

LINB3.0 教程 —— 入门篇

最后更新於 2010 年 7 月 15 日

| | |
|--------------------------------------|----|
| 缘 起 | 7 |
| 第一章 从“Hello World”开始 | 8 |
| 第一节 环境准备 | 8 |
| 1. 下载发布包 | 8 |
| 2. 发布包放在哪里..... | 8 |
| 3. 首先浏览一下例子和 API | 9 |
| 第二节 Hello World 来了 | 10 |
| 第三节 一个传统的需求—html 布局替换 | 12 |
| 第四节 让我们用一下设计器..... | 13 |
| 第五节 有必要了解一下加载过程..... | 18 |
| 第六节 再体验一下代码编辑器..... | 20 |
| 1. 从类结构窗口定位代码..... | 20 |
| 2. 代码折叠 | 21 |
| 3. 代码智能提醒..... | 21 |
| 1) 当输入上下文不识别的字符 | 22 |
| 2) 当输入 “.” | 22 |
| 3) 当用快捷键 Atl+1 或双击选中词条 | 23 |
| 4. 控件切换到定义代码..... | 23 |
| 5. 事件代码自动构造..... | 24 |
| 第二章 来认识一下控件们..... | 25 |
| 第一节 搭建脚本测试环境..... | 25 |
| 第二节 在 env.html 中试一下 hello world..... | 26 |
| 第三节 产生控件的方式和运行时(runtime)更新..... | 27 |
| 第四节 Button 相关..... | 28 |
| 6. 按钮的 onClick 事件 | 28 |
| 7. 状态按钮与 CheckBox | 29 |
| 8. 更简单的按钮——Link 控件 | 29 |
| 第五节 Label 相关 | 30 |
| 第六节 Input 相关 | 30 |
| 2. 得到和设置输入值..... | 31 |
| 3. 脏数据标识 | 31 |
| 4. 密码框 | 31 |
| 5. 多行输入 | 32 |
| 6. 输入验证 | 32 |
| 7. 动态输入验证..... | 33 |
| 8. 输入验证消息的显示方式..... | 33 |
| 1) 验证出错图标 | 33 |
| 2) 验证提示信息 | 34 |
| 3) 验证信息绑定 | 34 |

| | |
|---|----|
| 4) 自定义验证信息 | 34 |
| 9. 掩码输入——mask input | 35 |
| 10. 终于轮到 ComboInput | 36 |
| 1) 列表选择 | 36 |
| 2) combobox、listbox 和 helpinput 的区别 | 36 |
| 3) 日期选择 | 37 |
| 4) 时间选择 | 38 |
| 5) 颜色选择 | 38 |
| 6) 文件选择 | 39 |
| 7) 赋值命令 | 39 |
| 8) 自定义选择 | 40 |
| 9) comboInput 中的命令按钮 | 40 |
| 11. 富文本编辑框 RichEditor | 41 |
| 第七节 List 相关 | 42 |
| 1. 简单列表 | 42 |
| 2. 稍微复杂一点的 | 43 |
| 3. RadioBox | 44 |
| 4. IconList 和 Gallery | 44 |
| 5. 代码控制条目选择 | 45 |
| 第八节 容器相关 | 45 |
| 1. Pane 和 Panel 的区别 | 46 |
| 2. Pane 和 Block 的区别 | 47 |
| 第九节 对话框 | 47 |
| 1. 最普通的 | 47 |
| 2. 最大化、最小化 | 48 |
| 3. 模式对话框 | 49 |
| 第十节 布局控件 | 49 |
| 第十一节 多页控件 | 51 |
| 1. 是否自带容器面板 | 51 |
| 2. ButtonViews 的 4 个种类 | 52 |
| 3. 代码控制页选择 | 53 |
| 4. 动态增加和删除页 | 53 |
| 1) 页的关闭按钮 | 53 |
| 2) 代码添加页和删除页 | 55 |
| 5. 动态内容加载 | 55 |
| 第十二节 菜单和工具栏 | 57 |
| 1. 弹出菜单 | 57 |
| 2. 菜单栏 | 57 |
| 3. 工具栏 | 58 |
| 第十三节 树形栏 | 59 |
| 1. 三种选择模式 | 59 |
| 1) 不可选择 | 59 |
| 2) 单项选择 | 60 |
| 3) 多项选择 | 60 |

| | |
|---------------------|----|
| 2. 组条目 | 61 |
| 3. 默认展开到节点..... | 61 |
| 4. 互斥展开 | 62 |
| 5. 动态销毁 | 63 |
| 6. 树节点动态加载..... | 63 |
| 第十四节 树形表格 | 64 |
| 1. 给表头和表格赋值..... | 64 |
| 1) 按照标准格式赋值 | 65 |
| 2) 按照简化格式赋值 | 66 |
| 2. 获得表头的三种数据格式..... | 66 |
| 3. 获得表格的三种数据格式..... | 67 |
| 4. 表格的三种活动模式..... | 68 |
| 1) 无活动模式 | 68 |
| 2) 行活动模式 | 69 |
| 3) 单元格活动模式 | 69 |
| 5. 表格的六种选择模式..... | 70 |
| 1) 不可选择 | 70 |
| 2) 可选择一行 | 71 |
| 3) 可选择多行 | 71 |
| 4) 可选择一个单元格 | 72 |
| 5) 可选择多个单元格 | 72 |
| 6. 树状表格 | 72 |
| 7. 配置列 | 74 |
| 1) 特殊的第一列：行头列 | 74 |
| 2) 列的宽度 | 75 |
| 3) 通过拖拽改变列宽 | 75 |
| 4) 通过拖拽改变列的位置 | 76 |
| 5) 列排序 | 76 |
| 6) 列的自定义排序 | 76 |
| 7) 列是否可以隐藏 | 77 |
| 8) 列的单元格种类 | 77 |
| 9) 列头的样式 | 78 |
| 10) 给列头加图标 | 79 |
| 11) 动态更新列头 | 79 |
| 8. 配置行 | 80 |
| 1) 行的高度 | 80 |
| 2) 通过拖拽改变行高 | 80 |
| 3) 行的单元格种类 | 81 |
| 4) 行的样式 | 81 |
| 5) 自动行号 | 82 |
| 6) 自定义行号 | 82 |
| 7) 间隔色 | 83 |
| 8) 分组 | 84 |
| 9) 预览区域和小结区域 | 84 |

| | | |
|------|----------------------------------|-----|
| 10) | 动态更新行 | 85 |
| 9. | 配置单元格 | 86 |
| 1) | 单元格的种类 | 86 |
| 2) | 单元格的样式 | 87 |
| 3) | 动态 update 单元格 | 87 |
| 10. | 界面编辑状态 | 88 |
| 1) | 表格可编辑 | 89 |
| 2) | 列可编辑 | 89 |
| 3) | 行可编辑 | 89 |
| 4) | 单元格可编辑 | 90 |
| 5) | 编辑器的设置 | 90 |
| 6) | 自定义单元格编辑器 | 91 |
| 11. | 增加行和删除行 | 93 |
| 第十五节 | 其他控件 | 93 |
| 1. | ProgressBar 控件 | 93 |
| 2. | Slider 控件 | 94 |
| 3. | Image 控件 | 94 |
| 4. | PageBar 控件 | 95 |
| 5. | 高级控件 | 95 |
| 第三章 | 与后台服务的数据交互 | 97 |
| 第一节 | 最好先安装 Fiddler | 98 |
| 第二节 | 获取文件的内容 | 98 |
| 第三节 | 同步数据交换 | 98 |
| 第四节 | 与异域交换数据 | 99 |
| 1. | 对 SAjax 的数据流监控 | 99 |
| 2. | 对 IAjax 的数据流监控 | 100 |
| 第五节 | 文件上传 | 101 |
| 1. | 用 ComboInput 来选择上传文件 | 101 |
| 2. | 用 IAjax 来执行上传 | 102 |
| 第六节 | 处理数据交换的基本函数模板 | 102 |
| 第七节 | XML 数据 | 103 |
| 第八节 | 综合示例 | 104 |
| 第四章 | 分布式 UI 入门 | 105 |
| 第一节 | 来自远程 js 文件的对话框模块 | 105 |
| 第二节 | linb.Com 和 linb.ComFactory | 106 |
| 1. | linb.ComFactory 的配置 | 106 |
| 2. | 应用入口 linb.Com.Load | 107 |
| 3. | newCom 和 getCom | 108 |
| 4. | linb.UI.Tag 的作用 | 109 |
| 5. | 销毁分布式 UI 模块 | 109 |
| 6. | 对于代码已加载的 Com | 109 |
| 第五章 | 一些基本的问题 | 110 |
| 第一节 | 弹出窗口 | 110 |
| 1. | alert | 110 |

| | | |
|-----|-------------------------------------|-----|
| 2. | confirm..... | 110 |
| 3. | prompt..... | 111 |
| 4. | pop | 111 |
| 第二节 | 异步执行 | 112 |
| 1. | asyRun | 112 |
| 2. | resetRun | 112 |
| 第三节 | 改变皮肤 | 112 |
| 1. | 改变系统皮肤..... | 112 |
| 2. | 改变单个控件的皮肤..... | 113 |
| 第四节 | 改变页面的语言显示..... | 113 |
| 第五节 | DOM 节点的基本操作 | 114 |
| 1. | 节点的生成和插入..... | 114 |
| 2. | 节点属性和 CSS 属性 | 115 |
| 3. | 对 CSS 类的操作 | 115 |
| 4. | 操作 Dom 事件..... | 116 |
| 5. | 拖拽 Dom 节点..... | 117 |
| | 1) 拖拽概要对象 | 118 |
| | 2) 拖拽的相关事件 | 119 |
| 第六节 | 模板——linb.Template..... | 120 |
| 1. | 一个较简单的模板应用..... | 121 |
| 2. | 稍微复杂点的模板应用..... | 121 |
| 3. | 用 linb.Template 实现一个 SButton 吧..... | 122 |
| 第七节 | 关于调试 | 123 |
| 1. | 用于调试的代码包..... | 123 |
| 2. | 调试工具 | 123 |
| 3. | linb 的内置工具 | 124 |
| 第六章 | 一些典型的需求..... | 125 |
| 第二节 | 布局 | 125 |
| 1. | 用 dock 完成布局..... | 125 |
| 2. | 用 Layout 控件完成布局 | 125 |
| 3. | 相对位置布局..... | 126 |
| 第三节 | UI 拖拽 | 128 |
| 1. | 在容器面板间拖拽控件..... | 128 |
| 2. | List 拖拽排序 1 | 128 |
| 3. | List 拖拽排序 2 | 129 |
| 第四节 | Form 表单..... | 130 |
| 1. | 表单 1 | 130 |
| 2. | 表单 2 | 131 |
| 第五节 | 定制界面入门..... | 132 |
| 1. | 只改变一个实例的样式 1..... | 132 |
| 2. | 只改变一个实例的样式 2..... | 133 |
| 3. | 只改变一个实例的样式 3..... | 133 |
| 4. | 只改变一个实例的样式 4..... | 133 |
| 5. | 只改变一个实例的样式 5..... | 134 |

| | | |
|-------|-----------------------------------|------------|
| 6. | 只改变一个实例的样式 6..... | 134 |
| 7. | 改变控件类的样式..... | 135 |
| 8. | 自定义皮肤 | 135 |
| | 1) 第一步: Copy 一套皮肤到自己的目录..... | 135 |
| | 2) 第二步: 改变图片或 theme.css 文件内容..... | 136 |
| 第六节 | 自定义控件入门..... | 错误! 未定义书签。 |
| 1. | 给 List 加上 CheckBox | 错误! 未定义书签。 |
| 入门篇结语 | | 137 |

缘 起

linb 框架是一个 AJAX RIA 开源解决方案,可以在 LGPL 协议下可免费应用于个人或商业目的。除了那些必要的底层实现外,框架目前还包含了 40 多个图形界面控件,例如标签(Tabs),窗口对话框(Dialog),树(Tree),树型表格(TreeGrid),时间线(TimeLine)等等,并且还在不断扩展和完善中。

比较有特点的是,linb 本身带有一个所见即所得的界面编辑器。通过简单的拖拽,程序员可以在很短的时间内构造非常复杂的用户界面,在节省了大量开发时间的同时,程序员能把更多的精力投放在业务逻辑上。编辑器本身也是使用 linb 完成的,所以设计时的用户界面和运行时完全一致。linb 界面编辑器还可以将界面代码保存成一个单独的类文件,并作为分布式界面的一个模块供 Web 应用随时“按需动态加载”。多个分布式界面模块之间也可以通过类似于“控制反转”的方式实现无缝装配,所以特别适合于团队的分布式开发。linb 对多浏览器的兼容,包括 IE6+, firefox1.5+, opera9+, safari3+和 Chrome 等等。

linb 框架基于 Web 的富客户端技术可以和任意后端技术(包括但不限于 php, .Net, Java, python)相匹配。彻底的 OO 设计使得她从最底层支持了名字控件和类的封装和继承,同时充分兼容其他流行的 Ajax 框架,如 jQuery, prototype, mootools 等。

虽然 linb 框架的英文原版最早于 2005 年就已经发布在我的个人网站(www.linb.net)上,之后又于 2007 年发布到 Google Code(<http://code.google.com/p/linb>)上去。但一直以来都是以英文的形式存在,直到 2009 年初才有将 linb 在国内推广的想法。

一直以来 linb 在文档方面显得非常薄弱。究其原因 linb 的英文文档和例子本来就少,再加上在国内上中文的时间很短。我所接触到的几乎每一个用户都在敦促有一个简单的教程出来,那么就在现在,我们开始。

教程预计包含三个部分:入门篇、提高篇和高级篇。入门篇主要讲的是一些直观上的应用;提高篇中主要是如何在真实的程序中应用 linb(包括一些定制化的应用);提高篇讲的是 linb 内部的机制和其他一些稍微抽象的话题。

入门篇我会尽快写出来,但由于时间的问题,会比较简要一些。但大家对照文字和代码理解起来应该很容易。提高篇和高级篇可能还要有些时日才会写,主要看用户的需求情况。

关于 linb 的最新信息,大家可以到以下网站获得: <http://www.linb.net>。

如果有什么好的建议,可以给我发邮件: linb.net@gmail.com。

第一章 从“Hello World”开始

正如很多人都期望或不期望的一样，第一个例子就是“Hello World”。

第一节 环境准备

首先，需要注意的是，本教程的所有实例都建立在 3.0 版本上。所以我们的第一个任务是下载 3.0 的发布包，并建立本地环境。

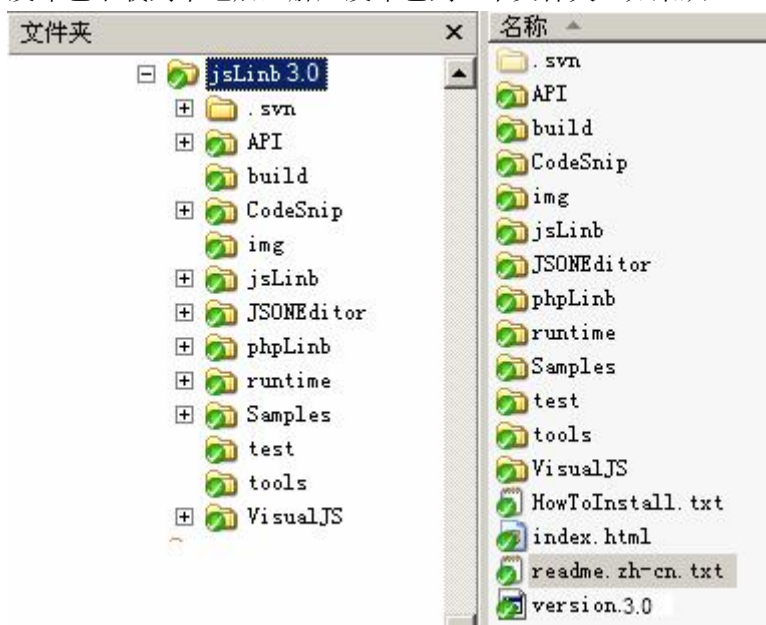
1. 下载发布包

可以从 <http://code.google.com/p/linb/downloads/list> 下载 3.0 版本的发布包。google group 里面会保留 3.0 版本的最新稳定版本的下载，但不总是最新的代码。建议你最好还是从 svn 中取 3.0 版本的最新代码。对于没用过 svn 的人也顺便学习下 svn 的用法，毕竟网上很多开源的东西最新代码都是以 svn 的形式给出的。如果不知道如何用 svn 可以到网上查一下，在 windows 下面推荐使用 svn 工具 TortoiseSVN。

3.0 版的 svn 地址为：<http://linb.googlecode.com/svn/trunk/jsLinb3.0/>。

2. 发布包放在哪里

发布包下载到本地后，解压发布包到一个文件夹（如果从 svn 中取不用解压）。



发布包的文件夹内容

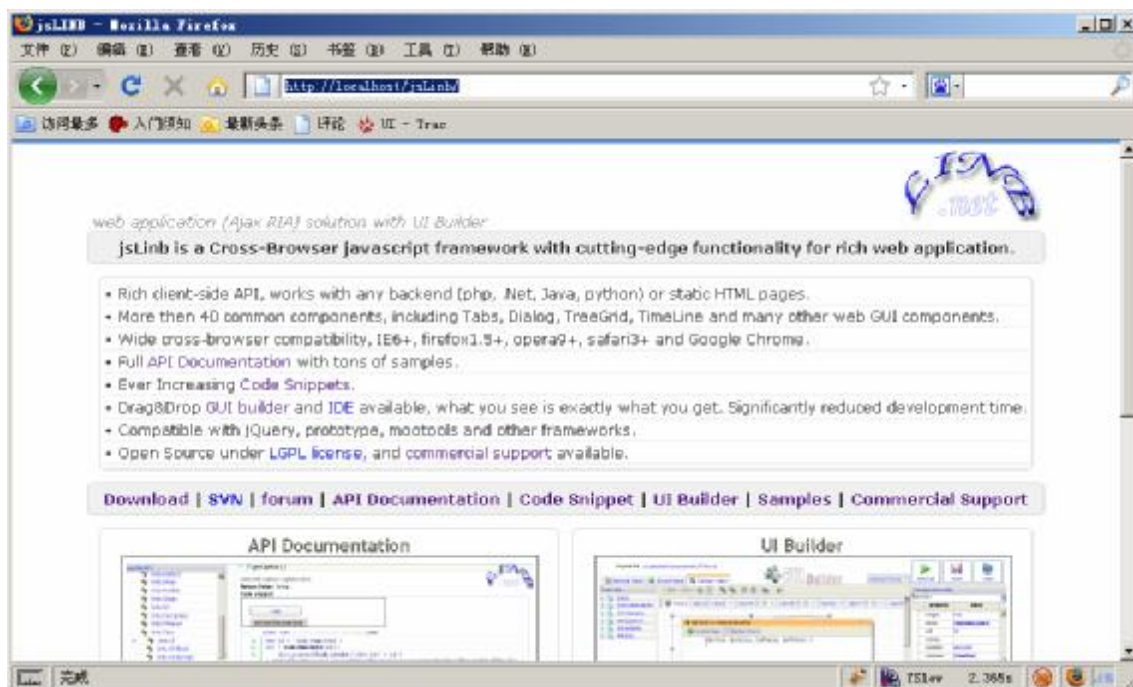
注：里面的 .svn 目录都是 svn 文件夹，阅读的时候忽略就行了。

默认情况下发布包里面的大部分例子都可以直接在本地运行，但少部分例子由于用到了 php 后台(版本 5 以上)或 mysql (版本 5 以上)数据库，需要在 apache 服务器 (版本 2 以上)下才能正常运行。所以要把发布包的文件夹 **copy** 到有效的一个 **apache web** 目录下。

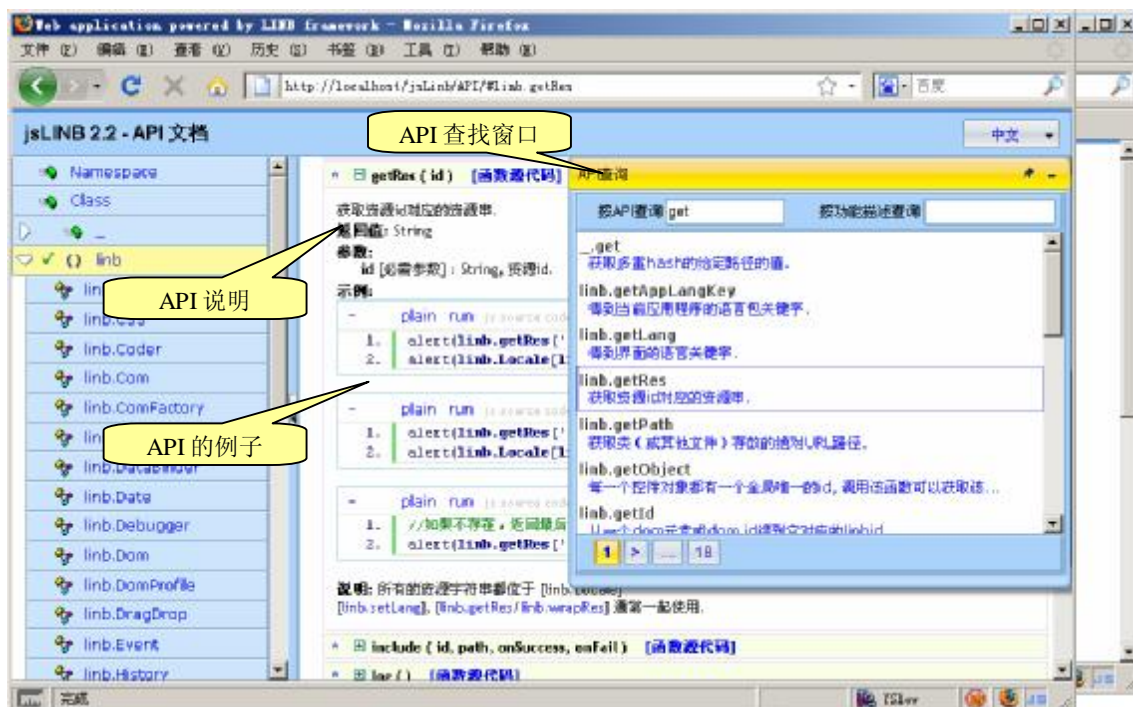
如果你目前没有 apache + php + mysql 的环境，可以从网上下载一个。网上有不少方便的 apache + php + mysql 安装包可以免费使用，我自己用的是 xampplite，感觉还不错。你也可以直接到 <http://www.apachefriends.org/en/xampp-windows.html> 下载一个 xampplite 安装包，然后运行并安装一下就可以了。

3. 首先浏览一下例子和 API

如果你的 xampplite (或其他的相似环境) 安装成功，把发布包文件夹 (例如 jsLinb) copy 到 apache 的 web 目录下 (本教程中都假设你的根目录是 <http://localhost/jsLinb/>) 后，你就应该在浏览器中可以看到 <http://localhost/jsLinb/> 的内容：



现在可以到 <http://localhost/jsLinb/Samples/> 浏览一下各个例子，然后到 <http://localhost/jsLinb/API/> 看一下 API：



建议先简单浏览一下 API，掌握一下如何阅读 API、如何运行里面的小例子和如何查找 API。

由于 linb 里面的 API 非常多，教程中不能一一列举和讲述，很多功能还是要依照这个 API 文档来编码。

大家如果做真实的项目的话时间都是比较紧的，基本上不可能是什么 API 都能记住的，都是边做程序边看 API。拿我自己来说，即使是用 linb 来做程序的话也要边做边看 API（自己写的都能忘记？不是的，是大部分根本没必要记住，太浪费脑内存，只要知道有就可以了）。

第二节 Hello World 来了

现在，在你的发布包目录下建立一个 **cookbook** 目录，在 **cookbook** 目录下建立一个 **chapter1** 目录，**chapter1** 目录下建议一个 **helloworld.html** 文件，并输入以下内容：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="Content-Style-Type" content="text/css" />
  <meta http-equiv="imagetoolbar" content="no" />
  <script type="text/javascript" src="../../runtime/jsLinb/js/linb-all.js"></script>
  <title>jsLinb Case</title>
</head>
<body>
  <script type="text/javascript">
    linb.main(function(){
      linb.alert("Hi", "Hello World");
    });
  </script>
</body>
</html>

```

最好用 strict 模式

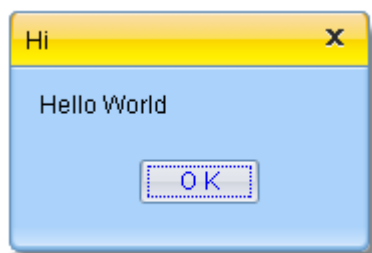
程序的入口

库文件的路径

cookbook/chapter1/helloworld.html 文件内容

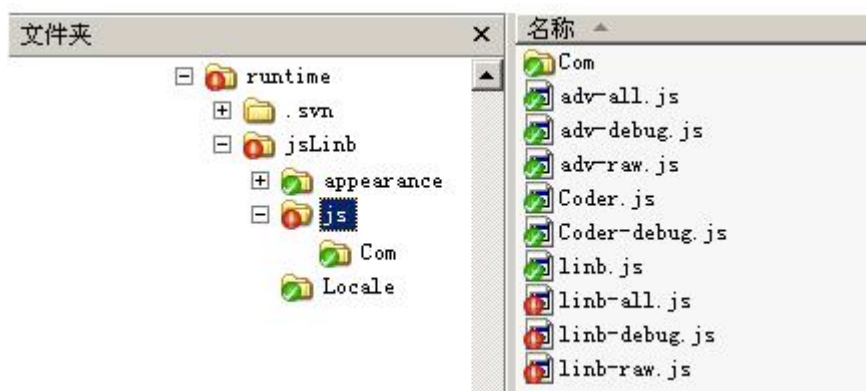
注：教程的所有例子都在 **cookbook** 文件夹下面，你可以在这里找到本教程例子的所有源代码。

你可以双击 helloworld.html，也可以在浏览器中（在程序开发阶段，建议用 firefox）打开 <http://localhost/jsLinb/cookbook/chapter1/helloworld.html>，就可以看到这个例子的效果。



你可能注意到，在以上的代码中并未引入 css 文件。是的，**jsLinb** 中的 **css** 文件是自动生成的，不用手工 include。

“linb-all.js”文件包含了除了高级控件外的所有代码，什么 Button, Input, CombInput, Tabs, TreeBar, TreeGrid 都在这里。这个文件默认的是放在发布包的 runtime/js 目录下。下面是 runtime 目录下的文件：



除了 linb-all.js 文件外，runtime/js 目录里还有其他的一些 js 文件，它们各自的用处在使用到

的时候再讲。

第三节 一个传统的需求—html 布局替换

很多用户直接要求在传统的 html 里面用替换某个节点元素（如某个 div）的方法来加入一些控件。就像某个公司的项目经理在描述需求时说的那样：“我们的美工人员生成一个 html 文件，在 html 里面画出一个 id 为 ‘grid’ 的 div 来，然后技术人员的工作就是用一个真正的 grid 控件来替换这个 div”。

这个例子我们写在 chapter1 的 renderonto.html 里：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="Content-Style-Type" content="text/css" />
  <meta http-equiv="imagetoolbar" content="no" />
  <script type="text/javascript" src="../../runtime/jsLinb/js/linb-debug.js"></script>
  <title>jsLinb Case</title>
</head>
<body>
  <div id="grid" style="position:absolute;left:100px;top:100px;width:300px;height:200px;"></div>
  <script type="text/javascript">
    linb.main(function(){
      var grid = new linb.UI.TreeGrid();
      grid.setGridHandlerCaption('grid')
      .setRowNumbered(true)
      .setHeader(['col 1','col 2','col 3'])
      .setRows([
        ['a1','a2','a3'],
        ['b1','b2','b3'],
        ['c1','c2','c3'],
        ['d1','d2','d3'],
        ['e1','e2','e3'],
        ['f1','f2','f3']
      ]);
      grid.renderOnto('grid');
    });
  </script>
</body>
</html>
```

需要代替的 div 元素

设置列头 handler 的显示值

显示行号

设置列头

设置 grid 数据

代替原来的节点

cookbook/chapter1/renderonto.html 文件内容

运行结果如下：

| grid | col 1 | col 2 | col 3 |
|------|-------|-------|-------|
| 1 | a1 | a2 | a3 |
| 2 | b1 | b2 | b3 |
| 3 | c1 | c2 | c3 |
| 4 | d1 | d2 | d3 |
| 5 | e1 | e2 | e3 |
| 6 | f1 | f2 | f3 |

还有两种方法可以得到相同的结果，代码分别放在了 `renderonto2.html` 和 `renderonto3.html` 里面。

文件 `renderonto2.html` 里用的方法：

```
linb.main(function(){
  (new linb.UI.TreeGrid({
    gridHandlerCaption:'grid',
    rowNumbered:true,
    header:['col 1','col 2','col 3'],
    rows:[['a1','a2','a3'],['b1','b2','b3'],['c1','c2','c3'],
          ['d1','d2','d3'],['e1','e2','e3'],['f1','f2','f3']]
  })).renderOnto('grid');
});
```

属性一股脑的都放在这里，Ext 和其他的库都是这样的。看着字符少其实压缩起来还是 get/set 省空间哦！

cookbook/chapter1/renderonto2.html 文件内容

文件 `renderonto3.html` 里用的方法：

```
linb.main(function(){
  linb.create('TreeGrid',{
    gridHandlerCaption:'grid',
    rowNumbered:true,
    header:['col 1','col 2','col 3'],
    rows:[['a1','a2','a3'],['b1','b2','b3'],['c1','c2','c3'],
          ['d1','d2','d3'],['e1','e2','e3'],['f1','f2','f3']]
  }).renderOnto('grid');
});
```

也可以用 `linb.create` 方法

cookbook/chapter1/renderonto3.html 文件内容

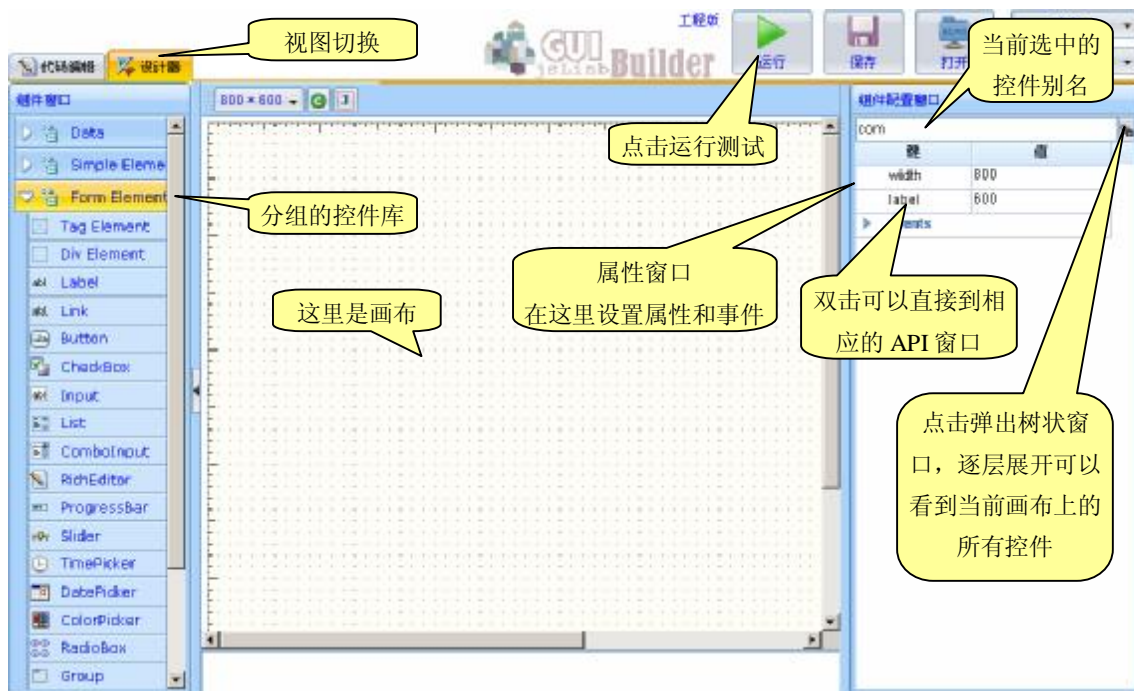
以上三种生成控件的方法得到的是相同的结果，你可以按照项目的需求和自己的习惯来选择具体用什么方法来生成控件。一般来讲，建议用第一种方式（标准的 `new` 和 `setXX` 方式）。

第四节 让我们用一下设计器

让我们用设计器得到同样的结果。

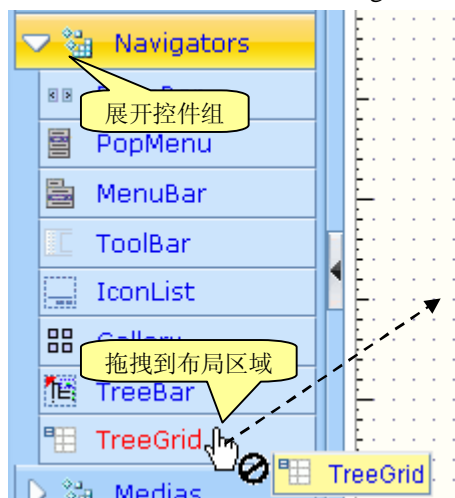
linb 里面有两个设计器：一个是简化版的，另一个是带文件管理功能的高级版。其实这两个本质上没有大的区别，都是为了节省程序员在 `layout` 上的时间的工具。在入门篇里面我们只讲简化版。

访问 <http://localhost/jsLinb/VisualJS/UIBuilder.html>，即可以进入设计器的界面：

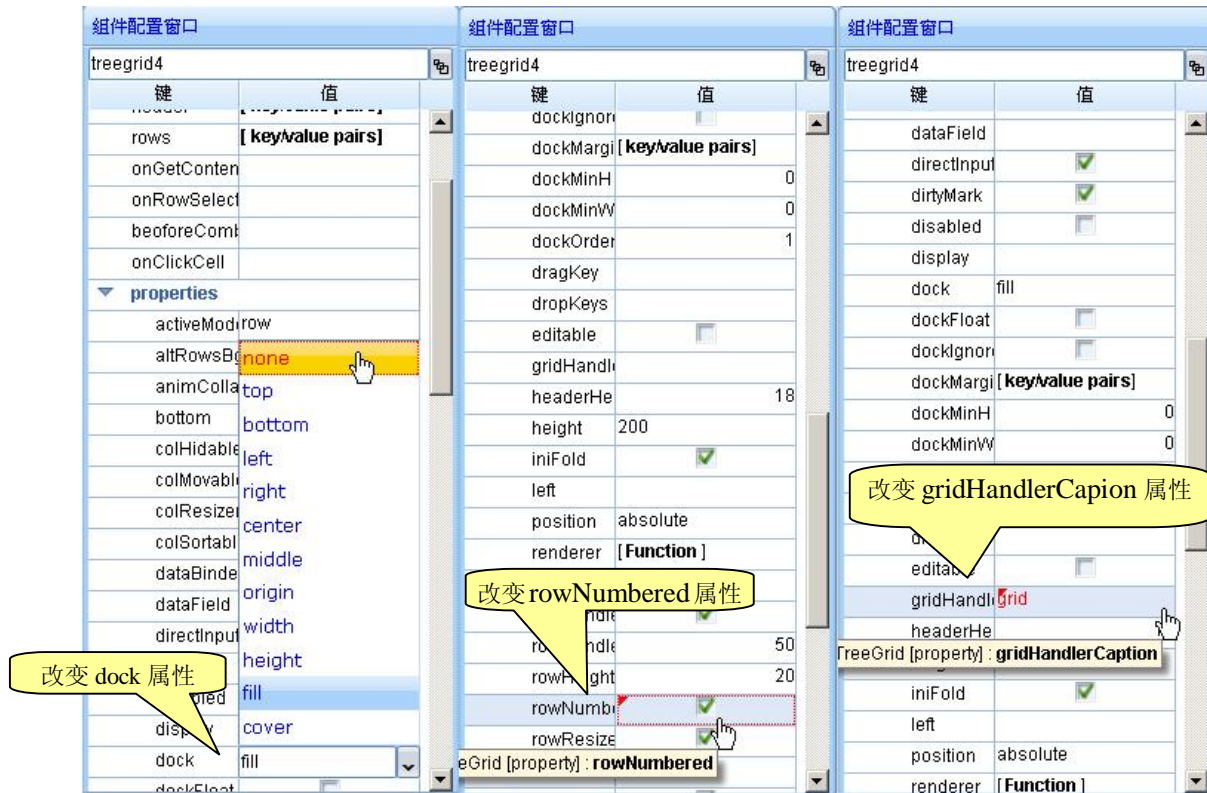


下面开始在设计器中做出这个 grid:

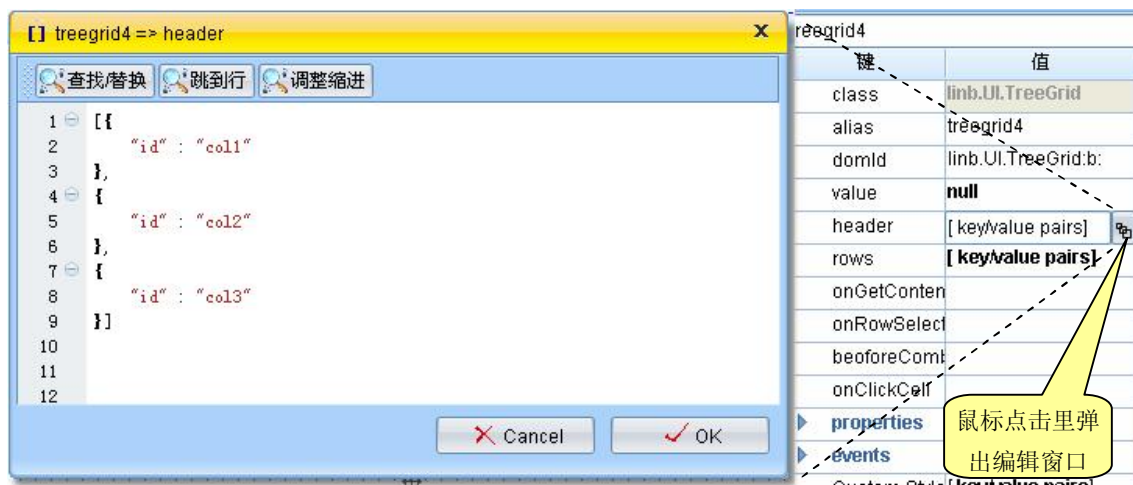
1. 在控件库中打开 Navigators 节点, 拖拽 TreeGrid 到画布上生成一个 TreeGrid;

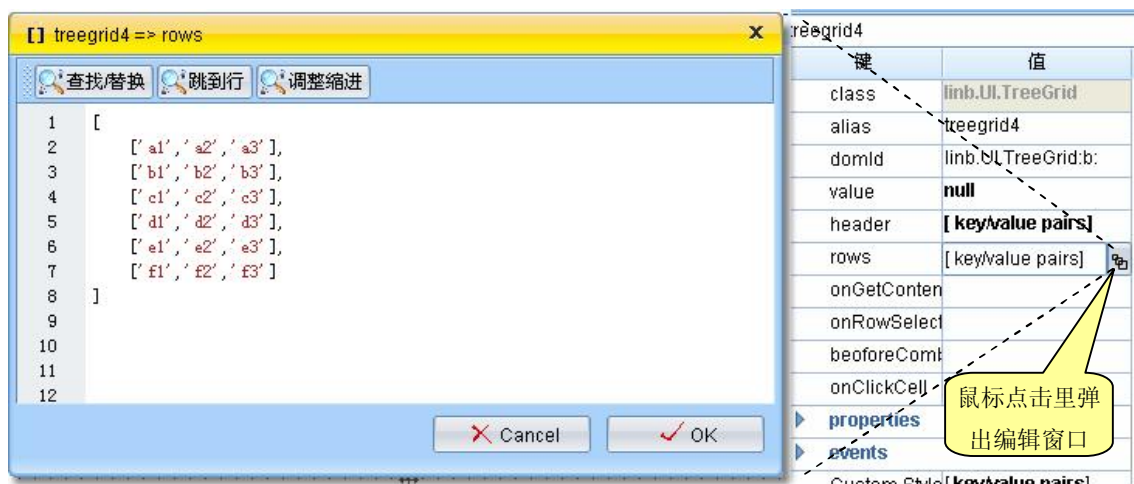


2. 设计器会给 TreeGrid 设置一些默认的值: 例如, dock 设置成了 'fill', header 和 rows 也有 demo 值。如果我们不需要这些默认的值, 可以去掉, 并设置我们需要的值。选中 treegrid, 在右面的属性窗口分别按照如下图设置: 把 dock 属性设置成 'none'; 把 rowNumbered 属性设置成 false; 把 gridHandlerCaption 属性设置为 'grid'。

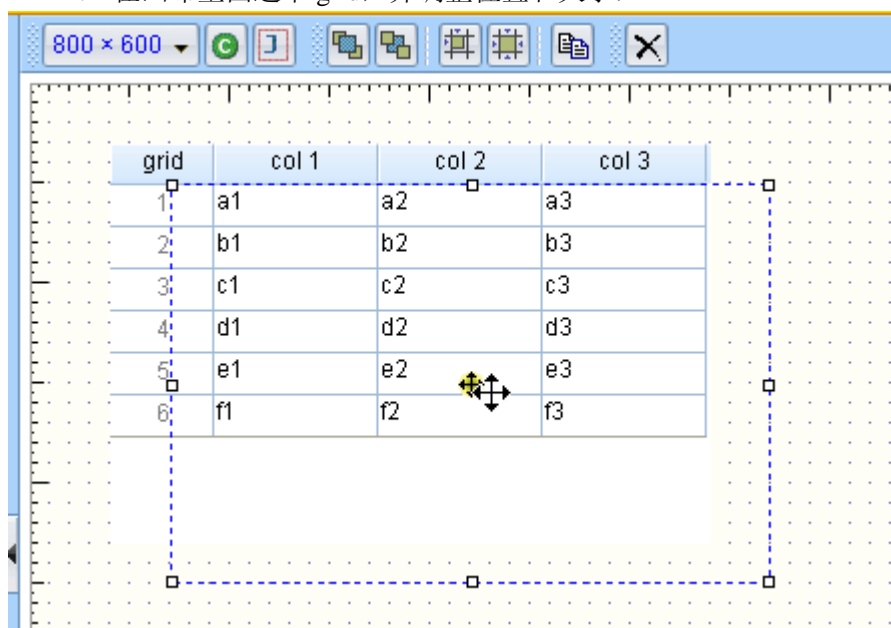


3. 设置 header 和 rows 值。按照图示，弹出属性设置对话框后，分别输入 header 和 rows 的值：

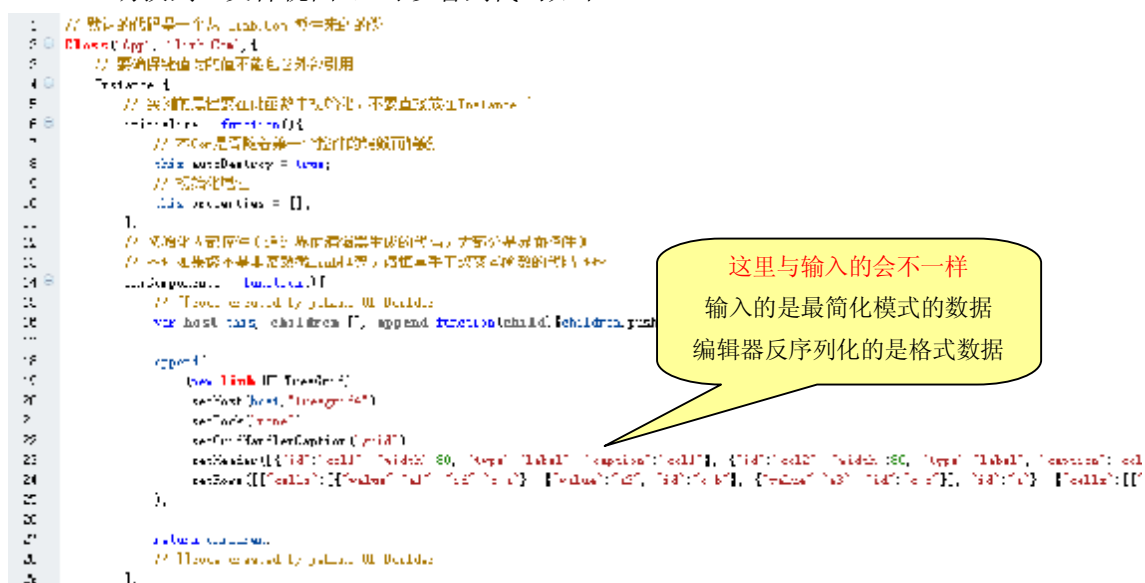




4. 在画布里面选中 grid，并调整位置和大小：



5. 切换到“文件视图”，可以看到代码如下：



由于我们设置的 header 和 rows 的值是最简化模式的，设计器在序列化的时候就给 header

和 rows 附加了些 id 什么的，这个不要紧。其实 TreeGrid 的 getHeader 和 getRows 有三种模式：raw（原始模式），data（数据模式）或 min（最小化模式，header 是一个字符串数组，rows 是一个双层数组）。在设计器里得到的是“data”模式的数据，如果用“min”模式，会得到我们设置的最简化的值。例如：treegrid.getHeader('min') 则得到的是 ['col 1','col 2','col 3']。

6. 下面我们点击“运行”按钮，就会得到与前一节相同的展现。



重点

设计器的用途（特别是这个简化的设计器）不是直接生成用来发布的程序（虽然有运行和保存功能）。实际项目中，设计器的用处主要是用来帮助程序员快速的生成、测试和编辑 UI 代码。

我们用设计器 **è** 在所见即所得的界面上生成 UI（或从其他文件中把 UI 代码 copy 过来）**è** 运行测试一下 **è** 最后把我们要用的代码 copy 出来。

7. 测试通过后，现在可以把里面的代码 copy 出来，放在 designer.grid.html 文件里面：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="Content-Style-Type" content="text/css" />
  <meta http-equiv="imagetoolbar" content="no" />
  <title>Web application powered by LINB framework</title>
</head>
<body>
  <div id="loading"></div>
  <script type="text/javascript" src="../../runtime/jsLinb/js/linb-all.js"></script>
  <script type="text/javascript">
    Class('App', 'linb.Com',{
      Instance:{
        initComponents:function(){
          // [[code created by jsLinb UI Builder
          var host=this, children=[], append=function(child){children.push(child.get(0))};

          append((new linb.UI.TreeGrid)
            .host(host, "treegrid4")
            .setDock("none")
            .setLeft(60)
            .setTop(50)
            .setRowNumbered(true)
            .setGridHandlerCaption("grid")
            .setHeader([{"id":"col 1", "width":80, "type":"label", "caption":"col 1"}, {"id":"col 2",
"width":80, "type":"label", "caption":"col 2"}, {"id":"col 3", "width":80, "type":"label", "caption":"col 3"}])
            .setRows([{"cells":[{"value":"a1"}, {"value":"a2"}, {"value":"a3"}], "id":"j"},
{"cells":[{"value":"b1"}, {"value":"b2"}, {"value":"b3"}], "id":"k"}, {"cells":[{"value":"c1"}, {"value":"c2"},
{"value":"c3"}], "id":"l"}, {"cells":[{"value":"d1"}, {"value":"d2"}, {"value":"d3"}], "id":"m"}, {"cells":[{"value":"e1"},
{"value":"e2"}, {"value":"e3"}], "id":"n"}, {"cells":[{"value":"f1"}, {"value":"f2"}, {"value":"f3"}], "id":"o"}])
          );

          return children;
          // ]]code created by jsLinb UI Builder
        }
      }
    });
    linb.Com.load('App', function(){
      linb('loading').remove();
    });
  </script>
</body>
</html>

```

先显示一个加载条

js 库放在 body 里
加速浏览器响应速度

这个 UI 的类（是从设计器生成的代码中直接 copy 过来的）
可以把它放在一个单独的 js 文件中 à **App/js/index.js**

用这种异步的方法来加载 UI
如果当前没有 App 类存在，系统会
先自动加载 **App/js/index.js** 文件

[cookbook/chapter1/designer.grid.html](#) 文件内容

第五节 有必要了解一下加载过程

在 **designer.grid.html** 里面，我把 UI 类的实现代码都放在了 html 里，一般情况下只有较小的程序才会这么做，对于稍大一些的程序，在开发阶段，把每一个 UI 类的实现代码都按照规则存放在单独的 js 文件中几乎是必须的（发布程序前再把这些 js 文件链接、压缩到一个 js 文件里面 à 通常是 **App/js/index.js**）。

下面我们把 UI 实现代码从 designer.grid.html 分离出来到 `designer.grid.standard.html` 文件和 `App/js/index.js` 文件。designer.grid.standard.html 文件现在是：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="Content-Style-Type" content="text/css" />
  <meta http-equiv="imagetoolbar" content="no" />
  <title>Web application powered by LINB framework</title>
</head>
<body>
  <div id="loading"></div>
  <script type="text/javascript" src="../runtime/jsLinb/js/linb-all.js"></script>
  <script type="text/javascript">
    linb.Com.load('App', function(){
      linb('loading').remove();
    });
  </script>
</body>
</html>
```

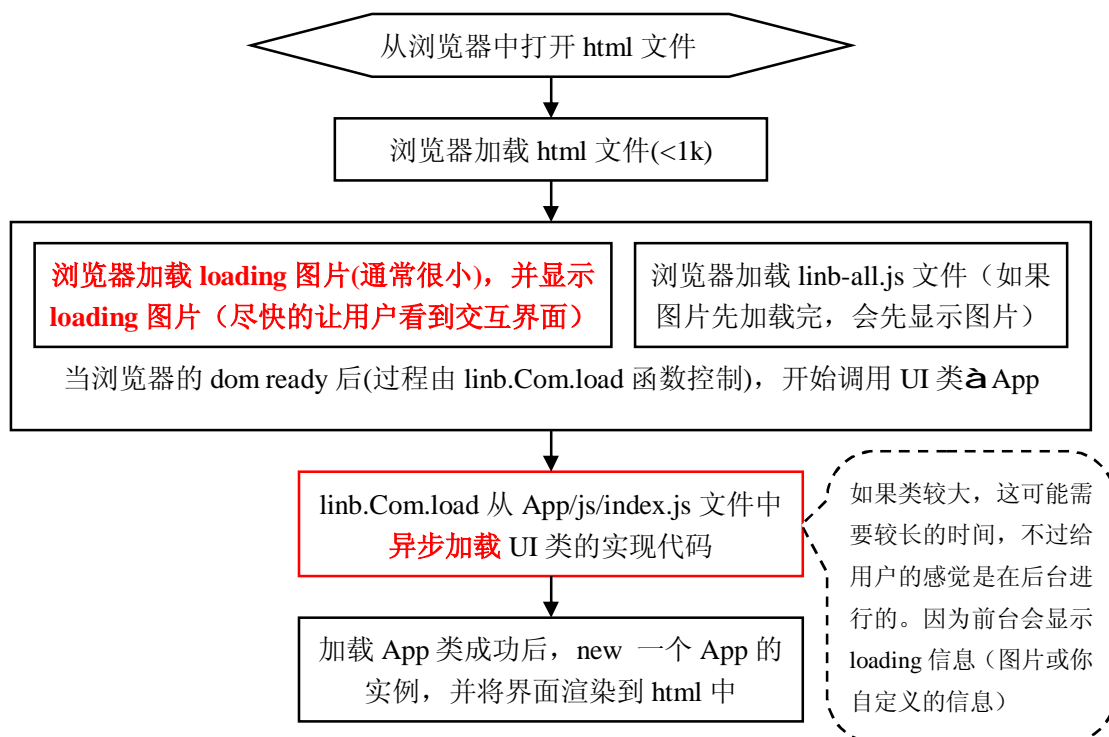
会自动加载 App/js/index.js 文件

最后移除 loading 图片

cookbook/chapter1/designer.grid.standard.html 文件内容

注：这个 html 文件可以作为你应用 linb 的基本 html 模板来使用。

在这里，有必要描述一下 `designer.grid.standard.html` 文件加载的过程：



第六节 再体验一下代码编辑器

虽然 linb 的代码编辑器和 UI 设计器是跨平台运行的，但由于 ie 的性能问题和 opera 对键盘事件的有限支持问题，建议大家最好在 firefox 或 chrome 中用，这样在性能和功能上都会得到最好的保障。

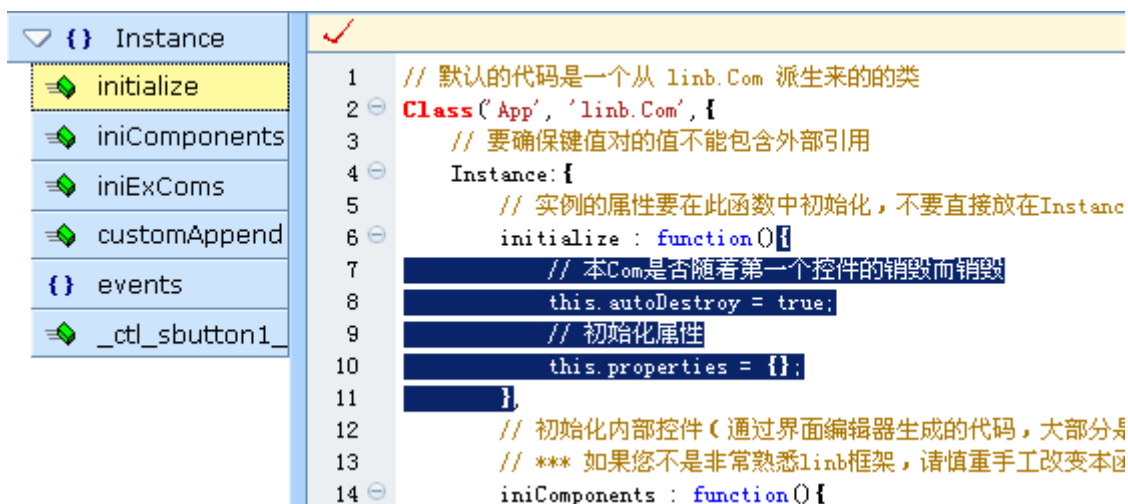


UIBuilder 共有两个页面：“设计器”页面和“代码编辑”页面，简化版的 Builder 默认会到“设计器”页面，点击“代码编辑”项，会切换到“代码编辑”页面。



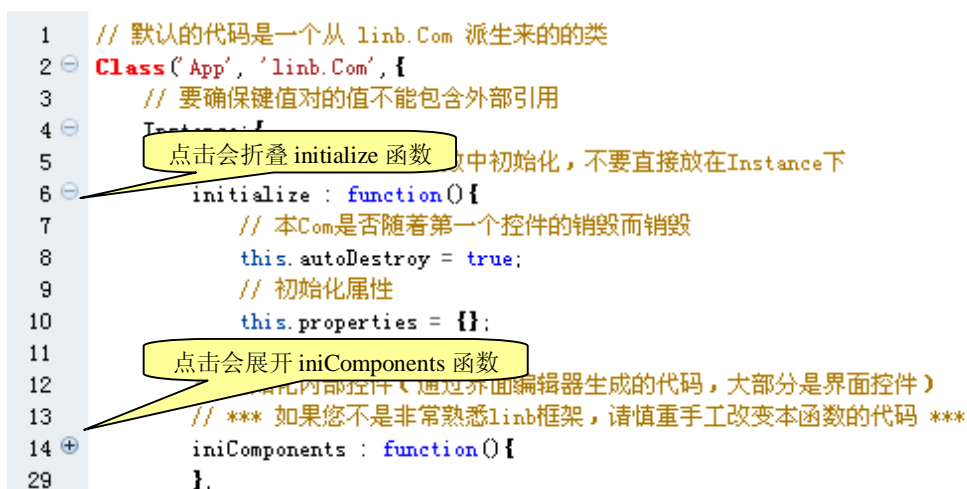
1. 从类结构窗口定位代码

“代码编辑”页面的左侧是树状“类结构”视图，点击“类结构”视图里面的成员或方法名，系统会自动选择在右侧编辑窗口中的相应代码，并滚动编辑窗口到适当的位置。



2. 代码折叠

编辑器支持“代码折叠”功能，点击“编辑窗口”左侧行号区域里的“加号”或“减号”会分别展开或折叠代码。“代码折叠”功能与其他的编辑器操作基本上都是一样的，所以就不在赘述了。



注：由于在浏览器对 iframe 内容的控制效率较低，所以如果你的开发机器性能不太好的话，请尽量不要频繁地折叠或展开大的函数或对象体。

3. 代码智能提醒

目前支持 3 种类型的智能提醒：当输入上下文不识别的字符时；当在编辑器识别的全局变量或局部变量后输入“.”时；当按下快捷键 Alt+1 (或双击选中词条) 时。

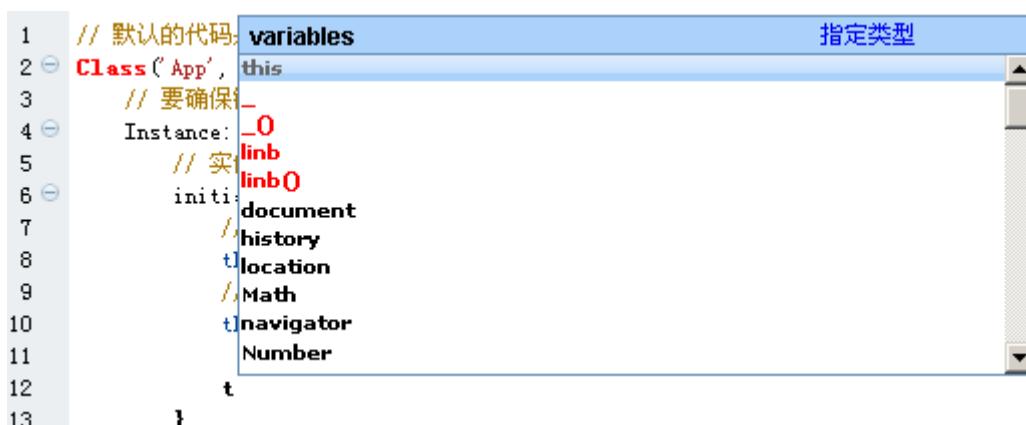
智能提醒窗口弹出后的键盘操作：

- l 用“上箭头”使智能提醒窗口 list 中的选择向上移动；
- l 用“下箭头”使智能提醒窗口 list 中的选择向下移动；

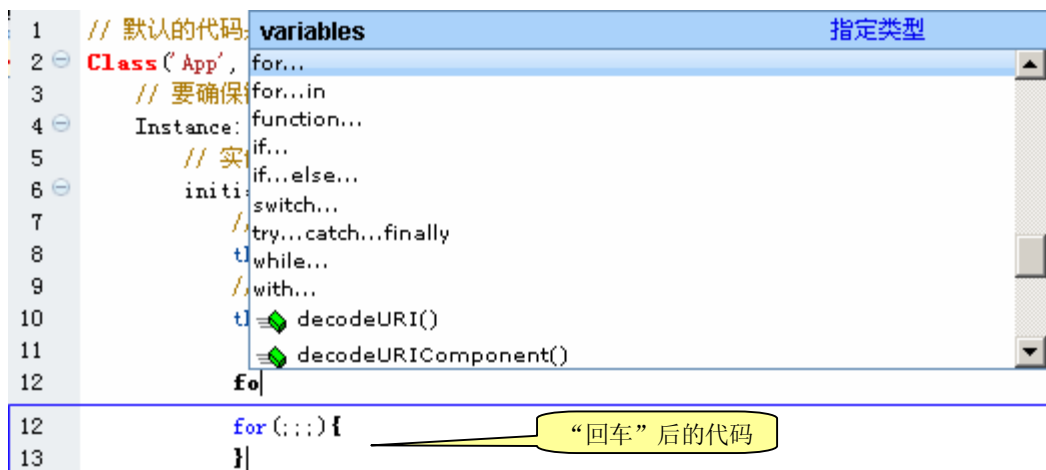
- l 用“回车键”使智能提醒窗口 list 中的当前选择输入到编辑中；
- l 用“ESC 键”关闭智能提醒窗口；
- l 当输入可见字符，编辑器会自动查找到第一个匹配项，并在智能列表中选中。

1) 当输入上下文不识别的字符

在输入上下文不识别的字符或字符串时，编辑器会弹出局部变量、全局变量、系统关键字和全局函数等供选择。在下面窗口中，“this”是局部变量，下面显示出的是全局变量，全局变量下面是系统关键字和全局函数。

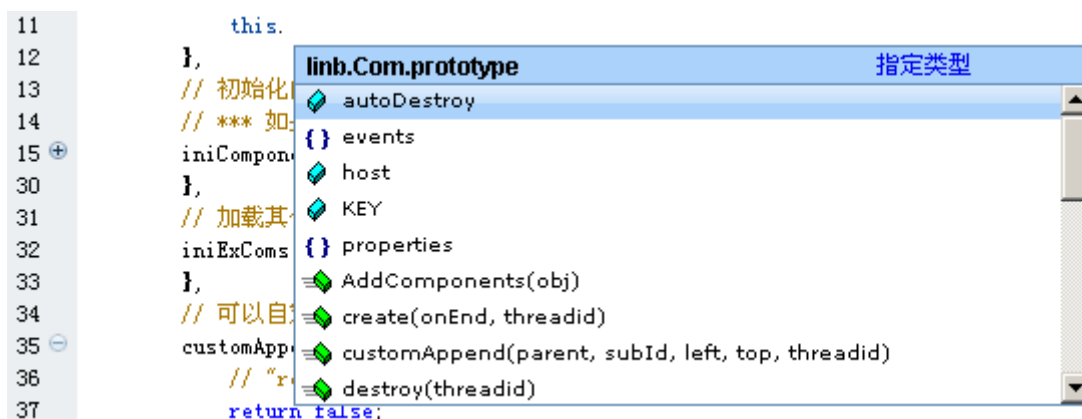


如果此时你输入的是类如“for”这样的 javascript 关键字，编辑器会提供这些代码结构的方便选择。如，在下图的情况下用“回车”，编辑器会自动插入“for”循环的代码。

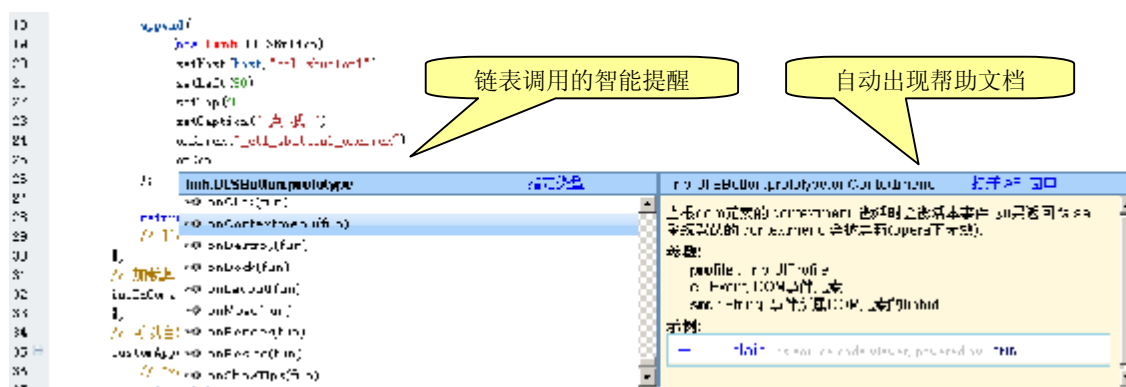


2) 当输入“.”

当在编辑器识别的全局或局部变量后输入“.”，会弹出这个全局或局部变量的成员和方法的智能提醒列表。下面窗口弹出的是当前作用域内 this 的成员和方法的智能提醒列表。

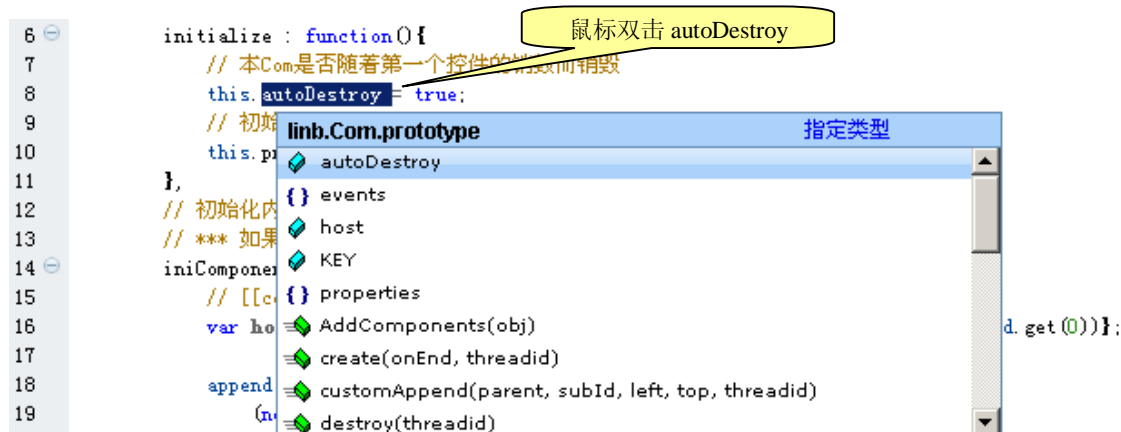


智能提醒不仅限于可以出现在全局或局部变量的后面，链表型调用也可以出现智能提醒，例如下图：



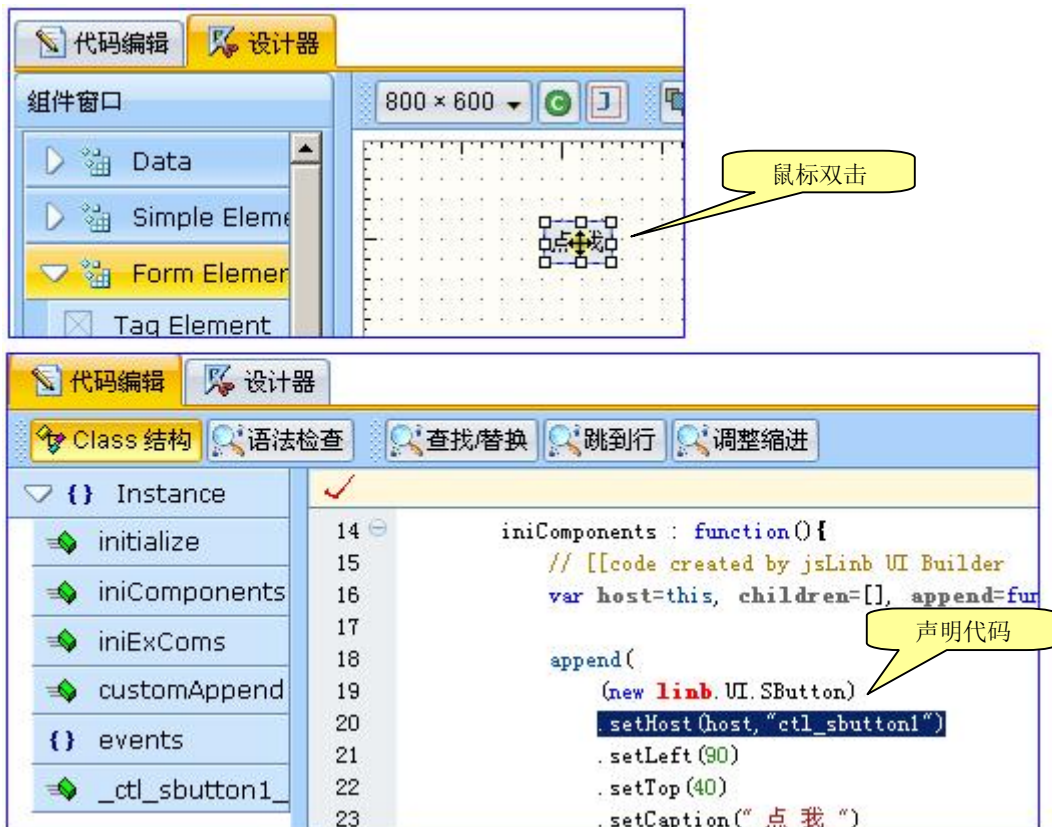
3) 当用快捷键 Atl+1 或双击选中词条

当鼠标位置在一个词条中按下快捷键 Atl+1，或双击选中这个词条时，编辑器会依据这个词条的上下文来判断是否有智能提醒可以显示。



4. 控件切换到定义代码

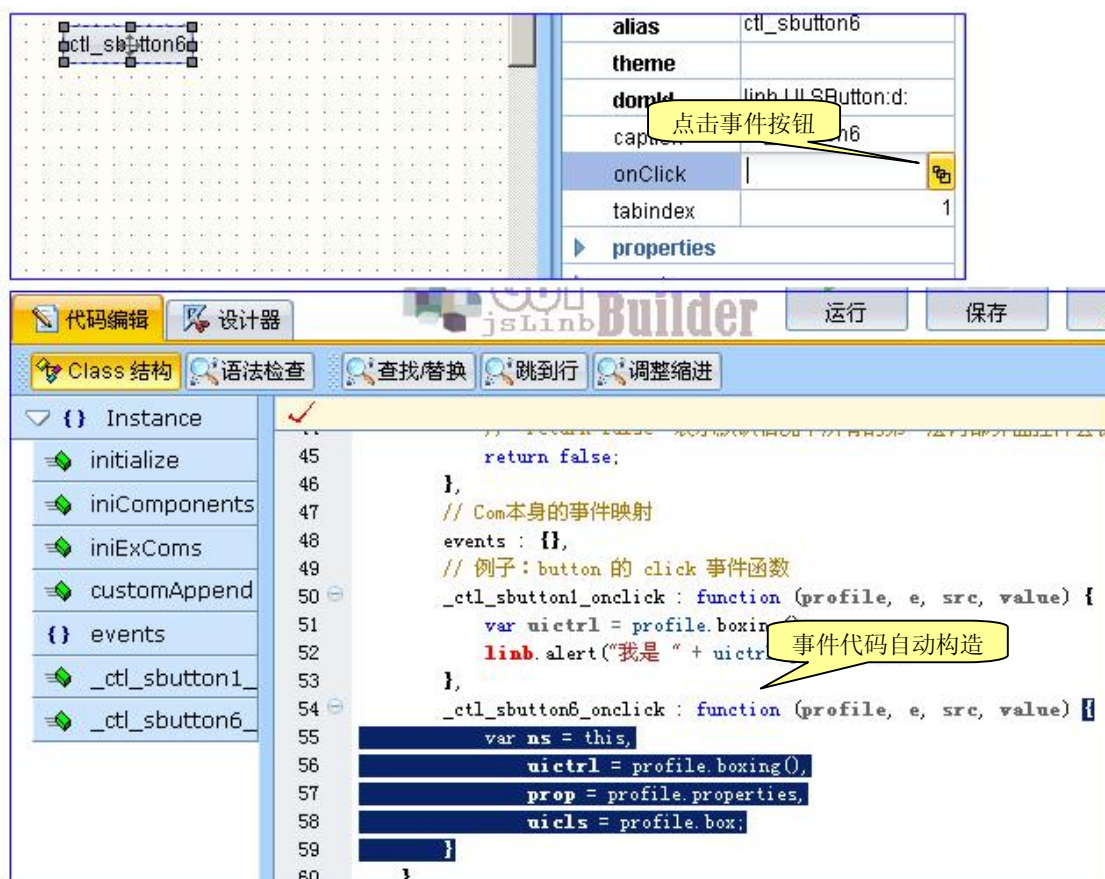
在“设计器”页面，鼠标双击某个控件，会自动切换到“代码编辑”页面，并选中此控件的“声明代码”。



5. 事件代码自动构造

在“设计器”页面，先选中某个控件，在右侧的“组件配置窗口”展开事件，然后用鼠标点击这个事件的按钮，编辑器会自动加入该事件的代码，并切换到“代码编辑”页面，选中事件代码。

如果事件的代码已经存在，点击事件按钮会切换到“代码编辑”页面，并选中这个事件的代码。



第二章 来认识一下控件们

很多初学者都首先对控件特别感兴趣，我们在本章就粗略地过一下 linb 中的基本控件。下面分别对她们做一下简要的介绍。要啰嗦一下，由于每个控件的函数非常多，都讲到的话篇幅会非常的大，所以这里只是简要介绍。每个控件具体的功能可以到 API 上去详细研究！

第一节 搭建脚本测试环境

在此之前，我们先要做一个测试 linb 代码的环境。推荐使用 firefox，如果实在不喜欢 firefox，用 ie8 也可以。

如果是用 **firefox**：

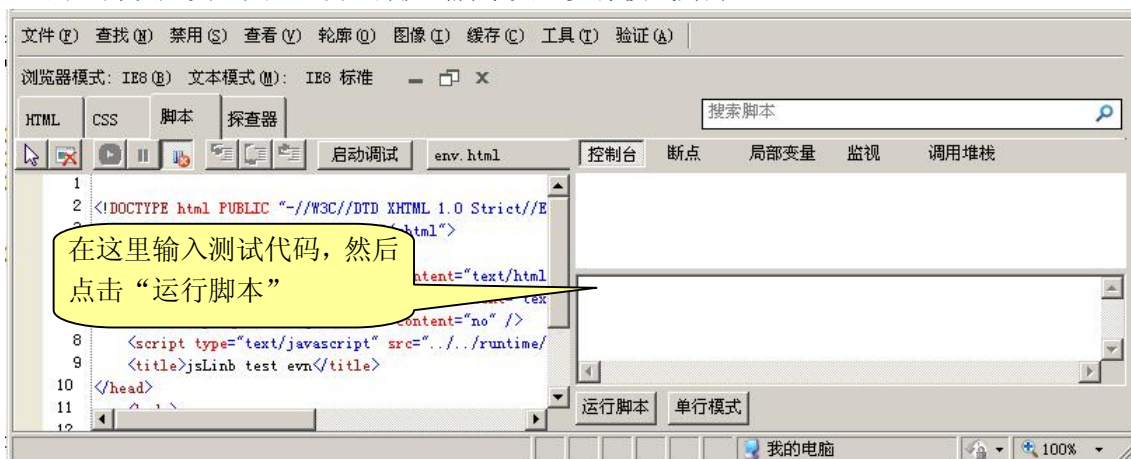
1. 安装 firefox，然后安装 firebug；(这个环境测试比较方便)
2. 如果你是单独下载的教程及示例代码，需要先将 env.html 文件所有的示例文件夹 copy 至 jsLinb 根目录的 **cookbook** 下。
3. 在 firefox 下打开 **cookbook/env.html** 文件（最好不要双击打开，用 http 的方式访问）；
4. 打开 firebug 的控制台（切换到多行输入方式）如下：



如果对 firefox 或 firebug 不熟悉的话，务必先要花些精力熟悉一下，然后再阅读以下的教程。

如果你选择用 IE8:

在 ie8 下打开 [cookbook/env.html](#) 文件（最好不要双击打开，用 http 的方式访问），并开启 ie8 的“开发人员工具”，最后切换到脚本页（多行模式为好）：



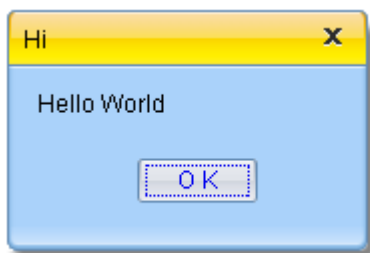
通常，你可以用 [cookbook/env.html](#) 文件中的“全部清除”按钮来清除上一次运行代码的结果。但这在少数情况下是无效的（因为本教程中有些代码带来的不仅仅是对界面的影响），所以，如果有的时候界面看起来有什么不对的地方，需要先刷新一下浏览器。

第二节 在 env.html 中试一下 hello world

打开 env.html，输入并运行以下脚本：

```
linb.alert("Hi", "Hello World!");
```

运行结果同样为：



第三节 产生控件的方式和运行时(runtime)更新

linb 控件产生的方式大概可以分为三类：

```
//Method 1
linb.create("SButton", {
  caption: "Using linb.create function",
  position: "relative"
}).show();

//Method 2
(new linb.UI.SButton({
  caption: "Using new and key/value pairs",
  position: "relative"
})).show();

//Method 3
(new linb.UI.SButton())
.setCaption("Using new and get/set")
.setPosition("relative")
.show();
```

设计器默认的生成方式是 get/set 方式

在“设计时”（控件没有渲染到浏览器之前），用以上 3 种方式生成的控件是完全一致的。需要注意的是，如果**要实现运行时的更新功能**（适用于在控件在渲染到浏览器之后对控件进行更新的需求），就要用 get/set 方法来实现。

这里有必要先提一下 linb 的“运行时”支持，linb 中所有控件的大部分属性都是支持运行时更新的。这里先举个例子：

```
var dlg=linb.create("Dialog", {caption: "runtime "}).show();
_.asynRun(function(){
  dlg.setCaption("updated");
},500);
_.asynRun(function(){
  dlg.setMaxBtn(false);
},1000);
_.asynRun(function(){
  dlg.setStatus("max");
},1500);
_.asynRun(function(){
  dlg.destroy();
},2000);
```

产生一个对话框窗口

运行时改变 caption

运行时改变命令按钮

运行时改变状态

2 秒后销毁

第四节 Button 相关

本节涉及到 `linb.UI.Link`，`linb.UI.SButton`，`linb.UI.Button`，`linb.UI.SCheckBox` 和 `linb.UI.CheckBox`。按钮的作用主要是 `onClick` 事件；checkbox 的作用主要是显示和获取布尔值。

6. 按钮的 onClick 事件

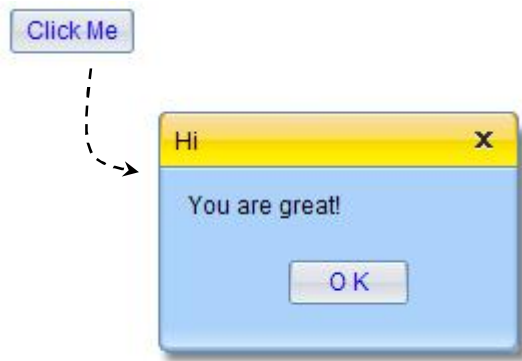
先来个 `linb.UI.SButton` 的：

```
var btn=new linb.UI.SButton();
btn.setCaption("Click Me");
.btn.onClick(function(){
    linb.alert("Hi","You are great!");
});
btn.show();
```

设置 caption

设置 onClick 事件

运行后，点击按钮，结果为：



再来个 `linb.UI.Button` 的：

```
var btn=new linb.UI.Button();
btn.setCaption("Click Me");
.btn.onClick(function(){
    linb.alert("Hi","You are great!");
});
btn.show();

_.asynRun(function(){
    btn.setHeight(80)
    .setShadow(true)
    .setType("drop")
},1000);
```

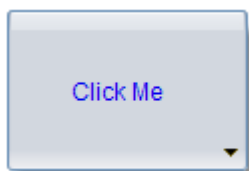
设置 caption

设置 onClick 事件

这些功能 SButton 都没有，还有更多

1 秒后运行这些代码

运行以上代码，看看有什么不同。



可以看到，linb.UI.Button 的功能更加丰富一些。

重点

linb.UI.Button 是一个有着很多功能的复杂控件，如果你只要一个普通按钮的功能用 linb.UI.SButton 就可以了。同样的这也适用于 linb.UI.SLabel/linb.UI.Label 和 linb.UI.SCheckBox/linb.UI.CheckBox。

7. 状态按钮与 CheckBox

在 linb 中，有 3 个控件可以用来表示和编辑布尔值。尝试一下代码（这回我们换一种方式来生成控件）：

```
var btn= (new linb.UI.Button({position: "relative", caption:"Button", type:"status"})).show();
var scb= (new linb.UI.SCheckBox({position: "relative", caption:" SCheckBox"})).show();
var cb= (new linb.UI.CheckBox({position: "relative", caption:" CheckBox"})).show();

_.asRun(function(){
    btn.setValue(true,true);
    scb.setValue(true,true);
    cb.setValue(true,true);
},1000);
```

设置相对位置

1 秒后都设置为 true



项目中具体用哪个控件要看具体要求了。不过原则上如果没有特殊需要的话，**linb.UI.CheckBox 就不要用了。**

8. 更简单的按钮——Link 控件

Link 控件也可以当作一个按钮来用，不过她更简洁，基本上没有什么装饰，外观与 HTML 的 link 差不多。

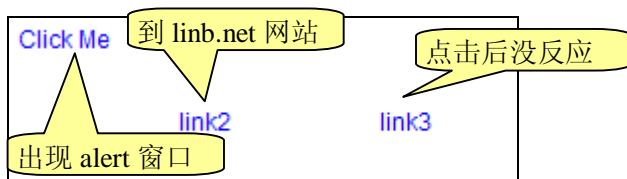
```

var btn=new linb.UILink();
btn.setCaption("Click Me");
.onClick(function(){
    linb.alert("Hi","You are great!");
});
btn.show();

// href property
linb.create("Link",{href: "http://www.linb.net"}).show(null,null,100,100)

// href was disabled when return false
linb.create("Link",{href: "http://www.longboo.com"}).show(null,null,200,100)
.onClick(function(){
    return false;
});

```



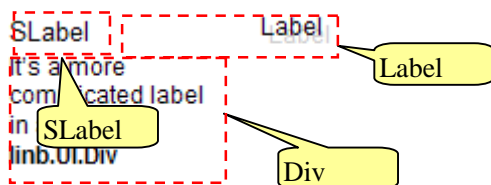
第五节 Label 相关

本节涉及到 `linb.UI.SLabel`, `linb.UI.Label` 和 `linb.UI.Div`。这 3 个控件可以作为标签来使用，不同的是 `linb.UI.Label` 是一个较复杂的控件，一般不会用到，`linb.UI.Div` 一般用在标签的内容比较复杂的情况（例如，标签的内容是一个 `table`）。

```

(new linb.UI.SLabel()).setCaption("SLabel").setPosition("relative").show();
(new linb.UI.Label()).setCaption("Label").setPosition("relative").setShadowText(true).show();
(new linb.UI.Div()).setHtml("It's a more complicated label in a <br /><b>linb.UI.Div</b>")
.setCaption("Div").setPosition("relative").show();

```



总之还是一句话，没什么特殊要求，尽量用 `linb.UI.SLabel`。

第六节 Input 相关

本节涉及到 `linb.UI.Input`, `linb.UI.ComboInput` 和 `linb.UI.RichEditor`。`ComboInput` 是 `Input` 的加强版（可以通过弹出窗口来选择值等）；`RichEditor` 是用来辅助输入富文本的控件。

2. 得到和设置输入值

从用户的角度说，linb 的值控件（所有派生自 `linb.absValue` 的控件）有两个值：一个是“界面临时值”，另一个是“控件值”。“界面临时值”和“控件值”可以相同也可以不相同。例如，对于一个初始值为空的 `Input` 控件来说，当用户输入值“abc”后，“界面临时值”为“abc”（此时“控件值”为空）；如果对这个 `Input` 调用代码 `updateValue()`，“界面临时值”就会 copy 到“控件值”（此时“界面临时值”和“控件值”都为“abc”）。

“界面临时值”的 `get/set` 函数是 `getUIValue()` / `setUIValue()`；“控件值”的 `get/set` 函数是 `getValue()` / `setValue()`。

```
var input = (new linb.UI.Input()).show();
linb.message(input.getUIValue()+"."+input.getValue());
_.asynRun(function(){
  input.setUIValue('uivalue');
  linb.message(input.getUIValue()+"."+input.getValue());
},2000);
_.asynRun(function(){
  input.updateValue();
  linb.message(input.getUIValue()+"."+input.getValue());
},4000);
```

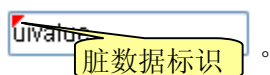
新建一个 Input

模仿键盘输入

更新界面值到真实值

3. 脏数据标识

如果 `dirtyMark` 属性为 `true`，当控件 UI 临时值和控件内部值不相等的时候，`Input` 控件就会显示脏数据标识，默认的脏标识是控件在的左上角有一个红色的小三角形：



```
var input = (new linb.UI.Input()).show();
_.asynRun(function(){
  input.setUIValue('uivalue');
},1000);
_.asynRun(function(){
  input.updateValue();
  input.setDirtyMark (false);
},2000);
_.asynRun(function(){
  input.setUIValue('uivalue 2');
},3000);
```

这时候会显示脏标识

脏标识会消失

取消脏标识功能

脏标识不会再出现

linb 的值控件（派生自 `linb.absValue` 的控件）都有 `dirtyMark` 属性。当不想要“脏标识”的时候，可以用 `setDirtyMark(false)`来达到目的。

4. 密码框

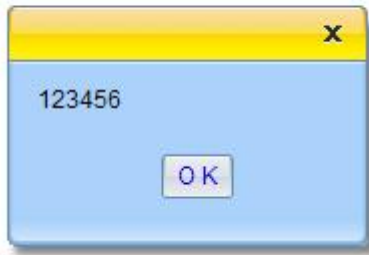
只需要把 `Input` 控件的 `type` 属性设置为“password”就可以得到密码框。

```
var input = (new linb.UI.Input({type: 'password'})).show();
```

```
_.asynRun(function(){
  input.setUIValue('123456').updateValue();
  linb.pop(input.getValue());
},1000);
```

密码框

模仿键盘输入,并刷新控件值

5. 多行输入

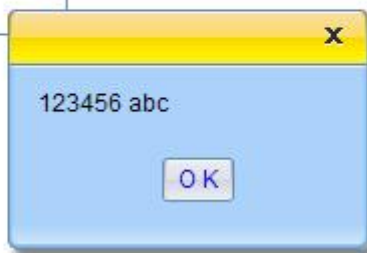
只需要把 Input 控件的 multiLine 属性设置为 true 既可以得到多行输入框（对应于 html 的 textarea）。

```
var input = (new linb.UI.Input()).setMultiLines(true).setHeight(100).show();
```

```
_.asynRun(function(){
  input.setUIValue('123456 \n abc').updateValue();
  linb.pop(input.getValue());
},1000);
```

设置多行输入

模仿键盘输入,并刷新控件值

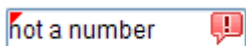
6. 输入验证

有两种方式可以实现输入验证，第一个是用 valueFormat 属性：


```
var input = (new linb.UI.Input())
    .setValueFormat("^-?\\d\\d*$")
    .show();
```

只能输入数字

运行以上代码，如果你在在输入框中输入的内容不是数字，在焦点离开后，输入框会有验证出错图标出现：



valueFormat 属性是一个代表正则表达式的字符串。如果你用设计器的话，linb 设计器里面自带了一些常用的正则表达式供选择。

第二种方法是用 beforeFormatCheck 事件：

```
var input = (new linb.UI.Input())
    .beforeFormatCheck(function(profile,value){
        if(value!=parseFloat(value).toString())
            return false;
    })
    .show();
```

只能输入数字

在这两种方法中，beforeFormatCheck 优先。也就是当 beforeFormatCheck 返回 false 后就不会再用 valueFormat 属性来验证输入了。

7. 动态输入验证

上面的输入验证只有在鼠标焦点离开 Input 的时候才会触发，如果想要在输入的同时实时触发输入验证，需要设置 dynCheck 属性为 true：

```
var input = (new linb.UI.Input())
    .setDynCheck(true)
    .setValueFormat("^-?\\d\\d*$")
    .show();
```

设置动态验证

8. 输入验证消息的显示方式

1) 验证出错图标

默认的，当输入验证出现错误的时候，系统显示的消息为：



表示输入验证错误

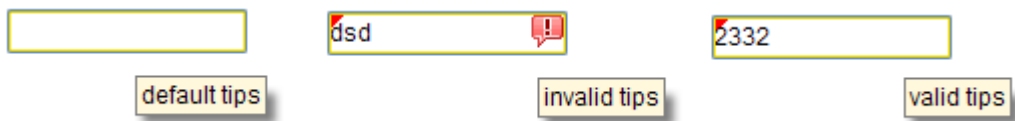
2) 验证提示信息

Input 有 3 个 tips 属性: tips, tipsOK 和 tipsErr, 分别是“默认的提示”, “输入验证通过时的提示”和“输入验证错误时的提示”。运行以下代码:

```
var input = (new linb.UI.Input())
    .setTips("default tips")
    .setTipsErr("invalid tips ")
    .setTipsOK("valid tips")
    .setValueFormat("^-?\d\d*$")
    .show();
```

设置这些提示信息

在 3 中状态（默认，验证出错，验证通过）下分别得到:



3) 验证信息绑定

上面的 tips 信息可以绑定到一个 linb.UI.Span, linb.UI.Div 或 linb.UI.SLabel 中显示:

```
var slbl= (new linb.UI.SLabel({position:'relative'})).setCustomStyle({KEY:'padding-left:10px'});
var input = (new linb.UI.Input({position:'relative'}))
    .setValueFormat("^-?\d\d*$")
    .setTipsBinder(slbl)
    .setDynCheck(true)
    .setTips(" default tips")
    .setTipsErr(" invalid tips ")
    .setTipsOK(" valid tips")

input.show();
slbl.show();
```

设置 tipsBinder

SLabel 显示在后面



4) 自定义验证信息

通过 beforeFormatMark 事件, 我们也可以自定义验证信息:

```
var input = (new linb.UI.Input())
    .setValueFormat("^-?\\d\\d*$")
    .beforeFormatMark(function(profile,err){
        if(err)
            linb.alert("Invalid input!", "Only number allowed!", function() {
                profile.boxing().activate();
            });
        return false;
    }).show();
```

自定义信息，并再次设置焦点

返回 false

如果输入非数字信息，我们会得到以下效果（不再显示验证出错图标）：



9. 掩码输入——mask input

以下是掩码输入的效果图：

| | | | |
|-----------|--|----------------|---|
| 111111111 | <input type="text" value="111111111"/> | (111) 111-1111 | <input type="text" value="() -"/> |
| ~1.11 | <input type="text" value="111111111"/> | (111) a-a *\$* | <input type="text" value="() - _ \$"/> |

Mask 属性 显示结果

掩码输入由 mask 属性控制，是一个字符串。在这个字符串中： '~'代表正号或负号（[+-]）； '1'代表从 0 到 9 的数字（[0-9]）； 'a'代表英文字母（[A-Za-z]）； '*'代表英文字母或数字（[A-Za-z0-9]）。其他的字母都代表固定输入。

下面是个电话号码掩码输入的例子：

```
var input = (new linb.UI.Input())
    .setMask("(1111)11111111-11")
    .show();
```


以上是 linb.UI.Input 的部分重要功能，篇幅所限，到此打住。下面来了解一下 linb.UI.Input 的一个子类 linb.UI.ComboInput。

注释： 在 chapter2\Input 文件夹下有一个由设计器辅助生成的 linb.UI.Input 的综合例子。

10. 终于轮到 ComboInput

linb.UI.ComboInput 是 linb.UI.Input 的子类，所以重复的功能就不再讲了。

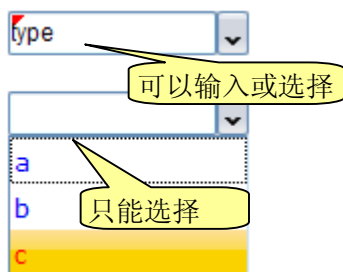
1) 列表选择

设置 type 属性为 combobox、listbox 或 helpinput 表示会弹出列表选择窗口。其中 combobox / helpinput 的默认为输入框可编辑状态。

```
(new linb.UI.ComboInput({items: ["a", "b", "c"]})).show();
(new linb.UI.ComboInput({type: "listbox", top: "100", items: ["a", "b", "c"]})).show();
```

默认是 combobox

listbox 的输入框只读



2) combobox、listbox 和 helpinput 的区别

分辨这三种类型首先要弄清楚 items 属性的结构。通常我们输入的 item 是类如 ["ia", "ib", "ic"] 的一个字符串数组，但其实这个数组在 js 变量中被格式化为如下形式：

```
[
  {
    id: "ia",
    caption: "ia"
  },
  {
    id: "ib",
    caption: "ib"
  },
  {
    id: "ic",
    caption: "ic"
  }
]
```

按照这个格式化后的结构，以下三种类型的区别为：

- combobox: 列表显示的是 caption 值，输入框显示的是 caption 值，getValue 得到的是 caption 值。（可以输入，也可以选择，所以叫 combobox）
- listbox: 列表显示的是 caption 值，输入框显示的是 caption 值，getValue 得到的是 id 值。（只可以从列表中选择，所以叫 listbox）

- 1 helpinput: 列表显示的是 caption 值，输入框显示的是 id 值，getValue 得到的是 id 值。（列表中显示的是“辅助输入的文字”，所以叫 helpinput）

```
var items=[
  {
    id : "id1",
    caption : "caption1"
  },{
    id : "id2",
    caption : "caption2"
  },{
    id : "id3",
    caption : "caption3"
  }
];
linb.create('ComboInput',{position:'relative',items:items}).show();
linb.create('ComboInput',{position:'relative',items:items,type:'listbox'}).show();
linb.create('ComboInput',{position:'relative',items:items,type:'helpinput'}).show();
```

以上代码运行后，手工地把三个 ComboInput 都选择弹出 list 中的第 3 项，屏幕显示为：



此时，这 3 个的实际“界面临时”值为：“caption3”，“id3”和“id3”。

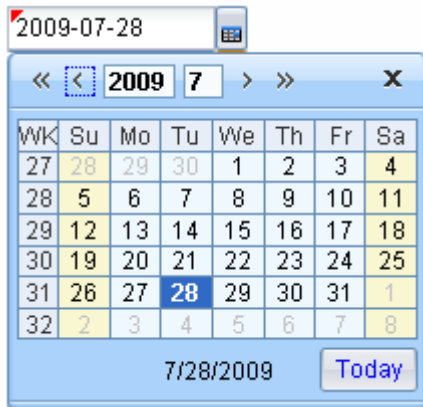
3) 日期选择

当 type 属性设置为 date 时，ComboInput 就是一个日期选择控件。

```
var ctrl=linb.create('ComboInput')
.setType('date')
.setValue(new Date)
.show();

_.asRun(function(){
  alert("The value is a timestamp string:"+ctrl.getValue());
  alert("You can convert it to date object:"+new Date(parseInt(ctrl.getValue())));
});
```

可以设置日期对象，字符串或是数字



4) 时间选择

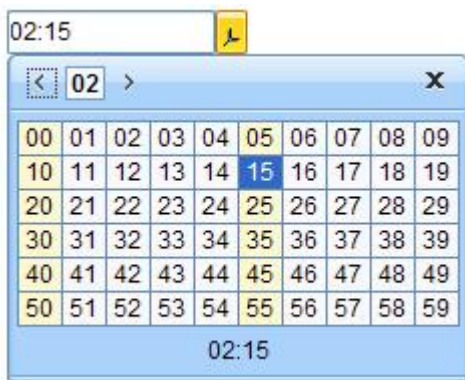
当 type 属性设置为 time 时，ComboInput 就是一个时间选择控件。

```
var ctrl=linb.create('ComboInput')
.ctrl.setType('time')
.ctrl.setValue('2:15')
.ctrl.show();
```

设置字符串

得到的还是字符串

```
linb.alert("The value is a string : "+ctrl.getValue());
```



5) 颜色选择

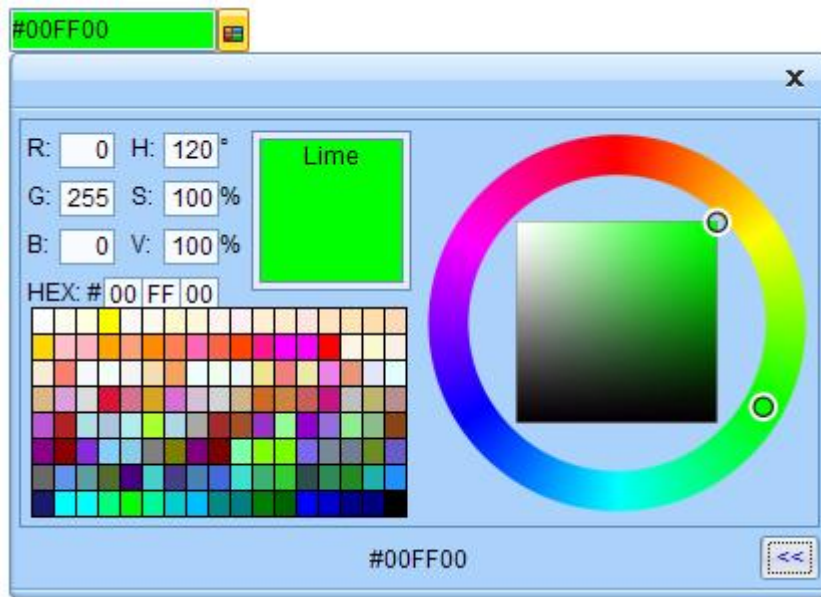
当 type 属性设置为 color 时，ComboInput 就是一个颜色选择控件。

```
var ctrl=linb.create('ComboInput')
.ctrl.setType('color')
.ctrl.setValue('#00ff00')
.ctrl.show();
```

设置字符串

得到的还是字符串

```
linb.alert("The value is a string : "+ctrl.getValue());
```



6) 文件选择

当 type 属性设置为 upload 时，ComboInput 就是一个文件选择控件。

```
var ctrl=linb.create('ComboInput')
  .setType('upload')
  .show();
```



在选择完文件后，要用 getUploadObj 方法来得到文件的句柄。

```
ctrl.getUploadObj)
```

7) 赋值命令

当 type 属性设置为 getter 时，ComboInput 就是一个赋值控件。

```
var ctrl=linb.create('ComboInput')
  .setType('getter')
  .beforeComboPop(function(profile){
    profile.boxing().setUIValue(_id())
  })
  .show();
```

用 beforeComboPop 事件来赋值

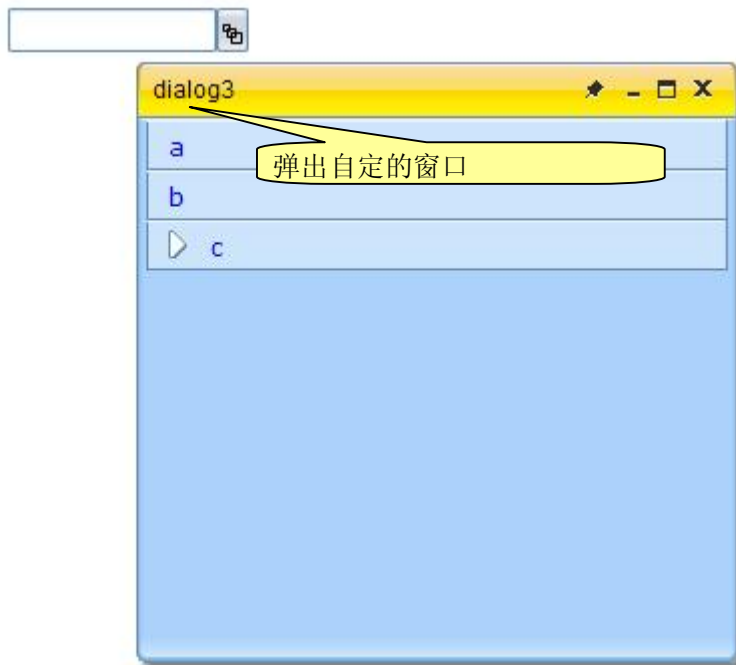


8) 自定义选择

当 type 属性设置为 cmdbox 或 popbox 时，ComboInput 就是一个自定义选择控件。

```
var ctrl=linb.create('ComboInput')
.setType('popbox')
.beforeComboPop(function(profile){
    var dlg=new linb.UI.Dialog, tb;
    dlg.append(tb=new linb.UI.TreeBar({items:["a", "b",{id:"c",sub:["c1", "c2", "c3"]}]}));
    tb.onItemSelected(function(profile,item){
        ctrl.setUIValue(item.id);
        dlg.destroy();
    });
    dlg.show(null,true,100,100)
})
.show();
```

用 beforeComboPop 事件来实现

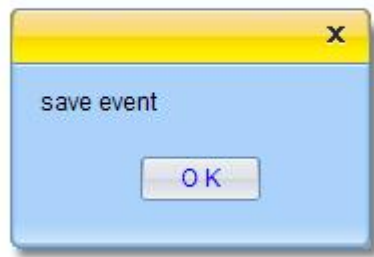


9) comboInput 中的命令按钮

用 commandBtn 属性，可以给 ComboInput 设置一个命令按钮。命令按钮可以是：

- ! “none”：没有按钮出现
- ! “save”：显示保存图标
- ! “add”：显示添加图标
- ! “remove”：显示移除图标
- ! “delete”：显示删除图标
- ! “custom”：显示自定义图标。自定义图标由 image 和 imagePos 属性来指定。


```
(new linb.UI.ComboInput).setPosition('relative').setCommandBtn('none').show();  
(new linb.UI.ComboInput).setPosition('relative').setCommandBtn('save').onCommand(function(){ linb.alert('save event'); }).show();  
(new linb.UI.ComboInput).setPosition('relative').setCommandBtn('add').onCommand(function(){ linb.alert('add event'); }).show();  
(new  
linb.UI.ComboInput).setPosition('relative').setCommandBtn('remove').onCommand(function(){ linb.alert('remove event'); }).show();  
(new linb.UI.ComboInput).setPosition('relative').setCommandBtn('delete').onCommand(function(){ linb.alert('delete event'); }).show();  
(new linb.UI.ComboInput).setPosition('relative').setCommandBtn('save').onCommand(function(){ linb.alert('save event'); }).show();
```

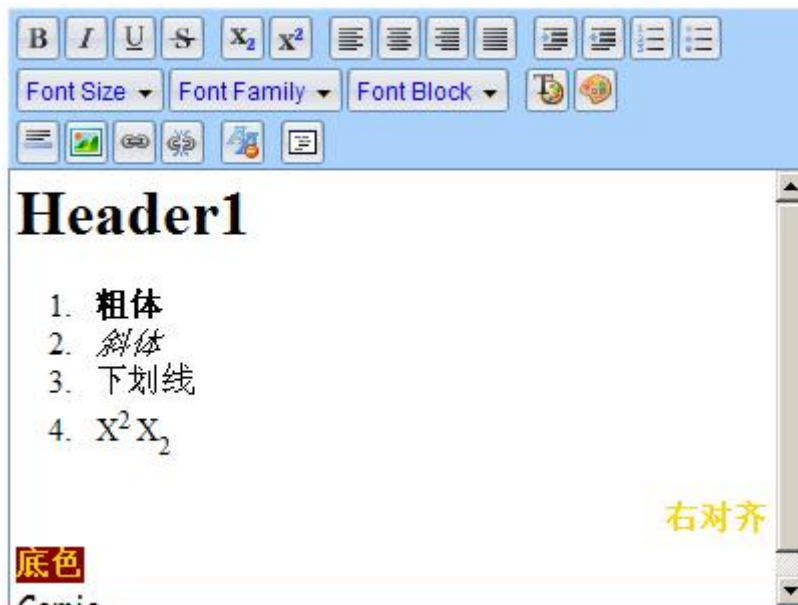


注释：在 chapter2\ComboInput 文件夹下有一个由设计器辅助生成的 linb.UI.ComboInput 的综合例子。

11. 富文本编辑框 RichEditor

运行一下代码：

```
(new linb.UI.RichEditor()).show();
```



富文本编辑器的各个按钮的命令在这里就不详细介绍了。

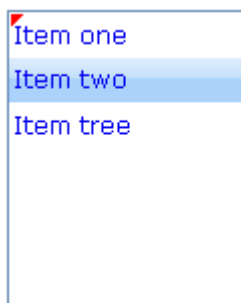
第七节 List 相关

本节涉及到 `linb.UI.List`, `linb.UI.RadioBox`, `linb.UI.IconList` 和 `linb.UI.Gallery`。后三个控件是第一个控件 `List` 的子类。

1. 简单列表

```
linb.create("List")
.setItems(["Item one", "Item two", "Item three"])
.onItemSelected(function(profile, item){
    linb.message(item.id);
})
.show();
```

选择的时候触发事件



2. 稍微复杂一点的

```
var renderer=function(o){
    return '<span style="width:40px">'+o.col1+'</span>' + ' <span style="width:60px">'+o.col2+'</span>' +
    '<span style="width:40px">'+o.col3+'</span>';
};

linb.create("List")
.setWidth(160)
.setItems([[
    {
        id:"a",
        col1:'Name',
        col2:'Gender',
        col3:'Age',
        renderer:renderer,
        itemStyle:'border-bottom:solid 1px #C8E1FA;font-weight:bold;'
    },
    {
        id:"b",
        col1:'Jack',
        col2:'Male',
        col3:'23',
        renderer:renderer
    },
    {
        id:"c",
        col1:'Jenny',
        col2:'Female',
        col3:'32',
        renderer:renderer
    }
]])
.beforeUIValueSet(function (profile, ov, nv){
    return nv!="a"
})
.show();
```

设置一个 render 函数

附加的变量

运行结果为：

| Name | Gender | Age |
|-------|--------|-----|
| Jack | Male | 23 |
| Jenny | Female | 32 |

List 模拟表格

上面用到的 `renderer` 函数可以应用在任何一个有 `caption` 属性（如 `Button`, `Label`），或子项里面有 `caption` 属性（如 `List`, `TreeBar`）的控件上。在控件渲染到 DOM 之前，`renderer` 函数返回的字符串会作为 `caption` 来使用。例如，

```
linb.create("SCheckBox")
.setCaption("caption")
.setRenderer(function(prop){return prop.caption+"-"+this.key})
.show();
```

当前的 scope 是控件的概要对象，
传入的第一个参数是属性键值对

 caption+linb.UI.SCheckBox

3. RadioBox

linb.UI.RadioBox 是 linb.UI.List 的子类。直接把 linb.UI.List 的第一个例子改成 linb.UI.RadioBox 的：

```
linb.create("RadioBox")
.setItems(["a","b","c"])
.onItemSelected(function(profile,item){
    linb.message(item.id);
})
.show();
```



4. IconList 和 Gallery

这两个也都是 linb.UI.List 的子类。

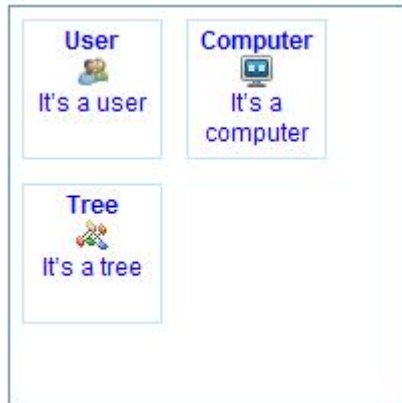
```
linb.create("IconList")
.setItems([{id:'a',image:'img/a.gif'}, {id:'b',image:'img/b.gif'}, {id:'c',image:'img/c.gif'}])
.onItemSelected(function(profile,item){
    linb.message(item.id);
})
.show();
```



IconList 一般用在列出小图标的情况，Gallery 一般用在列出较大的图标的情况，并且可以带有 caption 和 comment 信息。

```
linb.create("Gallery")
.setItemWidth(64).setItemHeight(64)
.setItems([{id:'a',image:'img/a.gif',caption:'User',comment:'It's a user'},
{id:'b',image:'img/b.gif',caption:'Computer',comment:'It's a computer'},
{id:'c',image:'img/c.gif',caption:'Tree',comment:'It's a tree'}])
.onItemSelected(function(profile,item){
    linb.message(item.id);
})
.show();
```

设置项的大小



5. 代码控制条目选择

可以用 `setUIValue` 来选择 `absList` 的条目，也可以用 `fireItemClickEvent` 来选择。两者的主要区别是 `fireItemClickEvent` 是完全模拟用户用鼠标来点击 `ITEM` 节点，会触发 `onItemSelected` 事件，`setUIValue` 只会改变界面的选择，不会触发 `onItemSelected` 事件。

```
var ctrl=linb.create("List")
.setItems(["Item one","Item two","Item tree"])
.onItemSelected(function(profile,item){
    linb.message(item.id);
})
.show();

_.asynRun(function(){
    ctrl.fireItemClickEvent("Item two");
},1000);

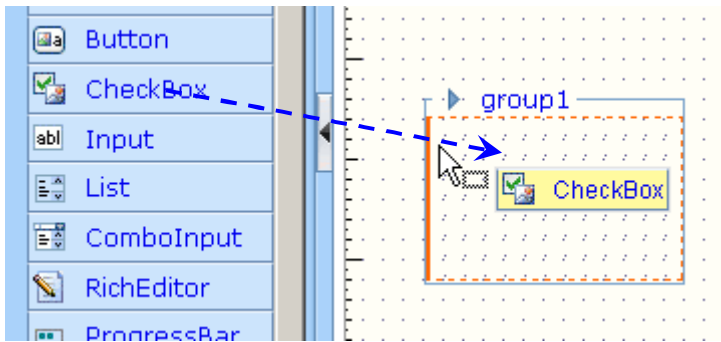
_.asynRun(function(){
    ctrl.setUIValue("Item one");
},2000);
```

会触发 `onItemSelected` 事件

第八节 容器相关

本节涉及到容器控件 `linb.UI.Group`，`linb.UI.Pane`，`linb.UI.Panel` 和 `linb.UI.Block`。其他两个容器控件 `linb.UI.Dialog`、`linb.UI.Layout` 和 `linb.UI.Tabs`(还有其子类 `Stacks` 和 `ButtonViews`) 会在下几节中讲到。

容器控件就是那些可以有子控件的控件。在 `linb` 设计器中，你可以用拖拽的方式把其他空间拖拽到容器控件的容器面板内：

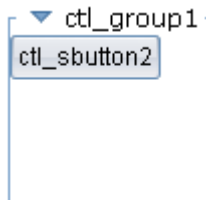


在实际的项目中，推荐用 linb 设计器来快速布局容器内的控件。

在代码层面，把一个控件加入容器控件用 `append` 方法：

```
(new linb.UI.Group)
.append(new linb.UI.SButton)
.show();
```

用 `append` 方法



也可以用这种方法，

```
var con = new linb.UI.Group;
con.show();
(new linb.UI.SButton).show(con);
```

用 `show` 方法

还有，

```
linb.create({
  key: "linb.UI.Group",
  children: [{key: "linb.UI.SButton"}]}
}).show();
```

用概要对象的结构来创建

1. Pane 和 Panel 的区别

`linb.UI.Pane` 只有一个单独的 DOM 节点，可以看作是给 `linb.UI.Div` 加上了容器的功能。`linb.UI.Panel` 比 `Pane` 多了个装饰框和一个标题栏。

```
(new linb.UI.Pane)
.append(new linb.UI.SButton)
.show()
```

运行结果：Pane 是透明无边框的

```
(new linb.UI.Panel)
.setDock("none")
.append(new linb.UI.SButton)
.show()
```

Panel 的默认 dock 是 'fill'



2. Pane 和 Block 的区别

linb.UI.Block 除了比 linb.UI.Panel 多了一个边框外，linb.UI.Block 还可以有 border, resizer 和 shadow。

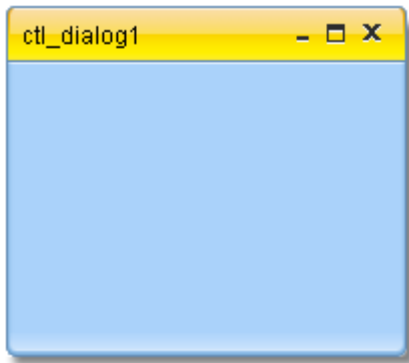
```
linb.create("Block",{position:'relative',borderType:'none'}).show()
linb.create("Block",{position:'relative',borderType:'flat'}).show()
linb.create("Block",{position:'relative',borderType:'inset'}).show()
linb.create("Block",{position:'relative',borderType:'outset'}).show()
linb.create("Block",{position:'relative',borderType:'groove'}).show()
linb.create("Block",{position:'relative',borderType:'ridge'}).show()
linb.create("Block",{position:'relative',borderType:'none',border:true,shadow:true,resizer:true}).show()
```



第九节 对话框

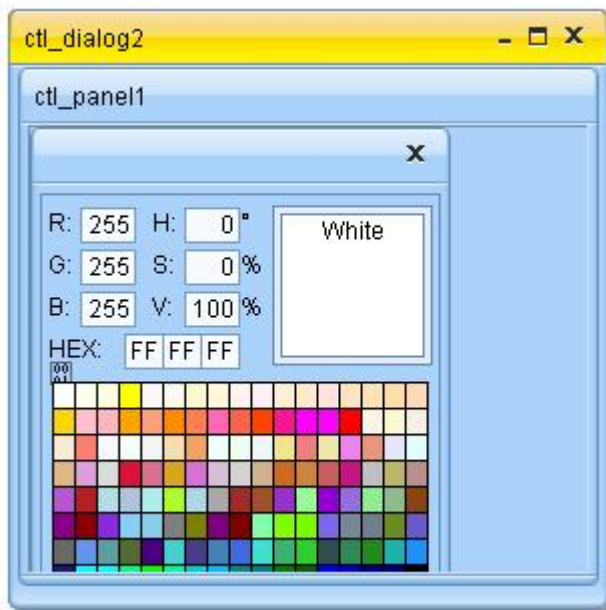
1. 最普通的

```
(new linbUI.Dialog).show()
```



容器功能：

```
var dlg = (new linb.UI.Dialog).show();
var panel;
_._asynRun(function(){
    dlg.append(panel=new linb.UI.Panel)
},1000);
_._asynRun(function(){
    panel.append(new linb.UI.ColorPicker)
},2000);
```



2. 最大化、最小化

```
var dlg = (new linb.UI.Dialog).setStatus("min").show();
_._asynRun(function(){
    dlg.setStatus("normal");
},1000);
_._asynRun(function(){
    dlg.setStatus("max");
},2000);
```




3. 模式对话框

```
var dlg = (new linb.UI.Dialog).show();
dlg.append(panel=new linb.UI.SButton({
  caption: "Pop a modal dialog"
}),
{onClick:function(){
  linb.create("Dialog",{
    width:200,
    height:100,
    html:"The second modal dialog"
  }).showModal(dlg);
}}
))

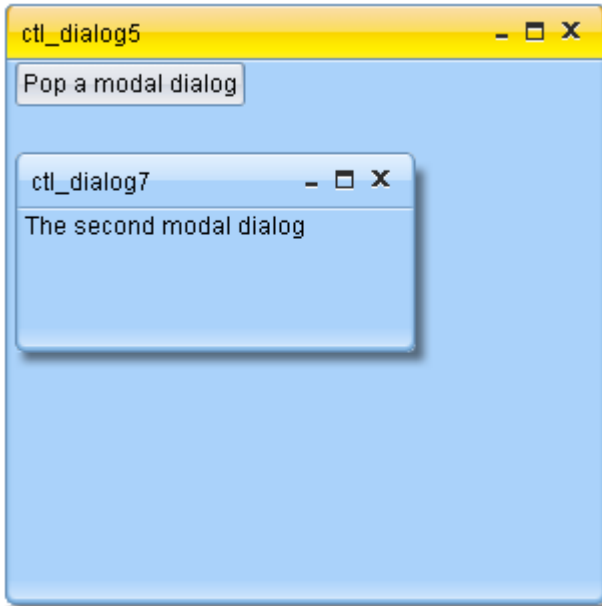
(new linb.UI.Dialog)
.setHtml("The first modal dialog")
.show(null,true);
```

属性键值对

事件键值对

父窗口是 dlg

父窗口是 Dom 的 body



第十节 布局控件

linb.UI.Layout 是布局控件，布局控件的默认方向是垂直的。

```

var block=linb.create("Block").setWidth(300).setHeight(300);
var layout=linb.create("Layout",{items:[
  {id:'before',
    pos:'before',
    size:100,
    cmd:true
  },
  {id:'after',pos:'after',size:100}
]});
block.append(layout).show();

```

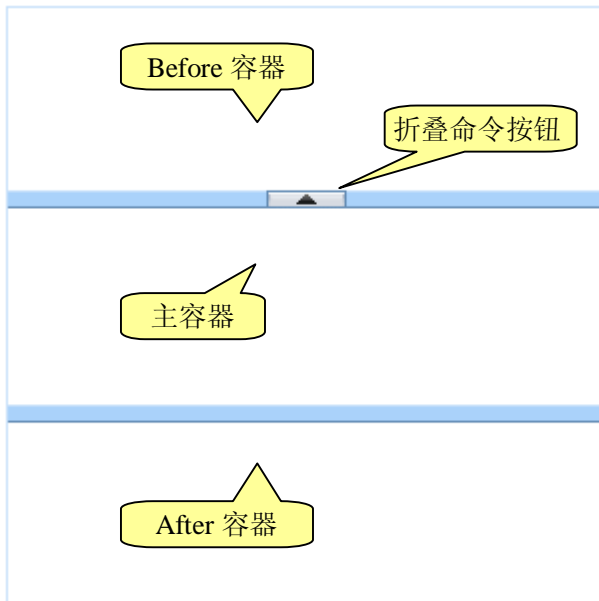
位置在前面

默认大小是 100

有折叠命令按钮

放在一个 Block 里面

运行结果为：



水平方向的需要把 type 属性设置为 “horizontal”：

```

var block=linb.create("Block").setWidth(400).setHeight(100);
var layout=linb.create("Layout",{items:[
  {id:'before',
    pos:'before',
    size:100,
    cmd:true,
    folded:true,
    max:120,
    min:80
  },
  {
    id:'after',
    pos:'after',
    cmd:true,
    locked:true,
    size:50
  },
  {
    id:'after2',
    pos:'after',
    size:50
  }
]}.type: 'horizontal');
block.append(layout).show();

```

默认的容器面板是折叠的

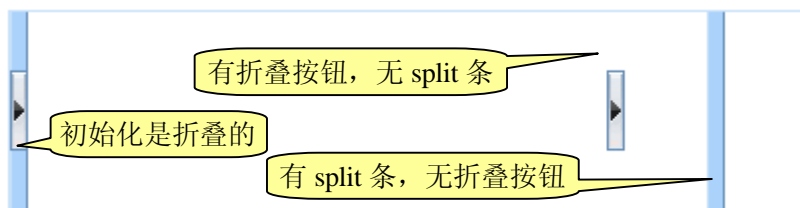
容器面板的最大值 120

容器面板的最小值 80

去掉可拖拽的 split 条

水平方向

运行结果为：



注释：在 chapter2/Layout 文件夹下有一个由设计器辅助生成的 linb.UI.Layout 的综合例子。

第十一节 多页控件

多页控件有 3 个：linb.UITabs，linb.UI.Stacks 和 linb.UI.ButtonViews。这 3 个控件的用法和 API 基本一样。

```
var block=linb.create("Block").setWidth(400).setHeight(100);
var pages=linb.create("Tabs",{
  items:["page1","page2","page3"],
  value:"page2"
});
block.append(pages).show();

_._asRun(function(){
  pages.append(new linb.UI.SButton,"page2")
},1000);
```

3 个页

放在一个 Block 里面

默认的页面

加一个 SButton 到第二个页

1. 是否自带容器面板

分页控件可以设置 noPanel 属性来决定是否自带容器面板。默认是自带容器面板的：一个页对应着一个容器面板。当 noPanel 属性设置为 true 的时候，分页控件没有自带容器面板，这意味着你就不要再调用 append 方法来加入子控件了。注意：linb.UI.Stacks 一定是要自带容器面板的，她没有 noPanel 属性。

```

var block=linb.create("Block").setWidth(400).setHeight(300).show();
var items=["page1","page2","page3"];
linb.create("Tabs",{
  items:items,
  value:"page2",
  position:'relative',
  width:'auto',
  height:'auto',
  dock:'none',
  noPanel:true
}).show(block);

linb.create("ButtonViews",{
  items:items,
  value:"page2",
  position:'relative',
  width:'auto',
  height:32,
  barSize:30,
  dock:'none',
  noPanel:true
}).show(block);

```

相对位置

自动宽

自动高

没有容器面板

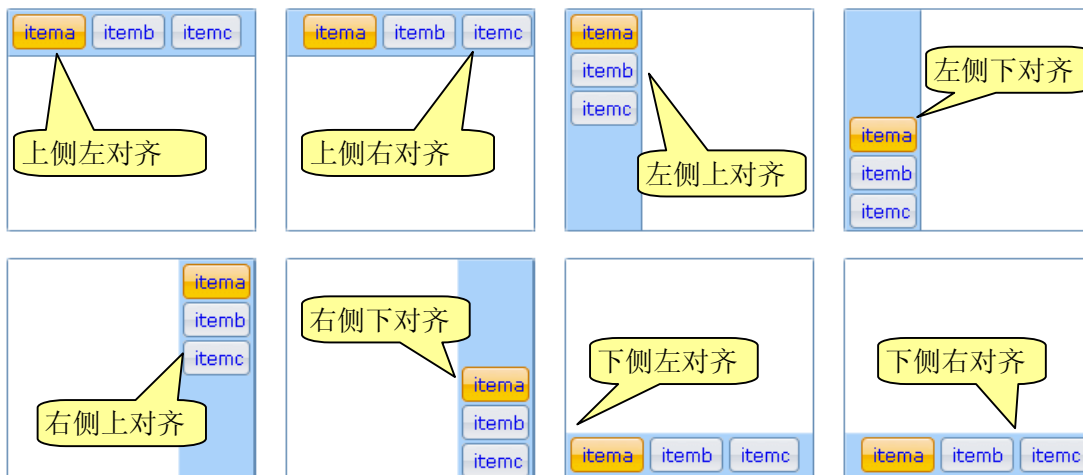
这个 ButtonViews 要设置高

没有容器面板



2. ButtonViews 的 4 个种类

ButtonViews 有三个用来定义命令按钮布局的属性：barLocation、barHAlign 和 barVAlign。barLocation 属性用来设定命令按钮栏的位置，有'top','bottom','left','right'四个选项。当 barLocation 属性是'top'或'bottom'时，可以用 barHAlign 属性来设置命令按钮的对齐方式（left 或 right）；当 barLocation 属性是'left'或'right'时，可以用 barVAlign 属性来设置命令按钮的对齐方式（top 或 bottom）。下图是 ButtonViews 的所有 8 个可能布局：



注释：在 chapter2\ButtonViews 文件夹下有一个由设计器辅助生成的 linb.UI.ButtonViews 的综合例子。

3. 代码控制页选择

可以用 setUIValue 来选择页，也可以用 fireItemClickEvent 来选择。两者的主要区别是 fireItemClickEvent 是完全模拟用户用鼠标来点击 ITEM 节点，会触发 onItemSelected 事件，setUIValue 只会改变界面的选择，不会触发 onItemSelected 事件。

```
var block=linb.create("Block").setWidth(400).setHeight(100).show();
var pages=linb.create("Tabs",{
    items:["page1","page2","page3"]
})
.onItemSelected(function(profile,item){
    linb.message(item.id);
})
.show(block);

_.asRun(function(){
    pages.fireItemClickEvent("page2");
},1000);

_.asRun(function(){
    pages.setUIValue("page1");
},2000);
```

会触发 onItemSelected 事件

4. 动态增加和删除页

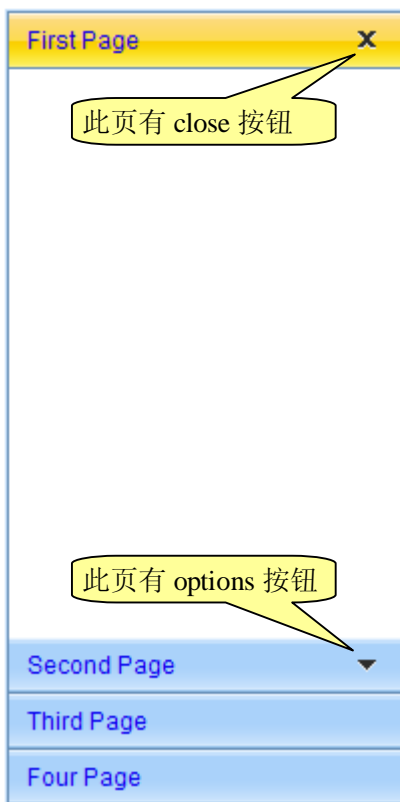
1) 页的关闭按钮

每个页可以有一个关闭按钮，当用户点击关闭按钮时，该页会被关闭（删除）。

```

var block=linb.create("Block").setWidth(200).setHeight(400).show(), stacks;
block.append(stacks=new linb.UI.Stacks({
  value:'a',
  items:[{
    id:'a',
    caption:'First Page',
    closeBtn:true
  },{
    id:'b',
    caption:'Second Page',
    optBtn:true
  },{
    id:'c',
    caption:'Third Page'
  },{
    id:'d',
    caption:'Fourth Page'
  }
]
}));
stacks.onShowOptions(function(profile,item){
  linb.message(" You clicked "+item.caption)
});

```



用页的关闭按钮来关闭页可以触发两个事件：`beforePageClose` 和 `afterPageClose`。在 `beforePageClose` 中返回 `false` 将会阻止当前页被关闭。可以到 API 去看一下这两个事件的详细信息。

2) 代码添加页和删除页

下面使用代码添加和删除页的例子：

```
var block=linb.create("Block").setWidth(400).setHeight(100).show(), tabs;
block.append(tabs=new linb.UI.Tabs({
  value:'a',
  items:[{
    id:'a',
    caption:'First Page'
  },{
    id:'b',
    caption:'Second Page'
  }]
}));
_.asynRun(function(){
  tabs.insertItems([
    id:'c',
    caption:'Third Page'
  ],{
    id:'d',
    caption:'Fourth Page'
  });
},500);
_.asynRun(function(){
  tabs.insertItems('Fifth Page');
},1000);
_.asynRun(function(){
  tabs.removeItem('d');
},1500);
_.asynRun(function(){
  tabs.removeItem(['b','c']);
},2000);
```

此页有 close 按钮

添加两页

再添加一页

删除一页

删除两页

5. 动态内容加载

多页控件的动态内容加载是一个比较常用的功能。在 linb 中可以用 `onIniPanelView` 事件来实现动态内容加载；也可以用 `beforeUITValueSet` 和 `afterUITValueSet` 两个事件来控制内容的动态加载。

`onIniPanelView` 方式：

```

var block=linb.create("Block").setWidth(400).setHeight(100).show(),
tabs=new linb.UI.Tabs({
  value:'a',
  items:[{
    id:'a',
    caption:'First Page'
  },{
    id:'b',
    caption:'Second Page'
  },{
    id:'c',
    caption:'Third Page'
  }]
});
tabs.onIniPanelView(function(profile,item){
  profile.boxing().getPanel(item.id).append(new linb.UI.SButton)
});
block.append(tabs);

```

每页第一次打开都会触发这个事件

beforeUIValueSet/afterUIValueSet 方式:

```

var block=linb.create("Block").setWidth(400).setHeight(100).show(), tabs;
block.append(tabs=new linb.UI.Tabs({
  value:'a',
  items:[{
    id:'a',
    caption:'First Page'
  },{
    id:'b',
    caption:'Second Page'
  },{
    id:'c',
    caption:'Third Page'
  }]
}));
tabs.beforeUIValueSet(function(profile,ovalue,value){
  if(value=='b')
    return false;
});
tabs.afterUIValueSet(function(profile,ovalue,value){
  if(value=='c'){
    var item=profile.getItemById(value);
    if(!item.$ini){
      profile.boxing().append(new linb.UI.SButton);
      item.$ini=true;
    }
  }
});

```

可以阻止翻到此页

得到页的内存对象

判断标识

设置标识

这种方式可能最初使用起来感到有些不太简便，但是 linb 的这种细粒度机制可以给程序员更大的发挥空间。

第十二节 菜单和工具栏

1. 弹出菜单

弹出菜单的 `items` 属性可以设置多层，表示多层弹出菜单。

下面一个基本的弹出菜单：

```
var pm=linb.create('PopupMenu')
.setItems([
  {"id":"itema", "caption":"itema", "tips":"item a"},
  {"type":"split"},
  {"id":"itemb", "type":"checkbox", value:true, "caption":"itemb", "tips":"item b"},
  {"id":"itemc", "caption":"itemc", "type":"checkbox", "tips":"item c"},
  {"id":"itemd", "caption":"itemd", "tips":"item d", sub:[
    {"id":"itemd1", "caption":"itemd1"},
    {"id":"itemd2", "caption":"itemd2"}
  ]},
  {"id":"iteme", "caption":"iteme", "tips":"item d", disabled:true}
])
.onMenuSelected(function(profile,item){
  linb.message(item.id + (item.type=="checkbox"? " : " + item.value:""))
});

linb.create('SButton')
.onClick(function(profile){
  pm.pop(profile.getRoot())
})
.show();
```

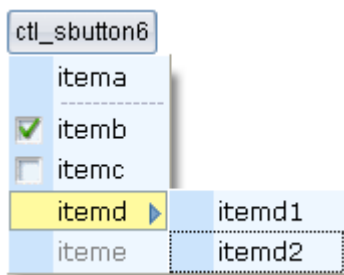
Checkbox 类型

下级弹出菜单

事件

菜单项不可用

在按钮的位置弹出菜单



2. 菜单栏

菜单栏的 `items` 属性要设置两层以上。第一层是菜单栏项，第二层以后是菜单栏的弹出菜单项目。

```

var pm=linb.create('MenuBar')
.setItems([
  {
    "id": "file", "caption": "File",
    "sub": [
      {
        "id": "newproject",
        "caption": "New Project"
      },
      {
        "id": "openproject", "caption": "Open Project",
        "add": "Ctrl+Alt+O",
        "image": "img/b.gif",
        "sub": ["option 1", "option 2"]
      },
      {
        "id": "closeproject", "caption": "Close Project"
      }
    ],
    "type": "split",
    {
      "id": "save", "caption": "Save",
      "image": "img/a.gif",
    },
    {
      "id": "saveall", "caption": "Save All",
      "add": "Ctrl+Alt+S",
      "image": "img/c.gif"
    }
  },
  {
    "id": "tools", "caption": "Tools",
    "sub": [
      {
        "id": "command", "caption": "Command Window"
      },
      {
        "id": "spy", "caption": "Components Spy"
      }
    ]
  },
  {
    "id": "build", "caption": "Build",
    disabled: true,
    "sub": [
      {
        "id": "debug",
        "caption": "Debug"
      }
    ]
  }
]).show()

```

下拉菜单

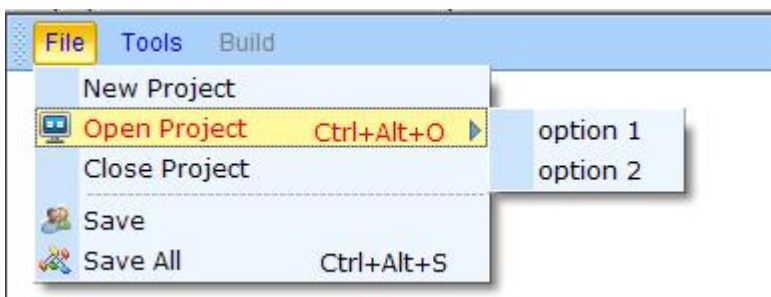
附加信息

下级菜单

分隔条

菜单项的图标

菜单项不可用



3. 工具栏

工具栏的 items 属性要设置两层。第一层是“工具组”，第二层才是工具按钮项。

```

linb.create('ToolBar',{items:[{
  "id": "align",
  "sub": [
    {"id": "left","caption": "left"},
    {"id": "center","caption": "center"},
    {type:'split'},
    {"id": "right","caption": "center"}
  ]
},{
  "id": "code",
  "sub": [{
    "id": "format","caption": "format",
    label:"label",
    image:"img/a.gif",
    "dropButton": true
  }]
}])
})
.onClick(function(profile,group,item){
  linb.message(group.id + " : " + item.id)
})
.show();

```

第一层是按钮组

第二层是按钮

分隔线

带有 label

图片

是下拉按钮

按钮组对象

按钮对象



第十三节 树形栏

1. 三种选择模式

TreeBar 控件有三种选择模式：

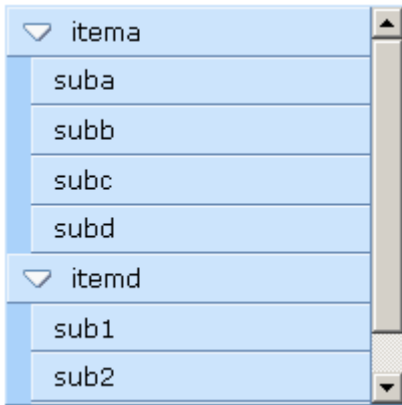
1) 不可选择

```

var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{items:[{ id : "itema", sub : ["suba","subb","subc","subd"]},
{id : "itemd", sub : ["sub1","sub2","sub3"]}]})
.setSelMode("none")
.onItemSelected(function(profile,item){
  linb.message(item.id);
}).show(block);

```

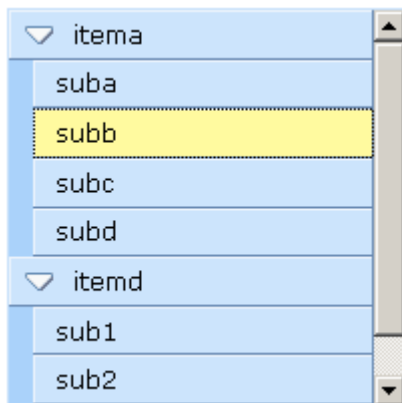
设置不可选择模式



2) 单项选择

```
var block=new linb.UI.Block({ width:200,height:200}).show();
linb.create("TreeBar",{items:[{ id : "itema", sub : ["suba","subb","subc","subd"]},
{id : "itemd", sub : ["sub1","sub2","sub3"]}]})
.setSelMode("single")
.onItemSelected(function(profile,item){
    linb.message(item.id);
}).show(block);
```

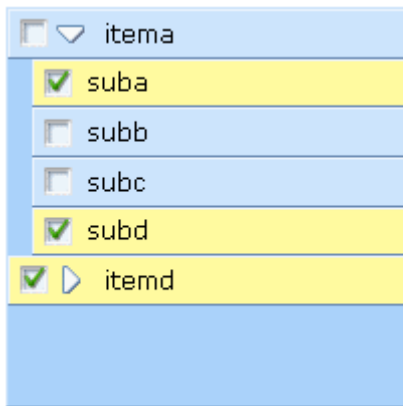
设置单项选择模式（默认的）



3) 多项选择

```
var block=new linb.UI.Block({ width:200,height:200}).show();
linb.create("TreeBar",{items:[{ id : "itema", sub : ["suba","subb","subc","subd"]},
{id : "itemd", sub : ["sub1","sub2","sub3"]}]})
.setSelMode("multi")
.onItemSelected(function(profile,item){
    linb.message(item.id);
}).show(block);
```

设置多项选择模式



2. 组条目

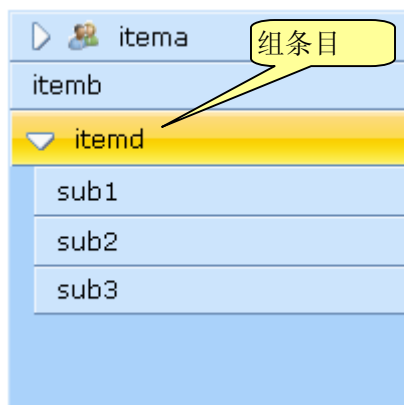
可以设置 `group` 属性把所有的有子项的条目设置成“组条目”，也可以在子项目的选项中设置 `group` 为 `true` 达到目的。

```
var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{items:[{
  id:"itema",
  image:"img/a.gif",
  sub:["suba","subb","subc","subd"]
},
{id:"itemb"},
{
  id:"itemd",
  group:true,
  sub:["sub1","sub2","sub3"]
}
]).show(block);
```

图标

第二层

这是一个组

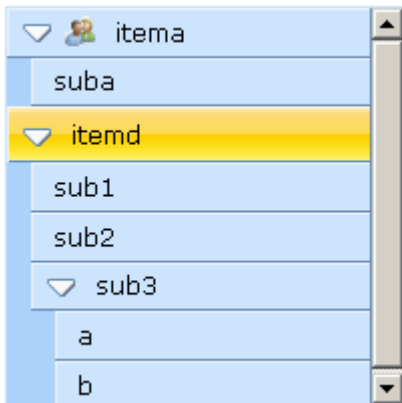


3. 默认展开到节点

`TreeBar` 的所有默认节点是折叠的（默认情况下并没有渲染），可以用 `iniFold` 属性来设置其节点默认是展开的。

```
var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{
  iniFold:false,
  items:[{
    id:"itema",
    image:"img/a.gif",
    sub:["suba"]
  },
  {
    id:"itemd",
    group:true,
    sub:["sub1","sub2",{
      id:"sub3",sub:["a","b"]
    }]
  }
]).show(block);
```

节点默认展开



4. 互斥展开

默认情况下，树的节点是可以任意展开的。有的时候为了节省空间，需要在展开一个节点的同时让相邻的已展开折叠。设置 `singleOpen` 属性为 `true` 可以达到这样的效果。

```
var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{
  singleOpen:true,
  items:[{
    id:"itema",
    image:"img/a.gif",
    sub:["suba"]
  },
  {
    id:"itemd",
    group:true,
    sub:["sub1","sub2",{
      id:"sub3",sub:["a","b"]
    }]
  }
]).show(block);
```

互斥展开

5. 动态销毁

有的企业级程序可能有成千上万个树的节点，如果终端客户把这些节点全部展开，可能会导致浏览器崩溃。dynDestory 属性就是为此设计的（大部分情况下 dynDestory 属性会与 singleOpen 属性配合使用）。

```
var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{
  dynDestory:true,
  items:[{
    id:"itema",
    image:"img/a.gif",
    sub:["suba"]
  },
  {
    id:"itemd",
    group:true,
    sub:["sub1","sub2",{
      id:"sub3",sub:["a","b"]
    }]
  }
]).show(block);
```

动态销毁

以上的树节点打开再关闭后，不能再次打开，是因为被“动态销毁”了。很显然，与“动态销毁”相对应，我们需要一个“动态创建”的功能。

6. 树节点动态加载

树节点动态加载就是这样的“动态创建”功能。

```
var block=new linb.UI.Block({width:200,height:200}).show();
linb.create("TreeBar",{
  singleOpen:true,
  dynDestory:true,
  items:[{
    id:"itema",
    sub:true
  },
  {
    id:"itemb",
    sub:true
  }
])
.onGetContent(function(profile,item,callback){
  if(item.id=="itema"){
    var rnd=_();
    callback([rnd+"-a",rnd+"-b",rnd+"-c"]);
  }
  if(item.id=="itemb")
    return ["itemsub1","itemsub2","itemsub3"];
})
.show(block);
```

互斥展开

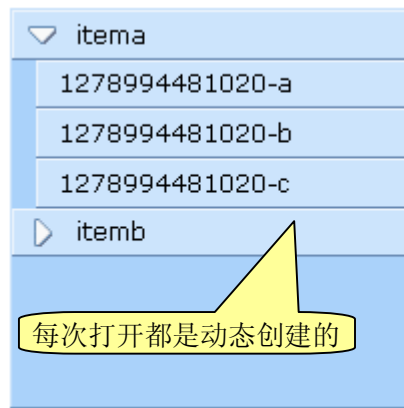
动态销毁

表示有子项要动态加载

用时间戳作为随机数

可以是异步模式调用 callback

也可以直接返回

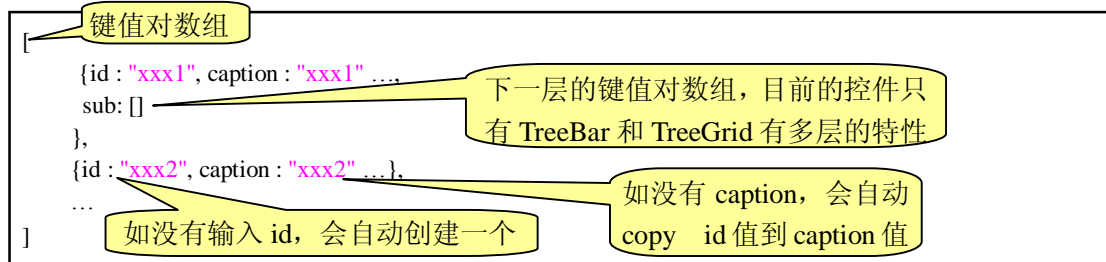


第十四节 树形表格

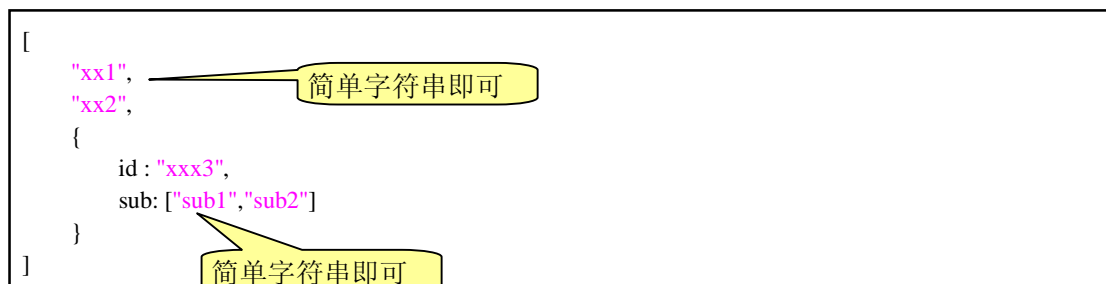
1. 给表头和表格赋值

TreeGrid 的表头和表格的数据格式与其他的带有子项的控件（派生于 `linb.UI.absList`，如 `List`，`TreeBar`）的 `items` 格式保持一致，是一个键值对数组。所以以下描述的原理也适用于其他的带有子项的控件。

这个键值对数组的通用描述方式如下：



键值对数组有一种简化的格式：



这种简化的数据会被系统格式化为：


```
[
  {id:"xx1",caption:"xx1"},
  {id:"xx2",caption:"xx2"}
  {
    id : "xxx3", caption : "xxx3",
    sub: [
      {id:"sub1",caption:"sub1"},
      {id:"sub2",caption:"sub2"}
    ]
  }
]
```

可以仔细比对以上两段代码，以对“子项数据格式”获得清晰的概念。

具体到 TreeGrid 的表头和表格的数据，我们可以用以上两种方式来给一个 TreeGrid 赋值：

1) 按照标准格式赋值

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false);
tg.setHeader([
  {id:"col1",caption:"Name"},
  {id:"col2",caption:"Age",width:40}
]).setRows([
  {id:"row1",cells:[{
    value:'Jack',caption:'Jack'
  },{
    value:23,caption:'23'
  }]},
  {id:"row2",cells:[{
    value:'John',caption:'John'
  },{
    value:32,caption:'32'
  }]}
]).show(block);
```

没有第一列

自定义宽

row 里面的 cells 节点下是单元格数据

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

2) 按照简化格式赋值

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.show(block);
```

这里设置的是 value 值

对比简化格式来说，标准格式可以给 TreeGrid 更细致的控制。

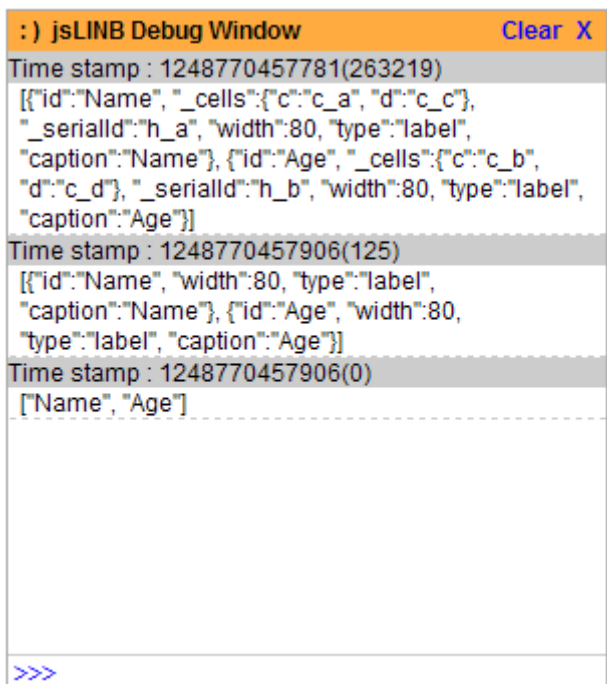
2. 获得表头的三种数据格式

用 getHeader 函数可以得到 TreeGrid 的表头数据。getHeader 函数有三种返回：

- ! getHeader()(默认的)：返回内存中正在用的表头数据；
- ! getHeader("data")：返回内存中表头的简化数据 copy；
- ! getHeader("min")：返回内存中表头的最简化数据 copy；

```
var block=new linb.UI.Block({ width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.show(block);
linb.log(tg.getHeader());
linb.log(tg.getHeader("data"));
linb.log(tg.getHeader("min"));
```

对比一下这 3 种格式



3. 获得表格的三种数据格式

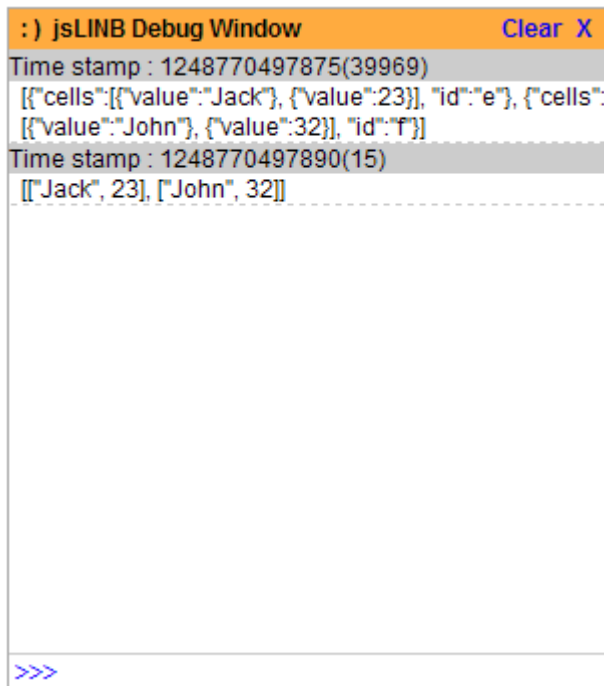
用 `getRows` 函数可以得到 `TreeGrid` 的表格数据。`getRows` 函数有三种返回：

- l `getRows()`(默认)：返回内存中正在用的表格数据；
- l `getRows("data")`：返回内存中表格数据的简化 copy；
- l `getRows("min")`：返回内存中表格数据的最简化 copy；

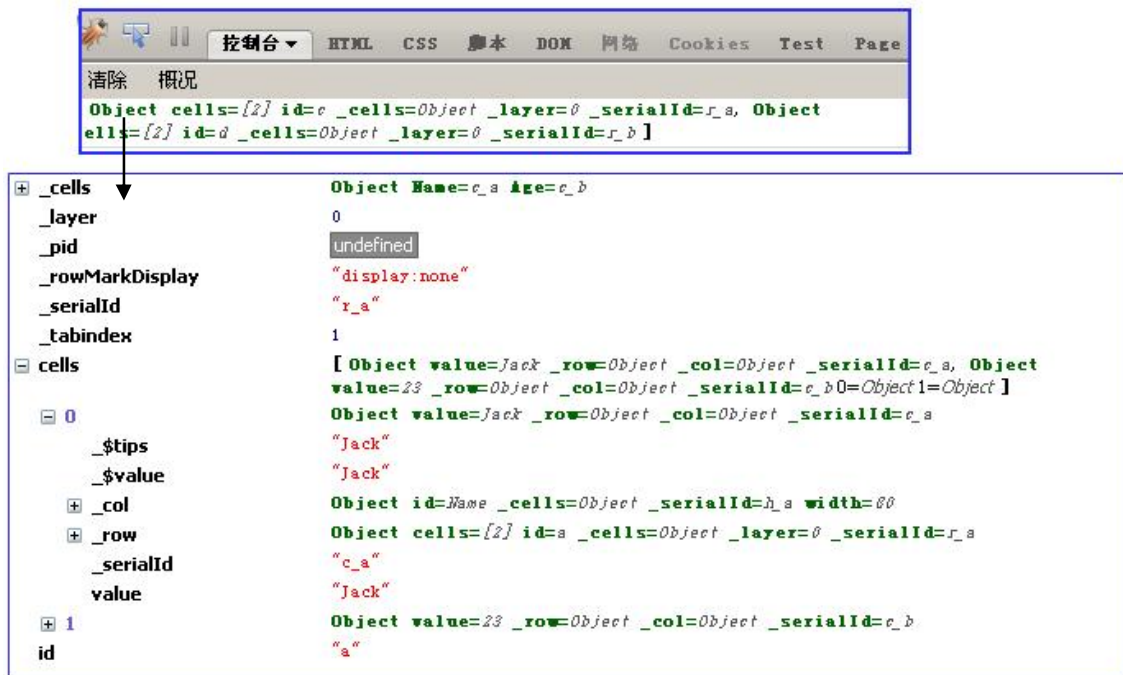
```
var block=new linb.UI.Block({ width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.show(block);
linb.log(tg.getRows("data"));
linb.log(tg.getRows("min"));
//console.log(tg.getRows());
```

对比一下这 3 种格式

内存数据有循环引用，不能直接序列化，如果有 firebug，可以运行这行程序看看内存数据的构造



上图为表格数据的两种简化 copy



上图为表格数据的内存情况

4. 表格的三种活动模式

表格的 `activeMode` 属性表示活动模式，有三种：

- l 无活动模式： `activeMode` 为“none”；
- l 行活动模式(默认的)： `activeMode` 为“row”；
- l 单元格活动模式： `activeMode` 为“cell”；

1) 无活动模式

```
var block=new linb.UI.Block({ width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[ 'Jack', 23], [ 'John', 32]])
.setActiveMode("none")
.show(block)
```

没有必要显示行头

设置无活动模式

```
. afterRowActive (function(profile,row){
    linb.message(row.id);
})
. afterCellActive (function(profile,cell){
    linb.message(cell.value);
})
```

不会触发事件

| Name | 外观 |
|------|----|
| Jack | 23 |
| John | 32 |

2) 行活动模式

```
var block=new linb.UI.Block({ width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false) // 没有必要显示行头
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]]) // 设置无活动模式
.setActiveMode("row")
.show(block)

// 会触发这个事件
tg.afterRowActive (function(profile,row){
    linb.message(row.id);
})
tg.afterCellActive (function(profile,cell){
    linb.message(cell.value);
})
```

| Name | 外观 | Age |
|------|----|-----|
| Jack | 23 | |
| John | 32 | |

3) 单元格活动模式

```
var block=new linb.UI.Block({ width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false) // 没有必要显示行头
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]]) // 设置无活动模式
.setActiveMode("cell")
.show(block)

// 会触发这个事件
tg.afterRowActive (function(profile,row){
    linb.message(row.id);
})
tg.afterCellActive (function(profile,cell){
    linb.message(cell.value);
})
```

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

外观

5. 表格的五种选择模式

TreeGrid 有五种选择模式分别为：

- ❑ 不可选择行或列：activeMode 属性为'none'， selMode 属性为'none'；
- ❑ 可以选择单行(默认的)： activeMode 属性为'row'， selMode 属性为'single'；
- ❑ 可以选择多行： activeMode 属性为'row'， selMode 属性为'multi'；
- ❑ 可以选择一个单元格： activeMode 属性为'cell'， selMode 属性为'single'；
- ❑ 可以选择多个单元格： activeMode 属性为'cell'， selMode 属性为'multi'。

1) 不可选择

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.setSelMode("none")
.show(block)

.afterUIValueSet(function(profile, ovalue, value){
    linb.message(value);
});
```

没有必要显示行头

设置不可选择模式

不会触发事件

以上代码设置了不可选择，行背景的变化是 active row 的结果，并非选择。

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

这是活动行的外观，并不是选择

2) 可选择一行

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.setSelMode("single")
.show(block)

.afterUIValueSet(function(profile, ovalue, value){
    linb.message(value);
});
```

没有必要显示行头

设置单项选择模式

值是行的 id

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

选择行的外观

3) 可选择多行

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(24)
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.setSelMode("multi")
.show(block)

.afterUIValueSet(function(profile, ovalue, value){
    linb.message(value);
});
```

设置行头的宽

设置多项选择模式

值是行的 id 们

| <input type="checkbox"/> | Name | Age |
|-------------------------------------|------|-----|
| <input checked="" type="checkbox"/> | Jack | 23 |
| <input checked="" type="checkbox"/> | John | 32 |

4) 可选择一个单元格

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false) // 没有必要显示行头
.setActiveMode("cell")
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.setSelMode("single") // 设置单项选择模式
.show(block)

.afterUIValueSet(function(profile,ovalue,value){
    linb.message(value);
});
```

值: "行 id"+"|"+"列 id"

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

5) 可选择多个单元格

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false) // 设置行头列的宽
.setActiveMode("cell")
.setHeader(["Name", "Age"])
.setRows([[Jack', 23], [John', 32]])
.setSelMode("multi") // 设置多项选择模式
.show(block)

.afterUIValueSet(function(profile,ovalue,value){
    linb.message(value);
});
```

值: 多个"行 id"+"|"+"列 id"

| Name | Age | |
|------|-----|--|
| Jack | 23 | |
| John | 32 | |

6. 树状表格

当行数据里面有 sub 项的时候, 会自动生成树状表格。


```

var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(20)
.setHeader([
    {id:"col1", caption:"Name"},
    {id:"col2", caption:"Age", width:40}
]).setRows([
    {id:"row1",cells:['Jack',23]},
    {id:"row2",cells:['John',32],
      sub:[{id:"row21",cells:['Tom',24]},
            {id:"row22",cells:['Bob',25]}
    ]}
]).show(block)

```

树状表格要有行头列

树的下一级

| | Name | Age |
|---|------|-----|
| | Jack | 23 |
| ▼ | John | 32 |
| | Tom | 24 |
| | Bob | 25 |

上图中看到，树状表格的第一列并没有缩进，缩进是在行头里表现的。运行一下代码：

```

var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(20)
.setGridHandlerCaption("Name")
.setRowHandlerWidth(80)
.setHeader([
    {id:"col2", caption:"Age", width:40}
]).setRows([
    {id:"row1",caption: 'Jack',cells:[23]},
    {id:"row2",caption: 'John',cells:[32],
      sub:[{id:"row21",caption: 'Tom',cells:[24]},
            {id:"row22", caption: 'Bob',cells:[25]}
    ]}
]).show(block)

```

行头文字

每行有 caption

| | Name | Age |
|---|------|-----|
| | Jack | 23 |
| ▼ | John | 32 |
| | Tom | 24 |
| | Bob | 25 |

行头缩进

7. 配置列

1) 特殊的第一列：行头列

在 `rowHandler` 属性为 `true` 的时候，第一列是行头列。行头里面的单元格是特殊的单元格，它们显示的并不是 `cells` 里面的数据，行头单元格里面现实的是“行号”、“表示选中的 `checkbox`”和“树状折叠命令按钮”等信息。

```
var block=new linb.UI.Block({width:200,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23]},
    {id:"row2",caption:'John',cells:[32],
      sub:[{id:"row21",caption:'Tom',cells:[24]},
            {id:"row22",caption:'Bob',cells:[25]}
    ]}
]).show(block)
```

树状表格要有行头列

树的下一级

| | Name | Age |
|---|------|-----|
| | Jack | 23 |
| ▼ | John | 32 |
| | Tom | 24 |
| | Bob | 25 |

| | Name | Age |
|---|------|-----|
| | Jack | 23 |
| ▼ | John | 32 |
| | Tom | 24 |
| | Bob | 25 |

上一节中缩进的例子

2) 列的宽度

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80) // 第一列(行头列)的宽度
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1",caption:"Age",width:40}, // 列的宽度
  {id:"col2",caption:"Part-time",width:90}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23,true]},
  {id:"row2",caption:'John',cells:[32,false]}
]).show(block)
._asRun(function(){
  tg.updateHeader("col2",{width:70}); // 动态更改列的宽度
},1000);
```

| Name | Age | Part-time | |
|------|-----|-----------|--|
| Jack | 23 | true | |
| John | 32 | false | |

3) 通过拖拽改变列宽

TreeGrid 的 colResizer 是控制列是否可以通过拖拽来改变列宽的属性。每一个列头数据中也可以包含一个 colResizer 项，列头数据中的 colResizer 项优先于 TreeGrid 的 colResizer 属性。这种细粒度设置优先于粗粒度设置的规则对其他的属性来说也是一样的。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80) // 不能通过拖拽来改变列宽
.setColResizer(false)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1",caption:"Age",width:40},
  {id:"col2",caption:"Part-time",width:90,colResizer:true} // 只有这列可以
]).setRows([
  {id:"row1",caption:'Jack',cells:[23,true]},
  {id:"row2",caption:'John',cells:[32,false]}
]).show(block)
```

| Name | Age | Part-time | |
|------|-----|-----------|--|
| Jack | 23 | true | |
| John | 32 | false | |

可以改变列宽

4) 通过拖拽改变列的位置

TreeGrid 的 colMovable 是控制列是否可以通过拖拽来改变列的位置。

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColMovable(false)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40},
  {id:"col2", caption:"Part-time", width:90,colMovable:true}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23, true]},
  {id:"row2",caption:'John',cells:[32, false]}
]).show(block)
```

不能通过拖拽来改变列的位置

只有这列可以

| Name | Age | Part-time |
|------|-----|-----------|
| Jack | 23 | true |
| John | 32 | false |

5) 列排序

TreeGrid 的 colSortable 是控制列是否可以排序的属性。

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40},
  {id:"col2", caption:"Part-time", width:90,colSortable:true}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23, true]},
  {id:"row2",caption:'John',cells:[32, false]}
]).show(block)
```

不能排序

只有这列可以

| Name | Age | Part-time <input checked="" type="checkbox"/> |
|------|-----|---|
| Jack | 23 | true |
| John | 32 | false |

排序图标

6) 列的自定义排序

列头数据的 sortBy 项是自定义排序功能的函数。

```

var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1", caption:"Age", width:40},
    {id:"col2", caption:"Part-time", width:90,sortby:function(x,y){return -1}}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23, true]},
    {id:"row2",caption:'John',cells:[32, false]}
]).show(block)

```

自定义排序函数

7) 列是否可以隐藏

TreeGrid 的 colHidable 是控制列是否可以通过选择来隐藏的属性。

```

var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColHidable (false)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1", caption:"Age", width:40, colHidable:true },
    {id:"col2", caption:"Part-time", width:90, colHidable:true }
]).setRows([
    {id:"row1",caption:'Jack',cells:[23, true]},
    {id:"row2",caption:'John',cells:[32, false]}
]).show(block)

```

不能隐藏

设置这两列
可以隐藏

| Name | Age | Part-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | <input checked="" type="checkbox"/> |

8) 列的单元格种类

单元格的种类的关键字为“type”，可以在列头数据中设置，也可以在行数据中设置，也可以在单元格数据中设置。优先级为：单元格 > 行 > 列。这个优先级顺序也适用于其他的关键字（如 cellStyle、cellClass、cellRenderer、disabled、editable）。

单元格种类目前有以下儿种：

- l 'label': 标签;
- l 'button': 按钮;
- l 'input': 输入框;
- l 'textarea': 多行输入框;

- | 'number': 数字输入框;
- | 'progress': 进程框;
- | 'combobox': 下拉列表输入框;
- | 'listbox': 只读下拉列表输入框;
- | 'getter': 获得数据输入框;
- | 'helpinput': 辅助下拉列表输入框;
- | 'cmdbox': 命令输入框;
- | 'popbox': 弹出窗口输入框;
- | 'time': 时间下拉列表输入框;
- | 'date': 日期下拉列表输入框;
- | 'color': 颜色下拉列表输入框;

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Part-time",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23,true]},
    {id:"row2",caption:'John',cells:[32,false]}
]).show(block)
```

数字

checkbox

| Name | Age | Part-time |
|------|-----|-----------|
| Jack | 23 | true |
| John | 32 | false |

| Name | Age | Part-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | <input type="checkbox"/> |

9) 列头的样式

列头数据的 headerStyle 和 headerClass 关键字是用来自定义的设置列头的样式的。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Naem")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Part-time",width:90,type:"checkbox",headerStyle:"font-weight:bold;"}
]).show(block)
```


设置粗体

| Naem | Age | Part-time |
|------|-----|-----------|
|------|-----|-----------|

10) 给列头加图标

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40, type: "number"},
  {id:"col2", caption:"Part-time", width:90, type: "checkbox", renderer: function(h){
    return "<img style='vertical-align:middle' src='img/a.gif'> " + h.caption;
  }}
]).show(block)
```

用 renderer 来加图标

| Name | Age |  Part-time |
|------|-----|---|
|------|-----|---|

11) 动态更新列头

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40, type: "number"},
  {id:"col2", caption:"Part-time", width:90, type: "checkbox"}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23, true]},
  {id:"row2",caption:'John',cells:[32, false]}
]).show(block)

_.asynRun(function(){
  tg.updateHeader('col2','Full-time')
},1000)

_.asynRun(function(){
  tg.updateHeader('col2',{caption:'Part-time', width:40, headerStyle:'font-weight:bold', colResizer:false,
  colSortable:false, colMovable:true, colHidable:true})
},2000)
```

只更新列头的标题

还可以更新这些

8. 配置行

1) 行的高度

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Full-time",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23,true],height:120},
    {id:"row2",caption:'John',cells:[32,false]}
]).show(block)
```

| Name | Age | Full-time | |
|------|-----|-------------------------------------|--|
| Jack | 23 | <input checked="" type="checkbox"/> | |
| John | 32 | <input type="checkbox"/> | |

2) 通过拖拽改变行高

TreeGrid 的 `rowlResizer` 是控制列是否可以通过拖拽来改变行高的属性。每一个行数据中也可以包含一个 `rowlResizer` 项，行数据中的 `rowlResizer` 项优先于 TreeGrid 的 `rowlResizer` 属性。这种细粒度设置优先于粗粒度设置的规则对其他的属性来说也是一样的。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setRowResizer(false)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40},
    {id:"col2",caption:"Full-time",width:90}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23,true]},
    {id:"row2",caption:'John',cells:[32,false],rowResizer:true}
]).show(block)
```


| Name | Age | Full-time |
|------|-----|-----------|
| Jack | 23 | true |
| John | | |

此处可拖拽

3) 行的单元格种类

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40, type: "number"},
  {id:"col2", caption:"Part-time", width:90, type: "checkbox"}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23, true]},
  {id:"row2",caption:'John',cells:[32, false],type: "label"}
]).show(block)
```

此行全部设为 label

| Name | Age | Part-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | <input type="checkbox"/> |

| Name | Age | Part-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | false |

4) 行的样式

行数据的 rowStyle 和 rowClass 关键字是用来自定义的设置列头的样式的。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setGridHandlerCaption("Name")
.setHeader([
  {id:"col1", caption:"Age", width:40, type: "number"},
  {id:"col2", caption:"Full-time", width:90, type: "checkbox"}
]).setRows([
  {id:"row1",caption:'Jack',cells:[23, true]},
  {id:"row2",caption:'John',cells:[32, false],rowStyle: "background-color:#00ff00"}
]).show(block)
```

设置此行的样式

| Name | Age | Full-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | <input type="checkbox"/> |

5) 自动行号

TreeGrid 的 `rowNumbered` 属性可以设置自动行号。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setRowNumbered(true)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Full-time",width:90,type:"checkbox",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23]},
    {id:"row2",caption:'John',cells:[32],
      sub:[{id:"row21",caption:'Tom',cells:[24]},
            {id:"row22",caption:'Bob',cells:[25]}
    ]
}
]).show(block)
```

设置行号

| | Name | Age | Full-time |
|-----|------|-----|--------------------------|
| 1 | Jack | | <input type="checkbox"/> |
| 2 | John | 32 | <input type="checkbox"/> |
| 2.1 | Tom | 24 | <input type="checkbox"/> |
| 2.2 | Bob | 25 | <input type="checkbox"/> |

行号

6) 自定义行号

用 TreeGrid 的自定义函数 `getNumberedStr` 可以设置自定义的行号显示。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setRowNumbered(true)
.setGridHandlerCaption("姓名")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Full-time",width:90,type:"checkbox",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23]},
    {id:"row2",caption:'John',cells:[32],
        sub:[{id:"row21",caption:'Tom',cells:[24]},
            {id:"row22",caption:'Bob',cells:[25]}
        ]
    }
])
.setCustomFunction('getNumberedStr',function(no){
    var a=no.split('.');
    a[0]=[1:'I',2:'II'][a[0]];
    return a.join('.')
})
.show(block)

```

自定义行号

| 姓名 | Age | Full-time |
|----------|-----|--------------------------|
| I Jack | 23 | <input type="checkbox"/> |
| II John | 32 | <input type="checkbox"/> |
| II-1 Tom | 24 | <input type="checkbox"/> |
| II-2 Bob | 25 | <input type="checkbox"/> |

7) 间隔色

TreeGrid 的 altRowsBg 属性可以设置间隔色。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setAltRowsBg(true)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Full-time",width:90,type:"checkbox",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23]},
    {id:"row2",caption:'John',cells:[32],
        sub:[{id:"row21",caption:'Tom',cells:[24]},
            {id:"row22",caption:'Bob',cells:[25]}
        ]
    }
])
.show(block)

```

设置间隔色

| Name | Age | Full-time |
|--------|-----|--------------------------|
| Jack | 23 | <input type="checkbox"/> |
| ▼ John | 32 | <input type="checkbox"/> |
| Tom | 24 | <input type="checkbox"/> |
| Bob | 25 | <input type="checkbox"/> |

8) 分组

用行数据的 `group` 参数可以把某行定义为组。

```
var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1",caption:"Age",width:40,type:"number"},
    {id:"col2",caption:"Full-time",width:90,type:"checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23]},
    {id:"row2",caption:'John',cells:[32],group:true,
      sub:[{id:"row21",caption:'Tom',cells:[24]},
           {id:"row22",caption:'Bob',cells:[25]}
    ]}
]).show(block)
```

分组行

| Name | Age | Full-time |
|--------|-----|--------------------------|
| Jack | 23 | |
| ▼ John | | |
| Tom | 24 | <input type="checkbox"/> |
| Bob | 25 | <input type="checkbox"/> |

分组

9) 预览区域和小结区域

用行数据的 `preview` 参数和 `summary` 参数可以给某行定义预览区域和小结区域。

```

var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1", caption:"Age", width:40, type: "number"},
    {id:"col2", caption:"Full-time", width:90, type: "checkbox"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23], preview: '<strong>Attention:</strong>',summary: '<em>Jack is athe right one</em>' },
    {id:"row2",caption:'John',cells:[32], preview: 'John is OK',
        sub:[{id:"row21",caption:'Tom',cells:[24]},
            {id:"row22",caption:'Bob',cells:[25]}
        ]
    }
]).show(block)

```

| Name | Age | Full-time |
|------------------------|-----|--------------------------|
| Attention: | | |
| Jack | 23 | <input type="checkbox"/> |
| Jack is athe right one | | |
| John is OK | | |
| ▶ John | 32 | <input type="checkbox"/> |

10) 动态更新行

可以用 `updateRow` 函数来动态更新某行。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80).setGridHandlerCaption("Name")
.setHeader([
{id:"col1", caption:"Age", width:40, type: "number"},{id:"col2", caption:"Full-time", width:90, type:
"checkbox"}
]).setRows([
{id:"row1",caption:'Jack',cells:[23, true]},
{id:"row2",caption:'John',cells:[32,
sub:[{id:"row21",caption:'Tom',cells:[24]},
{id:"row22",caption:'Box',cells:[25]}
]}
]).show(block)

_.asynRun(function(){
    tg.updateRow('row2', 'Jerry')
},1000)

_.asynRun(function(){
    tg.updateRow('row2',{caption:'Group', height:30, rowStyle:'background-color:#00ff00;', rowResizer:false,
group:true, preview:'preview', summary:'summary'})
},2000)

_.asynRun(function(){
    tg.updateRow('row1', {sub:[{ value:"Kate",cells:[24,true]}]})
},3000)

```

只更新第一个单元格的显示值

还可以更新这些

更新所有的单元格

| Name | Age | Full-time |
|---------|-----|-------------------------------------|
| ▶ Jack | 23 | <input checked="" type="checkbox"/> |
| preview | | |
| ▼ Group | | |
| Tom | 24 | <input type="checkbox"/> |
| Box | 25 | <input type="checkbox"/> |
| summary | | |

9. 配置单元格

1) 单元格的种类

用单元格数据的 type 参数可以自定义这个单元格的种类。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1", caption:"Age", width:40, type: "number"},
    {id:"col2", caption:"Full-time", width:90, type: "label"}
]).setRows([
    {id:"row1",caption:'Jack',cells:[23, true]},
    {id:"row2",caption:'John',cells:[32, {value:false, type: "checkbox"}]}
]).show(block)

```

优先级没有在单元
格里面设置的高

设置此单元格的种类

| Name | Age | Full-time |
|------|-----|--------------------------|
| Jack | 23 | true |
| John | 32 | <input type="checkbox"/> |

2) 单元格的样式

用单元格数据的 cellStyle 参数或 cellClass 参数可以自定义这个单元格的样式。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80)
.setColSortable(false)
.setGridHandlerCaption("Name")
.setHeader([
    {id:"col1", caption:"Age", width:40, type: "number"},
    {id:"col2", caption:"Full-time", width:90, type: "checkbox", cellStyle: "background-color:#00ff00;"}
]).setRows([
    {id:"row1",caption:'Jack', cells:[23, true]},
    {id:"row2",caption:'John', cellStyle: "background-color:#0000ff;",cells:[
        32,
        {value:false, cellStyle: "background-color:#ff0000;"}
    ]}
]).show(block)

```

| | Age | Full-time |
|------|-----|-------------------------------------|
| Jack | 23 | <input checked="" type="checkbox"/> |
| John | 32 | <input type="checkbox"/> |

行设置的效果

列设置的效果

单元格设置的效果

3) 动态 update 单元格

用 updateCellByRowCol 方法可以动态的更新这个单元格。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandlerWidth(80).setGridHandlerCaption("Name")
.setHeader([
{id:"col1", caption:"Age", width:40, type: "number"},{id:"col2", caption:"Full-time", width:90, type:
"checkbox"}
]).setRows([
{id:"row1",caption:'Jack',cells:[23, true]},
{id:"row2",caption:'John',cells:[32]}
]).show(block)

_.asynRun(function(){
  tg.updateCellByRowCol('row2','col1', 18)
},1000)

_.asynRun(function(){
  tg.updateCellByRowCol('row2','col1',{value:18,cellStyle:'background-color:#00ff00;'})
},2000)

_.asynRun(function(){
  tg.updateCellByRowCol ('row2','col1', {type:"listbox",value:"20", editorListItems:["20","30","40"],
editable:true})
},3000)

```

只更新这个单元格的值

还可以更新样式

更新单元格的种类

10. 界面编辑状态

TreeGrid 的 `editable` 属性控制着界面的编辑状态。设置控件的 `editable` 属性为 `true`，表示界面的单元格都可以编辑；设置列头数据的 `editable` 属性为 `true`，表示该列都可以编辑；设置行数据的 `editable` 属性为 `true`，表示该行都可以编辑；设置单元格数据的 `editable` 属性为 `true`，表示该单元格可以编辑。**editable 遵循细粒度设置优先的原则。**以下举例说明：

- l 只有一个单元格可以编辑：设置控件的 `editable` 为 `false`，同时设置此单元格数据的 `editable` 为 `true`；
- l 只有一个列可以编辑：设置控件的 `editable` 为 `false`，同时设置此列数据的 `editable` 为 `true`；
- l 只有一个行可以编辑：设置控件的 `editable` 为 `false`，同时设置此行数据的 `editable` 为 `true`；
- l 只有一个单元格不可以编辑：设置控件的 `editable` 为 `true`，同时设置此单元格数据的 `editable` 为 `false`；
- l 只有一个列不可以编辑：设置控件的 `editable` 为 `true`，同时设置此列数据的 `editable` 为 `false`；
- l 只有一个行不可以编辑：设置控件的 `editable` 为 `true`，同时设置此行数据的 `editable` 为 `false`；

另外，要特别注意的是：行头的单元格不可编辑；`type` 为 “label” 或 “button” 属性的单元格不可编辑。

1) 表格可编辑

表格是否可以编辑可以用表格的 `editable` 属性定义。

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(true)
.setHeader([
{id:"col1", caption:"Name", width:60, type: 'input'},
{id:"col2", caption:"Age", width:40, type: "number"},
{id:"col3", caption:"Gender", width:40, type: "listbox", editorListItems:[{id:'male',caption:'Male'},{id:'female',
caption:'Female'}]}
]).setRows([
[Jack',23, { value:'male',caption:'Male'}],
[John',25, { value:'female',caption:'Female'}]
]).show(block)
```

表格可编辑

定义 editor 的 list

list 需要 value 和 caption

| Name | Age | Gender |
|------|-----|--------|
| Jack | 23 | Male |
| John | 25 | Female |

| Name | Age | Gender |
|------|-------|--------|
| Jack | 23 | Male |
| John | 25.00 | Female |

| Name | Age | Gender |
|------|-----|--------|
| Jack | 23 | Male |
| John | 25 | Fen |
| | | Male |
| | | Female |

2) 列可编辑

某列是否可以编辑可以用列数据的 `editable` 参数定义。

```
var block=new linb.UI.Block({ width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(false)
.setHeader([
{id:"col1", caption:"Name", width:60, type: 'input'},
{id:"col2", caption:"Age", width:40, type: "number"},
{id:"col3", caption:"Gender", width:40, type: "listbox",
editorListItems:[{id:'male',caption:'Male'},{id:'female', caption:'Female'}]}
]).setRows([
[Jack',23, { value:'male',caption:'Male'}],
[John',25, { value:'female',caption:'Female'}]
]).show(block)
```

表格不可编辑

此列可以编辑

3) 行可编辑

某行是否可以编辑可以用行数据的 `editable` 参数定义。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(false)
.setHeader([
{id:"col1", caption:"Name", width:60, type: 'input'},
{id:"col2", caption:"Age", width:40, type: "number"},
{id:"col3", caption:"Gender", width:40, type: "listbox", editorListItems:[{id:'male',caption:'Male'},{id:'female',caption:'Female'}]}
]).setRows([
[Jack',23, { value:'male',caption:'Male'}],
{ cells:[John',25, { value:'female',caption:'Female'}],editable:true }
]).show(block)

```

表格不可编辑

此行可以编辑

4) 单元格可编辑

某单元格是否可以编辑可以用单元格数据的 `editable` 参数定义。

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(false)
.setHeader([
{id:"col1", caption:"Name", width:60, type: 'input'},
{id:"col2", caption:"Age", width:40, type: "number"},
{id:"col3", caption:"Gender", width:40, type: "listbox", editorListItems:[{id:'male', caption:'Male'}, {id:'female',caption:'Female'}]}
]).setRows([
[Jack',23, { value:'male', caption:'Male'}],
[John',25, { value:'female', caption:'Female', editable:true } ]
]).show(block)

```

表格不可编辑

此单元格可以编辑

5) 编辑器的设置

当界面（具体到单元格）处于编辑状态时，激活单元格会出现单元格编辑器。根据单元格种类的不同，默认的编辑器有以下几种：

- l 'label': 只读，无编辑器；
- l 'button': 只读，无编辑器
- l 'input': 普通输入框（linb.UI.Input）；
- l 'textarea': 多行输入框（linb.UI.Input，可以输入多行）；
- l 'number': 数字输入框（linb.UI.Input，只可以输入数字）；
- l currency: 货币输入框（linb.UI.Input，只可以输入货币数字）；
- l 'progress': 数字输入框（linb.UI.Input，可以输入 0 到 1 之间）；
- l 'combobox': 下拉列表输入框（linb.UI.ComboInput, combobox）；
- l 'listbox': 只读下拉列表输入框（linb.UI.ComboInput, listbox）；
- l 'getter': 获得数据输入框（linb.UI.ComboInput, getter）；
- l 'helpinput': 辅助下拉列表输入框（linb.UI.ComboInput, helpinput）；
- l 'cmdbox': 命令输入框（linb.UI.ComboInput, cmdbox）；

- | 'popbox': 弹出窗口输入框 (linb.UI.ComboInput, popbox);
- | 'time': 时间下拉列表输入框 (linb.UI.ComboInput, time);
- | 'date': 日期下拉列表输入框 (linb.UI.ComboInput, date);
- | 'color': 颜色下拉列表输入框 (linb.UI.ComboInput, color);

```
var block=new linb.UI.Block({ width:300,height:340}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(true)
.setHeader(["Type", "Cell UI"]).setRows([
  { cells:['label',{ type:'label',value:'label'}]},
  { cells:['button',{ type:'button',value:'button'}]},
  { cells:['input',{ type:'input',value:'input'}]},
  { cells:['textarea',{ type:'textarea',value:'textarea'}]},
  { cells:['number',{ type:'number',value:'1.23'}]},
  { cells:['progress',{ type:'progress',value:'0.85'}]},
  { cells:['combobox',{ type:'combobox',value:'combobox'}]},
  { cells:['listbox',{ type:'listbox',value:'listbox'}]},
  { cells:['getter',{ type:'getter',value:'getter'}]},
  { cells:['helpinput',{ type:'helpinput',value:'helpinput'}]},
  { cells:['cmdbox',{ type:'cmdbox',value:'cmdbox'}]},
  { cells:['popbox',{ type:'popbox',value:'popbox'}]},
  { cells:['time',{ type:'time',value:'12:08'}]},
  { cells:['date',{ type:'date',value:(new Date).getTime()]}},
  { cells:['color',{ type:'color',value:'#00ff00'}]}
]).show(block)
```

| Type | Cell UI |
|-------------|-----------|
| label | label |
| button | button |
| input | input |
| textarea | textarea |
| number | 1.23 |
| progress | 85% |
| combobox | combobox |
| listbox | listbox |
| getter | getter |
| helpinput | helpinput |
| cmdbox | cmdbox |
| popbox | popbox |
| timepicker | 12:08 |
| datepicker | 7/29/2009 |
| colorpicker | #00FF00 |

6) 自定义单元格编辑器

除了默认的编辑器之外，可以通过 beforeIniEditor 事件来自定义编辑器。

```

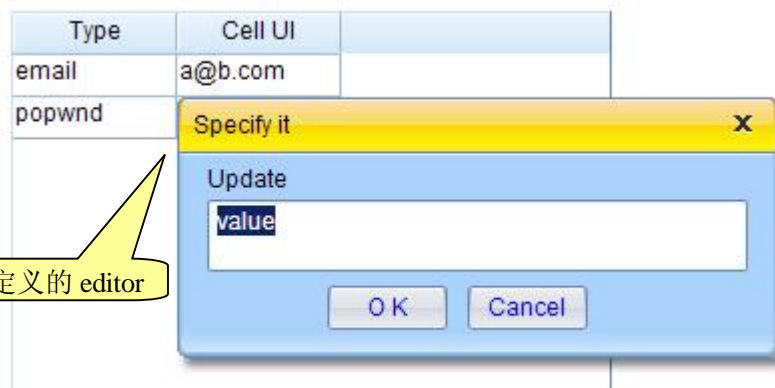
var block=new linb.UI.Block({width:300,height:340}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(true)
.setHeader(["Type", "Cell UI "]).setRows([
  { cells:['email',{type:'email',value:'a@b.com'}]},
  { cells:['popwnd',{type:'popwnd',value:'value'}]}
])
.beforeIniEditor(function(profile, cell, cellNode){
  var t=cell.type;
  if(t=='email'){
    var editor = new linb.UI.Input({ valueFormat:"^([\\w\\.-]+@[\\w\\.-]+|\\.([\\w\\.-]{2,4})$"});
    return editor;

    if(t=='popwnd'){
      var dlg=linb.prompt('Specify it','Update',cell.value, function(value){
        if(cell.value!=value)
          profile.boxing().updateCell(cell, value);
      });
      dlg.getRoot().cssPos(cellNode.offset());
      return false;
    }
  }
}).show(block);

```

直接返回简单的自定义 editor
只能是 Input 或 CombInput

完全自定义 editor，要返回 false



11. 增加行和删除行

```

var block=new linb.UI.Block({width:240,height:200}).show();
var tg=new linb.UI.TreeGrid;
tg.setRowHandler(false)
.setEditable(true)
.setHeader([
{id:"col1",caption:"Name",width:60,type:'input'},
{id:"col2",caption:"Age",width:40,type:"number"}
]).setRows([
{id:'row1',cells:['Jack',23]},
{id:'row2',cells:['John',25]}
]).show(block);

_.asRun(function(){
    tg.insertRows([[]]) // 加入一个空行
},1000);
_.asRun(function(){
    tg.insertRows(["Tom",30]) // 加入一行
},2000);
_.asRun(function(){
    tg.insertRows([{'id':'row3',cells:['Jerry',19]},{'id':'row4',cells:['Mark',31]}]) // 加入两行
},3000);
_.asRun(function(){
    tg.removeRows('row1') // 删除指定行
},4000);
_.asRun(function(){
    tg.removeRows(['row2','row3']) // 删除指定两行
},5000);
_.asRun(function(){
    tg.insertRows([{'id':'row4',cells:['Jack',23]}],null,null,true) // 加入一行到表头
},6000);
_.asRun(function(){
    tg.insertRows([{'id':'John',23}],null,'row1',false) // 加入一行到'row1'的后面
},7000);

```

注释:

在 chapter2/TGDynamic 文件夹下有一个由设计器辅助生成的 linb.UI.TreeGrid 的综合例子; 在 chapter2/ TreeGrid.Paging 文件夹下有一个由设计器辅助生成的 linb.UI.TreeGrid 翻页的例子。

第十五节 其他控件

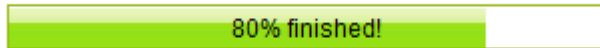
1. ProgressBar 控件

进度条组件是一个比较简单的控件，经常用到的属性有 value 何 captionTpl 两个。

```
linb.create('ProgressBar')
.setCaptionTpl("{value}% finished!")
.setValue(80)
.show();
```

设置文字显示模板，{value}表示进度数值

进度数值



2. Slider 控件

Slider 控件是一个显示和编辑数值（或数值区间）的控件。

```
linb.create('Slider')
.setPosition('relative')
.setSteps(100)
.setValue("20:50")
.show()

linb.create('Slider')
.setSteps(10)
.setType("vertical")
.setIsRange(false)
.setValue(2)
.setHeight(200)
.show();
```

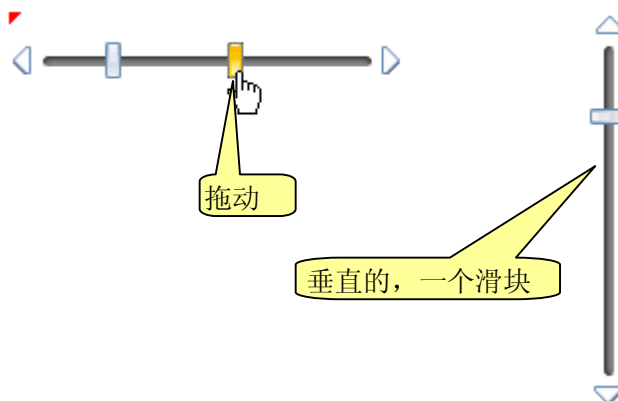
设置步长为 100

设置两个滑块的位置，值里面有两个部分

设置步长为 10

是个垂直的 slider

只有一个滑块



3. Image 控件

linb.UI.Image 控件是对 Html Image 的封装，常用的属性和事件主要是 src 属性，beforeLoad 事件（图片装载前调用，返回 false 表示不加载图片）和 afterLoad 事件（图片装载成功时调用）。

```
linb.create("Image")
.setSrc("img/a.gif")
.afterLoad(function(){
    linb.message("The picture is loaded.");
})
.show();

linb.create("Image")
.setSrc("img/b.gif")
.beforeLoad(function(){
    return false;
})
.show();
```

图片加载成功事件

阻止图片加载

4. PageBar 控件

linb 中的翻页控件是一个不太传统的控件。要注意的是 **value** 属性包括三个部分：最小页的值；当前页的值；最大页的值。

```
var onclick=function(profile,page){
    profile.boxing().setPage(page);
};

// a PageBar
linb.create("PageBar")
.set("1:5:12")
.onClick(onclick)
.show();

// another PageBar
linb.create("PageBar")
.set("1:5:12")
.setTop(100)
.setCaption("")
.setPrevMark("<<")
.setNextMark(">>")
.setTextTpl("[ * ]")
.onClick(onclick)
.show();
```

设置页码

同时设置 3 个值：最小页码，目前页码和最大页码

事件

注意：值里面包括三个部分

控件的标签

上一页的标签

下一页的标签

页标签模板，*是变量值

Page: 1 ... < 5 > ... 12

[1] [...] << [5] >> [...] [12]
[7] [8] [9] [10] [11]

5. 高级控件

linb 中还有一些高级控件是我以前在做实际项目的时候，根据用户的需求定制的。这些不在入门篇教程的讨论范围之内：

- | `linb.UI.TextEditor`: 为代码编辑（非即时高亮）定制的控制件。
- | `linb.UI.Range`: 为显示和编辑价格区间定制的控制件。
- | `linb.UI.Poll`: 为投票和调查定制控制件。
- | `linb.UI.StatusButtons`: 为多个状态定制的控制件。
- | `linb.UI.FoldingList`: 为仿 google group 定制的控制件。
- | `linb.UI.ColLayout`: igoogelayout 类似的控制件。
- | `linb.UI.Calendar`: 为时间管理（任务管理）功能定制的控制件。
- | `linb.UI.TimeLine`: 为时间管理（时间线管理）功能定制的控制件。

第三章 与后台服务的数据交互

linb 是一个纯客户端解决方案，也就是她不和任何后台进行绑定，前台和后台的数据交互是完全解耦的。前台不需要、也不关心用的是哪个后台，前台的工作只是向给定的服务接口（一般是一个 url）发送数据请求，然后接收数据返回结果即可。

linb 中的负责数据交互（IO）类有三个：linb.Ajax、linb.SAjax 和 linb.IAjax：

- l linb.Ajax：是对浏览器中 XMLHttpRequest 对象的封装。其特点是：
 - n 默认不能进行异域访问；
 - n 可以实现与服务端同步交互数据；
 - n linb.Ajax 可以通过 get 或 post 的方式来发送数据；
 - n 返回的内容为字符串。可以返回 xml 数据，JSON 数据或任何其他字符串。
- l linb.SAjax：是对“通过动态添加 script tag 来实现数据交互功能”的封装。其特点是：
 - n 可以进行异域访问；
 - n 可以处理较大的数据；
 - n 只能通过 get 的方式来发送数据；
 - n 返回的内容被封装为 javascript 的 Object，所以对服务端的返回数据格式有要求：

linb.SAjax 的发送的请求数据中会包括一个 callback 参数（回调函数名，默认为字符串“linb.SAjax.\$response”）和一个 id 参数（请求的唯一标识，假设为字符串“12483145855311”）。要求服务端返回的数据为：

```
linb.SAjax.$response({id: "12483145855311"/*,返回的其他数据*/})
```

返回的以上数据会被 linb.SAjax 直接放在一个动态生成的 script tag 里面，这样上述的返回数据就会直接执行 linb.SAjax.\$response 方法来实现当前数据请求的回调功能。

- l linb.IAjax：是对“通过动态添加 iframe 来实现数据交互功能”的封装。其特点是：
 - n 可以进行异域访问；
 - n 可以上传文件；
 - n linb.IAjax 可以通过 get 或 post 的方式来发送数据；
 - n 返回的内容被封装为 javascript 的 Object，所以对服务端的返回数据格式有要求：

linb.IAjax 的发送的请求数据中会包括一个 id 参数（请求的唯一标识，假设为字符串“12483145855311”）。要求服务端返回的数据为：

```
<script type='text' id='json'>{"id": "12483161278402"/*,返回的其他数据*/}}</script>
<script type='text/javascript'>
window.name=document.getElementById('json').innerHTML;
</script>
```

linb.request 是对以上三个 IO 类的封装，她会自动根据跨域情况和提交方法来判断使用 linb.Ajax、linb.SAjax 或 linb.IAjax 之中的一个。

注：本章的例子都要以 web 的方式来运行，不要双击直接打开。

第一节 最好先安装 Fiddler

为了能够清晰的了解 Ajax 的数据交流状况，需要一个能够监控网络数据流的工具。我一直用的是 Fiddle，你也可以用其他的功能相似软件来监控和调试网络数据流。

如果你还没有 Fiddler，可以到 <http://www.fiddler2.com/fiddler2/> 去下载一份最新的版本，并安装到你的机器上。Fiddler 运行后会自动配置 IE 的 proxy（配置之后，所有用 IE 的请求和返回数据都会在 Fiddler 里截获）。如果你的浏览器是 firefox，你需要手工配置一下 proxy（Fiddler 默认的地址和端口是：127.0.0.1:8888）。当然，你也可以用一些 firefox 的插件来管理 fiddler。

关于 Fiddler 的用法和 firefox 的插件用法在这里就不详述了。

要点是：在进行本章之前，请你最好要在本机配置好一个能够监控网络数据流的工具，并学会用它。

第二节 获取文件的内容

一般用 linb.Ajax 来获取文件内容。

```
linb.Ajax('data/ajax.js', " ", function(rsp){
    linb.message(rsp)
},function(errMsg){
    linb.alert(errMsg)
}).start();
```

文件或服务 uri 地址

传递的参数，可以输入一个随机数来避免浏览器缓存

返回成功的回调函数

返回错误的回调函数

是一个线程封装，需要调用 start 方法开始

用 Fiddler 来监控上面代码的运行结果，你会发现返回的是：



第三节 同步数据交换

只有 linb.Ajax 支持同步的数据交换。将 linb.Ajax 的最后一个参数（第 6 个参数 options，是一个键值对参数）的 asy 设置成 false，表示是一个同步数据请求。

```
var url="chapter3/request.php";
linb.Ajax(url, {
    key:'test',
    para:{p1:'para 1'}
},function(rsp){
    linb.log(rsp);
},function(errMsg){
    linb.alert(errMsg)
}, null, {
    asy:false
}).start();
```

同步数据交换

一下是用 fiddler 监控的个结果：

请求 url 为：

```
GET /jsLinb2.2/cases/chapter3/request.php?%7B%22key%22%3A%22test%22%2C%20%22para%22%3A%7B%22p1%22%3A%22para%201%22%7D HTTP/1.1
```

返回的数据为：

| Transformer | Headers | TextView | ImageView | HexView | WebView | JSON | Auth | Caching | Privacy | Raw |
|--|---------|----------|-----------|---------|---------|------|------|---------|---------|-----|
| {"data":[{"p1":"para 1","p2":"server_set","time":"2009-07-23 03:05:39","rand":"03-05-397jaso7bqm0f8op0rw"}]} | | | | | | | | | | |

这个是异步数据交换的对比例子：

```
var url="chapter3/request.php";
linb.Ajax(url, {
    key:'test',
    para:{p1:'para 1'}
},function(rsp){
    linb.log(rsp);
},function(errMsg){
    linb.alert(errMsg)
}).start();
```

请求数据

如果是在本地测试，可能在感觉上以上两个例子会基本相同。但要记住第一个是同步的数据交换（XMLHttp 没有返回数据前，浏览器是锁死的，不能继续操作），第二个是异步的数据交换（XMLHttp 没有返回数据前，浏览器上可以进行其他操作）。

第四节 与异域交换数据

linb.SAjax 和 linb.IAjax 可以与异域交换数据。但是在以下两种情况下不能用 linb.SAjax：

- ① 需要 post 数据
- ② 需要上传文件（也是 post 数据）

本节中代码的 url 实际是在本域的地址，你可以把它们放在一个异域来测试。

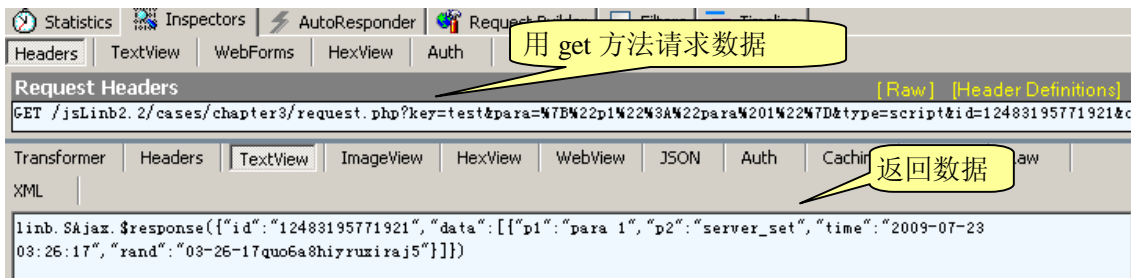
1. 对 SAjax 的数据流监控

以下是一个 SAjax 的示例，运行它，并用 fiddler 监控网络数据。

```
var url="chapter3/request.php";
linb.SAjax(url, {
  key:'test',
  para:{p1:'para 1'}
},function(rsp){
  linb.log(rsp);
},function(errMsg){
  linb.alert(errMsg)
}).start();
```

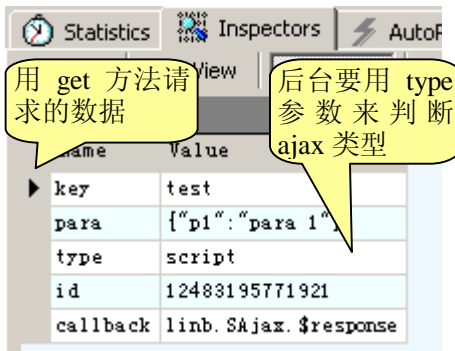
请求数据

Fiddler 的监控结果:



用 get 方法请求数据

返回数据



用 get 方法请求的数据

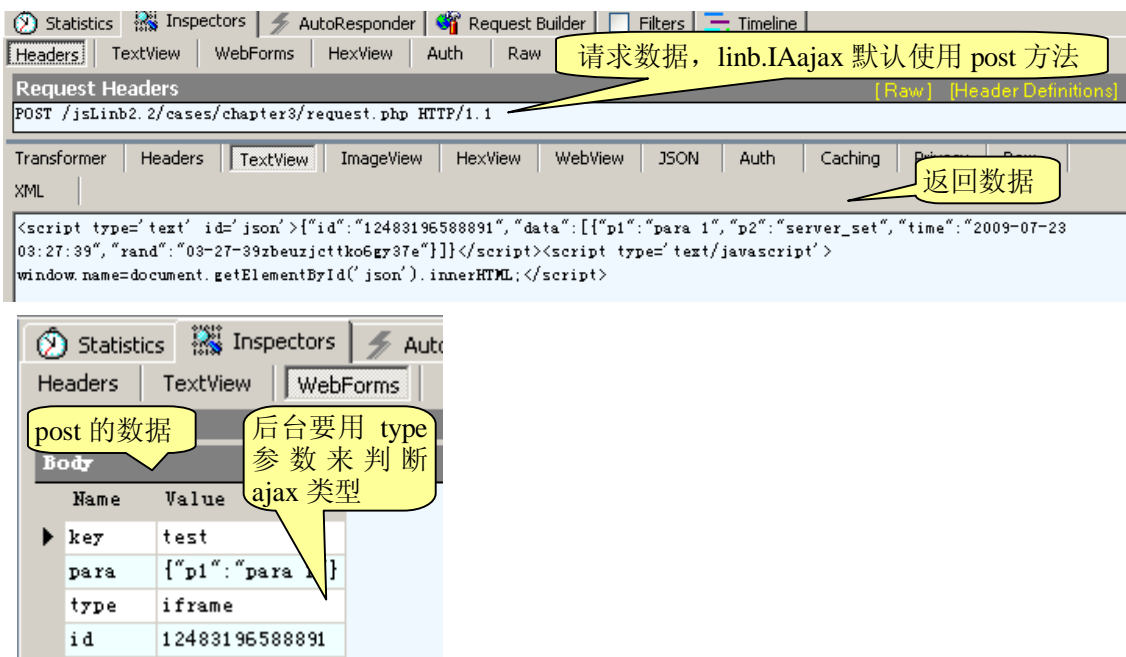
后台要用 type 参数来判断 ajax 类型

2. 对 IAjax 的数据流监控

以下是一个 IAjax 的示例，运行它，并用 fiddler 监控网络数据。

```
var url="chapter3/request.php";
linb.IAjax(url, {
  key:'test',
  para:{p1:'para 1'}
},function(rsp){
  linb.log(rsp);
},function(errMsg){
  linb.alert(errMsg)
}).start();
```

请求数据



IAajax 默认用的是 post 方式, 如果想用 get 方式, 可以设置 method 选项。

```
var url="chapter3/request.php";
linb.IAajax(url, {
  key:'test',
  para:{p1:'para 1'}
},function(rsp){
  linb.log(rsp);
},function(errMsg){
  linb.alert(errMsg)
},null,{
  method: 'get'
}).start();
```

请求数据

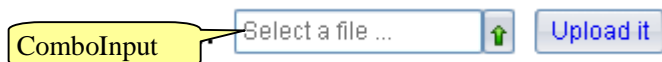
第五节 文件上传

文件上传只能用 IAajax。

本节中的代码在 chapter3/upload/ 目录下。

1. 用 ComboInput 来选择上传文件

选择上传文件用 linb.UI.ComboInput 控件, 将 type 设置为 “upload” 即可。



2. 用 IAjax 来执行上传

```

Class('App', 'linb.Com',{
  Instance:{
    initComponents:function(){
      // [[code created by jsLinb UI Builder
      var host=this, children=[], append=function(child){ children.push(child.get(0));

      append((new linb.UI.SLabel)
        .setHost(host,"label1")
        .setLeft(40)
        .setTop(44)
        .setCaption("Select your file: ")
      );

      append((new linb.UI.ComboInput)
        .setHost(host,"upload")
        .setLeft(140)
        .setTop(40)
        .setWidth(140)
        .setReadOnly(true)
        .setType("upload")
        .setValue("Select a file ...")
      );

      append((new linb.UI.SButton)
        .setHost(host,"sbutton3")
        .setLeft(290)
        .setTop(40)
        .setCaption("Upload it")
        .onClick("_sbutton3_onclick")
      );

      return children;
      // ]]code created by jsLinb UI Builder
    },
    _sbutton3_onclick:function (profile, e, src, value) {
      var file=this.upload.getUploadObj();
      if(file){
        linb.IAjax('../request.php',{key:'upload',para:{},file:file},function(rsp){
          linb.alert(rsp.data.message);
        },function(errMsg){
          linb.alert(errMsg)
        }).start();
      }
    }
  }
});

```

设计器生成的代码

选择文件控件

得到文件内容

用 IAjax 上传

成功返回

第六节 处理数据交换的基本函数模板

在实际的应用中，可以按照实际情况来选择 linb.Ajax、linb.SAjax 和 linb.IAjax 的应用。

大一点的项目对处理数据交换的功能做一下再封装比较好，把所有情况封装成一个统一的接口来处理与后台的交互。下面是一个用 `linb.observableRun`(在与后台交互的时候前台显示 busy)来封装 `linb.request` 的例子，仅供参考（以下代码为非运行示例，不要直接运行）。

```
request=function(service,
  requestData,
  onOK,
  onStart,
  onEnd,
  file
){
  _tryF(onStart);
  linb.observableRun(function(threadid){
    var options;
    if(file){
      requestData.file=file;
      options={method:'post'};
    }
    linb.request(service, requestData, function(rsp){
      if(typeof rsp=='string')
        rsp=_unserialize(rsp);

      if(rsp){
        if(!rsp.error)
          _tryF(onOK, [rsp]);
        else
          linb.pop(_serialize(rsp.error));
      }else{
        linb.pop(_serialize(rsp));
      }
      _tryF(onEnd);
    },function(rsp){
      linb.pop(_serialize(rsp));
      _tryF(onEnd);
    }, threadid,options)
  });
};
```

服务地址

请求的数据（键值对）

自定义的成功回调函数

文件

请求开始和结束时的回调函数

linb.Ajax 会返回字符串

成功

失败

失败

失败

第七节 XML 数据

如果从服务端返回的是 xml 数据，需要用 `linb.XML` 来把 XML 数据转换为 JSON 数据：

```
linb.Ajax('data/ajax.xml', "", function(rsp){
  alert (rsp)
  var obj = linb.XML.xml2json(linb.XML.parseXML(rsp));
  linb.pop(obj.message);
},function(errMsg){
  linb.alert(errMsg)
}).start();
```

XML 转换到 JSON

第八节 综合示例

文件夹 chapter3/io/下面是一个 IO 的综合示例。

how to use linb.absIO to interact with service

| | get | post | post file | cross domain | |
|-------------------|-----|------|-----------|----------------------------------|------|
| | | | | get | post |
| linb.Ajax | yes | yes | no | no | no |
| linb.SAjax | yes | no | no | yes (best for "return big data") | no |
| linb.IAjax | yes | yes | yes | yes | yes |

Request data:

```
{
  key:'test',
  para:{
    p1:'client_set'
  }
}
```

Response data:

```
{"p1":"client_set", "p2":"server_set", "time":"2009-07-23 03:48:55", "rand":"03-48-55hjww63a7kenm3h7wr"}
```

请求的数据

分别用 Ajax 的 get 和 post 方法

用 SAjax

分别用 IAjax 的 get 和 post 方法

用 linb.request 方法

返回结果窗口

第四章 分布式 UI 入门

有的时候，特别是一些较大的应用，需要把一些不常用的模块单独的分离出来。这些模块的代码并不会在程序初始化的时候被加载到浏览器。当应用调用到某个模块的时候，程序会自动从“远程文件”里加载这个模块的代码，生成实例，然后供其他的模块调用。这是一种分布式 UI 的概念，实现某个模块的“远程文件”可以在不同的服务器上，甚至是不同的域的服务器下。

第一节 来自远程 js 文件的对话框模块

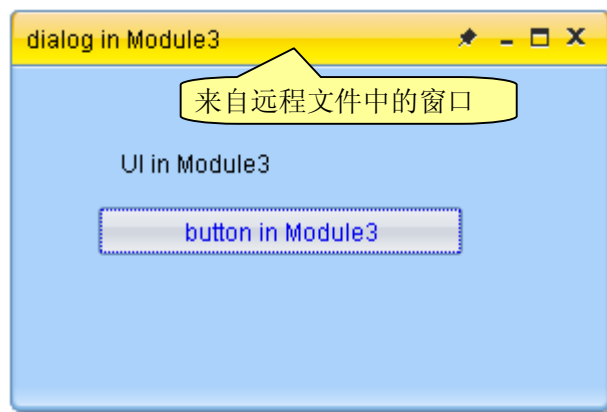
在 `cookbook\chapter4\distributed\App\js\` 文件夹下有一个 `Module3.js` 文件，文件中有一个类名为“`App.Module3`”的 UI 类。我们可以用下面的代码，来调用他：

```
Namespace("App");
linb.include("App.Module3",
    linb.getPath("chapter4/distributed/App/js/", "Module3.js"),
    function(){
        var ins=new App.Module3();
        ins.show();
    },function(){
        linb.alert("fail");
    }
);
```

这里先需要一个命名空间 `App`

动态异步加载远程文件

加载远程文件成功后，生成实例并显示



你也可以试着从另一个域内加载远程文件。如果这个异域远程文件存在的话，应该可以得到正确的结果。

```

Namespace("App");
linb.include("App.Module3",
"http://www.linb.net/Samples/linb/app/distributed/App/js/Module3.js",
function(){
    var ins=new App.Module3();
    ins.show();
},function(){
    linb.alert("fail");
}
);

```

文件来自异域

第二节 linb.Com 和 linb.ComFactory

事实上，实际的绝大部分企业应用都不会有从异域加载代码的需求。从另一个角度说，大部分的分布式 UI 代码都是从应用程序的制定目录下加载的。

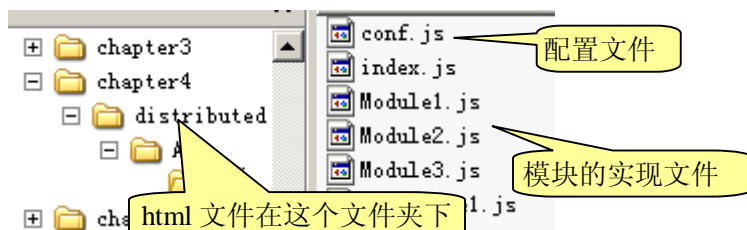
在这样的大多数情况下，我们可以用 linb.Com 和 linb.ComFactory 来方便的加载分布式 UI。用这种方法的前提是——需要动态加载的模块必须派生自 linb.Com，并按照指定规则来命名和放在指定的目录下。

linb.ComFactory 实现了一种对 linb.Com 管理的机制：

- ！ 可以按照指定的规则（这个规则可以从类名找到文件的位置）来从文件中加载代码；
- ！ 可以通过配置来管理多个继承自 linb.Com 的模块。

当然，对于那些需要从其他域下加载代码的需求，可以用本章第一节的方法来实现，或者用 iframe 的方法来实现（用 iframe 方法实现是一种相对安全的方式，但是无法实现代码间的交互）。

以 chapter4/distributed 下的综合例子为例，



注：本节中的代码摘自 chapter4/distributed 下的综合例子，不能直接在 env.html 中运行。

1. linb.ComFactory 的配置

我们首先来看一下 linb.ComFactory 的装配设置（在 conf.js 文件里面）：

```

CONF={ComFactoryProfile:
{
  module1:{
    cls:'App.Module1',
    children:{
      tag_SubModule1:'submodule1'
    }
  },
  submodule1:{
    cls:'App.SubModule1'
  }
}}

```

模块 module1 来自于 Aoo.Module1 类

模块 module1 里有一个子模块 submodule1

模块 submodule1 来自于 Aoo.SubModule1 类

在模块加载之前，这个配置文件中关于 linb.ComFactory 的装配信息部分会被设置到 linb.ComFactory 里面（代码子在 index.js 文件里面）：

```
linb.ComFactory.setProfile(CONF.ComFactoryProfile);
```

设置模块装配信息

2. 应用入口 linb.Com.Load

应用的入口 index.html 在 distributed 目录下，这个 html 文件中的代码：

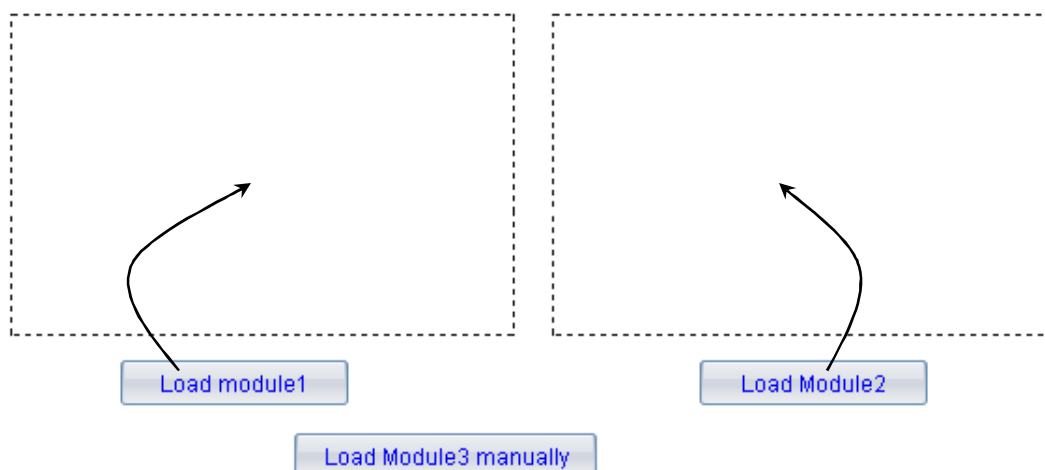
```
linb.Com.load ('App');
```

加载第一个 UI 类

会按照规则从 distributed/App/js/下查找 index.js 文件来加载 App 类的代码。App 类的 UI 在渲染到 Dom 之后为：

Loading code from outside dynamically!

Get Module code from out file on the fly, and append module UI to the current page



3. newCom 和 getCom

在 index.js 文件里面，“Load module3 manually” 的 onClick 事件代码为：

```
_button3_onclick:function (profile, e, value) {
  var host=this;
  linb.ComFactory.newCom('App.Module3',function(){
    this.show(linb([document.body]));
  });
}
```

用 newCom 方法来加载类

回调函数

这个参数要用类名

UI 加到 body 里

在代码中调用 `linb.CombFactory.newCom("App.Module3"..` 时，系统会按照规则从 `distributed/App/js/` 下查找 `Module3.js` 文件来加载 `App. Module3` 类的代码。加载成功后会调用回调函数。

可以注意到：**newCom 是不用配置文件的。**

“Load module1” 的 onClick 事件代码为：

```
_button9_onclick:function (profile, e, value) {
  var host=this;
  linb.ComFactory.getCom('module1',function(){
    var ns=this;
    host.div16.append(ns.getUIComponents(),false);
  });
}
```

用 getCom 方法来加载类

回调函数

这个参数要用配置中的模块名

UI 加到左侧的虚线框中

在代码中调用 `linb.CombFactory.getCom("module1"..` 时，系统会按照 `conf.js` 里面的装配配置，从 `distributed/App/js/` 下查找 `Module1.js` 文件来加载 `App. Module1` 类的代码。加载成功后即调用回调函数。

“Load module2” 的 onClick 事件代码为：

```
_button10_onclick:function (profile, e, value) {
  var host=this;
  linb.ComFactory.getCom('App.Module2',function(){
    var ns=this;
    host.div16.append(ns.getUIComponents(),false);
  });
}
```

用 getCom 方法来加载类

回调函数

getCom 用类名也是可以的，这样就不用配置了

UI 加到右侧的虚线框中

`getCom` 默认是 `singleton` 的，用 `getCom` 实例化的类会在 `CombFactory` 被缓存，下次再调用的时候不会再从远程文件中加载代码，也不回再次实例化。

可以把 `getCom` 方法的最后一个参数 `singleton` 设置为 `false`，那样的话 `getCom` 的作用就等同于 `newCom` 了。

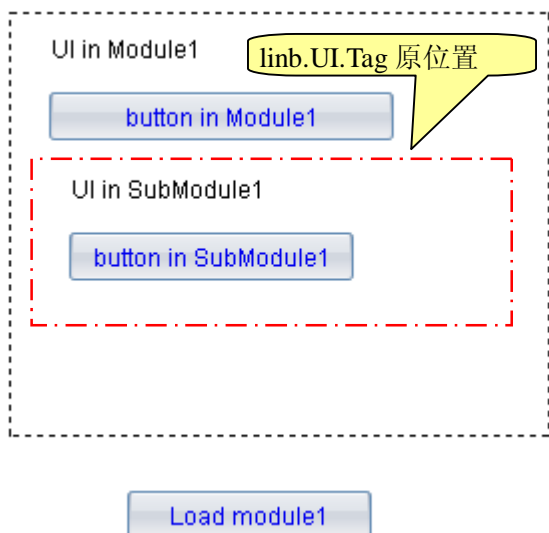
4. linb.UI.Tag 的作用

在 Module1.js 里面有一个 linb.UI.Tag 对象 tag2。

```
host.panelMain.append((new linb.UI.Tag)
    .host(host, "tag2")
    .setLeft(20)
    .setTop(70)
    .setWidth(218)
    .setHeight(98)
    .setTagKey("tag_SubModule1")
);
```

设置模块名字

这个 Tag 对象的作用是配置给“tag_SubModule1”模块配置大小和位置。按照装配信息，“tag_SubModule1”模块是“App.SubModule1”的实例，在 Module1 的实例产生时，系统会自动根据这个 Tag 对象的模块名来加载 SubModule1 模块并实例化它，并用 SubModule1 中的第一个 UI 控件的根 DOM 节点代替 Tag 的跟节点（Tag 被销毁）。



5. 销毁分布式 UI 模块

对于很大的系统来说，在一些不常用的模块被使用过后，需要销毁。彻底的销毁这样的分布式 UI 模块，需要两个步骤：销毁类的实例和销毁类本身。

销毁类的实例调用 linb.Com 的 destroy 方法即可，销毁类本身要用到 Class.destroy（“类的全名”）。当然，对于用 getCom 得到的 singleton 实例（会缓存）需要用到 linb.ComFactory.setCom（“模块的名字”，null）来清除缓存。

6. 对于代码已加载的 Com

如果是代码已加载的 Com，也可以不用 linb.ComFactory 来应用。可以直接地用以下方式：

```
linb('body').append(new App.Acom);
```

直接 new，然后加入 DOM

第五章 一些基本的问题

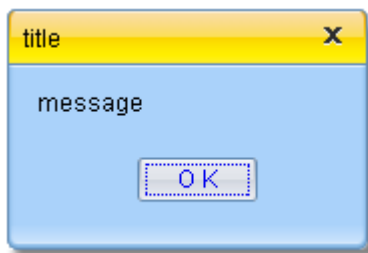
第一节 弹出窗口

1. alert

等同于 linb.UI.Dialog.alert，依赖于 linb.UI.Dialog，linb.UI.SButton。linb.alert 的窗口是一个 singleton 的实例。

```
linb.alert('title','message',function(){
    linb.message('You close this window!')
}, 'OK', 50, 100);
```

关闭窗口后触发事件



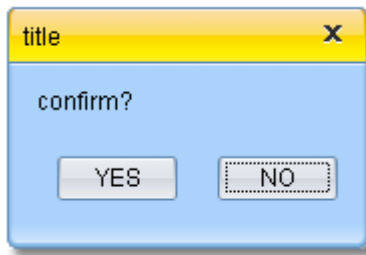
2. confirm

等同于 linb.UI.Dialog.confirm，依赖于 linb.UI.Dialog，linb.UI.SButton。linb.confirm 的窗口是一个 singleton 的实例。

```
linb.confirm('title','confirm?',function(){
    linb.message('You confirmed it')
},
function(type){
    linb.message(" You didn't cofirm it -" + type)
}, 'YES', 'NO', 50, 100);
```

点击 Yes 按钮后触发的事件

点击 No 按钮(或关闭窗口)后触发的事件



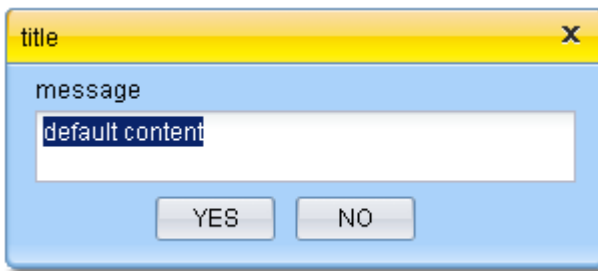
3. prompt

等同于 `linb.UI.Dialog.prompt`，依赖于 `linb.UI.Dialog`，`linb.UI.SButton`，`linb.UI.Input`。
`linb.prompt` 的窗口是一个 `singleton` 的实例。

```
linb.prompt('title', 'message', 'default content', function(msg){
    linb.message('You input - ' + msg)
}, function(){
    linb.message(" You cancel it")
}, 'YES', 'NO', 50, 100);
```

点击 OK 按钮后触发的事件

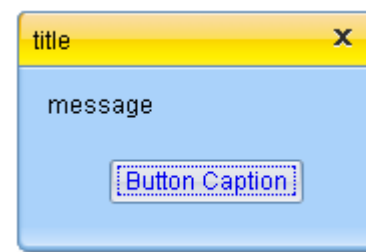
点击 Cancel 按钮(或关闭窗口)后触发的事件



4. pop

等同于 `linb.UI.Dialog.pop`，依赖于 `linb.UI.Dialog`，`linb.UI.SButton`，`linb.UI.Input`。

```
linb.pop('title', 'message', 'Button Caption', 50, 100);
```



第二节 异步执行

1. asyRun

`_asyRun` 函数是对 javascript 函数 `setTimeout` 的封装。

```

_asyRun(function(arg1,arg2){
    linb.pop("this===linb:"+(this===linb), arg1+"."+arg2)
},
500,
['arg1','arg2'],
linb)

```

要异步执行的函数

500 毫秒后执行

函数的参数

scope

2. resetRun

`_resetRun` 也是异步执行，与 `_asyRun` 不同的是它可以设置一个唯一标识，唯一标识相同的异步执行函数在系统中最多保留一个（也就是后一个会覆盖前一个）。

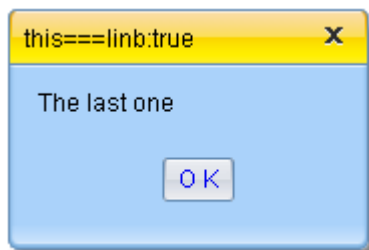
```

_resetRun('key1',function(arg){
    linb.pop("this===linb:"+(this===linb), arg)
},500,['The previous one'],linb)

_resetRun('key1',function(arg){
    linb.pop("this===linb:"+(this===linb), arg)
},500,['The last one'],linb)

```

最后会运行这个函数



第三节 改变皮肤

1. 改变系统皮肤

系统默认自带的有三套皮肤，分别为：`default`、`vista` 和 `aqua`。可以用 `linb.UI.setTheme` 来改变当前页面的皮肤。


```
linb.UI.setTheme("vista")
```

动态换皮肤（不会刷新页面）

2. 改变单个控件的皮肤

细粒度上，在 linb.UI 实例上调用 setTheme 提供了只改变某一个或一组控件皮肤的功能。用这种方式改变的控件皮肤不会用到系统皮肤的 css 设置。需要开发者单独来定义相应的 css。

```
ctl_button.setTheme("custom")
```

只改变 ctl_button 的皮肤（不会影响其他的控件）

第四节 改变页面的语言显示

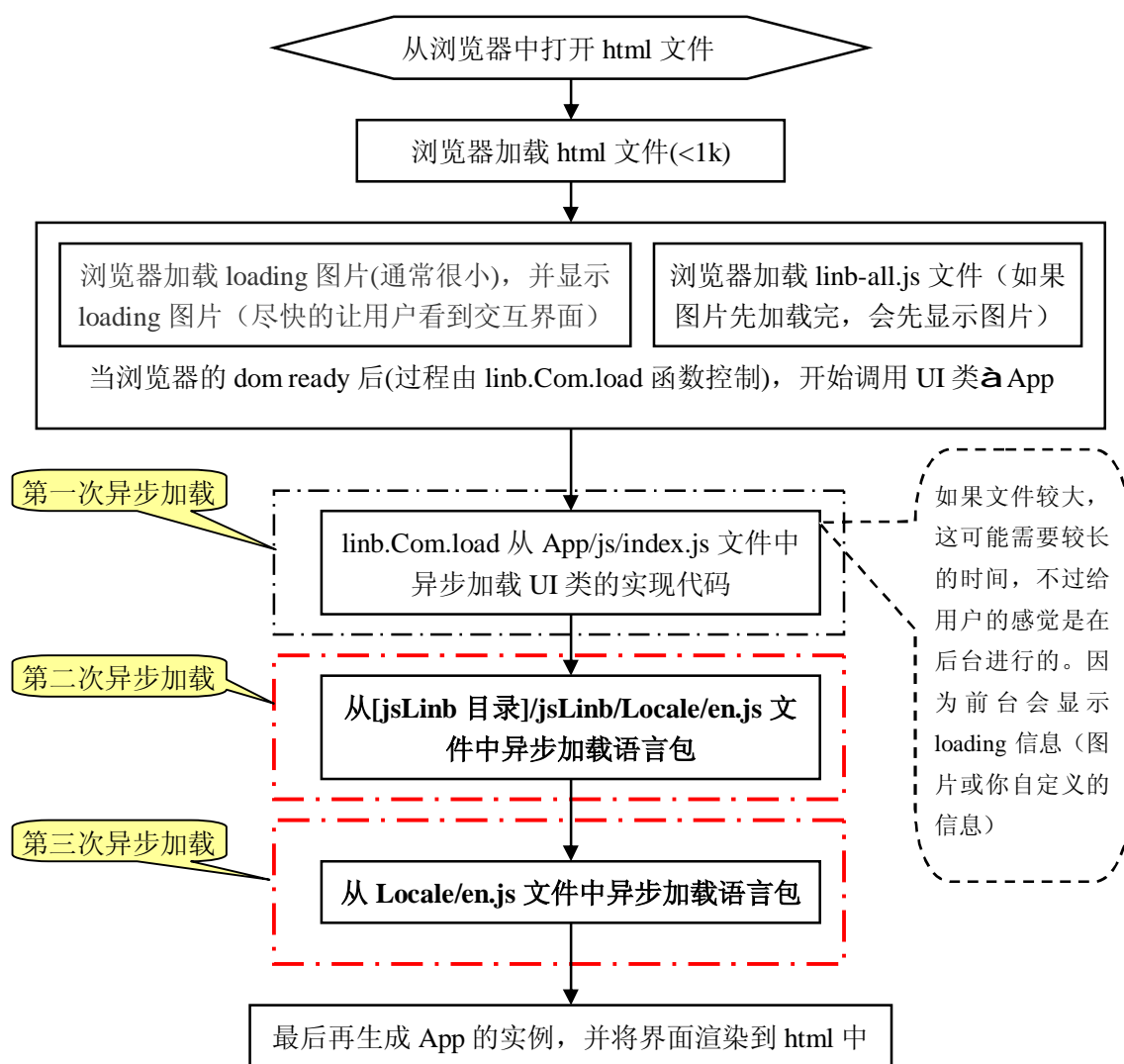
```
linb.alert(linb.getLang())
```

得到当前语言关键字

```
linb.setLang("cn");
```

动态改变语言（不会刷新页面）

例子 chapter5\lang 的文件加载过程：



第五节 DOM 节点的基本操作

类 `linb.Dom` 是针对 DOM 节点的跨浏览器封装, 里面的功能主要包括: DOM 节点的生成和移除; DOM 节点的属性管理; DOM 节点的 css 属性管理; DOM 节点事件的管理。

1. 节点的生成和插入

以下是一个 DOM 节点生成和插入的例子。生成节点的函数是 `linb.create`; 插入节点有 4 个函数: `append` (插入到内部的最后处), `prepend` (插入到内部的最前处), `addPrev` (插入到外部的前面) 和 `addNext` (插入到外部的后面)。

```

var firstNode;
linb('body').append(firstNode=linb.create('<div style="border:solid 1px">the first div</div>'));
firstNode.append('input')
.last().attr('value','append')
.parent()
.prepend('<button>prepend</button>')
.addPrev('<div style="border:solid 1px">prev div</div>')
.addNext('<div style="border:solid 1px">div</div>')

```

第一个节点

加到内部最后

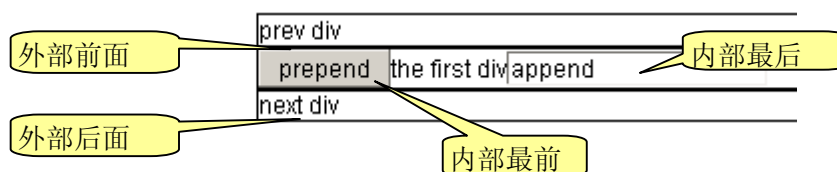
到 input, 并设置 value

再回到 firstNode

加到内部最前

加到外部前面

加到外部后面



2. 节点属性和 CSS 属性

可以用 css 方法来改变 DOM 节点的 css 样式, 用 attr 方法来改变 DOM 节点的属性值。

```

var node;
linb('body').append(node=linb.create('div'));
node.html('content<input value="ini">');
_.asynRun(function(){
  node.css('border','solid 1px');
},1000);
_.asynRun(function(){
  linb.message(node.css('fontSize'))
  node.css({background:'#00ff00',fontSize:'16px'});
},2000);
_.asynRun(function(){
  node.attr('style','border:none;font-size:18px;');
},3000);
_.asynRun(function(){
  linb.message(node.last().attr('value'))
  node.last().attr('value','updated');
},4000);

```

加入内容

改变 CSS border

得到 CSS fontSize

改变 fontSize 和背景

改变整个 style

得到 input 的 value 属性

改变 input 的 value 属性

3. 对 CSS 类的操作

linb 中对 css 类(className 属性)操作的函数有 4 个:

- 1 addClass: 为当前 DOM 节点加一个或多个 css 类。

- l removeClass: 从当前 DOM 节点中移除一个 css 类。
- l replaceClass: 按照给定的正则表达式来代替当前 DOM 节点的 css 类。
- l tagClass: 为当前 DOM 节点的所有 css 类添加（移除）css 类。

```

var node;
linb('body').append(node=linb.create('div'));

_._asyRun(function(){
  node.addClass("cls1 cls2 cls3");
  linb.message(node.hasClass('cls2'));
  node.text(node.attr('className'));
},1000);

_._asyRun(function(){
  node.removeClass("cls2");
  node.text(node.attr('className'));
},2000);

_._asyRun(function(){
  node.replaceClass(/cls/g,"class");
  node.text(node.attr('className'));
},3000);

_._asyRun(function(){
  node.tagClass("-mouseover",true);
  node.text(node.attr('className'));
},4000);

_._asyRun(function(){
  node.tagClass("-mouseover",false);
  node.text(node.attr('className'));
},6000);

```

加 CSS 类

判断是否存在该类名

移除此类名

改变类名

为每一个类附加一个 tag 类

按照规则清除所有附加类

4. 操作 Dom 事件

linb 的 DOM 事件里面可以有三组事件函数：[before 开头]的组，[on 开头] 的组和 [after 开头] 的组，每一组都是一个函数数组。

- l linb(/**/).onClick([function], 'label'): 为 [onclick] 事件函数组添加一个标签为'label'的事件函数。
- l linb(/**/).onClick([function]): 先清空 [onclick] 事件函数组,再重新加入一个[function] 函数。
- l linb(/**/).onClick(null, 'label') : 从 [onclick] 事件函数组中删除标签为'label'的事件函数。
- l linb(/**/).onClick(null) : 清空 [onclick] 事件函数组。
- l linb(/**/).onClick(null,null,true) : 清空 [beforeclick]、[onclick]和[afterclick] 事件函数组。
- l linb(/**/).onClick(): 触发事件，会按照顺序执行所有[onclick]事件函数组里面的事件函数。如果其间任何一个事件函数返回[false]，余下的事件函数将不被执行。
- l linb(/**/).onClick(true): 触发事件，会按照顺序执行所有[beforeclick]、[onclick]和

[afterclick]事件函数组里面的事件函数。如果其间任何一个事件函数返回[false]，余下的事件函数将不被执行。

```
var node;
linb('body').append(node=linb.create("<button>click me</button>"));

node.onClick(function(){
    alert('onClick');
    return false;
})
.beforeClick(function(){
    alert('beforeClick');
})
.afterClick(function(){
    alert('afterClick');
});

node.onClick(true);

_$.asyncRun(function(){
    node.onClick(null);
    node.onClick(true);
},2000);
```

加一个 onClick 事件

加一个 beforeClick 事件

加一个 afterClick 事件

触发全部 click 事件，相当于鼠标左键点击了按钮。
因为 onClick 返回了 false，afterClick 不会被触发。

清除掉 onClick 事件；其他两个事件还在。

再次触发全部的 click 事件。这次由于
onClick 已经不在。afterClick 会被触发。

5. 拖拽 Dom 节点

可以用 draggable 方法来设置某个 DOM 节点是否可以拖拽，用 droppable 方法来设置某个 DOM 节点是否可以允许拖拽的东西放下。

```
var btn,div;
linb('body').append(btn=linb.create("<button>drag me</button>"))
.append(div=linb.create("<div style='position:absolute;left:100px;top:100px;border:solid 1px;width:150px; height:150px;display:block;'> drop here </div>"))

btn.draggable(true,{dragType:'text'})
div.droppable(true,'dragkey')
.onDrop(function(){
    linb.alert(linb.DragDrop.getProfile().dragData);
});
```

设置拖拽

设置拖拽可以放在这里

onDrop 事件

drag me



1) 拖拽概要对象

“draggable”方法的第二个参数的作用是设置拖拽概要对象（一个键值对）。拖拽概要对象可以用 `linb.DragDrop.getProfile` 方法来得到，里面有以下参数可以用“draggable”方法的第二个参数来设置：

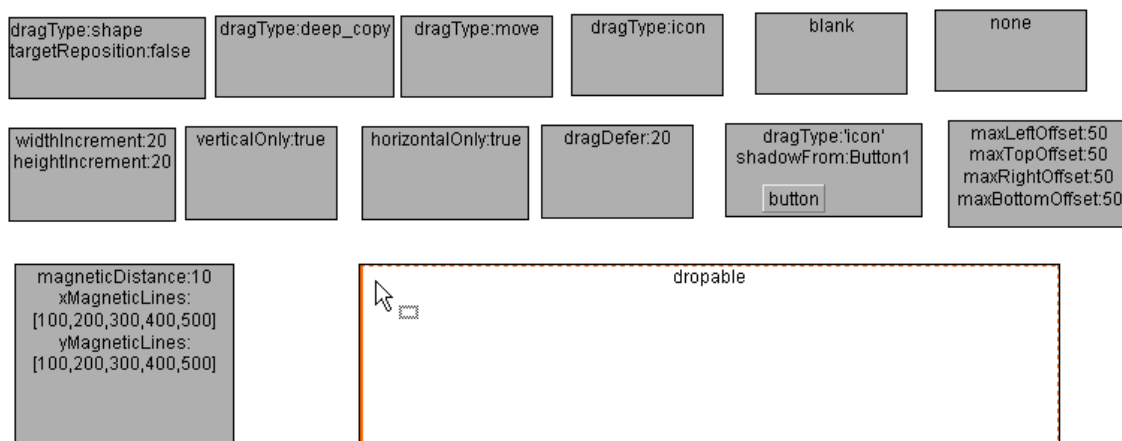
- | `dragType`: 可以是 'move','copy','deep_copy','shape','icon','blank' 或 'none'; 设置拖拽类型，默认为 'shape'。
- | `shadowFrom`: 可以是 DOM 节点，`linb.Dom` 对象或一个 `linbid` 字符串值；设置拖拽的时候要显示哪个 Dom 节点的影子。当 `dragType` 为 'icon' 的时候才有效。
- | `targetReposition`: 布尔值；设置是否最后要重置拖拽目标的位置，默认为 `[true]`。
- | `dragIcon`: 字符串；设置拖拽的时候显示图标的路径，默认为 `[linb.ini.path + 'ondrag.gif']`。
- | `magneticDistance`: 整数；设置磁性距离，默认为 0。
- | `xMagneticLines`: 整数数组；设置水平方向的磁性线数组，默认为 `[]`。
- | `yMagneticLines`: 整数数组；设置垂直方向的磁性线数组，默认为 `[]`。
- | `widthIncrement`: 整数；设置水平方向的最小增量值，默认为 0。
- | `heightIncrement`: 整数；设置垂直方向的最小增量值，默认为 0。
- | `dragDefer`: 整数；设置拖拽的延迟值。表示在`[document.onmousemove]`事件触发几次后拖拽才真正开始；默认为 0。
- | `horizontalOnly`: 布尔值；设置是否要只在水平方向拖拽，默认为 `[false]`。
- | `verticalOnly`: 布尔值；设置是否要只在垂直方向拖拽，默认为 `[false]`。
- | `maxBottomOffset`: 整数；设置下方向的最大拖拽距离，默认为 `[null]`。
- | `maxLeftOffset`: 整数；设置左方向的最大拖拽距离，默认为 `[null]`。
- | `maxRightOffset`: 整数；设置右方向的最大拖拽距离，默认为 `[null]`。
- | `maxTopOffset`: 整数；设置上方向的最大拖拽距离，默认为 `[null]`。
- | `targetNode`: 可以是 DOM 节点或 `linb.Dom` 对象；设置拖拽的目标。
- | `targetCSS`: 整数；设置拖拽目标的 CSS 键值对，默认为 `[null]`。
- | `dragKey`: 字符串；设置拖拽的数据键，默认为 `[null]`。
- | `dragData`: 对象；设置拖拽的具体数据，默认为 `[null]`。
- | `targetLeft`: 整数；设置拖拽目标的横向坐标，默认为 `[null]`。
- | `targetTop`: 整数；设置拖拽目标的纵向坐标，默认为 `[null]`。
- | `targetWidth`: 整数；设置拖拽目标的宽，默认为 `[null]`。

- | **targetHeight**: 整数; 设置拖拽目标的高, 默认为 `[null]`。
- | **targetOffsetParent**: 整数; 可以是 DOM 节点或 `linb.Dom` 对象; 设置拖拽目标的定位父元素(`offsetParent`), 默认为 `[null]`。

拖拽概要对象中还包括以下只读参数:

- | **dragCursor**: 可以是 `'none'`, `'move'`, `'link'`, 或 `'add'`, 得到鼠标当前的形状。
- | **x**: 整数; 得到鼠标当前的 X 值。
- | **y**: 整数; 得到鼠标当前的 Y 值。
- | **ox**: 整数; 得到鼠标最初的 X 值。
- | **oy**: 整数; 得到鼠标最初的 Y 值。
- | **curPos**: 键值对; `{left: 整数, top: 整数}`, 得到拖拽对象目前的 css 位置值。
- | **offset**: 键值对; `{x: 整数, y: 整数}`, 得到拖拽对象目前 css 位置相对于最初位置的偏离值。
- | **isWorking**: 布尔值; 得到拖拽是否在工作状态。
- | **restrictedLeft**: 整数; 得到拖拽在左侧的边界位置。
- | **restrictedRight**: 整数; 得到拖拽在右侧的边界位置。
- | **restrictedTop**: 整数; 得到拖拽在上侧的边界位置。
- | **restrictedBottom**: 整数; 得到拖拽在下侧的边界位置。
- | **proxyNode**: `linb.Dom` 对象; 得到当前拖拽代理的 DOM 元素。
- | **dropElement**: 字符串; 得到可以放下(drop)当前拖拽的 DOM 元素的 `linbid`。

在 `chapter3/dd/`下的 `ddProfile.html` 是一个“**draggable**”和“**droppable**”方法的综合实例。



2) 拖拽的相关事件

对于被拖拽的 DOM 节点, 事件有:

- | **onDragbegin**: 拖拽开始;
- | **onDrag**: 拖拽进行中;
- | **onDragstop**: 拖拽完毕。

对于可放置拖拽的 DOM 节点, 事件有:

- | **onDragenter**: 拖拽进入;

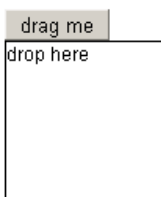
- l onDragleave: 拖拽离开;
- l onDragover: 拖拽在可 drop 区域移动;
- l onDrop: drop 拖拽。

```
var btn,div,elist;
linb('body').append(btn=linb.create("<button>drag me</button>"))
.append(div=linb.create("<div style='border:solid 1px;width:100px;height:100px;'>drop here</button>"))

linb('body').append(elist=linb.create('<div
style="position:absolute;left:140px;top:40px;width:600px;height:400px;overflow:auto;"></div>'))

btn.draggable(true,{dragType:'icon','dragkey','dragdata'})
.onDragbegin(function(){
    elist.append('<strong>onDragbegin </strong>');
})
.onDrag(function(){
    elist.append('<em>onDrag </em>');
})
.onDragstop(function(){
    elist.append('<strong>onDragend </strong>');
})

div.droppable(true,'dragkey')
.onDragenter(function(){
    elist.append('<strong>onDragenter </strong>');
})
.onDragover(function(){
    elist.append('<em>onDragover </em>');
})
.onDragleave(function(){
    elist.append('<strong>onDragleave </strong>');
})
.onDrop(function(){
    elist.append('<strong>onDrop </strong>');
});
```



onDragbegin onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag
onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag onDrag
onDrag onDrag onDrag **onDragenter** onDragover onDrag onDragover onDrag onDragover onDrag
onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag
onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag
onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag onDragover onDrag
onDragend onDrop

第六节 模板——linb.Template

模板类 `linb.Template` 是一个目的在于实现跨浏览器界面控件的封装，它完全不依赖于 `linb.UI` 及其所有派生类。如果你只需要实现比较简单的 UI 功能，并且认为 `linb.UI` 系的代码量过于庞大，就可以选择 `linb.Template` 来构建你的应用。

1. 一个较简单的模板应用

一个 `linb.Template` 主要有三个方面：模板（`template`），属性（`properties`）和事件（`events`）：

- ┌ 模板（`template`）：html 的模板字符串，大括号和里面的内容是需要代替的变量；是一个“键值对”对象。
- ┌ 属性（`properties`）：属性用来替换模板里面的变量；是一个“键值对”对象。
- ┌ 事件（`events`）：是对 DOM 节点的事件定义，并非 `linb.Template` 输出的事件；是一个“键值对”对象。

模板（`template`），属性（`properties`）和事件（`events`）“键值对”里面的“键”是对应的。

```
var tpl=new linb.Template;
tpl.setTemplate("<div [event]>{con}</div>")
//tpl.setTemplate({root:"<div [event]>{con}</div>"})
.tpl.setProperties({
  con:'click me'
})
.tpl.setEvents({
  root:{
    onClick:function(){
      linb.alert('Hi');
    }
  }
})
.show()
```

表示有事件

设置模板框架，等同于下面一行

设置属性，这里不用 root 关键字

设置 root 的事件

click me



2. 稍微复杂点的模板应用

下面是一个稍微复杂点的模板应用例子。

```

(tpl=new linb.Template)
.setTemplate({
  root : "<div style='width:200px;border:solid 1px;'><h3>{ head}</h3><ul>{ items}</ul><div
style='clear:both;'></div></div>"
  items: "<li [event] 'style=padding:4px;border-top:dashed 1px;'><div><div><a href='{href}'><img
src='{src}'></p><p>{ price}</p></a></div></div><div><a
href='{href}'><h4>{ title}</h4><div>{ desc}</div></a></div></li>"
})
.setEvents({
  items:{
    onmouseover:function(profile,e,src){
      linb.use(src).css('backgroundColor','#EEE');
      //Tips
      var item=profile.getItem(src),
      tpl=new linb.Template({ "root": "<div style='text-align:center;border:solid
1px;background:#fff;'><h4>{ title}</h4><div><p>{ desc}</p>" },item),
      html=tpl.toHtml();
      linb.Tips.show(linb.Event.getPos(e),html);
    },
    onmouseout:function(profile,e,src){
      linb(src).css('backgroundColor','transparent');
      linb.Tips.hide();
    }
  }
})
.setProperties({
  head:"On sale products",
  items:[{ id:"a", href:"#", price:"$ 18.99", title:"product #0", desc:"product #0 is on sale now!" },
        { id:"b", href:"#", price:"$ 23.99", title:"product #1", desc:"product #1 is on sale now!" },
        { id:"c", href:"#", price:"$ 23.99", title:"product #2", desc:"product #2 is on sale now!" } ]
})
.show()

```

有子模板

root 必须要有

Items 下的事件设置

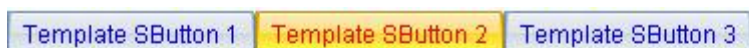
root 下的属性

items 下的属性



3. 用 linb.Template 实现一个 SButton 吧

在 **chapter5\SButton** 目录下是一个用 linb.Template 封装的 SButton, 实现了 linb.UI.SButton 的基本功能。虽然一个简易的版本, 但要加载的代码量要少的很多。

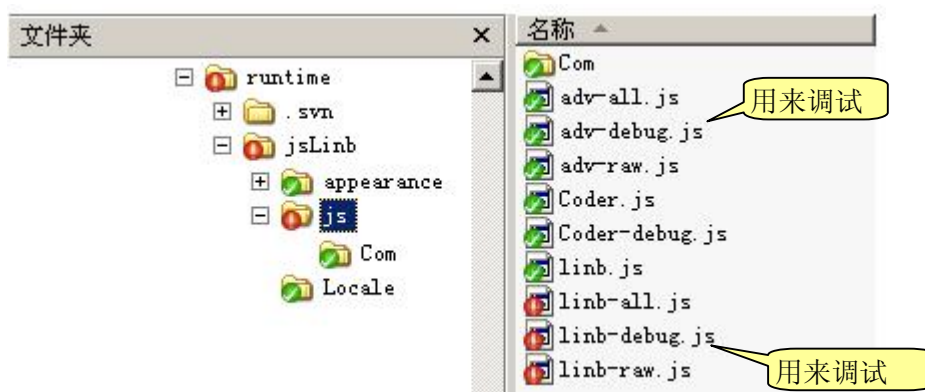


第七节 关于调试

无论用什么框架，或者是自己完全实现代码，做跨浏览器的重客户端程序都需要大量的调试工作。如果没有合适的工具或方法，代码的调试将会浪费掉你的大部分精力。在这里，针对代码调试，有以下三点要说明。

1. 用于调试的代码包

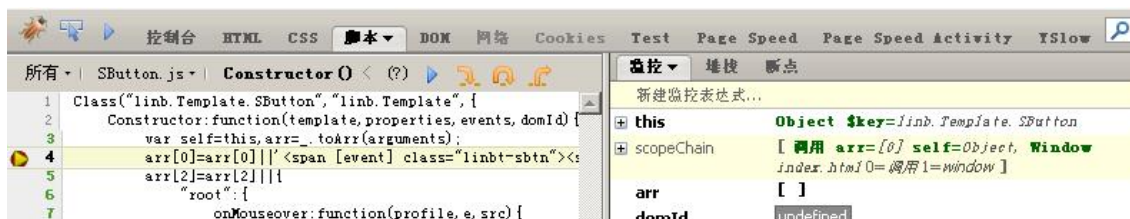
在 runtime/jsLinb/js/目录下，以“-debug.js”结束的文件是用来 debug 的代码包，这些代码包中的代码都保留了回车和空格，适合用来给 bug 定位。



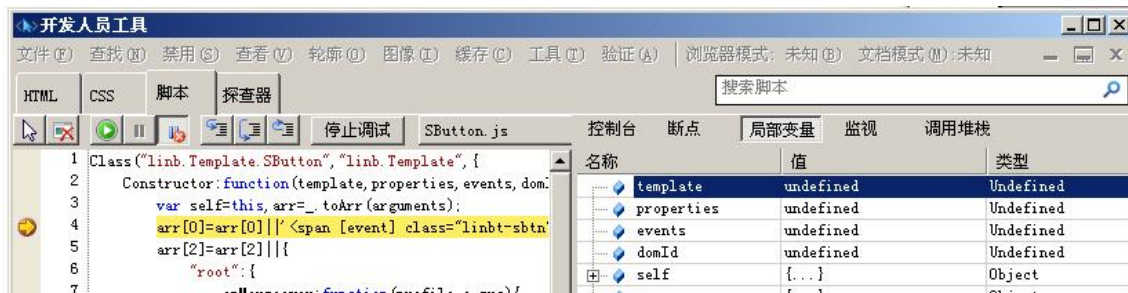
2. 调试工具

调试代码的时候推荐用 firefox，IE8、chrome 或 opera10 的“开发者工具”也可以。

Firefox 下的调试工具首选 firebug:



IE8 下用“开发人员工具”来调试：



其他的 IE 版本推荐用 VS2005 或 2008 的 web developer 工具来调试。

3. linb 的内置工具

linb 内置了一个可以 Trace 信息的工具。这个工具虽然不能真正的起到代码调试功能，但是却可以跨浏览器应用的。有的时候，把信息 log 出来对于调试程序是至关重要的。

想弹出 linb 的 Log 信息窗口，只需在代码中调用 `linb.log(xxx)` 即可。



第六章 一些典型的需求

第二节 布局

1. 用 dock 完成布局

如果要一个不需要拖拽来改变大小的布局，可以直接用控件的 dock 及其相关的属性。

```
linb.create('Block',{dock:"top",
    height:80,html:'top'
}).show();
linb.create('Block',{dock:"bottom",
    height:30,html:'bottom'
}).show();
linb.create('Block',{dock:"left",
    width:150,html:'left'
}).show();
linb.create('Block',{dock:"right",
    width:150,html:'right'
}).show();
linb.create('Block',{dock:"fill",
    html:'main',
    dockMargin:{left:10,right:10,top:10,bottom:10}
}).show();
```

上停靠

下停靠

左停靠

右停靠

充满空间

可设置外间距



2. 用 Layout 控件完成布局

如果要一个需要拖拽来改变大小的布局，可以用 linb.UI.Layout 控件。

```

var layout1=linb.create('Layout',{type:'vertical',
  items:[{
    pos:'before',
    id:'top',
    size:80
  },{
    pos:'after',
    id:'bottom',
    size:30
  }]
}).show();

linb.create('Layout',{type:'horizontal',
  items:[{
    pos:'before',
    id:'top',
    size:150
  },{
    pos:'after',
    id:'bottom',
    size:150
  }]
}).show(layout1);

```

垂直方向的 layout

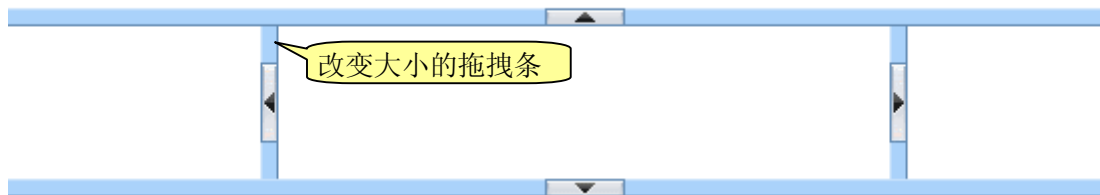
上方的容器

下方的容器

水平方向的 layout

左方的容器

右方的容器



如果想要“改变大小的拖拽条”消失，只需要在 items 项目数据中设置 locked 为 true 即可。

3. 相对位置布局

运行以下代码，并多次点击“Add content”按钮。

```

linb.create('Pane',{position:'relative',
    width:"auto",height:80,html:"the top div"
})
.setCustomStyle({"KEY":"border:solid 1px #888"})
.show()

var pane=linb.create('Pane',{position:'relative',
    width:"auto",height:"auto",
    html:"<strong>auto height</strong>"
})
.setCustomStyle({"KEY":"border:solid 1px #888"})
.show()

linb.create('Pane',{position:'relative',
    width:"auto",height:100,
    html:"<strong>bottom</strong>"
})
.setCustomStyle({"KEY":"border:solid 1px #888"})
.show()

linb.create("SButton")
.setLeft(140)
.setTop(30)
.setCaption("Add content")
.onClick(function(){
    pane.append(linb.create("Pane").setPosition("relative").setHeight(30).append("Input"))
})
.show()

```

上方的容器

中间的容器

下方的容器

向中间的容器加 relative 内容



在实际的项目应用中，可以结合以上三种方法来实现各种布局需求

第三节 UI 拖拽

1. 在容器面板间拖拽控件

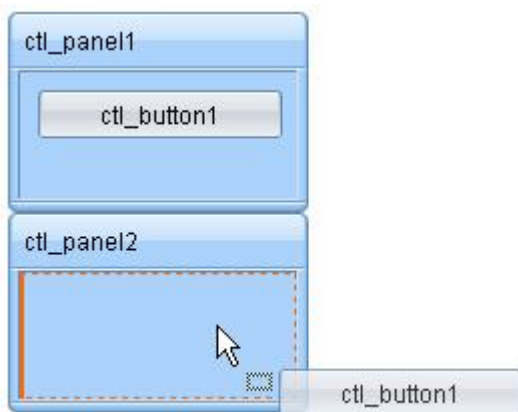
所有带有容器面板的控件都能加入子控件。

```
var panel1=linb.create('Panel',{position:'relative', dock:'none',width:150}).show();
var panel2=linb.create('Panel',{position:'relative', dock:'none',width:150}).show();
var btn=linb.create('Button',{left:10,top:10}).show(panel1);
var onDrop=function (profile, e, node, key, data) {
    var dd = linb.DragDrop.getProfile(), data = dd.dragData;
    if(data){
        var btn=linb.getObject(data);
        profile.boxing().append(btn.boxing());
    }
};
btn.draggable('iAny',btn.get(0).getId(),null,{shadowFrom:btn.getRoot()});
panel1.setDropKeys('iAny').onDrop(onDrop);
panel2.setDropKeys('iAny').onDrop(onDrop);
```

设置 onDrop 函数

设置 draggable

设置 droppable

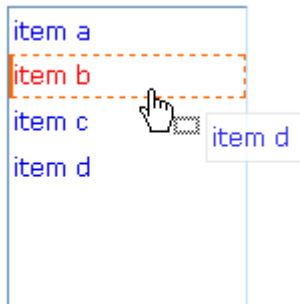


2. List 拖拽排序 1

List 的拖拽排序只需要分别设置控件的 dragKey 属性和 dropKeys 属性，并且 dropKeys 属性的值包含 dropKeys 属性的值即可。

```
linb.create("List",{
    items:["item a","item b","item c","item d"]
})
.setDragKey("list")
.setDropKeys("list")
.show()
```

分别设置托和拽的键值



3. List 拖拽排序 2

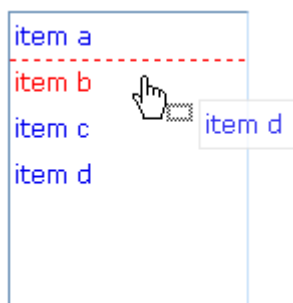
上个例子中，可 drop 的样式（左面是红色实线，其他三面是虚线的样式）是胸膛那个默认的，如果有需要我们可以通过 `onDropMarkShow` 和 `onDropMarkClear` 这两个事件来自定义。

```
linb.create("List",{
  items:["item a","item b","item c","item d"]
})
.setDragKey("list")
.setDropKeys("list")
.onDropMarkShow(function(profile,e,src,key,data,item){
  if(item){
    linb.DragDrop.setDragIcon('move');
    linb.DragDrop.setDropFace(null);
    profile.getSubNodeById('ITEM',item.id).css('borderTop','dashed 1px');
    return false;
  }
})
.onDropMarkClear(function(profile,e,src,key,data,item){
  if(item){
    linb.DragDrop.setDragIcon('none');
    profile.getSubNodeById('ITEM',item.id).css('borderTop','');
    return false;
  }
})
.show()
```

分别设置托和拽的键值

定制化可 drop 的样式

恢复样式



第四节 Form 表单

为了最求更精细的控制（类似于 DELPHI 或 VB）和实现灵活的设计器，linb 中去除了表单控件的概念（也就是没有封装 html form 元素的控件）。所以，在我们要提交数据前，需要先收集数据。

1. 表单 1

```

Class.destroy('App');
Class('App', 'linb.Com',{
  Instance:{
    initComponents:function(){
      // [[code created by jsLinb UI Builder
      var host=this, children=[], append=function(child){ children.push(child.get(0));
      append((new linb.UI.SLabel)
        .setHost(host,"slabel1").setLeft(80).setTop(60).setWidth(44).setCaption("Name:"));
      append((new linb.UI.SLabel)
        .setHost(host,"slabel2").setLeft(80).setTop(90).setCaption("Age:").setWidth(44));
      append((new linb.UI.Input)
        .setHost(host,"iName").setLeft(130).setTop(60).setValueFormat("[^.*]").setValue("Jack"));
      append((new linb.UI.ComboBox)
        .setHost(host,"iAge").setLeft(130).setTop(90).setType("spin").setIncrement(1).setMin(20).s
      etMax(60).setValue("35"));
      append((new linb.UI.SCheckBox)
        .setHost(host,"cFull").setLeft(130).setTop(130).setCaption("Full time"));
      append((new linb.UI.SButton)
        .setHost(host,"submit").setLeft(130).setTop(170).setCaption("SUBMIT").onClick("_submit
      _onclick"));
      return children;
      // ]]code created by jsLinb UI Builder
    },
    _submit_onclick:function (profile, e, src, value) {
      if(!this.iName.checkValid()){
        linb.alert("You must specify Name");
        return;
      }
      var name=this.iName.updateValue().getValue(), age=this.iAge.updateValue().getValue(),
        full=this.cFull.updateValue().getValue();
      linb.alert(_serialize({name:name,age:age,full:full}))
    }
  }
});
(new App).show();

```

设计器辅助生成

事件

验证

收集数据



2. 表单 2

从上一节中看到，显然用 linb 提交一个表单显得有些复杂。在实际项目中，我们一般用设计器（非常容易的通过拖拽来给每个控件布局）和 `DataBinder`（非常方便地来从一组值控件中取得数据，或给一组值控件赋值）来减少这种复杂性。

```

Class.destroy('App');
Class('App', 'linb.Com', {
  Instance: {
    initComponents: function() {
      // [[code created by jsLinb UI Builder
      var host=this, children=[], append=function(child){ children.push(child);
      append((new linb.DataBinder).setHost(host, "binder").setName("binder"));
      append((new linb.UI.SLabel)
        .setHost(host, "slabel1").setLeft(80).setTop(60).setCaption("Name:"));
      append((new linb.UI.SLabel)
        .setHost(host, "slabel2").setLeft(80).setTop(90).setCaption("Age:").setWidth(44));
      append((new linb.UI.Input).setDataBinder("binder").setDataField("name")
        .setHost(host, "iName").setLeft(130).setTop(60).setValueFormat("[^.*]").setValue("Jack"));
      append((new linb.UI.ComboInput).setDataBinder("binder").setDataField("age")
        .setHost(host, "iAge").setLeft(130).setTop(90).setType("spin").setIncrement(1).setMin(20).set
      etMax(60).setValue("35"));
      append((new linb.UI.SCheckBox).setDataBinder("binder").setDataField("isfull")
        .setHost(host, "cFull").setLeft(130).setTop(130).setCaption("Full time"));
      append((new linb.UI.SButton)
        .setHost(host, "submit").setLeft(130).setTop(170).setCaption("SUBMIT").onClick("_submit
      _onclick"));
      return children;
      // ]]code created by jsLinb UI Builder
    },
    _submit_onclick: function (profile, e, src, value) {
      if(!this.binder.checkValid()){
        linb.alert('One or some invalid fields exists!');
        return;
      }
      linb.alert(_serialize(this.binder.getValue()))
    }
  }
});
(new App).show();
  
```

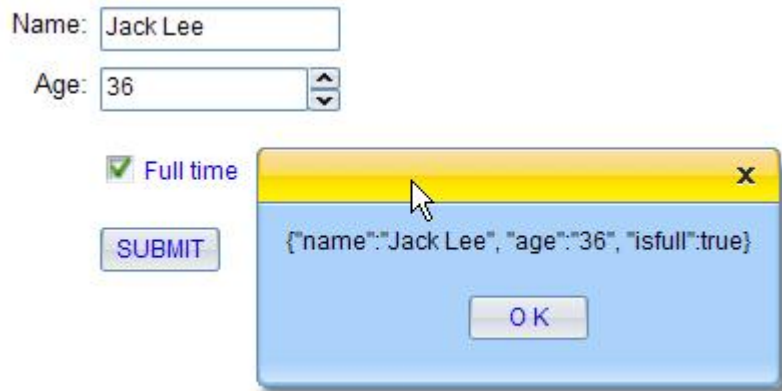
代码都是设计器辅助生成的

加一个 `DataBinder`，并设置 `name` 属性

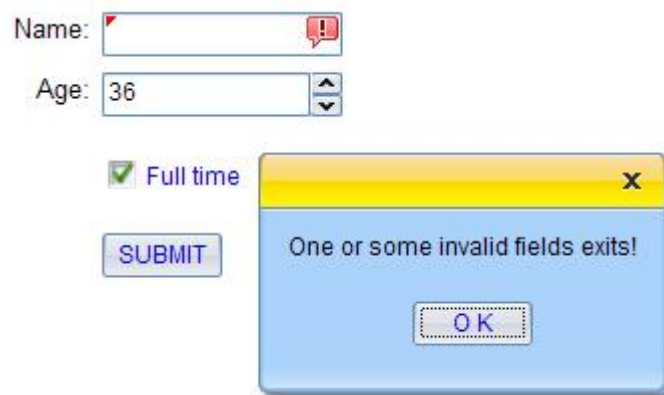
给值控件设置 `dataBinder` 和 `dataField` 属性

表单验证

收集数据



上面的例子中，你可以试着把“Name”值清空，再点击“submit”，就会出现“表单验证”效果。



第五节 定制界面入门

定制界面是一个常有的需求，linb 提供了在不同层次上自定义界面的功能。下面就以下几个层次简单谈一下定制界面的问题，可以作为 linb 自定义界面的入门资源。

1. 只改变一个实例的样式 1

可以用 **setTheme** 方法来改变控件实例的 CSS 样式。

```
linb.CSS.remove("id", "my_css");
linb.CSS.addStyleSheet(".linb-sbutton custom-focus { font-weight: bold; color: #ff0000; }", "my_css");

(new linb.UI.SButton)
.setCaption("Use setCustomClass ")
.setTheme("custom")
.show();
```

自定义这个实例的 theme

theme 关键字在这里

加入一个 CSS 定义，实际应用中应该把自定义的 CSS 都写在一个单独的 css 文件中

Use setCustomStyle

2. 只改变一个实例的样式 2

可以用 `setCustomStyle` 方法来改变控件实例的某个或多个节点的 CSS 样式。

```
(new linb.UI.SButton)
.setCaption("Use setCustomStyle")
.setCustomStyle({
  FOCUS:"font-weight:bold;color:#ff0000;"
})
.show();
```

自定义这个实例 FOCUS 节点的样式

Use setCustomStyle

3. 只改变一个实例的样式 3

可以用 `setCustomStyle` 方法来改变控件实例的某个或多个节点的 CSS 样式。

```
linb.CSS.remove("id","my_css");
linb.CSS.addStyleSheet(".my-class{ font-weight:bold;color:#ff0000;}", "my_css");

(new linb.UI.SButton)
.setCaption("Use setCustomClass ")
.setCustomClass({
  FOCUS:"my-class"
})
.show();
```

加入一个 CSS 定义，实际应用中应该把自定义的 CSS 都写在一个单独的 css 文件中

自定义这个实例 FOCUS 节点的 css 类名

Use setCustomClass

4. 只改变一个实例的样式 4

可以通过 `domId` 来改变控件示例的样式。

```
linb.CSS.remove("id","my_css");
linb.CSS.addStyleSheet("#myctrl1 .linb-sbutton-focus{font-weight:bold;color:#ff0000;}", "my_css");

(new linb.UI.SButton)
.setCaption("Use domId")
.setDomId("myctrl1")
.show();
```

加入一个 CSS 定义，实际应用中应该把自定义的 CSS 都写在一个单独的 css 文件中

为这个控件设置 domId

Use domId

5. 只改变一个实例的样式 5

可以用 `getSubNode` 方法来得到控件里面的 DOM 节点，然后用 `css` 函数来改变样式。

```
(new linb.UI.SButton)
.setCaption("Use getSubNode and css ")
.onRender(function(profile){
    profile.getSubNode('FOCUS').css({
        fontWeight:'bold',
        color:'#ff0000'
    });
})
.show()
```

必须在控件渲染到 dom 之后再调用

Use getSubNode and css

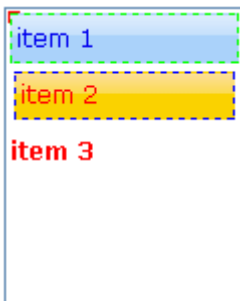
6. 只改变一个实例的样式 6

对于有子项数据的控件（TreeGrid 除外），可以用 `itemStyles` 属性或 `itemClass` 属性来改变控件中单个项的 CSS 样式。

```
linb.CSS.remove("id","my_css");
linb.CSS.addStyleSheet(".my-listitem{ font-weight:bold;color:#ff0000;}", "my_css");

linb.create('List',{items:[{
    id:"item 1",
    itemStyle:"border:dashed 1px #00ff00;margin:2px;"
},{
    id:"item 2",
    itemStyle:"border:dashed 1px #0000ff;margin:4px;"
},{
    id:"item 3",
    itemClass:"my-listitem"
}]}).show()
```

自定义这个实例 FOCUS 节点的样式



7. 改变控件类的样式

```
linb.CSS.remove("id","my_css");
```

```
linb.CSS.addStyleSheet(".linb-sbutton-focus{font-weight:bold;color:#ff0000;}", "my_css");
```

```
(new linb.UI.SButton({position:'relative'})).show();
```

```
(new linb.UI.SButton({position:'relative'})).show();
```

```
(new linb.UI.SButton({position:'relative'})).show();
```

```
(new linb.UI.SButton({position:'relative'})).show();
```

```
(new linb.UI.SButton({position:'relative'})).show();
```

加入一个 CSS 定义，实际应用中应该把自定义的 CSS 都写在一个单独的 css 文件中

每一个实例的样式都改变了

sbutton7

sbutton8

sbutton9

sbutton10

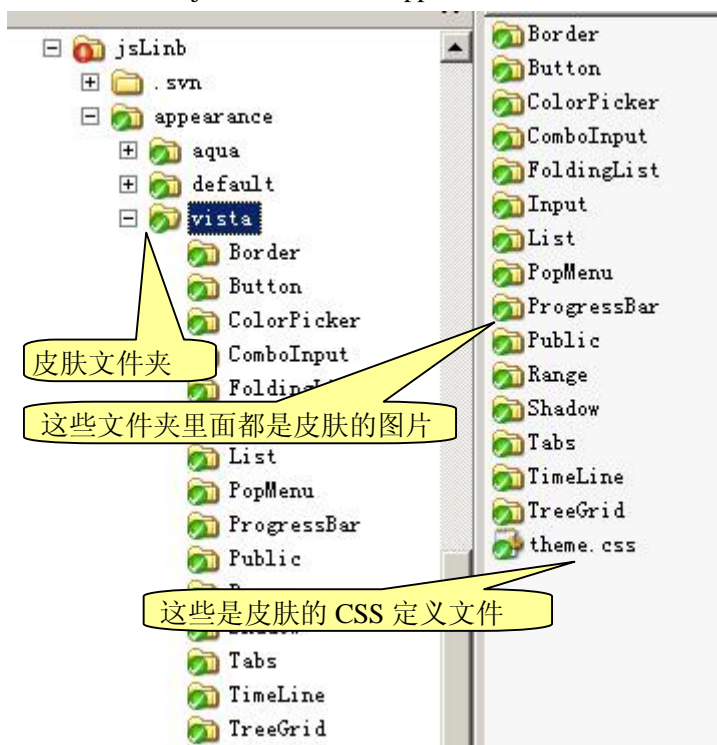
sbutton11

8. 自定义皮肤

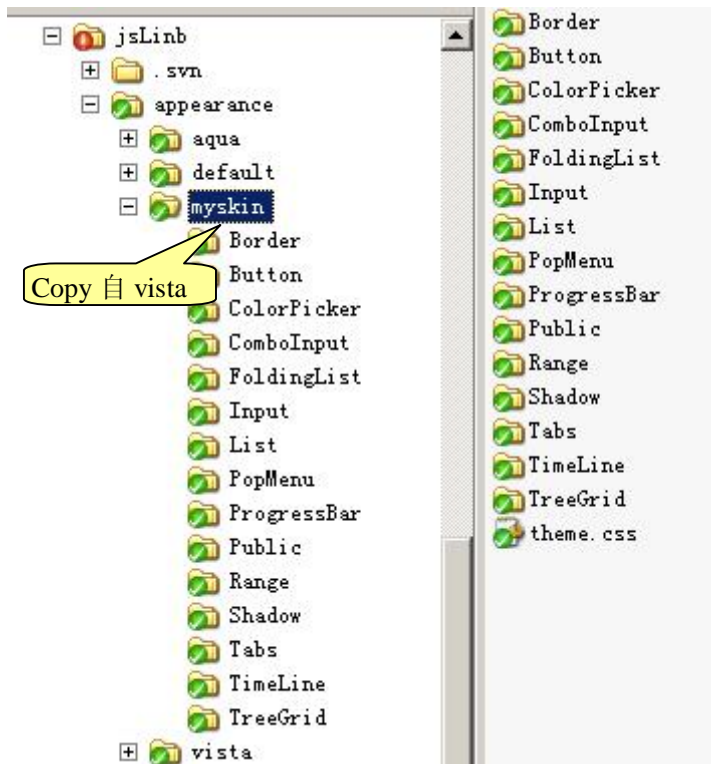
linb 默认有三套皮肤：default、vista 和 aqua。你也可以再给 linb 加上自己自定义的皮肤。方法很简单，只需要两个步骤：

1) 第一步：Copy 一套皮肤到自己的目录

皮肤在 runtime/jsLinb 文件夹下 appearance 目录下。



让我们 copy vista 文件夹，并且命名为 myskin。



2) 第二步：改变图片或 theme.css 文件内容

例如我改变一下 Button 文件夹下的 corner.gif 文件。



现在运行一下以下代码：

```
linb.create('Button').show();
_.asynRun(function(){
    linb.UI.setTheme('myskin')
},2000);
```

可以发现，按钮的边缘部分都已经是新的皮肤的图片了。



接下来要做的很清楚了吧，继续按照你的需求一步一步的改变图片和 theme.css 的内容。一个新的皮肤就会按部就班地来到你的面前。

注：用 **firebug** 或 **ie8/chrome/opera** 的开发工具来查看某个节点上的具体 **css** 设置。

入门篇结语

至此，linb 教程的入门篇内容就告以段落了。希望大家在一边看教程一边运行示例代码的过程中已经对 linb 有了基本的了解。

入门篇的意义应该在于浅显易懂并且容易操作，但由于写作水平有限，教程里面可能会有很多不恰当的文字，希望大家海涵。我当然也希望大家多提出些意见，让我能有机会来把这个教程做的更好。对于本教程每一次较大的改动，我都会用版本号的方式来发布教程，最终的目的是希望这本 linb 教程能够把学习 linb 的程序的入门门槛降到最低。

linb