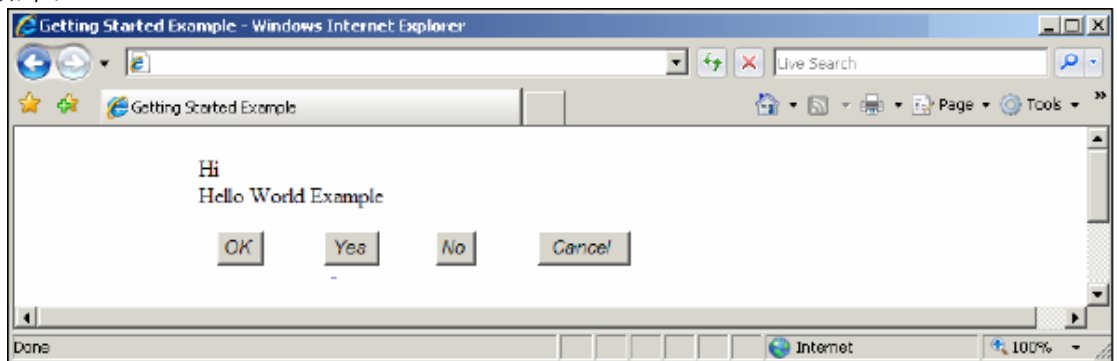
不工作了？

当库未准备好时，我们会接收到一个'Ext' is undefined 的错误。



这个消息代表 Ext 并没有被加载。可能是一个或者多个文件的引用路径不对，复核一下路径让他们指定到已存在的文件。如果一切都正常的话，你应该可以看到 adapter 文件夹和 ext-all.js 以及 ext-all-bug 在 lib/extjs 文件夹下。

另外一个常见的问题是，CSS文件的丢失或者不正常的引用。这样的话，显示会不正常，如下：



如果发生这样的状况，请检查你是否从 SDK 中获得了 CSS 文件，以及路径是否正确。resources 文件夹需要在 lib/extjs 文件夹下。

适配器（Adapters）：

Ext 开发之初，需要 YUI 库来做“幕后”的工作。随后，Ext 采用了 2 个新的框架——jQuery 和 scriptaculous（Protaculous）。

这意味着当你已经采用了其他的库或者其他的库可以刚好地满足你的需求是，我们可以使用适配器来连接这一切。Ext 还可以保证它的功能，组件还如之前一样，不论你选择何种适配器。

Ext 也有它自己的适配器，如果你对其他的库或者框架没有偏爱的话，就用 Ext 的内建适配器吧。

使用适配器：

为了使用适配器，你需要先引入扩展的库，再加入 Ext SDK 中相关的适配器文件。我们的示例用的是 Ext 适配器。如果使用其他库，只要把 Ext 默认的适配器引入脚本行替换为新的库，如下所示：

```
<script src="lib/extjs/adapters/ext/ext-base.js"></script>
```

对于 jQuery，引入如下文件：

```
<script src="lib/jquery.js"></script>
```

```
<script src="lib/jquery-plugins.js"></script>
```

```
<script src="lib/extjs/adapters/jquery/ext-jquery-adapter.js">
```

```
</script>
```

对于 YUI，把如下文件加入 head：

```
<script src="lib/utilities.js"></script>
```

```
<script src="lib/extjs/adapters/yui/ext-yui-adapter.js"></script>
```

对于 “Prototype + Scriptaculous”：

```
<script src="lib/prototype.js"></script>
```

```
<script src="lib/scriptaculous.js?load=effects"></script>
```

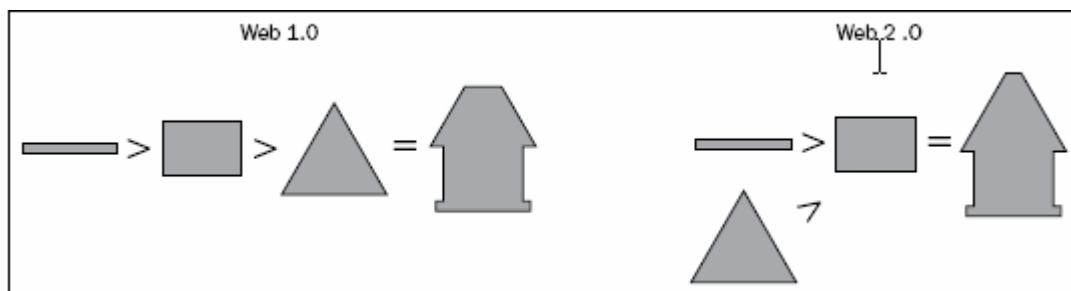
```
<script src="lib/extjs/adapters/prototype/ext-prototype-adapter.js"></script>
```

在引入适配器和基本库之后，我们只需在引入 ext-all.js 或者 ext-all.debug.js 文件。

我是异步的！

web1.0 代码的运行点是建立在完成的基础上的——一直等到每一行代码都执行完全后再进行下一个任务。如同建造一个房子一样，必须在砌墙前打地基，之后才能盖房顶。

在 Ext 中我们可以先盖房顶后打地基。想象一下，房顶正在工厂中制造，而地基此时正在夯实，接下来是筑墙。当一切就绪后，房顶会立刻安置在墙的上方。



这里的介绍我们事先可能没有接触过，也没必要弄得很清楚，像房顶在墙筑好前建造完。我们不需要再一行一行地进行 web 开发了。

Ext 给我们提供了事件和句柄来实现功能。我们可以设计事件来监视墙何时能建好，这样我们就可以第一时间安上房顶。这种思想可能让很多 web 开发者难以接收，但是您一定会很快掌握的。

N: 传统的 JS 警告框会阻止代码的执行，将带来不可预期的结果。你不可以采用传统的警告框，这样可以显示 MessageBox 的同时不产生暂停。

地方化：

Ext 可以在按照您的语言来查看，现在有 40 种翻译（遗憾的是，克林贡文不在其内）。所有的翻译来自用户的贡献——我们需要在自己的母语中使用 Ext。所以我们可以选择自己语言的版本，并把它拷贝到 lib 文件夹下。通过复制语言文件到 lib 文件夹，我们可以编辑它并添加翻译到其中，但要保证在升级时不要覆盖了它。

有三种地方化的方案，对应三种不同的方法：

- 只有英语
- 非英语的单一语种
- 多语言

只有英语：

这种方案不需要做任何改动，也不需要添加任何文件，因为英语的翻译已经存在于 ext-all.js 文件当中了。

一种非英语语言：

第二种方案需要从 build/locale 文件夹中引入一个语言文件。是通过重写英语文本来实现此方案的，所以，它需要在其他所有的库文件引入后才能引入，如下所示：

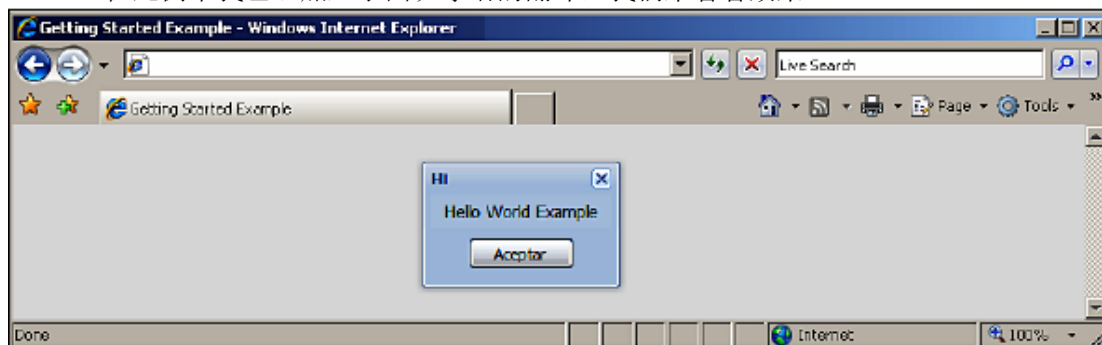
```
<link rel="stylesheet" type="text/css" href="lib/extjs/resources/css/ext-all.css" />
```

```
<script src="lib/extjs/adaptor/ext/ext-base.js"></script>
```

```
<script src="lib/extjs/ext-all-debug.js"></script>
```

```
<script src="lib/extjs/build/locale/ext-lang-es.js"></script>
```

在此例中我已经加入了西班牙语的翻译，我们来看看效果：



部分 UI 界面被本地化了——大概包括日历文本、错误信息、tooltip 信息，分页信息以及加载指示。在你的程序中指定的信息，如 Hi 标题以及 Hello World Example 文本则可以对其进行翻译并加到 `ext-lang-XX.js` 文件中（“XX”是你语言的缩写）或者添加到一个你自定义的语言文件中。推荐的方式是建立一个你自己的语言文件来用来记录改动和增加的内容。这样可以使方便我们准备升级以及修复主要的语言文件。

多语言：

第三种方案是对第二种方案的一点点改动。我们只需要添加一些服务端脚本来使得页面可以在语言之间切换。遗憾的是，语言间的切换并不是完全动态的，换句话说，我们不可能第一时间感受到改动在屏幕上的显示效果。

Ext 在线社区：

Ext 在线社区云集了一大批高手，Ext 开发者也会经常性在论坛上回复大家的问题：

<http://www.extjs.com/forum/>

如果你遇到了困难，可以在论坛上搜索你需要的回答。我建议你使用在 **Learn** 板块中 **Google** 论坛搜索工具，该板块位于：

<http://www.extjs.com/learn/>

H: 在论坛提问之前，您最好附上详细的错误信息，以及错误代码和消息。这是您获得良好反馈的关键所在。

总结：

在这章里，我们介绍了如何正常运行Ext的基础知识以及简单的脚本。我们经常会因为落下了一些细节从而导致在某些错误上浪费宝贵的时间。但是现在你应该准备好了去克服将要出现的错误。

本章的示例让我们看到了Ext的核心所在：提供用户界面。我们只使用了一个对话框，但是，如你所见，展现Ext组件只需要简单的几行代码。这章的主要目的在于让您知道如何获得并安装Ext使其工作。所以，我们可以开始亲手做一些真正精彩的控件。

第二章 Ext 概览（1）

在这章里，我们将要开始使用 Ext 控件，我们将建立一系列的对话框来实现用户和浏览器的交互。我们将使用 `onReady`、`MessageBox`、以及 `get` 方法来了解如何建立不同种类的对话框并且学习如何在页面中修改 HTML 和样式。除此之外，在这章里，我们还会：

- 找到如何简单配置 Ext 控件的方法；
- 用 DOM 做基础实现用户交互；
- 用对话框指出用户想要做的事情；
- 动态地响应用户的输入，并改变 HTML 和 CSS。

我们将要从对 Ext 中几个主要函数的介绍来开始学习。先回想一下第一章中介绍的示例，我们将会对它进行拓展。这些核心函数我们将会在本书的哥哥部分使用：

Ext.onReady: 这个函数用来判断页面已经准备好来执行 Ext 的内容，即用来判断页面是否加载完全；

Ext.Msg: 用来产生一个有固定样式的消息窗口；

configuration objects(配置对象): 这个函数定义控件将如何展示和工作；

Ext.get: 这个函数用来访问和控制 DOM 中的而元素。

准备，出发！

在这部分中，我们将要学习 `onReady` 事件——在您使用 Ext 之前首先要了解东西。我们还讲学习如何显示不同种类的对话框，并且如何相应用户跟对话框的交互。在这之前，我们需要了解使用 Ext 的一些基本准则。

空白图片：

在进行下一步之前，我们应该为 Ext 提供它所需要的——空白图片。Ext 需要一个 1×1 像素的透明 gif 图片，采用不同的方式来拉伸从而填补控件的宽度。我们需要如下定义该图片的路径：

```
Ext.onReady(function(){  
  
    Ext.BLANK_IMAGE_URL = 'images/s.gif';  
  
});
```

您可能会问为什么需要这张图片。Ext 的用户界面是依靠 CSS 的，但是 CSS 是通过为很多 HTML 元素提供样式来拼凑出 Ext 的组件的。唯一能够跨浏览器且保持精准大小的只有图片。所以图片被用来定义 Ext 组件的如何展现。这是 Ext 提供浏览器兼容性的一种主要方式。

控件(widget):

Ext 包含很多控件(widgets)。他们包含了很多的组件(components)，如：消息框、表格、窗口以及其他承担的特定的与用户界面相关的功能。我喜欢把类似于 `onReady` 的组件定义为核心函数或者方法，把能够提供某种用户界面智能的组件叫做控件“`widget`”——如同用来展示数据给用户的表格。

(PS: 由于翻译水平有限，其实我也找不到合适的词来形容 `widget` 和 `component` 了，所以，后面我们凡是说控件，就是指“`widget`”，组件则指“`component`”。)

动作时间:

我们来新建一个页面（或者直接修改“`getting started`”示例）并添加代码以显示一个对话框：

```
Ext.onReady(function(){

    Ext.BLANK_IMAGE_URL = 'images/s.gif';

    Ext.Msg.show({

        title: 'Milton',

        msg: 'Have you seen my stapler?',

        buttons: {

            yes: true,

            no: true,

            cancel: true

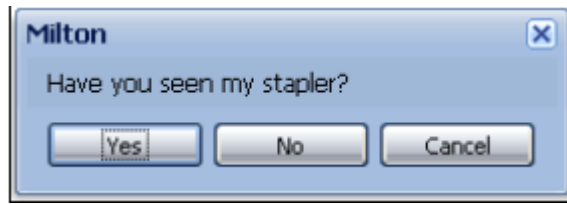
        }

    });

});
```

在之前的章节里，我们已经把 Ext 代码放在了 `onReady` 的 `function` 中了。接下来就可以通过配置对象（`config object`）来配置对话框了。这个配置对象一共包含 3 个元素，最后一个是个嵌套的对象用来展示 3 个按钮。

以下是显示效果：



这看起来是一个最简单的对话框，但是当我们点击东西的时候，Ext 内置的功能变显现出来。这个对话框可以被来回拖动，如同传统的桌面程序一般。同时，你还可以看到一个关闭按钮，在对话框处于激活状态下可以通过按“Esc”键来关闭对话框，或者按 **Cancel** 按钮来关闭对话框。

到底发生了什么？

让我们仔细看一下使用的两个 Ext 的核心函数：

Ext.onReady: 这个函数用来保证让 Ext 在等待 DOM 加载完后执行所有的事情。这个是必须的，因为 JavaScript 实在访问页面的同时执行的，在这个时间点上，DOM 可能还没有生成完全。

Ext.Msg.show: 这是用来建立对话框的主要函数，它管理了对话框得以运行的所有必需的资源。有很多对话框类型的缩略短语，可以帮助您节省时间。我们用几分钟时间来介绍他们。

使用 onReady:

现在检验我们刚才的代码：

```
Ext.onReady(function(){

    Ext.BLANK_IMAGE_URL = 'images/s.gif';

    Ext.Msg.show({

        title: 'Milton',

        msg: 'Have you seen my stapler?',

        buttons: {

            yes: true,

            no: true,

            cancel: true

        }

    });

});
```

`onReady` 函数是用来保证代码等待页面加载完全的。给 `onReady` 传递的是一个函数，可以只传一个函数名，或者直接写在内部，就像我们的示例一样。这种写在 `onReady` 内部的方式叫做匿名函数，用来一次性调用特定的函数。

如果我们想多次执行一个函数，我们需要如下定义：

```
Function stapler(){  
  
    Ext.Msg.show({  
  
        title: 'Milton',  
  
        msg: 'Have you seen my stapler?',  
  
        buttons: {  
  
            yes: true,  
  
            no: true,  
  
            cancel: true  
  
        }  
  
    });  
}  
  
Ext.onReady(stapler());
```

为了不使程序过于庞大，我们最好不要使用匿名函数，我们可以采用可重用的函数。

N: 你可以定义想要显示在按钮上的文字。这样会传递你想要的文字而不是布尔值，你可以这样使用：{yes: 'Maybe'}。

更多的惊喜：

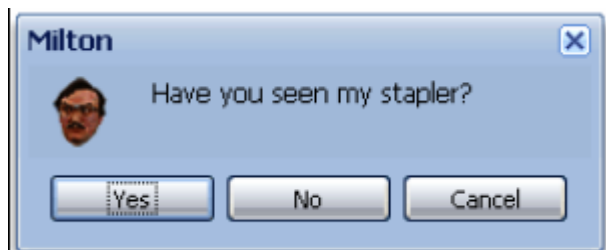
让我们回到正题，现在来美化下我们的程序，添加一些图标和按钮。这一切可以通过添加图标样式来完成。你可以在配置对象里添加 `icon` 和 `buttons` 的条目来完成这一切。

首先我们先讨论下所需要的 `CSS`。添加以下样式到你的页面中：

```
.milton-icon {  
  
    background: url(milton-head-icon.png) no-repeat;  
  
}
```

然后，我们需要改变控件的配置。`icon` 属性的属性值是样式的名称：`milton-icon`（就是选择器的名称），我们可以添加一个函数用来处理用户点击按钮的事件。这些函数是匿名函数，在这个示例中，它只是用来传递变量：

```
Ext.Msg.show({  
    title: 'Milton',  
    msg: 'Have you seen my stapler?',  
    buttons: {  
        yes: true,  
        no: true,  
        cancel: true  
    },  
    icon: 'milton-icon',  
    fn: function(btn) {  
        Ext.Msg.alert('You Clicked', btn);  
    }  
});
```



在我们的例子里，函数只有一个参数，就是点击的按钮的名字。所以当我们单击 **Yes** 按钮，`btn` 变量的值就应该是 `yes`。在示例里我们获得按钮的名字并把它作为警告信息来显示。

N: **Cancel** 按钮的功能是隐藏在内部的，右上角 **close** 图标和 **Esc** 键都能产生取消的动作。这是 **Ext** 提供了一种使得编码简单的方式。

JSON 和配置对象：

我们在示例里使用了配置对象，这是一种告诉 **Ext** 你想怎么做的方法。它提供了很多用来配置相应功能的选项。

传统方式：

以前我们需要在调用函数时传递事先给定的参数，这需要我们记住参数的顺序以便使用该函数。

```
var test = new TestFuntion(  
    'three',  
    'fixed',  
    'arguments'  
);
```

这种传统的函数使用方法会导致很多问题：

- 它要求我们记住参数的顺序；
- 它没有描述参数的含义；
- 参数的可选性很差。

新的方式——配置对象：

利用配置对象，我们可以通过解释文本了解各个变量的意义以及跟更高的灵活性。参数的顺序不再重要，可以随机排列。由于该特性，参数不再让我们束手束脚了。

```
var test = new TestFunction({  
    firstWord: 'three',  
    secondWord: 'fixed',  
    thirdWord: 'arguments'  
});
```

这种方式可以让我们无限制地扩展函数的参数，是加是减完全随意，而且十分简单。还有一个好处就是之前使的函数不会受后来对其添加或者减少参数的影响。

```
var test = new TestFunction({  
    secondWord: 'three'  
});  
  
var test = new TestFunction({  
    secondWord: 'three',
```

```
fourthWord: 'wow'
```

```
});
```

第二章 Ext概览（2）

什么是配置对象（config object）？

如果你对 CSS 和 JSON 熟悉的话，你可以发现配置对象和它们有相似之处，因为他们几乎完全一样。配置对象的作用就是让程序语言更好地读懂数据的结构——这里所谓的程序语言就是 Javascript。

举个例子来看看配置在我们代码中的体现：

```
{  
  
    title: 'Milton',  
  
    msg: 'Have you seen my stapler?',  
  
    buttons: {  
  
        yes: true,  
  
        no: true,  
  
        cancel: true  
  
    },  
  
    icon: 'milton-icon',  
  
    fn: function(btn) {  
  
        Ext.Msg.alert('You Clicked', btn);  
  
    }  
  
}
```

智力的配置可能显得过于复杂，但是一旦我们了解了它，一切就将变的十分简单和迅速。所有的 Ext 控件都需要配置对象，所以我们必须对它十分熟悉。配置对象是我们相当有用的朋友。

这里提几点在使用配置对象时的注意事项：

- 配置条目（record）要被花括号包围，每一个括号中的条目都是对象的一部分——{records};
- 每个条目都有属性名和属性值，其间被冒号分割，条目之间靠逗号分割——{name0: value0, name1: value1};
- 条目的属性值可以是任何数据类型，包括布尔型，数组，函数以及对象——{name0: true, name1: {name2: value2}};

- 方括号代表数组——`{name: ['one', 'two', 'three']}`。数组中的元素也可以是包含对象、值与数字的任何东西。

使用 JSON 格式的最大好处就是你可以通过添加配置来修改控件。不像原始的函数，配置选项现在变得毫不相干，可以任意增减。

JSON 是如何工作的？

你可能听别人提起过 `eval`，它就大体上来自于 JSON。`eval` 函数就是 JavaScript 用来解释 JSON 字符串的，把字符串转化为对象、数组、函数或者其它。

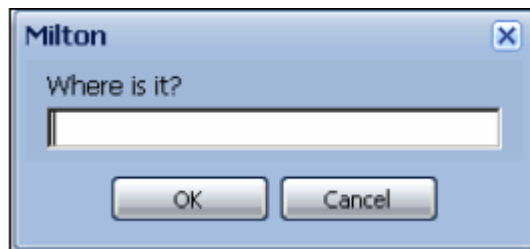
该行动了：

好了，我们了解了 Ext 的工作机制并对使用者进行了提问。现在我们可以采用问题的回答了。让我们在对话框中添加函数以用来表明在点击按钮后执行什么相应。`switch` 可以帮助我们完成这些：

```
fn: function(btn) {  
    switch(btn){  
        case 'yes':  
            Ext.Msg.prompt('Milton', 'Where is it?');  
            break;  
        case 'no':  
            Ext.Msg.alert('Milton',  
                'Im going to burn the building down!');  
            break;  
        case 'cancel':  
            Ext.Msg.wait('Saving tables to disk...','File Copy');  
            break;  
    }  
}
```

还记得前面提及的对话种类吗？我们现在介绍一些，它们帮助我们完成特定的任务，不需要我们为区分任务的不同而专门进行配置。

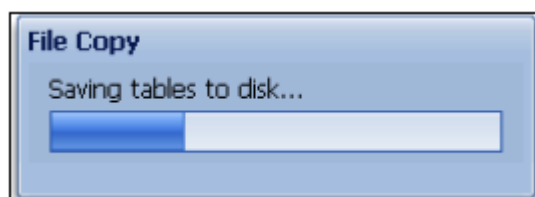
点击 **OK** 你可以得到一个提示（`prompt`）对话框。提示对话框的作用是让你输入单一的值，在几乎所有的用户界面中它是一个标准的元素。



点击 **No** 你会得到一个警告(alert)。你一定熟悉 JS 中的传统的警告框。我还清楚地记得第一次使用 JS 的警告框的时候，我十分激动，警告的信息是：“如果你是傻子，请点击 OK”。



点击 **Cancel** 按钮（或者点击 close 按钮或者按下 Esc 键）你可以得到进度对话框。



Ext 可以控制进度对话框并通知它何时消失。但是为了简单，在本例中，我们让它一直运行。

N: Ext 中内置了按钮的聚焦和 Tab 键带来的顺序移动。一般来说 **OK** 或者 **Yes** 是默认的动作。按回车键可以触发这个按钮，按 **Tab** 键可以让焦点在对话框中的按钮和其他元素上移动。

点火:

现在我们可以添加一些基于用户对于对话框操作的动作响应。我们将通过修改 switch 结构来管理 Yes 按钮的单击。提示窗口 (prompt) 可以接收第三个参数，就是点击 Yes 按钮后执行的函数。我们第一这个函数来检验用户输入的值是否等于 “the office”，如果不等于，把该值写在页面的 DIV 标签中，如果值等于 “the office” 的话，我们在 DIV 标签中默认添加 “Dull Work” 文本。在同一个 DIV 上，我们采用一个 “Swingline” 订书机作为背景图片。

```
case 'yes':
```

```
Ext.Msg.prompt('Milton', 'Where is it?', function(btn,txt)
```

```
{
```

```

        if (txt.toLowerCase() == 'the office') {

            Ext.get('my_id').dom.innerHTML = 'Dull Work';

        }else{

            Ext.get('my_id').dom.innerHTML = txt;

        }

        Ext.DomHelper.applyStyles('my_id',{

            background: 'transparent

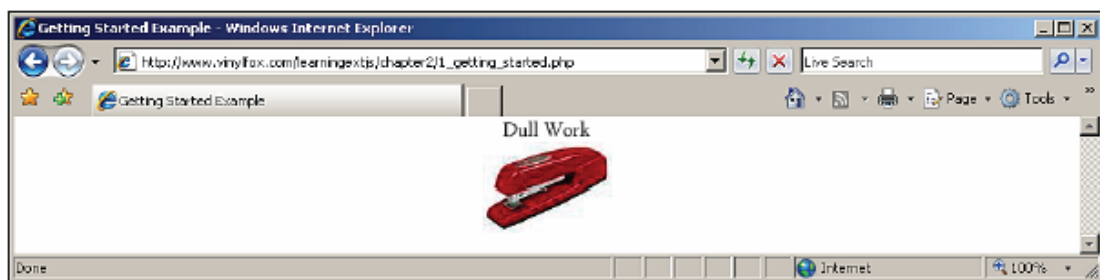
            url(images/stapler.png) 50% 50% no-repeat'

        });

    });

    break;

```



对于 no 的情况，将会出现一个警告对话框，在点击 **No** 按钮的同时，页面也将会改变：

```

case 'no':

    Ext.Msg.alert('Milton','Im going to burn the building down!',

        function() {

            Ext.DomHelper.applyStyles('my_id',{

                'background': 'transparent url(images/fire.png) 0 100%

repeat-x'

            });

            Ext.DomHelper.applyStyles(Ext.getBody(),{

                'background-color': '#FF0000'

            });

            Ext.getBody().highlight('FFCC00',{

```

```

        endColor:'FF0000',

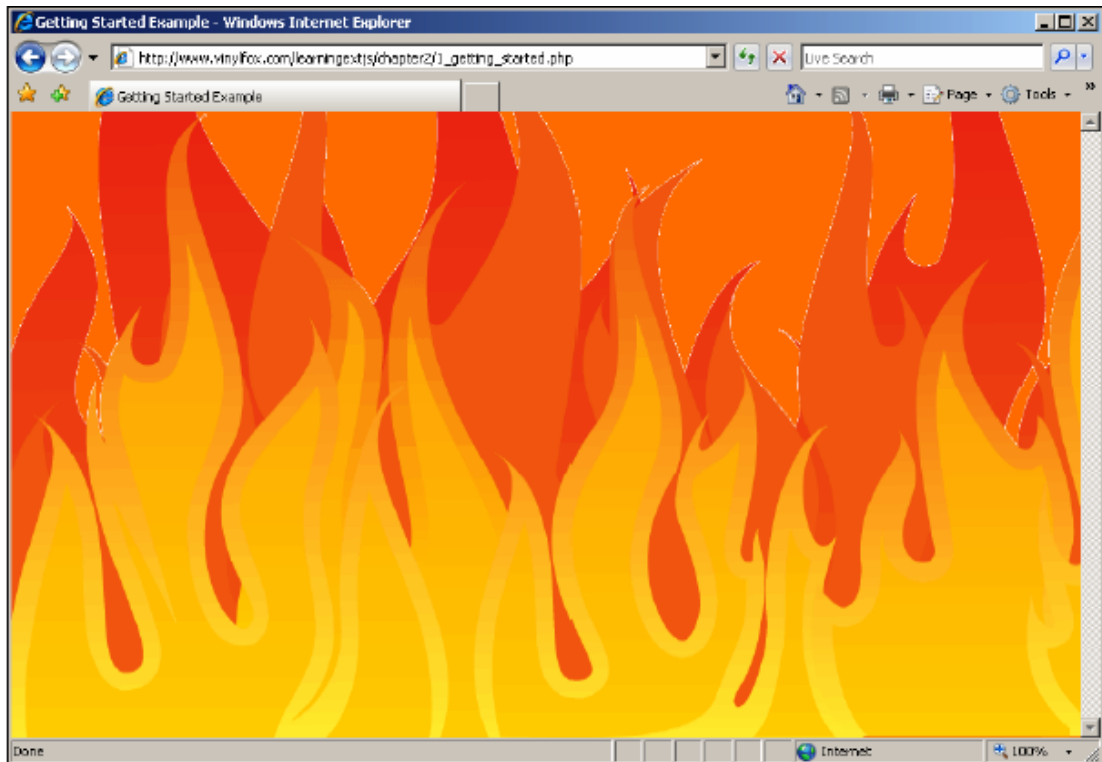
        duration: 6

    });

});

break;

```



肩负重担的 Ext.get:

Ext 之所以工作的这么出色，是因为他有一套访问 DOM 的机制，很多函数和方法都是用来操控 DOM 的，这些方法中，有一个最为常用：

```
Ext.get('my_id');
```

它让我们可以通过页面中的元素 ID 来访问某个元素。如果我们回首看一下第一个例子，里头采用了 `getBody`，我们就是靠其来获得 `body` 元素并为之添加效果的。现在对于 `my_id` 来说，我们需要真正添加一个 `my_id` 元素：

```
<div id='my_id' style='width:200px;height:200px;'>test</div>
```

当我们把上面的 `div` 加入页面中，并把之前的对 `body` 的引用转化为对 `my_id` 的引用，显示的效果就只发生在 `id` 为 `my_id` 的 `div` 上。

```
Ext.get('my_id').highlight('FF0000',{
```

```
endColor:'0000FF', duration: 3

});
```

现在让我们看下浏览器，你将会发现只有 200 像素的区域改变了颜色，而不是整个页面。

切记 ID 一定要唯一，一旦采用了 `my_id` 就不能再在同一页面中使用相同的 `id` 了。当 ID 重复时，用 `Ext.get` 只能访问最后一个。这只能被当做一个 `bug` 而不能当做一种设计。对于大部分代码，`Ext` 建立并跟踪自己的 ID，在大多数情况下，我们只需要默认使用 `Ext` 对页面元素的自动跟踪，而不需要亲自来创建元素或者定位它们。

N: 重复的 ID 会导致奇怪的现象，如控件只显示在浏览器左上角，所以应该尽量避免重复 ID。

效率上的建议：

这个不能完全算一个效率上的建议，但是采用“flyweight”完成任务可以节省内存，这样可以通过避免占用浏览器的内存来提高效率。

上面那个高亮效果的例子可以采用 `flyweight` 来重写：

```
Ext.fly('my_id').highlight('FF0000',{

    endColor:'0000FF', duration: 3

});
```

`flyweight` 被用在我们想用一行代码对某一元素执行一次操作时，而且我们不需要再次使用之前靠 `flyweight` 引用的元素。在每次被调用时，`flyweight` 将会只重复使用同一内存。

下面这个使用 `flyweight` 的例子是不正确的：

```
var my_id = Ext.fly('my_id');

Ext.fly('another_id');

my_id.highlight('FF0000',{endColor:'0000FF', duration: 3});
```

因为 `flyweight` 在被调用时使用的是同一内存，所以当我们使用 `my_id` 来改变高亮显示的时候，引用已经指向了 `another_id` 元素。

总结：

只用了几行代码，我们就创建了一个能让您兴奋半天的有趣的程序。好吧，可能达不到半天，但是至少会有几分钟吧。但是，我们已经实现了传统桌面程序的基本功能和用户界面。

我们学习使用了基本的配置对象（**config object**），我确定我们会在使用更多控件后了解更多只是。配置对象是我们使用 **Ext** 的基础，你对它接受的越快，接下来的学习就会越简单。

如果你对配置对象不能完全适应的话，请不要担心，我们还会通过大量的事件来对它进行说明，现在，我们把话题转移到我最喜欢的东西之一——表单(**forms**)。

.

第三章 表单（1）

在这章里，我们将学习 Ext 表单(forms)。它和 HTML 的表单相似，但却没有约束和枯燥的界面。我们用不同的表单字段类型来生成一个表单，而且它的验证和提交过程是异步的。然后我们会制作数据库驱动的下拉菜单，即 **ComboBox**，然后添加复杂的字段验证和 **mask**（相当于一种掩码）。然后我们再进行进一步的改进，为其添加一系列的令人啧啧的功能。

本章的目的包括：

- 制作一个靠 **AJAX** 提交的表格；
- 检验字段数据并建立常用验证；
- 从数据库中加载表格。

表单的主要组件：

Ext 的表单蕴含了无穷无尽的可能。按键监听、检验、错误信息、输入约束都在配置项里有支持。扩展表单的配置以达到你的需要是十分容易的，这在本章中会做相应介绍。接下来介绍一些表单的核心组件：

- **Ext.form.FormPanel**：有点像传统 HTML 的表单，它是一个结合了很多字段的的面板；
- **Ext.form.Field**：他是表单中主要的元素，用来和用户交互，也是很多字段的基础，可以拿它和 HTML 中的 **INPUT** 标签来做类比。

我们的第一个表单：

为了开始，让我们先做一个集合了各种字段类型的表单：一个日期选择器、校验、错误信息和 **AJAX** 提交——放心，对于我们第一尝试来讲，这个表单并不复杂。

在这个示例里，我们通过配置对象来建立字段，而不是通过去实例化一个 **Ext.form.Field** 组件。这种方法很好，可以节约很多时间，也能让代码运行更快。前面示例中饿 HTML 页面可以做本示例的开始（你可以免去很多引入 Ext 库的工作）。Ext 库文件一定要被正确引入，Ext 的代码要放在 **onReady** 函数中。

```
Ext.onReady(function(){  
  
    var movie_form = new Ext.FormPanel({  
  
        url: 'movie-form-submit.php',  
  
        renderTo: document.body,  
  
        frame: true,
```

```

        title: 'Movie Information Form',

        width: 250,

        items: [{

            xtype: 'textfield',

            fieldLabel: 'Title',

            name: 'title'

        }, {

            xtype: 'textfield',

            fieldLabel: 'Director',

            name: 'director'

        }, {

            xtype: 'datefield',

            fieldLabel: 'Released',

            name: 'released'

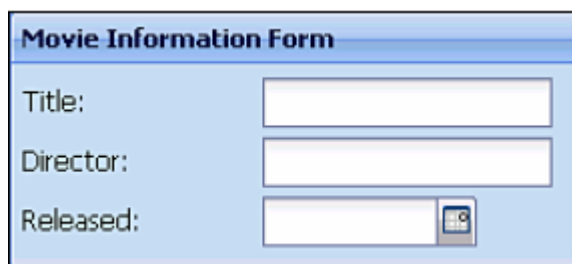
        }

    ]

});

```

当我们在浏览器中运行上面的代码，会出现一个如下的表单面板：



好表单——它是怎么工作的？

这个 `FormPanel` 和 HTML 表单很相似。它的作用是提供字段的容器。我们的表单有一个 `url` 配置项，让我们的表单知道提交的地址。它还有一个 `renderTo` 配置项，定义了 `form` 渲染显示的位置。

`items` 配置项很重要，它是用来定义表单中包含的字段集合的。它是一个数组，元素就是表单中的各个字段。每个字段都有一个 `xtype` 属性用来说明字段组件的种类：`text`、`date` 或者 `number`。它甚至可以是一个表格或者其他 `Ext` 组件。

表单字段：

现在我们知道字段类型是靠 `xtype` 定义的。但是 `xtypes` 源自何处，有多少种呢？一个 `xtype` 对应了一个特定的 `Ext` 组件，所以“`textfield`” `xtype` 指的是 `Ext.form.TextField`。这里罗列出一些 `xtype`：

- `textfield`
- `timefield`
- `numberfield`
- `datefield`
- `combo`
- `textarea`

因为这些多是 `Ext` 组件，所以我们可以将其设置为表格、工具栏或者按钮——好多东西。`Ext` 的重用准则的重要体现就在于一切都是可以相互替换的，所有的组件都共享了核心的函数和方法。所以 `Ext` 可以以不变应万变，在各种需求下游刃有余。

我们的基本字段配置如下：

```
{  
  
    xtype: 'textfield',  
  
    fieldLabel: 'Title',  
  
    name: 'title'  
  
}
```

当然，我们有 `xtype` 来定义字段的类型——本例中是 `textfield`。`fieldLabel` 值得是字段左边的标签，它还可以被配置为显示在字段的上部或者右部。`name` 配置项如同在 `HTML` 中一样，在于后台交互时作为变量的名称。

N: `name` 在 `Ext` 大多数组件的配置项里的意义如同 `HTML` 中一样。因为 `Ext` 毕竟是 `web` 开发者的杰作，也就会服务于 `web` 开发者。

对于日期字段，和文本字段（`text field`）差不多，只需改变 `xtype` 的类型：

```
{  
  
    xtype: 'datefield',
```

```
        fieldLabel: 'Released',

        name: 'released'

    }
}
```

校验:

示例中的一些组件可以添加校验来在用户操作或者输入错误时报错。我们为自己的第一个表单添加校验。一种常用的校验是看用户是否输入了任何内容，我们把该检验运用于 **movie title**（指的就是第一个文本字段，看来作者对电影情有独钟），换句话说，我们要让这个字段符合要求：

```
{

    xtype: 'textfield',

    fieldLabel: 'Title',

    name: 'title',

    allowBlank: false

}
```

建立一个 **allowBlank** 配置项并把它设置为 **false**（默认为 **true**）就够了。很多表单都对字段有这样的要求。

每种 **Ext** 表单字段都有它独特的数据校验方式。举例来说，时间字段可以阻止你选择一个星期的特定几天，或者使用正则表达式来排除一些日期。一下的代码用来排除（**disable**）除了周六和周日以外的所有日子。

```
{

    xtype: 'datefield',

    fieldLabel: 'Released',

    name: 'released',

    disabledDays: [1,2,3,4,5]

}
```

Movie Information Form

Title:

Director:

Released:

January 2008

S	M	T	W	T	F	S
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Today

在这个例子里，除了周六、日以外的其他日子都被排除掉了，一定要记得 0 代表 Sunday！

当我们使用其他种类的字段时，我们还有不同的校验方法，如数字字段（number field），我恶魔呢可以添加数字大小的限制或者小数的位数。标准的校验都可以在 API 中找到。

内置校验——vtype:

另一种复杂的校验方式叫做 vtype。它可以被用来校验和约束用户的输入，并且抛出错误的信息。它可以在任何场景里使用，只要你能想得到！因为它的基础是正则表达式，现在介绍几种用得着的 vtype:

- email;
- url;
- alpha;
- alphanum。

这些内置的 vtype 被规定的过于单一，你可以把它作为建立你自己的 vtype 的基础。

以下是一个 alpha vtype，用 QuickTips（Ext 的一种提示信息的方式）气球图标作为错误消息。

```
Ext.onReady(function(){

    var movie_form = new Ext.FormPanel({

        url: 'movie-form-submit.php',

        renderTo: document.body,

        frame: true,

        title: 'Movie Information Form',

        width: 250,

        items: [{

            xtype: 'textfield',

            fieldLabel: 'Title',

            name: 'title',

            allowBlank: false

        },{

            xtype: 'textfield',

            fieldLabel: 'Director',

            name: 'director',

            vtype: 'alpha'

        },{

            xtype: 'datefield',

            fieldLabel: 'Released',

            name: 'released',

            disabledDays: [1,2,3,4,5]

        }

    ]

});

});
```

我们所做的一切就是为名字叫做 **director**（导演）的字段添加一个 **vtype**。它被用来限定输入只能匹配文字字符。

A screenshot of a web form titled "Movie Information Form". It has three input fields: "Title:" (empty), "Director:" (filled with "MikeJudge"), and "Released:" (empty). The "Released:" field has a small calendar icon to its right.

现在我们发现自己的 **vtype** 过于基础，**alpha vtype** 只能限制输入的为字符。在本例中，我们限制用户输入导演的名字只能为字符，包含一个空格或者连字符。大写第一个字符可以让显示更加的美观。

N: 搜索 **Ext** 论坛可以让你从别人那里获得你想要的 **vtype**，或者以此为参照做出你想要的效果。

显示错误的方式：

Ext 表单的默认报错方式，是在你输入的位置添加一个红色的框体。有点模仿微软在拼写单词错误的意思，我们还有其他显示错误信息的配置方式，但是你需要告诉 **Ext** 来使用它们。

一个好的配置就是让你的错误信息通过气球（就是提示信息前有一个貌似气球的小图标）信息来提示。在报错的同时，它会同时使用 **Ext** 默认的弯曲的曲线狂来提示，同时还会在鼠标移近时显示出一个气球提示框。

Two side-by-side screenshots of the "Movie Information Form" showing error messages. The left screenshot shows the "Director:" field with a red dashed border and a red balloon icon next to it containing the text "This field is required". The right screenshot shows the "Released:" field with a red dashed border and a red balloon icon next to it containing the text "This field should only contain letters and _".

我们只需要在表单创建前添加一行代码来初始化气球提示信息。一般位于 **onReady** 函数中的第一行：

```
Ext.onReady(function(){  
  
    Ext.QuickTips.init();  
  
    // our form here  
  
});
```

这就是我们用来显示可爱的气球提示信息所需要做的准备。

第三章 表单（2）

自定义校验——建立你自己的 **vtype**:

如果你和我一样，正则表达式会让你对着电脑屏幕犯傻的（作者太谦虚了，实则正则表达式的确不容易），所以我一直在寻觅和我的需求相关的正则表达式并对之加以修改而不是从起跑线开始去忍受学习正字表达式（regular expressions）的痛苦。

在建立一个自己的 **vtype** 之前我们需要把它添加到 **vtype** 的定义中去，每一个定义都包含了值（**value**），掩码（**mask**）和用来测试字段值的函数。

- **xxxVal**: 这是用来匹配的正则表达式;
- **xxxMask**: 这是用来约束用户输入的掩码;
- **xxxText**: 这是用来显示的错误信息。

上例表格中的导演(**director**)字段正实用于我们发挥正则表达式的作用——所以我们先来尝试一个。我们所需的正则表达式应该能匹配两个别都空格分割的字符串，并且每个字符串应该以大写字母开头，对于校验名字来说，这听起来很不错，对吗？

```
Ext.form.VTypes['nameVal'] = /^[A-Z][A-Za-z\ -]+[A-Z][A-Za-z\ -]+$/;
```

```
Ext.form.VTypes['nameMask'] = /[A-Za-z\ - ]/;
```

```
Ext.form.VTypes['nameText'] = 'In-valid Director Name.';
```

```
Ext.form.VTypes['name'] = function(v){  
    return Ext.form.VTypes['nameVal'].test(v);  
}
```

一气看懂太难了，所以我们各个攻破。我们先从正则表达式开始：

```
Ext.form.VTypes['nameVal'] = /^[A-Z][A-Za-z\ -]+[A-Z][A-Za-z\ -]+$/;
```

然后我们添加了掩码（**mask**），它定义了什么样的字符能够在表单中填写，同样用正则表达式来表示：

```
Ext.form.VTypes['nameMask'] = /[A-Za-z]/;（作者在这里有误，合起来写的时候允许采用分隔符“-”，分开写的又没有分隔符了，不是我的错哦）
```

然后，我们定义在气球报错消息中应该展现的文字：

```
Ext.form.VTypes['nameText'] = 'In-valid Director Name.';
```

最后是将一切“凝聚”起来的部分——用来测试字段值是否正确的函数：

```
Ext.form.VTypes['name'] = function(v){
```

```

return Ext.form.VTypes['nameVal'].test(v);
}

```

在把这些添加完全后我们就定义了自己的 **vtype**，其实并没花太多精力就做出了这样可以重复使用的校验。

Masking（掩码）——不要按那个键盘！

Masking 是用来让字段只接受特定的按键，如数字或者字母甚至大写字母。它无所不能，因为它依靠的是正则表达式。一下的这个 **mask** 的示例就是让用户只能输入大写字母：

```
masker: /A-Z/
```

考虑通过建立一个 **vType** 来完成 **masking** 的功能来替代 **masking** 的配置。如果格式化的需求发生了改变，这样做会方便做相应的改动的。

所以当有一天，你的老板对你气急败坏的说：“记住产品编号以前只能是十位数字，现在它变成八位了”，这时你就可以改变你的 **vType**，然后用这天其他的时间玩吉他去吧。

单选按钮（radio buttons）和复选框（check boxes）：

我们是离不开单选按钮和复选框的，它们很笨拙并且难于着手。我一般不到“绝境”是不会用这两个东西的。但是我们还是得把它们加进来，这样我们就可以安心地说我们做过了！

它不是按钮，是一个单选按钮：

我们首先在表格中添加一组单选按钮：

```

{
    xtype: 'radio',
    fieldLabel: 'Filmed In',
    name: 'filmed_in',
    boxLabel: 'Color'
}, {
    xtype: 'radio',
    hideLabel: true,
    labelSeparator: '',
    name: 'filmed_in',
    boxLabel: 'Black & White'
}

```

这些单选按钮的工作机制和 HTML 中的单选按钮一样。他们每一个按钮被赋予了相同的名字，并且协同工作。我一般喜欢为拓展的单选按钮隐藏标签，只需把 `hideLabel` 属性设置为 `true` 并且把 `LabelSeparator` 属性设置为空值，这样做出的单选按钮会比较简洁。

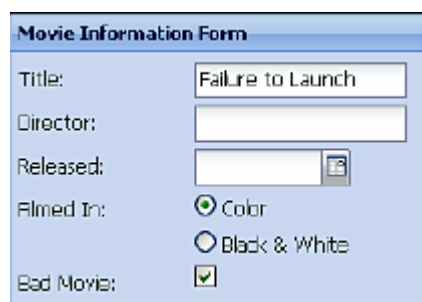
用复选框表示“是否”：


有的时候，我们需要用复选框来表示布尔值——如同一个开关：

```

{
    xtype: 'checkbox',
    fieldLabel: 'Bad Movie',
    name: 'bad_movie'
}

```



Movie Information Form	
Title:	<input type="text" value="Failure to Launch"/>
Director:	<input type="text"/>
Released:	<input type="text"/> 
Filmed In:	<input checked="" type="radio"/> Color <input type="radio"/> Black & White
Bad Movie:	<input checked="" type="checkbox"/>

组合框(ComboBox):

ComboBox 如同 HTML 中的 SELECT 一样，也被叫做下拉菜单，是一个经常用到的表单元素。它可以让用户不必从键盘输入，从而省去了一些时间。Ext 的组合框有很多用处，包含很多的配置项。

首先，我们建立一个使用本地数据的 combo。为了实现这个功能，我们需要建立一个 data store（数据集合）。data store 的种类有许多种，每一种可以被用在特定的情况下。但是对于这个例子，我们使用一个 simple store:

```
var genres = new Ext.data.SimpleStore({  
    fields: ['id', 'genre'],  
    data : [['1','Comedy'], ['2','Drama'], ['3','Action']]  
});
```

如同其他表单中的字段一样，对于 combobox 我们也要为它添加配置。store 配置项就是用来说明 combo 中采用的数据的。与此同时我们还需要 mode 配置项，用来说明数据的来源是 local source（本地）还是 remote source（远程），当然还有 displayField 数据项，用来说明把哪一列数据展现在 combo 的选项中：

```
{  
    xtype: 'combo',  
    name: 'genre',  
    fieldLabel: 'Genre',  
    mode: 'local',  
    store: genres,  
    displayField: 'genre',  
    width: 120  
}
```

这里的 combo box 采用的是本地数据。本地数据一般在数据量很小且不经常改变时候采用。但是，当我们需要从数据库中拿出数据时该怎么办呢？

数据库驱动的 ComboBox:

最大的区别在于服务器端——获得数据后并把它转化为 combo box 可以使用 JSON 字符串。不论服务器端语言是什么，我们都需要 JSON 来编码数据。如果我们使用的是 PHP 5.1 或者更高的版本的话，这一切都是内置好的。

N: 我们可以通过在终端窗口执行一条命令或者执行一条 PHP 代码来得知 PHP 的版本。如果采用命令行 `php -v` 可以显示版本。如果采用 PHP 代码，如下所示一样可以检测到版本：

```
<?php phpinfo(); ?>
```

以下是我们通过 PHP 5.1 以及其更高版本来获取 JSON 数据的方式：

```
<?php

// connection to database goes here

$result = mysql_query('SELECT id, genre_name FROM genres');

if (mysql_num_rows($result) > 0) {

    while ($obj = mysql_fetch_object($result)) {

        $arr[] = $obj;

    }

}

Echo '{rows:'.json_encode($arr).'}';

?>
```

当我们使用远程数据，还需要做一些准备。首先，**data store** 需要知道数据的格式。我们通过 **data reader** 来实现这一功能，这里，我们采用 **JSON Reader**。

```
var genres = new Ext.data.Store({

    reader: new Ext.data.JsonReader({

        fields: ['id', 'genre_name'],

        root: 'rows'

    }),

    proxy: new Ext.data.HttpProxy({

        url: 'data/genres.php'

    })

});
```

data reader 的第一个参数是该 **reader** 的配置对象——用来说明 **fields**(数据字段) 有哪些以及 **field** 的名字。注意，我们没有使用 **sort_order**——对于数据集来见 **field** 并

不是现成的。`root` 代表了包含数据数组的元素，在本例中叫做 `rows`，你也可以给它取名叫做 `bobs-crab-shack`，或者任何你想取的名字：

```
{rows:[
  {
    "id":"1",
    "genre_name":"Comedy",
    "sort_order":"0"
  },
  {
    "id":"2",
    "genre_name":"Drama",
    "sort_order":"1"
  }
// snip...//
}]
}
```

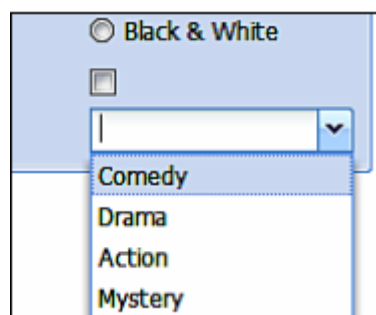
我们也建立了 `proxy`(代理)——通常为 `HTTP proxy`，可以取回和页面在同一域（`domain`）中的数据。这是一个最普通的方法，还有 `ScriptTagProxy`，可以从不同的域中拿到数据。我们只需要为 `proxy` 提供 `URL` 属性来让它知道从哪里获得数据。

N: 当我们采用 `proxy` 其实也就采用了 `AJAX`。它需要后台在运行，否则 `AJAX` 是不会工作的。你可以在浏览器所在系统中运行以上代码（无服务器支持），`AJAX` 是不会工作的。

我们还需要通过 `load` 方法来让数据在用户与其交互前加载到 `combo box` 中：

```
genres.load();
```

以下是一个数据库驱动的 `combo box`：



还有一种提前加载数据的方式，把 `autoLoad` 配置项设置为 `true`:

```
var genres = new Ext.data.Store({  
    reader: new Ext.data.JsonReader({  
        fields: ['id', 'genre_name'],  
        root: 'rows'  
    }  
    },  
    proxy: new Ext.data.HttpProxy({  
        url: 'data/genres.php'  
    }  
    },  
    autoLoad: true  
});
```

`TextArea`（文本域）和 `HTMLEditor`（html 编辑器）

我们为电影信息提供一个 `text field`（文本字段），`Ext` 会为之提供很多的配置项。`women` 可以使用标准的 `textarea`，如同在传统的 `HTML` 中，或者使用 `HTMLEditor`，可以为我们提供丰富的编辑功能：

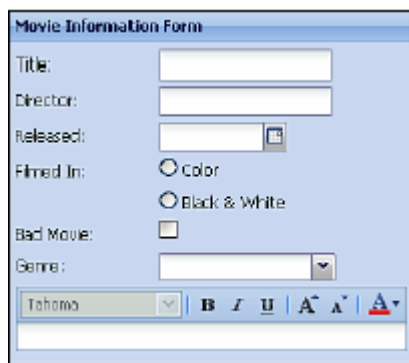
- `textarea`：如同传统的 `HTML` 中的 `textarea` 字段；
- `htmleditor`：一个带按钮工具栏的文本编辑器。

如果我们把 `hideLabel` 设置为 `true` 并且清除了标签分隔符(`label separator`)，我们就可以使用 `textarea` 来横向填充表单面板。这样做会很好看：

```
{  
    xtype: 'textarea',  
    name: 'description',  
    hideLabel: true,  
    labelSeparator: '',  
    height: 100,  
    anchor: '100%'  
}
```


只需改变 **xtype**，如下所示。我们就可以添加一个 **HTMLEditor**。他包含了丰富的诸如字体、字体大小、加粗、倾斜等的内置按钮。这是第一个我们需要 **QuickTips** 组件的 **Ext** 组件，我们必须先初始化 **QuickTips** 然后才能使用 **HTMLEditor**。

```
{  
  
    xtype: 'htmleditor',  
  
    name: 'description',  
  
    hideLabel: true,  
  
    labelSeparator: ",  
  
    height: 100,  
  
    anchor: '100%'  
  
}
```



监听表单字段事件：

Ext 使得对用户某个动作的监听特别简单，诸如单击某个元素或者按下某个键盘上的键。

一个经常性的任务就是监听回车按键，然后提交表单，让我们看如何完成这一任务：

```
{  
  
    xtype: 'textfield',  
  
    fieldLabel: 'Title',  
  
    name: 'title',  
  
    allowBlank: false,  
  
    listeners: {  
  
        specialkey: function(f,e){
```

```
        if (e.getKey() == e.ENTER) {  
            movie_form.getForm().submit();  
        }  
    }  
}  
}
```

specialkey 监听器（**listener**）用来监听任何你需要监听的按键。这个监听器可以在上下左右箭头按键被按下时触发，或者 **Tab**、**Esc** 等等。这就是为什么我们要先判断是否按下的是回车键，这样的话，表单只能通过回车键提交。

第三章 表单（3）

ComboBox 事件：

貌似所有的 combo box 都需要绑定事件。现在我们建立一个叫做 genres 的 combo box 并监听选择下拉菜单条目的事件。

首先我们添加一些数据，第一项叫做 New Genre：

```
var genres = new Ext.data.SimpleStore({  
    fields: ['id', 'genre'],  
    data : [  
        ['0','New Genre'],  
        ['1','Comedy'],  
        ['2','Drama'],  
        ['3','Action']  
    ]  
});
```

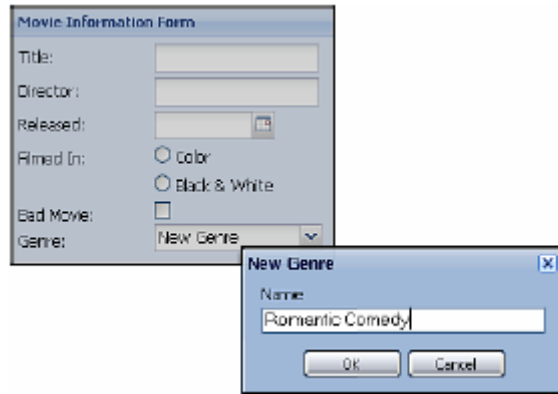
然后添加监听器(listener)：

```
{  
    xtype: 'combo',  
    name: 'genre',  
    fieldLabel: 'Genre',  
    mode: 'local',  
    store: genres,  
    displayField:'genre',  
    width: 130,  
    listeners: {  
        select: function(f,r,i){  
            if (i == 0){  
                Ext.Msg.prompt('New Genre','Name',Ext.emptyFn);  
            }  
        }  
    }  
}
```

```

    }
}
}
}

```



这个监听器备用来等待监听选择事件并且执行相应的函数。每一种监听器都有它自己默认传递的参数——可以从 API 中查找到。

对于选择事件，我们的函数传递了一下三个东西：

- 表单字段；
- 被选择的下拉项目的数据记录；
- 被点击的条目的 **index**（索引）。

当某一项被选中时，我们可以在列表中查看到被选中的项。第三个参数就传递了被点击项目的索引。如果它的值是 **0**（第一个下拉列表项），就会弹出一个提示对话框并且要求用户来添加一个新的 **genre**，这在前面的一章里介绍过。

N: 每一个 Ext 组件都有监听器(listener)。这些被监听的事件都可以在 API 的底部找到。

按钮和表单动作：

现在我们只存留一个问题，现在的这个表单还不具备提交功能。为了实现这个功能，我们需要新建一些按钮。

按钮的添加靠的是配置对象中的 **buttons** 配置项。这些按钮其实只需要 **2** 个东西：显示在其上的文本和点击后执行的函数。

```

buttons: [{
    text: 'Save',
    handler: function(){

```

```

        movie_form.getForm().submit({
            success: function(f,a){
                Ext.Msg.alert('Success', 'It worked');
            },
            failure: function(f,a){
                Ext.Msg.alert('Warning', 'Error');
            }
        });
    }
}, {
    text: 'Reset',
    handler: function(){
        movie_form.getForm().reset();
    }
}]

```

`handler` 提供了对一个函数的引用——当点击时候触发的函数。在这个示例里，我们使用了匿名函数。

表单提交：

`FormPanel` 有一个 `url` 配置项用来说明发送数据的目的文件。这个十分简单——就像传统 `HTML` 中的表单，所有字段信息将被发送到这个 `url`，然后在服务器端做处理。

在 **Save** 按钮里，我们有一个匿名函数来执行接下来的代码来进行真正的提交，把数据通过 `AJAX` 传送给服务器。不需要刷新页面，所有都在后台完成，页面看起来保持一样：

```

movie_form.getForm().submit({
    success: function(f,a){
        Ext.Msg.alert('Success', 'It worked');
    },
    failure: function(f,a){

```

```
Ext.Msg.alert('Warning', 'Error');

    }

});
```

N: 为了让提交正常，代码必须配合服务器端执行。

`success` 和 `failure` 配置项用来在服务器响应后调用 `submit`（提交）句柄。他们都是异步函数，但是可以通过引用之前定义的函数来替代。

你发现了吗，这些函数都会传递 2 个参数？它们是用来指出服务器端给予的响应的。但是首先我们要讨论下如何从服务器端提供响应。

顶嘴（作者以此说明服务器端的响应）——服务器的响应

当表单提交后，服务器会处理提交的数据来确定是返回“SUCCESS”消息是 `true` 还是 `false` 并且传回客户端。错误信息也可以通过响应来传递，其中可以包含和表单字段名称相对应的信息。

当使用表单和服务器端的校验时，需要给 `success` 赋予布尔值。下面举一例：

```
{
    success: false,
    errors: {
        title: "Sounds like a Chick Flick"
    }
}
```

当 `success` 的标志位为 `false` 时，它会然 `Ext` 去读它带来的错误信息，然后添加到表单验证中并给予用户错误提示。

服务器端的校验给我们提供了在服务器端检验信息的方式，并以此返回错误。如果我们有一个数据库，记录了所有的不良电影，那么我们可以在服务器端进行校验，然后通过数据库检索电影名字来进行响应。

如果我们行过滤掉 `chick flicks`（专门给女人看的一种情感剧），我们可以这样做：

```
{
    success: false,
    errors: {
        title: "Sounds like a Chick Flick"
    }
}
```

```

    },

    errormsg: "That movie title sounds like a chick flick."

}

```

当 **SUCCESS** 的值为 **false** 时，会触发表单显示错误信息。一个 **errors** 对象都包含于响应之中。表单通过这个对象来决定错误信息。**name/value** 的形式说明了对应的表单字段上显示的错误信息，如本例的 **title**: “**Sounds like a Chick Flick**”。

本例的响应里还包含了一个 **errormsg**。它不是被用来提供给表单的，它将被用来展示我们自己的错误信息。让我们用它来显示一个警告对话框。

```

buttons: [{

    text: 'Save',

    handler: function(){

        movie_form.getForm().submit({

            success: function(f,a){

                Ext.Msg.alert('Success', 'It worked');

            },

            failure: function(f,a){

                Ext.Msg.alert('Warning', a.result.errormsg);

            }

        });

    }

}, {

    text: 'Reset',

    handler: function(){

        movie_form.getForm().reset();

    }

}]

```

提交动作向 **success** 和 **failure** 句柄中传递了参数。第一个是 **Ext** 表单对象，第二个是 **Ext** 的 **action** 对象，让我们来研究一下 **Ext action** 对象：

选项		说明
----	--	----

failureType	String	报告服务器端和客户端的错误信息
response	Object	包含了服务器端的原始响应信息，包括有用的 header 信息
result	Object	从服务器端响应的 JSON 对象
type	String	在提交和加载时执行的动作类型

现在我们知道对于 failure 句柄来说所必须的东西了，我们现在建立一些简单的错误检验：

```
failure: function(f,a){

    if (a.failureType === Ext.form.Action.CONNECT_FAILURE)

    {

        Ext.Msg.alert('Failure', 'Server reported: '+a.response.status+'
'+a.response.statusText);

    }

    if (a.failureType === Ext.form.Action.SERVER_INVALID){

        Ext.Msg.alert('Warning', a.result.errormsg);

    }

}
```

通过检查 failure type，我们可以确定是否有数据库连接错误，甚至可以通过使用 result 属性来说明错误信息。

加载表单数据：

用户界面中的表单有三种基本功能：

- 输入数据来执行个别的操作——如谷歌搜索；
- 建立数据；
- 改变现有数据。

我们现在对最后一个功能比较感兴趣。为了完成它，我们需要知道如何从数据源（静态的或者数据库中的）中加载数据到用户界面。

加载静态数据：

我们可以通过执行代码来从某处获取数据并且在表单的某个字段中显示对应的值。下面的代码用来为字段设置值：

```
movie_form.getForm().findField('title').setValue('Dumb & Dumber');
```


当我们使用一个复杂的表格时，采用这个方法显得很困难。这就是我们采用 AJAX 来加载数据的原因。服务器端应该会像在 **combo box** 中加载数据一样工作。

```
<?php

// connection to database goes here

$result = mysql_query('SELECT * FROM movies WHERE id = '.$_
REQUEST['id']);

If (mysql_num_rows($result) > 0) {

    $obj = mysql_fetch_object($result);

    Echo '{success: true, data:'.json_encode($obj).'}';

}else{

    Echo '{success: false}';

}

?>
```

这样会返回一个包含 **success** 标志位的 **JSON** 对象。并且会传回一个运用于表单字段的数据对象，如下所示：

```
{

    success: true,

    data:{

        "id":"1",

        "title":"Office Space",

        "director":"Mike Judge",

        "released":"1999-02-19",

        "genre":"1",

        "tagline":"Work Sucks",

        "coverthumb":"84m.jpg",

        "price":"19.95",

        "available":"1"

    }

}
```

```
}
```

我们还需要用 `load` 句柄来触发它：

```
movie_form.getForm().load({  
  
    url:'data/movie.php',  
  
    params:{  
  
        id: 1  
  
    }  
  
});
```

只需要提供 `url` 和 `params` 配置就可以完成。`params` 配置代表了把什么发送到了服务器端。可以通过 `post` 和 `get` 两种方式，默认为 `post`。

引用对象或者配置组件：

通过前面这几张的学习，我们使用了越来越多的配置对象来建立 `ExtJs` 的组件，而不是通过实例化它们。让我们比较一下这两种方法。

实例化对象：

```
var test = new Ext.form.TextField({  
  
    fieldLabel: 'Title',  
  
    name: 'title',  
  
    allowBlank: false  
  
});
```

像这样，我们建立了一个组件并为之分配了内存，虽然它还没有出现在屏幕上。你的最终用户可能并不会使用这个特定的文本字段。但是，如果用户需要的话，它会显示的很快。

配置组件（用配置对象的方式来生成组件）

```
{  
  
    xtype: 'textfield',  
  
    fieldLabel: 'Title',  
  
    name: 'title',  
  
    allowBlank: false
```

```
}
```

通过采用配置对象，我们必须说明在用户使用字段时发生的一切，之前内存是不会被分配给该字段的。内存只会在用户需要它的时候被占用。所以，只有当用户点击或者采取某种交互后这个字段才会被渲染，所以初始化显示会慢一些。这样做还有其他好处。其中一个好处就是可以通过“**over the wire**”的方式来配置，“**over the wire**”指的是是服务器端产生配置来在客户端建立组件。

总结：

我们介绍了一个最典型的 **web** 引用——表单——并且在其内部诸如了强大的 **ExtJs** 的特性，讲理了一个灵活的强大的用户界面。本章中的表单可以校验用户输入，从数据库加载数据并且向服务器发送数据。凭借本章告诉我们的方法，我们可以建立表单来完成简单的文本搜索或者完成复杂的数据校验。

第四章 按钮、菜单和工具栏（1）

那些每个程序中的幕后英雄都是最简单的东西，如按钮、菜单和工具栏。在这章里，我们将学习如何把它们添加到自己的应用里。

我们的示例将会包含一些不同种类的按钮，有的有菜单，有的没有菜单。一个按钮外观上看可以是一个图标或者一段文本又或者两者兼而有之。工具栏里可以包含一些机械元素，如空格和分隔符，用来组织工具栏中的按钮。

我们还会介绍如何让这些元素对用户的交互做出反应。

适应各种场合的 **toolbar**（工具栏）：

几乎所有的 Ext 组件——面板、窗口、表格都可以容纳一个顶部或者底部的 **toolbar**。Ext 使得 **toolbar** 在渲染时添加到任何 DOM 元素中。**toolbar** 是一个非常灵活有用的组件，可以在任何应用中使用。

- **Ext.Toolbar**：按钮的主要容器；
- **Ext.Button**：建立一个按钮并提供交互功能；
- **Ext.menu**：一个菜单。

Toolbars(工具栏)：

我们的第一个工具栏将独立渲染在页面中。我们将分别添加一些主要的按钮类型，然后做分别的研究：

- **Button——tbbutton**：这是我们都熟悉的默认的按钮类型；
- **Split Button——tbsplit**：split button 中包含了一个可选菜单并提供默认的按钮动作。这种按钮被用在为按钮添加多个可选项的时候。当然，显示给用户的是最常用的默认选项；
- **Menu——tbbutton+menu**：Menu（菜单）其实就是一个用菜单配置项来配置的按钮。

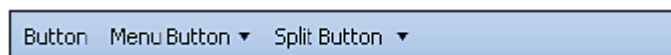
```
Ext.onReady(function(){  
  
    new Ext.Toolbar({  
  
        renderTo: document.body,  
  
        items: [{  
  
            xtype: 'tbbutton',  
  
            text: 'Button'
```

```

    }, {
        xtype: 'tbbutton',
        text: 'Menu Button',
        menu: [{
            text: 'Better'
        }, {
            text: 'Good'
        }, {
            text: 'Best'
        }]
    }, {
        xtype: 'tbsplit',
        text: 'Split Button',
        menu: [{
            text: 'Item One'
        }, {
            text: 'Item Two'
        }, {
            text: 'Item Three'
        }]
    }
];

});

```



像之前一样，所有的东西都包含在 `onReady` 中。`items` 配置项包含了所有的工具栏元素——我之所以说元素而不说按钮是因为工具栏可以包含很多不同种类的 `Ext` 组件包括表单字段——我们将在本章的后部介绍。

N: 所有元素的默认 `xtype` 是 `tbutton`。我们可以不去配置 `xtype`，如果被配置的元素是 `tbutton` 的话。但是我喜欢加上 `xtype` 以方便以后的跟踪。

按钮：

建立一个按钮十分简单。主要的配置项就是先是在按钮上的文本。我们还会添加 `icon`(图标)来让一切更生动。

这里展示一个 `stripped-down` 按钮：

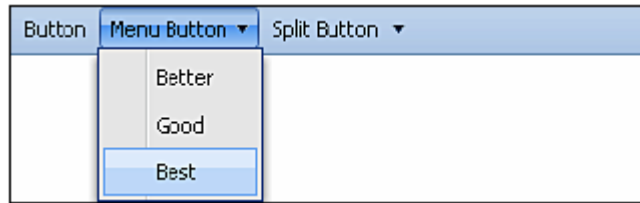
```
{  
  
    xtype: 'tbutton',  
  
    text: 'Button'  
  
}
```



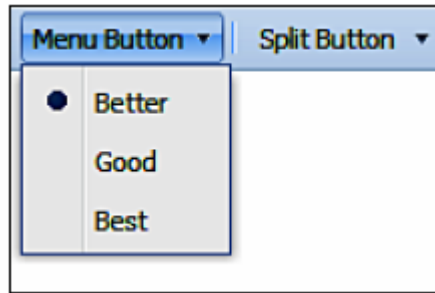
菜单（Menu）：

菜单其实就是一个包含菜单配置项的按钮。菜单的 `items`（项目）和 `buttons` 的原理一样。他们可以有图标、样式表，以及句柄。菜单的所有 `items` 可以组合起来形成一些列可选择的按钮，首先我们创建一个标准的菜单。

```
{  
  
    xtype: 'tbutton',  
  
    text: 'Button',  
  
    menu: [{  
  
        text: 'Better'  
  
    }, {  
  
        text: 'Good'  
  
    }, {  
  
        text: 'Best'  
  
    }  
    ]  
  
}
```



如果我们想给菜单分组，即让 Better、Good 和 Best 分为一组（如同 radio 单选按钮一样，每次只有一个项目被 check（选中）），我们需要为每个项目添加一个 group 配置项，并设置 checked 属性为 true 或者 false。这两个属性缺一不可！



Split button(分割按钮):

split button 听起来很复杂，其实就是按钮和菜单的结合。在使用它的时候只要在 menu 配置项目中添加按钮的属性就可以了。点击按钮左边部分可以触发按钮本身的事件，点击按钮右边的部分（有个倒三角）可以展开菜单。

```
{
    xtype: 'tbsplit',
    text: 'Split Button',
    menu: [{
        text: 'Item One'
    }, {
        text: 'Item Two'
    }, {
        text: 'Item Three'
    }]
}
```

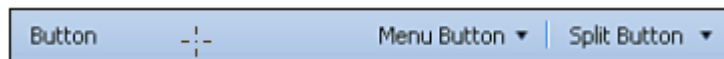


工具栏项目的排列、分割和间隔：

默认的，每一个工具栏中的元素都和放置在最左端。如果我们想把所有工具栏上的按钮放在最右端，我们需要添加一个 `fill` 作为工具栏的第一个元素。如果我们还想让表单元素分局左右两侧，我们同样也需要一个 `fill`：

```
{
    xtype: 'tbfill'
}
```

在任何你想要添加空白的地方放置一个 `fill`，就会让元素置于最左端或者最右端，如下所示：



我们还有元素可以提供空白和垂直分割线，如同在 `Menu Button` 和 `Split Button` 中的一样。

空白（`spacer`）会添加几个像素的空白，可以用来分割按钮，或者将其它元素设置得离按钮的边界远些：

```
{
    xtype: 'tbspacer'
}
```

分割线（`divider`）可以以同样的方式加在 `toolbar` 中：

```
{
    xtype: 'tbseparator'
}
```


第四章 按钮、菜单和工具栏（2）

Shortcuts（快捷方式）：

Ext 有很多 shortcuts，使得编码变的更快了。shortcuts 就是一个或者连个字符组成的用来做显示控制的配置对象。如下，十一哥标准的工具栏 filler(就是之前的 fill) 配置：

```
{  
  
    xtype: 'tbfill'  
  
}
```

它的 shortcuts 就是 “->”。

不是所有的快捷方式都能在文档中查到，你可以参看源码，找到你需要的。这里列出几种常用的 shortcuts：

组件	Shortcut	说明
Fill	“->”	用来将其后的组件推到工具栏最右侧
Separator	“-”或“separator”	一个用来分割左右的垂直竖杠
Spacer	“ ”	用来做分割元素的空白，每个 2 像素的宽度，可以通过修改 ytb-spacer CSS 类来改变其大小
TextItem	‘你需要的文本’	直接添加文本或者 html 到工具栏，左右用逗号分割

图标按钮：

标准的图标按钮就是你在文本编辑器里看到的使字体加粗或者倾斜的那些按钮。建立图标按钮需要两步——定义被用作图标的图片和适当的 CSS 类。

```
{  
  
    xtype: 'tbbutton',  
  
    cls: 'x-btn-icon',  
  
    icon: 'images/bomb.png'  
  
}
```



可以让图标和文字共存，只要调整 CSS 类并且让添加 text 配置项：

```
{  
  
    xtype: 'tbbutton',
```

```

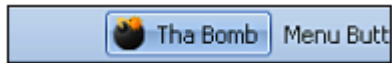
        cls: 'x-btn-text-icon',

        icon: 'images/bomb.png',

        text: 'Tha Bomb'

    }

```



按钮操作函数(handler)——点击我！

按钮不仅需要好看而且还要能完成和用户的交互。这就是操作函数(handler)为什么会被引入进来。一个 handler 是一个在按钮被单击后调用的函数。

handler 配置项就是我们加入自己函数的地方：

```

{

    xtype: 'tbutton',

    text: 'Button',

    handler: function(){

        Ext.Msg.alert('Boo', 'Here I am!');

    }

}

```

这段代码会在按钮单击后弹出警告窗口。有些时候，我们需要在单击后改变按钮的状态，所以每个按钮的 handler 都会传递按钮本身作为函数的参数。第一个参数就是对触发事件的组件的引用！

```

{

    xtype: 'tbutton',

    text: 'Button',

    handler: function(f){

        f.disable();

    }

}

```

我们可以利用这个对按钮本身的引用去访问按钮的属性和方法。在本例里我们调用了 disable 方法来让按钮变灰且不可点击。

我们可以做的更多，让我们尝试些更有用的东西。

在点击菜单项时加载内容：

让我们以按钮单击为基础做一些更有意义的事情。在这个例子里，我们将为每个菜单项添加配置项来决定加载那个文件的内容到网页中：

```
{
    xtype: 'tbsplit',
    text: 'Help',
    menu: [{
        text: 'Genre',
        helpfile: 'genre',
        handler: Movies.showHelp
    },{
        text: 'Director',
        helpfile: 'director',
        handler: Movies.showHelp
    },{
        text: 'Title',
        helpfile: 'title',
        handler: Movies.showHelp
    }]
}
```

注意 **helpfile** 配置项。我们使用这个配置项来为每个菜单项存储一个唯一的变量。这样做是可行的，因为配置属性可以是任意设置。这里我们采用一个配置属性作为一个变量来存储我们需要加载的文件名称。

我们还要做的一个新的工作就是建立一些函数来处理按钮单击事件，这些函数都被组织在叫 **Movies** 的类中：

```
var Movies = function() {
    return {
```

```

showHelp : function(btn){

    var helpbody = Ext.get('helpbody');

    if (!helpbody) {

        Ext.DomHelper.append(Ext.getBody(),
{tag:'div',id:'helpbody'});

    }

    Movies.doLoad(btn.helpfile);

},

doLoad : function(file){

    Ext.get('helpbody').load({

        url: 'html/' + file + '.txt'

    });

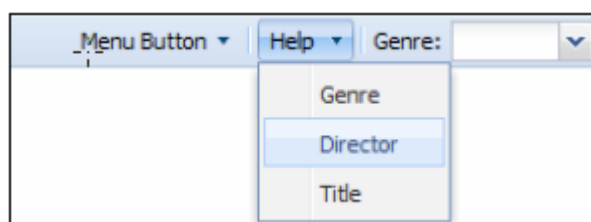
}

};

}());

```

我不想谈及过多的和这个类相关的细节，重要的是那些处理菜单项单击事件的函数。这个类会把文件内容通过 **AJAX** 请求加载到页面中——哪个文件被加载是和相应的选项相关联的。所以，在你把这个类放到你的页面中后，你可以通过单击不同的菜单项来加载不同的帮助文件。



接下来，我们将使用文本框实现同样的功能。

工具栏中的表单字段：

和大多数 Ext 中的组件一样，**toolbar** 里可以包含几乎所有的 Ext 组件。当然，表单字段和 **combobox** 也是在 **toolbar** 中相当有用的元素。

```

{

    xtype: 'textfield'

```

```
}
```

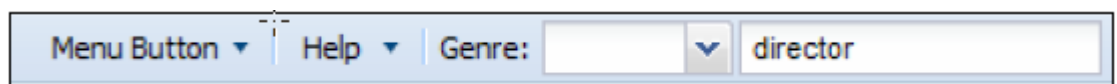
我们像在上章一样建立表单字段，我们需要把表单字段添加到 `items` 的数组中，这样就把它放到了 `toolbar` 里。

现在让我们的表单字段做些有意义的事情，提供帮助菜单一样的功能，但是通过一种更为灵活的途径。

```
{  
  
    xtype: 'textfield',  
  
    listeners: {  
  
        specialkey: Movies.doSearch  
  
    }  
  
}
```

这里的监听器 `listener` 直接放到了表单字段的配置对象里。这里我们使用 `specialkey` listener，在前面的章节中使用过，用来监听诸如回车或者 `Delete` 按键等键盘按键的按下。`handler` 函数需要加在 `Movies` 类中：

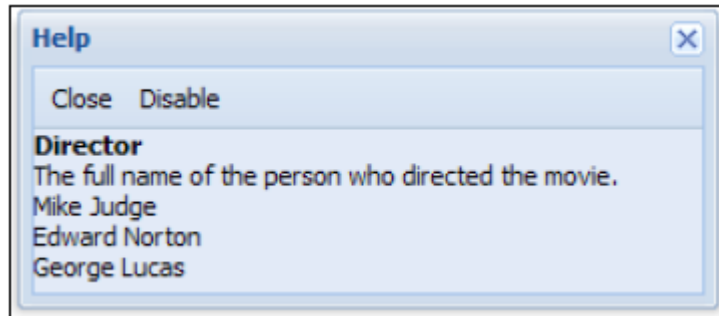
```
doSearch : function(frm,evt){  
  
    if (evt.getKey() == evt.ENTER) {  
  
        Movies.doLoad(frm.getValue());  
  
    }  
  
}
```



这样我们就实现了通过输入名字来动态加载文件的功能。

窗口、表格、面板中的工具栏：

所有的我们之前接触的工具栏都包含 `items` 配置项。如果我们想把这些工具栏放在其他的 Ext 组件中，如 `panel`（面板）或者窗口（`window`）里，我们可以把工具栏中的 `items` 配置项提取出来放在两个实现准备好的、存在于和 `panel` 类似的控件里的两个容器内。



这里所说的和 `panel` 类似的组件就是 `window` 和 `grid`(表格)等，而上面所说的两个容器，就是它们提供的顶部和底部工具栏：

- `tbar`：顶部工具栏
- `bbar`：底部工具栏

如果我们想在 `window` 的最上方加入工具栏，我们可以在 `tbar` 里加入一系列的
工具栏元素：

```
new Ext.Window({  
    title: 'Help',  
    id: 'helpwin',  
    width: 300,  
    height: 300,  
    tbar: [{  
        text: 'Close',  
        handler: function(){  
            Ext.getCmp('helpwin').close();  
        }  
    }, {  
        text: 'Disable',  
        handler: function(t){  
            t.disable();  
        }  
    }],  
    autoLoad: 'html/' + btn.helpfile + '.txt'
```

```
}).show();
```

Ext 还有自定义的专门为分页表格准备的工具栏。我们将在本书的 **grid**（表格）一章进行着重讨论。

总结：

在本章中，我们有机会采用不同方式来建立工具栏元素，包括配置对象和 **shortcut**。工具栏有很多的选项，可以让你建立从简单的按钮工具栏到复杂的包含诸如按钮、菜单、表单字段的各式各样的工具栏。通过内置操作函数（**handler**）我们可以很容易的完成同按钮、菜单、表单字段的交互。

使用grid显示数据（1）

毫无疑问，grid 是 Ext 中使用最广泛的组件之一。我们需要以一种易于理解的方式为最终用户展现我们的数据，电子表格就是这种方式之一，而且相当完美——应为其做到了，并且让这一观念深入人心。Ext 接受了该观点，并使得它更加灵活而且奇妙。

在这一章中我们会：






- 使用 GridPanel 来在用户友好的表格中展示结构化的数据；
- 从服务器或数据库加载数据显示在表格中；
- 使用表格提供的事件来控制表格的功能和显示；
- 在表格中采用高级的数据格式化技术；
- 建立分页表格。

我们将说明如何定义表格的行和列，但是更为重要的是，我们将学习如何使表格更为美观。我们可以添加自定义渲染的单元格，并且在其中添加图片，我们还可以根据数据值来改变显示样式。

到底什么是表格？

Ext 的表格和电子表格相似，包含 2 个重要的部分：

- 行
- 列

Movie Database				
Title ▼	Released	Genre	Price	
 The Big Lebowski <i>Joel Coen</i> The "Dude"	03/06/1998	Comedy	\$21.95	
 Super Troopers <i>Jay Chandrasekhar</i> Altered State Police	02/15/2002	Comedy	\$14.95	
 Office Space <i>Mike Judge</i> Work Sucks	02/19/1999	Comedy	\$19.95	
 Fight Club <i>David Fincher</i> How much can you know about yourself...	10/15/1999	Action	\$19.95	
 American Beauty <i>Sam Mendes</i> Lust, Life, and Loss	10/01/1999	Drama	\$19.95	

在这里，我们的列是：Title, Released, Genre, 和 Price。每一行包含像 The Big Lebowski, Super Troopers 之类的电影。这些行都是我们的数据；表格中的每一行代表数据中的一条记录。

在 GridPanel 中显示结构化的数据：

在表格中显示数据需要两个 Ext 组件：

- store——像数据库一样，追踪要显示的数据；
- 表格面板——提供展现 store 中数据的方式。

在我们开始来实现这些之前，让我们先来看看将被我们使用到的术语，因为这些术语可能会把我们搞糊涂：

- Columns（列）：它引用整个数据列，包含显示数据有关的信息，包括列的 head（列标题），在 ExtJS 中，这些都是 Column Model（列模型）的一部分；
- Fields（字段）：它也引用了整个数据列，但是它引用的是实际的数值。在 ExtJS 里，它被在 reader（Ext 读取数据的组件）中，用来加载数据。

建立 data store（数据容器）：

我们需要做的第一件事就是建立我们的数据，它将会被放到 **data store** 之中。**data store** 有不同的类型，可以让我们读取不同形式的数 据（XML，JSON 等等），并且可以在不同的 **Ext** 控件中来完成读取数据的过程。不管数据是 **JSON**，**XML** 还是数组，甚至是我们自己自定义的数据类型，我们都可以通过相同的方式访问，这要感谢 **data store**。

下面是一些 **Ext** 中默认的 **data store** 类型：

- Simple (Array)（数组）
- XML
- JSON

A custom data store could be created to read data that does not fit into these categories.

可以创建自定义的 **data store** 来读取其他类型的数据。**Ext** 论坛提供了一些用户贡献的 **data store**，例如 CSV 和 ColdFusion 形式的数据。

在 **data store** 中添加数据：

初次尝试，我们将建立一个以本地数组为数据源的表格。接下来我们使用到的数据来自一个小的电影数据库，其中包含了一些我最喜爱的电影，这与这章后面用到的来自真实数据库的数据很相似。

The data store needs two things: the data itself, and a description of the data—or what could be thought of as the fields.

data store 需要两个东西：数据本身和数据描述——如同字段以及其它你能想到的类似的东西。

reader（读取器）读取器将被用来读取来自数组的数据，在其中我们会定义数组数据的字段名（**field**）。读取器相当于一个翻译，它将数据字符串翻译为一行一行的数据来供 **Ext** 使用。

接下来的代码应该被放到 **Ext** 的 **OnReady** 函数内部：

```
var store = new Ext.data.Store({  
  
    data: [  
  
        [  
  
            1,  
  
            "Office Space",
```

```

        "Mike Judge",
        "1999-02-19",
        1,
        "Work Sucks",
        "19.95",
        1
    ],[
        3,
        "Super Troopers",
        "Jay Chandrasekhar",
        "2002-02-15",
        1,
        "Altered State Police",
        "14.95",
        1
    ]
    //...more rows of data removed for readability...//
],
reader: new Ext.data.ArrayReader({id:'id'}, [
    'id',
    'title',
    'director',
    {name: 'released', type: 'date', dateFormat: 'Y-m-d'},
    'genre',
    'tagline',
    'price',
    'available'

```

```
    ]  
  });
```

如果我们在浏览器中查看这段代码，我们不会看到任何东西——因为 **data store** 只是用来加载并跟踪数据的一种方式。现在浏览器的内存中已经有我们的数据了。现在我们只需要决定怎样将它显示给用户。

为 **data store** 定义数据：

reader 需要知道从数据存储器中读取哪些字段，所以我们需要定义这些字段。

字段使用由对象组成的数组来定义——如果数据被挨个读取，那么只需要指定这个字段的名称就可以了。在这个例子中，除了一个字段，其他的都可以通过简单的命名来定义。

举个例子，标题字段可以用如下的对象来定义：

```
{name: 'title'}
```

然而，在我们的用例中，因为我们仅仅把这个字段以一个字符串的形式读取，我们可以简单的定义这个字段的名称以减少一些输入：

```
'title'
```

released 字段很特殊，是因为我们需要适当处理它的数据，设置它的 **type** 为 **data**。对于每种字段格式类型（**type**），都有一些选项来更明确的定义它的格式。使用日期类型（**date type**）时，还需要定义 **dateFormat**（时间格式化）属性，它是一个字符串（本例中是 **'Y-m-d'**）。如果你用过 **PHP** 的话，时间格式化字符串看起来很熟悉，因为 **Ext** 使用了与 **PHP** 相似的时间格式化字符串。

```
{name: 'released', type: 'date', dateFormat: 'Y-m-d'}
```

指定的数据类型：

ExtJs 有许多途径来适当读取特定数据类型，它们都列在下面：

字段类型	说明	信息
string	字符串数据	
int	数字	使用 JS 的 <code>parseInt</code> 函数
float	浮点型数字	使用 JS 的 <code>parseFloat</code> 函数
boolean	true/false	
date	日期数据	需要 <code>dateFormat</code> 配置

以下是一些我经常使用的数据类型：

字段类型	说明	用法
date	包含信息的数据	<p>你需要定义 <code>dateFormat</code>，它告诉 Ext 如何把字符串转化为日期。</p> <p>Y-m-d 代表了“4 字符的年-数字月-数字日”</p> <p>Ext 中的 <code>dateFormat</code> 和 PHP 一样，一下连接有对格式化日期的详细说明：</p> <p>http://www.php.net/date</p>
int	数字数据	将数值视为整数——int 数据常常可以用来做比较，进而完成一些事情，如对两列数据求和。
boolean 或者 bool	True/False	注意布尔值展现的各种形式，你可以把一个字符串强制转化为布尔值，也可以用 0、1 分别转化为布尔值。

如果我们仅仅用刚才的 `reader` 显示数据的话，最终会是这样的结果：

Movie Database			
Released	Price	Avail	
Fri Feb 19 1999 00:00:00 GMT-0700 (Mountain Standard Time)	19.95	true	
Fri Feb 15 2002 00:00:00 GMT-0700 (Mountain Standard Time)	14.95	true	
Fri Oct 01 1999 00:00:00 GMT-0600 (Mountain Daylight Time)	19.95	true	
Fri Mar 06 1998 00:00:00 GMT-0700 (Mountain Standard Time)	21.9	true	
Fri Oct 15 1999 00:00:00 GMT-0600 (Mountain Daylight Time)	19.95	true	

看上去很丑陋，近乎让人崩溃：

- `released` 列的 `type` 设置为 `date`，也被从字符串转化日期，但是其展现形式却是标准的且丑陋的 JS 的日期格式——幸运的是，Ext 可以让它看上去更漂亮；
- `price` 列被设置为浮点数，但是这里并不需要精确到小数；
- `Avail` 列被解析成布尔值，即便原始数据不是一个 `true` 值。

这就是为什么要指定数据类型并且添加一些特殊的选项的原因。

显示 GridPanel（表格面板）：

将所有的东西集成到一起的就是 `GridPanel`，它专注于将数据放置到行跟列中，连同列标题，并将所有的东西做简洁的封装。

对任何人来说，只是将电影的数据存放于电脑的内存中并不是够，我们需要在表格中展现它：

1. 把 data store 放在 GridPanel 的代码中：

```
Ext.onReady(function(){  
  
    // add your data store here  
  
    var grid = new Ext.grid.GridPanel({  
  
        renderTo: document.body,  
  
        frame:true,  
  
        title: 'Movie Database',  
  
        height:200,  
  
        width:500,  
  
        store: store,  
  
        columns: [  
  
            {header: "Title", dataIndex: 'title'},  
  
            {header: "Director", dataIndex: 'director'},  
  
            {header: "Released", dataIndex: 'released',  
              renderer: Ext.util.Format.dateRenderer('m/d/Y')},  
  
            {header: "Genre", dataIndex: 'genre'},  
  
            {header: "Tagline", dataIndex: 'tagline'}  
  
        ]  
  
    });  
  
});
```

在浏览器中，你看到的将是这样：

Movie Database				
Title	Director	Released	Genre	Tagline
Office Space	Mike Judge	02/19/1999	1	Work Sucks
Super Troopers	Jay Chandrasekhar	02/15/2002	1	Altered State Police
American Beauty	Sam Mendes	10/01/1999	2	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	1	The "Dude"
Fight Club	David Fincher	10/15/1999	3	How much can you

究竟是怎么做到的呢？

我们的 data store 和一个决定 column（列）跟 column header（列标题）应该怎样被显示的（column model）列模型加载到了表格中。和先前在 reader 中定义的字段不同，由于现在已经被定义了，所以 reader 知道怎么去读数据。

配置 GridPanel:

The GridPanel is the widget that ties everything together: Ext.grid.GridPanel

GridPanel 是一个将一切捆绑到一起的控件，我们只需要一些基础的东西来建立它：

字段类型	说明	用法
renderTo	说明 GridPanel 显示的位置	它的值需要是一个有效的 DOM 对象，或者 DOM 元素的 id。接下来，我们将会把 GridPanel 直接传给其他控件，所以这个配置项将可以不做设置。
frame	为 GridPanel 加框	为 GridPanel 添加一个漂亮的边框，这项不是必须的，但是会使得 GridPanel 在渲染时更加漂亮。
height 和 width	定义 GridPanel 的大小	高度是必须被定义的，因为 grid 不会自己计算表格的高度。但是当我们在布局（layout）中添加 grid 时，这项就不需要了。
store	我们的数据	对包含数据的 data store 的引用。
columns	Column model（列模型）	定义 GridPanel 中列的数组。
stripeRows	设置行条纹	当该项为 true 时，行的颜色将交替变化。

grid panel 的基本的设置大概象这样：

```
var grid = new Ext.grid.GridPanel({  
    renderTo: Ext.getBody(),  
    frame:true,  
    title: 'Movie Database',  
    height:200,  
    width:500,  
    store: store,  
    columns: [ insert columns here ]  
});
```

我们几乎可以象读句子一样来读懂这段配置：

将我们的 **grid** 渲染到页面的 **body** 当中，为其加边框，给他一个 ‘Movie Database’ 的标题。它的高度为 200，宽度为 500；它将使用我们的 **store** 作为 **data store**，并且有指定的列模型。

The one reason why I love object-based configuration so much is that it is human readable.

我喜欢基于对象的配置方法的原因之一就是它的易读性。我们不在用去打开手册查找函数 **x** 的第三个参数是什么！只要简单的说“让高为 200 宽为 500”就可以了。

定义 **grid** 的列模型：

为了定义我们的表格列，我们需要先创建一个定义了这些列应该怎么显示、处理的对象数组。

```
columns: [  
    {header: 'Title', dataIndex: 'title'},  
    {header: 'Director', dataIndex: 'director'},  
    {header: 'Released', dataIndex: 'released'},  
    {header: 'Genre', dataIndex: 'genre'},  
    {header: 'Tagline', dataIndex: 'tagline'}  
]
```

column header（列标题）看上去像这样：

Title	Director	Released	Genre	Tagline
-------	----------	----------	-------	---------

列模型中的每个列有很多配置项，但是必须至少定义 **header** 和 **dataIndex**。**header** 用来说明列标题（column header）的内容，**dataIndex** 表示列中数据字段的名称——我们在建立 **reader** 的时候已经定义好了这些。

这里还有其它一些列模型的比较有用的配置选项：

选项	说明	用法
renderer	定义数据的显示形式	格式化该列的数据，使其变成我们需要的形式。所有类型的数据都可以被转化，我们将在接下来的几章里做进一步学习。
hidden	隐藏该列	布尔值，决定是否显示该列
width	定义列的宽度，单位像素	定义列的宽度，默认 100 像素，超出的部分被隐藏起来。
sortable	决定该是否对该列进行排序	布尔值，决定该列是否能够被排序。

第五章 使用 grid 显示数据（2）

使用 cell renderer（单元格渲染器）：

我们可以使用单元格渲染来干一些漂亮活。对于如何使得单元格变漂亮以及让它包含更多我们想要的东西来说，几乎没有任何阻碍。我们需要做的，只是定义好 ExtJS 为我们提供的单元格内置格式化函数。如 `usMoney`，或者建立我们自己的 `cell renderer`（单元格渲染器）。让我们先看下如何使用内置的 `cell renderer`，接下来再建立自己的 `cell renderer`。

利用内置的 cell renderer 格式化数据：

很多内置的格式化函数是用来满足通常的渲染需求的。其中我常常用到的是 `date renderer`：

```
renderer: Ext.util.Format.dateRenderer('m/d/Y')
```

还有一些 `renderer` 包含了一些常用的格式化功能，如货币（这里指的是格式化为货币符号），大写，小写。

这里给出一些人们经常要用到的 `renderer`：

Renderer	说明	用途
<code>dateRenderer</code>	格式化用来显示的日期	格式化该列日期为你喜欢的形式，所有形式的日期都可以被转化。
<code>uppercase</code> <code>lowercase</code>	大写和小写转化	转化字符串为完全大写或者小写。
<code>Capitalize</code>	美化文本	格式化文本，让它的大小写都正确。

建立可查询的 data store——自定义单元格渲染：

我们将重点聚焦到 `'genre'` 一列上，它里面的值都是数值，我们当前的需求是，能够通过查询在“表单”一章中的 `data store` 来找到 `genre` 数字对应的文本信息。

首先，我们需要往 `column model` 中添加一个 `renderer` 配置，告诉表格在单元格渲染的时候使用哪个函数：

```
{header: 'Genre', dataIndex: 'genre', renderer: genre_name}
```

现在，我们来写这个函数。这个函数传递了单元格中的值作为第一个参数。第二个参数是单元格对象本身，第三个参数是该表格的 **data store**——这两个我们都不需要，所以让我们先不去管它们。

```
function genre_name(val){  
  
    //genres 就是之前第三章中的那个 simpleStore  
  
    return genres.queryBy(function(rec){  
  
        if (rec.data.id == val){  
  
            return true;  
  
        }else{  
  
            return false;  
  
        }  
  
        //如果匹配，则返回相应的文本内容  
  
    }).itemAt(0).data.genre;  
  
}
```

renderer 函数传递了单元格当前的数值。这个数值可以被检验并且可以被修改——任何 **return** 回来的数值都会被渲染在单元格上。**queryBy** 的作用是从 **store** 中过滤数据，它接收一个函数来比较每一行的数据，当 **return** 为 **true** 时，它就使用匹配的行。

为了方便考量，我们现在提供这个函数的另一个版本，它和上面的函数功效相同，但是不如第一个易读：

```
function genre_name(val){  
  
    return genres.queryBy(function(rec){  
  
        return rec.data.id == val;  
  
    }).itemAt(0).data.genre;  
  
}
```

合并两列数据：

上面介绍的能够查询 **data store** 的 **renderder** 很有用。但是，更为经常使用的一个功能是，我们会合并两列为一个单元格，例如，相加两列数值，求平均，求差值，或者合并两列字符串。

以电影标题（title）列举例，我们可以把 tagline 和它合并，这样我们就不需要 tagline 这一列了。首先，我们需要在 column model 中把 tagline 这一列的 hidden 属性设置为 true，即隐藏该列。

```
{header: 'Tagline', dataIndex: 'tagline', hidden: true}
```

接下来配置 renderer 函数，让它来完成合并列的工作：

```
function title_tagline(val, x, store){  
  
    return '<b>'+val+'</b><br>'+store.data.tagline;  
  
}
```

在这个函数里，我们 title 加粗，用以区分 title 和 tagline（其实作者还在中间加了换行呢，哈哈）。你可以看到，HTML 标签在这里同样好用。最后，不要忘了把这个函数加到 column model 里啊：

```
{header: 'Title', dataIndex: 'title', renderer: title_tagline}
```

现在的 title 列如下所示：

Title	Director	Released	Genre	Price
Office Space Work Sucks	Mike Judge	02/19/1999	Comedy	\$19.95

生成 HTML 和图片：



让我们为每行添加图片来显示每个电影的封面（cover），我们之前发现可以在 cell 中使用 HTML 语言，那么现在我们可以通过添加标签，并且在 src 中指定图片路径来完成这一功能。

```
function cover_image(val){  
  
    return '<img src=images/'+val+'>';  
  
}
```

别忘了设置 column 的 renderer：

```
{header: 'Cover', dataIndex: 'coverthumb', renderer: cover_image}
```

现在的 grid 看起来是这样的：

Movie Database						
Cover	Title	Director	Released	Genre	Price	
	Office Space Work Sucks	Mike Judge	02/19/1999	Comedy	\$19.95	
	Super Troopers Altered State Police	Jay Chandrasekhar	02/15/2002	Comedy	\$14.95	

内置特性：

Ext 有很多好的内置特性，用来完成更好的电子表格界面。每一列都有一个内置的菜单，提供排序，显示/隐藏列的功能。

Movie Database				
Title	Director	Released	Genre	
Office Space		19/1999	1	
Super Troopers		15/2002	1	
American Beauty		01/1999	2	
The Big Lebowski				
Fight Club	David Fincher	10		

客户端排序：

除非指定 `grid` 为服务器端排序，否则 Ext 是可以在客户端对列进行排序。服务器端排序用在分页数据，或者数据格式不能被客户端排序的时候。客户端排序快捷、简单而且是内置的：

```
{header: 'Tagline', dataIndex: 'tagline', sortable: true}
```

我们也可以在 `grid` 渲染完毕后再实现排序功能；

```
var colmodel = grid.getColumnModel();
```

```
colmodel.getColumnById('tagline').sortable = true;
```

我们的列模型(column model)控制着列和列标题的显示。如果我们能从 grid 中获得 column model 的引用，那么我们就可以在渲染后改变列。我们通过使用 getColumnById 函数来获得某列，只需传递该列的 ID 到此函数。

隐藏/显示列：

利用列标题（column header）中的菜单，我们可以让某列显示或者隐藏。在配置里我们也可以对这一属性做设置，让列默认为隐藏的，如下所示：

```
{header: "Tagline", dataIndex: 'tagline', hidden: true}
```

我们经常在表格渲染完毕后执行列的隐藏，我们可以使用 Ext 提供的函数来实现这一令人兴奋的功能：

```
Var colmodel = grid.getColumnModel();
```

```
colmodel.setHidden(colmodel.getColumnIndexById('tagline'),true);
```

获取 column model 的引用可以让我们对列的显示加以改变。

列（column）的渲染：

我们可以通过拖拽列标题（column header）来实现列之间的重新排序。这都是 Ext 表格的内置功能。



Title	Director	Released	Price	Tagline
Office Space	Mike Judge	Fri Feb 19 1999 00:	19.95	Work Sucks
Super Troopers	Jay Chandrasekhar	Fri Feb 15 2002 00:	14.95	Altered State Police
American Beauty	Sam Mendes	Fri Oct 01 1999 00:	19.95	... Look Closer
The Big Lebowski	Joel Coen	Fri Mar 06 1998 00:	21.9	The "Dude"
Fight Club	David Fincher	Fri Oct 15 1999 00:	19.95	How much can you

所有的列都可以被任意的拖拽。以上显示的过程是把 price 拖到 title 和 director 列之间。

我们可以通过把 GridPanel 中的 enableColumnMove 属性设置为 false 来屏蔽列拖拽的功能。

```
enableColumnMove: false
```

列移动的事件——和 grid 中的其它事件——都可以被监测到并产生相应的响应。据个例子，我们可以监测列的移动然后弹出信息，告诉用户把列拖到了什么位置：

```

grid.getColumnModel().on('columnmoved',
    function(cm,oindex,nindex) {
        var title = 'You Moved '+cm.getColumnHeader(nindex);
        if (oindex > nindex){
            var dirmsg = (oindex-nindex)+' Column(s) to the Left';
        }else{
            var dirmsg = (nindex-oindex)+' Column(s) to the Right';
        }
        Ext.Msg.alert(title,dirmsg);
    }
);

```

很多其他的事件都可以用同样的方式监听到。grid、data store、column model 都有它们自己可以被监听到的事件，我们将在这章里学到更具体的内容。

在 grid 里显示服务器端的数据：

利用 Ext，我们可以把数据以各种方式添加到页面中。我们先前为表格添加本地数组数据。现在我们将要把数据从附加的文件和服务端中取回。

从 XML 文件中加载电影数据：

我们已经有了电影数据，但是当我想要添加一个新的电影时我需要编辑 JS 的数组。所以，何不把数据通过 XML 文件来存取呢？这样做会方便升级，XML 文件可以通过数据库查询或者自定义脚本生成。

我们先看一下 XML 文件的示例，看看它到底是怎样的：

```

<?xml version="1.0" encoding="UTF-8"?>

<dataset>

    <row>

        <id>1</id>

        <title>Office Space</title>

        <director>Mike Judge</director>

        <released>1999-02-19</released>

```

```

        <genre>1</genre>

        <tagline>Work Sucks</tagline>

        <coverthumb>84m.jpg</coverthumb>

        <price>19.95</price>

        <active>1</active>

    </row>

    <row>

        <id>3</id>

        <title>Super Troopers</title>

        <director>Jay Chandrasekhar</director>

        <released>2002-02-15</released>

        <genre>1</genre>

        <tagline>Altered State Police</tagline>

        <coverthumb>42m.jpg</coverthumb>

        <price>14.95</price>

        <active>1</active>

    </row>

    //...more rows of data removed for readability...//

</dataset>

```

我们需要改动数据的 **reader**，并且设置 XML 的位置来让 **data store** 知道从哪里获得数据。在把数据源从本地向远程转化时有四个基本的更改：

- **url** 配置项，指定数据的位置，它被用来替换本地数据读取时的 **data** 配置项；
- **reader** 由 **ArrayReader** 转化为 **XmlReader**，以此来处理数组格式到 XML 格式的转化；
- 通过设置 **record** 配置项，**XmlReader** 可以告知哪个元素包含了数据记录或者一行数据，；
- 需要调用 **load** 方法来加载数据，让 **data store** 把数据从文件中放到计算机的内存中。


```

var store = new Ext.data.Store({

    url: 'movies.xml',

    reader: new Ext.data.XmlReader({

        record:'row',

        id:'id'

    }, [

        'id',

        'coverthumb',

        'title',

        'director',

        {name: 'released', type: 'date', dateFormat: 'Y-m-d'},

        'genre',

        'tagline',

        {name: 'price', type: 'float'},

        {name: 'available', type: 'bool'}

    ])

});

store.load();

```

在做了这些改动后，我们可以看看 **grid** 是否还能工作——好像不需要我们在改动数据源和格式了。

注意从本地到远程数据的转化，以及从数组格式到 **XML** 格式的转化，我们只需要让 **data store** 加以改变即可。**Ext** 通过使用通用的 **data store** 来添加不同的 **reader** 扩展，从而读取各种格式的数据，这样，就可以把改变隔离化，除了 **data store**，不会影响其他代码。

从 JSON 文件中加载电影数据：

和 **XML** 一样，我们只需要改变 **reader** 并且建立一些配置项来完成所有的改动。

JSON 在介绍表单时提到过——我们的 **movies.json** 文件中的数据如下所示：

```

{

```

```
success:true,

  rows:[

    {

      "id":"1",

      "title":"Office Space",

      "director":"Mike Judge",

      "released":"1999-02-19",

      "genre":"1",

      "tagline":"Work Sucks",

      "coverthumb":"84m.jpg",

      "price":"19.95",

      "active":"1"

    },{

      "id":"3",

      "title":"Super Troopers",

      "director":"Jay Chandrasekhar",

      "released":"2002-02-15",

      "genre":"1",

      "tagline":"Altered State Police",

      "coverthumb":"42m.jpg",

      "price":"14.95",

      "active":"1"

    }

    //...more rows of data removed for readability...//

  ]

}
```

XML reader 和 JSON reader 的主要不同就在于 JSON 需要知道 root 元素的名字，它是用来包含对象数组的（array objects 也就是我们的数据）。所以，与 XML reader 中设置 record 配置不同的是，我们需要设置 root 配置项：

```
var store = new Ext.data.Store({

    url: 'movies.json',

    reader: new Ext.data.JsonReader({

        root: 'rows',

        id: 'id'

    }, [

        'id',

        'coverthumb',

        'title',

        'director',

        {name: 'released', type: 'date', dateFormat: 'Y-m-d'},

        'genre',

        'tagline',

        {name: 'price', type: 'float'},

        {name: 'available', type: 'bool'}

    ])

});

store.load();
```

这个表格和之前的数组、XML 表格显示效果一样，功能也一样。

N: JSON 格式源自 JavaScript，演变成为了一种 data store 能够读取的最为快捷的数据格式。我们的 grid 会因为采用 JSON 而在显示的时候变得更快。

第五章 使用 grid 显示数据（3）

利用 PHP 从数据库加载数据：

GridPanel 的设置保持不变。和从静态文件中读取 JSON 数据不同的是，我们需要利用 PHP 来从数据库中获取数据，然后格式化为 Ext 可以读懂的 JSON：

```
<?php

// connect to database

$sql = "SELECT * FROM movies";

$arr = array();

if (!$rs = mysql_query($sql)) {

    Echo '{success:false}';

}else{

    while($obj = mysql_fetch_object($rs)){

        $arr[] = $obj;

    }

    Echo '{success:true,rows:'.json_encode($arr).'}';

}

?>
```

N：这里的 PHP 代码的旨在用最简单的语句来完成任务。在真正的产品中，你可能要考虑 SQL 注入攻击、错误监测和用户验证——本例的代码并没有设计这么多。

编写 grid：

之前我们的代码都着眼于如何让 grid 显示内容，但是我们会要求 grid 能够对用户的输入做相应。一种最常用的交互方式就是选中或者移动一行或者几行数据。ExtJS 引用了这些交互，用 selection model（选择模型）来处理它。让我们来看看如何建立这个 selection model。

处理单元格和行的选择：

Ext 的 grid 提供了监听用户和表格的行、单元格以及列交互事件的工具，那就是 selection model（选择模型）。selection model 用来决定行、列或者单元格怎样被选中以及允许多少项可以被同时选中。它允许我们为选择事件添加一些 listener，同时给我们提供了一种查询选择行的方法。

列出一些 selection model:

- **CellSelectionModel**: 让用户只能选择单一的单元格;
- **RowSelectionModel**: 让用户选择完整的行;
- **ColumnSelectionModel**: 让用户选择完整的列;
- **CheckBoxSelectionModel**: 让用户使用 checkbox (复选框) 来做行的选择;

我们需要根据自己项目的需求来选择 selection model, 对于我们的电影数据, 我们需要使用 row selection model, 这是一种最普遍的 selection model。

在 GridPanel 中使用 cm 配置项来定义 selection model。

```
sm: new Ext.grid.RowSelectionModel({  
  
    singleSelect: true  
  
})
```

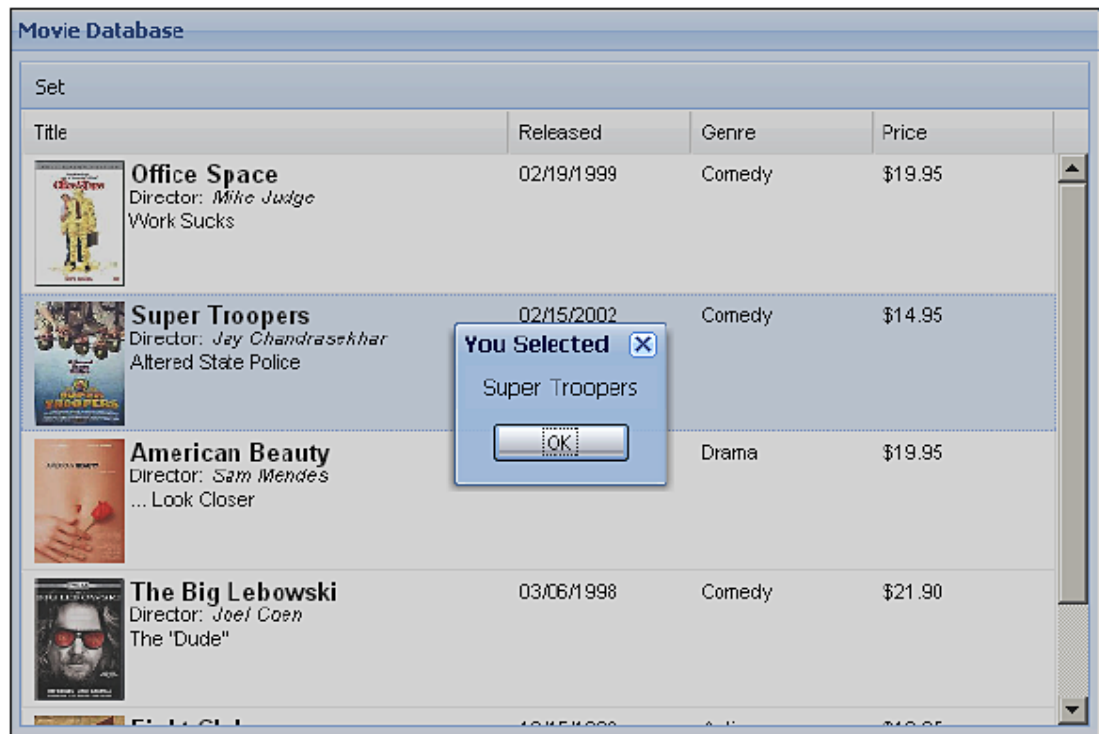
我们还需要让 singleSelect 配置项为 true, 这样就可以阻止用户同时选择多行。

监听 selection model 的选择事件:

listener (监听器) 可以在任何地方引入, 完全取决于需要的交互。开始, 我们将 listener 放在 column model 中, 因为我们需要监听列的动作。

但在这里, 我们将 listener 加在 selection model 中, 因为我们想知道用户在什么时候选择了一个电影。

```
sm: new Ext.grid.RowSelectionModel({  
  
    singleSelect: true,  
  
    listeners: {  
  
        rowselect: {  
  
            fn: function(sm,index,record) {  
  
                .Msg.alert('You Selected',record.data.title);  
  
            }  
  
        }  
  
    }  
  
})
```



选择一行会弹出一个 alert 提示框，让我们看看到底发生了什么：

- 我们为 rowselect 事件建立了一个 listener，它用来等待行被选中，从而执行相应的函数；
- 向 function 传递 selection model，选中行的序号（第一行为 0），以及选中行的数据记录；
- 利用 function 中接收的数据记录来获取电影的题目然后在信息框中显示。

用代码操控 grid（以及它的数据）：

操控 grid 和它的数据的函数有很多，可以绑定到其他的 Ext 控件上来完美地满足很多功能需求。

在点击按钮时改变 grid:

这里，我们准备添加一个顶部工具栏，它将包含一个按钮来弹出提示窗口，用来编辑电影的标题。

```
tbar: [{
    text: 'Change Title',
    handler: function(){
        var sm = grid.getSelectionModel();
        if (sm.hasSelection()){
```

```

var sel = sm.getSelected();

Ext.Msg.show({

    title: 'Change Title',

    prompt: true,

    buttons: Ext.MessageBox.OKCANCEL,

    value: sel.data.title,

    fn: function(btn,text){

        if (btn == 'ok'){

            sel.set('title', text);

        }

    }

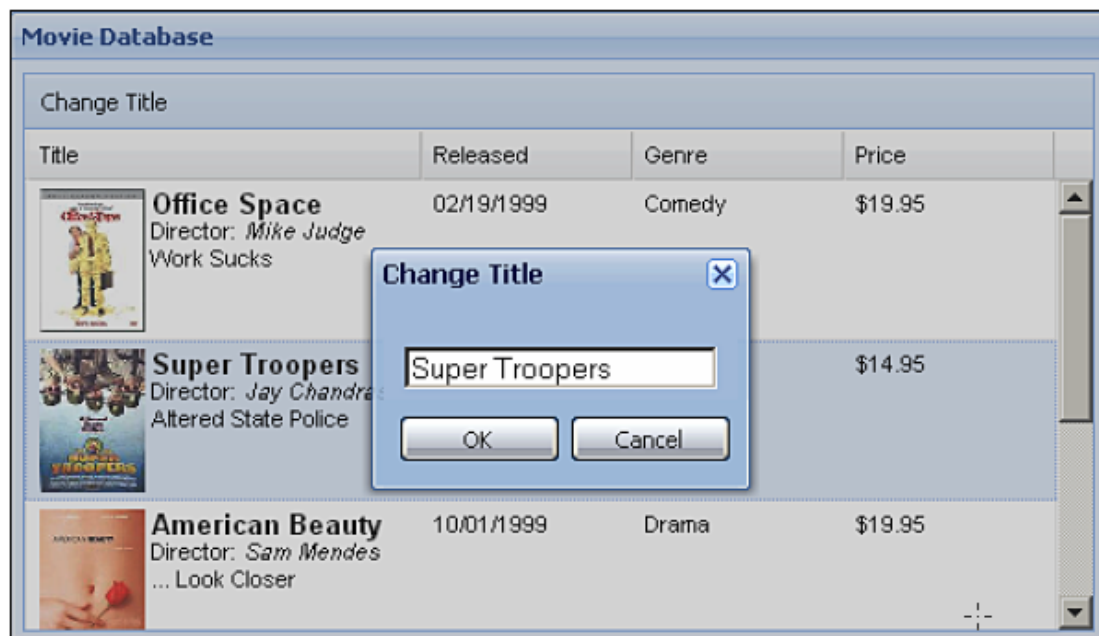
});

}

}

}

```



我们只需要改变 **data store** 即可。服务器上的数据库还是保持不变的，服务器也不知道到底有改变没有。我们的任务就是通过 **AJAX** 请求或者其它方式来让服务器得知数据的改变。

让我们快速地下这里发生了什么：

- `sm`: 从 `grid` 中获得的 `selection model`;
- `sel`: 利用 `selection model` 来获得被选中的行;
- `sel.data`: 使用被选中项的数据对象, 我们可以获得它的数据。

这个基本的方法可以用来建立很多有趣的用户交互, 我们唯一的限制就是一天只有 24 小时, 并且我们还需要睡觉。(作者够幽默)

高级 `grid` 格式化:

因为我们都想建立一些用户与 `grid` 之间的交互, 那就让我们先添加更多的按钮来做更好玩的事情。

这里有一个可以添加到工具栏上的按钮, 它能偶让我们隐藏以及显示列。我们还可以基于列的可见与否动态改变按钮上的文字。

```
{
    text: 'Hide Price',
    handler: function(btn){
        var cm = grid.getColumnModel();
        var pi = cm.getColumnIndexById('price');
        if (cm.isHidden(pi)){
            cm.setHidden(pi,false);
            btn.setText('Hide Price');
        }else{
            cm.setHidden(pi,true);
            btn.setText('Show Price');
        }
        btn.render();
    }
}
```

我们使用了一个新的方法——`getColumnIndexById`, 你可以想象——获得列的序号 (从 0 到比列总数少 1), 它可以被用来得知一个列和其它列的关系。在我们的 `grid` 代码中, `price` 列是第四列, 所以他的 `index` (序号) 是 3, 因为是从 0 开始计数的。

分页表格：

分页需要服务器端的元素（脚本）类按页分解数据。PHP 对此很合适，并且它的代码很易懂，也很容易被翻译为其它语言。所以我们使用 PHP 来做示例。

当一个分页表格被分页时，它想后台传递了 2 个参数：**start**、**limit**。它们被用来寻找记录的子集。我们的脚本可以读取这些参数，让后利用他们来做数据库查询。

这里给出典型的 PHP 脚本，它可以处理分页。我们将文件命名为 **movies-paging.php**。

```
<?php

// connect to database

$start = ($_REQUEST['start'] != "") ? $_REQUEST['start'] : 0;

$limit = ($_REQUEST['limit'] != "") ? $_REQUEST['limit'] : 3;

$count_sql = "SELECT * FROM movies";

$sql = $count_sql . " LIMIT ".$start.", ".$limit;

$arr = array();

if (!$rs = mysql_query($sql)) {

    Echo '{success:false}';

}else{

    $rs_count = mysql_query($count_sql);

    $results = mysql_num_rows($rs_count);

    while($obj = mysql_fetch_object($rs)){

        $arr[] = $obj;




    }

    Echo '{success:true,results:'. $results.',rows:'.json_encode($arr).'}';

}

?>
```

这个 PHP 脚本会处理服务器端的分页。所以我们现在只需要为 **grid** 添加分页工具栏——这十分简单！

Movie Database				
Title	Released	Genre	Price	
 American Beauty Director: <i>Sam Mendes</i> ... Look Closer	10/01/1999	Drama	\$19.95	
 Fight Club Director: <i>David Fincher</i> How much can you know	10/15/1999	Action	\$19.95	
 Office Space Director: <i>Mike Judge</i> Work Sucks	02/19/1999	Comedy	\$19.95	

Page 1 of 2

开始的时候我们使用顶部工具栏来包含一些按钮用来“扰乱”我们的 `grid`。现在我们将分页工具栏放在底部工具栏的位置（主要是我觉得把这家伙放上面太丑了）。

下面的代码用来添加分页工具栏：

```
bbar: new Ext.PagingToolbar({
    pageSize: 3,
    store: store
})
```

当然，我们需要把 `data store` 的 `url` 设置为 `PHP` 的服务器端代码的地址。`totalProperty` 配置我们也需要，它是所有数据行的总数的变量名。

```
var store = new Ext.data.Store({
    url: 'movies-paged.php',
    reader: new Ext.data.JsonReader({
        root: 'rows',
        totalProperty: 'results',
        id: 'id'
    }, [
        // data column model removed for readability
    ])
})
```

```
});
```

分组（grouping）：

`grouping grid`（分组表格）用来提供类似的行之间的可见的指标（真绕口，这句话是直译的，意译就是为分组提供指标和指示）。也为我们提供了对每一组别的排序。所以当我们对 `price` 列进行排序，`price` 只在每一组中进行排序。

`grouping store`（分组 `store`）：

我们需要一个特殊的 `store`，叫做 `GroupingStore`！

设置和标准的 `store` 一样，我们只需要提供一些配置项，如 `sortInfo` 和 `groupField`。不必对实际数据做改动，因为 `ExtJS` 在客户端对 `grouping` 会做处理。

```
var store = new Ext.data.GroupingStore({  
  
    url: 'movies.json',  
  
    sortInfo: {  
  
        field: 'genre',  
  
        direction: "ASC"  
  
    },  
  
    groupField: 'genre',  
  
    reader: new Ext.data.JsonReader({  
  
        root: 'rows',  
  
        id: 'id'  
  
    }, // reader column model here //)  
  
});
```

我们还需要为 `grid panel` 添加 `view` 配置项，它用来帮助 `grid` 对分组数据进行显式的说明。

```
var grid = new Ext.grid.GridPanel({  
  
    renderTo: document.body,  
  
    frame: true,  
  
    title: 'Movie Database',  
  
    height: 400,
```

```

width:520,

store: store,

autoExpandColumn: 'title',

columns: // column model goes here //,

view: new Ext.grid.GroupingView()

});

```

做了这些改变后，结果如下：



Title	Released	Genre ▲	Price
Genre: Comedy			
 The Big Lebowski Director: <i>Joel Coen</i> The "Dude"	03/06/1998	Comedy	\$21.90
 Office Space Director: <i>Mike Judge</i> Work Sucks	02/19/1999	Comedy	\$19.95
 Super Troopers Director: <i>Jay Chandrasekhar</i> Altered State Police	02/15/2002	Comedy	\$14.95
Genre: Drama			
 American Beauty Director: <i>Sam Mendes</i> ... Look Closer	10/01/1999	Drama	\$19.95

现在你展开列标题中的背景菜单，你会发现新的菜单项——Group By。这个字段可以允许用户改变分组列。

总结：

我们在这章里学习了很多如何在 **grid** 中展示数据的方式。利用这些只是，我们可以将大量的数据组织在易懂的表格之中。

特别地，我们介绍了：

- 建立 **data store** 和 **grid**；
- 从服务器读取 **XML** 和 **JSON** 数据并显示在 **grid** 中；

- 在渲染单元格式采用各式各样格式化数据的方式；
- 基于用户交互动态改变 `grid`。

我们还讨论了每一个元素的复杂点，如从本地或者服务器读取数据——包括分页。我们还介绍了如何利用 `HTML`、图片甚至查询等方式格式化数据。

现在，我们学习了标准的 `grid`，我们将要进行更高层次的学习，让 `grid` 的单元格式像电子表格一样可以被编辑——这就是咱们下一章的话题。

Editor Grids（可编辑表格）

在之前的一章中我们学习了如何在结构化的表格中显示用户可操作的数据。但是，这种表格有一个最大的限制，那就是用户无法自己编辑表格中的数据。幸运的是，Ext 提供了 `EditorGridPanel` 这一控件，它支持用户对表格进行编辑——我们现在就对这一控件进行学习。它的工作形式类似于 Excel，允许用户在点击单元格后进行数据的编辑操作。


在这章里，我们将会学到：

- 为用户提供连接到 `data store` 的可编辑表格；
- 把编辑后的数据发给服务器，允许用户通过 ExtJS 的 `editor grid` 更新服务器端的数据；
- 通过编码操控 `grid` 并对事件做出相应；
- 教给你一些高级的格式化小窍门并且让您建立更加强大的 `editor grid`。

但是，首先，我们要看看可以用可编辑表格能够做什么事情。

我们能用可编辑表格做什么？

`EditorGridPanel` 和我们之前使用到的表单（form）很相似。实际上，`editor grid` 使用了和 form 中完全一样的表单字段。通过使用表单字段来实现对 `grid` 单元格进行编辑，我们可以充分利用表单字段提供的同样的功能。包括输入约束以及数值验证等。把它和 Ext 的 `GridPanel` 联合起来，就造就了可以满足我们任何需求的漂亮的控件。

Movie Database				
 Add Movie  Remove Movie				
Title	Director	Released	Genre	Tagline
Office Space	Mike Judge	02/19/1999	Comedy	Work Sucks
Super Troopers	Jay Chandrasekhar	02/15/2002	Comedy	Altered State Police
American Beauty	Sam Mendes	10/01/1999	Drama	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	Comedy	The "Dude"
Fight Club	David Fincher	10/15/1999	Action	How much can you

这个表格中的所有字段都可以被编辑，如同使用表格字段中的 `text field`（文本字段）、`date picker`（日期选择器）、`combo box`（组合框/下拉菜单）一般。

使用可编辑的表格：

可编辑和不可编辑表格的区别是一个相当简单的开始的过程。复杂的使我们需要处理编辑状态然后传回服务器。但是一旦你学会了如何去做，这部分也就同样变得相当的简单。

让我们先看看如何把第五章开头部分的表格升级到可以编辑电影标题（title）、导演（director）、和标语（tagline）。修改后的代码如下所示：

```
var title_edit = new Ext.form.TextField();

var director_edit = new Ext.form.TextField({vtype: 'name'});

    var tagline_edit = new Ext.form.TextField({
        maxLength: 45
    });

var grid = new Ext.grid.EditorGridPanel({
    renderTo: document.body,
    frame:true,
    title: 'Movie Database',
    height:200,
    width:520,
    clickstoEdit: 1,
    store: store,
    columns: [
        {header: "Title", dataIndex: 'title',editor: title_edit},
        {header: "Director", dataIndex: 'director',editor: director_edit},
        {header: "Released", dataIndex: 'released',renderer:
Ext.util.Format.dateRenderer('m/d/Y')},
        {header: "Genre", dataIndex: 'genre',renderer: genre_name},
        {header: "Tagline", dataIndex: 'tagline',editor: tagline_edit}
    ]
});
```

为了让 grid 可编辑，我们需要做四件事情。它们是：

- 表格的定义由 `Ex.grid.GridPanel` 转化为 `Ext.grid.EditorGridPanel`;

- 为 grid 的配置添加 clicksToEdit——这个选项不是必须的，默认双击触发编辑；
- 为每列建立一个表单字段用来编辑；
- 通过 editor 配置把表单字段传递给 column model。

editor 可以是 ExtJS 中的任何表单字段，或者是你自定义的字段。我们先从建立文本字段编辑电影标题开始。

```
var title_edit = new Ext.form.TextField();
```

然后，我们利用 editor 配置向 column model 添加表单字段：

```
{header: "Title", dataIndex: 'title', editor: title_edit}
```

接下来把 GridPanel 组件改为 EditorGridPanel，然后添加 clicksToEditor 配置：

```
var grid = new Ext.grid.EditorGridPanel({
    renderTo: document.body,
    frame:true,
    title: 'Movie Database',
    height:200,
    width:520,
    clicksToEdit: 1,
    // removed extra code for clarity
})
```

通过进行这些改变，静态的表格就变为了可编辑的表格，我们可以通过点击任何建立了 editor 的字段来进行编辑。

Movie Database				
Title	Director	Released	Genre	Tagline
Office Space The Musical	Mike Judge	02/19/1999	Comedy	Work Sucks
Super Troopers The Musical	Jay Chandrasekhar	02/15/2002	Comedy	Altered State Police
American Beauty The Musical	Sam Mendes	10/01/1999	Drama	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	Comedy	The "Dude"
Fight Club	David Fincher	10/15/1999	Action	How much can you kn

我们看到了有的电影标题发生了改变，把他们变成了音乐剧(musical)。editor（编辑器）只需你单击便可被激活。按下 **Enter**、**Tab** 键或者在该字段外点击鼠标都可以保存改动。按下 **Esc** 键可以放弃改动。它和表单字段的工作方式很想象……因为……它本身就是个表单字段。

左上角的红色标记表示这个单元格是“dirty”的，即我们只是做了暂时的改动（还没有被提交）。首先，让我们做一些更复杂的可编辑单元格。

编辑更多单元格中的数据：

对于我们基本的 editor grid，我们从使单独的列可编辑开始。为了建立 editor（编辑器），我们建立了对表单字段的引用：

```
var title_edit = new Ext.form.TextField();
```

然后我们把这个表单字段当做列中的 editor 使用：

```
{header: "Title", dataIndex: 'title', editor: title_edit}
```

这就是对每个字段的基本要求，现在让我们扩展一下知识。

使用更多的类型的字段进行编辑：

现在我们利用其它的表单字段来建立 editor。不同的数据类型有不同的 editor 与之对应，每一种字段也有着不同的配置。

所有类型的字段都可以被用来充当 editor，这里列出一些标准的类型：

- TextField
- NumberField
- ComboBox
- DateField
- TimeField
- CheckBox

这些 editor 可以被拓展，从而满足各种编辑需求，但是现在，我们先从编辑电影表格的字段开始——release date（发行日期）列和 genre 列。

编辑日期数值：

对于 release date（发行日期）一列中的单元格我们可以通过使用 DateField 进行编辑。我们先得建立编辑字段并且提供相应格式字符串：

```
release_edit = new Ext.form.DateField({
```

```
format: 'm/d/Y'
```

```
});
```

接下来我们可以把这个 `editor` 应用到列中，采用的方式仍然是之前提及的 `renderer`:

```
{header: "Released", dataIndex: 'released', renderer:
Ext.util.Format.dateRenderer('m/d/Y'), editor: release_edit}
```



这个列还是利用了 `renderer`，`renderer` 和 `editor` 将同时共存。当 `editor` 被激活后，`renderer` 就会对 `editor` 进行渲染等操作。所以当我们完成编辑后，`renderer` 将会进行对该字段的格式化操作。

利用 **ComboBox** 编辑:

让我们为 `genres` 列建立一个 `editor`，用来提供一个显示合法的 `genres` 的下拉列表——听起来很适合采用 `combo box`。

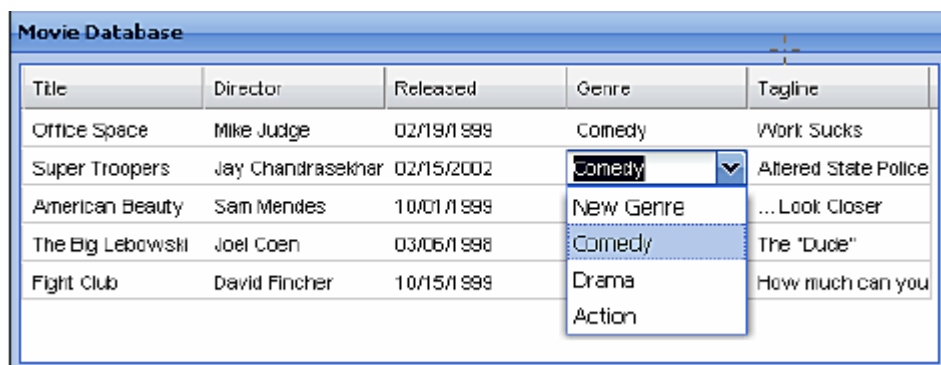
```
var genre_edit = new Ext.form.ComboBox({
    typeAhead: true,
    triggerAction: 'all',
    mode: 'local',
    store: genres,
    displayField: 'genre',
    valueField: 'id'
```

```
});
```

简单的把这个 **editor** 添加到 **column model** 中：

```
{header: "Genre", dataIndex: 'genre', renderer: genre_name, editor: genre_edit}
```

效果如下：



Title	Director	Released	Genre	Tagline
Office Space	Mike Judge	02/19/1999	Comedy	Work Sucks
Super Troopers	Jay Chandrasekhar	02/15/2002	Comedy	Altered State Police
American Beauty	Sam Mendes	10/01/1999	New Genre	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	Comedy	The "Duce"
Fight Club	David Fincher	10/15/1999	Drama	How much can you
			Action	

对单元格编辑做出反应：

当然，我们需要指出如何保存编辑后的结果。用户不会愿意丢弃所有改动的。我们可以通过监听特殊的编辑事件来进行保存，然后通过自定义的函数来对它们做出反应。在我们开始编码前，我们需要了解 **editor grid** 的工作方式。

什么是 **dirty cell**（脏单元格）？

当一个字段被编辑后而且它的值被改动，这时候我们可以称之为 **dirty**（脏），除非 **data store** 得知改到并做保存改动，否则它将一直处于 **dirty** 的状态。这个“dirty”的值被保存在一个暂时的 **data store** 中，这个暂时的 **data store** 保存了包含所有改动的一个数据版本。我们的本来的那个 **data store** 依然保持不变。

我们可以通过调用 **commit** 函数来保存改动，或者通过 **reject** 函数来忽略改动。这些函数可以为整个表格或者某个单元格所调用，或者作为某一事件的结果。

让我们把 **e** 想象为一个 **edit**（编辑）事件对象。我们可以拒绝改动一条记录通过调用 **reject** 函数：

```
e.record.reject();
```

或者，我们可以通过提交来保存改动：

```
e.record.commit();
```

当编辑发生时进行反应：

为了把改动保存到 **data store**，我们将要监听编辑完成这一事件，这一事件的名称叫做 **afteredit**（编辑完成之后）：

```

var grid = new Ext.grid.EditorGridPanel({

    // more config options clipped //,

    title: 'Movie Database',

    store: store,

    columns: // column model clipped //,

    listeners: {

        afteredit: function(e){

            if (e.field == 'director' && e.value == 'Mel Gibson'){

                Ext.Msg.alert('Error','Mel Gibson movies not
allowed');

                e.record.reject();

            }else{

                e.record.commit();

            }

        }

    }

});

```

和其它的 listener（监听器）一样，editor grid 的 listener 提供了一个当事件发生时执行的函数。afteredit 中的 function 在调用时只为其传递了一个参数：一个包含很多有用属性的对象。我们可以利用这些属性来对已经发生的编辑做出决定。

属性	说明
grid	编辑事件所发生的 grid。
record	正在编辑的记录；其他列的数据可以通过使用这个对象的“data”属性找到。
field	被编辑列的名字。
value	包含该单元格改动后的数据的字符串。
originalValue	包含该单元格原始数据的字符串。
row	被编辑的行的 index（序号）。
column	被编辑列的 index（序号）。

举个例子，如果我们想要确定由 Mel Gibson 指导的电影永远不要存进数据库中，我们可以添加一个简单的检验：

```

if (e.field == 'director' && e.value == 'Mel Gibson'){

    Ext.Msg.alert('Error','Mel Gibson movies not allowed');
}

```

```

        e.record.reject();

    }else{

        e.record.commit();

    }

```

首先，我们要确定被编辑的字段是“director”。然后，我们需要保证输入到单元格中的新的值不是 Mel Gibson。如果其中一个是 false，我们便可以把这条记录提交到 data store。这意味着，一旦我们调用了 commit 函数，我们的原始的 data store 就被更新为新的值。

```
e.record.commit();
```

我们还可一拒绝提交——永久删除改动：

```
e.record.reject();
```

当然，现在我们所要做的就是更新在浏览器中存储的数据，我肯定你对如何更新浏览器还是手足无措，我们将很快学到这些。

向 data store 中添加或者删除数据:

我们将添加两个按钮，用来改变 data store——添加或者删除成行的数据。让我们在 grid 中建立一个顶部工具栏(tbar)来放置这些按钮：

```

var grid = new Ext.grid.EditorGridPanel({

    // more config options clipped //,

    tbar: [{

        text: 'Remove Movie'

    }]

})

```

从 data store 中删除 grid 中的行:

我们先往 toolbar 中添加一个 remove 按钮。当我们点击后，将会弹出对话框来显示电影标题。如果我们点击 Yes 按钮，我们就可以从 data store 中移除选中行，否则，将不执行任何动作。

```

{

    text: 'Remove Movie',

    icon: 'images/table_delete.png',

```

```

cls: 'x-btn-text-icon',

handler: function() {

    var sm = grid.getSelectionModel();

    var sel = sm.getSelected();

    if (sm.hasSelection()){

        Ext.Msg.show({

            title: 'Remove Movie',

            buttons: Ext.MessageBox.YESNOCANCEL,

            msg: 'Remove '+sel.data.title+'?',

            fn: function(btn){

                if (btn == 'yes'){

                    grid.getStore().remove(sel);

                }

            }

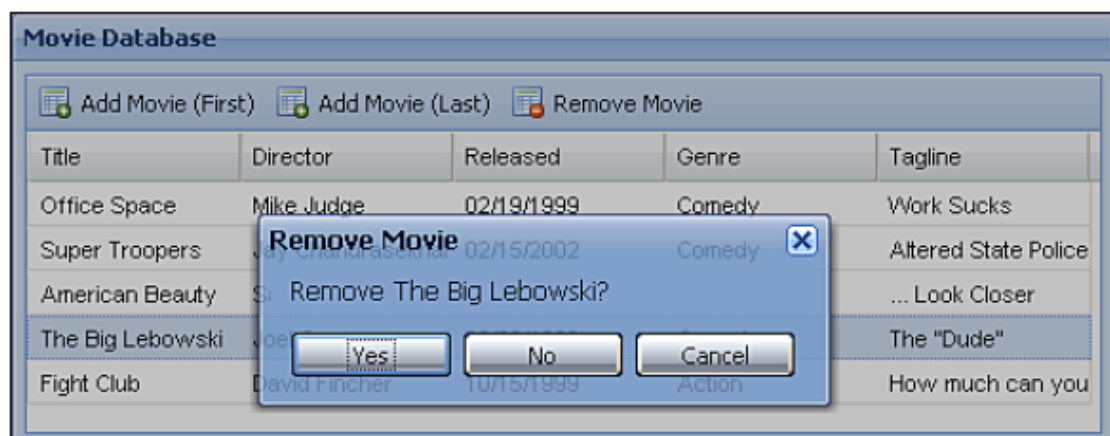
        });

    }

}

}

```



让我们看看这里发生了什么。我们需要定义一些变量用来判断是否有选择事件发生，以及选择了什么：

- sm: 从 grid 中获得的 selection model;

- sel: 我们使用 selection model 来获得被选中的行;
- grid.getStore().remove(sel): 对 data store 执行 remove 函数, 传递被选择的行, 从而从 data store 中移除该行并且更新 grid。

就是这么简单。在浏览器内存中的本地 data store 被更新了。但是你还不能添加东西, 耐心点, 我们接下来就会接触到。

向 grid 中添加行:

为了增加一行我们需要做一些小小的改变。我们需要定义数据是如何展现的, 从而使得我们可以新建一行——如同我们在 data reader 中建立一样。

```
var ds_model = Ext.data.Record.create([  
    'id',  
    'coverthumb',  
    'title',  
    'director',  
    {name: 'released', type: 'date', dateFormat: 'Y-m-d'},  
    'genre',  
    'tagline',  
    {name: 'price', type: 'float'},  
    {name: 'available', type: 'bool'}  
]);
```

当我们为数据做了如上的定义, 我们就可以插入一新行了。可以为 toolbar 添加这样一个按钮来插入行:

```
{  
    text: 'Add Movie',  
    icon: 'images/table_add.png',  
    cls: 'x-btn-text-icon',  
    handler: function() {  
        grid.getStore().insert(  
            0,
```

```

        new ds_model({
            title:'New Movie',
            director:"",
            genre:0,
            tagline:"
        })
    );
    grid.startEditing(0,0);
}
}

```

`insert` 函数的第一个参数是插入数据的位置。我选择了 `0`，所以该记录会被加在最顶端。如果我们想要在最后加入记录，只需要获得行的总数即可，因为行的总数就是比现有最后一条记录大 `1` 的数字。

```

grid.getStore().insert(
    grid.getStore().getCount(),
    new ds_model({
        title:'New Movie',
        director:"",
        genre:0,
        tagline:"
    })
);
grid.startEditing(grid.getStore().getCount()-1,0);

```


Movie Database

Add Movie (First)

Add Movie (Last)

Remove Movie

Title	Director	Released	Genre	Tagline
Office Space	Mike Judge	02/19/1999	Comedy	Work Sucks
Super Troopers	Jay Chandrasekhar	02/15/2002	Comedy	Altered State Police
American Beauty	Sam Mendes	10/01/1999	Drama	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	Comedy	The "Dude"
Fight Club	David Fincher	10/15/1999	Action	How much can you
New Movie			New Genre	

回到正题，第二个参数是新的记录的定义，可以向其传递一些默认值：

```
new ds_model({
    title:'New Movie',
    director:"",
    genre:0,
    tagline:"
})
```

在插入新行之后，我们调用一个函数来激活单元格编辑状态。这个函数只需要行和列的 index 作为参数即可。

```
grid.startEditing(0,0);
```

它可以使用户在点击 **Add Movie** 按钮后直接开始编辑电影名称。

Movie Database

Add Movie (First)

Add Movie (Last)

Remove Movie

Title	Director	Released	Genre	Tagline
New Movie			New Genre	
Office Space	Mike Judge	02/19/1999	Comedy	Work Sucks
Super Troopers	Jay Chandrasekhar	02/15/2002	Comedy	Altered State Police
American Beauty	Sam Mendes	10/01/1999	Drama	... Look Closer
The Big Lebowski	Joel Coen	03/06/1998	Comedy	The "Dude"
Fight Club	David Fincher	10/15/1999	Action	How much can you

将编辑过的数据保存至服务器：

我们现在已经可以更新浏览器中的本地的 **data store** 了。我们还会希望数据可以被保存在服务器端从而更新数据库、文件系统或者之类的什么东西。这段内容将会介绍一些 **web** 应用中 **grid** 的常用需求来更新服务器端的信息。

- 更新一条记录
- 建立一条记录
- 删除一条记录

将更新传至数据库：

之前，我们已经建立了一个针对 **afteredit** 的 **listener**。我们将使用 **afteredit** 事件来向服务器端传递以单元格为单位的数据变动。

为了向服务器更新单元格的改动，我们需要知道 3 个东西：

- **field**：那个字段改变了；
- **value**：字段的新值是什么；
- **record.id**：这个字段属于数据库中的哪一行。

这些信息足够让我们对数据库进行单独的更新。我们通过调用 **connection** 的 **request** 方法来和服务器进行通信。

```
listeners: {  
    afteredit: function(e){  
        var conn = new Ext.data.Connection();  
        conn.request({  
            url: 'movie-update.php',  
            params: {  
                action: 'update',  
                id: e.record.id,  
                field: e.field,  
                value: e.value  
            },  
            success: function(resp,opt) {  
                e.commit();  
            }  
        });  
    }  
}
```

```

    },

    failure: function(resp,opt) {

        e.reject();

    }

});

}

}

```

请求将发给 `movie-update.php`，一共有四个参数。`movie-update` 需要识别 `update` 这一动作，然后读取 `id`、`field` 和 `value` 数据，然后更新文件系统或者数据库，或者其它我们需要改变的东西。

以下是我们使用 `afteredit` 事件的现成的参数：

属性	说明
<code>grid</code>	编辑事件所发生的 <code>grid</code> 。
<code>record</code>	正在编辑的记录；其他列的数据可以通过使用这个对象的“ <code>data</code> ”属性找到。
<code>field</code>	被编辑列的名字。
<code>value</code>	包含该单元格改动后的数据的字符串。
<code>originalValue</code>	包含该单元格原始数据的字符串。
<code>row</code>	被编辑的行的 <code>index</code> （序号）。
<code>column</code>	被编辑列的 <code>index</code> （序号）。

（作者写两次，不觉得烦啊。。。）

从服务器删除数据：

当我们想从服务器删除数据时，我们可以利用和更新同样的方式来处理——向服务器端脚本发请求，然后告诉服务器我们想做什么。

对于删除操作，我们在 `grid` 的 `toolbar` 里使用另一个按钮。

```

{

    text: 'Remove Movie',

    icon: 'images/table_delete.png',

    cls: 'x-btn-text-icon',

    handler: function() {

        var sm = grid.getSelectionModel();

        var sel = sm.getSelected();
    }
}

```

```

        if (sm.hasSelection()){

            Ext.Msg.show({

                title: 'Remove Movie',

                buttons: Ext.MessageBox.YESNOCANCEL,

                msg: 'Remove '+sel.data.title+'?',

                fn: function(btn){

                    if (btn == 'yes'){

                        var conn = new

Ext.data.Connection();

                        conn.request({

                            url: 'movie-update.php',

                            params: {

                                action: 'delete',

                                id: e.record.id

                            },

                            success: function(resp,opt) {

                                grid.getStore().remove(sel);

                                },

                                failure: function(resp,opt) {

                                    Ext.Msg.alert('Error',

                                        'Unable to delete

movie');

                                }

                            });

                        }

                    }

                });

            }

        };

```

```

    }

}

```

我们现在可以通过发请求来删除一行数据。movie-update.php 得知动作为 delete 时会执行相应操作。

在服务器端保存新行：

我们将添加另一个按钮用来增加新的一行。它向服务器发送请求，传递适当的参数，然后从服务器端的响应中读取 insert id。我们可以利用服务器产生的唯一标识——insert id——向 data store 中添加一行。

```

{
    text: 'Add Movie',
    icon: 'images/table_add.png',
    cls: 'x-btn-text-icon',
    handler: function() {
        var conn = new Ext.data.Connection();
        conn.request({
            url: 'movies-update.php',
            params: {
                action: 'insert',
                title: 'New Movie'
            },
            success: function(resp, opt) {
                var insert_id =
Ext.util.JSON.decode(resp.responseText).insert_id;
                grid.getStore().insert(0,
                    new ds_model({
                        id: insert_id,
                        title: 'New Movie',
                        director: '',
                        genre: 0,

```

```

tagline:"

    })

    );

    grid.startEditing(0,0);

    },

    failure: function(resp,opt) {

        Ext.Msg.alert('Error','Unable to add movie');

    }

    });

}

}

```

和编辑删除很相似，我们将会想服务器发送请求来实现插入一行。这次，我们先看下响应。响应中包含了 insert id（行的唯一标识符），g 所以当我们开始编辑那一行时，将很容易保存改动。

```

success: function(resp,opt) {

    var insert_id = Ext.util.JSON.decode(resp.responseText).insert_id;

    grid.getStore().insert(0,

        new ds_model({

            id:insert_id,

            title:'New Movie',

            director:"",

            genre:0,

            tagline:"

        })

    );

    grid.startEditing(0,0);

}

```

`success` 函数有 2 个参数，第一个是响应对象，包含了从 `movie-update.php` 返回的响应文本。因为响应是 JSON 格式，所以我们需要把他 `decode`（解码）成可用的对象然后获取 `insert id`。

```
var insert_id = Ext.util.JSON.decode(resp.responseText).insert_id;
```

当我们将该行插入 `data store` 时，我们可以使用获得的 `insert id` 值。

总结：

`ExtJS` 中的 `grid` 是该架构中最为高级的部分之一。利用 `Ext.data` 包，`grid` 可以从远程数据库采用集成的方法获取信息——这种支持在 `grid` 类中是内置的。感谢现成的配置对象，我们可以把数据以各种形式展示出来并且让用户得以控制。

在这章里，我们知道 `grid` 如何提供数据支持，用来使开发者进行他们所熟知的数据操作。修改和提交的方法让数据得到了很好的控制，并且可以在发送给服务器时进行校验、拒绝改动等操作。除了介绍修改数据，我们还知道了如何提供增加和删除行数据的功能。

我们还看到了标准的 `ExtJS` 表单字段，例如 `ComboBox`，是如何集成到 `grid` 中。凭借这样有力的数据输入支持，`grid` 为应用开发者提供了异常强大的工具。

在接下来的一章，我们将演示诸如 `grid` 这样的组件是如何同其它组件集成到一块显示在屏幕上的。利用广泛的 `layout`（布局），我们可以轻松实现所述的这种功能。

第七章 layout（布局）

layout（布局）可以让表单、表格等控件继承到一个 web 应用中。最常见的布局可以在操作系统中找到，例如微软的 windows，它利用的就是 border layout（边界布局），resizable region（可改变大小的区域），accordions（层叠面板），tabs（标签页）和其它能想到的东西。

为了让不同浏览器有同样的用户界面，ExtJS 提供了强大的 layout management system（布局管理系统）。布局中的每一部分都可以被管理和控制，你可以移动或者隐藏它们，然后你可以通过单击一个按钮来在你需要的时候显示它们。

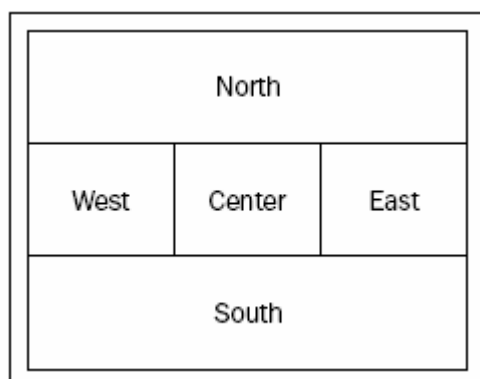
在这章中，我们将学到：

- 为一个应用进行布局；
- 建立标签布局；
- 通过 layout 管理 Ext 控件；
- 学习高级嵌套的 layout。

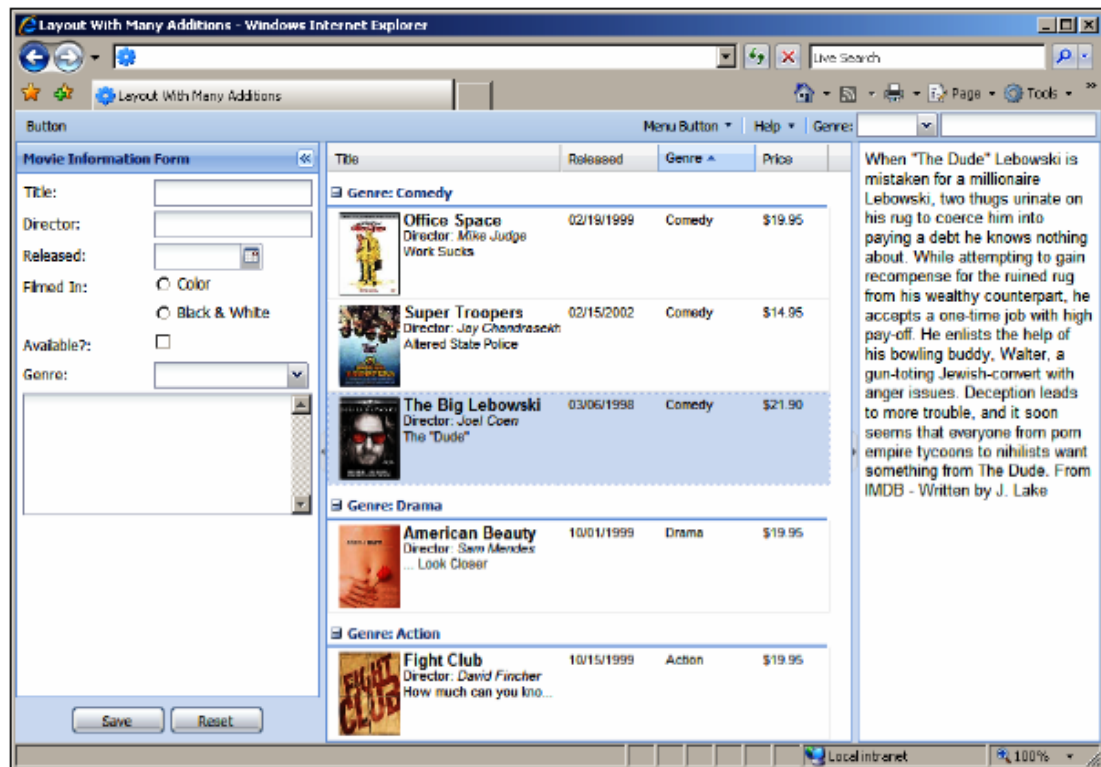
什么是 layout，region 和 viewport？

Ext 使用 **Panel**（面板）作为大多数 layout 的基础。我们已经使用了其中的一些，如 **FormPanel**（表单面板）、**GridPanel**（表格面板）。**viewport** 有点像 panel，它包含了整个布局，填充浏览器的整个可视区域。在我们的第一个例子中，我们将使用 viewport 和 border layout 来包含多个 Panel。

viewport 有 region（区域），如同罗盘一样，有 **North**（北）、**South**（南）、**East**（东）和 **West**（西）这四个 region——**Center**（中央）region 显示在中间。这些方向告诉 Panel 该处于 viewport 的哪个方位上。在形成多 panel 后，他们之间还会有可移动的边界。



显示效果如下，它集成了很多我们之前用到的示例：



这种 layout 叫做“border” layout，region 之间靠一个可拖拽的三维分界线分开。这个示例包含了 4 个 panel region。

North：工具栏；

West：表单；

Center：在标签面板中的 grid；

East：包含了文字的面板。

注意，这里没有“South”panel——不是所有的区域都要被设置。

我们的第一个 layout：

在我们建立如上所示的包含四个 region 的 layout 之前先建立一个包含全部区域的 layout，然后再把 south 面板去掉。我们把所有区域都应用为 Panel。

```
var viewport = new Ext.Viewport({
    layout: 'border',
    renderTo: Ext.getBody(),
    items: [{
        region: 'north',
        xtype: 'panel',
```

```

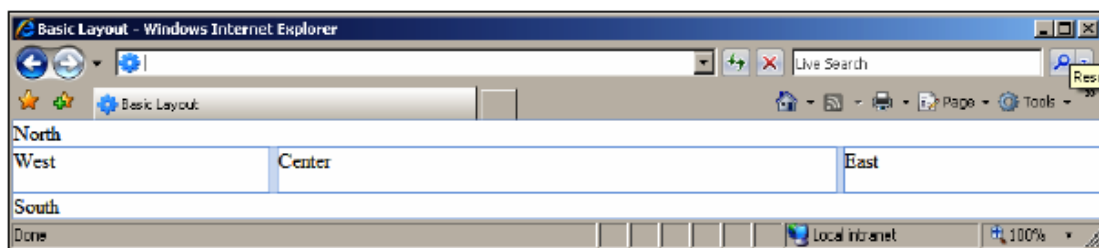
        html: 'North'
    }, {
        region: 'west',
        xtype: 'panel',
        split: true,
        width: 200,
        html: 'West'
    }, {
        region: 'center',
        xtype: 'panel',
        html: 'Center'
    }, {
        region: 'east',
        xtype: 'panel',
        split: true,
        width: 200,
        html: 'East'
    }, {
        region: 'south',
        xtype: 'panel',
        html: 'South'
    }
    ]
});

```

每个 `region` 被定义了四个方向——`East`、`West`、`North`、`South` 中的一个。`center` 区域会被用来填充剩余空间。为了区别各个区域，我们在 `html` 配置项中添加了相应的文本（你可以使用更复杂的 `HTML`，但是我们现在使用最简单的来说明问题）。

N: ExtJS 提供了跨浏览器的、简单的对 `body` 元素的引用方法，使用 `Ext.getBody()`；

如果工作正常，浏览器如下所示：



可以往每个 **region**（区域）中添加其他的控件，也可以通过分割来嵌套区域，例如，**center** 可以被水平分割出自己的 **south** 区域。

N: “Center” 区域必须被定义，否则会产生错误并显示混乱。

分割 **region**:

分割线的建立是靠配置 **split** 属性——分割线的位置由布局中的面板决定。

split: true

对于该页面，我们为 **West** 和 **East** 区域设置了 **split**。默认的，它会使得边界成为一个可改变大小的元素，从而用来改变面板的大小。

我需要配置项:

通常来说，当使用了 **split**，它会和其他一些有用的配置项一起使用，例如：**width**、**minSize** 和 **collapseMode**。

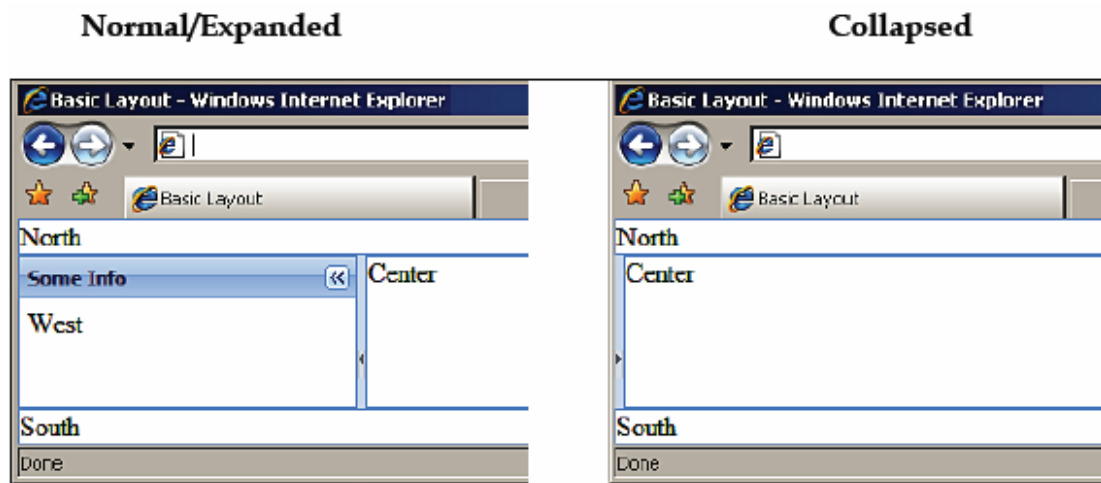
这里列出一些常用的选项:

选项	值	说明
split	true/false	布尔值，决定是否在两个区域间放置可拖拽分割线。
collapsible	true/false	布尔值，向标题栏中添加一个按钮，用户可以通过单击来收起一个区域。
collapseMode	只有“mini”一种 mode（模式），对于其他 mode，值为 undefined	当设置为“mini”时候，分割线上会出现一个收起按钮，点击后，会将面板缩起来。
title	String	标题栏中的标题。
bodyStyle	CSS	Panel 的 body 元素的 CSS 样式。
minsize	像素	用户可以把 panel 拖拽的最小值。
maxsize	像素	用户可以把 panel 拖拽的最大值。
margins	按照上右下左的顺寻设置的像素值	Panel 和边缘的距离，添加的空白处于 panel 的 body 之外。
cmargins	同上	和 margins 的思想一样，但是只应用于 panel 收起之后。

让我们在 west 面板中添加一些这里的配置项：

```
{
    region: 'west',
    xtype: 'panel',
    split: true,
    collapsible: true,
    collapseMode: 'mini',
    title: 'Some Info',
    bodyStyle: 'padding:5px;',
    width: 200,
    minSize: 200,
    html: 'West'
}
```

会有如下的效果：



N: 展开和收起一个 panel 时，如果 panel 没有设置 width，将会出现渲染错误。因此，最好为 panel 设置宽度——当然除了 center，因为这个 panel 是自动填充剩余空间的。

Tab panels（标签面板）：

在 Ext 中 tab panels 又被叫做“卡片”布局，因为它工作起来就像层叠的卡片，每个卡片都可以存在于其它的卡片上方或者下方。我们可以发现 tab panel 的基本功能和普通的 panel 一样，包含标题，工具栏，和其他常用配置。

添加一个 tab panel：

如果 ExtJS 的组件是一个 panel 类型的组件，例如 GridPanel 和 FormPanel，我们就可以通过使用 xtype 直接将其加入布局中。让我们从建立一个 tabPanel 开始：

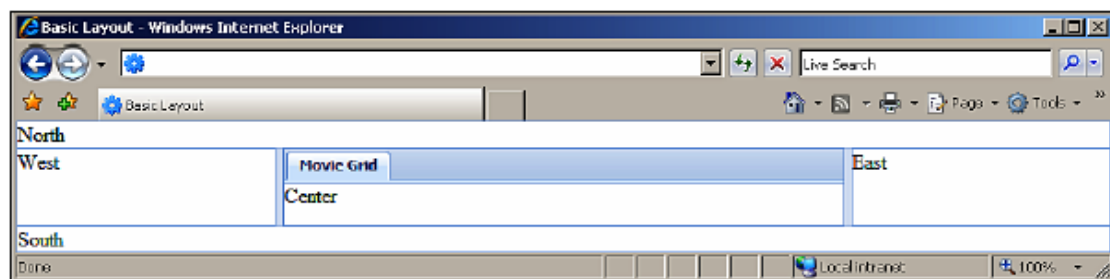
```
{
    region: 'center',
    xtype: 'tabpanel',
    items: [{
        title: 'Movie Grid',
        html: 'Center'
    }]
}
```

items 配置是一个包含对象的数组，定义了每个标签页。title 是唯一一个必须设定的属性，html 用来填充标签页，来给我们空白的标签一些内容。

我们还需要添加一个 **activeTab** 配置来设置初始激活的标签为哪个。它的值为从左到右的标签的序号（从 0 开始）。如果不设置 **activeTab** 则初始化为空面板，直到选择某个标签。

```
{  
  
    region: 'center',  
  
    xtype: 'tabpanel',  
  
    activeTab: 0,  
  
    items: [{  
  
        title: 'Movie Grid',  
  
        html: 'Center'  
  
    }]  
  
}
```

结果如下：



为 **items** 添加更多项目可以新建标签。每个 **item** 就是一个面板(panel)，显示与否取决于你是否单击了对应的标签 **title**。

```
{  
  
    region: 'center',  
  
    xtype: 'tabpanel',  
  
    activeTab: 0,  
  
    items: [{  
  
        title: 'Movie Grid',  
  
        html: 'Center'  
  
    }, {  
  
        title: 'Movie Descriptions',  
  
    }]  
  
}
```

```

        html: 'Movie Info'

    }}

}

```

Movie Grid 和 Movie Descriptions 标签现在还是简单的面板。所以，让我们为其添加一些配置项和控件。

随处的控件：

前面我曾提到所有类型的 Panel 都可以直接加到布局中，如同标签面板中一样。让我们往布局中再添加另一个控件——grid。

向 tabpanel 中添加 grid：

我们可以先把 grid 添加到一个标签中：添加 xtype 配置项到 grid 的配置代码中（该代码在第五章中有），我们就可以建立一个 grid 标签：

```

{
    region: 'center',

    xtype: 'tabpanel',

    activeTab: 0,

    items: [{

        title: 'Movie Grid',

        xtype: 'gridpanel',

        store: store,

        autoExpandColumn: 'title',

        columns: // add column model //,

        view: // add grid view spec //

    }, {

        title: 'Movie Descriptions',

        html: 'Movie Info'

    }]

}

```

N: xtype 提供了一种实例化新组件的快捷方式。有时我们称这种方式为“lazy rendering”（惰性渲染），因为组件在它们执行任何代码前一直闲置地等待显示。这种方式可以帮助我们节省内存。

当我们向标签中添加 grid 的时候，我们不再需要一些配置了（如 renderT、width、height、frame）。grid 的大小，标题和 grid 的边框都被 tab panel 处理了。

现在布局如下所示：



Accordions（手风琴<层叠布局>）：

accordion 是一种很有用的 layout，它的工作方式有点像 tab panel，在同一空间里集成了多个部分，但是每次只能显示其中一个。这种布局常常用在缺乏水平空间但是有充分的垂直空间的时候（tab panel 解决不了）。当其中一个 accordion 面板展开的时候，其它的将会折叠起来。收起和折叠一个 panel 可以通过点击 panel 的标题或者点击最右端的 plus/minus 图标。

在 tab 中嵌入 accordion：

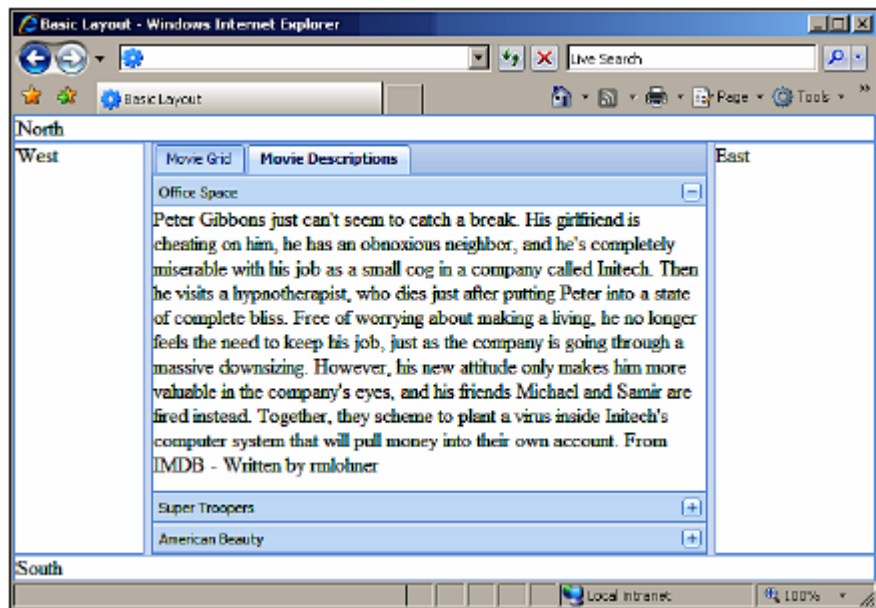
我们可以通过嵌套来实现复杂的布局。在本例中，我们将一个 **accordion** 面板嵌套到一个标签页中。

将 **layout** 设置为 “**accordion**” 并且添加 3 个 **item**，我们可以在 **accordion** 布局中建立 3 个面板(**panel**)。

```
{
    title: 'Movie Descriptions',
    layout: 'accordion',
    items: [{
        title: 'Office Space',
        autoLoad: 'html/1.txt'
    }, {
        title: 'Super Troopers',
        autoLoad: 'html/3.txt'
    }, {
        title: 'American Beauty',
        autoLoad: 'html/4.txt'
    }]
}
```

现在，一个标签页中包含了 3 个 **accordion** 面板，分别加载了 3 个文本文件。注意，这里的配置和 **tab panel** 很相似——控件间的一致性可以是我们在建立不同控件的时候不用查看每个的 **API** 文档。

当我们切换到 **Movie Description** 标签的时候，会看到如下结果：



现在每个面板中都包含了电影简介了。让我们仔细看下以下配置项：

```
autoLoad: 'html/1.txt'
```

它从一个 URL 加载了一个文件到 panel 中，这个文件中可以包含任何种类的 HTML 文本，文本的内容会显示在浏览器中。这里依靠的是 AJAX，如果你不在本地 web 服务器上做这个示例而是在你自己的文件系统中做，那么你会发现，文件不会被加载。

N: 注意包含在文件内容中的 JS 脚本不会被执行，这时所有 HTML 会被忽略。

向布局（layout）中添加 toolbar（工具栏）：

接下来，让我们为 North 部分添加一个 toolbar。我们可以使用包含 menu、button、一系列 filed 的 toolbar，或者只是为了显示闪烁的包含这名字信息的滚动文字。我们一会儿可以很方便的改动。

我们使用第四章介绍的 toolbar 中的 item——Button、Menu 和 Toolbar——然后把它们加入到这个 toolbar 中。我们还需要复制 Movies 类（第四章中的），如果我们想要让按钮工作的话。将 xtype 设置为 toolbar 然后复制 toolbar 的 item 数组，我们将在屏幕顶端建立一个漂亮的菜单栏。

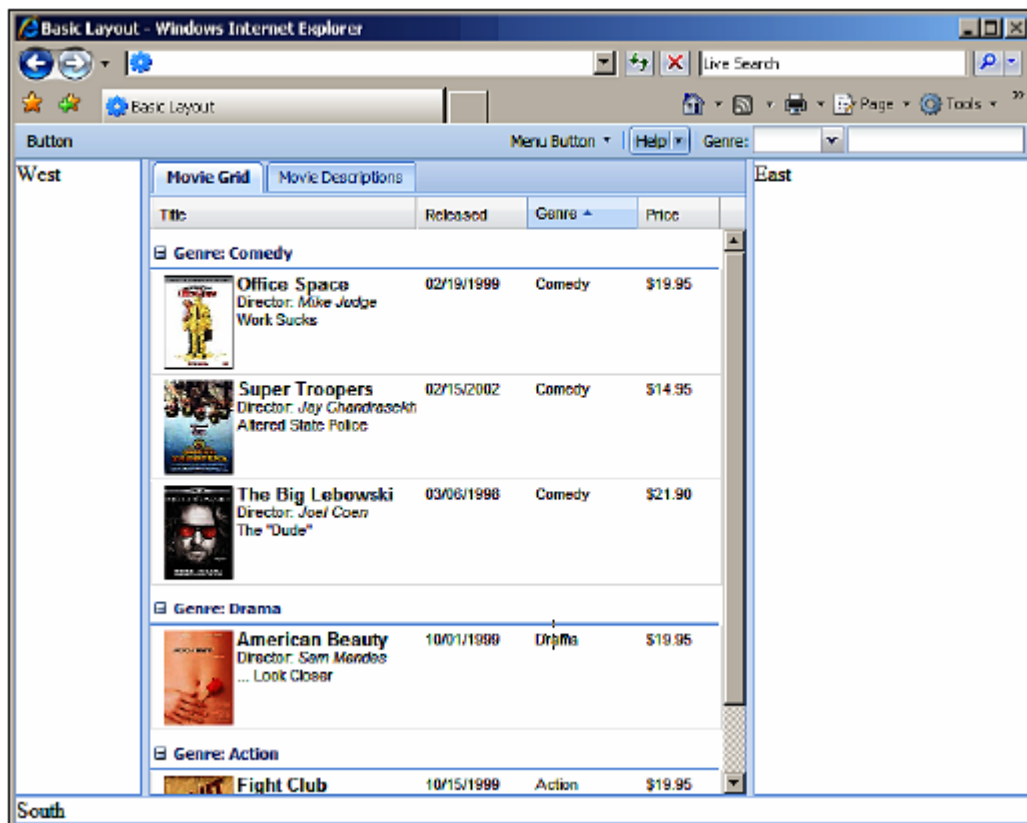
```
{
    region: "north",
    xtype: 'toolbar',
    items: [{
        xtype: 'tbspacer'
```

```

    }, {
        xtype: 'tbutton',
        text: 'Button',
        handler: function(btn) {
            btn.disable();
        }
    }, {
        type: 'tbfill'
    },
    // more toolbar items here //]
}

```

这样，我们就有了一个很好地填充布局顶部的工具栏——如同一个桌面应用程序中的 toolbar 或者菜单栏一样。最后如下显示：



虽然这个 toolbar 里没有闪烁滚动我的名字。但是它对于一个应用来说十分有用。所有应用导航都可以放在里面，可以通过它来激活中央区域的菜单，或者用它来查找电影已经进行其他操作。

一个用来添加新电影的表单（form）：

对于左边的区域我们正好可以添加一个表单。因为它是一个 **Panel** 类型的组件，所以我们可以直接将其加载 **layout** 中。让我们为 **west** 区域添加一个电影表单（在表单一章中使用到的）。但是，不同的是，我们不去实例化它，而是直接添加 **xtype** 配置来惰性渲染整个表单面板。

```
{  
  
    region: 'west',  
  
    xtype: 'form',  
  
    items: // form fields //  
  
    buttons: // form panel buttons //  
  
}
```

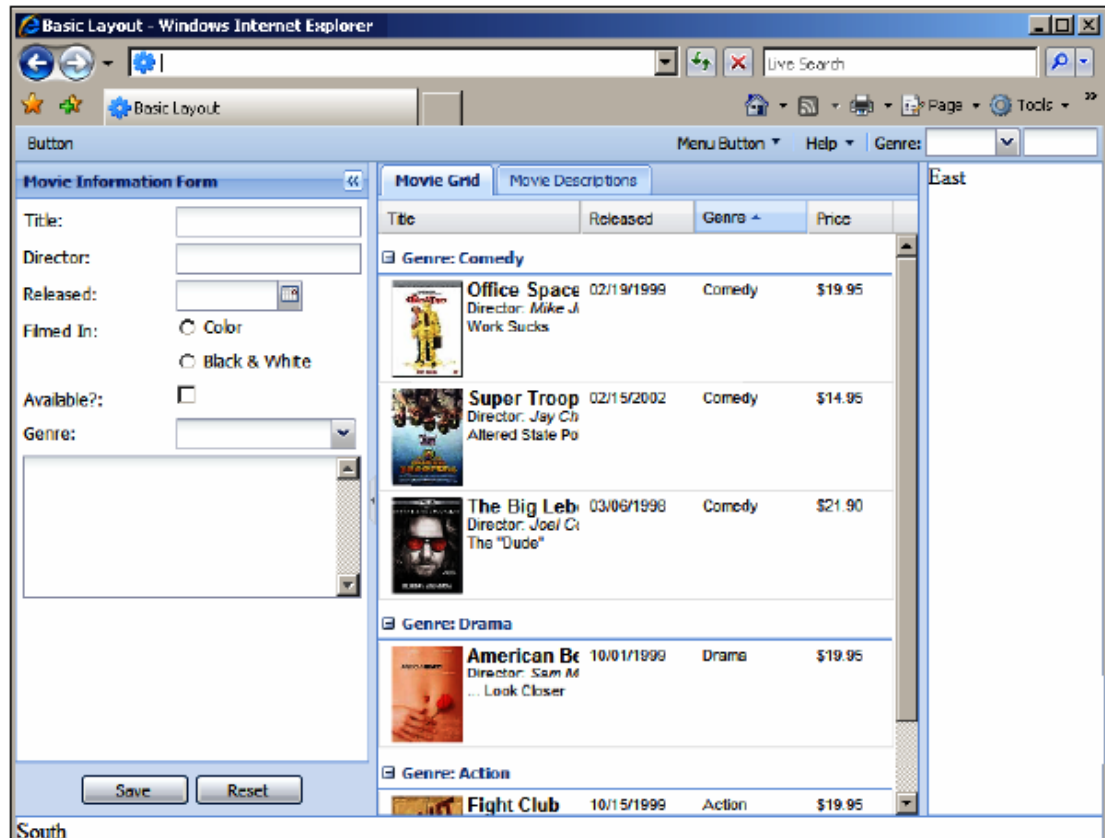
items 配置项包含了所有的表单字段：

```
items: [{  
  
    xtype: 'textfield',  
  
    fieldLabel: 'Director',  
  
    name: 'director',  
  
    anchor: '100%',  
  
    vtype: 'name'  
  
},{  
  
    xtype: 'datefield',  
  
    fieldLabel: 'Released',  
  
    name: 'released',  
  
    disabledDays: [1,2,3,4,5]  
  
},{  
  
    xtype: 'radio',  
  
    fieldLabel: 'Filmed In',  
  
    name: 'filmed_in',  
  
    boxLabel: 'Color'
```

```
} // more fields go here //
```

N: 有很多 xtype，但是它们的名字你不可能准确的猜出——你可以通过查看 API 中的 Component 来找到相应的名字。

通过往表单中添加 items 和按钮，我们的 layout 如下所示：



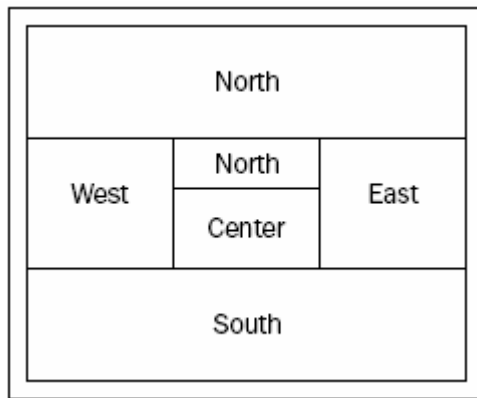
敲门和高级 layout:

一些更加复杂的应用不仅仅局限于添加一些配置，例如 layout 嵌套或者在标签上添加图标。但是在 ExtJS 中这些事情变得很简单。

嵌套布局:

当我们把一个 layout 嵌套到另一个 layout 的区域中，我们会占据这个区域的所有空间。而被嵌套着的布局中的区域是用来展示内容的。

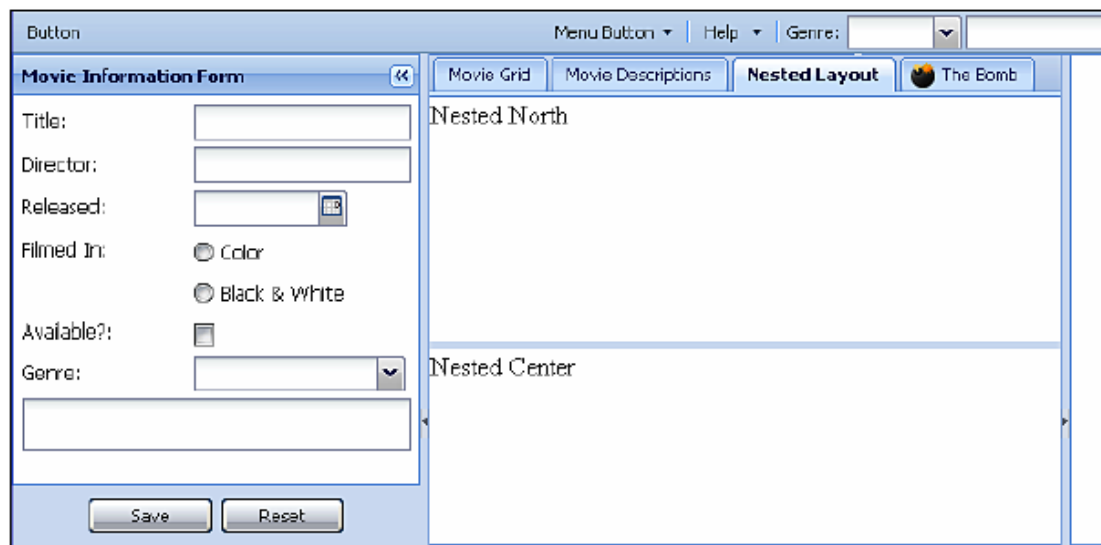
例如，如果我们想把 center 区域水平分成两部分，我们需要嵌套一个拥有 Center 和 North 的布局。这是一种常用的应用——你在 Center 面板中显示 email 的信息，然后在 South 面板中预览整个 email，而 North 面板中展示列表。



要实现嵌套布局，还有事情要做——**layout** 的类型需要定义，在这个例子中，我们为 **border** 选项赋 **false** 以防止出现两个边界，因为容器有它自己的边界。每一个 **item** 表示一个嵌套的区域。

```
{  
  title: 'Nested Layout',  
  layout: 'border',  
  border: false,  
  items: [{  
    region: 'north',  
    height: 100,  
    split: true,  
    html: 'Nested North'  
  }, {  
    region: 'center',  
    html: 'Nested Center'  
  }]  
}
```

结果如下所示：



N: border 布局不允许使用百分数。必须有“center” region 来占据剩余空间，大小全部以像素为单位定义。

标签上的图标：

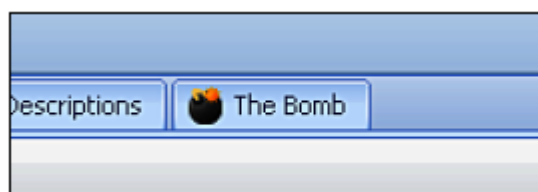
你不希望标签上有能够图形化显示该标签作用的东西吗？那就是标签上的图标。这个图标有点像我们之前章节中添加到按钮上的图标。我们所要做的就是为图标建立样式然后添加到标签的配置中。

样式如下：

```
bomb {
    background-image:url(images/bomb.png) !important;
}
```

标签配置中需要添加 iconCls 属性，属性值为刚才的样式名称：

```
{
    title: 'The Bomb',
    iconCls: 'bomb',
    html: 'Boom!'
}
```



不要太快点击这个标签，它可能会消失！（作者大哥，还有这种事情？这该算 bug 了。）

程序化控制布局：

我们可以在 `layout` 渲染之后修改几乎所有东西。例如，我们可以添加新的标签，隐藏和显示面板以及改变任意面板的内容。让我们研究一下这些。

你看见我了，我又消失了！

程序控制收起和展开 `layout` 中的某一部分是一种在应用中最常见的需求。我们现在可以不用惊奇 `ExtJS` 可以做到这点。

我们要做的第一件事就是给我们 `Panel` 和 `viewport` 一个 `id`，方便我们定位它们。我们可以通过添加 `id` 配置来实现：

```
var viewport = new Ext.Viewport({  
  
    layout: 'border',  
  
    id: 'movieview',  
  
    renderTo: document.body,  
  
    items: [{  
  
        // extra code removed //  
  
        region: 'east',  
  
        xtype: 'panel',  
  
        id: 'moreinfo'  
  
        // extra code removed //  
  
    }  
  
});
```

现在，`layout` 和 `panel` 都有了 `id`，我们可以利用 `id` 来实现和这些组件进行交互，使用 `getCmp`。

```
var moreinfo = Ext.getCmp('movieview').findById('moreinfo');  
  
if (!moreinfo.isVisible()){  
  
    moreinfo.expand();  
  
}
```


唯一的一点代码使用来判断 `panel` 是否 `visible`（可见，即是否 `expanded`），如果不是的话，就去展开它。

给我另一个标签：

添加一个标签和创建一个单独的标签一样简单。我们需要先定位 `tab panel`。幸运的是我们定义了 `id`，所以很容易定位。

```
{  
  
    region: 'center',  
  
    xtype: 'tabpanel',  
  
    id: ' movietabs',  
  
    activeTab: 0,  
  
    items: [{  
  
        title: 'Movie Grid',  
  
        // extra code removed //  
  
    }, {  
  
        title: 'Movie Descriptions',  
  
        html: 'Movie Info'  
  
    }]  
}
```

然后我们可以调用 `add` 函数来添加标签，为该函数传递一个基本的对象：

```
Ext.getCmp('movieview').findById('movietabs').add({  
  
    title: 'Office Space',  
  
    html: 'Movie Info'  
  
});
```

这样就为 `movietabs` 标签面板添加了一个标题为 `Office Space` 的标签。

`add` 函数是向 `layout` 或者控件中添加 `item` 的一种方式。一般来说，`item` 的所有配置都可以传递给 `add` 函数。

总结：

在这章里，我们在**layout**（布局）中使用了很多之前章节中的组件。**layout**真的可以管理多样的组件，并且利用它们创建一个**web**应用。我们看到，**layout**可以集成不同的组件到一个应用中。我们也学到了如何改变**panel**的状态，建立嵌套布局，动态加载内容。

第八章 Tree(树) (1)

层级数据是很多开发者所熟知的。“根-枝-叶”的结构是很多用户界面的最基本的特征。windows 的资源管理器中就展示了一个包含子节点、父节点和更深层次节点的树，以此来展示文件夹和文件的层级关系。Ext.tree 允许开发者只通过使用几行代码就展示出这样的层级数据，并且提供了大量的简单的配置来适应更广泛的需求。

虽然 ExtJS 默认的 tree 节点是 file 和 folder 图标样式，但是它不会仅仅将树局限于文件系统这一概念里。每一项的图标和文字，或者树中的节点，都可以根据动态或者静态的数据来改变——不需要自己写代码。想想，我们如果希望建立一个用户组，为每个用户展示它们自己的图标；又或者希望展示一画廊的图片并且在图标中预览这些图片。ExtJS 可以帮助我们实现这些愿望，而且十分简单。

种植未来：（作者使用培育植物的过程形象化地比喻建立 tree 的过程）

ExtJS 的 tree 不会关心你显示什么样的数据，因为它可以随心所欲地处理任何你碰到情况。数据可以实现就加载好，又或者从逻辑上进行分割。你可以直接在 tree 中编辑数据，改变标签和位置，或者你可以修改整棵树的样子以及每个节点的外观，一切都为了用户体验。

ExtJS 的 tree 是从 Component 模型上建立起来的。Component 是真个 ExtJS 框架的基础。也就是说，开发者从熟悉的 Component 的系统中获得了便利，因为用户能得到一种统一的和集成的体验效果。你同样可以保证 tree 和应用中的其他组件天衣无缝地工作在一起。

从小种子开始：

在这章里，你可以看到如何使用很少的代码建立一棵树。我们还将讨论利用唯一的数据结构来产生一个 tree，以及如何使用数据可以让你操控重要的配置项。ExtJS 树提供了很多高级支持，例如排序、拖拽等。但是，如果你想要真的定制一棵，我们还需要探索如何重写或者扩展 configure options(配置项目)、methods(方法)、events(事件)的方法。

tree 是通过实例化 Ext.tree.TreePanel 类来建立的，它包含了很多 Ext.tree.TreeNode 的节点类。这两个类是 ExtJS 树的核心，也是我们这章讨论的主题。但是，还有很多其他先关的类需要介绍，我们现在列出 Ext.tree 包中的全部条目：

AsyncTreeNode	允许子节点异步加载的节点
DefaultSelectionModel	标准的 TreePanel 的单选模式
MultiSelectionModel	允许多选节点的选择模式
RootTreeNodeUI	作为 TreePanel 根的特殊的 TreeNode
TreeDragZone	为 TreeNode 的抓起提供支持
TreeDropZone	为 TreeNode 的放下提供支持
TreeEditor	允许节点标签被编辑
TreeFilter	对 TreePanel 中子节点的过滤进行支持
TreeLoader	从指定的 URL 生成 TreePanel

TreeNode	在 TreePanel 中显示节点的最主要的类
TreeNodeUI	为 TreeNode 提供最基本的界面
TreePanel	树状结构显示数据——最主要的树类
TreeSorter	支持 TreePanel 中节点的排序

天啊！幸运的是，你不用同时全部使用它们。TreeNode 和 TreePanel 提供了最基础的东西，其它的类是用来提供额外功能的。我们将一个一个对其介绍，讨论如何使用它们并且展示几个练习示例。

我们的第一个幼苗：

现在，你可能还在思考 ExtJS 树带来的各种可能性，想亲手去干。尽管 Ext.tree 类包含了丰富的功能，但是你只要需要几行代码就能让一切跑起来。

在接下来的例子中，我们假设你有一个准备好的空白 HTML 页面，包含了所有 ExtJS 所需要的引用。大部分的代码都基于以前的章节，好让我们抓住重点，你要孤立地看待它们。最好的方式是吧 JS 分别放在各个文件中并把代码放在 Ext.onReady 函数中。但是，你依然可以根据你自己的编码风格来处理。

准备好土地：

首先我们需要建立<div>元素，用来向其渲染 TreePanel。因此我们需要把它设置为我们想要的 tree 的大小：

```
<div id="treecontainer" style="height:300px; width:200px;"></div>
```

tree 的 JS 代码可以分为三部分。首先，我们需要确定 tree 展现的方式。Ext.tree.TreeLoader 类提供了这样的功能，现在我们采用最简单的办法来使用它：

```
var treeLoader = new Ext.tree.TreeLoader({  
  
    dataUrl:'http://localhost/samplejson.php'  
  
});
```

dataUrl 配置项说明了提供产生树所需要的 JSON 数据的脚本存在的位置。我现在不想讨论 JSON 的具体结构，让我们先保留这个问题。

每个 tree 都需要一个根节点，它扮演了所有后裔的终极祖先的角色。为了建立 root node（根节点），我们使用 Ext.tree.AsyncTreeNode 类：

```
var rootNode = new Ext.tree.AsyncTreeNode({  
  
    text: 'Root'  
  
});
```

之所以使用 AsyncTreeNode 而不是 TreeNode，是因为我们需要从服务器端获得节点，并且一层一层地去加载而不是一次性加载。

N: AsyncTreeNode 使用 AJAX 来保证用户不用花太多时间等待数据加载以及首次节点的渲染。

最后，我们建立 tree 本身，利用 Ext.tree.TreePanel 类：

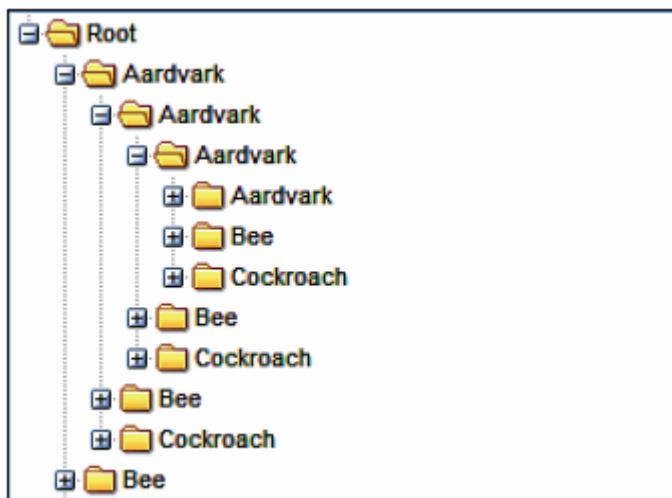
```
var tree = new Ext.tree.TreePanel({  
  
    renderTo:'treecontainer',  
  
    loader: treeLoader,  
  
    root: rootNode  
  
});
```

只需要把 root node 和 TreeLoader 放在配置中，再加上决定 TreePanel 渲染位置的 renderTo 配置，就可以显示 tree 了。

再次提醒，你一定要记住把这些代码放在 Ext.onReady 中，以此确定 DOM 在代码执行前已经准备好了。

tree(树)离开 data(数据)生长不了：

我们看到，只需要十一行数据就可以显示如下的用户界面：



我猜这还不够，但是这十一行代码提供了大量的功能。利用异步远程加载子节点，我们获得了统一的交互界面和体验。并不只是这么简单，因为我们将要介绍 Ext 树的最重要的部分——data（数据）。

JSON:

标准的 TreeLoader 支持以一种特定的格式支持 JSON 数据——包含 node 定义的数组，如下所示：

[

```
{ id: '1', text: 'No Children', leaf: true },  
  
{ id: '2', text: 'Has Children',  
  
  children: [{  
  
    id: '3',  
  
    text: 'Youngster',  
  
    leaf: true  
  
  ]  
  
}  
  
]
```

`text` 属性代表了 `tree` 里 `node` 的标签文本。`id` 属性用来唯一标识一个 `node`，并且将被用来决定选中和展开哪个节点。利用 `id` 属性我们可以使得 `TreePanel` 中高级功能的实现变的简单，这在之后会介绍到。`children` 属性是可选的。`leaf` 属性是被用来判断是否为叶子节点。在 `tree` 中 `leaf` 节点不能被展开，也不会有节点前卖弄的加号图标。

ID 简介:

默认来说，`TreeNode` 会被分配一个自动产生的 ID，说明 `id` 配置可有可无。这个自动产生的 `id` 是一个字符串，形式如：`ynode-xx`，`xx` 代表数字。`id` 可以被用来获得一个你之前引用的节点。但是，你很可能自己分配 `id`。当你异步加载节点的时候，服务器脚本需要准确知道那个节点被点击从而传回其子节点数据。通过设置 `id`，你可以发现在服务器端匹配节点变得很简单。

额外的数据:

虽然 `id`、`text` 和 `leaf` 属性十分常用，但是 `JSON` 的用途不只局限于此。事实上，任何 `TreeNode` 的配置都可以被 `JSON` 初始化，这是我们探索 `tree` 的高级功能的一个小技巧。你可以添加应用需要的特定的数据——例如也许你的节点代表了产品并且你希望它们包含价格信息。任何属性都可以以 `TreeNode` 配置项的方式组织起来，并且包含在 `TreeNode` 的 `attributes` 属性之中。（也就是说，`TreeNode` 本身没有的属性会被组织在 `attributes` 属性之中）。

XML:

`XML` 不是直接被 `tree` 支持的。但是你可以利用 `ExtJS` 的数据支持来让 `XML` 也能被读取。通常，使用 `JSON` 会让一切变得简单，虽然一些程序会使用 `XML` 传送数据。所以它值得我们讨论一下。

我们可以使用 `Ext.data.HttpProxy` 来获得数据，但是我们需要在读取数据的时候把 XML 转换一下：

```
var xmltree = new Ext.tree.TreePanel({el: 'treeContainer'});

var proxy = new Ext.data.HttpProxy({url:

    'http://localhost:81/ext/treexml.php'});

proxy.load(null, {

    read: function(xmlDocument) {

        parseXmlAndCreateNodes(xmlDocument);

    }

}, function(){ xmltree.render(); });
```

我们建立了一个新的 `TreePanel` 和 `HttpProxy`，并且指定在 `proxy` 加载时使用 `Ext.data.Reader` 来处理进入的 XML 数据。我们接下来会告诉 `reader` 把 XML 传递到 `parseXmlAndCreateNodes`。在这个函数中，你可以根据 XML 数据来建立根节点和子节点，我们向它直接传递了一个 XML 文档。

JS 完全能够处理 XML 数据，虽然你可能更习惯于转换 XHTML 文档中 DOM 的方式。通过读取 XML 文档你可以建立和使用 `textnodes`（文本节点）。因为在这种方式里，你需要访问原始的 XML 节点，你可以完全地控制由此产生的 `tree` 和相应的树节点。

照料你的树：

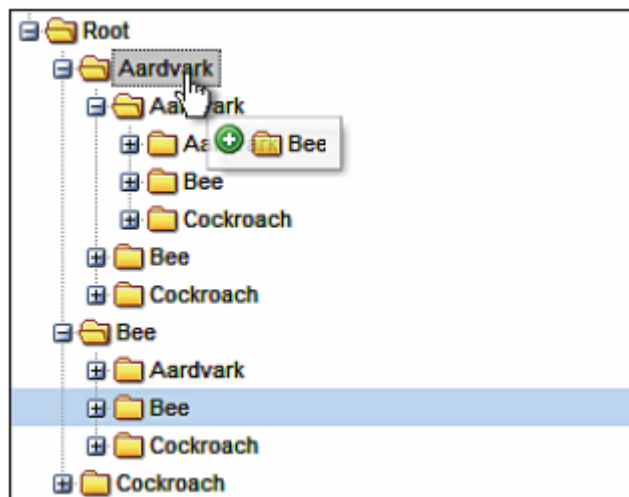
我们将讨论使你的 `tree` 更加实用的功能。拖拽、排序和编辑节点。

拖拽：

当你使用 `TreePanel` 的时候，`ExtJS` 管理着由拖拽产生的用户界面。只要添加 `enableDD: true` 配置到 `tree` 里，这时你可以通过拖拽重新组织树的节点，当显示绿色加号的时候，代表你可以向目标防止该节点。

N： `TreePanel` 不仅仅只会“照顾”它们自己的节点，当你的界面中有两棵树的时候，他们的节点可以在树之间相互拖拽。

但这还没有完。当你刷新你的页面，所有的节点又回到原位。因为 `TreePanel` 不会自动知道你是否需要保存改动，为了实现保存，我们需要借助一些事件（`event`）。



TreePanel 的 beforemovenode 事件在拖放的时候松开鼠标，但是在 TreePanel 反映变化前这一时间点触发。我们可以添加如下代码来告诉服务器移动节点这一事件：

```
tree.on('beforemovenode', function(tree, node, oldParent, newParent, index) {  
  
    Ext.Ajax.request({  
  
        url: 'http://localhost/node-move.php',  
  
        params: {  
  
            nodeid: node.id,  
  
            newparentid: newParent.id,  
  
            oldparentid: oldParent.id,  
  
            dropindex: index  
  
        }  
  
    });  
  
});
```

我们为 beforemovenode 事件添加了事件处理函数。这个函数在调用的时候向其传递了几个有用的参数：

- 1, tree: 该事件所发生的 TreePanel;
- 2, node: 被移动的节点;
- 3, oldParent: 被移动节点之前的父节点;
- 4, newParent: 被移动节点的新的父节点;

5, index: 移动目的的序号。

我们利用以上这些来形成 **AJAX** 对服务器发请求时传递的参数。这样，你可以获得 **tree** 现在的任何信息和状态，你的服务端脚本可据此以完成任何你需要的动作。

在某些桩抗下，你可能需要取消拖放节点。当你在 **beforemovenode** 函数中出现了逻辑上的错误，你需要还原改动。如果你不发送 **AJAX** 请求，可以直接在函数的最后返回 **false** 即可，相应的动作将被取消。但是如果你采用了 **AJAX**，就困难了许多，因为 **XMLHttpRequest** 是异步发生的，而且这个事件函数会执行一些默认的动作，其中就包括允许节点移动。

即然这样，你需要确定你为 **AJAX** 提供了 **failure** 函数的配置，向这个函数传递足够的参数，好让树还原到以前的状态。因为 **beforemovenode** 通过默认传递的参数提供了大量的信息，所以你可以传递必须的数据来对产生的错误进行管理。

排序：

我们可以通过一种灵活的方法为 **TreePanel** 排序——**TreeSorter**。在先前代码的基础上，我们可以建立一个如下的 **TreeSorter**：

```
new Ext.tree.TreeSorter(tree, {  
  
    folderSort: true,  
  
    dir: "asc"  
  
});
```

因为 **TreeSorter** 采用了几个约定——**leaf** 节点由 **leaf** 属性标明并且标签文本在 **text** 属性里说明——我们可以很容易地按字母进行排序。**dir** 属性 **g** 奥素我们 **TreeSorter** 是按照升序还是降序进行排序的。**folderSort** 为 **true**（默认为 **false**）表明叶子节点需要在非叶子节点下进行排序，也就是说，整个树每层都要进行排序。

如果你的数据不仅仅是简单的文本，我们可以通过 **sortType** 配置项自定义排序。**sortType** 的值是一个函数，而且只向其传递一个参数：a **TreeNode**。

sortType 允许你为 **TreeNode** 新加自定义属性——这个属性可能是服务器传来的和业务相关的信息——并且把它转化成 **Ext** 可以排序的格式，换句话说，也就是转化为标准的 **JS** 类型，如整型、字符串、日期。**sortType** 一般用在原有的数据格式不能或者不方便做搜索以及排序的时候——从服务器返回的数据可能会被用于不同的目的，我们可以把日期转化为标准的格式——从 **US** 样式的 **MM/DD/YY** 转化为 **YYYYMMDD** 样式（以便排序）——或者我们要去掉货币中无用的符号从而转化为数字。

```
sortType: function(node) {  
  
    return node.attributes.creationDate  
  
}
```

在上面的这个例子中，我们返回了 `node` 里自定义的属性，因为这个属性的值是有效的 JS 日期，`Ext` 可以对其进行排序。这就是一个关于如何通过 `sortType` 选项让 `TreeSorter` 对任何服务器端数据进行排序的演示。

第八章 Tree(树) (2)

编辑:

很多时候，如果能编辑节点的值将是很有用的一个功能。在观察分类产品的层级结构时，你可能希望对产品的种类或者产品的名字做直接的修改，而不希望再去访问别的页面。那么你就可以通过 `Ext.tree.TreeEditor` 来实现这个功能。

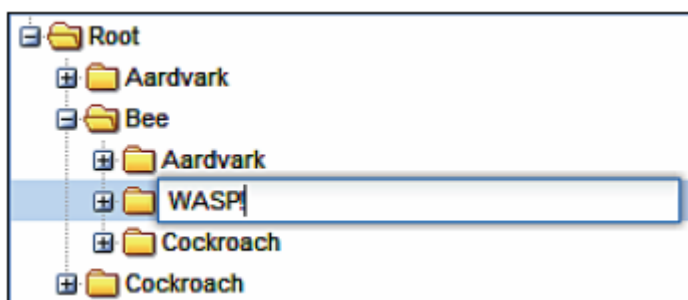
`TreeEditor` 默认在你双击节点标签的时候会为你的节点提供一个用于编辑的 `TextField`。但是，和 `drag-and-drop`（拖拽）一样，开启这个功能并不会自动向服务器端保存改动。你需要在编辑完节点后触发某个事件然后调用事件提供的函数来进行相应的操作：

```
editor.on('beforecomplete', function(editor, newValue, originalValue)
{
    // Possible Ajax call?
});
```

`beforecomplete` 事件函数有如下三个参数：

1. `editor`: 用来编辑这个节点的编辑字段；
2. `newValue`: 输入的值；
3. `originalValue`: 改动前的值。

然而，你需要注意的是 `editor` 这个参数不是一个普通的 `Ext.form.Field`。它有一些额外的属性，其中最有用的就是 `editNode`——对编辑节点的引用。你可以通过这个属性来获得节点的 `id`，这个属性在向服务器发请求来和数据库同步编辑的数据时很重要。



与 `TreePanel` 的 `beforemovenode` 事件相比，`beforecomplete` 可以让用户通过处理函数的结尾返回 `false` 来取消编辑改动。AJAX 请求需要提供 `failure` 处理函数来手动地还原先前的数值。

现在我们介绍了如何建立一个简单的节点编辑器，但这还意味着我们还需要完成一些更复杂的功能。`TreeEditor` 的构造函数里有两个可选的参数（一共三个，后两个

可选)。一个是 `field` 的配置对象，一个是 `TreeEditor` 的配置对象。`field` 的配置可以是一下两者之一：

- 标准 `TextField` 编辑器的配置对象；
- 已经建立的表单字段的实例。

如果是后者，你可以在此采用 `NumberField`，`DateField` 或者其他的 `Ext.form.Field`。

第二个可选参数可以让你配置 `TreeEditor`，只要一点小小的改动就能实现令人激动的功能。例如，我们可以使用 `cancelOnEsc` 来允许用户通过按下 `Esc` 键取消任何编辑；或者使用 `ignoreNoChange` 来避开编辑完成事件——如果值在编辑后并没有改变。

修剪树枝：

关于 `TreePanel`，还支持一些其它的技巧——改变 `selection model`（选择模型），过滤节点，以及显示背景菜单等。所以，让我们现在就学习它们。

Selection models（选择模型）：

在我们之前的示例代码里，我们可以通过拖拽和编辑 `TreeNodes` 来即时改变 `tree`。`TreePanel` 默认使用 `single-selection model`（单选模型）。我们之前的代码都需要去选择节点，但是对于 `tree` 来讲，简单地选择一个节点并不能满足所有的需求，所以我们需要知道控制选择方式的更多的方法和特性。

一个很好的例子就是，我们在选择一个节点以后，会自动产生一个信息面板来展示节点的细节。也许你有一个 `tree` 用来展示产品，点击一个节点需要显示该产品的价格和库存量。我们可以使用 `selectionchange` 事件来让这一切得以实现。我们还是使用先前的代码做开始，我们添加一下的代码：

```
tree.selModel.on('selectionchange', function(selModel, node) {  
  
    var price = node.attributes.price;  
  
});
```

`selectionchange` 事件的第二个参数 `node` 使得我们很容易获得自定义的节点属性值。

如果我们允许多节点选择呢？我们怎么去实现，我们怎么去处理 `selectionchange` 事件呢？我们可以使用 `Ext.tree.MultiSelectionModel` 来建立我们的 `TreePanel`：

```
var tree = new Ext.tree.TreePanel({  
  
    renderTo:'treeContainer',  
  
    loader: treeLoader,
```

```

        root: rootNode,

        selModel: new Ext.tree.MultiSelectionModel()

    });

```

配置就是这么简单。虽然 `selectionchange` 的处理方法和默认的 `selection model` 很相似，但是还是有重要的区别。第二个参数是节点的数组，而不是单一的节点。



`selection model` 并不只局限于获得与选择相关的信息。它还允许控制当前的选择。例如：`MultiSelectionModel.clearSelections()` 方法在你处理有关多节点事件之后可以帮助你清除所有的选中状态。`DefaultSelectionModel` 有方法（`selectNext` 和 `selectPrevious`）可以让你在 `tree` 中定位，在节点的层级中根据需求向上或者向下移动。

背景菜单（context menu）：

我们现在已经介绍了很多 `TreePanel` 提供的特性，所以让我们用一些练习来巩固一下。当你右击节点时弹出一个 `context menu`，这是一种界面上的“捷径”。我们使用的代码在之前的段落中已经介绍过，首先，让我们建立 `menu`，然后把它加到 `TreePanel` 中：

```

var contextMenu = new Ext.menu.Menu({

    items: [

        { text: 'Delete', handler: deleteHandler },

        { text: 'Sort', handler: sortHandler }

    ]

});

tree.on('contextmenu', treeContextHandler);

```

`TreePanel` 提供了一个 `contextmenu` 事件来监听右击节点。注意，我们的监听器不像之前的例子中一样采用匿名函数——为了让代码更易读。

首先，`treeContextHandler` 用来处理 `contextmenu` 事件：

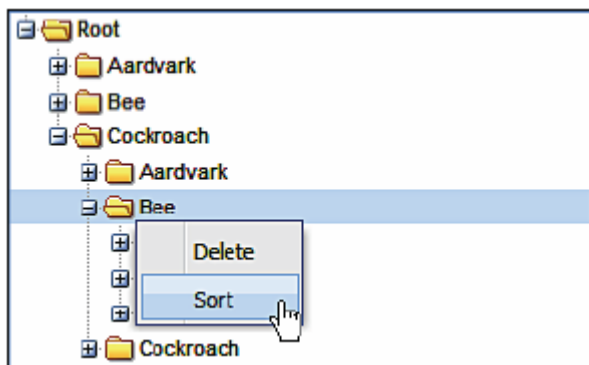
```
function treeContextHandler(node) {

    node.select();

    contextMenu.show(node.ui.getAnchor());

}
```

这个函数在调用时有一个 `node` 参数，我们需要在这个函数中先选择这个 `node`。然后我们用 `show` 方法弹出 `context menu`，这个方法有一个参数，用来说明弹出的位置。在这个例子里，我们的位置为 `TreeNode` 的文本处。



菜单处理：

这个菜单有两个入口——**Delete** 和 **Sort**，让我们看下 **Delete** 的处理函数：

```
function deleteHandler() {

    tree.getSelectionModel().getSelectedNode().remove();

}
```

利用我们之前的有关 `selection model` 的知识，我们可以获得在 `treeContextHandler` 中选择的节点，然后调用它的 `remove` 方法。这将从 `TreePanel` 中删除这个节点和它的子节点。注意，我们不把改变传递给服务器，但是如果你需要这么做，`TreePanel` 有 `remove` 事件，你可以用它的处理函数来提供这个功能。

Sort 处理函数如下：

```
function sortHandler() {

    tree.getSelectionModel().getSelectedNode().sort(

        function (leftNode, rightNode) {

            return (leftNode.text.toUpperCase() <
rightNode.text.toUpperCase() ? 1 : -1);

        }

    )

}
```

```
);  
  
}
```

再一次地，我们使用 `selection model` 来获得选中的节点。ExtJS 提供了 `sort` 方法，这个方法的第一个参数是一个函数，拥有两个参数：一对节点。在这个例子里，我们通过节点的 `text` 属性来进行降序排序，但是你可以利用别的自定义节点属性。

N: 使用这个方法和 `TreeSorter` 的排序不冲突，因为 `TreeSorter` 的排序只监听 `beforechildrendered`、`append`、`insert` 和 `textchange` 事件。其他的改动并不受影响。

`Delete` 动作将彻底地删除选择的节点，`Sort` 动作将依照 `text` 标签来对子节点进行排序。

过滤:

`Ext.tree.TreeFilter` 类在 ExtJS 2.2 中被冠以“试验”的标记，所以我只简要介绍。它用在用户想要根据节点某个特定的属性去检索节点时。这个属性可以是 `text`、`id` 或者其他在创建节点时自定义的信息。让我们利用之前的 `context menu` 来演示过滤功能，首先，我们需要建立 `TreeFilter`：

```
var filter = new Ext.tree.TreeFilter(tree);
```

你需要回到 `context menu` 的配置里，在 `items` 里为其添加一个新的入口：

```
{ text: 'Filter', handler: filterHandler }
```

我们现在需要建立一个 `filterHandler` 函数来实现 `filter` 动作：

```
function filterHandler() {  
  
    var node = tree.getSelectionModel().getSelectedNode();  
  
    filter.filter('Bee', 'text', node);  
  
}
```

像别的处理函数一样，我们先获得当前选中的节点，然后调用 `filter` 函数。这个函数有三个参数：

1. 被过滤的值
2. 过滤的属性（可选，默认为 `text`）
3. 开始过滤的节点

我们让被选中的节点作为开始过滤的节点，这意味着我们将对右击的产生菜单的节点的子节点通过特定的值做过滤。

我们的上面的例子（包含 `aardvark`、`bee`、`cockroach` 这些动物的例子）不需要做过滤，但是在某些状况中需要。线上的软件文档（多级且详细），采用过滤可以对标题迅速检索。你可以使用 `checkbox` 或者弹出对话框来让用户输入过滤的数据，这样会带来灵活的用户体验。

根：

虽然我们演示了很多 `Ext` 中树的功能，但是 `tree` 真正的强大之处在于它的设置、方法、各种类提供的加载点。我们已经了解了很多配置 `TreePanel` 和 `TreeNode` 的方法，让我们得以实现强大的功能。然而，还有很多的配置项可以加强我们的 `tree`，我们将会对一些有趣的配置进行说明。

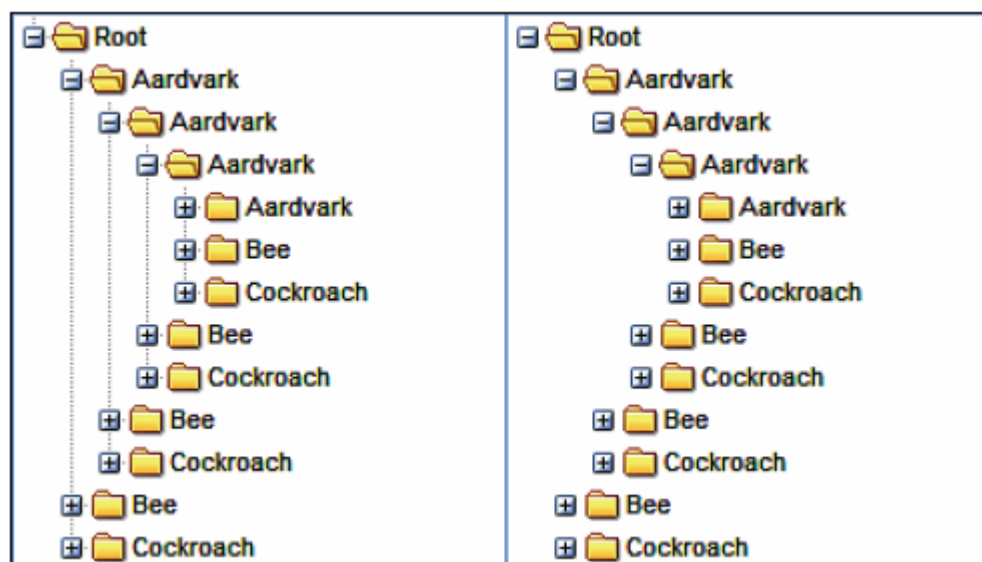
调整 `TreePanel`：

默认的，`TreePanel` 有很多图形界面上的加强配置，可以满足一些需求。例如，把 `animate` 设置为 `false` 可以阻止节点的展开和收拢的平滑的动画显示效果。这点在用户频繁展开关闭节点时很有帮助，可以放置动画效果带来的显示不畅。

因为 `TreePanel` 是扩展自 `Ext.Panel`，所以它支持所有标准的 `Panel` 的功能。它可以支持 `tbar` 和 `bbar`(顶部和底部工具栏)，`header` 和 `footer` 元素，收起/展开的功能。`TreePanel` 同样可以包含于 `Ext.ViewPort` 和 `Ext.layout` 之中。

装饰：

对于纯粹的装饰属性，`TreePanel` 提供了 `lines` 选项，当它被设置为 `false`，将把 `TreeNodes` 之间的层级导航线禁止掉。这点在形成简单的 `tree` 时很有用，可以放置界面的混乱。



`hlColor` 属性适用于 `drag-drop` 属性开启的 `tree`。它控制节点的颜色高亮（属性值为十六位数字串，例如 `990000`），当节点放下时被触发——可以通过设置 `dlDrop` 属

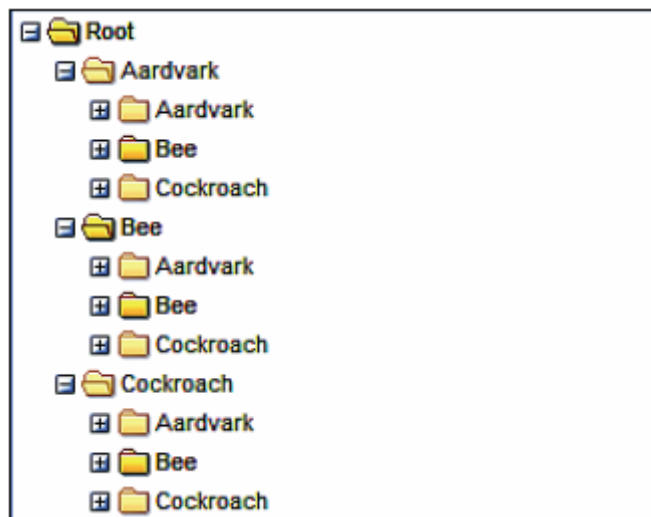
性为 `false` 来禁止。设置 `trackMouseOver` 属性为 `false` 可以禁止悬停在某个节点上时产生的高亮。

调整 `TreeNode`:

在很多情况下，你不能人工地建立 `TreeNode`（除了 `root` 节点），所以你可能想，那些配置选项对你来说没什么用。其实不是这样的，因为不只是 `id` 和 `text` 属性可以拿来创建节点——所有 `JSON` 中的 `TreeNode` 属性（满足配置属性要求的）都可以被用来建立节点。如果你有如下样子的 `JSON`:

```
[
  {
    text: 'My Node', disabled: true, href: 'http://extjs.com'}
]
```

你将获得一个开始为 `disabled` 的节点，但是一旦它设置为 `enabled`，它将成为到 `extjs.com` 的链接。



这个功能在传递应用的特殊信息的时候很有用。例如，你的服务逻辑要求你某些节点不能拥有子节点。这是你设置 `allowChildren: false` 就可以实现这样的功能（该节点不能作为 `drop` 的目标）。你还可以设置 `draggable: false`，这样你就可以组织某些节点的拖放。我们可以通过设置 `checked: true` 来使得某个节点拥有 `checkbox`（事实上，`checked` 不管设置为 `true` 或者 `false` 都起作用）。这些配置选项允许你规定节点的行为。

还有许多其它的和 `TreeNode` 相关的配置。你可以使用 `icon` 配置来提供自定义的图标，或者通过 `cls` 属性来提供 `CSS` 样式。`qtip` 选项允许你弹出 `tooltip`——可以是对节点的说明。

控制操作:

当 `TreePanel` 被配置后，我们可以操控它的节点。`TreePanel` 允许你在不同的层级之间“航行”，从选择的节点转移到父节点或者子节点，或者顺着当前的层级向上或者向下移动。我们也可以根据节点的路径选择或者展开节点，也可以以此找到某个节点。`expandAll` 和 `collapseAll` 方法可以使我们展开合起所有节点，可以被用来还原默认的状态。每个方法都有一个布尔值的参数，用来说明改变是否需要 `animate`（有动画效果）。

`expandPath` 方法的第一个参数是节点的 `path`(路径)。这个 `path` 唯一的利用层级确定了某个节点——例如：`/n-15/n-56/n-101`。

我们从这个路径可以知道，目标节点的 `id` 为 `n-101`，`n-15` 是 `root` 节点，`root` 节点有一个 `id` 为 `n-56` 的子节点。如果你了解 `XPath`，这个就很好理解。如果你不了解的话，你可以把她和 `IP` 地址做对比——提供了为一路径引用方式。

通过向 `expandPath` 传递这个参数，`tree` 可以定位到指定的节点，并且展开之。想象一下代码：

```
Ext.Msg.prompt('Node', 'Please enter a product name',

function(btn, text){

    if (btn == 'ok'){

        var path = GetNodePathFromName(text);

        tree.expandPath(path);

    }

});
```

`GetNodePathFromName` 函数可以查询服务器并且返回节点的 `id`，可以通过用户的输入快速定位节点。`TreePanel.getNodeById` 也可以通过类似的方式使用。不仅是展开节点，还可以进行进一步的控制。

在某些环境下，你可能需要做一些撤销动作，这是你需要获得节点的路径。`TreeNode.getPath` 提供了这样的功能，你可以用这个方法存储节点的位置。

更多的方法：

`TreeNode` 还有很多其他的方法。我们已经介绍了 `sort` 和 `remove`，但是现在我们可以添加一些基本应用程序的方法，如 `collapse` 和 `expand`、`enabled` 和 `disable`，`expandChildNodes` 和 `collapseChildNodes`（可以用来展开合起所有的子节点）。`findChild` 和 `findChildBy` 方法可以允许简单或者自定义的对节点的搜索，以下这个例子我们用来查找第一个 `price` 属性值为 300 的节点：

```
var node = root.findChild('price', 300);
```

在某些情况下，你可能需要对大量的节点属性进行操控，你可以使用 `TreeNode.eachChild` 方法：

```
root.eachChild(function(currentNode) {  
  
    currentNode.attributes.price += 30;  
  
});
```

因为第一个参数是一个函数，我们可以展示不同的需求带来的逻辑。

事件捕获：

我们已经演示了很多用户和 `tree` 交互的方法，但是还有很多有用的事件。先前，我们讨论了 `TreeNode` 的 `checked` 配置项的用法，当 `checkbox` 被勾选或者取消，`checkchange` 事件被激发。以下代码可以使得选中状态高亮显示：

```
tree.on('checkchange', function(node, checked) {  
  
    node.eachChild(function(currentNode) {  
  
        currentNode.ui.toggleCheck();  
  
    });  
  
}
```

我们让选中对 `TreeNode` 的子节点都生效，我们可以高亮显示节点来清楚地显示选中状态，或者实现其他的逻辑，例如在页面的任何位置展示新被选中的节点的信息。



`TreePanel` 事件的一个更通常的应用就是检查改动和与服务端同步改动。例如，一棵分类产品的树有着逻辑约束——某些特价商品需要表明最高价格。我们可以使用 `beforeappend` 事件来检测：

```
tree.on('beforeappend', function(tree, parent, node) {  
  
    return node.attributes.price <  
parent.attributes.maximumPrice;  
  
});
```

这个示例演示了 ExtJS 提供的一种模式——返回 `false` 来取消动作的执行。在这里，如果价格超过了最高价，就会返回 `false`，节点就不会被加入。

记录状态：

在很多应用中，`TreePanel` 是用来做导航的，展示层级结构，每个节点是一个 HTML 的链接。在这种场景里，如果用户希望观看多个页面，一个接着一个，默认的 `TreePanel` 的行为会导致混乱。因为 `tree` 在页面刷新时不会保持状态，当用户返回时，所有的展开的节点将被收起。所以，当用户需要经常定位他们感兴趣的页面时，往往会因此失去耐性。

StateManager（状态管理器）：

现在我们有一个管理 `TreePanel` 的好方法，可以方便地存储和还原状态。我们需要存储每个 `TreeNode` 的展开的状态，当页面重新加载的时候，我们可以重现展开的状态。我们可以使用 `Ext.state.Manager` 和它的 `CookieProvider` 来存储展开状态。我们可以这样初始化：

```
Ext.state.Manager.setProvider(new Ext.state.CookieProvider());
```

这是一个标准地设定 `state provider` 的方式，我们现在需要建立我们所需要存储的东西，我们选择存储节点的路径——这可以使得用户方便地访问他/她所感兴趣的最末端的节点：

```
tree.on('expandnode', function (node){ Ext.state.Manager.  
  
    set("treestate", node.getPath());  
  
});
```

在这段代码里，我们使用 `expandnode` 事件来记录路径（使用 `TreeNode.getPath` 方法）。这样 `treestate` 就包含了最后一次展开的节点。我们可以在页面加载的时候检查这个值：

```
var treeState = Ext.state.Manager.get("treestate");  
  
if (treeState)  
  
    tree.expandPath(treeState);
```

如果 `treestate` 之前被记录过，我们使用它来展开最后一次展开的节点。

警告：

需要强调的是，这个只是单纯地执行。它不会去管你在展开节点后是否合起来，因为合起节点的状态不会被保存。所以当你还原状态的时候，合起的节点又将展开。通过对 `collapsenode` 事件进行处理，我们可以解决这个问题。另外，对于展开多

节点也存在着问题。如果很多节点同时展开，只会展开最近展开的一个。存储一个展开节点的数组是一个好的解决办法。

总结：

用十一行代码展示一个丰富的 `Ext.tree.TreePanel` 留给了我们很深的印象。本章还进一步演示了 `TreePanel` 的强大功能——它为我们提供了丰富的配置选项。

异步加载是一个很重要的功能，因为它提供了展示大量动态信息的一种方式。在 `Ext.tree` 中这个功能得到了很好的处理，这意味着无论对于开发者还是最终用户都能得到好处。

尽管很强大，但是 `Ext.tree` 类仍然让人感觉是一种很轻量级的应用。使用配置选项以及各种 `TreePanel` 和 `TreeNode` 提供的方法、事件可以方便地驾驭这种“强大”（并不仅仅局限于这些类里）。`TreeSorter` 和 `TreeNodeUI` 也是解决问题的关键所在，提供了自定义显示和用户体验的功能。因为 `Ext.TreePanel` 拓展自 `Ext.Panel`，而 `Ext.Panel` 又是 `BoxComponent` 的拓展，我们可以获得强大的 `component`（组件）和 `layout`（布局）的支持，这意味这 `tree` 可以被轻松地包含于 `Ext.Window` 中，而这是我们下一章讨论的主题。