# 第1章 – ExtJS入门指南

前言

本来我是打算自己写一个系列的 ExtJS 6 学习笔记的，因为 ExtJS 6 目前的中文学习资料还很少。google 搜索资料时找到了一本国外牛人写的关于 ExtJS 6 的电子书。这份资料在 [PACKT](https://www.packtpub.com/all/?search=ext%20js%206)上卖 35.99 刀的，当然了万能的 google 还是帮我下载到了 PDF 文档。大概看了一下，讲的很详细，例子也比较简单，容易理解，现我准备利用工作之余翻译这份文档，为自己学习加深理解，也希望能帮助更多的人学习。

翻译时，我并不会 100% 按照原文照搬，而是可能加上我的理解和我自己的看法，所谓求其上而得其中，求其中而得其下，如果你看到这个文章，也不要以此作为标准，保持你的意见和思想。

正文

现在如果你打算学习 Ext JS 了，那你应该告别使用普通的 JavaScript 了。JavaScript 是一个伟大，卓越的语言，但是随着 web 应用越来越大，我们的代码也越来越难以维护，这是十分困难并耗费时间的。

普通的 JavaScript 非常的灵活，这有很多优点，同时这样也会带来很多问题，随着你的应用越来越大，参与的人就越来越多，普通的 JavaScript 让 10 个人来写，就会有 10 种不同的风格，往往有些代码只有写代码的当事人知道为什么这么写，如果这个人不再负责这个功能，就会成为后来人的坑。所以在代码上的约束和规范是十分有必要的。这也是为什么我一直喜欢 ExtJS 的原因，ExtJS 写起来代码就像 Java 一样，非常严谨，非常面向对象。缺点就是在前端 JS 库中属于比较重量级，学习曲线相对陡峭。

当涉及到 JavaScript 框架时，有客户端的 JavaScript 框架以及服务端的 JavaScript 框架。Ext JS 就是一个客户端的 JavaScript 框架。

在本章基本上是讲述关于如何使用我们所需的工具来设置开发环境以及Ext JS 的基本介绍。在这一章，我们主要涉及以下几点：

* 使用 Ext JS 的优点
* 对 Ext JS 的介绍
* 安装设置 Sencha Cmd 和 Ext JS
* 使用 SenchaCmd 搭建 Ext JS 应用
* Ext JS 应用的体系结构
* 探索 Sencha Cmd 命令
* 如何调试 Ext JS 应用
* 使用开发工具 IDE

为什么使用 Ext JS?

既然推荐你使用，自然要告诉你使用 Ext JS 都有哪些看得见的好处。现在让我们瞧一瞧，在你的应用里使用 Ext JS 相较普通 JavaScript 有哪些优点。

跨浏览器支持

只要你做 web 开发，你一定解决过浏览器兼容问题，这有多么消耗时间和影响工作进度你一定有体会，你可能会花上几小时甚至几天来解决跨浏览器 bug。你为什么不把这些时间用来关注你的业务功能呢？相反，如果使用 Ext JS 这个 JavaScript 框架，这种事情将由它去考虑，而你可以专注于业务功能。

丰富的 UI 组件

Ext JS 提供了大量丰富的 UI 控件，如 data grid ，tab panels,tree 控件，日期选择工具，图表 等等，Ext JS 作为一站式的前端框架，有很多优秀且实用的控件，尤其是表格对复杂数据展示的支持是其他的 JavaScript 框架不具备的， 这些都能大量节省你的开发时间。

双向数据绑定

双向数据绑定意味着当视图 view 的数据发生变化，你的模型 model 将自动更新。同样的，当你的应用更新这个模型 model，这个数据也将自动传递到 view。

就拿编辑页面来举栗。 当这个页面(view)加载，这时已经渲染了来自模型(model) 的数据到 HTML，并且当用户在页面(view)上更新这个数据，这时候你需要更新你的模型(model)，但如果你使用 Ext JS 将不需要你自己去做这些编程。

JavaScript 的架构模式(MVC/MVVM)

随着越来越多的代码转移到客户端，维护客户端的 JavaScript 代码变的非常困难。通过在客户端使用 MVC (Model View Controller)/MVVM (Model View, View Model)  使维护客户端的 JavaScript 变得更容易，并提高了生产效率。关于 MVC 和 MVVM 架构模式会在本章后面解释。

简化复杂的操作

想象一下使用纯 JavaScript 写一个 AJAX 调用，你需要考虑确保支持所有浏览器，这你可以看一下任何的 JavaScript 框架里进行 AJAX 调用的方法的源码。想象一下使用纯 JavaScript 创建一个复杂的控件，例如 grid ，分页，排序，过滤，分组，可编辑字段等等。Ext JS 不需要你自己来做这些工作。这里其实是废话，没有谁闲的蛋疼重复造轮子。但是其他框架一定没有 Ext JS 这种一站式的支持。

访问 DOM 变得容易

普通的纯 JavaScript 里，你可以访问 DOM，但是这有点复杂。使用 Ext JS 就会很容易查找 DOM。当然使用 jquery 的人肯定不信服，还能比 jquery 查找 DOM 更方便吗？我也常用 jquery ，这里我可以说 Ext JS 相较 jquery 查找 DOM 不逞多让。

客户端路由

在 web 应用里，URL 映射到 web 页面的路由方式背后有一系列的逻辑。路由可以在服务端也可以在客户端。通常客户端路由都是用单页式应用实现的。Single-Page Application (SPA)。

支持无障碍访问

无障碍访问特性意味着这个应用程序的内容必须是可以方便的让视力受损的人借助辅助设备访问(例如屏幕阅读器)。 而开发一个应用程序能够很好的支持无障碍特性是非常困难的。这里 Ext JS 已经为你做了。

在美国，如果你开发的软件供联邦和州政府职员使用，在绝大多数情况下，你需要确保你的程序是支持无障碍访问特性的。这一点，很少 JavaScript 框架能对无障碍访问特性有较好的支持。而 Ext JS 提供了完美的无障碍访问支持。

        World Wide Web Consortium (W3C)已经创立了一个名为WAI-ARIA (Web Accessibility Initiative – Accessible Rich Internet Applications) 的技术规范。

* WAI-ARIA, 是Web Accessibility Initiative – Accessible Rich Internet Applications 的缩写，指无障碍网页应用技术。她主要解决的一个问题:让残障人士能无障碍地访问网页上的动态内容。

这种规范方式使残疾人也能无障碍访问 web 应用。 Ext JS 对此具有良好的支持，并且所有的 控件/部件 都可以支持，不需要你做任何额外的编码。

介绍 Ext JS

Ext JS 作为一个一站式的开发富 UI 应用的框架，它提供支持 MVC,MVVM, 双向绑定，跨浏览器兼容，路由功能，一组广泛的丰富的 UI 组件，图表等等。对于框架中所有的 API ， Ext JS 还拥有一个非常优秀的文档。Ext JS 最初的建立是由 Jack Slocum 开发的，做为 YUI 的一个附加扩展库使用，现在则是成为 Sencha 公司的产品。

在 Ext JS 中，你写的代码基本上就是 JavaScript，因为你不需要写 HTML。Ext JS 附带一组庞大的丰富的 UI 组件，这在你的开发过程中会节约你相当多的时间。

本书中，所有的样例代码和示例项目代码将使用最新的 Ext JS Version 6，但是大多数代码是兼容以前的 Ext JS 5 的。Ext JS 5 和 Ext JS 6 的大多数概念是相同的。所以如果你在使用 Ext JS 5，你仍然可以从这本书中受益不少。但是请记住这本书的代码将不会运行在 Ext JS 5 并且可能需要做一些小修改才能运行在 Ext JS 5。在 Ext JS 6 中最重要的变化是它合并了两个框架：Ext JS 和 Sencha Touch 合并成为了一个框架。Ext JS 6 还带来了一个新的 SASS 编译器叫 Fashion，以及3D 图表的改进等等。

要明白为什么会发生合并 Ext JS 和 Sencha Touch , 我们需要回顾一下。

Sencha Touch 是一个独立的产品，专用于为移动设备和平板电脑创建可支持触摸的应用，它利用硬件加速技术为移动设备提供高性能的 UI 组件。

Ext JS 4 和 Ext JS 5 主要用于开发桌面级的 web 应用。 如果你已经在 Ext JS 5 或 Ext JS 4 创建了用于桌面的 web 应用，这仍然可以在移动设备和平板电脑上运行，但是它将不支持一些特定的触摸功能，并且不能利用硬件加速为移动设备提供高性能 UI 组件。所以为了更好的支持移动设备，Sencha 开发人员被告知要使用 Sencha Touch。

使用 Sencha Touch 有很多优势。 在 Sencha Touch 编写的应用将具备移动平台本地应用一样的外观，并且性能将更好。然而许多开发人员有一个抱怨,因为他们被迫保持两套相同的应用程序的代码库。

尽管 Sencha Touch 和 Ext JS 有很多差异是完全不同的产品，但他们的概念和思想的框架非常相似，如果你会用 Ext JS,那么学习 Sencha Touch 也非常容易。

长期以来,许多 Ext JS 和 Sencha Touch 用户问为什么不把两个产品合二为一使 Ext JS 能够带触摸功能。在 Ext JS 6 中,Sencha 决定将这两个产品合并到一个单一的产品。

现在，在 Ext JS 6 你仍然可以维护一套独立的代码。而要 Sencha Touch 和 Ext JS 6 两者兼存，有些视图的代码你可能需要单独的编写，但是大部分代码都是共享的。

在 Ext JS 6 合并后将两者通用的代码做为核心(两者公用一个核心)，并且引入了一个 toolkit(工具箱) 的概念。toolkit 是一个可视化组件的包，例如 button，panel 等等。Ext JS 6 有两个工具包：classic(古典) 和 modern(现代)。原来的 Ext JS 的可视化组件放在 classic 工具包，原来的 Sencha Touch 的可视化组件则放在 modern 工具包。

在 Ext JS 6 中选择你想使用的 toolkit(工具包)很简单，如果你正在开发的程序只是针对移动设备的，你可以选择 modern，如果你只用于桌面那么你可以选择 classic。默认则是通用的，通用的应用你在桌面端访问 Ext JS 6 应用，就会自动展现 classic 风格，如果用移动设备或平板电脑访问则自动展现 modern 风格。

通用应用

如果你的应用要同时支持桌面和移动设备，在 Ext JS 6 你可以创建一个通用的应用，这时候将会同时应用两个工具包。你可以添加以下的构建配置(这个配置在程序根目录 app.json 配置文件中，详细的后面会讲到)，来指定构建使用的工具包和主题：

1. "builds": {
2. //这里就很简单了，如果你只想用 classic 那么就注释 modern 的配置即可。
3. "classic": {
5. "toolkit": "classic",
7. "theme": "theme-triton"
9. },
11. "modern": {
13. "toolkit": "modern",
15. "theme": "theme-neptune"
17. }
19. }

这样做的大概路子就是，在 Ext JS 这个框架里用这两套工具包分别对应 桌面 和 移动设备。程序在构建过程中，会构建两套 UI 界面出来，而不是我们平时那种响应式的应用，响应式是一套 UI 自动根据不同设备改变布局。实际上在企业开发中，甚至互联网开发中，往往都是独立再开发一套支持移动设备的前端 UI，而 Ext JS 6 将此合并我认为是非常好的，这样你桌面和移动端都有一套标准，而且很多逻辑可以共享复用，不同的 view 写在独立的工具包里。维护起来也很方便。

如果你是一个 Ext JS 新手，看到这里肯定非常困惑，不过你现在不用担心，这些在后面都会有我们的样例代码来说明，会让你有更深的理解。

Ext JS 6 附带了两个主题包， classic 和 modern 主题包，这两个主题包里分别有各自支持的几种主题。

在 Ext JS 中的一些特定主题，提供了在 Windows，Android，iPhone 等设备上类似原生应用的外观。这些你将在第八章(主题和响应设计)学习。

设置 Ext JS 开发环境

你需要安装一个叫 Sencha Cmd 的工具，这会让你的 Ext JS 应用开发变得简单。它可用于Windows、Mac 和 Linux。

* Sencha Cmd 并不是开发 Ext JS 应用必须用的，但是它会让你的工作非常轻松，所以这里强烈推荐使用 Sencha Cmd。

Sencha Cmd

Sencha Cmd 是在 sencha 产品下做开发时使用的命令行工具，例如开发 Ext JS ,Sencha Touch 应用时都可以用。它通过许多的自动化任务帮助你提高生产力。它包含以下功能，包管理，JS 编译器，构建脚本，主题等等。

在安装 Sencha Cmd 之前，你需要安装 JRE 环境，如果你使用的是 Sencha Cmd 5，那么你还需要安装 Ruby。这里如何安装 JRE 或 JDK，还是也翻译一下，因为 Sencha Cmd 使用的是 apache ant 作为构建工具，所以需要安装 JRE 环境。而在 Sencha Cmd 5 编译 SASS 时则需要使用 Ruby，上面说过 Ext JS 6 使用 fashion 编译 SASS 了，下载最新的 Sencha Cmd 6 则不需要安装 Ruby了。

Java 运行时环境 (JRE)

在命令行窗口输入以下命令检查 java 是否能正常运行在你的设备上：

1. java -version

如果你已经在你的设备上安装了 java，那么你应该看到类似下面的代码；否则请下载安装 JRE 或 JDK：

java version “1.8.0\_25”

Java(TM) SE Runtime Environment (build 1.8.0\_25-b17)

Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)

这一步报错的话，检查一下你是否配置了 java 环境变量，不知道怎么配置请自行搜索。

Ruby

注意，如果你使用 Sencha Cmd 6 就不需要安装 Ruby，但是如果你正在使用 Sencha Cmd 5，那还是要安装 Ruby 的。输入以下命令，检查 Ruby 是否安装成功:

1. ruby --version

如果已经安装完成，应该看到类似下面的代码；否则请下载安装 Ruby：

ruby –version

ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86\_64-darwin14]

Ruby 也是需要配置环境变量的，这里不再介绍方法，不会的自行搜索。

安装 Sencha Cmd

现在从 Sencha 官网上下载并安装 Sencha Cmd 。安装完成后在命令行窗口输入以下命令检查是否正常运行(同样需要配置环境变量)：

1. sencha which

正常情况下应该显示类似以下的代码：

Sencha Cmd v6.0.0.92

/bin/Sencha/Cmd/6.0.0.92/

如果报错，你应该配置环境变量，在 Mac 电脑上运行以下命令添加安装路径到 PATH 变量：

1. export PATH=~/bin/Sencha/Cmd/6.0.0.92:$PATH

在 Windows，运行以下命令添加环境变量(这一步你其实可以略过的，我想能看这篇文章的人都知道怎么做)：

1. set PATH=%PATH%;C:\Sencha\Cmd\6.0.0.92

用 Sencha Cmd 生成第一个 Ext JS 应用

打开命令行窗口键入以下命令：

1. sencha generate app --ext MyApp ./myapp

运行上面的命令将会创建名为 MyApp 的 Ext JS 应用，应用所有的文件都放在当前目录下名为 myapp 的文件夹。

注意，上面的命令生成的 Ext JS 应用代码，包含两个工具包：classic 和 modern。因为你不明确指定需要用那个工具包的时候，默认创建的就是通用的应用。如果你需要指定使用 classic 或者 modern 工具包。那么用 –modern 或者 –classic 参数，如以下命令所示：

1. sencha generate app --ext --modern MyApp ./myapp

当你第一次运行这个命令时，这应该会自动下载 Ext JS 6。如果没有自动下载，那你需要手动的取下载 Ext JS 6，这里贴出来 GPL 协议的 Ext JS 6 官网下载地址 <http://cdn.sencha.com/ext/gpl/ext-6.0.0-gpl.zip>这里下载后解压，这时候生成 Ext JS 应用时就可以使用以下命令以指定 SDK 的形式生成了：

1. sencha -sdk /path/to/sdk generate app MyApp /path/to/myapp

Sencha Cmd 支持 Ext JS 4.1.1a 以及更高版本，支持 Sencha Touch 2.1 以及更高版本。在你的电脑里可以有多个版本的 SDK 。 上面的命令是基于一个特定的 Sencha SDK 来生成的 Ext JS 应用。

下面的例子，在目录 /projects/extjs/myapp 下生成名为 MyApp 的 Ext JS 应用：

1. sencha -sdk /bin/Sencha/ext/6.0.0/ generate app MyApp /projects/extjs/myapp

OK，现在可以查看已经创建的应用了，运行以下命令：

1. cd /projects/extjs/myapp
3. sencha app watch

这时会运行一些构建相关的任务，最终你将在命令行窗口看到类似下图的界面：

上面的命令监控了任意代码修改保存后，都会在浏览器刷新时反应最新的改动。

在浏览器输入默认 URL (http://localhost:1842)，如图所示，看到类似以下界面：

 默认当你使用电脑访问 URL (http://localhost:1842)，应用会自动检测并为你展示 classic 工具包的 UI。如果访问是来自一个移动端浏览器，它将展示 modern 工具包。如何在电脑上看 modern 风格的应用呢？附加参数  (http://localhost:1842?profile=modern)，你将看到以下截图：

MyApp 应用的整体目录结构如下图。我们瞧一瞧这个实例程序都有哪些比较重要的文件。

应用包含了 model, store, 和 application.js 。你就把 store 看成是一个 model 实例的集合。store 是为你的程序功能提供并加载数据用的，你可以认为 store 就是一个数据源，它支持排序，过滤，分页等等，经常用到 store 的就是 grid 组件。这些都会在后面一一讲到。

在下面截图中,有 classci 和 modern 这两个文件夹。这两个文件夹包含使用不同工具包时写的 view(视图)代码，例如你 classic 风格的 view(视图) 就应该写在 classic 目录下，modern 风格的 view(视图) 就应该写在 modern 目录下。

下图为 classic 和 modern 文件夹下的结构。classic 和 modern 都包含 src 目录，而你的 view(视图) 就应该写在  src 里面。

而这个 main.scss 文件是样式文件，在 classic 和 modern 工具包都存在，对应桌面和移动设备的样式。

还有就是在根目录也有一个 sass 文件夹，那里是放置应用所有设备共用的样式。

SASS (Syntactically Awesome Stylesheets)是一种样式语言。Ext JS 中大量的用到 SASS。你将在第8章(主题和响应设计)学习到更多关于样式的知识。

注意 classic 和 modern 目录，这里面都不是 Ext JS 框架的工具包源码，这里是程序的代码，classic 和 modern 两个工具包的源码在根目录 ext 文件夹里：

在下一节，我将给你介绍 MVC ，并且会在 MyApp 这个应用下使用 Sencha Cmd 生成这些 MVC 相关的文件以及学习它。

应用的体系结构

Ext JS 提供支持两种应用架构 MVC 和 MVVM。

Model(模型)

这代表着数据层。model 保存的数据可以包含数据验证和逻辑，model 经常用于 store 中。上文已经讲过，store 就是多个 model 的集合。

View(视图)

这一层就是用户界面。包含有 button，form，和 message box 等等组件。

Controller(控制器)

控制器处理 view(视图)相关的逻辑，例如 view 的 event(事件)处理，还有任何程序相关逻辑都可以写在这里。

View model (视图模型)

view model 封装了 view(视图)所需要的展示逻辑，绑定数据到 view 并且每当数据改变时处理更新。

我们看一下通过 Sencha Cmd 创建的 view，controller，view model 相关的文件。

如果你打开 app.js，你将看到下列代码，这是 Ext JS 应用的启动代码：

1. Ext.application({
2. name : 'MyApp',
3. extend : 'MyApp.Application',
4. requires : [
5. 'MyApp.view.main.Main'
6. ],
7. mainView : 'MyApp.view.main.Main'
8. });

在上面代码中，name 定义了程序的名称，extend 表示继承 MyApp.Application 类，这个类文件定义在 app 文件夹下名为 Application.js:

1. extend: ‘MyApp.Application’

requires 部分指定了这个类需要的类列表。这样在 requires 里面添加的类在当前类中首次实例化时，会去先加载它，你可以把 requires 的作用理解为 java 中的 import 关键字。mainView 指定的是要初始化创建的view(视图)。

继续，现在查看 app 文件夹，你将看到文件  Application.js，和 model， view， store 等等。

下面是 Application.js 文件里的代码：

1. Ext.define('MyApp.Application', {
2. extend: 'Ext.app.Application',
3. name: 'MyApp',
4. stores: [
5. // TODO: add global / shared stores here
6. ],
7. launch: **function** () {
8. // TODO - Launch the application
9. }
10. });

这里你可以看到 MyApp.Application 继承自Ext.app.Application。这个 launch 函数是在 Ext.app.Application 类里。这个函数将会在页面加载完成后调用。

这个 stores 是指定需要的 store 的，这里并没有用到，你将在后面的章节中学习到关于 store 的更详细的知识。

视图模型 – MainModel.js

现在看一下 \app\view\main\ 目录下的 MainModel.js 文件。这个类是 Main 视图的 view model(视图模型)。这个视图模型继承自 Ext.app. ViewModel，代码如下所示：

1. Ext.define('MyApp.view.main.MainModel',{
2. extend : 'Ext.app.ViewModel',
3. alias : 'viewmodel.main',
4. data : {
5. name : 'MyApp',
6. loremIpsum : 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'
7. }
8. });

控制器 – MainController.js

这个类是 Main 视图的控制器。在下列代码中你可以看到 onItemSelected 函数，这个函数将在视图里的 grid 中选中某一项时触发调用。这是什么原理呢？后面会讲到的。

1. Ext.define('MyApp.view.main.MainController', {
2. extend : 'Ext.app.ViewController',
3. alias : 'controller.main',
4. onItemSelected : **function**(sender, record) {
5. Ext.Msg.confirm('Confirm', 'Are you sure?', 'onConfirm', **this**);
6. },
7. onConfirm : **function**(choice) {
8. **if** (choice === 'yes') {
9. //
10. }
11. }
12. });

我们可以看到 extend 继承了 Ext.app.ViewController 这个类。Ext JS 中有两种类型的控制器： Ext.app.ViewController 和 Ext.app.Controller。

在后面的章节中你将会学习到这两种控制器有何区别。

视图 – Main.js

如果你用的是 Sencha Cmd 5,，并且你生成应用时执行了使用哪种 toolkits 工具包，是 –modern 或 –classic ，如果是这样那么 Main.js 文件就在 \app\view\main 文件夹下面。但是如果你用 Sencha Cmd 6 生成的是通用的应用，那么将会有两个 Main.js 文件，分别在 \modern\src\view\main\ 和 \classic\src\view\main\ 目录下面。

在查看这两个文件的内容之前，我先通过这两个不同路径的 Main.js 文件来解释一下。在本章的上文中，你已经知道为什么 Ext JS 6 合并了 Ext JS 和 Sencha Touch 为一个框架。这两个框架合并后共用一个核心，剩下的代码则分为两部分 classic 和 modern。传统的 Ext JS 代码移动到 classic 工具包，而 modern 的代码支持触摸和 HTML5 在 modern 工具包。所以这里需要两个工具包，程序会根据访问设备自动使用对应的工具包里的 UI 类(view)。

* 应用分两个工具包并共享核心资源和逻辑，这称为通用应用。

现在我们看一下在 modern 下的 Main.js 文件：

1. Ext.define('MyApp.view.main.Main', {
2. extend : 'Ext.tab.Panel',
3. xtype : 'app-main',
4. requires : [
5. 'Ext.MessageBox',
6. 'MyApp.view.main.MainController',
7. 'MyApp.view.main.MainModel',
8. 'MyApp.view.main.List'
9. ],
10. controller : 'main',
11. viewModel : 'main',
12. defaults : {
13. styleHtmlContent : **true**
14. },
15. tabBarPosition : 'bottom',
16. items : [{
17. title : 'Home',
18. iconCls : 'fa-home',
19. layout : 'fit',
20. items : [ {
21. xtype : 'mainlist'
22. } ]
23. }, {
24. title : 'Users',
25. iconCls : 'fa-user',
26. bind : {
27. html : '{loremIpsum}'
28. }
29. }, {
30. title : 'Groups',
31. iconCls : 'fa-users',
32. bind : {
33. html : '{loremIpsum}'
34. }
35. }, {
36. title : 'Settings',
37. iconCls : 'fa-cog',
38. bind : {
39. html : '{loremIpsum}'
40. }
41. }]
42. });

这个 Main 视图是一个 tab panel，因为它继承了 Ext.tab.Panel 。这个类有属性 controller， viewmodel，requires 配置了需要依赖的类。创建了四个 tab 页(items属性)，并且绑定了数据 ViewModel 里的loremIpsum 属性。你将会在后续章节中了解关于此更多的详细信息。

接着看一下在 \classic\src\view\main\ 下的 Main.js 文件内容：

1. Ext.define('NewApp.view.main.Main', {
2. extend : 'Ext.tab.Panel',
3. xtype : 'app-main',
4. requires : [
5. 'Ext.plugin.Viewport',
6. 'Ext.window.MessageBox',
7. 'NewApp.view.main.MainController',
8. 'NewApp.view.main.MainModel',
9. 'NewApp.view.main.List'
10. ],
11. controller : 'main',
12. viewModel : 'main',
13. ui : 'navigation',
14. tabBarHeaderPosition : 1,
15. titleRotation : 0,
16. tabRotation : 0,
17. header : {
18. layout : {
19. align : 'stretchmax'
20. },
21. title : {
22. bind : {
23. text : '{name}'
24. },
25. flex : 0
26. },
27. iconCls : 'fa-th-list'
28. },
29. tabBar : {
30. flex : 1,
31. layout : {
32. align : 'stretch',
33. overflowHandler : 'none'
34. }
35. },
36. responsiveConfig : {
37. tall : {
38. headerPosition : 'top'
39. },
40. wide : {
41. headerPosition : 'left'
42. }
43. },
44. defaults : {
45. bodyPadding : 20,
46. tabConfig : {
47. plugins : 'responsive',
48. responsiveConfig : {
49. wide : {
50. iconAlign : 'left',
51. textAlign : 'left'
52. },
53. tall : {
54. iconAlign : 'top',
55. textAlign : 'center',
56. width : 120
57. }
58. }
59. }
60. },
61. items : [ {
62. title : 'Home',
63. iconCls : 'fa-home',
64. items : [ {
65. xtype : 'mainlist'
66. } ]
67. }, {
68. title : 'Users',
69. iconCls : 'fa-user',
70. bind : {
71. html : '{loremIpsum}'
72. }
73. }, {
74. title : 'Groups',
75. iconCls : 'fa-users',
76. bind : {
77. html : '{loremIpsum}'
78. }
79. }, {
80. title : 'Settings',
81. iconCls : 'fa-cog',
82. bind : {
83. html : '{loremIpsum}'
84. }
85. } ]
86. });

上面代码中，items 中的代码几乎和 modern 工具包中的是一样的。此外，这个文件有些配置是专用于支持响应设计的。下列代码告诉框架使用的 ui  组件为 navigation：

1. ui: ‘navigation’

在第8章(主题和响应设计)中会介绍关于这个 UI 配置和响应设计的内容。

同样的，如果你打开 classic 或 modern 下的 List.js ，你会发现他们只有很细微的区别。

探索 Sencha Cmd 命令

现在让我们学习 Sencha Cmd 的一些非常有用的命令。

Sencha 命令格式

Sencha 命令采取以下格式：

sencha [category] [command] [options…] [arguments…]

在 Sencha Cmd 中许多命令和可选项。我们看一下都有哪些比较重要的命令。

Help

键入以下命令，你将获取一个 categories(类别)列表，一个顶层的 commands(命令)列表，一个可用的 options(选项)列表：

1. sencha help

获取一个特定类别的帮助信息，类别名称紧随在 help 后面，例如获取一个类别 app 的帮助信息，运行下列命令：

1. sencha help app

将产生以下输出：

如果你想进一步获取 app 命令下的子命令的帮助信息，你只需要在最后添加子命令例如 clean，如以下代码所示：

1. sencha help app clean

将产生以下输出：

升级 Sencha Cmd

如果你想检查是否有 Sencha Cmd 可用的更新，使用以下命令：

1. sencha upgrade --check

如果你要升级 Sencha Cmd，只需要移除 –check 选项，如以下代码所示：

1. sencha upgrade

升级 ExtJS 应用

升级一个现有的 ExtJS 应用到最新版本，需要先进入到你的 ExtJS 工程目录，使用一下命令：

1. sencha app upgrade 后面跟上 sdk 路径

不加路径的话，就会自动从官网下载最新版本的 ExtJS 框架。

注意：如果 sencha cmd 与要升级到的 ExtJS 框架版本不兼容，需要先升级 SenchaCMD 再升级 ExtJS。

生成一个应用

Sencha Cmd 支持 Ext JS 4.1.1a 及更高版本和支持 Sencha Touch 2.1 及更高版本。你电脑上可以存在多个版本的 SDK。这是基于 Sencha SDK 生成应用的命令格式，例如 Ext JS 或者 Sencha Touch：

1. sencha -sdk /path/to/sdk generate app [--modern/classic] MyApp /path/to/ myapp

这个示例代码将在目录 /Users/SomeUser/projects/extjs/myapp 下生成名为 MyApp 的 Ext JS 6 应用 ：

1. sencha -sdk /Users/SomeUser/bin/Sencha/Cmd/repo/extract/ext/6.0.0/ generate app MyApp /Users/SomeUser/projects/extjs/myapp

构建应用

运行下列命令将进行构建 HTML，JS，SASS 等等：

1. sencha app build

使用 Sencha Cmd 6 构建 Ext JS 6 应用，你还可以运行下列命令选择构建 moern 或 classic 风格的应用：

1. sencha app build modern
2. sencha app build classic

这里说一下，modern 和 classic 的构建配置在 app.json。 默认 Sencha Cmd 运行两个构建配置： classic 和 modern 。如果需要你也可以在 app.json 中添加额外的构建配置。

启动应用

watch 命令用于重新构建并启动应用。这不仅会启动应用程序,还监视任何代码更改，一旦代码改变，浏览器刷新将包括最新的代码：

1. sencha app watch

在 Sencha Cmd 6 和 Ext JS 6，你也可以运行下列命令选择 modern 或 classic：

1. sencha app watch modern
2. sencha app watch classic

代码生成

用Sencha Cmd，你可以生成 Ext JS 代码，例如 view，controller，model：

1. sencha generate view myApp.MyView
3. sencha generate model MyModel id:int,fname,lname
5. sencha generate controller MyController

当你生成 model 时如果不指定字段类型，默认类型是 string。

升级你的应用

Sencha Cmd 升级 SDK 的版本是很容易的。使用这个升级命令将你的程序升级到新框架：

1. sencha app upgrade [ path-to-new-framework ]

调试 Ext JS 应用

你可以使用浏览器默认的调试器来调试 Ext JS 代码，但是使用火狐浏览器的 firebug 插件再安装 Illumination 或者使用 Chrome 的 Inspector 插件调试这将会容易得多。

Illumination

Illumination 是一个第三方工具。它并不是 Sencha 的一个产品，目前它只支持火狐浏览器。

Illumination 的特性

这有些 Illumination 的特性，这将减少你在调试上花的时间。

对象命名

Illumination 会很容易识别出 Ext JS 组件，所以在 illumination 标签页你能看到 Ext JS 组件名称，例如 Ext.panel.Panel 而不是像在 firebug 的 DOM 页里那样显示成 Object。

Element 高亮

如果在 Illumination 窗口你鼠标悬停在任意对象上，将会突出高亮在 HTML 页面中的组件。

上下文菜单The contextual menu

一个 Ext JS 组件是由数个 HTML 元素组成的。如果你在页面右击选择使用 firebug 查看元素，你会看到元素是嵌套在 Ext JS 组件里，但是如果你选择 Illumination 来查看元素，会直接显示 Ext JS 组件，这更有便于检查组件的方法属性和事件。

firebug DOM 标签页里对象是如何展示的：

再看一下 Illumination 标签页下对象是如何展示的，你会发现所有组件都显示在下列截图中：

尽管 Illumination 使调试 Ext JS 应用变得容易，但是并不是必须用它。它并不是免费的，你不想购买的话，仍然可以使用 firebug 调试，但你也许会多花一些时间调试，或者使用 App Inspector 插件或者 Sencha Fillde 调试。然并卵，我还是建议使用 firebug 就行了。

App Inspector

App Inspector是一个由 Sencha 开发的免费的 Chrome 插件。它支持所有 Illumination 支持的功能。

相比使用 Illumination一些信息在 App Inspector查找更容易方便，并且使用 Illumination 比使用App Inspector 载入程序时间更长。

App Inspector 的截图：

Sencha Fiddle

这是另外的一个调试工具。这也是一个在线的基于 web 的 IDE 并提供了一些调试功能，如下图：

开发 IDE

尽管你可以使用任何简单的文本编辑器来编写 Ext JS 代码，使用 IDE 开发肯定更简单。Sencha 为JetBrains 提供 Sencha JetBrains 插件支持，例如  IntelliJ，WebStrome，PHPStorm，和 RubyMine。

如果你在寻找一些免费的 IDE，那么你可以看一下 Visual Studio Code 和 Brackets.io 。这两个都是非常轻量级，并且开源的文本编辑器，支持 Mac，Windows，Linux，下图是 Visual Studio Code

总结

在本章中, 我们看到了使用 JavaScript 框架相较于使用传统纯 JavaScript 的优势。也了解了 Ext JS 应用的体系结构，你也学习到如何配置你的开发环境以及如何用 Sencha Cmd 搭建 Ext JS 应用。

在下一章中, 您将了解Ext JS的核心概念和基本知识。

# 第2章 - 核心概念

在下一章我们会构建一个示例项目，而在这之前，你需要学习一些在 Ext JS 中的核心概念，这有助于你更容易理解示例项目。这一章我们将学习以下知识点：

* 类系统，创建和扩展类
* 事件
* Ext JS 对象的查询
* 容器
* 布局

class system(类系统)

Ext JS 提供了很多功能，使得它创建和处理类变得简单。以下是在 Ext JS 6 的类系统中的几大组成类：

* Ext
* Base
* Class
* ClassManager
* Loader

Ext 类

Ext 是一个全局单例的对象，在 Sencha library 中它封装了所有的类和许多实用的方法。许多常用的函数都定义在 Ext 对象里。它还提供了像其他类中一些频繁使用的方法的快速调用。

我们看一下在 Ext 类中定义的方法和属性：

application 方法

这里应用是用 Ext.application 方法初始化的。这个方法的参数是一个 Ext.app.Application 对象，这个方法会加载 Ext.app.Application 类，并在页面加载完成后开始应用给定的配置。

Ext.app.Application 这个类代表我们的整个应用，这在第1章(入门指南)讲过，讲过的吧？是吧？下面是 Ext.app.Application 的使用例子：

1. Ext.application({
2. name: 'MyApp',
3. extend:'MyApp.Application',
4. launch: **function**() {
6. }
7. }) ;

上面代码创建一个名为 MyApp 的全局变量。我们的应用里所有的类都将归属于在这样一个命名空间下面。这将降低全局变量产生冲突的可能。

define 方法

你可以用这个方法定义或者重写一个类。 这个方法有三个参数，如以下代码所示。 在这里 name 参数是你要定义的类名，data 参数是应用于这个类的属性，callback 是可选参数，这个函数将会在这个类被创建后调用：

1. Ext.define(name,data, callback)

下列代码创建一个名为 Car 的类：

1. Ext.define('Car', {
2. name: **null**,
3. constructor: **function**(name) {
4. **if** (name) {
5. **this**.name = name;
6. }
7. },
8. start: **function**() {
9. alert('Car started');
10. }
11. }) ;

你还可以使用 define 继承扩展一个类：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53893908) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53893908)

1. Ext.define('ElectricCar', {
2. extend: 'Car',
3. start: **function**() {
4. alert("Electric car started");
5. }
6. }) ;

如果你想替换一个父类方法的实现，你可以使用 Ext.define 来重写这个方法，如以下代码所示：

1. Ext.define('My.ux.field.Text', {
2. override: 'Ext.form.field.Text',
3. setValue: **function**(val) {
4. **this**.callParent(['In override']);
5. **return** **this**;
6. }
7. });

细心的同学可能发现了当我们继承和重写时使用的属性是不同的，继承我们使用 extend 而重写使用 override ，这两者之间有什么区别呢？你一定感到疑惑，这里我非常有必要给你解释清楚，听我慢慢道来。

首先说继承并扩展一个类，这等同于是一个新的类，仍然可以在这个新的类里增加自己独有的方法和属性或者重写父类的方法。

1. Ext.define('MyApp.view.main.Test', {
2. extend: 'Ext.grid.Panel',
3. xtype: 'maintest',
4. title: 'Personnel',
5. say:**function**(){
6. alert(123);
7. }
8. });

这里我继承了 gridpanel 类，并增加了一个 say方法，以下是输出调用 say 方法的运行结果。

这应该很好理解，Test 继承了 grid panel 之后是一个新的类了，而这里如果创建 Ext.grid.Panel 对象是调用不了 say 方法的。

那么现在我把 extend 改为 override 我们再看一下：

1. `Ext.define('MyApp.view.main.Test', {
2. override: 'Ext.grid.Panel',//改为 override 了
3. xtype: 'maintest',
4. title: 'Personnel',
5. say:**function**(){
6. alert(123);
7. }
8. });
9. Ext.define('MyApp.Application', {
10. extend: 'Ext.app.Application',
12. name: 'MyApp',
13. requires: [
14. 'MyApp.view.main.Main',
15. 'MyApp.view.main.Test'//我这里引入了 Test
16. ],
17. stores: [
18. // TODO: add global / shared stores here
19. ],
21. launch: **function** () {
22. **var** test = Ext.create("MyApp.view.main.Test",{
23. title: 'Personnel'
24. });
25. test.say();
26. },
28. onAppUpdate: **function** () {
29. Ext.Msg.confirm('Application Update', 'This application has an update, reload?',
30. **function** (choice) {
31. **if** (choice === 'yes') {
32. window.location.reload();
33. }
34. }
35. );
36. }
37. });

运行结果：

我们可以看到我只是简单的把 extend 替换成 override 就报错说不能识别的类名。但是上面我也是引入了它的引用的(requires)，可见 extend 和 override 还是有区别的，我们不是重写(override)的是 grid panel 吗？那我们试试创建一个 Ext.grid.Panel 对象试试。

1. Ext.define('MyApp.Application', {
2. extend: 'Ext.app.Application',
4. name: 'MyApp',
5. requires: [
6. 'MyApp.view.main.Main',
7. 'MyApp.view.main.Test'
8. ],
9. stores: [
10. // TODO: add global / shared stores here
11. ],
13. launch: **function** () {
14. //这里改成了 grid panel
15. **var** test = Ext.create("Ext.grid.Panel",{
16. title: 'Personnel'
17. });
18. test.say();
19. },
21. onAppUpdate: **function** () {
22. Ext.Msg.confirm('Application Update', 'This application has an update, reload?',
23. **function** (choice) {
24. **if** (choice === 'yes') {
25. window.location.reload();
26. }
27. }
28. );
29. }
30. });

再次运行结果：

这次正常了，我们使用 Ext.define 定义的类，使用了 override 重写组件，发现并不能新建我们定义的类，而是被重写的父类被新增了 say 方法。

所以总结一下，extend 会创建一个新的类，并继承父类的属性和方法，你也可以重写父类的方法。而 override 并不会创建一个新的类，而是修改这个被重写的父类。

* 注意：例如上面的例子，Test 重写了 grid panel，我在其他类中创建 grid panel 时如果不引用 Test 那么重写仍然是不生效的。只有引用了 Test 重写才会生效 grid panel 才会具有 say 方法。不要问我为什么。

 如果你想创建一个单例类，那么你在定义类时用 singleton 属性，如以下代码所示：

1. Ext.define('Logger', {
2. singleton: **true**,
3. log: **function**(msg) {
4. console.log(msg);
5. }
6. }) ;

create 对象

你可以使用下列代码来创建一个类的实例：

1. Ext.create(Class,Options);

下列代码创建了一个 ElectricCar 类的实例，并传递一个值(name)：

1. **var** myCar = Ext.create('ElectricCar',{ name: 'MyElectricCar' }) ;

如果 Ext.Loader 是开启的，Ext.create 执行时如果 ElectricCar 类不存在将会自动的下载对应的 JS 文件。默认 Ext.Loader 是开启的；你可以通过以下方式来关闭它。

1. Ext.Loader.setConfig({
2. enabled: **true**
3. });

这里你也可以使用 new 关键字来创建一个实例，如以下代码所示；可是如果这个类不存在，使用 new 关键字创建实例并不会自动下载对应的 JS 文件：

1. **var** myCar = **new** ElectricCar('MyElectricCar');

* ps：你只需要掌握使用 Ext.create 创建实例就行了，new 关键字可能是兼容 Ext 3.x 的使用方式而继续支持的。new 关键字官方是不推荐用的。

onReady

这个函数在页面加载完成后调用：

1. Ext.onReady(**function**(){
2. **new** Ext.Component({
3. renderTo: document.body,
4. html: 'DOM ready!'
5. });
6. }) ;

大多时候，在你的代码里不会用到 onReady 这个方法，因为 Ext 建议你一个应用就是一个页面(单页式应用)，只有一个页面的话自然没有那么多场景会需要用到。只有在极少数的一些特殊情况，你可能需要用它。这里要强调一点，如果你具有使用 jQuery 的基础，不要把 onReady 方法像 jQuery 中的 $( document ).ready() 那样频繁使用。

widget (部件)

当定义一个类时，你可以为这个类增加一个便于记忆的别名。例如： Ext.panel.Panel 的别名为 widget.panel 。定义别名时，如以下代码所示指定 alias 属性：

1. Ext.define('Ext.panel.Panel', {
2. extend: 'Ext.container.Container',
3. alias: 'widget.panel'//这里是定义的别名，
4. });

你也可以使用 xtype 为这个类给定一个别名。这个 xtype 是非常有用的，当你以指定 xtype 的方式应用部件时，并不会创建实例，而是在真正调用展示的时候才会创建这个类的实例。在本章后面介绍 容器和布局 时你将会学到更多关于 xtype 的使用。

Ext.widget 方法是通过类的 xtype 快速创建部件的。

例如，不使用 widget 方法，你可以通过下列代码创建 Ext.panel.Panel 的实例：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. title: 'Panel'
4. });

反之，通过以下方式快速创建 panel 的实例：

1. Ext.widget('panel', {
2. renderTo: Ext.getBody(),
3. title: 'Panel'
4. });

这里 panel 是一个容器，关于 panel 会在后面详细讲解。下面代码的作用相当于与上面：

1. Ext.create('widget.panel', {
2. renderTo: Ext.getBody(),
3. title: 'Panel'
4. });

注意： 你阅读此文档的同时，这里面的大部分代码都是可以运行的，你可以选择在你本地设备上或者在 Sencha Fiddle 上执行这些示例代码。你可以访问 Sencha Fiddle 并将上面的代码键入到 launch 函数中，运行并查看结果。如果你访问了 [https://fiddle.sencha.com](https://fiddle.sencha.com/) 将会看到下列代码：

1. Ext.application({
2. name : 'Fiddle',
3. launch : **function**() {
4. Ext.Msg.alert('Fiddle', 'Welcome to Sencha Fiddle!');
5. }
6. }) ;

现在，粘贴创建 panel 部件的代码如以下示例，运行并查看结果。复制粘贴时，注意单引号不要写成中文标点单引号：

1. Ext.application({
2. name : 'Fiddle',
3. launch : **function**() {
4. Ext.create('widget.panel', {
5. renderTo: Ext.getBody(),
6. title: 'Panel'
7. });
8. }
9. }) ;

* 不是所有的代码都可以这样运行，此外并非所有的示例代码都会有视觉呈现。

getClass

如果创建的实例是用 Ext.define 定义的，那么返回这个给定对象的类，否则返回 null：

1. **var** button = **new** Ext.Button();
2. Ext.getClass(button); // returns Ext.Button

 getClassName

通过它的引用或实例返回类名称：

1. Ext.getClassName(Ext.Button); //returns "Ext.Button"

Ext.Base

这是所有 Ext 类的基础。所有的 Ext 类都继承自 Ext.Base。该类所有的原型和静态成员都会传递给继承它的类。

Ext.Class

这是一个低级别的工厂类，用于通过 Ext.ClassManager 定义一个类。所以不应该在你的代码中直接访问；你应该使用 Ext.define。

Ext.ClassManager

它管理所有的类同时处理类反射。通常通过下面几个方法访问：

* define
* create
* widget
* getClass
* getClassName

在本章我们已经讨论使用过这些方法。

Ext.Loader

用于动态的加载依赖。通常使用 Ext.require 来指定依赖。当你定义一个类时，这样指定组件的依赖列表是一个很好的做法，如以下代码所示：

1. Ext.require(['widget.window', 'layout.border','Ext.data.Connection']);

如果你需要引入一个指定命名空间下所有的 组件/类 时，使用通配符，如以下代码所示：

1. Ext.require(['widget.\*', 'layout.\*', 'Ext.data.\*');

用以下语法排除掉不需要的类：

1. Ext.exclude('Ext.data.\*').require('\*');

用这种方式，依赖的类是异步加载的。如果在你定义的类中没有指定依赖的类，那么当使用 Ext.Create 创建实例时，如果它是未加载的，这时将会同步加载这些类文件。这对性能有一定的影响，所以当你定义类时，使用 Ext.require 指定所需的类总是更好的。

* requires 属性加载需要的类时，当前类初始化之前被加载。
* uses 属性加载需要的类时，当前类初始化之后被加载。
* singleton:true 属性当前类初始化时,该实例是一个单例对象。

注意：定位类的文件路径是基于类名的。例如：MyApp.view.About 类的路径应该是 \myapp\view\ about.js 。

Events(事件)

一个事件可以是一个用户操作，一个 Ajax 调用的响应等等。

Adding listeners(添加事件监听)

当你创建对象或者创建以后都可以为这个对象添加监听器。下列示例代码为这个对象添加了一个 单击事件 的监听：

1. Ext.create('Ext.Button', {
2. renderTo: Ext.getBody(),
3. listeners: {
4. click: **function**() {
5. Ext.Msg.alert('Button clicked!');
6. }
7. }
8. }) ;

你可以添加多个事件监听，如以下代码示例：

1. Ext.create('Ext.Button', {
2. renderTo: Ext.getBody(),
3. listeners: {
4. mouseout: **function**() {
5. //Do something
6. },
7. click: **function**() {
8. // Do something
9. }
10. }
11. });

你也可以在对象创建之后，使用 on 方法为对象添加事件监听：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53893908) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53893908)

1. **var** button = Ext.create('Ext.Button');
2. button.on('click', **function**() {
3. //Do something
4. }) ;

同样的，你也可以使用 on 方法一次添加多个事件的监听，如以下代码示例：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53893908) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53893908)

1. **var** button = Ext.create('Ext.Button');
2. button.on({
3. mouseover: **function**() {
4. //Do something
5. },
6. mouseover: **function**() {
7. //Do something
8. }
9. }) ;

Removing listeners (删除事件监听)

你也可以移除事件监听器, 但是你必须有 事件处理函数的引用; 你不能用匿名函数

1. **var** HandleClick= **function**() {
2. Ext.Msg.alert('My button clicked!');
3. }
5. Ext.create('Ext.Button', {
6. listeners: {
7. click: HandleClick
8. }
9. }) ;
11. button.un('click', HandleClick);

页面 DOM 元素的事件处理

你可以将监听器添加到 DOM 元素，如下所示。

假设在你的 HTML 代码中，有一个 div 元素 id=mydiv ，如以下代码所示：

1. **<div** id="mydiv"**></div>**

用下列代码为它添加事件监听：

1. **var** div = Ext.get('mydiv');
2. div.on('click', **function**(e, t, eOpts) {
3. // Do something
4. });

访问 DOM

有三种方法来访问 DOM 元素：get，query，和 select 。

Ext.get

get 方法是根据这个 DOM 元素的 ID 检索获取并封装为 Ext.dom.Element 对象：

1. **var** mydiv = Ext.get('myDivId');

Ext.query

这种方式基于传入的 CSS 选择器 从给定的根节点开始查找。它返回一个匹配选择器的元素(HTMLElement[]/Ext.dom.Element[])数组。如果没有匹配的，返回一个空值的数组对象。

在下面示例中，myCustomComponent.getEl().dom 是传递的根节点。Ext.query 将检索这个节点内的子元素，并返回一个数组包含 CSS class 为 ‘oddRow‘ 的的元素：

1. **var** someNodes = Ext.query('.oddRow', myCustomComponent.getEl().dom);

Ext.select

给出一些 CSS/XPath 选择器，Ext.select 方法返回一个 CompositeElement 类型的对象，代表一个元素的集合。

这个 CompositeElement 对象可以进行过滤，迭代，和对整个集合执行整体操作等等：

1. **var** rows = Ext.select('div.row'); ////Matches all divs with class
2. row rows.setWidth(100); // 这是设置所有元素的宽度为 100

你也可以用一行代码，如下所示：

1. Ext.select('div.row').setWidth(100);

多重选择器

在方法调用时通过指定多个搜索条件可以用来匹配多个元素：

1. Ext.select('div.row, span.title'); //匹配所有的 class 用 .row 的 div 元素，和匹配所有 class 用 .title 的 span 元素

选择器 根

当你使用 select ，它默认取 HTML body 作为根并从默认的 body 开始检索整个 DOM 树。你可以通过制定一个根元素来避免这种情况，这样它将只搜索给定的根的子节点。

1. Ext.get('myEl').select('div.row');

这儿使用了 ‘myEl’ 作为根节点。这将首先找到 id 为 ‘myEl’ 的元素，然后将在根元素(myEl)下面搜索出 class 为 ‘row’ 的 div 标签。

1. Ext.select('div.row', **true**, 'myEl');// This is equivalent to the previous line.

链式选择器

下列的查询方式会匹配 class 为 row 并且 title 属性值为 bar 的 div ，这个 div 属于其父元素的首个子元素：

1. Ext.select('div.row[title=bar]:first')

Ext.ComponentQuery

这允许你用 ID，xtype，和 属性查找一个组件。你可以全局搜索或者指定一个根组件。

下列查询将返回所有的 xtype 为 button 的组件：

1. Ext.ComponentQuery.query('button');

得到一个 id 为 foo 的组件，用以下代码：

1. Ext.ComponentQuery.query('#foo');

下列代码将返回所有的 xtype 为 button 并且 title 属性值为 my button 的组件：

1. Ext.ComponentQuery.query("button[title='my button']");; //or parent.query('textfield[title=my button]');

你也可以使用嵌套选择器如下：

1. Ext.ComponentQuery.query('formpanel numberfield'); // 这里获取 xtype 为 frompanel 下面的 xtype　为 numberfield 的组件

下列代码返回这个 parent 容器内匹配传递进来的选择器的第一个直接子组件，如果没有匹配上，返回 null 。

1. parent.child('button[itemId=save]');

同样的，你也可以使用其他的方法，例如 nextNode, up, down, previousSibling 等等。

组件，容器，和布局

Ext JS 提供了一组丰富的组件和布局，这使在 Ext JS 中开发 UI 变得超级简单，甚至非专业 UI 开发人员也能够轻易的使用。

组件

从简单的组件说起，例如 button 和 label ，到复杂的组件，例如 Tree Panel，Grids 等等，Ext JS 有大量的内置组件。所有的组件都派生自 Ext.Component 类，它提供支持创建，重绘，渲染和处理组件。

所有的组件都有一个属性叫做 xtype 。它是非常有用的，它用在当你不想马上实例化这个组件时，而是想让这个组件在实际被应用时才创建，就是我们俗称的懒加载。

容器

容器是一个特殊的组件类型，它能够持有其他组件。在 Ext JS 中 Ext.container.Container 类是所有的容器的基础类。

Ext.toolbar.Toolbar, Ext.panel.Panel,  和 Ext.Editor 是一些内置组件。这些组件都是可以包含其他组件。而像 Ext.button.Button 类就不是派生自 Ext.container.Container ，所以它不能够包含其他组件。

一个典型的 Ext JS 应用包含一组嵌套的组件。看下面这个例子并思考：

1. Ext.create('Ext.panel.Panel', {
2. renderTo:Ext.getBody(),
3. width:700,
4. height:400,
5. items:[{
6. xtype: 'panel',
7. title: 'Panel 1',
8. },{
9. xtype: 'panel',
10. title: 'Panel 2',
11. height: 200,
12. items: [{
13. xtype: 'button',
14. text: 'Click Me'
15. }]
16. },{
17. xtype: 'panel',
18. title: 'Panel 3',
19. width: 150,
20. height: 100
21. }]
22. });

在前面的代码中，这是嵌套的组件，结构如下图所示：

上面的代码运行后将输出类似以下截图：

布局

布局定义了包含的组件是如何定位的以及设定组件的尺寸大小。每一个容器都有一个布局。默认布局是 auto 。这将不会为子组件指定任何关于位置和大小的规则。

在上面的图中，你可能已经注意到这些子组件只是一个接一个嵌套在父级容器中。这是在代码中因为我们还没有为这些组件制定任何布局，默认情况下使用的是 auto 布局。

现在，让我们在相同的代码里使用一些布局。下列示例中，我们将使用 column 布局和 center 布局。

当你使用 column 布局，你可以指定 columnWidth 。所有的列的 columnWidth 的值的总和必须等于 1 。你也可以为一些列指定固定宽度，如以下代码所示。这里，Panel3 取了一个 150 的固定宽度，然后剩下的两列按照 columnWidth 的值分配剩下的宽度：

1. Ext.create('Ext.panel.Panel', {
2. renderTo:Ext.getBody(),
3. width:700,
4. height:400,
5. layout:'column',
6. items: [{
7. xtype: 'panel',
8. title: 'Panel 1',
9. columnWidth: 0.4,
10. height: 400,
11. },{
12. xtype: 'panel',
13. title: 'Panel 2',
14. columnWidth: 0.6,
15. layout: 'center',
16. height: 400,
17. items: [{
18. xtype: 'button',
19. text: 'Click Me'
20. }]
21. },{
22. xtype: 'panel',
23. title: 'Panel 3',
24. width: 150,
25. height: 400
26. }]
27. });

以上代码输出为：

updateLayout

updateLayout 是  Ext.container.Container 对象里的一个方法。这可以用来根据布局规则重新定位子组件。例如你修改了布局方式，需要动态的更新布局时。

suspendLayout

大多数时候你不会用到这个 updateLayout  方法，然而有些时候你必须调用它。

这个 updateLayout  方法是在你重绘和当你添加或删除了一个组件时自动调用。有时候你可能需要它暂停一下，不是马上就调用，特别是当你添加或删除多个子组件时。所以在这种情况下，你可以设置suspendLayout  属性为 true ，一旦你完成添加或删除组件的操作，你可以设置 suspendLayout 为 false 并在你的代码中手动调用 updateLayout 方法。

同样的如果你想对整个框架停止更新布局，你可以调用 Ext.suspendLayouts() ，然后在你的操作完成后你可以通过调用 Ext.resumeLayouts(true) 恢复它。

以下是 Ext JS 中可用的布局：

* absolute
* accordion
* anchor
* border
* card
* center
* column
* fit
* hbox
* table
* vbox

absolute 绝对布局

这个布局使用 x 和 y 属性来指定组件的绝对定位：

1. Ext.create('Ext.panel.Panel', {
2. renderTo:Ext.getBody(),
3. width:700,
4. height:400,
5. layout:'absolute',
6. items: [{
7. xtype: 'panel',
8. title: 'Panel 1',
9. x: 12,
10. y: 20,
11. height: 250
12. },{
13. xtype: 'panel',
14. title: 'Panel 2',
15. x: 200,
16. y: 150,
17. height: 200
18. },{
19. xtype: 'panel',
20. title: 'Panel 3',
21. x: 400,
22. y: 250,
23. width: 150,
24. height: 100
25. }]
26. });

 这里所示的输出，你可以重叠组件因为他们用绝对位置定位的：

accordion 手风琴(可折叠)布局

这个布局展示了在一个时间里只有一个内置的可支持折叠和展开的子级 panel 。瞧一下以下示例：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout: 'accordion',
6. items: [{
7. title: 'Item 1',
8. html: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum'
9. },{
10. title: 'Item 2',
11. html: 'some content here'
12. },{
13. title: 'Item 3',
14. html: 'empty'
15. }]
16. });

这里显示的输出，这个 Item 1 是展开的，而其他的 panel 是折叠的：

anchor 锚点布局

这个布局使你能够指定子级组件的大小，而这是相对于布局容器的。首先容器根据指定的锚点规则调整，然后所有的子级组件再作调整：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout: 'anchor',
6. items: [{
7. title: 'Item 1',
8. html: 'Item 1',
9. anchor: '50%'
10. },{
11. title: 'Item 2',
12. html: 'Item 2',
13. anchor: '-20 -200'
14. },{
15. title: 'Item 3',
16. html: 'Item 3',
17. anchor: '-200'
18. }]
19. });

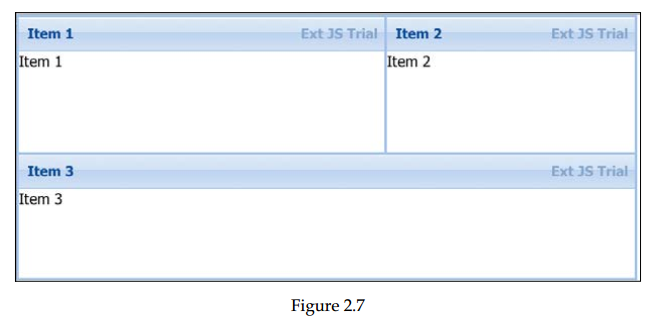
输入如以下截图：

border 布局

这个布局允许你为子组件指定一个区域位置，例如 center，north，south，west 和 east。当你使用 border 布局时，在其内的组件必须有一个指定区域为 center，如下列代码所示：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout: 'border',
6. items: [{
7. title: 'Item 1',
8. html: 'Item 1',
9. region: 'center'
10. },{
11. title: 'Item 2',
12. html: 'Item 2',
13. region: 'east',
14. width: 200
15. },{
16. title: 'Item 3',
17. html: 'Item 3',
18. region: 'south',
19. height: 100
20. }]
21. }) ;

以上代码输出类似下列视图：

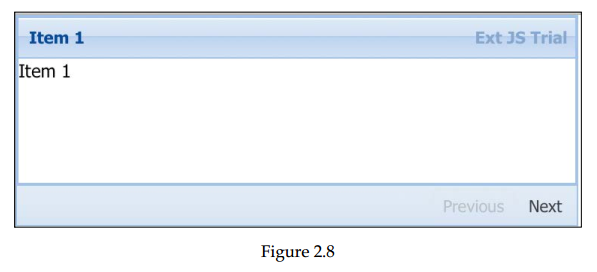


card 卡片布局

在此布局中，只有一个子组件是可见的，这个组件基本上充满整个容器。卡片布局一般应用在向导或者 tabs：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout: 'card',
6. defaultListenerScope: **true**,
7. bbar: ['->',{
8. itemId: 'btn-prev',
9. text: 'Previous',
10. handler: 'showPrevious',
11. disabled: **true**
12. },{
13. itemId: 'btn-next',
14. text: 'Next',
15. handler: 'showNext'
16. }],
17. items: [{
18. index: 0,
19. title: 'Item 1',
20. html: 'Item 1'
21. },{
22. index: 1,
23. title: 'Item 2',
24. html: 'Item 2'
25. },{
26. index:2,
27. title: 'Item 3',
28. html: 'Item 3'
29. }],
30. showNext: **function** () {
31. **this**.navigate(1);
32. },
33. showPrevious: **function** () {
34. **this**.navigate(-1);
35. },
36. navigate: **function** (incr) {
37. **var** layout = **this**.getLayout();
38. **var** index = layout.activeItem.index + incr;
39. layout.setActiveItem(index);
40. **this**.down('#btn-prev').setDisabled(index===0);
41. **this**.down('#btn-next').setDisabled(index===2);
42. }
43. });

卡片布局的输出。当你点击 next 按钮，将会显示 Item 2 面板：



center 中心布局

这种布局，容器的子组件在中间。在本章中开始介绍布局的部分，我们已经有一个例子了。

column 列布局

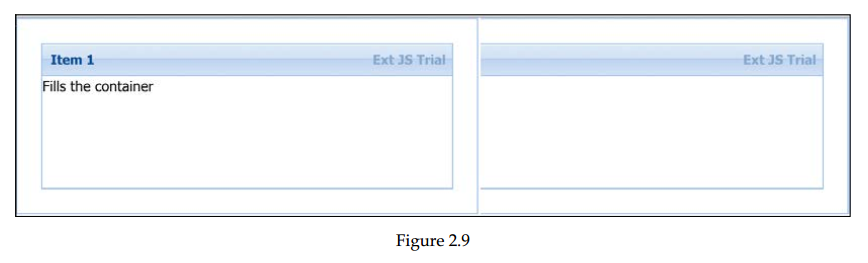
用此布局，你可以将容器划分为指定数量的列并指定每列所占的大小。这个例子也在本章开始介绍布局的部分中可以找到。

fit 适配布局

在此布局中，子组件将会自适应容器的尺寸。如下：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout: 'fit',
6. bodyPadding: 20,
7. items: [{
8. title: 'Item 1',
9. html: 'Fills the container',
10. }]
11. });

下列截图展示以上代码的输出。注意：Item 1 组件与父级容器之间的空隙是我们指定了 bodyPadding 属性：



hbox 布局

这种布局与 column 布局几乎是一样的，但是这种布局允许你拉伸列的高度。这里使用 flex 选项为子组件设置水平的相对值：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53893908) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53893908)

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout:{
6. type: 'hbox',
7. pack: 'start',
8. align: 'stretch',
9. },
10. items: [{
11. title: 'Item 1',
12. html: 'Item 1',
13. flex: 1
14. },{
15. title: 'Item 2',
16. html: 'Item 2',
17. width: 100
18. },{
19. title: 'Item 3',
20. html: 'Item 3',
21. flex: 2
22. }]
23. });

上面代码输出：

table 表格布局

这个布局允许你渲染一个表格出来。你可以指定列数和行数，使用 rowspan 和 colspan 创建复杂布局：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53893908) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53893908)

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout:{
6. type: 'table',
7. columns: 3,
8. tableAttrs: {
9. style: {
10. width: '100%'
11. }
12. }
13. },
14. items: [{
15. rowspan: 3,
16. title: 'Item 1',
17. html: 'Item 1'
18. },{
19. title: 'Item 2',
20. html: 'Item 2'
21. },{
22. title: 'Item 3',
23. rowspan: 2,
24. html: 'Item 3'
25. },{
26. title: 'Item 4',
27. html: 'Item 4'
28. },{
29. title: 'Item 5',
30. html: 'Item 5'
31. },{
32. title: 'Item 6',
33. html: 'Item 6'
34. },{
35. title: 'Item 7',
36. html: 'Item 7'
37. }]
38. });

以下截图所示为前面 table 布局代码的输出结果：

VBox 布局

这个布局内，子组件是垂直向下一个接一个排列。看一下以下的示例代码：

1. Ext.create('Ext.panel.Panel', {
2. renderTo: Ext.getBody(),
3. width: 700,
4. height: 400,
5. layout:{
6. type: 'vbox',
7. pack: 'start',
8. align: 'stretch',
9. },
10. items: [{
11. title: 'Item 1',
12. html: 'Item 1',
13. flex: 1
14. },{
15. title: 'Item 2',
16. html: 'Item 2',
17. height: 100
18. },{
19. title: 'Item 3',
20. html: 'Item 3',
21. flex: 2
22. }]
23. });

这段代码所示的输出：

总结

在本章中，我们了解学习了在 Ext JS 中的基础类和这些类中常用的方法，你还学习了如何创建和扩展一个类。还有如何使用 事件 和 查询元素及组件的功能。

在下一章，我们将会看一看许多有用的组件，例如 buttons ，menus ，toolbars 等等。我们也将根据所学创建一个小型的计算器应用。

# 第3章 - 基础组件

在本章中，你将学习到一些 Ext JS 基础组件的使用。同时我们会结合所学创建一个小项目。这一章我们将学习以下知识点：

* 熟悉基本的组件 – 按钮，文本框，日期选择器等等
* 表单字段的校验
* 菜单和工具栏
* 设计一个表单
* 计算器程序– 本章的示例项目

本章的主要目的是创建一个表单设计和一个计算器示例项目。以下图分别展示了表单设计和计算器设计。

首先，你观察下列表单设计，你会发现我们使用了大量的控件，例如 label 和文本框。

以下图展示了表单的设计：

继续，设计计算器程序大量的使用了按钮控件。所以你首要学习的是按钮和 handler 。随后在本章最后我们将会构建一个 计算器程序。在这个过程中，你会知道如何使 view(视图) 和 controller(控制器)进行交互并协同工作。我们还将看到如何绑定 view model(视图模型) 的属性到一个 view(视图) 的字段上。

下图为计算器的设计展示：

熟悉基本组件

Ext JS 有大量的优秀的控件，现在让我们开始认识这些基础的组件吧。

Ext.Button

这是一个很常用的控件；handler 是用于处理单击事件，如以下代码所示：

1. Ext.create('Ext.Button', {
2. text: 'My Button',
3. renderTo: Ext.getBody(),
4. handler: **function**() {
5. alert('click');
6. }
7. });

前面代码的输出：

我在第二章已经介绍过如何运行样例代码，但这里我还想再次重申这一点，此文档中的大部分样例代码都是可以直接运行的。你可以选择在你本地设备上或者在 Sencha Fiddle 上执行这些示例代码。你可以访问Sencha Fiddle 并将上面的代码键入到 launch 函数中，运行并查看结果。如果你访问了 [https://fiddle.sencha.com](https://fiddle.sencha.com/) 将会看到下列代码：

1. Ext.application({
2. name : 'Fiddle',
3. launch : **function**() {
4. Ext.Msg.alert('Fiddle', 'Welcome to Sencha Fiddle!');
5. }
6. }) ;

现在粘贴下列的创建按钮的样例代码，运行并查看结果：

1. Ext.application({
2. name : 'Fiddle',
3. launch : **function**() {
4. Ext.create('Ext.Button', {
5. text: 'My Button',
6. renderTo: Ext.getBody(),
7. handler: **function**() {
8. alert('click');
9. }
10. });
11. }
12. });

不是所有的代码都可以这样运行，此外并非所有的示例代码都会有视觉呈现。

你还可以使用 listeners 配置添加更多的事件处理器，如以下代码所示：

1. Ext.create('Ext.Button', {
2. text: 'My Button',
3. renderTo: Ext.getBody(),
4. listeners: {
5. click: {
6. fn: **function**(){
7. //Handle click event
8. alert('click');
9. }
10. },
11. mouseout: {
12. fn: **function**(){
13. //Handle double click event
14. alert('Mouse out');
15. }
16. }
17. }
18. });

以上代码只是创建了一个简单的按钮，你还可以创建很多不同的按钮，有 link button(连接按钮)，menu button(菜单按钮)，toggle button(开关按钮) 等等；

来创建一个链接按钮，设置 href 属性，如以下代码所示：

1. Ext.create('Ext.Button', {
2. renderTo: Ext.getBody(),
3. text: 'Link Button',
4. href: 'http://www.sencha.com/'
5. });

上面创建的链接按钮输出如图。当点击它则打开链接：

通过设置 menu 属性，创建一个菜单按钮，如以下代码所示：

1. Ext.create('Ext.Button', {
2. text: 'My Button',
3. renderTo: Ext.getBody(),
4. menu: [{
5. text: 'Item 1'
6. }, {
7. text: 'Item 2'
8. }, {
9. text: 'Item 3'
10. }]
11. });

输出如下，当点击时出现下拉菜单：

Ext.Button 还有许多属性，例如 bind， cls， disabled，html，tooltip，tpl 等等，你可以根据自己需求使用。

Ext.MessageBox

Ext.window.MessageBox 类提供了 message box 实现。Ext. MessageBox 是一个单例对象。你可以使用 MessageBox 弹出一个警告，信息确认，提示输入等等。

下列代码将弹出一个简单的提示信息。这里解释一下 Ext.Msg 是 Ext. Messagebox 类的别名：

1. Ext.Msg.alert('Info', 'Document saved!');

下列代码将弹出一个消息确认框，button 为选择的值，取 yes 或 no ：

1. Ext.Msg.confirm('Confirm', 'Are you want to cancel the updates?', **function**(button){
2. **if**('yes'==button) {
4. }   **else** {
6. }
7. });

你也可以自定义这个 message box 如下：

1. Ext.MessageBox.show({
2. title:'Save Changes?',
3. msg: 'Do you want to save the file?',
4. buttons: Ext.MessageBox.YESNO,
5. fn: **function**(button){
6. **if**('yes'==button){
8. }**else** **if**('no'==button){
10. }
11. },
12. icon: Ext.MessageBox.QUESTION
13. }) ;

上面代码输出如下：

表单和表单字段

现在我们看一下都有哪些表单相关的组件。

Ext.form.Panel

这个 form panel (表单面板)继承自 panel  并添加了表单相关的功能，例如字段管理，校验，提交等等。form panel 的默认布局是 anchor layout ，但是如果需要你可以改变这个配置。

form panel 有一个很方便的配置为 fieldDefaults，它可以用于指定表单内所有字段的默认类型。

fields (字段/表单域)

Ext JS 提供了很多内置的表单字段。比较常用的一些字段：

1. Ext.form.field.Checkbox
3. Ext.form.field.ComboBox
5. Ext.form.field.Date
7. Ext.form.field.File
9. Ext.form.field.Hidden
11. Ext.form.field.HtmlEditor
13. Ext.form.field.Number
15. Ext.form.field.Radio
17. Ext.form.field.Text
19. Ext.form.field.TextArea
21. Ext.form.field.Time

我们看一下其中的一些字段的应用。

Ext.form.field.Text

这是一个基本的文本框，它具有很多有用的属性和配置。其中有一个很有用的属性是 vtype 它是用于校验的。 例如以下代码，这个 vtype 属性为 email 用于验证输入内容是否是有效的电子邮箱：

1. Ext.create('Ext.form.field.Text', {
2. renderTo: Ext.getBody(),
3. name: 'email',
4. fieldLabel: 'Email',
5. allowBlank: **false**,
6. vtype: 'email'
7. });

这里 allowBlank 也是一个校验属性。通过设置 allowBlank 属性为 false ，如果这个字段为空白，将会提示校验不通过。

Ext.form.field.Number

number 字段继承自 spinner 字段，spinner 字段则继承自 text 字段，进而的 number 等于是继承了两者。这个 number 字段提供了几个选项来处理数值。下列代码创建了一个数值文本框：

1. Ext.create('Ext.form.field.Number', {
2. renderTo: Ext.getBody(),
3. name: 'Count',
4. fieldLabel: 'Count',
5. value: 0,
6. maxValue: 10,
7. minValue: 0
8. });

你可以移除下拉按钮，方向键，鼠标滚轮监听，用配置：hideTrigger， keyNavEnabled，和 mouseWheelEnabled 。

Ext.form.field.ComboBox

下列代码创建了一个月份下拉菜单。这个 combobox 有一个配置为 store。 这个 store 是数据源，为此下拉菜单提供数据。store 是属于 ExtJS 中数据包部分， 在接下来的章节中我们会详细介绍的。

combobox 中另一个重要的配置是 queryMode 。这个属性取值可以是 ‘local’ 或者 ‘remote’。如果你设置为 remote 了，那么这个数据源 store 将在运行加载数据时发送请求从远程服务器获取数据：

1. **var** months = Ext.create('Ext.data.Store', {
2. fields: ['abbr', 'name'],
3. data: [{"abbr":"JAN", "name":"January"},
4. {"abbr":"FEB", "name":"February"},
5. {"abbr":"MAR", "name":"March"},
6. {"abbr":"APR", "name":"April"},
7. {"abbr":"MAY", "name":"May"},
8. {"abbr":"JUN", "name":"June"},
9. {"abbr":"JUL", "name":"July"},
10. {"abbr":"AUG", "name":"August"},
11. {"abbr":"SEP", "name":"September"},
12. {"abbr":"OCT", "name":"October"},
13. {"abbr":"NOV", "name":"November"},
14. {"abbr":"DEC", "name":"December"}]
15. }) ;
17. Ext.create('Ext.form.ComboBox', {
18. fieldLabel: 'Choose Month',
19. store: months,
20. queryMode: 'local',
21. displayField: 'name',
22. valueField: 'abbr',
23. renderTo: Ext.getBody()
24. });

以上代码的输出如下：

Ext.form.field.HtmlEditor

Ext JS 也有一个非常优秀的 HTML 编辑器，它提供直接在 web 页面上处理文字的能力，如以下代码所示：

1. Ext.create('Ext.form.HtmlEditor', {
2. width: 800,
3. height: 200,
4. renderTo: Ext.getBody()
5. });

以上代码输出如下：

 表单字段的校验

大多数表单都有自己的校验规则，例如你键入了一个非数值的内容到 number 字段，它将显示一个验证无效的提示。再有这个 text 字段(文本框) 校验属性有  allowBlank，minLength，和 maxLength 。 更进一步的，还有 regex 属性可以使用正则表达式自定义校验。

form panel 的事件

form panel 支持的部分事件：

* beforeaction: 任意动作执行前触发，例如 submit，load，doAction 这些动作执行时
* actionfailed: 执行一个动作失败时触发
* actioncomplete: 在一个动作执行完成之后触发
* validitychange: 表单键入的内容有效性发生变化时触发
* dirtychange: 表单的dirty状态改变时触发

表单字段容器

以下是一些 from panel 里很有用的容器。

Ext.form.CheckboxGroup

CheckboxGroup 继承自 FieldContainer 用于组织复选框。下列示例中，复选框组的 items 中所有的项都有相同的 name ；这有助于将得到的值作为一个单一的参数传递给服务器。

1. Ext.create('Ext.form.CheckboxGroup', {
2. renderTo: Ext.getBody(),
3. fieldLabel: 'Skills ',
4. vertical: **true**,
5. columns: 1,
6. items: [{ boxLabel: 'C++', name: 'rb', inputValue: '1' },
7. { boxLabel: '.Net Framework', name: 'rb', inputValue: '2', checked: **true** },
8. { boxLabel: 'C#', name: 'rb', inputValue: '3' },
9. { boxLabel: 'SQL Server', name: 'rb', inputValue: '4' }]
10. }) ;

以上代码输出如下：

Ext.form.FieldContainer

FieldContainer 是很有用的，当你想将一组相关字段附加到一个标签时。

以下代码的输出你会发现一个 label 后面绑定了两个文本框：

1. Ext.create('Ext.form.FieldContainer', {
2. renderTo: Ext.getBody(),
3. fieldLabel: 'Name',
4. layout: 'hbox',
5. combineErrors: **true**,
6. defaultType: 'textfield',
7. defaults: {
8. hideLabel: 'true'
9. },
10. items: [{
11. name: 'firstName',
12. fieldLabel: 'First Name',
13. flex: 2,
14. emptyText: 'First',
15. allowBlank: **false**
16. }, {
17. name: 'lastName',
18. fieldLabel: 'Last Name',
19. flex: 3,
20. margin: '0 0 0 6',
21. emptyText: 'Last',
22. allowBlank: **false**
23. }]
24. });

Ext.form.RadioGroup

RadioGroup 继承自 CheckboxGroup 用于组织单选按钮。items 中的项都有相同的 name，另外这是单选的，如以下代码所示：

1. Ext.create('Ext.form.RadioGroup', {
2. renderTo: Ext.getBody(),
3. fieldLabel: 'Sex ',
4. vertical: **true**,
5. columns: 1,
6. items: [{
7. boxLabel: 'Male',
8. name: 'rb',
9. inputValue: '1'
10. },{
11. boxLabel: 'Female',
12. name: 'rb',
13. inputValue: '2'
14. }]
15. });

代码输出：

提交表单

使用 form 的 submit 方法提交表单。使用 getForm 方法获取表单并 isValid 方法进行提交前的表单内容校验。如以下代码所示：

1. **var** form = **this**.up('form').getForm();
2. **if** (form.isValid()) {
3. form.submit({
4. url: 'someurl',
5. success: **function** () {
7. },
8. failure: **function** () {
10. }
11. });
12. } **else** {
13. Ext.Msg.alert('Error', 'Fix the errors in the form')
14. }

菜单和工具栏

对于你能想到的任何的菜单和工具栏 Ext JS 提供了最完整的支持。Ext.toolbar.Toolbar 用于构建一个工具栏。默认情况下任何子项在 Ext.toolbar.Toolbar 都是按钮，但是你可以添加任意控件进去，例如一个文本框，一个数值框，一个图标，一个下拉菜单等等。

规范整理你的工具栏中的项，你可以使用 空格(Ext.toolbar.Spacer)， 分隔符(Ext.toolbar. Separator)，和 使控件右对齐(Ext.toolbar.Fill) 。这里也可以使用快捷方式  ‘ ‘ (空格)，’-‘ 和 ‘|’ (都是分隔符，只有很小的差别)，和 ‘->‘ (右对齐)。

Ext.menu.Menu 用于构建一个菜单，items 属性中为 Ext.menu.Item 一个个菜单项。

一个简单的代码示例和以下截图的输出：

1. Ext.create('Ext.toolbar.Toolbar', {
2. renderTo: Ext.getBody(),
3. width: 800,   items: [{
4. text: 'My Button'
5. },{
6. text: 'My Button',
7. menu: [{
8. text: 'Item 1'
9. }, {
10. text: 'Item 2'
11. }, {
12. text: 'Item 3'
13. }]
14. },{
15. text: 'Menu with divider',
16. tooltip: {
17. text: 'Tooltip info',
18. title: 'Tip Title'
19. },
20. menu: {
21. items: [{
22. text: 'Task 1',
23. // handler: onItemClick
24. }, '-', {
25. text: 'Task 2',
26. // handler: onItemClick
27. }, {
28. text: 'Task 3',
29. // handler: onItemClick
30. }]
31. }
32. },'->',{
33. xtype: 'textfield',
34. name: 'field1',
35. emptyText: 'search web site'
36. },'-','Some Info',{
37. xtype: 'tbspacer'
38. },{
39. name: 'Count',
40. xtype: 'numberfield',
41. value: 0,
42. maxValue: 10,
43. minValue: 0,
44. width: 60
45. }]
46. });

设计一个(客户反馈)表单

现在根据之前所学，我们来设计一个表单。

我们将设计如图所示的表单：

以下是这个表单的代码。这里我维护着一个这个例子的完整的源码 <https://github.com/ananddayalan/extjs-by-example-customer-feedback-form>

这里我们所有的组件都在 Viewport 中。 这是一个专用的容器，它代表浏览器里应用的视图区域。

在 Viewport 中我们设置 scrollable 选项将子组件设为滚动的，使用 true 或 false 。也可以取值为 x 或 y 表示只允许水平或垂直滚动：

1. Ext.create('Ext.container.Viewport', {
2. scrollable: **true**,
3. items: [{
4. xtype: 'container',
5. layout: {
6. type: 'hbox',
7. align: 'center',
8. pack: 'center'
9. },
10. items: [ {
11. xtype: 'form',
12. bodyPadding: 20,
13. maxWidth: 700,
14. flex: 1,
15. title: 'Custom Feedback',
16. items:[{
17. xtype: 'fieldcontainer',
18. layout: 'hbox',
19. fieldLabel: 'Name',
20. defaultType: 'textfield',
21. defaults: {
22. allowBlank: **false**,
23. flex: 1
24. },
25. items: [{
26. name: 'firstName',
27. emptyText: 'First Name
28. }, {
29. name: 'lastName',
30. margin: '0 0 0 5',
31. emptyText: 'Last Name'
32. }]
33. },{
34. xtype: 'datefield',
35. fieldLabel: 'Date of Birth',
36. name: 'dob',
37. maxValue: **new** Date() /\* Prevent entering the future date.\*/
38. }, {
39. fieldLabel: 'Email Address',
40. name: 'email',
41. vtype: 'email',
42. allowBlank: **false**
43. }, {
44. fieldLabel: 'Phone Number',
45. labelWidth: 100,
46. name: 'phone',
47. width: 200,
48. emptyText: 'xxx-xxx-xxxx',
49. maskRe: /[\d\-]/,
50. regex: /^\d{3}-\d{3}-\d{4}$/,
51. regexText: 'The format must be xxx-xxx-xxxx'
52. },{
53. xtype: 'radiogroup',
54. fieldLabel: 'How satisfied with our service?',
55. vertical: **true**,
56. columns: 1,
57. items: [ {
58. boxLabel: 'Very satisfied',
59. name: 'rb',
60. inputValue: '1'
61. }, {
62. boxLabel: 'Satisfied',
63. name: 'rb', inputValue: '2'
64. }]
65. },{
66. xtype: 'checkboxgroup',
67. fieldLabel: 'Which of these words would you use to describe our products? Select all that apply',
68. vertical: **true**,
69. columns: 1,
70. items: [{
71. boxLabel: 'Reliable',
72. name: 'ch',
73. inputValue: '1'
74. }]
75. },{
76. xtype: 'radiogroup',
77. fieldLabel: 'How likely is it that you would recommend this company to a friend or colleague?',
78. vertical: **false**,
79. defaults: { padding: 20 },
80. items: [ {
81. boxLabel: '1',
82. name: 'recommend',
83. inputValue: '1'
84. }],
85. buttons: [{
86. text: 'Submit',
87. handler: **function** () {
88. **var** form = **this**.up('form').getForm();
89. **if** (form.isValid()) {
90. form.submit({
91. url: 'cutomer/feedback',
92. success: **function** () {},
93. failure: **function** () {}
94. });
95. } **else** {
96. Ext.Msg.alert('Error', 'Fix the errors in the form')
97. }
98. }
99. }]
100. }]
101. }]
102. }]
103. });

在以上代码中通过在容器级设置 defaultType 属性，这样我们就可以不必在容器的每个子组件里重复的指定 xtype 属性了。这样默认情况下，所有子组件在没有显式指定 xtype 时默认的类型都是 textfield 。

form panel 上有一个 flex 配置用于填补父容器的宽度，同时通过设置 maxWidth 为 700 限制 form panel 的最大宽度。

字段容器使用 hbox 布局将 first name 和 last name 文本框放在一个 label 标签下。

 写一个计算器应用

现在我们结合目前所学构建一个完整的小项目。这是我们将要构建的计算器的设计：

 文件夹结构

这是我们创建的计算器工程的目录结构。这里我不是用 sencha Cmd 生成的项目，只是从 Ext JS 复制了一些必须的文件到项目文件夹中：

完整可用的项目在这里： <https://github.com/ananddayalan/extjs-by-example-calculator>.

App – app.js

在 app.js 文件里我们简单的创建了 Main 视图，作为可移动窗体浮动在浏览器：

1. Ext.application({
2. name: 'Calc',
3. launch: **function** () {
4. Ext.create('Calc.view.main.Main').show();
5. }
6. });

再谈 MVC 和 MVVM

第一章的时候，我们已经介绍过 MVC (Model View Controller) 和 MVVM (Model View ViewModel)。 这个示例项目的代码很好的展示了 视图，控制器，和视图模型之间的区别。

Model (模型)

这代表着数据层。model 保存的数据可以包含数据验证和逻辑。

View (视图)

这一层是用户界面。包含有 button，form，和 message box 等等组件。在我们这次写的计算器应用中 main.js 就是一个很好的视图例子。

Controller (控制器)

控制器处理 view(视图)相关的逻辑，例如 view 的 event(事件)处理，还有任何程序相关逻辑都可以写在这里。

ViewController (视图控制器) 和 Controller (控制器)

在 Ext JS 5 和 6 中，有两种类型的控制器：ViewController 和 Controller。 这个 ViewController 自 Ext JS 5 开始引进的。ViewController 是为一个指定的视图创建的控制器，但是这个控制器也可以交叉其他视图的逻辑。

ViewController 带来了一些新的概念，例如 引用和监听，简化视图与控制之间的关系。同时 View 销毁时 ViewController 也会被销毁，他们具有相同的生命周期，在这个例子中我们没有使用 引用和监听，但是在下一个例子中我们会使用的。

* 你可以使用 listeners  代替 handler 处理事件

 View model

view model 封装了 view(视图)所需要的展示逻辑，绑定数据到 view 并且每当数据改变时处理更新。

它有别于 model ，view model 主要是为一个指定的视图而创建的。一个 model 是一个纯粹的数据类并可用于整个应用中，但一个 view model 是起到一个 view 和 model 之间的数据粘合剂的作用。看一下main.js 的 视图模型绑定。

视图 — Main.js

这里我为这个计算器应用创建一个视图为 Main 。这个视图里包含所有的按钮，显示字段等等。相关的事件用 controller 的方法。这个视图的控制器已经使用 controller 配置指定了。

这个视图使用 table 布局，配置为 4 列。CSS 类使用 cls 属性指定。

代码里有附加的注释：

1. Ext.define('Calc.view.main.Main', {
2. extend: 'Ext.window.Window',
3. /\* 表示在当前视图加载之前先加载这些所需的类\*/
4. requires: [ 'Calc.view.main.MainController',
5. 'Calc.view.main.MainModel'],
6. xtype: 'app-main',
7. controller: 'main',
8. /\* 视图的 view model (视图模型)\*/
9. viewModel: { type: 'main' },
10. resizable: **false**,
11. layout: {
12. type: 'table',
13. columns: 4
14. },
15. /\* defaultType 和 defaults 属性是用于 items 内的子组件的，任何子组件都可以覆盖这些配置 \*/
16. defaultType: 'button',
17. defaults: {
18. width: 50,
19. height: 50,
20. cls: 'btn',
21. handler: 'onClickNumber'
22. },
23. /\* 这里我用 Ext.window.Window 的 header 显示计算器的结果。使用 header 你可以在浏览器里移动这个计算器。\*/
24. header: {
25. items: [{
26. xtype: 'displayfield',
27. colspan: 4,
28. width: 200,
29. cls: 'display',
30. bind: {
31. value: '{display}'
32. },
33. height: 60,
34. padding: 0
35. }]
36. },
37. items: [{
38. text: 'C',
39. colspan: 2,
40. width: 100,
41. cls: 'btn-green',
42. handler: 'onClickClear'
43. }, {
44. text: '+/-',
45. cls: 'btn-green',
46. handler: 'onClickChangeSign'
47. }, {
48. text: '÷',
49. cls: 'btn-orange',
50. handler: 'onClickOp'
51. },{
52. text: '7'
53. },{
54. text: '8'
55. },{
56. text: '9'
57. },{
58. text: '×',
59. cls: 'btn-orange',
60. handler: 'onClickOp'
61. },{
62. text: '4'
63. },{
64. text: '5'
65. },{
66. text: '6'
67. },{
68. text: '-',
69. cls: 'btn-orange',
70. handler: 'onClickOp'
71. },{
72. text: '1'
73. },{
74. text: '2'
75. },{
76. text: '3'
77. },{
78. text: '+',
79. cls: 'btn-orange',
80. handler: 'onClickOp'
81. },{
82. text: '0',
83. width: 100,
84. colspan: 2
85. },{
86. text: '.',
87. handler: 'onClickDot'
88. },{
89. text: '=',
90. cls: 'btn-orange',
91. handler: 'onClickOp'
92. }]
93. });

控制器 — MainController.js

虽然这个控制器的代码有点长，这是一个非常简单的代码。控制器中有很多方法处理按钮的点击事件，例如运算符和操作数的点击处理。控制器使用了一个 model 为 Main ：

1. Ext.define('Calc.view.main.MainController', {
2. extend: 'Ext.app.ViewController',
3. alias: 'controller.main',
4. views: ['Calc.view.main.Main'],
5. models: ['Main'],
6. //这个 state 是一个自定义属性，用来跟踪计算器的状态。
7. state: {
8. operatorClicked: **false**,
9. selectedOperator: **null**,
10. dotClicked: **false**,
11. op1: 0,
12. numberClicked: **false**,
13. sign: **true**,
14. decimal: **false**
15. },
16. onClickClear: **function** () {
17. **var** vm = **this**.getViewModel();
18. vm.set('display','0');
19. **this**.state.selectedOperator=**null**;
20. **this**.state.op1=0;
21. **this**.state.isPositive = **true**;
22. **this**.state.decimal = **false**;
23. **this**.state.sign = **true**;
24. },
25. onClickChangeSign: **function** (btn) {
26. **var** vm = **this**.getViewModel();
27. **var** cur = vm.get('display');
28. **if**(cur!='0') {
29. **if**(**this**.state.sign===**true** ) {
30. vm.set('display', '-' + cur);
31. }**else** {
32. vm.set('display', cur.toString().substring(1));
33. }
34. }
35. **this**.state.sign=!**this**.state.sign;
36. },
37. onClickOp: **function** (btn) {
38. **if**(**this**.state.selectedOperator!=**null** && **this**.state.numberClicked===**true**){
39. **var** vm = **this**.getViewModel();
40. **var** op2 = parseFloat(vm.get('display'));
41. **var** op1 = parseFloat(**this**.state.op1);
42. **var** result = 0;
43. **switch**(**this**.state.selectedOperator){
44. **case** '+':
45. result = op1 + op2;
46. **break**;
47. **case** '-':
48. result = op1 - op2;
49. **break**;
50. **case** '×':
51. result = op1 \* op2;
52. **break**;
53. **case** '÷':
54. result = op1 / op2;
55. **break**;
56. }
57. vm.set('display', Math.round(result \* 100) / 100);
58. **this**.state.selectedOperator=**null**;
59. }
60. **if**(btn.text!='=') {
61. **this**.state.operatorClicked = **true**;
62. }
63. **this**.state.selectedOperator = btn.text;
64. **this**.state.numberClicked = **false**;
65. },
66. onClickDot: **function** (btn) {
67. **if**(**this**.state.decimal===**false**) {
68. **var** vm = **this**.getViewModel();
69. vm.set('display', vm.get('display') + '.');
70. }
71. },
72. onClickNumber: **function** (btn) {
73. **this**.state.numberClicked = **true**;
74. **if**(**this**.state.selectedOperator ==='='){
75. **this**.onClickClear();
76. }
77. **var** vm = **this**.getViewModel();
78. **if**(**this**.state.operatorClicked===**true**) {
79. **this**.state.op1= vm.get('display');
80. vm.set('display', btn.text);
81. **this**.state.operatorClicked=**false**;
82. }**else**{
83. **var** cur = vm.get('display');
84. **if**(cur == '0') {
85. cur = '';
86. }
87. vm.set('display', cur + btn.text);
88. }
89. }
90. });

视图模型 — MainViewModel.js

这个 ViewModel 只有一个属性为 display 。这个用来绑定到计算器显示的值上。这里我们不会分别用一组字段创建模型，此外我们还将会硬编码数据。

1. Ext.define('Calc.view.main.MainModel', {
2. extend: 'Ext.app.ViewModel',
3. alias: 'viewmodel.main',
4. data: {
5. display: 0.0
6. }
7. });

在即将到来的章节中你将学习更多关于 模型，视图模型，字段，字段类型，校验 等等。

总结

在本章中，你了解了不同的基本组件，例如 文本框，数字框，按钮，菜单等等。你已经学会如何使用表单字段设计一个表单和我们之前创建了一个简单的计算器项目。

在下一章中，你将学习关于数据包的内容，例如 数据源 ，模型 ，代理等等。store ，model ，这将是有益于处理数据的。

# 第4章 - 数据包装

本章探索 Ext JS 中处理数据可用的工具以及服务器和客户端之间的通信。在本章结束时将写一个调用 RESTful 服务的例子。下面是本章的内容：

* 模型
* Schema
* Stores
* 代理
* 过滤和排序
* 做一个基于 RESTful 的小项目

Model（模型）

一个模型包含字段，字段类型，校验，关联和代理。它是通过扩展 Ext.data.Model 类来定义的。

其中类型，校验，关联和代理都是可选的。 当一个字段没有指定类型时，将使用默认类型 ‘auto’。 通常 代理都是设置在 store 中，但是 model 里也可以设置代理。

Field（字段）

Ext.data.field.Field 用于添加模型的属性。 这个属性字段的类型可以是预定义的或自定义类型。下列是可用的预定义类型：

* auto
* boolean
* date
* int
* number
* string

数据转换

默认当你为一个字段指定类型后，这个数据将在保存到该字段之前进行转换。转换的过程由内置的 convert 方法处理。auto 字段类型不具备 convert 方法，所以 auto 类型的字段不会发生转换。

其他所有的字段类型都有 convert 方法。如果你想避免其他字段转换来提高性能，你可以通过指定各自的转换配置为 null 。如下列 Employee 模型所示。

验证器/校验器

数据模型支持数据的校验。下列是支持的验证器：

* presence: 确保值是存在于一个字段
* format: 能够用一个正则表达式来进行验证
* length: 支持最大和最小长度的验证
* exclusion 和 inclusion: 你可以通过给出一组值来确保字段的值存在或不存在于这组值中。

下列代码示例展示一个 Employee 模型，在模型的字段上使用验证器：

1. Ext.define('Employee', {
2. extend: 'Ext.data.Model',
3. fields: [
4. { name: 'id', type: 'int', convert: **null** },
5. { name: 'firstName', type: 'string' },
6. { name: 'lastName', type: 'string' },
7. { name: 'fulltime', type: 'boolean', defaultValue: **true**, convert: **null** },
8. { name: 'gender', type: 'string' },
9. { name: 'phoneNumber', type: 'string'},],
10. validators: {
11. firstName: [{
12. type: 'presence'
13. },{
14. type: 'length',
15. min: 2
16. }],
17. lastName:[{
18. type: 'presence'
19. },{
20. type: 'length',
21. min: 2
22. }],
23. phoneNumber: {
24. type: 'format',
25. matcher: '/^[(+{1})|(00{1})]+([0-9]){7,10}$/'
26. },gender: {
27. type: 'inclusion',
28. list: ['Male', 'Female']
29. }
30. }
31. });

创建一个模型的实例，使用 Ext.create ，如以下代码所示：

1. **var** newEmployee = Ext.create('Employee', {
2. id : 1,
3. firstName : 'Shiva',
4. lastName : 'Kumar',
5. fulltime : **true**,
6. gender: 'Male',
7. phoneNumber: '123-456-7890'
8. });

 创建好的模型对象有 get 和 set 方法用来读取和设置字段值：

1. **var** lastName = newEmployee.get('lastName'); newEmployee.set('gender','Female');

 关系

定义实体之间的关系，使用下列关系类型之一：

One-to-one（一对一）

下列代码代表一对一关系：

1. Ext.define('Address', {
2. extend: 'Ext.data.Model',
3. fields: [
4. 'address',
5. 'city',
6. 'state',
7. 'zipcode']
8. });
9. Ext.define('Employee', {
10. extend: 'Ext.data.Model',
11. fields: [{
12. name: 'addressId',
13. reference: 'Address'
14. }]
15. });

One-to-many（一对多）

下列代码代表一对多关系：

1. Ext.define('Department', {
2. extend: 'Ext.data.Model',
3. fields: [{
4. name: 'employeeId',
5. reference: 'Employee'
6. }]
7. });
8. Ext.define('Division', {
9. extend: 'Ext.data.Model',
10. fields: [{
11. name: 'departmentId',
12. reference: 'Department'
13. }]
14. });

Many-to-many（多对多）

下列代码代表多对多关系：

1. Ext.define('Employee', {
2. extend: 'Ext.data.Model',
3. fields: [{
4. name: 'empId',
5. type: 'int',
6. convert: **null**
7. },{
8. name: 'firstName',
9. type: 'string'
10. },{
11. name: 'lastName',
12. type: 'string'
13. }],
14. manyToMany: 'Project'
15. });
16. Ext.define('Project', {
17. extend: 'Ext.data.Model',
18. fields: [
19. 'name'
20. ],
21. manyToMany: 'Employee'
22. });

 自定义字段类型

你也可以通过简单的扩展 Ext.data.field.Field 类来创建自定义字段类型。如以下代码所示：

1. Ext.define('App.field.Email', {
2. extend: 'Ext.data.field.Field',
3. alias: 'data.field.email',
4. validators: {
5. type: 'format',
6. matcher: /^\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$/,
7. message: 'Wrong Email Format'
8. }
9. });

Store

一个 store 代表一个 模型的实例 的集合并用于代理获取数据。store 还定义了集合操作，例如 排序，过滤等等。它是通过扩展 Ext.data.Store 定义的。

通常，当定义一个 store ，你需要为 store 指定一个 代理。这是一个配置属性，它告诉 store 如何读取和写入数据。在本章的后面我们将会看到更多详细的不同种类的代理。

 下列代码是一个 store 的例子，它使用 RESTful API 请求加载为 JSON 格式数据：

1. **var** myStore = Ext.create('Ext.data.Store', {
2. model: 'Employee',
3. storeId: 'mystore',
4. proxy: {
5. type: 'rest',
6. url: '/employee',
7. reader: {
8. type: 'json',
9. rootProperty: 'data'
10. }
11. },
12. autoLoad: **true**,
13. autoSync: **true**
14. });

这里 storeId 是 store 的唯一标示符。这个 store 有一个方法 load ，用于通过代理配置加载数据。如果你设置 autoLoad 为 true ，那么当 store 创建后将会自动调用 load 方法。如果设置 autoLoad 为 false ，那么你可以手动的调用 load 方法加载数据。

同样的，你可以设置 autoSync 为 true ，当你对 store 的记录进行修改新增和删除时将会自动发生同步。

在前面的例子中，当在 store 中执行修改，新增或删除时，它将调用 REST service API 。如果你设置这个属性为 false ，那么你可以调用 sync 方法执行同步操作。

调用 sync 方法将触发一个批量操作。所以如果你添加和删除了多条记录，那么调用 sync方法这个过程中将触发多次服务器调用。这是一个 sync 调用的例子：

1. store.sync({
2. callback: **function** (batch, options) {
3. //Do something
4. },
5. success: **function** (batch, options) {
6. //Do something
7. },
8. failure: **function** (batch, options) {
9. //Do something
10. },
11. scope: **this**
12. });

这里，当所有 sync 操作全部完成并且没有任何例外和失败时调用 success 方法。如果有一个或多个操作在 sync 过程中失败，将调用 failure 方法。callback 方法则会在同步操作完成后，不论成功失败都会被调用。

如果 failure 被调用，你可以检查 batch 异常数组来看到为什么操作失败。这里 options 是 sync 方法中传递的原始参数。

这个 sync 方法调用时也可以添加一个属性 params ，它可以用于在同步的时候传递任意你想附加的参数。

内联数据 store

如果你不想绑定 store 到服务器或外部存储，例如浏览器本地存储，但还想使用一些特殊的的静态数据，那么可以直接硬编码内联数据到 store ，如以下代码所示：

1. Ext.create('Ext.data.Store', {
2. model: 'Employee',
3. data: [{
4. firstName: 'Shiva',
5. lastName: 'Kumar',
6. gender: 'Male',
7. fulltime: **true**,
8. phoneNumber: '123-456-7890'
9. },{
10. firstName: 'Vishwa',
11. lastName: 'Anand',
12. gender: 'Male',
13. fulltime: **true**,
14. phoneNumber: '123-456-7890'
15. }]
16. });

过滤和排序

store 支持本地或远程的过滤和排序。下面是本地排序的例子：

1. **var** myStore = Ext.create('Ext.data.Store', {
2. model: 'Employee',
3. sorters: [{
4. property: 'firstName',
5. direction: 'ASC'
6. }, {
7. property: 'fulltime',
8. direction: 'DESC'
9. }],
10. filters: [{
11. property: 'firstName',
12. value: /an/
13. }]
14. });

使用远程排序和远程过滤，需要设置 remoteSort 和 remoteFilter 属性为 true 。如果你设置为 true ，那么在服务器端你必须要执行过滤和排序并为客户端返回过滤或排序后的数据。

访问 store

你也许需要在应用的其他地方访问这个 store 有很多方法可以用。

使用 StoreManager 访问 store

使用 store 管理器的 lookup 方法，你能够在程序的任何地方来访问 store。为此我们需要使用 storeId ，注意，当 store 通过一个控制器实例化的时设置 storeId，storeId 将会被覆盖，这是一个例子：

1. Ext.create('Ext.data.Store', {
2. model: 'Employee',
3. storeId: 'mystore',
4. proxy: {
5. type: 'rest',
6. url: '/employee',
7. reader: {
8. type: 'json',
9. rootProperty: 'data'
11. }
12. }
13. });

假设我们已经创建了 store 如上所示。现在你可以通过传递 storeId 到 store 管理器的 StoreManager.lookup 方法访问这个 store，如以下代码所示：

1. Ext.data.StoreManager.lookup('myStore');

你也可以使用 Ext.getStore 方法。Ext.getStore 是 Ext.data.StoreManager.lookup 的快捷方法。

使用 Ext.app.ViewModel 访问 store

你可以通过 Ext.app.ViewModel 的 getStore 方法 访问 store 。当你是在 ViewController 中，最好使用这种方式，下面是代码示例：

1. **var** myStore = **this**.getViewModel().getStore('myStore')

这个 getStore 方法也定义在 view 中，你也可以用它访问 store。

Store 事件

store 提供了很多可监听的事件。store 的一些常用事件：

* add: 当一条记录添加到 store 后调用
* beforeload: 在加载数据之前调用
* beforesync: 在进行同步操作之前调用
* datachanged: 当 store中的记录产生新增或删除时触发调用
* load: 当 store 读取一个远程数据源后触发调用
* remove: 从 store 移除一条记录触发调用
* update: 当 store 中的一条记录被更新触发调用

给出的一个 store 事件监听的例子：

1. Ext.create('Ext.data.Store', {
2. model: 'Employee ',
3. storeId: 'mystore',
4. proxy: {
5. type: 'rest',
6. url: '/employee',
7. reader: {
8. type: 'json',
9. rootProperty: 'data'
10. }
11. },
12. listeners: {
13. load: **function** (store, records, options) {
14. //Do something
15. }
16. }
17. });

如果你想在你的控制器中监听 store 事件，你可以这样做：

1. init: **function**() {
2. **this**.getViewModel().getStore('myStore').on('load', **this**.onStoreLoad, **this**);
3. }

在 ViewModel 中定义 store

你可以分别定义 store 和 ViewModel 或者定义到一起。通常可取的定义 store 在 ViewModel 自身。一个例子如下：

1. Ext.define('MyApp.view.employee.EmployeeModel', {
2. extend: 'Ext.app.ViewModel',
3. alias: 'viewmodel.employee',
4. stores: {
5. employee: {
6. fields: [{
7. name: 'id',
8. type: 'string'
9. },{
10. name: 'firstname',
11. type: 'string'
12. },{
13. name: 'lastname',
14. type: 'string'
15. }],
16. autoLoad: **false**,
17. sorters: [{
18. property: 'firstname',
19. direction: 'ASC'
20. }],
21. proxy: {
22. type: 'rest',
23. url: 'employee',
24. reader: {
25. type: 'json',
26. },
27. writer: {
28. type: 'json'
29. }
30. }
31. }
32. }
33. });

代理

所有的 stores 和 models 使用 proxy 来加载和保存数据。在代理中使用这个配置你可以指定如何读取和写入数据。你也可以指定调用 URL 读取数据；你可以告诉读取器这些数据的格式，不论是 JSON 还是 XML ，等等。

有两种类型的代理：客户端代理和服务器代理。

客户端代理

客户端代理是用于处理客户端本身数据的加载和保存。这分为三种客户端代理：内存，本地存储，和会话存储。

内存代理

内存代理是用于内存中的局部变量数据。下列代码展示了一个内存代理的例子。在这里这个数据的值没有硬编码。只要是合适的格式，可以是任何变量的数据:

1. **var** data = {
2. data: [{
3. firstName: 'Shiva',
4. lastName: 'Kumar',
5. gender: 'Male',
6. fulltime: **true**,
7. phoneNumber: '123-456-7890'
8. },{
9. firstName: 'Vishwa',
10. lastName: 'Anand',
11. gender: 'Male',
12. fulltime: **true**,
13. phoneNumber: '123-456-7890'
14. }]
15. };
16. **var** myStore = Ext.create('Ext.data.Store', {
17. model: 'Employee',
18. data : data,
19. proxy: {
20. type: 'memory',
21. reader: {
22. type: 'json',
23. rootProperty: 'Employee'
25. }
26. }
27. });

本地存储代理

这个是用于访问浏览器本地存储。它是一个键值对存储添加在 html5 ，所以需要浏览器支持：

1. **var** myStore = Ext.create('Ext.data.Store', {
2. model: 'Benefits',
3. autoLoad: **true**,
4. proxy: {
5. type: 'localstorage',
6. id: 'benefits'
7. }
8. });

会话存储代理

这个是用于访问浏览器会话存储。这也是一个 html5 特性，因此需要比较现代的浏览器才能支持。这些数据是当 session 超时后会被销毁：

1. **var** myStore = Ext.create('Ext.data.Store', {
2. model: 'Benefits',
3. autoLoad: **true**,
4. proxy: {
5. type: 'localstorage',
6. id : 'benefits'
7. }
8. });

服务器端代理

服务器端代理是向服务器通信来读取或保存数据。有四种代理：

* Ajax: 用于异步请求数据
* Direct: 使用 Direct 与服务器通信
* JSONP (JSON with padding): 这很有用，当你需要发送跨域请求时。而 ajax 只能请求相同的域。
* REST: 这会向服务器发送一个 ajax 请求，使用 RESTful 的风格，例如 GET，POST，PUT，和 DELETE。

在本章我们已经看到过一个 REST 代理的例子。让我们瞧一瞧 JSONP 的例子：

1. **var** myStore = Ext.create('Ext.data.Store', {
2. model: 'Products',
3. proxy: {
4. type: 'jsonp',
5. url : 'http://domain.com/products',
6. callbackKey: 'productsCallback'
7. }
8. });

To do 待办（RESTful 的示例项目）

现在，让我们运用本章和前面章节所学内容，创建一个 To do 待办 应用。这里我们将使用 store 的 REST 代理来连接 REST 服务。

我们来创建一个简单的 RESTful 服务，我们将使用 Go 语言，也称为 Golang 。它由 Google 开发，是静态语言，语法松散。

你不必学习 Go 语言，这个项目主要集中在 Ext JS 的教学。你可以用任意的你熟悉的语言替换 Go 语言开发的 RESTful 服务代码。

这是我们将要创建的应用的设计：

让我们瞧一瞧这个项目的一些重要文件。完整的代码在这里 <https://github.com/ananddayalan/extjs-by-example-todo>。

下面截图展示了这个程序的目录结构：

 首先，让我们先来创建 store ，我们已经在本章学习过了。这里我将 store 写在 ViewModel 中。

下列 ViewModel中有三个字段：id ，desc（描述） ，done（显示待办是否完成），代理类型为 rest 并设置 URL 地址为 tasks 。因为是 REST 代理，它将会根据不同的操作访问相应的 HTTP 服务。

例如，当你删除一条记录，这个服务请求 URL 将是 <base URL>/task/{id} 。如果你的应用部署在 localhost 端口为 9001 ，那么请求 URL 就是 http://localhost:9001/tasks/23333 ，并且 HTTP verb 是 DELETE ，你可以理解为提交的 HTTP 动作类型。这里 23333 是记录的ID。当你添加一条记录，URL 就是 http://localhost:9001/tasks ，这时 HTTP verb 就是 POST，同时要添加的记录也会以 JSON 形式发送给服务器：

1. Ext.define('ToDo.view.toDoList.ToDoListModel', {
2. extend: 'Ext.app.ViewModel',
3. alias: 'viewmodel.todoList',
4. stores: {
5. todos: {
6. fields: [ {
7. name: 'id',
8. type: 'string'
9. },{
10. name: 'desc',
11. type: 'string'
12. }],
13. autoLoad: **true**,
14. sorters: [{
15. property: 'done',
16. direction: 'ASC'
17. }],
18. proxy: {
19. type: 'rest',
20. url: 'tasks',
21. reader: {
22. type: 'json',
23. },
24. writer: {
25. type: 'json'
26. }
27. }
28. }
29. }
30. });

现在我们来创建视图，我们将创建一个 view 作为 To do 待办 列表，一个文本框用于键入新的待办，和一个添加按钮。

这个 To do 待办 列表的 UI 是基于 store 中的记录，动态的创建的，并且每当一个记录添加或移除，这个 view 都会相应的更新。Ext JS grid 组件可以实现我们的目的，但是你还没有学习过这个组件，所以这里我们不使用它来创建，而通过其他方式，你还将学习到如何处理自定义 UI。

动态创建 待办列表 UI，我把代码写在 ViewController 里。在这个 view 中，我添加了 文本框 和 按钮：

1. Ext.define('ToDo.view.toDoList.ToDoList', {
2. extend: 'Ext.panel.Panel',
4. /\* Marks these are required classes to be to loaded before loading this view \*/
5. requires: [
6. 'ToDo.view.toDoList.ToDoListController',
7. 'ToDo.view.toDoList.ToDoListModel'
8. ],
9. xtype: 'app-todoList',
10. controller: 'todoList',
11. /\* View model of the view. \*/
12. viewModel: {
13. type: 'todoList'
14. },
15. items: [{
16. xype: 'container',
17. items: [{
18. xtype: 'container',
19. layout: 'hbox',
20. cls: 'task-entry-panel',
21. defaults: {
22. flex: 1
23. },
24. items: [{
25. reference: 'newToDo',
26. xtype: 'textfield',
27. emptyText: 'Enter a new todo here'
28. },{
29. xtype: 'button',
30. name: 'addNewToDo',
31. cls: 'btn-orange',
32. text: 'Add',
33. maxWidth: 50,
34. handler: 'onAddToDo'
35. }]
36. }]
37. }]
38. });

上面的 view 代码里，我指定了两个 cls 属性分别是 btn-orange 和 task-entry-panel 。这是 CSS 类，用于添加一些 CSS 样式。在上面的 UI 截图中，你可以看到 Add 按钮并不是我们所用的主题（crisp）的默认颜色。因为我指定了 CSS 类来自定义这个按钮了。

现在我们可以创建 ViewController 。我们将通过在初始化函数中读取 store 中的记录来创建 待办列表 的UI 。这里，一旦应用加载，我们将调用 store 的 load 方法。这将使得 rest 服务调用，从服务器获取记录，并遍历结果，每一条记录我们都会创建一行在 UI 中。

提供删除功能，我会在每一条记录上添加一个删除图标。下列代码用于绑定这个图标的点击事件。待办记录 UI 是动态生成的，所以我们需要事件委派，如以下代码所示：

1. Ext.getBody().on('click', **function** (event, target) {
2. me.onDelete(event, target);
3. } , **null**, {delegate: '.fa-times' });

如果 UI 不是动态添加的，正常的绑定点击事件，如以下代码：

1. Ext.select('fa-times').on('click', **function** (event, target) {
2. me.onDelete(event, target);
3. });

以下是视图待办列表（ToDoList）  的视图控制器（ViewController）的代码。  onAddToDo 方法是添加按钮的点击事件处理，方法里通过 lookupReference 方法传递引用名称设置到 ToDoList 视图：

1. Ext.define('ToDo.view.toDoList.ToDoListController', {
2. extend: 'Ext.app.ViewController',
3. alias: 'controller.todoList',
4. views: ['ToDo.view.toDoList.ToDoList'],
5. init: **function** () {
6. **var** me = **this**;
7. //Here we're calling the load method of the store and passing an anonymous method for the callback. So, the load will call the server to get the data first and calls the anonymous method. The anonymous method then, add the data to the view.
8. **this**.getViewModel().data.todos.load(**function** (records) {
9. Ext.each(records, **function** (record) {
10. //Add a container for each record
11. me.addToDoToView(record);
12. });
14. });
16. Ext.getBody().on('click', **function** (event, target) {
17. me.onDelete(event, target);
18. }, **null**, {
19. delegate: '.fa-times'
20. });
21. },
22. onAddToDo: **function** () {
23. **var** store = **this**.getViewModel().data.todos;
24. //newToDo 就是文本框的 引用
25. **var** desc = **this**.lookupReference('newToDo').value.trim();
26. **if** (desc != '') {
27. store.add({
28. desc: desc
29. });
30. store.sync({
31. success: **function** (batch, options) {
32. **this**.lookupReference('newToDo').setValue('');
33. **this**.addToDoToView(options.operations.create[0]);
34. },
35. scope: **this**
36. });
37. }
38. },
39. addToDoToView: **function** (record) {
40. **this**.view.add([{
41. xtype: 'container',
42. layout: 'hbox',
43. cls: 'row',
44. items: [{
45. xtype: 'checkbox',
46. boxLabel: record.get('desc'),
47. checked: record.get('done'),
48. flex: 1
49. },{
50. html: '<a class="hidden" href="#"><i taskId="' + record.get('id') + '" class="fa fa-times"></i></a>',
51. }]
52. }]);
53. },
54. onDelete: **function** (event, target) {
55. **var** store = **this**.getViewModel().data.todos;
56. **var** targetCmp = Ext.get(target);
57. **var** id = targetCmp.getAttribute('taskId');
58. store.remove(store.getById(id));
59. store.sync({
60. success: **function** () {
61. **this**.view.remove(targetCmp.up('.row').id)
62. },
63. scope: **this**
64. });
65. }
66. });

最后，让我们来创建 REST 服务。Go 语言可以安装在 mac OS，Windows，linux 等等。这里下载 [https://golang.org](https://golang.org/).

 Go 语言安装完成之后，你需要设置 GOROOT 环境变量为 Go 语言的安装目录。linux 应该添加下列命令到 $HOME/.profile:

1. export GOROOT=/usr/local/go export PATH=$PATH:$GOROOT/bin

 针对本项目，我将使用一个名为 Gorilla 的路由模块，安装这个模块使用以下命令：

1. go get github.com/gorilla/mux

 以下是 REST 服务的代码。

* 这段代码不会存数据到数据库里，所有的数据都是在内存中，当你关闭程序，数据将被销毁。

1. package main
2. import ( "fmt"
3. "encoding/json"
4. "net/http" "strconv"
5. "github.com/gorilla/mux"
6. )
7. type Task struct {
8. Id string `json:"id"`
9. Desc string `json:"desc"`
10. Done bool `json:"done"`
11. }
12. var tasks map[string] \*Task
13. func GetToDo(rw http.ResponseWriter, req \* http.Request) {
14. vars := mux.Vars(req)
15. task := tasks[vars["id"]]
16. js, err := json.Marshal(task)
17. if err != nil {
18. http.Error(rw, err.Error(), http.StatusInternalServerError)
19. return
20. }
21. fmt.Fprint(rw, string(js))
22. }
24. func UpdateToDo(rw http.ResponseWriter, req \* http.Request) {
25. vars := mux.Vars(req)
26. task:= tasks[vars["id"]]
27. dec:= json.NewDecoder(req.Body)
28. err:= dec.Decode( & task)
29. if err != nil {
30. http.Error(rw, err.Error(), http.StatusInternalServerError)
31. return
32. }
33. task.Id = vars["id"]
34. retjs, err:= json.Marshal(task)
35. if err != nil {
36. http.Error(rw, err.Error(), http.StatusInternalServerError)
37. return
38. }
39. fmt.Fprint(rw, string(retjs))
40. }
42. func DeleteToDo(rw http.ResponseWriter, req \* http.Request) {
43. vars := mux.Vars(req)
44. delete(tasks, vars["id"])
45. fmt.Fprint(rw, "{status : 'success'}")
46. }
48. func AddToDo(rw http.ResponseWriter, req \* http.Request) {
49. task:= new(Task)
50. dec:= json.NewDecoder(req.Body)
51. err:= dec.Decode( & task)
52. if err != nil {
53. http.Error(rw, err.Error(), http.StatusInternalServerError)
54. return
55. }
56. tasks[task.Id] = task
57. retjs, err:= json.Marshal(task)
58. if err != nil {
59. http.Error(rw, err.Error(), http.StatusInternalServerError)
60. return
61. }
62. fmt.Fprint(rw, string(retjs))
63. }
65. func GetToDos(rw http.ResponseWriter, req \* http.Request) {
66. v := make([]\*Task, 0, len(tasks))
67. for \_, value := range tasks {
68. v = append(v, value)
69. }
70. js, err:= json.Marshal(v)
71. if err != nil {
72. http.Error(rw, err.Error(), http.StatusInternalServerError)
73. return
74. }
75. fmt.Fprint(rw, string(js))
76. }
78. func main() {
79. var port = 9001
80. router:= mux.NewRouter()
81. tasks = make(map[string] \*Task)
82. router.HandleFunc("/tasks", GetToDos).Methods("GET")
83. router.HandleFunc("/tasks", AddToDo).Methods("POST")
84. router.HandleFunc("/tasks/{id}", GetToDo).Methods("GET")
85. router.HandleFunc("/tasks/{id}", UpdateToDo).Methods("PUT")
86. router.HandleFunc("/tasks/{id}", DeleteToDo).Methods("DELETE")
87. router.PathPrefix("/").Handler(http.FileServer(http.Dir("../")))
88. fmt.Println("Listening on port", port)
89. http.ListenAndServe("localhost:" + strconv.Itoa(port), router)
90. }

使用下列命令运行服务：

1. go run ToDo.go

如果没有报错，应该显示类似于以下代码：

1. Listening on port 9001

 现在你可以访问 localhost:9001 查看应用了：

完整的代码在这里 [https://github.com/ananddayalan/extjs-byexample-todo.](https://github.com/ananddayalan/extjs-by-example-todo)

总结

在本章中，你学习了如何创建 model，store，代理和如何处理数据。同时也看到如何使用 Go 语言创建一个简单的 REST 服务。

# 第5章 - 表格组件(grid)

本章将探索 Ext JS 的高级组件 grid 。还将使用它帮助读者建立一个功能齐全的公司目录。本章介绍下列几点主题：

* 基本的 grid
* 排序
* 渲染器
* 过滤
* 分页
* 单元格编辑
* 行编辑
* 分组
* 分组 grid（pivot grid）
* 公司目录 —一个示例项目

grid 组件是 Ext JS 中最强大的一个组件。它有很多的选项和配置，能以任何你希望的形式来构建 grid。

Ext JS 的 grid 组件提供完整的支持分页，排序，过滤，搜索，编辑行，编辑单元格，分组，工具栏，滚动缓冲，列的调整和隐藏，列分组，多重排序，行扩展等，强大且实用的功能。

本章的主要目标是为你展示 Ext JS 的 grid 组件的功能以及如何利用这些提供的功能来构建一个简单的应用 公司目录(Company Directory)。最终的设计如下图所示：

这个示例项目中使用的最重要的组件就是 grid ，分页栏，和行编辑插件。同时这个示例项目中也涉及了其他的前面章节所学过的内容。

基本的 grid

让我们从最基本的 grid 开始，创建一个 grid 组件，使用 Ext.grid.Panel ，你至少需要制定 grid 面板有哪些列和指定 store 来获取数据。这个类名，在现代化工具包（modern toolkit）中是 Ext.grid.Grid，但是在经典工具包（classic toolkit）中是 Ext.grid.Panel 。这两者有一些细微的区别，但是大多数概念是相同的。

 我们首先创建一个 store 使用内置硬编码的 model 数据。

下列 model 使用的三个字符串类型的字段，store 使用内置数据并指定了所使用的 model ：

1. Ext.define('Product', {
2. extend: 'Ext.data.Model',
3. fields: [ 'id', 'productname', 'desc', 'price' ]
4. });
5. **var** productStore = Ext.create('Ext.data.Store', {
6. model: 'Product',
7. data: [{
8. id: 'P1',
9. productname: 'Ice Pop Maker',
10. desc: 'Create fun and healthy treats anytime',
11. price: '$16.33'
12. }, {
13. id: 'P2',
14. productname: 'Stainless Steel Food Jar',
15. desc: 'Thermos double wall vacuum insulated food jar',
16. price: '$14.87'
17. },{
18. id: 'P3',
19. productname: 'Shower Caddy',
20. desc: 'Non-slip grip keeps your caddy in place',
21. price: '$17.99'
22. }, {
23. id: 'P4',
24. productname: 'VoIP Phone Adapter',
25. desc: 'Works with Up to Four VoIP Services Across One Phone Port',
26. price: '$47.50'
27. }]
28. });

OK，现在可以使用 Ext.grid.Panel 来创建 grid 了，记得要指定 store 。每一列都可以设置 width 和 flex 。我们将 ‘Description’ 列设置 flex 为 1 ，指定这个属性，这一列会使用其他两列余下的所有宽度。

列的 dataIndex 属性是指定对应 Product 模型中的哪个字段。为 id 列设置 hidden 属性 为 true 使该列保持隐藏，如以下代码所示：

1. Ext.create('Ext.grid.Panel', {
2. renderTo: Ext.getBody(),
3. store: productStore,
4. width: 600,
5. title: 'Products',
6. columns: [{
7. text: 'Id',
8. dataIndex: 'id',
9. hidden: **true**
10. },{
11. text: 'Name',
12. width:150,
13. dataIndex: 'productname'
14. },{
15. text: 'Description',
16. dataIndex: 'desc',
17. sortable: **false**,
18. flex: 1
19. },{
20. text: 'price',
21. width: 100,
22. dataIndex: 'price'
23. }]
24. });

下列截图为以上代码的输入。默认的列宽度是可调整的；如果需要你可以指定一个固定宽度。

排序

这是默认的列菜单，可以用来排序，隐藏或显示列。点击每一列上的细箭头图标，这个列菜单就会显示。排序功能可以从界面或代码上添加：

使用代码的写法，为列 Description 设置  sortable 属性为 false  ，此选项将为这个列隐藏排序。

默认的排序是客户端排序。如果想开启服务端排序，你需要为 store 设置 remoteSort 属性 为 true 。这时候你界面选择排序后，store 将会把排序信息（字段和排序次序）作为参数发送到服务器。

renderer

列的 renderer 配置可以用来改变列内容的呈现方式。你也许已经注意到 price 列并没有显示货币符号。现在我们使用 renderer 属性为 price 列的值增加  $ 作为前缀：

1. Ext.create('Ext.grid.Panel', {
2. renderTo: Ext.getBody(),
3. store: productStore,
4. width: 600,
5. title: 'Products',
6. columns: [{
7. text: 'Id',
8. dataIndex: 'id',
9. hidden: **true**
10. }, {
11. text: 'Name',
12. width:150,
13. dataIndex: 'productname'
14. }, {
15. text: 'Description',
16. dataIndex: 'desc',
17. sortable: **false**,
18. flex: 1
19. }, {
20. text: 'price',
21. width: 100,
22. dataIndex: 'price',
23. renderer: **function**(value) {
24. **return** Ext.String.format('${0}', value);
26. }
27. }]
28. });

输出如下所示：

同样的，你可以使用 renderer 来描绘 HTML 标签到列中。还有 URL 和图片。

过滤

通过添加 Ext.grid.filters.Filters (ptype: gridfilters)  插件到 grid，并对列进行配置可以很容易实现过滤功能：

1. Ext.create('Ext.grid.Panel', {
2. renderTo: Ext.getBody(),
3. store: productStore,
4. plugins: 'gridfilters',
5. width: 600,
6. title: 'Products',
7. columns: [{
8. text: 'Id',
9. dataIndex: 'id',
10. hidden: **true**
11. },{
12. text: 'Name',
13. width:150,
14. dataIndex: 'productname',
15. filter:'string'
16. },{
17. text: 'Description',
18. dataIndex: 'desc',
19. sortable: **false**,
20. flex: 1,
21. filter: {
22. type: 'string',
23. itemDefaults: {
24. emptyText: 'Search for…'
25. }
26. }
27. },{
28. text: 'price',
29. width: 100,
30. dataIndex: 'price'
31. }]
32. });

对每一列你都可以指定过滤类型，例如 string， bool， 等等，还有检索字段的额外配置（例如 emptyText 就是空文本时的提示内容）。

这里是演示的在创建时添加过滤器，但也是可以在 grid 创建后再添加的。

分页

如果你有成千上万的数据，你肯定不想一次加载上万条记录(这句是废话啊)，更好的方法是添加分页工具栏，或者使用缓冲滚动条。

现在我们在之前的例子上添加分页工具栏 Ext.toolbar.Paging (xtype: pagingtoolbar)。这里我们将会把分页工具栏添加到 dockedItems ，它是 Ext.panel.Panel 的一个属性， 并且 dockedItems 里配置的项可以设置停驻在 panel 的任意 上，下，左或右边。

1. Ext.create('Ext.grid.Panel', {
2. renderTo: Ext.getBody(),
3. store: productStore,
4. width: 600,
5. title: 'Products',
6. columns: [{
7. text: 'Id',
8. dataIndex: 'id',
9. hidden: **true**
10. },{
11. text: 'Name',
12. width:150,
13. dataIndex: 'productname'
14. },{
15. text: 'Description',
16. dataIndex: 'desc',
17. sortable: **false**,
18. flex:1
19. },{
20. text: 'price',
21. width: 100,
22. dataIndex: 'price'
23. }],
24. dockedItems: [{
25. xtype: 'pagingtoolbar',
26. store: productStore,
27. dock: 'bottom',
28. displayInfo: **true**
29. }]
30. });

然后，下面这是 store 的代码，store 从 JSON 文件中读取数据：

1. **var** productStore = Ext.create('Ext.data.Store', {
2. model: 'Product',
3. pageSize: 10,
4. autoLoad: **true**,
5. proxy: {
6. type: 'ajax',
7. url: 'data.json',
8. reader: {
9. type: 'json',
10. rootProperty: 'data',
11. totalProperty: 'total'
12. }
13. }
14. });

rootProperty 属性告诉 store 从 JSON 文件中哪个地方查找记录，同时 totalProperty 属性让 store 知道从哪里读取记录总数。 为了得到正确的结果，当你使用服务器分页时，需要指定 totalProperty 属性，因为这个值将用于分页工具栏。

pageSize 属性的值为 10 它为每页显示的记录数。

* 这个参数用于服务器处理分页，并且返回所请求的数据，而不是整个集合。如果你是从文件中读，那么响应将包含 JSON 文件中的所有记录。

使用了分页工具栏的结果输出：

单元格编辑

grid 中的单元格编辑插件为 Ext.grid.plugin.CellEditing 。可以为 grid 添加单元格编辑的支持。

OK，现在为 grid 添加单元格编辑的功能，简单的添加 cellditing 插件并为必要的列设置编辑器。你可以添加所有列都可以支持单元格编辑，或者为指定列添加编辑器。

单元格编辑器可以是简单的文本框，或者你也可以添加一些复杂的组件，例如 combobox（附带的为 combobox 添加 store 支持）。

下列例子添加了一个文本框编辑器到一个列，和添加了一个 combobox （附带的为 combobox 添加 store 支持）到另一个列：

1. Ext.create('Ext.grid.Panel', {
2. renderTo: Ext.getBody(),
3. store: productStore,
4. plugins: ['cellediting','gridfilters'],
5. width: 600,
6. title: 'Products',
7. columns: [{
8. text: 'Id',
9. dataIndex: 'id',
10. hidden: **true**
11. },{
12. text: 'Name',
13. width:150,
14. dataIndex: 'productname',
15. filter:'string',
16. editor: {
17. allowBlank: **false**,
18. type: 'string'
19. }
20. },{
21. text: 'Description',
22. dataIndex: 'desc',
23. sortable: **false**,
24. flex: 1
25. },{
26. text: 'Price',
27. width: 100,
28. dataIndex: 'price'
29. },{
30. text: 'Type',
31. width: 100,
32. dataIndex: 'type',
33. editor: **new** Ext.form.field.ComboBox({
34. typeAhead: **true**,
35. triggerAction: 'all',
36. store: [
37. [ 'Bath','Bath'],
38. [ 'Kitchen','Kitchen'],
39. [ 'Electronic','Electronic'],
40. [ 'Food', 'Food'],
41. [ 'Computer', 'Computer' ]
42. ]
44. })
45. }]
46. });

上面的例子中，以下代码是用于设置编辑器的 Type ，不过这里我们使用了硬编码的内置数据，但是它是可以配置从服务器读取数据的：

1. editor: **new** Ext.form.field.ComboBox({
2. typeAhead: **true**,
3. triggerAction: 'all',
4. store: [
5. ['Bath','Bath'],
6. ['Kitchen','Kitchen'],
7. ['Electronic','Electronic'],
8. ['Food', 'Food'],
9. ['Computer', 'Computer']
10. ]
11. })

Type 可以是任意的受支持的表单组件，例如日期选择器，复选框，单选按钮，数字文本框等等。另外也可以为编辑器添加表单校验功能。

 截图为以上代码的输出结果：

一旦编辑了记录，默认不会存到服务器。你可以设置 store 的 autosync 属性为 true ，这将会触发一个 CRUD 的请求到服务器。

如果你不希望同步立即发生，那么你可以在需要时调用 store 的 save 或 sync 方法。例如可以在 grid 的头部添加 Save 按钮 来调用 store 的 save 或 sync 方法。

文本框编辑器，注意有一个小红色标记在第一行第一列的左上角。这个标记是让用户知道这行记录有更新。

 行编辑

在单元格编辑的一节中，你每次只能编辑一个单元格，但是在行编辑中，你每次可以编辑一行内容。

使用行编辑插件：Ext.grid.plugin.RowEditing 替换掉单元格编辑插件。你只需要在上面的代码中把 [‘cellediting’,‘gridfilters’]  替换为  [‘rowediting’,‘gridfilters’] 你将得到以下输出：

 这个行编辑插件也会应用到本章接下来的示例项目中，你会在那里找到额外的一些内容。

分组

为了对列进行分组，你需要在 store 的 groupField 属性上指定分组字段，并且我们需要在 grid 设置 Ext.grid.feature.Feature ，如以下代码所示：

1. **var** productStore = Ext.create('Ext.data.Store', {
2. model: 'Product',
3. pageSize: 10,
4. autoLoad: **true**,
5. proxy: {
6. type: 'ajax',
7. url: 'data.json',
8. reader: {
9. type: 'json',
10. rootProperty: 'data',
11. totalProperty: 'total'
12. }
13. },
14. groupField: 'type'
15. });
17. Ext.create('Ext.grid.Panel', {
18. renderTo: Ext.getBody(),
19. store: productStore,
20. width: 600,
21. title: 'Products',
22. features: [{
23. id: 'group',
24. ftype: 'grouping',
25. groupHeaderTpl : '{name}',
26. hideGroupedHeader: **true**,
27. enableGroupingMenu: **false**
28. }],
29. columns: [{
30. text: 'Id',
31. dataIndex: 'id',
32. hidden: **true**
33. },{
34. text: 'Name',
35. width:150,
36. dataIndex: 'productname'
37. },{
38. text: 'Description',
39. dataIndex: 'desc',
40. sortable: **false**,
41. flex: 1,
42. groupable:**false**
43. },{
44. text: 'Price',
45. width: 100,
46. dataIndex: 'price'
47. },{
48. text: 'Type',
49. width: 100,
50. dataIndex: 'type'
51. }]
52. });

以上代码所示的输出：

 下图显示了分组菜单。使用此菜单，可以在运行时通过 grid 的其他字段分组。这个选项可以设置 enableGroupingMenu 属性为 false 来关闭，上面的代码已经有例子了。

分组的模板可以用来添加一些额外的信息，例如分组后的数量或其他任何你想要的信息，通过 groupHeaderTpl : ‘{name}’，来实现。现在将上面的代码修改为 groupHeaderTpl : ‘{columnName}: {name} ({rows.length} Item{[values. rows.length > 1 ? “s” : “”]})’,  得到下列输出：

 pivot grid

这可以让你重新整理和汇总选定的列数据和行数据，以获得所需的报表。

比方说你有一个公司的员工提交的费用数据列表，然后你想看到每个类别，每个员工要求的总费用合计。

预期的结果的一个例子如下截图所示：

在 grid 里，你需要组织和总结列得到这样的结果，而不是得到费用列表。使用 pivot grid 可以很容易做到。注意除了第一列，其他的列标题的值为这个费用列表中费用类别的值，所以，你看到的数据是已经重新汇总和整理过的。

当你使用 pivot grid ，有两个重要的事需要你准备： axis(轴)和aggregates(聚合) 。 你应该使用 axis 指定行和列的位置以及使用聚合进行分组计算。

这是一个例子：

1. leftAxis: [{   width: 80,   dataIndex: 'employee',   header: 'Employee'}]
3. topAxis: [{   dataIndex: 'cat',   header: 'Category',   direction: 'ASC' }]

在轴中你可以指定排序，排序方向，过滤器等等。

1. aggregate: [{
2. measure: 'amount',
3. header: 'Expense',
4. aggregator: 'sum',
5. align: 'right',
6. width: 85,
7. renderer: Ext.util.Format.numberRenderer('0,000.00')
8. }

里聚合可以进行 sum ，avg ， min ， max ，等等and so on.

 在 pivot grid 中，你也可以指定 renderer 属性自定义格式化数据：

1. renderer: **function**(value, meta, record) {
2. **return** Ext.util.Format.number(value, '0,000.00');
3. }

现在可以创建 pivot grid 了。 这是用于 pivot grid 的 store 组件：

1. **var** store = **new** Ext.data.JsonStore({
2. proxy: {
3. type: 'ajax',
4. url: 'expense.json',
5. reader: {
6. type: 'json',
7. rootProperty: 'rows'
8. }
9. },
10. autoLoad: **true**,
11. fields: [{
12. name: 'id',
13. type: 'int'
14. },{
15. name: 'employee',
16. type: 'string'
17. },{
18. name: 'amount',
19. type: 'int'
20. },{
21. name: 'date',
22. type: 'date',
23. dateFormat: 'd/m/Y'
24. },{
25. name: 'cat',
26. type: 'string'
27. },{
28. name: 'year',
29. convert: **function**(v, record){
30. **return** Ext.Date.format(record.get('date'), "Y");
31. }
32. }]
33. });

下面是 pivot grid 的例子代码。这里你可以看到 leftAxis 是固定的，而 topAxis 为动态的，基于一个下拉选择来改变 topAxis 。

1. **var** pivotGrid = Ext.create('Ext.pivot.Grid', {
2. renderTo: Ext.getBody(),
3. title: 'Pivot Grid - Employee Expense Claims',
4. height: 600,
5. width: 700,
6. enableLocking: **false**,
7. viewConfig: {
8. trackOver: **true**,
9. stripeRows: **false**
10. },
11. tbar: [{
12. xtype: 'combo',
13. fieldLabel: 'Select report',
14. flex: 1,
15. editable: **false**,
16. value: '1',
17. store: [
18. ['1', 'How much an employee claimed in total?'],
19. ['2', 'What are the expense amounts of each employee in each category?'],
20. ['3', 'How much an employee claimed in a specific year?']
21. ],
22. listeners: {
23. select: **function**(combo, records, eOpts){
24. **switch**(records.get('field1')){
25. **case** '1':
26. pivotGrid.reconfigurePivot({
27. topAxis: []
28. });
29. **break**;
30. **case** '2':
31. pivotGrid.reconfigurePivot({
32. topAxis: [{
33. dataIndex: 'cat',
34. header: 'Category',
35. direction: 'ASC'
36. }]
37. });
38. **break**;
39. **case** '3':
40. pivotGrid.reconfigurePivot({
41. topAxis: [{
42. dataIndex: 'year',
43. header: 'Year',
44. direction: 'ASC'
45. }]
46. });
47. **break**;
48. }
49. }
50. }
51. }],
52. store: store,
53. aggregate: [{
54. measure: 'amount',
55. header: 'Expense',
56. aggregator: 'sum',
57. align: 'right',
58. width: 85,
59. renderer: Ext.util.Format.numberRenderer('0,000.00')
60. }],
61. leftAxis: [{
62. width: 80,
63. dataIndex: 'employee',
64. header: 'Employee'
65. }],
66. topAxis: []
67. });

下面的截图显示以上代码的输出：

公司目录 – 一个示例项目

好了，现在我们将使用所学知识创建一个示例项目。以下截屏显示示例项目的最终效果：

通过查看设计效果，想必你已找到这个项目中最重要的组件就是 grid 了。本示例中所应用的组件和概念有：

* Grid panel
* ViewModel
* Model
* 数据 store 和 rest 代理
* 容器和布局
* 行编辑插件
* 分页
* go 语言实现的 REST API
* 参考资料

 下列截图展示了使用了行编辑插件的 修改/新增 操作：

现在让我们看看这个示例项目中的一些重要文件。

* 完整的源码没有在这里提供，但是你可以从此地址去下载。 [https://github.com/ ananddayalan/extjs-by-example-companydirectory](https://github.com/%20ananddayalan/extjs-by-example-companydirectory)

 截图展示了示例项目的文件结构：

以下视图代码是本例中的重要部分。这个视图显示了这个应用的大部分的可视部分。它使用了 grid panel 和 行编辑插件：

plugins: [{   ptype: ‘rowediting’,   clicksToMoveEditor: 1,   autoCancel: false }] ,

为 grid 添加行编辑功能，最简单的方法就是使用 RowEditing 。使用 RowEditing 插件并不是必要的，如果你想，你也可以使用表单来做 新增/修改。

为了能够使用编辑功能，我们还需要设置字段的属性或者列的 editor 属性。下面这一行默认为所有的列都提供编辑功能：

editor: { xtype: ‘textfield’, allowBlank: false }

 在编辑器中设置校验规则，这样可以只允许满足验证的数据提交。另外 Update 按钮在插件里将是不可用的。

* 这个视图的代码有点长，这段代码中我已经截出来了重要部分的代码，完整的代码在这里。 [https://github.com/ ananddayalan/extjs-by-example-company-directory](https://github.com/%20ananddayalan/extjs-by-example-company-directory)

1. Ext.define('CD.view.contactList.ContactList', {
2. extend: 'Ext.panel.Panel',
3. requires: ['CD.view.contactList.ContactListController' ],
4. xtype: 'app-contactList',
5. controller: 'contactList',
6. items: [{
7. cls: 'contact-list',
8. xtype: 'grid',
9. reference: 'contactListGrid',
10. scrollable: **true**,
11. autoScroll: **true**,
12. plugins: [{
13. ptype: 'rowediting',
14. clicksToMoveEditor: 1,
15. autoCancel: **false**
16. }],
17. listeners: {
18. selectionchange: 'onSelectionChange'
19. },
20. flex:1,
21. store: 'contactList',
22. pageSize: 10,
23. title: 'Company Directory',
24. columns: {
25. defaults: {
26. editor: {
27. xtype: 'textfield',
28. allowBlank: **false**
29. }
30. },
31. items: [{
32. text: 'First Name',
33. width: 100,
34. dataIndex: 'fname'
35. },{
36. text: 'Email',
37. width: 250,
38. dataIndex: 'email',
39. editor: {
40. vtype: 'email'
41. }
42. }]
43. },
44. dockedItems: [{
45. xtype: 'pagingtoolbar',
46. store: 'contactList',
47. dock: 'bottom',
48. displayInfo: **true**
49. },{
50. xtype: 'toolbar',
51. dock: 'top',
52. ui: 'footer',
53. defaults: {
54. cls: 'btn-orange'
55. },
56. items: ['->', {
57. text: 'Remove',
58. disabled: **true**,
59. reference: 'btnRemoveContact',
60. listeners: {
61. click: 'onRemove'
62. },
63. }]
64. }]
65. }]
66. });

以上代码中，grid 使用了两个工具栏：一个分页工具栏和一个 grid 上方的包含按钮的工具栏。这些工具栏使用 dockedItems 属性配置，以下为示例代码。

在本章的早些时候你学过这个。‘dockedItems’  是 panel 的属性；它允许你设置一个组件停驻在 panel 上下，左边或右边。尽管通常它用于工具栏，但你可以停驻任意你想要的组件。

* 分页工具栏为了正确设置页码，数据数量等等，需要设置 store 。

1. dockedItems: [{
2. xtype: 'pagingtoolbar',
3. store: 'contactList',
4. dock: 'bottom',
5. displayInfo: **true**
6. } , {
7. xtype: 'toolbar',
8. dock: 'top',
9. ui: 'footer',
10. //This sets style to the component. The 'ui' is a property of the component. The default value of this property for all the component is 'default'. For details are given in the chapter that focus on theming and styling.
11. defaults: { cls: 'btn-orange' },
12. items: ['->', {
13. text: 'Remove',
14. disabled: **true**,
15. //We set disabled by default, and this will be enabled when a row in the grid is selected. Check the onSelectionChange method in the controller.     reference: 'btnRemoveContact',     listeners: { click: 'onRemove' },
16. }]
17. }]

这个 ViewController 的代码非常简单。它只处理了添加，删除，和  ContactList 视图的 selection change 事件。

看一下视图引用是如何访问到控制器的。例如，下列代码返回 grid 的引用：

var grid = this.lookupReference(‘contactListGrid’);

在这儿，contactListGrid 在前面视图中标记为一个引用：

下列代码中，访问 store 使用 grid.getStore() ； 也可以使用 Ext.getStore(contactList) 访问 store ：

1. Ext.define('CD.view.contactList.ContactListController', {
2. extend: 'Ext.app.ViewController',
3. alias: 'controller.contactList',
4. views: ['CD.view.contactList.ContactList'],
5. requires: ['CD.store.ContactList'],
6. onSave: **function**() {
7. //Note, this will trigger one or more calls to the server based on the number of operations performed in the store.
8. Ext.getStore('contactList').save();
9. },
10. onSelectionChange: **function**() {
11. **this**.lookupReference('btnRemoveContact').enable();
12. },
13. onRemove: **function**() {
14. **var** grid = **this**.lookupReference('contactListGrid');
15. **var** sm = grid.getSelectionModel();
16. //This line cancels the row/cell edit if it is active before we remove the item.
17. grid.plugins[0].cancelEdit();
18. grid.getStore().remove(sm.getSelection());
19. **if** (grid.getStore().getCount() > 0) {
20. sm.select(0);
21. }
22. },
23. onCreate: **function**() {
24. **var** grid = **this**.lookupReference('contactListGrid');
25. grid.plugins[0].cancelEdit();
26. //Create a model instance
27. **var** r = Ext.create('Contact');
28. grid.getStore().insert(0, r);
29. grid.plugins[0].startEdit(0, 0);
30. }
31. });

模型和视图的代码如下所示。rootProperty 属性让 store 知道在 API 响应的 JSON 数据中从哪里开始读取数据。

1. Ext.define('Contact', {
2. extend: 'Ext.data.Model',
3. fields: ['fname', 'lname', 'email', 'address','city','state','phone','type']
4. });
5. Ext.define('CD.store.ContactList', {
6. extend: 'Ext.data.Store',
7. storeId: 'contactList',
8. model: 'Contact',
9. pageSize: 10,
10. proxy: {
11. type: 'rest',
12. url: 'contactlist',
13. reader: {
14. type: 'json',
15. rootProperty: 'data',
16. totalProperty: 'total'
17. }
18. }
19. });
20. Ext.create('CD.store.ContactList').load();

totalProperty 字段让 store 知道服务器可获得的记录总数。grid 的分页工具栏会控制在视图中显示的分页信息例如 “Displaying 1 to 10 of 50” ；这里，50 是 totalProperty 的值。同时基于这个值，grid 知道什么时候该禁用下一页的按钮。

当你不指定模型的字段的类型。默认的类型将是自动的，如果需要你可以指定类型和验证规则。

再来一次，完整的源码在这里啊。 <https://github.com/ananddayalan/extjs-byexample-company-directory>

总结

在本章中我们详细探讨了 grid 。grid 有许多有用的配置和选项，还有 grid 的过滤，排序，和分组功能。你还学习了如何在 grid 使用不同的插件。还有用来做数据汇总报表的 pivot grid 。以及最后的示例项目。

# 第6章 - 高级组件

本章涵盖了高级组件，比如 tree 和 data view。它将为读者呈现一个示例项目为 图片浏览器，它使用 tree 和 data view 组件。以下是本章将要讨论的主题：

* Trees
* Data views
* 拖放
* 图片浏览器 — 一个示例项目

本章的主要目标是探索 tree panel 和 data view 并且使用他们来构建一个示例项目图片浏览器。图片浏览器的最终展示效果如下图。

这个项目中的最重要的组件是 tree panel 和 data view 。本项目中使用的组件和概念有：

* tree panel
* Data views
* Model
* store 和 rest 代理
* 容器和布局
* 引用
* 事件处理
* 过滤

除了 tree panel 和 data view 你已经在之前的章节中学习了所有的我们目前已用到的知识。所以在本章中，我们首先学习 tree panel 和 data view。

tree panel

这在 ExtJS 中是一个非常强大且常用的组件，你可以使用它构建任意类型的树。一个 tree panel 是一个树形结构的具有层次化数据的UI。

它和 Ext.grid.Panel 相似， Ext.tree.Panel 也继承自Ext.panel.Table 。所以，它也是支持多列的。

和 grid panel 不同的是，tree panel 需要一个 tree store (Ext.Store.TreeStore)。 tree store 具有一些 tree panel 的功能所需使用的特殊的属性。

**基本的 tree**

我们来用一个简单的例子演示。tree panel 至少需要一个 tree store 来提供数据。我们首先来创建 tree store 并硬编码内置数据：

1. **var** store = Ext.create('Ext.data.TreeStore', {
2. root: {
3. expanded: **true**,
4. text: 'Continents',
5. children: [{
6. text: 'Antarctica',
7. leaf: **true**
8. }, {
9. text: 'South America',
10. expanded: **true**,
11. children: [{
12. text: 'Brazil',
13. leaf: **true**
14. }, {
15. text: 'Chile',
16. leaf: **true**
17. }]
18. }, {
19. text: 'Asia',
20. expanded: **true**,
21. children: [{
22. text: 'India',
23. leaf: **true**
24. },{
25. text: 'China',
26. leaf: **true**
27. }]
28. }, {
29. text: 'Africa',
30. leaf: **true**
31. }]
32. }
33. });

接着继续创建 Ext.tree.Panel ：

1. Ext.create('Ext.tree.Panel', {
2. title: 'Basic Tree',
3. width: 200,
4. height: 450,
5. store: store,
6. rootVisible: **true**,
7. renderTo: Ext.getBody()
8. });

下列截图为以上代码的输出结果：

现在，我们创建一个高级点的树，它是可以拖拽的。同时还需要用到 tree panel 和 tree store 的一些额外选项。拖拽只需要添加一个插件叫做 treeviewdragdrop 。如以下代码所示：

1. **var** store = Ext.create('Ext.data.TreeStore', {
2. root: {
3. expanded: **true**,
4. text: 'Continents',
5. checked: **false**,
6. children: [{
7. text: 'Antarctica',
8. leaf: **true** ,
9. checked: **false**
10. },{
11. text: 'South America',
12. expanded: **false**,
13. checked: **true**,
14. children: [{
15. text: 'Chile',
16. leaf: **true**,
17. checked: **true**
18. }]
19. },{
20. text: 'Asia',
21. expanded: **true**,
22. checked: **true**,
23. children: [{
24. text: 'India',
25. leaf: **true**,
26. checked: **true**
27. },{
28. text: 'China',
29. leaf: **true**,
30. checked: **true**
31. }]
32. },{
33. text: 'Africa',
34. leaf: **true**,
35. checked: **true**
36. }]
37. }
38. });
40. Ext.create('Ext.tree.Panel', {
41. title: 'Basic Tree',
42. width: 200,
43. height: 450,
44. store: store,
45. rootVisible: **true**,
46. useArrows: **true**,
47. lines: **false**,
48. renderTo: Ext.getBody(),
49. viewConfig: {
50. plugins: {
51. ptype: 'treeviewdragdrop',
52. containerScroll: **true**
53. }
54. }
55. });

如以下截图所示的输出。我把节点  South America 拖拽至  Asia 节点之下：

tree grid

你可以将多个列添加到 tree ，同时也能创建 tree grid 。默认 tree 包含一列，用的是 tree store 中节点的文本字段。

在这个 store 中，你可以看到在每个节点上除了节点名称，还添加了一些其他的字段，这些字段用于在 tree panel 的列展示上。tree grid 的功能有例如 列调整，排序，过滤等等，以下是代码：

1. **var** store = Ext.create('Ext.data.TreeStore', {
2. root: {
3. expanded: **true**,
4. text: 'Continents',
5. children: [{
6. name: 'Antarctica',
7. population: 0,
8. area: 14,
9. leaf: **true**
10. },{
11. name: 'South America',
12. population: 385 ,
13. area: 17.84,
14. expanded: **false**,
15. children: [{
16. name: 'Chile',
17. population: 18,
18. area: 0.7,
19. leaf: **true**,
20. }]
21. },{
22. name: 'Asia',
23. expanded: **true**,
24. population: 4164,
25. area: 44.57,
26. children: [{
27. name: 'India',
28. leaf: **true**,
29. population: 1210,
30. area: 3.2
31. },{
32. name: 'China',
33. leaf: **true**,
34. population: 1357,
35. area: 9.5
36. }]
37. },{
38. name: 'Africa',
39. leaf: **true**,
40. population: 1110,
41. area: 30
42. }]
43. }
44. });

以下的 grid 和上面的 tree panel 差不多一样，只是添加为多列了，这个 xtyp treecolumn 提供缩进和文件夹结构。像一个正常的 grid 一样，tree grid 的列可以是任意类型的例如 checkbox，picture，button，URL 等等。

默认列大小是可调整的，如果需要你也可以固定它的宽度。看下面的代码：

1. Ext.create('Ext.tree.Panel', {
2. title: 'Tree Grid',
3. width: 500,
4. height: 450,
5. store: store,
6. rootVisible: **false**,
7. useArrows: **true**,
8. lines: **false**,
9. scope: **this**,
10. renderTo: Ext.getBody(),
11. columns: [{
12. xtype: 'treecolumn',
13. text: 'Name',
14. flex: 1,
15. sortable: **true**,
16. dataIndex: 'name'
17. } , {
18. text: 'Population (millons)',
19. sortable: **true**,
20. width: 150,
21. dataIndex: 'population'
22. } , {
23. text: 'Area (millons km^2)',
24. width: 150,
25. sortable: **true**,
26. dataIndex: 'area'
27. }]
28. });

这是上面 Tree Grid 的输出结果：

 Data views

Ext.view.View (xtype:dataview) 一个现实数据的自定义模板。你需要提供自定义的模板和数据源（store）。模板应该使用 Ext.XTemplate 。

data view 提供了内置的事件，例如 click，double-click，mouseover，mouseout，等等。

首先我们创建一个简单的 model 名为 Person ,还需要创建一个 store 并持有 Person 的列表，如以下代码所示：

1. Ext.define('Person', {
2. extend : 'Ext.data.Model',
3. fields : [ {
4. name : 'name',
5. type : 'string'
6. }, {
7. name : 'age',
8. type : 'int'
9. }, {
10. name : 'gender',
11. type : 'int'
12. } ]
13. });
15. Ext.create('Ext.data.Store', {
16. id : 'employees',
17. model : 'Person',
18. data : [{
19. name : 'Mike',
20. age : 22,
21. gender : 0
22. },{
23. name : 'Woo',
24. age : 32,
25. gender : 1
26. },{
27. name : 'John',
28. age : 33,
29. gender : 1
30. },{
31. name : 'Kalai',
32. age : 25,
33. gender : 0
34. }]
35. });

然后我们要来创建这个模板。下列模板使用 HTML 的 table 元素来呈现自定义格式的数据。

在模板中使用一个 model 的字段时，你可以使用花括号包括字段名的方式来使用它，例如：{fieldname}

XTemplate 支持有条件的展现和 if 语句，如以下代码所示：

1. **var** empTpl = **new** Ext.XTemplate(
2. '<tpl for=".">',
3. '<div style="margin-bottom: 10px;" class="data-view">',
4. '<table style="width:100%">',
5. '<tr>',
6. '<td style="font-size: 100px;width:100px;" rowspan="3"><i class="fa fa-user"></i></td>',
7. '<td>Name: {name}< /td>',
8. '</tr>',
10. '<tr>',
11. '<td>Age:{age}< /td>',
12. '</tr>',
14. '<tr>',
15. '<td>Gender: <tpl if="gender == 1">',
16. '<i class="fa fa-mars"></i>',
17. '<tpl else>',
18. '<i class="fa fa-venus"></i>',
19. '</tpl></td>',
20. '</tr></table> ',
21. '</div>',
22. '</tpl>'
23. );

看上面的例子，我使用了 awesome 字体图标的样式。你需要添加下列代码到你的 HTML 文件才行：

1. **<link** rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font- awesome/4.3.0/css/font-awesome.min.css"**>**

一下代码创建了一个 data view，并且它指定了使用的数据源 store ，template 和 itemSelector :

1. Ext.create('Ext.view.View', {
2. store : Ext.getStore('employees'),
3. tpl : empTpl,
4. itemSelector : 'div.data-view',
5. renderTo : Ext.getBody(),
6. listeners : {
7. itemclick : **function**(node, rec, item, index, e) {
8. alert(rec.data.name);
9. }
10. }
11. });

itemSelector 是一个必须的简单 CSS 选择器。这里 itemSelector 是应用于在 template 中的 HTML ，就是使用 data-view 类的 div 标签，最终根据这个模板，你在 data view 中选择的每一个 item ，就是这样一个 div 标签，设置了 itemSelector 属性，data view 会知道如何处理这些节点，itemSelector 是用于将 DOM 节点映射到 records 。

你可以监听的事件例如 click ，double-click ，等等，以上代码已经添加了监听，下列是输出结果：

## 图片浏览器 – 一个示例项目

惯例，我们将用一个示例项目来回顾本章所学，下面是示例项目的最终设计效果：

通过查看这个设计，你会看到我们使用的最重要的组件就是 tree panel 和 data view 。它们如何使用和一些概念已经在本章的前面部分提及。

 我们看看, 项目的目录结构。

* 完整的源码在这里 <https://github.com/ananddayalan/extjs-byexample-picture-explorer>

下列视图代码是本项目的重要部分。这个视图呈现了应用中大部分可视组件。它使用 tree panle 和 data view ：

1. Ext.define('PE.view.pics.Pics', {
2. extend : 'Ext.panel.Panel',
3. /\* Marks these are required classes to be to loaded before loading this view \*/
4. requires : [ 'PE.view.pics.PicsController' ],
5. xtype : 'app-pics',
6. controller : 'pics',
7. items : [ {
8. xtype : 'container',
9. layout : 'hbox',
10. cls : 'pics-list',
11. items : [ {
12. xtype : 'treepanel',
13. width : 200,
14. height : '100%',
15. store : 'albums',
16. border : **true**,
17. useArrows : **true**,
18. cls : 'tree',
19. rootVisible : **false**,
20. listeners : {
21. itemdblclick : 'onNodeSelect'
22. },
23. dockedItems : [ {
24. xtype : 'toolbar',
25. dock : 'top',
26. ui : 'footer',
27. items : [{
28. xtype : 'component',
29. flex : 1
30. },{
31. xtype : 'button',
32. text : 'Upload',
33. cls : 'btn-blue'
34. }]
35. }]
36. },{
37. xtype : 'dataview',
38. reference : 'picsList',
39. cls : 'pics-list-content',
40. store : 'pics',
41. tpl : [
42. '<tpl for=".">',
43. '<div class="thumb"><img src="{url}" title=""></div>',
44. '</tpl>'
45. ],
46. multiSelect : **true**,
47. minHeight : 400,
48. flex : 1,
49. trackOver : **true**,
50. overItemCls : 'x-item-over',
51. itemSelector : 'div.thumb',
52. emptyText : 'No images to display'
53. }]
54. }]
55. });

控制器 ViewController 里处理了 tree panel 的 itemdblclick 事件，只显示所选择节点下的图片。

还有一个 upload 按钮的 click 事件，这里是未处理的。额，这是你的作业啦。看看下列代码：

1. Ext.define('PE.view.pics.PicsController', {
2. extend : 'Ext.app.ViewController',
3. alias : 'controller.pics',
4. views : [ 'PE.view.pics.Pics' ],
5. requires : [ 'PE.store.Pics', 'PE.store.Albums' ],
6. onNodeSelect : **function**(node, rec, item, index, e){
7. **var** albums = [];
8. albums.push(rec.id);
9. rec.childNodes.forEach(**function**(item) {
10. albums.push(item.id);
11. });
13. Ext.getStore('pics').filter({
14. property : 'albumId',
15. operator : 'in',
16. value : albums
17. });
18. }
19. });

Model 和 Store 的代码在这儿。

* 注意：当你不指定 model 的字段类型时，将会自动猜测类型。

1. Ext.define('Pic', {
2. extend : 'Ext.data.Model',
3. fields : [ 'id', 'url', 'albumId' ]
4. });
6. Ext.define('PE.store.Pics', {
7. extend : 'Ext.data.Store',
8. storeId : 'pics',
9. model : 'Pic',
10. proxy : {
11. type : 'rest',
12. url : 'pics', // URL that will load data with respect to start and limit params
13. reader : {
14. type : 'json'
15. }
16. }
17. });
19. Ext.create('PE.store.Pics').load();
21. Ext.define('PE.store.Albums', {
22. extend : 'Ext.data.TreeStore',
23. storeId : 'albums',
24. root : {
25. expanded : **true**,
26. children : [ {
27. id : 100,
28. text : ' California',
29. expanded : **true**,
30. children : [ {
31. id : 600,
32. text : ' Big Sur',
33. leaf : **true**
34. }, {
35. id : 500,
36. text : ' Yosemite',
37. leaf : **true**
38. }]
39. }, {
40. id : 400,
41. text : ' Arizona',
42. expanded : **true**,
43. children : [ {
44. id : 300,
45. text : ' Horseshoe bend',
46. leaf : **true**
47. }]
48. }, {
49. id : 200,
50. text : ' Home',
51. leaf : **true**
52. }, {
53. id : 700,
54. text : ' India',
55. expanded : **true**,
56. children : [ {
57. id : 800,
58. text : ' Ooty',
59. leaf : **true**
60. }, {
61. id : 900,
62. text : ' Chennai',
63. leaf : **true**
64. }, {
65. id : 1000,
66. text : ' Munnar',
67. leaf : **true**
68. } ]
69. } ]
70. }
71. });
73. Ext.create('PE.store.Albums');

我是用的 Go 语言为此项目写的 REST API 。完整可用的代码在这里 <https://github.com/ananddayalan/extjs-byexample-picture-explorer>

图片浏览器这个示例是一个非常简单并用来学习 tree panel 和 data view 使用是很合适的。也可以通过添加更多功能来改进这个例子。例如如何通过拖拽将图片从一个相册移动到另一个相册中。 我会留给你作为一个编码的练习，但在这里，我给你简要的概述一下拖拽功能，这将帮助你在此项目中添加拖拽功能。

拖拽

任意元素或组件都能支持拖拽。使用拖拽有三个重要的事情：

* 配置 item 为可拖拽的
* 创建放置目标
* 完成放置目标

配置 item 为可拖拽的

想要拖拽一个 item ，你需要为每一个元素创建 Ext.dd.DD 实例。

查看下列代码，通过创建 Ext.dd.DD 让所有使用 pics 类的 div 元素成为可拖拽的：

1. // Configure the pics as draggable var pics = Ext.get('pics').select('div');
2. Ext.each(pics.elements, **function**(el) {
3. **var** dd = Ext.create('Ext.dd.DD', el, ' picsDDGroup', {
4. isTarget : **false**
5. });
6. });

创建放置目标

使用 Ext.dd.DDTarget 创建放置容器。以下代码为所有的使用 album 类的 div 元素创建放置目标：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894094) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894094)

1. **var** albums = Ext.get('album').select('div');
2. Ext.each(albums.elements, **function**(el) {
3. **var** albumDDTarget = Ext.create('Ext.dd.DDTarget', el,
4. 'picsDDGroup');
5. });

完成放置目标

当一个可拖拽项放置到一个放置容器，我们需要从这个 item 的源位置将它移动到目标位置。这通过覆盖 DD 的 onDragDrop 方法来实现。看一看下列代码：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894094) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894094)

1. **var** overrides = {
2. onDragDrop : **function**(evtObj, targetElId) {
3. **var** dropEl = Ext.get(targetElId);
4. **if** (**this**.el.dom.parentNode.id != targetElId) {
5. dropEl.appendChild(**this**.el);
6. **this**.onDragOut(evtObj, targetElId);
7. **this**.el.dom.style.position = '';
8. **this**.el.dom.style.top = '';
9. **this**.el.dom.style.left = '';
10. } **else** {
11. **this**.onInvalidDrop();
12. }
13. },
14. onInvalidDrop : **function**() {
15. **this**.invalidDrop = **true**;
16. }
17. };

因为 DD 元素已经是实例了，重写的方法需要应用 Ext.apply(dd, overrides) ，如以下代码所示：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894094) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894094)

1. **var** albums = Ext.get('album').select('div');
2. **var** pics = Ext.get('pics').select('div');
3. Ext.each(pics.elements, **function**(el) {
4. **var** dd = Ext.create('Ext.dd.DD', el, ' picsDDGroup', {
5. isTarget : **false**
6. });
7. Ext.apply(dd, overrides);
8. });

总结

在本章中，你学习到如何使用拖拽功能。我们也看了几个高级组件：tree panel 和 data view。最后结合所学创建了一个示例项目。

# 第7章 - 图表(chart)

本章中将探索在 ExtJS 中使用不同类型的图表并使用一个名为费用分析的示例项目结束本章所学。以下是将要所学的内容：

* 图表类型
* 条形图 和 柱形图 图表
* 区域 和 折线 图表
* 饼图 图表
* 3 D 图表
* 费用分析 – 示例项目

图表

在第一章中提过，我说 ExtJS 是一站式的几乎能满足你对所有的 JavaScript 框架的需求。这当然还包括了图表功能。

图表类型

有三种类型的图表：笛卡尔图表，极坐标图表, 和 空间图表。

笛卡尔图表

Ext.chart.CartesianChart (xtype: cartesian or chart)

一个笛卡尔图表具有两个方向：X 和 Y 。默认 X 是水平的，Y 是垂直的。使用笛卡尔坐标图表有柱形图，条形图，区域，折线和散射。

极坐标图表

Ext.chart.PolarChart (xtype: polar)

这个图表有两个轴：角向和径向。图表的标绘值用极坐标来表达，有饼图和雷达图。

空间图表

Ext.chart.SpaceFillingChart (xtype: spacefilling)

这个图表填充图表的面积。

条形图和柱状图

使用条形图和柱状图，你至少需要提供 store ，axes ，series。

基本的柱状图

首先，我们创建一个 store 使用硬编码的内置数据，如以下代码所示：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('MyApp.model.Population', {
2. extend : 'Ext.data.Model',
3. fields : [ 'year', 'population' ]
4. });
6. Ext.define('MyApp.store.Population', {
7. extend : 'Ext.data.Store',
8. storeId : 'population',
9. model : 'MyApp.model.Population',
10. data : [{
11. "year" : "1610",
12. "population" : 350
13. },{
14. "year" : "1650",
15. "population" : 50368
16. },{
17. "year" : "1700",
18. "population" : 250888
19. },{
20. "year" : "1750",
21. "population" : 1170760
22. },{
23. "year" : "1800",
24. "population" : 5308483
25. },{
26. "year" : "1900",
27. "population" : 76212168
28. },{
29. "year" : "1950",
30. "population" : 151325798
31. },{
32. "year" : "2000",
33. "population" : 281421906
34. },{
35. "year" : "2010",
36. "population" : 308745538
37. }]
38. });
39. **var** store = Ext.create("MyApp.store.Population");

使用 Ext.chart.CartesianChart (xtype: cartesian 或 chart ) 创建图表并应用上面所创建的 store 。

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.create('Ext.Container', {
2. renderTo : Ext.getBody(),
3. width : 500,
4. height : 500,
5. layout : 'fit',
6. items : [ {
7. xtype : 'chart',
8. insetPadding : {
9. top : 60,
10. bottom : 20,
11. left : 20,
12. right : 40
13. },
14. store : store,
15. axes : [ {
16. type : 'numeric',
17. position : 'left',
18. grid : **true**,
19. title : {
20. text : 'Population in Millions',
21. fontSize : 16
22. }
23. }, {
24. type : 'category',
25. title : {
26. text : 'Year',
27. fontSize : 16
28. },
29. position : 'bottom'
30. }],
31. series : [ {
32. type : 'bar',
33. xField : 'year',
34. yField : [ 'population' ]
35. } ],
36. sprites : {
37. type : 'text',
38. text : 'United States Population',
39. font : '25px Helvetica',
40. width : 120,
41. height : 35,
42. x : 100,
43. y : 40
44. }
45. }]
46. });

前面代码中比较重要的是 axes ，series ，和 sprite 。axes 有三种类型：numeric ，time ，和 category 。

而在 series 中，你可以看到设置类型为 bar 。在 ExtJS 中，将会呈现为柱状图或条形图， 虽然你指定了类型为 bar ，但是默认是作为柱状图展示的，如果你想要条形图，需要在图表配置中设置 flipXY  为 true。

这里 sprites 的配置相当简单。sprites 是一个可选项，不是必须的。sprites 这里使用的 text 的类型，就是画出来一个图形，设置字体 25 号，宽度 120，高度 35，并且 x 的位置为 100 ，y 的位置是 40 ，sprites 是ExtJS 中画图的一个对象，这里我们只是用来写了一行字。sprites 也可以接受一个数组，但是这里我们只用一个设置。

这个 insetPadding 属性是用于指定图表的 padding 。

以下截图为输出结果：

条形图

如之前所说，为了得到条形图，你可以使用同样的代码，但要指定 flipXY 为 true 并改变相应坐标的位置，下面把原来 year 换到了左边。如以下代码所示：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.create('Ext.Container', {
2. renderTo : Ext.getBody(),
3. width : 500,
4. height : 500,
5. layout : 'fit',
6. items : [ {
7. xtype : 'chart',
8. flipXY : **true**,
9. insetPadding : {
10. top : 60,
11. bottom : 20,
12. left : 20,
13. right : 40
14. },
15. store : store,
16. axes : [ {
17. type : 'numeric',
18. position : 'bottom',
19. grid : **true**,
20. title : {
21. text : 'Population in Millions',
22. fontSize : 16
23. }
24. }, {
25. type : 'category',
26. title : {
27. text : 'Year',
28. fontSize : 16
29. },
30. position : 'left'
31. }],
32. series : [ {
33. type : 'bar',
34. xField : 'year',
35. yField : [ 'population' ]
36. }],
37. sprites : {
38. type : 'text',
39. text : 'United States Population',
40. font : '25px Helvetica',
41. width : 120,
42. height : 35,
43. x : 100,
44. y : 40
45. }
46. }]
47. });

下列截图为以上的代码的输出结果：

堆叠条形图

现在假设你想在柱状图的每一个分类上标绘两个值。 你可以将他们堆叠起来或者在每个分类上使用两条图形。

我们更新柱状图例子来展示一个堆叠图表。为此我们需要在 store 中额外添加一个数值字段，同时在 series 中我们需要为 yField 指定两个字段。你也可以堆叠超过两个字段，但是本例中我们只用两个字段，看看下面的代码：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('MyApp.model.Population', {
2. extend : 'Ext.data.Model',
3. fields : [ 'year', 'total', 'slaves' ]
4. });
6. Ext.define('MyApp.store.Population', {
7. extend : 'Ext.data.Store',
8. storeId : 'population',
9. model : 'MyApp.model.Population',
10. data : [{
11. "year" : "1790",
12. "total" : 3.9,
13. "slaves" : 0.7
14. },{
15. "year" : "1800",
16. "total" : 5.3,
17. "slaves" : 0.9
18. },{
19. "year" : "1810",
20. "total" : 7.2,
21. "slaves" : 1.2
22. },{
23. "year" : "1820",
24. "total" : 9.6,
25. "slaves" : 1.5
26. },{
27. "year" : "1830",
28. "total" : 12.9,
29. "slaves" : 2
30. },{
31. "year" : "1840",
32. "total" : 17,
33. "slaves" : 2.5
34. },{
35. "year" : "1850",
36. "total" : 23.2,
37. "slaves" : 3.2
38. },{
39. "year" : "1860",
40. "total" : 31.4,
41. "slaves" : 4
42. }]
43. });
44. **var** store = Ext.create("MyApp.store.Population");
46. Ext.create('Ext.Container', {
47. renderTo : Ext.getBody(),
48. width : 500,
49. height : 500,
50. layout : 'fit',
51. items : [ {
52. xtype : 'cartesian',
53. store : store,
54. insetPadding : {
55. top : 60,
56. bottom : 20,
57. left : 20,
58. right : 40
59. },
60. axes : [ {
61. type : 'numeric',
62. position : 'left',
63. grid : **true**,
64. title : {
65. text : 'Population in Millions',
66. fontSize : 16
67. }
68. }, {
69. type : 'category',
70. title : {
71. text : 'Year',
72. fontSize : 16
73. },
74. position : 'bottom'
75. }],
76. series : [ {
77. type : 'bar',
78. xField : 'year',
79. yField : [ 'total', 'slaves' ]
80. }],
81. sprites : {
82. type : 'text',
83. text : 'United States Slaves Distribution 1790 to 1860',
84. font : '20px Helvetica',
85. width : 120,
86. height : 35,
87. x : 60,
88. y : 40
89. }
90. }]
91. });

堆叠柱状图的输出如下列截图：

如果你想呈现多个字段，并且让它们未叠加，那么只需要简单的设置 series 的 stacked 属性为 false 即可，请看以下输出：

在图表中还有很多可用的选项。让我们瞧一瞧一些常用的选项：

* tooltip: 这个属性可以在 series 里添加一个提示信息，鼠标放在每个柱状图上是会提示相应的信息。
* legend: 这可以呈现图表的说明到任意的四个边上
* sprites: 这个可以接收一个数组对象，上面介绍过啦，它是可以在你图标上添加一些图形，文本等等。

下面这是使用的相同的 store ，只是用了一些上面提的其他的选项：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.create('Ext.Container', {
2. renderTo: Ext.getBody(),
3. width: 500,
4. height: 500,
5. layout: 'fit',
6. items: [{
7. xtype: 'chart',
8. legend: { docked: 'bottom' },
9. insetPadding: {
10. top: 60,
11. bottom: 20,
12. left: 20,
13. right: 40
14. },
15. store: store,
16. axes: [{
17. type: 'numeric',
18. position: 'left',
19. grid: **true**,
20. title: {
21. text: 'Population in Millions',
22. fontSize: 16
23. },
24. minimum: 0,
25. }, {
26. type: 'category',
27. title: {
28. text: 'Year',
29. fontSize: 16
30. },
31. position: 'bottom',
32. }],
33. series: [{
34. type: 'bar',
35. xField: 'year',
36. stacked: **false**,
37. title: ['Total', 'Slaves'],
38. yField: ['total', 'slaves'],
39. tooltip: {
40. trackMouse: **true**,
41. style: 'background: #fff',
42. renderer: **function** (storeItem, item) {
43. **this**.setHtml('In ' + storeItem.get('year') + ' ' + item. field + ' population was ' + storeItem.get(item.field) + ' m');
44. }
45. }
46. }],
47. sprites: [{
48. type: 'text',
49. text: 'United States Slaves Distribution 1790 to 1860',
50. font: '20px Helvetica',
51. width: 120,
52. height: 35,
53. x: 60,
54. y: 40
55. },{
56. type: 'text',
57. text: 'Source: http://www.wikipedia.org',
58. fontSize: 10,
59. x: 12,
60. y: 440
61. }]
62. }]
63. });

输出你看到页脚，提示信息和说明信息都在下图：

3D 柱状图

如果你改变 series 的类型为 bar3d ，就能获得 3D 的柱状图，如下列截图所示：

区域和折线图

区域和折线图也属于是笛卡尔图表。

区域图表

呈现一个区域图，使用下列代码简单的替换前面的例子里的 series ：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. series: [ {
2. type : 'area',
3. xField : 'year',
4. stacked : **false**,
5. title : [ 'Total', 'slaves' ],
6. yField : [ 'total', 'slaves' ],
7. style : {
8. stroke : "#94ae0a",
9. fillOpacity : 0.6,
10. }
11. } ]

以上代码所示的输出：

类似于堆叠柱状图，你也可以在 series 中设置 stacked 为true 来实现堆叠。如果你在之前例子上将 stacked 改为 true ，那么将获得下列结果输出：

折线图

使用下列的配置在上面的例子中的 series ，折线图显示图下图：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. series: [ {
2. type : 'line',
3. xField : 'year',
4. title : [ 'Total' ],
5. yField : [ 'total' ]
6. },{
7. type : 'line',
8. xField : 'year',
9. title : [ 'Slaves' ],
10. yField : [ 'slaves' ]
11. }]

饼图

这是一个在很多应用中都很常用的图表和报表工具。呈现一个饼图使用 Ext.chart.PolarChart (xtype: polar) 。

基本的饼图

指定类型为 pie ，还要指定 angleField  和 label 来呈现饼图。angleField  这是角度字段，如以下代码所示：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('MyApp.store.Expense', {
2. extend : 'Ext.data.Store',
3. alias : 'store.expense',
4. fields : [ 'cat', 'spent' ],
5. data : [ {
6. "cat" : "Restaurant",
7. "spent" : 100
8. }, {
9. "cat" : "Travel",
10. "spent" : 150
11. }, {
12. "cat" : "Insurance",
13. "spent" : 500
14. }, {
15. "cat" : "Rent",
16. "spent" : 1000
17. }, {
18. "cat" : "Groceries",
19. "spent" : 400
20. }, {
21. "cat" : "Utilities",
22. "spent" : 300
23. } ]
24. });
26. **var** store = Ext.create("MyApp.store.Expense");
28. Ext.create('Ext.Container',{
29. renderTo : Ext.getBody(),
30. width : 600,
31. height : 500,
32. layout : 'fit',
33. items : [ {
34. xtype : 'polar',
35. legend : {
36. docked : 'bottom'
37. },
38. insetPadding : {
39. top : 100,
40. bottom : 20,
41. left : 20,
42. right : 40
43. },
44. store : store,
45. series : [ {
46. type : 'pie',
47. angleField : 'spent',
48. label : {
49. field : 'cat',
50. },
51. tooltip : {
52. trackMouse : **true**,
53. renderer : **function**(storeItem, item) {
54. **var** value = ((parseFloat(storeItem.get('spent')/ storeItem.store.sum('spent')) \* 100.0).toFixed(2));
55. **this**.setHtml(storeItem.get('cat') + ': ' + value + '%');
56. }
57. }
58. } ]
59. } ]
60. });

圆环饼图

这仅需要在之前的例子中设置 donut 属性的值为 40 ，你将获取下列图表。donut 这个值是半径的百分比：

3D饼图

在 ExtJS 6 中，对 3D 饼图做出了一些改进。现在 3D 饼图支持 label 和可配置的 3D 切面，例如厚度，变形等等。

我们使用同样的 model 和 store 使用前面饼图的例子创建一个 3D 饼图，如下：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.create('Ext.Container',{
2. renderTo : Ext.getBody(),
3. width : 600,
4. height : 500,
5. layout : 'fit',
6. items : [ {
7. xtype : 'polar',
8. legend : {
9. docked : 'bottom'
10. },
11. insetPadding : {
12. top : 100,
13. bottom : 20,
14. left : 80,
15. right : 80
16. },
17. store : store,
18. series : [ {
19. type : 'pie3d',
20. donut : 50,
21. thickness : 70,
22. distortion : 0.5,
23. angleField : 'spent',
24. label : {
25. field : 'cat'
26. },
27. tooltip : {
28. trackMouse : **true**,
29. renderer : **function**(storeItem, item) {
30. **var** value = ((parseFloat(storeItem.get('spent')/ storeItem.store.sum('spent')) \* 100.0).toFixed(2));
31. **this**.setHtml(storeItem.get('cat') + ': ' + value+ '%');
32. }
33. }
34. } ]
35. } ]
36. });

下面的图片显示了上面代码的输出：

费用分析器 – 示例项目

又是项目时间，现在你已经了解了 ExtJS 中不同的图表类型，我们来创建一个示例项目名为 费用分析器。下面是最终设计效果：

我们使用 Sencha Cmd 来构建应用。运行下列命令：

**[plain]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. sencha -sdk <path to SDK>/ext-6.0.0.415/ generate app EA ./expenseanalyzer

之后我们移除所有不必要的文件和代码，添加一些额外的文件。目录结构如下：

* 这里只有一些重要文件的代码，完整的代码在这里 https://github. com/ananddayalan/extjs-by-example-expense-analyzer

下列代码是创建 grid 。这个 List 视图继承自 Ext.grid.Panel ，数据使用 expense store ，它有三列：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('EA.view.main.List', {
2. extend : 'Ext.grid.Panel',
3. xtype : 'mainlist',
4. maxHeight : 400,
5. requires : [ 'EA.store.Expense' ],
6. title : 'Year to date expense by category',
7. store : {
8. type : 'expense'
9. },
10. columns : {
11. defaults : {
12. flex : 1
13. },
14. items : [ {
15. text : 'Category',
16. dataIndex : 'cat'
17. }, {
18. formatter : "date('F')",
19. text : 'Month',
20. dataIndex : 'date'
21. }, {
22. text : 'Spent',
23. dataIndex : 'spent'
24. } ]
25. }
26. });

我并没有在这里使用分页。maxHeight 是用于限制 grid 的高度，同时开启滚动条，因为有更多的数据。

以下代码创建了 expense store 。这个 store 使用了内嵌数据。这里我们并没有单独为 store 创建 model ：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('EA.store.Expense', {
2. extend : 'Ext.data.Store',
3. alias : 'store.expense',
4. storeId : 'expense',
5. fields : [ {
6. name : 'date',
7. type : 'date'
8. },
9. 'cat',
10. 'spent'
11. ],
12. data : {
13. items : [{
14. "date" : "1/1/2015",
15. "cat" : "Restaurant",
16. "spent" : 100
17. },{
18. "date" : "1/1/2015",
19. "cat" : "Travel",
20. "spent" : 22
21. },{
22. "date" : "1/1/2015",
23. "cat" : "Insurance",
24. "spent" : 343
25. }]
26. },
27. proxy : {
28. type : 'memory',
29. reader : {
30. type : 'json',
31. rootProperty : 'items'
32. }
33. }
34. });

继续创建柱状图。在柱状图中我们将使用另外一个 store 叫做 expensebyMonth store ，我们将从 expense 的数据填充进来。

下列 3D 柱状图有两个 axis 类型： numeric 和 category 。我们使用日期字段的月部分作为 category 。renderer 属性用于呈现日期字段的月份部分：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('EA.view.main.Bar', {
2. extend : 'Ext.chart.CartesianChart',
3. requires : [ 'Ext.chart.axis.Category',
4. 'Ext.chart.series.Bar3D',
5. 'Ext.chart.axis.Numeric',
6. 'Ext.chart.interactions.ItemHighlight' ],
7. xtype : 'mainbar',
8. height : 500,
9. padding : {
10. top : 50,
11. bottom : 20,
12. left : 100,
13. right : 100
14. },
15. legend : {
16. docked : 'bottom'
17. },
18. insetPadding : {
19. top : 100,
20. bottom : 20,
21. left : 20,
22. right : 40
23. },
24. store : {
25. type : 'expensebyMonthStore'
26. },
27. axes : [ {
28. type : 'numeric',
29. position : 'left',
30. grid : **true**,
31. minimum : 0,
32. title : {
33. text : 'Spendings in $',
34. fontSize : 16
35. }
36. }, {
37. type : 'category',
38. position : 'bottom',
39. title : {
40. text : 'Month',
41. fontSize : 16
42. },
43. label : {
44. font : 'bold Arial',
45. rotate : {
46. degrees : 300
47. }
48. },
49. renderer : **function**(date) {
50. **return** [ "Jan", "Feb", "Mar", "Apr", "May" ][date.getMonth()];
51. }
52. } ],
53. series : [ {
54. type : 'bar3d',
55. xField : 'date',
56. stacked : **false**,
57. title : [ 'Total' ],
58. yField : [ 'total' ]
59. } ],
60. sprites : [ {
61. type : 'text',
62. text : 'Expense by Month',
63. font : '20px Helvetica',
64. width : 120,
65. height : 35,
66. x : 60,
67. y : 40
68. } ]
69. });

现在为上面的柱状图创建 store 。model 为 MyApp.model.ExpensebyMonth 。这个 store 将用来显示每个月的花费总数。数据是通过对 expense store 的日期字段进行分组后填充的，现在我们瞧一瞧 data 属性是如何配置填充数据的：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('MyApp.model.ExpensebyMonth', {
2. extend : 'Ext.data.Model',
3. fields : [ {
4. name : 'date',
5. type : 'date'
6. }, 'total' ]
7. });
9. Ext.define('MyApp.store.ExpensebyMonth', {
10. extend : 'Ext.data.Store',
11. alias : 'store.expensebyMonthStore',
12. model : 'MyApp.model.ExpensebyMonth',
13. data : (**function**() {
14. **var** data = [];
15. **var** expense = Ext.createByAlias('store.expense');
16. expense.group('date');
17. **var** groups = expense.getGroups();
18. groups.each(**function**(group) {
19. data.push({
20. date : group.config.groupKey,
21. total :group.sum('spent')
22. });
23. });
24. **return** data;
25. })()
26. });

以下代码用于生成饼图。这个图表使用的的 store 是 expense ，但只显示了一次选择一个月的数据。在主视图上添加了一个下拉框用于选择月份。

这个 beforerender 事件是用于过滤 expense store 里用于显示的数据，只加载一月份的数据：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('EA.view.main.Pie', {
2. extend : 'Ext.chart.PolarChart',
3. requires : [ 'Ext.chart.series.Pie3D' ],
4. xtype : 'mainpie',
5. height : 800,
6. legend : {
7. docked : 'bottom'
8. },
9. insetPadding : {
10. top : 100,
11. bottom : 20,
12. left : 80,
13. right : 80
14. },
15. listeners : {
16. beforerender : **function**() {
17. **var** dateFiter = **new** Ext.util.Filter({
18. filterFn : **function**(item) {
19. **return** item.data.date.getMonth() == 0;
20. }
21. });
22. Ext.getStore('expense').addFilter(dateFiter);
23. }
24. },
25. store : {
26. type : 'expense'
27. },
28. series : [ {
29. type : 'pie3d',
30. donut : 50,
31. thickness : 70,
32. distortion : 0.5,
33. angleField : 'spent',
34. label : {
35. field : 'cat',
36. }
37. } ]
38. });

截止目前，我们创建好了 grid ，柱状图，饼图，和这个应用所需要的 store 。现在需要在主视图上把他们联系起来。以下代码展示了 main 视图的经典工具包里的片段。main 视图是一个选项卡控件，为每个选项卡指定视图：

**[javascript]** [view plain](http://blog.csdn.net/lovelyelfpop/article/details/53894103) [copy](http://blog.csdn.net/lovelyelfpop/article/details/53894103)

1. Ext.define('EA.view.main.Main', {
2. extend : 'Ext.tab.Panel',
3. xtype : 'app-main',
4. requires : [
5. 'Ext.plugin.Viewport',
6. 'Ext.window.MessageBox',
7. 'EA.view.main.MainController',
8. 'EA.view.main.List',
9. 'EA.view.main.Bar',
10. 'EA.view.main.Pie'
11. ],
12. controller : 'main',
13. autoScroll : **true**,
14. ui : 'navigation',
15. // Truncated code
16. items : [ {
17. title : 'Year to Date',
18. iconCls : 'fa-bar-chart',
19. items : [ {
20. html : '<h3>Your average expense per month is: ' +Ext.createByAlias('store.expensebyMonthStore').average('total') +'</h3>',
21. height : 70,
22. },{
23. xtype : 'mainlist'
24. },{
25. xtype : 'mainbar'
26. }]
27. },{
28. title : 'By Month',
29. iconCls : 'fa-pie-chart',
30. items : [ {
31. xtype : 'combo',
32. value : 'Jan',
33. fieldLabel : 'Select Month',
34. store : [ 'Jan', 'Feb', 'Mar', 'Apr', 'May' ],
35. listeners : {
36. select : 'onMonthSelect'
37. }
38. }, {
39. xtype : 'mainpie'
40. } ]
41. } ]
42. });

总结

在本章中，我们在 ExtJS 中学习到不同的图表。并且创建了一个示例项目来演示他们的应用。

# 第8章 - 主题和响应式设计

本章重点在 ExtJS 应用的主题和响应式设计。主要有以下几点内容：

* SASS 介绍和入门
* 主题
* 响应式设计

SASS 介绍和入门

SASS (Syntactically Awesome Stylesheets) 是一个样式语言，当你使用 SASS 代替 CSS 来写样式，之后 SASS 将使用编译器编译成 CSS ，SASS 有更好的语法和一组特性使我们更容易编写样式。同时维护 SASS 代码比维护 CS 更容易。相比直接写 CSS 你将写更少的 SASS 代码。

通常在你构建你的应用时，编译器将会编译 SASS 为适用于浏览器的 CSS 文件。

关于 SASS 的话题超出了本书的范围。你不必知道的太详细，但是一些基本知识还是必要的。所以这里我们学习 SASS 的最低限度要求的知识。

你不需要另外的安装 SASS 编译器，sencha Cmd 6 已经有了，它使用了新的 SASS 编译器，叫做 fashion ，你安装好 sencha Cmd 6 ，它也就已经安装好了。

* 如果你使用的是 Ext JS SDK 5 ，那么你需要安装 ruby 来编译 SASS。

SASS 有两个语法；默认名为 SCSS 的新语法和旧的 SASS 语法。这里我们自然使用最新的。接下来你将了解 SASS 的基本知识，例如变量，混合，嵌套。

变量

CSS 不支持变量。这一点 SASS 可以，SASS 变量可以存储例如颜色，字体或者任意的你想要重用的 CSS 值。看一看下列代码：

1. $body-**background-color**: transparent;
2. $base-**color**: #808080;
3. $**font-family**: helvetica , arial , verdana , sans-serif;
4. body {
5. **background**: $body-background-color;
6. **font**: 100% $font-family;
7. **color**: $base-color;
8. }

上面的代码在处理后，将输出为正常的 CSS ，如以下代码所示。之前定义的变量可以在多个地方复用：

1. body {
2. **background**: transparent;
3. **font**: 100% helvetica , arial , verdana , sans-serif;
4. **color**: #808080;
5. }

混合

这就像是 宏 — 由一条指令扩展到多条指令。看下列代码：

1. @mixin border-radius($radius) {
2. -moz-border-radius: $radius;
3. -webkit-border-radius: $radius;
4. -ms-border-radius: $radius;
5. border-radius: $radius;
6. }
8. a.button {
9. **background**: black;
10. **color**: white;
11. **padding**: 10px 20px;
12. @include border-radius(10px);
13. }

上面的 SCSS 代码输出后就是下列的代码：

1. a.button {
2. **background**: black;
3. **color**: white;
4. **padding**: 10px 20px;
5. -moz-border-radius: 10px;
6. -webkit-border-radius: 10px;
7. -ms-border-radius: 10px;
8. border-radius: 10px;
9. }

嵌套

不同于 HTML ，CSS 不支持 CSS 选择器的嵌套。SASS 允许你这么做。

使用 CSS 你要这样写代码：

1. .table {
2. **background**: blue;
3. }
5. .table .col {
6. **padding**: 10px;
7. }
9. .table .row .col {
10. **color**: red;
11. }

在 SASS 中，你可以这样写，这有点像链式编程：

1. .table {
2. **background**: blue;
3. .col {
4. **padding**: 10px;
6. .row {
7. **color**: white;
8. }
9. }
10. }

 主题

Ext JS 制作主题是很容易且便于维护的，能让你很轻松的从现有主题进行扩展。

首先需要建立一个工作空间，运行下列命令在当前目录的 myapp 文件夹里生成应用：

1. sencha -sdk <path to SDK> generate app MyApp ./myapp

Ext JS 提供了一组主题，你可以选择其中之一或者通过扩展 sencha 提供的主题来创建你自己的主题。

默认 sencha Cmd 6 使用 theme-neptune 主题，你可以在之前生成的 MyApp 应用中的 app.json 文件里找到以下代码。在这里你可选择切换现代和经典工具包所使用的主题：

1. "builds": {
2. "classic": {
3. "toolkit": "classic",
4. "theme": "theme-triton"
5. },
6. "modern": {
7. "toolkit": "modern",
8. "theme": "theme-neptune"
9. }
10. }

* 如果你使用是 Ext JS 5 ，文件里的代码就不是上面那样，而是如下所示：

1. "theme": "ext-theme-neptune"

* 在 Ext JS 6 以前，所有的主题的名称前缀都是 ext- ，在 Ext JS 6 中这样的前缀被移除了。

如果你想切换到其他的内置主题，只需要修改 theme 为你想要的主题的名称就行了。尝试一下，刷新页面看看有何不同。

 创建自定义主题

首先我们先来运行 MyApp 应用，执行下列命令：

1. sencha app watch

应用将会运行在 http://localhost:1841 (默认地址).

下列截图是使用默认的主题 theme-neptune 的输出结果：

我们在 MyApp 应用中定制并生成自己的主题。

第一步在应用的目录下，运行下列命令生成自定义主题：

1. sencha generate theme my-theme

如果你现在查看应用文件夹下的 packages 目录，你会发现已经生成了 my-theme 下列是文件夹里的内容：

我们来看一下这里面的一些文件和文件夹的用途：

* package.json: 这个文件里有所有的配置和包的属性
* sass/var: 这个目录中包含所有的 SASS 变量
* sass/src: 这里是所有的 SASS 规则和混合
* sass/etc: 这里包含一些通用的函数和混合
* resources: 主题所使用的图片和其他的静态资源
* overrides: 这个目录包含了使用 JavaScript 复写的主题（例如你想覆盖 panel 的主题，就在这里写）

 这里，你的组件的样式文件和文件路径，应在 sass/var/ ， sass/src ，和 overrides 文件夹结构中与你的组件包路径是匹配的。例如，如果要使用变量改变 Ext.panel.Panel 的样式，那么就应该在 sass/var/ 目录下创建 panel 文件夹，并在 panel 目录里建立名为 Panel.scss 的文件。将变量写在这个文件里。

下一步，我们要决定需要扩展哪些主题。Ext JS 6 有两套主题：一套经典的，一套现代的。这些主题如下图所示。除了 Base 和 Neutral 你可以基于任意主题扩展：

当你生成一个自定义主题，默认它是使用 theme-classic 。现在切换为 theme-crisp 。更新 packages/local/my-theme/package.json 文件中的

“extend”: “theme-classic”,

替换为

“extend”: “theme-crisp”,

* 生成主题的命令还可以接受一个可选的参数来选择基于何种主题进行扩展。如果你不指定参数，那么它将继承自 theme-classic 。

接下来，完成下列的几项：

* 配置 SCSS 变量
* 创建 SCSS 规则和混合
* 覆盖图片
* 覆盖 JavaScript 样式

SASS 变量

Ext JS 已经定义了一大堆的 SASS 变量，你可以根据你的需求定制覆盖他们。现在我们来更新一些全局的变量。在 var 目录下，创建一个文件名为 component.scss 并添加下列代码：

1. $base-**color**: #F17C26 !default;
3. $**color**: #404040 !default;
5. $**font-size**: 15px !default;

这个 !default 是必须的，作用是允许它被覆盖。这里我找个一个图解释了为什么需要加 !default

现在在浏览器刷新页面，将会呈现下面的应用：

 使用 SASS 混合来新建一个组件 UI

你应该做的第一件事是尝试你是否能够通过只用变量来做到。在大多数情况下，变量都能帮助做到自定义 UI 。有时，这可能是不够的。只有在这种情况下，可以考虑创建组件UI和混入。

在 Ext JS 中大部分组件都定义了 SASS 混合；你可以调用这些混合来生成新的组件 UI 。大多数组件在 Ext JS 中都有组件的 UI ，它们默认情况下都使用一个默认的 UI ，你可以创建自己的组件的 UI 并且使用它们。

例如，我们创建了一个自定义的 panel UI 。在目录 my-theme/sass/src/panel/ 下创建一个名为 Panel.scss 的文件并键入下列代码：

1. @include extjs-panel-ui(
3. $ui: 'dark',
5. $ui-header-**background-color**: #404040,
7. $ui-**border-color**: #404040,
9. $ui-body-**background-color**: #404040,
11. $ui-body-**border-color**: #404040
13. ) ;

现在你可以使用这个 UI 了，如以下代码所示：

1. Ext.define('MyApp.view.main.MyPanel', {
2. extend : 'Ext.panel.Panel',
3. xtype : 'my-panel',
4. ui : 'dark',
5. });

@include 这是用来调用声明的混合的，设置的 ui 就是一个参数，然后在组件中应用了 ui 属性之后，组件在渲染时就会以 ui 属性作为组件 body 内 dom 元素的 id ，而上面应用的样式对 ui 的定义也会相应生效。

 覆盖 JavaScript 样式

极少数情况下，你可能要改变外观，只能在JS来完成。在这种情况下，你可以使用 JS 覆盖。例如在 my-theme\overrides\panel\ 目录下创建名为 Panel.js  的文件，然后键入以下代码，刷新应用查看有何不同：

1. Ext.define('my-theme.panel.Panel', {
2. override : 'Ext.panel.Panel',
3. titleAlign : 'center',
4. padding : 20
5. });

以上代码的输出如下，你会发现右边 grid 的 padding 距离明显增大了好多：

图片

在你的主题里你也可以很容易的自定义图标，把图片放在 my-theme\resources\images\ 目录下即可，图片必须和所替换的应用相同的文件名。

设计你的应用的样式

你学会了在主题中为你的应用的特定需求设计样式，如果你想明确的为 classic/src/view/main/Main.js 设置一个变量，那么你需要添加一个 scss 文件，名为 classic/sass/src/view/main/Main.scss 并在文件中设置变量，你可以添加 SASS ，CSS 样式，以及你也可以添加混合，创建特定的组件 UI 。

* 任意的 CSS 都是有效的 SCSS ，所以 SCSS 文件中可以是纯 CSS 代码。

Sencha Cmd 已经为视图 classic/src/view/main/Main.js 生成了 classic/sass/src/view/main/Main.scss 文件。并且你会发现下列代码。同样，你可以在文件中为 Main.js 写任意的样式代码：

1. @include extjs-panel-ui(
2. $ui: 'navigation',
3. $ui-header-**color**: #fff,
4. $ui-header-glyph-**color**: #fff,
5. $ui-header-glyph-opacity: 1,
6. $ui-header-**font-size**: 20px,
7. $ui-header-**line-height**: 24px,
8. $ui-header-**font-weight**: bold,
9. $ui-header-icon-**height**: 24px,
10. $ui-header-icon-**width**: 24px,
11. $ui-header-icon-spacing: 15px,
12. $ui-header-**background-color**: $base-color,
13. $ui-header-**padding**: 0,
14. $ui-header-text-**margin**: 36px,
15. $ui-header-noborder-adjust: false
16. ) ;
18. .x-tab-icon-el-navigation {
19. **font-family**: FontAwesome;
20. **color**: #acacac;
21. .x-tab-over & {
22. **color**: #c4c4c4;
23. }
24. .x-tab-active & {
25. **color**: #fff;
26. }
27. }

将一个应用程序之间共享的样式应放置在该主题，而不是在这里。

如果你的样式，混合或变量没有指定到对应的视图，那么你可以放置到 /sass/etc/all.scss 。例如，如果你想覆盖基本的颜色，在 /sass/etc/all.scss 中写入下列代码：

1. $base-**color**: #404040;

 响应式设计

通过 Sencha Cmd 生成的应用默认是响应式的。你可以尝试调整应用的大小，你会发现左侧的 panel 移动到了顶部，显示如以下截图：

如果你查看 classic\src\view\main\Main.js ，将看到以下代码：

1. responsiveConfig: {
2. tall: {
3. headerPosition: 'top'
4. },
5. wide: {
6. headerPosition: 'left'
7. }
8. } ,
9. defaults: {
10. bodyPadding: 20,
11. tabConfig: {
12. plugins: 'responsive',
13. responsiveConfig: {
14. wide: {
15. iconAlign: 'left',
16. textAlign: 'left'
17. },
18. tall: {
19. iconAlign: 'top',
20. textAlign: 'center',
21. width: 120
22. }
23. }
24. }
25. }

responsiveConfig （响应式配置）

Ext JS 提供了 responsiveConfig 能动态的根据屏幕大小来进行响应。这是一个对象使用 key 表示条件，这些条件不满足时，某些配置将会被应用。

默认组件不支持响应式。为类定义或组件实例添加下列应用：

plugins: ‘responsive’

一下变量是可以用于  responsiveConfig 规则：

* landscape: 如果设备方向是横向此项设置为 true ，在桌面设备上此项永远为 true 。
* portrait: 如果设备方向是纵向此项设置为 true ，此项在桌面设备上永远为 true 。
* tall: 如果宽度小于高度时此项设置为 true ，不考虑设备的类型。
* wide: 如果宽度大于高度此项设置为 true ，不考虑设备的类型。
* width: 这表示浏览器窗口的宽度。
* height: 这表示浏览器窗口的高度。

这里你可以看到 iconAlign 和 textAlign 在 wide 和 tall 中的值是不同的，并且 tall 中仅设置了宽度，如果宽度小于高度，那么这个配置将会被应用；否则将会使用正常的配置：

1. tabConfig: {
2. plugins: 'responsive',
3. responsiveConfig: {
4. wide: {
5. iconAlign: 'left',
6. textAlign: 'left'
7. },
8. tall: {
9. iconAlign: 'top',
10. textAlign: 'center',
11. width: 120
12. }
13. }
14. }

这些规则也可以很复杂，例如下列代码，第一个规则检查两个条件，宽度小于 768 并且宽度小于高度时：

1. responsiveConfig: {
2. 'width < 768 && tall': {
3. visible: **true**
4. },
5. 'width >= 768': {
6. visible: **false**
7. }
8. }

同样的，你也可以使用 platformConfig ，如以下代码所示：

1. platformConfig: {
2. 'windows || desktop': {
3. visible: **true**
4. },
5. android && ios: {
6. visible: **false**
7. }
8. }

在上面的代码中，Ext. platformTags 配置了 Windows ，desktop ，Android ，iOS 。也包含一些其他的平台 phone ，Firefox ，Chrome ，Safari ，触摸，平板等等。

 总结

在本章中，你学习了SASS 的基础。我们也充分理解了如何在 Ext JS 中使用主题。同时我们还看了响应式设计相关的配置。