



第 2 章 Ext 起步

Ext 是一个工具库(Library), 还是一套框架(Framework)呢? “框架”字眼似乎有“专制”的味道——因为人们总想冲破各种条条框框。实际上使用了 Ext 就可以满足一般性的软件 GUI 开发要求, 因此按照作者 Jack 的原话, Ext 定位于“真正地整合好的和完整的程序开发平台”。

要全面掌握 Ext, 并非一时半刻可以完成, 必须经过一个阶段的学习。就让我们从最简单的对 Ext 的了解开始吧。



关键词

- Ext 初学者手把手入门
- Ext 及其 Core 的简介
- 兼容其他 JS 库
- 从哪开始学习 Ext?
- Ext 的重点是什么?



2.1 获取 Ext 的方法

由于 JavaScript 是一门解释型的语言而非编译型语言,因此可以说 Ext 程序便是 Ext JavaScript 的源码本身。获取 Ext 的具体方法如下面介绍的流程。

(1) 登录官方网站 <http://www.extjs.com>。

(2) 进入 download 页面,选择当前最新的版本 Stable Release 3.0 下载。如需要下载以往版本(1.x~2.x),可以在 http://extjs.com/learn/Ext_Version_Archives 位置找到。

(3) 下载并解压缩后,浏览它的文件夹,会发现如下的安排(以 Ext 3.2 为例):

Ext 根目录

```
|+-- adapter      (Ext 库底层的适配器)
|   |
|   +-- ext       (Ext 自家的适配器)
|   +-- jquery    (为结合 jQuery 库的适配器)
|   +-- prototype (为结合 Prototype.js 库的适配器)
|   +-- yui       (为结合 YUI Library 库的适配器)
|+-- pkgs        (各种常用的包, 体积更小)
|+-- docs        (Ext 文档中心, 浏览需要服务端的支持)
|+-- example     (100 多个自带的范例)
|+-- resources   (各种 CSS 样式表、图片等资源文件)
|+-- source      (全部的源代码)
|+-- welcome     (欢迎页面)
```

(4) 下载回来的 ZIP 文件已经包括文档中心了。除了这里有之外,官方还提供 AIR 版本的文档,内容和用法都相同,只是在 Adobe AIR 环境中运行而已,可独立执行。

这些代码有调试的版本和供发布时的产品版本。产品版本已经做压缩(消除空白符、硬回车和注释)和混淆的处理(所有局部变量重命名为短的名称,使用 YUI Compressor)。在开发阶段,您应使用的是-debug 版本,这样才会看到未混淆过的错误信息。

要引入 Ext 的开发版本,这样引入 JavaScript 文件就可以了:

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

要引入产品版本(打开 gzipped 后约 250KB 大小),就省略掉“-debug”:

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

2.2 Ext 运行环境

要保证您所部署的浏览器环境运行 Ext 正常,浏览器必须满足下列版本的要求:

- Windows Internet Explorer 6 及更高版本

- Mozilla Firefox 1.5 及更高版本(PC 和 Macintosh)
- Apple Safari 3 及更高版本
- Opera 9 及更高版本(PC 和 Macintosh)
- Google Chrome

另外, 支持 JavaScript 的移动设备的浏览器亦可支持 Ext, 如图 2.1 所示是诺基亚 E90 手机上的截图。

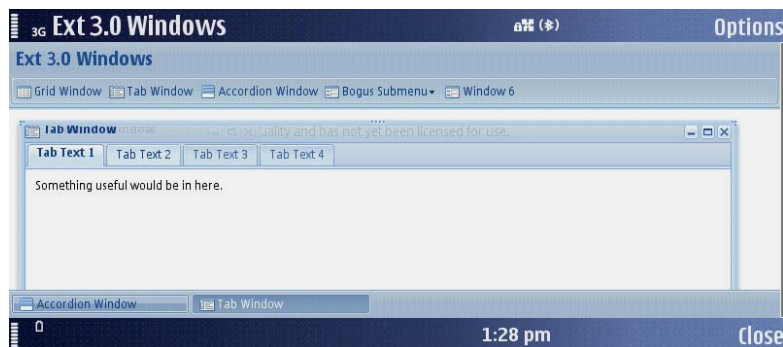


图 2.1 高端手机 Nokia E90 内嵌的浏览器支持 JavaScript 1.5

原来 Ext 也可以这样地用! ECMAScript 开拓了我们的眼界, 也真正地拓展了 Ext 或者是 Ajax 程序的应用。

2.3 如何兼容其他 JavaScript 库

Ext 库具备全面的 API, 尝试着为我们提供完整的整合开发平台, 但是如果结合遗留项目会存在着如何兼容其他 JavaScript 库的问题。为解决此问题, 它采用了适配器(Adapter)模式的策略, 适配器另外还具备以下功能:

- 早期 Ext 的开发阶段, 其底层代码是依附其他 JavaScript 库。
- Ext 的组件代码不是按其他 Ajax 库的风格写的。Adapter 会把其他 Ajax 库的代码转换成自己风格的代码。

如图 2.2 所示, 当前 Ext 为流行的 JavaScript 库提供了功能的转换, 包括 Prototype.js, jQuery 和 Yahoo UI! Library。

当最早期 Ext 还是作为 Yahoo UI! Library(简称 YUI)的 JavaScript 库的扩展的时候, 它依赖 YUI 的底层代码来处理跨浏览器的问题。现在 Ext 已经是独立、免依赖的库了(Standalone), 你可将 YUI Library 替换为另外你所选择的 JavaScript 库, 如 Prototype.js、jQuery 或者是这些之中的最佳选择, 就是 Ext 自带的底层库。负责将这些库映射为其底层库的这部分代码, 我们称之为适配器(Adapters)。这部分源码位于 source/adapter 的子目录中, 未压缩的版本各适配器位于/adapters/<库名称>/。如果你在项目中遇到了库与库之间命名冲突、版本管理等的情况, 就必须使用相应的适配器。在使用过程中, 应按照表 2.1 中的文件引入顺序来引



入文件。

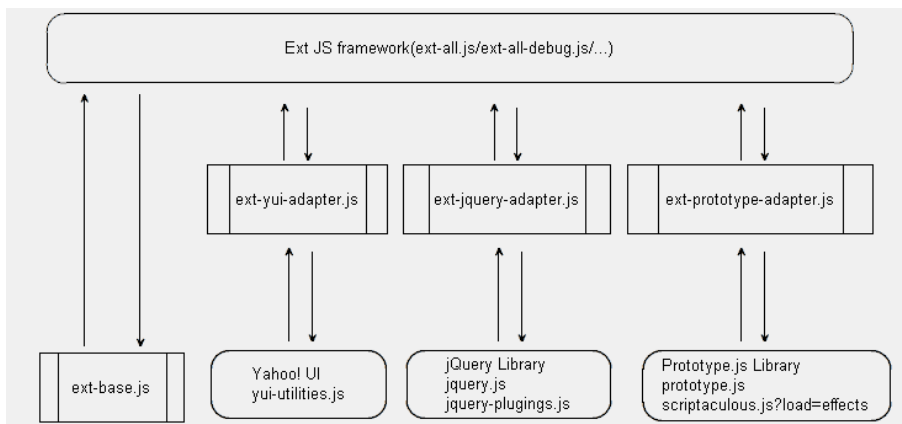


图 2.2 与各底层库的关系

表 2.1 适配器文件引入顺序

适 配 器	文件引入顺序
Ext Stand-alone(Ext 独立适配器)	ext-base.js ext-all.js
Yahoo! UI (0.12+)	yui-utilities.js ext-yui-adapter.js ext-all.js
jQuery (1.1+)	jquery.js ext-jquery-adapter.js ext-all.js
Prototype.js (1.5+) / Scriptaculous (1.7+)	prototype.js scriptaculous.js?load=effects(指定的特效) ext-prototype-adapter.js ext-all.js

值得一提的是，除了先前发布的 Ext JS/Ext GWT，官方产品还发布了精简版 Ext Core，它剔除了 Ext 框架中那些复杂的 UI 组件，把最常用的功能保留，例如 DOM 操作、Ajax/XHR、事件、动画、模板、OO 机制……这些 Web 开发中寻常遇到的任务，被裁剪出来，形成迷你版的 Core 发布。Core 基于 MIT 许可的开源协议，不同于 Ext JS 的 GPL v3，该 MIT 的协议自由度更大、更彻底。

之所以存在相对独立的 Ext Core 和 Ext JS/Ext GWT，此做法显然是让受众群体觉得其分工愈加明确。精简化之后一跃使 Ext 放在 jQuery/Prototype.js/Montools 同等的应用水平。有关 JavaScript 库“尖端/日常”的应用分工，我们还会在第 3 章 3.3 节的“元素小结”中进行详细的归纳与分析。

有趣的是,有别于 jQuery/Prototype.js/Mootools 先有 Core 库再发展其 UI 库,这是一般发展的顺序,Ext 则顺序颠倒,先有 UI 库后有 Core 库。不过还好 Ext 的架构中各单元布局合理,各层次明晰,裁剪一个精简版的 Core 出来不见得是一件困难的事情。

2.4 如何学好 Ext 框架

正如每一道锁都配有一把钥匙去开启一样,我们总谋求在学习某一领域的过程中找到关键的捷径,“不走弯路”是大多初学者的殷切诉求和诚挚盼望的事情,躲避“技术泡沫”也是老手们所切记的,然而能否做到这两点?实际上每个人因自己的情况差异而不同。

接触 Ext 至少有一段时间了。根据作者对用户平时感受的观察,Ext 与同类型框架 Dojo 或者有着多少相仿的际遇:虽然是开源的却语法晦涩;HTML/CSS 属基本功但大家还是不愿碰;JavaScript 也都在用,然而还是避之则吉……凡此等等的问题尽管不都是每个人都遇到,而且业务上也有明确的分工,但至少总有挥之不去的抵触情结困扰着自己的学习信心,难道真的没有方法能快速掌握这个框架吗?

非也。软件终究是人想出来怎么做的。——没有遵从某方面的经验法则,并且是成功的、千锤百炼的法则,何以能做出手中的干将莫邪神兵利器?凭心而论,探求业务精练之道乃是我辈之追求。可喜的是,Ext 不但没有背离当今主流的软件方法论,而且反复使用着这些理论、体系。从另外一个角度讲,它是务求与我们的知识储备相吻合的,而不是旁门左道式的“脚本编程(Hack it everywhere)”——广大遵守 OO 的开发者应有此体会!

可见,今日的 Ext 之所以备受开发人员的垂青,除优秀的前端框架乏善可陈外(包括 Flex 的也统计在内),也与它们所使用的技术底子有很大的关系。拆开 Ext 的表面,究其筋骨处处渗透着先进思想的影响。



提示: Ext 是经典的 MVC 模式。

哪些是 DHTML 最佳实践?哪些是 Ajax 企业级应用?到 Ext 身上自然就不遑多论了。

总之,闪光的地方都是我们该学习的,并不因为开头所说的种种抵触而认为不值一谈,而且需要真正掌握的,毕竟还是以继承方法 Ext.extend()为先。Ext 是通过继承很多现成的组件来方便我们开发的,故此有必要先入为主地谈谈 Ext.extend()。

关于 Ext.extend(),其实是创建一个新类(Class)的方法,其方法形式为:

```
Ext.extend(baseClass, SubClass, Overrides)
```

第一个参数是父类,第二个参数是属性/函数的列表。第二个参数加入到对象的 prototype 中 extend 过后,Ext.extend 还会产生一个 superclass 的引用,用于访问父类的成员。这在下面第二个类 Developer 中有演示。



例程 2.1

```
// 基类 Person, 继承自 Object
Person = Ext.extend(Object, {
    constructor: function(first, last){
        this.firstName = first;
        this.lastName = last;
    },

    getName: function(){
        return this.firstName + ' ' + this.lastName;
    }
});

// 继承 Person 得到 Developer 的子类
Developer = Ext.extend(Person, {
    getName: function(){ // 重写了 getName() 方法
        if(this.isCoding){
            return 'Go Away!';
        }else{
            // 访问父类的方法
            return Developer.superclass.getName.call(this);
        }
    }
});

// 测试是否成功继承
var p = new Person('John', 'Smith');
alert("Hi, " + p.getName() + "欢迎来到学习 Ext 的殿堂: ");
```

在子类的 Developer 中我们重写(override)了 getName()方法, 还可以选择调用父类的原 getName()方法。虽然书写起来比较冗长, 但这正是 Ext OO 世界中所允许我们做的。

以类(Class)为单元是 Ext 面向对象编程的重要特征, 每一个 Ext 组件便是一个类, 就是这样一个个组件, 一个个“类”构成了庞大 Ext UI 库。事实上, 一般熟悉 Java/C#/PHP5 的用户在了解 Ext 的继承方式后, 只要克服了语法上差异等客观的因素, 不谈论末梢之处, 就可以快速地掌握 Ext OO、组件的原理, 并在 API 帮助下, 得心应手地写出自己的 UI 扩展。

此外, Ext 的 Examples 目录提供的例子达 100 多个, 可见官方真的是为我们一线用户着想的, 很多玩家一开始接触 Ext 之时便是从官方例子开始尝鲜的, 能好好地分析一下这些例子, 对认识 Ext 会有较大的帮助。再加上本书提供的大概 115 个例子, 将近有 200 多个例子了, 基本上涵盖了 Ext 的所有功能。

第6章 展示数据

大家喜欢 Ext，一个很重要的原因是喜欢 Ext 的 Grid。作者在没有使用 Ext 之前一直使用 dhtmlxGrid，其功能比 Ext 的 Grid 强大，但是缺点就是缺乏一个整合的框架平台，需要与其他平台结合使用。在 1.1 版本时，Grid 的功能还是比较弱的，作者的项目就是把 Ext 与 dhtmlxGrid 结合起来使用。从 2.0 版本后，作者就彻底放弃 dhtmlxGrid 了。虽然 dhtmlxGrid 的功能仍然比 Ext 的 Grid 强，但是 Ext 的 Grid 的功能已经能满足作者的需要了，而且由于平台整合在一起，不用再为平台不同造成的 CSS 与脚本冲突而处理很多细节问题。

Ext 的 Grid 最大特点就是显示模型和数据模型分离，数据的处理都在 Store 中，Grid 只负责显示和操作控制，这非常符合当前的开发潮流。

通过本章的学习，您将会了解 Ext 主要部件全方位的使用方法与技巧。



关键词

- Java/C#与前端的通信格式
- 接触 Ext 关键设计模式 MVC
- Grid/Form/Tree 等的数据绑定
- 一些辅助扩展、插件



6.1 如何产生组件读取的数据

6.2 表格组件 Grid

6.2.1 Grid 入门

Grid(表格、或称高级表格)是我们在展示数据的时候,尤其在我们希望非常清楚、非常有条理地将这些记录呈现在用户面前的时候一个非常重要的组件。从人机交互的角度和使用的频率方面来判别,Grid 的各项应用更能显出它作为拳头级别 UI 之举足轻重的位置。Ajax 尚未在大规模应用之前,我们如果要为表格(HTML Table)增添许多功能的话,还需要服务端编程交叉的帮助(不断地 Request/Response),可说是相当恼人与琐碎。

Ext 推出以后,其 Grid 组件(完全命名方式是 Ext.grid.GridPanel)解决了拼凑 HTML 表格生成 UI 的技术问题,从一个侧面来说省下了美工设计表格的工作量,而且封装了列移动、排序、即时编辑……那些本来就能在客户端完成的功能。与此同时,Grid 渐渐地也深受广大用户的欢迎,而且围绕 Grid 的扩展、插件也越发层出不穷,为 Grid 本身可扩展、优异的模型体系(Architecture Model)添加了鲜明的注脚。

许多人觉得一谈到“对象模型”,心里想:怎么又是泛泛而玄虚的概念,可没料到 UI 用上如此复杂的概念的呀!无怪乎 Ext 难学不绝于耳。当然,随手应用 Grid 即可以非常轻松地做出我们所需的“表格”,因此 Grid “面向程序员友好”这点是毫无疑问的,但倘若我们须做出某种程度的修改,又或者出于提高工作效率的目的,把 CRUD 的功能封装到一起、务求与 Grid UI 配合,如此的话便不得不深入 Grid 的内部,甚至揣摩其最初的开发初衷,而非仅仅停留在怎么用 Grid 和翻阅 API 文档之上。

追根溯源,当初 Ext 的 Grid 不正是参考了 Swing 的 Table 实现才可以有现今成熟的底子吗?再者,无论免费与否,既然 Ext 在大众面前提供如此这般的源代码、翔实的文档,我们只有好好研究一番才不会辜负 Jack 的一片苦心。



提示: 业界中不乏优秀的 Grid 控件,较著名的有国人的 GT-Grid、Dojo 基金辖下的 TurboGrid(捐赠开源项目)和能与 Ext Grid 匹敌的 dhtmlxGrid(半开源项目)等。有兴趣的读者可以多了解,学习 Ext 的 Grid 后即可对其他 Grid 触类旁通。

言归正传,我们即将步入 Grid 的世界,同时扩大议题来说,也是进入数据与 UI 实体如何结合的这一议题,该议题通常经过高度抽象化后才实施解决。我们先看一张图(见图 6.3)。

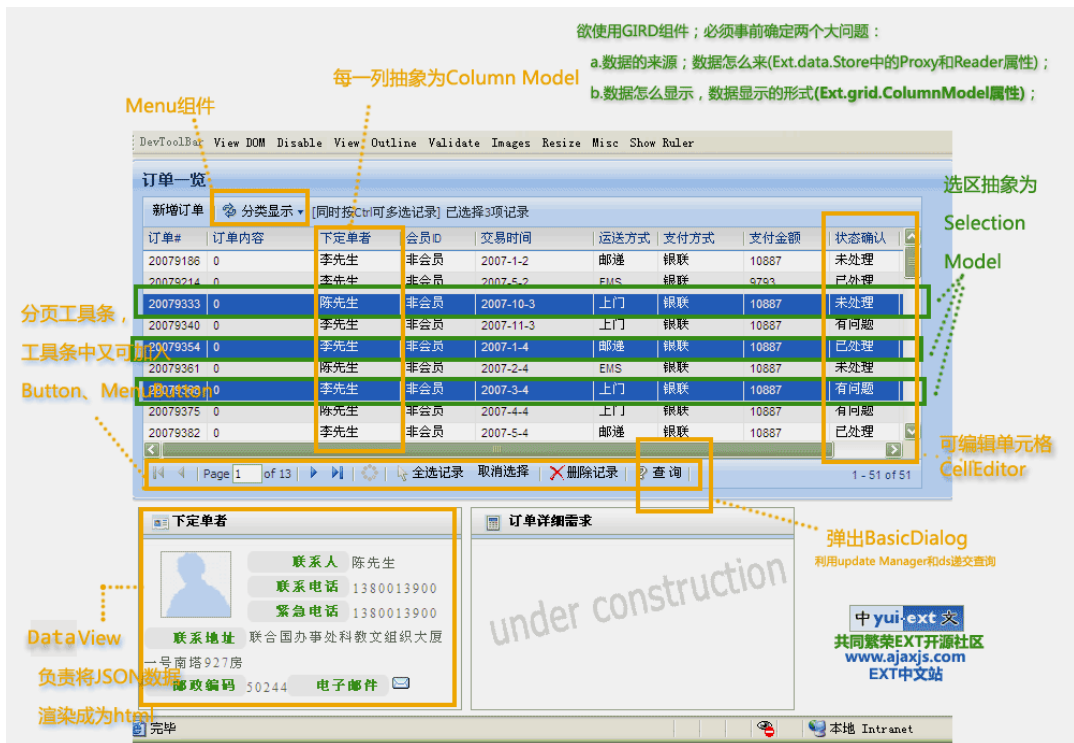


图 6.3 Grid UI 的分解图

图 6.3 大纲性地指出了对象所在, 当中有很多信息可以挖掘, 但是完全理解之后, 您会更清楚地认识到 Grid 是怎么组成的。实际上, 我们将详细地介绍藏匿于 Grid 背后的模型体系结构, 力图更细致地观察, 并讨论其涉及的数据模型, 当然该数据模型也可以延伸到 Tree、ComboBox 等其他组件的理解上。必须强调, 在运用新定义有别于默认组件行为的时候, 了解这些模型是不可或缺的, 例如, 如果需要一个显示不同数据类型的 UI 或需要一个按照非标准方式的事件的组件, 则必须使用 Ext 的数据模型。相关概念可不是腾云驾雾, 我们希望能够从以下这两方面来体现数据模型的作用:

- Ext 在 Grid 所要求的数据(模型)和实际渲染的数据(视图)之间进行了非常清楚的区分。这种区分的优点, 不但理论上被计算机科学家们所倡导着, 而且对所有开发人员而言也是一个重要的启示, 有效的区分意味着组件是极其灵活的, 很容易就修改 Grid 来显示自定义的新型数据, 或者不破坏最初的设计。
- 外观皮肤独立于 Ext。对象模型从不硬性陈述 CSS 规则。独立的 CSS 完全控制了组件的外观皮肤, 组件的外观都依靠 CSS 控制来控制屏幕上的显示形式, 并且可以让用户更多地控制应用程序的外观。即使在创建应用程序之后, 您也可在几种预设好的皮肤中选择。如需要打造个性的风格, 那 Ext 也支持您方便地这么做。

需要澄清的是, 上述概念非 Ext 独享。多年以来, Web 开发所强调的一项便是创建一个分离外观的中间层。但最常使用的实现模型, 却让客户主机变成“哑终端”的瘦模型。用大部分处理过程集中在后台主机服务器模型层上, 性能负荷比较大(这不属于优化不优化的问



题),而且存在的问题还包括外观、数据和业务逻辑都交织在一起,因此,实质上很难分离各个层,也就无法更新或替换其中的任何一个。

Web 应用设计是经由应用各层的分离演进:服务端成熟而趋向饱和,表示层的解放则落实在 RIA 中。尤其在现代 RIA 系统中,彻底将经典的 Web 三层结构作物理上的分离(见图 6.4),也是一种越来越受欢迎的方法,如 Flex/Sliverlight/Java Fx 等的 RIA 方案皆是。

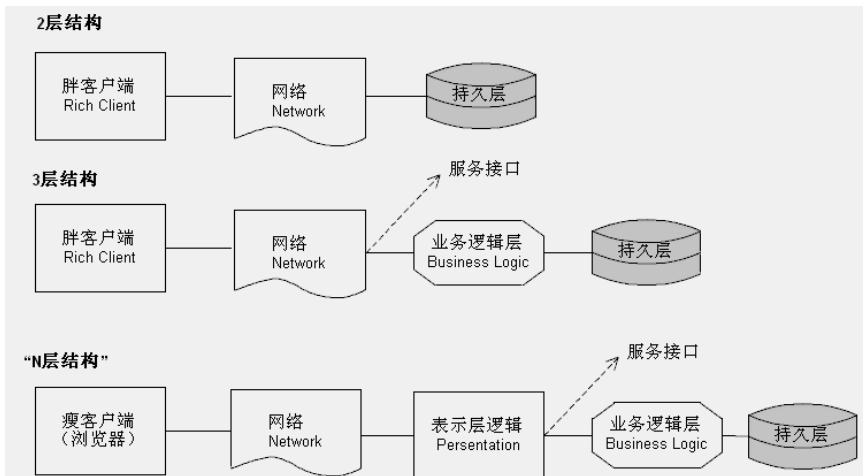


图 6.4 B/S 多层模型

这些 business 业务的东西,应该放到后端,前台只是展示结果的地方,这样的结果就是前端完全独立于后端,三层结构会更泾渭分明。作为中心的业务逻辑层,要求与数据层和外观层各自的关系分离开来。通常与外观层怎样耦合便是我们当前讨论的焦点,其分离难度也大。图 6.4 指出了几种客户端/服务端的比较,中间的三层结构是较为理想的模式。

若大部分业务都很类似,等于变化最小的可能就是数据,开发环境中最稳定的组件是数据库模式。在成熟的技术下,数据库随时都可以改变而业务逻辑保持不变。另一方面,系统中最不稳定的部份可能就是外观表示层,它可以由于新技术,公司的新战略或其他种种原因而做出改变。因此,我们必须从业务逻辑与外观表示层的混沌中勾勒出一条清晰的思路。

显然我们目标就是,外观层可以修改,甚至移动到另外一个新的载体,而不会对业务层产生影响,也就是刚才提到的“物理上的分离”便可以做到。将业务处理与外观分离会根据您的环境类型的差异有很大的不同,你可在业务层创建,更新数据库表的对象。接着根据业务处理的情况,在调用这些对象的外观层对象中将它们进行包装,供显示和渲染用。这样,外观层从服务端解放出来,而落实到我们开发者要做的,便是在上述方针指导下“加厚”、“加深”外观层。

支撑外观的一层需要对象映射的持久化为其服务,加入属于该层的数据实体。这种模型不但是远程的,还有一种向正交持久性(Orthogonal Persistence)的映射模型。其优点在于数据访问本身可以被不同的表格、树、下拉列表等的 UI 组件交互,UI 组件本身更可以被灵活替换,不会对业务层或数据层产生影响。Ext.Data.*便属于此数据实体工具之列,也就是显示模

型和数据模型又一次彻底地分离，数据的处理都在 `Ext.Data.Store` 中，`Grid` 只负责显示和操作控制，一方面也体现了外观层的“厚度”。



提示：这样一来，在 Ajax 前端中使用此方法意味着从一个 Dojo 或其他应用程序中访问这些数据与从一个 Ext 对象中访问一样容易。而 3.0 中的 `Ext.Direct` 则让分离的方法更进一步，仿佛就是直接联通外观层与数据层的桥梁。

既然独立分离和“加厚”外观层势在必行，那我们看看应该怎样设计它才好，`Grid` 就是最好的例子。下面的内容就直接进入 `Grid` 的主题。

理解 `Grid` 跟理解一个经典的“模型-视图-控制器”的步骤大致相似，也就是我们常说的“MVC”体系结构。实际上使用 MVC 作与组件的基本设计在 Ext 中许多地方都可以见到，Ext 普遍应用着 MVC 模式。理论上，MVC 将 GUI 组件拆分成三个元素，每个元素相互影响，也密切联系，对于组件如何表现起着至关重要的作用。怎么去理解呢？我们分别来看看 M、V、C 各自的定义。

1. Model

Model 叫作模型，用来存储组件的数据和状态。组件的数据往往是业务逻辑在前端的投影，也就是说，操作数据库的部分结合在表示层的 **Model** 中。

每一个 UI 组件都有其状态数据信息，该信息是由“模型”所定义的。不同类型的组件有不同的模型。一个完整组件的模型 **Model** 细节可能会很丰富，视组件的复杂程度而言。**Model** 的状态数据与行为准则构成了一个组件实例。我们先分析一下 **Model** 的状态数据是什么。如图 6.5 所示，`Slider` 活动杆组件的 **Model** 涉及了“滑杆”有关可调整的位置、最大值、最小值等的信息。所有这些信息必须配有一个或多个公共方法，使得开发者可以获取或改变它们的值。另外，不论组件在屏幕上是如何被描绘的，这些信息总是相同的，**Model** 数据总是独立于组件的可视化表示。

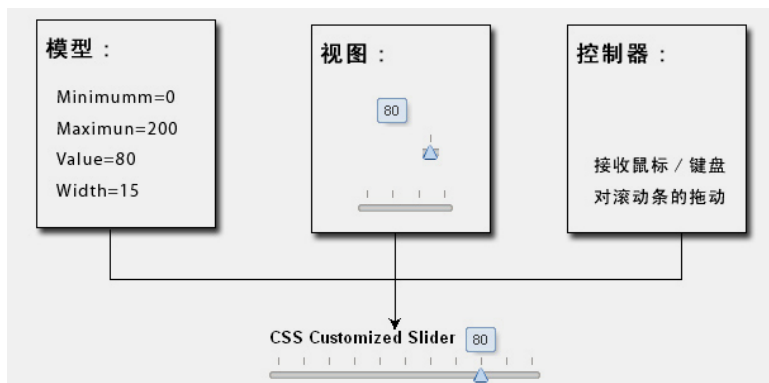


图 6.5 模型-视图-控制器的三个要素

一个组件可能包含多个 **Model** 对象在其中。这些 **Model** 的行为策略，经过代码封装后，

便实现成为组件实例对象的方法(可公有的或私有的)。

那么,为什么一个组件底下可能有不止一个 **Model** 设计为其服务呢?越是复杂的组件,特别 **Grid**、**Tree** 复杂的组件,将其庞大的应用分割为易于管理的多个模型 **Model**,进而对每个部分做深入的合理性、可用性、复用性的组件规划。一般我们经常接触的 **Store**、**ColumnModel**、**CellSelectionModel**、**RowSelectionModel**(**AbstractSelectionModel**)或其子类等的就是 **Grid** 模型的讨论范围。

单独阐述 **Model** 并不能有足够的信息来完整地描述 **MVC**, **MVC** 的优点之一在于能够将多个 **View** 视图与一套 **Model** 绑定在一起,这为开发者提供了选择组件外观而又避免影响数据的好处。我们接着看看 **View** 部分。

2. View

View 叫作视图,是指 **UI** 组件在屏幕上的表现形式。举个现实中的例子说,几乎窗体都有一个窗口顶端的标题栏。不过在标题栏左边的可能是一个关闭按钮(**Mac OS** 平台),或者可能在标题栏右边有一个关闭按钮(**Windows** 平台)。不同的 **GUI** 平台决定了设计上还是有区别的,那就是 **View** 范畴内的区别。

View 层与 **HTML/CSS** 很接近,以致有很多朋友可能第一反应便是“皮肤/模版”,我换了一套模版等于换了一套皮肤,皮肤/模版可以理解为视图 **View** 吗?——其实不尽然。视图属于 **MVC** 模式高度抽象后的概念,不是直观对象的产物,直接等同于“皮肤/模版”的话便有张冠李戴之嫌。应该说它们是不处于同一概念层面上的两种事物,而且横向看,视图的概念比模板广泛得多,充其量只能说, **MVC** 之要义“试图把视图和数据分开的原则”,在模板身上也有共同体现而已。

视图的输入和输出都依赖于 **Controller** 和 **Model** 协调参与。如 **Slider** 组件。如果不能获取 **Slider** 的模型信息, **Slider** 的视图就不能绘制滚动尺杆,只有获取尺杆相对于最大值和最小值的当前位置和宽度, **Slider** 才能知道绘制尺杆的位置。

再如 **Ext.Component/Ext.Container**,可以形容是把 **View** 顺利渲染出来的“总管”,它不但具备组件管理的功能,而且还定义了组件生存周期。生存周期完整地勾勒出了一个组件在复杂的布局中的表现形式, **Ext** 自动化的布局调整都离不开这两个基类。具体一点说,浏览器的问题亦属于 **View** 的职责范围内,比如在 **View** 交付渲染的过程中,就因应每个浏览器的差异作出正、负 2 像素的修正计算。

View 决定了数据如何显示,合并提供的数据以“界面的形式”显示给用户。**View** 不关心数据来自哪里或者怎么获取到,它只负责接收数据后使用的事宜。

3. Controller

一般来说, **View** 层不能运行他们自己的方法,举个例子来说,对话框 **Dialog** 不能自己控制开关,应该由 **Controller** 来控制它是否开关。

Controller 叫做控制器,主要职责是接受 **UI** 命令以及控制、更新 **UI**。尽管说得有点拗口了,其实“接受 **UI** 命令”是接受来自用户与视图互动之后形成的指令,然后进一步通知模型;

而“控制、更新 UI”就是此反向的流程，当数据操作或模型状态发生变化的期间提示视图要更新。从 Model 到 View 的数据必须通过 Controller，Controller 也负责这两者状态的维护，如果一个用户点击了一个 Dialog 中的保存按钮，这个点击动作由 Controller 来接收，Controller 发送一个动作来决定 Dialog 该做什么。可能是关闭 Dialog，也能是显示正在处理……一旦数据保存了，XHR 完成后会触发 Controller 发出另一个“隐藏指令”来关闭 Dialog。

Controller 的一个特点是它不是直接与用户操作面对面，不容易具象地说出它在哪里。再如 Slider 组件的场景，发生了鼠标单击的事件，可通过视图实际产生的滑杆宽度，而得知 Slider 上(或外)发生的单击事件。实际上视图就界定了 Slider 是否接收用户指令——如果是，便发送信息给 Controller，由此决定 Controller 如何处理这些事件。假如用户拖动了滑杆，此时此刻 Controller 清楚发生了哪一些变化，它就会将变化的位置告诉 Model 对此做出反映，最终令 View 层发生变化。

在我们的 Ext 里面，无论是哪一种流程，Controller 的行为均以事件 Observable 的解耦模型为基础，进而与其他 View、Model 表述，但切记 Observable 不等于 Controller，千万不要混淆在一起。事件的形式有很多种，例如鼠标单击、特定菜单的组合快捷键得到了响应，或者有新的记录来到要重新刷新 Grid。Controller 掌握了事件发生的准确时间点，也决定了每个组件如何对事件做出反应。

以上描述了 M、V、C 三者的相互关系，我们还可以参考一下简图 6.6。透过 MVC 所规定的方式，MVC 三个元素中的每一个元素都要求维护着其他的元素，来保持自身不断地更新。

到目前为止，我们的例子还局限在比较简单的 Slider 论述中，下面就结合 Grid 与 MVC 模式展开这方面的论述。

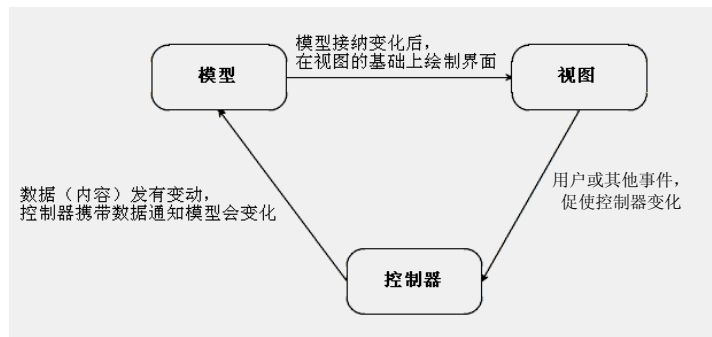


图 6.6 MVC 关系图

GridPanel 与 GridView 是创建了 Grid 的 UI 视图。初始化 Grid 的时候，读取 ColumnModel 模型上各个列的配置信息，告诉这一列的内容显示什么，是否可即时输入，又或者弹出一个“日历选取 Widget”等类问题，而当中必不可少的是 dataIndex 的配置，该配置的值将对应 Store 集合中的一行记录，取得所在列的数据后就成为单元格的原始数据，原始数据是不是要转换为真正的显示用数据？对的，我们定义的 ColumnModel 中的 render 渲染函数，就可告诉 GridPanel 应该怎么输出数据(Model & View)。



Store 和 ColumnModel 不约而同地都扮演着模型 Model 所分配的角色，而且干得很好，而 Column 定义的 render 我们不可以硬说它是 Model，应该叫 Controller 才对，因为它控制原始数据如何转换为 View 视图的数据。

Ext 中用 store.load() 或 grid.getStore().load() 加载 Grid 数据——明显地这不是 Grid 加载数据而是 Store 在加载数据。准确地说 Store 实际是在客户端的一堆记录集合而已。Ext.data.DataProxy.* 就更像是一个 Controller 的作用，它指定了 Grid 模型数据是本地的，还是 HTTP 的，或者是跨域的 ScriptTag 的，最后通知 Store 模型数据记录记载完毕，接下就是别的数据解析工作，Controller 方面的工作完成。Ext.data.DataReader.* 用于读取结构化数据(来自数据源)然后转换为 Record 对象集合和元数据 Store 二者合成的对象。结果是 Store 产生了符合 Grid 读取的合法数据结构(Controller & Model)。

再如某一项操作，用户需要删除某一条记录，首先是选择了 Grid 的某一行，这触发了 RowSelectionModel 的选择事件，视图方面就会突出显示这条记录；用户再按下给出“删除”按钮——确认删除，这时该按钮所分配的事件处理函数就相当于控制器(用户新的控制器)，得到“删除”的指令后通知 Store 模式删除记录，Store 与 Grid 是同步的，整个 Grid 就会刷新，从而完成一次控制器到的视图控制(Controller & View)。

可是 Grid 的 MVC 应用的地方还不止于此，里面许多的起承转合关系仍离不开 M、V、C 三者的要素结合与分工。小结一下说，Grid 结合 MVC 之要义，依然还是万变不离其宗。进一步学习 Grid，就要从其 API 入手，清晰地理解有什么类参与其中，代表着哪些的作用。Grid API 所涉及的类有 16 个之多，我们选出其中较关键、较具代表性的 7 类，另外还加上非常重要的 Store “客户端记录缓存类”列在表 6.5 中，详细予以说明。

表 6.5 测试返回结果

类	作 用
<code>class GridPanel</code> <code>extends Ext.Panel</code>	集合 <code>Grid</code> 的主要接口的面板类。一般就是实例化这个类来创建 <code>Grid</code> 控件。大体来说, 涉及 <code>Grid</code> 的各项操作就以它为起点。 <code>GridPanel</code> 依赖以下几个要素。 <code>Store</code> : 数据记录的模型。 <code>ColumnModel</code> : 列怎么显示。 <code>GridView</code> : 封装了用户界面。 <code>AbstractSelectionModel</code> 或其子类: 选择行为的模型
<code>class GridView</code> <code>extends Ext.util.Observable</code>	<code>GridView</code> 类是对 <code>GridPanel</code> 用户界面的封装。为产生特殊的现实效果, 可用该类的一些方法来控制用户界面。 <code>GridView</code> 类不提供其所属数据的操作方式。 <code>Grid</code> 的数据模型是交由 <code>Store</code> 实现的
<code>class ColumnModel</code> <code>extends Ext.util.Observable</code>	为 <code>Grid</code> 的垂直列(<code>Columns</code> , 相对于 <code>Row</code> 行)定义的模型。此模型提供的信息用于显示和编辑单元格、显示列标题栏(<code>Title</code>)、显示列宽度(<code>Width</code>)等诸多配置的信息, 也是开发者要精心制作的部分
<code>class AbstractSelectionModel</code> <code>extends Ext.util.Observable</code>	<code>Grid</code> 选择模型的抽象基类, 让 <code>Grid</code> 的数据可以被选取选中。它直接继承事件 <code>Observable</code> 来为子类提供接口, 该抽象类不能直接实例化
<code>class CellSelectionModel</code> <code>extends AbstractSelectionModel</code>	API 默认自带两种选择模型, 分别针对单元格(<code>Cell</code>)和行记录(<code>Row</code>)。 <code>CellSelectionModel</code> 类提供了 <code>Grid</code> 单元格选区的基本实现, 可通过 <code>Checkbox</code> 实现选择或不选择行
<code>class RowSelectionModel</code> <code>extends AbstractSelectionModel</code>	为 <code>Grid</code> 提供行(<code>Row</code>)选择功能。用户对 <code>Grid</code> 做单一选择、连续区间选择(配合 <code>Shift</code> 键)和多重选择(配合 <code>Ctrl</code> 键)时, 都发生在这个类中。如果用户手工选择 8 行不连续的记录时, 就会产生 8 个选区选择事件。但如果先选择第一行, 再用 <code>Shift</code> 键选中最后的行记录, 最终选中 8 行记录, 就只会产生两个选择事件
<code>class CheckboxSelectionModel</code> <code>extends RowSelectionModel</code>	通过 <code>checkbox</code> 选择或反选时触发选区轮换的一个制定选区模型。它是基于 <code>RowSelectionModel</code> 的一个扩展
<code>class Ext.data.Store</code> <code>extends Ext.util.Observable</code>	前面所述的包括 <code>GridPanel</code> 在内许多对象都存储了大量的 <code>Grid</code> 配置信息, 却不包括 <code>Grid</code> 的实际数据记录。 <code>Grid</code> UI 中显示的记录统统保存在 <code>Store</code> 对象里面, 不单 <code>Grid</code> , 许多组件模型的记录源于 <code>Store</code> 。放在一起就是“怎么显示”与“显示什么”的问题。 <code>Store</code> 与 <code>GridPanel</code> 一起构成完整的 <code>Grid</code> 控件。支撑起 <code>Store</code> 的服务有多种的形式, 例如 <code>JsonReader...</code> / <code>HttpProxy</code> 、 <code>DwrProxy</code> 、 <code>ScriptTagProxy...</code> 这些是“数据驱动”部分内容

API 如此丰富和铺张地摆到我们面前, 我们先不必“囫圇吞枣”。实作上, 官方提供了很多 `Grid` 的使用例子, 这些都是重要的参考来源。配合下面给出的剖析图, 我们可以直观地



了解一下 Grid 组件的轮廓大致为如何。主要来说, Ext.data.Store 把任何格式的数据转化成 grid 可以使用的形式, proxy 告诉我们从哪里获得数据, reader 告诉我们如何解析这个数据。SelectionModel 就是 UI 渲染之后, 用户选中记录的堆栈。Grid 这个庞大的对象就是依靠下面许多“小”的对象所组装而成的。请读者预览图 6.7, 以便下一章正式进入 Grid 的实战。

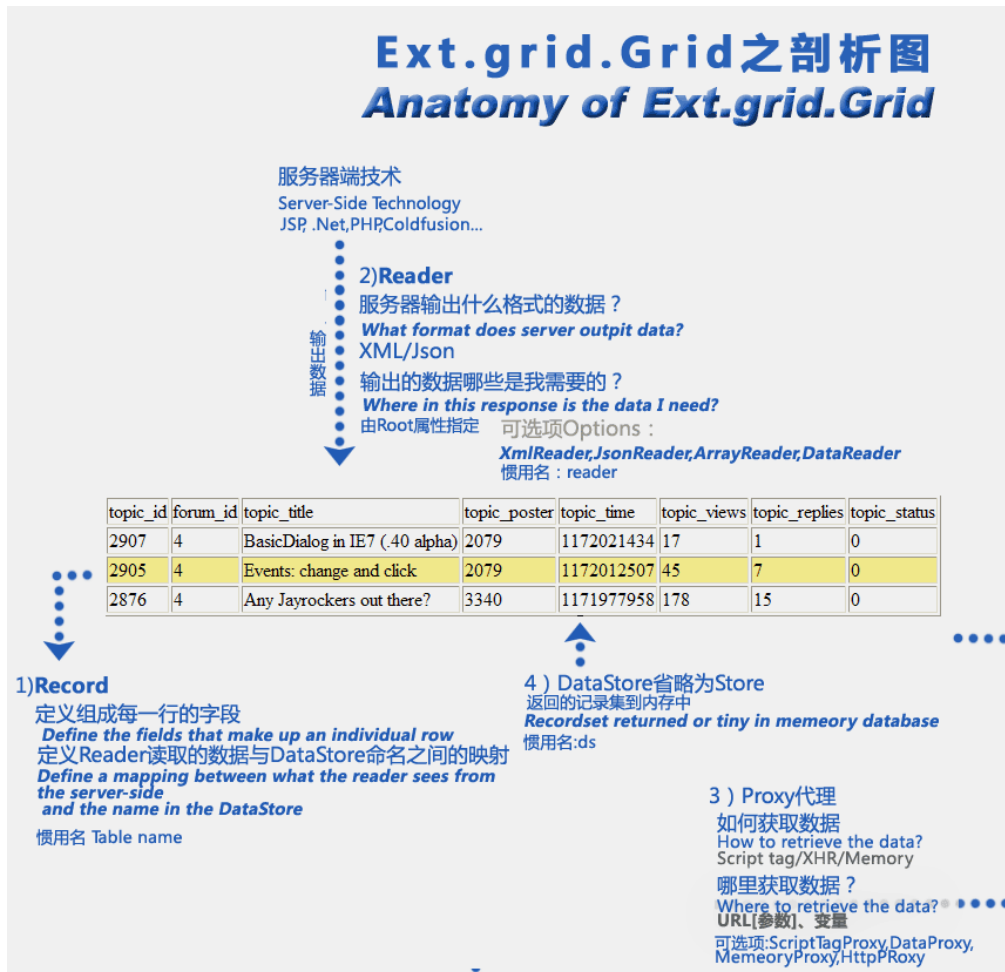


图 6.7 Grid 的剖析图

图 6.7 的说明是围绕 Grid 以及 Grid 各辅助组件出发的。从扁平化的视觉角度来看, 对 Grid 的编程一般涉及这几个步骤: ①准备好后台分页或者查询的数据, 本章第一小节已经讨论过(如测试为先, 测试数据不一定要分页, 只要能生成 Grid 格式的 XML/JSON 即可, 分页可放在最后); ②定义数据映射实体 Store, 供任意一个 UI 组件用, 该部分内容权重较大, 另开新一章讨论(第 7 章“数据驱动”); ③制订列模型 Column, 这是典型的 UI 设计部分; ④Grid 是否要求可编辑单元格, 安排单元格特定控件; ⑤Grid 大概轮廓已经定型, 调整其外观和细节, 具体一点说, 就是高度、宽度是否正确显示, 有没有出现滚动体等更细节的问题; ⑥对 Grid 的事件编程, 与其他部分发生聚合关系。

6.2.2 Store 结构

这里所说的结构并不是 Grid UI 本身如何形成数据结构, 而是 Grid 读取的内容的流程结构, 也就是讨论 Store。Store 可看作是由 Ext.data.Record 记录构成的 UI 专用数据源。服务端逻辑答应客户端请求后, 原始数据还不能立刻成为页面数据, 先“卡”在 Store 中统一化处理再分配各个 UI。

Store 就是对这些原始数据的前期初始化和健壮性的封装(以 MixCollection 基础类型为数据容器)。对于各个 Store 子对象, 最先开始工作的是 Ext.data.DataProxy 对象:

```
var ds = new Ext.data.Store({
    proxy: new Ext.data.HttpProxy({url: 'getUserList.do'}),
});
```

大多数场合使用的 DataProxy 都是 HttpProxy, HttpProxy 是 DataProxy 的特例, 其他的特例还有 MemoryProxy(客户端内容代理)、ScriptTagProxy(用于跨域访问资源)和 DirectProxy(模仿 Java DWR 框架之原理)等。我们传入 url 的配置项即可让 Store 得知从哪里获取远端数据的入口(如 url: 'getUserList.xml', 可以是*.aspx/*.php/*.do 等类型的后台通讯地址)。Proxy 通讯完毕后即会触发 load 的方法告知 Grid 即使现在断开网络也可以了, 因为此时 Store 已经拿到数据了。

另外一个重要的问题是, 开始得到的这些是原始的数据, 不能直接派上用场, 怎样转化为 Store 可读取的数据类型? 我们听从 Store 的安排, 交给 DataReader 负责解析原始数据, 诸如字段类型、长度、排序、映射字段此等读取解析的问题, 如例程 6.5(a)所示。



注意: 例子需要在服务器上跑起来。我们这里 url 指定的是一个普通的 getUserList.xml 文档用于测试, 如果不是测试环境的话那就是某后台地址的入口, 动态生成 XML 内容。

例程 6.5(a)

```
alert('本例须在一个服务器上跑, 请注意。');
var ds = new Ext.data.Store({
    proxy: new Ext.data.HttpProxy({url: 'getUserList.xml'}),
    // 假设返回 XML 文档, 便要用 XmlReader
    reader: new Ext.data.XmlReader({
        record: 'item', // <userList>标签下的<item></item>子元素才是一笔笔的记录
        id: 'ID',
        totalRecords: "recCount" // 总数一定要告知 Ext……分页时有用
    }, [
        'Name', 'bIsAdministrator',
        'bIsActive', 'tsDateLastLogin' // 数据记录的各个字段组成的数组
    ])
});
// 登记 load 事件, 通知 getUserList.xml 中的记录加载完毕
ds.on('load', function(storeObj, recordList, paramsObj){
```



```
alert('getUserList.xml 中的记录加载完毕! \n 当前取得的记录数: '
+ storeObj.getCount());
});

ds.load({params:{start: 0, limit: 25}});
```

XML 格式的就用 XMLReader; JSON 格式的就用 JSONReader。当然, 我们遇到更多的格式也可以继承 Ext.data.DataReader, 又或者“超 JSON”格式的就继承 Ext.data.JSONReader 创建新类 SuperJSONReader(如 JSON-P、myXMLReader 任何一种), 然后和 Proxy 一样交给 Grid 全权负责统筹, 技术上实现的模式类似于“构造式”对象注入。总之对于任何的 UI 而言均是: 同一的接口, 同一的 url, 同一生成的记录实体。

Store 的定义设置完毕无误后, 可执行 Store.load()方法加载数据。我们打算先取出一部分记录, 如参数“{params:{start: 0, limit: 25}}”的 25 笔记录, 但当前例子还未实现, 当前取出的却是全部的 104 笔记录(见图 6.8), 为什么呢? 我们不妨看看, 因为当前 getUserList.xml 文件是静态的模拟文件, 没有分页的控制。下面的例子将集中为读者介绍。需要指出的是, 分页的算法由 UI 组件 Ext.PagingToolbar 接受用户交互行为自动调整, 故一般无须额外设置(除非不是 start/limit 式分页或其他需求上的变动)。

```
ds.load({params:{start: 0, limit: 25}});
```



图 6.8 说明了未分页

宏观而言, 为实现状态迁移(State Transfers), Web 世界充满着异步的概念, load()也是一个异步操作。UI 组件等待 Web Server 返回信息, 也就是得到这个 load()完成的消息后才可以刷新视图。必须再次说明的是, 获取那些得到的数据正是在 Store 底下进行的, 记录实体就保存在 Store.data 中(data 为 MixedCollection 结构), 而不是 Grid。如果划分“分页”的职责, 就当仁不让地交给 Ext.PagingToolbar 负责相关逻辑, 故 Store 没有自己的分页命令, 定位在 UI 控制器与数据驱动的绑定服务, 提高相关的操作接口, 本身并不关心数据实体的各种数据结构。值得一提的是, 我们还可以微妙地理解 Store 成为客户端的一级缓存(cache)。Store 与单个或多个 UI 组件一旦建立绑定的关系之后, 开发人员无须为彼此对象之间的视图与数据的同步而担心。若手动联系它们则是多此一举。Store 有新数据会立刻告知 UI 刷新。在 MVC 理念的指导下, 它们清楚这一契约关系应当是自动实现的。

另有一个问题是, 会不会 Store 加载了数据 UI 却还未来得及渲染呢? 有这种可能, 故此我们一般是在 Grid 初始化甚至 render()完毕后, 才调用 ds.load(), 以避免不同步的问题。

那么 load()到底返回了什么数据呢? 在 Firebug 的调试下返回了什么显而易见, 各位读者应好好利用 Firebug 的各项功能为自己的调试服务, 注意观察 B/S 之间的往返数据是其中的关键。下面 XML 是两行样本数据, 由 getUserList.xml 返回。

例程 6.5 getUserList.xml 片断

```
<userList>
<recCount>99</recCount>
<item>
```

```
<ID>2350</ID>
<Name>刘大同</Name>
<bIsAdministrator>0</bIsAdministrator>
<bIsActive>1</bIsActive>
<tsDateLastLogin>2007-05-01T14:34:57</tsDateLastLogin>
</item>
<item>
  <ID>4027</ID>
  <Name>王小平</Name>
  <bIsAdministrator>0</bIsAdministrator>
  <bIsActive>1</bIsActive>
  <tsDateLastLogin>2007-04-29T16:29:33</tsDateLastLogin>
</item>
...
</userList>
```

这段 XML 透露了几个关键的地方, `<item>` 与 `Ext.data.XmlReader.record` 配置项对应; `<ID>` 与 `Ext.data.XmlReader.record.id` 对应, `id` 是 Store 每笔记录的主键, Store 不关心其他字段, 除 `id` 是值得注意的。 `<recCount>` 与 `Ext.data.XmlReader.totalRecords` 对应, 是该一次查询结果的记录总数, 指的是所有的记录数, 不是当前取的记录数, 用于分页。保证这些字段与你的 XML 数据中元素名称是一致的, 否则 Ext 不能成功映射(Mapping)。当然还要让 Grid 知道 XML 文档定义了每一行是什么记录。我们 XML 样本下的 `<item>` 就代表你需要显示的字段(Fields), 因此 `Ext.data.XmlReader` 构造器的第 2 个参数便是一个数组参数, 把这些字段放到一个数组之中, 用于转化 `Ext.data.Record`。



注意: 使用 XML 例子会比较清晰地说明结构如何。如用 JSON 结构虽不同但所讨论的要素也是相同的。可参考第二个(b)例程或 6.1.2 小节输出 Grid 的 JSON 文本格式。

例程 6.5(b)

```
var ds = new Ext.data.Store({
  proxy: new Ext.data.HttpProxy({url: 'getUserList.json'}),
  // 假设返回 JSON, 便要用 JsonReader
  reader: new Ext.data.JsonReader({
    root: 'rows',           // 该属性指定包含所有行对象的数组
    id: 'ID',               // 该属性指定包含所有行对象的数组
    totalProperty: 'recCount' // 该属性指定记录集的总数(可选)
  }, [
    'Name', 'bIsAdministrator',
    'bIsActive', 'tsDateLastLogin' // 数据记录的各个字段组成的数组
  ])
});
// 登记 load 事件, 通知 getUserList.xml 中的记录加载完毕
ds.on('load', function(storeObj, recordList, paramsObj){
  alert('getUserList.xml 中的记录加载完毕! \n 当前取得的记录数: '
    + storeObj.getCount());
});
```



```
ds.load({params:{start: 0, limit: 25}});
```

例子程序 6.5(b) getUserList.json 文本

```
{
  recCount: 114,
  rows:[
    {
      "ID":2350,"Name":"刘大同","bIsAdministrator":
        0,"bIsActive":1,"tsDateLastLogin":"2007-05-01T14:34:57"
    },
    {
      "ID":4027,"Name":"王小平","bIsAdministrator":
        0,"bIsActive":0,"tsDateLastLogin":"2007-04-29T16:29:33"
    },
    {
      "ID":2340,"Name":"陈二","bIsAdministrator":0,"bIsActive":
        1,"tsDateLastLogin":"2007-05-01T14:34:57"},
    {
      "ID":4028,"Name":"张三","bIsAdministrator":
        0,"bIsActive":0,"tsDateLastLogin":"2007-04-29T16:29:33"
    }
  ]
}
```

需要说明的是,例程 6.5 定义字段数据(Fields)采用最简的形式,数组中没有更详细的信息,但如果是日期类型的值、就不得不需要把字符串转换为 JavaScript 日期类型。

Ext.data.Record 考虑了这点,提供如下颗粒度更细的记录定义:

```
var reader = new Ext.data.XmlReader({
    record: 'item',
    id: 'ID',
    totalRecords: "recCount"
}, [
    {name: 'title', mapping: 'topic_title'},
    {name: 'author', mapping: 'username'},
    {name: 'totalPosts', mapping: 'topic_replies', type: 'int'},
    {name: 'lastPost', mapping: 'post_time', type: 'date'},
    {name: 'lastPoster', mapping: 'user2'},
    {name: 'excerpt', mapping: 'post_text'}
]);
```

totalPosts 字段定义了映射字段名为 topic_replies, 类型是整数类型 int; lastPost 字段定义了映射字段名为 post_time, 类型是日期类型 date; 其他字段也有重新映射的字段,用于在 Reader 的数据对象中提取真实的数据。

除此之外,尚有以下 API 提供的配置信息定义字段,以加强数据的健壮性和颗粒精度,字段所允许的定义项还有 dateFormat(传入 Date.parseDate()的字符串参数)、defaultValue(当 mapping 的引用项是 undefined 的时候会使用该值,默认为'')、sortDir(排序方向)、sortType(排

序函数)、convert(Reader 转换值的函数), 在文档中都有详细的说明。

我们总结运行 Store 依赖的对象有如下几个。

- Ext.data.DataProxy: 获取未格式化的原始数据。常用子类为 HttpProxy、ArrayProxy、ScriptTagProxy 等。
- Ext.data.DataReader: 读取来自 DataProxy 的结构化数据, 转换为 Ext.data.Record 对象集合。常用子类为 XmlReader、JsonReader、DwrReader、DirectReader 等。
- Ext.data.DataWriter: 如果“读(GET)”操作在 Ext 中没有提的话, 那么其他三种操作就应该归类为“写(POST/PUT)”操作了。这部分的内容, 在 Ext 3.0 中将由 Writer 来负责完成。Ext.data.DataWriter 提供了增、删、改、查的实现, 让 Ext.data.Store 与服务端框架密切通讯。Writer 控制了 Store 了自动管理 Ajax 的请求, 让其成为 Store CRUD 操作的管道。换句话说, 这个 DataWriter 相对于远程 CRUD 操作而言, 是一个在前端的初始化部分, 负责写入单个或多个 Ext.data.Record 对象。
- Ext.data.Record: Record 类不但封装了 Record 字段信息, 还封装了 Store 里面所使用的 Record 对象的值信息, 并且方便任何透过 Store(模型见图 6.9)来访问 Records 缓存之信息的代码。
- Ext.data.Field: 该类封装了 Ext.data.Record.create()所传入字段定义对象的信息。开发者一般不需要实例化该类。方法执行过程中创建该实例, 并且在 Record 构造器的原型中, Ext.data.Record.fields 属性就是该实例。
- Ext.Error: 异常处理开始进入了 Ext 前端开发的视野。开发过程中, 尤其对于经验不怎么丰富的程序员来说, 准确的异常信息更能友好地帮助开发者解决问题和异常。如果系统能够提供更多这方面的信息, 就相应地添置新模块。在 Ext 3.0 中, 我们称作 Ext.Error 专门应付可知的异常(不可知异常由 try...catch(e)...捕获)。通过 Ext.Error 可定义新的异常, 程序员根据特定情况选择那一时刻实例化 Ext.Error, 抛出异常。例如下面是 data 空间下的报错信息——“DataProxy 尝试执行 API 动作但 url/function 未定义。请检查 Proxy 的 url 或 API 的配置。不能在服务器上定位 root 属性, 请参阅 JsonReader 理解更多。”

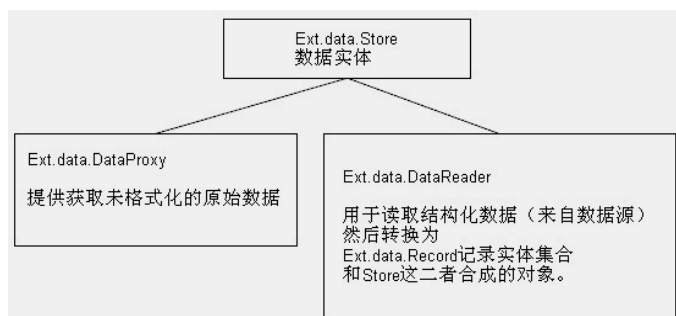


图 6.9 Store 基本模型

最后把这些对象都注入 Store, 一般是构造式注入。Proxy 和 Reader 就是 Ext 获取数据的最基本要素了, 有时候 Ext 还允许我们什么 Store 也不写, 传入 Record 数组即可, 当然最终



内部还会转换为 Store 对象。大量的细节尽管在封装过后理应无须再去深入，但我们还是鼓励大家有条件的话进入 Store 对象模型的高级特性，因为本章节以 Grid 内容为重点，Store 限于蜻蜓点水地进行介绍，但 Store 所涉及的“数据驱动”，包括 Ext.Direct，都是纵深面不俗的层面，在提取记录、分组信息和复杂多维的查询等操作中都涉及 Store，我们将安排专门一章(第 7 章“数据驱动”)，来负责全面地介绍。

6.2.3 分页、查询、排序

1. 用 PagingToolbar 作分页的查询

因为 Grid 只负责数据的显示和与用户的互动，也可作为纯 UI 部分了，所以 Grid 的分页、查询和远程排序实际上是对 Store 进行分页、查询和远程排序。由于 Store 本身没有分页的操作，可看作纯粹一个数据实体的容器，所以在 Ext.PagingToolbar 控制下 Store 就能分页了。

不知读者是否还记得上面 Store 返回的结构，分页工具栏的数量总条目来自 JSON 的 totalProperty 属性。该名称可以修改为其他的，如前例的 totalRecords、totalRecords……但总得存在。



提示：按照上述逻辑，分页工具条 PagingToolbar 可以不知道哪个 Grid，但不能不知道哪个 Store 是它所服务的。

每一种数据库查询方式都有其特定的方案来解决分页的问题，以上提出的要预知 totalProperty 总数就是其中的一种。Ext 好像与开源软件更志同道合，本身直接支持 MySQL/PostgreSQL 数据库 LIMIT/OFFSET 方式的查询分页。分页的参数由分页工具条 PageToolbar 传入的变量 limit 和 start 所决定，得到 ds.load({params:{start: 0, limit: 25}});参数后的 SQL 语句可直接返回指定的记录，至于当前页、每页记录数就由 PageToolbar 决定；如使用 Oracle 数据库也可用该模式分页，要注意的是 Oracle 的 RowNum 代替了 limit 关键字；如果是 MS SQL Server/Access 数据库的话，就不具备 LIMIT/OFFSET 的特性，但可以用 TOP 关键字写一个转换，基本过程如下：

```
// 变量 start、limit 是输入的起始数、记录数；变量 sql 是输出查询的 SQL 语句
sql = 'SELECT * FROM ('
    + 'SELECT TOP {0} * FROM (SELECT TOP {1} FORM Users ORDER BY ID DESC)'
    + ' AS foo' + ' ORDER BY ID ASC) AS bar' + ' ORDER BY ID DESC';
sql = sql.format(limit, start + limit); // format()格式化合字符串
```



注意：①上述 SQL 语句中，ID 字段是“限制行数结婚集的倒序”的字段，可因应实际的字段而修改。Users 是当前的表名。②Grid 的分页必须依靠服务端来划分好每一页的数据，才可以完成。不能把所有数据塞给前端进而在前端上进行分页。

下面进入实际的案例。例程 6.6 不但实现了分页功能，还提供分类信息、查询和排序的

功能,较完整地演示了 Grid 复杂的应用。

首先见其截图 6.10,我们可以看到,左边有一个类别树,右边顶部是一个分页工具条,工具条里有一个搜索输入框,下面是一个主体 Grid。



图 6.10 Grid 的分页、查询、远程排序

顶部 `tbar` 的分页栏就有一个 `SearchField` 搜索查询控件,用于接收用户查询的关键字。先看看代码是如何实现分页的。在定义分页工具条 `pageToolbar` 的时候定义了参数 `pageSize`,这个便是每页的数据总数。而 `store` 参数与 Grid 的定义参数 `Store` 是一样的。在 `items` 里加入 `SearchField`,它的参数 `store` 也与 Grid 的定义参数 `Store` 是一样的。从这里可以看到,透过设置 `store` 参数的定义就可以将分页、查询与远程排序等操作与 `Store` 紧密地联系起来了,以上的控件的任何操作都会触发 `Store` 更新提交参数并重新提交获取新的数据。

例程 6.6

```
var pageToolbar = new Ext.PagingToolbar({
    pageSize: 10,
    displayInfo: true,
    store: store,
    items: ['- ', new Ext.app.SearchField({store: store})]
});
```

在树中还定义了树的单击事件。每当单击树的节点时,就更新 `store` 的 `baseParams` 参数的 `classid` 属性为当前单击节点的 `id`,再设置 `query` 属性为空,然后通过 `load()` 方法加载属性数据。通过该操作可实现 Grid 的分类显示。

```
function changClass(node, e) {
    store.baseParams.classid = node.id
    store.baseParams.query = "";
    store.load();
}
```

要实现树节点接收数据,树节点的属性 `leaf` 一定要设置为 `false`,原因是设置为 `true` 后,



该节点为子节点，是不允许放下数据的。但是该节点实际上是子节点，此时该怎么办呢？只要为该节点增加一个 **children** 属性，其值为一个空的数组就可以了，格式如{"id": /*编号*/, "text": /* 显示文本*/, "allowDrag": false, "children": []}, 总不能让输出的 JSON 为 null。

下面看看后台如何处理前台的操作。尽管是 C# 为范例的，但很容易转变为 Java 语法。

例程 6.6 C# 代码清单 grid3_action.ashx

```
// 返回图片数据
private string List(HttpContext context)
{
    int currentPage = 1;
    int pageSize = 10;
    int limit;
    int.TryParse(context.Request.Params["limit"], out limit);
    int start;
    int.TryParse(context.Request.Params["start"], out start);
    int classid;
    int.TryParse(context.Request.Params["classid"], out classid);
    if (classid == 0) classid = -99;
    string orderColumn = context.Request.Params["sort"];
    string orderBy = context.Request.Params["dir"] == "ASC" ? "asc" : "desc";
    switch (orderColumn)
    {
        case "id":
            orderColumn = "id";
            break;
        case "title":
            orderColumn = "title";
            break;
        case "filename":
            orderColumn = "filename";
            break;
        case "filesize":
            orderColumn = "filesize";
            break;
        default:
            orderColumn = "createtime";
            break;
    }
    string filter = context.Request.Params["query"];
    if (filter == null) filter = "";
    if (filter == "")
    {
        if (classid == -1) classid = 0;
        filter = (classid == -99) ? "" : ("classid=" + classid.ToString());
    }
    else
    {
        filter = filter.Replace("'", "''");
        filter = String.Format(
```

```
        "title like '{0}%' or filename like '{0}%', filter);  
    }  
    currentPage = start / pageSize + 1;  
    return MyPageList.PageList.getList(  
        "t_dataview", "id", orderColumn + " "  
        + orderBy, " id, title, filename, sortorder, createtime, filesize, width,  
        height, classid", pageSize, currentPage, filter);  
    }
```

在 List 方法中, 变量 `currentPage` 表示当前页数, 默认是 1。

变量 `pageSize` 表示每页的数据总数, 默认值是 10, 当然这个参数也可以通过前台获取。

变量 `limit` 与 `pageSize` 一样, 表示每页的数据总数, 不过它的数值是通过前台提交的 `limit` 参数获取的。

变量 `start` 表示当前页数的开始序号, 通过它与每页的数据总数计算出当前页数。变量 `classid` 则表示类别的编号, 如果前台没提交该参数, 则将该值设置为代表全部数据的编号值 -99。

变量 `orderColumn` 获取前台提交的排序列, 该排序列提交的其实就是 `store` 中定义的字段名称, 虽然 `store` 中的字段名称与实际的字段名称可能完全一样, 但是为了安全, 最好还是通过 `switch` 语句将提交列字段名称转换为实际的字段名称。

变量 `orderBy` 获取的是前台提交的排序序列是顺序还是倒序, 为了安全, 这里也需要做一次转换。

变量 `filter` 就是获取前台 `SearchField` 中提供的查询值, 如果没有定义 `SearchField` 的参数 `paramName`, 则 `store` 会以 `query` 作为默认参数提交。如果有查询值, 则需要组合查询条件, 本例子是查询 `title` 字段与 `filename` 字段中包含有查询值的数据。如果没有查询值则判断是否需要根据分类获取数据, 如果值为 -99, 则获取全部数据, 否则设置查询条件为 `classid` 等于获取的类别值。

当以上的变量计算完毕时, 就通过执行自定义的一个分页类 `PageList`, 通过存储过程获取分页数据。

从以上的一个典型的例子可以看到要实现的 Grid 怎么分页, 下面就看看怎么为数列排列顺序。远程排序的话, 只要在定义 `Store` 时设置参数 `remoteSort` 为 `true` 即可:

```
var store = new Ext.data.Store({  
    url: 'grid3_action.ashx?act=list',  
    remoteSort: true,  
    ...// 其他 Store 的配置略  
}); //store
```

2. 怎么来排序

现在我们设定 `remoteSort: false`, 表示利用 JavaScript 本地的排序。Grid 的本地排序有一个 bug, 便是不能进行中文排序。要修正这个 bug, 需要覆盖 `Store` 的 `sortData` 方法。其原理



是，在 `return` 语句之前加入一个判断，如果比较类型是字符串，则使用 JavaScript 提供的 `localeCompare()` 方法代替原来的比较方法，新重载 `sortData` 方法的代码如下：

```
if(Ext.data.Store){
    Ext.data.Store.prototype.sortData = function(f, direction){
        direction = direction || 'ASC';
        var st = this.fields.get(f).sortType;
        var fn = function(r1, r2){
            var v1 = st(r1.data[f]), v2 = st(r2.data[f]);
            if(typeof(v1) == "string"){
                // 用 localeCompare 比较汉字字符串，Firefox 与 IE 均支持
                return v1.localeCompare(v2);
            }
            // 添加结束
            else return v1 > v2 ? 1 : (v1 < v2 ? -1 : 0);
        };
        this.data.sort(direction, fn);
        if(this.snapshot && this.snapshot != this.data){
            this.snapshot.sort(direction, fn);
        }
    }
}
```



提示：①将以上代码放到汉化文件 `ext-lang-zh_CN.js` 里即可解决中文排序的 bug；②常规的汉字排序是根据 GB2313 国标字库里的内码顺序来排的，表现到用户界面就是常用汉字按照拼音排序。

在进行本地排序时，还有一个经常被忽略的地方：在定义 `Store` 的字段时，没有声明字段的类型。这样当排序的时候，应该按数字、日期等排序方式进行排序的字段，都按字符串排序了，从而产生错误。通过下面的例子便可较清楚地了解上述情况(例程 6.7 和图 6.11)。

例程 6.7

```
...
var store = new Ext.data.ArrayStore({
    remoteSort: false,
    fields: [{
        name: 'num1'
    }, {
        name: 'num2',
        type: 'int'
    }, {
        name: 'num3'
    }, {
        name: 'number',
        type: 'float'
    }, {
        name: 'datetime',
```



```
        type: 'date',  
        dateFormat: 'Y-m-d H:i:s'  
    }, {  
        name: 'file'  
    }, {  
        name: 'tips'  
    }  
    }  
});  
...
```



数字1	数字2	数字3	浮点数	时间	英文	中文
1	1	1	-10	2008-01-01 12:01:01	Atlantic Spad...	一
11	11	11	1	2008-02-16 12:01:02	Bat Ray.jpg	二
13	13	13	0	2008-03-15 12:01:03	Blue Angelfis...	三
2	2	2	10.01	2008-04-14 12:01:04	Bluehead Wvr...	四
23	23	23	200.9	2008-05-13 12:01:05	Cabezon.jpg	五
62	62	62	-200.01	2008-06-12 12:01:06	California Mo...	六

图 6.11 中文字段排序

例子 6.10 中, 第 1 到 3 列的数据是相同的, 不同的地方在第 1 列定义的数据是整数值, 第 2、3 列定义的数据是字符串值, 而字段定义中, 第 1、3 列的字段没有定义数据类型, 第 2 列则定位类型为 `int`。通过单击 `Grid` 的列标题进行排序, 可以看到第 1、2 列的排序是根据整型值的大小排序的, 而第 3 列则是根据字符串值排序的。为什么会这样呢? 原因就是, 第 1 列虽然没定义数据类型, 但是值的类型是整数, 所以比较大小的时候还是根据整型值来比较。第 2 列虽然数据定义为字符串, 但是数据类型定义是 `int`, 所以在比较时, 会先进行数据类型转换, 再进行比较, 因而比较结果是根据整型值的大小进行排序。第 3 列的数据定义为字符串, 而且数据类型没有定义为 `int`, 所以只能根据字符串规则进行排序。

在例子中还测试了浮点数、日期、字符串和中文等的本地排序情况。读者可以将浮点数、日期类型数据中的字段类型去掉, 看看排序结果是怎样的。

要保证本地排序的正确性, 最好的方法就是定义好字段的类型。这样做是为了返回的任意的数据格式, 都能够以正确的数据类型进行排序。

本节内容除了谈到 `Grid` 分页和排序, 还着墨了查询组件 `SearchField` 的用法。`Ext` 为我们提供了一个桥梁 `Ext.data.Store`, 通过它我们可以把任何格式的数据转化成 `Grid` 可以读取的形式。如此 `Store` 和 `ColumnModel` 两者构成了运行 `Grid` 的最基本条件, 下面就重点讲述 `Column` 可编程的内容。



6.2.4 列模型与单元格

与“行(Row)”对应着，一个表格应该有“列(Col)”的定义，即定义表头 `ColumnModel`。

1. 制定 Grid 的列模型

有些时候，需要在每行记录中显示一个链接(HTML Link，见图 6.12)，或根据数字的大小显示不同的颜色，又或者使用固定格式显示记录编号，又或者加上可爱的小图标，这就需要自定义单元格的显示格式。`Ext` 赋予我们 `ColumnModel`，轻松地来实现自定义单元格的显示格式，细化控制输出，而且灵活性较大。在 `Grid` 定义 `ColumnModel` 之后，如 `colModel: myColModel`，`myColModel` 就是一个 `ColumnModel` 实例，`myColModel` 需要自定义显示格式的列中加入参数 `renderer`，该参数指向一个函数，在显示时会以函数返回的数据作为显示数据，比如最简单的就是定义日期的显示格式，如下面的代码片段所示：

```
{header: "创建时间", width: 85, sortable: true, renderer:  
  Ext.util.Format.dateRenderer('Y-m-d'), dataIndex: 'createtime'}
```

其中我们一定要设置无误的就是 `dataIndex`，它是映射到 `Store` 的字段索引，与数据实体相匹配的，如果不正确，该列无法显示要求的数据。

电话	email	操作
1	1	修改 删除
ewewe	wew	修改 删除
232	232@abc.com	修改 删除

图 6.12 自定义 renderer

配置属性 `renderer` 指向为一个函数，`Ext.util.Format.dateRenderer()` 执行后返回的也正是一个函数。该函数有一个默认的 `value` 参数传入，`value` 参数的值最终输出为单元格 HTML。函数定义了单元格怎么来显示数据，执行后返回值就是显示的数据，但一般我们不执行这个 `renderer` 函数，只是定义它便可。



提示：这里的 `renderer` 后面怎么有参数传入？原因是这两 `Ext.util.Format.dateRenderer()` 返回的也是一个函数，准确地说，是函数类型的值，因此当然可以符合 `renderer` 的要求。`dateRenderer()` 返回的是这个处理日期格式的函数，不负责具体处理日期的功能，严格地说，`dateRenderer()` 只是产生这个“函数”。

如上面的代码，API 自带的 `Renderer` 可转换许多日期格式以满足 `renderer` 渲染器的要求，但万一还是不能满足要求的话，我们可以尝试写一个特定的 `renderer`。好像微软 SQL Server 查询日期类型形如“2008-10-08 14:26:01.0”，`Ext.util.Format.dateRenderer` 就没有产生相对应的转换函数。

遇到这类的日期，我们新定义这样一个 `renderDate()`：

```
function renderDate(value){
    var dt = new Date();
    dt = Date.parseDate(value, "Y-m-d h:i:s.0"); // 注意, 最后必须有个 0
    return dt.format('F d, Y'); // BOM 本身没有 Date.format 方法,
    // 这是 Ext 后来加上去的
}
```

ColumnModel 还有更多的配置参数让开发人员调配, 具体而言, 它们是关于字段标题、宽度、居中/居左/居右、可否排序等的配置项(API 都详尽列明)。例程 6.8 列出了怎样定义一个 Column Model 实例(并有更多的 renderer 函数示例), 截图见图 6.13。

例程 6.8

```
// 在第 1 列“编号”中, 其所属 leftPad() 的作用是产生固定长度为 5 位, 不足 5 位以 0 补齐的格
// 式的数字。该函数使用 Ext 字符串 leftPad 方法补 0, 然后将转换后的值返回
```

```
function leftPad(val) {
    return String.leftPad(val, 5, "0");
}
```

```
// 第 2 列连接演示了如何在单元格中显示链接
```

```
function linker(val) {
    /*
```

注意, 第 2 列使用了 JSON 格式的数据, 如 “{text: 'Ext', url: 'http://extjs.com'}”, 标签 text 内的值是单元格显示的数据, 而标签 url 的值是链接地址。在函数中先判断原始值是否为对象, 如果是则使用 text 和 url 组合一个 HTML 标记并返回。在 HTML 标记中定义了两个样式, 其目的是让 HTML 标记显示时能填满单元格, 这样当鼠标移动到该单元格空白处时, 也能显示提示信息。提示信息由 HTML 标记的 title 属性来实现

```
*/
    if (typeof val == 'object') {
        return '<a style="display:table;width:100%;" title="' + val.url
            + '" target="_blank" href="' + val.url + '">' + val.text + '</a>'
    }
    return val;
}
```

```
var grid = new Ext.grid.GridPanel({
    height: 350,
    width: 800,
    store: store,
    title: '自定义单元格的显示格式',
    frame: true,
    columns: [
        {
            header: '编号',
            width: 80,
            sortable: true,
            renderer: leftPad,
            dataIndex: 'id'
        },
        { header: "链接", width: 75, sortable: true, renderer: linker,
            dataIndex: 'linker' },
        ...
    ]
});
```

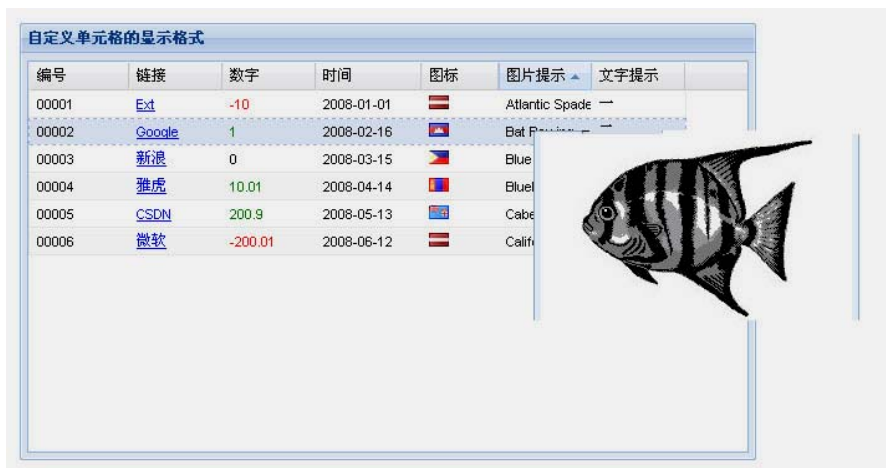

$$\left\{ \begin{array}{l}] , \\ \dots \\ \end{array} \right\}$$


图 6.13 QuickTip 灵活作为图片提示用

是否留意到，Column 列模型的第一个列的定义“编号”，其 `renderer` 指定的渲染函数就没有常数？实际上在 Ext 许多渲染 UI 地方都曾出现这种规律，也就是函数作为“渲染器”，控制原始的 `data` 怎么输出为符合 HTML 显示要求的这么一个过程，而且过程中肯定有 `return` 语句。我们遇到渲染器的时候，不应执行它(即 `function()` 后面不加上代表“执行”的括号 `()`)，而是定义后分配即可。以下代码是例程 6.11 中剩余的渲染器(附有完整的注释)。

例程 6.8 第 65 行至结束

```
// 第 3 列“数字”则演示了根据数值的大小使用不同颜色作为显示文字的颜色，
// 该单元格中负数显示为红色，0 为黑色，正数为绿色
function num(val) {
    if (val > 0) { // 函数中控制颜色的显示是通过 HTML 标记 SPAN 的样式实现的
        return '<span style="color:green;">' + val + '</span>';
    } else if (val < 0) {
        return '<span style="color:red;">' + val + '</span>';
    }
    return val;
}

// 第 4 列“时间”使用了 Ext 的日期格式函数控制日期的显示格式。在这里要特别注意的是日期的
// 值格式与 store 字段中定义的格式一定要相同，不然 Grid 会显示为空白值。例如例子中的日期值为
// “2008-01-01 12:01:01”，则定义日期的格式必须为“Y-m-d H:i:s”
var showDate = Ext.util.Format.dateRenderer('Y-m-d');

// 第 5 列“图标”演示了如何在单元格中显示小图片
function icon(val) {
    // 函数将包含图片文件名的原始值转换为 HTML 标记 IMG，就可实现在单元格内显示图片了
    return '';
}
```

// 第 6 列“图片提示”演示了通过 Ext 的 QuickTip 显示图片提示

```
function qtips(val) {
```

```
/*
```

要在 HTML 标记中使用 QuickTip 显示提示，只要在标记中加入一个属性 qtip 就行了。在函数中可以看到将一个 IMG 标记作为 qtip 的显示内容。不过，别忘了在 OnReady 函数中对 QuickTip 进行初始化。标记 SPAN 也定义了两个样式，其作用与第 2 列的一样

```
*/
```

```
    return '<span style="display:table;width:100%;" qtip=\'\>' + val + '</span>'
```

```
}
```

// 第 7 列“文字提示”使用了 HTML 标记的 title 属性来显示提示，与第 2 列“链接”的原理是一样的，只是该列使用了 SPAN 标记

```
function tips(val) {
```

```
    if (typeof val == 'object') {
```

```
        return '<span style="display:table;width:100%;" title="' +  
            + val.tips + '">' + val.text + '</span>'
```

```
    }
```

```
    return val
```

```
}
```

如果我们不想只限定当前“列”的值，应用其他字段的值也是允许的。如下例某片段，渲染器第 3 个参数 record，其属性 data 便代表全体的实体记录，获取 LastName 的值与 value 值一起联合输出。对了，还有第 2 个参数那是什么？我们不是卖关子，而是希望感兴趣的读者可趁机实践一下，p 自然手到擒来……

```
function renderName(value, p, record){  
    return String.format("{0}?{1}", value, record.data['LastName']);  
}
```

当前简单地演示了几种自定义显示格式的单元格。只要灵活使用 HTML 标记，还可以实现更多的自定义格式，如怎么为每行记录都添加一个操作列这类的问题都可应用该方法来实现(见图 6.12)。因为始终都是转化为 HTML 的关系，如法炮制即可实现我们想要的 UI。另外，社区提供的 RowAction 这个扩展，其目的也与本用途一致。



提示：现在遇到一个问题，就是 Grid 不能很好填充布局，应该怎样修正较好看？指定 Ext.Container 配置项 layout 为“fit”的布局就可以很好地自适应容器的拉伸了，这对于任何类型的容器都是有效的。如果 Grid Column 自适应宽度不能解决，还有一个来自坊间的办法：固定 GridPanel 的 width，比如 600px，按照比例配置每一个 Column 的宽度，比如 300,200,100，最后设置 autoExpandColumn，然后在 GridPanel 外面嵌套一个 Panel layout:‘fit’。

到现在我们可以总结一下，在 Column 中我们可以做些什么——排序；列隐藏；自定义单元格的编辑器 UI；自定义渲染结果；数据绑定(dataIndex)；插件(分组、摘要、折合面板等)。当然还可以发挥社区的功效，寻找更多符合项目需求的插件/扩展。



2. 可编辑的单元格

API 提供了 GridPanel、EditorGridPanel 和 PropertyGrid 三种 Grid。GridPanel 主要用于一般性质的 Grid 显示；PropertyGrid 是 Grid 的变种，用于类似 IDE 分配参数的 Key&Value 界面；EditorGridPanel 在 Ext 是用得比较多的一种 UI 组件，实现编辑单元格(Cell)数据也比较方便，带来较好的用户体验。官方附带的 Examples/grid/edit-grid.html 例子就是一个不错的教学示范，它本身带有以下几项功能的演示：

- 文本型(重用 Ext.form.TextField)的单元格编辑控件。
- 下拉组合框“多选”(重用 Ext.form.ComboBox)的单元格编辑控件。
- 数字型(重用 Ext.form.NumberField)的单元格编辑控件。
- 日期型(重用 Ext.form.DateField)的单元格编辑控件。
- 布尔型“Checkbox 打勾”控件的特殊列(new Ext.grid.CheckColumn())，需插件 Ext.grid.CheckColumn 支持。
- 新增行记录“Add Plant”。



提示：所谓可编辑的单元格，就是在设置 ColumnModel 列模型中，提供一项“editor”的配置项属性，让用户选定哪个编辑器。Ext 默认带有的编辑器很多是原来 Ext.form.*的控件。

官方例子提供这一批丰富的 UI 结合 EditorGrid，却没有进一步提供保存功能 afterEdit 的演示。我们在此例子的基础上为大家介绍该功能的实现。一般的操作是编辑完一个单元格自动保存数据，再刷新 Grid。但这对于频繁修改数据的用户来说会带来性能上的损失和用户体验的迟滞。现介绍一种完整提交修改和删除数据的方法，这样只需提交一次数据给服务器。

例程 6.9(见图 6.14)提交 EditorGrid 编辑过的数据主要是围绕 Store 对象展开的。无论是新增加的记录还是修改现有的数据，只要是从 Grid 编辑过的，都应被视为提交的记录集收集起来，然后传送到后台逻辑保存在数据库中。



图 6.14 B/S 多层模型

要得到这些数据,可执行 `Stroe.getModifiedRecords()` 来获取(返回 `Record` 对象构成的数组)。注意新增加记录之余必须同步更新 `Store` 里面的记录,如例程“Add Plant”按钮中的 `store.insert(0,p);`语句表示将用户新写入的记录插入 `Store`。

例程 6.9

```
var saveBtn = new Ext.Button({
    text: '保存',
    handler: function(saveBtn, event){
        var storeObj = this; // 注意下面指定了 storeObj 作用域
        var modifiedRecords = storeObj.getModifiedRecords();
        var submitRecords = [];

        if(modifiedRecords.length == 0){
            alert('没有发现修改过的记录, 没有内容提交! ');
            return;
        }
        for(var i=0, len=modifiedRecords.length; i<len; i++){
            submitRecords.push(
                storeObj.isWithAllFields === true
                ? modifiedRecords[i].data
                : modifiedRecords[i].modified
            ); // 获取 Record 之中的实体数据
        }
        debugger
        saveBtn.disable();
        if(confirm("您确定要进行该操作?")==true){
            // 正在保存改变, 请等待...
            Ext.Ajax.request({
                url: 'plants.xml'
                ,params:{
                    submitData: Ext.util.JSON.encode(submitRecords)
                }
                ,success: function() {
                    alert("保存已成功。");
                    storeObj.commitChanges();
                    saveBtn.enable();
                },
                failure: function() {
                    alert('你的改变未能正常保存成功!');
                    saveBtn.enable();
                }
                ,scope: storeObj
            });
        }else{
            saveBtn.enable();
        }
    }
    ,scope:store
});
var undoBtn = new Ext.Button({
```



```
text: '撤消编辑',  
handler: function(){  
    var storeObj = this;  
    if(confirm("您确定要进行该操作?")==true){  
        storeObj.rejectChanges();  
    }  
}  
,scope: store  
});
```

Ajax 提供记录成功后(success/failure 处理函数对应成功或不成功的过程),还应该将修改过的记录置为空,如果不清空,则修改过的数据会一直保留,即 UI 上反映就是右上角一直标识着“深红色的箭头”。

我们应该更新一下 Store, 去掉箭头的标识,告诉用户属于上一次编辑的行为已经完毕了,不然用户很容易会对哪一次跟哪一次编辑的记录、是否保存过而感到紊乱。

执行 Store 的 storeObj.commitChanges();即可清空修改过的标识,而且应该是后台数据库执行完毕后才可执行,原因是,如果在后台更新时失败怎么办?所以下面这样好一些:

```
success: function(form, action) {  
    Ext.MessageBox.alert("保存已成功。");  
    dataStore.commitChanges();  
}
```

编辑 Grid 过后, 如果不想做出任何修改, 我们可以单击“撤消”按钮试试, 其实就是通过 storeObj.rejectChanges()恢复编辑前的状态。

6.2.5 多层表头

6.2.6 使用拖放行为

6.2.7 PropertyGrid 扩展简介

6.2.8 使用 DataView 组件

6.3 表单组件 Form

前面已经提到过表单对象受上层容器的控制、协调布局,属于页面布局的一部分。本节会侧重谈表单数据绑定的具体操作。

6.4 列表组件 ComboBox

6.5 树状组件 Tree

树是 UI 界面编程中经常使用的一种显示结构化的，具有继承关系的数据的方法。`Ext.tree.*`内建了对树形控件中节点的编辑、排序、过滤，拖拽等功能的支持，同时提供了一个完善的事件侦听和触发机制。综合使用树和 `Ext` 中的其他组件，可以构建出功能强大的复合 UI 组件。

第9章 大型 UI 控件

Ext 提供了纯客户端的“控件”供用户选择使用，不过在讨论 Ext 的时候往往倾向于使用更专注界面构成方式的“组件”一词代替，当然继续使用“控件”一词去理解也未尝不可；至于 widget——“小部件”也具有相同意义，在“企业级”应用的范围外小部件 widget 一词更流行。这些组件并非都是网页原生的控件对象，而是利用 HTML/CSS 的资源复合而成的，在某种意义上也体现了 Ajax 是基于通用标准的技术方案而实现的。

在这里，我们将编写一个大型的 Ext 控件，让读者知道各个控件的使用方式与作用，并了解组件在整合应用时可能遇到的各种问题。希望读者在看完本章后能对 Ext 有一个更加完整的认识。



关键词

- UI 控件/组件定义
- 组合 UI 控件
- Ext.Direct
- 自定义控件



9.1 Edk 简介

该大型 UI 控件从属于 Edk 框架的一部分。Edk 是一个 ECMAScript 服务端的框架，用于实现小型项目的 DSL 平台。Edk 已共享在 Google Code 的开源服务上，如图 9.1 所示，访问地址是：

<http://code.google.com/p/naturaljs/>



图 9.1 开源 JS 库：Edk

Deepblog 是基于 Edk 框架的二次开发程序，现用于本章节的教学。Deepblog 保持不断更新，访问地址是：

<http://code.google.com/p/deepcms/>

该大型控件和 Edk 会保持不断更新，但基本架构不变，与本章介绍的情况吻合。

9.1.1 项目起步

新建一份 HTML 文档 admin.htm，并引入 Ext 库文件，有 ext-base.js/ext-all.js，选择 debug 的版本。当前的是 Ext 3.1 版本。样式先行引入，此时的 HTML 如下：

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>DeepBlog 系统</title>
    <!--引入 Ext 库文件，有 ext-base.js/ext-all.js，选择 debug 的版本-->
    <script src="http://localhost/Ext3.1/adaptor/ext/ext-base.js">
    </script>
    <script src="http://localhost/Ext3.1/ext-all-debug.js"></script>
    <!--End Block-->
  </head>
  <body>
  </body>
</html>
```

必须引入 Ext 扩展、插件库的文件——<http://localhost/Edk3.1/Ext-ux/Ext.ux.js>。



提示: 为清晰地说明 Ext 及组件应用, 我们尽可能将客户端/服务端的逻辑管治方式划分开来, 故造成 admin.htm 为纯静态的 HTML 文件, 而 Service 入口就在“service/default.asp”或“service/blog.asp”。让 Ajax 请求 Service 入口的业务系统然后返回, 就可以得到纯数据的部分了(JSON)。若在页面加入动态内容有困难? 可以采取动态脚本的方法解决, 下面将会详细介绍。

开始引入 Edk 库文件。Edk 的目录结构分为两部分, core(核心函数库)和 client(绘制 UI 的扩展库): 其中 core 中又有 core.js 和 lib.js, lib.js 文件提供 JSON 编码、Edk.Template.*模板、Edk.Event.*事件的功能模块(独立的方案), 无论浏览器还是服务端的 JS 皆可运行 core.js 和 lib.js; 而 client 目录下是 Edk 对 Ext 的扩展, 也就是本章“大型 Ext 控件”的主题所在部分。client 各子模块的作用如下。

- **Edk.grid(\$\$.grid):** 实现了基本 CRUD 功能及界面; 支持工具条的自动禁用/启用(toggleButtons); 提供默认的列模型。
- **Edk.tree(\$\$.tree):** 实现了分类的树节点。支持右键菜单、隐藏固定节点的功能。该类没有从 Ext.TreePanel 继承, 而是直接从 Ext.tree.AsyncTreeNode 继承获得新的节点。
- **Ext.form(\$\$.form):** 实现了普通的 HTML 表单控制。Ext.form 支持服务端生成的 FORM 标签, 但不需要服务端的生成控件的细节, 仍然是通过 Ext/Edk 提供控件支持。该类利用了 Ext.Observale 新增加了特定的事件。
- **Edk.attachment(\$\$.attachment):** 实现了上传文件。支持图片格式的预览图、单文件/多文件、无刷新上传。

在 admin.htm 页面中引入 Edk 文件:

```
<!--引入 UI 库文件-->
<link type="text/css" rel="stylesheet"
      href="http://static.ajaxjs.com/UI/style/common.css" />
<link type="text/css" rel="stylesheet"
      href="http://static.ajaxjs.com/UI/style/Edk-x.css" />
<script src="http://localhost/Edk/client/Edk.grid.js"></script>
<script src="http://localhost/Edk/client/Edk.tree.js"></script>
<script src="http://localhost/Edk/client/Edk.form.js"></script>
<script src="http://localhost/Edk/client/Edk.attachment.js"></script>
```

各个模块聚合在一起, 便构成了这个“大型 Ext 控件”的基本模块。其中目的之一是便于以后对代码进行复用, 但位于此层的代码当前尚不能直接地使用它们, 还需要在它们的基础上继续扩展, 写出实体类(UI for Domain Object)。从特定的模式来看, 该实体类是把这些解耦的各个类(Grid、Tree、Form、Action)内聚在一起, 整合之后形成成为该实体类所服务的 UI 机制。不同域的对象便有不同业务对象的 UI 类。当前这个实体就是“博客文章”。blog.js 保存了“博客 UI 类”\$\$.blogAdminUI 的源码:

```
<!--引入博客库文件-->
<script src="blog.js"></script>
```



位于`$.blogAdminUI`层的代码当前尚不能直接地使用,还需要在它们的基础上继续扩展,写出针对应用程序最终效果的子类,才算界面启动的类:`deepblog.blogPanel`,即文件“`asset/deepblog.js`”引入的部分。

然而,除了`deepblog.js`和`DeepBlog`项目的样式,`<head>`标签中还引入动态 JavaScript,由程序运行时决定的逻辑过程。脚本地址“`service/?action=getREMOTING_API`”也是返回 JavaScript 代码,涉及到`Ext.Direct`的 API 描述,由服务端生成,提供不同的业务信息。脚本地址“`service/?action=getConfig`”则是该程序的参数列表或配置信息,例如博客名称、博客主人等的信息。因为例子中采用了`Client/Server`分离的模式,所以借助于`<script>`标签返回动态内容:

```
<link type="text/css" rel="stylesheet" href="asset/images/deepblog.css" />
<link type="text/css" rel="stylesheet" href="asset/welcomePage/news.css" />
<script src="service/?action=getREMOTING_API"></script>
<script src="service/?action=getConfig"></script>
<script src="asset/deepblog.js"></script>
```

最后`admin.htm`加入若干的 HTML,用户可自行修改为项目特定的显示界面:

```
<noscript>
    Please enable JavaScript!
</noscript>

<!--顶部区域-->

<table id="navBar" width="100%">
    <tr>
        <td>DeepBlog</td>
        <td class="right" align="right">
            <a href="http://" target="_blank" id="edk_qiye_link"></a>企业专属版
        </td>
    </tr>
</table>

<!--END 顶部区域-->
```



提示: 这里的`id="navBar"`会是`Ext`布局中`North`的区域(应用`Ext.BoxComponent`特例)。不需要特别的脚本编程,不像复杂的`Ext`组件那样控制 HTML,`Ext`通过`id`或`CSS Selector`定位也可以直接整合页面中普通的 HTML。

HTML 纯静态的部分至此介绍完毕了。`default.htm`中虽然看到的是`<body>...</body>`的若干行代码,但我们明白,经过浏览器的渲染就会产生复杂的 HTML 结构。整个界面如图 9.2 所示(图 9.2 是一般使用模式,单个`blogAdmin`实例;图 9.3 是多个`blogAdmin`实例)。操控、配置 HTML 结构的便是 JavaScript 部分。

下面我们将进入源码的分析与理解。

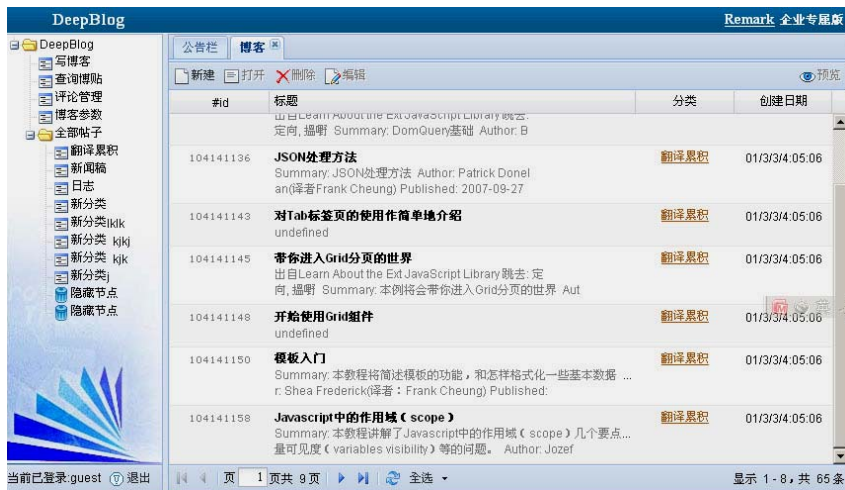


图 9.2 Deepblog 默认界面(viewport 为容器)

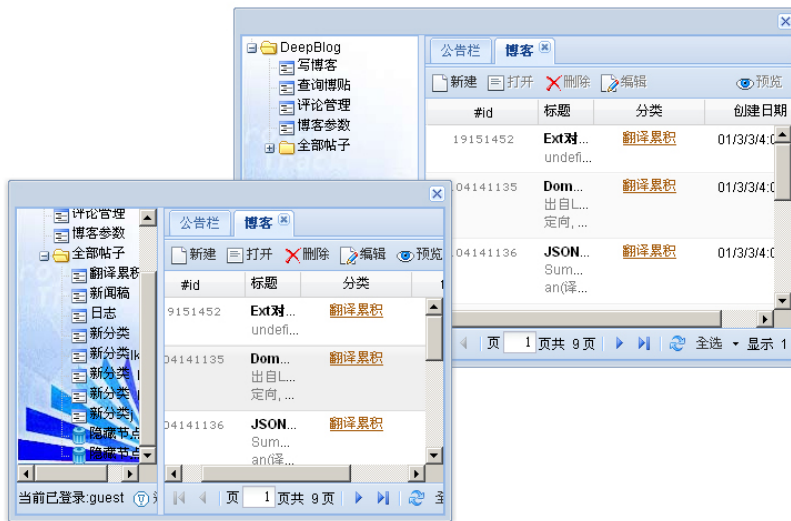


图 9.3 多个 blogAdmin 实例(window 为容器)

9.1.2 顶层源码概览

创建复合型 Ajax 前端控件的关键之处仍然离不开 Ext 组件的合理使用。针对该控件的编码过程有下列的目标。

- 适用性、通用性：为一般记录提供典型的 CRUD 操作服务界面。有记录分类功能。
- 可复用性：编码者尽量精简代码，以提高代码的复用率；例如我们应该使用 Ext.Panel 作为容器，就不使用 Ext.Window，因为 Panel 可以融入 Window 中，而比 Window 更通用。
- 规范性、示范性：虽然 JS 语言风格自由，变化多端，但作为示范代码，实在不宜有

过多的技巧性、生僻的代码。

- 可扩展性：除了可作为示范用途，还具备投入实际应用的能力。这就要求该扩展有一定的可扩展性。对此我们采取的策略模式，除了沿用 Ext 的生存周期的方法+自定义事件耦合，还有接下来要介绍的“1+2”模式。当然前提是我们对前面几章关于 Ext 组件的用法已经有初步的认识。



提示：为方便代码的书写，Edk 中设定 \$\$ = Edk，两者是等价的。

文件夹 Edk/client(即 Edk.client.*命名空间下)是 Edk 库的 UI 部分。当前的版本有 Edk.grid、Edk.tree、Ext.form 和 Edk.attachment 四个大类。每一个类便是一个文件。每个类约定写法如下：

```
;(function(){
    // 私有的变量
    // 注意这里的变量是静态的 Static
    var fooObj, barObj, ...;
    // 私有的方法
    function foo(){}
    function bar(){}
    // 或 var foo = function(){}; var bar = function(){};
    // 类“主体”
    Edk.grid = Ext.extend(...{
        // 开始实例方法
        foo: function(arg1){
            this.fooObj = arg1; // 使用实例变量
            // 限该闭包内使用下列的方法、变量
            foo(fooObj); // 使用静态变量
        }
        ,bar : ...
    });
    // 公开的静态方法
    Edk.grid.foo = function(){...}
    Edk.grid.bar = function(){...}
})();
```

我们看看约定中有什么内容。每份源码中，最外层的是一个匿名函数，目的是在解释器中产生一个“域(scope)”，“域”是为当前类所享用的。在域场中，允许定义私有(private)/公开(public)的成员，原理在于是否有“var”作用的参与。但使用关键字“var”，表示该成员的内存空间分配在当前域场下面，即最近的 function 作用域内。反之，如果没有使用关键字“var”，则表示成员的内存空间分配在“全局空间”下面，脱离于当前的域，如 Edk.gird 便是在全局变量上分配了一个 grid 的空间。界定函数(方法)是否为私有的成员，也可以用“var”表示，如上述注释中的 var foo = function(){}; var bar = function(){};。

全局变量 Edk 的使用是 JavaScript“逃逸变量”的特性表现之一。我们知道本来 JavaScript 没有所谓的“静态方法”或者“实例方法”，而在上面的模板中，我们却“实现”了。静态

方法任何时间都可以调用,但实例方法必须经过实例化的过程,在类的特例中所使用,要访问实例,就必须通过“this”对象指针访问,有上下文的限制。如果访问静态的变量就不需要this,利用“逃逸变量”就可以访问上层或上层以外域的变量(若执行“var 相同变量名”的操作,则覆盖原有已存在变量,等于弃用原有域的成员)。这种变量我们称作“静态变量”。

无论是私有的性质还是公开的性质,一旦静态变量的值被修改了,就固定下来。JavaScript中没有finally、constant关键字。



提示:以上内容尚未能“贴近现场”般地叙述清楚,但相信,下面的9.2节“如何重用Ext组件”的内容可以不知不觉地告诉大家,这种编码上的策略是如何真实地被应用,有效地配合我们开发。

9.1.3 数据通信机制 Ext.Direct

DeepBlog基本做到了前后台的彻底分离,以呈现更清晰的架构。DeepBlog的Remote通讯全部基于Ext.Direct+微软Classic ASP的实现。使用Ext.Direct的一个前提条件是准备好Ext.Direct后台服务包(Pack),依照服务端的不同来决定服务包是什么语言的。当今流行一种的概念ORM。一般来说,ORM解决了OO语言与关系型数据库的阻抗不匹配,减少SQL语句的使用,能够方便地把数据和操作封装起来,映射一个实际业务中的对象。至于服务端方面,Ext.Direct ServrSide包的主要工作就是把这些OO方法相关反射的信息告诉前端的Ext。DeepBlog后台是JScript服务端编程,也就是运行在服务端的JavaScript代码。

Ext.Direct的架构属于“供应器(provider)”的架构,一个或多个供应器负责将数据传输到服务器上。当前有几种关键的供应器:

- JsonProvider——简易的JSON操作。
- PollingProvider——轮询供应器,用于重复的请求。
- RemotingProvider——向客户端暴露了服务端的供应器。

当前DeepBlog所采用的是RemotingProvider供应器。每当前端页面请求的地址是<script src=“service/?action=getREMOTING_API”></script>的时候,此刻,服务端方面就会调用全局函数getREMOTING_API()返回服务端输出的供应器被序列化后(Edk.JSON.encode(REMOTING_API))的字符串。这段字符串中包含了url通讯地址、Remote的类型、请求id(如果你没有用全局变量保存着这个供应器,那么你就应该分配一个id来记住这个供应器,以便日后寻找)和actions执行动作。

actions对象下面的属性每个都代表着对应服务端的业务类,类的数量至少有一个或多个。当前,actions[‘blogService’]表示只有一个业务类是blogService。Ext方面会遍历actions的hash得到“blogService”字符串,并且获取actions[‘blogService’]的值,所指向的值就是一系列Object Literal对象类型构成的数组。该Object Literal必须包含暴露给Ext的方法名称以及参数个数(个数一定要是数字类型,而不能是字符串)。有了这两个已知条件,Ext前端方面就能够以此进行基于Ext.Direct的通讯。



```
getREMTING_API = function(){
    // this file Name!
    var urlStr = "service/blog.asp";

    var REMTING_API = {
        "url"      : urlStr
        ,"type"    : "remoting"
        ,'id'      : 'blog'
        ,actions   : {}
        ,dataSource : [...] // 暂略
    };
    REMTING_API.actions['blogService'] = [
    {
        "name": "getEntity",
        "len": 1
    }
    ,{
        "name": "viewPost",
        "len": 1
    }
    ,{
        "name": "editPost",
        "len": 1
    }
    ,{
        "name": "loadCreate",
        "len": 0
    }
    ,{
        "name": "getCatalogAsTreeNode",
        "len": 1
    }
    ,{
        "name": "create",
        "len": 1
    }
    ,{
        "name": "update",
        "len": 1
    }
    ,{
        "name": "del",
        "len": 1
    }
    ,{
        "name": "setState",
        "len": 1
    }
    ,{
        "name": "renameCatalog",
```

```
        "len": 2
      }
    ], {
      "name": "addCatalog",
      "len": 1
    }
  ];
  Response.write(
    'Ext.Direct.addProvider({0});'.format(Edk.JSON.encode(REMOTING_API));
  )
}
```

供应器传送到客户端之后, 本身不能直接的使用, 应该通过 `Ext.Direct.add()` 加入, 即最后一句 “`Response.write('Ext.Direct.addProvider({0});'...`” 的作用。



提示: 关于供应器的用法细节, 可参见文档详细的介绍。这里说说比较实用的配置项 `namespace`, 它指定了远程供应器的命名空间(默认为浏览器的全局空间 `window`)。完全命名这个空间, 或者指定一个 `String` 执行隐式命名。

另外, 供应器的请求过程中加入了特殊的字段。因为既然我们打算动态生成 `Ext.Store` 的描述字段(用于配置 `Record` 的 `Schema`), 而且同样是前台所接纳的 `JSON` 结构, 则可以在 `Ext.Direct` 的 API 输出中一并返回 `Ext.Store` 字段。我们对 API 额外的描述是人为扩展的性质, 另外还可以加入其他需求的字段信息。这里具体就是加入 `REMOTING_API.dataSource` 属性:

```
var REMOTING_API = {
  ...
  ,dataSource : [
    {
      totalProperty: "totalNum",
      root: "results",
      idProperty: "ID"
    }
  ],
  [
    {name: 'Title',          type: 'string'}
    , {name: 'ID',           type: 'int'}
    , {name: 'ClassID',      type: 'int'}
    , {name: 'Brief',        type: 'string'}
    , {name: 'createDate'}
    , {name: 'catalogTitle', type: 'string'}
  ]
];
```



提示: 在上述的演示代码中, 包括 API 方法输出和 `Store` 配置输出, 都是静态的演示代码, 视实际情形, 可以转变为动态的。

然后客户端方面的准备, 就是要提取 `Store` 的信息, 随时供 `Grid` 注入 `Store` 时所用。详

见 `blog.js` 中的 `$.blogAdminUI.getDataSource()` 就是 Store 的构建过程:

```
,getDataSource: function(){
    var dataSoureCfg = Ext.Direct.getProvider('blog').dataSource; // API 已通
    // 过<script>...</script>引入, 读取其 dataSource 属性, 就是 Store 的配置内容
    // 建立 Store
    var store = new Ext.data.Store({
        proxy : new Ext.data.DirectProxy({
            // 传入 API 方法, 注意这就是方法 function 的引用
            directFn: blogService.getEntity
        })
        ,reader: new Ext.data.JsonReader(dataSoureCfg[0],
            dataSoureCfg[1]) // 定义在数组第一、第二个元素上
    });
    return store;
}
```

`blogService.getEntity` 是 JavaScript 方法, 但不能立刻被执行。它的作用是作为“方法引用”传入到 Ext 数据驱动的服务层。`Ext.data.DirectProxy` 是 Ext 3.0 新增的 Proxy 类, `directFn: blogService.getEntity` 就是 ExtDirect 与 Provider 具体哪一个业务方法的对接口, `directFn` 值的类型必须是 Function 类型。引入 ExtDirect 之后, url 地址表面上消失了, Store/TreeLoader 乃至 Ajax 的 XHR 都看不见通往后台“`http://地址`”, 而真实的就是 Provider 在其中已经定义过了, 且一般而言此处是整个单页面程序的统一、唯一的 URL 入口。

参见 `Edk.tree.js` 里的 `constructor` 构造器(第 6 行), 其中的 `TreeLoader` 也是分配对应的业务方法:

```
this.loader = new Ext.tree.TreeLoader({
    directFn: blogService.getCatalogAsTreeNode
});
```

以上的重点是“方法(函数)的引用”, 若要讨论方法的参数是怎样传入的, 应当结合 `Store.baseParams/TreeLoaer.baseParams` 讨论, 它们正是控制“参数”的地方。上面的 Grid 例子中, Store 执行 `blogService.getEntity({start:0, limit:8}, callback)`, 服务端的“`getEntity()`”函数会送入一个包含两个属性(0 和 8)分页对象, `getEntity()` 函数就返回 Grid 记录集合, 最后变为 `DirectProxy` 的 `result` 变量, 渲染结果。此时可以认为服务端接洽此请求的角色是“路由器(Router)”。由于 `DirectProxy/TreeLoader` 乃异步 Remote 请求的缘故, 执行该方法的时刻程序员在语法上一般是不可见。不过借助于 Firebug 等的测试工具, 仍然可以观察其详细的往返过程, 如图 9.4 所示。

HTTP Raw POST 是 Ext.Direct 进行请求的类型之一, 其中 POST 体的数据结构可参考 7.9.4 小节“路由器”的说明。返回的 Response 依据 UI 绑定的结构不同而有所区别, 如 `TreeLoader` 的格式与 Grid 的又有不同, 甚至返回 HTML 文本, 又要另作处理, ——但有一点可以肯定的是, POST 请求的结构均是一致的。一般采用 `DirectProxy/TreeLoader` 的话就已经有内置的 Callback 回调处理手段, 换言之就是将得到数据渲染行/单元格, 或者渲染树节点。如果供应

器的 API 方法是直接执行的,那么方法最后的一个参数必定是一个 Function 类型的回调函数(或居次二,可能有函数的作用域要指定)。

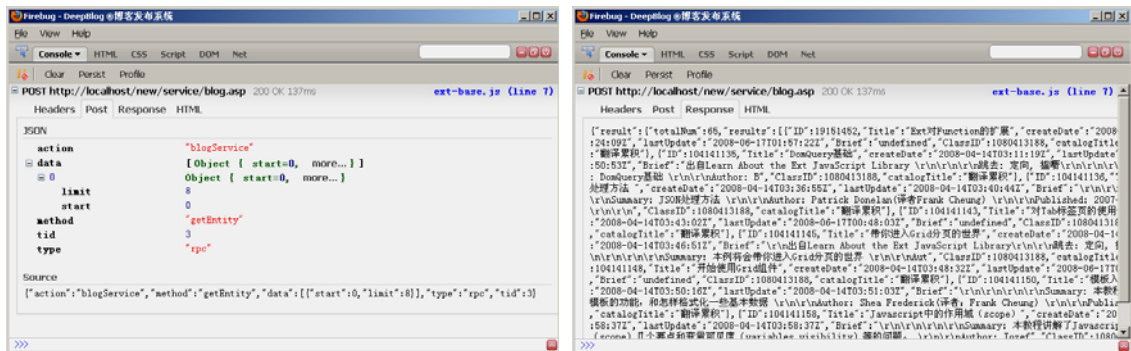


图 9.4 刷新 Grid 时浏览器发出的请求动作

如在 TabPanel 中双击一笔记录,随即打开“查看记录”的面板,加入到 TabPanel 中,此过程如下(保存在 Edk.grid.js/var defaultActions.read 成员,这是一个 Action 的配置对象):

```
var id = itemObj.id;
var viewPanel = new Ext.Panel({
    id      : "viewPanel_" + id
    ,title   : itemObj.data['Title']
    ,closable : true
    ,autoScroll: true
});
viewPanel.recordId = id;

this.add(viewPanel);
// 直接 API 方法, {ID: id}是参数, viewPanel.body.update 是回调函数,
// viewPanel.body 是回调函数的作用域
blogService.viewPost({ID: id}, viewPanel.body.update, viewPanel.body);
// 查看模板
```

blogService.viewPost({ID: id}, viewPanel.body.update, viewPanel.body)就是直接执行 Provider 定义的方法,其参数的多少来自于 Provider 的{"name": "viewPost","len": 1}。此时的我们不再使用方法的引用,而是执行了 viewPost 方法。

如果要快速测试一下 API 方法,可以让 Firebug 方法 console.info 作为回调的函数,即 blogService.viewPost({ID: id}, console.info)。当前 viewPanel.body.update 的动作就是更新面板的内容(viewPanel.body.el.innerHTML=(String)result)。



提示: 注意返回纯 HTML 字符串的时候,也必须符合 Ext.Direct 格式,成为可辨认的结果,而不能直接返回 HTML 文本。可参考本例的“viewPost”。

另外,设置 Provider 的 enableBuffer 为 true 或 false 决定是否捆绑多个方法调用在一起。



如果指定一个数字表示,就等待形成一次批调用的机会(默认为 10 毫秒)。多个调用按一定的时候聚集在一起然后统一在一次请求中发出,就可以优化程序了,因为减少了服务端之间的往返次数。举一个 Car 的例子如下:

```
Example.Car.start();  
Example.Car.go(80);  
Example.Car.stop();  
// 业务方法转化到客户端上
```

请求:

```
[  
  {"action": "Car", "method": "start", "data": null, "type": "rpc", "tid": 4},  
  {"action": "Car", "method": "go", "data": [80], "type": "rpc", "tid": 5},  
  {"action": "Car", "method": "stop", "data": null, "type": "rpc", "tid": 6}  
]
```

响应:

```
[  
  {"type": "rpc", "tid": 4, "action": "Car", "method": "start",  
    "result": "Started"},  
  {"type": "rpc", "tid": 5, "action": "Car", "method": "go",  
    "result": "The speed is 80"},  
  {"type": "rpc", "tid": 6, "action": "Car", "method": "stop",  
    "result": "Stopped"}  
]
```

无论使用 Ext.Direct 与否,都不影响 Edk UI 的对象模型,因为 Ext 的 UI 与数据模型是彻底分离的。Ext.Direct 是 Ext.Data.*完善升级,使得我们允许以更优雅的姿态填补客户端与服务端之间的鸿沟,简单可行并使应用具有弹性。

9.2 如何复用 Ext 组件

假设当前博客业务需要用到 Grid、Tree、Form、Tab 四大组件,还有动作 Action,最终目的就是封装它们,整合到一块,使之成为独立的 UI 单元。



提示: Aciton 就是排除那些 UI 的非可视部分,把属于 UI 的行为、动作部分抽象而成的逻辑。Action 对应 Ext 中的基类是 Ext.Action。后面在 9.3 节中将会谈到。

如何合理重用/复用 Ext 的 UI 组件对象,始终是我们关心的问题,亦是重点所在。首先,Ext.extend()方法为我们展现了 JavaScript object-oriented 的宏伟画卷;然后,我们可以使用更多的“设计模式”在其中。众所周知,“设计模式”属于开发阶段中比较高级的内容,但也不是不可以用在设计 Ext 组件中的,——熟悉该方面内容的高级用户其实可以在 Ext 画卷

中纵横驰骋。不过，在此高低、强弱之间，本小节尝试利用 Ext 继承架构，糅合另外两种本来固有的方法，形成一种新的代码重用模式。对于这种 Edk UI 的继承模式，我们可形象地归纳为“1+2”模式。

9.2.1 “1+2”之一

“1+2”中第一种大模式，就是使用 Ext.extend()。

如图 9.5 水平方向，deepblog.blogPanel 继承于 Edk.blogAdminUI，Edk.blogAdminUI 继承于 Ext.Panel。作为一个标准 Ext 组件，blogAdminUI 可以插入到任意一个 Ext 容器中去。实现该特性的就是来自于 Ext.Panel 的继承关系。

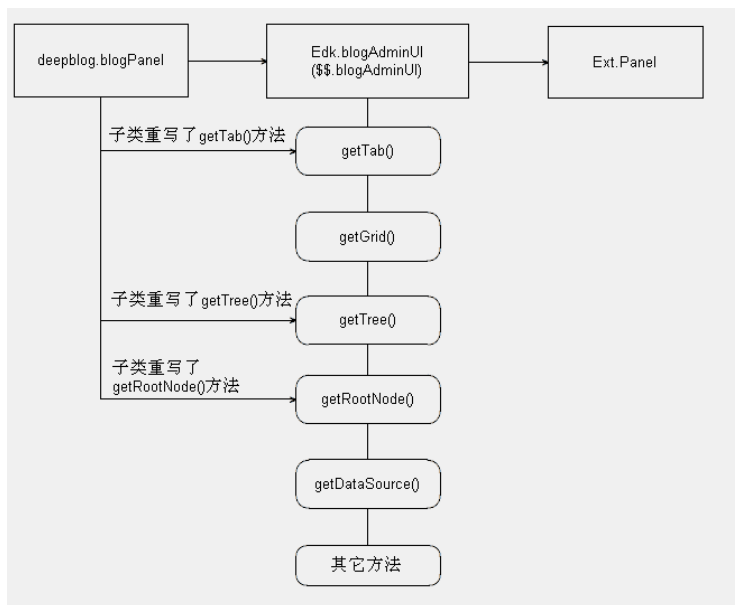


图 9.5 业务层 UI 的继承层次

Edk.blogAdminUI 是 Ext.Panel 的子类，配置为 layout: ‘border’ 的风格，插入其中的组件就是当前 UI 业务需要用到 Tab、Grid、Tree、Form 四大组件，具体的分布如下：

- 左边的树状组件。
- 右边的 Tab 多标签页容器。
- Tab 容器中又有 Grid、Form。

如 `$.blogAdminUI.initComponent` 方法所示：

```
,initComponent: function(){
    var tab, grid, tree;
    tab = new Ext.TabPanel(this.getTab());
    tree = new Ext.tree.TreePanel(this.getTree()); // This is a instance
                                                    // of TreePanel now.
```



```
// 该方法供 grid/tree 等组件所引用, 返回 tab 对象
this.getTabContainer = function(){
    return tab;
}
grid = Ext.ComponentMgr.create(this.getGrid()); //shorthand: Ext.create()
grid.on('afterRender', this.syncTreePanel, tree);
tab.add(grid);

this.items = [tree, tab];
$$blogAdminUI.superclass.initComponent.call(this);
}
```

这取决于 UI 界面如何安排。当前博客的设计是最简单的一类, 请读者以更丰富的想象力和参考别的 B/S 程序的经验, 打造新的 UI 布局。当前容器初始显示的控件默认为 Grid, 即显示出实体的数据表, 如 `tab.add(grid)` 所示。

总之在 Panel 渲染管辖下的子组件通通在 `$$blogAdminUI.initComponent` 方法内宣告完成, 最后 “`this.items = [tree, tab];`” 确定了子项后便加入其中。

注入子组件, 可以选择注入默认 Ext API 的组件, 也可以注入自己写的组件。如 `tab = new Ext.TabPanel(this.getTab())` 就是默认的 Tab 组件, `tree = new Ext.tree.TreePanel(this.getTree())` 就是默认的树组件。如果自定义 Grid 的功能较多, 新写一个 Grid 扩展是一个好的做法。如本例的 Grid 便是如此。这样一来, 又一次使用了 `Ext.extend()` 继承方法, 详见代码 “`$$grid=Ext.extend(Ext.grid.GridPanel, ...)`”。我们在 Form、Tree、Tab 上面如此类推, 还可以在此架构的纵深上实现更复杂的构造。



提示: 此处的 `Ext.ComponentMgr.create(this.getGrid())` 使用了 Xtype 创建 Grid 实例, 实质与 `new Edk.grid(this.getGrid())` 是等价的。

`blogAdminUI` 不但明确了有哪些组件参与容器中, 还扮演着各 Ext UI 管理者类的角色, 如 “`grid.on('afterRender', this.syncTreePanel, tree);`”, 说明 Grid 渲染数据之后, 通知 TreePanel 加入 “分类节点”, 也就是说通过事件协调了双方组件的操作。当然我们也可以通过其他更复杂手段加入 UI 交互的内容。

`deepblog.blogPanel` 是 `Edk.blogAdminUI` 的子类(源码位于 `asset/deepblog.js`)。尽管除了 `getTab()`、`getTree()`、`getRootNode()` 方法外, 我们发现 `blogPanel` 没有其他额外的代码行数, 但不表明我们不能对 `Edk.blogAdminUI` 有更多的修改。

`deepblog.blogPanel` 层的逻辑代码就是针对应用程序的配置层, 从而进行一些个性化的配置、新功能的加減与修正等一系列 `Edk.blogAdminUI` 层次不宜完成的任务, 均放在其子类去完成。这正是 `Edk.blogAdminUI` 不立刻终结继承关系, 还要延伸多一层 `deepblog.blogPanel` 的原因。



提示: 当前只是具体问题具体分析。如果要用更抽象的思维去划分, 其实可以跳出当前命题, 达至……N 层。

厘清了各层次的纵深关系之后,我们晓得,结合 Ext UI 多方复合组件的话,就设定一个类,使用 JSON 配置项将 UI 参与进来,进行新的参数项配置。Edk.blogAdminUI 便是实施此种代码的组织类。可以说,通过这样的架构,我们解决了宏观设计层次的问题,而各个 UI 组件的细节设定,则有待下面的叙述。

9.2.2 “1+2”之二

仍然拿来图 9.5 进行分析,乍看上去就是中间“重”,两头轻,不得不说,位于中间的 Edk.blogAdminUI 的确是关键的一个类。前文中大多是针对架构作横向介绍——只是列明前后继承的对象,尚未就内部的问题,交待清楚怎么解决复用 Grid、Tree、Form、Action……即使介绍的 initComponents() 也属于垂直方向的介绍,但仅限于浅层介绍。要全面理解 Edk.blogAdminUI 的作用,应该以更加垂直的角度深入 Edk.blogAdminUI.* 的各个方法,然后延伸至 deepblog.blogPanel.*。



提示: 复用 Grid、Tree、Form、Action 其实质就是复用配置项内容。

打开 blog.js 源码, getTab()、getTree() 方法对应着 Tab、Tree 的配置内容:

```
,getTab: function(){
    var tab = {
        xtype: 'tabpanel'
        ,closable : true
        ,activeTab: 0
        ,region: 'center'
        ,plugins : new Ext.ux.TabCloseMenu()
        //@override
        ,add: function(container){
            var hasInserted = !!(this.items
                && this.getComponent(container.id))
                && container.recordId);
            if (hasInserted) {
                this.setActiveTab(container); // 对已经打开了 tab, 设置 Active
            }else {
                Ext.TabPanel.prototype.add.call(this, container);
                this.setActiveTab(container);
            }
        }
        return this;
    };
    return tab;
}

, ...
/**
 * @return {Ext.tree.TreePanel}
 */
```



```
,getTree: function(){
    var treePanel = {
        xtype       : 'treepanel'
        ,region      : 'west'
        ,split       : true
        ,border      : true
        ,cls         : 'blogTree'
        ,containerScroll : true
        ,enableDragDrop : true
        ,ddGroup     : 'myDD_group'
        ,root        : this.getRootNode() // new Ext.tree.AsyncTreeNode
        ,autoScroll   : true
        ,width       : 160
        ,selModel     : new Ext.tree.MultiSelectionModel()

    };
    return treePanel;
}
```

从方法命名的规律上可以看出，UI 具体的配置项内容都写在 `getTab()`、`getTree` 等以 `get` 为开头的方法中。一般这些方法返回其各自的 UI 组件配置项。每一个配置项对象都是普普通通的 JavaScript 对象(Plain Old JavaScript Object)，并不是其他类型的对象。另外，`getTab()`、`getTree` 方法的命名规则还有一个重要的暗示，便是这些方法可以被重写的(override)，而且配置项对象与重写方法之间有密切的联系，构成了相辅相成、缺一不可的策略模式。这两点便是“1+2”中的 2。重写方法的叙述我们按下不表，先完成本小节的任务，把组件配置项说明清楚。



提示：`getGrid()`为什么不返回组件实例，如 `new Ext.Grid()`？

当然不可以，实例化后就是硬性的对象。我们使用 JSON 配置项的目的就是为了避免操控实例对象，尤其在多个类、多个层次的继承关系中——虽然 Ext 为我们提供了后期修改方法，但更合适的方法是修改前期的配置项内容。实例化的过程只有一次，而且安排在最后执行。具体原理可回顾 `initComponent()` 初始组件的过程。

组件配置项为常见的对象的配对结构(key/value)。经过 Edk 一系列的约定，其意义与我们熟悉的 `Xtype` 既有区别又有联系。主要区别在于 `Xtype` 是隐式构造 UI 组件，对于程序员而言背后的代码不可见；而我们当前的配置项虽然也是 JSON 结构，但用于设定若干默认的配置，可随着不同需求反复进行修改，最终送入到 `Edk.blogAdminUI.initComponent()` 方法内，传入到 `new Ext.TabPanel(...)`、`new Ext.tree.TreePanel(...)` 等的构造参数中。



提示：能不能不写方法，直接写属性代替，如：`grid_height: 100`、`grid_cols:[{...}]`、`tree_weight: 350... .. ?`

当然可以，理论上没任何问题，但请注意，尽管原理上通过，然而有许多不同的 UI 组件，这样的 API 容易写得乱，还是以方法 `getGrid`、`getTab`……来区分好。

9.2.3 “1+2”之三

前文谈到划分各个层次的布局原因,即不同职责的逻辑单元类有其不同的代码呈现。UI组件的配置项参数亦同样如此。相对地,上一层父类提供了默认配置项,下一层的子类继续使用父类配置的同时,得到更进一步修改或调整的权利,也就是说,配置项参数应分布在多个层次上,各层之间可以非常灵活地修改。从扁平化角度看,重写方法的目的在于让各个环节的同名方法连接在一起,也就是每一个独立的方法都是一个加工的工序。

既然我们晓得这点,那么下一步应该怎么来实施?此时就必须理解 Ext 的重写方法机制并加以运用。Edk.blogAdminUI.getTab()/getGrid()/getTree()的方法固然可以在本类中直接访问,而且允许在子类中访问它们。正因为有了这样的实例方法返回 UI 配置项内容,才可以允许子类继续修改配置内容。Ext 的 OO 机制允许我们重写实例方法(override),分阶段划分逻辑单元,以产生略加修改或配置逻辑。本例之中,父类 Edk.blogAdminUI 的 getTab()、getTree()、getRootNode()方法都在其子类 deepblog.blogPanel 类中得到重写。如 asset/deepblog.js 文件所示:

```
deepblog.blogPanel = Ext.extend($.blogAdminUI, {
    getTab: function(){
        var firstPanel = new Ext.Panel({
            title      : '公告栏'
            ,autoLoad   : 'service/tpl/welcome.htm'
            ,autoScroll : true
        });
        var tabPanel = deepblog.blogPanel.superclass.getTab.call(this);
        tabPanel.items = firstPanel;
        return tabPanel;
    }
    /**
     * @param {Edk.grid} grid
     * @return {Ext.tree.AsyncTreeNode}
     */
    ,getRootNode: function(grid){
        var rootNode =
            deepblog.blogPanel.superclass.getRootNode.call(this, grid);
        rootNode['text'] = 'DeepBlog';
        return rootNode;
    }
    /**
     * @param {Edk.grid} grid
     * @return {Ext.tree.TreePanel}
     */
    ,getTree: function(grid){
        var treePanel =
            deepblog.blogPanel.superclass.getTree.call(this, grid);
        var bbar = [
```






```
new Ext.form.Label({html: "当前已登录:{0} "
    .format(Edk.config.getConfig()['edk_logged_user']['title'])})
,Edk.UI.actions.user.logout
];
treePanel.bbar = bbar;
return treePanel;
}
});
```

打开文件 `blog.js` 对比 `Edk.blogAdminUI`, 我们通过 `deepblog.blogPanel.superclass.getTab()/getRootNode()/getTree()` 访问父类方法的引用, 并使用 `call()/apply(this, 参数列表)` 的手段强行指定函数的作用域为当前的实例(`deepblog.blogPanel` 实例), 相当于标准 OO 语言中的 `super()`。这个过程就是重写方法的过程。

因为约定之中, 以 `get` 前缀的 `getXXX()` 方法返回的是 JSON 对象而不是已经实例化的 `Ext` 对象, 所以可以直接修改该 JSON 结构, 并一定要将其返回。`deepblog.blogPanel` 中各个方法的作用如表 9.1 所示。

表 9.1 override 方法的作用

方法名称	作用	产生的效果
重写 <code>getTab()</code>	创建一个“公告栏”的 <code>Ext.Panel</code> , 这是一个带欢迎性质的公告栏, 加入到 <code>TabPanel</code> 中	
重写 <code>getRootNode()</code>	仅仅设置根节点的 <code>text</code> , 即 <code>rootNode['text'] = 'DeepBlog'</code>	
重写 <code>getTree()</code>	替 <code>TreePanel</code> 加入 <code>bottom bar</code> (内有登录人员名称、“退出”按钮)	

根据项目需求的不同, 适宜在 `deepblog.blogPanel` 最后一层做出修改。

与此同时, 我们不能忽略组件级别的配置项方法的重写技巧, 如本例中 `TabPanel` 和 `Grid`, 就是一个明显的例子, 而且表现得更加灵活。例如 `blog.js` 的 `$.blogAdminUI.getTab` 方法的部分片断:

```
var tab = {
    ...
    ,add: function(container){
        var hasInserted = !!(this.items && this.getComponent(container.id))
        && container.recordId);
```



```
if (hasInserted) {  
    this.setActiveTab(container); // 对已经打开了 tab, 设置 Active  
} else {  
    Ext.TabPanel.prototype.add.call(this, container);  
    this.setActiveTab(container);  
}  
return this;  
}  
};
```

其中 `Ext.TabPanel.prototype.add.call(this, container)` 就是切换到那个 Tab 显示, 参数 `container` 指定了具体的 Tab。`add()` 定义后必然覆盖原实例中的 `add()` 方法, 但仍可以用 `Ext.TabPanel.prototype.add` 的寻址模式找到原实例 `add` 方法, 即父类方法在内存中没用被删除。JavaScript 很活很自由, 但不等于无迹可寻, 明确某一对象究竟为何即明确对象的成员。这是重写方法新的技巧性之一。

重写方法的时候, 之所以加入一个 `if` 判断, 是希望重写该方法不会为 `add()` 带来副作用, `else` 部分就和原来的 `add()` 方法一样。



提示: UI 制作经验谈: 如何不会重复打开 Tab?

每一张 Tab 面板都代表记录的编辑/新建的状态。当用户打开了某个记录之后, 怎么限制用户再次打开而出现两张相同的 Tab 呢? 因为合理的设计下是如果重复打开, 就切换到那张 Tab, 而不是再次打开。我们知道, 容器 `TabPanel` 有加入 Tab 的方法 `add()`, 默认不会检测是否有重复的 Tab。

充分利用重写方法这一技巧还不止于 `TabPanel`, 同样道理, 我们在创建 `Grid` 的时候也采用了该模式。请参见 `Grid` 类图 9.6, 然后打开 `blog.js` 观察 `gridCfg` 变量, 属于 `function` 类型的都是需要重写父类方法的, 这时候我们要注意 `$.grid.prototype` 的用法。

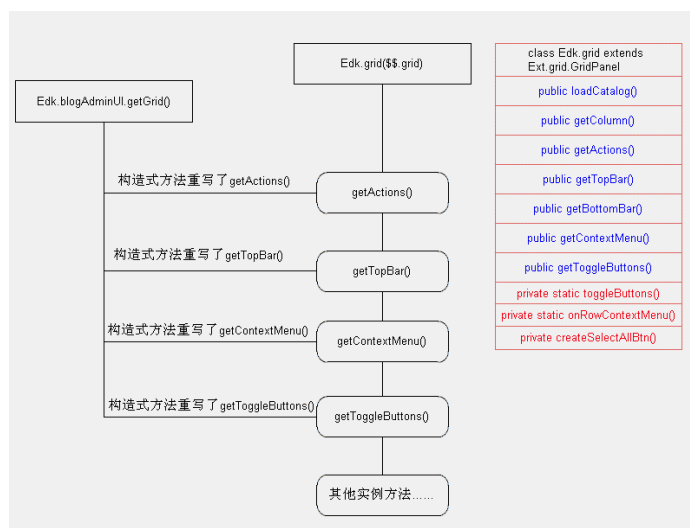


图 9.6 构造器式(非新建类)重写父类方法



```

$.blogAdminUI = Ext.extend(Ext.Panel, {
    ...
    ,getGrid: function(){
        var gridCfg = {
            xtype      : 'EdkGrid'
            ,title      : '博客'
            ,store      : this.getDataSource()
            ,getActions : function(){
                var actions = $.grid.prototype.getActions.call(this);
                actions['perviewAction'] = new Ext.Action({
                    text      : '预览'
                    ,iconCls  : 'icon16-perview'
                    ,disabled : true
                    ,handler  : function(){
                        var client =
                            Edk.config.getConfig()['edk_website_XD_URL'];
                        window.open("{0}/?ID={1}".format(client,
                            this.getSelected().id));
                    }
                },scope: this.sm
            ));
                return actions;
            }
        },getTopBar : function(){
            var actions = $.grid.prototype.getTopBar.call(this);
            actions.push('->');
            actions.push(this.actions.perviewAction);
            return actions;
        }
        // 返回的值注意 menu 的数据结构
        ,getContextMenu : function(){
            var menuCfg = $.grid.prototype.getContextMenu.call(this);
            menuCfg['items'].push('-', this.actions.perviewAction);

            return menuCfg;
        }
        ,getToggleButton : function(){
            var toggleButtons =
                $.grid.prototype.getToggleButton.call(this);
            toggleButtons.push(this.actions.perviewAction);

            return toggleButtons;
        }
    };
    gridCfg.getTabContainer = this.getTabContainer; // 对外暴露 Tab

    return gridCfg;
}
...
});

```

`getGrid()`是对 `Edk.grid` 的重新配置过程(见图 9.6)。尽管上下文中不曾发现 `Edk.grid` 的字样,但最后 `gridCfg` 会传入 `Edk.grid` 的构造函数中,连同 `getActions()/getContextMenu()/getTopBar()/getToggleButtons()`需要重写的方法,也就是利用了重写方法的技巧性。怎么能够修改 `Grid` 的 UI 配置?以 `getTopBar()`为例, `Edk.grid.getTopBar()`返回“增、删、改、查”四个默认的按钮——对于 `deepBlog` 肯定不够。此时我们说,要为博客管理的 UI 添加一个“预览 `perview`”的按钮,允许用户预览博客文章的这么一个按钮的话(如图 9.7 所示),其结果就是在 `$.blogAdminUI.getGrid()`中的 `gridCfg.getTopBar()`代码部分,修改顶部工具条的配置(配置点位于 `Grid.tbar` 的 `buttons []`),把“预览”按钮加入 `actions` 数组:

```
var actions = $.grid.prototype.getTopBar.call(this);
actions.push('->'); // '->'表示“占位符”,后面的按钮居右列出
actions.push(this.actions.perviewAction); //perviewAction 是 Ext.Action 的特例
```



图 9.7 加入特定的按钮

根据不同的业务需求,可加入更多的按钮,甚至允许修改原来的“增、删、改、查”四个默认的按钮。

另外,不但顶部、底部的工具条可自由配置,而且延伸开来,包括右键菜单(`getContextMenu()`)、状态翻转按钮(`getToggleButtons()`)、列模型(`getColumn()`)、UI 动作(`getActions()`)……都可以应用重写模式的模式获得更灵活可控的配置项参数。

9.2.4 小结

“1+2”的中 3 种模式(`Ext.extend()`/配置项/重写方法)都是源自于地道的 `Ext` 方式方法,经过作者包装些许而成。

换句话说,本节介绍的模式只是一种轻型的控件封装模式,尚未加入大家跃跃欲试的 23 条“设计模式”。

一些用户可能觉得设计模式“只此一家,别无分店”,脚本语言很少有机会用,但作者深信,达到了某种足够的程度,我们的 UI 也可以让更多的“设计模式”一项项地参与近来,也必定是一道道美丽的风景线,能更好地释放大家的生产力。

至于另外一个地道的 `Ext` 用法——“事件”,“1+2”还没有得到有效的发挥。这是该模式的缺点之一。下面 `Edk.formPanel` 就会有充分使用“事件”的地方。



9.3 Edk.grid.*

9.4 Edk. tree.*

9.5 Edk.formPanel.*

9.6 Edk.attachment.*