

Vaadin之书

Vaadin 7 - 第4版

vaadin }>

Marko Grönroos

2014
Vaadin 有限公司

中文翻译：
李颖 (liying.cn.2010@gmail.com)

Vaadin之书: Vaadin 7 - 第4版

Vaadin 有限公司

Marko Grönroos

Vaadin 7 - 第4版 版

Vaadin Framework 7.3.0

Published: 2014-11-19

版权 © 2000-2014 Vaadin 有限公司

摘要

Vaadin 是一种 AJAX web 应用程序开发框架，通过它可以开发出高质量用户界面的应用程序，开发者可以在服务器和客户端两端都使用 Java 语言。Vaadin 框架提供了一系列功能强大、便于使用的 UI 组件，还提供了一个干净的基础框架供开发者创建新的自定义组件。Vaadin 框架关注于软件的易用、复用、易扩展，完全能够胜任大型企业级应用程序的开发工作。

保留所有权利。本文档根据 Creative Commons 协议发布 CC-BY-ND License Version 2.0.

目录

前言	xv
Chapter 1. 简介	21
1.1. 概述	21
1.2. 示例程序一瞥	23
1.3. 对 Eclipse IDE 的支持	24
1.4. Vaadin 的目标与哲学	25
1.5. 背景	25
Chapter 2. 开始使用 Vaadin	27
2.1. 概述	27
2.2. 设置开发环境	28
2.2.1. 安装 Java SDK	29
2.2.2. 安装 Eclipse IDE	30
2.2.3. 安装 Apache Tomcat	31
2.2.4. Firefox 和 Firebug	31
2.3. Vaadin 库概述	32
2.4. 安装 Vaadin Plugin for Eclipse	33
2.4.1. 安装 IvyDE Plugin	33
2.4.2. 安装 Vaadin Plugin	34
2.4.3. 更新插件	36
2.4.4. 更新 Vaadin 库	36
2.5. 使用 Eclipse 创建和运行一个工程	36
2.5.1. 创建工程	37
2.5.2. 浏览一下工程结构	40
2.5.3. 针对 Eclipse 的编码提示	42
2.5.4. 配置和启动 Web 服务器	42
2.5.5. 运行和调试	44
2.6. 通过 Maven 使用 Vaadin	45
2.6.1. 通过命令行使用 Maven	45
2.6.2. 编译和运行应用程序	46
2.6.3. 使用插件和定制 Widget 群	46
2.7. 使用 NetBeans IDE 创建工程	46
2.7.1. 使用 Vaadin Archetype 的 Maven 工程	47
2.8. 使用 IntelliJ IDEA 创建工程	47
2.8.1. 配置应用程序服务器	47
2.8.2. 创建 Vaadin Web 应用程序工程	48
2.8.3. 创建 Maven 工程	52
2.9. Vaadin 安装包	56
2.9.1. 包中的内容	56
2.9.2. 安装库文件	57
2.10. 在 Scala 中使用 Vaadin	57
Chapter 3. 整体架构	59
3.1. 概述	59
3.2. 技术背景	61
3.2.1. HTML 和 JavaScript	61
3.2.2. 使用 CSS 和 Sass 控制样式	62
3.2.3. AJAX	62
3.2.4. Google Web Toolkit	62
3.2.5. Java Servlet	63
3.3. 客户端引擎	63

3.4. 事件和监听器	64
Chapter 4. 编写服务器端 Web 应用程序	67
4.1. 概述	67
4.2. 构建 UI	70
4.2.1. 应用程序架构	71
4.2.2. 组合组件	71
4.2.3. 视图导航	72
4.2.4. 访问 UI, Page, Session, 以及 Service	73
4.3. 使用监听器来处理事件	74
4.3.1. 使用匿名类	74
4.3.2. Java 8 中的事件处理	74
4.3.3. 使用常规类来实现监听器	75
4.3.4. 区分事件的发生源	75
4.4. 图片及其他资源	76
4.4.1. 资源接口和类	76
4.4.2. 文件资源	77
4.4.3. Class Loader 资源	77
4.4.4. Theme 资源	77
4.4.5. 流资源	78
4.5. 错误处理	80
4.5.1. 错误指示器和消息	80
4.5.2. 自定义系统消息	80
4.5.3. 管理未被 catch 的例外	81
4.6. 通知	82
4.6.1. 通知类型	83
4.6.2. 定制通知	84
4.6.3. 使用 CSS 来控制样式	84
4.7. 应用程序的生命周期	85
4.7.1. 发布	85
4.7.2. Vaadin Servlet, Portlet, 和 Service	86
4.7.3. 用户 Session	86
4.7.4. 装载 UI	87
4.7.5. UI 过期	88
4.7.6. 显式地关闭 UI	88
4.7.7. Session 过期	89
4.7.8. 关闭 Session	89
4.8. 发布应用程序	90
4.8.1. 在 Eclipse 中创建发布用的 WAR 文件	90
4.8.2. Web 应用程序内容	91
4.8.3. Web Servlet 类	91
4.8.4. 使用部署描述文件 web.xml	92
4.8.5. Servlet 与 URL 模式的映射	93
4.8.6. Servlet 的其他配置参数	94
4.8.7. 发布配置	96
Chapter 5. UI 组件	97
5.1. 概述	98
5.2. 接口和抽象类	99
5.2.1. Component 组件	99
5.2.2. AbstractComponent	101
5.3. 组件的共通功能	101
5.3.1. 标题	101
5.3.2. 描述信息和提示信息	102

5.3.3. 激活与禁用	103
5.3.4. 图标	104
5.3.5. 语言环境(Locale)	105
5.3.6. 只读	108
5.3.7. 样式名称	109
5.3.8. 可见和隐藏	109
5.3.9. 控制组件的尺寸	110
5.3.10. 管理输入焦点	111
5.4. Field 组件	112
5.4.1. Field 接口	112
5.4.2. 数据绑定和数据转换	113
5.4.3. 处理 Field 值的变更	114
5.4.4. Field 值的缓存	114
5.4.5. Field 值校验	115
5.5. 选择组件	117
5.5.1. 将选择组件绑定到数据	117
5.5.2. 添加新的选择项	118
5.5.3. 项目标题	118
5.5.4. 取得和设置当前选中项目	120
5.5.5. 处理选择项的变化事件	121
5.5.6. 允许添加新项目	121
5.5.7. 复数选择	122
5.5.8. 项目图标	123
5.6. 组件的扩展	123
5.7. Label	123
5.7.1. 文本的宽度与折行	124
5.7.2. 内容模式	125
5.7.3. 用 Label 来控制空白	127
5.7.4. 数据绑定	128
5.7.5. CSS 样式规则	128
5.8. Link	128
5.9. TextField	130
5.9.1. 数据绑定	131
5.9.2. 字符串长度	132
5.9.3. 处理 Null 值	132
5.9.4. 文本变更事件	133
5.9.5. CSS 样式规则	134
5.10. TextArea	134
5.11. PasswordField	136
5.12. RichTextArea	137
5.13. 使用 DateField 输入日期和时间	139
5.13.1. PopupDateField	140
5.13.2. InlineDateField	143
5.13.3. 日期与时间的粒度	145
5.13.4. DateField 的本地化	145
5.14. Button	145
5.15. CheckBox	146
5.16. ComboBox	147
5.16.1. 被过滤的选择项	147
5.17. ListSelect	149
5.18. NativeSelect	150
5.19. OptionGroup	151
5.19.1. 禁用项目	152

5.20. TwinColSelect	154
5.21. Table	155
5.21.1. 在 Table 内选择项目	157
5.21.2. Table 的功能	158
5.21.3. 在 Table 内编辑数据值	162
5.21.4. 列头和列脚	165
5.21.5. 动态生成的列	167
5.21.6. 列的格式控制	170
5.21.7. CSS 样式规则	172
5.22. Tree	174
5.23. MenuBar	175
5.24. Upload	178
5.25. ProgressBar	181
5.26. Slider	184
5.27. Calendar	186
5.27.1. 日期范围与显示模式	187
5.27.2. 日历中的事件	187
5.27.3. 从容器得到事件	189
5.27.4. 实现 Event Provider	191
5.27.5. 控制日历的样式	193
5.27.6. 可见的小时和日	195
5.27.7. 拖放	195
5.27.8. 使用上下文菜单	196
5.27.9. 本地化与格式化	196
5.27.10. 定制日历	197
5.27.11. 日期的前方和后方跳转	198
5.27.12. 处理日期的点击	198
5.27.13. 处理周的点击	199
5.27.14. 处理事件的点击	199
5.27.15. 事件的拖动	200
5.27.16. 处理拖动式选择(Drag Selection)	201
5.27.17. 变更事件的长度	202
5.28. 使用 CustomComponent 创造复合组件	202
5.29. 使用 CustomField 组合 Field 组件	203
5.30. 内嵌资源	203
5.30.1. 内嵌的 Image	204
5.30.2. Adobe Flash 动画	204
5.30.3. BrowserFrame	205
5.30.4. 通用的 Embedded 对象	205
Chapter 6. 布局管理	207
6.1. 概述	208
6.2. UI, Window, 以及 Panel 内容	209
6.3. VerticalLayout 和 HorizontalLayout	210
6.3.1. 在有顺序的布局中控制组件间隔	211
6.3.2. 控制布局内组件的尺寸	211
6.4. GridLayout	215
6.4.1. 调整单元格尺寸	216
6.5. FormLayout	218
6.6. Panel	220
6.6.1. 滚动 Panel 的内容	220
6.7. 子窗口	222
6.7.1. 打开和关闭子窗口	222
6.7.2. 窗口位置	224

6.7.3. 滚动窗口内容	224
6.7.4. 模态子窗口	224
6.8. HorizontalSplitPanel 和 VerticalSplitPanel	225
6.9. TabSheet	227
6.9.1. 添加 Tab	228
6.9.2. Tab 对象	228
6.9.3. Tab 切换事件	229
6.9.4. 允许关闭 Tab, 处理 Tab 的关闭事件	230
6.10. Accordion	231
6.11. AbsoluteLayout	232
6.12. CssLayout	235
6.12.1. CSS 注入	235
6.12.2. 浏览器兼容性	236
6.13. 布局格式控制	237
6.13.1. 布局的尺寸	237
6.13.2. 组件的扩张	238
6.13.3. 布局单元格的对齐	239
6.13.4. 布局单元格的间隔空白	241
6.13.5. 布局的余白	241
6.14. 自定义布局	242
Chapter 7. Themes	245
7.1. 概述	245
7.2. 层级样式表(CSS, Cascading Style Sheets)简介	247
7.2.1. 将 CSS 应用于 HTML	247
7.2.2. 基本的 CSS 规则	247
7.2.3. 使用元素的 class 进行匹配	248
7.2.4. 使用元素的层级关系进行匹配	249
7.2.5. 层级的重要性	250
7.2.6. Vaadin UI 的样式类层级	251
7.2.7. 关于兼容性的注意事项	254
7.3. 优良语法样式表(Sass, Syntactically Awesome Stylesheets)	254
7.3.1. Sass 概述	254
7.3.2. Vaadin 中的 Sass	256
7.3.3. 编译 Sass Theme	256
7.4. Theme 的创建和使用	259
7.4.1. Sass Theme	259
7.4.2. 旧式 CSS Theme	260
7.4.3. 控制标准组件的样式	260
7.4.4. 内建 Theme	260
7.4.5. Add-on 的 Theme	262
7.5. 在 Eclipse 中创建 Theme	262
7.6. Valo Theme	263
7.6.1. 基本使用	264
7.6.2. 共通设定	264
7.6.3. Valo 中的 Mixin 和函数	267
7.6.4. Valo 的字体	268
7.6.5. 组件样式控制	268
7.6.6. Theme 优化	269
7.7. 字体图标	269
7.7.1. 装载图标字体	269
7.7.2. 基本使用	270
7.7.3. 在 HTML 中使用字体图标	270
7.7.4. 在其他文本中使用字体图标	271

7.7.5. 定制字体图标	271
7.8. 自定义字体	273
7.8.1. 装载字体	273
7.8.2. 使用自定义字体	273
7.9. 条件式 Theme	273
Chapter 8. 组件与数据绑定	279
8.1. 概述	279
8.2. 属性(Property)	281
8.2.1. 属性的查看器和编辑器	282
8.2.2. ObjectProperty 实现	283
8.2.3. 在属性类型与表达之间转换	283
8.2.4. 实现 Property 接口	285
8.3. 在项目(Item)中保存属性(Property)	287
8.3.1. PropertysetItem 实现	287
8.3.2. 使用 BeanItem 包装一个 Bean	287
8.4. 通过 Field 与项目的绑定来创建 Form	289
8.4.1. 简易绑定	289
8.4.2. 使用 FieldFactory 创建并绑定 Field	290
8.4.3. 绑定成员 Field	290
8.4.4. Form 的缓冲	291
8.4.5. 将 Field 绑定到 Bean	292
8.4.6. Bean 的校验	292
8.5. 在容器(Container)中保存项目(Item)	294
8.5.1. 容器的基本使用	294
8.5.2. 容器的下级接口	296
8.5.3. IndexedContainer	296
8.5.4. BeanContainer	297
8.5.5. BeanItemContainer	300
8.5.6. 在容器内遍历	301
8.5.7. Filterable 容器	302
Chapter 9. Vaadin SQLContainer	305
9.1. 架构	306
9.2. SQLContainer 入门	306
9.2.1. 创建连接池	306
9.2.2. 创建查询代理 TableQuery	306
9.2.3. 创建容器	307
9.3. 过滤与排序	307
9.3.1. 过滤	307
9.3.2. 排序	308
9.4. 编辑	308
9.4.1. 添加项目	308
9.4.2. 取得数据库生成的row key	308
9.4.3. 对版本控制列的要求	308
9.4.4. 自动 Commit 模式	309
9.4.5. 更新状态	309
9.5. 缓存, 分页和刷新	309
9.5.1. 容器大小	309
9.5.2. 分页长度与缓存大小	310
9.5.3. 刷新容器	310
9.5.4. 缓存 Flush 通知机制	310
9.6. 刷新其他 SQLContainer	311
9.7. 使用任意查询	311

9.8. 未实现的方法	312
9.9. 已知的问题与限制事项	313
Chapter 10. 在 Eclipse 中进行可视化的 UI 设计	315
10.1. 概述	315
10.2. 创建新的复合组件	316
10.3. 使用可视化编辑器	318
10.3.1. 添加新组件	318
10.3.2. 设置组件属性	319
10.3.3. 编辑 AbsoluteLayout	322
10.4. 可视化可编辑组件的结构	323
10.4.1. 子组件的引用	323
10.4.2. 子组件构建方法	324
10.4.3. 构造函数	324
Chapter 11. Web 应用程序开发的高级问题	327
11.1. 管理浏览器窗口	327
11.1.1. 打开弹出式窗口	328
11.1.2. 关闭弹出式窗口	330
11.2. 在 Web 页面中嵌入 UI	330
11.2.1. 在 div 元素内嵌入 UI	331
11.2.2. 嵌入到 iframe 元素中	335
11.2.3. 使用 Vaadin XS Add-on 实现跨站嵌入	337
11.3. Debug 模式和 Debug 窗口	338
11.3.1. 打开 Debug 模式	338
11.3.2. 打开 Debug 窗口	338
11.3.3. Debug 日志	339
11.3.4. 一般信息	340
11.3.5. 查看组件层级关系	340
11.3.6. 通信 Log	342
11.3.7. Debug 模式	342
11.4. 请求处理器(Request Handler)	342
11.5. 快捷键	343
11.5.1. 默认按钮的快捷键	343
11.5.2. 控制 Field 的焦点快捷键	343
11.5.3. 通用的快捷键 Action	344
11.5.4. 对键码(Key Code)和修饰键(Modifier Key)的支持	346
11.6. 打印	347
11.6.1. 打印浏览器窗口	347
11.6.2. 打开打印窗口	347
11.6.3. 打印 PDF	348
11.7. 与 Google App Engine 的集成	349
11.8. 共通的安全问题	350
11.8.1. 处理用户输入内容, 防止跨站脚本攻击	350
11.9. 应用程序内的导航跳转	351
11.9.1. 导航设置	351
11.9.2. 实现 View	352
11.9.3. 处理 URI 片段路径	352
11.10. 应用程序高级架构	355
11.10.1. 分层架构	355
11.10.2. 模型(Model)-视图(View)-展现者(Presenter) 模式	356
11.11. 管理 URI 片段	360
11.11.1. 设置 URI 片段	360
11.11.2. 读取 URI 片段	360

11.11.3. 监听 URI 片段的变更	360
11.11.4. 支持 Web 爬虫	361
11.12. 拖放	362
11.12.1. 处理拖放	362
11.12.2. 拖放项目到 Tree 上	363
11.12.3. 拖放项目到 Table 上	365
11.12.4. 接受拖放	365
11.12.5. 拖动组件	368
11.12.6. 拖动到组件上	369
11.12.7. 从浏览器之外拖放文件	370
11.13. 日志	370
11.14. 与 JavaScript 集成	371
11.14.1. 调用 JavaScript	371
11.14.2. 处理 JavaScript 函数的回调	372
11.15. 访问 Session 全局数据	372
11.15.1. 传递对象引用	374
11.15.2. 覆盖 attach() 方法	374
11.15.3. ThreadLocal 模式	375
11.16. 服务器端 PUSH	376
11.16.1. 安装 PUSH 功能	376
11.16.2. 对一个 UI 允许 PUSH 功能	377
11.16.3. 在其他线程中访问 UI	378
11.16.4. 向其他用户发送广播	379
Chapter 12. 与 Portal 集成	383
12.1. 概述	383
12.2. 在 Eclipse 中创建一个常规的 Portlet 工程	383
12.2.1. 使用 Vaadin Plugin 来创建一个工程	384
12.3. 为 Liferay 开发 Vaadin Portlet	386
12.3.1. 为 Maven 定义 Liferay Profile	386
12.3.2. 使用 Maven 创建 Portlet 工程	388
12.3.3. 在 Liferay IDE 中创建 Portlet	390
12.3.4. 删除 Liferay 自带的 Vaadin	390
12.3.5. 安装 Vaadin 资源	390
12.4. Portlet UI	391
12.5. 部署到 Portal 中	392
12.5.1. Portlet 部署描述符	394
12.5.2. Liferay Portlet 描述符	394
12.5.3. Liferay Display 描述符	395
12.5.4. Liferay Plugin Package Properties 文件	396
12.5.5. 使用单一的 Widget 群	397
12.5.6. 构建 WAR 包文件	397
12.5.7. 部署 WAR 包文件	397
12.6. Portlet Context	397
12.7. Liferay 的 Vaadin IPC add-on	398
12.7.1. 安装 Add-on	399
12.7.2. 基本的通信功能	399
12.7.3. 注意事项	400
12.7.4. 使用 Session 属性进行通信	401
12.7.5. 数据的序列化和编码	402
12.7.6. 与非 Vaadin Portlet 通信	403
Chapter 13. 客户端 Vaadin 开发	405
13.1. 概述	405

13.2. 安装客户端开发环境	406
13.3. 客户端模块描述文件	406
13.3.1. 指定样式表	406
13.3.2. 限定编译目标	406
13.4. 编译客户端模块	407
13.4.1. Vaadin 编译器概述	407
13.4.2. 在 Eclipse 环境中编译	407
13.4.3. 使用 Ant 编译	408
13.4.4. 使用 Maven 编译	408
13.5. 创建自定义 Widget	408
13.5.1. 一个简单的 Widget	408
13.5.2. 使用 Widget	409
13.6. 调试客户端代码	409
13.6.1. 启动开发模式	409
13.6.2. 启动超级开发模式	410
13.6.3. 在 Chrome 内调试 Java 代码	411
Chapter 14. 客户端应用程序	413
14.1. 概述	413
14.2. 客户端模块的 Entry-Point	414
14.2.1. 模块描述文件	415
14.3. 编译和运行客户端应用程序	415
14.4. 装载客户端应用程序	416
Chapter 15. 客户端 Widget	417
15.1. 概述	417
15.2. GWT Widget	418
15.3. Vaadin Widget	418
Chapter 16. 与客户端集成	419
16.1. 概述	419
16.2. 使用 Eclipse 简化工作	422
16.2.1. 创建 Widget	423
16.2.2. 编译 Widget Set	425
16.3. 创建服务器端组件	425
16.3.1. 基本的服务器端组件	425
16.4. 使用连接器实现客户端与服务器端的集成	426
16.4.1. 一个基本的连接器	426
16.4.2. 与服务器端通信	427
16.5. 状态信息共享	427
16.5.1. 在服务器端访问共享的状态信息	427
16.5.2. 在连接器中管理共享的状态信息	427
16.5.3. 使用 @OnStateChange 处理属性状态的变更	428
16.5.4. 将状态属性转发给 Widget	428
16.5.5. 在共享的状态信息中参照组件	429
16.5.6. 共享资源	429
16.6. 客户端与服务器端之间的 RPC 调用	430
16.6.1. 对服务器端的 RPC 调用	430
16.7. 组件与 UI 扩展	431
16.7.1. 服务器端扩展 API	432
16.7.2. 扩展的连接器	433
16.8. Widget 的样式控制	433
16.8.1. 确定 CSS 样式类	434
16.8.2. 默认的样式表文件	434
16.9. 组件容器	435

16.10. 客户端的一些高级问题	435
16.10.1. 客户端处理的各个阶段	435
16.11. 创建 Add-on	435
16.11.1. 在 Eclipse 中导出 Add-on	436
16.11.2. 使用 Ant 构建 Add-on	437
16.12. 从 Vaadin 6 迁移	441
16.12.1. 快速(而且肮脏)地迁移	441
16.13. JavaScript 组件与扩展的集成	441
16.13.1. JavaScript 库示例	441
16.13.2. 供 JavaScript 组件使用的服务器端 API	443
16.13.3. 定义一个 JavaScript 连接器	444
16.13.4. 从 JavaScript 到服务器端的 RPC 调用	444
Chapter 17. 使用 Vaadin Add-on	447
17.1. 概述	447
17.2. 通过 Vaadin Directory 下载 Add-on	448
17.2.1. 使用 Ant 脚本编译 Widget Set	448
17.3. 在 Eclipse 中使用 Ivy 安装 Add-on	448
17.4. 在 Maven 工程中使用 Add-on	450
17.4.1. 添加依赖	450
17.4.2. 编译工程的 Widget Set	451
17.4.3. 启动 Widget Set 编译功能	452
17.5. 问题诊断	453
Chapter 18. Vaadin Charts	455
18.1. 概述	455
18.2. 安装 Vaadin Charts	457
18.3. 基本使用	457
18.3.1. 显示多个数据序列	459
18.3.2. 混合类型图表	460
18.3.3. 图表 Theme	461
18.4. 图表类型	461
18.4.1. 折线图(Line Chart)与曲线图(Spline Chart)	461
18.4.2. 面积图(Area Chart)	462
18.4.3. 柱形图(Column Chart)与条形图(Bar Chart)	462
18.4.4. 错误条(Error Bar)	463
18.4.5. 箱形图(Box Plot Chart)	464
18.4.6. 散点图(Scatter Chart)	465
18.4.7. 气泡图(Bubble Chart)	468
18.4.8. 饼图(Pie Chart)	469
18.4.9. 仪表图(Gauge Chart)	472
18.4.10. 区域范围图(Area Range Chart)与列范围图(Column Range Chart)	473
18.4.11. 极坐标图(Polar Chart), 风玫瑰图(Wind Rose Chart), 与蜘蛛网图 (Spiderweb Chart)	474
18.4.12. 漏斗图(Funnel Chart)	475
18.4.13. 瀑布图(Waterfall Chart)	477
18.5. Chart 配置	479
18.5.1. 绘图选项(Plot Option)	479
18.5.2. 坐标轴	480
18.5.3. 图例	481
18.6. 图表数据	481
18.6.1. List 数据序列	481
18.6.2. 一般数据序列	481
18.6.3. 范围数据序列	483

18.6.4. 容器数据序列	483
18.7. 高级使用	484
18.7.1. 服务器端描绘与导出	484
18.8. 时间线	486
18.8.1. 图像类型	486
18.8.2. 用户操作元素	487
18.8.3. 事件标记	490
18.8.4. 效率	488
18.8.5. 对数据源的要求	489
18.8.6. 事件与监听器	490
18.8.7. 可配置性	490
18.8.8. 本地化	491
18.8.9. 时间线图表的教程	491
Chapter 19. Vaadin JPAContainer	499
19.1. 概述	499
19.2. 安装	502
19.2.1. 下载安装包	502
19.2.2. 安装包的内容	502
19.2.3. 使用 Maven 下载	502
19.2.4. 将库添加到你的工程中	503
19.2.5. 持久化配置	503
19.2.6. 故障诊断	505
19.3. 定义业务数据模型(Domain Model)	505
19.3.1. 持久化元数据	506
19.4. JPAContainer 的基本使用	508
19.4.1. 使用 JPAContainerFactory 创建 JPAContainer	509
19.4.2. 创建和访问实体	510
19.4.3. 嵌套属性	511
19.4.4. 层级数据容器	512
19.5. 实体提供者(Entity Provider)	513
19.5.1. 内建的 Entity Provider	514
19.5.2. 在 JEE6 环境中使用 JNDI Entity Provider	515
19.5.3. EJB 形式的 Entity Provider	515
19.6. 在 JPAContainer 中过滤	516
19.7. 使用 Criteria API 进行查询	517
19.7.1. 对查询进行过滤	517
19.7.2. 兼容性	518
19.8. Form 的自动生成	518
19.8.1. 配置 Field Factory	518
19.8.2. 使用 Field Factory	519
19.8.3. 主-从(Master-Detail)数据编辑器	520
19.9. JPAContainer 与 Hibernate 的结合使用	520
19.9.1. 延迟装载(Lazy loading)	520
19.9.2. 每个请求一个实体管理器(EntityManager-Per-Request)模式	521
19.9.3. Hibernate 与 EclipseLink 中的表连接(Join)对比	522
Chapter 20. 使用 TouchKit 创建移动设备应用程序	523
20.1. 概述	523
20.2. 针对移动设备浏览器应当考虑的问题	525
20.2.1. 移动设备的人机界面	526
20.2.2. 带宽与性能	526
20.2.3. 移动设备的功能特性	526
20.2.4. 兼容性	526

20.3. 安装 Vaadin TouchKit	527
20.3.1. 以 Ivy 依赖项方式安装	527
20.3.2. 声明 Maven 依赖项	527
20.3.3. 使用 Zip 包安装	528
20.4. 导入 Parking 示例程序	528
20.5. 创建新的 TouchKit 工程	528
20.5.1. 使用 Maven Archetype	529
20.5.2. 从新的 Eclipse 工程开始创建 TouchKit 工程	530
20.6. TouchKit 应用程序的组成元素	531
20.6.1. Servlet 类	531
20.6.2. 使用 web.xml 部署描述文件定义 Servlet 和 UI	531
20.6.3. TouchKit 设定	532
20.6.4. UI	533
20.6.5. 移动设备 Widget Set	534
20.6.6. 移动设备 Theme	534
20.6.7. 使用字体图标	535
20.7. 移动设备 UI 组件	536
20.7.1. NavigationView	537
20.7.2. Toolbar	538
20.7.3. NavigationManager	538
20.7.4. NavigationButton	540
20.7.5. Popover	542
20.7.6. SwipeView	545
20.7.7. Switch	546
20.7.8. VerticalComponentGroup	546
20.7.9. HorizontalButtonGroup	548
20.7.10. TabBarView	548
20.7.11. EmailField	550
20.7.12. NumberField	550
20.7.13. UrlField	551
20.8. 移动设备高级特性	551
20.8.1. 提供一个备用(Fallback)UI	551
20.8.2. 地理位置	552
20.8.3. 在本地存储中保存数据	554
20.8.4. 上传内容	555
20.9. 离线模式(Offline Mode)	558
20.9.1. 启用缓存配置(Cache Manifest)	559
20.9.2. 启用离线模式	559
20.9.3. 离线模式下的 UI	559
20.9.4. 向服务器发送数据	559
20.9.5. 离线模式下的 Theme	560
20.10. 构建最优化的 Widget Set	560
20.10.1. 生成 Widget Map	560
20.10.2. 定义 Widget 的装载方式	561
20.10.3. 应用自定义的 Widget Map Generator	561
20.10.4. 部署	561
20.11. 在移动设备上测试和调试	562
20.11.1. 调试	562
Chapter 21. Vaadin TestBench	563
21.1. 概述	563
21.2. 快速入门	567
21.2.1. 安装 License Key	567
21.2.2. Eclipse 环境下的快速入门	569

21.2.3. 使用 Maven 的快速入门	570
21.3. 安装 Vaadin TestBench	570
21.3.1. 测试程序开发环境	570
21.3.2. 分布式测试环境	571
21.3.3. 安装包的内容	572
21.3.4. TestBench 示例程序	572
21.3.5. 安装浏览器驱动程序	573
21.3.6. 测试节点的配置	574
21.4. 开发 JUnit 测试程序	575
21.4.1. 测试用例(Test Case) 的基本结构	575
21.4.2. 在 Eclipse 中运行 JUnit 测试程序	578
21.5. 创建 Test Case	579
21.5.1. 测试的启动	579
21.5.2. 测试用例(Test Case) 的基本结构	579
21.5.3. Web Driver 的创建和关闭	580
21.6. 查询页面元素	582
21.6.1. 使用 Debug 窗口生成查询	582
21.6.2. 使用组件类型(\$)查询页面元素	583
21.6.3. 非递归的组件查询 (\$\$)	583
21.6.4. 页面元素类	583
21.6.5. ElementQuery 对象	583
21.6.6. 查询结束符	583
21.7. 元素选择器	584
21.7.1. 通过 ID 查找	584
21.7.2. 通过 CSS 类查找	584
21.8. 测试中的一些特殊问题	585
21.8.1. 等待 Vaadin 处理完毕	585
21.8.2. 测试提示信息(Tooltip)	585
21.8.3. 滚动	586
21.8.4. 测试通知信息	586
21.8.5. 测试上下文菜单	586
21.8.6. 测量测试程序的执行时间	587
21.9. 创建可维护的测试程序	589
21.9.1. 增强选择器的健壮性	589
21.9.2. 页面对象模式(Page Object Pattern)	590
21.10. 屏幕截图的取得和比较	592
21.10.1. 屏幕截图参数	592
21.10.2. 测试失败时取得屏幕截图	593
21.10.3. 取得用于比较的屏幕截图	593
21.10.4. 处理屏幕截图时的一些实际经验	595
21.10.5. 已知的兼容性问题	595
21.11. 运行测试	596
21.11.1. 使用 Ant 运行测试	596
21.11.2. 使用 Maven 运行测试	597
21.12. 在分布式环境中运行测试	598
21.12.1. 远程运行测试	598
21.12.2. 启动 Hub	599
21.12.3. 测试节点的服务配置	600
21.12.4. 启动一个测试网格节点	602
21.12.5. 移动设备测试	602
21.13. 测试程序的并行执行	603
21.13.1. 本地并行测试	603
21.13.2. 在网格环境中使用多个浏览器执行测试程序	603

21.14. 无头(Headless)测试	604
21.14.1. 运行无头测试所需要的基本设置	604
21.14.2. 在分布式环境中运行无头测试	604
21.15. 行为驱动开发	605
21.16. 已知的问题	606
21.16.1. 在 Mac OS X 上运行 Firefox 测试程序	606
索引	607

前言

本书简要介绍 Vaadin Framework，并讨论使用 Vaadin 进行应用程序开发过程中可能遇到的一些重要问题。关于 Vaadin 中各种类、接口、方法的详细文档，请参照 Vaadin API Reference。

本书从它的第一个版本完成以来，内容已经大大增加，因此已经变得太厚，很难放进口袋里了。为了在本书的打印版中包含所有内容，现在我们将本书分成 2 卷，其中，入门学习 Vaadin 时所需要了解的内容将被包含在第 1 卷内。

本书的这个版本主要以 2014 年夏发布的 Vaadin 7.3 版为讨论对象。Vaadin 7.3 版最值得注意的特性是它的新的、高度可定制的 Valo theme。

除了 Framework 核心部分的变化之外，本书的这个版本还将介绍 TestBench 4 插件和 TouchKit 4 插件，这些插件在本书编写时还未正式发布。本书的相关章节是基于这些插件的预发布版编写的，因此，最终发布版可能会包含一些变化。

本书的编写还在随时进行之中，而且由于 Vaadin 本身还在快速开发中，因此 Vaadin 的某些功能可能在本书中没有介绍到。
关于本书的最新版本，请访问在线的最新版本：
<http://vaadin.com/book>. 在这个页面你还能找到本书的 PDF 和 EPUB 格式版本。你可能会发觉，相比纸质印刷版本，电子版要更易于检索一些。本书的索引部分还不完善，将来会补充完整。本书的 Web 版还包含了一些额外的技术性内容，比如示例代码，以及实际开发中你可能需要的其他技术细节。纸质印刷版本对这些内容进行了删节，目的是更加偏重于创造一份介绍 Vaadin 的小教材，并且控制其篇幅不要太多。

此外，Vaadin 7 的很多功能都已提供演示教程，在 Vaadin Wiki 可以找到这些教程：
<https://vaadin.com/wiki/-/wiki/Main/Vaadin+7>.

本书的目标读者是谁？

本书面向软件开发人员。如果你正在使用，或考虑使用 Vaadin 来开发 Web 应用程序，你就是本书合适的读者。

本书假定读者已有 Java 程序开发经验，即使没有 Java 开发经验，使用 Vaadin 开发程序也与使用其他 UI 框架一样简单。读者不必具备 AJAX 知识，因为在 Vaadin 中，已经很好地隐藏了与 AJAX 相关的实现细节。

你也许曾经使用过针对桌面应用程序的某种 UI 开发框架，比如 Java 的 AWT, Swing, SWT，或 C++ 的 QT。这些知识有助于你理解 Vaadin 的基本理念，如事件驱动开发模型，以及 UI 框架的其他共通概念，但没有这类知识也并不妨碍你阅读本书。

如果你的团队中没有专职的美工设计人员，那么你需要具备一些基本的 HTML, CSS 知识，这将有助于为你的应用程序开发表现层 theme。本书提供了对 CSS 的简要介绍。如果你需要开发新的客户端组件的话，Google Web Toolkit (GWT) 的相关知识将会很有帮助。

本书的组织

《Vaadin 之书》将向读者介绍 Vaadin 是什么，以及开发者如何运用它来开发 Web 应用程序。

第 1 卷：

第 1 章 简介

本章简要介绍以下内容：1, Vaadin 开发的应用程序的基本架构，2, 框架背后隐含的核心设计思路，3, Vaadin 的一些历史背景。

第 2 章 开始使用 Vaadin

本章介绍以下内容: 1, 如何安装 Vaadin 及其他相关工具, 如 Vaadin Plugin for Eclipse, 2, 如何运行和调试示例程序, 3, 如何在 Eclipse IDE 中创建你自己的应用程序 project .

第 3 章 整体架构

本章介绍 Vaadin 的架构, 它使用到的主要技术, 如 AJAX, Google Web Toolkit, 及事件驱动开发.

第 4 章 编写服务器端 Web 应用程序

本章介绍开发 Vaadin 应用程序需要的基本知识, 如窗口管理, 应用程序的生命周期, 如何将应用程序发布到 Servlet 容器中, 以及如何处理事件, 如何处理错误, 如何管理资源.

第 5 章 UI 组件

本章介绍 Vaadin 的(layout 之外的)所有 UI 组件的基本使用方法, 讲解各组件的主要功能特性. 还给出了每个组件的使用示例, 以及用于控制外观样式的 CSS/Sass 示例.

第 6 章 布局管理

本章介绍布局(Layout)组件, 与桌面应用程序的 UI 开发框架一样, Vaadin 也使用布局组件来管理UI的布局方式.

第 2 卷:

第 7 章 Themes

本章介绍层叠样式表(CSS, Cascading Style Sheets) 和 Sass, 并介绍如何使用它们来为你的应用程序创建自定义的 theme 外观.

第 8 章 组件与数据绑定

本章介绍 Vaadin 内建的数据模型, 包括属性(property), 项目(item), 和容器(container).

第 9 章 Vaadin SQLContainer

本章介绍 SQLContainer, 可用于将 Vaadin 组件绑定到 SQL 查询.

第 10 章 在 Eclipse 中进行可视化的 UI 设计

本章介绍如何在 Eclipse 中使用 Vaadin Plugin 的可视化编辑器功能.

第 11 章 Web 应用程序开发的高级问题

本章讨论应用程序开发中的一些共通问题, 比如在浏览器内打开新窗口, 将应用程序嵌入到通常的 Web 页面中成为页面的一部分, 资源的低阶管理功能, 使用快捷键, 调试, 等等等等.

第 12 章 与 Portal 集成

本章介绍如何使用 Vaadin 开发 Portlet 式应用程序, Portlet 可发布到支持 Java Portlet API 2.0 (JSR-286) 标准的任何 Portal 服务器中. 本章还介绍对 Liferay Portal 服务器的支持, 以及控制面板, IPC, WSRP 插件(add-on).

第 13 章 客户端 Vaadin 开发

本章介绍如何开发客户端应用程序及Widget, 包括安装, 编译, 及调试等内容.

第 14 章 客户端应用程序

本章介绍如何开发客户端应用程序, 以及如何将客户端应用程序与后端服务集成在一起.

第 15 章 客户端 Widget

本章介绍客户端开发中可以用到的内建 Widget(即客户端组件). 内建 Widget 包括 Google Web Toolkit 的 Widget 和 Vaadin Widget.

第 16 章 与客户端集成

本章介绍为了创建新的服务端组件, 如何将客户端 Widget 与对应的服务器端逻辑集成起来. 本章也包括一些 JavaScript 组件的相关介绍.

第 17 章 使用 Vaadin Add-on

本章介绍如何通过 Vaadin Directory 下载和安装 add-on 组件.

第 18 章 Vaadin Charts

本章介绍如何使用 Vaadin Charts add-on 组件创建各种图表. 这个 add-on 包括 Chart 组件和 Timeline 组件.

第 19 章 Vaadin JPAContainer

本章介绍如何使用 JPAContainer add-on, 使用这个 add-on 可通过 Java Persistence API (JPA) 将 Vaadin 组件直接绑定到关系型(或非关系型)数据库.

第 20 章 使用 TouchKit 创建移动设备应用程序

本章介绍如何使用 Vaadin TouchKit add-on 来开发移动应用程序.

第 21 章 Vaadin TestBench

本章介绍如何使用 Vaadin TestBench 工具来对 Vaadin 应用程序进行自动化 UI 回归测试

补充资料

Vaadin 网站提供了大量资料, 可以帮助理解 Vaadin 是什么, 你可以通过 Vaadin 做什么, 以及如何去做.

示例应用程序

Vaadin 最重要的示例程序是 Sampler, 它展示了全部基本组件的使用方法和功能. 你可以在线运行 Sampler, 地址是 <http://demo.vaadin.com/>, 也可以以 WAR 方式下载, 地址是 Vaadin download page.

本书中出现的示例代码, 以及其他更多示例, 都可以在以下网站得到:
<http://demo.vaadin.com/book-examples-vadin7/book/>.

架构图

2页的 Vaadin 概要架构图, 描述各种 UI 类/接口、数据绑定类/接口之间的相互关系. 可在以下地址下载 <http://vaadin.com/book>.

Refcard

6页的 DZone Refcard, 简单介绍了如何使用 Vaadin 开发应用程序. 其中包含一些图表, 描述各种 UI 及数据绑定类/接口. 详情请参照以下页面 <https://vaadin.com/refcard>.

教程程序: 地址簿

地址簿是一个教程中附带的示例程序, 这个教程一步一步演示了如何使用 Vaadin 开发一个真实的 Web 应用程序. 教程可在 Vaadin 网站上找到.

开发者网站

Vaadin 的开发网站在 <http://dev.vaadin.com/>, 这个网站提供了许多在线资源, 比如 Bug 追踪系统, 供开发者使用的 wiki 系统, 源代码版本库, Vaadin 相关活动的时间线, 开发里程碑, 等等等等.

wiki 为开发者提供了很多指导性信息, 尤其是那些希望从版本库中取得源代码然后自行编译 Vaadin 的开发者, 更应该仔细阅读 Vaadin 的 wiki. 此外还包括一些技术文章, 介

绍 Vaadin 应用程序如何与其他系统集成 (如 JSP, Maven, Spring, Hibernate, Portal).
wiki 还对一些常见问题(FAQ) 提供了解答.

在线文档

你可以在线阅读本书, 地址是 <http://vaadin.com/book>. 此外, 还有很多其他资料, 如各种 HOWTO 类的技术文章, 常见问题(FAQ)的解答, 以及其他各种文档, 都可以在以下网站得到: Vaadin web-site.

技术支持

遇到问题了吗? 不用害怕, Vaadin Framework 开发者社区和 Vaadin 公司都可以为你提供帮助.

开发者社区论坛

在 Vaadin 开发者论坛可以找到 Vaadin 的使用者和开发者, 论坛地址是:
<http://vaadin.com/forum>. 关于你遇到的问题, 希望增加的功能等等, 都可以在论坛中与大家一起讨论. 你想要询问的问题, 可能已经在旧帖子中讨论过了, 所以在你发问之前, 请务必先在旧帖中搜索一下.

报告 Bug

如果你在 Vaadin, 或示例程序, 或文档发现了可能是bug 的问题, 请向Vaadin 开发者报告, 方法是在开发者网站上提交一个 bug 票, 开发者网站地址是: <http://dev.vaadin.com/>. 提交新的 bug 票前, 你应该先检索既有的 bug 票, 以免重复. 如果希望增加某种新功能, 或者改善某个已有的功能, 同样可以在开发者网站上起票.

商业支持

Vaadin 公司对 Vaadin Framework 及相关产品提供完全的、商业化的支持和培训服务.
详情请查看 <http://vaadin.com/pro>.

关于本书作者

Marko Grönroos 是一位专业作家, 也是一位软件开发者, 供职于芬兰 Turku 市的 Vaadin 公司. 他于1994年开始从事 Web 应用程序的开发工作, 还参与了C, C++, Java 语言的几种应用程序框架的开发工作. 他活跃于很多开源软件项目中, 拥有 Turku 大学计算机科学专业的硕士学位.

致谢

本书的大部分内容都是在 Vaadin 公司与 Vaadin 开发团队共同工作的产物 . Joonas Lehtinen, Vaadin 公司 CEO, 编写了本书最初的大纲, 这个大纲后来成为了本书第1和第2章的基础. 此后, Marko Grönroos成为本书主要的作者和编辑者. Vaadin 开发团队在本书编写过程中贡献了很多段落的内容, 回答了无数技术问题, 审阅了手册, 并修正了很多错误.

参与编写本书的有贡献人员 (按大致的时间顺序):

Joonas Lehtinen
Jani Laakso
Marko Grönroos
Jouni Koivuvirta
Matti Tahvonen
Artur Signell
Marc Englund
Henri Sara
Jonatan Kronqvist
Mikael Granqvist (TestBench)
Teppo Kurki (SQLContainer)
Tomi Virtanen (Calendar)

Risto Yrjänä (Calendar)
John Ahlroos (Timeline)
Petter Holmström (JPAContainer)
Leif Åstrand

关于 Vaadin 有限公司

Vaadin 有限公司是一家芬兰软件公司, 专长是丰富性网络应用程序(RIA, Rich Internet Applications)的设计和开发. 该公司为客户提供软件项目的策划、实现以及后续支持服务, 同时也承接软件的转包开发. Vaadin Framework (前身叫做 IT Mill Toolkit), 是该公司的一款旗舰开源产品, 该公司也提供 Vaadin Framework 相关的商业性开发或支持服务.

1.1. 概述	21
1.2. 示例程序一瞥	23
1.3. 对 Eclipse IDE 的支持	24
1.4. Vaadin 的目标与哲学	25
1.5. 背景	25

本章将简要介绍使用Vaadin的软件开发方法。除此之外还将介绍一些Vaadin的发展历史，以及它背后的设计理念。

1.1. 概述

Vaadin 是一种 Java Web 应用程序的开发框架，其设计目标是便利地创建和维护高质量的 Web UI 应用程序。Vaadin 支持两种不同的开发模式：服务器端开发和客户端开发。服务器端开发方式是这二者中更为强大的一种。它能帮助开发者忘记 Web 程序的各种实现细节，使得 Web 应用程序的开发变得就象过去使用便利的 Java 开发工具（如 AWT, Swing, SWT）来开发桌面应用程序一样，甚至更简单。

从耗费大把时间学习 Web 新技术的角度来看，传统的 Web 程序开发方式也许是一种有趣的方法，但你也许更希望提高生产效率，把精力更多地集中到应用程序业务逻辑上。服务器端 Vaadin 开发框架将会代替你管理浏览器内的 UI 组件、帮助你维护浏览器与服务器之间的 AJAX 通信。使用 Vaadin 方案，你就不再需要学习和直接处理那些浏览器端的技术细节，比如 HTML, JavaScript 等。

图 1.1. Vaadin 应用程序架构

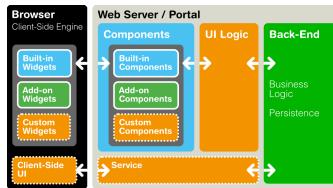


图 1.1 “Vaadin 应用程序架构”描述了使用 Vaadin 开发的 Web 应用程序的基本架构。服务器端应用程序的架构包括服务器端框架，和客户端引擎。客户端引擎以 JavaScript 代码的形式在浏览器内运行，它负责呈现 UI，并把用户操作发送到服务器端。应用程序的 UI 逻辑则以 Java Servlet 的形式在 Java 应用程序服务器内运行。

由于客户端引擎以 JavaScript 形式运行在浏览器内，所以 Vaadin 开发的应用程序在运行时不需要额外的浏览器插件(plugin)的支持。因此，与那些基于 Flash, Java Applets, 或其他各种浏览器插件(plugin)的开发框架相比，Vaadin 更具有优势。Vaadin 底层依赖于 Google Web Toolkit 的支持，实现了跨浏览器能力，因此开发者再也不必担心兼容多种浏览器的问题。

由于 HTML, JavaScript, 以及其他浏览器相关技术对于应用程序逻辑是隐藏的，你可以把 Web 浏览器想象为一个瘦的客户端平台。这个瘦客户端将 UI 展现给用户，又将用户的交互行为发送到服务器端。UI 的控制逻辑与业务逻辑共同运行在基于 Java 的 Web 服务器上。与这种模式不同，传统的客户端/服务器架构存在一个专门的客户端应用程序部分，其中需要包含大量的客户端/服务器双向通讯，而这种通讯又往往是与具体的应用程序高度相关的。我们的方案从逻辑上删去了这个 UI 层，因此变得非常高效。

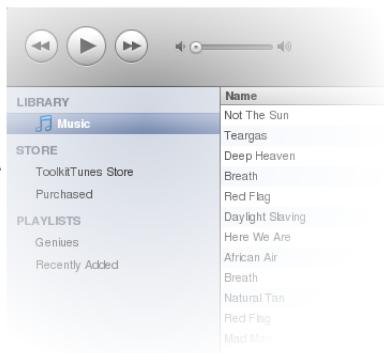
在服务器端开发模型的背后，Vaadin 灵活运用了 AJAX 技术(*Asynchronous JavaScript and XML*，详情参见 第 3.2.3 节 “AJAX”)，AJAX 技术的运用使得 Vaadin 可以创建出功能强大的丰富性网络应用程序 (Rich Internet Application, RIA)，而且这些 RIA 程序的响应速度、易交互性可以达到与桌面应用程序相同的程度。

除了服务器端的 Java 应用程序开发外，你也可以用 Java 语言编写新的 widget 来进行客户端开发，Vaadin 甚至还可以用于开发纯的客户端应用程序，这类应用程序可以在浏览器内独立运行，而不必与服务器交互。Vaadin 的客户端开发框架包括了 Google Web Toolkit (GWT)，GWT 提供了一个编译器，可将 Java 程序编译为 JavaScript 代码，然后在浏览器内运行。Vaadin 的客户端开发框架还包括功能完整的 UI 组件。无论是在客户端还是在服务器端，Vaadin 都使用纯 Java 进行开发，而不必引入其他语言。

Vaadin 服务器端应用程序的 UI 部分由客户端引擎负责呈现在浏览器中。客户端与服务器端的一切通信都被妥善的屏蔽起来。Vaadin 被设计为具有很高的可扩展性，所以除了 Vaadin 本身提供的组件之外，你还可以非常方便地使用第3方 widget。实际上，在 Vaadin Directory 中你可以找到数百个插件(Add-on)。

Vaadin 框架将 UI 组件和它的具体呈现非常清楚地分离为不同的部分，因此允许你分别开发这两部分。我们的方案是使用 theme，theme 使用 CSS 和 HTML 页面模板(可选)来控制 UI 组件的具体呈现。Vaadin 提供了非常完善的默认 theme，所以你通常并不需要做太多的定制，但只要你需要，你总是可以自由地定制 UI 组件的外观。关于 theme 的详细信息，请参见 第 7 章 Themes。

关于 Vaadin 的基本架构和主要功能，我们暂时只介绍到这里。更多详细内容请阅读 第 3 章 整体架构，或者直接阅读更加贴近实战的 第 4 章 编写服务器端 Web 应用程序。



1.2. 示例程序一瞥

下面我们遵照软件业的悠久传统，在学习一个新的开发框架时，先来写写著名的 "Hello World!"。首先，我们使用最基本的服务器端 API。

```
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

@Title("My UI")
@Theme("valo")
public class HelloWorld extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Create the content root layout for the UI
        VerticalLayout content = new VerticalLayout();
        setContent(content);

        // Display the greeting
        content.addComponent(new Label("Hello World!"));

        // Have a clickable button
        content.addComponent(new Button("Push Me!",
            new ClickListener() {
                @Override
                public void buttonClick(ClickEvent e) {
                    Notification.show("Pushed!");
                }
            }));
    }
}
```

一个 Vaadin 应用程序包含一个或多个 **UI**, **UI** 由 **com.vaadin.ui.UI** 类继承而来。一个 **UI** 就是 Vaadin 应用程序运行时所在的 Web 页面的一部分。在同一页面中，一个应用程序也可以包含多个 **UI**，在 portal 中更是如此，在浏览器的不同窗口或不同 tab 中当然也是如此。一个 **UI** 会关联到一个用户 session，而 session 会为每一个使用这个应用程序的用户单独创建。在我们的 Hello World **UI** 中，当用户访问页面时会发生对应用程序的初次访问，此时 session 就被创建出来，**init()** 方法就在这个时候被调用，目前我们只需要知道这些就够了。

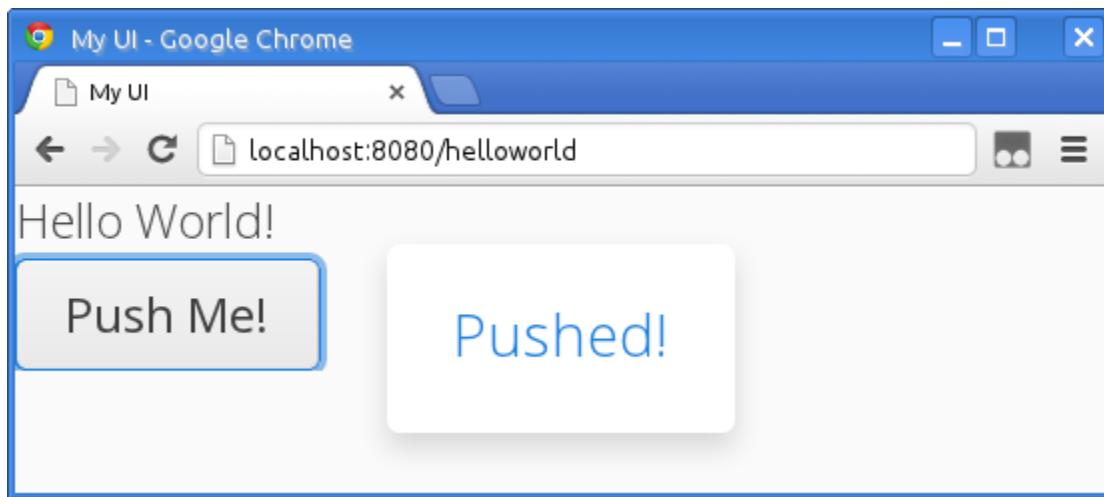
在上面的示例程序中，页面的标题(就是显示在浏览器窗口或tab上的标题部分)，是通过 Java 标注(annotation)方式来定义的。示例程序使用一个布局(layout)组件作为 **UI** 内容的最顶级，这也是大多数 Vaadin 应用程序的通常做法，因为 **UI** 通常会包含很多个组件，需要靠布局(layout)组件来管理它们的位置。然后示例程序创建了一个 **Label** 类型的 **UI** 组件，它用来显示简单的文字，然后将文字内容设置为 "Hello World!"。之后，将 **Label** 添加到布局(layout)组件中。

上面的示例还演示了如何创建一个按钮，以及如何处理按钮的点击事件。关于事件处理，基本原理请参见第 3.4 节“事件和监听器”，具体的应用实例请参见第 4.3 节“使用监听器来处理事件”。除了通常的添加监听器的方式之外，在 Java 8 中，你还可以使用 lambda 表达式来处理事件，这种方式可以大大地简化事件处理代码。

```
content.addComponent(new Button("Push Me!",
    event -> Notification.show("Pushed!")));
```

通过浏览器访问 Hello World 程序时的运行结果，参见图 1.2 “Hello World 应用程序”。

图 1.2. Hello World 应用程序



要运行一个程序，你需要将它打包为 Web 应用程序 WAR 包，然后部署到服务器上，详情参见第 4.8 节“发布应用程序”。在开发阶段，你可以使用 IDE，将应用程序发布到 IDE 内集成的应用程序服务器上。

如果要开发一个纯客户端应用程序，你可以非常简单地输出 Hello World 文字，而且开发语言仍然使用 Java，如下：

```
public class HelloWorld implements EntryPoint {
    @Override
    public void onModuleLoad() {
        RootPanel.get().add(new Label("Hello, world!"));
    }
}
```

这个例子中我们没有设置页面标题，因为标题通常由上述代码执行时所在的 HTML 页面来定义。这个应用程序将被 Vaadin 客户端编译器（或者 GWT 编译器）编译为 JavaScript。这种方法通常用来开发客户端 widget，然后供服务器端 Vaadin 应用程序使用。关于客户端开发的详情，参见第 13 章客户端 Vaadin 开发。

1.3. 对 Eclipse IDE 的支持

Vaadin 并不局限于某一种特定的 IDE，甚至在没有 IDE 的情况下也能很容易地使用 Vaadin，但我们为 Eclipse IDE 提供了特别的支持，因为 Eclipse IDE 目前已经是 Java 开发中最常用的开发环境。对 Eclipse IDE 的支持是通过 Vaadin Plugin 实现的，它可以帮助你：

- 创建新的 Vaadin 工程
- 创建自定义的 theme
- 创建自定义的 widget
- 通过可视化的编辑器创建复合组件
- 便利地升级到 Vaadin 库的新版本

要在开发环境中安装 Vaadin，推荐的方法是使用 Vaadin Plugin for Eclipse。如果不使用 Vaadin Plugin for Eclipse，你也可以手动下载包含所有 JAR 文件的安装包，或者也可以在你的 Maven 工程中加入对 Vaadin 的依赖。

Eclipse plugin 的安装或升级方法，参见 第 2.4 节“安装 Vaadin Plugin for Eclipse”，使用这个插件来创建新的 Vaadin 工程，参见 第 2.5.1 节“创建工程”。关于这个插件各种功能的使用方法，参见 第 7.5 节“在 Eclipse 中创建 Theme”，第 16.2 节“使用 Eclipse 简化工作”，以及 第 10 章 在 Eclipse 中进行可视化的 UI 设计。

1.4. Vaadin 的目标与哲学

简单来说，Vaadin 的野心是成为创建商业化应用程序 Web UI 的最佳工具。Vaadin 易于学习，它既适合于入门级程序员，也适合于高级程序员、UI 可用性专家和图像设计师。

在设计 Vaadin 时，我们遵循以下几条设计哲学。

为正确的目标提供正确的工具

我们的理想很高远，所以目标就应该非常集中。Vaadin 的设计目标是用于开发 Web 应用程序。Vaadin 的目标不是用来创建 Web 站点，也不是用来创建广告演示程序。对于这些用途，你可能会发现更适合使用其他工具，比如 JSP/JSF 或 Flash。

简易性与可维护性

我们希望达成的目标是健壮性、简易性，和可维护性。为了这个目的，就需要遵从 UI 框架中的最佳实践经验，要保证具体的实现代码忠实体现其目标的最佳方案，不能过于混乱或庞大。

XML 不是用来编程的

Web 页面生来就是以文档为中心的，而且其 UI 严重依赖于声明式的表现方法。Vaadin 框架将程序员从这些限制中解放出来。Vaadin 框架以编程的方式创建 UI，这比使用传统的声明式模板来定义 UI 的方式要自然得多，传统方式对于复杂的、动态的用户交互性 UI 来说，缺乏足够的灵活性。

工具不应对你的工作造成限制

当你使用 Vaadin 框架时，对你能做什么不应造成任何限制：如果出于某些理由，UI 组件不能达到你的目的，框架应该支持你很便利的增加新组件到你的应用程序中。如果你需要自行创建新组件，Vaadin 框架扮演的角色将十分关键：它帮助你便利地创建可复用、易维护的新组件。

1.5. 背景

Vaadin 框架不是一夜之间出现的。在 Web 页面刚刚出现的时代，就有开发者开始开发 Web UI 了。2000 年，有一群开发者聚集在一起创建了 IT Mill 公司。这个团队十分渴望开发出一种新的编程方式，能够用真正的编程语言，为真正的应用程序开发出真正的 UI。

这个开发库最初命名为 Millstone Library。它的第一个版本被应用到一个大型生产性应用程序中，这个程序是 IT Mill 公司为一家国际制药公司设计和开发的。IT Mill 公司在 2001 年将这个应用程序开发完成，而且它直到现在还在使用中。此后，IT Mill 公司又使用这个库开发了几十个大型商业应用程序，在这些应用程序的开发过程中，Millstone Library 证明了它简单快速地解决复杂问题的能力。

Millstone Library 的下一代, IT Mill Toolkit Release 4, 发布于 2006 年. 它引入了基于 AJAX 技术的全新的表现引擎. 这就使得开发者可以开发出 AJAX 应用程序, 而完全不必关注客户端与服务器端的通信细节.

第 5 版, 走向开源

IT Mill Toolkit 5, 初次发布于 2007 年末, 在 AJAX 方面又迈进了一大步. UI 组件的客户端描绘改用 GWT(Google Web Toolkit) 完全重写过了.

IT Mill Toolkit 5 在服务端 API 和功能性两方面都引入了很多重大改进. 使用 GWT 改写客户端引擎使得客户端和服务端开发都可以统一使用 Java 语言. 从 JavaScript 到 GWT 的转换, 使得自定义组件的开发和集成, 以及对既有组件的定制化都变得比从前更加简单, 也使得开发者可以非常简单地集成既有的 GWT 组件. 在客户端采用 GWT 并没有导致服务器端 API 的变化, 因为 GWT 是浏览器端的技术, 它很好地隐藏在 API 之后. 另外, 在 IT Mill Toolkit 5 中, Theme 也经过了完全的修订.

Release 5 采用无任何限制的开源许可协议 Apache License 2 发布, 因此使得用户数量更快增长, 开发者社区也因此逐渐产生了.

Vaadin Release 6 的诞生

2009 年春, IT Mill Toolkit 改名为 Vaadin Framework, 简称 Vaadin. 之后不久 IT Mill 公司也相应的改名为 Vaadin 有限公司. Vaadin 是芬兰语, 指一种半驯化的成年雌性驯鹿.

在 Vaadin 6 时代, 这个框架的用户数量大大扩展了. 随着 Vaadin 6 的发布, 还发布了 Vaadin Plugin for Eclipse, 可以帮助开发者创建 Vaadin 工程. 2010 年早期, 随着 Vaadin Directory 的引入, 可用 UI 组件的数量几乎在一夜之间增长了数倍, 带动了 Vaadin 更快的发展. 早期的很多实验性组件, 逐渐发展成熟, 现在已被成千上万的开发者采用了. 在 2013 年, 我们看到 Vaadin 周边生态系统发生了巨大的增长, 用户社区的规模, 单以论坛活跃程度来计算的话, 已经超越了其他服务器端开发框架, 甚至也超越了 GWT.

目前的主版本 Vaadin 7

Vaadin 7 是最新发布的主版本, 这个版本中 Vaadin API 的变化比 Vaadin 6 更多. 与 Vaadin 6 相比, Vaadin 7 更加以 Web 为中心. 我们已经竭尽所能帮助 Vaadin 在 Web 开发的世界中达到新的高度. 我们所做的工作有些是很简单的, 甚至只是例行公事, 比如修改 bug, 实现新功能. 但站得更高也需要站得更稳. 这也是 Vaadin 7 的目标之一: 重新设计这个产品, 用新的架构帮助 Vaadin 迎接更长期的挑战. Vaadin 7 的很多变化使得它失去了与 Vaadin 6 在 API 层次上的兼容性, 尤其是在客户端, 但作出这样的抉择是因为我们强烈地希望不要将不必要的历史性包袱继续背负下去. Vaadin 7 也包含了一个兼容层, 以便使在既有的应用程序中采用 Vaadin 7 更加简便一些.

在 Vaadin 7 中包含 Google Web Toolkit 是一个重大进展, 这也就意味着我们现在支持 GWT 了. 2012 年夏季 Google 公开了 GWT, Vaadin 公司加入了新的 GWT 指导委员会. 作为委员会的一员, Vaadin 公司可以为 GWT 的成功贡献力量, 努力推动 GWT 成为 Java Web 开发社区的基础.

Vaadin

2.1. 概述	27
2.2. 设置开发环境	28
2.3. Vaadin 库概述	32
2.4. 安装 Vaadin Plugin for Eclipse	33
2.5. 使用 Eclipse 创建和运行一个工程	36
2.6. 通过 Maven 使用 Vaadin	45
2.7. 使用 NetBeans IDE 创建工程	46
2.8. 使用 IntelliJ IDEA 创建工程	47
2.9. Vaadin 安装包	56
2.10. 在 Scala 中使用 Vaadin	57

本章介绍如何安装 Vaadin 推荐的开发工具群、Vaadin 库以及它依赖的其他库，以及如何创建新的 Vaadin 工程。

2.1. 概述

本质上说，只要带有 Java SDK 和 Java Servlet 容器，你可以使用任何开发环境来开发 Vaadin 应用程序。Vaadin 对 Eclipse IDE 有特别的支持，但对于 NetBeans IDE 和 IntelliJ IDEA，也有开发者社区提供支持，你也可以使用其他任何 Java IDE，甚至完全不使用 IDE。

手动管理 Vaadin 库及其他 Java 库将会是十分枯燥无聊的，因此我们建议使用某种能够自动管理包依赖关系的 build 工具。Vaadin 通过 Maven central repository 发布，因此任何一种能够访问 Maven 库的 build 工具或包依赖管理工具都可以使用 Vaadin，比如 Ivy, Gradle, 或 Maven。

对于不同的 IDE, 包依赖管理工具, Vaadin 有多种不同的安装方法，另外你也可以使用安装包来安装 Vaadin:

- 对于 Eclipse IDE, 请使用 Vaadin Plugin for Eclipse, 详情参见 第 2.4 节 “安装 Vaadin Plugin for Eclipse”
- 使用 Vaadin plugin for NetBeans IDE (第 2.7 节 “使用 NetBeans IDE 创建工程”) 或 IntelliJ IDEA
- 使用 Maven, Ivy, Gradle, 或其他的 Maven兼容的包依赖管理工具, 可用于 Eclipse, NetBeans, IDEA, 或使用命令行工具, 详情参见 第 2.6 节 “通过 Maven 使用 Vaadin”
- 无包依赖管理工具时, 请使用安装包, 详情参见 第 2.9 节 “Vaadin 安装包”

2.2. 设置开发环境

本节指导你设置一个参考性的开发环境. Vaadin 支持非常多的开发工具, 因此你可以使用任何 IDE 编写代码, 可以使用几乎任何一种 Java web 服务器来发布应用程序, 可以用任何大多数 Web 浏览器来使用你的应用程序, 可以使用Java 支持的任何一种操作系统

在本节的示例中, 我们使用以下工具群:

- Windows, Linux, 或 Mac OS X
- Sun Java 2 Standard Edition 6.0 (需要 JDK 1.6 或更高版本)
- Eclipse IDE for Java EE Developers
- Apache Tomcat 7.0 (Core) 或更高版本
- Mozilla Firefox 浏览器
- Firebug 调试工具 (可选)
- Vaadin 框架

上述工具群是一组很好的选择, 但你也可以自由选择你所熟悉的其他工具.

开发 Vaadin 应用程序时, 我们建议使用 Java 8, 但你需要确认你的整个开发环境是否全部支持 Java 8. 如果你打算使用服务器端 PUSH 功能, 你需要使用至少兼容 Java EE 7, 并支持 WebSocket 功能的服务器, 比如 Glassfish, TomEE, 等等.

图 2.1. 开发用工具群以及开发流程

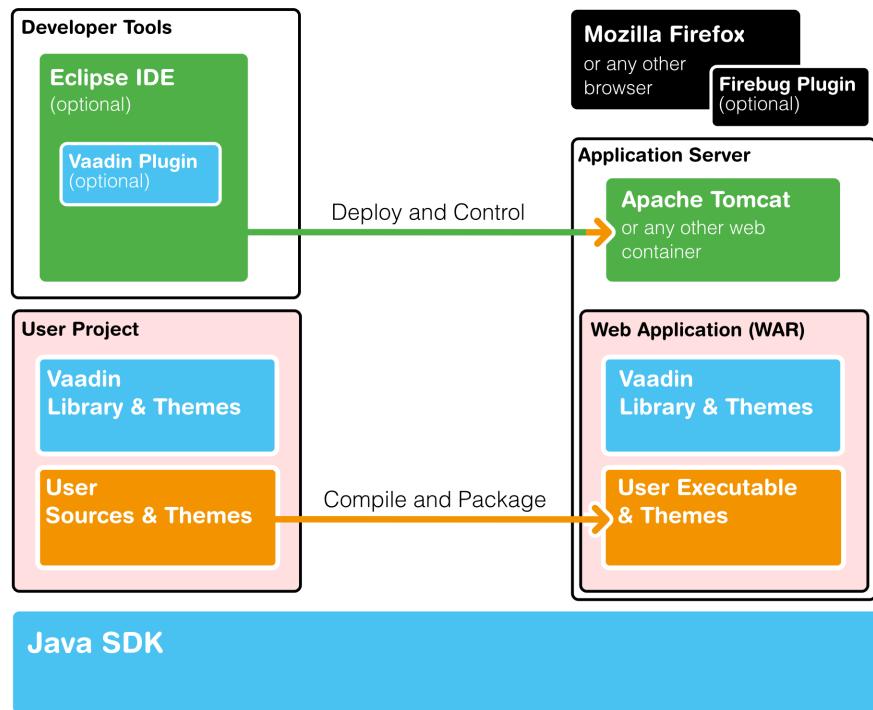


图 2.1 “开发用工具群以及开发流程”描述了开发中使用到的工具群. 应用程序以 Eclipse 过程的形式编写. 除你的应用程序源代码之外, 工程还必须包括 Vaadin 库. 也可以包含工程独有的 theme.

你需要编译你的工程, 并发布到 Web 容器中, 之后你才能使用你的应用程序. 在 Eclipse 环境中, 工程的发布可以使用 WTP (Web Tools Platform)(已包含在 Eclipse EE 包中), WTP 可以从 Eclipse 环境中自动发布 Web 应用程序. 你也可以手动发布工程, 方法是创建一个WAR (Web application archive), 然后将它发布到 Web 容器中.

2.2.1. 安装 Java SDK

Vaadin 和 Eclipse IDE 都需要依赖于 Java SDK. (译注: 此处不正确, Eclipse IDE 自带编译器, 因此它只需要 JRE, 不一定需要 JDK. Vaadin 不明) Vaadin 兼容于 Java 1.6 及以后版本. 如果通过 WebSocket 实现服务器端 PUSH 功能, 则需要 Java EE 7.

Windows

1. 下载 Sun Java 2 Standard Edition 7.0, 地址是 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. 运行安装程序即可安装 Java SDK. 安装过程中使用默认选项即可.

Linux / UNIX

大多数 Linux 系统要么已预装有 JDK, 要么允许你通过包管理系统安装 JDK. 但请注意, Linux 系统一般都选用 OpenJDK 作为默认的 Java 实现. 尽管 OpenJDK 已被测试过可以支持 Vaadin, 而且也可能支持开发用的工具群, 但我们并不对 OpenJDK 提供特别的支持.

对于 OS X 系统, 请注意在 OS X 10.6 或以上版本已包含了 JDK 1.6 或以上版本.

其他情况:

1. 下载 Sun Java 2 Standard Edition 6.0 , 地址是 <http://www.oracle.com/technetwork/java/javase/downloads/>

2. 在某个适当的目录, 例如 /opt, 将下载的安装包解压. 例如, 对于 Java SDK, 输入以下命令(在 Linux 下需要以 root 帐号运行, 或通过 **sudo** 运行):

```
# cd /opt  
# sh (path-to-installation-package)/jdk-7u1-linux-i586.bin
```

然后遵照安装程序的指示进行.

3. 设置 JAVA_HOME 环境变量, 指向到 Java 安装目录. 另外, 还需要将 \$JAVA_HOME/bin 目录追加到 PATH 环境变量中. 具体方法根据你使用的 UNIX 版本不同而不同. 比如, 在 Linux 中使用 Bash shell 时, 你需要将以下内容添加到你的 home 目录下的 .bashrc 或 .profile 文件中:

```
export JAVA_HOME=/opt/jdk1.7.0_01  
export PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
```

你也可以将上述设定设置为全系统所有用户的公共设定, 方法是将上述内容添加到 /etc/bash.bashrc, 或 /etc/profile, 或者你的 Linux 系统的某个等价的文件中. 如果你安装了 Apache Ant 或 Maven, 你也许希望将它们也添加到 path 路径中.

如果通过 bashrc 文件进行设定, 那么你需要打开一个新的 shell 窗口才能让这些设定生效. 如果通过 profile 文件进行设定, 则需要你退出系统并重新 login 一次才能让这些设定生效. 当然, 你也可以在当前的 shell 中执行上述设定命令, 设定将立即生效.

2.2.2. 安装 Eclipse IDE

Windows

1. 下载 Eclipse IDE for Java EE Developers , 地址是 <http://www.eclipse.org/downloads/>
2. 在某个适当的目录解压 Eclipse IDE 安装包. 你可以选择你喜欢的任何一个 ZIP 解压程序, 你可以把文件解压到你喜欢的任何目录, 但在本示例中, 我们双击 ZIP 文件, 然后在 Windows 的压缩目录中选择"解压缩有文件". 在我们的示例中, 我们使用 C:\dev 作为解压目录.

执行以上操作之后, Eclipse 将被安装到 C:\dev\eclipse , 在这个目录下双击 eclipse.exe 即可启动 Eclipse.

Linux / OS X / UNIX

在 Linux 或其他 UNIX 系统下, 我们建议手动安装 Eclipse, 方法如下.

1. 下载 Eclipse IDE for Java EE Developers, 地址是 <http://www.eclipse.org/downloads/>
2. 在适当的目录下解压 Eclipse 包. 注意, 目标目录下一定不要有旧的 Eclipse 存在. 在既有的 Eclipse 目录中安装新版本的 Eclipse 可能会造成它无法运行.
3. 一般来说应该以一般用户身份来安装 Eclipse, 这样的话将来安装插件会容易一些. Eclipse 也会在它的安装目录下保存一些用户设置信息. 安装 Eclipse 请执行以下命令:

```
$ tar zxf (path-to-installation-package)/eclipse-jee-ganymede-SR2-linux-gtk.tar.gz
```

以上命令会将安装包解压到一个名为 `eclipse` 的子目录中.

4. 如果你希望能够从命令行启动 Eclipse, 你需要将 Eclipse 安装目录追加到你的系统 PATH 路径或当前用户 PATH 路径中去, 也可以创建一个符号链接, 或创建一个脚本, 指向 Eclipse 的可执行程序.

除了以上手动安装方法之外, 另一种替代方法是通过你的操作系统的包管理工具来安装 Eclipse. 但是, 我们不推荐这种方法, 因为当你安装 Eclipse 插件时, 你将需要对 Eclipse 安装目录的写权限, 而且, 由操作系统的包管理工具安装的 Eclipse 可能并非最新版本, 因此导致版本兼容问题.

2.2.3. 安装 Apache Tomcat

Apache Tomcat 是一种轻量级 Java Web 服务器, 既可用于开发环境, 也可用于生产环境. 安装 Apache Tomcat 的方法很多, 在本示例中我们只简单的解压 Apache Tomcat 的安装包.

应该使用一般用户权限来安装 Apache Tomcat. 开发阶段中, 你通常会用一般用户权限来运行 Eclipse 或其他 IDE, 如果 Tomcat 服务器不是某个用户专有而是系统级服务, 那么向 Tomcat 服务器发布 Web 应用程序将需要管理员或 root 权限.

1. 下载安装包:

下载 Apache Tomcat 7.0 (Core Binary Distribution), 地址是
<http://tomcat.apache.org/>

2. 在某个适当的目录解压 Apache Tomcat 包, 比如 `C:\dev` (Windows 系统) 或 `/opt` (Linux 或 Mac OS X 系统). Apache Tomcat 的主目录将是 `C:\dev\apache-tomcat-7.0.x` 或 `/opt/apache-tomcat-7.0.x`.

2.2.4. Firefox 和 Firebug

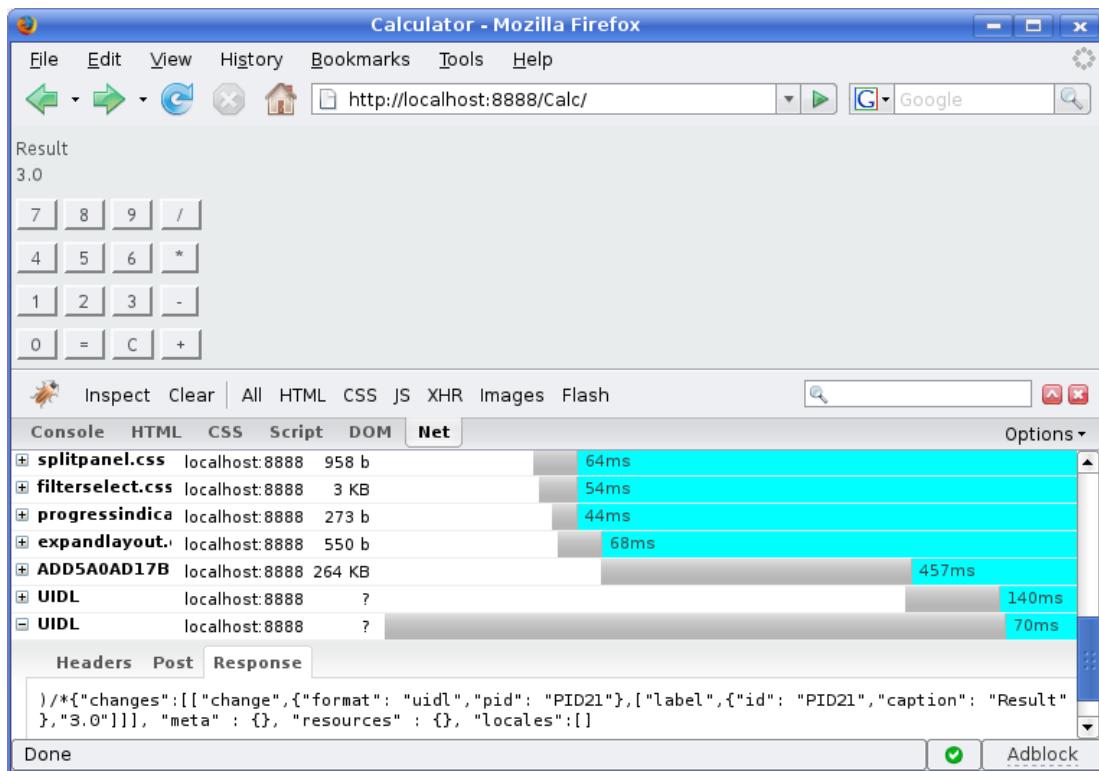
Vaadin 支持很多种 Web 浏览器, 你可以使用其中任何一种用于开发. 如果你打算创建自定义的主题, 定制化的布局, 或者创建新组件, 我们建议你使用 Firefox 浏览器配合调试工具 Firebug, 或者使用 Google Chrome 浏览器, 其中带有内建的调试工具, 与 Firebug 类似.

使用 Firebug 调试 Vaadin

安装完 Firefox 后, 使用它打开以下地址: <http://www.getfirebug.com/>. 遵照网站上的指示即可安装最新版本的 Firebug. 要允许 Firefox 安装插件, 你可能需要点击浏览器窗口顶端的黄色警告条.

Firebug 安装完成后, 你随时可以通过 Firefox 工具条激活它. 图 2.2 “Firefox 调试工具 Firebug” 展示了 Firebug 激活后的状况.

图 2.2. Firefox 调试工具 Firebug



Firebug 最重要的功能是查看页面中的 HTML 元素。在某个页面元素上点击鼠标右键，然后在菜单中选择 **Inspect Element with Firebug** 即可查看这个元素。Firebug 将显示 HTML 元素的树型构造，除此之外还显示适用于各个元素上的 CSS 规则，我们构建 theme 时使用的正是这些 CSS 规则。你也可以在运行中修改 CSS 风格的内容，随时验证各种风格的实际效果。

2.3. Vaadin 库概述

Vaadin 带有很多 JAR 库文件，有些库文件是可选的，有些是可以相互替代的，具体情况取决于你开发的是服务器端还是客户端应用程序，取决于你是否使用了 add-on 组件，也取决于你使用 CSS 还是 Sass 的 theme。

vaadin-server-7.x.x.jar

这是用于开发服务器端 Vaadin 应用程序的主要库文件，详情见 第 4 章 编写服务器端 Web 应用程序。这个库文件依赖于 vaadin-shared 和 vaadin-themes 库文件。如果你没有用到 add-on 组件和定制化 widget 的话，一般来说你可以使用预编译的 vaadin-client-compiled 进行服务器端开发。

vaadin-shared-7.x.x.jar

服务器端开发与客户端开发使用到的共通库文件。一般来说这个库文件总是需要的。

vaadin-client-7.x.x.jar

客户端 Vaadin 框架，包含基本的 GWT API 和 Vaadin 独有的 widget，以及其他附加内容。当使用 vaadin-client-compiler 编译客户端模块时，需要用到这个库文件。如果你的开发中只使用到服务器端框架和预编译的客户端引擎，那么这个库文件就是不必用到的。这个库文件不应该随 Web 应用程序一起发布到运行环境。

vaadin-client-compiler-7.x.x.jar

Vaadin 客户端编译器是一个从 Java 到 JavaScript 的编译器, 可用于创建客户端模块, 如服务器应用程序所需要的客户端引擎(widget 群). 编译器的应用场合, 举例来说, 可以将 add-on 组件编译为应用程序的 widget 群, 详情参见第 17 章 使用 Vaadin Add-on. 关于编译器的详细情况, 参见第 13.4 节 “编译客户端模块”. 注意, 这个库不应该随 Web 应用程序一起发布到运行环境.

vaadin-client-compiled-7.x.x.jar

这个库是预编译的 Vaadin 客户端引擎(widget 群), 包含 Vaadin 内建的基本的 widget. 如果你使用 Vaadin 客户端编译器来编译应用程序的 widget 群, 那么这个库是不必用到的.

vaadin-themes-7.x.x.jar

这个库是 Vaadin 的内建 theme, 包含 SCSS 源文件, 以及预编译的 CSS 文件. 不论是使用基本的 CSS theme, 还是编译 Sass theme, 都需要使用这个库.

vaadin-sass-compiler-1.x.x.jar

Vaadin Sass 编译器负责将 Sass theme 编译为 CSS, 详情请参见第 7.3 节 “优良语法样式表(Sass, Syntactically Awesome Stylesheets)”. 这个库文件依赖于 vaadin-themes-7.x.x.jar, 其中包含 Vaadin 内建 theme 的 Sass 源文件. 在开发模式下, 为了能够实时编译 theme, 需要将这个库文件包含在发布内容中, 但在生产环境下, 发布之前会预编译 theme, 因此应用程序发布时不需要包含这个库文件.

以上各个库文件, 有些会有相互依赖, 而且它们还依赖于安装包的 lib 文件夹内提供的其他库文件, 尤其是 lib/vaadin-shared-deps.jar.

库文件的几种安装方法将在后续章节中详细介绍.

注意, vaadin-client-compiler 和 vaadin-client 库文件不应该与 Web 应用程序一起发布, 因此不要把它们放在 Web 应用程序的 WEB-INF/lib 文件夹下. 其他某些库, 比如 vaadin-sass-compiler, 只在开发环境下需要, 在生产环境下发布时也是不需要的.

2.4. 安装 Vaadin Plugin for Eclipse

如果你使用的是 Eclipse IDE, 那么 Vaadin Plugin for Eclipse 将对你的开发提供很大的帮助. 注意, 在 Eclipse 中你也可以用 Maven 工程的方式创建 Vaadin 工程.

本 plugin 包括:

- 向导, 可用于创建新的 Vaadin 工程, theme, 客户端 widget 或 widget 群.
- 可视化编辑器, 以所见即所得的(WYSIWYG)方式创建组合式 UI 组件. 可视化编辑器支持从源代码到可视模型, 以及从可视模型到源代码的双方向转换能力, 因此这个编辑器可以与你的整个开发流程紧密融合在一起.

2.4.1. 安装 IvyDE Plugin

Vaadin Plugin for Eclipse 依赖于 Apache IvyDE plugin, 因此在安装 Vaadin plugin 之前需要手动安装 Apache IvyDE plugin

1. 选择以下菜单项: **Help → Install New Software....**

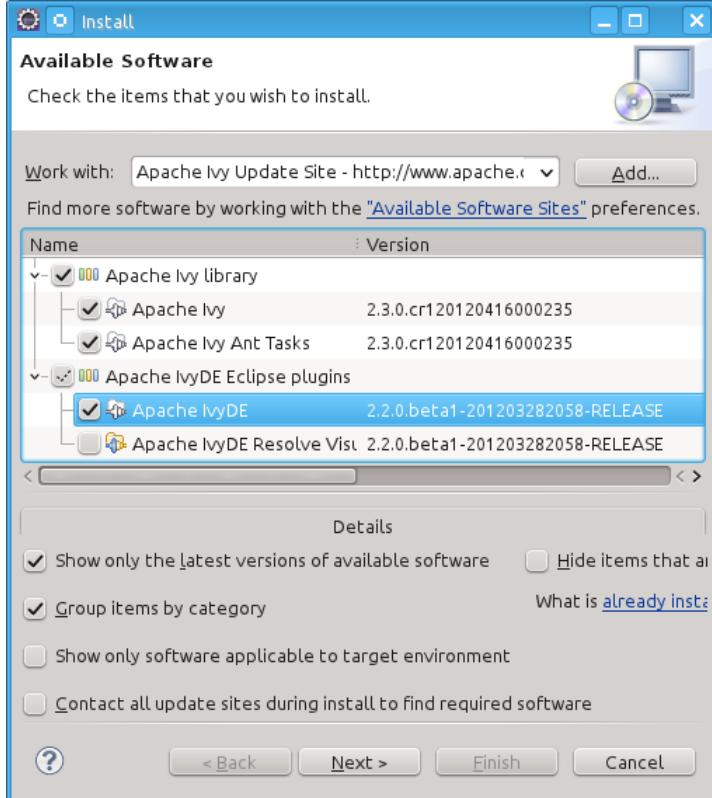
2. 按下 **Add...** 按钮, 添加 IvyDE 的更新站点地址.

输入一个名称, 比如 "Apache Ivy Update Site", 再输入更新站点 URL:

<http://www.apache.org/dist/ant/ivyde/updatesite>

然后点击 **OK** 按钮. Apache Ivy 的更新站点将出现在**Available Software** 窗口中.

3. 在**Work with** 列表中选择我们上面新添加的 "Apache Ivy Update Site".
4. 选择 **Apache Ivy, Apache Ivy Ant Tasks**, 以及 **Apache IvyDE**.



Apache IvyDE Resolve Visualizer 是可选的, 如果选择安装它, 会导致其他一些被依赖的插件也被安装进来.

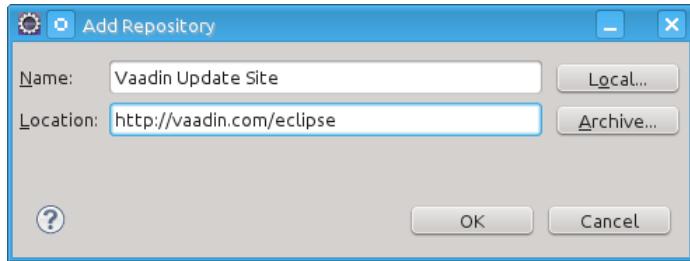
然后, 点击 **Next** 按钮.

5. 确认一下即将安装的插件的详细信息, 无问题的话, 点击 **Next** 按钮.
6. 选择接受或不接受许可协议. 最后, 点击 **Finish** 按钮.
7. Eclipse 可能会警告说某些内容没有数字签名. 如果你觉得没问题的话, 点击 **OK** 按钮.
8. 插件安装完成后, Eclipse 将提示需要重新启动. 你可以暂时不必重启, 等待安装完 Vaadin plugin 后再重启, 安装方法见后续章节, 所以这时你可以选择 **Apply Changes Now**.

2.4.2. 安装 Vaadin Plugin

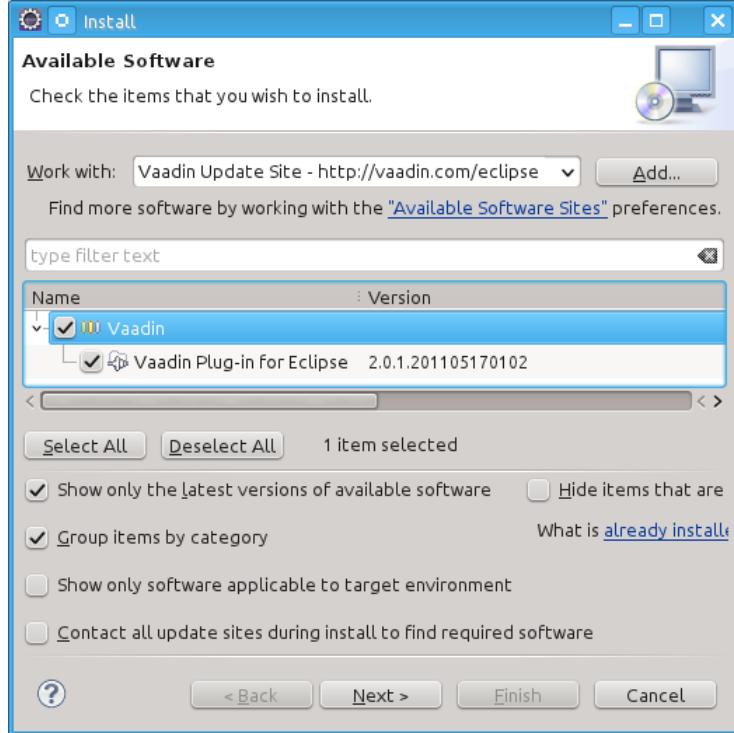
本插件的安装方法如下:

1. 选择以下菜单项: **Help → Install New Software...**
2. 点击 **Add...** 按钮, 添加 Vaadin plugin 更新站点.



输入一个名称，如 "Vaadin Update Site"，然后输入更新站点 URL: <http://vaadin.com/eclipse>. 本插件的最新的、非稳定版本更适合于 Vaadin 的开发中版本或 beta 发布版，如果你希望使用，可以输入以下地址: <http://vaadin.com/eclipse/experimental>, 并为它指定一个单独的名称，比如 "Vaadin Experimental Site". 然后点击 **OK** 按钮. Vaadin 更新站点将出现在 **Available Software** 窗口中.

3. 目前，如果使用本插件的稳定版本，**Group items by category** 选项将变为有效. 如果使用本插件的试验版本，这个选项将变为无效. 但这一点将来可能会有变化.
4. 在插件树中选择所有的 Vaadin 插件.



然后，点击 **Next** 按钮.

5. 确认一下即将安装的插件的详细信息，无问题的话，点击 **Next** 按钮.
6. 选择接受或不接受许可协议. 最后，点击 **Finish** 按钮.
7. 插件安装完成后，Eclipse 将提示需要重新启动. 点击 **Restart** 按钮.

如果你需要使用可视化编辑器, Eclipse 的内嵌浏览器必须设置为有效. 大多数操作系统上的 Eclipse 发行版都包含了一个适当的浏览器引擎, 如果没有, 你可能需要自行安装, 详情请参见 第 10 章 在 Eclipse 中进行可视化的 UI 设计.

关于 Eclipse plugin 安装的更多指导信息, 请参见 <http://vaadin.com/eclipse>.

2.4.3. 更新插件

如果你在 Eclipse 中打开了自动更新选项(参见菜单项 **Window → Preferences → Install/Update → Automatic Updates**), Vaadin plugin 将会与其他插件一样自动更新. 如果没有打开自动更新选项, 你可以手动更新它, 方法如下:

1. 选择菜单项 **Help → Check for Updates**. Eclipse 将访问已安装的各个插件的更新站点.
2. 更新安装完成后, Eclipse 将提示需要重新启动. 点击 **Restart** 按钮.

注意更新 Vaadin plugin 时将只更新插件本身, 而不会更新 Vaadin 库, Vaadin 库的版本是由各工程分别指定的. 更新Vaadin 库的方法请见下一节.

2.4.4. 更新 Vaadin 库

更新 Vaadin plugin 不会更新 Vaadin 库. 库的版本是由各工程分别指定的, 因为不同的工程可能需要使用不同版本的 Vaadin 库, 因此你必须为你的各个工程分别更新 Vaadin 库.

1. 在 Eclipse 编辑器中打开 ivy.xml 文件.
2. 在文件最前部的 entity 定义部分修改 Vaadin 版本.

```
<!ENTITY vaadin.version "7.0.1">
```

你可以象上面的例子那样指定一个固定的版本号, 也可以使用一个动态的版本号标记, 如 latest.release. 关于包依赖关系声明方法的更多信息, 请阅读 Ivy 的文档.

3. 在工程上单击鼠标右键, 选择菜单项 **Ivy → Resolve**.

版本库文件的更新过程可能需要耗费数分钟. 进度状况可以在 Eclipse 状态栏中看到. 点击状态栏中的指示图标可以看到进度状况的更详细信息.

4. 如果你曾经为你的工程编译过 widget 群, 那么 Vaadin 库版本更新后需要重新编译, 方法是在 Eclipse 工具栏中点击 **Compile Vaadin widgets** 按钮.
5. 将 Eclipse 集成的 Tomcat (或者其他类型的服务器) 停止运行, 清空它的缓存, 方法是在服务器上单击鼠标右键, 然后选择菜单项 **Clean** 和 **Clean Tomcat Work Directory**, 然后重启服务器.

如果更新完 Vaadin 库文件之后遭遇到问题, 你可以尝试清空 Ivy 的解析结果缓存, 方法是在工程上单击鼠标右键, 然后选择菜单项 **Ivy → Clean all caches**. 然后重新执行 **Ivy → Resolve** 和其他操作.

2.5. 使用 Eclipse 创建和运行一个工程

本节介绍如何使用 Vaadin Plugin 创建新的 Eclipse 工程. 包括以下几个步骤:

1. 创建新工程
2. 编写源代码

3. 配置和启动 Tomcat (或其他 Web 服务器)

4. 打开 Web 来访问 Web 应用程序

我们还将介绍在 Eclipse 中如何使用 debug 模式来调试应用程序.

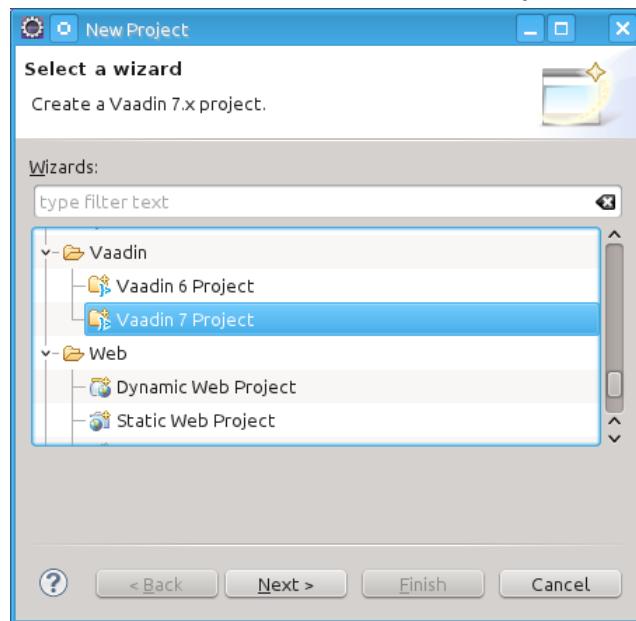
下面的教程假定你已经安装了 Vaadin Plugin for Eclipse, 并已配置好了你的开发环境, 如 第 2.2 节“设置开发环境”所述.

2.5.1. 创建工程

下面让我们用前面章节中安装好的开发工具来创建我们的第一个应用程序工程. 首先, 启动 Eclipse, 然后执行以下步骤:

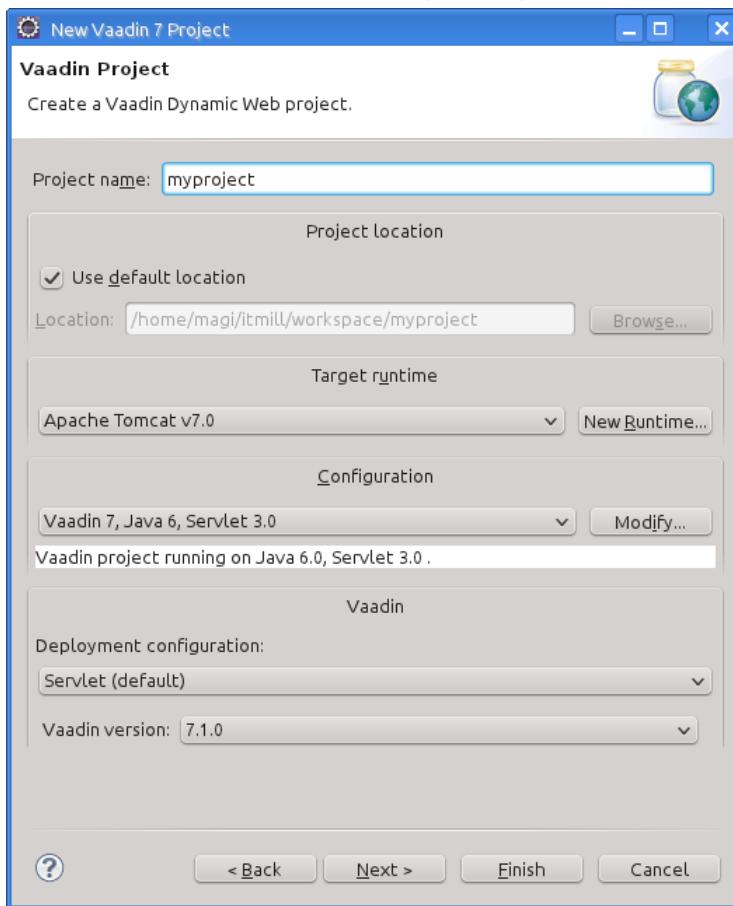
1. 创建一个新工程, 方法是选择菜单项 **File → New → Project...**

2. 在 **New Project** 窗口中, 选择菜单项 **Web → Vaadin 7 Project**, 并点击 **Next** 按钮.



如果你选择 Vaadin 6, 请阅读本书的 Vaadin 6 版.

3. 在向导的 **Vaadin Project** 窗口, 你需要为 Web 工程做一些基本设定. 首先你至少需要指定 *project name* 以及这个工程使用的运行环境 (*runtime*); 其他设定使用默认值即可.



Project name

为工程指定一个名称. 工程名必须是一个跨平台合法的文件名, 也必须是 URL 中合法的组成部分, 因此建议在工程名中只使用小写字母, 数字, 下划线, 以及横线.

Use default location

指定工程在文件系统中的存储目录. 默认位置是在你的 workspace 文件夹下, 一般来说不必修改. 但有些情况下你也许需要指定工程的存储目录, 比如, 如果你希望在版本管理系统管理下的源代码树中创建 Eclipse 工程, 那么工程的位置就很重要了.

Target runtime

指定用来发布应用程序的服务器. 你安装过的服务器, 比如 Apache Tomcat, 将被自动选中. 如果没有的话, 请点击 **New** 按钮, 可在 Eclipse 中配置新的服务器.

Configuration

选择你希望使用的配置; 一般来说应该使用应用程序服务器的默认配置. 如果你需要修改 project facet, 请点击 **Modify** 按钮. 推荐使用 Servlet 3.0 配置, 它使用 @WebServlet 进行部署, Servlet 2.4 配置则使用老式的 web.xml 进行部署.

Deployment configuration

这个选项用于设定应用程序将被发布到的运行环境, 以便生成正确的工程目录结构及相应的设定文件. 可用的选项包括:

- **Servlet (default)**

- **Google App Engine Servlet**

- **Generic Portlet (Portlet 2.0)**

New Project 向导的后续步骤依赖于你选择的发布配置的内容; 本节中列出的步骤只适用于默认的 servlet 配置. 其他几种运行环境下 Vaadin 的使用方法, 请参见第 11.7 节“与 Google App Engine 的集成”和第 12 章与 Portal 集成.

Vaadin version

选择使用的 Vaadin 版本. 下拉框中默认选中的是 Vaadin 的最新可用版本. 如果你希望随时使用最新的、非稳定版本的话, 你可以选择每夜 SNAPSHOT 编译版.

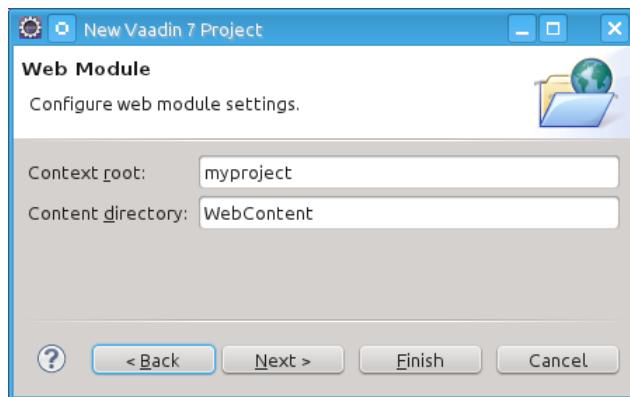
在这里选定的版本, 将来还可以在 `ivy.xml` 中修改.

Create TestBench test

当这个选项有效时, 自动生成的应用程序框架(stub)代码中, 将包括测试用例(test case), 它将使用 Vaadin TestBench 来测试 UI, 详情请参见 第 21 章 Vaadin TestBench. Vaadin TestBench API 库文件将会以一个依赖项的形式, 包含在 `ivy.xml` 之内. 创建应用程序框架代码需要使用 Vaadin 7.3 或更高版本.

如果其他所有设定都使用默认值, 可以在这一步点击 **Finish** 按钮, 否则请点击 **Next** 按钮.

4. 在向导的 **Web Module** 窗口中可以配置 Web 应用程序(WAR) 的基本发布设定, 以及 Web 应用程序工程的结构. 所有设定项都有预先指定的默认值, 一般来说你应该这些默认设定.



Context Root

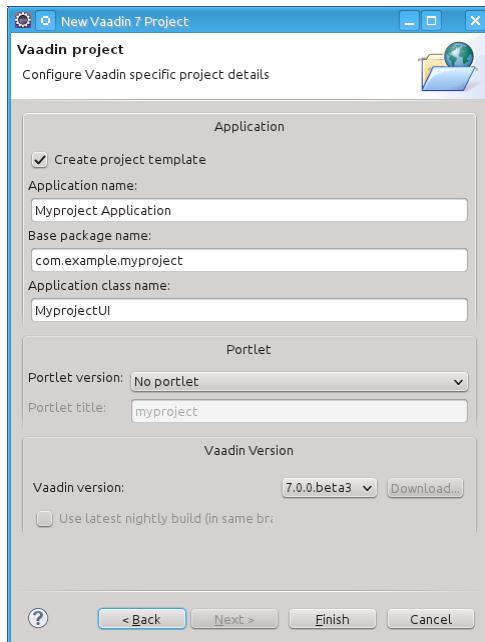
应用程序的 context root 指定了访问应用程序时的 URL. 比如, 如果工程的 context 被设定为 myproject, 并且只有一个唯一的 UI, 那么 URL 将是 `http://example.com/myproject`. 向导默认使用第一步中指定的工程名作为 context 名. 你将来可以在 Eclipse 工程的属性中修改 context root.

Content Directory

指定一个目录名, 这个目录之下的一切内容都将被包含到 Web 应用程序(WAR) 中, 然后被发布到 Web 服务器上. 这里输入的目录是从工程根路径开始的相对路径.

你可以使用默认设定, 点击 **Next** 按钮.

5. 向导的 **Vaadin project** 窗口包括一些 Vaadin 专有的设定内容. 如果你是初次使用 Vaadin, 建议不要修改默认设定. 大多数设定项都可以留在将来再修改, 只有 portlet 相关设定除外.



Create project template

让向导帮你创建 UI 类的基本框架.

Application Name

应用程序 UI 的名称, 将显示在浏览器窗口的标题栏中.

Base package name

Java 包的名称, 应用程序 UI 类将被置于这个包之下.

Application/UI class name

应用程序 UI 类的名称, 应用程序的 UI 代码将在这个 UI 类中开发.

Portlet version

当 portlet 版本有指定时 (只支持 Portlet 2.0), 向导将创建应用程序在 portal 中运行时所需要的文件. 关于 portlet 的详情请参见 第 12 章 与 Portal 集成 .

最后, 点击 **Finish** 即可创建工作.

2.5.2. 浏览一下工程结构

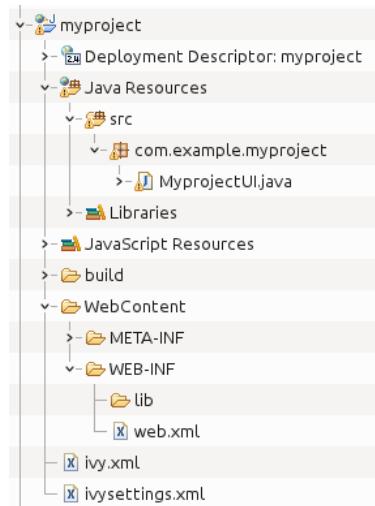
New Project 向导运行完成后, 它将为你完成所有工作: UI 类的基本框架已经生成到 src 目录下, WebContent/WEB-INF/web.xml 文件中已经包含了应用程序的部署描述信息. Eclipse 的 Project Explorer 视图中显示的工程层级结构参见 图 2.3 “一个新的 Vaadin 工程”.

Vaadin 库以及工程依赖的其他包由 Ivy 负责管理. 注意, 库文件并没有存储到工程文件夹之内, 但它们仍然被显示在虚拟文件夹 **Java Resources** → **Libraries** → **ivy.xml** 之下.

UI 类

插件自动创建的 UI 类包含以下代码:

图 2.3. 一个新的 Vaadin 工程



```

package com.example.myproject;

import com.vaadin.ui.UI;
...

@SuppressWarnings("serial")
@Theme("myproject")
public class MyprojectUI extends UI {

    @WebServlet(value = "/*", asyncSupported = true)
    @VaadinServletConfiguration(
        productionMode = false,
        ui = MyprojectUI.class)
    public static class Servlet extends VaadinServlet {
    }

    @Override
    protected void init(VaadinRequest request) {
        final VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        setContent(layout);

        Button button = new Button("Click Me");
        button.addClickListener(new Button.ClickListener() {
            public void buttonClick(ClickEvent event) {
                layout.addComponent(
                    new Label("Thank you for clicking")));
            }
        });
        layout.addComponent(button);
    }
}

```

在 Servlet 3.0 工程中，部署的配置使用 servlet 类，以及一个 @WebServlet 标注。插件创建的框架代码中包含一个 servlet 类，但是是 static inner 类的形式。如果你希望，也可以将它重构为一个单独的一般类。

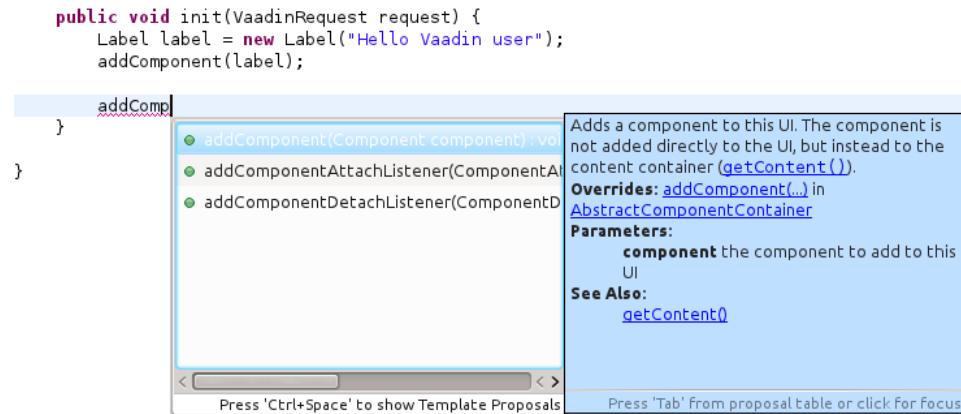
对于 Servlet 2.3 工程，你将得到一个 web.xml 部署描述文件。

关于发布的详细信息, 请参见 第 4.8.4 节 “使用部署描述文件 web.xml”.

2.5.3. 针对 Eclipse 的编码提示

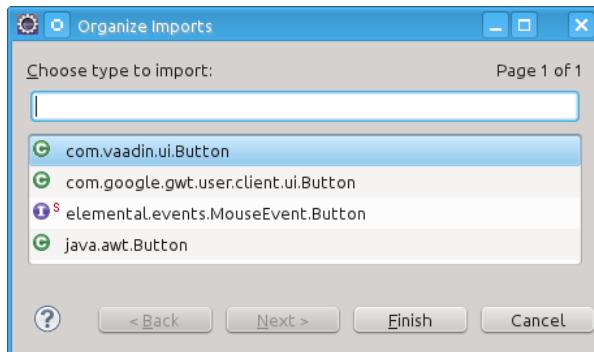
Eclipse 最有用的功能之一就是 自动代码补完. 在编辑器中按下 **Ctrl+Space** 键, 将弹出一个下拉列表, 其中包括所有可能的类名或方法名的补完提示, 参见 图 2.4 “Eclipse 中的 Java 源代码自动补完功能”, 自动补完提示的内容将根据目前光标所在位置的源代码上下文而不同.

图 2.4. Eclipse 中的 Java 源代码自动补完功能



要向一个类添加一条 import 语句, 比如 **Button**, 只需要按下 **Ctrl+Shift+O** 键, 或者在编辑器窗口的左侧点击红色的错误提示图标. 如果希望引入的类存在于多个包中, 将会出现一个选项列表, 参见 图 2.5 “自动导入 Java 类”. 对于服务器端开发, 一般应该使用 com.vaadin.ui 或 com.vaadin.server 包下的类. 你不能将客户端类(位于 com.vaadin.client 包之下)或 GWT 类用于服务器端开发.

图 2.5. 自动导入 Java 类



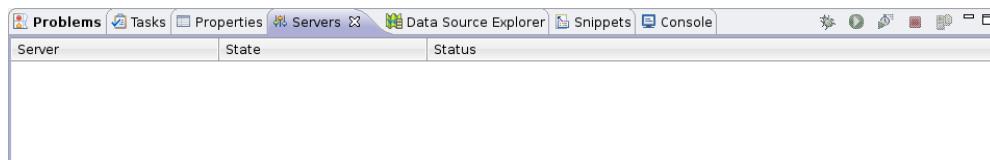
2.5.4. 配置和启动 Web 服务器

Eclipse IDE for Java EE Developers 包含了 WST (Web Standard Tools), WST 可以管理多种 Web 服务器, 当工程中文件发生变化时, WST 可以将 Web 内容自动发布到服务器中.

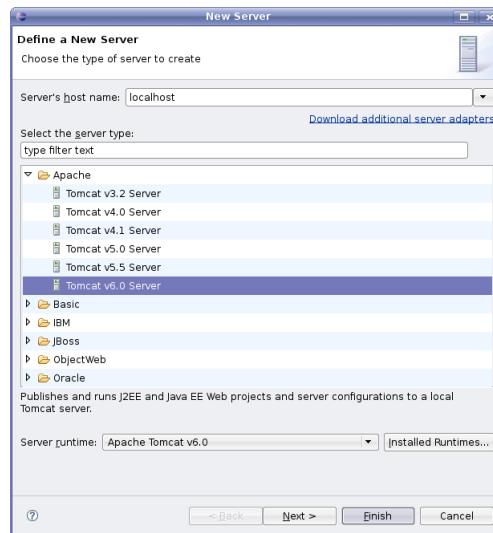
请确认 Tomcat 是在一般用户权限下安装的. 如果运行 Eclipse 的用户对 Tomcat 安装目录下的配置和发布目录没有写入权限, 在 Eclipse 中配置 Web 服务器时将会失败.

然后请执行以下几步.

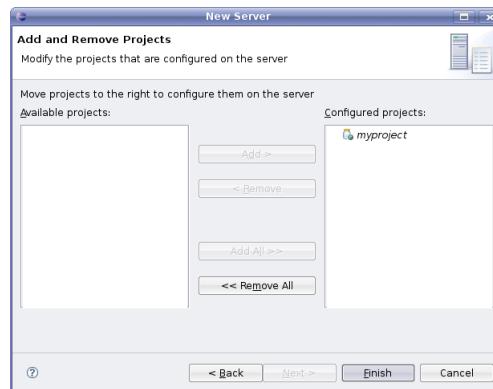
1. 在 Eclipse 下方窗口栏中切换到 **Servers** 页. Eclipse 初期安装完毕时, 这个窗口中显示的服务器列表将是空的. 在窗口的空白位置点击鼠标右键, 选择菜单项 **New → Server**.



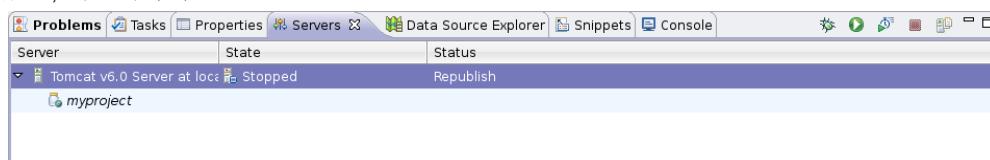
2. 选择 **Apache → Tomcat v7.0 Server**, 然后将 **Server's host name** 设置为 `localhost`, 默认值就是如此. 如果你的系统中只安装过一份 Tomcat, **Server runtime** 将只有唯一一个可选项. 点击 **Next** 按钮.



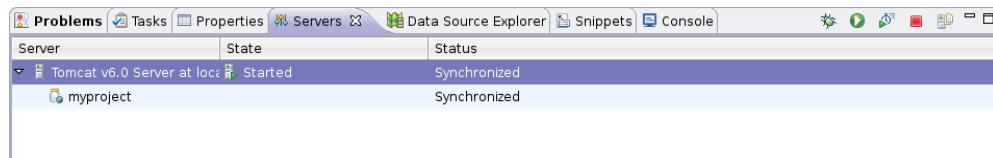
3. 将你的工程添加到服务器中, 方法是在左侧列表中选中工程, 然后点击 **Add** 按钮, 将它追加到右侧的 **Configured projects** 列表. 然后点击 **Finish** 按钮.



4. 至此, 服务器与工程都已在 Eclipse 中设置完毕, 将显示在 **Servers** 窗口中. 要启动服务器, 请在服务器上点击鼠标右键, 选择菜单项 **Debug**. 如果要在非 debug 模式下启动服务器, 请选择菜单项 **Start**.



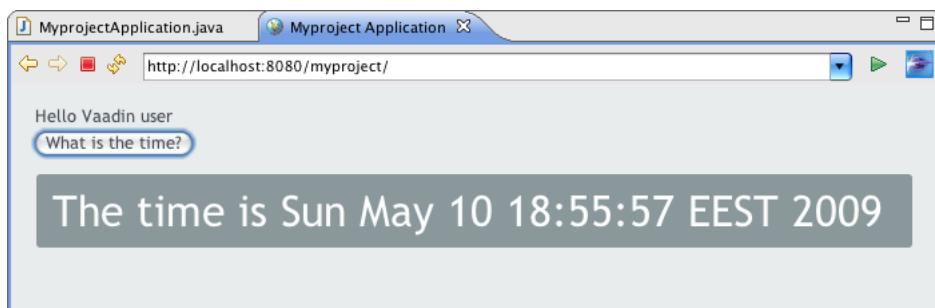
5. 服务器将开始运行，工程的 WebContent 目录将被发布到服务器上，其地址为 <http://localhost:8080/myproject/>.



2.5.5. 运行和调试

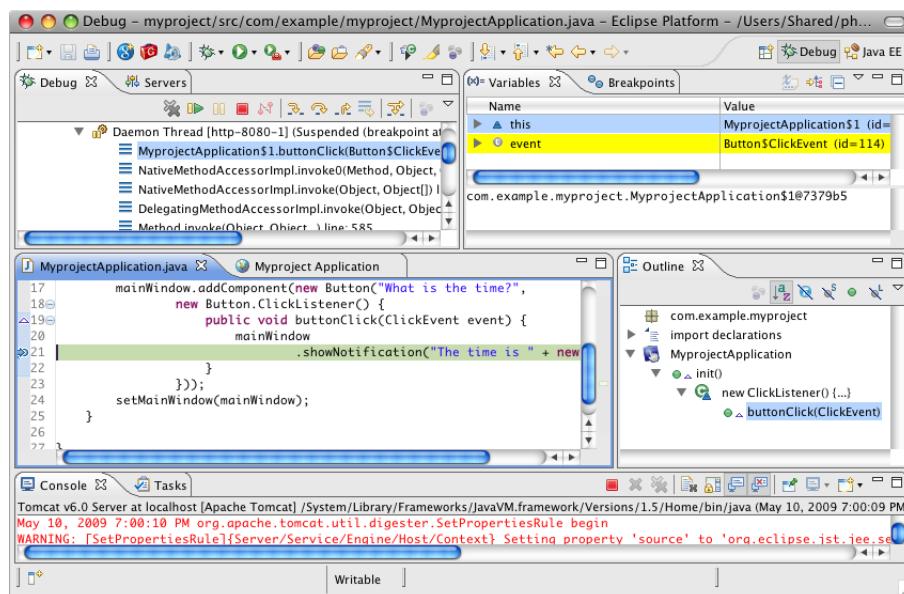
运行应用程序是很容易的，方法是在 **Project Explorer** 窗口中选择 **myproject**，然后选择菜单项 **Run → Debug As → Debug on Server**. Eclipse 会在它的内嵌 Web 浏览器中打开应用程序.

图 2.6. 运行 Vaadin 应程序



你可以在 Java 代码中设置断点，方法是源代码编辑器窗口左侧空白位置双击鼠标。比如，如果你在 `buttonClick()` 方法上添加断点，然后在浏览器中点击 **What is the time?** 按钮，Eclipse 将询问是否切换到 Debug 透视图(perspective)。Debug 透视图将显示当前程序的运行停止在断点的何处。此时你可以查看应用程序的各种状态，也可以修改这些状态。要继续程序的运行，请从 **Run** 菜单中选择菜单项 **Resume**.

图 2.7. 调试 Vaadin 应程序



上文我们介绍了如何调试服务器端应用程序. 客户端应用程序和 widget 的调试方法参见第 13.6 节“调试客户端代码”.

2.6. 通过 Maven 使用 Vaadin

Maven 是一个被广泛使用的构建和包依赖管理系统. Vaadin 核心库以及所有的 Vaadin 插件都可以通过 Maven 得到. 你可以将 Eclipse 或 NetBeans 中作为前端工具来使用 Maven, 或者也可以通过命令行来使用 Maven, 详情见本节余下内容.

除了通常的 Maven 方式外, 你还可以任何一种与 Maven 兼容的构建和包依赖管理系统, 比如 Ivy 或 Gradle. 关于 Gradle 的使用, 参见 Gradle Vaadin Plugin. Vaadin Plugin for Eclipse 使用 Ivy 来为 Vaadin 工程解析它的包依赖关系, Vaadin Plugin for Eclipse 还会为你提供基本的 Ivy 配置.

2.6.1. 通过命令行使用 Maven

你可以通过以下命令新建一个 Maven 工程:

```
$ mvn archetype:generate
-DarchetypeGroupId=com.vaadin
-DarchetypeArtifactId=vaadin-archetype-application
-DarchetypeVersion=7.x.x
-DgroupId=your.company
-DartifactId=project-name
-Dversion=1.0
-Dpackaging=war
```

各参数的说明如下:

archetypeGroupId

指定 archetype 的 group ID, 对 Vaadin archetype, 应该设置为 com.vaadin.

archetypeArtifactId

指定 archetype ID. Vaadin 7 目前支持 vaadin-archetype-application, 用于服务器端应用程序工程, 以及 vaadin-archetype-widget, 用于客户端 widget 工程.

archetypeVersion

指定 archetype 的版本. 对于一般的 Vaadin 发行版, 应该使用 LATEST. 对于预发布版应该使用明确的版本号, 比如 7.0.0.beta3.

groupId

指定你的工程的 Maven group ID. 一般来说应该使用你所属组织的域名的反序, 比如 com.example. group ID 还被用作你工程源代码中的 Java 包名前缀, 所以它必须是合法的 Java 包名 - 其中只能使用字母, 数字, 以及下划线.

artifactId

artifact(也就是你的工程)的 ID. 这个 ID 可以包含字母, 数字, 横线, 以及下划线. 它将添加到 group ID 之后, 组成源代码中的 Java 包名称. 比如, 如果 group ID 是 com.example, artifact ID 是 myproject, 那么工程的源代码将被放在 com.example.myproject 包之下.

version

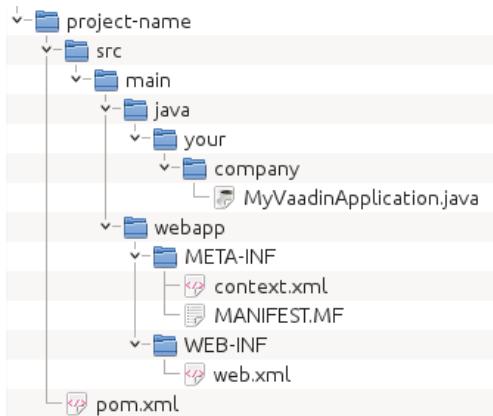
指定你应用程序的初始版本号. 版本号必须遵守 Maven 版本号格式规则.

packaging

指定你的工程的打包方式. 通常使用 war.

创建工程可能会耗费一些时间, 因为 Maven 需要下载所有的依赖包. 创建出来的工程结构参见图 2.8 “使用 Maven 新建的 Vaadin 工程”.

图 2.8. 使用 Maven 新建的 Vaadin 工程



2.6.2. 编译和运行应用程序

应用程序发布之前, 它必须编译并打包为 WAR 形式. 你可以运行 Maven 的 package 目标(goal)来执行编译和打包任务, 方法如下:

```
$ mvn package
```

最终打包完成的 WAR 包文件的位置, 将输出到命令行中. 然后你就可以将这个 WAR 包文件发布到你喜欢的应用程序服务器了.

通过 Maven 运行 Vaadin 应用程序的最简便方法是使用轻量的 Jetty Web 服务器. 编译完成后, 你需要做的仅仅是:

```
$ mvn jetty:run
```

上述 Maven 目标(goal)会在 8080 端口上启动 Jetty 服务器, 然后将应用程序部署到服务器上. 然后你可以在浏览器中打开应用程序, 地址是 <http://localhost:8080/project-name>.

2.6.3. 使用插件和定制 Widget 群

如果你使用了包含 widget 群的 Vaadin 插件, 或者希望创建自定义的 widget, 你需要在 POM 设定文件中打开 widget 群的编译功能. 详细的设定方法参见第 17.4 节 “在 Maven 工程中使用 Add-on”.

2.7. 使用 NetBeans IDE 创建工程

在 NetBeans IDE 中开发 Vaadin 应用程序的最简便方法是使用 Vaadin Plugin for NetBeans. 这个插件可以帮助你便利地创建 Vaadin 工程, 还提供了很多其他功能. 插件可在以下地址下载: <http://plugins.netbeans.org/plugin/50531/vaadin-plug-in-for-netbeans>. 下载页面还包括一个链接, 指向该插件功能概要介绍的 NetBeans Wiki 页面.

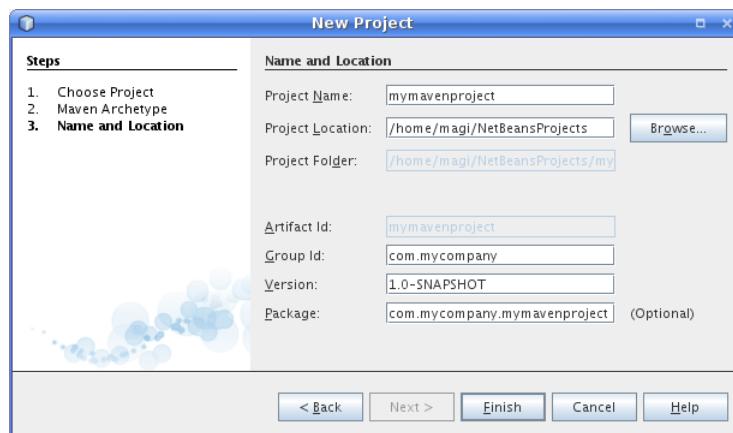
不使用上述插件的话, 最简便方法是以“使用 Vaadin Archetype 的 Maven 工程”方式来创建 Vaadin 工程. 也可以作为通常的 Web 应用程序来创建, 但这种方式需要很多手工步骤来安装 Vaadin 库, 创建 UI 类, 配置 Servlet, 创建 theme, 等等等等.

2.7.1. 使用 Vaadin Archetype 的 Maven 工程

创建 Vaadin archetype 的 Maven 工程, 将自动生成应用程序的基本框架, 包括 UI 类和工程的 theme, 还会自动定义 web.xml 部署描述文件, 而且还将自动取得最新版本的 Vaadin 库.

1. 选择菜单项 **File → New Project**.
2. 选择 **Maven → Project from Archetype**, 然后点击 **Next** 按钮.
3. 找到 vaadin-archetype-application, 选中它, 然后点击 **Next** 按钮.
4. 在向导的 **Name and Location** 窗口, 输入 **Project Name**, 建议只是使用小写英文字母, 因为它也会被用作工程内 Java 包名. 修改其他设定参数, 然后点击 **Finish** 按钮.

图 2.9. 在 NetBeans 中添加新的 Maven 工程



创建工程可能需要耗费一些时间, 因为 Maven 需要装载所有依赖到的包. 创建完成后, 你可以通过 **Projects** 窗口, 在工程上点击鼠标右键, 然后选择菜单项 **Run** 来运行工程. 这里将出现 **Select deployment server** 窗口, 请在这个窗口中选择 **Glassfish** 或 **Apache Tomcat**, 然后点击 **OK** 按钮. 如果一切顺利的话, NetBeans 将在 8080 端口上启动服务器, 然后打开默认浏览器来显示你的 Web 程序. 默认浏览器是什么, 取决于你的操作系统设定. 如果 Web 程序未能自动显示, 你可以手动访问它, 地址是 <http://localhost:8080/myproject>. 默认情况下, 工程名称会被使用为应用程序的 context 路径.

2.8. 使用 IntelliJ IDEA 创建工程

IntelliJ IDEA 的 Ultimate 版支持创建 Vaadin 应用程序, 还可以在集成的应用服务器中运行或调试它. 如果使用 Community 版, 创建 Vaadin 应用程序的最简便方法是创建 Vaadin Archetype 的 Maven 工程, 然后通过 Maven 的 run/debug 配置将应用程序发布到服务器上.

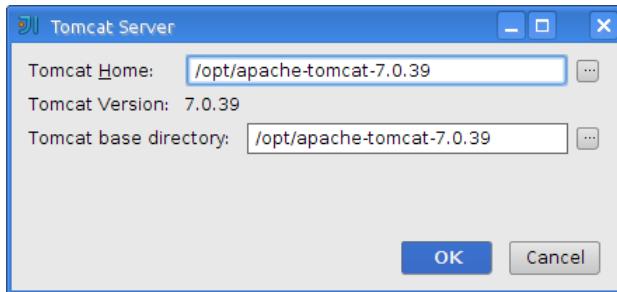
详情请到 IntelliJ IDEA 百科全书 WIKI 系统阅读以下文章, "Creating a simple Web application and deploying it to Tomcat".

2.8.1. 配置应用程序服务器

在 IntelliJ IDEA 的 Ultimate 版中运行应用程序, 首先需要安装和配置与 IDE 集成的应用服务器. IntelliJ IDEA 的 Ultimate 版集成了很多常用的应用服务器.

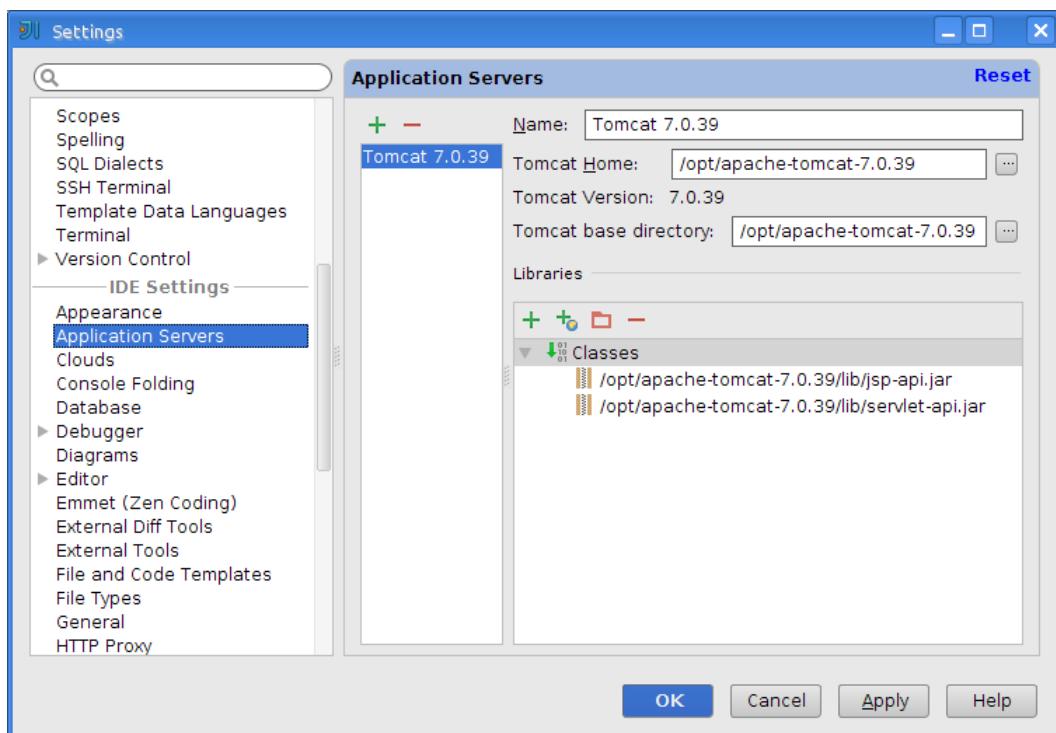
在本示例中我们配置 Apache Tomcat, 步骤如下:

1. 下载 Tomcat 安装包，并解压缩到适当的本地目录，详情参见 第 2.2.3 节“安装 Apache Tomcat”。
2. 选择菜单项 **Configure → Settings**。
3. 选择 **IDE Settings → Application Servers**。
4. 选择 **+ → Tomcat Server**，添加一个 Tomcat 服务器，也可以选择其他服务器类型。注意，对于服务器端 PUSH 功能，需要使用支持 WebSocket 的服务器，比如 Glassfish 或 TomEE。
5. 在 Tomcat Server 对话框中，指定服务器的 home 目录。



点击 **OK** 按钮。

6. 复查应用程序服务器的设定内容是否正确。

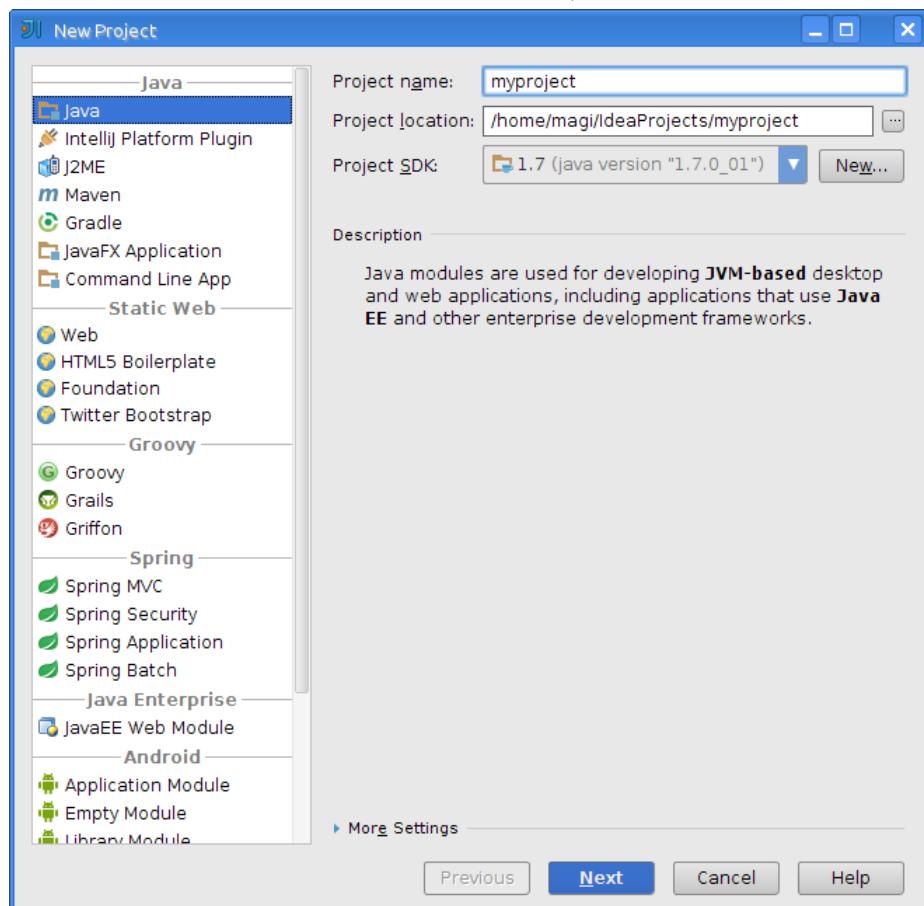


然后点击 **OK** 按钮。

2.8.2. 创建 Vaadin Web 应用程序工程

在 Welcome 页中，进行如下操作：

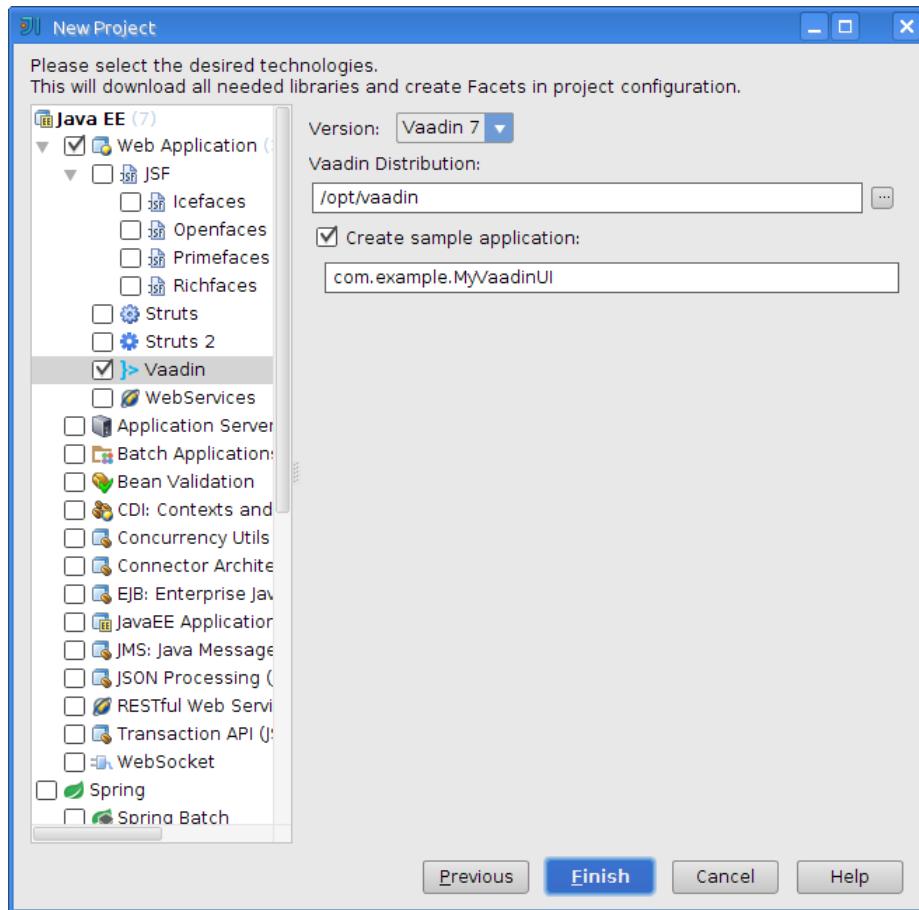
1. 下载 Vaadin 安装包, 解压缩到本地目录, 详情参见 第 2.9 节 “Vaadin 安装包”.
2. 选择菜单项 **New Project**
3. 在 **New Project** 窗口, 选择 **Java**
4. 输入 **Project name** 和 **Project location**, 然后选择工程使用的 **Java SDK**. Vaadin 需要 Java 6 以上版本. 如果你没有预先配置 Java SDK, 你可以在这个界面中配置它.



点击 **Next** 按钮.

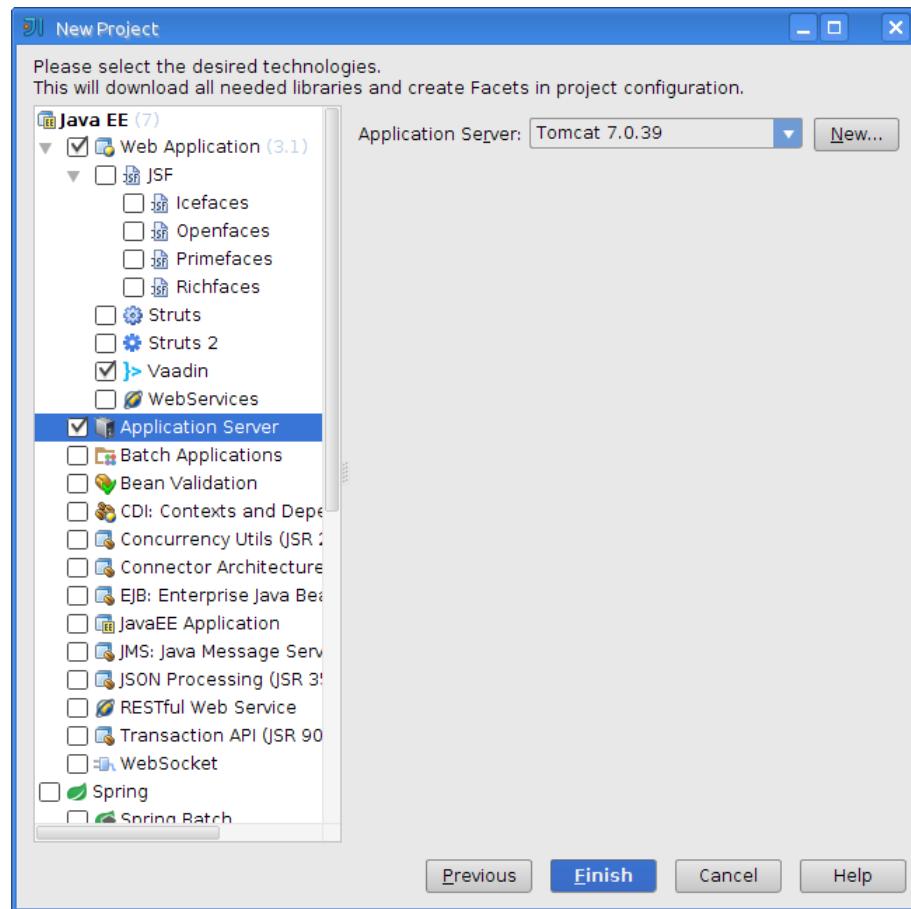
5. 选择 **Web Application** → **Vaadin**, 将 Vaadin 追加到工程中.

6. 选择 Vaadin **Version** 和 **Distribution** 安装路径. 你可能希望自动生成应用程序框架, 请选择 **Create sample application**, 并指定 UI 类的名称.



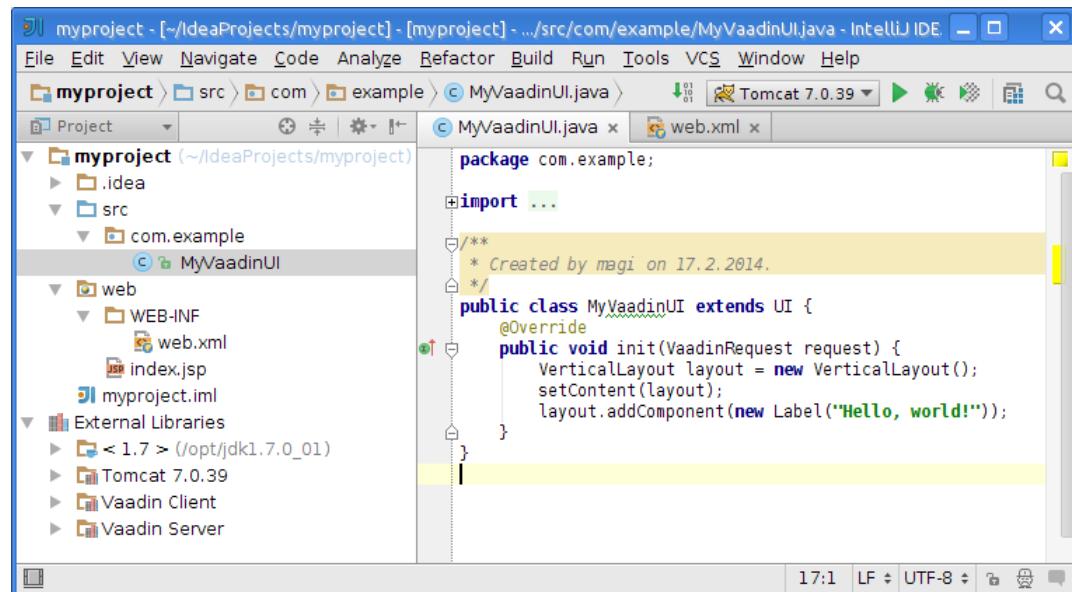
这里暂时 不要点击 **Finish** 按钮.

7. 在同一个窗口中选择 **Application Server**. 将它设置为与 IntelliJ IDEA 集成的服务器, 服务器的配置方法参见 第 2.8.1 节 “配置应用程序服务器”.



8. 点击 **Finish** 按钮.

工程创建完成, 其中包含 UI 类的框架, 以及 web.xml 部署描述文件.



向导目前还不能自动生成 Servlet 类, 而且它目前使用的还是 Servlet 2.4 规格的部署方式, 部署设定由 web.xml 文件指定.

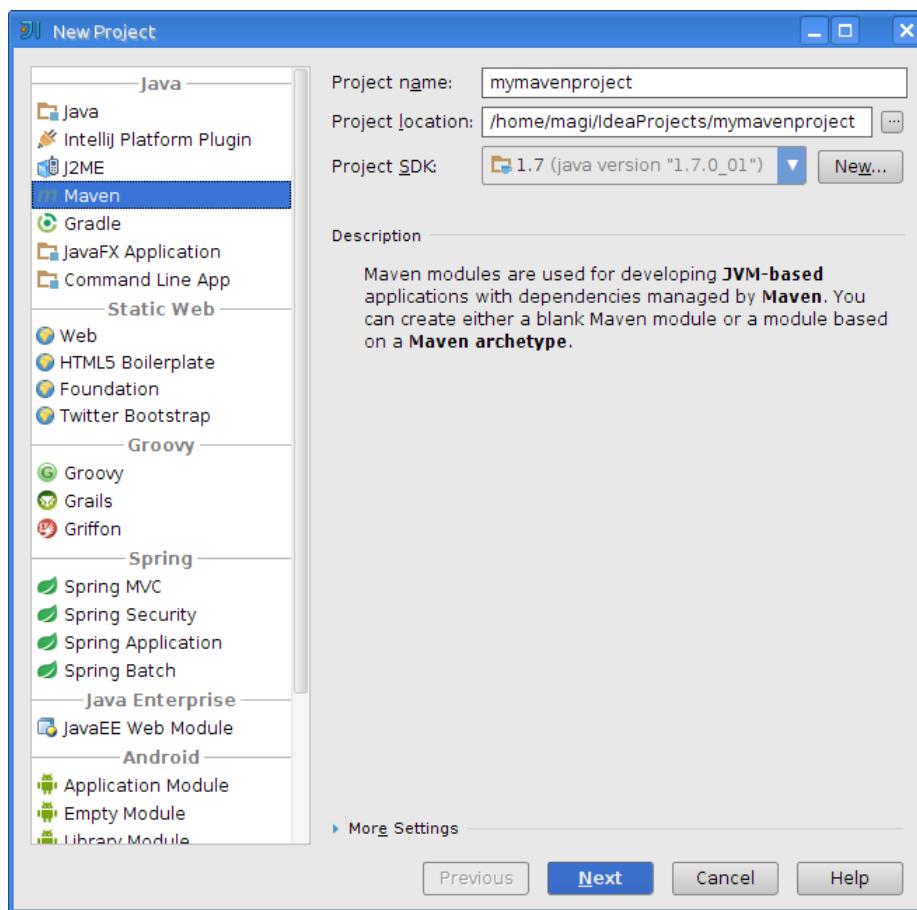
发布工程

要将应用程序发布到集成的 Web 服务器上, 请在工程内的 index.jsp 文件上点击鼠标右键, 然后选择菜单项 **Run 'index.jsp'**. 如果集成的服务器尚未启动的话, 以上操作将启动它, 并将在默认的浏览器内打开应用程序页面.

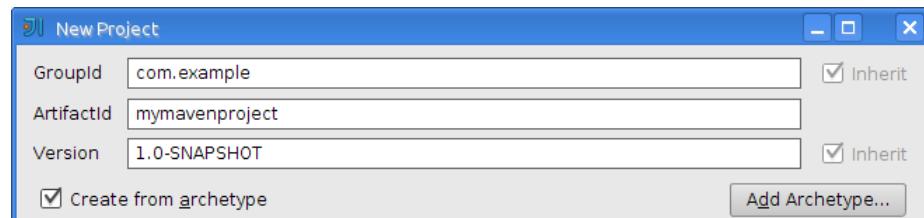
2.8.3. 创建 Maven 工程

你也可以在 IntelliJ IDEA 中创建 Maven 工程. 当使用 IntelliJ IDEA 的 Community 版时, 推荐使用这种方式. 这种方式将不会与应用程序服务器自动集成, 但可以使用 run/debug 配置将应用程序发布到服务器上.

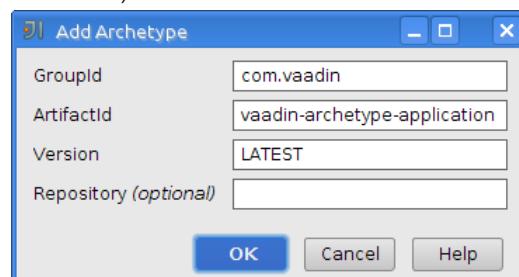
1. 选择菜单项 **New Project**
2. 在 **New Project** 窗口中, 选择 **Maven**
3. 输入工程名, 位置, 以及工程使用的 Java SDK. Vaadin 需要 Java 6 以上版本. 点击 **Next** 按钮.



4. 为工程指定 Maven **GroupId**, **ArtifactId**, 以及 **Version**, 这些项目都可以使用默认值.

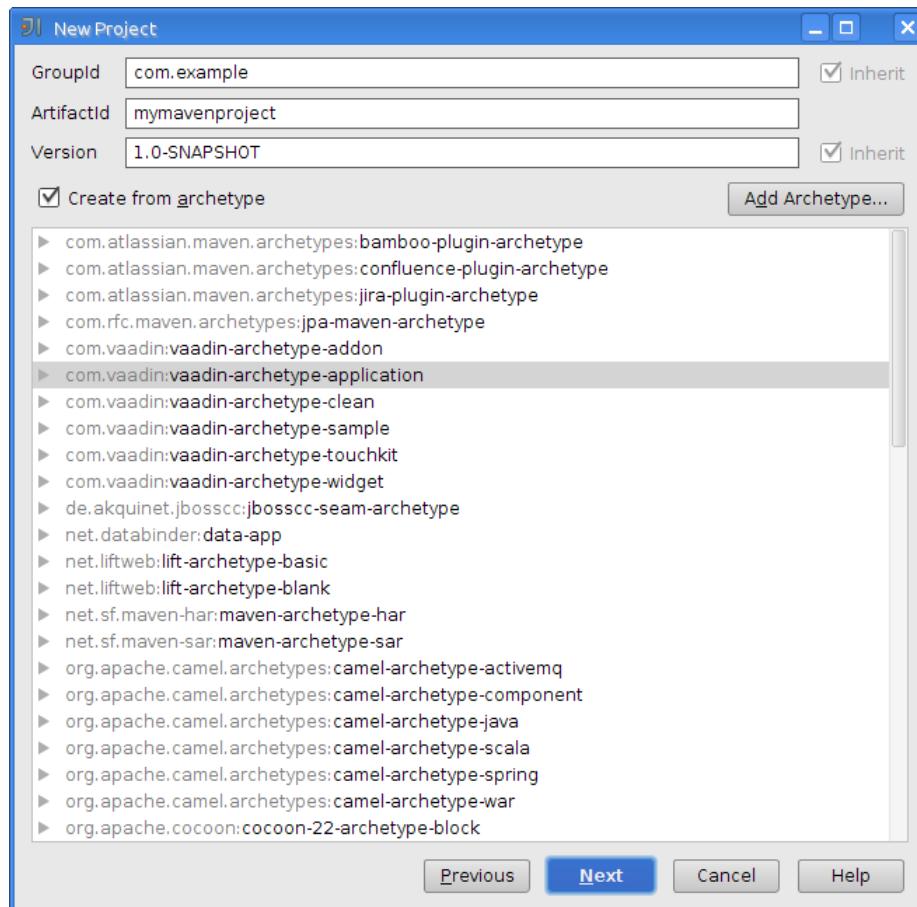


5. 选中 **Create from archetype** 选项
6. 如果选项列表中不包含 Vaadin archetype，点击 **Add archetype** 按钮，输入 **GroupId** com.vaadin, **ArtifactId** vaadin-archetype-application, 以及 **Version** LATEST (也可以指定某个具体的版本号)。



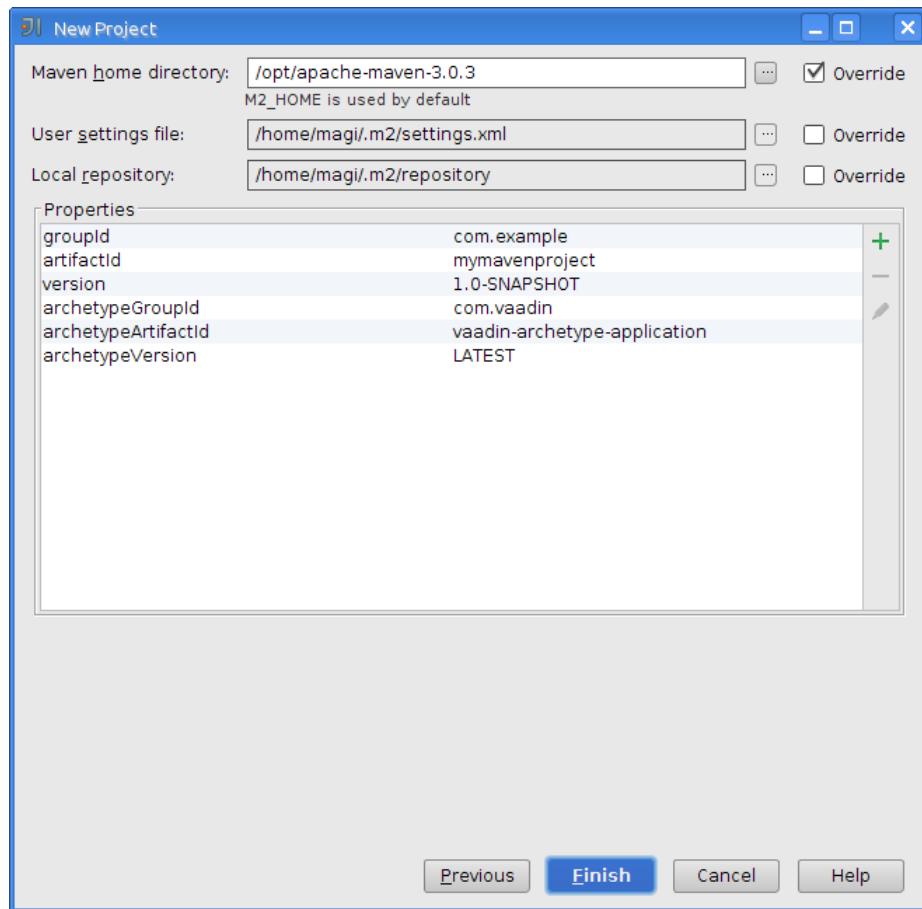
点击对话框的 **OK** 按钮。

7. 选择 com.vaadin:vaadin-archetype-application.



点击 **Next** 按钮.

8. 为你的新工程复查 Maven 的通用设定项, 以及工程专有的设定项. 你可能需要修改某些设定, 尤其是当你初次创建 Maven 工程时. 点击 **Finish** 按钮.



创建 Maven 工程将耗费一些时间, 因为 Maven 需要下载所有的依赖包. 完成之后, 工程将被创建出来, Maven POM 将在编辑器中打开.

编译工程

要使用 Maven 编译 Vaadin 应用程序, 你可以定义一个 run/debug 配置项来执行 Maven 目标 (goal), 比如 package, 来构建一个可发布的 WAR 包. 如果需要的话, 它还会编译 widget 群和 theme. 详情参见第 2.6.2 节“编译和运行应用程序”.

以下操作手顺介绍如何发布应用程序, 其中也包括如何编译应用程序.

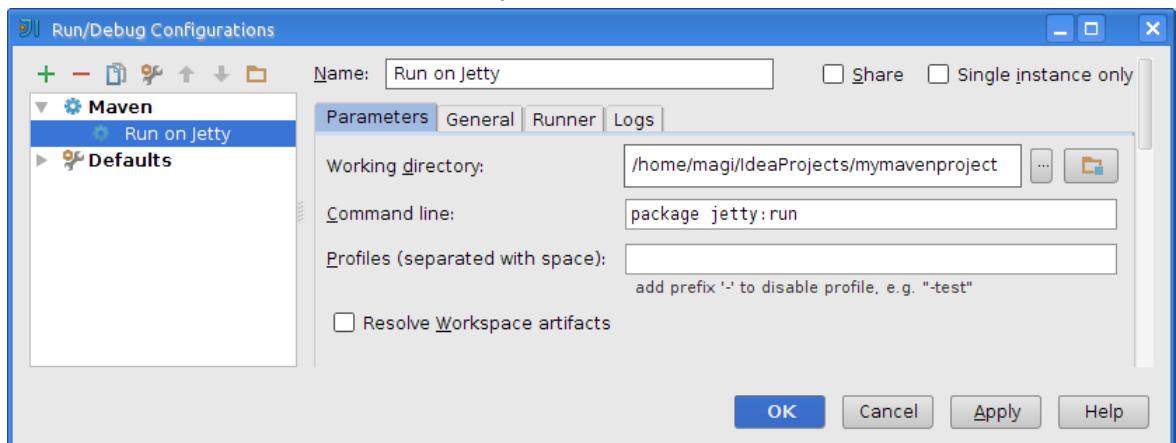
发布到服务器

有很多 Maven 插件, 可用于发布应用程序到各种类型的服务器. 比如, 要发布到 Apache Tomcat, 你可以配置 tomcat-maven-plugin, 然后执行 Maven 目标(goal) tomcat:deploy. 详情请参照插件本身的文档. 如果对某种特定的服务器类型, 没有可用的 Maven 插件, 你也可以通过某些低级别的方法发布你的应用程序, 比如, 执行某个 Ant task.

下面我们介绍如何创建一个 run/debug 配置项, 来构建一个 Vaadin Maven 应用程序, 并在轻量级 Jetty Web 服务器上发布和运行它.

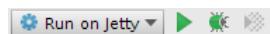
1. 选择 **Run → Edit Configurations**.
2. 选择 **+** → **Maven**, 创建新的 Maven run/debug 配置项.

3. 为 run 配置项输入 **Name**. 对于 **Command line**, 输入 "package jetty:run, 这个命令首先将编译并打包工程, 然后将启动 Jetty 来运行这个工程.



点击 **OK** 按钮.

4. 在工具条中选择 run 设定项, 并点击它旁边的 **Run** 按钮.



初次编译工程时将耗费一些时间, 因为需要编译 widget 群和 theme. 当运行结果的控制台输出面板中提示 Jetty 服务器启动完毕, 你就可以打开浏览器访问默认的URL:
<http://localhost:8080/>.

2.9. Vaadin 安装包

我们推荐的 Vaadin 安装方式是使用 Eclipse 插件, 或者使用其他 IDE 的插件, 或者使用包依赖管理系统, 比如 Maven, 但 Vaadin 也可以通过 ZIP 包的形式安装.

你可以通过下载页面得到最新的 Vaadin 安装包, 地址是 <http://vaadin.com/download/>. 请用你操作系统的 ZIP 解压缩工具将 ZIP 包中的文件解开.

2.9.1. 包中的内容

`README.TXT`

这是 Readme 文件, 简单介绍如何将 Vaadin 安装到你的工程中.

`release-notes.html`

发布说明文件, 其中的信息包括: 这个发布版的新功能特性, 升级方法指南, 与旧版本之间的兼容问题, 等等等等. 这个 HTML 请通过浏览器阅读.

`license.html`

Apache License version 2.0. 这个 HTML 请通过浏览器阅读.

`lib folder`

Vaadin 所依赖的全部库文件都包含在这个 lib 文件夹中.

`*.jar`

Vaadin 库, 详情参见 第 2.3 节 “Vaadin 库概述”.

2.9.2. 安装库文件

你可以通过以下几个简单的步骤来安装 Vaadin ZIP 包:

1. 将包内根路径下的 JAR 文件复制到工程的 Web 库文件夹 WEB-APP/lib 之下. 库文件中有一部分是可选的, 详情参见 第 2.3 节 “Vaadin 库概述”.
2. 将 lib 文件下的依赖 JAR 文件也复制到工程的 Web 库文件夹 WEB-APP/lib 之下.

在不同的开发环境中, 工程的组织结构会不同, 因此 WEB-APP/lib 文件夹的位置也会不同.

- 在 Eclipse Dynamic Web Application 工程中, 它的路径是: WebContent/WEB-INF/lib.
- 在 Maven 工程中, 它的路径是: src/main/webapp/WEB-INF/lib.

2.10. 在 Scala 中使用 Vaadin

在任何 JVM 兼容的语言中都可以使用 Vaadin, 比如 Scala 或 Groovy. 但是, 也存在一些与库文件及工程设定相关的注意事项. 本节将介绍在 Eclipse 中如何用 Scala 创建 UI, 用到的工具是 Scala IDE for Eclipse 和 Vaadin Plugin for Eclipse.

1. 安装 Scala IDE for Eclipse, 可以使用 Eclipse 更新站点, 也可以使用打包好的 Eclipse 发布包.
2. 打开一个已有的 Vaadin Java 工程, 或者也可以创建一个新的工程, 方法参见 第 2.5 节 “使用 Eclipse 创建和运行一个工程”. 你可以删除向导自动创建的 UI 类.
3. 在 Eclipse 窗口右上侧点击 perspective, 切换到 Scala 透视图(perspective).
4. 在 **Project Explorer** 中, 在 project 上点击鼠标右键, 选择菜单项 **Configure → Add Scala Nature**.
5. Web 应用程序将需要 scala-library.jar 添加到它的类路径中. 如果使用 Scala IDE, 你可以将这个 JAR 文件从你的 Eclipse 安装路径下的某个地方复制到 Web 应用程序的类路径中, Web 应用程序的类路径可以是工程下的 WebContent/WEB-INF/lib 文件夹, 或者是应用程序服务器的库文件路径. 如果 JAR 文件的复制操作是在 Eclipse 外部进行的, 你需要刷新工程信息才能在 Eclipse 内看到复制后的文件, 方法是选中工程, 然后按下快捷键 **F5**.

你也可以在 Ivy 或 Maven 中添加对 Scala 的依赖来得到 Scala 运行库, 但主要要将你依赖的 Scala 版本设置为 Scala IDE 使用的 Scala 版本一致.

现在可以用 Scala 创建 UI 类了, 方法如下:

```
@Theme("mytheme")
class MyScalaUI extends UI {
    override def init(request: VaadinRequest) = {
        val content: VerticalLayout = new VerticalLayout
        setContent(content)

        val label: Label = new Label("Hello, world!")
        content.addComponent(label)

        // Handle user interaction
        content.addComponent(new Button("Click Me!",
            new ClickListener {
```

```
override def buttonClick(event: ClickEvent) =  
    Notification.show("The time is " + new Date)  
}  
}  
}
```

当你按下快捷键 **Ctrl+Shift+O**, Eclipse 和 Scala IDE 将会自动导入 Vaadin 类.

你需要将 Scala 语言的 UI 类定义在 Servlet 类中(对于 Servlet 3.0 工程), 或者定义在 web.xml 部署描述文件中, 方法与 第 2.5.2 节“浏览一下工程结构”中介绍的 Java 语言 UI 类一样.

插件 Scaladin add-on 提供了一种更加符合 Scala 语言风格的 Vaadin API. 这个插件与 Vaadin 7 兼容的版本目前还在开发中.

3.1. 概述	59
3.2. 技术背景	61
3.3. 客户端引擎	63
3.4. 事件和监听器	64

在第1章简介中，我们简略的介绍了Vaadin的架构概要。本章将在技术层面上更深入的讲解这个问题。

3.1. 概述

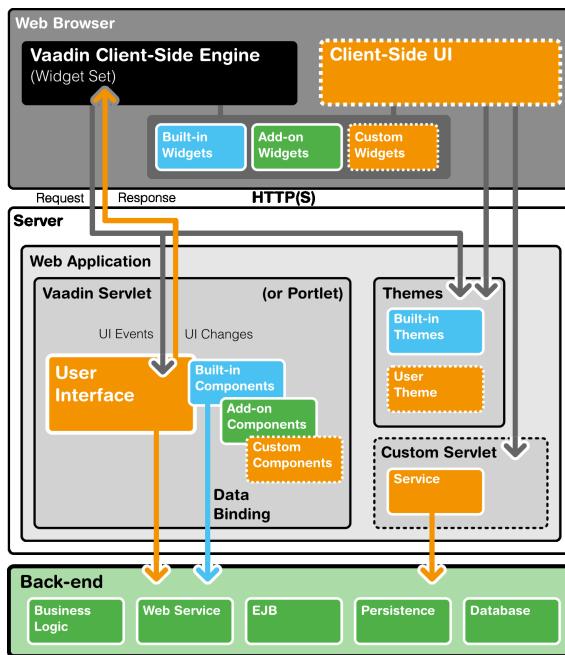
Vaadin为Web应用程序提供两种开发模式：客户端（浏览器端）和服务器端。服务器端开发模型功能更为强大一些，可以在服务器端开发完整的应用程序，使用基于AJAX的Vaadin客户端引擎将UI展现在浏览器中。客户端开发模型可以用Java语言开发widget或客户端应用程序，然后编译为JavaScript形式在浏览器内运行。两种开发模型可以共享UI widget, theme, 以及后端代码和服务，这两种开发模型也可以容易的结合在一起。

图3.1“Vaadin运行期架构”展示了客户端与服务器端通信的基本架构，图中展示的是，包含客户端代码（客户端引擎或应用程序）的页面在浏览器中初期装载后的运行场景。

Vaadin框架包括服务器端API, 客户端API, 一组UI组件和widget, UI组件和widget在两端都存在, theme用于控制具体的UI表现, 数据模型用于将服务器端组件直接绑定到数据上。对于客户端开发模型, 还包括VaadinCompiler, 可以将Java代码编译为JavaScript。

服务器端Vaadin应用程序以Servlet形式运行在Java Web服务器中，提供HTTP服务。Servlet类通常使用**VaadinServlet**。Servlet接受客户端请求，并将它解释为某个用户Session的事件。事件关联到UI组件上，并被派发给应用程序中指定的事件监听器。如果UI逻辑变更了服务器端UI组件的状态，Servlet会将UI组件的状态变化通过HTTP应答返回给Web浏览器端。客户端引擎

图 3.1. Vaadin 运行期架构



运行在浏览器内，它接受服务器端的应答，如果其中包含 UI 组件的状态变化，它将相应地修改浏览器内页面的状态。

服务器端开发架构包括的主要部分，及各部分的功能如下：

用户界面

Vaadin 应用程序向用户提供 UI，作为用户操作应用程序业务逻辑和数据的接口。从技术层面来说，UI 可以认为是 UI 类，继承自 `com.vaadin.ui.UI` 类。它的主要任务是使用 UI 构造出初始的用户界面，并创建各种事件监听器来处理用户的输入。UI 可以通过访问其 URL 来装载进浏览器，也可以嵌入到任意的 HTML 页面内。实现一个 UI 的详细方法，请参见第 4 章 编写服务器端 Web 应用程序。

请注意，“UI”这个词在本书中有时指一般的 UI 概念，也有时指 Vaadin 中 UI 类的概念。

UI 组件和 Widget

应用程序创建一系列组件，并管理其布局，由此构成了 Vaadin 应用程序的 UI。服务器端的每个组件在客户端都有一个对应的模块，也就是“widget”，组件将通过 widget 展现在浏览器中，用户也正是通过 widget 来与应用程序交互。客户端 widget 也可以供客户端应用程序使用。服务器端组件把它的事件转发给应用程序逻辑。拥有某种数据值供用户查看和编辑的 Field 组件，可以与数据源绑定在一起（详情见后文）。关于 UI 组件架构，更详细的介绍请参见第 5 章 UI 组件。

客户端引擎

Vaadin 的客户端引擎负责将 UI 展现在 Web 浏览器中，它将使用到各种客户端 widget，widget 将与服务器端组件一一对应。客户端引擎将用户的交互动作发送给服务器端，然后将服务器端 UI 的变化展现到浏览器端。客户端与服务器端的通信使用异步的 HTTP 或 HTTPS 请求。详情参见第 3.3 节“客户端引擎”。

Vaadin Servlet

客户端 Vaadin 应用程序工作在 Java Servlet API 之上（详情参见第 3.2.5 节“Java Servlet”）。Vaadin servlet，更确切的说 **VaadinServlet** 类，从不同的客户端接受请求，从

cookie 中取出 Session 信息, 判定请求属于哪个用户 Session, 然后将请求转发到它所属的 Session. 你可以继承 Vaadin servlet 来改造它.

Theme

Vaadin 将 UI 的组件结构与具体的表现分隔为不同的部分. UI 的逻辑由 Java 代码来处理, 界面表现则定义在 *theme* 中, theme 使用 CSS 或 Sass 形式. Vaadin 提供了很多默认 theme. 用户自定义的 theme 除样式表(style sheet)外, 还可以包含 HTML 模板, 用来定制 HTML 元素布局和其他资源, 比如图片, 字体. 关于 Theme 的详细介绍请参见第 7 章 Themes.

事件

用户与 UI 组件的交互将产生事件, 事件首先由客户端的 widget 处理, 然后依次传递给 HTTP 服务器, Vaadin Servlet, UI 组件, 最后到达应用程序中指定的事件监听器. 详情参见第 3.4 节“事件和监听器”.

服务器端 PUSH

除服务器端开发模式外, Vaadin 还支持服务器端 PUSH 功能, UI 的变化可以从服务器端直接 PUSH 到客户端, 而不必由客户端发起请求或发送事件来获得 UI 的更新信息. 这个功能使得我们可以从其他线程或其他 UI 中直接立即更新某个 UI, 而不必等待客户端发起请求之后才刷新浏览器端的 UI. 详情参见第 11.16 节“服务器端 PUSH”.

数据绑定

除 UI 模型外, Vaadin 还提供了 数据模型, 用于将 Field 组件(比如文本框, 复选框, 单选框)中展现的数据, 绑定到数据源上. 使用数据模型, UI 组件可以直接更新应用程序数据, 通常不必编写任何控制代码. Vaadin 中所有 Field 组件内部都使用这个数据模型, 当然它们也都可以任意绑定到不同类型的数据源. 比如使用 数据源, 你可以绑定一个 Table 组件到 SQL 查询结果上. Vaadin 数据模型的详细介绍, 请参见第 8 章 组件与数据绑定.

客户端应用程序

除服务器端 Web 应用程序外, Vaadin 还支持客户端应用程序模块的开发. 客户端模块将运行在浏览器内, 可以与服务器端应用程序一样, 使用相同的 widget, theme, 和后端服务. 客户端模块主要用于开发高流畅度的 UI 逻辑, 比如游戏客户端, 或高并发量客户端加无状态服务器端, 也可以用于其他场景, 比如为服务器端应用程序提供离线模式功能. 详情请参见第 14 章 客户端应用程序.

后端

Vaadin 的主要目的是构建 UI, 我们建议应用程序的其他逻辑层应该与 UI 层分离开. 业务逻辑可以与 UI 代码运行在同一个 Servlet 内, 但至少要以 Java API 的形式分离开, 或者使用 EJB 也可. 业务逻辑也可以以后端服务的方式分散到其他服务器上. 数据存储通常分散到数据库系统中, 然后某种持久化方案来进行访问, 比如使用 JPA.

3.2. 技术背景

本节介绍 Vaadin 背后的一些技术背景和设计思路. 这些知识对于 Vaadin 的使用并非必须的, 但如果你需要对 Vaadin 做一些底层扩充, 就非常有用.

3.2.1. HTML 和 JavaScript

World Wide Web 上所有的网站和大多数应用程序, 都是基于超文本标记语言 (HTML) 构建的. HTML 定义了网页的构造和格式, 还可以在网页中包含图像和其他资源. HTML 由元素(element)的阶层组成, 元素则由起始 tag 和结束 tag 来表示, 比如 <div> ... </div>. Vaadin 使用 HTML 5, 但 HTML 5 还没有被所有的浏览器完整地支持, 因此 Vaadin 很小心地只使用了 HTML 5 中主流浏览器最常用版本目前支持的部分.

JavaScript, 是一种嵌入到 HTML 页面中的编程语言. JavaScript 程序可以通过页面的文档对象模型 (DOM) 操纵 HTML 页面的内容. 也可以处理用户交互事件. Vaadin 的客户端引擎和客户端 widget 所做的工作正是如此, 虽然它们其实是用 Java 开发的, 但它们被 Vaadin Client Compiler 编译为 JavaScript 代码.

Vaadin 隐藏了 HTML 的使用细节, 因此你可以专注于 UI 组件的结构和控制逻辑. 在服务器端开发中, UI 用 Java 语言使用 UI 组件来开发, 然后被客户端引擎展现为 HTML 形式, 但也可以使用 HTML 模板来定义 HTML 元素的结构和格式. 开发客户端 widget 和 UI 时, Vaadin 内建的 widget 同样隐藏了操纵 HTML DOM 的大部分细节.

3.2.2. 使用 CSS 和 Sass 控制样式

HTML 定义了 Web 页面的内容和结构, 层叠样式表(CSS)则是一种语言用来定义页面的显示风格, 比如颜色, 文字大小, 留白尺寸. CSS 基于一组规则, 浏览器将这些规则匹配到 HTML 元素结构上. CSS 规则中的定义的各种属性决定了与规则匹配的 HTML 元素的视觉表现.

```
/* Define the color of labels in my view */
.myview .v-label {
    color: blue;
}
```

Sass, 全称 优良语法样式表(*Syntactically Awesome Stylesheets*), 是 CSS 语言的扩展, 允许使用变量, 嵌套, 以及其他各种语法特性, 使得对 CSS 的使用更加简便, 更加清晰. Sass 有两种格式, 一种是 SCSS 格式, 是 CSS3 语法规的超集, 另一种是 SASS 格式, 是更为简单一些的老式的缩进式语法, Vaadin 的 Sass 编译器支持的是 SCSS 语法.

Vaadin 管理页面元素样式时, 使用 CSS 或 Sass 定义的 *theme*, 以及与 theme 相关联的图像, 字体, 和其他资源. Vaadin 的 theme 使用 Sass 来书写. 开发模式下, Sass 文件会被自动编译为 CSS. 生产环境下, 你可以使用 Vaadin 附带的编译器将 Sass 文件编译为 CSS. theme 使用方法的详细说明请参见第 7 章 *Themes*, 这一章还介绍了 CSS 和 Sass.

3.2.3. AJAX

AJAX, 全称是异步 JavaScript 与 XML 技术(Asynchronous JavaScript and XML), 常用来开发具有高度交互性 UI 的 Web 应用程序, 这类 Web 应用程序的 UI 类似于传统的桌面应用程序. 传统的 Web 应用程序, 不论有没有 JavaScript 支持, 要从服务器得到新的页面内容时只能装载整个全新的页面. 带有 AJAX 支持的页面则不同, 它使用 JavaScript 来处理用户交互, 向服务器发送异步的请求(不会重装载整个页面), 通过服务器的应答得到更新后的内容, 然后根据新内容修改页面内相应部分. 通过这种方式, 只有页面中的一小部分数据需要重新装载. 上述目的是通过使用以下几种技术实现的: HTML, CSS, DOM, JavaScript, 以及 JavaScript 的 XMLHttpRequest API. XML 只是客户端与服务器端之间序列化数据的其中一种方法, 在 Vaadin 中, 数据的序列化使用更高效的 JSON 方式.

AJAX 使用的异步请求由 JavaScript 中的 XMLHttpRequest 类实现. 这个 API 在所有的主流浏览器中都是可用的, 而且它即将成为 W3C 标准.

在浏览器与服务器之间传递复杂数据需要将数据对象 序列化(serialization) (或者叫 汇集(marshalling)). Vaadin Servlet 和客户端引擎会将共享的状态对象从服务器端组件序列化到客户端, 同时也会处理 widget 与服务器端组件之间的远程过程调用(RPC)的序列化问题.

3.2.4. Google Web Toolkit

Vaadin 的客户端框架构建在 Google Web Toolkit (GWT) 的基础之上. 它的目的是帮助程序员使用 Java 语言便利地开发浏览器内运行的 Web UI, 而不是使用 JavaScript. 客户端模块用 Java 语言开发然后由 Vaadin Compiler 编译为 JavaScript, Vaadin Compiler 是 GWT Compiler 的扩展. 客

客户端框架还隐藏了很多 HTML DOM 操纵的细节, 还运行我们用 Java 语言来处理浏览器端的各种事件.

GWT 本质上是一种客户端技术, 通常用于开发 Web 浏览器端的 UI 逻辑. 纯客户端模块仍然需要与服务器通信, 方法是使用 RPC 调用, 并序列化需要的数据. Vaadin 的服务器端开发非常有效地隐藏了客户端和服务器端之间的所有通信细节, 并允许我们在服务器端应用程序内处理 UI 逻辑. 这就使得基于 AJAX 的 Web 应用程序的架构变得简单得多. 当然, Vaadin 仍然可以开发纯客户端应用程序, 详情参见第 14 章 客户端应用程序.

Vaadin 的客户端引擎是如何使用基于 GWT 的客户端框架的, 详情请参见第 3.3 节“客户端引擎”. 关于客户端开发, 请参见第 13 章 客户端 Vaadin 开发, 关于客户端 widget 与服务器端组件的集成, 请参见第 16 章 与客户端集成.

3.2.5. Java Servlet

Java Servlet 是一个类, 运行在 Java Web 服务器内(也叫 Servlet 容器), 用于为服务器扩展新的功能. 实际运用中, Java Servlet 通常是 Web 应用程序的一部分, Web 应用程序中可以包含 HTML 页面来提供静态内容, 还可以包含 JavaServer Page (JSP) 和 Java Servlet 来提供动态内容. 详情参见图 3.2 “Java Web 应用程序和 Servlet”.

Web 应用程序部署到服务器上之前, 通常需要打包为 WAR (*Web application ARchive*) 文件形式, WAR 也是 Java JAR 文件格式, JAR 文件实际上是一种 ZIP 压缩后的包. Web 应用程序通过部署描述文件 WEB-INF/web.xml 来定义, 部署描述文件定义了 Servlet 类, 以及客户端请求的 URL 路径与各个 Servlet 之间的对应关系. 详情参见第 4.8.4 节“使用部署描述文件 web.xml”. Servlet 以及它依赖的其他类的查找路径包括 WEB-INF/classes 和 WEB-INF/lib 文件夹. WEB-INF 是一个特殊的隐藏文件夹, 它将无法从外部以 URL 路径地方式访问.

Servlet 是 Java 类, 它处理服务器通过 Java Servlet API 传递给它的 HTTP 请求. Servlet 可以通过应答的方式产生 HTML 或其他类型的内容. JSP 页面则是 HTML 页面, 其中允许嵌入 Java 源代码. 实际上 Servlet 容器会将 JSP 转换为等价的 Java 源文件, 然后编译为 Servlet.

服务器端 Vaadin 应用程序的 UI 以 Servlet 的形式运行. UI 被包裹在 **VaadinServlet** Servlet 类之内, VaadinServlet 会处理 session 追踪以及其他一些任务. 在最初的客户端请求发生时, 它会返回一个 HTML 装载页, 以及大部分 JSON 应答, 用于同步 widget 及与 widget 对应的服务器端组件. 它还会对外提供各种资源, 比如 theme. 服务器端 UI 以从 **UI** 继承的类的形式实现, 详情参见第 4 章 编写服务器端 Web 应用程序. 在部署描述文件 web.xml 中, UI 类将以参数的形式传递给 Vaadin Servlet.

Vaadin 客户端引擎和客户端 Vaadin 应用程序都以静态 JavaScript 的形式装载到浏览器中. 客户端引擎, 用技术上术语的术语来说也可以叫做 Widget 群, 需要位于 Web 应用程序的 VAADIN/widgetsets 路径下. 预编译的默认 Widget 群位于 vaadin-client-compiled JAR 包中, 由 Vaadin Servlet 向外提供.

3.3. 客户端引擎

服务器端 Vaadin 应用程序的 UI 由 Vaadin 客户端引擎展现在浏览器中. 当包含 Vaadin UI 的页面在浏览器中打开时, Vaadin 客户端引擎就被装载进浏览器了. 服务器端 UI 组件在客户端的展现由 widget(与 Google Web Toolkit 中的术语一样) 负责. 客户端引擎的架构参见图 3.3 “Vaadin 客户端引擎”.

图 3.2. Java Web 应用程序和 Servlet

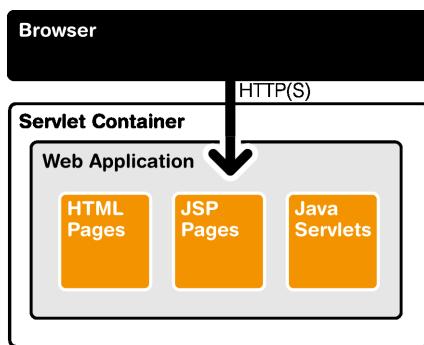
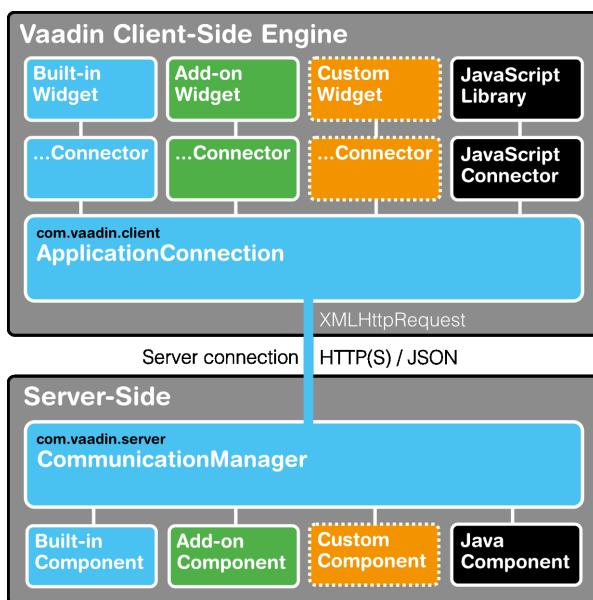


图 3.3. Vaadin 客户端引擎



客户端框架包含两种内建的 Widget: GWT widget 和 Vaadin 特有的 widget. 两种 widget 的内容有很大的重叠, 但 Vaadin widget 提供的功能与 GWT widget 略有不同. 此外, 还存在大量的 widget 插件及其服务器端对用组件, 你可以很容易地下载并安装它们, 详情参见 第 17 章 使用 Vaadin Add-on. 你也可以开发自己的 widget, 详情参见 第 13 章 客户端 Vaadin 开发.

通过 Widget 展现 UI 组件, 以及与服务器端通信, 由 **ApplicationConnection** 类负责处理. 由 connector 负责将 Widget 与它对应的服务器端组件连接在一起, 对每一个拥有服务器端对应组件的客户端 widget 来说, 都存在一个 connector. 框架负责组件状态数据的序列化, 并且在客户端与服务器端两端都包含 RPC 机制. 如何集成 Widget 与其服务器端对应组件, 详情参见 第 16 章 与客户端集成.

3.4. 事件和监听器

Vaadin 提供了一种事件驱动的编程模型, 用于处理用户交互. 当用户在 UI 中执行了某种操作, 比如点击按钮, 或选择一个数据项, 应用程序需要意识到这种用户操作. 很多基于 Java 的 UI 开发框架都遵循 事件/监听器设计模式(也叫 观察者(Observer)设计模式), 用于将用户操作与应用程序逻辑之间的沟通. Vaadin 也使用同样的思路. 这种设计模式包含两个元素: 一个对象负责产生(或者叫

"激发"或"发起")事件,以及复数个监听器负责监听事件。当事件发生时,对象负责向所有的监听器发送事件通知。对于一个事件,通常情况下只有一个监听器。

事件可以用来实现很多种目的。在 Vaadin 中,事件通常用于处理用户在 UI 中的操作。Session 管理可能需要用到特别的事件,比如超时事件,这种情况下事件其实与用户操作无关。超时是定时事件的一种特殊情况,定时事件是指在某个特定的日期/时刻发生的事件,或者某段特定长度的时间流逝之后发生的事件。

要接受某种特定类型的事件,应用程序必须向事件的发生源注册监听器对象。监听器使用 `add*Listener()` 方法注册到组件上(方法名与具体的监听器类型一一对应)。

大多数与事件有关的组件都定义了它们独有的事件类,以及与事件对应的监听器类。比如, the **Button** 的事件是 **Button.ClickEvent**,这个事件由 **Button.ClickListener** 监听器负责监听。

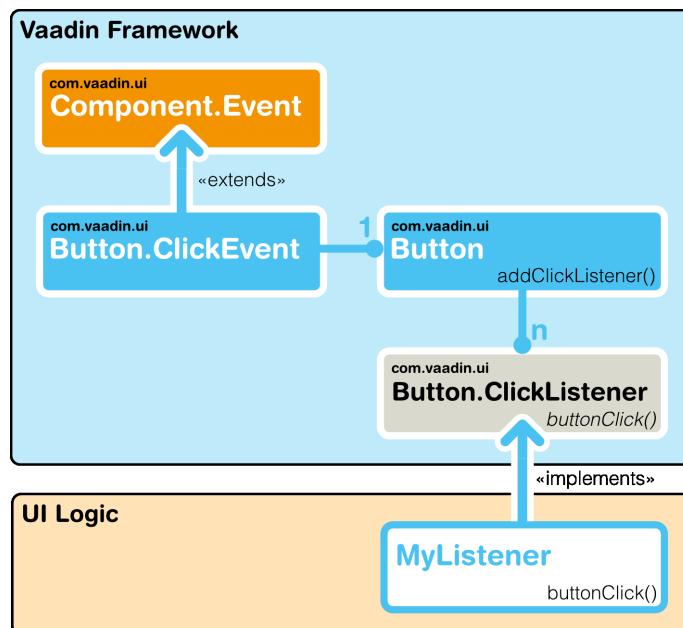
下面,我们使用一个由匿名类(anonymous class)实现的监听器来处理按钮的点击事件:

```
final Button button = new Button("Push it!");

button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        button.setCaption("You pushed it!");
    }
});
```

图 3.4 “按钮点击事件监听器的类结构图”展示了[一个应用程序专有的类继承自 **Button.ClickListener** 接口,监听按钮的点击事件。应用程序必须创建监听器类的示例,然后使用 `addClickListener\(\)` 方法注册它。监听器可以是匿名类,就像上面的例子那样。当事件发生时,事件对象将被初始化好,在上例中是 **Button.ClickEvent** 对象。事件对象知道与自己相关的 UI 组件是谁,在上例中是 **Button** 对象。](#)

图 3.4. 按钮点击事件监听器的类结构图



今天的监听器的大部分功能,在古老的 C 编程时代,由回调函数(callback function)实现。在面向对象语言中,我们通常只有类和方法可用,而没有函数的概念,因此应用程序必须向开发框架提供一个类接口,而不再是回调函数指针。

第 4.3 节“使用监听器来处理事件”详细讲解事件处理的更多实例.

Web

4.1. 概述	67
4.2. 构建 UI	70
4.3. 使用监听器来处理事件	74
4.4. 图片及其他资源	76
4.5. 错误处理	80
4.6. 通知	82
4.7. 应用程序的生命周期	85
4.8. 发布应用程序	90

本章概要介绍如何进行 Vaadin 服务器端 Web 应用程序开发，主要从实践角度介绍应用程序的基本元素。

4.1. 概述

服务器端 Vaadin 应用程序以 Java Servlet 形式运行在 Servlet 容器内。但 Java Servlet API 已被隐藏在开发框架之后。应用程序的 UI 由 UI 类实现，它需要创建并管理构成用户界面的那些 UI 组件。用户输入由事件监听器负责处理，当然也可以将 UI 组件直接绑定到数据上。应用程序的外观风格由 CSS 和 SCSS 形式的 theme 来定义。图标、其他图像，以及可被下载的文件以资源的形式管理起来，资源可以是外部资源，也可以由应用程序服务器或应用程序本身向外提供。

图 4.1. 服务器端应用程序架构

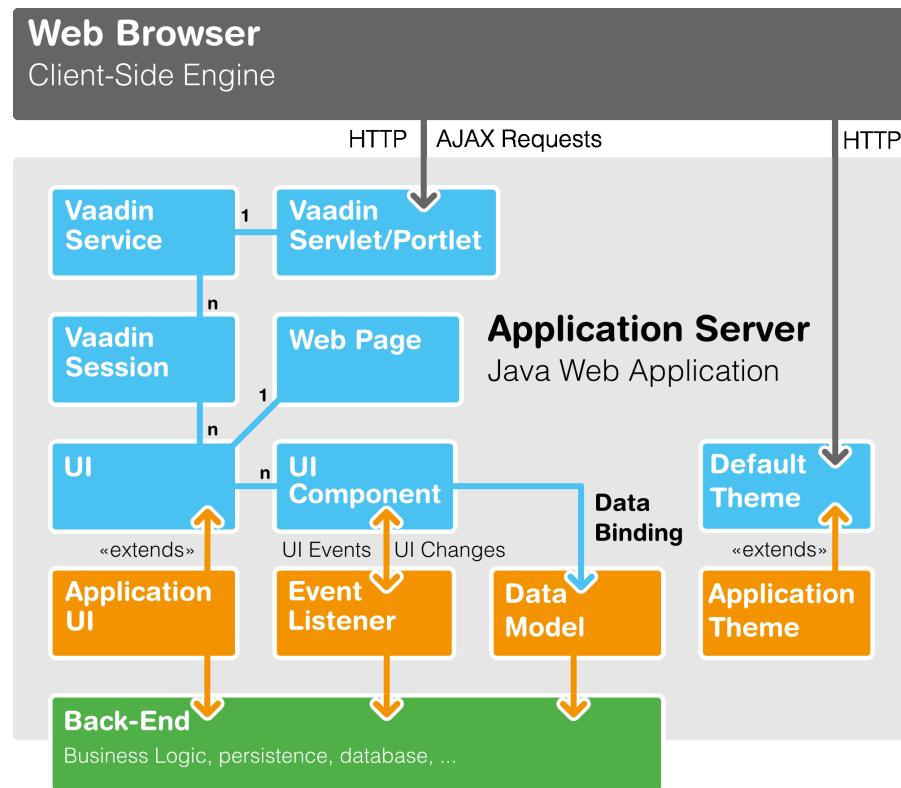


图 4.1 “服务器端应用程序架构”展示了用 Vaadin 框架开发的应用程序的基本架构，其中包含所有的主要元素，详情将在本章后续部分中进行介绍。

首先，Vaadin 应用程序必须包含一个或多个 UI 类，UI 类继承自抽象类 `com.vaadin.ui.UI`，并实现 `init()` 方法。可以用 Java 注解的方式为 UI 指定自定义的 theme。

```
@Theme("hellotheme")
public class HelloWorld extends UI {
    protected void init(VaadinRequest request) {
        ... initialization code goes here ...
    }
}
```

UI 是 Vaadin 应用程序运行在 Web 页面内时展现给用户的可见部分。Web 页面内可以包含多个这样的 UI。对于 portal 中的 portlet，这样的情况是很典型的。应用程序可以在浏览器的多个窗口中运行，每个窗口都拥有一个单独的 UI 实例。应用程序的 UI 实例可以是相同的 UI 类，也可以是不同的 UI 类。

Vaadin 框架内部会处理 Servlet 请求，并将请求与相应的用户 Session 和 UI 状态关联起来。因此，你可以用类似于桌面应用程序的方式来开发 Vaadin 应用程序。

应用程序的初始化部分最重要的任务是创建初期 UI。UI 的创建，以及将 UI 以 Java Servlet 的形式发布到 Servlet 容器中，是开发一个应用程序所需要的最低要求。关于 UI 的发布，详情参见第 4.8 节“发布应用程序”。

除 UI 外，应用程序还包括以下基本一些元素：

UI

一个 **UI** 表现为 Web 页面内的一个 HTML 片段, Vaadin 应用程序运行就运行在这个片段内。UI 通常占满整个页面, 但也可以只是页面的一部分。开发一个 Vaadin 应用程序的方法, 通常是扩展 **UI** 类, 并向其中添加内容。UI 本质上来说是一个可视区域, 与应用程序的用户 Session 相关联, 在应用程序中你可以拥有很多这样的可视区域, 尤其是多窗口应用程序的情况。一般来说, 当用户使用 Vaadin UI 的 URL 打开一个新页面时, 一个新的 **UI** 实例(以及与它关联的 **Page** 对象)将会自动创建。对同一个客户端来说, 所有的 UI 实例都共享同一个用户 Session。

当前的 UI 对象, 在任何地方都可以通过 `UI.getCurrent()` 方法得到。这个静态方法针对当前处理中的请求, 返回一个 UI 的 thread-local 实例, (详情参见 第 11.15.3 节“ThreadLocal 模式”。

Page

一个 **UI** 对象总是关联到一个 **Page** 对象上, **Page** 对象表示 UI 运行时所属的 Web 页面和浏览器窗口。

在 Vaadin 应用程序内, 当前处理中的请求所关联的 **Page** 对象, 在任何地方都可以通过 `Page.getCurrent()` 方法取得。这个方法等价于调用 `UI.getCurrent().getPage()` 方法。

Vaadin Session

一个 **VaadinSession** 对象代表应用程序中的一个用户 Session, 其中有一个或多个 UI 正在使用中。当用户初次打开 Vaadin 应用程序的 UI 时 Session 开始, 当服务器端 Session 超期时, Session 关闭, Session 也可以显式关闭。

UI 组件

UI 由应用程序创建的组件构成。组件的层级布局由特别的 布局组件 管理, 层级的最高层是内容的根(content root)。用户对组件的操作会激发与这个组件相关的事件, 事件由应用程序处理。Field 组件用于输入数据, 它可以通过 Vaadin 数据模型直接绑定到数据上。你可以用继承或组合的方式编写你自己的 UI 组件。关于各种 UI 组件的详细介绍, 请参见第 5 章 UI 组件, 关于布局组件, 请参见第 6 章 布局管理, 关于组件的组合, 请参见第 5.28 节“使用 **CustomComponent** 创造复合组件”。

事件与监听器

Vaadin 遵循事件驱动编程范式, 在这种编程范式中, 事件和负责处理事件的监听器, 是用来处理应用程序中用户操作的基本方法。(当然, 服务器端 PUSH 也是另一种可能的手段, 详情参见 第 11.16 节“服务器端 PUSH”) 第 3.4 节“事件和监听器”从技术层面介绍事件和监听器, 本章后续部分的 第 4.3 节“使用监听器来处理事件”则从更侧重实践的角度进行介绍。

资源

UI 可以显示图片, 也可以包含 Web 页面链接, 或可供下载的文档链接。这些都以 *resource* 的形式进行处理, 资源可以是外部的, 也可以由 Web 服务器或应用程序本身提供。第 4.4 节“图片及其他资源”概要介绍各种不同类型的资源。

Theme

UI 的外观表现与它的控制逻辑是分离的。UI 的逻辑由 Java 代码控制, 外观表现则由 CSS 或 SCSS 形式的 *theme* 来定义。Vaadin 包含了一些内建的 theme。用户自定义的 theme, 除样式表外, 还可以包含用于定义 HTML 元素布局的 HTML 模板, 以及其他 theme 资源, 比如图片。Theme 的详细介绍参见 第 7 章 Themes, HTML 元素布局参见第 6.14 节“自定义布局”, theme 资源参见 第 4.4.4 节“Theme 资源”。

数据绑定

Field 组件本质上来说就是数据的外部展现, 数据是通过 Vaadin 数据模型表达的. 使用这个数据模型, 组件可以从模型直接得到数据值, 也可以将用户输入直接更新到模型中, 而不必编写任何控制代码. 单个 Field 组件总是绑定到 *property* 上, 一组 Field 总是绑定到一个管理 *property* 的 *item* 上. Item 由 *container* 管理, 对某些组件来说, 比如 table 或 list, container 可以看作一个数据源. 所有的组件都有一个默认的数据模型, 它们也可以绑定到用户定义的数据源. 比如, 你可以绑定一个 **Table** 组件到一个 SQL 查询结果上. 关于 Vaadin 中数据绑定的完整介绍, 请参见 第 8 章 组件与数据绑定.

4.2. 构建 UI

Vaadin UI 是由多个组件的层级构成的, 因此下层组件被包含在布局组件或其他某种组件容器之内. 构建组件层级结构首先从最顶层开始(从最底层开始也可以 - 取决于你考虑这个问题的方式), 最顶层就是应用程序的 **UI** 类. 通常我们将一个布局组件设置为 UI 的内容, 并将其他组件填充到布局组件内.

```
public class MyHierarchicalUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // The root of the component hierarchy
        VerticalLayout content = new VerticalLayout();
        content.setSizeFull(); // Use entire window
        setContent(content); // Attach to the UI

        // Add some component
        content.addComponent(new Label("Hello!"));

        // Layout inside layout
        HorizontalLayout hor = new HorizontalLayout();
        hor.setSizeFull(); // Use all available space

        // Couple of horizontally laid out components
        Tree tree = new Tree("My Tree",
            TreeExample.createTreeContent());
        hor.addComponent(tree);

        Table table = new Table("My Table",
            TableExample.generateContent());
        table.setSizeFull();
        hor.addComponent(table);
        hor.setExpandRatio(table, 1); // Expand to fill

        content.addComponent(hor);
        content.setExpandRatio(hor, 1); // Expand to fill
    }
}
```

组件层级结构类似于树的形式, 如下:

```
UI
`-- VerticalLayout
  |-- Label
`-- HorizontalLayout
  |-- Tree
  '-- Table
```

以上代码的运行结果见 图 4.2 “简单的层级式 UI”.

图 4.2. 简单的层级式 UI

The screenshot shows a Vaadin application interface. On the left, there is a tree view with the following structure:

- My Tree
 - +5 Quarterstaff (blessed)
 - +3 Elven Dagger (blessed)
 - +5 Helmet (greased)
 - ▶ Sack
 - ▶ Bag of Holding
 - ▶ Chest

On the right, there is a table view titled "My Table" with the following data:

name	city	year
Charles Darwin	Oxford	1 9
Charles Adams	London	1 8
Isaac Lovelace	Oxford	1 9
Charles Newton	Oxford	1 8
Charles Adams	London	1 8

Vaadin 的内建组件参见 第 5 章 UI 组件, 布局组件参见 第 6 章 布局管理.

上面给出的示例程序还是都做不了. 用户操作由事件监听器处理, 详情参见 第 4.3 节 “使用监听器来处理事件”.

4.2.1. 应用程序架构

一旦你的应用程序增长到数十行的程度(通常很快就会如此), 你就需要更仔细地考虑应用程序架构的问题了. 你可以自由地使用 Java 中的任何面向对象技术, 将你的代码组织为方法, 类, 包, 以及库的形式. 应用程序架构定义了模块之间如何通信以及各模块间的相互依赖关系, 还定义了应用程序关注的范围. 在本书中, 我们只略微提及 Vaadin 应用程序中常见的一些架构模式.

后续的各节将介绍一个基本的应用程序模式. 关于常见的应用程序架构, 详情请参见第 11.10 节 “应用程序高级架构”, 其中将介绍多层架构, Model-View-Presenter (MVP) 模式, 等等. 第 11.15 节 “访问 Session 全局数据” 讨论如何传递全局数据的问题, 这个问题在 第 4.2.4 节 “访问 UI, Page, Session, 以及 Service” 中也有部分讨论.

4.2.2. 组合组件

UI 通常包含多个 UI 组件, 这些组件以某种布局层级结构组织起来. Vaadin 提供了很多布局组件来管理组件, 包括垂直布局, 水平布局, 网格布局, 以及其他各种方式的布局. 你可以扩展布局组件来创建组合组件.

```
class MyView extends VerticalLayout {
    TextField entry = new TextField("Enter this");
    Label display = new Label("See this");
    Button click = new Button("Click This");
```

```

public MyView() {
    addComponent(entry);
    addComponent(display);
    addComponent(click);

    // Configure it a bit
    setSizeFull();
    addStyleName("myview");
}
}

// Use it
Layout myview = new MyView();

```

这种组合模式尤其适合于创建 Form, 详情参见 第 8.4.3 节“绑定成员 Field”.

为实现组件的组合, 象上面的例子那样, 直接从布局组件继承是一种很简单的方式. 但是, 在实践中我们可能期望将具体的实现细节隐藏起来, 比如, 具体使用了什么样的布局组件可能是我们不希望使用者知道的细节. 上例中使用的组合方式实际上已经依赖到具体的实现细节, 这就使得将来的代码变更变得比较困难. 为了隐藏组合组件的内部实现细节, Vaadin 提供了一个专门的 **CustomComponent** 封装类, 它隐藏了其内容的具体表达方式.

```

class MyView extends CustomComponent {
    TextField entry = new TextField("Enter this");
    Label display = new Label("See this");
    Button click = new Button("Click This");

    public MyView() {
        Layout layout = new VerticalLayout();

        layout.addComponent(entry);
        layout.addComponent(display);
        layout.addComponent(click);

        setCompositionRoot(layout);

        setSizeFull();
    }
}

// Use it
MyView myview = new MyView();

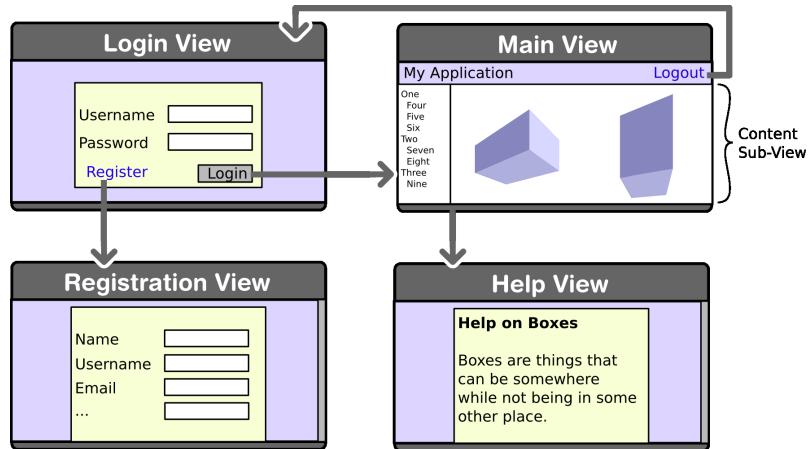
```

CustomComponent 的更详细介绍, 请参见 第 5.28 节“使用 **CustomComponent** 创造复合组件”. Vaadin Plugin for Eclipse 中包含了一个可视化编辑器, 可用来设计组合组件, 详情参见 第 10 章在 Eclipse 中进行可视化的 UI 设计.

4.2.3. 视图导航

简单应用程序大多只有单个 视图 (或者叫 画面), 但一般的应用程序通常会有很多个视图. 即使在单个视图呢, 你也可能希望拥有多个子视图, 比如, 为不同的场景显示不同的内容. 图 4.3 “在多个视图之间跳转”展示了在应用程序的多个顶级视图之间跳转, 以及在主视图内包含多个子视图的情况.

图 4.3. 在多个视图之间跳转



第 11.9 节“应用程序内的导航跳转”中介绍的 **Navigator** 类，是视图的管理器，提供了灵活的手段实现在多个视图和子视图之间跳转，还可以管理页面 URL 内的 URI 片段，利用这些 URI 片段，可以将视图添加为浏览器书签，链接，或在浏览器访问历史中回退。

Vaadin 应用程序的视图通常不是独立存在的，而是某个更大更复杂页面的一部分。这种情况下，你可能需要将 Vaadin 应用程序集成到别的网站中。你可以使用嵌入技术，详情参见 第 11.2 节“在 Web 页面中嵌入 UI”。

4.2.4. 访问 UI, Page, Session, 以及 Service

你可以通过 `getUI()` 和 `getPage()` 方法，得到某个组件所关联的 UI 和 page。

但是，在组件关联到 UI 之前，上述方法的返回值将是 `null`，通常当你在类的构造方法中需要访问以上方式时，组件是未关联到 UI 的。因此，更好的方法是在应用程序的任何地方，在 **UI**, **Page**, **VaadinSession**, 以及 **VaadinService** 类上使用静态方法 `getCurrent()`，得到当前的 UI, page, session，以及 service 对象。

```
// Set the default locale of the UI
UI.getCurrent().setLocale(new Locale("en"));

// Set the page title (window or tab caption)
Page.getCurrent().setTitle("My Page");

// Set a session attribute
VaadinSession.getCurrent().setAttribute("myattrib", "hello");

// Access the HTTP service parameters
File baseDir = VaadinService.getCurrent().getBaseDirectory();
```

page 和 session 对象也可以通过 **UI** 的 `getPage()` 和 `getSession()` 方法得到，service 对象可通过 **VaadinSession** 的 `getService()` 方法得到。

上述静态方法使用这些类内建的 ThreadLocal 支持。ThreadLocal 模式的详细介绍请参见第 11.15.3 节“ThreadLocal 模式”。

4.3. 使用监听器来处理事件

我们在第3.4节“事件和监听器”中已经学习了事件处理的基本原理，下面来进行一点实战。监听器可以在常规类中实现，但这种方式带来一个问题，就是在监听器中难以区分事件来源于哪个组件。大多数情况下我们推荐使用匿名类来实现监听器。

4.3.1. 使用匿名类

在Java 6和7中，最简单最常用的事件处理方法是使用匿名的局部类。这种方法将事件的处理封装在组件定义的相同位置，而且可以省去管理类，实现接口之类的麻烦。下面的例子定义了一个匿名类，它继承自**Button.ClickListener**接口。

```
// Have a component that fires click events
final Button button = new Button("Click Me!");

// Handle the events with an anonymous class
button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        button.setCaption("You made me click!");
    }
});
```

被匿名类访问的局部对象，比如上例中的**Button**，必须声明为final。

大多数组件都可以在构造函数中以参数的形式传入事件监听器，因此可以减少1、2行代码。但要注意，如果在匿名类内部访问当前正在创建中的组件，你必须在构造函数执行之前就声明好这个组件的引用，比如，可以使用外部类中的成员变量。如果组件的声明语句与构造函数的执行语句是在同一个表达式之内，那么构造函数执行时，组件的声明其实还并不存在。这时，在匿名类内，你就需要通过event参数得到组件的引用。

```
final Button button = new Button("Click It!",
    new Button.ClickListener() {
        @Override
        public void buttonClick(ClickEvent event) {
            event.getButton().setCaption("Done!");
        }
});
```

4.3.2. Java 8 中的事件处理

Java 8引入了一个新特性：lambda 表达式，它可以用来替代事件监听器。如果监听器内只有一个方法需要重写，那么你可以直接使用lambda 表达式来替代监听器。

比如下例中，我们在构造函数中使用lambda 表达式，来处理按钮的点击事件：

```
layout.addComponent(new Button("Click Me!",
    event -> event.getButton().setCaption("You made click!")));
```

Java 8代表了Java 的最新发展方向，而且，由于Vaadin API 中广泛使用了事件监听器，因此，使用lambda 表达式将使得UI 代码更加易读。

使用方法引用(method reference)，可以很便利地将事件转发给监听器方法：

```
public class Java8Buttons extends CustomComponent {
    public Java8Buttons() {
        setCompositionRoot(new HorizontalLayout(
            new Button("OK", this::ok),
```

```

        new Button("Cancel", this::cancel));
    }

    public void ok(ClickEvent event) {
        event.getButton().setCaption ("OK!");
    }

    public void cancel(ClickEvent event) {
        event.getButton().setCaption ("Not OK!");
    }
}

```

4.3.3. 使用常规类来实现监听器

下面的例子遵循经典开发模式, 你有一个 **Button** 组件, 还有一个监听器负责处理用户操作(也就是点击), 用户操作以事件的形式发送给应用程序. 这里我们定义一个类来监听点击事件.

```

public class MyComposite extends CustomComponent
    implements Button.ClickListener {
    Button button; // Defined here for access

    public MyComposite() {
        Layout layout = new HorizontalLayout();

        // Just a single component in this composition
        button = new Button("Do not push this");
        button.addClickListener(this);
        layout.addComponent(button);

        setCompositionRoot(layout);
    }

    // The listener method implementation
    public void buttonClick(ClickEvent event) {
        button.setCaption("Do not push this again");
    }
}

```

4.3.4. 区分事件的发生源

如果应用程序从多个事件源接收到相同类型的多个事件, 比如有存在按钮的情况, 应用程序必须能够区分各个事件来源于哪个组件. 如果使用常规类来实现监听器, 可以将事件来源与所有组件逐个比较, 以此来区分事件来源组件. 判断事件源的方法根据事件类型而不同.

```

public class TheButtons extends CustomComponent
    implements Button.ClickListener {
    Button onebutton;
    Button toobutton;

    public TheButtons() {
        onebutton = new Button("Button One", this);
        toobutton = new Button("A Button Too", this);

        // Put them in some layout
        Layout root = new HorizontalLayout();
        root.addComponent(onebutton);
        root.addComponent(toobutton);
        setCompositionRoot(root);
    }
}

```

```

    }

    @Override
    public void buttonClick(ClickEvent event) {
        // Differentiate targets by event source
        if (event.getButton() == onebutton)
            onebutton.setCaption ("Pushed one");
        else if (event.getButton() == toobutton)
            toobutton.setCaption ("Pushed too");
    }
}

```

区分不同的事件源还有其他方法，比如使用对象的属性，名称，或标签文字来区分组件。使用标签文字或者其他可见文字的方式通常来说是不好的，因为这种方法在应用程序国际化时会发生问题。使用其他符号化字符串也是危险的，因为这些字符串内容只有在实际运行时才会被检查。

4.4. 图片及其他资源

Web 应用程序可以显示各种 资源，比如图片，其他嵌入式内容，或可下载的文档，这些资源都由浏览器从服务器端取得。图片资源一般使用 **Image** 组件来显示，或者显示为某个组件的图标。Flash 动画可以使用 **Flash** 组件显示，内嵌的浏览器 frame 可以用 **BrowserFrame** 组件显示，其他内容可以用 **Embedded** 组件显示，详情参见第 5.30 节“内嵌资源”。可下载的文件通常以 **Link** 的方式提供，点击链接即可下载文件。

Web 服务器向外提供资源的方式有很多种。静态资源可以直接向外提供，不必通过应用程序。对于动态资源的情况，应用程序必须动态地创建其内容。Vaadin 的资源请求接口既允许应用程序访问静态资源，也允许应用程序动态创建资源。动态创建资源需要用到 **StreamResource** 和 **RequestHandler** 类，详情请参见第 11.4 节“请求处理器(Request Handler)”。

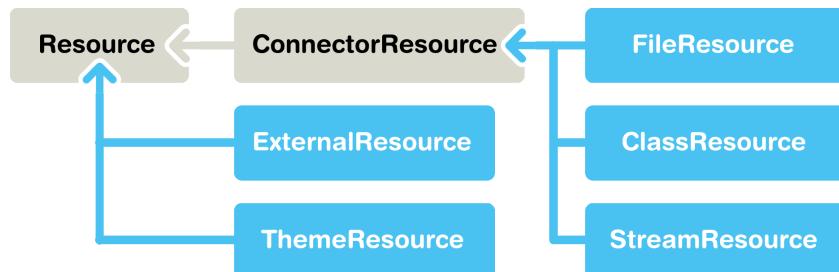
Vaadin 还提供了低级工具来取得 HTTP 请求的 URI 和其他参数。我们首先看看应用程序如何提供各种类型的资源，然后再介绍用于对外提供资源和其他功能，与 URI 和 HTTP 参数处理相关的低级接口。

注意，对于 Vaadin 或一般的 AJAX 应用程序来说，使用请求处理器来创建“页面”通常是什么意义的。关于这个问题的详细解释，请参见第 3.2.3 节“AJAX”。

4.4.1. 资源接口和类

Vaadin 中的资源类属于两类接口：一个是较为通用的 **Resource** 接口，另一个是较为特殊化的 **ConnectorResource** 接口，**ConnectorResource** 用于通过 Servlet 向外提供的资源。

图 4.4. 资源接口和类的关系图



4.4.2. 文件资源

文件资源是指存储在文件系统中任意位置的文件。因此，文件资源不能直接通过服务器上某个特定的 URL 来访问，而必须经过 Vaadin Servlet 来请求。对于没有打包进 Web 应用程序的用户持久化数据，通常就需要使用文件资源来存取。

文件对象使用标准的 **java.io.File** 类来定义，它可以作为文件资源来访问。创建文件时，你可以使用绝对路径，也可以使用相对路径，但相对路径所使用的起始路径将根据 Web 服务器的安装状况而不同。比如，对于 Apache Tomcat，默认的当前目录将是 Tomcat 的安装路径。

下面的例子中，我们通过存储在 Web 应用程序中的文件来向外提供一个图片资源。注意，图片存储在 WEB-INF 文件夹之下，这个目录是一个特殊目录，它与 Web 应用程序中的其他目录不同，不可以使用 URL 来访问其中的内容。这是保证系统安全性的一种方案，另一种方案是将资源存储在文件系统的其他位置，而不是 Web 应用程序之内。

```
// Find the application directory
String basepath = VaadinService.getCurrent()
    .getBaseDirectory().getAbsolutePath();

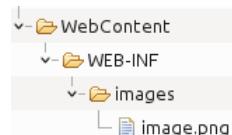
// Image as a file resource
FileResource resource = new FileResource(new File(basepath +
    "/WEB-INF/images/image.png"));

// Show the image in the application
Image image = new Image("Image from file", resource);

// Let the user view the file in browser or download it
Link link = new Link("Link to the image file", resource);
```

上面例子的运行结果，以及在通常的 Eclipse Vaadin 工程中存储的文件结构，参见图 4.5 “文件资源”。

图 4.5. 文件资源



4.4.3. Class Loader 资源

ClassResource 可以使用 Java Class Loader 从类路径中装载资源。通常，使用的类路径是 Web 应用程序中的 WEB-INF/classes 文件夹，这里就是 Java 编译器生成 Java 类文件，并从源代码树中复制其他资源文件时的输出位置。

下面的示例从应用程序包中装载一个图片资源，并显示在一个 **Image** 组件中。

```
layout.addComponent(new Image(null,
    new ClassResource("smiley.jpg")));
```

4.4.4. Theme 资源

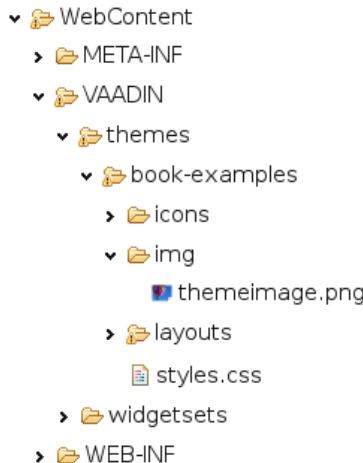
ThemeResource 类是 Theme 资源，它是 Theme 内的文件，通常是图片。Theme 位于 Web 应用程序的 VAADIN/themes/themename 目录下。Theme 资源的名字作为参数传递给构造函数，其内容是资源文件相对于 theme 文件夹的相对路径。

```
// A theme resource in the current theme ("mytheme")
// Located in: VAADIN/themes/mytheme/img/themeimage.png
ThemeResource resource = new ThemeResource("img/themeimage.png");

// Use the resource
Image image = new Image("My Theme Image", resource);
```

上面示例的运行结果，参见图 4.6 “Theme 资源”，此图还展示了 Eclipse 工程中 Theme 资源文件的目录结构。

图 4.6. Theme 资源



为了使用 Theme 资源，你需要为 UI 设置 theme。关于 Theme 的更多介绍，详情请参见 第 7 章 *Themes*。

4.4.5. 流资源

流资源可用来创建动态的资源内容。图表是动态图片的典型例子。要生成一个流资源，你需要实现 **StreamResource.StreamSource** 接口，及其中的 `getStream()` 方法。这个方法需要返回一个 **InputStream** 对象，从这个 `InputStream` 中应该能够读取到资源的内容。

下面的例子演示如何创建一个简单的 PNG 格式图片。

```
import java.awt.image.*;

public class MyImageSource
    implements StreamResource.StreamSource {
    ByteArrayOutputStream imagebuffer = null;
    int reloads = 0;

    /* We need to implement this method that returns
     * the resource as a stream. */
    public InputStream getStream () {
        /* Create an image and draw something on it. */
        BufferedImage image = new BufferedImage (200, 200,
                                                BufferedImage.TYPE_INT_RGB);
        Graphics drawable = image.getGraphics();
        drawable.setColor(Color.lightGray);
        drawable.fillRect(0,0,200,200);
        drawable.setColor(Color.yellow);
        drawable.fillOval(25,25,150,150);
```

```

        drawable.setColor(Color.blue);
        drawable.drawRect(0,0,199,199);
        drawable.setColor(Color.black);
        drawable.drawString("Reloads="+reloads, 75, 100);
        reloads++;

        try {
            /* Write the image to a buffer. */
            imagebuffer = new ByteArrayOutputStream();
            ImageIO.write(image, "png", imagebuffer);

            /* Return a stream from the buffer. */
            return new ByteArrayInputStream(
                imagebuffer.toByteArray());
        } catch (IOException e) {
            return null;
        }
    }
}

```

上面例子生成的图片内容是动态的，因为每次访问它，它都会更新自己的访问次数。**ImageIO.write()** 方法将图片内容写入到 output 流中，但我们需要返回 input 流，所以我们将图片内容保存到一个临时缓冲区中。

下面我们使用 **Image** 组件来显示图片。

```

// Create an instance of our stream source.
StreamResource.StreamSource imagesource = new MyImageSource();

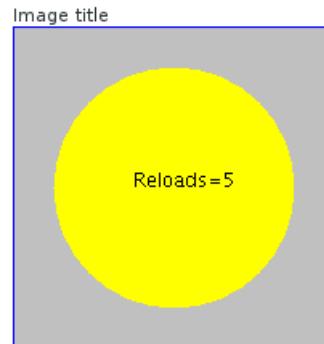
// Create a resource that uses the stream source and give it a name.
// The constructor will automatically register the resource in
// the application.
StreamResource resource =
    new StreamResource(imagesource, "myimage.png");

// Create an image component that gets its contents
// from the resource.
layout.addComponent(new Image("Image title", resource));

```

上面示例的运行结果见图 4.7 “流资源”。

图 4.7. 流资源



创建动态内容的另一种方法是使用请求处理器，详情请参见 第 11.4 节 “请求处理器(Request Handler)”。

4.5. 错误处理

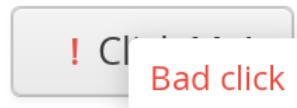
4.5.1. 错误指示器和消息

所有的组件都带有内建的错误指示器，当某个组件的输入校验发生错误时，错误指示器将被激活，错误指示器还可以使用 `setComponentError()` 方法显式地激活。错误指示器通常位于组件标题的右侧。错误指示器是组件标题的一部分，因此它的位置通常由组件所属的布局管理组件来控制，但某些组件会自行控制错误指示器。将鼠标指针移动到组件上方，会显示它的错误消息。

```
textfield.setComponentError(new UserError("Bad value"));
button.setComponentError(new UserError("Bad click"));
```

上述示例程序的运行结果见图 4.8 “激活时的错误指示器”。

图 4.8. 激活时的错误指示器



4.5.2. 自定义系统消息

系统消息是一种通知，它指出应用程序处于严重的不正常状态，通常需要重启应用程序。最典型的例子就是 Session 超时。

系统消息由 **SystemMessages** 类管理的字符串，。

`sessionExpired`

应用程序 Servlet 的 Session 过期。如果在指定的 Session 超期时限内没有发生对服务器的请求，Session 就会过期。Session 超期时限可以通过 `web.xml` 文件中的 `session-timeout` 参数进行配置，详情参见第 4.8.4 节“使用部署描述文件 `web.xml`”。

`communicationError`

Vaadin 客户端引擎与应用程序服务器之间发生了不明的通信错误。原因可能是服务器已经无法使用，或者发生了其他问题。

`authenticationError`

当向服务器的某个请求得到 401 (Unauthorized) 应答时，将发生这个错误。

`internalError`

严重的内部错误，原因可能是 Vaadin 客户端引擎本身的 bug，也可能是自定义的客户端代码的 bug。

`outOfSync`

客户端状态与服务器端状态不一致。

`cookiesDisabled`

这个错误告诉用户，浏览器禁用了 cookie，应用程序无法在没有 cookie 的状态下工作。

每一条消息都有 4 个属性：1，短标题，2，完整的消息内容，3，显示完消息之后应该跳转去的 URL 地址，4，是否允许通知。

详细信息(英文)可能会输出到调试窗口，调试窗口参见第 11.3 节“Debug 模式和 Debug 窗口”。

你可以替换掉默认的系统消息，方法是在 **VaadinService** 中设置 **SystemMessagesProvider**. 你需要实现 **getSystemMessages()** 方法，这个方法需要返回 **SystemMessages** 对象. 定制消息的最简单方法是使用 **CustomizedSystemMessages** 对象.

在自定义 Servlet 类的 **servletInitialized()** 方法中，你可以设置 system message provider，示例如下：

```
getService().setSystemMessagesProvider(
    new SystemMessagesProvider() {
        @Override
        public SystemMessages getSystemMessages(
            SystemMessagesInfo systemMessagesInfo) {
            CustomizedSystemMessages messages =
                new CustomizedSystemMessages();
            messages.setCommunicationErrorCaption("Comm Err");
            messages.setCommunicationErrorMessage("This is bad.");
            messages.setCommunicationErrorNotificationEnabled(true);
            messages.setCommunicationErrorURL("http://vaadin.com/");
            return messages;
        }
    });
});
```

定制 Vaadin Servlet 的方法，请参见 第 4.7.2 节 “Vaadin Servlet, Portlet, 和 Service”

4.5.3. 管理未被 catch 的例外

事件处理可能导致在应用程序逻辑中，甚至框架本身中，发生例外，但这些例外有可能没有被应用程序正确地 catch 住. 任何一个未被应用程序 catch 的例外最终都会被框架 catch 住. 框架将例外转发给 **DefaultErrorHandler** 类，这个类将错误消息显示为组件错误，也就是，在组件上显示一个小小的红色 "!" 符号 (具体表现取决于 theme). 如果用户将鼠标指针移动到这个符号之上，例外的全部 backtrace 信息将被显示在一个大的 tooltip 框中，参见 图 4.9 “未 catch 的例外被显示在组件的错误指示器中”.

图 4.9. 未 catch 的例外被显示在组件的错误指示器中



你可以改变默认的错误处理方式，方法是实现自定义的 **ErrorHandler**，并在组件层级中的任何组件上通过 **setErrorHandler()** 方法来激活它，也可以在 **UI** 或 **VaadinSession** 对象上使用你的自定义错误处理器. 你可以实现 **ErrorHandler** 接口，也可以从 **DefaultErrorHandler** 类继承. 下面的例子中，我们修改默认的错误处理器的行为.

```
// Here's some code that produces an uncaught exception
final VerticalLayout layout = new VerticalLayout();
final Button button = new Button("Click Me!",
    new Button.ClickListener() {
        public void buttonClick(ClickEvent event) {
```

```

        ((String)null).length(); // Null-pointer exception
    }
});

layout.addComponent(button);

// Configure the error handler for the UI
UI.getCurrent().setErrorHandler(new DefaultErrorHandler() {
    @Override
    public void error(com.vaadin.server.ErrorEvent event) {
        // Find the final cause
        String cause = "<b>The click failed because:</b><br/>";
        for (Throwable t = event.getThrowable(); t != null;
             t = t.getCause())
            if (t.getCause() == null) // We're at final cause
                cause += t.getClass().getName() + "<br/>";

        // Display the error message in a custom fashion
        layout.addComponent(new Label(cause, ContentMode.HTML));
    }

});
}
);

```

上面的例子还演示了如何从例外的 cause stack 中找出例外发生的最初原因.

当从 **DefaultErrorHandler** 类继承时, 你可以象上面的例子一样, 调用 `doDefault()` 方法, 来执行默认的错误处理动作, 比如为发生例外的组件设置错误信息. 具体细节请查看源代码. 你可以调用 `findAbstractComponent(event)` 方法, 来找出导致错误的组件. 如果错误没有关联到某个组件, 这个方法的返回值将为 `null`.

4.6. 通知

通知是在界面上短暂显示的错误信息或提示信息, 一般显示在屏幕的正中. 通知框有一个标题, 还有可选的描述信息和图标. 通知框在屏幕上停留预先指定的时间后消失, 或者用户点击它也可以让它消失. 通知的类型决定了一个通知的默认表现和行为.

有两种方法创建通知. 最简单的方法是使用静态方法 `Notification.show()`, 这个方法的参数是通知的标题, 以及可选的通知描述信息和通知类型, 使用这个方法可以将通知显示在当前页面中.

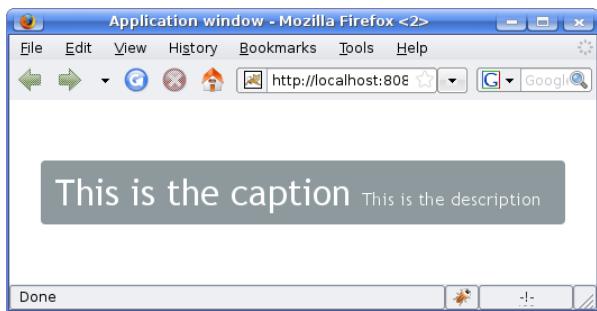
```
Notification.show("This is the caption",
                  "This is the description",
                  Notification.Type.WARNING_MESSAGE);
```

为了对通知进行更精确地控制, 你可以创建一个 **Notification** 类. 这个类的构造方法存在多个不同版本, 各构造方法的参数可以有标题, 可选的描述信息, 通知类型, HTML 是否可用. 通知显示在 **Page** 中, 通常就是当前页面.

```
new Notification("This is a warning",
                 "<br/>This is the <i>last</i> warning",
                 Notification.TYPE_WARNING_MESSAGE, true)
                 .show(Page.getCurrent());
```

标题和描述信息默认显示在同一行中. 如果你希望在它们之间显示一个换行符, 在 HTML 可用的情况下请用 HTML 的换行标记 "`
`", HTML 禁用时可以使用 "`\n`". HTML 默认是禁用的, 但可以通过 `setHtmlContentAllowed(true)` 来允许使用 HTML. 当 HTML 允许使用时, 你可以在通知的标题和描述信息中使用任意的 HTML 标签. 如果你的通知内容有可能来自用户的输入, 你应该

图 4.10. 通知



禁用 HTML, 否则的话, 应该仔细处理通知内容, 以防发生 HTML 注入, 详情参见 第 11.8.1 节“处理用户输入内容, 防止跨站脚本攻击”.

图 4.11. 使用 HTML 格式化的通知



4.6.1. 通知类型

通知类型决定了一个通知的默认 style 和默认行为. 如果没有指定通知类型, 默认会使用 "humanized" 类型. 如下所示的各通知类型, 声明在 **Notification.Type** 类中.

TYPE_HUMANIZED_MESSAGE 一种用户友好的消息, 不会过度干扰用户: 它不要求用户点击来确认它, 而且会快速消失. 它显示为灰色, 位于屏幕中间.	Humanized message For minimal annoyance
TYPE_WARNING_MESSAGE 警告, 是中等重要度的消息. 它显示为既不太黑白, 也不过于引人注目的彩色. 警告显示 1.5 秒, 但用户可以点击警告框, 立即关闭它. 即使在警告显示时, 用户也可以继续操作应用程序.	Warning message For notifications of medium importance
TYPE_ERROR_MESSAGE 错误消息是通知的一种, 它要求用户的高度注意, 它显示为警报的颜色, 而且要求用户点击错误消息框才能关闭它. 错误消息框本身并没有指示用户去点击消息, 不过在它的右上角会显示一个关闭框. 与其他类型的通知不同, 当错误消息显示时, 用户不可以操作应用程序.	Error message For important notifications



TYPE_TRAY_NOTIFICATION
托盘通知显示在"系统托盘"区域,也就是,浏览器的右下角区域.由于托盘通知通常不会遮挡任何用户界面,因此它们的显示时间比"humanized"类型或警告类型要长,默认为3秒.托盘通知显示时,用户可以继续正常操作应用程序.

4.6.2. 定制通知

上述各通知类型的一切功能,都可以通过 **Notification** 的属性来精确控制.通知设置完成之后,你还需要将它显示在当前页面中.

```
// Notification with default settings for a warning
Notification notif = new Notification(
    "Warning",
    "<br/>Area of reindeer husbandry",
    Notification.TYPE_WARNING_MESSAGE);

// Customize it
notif.setDelayMsec(20000);
notif.setPosition(Position.BOTTOM_RIGHT);
notif.setStyleName("mystyle");
notif.setIcon(new ThemeResource("img/reindeer.png"));

// Show it in the page
notif.show(Page.getCurrent());
```

setPosition() 方法可用来设置通知的显示位置.位置可以由枚举型 **Position** 中定义的常数来指定.

setDelayMSec() 方法可用来设置通知的显示时间,单位是毫秒.参数值 -1 表示消息将永远显示,直到用户点击消息框它才会消失.这个参数值还会禁止用户操作应用程序的其他部分,这也是错误型通知的默认行为.但它不会象错误型通知那样,在消息框上增加一个关闭框.

4.6.3. 使用 CSS 来控制样式

```
.v-Notification {}
.popupContent {}
.gwt-HTML {}
h1 {}
p {}
```

通知消息框是一个浮动的 `div` 元素,位于页面的 `body` 元素之下.它的最顶层样式是 `v-Notification`.通知的内容被包装在一个元素内,这个元素的样式是 `popupContent`.标题包含在 `h1` 元素内,描述信息则在 `p` 元素内.

想要定制通知的外观表现,首先可以通过 `setStyleName("mystyle")` 方法为 **Notification** 对象添加一个样式,然后在 theme 内设置这个样式的外观表现,例子如下:

```
.v-Notification.mystyle {
    background: #FFFF00;
    border: 10px solid #C00000;
    color: black;
}
```

上述示例的运行结果见图 4.12 “自定义样式的通知”，图中显示的图标，是由更前面的“定制通知”中的示例设定的。

图 4.12. 自定义样式的通知



4.7. 应用程序的生命周期

本节中，我们从技术细节的层面详细讨论应用程序的发布，用户 Session, UI 实例的生命周期等等问题。对于 Vaadin 应用程序开发来说，这些技术细节并不是必须的知识，但有助于理解 Vaadin 应用程序究竟是如何工作的，尤其是可以帮助理解应用程序会在什么情况下结束运行。

4.7.1. 发布

Vaadin 应用程序可供用户使用之前，必须发布到 Java Web 服务器上，详情请参见 第 4.8 节“发布应用程序”。发布过程将读取 Servlet 类设定，这个设定可以通过 @WebServlet 注解来指定（Servlet 3.0 环境下），也可以使用应用程序内的部署描述文件 web.xml 来指定（Servlet 2.4 环境下），得到 Servlet 类设定信息后，发布过程会为特定的 URL 路径注册 Servlet，并装载 Java 类。到此为止发布过程还不会运行应用程序内的任何代码，不过 Java 类被装载时，其中的 static 代码段会被执行。

卸载和重发布

应用程序的卸载发生在以下3种情况：1，服务器停止运行，2，应用程序重发布，3，被管理者明确地卸载。卸载一个服务器端 Vaadin 应用程序将终止它的运行，应用程序内的所有 Java 类将被卸载，应用程序分配的全部堆空间(heap space)将被释放，未来会被 Java 机器的垃圾收集器回收。

如果此时存在活动的用户 Session，UI 的客户端状态将被挂起，下次发生向服务器的请求时，将出现一个同步失败的错误消息。

重发布与序列化

某些服务器，比如 Tomcat，支持 热部署，这种情况下类可以被重新装载，同时保持应用程序的内存状态不变。这种功能是通过以下手段实现的：1，首先将应用程序状态序列化，2，重新装载 Java 类后，再反序列化，恢复原来的应用程序状态。如果你使用基本的 Eclipse 加 Tomcat 配置，而且 UI 被标记了 @PreserveOnRefresh 注解，这个功能就会生效。如果你希望强制应用程序重启动，你需要在 URL 中加上 ?restartApplication 参数才可以。JRebel 之类的工具的功能更为强大，它可以在适当的时刻重新装载 Java 类的代码，而不需要使用序列化/反序列化手段。服务器关闭和重启时也可以序列化应用程序状态，这样在服务器重启时就可以保持住 Session 信息了。

序列化功能需要应用程序是 可序列化的，也就是说，所有的类必须实现 Serializable 接口。Vaadin 的所有类都已实现了这个接口。如果你需要扩展 Vaadin 类，或实现接口，你可以给定一个可选的序列化 key，如果你使用 Eclipse 的话，这个 key 可以自动生成。序列化也被用于集群计算和云计算环境，比如 Google App Engine，详情请参见 第 11.7 节“与 Google App Engine 的集成”。

4.7.2. Vaadin Servlet, Portlet, 和 Service

VaadinServlet 类, 或 portal 中的 **VaadinPortlet** 类, 接受所有向服务器的请求, 请求通过 URL 映射到这些类, 映射关系由发布设定来配置, 然后 VaadinServlet 或 VaadinPortlet 将请求关联到 Session 上. Sessions 再将请求关联到某个特定的 UI.

当收到请求时, Vaadin Servlet 或 portlet 通过 **VaadinService** 来处理 Servlet 和 portlet 共有的任务. 这个类负责管理 Session, 提供对部署设定信息的访问能力, 管理系统消息, 并负责其他很多任务. 与 Servlet 或 portlet 相关的更多特别处理, 由 **VaadinServletService** 类或 **VaadinPortletService** 类来处理. Service 以主要低阶定制层的形式处理客户端请求.

定制 Vaadin Servlet

在 Servlet 类中需要完成很多共通的配置任务, 如果你使用 Servlet 3.0 环境的 @WebServlet 注解来部署你的应用程序, 你的配置任务其实已经完成了. 你可以重载 `servletInitialized()` 方法来实现大部分的定制任务, 在这个方法中可以通过 `getService()` 方法得到 **VaadinService** 对象(这个对象在构造方法中是无法得到的). 在你的代码中, 首先必须调用 `super.serveletInitialized()` 方法.

```
public class MyServlet extends VaadinServlet {
    @Override
    protected void servletInitialized()
        throws ServletException {
        super.serveletInitialized();

        ...
    }
}
```

在处理客户端请求的过程中, 想要追加某种定制的功能的话, 你可以重载 `service()` 方法.

在 Servlet 2.4 环境下, 想要使用定制的 Servlet 类的话, 你需要在部署描述文件 `web.xml` 中指定使用你的 Servlet 类, 而不是通常的 **VaadinServlet** 类, 详情请参见 第 4.8.4 节“使用部署描述文件 `web.xml`”.

定制 Vaadin Portlet

本节未完成

定制 Vaadin Service

想要定制 **VaadinService**, 你首先需要继承 **VaadinServlet** 或 **Portlet** 类, 重载 `createServletService()` 方法, 然后创建定制的 Service 对象.

4.7.3. 用户 Session

当用户打开某个**UI** 的 URL, 初次访问一个 Vaadin servlet 或 portlet 时, 用户 Session 就开始活动了. 属于某个 UI 类的全部服务器请求都由 **VaadinServlet** 或 **VaadinPortlet** 类来处理. 当一个新的客户端连接到应用程序时, 它会创建出新的用户 Session, **VaadinSession** 类的一个实例就代表一个用户 Session. Session 的追踪是使用浏览器端保存的 cookie 来实现的.

你可以使用 **UI** 类的 `getSession()` 方法得到它的 **VaadinSession**, 或者通过全局方法 `VaadinSession.getCurrent()` 也可以. 通过 **WrappedSession** 还可以访问到低阶 Session 对象, 也就是 `HttpSession` 和 `PortletSession`. 通过 **VaadinSession** 你还可以访问到部署配置信息, 详情请参见 第 4.8.7 节“发布配置”.

当 **UI** 实例过期或被关闭时, 用户 Session 就会结束. 详情参见本节后述内容.

处理 **Session** 的初始化和消灭

你可以处理 Session 的初期化和消灭, 方法是分别实现 `SessionInitListener` 和 `SessionDestroyListener` 接口, 然后将这两个监听器追加到 **VaadinService** 中. 实现以上任务最简单的方法是扩展 **VaadinServlet** 类, 重载它的 `servletInitialized()` 方法, 详情请参见 第 4.7.2 节 “Vaadin Servlet, Portlet, 和 Service”.

```
public class MyServlet extends VaadinServlet
    implements SessionInitListener, SessionDestroyListener {

    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();
        getService().addSessionInitListener(this);
        getService().addSessionDestroyListener(this);
    }

    @Override
    public void sessionInit(SessionInitEvent event)
        throws ServiceException {
        // Do session start stuff here
    }

    @Override
    public void sessionDestroy(SessionDestroyEvent event) {
        // Do session end stuff here
    }
}
```

如果使用的是 Servlet 2.4, 你需要将部署描述文件 `web.xml` 中的 `servlet-class` 参数设置为你的自定义的Servlet 类, 而不是通常的 **VaadinServlet** 类, 详情请参见 第 4.8.4 节 “使用部署描述文件 `web.xml`”.

4.7.4. 装载 UI

当浏览器初次访问某个 UI 的 URL 时, Vaadin Servlet 会生成一个装载页面. 这个页面会装载客户端引擎(widget set), 客户端引擎再向 Vaadin Servlet 发起一个独立的请求来装载 UI.

当客户端引擎发起它的初次请求时, **UI** 类的一个实例会被创建出来. Servlet 使用 **UIProvider** 来创建 UI 实例, **UIProvider** 则被注册在 **VaadinSession** 实例中. 一个 Session 至少拥有一个 **DefaultUIProvider** 来管理用户打开的 UI. 如果应用程序允许用户使用 **BrowserWindowOpener** 打开弹出式窗口, 那么每个弹出式窗口都将拥有独自的 UI Provider.

新的 UI 实例创建出来之后, 它的 `init()` 方法将被调用. 这个方法将收到 **VaadinRequest** 类代表的请求对象.

定制装载页面

装载页面的 HTML 内容是以 HTML DOM 对象方式生成的, 其内容是可以定制的, 方法是实现一个 `BootstrapListener`, 在这个监听器中修改 DOM 对象. 为了这个目的, 你需要扩展 **VaadinServlet** 类, 向 Service 对象追加一个 `SessionInitListener`, 详情请参见 第 4.7.3 节 “用户 Session”. 然后在 Session 初期化时, 通过 `addBootstrapListener()` 方法向 Session 追加一个 `BootstrapListener`.

Widget 群通过装载页面来装载，负责处理这个任务的函数定义在单独的 `vaadinBootstrap.js` 脚本文件中。

你也可以使用完全自定义的装载代码，比如一个静态的 HTML 页面，详情请参照 第 11.2 节“在 Web 页面中嵌入 UI”。

定制 UI Provider

你可以根据请求参数(比如 URL 路径)来动态地创建 UI 对象，方法是定义一个定制的 `UIProvider`。你需要将定制的 UI Provider 添加到 `Session` 对象中，`Session` 对象负责调用 `UI Provider`。`UI Provider` 是串联工作的，因此最后添加的那个 `UI Provider` 将会第一个被调用，如果串联起来的多个 `UI Provider` 中有一个返回了 `UI`，那么它之后的 `UI Provider` 就不会被调用，相反，如果某个 `UI Provider` 返回了 `null`，那么它之后的 `UI Provider` 会被继续调用。将 `UI Provider` 添加到 `Session` 对象中，最简便的方法是实现一个定制的 `Servlet`，然后在 `SessionInitListener` 监听器中，将 `UI Provider` 添加到 `Session` 中。

关于定制 `UI provider` 的例子，请参见 第 20.8.1 节“提供一个备用(Fallback)UI”。

阻止 UI 刷新

在浏览器中重新装载页面通常会产生一个新的 `UI` 实例，而旧的 `UI` 将会残留在服务器中，直到一段时间后才会被清除。这样的结果可能是不理想的，因为对使用者来说，`UI` 的原有状态被清除了。为了保护 `UI` 不被刷新，你可以对 `UI` 类使用 `@PreserveOnRefresh` 注解。你也可以使用一个 `UIProvider`，定制它的 `isUiPreserved()` 方法。

```
@PreserveOnRefresh
public class MyUI extends UI {
```

在 URL 中添加 `?restartApplication` 参数，会告诉 Vaadin Servlet 在装载页面时创建新的 `UI` 实例，因此也就推翻了 `@PreserveOnRefresh` 注解的设定。Eclipse 在重发布应用程序时通常会保持住应用程序状态，而你修改代码后往往希望重启 `UI`，因此在 Eclipse 中开发 `UI` 时常常有必要使用这个参数。如果你的 URL 中含有 URI 片段，这个参数需要出现在 URI 片段之前。

4.7.5. UI 过期

如果在一定时间内没有收到来自于 `UI` 的通信，`UI` 实例会被清除。当没有向服务器发送其他请求时，客户端会定期发送心跳请求，以便保持 `UI` 处于活动状态。只要有来自 `UI` 的请求，或心跳信号，`UI` 就会一直保持激活。如果连续3次心跳信号没有收到，那么这个 `UI` 将会过期。

心跳信号的发生间隔时间是 5 分钟，这个间隔时间可以通过 `Servlet` 的 `heartbeatInterval` 参数来修改。你可以在 `@VaadinServletConfiguration` 类中配置这个参数，也可以在 `web.xml` 文件中配置它，详情请参见 第 4.8.6 节“`Servlet` 的其他配置参数”。

当 `UI` 被清除时，`DetachEvent` 事件会被发送给这个 `UI` 上的所有 `DetachListener` 监听器。当 `UI` 被从 `Session` 中断开时，它的 `detach()` 方法会被调用。

4.7.6. 显式地关闭 UI

你可以使用 `close()` 方法，显式地关闭一个 `UI`。这个方法会在当前请求处理完毕之后，将 `UI` 标记为与 `session` 脱离。因此，这个方法不会使 `UI` 实例立即失效，当前请求的应答也可以正常发回给客户端。

解除一个 `UI` 不会关闭 `UI` 所在的浏览器端页面或窗口，但此后向服务器发起的请求都会导致错误。通常，你可能希望关闭窗口，或者重新载入页面，或者跳转到其他 URL。如果这个页面是一个通常的浏览器窗口或 tab 页，浏览器一般不会允许通过程序来关闭它，但跳转到其他地址是允许的。你可

以使用 `setLocation()` 方法将窗口重定向到另一个 URL, 比如 第 4.7.8 节“关闭 Session”中的示例程序中就是如此. 对于弹出窗口, 你可以调用 JavaScript `close()` 函数来关闭它, 详情请参见第 11.1.2 节“关闭弹出式窗口”.

如果你关闭与当前请求关联的 UI 实例之外的其他 UI 实例, 那么这个 UI 实例不会在当前请求结束时解除, 而是在它自己发起的下一次请求结束之后才解除. 你可以让 UI 心跳间隔更快一些, 使得 UI 的解除尽快发生, 也可以使用服务器端 PUSH 来让 UI 的解除立即发生.

4.7.7. Session 过期

当用户在应用程序中进行操作时, 就会发生对服务器的请求, 同时 UI 自身还会发送心跳请求, 这两种请求将会保持 Session 激活. 当所有 UI 都失效时, Session 仍会保持住, 当 Web 应用程序的 Session 过期时间到达之后, Session 才会过期, 然后被服务器清除.

如果应用程序中还存在激活的 UI, 它们的心跳请求将保持 Session 永远激活. 当用户长期不操作应用程序时, 你可能会希望 Session 超时, 这也正是 Session 超时设定本来的意图. 如果在 `web.xml` 文件中, Servlet 的 `closeIdleSessions` 参数被设置为 `true`, (详情参见第 4.8.4 节“使用部署描述文件 `web.xml`”), 那么当最后一次非心跳请求之后, Servlet 的 `session-timeout` 参数指定的超时时间达到之后, Session 会被判定为超时, Session 及其中所有的 UI 都会被关闭. 一旦 Session 消失, 浏览器端下一次向服务器发起请求时, 将显示一个同步失败的错误消息. 如果不想看到这个丑陋的消息, 你可能希望为 UI 设置一个重定向 URL, 详情请参见第 4.5.2 节“自定义系统消息”.

相关的设定参数请参见 第 4.8.6 节“Servlet 的其他配置参数”.

你可以在服务器端使用 `SessionDestroyListener` 监听器来处理 Session 过期, 详情请参见第 4.7.3 节“用户 Session”.

4.7.8. 关闭 Session

你可以调用 **VaadinSession** 的 `close()` 方法来关闭一个 session. 这种方式通常用于用户从系统中 Log out 时, session 以及所有属于这个 session 的 UI 都需要关闭. 调用这个方法后, session 会被立即关闭, 所有与之相关的对象都将不可使用.

当从 UI 中关闭 session 时, 你通常会希望将用户导向另一个 URL. 你可以使用 **Page** 的 `setLocation()` 方法来实现页面跳转. 这个跳转需要在关闭 session 之前执行, 因为 session 关闭之后 UI 和 page 都将不可用. 下例中, 我们显示一个 logout 按钮, 它将关闭用户 session.

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Logout", event -> { // Java 8
            // Redirect this page immediately
            getPage().setLocation("/myapp/logout.html");

            // Close the session
            getSession().close();
        }));
        // Notice quickly if other UIs are closed
        setPollInterval(3000);
    }
}
```

这样还不够. 当从一个 UI 关闭 session 时, 与这个 session 关联的其他 UI 还会残留在浏览器端. 当客户端引擎发现服务器端的 UI 和 session 消失之后, 它会显示一个 "Session Expired" 消息, 而

且, 当用户点击这个消息之后, 它会默认地重新装载 UI. 你可以在系统消息中定制这个消息内容, 以及点击消息之后的跳转 URL, 详情请参见 第 4.5.2 节 “自定义系统消息”.

当用户操作导致需要发起一次服务器请求, 或者当 UI 心跳发生时, 客户端引擎就会注意到 session 超期. 为了让 UI 更快地发现 session 超期, 你需要让 UI 心跳的间隔时间更快一些, 上面的例子就是这样做的. 你也可以使用服务器端 PUSH 来立即关闭其他 UI 实例, 如下例所示. 对 UI 的访问必须使用同步方式, 详情请参见 第 11.16 节 “服务器端 PUSH”.

```
@Push
public class MyPushyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Logout", event -> { // Java 8
            for (UI ui: VaadinSession.getCurrent().getUIs())
                ui.access(() -> {
                    // Redirect from the page
                    ui.getPage().setLocation("/logout.html");
                });
            getSession().close();
        }));
    }
}
```

在上例中, 我们假定 session 内的所有 UI 都激活了 PUSH 功能, 还假定它们都需要跳转到其他地址; 对于弹出窗口, 通常你会希望关闭它们, 而不是跳转到其他地址. 没有必要对它们分别调用 close() 方法, 因为我们在后面的代码中关闭了整个 session.

4.8. 发布应用程序

Vaadin 应用程序以 Java Web 应用程序的形式发布, 其中可以包含多个 Servlet, 各 Servlet 可以是一个 Vaadin 应用程序, 也可以是其他种类的 Servlet, Web 应用程序中还可以包含静态资源, 比如 HTML 文件等. 这样的 Web 应用程序通常打包为 WAR (Web application ARchive) 文件, WAR 可以被发布到 Java 应用程序服务器(确切的说应该叫做 Servlet 容器). WAR 文件的扩展名为 .war, 是 JAR (Java ARchive) 文件的一种, 象通常的 JAR 文件一样, WAR 文件也是 ZIP 压缩文件, 但其中包含特殊的内容.

关于 Web 应用程序如何打包, 请参考关于 Java Servlet 的书籍.

用 Java Servlet 的术语来说, 一个 "Web 应用程序" 指的是一组 Java Servlet 或 portlet, JSP, 静态的 HTML 页面, 以及用于构成应用程序的各种其他资源. 这样的 Java Web 应用程序通常打包为 WAR 包用于部署. Server 端 Vaadin UI 以 Servlet 的方式运行在这样的 Java Web 应用程序之中. 当然也存在其他类型的 Web 应用程序. 为了避免与通常所说的 "Web 应用程序"混淆, 在本书中我们讨论 Java Web 应用程序时, 一般使用 "WAR", 尽管用这个词并不十分准确.

4.8.1. 在 Eclipse 中创建发布用的 WAR 文件

要把应用程序发布到 Web 服务器上, 你首先需要创建 WAR 包. 以下给出在 Eclipse 中的操作步骤.

1. 选择菜单项 **File → Export** 然后选择 **Web → WAR File**. 或者, 在 Project Explorer 中, 在工程上点击鼠标右键, 然后选择菜单项 **Web → WAR File**.
2. 在 **Web project** 中选择需要导出的过程. 在 **Destination** 中输入导出的目标文件名 (扩展名应该是 .war).
3. 在对话框中输入其他设定, 然后点击 **Finish** 按钮.

4.8.2. Web 应用程序内容

要运行一个 Web application, 其中需要包含以下文件.

Web 应用程序的内部组织

WEB-INF/web.xml (对于 Servlet 3.0 来说, 这个文件是可选的)

这是 Web 应用程序的描述文件, 其中定义了应用程序如何组织, 也就是说, 应用程序中包含哪些 Servlet. 关于这个描述文件的内容, 请参考其他 Java 书籍. 如果你在 Servlet API 3.0 环境中使用 @WebServlet 注解来定义 Vaadin Servlet, 那么这个描述文件是不需要的.

WEB-INF/lib/*.jar

这些文件是 Vaadin 库, 以及 Vaadin 库所依赖的其他库. 这些文件可以在安装包中得到, 或者通过某种依赖管理系统(比如 Maven 或 Ivy)得到.

你编写的 UI 类

你必须将你的 UI 类包含在 JAR 文件中, 放在 WEB-INF/lib 目录下, 或者以 class 文件的形式放在 WEB-INF/classes 目录下

你自己的 theme 文件(可选)

如果你的应用程序使用了特别的 theme (外观), 你必须将你的 theme 包含在 VAADIN/themes/themename 目录下.

Widget 群 (可选)

如果你的应用程序使用了独有的 widget 群, 它必须被编译后放在 VAADIN/widgetset/ 目录下.

4.8.3. Web Servlet 类

使用 Servlet 3.0 API 时, 你通常可以使用 @WebServlet 注解来定义 Vaadin Servlet 类. 与 Servlet 关联的 Vaadin UI 以及其他 Vaadin 相关参数使用另外的 @VaadinServletConfiguration 注解来指定.

```
@WebServlet(value = "/*",
            asyncSupported = true)
@VaadinServletConfiguration(
    productionMode = false,
    ui = MyProjectUI.class)
public class MyProjectServlet extends VaadinServlet { }
```

Vaadin Plugin for Eclipse 将 Servlet 类创建为 UI 类的静态 inner 类. 通常你可能会希望 Servlet 类作为一个普通类存在.

其中的 value 参数是映射到这个 Servlet 上的 URL 模式, 详情请参见第 4.8.5 节 “Servlet 与 URL 模式的映射”. ui 参数是 UI 类. 生产模式默认是关闭的, 因此可以即时编译 theme, 可以显示调试窗口, 其他各种开发期专有功能也都是激活的. 关于 Servlet 和 Vaadin 的其他配置参数, 请阅读后续各章节.

在 Servlet 3.0 工程中, 你也可以使用部署描述文件 web.xml.

4.8.4. 使用部署描述文件 web.xml

部署描述文件是一种 XML 文件, 其文件名为 web.xml, 位于 Web 应用程序的 WEB-INF 子目录下. 在 Java EE 中这个文件是一个标准组件, 它定义了 Web 应用程序应该如何部署. 对于 Servlet API 3.0, 这个描述文件不是必须的, 前面已经说过, 你可以使用 **@WebServlet** 注解来定义 Servlet, 也可以使用 web fragment 来定义, 也可以使用程序来定义. 在同一个应用程序中, 你可以同时使用 web.xml 和 WebServlet.web.xml 文件中的设定将覆盖程序注解中的设定.

下面的例子演示 Servlet 2.4 应用程序部署描述文件的基本内容. 这里只简单地通过 UI 参数为 **com.vaadin.server.VaadinServlet** 指定 UI 类. Servlet 通过 Java Servlet 的标准方式映射到 URL 路径上.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>myservlet</servlet-name>
        <servlet-class>
            com.vaadin.server.VaadinServlet
        </servlet-class>

        <init-param>
            <param-name>UI</param-name>
            <param-value>com.ex.myproj.MyUI</param-value>
        </init-param>

        <!-- If not using the default widget set-->
        <init-param>
            <param-name>widgetset</param-name>
            <param-value>com.ex.myproj.MyWidgetSet</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>myservlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

描述文件定义了一个 Servlet, 名为 myservlet. Servlet 类, **com.vaadin.server.VaadinServlet**, 是 Vaadin 框架提供的, 对所有的 Vaadin 工程来说, 通常都使用同样的 Servlet 类. 为了某些特别目的, 你可能会需要使用继承自 **VaadinServlet** 的自定义 Servlet 类. 注意, 类名需要包含完整的包路径.

Servlet API 版本

上面的描述文件示例是适用于 Servlet 2.4 环境的. 对于比较新的版本, 比如 Servlet 3.0, 你应该使用:

```
<web-app
    id="WebApp_ID" version="3.0"
    xmlns="http://java.sun.com/xml/ns/j2ee"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

```

至少对于服务器 PUSH 功能来说, Servlet 3.0 是需要的.

Widget 群

如果 UI 使用了插件组件, 或使用了定制的 widget, 它就需要定制的 widget 群, 定制的 widget 群可以通过 Servlet 的 `widgetset` 参数来指定. 或者, 你也可以对 UI 类使用 `@WidgetSet` 注解来指定它. 这个设定的参数是一个类名, 这个类名与 widget 群定义文件的路径相同, 但不带 `.gwt.xml` 扩展名. 如果这个参数没有指定, 那么默认会使用 `com.vaadin.DefaultWidgetSet`, 这个类中包含内建的 Vaadin 组件所用到的所有 widget.

除非使用默认的 widget 群 (包含在 `vaadin-client-compiled` JAR 文件中), 否则 widget 群必须编译, 详情请参见第 17 章 使用 Vaadin Add-on 或 第 13.4 节 “编译客户端模块”, 还需要与应用程序一起正确地发布到服务器上.

4.8.5. Servlet 与 URL 模式的映射

Servlet 需要映射到 URL 路径上, 这个 URL 路径就成为 Servlet 所处理的请求的路径.

对 Servlet 类使用 `@WebServlet` 注解来指定映射 URL 路径:

```
@WebServlet(value = "/*", asyncSupported = true)
```

在 `web.xml` 文件中设置如下:

```

<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>

```

上面的示例中, URL 模式被设置为 `/*`. 这个 URL 模式将与工程 context 下的全部 URL 相匹配. 上面的例子中我们将工程 context 设定为 `myproject`, 因此 UI 页面的 URL 将是 `http://localhost:8080/myproject/`.

映射到子路径上

如果应用程序有多个 UI, 或多个 Servlet, 它们必须对应到 URL 的不同路径上, 匹配不同的 URL 模式. 而且, 你可能需要使用某些路径对外提供静态内容. 使用 `/myui/*` 这样的 URL 模式, 将匹配到 `http://localhost:8080/myproject/myui/` 这样的 URL. 注意在 URL 模式的末尾必须包含斜线和星号. 这时, 你还需要将 `/VAADIN/*` 这样的 URL 映射到 Servlet 上 (除非你用静态的方式为这个 URL 提供服务, 详情参见后面的章节).

对 Servlet 类使用 `@WebServlet` 注解时, 你可以用列表的形式指定多个 URL 映射, 列表使用大括号括起, 示例如下:

```
@WebServlet(value = {"/myui/*", "/VAADIN/*"}, 
            asyncSupported = true)
```

在 `web.xml` 文件中设置如下:

```

...
<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/myui/*</url-pattern>

```

```

    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>myservlet</servlet-name>
        <url-pattern>/VAADIN/*</url-pattern>
    </servlet-mapping>

```

如果你有多个 servlet, 你只能指定一个 /VAADIN/* 映射. 只要你的 Servlet 是 Vaadin Servlet, 那么你将这个 URL 路径映射到哪个 Servlet 上是无所谓的.

如果你的 widget 群和 theme (无论是默认的还是定制的), 都以 Web 应用程序的 /VAADIN 目录下的静态内容的方式向外提供, 那么你就不必指定上面说的 /VAADIN/* 路径映射. 这个映射只是为了从 Vaadin JAR 中动态的向外提供这些内容. 在生产环境下, 推荐以静态文件的方式向外提供这些内容, 因为这样速度更快. 如果你在同一个应用程序内向外提供这些内容的话, 你不应该将根路径 /* 映射到 Vaadin Servlet 上, 因为这样会将所有的请求都映射给 Servlet.

4.8.6. Servlet 的其他配置参数

Servlet 类, 或者部署描述文件, 可以使用很多参数来控制 Servlet 的运行. 关于 Servlet 参数的完整文档, 请参阅 Java Servlet 规范. **@VaadinServletConfiguration** 则使用一些特有的参数, 详情见下文.

在 web.xml 中, 大多数参数可以通过两种方式设定, 1, <context-param> 参数应用于整个 Web 应用程序, 因此会对所有的 Vaadin Servlet 有效, 2, <init-param> 参数, 只是用于单个 Servlet. 如果以上两种参数都有指定, 那么 Servlet 参数将覆盖 context 参数.

生产模式

Vaadin 应用程序默认运行在 调试模式 下 (或者叫 开发模式), 这种模式适用于应用程序的开发阶段. 它打开了各种调试功能. 对于生产环境, 你应该在 **@VaadinServletConfiguration** 中设置 productionMode=true, 或者在 web.xml 中设置如下:

```

<context-param>
    <param-name>productionMode</param-name>
    <param-value>true</param-value>
    <description>Vaadin production mode</description>
</context-param>

```

这个设置参数, 以及调试模式和生产模式的详细信息, 请参见 第 11.3 节 “Debug 模式和 Debug 窗口”.

定制的 UI Provider

Vaadin 通常使用 **DefaultUIProvider** 来创建 **UI** 类的实例. 如果你需要使用定制的 UI Provider, 你可以使用 *UIProvider* 参数来配置 UI Provider 类. UI Provider 类将被注册到 **VaadinSession** 中.

在 web.xml 文件中设置如下:

```

<servlet>
    ...
    <init-param>
        <param-name>UIProvider</param-name>
        <param-value>com.ex.my.MyUIProvider</param-value>
    </init-param>

```

这个参数通常与特定的 Servlet 相关联, 但也可以在 context 范围内指定.

UI 心跳

Vaadin 使用心跳来追踪 UI, 详情请参见 第 4.7.5 节 “UI 过期”. 如果用户关闭了 Vaadin 应用程序的浏览器窗口, 或者跳转到了其他的页面, 那么页面中运行的客户端引擎将停止向服务器发送心跳请求, 服务器最终将清除 UI 实例.

心跳请求的间隔时间可以通过 `heartbeatInterval` 参数指定, 单位为秒, 这个参数可以作为 `context` 参数对整个 Web 应用程序全局设置, 也可以作为 `init` 参数对单个 Servlet 进行设置. 默认值是 300 毫秒 (5 分钟).

在 `web.xml` 文件中设置如下:

```
<context-param>
    <param-name>heartbeatInterval</param-name>
    <param-value>300</param-value>
</context-param>
```

用户停止操作后的 Session 超时

在通常的 Servlet 操作中, Session 超时时间定义了 Session 不再活动时, 服务器清除 Session 之前应该等待的时间. Session 不再活动是从最后一次向服务器发送请求开始计算的. 不同的 Servlet 容器使用不同的超时时间默认值, 比如 Apache Tomcat 的默认超时时间是 30 分钟. 你可以在 `<web-app>` 之下设置超时时间:

在 `web.xml` 文件中设置如下:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

Session 超时时间的设定应该比心跳间隔时间长, 否则心跳请求还来不及保持 Session 激活, Session 就已经被关闭了. 由于 Session 超期时, UI 仍会假定 Session 继续存在, 因此在浏览器中会出现同步失败的错误通知消息.

但是, 通常情况下, 如果心跳间隔比 Session 超时时间短, 会使得 Session 永远不过期. 如果 Servlet 的 `closeIdleSessions` 参数被启用(默认为禁用), 最后一次非心跳请求之后, 如果经过了 `session-timeout` 参数指定的时间, Vaadin 将关闭 UI 和 Session.

在 `web.xml` 文件中设置如下:

```
<servlet>
    ...
    <init-param>
        <param-name>closeIdleSessions</param-name>
        <param-value>true</param-value>
    </init-param>
```

PUSH 模式

对于一个 UI, 你可以打开服务器 PUSH 功能, 方法是对 UI 类使用 `@Push` 注解, 或在部署描述文件中配置, 详情请参见 第 11.16 节 “服务器端 PUSH”. PUSH 模式由 `pushmode` 参数定义. `automatic` 模式会在 `access()` 结束之后, 将变化信息自动推送到浏览器端. 使用 `manual` 模式, 你需要使用 `push()` 方法自行显式地进行推送. 如果你使用兼容 Servlet 3.0 的服务器, 你可能会希望使用 `async-supported` 参数允许异步处理.

在 `web.xml` 文件中设置如下:

```

<servlet>
...
<init-param>
    <param-name>pushmode</param-name>
    <param-value>automatic</param-value>
</init-param>
<async-supported>true</async-supported>

```

防止跨站请求伪装

Vaadin 使用一种保护机制来防止恶意的跨站请求伪装 (cross-site request forgery, 简称 XSRF 或 CSRF), 也叫 one-click 攻击, 或者叫 session riding, 这是一种安全漏洞, 可被用来在Web 服务器上执行未授权的命令. Vaadin 的保护机制默认是启用的. 但是它同时也阻止了 Vaadin 应用程序的一些测试方法, 比如使用 JMeter. 这种情况下, 你可以禁用这个保护机制, 方法是将 `disable-xsrf-protection` 参数设置为 `true`.

在 `web.xml` 文件中设置如下:

```

<context-param>
    <param-name>disable-xsrf-protection</param-name>
    <param-value>true</param-value>
</context-param>

```

4.8.7. 发布配置

部署配置中的 Vaadin 专有参数, 可以通过 **VaadinSession** 管理的 **DeploymentConfiguration** 对象得到.

```

DeploymentConfiguration conf =
    getSession().getConfiguration();

// Heartbeat interval in seconds
int heartbeatInterval = conf.getHeartbeatInterval();

```

Java Servlet 定义中指定的参数, 比如 Session 超时时间, 可以通过低阶的 **HttpSession** 或 **PortletSession** 对象得到, 这些对象被包装在 Vaadin 的 **WrappedSession** 中. 要得到这个包装对象, 你可以使用 **VaadinSession** 的 `getSession()` 方法.

```

WrappedSession session = getSession().getSession();
int sessionTimeout = session.getMaxInactiveInterval();

```

你也可以通过 **HttpSession** 和 **PortletSession** 接口访问 Session 的其他属性, 比如设置和读取 session attribute, 某个 Servlet Session 或 Portlet Session 下属的所有 Servlet, 可以通过这些 attribute 共享数据.

UI

5.1. 概述	98
5.2. 接口和抽象类	99
5.3. 组件的共通功能	101
5.4. Field 组件	112
5.5. 选择组件	117
5.6. 组件的扩展	123
5.7. Label	123
5.8. Link	128
5.9. TextField	130
5.10. TextArea	134
5.11. PasswordField	136
5.12. RichTextArea	137
5.13. 使用 DateField 输入日期和时间	139
5.14. Button	145
5.15. CheckBox	146
5.16. ComboBox	147
5.17. ListSelect	149
5.18. NativeSelect	150
5.19. OptionGroup	151
5.20. TwinColSelect	154
5.21. Table	155
5.22. Tree	174
5.23. MenuBar	175

5.24. Upload	178
5.25. ProgressBar	181
5.26. Slider	184
5.27. Calendar	186
5.28. 使用 CustomComponent 创造复合组件	202
5.29. 使用 CustomField 组合 Field 组件	203
5.30. 内嵌资源	203

本章介绍 Vaadin 中主要的 UI 组件(layout 管理组件除外).

对于 Vaadin 7 版本, 本章的一部分内容还有待审核, 尤其是关于 **Table** 组件的数据绑定问题, 为了尽快完成本书, 我们暂时发布这个未审核完毕的版本. 因此请随时查看本书的 Web 版有无更新, 或者请关注印刷版的下一个版本.

5.1. 概述

Vaadin 提供了功能广泛的 UI 组件, 还允许你开发自定义组件. 图 5.1 “UI 组件的类层级关系图”展示了 UI 组件类和接口的继承层级关系. 接口表示为灰色, 抽象类为桔黄色, 一般类为蓝色. 这个图的详细注解版本可参见 *Vaadin Cheat Sheet*.

在接口层级关系最顶端的是 **Component** 接口. 在类层级关系最顶端的是 **AbstractComponent** 类. 从这个类继承了两个其他抽象类: **AbstractField**, 是所有 Field 组件的基类, 以及 **AbstractComponentContainer**, 是各种容器组件和 Layout 管理组件的基类. 不与数据模型绑定的组件, 比如 Label 和 Link, 直接继承自 **AbstractComponent**.

同一个窗口内的多个组件, 其布局由布局管理组件来控制, 这一点与桌面应用程序的 Java UI 开发工具类似. 此外, 使用 **CustomLayout** 组件, 你可以编写自定义的布局组件, 自定义布局是 HTML 模板形式, 模板中包含了其他子组件的位置. 阅读上面的继承关系图, 我们可以看到布局管理组件继承自 **AbstractComponentContainer** 类, 以及 **Layout** 接口. 布局管理组件的详情, 请参见第 6 章 布局管理.

从对象包含层级关系的角度来看, 最顶层是 **Window** 对象, 它之下包含了多层次的布局管理组件, 这些布局管理组件再包含其他布局管理组件, Field 组件, 以及其他各种可见组件.

你可以在 Vaadin Demo 的示例应用程序中浏览 Vaadin 库中的内建 UI 组件. 这个示例程序会为每个 UI 组件显示一段介绍信息, JavaDoc 文档, 以及示例代码.

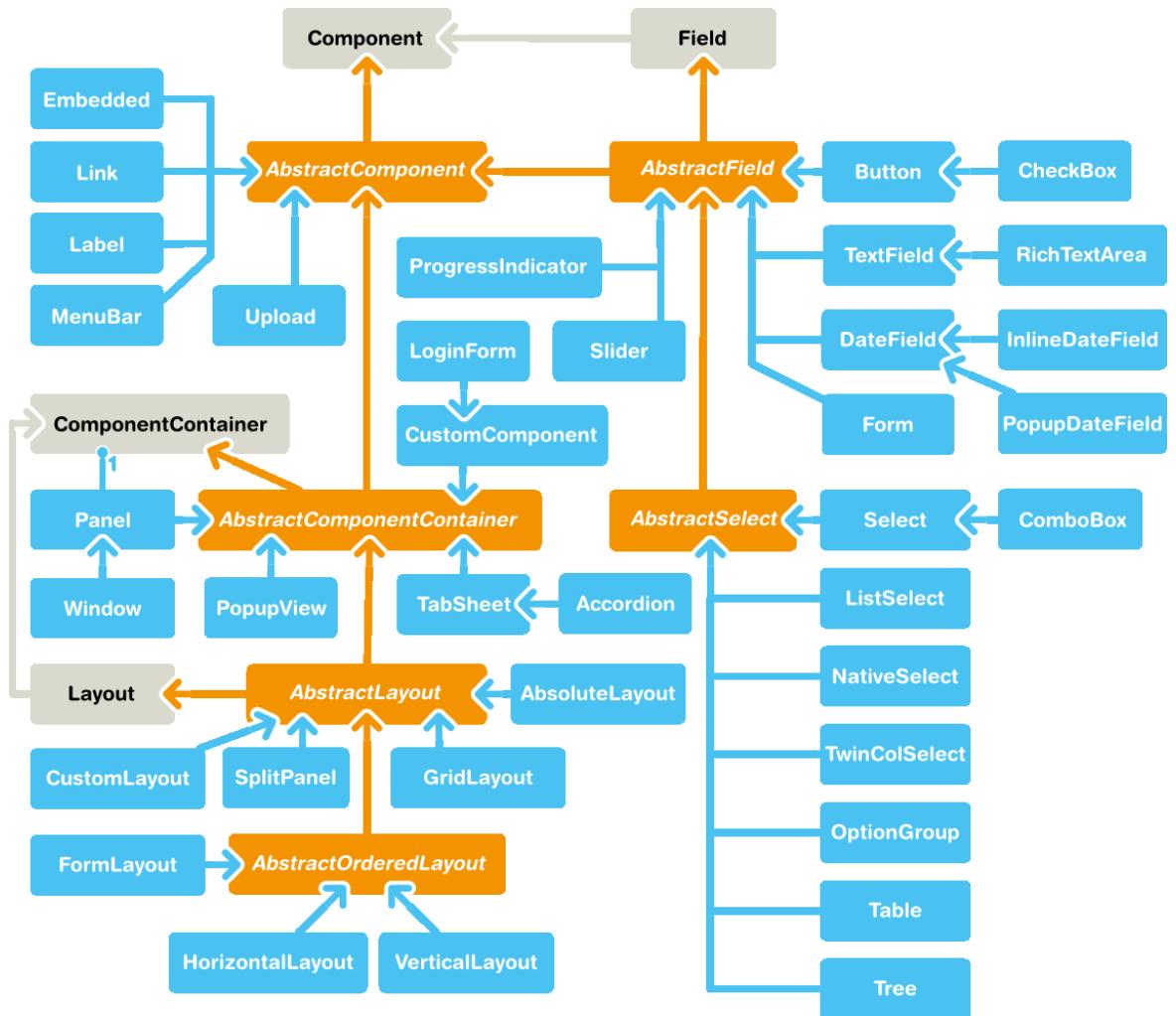
除内建组件外, 还有很多组件可以通过插件的形式获得, 插件可从 Vaadin Directory 得到, 或者从其他来源得到. 既有商业插件, 也有免费插件. 关于插件的安装方法, 请参见第 17 章 使用 Vaadin Add-on.


Vaadin Cheat Sheet 和 Refcard

图 5.1 “UI 组件的类层级关系图” 包含在 Vaadin Cheat Sheet 中, Vaadin Cheat Sheet 展示了 UI 组件级数据绑定类/接口的基本层级关系. 你可以在以下地址下载 Vaadin Cheat Sheet: <http://vaadin.com/book>.

上图也包含在 DZone Refcard 中, DZone Refcard 可以在以下地址得到: <https://vaadin.com/refcard>.

图 5.1. UI 组件的类层级关系图



5.2. 接口和抽象类

Vaadin UI 组件构建在一系列接口和抽象类的基础之上, 这些接口和抽象类定义并实现了所有组件的共通功能, 还定义并实现了组件状态在服务器端和客户端之间序列化的基本逻辑.

本节介绍基本的组件接口和抽象类. 关于布局组件和其他组件容器的抽象类, 请参见第 6 章 布局管理. Vaadin 数据模型相关的接口, 请参见第 8 章 组件与数据绑定.

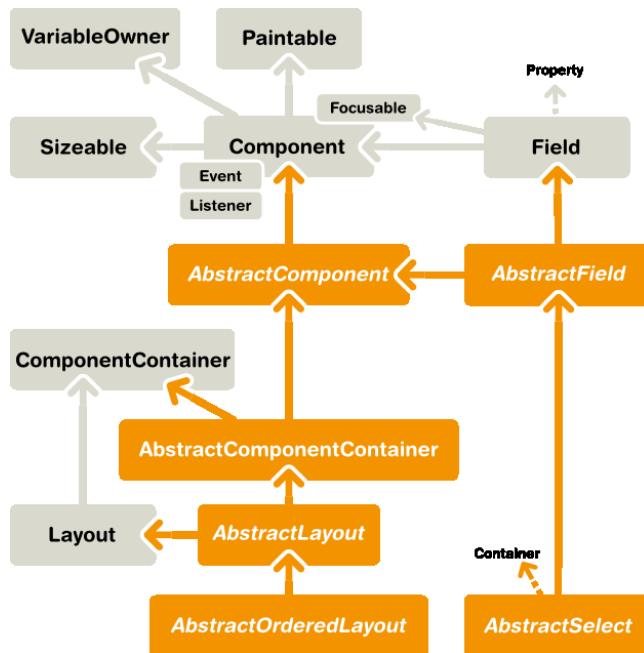
所有的组件也都实现了 **Paintable** 接口, 这个接口用于将组件序列化 ("描绘") 到客户端, 还实现了相反地 **VariableOwner** 接口, 这个接口用于将组件状态或用户操作从客户端反序列化到服务器端.

除 Vaadin 框架定义的接口外, 所有的组件还都实现了 **java.io.Serializable** 接口, 用于序列化. 对于很多集群计算和云计算解决方案来说, 序列化功能是必须的.

5.2.1. Component 组件

Component 接口与 **AbstractComponent** 类相对应, **AbstractComponent** 类实现了 **Component** 接口定义的所有方法.

图 5.2. 组件接口和抽象类



组件包含层级关系的管理

组件以 UI 层级关系的形式管理. 层级关系由布局管理组件控制, 或者更一般地说, 由实现了 **ComponentContainer** 接口的容器组件控制. 因此一个容器就是它包含的子组件的父组件.

`getParent()` 方法可以得到一个组件的父组件. 与此对应的存在一个 `setParent()` 方法, 但一般很少使用这个方法, 通常我们使用 **ComponentContainer** 接口的 `addComponent()` 方法来添加子组件, 这个方法将自动地为子组件设置它的父组件.

组件创建过程中还不能确定自己的父组件, 所以在组件的构造方法中, 你无法通过 `getParent()` 方法得到父组件.

将组件关联到 UI, 会触发对 `attach()` 方法的调用. 相对的, 将组件从容器中删除, 会触发对 `detach()` 方法的调用. 如果新添加的组件的父组件已经被关联到 UI 上, 那么 `setParent()` 方法会立即调用 `attach()` 方法.

```

public class AttachExample extends CustomComponent {
    public AttachExample() {
    }

    @Override
    public void attach() {
        super.attach(); // Must call.

        // Now we know who ultimately owns us.
        ClassResource r = new ClassResource("smiley.jpg");
        Image image = new Image("Image:", r);
        setCompositionRoot(image);
    }
}
  
```

组件与 UI 的绑定逻辑由 **AbstractComponent** 类实现，详情请参见 第 5.2.2 节“**AbstractComponent**”。

5.2.2. AbstractComponent

AbstractComponent 是所有 UI 组件的基类。它是**Component** 接口的(唯一的)实现类，它实现了 Component 接口定义的所有方法。

AbstractComponent 中有唯一一个抽象方法, `getTag()` 方法，它返回各个特定组件类的序列化 ID。当(而且仅当)创建全新的组件时，需要实现这个方法。**AbstractComponent** 类管理了组件状态在客户端与服务器端之间序列化的大多数任务。关于创建新组件以及组件的序列化，详情请参见第 16 章 与客户端集成。

5.3. 组件的共通功能

组件的基类和接口提供了大量的功能。下面我们来看看最常用到的功能。本文未介绍到的其他功能，请参见 Java API 文档。

Component 接口定义了很多属性，通过对应的 getter 方法可以得到属性值，通过 setter 方法可以控制属性值。

5.3.1. 标题

标题是与 UI 组件相伴随的文字，用于解释 UI 组件，通常显示在组件的上方，左方，或内部。标题的内容会被自动理解为原始字符串，因此在标题中不能显示原生的 HTML 内容。

标题文本通常通过组件构造函数的第一个参数来设置，或者使用 `setCaption()` 方法来设置。

```
// New text field with caption "Name"
TextField name = new TextField("Name");
layout.addComponent(name);
```

组件标题的管理和显示，默认由这个组件所属的布局管理组件或组件容器负责。比如，布局管理组件 **VerticalLayout** 将它的子组件的标题在各组件的上方左对齐显示，而布局管理组件 **FormLayout** 将它的子组件垂直排列，并将标题显示在各组件左侧，标题与对应的组件在各自的列中左对齐显示(译注：原文如此，参照下面的图，貌似组件左对齐显示，标题右对齐显示)。**CustomComponent** 不管理它的复合组件的根组件的标题，因此即使根组件有标题，它也不会被显示在画面中。

某些组件，比如 **Button** 和 **Panel**，会管理自己的标题，将标题显示在自己内部。

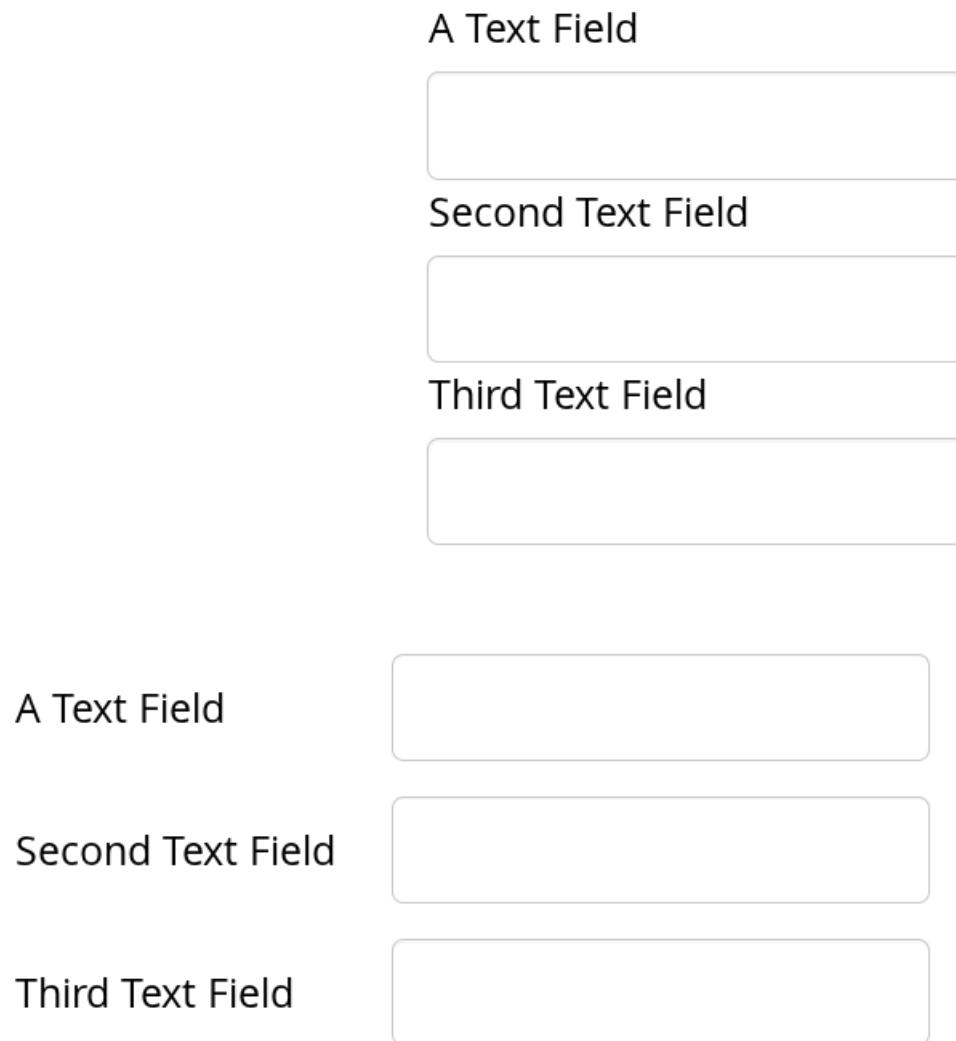
图标(参见第 5.3.4 节“图标”)与标题关系密切，通常显示在标题的左侧或右侧，具体情况由不同的组件及组件所属的布局决定。此外，Field 组件的“必须项目指示器”通常也显示在标题的左侧或右侧。

实现标题的另一种方案，是使用另一个组件来当作标题，通常是用 **Label**, **TextField**, 或 **Panel**。以 **Label** 为例，它可以使用 HTML 来高亮显示快捷键，也可以将标题内容绑定到数据源。**Panel** 则提供了一种简便方法，可以在组件外围同时添加标题和边框。

CSS 样式规则

```
.v-caption {}
.v-captiontext {}
.v-caption-clear elem {}
.v-required-field-indicator {}
```

图 5.3. 由 **VBoxLayout** 和 **FormLayout** 管理的组件标题.



标题被描绘在 HTML 元素的内部, 这个元素带有 `v-caption` CSS 样式类. 组件所属的布局管理组件可能会将组件标题封装在其他某种与标题相关的 HTML 元素内部.

某些布局方案将标题文本放在 `v-captiontext` 元素内. 某些布局方案在标题内使用 `v-caption-clear elem` 来清除 CSS `float` 属性. Field 组件的(可选的)"必须项目指示器"被包含在一个独立的元素中, 这个元素的样式是 `v-required-field-indicator`.

5.3.2. 描述信息和提示信息

继承自 **AbstractComponent** 的所有组件, 除标题之外还有一个描述信息. 描述信息通常以提示信息(tooltip)的方式显示, 当鼠标指针在组件上停留一个短暂的时间之后, 提示信息就会显示出来.

描述信息可以使用 `setDescription()` 方法来设置, 使用 `getDescription()` 方法来取得.

```
Button button = new Button("A Button");
button.setDescription("This is the tooltip");
```

提示信息的显示状况参见 图 5.4 “组件描述信息以提示信息Tooltip的方式显示”.

图 5.4. 组件描述信息以提示信息(Tooltip)的方式显示



描述信息在大多数组件中都作为提示信息来显示.

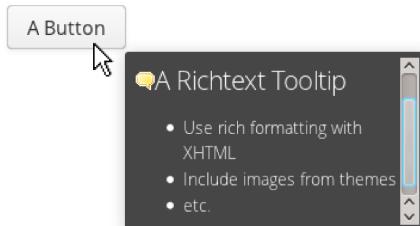
当使用 `setComponentError()` 方法设置了组件错误信息时, 错误信息通常也会被显示在提示信息中, 但在描述信息的下方. 处于错误状态的组件还会显示错误指示器. 详情请参见 第 4.5.1 节“错误指示器和消息”.

描述信息实际上不是纯文本, 你可以使用 HTML tag 控制它的格式. 这种富文本的描述信息中可以包含任意的 HTML 元素, 包括图片.

```
button.setDescription(
    "<h2><img src=\"../VAADIN/themes/sampler/icons/comment_yellow.gif\"/>" +
    "A richtext tooltip</h2>" +
    "<ul>" +
    "  <li>Use rich formatting with HTML</li>" +
    "  <li>Include images from themes</li>" +
    "  <li>etc.</li>" +
    "</ul>");
```

运行结果参见 图 5.5 “富文本的提示信息”.

图 5.5. 富文本的提示信息



注意, 这个属性的设置方法和取得方法定义在 **Field** 接口中, 是属于所有 Field 的, 而不是定义在 **Component** 接口中, 属于所有组件的.(译注, 这句话对于 Vaadin 6 是正确的, 但在 Vaadin 7 中, 这个属性的取得方法定义在 **Component** 接口中, 设置方法定义在 **AbstractComponent** 类中)

5.3.3. 激活与禁用

`enabled` 属性控制用户是否可以使用这个组件. 被禁用的组件是可见的, 但被显示为灰色, 表示它目前不可使用.

组件默认总是激活的. 你可以使用 `setEnabled(false)` 方法禁用一个组件.

```
Button enabled = new Button("Enabled");
enabled.setEnabled(true); // The default
layout.addComponent(enabled);

Button disabled = new Button("Disabled");
disabled.setEnabled(false);
layout.addComponent(disabled);
```

图 5.6 “激活和禁用的 **Button**” 展示了激活的按钮和禁用的按钮.

图 5.6. 激活和禁用的 **Button**



被禁用的组件会自动进入只读状态。这种组件不会有用户操作发送到服务器端，而且，作为一种重要的安全特性，服务器端组件处于只读状态时也不会接收来自客户端的状态更新。这个特性存在于 Vaadin 的所有内建组件中，而且对于所有 **Field** 组件的值属性也自动做了这类处理。对于自定义 widget，你需要保证在服务器端对于所有安全攸关的变量，都有正确地检查只读状态。

CSS 样式规则

被禁用的组件除组件原有的样式外，还会带有 `v-disabled` CSS 样式。如果要匹配同时具有这两种样式的组件，你需要将两种样式类名用点号结合起来，参见下面的例子。

```
.v-textfield.v-disabled {
    border: dotted;
}
```

这个 CSS 规则会让被禁用的文本输入框的边框显示为点状。

在 Valo Theme 中，被禁用组件的不透明度是由 `$v-disabled-opacity` 参数指定的，详情请参见第 7.6.2 节“共通设定”。

5.3.4. 图标

图标是与 UI 组件相伴随的一种图像标记，用于解释组件的含义，通常显示在组件上方，左方，或组件内部。图标与标题密切相关（参见第 5.3.1 节“标题”），通常显示在标题的左侧或右侧，具体如何由组件和包含组件的布局管理器决定。

可以通过 `setIcon()` 方法设置组件的图标。图片以资源的方式提供，最通用的是 **ThemeResource**。

```
// Component with an icon from a custom theme
TextField name = new TextField("Name");
name.setIcon(new ThemeResource("icons/user.png"));
layout.addComponent(name);

// Component with an icon from another theme ('runo')
Button ok = new Button("OK");
ok.setIcon(new ThemeResource("../runo/icons/16/ok.png"));
layout.addComponent(ok);
```

组件图标的管理和显示，默认由组件所属的布局管理组件或组件容器负责。比如，布局管理组件 **VerticalLayout** 将它的子组件的图标在各组件的上方左对齐显示，而布局管理组件 **FormLayout** 将它的子组件垂直排列，并将图标显示在各组件左侧，图标与对应的组件在各自的列中左对齐显示。**CustomComponent** 不管理它的复合组件的根组件的图标，因此即使根组件有图标，它也不会被显示在画面中。

图 5.7. 使用 **Theme Resource** 显示一个图标.



某些组件, 比如 **Button** 和 **Panel**, 会管理自己的图标, 将图标显示在自己内部.

除图片资源外, 你还可以使用 字体图标, 这是一种包含在特殊字体之内的图标, 但以特殊资源的方式进行处理. 详情请参见 第 7.7 节“字体图标”.

CSS 样式规则

标签被描绘在 HTML 元素的内部, 这个元素带有 v-icon CSS 样式类. 组件所属的布局管理组件可能会将组件图标和标签封装在与标签相关的 HTML 元素内部, 比如 v-caption.

5.3.5. 语言环境(Locale)

locale 属性定义了组件使用的国家和语言. 你可以将 locale 信息与国际化方案结合起来, 实现资源的本地化. 某些组件, 比如 **DateField**, 使用语言环境来实现组件的本地化.

你可以使用 setLocale() 方法来设置一个组件(或整个应用程序)的语言环境, 例子如下:

```
// Component for which the locale is meaningful
InlineDateField date = new InlineDateField("Datum");

// German language specified with ISO 639-1 language
// code and ISO 3166-1 alpha-2 country code.
date.setLocale(new Locale("de", "DE"));

date.setResolution(Resolution.DAY);
layout.addComponent(date);
```

日期输入框的运行结果参见 图 5.8 “为 **InlineDateField** 设置语言环境”.

图 5.8. 为 **InlineDateField** 设置语言环境

取得语言环境

你可以使用 `getLocale()` 方法得到组件的语言环境。如果组件的语言环境没有指定，也就是说没有被明确地设置过，将会使用父组件的语言环境。如果所有父组件的语言环境都没有设置过，将会使用 UI 的语言环境，如果 UI 的语言环境也没有设置过，则会使用系统默认的语言环境，系统默认语言环境由 `Locale.getDefault()` 决定。

如果组件还没有关联到 UI，那么 `getLocale()` 方法会返回 `null`，在组件构造方法中通常就会发生这种情况，所以使用这个方法进行国际化需要用一种比较笨拙的手段。你可以在 `attach()` 方法内取得语言环境，例子如下：

```
Button cancel = new Button() {
    @Override
    public void attach() {
        super.attach();
        ResourceBundle bundle = ResourceBundle.getBundle(
            MyAppCaptions.class.getName(), getLocale());
        setCaption(bundle.getString(MyAppCaptions.CancelKey));
    }
};
layout.addComponent(cancel);
```

但在实际运用中比较好的一种手段是，在组件的创建过程中，使用当前 UI 的语言环境来取得本地化资源。

```
// Captions are stored in MyAppCaptions resource bundle
// and the UI object is known in this context.
ResourceBundle bundle =
    ResourceBundle.getBundle(MyAppCaptions.class.getName(),
        UI.getCurrent().getLocale());

// Get a localized resource from the bundle
Button cancel =
```

```

        new Button(bundle.getString(MyAppCaptions.CancelKey));
layout.addComponent(cancel);

```

选择语言环境

对许多应用程序来说面临的一个共同任务就是选择语言环境。在下面的例子中，我们用一个**ComboBox**来实现这个任务，这个下拉框的选择项列出了Java中所有可用的语言环境。

```

// The locale in which we want to have the language
// selection list
Locale displayLocale = Locale.ENGLISH;

// All known locales
final Locale[] locales = Locale.getAvailableLocales();

// Allow selecting a language. We are in a constructor of a
// CustomComponent, so preselecting the current
// language of the application can not be done before
// this (and the selection) component are attached to
// the application.
final ComboBox select = new ComboBox("Select a language") {
    @Override
    public void attach() {
        super.attach();
        setValuegetLocale();
    }
};
for (int i=0; i<locales.length; i++) {
    select.addItem(locales[i]);
    select.setItemCaption(locales[i],
        locales[i].getDisplayName(displayLocale));

    // Automatically select the current locale
    if (locales[i].equals(getLocale()))
        select.setValue(locales[i]);
}
layout.addComponent(select);

// Locale code of the selected locale
final Label localeCode = new Label("");
layout.addComponent(localeCode);

// A date field which language the selection will change
final InlineDateField date =
    new InlineDateField("Calendar in the selected language");
date.setResolution(Resolution.DAY);
layout.addComponent(date);

// Handle language selection
select.addValueChangeListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        Locale locale = (Locale) select.getValue();
        date.setLocale(locale);
        localeCode.setValue("Locale code: " +
            locale.getLanguage() + "_" +
            locale.getCountry());
    }
});
select.setImmediate(true);

```

上面的例子产生的 UI 参见图 5.9 “选择语言环境”.

图 5.9. 选择语言环境



5.3.6. 只读

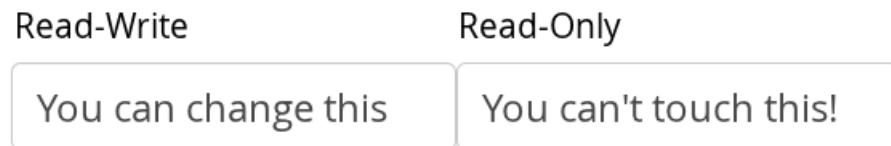
这个属性决定组件的值是否可以变更. 这个属性通常适用于 **Field** 组件, 这类组件拥有可被用户编辑的值.

```
TextField readwrite = new TextField("Read-Write");
readwrite.setValue("You can change this");
readwrite.setReadOnly(false); // The default
layout.addComponent(readwrite);

TextField readonly = new TextField("Read-Only");
readonly.setValue("You can't touch this!");
readonly.setReadOnly(true);
layout.addComponent(readonly);
```

上述例子的运行结果是产生一个只读的文本输入框, 参见 图 5.10 “只读组件.”.

图 5.10. 只读组件.



将布局管理组件或者别的某种组件容器设置为只读, 通常不会导致它包含的子组件也递归地设置为只读. 这一点与其他属性不同, 比如禁用状态, 通常就会递归地适用到包含的子组件上.

注意 Selection 组件的值是一个选项列表, 而不是其中的选择项. 所以一个只读的 Selection 组件只是禁止它的选项列表发生变更, 但其他变更是允许的. 比如, 如果你有一个只读的 **Table** 处于可编辑模式, 那么它包含的 Field, 以及底层的数据模型仍然是可以编辑的, 而且用户还可以对表进行排序, 也可以调整列的顺序.

客户端状态变更无法被发送到服务器端, 而且更重要的是, 服务器端不会接受对一个只读的 **Field** 组件的值的变更. 后面这一点是很重要的安全特性, 因为恶意用户可以对只读的 Field 伪造状态变更请求. 这个功能由 **AbstractField** 类的 `setValue()` 方法进行处理, 所以你也不能通过程序来改变只读 Field 组件的值. 在只读 Field 组件上调用 `setValue()` 方法将导致 **Property.ReadOnlyException** 例外.

还要注意，当只读状态被应用到 Field 的属性值上时，它并不会被应用到组件的其他变量上。只读组件可以从客户端收到其他变量的变更请求，某些变更请求是可以接受的，比如 **Table** 的滚动条位置。自定义的 widget 应该对绑定到业务数据上的变量检查其只读状态。

CSS 样式规则

将一个通常可编辑的组件设置为只读状态，可能会改变它的外观表现，以便禁止用户编辑它。只读状态下，除 CSS 样式外，HTML 接口也可能变化。比如，**TextField** 的文本编辑框会消失，它的外观表现会变得类似于 **Label**。

只读组件将带有 `v-readonly` 样式。以下 CSS 样式规则将使所有的只读 **TextField** 组件的文字显示为斜体字。

```
.v-textfield.v-readonly {
    font-style: italic;
}
```

5.3.7. 样式名称

`style name` 属性为组件指定一个或多个自定义的 CSS 样式类。`getStyleName()` 方法返回当前的样式名称列表，多个样式名称之间以空格分隔。`setStyleName()` 方法将目前的样式替换为指定的样式名称，或以空格分隔的样式名称列表。你也可以使用 `addStyleName()` 和 `removeStyleName()` 方法增加或删除单个的样式名称。这里的样式名称必须是有效的 CSS 样式名。

```
Label label = new Label("This text has a lot of style");
label.addStyleName("mystyle");
layout.addComponent(label);
```

样式名称将以两种方式出现在组件的 HTML 元素上：1. 以你指定的样式名原样出现 2. 以组件特有的样式名为前缀，你指定的样式名附加在其后。比如，如果你在 **Button** 上添加了一个样式名 `mystyle`，这个组件将同时带有 `mystyle` 和 `v-button-mystyle` 样式。这两种形式的样式都可能与 Vaadin 内建的样式名冲突。比如，`focus` 样式可能与内建的同名样式发生冲突，**Panel** 组件的 `content` 样式会与内建的 `v-panel-content` 样式冲突。

下面的 CSS 规则将适用于带有 `mystyle` 样式的所有组件。

```
.mystyle {
    font-family: fantasy;
    font-style: italic;
    font-size: 25px;
    font-weight: bolder;
    line-height: 30px;
}
```

上例的运行结果参见 图 5.11 “带有自定义样式的组件”

图 5.11. 带有自定义样式的组件



5.3.8. 可见和隐藏

通过将 `visible` 属性设置为 `false`，可以隐藏一个组件、标题、图标，以及组件的其他一切元素全部都会被隐藏。隐藏组件不仅仅是不可见，而且它的内容也会完全不发送到浏览器端。也就是说，隐藏组

件不仅仅是通过 CSS 规则达到不可见的效果。如果你的某个组件包含了机密信息，必须只在某些特定的状态下才显示，那么这个功能对于安全性是很重要的，因为将组件设置为隐藏可以保证它的内容在浏览器中完全不出现。

```
TextField invisible = new TextField("No-see-um");
invisible.setValue("You can't see this!");
invisible.setVisible(false);
layout.addComponent(invisible);
```

上面例子生成的不可见组件参见图 5.12 “不可见组件.”。

图 5.12. 不可见组件。

注意不可见元素还是会留下一些痕迹。不可见组件所属的布局元素不会消失，而会在整体布局中留下额外的空白。而且组件的扩张比例也会与组件可见时一样有效，因为扩张比例是由组件所属的布局元素实现的，而不是组件本身实现的。

如果你希望组件只是在视觉上不可见，你可以使用自定义 theme 来将它设置为 `display: none` 样式。对某些特别的组件来说，即使在 CSS 中将它们设置为不可见，它们也会留下一个界面效果，那么前面讲的这种办法就很有用。如果被隐藏的组件的尺寸未指定，而且它所属的布局管理组件的尺寸也未指定，那么当组件消失时，布局容器本身也会消失。如果你希望组件保持它的尺寸，你必须将它的字体等等所有属性都设置为透明，来达到让它不可见的效果。这种情况下，在浏览器端其实很容易将组件的不可见内容变为可见的。

通过 `visible` 属性变为不可见的组件不存在一个对应的 CSS 样式类来标识它是隐藏的。组件的 HTML 元素虽然存在，但带有 `display: none` 样式，这个样式将覆盖其他所有的 CSS 样式。

5.3.9. 控制组件的尺寸

Vaadin 组件的尺寸是可控的；你可以自由控制组件在屏幕上显示的大小。

所有的组件都实现了 **Sizeable** 接口，这个接口提供了很多控制方法和常数，用于设置组件的宽和高，尺寸可以是绝对单位也可以是相对单位，也可以将尺寸保留为未定义的状态。

组件的尺寸可以通过 `setWidth()` 和 `setHeight()` 方法来设置。这些方法接受浮点值作为尺寸参数。在第 2 个参数中你需要给定一个尺寸单位。可用的单位参见下面的表 5.1 “尺寸单位”。

```
mycomponent.setWidth(100, Sizeable.UNITS_PERCENTAGE);
mycomponent.setWidth(400, Sizeable.UNITS_PIXELS);
```

另一种方法是以字符串的形式指定尺寸。字符串格式遵照 HTML/CSS 中用于指定元素尺寸的格式标准。

```
mycomponent.setWidth("100%");
mycomponent.setHeight("400px");
```

百分数 "100%" 将使组件占据某个方向上所有的可用空间（参见下表中的 `Sizeable.UNITS_PERCENTAGE`）。你也可以使用比较简便的 `setSizeFull()` 方法，将纵横两个方向的尺寸都设置为 100%。

组件在单个或两个方向上的尺寸可以是未定义，这时组件会占据它所需要的最小空间。大多数组件默认尺寸都是未定义，但某些布局管理组件会在水平方向上占据全尺寸。你可以对 `setWidth()` 和 `setHeight()` 方法使用 `Sizeable.SIZE_UNDEFINED` 参数，将高度或宽度设置为未定义。

要时刻注意，尺寸未定义的布局组件，其中包含的组件的尺寸不能定义为相对大小，比如“全尺寸”。详情请参见 第 6.13.1 节“布局的尺寸”。

表 5.1 “尺寸单位”中列出了所有可用的尺寸单位，以及这些单位在 **Sizeable** 接口中对应的常数。

表 5.1. 尺寸单位

<i>Unit.PIXELS</i>	px	像素是基本的硬件单位，指显示设备上的一个物理像素。
<i>Unit.POINTS</i>	pt	磅是一种印刷单位，1 磅通常定义为 1/72 英寸，也就是大约 0.35 毫米。但是，在不同的显示度量标准下，磅的实际显示大小会有显著的不同。
<i>Unit.PICAS</i>	pc	pica 是一种印刷单位，1 pica 定义为 12 磅，也就是 1/7 英寸(译注：原文如此，貌似应该是 1/6 英寸才对)，大约 4.233 毫米。在不同的显示度量标准下，pica 的实际显示大小会有显著的不同。
<i>Unit.EM</i>	em	这是与当前使用的字体相关的一个单位，1 em 等于大写字母 "M" 的宽度。
<i>Unit.EX</i>	ex	这是与当前使用的字体相关的一个单位，1 ex 等于小写字母 "x" 的高度。
<i>Unit.MM</i>	mm	物理长度单位，指显示设备上的 1 毫米。但实际的显示尺寸决定于：1，显示设备本身，2，显示设备在操作系统中的度量标准，3，浏览器。
<i>Unit.CM</i>	cm	物理长度单位，指显示设备上的 1 厘米。但实际的显示尺寸决定于：1，显示设备本身，2，显示设备在操作系统中的度量标准，3，浏览器。
<i>Unit.INCH</i>	in	物理长度单位，指显示设备上的 1 英寸。但实际的显示尺寸决定于：1，显示设备本身，2，显示设备在操作系统中的度量标准，3，浏览器。
<i>Unit.PERCENTAGE</i>	%	相对于可用尺寸的百分比。比如，对最顶层的布局管理器来说 100% 就等于浏览器窗口的全部宽度和高度。百分比值必须在 0 到 100 之间。

如果 **HorizontalLayout** 或 **VerticalLayout** 之内的某个组件在水平(或垂直)方向上为全尺寸，那么这个组件将会占据其他组件没有占据的所有可用空间。详情请参见 第 6.13.1 节“布局的尺寸”。

5.3.10. 管理输入焦点

当用户点击一个组件时，组件就会获得 输入焦点，这个状态使用组件样式来高亮度显示。如果组件允许输入文字，焦点和文字插入位置将通过光标来显示。按下 **Tab** 键会使输入焦点按照 焦点顺序 跳转到下一个组件。

所有的 **Field** 组件都支持输入焦点，**Upload** 也支持输入焦点。

组件的焦点顺序，或者叫 TAB 序号 是通过一个正整数定义的，可以通过 `setTabIndex()` 方法设置这个值，可以通过 `getTabIndex()` 方法得到这个值。TAB 序号由组件所属的页面整体进行管理。因此焦点顺序在不同层次的组件容器之间也可以跳转，比如子窗口和面板。

默认的焦点顺序由组件自然的层次关系顺序决定，也就是与组件添加到父组件的先后顺序一致。组件默认的 TAB 序号是 0。

将组件的 TAB 序号指定为负数，会将这个组件从焦点顺序中彻底删除。

CSS 样式规则

获得焦点的组件将带有一个额外的样式，样式名后缀为 `-focus`。比如，**TextField**，通常的样式为 `v-textfield`，获得焦点后将额外带有一个 `v-textfield-focus` 样式。

比如，下面的例子会让获得焦点的文本输入框显示为蓝色。

```
.v-textfield-focus {
    background: lightblue;
}
```

5.4. Field 组件

Field 是一类特殊的组件，它拥有数据值，并允许用户通过 UI 编辑这个值。图 5.13 “Field 组件” 展示了 Field 组件的继承关系及一些重要的接口和类。

Field 组件的基本框架定义在 **Field** 接口和 **AbstractField** 基类中。**AbstractField** 是所有 Field 组件的基类。这个基类除继承了 **AbstractComponent** 的功能外，还实现了很多其他功能，这些功能定义在 **Property**, **Buffered**, **Validatable**, 和 **Component.Focusable** 接口中。

对 Field 接口和基类的介绍分为以下几节。

5.4.1. Field 接口

Field 接口继承了 **Component** 接口，另外它还继承了 **Property** 接口，因此 Field 拥有数据值。**AbstractField** 是 **Field** 接口唯一的直接实现类。这些接口和类的关系请参见图 5.14 “Field 接口的继承关系图”。

你可以通过 `setValue()` 方法设置 Field 值，通过 `getValue()` 方法获得 Field 值，`getValue()` 方法定义在 **Property** 接口中。值的实际数据类型由各个 Field 组件决定。

Field 接口定义了很多属性，你可以通过相应的取得方法和设置方法来获取和操纵这些属性。

description

所有 Field 都有一个描述信息。请注意这个属性虽然定义在 **Field** (译注：此处有误，应该是 Component) 接口中，但它在 **AbstractField** 类中实现，**AbstractField** 并没有直接实现 **Field** (译注：此处有误，应该是 Component) 接口，而是通过 **AbstractField** (译注：此处有误，应该是 AbstractComponent) 类实现的。

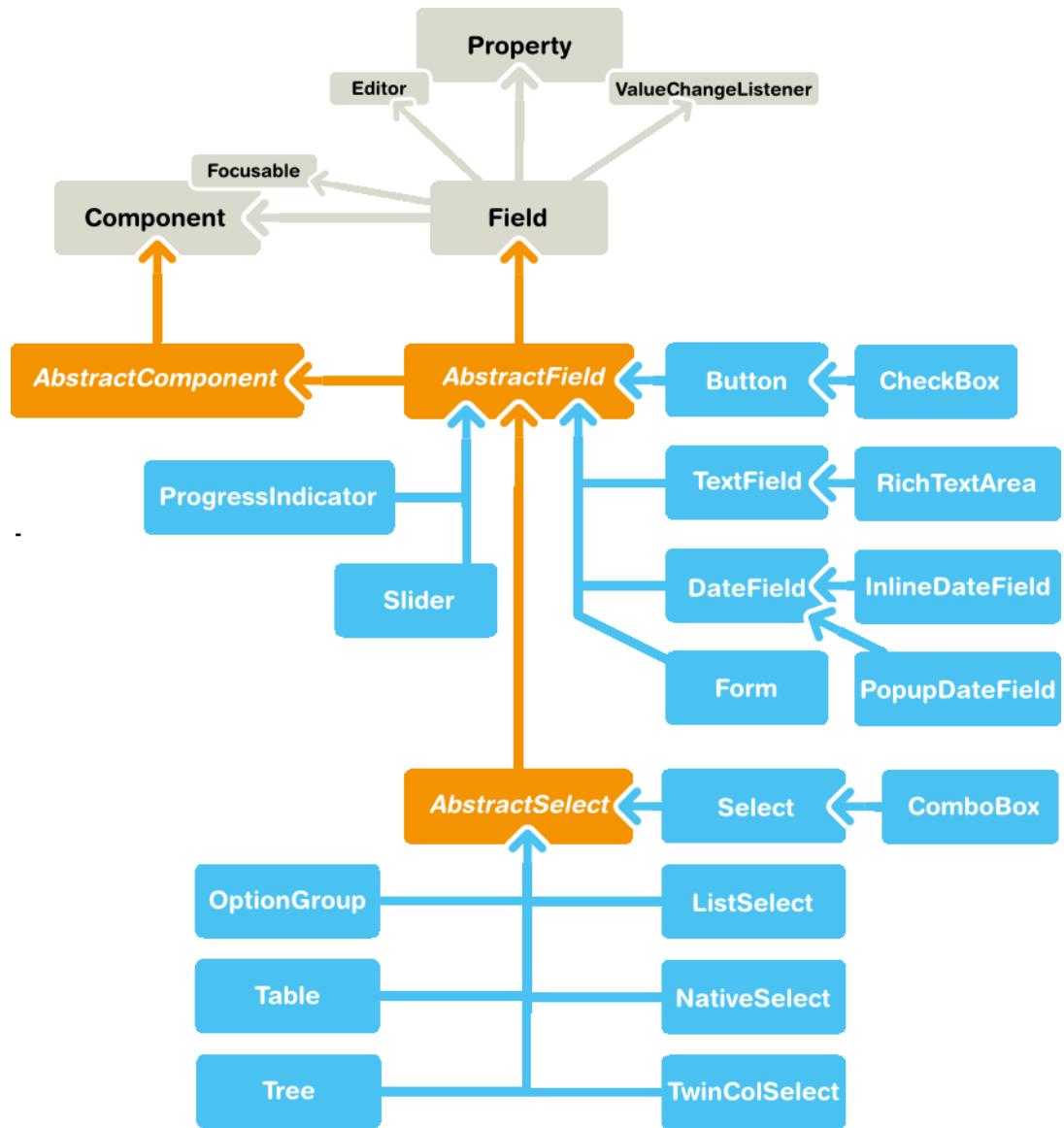
required

当这个特性打开时，“必须项目指示器”(通常是一个 * 符号)会显示在 Field 组件的左侧，上方，或右侧，具体的位置决定于组件所属的布局，以及附件是否带有标题。如果组件的输入校验功能设置为有效，但输入值为空，并且组件被设置了 `requiredError` 属性(详情见后文)，那么会显示一个错误指示器，并且组件的错误信息会被设置为 `error` 属性中指定的文字。输入校验功能无效时，“必须项目指示器”仅仅只是一种提示信息，并不强制要求必须输入数据值。

requiredError

定义了数据值未输入的错误消息，当 Field 组件值必须输入但用户没有输入值时，就会显示这个错误。这个错误消息会被设置为 Field 组件的错误信息，通常显示在提示信息 (tooltip) 中，当鼠标指针移动到错误指示器上时，提示信息就会显示。

图 5.13. Field 组件

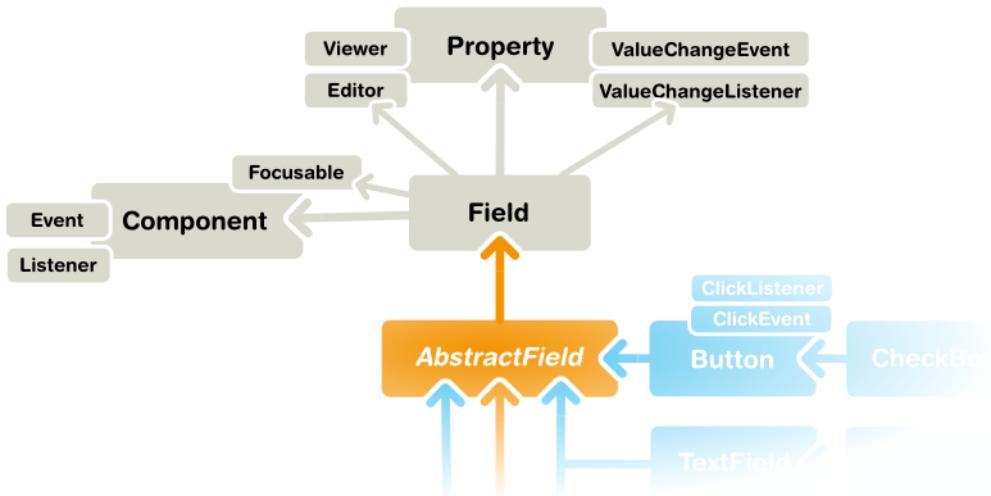


5.4.2. 数据绑定和数据转换

Field 通常会与 Vaadin 数据模型结合在一起。Field 的数据值会以 Field 组件的 **Property** 的形式管理，详情请参见第 8.2 节“属性(Property)”。选择 Field 可以通过 **Container** 接口管理它的选择项。

Field 是某种数据类型的 编辑器。比如，**TextField** 可以编辑 **String** 类型的值。当绑定到数据源时，数据模型中的数据类型可能会与界面输入的值不同，比如可能是 **Integer** 类型。**Converters** 被用来在数据模型与数据的界面表现之间做相互转换。详情请参见第 8.2.3 节“在属性类型与表达之间转换”。

图 5.14. Field 接口的继承关系图



5.4.3. 处理 Field 值的变更

Field 继承了 **Property.ValueChangeListener**, 因此可以监听 **Field** 数据值变化的事件, 还继承了 **Property.Editor**, 因此可以编辑数据值.

当 **Field** 中的数据值发生变化时, 会在这个 **Field** 上触发 **Property.ValueChangeEvent** 事件. 你不应该在继承自 **AbstractField** 的类中实现 `valueChange()` 方法, 因为这个方法已经由 **AbstractField** 实现了. 你应该以事件监听器的方式另外添加类, 在这个监听器类中实现 `valueChange()` 方法.

5.4.4. Field 值的缓存

Field 组件实现了 **Buffered** 和 **BufferedValidatable** 接口. 当通过 `setBuffered(true)` 将 **Field** 的缓存功能设置为有效时, 在 **Field** 的 `commit()` 方法被调用之前, 输入的数据值不会被写入到数据源中去. 调用 `commit()` 方法同时也会执行 **Field** 上的校验器, 如果有任何一个校验器校验失败(并且 `invalidCommitted` 被设置为禁用), 那么值就不会被写入.

```

form.addComponent(new Button("Commit",
    new Button.ClickListener() {
        @Override
        public void buttonClick(ClickEvent event) {
            try {
                editor.commit();
            } catch (InvalidValueException e) {
                Notification.show(e.getMessage());
            }
        }
    }));
  
```

调用 `discard()` 方法会从数据源中重新读入数据, 然后反应到输入界面中.

如果 **Field** 被绑定到 **FieldGroup** 中, 并且 **FieldGroup** 的缓存功能有效, 对 **FieldGroup** 调用 `commit()` 方法会执行 **FieldGroup** 内所有 **Field** 的所有校验器, 如果校验成功, 则所有的 **Field** 数据值都会被写入到数据源中. 详情请参见第 8.4.4 节“Form 的缓冲”.

5.4.5. Field 值校验

Field 组件的输入在语法上或在语义上都有可能是无效的. Field 实现了 `Validatable` 接口, 因此可以使用实现了 `Validator` 接口的 校验器 来检查用户输入是否正确. 你可以通过 `addValidator()` 方法来向 Field 添加校验器.

```
TextField field = new TextField("Name");
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));
layout.addComponent(field);
```

输入校验的失败通过 Field 组件的错误指示器来表示在 UI 中, 详情请参见 第 4.5.1 节 “错误指示器和消息”, 这个功能可以通过 `setValidationVisible(false)` 来禁用. 鼠标指针移动到 Field 上方时, 会显示校验失败的错误消息, 这个错误消息是以参数的形式设置给校验器的. 如果通过 `validate()` 方法显式地进行校验(详情参见后文), 那么在校验失败时会抛出 **InvalidValueException** 例外, 例外信息中也带有前面所说的错误消息. 错误消息中的 `{0}` 会被替换为用户输入的不正确的值.

输入值首先进行类型转换, 校验器的校验对象是转换后的数据类型, 而不是界面上显示的类型. 比如, **IntegerRangeValidator** 要求数据源属性值的数据类型为 **Integer**.

内建的校验器

Vaadin 包含以下内建的校验器. 相应的属性值数据类型在冒号后给出.

BeanValidator

依照 Bean Validation API 1.0 (JSR-303) 定义的注解标准来校验 Bean 的属性. 这个校验器通常不会显式使用, 当将 Field 绑定到 **BeanFieldGroup** 时, 会隐含地创建这个校验器. 使用 Bean 校验需要 Bean Validation API 的实现库. 详情请参见 第 8.4.6 节 “Bean 的校验”.

CompositeValidator

使用逻辑 AND 和 OR 操作将多个校验器组合起来.

DateRangeValidator: Date

检查日期值是否在指定的日期/时间范围之内.

DoubleRangeValidator: Double

检查 Double 值是否在指定的范围之内.

EmailValidator: String

检查字符串值是否合法的邮件地址. 这个校验器使用的文法规则接近于 RFC 822 中定义的邮件地址.

IntegerRangeValidator: Integer

检查整数值是否在指定的范围之内.

NullValidator

检查值是否为 null.

RegexpValidator: String

检查值是否符合指定的正規表达式.

StringLengthValidator: String

检查输入的字符串长度是否都在指定的范围之内.

以上各校验器的详细信息请阅读 API 文档.

自动校验

当 Field 的 validationVisible 为 true 时, 如果 Field 的输入发生变化, 校验器通常会在下一次向服务器发送请求时隐含地执行. 如果 Field 处于 immediate 模式, 它 (以及输入值发生了变化的其他 Field) 会在输入焦点离开时立即进行校验.

```
TextField field = new TextField("Name");
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));
field.setImmediate(true);
layout.addComponent(field);
```

显式校验

当调用 Field 的 validate() 或 commit() 方法时, 会执行校验器.

```
// A field with automatic validation disabled
final TextField field = new TextField("Name");
layout.addComponent(field);

// Define validation as usual
field.addValidator(new StringLengthValidator(
    "The name must be 1-10 letters (was {0})",
    1, 10, true));

// Run validation explicitly
Button validate = new Button("Validate");
validate.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        field.setValidationVisible(false);
        try {
            field.validate();
        } catch (InvalidValueException e) {
            Notification.show(e.getMessage());
            field.setValidationVisible(true);
        }
    }
});
layout.addComponent(validate);
```

实现自定义校验器

你可以实现 Validator 接口, 并实现其中的 validate() 方法, 这样既可创建自定义的校验器. 如果校验失败, 这个方法应该抛出 **InvalidValueException** 或 **EmptyValueException** 异常.

```
class MyValidator implements Validator {
    @Override
    public void validate(Object value)
        throws InvalidValueException {
        if (!(value instanceof String &&
            ((String)value).equals("hello")))
            throw new InvalidValueException("You're impolite");
    }
}
```

```
final TextField field = new TextField("Say hello");
field.addValidator(new MyValidator());
field.setImmediate(true);
layout.addComponent(field);
```

在 **Field Group** 中进行校验

如果 Field 绑定到 **FieldGroup** 上,(详情请参见第 8.4 节“通过 Field 与项目的绑定来创建 Form”),那么调用 FieldGroup 的 commit() 方法,会执行 FieldGroup 内所有 Field 的校验器,只有校验成功时,输入数据才会写入数据源中.

5.5. 选择组件

Vaadin 提供了很多种不同的方式,用于选择一个或多个项目.核心库包含以下几种选择组件,都继承自 **AbstractSelect** 基类:

ComboBox (第 5.16 节)

带下拉列表的文本输入框,用户可以输入文本来查找与之匹配的选项.这个组件还提供一个输入提示,允许用户添加新的选项.

ListSelect (第 5.17 节)

垂直列表框,可以选择一个或多个项目.

NativeSelect (第 5.18 节)

使用浏览器自己的选择框提供选择能力,单选通常使用下拉列表,多选通常使用多行列表.这个组件使用 HTML 的 <select> 元素.

OptionGroup (第 5.19 节)

一组选择项垂直排列,单选模式下选择项表示为 Radio Button,多选模式下选择项表示为 Check Box.

TwinColSelect (第 5.20 节)

并列显示两个列表框,一个是可选项目的列表,另一个是已选项目的列表,用户可以使用控制按钮在这两个列表框中选择项目.

除以上组件之外, **Tree**, **Table**, 以及 **TreeTable** 组件也支持特别形式的项目选择能力.这些组件也继承自 **AbstractSelect** 基类.

5.5.1. 将选择组件绑定到数据

选择组件与 Vaadin 数据模型(详情请参见 第 8 章 组件与数据绑定)结合得非常紧密.在所有的选择组件中,可选项目必须是实现了 **Item** 接口的对象,可选项目包含在 **Container** 之内.

所有的选择组件本身都是数据项目的容器(container),它们只是简单地将所有的容器操作抛发给下层的数据源.你可以在构造函数中给定一个数据容器,也可以通过 setContainerDataSource() 方法来设置.关于这个问题,将在 第 8.5.1 节“容器的基本使用”中详细介绍.

```
// Have a container data source of some kind
IndexedContainer container = new IndexedContainer();
container.addContainerProperty("name", String.class, null);
...

// Create a selection component bound to the container
OptionGroup group = new OptionGroup("My Select", container);
```

如果你没有将一个选择组件绑定到一个数据源, 那么它将会使用一个默认的数据容器. 默认数据容器通常是一个 **IndexedContainer** 或一个 **HierarchicalContainer**.

选择组件的当前选中项目绑定到 **Property** 接口, 因此你可以通过选择组件的值得到它的当前选中项目. 同样的, 选择项目的变更也以值变更事件的方式来处理, 详情将在后续章节中介绍.

5.5.2. 添加新的选择项

使用 **Container** 接口的 `addItem()` 方法可以追加新的选择项. 详情请参见 第 8.5.1 节“容器的基本使用”.

```
// Create a selection component
ComboBox select = new ComboBox("My ComboBox");

// Add items with given item IDs
select.addItem("Mercury");
select.addItem("Venus");
select.addItem("Earth");
```

`addItem()` 方法创建空的 **Item**, 项目通过它的 **项目 ID(item identifier)** (IID) 对象来区分, 项目 ID 通过参数给定. 项目 ID 默认也会用作项目的标题, 详情请见后文.

我们要强调的是 `addItem()` 是一个工程方法, 它接受的参数是项目 ID, 而不是真实的项目 - 项目是这个方法的返回值. 项目的类型由容器决定, 而且在大多数选择组件中类型并不重要, 因为项目仅有项目 ID, 其他所有属性都不会被使用(但在 **Table** 中例外).

项目 ID 通常是字符串, 这种情况下它可以用作项目标签, 但项目 ID 也可以是任何对象类型. 我们完全可以使用整数作为项目 ID, 然后使用 `setItemCaption()` 方法明确设定项目标签. 你也可以使用无参数的 `addItem()` 方法来添加项目, 这个方法会返回一个自动生成的项目 ID.

```
// Create a selection component
ComboBox select = new ComboBox("My Select");

// Add an item with a generated ID
Object itemId = select.addItem();
select.setItemCaption(itemId, "The Sun");

// Select the item
select.setValue(itemId);
```

某些容器类型可能支持传递真实数据对象来添加项目. 比如, 你可以使用 `addBean()` 方法向 **BeanItemContainer** 添加项目. 这样的容器实现可以使用一个独立的项目 ID 对象, 也可以使用真实数据对象本身作为项目 ID, `addBean()` 方法正是如此. 后一种情况下你不能使用默认的方式得到项目标签; 取得项目标签的其他方法见后文.

下面的小节介绍取得项目标签的不同方式.

5.5.3. 项目标题

选择组件中显示的项目标签可以使用 `setItemCaption()` 方法明确指定, 也可以通过项目 ID 或项目属性自动获得. 标题的取得方式由 标题模式 决定, 标题模式定义在 **AbstractSelect.ItemCaptionMode** 枚举型内, 你可以使用 `setItemCaptionMode()` 方法来设置标题模式. 默认的模式是 `EXPLICIT_DEFAULTS_ID`, 这个模式下, 除非明确给定标题, 否则就使用项目 ID 作为标题.

除标题外, 项目还可以有图标. 图标使用 `setItemIcon()` 方法设置.

标题模式定义在 **ItemCaptionMode** 中, 如下:

选择组件的标题模式

EXPLICIT_DEFAULTS_ID

这是默认的标题模式, 这是一种很灵活的模式, 可以用于大多数情况。默认会使用项目 ID 作为标题。项目 ID 对象不必一定是字符串类型; 标题文字可以通过 `toString()` 方法得到。如果使用 `setItemCaption()` 方法明确指定了标题, 那么将会使用指定的标题, 而不是项目 ID。

```
// Create a selection component
ComboBox select = new ComboBox("Moons of Mars");
select.setItemCaptionMode(ItemCaptionMode.EXPLICIT_DEFAULTS_ID);

// Use the item ID also as the caption of this item
select.addItem(new Integer(1));

// Set item caption for this item explicitly
select.addItem(2); // same as "new Integer(2)"
select.setItemCaption(2, "Deimos");
```

EXPLICIT

这个模式下, 必须使用 `setItemCaption()` 方法明确指定标题。如果没有指定, 标题将为空。标题为空的项目仍然会显示在选择组件中。如果带有图标, 项目在界面上就是可见的了。

ICON_ONLY

这个模式下只显示图标, 标题将被隐藏。

ID

这个模式使用项目 ID 对象的字符串表达作为标题。当项目 ID 为字符串型时, 这个模式是很有用的, 如果项目 ID 是一个复杂对象, 但有字符串形式的表达, 也可以使用这个模式。比如:

```
ComboBox select = new ComboBox("Inner Planets");
select.setItemCaptionMode(ItemCaptionMode.ID);

// A class that implements toString()
class PlanetId extends Object implements Serializable {
    String planetName;

    PlanetId (String name) {
        planetName = name;
    }
    public String toString () {
        return "The Planet " + planetName;
    }
}

// Use such objects as item identifiers
String planets[] = {"Mercury", "Venus", "Earth", "Mars"};
for (int i=0; i<planets.length; i++)
    select.addItem(new PlanetId(planets[i]));
```

INDEX

这个模式使用项目的索引顺序作为标题。只有在数据源实现了 **Container.Indexed** 接口时才可以使用这个模式。否则, 组件将抛出一个 **ClassCastException** 异常。

AbstractSelect 类本身没有实现这个接口，因此这个模式必须与一个独立的数据源组合使用，比如 **IndexedContainer**.

ITEM

这个模式下，通过项目的 `toString()` 方法得到的 **String** 型表达，被用作标题。这个模式主要适用于自定义 **Item** 类的情况，此时还需要自定义 **Container** 类，并将它用作选择组件的数据源。如果你希望使用某个特定的属性作为项目标题，这个模式就很有用。

PROPERTY

这个模式下，项目标题通过项目属性值的 **String** 表达获得，用作标题的属性 ID 使用 `setItemCaptionPropertyId()` 方法来指定。当你使用一个容器作为选择组件的数据源，如果你希望使用某个特定的属性作为项目标题，这个模式就很有用。

下面的例子中，我们将一个选择组件绑定到一个 Bean 容器，并使用 Bean 的一个属性作为选择项目的标题。

```
/** A bean with a "name" property. */
public class Planet implements Serializable {
    int id;
    String name;

    public Planet(int id, String name) {
        this.id = id;
        this.name = name;
    }

    ... setters and getters ...
}

public void captionproperty(VBoxLayout layout) {
    // Have a bean container to put the beans in
    BeanItemContainer<Planet> container =
        new BeanItemContainer<Planet>(Planet.class);

    // Put some example data in it
    container.addItem(new Planet(1, "Mercury"));
    container.addItem(new Planet(2, "Venus"));
    container.addItem(new Planet(3, "Earth"));
    container.addItem(new Planet(4, "Mars"));

    // Create a selection component bound to the container
    ComboBox select = new ComboBox("Planets", container);

    // Set the caption mode to read the caption directly
    // from the 'name' property of the bean
    select.setItemCaptionMode(ItemCaptionMode.PROPERTY);
    select.setItemCaptionPropertyId("name");

    ...
}
```

5.5.4. 取得和设置当前选中项目

选择组件的当前选中项目以组件的属性(**Property** 接口)的方式提供。这个属性的值是项目 ID 对象，ID 对象标识了被选中的项目是哪个。你可以使用 **Property** 接口的 `getValue()` 方法来得到当前选中项目的 ID。

使用对应的 `setValue()` 方法可以选中某个项目。在多选模式下，属性值应该是项目 ID 组成的不可变集合。如果没有项目被选中，单选模式下的属性值应该是 `null`，多选模式下应该是空集合。

当没有项目被选中时, **ComboBox** 和 **NativeSelect** 将会显示空的选中项. 也就是 `null` 选择项目 `ID`. 你可以使用 `setNullSelectionItemId()` 方法设置一个替代的 `ID`. 设置替代的 `null` `ID` 只会影响界面表现文字; 没有项目被选中时, `getValue()` 方法在单选模式下仍然会返回 `null` 值, 在多选模式下仍然会返回空集合.

5.5.5. 处理选择项的变化事件

当前选中项目的 `ID` 将被设置为选择组件的属性. 你可以通过组件本身的 **Property** 接口的 `getValue()` 方法访问它. 还可以使用 **Property.ValueChangeListener** 监听器来处理选择项的变化事件, **ValueChangeEvent** 事件的属性将是被选中的项目, 可以通过 `getProperty()` 方法访问.

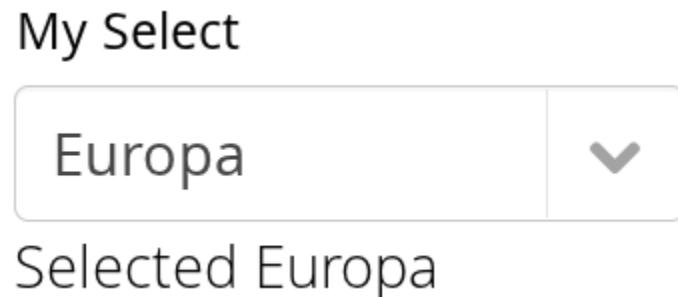
```
// Create a selection component with some items
ComboBox select = new ComboBox("My Select");
select.addItems("Io", "Europa", "Ganymedes", "Callisto");

// Handle selection change
select.addValueChangeListener(event -> // Java 8
    layout.addComponent(new Label("Selected " +
        event.getProperty().getValue())));

```

用户操作这个 `ComboBox` 之后的运行结果, 参见 图 5.15 “被选中的项目”.

图 5.15. 被选中的项目



5.5.6. 允许添加新项目

某些选择组件可以允许用户添加新项目. 目前只有 **ComboBox** 支持这个功能, 当用户输入一个值, 并按下 **Enter** 键时, 新项目就被添加了. 你需要使用 `setNewItemAllowed(true)` 方法打开这个模式. 有可能还需要将组件设置为立即模式, 否则新项目可能不会在用户操作这个组件时立即添加, 而是在其他操作之后才出发服务器请求.

```
myselect.setNewItemAllowed(true);
myselect.setImmediate(true);
```

用于添加新项目的用户界面由各个选择组件决定. 通常的 **ComboBox** 组件允许你在复合框内直接输入新项目内容, 然后按下 **Enter** 键就可以添加这个新项目.

当组件为只读时, 或者组件绑定的 **Container** 不支持添加新项目时, 向组件添加新项目是不可能的, 尝试添加新项目可能会导致一个例外.

处理新项目

新项目的添加由 **NewItemHandler** 负责处理, 它的 `addNewItem()` 方法将接受项目标题字符串作为参数. 默认实现类是 **DefaultNewItemHandler**, 它将检查组件的只读状态, 将用户输入的标

题作为项目 ID 来添加新项目，如果选则组件从项目属性中获取项目标题的话，它还要将标题复制到对应的属性中。它还会选中这个项目。这个默认实现类可能不能适用于所有的容器类型，这种情况下你需要创建自定义的处理器。比如，**BeanItemContainer** 要求项目使用 Bean 对象本身作为 ID，而不是使用字符串作为 ID。

```
// Have a bean container to put the beans in
final BeanItemContainer<Planet> container =
    new BeanItemContainer<Planet>(Planet.class);

// Put some example data in it
container.addItem(new Planet(1, "Mercury"));
container.addItem(new Planet(2, "Venus"));
container.addItem(new Planet(3, "Earth"));
container.addItem(new Planet(4, "Mars"));

final ComboBox select =
    new ComboBox("Select or Add a Planet", container);
select.setNullSelectionAllowed(false);

// Use the name property for item captions
select.setItemCaptionPropertyId("name");

// Allow adding new items
select.setNewItemHandler(true);
select.setImmediate(true);

// Custom handling for new items
select.setNewItemHandler(new NewItemHandler() {
    @Override
    public void addNewItem(String newItemCaption) {
        // Create a new bean - can't set all properties
        Planet newPlanet = new Planet(0, newItemCaption);
        container.addBean(newPlanet);

        // Remember to set the selection to the new item
        select.select(newPlanet);

        Notification.show("Added new planet called " +
            newItemCaption);
    }
});
```

5.5.7. 复数选择

某些选择组件，比如 **OptionGroup** 和 **ListSelect** 支持多选模式，你可以使用 `setMultiSelect()` 方法打开多选模式。对于 **TwinColSelect** 来说，它本身的目的就是选择多个项目，因此它默认就是多选模式。

```
myselect.setMultiSelect(true);
```

在单选模式下，组件的属性值就代表它目前选中的项目。但在多选模式下，属性值将是当前被选中的项目 ID 组成的 **Collection**。你可以使用 `getValue()` 和 `setValue()` 方法读写属性值。

选择状态的变化将激发 **ValueChangeEvent** 事件，可以使用 **Property.ValueChangeListener** 监听器来处理这个事件。通常应该使用 `setImmediate(true)`，当用户修改选择状态时立即激发事件。下面的例子演示如何使用监听器来处理选择的变化。

```

// A selection component with some items
ListSelect select = new ListSelect("My Selection");
select.addItems("Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn", "Uranus", "Neptune");

// Multiple selection mode
select.setMultiSelect(true);

// Feedback on value changes
select.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            // Some feedback
            layout.addComponent(new Label("Selected: " +
                event.getProperty().getValue().toString()));
        }
});
select.setImmediate(true);

```

5.5.8. 项目图标

你可以使用 `setItemIcon()` 方法为每个项目设置图标, 或者使用 `setItemIconPropertyId()` 方法指定一个项目属性来提供图标资源, 具体方法与我们前面讲过的"通过属性来指定标题"是类似的. 注意, 图标在 **NativeSelect** 组件, **TwinColSelect** 组件, 以及其他某些选择组件或模式下是不支持的. 这是因为 HTML 不支持在原生的 `select` 元素旁边显示图片. Icons are also not really visually applicable(译注: 这句不理解, 待校).

5.6. 组件的扩展

组件和 UI 可以拥有动态关联在其上的扩展(extension). 很多插件实际上都是扩展.

如何扩充一个组件的功能由具体的扩展来决定. 扩展通常会有一个 `extend()` 方法, 参数是需要扩充的对象组件.

```

TextField tf = new TextField("Hello");
layout.addComponent(tf);

// Add a simple extension
new CapsLockWarning().extend(tf);

// Add an extension that requires some parameters
CSValidator validator = new CSValidator();
validator.setRegExp("[0-9]*");
validator.setError("Must be a number");
validator.extend(tf);

```

如何开发自定义的扩展, 详情请参阅 第 16.7 节“组件与 UI 扩展”.

5.7. Label

Label 组件显示一段不可编辑的文本. 这段文本可以用来显示简短的标签, 也可以用来显示长文本, 比如一个段落. 通过设置 **Label** 组件的内容模式, 文本可以使用 HTML 格式, 也可以使用预格式化的文本.

可以在构造函数中指定 `Label` 的文本内容, 这是最方便的方式, 如下例所示. `Label` 的默认宽度为 100%, 因此包含它的布局组件宽度必须为固定值.

```
// A container that is 100% wide by default
VerticalLayout layout = new VerticalLayout();

Label label = new Label("Labeling can be dangerous");
layout.addComponent(label);

Label implements the Property interface to allow accessing the text value, so you can get and set the text with getValue() and setValue().

// Get the label's text to initialize a field
TextField editor = new TextField(null, // No caption
                                label.getValue());

// Change the label's text
editor.addValueChangeListener(event -> // Java 8
    label.setValue(editor.getValue()));
editor.setImmediate(true); // Send on Enter
```

Label 还支持绑定到一个属性(property)数据源, 详情将在 第 5.7.4 节“数据绑定”中介绍。但是, 这时无法通过 **label** 来设置数据值, 因为 **Label** 不是一个 **Property.Editor**, 不允许向绑定的属性写入数据。

虽然 **Label** 是一段文本, 通常用做标题文字, 但它也是一个通常的组件, 因此它本身也有标题, 你可以通过 `setCaption()` 方法来设置标题。和大多数其他组件一样, **Label** 的标题也是由包含它的布局组件来管理的。

5.7.1. 文本的宽度与折行

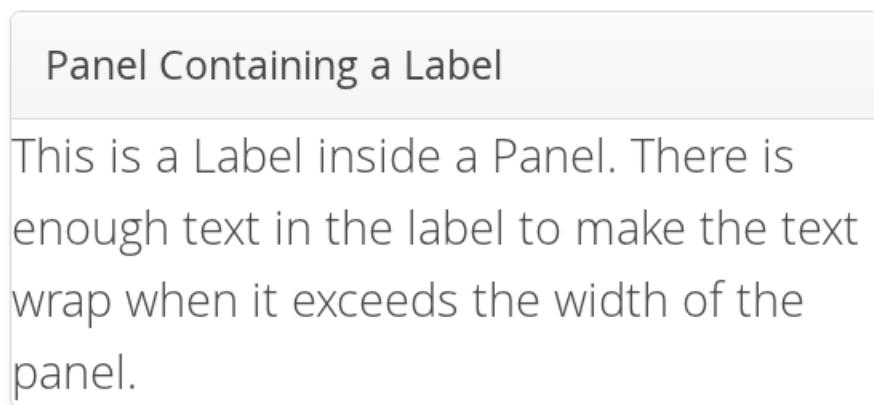
Label 的默认宽度为 100%, 因此包含它的容器布局必须有一个给定的宽度。如果 **Label** 中文本的宽度超出了 **Label** 的宽度, 文本将会折行。某些布局组件默认宽度为未指定, 比如 **HorizontalLayout**, 因此你需要特别注意。

```
// A container with a defined width.
Panel panel = new Panel("Panel Containing a Label");
panel.setWidth("300px");

panel.setContent(
    new Label("This is a Label inside a Panel. There is " +
              "enough text in the label to make the text " +
              "wrap when it exceeds the width of the panel."));
```

上例中 **Panel** 的尺寸为固定值, **Label** 的宽度为默认值 100%, 因此 **Label** 中的文字将折行显示, 宽度与 **Panel** 相适应, 运行结果参见 图 5.16 “Label 组件”。

图 5.16. Label 组件



将 **Label** 宽度设置为未定义，将导致它的内容不折行，因为此时 Label 的宽度将由文本内容的宽度决定。如果 Label 所属布局组件的宽度有指定，**Label** 将在水平方向上超过布局组件的显示范围，超出的部分通常会被截断，不显示。

5.7.2. 内容模式

Label 内容的格式由 内容模式 决定。默认情况下，内容被认为是纯文字，任何特殊的 XML 字符都将被适当地转换，以 HTML 形式在浏览器内正确地显示 Label 内容。内容模式可以在构造函数中设置，也可以通过 `setContentMode()` 方法设置，可指定的值由 com.vaadin.shared.ui.label 包的 **ContentMode** 枚举型定义：

TEXT

这种模式下 Label 只包含纯文本。文本中可以包含任意字符，包括 XML 和 HTML 中的特殊字符 <, >, 和 &，这些特殊字符在显示组件时会被转义为适当的 HTML。这个模式是默认的内容模式。

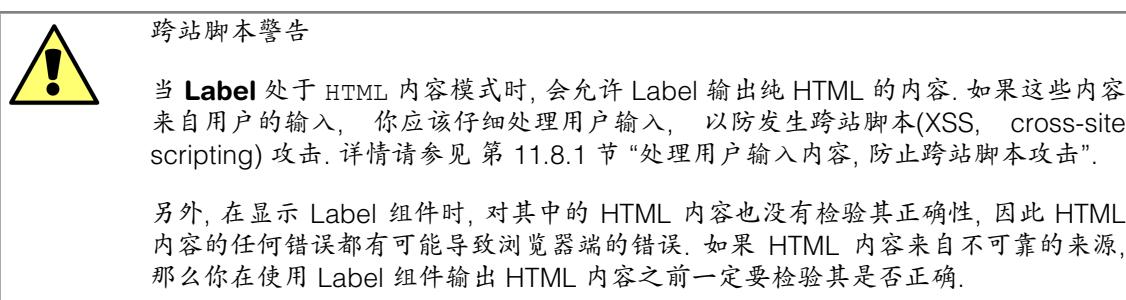
PREFORMATTED

这种模式下 Label 包含的是格式化文本。默认情况下会显示为定宽字体。格式化文本可以包含换行符，Java 语言中换行符(ASCII 0x0a)书写为转义字符串 \n，也可以包含制表符，Java 语言中制表符(ASCII 0x09)书写为转义字符串 \t。

HTML

这个模式下 Label 包含的是 HTML。

请注意 HTML 内容模式下，存在以下安全性和校验问题。



下面的例子演示了 **Label** 不同内容模式的使用方法。

```
Label textLabel = new Label(
    "Text where formatting characters, such as \\n, " +
    "and HTML, such as <b
>here</b
>, are quoted.",
    ContentMode.TEXT);

Label preLabel = new Label(
    "Preformatted text is shown in an HTML <pre
> tag.\n" +
    "Formatting such as\n" +
    "  * newlines\n" +
    "  * whitespace\n" +
    "and such are preserved. HTML tags, \n"+
    "such as <b
>bold</b
>, are quoted.",
    ContentMode.PREFORMATTED);

Label htmlLabel = new Label(
    "In HTML mode, all HTML formatting tags, such as \n" +
    "<ul
>" +
    "  <li
><b
>bold</b
></li
>" +
    "  <li
>itemized lists</li
>" +
    "  <li
>etc.</li
>" +
    "</ul
> " +
    "are preserved.",
    ContentMode.HTML);
```

上例的显示效果见图 5.17 “Label 的各种内容模式”。

图 5.17. Label 的各种内容模式

TEXT	Text where formatting characters, such as \n, and HTML, such as here, are quoted.
PREFORMATTED	Preformatted text is shown in an HTML <pre> tag. Formatting such as * newlines * whitespace and such are preserved. HTML tags, such as bold, are quoted.
HTML	In HTML mode, all HTML formatting tags, such as <ul style="list-style-type: none">• bold• itemized lists• etc. are preserved.

5.7.3. 用 Label 来控制空白

你可以使用 **Label** 在布局中生成垂直或水平的空白。如果你在垂直布局中需要一个空"行", 那么使用一个文本为空的 Label 是不够的, 因为它的高度将会为 0. 文本为一个空格的 Label 也会产生同样的结果。你需要使用不换行空格(non-breaking space)字符, 可以是 或 :

```
layout.addComponent(new Label("&nbsp;", ContentMode.HTML));
```

使用 *ContentMode.PREFORMATTED* 模式也有同样效果; 格式化文本中的空格在垂直布局中不会消失。在 **HorizontalLayout** 中, 如果 Label 使用比例字体(proportional font), 则空白字符的宽度将是不可预测的, 你可以使用格式化文本内容模式来添加 em 单位宽度的空白。

如果你希望空白区域拥有可调节的宽度或高度, 那么你可以使用空白 Label, 并指定它的宽度或高度。比如, 如果要在 **VerticalLayout** 内创建垂直空白的话:

```
Label gap = new Label();
gap.setHeight("1em");
verticalLayout.addComponent(gap);
```

你可以创造一个灵活扩展的空白区域, 方法是使用一个空白 Label, 设置它的高度或宽度为相对大小 100%, 并将它设置为随布局一起扩展。

```
// A wide component bar
HorizontalLayout horizontal = new HorizontalLayout();
horizontal.setWidth("100%");

// Have a component before the gap (a collapsing cell)
Button button1 = new Button("I'm on the left");
horizontal.addComponent(button1);
```

```
// An expanding gap spacer
Label expandingGap = new Label();
expandingGap.setWidth("100%");
horizontal.addComponent(expandingGap);
horizontal.setExpandRatio(expandingGap, 1.0f);

// A component after the gap (a collapsing cell)
Button button2 = new Button("I'm on the right");
horizontal.addComponent(button2);
```

5.7.4. 数据绑定

虽然 **Label** 不是一个 Field 组件, 但它是一个 Property.Viewer, 可以绑定到 property 数据源, 详情请参见 第 8.2 节 “属性(Property)”. 你可以在构造函数中指定数据源, 或使用 `setPropertyDataSource()` 方法.

```
// Some property
ObjectProperty<String> property =
    new ObjectProperty<String>("some value");

// Label that is bound to the property
Label label = new Label(property);
```

此外, **Label** 也是一个 Property, 因此你可以使用属性编辑器来编辑它的值, 比如使用一个 Field:

```
Label label = new Label("some value");
TextField editor = new TextField();
editor.setPropertyDataSource(label);
editor.setImmediate(true);
```

但是, **Label** 不是一个 Property.Editor, 因此它绑定到数据源之后, 将是只读的. 因此, 你不能通过 **Label** 的 `setValue()` 方法来设置绑定的数据源的值, 也不能将 Label 绑定到一个编辑 Field, 这种情况下对值的写操作将被代理到 Label 之上.

5.7.5. CSS 样式规则

```
.v-label { }
pre { } /* In PREFORMATTED content mode */
```

Label 组件的最外层样式为 `v-label`. 在 PREFORMATTED 内容模式下, 文本将被包装在一个 `<pre>` 元素之内.

5.8. Link

Link 组件可以创建超链接(hyperlink). 链接目标地址以资源对象的方式表达, 详情请参见 第 4.4 节 “图片及其他资源”. **Link** 是通常的 HTML 链接, 也就是一个由浏览器处理的 `<a href>` 元素. 与点击 **Button** 不同, 点击 **Link** 不会在服务器端触发事件.

指向任意 URL 的链接可以使用 **ExternalResource** 生成, 如下例:

```
// Textual link
Link link = new Link("Click Me!",
    new ExternalResource("http://vaadin.com/"));
```

你可以使用 `setIcon()` 方法来生成图片链接, 如下例:

```
// Image link
Link iconic = new Link(null,
    new ExternalResource("http://vaadin.com/"));
iconic.setIcon(new ThemeResource("img/nicubunu_Chain.png"));

// Image + caption
Link combo = new Link("To appease both literal and visual",
    new ExternalResource("http://vaadin.com/"));
combo.setIcon(new ThemeResource("img/nicubunu_Chain.png"));
```

上例的运行结果见图 5.18 “**Link** 示例”. 你可以对图标元素添加一个 “display: block” 样式, 将标签放置到图标的下方.

图 5.18. Link 示例



在上例中使用的简单构造方法, 会将链接对象资源在当前窗口中打开. 如果通过构造方法参数指定, 或者通过 `setTargetName()` 方法来指定链接的打开对象窗口, 你可以将链接对象资源在其他窗口中打开, 比如浏览器的弹出窗口/TAB. 由于对象窗口名称是由浏览器管理的 HTML `target` 字符串, 因此对象窗口可以是任何窗口, 包括不被应用程序管理的窗口. 你还可以使用带下划线的特殊窗口名, 比如 `_blank`, 可以在新的浏览器窗口或 TAB 中打开链接.

```
// Hyperlink to a given URL
Link link = new Link("Take me a away to a faraway land",
    new ExternalResource("http://vaadin.com/"));

// Open the URL in a new window/tab
link.setTargetName("_blank");

// Indicate visually that it opens in a new window/tab
link.setIcon(new ThemeResource("icons/external-link.png"));
link.addStyleName("icon-after-caption");
```

链接的图标通常在标题之前. 你可以让它出现在标题右侧, 方法是在 HTML 容器元素中颠倒文字方向.

```
/* Position icon right of the link caption. */
.icon-after-caption {
    direction: rtl;
}
/* Add some padding around the icon. */
.icon-after-caption .v-icon {
    padding: 0 3px;
}
```

上面例子的运行结果见图 5.19 “打开新窗口的链接”.

图 5.19. 打开新窗口的链接



使用目标窗口名 `_blank`, 会打开一个通常的浏览器新窗口. 如果你希望打开一个弹出窗口(或 TAB), 你需要使用 `setTargetWidth()` 和 `setTargetHeight()` 方法指定窗口的大小. 你可以使用 `setTargetBorder()` 方法控制窗口的边框风格, 这个方法的参数可以是以下窗口边框风格: `TARGET_BORDER_DEFAULT`, `TARGET_BORDER_MINIMAL`, 以及 `TARGET_BORDER_NONE`. 但最终的运行结果由浏览器决定.

```
// Open the URL in a popup
link.setTargetName("_blank");
link.setTargetBorder(Link.TARGET_BORDER_NONE);
link.setTargetHeight(300);
link.setTargetWidth(400);
```

除 **Link** 组件外, Vaadin 还允许其他方式创建超链接. **Button** 组件有一种 `Reindeer.BUTTON_LINK` 样式, 使得按钮的外观类似于超链接, 但它可以在服务器端的监听器中处理点击事件, 而不是由浏览器自行处理. 此外, 你还可以使用 HTML 内容模式的 **Label** 来创建超链接(或者其他任何 HTML 内容).

CSS 样式规则

```
.v-link { }
a { }
.v-icon {}
span {}
```

Link 组件的最外层样式为 `v-link`. 它是包含 `<a href>` 超链接元素的根元素. 在链接之内是图标, 带有 `v-icon` 样式, 以及标题, 在 `span` 内.

超链接有大量的 伪类(*pseudo-class*), 分别在不同的场合有效. 未访问过的链接带有 `a:link` 类, 已访问过的链接则带有 `a:visited`. 鼠标指针移动到链接上方时, 链接将带有 `a:hover` 类, 当鼠标指针在链接上方按下时, 链接将带有 `a:active` 类. 如果在 CSS 选择器中组合使用这些伪类时, 请注意 `a:hover` 必须出现在 `a:link` 和 `a:visited` 之后, `a:active` 必须出现在 `a:hover` 之后.

5.9. TextField

TextField 是最常用的 UI 组件之一. 它是 **Field** 组件的一种, 可通过键盘来输入文本值.

下面的例子创建一个简单的 **TextField**:

```
// Create a text field
TextField tf = new TextField("A Field");

// Put some initial content in it
tf.setValue("Stuff in the field");
```

运行结果参见图 5.20 “**TextField Example**”.

图 5.20. **TextField Example**

A Field

Stuff in the field

与其他大多数 Field 组件一样, 值的变化由 **Property.ValueChangeListener** 监听器处理. 值可以通过 TextField 的 `getValue()` 方法直接取得, 如下面的例子所示, 也可以通过事件绑定的 property 得到.

```
// Handle changes in the value
tf.addValueChangeListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        // Assuming that the value type is a String
        String value = (String) event.getProperty().getValue();

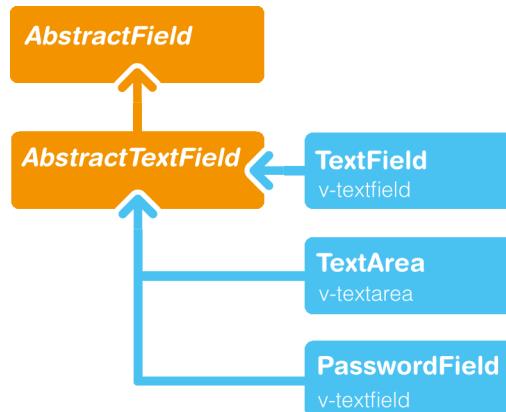
        // Do something with the value
        Notification.show("Value is: " + value);
    }
});

// Fire value changes immediately when the field loses focus
tf.setImmediate(true);
```

和其他的事件监听器一样, 在 Java 8 中, 你可以使用单个参数的 lambda 表达式来处理事件.

TextField 的大多数 API 定义在 **AbstractTextField** 中, 从这个类可以继承出各种不同类型的文本输入组件, 比如 富文本编辑器, 这些文本输入组件的功能与单行文本框并不完全相同.

图 5.21. 文本编辑框的类关系



5.9.1. 数据绑定

TextField 用于编辑 **String** 类型值, 但你可以将它绑定到任意的属性类型上, 只要存在适当的转换器, 详情请参见 第 8.2.3 节 “在属性类型与表达之间转换”.

```
// Have an initial data model. As Double is unmodifiable and
// doesn't support assignment from String, the object is
// reconstructed in the wrapper when the value is changed.
Double trouble = 42.0;

// Wrap it in a property data source
final ObjectProperty<Double> property =
    new ObjectProperty<Double>(trouble);

// Create a text field bound to it
// (StringToDoubleConverter is used automatically)
TextField tf = new TextField("The Answer", property);
tf.setImmediate(true);
```

```
// Show that the value is really written back to the
// data source when edited by user.
Label feedback = new Label(property);
feedback.setCaption("The Value");
```

当你将一个 **Table** 设为可编辑模式时, 或使用 **FieldGroup** 创建 Field 时, **DefaultFieldFactory** 默认就会创建 **TextField** 用作大多数属性类型的编辑器。你经常会需要编写一个自定义的 Factory, 以便自定义 TextField 的创建过程, 设置 Field 提示信息, 校验, 格式, 等等。

关于数据绑定, 详情请参见 第 8 章 组件与数据绑定, 关于 **Table** 的 Field Factory, 详情请参见 第 5.21.3 节 “在 Table 内编辑数据值”, 关于 Form, 详情请参见 第 8.4 节 “通过 Field 与项目的绑定来创建 Form”。

5.9.2. 字符串长度

`setMaxLength()` 方法可以设置字符串输入的最大长度, 浏览器将禁止用户输入更长的文字。作为一种安全功能, 输入的字符串会在服务器端自动截断, 因为最大长度限制在客户端有可能被恶意用户绕过。最大长度属性定义在 **AbstractTextField** 中。

注意, 最大长度设置不会影响 Field 的宽度。与其他组件一样, 你可以使用 `setWidth()` 方法来设置宽度。建议使用 `em` 单位的宽度, 因为这个单位与当前使用的字体相关, 显示效果较好。在 HTML 中没有标准的方法可以将宽度设置为恰好等于多少个字母 (使用等宽字体时)。你可以绕过这个限制, 方法是将 TextField 放在一个未定义宽度的 **VerticalLayout** 内, 同时再放一个未定义宽度的 **Label**, Label 中包含一串示例文本, 然后将 TextField 的宽度设置为 100%。布局管理器将根据 Label 的宽度决定自己的宽度, 然后 TextField 将与布局管理器的宽度相同。

5.9.3. 处理 Null 值

与任何一种 Field 相同, **TextField** 的值可以被设置为 `null`。如果你创建了一个新的 Field 但没有为它设置值, 或者将 Field 值绑定到一个允许 Null 值的数据源, 就会发生这样的情况。这种情况下, 你可能希望为 Null 值显示某个特别的值。你可以使用 `setNullRepresentation()` 方法设置 Null 值的表现。一般可以用空字符串作为 Null 值的表现, 除非你希望将 Null 值与空字符串明确地区分开。默认的 Null 值表现是 “`null`”, 这个表现的本来目的是提示你有可能忘记了正确地初始化数据对象。

`setNullSettingAllowed()` 方法控制用户能否使用 Null 值表现来输入 Null 值。如果设定为 `false`, 默认值就是如此, 那么用户输入的 Null 值表现字符串会被当作真实的字符串输入, 而不会被识别为 Null 值。这种默认假设是一种安全措施, 因为数据源可能不接受 Null 值。

```
// Have a property with null value
ObjectProperty<Double> dataModel =
    new ObjectProperty<Double>(new Double(0.0));
dataModel.setValue(null); // Have to set it null here

// Create a text field bound to the null data
TextField tf = new TextField("Field Energy (J)", dataModel);
tf.setNullRepresentation("-- null-point --");

// Allow user to input the null value by its representation
tf.setNullSettingAllowed(true);
```

上例中的 **Label**, 绑定到 **TextField** 的值上, 讲 Null 值显示为空字符串。上例的运行结果参见图 5.22 “Null 值的表现”。

图 5.22. Null 值的表现

Field Energy (J)

-- null-point --

5.9.4. 文本变更事件

当 TextField 值发生变更时，你通常希望立即收到一个变更事件。*immediate* 模式并不是字面意思那样的立即模式，因为仅在 Field 失去输入焦点后才会发生变更事件。另一种极端情况是，对每一次按键都使用键盘事件，但这会让输入过程慢到不可忍受，而且对于大多数目的来说，处理每个按键的事件也太过于复杂。文本变更事件会在输入发生后异步的发送到服务器，事件处理过程不会阻碍客户端的输入过程。

文本变更事件通过 **TextChangeListener** 监听器来接收，如下例所示，我们演示如何实现一个文字长度计数器：

```
// Text field with maximum length
final TextField tf = new TextField("My Eventful Field");
tf.setValue("Initial content");
tf.setMaxLength(20);

// Counter for input length
final Label counter = new Label();
counter.setValue(tf.getValue().length() +
    " of " + tf.getMaxLength());

// Display the current length interactively in the counter
tf.addTextChangeListener(new TextChangeListener() {
    public void textChange(TextChangeEvent event) {
        int len = event.getText().length();
        counter.setValue(len + " of " + tf.getMaxLength());
    }
});

// The lazy mode is actually the default
tf.setTextChangeEventMode(TextChangeEventMode.LAZY);
```

运行结果参见图 5.23 “文本变更事件”。

图 5.23. 文本变更事件

My Eventful Field

Initial content|

15 of 20

文本变更事件的模式决定了变更以什么样的速度传送到服务器并触发服务器端事件。当用户输入速度很快时，比较慢速的变更事件可以一次性传输比较大的变更，因此可以减少对服务器的请求次数。

你可以使用 **TextField** 的 `setTextChangeEventMode()` 方法设置它的文本变更事件模式。合法的模式定义在 **TextChangeEventMode** 枚举型中，说明如下：

`TextChangeEventMode.LAZY`(默认)

当文本编辑发生暂停时触发事件。暂停的长度可以通过 `setInputEventTimeout()` 方法设置。与 `TIMEOUT` 模式一样，在 **ValueChangeEvent** 事件之前会强制发生一个文本变更事件，即使用户在输入文本时没有出现暂停。

这是默认模式。

`TextChangeEventMode.TIMEOUT`

UI 中的文本变更，会在一个 `TIMEOUT` 期间之后发送事件到应用程序中。如果在这个 `TIMEOUT` 期间之内又发生了更多的变更，发送给服务器端的事件会包含最新的变更。`TIMEOUT` 期间的长度可以使用 `setInputEventTimeout()` 方法设置。

如果 **ValueChangeEvent** 事件在 `TIMEOUT` 期间之前发生，并且在前一次 **TextChangeEvent** 事件之后发生了文本变更，那么在 **ValueChangeEvent** 事件之前会触发 **TextChangeEvent** 事件。

`TextChangeEventMode.EAGER`

每当文本内容发生变更(通常由键盘击键导致)时，事件都会立刻触发。多个事件会分隔开，并按顺序逐个处理。但事件是异步传输到服务器的，因此在事件的处理过程中还可以继续输入。

5.9.5. CSS 样式规则

```
.v-textfield { }
```

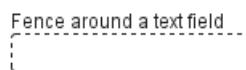
TextField 的 HTML 结构非常简单，只包含一个带有 `v-textfield` 样式的元素。

比如，下面的自定义风格为文本输入框使用虚线边框：

```
.v-textfield-dashing {
    border: thin dashed;
    background: white; /* Has shading image by default */
}
```

运行结果参见图 5.24 “使用 CSS 控制 **TextField** 样式”。

图 5.24. 使用 **CSS** 控制 **TextField** 样式



TextField 使用的样式名，也被其他几种含有文本输入框的组件使用，即便这些文本输入框实际上不是 **TextField**。这一点保证不同的文本输入框使用类似的样式。

5.10. TextArea

TextArea 是 **TextField** 组件的多行版本，**TextField** 参见第 5.9 节“**TextField**”。

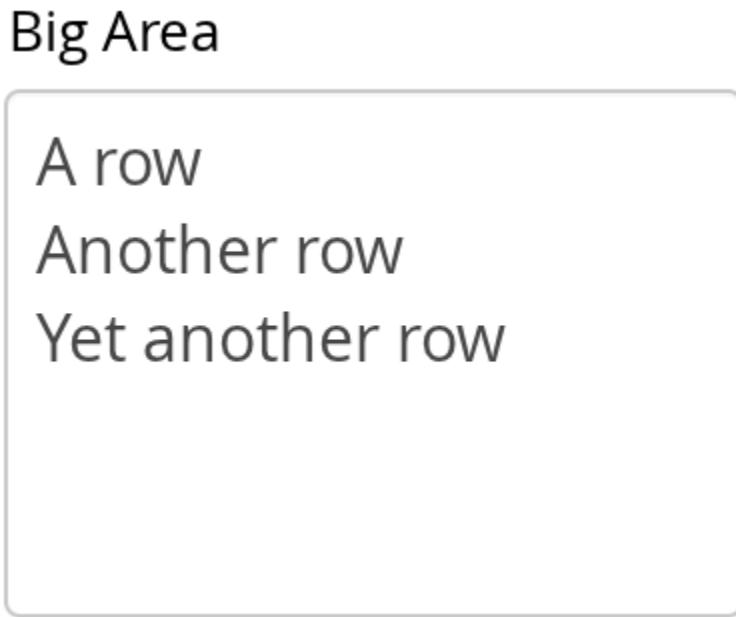
下例创建一个简单的 TextArea:

```
// Create the area
TextArea area = new TextArea("Big Area");

// Put some content in it
area.setValue("A row\n"+
    "Another row\n"+
    "Yet another row");
```

运行结果参见图 5.25 “**TextArea** 示例”。

图 5.25. **TextArea** 示例



你可以使用 `setRows()` 方法设置可见的行数，也可以使用通常的 `setHeight()` 方法以其他单位来控制高度。如果内容的行数超过了可见行数，会出现垂直滚动条。使用 `setRows()` 方法设置高度会为水平滚动条留下空间，因此当水平滚动条不出现时，实际的可见行数可能会比设定值高一行。

你可以使用通常的 `setWidth()` 方法来设置宽度。推荐使用 `em` 单位来设置尺寸，这个单位与当前使用的字体相关。

单词折行

`setWordwrap()` 方法控制当行的长度达到表示区域最大宽度时，长的行是否折行表示(默认设定为 `true`)。如果禁止折行(`false`)，则会出现垂直滚动条(译注：原文如此，似乎应该是水平滚动条才对)。折行显示只是一种视觉上的效果，长行的折行不会在 Field 值中插入换行符；如果将被折行显示的内容变短，就可以看到折行效果消失了。

```
TextArea areal = new TextArea("Wrapping");
areal.setWordwrap(true); // The default
areal.setValue("A quick brown fox jumps over the lazy dog");

TextArea area2 = new TextArea("Nonwrapping");
area2.setWordwrap(false);
```

```
area2.setValue("Victor jagt zw&ouml;lfe Boxk&auml;mpfer quer "+  
"&uuml;ber den Sylter Deich");
```

运行结果参见 图 5.26 “**TextArea** 中的单词折行”.

图 5.26. **TextArea** 中的单词折行

Wrapping

A quick brown fox
jumps over the lazy
dog

Nonwrapping

Victor jagt zwölf B

CSS 样式规则

```
.v-textarea { }
```

TextArea 的 HTML 结构非常简单, 只包含一个带有 v-textarea 样式的元素.

5.11. PasswordField

PasswordField 是 **TextField** 的变体, 它会将输入的内容隐藏起来.

```
PasswordField tf = new PasswordField("Keep it secret");
```

运行结果参见 图 5.27 “**PasswordField**”.

图 5.27. **PasswordField**

Keep it secret

.....|

要注意, **PasswordField** 对输入内容的隐藏只对 "躲在人背后" 的肉眼窥探者有效。除非与服务器之间使用加密连接, 比如 HTTPS, 否则输入内容仍然会以明文传输, 因此可以被那些能够监视网络的人窥探到。此外, 利用浏览器端的 JavaScript 执行安全漏洞, 攻击者也可以发起钓鱼攻击, 拦截浏览器端的输入内容。

CSS 样式规则

```
.v-textfield { }
```

PasswordField 没有独自的 CSS 样式名, 与通常的 **TextField** 使用相同的 v-textfield 样式。关于它的样式控制, 请参见 第 5.9.5 节 “CSS 样式规则”。

5.12. RichTextArea

RichTextArea 组件可用于输入和编辑格式化的文本。它的工具条提供了所有基本的编辑功能。**RichTextArea** 的文本内容以 HTML 格式表达。**RichTextArea** 继承自 **TextField**, 没有增加任何 API 功能。你可以扩展客户端组件 **VRichTextArea** 和 **VRichTextToolbar** 来添加功能。

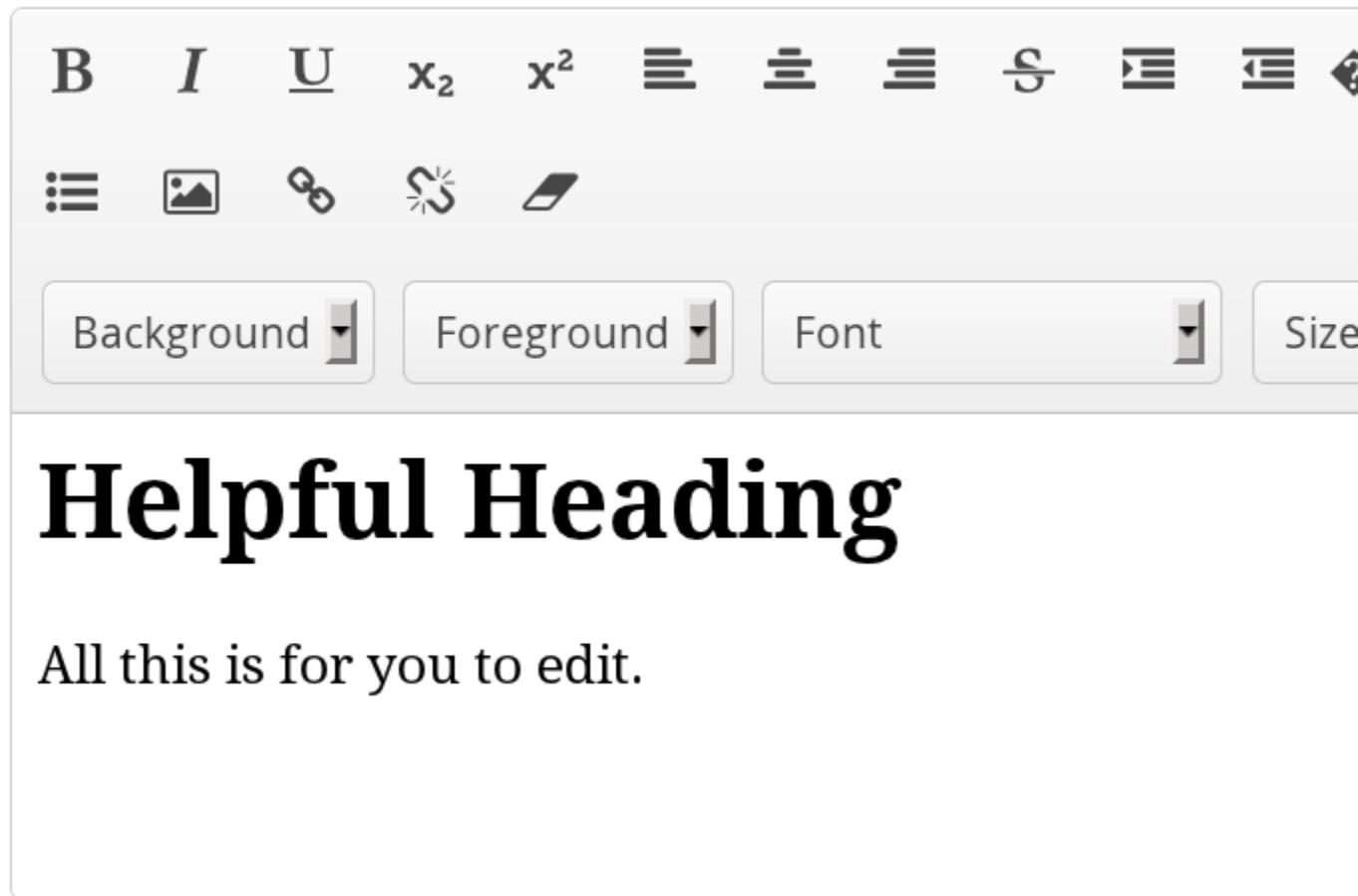
与 **TextField** 一样, **RichTextArea** 的文本内容是 Field 的 **Property**, 可以使用 `setValue()` 方法来设置, 可以使用 `getValue()` 方法来获取。

```
// Create a rich text area
final RichTextArea rtarea = new RichTextArea();
rtarea.setCaption("My Rich Text Area");

// Set initial content as HTML
rtarea.setValue("<h1>Hello</h1>\n" +
    "<p>This rich text area contains some text.</p>");
```

图 5.28. Rich Text Area 组件

My Rich Textarea



上例中，我们在初始的 HTML 内容中使用了内容相关的 tag，比如 `<h1>`。RichTextArea 组件不支持创建这类 tag，只能创建格式化 tag，但它会保留内容中既有的这类 tag，除非用户将这类 tag 编辑删除掉。任何不可见的空白字符，比如换行符(`\n`)都会从内容中删除。比如，上例中设置的值，使用 `getValue()` 方法从 Field 中读取时，结果会是：

```
<h1>Hello</h1> <p>This rich text area contains some text.</p>
```

 跨站脚本警告

用户使用 **RichTextArea** 输入的内容会以 HTML 的形式从浏览器传输到服务器端，其内容不会被过滤。由于 **RichTextArea** 组件的目的是用于输入格式化的文本，你过滤用户输入内容时，不能简单地删除所有的 HTML tag。此外还有很多属性，比如 `style`，也

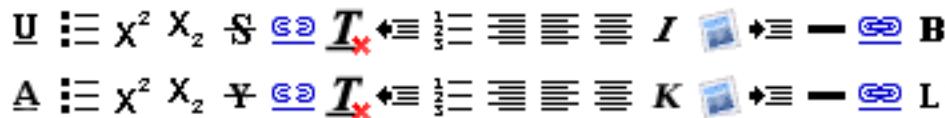
关于跨站脚本问题，以及对用户输入的过滤问题，详情请参见 第 11.8.1 节“处理用户输入内容，防止跨站脚本攻击”。

对 **RichTextArea** 的工具条进行本地化翻译

RichTextArea 是 Vaadin 中少数带有文本标签的组件之一. 工具条中的选择框语言是英文, 目前没有任何其他方法本地化, 只能继承或重新实现客户端的 **VRichTextToolbar** Widget. 按钮可以使用 CSS 简单地本地化, 只需要下载工具条背景图, 编辑它的内容, 然后替换掉默认的工具条. 工具条是单个图片文件, 各个工具条按钮从中取得自己的图片, 因此图标顺序与画面中的显示顺序会不同. 具体的图片文件取决于工具条的客户端实现.

```
.v-richtextarea-richtextexample .gwt-ToggleButton
.gwt-Image {
    background-image: url(img/richtextarea-toolbar-fi.png)
    !important;
}
```

图 5.29. Rich Text Area 工具条通常的英文版, 以及本地化版



CSS 样式规则

```
.v-richtextarea { }
.v-richtextarea .gwt-RichTextToolbar { }
.v-richtextarea .gwt-RichTextArea { }
```

RichTextArea 包括两个主要部分: 工具条, 它的最上层样式是 `.gwt-RichTextToolbar`, 以及编辑器, 样式是 `.gwt-RichTextArea`. 编辑器部分显然包含 HTML 内容中的所有元素及其样式. 工具条则包含工具按钮和下拉列表, 它们的样式名如下:

```
.gwt-ToggleButton { }
.gwt-ListBox { }
```

5.13. 使用 **DateField** 输入日期和时间

DateField 组件用于显示和输入日期/时间. 这个组件有两种不同的形式: **PopupDateField**, 它包括一个数字输入框, 和一个弹出式日历, **InlineDateField**, 直接显示日历. **DateField** 基类默认为弹出式输入方式.

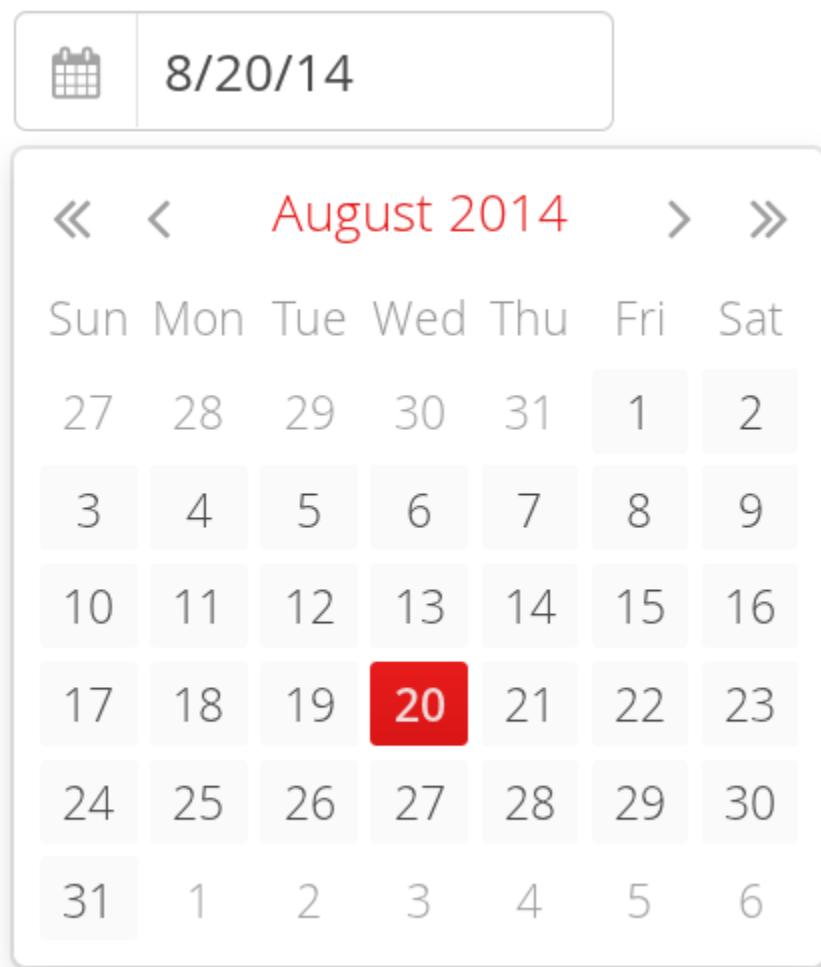
下面的例子演示如何使用 **DateField** 基类, 它与 **PopupDateField** 是等价的. 我们将 **DateField** 的初始时刻设置为当前时刻, 当前时刻使用 `java.util.Date` 类的默认构造函数得到.

```
// Create a DateField with the default style
DateField date = new DateField();

// Set the date and time to present
date.setValue(new Date());
```

运行结果参见 图 5.30 “使用 **DateField (PopupDateField)** 选择日期和时刻”.

图 5.30. 使用 **DateField (PopupDateField)** 选择日期和时刻



5.13.1. PopupDateField

PopupDateField 通过一个文本框来输入日期和时间. 由于 **DateField** 默认就是这个组件, 因此这个组件的使用方法与前面的例子完全相同. 点击日期右侧的按钮会打开一个弹出窗口, 可用来选择年, 月, 日, 以及时间. 此外, **Down** 键也会打开这个弹出窗口. 弹出窗口打开后, 用户可以使用方向键在日历中滚动.

弹出窗口中选择的日期和时间会显示在文本框中, 使用目前语言环境的日期时刻格式, 或者使用 `setDateFormat()` 方法指定的日期时刻格式. 用户输入的日期时刻字符串也使用同样的格式来解析.

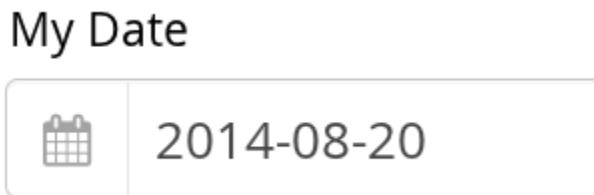
日期和时间的格式

日期和时刻通常使用当前语言环境的格式来显示(参见 第 5.3.5 节 “语言环境(Locale)”). 你也可以使用 `setDateFormat()` 方法指定自定义格式. 这个方法接受的格式字符串遵循 Java 的 **SimpleDateFormat** 中定义的规则.

```
// Display only year, month, and day in ISO format
date.setDateFormat("yyyy-MM-dd");
```

运行结果参见 图 5.31 “**PopupDateField** 的自定义日期格式”.

图 5.31. **PopupDateField** 的自定义日期格式



用户输入日期和时刻的解析，也使用相同的格式，详情见后文。

处理不规则的用户输入

用户经常会输入格式不正确的，或者无效的日期和时刻。**DateField** 有两个层次的校验：第 1 层在客户端，第 2 层在服务器端。

输入的日期首先在客户端校验，当输入框失去焦点时立刻进行校验。如果日期格式无效，会给组件设置上 `v-datepicker-parseerror` 样式。这个样式是否会显示一个可见的错误指示器，由 theme 决定。内建的 reindeer theme 默认不会显示错误指示器，使得服务器端处理这个问题变得更方便一些。

```
.mydate.v-datepicker-parseerror .v-textfield {
    background: pink;
}
```

`setLenient(true)` 可以将日期的解释设置为宽松模式，所以某些不正确的日期，比如 2 月 30 日或 3 月 0 日，会被解释为下一月或前一月的日期。

当日期值发生到服务器端时，或发生服务器端校验。如果日期 Field 被设置为立即模式，校验会在 Field 失去焦点后立刻发生。校验完成后，如果结果不正确，组件的旁边会出现错误指示器。将鼠标指针移动到指示器上，将显示错误信息。

你可以重载 `handleUnparsableDateString()` 方法来处理日期的校验错误。这个方法接受的参数是用户输入的日期字符串，可以在这个方法内实现你自定义的日期解析逻辑，如下例所示。

```
// Create a date field with a custom parsing and a
// custom error message for invalid format
PopupDateField date = new PopupDateField("My Date") {
    @Override
    protected Date handleUnparsableDateString(String dateString)
        throws Property.ConversionException {
        // Try custom parsing
        String[] fields = dateString.split("/");
        if (fields.length >= 3) {
            try {
                int year = Integer.parseInt(fields[0]);
                int month = Integer.parseInt(fields[1]) - 1;
                int day = Integer.parseInt(fields[2]);
                GregorianCalendar c =
                    new GregorianCalendar(year, month, day);
                return c.getTime();
            } catch (NumberFormatException e) {
                throw new Property.
                    ConversionException("Not a number");
            }
        }
    }
}
```

```

        }
    }

    // Bad date
    throw new Property.
        ConversionException("Your date needs two slashes");
}
};

// Display only year, month, and day in slash-delimited format
date.setDateFormat("yyyy/MM/dd");

// Don't be too tight about the validity of dates
// on the client-side
date.setLenient(true);

```

上面的错误处理方法要么返回一个 **Date** 作为解析结果, 要么抛出一个 **ConversionException** 异常代表解析失败. 返回 `null` 会将 Field 值设置为 `null`, 并清空输入框中的文字.

自定义错误信息

除了自定义日期解析之外, 重载上面说的错误处理方法还可以用于国际化, 以及错误信息的定制化. 你也可以使用它作为另一种报告错误的手段, 如下例所示:

```

// Create a date field with a custom error message for invalid format
PopupDateField date = new PopupDateField("My Date") {
    @Override
    protected Date handleUnparsableDateString(String dateString)
        throws Property.ConversionException {
        // Have a notification for the error
        Notification.show(
            "Your date needs two slashes",
            Notification.TYPE_WARNING_MESSAGE);

        // A failure must always also throw an exception
        throw new Property.ConversionException("Bad date");
    }
};

```

如果输入内容无效, 你应该抛出异常; 返回 `null` 值会清空输入框文本, 这通常不是我们期待的结果.

输入提示

与拥有输入框的其他 Field 一样, **PopupDateField** 可以带有一个输入提示, 输入提示会一直显示, 直到用户输入了值. 你可以使用 `setInputPrompt` 方法来设置提示信息.

```

PopupDateField date = new PopupDateField();

// Set the prompt
date.setInputPrompt("Select a date");

// Set width explicitly to accommodate the prompt
date.setWidth("10em");

```

`DateField` 不会自动调整自己的大小来适应提示信息, 所以你需要使用 `setWidth()` 方法来明确地设定宽度.

输入提示信息在 **DateField** 基类中不可用.

CSS 样式规则

```
.v-datepicker, v-datepicker-popupcalendar {}
.v-textfield, v-datepicker-textfield {}
.v-datepicker-button {}
```

DateField (以及它的所有子类)的顶级 HTML 元素带有 v-datepicker 样式. **DateField** 基类和 **PopupDateField** 类还带有 v-datepicker-popupcalendar 样式.

此外, 顶级元素还带有一个代表时间粒度的样式, 样式名以 v-datepicker- 开头, 后面附带一个后缀, 后缀可以是 full, day, month, 或 year 之一. 当时间粒度小于 1 日时使用 -full 样式. 这些样式主要用于控制弹出式日历的画面表现.

文本输入框带有 v-textfield 和 v-datepicker-textfield 样式, 日历按钮带有 v-datepicker-button 样式.

弹出式日历打开后带有以下样式:

```
.v-datepicker-popup {}
.v-popupcontent {}
.v-datepicker-calendarpanel {}
```

弹出式日历的顶级元素带有 .v-datepicker-popup 样式. 注意, 弹出框处于组件 HTML 结构之外, 因此它没有包括在 v-datepicker 元素之内, 也不带有任何自定义的样式. v-datepicker-calendarpanel 之内的内容, 与 **InlineDateField** 相同, 详情请参见 第 5.13.2 节 “**InlineDateField**”.

5.13.2. **InlineDateField**

InlineDateField 提供一个日期选择组件, 表现为月历形式. 用户可以点击适当的箭头按钮在不同的年份和月份间跳转. 与弹出式日历不同, 这个日历会永远显示在界面中.

```
// Create a DateField with the default style
InlineDateField date = new InlineDateField();

// Set the date and time to present
date.setValue(new java.util.Date());
```

运行结果参见 图 5.32 “**InlineDateField** 示例”.

图 5.32. **InlineDateField** 示例

用户也可以使用方向键在日历中跳转.

CSS 样式规则

```
.v-datepicker {
  .v-datepicker-calendarpanel {}
  .v-datepicker-calendarpanel-header {}
  .v-datepicker-calendarpanel-prevyear {}
  .v-datepicker-calendarpanel-prevmonth {}
  .v-datepicker-calendarpanel-month {}
  .v-datepicker-calendarpanel-nextmonth {}
  .v-datepicker-calendarpanel-nextyear {}
  .v-datepicker-calendarpanel-body {}
  .v-datepicker-calendarpanel-weekdays,
  .v-datepicker-calendarpanel-weeknumbers {}
  .v-first {}
  .v-last {}
  .v-datepicker-calendarpanel-weeknumber {}
  .v-datepicker-calendarpanel-day {}
  .v-datepicker-calendarpanel-time {}
  .v-datepicker-time {}
  .v-select {}
  .v-label {}
}
```

顶级元素带有 `v-datepicker` 样式. 除此之外, 顶级元素还带有一个代表时间粒度的样式, 样式名以 `v-datepicker-` 开头, 后面附带一个后缀, 后缀可以是 `full`, `day`, `month`, 或 `year` 之一. 当时间粒度小于 1 日时使用 `-full` 样式.

当星期数的显示被打开时，会出现 `v-datepicker-calendarpanel-weeknumbers` 和 `v-datepicker-calendarpanel-weeknumber` 样式。其中前一个样式用来控制星期数的表头部分的表现，后一个用来控制实际的星期数。

其他样式名称的含义非常明显。对于 `weekdays` 来说，`v-first` 和 `v-last` 样式 allow making rounded endings for the weekday bar. (译注：这段不理解，待校)

5.13.3. 日期与时间的粒度

除了显示日期单位的日历之外，**DateField** 也可以显示到时、分，或者只显示月或年。各项目的可见度由 **时间粒度** 来控制，时间粒度可以使用 `setResolution()` 方法来设置。这个方法接受的参数是界面中可见的最小项目，对于日期来说，一般使用 `DateField.Resolution.DAY`，对于日期加时分来说，使用 `DateField.Resolution.MIN`。时间粒度参数的完整列表请参见 API 文档。

5.13.4. DateField 的本地化

日期和时间的显示格式遵循用户的语言环境，语言环境由浏览器报告给服务器。你可以使用 **AbstractComponent** 的 `setLocale()` 方法，设定自定义的语言环境，详情请参见 第 5.3.5 节“语言环境(Locale)”。注意，只支持 Gregorian 日历。

5.14. Button

Button 组件通常用于激发某种动作，比如完成 Form 的输入。当用户点击按钮时，会激发 **Button.ClickEvent** 事件，这个事件可以通过 `Button.ClickListener` 监听器的 `buttonClick()` 方法来处理。

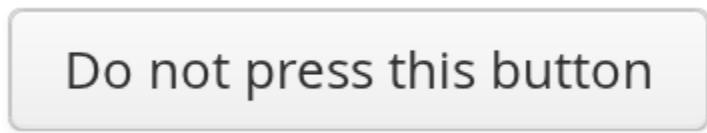
你可以使用匿名类处理按钮的点击事件，如下例所示：

```
Button button = new Button("Do not press this button");

button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        Notification.show("Do not press this button again");
    }
});
```

运行结果参见 图 5.33 “Button 示例”。监听器也可以在构造函数中指定，这样通常可以使代码更简单一些。

图 5.33. Button 示例



Do not press this button

如果你在同一个监听器中处理多个按钮的事件，你可以通过 **Button.ClickEvent** 的 `getButton()` 方法得到与事件相关的 **Button** 对象，然后确定激发事件的是哪个按钮。详情请参见 第 3.4 节“事件和监听器”，其中有详细的介绍和例子。

CSS 样式规则

```
.v-button { }
.v-button-wrap { }
.v-button-caption { }
```

按钮的最外层样式是 `v-button`, 标签带有 `v-button-caption` 样式, 这二者之间还存在一个封装元素, 某些情况下可以帮助控制样式.

某些内建的 theme 包含 `small` 样式, 你可以添加 `Reindeer.BUTTON_SMALL` 样式来使用它. `BaseTheme` 也带有 `BUTTON_LINK` 样式, 可以让按钮的外观类似于超链接.

5.15. CheckBox

CheckBox 是一种两状态的选择组件, 可以处于选中或非选中状态. **CheckBox** 的标题会放在真实的选择框的右侧. Vaadin 提供了两种方式创建 **CheckBox**: 1, 使用 **CheckBox** 组件, 创建独立的 **CheckBox**, 详情请见本节的介绍, 2, 使用 **OptionGroup** 组件的多选模式, 创建 **CheckBox** 组, 详情请参见第 5.19 节 “**OptionGroup**”.

点击 **CheckBox** 将改变它的状态. 状态是 `Boolean` 类型属性, 可以使用 `Property` 接口的 `setValue()` 方法来设置, 使用 `getValue()` 方法得到. 改变 **CheckBox** 的值将触发一个 `ValueChangeEvent` 事件, 这个事件可以使用 `ValueChangeListener` 监听器来处理.

```
CheckBox checkbox1 = new CheckBox("Box with no Check");
CheckBox checkbox2 = new CheckBox("Box with a Check");

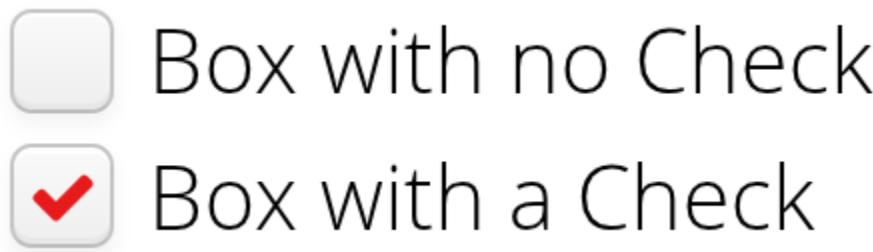
checkbox2.setValue(true);

checkbox1.addValueChangeListener(event -> // Java 8
    checkbox2.setValue(! checkbox1.getValue()));

checkbox2.addValueChangeListener(event -> // Java 8
    checkbox1.setValue(! checkbox2.getValue()));
```

运行结果参见 图 5.34 “Check Box 示例”.

图 5.34. Check Box 示例



关于在 Table 内使用 **CheckBox** 的例子, 请参见第 5.21 节 “**Table**”.

CSS 样式规则

```
.v-checkbox { }
.v-checkbox > input { }
.v-checkbox > label { }
```

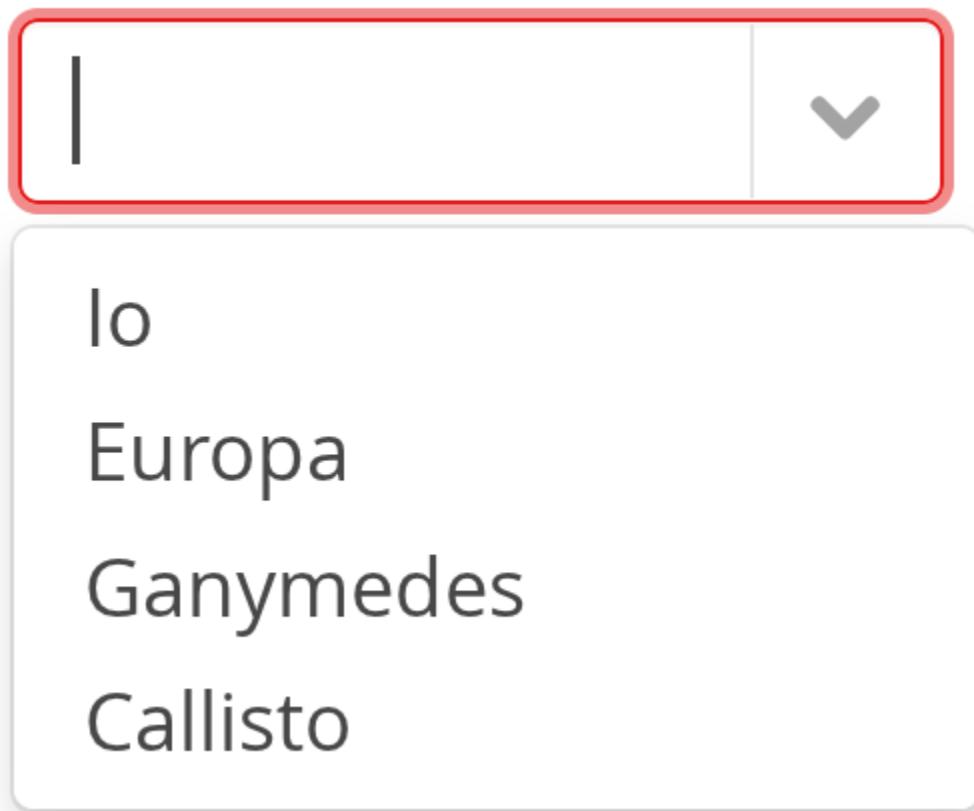
CheckBox 的顶级元素带有 v-checkbox 样式，其中包含两个子元素：1. 真实的选择框 input 元素，2. label 元素。如果你希望标题显示在左侧，你可以对顶级元素使用 "direction: rtl" 来修改标题的位置。

5.16. ComboBox

ComboBox 是一个选择组件，它允许通过下拉列表来选择项目。这个组件也带有一个文本输入框，允许输入查询文字，通过这个查询文字来过滤下拉列表中出现的项目。关于选择组件的共通功能，请参见 第 5.5 节“选择组件”。

图 5.35. ComboBox 组件

My Select

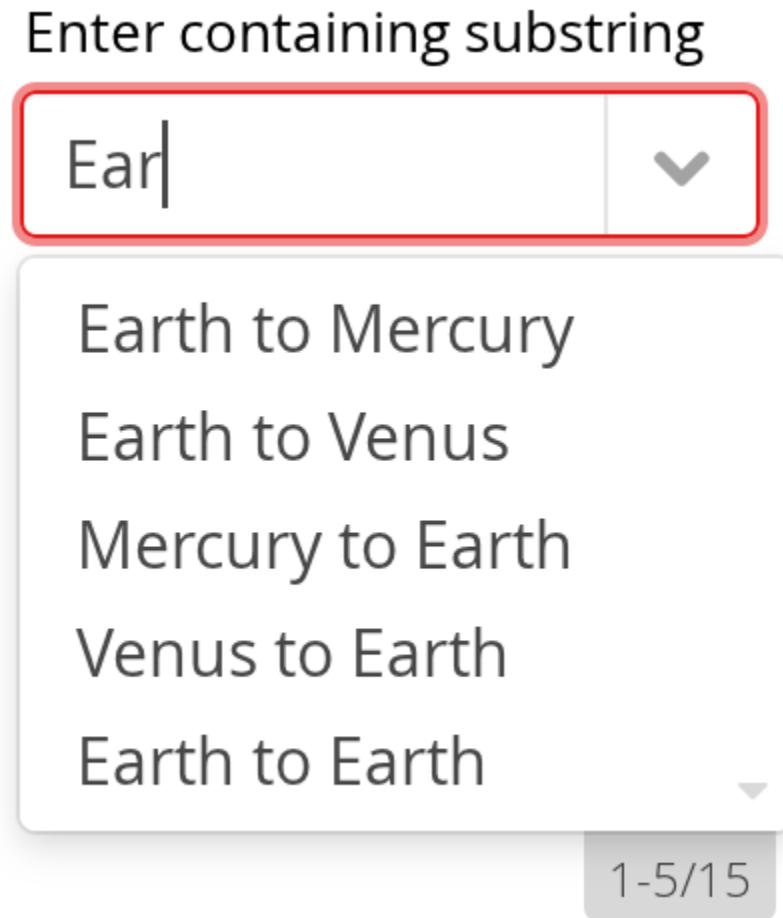


当用户按下 **Enter** 时，**ComboBox** 可以添加新的选择项目，详情请参见 第 5.5.6 节“允许添加新项目”。

5.16.1. 被过滤的选择项

ComboBox 允许在文本输入框中输入字符串来过滤下拉列表中的可选项目。

按下 **Enter** 键可以完成文本的输入。按下 **Up**- 和 **Down**- 键可以从下拉列表中选择项目。下拉列表内容是分页显示的，点击翻滚按钮可以滚动到下一页或前一页。列表选择也可以通过键盘上的方向键来实现(译注：这句不理解，待校)。界面中显示的项目只在需要的时候才从服务器端载入，因此这个组件内包含的选项数目可以非常大。与过滤条件相匹配的项目数会显示在下拉列表中。

图 5.36. 在 **ComboBox** 中过滤可选项目

通过 `setFilteringMode()` 方法设置一个 过滤模式, 可以激活过滤功能 .

```
cb.setFilteringMode(FilteringMode.CONTAINS);
```

过滤模式定义在 **FilteringMode** 枚举型中, 如下:

CONTAINS

在组件的文本框中输入的文本为查询条件, 凡包含这串文本的项目都将匹配成功.

STARTSWITH

只匹配以查询文字开头的项目

OFF(默认)

默认情况下, 过滤功能是关闭的, 此时所有选项都会显示.

上例使用"包含"模式过滤, 匹配包含查询字符串的所有项目. 运行结果见下面的 图 5.36 “在 **ComboBox** 中过滤可选项目”, 我们在文本输入框中输入字符串时, 下拉列表将显示所有匹配成功的项目.

CSS 样式规则

```
.v-filterselect { }
.v-filterselect-input { }
.v-filterselect-button { }

// Under v-overlay-container
.v-filterselect-suggestpopup { }
.popupContent { }
.v-filterselect-prevpage,
.v-filterselect-prevpage-off { }
.v-filterselect-suggestmenu { }
.gwt-MenuItem { }
.v-filterselect-nextpage,
.v-filterselect-nextpage-off { }
.v-filterselect-status { }
```

在默认状态下, **ComboBox** 组件只有文本输入框是可见的. 整个组件包含在 v-filterselect 样式的元素之内(这个样式名是旧版本遗留下来的), 文本输入框带有 v-filterselect-input 样式, 右侧负责打开和关闭下拉列表的按钮带有 v-filterselect-button 样式.

选择项目的下拉列表最外层样式是 v-filterselect-suggestpopup. 它包含一系列的选择项目, 样式为 v-filterselect-suggestmenu. 当存在更多的项目, 无法一次显示完毕时, 会显示翻页导航按钮, 样式为 v-filterselect-prevpage 和 v-filterselect-nextpage. 翻页导航按钮不显示时, 对应的页面元素样式名将带上 -off 后缀. 最底部的状态条会显示翻页状态, 样式为 v-filterselect-status.

5.17. ListSelect

ListSelect 组件是一个列表框, 其中显示可选择的项目列表, 垂直排列. 如果选择项目的数量超过了组件高度, 会显示滚动条. 这个组件支持单选模式也支持多选模式, 可以通过 setMultiSelect() 方法设置. 这两种模式的外观表现是完全一样的.

```
// Create the selection component
ListSelect select = new ListSelect("The List");

// Add some items (here by the item ID as the caption)
select.addItems("Mercury", "Venus", "Earth", ...);

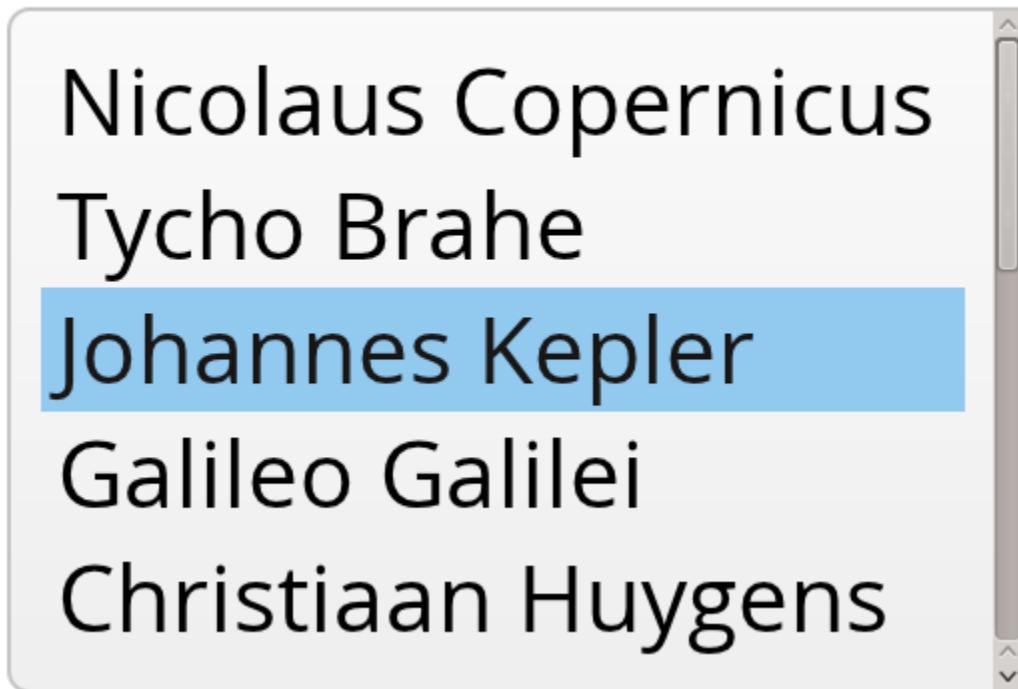
select.setNullSelectionAllowed(false);

// Show 5 items and a scrollbar if there are more
select.setRows(5);
```

组件中可见项目的数量可以使用 setRows() 方法设置.

图 5.37. ListSelect 组件

The List



关于选择组件的共通功能, 请参见 第 5.5 节“选择组件”.

CSS 样式规则

```
.v-select {}
.v-select-select {}
option {}
```

这个组件的最外层样式为 v-select. 浏览器原生的 <select> 元素样式为 v-select-select. 选择项目使用 <option> 元素来表示

5.18. NativeSelect

NativeSelect 是一个下拉选择组件, 使用 Web 浏览器原生的(native) select 输入框实现, 也就是 HTML <select> 元素.

```
// Create the selection component
NativeSelect select = new NativeSelect("Native Selection");

// Add some items
select.addItems("Mercury", "Venus", ...);
```

setColumns() 方法可以设置列表的宽度, 单位为 "列(column)", 具体尺寸由浏览器决定.

图 5.38. NativeSelect 组件



关于选择组件的共通功能, 请参见 第 5.5 节“选择组件”。

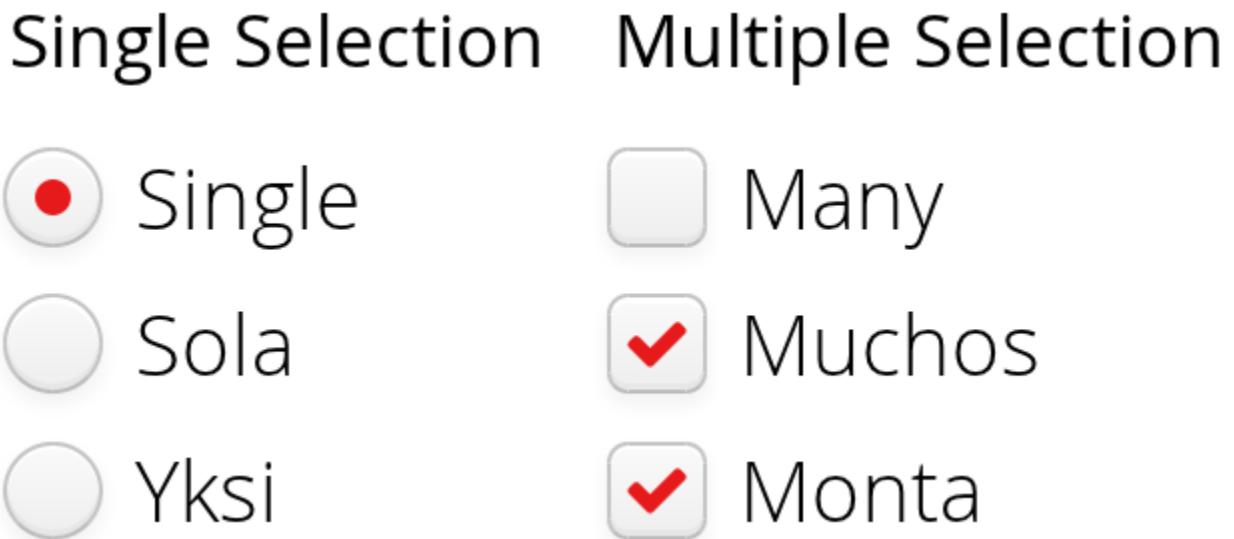
CSS 样式规则

```
.v-select {}  
.v-select-select {}
```

这个组件最外层样式为 v-select. 原生的 select 元素的样式为 v-select-select.

5.19. OptionGroup

OptionGroup 是一个选择组件, 对于单选模式它使用一组 Radio Button. 对于多选模式, 它使用一组 Check Box. 关于选择组件的共通功能, 请参见 第 5.5 节“选择组件”.

图 5.39. **OptionGroup** 的单选模式和多选模式

Option group 默认为单选模式. 使用 `setMultiSelect()` 方法可以激活多选模式.

```
// A single-select radio button group
OptionGroup single = new OptionGroup("Single Selection");
single.addItems("Single", "Sola", "Yksi");

// A multi-select check box group
OptionGroup multi = new OptionGroup("Multiple Selection");
multi.setMultiSelect(true);
multi.addItems("Many", "Muchos", "Monta");
```

图 5.39 “OptionGroup 的单选模式和多选模式”展示了 **OptionGroup** 的单选和多选模式.

当然你也可以使用 **CheckBox** 类创建独立的 Check Box, 参见 第 5.15 节 “**CheckBox**”. **OptionGroup** 组件的优势在于它会维护 Check Box, 你可以很容易地得到当前选中项目的数组, 而且你也可以容易地修改整个组件的外观表现.

5.19.1. 禁用项目

你可以使用 `setItemEnabled()` 方法禁用 **OptionGroup** 内的单个项目. 在多选模式下, 对于被禁用的项目, 用户不能选中它, 也不能解除选中, 但在单选模式下, 用户可以将当前选项从一个被禁用的项目变为另一个有效的项目. 选中项可以通过程序来变更, 无论项目是有效还是禁用. 你可以使用 `isItemEnabled()` 方法检查某个项目是有效还是禁用.

`setItemEnabled()` 使用项目的 ID 来指定它将要禁用或允许的项目.

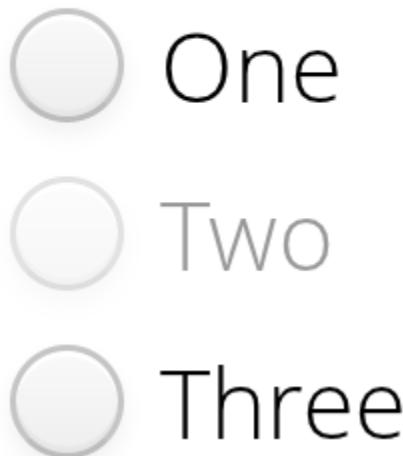
```
// Have an option group with some items
OptionGroup group = new OptionGroup("My Disabled Group");
group.addItems("One", "Two", "Three");

// Disable one item by its item ID
group.setItemEnabled("Two", false);
```

上面的例子中, 项目 ID 也用作项目标题. 运行结果见图 5.40 “带有禁用项目的 **OptionGroup**”.

图 5.40. 带有禁用项目的 OptionGroup

My Disabled Group



将项目设置为禁用, 将使它带有 v-disabled 样式.

CSS 样式规则

```
.v-select-optiongroup {}
.v-select-option.v-checkbox {}
.v-select-option.v-radiobutton {}
```

v-select-optiongroup 是这个组件的最外层样式. 各个 Check Box 将带有 v-checkbox 样式, 借用自 **CheckBox** 组件, 各个 Radio Button 带有 v-radiobutton 样式. Radio button 和 Check Box 都带有 v-select-option 样式, 无论组件是单选还是多选模式, 都可以使用这个样式进行样式控制. 禁用的项目将额外带有 v-disabled 样式.

选择项目通常是垂直排列的. 你可以让它们变为水平排列, 方法是为其设置 display: inline-block 样式. 如果布局组件在水平方向上空间不足, 或者水平宽度未指定时, 选项通常会换行表示, 对最外层元素设置 nowrap 样式可以禁止换行.

```
/* Lay the options horizontally */
.v-select-optiongroup-horizontal .v-select-option {
    display: inline-block;
}

/* Avoid wrapping if the layout is too tight */
.v-select-optiongroup-horizontal {
    white-space: nowrap;
}

/* Some extra spacing is needed */
.v-select-optiongroup-horizontal
    .v-select-option.v-radiobutton {
        padding-right: 10px;
}
```

要使用上面的样式规则, 需要对组件设置一个自定义的 horizontal 样式名称. 运行结果见图 5.41 “水平排列的 **OptionGroup**”.

图 5.41. 水平排列的 **OptionGroup**

Horizontal Group



5.20. TwinColSelect

TwinColSelect 是一个多选组件, 它并列显示两个列表框, 左侧是未选择的项目, 右侧是已选择的项目. 用户可以从左侧列表选择项目, 然后点击 ">>" 按钮将它移动到右侧列表中. 取消选择一个项目, 可以在右侧列表选中它, 然后点击 "<<" 按钮.

TwinColSelect 永远是多选模式, 因此它的属性值永远是被选中项目 ID 的集合, 也就是它右侧列表中项目的 ID 集合.

除整个组件的标题(由所属的布局来管理)之外, 左侧和右侧选择列也可以有自己的标题. 你可以使用 `setLeftColumnCaption()` 和 `setRightColumnCaption()` 方法来设置选择列的标题.

```

TwinColSelect select = new TwinColSelect("Select Targets");

// Put some items in the select
select.addItems("Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune");

// Few items, so we can set rows to match item count
select.setRows(select.size());

// Preselect a few items by creating a set
select.setValue(new HashSet<String>(
    Arrays.asList("Venus", "Earth", "Mars")));

// Handle value changes
select.addValueChangeListener(event -> // Java 8
    layout.addComponent(new Label("Selected: " +
        event.getProperty().getValue())));
  
```

运行结果参见 图 5.42 “双列选择组件”.

`setRows()` 方法设置组件的高度, 单位是选择框中可见项目的数量. 使用 `setHeight()` 方法设置高度, 会覆盖掉行数设定.

关于选择组件的共通功能, 请参见 第 5.5 节 “选择组件”.

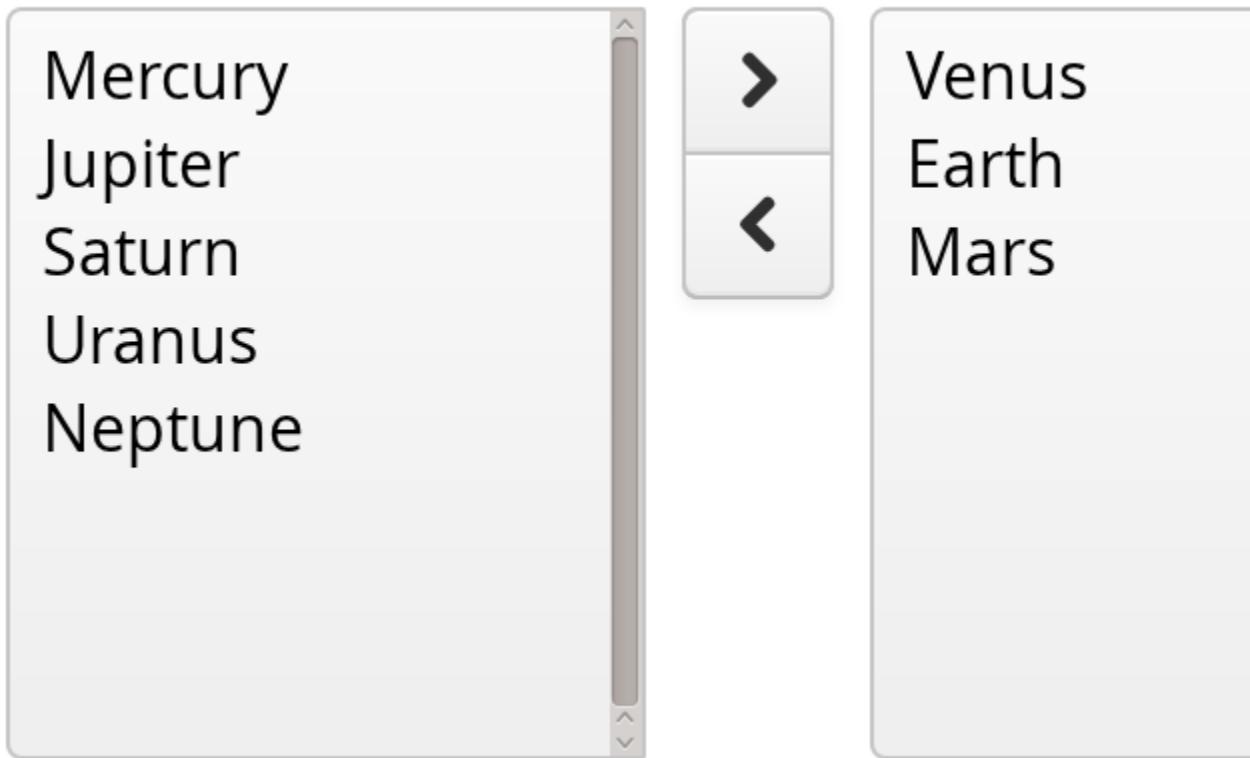
CSS 样式规则

```

.v-select-twincol {}
.v-select-twincol-options-caption {}
  
```

图 5.42. 双列选择组件

Select Targets



```
.v-select-twincol-selections-caption {}
.v-select-twincol-options {}
.v-select-twincol-buttons {}
  .v-button {}
    .v-button-wrap {}
      .v-button-caption {}
  .v-select-twincol-deco {}
.v-select-twincol-selections {}
```

TwinColSelect 组件的最外层样式为 `v-select-twincol`. 如果左侧和右侧列表的标题有设定, 标题的样式将分别是 `v-select-twincol-options-caption` 和 `v-select-twincol-options-caption`. 左侧列表显示未选择的项目, 样式为 `v-select-twincol-options-caption`, 右侧列表显示已选择的项目, 样式为 `v-select-twincol-options-selections`. 这两个列表之间是按钮栏, 其最外层样式为 `v-select-twincol-buttons`; 实际的按钮共用 **Button** 组件的样式, 按钮之间是分隔元素, 样式为 `v-select-twincol-deco`.

5.21. Table

Table 组件的目的是以行和列的形式表现表格式的数据. **Table** 是 Vaadin 中最通用的组件之一. Table 的单元格可以包含文字, 或者任意的其他组件. 你可以很简便地实现 Table 内的数据编辑, 比如, 点击一个单元格可以将它变为一个文本输入框, 然后就可以编辑这条数据.

Table 内包含的数据是使用 Vaadin 数据模型(参见 第 8 章 组件与数据绑定)来管理的, 具体来说是通过 **Table** 类的 **Container** 接口. 因此 Table 可以直接绑定到一个数据源, 比如一个数据库查询. 只有在 Table 内可见的部分会被装载进浏览器中, 使用滚动条移动可见区域会从服务器端装载

新的数据内容，数据装载完毕后，会显示一个提示信息，其中包括 Table 内项目的总件数，以及当前显示的项目范围。Table 的行就是容器内的 项目(item)，Table 的列则是项目的 属性(property)。Table 的各行(项目)使用 项目 ID(item identifier)(IID) 来区分，各列(property)使用 属性 ID(property identifier)(PID) 来区分。

创建 Table 时，你首先需要使用 addContainerProperty() 方法定义列。这个方法有两种使用形式。较简单的形式接受列的属性 ID 作为参数，并且将属性 ID 用作列标题。较复杂的形式可以将列的属性 ID 和标题区分开。这可以使得对 Table 的表头文字进行国际化更容易一些，因为如果 PID 被国际化了，那么这个 PID 出现的一切地方都要进行同样的国际化。这个方法的较复杂的形式还允许为列定义图标资源。当新的属性(列)添加到 Table 中时，会使用“默认值”参数来填充未指定的项目值。(通常情况下这个默认值并没有什么意义，如下例所示，我们在定义属性之后再添加项目。)

```
Table table = new Table("The Brightest Stars");

// Define two columns for the built-in container
table.addContainerProperty("Name", String.class, null);
table.addContainerProperty("Mag", Float.class, null);

// Add a row the hard way
Object newItemId = table.addItem();
Item row1 = table.getItem(newItemId);
row1.getItemProperty("Name").setValue("Sirius");
row1.getItemProperty("Mag").setValue(-1.46f);

// Add a few other rows using shorthand addItem()
table.addItem(new Object[]{"Canopus", -0.72f}, 2);
table.addItem(new Object[]{"Arcturus", -0.04f}, 3);
table.addItem(new Object[]{"Alpha Centauri", -0.01f}, 4);

// Show exactly the currently contained rows (items)
table.setPageLength(table.size());
```

上面的例子中，我们使用一个序列增长的 **Integer** 对象作为项目 ID，项目 ID 作为 addItem() 的第 2 个参数。实际的行以简单的对象数组的形式表达，数组内数据的顺序与我们定义的属性顺序一致。各个数据对象的类型必须正确，各属性的值数据类型通过 addContainerProperty() 方法指定。

图 5.43. **Table** 的基本示例

The Brightest Stars

Name	Mag
Sirius	-1,46
Canopus	-0,72
Arcturus	-0,04
Alpha Centauri	-0,01

Table 的可扩展性(Scalability)很大程度上由容器决定。默认的 **IndexedContainer** 类相对来说比较重，某些时候，比如更新数据值时，可能导致可扩展性问题。我们推荐你使用一种为你的应用程序专门优化过的容器。**Table** 对于项目的数量没有限制，对于几十万条数据，它也和只有几条数据时一样运行迅速。如果使用滚动功能的当前实现，会有大约 50 万行左右的数量限制，具体数字取决于浏览器和各行的高度。

关于选择组件的共通功能，请参见 第 5.5 节“选择组件”。

5.21.1. 在 **Table** 内选择项目

在 **Table** 中可以通过鼠标点击来选择一个或多个项目。当用户选择一个项目时，项目的 IID 会被设置为 **Table** 的属性值，同时触发 **ValueChangeEvent** 事件。你需要将 **Table** 设置为 可选择，才能允许选择 **Table** 内的项目。大多数情况下，你还需要将 **Table** 设置为 立即模式，如下例所示，因为如果不是立即模式，属性的变化不会立即发送到服务器端。

下面的例子演示如何允许在 **Table** 内选择项目，以及如何处理由选择项变化造成的 **ValueChangeEvent** 事件。你需要使用 **Property.ValueChangeListener** 接口的 **valueChange()** 方法来处理这个事件。

```
// Allow selecting items from the table.
table.setSelectable(true);

// Send changes in selection immediately to server.
table.setImmediate(true);
```

```
// Shows feedback from selection.
final Label current = new Label("Selected: -");

// Handle selection change.
table.addValueChangeListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        current.setValue("Selected: " + table.getValue());
    }
});

});
```

图 5.44. Table 内项目选择的示例

First Name	Last Name	Year	
Nicolaus	Copernicus	1473	▲
Tycho	Brahe	1546	⋮
Giordano	Bruno	1548	⋮
Galileo	Galilei	1564	⋮
Johannes	Kepler	1571	▼

Selected: 2

如果用户点击一个已经选中的项目，那么这个项目会取消选中，Table 的属性值将成为 `null`. 你可以通过 `setNullSelectionAllowed(false)` 来禁止这种行为.

选中的项目就是 Table 的属性的值，因此你可以通过 `getValue()` 得到它. 你同样可以通过 Table 本身得到它. 单选模式下，这个值是被选中的项目的 ID，如果没有项目被选中的话，就是 `null`. 在多选模式下(见下文)，属性值是项目 ID 组成的 **Set**. 注意这个集合是不可修改的，因此你不可以通过改变这个集合的内容来改变项目的选中状态.

多选模式

Table 也可以处于 多选模式，这时用户可以按住 **Ctrl** 键(或 **Meta** 键)不放，然后鼠标点击多个项目来选择它们. 如果 **Ctrl** 键不按下，点击一个项目会选中它，但其他被选中的项目会被取消选中. 用户可以选择一个范围内的所有项目，方法是先选中一个项目，按住 **Shift** 键不放，然后点击另一个项目，这两个项目之间的所有项目都会被选中. 还可以选择多个范围，方法是先选中一个范围，然后按住 **Ctrl** 键不放，再同时按住 **Ctrl** 和 **Shift** 键不放，选择另一个项目.

使用 Table 的基类 **AbstractSelect** 中的 `setMultiSelect()` 方法可以打开多选模式. 将 Table 设为多选模式不会隐含地将它设置为 可选择，因此还必须设置 Table 为可选择.

`setMultiSelectMode()` 方法决定多选的控制模式: `MultiSelectMode.DEFAULT` 是默认模式，这种模式下选择项目需要按住 **Ctrl** (或 **Meta**) 键不放，`MultiSelectMode.SIMPLE` 模式下则不需要按住 **Ctrl** 键. 在 simple 模式下，要取消一个已选中的项目，必须再次点击它.

5.21.2. Table 的功能

页长与滚动条

Table 的默认风格是一个带滚动条的表格. 滚动条在表格的右侧，只在 Table 内项目的数量超过页长时才显示，页长也就是可见项目的数量. 你可以使用 `setPageLength()` 方法设置页长.

将页长设置为 0，会显示 Table 内所有项目，无论项目数量有多大. 注意，这样的设置也会导致缓冲功能失效，因为 Table 内所有数据都必须一次性装载到浏览器端. 使用这样的 Table 来生成报表不

适用于大量数据的情况，因为它会在使用 Ajax 描绘 Table 时出现大量的数据传输。对于大量数据的报表，直接生成 HTML 内容会更高效一些。

拖动列的宽度

你可以在服务器端程序中使用 `setColumnWidth()` 方法设置列的宽度。列通过对应的属性 ID 来标识，宽度单位为像素。

用户也可以拖动两列之间的 resize 条来调整 Table 的列宽。重新调整列宽会激发 **ColumnResizeEvent** 事件，这个事件可以通过 **Table.ColumnResizeListener** 监听器来处理。通常你会希望立即收到列宽变化的事件，这时 Table 必须设置为立即模式。

```
table.addColumnResizeListener(new Table.ColumnResizeListener() {
    public void columnResize(ColumnResizeEvent event) {
        // Get the new width of the resized column
        int width = event.getCurrentWidth();

        // Get the property ID of the resized column
        String column = (String) event.getPropertyId();

        // Do something with the information
        table.setColumnFooter(column, String.valueOf(width) + "px");
    }
});

// Must be immediate to send the resize events immediately
table.setImmediate(true);
```

图 5.45 “拖动列的宽度”是 Table 的列宽被调整过之后的结果。

图 5.45. 拖动列的宽度

ColumnResize Events	
NAME	BORN IN
Väisälä	1891
Valtaoja	1951
Galileo	1564
124px	248px

拖动列的顺序

如果设置了 `setColumnReorderingAllowed(true)`，用户可以调整 Table 内列的顺序，方法是用鼠标拖动列头部分。

将列收起

如果设置了 `setColumnCollapsingAllowed(true)`，Table 头部的右侧会显示一个下拉列表框，在这个列表框中可以选择显示哪些列。将列收起，与使用 `setVisibleColumns()` 方法将列隐藏起来是不同的，后一种方法会导致列完全被隐藏，用户不能通过 UI 操作将这个列显示出来。

你可以通过程序使用 `setColumnCollapsed()` 方法将列收起。将列收起之前，必须用前面介绍的方法将 Table 设置为可收起模式，否则将抛出 **IllegalAccessException** 例外。

```
// Allow the user to collapse and uncollapse columns
table.setColumnCollapsingAllowed(true);
```

```
// Collapse this column programmatically
try {
    table.setColumnCollapsed("born", true);
} catch (IllegalAccessException e) {
    // Can't occur - collapsing was allowed above
    System.err.println("Something horrible occurred");
}

// Give enough width for the table to accommodate the
// initially collapsed column later
table.setWidth("250px");
```

运行结果见图 5.46 “将列收起”。

图 5.46. 将列收起

Column Collapsing	
NAME	DIED
Galileo	1642
Väisälä	1971
Valtaoja	

- Name
- Died
- Born**

如果不指定 Table 的宽度, 它会将自己的宽度最小化到与可见的列相适应的程度。如果某些列初始化就是被收起的, 那么这些列再次展开之后, Table 的宽度可能会不足以显示它们, 这时会出现一个难看的水平滚动条。为了避免这种情况, 你需要考虑为 Table 设置足够的宽度, 以便显示被用户展开后的列。

Table 内的组件

Table 内的单元格不仅可以显示字符串, 而且可以包含任意的 UI 组件。如果行的高度超过默认 theme 定义的行高, 那么你就需要使用自定义 theme 来定义适当的行高。

对于 Table 内的组件, 比如下例中的 **Button**, 当你处理它的事件时, 你需要知道这个组件所属的项目。组件本身是不知道 Table 的, 也不知道组件所属的项目。因此, 事件处理代码需要使用其他手段来确定当前项目的 ID。可能的方法有几种。通常最简单的办法是使用 `setData()` 方法将任意对象绑定到组件上。你也可以继承组件, 然后在子类中存放项目 ID 信息。你也可以简单地在整个 Table 内搜索组件所属的项目, 不过这种方案的可扩展性可能不太好。

下联的例子在 Table 的行内嵌入一个 HTML 内容模式的 **Label**, 一个多行的 **TextField**, 一个 **CheckBox**, 以及一个显示为链接风格的 **Button**。

```
// Create a table and add a style to allow setting the row height in theme.
final Table table = new Table();
table.addStyleName("components-inside");

/* Define the names and data types of columns.
 * The "default value" parameter is meaningless here. */
table.addContainerProperty("Sum", Label.class, null);
table.addContainerProperty("Is Transferred", CheckBox.class, null);
table.addContainerProperty("Comments", TextField.class, null);
table.addContainerProperty("Details", Button.class, null);

/* Add a few items in the table. */
for (int i=0; i<100; i++) {
    // Create the fields for the current table row
    Label sumField = new Label(String.format(
        "Sum is <b>$%04.2f</b><br/><i>(VAT incl.)</i>",
```

```
new Object[] {new Double(Math.random()*1000)}),  
    ContentMode.HTML);  
CheckBox transferredField = new CheckBox("is transferred");  
  
// Multiline text field. This required modifying the  
// height of the table row.  
TextField commentsField = new TextField();  
commentsField.setRows(3);  
  
// The Table item identifier for the row.  
Integer itemId = new Integer(i);  
  
// Create a button and handle its click. A Button does not  
// know the item it is contained in, so we have to store the  
// item ID as user-defined data.  
Button detailsField = new Button("show details");  
detailsField.setData(itemId);  
detailsField.addClickListener(new Button.ClickListener() {  
    public void buttonClick(ClickEvent event) {  
        // Get the item identifier from the user-defined data.  
        Integer iid = (Integer)event.getButton().getData();  
        Notification.show("Link " +  
            iid.intValue() + " clicked.");  
    }  
});  
detailsField.addStyleName("link");  
  
// Create the table row.  
table.addItem(new Object[] {sumField, transferredField,  
    commentsField, detailsField},  
    itemId);  
}  
  
// Show just three rows because they are so high.  
table.setPageLength(3);
```

行高必须设置为比默认高度更高一些，方法是使用以下样式规则：

```
/* Table rows contain three-row TextField components. */  
.v-table-components-inside .v-table-cell-content {  
    height: 54px;  
}
```

上面示例的运行结果见图 5.47 “Table 内嵌的 UI 组件”。

图 5.47. Table 内嵌的 UI 组件

Sum	Is Transferred	Comments	Details
Sum is \$777,60 (VAT incl.)	<input checked="" type="checkbox"/> is transferred	We sent this money already in last week.	show details
Sum is \$500,40 (VAT incl.)	<input type="checkbox"/> is transferred		show details
Sum is \$836,10 (VAT incl.)	<input type="checkbox"/> is transferred		show details

遍历一个 Table

由于 **Table** 内的项目是无下标索引的，因此要遍历所有的项目必须使用 iterator. **Table** 类的 **Container** 接口的 `getItemIds()` 方法返回一个 **Collection**, 其中的内容是项目 ID, 你可以使用 **Iterator** 来遍历这个 Collection. 遍历 **Table** 的例子, 请参见 第 8.5 节 “在容器(Container)中保存项目(Item)”. 注意, 在遍历过程中你不能修改 **Table** 内容, 也就是说, 不能增加或删除项目. 但修改项目内的数据是允许的.

过滤 Table 内容

如果 Table 的容器数据源实现了 **Filterable** 接口, Table 内容可以过滤, 默认的 **IndexedContainer** 是实现了这个接口的. 详情请参见 第 8.5.7 节 “**Filterable** 容器”.

5.21.3. 在 Table 内编辑数据值

Table 通常只是简单地以文本的方式显示项目中的各字段. 如果你希望允许用户编辑数据值, 你可以象前面的例子一样将各字段放在组件内, 也可以简单地调用 `setEditable(true)` 方法, 这时各单元格就会自动地变为可编辑的字段.

我们首先来看看一个通常的 Table, 其中包含几个列, 类型为通常的 Java 类型, 一个 **Date**, 一个 **Boolean**, 以及一个 **String**.

```
// Create a table. It is by default not editable.
final Table table = new Table();

// Define the names and data types of columns.
table.addContainerProperty("Date", Date.class, null);
table.addContainerProperty("Work", Boolean.class, null);
table.addContainerProperty("Comments", String.class, null);

// Add a few items in the table.
for (int i=0; i<100; i++) {
    Calendar calendar = new GregorianCalendar(2008, 0, 1);
    calendar.add(Calendar.DAY_OF_YEAR, i);

    // Create the table row.
    table.addItem(new Object[] {calendar.getTime(),
        new Boolean(false),
        ""},
        new Integer(i)); // Item identifier
}
```

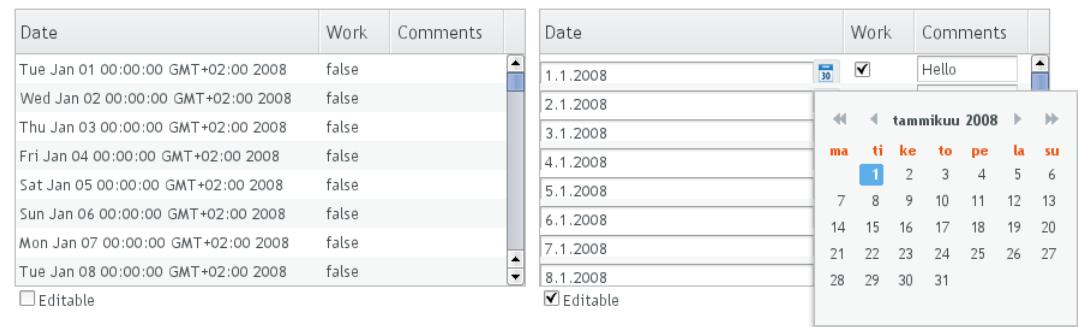
```
table.setPageLength(8);
layout.addComponent(table);
```

如果需要的话, 你立刻就可以将 Table 设置为可编辑模式. 我们下面扩展一下这个例子, 为它增加某种机制, 可以将 **Table** 在可编辑/不可编辑模式之间切换.

```
final CheckBox switchEditable = new CheckBox("Editable");
switchEditable.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            table.setEditable(((Boolean)event.getProperty()
                .getValue()).booleanValue());
        }
    });
switchEditable.setImmediate(true);
layout.addComponent(switchEditable);
```

现在, 如果你选中 CheckBox, Table 内的组件就会切换到可编辑模式, 见 图 5.48 “处于正常的、可编辑模式的 Table”.

图 5.48. 处于正常的、可编辑模式的 Table



Field 工厂

用于在 Table 内编辑某种特定类型数据的 Field 组件, 是由 Field 工厂创建的, Field 工厂实现 **TableFieldFactory** 接口. 默认实现是 **DefaultFieldFactory** 类, 它提供以下几种粗燥的对应:

表 5.2. DefaultFieldFactory 中, 数据类型与 Field 组件的对应

属性数据类型	对应的 Field 组件类
Date	DateField 组件.
Boolean	CheckBox 组件.
Item	Form (Vaadin 7 中已废弃). Form 中的各个 Field 是使用 FormFieldFactory , 根据项目各个属性来自动创建的. 这种属性类型更常用于 Form 之内, 在 Table 内的用途不大.
其他	TextField 组件. 如果从原始数据类型到字符串之间可以正确变换的话, 这个变换工作由 TextField 负责管理.

Field 工厂的详细介绍请参见 第 8.4 节“通过 Field 与项目的绑定来创建 Form”. 你也可以仅仅实现 **TableFieldFactory** 接口, 但我们推荐你按照自己的需要继承 **DefaultFieldFactory** 类. 在默认的实现类中, 数据类型与 Field 组件之间的对应关系由 `createFieldByPropertyType()` 方法决定(你可以会想看看其中的源代码), 不论是对 Table 还是 Form 都是如此.

在可编辑模式下跳转

在可编辑模式下, 编辑器 Field 组件可以拥有输入焦点. 按下 **Tab** 键可以将焦点移动到下一列, 如果目前已经是最最后一列, 那么会移动到下一个项目的第一列. 与此对应, 按下 **Shift+Tab** 键会将焦点反向移动. 如果焦点在最后一个可见项目的最后一列, 按下 **Tab** 键会将焦点移动到 Table 之外. 从第一个项目的第一列反向移动焦点, 会将焦点移动到 Table 自身. 对 Table 的某些变更, 比如变更表头, 表脚, 恢复一个列, 都可以将焦点从编辑器组件移动到 Table 自身.

很多情况下 Table 的默认行为可能不适应你的需求. 比如, 焦点还会跳转到只读的编辑器组件上, 还会不适当移出 Table 之外. 你可以实现更好的焦点跳转, 方法是用事件监听器来处理某些快捷键, 比如 **Tab**, **Arrow Up**, **Arrow Down**, 以及 **Enter**.

```
// Keyboard navigation
class KbdHandler implements Handler {
    Action tab_next = new ShortcutAction("Tab",
                                         ShortcutAction.KeyCode.TAB, null);
    Action tab_prev = new ShortcutAction("Shift+Tab",
                                         ShortcutAction.KeyCode.TAB,
                                         new int[] {ShortcutAction.ModifierKey.SHIFT});
    Action cur_down = new ShortcutAction("Down",
                                         ShortcutAction.KeyCode.ARROW_DOWN, null);
    Action cur_up   = new ShortcutAction("Up",
                                         ShortcutAction.KeyCode.ARROW_UP, null);
    Action enter   = new ShortcutAction("Enter",
                                         ShortcutAction.KeyCode.ENTER, null);
    public Action[] getActions(Object target, Object sender) {
        return new Action[] {tab_next, tab_prev, cur_down,
                            cur_up, enter};
    }

    public void handleAction(Action action, Object sender,
                           Object target) {
        if (target instanceof TextField) {
            // Move according to keypress
            int itemid = (Integer) ((TextField) target).getData();
            if (action == tab_next || action == cur_down)
                itemid++;
            else if (action == tab_prev || action == cur_up)
                itemid--;
            // On enter, just stay where you were. If we did
            // not catch the enter action, the focus would be
            // moved to wrong place.

            if (itemid >= 0 && itemid < table.size()) {
                TextField newTF = valueFields.get(itemid);
                if (newTF != null)
                    newTF.focus();
            }
        }
    }
}

// Panel that handles keyboard navigation
```

```

Panel navigator = new Panel();
navigator.addStyleName(Reindeer.PANEL_LIGHT);
navigator.addComponent(table);
navigator.addActionHandler(new KbdHandler());

```

在可编辑的 Table 中使用键盘来跳转焦点，主要的问题在于编辑器组件意识不到它所在的 Table。要找出它所在的父 Table，你可以通过组件的嵌套层次来向上查找，或者简单地使用 Field 组件的 `setData()` 方法来保存它所属的 Table。另一个问题是，你无法从 **Table** 组件得到编辑器 Field 组件。一种解决方法是使用某种额外的 collection，比如 **HashMap**，将项目 ID 与编辑器 Field 组件对应起来。

```

// Can't access the editable components from the table so
// must store the information
final HashMap<Integer,TextField> valueFields =
    new HashMap<Integer,TextField>();

```

向这个 Map 填充数据的是 **TableFieldFactory**，如下例所示。在这里你还需要设置 table 的参照，你还可以在这里设置焦点的初始位置。

```

table.setTableFieldFactory(new TableFieldFactory () {
    public Field createField(Container container, Object itemId,
        Object propertyId, Component uiContext) {
        TextField field = new TextField((String) propertyId);

        // User can only edit the numeric column
        if ("Source of Fear".equals(propertyId))
            field.setReadOnly(true);
        else { // The numeric column
            // The field needs to know the item it is in
            field.setData(itemId);

            // Remember the field
            valueFields.put((Integer) itemId, field);

            // Focus the first editable value
            if (((Integer)itemId) == 0)
                field.focus();
        }
        return field;
    }
});

```

由于编辑器 Field 组件并没有对 Table 内的全部项目生成出来，而只对可见项目及其前后少部分项目生成，因此问题变得更复杂了。比如，如果一个很大的可滚动的 Table 的最前面部分是可见的，那么此时最后一个项目的编辑器组件是不存在的。如果你希望焦点循环滚动，也就是从最后一个项目跳转到第一个项目，反之亦然，那么这个问题会变得很重要。

5.21.4. 列头和列脚

Table 同时支持列头和列脚；列头默认是有效的。

列头

列头显示在 Table 的最顶端。前面已经介绍过，你可以使用列头来拖动列，或改变列的宽度。默认情况下，除非使用 `setColumnHeader()` 方法明确地指定列头内容，否则列头内容为列对应的属性 ID。

```

// Define the properties
table.addContainerProperty("lastname", String.class, null);
table.addContainerProperty("born", Integer.class, null);
table.addContainerProperty("died", Integer.class, null);

// Set nicer header names
table.setColumnHeader("lastname", "Name");
table.setColumnHeader("born", "Born");
table.setColumnHeader("died", "Died");

```

列头的文字以及列头是否可见 取决于 列头模式。列头默认是可见的，但你可以使用 setColumnHeaderMode(Table.COLUMN_HEADER_MODE_HIDDEN) 方法禁用它。

列脚

Table 的列脚可用来显示某列的合计或平均值，等等信息。列脚默认是不可见的；你可以使用 setFooterVisible(true) 方法显示它。与列头不同，列头(译注：原文如此，应为“列脚”)默认是空的。你可以使用 setColumnFooter() 方法为列脚设置值。这个方法中，列通过对应的属性 ID 来指定。

下例演示如何计算一个列的平均值：

```

// Have a table with a numeric column
Table table = new Table("Custom Table Footer");
table.addContainerProperty("Name", String.class, null);
table.addContainerProperty("Died At Age", Integer.class, null);

// Insert some data
Object people[][] = {{"Galileo", 77},
                     {"Monnier", 83},
                     {"Vaisala", 79},
                     {"Oterma", 86}};
for (int i=0; i<people.length; i++)
    table.addItem(people[i], new Integer(i));

// Calculate the average of the numeric column
double avgAge = 0;
for (int i=0; i<people.length; i++)
    avgAge += (Integer) people[i][1];
avgAge /= people.length;

// Set the footers
table.setFooterVisible(true);
table.setColumnFooter("Name", "Average");
table.setColumnFooter("Died At Age", String.valueOf(avgAge));

// Adjust the table height a bit
table.setPageLength(table.size());

```

运行结果见图 5.49 “带列脚的 Table”。

图 5.49. 带列脚的 Table

Custom Table Footer	
NAME	DIED AT AGE
Galileo	77
Monnier	83
Väisälä	79
Oterma	86
Average	81.25

处理列头和列脚的鼠标点击事件

通常, 如果数据源是 **Sortable**, 并且排序功能未被禁止, 当用户点击列头时, Table 将按照这个列进行排序. 某些情况下, 你可能希望在用户点击列头时实现一些别的功能, 比如以某种方式选中整个列.

点击列头会激发 **HeaderClickEvent** 事件, 你可以使用 **Table.HeaderClickListener** 监听器来处理. 列头(以及列脚)上的点击事件, 与按钮的点击一样, 会立即发送到服务器端, 因此不必使用 `setImmediate()` 方法将 Table 设置为立即模式.

```
// Handle the header clicks
table.addHeaderClickListener(new Table.HeaderClickListener() {
    public void headerClick(HeaderClickEvent event) {
        String column = (String) event.getPropertyId();
        Notification.show("Clicked " + column +
                           " with " + event.getButtonName());
    }
});

// Disable the default sorting behavior
table.setSortDisabled(true);
```

设置一个点击事件监听器不会自动地将列头的排序动作禁止掉; 你需要调用 `setSortDisabled(true)` 来明确地禁止它. 当用户点击列头来调整列宽, 或拖动列的顺序时, 不会触发列头点击事件.

HeaderClickEvent 对象通过 `getPropertyId()` 方法提供被点击的列的 ID. `getButton()` 方法报告点击时按下的鼠标键是哪一个: `BUTTON_LEFT`, `BUTTON_RIGHT`, 或者 `BUTTON_MIDDLE`. `getButtonName()` 方法返回一个易读的鼠标键英文名: `"left"`, `"right"`, 或者 `"middle"`. `isShiftKey()`, `isCtrlKey()`, 等方法判断 **Shift**, **Ctrl**, **Alt** 等辅助案件在鼠标点击时是否有被按下.

点击列脚会激发 **FooterClickEvent** 事件, 你可以使用 **Table.FooterClickListener** 监听器处理. 列头点击后默认会排序, 但列脚点击后没有默认的处理动作. 除了这一点之外, 列脚点击事件的处理与列头点击事件是完全一样的.

5.21.5. 动态生成的列

你可能会希望某个列的内容由其他列计算而来. 或者你也可能希望用某种方式格式化某个列, 比如, 如果某些列显示货币值的情况. **ColumnGenerator** 接口可以用来为这样的列创建自定义的列生成器.

你可以使用 `addGeneratedColumn()` 方法将新生成的列添加到 **Table** 中。这个方法接受的参数是列 ID。通常你会希望列头内容更加用户友好一些，最好能国际化。在添加新生成的列之前，你可以使用 `addContainerProperty()` 方法设置列头和图标。

```
// Define table columns.
table.addContainerProperty(
    "date", Date.class, null, "Date", null, null);
table.addContainerProperty(
    "quantity", Double.class, null, "Quantity (l)", null, null);
table.addContainerProperty(
    "price", Double.class, null, "Price (e/l)", null, null);
table.addContainerProperty(
    "total", Double.class, null, "Total (e)", null, null);

// Define the generated columns and their generators.
table.addGeneratedColumn("date",
    new DateColumnGenerator());
table.addGeneratedColumn("quantity",
    new ValueColumnGenerator("%.2f l"));
table.addGeneratedColumn("price",
    new PriceColumnGenerator());
table.addGeneratedColumn("total",
    new ValueColumnGenerator("%.2f e"));

注意,即使你事先定义了列的顺序,addGeneratedColumn() 方法永远会将动态创建的列添加为最末列。你必须使用 setVisibleColumns() 方法来设置适当的列顺序。
```

```
table.setVisibleColumns(new Object[] {"date", "quantity", "price", "total"});
```

列生成器是实现了 **Table.ColumnGenerator** 接口及其中的 `generateCell()` 方法的对象。这个方法接受的参数是项目 ID、列 ID，以及 Table 对象。方法必须返回一个组件对象。

下例创建了一个列生成器，它使用一个格式化字符串来格式化一个 **Double** 值的 Field (与 **java.util.Formatter** 一样)。

```
/** Formats the value in a column containing Double objects. */
class ValueColumnGenerator implements Table.ColumnGenerator {
    String format; /* Format string for the Double values. */

    /**
     * Creates double value column formatter with the given
     * format string.
     */
    public ValueColumnGenerator(String format) {
        this.format = format;
    }

    /**
     * Generates the cell containing the Double value.
     * The column is irrelevant in this use case.
     */
    public Component generateCell(Table source, Object itemId,
        Object columnId) {
        // Get the object stored in the cell as a property
        Property prop =
            source.getItem(itemId).getItemProperty(columnId);
        if (prop.getType().equals(Double.class)) {
            Label label = new Label(String.format(format,
                new Object[] { (Double) prop.getValue() }));
            return label;
        }
    }
}
```

```

        // Set styles for the column: one indicating that it's
        // a value and a more specific one with the column
        // name in it. This assumes that the column name
        // is proper for CSS.
        label.addStyleName("column-type-value");
        label.addStyleName("column-" + (String) columnId);
        return label;
    }
    return null;
}
}

```

对于 Table 内的所有可见项目(更准确的说,应该是所有被缓存的项目),列生成器都会被调用.如果用户在 Table 内滚动到另一个位置,对于新的可见行,会动态生成列内容.当某个项目的值发生变更时,在可见(缓存)行中的列也会被自动生成.因此使用不同的行(项目)中的值来计算某个动态生成的单元格的值,通常是安全的.

当你将 Table 设置为 ,通常的 Field 会变为可编辑的 Field.当用户修改了 Field 内的值,动态生成的列会被自动更新.将一个包含动态生成的列的 Table 设置为可编辑模式会有点怪异. **Table** 的可编辑模式不会影响到动态生成的列.你有两个选择:要么在列生成器中生成可编辑的 Field 组件,或者,对于使用列生成器来格式化列内容的情况,可以在可编辑模式下删除这个生成器.下例使用后一种方案.

```

// Have a check box that allows the user
// to make the quantity and total columns editable.
final CheckBox editable = new CheckBox(
    "Edit the input values - calculated columns are regenerated");

editable.setImmediate(true);
editable.addClickListener(new ClickListener() {
    public void buttonClick(ClickEvent event) {
        table.setEditable(editable.booleanValue());

        // The columns may not be generated when we want to
        // have them editable.
        if (editable.booleanValue()) {
            table.removeGeneratedColumn("quantity");
            table.removeGeneratedColumn("total");
        } else { // Not editable
            // Show the formatted values.
            table.addGeneratedColumn("quantity",
                new ValueColumnGenerator("%.2f l"));
            table.addGeneratedColumn("total",
                new ValueColumnGenerator("%.2f e"));
        }
        // The visible columns are affected by removal
        // and addition of generated columns so we have
        // to redefine them.
        table.setVisibleColumns(new Object[] {"date", "quantity",
            "price", "total", "consumption", "dailycost"});
    }
});

```

你还需要将 Field 编辑组件设置为 模式,这样才能在 Field 编辑组件失去焦点时,让数据值立即更新.你可以使用定制的 **TableFieldFactory**,来将Field 编辑组件设置为 模式,如下例所示,我们在这里只是简单地继承默认实现,然后设置了编辑组件的模式:

```

public class ImmediateFieldFactory extends DefaultFieldFactory {
    public Field createField(Container container,
                            Object itemId,
                            Object propertyId,
                            Component uiContext) {
        // Let the DefaultFieldFactory create the fields...
        Field field = super.createField(container, itemId,
                                         propertyId, uiContext);

        // ...and just set them as immediate.
        ((AbstractField)field).setImmediate(true);

        return field;
    }
}
...
table.setTableFieldFactory(new ImmediateFieldFactory());

```

如果你使用列生成器来生成 Field 编辑组件, 你就没必要使用上面那样的 Field 工厂了, 但是当然, 你必须为正常模式和可编辑模式两种情况生成 Field.

图 5.50 “带有动态生成的列的 Table, 正常模式和可编辑模式”展示了两个 Table, 其中带有计算生成的列(蓝色)以及简单地格式化的列(黑色), 这两种列使用列生成器动态创建.

图 5.50. 带有动态生成的列的 Table, 正常模式和可编辑模式

Date	Quantity (l)	Price (€/l)	Total (€)	Consumption (l/day)	Daily Cost (€/day)
2005-02-19	44,96 l	1,14 €	51,21 €	N/A	N/A
2005-03-30	44,91 l	1,20 €	53,67 €	1,15 l	1,38 €
2005-04-20	42,96 l	1,14 €	49,06 €	2,05 l	2,34 €
2005-05-23	47,37 l	1,17 €	55,28 €	1,44 l	1,68 €
2005-06-06	35,34 l	1,17 €	41,52 €	2,52 l	2,97 €
2005-06-30	16,07 l	1,24 €	20,00 €	0,67 l	0,83 €
2005-07-02	36,40 l	0,99 €	36,19 €	18,20 l	18,10 €

Date	Quantity (l)	Price (€/l)	Total (€)	Consumption (l/day)	Daily Cost (€/day)
2005-02-19	44.96	1,14 €	51.21	N/A	N/A
2005-03-30	44.91	1,20 €	53.67	1,15 l	1,38 €
2005-04-20	42.96	1,14 €	49.06	2,05 l	2,34 €
2005-05-23	47.37	1,17 €	55.28	1,44 l	1,68 €
2005-06-06	35.34	1,17 €	41.52	2,52 l	2,97 €
2005-06-30	16.07	1,24 €	20.0	0,67 l	0,83 €
2005-07-02	36.4	0,99 €	36.19	18,20 l	18,10 €

5.21.6. 列的格式控制

Table 内显示的属性值, 其输出格式通常使用各个属性的 `toString()` 方法来控制. 自定义一个列的格式可以由以下几种方法实现:

- 使用 **ColumnGenerator** 来生成新的、格式化的列. 原来的列需要被设置为不可见. 详情请参见第 5.21.5 节“动态生成的列”.
- 使用 **PropertyFormatter** 作为 Table 和数据属性之间的代理. 这种方法通常需要在 Table 中使用一个间接的数据容器.
- 覆盖 **Table** 类默认的 `formatPropertyValue()` 方法.

一般来说, 使用 **PropertyFormatter** 要比覆盖 `formatPropertyValue()` 方法笨拙一些, 所以这里我们不详细介绍这种方案.

覆盖 `formatPropertyValue()` 方法的例子如下:

```
// Create a table that overrides the default
// property (column) format
final Table table = new Table("Formatted Table") {
    @Override
    protected String formatPropertyValue(Object rowId,
                                         Object colId, Property property) {
        // Format by property type
        if (property.getType() == Date.class) {
            SimpleDateFormat df =
                new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
            return df.format((Date)property.getValue());
        }

        return super.formatPropertyValue(rowId, colId, property);
    }
};

// The table has some columns
table.addContainerProperty("Time", Date.class, null);
...
Fill the table with data ...
```

你可以使用 `colId` 参数来区分不同的列, 这个参数是列对应的属性 ID. **DecimalFormat** 很适用于格式化十进制小数值.

```
... in formatPropertyValue() ...
} else if ("Value".equals(pid)) {
    // Format a decimal value for a specific locale
    DecimalFormat df = new DecimalFormat("#.00",
                                         new DecimalFormatSymbols(locale));
    return df.format((Double) property.getValue());
}
...
table.addContainerProperty("Value", Double.class, null);
```

带有格式化的日期列和格式化的十进制值列的 Table, 见图 5.51 “Table 中格式化的列”.

图 5.51. Table 中格式化的列

Formatted Table		
TIME	VALUE	MESSAGE
1970-01-01 02:00:00	708,42	Msg #55
1970-03-29 05:16:33	44,31	Msg #33
1970-07-22 04:40:13	741,61	Msg #1
1970-09-01 12:11:49	757,91	Msg #36
1970-12-21 02:32:50	793,82	Msg #92
1971-01-13 05:25:15	700,79	Msg #65

你可以通过 **CellStyleGenerator**, 来以 CSS 的方式进一步控制 Table 中各行, 各列, 各个独立的单元格的样式. 详情请参见第 5.21.7 节 “CSS 样式规则”.

5.21.7. CSS 样式规则

Table 的样式规则如下.

```
.v-table {}
.v-table-header-wrap {}
.v-table-header {}
.v-table-header-cell {}
.v-table-resizer {} /* Column resizer handle. */
.v-table-caption-container {}

.v-table-body {}
.v-table-row-spacer {}
.v-table-table {}
.v-table-row {}
.v-table-cell-content {}
```

注意, Table 内某些元素的宽度和高度是动态计算的, 无法通过 CSS 来设置.

设置独立的单元格样式

Table.CellStyleGenerator 接口允许你为 Table 内每一个独立的单元格设置 CSS 样式. 你需要实现这个接口的 `getStyle()` 方法, 这个方法接受的参数是行(项目)和列(属性)的 ID, 并为这个单元格返回一个样式名称. 返回的样式名会被自动添加一个前缀字符串 "v-table-cell-content-".

`getStyle()` 方法也会对每一行本身调用, 此时 `propertyId` 参数为 `null`. 这种机制用于为行本身设置样式.

或者, 你也可以使用 **Table.ColumnGenerator** (参见 第 5.21.5 节 “动态生成的列”) 来为每个单元格生成 UI 组件, 并为这些 UI 组件添加样式.

```
Table table = new Table("Table with Cell Styles");
table.addStyleName("checkerboard");

// Add some columns in the table. In this example, the property
// IDs of the container are integers so we can determine the
// column number easily.
table.addContainerProperty("0", String.class, null, "", null, null);
for (int i=0; i<8; i++)
    table.addContainerProperty(""+(i+1), String.class, null,
        String.valueOf((char) (65+i)), null, null);

// Add some items in the table.
table.addItem(new Object[] {
```

```

    "1", "R", "N", "B", "Q", "K", "B", "N", "R"}, new Integer(0));
table.addItem(new Object[]{ 
    "2", "P", "P", "P", "P", "P", "P", "P"}, new Integer(1));
for (int i=2; i<6; i++) {
    table.addItem(new Object[]{String.valueOf(i+1),
        "", "", "", "", "", "", ""}, new Integer(i));
table.addItem(new Object[]{ 
    "7", "P", "P", "P", "P", "P", "P", "P"}, new Integer(6));
table.addItem(new Object[]{ 
    "8", "R", "N", "B", "Q", "K", "B", "N", "R"}, new Integer(7));
table.setPageLength(8);

// Set cell style generator
table.setCellStyleGenerator(new Table.CellStyleGenerator() {
    public String getStyle(Object itemId, Object propertyId) {
        // Row style setting, not relevant in this example.
        if (propertyId == null)
            return "green"; // Will not actually be visible

        int row = ((Integer)itemId).intValue();
        int col = Integer.parseInt((String)propertyId);

        // The first column.
        if (col == 0)
            return "rowheader";

        // Other cells.
        if ((row+col)%2 == 0)
            return "black";
        else
            return "white";
    }
});
```

然后你可以控制单元格的样式, 如下例:

```

/* Center the text in header. */
.v-table-header-cell {
    text-align: center;
}

/* Basic style for all cells. */
.v-table-checkerboard .v-table-cell-content {
    text-align: center;
    vertical-align: middle;
    padding-top: 12px;
    width: 20px;
    height: 28px;
}

/* Style specifically for the row header cells. */
.v-table-cell-content-rowheader {
    background: #E7EDF3
    url(..../default/table/img/header-bg.png) repeat-x scroll 0 0;
}

/* Style specifically for the "white" cells. */
.v-table-cell-content-white {
    background: white;
    color: black;
```

```

}

/* Style specifically for the "black" cells. */
.v-table-cell-content-black {
    background: black;
    color: white;
}

```

Table 的外观参见 图 5.52 “Table 的单元格样式生成器”.

图 5.52. Table 的单元格样式生成器

	A	B	C	D	E	F	G	H
1	R	N	B	Q	K	B	N	R
2	P	P	P	P	P	P	P	P
3								
4								
5								
6								
7	P	P	P	P	P	P	P	P
8	R	N	B	Q	K	B	N	R

5.22. Tree

Tree 组件以一种自然的方式来表现层级式关系的数据，比如文件系统，或 BBS 上的消息主题. Vaadin 中的 **Tree** 组件类似于大多数现代桌面 UI 开发工具中的 Tree 组件，比如，目录系统.

Tree 组件的一种常见用途是现实层级式菜单，类似于显示在屏幕左侧的菜单，参见 图 5.53 “使用 **Tree** 组件作为菜单”，也可用于显示文件系统或其他层级式的数据集. *menu* 样式可以使 Tree 的显示风格更适合于这种目的.

```

final Object[][] planets = new Object[][]{
    new Object[]{"Mercury"},
    new Object[]{"Venus"},
    new Object[]{"Earth", "The Moon"},
    new Object[]{"Mars", "Phobos", "Deimos"},
    new Object[]{"Jupiter", "Io", "Europa", "Ganymedes",
        "Callisto"},
    new Object[]{"Saturn", "Titan", "Tethys", "Dione",
        "Rhea", "Iapetus"},
    new Object[]{"Uranus", "Miranda", "Ariel", "Umbriel",
        "Titania", "Oberon"},
    new Object[]{"Neptune", "Triton", "Proteus", "Nereid",
        "Larissa"}};

Tree tree = new Tree("The Planets and Major Moons");

/* Add planets as root items in the tree. */
for (int i=0; i<planets.length; i++) {
    String planet = (String) (planets[i][0]);
    tree.addItem(planet);
}

```

```

if (planets[i].length == 1) {
    // The planet has no moons so make it a leaf.
    tree.setChildrenAllowed(planet, false);
} else {
    // Add children (moons) under the planets.
    for (int j=1; j<planets[i].length; j++) {
        String moon = (String) planets[i][j];

        // Add the item as a regular item.
        tree.addItem(moon);

        // Set it to be a child.
        tree.setParent(moon, planet);

        // Make the moons look like leaves.
        tree.setChildrenAllowed(moon, false);
    }

    // Expand the subtree.
    tree.expandItemsRecursively(planet);
}
}

main.addComponent(tree);

```

图 5.53 “使用 Tree 组件作为菜单” 是上面示例代码的运行结果.

你可以使用 **Tree** 组件的属性值来获得或设定当前选中的项目，也就是使用 `getValue()` 和 `setValue()` 方法。当用户点击 Tree 内某个项目时，Tree 会收到 **ValueChangeEvent** 事件，你可以使用 **ValueChangeListener** 监听器来处理这个事件。如果要在用户点击后立即接收这个事件，你需要对 Tree 设置 `setImmediate(true)`。

Tree 组件与 **Table** 组件类似，使用 **Container** 数据源，不同的是，Tree 还需要获取数据之间的层级信息，这个信息由 **HierarchicalContainer** 管理。容器内的项目可以是容器所允许的任意类型。默认的容器，以及它的 `addItem()` 方法假设项目是字符串类型，并将这个字符串值用作项目 ID。

5.23. MenuBar

MenuBar 组件用于创建水平排列的下拉菜单，类似于桌面应用程序中的主菜单。

图 5.54. Menu Bar

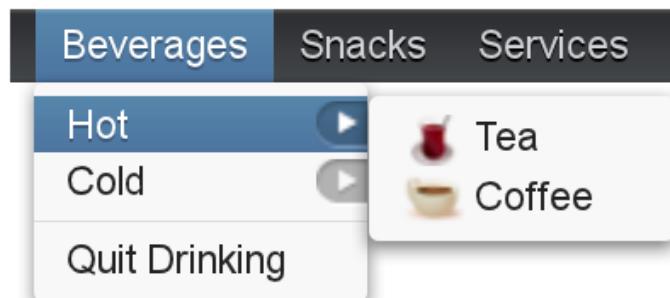
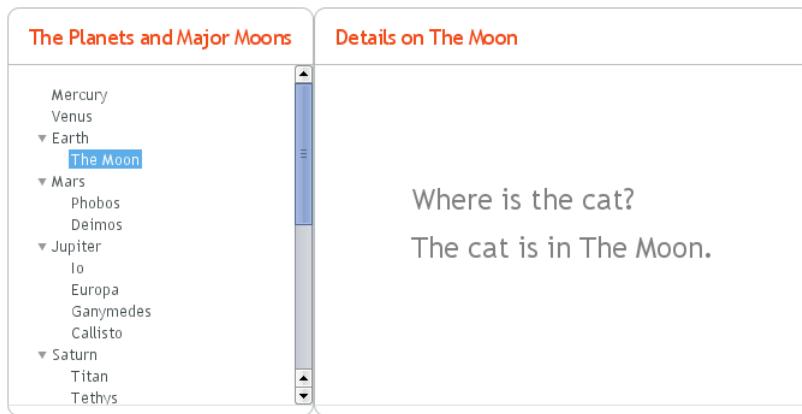


图 5.53. 使用 **Tree** 组件作为菜单

创建菜单

MenuBar 组件的创建过程第一步如下:

```
MenuBar menubar = newMenuBar();
main.addComponent(menubar);
```

然后你需要使用 `addItem()` 方法将最顶层菜单项添加到 **MenuBar** 对象中。这个方法接受的参数是一个字符串类型的标题，一个图标资源，和一个对应的命令。图标和命令不是必须参数，可以为 `null`。`addItem()` 方法返回一个 **MenuBar.MenuItem** 对象，你可以再向它添加子菜单项。**MenuItem** 也有一个相同的 `addItem()` 方法。

示例 (关于命令的部分，后文会解释):

```
// A top-level menu item that opens a submenu
MenuItem drinks = barmenu.addItem("Beverages", null, null);

// Submenu item with a sub-submenu
MenuItem hots = drinks.addItem("Hot", null, null);
hots.addItem("Tea",
    new ThemeResource("icons/tea-16px.png"), mycommand);
hots.addItem("Coffee",
    new ThemeResource("icons/coffee-16px.png"), mycommand);

// Another submenu item with a sub-submenu
MenuItem colds = drinks.addItem("Cold", null, null);
colds.addItem("Milk", null, mycommand);
colds.addItem("Weissbier", null, mycommand);

// Another top-level item
MenuItem snacks = barmenu.addItem("Snacks", null, null);
snacks.addItem("Weisswurst", null, mycommand);
snacks.addItem("Bratwurst", null, mycommand);
snacks.addItem("Currywurst", null, mycommand);

// Yet another top-level item
MenuItem servs = barmenu.addItem("Services", null, null);
servs.addItem("Car Service", null, mycommand);
```

菜单选择的处理

当用户在菜单中选中一个项目时, 菜单的选择以执行一个 **命令** 的方式来处理. 命令是一个回调类, 它实现 **MenuBar.Command** 接口.

```
// A feedback component
final Label selection = new Label("-");
main.addComponent(selection);

// Define a common menu command for all the menu items.
MenuBar.Command mycommand = new MenuBar.Command() {
    public void menuSelected(MenuItem selectedItem) {
        selection.setValue("Ordered a " +
            selectedItem.getText() +
            " from menu.");
    }
};
```

菜单项

菜单项的属性包括标题, 图标, 是否有效, 是否可见, 以及描述信息(提示信息). 这些属性的含义与组件中的属性相同.

使用 `addItem()` 或 `addItemBefore()` 方法向一个菜单项添加下级菜单项, 就创建出了子菜单.

`command` 属性是 **MenuBar.Command** 类型的对象, 当对应的菜单项被选中时, 它会被调用. `menuSelected()` 回调方法的参数是被点击的菜单项.

菜单可以包含 分隔项, 在菜单项上使用 `addSeparatorBefore()` 或 `addSeparator()` 方法可以在菜单项之前或之后创建分隔项.

```
MenuItem drinks = barmenu.addItem("Beverages", null, null);
...
// A sub-menu item after a separator
drinks.addSeparator();
drinks.addItem("Quit Drinking", null, null);
```

在菜单项上使用 `setCheckable()` 方法, 允许这个菜单项 `checkable`, 用户就可以点击这个菜单项, 使它在选中和非选中状态之间切换. 你可以使用 `setChecked()` 方法设置选中状态. 注意, 如果这个菜单项带有命令, 那么选中状态不会自动切换, 你需要在代码中明确地切换选中状态.

菜单项还有其他一些属性, 详情请阅读 API 文档.

CSS 样式规则

```
.v-menubar { }
.v-menubar-submenu { }
.v-menubar-menuitem { }
.v-menubar-menuitem-caption { }
.v-menubar-menuitem-selected { }
.v-menubar-submenu-indicator { }
```

MenuBar 的最外层样式是 `.v-menubar`. 每一个菜单项的样式通常为 `.v-menubar-menuitem`, 如果菜单项被选中, 也就是鼠标指针移动到它上方时, 还会额外带有一个 `.v-menubar-selected` 样式. 项目标题包含在 `v-menubar-menuitem-caption` 之内. 最顶层菜单条之内的菜单项, 直接位于组件的 HTML 元素之内.

子菜单是浮动的 `v-menubar-submenu` 元素，位于 `MenuBar` 的 HTML 元素之外。因此，你不应该试图在 `MenuBar` 的 HTML 元素之内匹配子菜单的弹出元素。在子菜单之下如果包含更下级子菜单，则以一个指示器来表示，指示器的样式为 `v-menubar-submenu-indicator`。

控制菜单项样式

与其他组件一样，你可以使用 `setStyleName()` 方法设置菜单项的 CSS 样式名。样式名会被自动添加 `v-menubar-menuitem-` 前缀。由于 `MenuBar` 不能以任何方式来指出前次选中的项目，你可以高亮显示选中的项目，来自行实现这个功能，参见下例。但是，要注意菜单项的 `□` 样式，也就是，`v-menubar-menuitem-selected`，这个样式已经被用于指示鼠标在菜单内的移动位置。

```
MenuBar barmenu = new MenuBar();
barmenu.addStyleName("mybarmenu");
layout.addComponent(barmenu);

// A feedback component
final Label selection = new Label("-");
layout.addComponent(selection);

// Define a common menu command for all the menu items
MenuBar.Command mycommand = new MenuBar.Command() {
    MenuItem previous = null;

    public void menuSelected(MenuItem selectedItem) {
        selection.setValue("Ordered a " +
            selectedItem.getText() +
            " from menu.");

        if (previous != null)
            previous.setStyleName(null);
        selectedItem.setStyleName("highlight");
        previous = selectedItem;
    }
};

// Put some items in the menu
barmenu.addItem("Beverages", null, mycommand);
barmenu.addItem("Snacks", null, mycommand);
barmenu.addItem("Services", null, mycommand);
```

你可以使用下面的 CSS 来控制高亮部分的样式：

```
.mybarmenu .v-menubar-menuitem-highlight {
    background: #000040; /* Dark blue */
}
```

5.24. Upload

Upload 组件可供用户上传文件到服务器端。这个组件显示一个文件名输入框，一个文件选择按钮，以及一个上传提交按钮。用户可以在文件名输入框中输入文件名，也可以点击 浏览 按钮来选择文件。选择文件之后，用户可以点击上传提交按钮将文件发送到服务器端。

上传的文件在服务器端需要一个接收器，接收器需要实现 `Upload.Receiver` 接口，并提供一个输出流，文件内容将由服务器写入到这个输出流中。

```
Upload upload = new Upload("Upload it here", receiver);
```

图 5.55. Upload 组件



你可以使用 `setButtonCaption()` 方法设置上传按钮的文字。注意，修改浏览按钮的文字或外观是困难的。这是浏览器的安全特性。浏览按钮的语言由浏览器决定，因此，如果你希望 **Upload** 组件内的语言保持一致，你可能必须使用与应用程序相同的语言。

```
upload.setButtonCaption("Upload Now");
```

你也可以在 theme 中使用 `.v-upload .v-button {display: none}` 样式来隐藏上传按钮，然后实现自定义的逻辑开始上传，并调用 `startUpload()` 方法来启动上传动作。如果 **Upload** 组件被设置为 `setImmediate(true)`，文件被选中之后就会立即开始上传。

接收上传的数据

被上传的文件通常保存为文件系统中的文件，或保存在数据库中，或保存为内存中的临时对象。**Upload** 组件将接收到的数据写入一个 **java.io.OutputStream**，因此你有很大的自由来决定如何处理上传的内容。

要使用 **Upload** 组件，你需要实现 **Upload.Receiver** 接口。当用户按下提交按钮时，接收器中的 `receiveUpload()` 方法会被调用。这个方法必须返回一个 **OutputStream**。为了实现这一点，通常创建一个文件，或者一个内存中的缓冲区作为流的写入对象。这个方法接受的参数是被上传文件的文件名和 MIME 类型，这些信息由浏览器报告给服务器。

在上传过程中，可以通过 **Upload.ProgressListener** 监听器来监视上传进度。监听器的 `updateProgress()` 方法接受的参数是已读入的字节数，以及内容的总长度。内容的总长度由浏览器报告给服务器，这个数字不一定是可靠的，而且有可能为 -1，代表长度未知。因此，推荐在上传进度监听器中跟踪上传进度，并检查文件大小。上传可以中途停止，方法是在 **Upload** 组件上调用 `interruptUpload()` 方法。你也许会希望使用 **ProgressBar** 来显示上传进度，如果上传内容的长度未知，可将进度条设置为模糊模式。

当上传结束后，无论成功还是不成功，**Upload** 组件都会激发 **Upload.FinishedEvent** 事件，你可以使用 **Upload.FinishedListener** 监听器来处理这个事件。事件对象中包含的信息有，文件名，MIME 类型，以及文件最终长度。**Upload.FailedEvent** 和 **Upload.SucceededEvent** 事件更为详细一些，分别会在上传失败或上传成功时被调用。

下例上传一个图片到(UNIX)文件系统的 `/tmp/uploads` 目录下(这个目录必须存在，否则上传会失败)。然后将上传的图片显示在 **Image** 组件中。

```
// Show uploaded file in this placeholder
final Embedded image = new Embedded("Uploaded Image");
image.setVisible(false);

// Implement both receiver that saves upload in a file and
// listener for successful upload
class ImageUploader implements Receiver, SucceededListener {
    public File file;

    public OutputStream receiveUpload(String filename,
                                      String mimeType) {
        // Create upload stream
        FileOutputStream fos = null; // Stream to write to
        try {
            fos = new FileOutputStream(new File(filename));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        return fos;
    }

    public void onSuccess(File file) {
        image.setImage(file);
    }
}
```

```
// Open the file for writing.
file = new File("/tmp/uploads/" + filename);
fos = new FileOutputStream(file);
} catch (final java.io.FileNotFoundException e) {
    new Notification("Could not open file<br>",
                    e.getMessage(),
                    Notification.Type.ERROR_MESSAGE)
        .show(Page.getCurrent());
    return null;
}
return fos; // Return the output stream to write to
}

public void uploadSucceeded(SucceededEvent event) {
    // Show the uploaded file in the image viewer
    image.setVisible(true);
    image.setSource(new FileResource(file));
}
};

ImageUploader receiver = new ImageUploader();

// Create the upload with a caption and set receiver later
Upload upload = new Upload("Upload Image Here", receiver);
upload.setButtonCaption("Start Upload");
upload.addSucceededListener(receiver);

// Put the components in a panel
Panel panel = new Panel("Cool Image Storage");
Layout panelContent = new VerticalLayout();
panelContent.addComponents(upload, image);
panel.setContent(panelContent);
```

注意，上例并没有检查上传文件的类型，如果文件不是图片，可能会导致错误。例子程序还假设通过文件扩展名能够正确地解析得到文件 MIME 类型。图片上传完成后，运行结果类似于图 5.56 “图片上传示例”。

图 5.56. 图片上传示例



CSS 样式规则

```
.v-upload { }
.gwt-FileUpload { }
.v-button { }
.v-button-wrap { }
.v-button-caption { }
```

Upload 组件的最外层样式为 v-upload. 上传按钮的 HTML 元素结构与通常的 **Button** 组件一样.

5.25. ProgressBar

ProgressBar 组件可用来显示某个任务的执行进度. 进度以 0.0 和 1.0 之间的浮点数值来表示.

图 5.57. ProgressBar 组件



要显示 **Upload** 组件上传文件的进度, 你可以在 ProgressListener 监听器中更新 ProgressBar 的状态.

如果在后台线程中变更了 ProgressBar 的进度位置, 这个变化不会立即显示到浏览器中. 你需要使用客户端轮询技术或服务器端 PUSH 技术来更新浏览器端状态. 你可以使用当前 UI 实例的

`setPollInterval()` 方法打开轮询功能。关于服务器端 PUSH 功能的使用方法，详情请参见第 11.16 节“服务器端 PUSH”。无论你使用哪一种方法来更新 UI，很重要的一个问题是要锁定用户 Session，方法是在 `access()` 调用之内修改 `ProgressBar` 的值，如下例所示，这个问题的详情请参见第 11.16.3 节“在其他线程中访问 UI”。

```
final ProgressBar bar = new ProgressBar(0.0f);
layout.addComponent(bar);

layout.addComponent(new Button("Increase",
    new ClickListener() {
        @Override
        public void buttonClick(ClickEvent event) {
            float current = bar.getValue();
            if (current < 1.0f)
                bar.setValue(current + 0.10f);
        }
    }));
}
```

模糊模式

在模糊模式下，会连续不断地显示一个进度不明的指示器。在内建的 Theme 中，模糊模式的指示器是一个圆环形，进度值在模糊模式下是无意义的。

```
ProgressBar bar = new ProgressBar();
bar.setIndeterminate(true);
```

图 5.58. 模糊模式下的 `ProgressBar`



高负荷计算

进度指示器常用来表示服务器端的高负荷计算工作的进度状况，这个计算通常运行在后台线程中。UI，包括 `ProgressBar`，可以使用客户端轮询技术或服务器端 PUSH 技术来更新。此时，你必须确保线程安全，最简单的方法是在 `Runnable` 接口的发起的 `UI.access()` 调用之内更新 UI，详情请参见第 11.16.3 节“在其他线程中访问 UI”。

下例中，我们在服务器端创建一个线程来进行某种“高负荷工作”，并使用客户端轮询方式来更新 UI。线程所需要做的只是使用 `setValue()` 方法设置 `ProgressBar` 的值，当浏览器向服务器发起轮询时，当前的进度就会自动显示出来。

```
HorizontalLayout barbar = new HorizontalLayout();
layout.addComponent(barbar);

// Create the indicator, disabled until progress is started
final ProgressBar progress = new ProgressBar(new Float(0.0));
progress.setEnabled(false);
barbar.addComponent(progress);

final Label status = new Label("not running");
barbar.addComponent(status);

// A button to start progress
final Button button = new Button("Click to start");
```

```

layout.addComponent(button);

// A thread to do some work
class WorkThread extends Thread {
    // Volatile because read in another thread in access()
    volatile double current = 0.0;

    @Override
    public void run() {
        // Count up until 1.0 is reached
        while (current < 1.0) {
            current += 0.01;

            // Do some "heavy work"
            try {
                sleep(50); // Sleep for 50 milliseconds
            } catch (InterruptedException e) {}

            // Update the UI thread-safely
            UI.getCurrent().access(new Runnable() {
                @Override
                public void run() {
                    progress.setValue(new Float(current));
                    if (current < 1.0)
                        status.setValue("'" +
                            ((int)(current*100)) + "% done");
                    else
                        status.setValue("all done");
                }
            });
        }
    }

    // Show the "all done" for a while
    try {
        sleep(2000); // Sleep for 2 seconds
    } catch (InterruptedException e) {}

    // Update the UI thread-safely
    UI.getCurrent().access(new Runnable() {
        @Override
        public void run() {
            // Restore the state to initial
            progress.setValue(new Float(0.0));
            progress.setEnabled(false);

            // Stop polling
            UI.getCurrent().setPollInterval(-1);

            button.setEnabled(true);
            status.setValue("not running");
        }
    });
}

// Clicking the button creates and runs a work thread
button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        final WorkThread thread = new WorkThread();
        thread.start();
    }
});

```

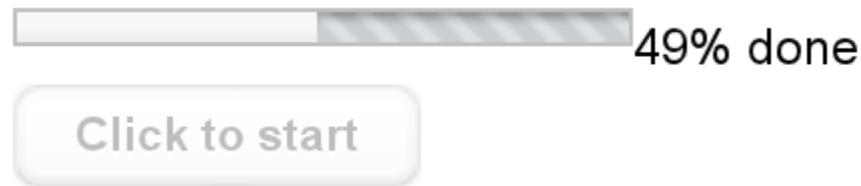
```
// Enable polling and set frequency to 0.5 seconds
UI.getCurrent().setPollInterval(500);

// Disable the button until the work is done
progress.setEnabled(true);
button.setEnabled(false);

status.setValue("running...");
}
});

上例的运行结果见图 5.59 “执行高负荷工作”。
```

图 5.59. 执行高负荷工作



CSS 样式规则

```
.v-progressbar, v-progressbar-indeterminate {}
.v-progressbar-wrapper {}
.v-progressbar-indicator {}
```

ProgressBar 的最外层样式是 v-progressbar. 动画部分是 v-progressbar-wrapper 样式元素的背景图, 默认是一个动画 GIF 图片. 进度是 wrapper 内的一个 v-progressbar-indicator 样式的元素, 因此显示在 wrapper 的顶部. 当进度元素增长时, 它会越来越多地将背景动画图片覆盖住.

在模糊模式下, 顶层元素还带有 v-progressbar-indeterminate 样式. 内建的 Theme 只在顶层元素中简单地显示一个动画 GIF, 然后将下层元素全部禁用.

5.26. Slider

Slider 是一个垂直条或水平条, 用户可以使用鼠标拖动其中的滑块, 即可在指定的范围之内设定一个数值. 鼠标拖动滑块时, 会显示对应的数值.

Slider 有很多不同版本的构造器, 这些构造器接受以下参数的组合: 标题, 最小值, 最大值, 精度, 以及 slider 的方向.

```
// Create a vertical slider
final Slider vertslider = new Slider(1, 100);
vertslider.setOrientation(SliderOrientation.VERTICAL);
```

Slider 的属性

min

Slider 可选择的数值范围最小值. 默认为 0.0.

max

Slider 可选择的数值范围最大值. 默认为 100.0.

resolution

小数点后数字的位数. 默认为 0.

orientation

方向可以是水平 (*SliderOrientation.HORIZONTAL*) 或垂直 (*SliderOrientation.VERTICAL*). 默认为水平.

由于 **Slider** 是一个 Field 组件, 你可以使用 **ValueChangeListener** 监听器来处理它的值变更事件. **Slider** Field 的值是 **Double** 对象.

```
// Shows the value of the vertical slider
final Label vertvalue = new Label();
vertvalue.setSizeUndefined();

// Handle changes in slider value.
vertslder.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            double value = (Double) vertslder.getValue();

            // Use the value
            box.setHeight((float) value, Sizeable.UNITS_PERCENTAGE);
            vertvalue.setValue(String.valueOf(value));
        }
    });
}

// The slider has to be immediate to send the changes
// immediately after the user drags the handle.
vertslder.setImmediate(true);
```

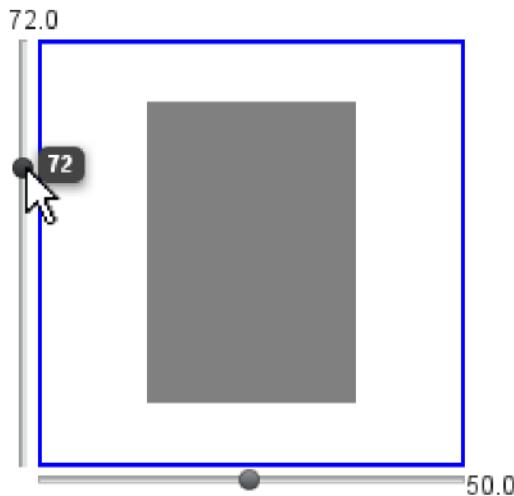
你可以使用 **Slider** 类的 `setValue()` 方法来设置 Slider 的值, 方法参数为 double 类型值. 这个方法可能会抛出 **ValueOutOfBoundsException** 异常, 你必须处理这个异常.

```
// Set the initial value. This has to be set after the
// listener is added if we want the listener to handle
// also this value change.
try {
    vertslder.setValue(50.0);
} catch (ValueOutOfBoundsException e) {
}
```

或者你也可以使用通常的 `setValue(Object)` 方法, 这个方法不会进行值的上下边界检查.

图 5.60 “**Slider** 组件” 显示了垂直(与上面的示例代码一致)和水平的 Slider, 这两个 Slider 控制方框的尺寸. Slider 的值也分别显示在 Label 组件中.

图 5.60. Slider 组件



CSS 样式规则

```
.v-slider {}
.v-slider-base {}
.v-slider-handle {}
```

Slider 组件的最外层样式为 `v-slider`. Slider 条的样式为 `v-slider-base`. 虽然滑块比 Slider 条要更高(对于水平条情况)或更宽(对于垂直条的情况), 但滑块的 HTML 元素还是包含在 Slider 条的 HTML 元素之内. 滑块的外观表现是通过它的 CSS 属性 `background` 指定的背景图片来实现的.

5.27. Calendar

Calendar 组件用于组织和显示日历上的事件. 日历的主要功能包括:

- 月单位, 周单位, 日单位的显示模式
- 支持两种事件: 全日事件, 和某个时间范围内的事件
- 所有事件, 直接来自于 **Container**, 或者由 Event Provider 提供.
- 控制可见的日期范围
- 通过鼠标拖动来选择或编辑日期/时刻的范围
- 将事件拖放到日历中
- 支持本地化和时区

日历中的元素, 比如日期和周的标题, 事件, 用户对它们的操作使用事件监听器来处理. 此外, 日期/时间范围的选择, 事件的拖放, 以及事件长短的变更都可以在服务器端监听. 周单位显示模式带有遍历按钮, 可以将时间向前或向后翻滚. 这些动作也由服务器来监听. 可以使用事件处理器来实现自定义的遍历动作, 详情请参见 第 5.27.10 节“定制日历”.

日历的数据源实际上可以是任意的对象, 因为日历中的事件是有组件动态请求的. 你可以将日历绑定到一个 Vaadin 容器, 或者实现 Event Provider, 来将日历绑定到任意的数据源.

Calendar 默认尺寸为未定义, 你通常会希望指定一个固定尺寸或相对尺寸, 如下例.

```
Calendar cal = new Calendar("My Calendar");
cal.setWidth("600px");
cal.setHeight("300px");
```

创建完日历后, 你需要为它设置时间范围, 时间范围同时也控制日历的显示模式, 你还需要为日历中的事件设置数据源.

5.27.1. 日期范围与显示模式

Vaadin Calendar 有两种显示模式, 显示哪一种取决于日历的日期范围. 周单位显示模式默认显示一周. 它可以显示一周 7 天之内的信息, 而且它还用作日单位显示. 显示模式由日历的日期范围来决定, 日期范围是指日历的开始日期和结束日期. 如果日期范围超过 1 周(7 日), 日历将显示为月单位显示模式. 日期范围以毫秒为单位进行计算.

月单位显示模式, 见图 5.61 “月单位显示模式, 包含全日事件和通常事件”, 可以很容易地控制所有类型的事件, 但它最适用于那些持续时间超过 1 日或多日的事件. 你可以拖动事件来移动它. 在图中, 你可以看到两个较长的事件, 显示为蓝色和绿色的高亮背景色. 其他几个标记代表较短的事件, 其持续时间不到 24 小时. 在月单位显示模式中, 这样的事件不可以通过拖动来移动.

在图 5.62 “周单位显示模式” 中, 你可以看到 4 个通常的日事件, 在时间线表格上方还有全日事件.

下面, 我们将日历设置为只显示 1 日, 也就是当前日.

```
cal.setStartDate(new Date());
cal.setEndDate(new Date());
```

注意, 我们上面设置的日期范围长度其实为 0, 但日历仍然会显示从 00:00 到 23:59 的时间. 这是正常的, 因为 Vaadin Calendar 会保证至少显示你指定的日期范围, 但也有可能显示更大的范围. 当我们实现自定义的 Event Provider 时, 一定要注意日历的这种行为.

5.27.2. 日历中的事件

在日历内发生的一切事情都表达为 事件(Event). 你有三种方法管理日历内的事件:

- 使用 `addEvent()` 方法将事件直接加入到 **Calendar** 对象中
- 使用一个 Container 作为数据源
- 使用 *Event Provider* 机制

你可以使用 `addEvent()` 方法来添加事件, 使用 `removeEvent()` 方法删除. 这些方法会使用底层隐含的 Event Provider 来将变更写入到数据源中.

事件的接口和提供者

事件通过 `CalendarEvent` 接口来管理. 具体的 `Event` 类是什么, 由日历的 **CalendarEventProvider** 来决定.

默认情况下, **Calendar** 使用 **BasicEventProvider** 来提供事件, 这个 Provider 使用 **BasicEvent** 类.

日历并不依赖于任何特定的数据源实现类. 事件由 **Calendar** 向 Provider 请求取得, Provider 只需要实现 `CalendarEventProvider` 接口. 由 Event Provider 负责保证 **Calendar** 得到正确的事件.

图 5.61. 月单位显示模式，包含全日事件和通常事件

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1 Jul	2
27						
28	3	4	5	6	7	8
29	10	11	12	13	14	15
Whole week:						
	9:30 AM App	11:00 AM Tra	All day event	Second allday		9:00 AM Free
17	18	19	20	21	22	23
30	24	25	26	27	28	29
31	31	1 Aug	2	3	4	5
32						6

图 5.62. 周单位显示模式

Sun	Mon	Tue	Wed	Thu	Fri	Sat
7/10/11	7/11/11	7/12/11	7/13/11	7/14/11	7/15/11	7/16/11
Whole						
1 AM						
2 AM						
3 AM						
4 AM						
5 AM						
6 AM						
7 AM						
8 AM						
9 AM						
10 AM						
11 AM						
12 PM						
1 PM						
2 PM						
3 PM						
4 PM						
5 PM						
6 PM						
7 PM						
8 PM						
9 PM						
10 PM						
11 PM						

你可以将任意的 Vaadin **Container** 绑定到日历，这时会透明地使用 **ContainerEventProvider**. 容器内的数据必须按事件的开始日期/时刻排序. 关于容器，详情请参见 第 8.5 节 “在容器(Container)中保存项目(Item)”.

事件的类型

日历中的事件需要指定开始时刻和结束时刻. 这只是必须属性. 除此之外, 事件还可以设置 `all-day` 属性而成为全日事件(All-Day Event). 你还可以设置事件的 `Tooltip`, 这个信息会在 UI 中显示为提示信息(Tooltip).

如果事件的 `all-day` 属性为 `true`, 那么事件会永远显示为全日事件. 在月单位显示模式下, 全日事件在 UI 中不会显示开始时刻, 而且会有彩色的背景色. 在周单位显示模式下, 全日事件显示在画面的上部, 显示方式与月单位显示模式类似. 此外, 如果一个事件的时间范围为 24 小时或更长, 它在月单位显示模式下也会以全日事件的方式显示.

如果事件的时间范围(以毫秒精度计算)等于或小于 24 小时, 这个事件会被认为是通常的日事件. 通常事件有开始时刻和结束时刻, 这两个时刻可以属于不同的日.

基本事件

在日历中添加和管理事件的最简单方法是使用 `BasicEvent` 的管理 API. 日历默认使用 `BasicEventProvider`, 它将事件以某种内部形式保存在内存中.

比如, 下例添加一个 2 小时长的事件, 开始时刻为现在时刻. 标准的 Java `GregorianCalendar` 类提供了几种方式来操作日期和时间.

```
// Add a two-hour event
GregorianCalendar start = new GregorianCalendar();
GregorianCalendar end = new GregorianCalendar();
end.add(Calendar.HOUR, 2);
calendar.addEvent(new BasicEvent("Calendar study",
    "Learning how to use Vaadin Calendar",
    start.getTime(), end.getTime()));
```

这段代码添加一个持续 3 小时(译注: 看代码貌似应该是 2 小时)的事件. 由于 `BasicEventProvider` 和 `BasicEvent` 实现了日历包中的一些额外的事件接口, 因此我们不需要刷新日历. 只需要创建事件, 设置它的属性, 然后添加到 Event Provider 中即可.

5.27.3. 从容器得到事件

你可以使用任意的 Vaadin Container 作为日历事件的数据源, 但容器需要实现 `Indexed` 接口. `Calendar` 会监听容器中的事件变化, 也会将变化写回到容器. 你可以使用 `setContainerDataSource()` 方法将容器绑定到 `Calendar`.

下例中, 我们为日历绑定一个 `BeanItemContainer`, 容器中包含的是内建的 `BasicEvent` 事件类.

```
// Create the calendar
Calendar calendar = new Calendar("Bound Calendar");

// Use a container of built-in BasicEvents
final BeanItemContainer<BasicEvent> container =
    new BeanItemContainer<BasicEvent>(BasicEvent.class);

// Create a meeting in the container
container.addBean(new BasicEvent("The Event", "Single Event",
    new GregorianCalendar(2012,1,14,12,00).getTime(),
    new GregorianCalendar(2012,1,14,14,00).getTime()));

// The container must be ordered by the start time. You
// have to sort the BIC every time after you have added
// or modified events.
```

```

container.sort(new Object[]{"start"}, new boolean[]{true});

calendar.setContainerDataSource(container, "caption",
    "description", "start", "end", "styleName");

```

容器必须对事件的各种数据使用默认的属性 ID, 这些属性 ID 应该与 CalendarEvent 接口中的定义一致, 否则的话, 应该像上面的示例代码那样, 在 setContainerDataSource() 方法的参数中指定这些属性 ID.

保证容器内数据的顺序正确

容器内的事件 必须 按它们的开始日期/时刻排序. 如果排序不正确, 会导致事件在日历中无法正确显示.

排序是由容器决定的. 对于某些容器, 比如 **BeanItemContainer**, 每次你添加事件或修改事件之后都必须明确地排序, 通常是使用 sort() 方法, 像上面的示例代码那样. 某些容器, 比如 **JPACContainer**, 如果你指定了排序规则, 容器会自动排序.

比如, 假设事件的开始日期/时刻保存在 startDate 属性中, 你可以使用以下规则对 **JPACContainer** 排序:

```

// The container must be ordered by start date. For JPACContainer
// we can just set up sorting once and it will stay ordered.
container.sort(new String[]{"startDate"}, new boolean[]{true});

```

事件管理的委托

使用 setContainerDataSource() 方法将容器设置为日历的数据源, 会自动切换到 **ContainerEventProvider**. 你可以通过 **Calendar** 类中的 API 来操纵事件数据, 用户也可以通过 UI 来移动和拖动事件尺寸. Event Provider 负责将所有的这些日历操作代理到容器中.

如果你使用 **Calendar** API 来添加事件, 注意你有可能无法创建容器所要求的数据类型的事件对象, 对于那些需要特定操作的容器, 你也无法添加事件. 这种情况下, 你可能需要定制 addEvent() 方法.

比如, **JPACContainer** 要求使用 addEntity() 方法来创建新项目. 你可以先将项目直接添加到容器或其他的项目管理器中, 然后再将它传递给 addEvent() 方法. 但是, 如果项目类没有实现 CalendarEvent 接口的话, 这种方法是行不通的. 如果数据项目类的属性名与 CalendarEvent 接口的定义不一致的话, 就总会发生这个问题. 你可以在 addEvent() 方法内管理底层数据对象的创建过程, 如下例所示:

```

// Create a JPACContainer
final JPACContainer<MyCalendarEvent> container =
    JPACContainerFactory.make(MyCalendarEvent.class,
        "book-examples");

// Customize the event provider for adding events
// as entities
ContainerEventProvider cep =
    new ContainerEventProvider(container) {
        @Override
        public void addEvent(CalendarEvent event) {
            MyCalendarEvent entity = new MyCalendarEvent(
                event.getCaption(), event.getDescription(),
                event.getStart(), event.getEnd(),
                event.getStyleName());
            container.addEntity(entity);
        }
    };

```

```

        }

    // Set the container as the data source
    calendar.setEventProvider(cep);

    // Now we can add events to the database through the calendar
    BasicEvent event = new BasicEvent("The Event", "Single Event",
        new GregorianCalendar(2012,1,15,12,00).getTime(),
        new GregorianCalendar(2012,1,15,14,00).getTime());
    calendar.addEvent(event);

```

5.27.4. 实现 Event Provider

对于存储和管理日历中的事件, 如果上面介绍的两种简单方法不能满足你的需求的话, 你可能需要实现一个自定义的 Event Provider. 这是用来提供事件的最灵活方式. 你需要使用 `setEventProvider()` 方法将 Event Provider 关联到 **Calendar**.

日历会向 Event Provider 询问两个指定的日期之间的所有事件. 这两个日期的范围会保证至少与日历组件的开始日期到结束日期一样长. 但是, 日历组件也可以查询更长的范围, 以便保证自己的正常显示. 具体来说, 开始日期会被扩展到一日的最初时刻, 结束日期会被扩展到一日的最终时刻.

自定义事件

Event Provider 可以使用内建的 **BasicEvent** 类型, 但通常创建直接与数据源绑定的自定义事件类型会更适当一些. 自定义事件对于其他某些目的也很便利, 比如, 如果你需要在事件中增加额外的信息, 或者希望自定义它的取得方式.

自定义事件必须实现 `CalendarEvent` 接口, 或继承自某个已存在的事件类. 关于如何实现简单的事件类, 内建的 **BasicEvent** 类是很好的例子. 这个类将相关数据都保存在成员变量中.

```

public class BasicEvent
    implements CalendarEventEditor, EventChangeNotifier {
    ...

    public String getCaption() {
        return caption;
    }

    public String getDescription() {
        return description;
    }

    public Date getEnd() {
        return end;
    }

    public Date getStart() {
        return start;
    }

    public String getStyleName() {
        return styleName;
    }

    public boolean isAllDay() {
        return isAllDay;
    }
}

```

```

public void setCaption(String caption) {
    this.caption = caption;
    fireEventChange();
}

public void setDescription(String description) {
    this.description = description;
    fireEventChange();
}

public void setEnd(Date end) {
    this.end = end;
    fireEventChange();
}

public void setStart(Date start) {
    this.start = start;
    fireEventChange();
}

public void setStyleName(String styleName) {
    this.styleName = styleName;
    fireEventChange();
}

public void setAllDay(boolean isAllDay) {
    this.isAllDay = isAllDay;
    fireEventChange();
}

public void addEventChangeListener(
    EventChangeListener listener) {
    ...
}

public void removeListener(EventChangeListener listener) {
    ...
}

protected void fireEventChange() {...}
}

```

你可能注意到了，在 **BasicEvent** 类中还有一些与 `CalendarEvent` 接口无关的代码。也就是说，**BasicEvent** 还实现了下面两个接口：

`CalendarEditor`

这个接口为所有 Field 定义了 Setter 方法，对于日历中的某些默认的处理是必要的。

`EventChangeListener`

这个接口可以监听事件的变更，**Calendar** 因此可以立即将这些变更显示到画面上。

开始时刻和结束时刻是必须的属性，但标题、描述信息、样式名不是必须的。样式名会被用作事件的 HTML DOM 元素的 CSS 类名的一部分。

除基本的事件接口外，你还可以使用 **EventChange** 和 **EventSetChange** 事件来进一步扩充你的事件和 Event Provider 的功能。这些事件使得 **Calendar** 组件知道事件的变更，并对它自身做相应的更新。前面给出的 **BasicEvent** 和 **BasicEventProvider** 例子包含了这些接口的简单实现。

实现 Event Provider

Event Provider 需要实现 `CalendarEventProvider` 接口. 这个接口只有一个方法需要实现. 当日历显示在画面上时, `getEvents(Date, Date)` 方法会被调用, 它必须返回指定的开始日期到结束日期之间的所有事件的列表.

下例中的实现只返回一个示例事件. 这个事件开始时刻是现在时刻, 并持续 5 小时长.

```
public class MyEventProvider implements CalendarEventProvider{
    public List<Event> getEvents(Date startDate, Date endDate) {
        List<Event> events = new ArrayList<Event>();
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(new Date());

        Date start = cal.getTime();
        cal.add(GregorianCalendar.HOUR, 5);
        Date end = cal.getTime();
        BasicEvent event = new BasicEvent();
        event.setCaption("My Event");
        event.setDescription("My Event Description");
        event.setStart(start);
        event.setEnd(end);
        events.add(event);

        return events;
    }
}
```

要注意, **Calendar** 向 Event Provider 查询的日期范围, 可能会比日历的开始日期到结束日期范围更大一些. 具体来说, 日历可能会扩张查询日期范围, 以便保证日历的 UI 能够正确显示.

5.27.5. 控制日历的样式

控制 Vaadin Calendar 组件的画面表现是一个很基本的任务. 你至少需要考虑它在你的 UI 中的尺寸. 你很可能还会希望为事件使用某些特别的颜色.

尺寸

Calendar 组件允许使用指定的尺寸(固定尺寸或相对尺寸都可以). 如果对日历使用未定义的尺寸, 所有的尺寸都由 CSS 来决定. 此外, 如果高度未定义, 会在周单位显示模式中出现一个滚动条, 以便在 UI 中更好地调控单元格.

对于未定义尺寸的日历, 用于定义它的各种尺寸的样式规则如下(这些都是默认值):

```
.v-calendar-month-sizedheight .v-calendar-month-day {
    height: 100px;
}

.v-calendar-month-sizedwidth .v-calendar-month-day {
    width: 100px;
}

.v-calendar-header-month-Hsized .v-calendar-header-day {
    width: 101px;
}

/* for IE */
.v-ie6 .v-calendar-header-month-Hsized .v-calendar-header-day {
```

```

        width: 104px;
    }

/* for others */
.v-calendar-header-month-Hsized td:first-child {
    padding-left: 21px;
}

.v-calendar-header-day-Hsized {
    width: 200px;
}

.v-calendar-week-numbers-Vsized .v-calendar-week-number {
    height: 100px;
    line-height: 100px;
}

.v-calendar-week-wrapper-Vsized {
    height: 400px;
    overflow-x: hidden !important;
}

.v-calendar-times-Vsized .v-calendar-time {
    height: 38px;
}

.v-calendar-times-Hsized .v-calendar-time {
    width: 42px;
}

.v-calendar-day-times-Vsized .v-slot,.v-calendar-day-times-Vsized .v-slot-even {
    height: 18px;
}

.v-calendar-day-times-Hsized, .v-calendar-day-times-Hsized .v-slot,.v-calendar-
day-times-Hsized .v-slot-even {
    width: 200px;
}

```

事件的样式

可以对事件设置一个样式名称后缀来使用 CSS 控制事件的样式。这个后缀会通过 `CalendarEvent` 接口的 `getStyleName()` 方法来取得。如果你使用 `BasicEvent` 事件，你可以通过 `setStyleName()` 方法来设置这个后缀。

```

BasicEvent event = new BasicEvent("Wednesday Wonder", ... );
event.setStyleName("mycolor");
calendar.addEvent(event);

```

后缀 `mycolor` 会为通常事件创建 `v-calendar-event-mycolor` 样式，为全日事件创建 `v-calendar-event-mycolor-all-day` 样式。你可以使用以下规则来控制事件样式：

```

.v-calendar .v-calendar-event-mycolor {}
.v-calendar .v-calendar-event-mycolor-all-day {}
.v-calendar .v-calendar-event-mycolor .v-calendar-event-caption {}
.v-calendar .v-calendar-event-mycolor .v-calendar-event-content {}

```

5.27.6. 可见的小时和日

在第 5.27.1 节“日期范围与显示模式”中我们看到，你可以设置日历显示的日期范围。但是如果你希望显示整个月，但又隐藏周末，该如何实现呢？或者如果希望在周单位显示模式下只显示 8 时到 16 时，该如何实现？`setVisibleDays()` 和 `setVisibleHours()` 方法允许你进行这类设置。

```
calendar.setVisibleDays(1,5); // Monday to Friday
calendar.setVisibleHours(0,15); // Midnight until 4 pm
```

进行上面的设定后，只有星期一到星期五会被显示。而且，当日历处于周单位显示模式时，只有从 00:00 到 16:00 时刻范围会被显示。

注意，被排除在外的时间范围永远不会显示，因此你在设置日期范围时需要小心。如果日历的日期范围只包含被排除的日期/时间，那么日历中就不会显示任何东西了。还要注意，即使上面的设置导致一部分事件不在画面上显示，但对于这些显示范围外的日期，Event Provider 还是会被查询其中的事件。

5.27.7. 拖放

Vaadin Calendar 可以在拖放操作中作为 Drop 动作的目标，详情请参见第 11.12 节“拖放”。使用这个功能，用户可以拖动事件，比如，从一个 Table 组件拖动到一个日历组件。

要支持 Drop 动作，**Calendar** 需要有一个 Drop 处理器。设置好 Drop 处理器之后，月单位显示模式中的日，以及周单位显示模式中的时间条，可以接收 Drop 动作。日历的其他位置，比如周单位显示模式中的日名称，目前还不能接收 Drop 动作。

日历使用它自己实现的 TargetDetails：**CalendarTargetdetails**。它会管理 Drop 位置信息，对 **Calendar** 来说也就意味着对应的日期和时间。Drop 动作的位置可以通过 `getDropTime()` 方法取得。如果 Drop 动作发生在月单位显示模式中，那么这个方法返回的日期值中不会包含具体的时间信息。如果 Drop 动作发生在周单位显示模式中，返回的日期值会包含对应的时刻条的开始时间。

下面是一个简单的例子，演示如何编写一个 Drop 处理器，并使用 Drop 信息来创建新的事件：

```
private Calendar createDDCalendar() {
    Calendar calendar = new Calendar();
    calendar.setDropHandler(new DropHandler() {
        public void drop(DragAndDropEvent event) {
            CalendarTargetDetails details =
                (CalendarTargetDetails) event.getTargetDetails();

            TableTransferable transferable =
                (TableTransferable) event.getTransferable();

            createEvent(details, transferable);
            removeTableRow(transferable);
        }

        public AcceptCriterion getAcceptCriterion() {
            return AcceptAll.get();
        }
    });

    return calendar;
}
```

```

protected void createEvent(CalendarTargetDetails details,
    TableTransferable transferable) {
    Date dropTime = details.getDropTime();
    java.util.Calendar timeCalendar = details.getTargetCalendar()
        .getInternalCalendar();
    timeCalendar.setTime(dropTime);
    timeCalendar.add(java.util.Calendar.MINUTE, 120);
    Date endTime = timeCalendar.getTime();

    Item draggedItem = transferable.getSourceComponent().
        getItem(transferable.getItemId());

    String eventType = (String)draggedItem.
        getItemProperty("type").getValue();

    String eventDescription = "Attending: "
        + getParticipantString(
            (String[]) draggedItem.
                getItemProperty("participants").getValue());
}

BasicEvent newEvent = new BasicEvent();
newEvent.setAllDay(!details.hasDropTime());
newEvent.setCaption(eventType);
newEvent.setDescription(eventDescription);
newEvent.setStart(dropTime);
newEvent.setEnd(endTime);

BasicEventProvider ep = (BasicEventProvider) details
    .getTargetCalendar().getEventProvider();
ep.addEvent(newEvent);
}
}

```

5.27.8. 使用上下文菜单

Vaadin Calendar 允许使用上下文菜单(点击鼠标右键)来管理事件. 在 Vaadin 的其他上下文菜单中, 菜单项是作为 Vaadin 的 动作(Action), 由一个 动作处理器(Action Handler) 负责处理的. 要启用上下文菜单, 你必须实现 Vaadin 的 Action.Handler 接口, 然后使用 addActionHandler() 方法将它添加到日历中.

动作处理器必须实现两个方法: getActions() 和 handleAction(). 对日历中显示的每一日都会调用 getActions() 方法. 这个方法应该返回对于这一天所允许执行的所有动作的一个列表, 这个列表就会成为上下文菜单中的菜单项. target 参数代表点击的上下文环境 - 这里是横跨这一天的一个 **CalendarDateRange**. sender 参数是 **Calendar** 对象.

handleActions() 方法通过 target 参数接受目标上下文. 如果上下文菜单在一个事件上打开, 目标上下文将是 Event 对象, 否则它将是 **CalendarDateRange** 对象.

5.27.9. 本地化与格式化

设置语言环境(Locale)和时区(Time Zone)

月和星期的名称使用 **Calendar** 的语言环境所规定的语言来显示. 翻译需要使用标准的 Java 语言环境数据来进行. 默认情况下, **Calendar** 使用系统默认的语言环境来处理内部的日期时刻, 但你可以使用 setLocale(Locale locale) 方法来改变语言环境. 设置语言环境还会更新于语言环境相关的其他日期和时间设定, 比如周的起始日, 时区, 以及时间的显示格式. 但是, 时区和时间格式可以在 **Calendar** 中单独设置.

比如, 下例会将语言设置为美国英语:

```
cal.setLocale(Locale.US);
```

语言环境定义了默认的时区. 你可以使用 `setTimeZone()` 方法来改变时区设定, 这个方法接受的参数是 **java.util.TimeZone** 对象. 将时区设置为 null 会将它重置为当前语言环境的默认时区.

比如, 下例将时区设置为芬兰时区, 也就是 EET (欧洲东部时间)

```
cal.setTimeZone(TimeZone.getTimeZone("Europe/Helsinki"));
```

时间与日期标题格式

时间可以显示为 24 小时或 12 小时格式. 默认格式由语言环境决定, 但你可以使用 `setTimeFormat()` 方法来设置格式. 格式设置为 null 会将时间格式重置为语言环境的默认格式.

```
cal.setTimeFormat(DateFormat.Format12H);
```

你可以使用 `setWeeklyCaptionFormat(String dateFormatPattern)` 方法来修改周单位显示模式中的日期标题格式. 这个方法接受的日期格式字符串需要遵守标准的 Java **java.text.SimpleDateFormat** 类中的格式规则.

示例

```
cal.setWeeklyCaptionFormat("dd-MM-yyyy");
```

5.27.10. 定制日历

在本节中, 我们给出一段示例, 演示如何对 Vaadin 日历进行一些基本的定制. Event Provider 和样式控制在前文已经介绍过了, 所以现在我们集中讨论日历 API 的其他功能.

事件处理器概览

与日历事件相关的大多数处理器都已经有了不错的默认处理器. 这些默认处理器可以在 com.vaadin.ui.handler 包中找到. 默认处理器及其功能介绍如下.

- **BasicBackwardHandler**. 周单位显示模式下, 处理回退按钮的点击事件, 此时日历中显示的时间范围会被切换为上一月(译注: 原文如此, 应为"周").
- **BasicForwardHandler**. 周单位显示模式下, 处理前进按钮的点击事件, 此时日历中显示的时间范围会被切换为下一月(译注: 原文如此, 应为"周").
- **BasicWeekClickHandler**. 月单位显示模式下, 处理周数字的点击事件, 此时日历的显示范围会切换到被点击的那一周.
- **BasicDateClickHandler**. 在月单位和周单位两种显示模式中, 处理日期的点击事件. 此时日历中的显示范围会变为只显示被点击的那一日.
- **BasicEventMoveHandler**. 在月单位和周单位两种显示模式中, 处理事件的移动. 事件可以通过 UI 操作来移动, 它的开始日期结束日期会被正确地更新, 但这一点只在事件实现了 **CalendarEventEditor** 接口时才有效 (这个接口由 **BasicEvent** 类实现).
- **BasicEventResizeHandler**. 周单位显示模式下, 处理事件被鼠标拖动改变大小. Event 的大小可以通过鼠标拖动来改变, 它的开始日期结束日期会被正确地更新, 但这一点只在事件实现了 **CalendarEventEditor** 接口时才有效 (这个接口由 **BasicEvent** 类实现).

创建新的 **Calendar** 时，会自动设置上述所有的处理器。如果你希望关闭某种默认功能，你可以设定对应的事件处理器为 `null`。这样就可以阻止相关的功能，被阻止的功能不会出现在 UI 中。比如，如果你将 **EventMoveHandler** 设置为 `null`，用户将不能在浏览器端移动日历中的事件。

创建日历

我们首先创建一个新的 **Calendar** 实例。这里我们使用自己的 Event Provider, **MyEventProvider**, 详情请参见“实现 Event Provider”一节。

```
Calendar cal = new Calendar(new MyEventProvider());
```

以上代码初始化了一个日历。为了定制日历的可见日期范围，我们必须设置它的开始日期和结束日期。

在时间线中只有一个可见的事件，这个事件的开始时间为当前时刻。这就是我们的Event Provider 向外提供的事件。

如果能控制日历中显示日期的前后跳转就好了。默认的跳转行为是由默认的处理器实现的，但是我们也许会希望限制用户，使他们只能在当前年份中跳转。我们可能还希望对星期数和日期的点击做出一些别的限制。

这些限制，以及其他自定义逻辑可以由自定义的处理器实现。你可以在 `com.vaadin.addon.calendar.ui.handler` 包中找到这些默认的处理器，也可以很容易地继承它们。注意，如果你不希望继承默认的处理器，你也完全可以自行实现。相关的接口请参见 `CalendarComponentEvents`。

5.27.11. 日期的前方和后方跳转

Vaadin 日历的日期跳转功能只有有限的内建支持。周单位显示模式的左上角和右上角有日期跳转的按钮。

你可以使用 `BackwardListener` 和 `ForwardListener` 监听器，来处理日期的前方和后方跳转事件。

```
cal.setHandler(new BasicBackwardHandler() {
    protected void setDates(BackwardEvent event,
                           Date start, Date end) {

        java.util.Calendar calendar = event.getComponent()
            .getInternalCalendar();
        if (isThisYear(calendar, end)
            && isThisYear(calendar, start)) {
            super.setDates(event, start, end);
        }
    });
});
```

向前跳转的处理器可以用与上例相同的方式来实现。上例中的处理器限制只能在当前年份内跳转。

5.27.12. 处理日期的点击

默认情况下，在月单位或周单位显示模式下点击一个日期，会切换到单日显示模式。日期的点击事件由 `DateClickHandler` 来处理。

下例演示如何处理点击事件，在周单位显示模式下点击日期，会切换到单日显示模式，在单日显示模式下再次点击日期，会切换回到周单位显示模式。

```

cal.setHandler(new BasicDateClickHandler() {
    public void dateClick(DateClickEvent event) {
        Calendar cal = event.getComponent();
        long currentCalDateRange = cal.getEndDate().getTime()
            - cal.getStartDate().getTime();

        if (currentCalDateRange < VCalendar.DAYINMILLIS) {
            // Change the date range to the current week
            cal.setStartDate(cal.getFirstDateForWeek(event.getDate()));
            cal.setEndDate(cal.getLastDateForWeek(event.getDate()));

        } else {
            // Default behaviour, change date range to one day
            super.dateClick(event);
        }
    }
});

```

5.27.13. 处理周的点击

月单位显示模式会为每一个周的行，在日期表格的左侧显示周数字。周数字是可以点击的，你可以为 **Calendar** 对象设置一个 **WeekClickHandler** 来处理这个点击事件。默认的处理器会将日历显示的日期范围切换为被点击的那一周。

下例中，我们添加一个周点击事件处理器，它会将日历的日期范围切换到被点击的周，但要求这一周的开始日和结束日都属于当前月份。

```

cal.setHandler(new BasicWeekClickHandler() {
    protected void setDates(WeekClick event,
                           Date start, Date end) {
        java.util.Calendar calendar = event.getComponent()
            .getInternalCalendar();
        if (isThisMonth(calendar, start)
            && isThisMonth(calendar, end)) {
            super.setDates(event, start, end);
        }
    }
});

```

5.27.14. 处理事件的点击

在任何一种显示模式下，日历内的事件都是可以点击的。事件的点击没有默认的处理器。与日期和周的点击处理器一样，设置事件点击处理器的方法是，为 **Calendar** 对象设置一个 **EventClickHandler**。

在传递给处理器的 **EventClick** 对象参数中，使用 **getCalendarEvent()** 方法，可以得到被点击的事件，如下例所示。

```

cal.setHandler(new EventClickHandler() {
    public void eventClick(EventClick event) {
        BasicEvent e = (BasicEvent) event.getCalendarEvent();

        // Do something with it
        new Notification("Event clicked: " + e.getCaption(),
                        e.getDescription()).show(Page.getCurrent());
    }
});

```

5.27.15. 事件的拖动

用户可以在 UI 中拖动一个事件，改变事件在时间线上的位置。默认处理器会相应的设置事件的开始时间和结束时间。你可以使用自定义的处理器来做更多的事情，比如，限制事件的移动范围。

下例中，我们为 **Calendar** 添加了一个 `EventMoveHandler`。这个事件处理器会将新的位置更新到数据源中，但要求新的日期必须属于当前月份。为完成这个功能，需要对 `Event Provider` 类做一些修改。

```
cal.setHandler(new BasicEventMoveHandler() {
    private java.util.Calendar javaCalendar;

    public void eventMove(MoveEvent event) {
        javaCalendar = event.getComponent().getInternalCalendar();
        super.eventMove(event);
    }

    protected void setDates(CalendarEventEditor event,
                           Date start, Date end) {
        if (isThisMonth(javaCalendar, start)
            && isThisMonth(javaCalendar, end)) {
            super.setDates(event, start, end);
        }
    }
});
```

为了让上面的示例代码工作起来，前面我们讨论过的 `Event Provider` 例子需要略作修改，让它在 `getEvents()` 被调用时，不要永远创建新的事件。

```
public static class MyEventProvider
    implements CalendarEventProvider {
    private List<CalendarEvent> events =
        new ArrayList<CalendarEvent>();

    public MyEventProvider() {
        events = new ArrayList<CalendarEvent>();
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(new Date());

        Date start = cal.getTime();
        cal.add(GregorianCalendar.HOUR, 5);
        Date end = cal.getTime();
        BasicEvent event = new BasicEvent();
        event.setCaption("My Event");
        event.setDescription("My Event Description");
        event.setStart(start);
        event.setEnd(end);
        events.add(event);
    }

    public void addEvent(CalendarEvent BasicEvent) {
        events.add(BasicEvent);
    }

    public List<CalendarEvent> getEvents(Date startDate,
                                         Date endDate) {
        return events;
    }
}
```

经过上面的修改后，用户就可以移动事件了，但拖放一个事件时，事件的开始日期和结束日期会被服务器端检查。注意，服务器端必须将事件移动到适当的位置，这样才能将事件显示在它被拖放的位置。服务器端也可以拒绝移动事件，方法是收到移动事件后不作任何处理。

5.27.16. 处理拖动式选择(Drag Selection)

在月单位和周单位两种显示模式下，拖动式选择(Drag Selection)都是有效的。为了监听拖动式选择事件，你可以为 **Calendar** 添加一个 RangeSelectListener 监听器。对于范围选择，没有默认的处理器。

下例中，当用户选中任何一个日期范围时，我们创建一个新的事件。拖动选择时会打开一个新的窗口，要求用户在这个窗口中为新的事件输入一个标题。输入完成后，新事件会被传递给Event Provider，并且日历的显示会被更新。注意，我们的示例 Event Provider 和 Event 类没有实现事件变更的接口，因此我们变更事件之后必须手动刷新 **Calendar**。

```
cal.setHandler(new RangeSelectHandler() {
    public void rangeSelect(RangeSelectEvent event) {
        BasicEvent calendarEvent = new BasicEvent();
        calendarEvent.setStart(event.getStart());
        calendarEvent.setEnd(event.getEnd());

        // Create popup window and add a form in it.
        VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        layout.setSpacing(true);

        final Window w = new Window(null, layout);
        ...

        // Wrap the calendar event to a BeanItem
        // and pass it to the form
        final BeanItem<CalendarEvent> item =
            new BeanItem<CalendarEvent>(myEvent);

        final Form form = new Form();
        form.setItemDataSource(item);
        ...

        layout.addComponent(form);

        HorizontalLayout buttons = new HorizontalLayout();
        buttons.setSpacing(true);
        buttons.addComponent(new Button("OK", new ClickListener() {

            public void buttonClick(ClickEvent event) {
                form.commit();

                // Update event provider's data source
                provider.addEvent(item.getBean());

                UI.getCurrent().removeWindow(w);
            }
        }));
        ...
    }
});
```

5.27.17. 变更事件的长度

用户可以在 UI 中拖动事件的左侧或右侧来改变事件的长度，同时也改变事件的开始事件或结束时间。这个功能提供了一种改变事件时间的便利方式，不必进行任何键盘输入。事件长度变更的默认处理器会根据长度的变化对应的设置事件的开始时间和结束时间。

下例中，我们对事件的长度变化设置一个自定义的处理器。这个处理器禁止事件长度超过 12 小时。注意，这个限制不会阻止用户在客户端将事件拉到 12 小时以上。事件长度的变更只会在服务器端被纠正。

```
cal.setHandler(new BasicEventResizeHandler() {
    private static final long twelveHoursInMs = 12*60*60*1000;

    protected void setDates(CalendarEventEditor event,
                           Date start, Date end) {
        long eventLength = end.getTime() - start.getTime();
        if (eventLength <= twelveHoursInMs) {
            super.setDates(event, start, end);
        }
    }
});
```

5.28. 使用 **CustomComponent** 创造复合组件

能够简便地创建新 UI 组件是 Vaadin 的核心功能之一。通常你只需要简单地将已有的内建组合起来，就可以创造出复合组件。在很多应用程序中，大多数 UI 是由这样的复合组件构成的。

在第 4.2.2 节“组合组件”中我们曾介绍过，有两种基本的方法来创建复合组件 - 可以使用布局组件，或者使用 **CustomComponent** 组件。**CustomComponent** 之内通常也会封装一个布局组件。将布局组件封装在 **CustomComponent** 之内的好处主要在于信息的封装性 - 能够隐藏复合组件的内部实现细节。否则，复合组件的使用者编写的代码，可能会依赖于其内部的实现细节，比如具体的布局组件类。

要创建一个复合组件，你需要继承 **CustomComponent** 类，并在构造方法中设置复合组件的根组件。复合组件的根组件通常是一个布局管理组件，并在其中包含其他组件。

示例

```
class MyComposite extends CustomComponent {
    public MyComposite(String message) {
        // A layout structure used for composition
        Panel panel = new Panel("My Custom Component");
        panel.setContent(new VerticalLayout());

        // Compose from multiple components
        Label label = new Label(message);
        label.setSizeUndefined(); // Shrink
        panel.addComponent(label);
        panel.addComponent(new Button("Ok"));

        // Set the size as undefined at all levels
        panel.getContent().setSizeUndefined();
        panel.setSizeUndefined();
        setSizeUndefined();

        // The composition root MUST be set
        setCompositionRoot(panel);
    }
}
```

```

    }
}

```

如果想要让自定义组件的大小收缩到与它内含的子组件相适应，需要注意尺寸问题。你需要在组件层级关系的每一层都将尺寸设置为未定义；复合组件的尺寸与根组件的尺寸是不同的。

你可以象下例这样使用这个自定义组件：

```
MyComposite mycomposite = new MyComposite("Hello");
```

自定义组件的画面表现，见图 5.63 “自定义的复合组件”。

图 5.63. 自定义的复合组件



你也可以继承其他组件，比如布局管理组件，来实现类似的组件组合工作。甚至，你还可以创建全新的低阶组件，方法是集成纯客户端组件，或扩展内建组件的客户端功能。开发新组件的方法，请参见第 16 章与客户端集成。

5.29. 使用 **CustomField** 组合 Field 组件

CustomField 与 **CustomComponent** 类似，也是一种创建复合组件的方式，但是他实现了 **Field** 方法，并继承了 **AbstractField** 类，参见第 5.4 节“Field 组件”。**Field** 组件可以编辑 Vaadin 数据模型中的属性值，也可以使用 **Field Group** 绑定到数据上，参见第 8.4 节“通过 Field 与项目的绑定来创建 Form”。**Field** 值会被缓存起来，可以使用校验器来校验。

复合的 **Field** 类必须实现 **getType()** 和 **initContent()** 方法。后一个方法应该返回 **Field** 中的内容的组合，通常是一个布局管理组件，但也可以是任何别的组件。

也可以覆盖 **validate()**, **setInternalValue()**, **commit()**, **setPropertyDataSource()**, **isEmpty()** 和其他方法，以便在 **Field** 中实现各种不同功能。覆盖 **setInternalValue()** 方法时，应该调用基类中的方法。

5.30. 内嵌资源

在 Vaadin UI 中，你可以使用 **Image** 组件嵌入图像，使用 **Flash** 嵌入 Adobe Flash 动画，还可以使用 **BrowserFrame** 来嵌入其他的 Web 内容。还有通用的 **Embedded** 组件可以嵌入其他类型的对象。嵌入内容通过 **resources** 来引用，详情请参见第 4.4 节“图片及其他资源”。

下例演示如何使用类记载器装载一个类资源，并将它显示为图片：

```
Image image = new Image("Yes, logo:",
    new ClassResource("vaadin-logo.png"));
main.addComponent(image);
```

将标题设置为 **null** 可以删除标题。如果将标题设置为空字符串的话，会显示一个空的标题，仍然会占据一点点空间。标题由组件所属的布局负责管理。

你可以使用 **setAlternateText()** 方法为内嵌资源设置一个替代文字，如果出于某种原因，图片被浏览器禁用，那么就会显示这个替代文字。这个文字还可用来提高界面对残障人士的易用性，比如，可以使用自动语音合成技术来朗读它。

5.30.1. 内嵌的 Image

Image 组件可用来在 Vaadin UI 中嵌入一个图像资源.

```
// Serve the image from the theme
Resource res = new ThemeResource("img/myimage.png");

// Display the image without caption
Image image = new Image(null, res);
layout.addComponent(image);
```

Image 组件在长宽方向上默认都为未定义尺寸, 因此它会自动适应图片的尺寸. 如果你希望使用滚动条来滚动, 你可以将 Image 放置在 **Panel** 之内, 为 Panel 设置固定的尺寸, 并允许滚动, 详情请参见第 6.6.1 节 “滚动 Panel 的内容”. 你也可以将 Image 放在其他组件容器内, 并在 Theme 中为容器的 HTML 元素设置 CSS 属性为: overflow: auto, 这样就可以自动出现滚动条.

图片的生成和重新装载

你也可以使用 **StreamResource** 来动态生成图片内容, 详情请参见第 4.4.5 节 “流资源”, 也可以使用 **RequestHandler**.

如果图片内容发生了变化, 浏览器需要重新装载它. 简单的更新流资源是不够的. 由于某些浏览器端对缓存的管理方式, 让图片重新装载的最简便方法是用一个唯一文件名来重命名资源文件名, 比如, 使用时间戳作为文件名的一部分. 创建资源时, 你应该使用 `setCacheTime()` 方法将资源的缓存时间设置为 0.

```
// Create the stream resource with some initial filename
StreamResource imageResource =
    new StreamResource(imageSource, "initial-filename.png");

// Instruct browser not to cache the image
imageResource.setCacheTime(0);

// Display the image
Image image = new Image(null, imageResource);
```

刷新时, 你还需要对 **Image** 对象调用 `markAsDirty()` 方法.

```
// This needs to be done, but is not sufficient
image.markAsDirty();

// Generate a filename with a timestamp
SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmssSSS");
String filename = "myfilename-" + df.format(new Date()) + ".png";

// Replace the filename in the resource
imageResource.setFilename(makeImageFilename());
```

5.30.2. Adobe Flash 动画

Flash 组件可用来在 Vaadin UI 中嵌入 Adobe Flash 动画.

```
Flash flash = new Flash(null,
    new ThemeResource("img/vaadin_spin.swf"));
layout.addComponent(flash);
```

你可以使用 `setParameter()` 方法设置 Flash 参数，这个方法接受的参数是 Flash 参数名和参数值，类型都是字符串。你还可以为 HTML 中的 Flash 对象元素设置 `codeBase`, `archive`, 以及 `standBy` 属性。

5.30.3. BrowserFrame

BrowserFrame 可用于在 HTML `<iframe>` 元素内嵌入 Web 内容。你可以使用 **ExternalResource** 来引用一个外部 URL。

由于 **BrowserFrame** 默认未定义尺寸，因此你必须为它指定一个有意义的尺寸，固定尺寸或相对尺寸都可以。

```
BrowserFrame browser = new BrowserFrame("Browser",
    new ExternalResource("http://demo.vaadin.com/sampler/"));
browser.setWidth("600px");
browser.setHeight("400px");
layout.addComponent(browser);
```

注意，Web 页面本身有可能会阻止自己嵌入到 `<iframe>` 中。

5.30.4. 通用的 Embedded 对象

除了使用特定的组件来嵌入图像、Flash 动画、浏览器 Frame 之外，还可以使用通用的 **Embedded** 组件来嵌入任何类型的对象，比如 SVG 图像、Java Applet，以及 PDF 文档。

比如，下例显示一个 Flash 动画：

```
// A resource reference to some object
Resource res = new ThemeResource("img/vaadin_spin.swf");

// Display the object
Embedded object = new Embedded("My Object", res);
layout.addComponent(object);
```

下例显示 SVG 图片

```
// A resource reference to some object
Resource res = new ThemeResource("img/reindeer.svg");

// Display the object
Embedded object = new Embedded("My SVG", res);
object.setMimeType("image/svg+xml"); // Unnecessary
layout.addComponent(object);
```

对象的 MIME 类型通常由 Vaadin 中的工具类 **FileTypeResolver** 根据文件扩展名来自动判定。如果无法自动判定，你可以使用 `setMimeType()` 方法来明确的设定它，如上例所示(在上例中其实是不必明确设定的)。

某些内嵌对象类型可能需要浏览器端的特殊支持。如果浏览器不支持这种内嵌类型，你需要确定存在一个适当的 fallback 机制来处理这种异常情况。

6.1. 概述	208
6.2. UI, Window, 以及 Panel 内容	209
6.3. VerticalLayout 和 HorizontalLayout	210
6.4. GridLayout	215
6.5. FormLayout	218
6.6. Panel	220
6.7. 子窗口	222
6.8. HorizontalSplitPanel 和 VerticalSplitPanel	225
6.9. TabSheet	227
6.10. Accordion	231
6.11. AbsoluteLayout	232
6.12. CssLayout	235
6.13. 布局格式控制	237
6.14. 自定义布局	242

自从远古时代施乐公司发明图形用户界面(GUI, Graphical User Interface)以来, 程序员们总是希望让 GUI 程序的开发变得更简单一些。最初的解决方案很简单。当 GUI 最初出现在 PC 桌面时, 所有的显示器其实都是 VGA 类型, 而且分辨率是固定的 640x480。Mac 或 UNIX 上的 X Window 系统实际上区别不大。这个图像分辨率很棒, 大家都很开心, 从未想过应用程序还需要在一个完全不同的屏幕尺寸下运行。他们认为最坏的情况下, 屏幕也只会变得更大, 出现更大的空间可以排放更多的窗口。在 80 年代, 在你的口袋里放一个计算机显示器纯粹是不现实的想法。因此, 这个时代的 GUI API 允许使用屏幕坐标来放置 UI 组件。Visual Basic 和其他开发系统, 为设计者提供了很简单的方式, 可以在固定尺寸的窗口中拖放组件。有些人可能会认为, 至少翻译人员会抱怨这种方案太笨拙, 但身为非工程技术人员, 实际上根本没有人听他们的意见, 或者至少根本没人在意他们的意见。最好的情况下, 开发者可以给他们一个资源编辑器, 让他们手动调整 UI 组件的尺寸。这就是当时关于 GUI 开发的基本理念。

Web 诞生之后，布局设计注定要永远改变了。起初，布局并不怎么重要，大家都满足于平文的标题，段落，以及这里那里出现的少量超链接。HTML 设计者希望页面可以在任意的屏幕尺寸上运行。这时的屏幕尺寸实际上不是像素单位，而是字符组成的行和列，因为婴儿期的 Web 确实只是超文本而已，而不是图像。但这种情况很快就变了。第一个基于 GUI 的浏览器，NCSA Mosaic，发起了一场革命，由 Netscape Navigator 将这场革命推向高潮。一夜之间，原先负责做广告招贴画的那些人们现在开始写 HTML 了。这就意味着，布局的设计必须是简单的，不仅是对程序员如此，还要让美工设计人员不必具备任何编程知识也能完成他/她的工作。W3C 委员会设计的 Web 标准提出了 CSS (Cascading Style Sheet, 层叠样式表) 规范，这个规范使得表现可以与内容分离，但分离的程度不高。HTML 延续了几个新版本，后来又出现了 XHTML 和 HTML 5，以及不计其数的其他标准。

页面描述和标记语言对于静态内容来说是非常不错的解决方案，比如书和大多数 Web 页面。但是真正的应用程序则需要更多的控制。它们需要能够在运行中改变 UI 组件的状态，甚至改变组件的布局。这个就导致了一个需求，应该在正确的层次上，将表现与内容分离开。

感谢美工设计人员的攻击，桌面应用程序，从它出现的时候开始，远远落后于 Web 设计。Sun Microsystems 公司于 1995 年发布了一个新的编程语言，Java，用来编写跨平台的桌面应用程序。Java 最初的 GUI 开发工具，AWT (Abstract Windowing Toolkit)，被设计为可运行在多种操作系统上，而且还能嵌入到 Web 浏览器中。AWT 中很特别的一部分就是布局管理器，它使得 UI 组件可以按照需要灵活地扩大或缩小。这就使得用户可以灵活地拉伸应用程序的窗口大小，而且也解决了本地化的需要，因为文本字符串的尺寸不再被限定为固定像素单位了。甚至还可以调整字体的尺寸，布局的其他部分会自动适应新的字体大小。

Vaadin 中的布局管理继承了 Web 中内容与表现分离的理念，也继承了 Java AWT 的解决方案，使用程序将布局与 UI 组件绑定起来。Vaadin 布局管理组件允许你将 UI 组件使用层级方式布置到屏幕上，类似于传统的 Java UI 开发工具，比如 AWT, Swing, 或 SWT。此外，你还可以通过 Web 方式来控制布局，使用 **CustomLayout** 组件你可以用 HTML 模板的方式编写布局，这个模板决定布局中包含的所有组件的位置。也可以使用 **AbsoluteLayout** 来实现老式的基于像素位置的布局控制，但除像素单位外，它也支持百分比单位，对于可伸缩的布局很有用。这个布局管理器作为一个组件拖放区域也是很有用的，用户可以通过拖放操作来移动组件的位置。

这段故事的含义是，由于 Vaadin 的目的是开发 Web 应用程序，因此外观表现是很重要的。解决方案必须在两个世界中都是最佳的，要满足两种艺术：编程和美工。在 API 这一边，布局由 UI 组件控制，具体来说是布局管理组件。在可视外观那一边，布局由 Theme 控制。Theme 可以包含任意的 HTML, Sass, CSS，以及 JavaScript，由你或者你的 Web 设计人员创建这些内容，以便为你的软件的使用者们提供更好的体验。

6.1. 概述

Vaadin 中的 UI 组件大体上可以分为两组：用户可以与之交互的组件，以及用于在 UI 中将其他组件放置到特定位置的布局管理组件。布局管理组件的目的，与通常的 Java 桌面应用程序开发框架中的布局管理器是一致的。你可以使用简单的 Java 程序来实现复杂的组件布局管理。

你首先要创建一个布局作为整个 UI 的内容，然后逐层添加布局管理组件，最后在组件层次树的最下层添加用户交互组件。

```
// Set the root layout for the UI
VerticalLayout content = new VerticalLayout();
setContent(content);

// Add the topmost component.
content.addComponent(new Label("The Ultimate Cat Finder"));

// Add a horizontal layout for the bottom part.
HorizontalLayout bottom = new HorizontalLayout();
content.addComponent(bottom);
```

```
bottom.addComponent(new Tree("Major Planets and Their Moons"));
bottom.addComponent(new Panel());
...

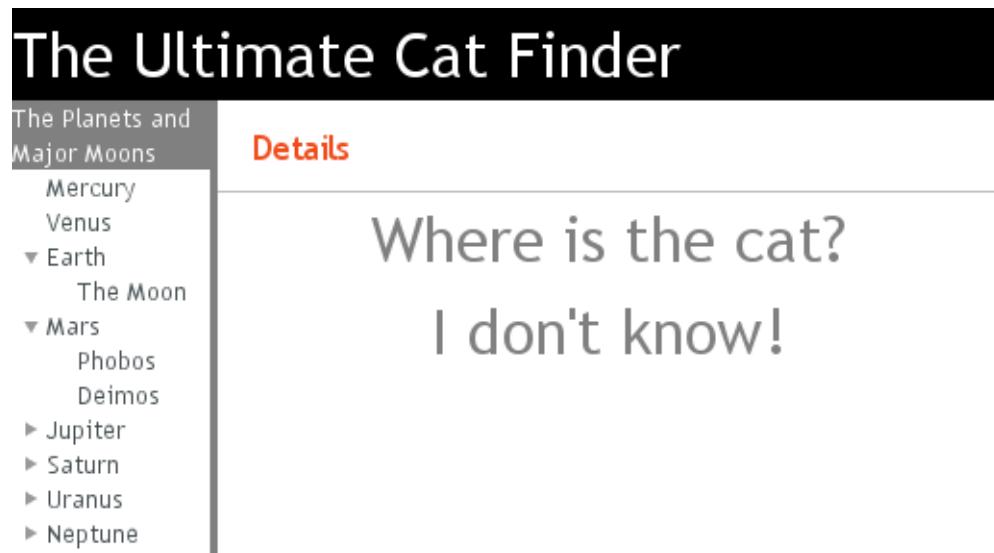
```

你通常需要轻微的调整布局管理组件, 设置它的尺寸, 扩张比例, 对齐方式, 间隔空白, 等等. 通常的设定请参见 第 6.13 节 “布局格式控制”.

布局是与 Theme 对应的, Theme 指定了布局的很多特性, 比如背景, 边框, 文字对齐方向, 等等. Theme 的定义和使用请参见 第 7 章 Themes

上面示例程序的最终完成结果见 图 6.1 “布局示例”.

图 6.1. 布局示例



除了使用布局管理组件之外, 另一种替代方案是使用特别的 **CustomLayout** 组件, 它允许使用 HTML 模板. 在这种方式下, 你可以让 Web 页面设计人员使用他们自己的开发工具来负责组件的布局. 但你将失去动态管理布局的能力.


可视化编辑器

你可以通过程序来手工维护布局, 但 Vaadin plugin for Eclipse IDE 也包含了可视化(所见即所得, WYSIWYG)的编辑器, 你可以使用这个编辑器来创建 UI. 编辑器生成代码, 这段代码会创建UI, 这种方式对于快速应用程序开发和快速原型开发非常有用. 如果你还在 Vaadin 框架的学习阶段, 它对你尤其有帮助, 因为自动生成的代码, 被设计为可重用性尽量高, 因此它也是如何使用 Vaadin 创建 UI 的很好的示例. 关于可视化编辑器, 参见 第 10 章 在 Eclipse 中进行可视化的 UI 设计.

6.2. UI, Window, 以及 Panel 内容

UI, **Window**, 以及它们的父类 **Panel** 都含有单个的内容组件, 你需要使用 `setContent()` 方法来设置这个内容组件. 内容通常是一个布局管理组件, 当然其他任何组件也都是允许的.

```
Panel panel = new Panel("This is a Panel");
VerticalLayout panelContent = new VerticalLayout();
panelContent.addComponent(new Label("Hello!"));
```

```

panel.setContent(panelContent);

// Set the panel as the content of the UI
setContent(panel);

```

内容的尺寸就是特定的布局管理组件的默认尺寸, 比如, 一个 **VerticalLayout** 的宽度为 100%, 高度则默认为未指定(恰巧与 **Panel** 和 **Label** 的默认尺寸一样). 这样一个未指定高度的布局, 如果它的高度超过浏览器窗口的高度, 它的一部分会滚动到可见区域之外, 因此会出现滚动条. 在很多应用程序中, 你会希望使用浏览器可见区域的全部空间. 将内容布局之内包含的组件设置为全尺寸是不够的, 而且, 如果内容布局本身的高度未指定, 这样做还会导致一个不正确的状态.

```

// First set the root content for the UI
VerticalLayout content = new VerticalLayout();
setContent(content);

// Set the content size to full width and height
content.setSizeFull();

// Add a title area on top of the screen. This takes
// just the vertical space it needs.
content.addComponent(new Label("My Application"));

// Add a menu-view area that takes rest of vertical space
HorizontalLayout menuvie = new HorizontalLayout();
menuvie.setSizeFull();
content.addComponent(menuvie);

```

关于布局的尺寸设置, 详情请参见 第 6.13.1 节 “布局的尺寸”.

6.3. VerticalLayout 和 HorizontalLayout

VerticalLayout 和 **HorizontalLayout** 是有序的布局, 用于将内部组件按垂直或水平方向排列. 它们和 **FormLayout** 一样, 都继承自 **AbstractOrderedLayout**. 这是 Vaadin 中最重要的两个布局管理组件, 通常会用 **VerticalLayout** 作为 UI 的内容根组件.

VerticalLayout 默认宽度为 100%, 默认高度为未指定, 因此它会在水平方向上充满它的父布局(或 UI), 在垂直方向上的高度会适应它本身内容的尺寸. **HorizontalLayout** 在垂直和水平方向上的尺寸都是未指定.

这两个布局的通常用法如下:

```

VerticalLayout vertical = new VerticalLayout ();
vertical.addComponent(new TextField("Name"));
vertical.addComponent(new TextField("Street address"));
vertical.addComponent(new TextField("Postal code"));
layout.addComponent(vertical);

```

组件的标题被放置在组件上方, 因此布局的显示大致如下:

Name	<input type="text"/>
Street address	<input type="text"/>
Postal code	<input type="text"/>

使用 **HorizontalLayout** 的布局显示如下:

Name	Street address	Postal code
------	----------------	-------------

6.3.1. 在有顺序的布局中控制组件间隔

有顺序的布局可以在水平和垂直方向的单元格之间控制间隔。
间隔功能可以使用 `setSpacing(true)` 方法启用。

间隔的默认宽度或高度可以通过 CSS 来定制。你需要为 `v-spacing` 样式的间隔元素设置宽度或高度。你还需要指定一个在 CSS 选择器中指定封装这个间隔元素的容器元素，比如，对于 **VerticalLayout** 来说，容器元素的样式为 `v-verticallayout`，对于 **HorizontalLayout**，容器元素的样式为 `v-horizontallayout`。你也可以用 `v-vertical` 和 `v-horizontal` 来控制所有的垂直或水平布局，比如 **FormLayout**。

比如，下面的例子对 UI 中所有的 **VerticalLayout** 和 **FormLayout** 设置间隔空白的尺寸：

```
.v-vertical > .v-spacing {
    height: 30px;
}
```

对于 **HorizontalLayout**，如下：

```
.v-horizontal > .v-spacing {
    width: 50px;
}
```

6.3.2. 控制布局内组件的尺寸

有顺序的布局之内包含的组件，有几种不同的排列方式，取决于你如何指定组件在布局的主方向上的高度/宽度。

图 6.2. **HorizontalLayout** 中组件的宽度

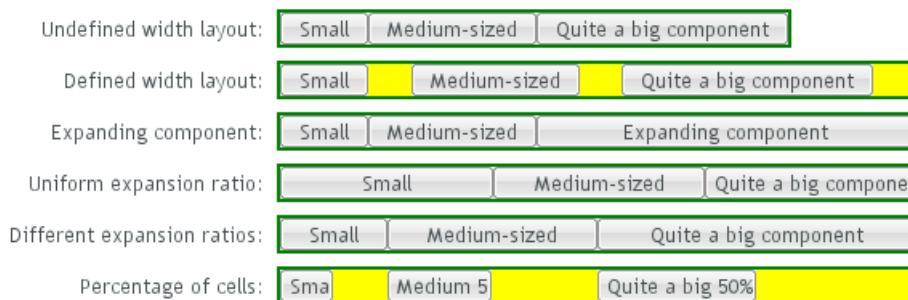


图 6.2 “**HorizontalLayout** 中组件的宽度”大致解释了在 **HorizontalLayout** 内可用的几种尺寸。上图的内容我们分成后面的几个小节来解释。

未指定尺寸的布局

如果 **VerticalLayout** 的高度未指定，或者 **HorizontalLayout** 的宽度未指定，这种情况下布局管理组件会缩小到适合于它内部组件的大小，因此内部组件之间不会带有额外的间隔空白。

```
HorizontalLayout fittingLayout = new HorizontalLayout();
fittingLayout.setWidth(Sizeable.SIZE_UNDEFINED, 0); // Default
```

```

fittingLayout.addComponent(new Button("Small"));
fittingLayout.addComponent(new Button("Medium-sized"));
fittingLayout.addComponent(new Button("Quite a big component"));
parentLayout.addComponent(fittingLayout);

```



垂直和水平布局的默认高度都是未指定，**HorizontalLayout** 的默认宽度也是未指定，而 **VerticalLayout** 默认为 100% 的相对宽度。

这种未指定高度的垂直布局，如果它的内容一致延续到窗口(**Window** 对象)底部的更下方，窗口会在右侧显示出垂直滚动条。这时，你的应用程序的显示就变得和“Web 页面”类似了。这种特性在 **Panel** 中同样存在。



如果布局中包含的组件使用了百分比尺寸，那么容器布局本身必须指定尺寸！

如果布局尺寸未指定，并且其中包含的组件使用相对尺寸，比如，100% 相对大小，那么组件会占据布局给出的所有空间，而布局却想要缩小自己的尺寸与它包含的组件相适应，于是发生矛盾。这里所说的限制对于高度和宽度两个方向都是存在的。使用调试窗口可以查看这种无效的情况；详情请参见 第 11.3.5 节“查看组件层级关系”。

上面这条规则有一个例外，假如布局的尺寸为未指定，内含的一个组件尺寸为固定或未指定，其他一个或多个组件为相对尺寸。这种情况下，内含组件中拥有固定(或未指定)尺寸的那个，可以用来决定布局容器的尺寸，于是解决了上面所说的矛盾。布局容器的尺寸确定之后，其他组件的相对尺寸也就确定了。

这种方法可以用来决定 **VerticalLayout** 的宽度，或者 **HorizontalLayout** 的高度。

```

// Vertical layout would normally have 100% width
VerticalLayout vertical = new VerticalLayout();

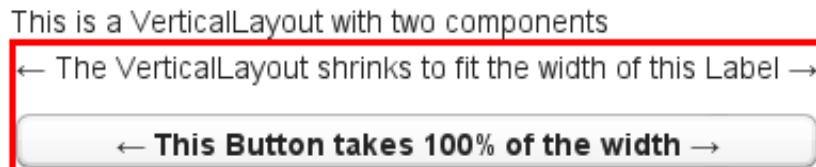
// Shrink to fit the width of contained components
vertical.setWidth(Sizeable.SIZE_UNDEFINED, 0);

// Label has normally 100% width, but we set it as
// undefined so that it will take only the needed space
Label label =
    new Label("\u2190 The VerticalLayout shrinks to fit "+
              "the width of this Label \u2192");
label.setWidth(Sizeable.SIZE_UNDEFINED, 0);
vertical.addComponent(label);

// Button has undefined width by default
Button butt = new Button("\u2190 This Button takes 100% "+
                           "of the width \u2192");
butt.setWidth("100%");
vertical.addComponent(butt);

```

图 6.3. 使用组件来控制尺寸



指定尺寸的布局

如果你将 **HorizontalLayout** 设置为固定的宽度, 或将一个 **VerticalLayout** 设置为固定高度, 假如布局除掉内含的组件之外还有剩余空间, 那么这部分剩余空间会在各个组件之间平均分配. 组件之间按照设定来对齐, 默认是左上对齐, 如下例所示.

```
fixedLayout.setWidth("400px");
```



对布局内的组件使用相对百分比尺寸, 需要回答一个问题, "这是相对于谁的百分比?" 在现在的布局实现中, 这个问题没有合理的默认答案, 所以在实际运用中, 你不应该单独定义 "100%" 尺寸.

组件的扩张

很多时候你会希望某个组件占据其他组件所留下的所有剩余空间. 你需要设置它的尺寸为 100%, 并使用 `setExpandRatio()` 方法将它设置为 扩张. 这个方法的第二个参数是扩张比例, 这个比例是指存在多个扩张组件时的相对比例, 只存在一个扩张组件时, 这个比例值是无意义的.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// These buttons take the minimum size.
layout.addComponent(new Button("Small"));
layout.addComponent(new Button("Medium-sized"));

// This button will expand.
Button expandButton = new Button("Expanding component");

// Use 100% of the expansion cell's width.
expandButton.setWidth("100%");

// The component must be added to layout before setting the ratio.
layout.addComponent(expandButton);

// Set the component's cell to expand.
layout.setExpandRatio(expandButton, 1.0f);

parentLayout.addComponent(layout);
```



注意, 你必须在调用 `addComponent()` 方法之后才可以调用 `setExpandRatio()` 方法, 因为对于布局内(暂时还)不存在的组件, 布局是无法操作它的.

扩张比例

如果对多个组件指定扩张比例, 它们会按照这个比例来分配可用的空间.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// Create three equally expanding components.
String[] captions = { "Small", "Medium-sized",
                      "Quite a big component" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions[i-1]);
```

```

button.setWidth("100%");
layout.addComponent(button);

// Have uniform 1:1:1 expand ratio.
layout.setExpandRatio(button, 1.0f);
}

```



上例中对所有组件使用相同的比例，所以文字内容较长的组件可能会截断它的内容。下例中，我们使用不同的比例：

```

// Expand ratios for the components are 1:2:3.
layout.setExpandRatio(button, i * 1.0f);

```



如果扩张组件的尺寸定义为百分比(一般是 "100%")，比例会根据相对大小的组件的整体可用空间来计算。比如，假定你的布局宽度为 100 像素，其中包含两个单元格，扩张比例分别为 1.0 和 4.0，而且布局内的两个组件都设置为 `setWidth("100%)`，那么这两个单元格宽度将分别是 20 像素和 80 像素，组件定义的最小尺寸会被忽略。

但是，如果内含组件的尺寸为未指定或者固定，扩张比例是 额外 可用空间的比例。这种情况下，扩张的是额外空间(译注：也就是组件宽度加它后面的间隔空白的宽度)，而不是组件本身。

```

for (int i = 1; i <= 3; i++) {
    // Button with undefined size.
    Button button = new Button(captions[i - 1]);

    layout4.addComponent(button);

    // Expand ratios are 1:2:3.
    layout4.setExpandRatio(button, i * 1.0f);
}

```



将百分比尺寸的扩张组件，与固定(或未指定)尺寸的组件组合在一起是无意义的。这样的组合，对百分比尺寸的组件可能导致一个非常意外的尺寸。

单元格的百分比

组件的百分比尺寸指定的是组件 在它的单元格之内的 尺寸。通常使用 "100%"，但一个更小的百分比，或者一个固定尺寸(比单元格更小)会在单元格内留下空白空间，并根据设定将组件在单元格内对齐，默认对齐到左上方。

```

HorizontalLayout layout50 = new HorizontalLayout();
layout50.setWidth("400px");

String[] captions1 = { "Small 50%", "Medium 50%",
                      "Quite a big 50%" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions1[i-1]);
    button.setWidth("50%");
    layout50.addComponent(button);
}

```

```

    // Expand ratios for the components are 1:2:3.
    layout50.setExpandRatio(button, i * 1.0f);
}
parentLayout.addComponent(layout50);

```



6.4. GridLayout

GridLayout 容器将组件摆放在一个网格中，网格由列数和行数定义。网格的列数和行数也是组件摆放时使用的坐标值。每个组件都可以使用网格内的多个单元格，以(x1,y1,x2,y2)形式的区域来指定，但通常只使用一个单元格。

网格布局管理一个游标，用来按照从左到右，从上到下的顺序添加组件。如果游标超过了右下角，它会自动扩张网格，在最下方增加一个新行。

下例演示 **GridLayout** 的使用方法。addComponent 方法接受的参数是一个组件，另外一个可选参数是坐标位置。坐标位置以x,y(列, 行)的顺序指定，可以是单个单元格，也可以是一个区域。坐标位置的起点为 0。如果坐标位置未指定，会使用游标位置。

```

// Create a 4 by 4 grid layout.
GridLayout grid = new GridLayout(4, 4);
grid.addStyleName("example-gridlayout");

// Fill out the first row using the cursor.
grid.addComponent(new Button("R/C 1"));
for (int i = 0; i < 3; i++) {
    grid.addComponent(new Button("Col " + (grid.getCursorX() + 1)));
}

// Fill out the first column using coordinates.
for (int i = 1; i < 4; i++) {
    grid.addComponent(new Button("Row " + i), 0, i);
}

// Add some components of various shapes.
grid.addComponent(new Button("3x1 button"), 1, 1, 3, 1);
grid.addComponent(new Label("1x2 cell"), 1, 2, 1, 3);
InlineDateField date = new InlineDateField("A 2x2 date field");
date.setResolution(DateField.RESOLUTION_DAY);
grid.addComponent(date, 2, 2, 3, 3);

```

上例的运行结果如下。我们将边框设置为可见，以便看清楚网格中的单元格。

图 6.4. Grid Layout 组件

R/C 1	Col 1	Col 2	Col 3
Row 1	3x1 button		
Row 2	1x2 cell	A 2x2 date field	
		November 2007 Sun Mon Tue Wed Thu Fri Sat 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	
Row 3			

放置在网格内的组件一定不能与已有的组件重叠.

组件间的位置冲突会抛出一个

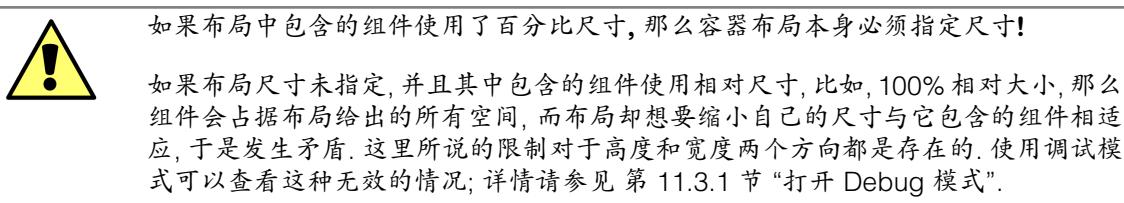
`GridLayout.OverlapsException` 例外.

6.4.1. 调整单元格尺寸

对于网格布局和它之内的组件, 你都可以使用固定或百分比单位定义其尺寸, 也可以将尺寸保留为未指定, 详情请参见 第 5.3.9 节“控制组件的尺寸”. 第 6.13.1 节“布局的尺寸”介绍布局本身的尺寸控制.

GridLayout 组件的尺寸默认为未指定, 因此它会缩小到与它内部的组件尺寸一致. 大多数情况下, 如果你为布局设定了一个指定的尺寸, 但没有将内部的组件设置为全尺寸, 那么网格中会留下一些未使用的空间. 非全尺寸的组件在单元格内的位置由组件的 对齐设置 来决定. 关于如何在单元格内控制组件的对齐, 请参见 第 6.13.3 节“布局单元格的对齐”.

GridLayout 内的组件, 可以以几种不同的方式摆放, 取决与你如何指定它们的高度或宽度. 布局选项类似于 **HorizontalLayout** 和 **VerticalLayout**, 参见 第 6.3 节“**VerticalLayout** 和 **HorizontalLayout**”.



你常常会想要一个或多个行或列占据其他非扩展现行或列留下的所有可用空间. 你需要使用 `setRowExpandRatio()` 和 `setColumnExpandRatio()` 方法, 将这些行或列设置为 扩张. 这两个方法的第一个参数是想要设置为扩张的对象行或列的索引. 第二个参数是扩张比例, 这个比例是指存在多个扩张行/列时的相对比例, 只存在一个扩张行/列时, 这个比例值是无意义的. 存在多个扩张行/列时, 扩张比例参数指定的是某个行/列占据的空间与其他扩张行/列之间的比例.

```
GridLayout grid = new GridLayout(3, 2);

// Layout containing relatively sized components must have
// a defined size, here is fixed size.
grid.setWidth("600px");
grid.setHeight("200px");
```

```

// Add some content
String labels [] = {
    "Shrinking column<br/>Shrinking row",
    "Expanding column (1:)<br/>Shrinking row",
    "Expanding column (5:)<br/>Shrinking row",
    "Shrinking column<br/>Expanding row",
    "Expanding column (1:)<br/>Expanding row",
    "Expanding column (5:)<br/>Expanding row"
};
for (int i=0; i<labels.length; i++) {
    Label label = new Label(labels[i], ContentMode.HTML);
    label.setWidth(null); // Set width as undefined
    grid.addComponent(label);
}

// Set different expansion ratios for the two columns
grid.setColumnExpandRatio(1, 1);
grid.setColumnExpandRatio(2, 5);

// Set the bottom row to expand
grid.setRowExpandRatio(1, 1);

// Align and size the labels.
for (int col=0; col<grid.getColumns(); col++) {
    for (int row=0; row<grid.getRows(); row++) {
        Component c = grid.getComponent(col, row);
        grid.setComponentAlignment(c, Alignment.TOP_CENTER);

        // Make the labels high to illustrate the empty
        // horizontal space.
        if (col != 0 || row != 0)
            c.setHeight("100%");
    }
}

```

图 6.5. **GridLayout** 中的扩张行/列

Shrinking column Shrinking row	Expanding column (1:) Shrinking row	Expanding column (5:) Shrinking row
Shrinking column Expanding row	Expanding column (1:) Expanding row	Expanding column (5:) Expanding row

如果内含组件的尺寸为未定义或者固定尺寸，扩张比例是 额外 可用空间的比例，见 图 6.5 “**GridLayout** 中的扩张行/列”(额外的水平空间显示为白色). 但是，如果在某个扩张行/列内所有组件的尺寸都定义为百分比，扩张比例会以百分比尺寸组件的整体可用空间来计算. 比如，如果有一个 100 像素宽的网格布局，包含两列，扩张比例分别为 1.0 和 4.0，网格内的所有组件都设置为 `setWidth("100%")`，那么列宽将分别是 20 和 80 像素，内含组件定义的最小尺寸会被忽略.

CSS 样式规则

```
.v-gridlayout {}
.v-gridlayout-margin {}
```

`v-gridlayout` 是 **GridLayout** 组件的根元素样式。`v-gridlayout-margin` 是根元素内的一个简单元素，用来设置外层元素与单元格之间的边距。

要分别控制单个单元格的样式，你应该控制插入到单元格内的组件的样式。网格本身的 HTML 元素结构在未来的实现中可能会变化，因此，像下例那样依赖于这个元素结构，一般是不推荐的。通常，如果你希望某个单元格拥有不同的颜色，首先应该将这个单元格内的组件设置为 `setSizeFull()`，然后对组件添加一个样式名。有时，为了样式控制，你可能需要在单元格和真实组件之间再使用一个布局管理组件。

下例演示如何让网格的边框变为可见，参见图 6.5 “**GridLayout** 中的扩张行/列”。

```
.v-gridlayout-gridexpandratio {
    background: blue; /* Creates a "border" around the grid. */
    margin:      10px; /* Empty space around the layout. */
}

/* Add padding through which the background color shows. */
.v-gridlayout-gridexpandratio .v-gridlayout-margin {
    padding: 2px;
}

/* Add cell borders and make the cell backgrounds white.
 * Warning: This depends heavily on the HTML structure. */
.v-gridlayout-gridexpandratio > div > div > div {
    padding: 2px; /* Layout background will show through. */
    background: white; /* The cells will be colored white. */
}

/* Components inside the layout are a safe way to style cells. */
.v-gridlayout-gridexpandratio .v-label {
    text-align: left;
    background: #fffffc0; /* Pale yellow */
}
```

要注意 CSS 中对 `margin`, `padding`, 和 `border` 的设置，它们可能会搞乱布局。布局的尺寸由 Vaadin 的客户端引擎计算，某些设置会妨碍这些计算工作。关于边框余白和间隔空白，请参见第 6.13.4 节“布局单元格的间隔空白”和第 6.13.5 节“布局的余白”。

6.5. FormLayout

FormLayout 将组件和它们的标题摆放为两列，另外对必须 Field 和每个 Field 的错误信息还带有可选的指示器。Field 标题除文字之外还可以带有图标。**FormLayout** 是一个有顺序的布局，与 **VerticalLayout** 很类似。关于有顺序的布局中的边框余白、间隔空白，以及其他功能，请参见第 6.3 节“**VerticalLayout** 和 **HorizontalLayout**”。

下例演示 **FormLayout** 在 Form 中的常见用法：

```
// A FormLayout used outside the context of a Form
FormLayout fl = new FormLayout();

// Make the FormLayout shrink to its contents
fl.setSizeUndefined();
```

```

TextField tf = new TextField("A Field");
fl.addComponent(tf);

// Mark the first field as required
tf.setRequired(true);
tf.setRequiredError("The Field may not be empty.");

TextField tf2 = new TextField("Another Field");
fl.addComponent(tf2);

// Set the second field straining to error state with a message.
tf2.setComponentError(
    new UserError("This is the error indicator of a Field."));

```

运行结果如下图。当你将鼠标指针移动到错误指示器上时，错误消息显示为提示信息。

图 6.6. 用于 Form 的 FormLayout



CSS 样式规则

```

.v-formlayout {}
.v-formlayout .v-caption {}

/* Columns in a field row. */
.v-formlayout-contentcell {} /* Field content. */
.v-formlayout-captioncell {} /* Field caption. */
.v-formlayout-errorcell {} /* Field error indicator. */

/* Overall style of field rows. */
.v-formlayout-row {}
.v-formlayout-firstrow {}
.v-formlayout-lastrow {}

/* Required field indicator. */
.v-formlayout .v-required-field-indicator {}
.v-formlayout-captioncell .v-caption
    .v-required-field-indicator {}

/* Error indicator. */
.v-formlayout-cell .v-errorindicator {}
.v-formlayout-error-indicator .v-errorindicator {}

```

FormLayout 的顶级元素带有 v-formlayout 样式。布局排列为三列：标签列、错误指示器列，以及 Field 列。这些列的样式可以分别通过 v-formlayout-captioncell、v-formlayout-errorcell 和 v-formlayout-contentcell 来控制。错误指示器显示为单独的一列，但必须 Field 的指示器目前显示为标题列的一部分。

关于边框余白和间隔空白, 请参见 第 6.3.1 节 “在有顺序的布局中控制组件间隔” 和 第 6.13.5 节 “布局的余白”.

6.6. Panel

Panel 是一个单组件容器, 在它的内容周围环绕着一个边框. Panel 有一个可选的标题, 图标, 这些都由 Panel 自身管理, 而不由它所属的布局管理组件来管理. Panel 自身不管理它内部包含的组件的标题. 你需要使用 `setContent()` 方法设置 Panel 的内容.

Panel 默认宽度为 100%, 默认高度为未指定. 与 **VerticalLayout** 的默认尺寸一致, 最常用作 **Panel** 内容的可能就是 **VerticalLayout**. 如果 Panel 的宽度或高度为未指定, 那么其中的内容在同一个方向上也必须有一个未指定的或固定的尺寸, 这样才能避免尺寸的矛盾.

```
Panel panel = new Panel("Astronomy Panel");
panel.addStyleName("mypanalexample");
panel.setSizeUndefined(); // Shrink to fit content
layout.addComponent(panel);

// Create the content
FormLayout content = new FormLayout();
content.addStyleName("mypanelcontent");
content.addComponent(new TextField("Participant"));
content.addComponent(new TextField("Organization"));
content.setSizeUndefined(); // Shrink to fit
content.setMargin(true);
panel.setContent(content);
```

结果见 图 6.7 “**Panel**”.

图 6.7. Panel



6.6.1. 滚动 Panel 的内容

通常, 如果 Panel 在某个方向上的尺寸未指定, 比如它的高度默认就是如此, 它会将自己的尺寸调整为与内容尺寸一致, 如果内容扩大 Panel 也会随之扩大. 但是, 如果它的尺寸为固定或百分比, 并且内容尺寸太大不能在 Panel 的内容区域内显示, 那么在 Panel 的这个方向上会出现一个滚动条. Panel 中的滚动条, 是使用 CSS 的 `overflow: auto` 属性, 由浏览器的原生功能处理的.

下例中, 我们有一个 300 像素宽, 并且非常高的 **Image** 组件, 以它作为 Panel 的内容.

```
// Display an image stored in theme
Image image = new Image(null,
    new ThemeResource("img/Ripley_Scroll-300px.jpg"));

// To enable scrollbars, the size of the panel content
// must not be relative to the panel size
image.setSizeUndefined(); // Actually the default
```

```
// The panel will give it scrollbars.
Panel panel = new Panel("Scroll");
panel.setWidth("300px");
panel.setHeight("300px");
panel.setContent(image);

layout.addComponent(panel);
```

结果见图 6.8 “带滚动条的 Panel”. 注意, 虽然 Panel 具有与内容一样的宽度(300 像素), 但还是显示出了水平滚动条 - Panel 的 300 像素宽度还包括了边框, 和垂直滚动条.

图 6.8. 带滚动条的 Panel



通过程序来滚动

Panel 实现了 `Scrollable` 接口, 因此可以通过程序来滚动它. 你可以使用 `setScrollTop()` 和 `setScrollLeft()` 方法, 来设置像素单位的滚动位置. 你也可以得到前面设定的规定位置, 但在浏览器端滚动 Panel 时, 不会将滚动位置信息更新到服务器端.

CSS 样式规则

```
.v-panel {}
.v-panel-caption {}
.v-panel-nocaption {}
.v-panel-content {}
.v-panel-deco {}
```

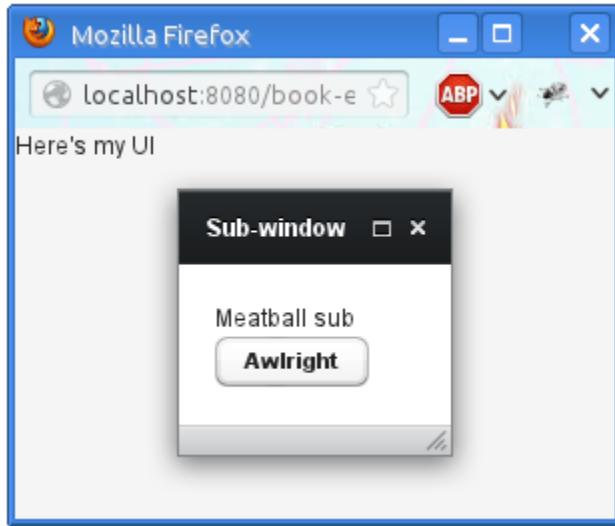
整个 Panel 带有 `v-panel` 样式. Panel 由三部分构成: 标题, 内容, 底部装饰(阴影). 这些部分可以分别使用 `v-panel-caption`, `v-panel-content`, `v-panel-deco` 来控制样式. 如果 Panel 没有标题, 标题元素将带有 `v-panel-nocaption` 样式.

在 Reindeer 和 Runo Theme 中, 对于 **Panel** 组件, 内建的 `light` 样式没有边框和边框装饰. 你可以使用 `Reindeer.PANEL_LIGHT` 和 `Runo.PANEL_LIGHT` 常数, 将这个样式添加给 Panel. 其他 Theme 可能也为 **Panel** 组件提供了 `light` 样式, 或其他样式.

6.7. 子窗口

子窗口是一个浮动的面板，位于原生的浏览器窗口内部。与原生的浏览器窗口不同，子窗口由 Vaadin 的客户端引擎使用 HTML 的功能来管理。Vaadin 允许子窗口的打开、关闭、改变大小、最大化、恢复，以及滚动子窗口中的内容。

图 6.9. 子窗口



子窗口通常用于对话框窗口和多文档界面(MDI, Multiple Document Interface)应用程序。子窗口默认不是模态的；你可以将它设置为模态，参加第 6.7.4 节“模态子窗口”。

6.7.1. 打开和关闭子窗口

你可以打开一个新的子窗口，方法是创建一个新的 **Window** 对象，然后使用 `addWindow()` 方法将它添加到 UI 中，这段处理通常在同一个事件监听器中实现。子窗口需要一个内容组件，它通常是一个布局。

下例中，我们在 UI 打开时立即显示一个子窗口：

```
public static class SubWindowUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Some other UI content
        setContent(new Label("Here's my UI"));

        // Create a sub-window and set the content
        Window subWindow = new Window("Sub-window");
        VerticalLayout subContent = new VerticalLayout();
        subContent.setMargin(true);
        subWindow.setContent(subContent);

        // Put some components in it
        subContent.addComponent(new Label("Meatball sub"));
        subContent.addComponent(new Button("Awlright"));

        // Center it in the browser window
        subWindow.center();
    }
}
```

```

        // Open it in the UI
        addWindow(subWindow);
    }
}

```

结果见图 6.9 “子窗口”. 子窗口默认在宽度高度都为未指定, 因此它会缩小到与其中的内容相适应.

用户可以点击右上角的关闭按钮来关闭子窗口. 这个按钮由 `closable` 属性控制, 因此你可以使用 `setClosable(false)` 方法来禁用这个按钮.

也可以通过程序来关闭一个子窗口, 方法是对子窗口调用 `close()` 方法, 通常会在 **OK** 或 **Cancel** 按钮的点击事件监听器中这样做. 你也可以对当前 **UI** 调用 `removeWindow()` 方法.

子窗口管理

通常, 为了实现你的特定用途的子窗口, 需要继承 **Window** 类, 如下例:

```

// Define a sub-window by inheritance
class MySub extends Window {
    public MySub() {
        super("Subs on Sale"); // Set window caption
        center();

        // Some basic content for the window
        VerticalLayout content = new VerticalLayout();
        content.addComponent(new Label("Just say it's OK!"));
        content.setMargin(true);
        setContent(content);

        // Disable the close button
        setClosable(false);

        // Trivial logic for closing the sub-window
        Button ok = new Button("OK");
        ok.addClickListener(new ClickListener() {
            public void buttonClick(ClickEvent event) {
                close(); // Close the sub-window
            }
        });
        content.addComponent(ok);
    }
}

```

你可以打开窗口, 方法如下:

```

// Some UI logic to open the sub-window
final Button open = new Button("Open Sub-Window");
open.addClickListener(new ClickListener() {
    public void buttonClick(ClickEvent event) {
        MySub sub = new MySub();

        // Add it to the root component
        UI.getCurrent().addWindow(sub);
    }
});

```

6.7.2. 窗口位置

创建后，子窗口默认尺寸和位置都将是未指定。你可以使用 `setHeight()` 和 `setWidth()` 方法，来设置窗口的尺寸。可以使用 `setPositionX()` 和 `setPositionY()` 方法，来设置窗口位置。

```
// Create a new sub-window
mywindow = new Window("My Dialog");

// Set window size.
mywindow.setHeight("200px");
mywindow.setWidth("400px");

// Set window position.
mywindow.setPositionX(200);
mywindow.setPositionY(50);

UI.getCurrent().addWindow(mywindow);
```

6.7.3. 滚动窗口内容

如果子窗口尺寸为固定或百分比，并且它的内容太大，超过了内容显示区域的大小，那么在对应的方向上会显示出一个滚动条。另一方面，如果子窗口在这个方向上的尺寸为未指定，它将调制自己的尺寸来适应内容大小，因此永远不会出现滚动条。子窗口的滚动条是由标准的 HTML 功能处理的，也就是 CSS 中的 `overflow: auto` 属性。

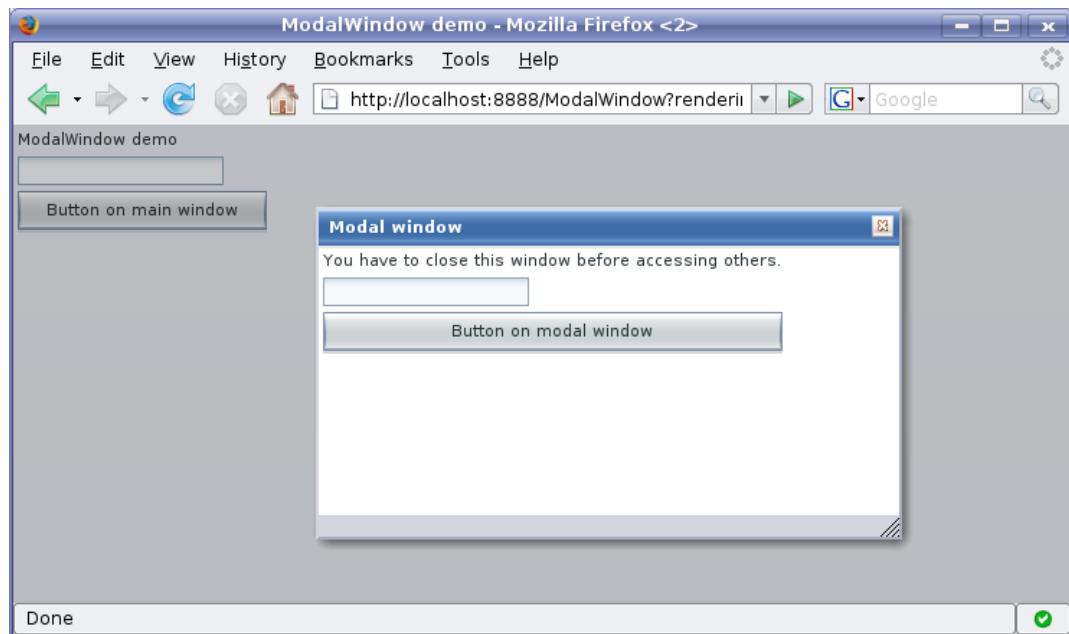
由于 **Window** 继承自 **Panel**，因此窗口也是 `Scrollable`。注意，这个接口定义的功能是通过程序来滚动，而不是由用户操作发生的滚动。详情请参见第 6.6 节“**Panel**”。

6.7.4. 模态子窗口

模态窗口是指子窗口显示时禁止用户操作 UI 的其他部分。对话框窗口，见图 6.10 “模态子窗口”，是模态窗口的典型情况。模态窗口的好处是在进行某个子任务时，可以限制用户操作的范围，因此应用程序的状态变化就很有限。模态窗口的缺点是它可能对用户工作流程造成太大的限制。

你可以使用 `setModal(true)` 方法将一个主窗口设置为模态窗口。

图 6.10. 模态子窗口



根据 Theme 的设定不同, 模态主窗口打开时, 父窗口有可能被变为灰色.



安全问题

子窗口的模态是纯粹的客户端功能, 因此是可以被客户端攻击代码绕过的. 在与系统安全相关的场合, 比如 Login 窗口, 你不应该信任子窗口的模态功能.

6.8. HorizontalSplitPanel 和 VerticalSplitPanel

HorizontalSplitPanel 和 **VerticalSplitPanel** 是双组件容器, 它们将可用的空间分为两个部分, 放置两个组件. **HorizontalSplitPanel** 用垂直分隔条将空间分为水平的两部分, **VerticalSplitPanel** 用水平分隔条将空间分为垂直的两部分. 用户可以拖拽分隔条来调整它的位置.

你可以使用 `setFirstComponent()` 和 `setSecondComponent()` 方法来设置容器内的两个组件, 也可以使用通常的 `addComponent()` 方法.

```
// Have a panel to put stuff in
Panel panel = new Panel("Split Panels Inside This Panel");

// Have a horizontal split panel as its content
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
panel.setContent(hsplit);

// Put a component in the left panel
Tree tree = new Tree("Menu", TreeExample.createTreeContent());
hsplit.setFirstComponent(tree);

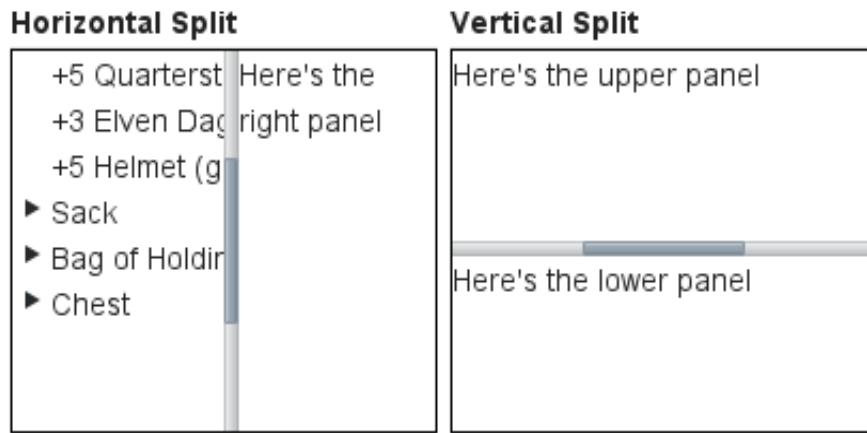
// Put a vertical split panel in the right panel
VerticalSplitPanel vsplit = new VerticalSplitPanel();
hsplit.setSecondComponent(vsplit);

// Put other components in the right panel
```

```
vsplit.addComponent(new Label("Here's the upper panel"));
vsplit.addComponent(new Label("Here's the lower panel"));
```

运行结果见图 6.11 “**HorizontalSplitPanel** 和 **VerticalSplitPanel**”(译注: 图内容与示例代码貌似不符). 注意, Tree 在水平方向上被切断了, 因为它尺寸太大, 无法适应容器尺寸. 如果它的高度超出了容器范围, 会自动出现垂直滚动条. 如果需要水平滚动条的话, 你可以将内容组件放在 **Panel** 内, 它在水平和垂直两个方向上都可以带有滚动条.

图 6.11. HorizontalSplitPanel 和 VerticalSplitPanel



你可以使用 `setSplitPosition()` 方法设置分隔位置. 这个方法可以接受 **Sizeable** 接口中定义的任何参数, 其中的百分比尺寸是相对于组件本身的尺寸.

```
// Have a horizontal split panel
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
hsplit.setFirstComponent(new Label("75% wide panel"));
hsplit.setSecondComponent(new Label("25% wide panel"));

// Set the position of the splitter as percentage
hsplit.setSplitPosition(75, Sizeable.UNITS_PERCENTAGE);
```

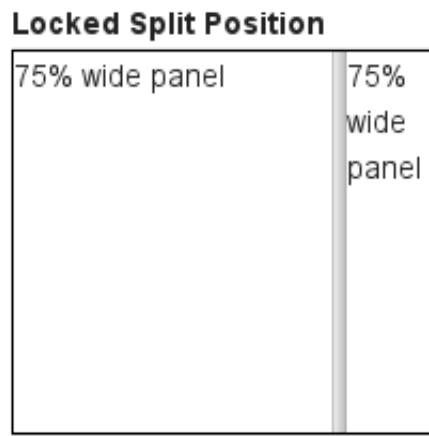
`setSplitPosition()` 方法的另一个版本可以不指定单位, 与前一次使用相同的单位. 这个方法还有一个版本, 接受一个 `boolean` 参数, `reverse`, 这个参数可以用来指定右部/下部的尺寸, 而不是左部/上部的尺寸.

用户可以用鼠标拖动分隔条来调整两部分内容的分隔位置. 可以使用 `setLocked(true)` 方法将分隔条锁定. 锁定之后, 分隔条中间的鼠标拖动区会被禁用.

```
// Lock the splitter
hsplit.setLocked(true);
```

上面的例子通过程序来设置分隔位置, 然后锁定分隔条, 运行结果见图 6.12 “**SplitPanel** 布局(译注: 图的标题貌似不正确)”.

图 6.12. SplitPanel 布局(译注: 图的标题貌似不正确)



注意, SplitPanel 本身在它的分隔方向上的尺寸必须是确定的.

CSS 样式规则

```
/* For a horizontal SplitPanel. */
.v-splitpanel-horizontal {}
.v-splitpanel-hsplitter {}
.v-splitpanel-hsplitter-locked {}

/* For a vertical SplitPanel. */
.v-splitpanel-vertical {}
.v-splitpanel-vsplitter {}
.v-splitpanel-vsplitter-locked {}

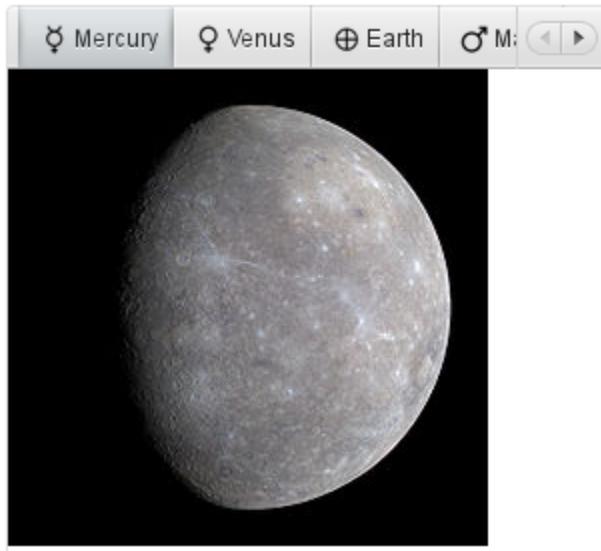
/* The two container panels. */
.v-splitpanel-first-container {} /* Top or left panel. */
.v-splitpanel-second-container {} /* Bottom or right panel. */
```

SplitPanel 整体带有 v-splitpanel-horizontal 或 v-splitpanel-vertical 样式. 两个内容 Panel 之间的分隔条, 或者叫 分隔器 带有 ...-splitter 或 ...-splitter-locked 样式, 具体是哪个样式, 由分隔条是否被锁定决定.

6.9. TabSheet

TabSheet 是一种多组件容器, 它可以用 "tab" 的形式在各个组件之间切换. Tab 以 TabSheet 上方的 Tab Bar 的形式组织. 点击一个 Tab, 会在布局的主显示区域中打开这个 Tab 中包含的组件. 如果 Tab 太多, 超过了 Tab Bar 的显示范围, 会出现导航按钮.

图 6.13. 一个简单的 TabSheet 布局



6.9.1. 添加 Tab

你可以使用 `addTab()` 方法将新的 Tab 添加到 TabSheet 中。这个方法的简单版本接受的参数是 Tab 内容的根组件。你可以使用根组件得到它对应的 **Tab** 对象。通常我们使用布局管理组件作为 Tab 内的根组件。

你也可以在 `addTab()` 方法的参数中指定 Tab 的标题和图标。下例演示了如何创建一个简单的 TabSheet，其中的各个 Tab 显示不同的 **Label** 组件。Tab 带有图标，(在这个例子中) 图标从应用程序的 Java 类资源中载入。

```
TabSheet tabsheet = new TabSheet();
layout.addComponent(tabsheet);

// Create the first tab
VerticalLayout tab1 = new VerticalLayout();
tab1.addComponent(new Embedded(null,
    new ThemeResource("img/planets/Mercury.jpg")));
tabsheet.addTab(tab1, "Mercury",
    new ThemeResource("img/planets/Mercury_symbol.png"));

// This tab gets its caption from the component caption
VerticalLayout tab2 = new VerticalLayout();
tab2.addComponent(new Embedded(null,
    new ThemeResource("img/planets/Venus.jpg")));
tab2.setCaption("Venus");
tabsheet.addTab(tab2).setIcon(
    new ThemeResource("img/planets/Venus_symbol.png"));
...
```

6.9.2. Tab 对象

TabSheet 中的各个 Tab 由 **Tab** 对象来表达，这个对象管理 Tab 的标题，图标，以及其他属性，比如隐藏和可见。你可以使用 `setCaption()` 方法设置标题，使用 `setIcon()` 方法设置图标。使用 `addTab()` 方法添加 Tab 时，如果被添加的组件带有标题或图标，这个标题或图标会被用作 **Tab** 对象的标题或图标。但是，如果之后再改变根组件的属性，不会影响到 Tab 对象，你必须通过 **Tab**

对象来设置这些属性。`addTab()` 方法返回一个新的 **Tab** 对象，因此你可以很容易地通过这个对象引用来自设置属性。

```
// Set an attribute using the returned reference
tabsheet.addTab(myTab).setCaption("My Tab");
```

禁用和隐藏 Tab

对 **Tab** 对象设置 `setEnabled(false)` 可以禁用 Tab，禁用后将无法选择这个 Tab。

对 **Tab** 对象设置 `setVisible(false)` 可以隐藏 Tab。`hideTabs()` 方法可以隐藏整个 Tab Bar。在 Tab 式文档界面(TDI, Tabbed Document Interfaces)中，如果只有一个 Tab，那么隐藏 Tab Bar 是很有用的。

图 6.14. 含隐藏 Tab 和禁用 Tab 的 TabSheet



6.9.3. Tab 切换事件

点击一个 Tab 会选中它。这个操作会激发一个 **TabSheet.SelectedTabChangeEvent** 事件，你可以实现 **TabSheet.SelectedTabChangeListener** 接口来处理这个事件。你可以使用 `getTabSheet()` 方法，取得事件相关的 TabSheet，还可以使用 `getSelectedTab()` 方法得到新选中的 Tab。

你可以通过程序选中 Tab，方法是使用 `setSelectedTab()` 方法，这个方法同样会激发 **SelectedTabChangeEvent** 事件(这时要小心递归事件)。已经选中的 Tab，再次选中它不会激发这个事件。

注意，当添加第一个 Tab 时，它会被选中，并且激发这个事件，所以如果你希望捕捉这个事件的话，你应该在添加 Tab 之前先添加事件监听器。

动态创建 Tab 内容

下例中，我们创建 Tab，在其中只放置空的布局，然后在 Tab 被选中时动态地创建其中的内容：

```
TabSheet tabsheet = new TabSheet();

// Create tab content dynamically when tab is selected
tabsheet.addSelectedTabChangeListener(
    new TabSheet.SelectedTabChangeListener() {
        public void selectedTabChange(SelectedTabChangeEvent event) {
            // Find the tabsheet
            TabSheet tabsheet = event.getTabSheet();

            // Find the tab (here we know it's a layout)
            Layout tab = (Layout) tabsheet.getSelectedTab();

            // Get the tab caption from the tab object
            String caption = tabsheet.getTab(tab).getCaption();

            // Fill the tab content
            tab.removeAllComponents();
        }
    }
);
```

```

        tab.addComponent(new Image(null,
            new ThemeResource("img/planets/"+caption+".jpg")));
    }
});

// Have some tabs
String[] tabs = {"Mercury", "Venus", "Earth", "Mars"};
for (String caption: tabs)
    tabsheet.addTab(new VerticalLayout(), caption,
        new ThemeResource("img/planets/"+caption+"_symbol.png"));

```

6.9.4. 允许关闭 Tab, 处理 Tab 的关闭事件

你可以使用 **TabSheet.Tab** 对象的 `closable` 属性, 为各个 Tab 分别显示关闭按钮.

```
// Enable closing the tab
tabsheet.getTab(tabComponent).setClosable(true);
```

图 6.15. Tab 可关闭的 TabSheet



处理 Tab 的关闭事件

你可以实现自定义的 **TabSheet.CloseHandler** 处理器, 来处理 Tab 的关闭事件. 这个处理器的默认实现只是简单地对被关闭的 Tab 调用 `removeTab()`, 但你可以不调用这个方法, 这样就可以阻止 Tab 关闭. 利用这个机能我们可以打开一个确认对话框, 询问是否要关闭?.

```
tabsheet.setCloseHandler(new CloseHandler() {
    @Override
    public void onTabClose(TabSheet tabsheet,
                           Component tabContent) {
        Tab tab = tabsheet.getTab(tabContent);
        Notification.show("Closing " + tab.getCaption());

        // We need to close it explicitly in the handler
        tabsheet.removeTab(tab);
    }
});
```

CSS 样式规则

```
.v-tabsheets {}
.v-tabsheets-tabs {}
.v-tabsheets-content {}
.v-tabsheets-deco {}
.v-tabsheets-tabcontainer {}
.v-tabsheets-tabsheetpanel {}
.v-tabsheets-hidetabs {}

.v-tabsheets-scroller {}
.v-tabsheets-scrollerPrev {}
```

```
.v-tabsheet-scrollerNext {}
.v-tabsheet-scrollerPrev-disabled{}
.v-tabsheet-scrollerNext-disabled{}

.v-tabsheet-tabitem {}
.v-tabsheet-tabitem-selected {}
.v-tabsheet-tabitemcell {}
.v-tabsheet-tabitemcell-first {}

.v-tabsheet-tabs td {}
.v-tabsheet-spacer td {}}
```

TabSheet 整体带有 v-tabsheet 样式。Tabsheet 由三个主要部分组成：上方的 Tab，主内容显示区，以及 TabSheet 周围的装饰。

上方的 Tab 区域可以使用 v-tabsheet-tabs, v-tabsheet-tabcontainer 和 v-tabsheet-tabitem* 来控制样式。

v-tabsheet-spacer td 样式用于控制 Tab 之后剩余的空白区域。如果 TabSheet 的空间不足以显示所有的 Tab，会出现滚动按钮，用于查看完整的 Tab 列表。滚动按钮使用 v-tabsheet-scroller* 样式。

Tab 内容的显示区域使用 v-tabsheet-content 样式，TabSheet 周围的装饰使用 v-tabsheet-deco 样式。

6.10. Accordion

Accordion 是与 **TabSheet** 类似的多组件容器，区别是，它的“tab”是垂直排列的。点击一个 Tab，会在这个 Tab 和下一个 Tab 之间的位置显示它包含的组件。你可以用与 **TabSheet** 完全相同的方式使用 **Accordion**，Accordion 实际上就是从 TabSheet 继承而来的。详情请参见 第 6.9 节“**TabSheet**”。

下例演示如何创建一个简单的 Accordion。As the **Accordion** 本身不带任何装饰，我们将它放在 Panel 之内，用 Panel 为 Accordion 提供标题和边框。

```
// Create the Accordion.
Accordion accordion = new Accordion();

// Have it take all space available in the layout.
accordion.setSizeFull();

// Some components to put in the Accordion.
Label l1 = new Label("There are no previously saved actions.");
Label l2 = new Label("There are no saved notes.");
Label l3 = new Label("There are currently no issues.");

// Add the components as tabs in the Accordion.
accordion.addTab(l1, "Saved actions", null);
accordion.addTab(l2, "Notes", null);
accordion.addTab(l3, "Issues", null);

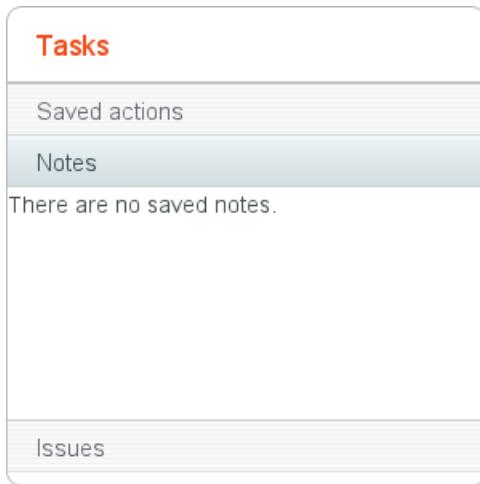
// A container for the Accordion.
Panel panel = new Panel("Tasks");
panel.setWidth("300px");
panel.setHeight("300px");
panel.setContent(accordion);

// Trim its layout to allow the Accordion take all space.
```

```
panel.setLayout().setSizeFull();
panel.setLayout().setMargin(false);
```

图 6.16 “Accordion” 展示了默认 Theme 下示例程序的运行结果.

图 6.16. Accordion



CSS 样式规则

```
.v-accordion {}
.v-accordion-item {}
.v-accordion-item-open {}
.v-accordion-item-first {}
.v-accordion-item-caption {}
.v-accordion-item-caption .v-caption {}
.v-accordion-item-content {}
```

Accordion 的顶级元素带有 v-accordion 样式. **Accordion** 由它之下的一系列项目元素组成, 每个项目元素中含有一个标题元素(Tab 本身), 和一个内容区域元素.

被选中的项目(Tab) 还带有 v-accordion-open 样式. 对于被关闭的项目, 内容区域不会显示.

6.11. AbsoluteLayout

AbsoluteLayout 可以将组件放置在布局区域内的任何位置. 位置在方法 addComponent() 中指定, 值是相对于布局区域边界的水平和垂直坐标. 位置还可以包含第三个维度: 深度, 也叫 z-下标, 它决定哪个组件显示在前方, 哪个组件在其他组件的后方.

位置以 CSS 绝对位置字符串的形式指定, 使用与 CSS 相同的 left, right, top, bottom, 和 z-index 属性. 下例中, 我们创建一个 300 x 150 像素大的布局, 然后将一个文本放置在距离左边界和上边界 50 像素的位置:

```
// A 400x250 pixels size layout
AbsoluteLayout layout = new AbsoluteLayout();
layout.setWidth("400px");
layout.setHeight("250px");

// A component with coordinates for its top-left corner
TextField text = new TextField("Somewhere someplace");
layout.addComponent(text, "left: 50px; top: 50px");
```

`left` 和 `top` 属性分别指定相对于左边界和上边界的距离. `right` 和 `bottom` 分别指定相对于左边界和上边界(译注: 原文如此, 应为"下边界")的距离.

```
// At the top-left corner
Button button = new Button("left: 0px; top: 0px;");
layout.addComponent(button, "left: 0px; top: 0px");

// At the bottom-right corner
Button buttCorner = new Button("right: 0px; bottom: 0px;");
layout.addComponent(buttCorner, "right: 0px; bottom: 0px");

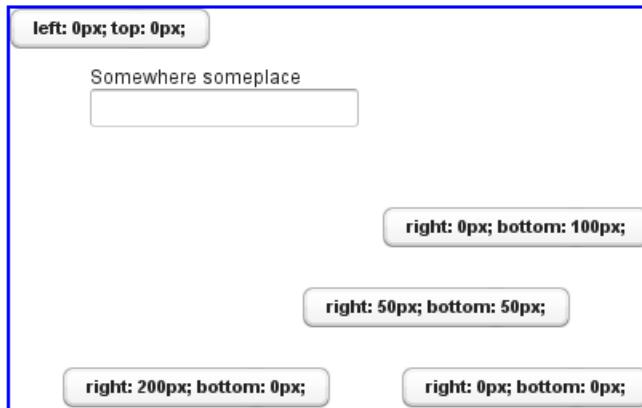
// Relative to the bottom-right corner
Button buttBrRelative = new Button("right: 50px; bottom: 50px;");
layout.addComponent(buttBrRelative, "right: 50px; bottom: 50px");

// On the bottom, relative to the left side
Button buttBottom = new Button("left: 50px; bottom: 0px");
layout.addComponent(buttBottom, "left: 50px; bottom: 0px");

// On the right side, up from the bottom
Button buttRight = new Button("right: 0px; bottom: 100px");
layout.addComponent(buttRight, "right: 0px; bottom: 100px");
```

上例的运行结果见图 6.17 “放置在各种位置上的组件”.

图 6.17. 放置在各种位置上的组件

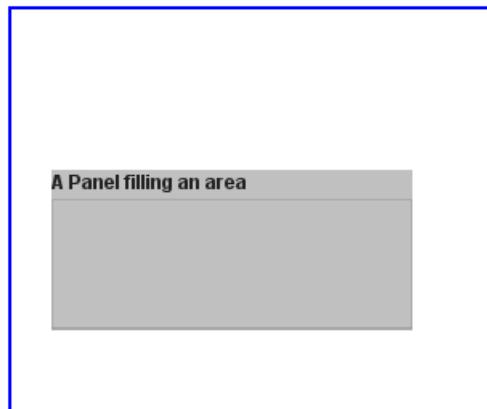


上面的示例中, 我们将组件尺寸设置为未指定, 并用一对坐标来指定组件位置. 另一种方法是指定一个区域, 然后为组件设置一个比例大小, 比如 "100%", 让它填充到这个区域中. 通常可以设置 `setSizeFull()`, 让组件占满布局分配给它的整个区域.

```
// Specify an area that a component should fill
Panel panel = new Panel("A Panel filling an area");
panel.setSizeFull(); // Fill the entire given area
layout.addComponent(panel, "left: 25px; right: 50px; "+
    "top: 100px; bottom: 50px");
```

运行结果见图 6.18 “用坐标指定一个区域, 组件充满这个区域”

图 6.18. 用坐标指定一个区域，组件充满这个区域

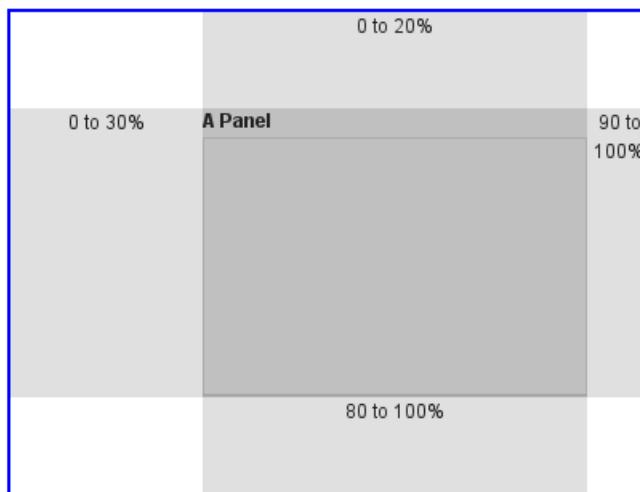


也可以用比例坐标来指定区域:

```
// A panel that takes 30% to 90% horizontally and
// 20% to 80% vertically
Panel panel = new Panel("A Panel");
panel.setSizeFull(); // Fill the specified area
layout.addComponent(panel, "left: 30%; right: 10%;" +
    "top: 20%; bottom: 20%;");
```

运行结果见 图 6.19 “通过比例坐标指定区域”

图 6.19. 通过比例坐标指定区域



对于 **AbsoluteLayout** 内的组件，使用拖放操作来移动它们的位置是很便利的方法。示例请参见第 11.12.6 节“拖动到组件上”。

使用 CSS 控制样式

```
.v-absolutelayout {}
.v-absolutelayout-wrapper {}
```

AbsoluteLayout 组件的顶层式样为 `v-absolutelayout`. 布局内的每个组件都包含在一个元素内, 这个元素的样式是 `v-absolutelayout-wrapper`. 组件标题在这个包装元素之外, 位于一个独立的元素之内, 使用通常的 `v-caption` 样式.

6.12. CssLayout

CssLayout 允许对布局内组件的样式进行高度控制. 组件被包含在 `<div>` 元素构成的简单的 DOM 结构中. 默认情况下, 被包含的组件水平排列, 达到布局最大款式都自然折行, 但你可以通过 CSS 来控制这个行为, 其他大多数行为也都可以通过 CSS 来控制. 你还可以为每个组件注入自定义的 CSS. 由于 **CssLayout** 的 DOM 结构非常简单, 也没有动态的描绘逻辑, 它只依赖于浏览器本身内建的描绘逻辑, 所以它是所有布局管理组件中执行速度最快的.

CssLayout 的基本用法与其他布局管理组件类似:

```
CssLayout layout = new CssLayout();

// Component with a layout-managed caption and icon
TextField tf = new TextField("A TextField");
tf.setIcon(new ThemeResource("icons/user.png"));
layout.addComponent(tf);

// Labels are 100% wide by default so must unset width
Label label = new Label("A Label");
label.setWidth(Sizeable.SIZE_UNDEFINED, 0);
layout.addComponent(label);

layout.addComponent(new Button("A Button"));
```

运行结果见图 6.20 “**CssLayout** 的基本使用”. 注意, 布局默认的间隔空白和对齐控制是非常粗糙的, 通常总是需要你通过 CSS 来控制其中的样式.

图 6.20. **CssLayout** 的基本使用



CssLayout 的 `display` 属性默认值是 `inline-block`, 因此组件在水平方向逐个排列. **CssLayout** 宽度默认为 100%. 如果内部组件达到了布局宽度的边界, 它们会向文本一样折到下一“行”. 如果你添加了一个 100% 宽度的组件, 它将占据整行, 它之前和之后都会出现折行.

6.12.1. CSS 注入

覆盖 `getCss()` 方法可以为每个组件注入自定义 CSS. 这个方法返回的 CSS 会被插入到组件的 `<div>` 元素的 `style` 属性中, 因此它会覆盖掉 CSS 文件导致的所有样式定义.

```
CssLayout layout = new CssLayout() {
    @Override
    protected String getCss(Component c) {
        if (c instanceof Label) {
            // Color the boxes with random colors
            int rgb = (int) (Math.random()*(1<<24));
            return "background: #" + Integer.toHexString(rgb);
        }
        return null;
    }
};
```

```

layout.setWidth("400px"); // Causes to wrap the contents

// Add boxes of various sizes
for (int i=0; i<40; i++) {
    Label box = new Label("&nbsp;", ContentMode.HTML);
    box.addStyleName("flowbox");
    box.setWidth((float) Math.random()*50,
                Sizeable.UNITS_PIXELS);
    box.setHeight((float) Math.random()*50,
                  Sizeable.UNITS_PIXELS);
    layout.addComponent(box);
}

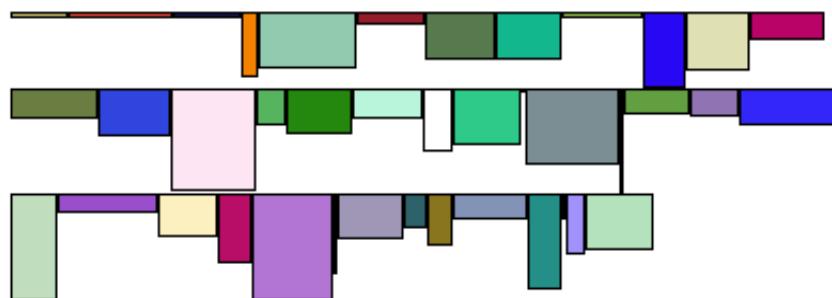
```

添加到组件上的样式名可以在 CSS 中定义共通样式:

```
.
.v-label-flowbox {
    border: thin black solid;
}
```

图 6.21 “getCss() 方法和折行的使用” 显示了上例的运行结果

图 6.21. getCss() 方法和折行的使用



6.12.2. 浏览器兼容性

CssLayout 的优点同时也是它的弱点。其他布局管理组件之后隐藏着很多逻辑，目的是实现良好的默认行为，并处理各种浏览器之间的差异。某些浏览器，我们就不必提它们的名字了，是著名的与 CSS 标准不兼容，因此对它们需要编写大量的定制 CSS。你可能会需要在应用程序的根元素上使用某些浏览器独有的样式类。其他布局中的功能甚至是用纯 CSS 无法实现的，至少不能对所有的浏览器实现。

使用 CSS 控制样式

```
.
.v-csslayout {}
.v-csslayout-margin {}
.v-csslayout-container {}
```

CssLayout 组件的根样式为 `v-csslayout`。余白元素使用 `v-csslayout-margin` 样式，这个元素永远是有效的。组件包含在 `v-csslayout-container` 样式的元素之内。

比如，我们可以对前面例子中的 **CssLayout** 定义它的样式，如下：

```
/* Have the caption right of the text box, bottom-aligned */
.csslayoutexample .mylayout .v-csslayout-container {
    direction: rtl;
    line-height: 24px;
```

```

    vertical-align: bottom;
}

/* Have some space before and after the caption */
.csslayoutexample .mylayout .v-csslayout-container .v-caption {
    padding-left: 3px;
    padding-right: 10px;
}

```

这个示例的显示效果见图 6.22 “控制 **CssLayout** 的样式”.

图 6.22. 控制 **CssLayout** 的样式



由布局来管理的标题和图标，会被包含在 `v-caption` 样式元素内。标题元素与真实组件处于同一个层次上，平行排列。`inline-block` 模式下折行时，可能会出问题，因为折行不仅发生在组件和组件之间，也可能发生在标题和对应的组件元素之间。因此这样的用法是不适当的。

6.13. 布局格式控制

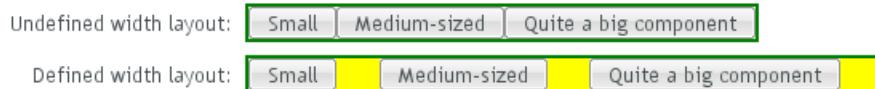
虽然布局的格式主要是由样式表实现的，和其他组件一样，某些情况下，样式表不是理想的解决方案，甚至完全不可用。比如，CSS 不允许定义表格单元格之间的间隔空白，间隔空白是通过 HTML 的 `cellspacing` 属性实现的。

此外，由于很多布局的尺寸是在 Vaadin 的客户端引擎中动态计算的，某些 CSS 设置可能会完全失效。

6.13.1. 布局的尺寸

和任何组件一样，布局组件的尺寸可以使用 `setWidth()` 和 `setHeight()` 方法来设置，这两个方法定义在 **Sizeable** 接口中。尺寸也可以是未指定，这时布局会缩小到适应于其中包含的组件大小。关于 **Sizeable** 接口，详情见第 5.3.9 节“控制组件的尺寸”。

图 6.23. **HorizontalLayout** 布局，未指定尺寸和指定尺寸的情况



很多布局组件默认占据 100% 宽度，默认高度则为未指定。

布局内部的组件的尺寸也可以定义为百分比，这个百分比相对于布局内的可用空间，比如，可以使用 `setWidth("100%");`，也可以使用 `setFullSize()`（这是最常用的），这个方法会将宽高都设置为 100%。如果在 **HorizontalLayout**, **VerticalLayout**, 或 **GridLayout** 内使用百分比尺寸，你还需要将组件设置为 扩张，详见下文。

!
警告

如果布局中包含的组件使用了百分比尺寸，那么容器布局本身必须指定尺寸！

如果布局尺寸未指定，并且其中包含的组件使用相对尺寸，比如，100% 相对大小，那么组件会占据布局给出的所有空间，而布局却想要缩小自己的尺寸与它包含的组件相适

应, 于是发生矛盾. 这里所说的限制对于高度和宽度两个方向都是存在的. 使用调试模式可以查看这种无效的情况; 详情请参见 第 11.3.5 节“查看组件层级关系”.

比如

```
// This takes 100% width but has undefined height.
VerticalLayout layout = new VerticalLayout();

// A button that takes all the space available in the layout.
Button button = new Button("100%x100% button");
button.setSizeFull();
layout.addComponent(button);

// We must set the layout to a defined height vertically, in
// this case 100% of its parent layout, which also must
// not have undefined size.
layout.setHeight("100%");
```

如果你在 **UI**, **Window**, 或 **Panel** 之内有一个布局, 比如 **VerticalLayout**, 高度为未指定, 然后在其中放置了足够多的内容, 使得它的尺寸超过了外部容器的可见区域, 这时就会出现滚动条. 如果你希望你的应用程序使用浏览器的全部视图, 不要更多也不要更少, 那么你应该对最顶层布局使用 `setSizeFull()`.

```
// Create the UI content
VerticalLayout content = new VerticalLayout();

// Use entire view area
content.setSizeFull();

setContent(content);
```

6.13.2. 组件的扩张

如果你将 **HorizontalLayout** 设置为确定的宽度, 或者将 **VerticalLayout** 设置为确定的高度, 并且除了内涵的组件外, 还有剩余的空间, 那么剩余空间会平均分配在各个组件单元格之间. 组件会在各自的单元格内按照设定的方向对齐, 默认是朝左上对齐, 如下例所示.



很多时候你不希望留下这些空白, 而希望一个或多个组件占据所有的剩余空间. 你需要将这些组件设置为 100% 尺寸, 并使用 `setExpandRatio()` 方法来设置扩张比例. 如果布局内这样的扩张组件只有一个, 那么扩张比例的值是无所谓的.



如果你设置了多个扩张组件, 那么它们各自的扩张比例决定了各个组件的尺寸(如果组件使用百分比尺寸, 那么将是组件自身的尺寸, 否则将是组件的额外尺寸)将在剩余空间中占据多大的比例. 下例中, 按钮的扩张比例是 1:2:3.



GridLayout 对于垂直和水平方向分别使用 `setRowExpandRatio()` 和 `setColumnExpandRatio()` 方法.

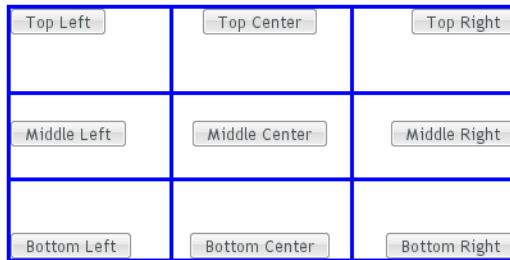
关于组件的扩张, 在支持这个功能的各个布局的文档中有详细介绍. 关于组件的相对尺寸, 参见 第 6.3 节“**VerticalLayout** 和 **HorizontalLayout**”和 第 6.4 节“**GridLayout**”.

6.13.3. 布局单元格的对齐

你可以使用 `setComponentAlignment()` 方法来设置组件在某个布局单元格内的对齐方式。这个方法接受的参数是单元格内的组件，以及它在水平和垂直方向上的对齐方式。

图 6.24 “布局单元格内组件的对齐”展示了 **GridLayout** 之内组件对齐的情况。

图 6.24. 布局单元格内组件的对齐



设置对齐方式的最简便方法是使用 **Alignment** 类中定义的常数。我们来看看上图中 **GridLayout** 第一行内的按钮是如何使用这些常数来对齐的：

```
// Create a grid layout
final GridLayout grid = new GridLayout(3, 3);

grid.setWidth(400, Sizeable.UNITS_PIXELS);
grid.setHeight(200, Sizeable.UNITS_PIXELS);

Button topleft = new Button("Top Left");
grid.addComponent(topleft, 0, 0);
grid.setComponentAlignment(topleft, Alignment.TOP_LEFT);

Button topcenter = new Button("Top Center");
grid.addComponent(topcenter, 1, 0);
grid.setComponentAlignment(topcenter, Alignment.TOP_CENTER);

Button topright = new Button("Top Right");
grid.addComponent(topright, 2, 0);
grid.setComponentAlignment(topright, Alignment.TOP_RIGHT);
...
```

下表列出了所有的 **Alignment** 常数，表中常数排列的位置代表它的对齐方向：

表 6.1. 对齐方式常数

<code>TOP_LEFT</code>	<code>TOP_CENTER</code>	<code>TOP_RIGHT</code>
<code>MIDDLE_LEFT</code>	<code>MIDDLE_CENTER</code>	<code>MIDDLE_RIGHT</code>
<code>BOTTOM_LEFT</code>	<code>BOTTOM_CENTER</code>	<code>BOTTOM_RIGHT</code>

指定对齐方式的另一种方法是创建一个 **Alignment** 对象，分别指定它在水平和垂直方向上的对齐方式。你可以指定其中一个，这时另一个方向的对齐设置不会变化，也可以在两个方向的设定上使用 bit 结合操作，同时指定两个方向的对齐设置。

```
Button middleleft = new Button("Middle Left");
grid.addComponent(middleleft, 0, 1);
grid.setComponentAlignment(middleleft,
```

```

new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
    Bits.ALIGNMENT_LEFT));

Button middlecenter = new Button("Middle Center");
grid.addComponent(middlecenter, 1, 1);
grid.setComponentAlignment(middlecenter,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_HORIZONTAL_CENTER));

Button middleright = new Button("Middle Right");
grid.addComponent(middleright, 2, 1);
grid.setComponentAlignment(middleright,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_RIGHT));

```

显然,你只能将一个垂直方向的常数与一个水平方向的常数结合,当然你也可以不管其中的某一个.下表列出了所有可用的对齐设定 Bitmask 常数:

表 6.2. 对齐设定的 **Bitmask** 常数

Horizontal	<i>Bits.ALIGNMENT_LEFT</i>
	<i>Bits.ALIGNMENT_HORIZONTAL_CENTER</i>
	<i>Bits.ALIGNMENT_RIGHT</i>
Vertical	<i>Bits.ALIGNMENT_TOP</i>
	<i>Bits.ALIGNMENT_VERTICAL_CENTER</i>
	<i>Bits.ALIGNMENT_BOTTOM</i>

你可以使用 `getComponentAlignment()` 方法来得到一个组件的对齐设定,返回值是 **Alignment** 对象.这个类提供了很多 getter 方法用于解析对齐设定,当然对齐设定也可以以 Bitmask 值的形式得到.

对齐的组件的尺寸

只有在某个方向上,组件的尺寸小于它所在单元格的尺寸时,你才可以控制它的对齐.如果组件宽度为 100%,很多组件的默认都是如此,那么水平对齐的设定不会有任何效果.比如, **Label** 默认宽度为 100%,因此不能进行水平对齐.这个问题可能很难注意到,因为 **Label** 之内的文字是左对齐的.

对于需要对齐的组件,你通常应该设置为固定大小,未指定大小,或小于 100% 的相对大小,也就是要小于布局的大小.

比如,假设一个 **Label** 的内容比较短,宽度小于它所在的 **VerticalLayout**,你可以将它居中对齐,如下:

```

VerticalLayout layout = new VerticalLayout(); // 100% default width
Label label = new Label("Hello"); // 100% default width
label.setSizeUndefined();
layout.addComponent(label);
layout.setComponentAlignment(label, Alignment.MIDDLE_CENTER);

```

如果设置组件的尺寸为未指定,而组件本身又包含组件,这是需要保证被包含的组件的尺寸要么是未指定,要么是固定尺寸.比如,如果设置 **Form** 的尺寸为未指定,它包含的 **FormLayout** 布局默认宽度为 100%,你必须将宽度也设置为未指定.但是这样以来, **FormLayout** 之内包含的所有组件也必须是未指定尺寸,或固定尺寸.

6.13.4. 布局单元格的间隔空白

VerticalLayout, **HorizontalLayout**, 和 **GridLayout** 布局提供了 `setSpacing()` 方法, 可以设置布局单元格之间的间隔空白.

比如

```
VerticalLayout layout = new VerticalLayout();
layout.setSpacing(true);
layout.addComponent(new Button("Component 1"));
layout.addComponent(new Button("Component 2"));
layout.addComponent(new Button("Component 3"));
```

VerticalLayout 和 **HorizontalLayout** 之内间隔空白的效果, 见图 6.25 “布局中的间隔空白”.

图 6.25. 布局中的间隔空白

		No spacing:	Vertical spacing:
No spacing:		Component 1 Component 2 Component 3	Component 1 Component 2 Component 3
Horizontal spacing:		Component 1 Component 2 Component 3	Component 1 Component 2 Component 3

间隔空白的具体大小由 CSS 决定. 如果默认设置不适合你的需要, 你可以使用自定义 theme 来定制它.

在 Valo theme 中, 你可以使用 `$v-layout-spacing-vertical` 和 `$v-layout-spacing-horizontal` 参数来指定间隔空白, 详情请参见第 7.6.2 节“共通设定”. 间隔空白的默认值由 `$v-unit-size` 指定.

当在其他 theme 中调整间隔空白时需要注意, 间隔空白的实现方式与布局的实现略有不同. 在有顺序的布局中, 间隔空白使用空白元素来实现, **GridLayout** 布局则有自己的特殊控制. 详情请阅读各布局组件的文档.

6.13.5. 布局的余白

大多数布局组件默认在它们的周围不带有余白. 有序的布局, 以及 **GridLayout**, 可以使用 `setMargin()` 方法启用余白功能. 这个方法会为布局组件的 HTML 元素各个方向的余白设置对应的 CSS 样式.

在 Valo theme 中, 余白尺寸默认由 `$v-unit-size` 指定. 你可以通过 `$v-layout-margin-top`, `right`, `bottom`, 和 `left` 来定制余白尺寸. 关于这些参数, 详情请参见第 7.6.2 节“共通设定”.

要在其他 theme 中定制默认的余白, 你可以使用 CSS 的 `padding` 属性来定义各个方向的余白. 你可能希望为布局组件指定自定义的 CSS 样式, 以便控制各个方向的余白, 下例中, 我们使用 `mymargins` 样式实现这一点:

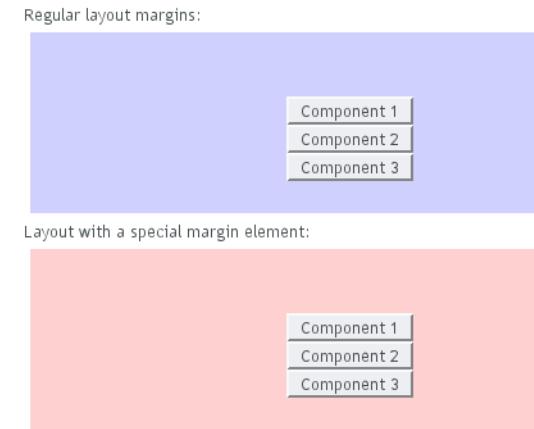
```
.mymargins.v-margin-left {padding-left: 10px;}
.mymargins.v-margin-right {padding-right: 20px;}
.mymargins.v-margin-top {padding-top: 40px;}
.mymargins.v-margin-bottom {padding-bottom: 80px;}
```

你可以只允许某些方向的余白，方法是传递一个 **MarginInfo** 对象作为 `setMargin()` 方法的参数。余白按上, 右, 下, 左的顺时针方向指定。下例会允许左侧和右侧余白：

```
layout.setMargin(new MarginInfo(false, true, false, true));
```

运行结果见图 6.26 “布局的余白”。两种方式产生完全相同的余白效果。

图 6.26. 布局的余白



6.14. 自定义布局

虽然几乎所有的典型的布局都可以使用标准的布局组件来创建，但是有些时候还是需要将代码与布局完全分离。使用 **CustomLayout** 组件，你可以以 XHTML 模板的方式编写自己的布局，由 HTML 模板来决定内含组件的位置。布局模板包含在 Theme 中。这种分离方式允许在代码之外设计布局，这个设计工作可以使用所见即所得(WYSIWYG)的 Web 设计工具，比如 Adobe Dreamweaver。

模板是 HTML 文件，位于 `WebContent/VAADIN/themes/` 下的某个 Theme 文件夹下的 `layouts` 文件夹之下，比如，`WebContent/VAADIN/themes/themename/layouts/mylayout.html`。（注意，存放 Theme 的根路径 `WebContent/VAADIN/themes/` 是固定的。）还可以通过 **InputStream** 来动态地提供模板，详情见下文。模板包含 `<div>` 元素，这个元素带有 `location` 属性，定义了定位标识符。所有的自定义布局 HTML 文件必须使用 UTF-8 编码来保存。

```
<table width="100%" height="100%">
  <tr height="100%">
    <td>
      <table align="center">
        <tr>
          <td align="right">User&nbsp;name:</td>
          <td><div location="username"></div></td>
        </tr>
        <tr>
          <td align="right">Password:</td>
          <td><div location="password"></div></td>
        </tr>
      </table>
    </td>
  </tr>
  <tr>
    <td align="right" colspan="2">
      <div location="okbutton"></div>
    </td>
  </tr>
</table>
```

```

</td>
</tr>
</table>
```

Vaadin 的客户端引擎会将定位元素的内容替换组件的内容。组件与定位元素使用定位标识符来绑定，定位标识符作为 `addComponent()` 方法的参数给定，见下例。

```

// Have a Panel where to put the custom layout.
Panel panel = new Panel("Login");
panel.setSizeUndefined();
main.addComponent(panel);

// Create custom layout from "layoutname.html" template.
CustomLayout custom = new CustomLayout("layoutname");
custom.addStyleName("customlayoutexample");

// Use it as the layout of the Panel.
panel.setContent(custom);

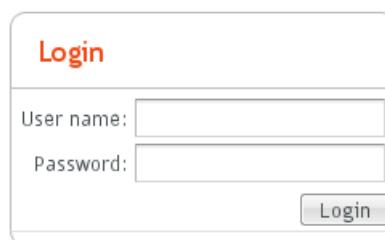
// Create a few components and bind them to the location tags
// in the custom layout.
TextField username = new TextField();
custom.addComponent(username, "username");

TextField password = new TextField();
custom.addComponent(password, "password");

Button ok = new Button("Login");
custom.addComponent(ok, "okbutton");
```

运行结果见图 6.27 “自定义布局组件示例”。

图 6.27. 自定义布局组件示例



如果某个定位标识符上已有组件，你也可以使用 `addComponent()` 来替换掉已有的组件。

除静态模板文件之外，你也可以使用 **CustomLayout** 构造器的另一个版本，这个版本接受 **InputStream** 参数来指定模板来源，通过这种方式可以动态提供模板内容。如下例：

```
new CustomLayout(new ByteArrayInputStream("<b>Template</b>".getBytes()));
```

或者

```
new CustomLayout(new FileInputStream(file));
```


Themes

7.1. 概述	245
7.2. 层级样式表(CSS, Cascading Style Sheets)简介	247
7.3. 优良语法样式表(Sass, Syntactically Awesome Stylesheets)	254
7.4. Theme 的创建和使用	259
7.5. 在 Eclipse 中创建 Theme	262
7.6. Valo Theme	263
7.7. 字体图标	269
7.8. 自定义字体	273
7.9. 条件式 Theme	273

Theme 控制了 Web 应用程序的外观, 本章详细介绍如何使用和创建 *Theme*. *Theme* 使用 Sass 或 CSS 来创建, Sass 是 CSS (Cascading Style Sheets) 的一种扩展. 本章也会介绍 CSS, 尤其是关于如何使用 HTML 元素的 class 来控制样式.

7.1. 概述

Vaadin 使用 *Theme* 将 UI 的外观表现与控制逻辑分离开. *Theme* 可以包含 Sass 或 CSS 样式表, 自定义的 HTML 布局, 以及所需要的任何图片. 在应用程序代码中, *Theme* 资源也可以通过 **ThemeResource** 对象得到.

自定义 theme 放在 Web 应用程序的 VAADIN/themes/ 文件夹下, 在 Eclipse 工程中, Web 应用程序目录在 WebContent 目录下, 在 Maven 工程中, Web 应用程序目录在 src/main/webapp 目录下. 这个目录是固定的 -- VAADIN 文件夹包含静态资源, 静态资源由 Vaadin Servlet 向外提供服务. Servlet 向外提供的资源包括: 这个文件夹下的文件, 以及 Java 类路径中各 JAR 文件内相应的 VAADIN 文件夹下的资源. 比如, 内建的主题保存在 vaadin-themes.jar 内.

图 7.1 “Theme 的内容”展示了 theme 的内容.

图 7.1. Theme 的内容



theme 文件夹的名称决定了 theme 的名称. theme 名称使用在 `@Theme` 注解中, 我们使用这个注解来设置使用的 theme. 一个 theme 必须包含一个名为 `styles.scss` 的 Sass theme 文件, 或者包含一个名为 `styles.css` 的通常 CSS theme 文件, 除此之外的内容可以自由命名. 我们建议将真实的 theme 内容保存为 SCSS 文件, 命名为与 theme 本身同名, 比如 `mytheme.scss`, 以便让文件名称更加清晰.

我们还建议遵循以下命名约定: 图像文件夹命名为 `img`, 自定义布局文件夹命名为 `layouts`, 额外的样式表文件夹命名为 `css`.

定制 theme 需要从一个基础 theme 继承, 详情请参见 第 7.4 节 “Theme 的创建和使用”. 复制并修改一个已有的 theme 也是可以的, 但不推荐这样做, 因为如果修改量较小时, 这种做法会导致较多的维护工作.

使用 theme 的方法是, 在应用程序的 UI 类中通过 `@Theme` 注解来指定 theme, 如下例:

```

@Theme("mytheme")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
    }
}
    
```

theme 中可以包含 UI 组件的多种样式, 可以根据需要来变更样式.

除样式表外, theme 还可以包含用于 **CustomLayout** 自定义样式的 HTML 模板. 详情请参见 第 6.14 节 “自定义布局”.

theme 提供的资源可以使用 **ThemeResource** 类来访问, 详情请参见 第 4.4.4 节 “Theme 资源”. 这种方法可以将 theme 资源用作组件图标, 用在 **Image** 组件内, 以及其他各种用途.

7.2. 层级样式表(CSS, Cascading Style Sheets)简介

层叠样式表(简称 CSS)是将 HTML 表达的 Web 页面内容与它的外观表现分离的基本技术。本节中，我们将介绍 CSS，并解释它与 Vaadin 软件开发的关系。

关于 CSS 的参考资源

本书中我们只能给出简短的介绍，因此我们建议你参考关于 CSS 的大量书籍，以及互联网上的丰富资源。在 W3C 网站你可以找到关于 CSS 的最权威的规格说明，在 Open Directory Project page on CSS 可以找到其他文章，参考手册，教程，当然你也可以在其他网站找到这类资源。

7.2.1. 将 CSS 应用于 HTML

我们来看看以下 HTML 文档，其中包含一些格式化文本的标记元素。Vaadin UI 使用的 HTML 文档与这里的例子本质上是一样的，不过其中使用了一些不同的元素来描绘 UI。

```
<html>
  <head>
    <title>My Page</title>
    <link rel="stylesheet" type="text/css"
          href="mystylesheet.css"/>
  </head>
  <body>
    <p>This is a paragraph</p>
    <p>This is another paragraph</p>
    <table>
      <tr>
        <td>This is a table cell</td>
        <td>This is another table cell</td>
      </tr>
    </table>
  </body>
</html>
```

上例中被粗体表示的 HTML 元素，下文中我们将通过与 CSS 规则匹配来控制它们的样式。

HTML 头部的 `<link>` 元素指定了使用的 CSS 样式表。这段定义由 Vaadin 在负责装载应用程序 UI 的 HTML 页面中自动生成。样式表也可以嵌入在 HTML 文档内，比如，使用 Vaadin TouchKit 来优化页面装载速度时就会如此。

7.2.2. 基本的 CSS 规则

样式表中包含了一组规则，规则可以与页面内的 HTML 相匹配。每一条规则包含一个或多个选择器，以逗号分隔，以及以大括号括起的声明块。声明块包含一系列的属性语句。每个属性中包含标签和值，以冒号分隔。属性语句以分号结尾。

我们来看看一个例子，这段样式表匹配前例中那个简单的 HTML 文档中的元素：

```
p, td {
  color: blue;
}

td {
  background: yellow;
  font-weight: bold;
}
```

`p` 和 `td` 是元素类型选择器，分别匹配到 HTML 中的 `<p>` 和 `<td>` 元素。上例中的第一条规则同时匹配这两种元素，第二条规则只匹配到 `<td>` 元素。假设你将上面的样式表保存为 `mystylesheet.css`，再假设 HTML 文件与它在同一个文件夹下。

图 7.2. 使用元素类型进行简单的样式控制

```
This is a paragraph.  
This is another paragraph.  


|                      |                            |
|----------------------|----------------------------|
| This is a table cell | This is another table cell |
|----------------------|----------------------------|


```

CSS 中的样式继承

CSS 拥有继承能力，下层元素可以继承父元素的属性。比如，我们将上例的样式表修改为以下内容：

```
table {  
    color: blue;  
    background: yellow;  
}
```

`<table>` 元素之内的所有元素都将带有相同的属性。比如，在 `<td>` 元素内的文字将变为蓝色。

HTML 元素类型

HTML 中有很多元素类型，其中每一种都可以接受一组特定的属性。`<div>` 元素是一种通用元素，使用其他特定 HTML 元素类型可以创建的布局和格式，几乎全部都可以使用 `div` 元素来实现。Vaadin 广泛使用 `<div>` 元素来描绘 UI，尤其是在布局管理组件中。

上例中介绍的，使用元素的类型来匹配样式的方法，在 Vaadin 应用程序的样式表中极少用到，甚至可以说几乎没有用到。我们演示上面的例子是因为，这是在使用不同元素来控制文本格式的 HTML 文档中，进行样式控制的通常方式，但这种方式并不适用于 Vaadin UI，因为其中包含的大多是 `<div>` 元素。你不应该使用元素类型，而应该使用元素 class 来匹配，详情见下文。

7.2.3. 使用元素的 `class` 进行匹配

使用 `class` 属性来匹配 HTML 元素是 Vaadin 样式表中最通用的匹配方式。此外，还可以使用 `id` 来匹配一个唯一的 HTML 元素。

HTML 元素的 `class` 通过它的 `class` 属性来指定，如下：

```
<html>  
  <body>  
    <p class="normal">This is the first paragraph</p>  
  
    <p class="another">This is the second paragraph</p>  
  
    <table>  
      <tr>  
        <td class="normal">This is a table cell</td>  
        <td class="another">This is another table cell</td>  
      </tr>  
    </table>
```

```
</body>
</html>
```

HTML 元素的 class 属性可以在 CSS 规则中使用 class 选择器来匹配, class 选择器中, 元素名之后带有一个点号, 点号之后是 class 名. 这种方式使我们可以通过元素的类型和 class 来匹配规则.

```
p.normal {color: red;}
p.another {color: blue;}
td.normal {background: pink;}
td.another {background: yellow;}
```

页面显示结果会是:

图 7.3. 使用 **HTML** 元素的类型和 **Class** 进行匹配

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class
--	---

我们也可以单独使用 class, 方法是对元素名使用通用选择器 *, 比如 *.normal. 通用选择器也可以省略, 因此我们可以使用点号加 class 名, 比如 .normal.

```
.normal {
    color: red;
}

.another {
    background: yellow;
}
```

这时, 规则会与相同 class 的所有元素匹配, 无论元素类型是什么. 运行结果见图 7.4 “只使用 HTML 元素的 class 进行匹配”. 这个例子展示了一种方法, 使得样式表可以保持兼容, 而无论用来描绘组件的具体的 HTML 元素类型是什么.

图 7.4. 只使用 **HTML** 元素的 **class** 进行匹配

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class
--	---

为了确保未来的兼容性, 我们建议你在 CSS 规则中只使用 class 进行匹配, 不要匹配具体的 HTML 元素类型, 因为 Vaadin 的未来版本有可能会改变组件描绘时的具体 HTML 实现. 比如, Vaadin 曾经使用 `<div>` 元素来描绘 **Button** 组件, 但后来改为使用 HTML 中不同种类的 `<button>` 元素. 由于在 CSS 规则中对按钮组件使用了 `v-button` 样式类, 因此样式表的变更非常少.

7.2.4. 使用元素的层级关系进行匹配

CSS 允许使用 HTML 元素的包含关系进行匹配. 比如, 对于以下 HTML 片段:

```
<body>
    <p class="mytext">Here is some text inside a
        paragraph element</p>
    <table class="mytable">
```

```
<tr>
  <td class="mytext">Here is text inside
    a table and inside a td element.</td>
</tr>
</table>
</body>
```

单独使用 class 名 `.mytext` 进行匹配, 将会匹配到 `<p>` 和 `<td>` 组件. 如果我们希望只匹配到 table 的单元格, 我们应该使用以下选择器:

```
.mytable .mytext {color: blue;}
```

要匹配成功, 规则中列出的 class 名不必是前一个 class 的直接后代, 只要是某个层次的后代即可. 比如, 选择器 "`.v-panel .v-button`" 会匹配所有带 `.v-button` class 的元素, 只要它在带有 `.v-panel` class 的某个元素之内.

7.2.5. 层级的重要性

CSS(层级样式表), 就象它的名称暗示的那样, 是关于 层级 的样式表, 也就是说样式表规则的适用遵循它们的来源, 重要度, 范围, 明确度, 以及顺序.

关于 CSS 中层级的确切规则, 请参见 CSS 规格书的 Cascading 小节.

重要度

CSS 的规则定义可以被指定为覆盖其他更高优先度的规则, 方法是为这条规则加上 `!important` 注解. 比如, HTML 元素中使用 `style` 属性指定的样式, 优先度高于 CSS 样式表中的一切规则.

```
<div class="v-button" style="height: 20px;">...
```

你可以使用 `!important` 注解来覆盖高优先度的规则, 如下例:

```
.v-button {height: 30px !important;}
```

明确度

一个规则的选择器如果指定元素的明确度更高, 那么它将覆盖那些明确度较低的规则. 由于 Vaadin theme 中最多使用的是元素 class 选择器, 这种情况下明确度由规则中 class 选择器的数量来决定.

```
.v-button {}
.v-verticallayout .v-button {}
.v-app .v-verticallayout .v-button {}
```

上例中, 最后一条规则的明确度最高, 因此它将匹配成功.

我们前面提到过, HTML 元素的 `style` 属性内定义的样式, 优先度高于 CSS 中的规则, 除非在 CSS 中使用了 `!important` 注解.

关于明确度的计算方式, 详情请参见 CSS3 选择器模块规格说明.

顺序

后出现的 CSS 规则, 优先度高于先出现的规则. 比如, 下例中, 后一条规则将覆盖前一条, 因此颜色将是黑色:

```
.v-button {color: white}
.v-button {color: black}
```

在层级优先度中，明确度的优先级高于顺序，因此，你可以为前一条规则添加明确度，使它的优先度变高，如下例：

```
.v-app .v-button {color: white}
.v-button {color: black}
```

有些情况下注意顺序是很重要的，因为 Vaadin 不保证 CSS 样式表在浏览器中的装载顺序，装载顺序实际上有可能会是随机的，因此导致无法预期的行为。这个问题在 Sass 样式表中不会发生，因为它会被编译为一个单独的样式表文件。对于 CSS 样式表，比如插件或 TouchKit 样式表，顺序是重要的。

7.2.6. Vaadin UI 的样式类层级

下面我们来看看一个真正的 Vaadin UI，这个简单的 Vaadin UI 例子，在 Vertical Layout 之内包含一个 Label 和一个 Button：

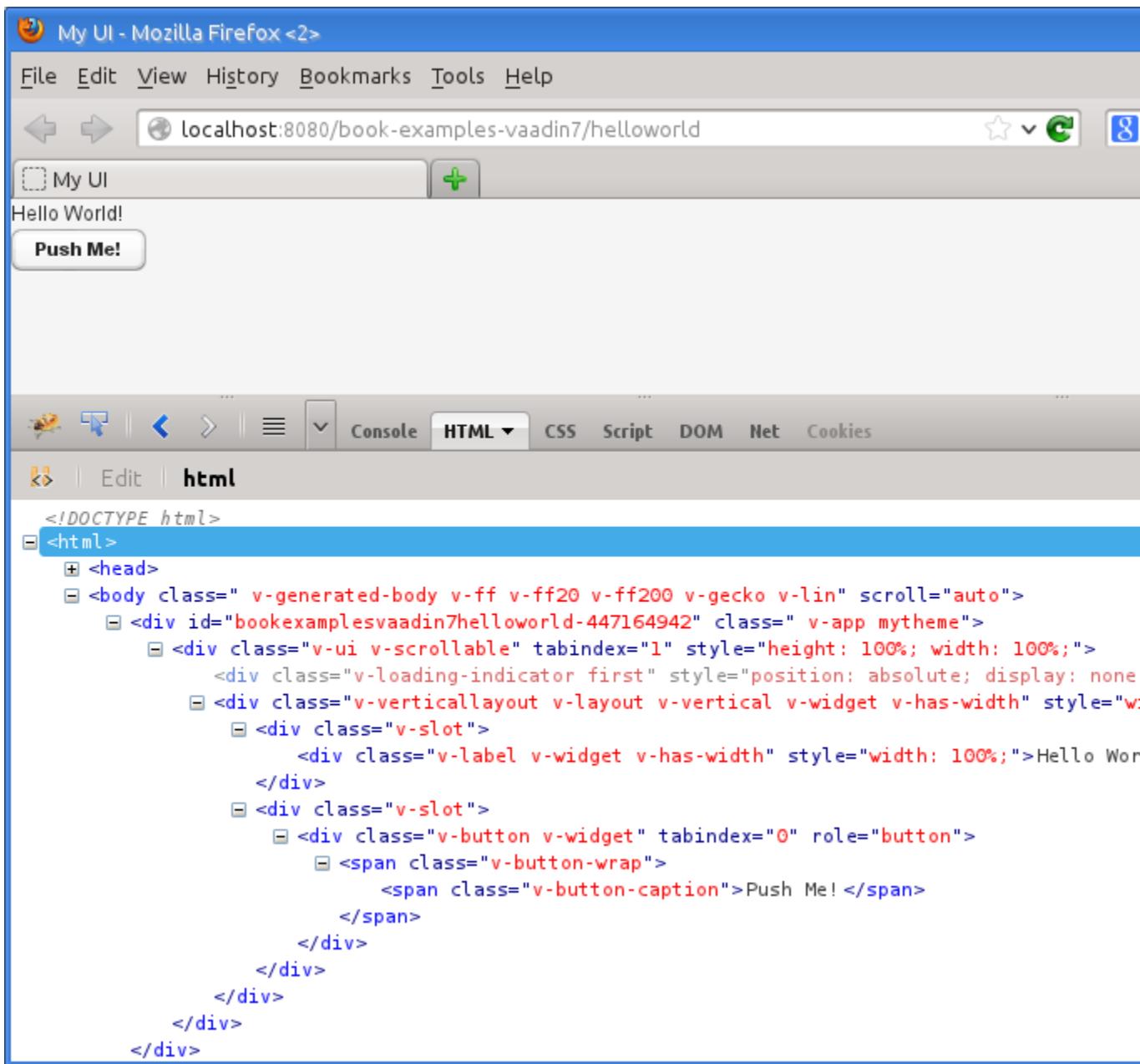
```
// UI has v-ui style class
@Theme("mytheme")
public class HelloWorld extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // VerticalLayout has v-verticallayout style
        VerticalLayout content = new VerticalLayout();
        setContent(content);

        // Label has v-label style
        content.addComponent(new Label("Hello World!"));

        // Button has v-button style
        content.addComponent(new Button("Push Me!",
            new Button.ClickListener() {
                @Override
                public void buttonClick(ClickEvent event) {
                    Notification.show("Pushed!");
                }
            }));
    }
}
```

这个 UI 的默认外观见图 7.5 “一个没有指定 theme 的 Vaadin UI”。使用 Firebug 之类的 HTML 查看器，你可以看到 HTML 元素树，各个元素的 class，以及应用在各个元素上的样式。

图 7.5. 一个没有指定 theme 的 Vaadin UI



下面我们来看看这个 UI 的 HTML 元素 class 结构, 使用 HTML 查看器可以看到以下内容:

```

<body class="v-generated-body v-ff v-ff20 v-ff200 v-gecko v-lin"
      scroll="auto">
  <div id="bookexamplesvaadin7helloworld-447164942"
       class="v-app mytheme">
    <div class="v-ui v-scrollable"
         tabindex="1" style="height: 100%; width: 100%;">
      <div class="v-loading-indicator first"
           style="position: absolute; display: none;"></div>
      <div class="v-verticallayout v-layout v-vertical
               v-widget v-has-width">

```

```
        style="width: 100%;"
<div class="v-slot">
    <div class="v-label v-widget v-has-width"
        style="width: 100%;">Hello World!</div>
</div>
<div class="v-slot">
    <div class="v-button v-widget"
        tabindex="0" role="button">
        <span class="v-button-wrap">
            <span class="v-button-caption">Push Me!</span>
        </span>
    </div>
</div>
</div>
...
</body>
```

下面，看看以下 theme，其中我们设置了各个元素的颜色和余白。这个 theme 是一个 Sass theme。

```
@import "../reindeer/reindeer.scss";

@mixin mytheme {
    @include reindeer;

    /* White background for the entire UI */
    .v-ui {
        background: white;
    }

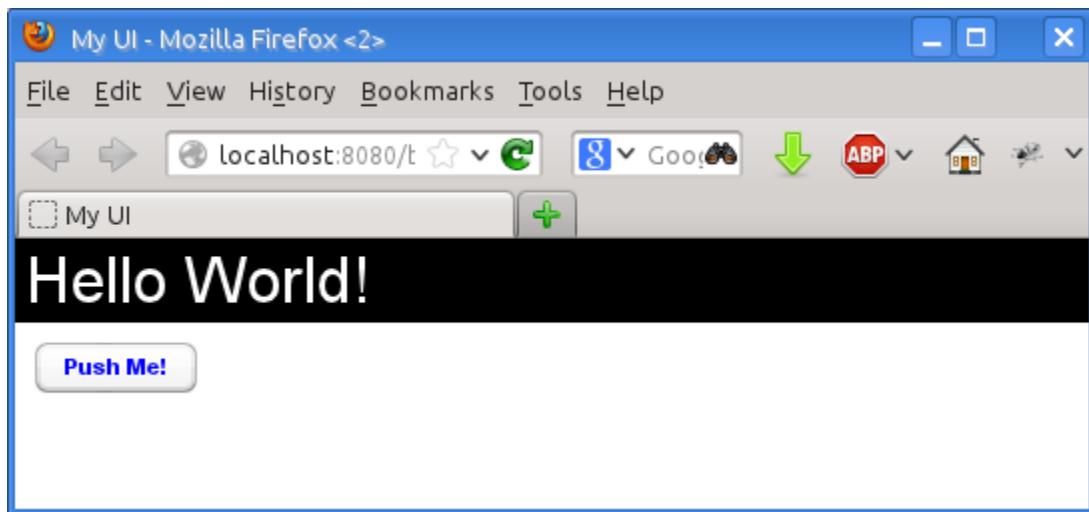
    /* All labels have white text on black background */
    .v-label {
        background: black;
        color: white;
        font-size: 24pt;
        line-height: 24pt;
        padding: 5px;
    }

    /* All buttons have blue caption and some margin */
    .v-button {
        margin: 10px;

        /* A nested selector to increase specificity */
        .v-button-caption {
            color: blue;
        }
    }
}
```

程序的外观变化见图 7.6 “指定 theme 后的 Vaadin UI”。

图 7.6. 指定 theme 后的 Vaadin UI



一个元素可以带有多个 class, 以空格分隔. 对于多个 class 的情况, CSS 规则只需要匹配其中的一个 class 就可以与这个元素匹配. Vaadin 的很多组件都使用了这个功能, 以便根据组件的不同状态来匹配规则. 比如, 当鼠标指针移动到 **Link** 组件上方时, over class 会被添加到组件上. 这样的样式大多是 Google Web Toolkit 中的功能.

7.2.7. 关于兼容性的注意事项

CSS 是一个不断开发中的标准. 它初次提出是在 1994 年. CSS 规格由 World Wide Web 协会(W3C)的 CSS 工作组负责维护. 它的版本以向下兼容的 "级别(Level)" 来标识, CSS Level 1 于 1996 年公布, Level 2 于 1998 年公布, 正在开发中的 CSS Level 3 开始于 1998 年. CSS3 分为几个不同的模块, 各模块分别独立开发, 很多模块已经达到了 Level 4.

至少从 1995 年起, 在所有的图形化 Web 浏览器中都支持了 CSS, 但是这种支持有时是很不完整的, 而且在各种浏览器之间还存在着大量的不兼容, 其数量之多只能以"不幸"来形容. 在 Vaadin 内建的 theme 中, 我们已经努力重视这些不兼容性, 但是你在开发自己的 theme 时还是需要考虑这些问题. 兼容性问题的详细介绍, 请参照 CSS 相关书籍.

7.3. 优良语法样式表(Sass, Syntactically Awesome Stylesheets)

Vaadin 的样式表使用 Sass. Sass 是 CSS3 的扩展, 在 CSS 的基础上增加了规则嵌套, 变量, mixin, 选择器继承, 以及其他功能. Sass 支持两种格式: Vaadin theme 使用 SCSS (.scss) 格式, 这个格式是 CSS3 的超集, 另外 Sass 也支持更简洁的缩进格式 (.sass).

在 Vaadin 应用程序中 Sass 可以通过两种基本的形式来使用, 可以将 SCSS 文件预编译为 CSS, 也可以实时编译. 当 Vaadin Servlet 的开发模式启动时就可以使用后一种方式, 详情请参见第 4.8.6 节 "Servlet 的其他配置参数".

7.3.1. Sass 概述

变量

Sass 允许定义变量, 并在规则中使用这些变量.

```
$textcolor: blue;
```

```
.v-button-caption {
  color: $textcolor;
}
```

上述规则编译为 CSS 的结果是:

```
.v-button-caption {
  color: blue;
}
```

mixin 也可以使用变量做为参数, 详情见后文.

嵌套

Sass 支持规则嵌套, 嵌套的规则会被编译为内部选择器(internal-selector). 比如:

```
.v-app {
  background: yellow;

  .mybutton {
    font-style: italic;

    .v-button-caption {
      color: blue;
    }
  }
}
```

编译结果为:

```
.v-app {
  background: yellow;
}

.v-app .mybutton {
  font-style: italic;
}

.v-app .mybutton .v-button-caption {
  color: blue;
}
```

Mixin

Mixin 是一种规则模板, 可以被其他规则引用. mixin 的定义以 @mixin 关键字作为前缀, 后面是 mixin 名称. 然后可以在其他规则中使用 @include 来引用这个 mixin. 也可以向 mixin 传递参数, 参数被 mixin 当作局部变量一样处理.

比如:

```
@mixin mymixin {
  background: yellow;
}

@mixin othermixin($param) {
  margin: $param;
}

.v-button-caption {
```

```

@include my mixin;
@include other mixin(10px);
}

```

上例的 SCSS 将被翻译为以下 CSS:

```

.v-button-caption {
  background: yellow;
  margin: 10px;
}

```

在 mixin 中也可以使用嵌套规则, 这两种功能的组合非常强大. 扩展 Vaadin theme 时常常常用到规则的 mixin, 详情请参见第 7.4.1 节 “Sass Theme”.

Vaadin theme 也定义为 mixin 的形式, 以便灵活使用, 比如可以在 portal 内对不同的 portlet 使用不同的 theme.

7.3.2. Vaadin 中的 Sass

我们在这里不会深入介绍 Sass, 而是向你推荐它的一份优秀文档: <http://sass-lang.com/>. 下面, 我们非常简单地介绍在 Vaadin 中如何使用 Sass.

你可以使用 Eclipse plugin 来创建基于 Sass 的新 theme, 详情请参见第 7.5 节 “在 Eclipse 中创建 Theme”.

7.3.3. 编译 Sass Theme

实时编译

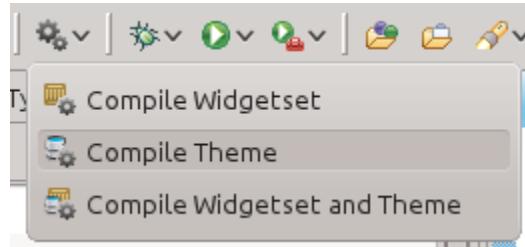
使用 Sass theme 的最简单方法是让 Vaadin servlet 在运行时编译它. 这种情况下, SCSS 源文件放置在 theme 文件夹下, 每当向服务器请求 styles.css 时都会进行编译.

实时编译会消耗一些时间, 因此它只在 Vaadin servlet 处于开发模式时有效, 详情请参见第 4.8.6 节 “Servlet 的其他配置参数”. 而且, 这种方式还要求 theme 编译器及其依赖的包放置在 servlet 的类路径下. 对于生产环境, 你应该将 theme 预编译为 CSS, 详情见后文.

在 Eclipse 中编译

如果你使用 Eclipse 和 Vaadin Plugin for Eclipse, 这个 Plugin 的工程向导会创建一个 Sass theme. 这个 Plugin 还会在工具条中包括 **Compile Theme** 命令, 可以将编译工程的 theme 编译为 CSS. 工具条上的另一个命令还可以编译 widget 群.

图 7.7. 编译 Sass Theme



WebContent/VAADIN/mytheme/styles.scss 文件, 以及由它所包含的其他所有 Sass 源文件, 都会被编译成为 styles.css 文件.

使用 Maven 编译

要使用 Maven 来编译 theme, 你需要引用内建的 theme 作为依赖项目:

```
...
<dependencies>
  ...
  <dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-themes</artifactId>
    <version>${vaadin.version}</version>
  </dependency>
</dependencies>
...
```

以上设置至少对 Vaadin 应用程序的 vaadin-archetype-application archetype 会自动引入. 进行实际的 theme 编译工作, 最方便的办法是使用 Vaadin Maven Plugin 执行 update-theme 和 compile-theme 目标.

```
...
<plugin>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-maven-plugin</artifactId>
  ...
  <executions>
    <execution>
      ...
      <goals>
        <goal>clean</goal>
        <goal>resources</goal>
        <goal>update-theme</goal>
        <goal>update-widgetset</goal>
        <goal>compile-theme</goal>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>

```

完成以上设定后, theme 的编译会成为相关动作, 比如 package, 的一部分.

mvn package

你也可以只编译 theme, 方法是使用 compile-theme 目标:

mvn vaadin:compile-theme

在命令行中编译

你可以使用 Vaadin Sass 编译器或标准编译器, 将 Sass 样式表编译为 CSS, 编译的输出将是自定义 theme 的 styles.css 文件. 如果发布前已经编译过 CSS, 那么运行时 Sass 源文件不必保存在 theme 文件夹下.

你可以在 theme 文件夹内运行 Vaadin Sass 编译器, 如下:

```
java -cp '.../.../WEB-INF/lib/*' com.vaadin.sass.SassCompiler styles.scss
styles.css
```

-cp 参数应该指向 Vaadin Sass 编译器和 JAR 文件所在的类路径. 上例中, 我们假设这些文件在 Web 应用程序的 WEB-INF/lib 文件夹之下. 如果使用 Ivy 装载了 Vaadin 库, (使用 Vaadin Plugin

for Eclipse 创建的工程通常都是如此), Vaadin 库文件会保存在 Ivy 的本地仓库中. 它的目录结构略为散乱, 因此我们建议你将需要的库文件保存到一个单独的目录中. 我们建议使用 Ant 脚本, 详见后文.

使用 Ant 编译

使用 Apache Ant, 你可以便利地利用 Ivy 来解析库依赖关系, 并利用 Vaadin 的 Theme Compiler 来编译 theme. 这个编译步骤可以很容易的包含在 WAR 编译脚本之内.

首先, 编译脚本配置如下:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
    name="My Project" basedir="../"
    default="package-war">

    <target name="configure">
        <!-- Where project source files are located -->
        <property name="src-location" value="src" />

        ... other project build definitions ...

        <!-- Name of the theme -->
        <property name="theme" value="book-examples"/>

        <!-- Compilation result directory -->
        <property name="result" value="build/result"/>
    </target>

    <!-- Initialize build -->
    <target name="init" depends="configure">
        <!-- Construct and check classpath -->
        <path id="compile.classpath">
            <!-- Source code to be compiled -->
            <pathelment path="${src-location}" />

            <!-- Vaadin libraries and dependencies -->
            <fileset dir="${result}/lib">
                <include name="*.jar"/>
            </fileset>
        </path>

        <mkdir dir="${result}"/>
    </target>
```

你首先需要将依赖的所有 Vaadin 库取到一个单独的目录下, 然后在发布过程中使用这些库, 当然也可以在 theme 编译中使用.

```
<target name="resolve" depends="init">
    <ivy:retrieve
        pattern="${result}/lib/[module]-[type]-[artifact]-[revision].[ext]"/>
</target>
```

然后编译 theme, 如下:

```
<!-- Compile theme -->
<target name="compile-theme"
    depends="init, resolve">
    <delete dir="${result}/VAADIN/themes/${theme}"/>
    <mkdir dir="${result}/VAADIN/themes/${theme}"/>
```

```

<java classname="com.vaadin.sass.SassCompiler"
      fork="true">
<classpath>
  <path refid="compile.classpath"/>
</classpath>
<arg value="WebContent/VAADIN/themes/${theme}/styles.scss"/>
<arg value="${result}/VAADIN/themes/${theme}/styles.css"/>
</java>

<!-- Copy theme resources -->
<copy todir="${result}/VAADIN/themes/${theme}">
  <fileset dir="WebContent/VAADIN/themes/${theme}">
    <exclude name="**/*.scss"/>
  </fileset>
</copy>
</target>
</project>

```

7.4. Theme 的创建和使用

自定义 theme 位于 Web 应用程序的 VAADIN/themes 文件夹下, Web 应用程序目录, 在 Eclipse 工程中是 WebContent, 在 Maven 工程中是 src/main/webapp, 参见图 7.1 “Theme 的内容”. 这个路径是固定的. 对于应用程序中使用到的每一个 theme, 都需要有一个 theme 文件夹, 当然应用程序很少用到多个 theme.

7.4.1. Sass Theme

在 Vaadin 中使用 Sass theme 的方式有两种, 你可以自行将它编译为 CSS, 也可以使用实时编译模式, 在浏览器请求 theme CSS 时由 Vaadin Servlet 进行编译, 详情请参见第 7.3.3 节 “编译 Sass Theme”.

要创建一个名称为 mytheme 的 Sass theme, 你必须将 styles.scss 文件放在 theme 文件夹 VAADIN/themes/mytheme 之下. 如果文件夹下不存在 styles.css 文件, 那么在浏览器请求 theme 时, Sass 源文件会被实时编译.

我们建议将 theme 组织为至少两个 SCSS 文件, 其中一个是 styles.scss, 由它负责 import 另一个文件, 另一个是真实的 theme 文件, 文件名可以更清楚一些, 这样在编辑器中我们可以更容易区分. Vaadin Plugin for Eclipse 创建新 theme 时使用的也是这样的组织方式.

如果你使用一个包含 theme 的 Vaadin add-ons, Vaadin Plugin for Eclipse 和 Maven 会将其中的 theme 自动添加到 addons.scss 文件中.

Theme SCSS

我们建议 theme 中的规则应该使用一个 theme 名称选择器做前缀. 在 Sass 中实现这个前缀的方法是, 将规则嵌套在另一个规则之内, 外部的规则使用 theme 名称选择器.

Theme 以 Sass 的 mixin 的形式定义, 所以当然你 import 了 mixin 之后, 你可以在 theme 规则中 @include 它, 如下:

```

@import "addons.scss";
@import "mytheme.scss";

.mytheme {
  @include addons;
}

```

```
@include mytheme;  
}
```

但是, 上面这种做法主要是用于在 portlet 中使用 UI 的情况, 这时每个 portlet UI 都需要不同的 theme, 或者在某些特殊场合, theme 中的 rule 使用空的父选择器 & 来引用 theme 名称.

其他情况下, 你可以不必使用嵌套的 theme 选择器也是安全的, 如下:

```
@import "addons.scss";  
@import "mytheme.scss";  
  
@include addons;  
@include mytheme;
```

实际的 theme 应该定义为一个 mixin, 其中 include 它所扩展的基础 theme, 如下.

```
@import "../reindeer/reindeer.scss";  
  
@mixin mytheme {  
    @include reindeer;  
  
    /* An actual theme rule */  
    .v-button {  
        color: blue;  
    }  
}
```

7.4.2. 旧式 CSS Theme

除 Sass theme 外, 你也可以创建旧式的 CSS theme. CSS theme 的限制比 Sass 要多 - 一个应用程序只能带有一个 CSS theme, 而 Sass theme 可以有多个.

CSS theme 定义在 VAADIN/themes/mytheme 文件夹下的 styles.css 文件中. 你需要 import 内建 theme 的 legacy-styles.css 文件, 如下:

```
@import "../reindeer/legacy-styles.css";  
  
.v-app {  
    background: yellow;  
}
```

7.4.3. 控制标准组件的样式

Vaadin 中的每个 UI 组件都带有一个 CSS 样式 class, 你可以用它来控制组件的外观表现. 很多组件还拥有额外的子元素, 这些子元素也可以控制样式. 你可以使用 addStyleName() 方法来为组件添加一些与上下文相关的样式名称. 注意 getStyleName() 方法只返回自定义的样式名, 不包含组件的内建样式名.

关于各组件的样式, 请阅读各个组件的对应章节. 这些样式名大多由各组件的客户端 widget 决定. 查看各 HTML 元素的样式, 最简单的方法是使用 HTML 查看器, 比如 FireBug.

某些客户端组件, 或某些组件样式, 可以在不同的服务器端组件之间共用. 比如, v-textfield 样式, 除 **TextField** 之外, 还被应用到所有组件内部的文字输入框上.

7.4.4. 内建 Theme

Vaadin 目前包含以下内建 theme:

- valo, 从 Vaadin 7.3 版开始的主要 theme
- reindeer, Vaadin 6 和 7 的主要 theme
- chameleon, 一个易于定制的 theme
- runo, IT Mill Toolkit 5 的默认 theme
- liferay, 用于 Liferay portlet

除此之外, 还有 base theme, 这个 theme 不应该直接使用, 但其他内建 built-in theme (valo 除外) 都从它扩展而来.

内建 theme 由对应的 vaadin-themes JAR 文件内的 VAADIN/themes/<theme>/styles.scss 样式表提供. 其中也包含预编译的 CSS 文件, 以便你直接使用这些 theme.

与内建 theme 相关的一些常数定义在 com.vaadin.ui.themes 包内的 theme 类中. 这些常数大多是特定组件的特定样式名.

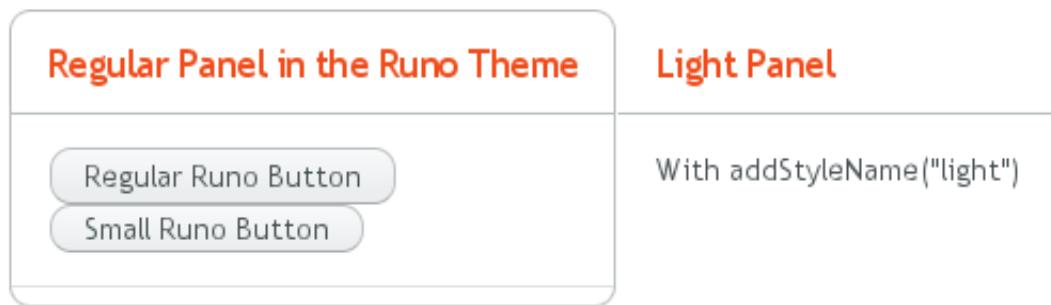
```
@Theme("runo")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
        Panel panel = new Panel("Regular Panel in the Runo Theme");
        panel.addComponent(new Button("Regular Runo Button"));

        // A button with the "small" style
        Button smallButton = new Button("Small Runo Button");
        smallButton.addStyleName(Runo.BUTTON_SMALL);

        Panel lightPanel = new Panel("Light Panel");
        lightPanel.addStyleName(Runo.PANEL_LIGHT);
        lightPanel.addComponent(
            new Label("With addStyleName(\"light\")"));
        ...
    }
}
```

上例使用 Runo theme, 运行结果见 图 7.8 “Runo Theme”.

图 7.8. Runo Theme



这个内建 theme 附带了一个定制的图标字体, FontAwesome, 用在这个 theme 的图标中, 你也可以使用它作为字体图标, 详情请参见 第 7.7 节 “字体图标”.



将内建 Theme 作为静态资源向外提供

Vaadin 库 JAR 文件中的 theme 将由 Servlet 动态地向外提供. 如果让 Web 服务器将 theme 和 widget 群以静态资源的形式向外提供, 性能会更高一些. 为了达到这个目的, 你需要从 JAR 文件中展开 VAADIN/ 目录, 放到 Web 内容目录下(在 Eclipse 工程中是 WebContent 目录, 在 Maven 工程中是 src/main/webapp 目录).

```
$ cd WebContent
$ unzip path-to/vaadin-server-7.x.x.jar 'VAADIN/*'
$ unzip path-to/vaadin-themes-7.x.x.jar 'VAADIN/*'
$ unzip path-to/vaadin-client-compiled-7.x.x.jar 'VAADIN/*'
```

你也可以使用前端的缓存服务器来向外提供静态内容, 这样可以降低应用程序服务器的负载. 在 portal 中, 你可以将 theme 安装到全局位置, 详情请参见 第 12.3.5 节 “安装 Vaadin 资源”.

当你升级到 Vaadin 的新版本时, 一定要确保静态内容也升级到新版.

关于如何为自定义的 GWT widget 创建默认 theme, 请参见 第 16.8 节 “Widget 的样式控制”.

7.4.5. Add-on 的 Theme

你可以在 Vaadin Directory 找到很多 add-on 形式的 theme. 此外, 很多 add-on 组件也包含了自己 theme.

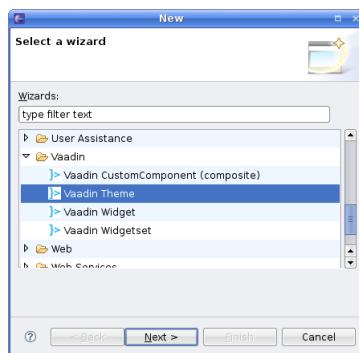
add-on theme 需要被 includ 到工程的 theme 中. Vaadin Plugin for Eclipse 和 Maven 会自动在工程的 theme 文件夹下的 addons.scss 文件中 include 这些 theme. 这个文件会被 include 到工程的 theme 中.

7.5. 在 Eclipse 中创建 Theme

The Eclipse plugin 会为新的 Vaadin 工程自动创建 theme 的基础代码. 它还带有一个向导, 用于创建新的自定义 theme. 按以下步骤可以创建新的 theme.

1. 在主菜单中, 选择菜单项 **File → New → Other...**, 或者右键单击 **Project Explorer**, 然后选择菜单项 **New → Other....** 会出现一个新窗口.

2. 在 **Select a wizard** 窗口中, 选择 **Vaadin → Vaadin Theme** 向导.



点击 **Next >** 按钮, 进行到下一步.

3. 在 **Create a new Vaadin theme** 窗口, 包括以下设定项:

Project (必须项)

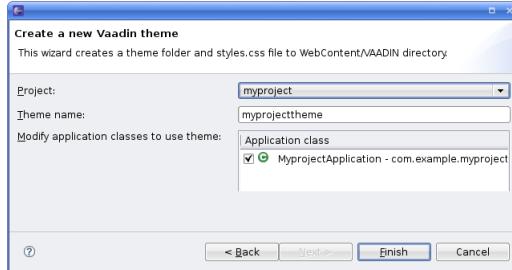
theme 应该在哪个工程内创建.

Theme name (必须项)

theme 名称, 用作 theme 文件夹名, 也用在 CSS 标签之内(附带 "v-theme-" 前缀), 因此名称必须是一个合法的标识符. 其中只允许使用拉丁字母, 下划线, 减号.

Modify application classes to use theme (可选项)

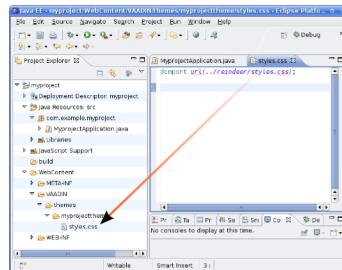
这个设定项允许向导在选定的应用程序 UI 类的构造函数中添加一条语句, 这条语句负责启用 theme. 如果你需要使用动态逻辑来控制 theme, 你可以不选中这个设定项, 或者之后手工修改向导产生的代码.



点击 **Finish** 按钮, 创建 theme.

向导会在 WebContent/VAADIN/themes 目录之下创建 theme 文件夹, 以及实际的样式表文件 mytheme.scss 和 styles.scss, 见图 7.9 “新创建的 Theme”.

图 7.9. 新创建的 Theme



新创建的 theme 使用 @import 语句扩展一个内建的基础 theme. 关于 theme 继承的讲解, 请参见第 7.4 节 “Theme 的创建和使用”. 注意 reindeer theme 不在 widgetsets 文件夹下, 而是在 Vaadin JAR 文件内. 关于向外提供内建的 theme, 详情请参见第 7.4.4 节 “内建 Theme”.

在创建 theme 的向导中, 如果你为设定项 **Modify application classes to use theme** 选择了 UI 类, 向导会为这个 UI 类添加 @Theme 注解.

如果之后你在 Eclipse 中修改了 theme 的名称, 要注意, 修改 theme 文件夹的名称不会自动地修改相应的 @Theme 注解的内容. 凡引用到 theme 名称的地方, 你需要手工修改.

7.6. Valo Theme

Valo 是芬兰语中“光”的意思. Valo theme 用它自己的理念将光的运用组合起来, 它使用这套理论来决定如何处理阴影和高亮. 它创建了线条, 边框, 高亮度, 与背景色协调的阴影, 一切都给人视觉感

受上的愉悦。Valo 使用一种色彩算法理论来计算辅助色，将辅助色柔和地融合到背景之中。除静态艺术之外，还辅以交互式的动画效果。

Valo 的真正力量在于它的可配置能力，这种可配置能力是通过参数，函数，以及 Sass mixin 实现的。在你自己的 theme 中，可以使用 Valo 中内建的预定义值，也可以覆盖这些默认值。关于可用的 mixin，函数，以及变量，详细文档请参见 Valo API 文档，地址是 <http://vaadin.com/valo>。

7.6.1. 基本使用

Valo 的使用方法与其他 theme 一样。它的可选参数必须在 @import 语句之前给出。

```
// Modify the base color of the theme
$v-background-color: hsl(200, 50%, 50%);

// Import valo after setting the parameters
@import "../valo/valo";

.mythemename {
  @include valo;

  // Your theme's rules go here
}
```

如果你需要覆盖 valo theme 中定义的 mixin 或函数，你必须在 import 语句之后做，但又在 include valo 的 mixin 之前。此外，通过某些配置参数，你可以使用 theme 中定义的变量。这种情况下，需要在 import 语句之后覆盖这些变量。

7.6.2. 共通设定

下面我们来介绍可选参数，用于控制 Valo theme 的视觉表现。除这里给出的参数之外，组件样式也有它们自己的参数，详情在各组件的章节中介绍。

一般设定

`$v-background-color (默认值: hsl(210, 0%, 98%))`

背景色是 Valo theme 中的主要控制参数，它被用来计算 theme 中的所有其他颜色。如果背景色是暗色（亮度较低，low luminance），那么 theme 会自动使用较亮的前景色，实现前景与背景之间的高对比度。

你可以通过 CSS 所支持的任何方式来指定背景色：可以是 16 进制的 RGB 颜色编码，可以是通过 `rgb()` 或 `rgba()` 指定的 RGB/A 值，也可以是通过 `hsl()` 或 `hsla()` 指定的 HSL/A 值。也可以使用颜色的名称，但应该尽量避免这种方式，因为目前还只支持 CSS 颜色名称中的一部分。

`$v-app-background-color (默认值: $v-background-color)`

UI 根元素的背景色。你可以使用 CSS 所支持的任何方式来指定这个背景色。

`$v-app-loading-text (默认值: "")`

客户端引擎装载和启动过程中，显示在“装载中...”动画下方的一段静态文字。指定这段文字时必须用引号括起。这段文字目前还不支持本地化。

`$v-app-loading-text: "Loading Resources...";`

`$v-line-height (默认值: 1.55)`

所有 widget 共通的基本行高设定。必须指定为一个无单位的数值。

`$v-line-height: 1.6;`

字体设定

`$v-font-size (默认值: 16px)`
基本字体大小. 使用像素单位指定.

```
$v-font-size: 18px;
```

`$v-font-weight (默认值: 300)`
通常字体的粗细. 应该以数值形式指定, 而不是符号.

```
$v-font-size: 400;
```

`$v-font-color (默认值: computed)`
前景文字颜色, 可以指定为任意的 CSS 颜色值. 默认是通过背景颜色计算的, 因此可以形成与背景较高的对比度.

`$v-font-family (默认值: "Open Sans", sans-serif)`
字体家族(Font family), 以及替代字体, 以逗号分隔的列表形式指定, 如果字体名包含空格, 则必须使用引号括起. 默认字体 "Open Sans" 是包含在 Valo theme 中的一种 Web 字体. 其他被用到的 Valo 字体必须在这个列表中指定, 才可以启用它们. 详情请参见第 7.6.4 节 "Valo 的字体".

```
$v-font-family: "Source Sans Pro", sans-serif;
```

`$v-caption-font-size (默认值: round($v-font-size * 0.9))`
组件标题的字体大小. 使用像素单位指定.

`$v-caption-font-weight (默认值: max(400, $v-font-weight))`
标题的字体粗细. 应该以数值形式指定, 而不是符号.

布局设定

`$v-unit-size (默认值: round(2.3 * $v-font-size))`
这个参数是各种布局尺寸计算中的基本尺寸. 在某些尺寸计算中会直接使用这个参数, 比如按钮高度, 布局余白, 而其他尺寸计算可能会间接使用到它. 这个参数的值必须以像素单位指定, 适当的范围应该在 18 到 50 之间.

```
$v-unit-size: 40px;
```

`$v-layout-margin-top, $v-layout-margin-right, $v-layout-margin-bottom, $v-layout-margin-left (默认值: $v-unit-size)`
所有内建布局组件的余白大小. 使用 `setMargin()` 方法可以使余白有效, 详情请参见第 6.13.5 节 "布局的余白".

`$v-layout-spacing-vertical 和 $v-layout-spacing-horizontal (默认值: round($v-unit-size/3))`
垂直或水平间隔空白的大小. 使用 `setSpacing()` 方法可以使布局的间隔空白有效, 详情请参见第 6.13.4 节 "布局单元格的间隔空白".

组件功能

以下设定应用于某些 UI 组件的各种可视化功能.

`$v-border`(默认值: `1px solid (v-shade 0.7)`)

这个参数指定边框规格, 适用于有边框线的组件. 边框的粗细必须使用像素单位指定. 边框的颜色, 可以使用 CSS 支持的任何颜色, 或使用 `v-tint`, `v-shade`, 和 `v-tone` 关键字之一, 关于这几个关键字, 详情将在本节后续内容中介绍.

`$v-border-radius`(默认值: `4px`)

边框线圆角的半径大小, 适用于有边框线的组件. 必须使用像素单位指定.

```
$v-border-radius: 8px;
```

`$v-gradient`(默认值: `v-linear 8%`)

颜色梯度的样式名称, 适用于有颜色梯度的组件. 颜色梯度样式可以使用以下关键字: `v-linear` 和 `v-linear-reverse`. 不透明度必须指定为 0% 到 100% 之间的百分比值.

```
$v-gradient: v-linear 20%;
```

`$v-bevel`(默认值: `inset 0 1px 0 v-tint, inset 0 -1px 0 v-shade`)

插入式阴影的样式, 用于指定某些组件在背景中的"凸起"效果. 这个样式值使用 CSS `box-shadow` 的语法, 应该是一系列高亮和阴影的 `inset` 值. 斜面部分的颜色, 可以使用 CSS 支持的任何颜色值, 或 `v-tint`, `v-shade`, 和 `v-tone` 关键字之一, 关于这些关键字, 详情将在本节后续内容中介绍.

`$v-bevel-depth`(默认值: `30%`)

指定斜面阴影的"深度", 用来控制斜面样式的颜色. `tint`, `shade`, 和 `tone` 的实际颜色, 都是由这个深度值计算得到的.

`$v-shadow`(默认值: `0 2px 3px v-shade`)

指定所有组件的默认阴影样式. 和 `$v-bevel` 一样, 这个样式的值使用 CSS `box-shadow` 的语法, 但没有 `inset` 部分. 阴影的颜色, 可以使用 CSS 支持的任何颜色, 或 `v-tint`, `v-shade` 关键字之一, 关于这些关键字, 详情将在本节的后续内容中介绍.

`$v-shadow-opacity`(默认值: `5%`)

指定阴影的不透明度, 用于控制阴影样式的颜色. `tint` 和 `shade` 的实际颜色都是由这个参数计算得到.

`$v-focus-style`(默认值: `0 0 0 2px rgba($v-focus-color, .5)`)

用作 `Field` 组件聚焦状态指示器的阴影样式. 这个值由空格分隔成几个部分: 水平阴影位置(像素单位), 垂直阴影位置(像素单位), 模糊距离(像素单位), 扩散距离(像素单位), 以及颜色. 颜色可以使用 CSS 支持的任何颜色. 你也可以只指定颜色, 这时阴影位置将使用默认值. 可以使用 `rgba()` 和 `hsla()` 来实现透明色.

比如, 以下代码将在 `Field` 组件周围创建一个 2 像素宽的橙色外框:

```
$v-focus-style: 0 0 0 2px orange;
```

`$v-focus-color`(默认值: `valo-focus-color()`)

`Field` 组件聚焦状态指示器的颜色. `valo-focus-color()` 函数会计算一个与当前环境颜色呈高对比度的颜色, 所谓当前环境颜色, 通常就是背景色. 这个颜色值可以使用 CSS 支持的任何颜色.

`$v-animations-enabled`(默认值: `true`)

指定是否使用各种 CSS 动画效果.

`$v-hover-styles-enabled`(默认值: `true`)

指定是否使用各种 `:hover` 样式来指示鼠标移动到元素之上的状态.

`$v-disabled-opacity` (默认值: 0.5)

被禁用组件的不透明度, 详情请参见第 5.3.3 节“激活与禁用”.

`$v-selection-color` (默认值: `$v-focus-color`)

选择组件中用于标识被选中项目的颜色.

`$v-default-field-width` (默认值: `$v-unit-size * 5`)

Field 组件的默认宽度, 可以通过 `setWidth()` 方法来覆盖默认设定.

`$v-error-indicator-color` (默认值: `#ed473b`)

组件的错误指示器的颜色. 详情请参见第 4.5.1 节“错误指示器和消息”.

`$v-required-field-indicator-color` (默认值: `$v-error-indicator-color`)

Field 组件的必须项目指示器的颜色, 详情请参见第 5.4.1 节“**Field** 接口”.

`$v-border`, `$v-bevel`, 以及 `$v-shadow` 中可以使用的颜色值, 除 CSS 支持的颜色之外, 还包括以下关键字:

`v-tint`

比背景色更亮.

`v-shade`

比背景色更暗.

`v-tone`

自适应颜色: 在亮背景中较暗, 在暗背景中为较亮. 这个值不能用在 `$v-shadow` 中.

比如:

```
$v-border: 1px solid v-shade;
```

你可以在括号内指定一个权重参数, 来精确调整对比度:

```
$v-border: 1px solid (v-tint 2);
```

```
$v-border: 1px solid (v-tone 0.5);
```

Theme 的编译与优化

`$v-relative-paths` (默认值: `true`)

这个标记指定 URL 相对路径是相对于当前解析总的 SCSS 文件, 还是相对于被编译的根文件, 以便保证对不同的资源来说路径都是正确的. Vaadin theme 编译器解析 URL 路径的方式不同于通常的 Sass 编译器(Vaadin 会修改 URL 相对路径). 在 Ruby 编译器中应该使用 `false`, 对 Vaadin Sass 编译器应该使用 `true`.

`$v-included-components` (默认值: component list)

Theme 优化参数, 指定 include 的组件 theme, 详情请参见第 7.6.6 节“Theme 优化”.

`$v-included-additional-styles` (默认值: `$v-included-components`)

Theme 优化参数, 指定哪些组件应该包含额外的组件样式名. 详情请参见第 7.6.5 节“组件样式控制” for more details.

7.6.3. Valo 中的 Mixin 和函数

Valo 大量使用了 Sass 的 mixin 和函数来计算 theme 的各种功能, 比如颜色和阴影. 而且, 所有的组件样式都是 mixin. 你可以使用内建的 mixin, 也可以覆盖它们. 关于 mixin 和函数的详细文档, 请参照 Valo API 文档: <http://vaadin.com/valo/api>.

7.6.4. Valo 的字体

Valo 包含以下自定义字体:

- Open Sans
- Source Sans Pro
- Roboto
- Lato
- Lora

对于 Valo theme, 使用的字体必须通过 `$v-font-family` 参数来指定, 参数中可以指定多个字体, 按优先度顺序排列. 字体家族中的字体在使用时按优先度递减的顺序逐个选择, 如果高优先度的字体在浏览器中不可用, 就以低优先度的字体来替代. 你可以指定任意字体家族, 再加上浏览器可能支持的通用字体家族(generic family). 除主字体家族外, 在你的应用程序中也可以使用其他字体家族. 为了使用 Valo 中包含的字体, 你需要在这个变量中列出需要的字体.

```
$v-font-family: 'Open Sans', sans-serif, 'Source Sans Pro';
```

上例中, 我们指定 Open Sans 为最优先的主字体, 并以浏览器支持的任何一种 sans-serif 字体作为后备. 此外, 我们还 include 了 Source Sans Pro 作为辅助字体, 以便在自定义规则中使用它, 如下:

```
.v-label pre {
    font-family: 'Source Sans Pro', monospace;
}
```

上例中的规则, 会在所有内容模式为 PREFORMATTED 的 **Label** 组件中使用这个字体.

7.6.5. 组件样式控制

很多组件都有一些独有的样式, 可以用来控制这些组件变得更大, 更小等等. 你可以使用 `addStyleName()` 方法来指定组件样式, 使用的样式常数定义在 **ValoTheme** 枚举型中.

```
table.addStyleName(ValoTheme.TABLE_COMPACT);
```

关于各组件样式的最新的完整列表, 请参照 Vaadin API 文档中的 **ValoTheme** 枚举类型. 其中一部分在组件样式的章节中也有介绍.

禁用组件样式

组件样式是可选的, 但默认都是启用的. 也可以使用 `$v-included-additional-styles` 参数, 决定对哪些组件启用. 这个参数的默认值是 `$v-included-components`, 而且 `$v-included-components` 参数也可以使用同样的方式来定制, 详情请参见第 7.6.6 节 “Theme 优化”.

配置参数

以下变量控制了一些共通的组件样式:

```
$v-scaling-factor--tiny (默认值: 0.75)
一个缩放系数, 适用于 TINY 组件样式.
```

`$v-scaling-factor--small` (默认值: 0.85)

一个缩放系数, 适用于 `SMALL` 组件样式.

`$v-scaling-factor--large` (默认值: 1.2)

一个缩放系数, 适用于 `LARGE` 组件样式.

`$v-scaling-factor--huge` (默认值: 1.6)

一个缩放系数, 适用于 `HUGE` 组件样式.

7.6.6. Theme 优化

Valo theme 允许优化编译后的 CSS 尺寸, 方法是只将应用程序使用到的组件的相关规则 include 到样式表中. 被 include 的组件样式可以使用 `$v-included-components` 变量来指定, 这个参数默认是 include 所有组件. 这个变量应该包含一个逗号分隔的列表, 其中元素是小写的组件名称. 同样的, 你也可以使用 `$v-included-additional-styles` 参数, 指定包含哪些额外的组件样式, 格式类似, 详情请参见第 7.6.5 节“组件样式控制”. 额外组件样式的默认值为 `$v-included-components`.

比如, 如果你的 UI 只包含 **VerticalLayout**, **TextField**, 以及 **Button** 组件, 你应该将这个参数定义为:

```
$v-included-components:
    verticallayout,
    textfield,
    button;
```

你可以反转使用 `remove()` 函数, 从标准选择中删除一部分组件.

比如, 使用以下代码, 你可以删除 **Calendar** 组件的 theme 定义:

```
$v-included-components: remove($v-included-components, calendar);
```

注意, 在这种情况下, 你需要将这条语句放在 Valo theme 的 `@import` 语句之后, 因为这条语句会使用到 Valo theme 内的一个变量.

7.7. 字体图标

字体图标是指字体内包含的图标文字. 与位图图片相比, 字体有很多优势. 浏览器描绘字体通常会比装载图片文件要快. Web 字体是矢量图, 因此适合于缩放. 由于字体图标都是文本字符, 所以你可以在 CSS 中使用通常的前景色属性来控制它们的颜色.

7.7.1. 装载图标字体

Vaadin 目前带有一个自定义的图标字体: `FontAwesome`. Valo theme 会自动启用这个字体. 在其他 theme 中, 你需要在你的 theme 文件中, 在 import 基本 theme 之后, 使用以下代码来包含这个字体:

```
@include fonticons;
```

如果你要使用其他图标字体(详情请参见第 7.7.5 节“定制字体图标”), 并且基础 theme 没有装载这个字体, 那么你就需要在 Sass 中使用 font mixin 来装载它, 详情请参见第 7.8.1 节“装载字体”.

7.7.2. 基本使用

字体图标是 **FontIcon** 类型的资源, 这个类实现了 `Resource` 接口. 你可以将这些特殊资源用作组件图标等, 但不能用作内嵌的图片.

每个图标都有一个 Unicode 代码, 你需要通过这个 Unicode 代码来使用对应的图标. Vaadin 包含了一个非常好的图标字体, `FontAwesome`, 这个类中包含了字体内所有图标的枚举列表.

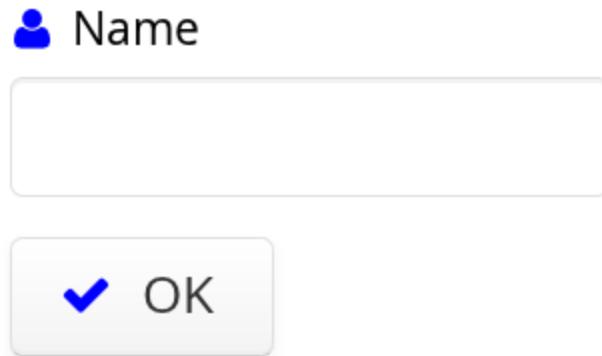
通常, 你可以使用以下方式为组件设置图标:

```
TextField name = new TextField("Name");
name.setIcon(FontAwesome.USER);
layout.addComponent(name);

// Button allows specifying icon resource in constructor
Button ok = new Button("OK", FontAwesome.CHECK);
layout.addComponent(ok);
```

运行结果见 图 7.10 “字体图标的基本使用”, 其中还带有彩色风格, 关于彩色风格, 将在下一段中介绍.

图 7.10. 字体图标的基本使用



控制图标的样式

由于字体图标都是通常的文本, 所以你可以使用 CSS 中控制前景文字颜色的 `color` 属性来指定图标的颜色. Vaadin 中所有显示图标的 HTML 元素都带有 `v-icon` 样式名.

```
.v-icon {
    color: blue;
}
```

如果你以其他的方式来使用字体图标资源, 比如用在 `Image` 组件中, 那么样式名将会不同.

7.7.3. 在 HTML 中使用字体图标

你可以在 HTML 代码中使用字体图标, 比如用在 `Label` 中, 这种情况下可以通过 `getHtml()` 方法得到用于显示图标的 HTML 代码.

```
Label label = new Label("I " +
    FontAwesome.HEART.getHtml() + " Vaadin",
    ContentMode.HTML);
```

```
label.addStyleName("redicon");
layout.addComponent(label);
```

图标的 HTML 代码带有 v-icon 样式名, 你可以在 CSS 中修改它的样式:

```
.redicon .v-icon {
    color: red;
}
```

运行结果见 图 7.11 “在 Label 中使用字体图标”, 其中还带有彩色风格, 关于彩色风格, 将在下一段中介绍.

图 7.11. 在 **Label** 中使用字体图标



你也可以在 Label 的 HTML 代码中设置字体颜色, 也可以为 UI 中所有图标设置颜色.

你也可以通过其他方式很简单地在 HTML 代码中使用字体图标, 只需要使用正确的字体家族, 然后使用图标对应的 16 进制的 Unicode 代码值. 比如, **FontAwesome** 的 getHtml() 方法的实现就是这样:

```
@Override
public String getHtml() {
    return "<span class=\"v-icon\" style=\"font-family: " +
        getFontFamily() + ";\\>&#x" +
        Integer.toHexString(codepoint) + "</span>";
}
```

7.7.4. 在其他文本中使用字体图标

你可以使用一个字体图标的 Unicode 代码值, 来将图标嵌入到任何文本中, Unicode 代码值可以通过 getCodePoint() 方法得到. 但是, 这时对于同一段文本内的其他文字, 你也需要使用相同的字体. Vaadin 附带的 FontAwesome 字体包含了基本的字符集.

```
TextField amount = new TextField("Amount (in " +
    new String(Character.toChars(
        FontAwesome.BTC.getCodepoint()))) +
    ")");
amount.addStyleName("awesomecaption");
layout.addComponent(amount);
```

你需要在 CSS 中设置字符家族.

```
.v-caption-awesomcaption .v-captions {
    font-family: FontAwesome;
}
```

7.7.5. 定制字体图标

你可以很简便地将现存字体内的图形当作图标来使用, 也可以创建你自己的字体图标.

使用 IcoMoon 创建新的图标字体

你可以完全自由地使用任何方法来创建图标，然后将它们嵌入到字体内。这里，我们简单介绍如何使用 IcoMoon 服务，通过它你可以选择在一个很大的很精美的图标库中选择你需要的图标。

Font Awesome 已经包含在 IcoMoon 的图标库选项之内了。注意，图标的 Unicode 代码值不是固定的，因此 **FontAwesome** 枚举型中的值，与这类自定义图标字体是不兼容的。

选择你的字体中希望包含的图标之后，你可以通过 ZIP 包的形式下载它们。ZIP 包内包含这些图标的多种文件格式，包括 WOFF, TTF, EOT, 以及 SVG。但其中任何一个格式，都不是被所有的浏览器支持的，因此，为了让你的应用程序支持所有的常用浏览器，就需要包含所有的文件格式。你需要从 ZIP 包中解开 fonts 目录，然后放在你的 theme 之下。

关于如何装载一个自定义字体，详情请参见 第 7.8.1 节“装载字体”。

实现 FontIcon 接口

你可以对浏览器支持的任何字体定义字体图标，方法是实现 FontIcon 接口。实现这个接口的通常模式是实现一个枚举，列举出字体内所有可用的符号。详情请参见 **FontAwesome** 类的实现。

使用图标时需要 FontIcon API。下例中，我们使用浏览器内建的普通的 sans-serif 字体来定义一个字体图标。

```
// Font icon definition with a single symbol
public enum MyFontIcon implements FontIcon {
    EURO(0x20AC);

    private int codepoint;

    MyFontIcon(int codepoint) {
        this.codepoint = codepoint;
    }

    @Override
    public String getMIMEType() {
        throw new UnsupportedOperationException(
            FontIcon.class.getSimpleName()
            + " should not be used where a MIME type is needed.");
    }

    @Override
    public String getFontFamily() {
        return "sans-serif";
    }

    @Override
    public int getCodepoint() {
        return codepoint;
    }

    @Override
    public String getHtml() {
        return "<span class=\"v-icon\" style=\"font-family: " +
            getFontFamily() + ";\\">&#x" +
            Integer.toHexString(codepoint) + ";</span>";
    }
}
```

然后, 你可以这样使用它:

```
TextField name = new TextField("Amount");
name.setIcon(MyFontIcon.EURO);
layout.addComponent(name);
```

你也可以使用类的方式来实现, 而不是使用枚举型, 以便通过其他方式来指定图标.

7.8. 自定义字体

除浏览器的内建字体, 以及 Vaadin theme 中包含的 Web 字体之外, 你还可以使用自定义的 Web 字体.

7.8.1. 装载字体

你可以使用 font mixin 来装载 Web 字体, 如下:

```
@include font(MyFontFamily,
  '../../.mytheme/fonts/myfontfamily');
```

这条语句必须放在 styles.scss 文件的 .mytheme {} 部分的 外部.

第一个参数是字体家族的名称, 用于标识字体. 如果字体家族名称中包含空格, 你需要使用单引号或双引号将名称括起.

第二个参数是字体文件名, 不带扩展名, 但包含相对路径. 注意, 路径是相对于基础 theme 的, 也就是定义这个 mixin 的 theme, 而不是当前使用的 theme. 我们建议将自定义字体文件放在 theme 下的 fonts 文件夹中.

并不是所有的浏览器都支持全部的字体文件格式, 因此字体文件使用的扩展名是 .ttf, .eot, .woff, 或 .svg, 具体使用哪个, 要根据浏览器所支持的字体格式来决定.

7.8.2. 使用自定义字体

字体装载完成后, 你就可以在 CSS 或其他地方通过字体名称使用自定义字体, 或字体家族.

```
.mystyle {
  font-family: MyFontFamily;
}
```

再次强调, 如果字体家族名称包含空格, 你需要使用单引号或双引号将名称括起.

7.9. 条件式 Theme

Vaadin 支持动态适应式的界面设计能力, 可以在 CSS 选择器中使用尺寸范围作为选择条件, 因此可以实现与客户端浏览器窗口尺寸大小动态匹配的条件式 CSS 规则. 详情请参见 Vaadin Blog 关于 Responsive design 的文章.

你可以使用 **Responsive** 来扩展任何组件, 通常是布局组件, 也可以扩展整个 UI. 你需要在静态方法 makeResponsive() 中指定被扩展的组件.

```
// Have some component with an appropriate style name
Label c = new Label("Here be text");
c.addStyleName("myresponsive");
content.addComponent(c);
```

```
// Enable Responsive CSS selectors for the component
Responsive.makeResponsive(c);
```

然后你可以在 CSS 选择器中使用 width-range 和 height-range 条件, 如下:

```
/* Basic settings for all sizes */
.myresponsive {
    padding: 5px;
    line-height: 36pt;
}

/* Small size */
.myresponsive[width-range~="0-300px"] {
    background: orange;
    font-size: 16pt;
}

/* Medium size */
.myresponsive[width-range~="301px-600px"] {
    background: azure;
    font-size: 24pt;
}

/* Anything bigger */
.myresponsive[width-range~="601px-"] {
    background: palegreen;
    font-size: 36pt;
}
```

选择条件中的尺寸范围可以重叠, 这时与当前尺寸匹配的所有选择器都将有效.

灵活折行

你可以使用 **CssLayout** 来实现布局内组件超出布局右边界时的自动折行. 折行的组件, 宽度必须为未指定或者为固定值, 因此不能使用屏幕的全部区域. 使用 **Responsive** 扩展, 你可以实现更灵活的折行, 允许组件填充为最大宽度.

下例中, 我们有一个文本和一个图片, 如果屏幕足够宽, 它们将水平排列, 分别占据 50% 宽度, 但如果屏幕比较窄, 则会折行, 垂直排列.

```
CssLayout layout = new CssLayout();
layout.setWidth("100%");
layout.addStyleName("flexwrap");
content.addComponent(layout);

// Enable Responsive CSS selectors for the layout
Responsive.makeResponsive(layout);

Label title = new Label("Space is big, really big");
title.addStyleName("title");
layout.addComponent(title);

Label description = new Label("This is a " +
    "long description of the image shown " +
    "on the right or below, depending on the " +
    "screen width. The text here could continue long.");
description.addStyleName("itembox");
description.setSizeUndefined();
layout.addComponent(description);
```

```
Image image = new Image(null,
    new ThemeResource("img/planets/Earth.jpg"));
image.addStyleName("itembox");
layout.addComponent(image);
```

SCSS 如下：

```
/* Various general settings */
.flexwrap {
    background: black;
    color: white;

    .title {
        font-weight: bold;
        font-size: 20px;
        line-height: 30px;
        padding: 5px;
    }

    .itembox {
        white-space: normal;
        vertical-align: top;
    }

    .itembox.v-label {padding: 5px}
}

.flexwrap[width-range~="0-499px"] {
    .itembox {width: 100%}
}

.flexwrap[width-range~="500px-"] {
    .itembox {width: 50%}
}
```

宽屏模式下的布局效果，见图 7.12 “灵活折行”。

图 7.12. 灵活折行



你还可以试试 `display: block` 和 `display: inline-block` 属性的效果。

注意, **Responsive** 扩展使得我们可以使用与组件尺寸相关的很多 CSS 技巧, 但与组件尺寸和布局尺寸相关的其他通常规则也会同时其作用, 详情请参见 第 6.13.1 节“布局的尺寸”及其他章节, 因此你需要时刻注意与组件尺寸相关的问题。在上例中, 我们将 `label` 宽度设置为未指定, 这就导致 `label` 内的文字折行被禁止, 因此我们需要再次启用它。

控制 `Display` 属性

`display` 属性提供了功能强大的能力, 可以针对不同的屏幕尺寸实现完全不同的 UI, 方法是根据需要启用和禁用 UI 元素。比如, 当空间太小时, 你可以禁用 UI 的一部分, 并显示导航按钮, 当导航按钮按下时, 通过操纵组件的样式来切换到原来隐藏的部分。

下面是一个简单的例子, 我们根据屏幕宽度不同显示不同的组件:

```
CssLayout layout = new CssLayout();
layout.setWidth("100%");
layout.addStyleName("toggledisplay");
content.addComponent(layout);

// Enable Responsive CSS selectors for the layout
Responsive.makeResponsive(layout);

Label enoughspace =
    new Label("This space is big, mindbogglingly big");
enoughspace.addStyleName("enoughspace");
layout.addComponent(enoughspace);

Label notEnoughspace = new Label("Quite small space");
notEnoughspace.addStyleName("notEnoughspace");
layout.addComponent(notEnoughspace);
```

SCSS 如下:

```
/* Common settings */
.toggledisplay {
    .enoughspace, .notenoughspace {
        color: white;
        padding: 5px;
    }

    .notenoughspace { /* Really small */
        background: red;
        font-weight: normal;
        font-size: 10px;
        line-height: 15px;
    }

    .enoughspace { /* Really big */
        background: darkgreen;
        font-weight: bold;
        font-size: 20px;
        line-height: 30px;
    }
}

/* Quite little space */
.toggledisplay[width-range~="0-499px"] {
    .enoughspace {display: none}
}

/* Plenty of space */
.toggledisplay[width-range~="500px-"] {
    .notenoughspace {display: none}
}
```

条件式 theme 的演示

在 demo.vaadin.com/responsive 可以看到一个条件式 theme 的简单示例. 这个示例演示“灵活折行”一节中介绍的灵活折行技术.

Book Examples 包括本章中的示例程序, 以及其他示例.

针对 TouchKit(参见第 20 章 使用 TouchKit 创建移动设备应用程序) 的 Parking demo, 使用条件式 theme 来自动适应移动设备的不同屏幕尺寸, 以及不同的屏幕方向.

8.1. 概述	279
8.2. 属性(Property)	281
8.3. 在项目(Item)中保存属性(Property)	287
8.4. 通过 Field 与项目的绑定来创建 Form	289
8.5. 在容器(Container)中保存项目(Item)	294

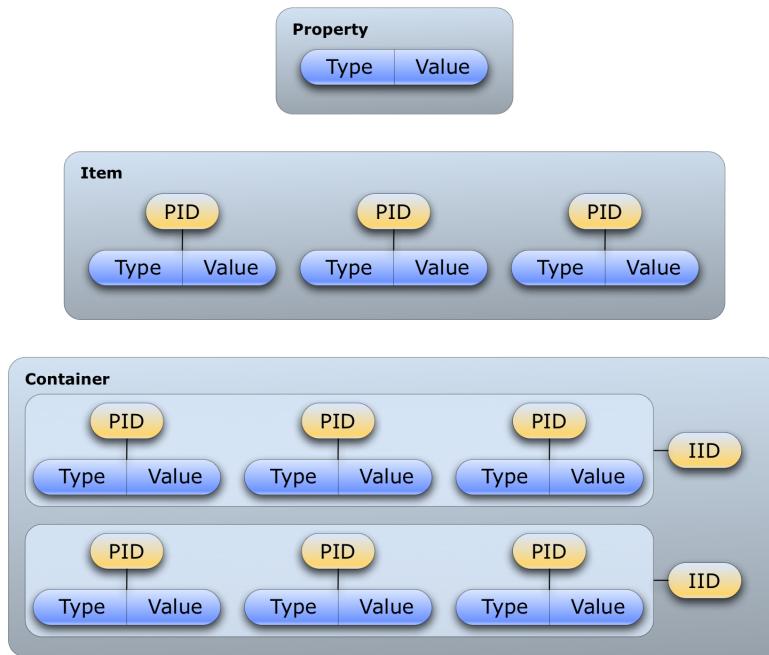
本章介绍 Vaadin Data Model，并讲解如何使用它将组件绑定到数据源，比如一个数据库查询。

8.1. 概述

Vaadin Data Model 是 Vaadin 的核心概念之一。为了让 view(UI 组件)模块直接访问应用程序的数据模型，我们引入了一套标准的数据接口。

这个数据模型允许 UI 组件直接绑定到数据上，然后 UI 组件就可以显示或者编辑这些数据。数据模型包含三个嵌套层次：属性(*property*)，项目(*item*)，以及 容器(*container*)。用电子表格应用程序来做类比的话，这三个层次分别对应于单元格，行，表。

图 8.1. Vaadin Data Model



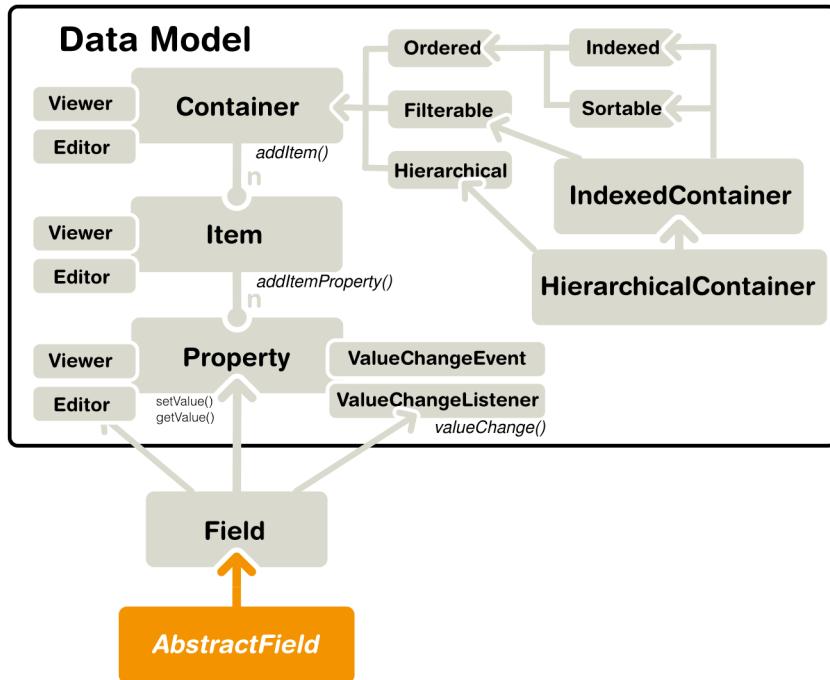
数据模型由 **com.vaadin.data** 包内的一组接口构成。这个包中包含 **Property**, **Item**, 和 **Container** 接口, 以及其他很多特化的接口和类。

注意, 数据模型并没有定义数据的表现形式, 只定义了接口。这种设计将数据的表现方式完全留给容器的具体实现来决定。数据的表现几乎可以是任何形式, 比如可以是简单 Java 对象(POJO, plain old Java object), 可以是文件系统, 也可是数据库查询。

数据模型在 Vaadin 的核心 UI 组件中广泛使用, 尤其是 Field 组件中, 也就是实现了 **Field** 接口的组件, 或者更典型地说, 继承自 **AbstractField** 的组件, **AbstractField** 类中定义了很多共通功能。内建的 Field 组件的一个关键功能是, 它们既可以自行维护数据, 也可以绑定到外部的数据源。Field 的值总是可以通过 **Property** 接口得到。由于绑定到同一个数据源的组件可以超过一个, 因此可以很容易实现多阅读者/多编辑者模式。

数据模型中各接口的关系, 见 图 8.2 “Vaadin Data Model 中各接口的关系”; 值变化的事件和监听器接口, 只显示了与 **Property** 接口相关的部分, 省略了通知器接口。

图 8.2. Vaadin Data Model 中各接口的关系



数据模型还有很多重要和有用的功能, 比如支持变更通知. 尤其是, 容器还带有很多辅助接口, 可用于数据的索引, 排序, 过滤. **Field** 组件也提供了很多与数据模型相关的功能, 比如缓冲, 校验, 以及 lazy load.

Vaadin 对数据模型接口提供了很多内建的实现类. 在很多 Field 组件中, 内建的实现类被用作默认的数据模型.

出内建的实现外, 还可以通过 add-on 得到很多数据模型的实现, 比如容器, add-on 可以从 Vaadin Directory 得到, 也可以从其他独立的来源得到. 既有商业化的实现, 也有免费的实现. JPAContainer, 参见 第 19 章 Vaadin JPAContainer, 是最常用到的商业化容器 add-on. add-on 的安装请参见 第 17 章 使用 Vaadin Add-on. 注意, 与大多数通常的 add-on 组件不同, 对于只包含数据模型实现的 add-on, 你不必为它编译 widget 群.

8.2. 属性(Property)

Property 接口是 Vaadin Data Model 的基础. 它提供了一组标准 API, 用于单个数据值对象的读 (get) 和写 (set). 一个属性永远是固定类型的, 但也可以支持数据类型的转换. 属性的类型可以是任意的 Java 类. 属性还可以向外提供事件, 以便外界追踪值的变化.

你可以使用 `setValue()` 方法来设置属性的值, 可以通过 `getValue()` 方法读取属性值.

下例中, 我们通过 **TextField** 组件来读写一个属性的值, 这个组件实现了 **Property** 接口, 因此可以访问 Field 内的值.

```

final TextField tf = new TextField("Name");

// Set the value
tf.setValue("The text field value");

// When the field value is edited by the user
  
```

```

tf.addValueChangeListener(
    new Property.ValueChangeListener() {
        public void valueChange(ValueChangeEvent event) {
            // Do something with the new value
            layout.addComponent(new Label(tf.getValue()));
        }
    });

```

属性值的变更通常会激发一个 **ValueChangeEvent** 事件，可以使用 **ValueChangeListener** 监听器来处理这个事件。事件对象通过 `getProperty()` 方法提供了指向属性的引用。注意，`getValue()` 方法返回值类型为 **Object**，因此你需要将它转换为正确的类型。

属性内部是没有名称概念的。它们会被组织在项目之内，在项目的层次上，属性与名称相关联，名称就是属性 **ID** 或者简称 **PID**。项目再被组织在容器中，项目的标识符称作 **项目ID** 或者简称 **IID**。用电子表格来类比的话，属性 **ID** 对应到列名，项目 **ID** 对应到行名。**ID** 可以是任意类型的对象，但必须实现 `equals(Object)` 和 `hashCode()` 方法，以便在标准的 Java **Collection** 中存储这些 **ID**。

使用 **Property** 时，可以直接实现这个接口，也可以使用内建的实现。Vaadin 包含了一些 **Property** 接口的实现，**MethodProperty** 类，用于任意函数对和 Bean 属性，**ObjectProperty** 类，用于简单的对象属性，详情见后文。

除简单组件外，选择组件也将它们的当前选中内容以属性值的形式向外提供。单选模式下，属性是单个的项目 **ID**，多选模式下，属性是一组项目 **ID** 的集合。详情请参见选择组件的相关文档。

可以绑定到属性上的组件，带有一个内部的默认数据源对象，一般是 **ObjectProperty**，关于这个对象的介绍见后文。由于这类组件都是查看器或编辑器（详情见后文），因此你可以使用 `setPropertyDataSource()` 方法将组件重绑定到任何的数据源上。

8.2.1. 属性的查看器和编辑器

Property（以及其他数据模型接口）的最重要的功能是，将数据模型接口的实现类直接连接到编辑器类和查看器类。也就是说将数据源（模型）与 UI 组件（视图）相连，实现对数据模型的编辑或查看功能。

属性可以通过 `setPropertyDataSource()` 方法绑定到实现了 **Viewer** 接口的组件。

```

// Have a data model
ObjectProperty property =
    new ObjectProperty("Hello", String.class);

// Have a component that implements Viewer
Label viewer = new Label();

// Bind it to the data
viewer.setPropertyDataSource(property);

```

你也可以使用 **Editor** 接口中的同一个方法，将某种类型属性的编辑组件绑定到一个属性上。

```

// Have a data model
ObjectProperty property =
    new ObjectProperty("Hello", String.class);

// Have a component that implements Viewer
TextField editor = new TextField("Edit Greeting");

```

```
// Bind it to the data
editor.setPropertyDataSource(property);
```

由于所有的 Field 组件都实现了 **Property** 接口，所以你可以将实现 **Viewer** 接口的任何组件绑定到任何的 Field 上，前提是这里的查看器有能力查看 Field 中的数据类型。继续上面的例子，我们可以绑定一个 **Label** 组件到 **TextField** 的值上：

```
Label viewer = new Label();
viewer.setPropertyDataSource(editor);

// The value shown in the viewer is updated immediately
// after editing the value in the editor (once it
// loses the focus)
editor.setImmediate(true);
```

如果 Field 带有校验器，参见第 5.4.5 节“Field 值校验”，校验器的执行会发生在值写入属性数据源之前，或者在对 Field 调用 validate() 或 commit() 方法时。

8.2.2. ObjectProperty 实现

ObjectProperty 类是 **Property** 接口的一个简单实现，它允许在属性中存储一个任意的 Java 对象。

```
// Have a component that implements Viewer interface
final TextField tf = new TextField("Name");

// Have a data model with some data
String myObject = "Hello";

// Wrap it in an ObjectProperty
ObjectProperty<String> property =
    new ObjectProperty<String>(myObject, String.class);

// Bind the property to the component
tf.setPropertyDataSource(property);
```

8.2.3. 在属性类型与表达之间转换

Field 用于编辑某个特定类型的数据，比如 **String** 或 **Date**。但是与它绑定的属性，可能包含完全不同的数据类型。在 Field 编辑的数据表达形式，与属性中定义的数据模型之间，由转换器来进行数据类型的转换，转换器需要实现 **Converter** 接口。

大多数常见类型之间的转换，比如字符串与整数的转换，由默认的转换器来处理。这些转换器由应用程序中的一个全局的转换器工厂负责创建。

转换器的基本使用

`setConverter(Converter)` 方法为 Field 设置转换器。这个方法定义在 **AbstractField** 类中。

```
// Have an integer property
final ObjectProperty<Integer> property =
    new ObjectProperty<Integer>(42);

// Create a TextField, which edits Strings
final TextField tf = new TextField("Name");

// Use a converter between String and Integer
tf.setConverter(new StringToIntegerConverter());
```

```
// And bind the field
tf.setPropertyDataSource(property);
```

内建的转换器如下：

表 8.1. 内建的转换器

转换器类	组件中的数据表现	模型中的数据类型
StringToIntegerConverter	String	Integer
StringToDoubleConverter	String	Double
StringToNumberConverter	String	Number
StringToBooleanConverter	String	Boolean
StringToDateConverter	String	Date
DateToLongConverter	Date	Long

除此之外，还有 **ReverseConverter** 类，它接受一个转换器作为参数，然后反转这个转换器的转换方向。

如果对于某个数据类型已经有转换器存在，那么 `setConverter(Class)` 方法可以从转换器工厂中得到这个数据类型对应的转换器，然后再将这个转换器设置到 Field 上。当绑定 Field 到属性数据源时，会隐含地使用这个方法。

转换器的实现

转换器总是出现在 表现类型 与 数据模型类型 之间，表现类型就是 Field 组件编辑的类型，数据模型类型就是属性数据源中的类型。转换器需要实现 `com.vaadin.data.util.converter` 包中的 `Converter` 接口。

比如，假设我们有一个简单的 **Complex** 类型，用于存储复数值。

```
public class ComplexConverter
    implements Converter<String, Complex> {
    @Override
    public Complex convertToModel(String value, Locale locale)
        throws ConversionException {
        String parts[] =
            value.replaceAll("[\\\"\\\\\\\\]\"", "").split(",");
        if (parts.length != 2)
            throw new ConversionException(
                "Unable to parse String to Complex");
        return new Complex(Double.parseDouble(parts[0]),
                           Double.parseDouble(parts[1]));
    }

    @Override
    public String convertToPresentation(Complex value,
                                       Locale locale)
        throws ConversionException {
        return "("+value.getReal()+","+value.getImag()+" )";
    }

    @Override
    public Class<Complex> getModelType() {
```

```

        return Complex.class;
    }

    @Override
    public Class<String> getPresentationType() {
        return String.class;
    }
}

```

转换器接受一个语言环境(locale)作为参数, 指定转换时使用的语言环境.

转换器工厂

如果一个Field不能直接编辑属性值的数据类型, 它会尝试使用应用程序全局的转换器工厂来创建一个默认转换器. 如果你定义了自己的转换器, 希望加入到转换器工厂中去, 你需要自己来实现一个转换器工厂. 你可以实现 `ConverterFactory` 接口, 但更简单的方法是继承 `DefaultConverterFactory` 类.

```

class MyConverterFactory extends DefaultConverterFactory {
    @Override
    public <PRESENTATION, MODEL> Converter<PRESENTATION, MODEL>
        createConverter(Class<PRESENTATION> presentationType,
                       Class<MODEL> modelType) {
        // Handle one particular type conversion
        if (String.class == presentationType &&
            Complex.class == modelType)
            return (Converter<PRESENTATION, MODEL>)
                new ComplexConverter();

        // Default to the supertype
        return super.createConverter(presentationType,
                                     modelType);
    }
}

// Use the factory globally in the application
Application.getCurrentApplication().setConverterFactory(
    new MyConverterFactory());

```

8.2.4. 实现 `Property` 接口

实现 `Property` 接口, 需要为属性值和只读模式分别定义 `set` 和 `get` 方法. 对于属性值的数据类型, 只需要 `get` 方法, 因为对于 `Property` 的实现类来说, 数据类型通常是固定的.

下例演示 `Property` 接口的一个简单的实现:

```

class MyProperty implements Property {
    Integer data      = 0;
    boolean readOnly = false;

    // Return the data type of the model
    public Class<?> getType() {
        return Integer.class;
    }

    public Object getValue() {
        return data;
    }
}

```

```

// Override the default implementation in Object
@Override
public String toString() {
    return Integer.toHexString(data);
}

public boolean isReadOnly() {
    return readOnly;
}

public void setReadOnly(boolean newStatus) {
    readOnly = newStatus;
}

public void setValue(Object newValue)
        throws ReadOnlyException, ConversionException {
    if (readOnly)
        throw new ReadOnlyException();

    // Already the same type as the internal representation
    if (newValue instanceof Integer)
        data = (Integer) newValue;

    // Conversion from a string is required
    else if (newValue instanceof String)
        try {
            data = Integer.parseInt((String) newValue, 16);
        } catch (NumberFormatException e) {
            throw new ConversionException();
        }
    else
        // Don't know how to convert any other types
        throw new ConversionException();

    // Reverse decode the hexadecimal value
}
}

// Instantiate the property and set its data
MyProperty property = new MyProperty();
property.setValue(42);

// Bind it to a component
final TextField tf = new TextField("Name", property);

```

组件通过 `toString()` 方法得到画面上显示的值，因此我们需要重载这个方法。为了编辑值，`toString()` 方法返回的值，其格式必须能被 `setValue()` 方法接受，除非属性是只读的。`toString()` 方法可以进行任何必要的类型转换，将内部数据类型转换为字符串，`setValue()` 方法则必须进行反向的转换。

上面的 `Property` 实现示例，没有向外发送属性值和只读模式的变更通知。通常你至少应该实现 **Property.ValueChangeNotifier** 和 **Property.ReadOnlyStatusChangeNotifier**。关于这些通知功能的实现方式，请参考 **ObjectProperty** 类。

8.3. 在项目(Item)中保存属性(Property)

通过 **Item** 接口可以访问一组有名称的属性。各个属性通过 属性ID (PID) 来区分，使用 **Item** 的 `getItemProperty()` 方法，可以根据 ID 得到对应的属性。

使用项目的例子包括：1, **Table** 中的行，其中的属性对应到 Table 的列, 2, **Tree** 中的节点, 3, 以及绑定到 **Form** 的数据，其中的属性分别绑定到 Form 中各个独立的 Field。

项目基本上等价于面向对象编程模型中的对象，区别在于项目是可配置的，而且提供了事件处理机制。使用 **Item** 的最简单方法是使用已有的实现类。Vaadin 提供的工具类包括可配置的属性集 (**PropertysetItem**) 以及 Bean 到项目的变换器(**BeanItem**)。此外，**Form** 也实现了 **Item** 接口，因此可以直接当作项目来使用。

除了被很多 UI 组件间接使用之外，项目也提供了 **Form** 组件之下的基本数据模型。简单情况下，Form 甚至可以通过项目来自动生成。项目中的属性与 Form 中的 Field 一一对应。

Item 接口定义了内部接口，用于维护项目中的属性集合，以及监听项目的变更事件。实现了 **PropertySetChangeNotifier** 接口的类可以激发 **PropertySetChangeEvent** 事件。这个事件可以通过 **PropertySetChangeListener** 接口来接收。

8.3.1. PropertysetItem 实现

PropertysetItem 是 **Item** 接口的一个通用实现，它可以存储属性。属性通过 `addItemProperty()` 方法加入到项目中，这个方法接受的参数是属性名和属性对象。

下例演示使用 **ObjectProperty** 在项目中存储属性的常见用法：

```
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty("Zaphod"));
item.addItemProperty("age", new ObjectProperty(42));

// Bind it to a component
Form form = new Form();
form.setItemDataSource(item);
```

8.3.2. 使用 BeanItem 包装一个 Bean

Item 接口的 **BeanItem** 实现是对 Java Bean 对象的一种包装。实际上，只有在序列化时需要用到 set 和 get 方法，Java Bean 的其他功能不是必须的，因此你可以使用这个类来包装几乎所有的 POJO 对象。

```
// Here is a bean (or more exactly a POJO)
class Person {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }
}
```

```

    }

    public void setAge(Integer age) {
        this.age = age.intValue();
    }
}

// Create an instance of the bean
Person bean = new Person();

// Wrap it in a BeanItem
BeanItem<Person> item = new BeanItem<Person>(bean);

// Bind it to a component
Form form = new Form();
form.setItemDataSource(item);

```

你可以使用 `getBean()` 方法取得项目之下包装的 Bean 对象.

嵌套 Bean

你经常会将多个类组装在一起, 其中一个类 "拥有" 其他类. 比如, 考虑下例中的 **Planet** 类, 它"拥有"一个"发现者":

```

// Here is a bean with two nested beans
public class Planet implements Serializable {
    String name;
    Person discoverer;

    public Planet(String name, Person discoverer) {
        this.name = name;
        this.discoverer = discoverer;
    }

    ... getters and setters ...
}

...
// Create an instance of the bean
Planet planet = new Planet("Uranus",
                           new Person("William Herschel", 1738));

```

在 **Form** 内显示时, 比如, 你希望列出组合 Bean 的属性, 以及嵌套 Bean 的属性. 你可以使用 **MethodProperty** 或 **NestedMethodProperty** 来单独绑定嵌套 Bean 的属性. 在绑定组合 Bean 的属性时, 你通常应该将嵌套 Bean 本身排除在外, 方法是在构造方法中只列出需要绑定的属性.

```

// Wrap it in a BeanItem and hide the nested bean property
BeanItem<Planet> item = new BeanItem<Planet>(planet,
                                              new String[]{"name"});

// Bind the nested properties.
// Use NestedMethodProperty to bind using dot notation.
item.addItemProperty("discoverername",
                      new NestedMethodProperty(planet, "discoverer.name"));

// The other way is to use regular MethodProperty.
item.addItemProperty("discovererborn",
                      new MethodProperty<Person>(planet.getDiscoverer(),
                                                 "born"));

```

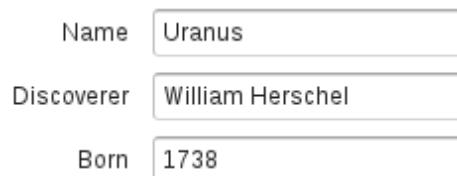
上面两个方法的区别在于, **NestedMethodProperty** 方法不会立即访问嵌套 Bean, 只会在访问属性值时才需要访问到嵌套 Bean, 而使用 **MethodProperty** 方法时, 嵌套 Bean 会在创建 method property 时立即访问. 只有在嵌套 Bean 可能为 null 时, 或者将来可能被变更时, 以上区别才是重要的.

这里创建的 Bean 项目, 举例来说, 可以在 **Form** 中使用, 如下:

```
// Bind it to a component
Form form = new Form();
form.setItemDataSource(item);

// Nicer captions
form.getField("discoverername").setCaption("Discoverer");
form.getField("discovererborn").setCaption("Born");
```

图 8.3. 使用嵌套 Bean 属性的 Form



BeanContainer 和 **BeanItemContainer** 可以使用 `addNestedContainerProperty()` 方法简单地定义嵌套 Bean 属性, 详情请参见“嵌套属性”一节.

8.4. 通过 Field 与项目的绑定来创建 Form

为了尽快完成本书, 我们来不及彻底地更新本章的内容. Form 处理的一些相关内容目前还在编写中, 尤其是 Form 的校验.

现实中的大多数应用程序都包含某种形式的 Form. Form 中包含 Field, 你会希望将 Field 绑定到数据源上, 也就是 Vaadin 数据模型中的一个项目. **FieldGroup** 提供了一种简便的方式来将 Field 绑定到项目中的属性. 你可以首先绑定创建一个布局, 和一些 Field, 然后使用它将 Field 绑定到数据源. 也可以让 **FieldGroup** 使用 Field 工厂来创建 Field. 它也会处理数据的提交. 注意, **FieldGroup** 不是一个 UI 组件, 因此你不能将它添加到布局中.

8.4.1. 简易绑定

让我们从一个数据模型开始, 这个数据模型中包含一个项目, 项目中包含几个属性. 前面我们已经介绍过, 项目可以是任意类型.

```
// Have an item
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Zaphod"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));
```

下面, 你需要设计一个 Form 来编辑数据. **FormLayout** (参见第 6.5 节 “**FormLayout**”) 最适合于 Form, 但你也可以使用其他布局.

```
// Have some layout and create the fields
FormLayout form = new FormLayout();

TextField nameField = new TextField("Name");
form.addComponent(nameField);
```

```
TextField ageField = new TextField("Age");
form.addComponent(ageField);
```

然后，我们可以将 Field 绑定到数据上，如下：

```
// Now create the binder and bind the fields
FieldGroup binder = new FieldGroup(item);
binder.bind(nameField, "name");
binder.bind(ageField, "age");
```

上面的数据绑定方式与直接调用 Field 的 `setPropertyDataSource()` 方法没有区别。但是，它会将 Field 注册到 FieldGroup 中，因此会通过 FieldGroup 来实现针对 Field 的缓冲和校验等功能，详情请参见第 8.4.4 节“Form 的缓冲”。

下面，我们来考查一下 **FieldGroup** 更具体的应用。

8.4.2. 使用 FieldFactory 创建并绑定 Field

通过 `buildAndBind()` 方法，**FieldGroup** 可以使用 `FieldGroupFieldFactory` 为你创建 Field，但是你仍然需要将这些 Filed 添加到布局内的正确位置上。

```
// Have some layout
FormLayout form = new FormLayout();

// Now create a binder that can also create the fields
// using the default field factory
FieldGroup binder = new FieldGroup(item);
form.addComponent(binder.buildAndBind("Name", "name"));
form.addComponent(binder.buildAndBind("Age", "age"));
```

8.4.3. 绑定成员 Field

FieldGroup 中的 `bindMemberFields()` 方法使用反射机制来将项目中的属性绑定到类的成员变量的 Field 组件上。因此，如果你以类的形式实现一个 Form，把 Field 作为成员变量保存在类中，你就可以使用这个方法非常便利地绑定这些 Field。

项目属性和成员变量通过属性 ID 和成员变量名来对应。如果你希望将某个成员变量映射到不同的属性 ID，你可以对这个成员变量使用 `@PropertyId` 注解，并将属性 ID 作为这个注解的参数。

例如：

```
// Have an item
PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Zaphod"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));

// Define a form as a class that extends some layout
class MyForm extends FormLayout {
    // Member that will bind to the "name" property
    TextField name = new TextField("Name");

    // Member that will bind to the "age" property
    @PropertyId("age")
    TextField ageField = new TextField("Age");

    public MyForm() {
        // Customize the layout a bit
    }
}
```

```

        setSpacing(true);

        // Add the fields
        addComponent(name);
        addComponent(ageField);
    }
}

// Create one
MyForm form = new MyForm();

// Now create a binder that can also creates the fields
// using the default field factory
FieldGroup binder = new FieldGroup(item);
binder.bindMemberFields(form);

// And the form can be used in an higher-level layout
layout.addComponent(form);

```

使用 **CustomComponent** 进行封装

使用 **CustomComponent** 比继承一个布局要好, 因为可以隐藏内部的实现细节. 此外, **FieldGroup** 的使用也可以封装在 Form 类中.

关于我们前面例子中演示的 Form 实现, 考虑一下下面这种替代方案:

```

// A form component that allows editing an item
class MyForm extends CustomComponent {
    // Member that will bind to the "name" property
    TextField name = new TextField("Name");

    // Member that will bind to the "age" property
    @PropertyId("age")
    TextField ageField = new TextField("Age");

    public MyForm(Item item) {
        FormLayout layout = new FormLayout();
        layout.addComponent(name);
        layout.addComponent(ageField);

        // Now use a binder to bind the members
        FieldGroup binder = new FieldGroup(item);
        binder.bindMemberFields(this);

        setCompositionRoot(layout);
    }
}

// And the form can be used as a component
layout.addComponent(new MyForm(item));

```

8.4.4. Form 的缓冲

与独立的 Field 一样, 参见 第 5.4.4 节 “Field 值的缓存”, **FieldGroup** 也可以处理 Form 内容的缓冲, 因此只有在对 FieldGroup 调用 `commit()` 方法时, 数据才会写入数据源中. FieldGroup 还会对其中的所有 Field 进行校验, 只有在所有的 Field 都通过校验时, 才会将值写入到数据源中. 调用 `discard()` 方法可以取消对数据的编辑, 这时 Field 值会从数据源中重新装载. 缓冲功能默认是有效的, 但可以使用 **FieldGroup** 的 `setBuffered(false)` 方法来禁用.

```

// Have an item of some sort
final PropertysetItem item = new PropertysetItem();
item.addItemProperty("name", new ObjectProperty<String>("Q"));
item.addItemProperty("age", new ObjectProperty<Integer>(42));

// Have some layout and create the fields
Panel form = new Panel("Buffered Form");
form.setContent(new FormLayout());

// Build and bind the fields using the default field factory
final FieldGroup binder = new FieldGroup(item);
form.addComponent(binder.buildAndBind("Name", "name"));
form.addComponent(binder.buildAndBind("Age", "age"));

// Enable buffering (actually enabled by default)
binder.setBuffered(true);

// A button to commit the buffer
form.addComponent(new Button("OK", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        try {
            binder.commit();
            Notification.show("Thanks!");
        } catch (CommitException e) {
            Notification.show("You fail!");
        }
    }
})));
// A button to discard the buffer
form.addComponent(new Button("Discard", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        binder.discard();
        Notification.show("Discarded!");
    }
})));

```

8.4.5. 将 Field 绑定到 Bean

BeanFieldGroup 可以简化 Field 与 Bean 的绑定工作。它还可以处理嵌套 Bean 属性的绑定。要创建一个与嵌套 Bean 的属性绑定的 Field，需要用点号的形式指定属性 ID。比如，如果 **Person** Bean 中有一个 `address` 属性，其类型为 **Address**，这个类中又有一个 `street` 属性，你可以使用 `buildAndBind("Street", "address.street")` 方法，创建一个绑定到这个属性上的 Field。

Field 绑定到 Bean 之后，其中的输入内容可以使用 Java Bean Validation API 来进行校验，详情请参见第 8.4.6 节“Bean 的校验”。如果在当前类路径中存在 Bean 校验器的实现类，**BeanFieldGroup** 会为每一个 Field 自动添加 **BeanValidator**。

8.4.6. Bean 的校验

对于绑定到 Bean 属性的 Field，Vaadin 允许在值写入到 Bean 之前，使用 Java Bean Validation API 1.0 (JSR-303) 来校验输入内容。校验基于 Bean 属性上的注解来实现，这些注解用来自动创建实际的校验器。关于校验，详情请参见 第 5.4.5 节“Field 值校验”。

使用 Bean 校验功能需要依赖于 Bean Validation API 的实现库, 比如 Hibernate Validator (`hibernate-validator-4.2.0.Final.jar` 或更高版本) 或 Apache Bean Validation. 使用 Bean 校验功能时, 实现库的 JAR 文件必须放在工程的类路径中, 否则将抛出内部错误.

在使用 Vaadin JPAContainer 来管理实体 Bean 的持久化存储时, Bean 校验功能尤其有用, 详情请参见 第 19 章 *Vaadin JPAContainer*.

注解

校验规则通过注解来定义. 比如, 考虑下面这个 Bean:

```
// Here is a bean
public class Person implements Serializable {
    @NotNull
    @javax.validation.constraints.Size(min=2, max=10)
    String name;

    @Min(1)
    @Max(130)
    int age;

    // ... setters and getters ...
}
```

关于各种数据类型可以使用的校验规约列表, 请参见 Bean Validation API documentation.

校验 Bean

Bean 的校验由 **BeanValidator** 处理, 这个校验器创建时需要指定它负责校验的 Bean 属性名称, 然后需要将它添加到负责编辑数据的 Field.

下例中, 我们对一个单独的、无缓冲的 Field 进行校验:

```
Person bean = new Person("Mung bean", 100);
BeanItem<Person> item = new BeanItem<Person>(bean);

// Create an editor bound to a bean field
TextField firstName = new TextField("First Name",
        item.getItemProperty("name"));

// Add the bean validator
firstName.addValidator(new BeanValidator(Person.class, "name"));

firstName.setImmediate(true);
layout.addComponent(firstName);
```

这种情况下, 当焦点离开 Field 时会立即执行校验. 你也可以对其他 Field 进行相同的处理.

使用 **BeanFieldGroup** 时, 会自动创建 Bean 校验器.

```
// Have a bean
Person bean = new Person("Mung bean", 100);

// Form for editing the bean
final BeanFieldGroup<Person> binder =
    new BeanFieldGroup<Person>(Person.class);
binder.setItemDataSource(bean);
layout.addComponent(binder.buildAndBind("Name", "name"));
layout.addComponent(binder.buildAndBind("Age", "age"));
```

```
// Buffer the form content
binder.setBuffered(true);
layout.addComponent(new Button("OK", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        try {
            binder.commit();
        } catch (CommitException e) {
        }
    }
}));
```

Bean 校验中的语言环境(Locale)设定

校验失败时的错误消息定义在 Bean 校验实现库的 ValidationMessages.properties 文件中。显示的消息内容会与 Form 的语言环境设置保持一致。默认语言是英语，但是，Hibernate Validator 也包含了很多语言中的翻译版本。如果需要使用其他语言，你需要自己提供这个 properties 文件的翻译版。

8.5. 在容器(Container)中保存项目(Item)

Container 接口是 Vaadin 数据模型包含关系中的最高层级，负责包含项目(行)，项目再负责包含属性(列)。因此容器可以表现表格式数据，这类数据可以通过 **Table** 组件或其他选择组件来查看，容器也可以表现层级式数据。

容器内包含的项目通过 项目 ID 简称 *IID* 来区分，项目内的属性通过 属性 ID 或者简称 *PID* 来区分。

8.5.1. 容器的基本使用

容器的基本使用涉及到创建容器，添加项目，以及将组件绑定到容器数据源。

默认容器和代理

在我们介绍如何创建容器之前，应该注意，所有可以绑定到容器数据源的组件，都会默认绑定到一个默认容器。比如，**Table** 绑定到 **IndexedContainer**，**Tree** 绑定到 **HierarchicalContainer**，等等。

所有使用容器的 UI 组件自身也实现了相关容器接口，因此可以通过组件的代理来访问底层的数据源。

```
// Create a table with one column
Table table = new Table("My Table");
table.addContainerProperty("col1", String.class, null);

// Access items and properties through the component
table.addItem("row1"); // Create item by explicit ID
Item item1 = table.getItem("row1");
Property property1 = item1.getItemProperty("col1");
property1.setValue("some given value");

// Equivalent access through the container
Container container = table.getContainerDataSource();
container.addItem("row2");
Item item2 = container.getItem("row2");
Property property2 = item2.getItemProperty("col1");
property2.setValue("another given value");
```

容器的创建和绑定

创建一个容器并绑定到组件的方法如下:

```
// Create a container of some type
Container container = new IndexedContainer();

// Initialize the container as required by the container type
container.addContainerProperty("name", String.class, "none");
container.addContainerProperty("volume", Double.class, 0.0);

... add items ...

// Bind it to a component
Table table = new Table("My Table");
table.setContainerDataSource(container);
```

大多数能够绑定到容器的组件，除了可以使用 `setContainerDataSource()` 方法之外，也允许在构造函数参数中指定容器。容器的创建方式取决于容器类型。对某些容器，比如 **IndexedContainer**，象上面的例子那样，你需要定义其中包含的属性(列)，其他容器则会自行判定。使用 `addContainerProperty()` 方法定义属性，需要指定唯一的属性 ID，类型，以及默认值。你也可以指定默认值为 `null`。

Vaadin 中有几个内建的内存容器实现，比如 **IndexedContainer** 和 **BeanItemContainer**，这些实现比较容易使用，因为不依赖于持久化的数据存储。对于持久数据，可以使用内建容器 **SQLContainer**，也可以使用 add-on 容器 **JPAContainer**。

添加项目与访问属性

可以使用 `addItem()` 方法向容器中添加项目。这个方法的无参数版本会自动生成项目 ID。

```
// Create an item
Object itemId = container.addItem();
```

要取得容器中的属性，可以首先使用 `getItem()` 方法得到项目，然后使用项目的 `getItemProperty()` 方法得到属性。

```
// Get the item object
Item item = container.getItem(itemId);

// Access a property in the item
Property<String> nameProperty =
    item.getItemProperty("name");

// Do something with the property
nameProperty.setValue("box");
```

你也可以使用 `getContainerProperty()` 方法，通过项目 ID 和属性 ID 直接得到属性。

```
container.getContainerProperty(itemId, "volume").setValue(5.0);
```

通过指定的 ID 来添加项目

某些容器，比如 **IndexedContainer** 和 **HierarchicalContainer**，允许使用指定的项目 ID 来添加项目，项目 ID 可以是任意的 **Object**。

```
Item item = container.addItem("agivenid");
item.getItemProperty("name").setValue("barrel");
Item.getItemProperty("volume").setValue(119.2);
```

注意，方法参数中给出的 不是 真实的项目，而只是项目的 ID，因为容器接口假定容器本身负责它包含的所有项目。某些容器的实现类会提供方法来添加外部创建的项目，它们甚至还可能会假定项目 ID 对象就是项目本身。Lazy 容器可能不会立即创建项目对象，而是延后到通过项目 ID 访问项目对象时创建。

8.5.2. 容器的下级接口

Container 接口中包含了一些下级接口，容器的实现类可以实现这些接口，来实现容器数据的表现组件所需要的各种功能。

Container.Filterable

可过滤容器允许使用过滤器来过滤容器内包含的项目，详情请参见第 8.5.7 节“**Filterable 容器**”。

Container.Hierarchical

层级容器允许表达项目与项目之间的层级关系，**Tree** 和 **TreeTable** 组件依赖于这个接口。**HierarchicalContainer** 是内建的内存容器，用于存储层级数据，也被用作 Tree 组件的默认容器。**FilesystemContainer** 可用于访问文件系统的内容。**JPAContainer** 也是层级式的，详情请参见第 19.4.4 节“层级数据容器”。

Container.Indexed

索引容器允许使用索引序号来访问项目，而不仅仅是使用项目 ID。某些组件需要这个功能，尤其是 **Table**，它需要实现对大容器的 lazy 访问。**IndexedContainer** 是这个接口的基本的内存实现，详情请参见第 8.5.3 节“**IndexedContainer**”。

Container.Ordered

有序容器允许在正向或反向上连续地遍历各个项目。大多数内建容器都是有序的。

Container.SimpleFilterable

这个接口允许使用 `addContainerFilter()` 方法指定的字符串来过滤容器内容。过滤方式可以是在属性值内的任何位置搜索指定的字符串，也可以只搜索字符串前缀。

Container.Sortable

某些允许对内容排序的组件需要使用可排序容器，比如 **Table**，当用户点击列头时会按照这一列来排序 Table 内的数据。某些其他组件，比如 **Calendar**，可能要求容器内容是以排序过的，才能保证正确显示。根据实现类的不同，可以只在 `sort()` 方法被调用时进行排序，也可能由容器自动保证内容的顺序，排序方式与 `sort()` 方法最后一次被调用时指定的排序方式一致。

这些接口的详情，请参见相应的 API 文档。

8.5.3. IndexedContainer

IndexedContainer 是一个内存容器，实现了 **Indexed** 接口，因此可以使用索引序号来访问项目。**IndexedContainer** 在 Vaadin 的大多数选择组件中会被用作默认容器。

属性需要使用 `addContainerProperty()` 方法来定义，这个方法的参数是属性 ID，类型，以及默认值。在向容器添加项目之前，必须先完成属性定义。

```
// Create the container
IndexedContainer container = new IndexedContainer();
```

```

// Define the properties (columns)
container.addContainerProperty("name", String.class, "noname");
container.addContainerProperty("volume", Double.class, -1.0d);

// Add some items
Object content[][] = {{"jar", 2.0}, {"bottle", 0.75},
                      {"can", 1.5}};
for (Object[] row: content) {
    Item newItem = container.getItem(container.addItem());
    newItem.getItemProperty("name").setValue(row[0]);
    newItem.getItemProperty("volume").setValue(row[1]);
}

```

新项目使用 `addItem()` 方法添加到容器中, 返回值是新项目的 ID, 也可以象前文介绍过的那样, 通过参数指定项目 ID 来添加项目. 注意, **Table** 组件, 使用 **IndexedContainer** 作为它的默认容器, 它有一个名为 `addItem()` 的方法, 这个方法可以使用对象 vector 来添加项目, vector 内容是项目各个属性的值.

实现了 `Container.Indexed` 功能的容器, 可以使用 `getIdByIndex()` 方法, 通过项目的索引序号来访问项目 ID. 索引功能主要是为满足某些组件的内部目的, 比如 **Table** 组件, 它需要使用索引功能来将 Table 数据以 lazy 模式传送到客户端.

8.5.4. BeanContainer

BeanContainer 是一个保存 Java Bean 对象的内存容器. 其中包含的每一个 Bean 都被封装在 **BeanItem** 之内. 项目属性通过检查类的 `get` 方法和 `set` 方法来自动判定. 因此要求 Bean 类的可见度为 `public`, `local` 类是不可使用的. 只有相同类型的 Bean 可以添加到这个容器中.

这个容器的泛型有两个参数: Bean 类型和项目 ID 类型. 项目 ID 可以通过指定一个自定义解析器来获得, 也可以使用特定的项目属性作为 ID, 或者明确给定一个项目 ID. 因此, 这个容器比 **BeanItemContainer** 更通用一些, `BeanItemContainer` 使用 Bean 对象本身作为项目 ID, 但是使用起来更简单一些. 管理项目 ID 使得 **BeanContainer** 的使用更复杂, 但在某些情况下这种工作是必须的, 比如 Bean 的 `equals()` 或 `hashCode()` 方法被重新实现的情况.

```

// Here is a JavaBean
public class Bean implements Serializable {
    String name;
    double energy; // Energy content in kJ/100g

    public Bean(String name, double energy) {
        this.name = name;
        this.energy = energy;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getEnergy() {
        return energy;
    }

    public void setEnergy(double energy) {
        this.energy = energy;
    }
}

```

```

    }

}

void basic(VerticalLayout layout) {
    // Create a container for such beans with
    // strings as item IDs.
    BeanContainer<String, Bean> beans =
        new BeanContainer<String, Bean>(Bean.class);

    // Use the name property as the item ID of the bean
    beans.setBeanIdProperty("name");

    // Add some beans to it
    beans.addBean(new Bean("Mung bean", 1452.0));
    beans.addBean(new Bean("Chickpea", 686.0));
    beans.addBean(new Bean("Lentil", 1477.0));
    beans.addBean(new Bean("Common bean", 129.0));
    beans.addBean(new Bean("Soybean", 1866.0));

    // Bind a table to it
    Table table = new Table("Beans of All Sorts", beans);
    layout.addComponent(table);
}

```

如果要明确指定项目 ID, 应该使用 addItem(Object, Object), addItemAt(int, Object, Object), 和 addItemAfter(Object, Object, Object) 方法.

除 Bean 的属性之外, 容器不允许添加额外的属性, 除非是嵌套 Bean 的属性.

嵌套属性

如果你在 **BeanContainer** 或 **BeanItemContainer** 包含的 Bean 类型之内还有 1:1 关系的嵌套 Bean, 你可以将嵌套 Bean 的属性添加到容器中, 方法是使用 `addNestedContainerProperty()` 方法. 这个功能定义在 **AbstractBeanContainer** 基类中.

和 Bean 容器中的 Bean 一样, 嵌套 Bean 的可见度也必须是 public, 否则会抛出访问异常. 从 Bean 容器中的 Bean 到嵌套 Bean 的直接引用, 可以是 null 值.

比如, 假定我们有下面两个 Bean, 第一个嵌套在第二个之中.

```

/** Bean to be nested */
public class EqCoord implements Serializable {
    double rightAscension; /* In angle hours */
    double declination; /* In degrees */

    ... setters and getters for the properties ...
}

/** Bean referencing a nested bean */
public class Star implements Serializable {
    String name;
    EqCoord equatorial; /* Nested bean */

    ... setters and getters for the properties ...
}

```

创建完容器后, 你可以使用 `addNestedContainerProperty()` 方法来声明嵌套属性, 属性 ID 中使用点号分隔.

```
// Create a container for beans
BeanItemContainer<Star> stars =
    new BeanItemContainer<Star>(Star.class);

// Declare the nested properties to be used in the container
stars.addNestedContainerProperty("equatorial.rightAscension");
stars.addNestedContainerProperty("equatorial.declination");

// Add some items
stars.addBean(new Star("Sirius", new EqCoord(6.75, 16.71611)));
stars.addBean(new Star("Polaris", new EqCoord(2.52, 89.26417)));

// Here the nested bean reference is null
stars.addBean(new Star("Vega", null));
```

如果你将这样一个容器绑定到 **Table** 组件, 你可能还需要设置列头. 注意, 嵌套 Bean 本身仍然是容器中的一个属性, 因此也会显示在它自己的列中. 嵌套 Bean 的 `toString()` 方法会被用来取得这个属性的显示值, 它的默认结果是一个对象引用地址. 通常你不会希望这样一个结果, 因此你可以使用 `setVisibleColumns()` 方法来隐藏这个列.

```
// Put them in a table
Table table = new Table("Stars", stars);
table.setColumnHeader("equatorial.rightAscension", "RA");
table.setColumnHeader("equatorial.declination", "Decl");
table.setPageLength(table.size());

// Have to set explicitly to hide the "equatorial" property
table.setVisibleColumns(new Object[]{"name",
    "equatorial.rightAscension", "equatorial.declination"});
```

Table 的运行结果见图 8.4 “使用嵌套属性绑定到 **BeanContainer** 的 **Table** 组件”.

图 8.4. 使用嵌套属性绑定到 **BeanContainer** 的 **Table** 组件

name	RA	Decl
Sirius	6,75	16,716
Polaris	2,52	89,264
Vega		

绑定到 **AbstractBeanContainer** 的 Bean, 通常使用 **Property** 接口的 **MethodProperty** 实现类, 来通过 Bean 的 get 方法和 set 方法访问 Bean 的属性. 对于嵌套属性, 使用的是 **NestedMethodProperty** 实现类.

定义一个 Bean ID 解析器

如果使用 `setBeanIdResolver()` 或 `setBeanIdProperty()` 方法设置了 Bean ID 解析器, 那么就可以使用 `addBean()`, `addBeanAfter()`, `addBeanAt()` 和 `addAll()` 方法来向容器添加项目. 如果调用了这些方法之一, Bean ID 解析器会被用来为项目生成 ID(返回的项目必须不是 null).

注意, 即使设置了项目 ID 解析器, 也可以使用明确指定的项目 ID 来调用 `addItem*`() 方法 - 项目 ID 解析器只被用在使用 `addBean*`() 或 `addAll(Collection)` 方法添加 Bean 的场合.

8.5.5. BeanItemContainer

BeanItemContainer 是一个 Java Bean 对象的容器, 其中的每个 Bean 都封装在 **BeanItem** 之内. 项目属性通过检查类的 get 方法和 set 方法来自动判定. 因此要求 Bean 类的可见度为 public, local 类是不可使用的. 只有相同类型的 Bean 可以添加到这个容器中.

BeanItemContainer 是 **BeanContainer** 的一个特殊版本, 关于 **BeanContainer** 请参见第 8.5.4 节 “**BeanContainer**”. 这个容器使用 Bean 本身作为项目 ID, 因此在很多场合, 它的使用比 **BeanContainer** 要简单一些. 但如果 Bean 类重新实现了 `equals()` 或 `hashCode()` 方法的话, 就需要使用 **BeanContainer**.

我们再来看看第 8.5.4 节 “**BeanContainer**” 中的例子, 现在我们改为使用 **BeanItemContainer**.

```
// Create a container for the beans
BeanItemContainer<Bean> beans =
```

```

new BeanItemContainer<Bean>(Bean.class);

// Add some beans to it
beans.addBean(new Bean("Mung bean", 1452.0));
beans.addBean(new Bean("Chickpea", 686.0));
beans.addBean(new Bean("Lentil", 1477.0));
beans.addBean(new Bean("Common bean", 129.0));
beans.addBean(new Bean("Soybean", 1866.0));

// Bind a table to it
Table table = new Table("Beans of All Sorts", beans);

```

除 Bean 的属性之外, **BeanItemContainer** 容器不允许添加额外的属性, 除非是嵌套 Bean 的属性, 关于嵌套属性, 详情请参见第 8.5.4 节 “**BeanContainer**”.

8.5.6. 在容器内遍历

由于 **Container** 内的项目不一定是有索引的, 因此对项目的遍历必须使用 **Iterator**. **Container** 的 `getItemIds()` 方法返回一个项目 ID 的 **Collection**, 你可以在这个 Collection 上进行遍历. 下例演示了一种常见的遍历情况, 这里遍历 **Table** 组件的一列中的 CheckBox 的值. 这段示例程序的使用场景, 对应于第 5.21 节 “**Table**” 中的示例.

```

// Collect the results of the iteration into this string.
String items = "";

// Iterate over the item identifiers of the table.
for (Iterator i = table.getItemIds().iterator(); i.hasNext();) {
    // Get the current item identifier, which is an integer.
    int iid = (Integer) i.next();

    // Now get the actual item from the table.
    Item item = table.getItem(iid);

    // And now we can get to the actual checkbox object.
    Button button = (Button)
        item.getItemProperty("ismember").getValue();

    // If the checkbox is selected.
    if (((Boolean)button.getValue()) == true) {
        // Do something with the selected item; collect the
        // first names in a string.
        items += item.getItemProperty("First Name")
            .getValue() + " ";
    }
}

// Do something with the results; display the selected items.
layout.addComponent (new Label("Selected items: " + items));

```

注意, `getItemIds()` 方法返回的是不可变更的 **Collection**, 因此 **Container** 在遍历过程中不可以修改内容. 比如, 你不可以在遍历过程中从 **Container** 内删除项目. 这里所说的修改, 也包括在其他线程内的修改. 如果 **Container** 在遍历过程中被修改, 会抛出 **ConcurrentModificationException** 异常, iterator 会处于一种无效状态.

8.5.7. Filterable 容器

实现了 **Container.Filterable** 接口的容器是可以过滤的。比如，内建的 **IndexedContainer** 和 Bean 项目容器都实现了这个接口。过滤功能通常用于过滤 **Table** 中的内容。

过滤器实现 **Filter** 接口，你需要使用 `addContainerFilter()` 方法，将过滤器添加到一个可过滤的容器中。容器中的项目，如果通过了过滤条件，会被保留并显示在可过滤的组件中。

```
Filter filter = new SimpleStringFilter("name",
    "Douglas", true, false);
table.addContainerFilter(filter);
```

如果容器中添加了多个过滤器，它们之间会用逻辑 AND 的方式组合起来，因此只有通过了所有过滤器的项目才会被保留下。

原子过滤器与组合过滤器

过滤器可以分为 原子 和 组合 两类。原子过滤器，比如 **SimpleStringFilter**，定义一个单独的过滤条件，通常应用与一个特定的容器属性。组合过滤器则根据一个或多个其他过滤器的结果来决定它的过滤结果。内建的组合过滤器实现了逻辑 AND, OR, 和 NOT 操作。

比如，以下组合过滤器会将 name 属性中包含 "Douglas" 或者 age 属性值小于 42 的项目排除(译注：这段说明与示例程序不符)。这两个属性的类型应该分别是 **String** 和 **Integer**。

```
filter = new Or(new SimpleStringFilter("name",
    "Douglas", true, false),
    new Compare.Less("age", 42));
```

内建的过滤器类型

内建的过滤器类型如下：

SimpleStringFilter

项目中某个指定的属性，类型必须是 **String**，如果属性值中包含指定的 **□□□□□**，那么这个项目将通过过滤。如果 `ignoreCase` 参数为 `true`，那么字符串查询是忽略大小写的。如果 `onlyMatchPrefix` 参数为 `true`，那么只会在属性值的前缀部分查找子字符串，如果这个参数为 `false`，那么子字符串也可以出现在其他位置。

IsNull

项目的指定属性值为 `null` 时，这个项目将通过过滤。对于内存内的过滤，只执行一个简单的 `==` 判断。对其他容器，比较的具体实现方法由各容器分别决定，但结果应该与内存内的 `null` 检查逻辑一致。

Equal, Greater, Less, GreaterOrEqual, and LessOrEqual

比较过滤器将指定的属性的值与指定的常数进行比较，如果比较结果为 `true`，则这个项目将通过过滤。比较操作包含在 **Compare** 抽象类中。

对于 **Equal** 过滤器，内建的内存容器会使用属性的 `equals()` 方法进行比较。其他类型的容器，比较方法由各容器决定，比如，可能会使用数据库的比较操作。

对于其他过滤器，属性值的类型必须实现 **Comparable** 接口，才可以在内建的内存容器中工作。对其他类型的容器，比较操作的具体实现由各容器决定。

And 和 Or

这些逻辑操作过滤器是组合过滤器，可以将多个其他过滤器的结果组合起来。

Not

这个逻辑非过滤器接受一个过滤器参数，逻辑非过滤器会将参数中指定的过滤器的结果反转。

实现自定义过滤器

自定义过滤器需要实现 **Container.Filter** 接口。

过滤器可以在过滤逻辑中使用一个或多个属性。`appliesToProperty()` 方法必须判定一个属性是否是过滤器使用到的属性。如果过滤器适用的目标属性是由使用者指定的，那么通常的习惯是将这些属性作为过滤器构造函数的第一个参数。

```
class MyCustomFilter implements Container.Filter {
    protected String propertyId;
    protected String regex;

    public MyCustomFilter(String propertyId, String regex) {
        this.propertyId = propertyId;
        this.regex      = regex;
    }

    /** Tells if this filter works on the given property. */
    @Override
    public boolean appliesToProperty(Object propertyId) {
        return propertyId != null &&
               propertyId.equals(this.propertyId);
    }
}
```

实际的过滤逻辑在 `passesFilter()` 方法中实现，如果项目应该通过过滤的话，这个方法返回 `true`，如果项目应该被排除的话，这个方法返回 `false`。

```
/** Apply the filter on an item to check if it passes. */
@Override
public boolean passesFilter(Object itemId, Item item)
    throws UnsupportedOperationException {
    // Acquire the relevant property from the item object
    Property p = item.getItemProperty(propertyId);

    // Should always check validity
    if (p == null || !p.getType().equals(String.class))
        return false;
    String value = (String) p.getValue();

    // The actual filter logic
    return value.matches(regex);
}
```

你可以使用这个自定义过滤器，方法与使用其他过滤器一样：

```
c.addContainerFilter(  
    new MyCustomFilter("Name", (String) tf.getValue()));
```

Vaadin SQLContainer

9.1. 架构	306
9.2. SQLContainer 入门	306
9.3. 过滤与排序	307
9.4. 编辑	308
9.5. 缓存, 分页和刷新	309
9.6. 刷新其他 SQLContainer	311
9.7. 使用任意查询	311
9.8. 未实现的方法	312
9.9. 已知的问题与限制事项	313

Vaadin SQLContainer 是一个容器实现类, 可以便利地存取各种 SQL 数据库中的数据, 也可以自由定制。

SQLContainer 支持两种类型的数据库存储. **TableQuery** 是预定义的查询产生器, 可以读取, 更新, 以及直接从容器向数据库表自动地插入数据, 而 **FreeformQuery** 允许开发者使用自己决定的查询, 可以是用于读取数据的更加复杂的查询, 也可以使用自定义的数据库写入语句, 自定义的过滤和排序 - 但项目和属性管理, 以及 lazy load 管理仍然是自动进行的.

除了自定义的数据库连接选项外, SQLContainer 还扩展了 Vaadin **Container** 接口, 实现更复杂, 更面向数据库的过滤规则. 最后, 这个 add-on 还提供了连接池的实现, 包括 JDBC 连接池和 JEE 连接池, 以及集成的事务支持; 另外还提供了自动 commit 模式.

本节的目的是简要介绍 SQLContainer 的整体架构以及它的部分内部工作原理. 本节还会向读者解释如何在自己的应用程序中使用 SQLContainer. 此外还讨论了 SQLContainer 的要求, 限制以及将来的开发思路.

SQLContainer 可以从 Vaadin Directory 得到, 使用的许可协议与 Vaadin Framework 一样, 是无限制的 Apache License 2.0 协议.

9.1. 架构

SQLContainer 的架构相对简单. **SQLContainer** 是 Vaadin **Container** 接口的实现类, 这个 add-on 的主要功能都由这个类提供. 标准的 Vaadin **Property** 和 **Item** 接口由 **ColumnProperty** 和 **RowItem** 类实现. 项目 ID 由 **RowId** 和 **TemporaryRowId** 类实现. **RowId** 类根据数据库表或查询接口中的主键字段来构建.

在 connection 包中, **JDBCConnectionPool** 接口定义了数据库连接池需要实现的功能. 本 add-on 提供了两种实现: **SimpleJDBCCConnectionPool** 是一个简单但有用的 JDBC 连接池实现. **J2EEConnectionPool** 是 J2EE DataSource 的连接池实现.

query 包中包含 **QueryDelegate** 接口, 定义了 SQLContainer 应该实现的数据库读写功能. 如前文所说, 本 add-on 提供了这个接口的两个实现: **TableQuery** 可用于自动读写数据库表, **FreeformQuery** 可以定制查询, 排序, 过滤以及更新的各种语句; 这些都是通过实现 **FreeformStatementDelegate** 接口的相关方法来实现的.

query 包中也包含 **Filter OrderBy** 类, 用于代替标准的 Vaadin 容器过滤器, 并使得非字符串属性的排序功能更易用一些.

最后, generator 包中包含 **SQLGenerator** 接口, 其中定义了 **TableQuery** 所要求的查询类型. 本 add-on 中提供的实现类支持 HSQLDB, MySQL, PostgreSQL (**DefaultSQLGenerator**), Oracle (**OracleGenerator**) 以及 Microsoft SQL Server (**MSSQLGenerator**). 将来可能会提供新的或修改过的实现以便实现对旧版本数据库或其他类型数据库的支持.

更多细节, 请参照 SQLContainer 的 API 文档.

9.2. SQLContainer 入门

使用 SQLContainer 进行开发是很简单直接的. 本节的目的是介绍如何创建需要的资源, 以及如何从绑定到容器的数据库表来读写数据.

9.2.1. 创建连接池

首先, 我们需要创建连接池, 供 SQLContainer 连接到数据库. 这里我们使用 **SimpleJDBCCConnectionPool**, 它是连接池的一个基本实现, 使用 JDBC 数据源. 下面的代码中, 我们创建一个连接池, 使用 HSQLDB 驱动, 以及一个 HSQLDB 内存数据库. 连接数的初期值是 2, 最大数设置为 5. 注意, 数据库驱动, 连接 URL, 用户名, 密码等参数, 在你的实际环境中可能是不同的.

```
JDBCConnectionPool pool = new SimpleJDBCCConnectionPool(
    "org.hsqldb.jdbc.JDBCDriver",
    "jdbc:hsqldb:mem:sqlcontainer", "SA", "", 2, 5);
```

9.2.2. 创建查询代理 **TableQuery**

连接池创建之后, 我们需要为 SQLContainer 创建查询代理. 最简便的方法是使用内建的 **TableQuery** 类. **TableQuery** 代理, 可访问数据库中的一个固定的表, 可直接用来读写数据, 不需要太多的定制

工作. 表的主键可以是数据库引擎支持的任何类型, 而且会在创建 **TableQuery** 实例来查询数据库时自动发现主键信息. 我们使用以下语句来创建 **TableQuery**:

```
TableQuery tq = new TableQuery("tablename", connectionPool);
```

为了支持多个用户 session 的并发写入, 我们还必须为 **TableQuery** 设置版本控制列(version column). 版本控制列是整数型, 或时间戳(timestamp)型的列, 每次变更这行数据时, 版本控制列的值会自动增长(整数型)或被设置为现在时刻(时间戳型). **TableQuery** 假定数据库负责更新版本控制列; 它只会在自己更新数据之前检验版本控制列的值是否正确. 如果另一个用户已经修改过这行数据, 数据库中的版本控制列值与内存中记录的原始值不符, 会抛出 **OptimisticLockException** 异常, 这时你可以刷新容器来恢复数据, 然后让用户合并数据. 以下代码设置版本控制列:

```
tq.setVersionColumn("OPTLOCK");
```

9.2.3. 创建容器

最后, 我们可以创建容器本身了. 代码很简单:

```
SQLContainer container = new SQLContainer(tq);
```

使用以上代码, **SQLContainer** 连接到数据库表 "tablename" 上, 可以用作 Vaadin 组件的数据源了, 比如 **Table** 或 **Form**.

9.3. 过滤与排序

SQLContainer 中项目的过滤和排序, 按照 **SQLContainer** 的设计意图, 永远是在数据库内实现的. 实际运用中, 这意味着无论何时, 只要过滤或排序规则发生变化, 就一定会发生一定量的数据库通信(最少需要使用新的过滤/排序规则取得新的数据行数).

9.3.1. 过滤

过滤使用 Vaadin 的过滤 API 实现, 这个 API 可以用简单的方式实现非常复杂的过滤. 关于过滤 API, 详情请参见 第 8.5.7 节 “**Filterable** 容器”.

除 Vaadin 提供的过滤器外, **SQLContainer** 还实现了 **Like** 和 **Between** 过滤器. 这两个过滤器都对应到 SQL 语句中的同名 WHERE 操作. 过滤器也可以应用于内存中的项目, 比如, 还未写入数据库的新项目, 或者被装载入内存后被修改过但还未保存的行.

下例演示新的过滤 API 中, 可以使用的各种复杂过滤. 我们希望找出所有全名为 "Paul Johnson", 并且年龄在 18 岁以下, 或 65 岁以上的人, 以及所有姓 "Johnsons", 并且名以字母 "A" 开头的人:

```
mySQLContainer.addContainerFilter(
    new Or(new And(new Equal("NAME", "Paul"),
                  new Or(new Less("AGE", 18),
                         new Greater("AGE", 65))),
           new Like("NAME", "A%")));
mySQLContainer.addContainerFilter(
    new Equal("LASTNAME", "Johnson"));
```

以上代码产生的 WHERE 查询条件为:

```
WHERE ((NAME = "Paul" AND (AGE < 18 OR AGE > 65)) OR NAME LIKE "A%")
AND LASTNAME = "Johnson"
```

9.3.2. 排序

排序可以使用标准的 Vaadin API 实现, 也就是使用 **Container.Sortable** 接口的 sort 方法. 其中的 *propertyId* 参数指定排序的列名.

```
public void sort(Object[] propertyId, boolean[] ascending)
```

除上面的标准方法外, 还可以使用 addOrderBy() 方法, 直接向容器添加一个 **OrderBy**. 这个方法允许开发者逐个添加排序列, 而不必以数组的方式一次性指定全部排序列.

使用 null 或空数组作为第一个参数来调用 sort 方法, 可以清除所有的排序规则.

9.4. 编辑

编辑 SQLContainer 中的项目(**RowItems**) 与编辑其他 Vaadin 容器的项目类似. **RowItem** 的 **ColumnProperties** 会自动通知 SQLContainer, 以确保项目的变更被正确记录下来, 并被保存到数据库中. 向数据库的保存可以是立即的, 也可能只在 commit 时保存, 具体如何由自动 commit 模式的设置决定.

9.4.1. 添加项目

向 **SQLContainer** 添加项目只能使用 addItem() 方法. 这个方法将创建一个新的 **Item** 对象, 其中的属性由数据库表的列属性决定. 新项目可能缓存在容器中, 也可能立即通过查询代理提交到数据库, 具体如何, 由自动 commit 模式(详情请见下节)是否打开来决定.

向容器添加项目时, 不可能精确地预知这一行的主键值是什么, 甚至无法预知这一行的插入是否成功. 所以 SQLContainer 会为新项目自动赋予一个 **TemporaryRowId** 作为 **RowId**. 我们后面会介绍, 当行插入成功后如何获取真实的主键值.

如果 **SQLContainer** 的自动 commit 模式有效, addItem() 方法会返回新项目最终的 **RowId**.

9.4.2. 取得数据库生成的row key

获取一个新插入的行的自动生成主键值, 这是一种常见的需求, 因此 **QueryDelegate** 接口为解决这个问题键入了一组监听器/通知器. 目前只有 **TableQuery** 类实现了 **RowIdChangeNotifier** 接口, 因此它可以发送行 ID 的变化通知. 在 **TableQuery** 的 commit() 方法完成后会激发这个事件; commit 方法会在必要的时候由 **SQLContainer** 调用.

要接收 row ID 的更新事件, 你可以使用以下代码(假定容器是 **SQLContainer** 的实例). 注意, 在自动 commit 模式下不会激发这些事件.

```
app.getDbHelp().getCityContainer().addListener(
    new QueryDelegate.RowIdChangeListener() {
        public void rowIdChange(RowIdChangeEvent event) {
            System.out.println("Old ID: " + event.getOldRowId());
            System.out.println("New ID: " + event.getNewRowId());
        }
    });
}
```

9.4.3. 对版本控制列的要求

如果你使用 **TableQuery** 类作为 **SQLContainer** 的查询代理, 而且希望支持数据写入功能, 这种情况下有一个强制要求, 必须为 **TableQuery** 指定版本控制列的名称. 为 **TableQuery** 指定版本控制列名称的代码如下:

```
tq.setVersionColumn("OPTLOCK");
```

版本控制列最好是**TableQuery** 连接的表中的整数型或时间戳型列. 这个列会被用做乐观锁; 修改某个行之前, **TableQuery**会检查版本控制列的值与数据读入容器时的值是否一致. 这样可以保证, 在当前用户的读操作和写操作之间, 没有其他用户修改过同一行数据.

注意! **TableQuery** 假定数据库会负责更新版本控制列的值, 方法可以是使用一个真实的 VERSION 列(前提是数据库支持这个功能), 也可以是使用触发器或者类似机制.

如果你很确定不需要乐观锁, 但是又希望支持数据写入功能, 你可以将版本控制列指定为, 举例来说, 表的主键列.

9.4.4. 自动 Commit 模式

SQLContainer 默认使用事务模式, 也就是说编辑, 添加或删除操作会被记录在容器内部. 可以调用 `commit()` 方法将这些操作提交到数据库中, 也可以调用 `rollback()` 方法废弃这些操作.

容器也可以设置为自动 commit 模式. 这种模式下, 一切变更都会被立即自动提交到数据库. 要打开或关闭自动 commit 模式, 请调用以下方法:

```
public void setAutoCommit(boolean autoCommitEnabled)
```

我们建议关闭自动 commit 模式, 因为这样可以保证, 如果发现容器中的项目有任何问题, 都可以随时取消对数据的修改. 如果数据库表中包含非 NULL 列, 使用自动 commit 模式还会导致增加项目失败.

9.4.5. 更新状态

在事务模式下使用时, 可能需要判断 **SQLContainer** 的内容是否有变化. 为了实现这个目的, 容器提供了 `isModified()` 方法, 它可以向开发者报告容器的状态. 如果有新项目添加到容器中, 或有项目从容器中删除, 或某个已有的项目的值有任何变化, 这个方法都将返回 `true`.

此外, 每个 **RowItem** 和每个 **ColumnProperty** 也有 `isModified()` 方法, 可用来判断更细节的变更状态. 注意, **RowItem** 和 **ColumnProperty** 对象的变更状态只取决于实际的 **Property** 值有没有变更. 也就是说, 它们不会反映整个 **RowItem** 是已被删除的行, 或者是新插入容器的行.

9.5. 缓存, 分页和刷新

为了减少向数据库发起的查询次数, **SQLContainer** 使用内部缓存来保存数据库内容. 缓存使用有限尺寸的 **LinkedHashMap** 来实现, 其中包含一个从 **RowId** 到 **RowItem** 的映射. 开发者一般不必修改缓存选项, 但如果有必要, 也可以做一些性能调整.

9.5.1. 容器大小

SQLContainer 会不断检查它所连接的数据表内的行数, 以便判断行的增加和删除. 默认情况下, 表行数的有效时间假定为 10 秒. 这个值可以通过 **SQLContainer** 的 `setSizeValidMilliseconds()` 方法来修改.

如果大小的有效时间到期后, 会在以下几个时刻自动更新行数:

- 调用 `getItemIds()` 方法时
- 调用 `size()` 方法时
- 调用 `indexOfId(Object itemId)` 方法时(某些情况下)

- 调用 `firstItemId()` 方法时
- 当容器读取一系列的行进入项目缓存(lazy loading)时

9.5.2. 分页长度与缓存大小

SQLContainer 的分页长度 决定每次查询数据库时读取的数据行数. 默认值是 100, 这个值可以使用 `setPageLength()` 方法修改. 为了避免发生大量查询, 建议将分页长度设置为 Vaadin **Table** 组件中显示行数的至少 5 倍; 显然, 这个设置也取决于 **Table** 组件的缓存比例设置.

SQLContainer 内部项目缓存的大小的计算方法如下: 分页长度 × 容器设置的缓存比率. 缓存比率只能通过代码来设置, 默认值为 2. 因此, 对于默认的分页长度 100, 内部缓存大小为 200 个项目. 这个值即使对于更大的 **Table** 也足够了, 同时还可以保证缓存不会消耗掉过多的内存.

9.5.3. 刷新容器

通常, **SQLContainer** 会在需要的时候自动处理刷新工作. 但是, 某些情况下会需要手动刷新, 比如, 在 Form 中打开项目进行编辑之前, 为了确保版本控制列的值保持同步, 可能就需要刷新容器数据. 对于这里目标, 容器提供了 `refresh()` 方法. 这个方法简单地清空所有缓存, 重置项目取得位置的偏移量, 然后将容器大小设为无效. 此后一切与项目相关的调用都会不可避免地导致更新行件数和更新项目缓存.

注意, 调用 `refresh` 方法不会影响或重置容器的以下属性:

- 容器的 **QueryDelegate**
- 自动 commit 模式
- 分页长度
- 过滤器
- 排序

9.5.4. 缓存 Flush 通知机制

在多用户应用程序中使用缓存, 总是会导致某种矛盾, 我们希望在数据库上执行的查询次数要少, 我们又希望用来缓存数据的内存消耗要少; 还有最重要的问题, 就是缓存中的数据过期的风险要小.

SQLContainer 为这个问题提供了一种实验性的补救方法, 它实现了一个简单的缓存 Flush 通知机制. 由于这些通知的性质, 它们默认是关闭的, 但在容器生命周期的任何时刻, 都可以通过 `enableCacheFlushNotifications()` 方法启用这些事件.

通知机制在一个 static 的 List 结构中对所有注册的容器保存一个弱引用. 为了将内存泄露的风险降到最低, 也为了避免引用 List 的无限增长, 已消亡的弱引用会被收集到一个引用队列中, 然后在新的 **SQLContainer** 加入通知列表时, 或者在某个容器调用通知方法时, 将已消亡的引用从列表中删除.

当 **SQLContainer** 的缓存通知被设置为有效, 它会调用静态方法 `notifyOfCacheFlush()`, 并把自己作为这个方法的参数. 这个方法会比较发起通知的容器, 以及引用列表中所有其他容器. 为了激发一个缓存 Flush 事件, 目标容器的 **QueryDelegate** 类型必须相同(可以是 **TableQuery** 或 **FreeformQuery**), 而且表名或查询语句必须与发起通知的容器一致. 如果发现了一致的容器, 对象容器的 `refresh()` 方法会被调用, 于是导致源容器的更新被刷新到目标容器中.

注意: Vaadin 的常见问题也适用于这个容器: 即使在服务器端刷新了 **SQLContainer**, 变化也不会反映到 UI 部分, 除非发生了一次与服务器的同步, 或者使用了某种服务器 PUSH 机制.

9.6. 刷新其他 **SQLContainer**

开发数据库应用程序时, 有一种常见需求是从一个或多个其他表中取得一个表的相关数据. 大多数情况下, 这种关系通过外键引用来实现, 也就是, 一个表的列中包含另一个表主键, 或候选键(candidate key, 译注: 不是主键, 但也是唯一键, 因此也可以被其他表的外键来引用).

对于这样的引用关系, **SQLContainer** 提供了有限的支持, 虽然所有的引用目前都是在 Java 端完成的, 因此数据库端不必创建约束. 创建引用关系可以使用以下方法:

```
public void addReference(SQLContainer refdCont,
                        String refingCol, String refdCol);
```

这个方法应该在引用关系的源容器端调用. 目标容器作为第一个参数给出. *refingCol* 参数是源容器中"外键"列的名称, *refdCol* 参数是目标容器中被引用的主键或候选键的列名称.

注意: 对于任何的 **SQLContainer**, 它所引用的目标容器必须是不同的. 不允许从同一个源容器两次引用同一个目标容器.

管理被引用的项目可以通过三个 set/get 方法, 完全删除引用关系可以使用 removeReference() 方法. 这些方法如下:

```
public boolean setReferencedItem(Object itemId,
                                  Object refdItemId, SQLContainer refdCont)
public Object getReferencedItemId(Object itemId,
                                   SQLContainer refdCont)
public Item getReferencedItem(Object itemId,
                               SQLContainer refdCont)
public boolean removeReference(SQLContainer refdCont)
```

set 方法需要三个参数: *itemId* 是(源容器中)引用项目的 ID, *refdItemId* 是(目标容器中的)被引用 ID, *refdCont* 是目标容器. 如果引用关系的设置成功, 这个方法将返回 true. 设置过引用关系后, 你必须象通常的数据变更一样, 在源容器上调用 commit() 方法, 将数据写入数据库中.

getReferencedItemId() 方法返回被引用项目的 ID. 这个方法的参数需要指定引用源项目的 ID, 以及被引用的目标容器. **SQLContainer** 还提供了一个简易方法 *getReferencedItem()*, 直接返回目标容器中的被引用项目, 而不是被引用项目的 ID.

最后, 在源容器上调用 *removeReference()* 方法, 以目标容器为参数, 可以删除它们之间的引用关系. 注意这个方法不会在数据库端造成任何改变; 它只是在 Java 端删除容器间的逻辑关系.

9.7. 使用任意查询

大多数情况下, 内建的 **TableQuery** 已经足够开发者便利地实现对 SQL 数据源的访问. 但是, 还存在很多情况下可能需要使用更复杂的查询, 比如, 需要使用 join 查询. 或者你可能会需要自行实现如何写入数据, 如何过滤数据. 查询代理类 **FreeformQuery** 就被用来实现这样的需求. 未经复杂配置的 **FreeformQuery** 可以实现对数据库的只读访问, 但经过一些扩展后, 也可以实现数据写入功能.

入门

FreeformQuery 的简单实用见下例. 数据库连接池的初始化处理与 **TableQuery** 中的示例程序类似, 因此此处省略. 注意, 主键的列名必须使用手工指定给 **FreeformQuery**. 这个要求是因为,

由于查询语句的不同, 查询结果集中可能包含主键列, 也可能不包含。下例中, 查询结果中包含主键列, 名为 'ID'。

```
FreeformQuery query = new FreeformQuery(  
    "SELECT * FROM SAMPLE", pool, "ID");  
SQLContainer container = new SQLContainer(query);
```

限制

虽然上例看起来和 **TableQuery** 一样简单, 但这里有几个重要的问题需要注意。以这种方式使用 **FreeformQuery** (不指定 **FreeformQueryDelegate** 或 **FreeformStatementDelegate**), 只能将查询结果用于只读窗口。这之外的过滤, 排序, lazy loading 功能都将不可用, 而且行数的取得会使用一种非常低效的方式。理解这些限制之后, 我们可以很明显地看出, 开发者需要实现 **FreeformQueryDelegate** or **FreeformStatementDelegate** 接口。

FreeformStatementDelegate 接口继承自 **FreeformQueryDelegate** 接口, 它返回 **StatementHelper** 对象, 而不是单纯的查询语句字符串。因此开发者可以使用 **PreparedStatement**(译注: 此处原文有拼写错误) 而不是通常的 **Statement**。强烈建议在所有的实现中都使用 **FreeformStatementDelegate**。从本章开始, 凡是可以使用 **FreeformQueryDelegate** 的场合, 我们都使用 **FreeformStatementDelegate**。

创建自己的 **FreeformStatementDelegate**

为了给 **FreeformQuery** 创建自己的代理, 你必须实现 **FreeformStatementDelegate** 接口的一部分或全部方法, 具体如何取决于你的使用场景。这个接口包含以下八个方法。更详细的信息, 请参阅这个接口的 JavaDoc 文档。

```
// Read-only queries  
public StatementHelper getCountStatement()  
public StatementHelper getQueryStatement(int offset, int limit)  
public StatementHelper getContainsRowQueryStatement(Object... keys)  
  
// Filtering and sorting  
public void setFilters(List<Filter> filters)  
public void setFilters(List<Filter> filters,  
                      FilteringMode filteringMode)  
public void setOrderBy(List<OrderBy> orderBys)  
  
// Write support  
public int storeRow(Connection conn, RowItem row)  
public boolean removeRow(Connection conn, RowItem row)
```

作为这个接口的一个简单的样例实现, 可以参考 SQLContainer 包中的 **com.vaadin.addon.sqlcontainer.demo.DemoFreeformQueryDelegate** 类。

9.8. 未实现的方法

由于 SQLContainer 连接到数据库, Vaadin 容器接口中的某些方法是无法实现的(或者说在数据库概念中是没有意义的)。这些方法列举如下, 如果调用它们, 会抛出 **UnsupportedOperationException** 异常。

```
public boolean addContainerProperty(Object propertyId,  
                                    Class<?> type,  
                                    Object defaultValue)  
public boolean removeContainerProperty(Object propertyId)  
public Item addItem(Object itemId)
```

```

public Object addItemAt(int index)
public Item addItemAt(int index, Object newItemId)
public Object addItemAfter(Object previousItemId)
public Item addItemAfter(Object previousItemId, Object newItemId)

```

此外, **Item** 接口的以下方法, 在 **RowItem** 类中也是不支持的:

```

public boolean addItemProperty(Object id, Property property)
public boolean removeItemProperty(Object id)

```

关于 **getIds()** 方法

为了正确地实现 Vaadin **Container** 接口, 在 **SQLContainer** 中必须实现 **getIds()** 方法。按照定义, 这个方法应该返回容器中所有项目的 ID。对 **SQLContainer** 来说, 容器必须查询数据, 取得当前关联的表中所有行的主键数据。

为了实现这样的方法, 显然有可能装载几万甚至几十万行数据, 于是导致 **SQLContainer** 的 lazy loading 完全失去意义, 所以我们提出以下警告:



警告

强烈建议不要调用 **getIds()** 方法, 除非明确地知道这个方法载入的项目 ID 数量不会太多。

9.9. 已知的问题与限制事项

目前, 对于 **SQLContainer** 的某些使用场景, 仍然还存在着一些已知的问题和限制事项。已知问题及简要的解释列举如下:

- 使用 **TableQuery** 时某些 SQL 数据类型没有写入能力:
 - 所有的 **binary** 类型
 - 所有的 **custom** 类型
 - **CLOB** (如果没有被目前使用的 JDBC 驱动程序自动转换为 **String**)
 - 详情请参见 **com.vaadin.addon.sqlcontainer.query.generator.StatementHelper**。
- 使用 Oracle 或 MS SQL 数据库时, 在 **SQLContainer** 连接的表中, "*rownum*" 不可以用作列的名称。

出现这个限制的原因是, 数据的分页存取需要依靠 **limit/offset** 语句来实现。如果数据库不支持 **limit/offset** 语句, 为了实现分页存取, 会使用一个动态生成的列 '*rownum*' 来计算行的位置, 相应的, 在真实的列中, 就不能出现这个名称。

永久存在的限制事项列举如下。在 **SQLContainer** 的未来版本中, 这些限制事项不会解决, 或者很可能不会解决。

- **getIds()** 方法有严重的效率问题 - 除非绝对必要, 否则尽量不要调用这个方法!
- 使用 **FreeformQuery** 时如果不指定 **FreeformStatementDelegate**, 行数的查询会有严重的效率问题 - 使用 **FreeformQuery** 时至少应该正确地实现行数查询功能。
- 使用 **FreeformQuery** 时如果不指定 **FreeformStatementDelegate**, 数据写入, 排序和过滤功能将不可用。

- 使用 Oracle 数据库时，大多数 numeric 类型会被 Oracle JDBC 驱动程序转换为 **java.math.BigDecimal**.

这是 Oracle DB 和 Oracle JDBC 驱动程序处理数据的特性.

Eclipse UI

10.1. 概述	315
10.2. 创建新的复合组件	316
10.3. 使用可视化编辑器	318
10.4. 可视化可编辑组件的结构	323

本章介绍如何使用 Vaadin Plugin for Eclipse IDE 开发 Vaadin 图形用户界面。

10.1. 概述

Vaadin Plugin for Eclipse 的可视化编辑器功能可用于开发整个应用程序的 UI, 也可用于开发某个特定的复合组件的 UI. 插件产生实际的 Java 代码, Java 代码被设计为可重用的, 因此你可以使用可视化编辑器来设计 UI 的基本架构, 然后在自动生成的代码基础之上, 开发用户操作相关的控制逻辑. 你可以通过继承和组合的方式来变更既有的组件功能.

编辑器可编辑的对象是继承自 **CustomComponent** 的组件类, 继承这个类也是 Vaadin 中创建复合组件的基本技术. 关于复合组件, 参见 第 5.28 节 “使用 **CustomComponent** 创造复合组件”. 可视化编辑器中不可以直接使用 **CustomComponent**; 你需要创建新的组件《方法见后文.

使用复合组件

你可以象使用其他任何 Vaadin 组件一样使用复合组件. 但是, 复合组件, 以及它的根布局(是一个 **AbsoluteLayout** 类), 默认为全尺寸(宽度和高度为 100%). 全尺寸的组件(扩张尺寸, 以充满所属的布局)在一个未指定尺寸的布局(会缩小尺寸, 以适应内容组件的大小)之内可能无法正常工作. 比

如, 如果你将一个复合组件放置在一个 **VerticalLayout** 之内, 这个布局的默认高度为未指定, 你必须明确地设置布局高度为一个确定值, 要么是固定值, 要么是全(100%)高度.

```
public class MyUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        // Create the content root layout for the UI  
        VerticalLayout content = new VerticalLayout();  
        setContent(content);  
  
        // Needed because the composites are full size  
        content.setSizeFull();  
  
        MyComposite myComposite = new MyComposite();  
        content.addComponent(myComposite);  
    }  
}
```

你可以(在可视化编辑器的组件属性中)将复合组件的根布局高度设置为固定值. 重要的是要注意 **AbsoluteLayout** 的尺寸不允许为未指定.

安装可视化编辑器

可视化编辑器目前是 Vaadin Plugin for Eclipse 的一部分. 它的安装, 请参见 第 2.4 节 “安装 Vaadin Plugin for Eclipse”.

编辑器运行在 Eclipse 的内嵌浏览器中. 实际使用的浏览器引擎由操作系统决定. 要使用内嵌浏览器, 必须通过以下菜单打开相关设定: **Window → Preferences → General → Web Browsers**.

在 Ubuntu 12.04 和其他某些版本中, 默认没有安装内嵌浏览器. 你至少可以使用 Firefox XULRunner 和 WebKit. 安装 WebKit 的方法如下:

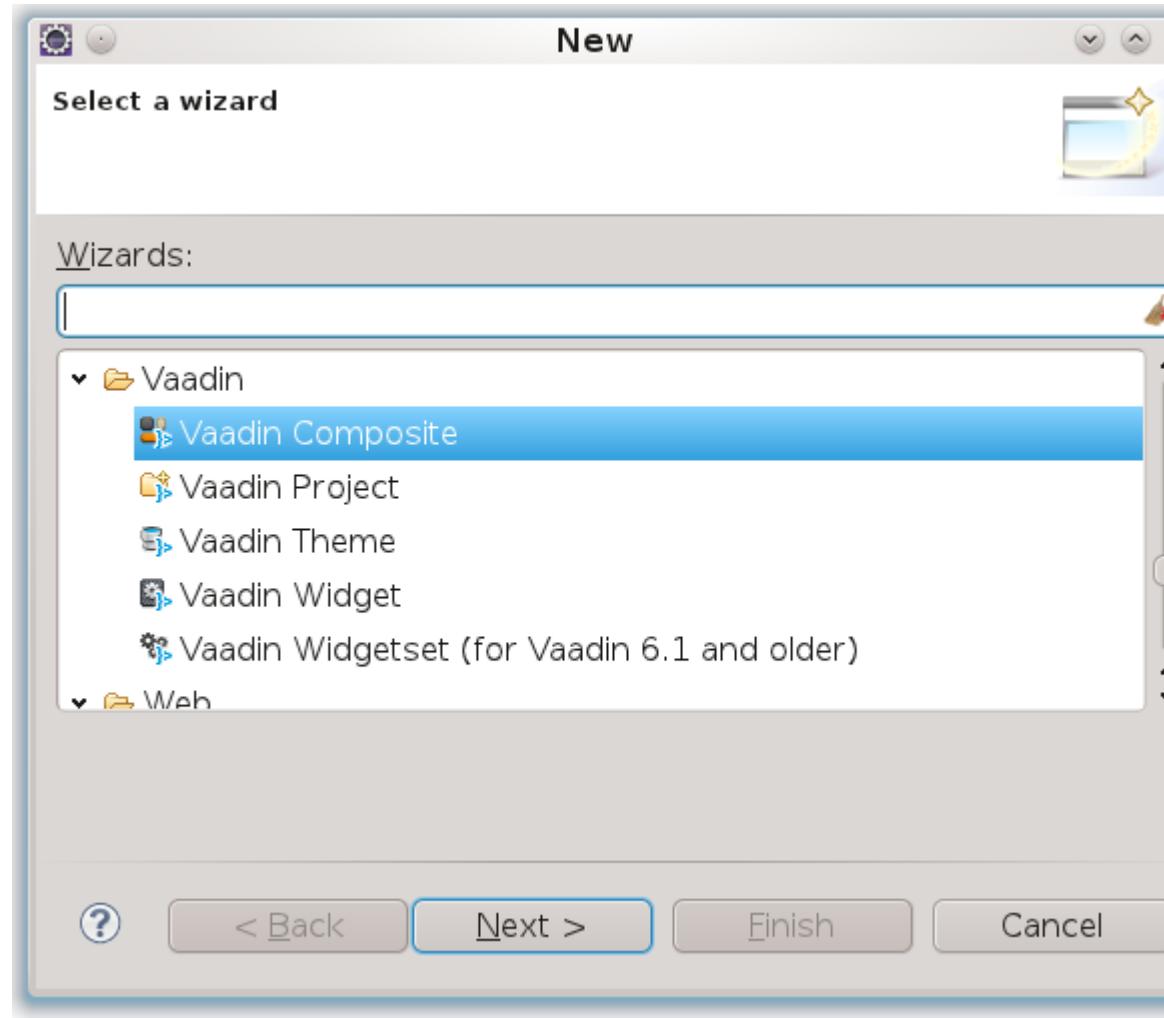
```
$ sudo apt-get install libwebkitgtk-1.0-0
```

然后, 重启 Eclipse, 查看内嵌浏览器是否已启用.

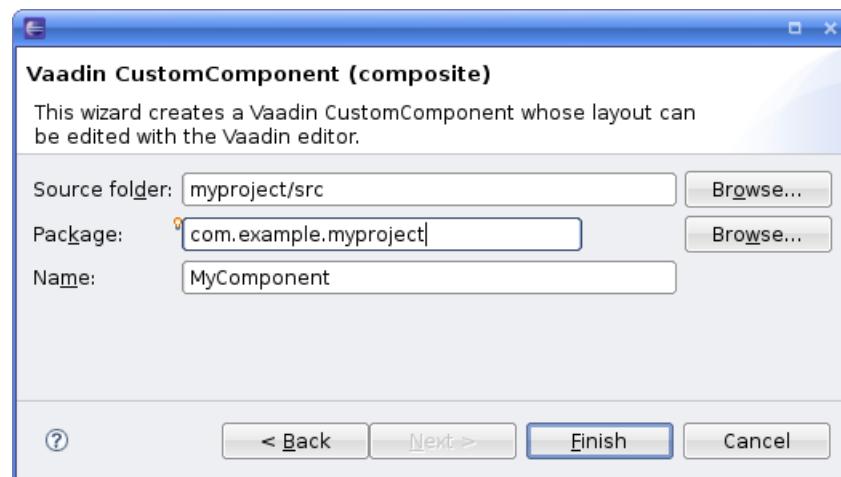
10.2. 创建新的复合组件

如果 Eclipse 中已安装了 Vaadin Plugin, 你可以创建新的复合组件, 方法如下.

1. 在主菜单中选择菜单项 **File → New → Other...**, 或者鼠标右键单击 **Project Explorer**, 然后选择菜单项 **New → Other...**, 打开 **New** 窗口.
2. 在第一步, **Select a wizard** 窗口中, 选择 **Vaadin → Vaadin Composite**, 然后单击 **Next** 按钮.



3. **Source folder** 是新组件创建时的源代码根目录. 这个项目的默认值是你的工程的默认源代码目录.

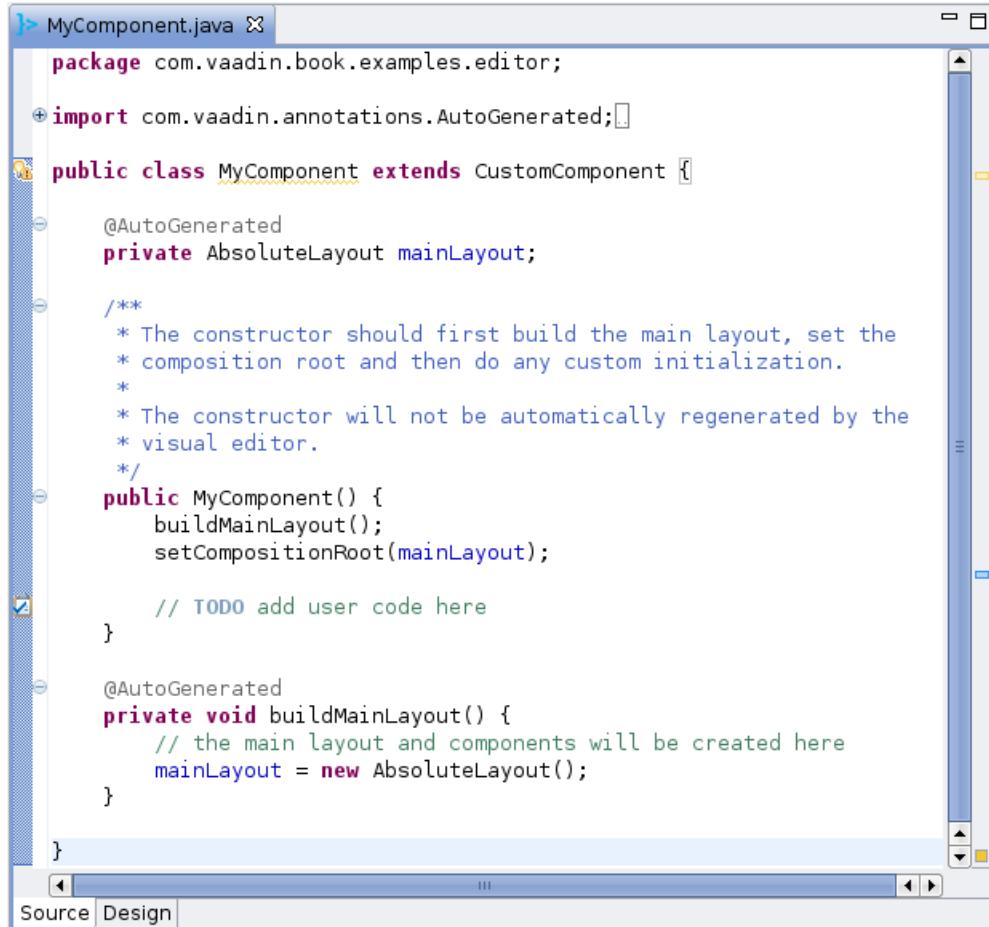


输入新组件创建时所属的 Java **Package** 名, 或者点击 **Browse** 按钮来选择包名. 还需要输入新组件的类 **Name**.

最后, 点击 **Finish** 按钮, 创建组件.

新创建的复合组件会在 **Design** 窗口中打开, 见图 10.1 “新创建的复合组件”.

图 10.1. 新创建的复合组件



```

MyComponent.java X
package com.vaadin.book.examples.editor;

import com.vaadin.annotations.AutoGenerated;

public class MyComponent extends CustomComponent {

    @AutoGenerated
    private AbsoluteLayout mainLayout;

    /**
     * The constructor should first build the main layout, set the
     * composition root and then do any custom initialization.
     *
     * The constructor will not be automatically regenerated by the
     * visual editor.
     */
    public MyComponent() {
        buildMainLayout();
        setCompositionRoot(mainLayout);
    }

    // TODO add user code here
}

@AutoGenerated
private void buildMainLayout() {
    // the main layout and components will be created here
    mainLayout = new AbsoluteLayout();
}

```

Source | Design

使用可视化编辑器来编辑组件时, 你可以看到在视图底部有两个 Tab: **Source** 和 **Design**. 这些 Tab 可用来在源代码视图和可视化设计视图之间切换.

如果你将来打开源代码来编辑, **Source** 和 **Design** Tab 会出现在代码编辑器的底部. 如果没有出现, 请在 Project Explorer 中用鼠标右键单击源代码文件, 然后选择菜单项 **Open With → Vaadin Editor**. (译注: 此处原文的菜单项 tag 语法有误)

10.3. 使用可视化编辑器

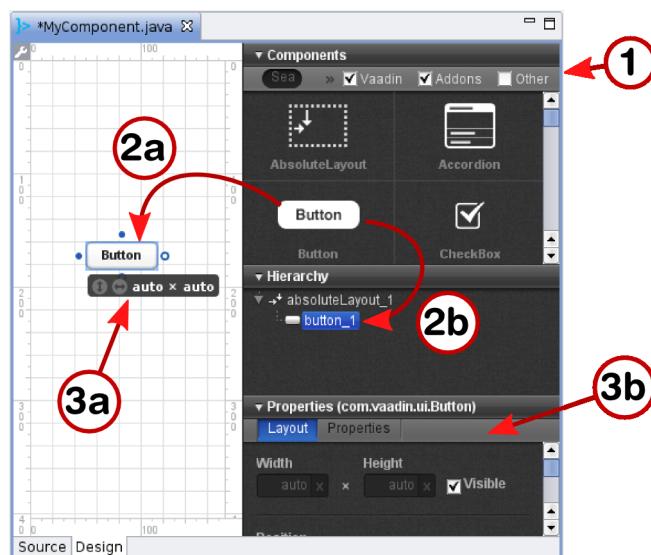
可视化编辑器视图由以下几部分组成: 左侧是编辑区, 显示目前的布局, 右侧是控制面板, 其中包括 1, 组件列表, 用于选择需要添加的新组件, 2, 目前的组件树, 3, 组件属性面板.

10.3.1. 添加新组件

向 UI 添加新组件的步骤如下: 从组件列表中拖放组件到编辑区, 或者拖放到组件树中. 如果将组件拖放到组件树的话,(译注: 此处疑似原文不完整)

1. 选择组件列表中显示的组件，方法是输入一个检索字符串，或者展开过滤器，然后只选择需要的组件类别。
2. 将一个组件从组件列表拖拽到：
 - a. 编辑区，在这里你可以便利地移动组件并调整其尺寸。将组件拖放到布局组件之上，会将它添加到这个布局之内，你也可以在布局内不拖动组件来调整其位置。
 - b. 组件树。注意，你只能将组件添加到布局组件或其他组件容器之下。
3. 编辑组件属性
 - a. 在编辑区，你可以移动组件并调整其尺寸，还可以设置它们在布局内的对齐方式。
 - b. 在属性面板中，你可以设置组件名称，尺寸，位置，以及其他属性。

图 10.2. 添加一个新的组件节点



你可以删除一个组件，方法是在属性树中用鼠标右键单击这个组件，然后选择菜单项 **Remove**。这个上下文菜单也可以用来复制和粘贴组件。

由 Plugin 创建的复合组件必须使用 **AbsoluteLayout** 作为它的根布局。虽然绝对布局适用于可视化编辑器，但在 Vaadin 应用程序的其他地方极少用到。如果你希望使用其他的根布局，你可以在 `mainLayout` 之内添加一个其他布局，然后在源代码视图中，使用 `setCompositionRoot()` 方法将它设置为根。当复合组件在应用程序中被使用到的时候，这个布局就会用作根布局。

10.3.2. 设置组件属性

控制面板中的属性设置子面板可用来设置组件的属性。这个面板有两个 Tab: **Layout** 和 **Properties**，其中 **Properties** Tab 定义了一些基本属性。

基本属性

Property 面板的第一部分，见图 10.3 “组件基本属性”，可用于设置组件的基本属性。这个面板还包括 Field 组件的 Field 属性。

图 10.3. 组件基本属性



基本属性如下

Name

组件名称, 用于引用这个组件, 因此它必须遵守 Java 变量的命名规则.

Style Name

组件的 CSS 样式类名称列表, 使用空格分隔. 关于 Theme 中的组件样式, 详情请参见第 7 章 Themes.

Caption

组件的标题通常显示在组件上方. 某些组件, 比如 **Button**, 标题显示在组件内部. 关于 **Label** 的文字, 你应该设置 Label 的值, 而不是标题, 标题应该是空.

Description (tooltip)

描述信息通常以提示信息的方式表示, 当鼠标指针停留在组件上方一段时间时会显示出来. 某些组件, 比如 **Form**, 显示描述信息的方式会不同.

Icon

组件的图标通常显示在组件上方, 在标题的左侧. 某些组件, 比如 **Button**, 会将图标显示在组件内部.

Content Type

有些组件允许不同的内容类型, 比如 **Label**, 它允许的内容包括 **TEXT**, **HTML**, 以及 **PREFORMATTED**.

Value

组件的值, 值的类型, 以及值如何被组件显示, 在各种组件中是不同的. 各值类型都有自己的编辑器. 点击 ... 按钮会打开编辑器.

组件的大多数基本类型定义在 **Component** 接口中; 详情请参见 第 5.2.1 节 “**Component** 组件”.

布局属性

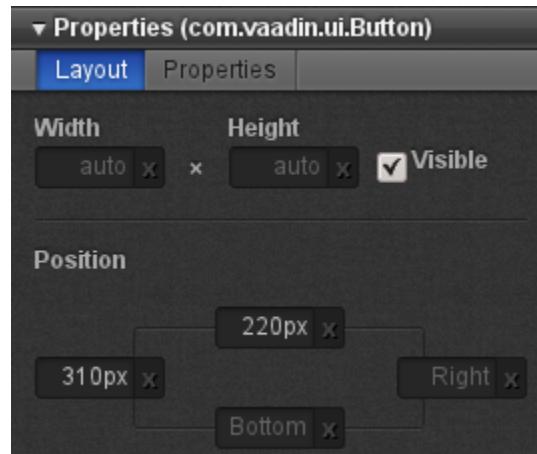
组件的尺寸由它的宽度和高度决定, 宽度和高度可以在控制面板的两个编辑框内输入. 你可以使用任意一种组件尺寸单位, 详情请参见 第 5.3.9 节 “控制组件的尺寸”. 尺寸编辑框中输入空白会使得这个尺寸“自动设定”, 也就是将尺寸设置为 未指定. 在生成的源代码中, 未定义的尺寸值将表达为 “-1px”.

对一个按钮, 设置它的宽度为 “100px”, 高度为 自动(未指定, 或者说空), 会产生以下设定代码:

```
// myButton
myButton = new Button();
...
myButton.setHeight("-1px");
myButton.setWidth("100px");
...
```

图 10.4 “布局属性” 展示了组件尺寸和位置的控制面板区

图 10.4. 布局属性



上面的示例图对应的自动生成的代码将是:

```
// myButton
myButton = new Button();
myButton.setWidth("100px");
myButton.setHeight("-1px");
myButton.setImmediate(true);
myButton.setCaption("My Button");
mainLayout.addComponent(myButton,
    "top:243.0px;left:152.0px;");
```

组件位置是 `addComponent()` 方法的第二个参数, 以 CSS 位置属性的形式给出. 对于宽度和高度, 值 “-1px” 会使得按钮将其大小自动调整为标题所需要的最小值.

对于 **AbsoluteLayout** 之内的组件, 编辑它的位置时, 编辑器会显示垂直和水平的向导线, 你可以用来设置组件位置. 关于绝对布局的编辑, 详情请参见 第 10.3.3 节 “编辑 **AbsoluteLayout**”.

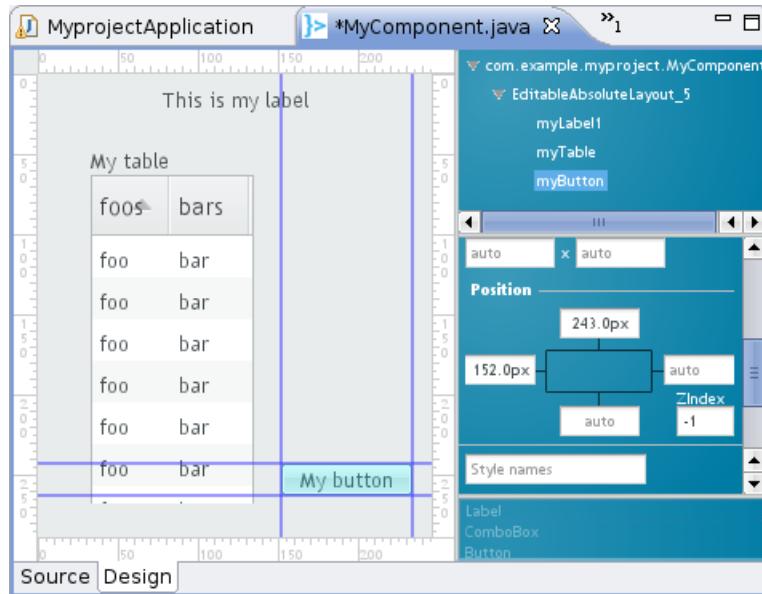
ZIndex 设置控制组件的 “Z 坐标”, 也就是当组件位置重叠时, 哪个组件显示在别的组件上方. 值 -1 代表自动, 这时后加入布局中的组件将显示在最上方.

10.3.3. 编辑 **AbsoluteLayout**

可视化编辑器针对 **AbsoluteLayout** 组件有特别的交互式支持，这个布局允许将组件定位到精确的坐标上。你可以使用向导线来定位组件，向导线控制组件的位置属性，位置属性显示在右侧的控制面板中。位置值是相对于边界的像素值；垂直和水平标尺显示从上边界和左边界开始的距离。

图 10.5 “使用 **AbsoluteLayout** 定位组件” 中显示了三个组件，一个 **Label**，一个 **Table**，一个 **Button**，这些组件包含在一个 **AbsoluteLayout** 之内。

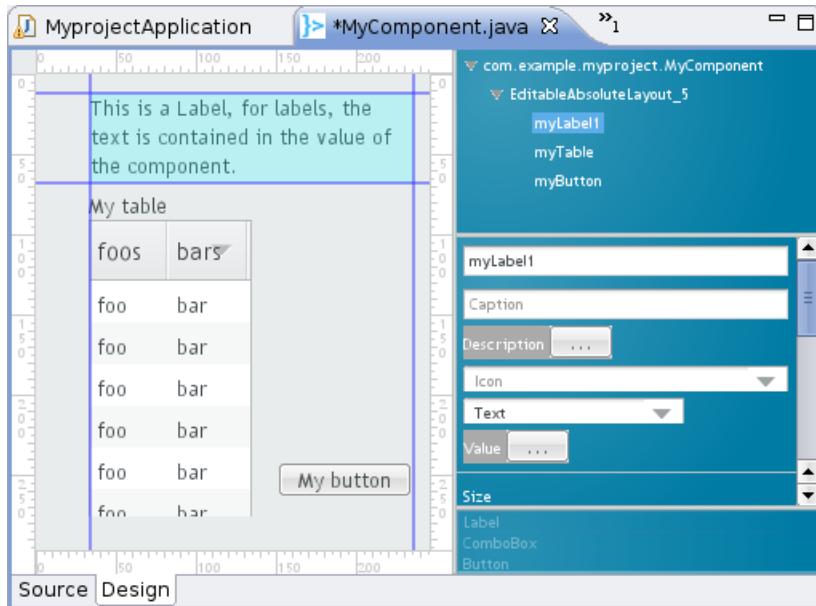
图 10.5. 使用 **AbsoluteLayout** 定位组件



位置属性为空代表 自动定位，结果可能是0(在边界上)，也可能是动态缩小到与内含组件的尺寸相适应，具体如何根据组件类型而不同。向导线对于这种自动定位属性也会显示，而且会自动移动位置；在图 10.5 “使用 **AbsoluteLayout** 定位组件” 中，**Button** 的右边界和下边界就是自动定位的。

手动移动一个自动定位的向导线，会使得向导线和对应的位置属性都变为非自动。要把一个手动设定的位置属性变为自动，你需要在控制面板中将属性设置为空。图 10.6 “手动定位的 **Label**” 显示了一个 **Label** 组件，它的四个边界都是手动设定的。注意，如果一个自动定位属性结果为 0，这时向导线会位于位置标尺的边界处。

图 10.6. 手动定位的 Label



10.4. 可视化可编辑组件的结构

由向导创建，然后由可视化编辑器编辑过的组件，会拥有一个非常特殊的结构，这个结构既允许你向组件中插入你的 UI 逻辑，同时又将不允许修改的代码保持在最小限度。你需要知道哪些代码是可以由你编辑的，哪些代码是只能由可视化编辑器管理的。在下文中你将看到，由可视化编辑器管理的成员变量和成员方法使用 **AutoGenerated** 注解来标记。

一个可以被可视化编辑器编辑的组件由以下几个部分组成：

- 成员变量，其中存放子组件的引用
- 子组件的构建方法
- 构造函数

复合组件的结构是层级式的，布局组件之下再包含布局组件和其他通常组件。组件树的根布局，或者叫 **CustomComponent** 的复合根，被命名为 `mainLayout`。关于自定义(复合)组件的结构，详情请参见第 5.28 节“使用 **CustomComponent** 创造复合组件”。

10.4.1. 子组件的引用

CustomComponent 类中，会以成员变量的形式引用每一个它包含的组件。其中最重要的是 `mainLayout`，它引用到复合组件的根布局。这些自动生成的成员变量会带有 `@AutoGenerated` 注解。它们由可视化编辑器来管理，因此你不应该手动编辑它们，除非你非常清楚这样做的后果。

以 **AbsoluteLayout** 作为根布局的复合组件，其中包含一个 **Button** 和一个 **Table**，那么复合组件中的引用将是如下：

```
public class MyComponent extends CustomComponent {

    @AutoGenerated
    private AbsoluteLayout mainLayout;
    @AutoGenerated
```

```
private Button myButton;  
@AutoGenerated  
private Table myTable;  
...
```

成员变量的名称在可视化编辑器的组件属性面板中，由 **Component name** 输入项指定，参见“基本属性”一节。虽然你可以修改任何其他组件的名称，但根布局的名称永远是 `mainLayout`。这个名称是固定的，因为可视化编辑器不会修改构造函数，详情请参见第 10.4.3 节“构造函数”。但是，你可以修改根布局的类型，默认类型是 **AbsoluteLayout**。

某些通常来说固定的组件，比如 **Label** 标签组件，不会存在对应的成员变量来保存它的引用。详情请参见下文中关于构建方法的说明。

10.4.2. 子组件构建方法

每一个由可视化编辑器管理的布局组件都将拥有一个构建方法，这个方法创建布局及布局内所有组件。构建方法将被创建的各组件的引用保存到对应的成员函数中，而且这个方法还将它创建的布局组件作为自己的返回值。

下例是 `mainLayout` 的初始化代码：

```
@AutoGenerated  
private AbsoluteLayout buildMainLayout() {  
    // common part: create layout  
    mainLayout = new AbsoluteLayout();  
  
    // top-level component properties  
    setHeight("100.0%");  
    setWidth("100.0%");  
  
    return mainLayout;  
}
```

注意，构建方法虽然返回它创建的组件的引用，但它也同时将引用直接写入到对应的成员函数中去。所以方法返回的引用有可能在自动生成的代码中（在构造函数中，或在构建方法中）完全不会被使用到，但你可以为自己的目的来使用这些返回值。

`mainLayout` 的构建方法被构造函数调用，详情请参见第 10.4.3 节“构造函数”。如果你的布局中又含有嵌套的布局组件，每个布局的构建方法为了创建它自己的内容，都会调用它内含的下层布局的构建方法。

10.4.3. 构造函数

使用向导创建新的复合组件时，它会为组件创建一个构造函数，并在其中填充一些基本代码。

```
public MyComponent() {  
    buildMainLayout();  
    setCompositionRoot(mainLayout);  
  
    // TODO add user code here  
}
```

构造函数中要做的最重要的事是，使用 `setCompositionRoot()` 方法，设置 **CustomComponent** 的复合根组件（关于复合根组件，详情请参见第 5.28 节“使用 **CustomComponent** 创造复合组件”）。自动生成的构造函数首先使用 `buildMainLayout()` 方法创建复合组件的根布局，然后使用 `mainLayout` 引用来设置复合根组件。

在这之后，可视化编辑器不会再修改构造函数，因此你可以安全地按照自己的需要修改它。可视化编辑器不允许修改保持根组件引用的成员函数，因此它的名称永远是 mainLayout。

Web

11.1. 管理浏览器窗口	327
11.2. 在 Web 页面中嵌入 UI	330
11.3. Debug 模式和 Debug 窗口	338
11.4. 请求处理器(Request Handler)	342
11.5. 快捷键	343
11.6. 打印	347
11.7. 与 Google App Engine 的集成	349
11.8. 共通的安全问题	350
11.9. 应用程序内的导航跳转	351
11.10. 应用程序高级架构	355
11.11. 管理 URI 片段	360
11.12. 拖放	362
11.13. 日志	370
11.14. 与 JavaScript 集成	371
11.15. 访问 Session 全局数据	372
11.16. 服务器端 PUSH	376

本章将介绍应用程序开发中常见的一些功能和问题。

11.1. 管理浏览器窗口

Vaadin 应用程序的 UI 运行在 Web 页面内, Web 页面则显示在浏览器窗口或 Tab 中。应用程序可以由不同窗口或不同 Tab 中的多个 UI 使用, 这些窗口或 Tab 可以是用户通过 URL 打开的, 也可以是 Vaadin 应用程序打开的。

除浏览器原生的窗口外, Vaadin 还有一个 **Window** 组件, 它是页面内的一个浮动的 panel, 或者叫子窗口, 详情请参见 第 6.7 节 “子窗口”.

- 原生弹出式窗口. 应用程序为执行某种子任务, 可以打开弹出式窗口.
- 基于页面的浏览. 应用程序可以允许用户在其他窗口中打开特定的内容. 比如, 在消息管理程序中, 就会需要在不同的窗口中打开不同的消息, 以便用户一边输入新消息一边浏览这些已有的消息.
- 书签. Web 浏览器中的书签可以记录下应用程序中某些内容的入口地址.
- 嵌入式 UI. UI 可以嵌入到 Web 页面中, 因此可以通过不同的页面为同一个应用程序提供不同的界面, 甚至可以在同一个界面内提供不同的界面, 同时还能将这些界面保持在同一个 session 内. 详情请参见 第 11.2 节 “在 Web 页面中嵌入 UI”.

在应用程序内使用多窗口可能需要为不同的窗口定义和提供不同的 UI. 应用程序中的 UI 共用相同的 session, 也就是同一个 **VaadinSession** 对象, 详情请参见 第 4.7.3 节 “用户 Session”. 各个 UI 通过与它绑定的 URL 来区分, 因此可以将应用程序中各 UI 的地址保存为浏览器书签. UI 实例甚至可以根据 URL 或其他请求参数来动态创建, 比如根据浏览器信息不同来创建不同的 UI, 详情请参见 第 4.7.4 节 “装载 UI”.

由于 AJAX 应用程序的特殊性质, 使用多窗口也会存在一些限制.

11.1.1. 打开弹出式窗口

弹出窗口是浏览器原生的窗口或 Tab, 它通过用户在一个既存窗口中的操作来打开. 由于浏览器安全性上的限制, 在 Web 页面中使用 JavaScript 命令来打开弹出窗口是很不方便的. 即使浏览器可以打开弹出窗口, 至少它也会询问用户是否允许打开弹出窗口. 可以用一些手段来绕过这个限制, 方法是当用户点击某些元素时, 直接使用 URL 来让浏览器打开新窗口或新 Tab. 这种方法在 Vaadin 中是通过 **BrowserWindowOpener** 组件扩展实现的, 它可以在组件被点击时让浏览器打开新窗口或新 Tab.

弹出窗口 UI

弹出窗口也会显示一个 **UI**. 弹出窗口的 UI 的声明与 Vaadin 应用程序的 UI 类似, 它也可以拥有 theme, 标题, 等等.

比如:

```
@Theme("book-examples")
public static class MyPopupUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().setTitle("Popup Window");

        // Have some content for it
        VerticalLayout content = new VerticalLayout();
        Label label =
            new Label("I just popped up to say hi!");
        label.setSizeUndefined();
        content.addComponent(label);
        content.setComponentAlignment(label,
            Alignment.MIDDLE_CENTER);
        content.setSizeFull();
        setContent(content);
    }
}
```

弹出它

弹出窗口使用 **BrowserWindowOpener** 扩展来弹出, 你可以将这个扩展绑定到任意组件上. 它的构造函数参数是希望打开的 UI 类.

你可以使用 `setFeatures()` 方法来配置弹出窗口的特性. 这个方法的参数是逗号分隔的窗口特性列表, 窗口特性遵照 HTML 规约中的定义.

`status=0|1`
 窗口下方的状态栏是否显示.

`scrollbars`
 当文档内容比窗口中的视图区域更大时, 显示滚动条.

`resizable`
 允许用户拉伸浏览器窗口大小(对 Tab 无效).

`menubar`
 显示浏览器菜单栏.

`location`
 显示地址栏.

`toolbar`
 显示浏览器工具栏.

`height=value`
 指定窗口高度, 单位为像素.

`width=value`
 指定窗口宽度, 单位为像素.

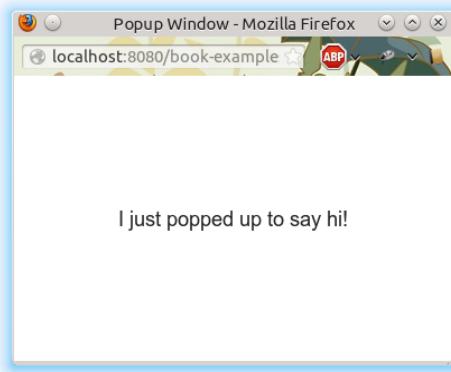
比如:

```
// Create an opener extension
BrowserWindowOpener opener =
    new BrowserWindowOpener(MyPopupUI.class);
opener.setFeatures("height=200,width=300,resizable");

// Attach it to a button
Button button = new Button("Pop It Up");
opener.extend(button);
```

按钮按下后出现弹出窗口, 运行结果见图 11.1 “弹出窗口”.

图 11.1. 弹出窗口



弹出窗口的名称(**Target**)

窗口的 target 名称可以是 HTML 默认 target 名称之一(`_new`, `_blank`, `_top`, 等等.), 也可以是自定义的 target 名称. 窗口具体会如何打开, 由浏览器决定. 支持 tab 式浏览的浏览器可能会在另一个 tab 内打开窗口, 具体取决于浏览器的设置.

URL 与 Session

弹出窗口 UI 的 URL 路径默认由 UI 的类名决定, 使用 "popup/" 前缀. 比如, 对于前面例子中给出的 UI, 它的 URL 将是 `/book-examples/book/popup/MyPopupUI`.

11.1.2. 关闭弹出式窗口

除了使用浏览器的窗口关闭按钮来关闭弹出窗口之外, 你还可以调用 JavaScript 的 `close()` 方法来关闭它, 如下:

```
public class MyPopup extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Close Window", event -> { // Java 8
            // Close the popup
            JavaScript.eval("close()");

            // Detach the UI from the session
            getUI().close();
        }));
    }
}
```

11.2. 在 Web 页面中嵌入 UI

很多 Web 站点并不是完全使用 Vaadin 开发的, 但 Vaadin UI 在其中用于某些特定的功能. 实际应用中, 很多 Web 应用程序是由动态 Web 页面混合而成的, 比如 JSP, Vaadin UI 则嵌入在这些页面之内.

将 Vaadin UI 嵌入到 Web 页面之内是很容易的, 而且有几种不同方法可以实现 UI 的嵌入. 第一种方法是为 UI 使用一个 `<div>` 占位符, 然后使用一段简单的 JavaScript 代码装载 Vaadin 客户端引擎. 另一种方法更简单, 只需要使用 `<iframe>` 元素即可. 这些方法都有各自的优点和缺点. `<iframe>` 方法的缺点之一是 `<iframe>` 元素的尺寸不能灵活适应 UI 的内容, 而 `<div>` 则是可以的. 以下小节详细介绍这两种嵌入方法.

11.2.1. 在 div 元素内嵌入 UI

你可以在 Web 页面内嵌入一个或多个 Vaadin UI，方法与非嵌入式 UI 中通过 Vaadin Servlet 来装载初始化页面的方法相同。通常，**VaadinServlet** 会生成初始化页面，其中包含用于装载特定 UI 的正确参数。你可以很容易的配置它，使它在同一个页面内装载多个 Vaadin UI。这些 UI 可以使用不同的 Widget 群和不同的 theme。

嵌入 UI 需要完成以下基本任务：

- 设置页面的 header 部
- 在页面内包含 GWT 的历史 frame
- 调用 `vaadinBootstrap.js` 文件
- 为 UI 定义 `<div>` 元素
- 配置并初始化 UI

注意，你可以很容易地查看 UI 的装载页面，方法是在 Web 浏览器中打开 UI，然后查看这个页面的 HTML 源代码。你可以从这个页面中将负责嵌入 UI 的部分代码复制并粘贴出来，但可能需要少量的修改和一些额外的设置，主要的修改与 URL 相关，URL 路径必须修改为相对于新的装载页面，而不是相对于 Servlet URL。

本书出版后，DIV 嵌入的相关 API 可能很快会修改。Vaadin 网站上应该会有关于这个功能的教程。

Head 部分

嵌入 Vaadin UI 的 HTML 页面必须是合法的 HTML 5 文档。Head 元素的内容很大程度上由你决定。字符编码必须是 UTF-8。为保持兼容性，还必须定义一些其他 meta 信息。你还可以在 Head 元素中设置页面的 favicon。

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <meta http-equiv="X-UA-Compatible"
          content="IE=9;chrome=1" />

    <title>This is my Embedding Page</title>

    <!-- Set up the favicon from the Vaadin theme -->
    <link rel="shortcut icon" type="image/vnd.microsoft.icon"
          href="/VAADIN/themes/reindeer/favicon.ico" />
    <link rel="icon" type="image/vnd.microsoft.icon"
          href="/VAADIN/themes/reindeer/favicon.ico" />
  </head>
```

Body 部分

在嵌入 Vaadin UI 之前，页面内容中必须包含一些 Vaadin 相关的定义信息。

`vaadinBootstrap.js` 脚本定义了 UI 启动相关的内容，在初始化 UI 之前必须调用这个脚本。这个脚本的源文件路径必须设置为相对于嵌入 UI 的页面路径。

```
<body>
<script type="text/javascript"
src=".//VAADIN/vaadinBootstrap.js"></script>
```

启动脚本由 vaadin-server JAR 内的 Vaadin Servlet 对外提供.

Vaadin, 更确切的说是 GWT, 需要一个不可见的历史 frame, 用来在浏览器内追踪页面或页面片段的历史.

```
<iframe tabindex="-1" id="__gwt_historyFrame"
style="position: absolute; width: 0; height: 0;
border: 0; overflow: hidden"
src="javascript:false"></iframe>
```

UI 占位元素

Vaadin UI 嵌入在 <div> 占位元素内. 它的设置必须如下:

- <div> 元素必须有 id 属性, 值必须是页面内的唯一 ID, 通常是一个可以唯一标识 UI 的 Servlet 的字符串.
- 它必须至少带有 v-app 样式.
- 它应该有一个内嵌的 <div> 元素, 带有 v-app-loading 样式. 这个是装载中状态指示器的占位元素, UI 装载过程中将会显示这个指示器.
- 它还需要包含一个 <noscript> 元素, 当浏览器不支持 JavaScript 或 JavaScript 被禁用时会显示这个元素. 这个元素的内容应该指导用户去启用浏览器中的 JavaScript 功能.

这个占位元素可以包含样式设定, 通常用来指定宽度和高度. 如果占位元素的尺寸未定义, UI 在对应的方向上的尺寸也会是未定义尺寸, UI 的尺寸必须与其中的 UI 组件的尺寸一致.

比如:

```
<div style="width: 300px; border: 2px solid green;">
  id="helloworldui" class="v-app">
    <div class="v-app-loading"></div>
    <noscript>You have to enable javascript in your browser to
      use an application built with Vaadin.</noscript>
</div>
```

初始化 UI

启动脚本中定义了 vaadin 对象, 对这个对象调用 initApplication() 方法, 即可装载 UI. 调用这个方法之前, 你应该检查启动脚本有没有正确装载.

```
<script type="text/javascript">//<! [CDATA[
  if (!window.vaadin)
    alert("Failed to load the bootstrap JavaScript:"+
      "VAADIN/vaadinBootstrap.js");
```

initApplication() 方法接受两个参数. 第一个参数是 UI 的标识符, 必须与占位符元素的 id 属性值完全一致. 第二个参数是一个Map, 其中包含传递给 UI 的参数.

Map 中必须包含以下项目:

browserDetailsUrl

这个参数是 UI 的 Vaadin Servlet 的 URL 路径(相对于嵌入 UI 的页面). 它被启动脚本用于传递浏览器的详细信息. 某些情况下可能需要在 URL 末尾带上斜线.

注意, 在 Servlet 生成的装载页面中不包含这个参数, 因为这种情况下, 它的默认值就是当前 URL.

serviceUrl

这个参数用于初期装载之后向服务器发送请求, 而且应该与 *browserDetailsUrl* 一样. 这两个参数目前是重复的, 将来可能会删除其中一个.

widgetset

这个参数应该是 UI 使用的 widget 群的类名, 也就是, 去掉 .gwt.xml 扩展名之后的文件名. 如果 UI 没有自定义 widget 群, 你可以使用 **com.vaadin.DefaultWidgetSet**.

theme

Theme 名称, 比如可以是内建 theme 之一(reindeer, runo, 或 chameleon), 也可以是自定义 theme. Theme 必须存在于 VAADIN/themes 目录之下.

versionInfo

这个参数自身又是一个 Map, 其中包含两个参数: *vaadinVersion* 是应用程序使用的 Vaadin 版本号. *applicationVersion* 是应用程序本身的版本号. 这个 Map 内的参数是可选的, 但 *versionInfo* 参数自身是必须的.

vaadinDir

VAADIN 目录的相对路径. 相对于嵌入 UI 的页面 URL.

heartbeatInterval

heartbeatInterval 参数指定 UI 的 keep-alive 心跳信号发送频度, 单位为秒, 详情请参见 第 4.7.5 节 “UI 过期”.

debug

这个参数指定 Debug 窗口是否激活, 详情请参见 第 11.3 节 “Debug 模式和 Debug 窗口”.

standalone

嵌入模式下这个参数应该为 *false*. 这个参数指定 UI 是在独有的浏览器窗口中展现, 还是被用于某种上下文环境中. Standalone 模式的 UI 的某些行为可能会干扰页面的其他部分, 比如它可能改变页面标题, 还可能在装载完成后要求聚焦. 嵌入模式下, UI 不是 Standalone 模式.

authErrMsg, comErrMsg, 和 sessExpMsg

这些参数指定客户端的错误消息, 分别对应于认证错误, 通信错误, session 过期. 这些参数本身都是 Map, 其中必须包含两组值: *message*, 指定 HTML 格式的错误信息文本, 以及 *caption*, 指定错误信息标题.

比如:

```
vaadin.initApplication("helloworldui", {
  "browserDetailsUrl": "helloworld/",
  "serviceUrl": "helloworld/",
  "widgetset": "com.example.MyWidgetSet",
  "theme": "mytheme",
  "versionInfo": {"vaadinVersion": "7.0.0"},
  "vaadinDir": "VAADIN/",
  "heartbeatInterval": 300,
```

```

    "debug": true,
    "standalone": false,
    "authErrMsg": {
        "message": "Take note of any unsaved data, "+
                    "and <u>click here</u> to continue.",
        "caption": "Authentication problem"
    },
    "comErrMsg": {
        "message": "Take note of any unsaved data, "+
                    "and <u>click here</u> to continue.",
        "caption": "Communication problem"
    },
    "sessExpMsg": {
        "message": "Take note of any unsaved data, "+
                    "and <u>click here</u> to continue.",
        "caption": "Session Expired"
    }
});//]]>
</script>

```

注意，还有很多参数是通常的部署参数，在部署描述文件中指定，详情请参见 第 4.8.6 节“Servlet 的其他配置参数”。

关于 Div 嵌入的总结

以下是在 `<div>` 元素中嵌入 UI 的完整例子。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <meta http-equiv="X-UA-Compatible"
          content="IE=9;chrome=1" />

    <title>Embedding a Vaadin Application in HTML Page</title>

    <!-- Set up the favicon from the Vaadin theme -->
    <link rel="shortcut icon" type="image/vnd.microsoft.icon"
          href="/VAADIN/themes/reindeer/favicon.ico" />
    <link rel="icon" type="image/vnd.microsoft.icon"
          href="/VAADIN/themes/reindeer/favicon.ico" />
</head>

<body>
    <!-- Loads the Vaadin widget set, etc. -->
    <script type="text/javascript"
           src="VAADIN/vaadinBootstrap.js"></script>

    <!-- GWT requires an invisible history frame. It is -->
    <!-- needed for page/fragment history in the browser. -->
    <iframe tabindex="-1" id="__gwt_historyFrame"
            style="position: absolute; width: 0; height: 0;
                   border: 0; overflow: hidden"
            src="javascript:false"></iframe>

    <h1>Embedding a Vaadin UI</h1>

```

```

<p>This is a static web page that contains an embedded Vaadin
application. It's here:</p>

<!-- So here comes the div element in which the Vaadin -->
<!-- application is embedded. -->
<div style="width: 300px; border: 2px solid green;" id="helloworld" class="v-app">

    <!-- Optional placeholder for the loading indicator -->
    <div class=" v-app-loading"></div>

    <!-- Alternative fallback text -->
    <noscript>You have to enable javascript in your browser to
        use an application built with Vaadin.</noscript>
</div>

<script type="text/javascript">//<![CDATA[
    if (!window.vaadin)
        alert("Failed to load the bootstrap JavaScript: "+
            "VAADIN/vaadinBootstrap.js");

    /* The UI Configuration */
    vaadin.initApplication("helloworld", {
        "browserDetailsUrl": "helloworld/",
        "serviceUrl": "helloworld/",
        "widgetset": "com.example.MyWidgetSet",
        "theme": "mytheme",
        "versionInfo": {"vaadinVersion": "7.0.0"},
        "vaadinDir": "VAADIN/",
        "heartbeatInterval": 300,
        "debug": true,
        "standalone": false,
        "authErrMsg": {
            "message": "Take note of any unsaved data, "+
                "and <u>click here</u> to continue.",
            "caption": "Authentication problem"
        },
        "comErrMsg": {
            "message": "Take note of any unsaved data, "+
                "and <u>click here</u> to continue.",
            "caption": "Communication problem"
        },
        "sessExpMsg": {
            "message": "Take note of any unsaved data, "+
                "and <u>click here</u> to continue.",
            "caption": "Session Expired"
        }
    });
}};//]]>
</script>

<p>Please view the page source to see how embedding works.</p>
</body>
</html>

```

11.2.2. 嵌入到 iframe 元素中

在 `<iframe>` 元素内嵌入 Vaadin UI 比前面将的方法更简单一些，因为这种方法不必定义任何 Vaadin 相关的参数。

你可以使用以下代码在元素内嵌入 UI:

```
<iframe src="/myapp/myui"></iframe>
```

`<iframe>` 元素用于嵌入 UI 有几个不利的方面。首先，它的尺寸无法自动适应 frame 内容的尺寸，相反，内容尺寸必须适应 frame 的尺寸。你可以使用 `height` 和 `width` 属性设置 `<iframe>` 元素的尺寸。其他问题与 theme 以及 frame 内容和页面其他部分之间的通信有关。

以下是在一个 Web 页面中使用 `<iframe>` 嵌入两个应用程序的完整例子。

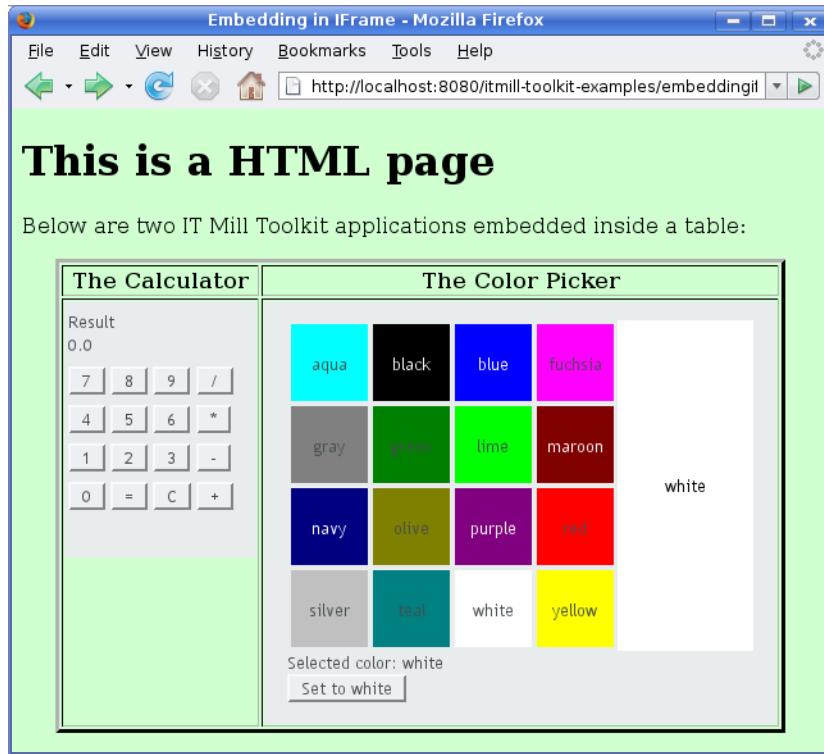
```
<!DOCTYPE html>
<html>
    <head>
        <title>Embedding in IFrame</title>
    </head>

    <body style="background: #d0ffd0;">
        <h1>This is a HTML page</h1>
        <p>Below are two Vaadin applications embedded inside
            a table:</p>

        <table align="center" border="3">
            <tr>
                <th>The Calculator</th>
                <th>The Color Picker</th>
            </tr>
            <tr valign="top">
                <td>
                    <iframe src="/vaadin-examples/Calc" height="200"
                           width="150" frameborder="0"></iframe>
                </td>
                <td>
                    <iframe src="/vaadin-examples/colorpicker"
                           height="330" width="400"
                           frameborder="0"></iframe>
                </td>
            </tr>
        </table>
    </body>
</html>
```

这个页面的运行结果，见图 11.2 “IFrame 内嵌入的 Vaadin 应用程序”(译注：此处的页面源代码与运行结果截图略有不符)。

图 11.2. IFrame 内嵌入的 Vaadin 应用程序



在 iframe 内你可以嵌入几乎任何东西, iframe 本质上等于一个浏览器窗口。但是, 这种方式也带来一些问题: iframe 的尺寸必须固定, 从嵌入 UI 的页面继承 CSS 是不可能的, 使用 JavaScript 进行交互也是不可能的, 因此导致不能混合多种内容, 等等等等。甚至不能使用 URI 片段来保存书签。

注意网站有可能会在 HTTP 应答中指定一个 X-Frame-Options: SAMEORIGIN 头, 禁止嵌入 iframe。

11.2.3. 使用 Vaadin XS Add-on 实现跨站嵌入

XS add-on 目前还不可用于 Vaadin 7.

前面的小节中, 我们介绍了在页面中嵌入 Vaadin 应用程序的两种基本方法: 使用 `<div>` 元素, 以及使用 `<iframe>` 元素。使用 `div` 元素嵌入的一个问题是, 它不能在不同的 Internet 域之间工作, 如果你希望让你的 Web 网站运行在一个服务器上, 而让你的 Vaadin 应用程序运行在另一个服务器上, 那么这就是个问题了。浏览器的安全模型强制要求 XMLHttpRequest 调用必须遵守同一来源政策, 这就有效地阻止了 Ajax 应用程序的这种跨站嵌入, 即使 Web 服务器和应用程序服务器运行在同一个域的不同端口上, 也无法工作。`iframe` 方式则更自由一些, 几乎允许嵌入任何位置的任何内容, 但正如前面介绍过的, 这种方式也有很多不利的方面。

Vaadin XS (Cross-Site) add-on 解决 `div` 元素的跨站嵌入问题, 它使用 JSONP 风格的通信, 而不是标准的 XMLHttpRequest。

嵌入方法很简单:

```
<script src="http://demo.vaadin.com/xseembed/getEmbedJs"
       type="text/javascript"></script>
```

以上代码在页面中引入一段自动生成的嵌入脚本, 它可以大大简化我们的嵌入工作。

以上代码假定有应用程序的主布局高度为未指定。如果高度为 100%，你必须将它包装在一个容器元素内，然后指定容器元素的高度。比如：

```
<div style="height: 500px;">
<script src="http://demo.vaadin.com/xsemebed/getEmbedJs"
        type="text/javascript"></script>
</div>
```

可以使用白名单来限制应用程序允许被嵌入到哪些位置。这个 add-on 还会对客户端/服务器端的通信进行加密，这一点对嵌入式应用程序来说，比通常应用程序更加重要。

你可以从 Vaadin Directory 得到 Vaadin XS add-on。它以 Zip 包的形式提供，下载并将安装包解开到本地目录。安装指南及更多详细信息请参见包内的 README.html 文件。

这个 add-on 也存在一些限制。在一个页面中只能存在一个内嵌应用程序。而且，某些第三方库可能会影响通信。其他注意事项请参见 README 文件。

11.3. Debug 模式和 Debug 窗口

Vaadin 应用程序可以运行在两种模式下：debug 模式和生产模式。默认是 Debug 模式，激活了很多内建的 debug 功能，可供 Vaadin 开发者使用：

- Debug 窗口
- 在 Debug 窗口以及服务器端控制台显示 debug 信息
- Sass theme 的即时编译

11.3.1. 打开 Debug 模式

使用 Eclipse plugin 或 Maven archetype 创建的 UI 模板中，默认启用 debug 模式，而生产模式是禁用的。也可以在 Vaadin Servlet 配置中指定一个 `productionMode=false` 参数来启用 debug 模式：

```
@VaadinServletConfiguration(
    productionMode = false,
    ui = MyprojectUI.class)
```

也可以在部署描述文件 `web.xml` 中指定一个 `context` 参数：

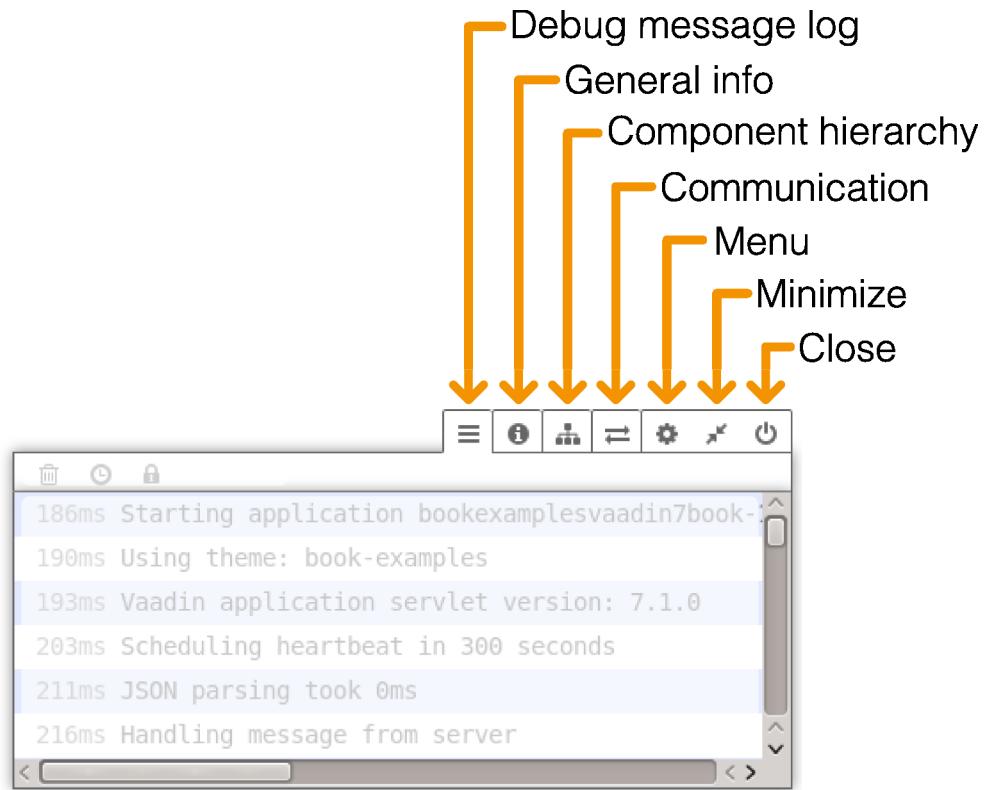
```
<context-param>
    <description>Vaadin production mode</description>
    <param-name>productionMode</param-name>
    <param-value>false</param-value>
</context-param>
```

启用生产模式会禁用 debug 功能，因此可以防止用户从浏览器中轻易地窥测到应用程序的内部工作状况。

11.3.2. 打开 Debug 窗口

在 debug 模式下运行应用程序，会激活浏览器中的客户端 Debug 窗口。你可以在 UI 的 URL 之后添加一个 `?debug` 来打开 Debug 窗口，比如，`http://localhost:8080/myapp/?debug`。Debug 窗口中有一些 debug 功能控制按钮，还有一个可滚动的日志栏，用于查看 debug 消息。

图 11.3. Debug 窗口



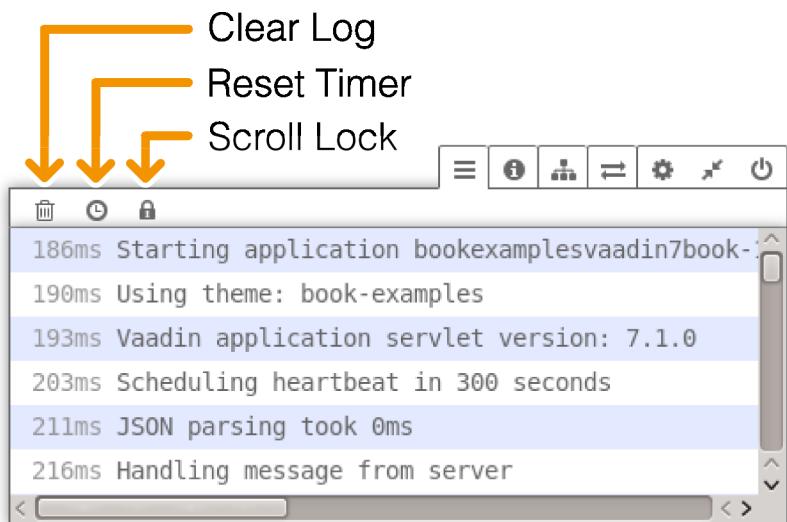
Debug 窗口的功能会在后续小节中详细介绍. 你可以拖动窗口的标题栏来移动它的位置, 也可以拖拽它的边角来调整大小. **Minimize** 按钮将 debug 窗口最小化到浏览器窗口之内, **Close** 按钮关闭 debug 窗口.

如果你在 Firefox 浏览器中使用 Firebug 插件, 或者在 Chrome 浏览器中使用 Developer Tools 控制台, 那么日志消息也会打印到 Firebug 或 Chrome 的控制台中. 这时, 你可能希望允许客户端调试, 但不要显示 Debug 窗口, 你可以在 URL 末尾添加 "?debug=quiet" 参数来实现这一点. 在 quiet debug 模式下, 日志信息只会输出到浏览器调试器的控制台中.

11.3.3. Debug 日志

debug 日志信息用于显示客户端 debug 信息, 还包含毫秒单位的计时器. 界面上有一些控制按钮, 可用于清除日志, 重置计时器, 或锁定滚动.

图 11.4. Debug 日志



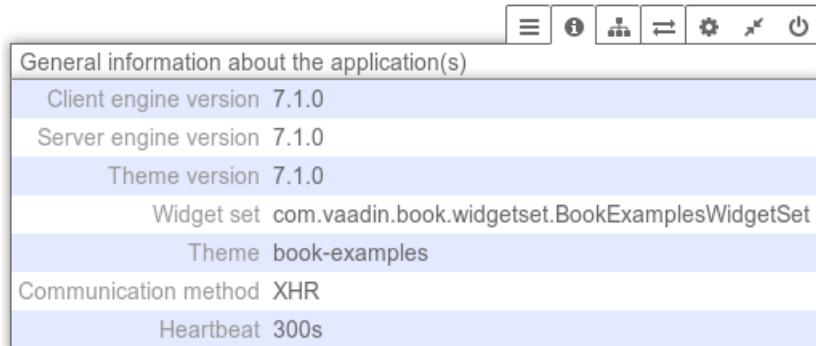
向 Debug 窗口输出日志

开发客户端组件时, debug 模式的日志功能可以为你带来很大的便利。你可以使用标准的 Java **Logger** 向输出日志, 日志信息也会被写入到 debug 窗口以及 Firebug 控制台。debug 窗口未打开时, 或应用程序运行于生产模式时, 不会有任何消息输出。

11.3.4. 一般信息

应用程序一般信息 tab 显示 UI 的各种信息, 比如客户端引擎, Servlet 引擎以及 theme 的版本号。如果这些版本不匹配, 你可能需要编译 widget 群或 theme。

图 11.5. 一般信息



11.3.5. 查看组件层级关系

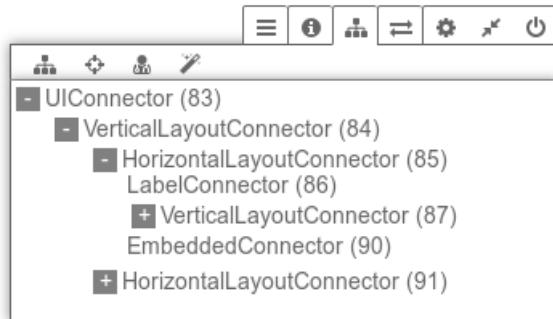
Component Hierarchy tab 有几个不同的子模式, 可以通过不同的方式调试组件树。

连接器的层级关系树

Show the connector hierarchy tree 按钮可以显示客户端连接器层级关系。客户端 widget 由连接器管理, 它负责处理与服务器端对应组件的通信, 详情请参见 第 16 章 与客户端集成。因此连接

器的层级对应到服务器端的组件树, 但客户端 widget 树和 HTML DOM 树比服务器端组件树要更复杂.

图 11.6. 连接器的层级关系树



点击一个连接器会将 UI 中对应的 widget 高亮显示.

查看组件信息

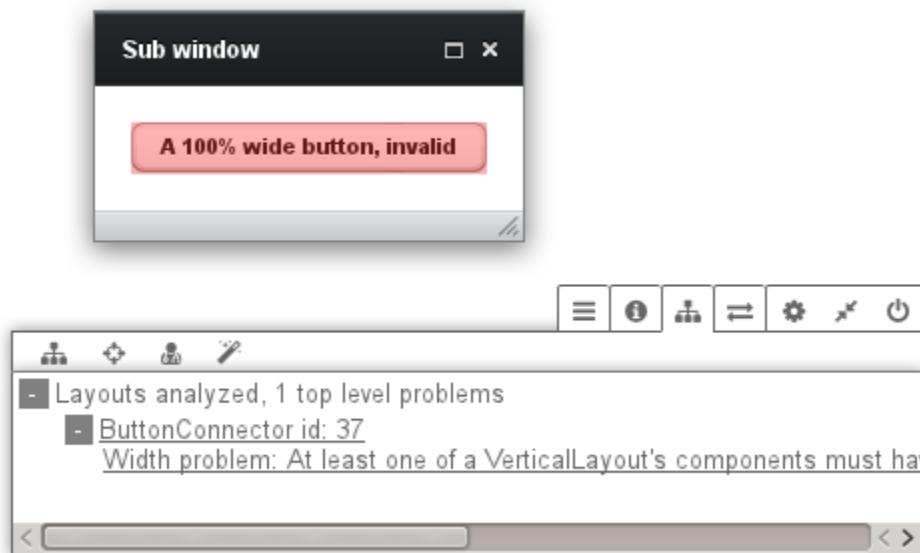
Select a component in the page to inspect it 按钮允许你选择 UI 中的一个组件, 你可以点击这个组件, 并显示它的客户端属性.

要查看 HTML 结构和 CSS 样式的更详细信息, 你可以使用 Firefox 中的 Firebug, 或 Chrome 中的 Developer Tools, 详情请参见 第 2.2.4 节 “Firefox 和 Firebug”. Firefox 也有内建的功能可以查看 HTML 和 CSS.

分析布局问题

Check layouts for potential problems 按钮可以分析目前可见的 UI, 并生成一份布局中可能存在的问题的报告. 检测出来的所有布局问题会显示在日志中, 也会输出到控制台.

图 11.7. Debug 窗口中显示的布局分析结果.



点击报告中的问题，可以将 UI 中对应的组件高亮显示。

造成布局问题的最常见原因是：将一个某方向(高度或宽度)为相对尺寸的组件，放置在一个未指定尺寸的容器(布局)之内。比如，将一个 100% 宽度的 **Button** 放置在一个未指定宽度的 **VerticalLayout** 之内。这种情况下，布局错误的结果见图 11.7 “Debug 窗口中显示的布局分析结果。”。

CustomLayout 组件不可以象其他布局一样进行分析。对于自定义布局，布局分析按钮会分析它包含的所有相对尺寸组件，并检查是否有某个相对尺寸被计算为 0，以至于组件不可见。错误日志中会对每个这样的不可见组件显示一条警告信息。高亮显示这些组件是没有意义的，因为它已经处于不可见状态了，因此当你选中这样一条错误信息时，如果可能的话，会高亮显示组件的父布局。

显示使用的连接器

最后一个按钮是，**Show used connectors and how to optimize widget set**，它会列出目前所有可见的连接器。它还会产生一个连接器群的加载器工厂，你可以用来定制 widget 群，使其中只包含目前 UI 中真实使用到的 widget。注意，这个按钮只会列出目前 UI 状态下可见的连接器，实际存在的连接器通常会比显示的要多。

11.3.6. 通信 Log

Communication tab 显示搜有的服务器请求。你可以展开这些请求，来查看它的细节，比如使用了哪个连接器，等等。点击连接器会高亮显示 UI 中对应的元素。

你可以使用 Firefox 中的 Firebug 或 Chrome 中的 Developer Tools，来查看请求和应答的更详细信息。

11.3.7. Debug 模式

窗口中的 **Menu** tab 打开一个子菜单，可以在这里选择很多设置。在这个菜单中你还可以启动 GWT SuperDevMode，详情请参见第 13.6 节“调试客户端代码”。

11.4. 请求处理器(Request Handler)

请求处理器是一种便利的工具，可以用于捕捉请求参数或生成动态内容，比如 HTML，图像，PDF，或其他内容。你可以便利地提供 HTTP 内容，也可以提供流资源，详情请参见第 4.4.5 节“流资源”。但流资源只能用于 Vaadin 应用程序之内，比如 **Image** 组件内。请求处理器允许对发送给应用程序 URL 的 HTTP 请求进行应答，可处理 GET 或 POST 参数。你也可以使用独立的 Servlet 来产生动态内容，但请求处理器是与 Vaadin Session 关联的，而且它可以方便地访问所有的 Session 数据。

要相应一个请求，你需要实现 **RequestHandler** 接口。**handleRequest()** 方法的参数是 session，request，以及 response 对象。

如果请求处理器输出了应答，它必须返回 `true`。这个返回值会阻止 Vaadin 继续执行其他可能存在的请求处理器。否则，应该返回 `false`，此时其他请求处理器就可以继续执行，并返回 HTTP 应答。最终，如果没有任何一个请求处理器返回了应答，这是会创建并初始化一个 UI。

下面的例子中，我们捕捉发送到 Servlet URL 之下的子路径的一个请求，并返回一个纯文本应答。Servlet 路径由 context 路径和 Servlet (子)路径组成。此外的任何附加路径都会被传递给请求处理器，成为 request 对象的 `pathInfo` 属性。比如，假定全路径是 /myapp/myui/rhexample，此时 `pathInfo` 将是 /rhexample。此外，请求参数也是允许的。

```
VaadinSession.getCurrent().addRequestHandler(  
    new RequestHandler() {  
        @Override  
        public boolean handleRequest(VaadinSession session,
```

```

        VaadinRequest request,
        VaadinResponse response)
    throws IOException {
    if ("/rhexample".equals(request.getPathInfo())) {
        response.setContentType("text/plain");
        response.getWriter().append(
            "Here's some dynamically generated content.\n"+
            "Time: " + (new Date()).toString());
        return true; // We wrote a response
    } else
        return false; // No response was written
}
});

// Find out the base path for the servlet
String servletPath = VaadinServlet.getCurrent()
    .getServletContext().getContextPath() + VaadinServletService
    .getCurrentServletRequest().getServletPath();

// Display the page in a popup window
Link open = new Link("Click to Show the Page",
    new ExternalResource(servletPath + "/rhexample"),
    "_blank", 500, 350, BorderStyle.DEFAULT);
layout.addComponent(open);

```

11.5. 快捷键

Vaadin 提供了为 Field 组件和默认按钮定义快捷键的便利方法, 还提供了基于动作(Action)的快捷键绑定通用低阶 API.

11.5.1. 默认按钮的快捷键

你可以向按钮添加或设定一个 点击快捷键, 通过这种方法可以将这个按钮设置为“默认”按钮; 在窗口内的任何组件上, 按下这个预定义的快捷键, 通常是 **Enter** 键, 都会触发对象按钮的点击事件.

定义点击快捷键可以使用 `setClickShortcut()` 方法:

```
// Have an OK button and set it as the default button
Button ok = new Button("OK");
ok.setClickShortcut(KeyCode.ENTER);
ok.addStyleName(Reindeer.BUTTON_DEFAULT);
```

`BUTTON_DEFAULT` 样式名会将按钮高亮显示, 表示这个按钮目前是默认按钮; 此时字体通常会更粗一些, 具体如何由 theme 觉得. 运行结果见 图 11.8 “设置了点击快捷键的默认按钮”.

图 11.8. 设置了点击快捷键的默认按钮



11.5.2. 控制 Field 的焦点快捷键

你可以定义一个让 Field 组件(继承自 **AbstractField** 的组件)获得焦点的快捷键, 方法是向 Field 添加一个 **FocusShortcut** 作为快捷键的监听器.

FocusShortcut 的构造函数第一个参数为 Field 组件, 第二个参数是快捷键代码, 第三个参数可选, 是修饰键列表, 详情请参见第 11.5.4 节“对键码(Key Code)和修饰键(Modifier Key)的支持”.

```
// A field with Alt+N bound to it
TextField name = new TextField("Name (Alt+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
        ModifierKey.ALT));
layout.addComponent(name);
```

你也可以使用更简短的注解来指定快捷键, 快捷键以&符号来标记.

```
// A field with Alt+A bound to it, using shorthand notation
TextField address = new TextField("Address (Alt+A)");
address.addShortcutListener(
    new AbstractField.FocusShortcut(address, "&Address"));
```

这种方式对国际化特别有用, 因为可以从翻译过的文字中自动检测出快捷键.

11.5.3. 通用的快捷键 Action

可以使用 **ShortcutAction** 类, 以动作的形式来定义快捷键. 这个类继承了共通基类 **Action**, 我们在 **Tree** 和 **Table** 的上下文菜单例子程序中使用过 **Action** 类. 目前, 唯一能够接受 **ShortcutAction** 的类是 **Window** 和 **Panel**.

为了处理键盘按下事件, 你需要实现 **Handler** 接口来定义一个动作处理器. 这个接口有两个方法需要实现: `getActions()` 和 `handleAction()`.

`getActions()` 方法必须针对组件返回一个 **Action** 对象数组, 对象组件通过方法的第二个参数来指定, 也就是动作的 `sender`. 对于键盘快捷键, 你可以使用 **ShortcutAction**. 这个方法的实现示例如下:

```
// Have the unmodified Enter key cause an event
Action action_ok = new ShortcutAction("Default key",
    ShortcutAction.KeyCode.ENTER, null);

// Have the C key modified with Alt cause an event
Action action_cancel = new ShortcutAction("Alt+C",
    ShortcutAction.KeyCode.C,
    new int[] { ShortcutAction.ModifierKey.ALT });

Action[] actions = new Action[] {action_cancel, action_ok};

public Action[] getActions(Object target, Object sender) {
    if (sender == myPanel)
        return actions;

    return null;
}
```

返回的 **Action** 数组必须是 static 的, 你也可以针对不同的 `sender` 按照你的需要动态创建不同的结果.

ShortcutAction 的构造函数第一个参数是 `action` 的一个符号化的标题; 在目前的实现中, 这个标题与 `Shortcut Action` 关系不大, 但将来可能会被同时用在菜单和 `Shortcut Action` 中. 第二个参数是键码(key code), 第三个参数是修饰键列表, 详情请参见第 11.5.4 节“对键码(Key Code)和修饰键(Modifier Key)的支持”.

下例演示如何在 UI 中定义一个默认按钮，以及一个通常的快捷键，使用 **Alt+C** 作为 **Cancel** 按钮的快捷键。

```
public class DefaultButtonExample extends CustomComponent
    implements Handler {
    // Define and create user interface components
    Panel panel = new Panel("Login");
    FormLayout formlayout = new FormLayout();
    TextField username = new TextField("Username");
    TextField password = new TextField("Password");
    HorizontalLayout buttons = new HorizontalLayout();

    // Create buttons and define their listener methods.
    Button ok = new Button("OK", this, "okHandler");
    Button cancel = new Button("Cancel", this, "cancelHandler");

    // Have the unmodified Enter key cause an event
    Action action_ok = new ShortcutAction("Default key",
        ShortcutAction.KeyCode.ENTER, null);

    // Have the C key modified with Alt cause an event
    Action action_cancel = new ShortcutAction("Alt+C",
        ShortcutAction.KeyCode.C,
        new int[] { ShortcutAction.ModifierKey.ALT });

    public DefaultButtonExample() {
        // Set up the user interface
        setCompositionRoot(panel);
        panel.addComponent(formlayout);
        formlayout.addComponent(username);
        formlayout.addComponent(password);
        formlayout.addComponent(buttons);
        buttons.addComponent(ok);
        buttons.addComponent(cancel);

        // Set focus to username
        username.focus();

        // Set this object as the action handler
        panel.addActionHandler(this);
    }

    /**
     * Retrieve actions for a specific component. This method
     * will be called for each object that has a handler; in
     * this example just for login panel. The returned action
     * list might as well be static list.
     */
    public Action[] getActions(Object target, Object sender) {
        System.out.println("getActions()");
        return new Action[] { action_ok, action_cancel };
    }

    /**
     * Handle actions received from keyboard. This simply directs
     * the actions to the same listener methods that are called
     * with ButtonClick events.
     */
    public void handleAction(Action action, Object sender,
```

```

        Object target) {
    if (action == action_ok) {
        okHandler();
    }
    if (action == action_cancel) {
        cancelHandler();
    }
}

public void okHandler() {
    // Do something: report the click
    formlayout.addComponent(new Label("OK clicked. "
        + "User=" + username.getValue() + ", password="
        + password.getValue()));
}

public void cancelHandler() {
    // Do something: report the click
    formlayout.addComponent(new Label("Cancel clicked. User="
        + username.getValue() + ", password="
        + password.getValue()));
}
}

```

注意，键盘动作目前只能绑定到 **Panel** 和 **Window**. 如果你的组件本身要求使用某个键，就会发生冲突。比如，多行的 **TextField** 需要使用 **Enter** 键。当某个特定组件获得焦点时，目前没有办法可以将快捷键过滤出来，因此你需要注意避免这类冲突。

11.5.4. 对键码(Key Code)和修饰键(Modifier Key)的支持

快捷键的定义需要一个键码，用来指定按下的键和修饰键，比如 Shift, Alt, 或 Ctrl.

键码定义在 **ShortcutAction.KeyCode** 接口中，如下：

A 到 Z 键
通常的字母键

F1 到 F12
功能键

BACKSPACE, DELETE, ENTER, ESCAPE, INSERT, TAB
控制键

NUM0 到 NUM9
右侧小键盘上的数字键

ARROW_DOWN, ARROW_UP, ARROW_LEFT, ARROW_RIGHT
方向键

HOME, END, PAGE_UP, PAGE_DOWN
其他移动键

修饰键定义在 **ShortcutAction.ModifierKey** 中，如下：

ModifierKey.ALT
Alt 键

ModifierKey.CTRL
Ctrl 键

ModifierKey.SHIFT
Shift 键

所有构造函数或方法，如果接受修饰键为参数，那么这个修饰键参数会出现在键码参数之后，并且是以逗号分隔的变长参数列表。比如，下例定义了一个 **Ctrl+Shift+N** 的按键组合。

```
TextField name = new TextField("Name (Ctrl+Shift+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
        ModifierKey.CTRL,
        ModifierKey.SHIFT)));
```

支持的快捷键组合

实际可用的键盘组合在不同的浏览器中可能会有很大区别，因为大多数浏览器都有大量的内建快捷键，这些浏览器自用的快捷键就不能在 Web 应用程序中使用了。比如，Mozilla Firefox 几乎允许绑定任意的按键组合，而 Opera 甚至连 Alt 键都不允许使用。其他浏览器大多处于这两种极端情况之间。此外，操作系统也可能保留某些按键组合，某些计算机制造厂商也预定义了他们自己的系统按键组合。

11.6. 打印

Vaadin 对于打印没有任何特别支持。打印有两种基本方式 - 使用应用程序服务器端控制的打印机，或者由用户在 Web 浏览器中打印。应用程序服务器端的打印工作与 UI 几乎无关，你只需要控制好打印命令，使它不要阻塞服务器请求，这一点可以通过使用独立线程运行打印命令来实现。

对于客户端的打印，大多数浏览器都支持 Web 页面的打印功能。你可以打印当前页面，也可以打印某个特别的打印页面。页面可以使用特别的 CSS 规则来控制打印时的样式，在打印时你可以隐藏不希望出现的元素。除 Vaadin UI 外，你还可以打印其他内容，比如 HTML 或 PDF。

11.6.1. 打印浏览器窗口

在浏览器中启动打印机，Vaadin 没有提供特别的支持，但你可以通过 JavaScript `print()` 方法很简单地实现，这个方法会打开浏览器的打印窗口。

```
Button print = new Button("Print This Page");
print.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        // Print the current page
        JavaScript.getCurrent().execute("print();");
    }
});
```

上例中的按钮会打印当前页面，包括按钮本身。你可以使用 CSS 来隐藏这类元素，也可以针对打印版进行专门的样式控制。专门针对打印的样式定义需要包括在 CSS 的 `@media print {}` 之内。

11.6.2. 打开打印窗口

你可以打开一个浏览器窗口，其中内容是一个专用于打印内容的 UI，并且自动启动打印动作。

```
public static class PrintUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
```

```

// Have some content to print
setContent(new Label(
    "<h1>Here's some dynamic content</h1>\n" +
    "<p>This is to be printed.</p>",
    ContentMode.HTML));

// Print automatically when the window opens
JavaScript.getCurrent().execute(
    "setTimeout(function() {" +
        "print(); self.close();}, 0);");
}

...
}

// Create an opener extension
BrowserWindowOpener opener =
    new BrowserWindowOpener(PrintUI.class);
opener.setFeatures("height=200,width=400,resizable");

// A button to open the printer-friendly page.
Button print = new Button("Click to Print");
opener.extend(print);

```

浏览器如何打开窗口, 是作为真实的(弹出)窗口, 还是作为一个 tab, 由浏览器决定. 打印完成后, 我们调用 JavaScript 的 `close()` 方法来自动关闭窗口.

11.6.3. 打印 PDF

为了以 PDF 形式打印内容, 你需要提供可下载的 PDF 内容, 这个内容可以是静态的也可以是动态生成的, 比如 **StreamResource** 形式.

你可以使用 **Link** 来让用户打开这个资源, 也可以使用与 **PopupWindowOpener** 绑定的其他组件. 当这个 Link 或 PopupWindowOpener 被点击之后, 浏览器会在浏览器内部打开 PDF, 或在外部阅读器(比如 Adobe Reader)中打开 PDF, 或者让用户将文档保存为本地文件.

有一个重要问题需要注意, 点击 **Link** 或 **PopupWindowOpener** 是发生在客户端的操作. 如果你通过某个 UI 状态创建出动态 PDF 内容, 这之前你无法使 Link 或 Opener 有效, 因为点击它不能得到当前 UI 的内容. 相反, 你必须在 Link 或 Opener 被点击之前就创建资源对象. 这种方案通常需要两步操作, 或者需要将打印操作放在另外的视图中.

```

// A user interface for a (trivial) data model from which
// the PDF is generated.
final TextField name = new TextField("Name");
name.setValue("Slartibartfast");

// This has to be clicked first to create the stream resource
final Button ok = new Button("OK");

// This actually opens the stream resource
final Button print = new Button("Open PDF");
print.setEnabled(false);

ok.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Create the PDF source and pass the data model to it
        StreamSource source =
            new MyPdfSource((String) name.getValue());
    }
});

```

```

// Create the stream resource and give it a file name
String filename = "pdf_printing_example.pdf";
StreamResource resource =
    new StreamResource(source, filename);

// These settings are not usually necessary. MIME type
// is detected automatically from the file name, but
// setting it explicitly may be necessary if the file
// suffix is not ".pdf".
resource.setMimeType("application/pdf");
resource.getStream().setParameter(
    "Content-Disposition",
    "attachment; filename="+filename);

// Extend the print button with an opener
// for the PDF resource
BrowserWindowOpener opener =
    new BrowserWindowOpener(resource);
opener.extend(print);

name.setEnabled(false);
ok.setEnabled(false);
print.setEnabled(true);
}
});

layout.addComponent(name);
layout.addComponent(ok);
layout.addComponent(print);

```

11.7. 与 Google App Engine 的集成

本节内容还没有针对 Vaadin 7 版本更新完毕.

针对 Vaadin 应用程序在 Google App Engine (GAE) 中的运行, Vaadin 提供了支持. GAE 中运行的最重要的要求是能够将应用程序状态序列化保存. Vaadin 应用程序通过 **java.io.Serializable** 接口实现序列化功能.

要作为 GAE 应用程序运行, 应用程序必须使用 **GAEVaadinServlet** 而不是 **VaadinServlet**, 而且对所有的持久化类都需要实现 **java.io.Serializable** 接口. 你还需要在 appengine-web.xml 中允许 session:

```
<sessions-enabled>true</sessions-enabled>
```

Vaadin 工程向导可以创建 GAE 部署需要的配置文件. 详情请参见 第 2.5.1 节“创建工程”. 当选择 Google App Engine 部署配置时, 向导会按照 GAE Servlet 规约来创建工程结构, 而不是使用通常的 Servlet 规约. 主要区别是:

- 源代码目录: src/main/java
- 输出目录: war/WEB-INF/classes
- Web 内容目录: war

规则与限制

在 Google App Engine 中运行 Vaadin 应用程序存在以下规则和限制:

- 不要使用 session 作为数据存储的目的, 通常的 App Engine 限制也适用于这个问题 (session 不同步, 也就是说, 是不可靠的).
- Vaadin 在互斥锁(mutex)中使用 memcache, 其中键值的形式是 _vmutex<sessionid>.
- Vaadin **WebApplicationContext** 类会被分别序列化进入 memcache 和 datastore 中; memcache 中键值是 _vac<sessionid>, datastore 中的 entity kind 是 _vac, ID 类型是 _vac<sessionid>.
- 向客户端提供 **ConnectorResource** 时 (比如 **ClassResource.getStream()**), 不要更新应用程序状态.
- 在 **TransactionListener** 之内 不要更新应用程序状态(如果一定要做, 需要特别小心) - 即使在应用程序未被锁定, 不会被序列化的时刻(比如使用 **ConnectorResource** 时)这个监听器也会被调用, 因此对应用程序状态的变更可能会丢失(对于那些可以丢弃的信息, 也就是说, 只对当前请求有效的那些信息, 是可以更新的).
- 上传文件时, 应用程序会保持锁死 - 进度条是不可用的.

11.8. 共通的安全问题

11.8.1. 处理用户输入内容, 防止跨站脚本攻击

在很多组件内都可以使用原生的 HTML 内容, 比如在 **Label** 和 **CustomLayout** 内, 以及在 tooltip 和通知信息内. 这种情况下, 如果内容有可能来自用户输入的话, 在显示这些内容之前你必须确保内容是安全的. 否则, 恶意用户可以很容易地在这类组件中注入攻击性的 JavaScript 脚本, 发起跨站脚本攻击. 关于跨站脚本攻击的更多介绍, 请阅读其他相关文档.

可以很容易地注入攻击代码, 方法是使用 `<script>` 标记, 或者使用其他 tag 的事件属性, 比如 `onLoad` 属性. 跨站脚本漏洞与浏览器相关, 取决于各种不同浏览器执行脚本的具体情况.

因此, 如果需要将一个用户创建的内容显示给其他用户, 内容首先要进行安全处理. 对用户输入内容的安全处理并不存在一个通用的方法, 因为不同的应用程序可能会允许不同的输入内容. 删去所有的 (X)HTML tag 是最简便的方法, 但某些应用程序会需要允许使用 (X)HTML 内容. 因此, 应用程序必须按照各自的需求来对内容进行安全处理.

字符编码问题会使得内容的安全处理变得更加困难, 因为攻击性 tag 可能是被编码过的, 因此安全处理程序可能无法识别它们. 比如, 攻击者可以利用 HTML 字符实体(character entity), 使用变宽字符集, 比如 UTF-8 或各种 CJK 字符集, 通过同一个字符的多种表达方式来欺骗安全处理程序. 最简单的例子, 你可以使用 `<` 和 `>` 来输入 `<` 和 `>`. 输入内容还可以是格式不正确的, 安全处理程序必须有能力使用与浏览器完全相同的方式来解释这段输入, 而不同的浏览器对格式不正确的 HTML 以及变宽字符集编码的解释有可能很不相同.

注意, 这个问题对于 **RichTextArea** 的输入也是有效的, 它的内容以 HTML 形式从浏览器传送到服务器端, 而且是未经过安全处理的. 由于 **RichTextArea** 组件的目的本身就是为了允许输入格式化的文本, 所以你不能简单地删除所有的 HTML tag. 此外还有很多 tag 属性, 比如 `style`, 在安全处理过程中也应该保留下.

11.9. 应用程序内的导航跳转

简单的 Vaadin 应用程序不存在 Web 页面跳转的问题，因为它们象所有的 Ajax 应用程序一样，通常只运行在一个页面内。但是，很多情况下，应用程序也可能带有多个视图，用户应该能在这些视图之间导航跳转。Vaadin 中的 **Navigator** 可以用来解决大多数这类跳转导航问题。导航器管理下的多个视图会自动获得不同的 URI 片段，这些 URI 片段可以用来作为视图及其状态的书签，还可以在浏览器的历史记录中向前或向后跳转。

11.9.1. 导航设置

Navigator 类管理多个 视图，这些视图需要实现 View 接口。视图可以预先注册，也可以通过 视图提供者 来获得。注册时，视图必须拥有一个名称标识符，然后使用 addView() 方法加入到导航器中。你可以在任何时刻注册新的视图。一旦注册完成，你就可以使用 navigateTo() 方法在视图中导航跳转。

Navigator 使用组件容器来管理导航跳转，容器可以是 ComponentContainer（大多数布局组件是 ComponentContainer）或 SingleComponentContainer（**UI**, **Panel**, 或 **Window** 是 SingleComponentContainer）。组件容器通过 ViewDisplay 来管理。ViewDisplay 有两种：**ComponentContainerViewDisplay** 和 **SingleComponentContainerViewDisplay**，分别对应于前面的两种组件容器类型。通常，你可以让导航器来创建内部的 ViewDisplay，下例中我们就是这样做的，但你也可以自己创建它，以便对它进行定制。

我们来看看下面的 UI，其中包含两个视图：开始视图和主视图。这里，我们使用 enum 来定义它们的名称，以便保证类型安全。我们使用 UI 类自身来管理导航跳转，这里的 UI 类是一个 SingleComponentContainer。

```
public class NavigatorUI extends UI {
    Navigator navigator;
    protected static final String MAINVIEW = "main";

    @Override
    protected void init(VaadinRequest request) {
        getPage().setTitle("Navigation Example");

        // Create a navigator to control the views
        navigator = new Navigator(this, this);

        // Create and register the views
        navigator.addView("", new StartView());
        navigator.addView(MAINVIEW, new MainView());
    }
}
```

Navigator 自动设定应用程序 URL 的 URI 片段。它还会在页面中注册一个 URIFragmentChangedListener（详情请参见 第 11.11 节“管理 URI 片段”），当用户手动输入某个地址时，或者在浏览器内发生跳转时，这个 URIFragmentChangedListener 用来显示 URI 片段对应的视图。这个功能也允许在应用程序内记录下浏览器的浏览历史。

视图提供者 (View Provider)

你可以使用 视图提供者 来动态创建新的视图，视图提供者需要实现 ViewProvider 接口。视图提供者通过 addProvider() 方法注册到 **Navigator** 中。

ClassBasedViewProvider 是一个视图提供者，它可以根据视图名称，动态地创建某个预先指定的视图类的新实例。

`StaticViewProvider` 根据视图名称返回一个既有的视图实例. **Navigator** 中的 `addView()` 方法实际上只是一个便捷方法, 其中的内容是为注册的每一个视图创建一个 `StaticViewProvider`.

视图变化监听器

实现 `ViewChangeListener` 接口, 并将它添加到 **Navigator** 中, 这样就可以处理视图的变化事件了. 当试图发生变化时, 监听器会收到一个 `ViewChangeEvent` 对象, 其中的信息包含旧视图和目前活动视图, 目前活动视图的名称, 以及 URI 片段参数.

11.9.2. 实现 View

视图可以是实现了 `View` 接口的任何对象. 当导航器的 `navigateTo()` 方法被调用时, 或者使用与视图关联的 URI 片段来打开应用程序时, 导航器会切换到这个视图, 并调用它的 `enter()` 方法.

继续前面的例子, 我们的开始视图很简单, 它只用来让用户跳转到主视图. 当用户跳转到开始视图时, 它只会弹出一个通知信息, 并显示一个导航跳转按钮.

```
/** A start view for navigating to the main view */
public class StartView extends VerticalLayout implements View {
    public StartView() {
        setSizeFull();

        Button button = new Button("Go to Main View",
            new Button.ClickListener() {
                @Override
                public void buttonClick(ClickEvent event) {
                    navigator.navigateTo(MAINVIEW);
                }
            });
        addComponent(button);
        setComponentAlignment(button, Alignment.MIDDLE_CENTER);
    }

    @Override
    public void enter(ViewChangeEvent event) {
        Notification.show("Welcome to the Animal Farm");
    }
}
```

如上例所示, 你可以在构造函数中初始化视图的内容, 或者也可以在 `enter()` 方法中初始化. 后一种方法的优点是, 视图只有在关联到视图容器时才会同时关联到 UI, 而在使用构造函数的方案中并不是如此.

11.9.3. 处理 URI 片段路径

URL 中的 URI 片段是指 # 字符以后的部分. 它用来表达 UI 内部的 URL, 因为在 URL 中, 只有这一部分可以通过页面内的 JavaScript 来改变, 而同时又不会导致页面重装载. 带 URI 片段的 URL 与其他 URL 一样, 可以用在超链接和浏览器书签中, 也可以用于浏览器访问履历. 此外, #! 符号表示页面是一个有状态的 AJAX 页面, 可以被搜索引擎抓取. 搜索引擎抓取页面时要求针对"可搜索内容"的 URL, 应用程序也能正常应答. URI 片段由 **Page** 管理, 它负责提供相关的低阶 API.

在 **Navigator** 中使用 URI 片段有两种方式: 跳转到某个视图, 或者跳转到视图内的某个状态. `navigateTo()` 方法接受的 URI 片段, 最前端是视图名称, 再一个斜线("/")之后可以附带片段参数. 这些参数会被传递给 `View` 的 `enter()` 方法.

下例中, 我们实现视图内部的导航条转.

```
/** Main view with a menu */
public class MainView extends VerticalLayout implements View {
    Panel panel;

    // Menu navigation button listener
    class ButtonListener implements Button.ClickListener {

        String menuitem;
        public ButtonListener(String menuitem) {
            this.menuitem = menuitem;
        }

        @Override
        public void buttonClick(ClickEvent event) {
            // Navigate to a specific state
            navigator.navigateTo(MAINVIEW + "/" + menuitem);
        }
    }

    public MainView() {
        setSizeFull();

        // Layout with menu on left and view area on right
        HorizontalLayout hLayout = new HorizontalLayout();
        hLayout.setSizeFull();

        // Have a menu on the left side of the screen
        Panel menu = new Panel("List of Equals");
        menu.setHeight("100%");
        menu.setWidth(null);
        VerticalLayout menuContent = new VerticalLayout();
        menuContent.addComponent(new Button("Pig",
                new ButtonListener("pig")));
        menuContent.addComponent(new Button("Cat",
                new ButtonListener("cat")));
        menuContent.addComponent(new Button("Dog",
                new ButtonListener("dog")));
        menuContent.addComponent(new Button("Reindeer",
                new ButtonListener("reindeer")));
        menuContent.addComponent(new Button("Penguin",
                new ButtonListener("penguin")));
        menuContent.addComponent(new Button("Sheep",
                new ButtonListener("sheep")));
        menuContent.setWidth(null);
        menuContent.setMargin(true);
        menu.setContent(menuContent);
        hLayout.addComponent(menu);

        // A panel that contains a content area on right
        panel = new Panel("An Equal");
        panel.setSizeFull();
        hLayout.addComponent(panel);
        hLayout.setExpandRatio(panel, 1.0f);

        addComponent(hLayout);
        setExpandRatio(hLayout, 1.0f);

        // Allow going back to the start
        Button logout = new Button("Logout",
                new Button.ClickListener() {

```

```

@Override
public void buttonClick(ClickEvent event) {
    navigator.navigateTo("");
}
});

addComponent(logout);
}

@Override
public void enter(ViewChangeEvent event) {
    VerticalLayout panelContent = new VerticalLayout();
    panelContent.setSizeFull();
    panelContent.setMargin(true);
    panel.setContent(panelContent); // Also clears

    if (event.getParameters() == null
        || event.getParameters().isEmpty()) {
        panelContent.addComponent(
            new Label("Nothing to see here, " +
                "just pass along."));
        return;
    }

    // Display the fragment parameters
    Label watching = new Label(
        "You are currently watching a " +
        event.getParameters());
    watching.setSizeUndefined();
    panelContent.addComponent(watching);
    panelContent.setComponentAlignment(watching,
        Alignment.MIDDLE_CENTER);

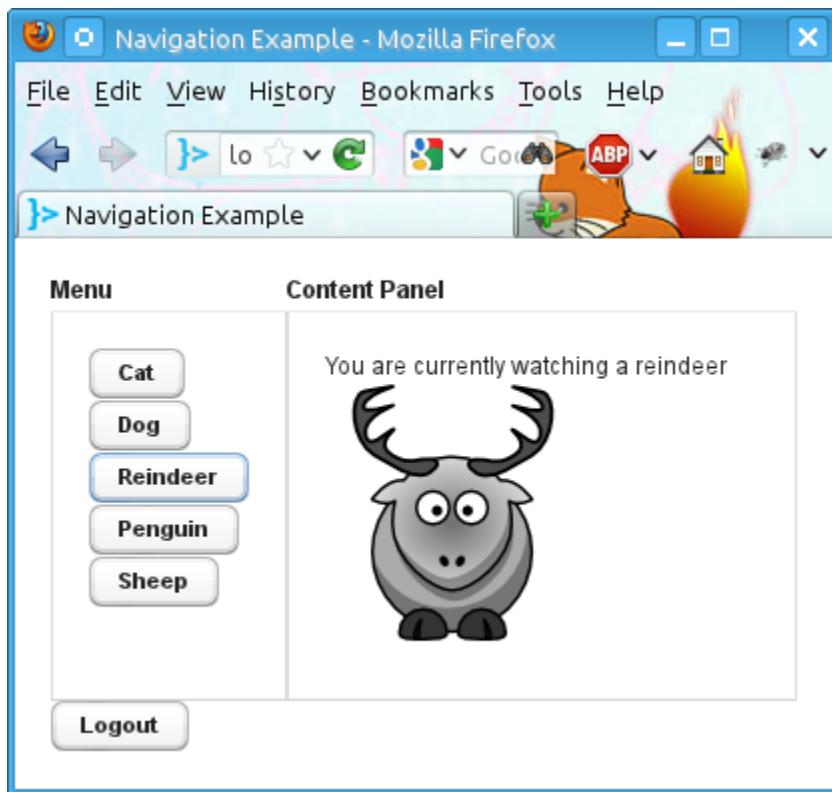
    // Some other content
    Embedded pic = new Embedded(null,
        new ThemeResource("img/" + event.getParameters() +
            "-128px.png"));
    panelContent.addComponent(pic);
    panelContent.setExpandRatio(pic, 1.0f);
    panelContent.setComponentAlignment(pic,
        Alignment.MIDDLE_CENTER);

    Label back = new Label("And the " +
        event.getParameters() + " is watching you");
    back.setSizeUndefined();
    panelContent.addComponent(back);
    panelContent.setComponentAlignment(back,
        Alignment.MIDDLE_CENTER);
}
}

```

主视图的运行结果见 图 11.9 “导航器的主视图”. 图中显示的状态，对应的 URL 应该是 <http://localhost:8080/myapp#!main/reindeer>.

图 11.9. 导航器的主视图



11.10. 应用程序高级架构

第 4.2 节“构建 UI”介绍过关于应用程序架构的基本信息，本节中我们详细讨论这个问题。本节还讨论 Vaadin 应用程序经常使用到的一些更高级的设计模式。

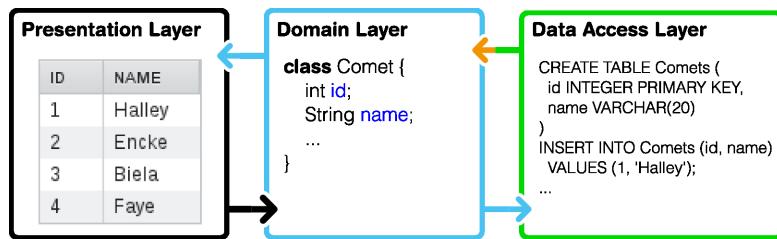
11.10.1. 分层架构

分层架构可能是最普遍使用的架构，它的每一层都负责一个清楚的、独立的职责。通常，应用程序至少会遵循三层架构原则：

- 用户界面层(或者叫表现层)
- 业务层
- 数据存储层

这样的架构首先需要一个 业务模型，其中定义应用程序的数据模型及“业务逻辑”，通常使用 POJO 来实现。用户界面构建在业务模型之上，针对我们情况，用户界面使用 Vaadin Framework 来构建。Vaadin UI 可以通过 Vaadin Data Model 直接绑定到数据模型，详情请参见 第 8 章 组件与数据绑定。在业务模型之下存在的是数据存储层，比如关系型数据库。各层之间的依赖关系受到严格限制，因此高层可以依赖低层，但不会出现反方向的依赖。

图 11.10. 三层架构



应用程序层(或者叫服务层)常常会从业务层中分离出来,以服务的形式实现业务逻辑,它可以被用户界面层使用,也可以被其他用户使用。在 Java EE 环境中,企业 JavaBean (EJB)通常用来构建这个层。

基础层(或者叫数据访问层)常常从数据存储层中分离出来,用来实现对数据存储逻辑的抽象。比如,它可能用到数据持久化方案,比如 JPA 和 EJB 容器。这一层与 Vaadin 的关系主要是,通过 JPAContainer 实现 Vaadin 组件与数据的绑定,详情请参见第 19 章 Vaadin JPAContainer。

11.10.2. 模型(Model)-视图(View)-展现者(Presenter) 模式

在使用 Vaadin 开发大型应用程序时,模型(Model)-视图(View)-展现者(Presenter) (MVP)模式是最常见的模式之一。它类似于旧的模型(Model)-视图(View)-控制器(Controller) (MVC) 模式, MVC 模式在 Vaadin 开发中意义不大。我们不使用与实现相关的控制器,而是使用与实现无关的展现者,它负责通过接口来操纵视图,视图不直接与模型发生交互。这种设计比 MVC 模式更好地隔离了视图的实现,而且可以更简单地实现对展现者和模型的单元测试。

图 11.11. 模型(Model)-视图(View)-展现者(Presenter) 模式

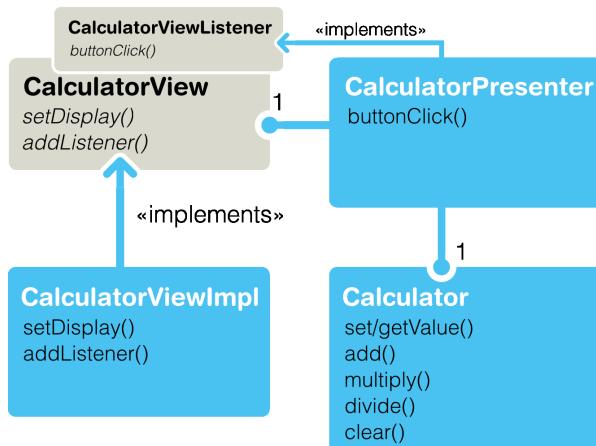


图 11.11 “模型(Model)-视图(View)-展现者(Presenter) 模式” 使用一个简单的计算器展示了 MVP 模式。业务模型由 **Calculator** 类实现,其中包含数据模型和一些业务逻辑。**CalculatorViewImpl** 是使用 Vaadin 实现的视图,视图由 **CalculatorView** 接口定义。**CalculatorPresenter** 类负责处理 UI 逻辑。视图中接受到的 UI 事件被翻译为与实现无关的事件,并交给展现者去处理(视图的实现类可以简单地调用展现者)。

我们首先来看看下例,其中模型和视图通过展现者绑定在一起:

```
// Create the model and the Vaadin view implementation
CalculatorModel model = new CalculatorModel();
```

```

CalculatorViewImpl view = new CalculatorViewImpl();

// The presenter binds the model and view together
new CalculatorPresenter(model, view);

// The view implementation is a Vaadin component
layout.addComponent(view);

```

你可以将这个视图添加到 Vaadin 应用程序中的任何位置, 因为它是一个复合组件.

模型

我们的业务模型非常简单, 其中只有一个值, 和对这个值的一系列操作.

```

/** The model */
class CalculatorModel {
    private double value = 0.0;

    public void clear() {
        value = 0.0;
    }

    public void add(double arg) {
        value += arg;
    }

    public void multiply(double arg) {
        value *= arg;
    }

    public void divide(double arg) {
        if (arg != 0.0)
            value /= arg;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }
}

```

视图

MVP 模式中视图的目的是用来显示数据, 并接受用户的操作. 它依赖于用户对展现者的操作, 这种操作与具体的视图实现无关, 也就是说, 展现者中不使用 Vaadin 事件. 这样设计的目的是允许 UI 接口可以有多种不同的实现.

```

interface CalculatorView {
    public void setDisplay(double value);

    interface CalculatorViewListener {
        void buttonClick(char operation);
    }
    public void addListener(CalculatorViewListener listener);
}

```

对于视图的设计可以有几种选择。它可以在构造函数中接受监听器，或者它可以直接知道展现者。这里，我们将按钮的点击作为一个与实现无关的事件转发出去。

由于我们在使用 Vaadin，我们使用 Vaadin 来实现上面的接口，如下：

```
class CalculatorViewImpl extends CustomComponent
    implements CalculatorView, ClickListener {
private Label display = new Label("0.0");

public CalculatorViewImpl() {
    GridLayout layout = new GridLayout(4, 5);

    // Create a result label that spans over all
    // the 4 columns in the first row
    layout.addComponent(display, 0, 0, 3, 0);

    // The operations for the calculator in the order
    // they appear on the screen (left to right, top
    // to bottom)
    String[] operations = new String[] {
        "7", "8", "9", "/", "4", "5", "6",
        "*", "1", "2", "3", "-", "0", "=", "C", "+";
    }

    // Add buttons and have them send click events
    // to this class
    for (String caption: operations)
        layout.addComponent(new Button(caption, this));

    setCompositionRoot(layout);
}

public void setDisplay(double value) {
    display.setValue(Double.toString(value));
}

/* Only the presenter registers one listener... */
List<CalculatorViewListener> listeners =
    new ArrayList<CalculatorViewListener>();

public void addListener(CalculatorViewListener listener) {
    listeners.add(listener);
}

/** Relay button clicks to the presenter with an
 * implementation-independent event */
@Override
public void buttonClick(ClickEvent event) {
    for (CalculatorViewListener listener: listeners)
        listener.buttonClick(event.getButton()
            .getCaption().charAt(0));
}
}
```

展现者

MVP 中的展现者是一个中间人，它负责所有的用户交互逻辑，但使用一种与实现无关的方式，因此它实际上并不意识到 Vaadin 的存在。它负责在视图中显示数据，并负责从视图中接受用户的交互。

```

class CalculatorPresenter
    implements CalculatorView.CalculatorViewListener {
    CalculatorModel model;
    CalculatorView view;

    private double current = 0.0;
    private char lastOperationRequested = 'C';

    public CalculatorPresenter(CalculatorModel model,
                               CalculatorView view) {
        this.model = model;
        this.view = view;

        view.setDisplay(current);
        view.addListener(this);
    }

    @Override
    public void buttonClick(char operation) {
        // Handle digit input
        if ('0' <= operation && operation <= '9') {
            current = current * 10
                + Double.parseDouble("") + operation);
            view.setDisplay(current);
            return;
        }

        // Execute the previously input operation
        switch (lastOperationRequested) {
        case '+':
            model.add(current);
            break;
        case '-':
            model.add(-current);
            break;
        case '/':
            model.divide(current);
            break;
        case '*':
            model.multiply(current);
            break;
        case 'C':
            model.setValue(current);
            break;
        } // '=' is implicit

        lastOperationRequested = operation;

        current = 0.0;
        if (operation == 'C')
            model.clear();
        view.setDisplay(model.getValue());
    }
}

```

上例中，我们在展现者中保持了一些状态信息。如果不使用这种方式的话，我们也可以在展现者与模型之间使用一个控制器来处理这些底层的按钮逻辑。

11.11. 管理 URI 片段

AJAX 应用程序中的一个主要问题是它们运行在单一的 Web 页面中，使用书签记录应用程序 URL（或者更一般地说 *URI*）只能记录下应用程序地址，而不是应用程序中的某一个状态。对很多应用程序来说这是一个问题，比如对于产品目录和论坛程序，如果能为某个特定的产品或某个特定的讨论信息记录下独立的地址将会更好一些。而且，由于浏览器使用 *URI* 来记录浏览履历，浏览履历和回退按钮都将无法正常工作。对这类问题的解决方法是使用 *URI* 中的 **片段标识符部分**，这个部分与 *URI* 的主部分（地址 + 路径 + 可选的查询参数）之间通过 # 号分隔开。比如：

```
http://example.com/path#myfragment
```

片段标识符部分的语法由 RFC 3986 标准定义（Internet 标准 STD 66），这个标准定义了整个 *URI* 的语法。片段可以只包含通常的 *URI* 路径字符（具体请参见上述标准），以及斜线和问号。

Vaadin 提供了两种方式来使用 *URI* 片段：一种是高阶的 **Navigator** 类，详情请参见第 11.9 节“应用程序内的导航跳转”，另一种是低阶 API，本节讨论这种低阶 API。

11.11.1. 设置 URI 片段

你可以在 **Page** 对象中使用 `setUriFragment()` 方法，设置当前片段标识符。

```
Page.getCurrent().setUriFragment("mars");
```

设置 *URI* 片段会导致一个 `UriFragmentChangeEvent` 事件，这个事件会在当前的服务器请求中处理。当前处理中的服务请求返回应答之后，通过 UI 的描绘处理，*URI* 片段会反映到浏览器中。

在片段标识符之前使用感叹号前缀，可以允许 Web 爬虫抓取，详情请参见第 11.11.4 节“支持 Web 爬虫”。

11.11.2. 读取 URI 片段

当前 *URI* 片段可以使用当前 **Page** 对象的 `getUriFragment()` 方法取得。当 UI 的 `init()` 方法被调用时，片段信息就是可知的了。

```
// Read initial URI fragment to create UI content
String fragment = getPage().getUriFragment();
enter(fragment);
```

为了在 *URI* 片段变更时重用代码，详情见下文，通常最好的办法是在一个独立的方法中构建 UI 中的相关部分。上面的例子中，我们调用了 `enter()` 方法，这种方式类似于使用 **Navigator** 来处理视图变更时的做法。

11.11.3. 监听 URI 片段的变更

UI 初始化完成后，*URI* 片段的变化可以使用 `UriFragmentChangeListener` 来处理。这个监听器会在 *URI* 片段发生变化时被调用，但在 UI 初始化时不会。如前文所述，UI 初始化时是可以通过 `page` 对象得到当前 *URI* 片段的。

比如，我们可以在 UI 类的 `init()` 方法中定义一个监听器，如下：

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().addUriFragmentChangedListener(
            new UriFragmentChangedListener() {
                public void uriFragmentChanged(
```

```

        UriFragmentChangedEvent source) {
    enter(source.getUriFragment());
}
});

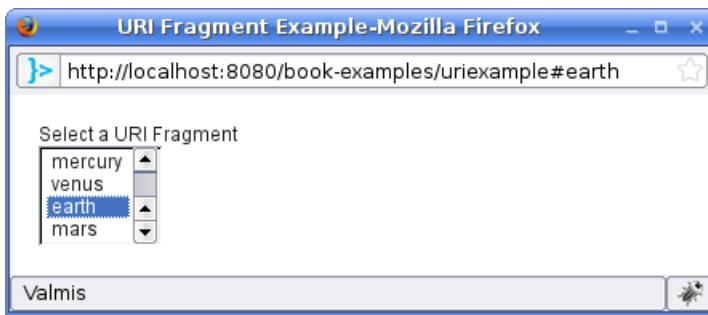
// Read the initial URI fragment
enter(getPage().getUriFragment());
}

void enter(String fragment) {
    ... initialize the UI ...
}
}
}

```

图 11.12 “使用 URI 片段来管理应用程序状态” 展示了一个应用程序，它可以使用 URI 片段来指定菜单的选中项，也可以在用户选择一个菜单项时设置对应的 URI 片段。

图 11.12. 使用 URI 片段来管理应用程序状态



11.11.4. 支持 Web 爬虫

有状态的 AJAX 应用程序通常不能被搜索引擎抓取，因为它们运行在单一页面中，而即使使用了 URI 片段，Web 爬虫也不能在应用程序的各个状态之间跳转。Google 搜索引擎和爬虫 支持一种规约，其中的片段标识符以感叹号为前缀，比如 `#!myfragment`。Servlet 需要有一个单独的可搜索的内容页面，这个页面使用相同的 URL 来访问，但带一个 `_escaped_fragment_` 参数。比如，对于 `/myapp/myui#!myfragment`，可搜索页面的 URL 应该是 `/myapp/myui?_escaped_fragment_=myfragment`。

你可以使用自定义的 Servlet 类，覆盖 `service()` 方法来提供可搜索的页面内容。对于通常的请求，你应该调用基类 **VaadinServlet** 的实现。

```

public class MyCustomServlet extends VaadinServlet
{
    @Override
    protected void service(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        String fragment = request
            .getParameter("_escaped_fragment_");
        if (fragment != null) {
            response.setContentType("text/html");
            Writer writer = response.getWriter();
            writer.append("<html><body>" +
                "<p>Here is some crawlable " +
                "content about " + fragment + "</p>");
        }
    }
}

```

```

        String items[] = {"mercury", "venus",
                           "earth", "mars"};
        writer.append("<p>Index of all content:</p><ul>");
        for (String item: items) {
            String url = request.getContextPath() +
                request.getServletPath() +
                request.getPathInfo() + "#!" + item;
            writer.append("<li><a href='" + url + "'>" +
                          item + "</a></li>");
        }
        writer.append("</ul></body>");
    } else
        super.service(request, response);
}
}

```

可被搜索引擎抓取的内容不需要是可供人类阅读的。它可以包含一些索引链接，指向应用程序的其他各种状态，如我们在上例中所作的那样。这些链接应该使用 "#!" 标注，但不应该是相对路径，这样才能避免在链接中出现 `_escaped_fragment_` 参数。

你应该在部署描述文件 `web.xml` 中使用这个自定义的 Servlet，而不是通常的 **VaadinServlet** 类，详情请参见第 4.8.4 节“使用部署描述文件 `web.xml`”。

11.12. 拖放

在一个对象上按下鼠标，并保持鼠标键按住不放，可以将这个对象从一个位置拖动到另一个位置，然后在目标位置上放开鼠标按钮，可以将这个对象“放”到另一个位置上。这种拖放操作是一种常用的方式，可用来移动、复制，或关联对象。比如，大多数操作系统允许在文件夹之间拖放文件，也支持拖动一个文档到程序上来打开这个文档。在 Vaadin 中支持组件的拖放，对于某些组件还可以拖放其中的一部分。

被拖动的对象，或者叫 可传送对象，本质上是数据对象。你可以在 **Table** 组件内拖放行，可以在 **Tree** 组件中拖放节点，这两种拖动都可以发生在组件之内，也可以在组件之间。你也可以将整个组件包在 **DragAndDropWrapper** 之内，然后就可以拖动这个组件。

拖动操作从 拖放源 开始，拖放源定义了可传送数据。可传送数据需要实现 **Transferable** 接口。对于绑定到 **Container** 数据源的 **Tree** 和 **Table**，对应于节点或者行的可传送数据，就是 Vaadin Data Model 中的一个 **Item**。被拖放的组件对应于一个 **WrapperTransferable** 对象。拖动开始操作不需要客户端到服务器的任何通信，你只需要允许拖动就可以了。拖和放的全部逻辑包括两个步骤：判定哪些位置可以允许放下拖动中的对象（属于称为 接受(*accepting*)），以及实际的放下动作。放的动作可以放在 拖放目标 中，拖放目标需要实现 **DropTarget** 接口。有三个组件实现了这个接口：**Tree**、**Table**，以及 **DragAndDropWrapper**。接受动作(*accepting*)和放下动作(*drop*)需要通过 *drop* 处理器来实现。本质上来说，对于拖放动作你所需要做的就是：在拖放源中允许拖动，并实现 **DropHandler** 接口中的 `getAcceptCriterion()` 和 `drop()` 方法。

Vaadin 的客户端/服务器端架构导致了拖放功能的一个问题。对一个拖动中的对象，它可以在哪些区域放下的判定处理，也就是，接受(*accepting*)一个放下动作，一般应该在客户端（也就是在浏览器内）完成。与服务器的通信太慢，因此这类判定无法在服务器端完成。因此，拖放功能提供了很多手段来避免发生服务器通信，以便提供更流畅的用户体验。

11.12.1. 处理拖放

大多数用户定义的拖放逻辑发生在 *drop* 处理器中，主要是实现 **DropHandler** 接口的 `drop()` 方法。另一个紧密相关的问题是 *drop* 动作的接受判定，这个处理定义在同一个接口的 `getAcceptCriterion()` 方法中。这个问题在后面的 第 11.12.4 节“接受拖放”中详细介绍。

`drop()` 方法收到的参数是 **DragAndDropEvent**. 通过这个事件对象可以得到两个重要的对象: **Transferable** 和 **TargetDetails**.

Transferable 中包含正在拖动中的对象(组件或数据项目)信息. Tree 或 Table 中的项目分别表现为 **TreeTransferable** 或 **TableTransferable** 对象, 其中包含正在被拖动中的 Tree 或 Table 项目的 ID. 这些特定的可传送对象是 **DataBoundTransferable** 类, 它绑定到容器中的某些数据. 被拖动的组件表现为 **WrapperTransferable** 对象, 叫这个名字是因为组件被包装在 **DragAndDropWrapper** 之内.

TargetDetails 对象中的信息是可传输对象被放下的确切位置. Detail 对象的具体类型由 drop 目标决定, 你需要将它转换为正确的类型才可以得到更多详细信息. 如果目标是一个选择组件, 也就是 Tree 或 Table, **AbstractSelectTargetDetails** 对象可以告诉我们推动对象被放在哪个数据项目之上. 对于 Tree, **TreeTargetDetails** 还给出了更多的信息. 对于被封装的组件, 这些信息由 **WrapperDropDetails** 对象给出. 除拖放的目标项目或目标组件外, Detail 对象还提供 drop 位置信息. 对于选择组件, 位置可以通过 `getDropLocation()` 方法得到, 对于封装的组件, 可以通过 `verticalDropLocation()` 和 `horizontalDropLocation()` 方法得到. 位置信息表现为 **VerticalDropLocation** 或 **HorizontalDropLocation** 对象. drop 位置对象还指明可传输对象被放在组件或数据项目的上方, 下方, 还是正好在其内部.

关于将对象拖放到 Tree, Table, 和封装组件之上的问题, 将在后面的小节中讨论.

11.12.2. 拖放项目到 Tree 上

你可以从 Tree 中拖动数据项目, 也可以拖动到 Tree 上, 也可以在 Tree 内部拖动. 将 Tree 设置为拖动源只需要简单的使用 `setDragMode()` 方法设置拖动模式即可. Tree 目前只支持一种拖动模式, `TreeDragMode.NODE`, 这种模式允许拖动单个的节点. 拖动时, 被拖动的节点表现为 **TreeTransferable** 对象, 这个类是 **DataBoundTransferable** 的子类. Tree 的节点通过容器中数据项目的 ID 来标识.

当可传送对象拖放到 Tree 上时, drop 位置保存在 **TreeTargetDetails** 对象内, 这个对象标识了 drop 动作发生位置的节点所对应的数据项目 ID. 你可以使用 **AbstractSelectTargetDetails** 类的 `getItemIdOver()` 方法得到项目 ID, **TreeTargetDetails** 继承自这个类. drop 动作可以发生在节点上, 也可以在它的上方或下方; 确切的位置信息是一个 **VerticalDropLocation** 对象, 你可以通过 `getDropLocation()` 方法得到这个对象.

下面的例子中, 我们有一个 Tree 组件, 我们允许用户通过拖放操作来重新排列其中节点的顺序.

```
final Tree tree = new Tree("Inventory");
tree.setContainerDataSource(TreeExample.createTreeContent());
layout.addComponent(tree);

// Expand all items
for (Iterator<?> it = tree.rootItemIds().iterator(); it.hasNext();)
    tree.expandItemsRecursively(it.next());

// Set the tree in drag source mode
tree.setDragMode(TreeDragMode.NODE);

// Allow the tree to receive drag drops and handle them
tree.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        // Wrapper for the object that is dragged
    }
});
```

```

Transferable t = event.getTransferable();

// Make sure the drag source is the same tree
if (t.getSourceComponent() != tree)
    return;

TreeTargetDetails target = (TreeTargetDetails)
    event.getTargetDetails();

// Get ids of the dragged item and the target item
Object sourceItemId = t.getData("itemId");
Object targetItemId = target.getItemIdOver();

// On which side of the target the item was dropped
VerticalDropLocation location = target.getDropLocation();

HierarchicalContainer container = (HierarchicalContainer)
tree.getContainerDataSource();

// Drop right on an item -> make it a child
if (location == VerticalDropLocation.MIDDLE)
    tree.setParent(sourceItemId, targetItemId);

// Drop at the top of a subtree -> make it previous
else if (location == VerticalDropLocation.TOP) {
    Object parentId = container.getParent(targetItemId);
    container.setParent(sourceItemId, parentId);
    container.moveAfterSibling(sourceItemId, targetItemId);
    container.moveAfterSibling(targetItemId, sourceItemId);
}

// Drop below another item -> make it next
else if (location == VerticalDropLocation.BOTTOM) {
    Object parentId = container.getParent(targetItemId);
    container.setParent(sourceItemId, parentId);
    container.moveAfterSibling(sourceItemId, targetItemId);
}
}
);
});

```

对 **Tree** 进行拖放操作的接受判定

Tree 定义了一些特殊的接受判定结果.

TargetInSubtree (客户端)

如果目标项目是某个特定的子树下的一个节点, 接受拖放. 子树通过它的根节点对应的项目 ID 来指定, 这个项目 ID 在构造函数中指定. 构造函数的另一个版本还包括一个深度参数, 从根节点向下固定深度之内的节点接受拖放. -1 代表无限深度, 也就是整个子树都可以接受拖放, 因此效果与第一个构造函数一样.

TargetItemAllowsChildren (客户端)

如果 **Tree** 对目标数据项目设置了 `setChildrenAllowed()`, 则接受拖放动作. 这个判定不需要参数, 因此这个类可以是一个单例(singleton), 并且可以通过 `Tree.TargetItemAllowsChildren.get()` 方法取得. 比如, 下例中的复合判定, 允许拖动到各节点之前或之后, 对于拖动到节点之下成为子节点的情况, 则要求对象节点必须允许子节点:

```
return new Or (Tree.TargetItemAllowsChildren.get(), new
Not(VerticalLocationIs.MIDDLE));
```

TreeDropCriterion (服务器端)

只允许拖动到一部分节点上, 这些节点通过一组项目 ID 来指定. 你必须扩展这个抽象类, 并实现 `getAllowedItemIds()` 方法来返回这个项目 ID 集合. 由于这个判定发生在服务器端, 它是延迟加载的, 因此对于拖动操作来说, 可接受的目标节点只会从服务器端加载一次. 示例请参见 第 11.12.4 节 “接受拖放”.

此外, **AbstractSelect** 中定义的接受判定类也可以用于 **Tree**, 详情请参见 第 11.12.4 节 “接受拖放”.

11.12.3. 拖放项目到 **Table** 上

你可以从 **Table** 中拖放一个数据项目, 也可以拖放到 **Table** 上, 也可以在 **Table** 内部拖放. 将 **Table** 设置为拖放源只需要简单地使用 `setDragMode()` 方法设置拖动模式即可. **Table** 可以使用 `TableDragMode.ROW` 模式拖动单行, 也可以使用 `TableDragMode.MULTIROW` 模式拖动多行. 拖动时, 被拖动的行表现为 **TreeTransferable** 对象, 它继承自 **DataBoundTransferable**. 行通过容器中数据项目的 ID 来标识.(译注: 此处原文有误)

当一个可传输对象被拖放到 **Table** 上时, `drop` 位置保存在 **AbstractSelectTargetDetails** 对象中, 它使用数据项目的 ID 来标识拖放的目标行. 你可以通过 `getItemIdOver()` 方法得到数据项目. `drop` 动作可以发生在行上, 也可以在行的上方或下方; 确切的位置是 **VerticalDropLocation** 对象, 可以通过 `Detail` 对象的 `getDropLocation()` 得到这个对象.

对 **Table** 进行拖放操作的接受判定

Table 定义了一些特殊的接受判定结果.

TableDropCriterion (服务器端)

只允许拖放到一部分项目上, 这些项目通过一组项目 ID 来指定. 你必须扩展这个抽象类, 实现 `getAllowedItemIds()` 方法来返回项目 ID 集合. 由于这个判定发生在服务器端, 它是延迟加载的, 因此对于拖动操作来说, 可接受的目标项目只会从服务器端加载一次.

11.12.4. 接受拖放

你不能将拖动中的对象放在任意的位置. 在你放下对象之前, 鼠标目前移动到的位置必须被判定为接受拖放. 当鼠标抓住一个拖动中的对象经过一个接受拖放的位置时, 会出现 接受拖放指示器, 用户可以通过这个指示器来正确地调整放下对象的位置. 由于这种检查必须在鼠标在拖放目标附近移动时反复执行, 因此不适合向服务器发送请求, 因此 接受判定通常是在客户端进行的.

`drop` 处理器必须定义接受判定条件(`AcceptCriterion`), 它负责判定拖动中的对象可以放在哪些对象上. 判定条件需要通过 **DropHandler** 接口的 `getAcceptCriterion()` 方法给出. 判定条件表现为一个 **AcceptCriterion** 对象, 它也可以是多个判定条件通过逻辑运算的组合. 有两种基本的判定条件: 客户端判定条件与 服务器端判定条件. Vaadin 有很多内建的判定条件, 可以实现基于源组件和目标组件是否一致的接受判定, 以及基于被拖动对象 `data flavor` 的接受判定.

要支持任意的可传送数据的拖放, 你可以返回一个接受一切数据的判定条件, 可以使用 `AcceptAll.get()` 方法得到这样的判定条件.

```
tree.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }
});
```

```

}
...

```

客户端判定条件

客户端判定条件，继承自 **ClientSideCriterion**，运行在客户端，在鼠标移动时会不断判定各组件能否接受目前拖动中的对象，但执行判定时不必发送服务器请求。

com.vaadin.event.dd.acceptcriterion 包内定义了以下客户端判定条件：

AcceptAll

对所有可传输数据和所有目标对象都接受拖放。

And

在两个或更多客户端判定条件上执行逻辑 AND 操作；如果所有的子条件都接受，那么这个复合条件就接受拖放。

ContainsDataFlavour

可传输数据必须包含指定的 data flavour。

Not

在一个客户端判定条件上执行一个逻辑 NOT 操作；只有在子条件不接受时，这个条件才接受拖放。

Or

在两个或更多客户端判定条件上执行逻辑 OR 操作；如果任意一个子条件接受，那么这个复合条件就接受拖放。

Sources

如果拖动中的可传输对象来自指定的源组件之一，那么就接受它。

SourcesTarget

当拖动数据的源组件与目标组件一致时，接受拖放。这个判定条件可用于确保数据项目只在同一个 Tree 或 Table 内部拖动，而不是来自外部。

TargetDetails

当 target detail 对象，比如 Tree 节点或 Table 行的数据项目，是某种指定的 data flavor，并且是某个指定的值时，接受拖放操作。

此外，目标组件，比如 **Tree** 和 **Table**，还定义了一些组件专有的客户端判定条件。详情请参见第 11.12.2 节“拖放项目到 **Tree** 上”。

AbstractSelect 为所有的选择组件，包括 **Tree** 和 **Table**，定义了以下判定条件。

AcceptItem

只接受某个特定的选择组件中的特定数据项目。选择组件继承自 **AbstractSelect**，作为判定条件构造方法的第一个参数指定。第二个参数是拖放源中可接受的项目 ID 列表。

AcceptItem.ALL

只要拖动中的可传输对象是数据项目，就接受拖放。

TargetItems

对指定的目标项目，接受它之上的所有拖放操作。这个判定条件的构造方法参数是目标组件(**AbstractSelect**)，以及允许拖放的目标项目 ID 列表。

VerticalLocationIs.MIDDLE, TOP, 和 BOTTOM

这三个静态的判定条件接受在一个数据项目之上, 上方, 或下方的拖放操作. 比如, 你可以使用以下代码, 只接受数据项目之间位置上的拖放:

```
public AcceptCriterion getAcceptCriterion() {
    return new Not(VerticalLocationIs.MIDDLE);
}
```

服务器端判定条件

服务器端判定条件运行在服务器端, 通过 **ServerSideCriterion** 类的 `accept()` 方法进行判定. 这种方式可以使用程序的全部逻辑功能来实现拖放接受的判定, 但缺点是它导致大量的服务器请求. 每次鼠标到达一个位置都需要发起一次请求. 很多情况下可以使用组件专有的延迟装载判定条件 **TableDropCriterion** 和 **TreeDropCriterion** 来减轻这个问题. 这些判定条件对每个拖放操作只执行一次服务器请求, 并针对目前拖动中的 **Transferable** 对象返回所有可接受的行或节点.

`accept()` 方法接受的参数是拖动事件, 因此这个方法的实现可以类似于 `drop()` 方法.

```
public AcceptCriterion getAcceptCriterion() {
    // Server-side accept criterion that allows drops on any other
    // location except on nodes that may not have children
    ServerSideCriterion criterion = new ServerSideCriterion() {
        public boolean accept(DragAndDropEvent dragEvent) {
            TreeTargetDetails target = (TreeTargetDetails)
                dragEvent.getTargetDetails();

            // The tree item on which the load hovers
            Object targetItemId = target.getItemIdOver();

            // On which side of the target the item is hovered
            VerticalDropLocation location = target.getDropLocation();
            if (location == VerticalDropLocation.MIDDLE)
                if (!tree.areChildrenAllowed(targetItemId))
                    return false; // Not accepted

            return true; // Accept everything else
        }
    };
    return criterion;
}
```

服务器端判定条件的基类 **ServerSideCriterion** 提供了一个共通的 `accept()` 方法. 更细化的 **TableDropCriterion** 和 **TreeDropCriterion** 类是便于使用的子类, 可以以数据项目集合的形式来指定允许拖放的目标项目. 这些类还提供了延迟加载的优化, 可以显著降低与服务器端的通信数量.

```
public AcceptCriterion getAcceptCriterion() {
    // Server-side accept criterion that allows drops on any
    // other tree node except on node that may not have children
    TreeDropCriterion criterion = new TreeDropCriterion() {
        @Override
        protected Set<Object> getAllowedItemIds(
            DragAndDropEvent dragEvent, Tree tree) {
            HashSet<Object> allowed = new HashSet<Object>();
            for (Iterator<Object> i =
                tree.getItemIds().iterator(); i.hasNext(); ) {
                Object itemId = i.next();
                if (tree.hasChildren(itemId))

```

```

        allowed.add(itemId);
    }
    return allowed;
}
};

return criterion;
}
}

```

接受指示器

当一个被拖动的对象经过 drop 目标上方时, 会显示 接受指示器, 告诉使用者这个位置是否允许拖放. 对于 MIDDLE 位置, 指示器是目标周边的一个方框 (Tree 的节点, Table 的行, 或组件). 对于垂直 drop 位置, 可接受的位置显示为水平线条, 对于水平 drop 位置, 显示为垂直线条.

对于拖放目标为 **DragAndDropWrapper** 的情况, 你可以禁止显示接受指示器和 拖放提示(*drag hint*), 方法是使用 *no-vertical-drag-hints*, *no-horizontal-drag-hints*, 和 *no-box-drag-hints* 样式. 你需要将这些样式添加到 包含 wrapper 对象的布局组件上, 而不是添加给 wrapper 对象自身.

```

// Have a wrapper
DragAndDropWrapper wrapper = new DragAndDropWrapper(c);
layout.addComponent(wrapper);

// Disable the hints
layout.addStyleName("no-vertical-drag-hints");
layout.addStyleName("no-horizontal-drag-hints");
layout.addStyleName("no-box-drag-hints");

```

11.12.5. 拖动组件

拖动组件需要将源组件封装在 **DragAndDropWrapper** 之内. 然后使用 `setDragStartMode()` 方法将 wrapper(以及组件)设置为拖动模式, 就可以实现拖动功能了. 这个方法支持两种拖动模式: `DragStartMode.WRAPPER` 和 `DragStartMode.COMPONENT`, 这两个模式决定拖动时显示的拖动图像是整个 wrapper, 还是仅仅只是被封装的组件.

```

// Have a component to drag
final Button button = new Button("An Absolute Button");

// Put the component in a D&D wrapper and allow dragging it
final DragAndDropWrapper buttonWrap = new DragAndDropWrapper(button);
buttonWrap.setDragStartMode(DragStartMode.COMPONENT);

// Set the wrapper to wrap tightly around the component
buttonWrap.setSizeUndefined();

// Add the wrapper, not the component, to the layout
layout.addComponent(buttonWrap, "left: 50px; top: 50px;");

```

DragAndDropWrapper 的高度默认为未指定, 但宽度默认为 100%. 如果你希望确保 wrapper 的尺寸紧密贴近于被封装的组件, 那么你应该对 wrapper 调用 `setSizeUndefined()` 方法. 这时, 你应该确保被封装的组件尺寸不是一个相对值, 否则会导致矛盾.

被拖动的组件使用 **WrapperTransferable** 类来表示. 你可以使用 `getDraggedComponent()` 方法得到被拖动的组件. 如果拖动中的可传输对象不是组件, 这个方法将返回 null. HTML5 拖动(详情见后述)也使用 WrapperTransferable 类来表示.

11.12.6. 拖动到组件上

将组件封装在 **DragAndDropWrapper** 之内, 也可以实现向组件的 drop. wrapper 也是一种普通组件; 它的构造函数参数是被封装的组件. 你只需要在 wrapper 上使用 `setDropHandler()` 方法来定义 **DropHandler** 即可.

下例中, 我们允许在绝对布局中移动组件. drop 处理器的细节将在后文中给出.

```
// A layout that allows moving its contained components
// by dragging and dropping them
final AbsoluteLayout absLayout = new AbsoluteLayout();
absLayout.setWidth("100%");
absLayout.setHeight("400px");

... put some (wrapped) components in the layout ...

// Wrap the layout to allow handling drops
DragAndDropWrapper layoutWrapper =
    new DragAndDropWrapper(absLayout);

// Handle moving components within the AbsoluteLayout
layoutWrapper.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        ...
    }
});
```

被封装组件的 Target Detail

drop 处理器收到一个 **WrapperTargetDetails** 对象, 这个对象实现了 **TargetDetails** 接口, 对象中包含的是 drop 目标的详细信息.

```
public void drop(DragAndDropEvent event) {
    WrapperTransferable t =
        (WrapperTransferable) event.getTransferable();
    WrapperTargetDetails details =
        (WrapperTargetDetails) event.getTargetDetails();
```

WrapperTargetDetails 中包含一个 **MouseEventDetails** 对象, 你可以通过 `getMouseEvent()` 方法得到它. 你可以通过这个对象得到鼠标按钮放开、拖动结束时的坐标位置. 类似的, 你可以通过可传输对象(如果是 **WrapperTransferable** 对象)使用 `getMouseDownEvent()` 方法找出拖动的开始位置.

```
// Calculate the drag coordinate difference
int xChange = details.getMouseEvent().getClientX()
    - t.getMouseDownEvent().getClientX();
int yChange = details.getMouseEvent(). getClientY()
    - t.getMouseDownEvent().getClientY();

// Move the component in the absolute layout
ComponentPosition pos =
    absLayout.getPosition(t.getSourceComponent());
pos.setLeftValue(pos.getLeftValue() + xChange);
pos.setTopValue(pos.getTopValue() + yChange);
```

你可以使用 `getAbsoluteLeft()` 和 `getAbsoluteTop()` 方法得到目标 wrapper 的 x 和 y 坐标绝对值, 这两个方法可以将鼠标位置的绝对坐标翻译为相对于 wrapper 对象的坐标. 注意, 坐标是相对于 wrapper 的, 而不是相对于被封装的组件的; wrapper 会保留一些空间, 用于显示接受指示器.

`verticalDropLocation()` 和 `horizontalDropLocation()` 方法可以得到目标对象上的更详细的 drop 位置.

11.12.7. 从浏览器之外拖放文件

DragAndDropWrapper 允许从浏览器之外拖动文件, 并 drop 到 wrapper 内封装的组件上. 被拖放的文件会自动上传到应用程序中, 然后可以通过 wrapper 的 `getFiles()` 方法得到. 文件以 **Html5File** 对象的形式表达, 这个类定义在 `inner` 类中. 你可以定义一个上传 **Receiver** 来接受文件内容, 并输出到一个 **OutputStream** 中.

拖放文件到浏览器中是 HTML 5 支持的功能, 因此需要兼容 HTML 5 标准的浏览器, 比如 Mozilla Firefox 3.6 或更高版本.

11.13. 日志

你可以在 Vaadin 应用程序中使用标准的 `java.util.logging` 功能来输出日志. 配置日志很容易, 只需要将名为 `logging.properties` 的配置文件放在你的 Vaadin 应用程序的默认包之下即可,(也就是 Eclipse 工程的 `src` 目录, Maven 工程的 `src/main/java` 或 `src/main/resources` 目录). 这个文件由 **Logger** 类的新实例创建时负责读取 .

在 Apache Tomcat 中输出日志

当 Vaadin 应用程序部署到 Apache Tomcat 中时, 你不需要做任何特殊的控制, 就可以将日志输出到 Tomcat 自身日志相同的位置. 如果你需要将 Vaadin 应用程序的日志输出到其他位置, 只需要在你的 Vaadin 应用程序默认包下添加一个自定义的 `logging.properties` 设置文件.

如果你希望通过其他日志工具来输出日志, 请参见下文的“使用 SLF4J 将日志转发给 Log4j”一节.

在 Liferay 中输出日志

Liferay 默认关闭了通过 `java.util.logging` 输出的日志. 为了打开日志, 你需要在你的 Vaadin 应用程序的默认包之下增加一个你自己的 `logging.properties` 配置文件. 这个文件应该定义至少一个日志信息的输出位置.

你也可以通过 SLF4J 来输出日志, 这个工具也被 Liferay 使用, 并且绑定在 Liferay 中. 详情请参见“使用 SLF4J 将日志转发给 Log4j”一节.

使用 SLF4J 将日志转发给 Log4j

使用 SLF4J (<http://slf4j.org/>) 可以很容易地将 `java.util.logging` 的日志转发给 Log4j. 基本方法是添加 SLF4J JAR 文件, 以及 `jul-to-slf4j.jar` 文件, 这个文件中实现了从 `java.util.logging` 到 SLF4J 的桥接功能. 你还需要添加第三方日志实现的 JAR 文件, 也就是, `slf4j-log4j12-x.x.x.jar`, 以便使用 Log4j 来输出日志. 关于这个问题的详情, 请参见 SLF4J 网站.

为了安装从 `java.util.logging` 到 SLF4J 的桥接, 你需要将以下代码段添加到你的 **UI** 类的最前面:

```
static {
    SLF4JBridgeHandler.install();
}
```

在 Vaadin 开始进行任何日志输出之前, 这段代码将确保桥接功能已安装并正常工作.



注意!

这种方法会产生严重的性能问题, 当日志输出禁用时, 日志语句的执行代价增加 60 倍, 当日志输出启用时, 执行代价增加 20%. 但是, Vaadin 输出的日志并不多, 因此性能影响很小, 几乎可以忽略.

使用日志输出器

你可以使用一种简单的模式来输出日志, 这种模式中你可以在每个需要输出日志的类中注册一个静态的日志输出器实例, 然后在类中需要输出日志的地方使用这个日志输出器. 如下例:

```
public class MyClass {
    private final static Logger logger =
        Logger.getLogger(MyClass.class.getName());

    public void myMethod() {
        try {
            // do something that might fail
        } catch (Exception e) {
            logger.log(Level.SEVERE, "FAILED CATASTROPHICALLY!", e);
        }
    }
}
```

为每个需要输出日志的类保持一个 `Logger` 日志输出器实例, 与在每个需要输出日志的类的实例中保持一个日志输出器相比, 使用静态的日志输出器可以节约极少量的内存和时间. 但是, 在某些应用程序服务器上部署应用程序时, 这种方法有可能会导致应用程序泄漏 PermGen 内存. 这个问题的原因是 `Logger` 类可能会保持它的实例的硬参照. 由于负责装载 `Logger` 类的类装载器在多个不同的 Web 应用程序之间共用, 因此由各应用程序类装载器装载的类的引用, 可能在应用程序重新部署之后阻止对这些类的垃圾收集, 因此发生内存泄漏(译注: 这段理解不能, 待校). 由于保持类对象的 PermGen 内存的尺寸是固定的, 因此在多次重新部署应用程序之后, 这个内存泄漏会导致服务器崩溃. 这个问题取决于服务器如何管理它的类装载器, 也取决于反向引用的坚固度(译注: 这段理解不能, 待校), 而且可能在 Java 6 和 7 之间也会不同. 因此, 如果你遇到 PermGen 问题, 或者希望你的程序更安全一些, 你应该考虑使用非静态的 `Logger` 实例.

11.14. 与 JavaScript 集成

Vaadin 支持双向的 JavaScript 调用, 既可以从服务器端调用 JavaScript, 也可以从 JavaScript 调用服务器端. 使用这种功能就可以与 JavaScript 代码交互, 而不必编写客户端代码.

11.14.1. 调用 JavaScript

你可以在服务端使用 `JavaScript` 类的 `execute()` 方法来执行对 JavaScript 的调用. 首先通过当前 `Page` 对象的 `getJavaScript()` 方法得到 `JavaScript` 实例.

```
// Execute JavaScript in the currently processed page
Page.getCurrent().getJavaScript().execute("alert('Hello')");
```

`JavaScript` 类本身有一个静态的便捷方法 `getCurrent()`, 可以从当前处理中的页面得到 `JavaScript` 实例.

```
// Shorthand
JavaScript.getCurrent().execute("alert('Hello')");
```

当前处理中的服务器请求返回之后, JavaScript 就会被执行。如果一次请求中多次调用了 JavaScript, 这些调用会在请求结束后顺序执行。因此, JavaScript 调用不会导致应用程序服务器端的运行暂停, 而且你不能在这种 JavaScript 中返回值。

11.14.2. 处理 JavaScript 函数的回调

你也可以从客户端的 JavaScript 来调用服务器端。这个功能要求你在服务器端注册 JavaScript 回调方法。你需要实现一个 **JavaScriptFunction**, 并使用当前 **JavaScript** 对象的 `addFunction()` 方法来注册这个函数。函数需要有名称, 包路径可选, 这些参数需要传递给 `addFunction()` 方法。你只需要实现 `call()` 方法, 就可以处理来自客户端 JavaScript 的调用。

```
JavaScript.getCurrent().addFunction("com.example.foo.myfunc",
    new JavaScriptFunction() {
        @Override
        public void call(JSONArray arguments) throws JSONException {
            Notification.show("Received call");
        }
    });

Link link = new Link("Send Message", new ExternalResource(
    "javascript:com.example.foo.myfunc()"));

客户端传递给 JavaScript 方法的参数, 会以 JSONArray 的形式传递给服务器端的 call() 方法。参数值可以使用 get() 方法取得, 这个方法的参数是希望取得的 JavaScript 参数下标, 也可以使用带类型转换的 get 方法。这些 get 方法必须与实际传递的参数的类型一致, 否则会抛出 JSONException 例外。
```

```
JavaScript.getCurrent().addFunction("com.example.foo.myfunc",
    new JavaScriptFunction() {
        @Override
        public void call(JSONArray arguments) throws JSONException {
            try {
                String message = arguments.getString(0);
                int value = arguments.getInt(1);
                Notification.show("Message: " + message +
                    ", value: " + value);
            } catch (JSONException e) {
                Notification.show("Error: " + e.getMessage());
            }
        }
    });

Link link = new Link("Send Message", new ExternalResource(
    "javascript:com.example.foo.myfunc(prompt('Message'), 42)"));
```

这里的函数回调机制, 与 JavaScript 组件集成中使用的 RPC 机制是一样的, 详情请参见第 16.13.4 节“从 JavaScript 到服务器端的 RPC 调用”。

11.15. 访问 Session 全局数据

本节大部分内容已按照 Vaadin 7 进行了更新, 但少量信息还需要修改。

应用程序通常会需要使用一些全局对象, 比如用户信息, 业务数据模型, 或数据库连接。创建并管理这些数据的通常是应用程序的 UI 类, 或者是 Session 或 Servlet。

比如, 你的 UI 类可以是这样:

```

class MyUI extends UI {
    UserData userData;

    public void init() {
        userData = new UserData();
    }

    public UserData getUserData() {
        return userData;
    }
}

```

Vaadin 提供了两种方式访问 UI 对象: 通过任何组件, 使用它的 `getUI()` 方法, 以及使用全局方法 `UI.getCurrent()`.

`getUI()` 方法的用法如下:

```
data = ((MyUI) component.getUI()).getUserData();
```

但是这种方式在大多数情况下是无用的, 因为它需要组件绑定到 UI 上. 在 UI 的创建过程中, 比如在构造方法中, 就无法满足这个要求了.

```

class MyComponent extends CustomComponent {
    public MyComponent() {
        // This fails with NullPointerException
        Label label = new Label("Country: " +
            getApplication().getLocale().getCountry());

        setCompositionRoot(label);
    }
}

```

通过取得当前 Servlet, Session, UI 的全局方法, 可以十分便利地得到数据:

```
data = ((MyUI) UI.getCurrent()).getUserData();
```

问题

访问 Session 全局数据的基本问题是 `getUI()` 方法只能在组件绑定到应用程序之后才可以工作. 在此之前, 这个方法只会返回 `null`. 在组件的构造方法中就会如此, 比如在 **CustomComponent** 的构造方法中:

使用静态变量, 或使用单子实现来访问用户 Session 数据是不可能的, 因为静态变量是在整个 Web 应用程序范围内唯一的, 而不仅仅是在当前用户 Session 中唯一. 这种方案在并发的多个 session 之间共享数据是有用的, 但为了在 session 内共享数据就不可行了.

这类数据会被所有的用户共享, 而且每次新用户打开应用程序时都会重新初始化这些数据.

解决方案概要

为了取得 `application` 对象或其他全局数据, 有以下几种解决方案:

- 将全局数据的引用作为参数传递
- 在 `attach()` 方法内初始化组件
- (如果使用视图导航的话)在视图导航跳转的 `enter()` 方法内初始化组件

- 使用 *ThreadLocal* 模式保存全局数据

上述各种解决方案会在以下各节分别解释.

11.15.1. 传递对象引用

你可以将对象的引用作为参数来传递. 这是面向对象编程中的常见方式.

```
class MyApplication extends Application {
    UserData userData;

    public void init() {
        Window mainWindow = new Window("My Window");
        setMainWindow(mainWindow);

        userData = new UserData();
        mainWindow.addComponent(new MyComponent(this));
    }

    public UserData getUserData() {
        return userData;
    }
}

class MyComponent extends CustomComponent {
    public MyComponent(MyApplication app) {
        Label label = new Label("Name: " +
            app.getUserData().getName());

        setCompositionRoot(label);
    }
}
```

如果在其他方法中需要这些引用, 那么你必须再次把它作为参数传递过去, 或者保存在成员变量中以便其他方法访问.

这个方案的问题是它会导致应用程序中所有构造方法都需要一个 application 参数, 而且在这些类内部还需要将 application 参数继续传递给需要它的方法, 这也是一个很麻烦的任务.

11.15.2. 覆盖 attach() 方法

当组件绑定到应用程序的组件包含层级关系中时, 会调用 attach() 方法. 这时 getApplication() 方法是可以使用的.

```
class MyComponent extends CustomComponent {
    public MyComponent() {
        // Must set a dummy root in constructor
        setCompositionRoot(new Label(""));
    }

    @Override
    public void attach() {
        Label label = new Label("Name: " +
            ((MyApplication) component.getApplication())
                .getUserData().getName());

        setCompositionRoot(label);
    }
}
```

```

    }
}

```

这种方案虽然可以工作，但略为混乱。你有可能会需要在构造方法中进行一些初始化工作，但如果需要使用全局数据的话，又必须在 `attach()` 方法内执行。尤其是，**CustomComponent** 需要在构造方法中调用 `setCompositionRoot()` 方法。如果你在构造方法中无法创建实际的根组件，你就必须使用一个临时性的 dummy 根组件，如上例所示。

如果你希望访问你的 `application` 类中定义的方法，那么使用 `getApplication()` 得到的 `application` 对象还需要进行类型转换。

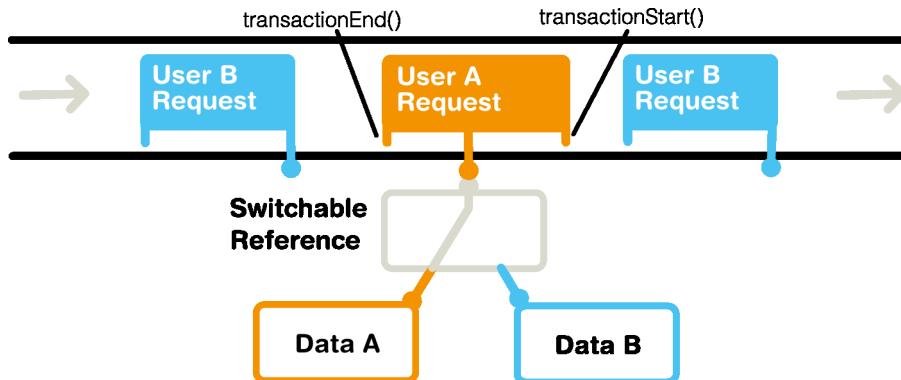
11.15.3. ThreadLocal 模式

Vaadin 使用 ThreadLocal 模式来实现当前服务器请求的 **UI** 和 **Page** 对象的全局访问，通过各类的 `getCurrent()` 静态方法就可以得到这些对象。本节介绍 Vaadin 为什么使用这种模式，以及它的工作原理。为了达到某种特定的目的，你可能会需要重新实现这种模式。

ThreadLocal 模式为了解决全局数据的访问问题，将其分解为与静态变量相关的两个子问题来分别解决。

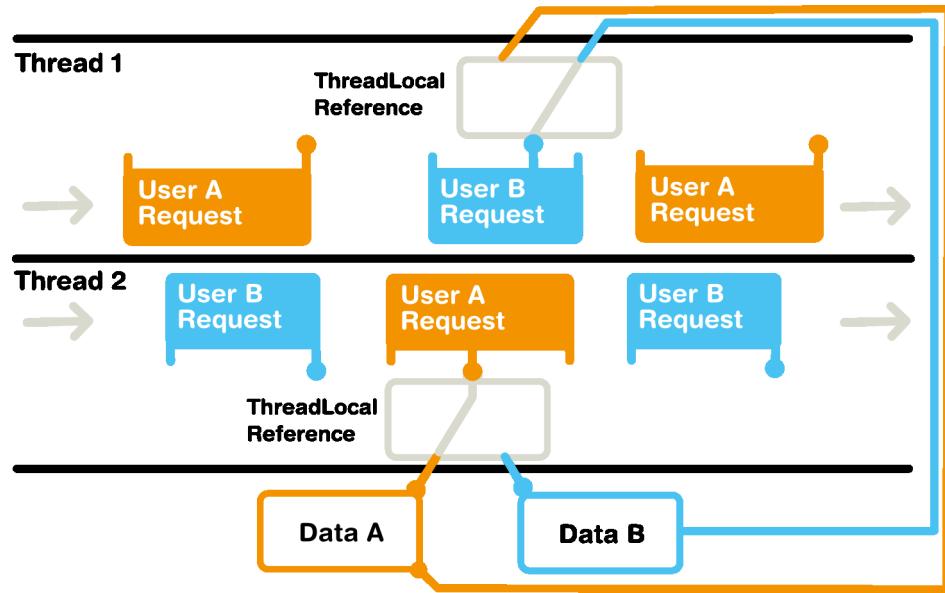
第一个子问题是，假定 Servlet 容器对多个用户(session)的请求顺序地处理。如果在属于某个用户的一个请求之内设置了静态变量，这个变量可能会被属于另一个用户的下一个请求读取，或者重新设置。解决这个问题的方法是，在每个 HTTP 请求的开始处，将全局引用设置为指向当期用户的数

图 11.13. 在顺序处理的多个请求中切换静态(或者 **ThreadLocal**)的对象引用



第二个子问题是 Servlet 容器通常会使用线程池中的多个工作线程来并发地处理多个请求。因此，设置静态引用，会使得它在并发执行的所有线程之内都被改变，此时可能恰好另一个线程正在处理属于另一个用户的请求。这个问题的解决方案是使用 `thread-local` 变量来保存数据引用，而不是使用静态变量。因此你可以使用 Java 的 **ThreadLocal** 类来切换数据引用。

图 11.14. 在并发处理多个请求时切换 ThreadLocal 引用



11.16. 服务器端 PUSH

如果你需要从一个 UI 来更新另一个 UI(可能属于另一个用户), 或者通过服务器端运行的背景线程来更新 UI, 你通常会希望这个更新动作立即发生, 而不是一直等到浏览器(不知何年何月.....)发起下一次请求时才有机会完成更新. 为了达到这个目的, 你可以使用 服务器端PUSH 功能, 即时发送数据到浏览器端. PUSH 的基础是客户端和服务器端之间的连接, 通常使用 WebSocket 连接, 由客户端负责创建这个连接, 服务器端可以使用这个连接将更新信息发送给客户端.

服务器端与客户端的通信默认使用 WebSocket 连接, 但前提是浏览器和服务器都要支持 WebSocket. 如果不支持, Vaadin 会使用浏览器能够支持的替代方案. Vaadin PUSH 功能的客户端与服务器端通信, 使用一个定制编译版的 Atmosphere framework.

11.16.1. 安装 PUSH 功能

Vaadin 所支持的服务器端 PUSH 需要单独的 Vaadin PUSH 库. 这个库包含在安装包中, 文件名是 vaadin-push.jar.

通过 Ivy 取得 PUSH 库

使用 Ivy 时, 你可以使用 ivy.xml 中的以下声明来得到 PUSH 库:

```
<dependency org="com.vaadin" name="vaadin-push"
           rev="&vaadin.version;" conf="default->default"/>
```

在某些服务器中, 你可能需要除去对 slf4j 的依赖, 如下:

```
<dependency org="com.vaadin" name="vaadin-push"
           rev="&vaadin.version;" conf="default->default">
    <exclude org="org.slf4j" name="slf4j-api"/>
</dependency>
```

注意, Atmosphere 库是以 bundle 形式提供的, 所以如果你使用 Ant, 你需要以 type="jar,bundle" 的方式来取得这个库.

使用 Maven 取得 PUSH 库

在 Maven 中, 你可以在 POM 中使用以下依赖关系来得到 PUSH 库:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-push</artifactId>
    <version>${vaadin.version}</version>
</dependency>
```

11.16.2. 对一个 UI 允许 PUSH 功能

要激活服务器端 PUSH 功能, 你需要定义 PUSH 模式, 可以使用描述符来定义, 也可以在 UI 类上使用 **@Push** 注解.

PUSH 模式

服务器端 PUSH 的模式有两种: **□□** 模式和 **manual** 模式. 自动模式会在 `access()` 结束后将更新内容自动 PUSH 到浏览器端. 手动模式下, 你可以使用 `push()` 方法显式地进行 PUSH, 这种方法的灵活性更高一些.

@Push 注解

如下例所示, 你可以使用 **@Push**注解, 对一个 UI 激活服务器端 PUSH功能. 这个注解默认是自动模式(`PushMode.AUTOMATIC`).

```
@Push
public class PushyUI extends UI {
```

要使用手动模式, 你需要使用 `PushMode.MANUAL` 参数, 如下例:

```
@Push(PushMode.MANUAL)
public class PushyUI extends UI {
```

Servlet 配置

通过 Servlet 配置也可以启用服务器端 PUSH 功能, 并设置 PUSH 模式, 方法是在部署描述文件 `web.xml` 中为 Servlet 指定 `pushmode` 参数. 如果使用兼容 Servlet 3.0 规范的服务器, 你还可以使用 `async-supported` 参数来启用异步处理. 注意, 需要在部署描述文件中使用 Servlet 3.0 的 schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    id="WebApp_ID" version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>Pushy UI</servlet-name>
        <servlet-class>
            com.vaadin.server.VaadinServlet</servlet-class>

        <init-param>
            <param-name>UI</param-name>
            <param-value>com.example.my.PushyUI</param-value>
        </init-param>
```

```
<!-- Enable server push -->
<init-param>
    <param-name>pushmode</param-name>
    <param-value>automatic</param-value>
</init-param>
<async-supported>true</async-supported>
</servlet>
</web-app>
```

11.16.3. 在其他线程中访问 UI

在其他线程中变更 **UI** 对象，并将这些变更 PUSH 到浏览器端，这个操作需要在访问 UI 时对用户 session 加锁。否则，在其他线程中对 UI 的变更会与通常的事件驱动的 UI 更新发生冲突，结果导致数据丢失，或死锁。由于这个问题，你只能使用 `access()` 方法来访问 UI，这个方法会锁住 session，防止前面所说的冲突。这个方法的参数是一个 `Runnable`，并在方法内部执行它。

比如：

```
ui.access(new Runnable() {
    @Override
    public void run() {
        series.add(new DataSeriesItem(x, y));
    }
});
```

在 Java 8 中，无参数的 lambda 表达式可会创建一个 `Runnable`，所以代码可以简化为：

```
ui.access(() ->
    series.add(new DataSeriesItem(x, y)));
```

如果 PUSH 模式是 模式，你需要使用 `push()` 方法，显式地将 UI 的变更 PUSH 到浏览器端。

```
ui.access(new Runnable() {
    @Override
    public void run() {
        series.add(new DataSeriesItem(x, y));
        ui.push();
    }
});
```

下面是一个完整的例子，我们在另一个线程中对 UI 进行更新。

```
public class PushyUI extends UI {
    Chart chart = new Chart(ChartType.AREASPLINE);
    DataSeries series = new DataSeries();

    @Override
    protected void init(VaadinRequest request) {
        chart.setSizeFull();
        setContent(chart);

        // Prepare the data display
        Configuration conf = chart.getConfiguration();
        conf.setTitle("Hot New Data");
        conf.setSeries(series);

        // Start the data feed thread
        new FeederThread().start();
    }
}
```

```

    }

    class FeederThread extends Thread {
        int count = 0;

        @Override
        public void run() {
            try {
                // Update the data for a while
                while (count < 100) {
                    Thread.sleep(1000);

                    access(new Runnable() {
                        @Override
                        public void run() {
                            double y = Math.random();
                            series.add(
                                new DataSeriesItem(count++, y),
                                true, count > 10);
                        }
                    });
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

当在多个 UI 实例(也就是多个用户 Session)之间共享数据时, 你需要更加小心地考虑消息传递机制, 详情见下一节.

11.16.4. 向其他用户发送广播

将需要 PUSH 的消息广播到属于其他用户 session 的 UI 中去, 这样的任务需要使用某种消息传递机制, 来将消息发送给所有注册为消息接收者的 UI. 由于不同 UI 的服务器请求, 是在应用程序服务器中的不同线程中并发处理的, 因此为了避免死锁, 需要对这些线程正确地加锁.

广播器

发送消息到其他用户的标准模式是使用 广播器(broadcaster) 单体, 它负责注册 UI, 并向 UI 安全地广播消息. 为了避免死锁, 建议在独立的线程中使用消息队列来发送消息. 通常最简单最安全的方法是, 使用一个运行在单线程内的 Java **ExecutorService**.

```

public class Broadcaster implements Serializable {
    static ExecutorService executorService =
        Executors.newSingleThreadExecutor();

    public interface BroadcastListener {
        void receiveBroadcast(String message);
    }
}

```

```

    }

    private static LinkedList<BroadcastListener> listeners =
        new LinkedList<BroadcastListener>();

    public static synchronized void register(
        BroadcastListener listener) {
        listeners.add(listener);
    }

    public static synchronized void unregister(
        BroadcastListener listener) {
        listeners.remove(listener);
    }

    public static synchronized void broadcast(
        final String message) {
        for (final BroadcastListener listener: listeners)
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    listener.receiveBroadcast(message);
                }
            });
    }
}

```

在 Java 8 中, 你可以使用 lambda 表达式代替接口来作为事件监听器, 还可以使用无参数的 lambda 表达式来创建 Runnable:

```

for (final Consumer<String> listener: listeners)
    executorService.execute(() ->
        listener.accept(message));

```

接收广播

广播消息的接收者需要实现 receiver 接口, 并将自己注册到广播器中, 才能接收到广播消息. UI 过期时, 监听器需要取消注册. 在消息接收者中更新 UI 时, 必须象前文介绍过的那样安全地实现更新, 更新动作必须通过 UI 的 access() 方法来完成.

```

@Push
public class PushAroundUI extends UI
    implements Broadcaster.BroadcastListener {

    VerticalLayout messages = new VerticalLayout();

    @Override
    protected void init(VaadinRequest request) {
        ... build the UI ...

        // Register to receive broadcasts
        Broadcaster.register(this);
    }

    // Must also unregister when the UI expires
    @Override
    public void detach() {
        Broadcaster.unregister(this);
        super.detach();
    }
}

```

```
    }

    @Override
    public void receiveBroadcast(final String message) {
        // Must lock the session to execute logic safely
        access(new Runnable() {
            @Override
            public void run() {
                // Show it somehow
                messages.addComponent(new Label(message));
            }
        });
    }
}
```

发送广播消息

为了使用上面介绍的广播器单子来发送广播消息，你只需要调用 `broadcast()` 方法，如下。

```
final TextField input = new TextField();
sendBar.addComponent(input);

Button send = new Button("Send");
send.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Broadcast the message
        Broadcaster.broadcast(input.getValue());

        input.setValue("");
    }
});
```


Portal

12.1. 概述	383
12.2. 在 Eclipse 中创建一个常规的 Portlet 工程	383
12.3. 为 Liferay 开发 Vaadin Portlet	386
12.4. Portlet UI	391
12.5. 部署到 Portal 中	392
12.6. Portlet Context	397
12.7. Liferay 的 Vaadin IPC add-on	398

12.1. 概述

Vaadin 支持在 portal 内以 Portlet 的形式运行 UI, portal 标准规约请参见 JSR-286 (Java Portlet API 2.0). Portlet UI 的定义与通常的 UI 一样, 但部署到 portal 中的方式与通常的 Web 应用程序部署方式略有不同, 需要特别的 portlet 描述文件, 等等. 当使用 Vaadin Plugin for Eclipse 或 Maven archetype 来创建 Portlet 工程时, 会自动生成所需要的描述文件.

除了通过 Vaadin UI 来实现用户界面外, portlet 还可以集成到 portal 内, 可以在不同的 portlet 模式中切换, 可以处理特定的 portal 请求, 比如动作和事件.

除了对所有实现了 portal 标准的 portal 提供一般支持之外, Vaadin 还对 Liferay portal 提供了特殊的 support, 本章中的 portal 配置都是针对 Liferay 的. Vaadin 还有特殊的 Liferay IPC add-on, 可以实现在 portlet 之间的通信.

12.2. 在 Eclipse 中创建一个常规的 Portlet 工程

本节中我们介绍如何在 Eclipse 内创建一个常规的 Portlet 工程. 在其他 IDE 中, 或者在无 IDE 的情况下, 你也可以使用 Maven archetype.

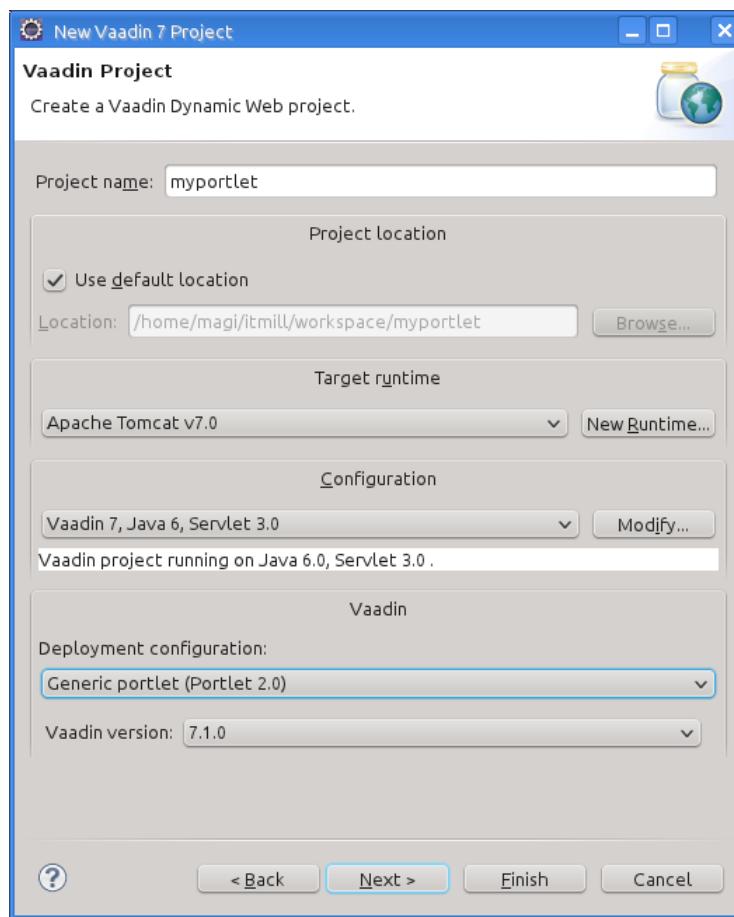
开发 Liferay 的 portlet 时, 你可能会希望使用 Maven archetype 或 Liferay IDE 来创建工程, 详情请参见第 12.3 节“为 Liferay 开发 Vaadin Portlet”.

12.2.1. 使用 Vaadin Plugin 来创建一个工程

Vaadin Plugin for Eclipse 中有一个向导, 可以很便利地创建常规的 portlet 工程. 向导会创建一个 UI 类, 以及所有必须的描述文件.

创建 portlet 工程与创建通常的 Vaadin Servlet 应用程序工程几乎一样. 关于 New Project 向导中的详细操作以及各种选项, 请参见第 2.5.1 节“创建工程”.

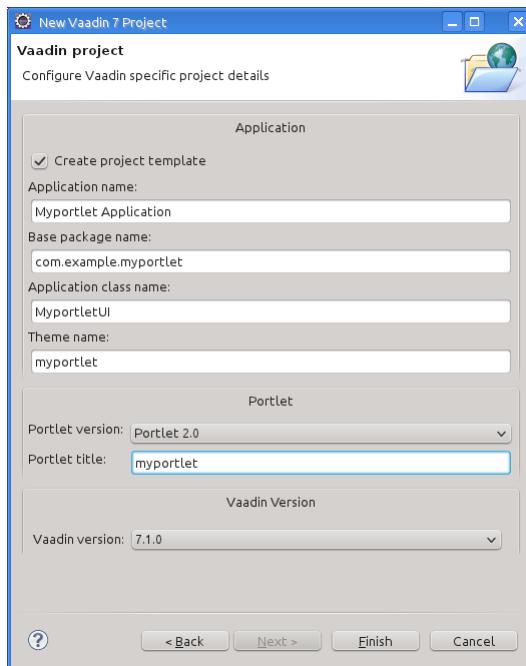
1. 首先从菜单中选择 **File → New → Project...**, 开始创建新工程.
2. 在 **New Project** 窗口中, 选择 **Web → Vaadin 7 Project** 然后点击 **Next** 按钮.
3. 在 **Vaadin Project** 窗口中, 你需要进行基本的 Web 工程设定. 你至少需要输入工程名称, 运行时, 在 **Deployment configuration** 中选择 **Generic Portlet**; 其他设置项目可以使用默认值.



在这一步中, 可以点击 **Finish** 按钮, 结束向导, 其他设置项目会全部使用默认值, 也可以点击 **Next** 按钮, 继续进行其他设置.

4. 在 **Web Module** 窗口, 可以设置基本的 servlet 设定项, 以及 Web 应用程序工程的结构. 这里所有的设定项都有默认值, 通常可以直接使用这些默认值, 然后点击 **Next** 按钮.

5. 在 **Vaadin project** 窗口中, 可以设置一些 Vaadin 相关的应用程序设定项. 这些设定项与通常的 Vaadin 应用程序很类似. 在这里设置它们是最简单的 - 如果将来修改的话, 需要修改很多不同的文件. **Create portlet template** 选项应该会自动选中. 你可以输入一个你希望的 portlet title. 其他项目都可以将来再变更.



Create project template

创建 UI 类, 以及所有需要的 portlet 部署描述文件.

Application name

应用程序名称用做浏览器窗口的标题, 但在 portlet 中通常是不可见的, 在各种部署描述文件中, 应用程序名称也用作标识符, 也可能用来作为某种前缀, 或者带有别的某种前缀.

Base package name

UI 所属的 Java 包名称.

Application class name

UI 类名称. 默认的类名会根据工程名自动得出.

Theme name

所使用的自定义 portlet theme 的名称.

Portlet version

与 Vaadin Project 窗口中的设定一致.

Portlet title

portlet 标题, 定义在 `portlet.xml` 文件中, 可被用作 portlet 的表示名称(至少在 Liferay 中如此). 标题的默认值是工程名称. 标题还用作 `liferay-plugin-package.properties` 文件中的简短描述.

Vaadin version

与 Vaadin Project 窗口中的设定一致.

最后, 点击 **Finish** 按钮, 创建工程.

6. Eclipse 可能会询问你是否要切换到 J2EE perspective. 动态 Web 工程会使用外部 Web 服务器, J2EE perspective 提供了很多工具用来控制服务器, 还会管理应用程序的部署. 因此建议点击 Yes 按钮.

12.3. 为 Liferay 开发 Vaadin Portlet

Vaadin portlet 需要使用资源, 比如服务器端 Vaadin 库, theme, 以及 widget set. 有两种基本方法来发布这些资源: 可以发布到 Liferay 的全局环境中, 这种情况下资源由所有的 Vaadin portlet 共用, 或者也可以使用自包含的 WAR 文件方式, 这种情况下每个 portlet 都带有自己独自的资源.

自包含的 WAR 文件方式比较简单, 而且对于新手来说比较灵活, 因为不同的 portlet 可能会使用不同版本的资源. 目前, 最新的 Maven archetype 支持自包含的 portlet, 而通过 Vaadin Plugin for Eclipse 创建的 portlet 只支持全局发布的方式.

如果在同一个页面上存在多个 Vaadin portlet, 那么使用共用资源的方式效率会更高, 因为它们可以共用相同的资源. 但是, 它们必须使用完全相同的 Vaadin 版本. 对于生产环境, 推荐使用这种方式, 这种环境下你甚至可以使用一个前端服务器(front-end server)来对客户端提供 theme 和 widget set. 关于共享资源的安装方式, 详情请参见第 12.3.5 节“安装 Vaadin 资源”.

在本书编写时, 最新的 Liferay 6.2 中捆绑了 Vaadin 6 版. 如果你希望以共用资源的方式来使用 Vaadin 7 版本的 portlet, 那么你首先需要删除旧版本的 Vaadin, 详情请参见第 12.3.4 节“删除 Liferay 自带的 Vaadin”.

12.3.1. 为 Maven 定义 Liferay Profile

当使用 Maven archetype 或使用 Liferay IDE 来创建 Liferay portlet 工程时, 你需要定义 Liferay profile. 使用 Liferay IDE 时, 你可以在创建工程时创建 profile, 详情请参见第 12.3.3 节“在 Liferay IDE 中创建 Portlet”, 但通过 Maven archetype 创建工程时, 你需要手工定义 profile.

在 settings.xml 中定义 **Profile**

可以在用户的或全局的 Maven settings.xml 文件中定义 Liferay profile. 全局设定文件的路径是 \${MAVEN_HOME}/conf/settings.xml, 用户设定文件是 \${USER_HOME}/.m2/settings.xml. 要创建用户设定文件, 至少需要从全局设定文件中复制相关的文件头和根元素.

```
...
<profile>
  <id>liferay</id>
  <properties>
    <liferayinstall>/opt/liferay-portal-6.2-ce-ga2
    </liferayinstall>
    <plugin.type>portlet</plugin.type>
    <liferay.version>6.2.1</liferay.version>
    <liferay.maven.plugin.version>6.2.1
    </liferay.maven.plugin.version>
    <liferay.auto.deploy.dir>${liferayinstall}/deploy
    </liferay.auto.deploy.dir>

    <!-- Application server version - here for Tomcat -->
    <liferay.tomcat.version>7.0.42</liferay.tomcat.version>
    <liferay.tomcat.dir>
      ${liferayinstall}/tomcat-${liferay.tomcat.version}
    </liferay.tomcat.dir>

    <liferay.app.server.deploy.dir>${liferay.tomcat.dir}/webapps
```

```
</liferay.app.server.deploy.dir>
<liferay.app.server.lib.global.dir>${liferay.tomcat.dir}/lib/ext
</liferay.app.server.lib.global.dir>
<liferay.app.server.portal.dir>${liferay.tomcat.dir}/webapps/ROOT
</liferay.app.server.portal.dir>
</properties>
</profile>
```

参数如下:

`liferayinstall`

Liferay 安装目录的全路径(绝对路径).

`liferay.version`

Liferay 版本, 使用 Maven 的版本号格式. 前两个数字(主版本号和次版本号)与安装包一致. 第三个数字(维护版本)从第一个 GA (general availability) 发布版开始从 0 计数.

`liferay.maven.plugin.version`

通常设定为与 Liferay 版本号一致.

`liferay.auto.deploy.dir`

Liferay 自动部署目录. 默认是 Liferay 安装路径之下的 deploy 目录.

`liferay.tomcat.version (optional)`

如果使用 Tomcat, 这个参数指定 Tomcat 的版本号.

`liferay.tomcat.dir`

Tomcat 安装目录的全路径(绝对路径). 对于 Liferay 中捆绑的 Tomcat, 它的安装目录在 Liferay 安装目录之下.

`liferay.app.server.deploy.dir`

portlet 在 Liferay 所使用的应用服务器上的发布目录. 这个参数的值取决于服务器类型 - 对于 Tomcat, 这个参数应该设置为 Tomcat 安装目录之下的 webapps 目录.

`liferay.app.server.lib.global.dir`

在应用服务器中全局可用的库应该被安装的路径.

`liferay.app.server.portal.dir`

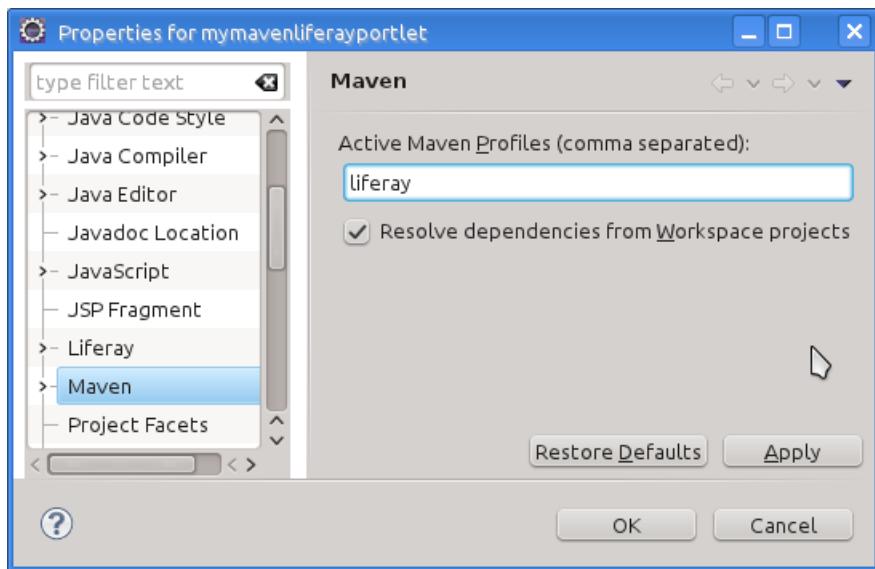
由应用服务器对外提供的静态资源的发布目录, 位于应用服务器的根路径之下.

如果你在工程创建之后修改了这些设定, 你需要更新一下工程内的 POM 文件, 以便重新装载设定内容.

激活 Maven Profile

Maven 2 Plugin for Eclipse (m2e) 必须知道你在工程中使用的是哪些 Maven profile. 这个信息配置在工程属性的 **Maven** 部分内. 在 **Active Maven Profiles** 项目中, 输入 `settings.xml` 文件中定义的 profile ID, 参见 图 12.1 “激活 Maven Liferay Profile”.

图 12.1. 激活 Maven Liferay Profile



12.3.2. 使用 Maven 创建 Portlet 工程

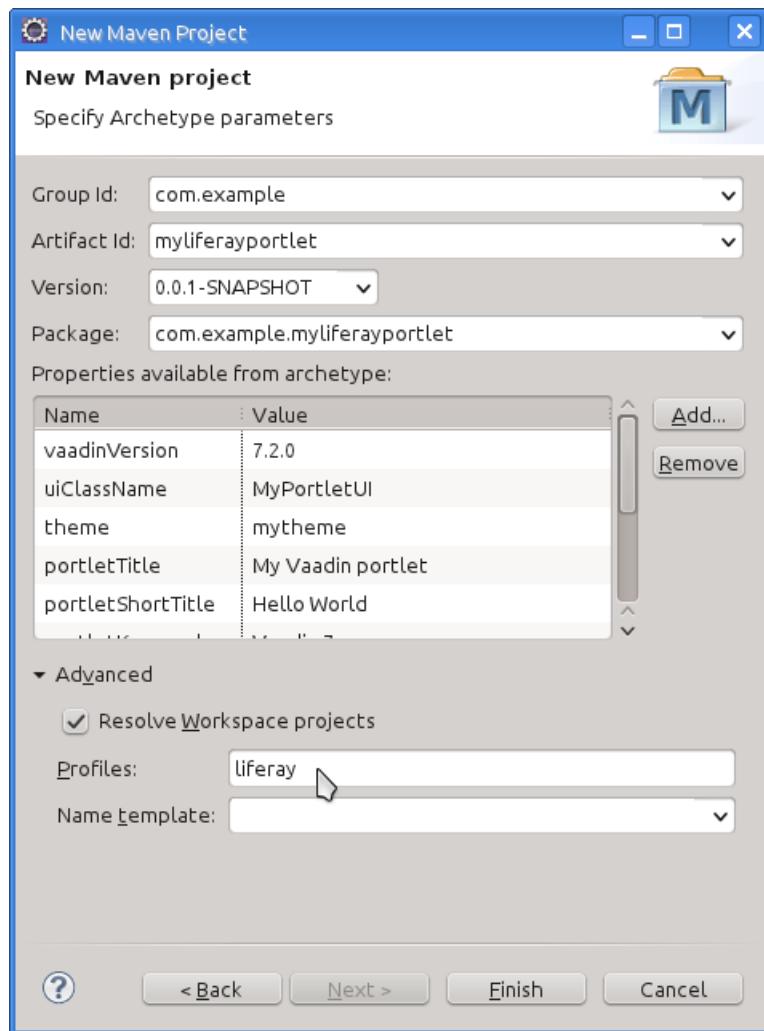
关于 Vaadin Maven 工程的创建方法, 详情请参见 第 2.6 节 “通过 Maven 使用 Vaadin”. 对于 Liferay 工程, 你应该使用 `vaadin-archetype-liferay-portlet`.

Archetype 参数

Archetype 需要指定很多参数. 如果你使用 Maven Plugin for Eclipse (m2e) 来创建工程, 那么你需要在选择 archetype 之后输入相关参数, 参见 图 12.2 “Liferay 工程的 Archetype 参数”.

你至少需要输入 artifact ID. 第 12.3.1 节 “为 Maven 定义 Liferay Profile” 介绍了 Maven profile 的创建, 要激活 Maven profile, 你需要在 **Advanced** 下的 **Profiles** 项目中指定 profile.

图 12.2. Liferay 工程的 Archetype 参数



其他参数如下：

`vaadinVersion`

在 Maven 依赖关系中使用的 Vaadin 发布版本号。

`uiClassName`

创建 UI 类框架代码时使用的类名。

`theme`

使用的 theme。你可以使用工程独自的 theme，但在发布前必须编译 theme，也可以使用 liferay theme。

`portletTitle`

显示在 portlet 标题栏中标题。

`portletShortTitle`

当在需要显示短标题的情况下使用的标题。

`portletKeywords`
在 Liferay 中查找这个 portlet 时使用的关键字.

`portletDescription`
关于这个 portlet 的描述信息.

`portletName`
这个 portlet 的 ID, 用于在配置文件中标识这个 portlet.

`portletDisplayName`
portlet 的名称, 用于需要显示 portlet 名称的环境.

12.3.3. 在 Liferay IDE 中创建 Portlet

Liferay IDE 与 Vaadin plugin 一样, 是以 plugin 形式安装到 Eclipse 中的, 它是用于 Liferay portlet 的开发环境. Liferay IDE 可以将 portlet 发布到开发环境集成的 Liferay 上, 就好象你可以在 Eclipse 内将 servlet 发布到应用程序服务器上一样. 工程创建向导支持创建 Vaadin portlet.

从 portlet 中装载 widget 群, theme, 以及 Vaadin JAR 库是可能的, 但前提是只存在单一的 portlet, 如果有多个 portlet 就会出现问题. 为解决这个问题, Vaadin portlet 需要使用在全局范围安装的 widget 群, theme, 和 Vaadin 库.

本书发布时, Liferay 的最新版本是 Liferay 6.2, 其中捆绑了旧的 Vaadin 6 版. 如果你希望使用 Vaadin 7, 那么你需要删除 Liferay 中捆绑的版本, 然后手工安装新版本, 具体方法参见本章的介绍.

在本章叙述的安装指南中, 我们假定你使用的是与 Apache Tomcat 绑定的 Liferay, 当然你也可以使用任何其他类型的应用程序服务器来运行 Liferay. Tomcat 本身已包含在 Liferay 的安装包之内, 在 `tomcat-x.x.x` 目录内.

12.3.4. 删 除 Liferay 自带的 Vaadin

在安装新版本的 Vaadin 之前, 你需要删除 Liferay 捆绑的版本. 你需要从 portal 的库目录, 以及根路径下的 VAADIN 目录中删除 Vaadin 库的 JAR 文件. 比如, 对于 Liferay 捆绑的 Tomcat, 需要删除的文件通常在以下目录中:

- `tomcat-x.x.x/webapps/ROOT/html/VAADIN`
- `tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar`

12.3.5. 安装 Vaadin 资源

为了使用多个 Vaadin portlet 所需要的共通资源, 你可以将它们安装为全局共用资源, 方法参见下文.

如果你实在捆绑了就版本 Vaadin 的 Liferay 中安装 Vaadin, 那么你首先需要删除相关资源, 详情请参见第 12.3.4 节“删除 Liferay 自带的 Vaadin”.

下文中, 我们假设你只使用 Vaadin 内建的 “liferay” theme, 以及默认的 widget set.

1. 从 Vaadin 下载页面取得 Vaadin 安装包
2. 从安装包中解开以下 Vaadin JAR 文件: `vaadin-server.jar` 和 `vaadin-shared.jar`, 以及它们依赖的 `lib` 目录下的 `vaadin-shared-deps.jar` 和 `jsoup.jar`.
3. 删除 JAR 文件名中的版本数字, 改为上述文件名

4. 将库文件放到 tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/ 目录中
5. 从 vaadin-server.jar, vaadin-themes.jar, 以及vaadin-client-compiled.jar 中解开 VAADIN 文件夹, 然后将其中的内容复制到 tomcat-x.x.x/webapps/ROOT/html/VAADIN 目录中.

```
$ cd tomcat-x.x.x/webapps/ROOT/html  
$ unzip path-to/vaadin-server-7.1.0.jar 'VAADIN/*'  
$ unzip path-to/vaadin-themes-7.1.0.jar 'VAADIN/*'  
$ unzip path-to/vaadin-client-compiled-7.1.0.jar 'VAADIN/*'
```

你需要在 Liferay 的 portal-ext.properties 配置文件中定义 widget 群, theme, 以及 JAR, 详情请参见前文. 这个文件通常位于 Liferay 安装目录中. 关于这个配置文件, 详情请参见 Liferay 文档.

以下是 portal-ext.properties 文件的例子:

```
# Path under which the VAADIN directory is located.  
# (/html is the default so it is not needed.)  
# vaadin.resources.path=/html  
  
# Portal-wide widget set  
vaadin.widgetset=com.vaadin.server.DefaultWidgetSet  
  
# Theme to use  
vaadin.theme=liferay
```

其中允许的参数有:

`vaadin.resources.path`
指定 portal 环境中的资源根路径. 这个设定项的默认值是 /html. 这个路径的实际位置取决于 portal 和应用程序服务器; 在 Liferay 和 Tomcat 环境中, 应该是 Tomcat 安装目录下的 webapps/ROOT/html.

`vaadin.widgetset`
使用的 widget set 类. 需要用点分隔格式指定类名的全路径. 如果这个参数未指定, 会使用默认的 widget 群.

`vaadin.theme`
使用的 theme 的名称. 如果这个参数未指定, 会使用默认的 theme, Vaadin 6 中的默认 theme 是 reindeer.

创建或修改 portal-ext.properties 文件之后, 你需要重启 Liferay.

12.4. Portlet UI

portlet UI 与通常的 Vaadin 应用程序中的 UI 一样, 继承自 **com.vaadin.ui.UI** 基类.

```
@Theme("myportlet")  
public class MyportletUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        final VerticalLayout layout = new VerticalLayout();  
        layout.setMargin(true);
```

```

        setContent(layout);

        Button button = new Button("Click Me");
        button.addClickListener(new Button.ClickListener() {
            public void buttonClick(ClickEvent event) {
                layout.addComponent(
                    new Label("Thank you for clicking"));
            }
        });
        layout.addComponent(button);
    }
}

```

如果你创建的是 Servlet 3.0 工程, 那么自动生成的 UI 框架代码将包含一个静态的 servlet 类, 并被标记为 **@WebServlet**, 详情请参见第 2.5.2 节“浏览一下工程结构”.

```

@WebServlet(value = "/*", asyncSupported = true)
@VaadinServletConfiguration(productionMode = false,
                            ui = MyportletUI.class)
public static class Servlet extends VaadinServlet {
}

```

这样的代码使你可以在开发过程中使用 servlet 容器来运行 portlet UI, 这样的方式要比将 portal 发布到服务器上简便一些. 对于 Servlet 2.4 工程, 会创建 web.xml 配置文件.

和通常的应用程序一样, Portlet 的 theme 使用 **@Theme** 注解来定义. UI 的 theme 必须与 portal 中安装的 theme 一致. 你可以使用 Vaadin 内建的任何一种 theme. 为了与 Liferay theme 兼容, 还存在一个特别的 liferay theme. 如果你使用自定义 theme, 你需要使用 theme 编译器将它编译为 CSS, 然后安装到 portal 的 VAADIN/themes 路径下, 并静态地向外提供.

除 UI 类之外, 你还需要 portlet 描述文件, Vaadin 库文件, 以及其他文件, 详情请见后文. 图 12.3 “Eclipse 中的 Portlet 工程结构”展示了在 Eclipse 中的完整的工程结构.

使用 **Add Application** 菜单可以将应用程序作为 portlet 安装到 Liferay 中, 此时的应用程序运行结果见图 12.4 “Hello World Portlet”.

图 12.4. Hello World Portlet

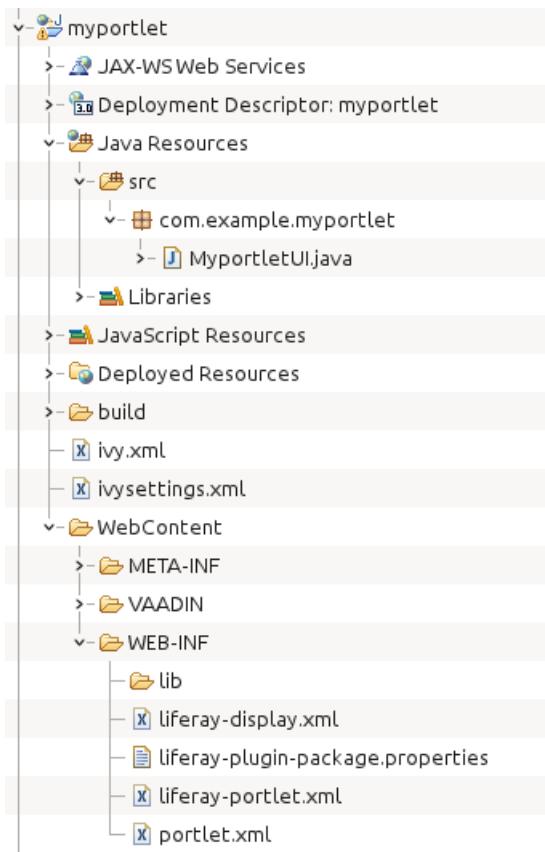


12.5. 部署到 Portal 中

要在 portal 中部署一个 portlet WAR 包, 你需要提供一个 portlet.xml 部署描述文件, 部署描述文件由 Java Portlet API 2.0 标准(JSR-286) 定义. 此外, 你可能还需要包含 portal 厂商独有的部署描述文件. Liferay 所需要的部署描述文件如下.

将一个 Vaadin UI 部署为 portlet, 与部署一个通常的应用程序同样简单. 你不需要对 UI 本身做任何修改, 只需要以下内容:

图 12.3. Eclipse 中的 Portlet 工程结构



- 将应用程序打包为 WAR 形式
 - WEB-INF/portlet.xml 部署描述文件
 - WEB-INF/liferay-portlet.xml Liferay 专有部署描述文件
 - WEB-INF/liferay-display.xml Liferay 专有部署描述文件
 - WEB-INF/liferay-plugin-package.properties Liferay 专有设置文件
- 安装 Widget 群到 portal 中(可选)
- 安装 theme 到 portal 中(可选)
- 安装 Vaadin 库文件到 portal 中(可选)
- Portal 配置文件(可选)

使用 Vaadin Plugin for Eclipse 创建 portlet 工程时, 向导会为你创建这些文件, 详情请参见第 12.2 节“在 Eclipse 中创建一个常规的 Portlet 工程”。

为了在单个的 portal 页面中同时运行两个或以上的 Vaadin portlet, 你需要安装 widget 群和 theme 到 portal 中。这种情况是很容易发生的, 因此我们通常建议你安装它们。针对 Liferay 的安装指南请参见第 12.3 节“为 Liferay 开发 Vaadin Portlet”, 对于其他 portal 服务器的情况, 安装过程类似。

除 Vaadin 库文件外, 你还需要将 portlet.jar 放在你的工程类路径中。但请注意, 你一定不能将 portlet.jar 文件和 Vaadin JAR 文件一样放在 WEB-INF/lib 目录下, 也不能将这个文件包含在需要部署的 WAR 之内, 因为这样会导致它与 portal 内部的 portlet 库文件发生冲突。这种冲突会导致 “ClassCastException: ...VaadinPortlet cannot be cast to javax.portlet.Portlet” 之类的错误。

12.5.1. Portlet 部署描述符

portlet WAR 文件中必须包含 portlet 描述符, 路径是 WEB-INF/portlet.xml. portlet 定义包括: portlet 名称, 与 servlet 的映射, portlet 支持的模式, 以及其他配置. 下面是 portlet.xml 部署描述文件中, 一个 portlet 定义的简单例子.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<portlet-app
    xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.0"
    xsi:schemaLocation=
        "http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
         http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">

    <portlet>
        <portlet-name>Portlet Example portlet</portlet-name>
        <display-name>Vaadin Portlet Example</display-name>

        <!-- Map portlet to a servlet. -->
        <portlet-class>
            com.vaadin.server.VaadinPortlet
        </portlet-class>
        <init-param>
            <name>UI</name>

            <!-- The application class with package name. -->
            <value>com.example.myportlet.MyportletUI</value>
        </init-param>

        <!-- Supported portlet modes and content types. -->
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>edit</portlet-mode>
            <portlet-mode>help</portlet-mode>
        </supports>

        <!-- Not always required but Liferay requires these. -->
        <portlet-info>
            <title>Vaadin Portlet Example</title>
            <short-title>Portlet Example</short-title>
        </portlet-info>
    </portlet>
</portlet-app>
```

在 portlet.xml 中列举所支持的 portlet 模式, 可以在 portal 用户界面中激活相应的控制功能, 来切换不同的 portlet 模式, 详情请见后文.

12.5.2. Liferay Portlet 描述符

Liferay 需要特殊的 liferay-portlet.xml 描述文件, 其中定义了 Liferay 独有的参数. 尤其是, Vaadin portlet 必须定义为 "*instanceable*", 而不是 "*ajaxable*".

前例中 portlet 的描述文件如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE liferay-portlet-app PUBLIC
```

```
--//Liferay//DTD Portlet Application 4.3.0//EN"
"http://www.liferay.com/dtd/liferay-portlet-app_4_3_0.dtd">

<liferay-portlet-app>
    <portlet>
        <!-- Matches definition in portlet.xml.          -->
        <!-- Note: Must not be the same as servlet name. -->
        <portlet-name>Portlet Example portlet</portlet-name>

        <instanceable>true</instanceable>
        <ajaxable>false</ajaxable>
    </portlet>
</liferay-portlet-app>
```

`liferay-portlet.xml` 部署描述文件的更多详情, 请参见 Liferay 的文档.

12.5.3. Liferay Display 描述符

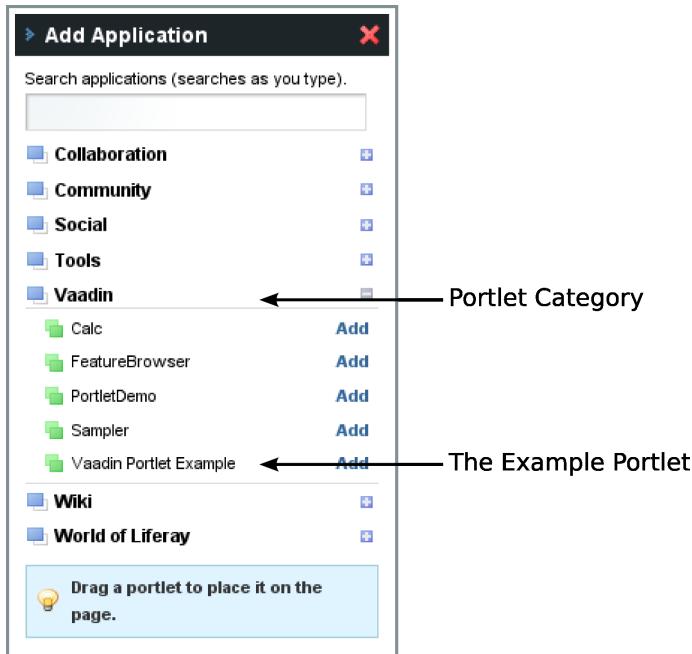
WEB-INF/liferay-display.xml 文件定义 portlet 在 Liferay 的 **Add Application** 画面中所属的种别. 如果没有这个定义, portlet 将属于 "Undefined" 种别.

下面的显示配置, 包含在示例 WAR 中, 将 Vaadin portlet 放在 "Vaadin" 种别中, 运行结果见图 12.5 "Add Application 窗口中的 Portlet 种别".

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC
    "-//Liferay//DTD Display 4.0.0//EN"
    "http://www.liferay.com/dtd/liferay-display_4_0_0.dtd">

<display>
    <category name="Vaadin">
        <portlet id="Portlet Example portlet" />
    </category>
</display>
```

图 12.5. Add Application 窗口中的 Portlet 种别



关于如何在 `liferay-display.xml` 部署描述文件中配置 Portlet 种别, 更多详细信息请参见 Liferay 文档.

12.5.4. Liferay Plugin Package Properties 文件

`liferay-plugin-package.properties` 文件定义了 portlet 的很多设定, 其中最重要的是指定如何使用 Vaadin JAR 文件.

```

name=Portlet Example portlet
short-description=myportlet
module-group-id=Vaadin
module-incremental-version=1
#change-log=
#page-uri=
#author=
license=Proprietary
portal-dependency-jars=\
    vaadin.jar

name
    plugin 名称必须与 portlet 名称一致.

short-description
    对 plugin 的简短描述. 默认值是工程名称.

module-group-id
    应用程序 group ID, 与 liferay-display.xml 中的 category ID 一致.

license
    plugin 的 license 种类; 默认值是 "proprietary".

```

portal-dependency-jars

指定 portlet 依赖的 JAR 库文件. 这个项目的值应该是 vaadin.jar, 除非你需要使用某个特定的版本. JAR 文件必须安装到 portal 内, 比如, 在与 Tomcat 绑定的 Liferay 环境中, 应该安装到 tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar 路径.

12.5.5. 使用单一的 Widget 群

如果你只有一个 Vaadin 应用程序需要运行在 portal 内, 那么你可以按照前面所说的方法部署 WAR 文件即可. 但是, 如果存在多个应用程序, 尤其是它们使用不同的自定义 widget 群时, 就会发生问题, 因为一个 portal 窗口在同一时刻只能装载一个单独的 Vaadin widget 群. 解决这个问题的方法, 可以是使用继承或复合的方式, 将所有应用程序中的不同的 widget 群组合为一个单独的 widget 群.

比如, 如果在某些 portlet 中使用默认的 widget 群, 那么对所有 portlet 都应该使用以下设置, 让它们都使用相同的 widget 群:

```
<portlet>
  ...
  <!-- Use the portal default widget set for all portal demos. -->
  <init-param>
    <name>widgetset</name>
    <value>com.vaadin.portal.PortalDefaultWidgetSet</value>
  </init-param>
  ...

```

PortalDefaultWidgetSet 类继承自 **SamplerWidgetSet**, 这个类又继承自 **DefaultWidgetSet**. **DefaultWidgetSet** 实际上是 **PortalDefaultWidgetSet** 的一个子集, **PortalDefaultWidgetSet** 中还包含示例程序所需要的 widget. 其他应用程序, 如果只需要通常的 **DefaultWidgetSet**, 并且不指定自己的 widget, 也可以使用更大的 widget 群, 使得它们与示例程序兼容. **PortalDefaultWidgetSet** 还会是与 Liferay 5.3 会更高版本绑定的默认 Vaadin widget 群.

如果你的 portlet 包含在多个 WAR 文件中, 这是经常发生的情况, 你需要安装 widget 群和 theme 到整个 portal 环境中, 使得所有的 portlet 都可以使用这些 widget 群和 theme. 关于如何在 portal 中配置 widget 群, 详情请参见第 12.3 节“为 Liferay 开发 Vaadin Portlet”.

12.5.6. 构建 WAR 包文件

为了部署 portlet, 你需要构建 WAR 包文件. 对于生产环境的部署, 你可能希望使用 Maven 或 Ant 脚本来构建包. 在 Eclipse 中, 你可以在工程上点击鼠标右键, 然后选择 **Export → WAR** 菜单. 然后为保指定名称和目标运行环境. 如果你已经在 portal 中安装了 Vaadin, 参见第 12.3 节“为 Liferay 开发 Vaadin Portlet”, 你应该从 WAR 包文件中排除 Vaadin 库, 以及 widget 群和 theme.

12.5.7. 部署 WAR 包文件

WAR 包文件具体的部署方法取决于 portal 类型. 在 Liferay 中, 只需要简单地将它放入 Liferay 安装目录的 `deploy` 子目录下. 部署过程由 Liferay 所在的应用服务器完成; 比如, 如果使用与 Tomcat 绑定的 Liferay, 你会在 Liferay 内包含的 Tomcat 安装目录下的 `webapps` 目录中, 看到展开的包.

12.6. Portlet Context

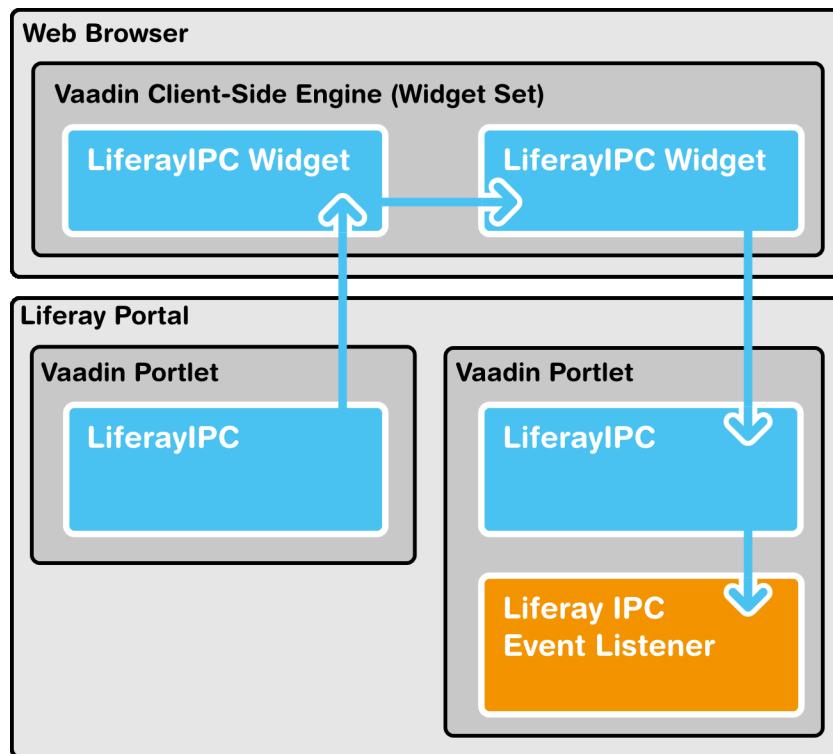
本节未完成.

12.7. Liferay 的 Vaadin IPC add-on

Portlet 通常不是独立存在的。一个页面可以包含多个 portlet，当用户操作一个 portlet 时，你可能需要让其他 portlet 立即作出反应。使用 Vaadin portlet 通常无法完成这个任务，因为 Vaadin 应用程序需要在客户端使用 Ajax 请求来变更它的界面。而另一方面，Portlet 2.0 规约中通常的 portlet 间通信(IPC, inter-portlet communication)机制需要刷新整个页面，但这种模式不适合于 Vaadin，也适合于通常的 Ajax 应用程序，因为它们不需要刷新整个页面。一种解决方法是在服务器端实现 portlet 间的通信，然后使用服务器 PUSH 机制来更新客户端界面。

Vaadin IPC for Liferay 插件使用另一种方案，它在客户端实现 portlet 间通信。事件(消息)通过 **LiferayIPC** 组件来发送，客户端 widget 负责将这些事件转发给其他 portlet，详情见图 12.6 “Vaadin IPC for Liferay 架构”。

图 12.6. Vaadin IPC for Liferay 架构

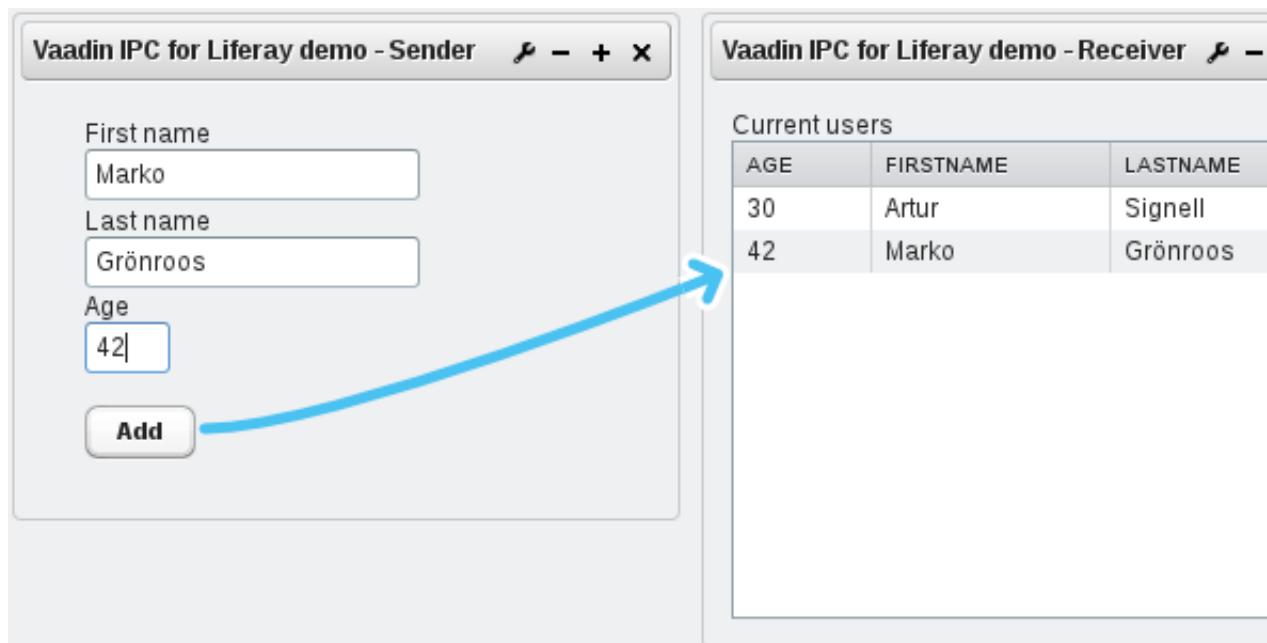


Vaadin IPC for Liferay 使用 Liferay JavaScript 事件 API 来实现客户端 portlet 间通信，因此你的 portlet 间通信可以和其他 Liferay portlet 一样简单地实现。

注意，你只能在同一个页面内的 portlet 之间使用这种通信方式。

图 12.7 “在两个 Portlet 中使用 Vaadin IPC Add-on 示例”展示了 Vaadin IPC for Liferay 的实例。在一个 portlet 内输入一个新的项目，会实时地更新到另一个 portlet 中。

图 12.7. 在两个 Portlet 中使用 Vaadin IPC Add-on 示例



12.7.1. 安装 Add-on

Vaadin IPC for Liferay add-on 可以通过 Vaadin Directory 得到, 也可以从 Maven repository 得到. 如果希望下载安装包, 或者希望查看在 Maven 或 Ivy 中的依赖设定, 请参见本 add-on 在 Vaadin Directory 内的页面, 关于 add-on 的安装, 详情请参见第 17 章 使用 Vaadin Add-on.

安装包的内容如下:

`vaadin-ipc-for-liferay-x.x.x.jar`

安装包中的 add-on JAR 文件必须安装到根路径的 WEB-INF/lib 目录下. 这个目录的位置取决于服务器类型 - 比如对于运行在 Tomcat 中的 Liferay, 它应该在服务器的 webapps/ROOT 路径下.

`doc`

文档目录包含 README.TXT 文件, 它简要描述了安装包的内容, 还有 licensing.txt 和 license-asl-2.0.txt 文件, 介绍了这个插件的许可证(使用 Apache License 2.0). 在 doc/api 文件夹之下包含了这个 add-on 完整的 JavaDoc API 文档.

`vaadin-ipc-for-liferay-x.x.x-demo.war`

一个包含 portlet 示例的 WAR 包. 安装库文件并编译 widget 群后, 详情见下文, 你可以将这个 WAR 部署到 Liferay 中, 然后添加其中的两个示例 portlet 到一个页面中, 见图 12.7 “在两个 Portlet 中使用 Vaadin IPC Add-on 示例”. 示例的源代码可以从 dev.vaadin.com/svn/addons/IPCforLiferay/trunk/ 得到.

add-on 中包含一个 widget 群, 你必须编译它, 并安装到 portal 中.

12.7.2. 基本的通信功能

LiferayIPC 是一个不可见的 UI 组件, 可以用来在两个或更多的 Vaadin portlet 之间发送消息. 就象通常的 UI 组件一样, 你可以将它添加到一个应用程序布局中.

```
LiferayIPC liferayipc = new LiferayIPC();
layout.addComponent(liferayipc);
```

如果你将来修改 portlet 的布局, 你应当小心, 不要从 portlet 中删除这个不可见组件.

这个组件既可以发送消息, 也可以接收消息, 详情见后述.

发送事件

你可以使用 `sendEvent()` 方法来发送事件(消息), 这个方法接受的参数是事件 ID 和消息数据. 事件会被广播发送到监听中的所有 portlet. 事件 ID 是字符串, 可以用来标识对一个事件的应答, 或者标识事件类型.

```
liferayipc.sendEvent("hello", "This is Data");
```

如果你需要发送更加复杂的数据, 你需要将你的数据格式化, 或者序列化为字符串形式, 详情请参见第 12.7.5 节“数据的序列化和编码”.

接收事件

如果一个 portlet 希望从其他 portlet 接收事件(消息), 它需要使用 `addListener()` 方法在组件中注册一个监听器. 监听器收到的消息表达为 **LiferayIPCEvent** 对象. 消息监听处理器内建了使用 ID 来过滤事件的功能, 你可以将监听的事件 ID 作为 `addListener()` 方法的第一个参数. 实际的消息数据保存在 `data` 属性中, 你可以通过 `getData()` 方法得到它.

```
liferayipc.addListener("hello", new LiferayIPCEventListener() {
    public void eventReceived(LiferayIPCEvent event) {
        // Do something with the message data
        String data = event.getData();
        Notification.show("Received hello: " + data);
    }
});
```

监听器添加到 **LiferayIPC** 之后, 可以使用 `removeListener()` 方法来删除.

12.7.3. 注意事项

当使用 Vaadin IPC for Liferay 进行 portlet 间通信时, 既要考虑安全性问题, 也要考虑效率问题.

浏览器安全问题

由于消息数据是通过客户端(浏览器)来传递的, 因此浏览器中运行的任何代码都可以访问这些数据. 你应当注意, 在客户端消息传递机制中, 不要暴露机密数据. 而且, 浏览器端运行的恶意代码还可以篡改或伪造消息. 对消息进行安全过滤可以解决消息伪造的问题, 数据加密则可以同时解决消息篡改和伪造的问题. 你也可以通过 `session` 属性或数据库来共享敏感数据, 而客户端 IPC 机制只用来发送通知, 告知相关使用者这些数据已经可用.

效率

使用浏览器来发送数据需要使用 HTTP 请求来装载和发送这些数据. 数据保存在浏览器的内存空间中, 而使用客户端 JavaScript 代码来处理大的数据会耗费一些时间. 注意, 大的消息数据可能会降低应用程序的应答速度, 而且极端情况下还有可能超过浏览器的内存极限, 或者超过 JavaScript 的执行时间极限.

12.7.4. 使用 Session 属性进行通信

在很多情况下，比如考虑安全性和效率的情况下，比较好的解决方案是在服务器端传递大块数据，而客户端的 IPC 机制只用来通知其他 portlet 读取新数据。Session 属性是一个在服务器端共享数据的便利方法。你还可以共享任意的对象，而不仅仅是字符串。

session 中的变量需要指定 范围，范围应该指定为 `APPLICATION_SCOPE`。其中的 "application" 代表变量在 portlet 所属的整个 Java Web 应用程序(WAR) 范围内有效。

如果参与通信的 portlet 属于相同的 Java Web 应用程序(WAR)，那么不必进行特别的配置。你也可以在属于不同的 WAR 的 portlet 之间通信，这种情况下你需要将 `liferay-portlet.xml` 中的 `private-session-attributes` 参数设置为 `false`，禁用它。关于 Liferay 配置的更多详细信息，请参见 Liferay 文档。

在同一个 WAR 内的多个 portlet 之间共享，你可以共享 Java 对象，而不仅仅是字符串。如果 portlet 属于不同的 WAR，它们通常会有不同的类装载器，因此会导致不兼容，因此你只能使用字符串进行通信，其他数据对象都需要序列化。

Session 属性可以通过 **PortletSession** 对象来访问，可以通过 Vaadin **Application** 类得到 portlet context，再得到 PortletSession 对象。

```
Person person = new Person(firstname, lastname, age);
...
PortletSession session =
    ((PortletApplicationContext2)getContext()).getPortletSession();
// Share the object
String key = "IPCDEMO_person";
session.setAttribute(key, person,
    PortletSession.APPLICATION_SCOPE);
// Notify that it's available
liferayipc.sendEvent("ipc_demo_data_available", key);
```

你可以在 **LiferayIPCEventListener** 中得到上例中的属性数据，如下：

```
public void eventReceived(LiferayIPCEvent event) {
    String key = event.getData();
    PortletSession session =
        ((PortletApplicationContext2)getContext()).getPortletSession();
    // Get the object reference
    Person person = (Person) session.getAttribute(key);
    // We can now use the object in our application
    BeanItem<Person> item = new BeanItem<Person> (person);
    form.setItemDataSource(item);
}
```

注意，当一个共享对象绑定到 UI 组件时，如果另一个 portlet 改变了这个对象，它并不会自动更新到 UI 组件中。这个问题与通常的双重绑定问题是一样的。

12.7.5. 数据的序列化和编码

IPC 事件只支持发送纯文本字符串，因此如果你的数据是对象，或者其他非字符串的数据，你就需要将数据格式化或者序列化为字符串形式。比如，示例应用程序将它的数据模型格式化为分号分隔的字符串序列，如下：

```
private void sendPersonViaClient(String firstName,
                                 String lastName, int age) {
    liferayIPC_1.sendEvent("newPerson", firstName + ";" +
                          lastName + ";" + age);
}
```

对于实现了 `Serializable` 接口的任何类，你都可以使用标准的 Java 序列化机制。但传输的数据中不能包含任何控制字符，因此你还要对字符串进行编码，比如，可以使用 Base64 编码。

```
// Some serializable object
MyBean mybean = new MyBean();
...

// Serialize
ByteArrayOutputStream baost = new ByteArrayOutputStream();
ObjectOutputStream oost;
try {
    oost = new ObjectOutputStream(baost);
    oost.writeObject(mybean); // Serialize the object
    oost.close();
} catch (IOException e) {
    Notification.show("IO PAN!"); // Complain
}

// Encode
BASE64Encoder encoder = new BASE64Encoder();
String encoded = encoder.encode(baost.toByteArray());

// Send the IPC event to other portlet(s)
liferayipc.sendEvent("mybeanforyou", encoded);
```

对于这样的消息，你可以在收信端将它反序列化，如下：

```
public void eventReceived(LiferayIPCEvent event) {
    String encoded = event.getData();

    // Decode and deserialize it
    BASE64Decoder decoder = new BASE64Decoder();
    try {
        byte[] data = decoder.decodeBuffer(encoded);
        ObjectInputStream ois =
            new ObjectInputStream(
                new ByteArrayInputStream(data));

        // The deserialized bean
        MyBean serialized = (MyBean) ois.readObject();
        ois.close();

        ... do something with the bean ...

    } catch (IOException e) {
        e.printStackTrace(); // Handle somehow
    } catch (ClassNotFoundException e) {
```

```
    e.printStackTrace(); // Handle somehow
}
}
```

12.7.6. 与非 Vaadin Portlet 通信

你可以使用 Vaadin IPC for Liferay 来实现 Vaadin 应用程序与其他 portlet, 比如 JSP portlet, 之间的通信. 这个 add-on 使用通常的 Liferay JavaScript 事件来传送事件. 示例 WAR 中包含了两个 JSP portlet, 演示了这样的通信功能.

从非 Vaadin portlet 发送事件时, 可以使用 JavaScript `Liferay.fire()` 方法来触发事件, 参数是事件 ID 和消息内容. 比如, 在 JSP 中你可以包含以下代码:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
           prefix="portlet" %>
<portlet:defineObjects />

<script>
function send_message() {
    Liferay.fire('hello', "Hello, I'm here!");
}
</script>

<input type="button" value="Send message"
       onclick="send_message()" />
```

你可以使用 Liferay JavaScript 事件处理器来接受事件. 可以使用 Liferay 对象的 `on()` 方法来定义处理器. 这个方法接受的参数是事件 ID 和回调函数. 同样的, 在 JSP 中你可以包含以下代码:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
           prefix="portlet" %>
<portlet:defineObjects />

<script>
Liferay.on('hello', function(event, data) {
    alert("Hello: " + data);
});
</script>
```


Vaadin

13.1. 概述	405
13.2. 安装客户端开发环境	406
13.3. 客户端模块描述文件	406
13.4. 编译客户端模块	407
13.5. 创建自定义 Widget	408
13.6. 调试客户端代码	409

本章将简要介绍 Vaadin 客户端 Framework, 它的架构, 以及相关的开发工具.

13.1. 概述

前面我们介绍过, Vaadin 支持两种开发模式: 服务器端开发和客户端开发. 客户端 Vaadin 代码以 JavaScript 形式运行在 Web 浏览器中. 客户端代码与 Vaadin 的其他代码一样, 是以 Java 语言编写的, 然后使用 Vaadin 客户端编译器编译为 JavaScript. 你可以开发客户端 widget, 将这些 widget 与对应的服务器端组件集成起来, 然后就可以在服务器端 Vaadin 应用程序中使用它们了. 服务器端 Framework 以及大多数 add-on 中的组件也是这样开发的. 除此之外, 你也可以创建纯客户端的 GWT 应用程序, 你可以使用 HTML 页面将这种客户端应用程序装载到浏览器中, 使用这种客户端应用程序时甚至不需要连接服务器.

客户端 Framework 是的基础是 Google Web Toolkit (GWT), 并在 GWT 之上增加了一些功能, 修正了一些 bug. Vaadin 与 GWT 兼容, 支持 GWT 的基本功能集. Vaadin 公司是 GWT 指导委员会的成员, 它与 Google 以及其他支持者一起工作, 确定 GWT 的未来发展方向.



Widget 与组件

Google Web Toolkit 使用术语 *widget* 来代表 UI 组件. 在本书中, 我们用 *widget* 这个词代表客户端组件, 用 *组件* 这个词来泛指一切组件, 有时也用来特指服务器端组件.

Vaadin 服务器端开发的主要思路是，通过客户端引擎将服务器端组件显示到浏览器中。客户端引擎实质上由一组 widget 构成，widget 与 连接器 配合，连接器将 widget 的状态和事件序列化，再发送给对应的服务器端组件。客户端引擎技术上叫做 *widget set*，因为它通常由 widget 组成，并且 widget set 可以相互组合，详情请见后文。

13.2. 安装客户端开发环境

客户端开发所需要的库文件，安装方法请参见 第 2 章 开始使用 Vaadin。最重要的是 `vaadin-client` 库文件，这个库中包含了客户端 Java API，以及 `vaadin-client-compiler`，这个文件中包含了 Vaadin 客户端编译器，用于将 Java 代码编译为 JavaScript 代码。

13.3. 客户端模块描述文件

Vaadin 客户端模块，比如 Vaadin 客户端引擎(widget set) 或纯客户端应用程序，都会被编译为 JavaScript，这些模块定义在 模块描述文件 (`.gwt.xml`)。

要创建一个 Vaadin 客户端引擎，需要定义一个 widget set，这时需要做的是继承一个 widget set 基类，通常应该从 `DefaultWidgetSet` 继承。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
  "-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
  "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">

<module>
  <!-- Inherit the default widget set -->
  <inherits name="com.vaadin.DefaultWidgetSet" />
</module>
```

如果你在开发一个纯客户端应用程序，你应该继承 `com.vaadin.Vaadin`，详情请参见 第 14 章 客户端应用程序。这时，模块描述文件中还需要指定一个 entry-point。

如果使用 Eclipse IDE，“New Vaadin Widget” 向导会自动创建 GWT 模块描述文件。详情请参见 第 16.2.1 节“创建 Widget”。

13.3.1. 指定样式表

客户端模块可以包含 CSS 样式表。模块编译时，这些样式表文件会被复制到编译结果中。在模块描述文件中，需要定义一个 `stylesheet` 元素。

比如，如果你在开发一个自定义 widget，并且希望它有一个默认的样式表，你可以使用如下定义：

```
<stylesheet src="mywidget/styles.css"/>
```

这里使用的文件路径，相对于模块描述文件所在文件夹之下的 `public` 文件夹。

13.3.2. 限定编译目标

widget set 的编译过程会耗费相当长的时间。你可以显著地减少编译时间，方法是只针对你的浏览器来编译 widget set，在开发阶段这是一种很有效的方法。你可以在模块描述文件中设置 `user.agent` 属性来实现这个目的。

```
<set-property name="user.agent" value="gecko1_8"/>
```

`value` 属性应该与你的浏览器类型一致. GWT 所支持的浏览器与 GWT 的版本有关, 以下是 GWT 所支持的浏览器 ID.

表 13.1. GWT 支持的浏览器类型

浏览器 ID	浏览器名称
ie6	Internet Explorer 6
ie8	Internet Explorer 8
gecko1_8	Mozilla Firefox 1.5 及以上版本
safari	Apple Safari 以及其他基于 Webkit 的浏览器, 包括 Google Chrome
opera	Opera
ie9	Internet Explorer 9

关于 GWT 模块描述文件的 XML 格式, 请参见 Google Web Toolkit 开发指南.

13.4. 编译客户端模块

一个客户端模块, 无论是 widget set 还是纯客户端应用程序, 都需要使用 Vaadin 客户端编译器来将它编译为 JavaScript. 开发过程中, 使用开发模式可以在你重新装载页面时进行自动编译, 这个页面则会在初次编译时产生.

由于大多数 Vaadin add-on 都包含 widget, 因此使用 add-on 时通常都需要编译 widget set. 这种情况下, 来自不同 add-on 的多个 widget set 需要编译到一个 *project widget set* 内, 详情请参见第 17 章 使用 Vaadin Add-on.

13.4.1. Vaadin 编译器概述

Vaadin 客户端编译器将 Java 代码编译为 JavaScript. 编译器在 `vaadin-client-compiler` JAR 文件中提供, 你可以使用 Java 命令的 `-jar` 参数来执行这个 JAR 文件. 它依赖于 `vaadin-client` JAR 文件, 其中包含 Vaadin 客户端 Framework.

编译器编译的对象是 客户端模块, 客户端模块可以是纯客户端模块, 也可以是 Vaadin widget set. 也就是 Vaadin 客户端引擎, 其中包含应用程序所使用的 widget. 客户端模块使用模块描述文件来定义, 详情请参见 第 13.3 节 “客户端模块描述文件”.

编译器将编译结果写入目标文件夹中, 其中包含编译产生的 JavaScript, 以及模块中包含的一切静态资源.

13.4.2. 在 Eclipse 环境中编译

Eclipse 中安装 Vaadin Plugin 后, 你只需要点击工具栏中的 **Compile Vaadin widgets** 按钮, 插件就会编译它在工程中找到的 widget set. 如果工厂中存在多个 widget set, 比如一个自定义 widget, 以及一个工程 widget 的情况, 你在点击编译按钮之前, 需要选择需要编译的 widget set 所对应的模块描述文件.

使用 Vaadin Plugin for Eclipse 来编译, 目前要求模块描述符文件名的后缀为 `Widgetset.gwt.xml`, 虽然你也可以使用它来编译 `widget set` 之外的其他客户端模块. 编译结果会输出到 `WebContent/VAADIN/widgetsets` 文件夹之下.

13.4.3. 使用 Ant 编译

在 Vaadin download page 可以找到一段脚本模板, 这段脚本可以使用 Ant 和 Ivy 来编译 widget set. 你可以将这段脚本复制到你的工程中, 配置完成后, 就可以使用 Ant 来运行这段脚本.

13.4.4. 使用 Maven 编译

你可以使用 vaadin:compile goal 来编译 widget set, 如下:

```
$ mvn vaadin:compile
```

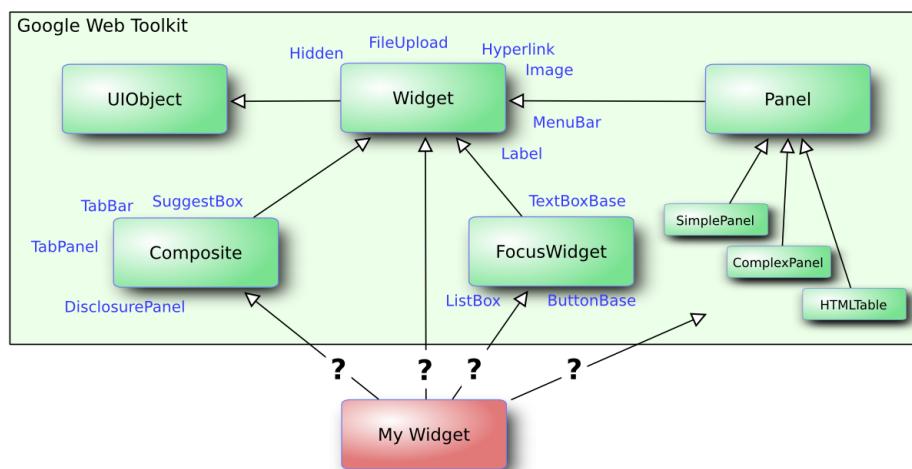
13.5. 创建自定义 Widget

创建一个新的 Vaadin 组件, 通常首先需要开发它的客户端 widget, 再与服务器端组件集成, 然后才可以在服务器端开发中使用这个组件. 除此之外, 你也可以选择创建纯客户端 widget, 这种情况我们会在本节之后的部分中介绍.

13.5.1. 一个简单的 Widget

所有的 widget 都继承自 **Widget** 类, 或这个类的某个子类. 你可以继承任何一种 GWT 核心 widget, 或 Vaadin 扩充的 widget. 通常是继承某个抽象类, 比如 **Composite**. GWT 基本 widget 组件的继承关系图见图 13.1 “GWT Widget 基类继承关系图”. 关于 widget 类的完整说明, 请参见 GWT API 文档.

图 13.1. GWT Widget 基类继承关系图



比如, 为了显示某种自定义文字, 我们可以继承 **Label** widget.

```
package com.example.myapp.client;

import com.google.gwt.user.client.ui.Label;

public class MyWidget extends Label {
    public static final String CLASSNAME = "mywidget";

    public MyWidget() {
        setStyleName(CLASSNAME);
        setText("This is MyWidget");
    }
}
```

```

    }
}

```

Eclipse plugin 自动生成的 widget 框架代码与上面的例子基本类似. 实际运用中建议为 widget 设置一个独有的样式类, 以便使用 CSS 来控制它的样式.

客户端源代码 必须 包含在模块描述文件所指定的包之下的 client 包中, 详情将在后文中介绍.

13.5.2. 使用 Widget

你可以使用自定义 widget, 方法与使用其他 widget 完全一样, 可以将 widget 与服务器端组件集成在一起, 也可以在纯客户端模块中使用 widget, 如下:

```

public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Use the custom widget
        final MyWidget mywidget = new MyWidget();
        RootPanel.get().add(mywidget);
    }
}

```

13.6. 调试客户端代码

Vaadin 目前包括 SuperDevMode 模式, 可以直接在浏览器内调试客户端代码.

SuperDevMode 的前身是 GWT Development Mode, 在最近版本的 Firefox 和 Chrome 中, 它已经不能工作了, 因为浏览器的某些 API 发生了变化. 它在某些平台上还可以工作, 但为了简单起见, 我们推荐使用 SuperDevMode.

13.6.1. 启动开发模式

开发模式会在浏览器内启动应用程序, 并在页面装载时编译客户端模块(或 widget set), 而且可以在 Eclipse 内调试客户端代码. 你可以运行 **com.google.gwt.dev.DevMode** 类来启动开发模式. 这个类需要一些参数, 详情请见后文.

Vaadin Plugin for Eclipse 可以为开发模式创建启动配置. 在工程属性的 Vaadin 页中, 点击 **Create development mode launch** 按钮, 这样就可以在工程中创建一个新的启动配置. 你可以通过菜单项 **Run → Run Configurations** 来编辑这个启动配置.

```

noserver -war WebContent/VAADIN/widgetsets
com.example.myproject.widgetset.MyWidgetSet -startupUrl
http://localhost:8080/myproject -bindAddress 127.0.0.1

```

相关参数如下:

-noserver

开发模式通常会启动自带的 Jetty 服务器, 以便向客户端提供 Web 内容. 如果你开发时使用的 IDE 会将应用程序发布到服务器上, 比如使用 Eclipse 时, 那么你可以使用这个参数来禁用开发模式自带的服务器.

-war

指定 JavaScript 编译输出的路径. 开发纯客户端模块时, 这个参数应该指向 WebContent (使用 Eclipse 时) 或者其下的某个目录. 当编译 widget set 时, 这个参数应该指向 WebContent/VAADIN/widgetsets 目录.

`-startupUrl`

指定应用程序装载页面的地址. 对于服务器端 Vaadin 应用程序, 这个参数应该是 Vaadin 应用程序 Servlet 的路径, 这个路径应该定义在部署描述文件中. 对于纯客户端 widget, 这个参数应该是包含应用程序的页面地址.

`-bindAddress`

这个参数指定开发模式运行时使用的主机 IP 地址. 如果仅在开发环境中调试代码, 这个参数可以直接使用 127.0.0.1. 如果将这个参数设置为主机的一个正确 IP 地址, 可以实现远程调试.

13.6.2. 启动超级开发模式

超级开发模式与旧的开发模式很类似, 区别是它不需要浏览器插件. 从 Java 代码到 JavaScript 的编译是增量进行的, 这样可以显著减少编译时间. 这种模式还可以在浏览器之内直接调试 JavaScript 代码, 甚至直接调试 Java 代码(目前只支持 Chrome).

你可以通过以下步骤启动超级开发模式:

- 你需要在 `.gwt.xml` 模块描述文件中设置一个重定向属性, 如下:

```
<set-configuration-property name="devModeRedirectEnabled" value="true"
/>
```

此外, 你还需要添加 `xsiframe` 链接. 它包含在 **com.vaadin.DefaultWidgetSet** 类和 **com.vaadin.Vaadin** 模块中. 如果不存在这个链接, 你需要包含它, 如下:

```
<add-linker name="xsiframe" />
```

- 编译模块(也就是, widget set), 比如, 在 Eclipse 中点击编译按钮.
- 如果你使用的是 Eclipse, 你需要为超级开发模式创建一个启动配置, 方法是在工程属性的 **Vaadin** 页面中点击 **Create SuperDevMode launch** 按钮.
 - 启动时执行的 main class 应该设置为 **com.google.gwt.dev.codeserver.CodeServer**.
 - 应用程序的参数是模块(或 widget set)的类全名(包括包名), 比如, **com.example.myproject.widgetset.MyprojectWidgetset**.
 - 如果工程的源代码不在工程类路径中, 则需要将源代码添加到启动设定的类路径中.

以上设置只需要进行一次. 完成之后, 你就可以启动超级开发模式, 方法如下:

- 使用前面创建的启动配置, 来运行超级开发模式的代码服务器. 此时会对你的模块或 widget set 进行初次编译.
- 为你的应用程序启动 servlet 容器, 比如, Tomcat.
- 打开你的浏览器, 输入应用程序 URL, 在 URL 末尾添加 `?superdevmode` 参数(如果你继承的不是 **DefaultWidgetSet**, 请参见下面的注意事项). 这个操作会重新编译代码, 完成之后页面会在超级开发模式下重新装载. 你也可以使用 `?debug` 参数, 然后在调试控制台中点击 **SDev** 按钮, 即可进入超级开发模式.

如果你更改了客户端代码, 并在浏览器中刷新页面, 客户端代码会被重新编译, 你可以立即看到修改后的结果.

上述第 3 步的前提是，假定你的模块继承自 **DefaultWidgetSet**. 如果不是这样，你需要在模块的 `onModuleLoad()` 方法的最开始部分添加以下代码：

```
if (SuperDevMode.enableBasedOnParameter()) { return; }
```

或者， 你也可以使用代码服务器(Code Server)提供的 bookmarklet 功能. 首先进入 `http://localhost:9876/`, 然后你可以将 "**Dev Mode On**" 和 "**Dev Mode Off**" 的 bookmarklet 拖放到书签栏中.

13.6.3. 在 Chrome 内调试 Java 代码

Chrome 支持代码映射功能, 使用这个功能, 你可以通过编译后的 JavaScript 来调试编译前的 Java 源代码.

在 Chrome 内点击鼠标右键, 然后选择 **Inspect Element** 即可打开 Chrome Inspector. 在这个界面中点击窗口下方角落处的 "设置" 图标, 然后选中 **Scripts → Enable source maps** 选项. 保持 Inspector 界面打开, 刷新页面, 这时你会在 script 栏中看到 Java 代码, 而不是之前的 JavaScript 代码.

14.1. 概述	413
14.2. 客户端模块的 Entry-Point	414
14.3. 编译和运行客户端应用程序	415
14.4. 装载客户端应用程序	416

本章介绍如何开发客户端 Vaadin 应用程序。

目前，我们只对这个问题进行一些非常简单的介绍。关于 GWT 各种功能的详细介绍，请参见 GWT 文档。

14.1. 概述

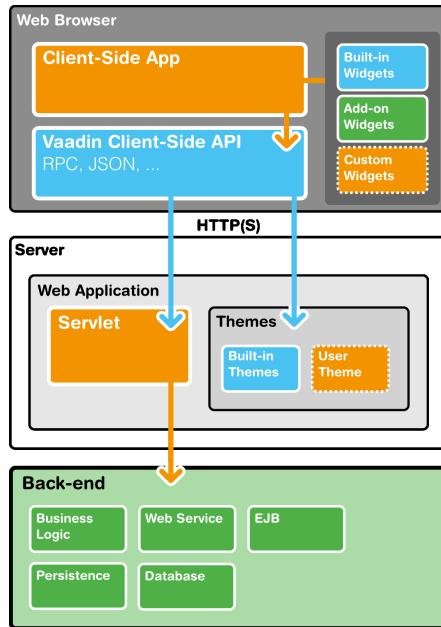
Vaadin 可以开发在浏览器中运行的客户端模块。客户端模块可以使用所有的 GWT widget，以及一部分 Vaadin 特有的 widget。此外，和服务器端应用程序一样，客户端模块也可以使用 theme。客户端应用程序运行在浏览器内，甚至可以不与服务器端发生通信。如果客户端应用程序与服务器端配合，通过服务器端服务来访问数据存储和业务逻辑，这种情况下客户端应用程序可以被看作“胖客户端”，通常的服务器端 Vaadin 应用程序使用的则是“瘦客户端”方案。服务器端服务可以使用与服务器端 Vaadin 应用程序相同的后端服务。在需要 UI 逻辑响应度很高的场合，胖客户端的用途十分广泛，比如游戏，或者使用服务器端无状态服务来支持巨量客户端的情况。

客户端应用程序通过模块来实现，模块就是一个 *entry-point* 类。当模块编译后的 JavaScript 代码被装载进入浏览器时，会执行模块的 `onModuleLoad()` 方法。

我们来看看以下客户端应用程序：

```
public class HelloWorld implements EntryPoint {  
    @Override  
    public void onModuleLoad() {  
        RootPanel.get().add(new Label("Hello, world!"));  
    }  
}
```

图 14.1. 客户端应用程序架构



}
}

客户端应用程序的 UI 构建在一个 HTML 根元素之下，这个根元素可以通过 `RootPanel.get()` 方法得到。关于 `entry-point` 的设计意图以及使用方法的详细文档，请参阅 第 14.2 节“客户端模块的 Entry-Point”。与服务器端 Vaadin UI 一样，客户端应用程序的 UI 也使用 `widget` 层级结构来构建。内建的 `widget` 及其相互关系请参见 第 15 章 客户端 Widget。你也可以使用 Vaadin add-on 中的大量 `widget`，或者创建自己的 `widget`。

客户端模块通过 模块描述文件 来定义，详情请参见 第 13.3 节“客户端模块描述文件”。模块使用 Java 开发，然后通过 Vaadin 编译器编译为 JavaScript 代码，Vaadin 编译器的使用请参见 第 13.4 节“编译客户端模块”。本章的 第 14.3 节“编译和运行客户端应用程序” 将介绍客户端应用程序编译的更多细节。编译产生的 JavaScript 可以通过任何 Web 页面来装载，详情请参见 第 14.4 节“装载客户端应用程序”。

也可以使用附带的 `UI Binder` 声明式(declaratively)地构建客户端 UI。

负责处理客户端 RPC 调用的 `Servlet`，可以使用附带的编译器自动生成。

即使是通常的服务器端 Vaadin 应用程序，如果能在网络连接断开时提供离线工作模式，也是非常有用的。离线模式可以将数据保存在浏览器的本地存储中，因此可以避免使用服务器端存储能力，然后等待连接恢复后再将数据发动到服务器。这种模式通常会与 Vaadin TouchKit 一起配合使用。

14.2. 客户端模块的 `Entry-Point`

客户端应用程序需要一个 `entry-point`，`entry-point` 就是程序执行的入口，类似于服务器端 Vaadin UI 中的 `init()` 方法。

我们来看看以下应用程序：

```
package com.example.myapp.client;
```

```

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.RootPanel;
import com.vaadin.ui.VButton;

public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Create a button widget
        Button button = new Button();
        button.setText("Click me!");
        button.addClickHandler(new ClickHandler() {
            @Override
            public void onClick(ClickEvent event) {
                mywidget.setText("Hello, world!");
            }
        });
        RootPanel.get().add(button);
    }
}

```

在编译之前，需要在一个模块描述文件中定义 entry-point，详情请参阅下节。

14.2.1. 模块描述文件

客户端应用程序的 entry-point 与其他各种配置一起，定义在客户端模块描述文件中，详情请参见第 13.3 节“客户端模块描述文件”。模块描述文件是一个 XML 文件，扩展名是 .gwt.xml.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
"-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
"http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">
<module>
    <!-- Builtin Vaadin and GWT widgets -->
    <inherits name="com.vaadin.Vaadin" />

    <!-- The entry-point for the client-side application -->
    <entry-point class="com.example.myapp.client.MyEntryPoint"/>
</module>

```

你可能会选择继承 **com.google.gwt.user.User**，只使用基本的 GWT widget，而不使用 Vaadin 专有的 widget 和类，在纯客户端应用程序中，这些 Vaadin 专有 widget 和类大多是无法使用的。

你可以将静态资源，比如图片或 CSS 样式表，放在模块描述文件所在文件夹之下的 public 文件夹内（注意，不是 Java 包）。编译模块时，这些资源会被复制到输出文件夹中。通常，在纯客户端应用程序开发中，使用 HTML 宿主文件（HTML host file），或使用 **ClientBundle** 来装载这些资源文件会比较容易一些（详情请参见 GWT 文档），但要注意，如果你的资源文件也用于与服务器端组件的集成，那么前面这些方法是不兼容的。

14.3. 编译和运行客户端应用程序

在编写本书的当前版本时，使用 *Vaadin Plugin for Eclipse* 来编译 widget set 之外的客户端模块，最近进行了一些功能更新，并存在一些限制，因此本书中给出的信息可能并不完全准确。

应用程序需要编译为 JavaScript 才能在浏览器内运行。在部署应用程序之前，以及在初次使用开发模式时，你需要使用 Vaadin 客户端编译器进行编译，详情请参见 第 13.4 节“编译客户端模块”。

在开发过程中，最简单的方式是使用 SuperDevMode，这种方式可以快速地启动客户端代码，并且支持调试功能。详情请参见 第 13.6 节“调试客户端代码”。

14.4. 装载客户端应用程序

你可以使用 HTML 宿主页面(*host page*) 来装载客户端应用程序的 JavaScript 代码，方法是使用 `<script>` 标签来 include 这些 JavaScript，如下：

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />

    <title>Embedding a Vaadin Application in HTML Page</title>

    <!-- Load the Vaadin style sheet -->
    <link rel="stylesheet"
          type="text/css"
          href="/myproject/VAADIN/themes/reindeer/legacy-styles.css"/>
  </head>

  <body>
    <h1>A Pure Client-Side Application</h1>

    <script type="text/javascript" language="javascript"
           src="clientside/com.example.myapp.MyModule/
           com.example.myapp.MyModule.nocache.js">
    </script>
  </body>
</html>
```

JavaScript 模块通过 `<script>` 元素来装载。`src` 属性应该是从宿主页面到编译后的 JavaScript 模块的相对地址。

除 GWT 的核心 widget 之外，如果应用程序还使用了其他的 Vaadin widget，如上例所示，你还需要 include 相关的 Vaadin theme。样式文件的确切路径取决于你的工程结构 - 上例中的路径与通常的 Vaadin 应用程序是一致的，这个例子中的 theme 放在 WAR 内的 VAADIN 文件夹中。

除 CSS 和脚本之外，你还可以在宿主页面中装载客户端应用程序所需要的其他任何资源。

Widget

15.1. 概述	417
15.2. GWT Widget	418
15.3. Vaadin Widget	418

本章概要介绍如何使用 Vaadin 客户端 Framework 来创建客户端应用程序和你自己的 widget.

关于这个主题, 目前我们在本章中只给出一个极简单的介绍. 关于 GWT widget 的更详细信息, 请参见 GWT 文档.

15.1. 概述

Vaadin 客户端 API 是基于 Google Web Toolkit 开发的. 它引入了 *widget* 的概念, 目的是使用 Java 对象来表达 UI 元素, 在浏览器中 UI 元素最终展现为 HTML DOM 元素. 用户在页面中的操作导致的事件, 会被转发给事件处理器, 你可以在事件处理器实现你的 UI 逻辑.

通常, 客户端 widget 分为两大类 - 基本的 GWT widget, 以及 Vaadin 独有的 widget. Vaadin 库中包含了 连接器(connector), 用来实现 Vaadin 独有的 widget 与对应的服务器端组件的集成, 这是 Vaadin 的服务器端开发模型的基础. 关于 widget 与服务器端组件的集成, 详情请参见第 16 章与客户端集成.

客户端 UI 的布局由 *panel* widget 来管理, 这个 widget 的功能等同于 Vaadin 服务器端 API 中的布局组件.

客户端 API 中, 除了显示 API 外, 还包括创建 HTTP 请求, 输出日志, 支持残障用户操作的辅助功能 (accessibility), 国际化, 以及测试, 等等各种功能.

关于 GWT 基本 Framework 的详情, 请参见 <https://developers.google.com/web-toolkit/overview>.

15.2. GWT Widget

GWT widget 是一组展现为 HTML 的 UI 元素. 使用 HTML 来展现 UI 元素, 有两种实现方式: 通过低阶的 DOM API 来操纵 HTML 文档对象模型(DOM), 或者使用 `setInnerHTML()` 方法简单地注入 HTML 代码. UI 的布局使用特殊的 panel widget 来管理.

关于 GWT 基本 widget 的详细信息, 请参见 GWT 开发者向导, 地址是 <https://developers.google.com/web-toolkit/doc/latest/DevGuideUi>.

15.3. Vaadin Widget

除 GWT widget 外, Vaadin 还附带了很多 Vaadin 独有的 widget, 这些 widget 中的一部分可以在纯客户端应用程序中使用. Vaadin widget 的功能与 GWT widget 略有不同, Vaadin widget 的初始目标是与服务器端组件相集成, 但其中的一部分 widget 也可以在客户端应用程序中使用.

```
public class MyEntryPoint implements EntryPoint {  
    @Override  
    public void onModuleLoad() {  
        // Add a Vaadin button  
        VButton button = new VButton();  
        button.setText("Click me!");  
        button.addClickHandler(new ClickHandler() {  
            @Override  
            public void onClick(ClickEvent event) {  
                mywidget.setText("Clicked!");  
            }  
        });  
  
        RootPanel.get().add(button);  
    }  
}
```

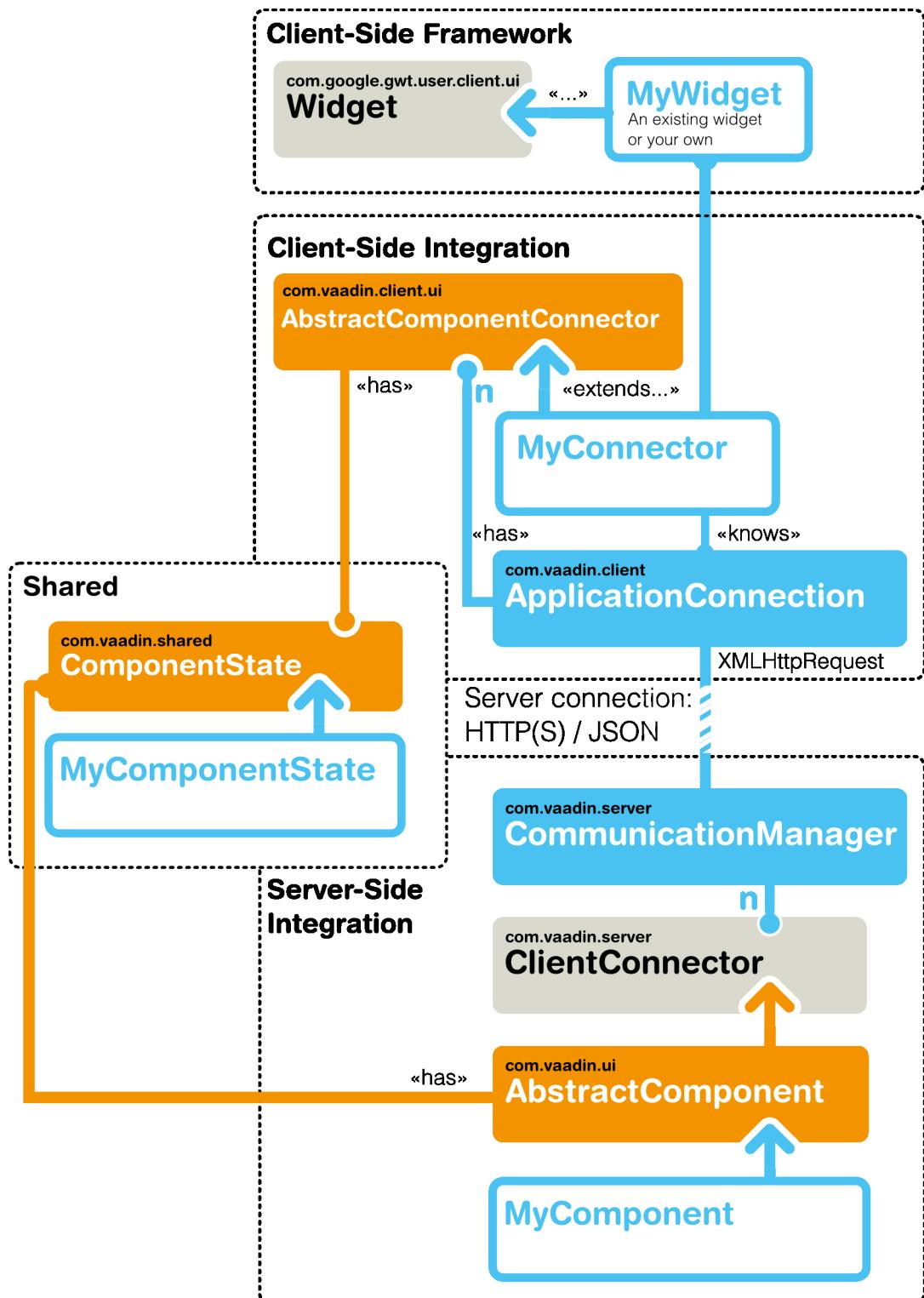
16.1. 概述	419
16.2. 使用 Eclipse 简化工作	422
16.3. 创建服务器端组件	425
16.4. 使用连接器实现客户端与服务器端的集成	426
16.5. 状态信息共享	427
16.6. 客户端与服务器端之间的 RPC 调用	430
16.7. 组件与 UI 扩展	431
16.8. Widget 的样式控制	433
16.9. 组件容器	435
16.10. 客户端的一些高级问题	435
16.11. 创建 Add-on	435
16.12. 从 Vaadin 6 迁移	441
16.13. JavaScript 组件与扩展的集成	441

本章将介绍如何实现客户端 widget 或 JavaScript 组件与服务器端组件的集成。Vaadin 中的所有服务器端标准组件，其客户端实现都使用相同的客户端接口和模式。

16.1. 概述

Vaadin 组件由两部分组成：服务器端组件，以及客户端组件。在 Google Web Toolkit (GWT) 术语中，客户端组件又被称为 *widget*。Vaadin 应用程序使用服务器端组件的 API，服务器端组件最终在浏览器中显示为客户端 *widget*。与服务器端一样，客户端 *widget* 构成一个布局 *widget* 的层级树，其他通常 *widget* 则是这个层级树上的最末端叶节点。

图 16.1. 客户端 Widget 的集成



客户端 widget 与服务器端组件之间的通信由 连接器(connector) 管理, 连接器负责在客户端与服务器端之间维护 widget 状态与事件的双向同步.

展现 UI 时, 会为每个服务器端组件创建对应的客户端连接器(connector)和客户端 widget. 组件与连接器的映射关系, 通过 connector 类中的 `@Connect` 注解来定义, widget 则由 connector 类负责创建.

服务器端组件的状态信息, 会使用 共享的状态信息 对象自动同步到客户端 widget 中. 共享的状态信息对象实现了 `ComponentState` 接口, 并在服务端组件与客户端组件中同时使用. 在客户端, 连接器始终可以访问到它的状态对象, 以及它的父组件和子组件状态.

这种状态信息共享方案假定状态信息中使用的是 Java 标准数据类型, 比如基本数据类型(int, float 等)或基本数据类型的封装类(Integer, Float 等), **String**, 数组, 以及由这些类型数据组成的几种集合类型 (**List**, **Set**, 和 **Map**). 除此之外, Vaadin 的 **Connector** 类和其他几种特别的内部类型也可以作为共享状态信息.

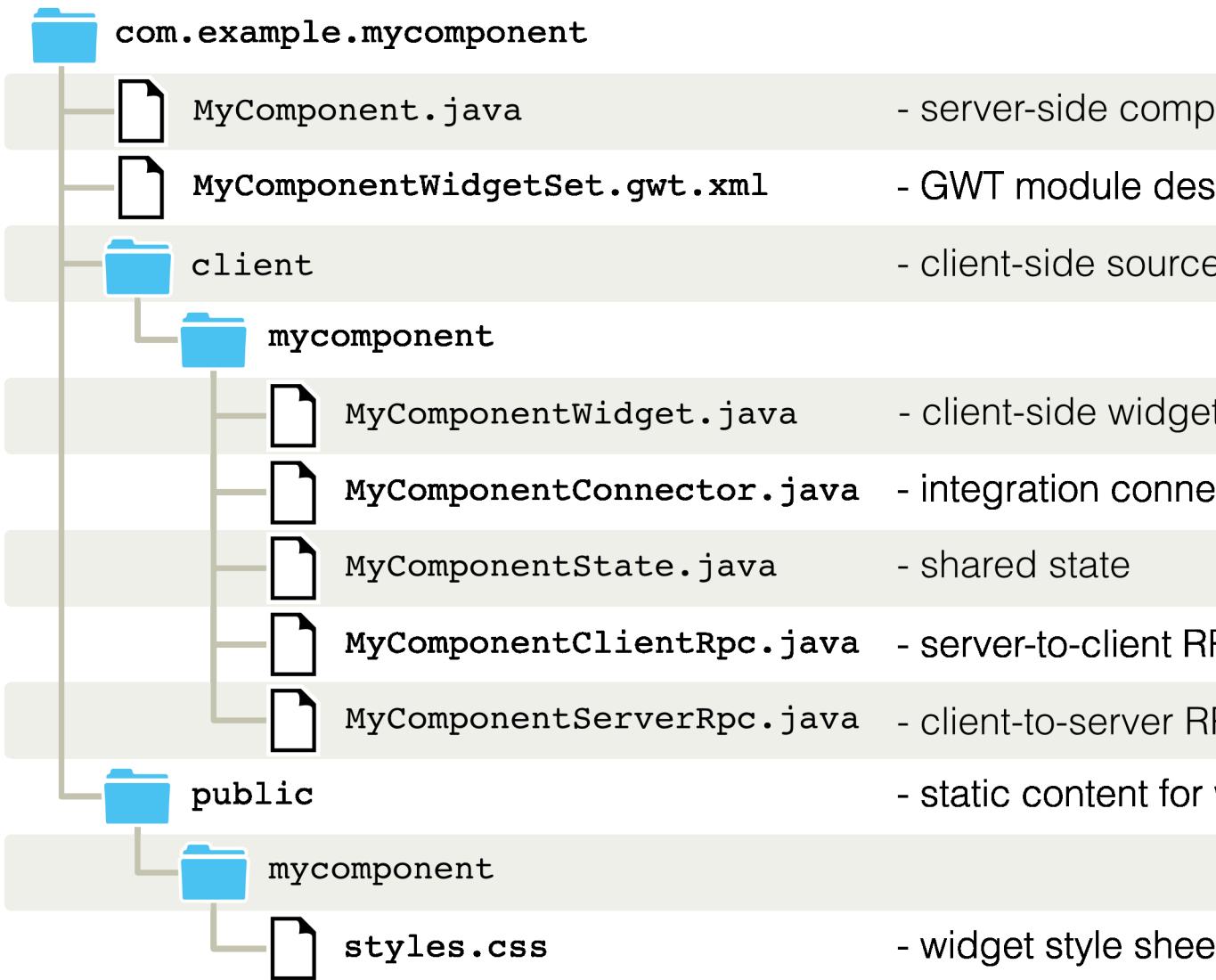
除共享状态信息之外, 服务端与客户端还可以向另一端发起远程过程调用(RPC, remote procedure call). RPC 最初用于事件通知. 比如, 当一个按钮的客户端连接器收到一个点击事件, 它会使用 RPC 将这个事件发送给服务器端.

工程结构

Widget set 的编译(详情请参见 第 13.3 节 “客户端模块描述文件”), 需要使用一种特定的工程结构, 客户端类放在模块描述文件所在包之下的 client 包中. 所有的静态资源, 比如样式表和图片文件, 应该放在 public 文件夹(注意, 不是 Java 包)中. 服务器端组件的源代码可以放在客户端代码包之外的任何位置.

基本的工程结构参见 图 16.2 “Widget 集成工程的基本结构”.

图 16.2. Widget 集成工程的基本结构



Eclipse 向导(详情请参见 第 16.2 节“使用 Eclipse 简化工作”),会帮助你创建 widget 集成工程的框架代码, 结构与上图一样.

JavaScript 组件的集成

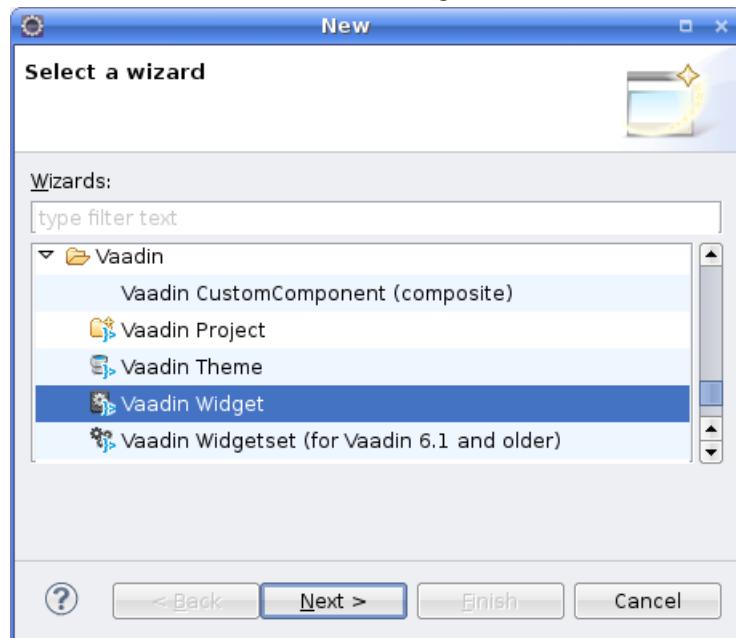
除 GWT widget 集成之外, Vaadin 还提供了简单的方法来集成纯 JavaScript 的组件. JavaScript 连接器代码由服务器端发布. 由于 JavaScript 集成与 GWT 无关, 因此不需要编译 widget set.

16.2. 使用 Eclipse 简化工作

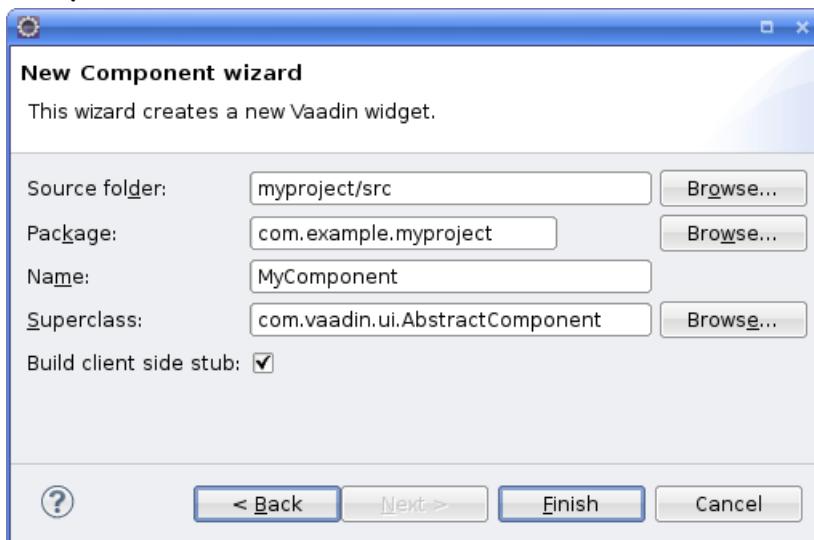
我们首先介绍使用 Eclipse 进行客户端集成的简单方式, 并创建一个简单的组件. 你当然可以使用任意的 IDE 来开发新 widget, 甚至可以不使用 IDE, 但你会发现使用 Eclipse 和 Vaadin Plugin 是非常便利的, 因为它们可以帮助你自动完成 widget 开发中的所有基本工作, 其中最重要的是创建新 widget.

16.2.1. 创建 Widget

1. 在工程上点击鼠标右键, 然后选择菜单项 **New → Other...**
2. 在向导选择窗口中, 选择 **Vaadin → Vaadin Widget**, 然后点击 **Next** 按钮.



3. 在 **New Component Wizard** 窗口中, 进行以下设定.



Source folder

指定整个源代码树的根目录. 默认值是你的工程的默认源代码树目录, 通常不要修改这个值, 除非你的工程结构与一般结构不同.

Package

新建的服务器端组件所属的父包. 如果工程中不存在已有的 widget set, 那么会在这个包之下的 widgetset 子包内创建新的 widget set. 子包中会包含 .gwt.xml 描述

文件, 负责定义 widget set, 在 widgetset.client 子包中还会包含 widget 的根(stub)代码.

Name

新建 服务器端组件 的类名. 客户端 widget 的根代码将会使用相同的名称, 但加上 "-Widget" 后缀, 比如, **MyComponentWidget**. 你可以在将来修改类名.

Superclass

服务器端组件的父类. 默认值是 **AbstractComponent** 类, 但 **com.vaadin.ui.AbstractField** 或 **com.vaadin.ui.AbstractSelect** 也是常用的父类. 如果你从一个已有的组件继承, 你应该选择这个组件作为父类. 这个父类也可以在将来修改.

Template

选择使用哪个模板. 默认值是 **Full fledged**, 这个模板会创建服务器端组件, 客户端 widget, 连接器, 共享的状态对象, 以及 RPC 对象. **Connector only** 选项可以指定不创建共享的状态对象和 RPC 对象.

最后, 点击 **Finish** 按钮, 创建新组件.

向导将会帮助你:

- 在基础包(base package)中创建服务器端组件的根代码
- 如果工程中不存在已有的 widget set, 向导会在基础包中创建一个 GWT 模块描述文件 (.gwt.xml), 并修改 servlet 类, 或修改 web.xml 部署描述文件, 为应用程序指定 widget set 类名参数
- 在基础包之下的 client.componentname 包中, 创建客户端 widget 根代码(以及连接器, 共享的状态对象, 以及 RPC 根代码)

关于服务器端组件和客户端 widget 的结构, 以及组件状态在服务器端与客户端之间的序列化传递, 将在本章后续小节中讲解.

要编译 widget set, 可以点击 Eclipse 工具条上的 **Compile widget set** 按钮. 详情请参见第 16.2.2 节“编译 Widget Set”. 编译完成后, 你就可以运行你的应用程序了, 方法与以前一样, 但这次是使用新的 widget set. 编译结果写入到 WebContent/VAADIN/widgetsets 文件夹下. 如果你需要在 Eclipse 中重编译 widget set, 请参见第 16.2.2 节“编译 Widget Set”. 关于 widget set 编译的详情, 请参见第 13.4 节“编译客户端模块”.

为了使用 widget set, 需要在 web.xml 部署描述文件中插入以下设定内容:

```
<init-param>
    <description>Application widgetset</description>
    <param-name>widgetset</param-name>
    <param-
value>com.example.myproject.widgetset.MyprojectApplicationWidgetset</param-
value>
</init-param>
```

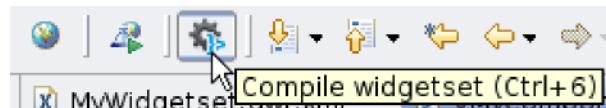
如果有需要, 你随时可以修改包结构, 但 GWT 编译器要求客户端代码所在的包 必须名为 "client", 或者是 widget set 描述文件中 source 元素定义的包.

16.2.2. 编译 Widget Set

编辑过 widget 之后, 你需要编译 widget set. Vaadin Plugin for Eclipse 在很多情况下会自动提示你编译 widget set, 比如当你保存了客户端源代码文件之后. 如果这个自动提示对你造成了干扰, 你可以在工程设置的 Vaadin 分组中选中 **Suspend automatic widgetset builds** 选项, 禁用自动重编译选项.

你可以点击 Eclipse 工具条中的 **Compile widgetset** 按钮来手工编译 widget set. 打开并选中工程后, 这个按钮如图 16.3 “Eclipse 工具条中的 **Compile Widgetset** 按钮” 所示. 如果工程中存在多个 widget set 定义文件, 你需要在 Project Explorer 选中需要编译的对象.

图 16.3. Eclipse 工具条中的 **Compile Widgetset** 按钮



编译进度会显示在 Eclipse 的 **Console** 面板中, 参见 图 16.4 “Widget Set 编译时的输出”. 你需要特别注意类路径中找到的 widget set 列表.

编译的输出路径是 WebContent/VAADIN/widgetsets 文件夹下的 widget set 对应文件夹.

在开发阶段, 你可以只为你的浏览器编译 widgetset, 这样可以显著提高编译速度. 生成的 .gwt.xml 描述文件的根代码中包含了一个被禁用的元素, 这个元素指定编译的目标浏览器. 关于 user-agent 属性的设置, 详情请参见 第 13.3.2 节 “限定编译目标”.

关于 widget set 的更多详细信息, 请参见 第 13.4 节 “编译客户端模块”. 如果你在 Eclipse 环境之外编译了 widget set, 你需要在 **Project Explorer** 中选中工程, 然后按下 **F5** 键来刷新工程信息.

16.3. 创建服务器端组件

通常的服务器端 Vaadin 应用程序使用服务器端组件, 服务器端组件在客户端使用对应的 widget 来展现. 服务器端组件除了实现它的服务器端逻辑之外, 还需要管理服务器端与客户端之间的状态同步问题.

16.3.1. 基本的服务器端组件

组件状态通常使用 共享的状态对象 来管理, 这个问题将在后面的 第 16.5 节 “状态信息共享” 中详细介绍.

```
public class MyComponent extends AbstractComponent {
    public MyComponent() {
        getState().setText("This is MyComponent");
    }

    @Override
    protected MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}
```

图 16.4. Widget Set 编译时的输出

```

Problems Servers Console Progress Error Log
Vaadin Widgetset Compilation
Compiling module com.example.myproject.widgetset.MyWidgetset
Scanning for additional dependencies: jar:file:/home/magi/itmill/workspace/Vaadin%20Project/com.example.myproject.widgetset/MyWidgetset/MyWidgetset.jar!/com/example/myproject/widgetset/MyWidgetset.gwt.xml
Computing all possible rebind results for 'com.vaadin.terminal.gwt.client.WidgetMap'
Rebinding com.vaadin.terminal.gwt.client.WidgetMap
Invoking <generate-with class='com.vaadin.terminal.gwt.widgetsetcompiler.client.WidgetSetGenerator'>
Detecting vaading components in classpath to generate WidgetMap
Widget set will contain implementations for following components
Done. (5seconds)
Compiling 6 permutations
Permutation compile succeeded
Linking into WebContent/VAADIN/widgetsets
Link succeeded
Compilation succeeded -- 79,772s

```

16.4. 使用连接器实现客户端与服务器端的集成

客户端 widget 与服务器端组件的集成是通过 **连接器(connector)** 实现的。连接器是一个客户端类，它负责将服务器端的状态变更发送到客户端 widget，并将客户端事件发送到服务器端。

连接器通常使用 **共享的状态对象** 来得到服务器端组件的状态，这个问题将在后面的第 16.5 节“**状态信息共享**”中详细介绍。

16.4.1. 一个基本的连接器

连接器的基本任务是挂接在 widget 上，处理用户操作导致的事件，以及从服务器端收到的状态变更。此外，连接器还需要实现一些底层基本方法。

```

@Connect (MyComponent.class)
public class MyComponentConnector
    extends AbstractComponentConnector {

    @Override
    public MyComponentWidget getWidget() {
        return (MyComponentWidget) super.getWidget();
    }

    @Override
    public MyComponentState getState() {
        return (MyComponentState) super.getState();
    }

    @Override
    public void onStateChanged(StateChangeEvent stateChangeEvent)
    {
        super.onStateChanged(stateChangeEvent);

        // Do something useful
        final String text = getState().text;
        getWidget().setText(text);
    }
}

```

这里，我们使用一个很粗糙的 `onStateChanged()` 方法来处理状态变更，当组件状态的任何属性发生变化时，这个方法都会被调用。更好更简单的方案是为每一个属性实现一个处理方法，并使用 `@OnStateChange` 来注解这些方法，或者在共享的状态对象的属性上使用 `@DelegateToWidget` 方法，这种方案将在后面的第 16.5 节“状态信息共享”中介绍。

16.4.2. 与服务器端通信

连接器的主要任务是，将用户在 `widget` 上的操作发送给服务器端，以及从服务器端接受状态变更并转发给 `widget`。

服务器端到客户端的通信通常需要使用 共享的状态对象，详情请参见 第 16.5 节“状态信息共享”，此外还需要使用 RPC 调用。状态数据的序列化会完全透明地处理，开发者不必关心。当客户端引擎从服务器端收到状态变更时，它会创建并通知负责管理 `widget` 的连接器。详情请参见 第 16.10.1 节“客户端处理的各个阶段”。

为实现客户端到服务器端的通信，连接器可以向服务器端发起远程过程调用(RPC)。而且，服务器端组件也可以向连接器发起 RPC 调用。关于 RPC 机制的详细介绍，请参见 第 16.6 节“客户端与服务器端之间的 RPC 调用”。

16.5. 状态信息共享

从服务器端组件到它对应的客户端 `widget` 的基本通信是使用 共享的状态信息 实现的。共享的状态信息的序列化处理会透明地实现。在客户端，共享的状态信息应该被看作是只读的，因为它不会被序列化后传回到服务器端。

共享的状态对象只需要简单地继承 `ComponentState` 类。状态对象的成员变量通常应该声明为 `public` 成员。

```
public class MyComponentState extends ComponentState {
    public String text;
}
```

共享的状态对象中不应该包含任何逻辑处理代码。如果因为某种原因，成员变量可见度为 `private`，你可以使用 `public` 的 `setter` 和 `getter` 方法，这时属性本身不可以为 `public`。

16.5.1. 在服务器端访问共享的状态信息

服务器端组件可以使用 `getState()` 方法来访问共享的状态对象。你需要重载基类中的这个方法，在新的实现中应该将共享的状态对象转换为正确的类型，然后再返回转换后的结果，如下：

```
@Override
public MyComponentState getState() {
    return (MyComponentState) super.getState();
}
```

然后你就可以使用 `getState()` 方法，得到正确类型的共享状态信息。

```
public MyComponent() {
    getState().setText("This is the initial state");
    ...
}
```

16.5.2. 在连接器中管理共享的状态信息

连接器可以使用 `getState()` 方法访问共享的状态信息。但这种访问必须是只读的。你需要重载父类的这个方法，在新的实现中为共享状态信息返回正确的数据类型，如下：

```

@Override
public MyComponentState getState() {
    return (MyComponentState) super.getState();
}

```

由服务器端导致的状态变化会被透明地发送到客户端。当发生状态变化时，连接器中的 `onStateChanged()` 方法会被调用。在你执行任何变更处理之前，首先应该调用父类的方法，以便实现组件共通属性的变更处理。

```

@Override
public void onStateChanged(StateChangeEvent stateChangeEvent) {
    super.onStateChanged(stateChangeEvent);

    // Copy the state properties to the widget properties
    final String text = getState().getText();
    getWidget().setText(text);
}

```

上例中的 `onStateChanged()` 方法很粗糙，状态信息中的任何属性发生变更时，这个方法都会被调用，因此你可以实现很复杂的逻辑，根据不同的状态变化来操纵 widget。但是大多数情况下，你可以用更简单而且更有效率的方式来处理属性变更，方法是为每个属性实现单独的变更处理方法，并对这些方法使用 **@OnStateChange** 注解，这种方案的详情将在后面的第 16.5.3 节“使用 **@OnStateChange** 处理属性状态的变更”中介绍，或者你也可以直接将属性值转发给 widget，详情请参见第 16.5.4 节“将状态属性转发给 Widget”。

状态信息变更的处理过程分为几个阶段，详情请参见第 16.10.1 节“客户端处理的各个阶段”。

16.5.3. 使用 **@OnStateChange** 处理属性状态的变更

@OnStateChange 注解可以用来标识连接器的一个方法，将它指定为状态信息中一个属性的变更处理方法，对象属性可以通过注解的参数来指定。除了代码更加清晰之外，这种方法还有一个优点，当状态信息中变更的属性仅有一个或一部分时，可以不必在处理方法中维护所有的属性。但是，如果状态信息的多个属性都可以发生变更，那么只有属性之间不存在相互影响，各个属性的变更处理代码不存在执行顺序的要求时，你才可以使用这种方案。

我们可以将前面的连接器示例代码中的 `onStateChange()` 方法替换为以下代码：

```

@OnStateChange("text")
void updateText() {
    getWidget().setText(getState().text);
}

```

如果在共享的状态信息与 widget 中，属性名称相同，而且不需要任何类型转换，(前面的例子正是这种情况)，这时可以采用更加简单的方案，也就是为状态信息的属性使用 **@DelegateToWidget** 注解，详情请参见第 16.5.4 节“将状态属性转发给 Widget”。

16.5.4. 将状态属性转发给 Widget

在状态信息的属性上使用 **@DelegateToWidget** 注解，可以将状态信息的属性值自动转发到同名同类型的 widget 属性上，Vaadin 会对 widget 调用这个属性对应的 setter 方法。

```

public class MyComponentState extends AbstractComponentState {
    @DelegateToWidget
    public String text;
}

```

上面的代码等价于在连接器内处理状态的变更, 参见 第 16.5.3 节 “使用 **@OnStateChange** 处理属性状态的变更” 中的示例代码.

如果你希望将状态信息的属性转发给 widget 的不同名称的属性, 你可以使用注解的字符串参数来指定属性名称.

```
public class MyComponentState extends AbstractComponentState {  
    @DelegateToWidget("description")  
    public String text;  
}
```

16.5.5. 在共享的状态信息中参照组件

虽然你可以通过共享的状态信息来传递任何常规的 Java 对象, 但如果要参照另一个组件则需要特别的处理, 因为在服务器端你只能参照一个服务器端组件, 而在客户端你只能得到 widget. 对组件的参照可以通过参照它们的连接器来实现(所有的服务器端组件都实现了 Connector 方法).

```
public class MyComponentState extends ComponentState {  
    public Connector otherComponent;  
}
```

然后, 在服务器端你应该使用以下方法访问组件:

```
public class MyComponent {  
    public void MyComponent(Component otherComponent) {  
        getState().otherComponent = otherComponent;  
    }  
  
    public Component getOtherComponent() {  
        return (Component)getState().otherComponent;  
    }  
  
    // And the cast method  
    @Override  
    public MyComponentState getState() {  
        return (MyComponentState) super.getState();  
    }  
}
```

在客户端, 你应该使用类似的方法, 将参照的组件转换为 **ComponentConnector**, 如果你确定地知道连接器类型, 也可以转换为更确定的类型.

16.5.6. 共享资源

资源, (通常是图标或其他图片), 是另一种需要特别处理的情况. Resource 对象只存在于服务器端, 在客户端你只能得到资源的 URL. 在服务器端你需要使用 `setResource()` 和 `getResource()` 方法来访问资源, 资源会被独立地序列化传送到客户端.

我们先来看看服务器端 API:

```
public class MyComponent extends AbstractComponent {  
    ...  
  
    public void setMyIcon(Resource myIcon) {  
        setResource("myIcon", myIcon);  
    }  
  
    public Resource getMyIcon() {
```

```
        return getResource("myIcon");
    }
}
```

然后，在客户端你可以使用 `getResourceUrl()` 方法得到资源的 URL.

```
@Override
public void onStateChanged(StateChangeEvent stateChangeEvent) {
    super.onStateChanged(stateChangeEvent);
    ...

    // Get the resource URL for the icon
    getWidget().setMyIcon(getResourceUrl("myIcon"));
}
```

然后 widget 就可以使用 URL, 如下:

```
public class MyWidget extends Label {
    ...

    Element imgElement = null;

    public void setMyIcon(String url) {
        if (imgElement == null) {
            imgElement = DOM.createImg();
            getElement().appendChild(imgElement);
        }

        DOM.setElementAttribute(imgElement, "src", url);
    }
}
```

16.6. 客户端与服务器端之间的 RPC 调用

Vaadin 支持在服务器端组件和对应的客户端 widget 之间进行远程过程调用(Remote Procedure Call, RPC). 与共享状态信息的变更不同, RPC 调用通常用于传递无状态的事件, 比如按钮的点击, 或其他用户操作事件. 服务器端和客户端都可以向另一端发起 RPC 调用. 当客户端 widget 发起调用时, 会产生向服务器的请求. 由服务器端向客户端发起的调用, 会作为这段代码所属的请求的应答的一部分传递给客户端.(译注: 服务器端代码通常不会主动执行, 它需要由客户端的一次请求来触发, 因此服务器端向客户端发起的调用, 也属于客户端的某个请求所触发的代码的一部分. 服务器端向客户端发起的调用, 会作为这个请求的应答内容的一部分, 传递给客户端.)

如果你使用 Eclipse, 并且在 New Vaadin Widget 向导中选择了 "Full-Fledged" widget, 那么向导会自动创建组件的 RPC 根代码(stub).

16.6.1. 对服务器端的 RPC 调用

从客户端向服务器端发起的 RPC 调用使用 RPC 接口来实现, RPC 接口继承自 `ServerRpc` 接口. 在 Server RPC 接口中, 只简单地定义通过这个接口可以调用的方法.

比如:

```
public interface MyComponentServerRpc extends ServerRpc {
    public void clicked(String buttonName);
}
```

上例定义了一个 `clicked()` RPC 调用, 参数是一个 **MouseEventDetails** 对象.

在 RPC 调用中可以传递大多数标准 Java 类型，比如基本数据类型(int, float 等)及对应的封装类(Integer, Float 等), **String**, 数组, 以及以上类型构成的一部分集合类型(**List**, **Set**, 和 **Map**). 此外, Vaadin 连接器和一些特殊的内部类型也可以使用.

RPC 方法返回值必须为 void - 否则 widget set 编译器会提示错误.

发起调用

在发起调用之前, 你需要使用 `RpcProxy.create()` 方法来初始化 Server RPC 对象. 然后, 就可以通过你定义的 Server RPC 接口来进行调用, 比如:

```
@Connect (MyComponent.class)
public class MyComponentConnector
    extends AbstractComponentConnector {

    public MyComponentConnector() {
        getWidget().addWidgetHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                final MouseEventDetails mouseDetails =
                    MouseEventDetailsBuilder
                        .buildMouseEventDetails(
                            event.getNativeEvent(),
                            getWidget().getElement());
                MyComponentServerRpc rpc =
                    getRpcProxy(MyComponentServerRpc.class);

                // Make the call
                rpc.clicked(mouseDetails.getButtonName());
            }
        });
    }
}
```

处理调用

RPC 调用由 Server RPC 接口的服务器端实现类来处理. 在一次 RPC 请求中, 调用及参数会被序列化, 然后传递给服务器, 这些处理是透明的, 开发者不必关心.

```
public class MyComponent extends AbstractComponent {
    private MyComponentServerRpc rpc =
        new MyComponentServerRpc() {
            private int clickCount = 0;

            public void clicked(String buttonName) {
                Notification.show("Clicked " + buttonName);
            }
        };

    public MyComponent() {
        ...
        registerRpc(rpc);
    }
}
```

16.7. 组件与 UI 扩展

为了给已有的组件添加新的功能, 我们可以选择继承这个组件类, 但带来的问题是不能便利地组合各种不同的功能. 比如, 某个 add-on 可以为 **TextField** 添加拼写检查功能, 另一个 add-on 可以添

加客户端校验功能，想要将这些 add-on 的功能组合起来，是非常困难的，有时甚至是不可能的。有时你可能希望将某个功能添加给几个组件，甚至所有的组件，但以类继承的方式来扩展所有这些组件的功能绝对不是一个好的方案。Vaadin 中包含一种组件插件机制来解决这个问题。这种插件机制就叫做 扩展(Extension)。

UI 也可以使用类似的方式来扩展。Vaadin 的某些功能，比如 JavaScript 执行功能，事实上就是 UI 扩展。

实现一个扩展需要定义一个服务器端扩展类，以及一个客户端连接器。和组件一样，扩展与连接器可以带有共享的状态信息，也可以使用 RPC。

16.7.1. 服务器端扩展 API

对于一个扩展来说，服务器端 API 包括继承自 **AbstractExtension** 的类。这个类通常有一个 `extend()` 方法，一个构造方法（或一个静态的帮助方法），接受的参数是被扩展的组件或 UI。这些参数会被传递给 `super.extend()`。

我们来看看下面的示例程序，它演示了三种不同的 API：

```
public class CapsLockWarning extends AbstractExtension {
    // You could pass it in the constructor
    public CapsLockWarning(PasswordField field) {
        super.extend(field);
    }

    // Or in an extend() method
    public void extend(PasswordField field) {
        super.extend(field);
    }

    // Or with a static helper
    public static addTo(PasswordField field) {
        new CapsLockWarning().extend(field);
    }
}
```

这个扩展可以使用以下方式添加到组件上：

```
PasswordField password = new PasswordField("Give it");

// Use the constructor
new CapsLockWarning(password);

// ... or with the extend() method
new CapsLockWarning().extend(password);

// ... or with the static helper
CapsLockWarning.addTo(password);

layout.addComponent(password);
```

使用这种“反转”的方式来添加新功能在 Vaadin API 中显得稍微有些怪异，但这种方式可以实现扩展的类型安全，因为扩展类中的方法可以限定这个扩展可以使用的目标类型，还可以限定扩展目标是通常的组件还是 UI。

16.7.2. 扩展的连接器

对于扩展来说，在客户端不存在对应的 widget，只有一个扩展连接器，继承自 **AbstractExtensionConnector** 类。和组件的连接器一样，扩展连接器使用 @Connect 注解来标记对应的服务器端扩展类。

扩展的连接器需要实现 extend() 方法，这个方法负责挂接到目标组件上。实现扩展的一般机制是根据需要修改目标组件，并向组件追加事件处理器来响应用户操作。与组件一样，扩展的连接器可以与服务器端扩展类共享状态信息，也可以发起 RPC 调用。

下面的示例中，我们实现一个“Caps Lock 警告”扩展。它监听 Caps Lock 键的状态变化，如果 Caps Lock 键被打开，则在目标组件上显示一个浮动的警告信息。

```
@Connect(CapsLockWarning.class)
public class CapsLockWarningConnector
    extends AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {
        // Get the extended widget
        final Widget pw =
            ((ComponentConnector) target).getWidget();

        // Preparations for the added feature
        final VOverlay warning = new VOverlay();
        warning.setOwner(pw);
        warning.add(new HTML("Caps Lock is enabled!"));

        // Add an event handler
        pw.addDomHandler(new KeyPressHandler() {
            public void onKeyPress(KeyPressEvent event) {
                if (isEnabled() && isCapsLockOn(event)) {
                    warning.showRelativeTo(passwordWidget);
                } else {
                    warning.hide();
                }
            }
        }, KeyPressEvent.getType());
    }

    private boolean isCapsLockOn(KeyPressEvent e) {
        return e.isShiftKeyDown() ^
            Character.isUpperCase(e.getCharCode());
    }
}
```

extend() 方法得到的参数是目标组件的连接器，在上例中是一个 **PasswordFieldConnector**。通过这个连接器的 getWidget() 方法可以得到组件的 widget。

扩展连接器需要包含到 widget set 之内。因此与组件的连接器一样，扩展的连接器类需要定义在 widget set 的 client 包之下。

16.8. Widget 的样式控制

为了让你的 widget 外观漂亮一些，你需要控制它的样式。定义组件的 CSS 样式有两种基本方式：可以在 widget 源代码中定义样式，也可以在 theme 中定义样式。widget 的源代码中应该定义默认样式，各种 theme 可以对默认样式进行不同的修改。

16.8.1. 确定 CSS 样式类

widget 元素的 CSS 类通常在 widget 类中使用 `setStyleName()` 方法来设置。widget 还应该根据自己的需求设置子元素的样式。

比如，你可以首先为一个组合 widget 设置全体样式，然后再为各个子 widget 设置各自的样式，如下：

```
public class MyPickerWidget extends ComplexPanel {
    public static final String CLASSNAME = "mypicker";

    private final TextBox textBox = new TextBox();
    private final PushButton button = new PushButton("...");

    public MyPickerWidget() {
        setElement(Document.get().createDivElement());
        setStylePrimaryName(CLASSNAME);

        textBox.setStylePrimaryName(CLASSNAME + "-field");
        button.setStylePrimaryName(CLASSNAME + "-button");

        add(textBox, getElement());
        add(button, getElement());

        button.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                Window.alert("Calendar picker not yet supported!");
            }
        });
    }
}
```

除此之外，所有的 Vaadin 组件都会带有 `v-widget` 样式类。如果一个 widget 继承已有的 Vaadin 或 GWT widget，那么它也会继承 widget 父类的 CSS 样式。

16.8.2. 默认的样式表文件

客户端模块，(通常是 widget set)，可以包含样式表文件。样式表文件必须放在 widget set 所在文件夹下的 `public` 文件夹之下，详情请参见 第 13.3.1 节“指定样式表”。

比如，对于前面例子中的 widget，你可以使用下面的样式表文件控制它的样式：

```
.mypicker {
    white-space: nowrap;
}

.mypicker-button {
    display: inline-block;
    border: 1px solid black;
    padding: 3px;
    width: 15px;
    text-align: center;
}
```

注意，组件的某些尺寸设置，可能会需要更复杂的控制，也可能需要动态地计算尺寸。

16.9. 组件容器

组件容器，比如布局组件，是一组特殊的组件，需要考虑更多的问题。除了管理自己的状态之外，它们还需要在服务器端和客户端之间传递自己内部组件的层级关系状态。

实现组件容器的最简便方法是继承 **AbstractComponentContainer** 类，这个类会管理组件容器从服务器端到客户端的同步问题。

16.10. 客户端的一些高级问题

下面，我们讨论在 widget 集成时你可能会遇到的一些问题。

16.10.1. 客户端处理的各个阶段

当服务器端状态发生变化时，Vaadin 客户端引擎的反应分为几个不同的阶段，在不同的连接器中，这些阶段的发生顺序可能会不同。处理发生在 **ApplicationConnection** 的 `handleUIDLMessage()` 方法中，但处理逻辑非常复杂，因此我们将各个处理阶段总结如下。

1. 在服务器端类中可以使用 **@JavaScript** 注解或 **@StyleSheet** 注解来定义的依赖的 JavaScript 或样式表，这一阶段首先装载这些资源，直到浏览器确认这些资源装载完毕，处理才会继续执行。
2. 初始化新的连接器，并对每个 `Connector` 执行 `init()` 方法。
3. 更新状态对象，但还不会激发状态变更事件。
4. 更新连接器层级关系，但还不会激发层级关系变更事件。在这个阶段会执行 `setParent()` 方法和 `setChildren()` 方法。
5. 这个阶段会激发层级关系变更事件。也就是说此时所有的状态对象和完整的层级关系已经更新完毕。当所有的层级关系变更事件处理完毕之后，理论上来说 DOM 元素的层级关系应该更新完毕，但是某些内建组件出于各种理由，不一定会保证这一点。
6. 更新标签，根据需要会在布局上调用 `updateCaption()` 方法。
7. 对于所有使用 **@DelegateToWidget** 注解的状态对象属性，处理相关的变更。
8. 对于有变更的状态对象，激发状态变更事件。
9. 对于旧版本遗留的连接器，调用 `updateFromUIDL()`。
10. 调用从服务器端收到的所有 RPC 方法。
11. 注销不再包含在层级关系中的连接器。这个过程会调用 `Connector` 的 `onUnregister()` 方法。
12. 从这里开始进入布局管理阶段，首先检查所有元素的尺寸和位置，然后向所有的 `ElementResizeListener` 发送通知，并对实现 **SimpleManagedLayout** 或 **DirectionalManagedLayout** 接口的连接器，调用适当的布局方法。

16.11. 创建 Add-on

不论是商业版还是免费版，Add-on 是重用 Vaadin 代码的最便利方法。Vaadin Directory 是 add-on 的下载中心。你可以使用 JAR 库或 Zip 包形式发布 add-on。

创建一个通常的 add-on 包，包括以下几个步骤：

- 编译服务器端类
- 编译 JavaDoc(可选)
- Build JAR 文件
 - 包含 Vaadin add-on manifest 文件
 - 包含编译后的服务器端类文件
 - 包含编译后的 JavaDoc (可选)
 - 包含客户端类源代码，以便编译 widget set(可选)
 - 包含所有依赖的 JavaScript 库(可选)
 - 不要包含工程中的所有测试代码或示例代码

具体的内容取决于 add-on 的类型。组件 add-on 通常包含 widget set，但也不是永远如此，比如 JavaScript 组件或纯服务器端组件就没有 widget set。你也开发数据容器 add-on 和 theme add-on，以及各种工具 add-on。

通常使用独立的 JAR 来发布 JavaDoc，但也可以将 JavaDoc 包含在同一个 JAR 中。

16.11.1. 在 Eclipse 中导出 Add-on

如果你对你的 add-on 工程使用 Vaadin Plugin for Eclipse，你可以简单地从 Eclipse 中导出 add-on.

1. 选中工程，然后在菜单中选择 **File → Export**
2. 在 export 向导中，选择 **Vaadin → Vaadin Add-on Package**，然后点击 **Next** 按钮
3. 在 **Select the resources to export** 窗口中，选择将包含在 add-on 包中的内容。通常，应该包含 `src` 文件夹中的源代码（至少需要客户端包中的源代码），编译后的服务器端类，`WebContent/VAADIN/themes` 文件夹中的 theme。这些内容都是自动包含的。你可能会希望从输出内容中删除示例代码。

如果你希望将 add-on 发布到 Vaadin Directory，**Implementation title** 项目中应该输入 add-on 在 Directory 中的名称。名称中可以包含空白或其他大多数字符。注意，名称在发布后无法修改。

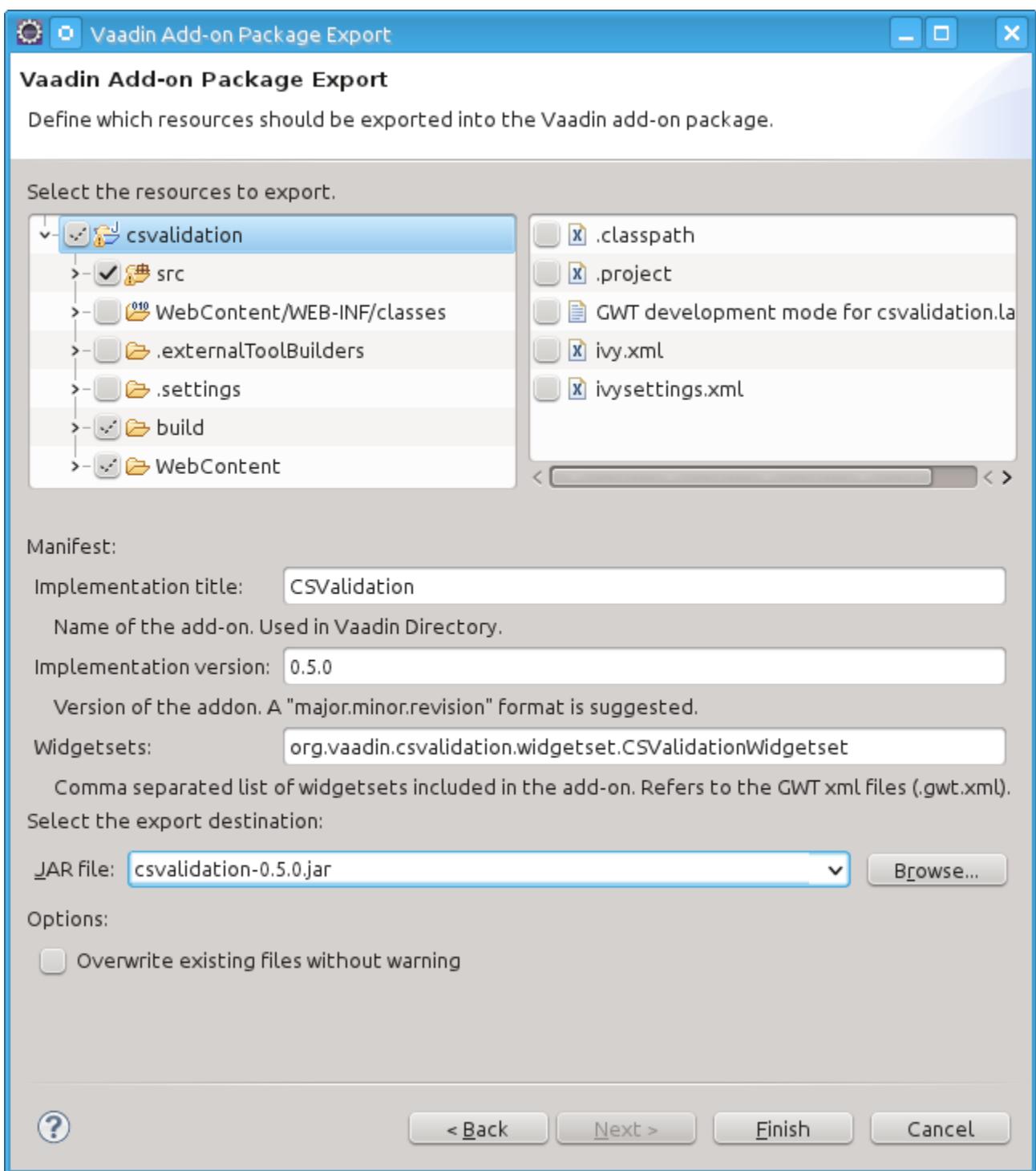
Implementation version 项目是你的 add-on 的版本号。通常试验版或 beta 版从 0.1.0 开始，稳定版从 1.0.0 开始。

Widgetsets 项目应该列出 add-on 中包含的所有 widget set，如果有多个，使用逗号分隔。列出 widget set 时应该使用类名，也就是说，不要带 `.gwt.xml` 扩展名。

JAR file 项目是输出的 JAR 文件名，通常应该包含 add-on 的版本号。你应该遵守 Maven 的命名规约，比如：`myaddon-1.0.0.jar`。

最后，点击 **Finish** 按钮。

图 16.5. 导出一个 Vaadin Add-on



16.11.2. 使用 Ant 构建 Add-on

使用 Ant 来构建 add-on 的方法与构建 Vaadin 应用程序类似。Vaadin 库及其他依赖的库会通过 Apache Ivy 自动取得，并包含到类路径中。

下面的示例中, 我们假定工程结构与前面的 Eclipse 工程示例相同. 我们将构建脚本放在工程根目录下的 build 文件夹中. 我们首先创建以下 Ant 脚本:

```
<?xml version="1.0"?>

<project xmlns:ivy="antlib:org.apache.ivy.ant"
    name="My Own add-on"
    basedir=".."
    default="package-jar">
```

在 Ant 1.6 或更高版本中使用 Ivy, 只需要声明相关的 namespace 即可. 对于更老的 Ant 版本, 请参见 Ivy 文档.

配置与初始化

在构建脚本示例中, 我们将大多数配置放在 configure 目标(target)中, 然后在 init 目标中初始化构建环境.

```
<target name="configure">
    <!-- Where project source files are located -->
    <property name="src-location" value="src" />

    <!-- Name of the widget set. -->
    <property name="widgetset"
        value="com.example.myaddon.widgetset.MyAddonWidgetset" />

    <!-- Addon version -->
    <property name="version" value="0.1.0" />

    <!-- Compilation result directory -->
    <property name="result-dir" value="build/result"/>

    <!-- The target name of the built add-on JAR -->
    <property name="target-jar"
        value="${result-dir}/myaddon-${version}.jar" />
</target>

<target name="init" depends="configure">
    <!-- Construct and check classpath -->
    <path id="compile.classpath">
        <pathelement path="build/classes" />
        <pathelement path="${src-location}" />
        <fileset dir="${result-dir}/lib">
            <include name="*.jar" />
        </fileset>
    </path>

    <mkdir dir="${result-dir}" />
</target>
```

除 configure 目标之外, 在 package-jar 目标中也需要进行一些配置.

编译服务器端代码

编译 add-on 需要使用 Vaadin 库和其他依赖库. 我们使用 Apache Ivy 来解析库之间的依赖关系, 并自动取得这些库的 JAR 文件.

```
<!-- Retrieve dependencies with Ivy -->
<target name="resolve" depends="init">
    <ivy:retrieve
        pattern="${result-dir}/lib/[artifact].[ext]"/>
</target>
```

<retrieve> 任务(task)的 pattern 属性指定依赖库取得之后存储在什么位置, 上面的例子中使用的是 build/result/lib 目录.

然后来编译服务器端类, 这个任务非常简单:

```
<!-- Compile server-side -->
<target name="compile-server-side"
    depends="init, resolve">
    <delete dir="${result-dir}/classes"/>
    <mkdir dir="${result-dir}/classes"/>

    <javac srcdir="${src-location}"
        destdir="${result-dir}/classes">
        <classpath>
            <path refid="compile.classpath"/>
        </classpath>
    </javac>
</target>
```

编译 JavaDoc

你可能会希望将 add-on 的 API 文档包含在同一个或者另一个 JAR 文件中. 你可以使用我们前面定义的配置, 通过以下方法实现. 如果源代码树中包含了客户端类以及测试和示例类, 你可能会希望在 JavaDoc 中排除它们, 下面的示例中就是这样做的.

```
<!-- Compile JavaDoc -->
<target name="compile-javadoc" depends="init">
    <delete dir="${result-dir}/javadoc"/>
    <mkdir dir="${result-dir}/javadoc"/>

    <javadoc destdir="${result-dir}/javadoc">
        <sourcefiles>
            <fileset dir="${src-location}" id="src">
                <include name="**/*.java"/>

                <!-- Excluded stuff from the package -->
                <exclude name="**/client/**/*"/>
                <exclude name="**/demo/**/*"/>
                <exclude name="**/MyDemoUI.java"/>
            </fileset>
        </sourcefiles>
        <classpath>
            <path refid="compile.classpath"/>
        </classpath>
    </javadoc>
</target>
```

打包 JAR 文件

一个 add-on 的 JAR 文件通常包含以下内容:

- Vaadin add-on manifest 文件

- 编译后的服务器端类文件
- 编译后的 JavaDoc (可选)
- 客户端类的源代码文件 (可选)
- 所有依赖的 JavaScript 库 (可选)

我们首先来编写 Ant 的构建目标. JAR 文件需要编译后的服务器端类, 以及 API 文档(可选).

```
<!-- Build the JAR -->
<target name="package-jar"
       depends="compile-server-side, compile-javadoc">
    <jar jarfile="${target-jar}" compress="true">
```

第一步, 你需要包含 manifest, 它负责定义 add-on 的基本信息. implementation title 项目必须设置为 add-on 的标题, 和在 Vaadin Directory 中显示的标题一致. vendor 项目是你自己. manifest 中还包括 add-on 的 license title 和 license file 参照地址.

```
<!-- Manifest required by Vaadin Directory -->
<manifest>
    <attribute name="Vaadin-Package-Version"
              value="1" />
    <attribute name="Vaadin-Widgetsets"
              value="${widgetset}" />
    <attribute name="Implementation-Title"
              value="My Own Addon" />
    <attribute name="Implementation-Version"
              value="${version}" />
    <attribute name="Implementation-Vendor"
              value="Me Myself" />
    <attribute name="Vaadin-License-Title"
              value="Apache2" />
    <attribute name="Vaadin-License-File"
              value="http://www.apache.org/licenses/LICENSE-2.0" />
</manifest>
```

package-jar 目标的其余部分如下. 和编译 JavaDoc 时一样, 这里你也需要排除工程中的测试和示例代码. 针对你的工程的具体结构, 你至少需要修改下例中粗体字的部分.

```
<!-- Include built server-side classes -->
<fileset dir="build/result/classes">
    <patternset>
        <include name="com/example/myaddon/**/*"/>
        <exclude name="**/client/**/*"/>
        <exclude name="**/demo/**/*"/>
        <exclude name="**/test/**/*"/>
        <exclude name="**/MyDemoUI*"/>
    </patternset>
</fileset>

<!-- Include widget set sources -->
<fileset dir="src">
    <patternset>
        <include name="com/example/myaddon/**/*"/>
    </patternset>
</fileset>

<!-- Include JavaDoc in the JAR -->
```

```
<fileset dir="${result-dir}/javadoc"
    includes="**/*"/>
</jar>
</target>
```

到这里，你应该可以使用 Ant 来运行构建脚本了。

16.12. 从 Vaadin 6 迁移

在 Vaadin 7 中，客户端架构几乎完全重新设计过了。在 Vaadin 6 中，状态同步是通过在服务器端和客户端显示地序列化/反序列化来实现的。在 Vaadin 7 中，序列化是 Vaadin 框架使用状态对象自动处理的。

在 Vaadin 6 中，服务器端组件会使用客户端的 `Paintable` 接口，将它的状态序列化到客户端，并使用 `VariableOwner` 接口来反序列化状态信息。在 Vaadin 7 中，这些工作都使用 `ClientConnector` 接口来实现。

在客户端，widget 使用 `Paintable` 接口反序列化它的状态，并通过 `ApplicationConnection` 对象发送状态变更。在 Vaadin 7 中，这些工作被 `ServerConnector` 替代了。

除状态信息的同步方式之外，Vaadin 7 还有 RPC 机制可用来传送事件。这种机制对于与状态变更无关的事件尤其有用，比如按钮的点击事件。

当监听器被调用时，Vaadin 框架会确保连接器层级关系与状态信息都是最新的。

16.12.1. 快速(而且肮脏)地迁移

Vaadin 7 中带有一个兼容层，可以用来快速转换一个 widget。

1. 创建一个连接器类，比如 `MyConnector`，继承自 `LegacyConnector`。实现 `getWidget()` 方法。
2. 将 `@ClientWidget(MyWidget.class)` 注解，从服务器端组件，（假定是 `MyComponent`），移动到 `MyConnector` 类中，并修改为 `@Connect(MyComponent.class)`。
3. 让服务器端组件实现 `LegacyComponent` 接口，以便允许 Vaadin 进行旧版本兼容处理。
4. 删除所有对 `super.paintContent()` 的调用
5. 在客户端代码中更新所有的 import 语句

16.13. JavaScript 组件与扩展的集成

除组件和 UI 扩展外，Vaadin 还可以简单地集成纯 JavaScript 组件。JavaScript 连接器代码由服务器端发布。由于 JavaScript 集成与 GWT 无关，因此不需要编译 widget set。

16.13.1. JavaScript 库示例

JavaScript 组件库有很多类型。下面的例子中，我们将演示一个简单的库，它提供一个面向对象的 JavaScript 组件。后面我们会使用这个例子来演示如何将它与服务器端 Vaadin 组件集成。

例子库中包含一个 `MyComponent` 组件，定义在 `mylibrary.js` 中。

```
// Define the namespace
var mylibrary = mylibrary || {};

mylibrary.MyComponent = function (element) {
    element.innerHTML =
        "<div class='caption'>Hello, world!</div>" +
        "<div class='textinput'>Enter a value: " +
        "<input type='text' name='value' />" +
        "<input type='button' value='Click' />" +
        "</div>";

    // Style it
    element.style.border = "thin solid red";
    element.style.display = "inline-block";

    // Getter and setter for the value property
    this.getValue = function () {
        return element.
            getElementsByTagName("input") [0].value;
    };
    this.setValue = function (value) {
        element.getElementsByTagName("input") [0].value =
            value;
    };

    // Default implementation of the click handler
    this.click = function () {
        alert("Error: Must implement click() method");
    };

    // Set up button click
    var button = element.getElementsByTagName("input") [1];
    var self = this; // Can't use this inside the function
    button.onclick = function () {
        self.click();
    };
}
```

在 HTML 页面中使用时，应该使用以下语句包含库文件：

```
<script type="text/javascript"
       src="mylibrary.js"></script>
```

然后你就可以在 HTML 文档的任何地方使用这个组件，如下：

```
<!-- Placeholder for the component -->
<div id="foo"></div>

<!-- Create the component and bind it to the placeholder -->
<script type="text/javascript">
    window.foo = new mylibrary.MyComponent(
        document.getElementById("foo"));
    window.foo.click = function () {
        alert("Value is " + this.getValue());
    }
</script>
```

图 16.6. JavaScript 组件示例



你可以通过 JavaScript 来与这个组件交互，比如：

```
<a href="javascript:foo.setValue('New value')">Click here</a>
```

16.13.2. 供 JavaScript 组件使用的服务器端 API

为了与这样一个 JavaScript 组件集成，你首先需要规划一下它在服务器端 Vaadin 应用程序中应该如何使用。组件应该支持值的写操作，还要能够监听值的变化。

```
final MyComponent mycomponent = new MyComponent();

// Set the value from server-side
mycomponent.setValue("Server-side value");

// Process a value input by the user from the client-side
mycomponent.addValueChangeListener(
    new MyComponent.ValueChangeListener() {
        @Override
        public void valueChange() {
            Notification.show("Value: " + mycomponent.getValue());
        }
    });
}

layout.addComponent(mycomponent);
```

基本的服务器端组件

JavaScript 组件继承自 **AbstractJavaScriptComponent**，这个类会为组件管理共享的状态对象，以及 RPC 调用。

```
package com.vaadin.book.examples.client.js;

@JavaScript({"mylibrary.js", "mycomponent-connector.js"})
public class MyComponent extends AbstractJavaScriptComponent {
    public interface ValueChangeListener extends Serializable {
        void valueChange();
    }
    ArrayList<ValueChangeListener> listeners =
        new ArrayList<ValueChangeListener>();
    public void addValueChangeListener(
        ValueChangeListener listener) {
        listeners.add(listener);
    }

    public void setValue(String value) {
        getState().value = value;
    }

    public String getValue() {
        return getState().value;
    }
}
```

```
    @Override
    protected MyComponentState getState() {
        return (MyComponentState) super.getState();
    }
}
```

注意, 本节后续部分中创建的 JavaScript 连接器, 它的名称必须与这个服务器端类的包名一致.

组件的共享状态信息如下:

```
public class MyComponentState extends JavaScriptComponentState {
    public String value;
}
```

如果成员变量是 private 的, 那么你需要为它定义 public 的 setter 和 getter 方法, 然后在组件中使用这些 public 方法.

16.13.3. 定义一个 JavaScript 连接器

JavaScript 连接器是一个函数, 它负责初始化 JavaScript 组件, 并处理服务器端与 JavaScript 代码之间的通信.

连接器定义为一个连接器初始化函数, 这个函数被添加到 window 对象中. 函数名必须与服务器端类名一致, 其中要包含完整的包路径. 但包名中的分隔符不使用 Java 的点号规约, 而改用下划线作为分隔符.

Vaadin 客户端框架会向连接器函数添加很多方法. this.getElement() 方法将返回组件的 HTML DOM 元素. this.getState() 方法返回共享的状态对象, 其中的状态信息将与服务器端保持同步.

```
window.com_vaadin_book_examples_client_js_MyComponent =
function() {
    // Create the component
    var mycomponent =
        new mylibrary.MyComponent(this.getElement());

    // Handle changes from the server-side
    this.onStateChange = function() {
        mycomponent.setValue(this.getState().value);
    };

    // Pass user interaction to the server-side
    var self = this;
    mycomponent.click = function() {
        self.onClick(mycomponent.getValue());
    };
};
```

上例中, 我们使用 JavaScript RPC 机制来传递用户操作, 详情将在下一节中介绍.

16.13.4. 从 JavaScript 到服务器端的 RPC 调用

用户对 JavaScript 组件的操作必须使用一种 RPC (Remote Procedure Call) 机制传递到服务器端. JavaScript RPC 机制基本上等同于通常的客户端 widget, 详情请参见 第 16.6 节 “客户端与服务器端之间的 RPC 调用”.

在服务器端处理 **RPC** 调用

我们首先来看看服务器端的 RPC 函数注册。RPC 调用在服务器端由函数处理器负责处理，函数处理器实现 `JavaScriptFunction` 接口。服务器端的一个函数处理器使用 `AbstractJavaScriptComponent` 的 `addFunction()` 方法来注册。服务器端的注册实际上会定义一个 JavaScript 方法，这个方法将供客户端连接器使用。

下面继续看我们前面的服务器端 `MyComponent` 示例，我们现在给它添加一个构造函数，其中会注册 RPC 函数。

```
public MyComponent() {
    addFunction("onClick", new JavaScriptFunction() {
        @Override
        public void call(JSONArray arguments)
            throws JSONException {
            getState().setValue(arguments.getString(0));
            for (ValueChangeListener listener: listeners)
                listener.valueChange();
        }
    });
}
```

从 **JavaScript** 发起 **RPC** 调用

发起一个 RPC 调用只需要简单地调用连接器中的 RPC 方法。在 JavaScript 连接器的构造函数中，你可以编写以下代码（完整的连接器代码请参见前面的示例）：

```
window.com_vaadin_book_examples_gwt_js_MyComponent =
function() {
    ...
    var connector = this;
    mycomponent.click = function() {
        connector.onClick(mycomponent.getValue());
    };
};
```

在这里，`mycomponent.click` 是 JavaScript 示例库中的一个函数，详情请参见 第 16.13.1 节“`JavaScript 库示例`”。`onClick()` 是我们在服务器端定义的方法，在这个调用中我们传递了一个简单的字符串参数。

在参数中，你可以传递 JSON 中合法的任意类型数据。

Vaadin Add-on

17.1. 概述	447
17.2. 通过 Vaadin Directory 下载 Add-on	448
17.3. 在 Eclipse 中使用 Ivy 安装 Add-on	448
17.4. 在 Maven 工程中使用 Add-on	450
17.5. 问题诊断	453

本章介绍如何通过 Vaadin Directory 安装 add-on 组件, theme, 数据容器, 以及其他工具, 此外还介绍如何使用 Vaadin 提供的商业 add-on.

17.1. 概述

除 Vaadin 核心库中内建的组件, 布局, theme, 数据源之外, 还存在其他很多 add-on 可供使用. Vaadin Directory 提供了大量的 Vaadin add-on, 当然你还可以从别的独立来源发现其他 add-on. Add-on 也是一种很好的方式, 可以在多个工程之间共享你自己的组件.

通过 Vaadin Directory 安装 add-on 是很简单的, 只需要在 Ivy 或 Maven 中添加依赖关系即可, 也可以下载 JAR 包然后将它放在工程的 web library 文件夹中. 大多数 add-on 都包含 widget set, 你需要编译它, 但通常并不困难, 只需要点击一下编译按钮, 或者执行一个简单的命令就可以完成.

试用过某个 add-on 之后, 你可以给这个 add-on 评分(可选则 1 到 5 星), 还可以留下评论, 将你的意见反馈给 add-on 作者. 大多数 add-on 还有一个讨论组, 可供使用者反馈意见, 或提出问题.

Vaadin Directory 中的 Add-on 发布时使用各种不同的许可协议, 有一部分 add-on 使用的是商业版许可协议. 虽然 add-ons 可以直接下载, 但你应该注意它们的许可协议, 以及其他相关的法律条款和限制条件. 一部分 add-on 使用双许可证发布, 在开源项目中可以免费使用, 还有一部分 add-on 允许在闭源开发中使用, 但设置了试用期限.

17.2. 通过 Vaadin Directory 下载 Add-on

如果你没有使用 Maven 兼容的依赖管理工具，或者你希望手动管理库文件，你可以从 Vaadin Directory 中的 add-on 详细信息页面下载包文件。

1. 选择版本；某些 add-on 同时存在多个版本。默认显示的是最新版，但你也可以从 add-on 详细信息页面上方的下拉菜单选择其他版本来下载。
 2. 点击 **Download Now** 按钮，将 JAR 文件或 Zip 文件保存到你的计算机上。
 3. 如果 add-on 包是 Zip 格式，你需要解开 Zip 包，并遵照包内提供的指示进行安装。通常，你只需要将一个 JAR 文件复制到你的 Web 工程的 WEB-INF/lib 目录中。
- 注意，某些 add-on 可能会依赖其他库。你可以手动解析这些依赖关系，但我们推荐在你的工程中使用依赖管理工具，比如 Ivy 或 Maven。
4. 更新你的工程，并重新编译。在 Eclipse 中，需要选中工程，然后按 F5 键。
 5. 你可能会需要编译 add-on 组件的客户端代码，也就是，*widget set*。对大多数 add-on 都是如此，但纯服务器端组件，theme，或数据绑定 add-on 除外。编译 widget set 的方法取决于构建环境。关于如何使用 Eclipse 和 Maven 来编译 widget set，请参见 第 17.2.1 节“使用 Ant 脚本编译 Widget Set”，本章后续部分也会介绍这个问题。
 6. 在你的开发用 Web 服务器上更新工程，这时有可能需要重启服务器。

17.2.1. 使用 Ant 脚本编译 Widget Set

如果你需要使用 Ant 脚本编译 widget set，你可以在以下地址找到脚本模板包：Vaadin download page。你可以将包内的文件复制到你的工程中，配置完成之后，就可以在它所在的目录运行 Ant 了。

如果你使用 IDE，比如 Eclipse，在 IDE 环境之外编译 widget set 之后，一定要记得刷新工程，以保证工程内容与文件系统内容保持一致。

17.3. 在 Eclipse 中使用 Ivy 安装 Add-on

Vaadin Plugin for Eclipse 使用 Apache Ivy 来解析库的依赖关系。依赖的库应该在工程根路径下的 ivy.xml 文件中列出。Vaadin Directory 允许通过 Maven repository 下载 add-on，Ivy 也可以访问这个 Maven repository。

你也可以在 Ant 脚本中使用 Ivy 来解析包依赖关系。

1. 在 Vaadin Directory 中打开 add-on 的页面。
2. 选择版本。默认显示的是最新版，但你也可以在 add-on 详细页面上部的下拉菜单中选择其他版本。
3. 点击 **Maven/Ivy** 按钮，显示 Ivy 依赖声明，见图 17.1。如果 add-on 有多种许可协议，系统会提示你为这个依赖选择一个许可协议。

图 17.1. Ivy 依赖声明



- 在你的 Eclipse 工程中, 使用 XML 编辑器或 Ivy 编辑器打开 ivysettings.xml 文件 (可以鼠标双击这个文件, 也可以右键点击它, 然后选择 **Open With → Ivy Editor**).

检查设定文件中是否存在 Vaadin Directory 页面中显示的那个 `<ibiblio>` 标签. 如果文件是由 Eclipse 的 Vaadin project wizard 创建的, 那么它应该有这个标签. 如果没有, 请将这个标签复制到设定文件中.

```

<chain name="default">
  ...
  <ibiblio name="vaadin-addons"
    usepoms="true"
    m2compatible="true"
    root="http://maven.vaadin.com/vaadin-addons" />
  ...
</chain>

```

如果你使用其他 repository 来获取 Vaadin addon, 比如使用你自己编译生成的本地 repository, 你需要在设定文件中为 repository 定义一个解析器.

- 在你的 Eclipse 工程中打开 ivy.xml 文件, 将 Ivy 依赖声明复制到 dependencies 标签中. 完成后的内容应该如下:

```

<dependencies>
  ...
  <dependency org="com.vaadin.addon"
    name="vaadin-charts"
    rev="1.0.0" />
</dependencies>

```

你可以指定一个固定的版本号, 也可以指定一个动态的版本标签, 比如 `latest.release`. 关于 依赖声明 的更多详细信息, 请参见 Ivy 文档.

如果 `ivy.xml` 文件中没有定义 `<configurations defaultconfmapping="default->default">`, 那么你还需要在依赖中定义 `conf="default->default"`, 才能正确地解析临时依赖(transient dependency).

保存设定文件后, IvyDE 会立即解析包依赖关系.

6. 点击工具栏中的 **Compile Vaadin widgets** 按钮, 编译 add-on 的 widget set.

图 17.2. 在 Eclipse 中编译 Widget Set



如果你在使用 Ivy 时遇到了问题, 首先请检查所有的依赖参数. IvyDE 有时可能会发生难以预料的问题. 你可以清空 Ivy 的依赖关系缓存, 方法是在工程上点击鼠标右键, 然后选择菜单项 **Ivy → Clean all caches**. 要刷新 Ivy 配置, 请选择菜单项 **Ivy → Refresh**. 要重新解析 Ivy, 请选择菜单项 **Ivy → Resolve**.

17.4. 在 Maven 工程中使用 Add-on

要在 Maven 工程中使用 add-on, 你只需要将 add-on 作为依赖添加到工程的 POM 中即可. 大多数 add-ons 会包含 widget set, 需要编译到工程的 widget set 中.

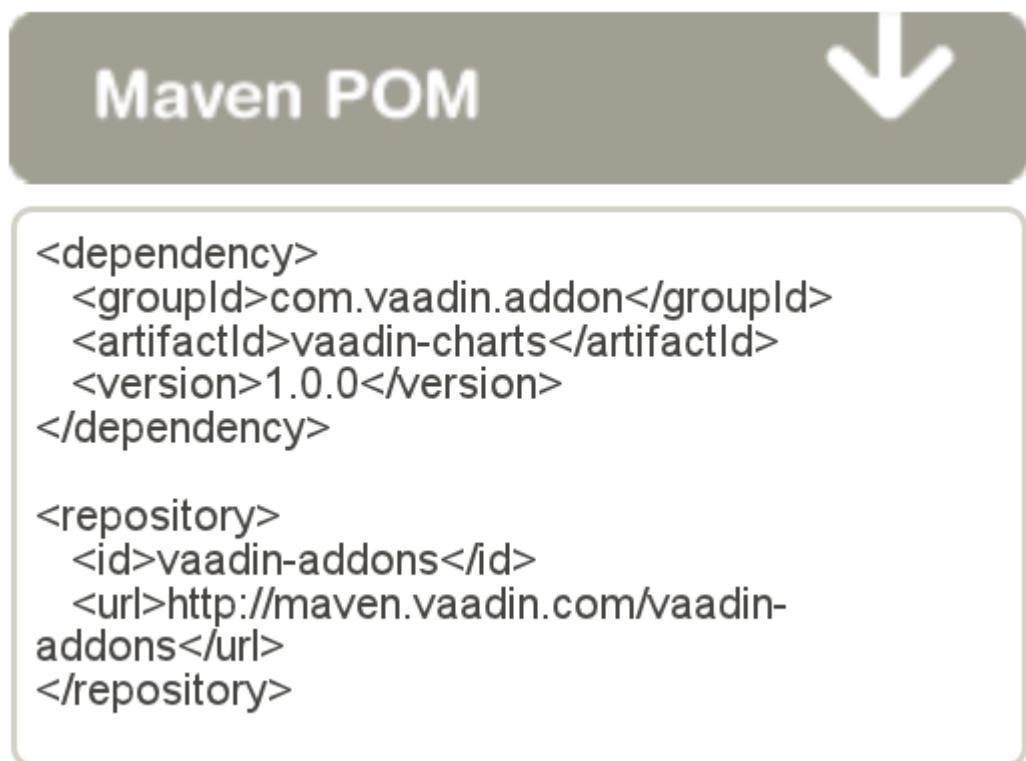
如何使用 Maven 来创建, 编译, 并打包 Vaadin 工程, 请参见 第 2.6 节 “通过 Maven 使用 Vaadin”.

17.4.1. 添加依赖

Vaadin Directory 为 Directory 中的所有 add-on 提供了一个 Maven repository.

1. 在 Vaadin Directory 中打开 add-on 的页面.
2. 选择版本. 默认显示的是最新版, 但你也可以在 add-on 详细页面上部的下拉菜单中选择其他版本.
3. 点击 **Maven/Ivy** 按钮, 显示 Maven 依赖声明, 见图 17.3. 如果 add-on 有多种许可协议, 系统会提示你为这个依赖选择一个许可协议.

图 17.3. Maven POM 定义



4. 将这段 声明复制到你工程的 pom.xml 文件的 dependencies 标签中.

```

...
<dependencies>
  ...
    <dependency>
      <groupId>com.vaadin.addon</groupId>
      <artifactId>vaadin-charts</artifactId>
      <version>1.0.0</version>
    </dependency>
</dependencies>

```

你可以使用固定的版本号, 就像上面的例子一样, 也可以使用 LATEST 标记, 永远使用这个 add-on 的最新版本.

Vaadin Directory 中显示的 POM 片段还包含一个 repository 定义, 如果你创建工程时使用的是 vaadin-archetype-application, 那么应该已经包含了这段 repository 定义.

5. 编译 widget set, 方法见下节.

17.4.2. 编译工程的 Widget Set

如果你使用 vaadin-archetype-application 来创建工程, 那么 pom.xml 文件将包含编译 widget set 所需要的所有声明. widget set 的编译发生在标准的 Maven 构建阶段, 比如使用 package 或 install 目标来进行构建时.

```
$ mvn package
```

然后, 只需要将 WAR 发布到你的应用程序服务器即可.

重编译 Widget Set

除非必要, 否则 Vaadin plugin for Maven 会尽量避免重新编译 widget set, 所以有时候虽然应该重编译, 但实际上不会重编译. 运行 clean 目标通常可以解决这个问题, 但会导致完全的重编译. 你可以运行 vaadin:compile 目标来手动编译 widget set.

```
$ mvn vaadin:compile
```

注意, 从类路径中查找新的 widget set 不会更新工程 widget set. 必须添加或删除 add-on 才会更新. 你可以在工程中运行 vaadin:update-widgetset 目标来更新工程 widget set. (译注: 本段不太理解, 待校)

```
$ mvn vaadin:update-widgetset
...
[INFO] auto discovered modules [your.company.gwt.ProjectNameWidgetSet]
[INFO] Updating widgetset your.company.gwt.ProjectNameWidgetSet
[ERROR] 27.10.2011 19:22:34
com.vaadin.terminal.gwt.widgetsetutils.ClassPathExplorer getAvailableWidgetSets
[ERROR] INFO: Widgets found from classpath:
...

```

不必在意这里的 "ERROR", 这些是 Vaadin Plugin for Maven 本身错误.

运行完 update 之后, 你需要运行 vaadin:compile 目标, 编译 widget set.

17.4.3. 启动 Widget Set 编译功能

如果你没有使用适当的 Vaadin archetype 创建的 POM 文件, 你可能需要手工启动 widget set 编译. 最简单的方法是从使用 Vaadin archetype 创建的 POM 文件中复制相关的定义. 尤其是, 你需要复制 plugin 定义. 此外还需要 Vaadin 依赖.

你需要创建一个新的 widget set 定义文件, widget set 编译时会将类路径中发现的 widget set 添加到这个文件.(在工程的包中)创建 src/main/java/com/example/AppWidgetSet.gwt.xml 文件, 其中包含一个空的 <module> 标签, 如下:

```
<module>
</module>
```

在 UI 中启用 Widget Set

如果你以前在工程中使用过默认的 widget set, 你需要在 web.xml 部署描述文件中启用工程 widget set. 编辑 src/main/webapp/WEB-INF/web.xml 文件, 添加或修改 servlet 的 widgetset 参数, 如下.

```
<servlet>
...
<init-param>
    <description>Widget Set to Use</description>
    <param-name>widgetset</param-name>
    <param-value>com.example.AppWidgetSet</param-value>
</init-param>
</servlet>
```

这个参数的值是 widget set 的类名, 不带 .gwt.xml 扩展名, 带 Java 包名, 使用 Java 的点分隔格式.

17.5. 问题诊断

如果你在使用 add-on 时遇到了问题, 可以尝试以下方法:

- 检查工程根包(root package)下的 .gwt.xml 描述文件. 比如, 如果工程的根包是 com.example.myproject, 那么 widget set 定义文件通常是在 com/example/project/AppWidgetset.gwt.xml. 文件路径不是固定的, 也可以在其他地方, 只要指向文件的参照是正确的. widget set 通过客户端模块描述文件来定义, 关于客户端模块描述文件的内容, 请参见 第 13.3 节 “客户端模块描述文件”.
- 检查 WEB-INF/web.xml 部署描述文件, 看看你的 UI 的 servlet 是否指定了 widget set 参数, 比如:

```
<init-param>
    <description>UI widgetset</description>
    <param-name>widgetset</param-name>
    <param-value>com.example.project.AppWidgetSet</param-value>
</init-param>
```

检查源代码树中 widget set 类与 .gwt.xml 文件是否一致.

- 查看 VAADIN/widgetsets 目录, 检查 widget set 是否存在. 你可以删除这个目录, 然后重编译 widget set, 看看编译动作是否正确.
- 使用 Firebug 的 **Net** 页面, 查看是否正确的装载了 widget set (和 theme).
- 对应用程序使用 ?debug 参数, 打开 debug 窗口, 检查 widget set 与 Vaadin 库, 或 theme 之间是否存在版本冲突. 详情请参见 第 11.3 节 “Debug 模式和 Debug 窗口”.
- 刷新工程, 并重编译. 在 Eclipse 中, 请选中工程, 然后按 **F5** 键, 停止应用程序服务器, 清空服务器临时目录, 然后再重启服务器.
- 在 Eclipse (或者你使用的别的 IDE) 中, 检查 Error Log 中输出的信息.

Vaadin Charts

18.1. 概述	455
18.2. 安装 Vaadin Charts	457
18.3. 基本使用	457
18.4. 图表类型	461
18.5. Chart 配置	479
18.6. 图表数据	481
18.7. 高级使用	484
18.8. 时间线	486

本章介绍 Vaadin Charts add-on.

18.1. 概述

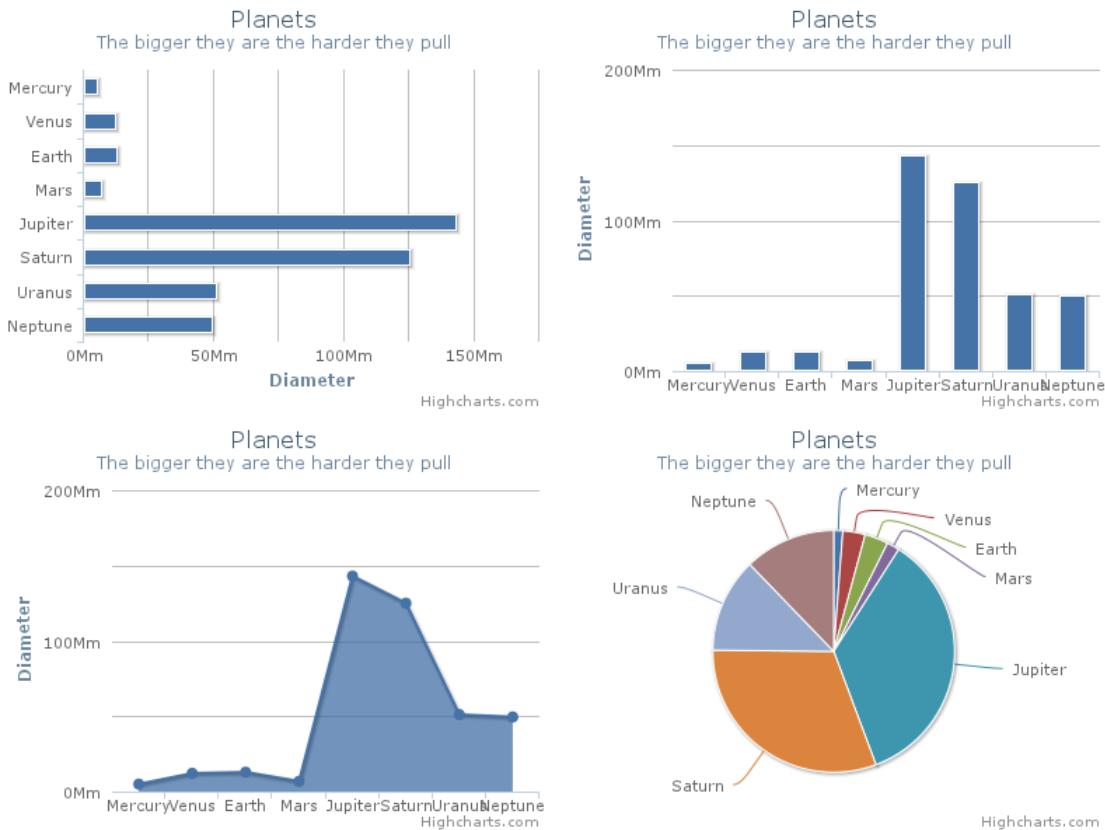
Vaadin Charts 是一个功能丰富的交互式 Vaadin 图表库。它提供了 **Chart** 和 **Timeline** 组件。**Chart** 可以可视化地显示一维度和二维的数据，支持多种图表类型。图表中的元素及显示方式都可以灵活配置。这个库还包括很多内建的 theme，你也可以扩展这些 theme。**Chart** 的基本功能还包括，允许用户通过多种方式与图表元素进行交互，你也可以通过点击事件来实现自定义的用户交互。**Timeline** 是一个专用于显示时间序列的组件，详情将在 第 18.8 节“时间线”中介绍。

图表中显示的数据可以是一维或二维的表格式数据，也可以是带有任意 X 和 Y 值的散点数据。范围图(Range Chart)中显示的数据带有最大值和最小值，而不仅仅是单一值(singular value)。

本章只介绍 Vaadin Charts 的基本使用，以及图表的基本配置。关于配置参数和类的详细文档，请参照这个库的 JavaDoc API 文档。

下面给出一个基本示例，(这个例子会在 第 18.3 节“基本使用”中继续扩展)，我们介绍如何在一个柱形图中显示一维数据，以及如何定制 X 轴和 Y 轴的标签和标题。

图 18.1. Vaadin Charts: 条形图(Bar Chart), 柱形图(Column Chart), 面积图(Area Chart), 饼图(Pie Chart)



```

Chart chart = new Chart(ChartType.BAR);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Planets");
conf.setSubTitle("The bigger they are the harder they pull");
conf.getLegend().setEnabled(false); // Disable legend

// The data
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
              6800, 143000, 125000,
              51100, 49500);
conf.addSeries(series);

// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addxAxis(xaxis);

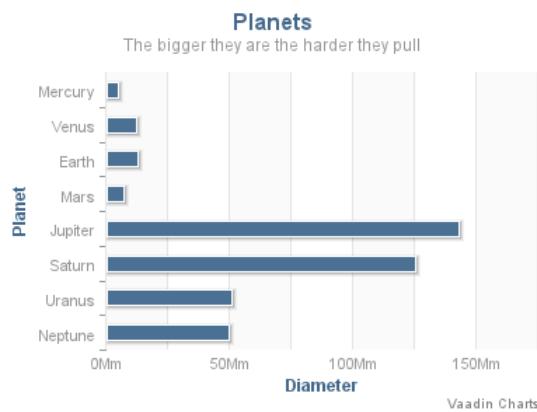
```

```
// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + \\'Mm\\';}");
yaxis.getLabels().setStep(2);
conf.addYAxis(yaxis);

layout.addComponent(chart);
```

运行结果见 图 18.2 “图表基本示例”.

图 18.2. 图表基本示例



Vaadin Charts 是基于 Highcharts JS 开发的, 这是一个 JavaScript 的图表库.

许可协议

Vaadin Charts 是一个商业产品, 使用 CVAL 许可协议(Commercial Vaadin Add-On License). 不论是发布到 Web 上的应用程序还是仅在局域网内使用, 任何使用都需要购买许可. 使用 Vaadin Charts 不需要另外购买 Highcharts JS 的许可.

可以在 Vaadin Directory 购买商业许可, 在这里还可以找到许可协议的详细内容, 也可以下载 Vaadin Charts.

18.2. 安装 Vaadin Charts

Vaadin Charts 可以用于 Vaadin 7 和 Vaadin 6. 可以使用安装包进行安装, 安装包可从 Vaadin Directory 下载, 也可以使用 Maven 或 Ivy 获得. 详细的安装指南, 请参见 第 17 章 使用 Vaadin Add-on.

将 Vaadin Charts 安装到你的工程之后, 你还需要编译 widget set.

18.3. 基本使用

Chart 是一个通常的 Vaadin 组件, 你可以将它添加到布局之内. 你可以在构造函数中指定图表类型, 也可以稍后在图表模型中设定. 图表默认高度为 400 像素, 并占据全部宽度, 你通常会需要定制这个默认设定.

```
Chart chart = new Chart(ChartType.COLUMN);
chart.setWidth("400px"); // 100% by default
chart.setHeight("300px"); // 400px by default
```

图表类型将在 第 18.4 节“图表类型”中介绍.

配置

创建图表后, 你需要对它进行更多配置. 在配置信息中, 你至少需要指定需要显示的值序列.

Chart 对象中的大多数方法负责处理它的 Vaadin 组件基本属性. 与图表相关的属性则在独立的 **Configuration** 对象中管理, 你可以通过 `getConfiguration()` 方法得到这个对象.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Reindeer Kills by Predators");
conf.setSubTitle("Kills Grouped by Counties");
```

`configuration` 的属性将在 第 18.5 节“Chart 配置”中详细介绍.

绘图选项(Plot Option)

图表的很多属性都可以通过图表或值序列的 绘图选项 来设置. 其中大部分设置项目是在所有图表类型中共通的, 另一部分与具体的图表类型相关的, 各图表类型的相关设置将在后文中详细介绍.

比如, 对于折线图(line chart), 你可以使用如下方法来禁用各数据点上的标记:

```
// Disable markers from lines
PlotOptionsLine plotOptions = new PlotOptionsLine();
plotOptions.setMarker(new Marker(false));
conf.setPlotOptions(plotOptions);
```

绘图选项可以对整个图表设置, 也可以对各个值序列分别设置, 以便实现混合类型的图表, 详情请参见 第 18.3.2 节“混合类型图表”.

关于绘图选项中的共通项目, 请参见 第 18.5.1 节“绘图选项(Plot Option)”.

图表数据

图表中显示的数据, 以 **Series** 对象列表的形式存储在图表的配置信息中. 可以使用 `addSeries()` 方法向图表中添加新的数据序列.

```
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
              6800, 143000, 125000,
              51100, 49500);
conf.addSeries(series);
```

数据可以通过多种不同类型的值序列来指定: **DataSeries**, **ListSeries**, **AreaListSeries**, 以及 **RangeSeries**. 关于数据的设置, 详情请参见 第 18.6 节“图表数据”.

坐标轴配置

图表中最常见的任务是定制坐标轴. 你至少会希望设置坐标轴的标题. 通常你还会希望指定数据值在坐标轴上的标签.

如果一个坐标轴不是数值而是分组, 你可以为各个分组指定标签. 分组标签的数量和顺序必须与数据序列中的值一致.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);
```

对于数值标签，它的格式可以通过 JavaScript 表达式来控制，如下例：

```
// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + 'Mm';}");
yaxis.getLabels().setStep(2);
conf.addYAxis(yaxis);
```

18.3.1. 显示多个数据序列

我们在本章前面的例子中看到的是最简单的数据，它只有一维，可以使用单个数据序列来表达。大多数图表类型支持多数据序列，用来表达二维数据。比如，在折线图(Line Chart)中可以有多条数据线，在柱形图(Column Chart)中，不同的值序列按不同的分组显示为多个柱形。不同的图表类型还支持不同的显示模式，比如堆栈式柱形(stacked column)图。图例会为各个值序列显示各自的符号。

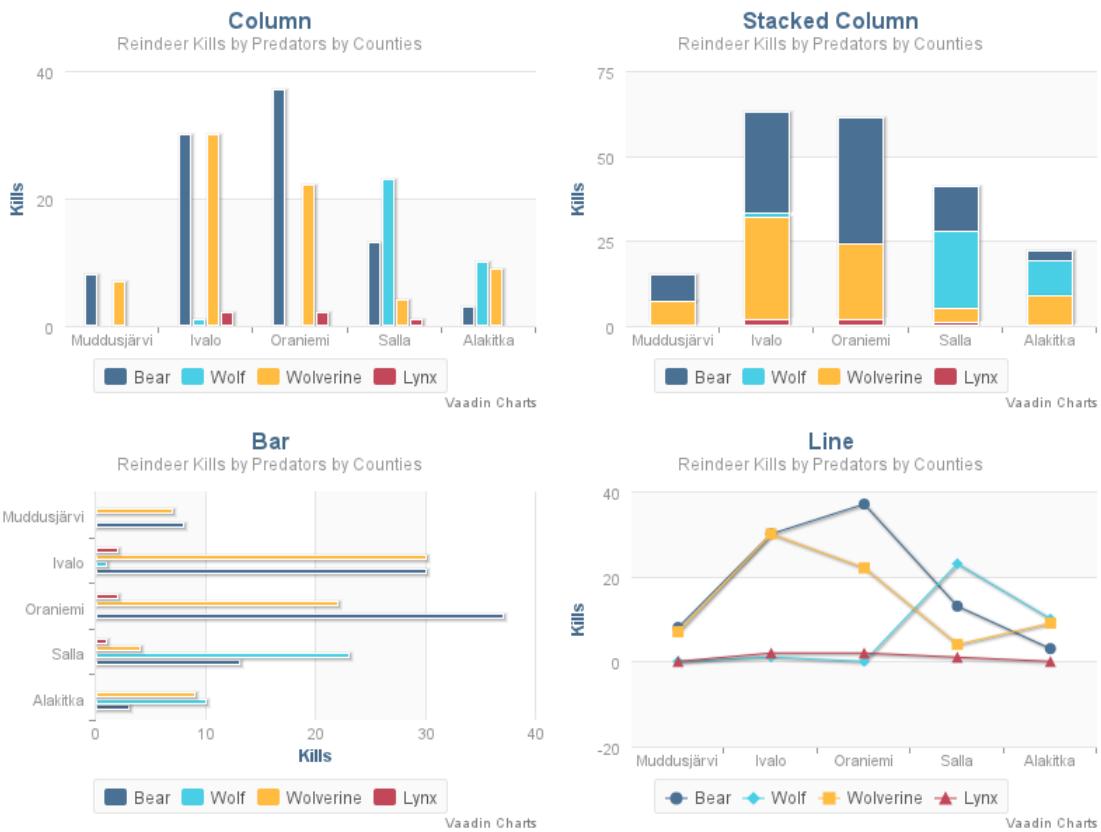
```
// The data
// Source: V. Maijala, H. Norberg, J. Kumpula, M. Nieminen
// Calf production and mortality in the Finnish
// reindeer herding area. 2002.
String predators[] = {"Bear", "Wolf", "Wolverine", "Lynx"};
int kills[][] = {
    {8, 0, 7, 0}, // Muddusjarvi
    {30, 1, 30, 2}, // Ivalo
    {37, 0, 22, 2}, // Oraniemi
    {13, 23, 4, 1}, // Salla
    {3, 10, 9, 0}, // Alakitka
};

// Create a data series for each numeric column in the table
for (int predator = 0; predator < 4; predator++) {
    ListSeries series = new ListSeries();
    series.setName(predators[predator]);

    // The rows of the table
    for (int location = 0; location < kills.length; location++)
        series.addData(kills[location][predator]);
    conf.addSeries(series);
}
```

上例的运行结果，包括正常图表和堆栈式柱形图表两种模式，见图 18.3 “图表中的多个数据序列”。可以使用 **PlotOptionsColumn** 类的 **setStacking()** 方法来切换为堆栈式柱形图。

图 18.3. 图表中的多个数据序列



18.3.2. 混合类型图表

与图表对象一样，每个数据序列都有自己的 **PlotOptions** 对象，可以控制各个数据序列的不同设置。其中包括各个数据序列的图表类型，因此你可以在同一个图表中混合使用不同的图表类型。

一个数据序列的图表类型通过绘图选项的类型决定。比如，为了得到一个折线图，你应该使用 **PlotOptionsLine** 类。

```
// A data series as column graph
DataSeries series1 = new DataSeries();
PlotOptionsColumn options1 = new PlotOptionsColumn();
options1.setFillColor(SolidColor.BLUE);
series1.setPlotOptions(options1);
series1.setData(4900, 12100, 12800,
    6800, 143000, 125000,
    51100, 49500);
conf.addSeries(series1);

// A data series as line graph
ListSeries series2 = new ListSeries("Diameter");
PlotOptionsLine options2 = new PlotOptionsLine();
options2.setLineColor(SolidColor.RED);
series2.setPlotOptions(options2);
series2.setData(4900, 12100, 12800,
    6800, 143000, 125000,
    51100, 49500);
conf.addSeries(series2);
```

18.3.3. 图表 Theme

图表的外观风格,以及其他一切设置,都可以通过 `theme` 来定义. UI 中显示的所有图表只能有一个 theme, 可以通过 **ChartOptions** 类的 `setTheme()` 方法来设置.

在 Vaadin 7 中, **ChartOptions** 类是一个 **UI** 扩展, 可以调用 `get()` 方法来创建和使用它, 如下例:

```
// Set Charts theme for the current UI
ChartOptions.get().setTheme(new SkiesTheme());
```

在 Vaadin 6 中, 它是一个不可见组件, 你需要创建它并添加到窗口中. 窗口中可能只有一个这样的组件, 它必须在所有的 **Chart** 组件之前创建.

```
ChartOptions options = new ChartOptions();
options.setTheme(new SkiesTheme());
content.addComponent(options);
```

Vaadin Charts 的默认图表 theme 是 **VaadinTheme**. 还有其他可用的 theme: **GrayTheme**, **GridTheme**, 以及 **SkiesTheme**. Highchart 的默认 theme 可以使用 **HighChartsDefaultTheme** 来设置.

theme 是 Vaadin Charts 配置的一种, 当在界面上描绘图表时, 会使用 theme 作为配置的模板.

18.4. 图表类型

Vaadin Charts 支持十多种不同的图表类型. 通常可以在 **Chart** 对象的构造方法中指定图表类型. 可用的图表类型定义在 **ChartType** 枚举型中. 也可以稍后再使用图表模型的 `chartType` 属性来读取或设置图表类型, 图表模型可以通过 `getConfiguration().getChart()` 方法取得.

各种图表类型都有它独自的绘图选项, 支持各自的图表功能. 对于数据序列也存在各自不同的要求.

在下面的小节中, 将介绍基本的图表类型和它们的一些变体.

18.4.1. 折线图(Line Chart)与曲线图(Spline Chart)

折线图将一系列的数据点通过线条连接起来. 在基本的折线图中, 连接线是直线, 在曲线图中, 会在数据点之间使用多项式插值算法生成比较光滑的曲线.

表 18.1. 线条图的子类型

图表类型	绘图选项类
LINE	PlotOptionsLine
SPLINE	PlotOptionsSpline

绘图选项(Plot Option)

折线图的绘图选项中的 `color` 属性控制线条的颜色, `lineWidth` 属性控制线条宽度, `dashStyle` 属性控制线条的点/线模式(dash pattern).

关于标记(marker)和其他数据点属性的绘图选项, 请参见 第 18.4.6 节 “散点图(Scatter Chart)”. 也可以对每个数据点配置标记.

18.4.2. 面积图(Area Chart)

面积图与折线图类似，区别是线条之间的面积，以及Y轴是使用透明颜色描绘的。除基本类型外，图表类型还支持曲线类型和区域类型的组合。

表 18.2. 面积图的子类型

图表类型	绘图选项类
AREA	PlotOptionsArea
AREASPLINE	PlotOptionsAreaSpline
AREARANGE	PlotOptionsAreaRange
AREASPLINERANGE	PlotOptionsAreaSplineRange

在面积范围图(Area Range Chart)中，位于数值下限与上限之间的面积会使用透明颜色描绘。数据序列必须指定为Y坐标之上的最小和最大值，最小和最大值可以使用 **RangeSeries** 来定义，详情请参见第 18.6.3 节“范围数据序列”，也可以使用 **DataSeries** 来定义，详情请参见第 18.6.2 节“一般数据序列”。

绘图选项(Plot Option)

面积图支持 堆栈 模式，因此多个数据序列会层叠式的堆积在一起。你可以使用绘图选项的 `setStacking()` 方法来启用堆栈模式。`Stacking.NORMAL` 模式使用通常的合计叠加方式，`Stacking.PERCENT` 模式则使用相对比例来显示数据。

面积的填充色由 `fillColor` 属性控制，透明度由 `fillOpacity` 属性控制(这个属性设置的是“不透明度”)，这个属性的有效值在 0.0 到 1.0 之间。

折线图的绘图选项中的 `color` 属性控制线条的颜色，`lineWidth` 属性控制线条宽度，`dashStyle` 属性控制线条的点/线模式(dash pattern)。

关于标记(marker)和其他数据点属性的绘图选项，请参见第 18.4.6 节“散点图(Scatter Chart)”。也可以对每个数据点配置标记。

18.4.3. 柱形图(Column Chart)与条形图(Bar Chart)

柱形图与条形图分别使用垂直和水平的棒图来显示数据。这两种图表本质上是一样的，只是数据轴的方向不同。

多个数据序列，也就是，二维数据，会通过各自的种类进行分组，然后显示为较细的横条或竖柱，详情请参见第 18.3.1 节“显示多个数据序列”。可以使用绘图选项的 `setStacking()` 方法来启用堆栈模式，堆栈模式将把不同数据序列的横条或竖柱叠加起来显示。

你也可以使用 `COLUMNRANGE` 图表，它显示数值下限与上限之间的一个范围，详情请参见第 18.4.10 节“区域范围图(Area Range Chart)与列范围图(Column Range Chart)”。范围图表需要使用 **RangeSeries** 来定义数值的下限和上限。

表 18.3. 柱形图与条形图的子类型

图表类型	绘图选项类
COLUMN	PlotOptionsColumn
COLUMNRANGE	PlotOptionsColumnRange

图表类型	绘图选项类
BAR	PlotOptionsBar

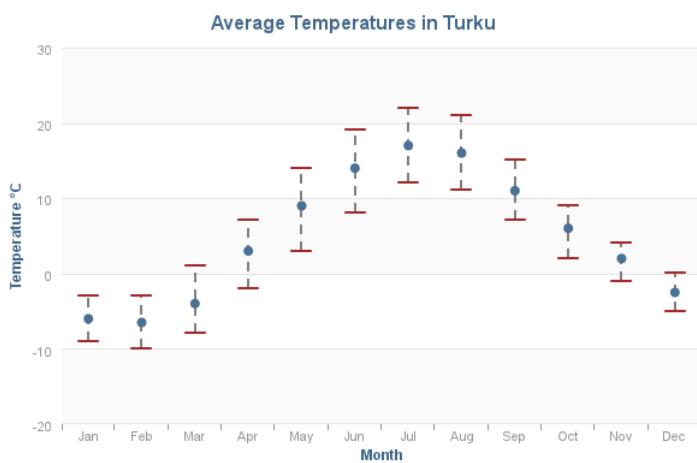
关于绘图选项的详情, 请参见 API 文档.

18.4.4. 错误条(Error Bar)

错误条用来显示统计数据中的错误值, 或者最高/最低值. 错误条通常代表数据中的最高/最低值, 或标准偏差的multitude(译注: 此处理解不能, 待校)的最高/最低值, 百分比的最高/最低值, 比值的最高/最低值. 最高/最低值用水平线条表示, 也叫 "晶须(whisker)", 中间由垂直短柄(stem)连接.

虽然技术上来讲, 错误条是一种图表类型(ChartType.ERRORBAR), 但通常会将它与其他图表类型混合使用, 比如散点图(Scatter Chart)或柱形图(Column Chart).

图 18.4. 散点图中的错误条



要为各个数据点显示错误条, 你需要用单独的数据序列来表达最高/最低值. 这个数据序列需要使用 **PlotOptionsErrorBar** 类型.

```
// Create a chart of some primary type
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("600px");
chart.setHeight("400px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Average Temperatures in Turku");
conf.getLegend().setEnabled(false);

// The primary data series
ListSeries averages = new ListSeries(
    -6, -6.5, -4, 3, 9, 14, 17, 16, 11, 6, 2, -2.5);

// Error bar data series with low and high values
DataSeries errors = new DataSeries();
errors.add(new DataSeriesItem(0, -9, -3));
errors.add(new DataSeriesItem(1, -10, -3));
errors.add(new DataSeriesItem(2, -8, 1));
...

```

```

// Configure the stem and whiskers in error bars
PlotOptionsErrorBar barOptions = new PlotOptionsErrorBar();
barOptions.setStemColor(SolidColor.GREY);
barOptions.setStemWidth(2);
barOptions.setStemDashStyle(DashStyle.DASH);
barOptions.setWhiskerColor(SolidColor.BROWN);
barOptions.setWhiskerLength(80); // 80% of category width
barOptions.setWhiskerWidth(2); // Pixels
errors.setPlotOptions(barOptions);

// The errors should be drawn lower
conf.addSeries(errors);
conf.addSeries(averages);

```

注意, 你应该先添加错误条的数据序列, 使错误条显示在图表中比较接近背景的位置, 而不要遮挡其他数据的显示.

绘图选项(Plot Option)

错误条图表的绘图选项类型为 **PlotOptionsErrorBar**. 它独有的绘图选项属性如下:

whiskerColor, whiskerWidth, 和 whiskerLength
用于表示最高/最低值的水平"晶须"的颜色, 宽度(竖直方向上的粗细), 和长度.

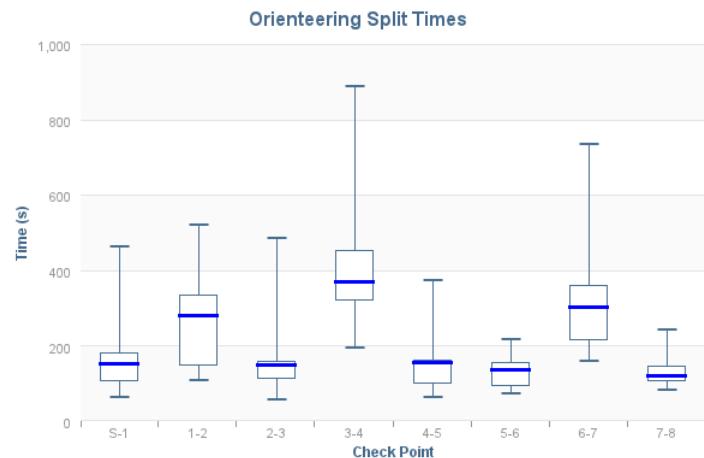
stemColor, stemWidth, 和 stemDashStyle
连接晶须的垂直短柄(stem)的颜色, 宽度(粗细), 和线条风格. 在箱形图(Box Plot Chart)中, 也存在从箱形方块中延伸出来的短柄(stem).

18.4.5. 箱形图(Box Plot Chart)

箱形图用于显示统计变量的分布状况. 数据点中的中位值, 显示为一个水平线条, 数据点中的第一和第三四分位值, 显示为箱形, 数据点中的最高/最低值, 显示为T形的"晶须". 箱形符号的确切含义由你自由决定.

箱形图与错误条的关系很紧密, 详情请参见第 18.4.4 节“错误条(Error Bar)”, 它们之间会共用箱形元素和晶须元素.

图 18.5. 箱形图



箱形图的图表类型是 `ChartType.BOXPLOT`. 通常你只需要一个值序列, 因此可以禁止图例的显示.

```
Chart chart = new Chart(ChartType.BOXPLOT);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Orienteering Split Times");
conf.getLegend().setEnabled(false);
```

绘图选项(Plot Option)

箱形图的绘图选项类型为 **PlotOptionsBoxPlot**, 它继承字更通用的 **PlotOptionsErrorBar** 类. 这两个类的绘图选项属性如下:

medianColor, medianWidth
中间值的水平指示线的颜色和宽度(垂直方向上的粗细).

例:

```
// Set median line color and thickness
PlotOptionsBoxPlot plotOptions = new PlotOptionsBoxPlot();
plotOptions.setMedianColor(SolidColor.BLUE);
plotOptions.setMedianWidth(3);
conf.setPlotOptions(plotOptions);
```

数据模型

由于箱形图中的数据点有 5 个不同的值, 而不是通常的一个值, 因此它们需要使用特殊的 **BoxPlotItem** 来表达. 你可以使用 `setter` 方法来分别指定这些值, 也可以在构造函数中一次性指定.

```
// Orienteering control point times for runners
double data[][] = orienteeringdata();

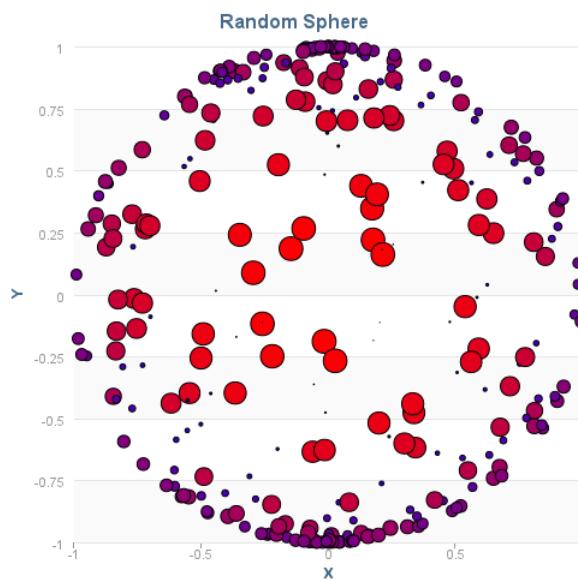
DataSeries series = new DataSeries();
for (double cpointtimes[]: data) {
    StatAnalysis analysis = new StatAnalysis(cpointtimes);
    series.add(new BoxPlotItem(analysis.low(),
                               analysis.firstQuartile(),
                               analysis.median(),
                               analysis.thirdQuartile(),
                               analysis.high()));
}
conf.setSeries(series);
```

如果 "low" 属性和 "high" 属性代表更小的分位数, 或者代表更大的标准偏差倍数, 你可以使用离散值(outlier). 你可以使用单独的数据序列来描绘离散值(outlier), with.....(译注: 此句原文不完整)

18.4.6. 散点图(Scatter Chart)

散点图用于显示一组互不相关的数据点. 名字叫做散点图, 是因为数据点的 X 和 Y 坐标都可以任意指定, 因此散点图中最常用的数据序列类型是 **DataSeries** 或 **ContainerSeries**.

图 18.6. 散点图



散点图的图表类型是 `ChartType.SCATTER`. 它的绘图选项可以通过 **PlotOptionsScatter** 对象来配置, 但这个类中不存在独有的绘图选项.

```
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("500px");
chart.setHeight("500px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Random Sphere");
conf.getLegend().setEnabled(false); // Disable legend

PlotOptionsScatter options = new PlotOptionsScatter();
// ... Give overall plot options here ...
conf.setPlotOptions(options);

DataSeries series = new DataSeries();
for (int i=0; i<300; i++) {
    double lng = Math.random() * 2 * Math.PI;
    double lat = Math.random() * Math.PI - Math.PI/2;
    double x   = Math.cos(lat) * Math.sin(lng);
    double y   = Math.sin(lat);
    double z   = Math.cos(lng) * Math.cos(lat);

    DataSeriesItem point = new DataSeriesItem(x,y);
    Marker marker = new Marker();
    // Make settings as described later
    point.setMarker(marker);
    series.add(point);
}
conf.addSeries(series);
```

运行结果见 图 18.6 “散点图”.

数据点的标记(Marker)

散点图和其他显示数据点的图表，比如折线图和曲线图，通过 **标记(marker)** 来显示数据点。标记可以使用 **Marker** 属性对象来配置，Marker 对象可以从相关图表类型的绘图选项中得到，也可以通过各个数据点对应的 **DataSeriesItem** 类得到。你需要创建标记，并通过 `setMarker()` 方法将它添加到绘图选项或数据序列项目中。

比如，为了给各个数据点设置各自的标记，可以使用以下代码：

```
 DataSeriesItem point = new DataSeriesItem(x, y);
 Marker marker = new Marker();
 // ... Make any settings ...
 point.setMarker(marker);
 series.add(point);
```

标记的形状属性

标记带有 `lineColor` 和 `fillColor` 属性，这些属性的值可以使用 **Color** 对象来设置。这些属性既支持单纯色，也支持渐变色。你可以使用 **SolidColor** 来指定一个单纯的填充色，颜色使用 RGB 值来表达，也可以使用类中预定义的颜色常数。

```
// Set line width and color
marker.setLineWidth(1); // Normally zero width
marker.setLineColor(SolidColor.BLACK);

// Set RGB fill color
int level = (int) Math.round((1-z)*127);
marker.setFillColor(
    new SolidColor(255-level, 0, level));
point.setMarker(marker);
series.add(point);
```

你也可以使用 **GradientColor** 来指定渐变色。这个类支持直线型的渐变色，也支持放射状的渐变色，还可以指定多个色彩停止点(color stop)。

标记的大小由 `radius` 参数指定，使用像素单位。画面上实际显示的半径还会包括线条宽度部分。

```
marker.setRadius((z+1)*5);
```

标记符号

标记在图表中显示为一个几何图形符号，或者一个图片符号。你可以选择 **MarkerSymbolEnum** 枚举型中预定义的图形(*CIRCLE*, *SQUARE*, *DIAMOND*, *TRIANGLE*, 或者 *TRIANGLE DOWN*)。这些图形可以由单纯的线条绘制，也可以填充色彩，填充色的设置方法请参见前面的示例。

```
marker.setSymbol(MarkerSymbolEnum.DIAMOND);
```

你也可以使用任意的图片符号，图片 URL 通过 **MarkerSymbolUrl** 来指定。如果图片文件随你的应用程序一起发布，比如放在 theme 文件夹之内，你可以使用以下方法取得它的 URL：

```
String url = VaadinServlet.getCurrent().getServletContext()
    .getContextPath() + "/VAADIN/themes/mytheme/img/smiley.png";
marker.setSymbol(new MarkerSymbolUrl(url));
```

对于图片符号来说，线条，半径，以及颜色属性都是无效的。

18.4.7. 气泡图(Bubble Chart)

气泡图是散点图的一种特殊类型，它使用不同的尺寸来表达三维数据点。在第 18.4.6 节“散点图(Scatter Chart)”中我们介绍了散点图中如何定义各数据点的尺寸，但气泡图中可以更简单地实现这个效果，它使用第三(Z)维数据，而不是 radius 属性。气泡尺寸与其他维度的数据一样，会自动调节比例。而且数据点的默认显示也更偏向气泡风格一些。

图 18.7. 气泡图



气泡图的图表类型是 `ChartType.BUBBLE`。它的绘图选项使用 **PlotOptionsBubble** 对象来配置，这个对象独有的属性只有一个，`displayNegative`，这个属性控制负值的气泡是否显示。通常你会更关心配置气泡的 `marker`。气泡的提示信息(tooltip)通过基本属性来配置。在提示信息的 JavaScript 格式化脚本中，可以通过 `this.point.z` 来引用数据点的 Z 坐标值。

气泡半径根据最大和最小半径按比例计算得到。如果你希望按面积比例而不是半径比例来控制气泡大小，你可以对 Z 值取平方根来实现。

下面的例子中，我们将一个气泡图叠在一个世界地图上。我们让气泡显示为球形渐变色。注意，我们对 Z 轴的值取了平方根。(译注：原文此句不完整)

```
// Create a bubble chart
Chart chart = new Chart(ChartType.BUBBLE);
chart.setWidth("640px"); chart.setHeight("350px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Champagne Consumption by Country");
conf.getLegend().setEnabled(false); // Disable legend
conf.getTooltip().setFormatter("this.point.name + '：" + "
    Math.round(100*(this.point.z * this.point.z))/100.0 + " +
    "' M bottles'");

// World map as background
String url = VaadinServlet.getCurrent().getServletContext()
    .getContextPath() + "/VAADIN/themes/mytheme/img/map.png";
conf.getChart().setPlotBackgroundImage(url);

// Show more bubbly bubbles with spherical color gradient
PlotOptionsBubble plotOptions = new PlotOptionsBubble();
Marker marker = new Marker();
GradientColor color = GradientColor.createRadial(0.4, 0.3, 0.7);
color.addColorStop(0.0, new SolidColor(255, 255, 255, 0.5));
color.addColorStop(1.0, new SolidColor(170, 70, 67, 0.5));
```

```
marker.setFillColor(color);
plotOptions.setMarker(marker);
conf.setPlotOptions(plotOptions);

// Source: CIVC - Les expéditions de vins de Champagne en 2011
DataSeries series = new DataSeries("Countries");
Object data[][] = {
    {"France",           181.6},
    {"United Kingdom",   34.53},
    {"United States",    19.37},
    ...
};
for (Object[] country: data) {
    String name = (String) country[0];
    double amount = (Double) country[1];
    Coordinate pos = getCountryCoordinates(name);

    DataSeriesItem3d item = new DataSeriesItem3d();
    item.setX(pos.longitude * Math.cos(pos.latitude/2.0 *
                                         (Math.PI/160)));
    item.setY(pos.latitude * 1.2);
    item.setZ(Math.sqrt(amount));
    item.setName(name);
    series.add(item);
}
conf.addSeries(series);

// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
xaxis.setExtremes(-180, 180);
...
conf.addxAxis(xaxis);

// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setExtremes(-90, 90);
...
conf.addyAxis(yaxis);
```

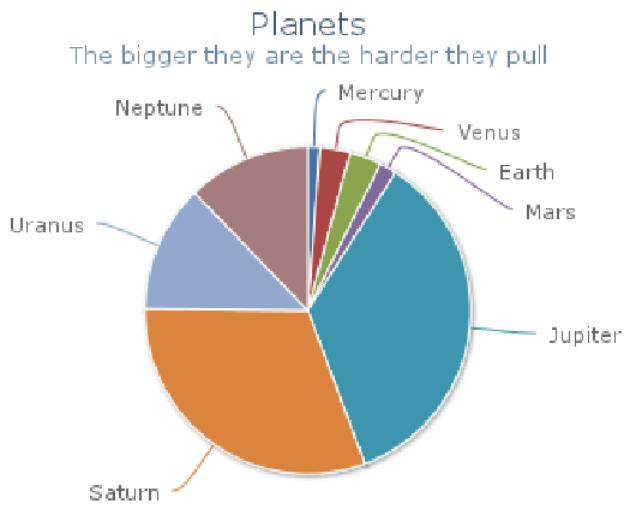
18.4.8. 饼图(Pie Chart)

饼图将各个数据值显示为扇形，扇形大小由数据值的比例决定。饼图对应的图表类型是 `ChartType.PIE`，你可以通过 **PlotOptionsPie** 对象来配置它的绘图选项，详情将在后文中介绍。

```
Chart chart = new Chart(ChartType.PIE);
Configuration conf = chart.getConfiguration();
...
```

饼图的运行结果见图 18.8 “饼图”。

图 18.8. 饼图



绘图选项(Plot Option)

饼图独有的绘图选项由 **PlotOptionsPie** 来配置.

```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize(0); // Non-0 results in a donut
options.setSize("75%"); // Default
options.setCenter("50%", "50%"); // Default
conf.setPlotOptions(options);
```

innerSize

内环尺寸如果大于 0, 饼图将变为 "圆环(Donut, 甜甜圈)" 形状. 内环尺寸可以表示为像素数字, 也可以是相对于图表面积的百分比字符串(比如 "60%"). 关于圆环图, 请参见后面的小节.

size

饼图的尺寸可以是一个像素数字, 也可以是相对于图表面积的百分比字符串(比如 "80%"). 默认尺寸是 75%, 保留了一些空间用于显示标签.

center

饼图中心点的 X 和 Y 坐标, 可以是像素数字, 也可相对于图表尺寸的百分比字符串. 默认值是 "50%", "50%".

数据模型

饼图各扇区的标签由数据点的标签决定. 饼图应该使用 **DataSeries** 或 **ContainerSeries** 来管理数据点, 这两个类允许为数据点指定标签.

```
DataSeries series = new DataSeries();
series.add(new DataSeriesItem("Mercury", 4900));
series.add(new DataSeriesItem("Venus", 12100));
...
conf.addSeries(series);
```

一个数据点, 也就是 **DataSeries** 中的一个 **DataSeriesItem**, 如果它的 *sliced* 属性为 true, 它会显示为从饼图中略微切出的样子.

```
// Slice one sector out
DataSeriesItem earth = new DataSeriesItem("Earth", 12800);
earth.setSliced(true);
series.add(earth);
```

圆环图(Donut Chart)

在饼图的绘图选项中, 将 `innerSize` 设置为大于 0 的值, 会导致饼图中心出现一个空洞.

```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize("60%");
conf.setPlotOptions(options);
```

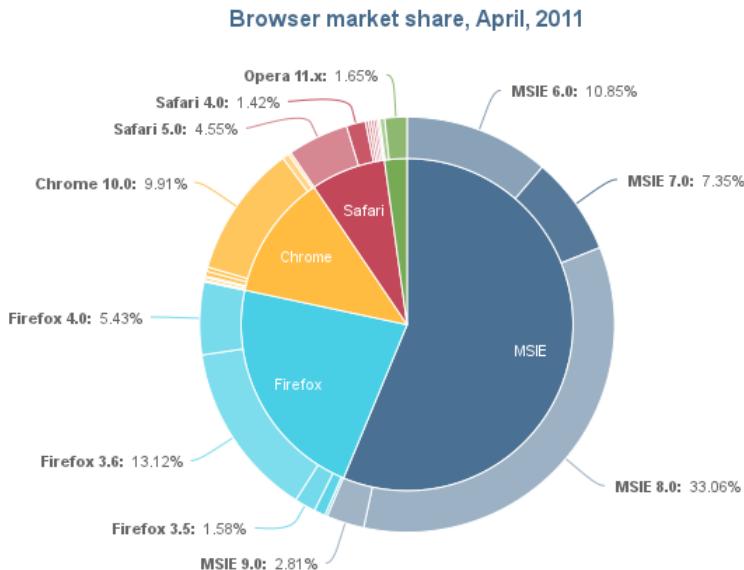
由于你可以对各个数据序列设置各自的绘图选项, 你可以将两个饼图叠加起来, 将较小的饼图放在圆环图中心的“空洞”处. 使用这种方式, 你可以在外环中绘制一个更多细节的饼图, 如下例所示:

```
// The inner pie
DataSeries innerSeries = new DataSeries();
innerSeries.setName("Browsers");
PlotOptionsPie innerOptions = new PlotOptionsPie();
innerOptions.setSize("60%");
innerSeries.setPlotOptions(innerOptions);
...

DataSeries outerSeries = new DataSeries();
outerSeries.setName("Versions");
PlotOptionsPie outerOptions = new PlotOptionsPie();
outerOptions.setInnerSize("60%");
outerSeries.setPlotOptions(outerOptions);
...
```

运行结果见 图 18.9 “层叠在一起的饼图与圆环图”.

图 18.9. 层叠在一起的饼图与圆环图



18.4.9. 仪表图(Gauge Chart)

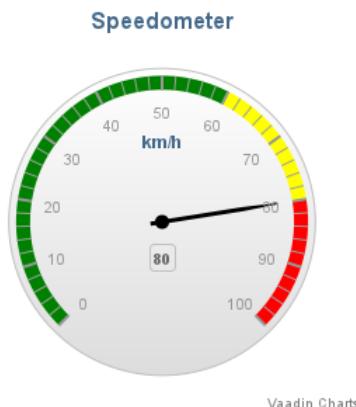
仪表图是一种一维图表，它的 Y 轴显示为环形，一个旋转的指针指向 Y 轴上的数值。仪表图实际上可以有多个 Y 轴，显示多种尺度。

我们来看看下面的仪表图：

```
Chart chart = new Chart(ChartType.GAUGE);
chart.setWidth("400px");
chart.setHeight("400px");
```

完成设置之后，运行结果见 图 18.10 “仪表图”，设置的详情参见后面的小节。

图 18.10. 仪表图



Vaadin Charts

仪表图的配置

仪表图的开始和结束角度可以在 **Pane** 对象中配置。角度的有效值范围是 -360 到 360 度，0 度代表圆环的正上方。

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Speedometer");
conf.getPane().setStartAngle(-135);
conf.getPane().setEndAngle(135);
```

坐标轴配置

仪表图只有 Y 轴。你需要指定 Y 轴的最小和最大值。

```
YAxis yaxis = new YAxis();
yaxis.setTitle("km/h");

// The limits are mandatory
yaxis.setMin(0);
yaxis.setMax(100);

// Other configuration
yaxis.getLabels().setStep(1);
yaxis.setTickInterval(10);
yaxis.setPlotBands(new PlotBand[]{
    new PlotBand(0, 60, SolidColor.GREEN),
    new PlotBand(60, 80, SolidColor.YELLOW),
    new PlotBand(80, 100, SolidColor.RED)});
```

```
conf.addYAxis(yaxis);
```

你可以对 Y 轴设置其他一切属性 - 关于可设置的属性, 请参见 API 文档.

设置和更新仪表图的数据

仪表图只显示一个单独的数值, 这个数值通过一个长度为 1 的数据序列来定义, 如下:

```
ListSeries series = new ListSeries("Speed", 80);
conf.addSeries(series);
```

仪表图用来显示变化的值非常有效. 你可以使用数据序列的 `updatePoint()` 方法来更新仪表图中显示的那个单独的数据值.

```
final TextField tf = new TextField("Enter a new value");
layout.addComponent(tf);

Button update = new Button("Update", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        Integer newValue = new Integer((String)tf.getValue());
        series.updatePoint(0, newValue);
    }
});
layout.addComponent(update);
```

18.4.10. 区域范围图(Area Range Chart)与列范围图(Column Range Chart)

范围图显示最大最小值之间的区域或列, 而不是显示单独的数据点. 这些图表需要使用 **RangeSeries**, 详情请参见第 18.6.3 节“范围数据序列”. 区域范围图使用的图表类型是 `AREARANGE`, 列范围图使用的是 `COLUMNRANGE`.

我们来看看下面的例子:

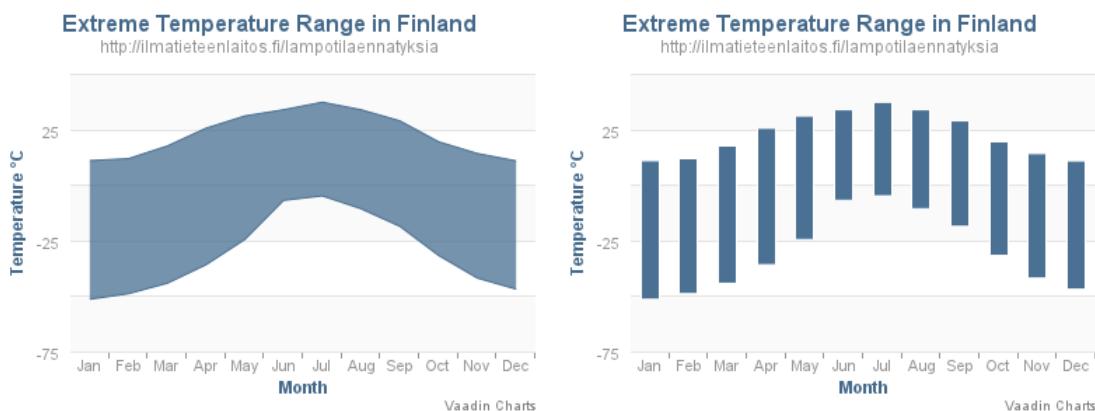
```
Chart chart = new Chart(ChartType.AREARANGE);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Extreme Temperature Range in Finland");
...

// Create the range series
// Source: http://ilmatieteenlaitos.fi/lampotilaennatyksia
RangeSeries series = new RangeSeries("Temperature Extremes",
    new Double[]{-51.5,10.9},
    new Double[]{-49.0,11.8},
    ...
    new Double[]{-47.0,10.8});//
conf.addSeries(series);
```

上例的运行结果, 以及等价的列范围图, 见图 18.11 “区域范围图与列范围图”.

图 18.11. 区域范围图与列范围图



18.4.11. 极坐标图(Polar Chart), 风玫瑰图(Wind Rose Chart), 与蜘蛛网图(Spiderweb Chart)

带有两个坐标轴的图表, 大多可以使用 极坐标(polar) 来表示, 其中 X 轴弯曲为一个圆环, Y 轴从圆环的中心点出发向边框延伸。极坐标图不是一个独立的图表类型, 它可以在大多数图表的数据模型中使用 `setPolar(true)` 方法来启用。因此, 各个图表的其他特性对于极坐标图也是有效的。

Vaadin Charts 支持多种常见的极坐标图类型, 比如 风玫瑰图(wind rose), 它是一种极坐标式的柱图, 以及 蜘蛛网图(spiderweb), 它以极坐标形式显示分组式数据, 效果类似一个多边形。

```
// Create a chart of some type
Chart chart = new Chart(ChartType.LINE);

// Enable the polar projection
Configuration conf = chart.getConfiguration();
conf.getChart().setPolar(true);
```

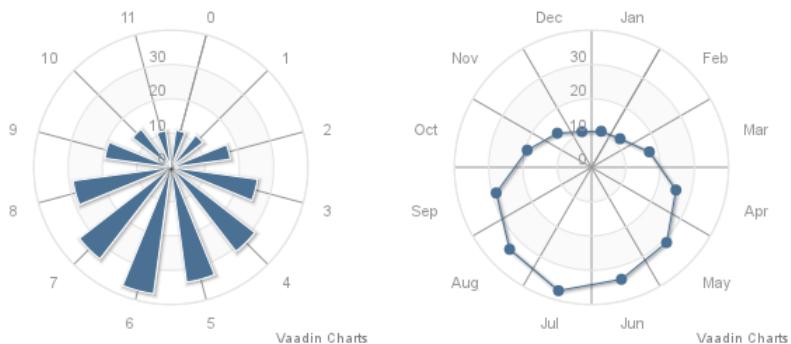
你需要在配置中使用 **Pane** 对象来定义极坐标投影的扇区。扇区使用角度来定义, 角度的起点是正北方向。你还需要使用 `setMin()` 和 `setMax()` 方法来定义 X 轴的值范围。

```
// Define the sector of the polar projection
Pane pane = new Pane(0, 360); // Full circle
conf.addPane(pane);

// Define the X axis and set its value range
XAxis axis = new XAxis();
axis.setMin(0);
axis.setMax(360);
```

风玫瑰图和蜘蛛网图见 图 18.12 “风玫瑰图与蜘蛛网图”。

图 18.12. 风玫瑰图与蜘蛛网图



蜘蛛网图(Spiderweb Chart)

蜘蛛网(spiderweb) 图是极坐标图表中的一种常见显示风格，它使用多边形边框而不是圆形。数据和 X 轴应该属于不同的分组，这样才能产生有意义的多边形扇区。扇区被假定为充满整个圆环，因此不需要指定 pane 的角度。注意下例中对坐标轴设置的样式：

```
Chart chart = new Chart(ChartType.LINE);
...
// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.getChart().setPolar(true);
...
// Create the range series
// Source: http://ilmatieteenlaitos.fi/lampotilaennatyksia
ListSeries series = new ListSeries("Temperature Extremes",
    10.9, 11.8, 17.5, 25.5, 31.0, 33.8,
    37.2, 33.8, 28.8, 19.4, 14.1, 10.8);
conf.addSeries(series);

// Set the category labels on the X axis correspondingly
XAxis xaxis = new XAxis();
xaxis.setCategories("Jan", "Feb", "Mar",
    "Apr", "May", "Jun", "Jul", "Aug", "Sep",
    "Oct", "Nov", "Dec");
xaxis.setTickmarkPlacement(TickmarkPlacement.ON);
xaxis.setLineWidth(0);
conf.addXAxis(xaxis);

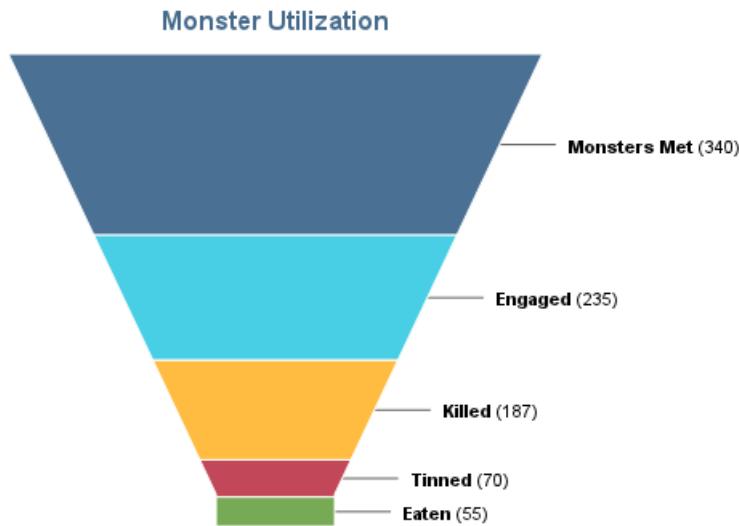
// Configure the Y axis
YAxis yaxis = new YAxis();
yaxis.setGridLineInterpolation("polygon"); // Webby look
yaxis.setMin(0);
yaxis.setTickInterval(10);
yaxis.getLabels().setStep(1);
conf.addYAxis(yaxis);
```

18.4.12. 漏斗图(Funnel Chart)

漏斗图将数据显示为从上到下逐渐缩小的漏斗形，通常用来显示销售过程中的各个阶段，也可以用于别的目的。漏斗图的布局类似堆栈式柱形(stacked column)图，但形状为漏斗形。漏斗最顶端的宽

度与整个图表的显示区域宽度相同，然后宽度逐渐缩小，直到漏斗的颈部，然后再延续到图表的最底部。

图 18.13. 漏斗图(Funnel Chart)



漏斗图的图表类型为 `FUNNEL`。

漏斗中各部分的标签默认显示在右侧，中间带有一条连接线。你可以通过绘图选项来配置它们的样式，如下例所示。

```
Chart chart = new Chart(ChartType.FUNNEL);
chart.setWidth("500px");
chart.setHeight("350px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Monster Utilization");
conf.getLegend().setEnabled(false);

// Give more room for the labels
conf.getChart().setSpacingRight(120);

// Configure the funnel neck shape
PlotOptionsFunnel options = new PlotOptionsFunnel();
options.setNeckHeightPercentage(20);
options.setNeckWidthPercentage(20);

// Style the data labels
Labels dataLabels = new Labels();
dataLabels.setFormat("<b>{point.name}</b> ({point.y:,0f})");
dataLabels.setSoftConnector(false);
dataLabels.setColor(SolidColor.BLACK);
options.setDataLabels(dataLabels);

conf.setPlotOptions(options);

// Create the range series
DataSeries series = new DataSeries();
series.add(new DataSeriesItem("Monsters Met", 340));
series.add(new DataSeriesItem("Engaged", 235));
```

```
series.add(new DataSeriesItem("Killed", 187));
series.add(new DataSeriesItem("Tinned", 70));
series.add(new DataSeriesItem("Eaten", 55));
conf.addSeries(series);
```

绘图选项(Plot Option)

漏斗图独有的绘图选项可以通过 **PlotOptionsFunnel** 来配置。它继承自共通的 **AbstractLinePlotOptions** 类，并带有漏斗图独有的属性，以下：

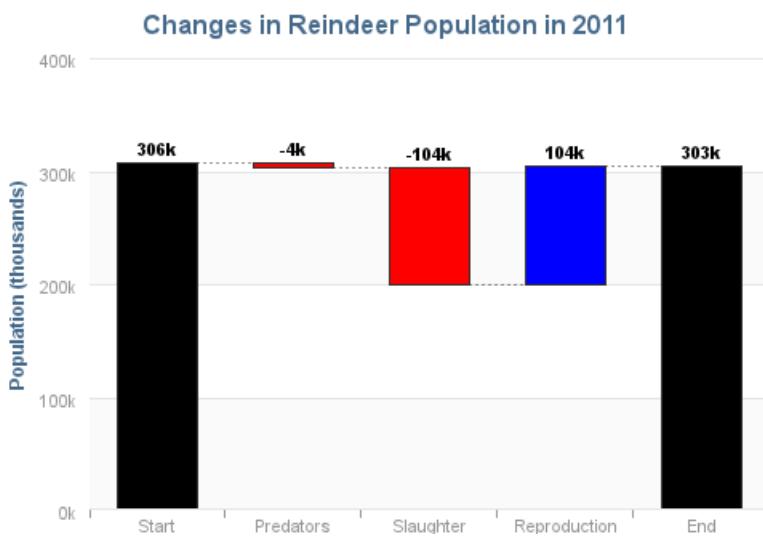
neckHeight 或 *neckHeightPercentage*
漏斗颈部的高度，单位为像素，或相对于漏斗全体高度的百分比。

neckWidth 或 *neckWidthPercentage*
漏斗颈部宽度，单位为像素，或相对于漏斗顶部宽度的百分比。

18.4.13. 瀑布图(Waterfall Chart)

瀑布图用于显示从某个开始层次到最终层次之间的一系列层次变化。变化以差值的形式显示，也可以带有多个中间合计值，这些中间合计值会被自动计算。

图 18.14. 瀑布图(Waterfall Chart)



瀑布图的图表类型是 WATERFALL。如下例：

```
Chart chart = new Chart(ChartType.WATERFALL);
chart.setWidth("500px");
chart.setHeight("350px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Changes in Reindeer Population in 2011");
conf.getLegend().setEnabled(false);

// Configure X axis
XAxis xaxis = new XAxis();
xaxis.setCategories("Start", "Predators", "Slaughter",
    "Reproduction", "End");
conf.addXAxis(xaxis);
```

```
// Configure Y axis
YAxis yaxis = new YAxis();
yaxis.setTitle("Population (thousands)");
conf.addYAxis(yaxis);
...
```

以上示例代码将在后续小节中继续补充完成。

绘图选项(Plot Option)

瀑布图的绘图选项类型为 **PlotOptionsWaterfall**, 它继承自共通的 **PlotOptionsColumn** 类. 它带有瀑布图独有的属性, 如下:

upColor
用于显示正数值的颜色. 对于负数值, 将使用 **PlotOptionsColumn** 类中的 **negativeColor** 属性.

以下代码中, 我们为各列定义颜色, 以及标签的样式和位置:

```
// Define the colors
final Color balanceColor = SolidColor.BLACK;
final Color positiveColor = SolidColor.BLUE;
final Color negativeColor = SolidColor.RED;

// Configure the colors
PlotOptionsWaterfall options = new PlotOptionsWaterfall();
options.setUpColor(positiveColor);
options.setNegativeColor(negativeColor);

// Configure the labels
Labels labels = new Labels(true);
labels.setVerticalAlign(VerticalAlign.TOP);
labels.setY(-20);
labels.setFormatter("Math.floor(this.y/1000) + 'k'");
Style style = new Style();
style.setColor(SolidColor.BLACK);
style.setFontWeight(FontWeight.BOLD);
labels.setStyle(style);
options.setDataLabels(labels);
options.setPointPadding(0);
conf.setPlotOptions(options);
```

值序列

瀑布图的值序列包含从一个起始值开始的一系列变化(delta)值, 以及一个或多个累计值. 至少要有一个最终合计值, 但中间合计值是可选的. 合计值表现为 **WaterFallSum** 数据项目, 但不必为这些数据项目指定值, 因为它们会自动计算. 对于中间合计值, 你应该将 *intermediate* 属性设置为 `true`.

```
// The data
DataSeries series = new DataSeries();

// The beginning balance
DataSeriesItem start = new DataSeriesItem("Start", 306503);
start.setColor(balanceColor);
series.add(start);
```

```
// Deltas
series.add(new DataSeriesItem("Predators", -3330));
series.add(new DataSeriesItem("Slaughter", -103332));
series.add(new DataSeriesItem("Reproduction", +104052));

WaterFallSum end = new WaterFallSum("End");
end.setColor(balanceColor);
end.setIntermediate(false); // Not intermediate (default)
series.add(end);

conf.addSeries(series);
```

18.5. Chart 配置

图表的所有内容配置都定义在 **Configuration** 对象的 **图表模型** 之内。你可以使用 `getConfiguration()` 方法来访问图表模型。

Configuration 类中与配置相关的属性总结如下：

- credits: 类型为 **Credits** (其中包含的属性为 text, position, href, enabled)
- labels: 类型为 **HTMLLabels** (包含的属性为 html, style)
- lang: 类型为 **Lang** (包含的属性为 decimalPoint, thousandsSep, loading)
- legend: 类型为 **Legend** (参见 第 18.5.3 节“图例”)
- pane: 类型为 **Pane**
- plotoptions: 类型为 **PlotOptions** (参见 第 18.5.1 节“绘图选项(Plot Option)”)
- series: 类型为 **Series**
- subTitle: 类型为 **SubTitle**
- title: 类型为 **Title**
- tooltip: 类型为 **Tooltip**
- xAxis: 类型为 **XAxis** (参见 第 18.5.2 节“坐标轴”)
- yAxis: 类型为 **YAxis** (参见 第 18.5.2 节“坐标轴”)

关于数据的配置, 请参见 第 18.6 节“图表数据”。

18.5.1. 绘图选项(Plot Option)

在配置中可以设置绘图选项, 设置对象可以是整个图表, 也可以对各个值序列分别设置。绘图选项中的一部分是各个图表类型独有的, 这类属性定义在各图表类型独有的绘图选项类中, 这些类都继承自 **AbstractPlotOptions**。

你需要创建绘图选项对象, 并通过 `setPlotOptions()` 方法指定它应用于整个图表还是应用于一个值序列。

比如, 以下示例程序将对柱形图(Column Chart)启用堆栈(stack)模式:

```
PlotOptionsColumn plotOptions = new PlotOptionsColumn();
plotOptions.setStacking(Stacking.NORMAL);
conf.setPlotOptions(plotOptions);
```

关于各个图表类型独有的设置选项的详情, 请参照各个图表类型及相应的绘图选项类的 API 文档, 关于各个图表类型共通的设置选项, 请参照 **AbstractPlotOptions** 的 API 文档.

18.5.2. 坐标轴

很多图表类型带有两个坐标轴, X 轴和 Y 轴, 分别由 **XAxis** 和 **YAxis** 类来表达. X 轴通常水平显示, 表示值序列中的各个元素, Y 轴通常垂直显示, 表示值序列中的值. 某些图表类型会反转这两个轴, 也可以在图表配置中使用 `getChart().setInverted()` 方法来显式地反转坐标轴. 坐标轴带有标题, 以及间隔出现的刻度标记, 用于指示数字值或者类别. 某些图表类型, 比如仪表(gauge)图, 只有 Y 轴, 它显示为仪表中的一个圆环, 某些图表类型, 比如饼(pie)图, 没有坐标轴.

坐标轴对象创建后, 需要使用 `addXAxis()` 和 `addYAxis()` 方法添加到配置对象中.

```
XAxis xaxis = new XAxis();
xaxis.setTitle("Axis title");
conf.addXAxis(xaxis);
```

图表可以拥有一个以上的 Y 轴, 通常用于显示不同单位或比例的多个值序列. 值序列与坐标轴之间的关联, 通过值序列对象的 `setYAxis()` 方法来设定.

关于坐标轴各种配置参数的完整信息, 请参照 Vaadin Charts 的 JavaDoc API 文档.

类别

在大多数图表类型中, X 轴默认会显示某种数值间隔的刻度标记和标签. 如果值序列中的元素不是数值, 而是某种符号, 你可以将数据元素关联到类别. 类别的标签将显示在坐标轴的两个刻度标记之间, 并于数据点对齐. 在某些图表中, 比如柱形图(Column Chart), 不同值序列中对应的值会被分组到同一个类别之中. 你可以使用 `setCategories()` 方法来设置类别标签, 这个方法接受的参数是类别的列表, 或者类别的 iterable 容器. 参数中的类别列表应该与值序列中的项目一致.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune");
```

标签

在大多数图表类型中, X 轴默认会显示某种数值间隔的刻度标记和标签. 坐标轴中标签的格式和样式定义在 **Labels** 对象中, 你可以通过坐标轴的 `getLabels()` 方法得到这个对象.

关于标签各种配置参数的完整信息, 请参照 Vaadin Charts 的 JavaDoc API 文档.

坐标轴范围

坐标轴的范围通常会自动设定为适应于数据内容, 但也可以显式设定. 坐标轴配置的 `extremes` 属性定义了坐标轴范围的最小和最大值. 你可以使用 `setMin()` 和 `setMax()` 方法来分别设置这两个值, 也可以使用 `setExtremes()` 方法来同时设置这两个值. 如果通过程序来改变坐标轴范围, 需要使用 `drawChart()` 方法来重绘图表.

18.5.3. 图例

图例是一个方块，用于描述图表中显示的各个值序列。图例默认是启用的，并且会自动显示值序列对象中定义的名称，以及这个值序列在图表中对应的颜色符号。

18.6. 图表数据

图表数据保存在值序列模型中，其中除数据点的值之外，还包含了相关的可视化表现信息。值序列有很多种不同类型 - **DataSeries**, **ListSeries**, **AreaListSeries**, 以及 **RangeSeries**。

18.6.1. List 数据序列

ListSeries 本质上是一个辅助类，它使得简单的连续数值的处理比使用 **DataSeries** 变得更简单一些。数据点在 X 轴上的间隔被假定为固定值，数据点的起始值由 `pointStart` 属性指定（默认为 0），间隔值由 `pointInterval` 属性指定（默认为 1.0）。这两个属性定义在值序列的 **PlotOptions** 中。

Y 轴的值通过 **List<Number>** 来指定，也可以使用数组，或者省略号形式的可变长参数。

```
ListSeries series = new ListSeries(
    "Total Reindeer Population",
    181091, 201485, 188105, ...);
series.getPlotOptions().setPointStart(1959);
conf.addSeries(series);
```

你也可以使用 `addData()` 方法来逐个添加 Y 轴值，从其他表现形式转换为 Y 轴值时，通常使用这个方法。

```
// Original representation
int data[][] = reindeerData();

// Create a list series with X values starting from 1959
ListSeries series = new ListSeries("Reindeer Population");
series.getPlotOptions().setPointStart(1959);

// Add the data points
for (int row[]: data)
    series.addData(data[1]);

conf.addSeries(series);
```

如果图表带有多个 Y 轴，你可以使用 `setyAxis()` 方法来指定值序列对应的 Y 轴，方法参数是所使用的 Y 轴的索引值。

18.6.2. 一般数据序列

DataSeries 可以表达一个数据点的序列，这些数据可以是间隔数据，也可以是散点数据。数据点使用 **DataSeriesItem** 类表达，这个类的 `x` 和 `y` 属性用于表达数据值。每个元素都可以指定一个类别名称。

```
DataSeries series = new DataSeries();
series.setName("Total Reindeer Population");
series.add(new DataSeriesItem(1959, 181091));
series.add(new DataSeriesItem(1960, 201485));
series.add(new DataSeriesItem(1961, 188105));
series.add(new DataSeriesItem(1962, 177206));
```

```
// Modify the color of one point
series.get(1960, 201485)
    .getMarker().setFillColor(SolidColor.RED);
conf.addSeries(series);
```

数据点还会关联到一些可视化表现参数: 标记(marker)的样式, 选中状态, 图例序号, 以及刻度的样式(仅限于仪表图). 这些参数大多可以在值序列的各个元素的层级上配置, 也可以在值序列整体层级, 或图表全体的绘图选项层级上配置. 关于配置选项, 详情请参见 第 18.5 节 “Chart 配置”. 某些参数只在单独的数据元素层级上有效, 比如饼图的 sliced 选项.

添加和删除数据项目

新建的 **DataSeriesItem** 项目可以使用 add() 方法添加到值序列中. 这个方法的基本版本只使用数据项目本身作为参数, 但其他版本还使用两个 boolean 参数. 如果 updateChart 参数为 false, 则图表不会立即被更新. 如果你在一次请求中添加了大量的数据点, 那么这个选项将是很有用的.

shift 参数, 当它为 true 时, 会导致值序列中第一个数据点遵照最优化的原则移动, 因此添加新的数据点时会使图表动态地向左侧移动(译注: 此处不确定, 待校). 这个选项对于平滑间隔的数据最有意义.

你也可以使用值序列的 remove() 方法删除数据点. 数据的删除通常不会使图表发生动态的变更(译注: 此处不确定, 待校), 除非同时添加了其他数据点, 因为 add() 方法的 shift 参数会导致图表的移动.

更新数据项目

如果你更新了 **DataSeriesItem** 对象的属性, 你需要调用值序列的 update() 方法, 并将变更过的数据项目作为参数. 通过这种方法变更一个数据点的坐标值会导致图表中一次动画效果的变更(译注: 此处不确定, 待校).

范围数据

范围图的 Y 值被指定为一对最小值-最大值的形式. **DataSeriesItem** 提供了 setLow() 和 setHigh() 方法, 可以设置数据点的最小值和最大值, 此外还提供了很多构造方法, 可以接受最小值和最大值.

```
RangeSeries series =
    new RangeSeries("Temperature Extremes");

// Give low-high values in constructor
series2.add(new DataSeriesItem(0, -51.5, 10.9));
series2.add(new DataSeriesItem(1, -49.0, 11.8));

// Set low-high values with setters
DataSeriesItem point2 = new DataSeriesItem();
point2.setX(2);
point2.setLow(-44.3);
point2.setHigh(17.5);
series2.add(point2);
```

RangeSeries 提供了一种略为简单的方法来添加范围值的数据点, 详情请参见 第 18.6.3 节 “范围数据序列”.

18.6.3. 范围数据序列

RangeSeries 是一个辅助类, 它继承自 **DataSet** 类, 使得固定间隔的数据的指定可以略为简单一些, 可以通过 Y 轴上的一系列最小值-最大值范围来指定. 你可以在范围图中使用这个值序列, 详情请参见 第 18.4.10 节 “区域范围图(Area Range Chart)与列范围图(Column Range Chart)”.

对于 X 轴, 各数据点的坐标将使用固定间隔值来自动生成, 起点值由 `pointStart` 属性指定(默认为 0), 间隔值由 `pointInterval` 属性指定(默认为 1.0).

设置数据

RangeSeries 中的数据, 通过 Y 轴上的最小值-最大值对的数组来指定. 最小值-最大值对本身也由数组来表达. 你可以在构造函数中指定数据, 也可以使用 `setData()` 方法来指定, 数据使用省略号形式的可变长参数:

```
RangeSeries series =
    new RangeSeries("Temperature Ranges",
        new Double[]{-51.5,10.9},
        new Double[]{-49.0,11.8},
        ...
        new Double[]{-47.0,10.8});
conf.addSeries(series);
```

除了使用可变长参数之外, 你也可以使用数组来传递, 如下例中的 `setData()` 方法调用所示:

```
series.setData(new Double[][] {
    new Double[]{-51.5,10.9},
    new Double[]{-49.0,11.8},
    ...
    new Double[]{-47.0,10.8}});
```

18.6.4. 容器数据序列

ContainerDataSet 是一个适配器(adapter)类, 用于将 Vaadin Container 数据源绑定到图表上. 容器中需要存在对应的属性, 分别定义各个数据点的名称, X 值, Y 值. 这三个属性的默认属性 ID 分别是 "name", "x", 和 "y". 你也可以使用 `setNamePropertyId()`, `setYPropertyId()`, 以及 `setXPropertyId()` 方法来设置对应的属性 ID. 如果容器中不存在 x 属性, 那么数据会被假定为属于不同的类别.

下例中, 我们有一个 **BeanItemContainer** 容器, 其中的元素是 **Planet**, 它带有 name 和 diameter 属性. 我们同时使用 Vaadin **Table** 和图表来显示容器中的数据.

```
// The data
BeanItemContainer<Planet> container =
    new BeanItemContainer<Planet>(Planet.class);
container.addBean(new Planet("Mercury", 4900));
container.addBean(new Planet("Venus", 12100));
container.addBean(new Planet("Earth", 12800));
...

// Display it in a table
Table table = new Table("Planets", container);
table.setPageLength(container.size());
table.setVisibleColumns(new String[]{"name", "diameter"});
layout.addComponent(table);

// Display it in a chart
```

```

Chart chart = new Chart(ChartType.COLUMN);
... Configure it ...

// Wrap the container in a data series
ContainerDataSeries series =
    new ContainerDataSeries(container);

// Set up the name and Y properties
series.setNamePropertyId("name");
series.setYPropertyId("diameter");

conf.addSeries(series);

```

由于 X 轴对应的是类别而不是数字值, 我们需要使用一个字符串数组来设置类别标签. 有几种不同的方式来实现这个设置, 有些方法的效率更高一些, 下面是一种实现方法:

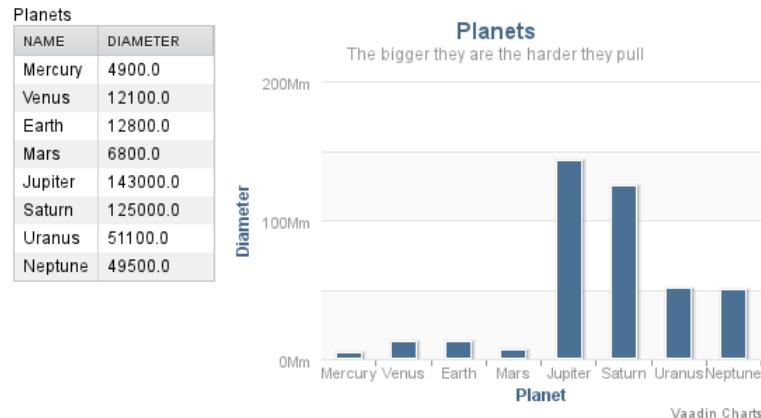
```

// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
String names[] = new String[container.size()];
List<Planet> planets = container.getItemIds();
for (int i=0; i<planets.size(); i++)
    names[i] = planets.get(i).getName();
xaxis.setCategories(names);
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);

```

上例的运行结果参见图 18.15 “绑定到数据容器的 Table 和 Chart”.

图 18.15. 绑定到数据容器的 Table 和 Chart



18.7. 高级使用

18.7.1. 服务器端描绘与导出

除了在 Vaadin UI 中使用图表外, 你还可能会需要将图表输出为图片或可下载的文档. Vaadin Charts 可以在服务器端描绘, 这时需要使用一个 headless 的 JavaScript 执行环境, 比如 PhantomJS.

Vaadin Charts 支持 HighCharts 远程导出服务, 但基于 PhantomJS 的 SVG 生成器更便于使用, 而且支持更强大的功能.

使用远程导出服务

HighCharts 是一个简单的内建导出功能, 它可以在远程服务器上进行导出. HighCharts 提供了一个默认的导出服务, 但你也可以配置构建你自己的导出服务.

你可以在图表配置中使用 `setExporting(true)` 来启用内建的导出功能.

```
chart.getConfiguration().setExporting(true);
```

为了进行更进一步的配置, 你可以提供一个自定义设置过的 **Exporting** 对象.

```
// Create the export configuration
Exporting exporting = new Exporting(true);

// Customize the file name of the download file
exporting.setFilename("mychartfile.pdf");

// Enable export of raster images
exporting.setEnableImages(true);

// Use the exporting configuration in the chart
chart.getConfiguration().setExporting(exporting);
```

如果你只希望启用下载功能, 你可以禁用打印按钮, 如下例:

```
ExportButton printButton = new ExportButton();
printButton.setEnabled(false);
exporting.setPrintButton(printButton);
```

导出功能默认会使用一个 HighCharts 导出服务. 如果希望使用自己的导出服务, 你需要在导出配置中构建并配置自己的导出服务, 如下例:

```
exporting.setUrl("http://my.own.server.com");
```

使用 **SVG** 生成器

Vaadin Charts 中的 **SVGGenerator** 提供了一种高级方式, 可以在服务器端将图表描绘为 SVG 格式. SVG 受到各种应用程序的普遍支持, 可以转换为几乎所有的其他图像格式, 也可以传递给 PDF 报表生成器.

生成器在服务器端使用 PhantomJS 来描绘图表. 你需要从 phantomjs.org 来安装 PhantomJS. 安装完成后, PhantomJS 应该在你的系统路径中. 如果没有, 你可以设置 JRE 的系统属性 `phantom.exec`, 指向 PhantomJS 可执行程序的路径.

要生成字符串形式的 SVG 图像内容(是 XML 格式), 只需要简单地对 **SVGGenerator** 的单例 (`singleton`) 调用 `generate()` 方法, 并将图表配置传递给这个方法作为参数.

```
String svg = SVGGenerator.getInstance()
    .generate(chart.getConfiguration());
```

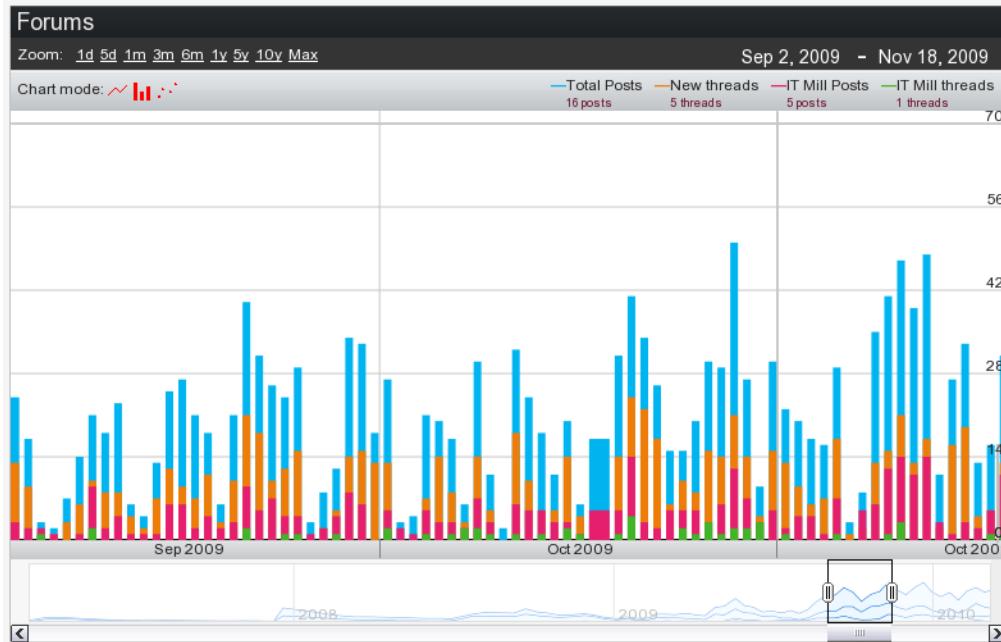
然后你就可以根据自己的需要使用 SVG 图片了, 比如, 通过 **StreamResource** 来下载它, 或者将它嵌入到 HTML, PDF, 或其他文档之内. 你可以使用 SVG 工具, 比如 Batik 或 iText 库来生成文档. 关于这个问题的完整示例程序, 你可以从 Subversion 库中取出 Charts Export Demo 的代码, 地址是 <http://dev.vaadin.com/svn/addons/vaadin-charts/chart-export-demo>.

18.8. 时间线

Timeline 是 Vaadin Charts add-on 中的另一个图表组件, 它与 **Chart** 组件不同. 它的目的是在一个水平的时间线轴上, 向用户直观地展现事件和变化趋势.

Timeline 使用自己独有的数据序列表现形式, 与 **Chart** 不同, 并针对数据的更新做了更多优化. 你可以表达几乎所有的时间相关的统计数据, 数据内容为时刻-数值的映射. 还可以使用多个数据源来实现不同数据之间的比较.

图 18.16. Timeline 组件



时间线图表可以将时间相关数据可视化地展现为图像而不是数字. 这种图表可以广泛应用于商业, 科学, 技术等等几乎所有领域, 比如在项目管理中, 可用来展现里程碑(milestone)和目标(goal), 在地质领域中, 可用来展现历史事件, 时间线图表最重要的用途可能是股票市场.

使用时间线图表, 你可以展现几乎所有的时间相关的统计数据, 数据内容为时刻-数值的映射. 还可以使用多个数据源来实现多种数据之间的比较. 它使得用户更容易地领会数据的变化, 并更容易地预测未来趋势和问题.

18.8.1. 图像类型

Vaadin Timeline 支持三种图像类型:

线图(*Line graphs*)

适合展现连续数据, 比如温度变化, 或股票变化.

条形图(*Bar graphs*)

适合展现离散的, 不连续的数据, 比如市场占有率或论坛中的消息数.

散点图(*Scatter graphs*)

适合于展现离散的, 不连续的数据.

如果你的时间线图表中存在多个图像，你也可以让他们堆积在一起，而不是分别描绘，方法是将 **Timeline** 的 `setGraphStacking()` 设置为 `true`.

18.8.2. 用户操作元素

用户可以通过几种方式与 Vaadin Timeline 进行交互.

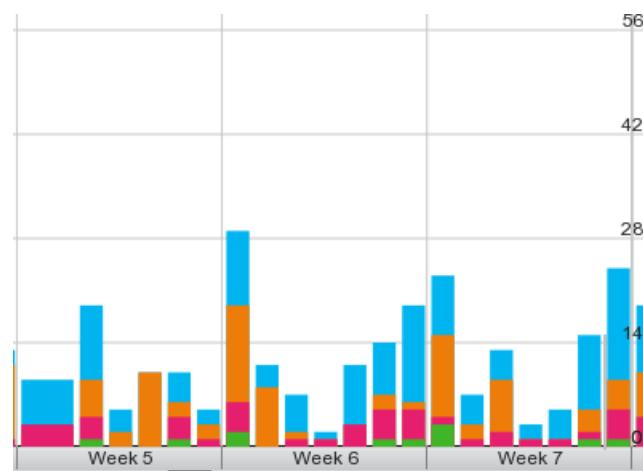
在时间线图表的底部有一个 滚动条区域，你可以在此处拖动时间范围方块，或点击向左或向右按钮，向前或向后滚动时间. 你还可以在滚动条区域中拖动时间范围方块的大小来改变时间范围. 当鼠标位于组件内部时你还可以使用鼠标滚轮来缩放图表大小.

图 18.17. 滚动条区域



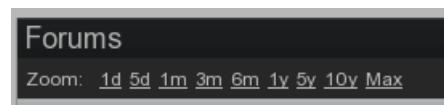
时间线图表的中间区域是 主区域，被选中的时间范围内的数据就显示在这个区域内. 时间比例尺显示在主区域下方. 具体使用哪个时间比例尺，取决于缩放级别，可用的时间单位包括“小时”到“年”. 数值比例尺显示在主区域右侧. 数值比例尺可以是一个静态的值范围，也可以是由被显示的数据计算得到的值范围. 用户可以使用鼠标向左或向右拖动主区域，来在时间中移动，也可以使用鼠标滚轮来缩放图表.

图 18.18. 主区域



你可以使用时间线图表上方的按钮来选择一个 预设定的缩放级别. 这时会改变显示的时间范围来与选中的缩放级别保持一致. 缩放级别可以通过 API 来定制，以便适应时间范围.

图 18.19. 预设定的缩放按钮



当前时间范围 显示在组件的右上角. 点击日期后可以编辑日期，然后你就可以手动变更时间范围. 图例显示在时间范围下方. 图例用来解释各种数据在图表中分别由哪个线条来表示，而且用户将鼠标移动到图表上时，图例中还会显示各个项目当前的数值.

图 18.20. 当前时间范围, 以及图例



最后, 可用的 图表模式显示在预定义缩放级别选项的下方. 可用的图表模式可以通过 API 来设定.

图 18.21. 图表模式



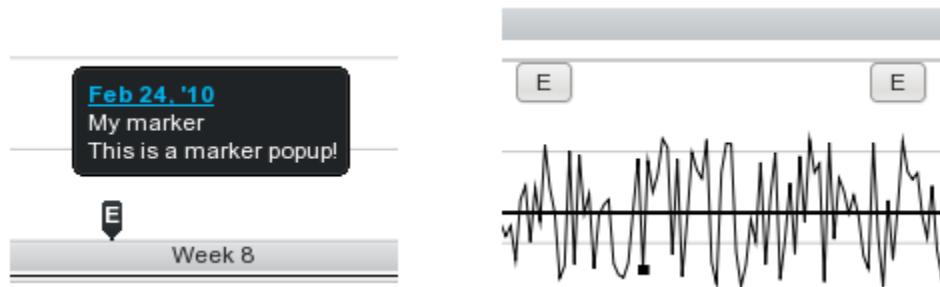
你可以按照自己的需要, 显示或者隐藏上述各种功能. 比如, 如果你只需要显示图表, 而不需要任何控制, 你可以使用 API 来隐藏上述所有功能

18.8.3. 事件标记

除图表外, 时间线图表还可以带有事件. 比如, 事件可以是广告的播放时间, 图表则显示网站的点击次数. 事件数据与图表结合在一起, 就使得用户可以可视化地查看广告与网站点击数之间的关联.

Vaadin Timeline 提供两种类型的事标记, 见 图 18.22 “时间线图表的事件标记”.

图 18.22. 时间线图表的事件标记



(左图) 标记带有自定义标记符号, 比如, 字母 'E'. 当用户将鼠标移动到事件上时, 标记会显示一个标题.

(右图) 标记外观类似按钮, 带有标记符号和标题.

18.8.4. 效率

Vaadin Timeline 使用两种方法来减少服务器与客户端之间的通信. 首先, 组件内展现的所有数据会根据需要动态地从服务器端获取. 也就是说, 当用户在时间线视图内滚动时, 组件会连续不断地从服务器端取得数据. 而且, 只有对于用户可见的那部分数据才会传送到客户端. 比如, 如果时间线图表中包含的数据每秒测量1次并持续一整年, 那么并不是所有数据都会发送到客户端. 只有将被描绘到屏幕上而且不会被遮盖的部分数据才会传送. 这个特性将会确保, 即使对于非常巨大的数据集合, 画面的装载时间也是很短的, 并且只有必须的数据才会通过网络进行传输.

第二, Vaadin Timeline 会在浏览器中缓存从服务器端接收的数据, 因此会尽量让数据在网络上只传输一次. 这个功能将提高时间范围浏览的速度, 因为数据可以从缓存中取得, 而不必通过网络再次获取.

18.8.5. 对数据源的要求

Vaadin Timeline 使用 Vaadin 容器作为图表和事件的数据源. 但是, 它对于容器存在一些要求, 容器需要满足这些要求才能与 Vaadin Timeline 兼容.

容器必须实现 `Container.Indexed` 接口, Vaadin Timeline 才可以使用这样的容器. 这是因为 Vaadin Timeline 会在需要时动态地从服务器端获取数据. 通过这种方式就可以使用巨大的数据集, 而不必在客户端一次性装载所有数据, 因此实现了巨大的性能提升.

另外一个要求是, 容器需要有一个属性类型为 **`java.util.Date`** (或者可以转换为 Date 的其他类型), 这个属性中需要包含数据点或事件发生时的时间戳. 这个属性需要通过 **Timeline** 的 `setGraphTimestampPropertyId()` 方法来设置. 如果没有设置 timestamp-property ID, 将会使用默认的属性 ID `Timeline.PropertyId.TIMESTAMP`.

图表容器还需要有一个 **值** 属性, 它负责定义数据点的值. 这个值可以是任意的数字值. 值属性可以通过 **Timeline** 的 `setGraphValuePropertyId()` 方法来设置. 如果没有设置值属性, 将会使用默认的属性 ID `Timeline.PropertyId.VALUE`.

下例演示如何构建一个图表容器:

```
// Construct a container which implements Container.Indexed
IndexedContainer container = new IndexedContainer();

// Add the Timestamp property to the container
Object timestampProperty = "Our timestamp property";
container.addContainerProperty(timestampProperty,
                               java.util.Date.class, null);

// Add the value property
Object valueProperty = "Our value property";
container.addContainerProperty(valueProperty, Float.class, null);

// Our timeline
Timeline timeline = new Timeline();

// Add the container as a graph container
timeline.addGraphDataSource(container, timestampProperty,
                             valueProperty);
```

事件和标记的容器也与此类似. 它们都需要 `timestamp` 属性, 类型应该为 **`java.util.Date`**, 以及 `caption` 属性, 类型应该为字符串. 标记容器还需要一个 `value` 属性, 它将显示在标记的弹出区域中.

下例演示如何构建一个标记容器或事件容器:

```
// Create the container
IndexedContainer container = new IndexedContainer();

// Add the timestamp property
container.addContainerProperty(Timeline.PropertyId.TIMESTAMP,
                               Date.class, null);

// Add the caption property
container.addContainerProperty(Timeline.PropertyId.CAPTION,
```

```
String.class, "");  
  
// Add the marker specific value property.  
// Not needed for a event containers.  
container.addContainerProperty(Timeline.PropertyId.VALUE,  
    String.class, "");  
  
// Create the timeline with the container as both the marker  
// and event data source  
Timeline timeline = new Timeline();  
timeline.setMarkerDataSource(container,  
    Timeline.PropertyId.TIMESTAMP,  
    Timeline.PropertyId.CAPTION,  
    Timeline.PropertyId.VALUE);  
  
timeline.setEventDataSource(container,  
    Timeline.PropertyId.TIMESTAMP,  
    Timeline.PropertyId.CAPTION);
```

上例中使用的是默认属性 ID, 你也可以根据自己的需求改变属性 ID.

Timeline 会监听容器中的数据变化, 并更新相应的图表. 当它更新图表时, 如果容器内的数据有添加或删除, 图表中当前选中的日期范围会继续保持选中. 浏览区中的选择跳会移动到正确位置, 保证当前的选择区域继续被选中. 如果你希望在容器内容变化时改变选择区域, 并保持选择区域固定, 你可以禁止选择区域的锁定, 方法是 `setBrowserSelectionLock()` 为 `false`.

18.8.6. 事件与监听器

使用 Vaadin Timeline 时可以使用两种类型的事件.

日期范围变更

当用户移动日期范围选择器, 拖动时间线, 或者手工输入新日期时, 将会修改选中的日期范围, 此时将会向服务器发送一个事件, 其中的信息是当前显示的日期范围是什么. 要监听这个事件, 你可以绑定一个 **DateRangeListener**, 它将会收到当前选中的开始日期和结束日期.

事件的点击

如果时间线中带有事件, 你可以添加一个 **EventClickListener** 来监听事件上的点击. 这个监听器将收到项目 ID 的列表, 其中内容就是被点击的事件在事件数据源中对应的项目 ID. 如果空间不足以显示所有事件, 多个事件可以组合为单个事件图标, 这种情况下, 用户的点击就会关联到多个事件的项目 ID.

18.8.7. 可配置性

Vaadin Timeline 是高度可定制的, 它的外观可以很容易的修改, 以符合你的需求. Timeline 的默认视图包含了所有可用的控制元素, 但它们通常是不需要的, 因此可以隐藏起来.

以下列表包含了你可以控制显示或隐藏的组件:

- 图表模式
- 文字输入的日期选择
- 浏览区(Timeline 的底部)
- 图例

- 缩放级别
- 标题

浏览区和主视图区的外观也可以改变. 以下是可以通过 API 修改的设定项目:

- 图表轮廓线颜色
- 图表轮廓线宽度
- 图表caps (仅对折线图有效)
- 图表填充颜色
- 图表是否可见
- 图表阴影

组件外观的其他变更可以使用 CSS 很容易地实现.

缩放级别也是完全可以定制的. 缩放级别定义为毫秒单位的值, 可以通过 `addZoomLevel()` 方法来添加. 缩放级别一定带有一个标题, 标题将成为缩放控制面板上可见的部分, 缩放级别还带有一个毫秒单位的值.

默认情况下, 图表被一个表格分割为大小相等的五个部分, 表格线条为灰色. 但是你可以使用 `setGridColor()` 和 `setVerticalGridLines()` 方法来自定义表格的描绘方式.

18.8.8. 本地化

Vaadin Timeline 默认使用英语来显示标题, 并使用应用程序的默认语言环境(locale)来显示日期.

你可以使用以下方法来修改各个标题:

- `setZoomLevelsCaption()` -- 设置显示在缩放级别之前的标题
- `setChartModesCaption()` -- 设置显示在图表模式之前的标题

此外, 你还可以修改时间线在水平标尺上显示日期时使用的语言环境(locale), 方法是通过 `setLocale()` 设定一个有效的语言环境(locale).

你还可以配置水平标尺中的日期以及右上角日期选择框的显示格式, 方法是使用 `getDateFormat()`-方法, 它将返回一个 **DateFormatInfo** 对象. 通过这个对象的属性, 你可以指定每个日期范围的显示格式. 请注意, 如果你使用长日期格式, 那么当标尺的空间不足以显示格式化之后的完整日期时, 日期文字可能会被截断一部分.

18.8.9. 时间线图表的教程

在以下教程中, 我们会一步一步演示如何创建一个时间线图表.

创建数据源

为了使用 Timeline, 你需要为它创建一些数据源. Timeline 使用 `Container.Indexed` 容器作为图表, 标记, 以及事件的数据源. 所以我们首先创建一个数据源, 来表示我们希望在时间线中显示的图表.

为了让 Timeline 理解容器内数据的结构，我们需要使用特别的属性 ID，这些属性 ID 描述了各个属性所表达的数据类型。为了让 Vaadin Timeline 正确工作，我们需要添加两个属性 ID，其中一个用于取得数值，另一个用于数值本身。Vaadin Timeline 预定义了这两个属性 ID: `Timeline.PropertyId.TIMESTAMP` 和 `Timeline.PropertyId.VALUE`。你可以使用这些预定义的属性 ID，也可以创建自己的属性 ID。

下面，我们来创建符合上述规则的容器。当我们创建工程时会自动创建一个主 UI 类，请打开这个主 UI 类，并添加以下方法。

```
/**
 * Creates a graph container with a month of random data
 */
public Container.Indexed createGraphDataSource() {

    // Create the container
    Container.Indexed container = new IndexedContainer();

    // Add the required property ids (use the default ones here)
    container.addContainerProperty(Timeline.PropertyId.TIMESTAMP,
        Date.class, null);
    container.addContainerProperty(Timeline.PropertyId.VALUE,
        Float.class, 0f);

    // Add some random data to the container
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.MONTH, -1);
    Date today = new Date();
    Random generator = new Random();

    while(cal.getTime().before(today)) {
        // Create a point in time
        Item item = container.addItem(cal.getTime());

        // Set the timestamp property
        item.getItemProperty(Timeline.PropertyId.TIMESTAMP)
            .setValue(cal.getTime());

        // Set the value property
        item.getItemProperty(Timeline.PropertyId.VALUE)
            .setValue(generator.nextFloat());

        cal.add(Calendar.DAY_OF_MONTH, 1);
    }

    return container;
}
```

这个方法将创建一个有索引的容器，并在其中填充一些随机数据点。你可以看到，我们使用了 **IndexedContainer**，并为它定义了前面介绍过的那两个属性。然后我们向容器中添加了一些随机数据。这里我们为时间戳和数值属性使用了默认属性 ID，但你也可以使用独自的属性 ID。后面我们会介绍，当使用独自的属性 ID 时，如何告诉 Timeline 应该使用哪个属性 ID。

下一步，向我们的图表中添加一些标记(Marker)。标记是类似箭头的形状，显示在时间线的底部，通过标记我们可以标注出这个时刻发生的事件。要创建标记，你也需要创建数据源。我们首先向你演示创建数据源的代码，然后再解释这些代码的含义。请添加以下方法到 UI 类中：

```
/**
 * Creates a marker container with a marker for each seven days
```

```
/*
public Container.Indexed createMarkerDataSource() {

    // Create the container
    Container.Indexed container = new IndexedContainer();

    // Add the required property IDs (use the default ones here)
    container.addContainerProperty(Timeline.PropertyId.TIMESTAMP,
        Date.class, null);
    container.addContainerProperty(Timeline.PropertyId.CAPTION,
        String.class, "Our marker symbol");
    container.addContainerProperty(Timeline.PropertyId.VALUE,
        String.class, "Our description");

    // Add a marker for every seven days
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.MONTH, -1);
    Date today = new Date();
    SimpleDateFormat formatter =
        new SimpleDateFormat("EEE, MMM d, ''yy");
    while(cal.getTime().before(today)){
        // Create a point in time
        Item item = container.addItem(cal.getTime());

        // Set the timestamp property
        item.getItemProperty(Timeline.PropertyId.TIMESTAMP)
            .setValue(cal.getTime());

        // Set the caption property
        item.getItemProperty(Timeline.PropertyId.CAPTION)
            .setValue("M");

        // Set the value property
        item.getItemProperty(Timeline.PropertyId.VALUE).
            setValue("Today is "+formatter.format(cal.getTime()));

        cal.add(Calendar.DAY_OF_MONTH, 7);
    }

    return container;
}
```

和图表数据容器的示例一样，这里我们首先创建了一个有索引的容器。注意，使用时间线图表组件时，所有的容器都必须是有索引的容器。

然后我们添加了时间戳属性，标题属性，以及数值属性。

时间戳属性与图表数据容器中一样，但标题与数值属性不同。标题属性描述标记的类型。标题将在时间线图表中显示在箭头图形的上方，因此它应该是一个很短的符号，最好只有一个字符长。标题属性的类型必须是字符串。

数值属性也应该是字符串，当用户将鼠标移动到标记上时，数值属性将会显示。数值属性的字符串可以为任意长度，其内容通常应该是对标记的某种描述。

第三个数据源是事件数据源。事件将显示在时间线图表的顶部，它支持分组，而且可以点击。事件在时间线图表中显示为类似按钮的图标。

事件数据源与标记数据源几乎完全一样，区别只是没有数值属性。下面我们来创建一个事件数据源，并为我们图表中的每一个星期日添加一个事件：

```

/**
 * Creates a event container with a marker for each sunday
 */
public Container.Indexed createEventDataSource() {

    // Create the container
    Container.Indexed container = new IndexedContainer();

    // Add the required property IDs (use the default ones here)
    container.addContainerProperty(Timeline.PropertyId.TIMESTAMP,
        Date.class, null);
    container.addContainerProperty(Timeline.PropertyId.CAPTION,
        String.class, "Our marker symbol");

    // Add a marker for every seven days
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.MONTH, -1);
    Date today = new Date();
    while(cal.getTime().before(today)){
        if(cal.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY){
            // Create a point in time
            Item item = container.addItem(cal.getTime());

            // Set the timestamp property
            item.getItemProperty(Timeline.PropertyId.TIMESTAMP)
                .setValue(cal.getTime());

            // Set the caption property
            item.getItemProperty(Timeline.PropertyId.CAPTION)
                .setValue("Sunday");
        }
        cal.add(Calendar.DAY_OF_MONTH, 1);
    }

    return container;
}

```

你可以看到，事件数据源与标记数据源的区别不大。但在使用中它们是不同的，事件可以比较紧密地排列在一起，但不影响使用。你可以向事件添加点击监听器，然后就可以捕捉到用户点击的事件。关于点击监听器，将在后面详细介绍。

现在我们已经有了三个数据源，可以在我们的应用程序中显示了。下一节，我们会将这些数据源使用到我们的时间线图表中，看看它们如何与时间线图表集成在一起。

创建时间线图表

现在我们已经有了数据源，让我们来看看我们的 Vaadin 应用程序的初始化方法。我们首先创建我们的时间线图表，因此，请在 **MytimelinedemoApplication** 的初始化方法的最后添加以下代码：

```
Timeline timeline = new Timeline("Our timeline");
timeline.setWidth("100%");
```

这段代码将创建我们需要的时间线图表，并设定其宽度为 100%。现在我们将数据源添加到时间线图表中：

```
timeline.addGraphDataSource(createGraphDataSource(),
    Timeline.PropertyId.TIMESTAMP,
    Timeline.PropertyId.VALUE);
```

```

        timeline.setMarkerDataSource(createMarkerDataSource(),
                                     Timeline.PropertyId.TIMESTAMP,
                                     Timeline.PropertyId.CAPTION,
                                     Timeline.PropertyId.VALUE);

        timeline.setEventDataSource(createEventDataSource(),
                                   Timeline.PropertyId.TIMESTAMP,
                                   Timeline.PropertyId.CAPTION);
    
```

最后, 将时间线图表添加到 UI 中. 下面是完整的初始化方法:

```

@Override
protected void init(VaadinRequest request) {
    VerticalLayout content = new VerticalLayout();
    setContent(content);

    // Create the timeline
    Timeline timeline = new Timeline("Our timeline");

    // Create the data sources
    Container.Indexed graphDS = createGraphDataSource();
    Container.Indexed markerDS = createMarkerDataSource();
    Container.Indexed eventDS = createEventDataSource();

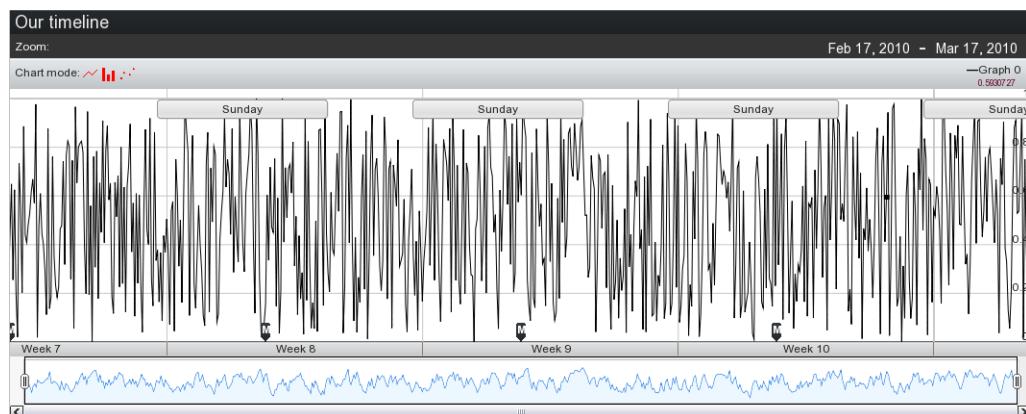
    // Add our data sources
    timeline.addGraphDataSource(graphDS,
                                Timeline.PropertyId.TIMESTAMP,
                                Timeline.PropertyId.VALUE);
    timeline.setMarkerDataSource(markerDS,
                                Timeline.PropertyId.TIMESTAMP,
                                Timeline.PropertyId.CAPTION,
                                Timeline.PropertyId.VALUE);
    timeline.setEventDataSource(eventDS,
                                Timeline.PropertyId.TIMESTAMP,
                                Timeline.PropertyId.CAPTION);

    content.addComponent(timeline);
}

```

现在你应该可以启动应用程序, 并在时间线图表中浏览了. 运行结果见图 18.23 “时间线图表示例应用程序”.

图 18.23. 时间线图表示例应用程序



最终调整

现在我们的时间线图表创建完成了，我们可能会希望对它略微进行一些定制。可定制的内容有很多，但我们首先只为我们的图表添加一些样式属性，并在图例部分添加一个标题。实现代码如下：

```
// Set the caption of the graph
timeline.setGraphLegend(graphDataSource, "Our cool graph");

// Set the color of the graph
timeline.setGraphOutlineColor(graphDataSource, Color.RED);

// Set the fill color of the graph
timeline.setGraphFillColor(graphDataSource, new Color(255,0,0,128));

// Set the width of the graph
timeline.setGraphOutlineThickness(2.0);
```

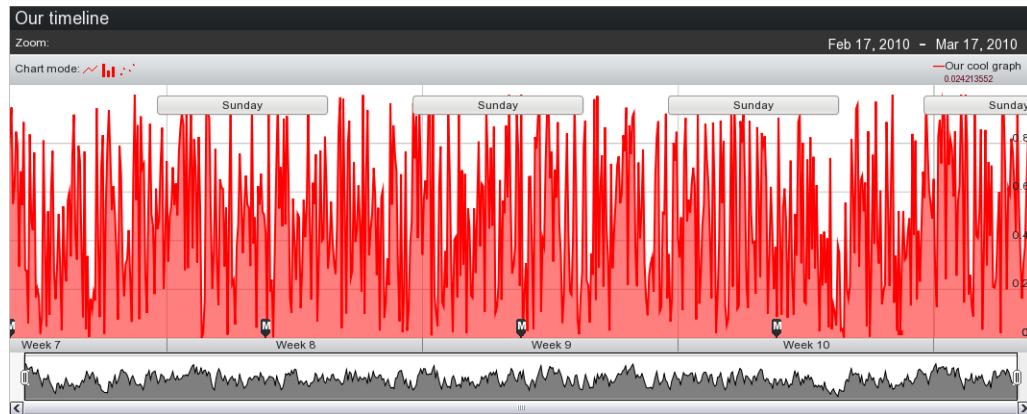
下面我们对浏览区的图表也做类似设置：

```
// Set the color of the browser graph
timeline.setBrowserOutlineColor(graphDataSource, Color.BLACK);

// Set the fill color of the graph
timeline.setBrowserFillColor(graphDataSource,
    new Color(0,0,0,128));
```

运行结果如下：

图 18.24. 控制时间线图表的样式



好的，现在图表看起来已经不同了，但还缺少了一些东西。请看图表左上角，那里没有任何缩放级别存在。时间线图表的默认设定就是没有缩放级别，因此我们需要根据自己的需要创建缩放级别。由于我们处理的是一个月的数据，所以我们创建一日、一周、以及一月的缩放级别。缩放级别是按毫秒单位定义的，因此我们需要计算各个缩放级别等于多少毫秒。添加缩放级别的实现代码如下：

```
// Add some zoom levels
timeline.addZoomLevel("Day", 86400000L);
timeline.addZoomLevel("Week", 7 * 86400000L);
timeline.addZoomLevel("Month", 2629743830L);
```

还记得我们前面添加的事件吗？你可以在图表中看到这些事件，但它们的功能还有一点不完整。我们可以向图表添加事件监听器，当用户点击事件按钮时，就会发送一个点击事件。为了演示这个功能，我们添加一个事件监听器，它会告诉用户，按下的“星期日按钮”代表的是哪一个日期。代码如下：

```
// Listen to click events from events
timeline.addListener(new Timeline.EventClickListener() {
    @Override
    public void eventClick(EventButtonClickEvent event) {
        Item item = eventDataSource.getItem(event.getItemId());
        Iterator<Object> iterator = item.iterator();
        iterator.next();
        Date sunday = (Date) item.getItemProperty(
            Timeline.PropertyId.TIMESTAMP).getValue();
        SimpleDateFormat formatter =
            new SimpleDateFormat("EEE, MMM d, ''yy");
        Notification.show(formatter.format(sunday));
    }
});
```

现在你可以试着点击一下事件，看看会发生什么效果！

以下就是示例应用程序的最终执行结果，你自己的程序运行结果可能会略有不同，因为程序中使用的是随机数据。

图 18.25. 示例程序最后效果



现在，我们希望你基本理解了时间线图表的工作原理，以及定制它的方法。此外还有一些功能我们在这个教程中没有介绍，比如在时间线图表中隐藏不必要的组件，以及向它添加多个图表，但这些功能都非常简单明了，因此你只要阅读相关的 JavaDoc 应该就能够理解了。

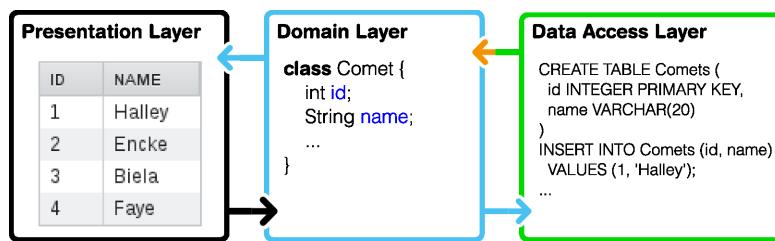
Vaadin JPACContainer

19.1. 概述	499
19.2. 安装	502
19.3. 定义业务数据模型(Domain Model)	505
19.4. JPACContainer 的基本使用	508
19.5. 实体提供者(Entity Provider)	513
19.6. 在 JPACContainer 中过滤	516
19.7. 使用 Criteria API 进行查询	517
19.8. Form 的自动生成	518
19.9. JPACContainer 与 Hibernate 的结合使用	520

本章介绍 Vaadin JPACContainer add-on 的使用.

19.1. 概述

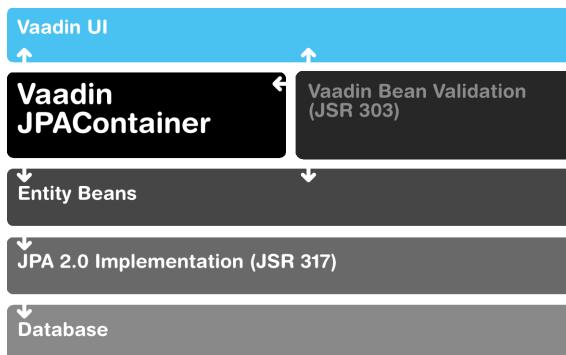
使用 Vaadin JPACContainer add-on, 可以很便利地通过 Java Persistence API (JPA) 来实现 UI 组件与数据库的绑定. JPACContainer 是 Container 接口的一个实现, 关于这个接口参见 第 8.5 节“在容器(Container)中保存项目(Item)”. JPACContainer 支持典型的三层应用程序架构, 它是用户界面层与数据访问层之间的 业务数据模型.

图 19.1. 使用 **JPACContainer** 和 **JPA** 的三层架构

Java Persistence API 的角色是负责处理业务数据模型在数据库中的持久化存储. 数据库通常是关系型数据库. Vaadin JPACContainer 将 UI 组件绑定到业务数据模型上, 并负责使用 JPA 来透明地处理数据库访问.

JPA 本身仅仅只是一个 API 定义, 存在很多不同的 JPA 实现. Vaadin JPACContainer 特别支持的是 EclipseLink(JPA 的参考实现) 和 Hibernate. 其它的兼容实现也应该能正常工作. 使用 JPACContainer 的应用程序的架构参见 图 19.2 “JPACContainer 架构”.

图 19.2. JPACContainer 架构



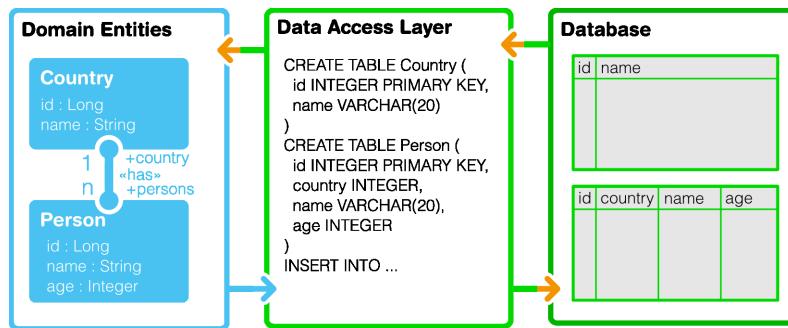
Vaadin JPACContainer 还整合了 Vaadin 支持的 Java Bean Validation (JSR 303).

Java Persistence API

Java Persistence API (JPA) 是一组 Object-Relational Mapping (ORM) API, 用于实现 Java 对象到关系型数据库之间的映射. 在 JPA 中, 更一般地说, 在实体-关系模型中, Java 类被看作 实体. 类(或者说实体)的实例对应到数据库表中的一行, 类的成员变量对应到数据库表中的列. 实体与其他实体之间也可以存在关联关系.

对象-关系映射请参见 图 19.3 “对象(Object)-关系(Relational) 映射”, 图中存在 两个实体, 它们之间存在一对多关系.

图 19.3. 对象(Object)-关系(Relational) 映射



实体之间的关系是使用元数据来声明的。使用 Vaadin JPACContainer，你可以在实体类中通过注解来提供元数据。JPA 实现库会使用反射来读取注解，并根据类定义自动产生对应的数据库模型。业务数据模型的定义及相关注解，请参见第 19.3.1 节“持久化元数据”。

JPA 中的主要接口是 `EntityManager`，通过它可以发起各种查询，查询语言可以使用 Java Persistence Query Language (JPQL)，原生 SQL，或者 JPA 2.0 的 Criteria API。你可以直接使用这个接口，同时只使用 Vaadin JPACContainer 来实现数据与 UI 的绑定。

Vaadin JPACContainer 支持 JPA 2.0 (JSR 317)。它的许可协议是 Apache License 2.0。

JPACContainer 中的基本概念

JPACContainer 是 Vaadin Container 接口的一个实现，因此你可以将它绑定到 UI 组件，比如 **Table**, **ComboBox**, 等等。

对持久化实体类的数据访问由 **实体提供者(entity provider)** 负责处理，定义在 `EntityProvider` 接口中。JPACContainer 为各种不同的使用场景和优化目的提供了多种不同的entity provider。关于内建的entity provider 请参见第 19.5 节“实体提供者(Entity Provider)”。

JPACContainer 默认是无缓冲的，因此当你对一个属性调用 `setValue()` 方法时，或者当用户编辑了一个与它绑定的 Field 组件时，实体的属性变更会立即写入数据库中。容器可以被设置为有缓冲的，这时数据变更会在调用 `commit()` 方法时写入数据库。缓冲可以发生在元素(item)级别，比如对元素属性值的更新进行缓冲，也可以发生在容器级别，比如对元素的增加和删除进行缓冲。只有可以批量处理的容器，也就是，使用可以批量处理的 entity provider 的容器，才可以进行缓冲。注意，如果两个用户可以同时更新同一个实体，并可能导致问题时，推荐使用缓冲功能。在无缓冲的容器中，在写入更新之前会先刷新实体，因此后一次写会胜利，并且同时更新冲突会被写入，然后会发生数据丢失(译注：此段意义不明，待校)。当两个用户编辑同一个元素时，有缓冲的容器会抛出 **OptimisticLockException** 异常，(而无缓冲的容器则不会抛出这个异常)，因此可以使用应用程序逻辑来处理这种异常状况。

文档与支持

除本书的这一章之外，安装包中还包含了关于 JPACContainer 的以下文档：

- API 文档
- JPACContainer 教程
- JPACContainer AddressBook 示例程序
- JPACContainer 示例程序

19.2. 安装

Vaadin JPAContainer 可以使用安装包进行安装, 安装包可通过 Vaadin Directory 下载, 也可以作为 Maven 依赖项目进行安装. 你可以使用 Maven archetype 来创建一个新的、支持 JPAContainer 的 Vaadin 工程.

19.2.1. 下载安装包

Vaadin JPAContainer 可从 Vaadin Directory 下载. 从 Vaadin Directory 下载安装包的基本操作请参见第 17.2 节“通过 Vaadin Directory 下载 Add-on”. 下载页面中还列出了使用 Maven 来取得这个库时需要使用的依赖项声明.

JPAContainer 是一个纯服务器端组件, 因此其中不包括需要你编译的 Widget Set.

19.2.2. 安装包的内容

解开到本地文件夹之后, 安装目录中的内容如下:

README

Readme 文件, 描述安装包的内容.

LICENSE

本库完整的许可协议文本.

vaadin-jpacontainer-3.x.x.jar

Vaadin JPAContainer 库文件.

vaadin-jpacontainer-3.x.x-sources.jar

本库源代码的 JAR 文件. 比如在 Eclipse 中, 你可以在你的工程的编译路径设定中, 将这个 JAR 文件与 JPAContainer JAR 文件关联在一起.

jpacontainer-tutorial.pdf

PDF 格式的教程.

jpacontainer-tutorial-html

HTML 格式的教程.

jpacontainer-addressbook-demo

本教程中将要介绍的 JPAContainer AddressBook 示例程序工程. 你可以使用 "**mvn package**" 命令编译这个工程并打包为一个 WAR, 也可以使用 "**mvn jetty:run**" 命令在 Jetty Web 服务器中执行它. 你也可以将这个示例工程导入到 Eclipse 中.

19.2.3. 使用 Maven 下载

Vaadin Directory 中的下载页面 给出了使用 Maven 来取得 Vaadin JPAContainer 库时需要的依赖项声明 .

```
<dependency>
  <groupId>com.vaadin.addon</groupId>
  <artifactId>jpacontainer-addon</artifactId>
  <version>3.1.0</version>
</dependency>
```

使用 LATEST 版本标记可以自动下载最新的稳定发布版本, 也可以向上例一样使用某个指定的版本号.

关于如何通过 Maven 使用 Vaadin add-on, 详情请参见 第 17.4 节 “在 Maven 工程中使用 Add-on”.

使用 **Maven Archetype**

如果你希望通过 Maven 来创建新的使用 JPAContainer 的 Vaadin 工程, 你可以使用 `vaadin-archetype-jpacontainer archetype`. 关于如何使用 Maven archetype 来创建 Vaadin 工程, 详情请参见 第 2.6 节 “通过 Maven 使用 Vaadin”.

19.2.4. 将库添加到你的工程中

Vaadin JPAContainer JAR 必须包括在 Web 应用程序的库目录中. 这个目录也就是 Web 应用程序中的 `WEB-INF/lib` 目录. 在通常的 Eclipse Web 工程中, 路径为 `WebContent/WEB-INF/lib`. 在 Maven 工程中, 只要正确地定义了依赖项目, JAR 文件就会被自动包含到目录中.

你将需要以下 JAR 文件:

- Vaadin Framework 库
- Vaadin JPAContainer
- Java Persistence API 2.0 (`javax.persistence` 包)
- JPA 实现库(EclipseLink, Hibernate, ...)
- 数据库驱动, 或内嵌式数据库 (H2, HSQLDB, MySQL, PostgreSQL, ...)

如果你使用 Eclipse, Vaadin Plugin for Eclipse 会帮助你自动下载并更新 Vaadin Framework 库.

为了使用 bean 校验功能, 你需要 Bean Validation 的实现库, 比如 Hibernate Validator.

19.2.5. 持久化配置

持久化配置由 `persistence.xml` 文件实现. 在通常的 Eclipse 工程中, 这个文件应该在 `WebContent/WEB-INF/classes/META-INF` 目录中. 在 Maven 工程中, 它应该在 `src/main/resources/META-INF` 目录中. 配置包含以下内容:

- 持久化单元(persistence unit)
- 持久化提供者(persistence provider)
- 数据库驱动程序及连接配置
- 日志

`persistence.xml` 文件在 WAR 中会被打包为 `WEB-INF/classes/META-INF/persistence.xml`. 在 Maven 编译的打包阶段会自动进行这个工作.

持久化配置 XML 文件的 Schema

`persistence.xml` 文件的开始部分定义所使用的 schema 和 namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
```

```
http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
```

定义持久化单元(**Persistence Unit**)

持久化定义的根元素是 persistence-unit. 为了通过 **JPAContainerFactory** 创建 **JPAContainer** 实例, 需要定义持久化单元的名称. 详情请参见 第 19.4.1 节 “使用 **JPAContainerFactory** 创建 **JPAContainer**”. 创建 JPA 实体管理器时, 也需要持久化单元的名称.

```
<persistence-unit name="addressbook">
```

持久化提供者(Persistence provider) 是指所使用的 JPA 提供者实现类. 比如, JPAContainer AddressBook 示例程序使用 EclipseLink JPA, 定义如下:

```
<provider>
    org.eclipse.persistence.jpa.PersistenceProvider
</provider>
```

持久化类需要使用 <class> 元素列出. 或者, 你也可以在持久化中允许包含未列出的类, 方法是覆盖 exclude-unlisted-classes 的默认设定, 如下:

```
<exclude-unlisted-classes>false</exclude-unlisted-classes>
```

各 JPA 提供者专有的参数在 properties 元素中给出.

```
<properties>
    ...
```

后续小节中我们将给出 JPAContainer AddressBook 示例程序所使用的 EclipseLink JPA 和 H2 的相关参数. 关于完整的参数细节, 详情请参见你使用的 JPA 提供者的文档.

数据库连接

EclipseLink 允许使用 JDBC 作为数据库连接. 比如, 如果我们使用 H2 数据库, 驱动程序定义如下:

```
<property name="eclipselink.jdbc.platform"
    value="org.eclipse.persistence.platform.database.H2Platform"/>
<property name="eclipselink.jdbc.driver"
    value="org.h2.Driver" />
```

数据库连接以 URL 形式指定. 比如, 使用嵌入式 H2 数据库, 数据存储在 home 目录, 数据库连接 URL 定义如下:

```
<property name="eclipselink.jdbc.url"
    value="jdbc:h2:~/my-app-h2db"/>
```

提示: 在 Eclipse 中开发 Vaadin 应用程序时, 如果使用嵌入式 H2 数据库, 你可能会希望在数据库连接 URL 中添加 ;FILE_LOCK=NO 设定, 以避免在重部署时发生文件锁定问题.

我们为 H2 数据库使用默认的用户名和密码:

```
<property name="eclipselink.jdbc.user" value="sa"/>
<property name="eclipselink.jdbc.password" value="sa"/>
```

日志配置

JPA 实现库, 以及数据库引擎都会输出日志, 日志也需要在持久化配置中进行设置. 比如, 如果使用 EclipseLink JPA, 你可以使用 FINE 日志级别, 得到包含所有 SQL 语句的日志:

```
<property name="eclipselink.logging.level"
          value="FINE" />
```

其他设置

剩下的是 EclipseLink 的一些数据定义语言设置. 在开发过程中, 当我们使用自动产生的示例数据时, 我们会希望 EclipseLink 在尝试创建数据库表之前, 首先删除已存在的表. 但在生产环境中, 你应该使用 create-tables.

```
<property name="eclipselink.ddl-generation"
          value="drop-and-create-tables" />
```

而且, 不需要产生 SQL 文件, 只需要在数据库中直接执行它们.

```
<property name="eclipselink.ddl-generation.output-mode"
          value="database"/>
      </properties>
    </persistence-unit>
</persistence>
```

19.2.6. 故障诊断

下面是使用 JPA 时可能遇到的一些常见错误. 这些错误与 JPAContainer 无关.

javax.persistence.PersistenceException: No Persistence provider for EntityManager
这个错误最常见的原因是, 源代码中或 persistence.xml 配置文件中的持久化单元名称不正确, 或者 persistence.xml 所在的位置不正确, 或者发生了其他问题. 请确认持久化单元名称是一致的, 并确认 persistence.xml 在发布后的环境中的 WEB-INF/classes/META-INF 文件夹下.

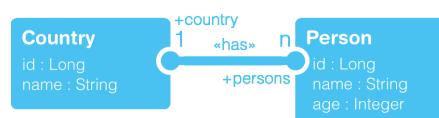
java.lang.IllegalArgumentException: The class is not an entity
持久化实体集合中缺少这个类. 如果 persistence.xml 配置文件中 exclude-unlisted-classes 没有定义为 false, 那么持久化类必须使用 <class> 元素列出.

19.3. 定义业务数据模型(Domain Model)

开发一个持久化应用程序的首先需要定义业务数据模型(domain model). 业务数据模型包括多个实体(类), 以及这些实体之间的关系.

图 19.4 “一个业务数据模型”以 URL 类图的形式展示了一个简单的业务数据模型. 其中有两个实体: **Country** 和 **Person**. 它们之间存在一种关系: “国家拥有人”. 这是一个一对多关系, 一个国家中存在多个人, 每个人都只属于一个国家.

图 19.4. 一个业务数据模型



使用 Java 来实现，类的代码如下：

```
public class Country {  
    private Long id;  
    private String name;  
    private Set<Person> persons;  
  
    ... setters and getters ...  
}  
  
public class Person {  
    private Long id;  
    private String name;  
    private Integer age;  
    private Country country;  
  
    ... setters and getters ...  
}
```

你应该将实体类设计为正确的 Java Bean，为它定义默认的构造函数，并实现 Serializable 接口。JPA 实体管理器创建实体类的实例时需要默认构造函数。让实体类可序列化不是必需的要求，但由于其他一些原因，常常是有用的。

有了基本的业务数据模型后，你需要对类使用注解来定义实体关系元数据。

19.3.1. 持久化元数据

实体之间的关系通过元数据来定义。元数据可以定义在 XML 元数据文件中，也可以使用 javax.persistence 包中的 Java 注解来定义。使用 Vaadin JPACContainer 时，你需要以注解的方式定义元数据。

比如，看看 JPACContainer AddressBook 示例程序中的 Person 类，我们对类中成员变量定义了与数据库相关的元数据，如下：

```
@Entity  
public class Person {  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
  
    private String name;  
    private Integer age;  
  
    @ManyToOne  
    private Country country;
```

JPA 实现库使用反射来读取注解，并根据类的定义自动地定义数据库模型。

下面我们来看看一些基本的 JPA 元数据注解。这些注解定义在 javax.persistence 包中。关于所有可用注解的完整列表，请参见 JPA 文档。

注解: @Entity

每个可作为持久化实体的类，都必须带有 @Entity 注解。

```
@Entity  
public class Country {
```

注解: @Id

实体必须有一个标识符(identifier), 标识符将被用作数据库表中的主键(primary key). 标识符在数据库查询中被用作多种用途, 最常见的是用来做表之间的结合.

```
@Id  
@GeneratedValue(strategy = GenerationType.AUTO)  
private Long id;
```

标识符的值会在数据库中自动生成. 标识符的生成策略通过 `@GeneratedValue` 注解来定义. 使用任何一种生成策略都可以.

注解: @OneToOne

`@OneToOne` 注解描述一个一对一关系, 某个类型的一个实体关联到另一个类型的一个(并且只有一个)实体. 比如, 一个人的邮政地址与一个人就可以通过一对一关系来描述.

```
@OneToOne  
private Address address;
```

使用 JPAContainer 的 **FieldFactory** 来为 Form 自动创建 Field 时, `@OneToOne` 关系会生成一个嵌套的 **Form** 来编辑数据. 详情请参见 第 19.8 节 “Form 的自动生成”.

注解: @Embedded

与 `@OneToOne` 注解类似, `@Embedded` 也描述一对关系, 但被引用的实体不是单独的数据库表, 而是存在于引用它的实体的同一个表的列中.

```
@Embedded  
private Address address;
```

被引用的类必须带有 `@Embeddable` 注解.

与 `@OneToOne` 一样, JPAContainer 的 **FieldFactory** 会为 `@Embedded` 生成一个嵌套的 **Form**.

注解: @OneToMany

业务数据模型中的 **Country** 实体中存在一个 one-to-many 关系, 指向 **Person** 实体 (也就是说 "国家拥有人"). 这个关系通过 `@OneToMany` 注解来表达. `mappedBy` 参数指定在 **Person** 实体中反向参照到 **Country** 实体的属性名.

```
@OneToMany(mappedBy = "country")  
private Set<Person> persons;
```

当使用 JPAContainer 的 **FieldFactory** 来为 Form 自动创建 Field 时, `@OneToMany` 关系会产生一个 **MasterDetailEditor** 来编辑数据元素. 详情请参见 第 19.8 节 “Form 的自动生成”.

注解: @ElementCollection

`@ElementCollection` 注解也可以用来定义一对多关系, 区别是 collection 中的元素是基本数据类型, 比如 **String** 或 **Integer**, 或者是带有 `@Embeddable` 注解的实体类. 被引用的实体类存储在一个独立的数据库表中, 表名通过 `@CollectionTable` 注解来定义.

```
@ElementCollection  
@CollectionTable(  
    name="OLDPEOPLE",
```

```
joinColumns=@JoinColumn(name="COUNTRY_ID"))
private Set<Person> persons;
```

与 @OneToMany 一样, JPACContainer 的 **FieldFactory** 会为 @ElementCollection 关系生成 **MasterDetailEditor**.

注解: @ManyToOne

很多人可以居住在同一个国家. 这个关系应该在 **Person** 类中使用 @ManyToOne 注解来表达.

```
@ManyToOne
private Country country;
```

JPACContainer 的 **FieldFactory** 会生成一个 **NativeSelect**, 用来从列表中选择一个项目. 你也可以在自定义的 Field 工厂中自行实现同样的功能. 这时你需要注意, 不要混淆了被引用的实体和它的 ID, 否则某些情况下甚至可能会导致向数据库插入错误的值. 你可以使用 **SingleSelectConverter** 在实体和实体 ID 之间进行转换, 如下:

```
@Override
public <T extends Field> T createField(Class<?> dataType,
                                         Class<T> fieldType) {
    if (dataType == Country.class) {
        JPACContainer<Country> countries =
            JPACContainerFactory.make(Country.class, "mypunit");
        ComboBox cb = new ComboBox(null, countries);
        cb.setConverter(new SingleSelectConverter<Country>(cb));
        return (T) cb;
    }
    return super.createField(dataType, fieldType);
}
```

JPACContainer 的 **FieldFactory** 内部会使用这个转换器, 因此建议你也使用它来避免上面提到的问题.

注解: @Transient

JPA 假定实体中所有的属性都需要持久化. 不需要持久化的属性应该使用 @Transient 注解, 标记为暂态属性.

```
@Transient
private Boolean superDepartment;
...
@Transient
public String getHierarchicalName() {
    ...
}
```

19.4. JPACContainer 的基本使用

Vaadin JPACContainer 提供了一组高度灵活的 API, 对于简单的问题可以非常容易地实现, 对于复杂的需求也提供了高度的灵活性. 首先, 它是一种 **Container**, 关于容器, 详情请参见 第 8.5 节 “在容器(Container)中保存项目(item)”.

在这一节中, 我们来看看如何创建和使用 **JPACContainer** 实例. 我们假定你象前一节中介绍的那样, 已经使用 JPA 注解定义了业务数据模型.

19.4.1. 使用 **JPAContainerFactory** 创建 **JPAContainer**

JPAContainerFactory 是用来创建 **JPAContainer** 的简便方法. 它针对你通常会遇到的大多数情况, 提供了一组 *make...()* 工厂方法. 每个工厂方法使用一种不同类型的 Entity Provider, 关于 Entity Provider, 详情请参见第 19.5 节 “实体提供者(Entity Provider)”.

工厂方法的第一个参数是实体类的类型. 第二个参数是持久化单元的名称(持久化环境上下文 persistence context), 或者一个 **EntityManager** 实例.

```
// Create a persistent person container
JPAContainer<Person> persons =
    JPAContainerFactory.make(Person.class, "book-examples");

// You can add entities to the container as well
persons.addEntity(new Person("Marie-Louise Meilleur", 117));

// Set up sorting if the natural order is not appropriate
persons.sort(new String[]{"age", "name"},
             new boolean[]{false, false});

// Bind it to a component
Table personTable = new Table("The Persistent People", persons);
personTable.setVisibleColumns(new String[]{"id", "name", "age"});
layout.addComponent(personTable);
```

就这么简单. 实际上, 如果你将上面的代码运行几次, 你就会厌烦了, 因为每次运行都会得到一个新的 Person 集合 - 这就是持久化容器的运行方式. 基本的 *make()* 方法使用的 Entity Provider 是 **CachedMutableLocalEntityProvider**, 它允许修改容器及其中的实体, 我们在上例中向容器添加了新的实体.

如果只使用持久化单元名称, 工厂类为持久化单元会创建 **EntityManagerFactory** 的实例, 并使用它来构建实体管理器. 你也可以自行创建实体管理器, 详情将在后文中介绍.

各个工厂方法分别对应如下各种 Entity Provider :

表 19.1. **JPAContainerFactory** 的方法

make()	CachingMutableLocalEntityProvider
makeReadOnly()	CachingLocalEntityProvider
makeBatchable()	BatchableLocalEntityProvider
makeNonCached()	MutableLocalEntityProvider
makeNonCachedReadOnly()	LocalEntityProvider

JPAContainerFactory 内部针对不同的持久化单元保持一个实体管理器工厂的缓存, 以便确保每个实体管理器工厂只创建一次, 因为实体管理器工厂的创建是一个很笨重的工作. 你可以通过 *createEntityManagerForPersistenceUnit()* 方法来访问这个缓存, 得到新的实体管理器.

```
// Get an entity manager
EntityManager em = JPAContainerFactory.
    createEntityManagerForPersistenceUnit("book-examples");

// Do a query
em.getTransaction().begin();
em.createQuery("DELETE FROM Person p").executeUpdate();
em.persist(new Person("Jeanne Calment", 122));
```

```
em.persist(new Person("Sarah Knauss", 119));
em.persist(new Person("Lucy Hannah", 117));
em.getTransaction().commit();
```

...

注意,如果你在与数据绑定的 **JPACContainer** 之外,使用实体管理器来更新持久化数据,你需要刷新容器,详情请参见第 19.4.2 节“创建和访问实体”.

手动创建 **JPACContainer**

通常最简单的方法是使用 **JPACContainerFactory** 来创建 **JPACContainer** 实例,但你也有可能会需要手动创建 **JPACContainer**. 比如,当你需要使用自定义的 Entity Provider 时,或者需要扩展 **JPACContainer** 时,手动创建就是必须的.

首先,我们需要创建一个实体管理器. 然后创建 Entity Provider,然后将 Entity Provider 绑定到一个 **JPACContainer** 上.

```
// We need a factory to create entity manager
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("book-examples");

// We need an entity manager to create entity provider
EntityManager em = emf.createEntityManager();

// We need an entity provider to create a container
CachingMutableLocalEntityProvider<Person> entityProvider =
    new CachingMutableLocalEntityProvider<Person>(Person.class,
                                                em);

// And there we have it
JPACContainer<Person> persons =
    new JPACContainer<Person>(Person.class);
persons.setEntityProvider(entityProvider);
```

如果通过 **JPACContainerFactory** 获取实体管理器,那么上例中的第一步可以省去.

19.4.2. 创建和访问实体

JPACContainer 与 JPA 实体管理器集成在一起,实体管理器通常用来通过 JPA 创建和访问实体. 也可以将实体管理器用于你需要的其他用途. **JPACContainer** 用来将实体与 UI 组件绑定在一起,比如 **Table**, **Tree**, 选择组件,或者 **Form**.

你可以使用 `addEntity()` 方法来向 **JPACContainer** 添加新的实体. 这个方法将返回新实体的 Item ID.

```
Country france = new Country("France");
Object itemId = countries.addEntity(france);
```

JPACContainer 使用的 Item ID 是由 `@Id` 注解定义的 ID 属性(列)的值. 在我们的 **Country** 实体中,它的类型是 **Long**. 当实体被持久化时, ID 的值会由实体管理器生成,然后通过 `set` 方法设置给 ID 属性.

注意, `addEntity()` 方法 不会 关联参数中给定的那个实体实例. 它会创建新的实例. 如果你需要使用实体,你需要从容器中取得实际被它管理的实体. 你可以通过得到 `addEntity()` 方法返回的 Item ID 得到对应的实体.

```
// Create a new entity and add it to a container
Country france = new Country("France");
Object itemId = countries.addEntity(france);

// Get the managed entity
france = countries.getItem(itemId).getEntity();

// Use the managed entity in entity references
persons.addEntity(new Person("Jeanne Calment", 122, france));
```

实体的 Item

在通常的 Vaadin Container 接口中定义了 `getItem()` 方法. 在 **JPAContainer** 中, 这个方法会返回一个 **EntityItem**, 这个对象内封装了一个实际的实体对象. 你可以通过 `getEntity()` 方法得到实体对象.

EntityItem 可以有一系列状态: 已持久化(persistent), 已修改(modified), 脏(dirty), 以及已删除(deleted). 脏状态和已删除状态只有在容器级缓存功能时才是有意义的, 已修改状态只有在元素级缓存功能时才是有意义的. 这两个缓存级别可以同时使用 - 用户输入首先写入到元素的缓存中, 然后再写入到实体实例中, 最后才写入到数据库中.

`isPersistent()` 方法检测 Item 是否已被持久化, 也就是说, 是否是从持久化存储中取得的, 或者仅仅是一个由容器创建并缓冲的暂态实体(transient entity).

`isModified()` 方法检查 **EntityItem** 中是否存在还未提交到实体实例中去的修改. 只有通过 `setBuffered(true)` 方法对 Item 启动缓存功能后, 这个功能才有效.

`isDirty()` 方法检查实体对象从 Entity Provider 中取得之后, 是否被修改过. 只有在容器的缓冲功能启用后, 脏状态才有可能发生.

`isDeleted()` 方法检查在一个有缓冲功能的容器中, Item 是否被 `removeItem()` 方法标记为已删除.

刷新 JPAContainer

万一你在容器之外修改了 **JPAContainer** Item, 比如通过 EntityManager, 或者数据库之内的数据发生了变化, 你需要刷新容器.

JPAContainer 实现了 `EntityContainer` 接口, 这个接口提供了两个方法来刷新容器. `refresh()` 会抛弃容器内的所有缓存, 然后刷新容器内已装载的所有 Item. 对容器中所有 Item 所做的一切更新都将被废弃. `refreshItem()` 方法只刷新单个 Item.

19.4.3. 嵌套属性

如果你有一对一或者多对一关系, 你可以在 **JPAContainer** 中将被引用的实体定义为 嵌套属性. 通过这种方式, 你可以通过第一个实体类型的容器直接访问属性, 就好象这些属性是它的属性一样. 相关接口与 第 8.5.4 节 “**BeanContainer**” 中介绍过的 **BeanContainer** 一样. 你只需要通过 `addNestedContainerProperty()` 方法添加所有的嵌套属性, 嵌套属性的名称使用点号分隔.

```
// Have a persistent container
JPAContainer<Person> persons =
    JPAContainerFactory.make(Person.class, "book-examples");

// Add a nested property to a many-to-one property
persons.addNestedContainerProperty("country.name");
```

```
// Show the persons in a table, except the "country" column,
// which is an object - show the nested property instead
Table personTable = new Table("The Persistent People", persons);
personTable.setVisibleColumns(new String[]{"name", "age",
                                         "country.name"});

// Have a nicer caption for the country.name column
personTable.setColumnHeader("country.name", "Nationality");
```

上例的运行结果见图 19.5 “嵌套属性”. 注意, 添加嵌套属性后, 容器中的 country 属性仍然会被保留, 因此我们必须将这个属性对应的列设置为不可见. 另外一个方法是, 我们可以重定义 country 对象的 `toString()` 方法, 让它显示国家的名称, 而不是一个对象引用.

图 19.5. 嵌套属性

The Persistent People			
NAME	AGE	NATIONALITY	
Jeanne Calment	122	France	^
Sarah Knauss	119	United States	▼
Marie-Louise Meilleur	117	Canada	^
Lucy Hannah	117	United States	▼
Tane Ikai	116	Japan	▼

你可以使用 * 通配符, 来添加嵌套 Item 中的所有属性, 比如, "country.*".

19.4.4. 层级数据容器

JPACContainer 实现了 `Container.Hierarchical` 接口, 因此可以绑定到层级式组件, 比如 **Tree** 或 **TreeTable**. 这个功能要求数据之间的层级关系通过一个 `parent` 属性表达, 这个属性应该引用到一个父 Item. 在数据库层面, 层级关系应该对应到一个列, 它指向其他表的 ID.

如下例:

```
@Entity
public class CelestialBody implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne
    private CelestialBody parent;
    ...
}

// Create some entities
CelestialBody sun      = new CelestialBody("The Sun", null);
CelestialBody mercury = new CelestialBody("Mercury", sun);
CelestialBody venus   = new CelestialBody("Venus", sun);
CelestialBody earth   = new CelestialBody("Earth", sun);
CelestialBody moon    = new CelestialBody("The Moon", earth);
CelestialBody mars    = new CelestialBody("Mars", sun);
...
```

你应该调用 `setParentProperty()` 方法将 **JPAContainer** 设置为级层容器. 方法参数是引用到父 Item 的那个属性名称. 在我们的示例中, 这个属性的名字恰巧就是 "parent":

```
// Create the container
JPAContainer<CelestialBody> bodies =
    JPAContainerFactory.make(CelestialBody.class, "my-unit");

// Set it up for hierarchical representation
bodies.setParentProperty("parent");

// Bind it to a hierarchical component
Tree tree = new Tree("Celestial Bodies", bodies);
tree.setItemCaptionMode(Tree.ITEM_CAPTION_MODE_PROPERTY);
tree.setItemCaptionPropertyId("name");
```

你可以使用 `rootItemIds()` 方法来获取(不存在父节点的)根节点元素所对应的 Item ID.

```
// Expand the tree
for (Object rootId: bodies.rootItemIds())
    tree.expandItemsRecursively(rootId);
```

不支持的层级功能

在容器中使用 `setParent()` 方法来定义数据间的父子关系, 这个功能是不支持的.

而且, 目前的实现库不支持 `setChildrenAllowed()` 方法, 这个方法控制用户能否点击开关来展开一个节点. 展开/收起开关默认对所有节点都是可见的, 即使某些节点之下不存在子节点. 不支持这个方法, 是因为它需要在实体之外保存一部分信息. 你也可以重载 `areChildrenAllowed()` 方法, 使用你的自定义逻辑来实现这个功能.

```
// Customize JPAContainer to define the logic for
// displaying the node expansion indicator
JPAContainer<CelestialBody> bodies =
    new JPAContainer<CelestialBody>(CelestialBody.class) {
        @Override
        public boolean areChildrenAllowed(Object itemId) {
            // Some simple logic
            return getChildren(itemId).size() > 0;
        }
    };
bodies.setEntityProvider(
    new CachingLocalEntityProvider<CelestialBody>(
        CelestialBody.class, em));
```

19.5. 实体提供者(Entity Provider)

实体持久化存储到数据存储中, Entity provider 实现对实体的访问功能. Entity provider 本质上就是对 JPA 实体管理器的包装, 并进行了优化, 另外还提供了一些重要功能, 用于将持久化数据绑定到 UI 组件.

如果你使用 **JPAContainerFactory** 来创建你的 **JPAContainer** 实例, 那么 Entity Provider 的选择和使用很大程度上是不可见的, 因为工厂类隐藏了这些细节.

JPAContainer 的 Entity Provider 可以定制, 对于某些目的来说定制是必须的. Entity Provider 可以是 Enterprise JavaBean(EJB), 在 Java EE 应用程序服务器中使用它会很有用.

19.5.1. 内建的 Entity Provider

JPAContainer 包含多种不同的内建 Entity Provider: 有缓存的和无缓存的, 可读写的和只读的, 以及支持批量处理的.

缓存对于提高性能是十分有效的, 但会消耗一些内存, 并会导致 Entity Provider 变成有状态的. 批量处理(Batching), 也就是在很大的批处理中运行多个更新处理, 也可以提高性能, 会与缓存同时使用. 它是无状态的, 但执行更新处理会比其他模式略为复杂一些.

如果写入功能不是必须的, 那么比较适当的选择是使用 只读容器.

所有的内建 Entity Provider 都是 局部的(*local*), 也就是说, 它们都使用局部的 JPA 实体管理器来访问实体.

通常推荐使用 **CachingMutableLocalEntityProvider** 来实现可读写的数据访问, 推荐使用 **CachingLocalEntityProvider** 实现只读访问.

LocalEntityProvider

这是一个只读, 延迟装载(lazy loading)的 Entity Provider, 它没有缓存功能, 直接从实体管理器读取数据.

你可以使用 **JPAContainerFactory** 的 `makeNonCachedReadOnly()` 方法来创建这个 Entity Provider.

MutableLocalEntityProvider

这个 Entity Provider 继承自 **LocalEntityProvider**, 增加了数据写入功能. 所有的更新都会直接发送给实体管理器.

事务可以由 Entity Provider 内部处理, 默认就是如此, 也可以由容器负责处理. 后一种情况下, 你可以继承类, 并使用注解来标记它, 例子可参见 第 19.5.1 节 “内建的 Entity Provider”.

Entity Provider 可以使用 `EntityProviderChangeNotifier` 接口, 将更新通知发送给实体.

BatchableLocalEntityProvider

这是 `BatchableEntityProvider` 接口的一个无缓存的简单实现. 它继承自 **MutableLocalEntityProvider**, 并简单地将自己传递给 `batchUpdate()` 回调方法. 如果实体中不包含对同一个容器所管理的任何其他实体的引用, 那么这种方式将会正常工作.

CachingLocalEntityProvider

这是一个只读, 延迟加载的 Entity Provider, 它既缓存实体, 也针对过滤器/排序的不同组合缓存查询结果. 缓存满后, 缓存中最老的实体将被移除. 实体的最大数目, 以及针对过滤/排序的每一种组合的被缓存的实体 ID, 都可以在 Entity Provider 进行配置. 缓存也可以手动刷新(flush). 当缓存满时, 最老的项目会被移除.

你可以使用 **JPAContainerFactory** 的 `makeReadOnly()` 方法创建这个 Entity Provider.

CachingMutableLocalEntityProvider

这个 Entity Provider 与 **CachingLocalEntityProvider** 类似, 但有数据写入功能. 对于数据的读, 缓存功能与只读 Entity Provider 中的工作模式一样. 当实体被添加或被修改后, 缓存会被刷新(flush),

以便确保使用过滤或排序功能时，添加或更新的实体能够正确显示。当实体被删除时，只有包含这个实体的过滤/排序缓存会被刷新。

对于大多数使用场合来说，这可能是最常用的 Entity Provider。你可以使用 **JPAContainerFactory** 的 `make()` 方法创建这个 Entity Provider。

CachingBatchableLocalEntityProvider

这个 Entity Provider 支持通过 批量处理 来进行更新。你需要实现一个 `BatchUpdateCallback` 来执行更新，并对 Entity Provider 调用 `batchUpdate()` 方法来执行批处理。

这个 Entity Provider 继承自 **CachingMutableLocalEntityProvider**，这个父类实现了 `BatchableEntityProvider` 接口。如果实体中不包含对同一个容器所管理的任何其他实体的引用，那么这种方式将会正常工作。

你可以使用 **JPAContainerFactory** 的 `makeBatchable()` 方法创建这个 Entity Provider。

19.5.2. 在 JEE6 环境中使用 JNDI Entity Provider

JPAContainer 2.0 引入的一组新的，适合于在 JEE6 环境中工作的 Entity Provider。在 JEE 环境中，你应该使用应用程序服务器提供的实体管理器，而且通常应该使用 JTA 事务，而不是 JPA 提供的事务。`com.vaadin.addon.jpacontainer.provider.jndijta` 包中的 Entity provider 的工作方式与前面讨论过的那些普通的 Entity Provider 基本一致，不同之处是，它们使用 JNDI 来查找得到 EntityManager 和 JTA 事务。

JNDI Entity Provider 几乎不需要任何特殊的配置。**JPAContainerFactory** 中带有工厂方法，可以创建各种类型的 JNDI Entity Provider。通常你只需要将一个 EntityManager 公布到一个 JNDI 地址上。默认情况下，JNDI Entity Provider 会在 "java:comp/env/persistence/em" 地址上查找 EntityManager。这个配置可以通过下例中的 web.xml 代码片段来实现，也可以使用注解来实现类似的配置。

```
<persistence-context-ref>
    <persistence-context-ref-name>
        persistence/em
    </persistence-context-ref-name>
    <persistence-unit-name>MYPU</persistence-unit-name>
</persistence-context-ref>
```

"MYPU" 就是你的 `persistence.xml` 文件中定义的持久化单元的标识符。

如果你选择使用注解来配置你的 Servlet（而不是向上例一样使用 `web.xml` 文件），你只需要向你的 Servlet 添加以下这段注解。

```
@PersistenceContext(name="persistence/em", unitName="MYPU")
```

如果你希望为你的持久化上下文使用其他地址，你可以使用 `setJndiAddresses()` 方法来定义 JNDI 地址。你还可以 JTA **UserTransaction** 的位置，但根据 JEE6 规范，它必须可以通过 "java:comp/UserTransaction" 地址来访问。

19.5.3. EJB 形式的 Entity Provider

Entity provider 可以是 Enterprise JavaBean (EJB)。当你在 Java EE 应用程序服务器中使用 JPAContainer 时，这种方式会很有用。这种情况下，你需要实现一个自定义的 Entity Provider，允许服务器将实体管理器注入进来。

比如, 如果你需要使用 Java Transaction API (JTA) 来管理 JPA 事务, 你可以通过如下方式实现这样的 Entity Provider. 只需要选择一个内建的 Entity Provider, 继承它, 然后对其中的实体管理器成员变量添加 @PersistenceContext 注解. Entity Provider 可以是无状态的, 也可以是有状态的 Session Bean. 如果你继承的是有缓存功能的 Entity Provider, 那么它必须是有状态的.

```
@Stateless
@TransactionManagement
public class MyEntityProviderBean extends
    MutableLocalEntityProvider<MyEntity> {

    @PersistenceContext
    private EntityManager em;

    protected LocalEntityProviderBean() {
        super(MyEntity.class);
        setTransactionsHandledByProvider(false);
    }

    @Override
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    protected void runInTransaction(Runnable operation) {
        super.runInTransaction(operation);
    }

    @PostConstruct
    public void init() {
        setEntityManager(em);
        /*
         * The entity manager is transaction-scoped, which means
         * that the entities will be automatically detached when
         * the transaction is closed. Therefore, we do not need
         * to explicitly detach them.
         */
        setEntitiesDetached(false);
    }
}
```

如果你有多个 EJB Provider, 你可能会希望为上例中的类创建一个抽象的父类, 然后将实体类型只定义在实现类中. 你也可以通过同样的方式, 在 Spring Framefork 中将 Entity Provider 实现为一个被管理的 Bean.

19.6. 在 JPAContainer 中过滤

通常, 一个 **JPAContainer** 包含某个实体类型在持久化上下文中的所有实例. 因此, 它等同于一个数据库表或查询. 与数据库查询一样, 你经常会希望介绍查询结果的数目. **JPAContainer** 实现了 Vaadin 容器的 Filterable 接口, 关于这个接口, 详情请参见 第 8.5.7 节 “**Filterable** 容器”. 所有的过滤处理都会以查询的方式在数据库级别实现, 而不是在容器内部.

比如, 让我们来过滤年龄为 117 岁以上的人:

```
Filter filter = new Compare.Greater("age", 117);
persons.addContainerFilter(filter);
```

以上代码产生的 JPQL 查询, 大致如下:

```
SELECT id FROM Person WHERE (AGE > 117)
```

查询使用 JPA 2.0 Criteria API 透明地实现。由于过滤是在数据库级别上实现的，使用 **Filterable** API 的自定义过滤将无法工作。

使用 Hibernate 时，注意它不支持隐含的表连接(join)。详情请参见 第 19.9.3 节“Hibernate 与 EclipseLink 中的表连接(Join)对比”。

19.7. 使用 Criteria API 进行查询

如果 **Filterable** API 不能适应你的需求，你需要更多控制时，你可以直接使用 JPA Criteria API 来创建查询。你还可能需要对排序或表的连接(join)进行定制，或者对查询进行其他修改。为了实现这一点，你需要实现 **QueryModifierDelegate** 接口，JPAContainer 的 Entity Provider 创建查询时会调用这个接口。最简单的方法是继承 **DefaultQueryModifierDelegate**，其中包含了所有方法的空实现，因此你可以只重载你需要的方法。

Entity Provider 在创建查询的各个阶段，会调用 **QueryModifierDelegate** 的不同方法。这些阶段包括：

1. 开始构建查询
2. 添加 "ORDER BY" 表达式
3. 添加 "WHERE" 表达式(过滤)
4. 结束查询的构建

在各个阶段之前和之后会被调用，你可以用来修改查询的方法，见下表：

表 19.2. QueryModifierDelegate 的方法

queryWillBeBuilt()
orderByWillBeAdded()
orderByWasAdded()
filtersWillBeAdded()
filtersWereAdded()
queryHasBeenBuilt()

所有方法都会收到两个参数。`CriteriaBuilder` 参数是构建器，你可以用来构建查询。`CriteriaQuery` 参数是正在被构建的查询。

你可以使用 `CriteriaQuery` 的 `getRoots().iterator().next()` 方法来得到被查询的“根”，比如，`PERSON` 表，等等。

19.7.1. 对查询进行过滤

我们来考虑一种情况，我们需要修改 **Person** 容器的查询，使容器只包含 116 岁以上的人。这个简单的例子与前面使用 **Filterable** 接口实现的例子是一样的。

```
persons.getEntityProvider().setQueryModifierDelegate(
    new DefaultQueryModifierDelegate () {
        @Override
        public void filtersWillBeAdded(
            CriteriaBuilder criteriaBuilder,
            CriteriaQuery<?> query,
```

```

        List<Predicate> predicates) {
    Root<?> fromPerson = query.getRoots().iterator().next();

    // Add a "WHERE age > 116" expression
    Path<Integer> age = fromPerson.<Integer>get("age");
    predicates.add(criteriaBuilder.gt(age, 116));
}
});

```

19.7.2. 兼容性

构建查询时, 你应该考虑不同的 JPA 实现之间的兼容性问题. 关于 Hibernate, 请参见 第 19.9.3 节 “Hibernate 与 EclipseLink 中的表连接(Join)对比”.

19.8. Form 的自动生成

JPAContainer 的 **FieldFactory** 实现了 `FormFieldFactory` 接口和 `TableFieldFactory` 接口, 它可以根据 POJO 类中的 JPA 注解来生成 Field. 它的功能比 **DefaultFieldFactory** 更强, **DefaultFieldFactory** 只能为基本数据类型生成简单的 Field. 使用这种方式, 你可以简单地创建 Form 来输入实体数据, 也可以实现表内的编辑功能.

默认生成的 Field 组件如下:

注解	对应的 Field 组件类
<code>@ManyToOne</code>	NativeSelect
<code>@OneToOne, @Embedded</code>	嵌套的 Form
<code>@OneToMany, @ElementCollection</code>	MasterDetailEditor (详情见后文)
<code>@ManyToMany</code>	可选择的 Table

Field Factory 是递归的, 因此你可以在一个 Form 之内编辑一个复杂的对象树.

19.8.1. 配置 Field Factory

FieldFactory 是可以高度配置的, 可以使用各种配置设定, 也可以扩展继承它.

`setMultiSelectType()` 方法和 `setSingleSelectType()` 方法允许你指定选择组件, 代替默认的选择组件, 用于被注解为 `@ManyToMany` 和 `@ManyToOne` 的 Field. 第一个参数是 Field 组件的类型, 第二个参数是选择组件的类型. 它必须是 **AbstractSelect** 的子类.

`setVisibleProperties()` 控制在生成的 Form, 子 Form, 以及 Table 中, 哪些属性(Field) 可见. 第一个参数是这个设置适用的数据类型, 第二个参数是可见属性的 ID .

将 Form 绑定到数据源时就会创建 Field 组件, 因此应该在绑定之前进行设置.

更多的配置必须通过继承这个类并覆盖其中的 `protected` 方法来实现. 详细的方法列表请参见 API 文档.

19.8.2. 使用 Field Factory

JPAContainer **FieldFactory** 的最基本的使用场景是用于一个 绑定到容器中元素上的 **Form**:

```
// Have a persistent container
final JPAContainer<Country> countries =
    JPAContainerFactory.make(Country.class, "book-examples");

// For selecting an item to edit
final ComboBox countrySelect =
    new ComboBox("Select a Country", countries);
countrySelect.setItemCaptionMode(Select.ITEM_CAPTION_MODE_PROPERTY);
countrySelect.setItemCaptionPropertyId("name");

// Country Editor
final Form countryForm = new Form();
countryForm.setCaption("Country Editor");
countryForm.addStyleName("bordered"); // Custom style
countryForm.setWidth("420px");
countryForm.setBuffered(true);
countryForm.setEnabled(false);

// When an item is selected from the list...
countrySelect.addValueChangeListener(new ValueChangeListener() {
    @Override
    public void valueChange(ValueChangeEvent event) {
        // Get the item to edit in the form
        Item countryItem =
            countries.getItem(event.getProperty().getValue());

        // Use a JPAContainer field factory
        // - no configuration is needed here
        final FieldFactory fieldFactory = new FieldFactory();
        countryForm.setFormFieldFactory(fieldFactory);

        // Edit the item in the form
        countryForm.setItemDataSource(countryItem);
        countryForm.setEnabled(true);

        // Handle saves on the form
        final Button save = new Button("Save");
        countryForm.getFooter().removeAllComponents();
        countryForm.getFooter().addComponent(save);
        save.addClickListener(new ClickListener() {
            @Override
            public void buttonClick(ClickEvent event) {
                try {
                    countryForm.commit();
                    countryForm.setEnabled(false);
                } catch (InvalidValueException e) {
                }
            }
        });
    }
});
countrySelect.setImmediate(true);
countrySelect.setNullSelectionAllowed(false);
```

以上代码将产生一个 Form, 见 图 19.6 “对一对多关系使用 FieldFactory”.

图 19.6. 对一对多关系使用 **FieldFactory**

The screenshot shows a user interface for managing countries. At the top, there is a dropdown menu labeled "Select a Country" with "Japan" selected. Below it is a section titled "Country Editor" containing a table and some buttons.

Name	AGE	NAME
Japan	116	Tane Ikai
	116	Kamato Hongo

Below the table, there is a "People" section and two buttons: "Add" and "Remove". At the bottom, there is a "Save" button.

如果你使用 Hibernate, 你还需要指定一个 **EntityManagerPerRequestHelper**, 可以通过构造函数参数来指定, 也可以使用 `setEntityManagerPerRequestHelper()` 方法来指定., 详情请参见第 19.9.2 节 “每个请求一个实体管理器(EntityManager-Per-Request)模式”.

19.8.3. 主-从(Master-Detail)数据编辑器

MasterDetailEditor 是一种 Field 组件, 可以编辑带一对多关系的元素属性. 元素可以是表中的一行, 也可以绑定到 Form 上. 这个组件使用一个可编辑的 **Table** 来显示主数据关联的详细数据, 并允许在 **Table** 中添加和删除数据.

你可以手动地使用 **MasterDetailEditor**, 更常见的方法是使用 JPACContainer **FieldFactory** 来自动创建它. 在 图 19.6 “对一对多关系使用 FieldFactory” 例子中已经展示过, Factory 对所有带有 `@OneToMany` 注解或 `@ElementCollection` 注解的属性, 都创建了 **MasterDetailEditor**.

19.9. JPACContainer 与 Hibernate 的结合使用

某些情况下, Hibernate 需要特别的处理.

19.9.1. 延迟装载(Lazy loading)

为了让延迟装载能够自动工作, 实体必须绑定到实体管理器上. 不幸的是, Hibernate 无法长期保持一个实体管理器. 为了绕过这个问题, 你必须对 Hibernate 使用一种特别的延迟装载代理(delegate).

JPACContainer 的 Entity Provider 使用代理来处理延迟装载, 代理由 `LazyLoadingDelegate` 接口定义. 针对 Hibernate 的默认实现定义在 **HibernateLazyLoadingDelegate**. 你可以创建一个实例, 并通过 Entity Provider 的 `setLazyLoadingDelegate()` 方法来使用它.

默认实现是可以工作的, 因此当通过 Vaadin Property 来访问一个延迟装载的属性时, 属性值会使用当前活动的实体管理器, 通过一个独立的 (JPA Criteria API) 查询来取得. 然后属性值会被手动关联到实体实例上, 而实体实例会与实体管理器脱离关联(译注: 此句意义不明, 待校). 如果这种默认实现不能满足你的需要, 你可能需要创建你自己的实现.

19.9.2. 每个请求一个实体管理器(EntityManager-Per-Request)模式

使用 Hibernate 时的一个问题是，它是设计用于短期 session 的，但一个实体管理器的生存周期通常大致等于一个用户 session。问题是，如果在 session 或实体管理器中发生了错误，实体管理器将处于不可用状态。这就导致了长生存期 session 的问题，而在 EclipseLink 中不会出现这种问题。

推荐的解决方法是使用 *每个请求一个实体管理器(EntityManager-per-Request)* 模式。使用 Hibernate 时强烈推荐一定要使用这种模式。

实体管理器只能在 Vaadin Servlet 的一次请求-相应周期之间打开，因此实体管理器的实例会在请求开始时创建，并在请求结束时关闭。

保存实体管理器

你首先需要实现 EntityManagerProvider 接口，它通过 getEntityManager() 方法返回一个保存的 EntityManager。实体管理器必须保存在 **ThreadLocal** 变量中。

```
public class LazyHibernateEntityManagerProvider
    implements EntityManagerProvider {
    private static ThreadLocal<EntityManager>
        entityManagerThreadLocal =
        new ThreadLocal<EntityManager>();

    @Override
    public EntityManager getEntityManager() {
        return entityManagerThreadLocal.get();
    }

    public static void setCurrentEntityManager(
        EntityManager em) {
        entityManagerThreadLocal.set(em);
    }
}
```

在请求开始时，你需要使用 setCurrentEntityManager() 方法创建并保存各个请求对应的实体管理器实例，并在请求结束时将其设置为 null 来清除它。

在 **Servlet Filter** 中创建实体管理器

你可以为各个请求创建实体管理器，方法可以是继承 **VaadinServlet** 并覆盖 service() 方法，也可以实现一个 Servlet Filter。下例中，我们介绍如何实现一个 Servlet Filter 来实现这个任务，但覆盖 Servlet 会更简单。

```
public class LazyHibernateServletFilter
    implements Filter {

    private EntityManagerFactory entityManagerFactory;

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
        entityManagerFactory = Persistence
            .createEntityManagerFactory("lazyhibernate");
    }

    @Override
    public void doFilter(ServletRequest servletRequest,
        ServletResponse servletResponse,
```

```
        FilterChain filterChain)
    throws IOException, ServletException {
try {
    // Create and set the entity manager
    LazyHibernateEntityManagerProvider
        .setCurrentEntityManager(
            entityManagerFactory
                .createEntityManager());
    // Handle the request
    filterChain.doFilter(servletRequest,
                        servletResponse);
} finally {
    // Reset the entity manager
    LazyHibernateEntityManagerProvider
        .setCurrentEntityManager(null);
}
}

@Override
public void destroy() {
    entityManagerFactory = null;
}
}
```

你需要在部署描述文件 web.xml 中定义这个 Servlet Filter , 如下:

```
<filter>
    <filter-name>LazyHibernateServletFilter</filter-name>
    <filter-class>com.example.LazyHibernateServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>LazyHibernateServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

其中的 url-pattern 设置必须与你的 Vaadin servlet 对应的 url-pattern 一致.

19.9.3. Hibernate 与 EclipseLink 中的表连接(Join)对比

EclipseLink 支持隐式表连接(join), 而 Hibernate 需要显式连接. 用 SQL 的术语来说, 显式的连接也就是 "FROM a INNER JOIN b ON a.bid = b.id" 表达式, 而隐式连接则在 WHERE 子句中实现, 比如: "FROM a,b WHERE a.bid = b.id".

使用 EclipseLink 时, JPAContainer 过滤中, 隐式连接可以是如下形式:

```
new Equal("skills.skill", s)
```

在 Hibernate 中, 你需要使用 **JoinFilter** 来进行显式连接:

```
new JoinFilter("skills", new Equal("skill", s))
```

TouchKit

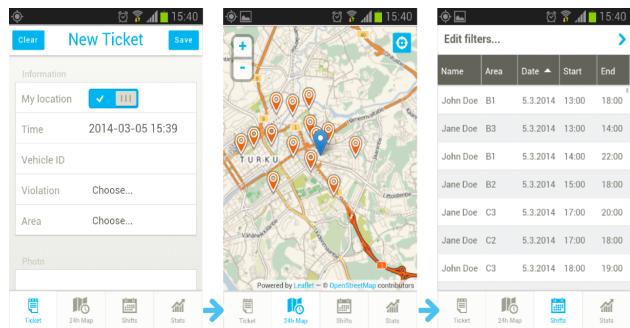
20.1. 概述	523
20.2. 针对移动设备浏览器应当考虑的问题	525
20.3. 安装 Vaadin TouchKit	527
20.4. 导入 Parking 示例程序	528
20.5. 创建新的 TouchKit 工程	528
20.6. TouchKit 应用程序的组成元素	531
20.7. 移动设备 UI 组件	536
20.8. 移动设备高级特性	551
20.9. 离线模式(Offline Mode)	558
20.10. 构建最优化的 Widget Set	560
20.11. 在移动设备上测试和调试	562

本章介绍如何使用 Vaadin TouchKit 创建移动设备应用程序.

20.1. 概述

移动设备上的 Web 浏览正在显著增加, Web 应用程序必须同时满足桌面计算机用户和移动设备用户的需求, 比如手机和平板用户. 虽然移动设备浏览器可以象通常的浏览器一样显示页面, 但为了使得用户体验更加舒适, 还应该注意屏幕尺寸, 手指触碰的精确性, 以及移动设备浏览器中其他应该考虑的因素. 在 Vaadin 创建通常 Web UI 的能力之外, Vaadin TouchKit 还为 Vaadin 添加了创建移动设备 UI 的能力. Vaadin Framework 的目的是创建类似桌面程序的 Web 应用程序, 与此类似, TouchKit 的目的是创建外观与本地移动设备应用程序类似的 Web 应用程序.

图 20.1. Vaadin TouchKit 的 Parking 示例程序



创建移动设备 UI 与创建通常的 Vaadin UI 很类似。你可以使用所有通常的 Vaadin 组件和 Vaadin Directory 中的所有 add-on，但最重要的，你可以使用针对移动设备优化过的 TouchKit 组件。

```
@Theme("mobiletheme")
@Widgetset("com.example.myapp.MyAppWidgetSet")
@Title("My Mobile App")
public class SimplePhoneUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Define a view
        class MyView extends NavigationView {
            public MyView() {
                super("Planet Details");

                CssLayout content = new CssLayout();
                setContent(content);

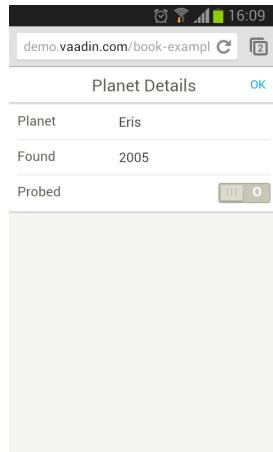
                VerticalComponentGroup group =
                    new VerticalComponentGroup();
                content.addComponent(group);

                group.addComponent(new TextField("Planet"));
                group.addComponent(new NumberField("Found"));
                group.addComponent(new Switch("Probed"));

                setRightComponent(new Button("OK"));
            }
        }

        // Use it as the content root
        setContent(new MyView());
    }
    ...
}
```

上例省略了 Servlet 类的定义，也没有任何 UI 逻辑代码，还省略了通常需要实现的视图，等等。UI 的运行结果见图 20.2 “简单的 TouchKit UI”。

图 20.2. 简单的 TouchKit UI

TouchKit 支持移动设备浏览器的很多特殊功能, 比如地理位置, 上下文相关的输入 Field, 以及返回 Home 界面. 在 iOS 上, 还支持 Splash 画面和 Web 应用程序模式.

除了开发通常的服务器端 UI 之外, TouchKit 还支持特殊的 离线模式(*offline mode*), 这是一种客户端 Vaadin UI, 保存在浏览器缓存中. 无论是在启动应用程序时, 还是在使用途中, 当网络连接不可用时, 都会自动切换到离线模式. 详情请参见 第 20.9 节“离线模式(Offline Mode)”.

本章中, 我们首先考虑移动设备浏览中的一些特殊问题. 然后, 我们再讨论如何使用 TouchKit 创建一个工程. TouchKit 提供了很多特殊的移动设备 UI 组件, 详情将在后面的小节中介绍. 我们会将手机和平板应用程序区别对待, 还将简要地讨论测试问题.

TouchKit 示例

Parking 实例程序演示了基于地理位置的移动设备商业应用程序中 TouchKit 最重要的功能. 这个应用程序的目的是帮助执法人员在街边开违章处罚单. 它使用的功能包括地理位置功能, 通过移动设备摄像头获取图像, 地图导航, 通过 Vaadin Chart 可视化表示数据, 以及由条件式布局实现的动态 UI. 你可以在以下地址试用这个程序: <http://demo.vaadin.com/parking>. 关于如何将这个工程导入到 Eclipse 内, 请参见 第 20.4 节“导入 Parking 示例程序”. 通过 Github 可以浏览或下载它的源代码.

另一个示例应用程序是 Mobile Mail, 它演示了如何实现深度分类目录树(category tree), 以及如何创建 Form. 你可以在以下地址试用这个程序: <http://demo.vaadin.com/mobilemail>. 你还可以在 Github 查看这个示例程序的源代码.

本章中的一部分示例程序可以在以下地址查看其实际运行效果: demo.vaadin.com/touchkit-sampler. 你可以在 GitHub 源代码库 中查看源代码, 或使用 Git 来复制源代码库.

许可协议

Vaadin TouchKit 是一个商业产品, 使用双许可协议. AGPL 许可协议允许开源项目使用这个组件, CVAL 协议需要购买许可, 允许用于闭源项目, 包括 Web 开发和内部使用. 商业许可可以通过 Vaadin Directory 购买, 你在这里可以找到许可协议的详细文本, 并下载 Vaadin TouchKit.

20.2. 针对移动设备浏览器应当考虑的问题

开发支持移动设备的 Web 应用程序时, 你需要考虑很多与非移动设备不同的问题. TouchKit 就是用来帮助你解决这些问题.

20.2.1. 移动设备的人机界面

移动设备使用一种与通常的计算机非常不同的人机界面。比如，屏幕可以很容易地在横向和纵向之间旋转。这个变化不仅改变了显示器的尺寸，而且还影响到如何排列组件才能实现最好的用户体验。除 TouchKit 外，Responsive add-on 也可以帮助应用程序实现灵活的动态布局，详情请参见第 7.9 节“条件式 Theme”。

移动设备的 UI 通过手指来使用，而不是鼠标，因此不存在“右键点击”之类的功能。使用鼠标时你可以在 UI 组件上双击(double-click)或点击右键(right-click)，但在触摸式设备中通常使用点击(tap)和“长按(long tap)”来代替。手指的动作也起到很重要的作用，比如使用一个垂直的拉动来滚动屏幕，而不是使用滚动条。某些浏览器还允许使用两个或多个手指的动作。

通常没有物理键盘，而只有一个屏幕上的虚拟键盘，而且键盘会根据环境上下文而发生变化。你还需要确保键盘弹出时，不会遮挡住用户试图输入数据的那个输入组件。这一点本来应该由浏览器负责处理，但这个问题在测试时需要特别注意。

20.2.2. 带宽与性能

移动设备的 Internet 连接通常会比有线连接慢很多。使用低速的移动设备网络连接时，比如 384 kbps，仅仅装载 Vaadin 客户端引擎本身也会耗费几秒钟时间。为解决这个问题，可以编译 Widget Set，将 theme 编译到 widget set 之内，使它只包含实际用到的组件所需要的 Widget，详情请参见第 20.10 节“构建最优化的 Widget Set”。

即使使用移动设备的低带宽，应用程序的延迟也是重要的因素，尤其是在高度交互的丰富应用程序 (Rich Application) 中更是如此。在有线连接中通常几乎注意不到存在延迟，一般会少于 100 毫秒，而在移动设备的 EDGE(Enhanced Data rates for GSM Evolution) 连接中通常会达到 500 毫秒左右，在网络不稳定时延迟有时甚至会更高。你可能会需要减少使用 immediate 模式，text change 事件，以及轮询(polling)。某些组件中的延迟补偿功能，比如 **NavigationManager** 中，允许在向服务器发送请求的过程中显示一个视图变更动画。

此外，选择使用什么组件也会影响性能。TouchKit 组件被设计得十分轻便。其他 Vaadin 组件，一部分比其他组件更轻便一些。尤其是，与轻量级的 **CssLayout** 相比，大部分其他布局组件都带有更深的 DOM 结构，因此渲染过程会更慢一些。TouchKit 也包含了一些特殊的样式可用于 **CssLayout**。

20.2.3. 移动设备的功能特性

手机和平板有很多集成的功能特性，通常在浏览器 UI 中也可以使用。地理位置感知是最近的功能特性之一。当然，你也可以发起一通电话通话。

20.2.4. 兼容性

移动设备浏览领域正在快速发展中，而且领先的厂商还引入了特别的习惯约定，在未来的几年中，这些约定将会稳定下来，成为新的 Web 标准。TouchKit 主要支持 WebKit 系列浏览器，这个引擎似乎正在成为移动设备上最领先的浏览器内核。除 Apple 产品之外，Android 的默认浏览器也使用 WebKit 作为排版引擎。当然这些浏览器也存在区别，Android 的 JavaScript 引擎，使用的是 Google Chrome 的 V8 引擎，Vaadin 与浏览器的 JavaScript 引擎存在紧密的关系。从 TouchKit 4 开始，也支持 Windows Phone 上的 Internet Explorer 浏览器。

关于最新版的 TouchKit 所支持的移动设备列表，请参见 Vaadin 网站上的 TouchKit 产品页面。

Vaadin TouchKit 会遵循这些主要平台上快速演变中的 API，它假定其他浏览器也会遵循这些领先者制定的标准。如果其他平台的用户份额增加，那么 TouchKit 也会支持它们。

回退按钮

某些移动设备, 尤其是 Android 和 Windows Phone 设备, 有一个专门的回退按钮, 而 iOS 设备则没有. TouchKit 不对回退按钮提供特别的支持, 但是, 由于这是一个通常的浏览器回退按钮, 你可以通过 URI 片段来管理它, 详情请参见 第 11.11 节 “管理 URI 片段”. 对于 iOS, 如果用户将应用程序添加到了 Home 画面, 浏览器回退按钮会被隐藏, 这种情况下你需要实现一段特别的应用程序逻辑来实现回退浏览.

20.3. 安装 Vaadin TouchKit

你可以通过 Vaadin Directory 以安装包的形式下载并安装 TouchKit, 地址是 <https://vaadin.com/addon/vaadin-touchkit>, 也可以通过 Maven 或 Ivy 得到它. 如果你的工程不能使用 AGPL 许可协议, 你可以通过 Vaadin Directory 购买 CVAL 许可协议, 或者通过 vaadin.com/pro 订购 Pro Tools 工具包.

关于 Add-on 的安装方法, 详情请参见 第 17 章 使用 Vaadin Add-on. 这个 Add-on 包含 Widget Set, 因此你需要编译为你的工程编译 Widget Set.

20.3.1. 以 Ivy 依赖项方式安装

如果你在通过 Vaadin Plugin for Eclipse 创建的工程中使用这个 Add-on, 你可以定义一个 Ivy 依赖项目, 即可自动下载库文件. 请将以下声明加入到 ivy.xml 文件的 dependencies 小节中:

```
<dependency org="com.vaadin.addon"
           name="vaadin-touchkit-agpl"
           rev="latest.release"
           conf="default->default" />
```

你可以使用 latest.release 版本标记来使用最新发布版, 或者指定一个具体的版本号. 当你保存这个文件时 IvyDE 会立即解析依赖项目. 详情请参见 第 17.3 节 “在 Eclipse 中使用 Ivy 安装 Add-on”.

20.3.2. 声明 Maven 依赖项

在 Maven 工程中, 你可以将 Vaadin TouchKit 添加为一个依赖项目来安装它, 详情见后述, 也可以使用 Maven archetype, 详情请参见 第 20.5.1 节 “使用 Maven Archetype”.

为了在 Vaadin 工程中使用 TouchKit, 你需要在 POM 中包含以下依赖项目. artifactId 应该是 vaadin-touchkit-agpl 或 vaadin-touchkit-cval, 使用哪一个取决于那种许可协议适合于你的项目需求.

```
<dependency>
    <groupId>com.vaadin.addon</groupId>
    <artifactId>vaadin-touchkit-agpl</artifactId>
    <version>LATEST</version>
</dependency>
```

你可以和上例一样, 使用 LATEST 版本号, 也可以指定一个具体的版本号.

你还需要在 <repositories> 元素下为 Vaadin add-on 定义 repository:

```
<repository>
    <id>vaadin-addons</id>
    <url>http://maven.vaadin.com/vaadin-addons</url>
</repository>
```

最后, 你还需要在 POM 中启动 Widget Set 编译功能, 详情请参见 第 17.4.3 节“启动 Widget Set 编译功能”, 然后编译 Widget Set.

20.3.3. 使用 Zip 包安装

Vaadin TouchKit 以 Zip 包的形式发布, 其中包含 TouchKit JAR, JavaDoc JAR, 许可协议文本文件, 以及其他文档. 你可以从 Vaadin Directory 下载 Zip 包. 针对两种不同的许可协议, 提供了两个不同的安装包, Vaadin Directory 将会询问你选择哪一个.

安装包中的 TouchKit JAR 应该放在 Web 应用程序的 WEB-INF/lib 文件夹下.

关于安装包的内容, 详情请参见 README.html 文件.

20.4. 导入 Parking 示例程序

Parking 示例程序, 参见概述中的 图 20.1 “Vaadin TouchKit 的 Parking 示例程序”, 演示了 Vaadin TouchKit 中的大部分功能. 你可以使用兼容 TouchKit 的 浏览器在线试用这个示例程序, 地址是 demo.vaadin.com/parking.

你可以在线浏览这个示例程序的源代码, 或者使用更便利的方式, 将这个工程导入到 Eclipse(或其他 IDE) 中. 由于这个工程是一个 Maven 工程, Eclipse 用户需要安装 m2e 插件, 才能导入 Maven 工程, 此外还需要 EGit 插件来导入 Git 源代码库. 这些插件安装完毕后, 你就可以导入 Parking 示例程序了, 方法如下.

1. 选择菜单项 **File → Import**
2. 选择 **Maven → Check out Maven Project from SCM**, 然后单击 **Next** 按钮.
3. 在 Eclipse 中需要使用源代码管理工具 EGit 来访问 Git 源代码库, 如果你没有安装过, 那么需要安装它. 如果 Git 没有出现在你的 Eclipse 源代码管理工具列表中, 请点击 **m2e marketplace**, 选择 EGit, 然后点击 **Finish**. 安装完毕后需要重启 Eclipse, 并重新执行前面说的 import 步骤.
- 除了使用 m2e EGit 之外, 还可以使用其他 Git 工具从源代码库中取出工程源代码, 然后在 Eclipse 中将它导入为一个 Maven 工程.
4. 在 **SCM URL** 项目中, 选择 **git**, 并输入源代码库的 URL <https://github.com/vaadin/parking-demo>.
5. 点击 **Finish** 按钮.
6. 编译 widget set, 方法是点击 Eclipse 工具条上的 **Compile Widgetset** 按钮, 或者使用 Maven 运行 vaadin:compile 目标.
7. 将应用程序发布到服务器上. 关于在 Eclipse 中发布程序的方法, 详情请参见 第 2.5.4 节“配置和启动 Web 服务器”.
8. 使用移动设备, 或使用一个兼容 WebKit 的浏览器(比如 Safari 或 Chrome), 访问应用程序地址 <http://localhost:8080/parking>, 来运行 Parking 示例程序.

20.5. 创建新的 TouchKit 工程

插件新的 TouchKit 应用程序的最简便方法是使用 Maven archetype, 或者先使用 Vaadin Plugin for Eclipse 创建通常的 Vaadin 工程, 然后再将它修改为 TouchKit 工程.

20.5.1. 使用 Maven Archetype

你可以使用 Maven `vaadin-archetype-touchkit` archetype 来创建新的 TouchKit 应用程序工程。关于使用 Maven 创建 Vaadin 工程，详情请参见 第 2.6 节“通过 Maven 使用 Vaadin”。

比如，要从命令行创建一个工程，你可以执行以下命令：

```
$ mvn archetype:generate \
-DarchetypeGroupId=com.vaadin \
-DarchetypeArtifactId=vaadin-archetype-touchkit \
-DarchetypeVersion=4.0.0 \
-DgroupId=example.com -DartifactId=myproject \
-Dversion=0.1.0 \
-DapplicationName=My -Dpackaging=war
```

archetype 的 `ApplicationName` 参数被用作各种框架代码类名称的前缀。比如，上例中的名称 “My” 会导致生成的类名为 **MyTouchKitUI** 等。

生成的工程将包含以下源代码文件：

`MyTouchKitUI.java`

TouchKit 应用程序的移动设备 UI。关于 TouchKit UI 的基本概念，请参见 第 20.6.4 节“UI”。示例程序的 UI 使用 **TabBarView** 作为根内容。第一个 tab 包含一个 **MenuView**（详情将在后文中介绍），以及一个工程中定义的功能导航视图框架。

`MyFallbackUI.java`

针对 TouchKit 不支持的浏览器的后备(fallback) UI，比如通常的桌面浏览器。关于后备(fallback) UI，详情请参见 第 20.8.1 节“提供一个备用(Fallback)UI”。

`MyServlet.java`

UI 的 Servlet 类，使用 Servlet API 3.0 中的 `@WebServlet` 注解来定义。生成的 Servlet 定制了 TouchKit，定义了 **MyUIProvider**，**MyUIProvider** 负责设定后备 UI。关于如何自定义 Servlet 来定制 TouchKit，详情请参见 第 20.6.1 节“Servlet 类”。

`MyUIProvider.java`

针对 TouchKit 支持的浏览器，它负责创建 **MyTouchKitUI**，针对不支持的浏览器，负责创建 **MyFallBackUI**。关于后备 UI，详情请参见 第 20.8.1 节“提供一个备用(Fallback)UI”。

`MenuView.java`

这是菜单视图的框架代码。菜单由 **VerticalComponentGroup** 中的一组 **NavigationButton** 构成。点击一个按钮后将会导航跳转到另一个视图；在框架代码中是跳转到 **FormView**（详情见后文）。

`FormView.java`

这是数据输入 Form 的框架代码。

`gwt/AppWidgetSet.gwt.xml`

工程的 Widget set 描述文件。编译时，这个文件将被自动更新，然后会包含工程内可能存在的其他 add-on 的 widget set。

`gwt/client/MyOfflineDataService.java`

数据服务程序的框架代码，用于在离线模式下保存数据。详情请参见 第 20.9 节“离线模式(Offline Mode)”。

`gwt/client/MyPersistToServerRpc.java`

客户端到服务器端 RPC 调用的框架代码，用于将离线模式下的数据保存到服务器端。

如果你将工程导入到 Eclipse 或其他 IDE 中, 你至少需要编译 widget set, 然后才可以发布这个工程. 你可以使用 IDE 中集成的 Maven 来执行这个任务, 或者在命令行中执行以下命令:

```
$ mvn vaadin:compile
```

详情请参见 第 2.6 节 “通过 Maven 使用 Vaadin”. 至少在 Eclipse 中, 现在你应该可以导入一个工程, 并将它发布到开发服务器上了. 你也可以编译工程, 并在 Jetty web 服务器(port 8080) 中启动它, 方法是使用以下命令:

```
$ mvn package
$ mvn jetty:run
```

注意, Archetype 生成的工程会使用 Servlet API 3.0 中的 @WebServlet 注解来定义 Servlet. 应用程序服务器必须支持 Servlet 3.0. 比如, 如果你使用 Tomcat, 那么至少需要 Tomcat 7 以上版本.

20.5.2. 从新的 Eclipse 工程开始创建 TouchKit 工程

Vaadin Plugin for Eclipse 可以创建通常的 Vaadin 工程(参见 第 2.5 节 “使用 Eclipse 创建和运行一个工程”), 你可以通过这个工程来创建 TouchKit 工程.

创建工程之后, 你需要执行以下操作:

1. 在 ivy.xml 中添加 TouchKit 的依赖项目, 将 TouchKit 库安装到工程中, 详情请参见 第 20.3.1 节 “以 Ivy 依赖项方式安装”, 然后编译 Widget Set.
2. 在 Servlet 类中继承 **TouchkitServlet** 而不是原有的 **VaadinServlet**, 详情请参见 第 20.6.1 节 “Servlet 类”. 建议将向导创建的静态内部类抽取为一个通常的类, 因为你很可能需要在其中添加一些额外的配置.

```
@WebServlet(value = "/*",
            asyncSupported = true)
@VaadinServletConfiguration(
    productionMode = false,
    ui = MyMobileUI.class)
public class MyProjectServlet extends TouchkitServlet {
```

3. 如果你希望将来再定义后备 UI(参见 第 20.8.1 节 “提供一个备用(Fallback)UI”), 你可以复制原来的 UI 类的框架代码, 直接将它用作后备 UI 类.
4. 为了快速启动我们的工程, 可以在 UI 类中使用 @Theme("touchkit") 来禁用自定义 Theme. 如果将来想要创建自定义的移动设备 Theme, 请参见 第 20.6.6 节 “移动设备 Theme”.

```
@Theme("touchkit")
public class MyMobileUI extends UI {
```

5. 使用 TouchKit 组件而不是通常的 Vaadin 核心组件来构建移动设备 UI, 详情请参见 第 20.6.4 节 “UI”.

相关问题, 以及其他更多问题, 我们将在 第 20.6 节 “TouchKit 应用程序的组成元素” 中详细讨论.

20.6. TouchKit 应用程序的组成元素

与通常的 Vaadin 应用程序一样, TouchKit 应用程序至少需要一个 UI 类, 它定义在部署描述文件中. 你通常会定义一个 Servlet 类, 在这个类中也可以进行一些 TouchKit 独有的配置. 你还可能需要使用自定义 Theme. 这些任务, 以及其他任务, 都将在后续的小节中详细介绍.

20.6.1. Servlet 类

使用 Servlet 3.0 兼容的应用程序服务器时, 你通常可以通过带有 @WebServlet 注解的 Servlet 类来定义 UI, 并进行基本的配置. Vaadin Plugin for Eclipse 会将 Servlet 类创建为 UI 类的静态内部类, 而 Maven archetype 会将它创建为独立的类, 这种方式通常更好一些.

通常, Servlet 类必须定义 UI 类. 此外, 你还可以在 Servlet 类中对 TouchKit 进行以下配置:

- 自定义书签图标或 Home 画面图标
- 自定义 Splash 画面图片
- 自定义 iOS 中的状态栏
- 在 iOS 中使用特殊的 Web 应用程序模式
- 提供一个后备 UI (第 20.8.1 节 “提供一个备用(Fallback)UI”)
- 启用离线模式

自定义的 Servlet 通常应该继承 **TouchKitServlet**. 你应该将你的代码放在 `servletInitialized()` 方法内, 并在方法内首先调用父类方法.

```
public class MyServlet extends TouchKitServlet {
    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        ... customization ...
    }
}
```

如果你需要继承其他的 Servlet, 比如别的 add-on 内的某个 Servlet, 这种情况下, 重新实现 **TouchKitServlet** 的功能并不复杂, 因为它只是负责管理 TouchKit 的一些设置对象.

如果使用 web.xml 部署描述文件, 而不是 **@WebServlet** 注解, 只有当你所需要实现上述自定义配置时, 你才需要实现自定义的 Servlet 类, 当然通常你都会需要这些自定义配置.

20.6.2. 使用 web.xml 部署描述文件定义 Servlet 和 UI

如果使用老式的 web.xml 部署描述文件, 你需要在 web.xml 中使用 **TouchKitServlet** 类而不是通常的 **VaadinServlet**. 在前一节中我们讨论过, 通常你会需要进行一些配置, 或者在自定义 Servlet 中添加一些特殊的逻辑, 这种情况下你需要在部署描述文件中使用你自己的 Servlet.

```
<servlet>
    <servlet-name>Vaadin UI Servlet</servlet-name>
    <servlet-class>
        com.vaadin.addon.touchkit.server.TouchKitServlet
    </servlet-class>
    <init-param>
```

```

<description>Vaadin UI class to start</description>
<param-name>ui</param-name>
<param-value>com.example.myapp.MyMobileUI</param-value>
</init-param>
</servlet>

```

20.6.3. TouchKit 设定

TouchKit 中存在很多设定, 你可以按照自己的需求进行定制. **TouchKitSettings** 配置信息对象由 **TouchKitServlet** 负责管理, 因此你可以任意修改它, 前文介绍已经过, 你需要实现一个自定义 Servlet.

```

public class MyServlet extends TouchKitServlet {
    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        TouchKitSettings s = getTouchKitSettings();
        ...
    }
}

```

设定项目中还包括一些 iOS 设备独有的设定, 这些设定信息包含在单独的 **IosWebAppSettings** 对象中, 可以通过 TouchKit 设定对象的 `getIosWebAppSettings()` 方法得到.

应用程序图标

移动设备的地址栏, 书签, 以及其他位置都可以显示 Web 应用程序的图标. 你可以在 **ApplicationIcons** 对象中设置图标(可以有多个), 这个对象负责管理不同分辨率下使用的各种图标. 尺寸与当前环境最适应的图标会被自动使用. iOS 设备上比较适当的图标尺寸是 57x57, 72x72, and 144x144 像素, Android 设备是 36x36, 48x48, 72x72, and 96x96 像素.

你可以使用 `addApplicationIcon()` 方法, 将图标添加到应用程序的图标集中. 应用程序的 URL 起始地址可以通过 Servlet Context 得到, 如下例.

```

TouchKitSettings s = getTouchKitSettings();
String contextPath = getServletConfig()
    .getServletContext().getContextPath();
s.getApplicationIcons().addApplicationIcon(
    contextPath + "VAADIN/themes/mytheme/icon.png");

```

这个方法的基本版本只接受图标名称作为参数, 另一个版本则允许你定义它的尺寸. 它还有一个 `preComposed` 参数, 当这个参数为 `true` 时, 会告诉 Safari 浏览器, 在 iOS 中对这个图标添加效果.

Viewport 设定

ViewPortSettings 对象, 可以通过 TouchKit 设定对象的 `getViewPortSettings()` 方法得到, 它负责管理与显示相关的设定, 其中最重要的是缩放限定.

```

TouchKitSettings s = getTouchKitSettings();
ViewPortSettings vp = s.getViewPortSettings();
vp.setViewPortUserScalable(true);
...

```

关于 iOS 浏览器中的这个功能, 详情请参见 Apple 开发者网站的 Safari 开发库.

iOS 中的启动画面图片

iOS 浏览器支持在应用程序装载过程中显示一个启动(splash)图片。你可以通过 **IosWebAppSettings** 对象的 `setStartupImage()` 方法来设置这个图片。应用程序的 URL 起始地址可以通过 Servlet Context 得到，如下例。

```
TouchKitSettings s = getTouchKitSettings();
String contextPath = getServletConfig().getServletContext()
    .getContextPath();
s.getIosWebAppSettings().setStartupImage(
    contextPath + "VAADIN/themes/mytheme/startup.png");
```

iOS 中的 Web 应用程序能力

iOS 支持一种特别的 Web 应用程序模式，可以从书签或从 Home 画面启动。当这个模式启用时，客户端可能会隐藏浏览器自身的 UI，以便为 Web 应用程序让出更多的屏幕空间，当然其他原因也可能导致这个效果。这个模式通过 Header 信息来启用，Header 信息负责告诉浏览器，这个应用程序是设计为以 Web 应用程序方式使用，还是作为 Web 页面来使用。

```
TouchKitSettings s = getTouchKitSettings();
s.getIosWebAppSettings().setWebAppCapable(true);
```

关于 iOS 浏览器中的这个功能，详情请参见 Apple 开发者网站的 Safari 开发库。

缓存 Manifest

ApplicationCacheSettings 对象管理缓存 Manifest，这个设置用来配置浏览器如何为这个 Web 应用程序缓存页面和其他资源。关于它的使用方法，详情请参见第 20.9 节“离线模式(Offline Mode)”。

20.6.4. UI

移动设备 UI 与通常的 Vaadin 应用程序一样，继承自 **UI** 类，并使用组件来构建 UI。

```
@Theme("mobiletheme")
@Widgetset("com.example.myapp.MyAppWidgetSet")
@Title("My Simple App")
public class SimplePhoneUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Create the content root layout for the UI
        TabBarView mainView = new TabBarView();
        setContent(mainView);

        ...
    }
}
```

由于 TouchKit 附带了自定义的 Widget Set，因此你需要为你的工程使用一个组合的 Widget Set，使用 `@Widgetset` 注解为 UI 定义 Widget Set。如过你安装了 TouchKit add-on，Vaadin Plugin for Eclipse 和 Maven 会自动生成组合 Widget Set 的描述文件。

通常情况下，你会组合使用 TouchKit 的三个主要组件作为 UI 的基础：**TabBarView**, **NavigationView**, 或 **NavigationManager**。

如果提供了离线 UI，那么需要在 UI 的初始化处理中启用这个离线 UI，详情请参见第 20.9 节“离线模式(Offline Mode)”。这段代码已经包含在由 Maven archetype 自动创建的工程框架代码中了。

20.6.5. 移动设备 Widget Set

TouchKit 包含 Widget Set, 因此需要为你的工程编译 Widget Set, 将 TouchKit 的 Widget Set 包含在内, 详情请参见第 17 章 使用 Vaadin Add-on. 无论你使用 Maven 还是 Eclipse plugin, 编译过程中都会自动产生工程 Widget Set 的描述文件.

注意, 如果你在非 TouchKit UI 的工程内使用了一部分 TouchKit UI, 你可能不希望将 TouchKit 的 Widget Set 编译到工程的 Widget Set 内. 由于自动生成的 Widget Set 描述文件会包含它在类路径中发现的一切 Widget Set, 因此结果可能是你不希望的, 这时你需要手工编辑 Widget Set 描述文件.

20.6.6. 移动设备 Theme

对于 TouchKit 应用程序, Sass 和 CSS 格式的 Theme 都可以使用, 虽然他们的定义方式与通常的 Vaadin Theme 略有不同. 为了优化 Theme 的装载过程, 你可以将 Theme 构建到 GWT client bundle 之内.

定义一个通常的 Theme

为移动设备应用程序定义一个简单的 Theme, 最简单的方式通常是使用单纯的 CSS, 因为使用 Sass 将无法象在通常的 Vaadin 应用程序中那样得到同样的益处. TouchKit 的 Widget Set 中包含了它自己的基础 Theme, 因此你不需要显式地 @import 它.

CSS Theme 定义在 VAADIN/themes/mymobiletheme/styles.css 文件中. 由于不需要(而且不应该) import 基础 Theme, 因此 CSS Theme 文件可以非常简单, 如下例:

```
.stylishlabel {
    color: red;
    font-style: italic;
}
```

和通常一样, 你需要在你的 UI 类中使用 @Theme ("mymobiletheme") 注解来设置 Theme.

你也可以使用 Sass, 首先创建 styles.scss 文件, 然后使用 Vaadin Theme 编译器将它编译为 CSS. 但是, 如上文所说, 你不应该 include 基础 Theme. CSS 规则不需要包装在 Theme 名称选择器之中, 但在通常的 Vaadin Theme 中是需要这样做的.

Responsive 移动设备 Theme

Responsive add-on 对于移动设备 Theme 非常有用, 因为通过它, 应用程序可以很方便地适应手机和平板上的不同布局, 而且可以适应屏幕方向的旋转. 使用这个 add-on, 当屏幕旋转方向变化时, 相应的 UI 布局变化, 完全由这个 add-on 在客户端处理, 使用 Theme 中的特殊的 CSS 选择器实现. 关于这个 add-on, 详情请参见第 7.9 节“条件式 Theme”.

Parking 示例程序使用了这个 add-on. 通过它的源代码(在 Github 上可以得到), 你可以学习如何使用条件选择器: GWT 客户端 bundle 中定义的 CSS.

比如, Parking 示例程序中 **Stats** tab 的 CSS, 定义了一个条件式的选择器, 如下所示, 目的是如果水平方向的空间足够时, 将两个图表并排显示:

```
.stats .statschart {
    margin-bottom: 30px;
    float: left;
    width: 100%;
}
```

```
.v-ui [width-range~="801px-"] .stats .statschart {
    width: 48% !important;
    margin: 0 1%;
}
```

通常, 如果屏幕水平宽度为 800 像素或更低, 那么每个图表都将占据 100% 的宽度, 因此第二个图表会折行, 显示在 **CssLayout** 容器中的下一行. 如果宽度超过 800 像素, 两个图表都会占据 48% 宽度, 因此它们都会显示在同一行内. 这种方式遵循灵活折行模式, 关于这种模式, 详情请参见“灵活折行”一节.

在 GWT Client Bundle 中定义 Theme

使用 GWT Theme 而不是通常的 Vaadin Theme, 这种方式会在移动设备上带来一些性能上的益处, 因为可以减少装载的资源数量. 所有的资源, 比如图片, 样式表, 都可以和 Widget Set 一起装载. 图片可以作为平铺(tile)在 Bundle 图片中的精灵(sprite)来处理.

GWT CSS 中的类有它们独有的格式, 与 Sass Theme 有些类似. 关于 Client Bundle, 以及如何定义图片, CSS, 以及其他资源, 详情请参见 GWT Developer's Guide.

要在 TouchKit 应用程序中使用 GWT Client Bundle, 你需要定义一个 *Theme* 装载器, 它继承自 TouchKit **ThemeLoader** 类, 并实现 `load()` 方法来注入 Bundle. *Theme* 装载器和 Client Bundle 都是客户端类, 会被编译进入 Widget Set, 因此必须定义在 `client` 目录之下.

For example, in the Parking Demo we have as follows:

```
public class ParkingThemeLoader extends ThemeLoader {
    @Override
    public final void load() {
        // First load the default TouchKit theme...
        super.load();

        // ... and add Parking Demo CSS from its own bundle
        ParkingBundle.INSTANCE.fontsCss().ensureInjected();
        ParkingBundle.INSTANCE.css().ensureInjected();
        ParkingBundle.INSTANCE.ticketsCss().ensureInjected();
        ParkingBundle.INSTANCE.statsCss().ensureInjected();
        ParkingBundle.INSTANCE.shiftsCss().ensureInjected();
        ParkingBundle.INSTANCE.mapCss().ensureInjected();
    }
}
```

你可以调用 `super.load()` 方法来装载默认的 TouchKit Theme, 但如果你不希望使用这个 Theme, 也可以省略这个调用. 这种情况下, 你的 GWT Theme 就应该显式地 import Vaadin 的基础 Theme.

Theme 装载器必须在 `.gwt.xml` Widget Set 描述文件中定义如下:

```
<replace-with
    class="com.vaadin.demo.parking.widgetset.client.theme.ParkingThemeLoader">

    <when-type-is
        class="com.vaadin.addon.touchkit.gwt.client.ThemeLoader" />
</replace-with>
```

关于如何定义 GWT theme, 完整的例子请参见 Parking 示例程序的源代码.

20.6.7. 使用字体图标

在大多数 TouchKit 组件中, 你也可以使用字体图标, 详情请参见 第 7.7 节 “字体图标”.

图 20.3. TabBarView 中的字体图标



比如，通过 Maven archetype 创建的工程中，UI 框架代码是这样实现的：

```
// Have a tab bar with multiple tab views
TabBarView tabBarView = new TabBarView();

// Have a tab
... create view1 ...
Tab tab1 = tabBarView.addTab(view1);

// Use the "book" icon for the tab
tab1.setIcon(FontAwesome.BOOK);
```

20.7. 移动设备 UI 组件

为了实现更好的用户交互，也为了使用移动设备的各种特殊功能，TouchKit 引入了很多移动设备 UI 的专用组件。

NavigationView

带导航条的视图。**NavigationBar** 类可以在 **NavigationManager** 内回退和前进)。

Toolbar

一个水平布局，专用于排列按钮。用作 **TabBarView** 或 **NavigationView** 中的子组件。

NavigationManager

一个组件容器，向服务器发送请求时，为了解决服务器响应延迟问题，它会在组件切换之间的等待过程中显示滑动式动画效果。这个容器内的组件通常是 **NavigationView** 或 **SwipeView**。

NavigationButton

一个特殊的按钮，它在客户端触发 **NavigationManager** 内的视图切换，用于解决服务器响应延迟问题。

Popover

一个浮动的，弹出式 frame，可以使用组件的相对位置来定位。

SwipeView

一种视图，使用水平滑动手势，在 **NavigationManager** 内回退或前进。

Switch

一种滑动式 on/off 切换按钮，用于输入布尔值。

VerticalComponentGroup

一种垂直布局，用于组件分组。

HorizontalButtonGroup

一种水平布局，用于特殊按钮的分组。

TabBarView

一种 tab 分页视图，它的内容区域位于上方，用于在各 tab 子视图之间切换的 **Toolbar** 位于下方。

EmailField, NumberField, 以及 UrlField

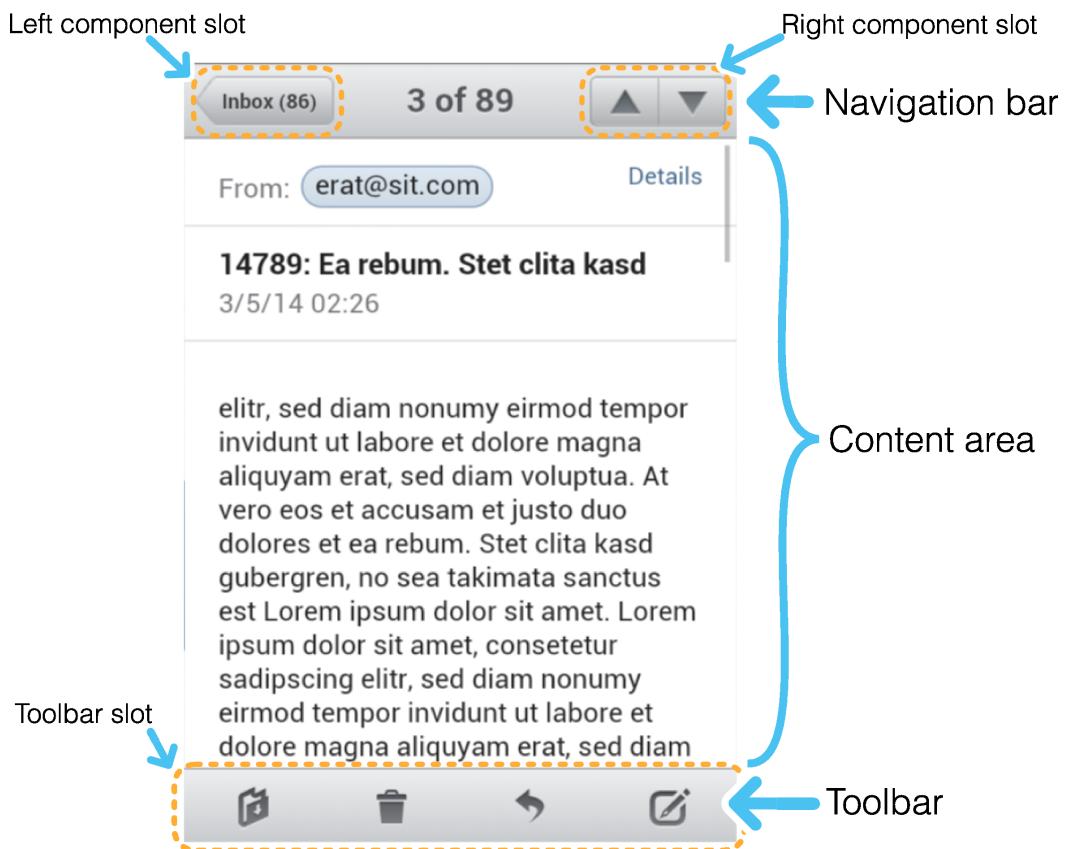
文本输入组件, 使用特殊的虚拟键盘, 分别输入 EMail 地址, 数字, 以及 URL.

以上各组件的细节, 将在后续小节中详情介绍.

20.7.1. NavigationView

NavigationView 是一个布局组件, 其中包含一个导航条和一个内容区域. 内容区域是可以滚动的, 因此不必再内嵌一个 panel 组件. 此外, 在 View 的底部还可以有一个可选的 toolbar 组件. **NavigationView** 通常用在 **NavigationManager** 之内, 以便得到视图变化时的动画效果.

图 20.4. NavigationView 的布局



NavigationView 默认为全尺寸, 内容区域会扩展, 占据导航条和工具条之外的所有空间.

导航条

NavigationView 上部的导航条是一个独立的 **NavigationBar** 组件, 其中包含两个组件区域, 左侧和右侧各有一个. 标题将显示在中间. **NavigationBar** 组件也可以单独使用.

当使用 **NavigationBar** 来控制导航, 并且使用 `setPreviousComponent()` 方法设置了前一个组件, 那么左侧的组件区域会自动出现一个 **Back** 按钮. 如果你在 **NavigationManager** 之内使用 **NavigationView**, 这个动作将会自动完成.

你可以通过 `getNavigationBar()` 方法得到导航条组件, 以便直接调用它的控制方法, 但 **NavigationView** 也提供了一些快捷方法: `setLeftComponent()`, `setRightComponent()`, 以及标题属性的 `set` 和 `get` 方法.

工具条

可选的工具条区域位于 **NavigationView** 的底部。工具条可以是任意组件，但 TouchKit 提供了 **Toolbar** 组件专用于实现工具条。关于这个组件，详情请参见 第 20.7.2 节 “**Toolbar**”。你也可以使用 **HorizontalLayout** 或 **CssLayout** 作为工具条。

工具条中的内容通常会使用有图标无标题的 **Button** 组件。可以通过 `setToolbar()` 方法来设置工具条。

使用 CSS 控制样式

```
.v-touchkit-navview { }
.v-touchkit-navview-wrapper {}
.v-touchkit-navview-toolbar {}
.v-touchkit-navview .v-touchkit-navview-notoolbar {}
```

根元素带有 `v-touchkit-navview` 样式。内容区域封装在 `v-touchkit-navview-wrapper` 样式元素之内。如果 View 带有工具条，工具条区域将带有 `v-touchkit-navview-toolbar` 样式，但如果沒有，则最顶层元素将带有 `v-touchkit-navview-notoolbar` 样式。

20.7.2. Toolbar

Toolbar 是一个水平布局组件，用于容纳 **Button** 组件。工具条默认占据 100% 宽度，并带有固定高度。内部的子组件将在水平方向上平均分布。**Toolbar** 会在 **TabBarView** 之内使用，详情请参见 第 20.7.10 节。

关于这个组件从父类中集成得到的功能，详情请参见 第 6.3 节 “**VerticalLayout** 和 **HorizontalLayout**”。

使用 CSS 控制样式

```
.v-touchkit-toolbar { }
```

这个组件的最外层样式为 `v-touchkit-toolbar`。

20.7.3. NavigationManager

NavigationManager 是一个视觉效果组件，在多个视图间切换时显示滑动式动画效果。你可以注册三个组件：当前显示的组件，位于左侧的前一个组件，位于右侧的下一个组件。你可以使用 `setCurrentComponent()`, `setPreviousComponent()`, 和 `setNextComponent()` 方法设置这些组件。

NavigationManager 组件见 图 20.5 “包含三个 **NavigationView** 的 **NavigationManager**”。

图 20.5. 包含三个 NavigationView 的 NavigationManager



导航管理器对于提高应用程序的相应速度是很重要的，因为前一个和后一个组件已存在于缓存中，而且在访问服务器，装载新的前一个或后一个组件时，还会显示滑动式动画效果。

你需要在这个组件的构造函数中用参数指定初始视图。通常会将导航管理器用作 UI 内容，或者用在 **TabBarView** 之内。

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        NavigationManager manager =
            new NavigationManager(new MainView());
        setContent(manager);
    }
}
```

切换视图

在多个视图(组件)之间切换，通常使用预定义的导航目标来实现，以便提高相应速度。点击 **NavigationButton**，也就是导航条上的按钮，会自动启动导航跳转，而不必发起一次服务器请求。屏幕滑动手势通过 **SwipeView** 组件来支持。

导航跳转也可以使用 `navigateTo()` 方法编程实现。如果启动了面包屑导航(Breadcrumb)，那么当前视图也会被压入到面包屑导航的历史堆栈中。要回退跳转，你可以调用 `navigateBack()` 方法，如果在 **NavigationView** 之内点击了 **Back** 按钮，也会隐含地调用这个方法。而且，如果导航跳转到了“前一个”组件，实际上就是隐含地调用了 `navigateBack()` 方法。

当导航跳转发生时，当前组件会被移动，变为前一个或后一个组件，具体变为哪一个，取决于跳转方向是向前还是向后。

处理视图变化事件

你可以向导航管理器添加任何类型的组件,但如果使用 **NavigationView**,可以启用一些特殊功能.当一个视图变为可见时,这个视图的 `onBecomingVisible()` 方法会被调用.你可以重载这个方法,但要记得调用父类的方法.

```
@Override
protected void onBecomingVisible() {
    super.onBecomingVisible();

    ...
}
```

此外,你还可以使用 `NavigationListener` 来处理导航管理器中的跳转变更事件. `direction` 属性可用来判断在面包屑导航历史堆栈中的跳转方向是向前还是向后,也就是说,导航跳转是 `navigateTo()` 还是 `navigateBack`. 当前组件,可以通过 `getCurrentComponent()` 方法得到,得到的结果指向导航跳转的目标组件.

```
manager.addNavigationListener(new NavigationListener() {
    @Override
    public void navigate(NavigationEvent event) {
        if (event.getDirection() ==
            NavigationEvent.Direction.BACK) {
            // Do something
            Notification.show("You came back to " +
                manager.getCurrentComponent().getCaption());
        }
    }
});
```

追踪面包屑导航历史

NavigationManager 还会对面包屑导航进行追踪. `navigateTo()` 方法会将当前视图压入面包屑导航历史堆栈的顶部,调用 `navigateBack()` 方法可以回退到面包屑导航历史的前一级.

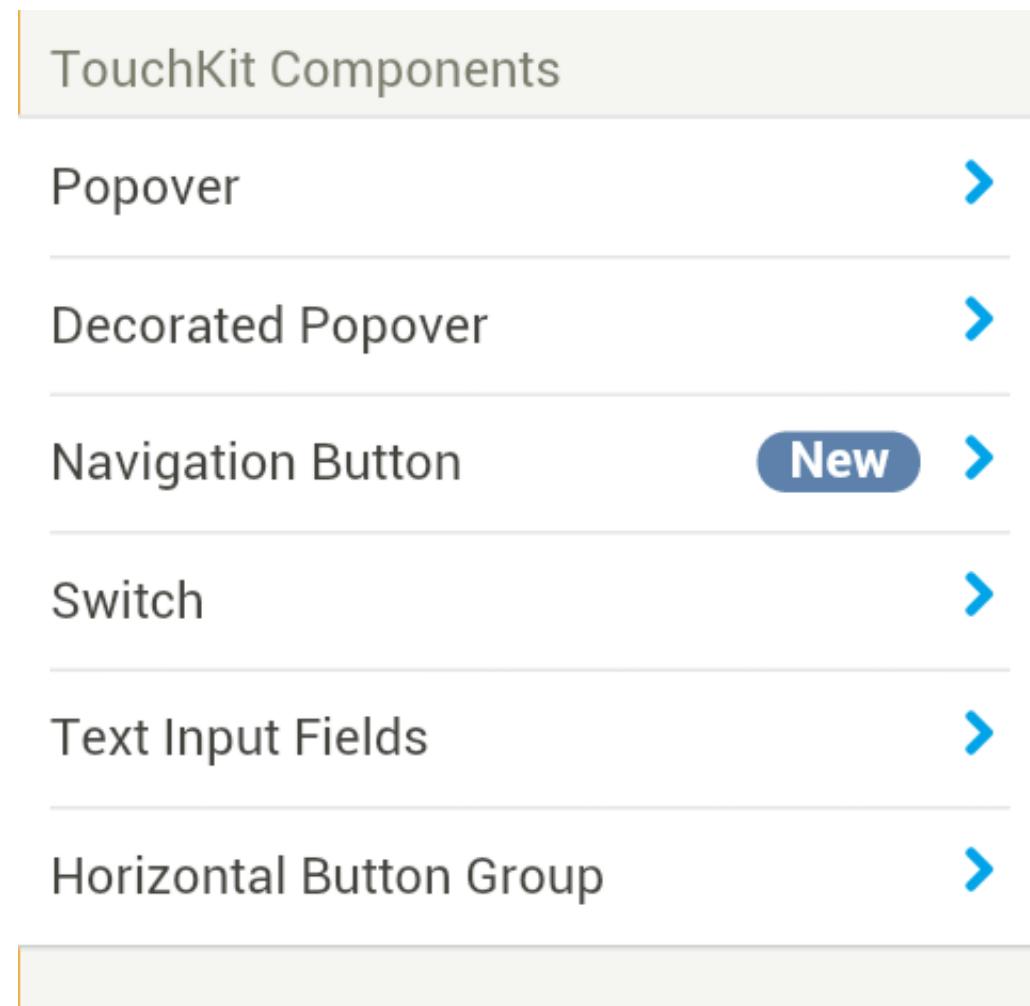
注意,以"前一个"组件为参数调用 `navigateTo()` 方法,等于调用 `navigateBack()`.

20.7.4. NavigationButton

NavigationButton 是通常的 **Button** 组件的一个特殊版本,用于在 **NavigationManager** 之内导航跳转(详情请参见第 20.7.3 节“**NavigationManager**”).点击一个导航按钮会自动跳转到一个预定义的目标视图.视图切换时的动画效果不必等待服务器请求结束,可以在按钮点击后立即显示.如果你没有定义跳转的目标视图,动画中会显示一个空的占位视图.当从服务器得到真实内容后,这个内容会填充到视图之内.

导航按钮默认不带边框,因为通常会在 **VerticalComponentGroup** 内使用多个导航按钮来创建菜单,见图 20.6 “垂直的组件 Group 内的 **NavigationButton**”.

图 20.6. 垂直的组件 Group 内的 NavigationButton



导航按钮带有标题，也可以有描述信息和图标。如果导航视图在按钮之前初始化完毕，而且没有明确给出按钮的标题，那么按钮会使用对应的导航视图的标题。图标显示在标题左侧，描述信息在按钮右侧。

你可以通过构造函数来指定跳转的目标视图，也可以通过 `setTargetView()` 方法，或者在按钮点击事件中再创建视图。

```
// Button caption comes from the view caption
box.addComponent(new NavigationButton(new PopoverView()));

// Give button caption explicitly
box.addComponent(new NavigationButton("Decorated Popover",
    new DecoratedPopoverView()));
```

如果在按钮点击之前，目标视图没有创建或初始化，那么在视图切换的动画中将没有标题。默认行为是使用按钮的标题作为目标视图的临时标题，但你也可以使用 `setTargetViewCaption()` 方法明确设置这个标题。临时标题会在视图切换时的滑动式动画效果显示期间内显示，直到从服务器端得到视图内容后，才会消失。然后会被视图真实的标题替换，你通常会希望这两个标题相同。如果导航按钮的标题未被明确指定，视图的临时标题还会被用作按钮标题。

```
final NavigationButton navButton = new NavigationButton();
navButton.setTargetViewCaption("Text Input Fields");
navButton.addClickListener(
    new NavigationButtonClickListener() {

        @Override
        public void buttonClick(NavigationButtonClickEvent event) {
            navButton.getNavigationManager()
                .navigateTo(new FieldView());
        }
    });
box.addComponent(navButton);
```

我们推荐通过这种方式来动态创建视图，因为可以减少内存消耗量。

注意，只有按钮在 **NavigationManager** 之内(也就是在它管理的视图之内)时，自动导航跳转才会有效。如果你只希望将导航按钮用作一个可视元素，你可以象通常的 **Button** 一样使用它，并通过 **ClickListener** 来处理它的点击事件。

使用 CSS 控制样式

```
.v-touchkit-navbutton { }
.v-touchkit-navbutton-desc { }
.v-icon { }
```

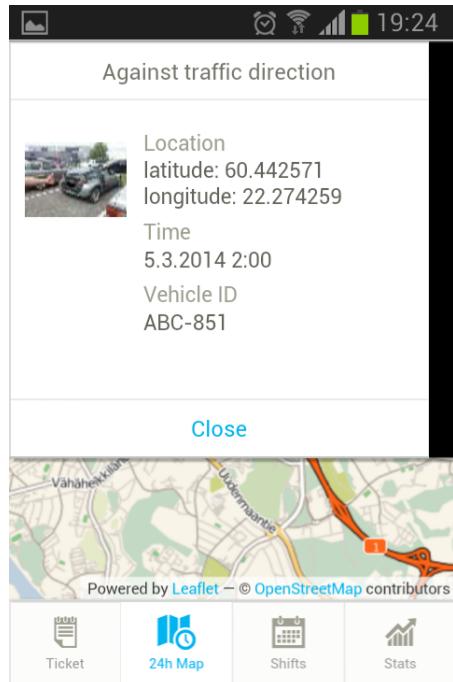
这个组件的最外层样式是 `v-touchkit-navbutton`。如果使用 `setDescription()` 方法为这个组件设置了描述信息，描述信息将显示在一个单独的 `` 元素内，这个元素将带有 `v-touchkit-navbutton-desc` 样式。描述信息还会带有一个可选的 `emphasis` 样式，还可以带有一个更粗的，圆角的，胶囊形状的 `pill` 样式，你可以通过 `addStyleName()` 方法添加这个样式。

导航按钮默认样式的设计目的，是为了将按钮放置在 **VerticalComponentGroup** 之内。当它放置在 **HorizontalButtonGroup** 之内时，以及在 **NavigationBar** 的左侧或右侧区域时，都会有不同的样式。

20.7.5. Popover

Popover 与通常的 Vaadin 子窗口非常类似，可以用于快速显示某种选项，或者与某个动作相关的小的 Form。与通常的子窗口不同，它不允许用户拖放位置或拖动大小。由于子窗口通常需要更大的屏幕尺寸，**Popover** 比子窗口更适合于平板设备。当在更小的设备上使用时，比如手机，**Popover** 会自动充满整个屏幕。

图 20.7. 手机上的 Popover



我们通常会使用 **NavigationView**, 以便实现边框装饰效果和标题. 下例中, 我们继承一个 **Popover** 来创建其中的内容.

```
class DetailsPopover extends Popover {
    public DetailsPopover() {
        setWidth("350px");
        setHeight("65%");

        // Have some details to display
        VerticalLayout layout = new VerticalLayout();
        ...
        NavigationView c = new NavigationView(layout);
        c.setCaption("Details");
        setContent(c);
    }
}
```

可以使用 `showRelativeTo()` 方法, 在相对于某个组件的位置上打开 **Popover**. 下例中, 我们在 Table 中的项目被点击时打开 popover.

```
Table table = new Table("Planets", planetData());
table.addItemClickListener(new ItemClickListener() {
    @Override
    public void itemClick(ItemChangeEvent event) {
        DetailsPopover popover = new DetailsPopover();

        // Show it relative to the navigation bar of
        // the current NavigationView.
        popover.showRelativeTo(view.getNavigationBar());
    }
});
```

```

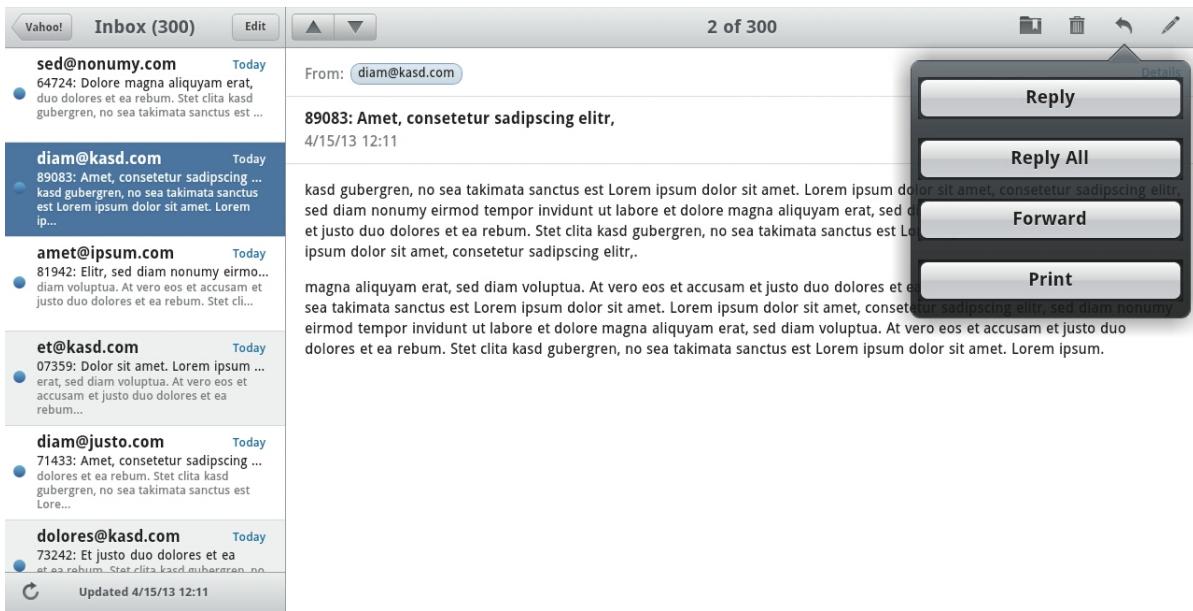
    }
});

```

也可以使用 `addWindow()` 方法将 **Popover** 添加到 **UI** 内.

Popover 显示在平板设备上的效果, 见图 20.8 “显示在平板设备上的 **Popover**”. 在这个示例中, **Popover** 中的内容是一个 **CssLayout**, 以及一些按钮.

图 20.8. 显示在平板设备上的 **Popover**



关闭一个 **Popover**

Popover 的 `closable` 选项默认是启用的, 这时可以点击弹出区域之外的任何地方来关闭它. 如果 **Popover** 充满整个屏幕, 这个功能就无法使用了, 用户将无法继续操作. **Popover** 可以通过程序调用 `close()` 方法来关闭. 比如, 你可以为 **Popover** 添加一个 `MouseEvents.ClickListener` 监听器, 允许用户点击 **Popover** 内部区域时关闭它.

如果 **Popover** 中包含可编辑的 **Field**, 你可能会希望在 **NavigationView** 的导航条中带有一个关闭按钮. 下例中, 我们在导航条的右侧区域中添加了一个关闭按钮(你需要在你的 **Theme** 中包含图标).

```

class DetailsPopover extends Popover
    implements Button.ClickListener {
    public DetailsPopover(Table table, Object itemId) {
        setWidth("350px");
        setHeight("65%");
        Layout layout = new FormLayout();
        ... create the content ...

        // Decorate with navigation view
        NavigationView content = new NavigationView(layout);
        content.setCaption("Details");
        setContent(content);

        // Have a close button
        Button close = new Button(null, this);
        close.setIcon(new ThemeResource("close64.png"));
    }
}

```

```

        content.setRightComponent(close);
    }

    public void buttonClick(ClickEvent event) {
        close();
    }
}

```

使用 CSS 控制样式

```

.v-touchkit-popover .v-touchkit-fullscreen { }
.v-touchkit-popover .v-touchkit-relative { }
.v-touchkit-popover .v-touchkit-plain { }

```

这个组件的最外层样式是 `v-touchkit-popover`. 如果处于全屏模式, 它还会带有 `v-touchkit-fullscreen` 样式, 如果它在相对于某个组件的位置上弹出, 它会带有 `v-touchkit-relative` 样式, 否则会带有 `v-touchkit-plain` 样式.

20.7.6. SwipeView

SwipeView 是一个封装容器, 允许通过向左或向右的水平滑动手势来实现视图之间的导航跳转. 这个组件与 **NavigationManager** 一起配合工作 (详情请参见第 20.7.6 节 “**SwipeView**”), 实现滑动时的视图切换, 以及视图切换时的动画效果. **SwipeView** 应该是 **NavigationManager** 的直接子元素, 但在它内部也可以包含 **NavigationView**, 来实现按钮方式的导航跳转.

下面我们来实现一个选择照片的浏览界面. 我们继承 **NavigationManager** 来实现滑动效果, 然后动态创建一个真实的图片查看视图. 在构造函数中, 我们创建前两幅图片的查看视图.

```

class SlideShow extends NavigationManager
    implements NavigationListener {
    String imageNames[] = {"Mercury.jpg", "Venus.jpg",
        "Earth.jpg", "Mars.jpg", "Jupiter.jpg",
        "Saturn.jpg", "Uranus.jpg", "Neptune.jpg"};
    int pos = 0;

    public SlideShow() {
        // Set up the initial views
        navigateTo(createView(pos));
        setNextComponent(createView(pos+1));

        addNavigationListener(this);
    }
}

```

各个图片查看视图中包含一个 **SwipeView** and the top(译注: 此句貌似原文有错误, 待校).

```

SwipeView createView(int pos) {
    SwipeView view = new SwipeView();
    view.setSizeFull();

    // Use an inner layout to center the image
    VerticalLayout layout = new VerticalLayout();
    layout.setSizeFull();

    Image image = new Image(null, new ThemeResource(
        "planets/" + imageNames[pos]));
    layout.addComponent(image);
    layout.setComponentAlignment(image,
        Alignment.MIDDLE_CENTER);
}

```

```

        view.setContent(layout);
        return view;
    }
}

```

当视图被朝左侧或右侧滑动时，我们需要在 **NavigationManager** 中动态地设置滑动方向上的下一幅图片。

```

@Override
public void navigate(NavigationEvent event) {
    switch (event.getDirection()) {
        case FORWARD:
            if (++pos < imageNames.length-1)
                setNextComponent(createView(pos+1));
            break;
        case BACK:
            if (--pos > 0)
                setPreviousComponent(createView(pos-1));
    }
}

```

20.7.7. Switch

Switch 组件是一个二状态选择器，外观类似于 Apple iOS 中的切换按钮，它的值可以通过点击或滑动来切换。它继承自 **CheckBox**，因此值类型为 **Boolean** 型。它的标题由包含它的布局组件来管理。

```

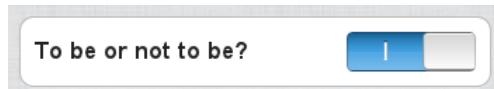
VerticalComponentGroup group =
    new VerticalComponentGroup();
Switch myswitch = new Switch("To be or not to be?");
myswitch.setValue(true);
group.addComponent(myswitch);

```

和其他的 Field 组件一样，你可以使用 **ValueChangeListener** 来处理值的变更事件。使用 **setImmediate(true)** 方法，可以将它设置为立即模式，在输入值发生变化时立即触发事件。

运行结果见图 20.9 “**Switch**”。

图 20.9. Switch



使用 CSS 控制样式

```
.
.v-touchkit-switch { }
.v-touchkit-switch-slider { }
```

这个组件的最外层样式是 **v-touchkit-switch**。滑块元素的样式是 **v-touchkit-switch-slider**。

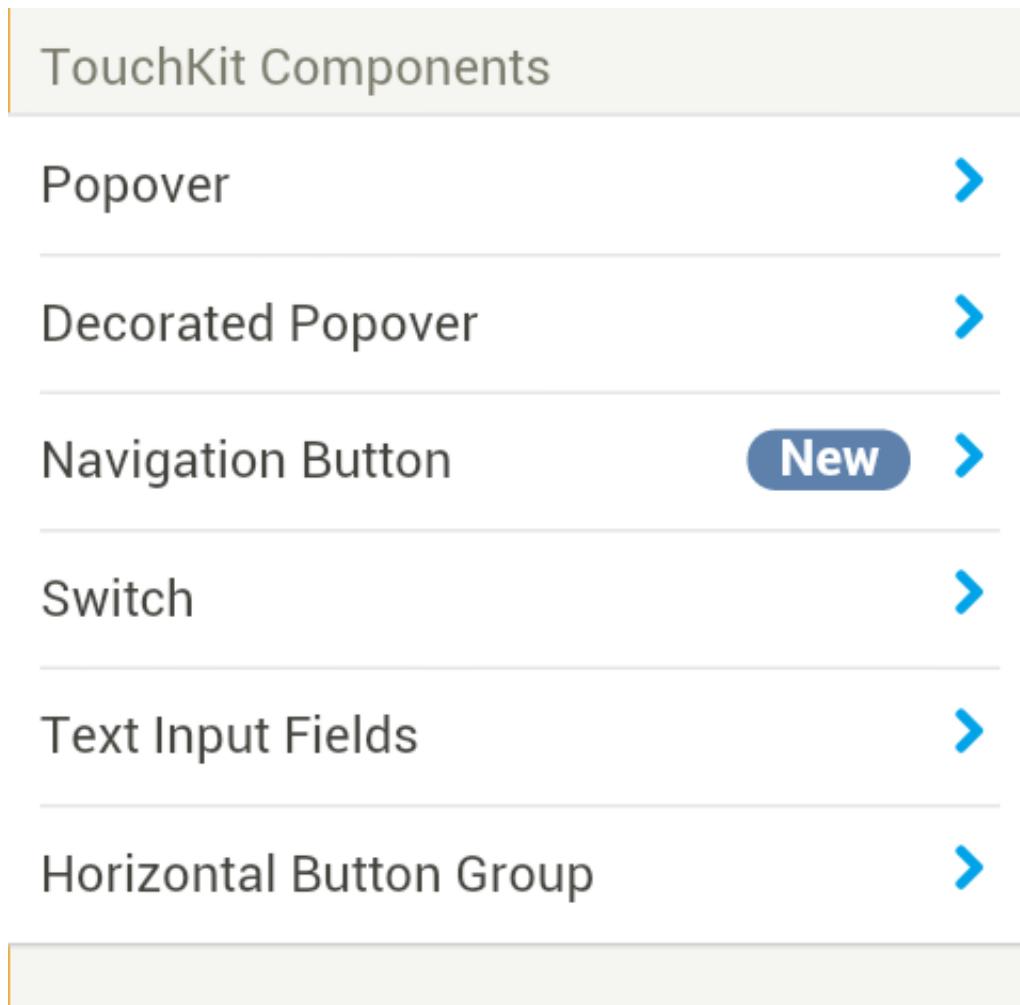
20.7.8. VerticalComponentGroup

VerticalComponentGroup 是一个布局组件，用于将组件以垂直层叠的方式组织在一起，并带有边框。组件标题放在组件左侧，组件本身则朝右侧对齐。组件 Group 通常用在 Form 内，或者和 **NavigationBar** 一起使用，创建导航菜单。

```
VerticalComponentGroup group =  
    new VerticalComponentGroup("TouchKit Components");  
group.setWidth("100%");  
  
// Navigation to sub-views  
group.addComponent(new NavigationButton(  
    new PopoverView()));  
group.addComponent(new NavigationButton(  
    new DecoratedPopover()));  
  
layout.addComponent(box);
```

运行结果见 图 20.10 “**VerticalComponentGroup**”。

图 20.10. **VerticalComponentGroup**



使用 CSS 控制样式

```
.v-touchkit-verticalcomponentgroup { }
```

这个组件的最外层样式为 v-touchkit-verticalcomponentgroup。如果组件带有标题，则会添加 v-touchkit-has-caption 样式。

20.7.9. HorizontalButtonGroup

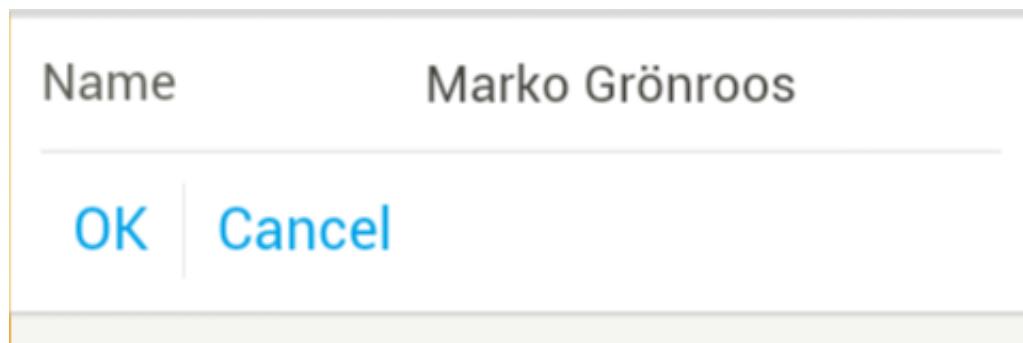
HorizontalButtonGroup 用于在 **VerticalComponentGroup** 之内将按钮组织在一起，并使用一种特殊的按钮 Group 样式。

```
VerticalComponentGroup vertical =
    new VerticalComponentGroup();
vertical.addComponent(new TextField("Name"));

HorizontalButtonGroup buttons =
    new HorizontalButtonGroup();
buttons.addComponent(new Button("OK"));
buttons.addComponent(new Button("Cancel"));
vertical.addComponent(buttons);
```

运行结果见 图 20.11 “HorizontalButtonGroup”

图 20.11. HorizontalButtonGroup



你也可以将单个的按钮包装在这个组件之内，以美化按钮的显示效果。**Upload** 组件也带有按钮，你可以为这个按钮添加 `v-button` 样式，使它的外观与 Button Group 内的按钮一致，详情请参见第 20.8.4 节“上传内容”。

Button Group 的设计目的是为了组织按钮，但你实际上可以不理会这一点，在这个组件内部放置任意类型的组件。这样做的结果是否有意义，取决于具体的组件。

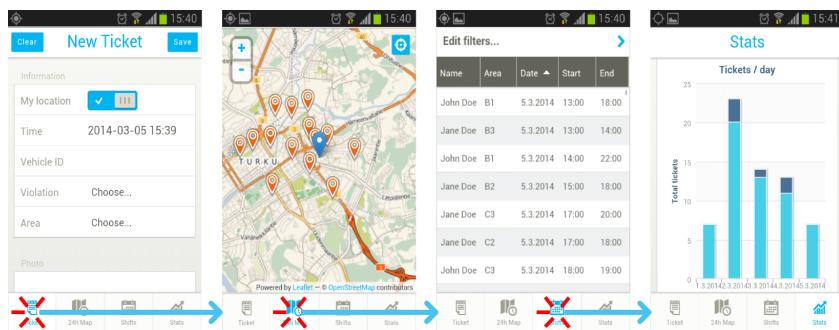
使用 CSS 控制样式

```
.v-touchkit-horizontalbuttongroup { }
```

这个组件的最外层样式为 `v-touchkit-horizontalbuttongroup`。我们前面提到过，TouchKit 的样式表，对于 Group 之内的带有 `v-button` 样式的组件，包含了一些特殊规则。

20.7.10. TabBarView

TabBarView 是一个布局组件，包括屏幕底部的 Tab 条和内容区域。每个 Tab 都包含自己的内容组件，当 Tab 选中时就会显示对应的内容组件。

图 20.12. 包含四个 **NavigationView** 的 **TabBar**

TabBarView 实现了 `ComponentContainer` 接口, 但使用它自己独自的 API 来操纵 Tab. 要添加新 Tab, 你需要调用 `addTab()`, 参数是新 Tab 的内容组件. 这个方法将创建 Tab 并返回一个 **Tab** 对象来管理这个新 Tab. 对于一个 Tab, 你至少需要设置它的标题和图标.

```
TabBarView bar = new TabBarView();

// Create some Vaadin component to use as content
Label content = new Label("Really simple content");

// Create a tab for it
Tab tab = bar.addTab(label);

// Set tab name and/or icon
tab.setCaption("tab name");
tab.setIcon(new ThemeResource(...));
```

Tab 可以通过 `removeTab()` 方法删除. 注意, 如果使用 `ComponentContainer` 中的 `addComponent()` 和 `removeComponent()` 方法, 会抛出 **UnsupportedOperationException** 异常.

改变选中的 Tab

当前选中的 Tab 可以通过 `getSelectedTab()` 方法得到, 也可以通过 `setSelectedTab()` 方法来设置. 无论是用户操作还是通过程序改变选中的 Tab, 都会导致一个 `SelectedTabChangeEvent` 事件, 你可以通过 `SelectedTabChangeListener` 来处理这个事件.

```
Tab selectedTab = bar.getSelectedTab();
bar.setSelectedTab(selectedTab);
```

使用 CSS 控制样式

```
.v-touchkit-tabbar {}
.v-touchkit-tabbar-wrapper {}
.v-touchkit-tabbar-toolbar {}
```

这个组件的最外层样式为 `v-touchkit-tabbar`. 内容区域封装在 `v-touchkit-tabbar-wrapper` 样式的元素之内. Tab 条控制区域本身的样式为 `v-touchkit-tabbar-toolbar`.

20.7.11. EmailField

EmailField 与通常的 **TextField** 很类似，区别是它关闭了自动首字母大写和自动校正功能。移动设备也会将这个组件识别为一个 EMail Field，并会显示一个虚拟键盘用于输入 EMail，因此这个虚拟键盘会包括 @ 符号和点号(.)，可能还会包含一个 .com 的快捷键。

图 20.13. 编辑中的 EmailField



使用 CSS 控制样式

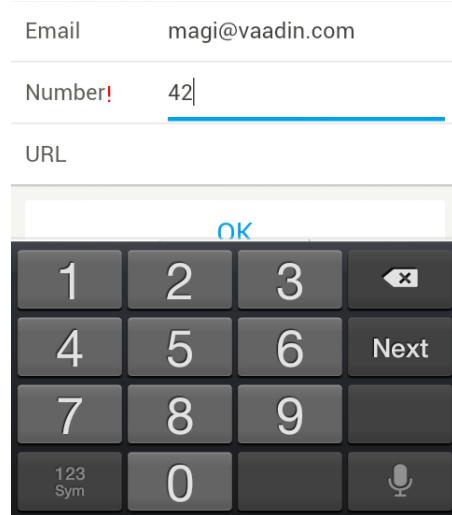
```
.v-textfield {}  
.v-textfield.v-textfield-error {}
```

EmailField 的最外层样式是 v-textfield，与通常的 **TextField** 组件一样。如果组件中包含错误，比如校验失败时，它还会带有 v-textfield-error 样式。

20.7.12. NumberField

NumberField 与通常的 **TextField** 类似，区别是它被标记为移动设备上的数字输入 Field，因此会显示虚拟的数字键盘，而不是默认的字母-数字键盘。

图 20.14. 编辑中的 NumberField



使用 CSS 控制样式

```
.v-textfield {}
.v-textfield.v-numberfield-error {}
```

NumberField 组件的最外层样式为 `v-textfield`, 与通常的 **TextField** 组件一样. 如果组件中包含错误, 比如校验失败时, 它还会带有 `v-numberfield-error` 样式.

20.7.13. UrlField

UrlField 与通常的 **TextField** 类似, 区别是它被标记为移动设备上的 URL 输入 Field, 因此会显示一个 URL 输入用的虚拟键盘, 而不是默认的字母-数字键盘. 这个组件有一个便捷方法 `getUrl()` 和 `setUrl(URL url)`, 用于将输入的值与 `java.net.URL` 类型相互转换.

使用 CSS 控制样式

```
.v-textfield {}
.v-textfield.v-textfield-error {}
```

UrlField 的最外层样式是 `v-textfield`, 与通常的 **TextField** 组件一样.

20.8. 移动设备高级特性

20.8.1. 提供一个备用(Fallback)UI

你可能会需要对移动设备的 TouchKit UI 和通常的浏览器使用同一个 URL, 因此也就是使用同一个 Servlet. 这种情况下, 你需要识别出兼容 Vaadin TouchKit 的移动设备浏览器, 并对其他浏览器提供一个备用(Fallback) UI. 备用 UI 可以是通常的 Vaadin UI, 可以是一条 "Sorry!" 信息, 或者重定向到另一个 UI.

你可以在 Servlet 的负责创建 UI 的自定义 **UIProvider** 内处理 Fallback 逻辑. 由于 TouchKit 只支持基于 WebKit 的浏览器以及 Windows Phone 的浏览器, 你可通过检查 `user-agent` 字符串中是否存在 "webkit" 或 "windows phone" 子串来识别浏览器类型, 如下例:

```

public class MyUIProvider extends UIProvider {
    @Override
    public Class<? extends UI>
        getUIClass(UIClassSelectionEvent event) {
        String ua = event.getRequest()
            .getHeader("user-agent").toLowerCase();
        if (ua.toLowerCase().contains("webkit")
            || ua.toLowerCase().contains("windows phone 8")
            || ua.toLowerCase().contains("windows phone 9")) {
            return MyUI.class;
        } else {
            return MyFallbackUI.class;
        }
    }
}

```

自定义的 UI Provider 必须添加到自定义的 Servlet 类中, 你需要在 web.xml 中定义这个 Servlet 类, 详情请参见 第 20.6.3 节 “TouchKit 设定”. 如下例:

```

public class MyServlet extends TouchKitServlet {
    private MyUIProvider uiProvider = new MyUIProvider();

    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();

        getService().addSessionInitListener(
            new SessionInitListener() {
                @Override
                public void sessionInit(SessionInitEvent event)
                    throws ServiceException {
                    event.getSession().addUIProvider(uiProvider);
                }
            });
    }

    ... other custom servlet settings ...
}

```

实际可运行的例子, 请参见 Parking 示例程序.

20.8.2. 地理位置

TouchKit 中的地理位置功能可以从移动设备得到地理位置信息, 浏览器将会询问用户, 要求确认是否允许当前网站获取位置信息. 点击 **Share Location** 按钮会允许网站获取位置信息. 浏览器将向服务器报告位置信息, 位置信息通过 GPS 定位, 移动电话蜂窝网络定位, 或 Wi-Fi 定位来获取, 具体使用何种手段由移动设备的许可决定.

地理位置通过调用 **Geolocator** 中的静态方法 `detect()` 来取得. 你需要提供一个 **PositionCallback** 处理程序, 当移动设备对你的请求有相应时, 这个处理程序将被调用. 如果地理位置定位请求成功, 将会调用 `onSuccess()` 方法. 如果请求失败, 比如, 如果用户没有允许共享他的位置信息, 将会调用 `onFailure()` 方法. 地理位置数据将以 **Position** 对象的形式提供.

```

Geolocator.detect(new PositionCallback() {
    public void onSuccess(Position position) {
        double latitude = position.getLatitude();
        double longitude = position.getLongitude();
        double accuracy = position.getAccuracy();
    }
})

```

```

    ...
}

public void onFailure(int errorCode) {
    ...
}
);

```

位置信息以带小数的度单位数值表示, 与 GPS 一样, 使用 WGS84 坐标系. 经度数值的正数表示东经(WGS84 坐标系的本初子午线以东), 负数表示西经(本初子午线以西). 精度值单位为米. 此外, 还提供了以下数据:

- 海拔高度
- 海拔高度精度
- 移动方向
- 移动速度

如果位置信息数据中的任何一项无法取得, 它的值将为 0.

如果因为某种原因无法取得地理位置信息, 则会调用 `onFailure()` 方法. `errorCode` 代表错误原因. 如果没有地理位置信息的取得权限, 会返回错误号 1, 如果地理位置信息不可用, 会返回错误号 2, 定位处理超时, 会返回错误号 3, 未知错误, 会返回错误号 0.

注意, 地理位置定位处理有可能耗费很长的时间, 具体如何取决于移动设备使用的定位方法. 使用 Wi-Fi 网络定位或移动电话蜂窝网络定位时, 定位时间通常少于 30 秒. 在新的设备上, 单独使用 GPS 初次进行定位时, 如果 GPS 接收信号较差, 定位时间可能会长达 15 分钟, 甚至更长. 但是, 一旦定位完成, 之后的位置更新会很迅速. 如果你在开发导航软件, 你需要多次调用 **Geolocator** 类中的 `detect()` 方法来频繁地更新位置信息数据.

在地图上显示当前位置

地理位置信息通常使用地图来表示. 有无数种方法来实现, 比如, 在 Parking 示例程序中我们使用 V-Leaflet add-on 组件来实现.

注意, 地理位置信息给出的位置信息使用的是 WGS84 坐标系, 与 GPS 一样. 很多 Internet 地图服务使用的也是同样的坐标系, 但并不保证所有的地图服务都是如此. 不仅如此, 在某些国家, 甚至还有法律要求地图数据不能给出正确坐标, 而必须包含一定程度的偏差.

Parking 示例程序中的 **MapView** 是一个 TouchKit 导航视图, 它使用 add-on 中的 **LMap** 组件来显示地图:

```

public class MapView extends CssLayout
    implements PositionCallback, LeafletClickListener {
    private LMap map;
    private final LMarker you = new LMarker();
    ...

```

当按钮按下时, 就会向移动设备请求位置信息:

```

locatebutton = new Button("", new ClickListener() {
    @Override
    public void buttonClick(final ClickEvent event) {
        Geolocator.detect(MapView.this);
    }
}

```

```

});  
locatebutton.addStyleName("locatebutton");  
locatebutton.setWidth(30, Unit.PIXELS);  
locatebutton.setHeight(30, Unit.PIXELS);  
locatebutton.setDisableOnClick(true);  
addComponent(locatebutton);

```

当 TouchKit 得到位置信息时, 我们相应地移动地图位置, 如下:

```

@Override  
public void onSuccess(final Position position) {  
    ParkingUI app = ParkingUI.getApp();  
    app.setCurrentLatitude(position.getLatitude());  
    app.setCurrentLongitude(position.getLongitude());  
  
    setCenter();  
  
    // Enable centering on current position manually  
    locatebutton.setEnabled(true);  
}  
  
private void setCenter() {  
    if (map != null)  
        map.setCenter(you.getPoint());  
}

```

20.8.3. 在本地存储中保存数据

LocalStorage UI 扩展允许服务器端应用程序使用 HTML5 本地存储来保存数据. Storage 是一个单例, 你可以使用 LocalStorage.get() 方法得到它.

```
final LocalStorage storage = LocalStorage.get();
```

存储数据

你可以使用 put() 方法, 用键-值对(key-value pair)形式在本地存储中保存数据. 键和值都必须是字符串. 保存数据会导致一次客户端调用, 因此存储操作的成功或失败可以通过可选的 LocalStorageCallback 来处理.

```

// Editor for the value to be stored  
final TextField valueEditor = new TextField("Value");  
valueEditor.setNullRepresentation("");  
layout.addComponent(valueEditor);  
  
Button store = new Button("Store", new ClickListener() {  
    @Override  
    public void buttonClick(ClickEvent event) {  
        storage.put("value", valueEditor.getValue(),  
            new LocalStorageCallback() {  
                @Override  
                public void onSuccess(String value) {  
                    Notification.show("Stored");  
                }  
  
                @Override  
                public void onFailure(FailureEvent error) {  
                    Notification.show("Storing Failed");  
                }  
            });

```

```

        }
    });
    layout.addComponent(store);
}

```

从存储中取得数据

你可以使用 `get()` 方法从本地存储中取得数据。这个方法的参数是数据的键，以及一个 `LocalStorageCallback`，这个回调程序负责接收取得的值，或者失败信息。将值取得到服务器端需要一次客户端调用，还需要在回调程序中发起一次服务器端请求来发送数据值。

```

storage.get("value", new LocalStorageCallback() {
    @Override
    public void onSuccess(String value) {
        valueEditor.setValue(value);
        Notification.show("Value Retrieved");
    }

    @Override
    public void onFailure(FailureEvent error) {
        Notification.show("Failed because: " +
            error.getMessage());
    }
});

```

20.8.4. 上传内容

从移动设备上传内容与通常的 Vaadin 应用程序一样，使用 **Upload** 组件实现。

但在离线 UI 或客户端代码中，你需要用别的方式来实现内容的上传，使用一个特殊的上传 Widget 或上传处理程序。

服务器端 **Upload** 组件

在服务器端 UI 中，你可以使用通常的 **Upload** 组件，详情请参见 第 5.24 节 “**Upload**”。选择文件时，移动设备将会要求用户选择文件，文件来源是本地文件系统，相册，摄像头，会其他可能的来源，具体情况由移动设备决定。通常使用中唯一的区别是，上传组件必须处于 *立即(immediate)* 模式。

大多数移动设备操作系统都支持上传，比如 iOS, Android，以及 Windows RT 移动设备，但某些操作系统不支持，比如 WP8。

下例演示如何实现一个简单的文件上传，并将上传的内容保存到内存中。

```

// Display the image - only a placeholder first
final Image image = new Image();
image.setWidth("100%");
image.setVisible(false);
layout.addComponent(image);

// Implement both receiver that saves upload in a file and
// listener for successful upload
class ImageUploader implements Receiver, SucceededListener,
    ProgressListener {
    final static int maxLength = 10000000;
    ByteArrayOutputStream fos = null;
    String filename;
    Upload upload;

    public ImageUploader(Upload upload) {

```

```
        this.upload = upload;
    }

    public OutputStream receiveUpload(String filename,
                                      String mimeType) {
        this.filename = filename;
        fos = new ByteArrayOutputStream(maxLength + 1);
        return fos; // Return the output stream to write to
    }

    @Override
    public void updateProgress(long readBytes,
                               long contentLength) {
        if (readBytes > maxLength) {
            Notification.show("Too big content");
            upload.interruptUpload();
        }
    }

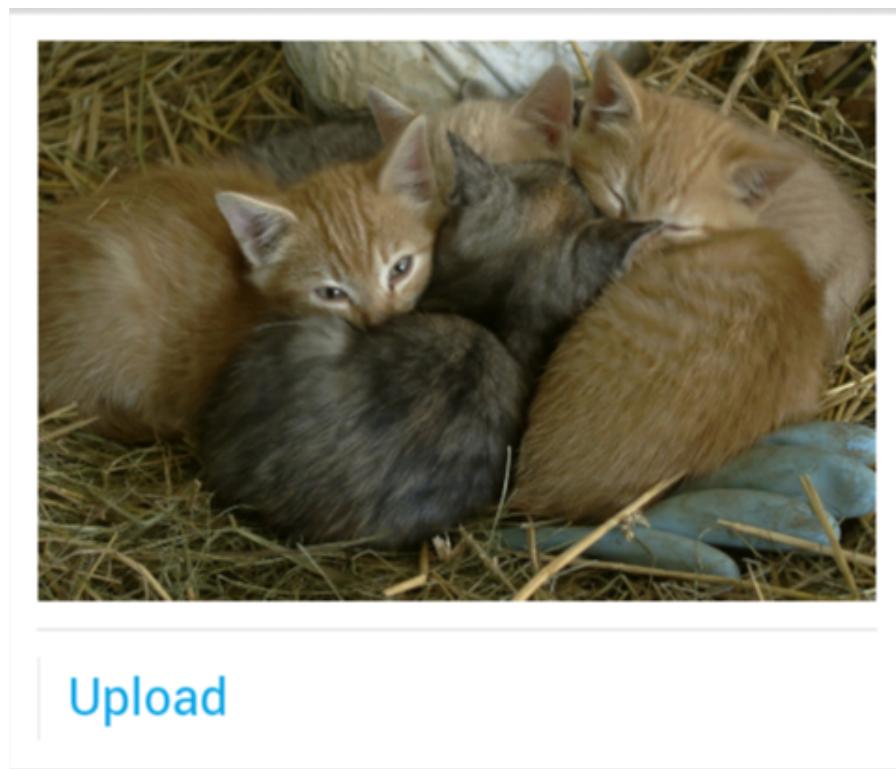
    public void uploadSucceeded(SucceededEvent event) {
        // Show the uploaded file in the image viewer
        image.setSource(new StreamResource(new StreamSource() {
            @Override
            public InputStream getStream() {
                byte[] bytes = fos.toByteArray();
                return new ByteArrayInputStream(bytes);
            }
        }, filename));
        image.setVisible(true);
    }
};

Upload upload = new Upload();
ImageUploader uploader = new ImageUploader(upload);
upload.setReceiver(uploader);
upload.addSucceededListener(uploader);
upload.setImmediate(true); // Only button

// Wrap it in a button group to give better style
HorizontalButtonGroup group = new HorizontalButtonGroup();
group.addComponent(upload);
layout.addComponent(group);
```

运行结果见图 20.15 “移动设备中的内容上传” (©2001 Marko Grönroos).

图 20.15. 移动设备中的内容上传



在客户端代码中实现内容上传

开发一个处理文件上传的 Widget 时，比如用于离线模式的文件上传组件，你可以使用 GWT 的 **FileUpload** 组件。这种情况下，你需要通过 RPC 调用将图片数据传递给服务器。

在移动设备中，最常见的上传任务可能就是使用设备上集成的摄像头拍摄一张照片。为了在客户端 UI 中正确地显示照片，你会希望确保照片尺寸和方向的准确性，而不需要发起一次服务器通信来做这个纠正处理。将照片发送到服务器时，你会希望避免使用太多的带宽。`lib-gwt-imageupload` add-on(可从 Vaadin Directory 得到) 中的 **ImageUpload** Widget，可以启动移动设备中的摄像头应用程序，并拍摄一张照片。如果需要，它还可以使用变换处理来定义一个图像处理管道，以便减小图片尺寸，还可以将图片方向修正为与 EXIF 数据一致，等等。修正后的图片会装载到内存缓冲区中，你可以使用其他 Widget 来显示它，发送到服务器，或者保存到本地存储中。

下例中，我们使用摄像头拍摄一张照片，修正图片并缩减它的尺寸，然后再次缩减尺寸以便在缩略图中显示它。注意，图像数据会被编码为一个 URL，因此编码后的数据可以使用在很多场合，比如 CSS 中。

```
final ImageUpload fileUpload = new ImageUpload();

// Have a separate button to initiate the upload
final VButton takePhotoButton = new VButton();
takePhotoButton.addClickHandler(new ClickHandler() {
    @Override
    public void onClick(ClickEvent event) {
        fileUpload.click();
    }
});
```

```

// Capture images from the camera, instead of allowing to
// choose from gallery or other sources.
fileUpload.setCapture(true);

// Normalize the orientation and make size suitable for
// sending to server
EXIFOrientationNormalizer normalizer =
    new EXIFOrientationNormalizer();
normalizer.setMaxWidth(1024);
normalizer.setMaxHeight(1024);
fileUpload.addImageManipulator(normalizer);
fileUpload.addImageLoadedHandler(new ImageLoadedHandler() {
    @Override
    public void onImageLoaded(ImageLoadedEvent event) {
        // Store the image data as encoded URL
        setImage(event.getImageData().getDataURL());
    }
});

// Reduce the size further for displaying a thumbnail
ImageTransformer thumbGenerator = new ImageTransformer();
thumbGenerator.setImageDataSource(fileUpload);
thumbGenerator.setMaxWidth(75);
thumbGenerator.setMaxHeight(75);
thumbGenerator.addImageLoadedHandler(new ImageLoadedHandler() {
    @Override
    public void onImageLoaded(ImageLoadedEvent event) {
        // Store the thumbnail image data as encoded URL
        setThumbnail(event.getImageData().getDataURL());
    }
});

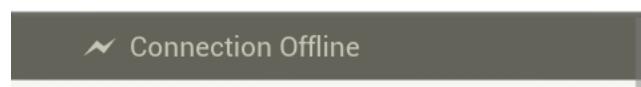
```

关于这个 add-on 的使用方法, 更多细节请参见 Parking 示例程序.

20.9. 离线模式(Offline Mode)

Vaadin TouchKit 应用程序通常都是服务器端应用程序, 但 TouchKit 也允许一种特殊的 离线模式, 这种模式是一个客户端 Vaadin UI, 当网络连接断开时就会自动切换到它. 离线 UI 包含在通常的服务器端 UI 的 Widget Set 之内, 并被保存在浏览器缓存之内. 只要提供一个特殊的缓存配置, 浏览器就会使用很高级别来缓存这个页面, 甚至浏览器重启之后页面也还可以继续存在, 因此 TouchKit 应用程序可以成为一种离线应用程序.

图 20.16. Parking 示例程序中的离线模式



在离线操作过程中, 离线 UI 可以将数据存储在移动设备浏览器的 HTML5 本地存储之内, 然后在网络连接恢复后再传递给服务器端应用程序.

通过 Maven archetype 创建的工程(详情请参见 第 20.5.1 节 “使用 Maven Archetype”), 它的框架代码会激活离线模式, 还包含了离线数据存储以及向服务器端发起 RPC 调用的类框架代码.

关于离线模式的完整示例, 请参见 Parking 示例程序, 以及它的源代码.

20.9.1. 启用缓存配置(Cache Manifest)

HTML5 支持 缓存配置(*cache manifest*), 因此可以利用这个功能来实现离线 Web 应用程序. 这个配置控制如何缓存各种资源. 配置会由 TouchKit 产生, 但你需要在 TouchKit 设定中启用它. 为了实现这一点, 你需要定义一个自定义 Servlet, 详情请参见 第 20.6.1 节 “Servlet 类”, 并对缓存设定调用 `setCacheManifestEnabled(true)`, 如下:

```
TouchKitSettings s = getTouchKitSettings();
...
s.getApplicationCacheSettings()
.setCacheManifestEnabled(true);
```

你还需要在 `web.xml` 部署描述文件中为 `manifest` 定义一个 MIME 类型, 如下:

```
<mime-mapping>
    <extension>manifest</extension>
    <mime-type>text/cache-manifest</mime-type>
</mime-mapping>
```

20.9.2. 启用离线模式

要启用离线模式, 你需要向 UI 添加 **OfflineMode** 扩展.

```
OfflineMode offlineMode = new OfflineMode();
offlineMode.extend(this);

// Maintain the session when the browser app closes
offlineMode.setPersistentSessionCookie(true);

// Define the timeout in secs to wait when a server
// request is sent before falling back to offline mode
offlineMode.setOfflineModeTimeout(15);
```

由 Maven archetype 创建的工程(详情请参见 第 20.5.1 节 “使用 Maven Archetype”), 它的框架代码中已经包含了以上代码.

你可以继承 **OfflineMode** 扩展, 以便以适当的方式, 从离线 UI 向服务器端传输数据, 详情请参见 第 20.9.4 节 “向服务器发送数据”.

20.9.3. 离线模式下的 UI

离线模式的构建方法与其他客户端模块一样, 详情请参见 第 13 章 客户端 Vaadin 开发. 在离线 UI 中你可以使用任意的 GWT Widget, Vaadin Widget, add-on Widget, 以及 TouchKit 的 Widget.

最常见的情况, 客户端应用程序会创建一个简单的 UI, 用于浏览和输入数据. 它会将数据存储在 HTML5 本地存储中. 然后它会监视服务器连接是否恢复, 如果恢复了, 就将用户输入的数据发送到服务器, 并提示用户回到连线模式.

`Parking` 示例程序提供了离线模式 UI 的一个示例实现. `Ticket` 视图实现为一个胖客户端 Widget, 而服务器端视图只负责将状态数据发送给 Widget 端.

20.9.4. 向服务器发送数据

一旦网络连接恢复, 离线模式 UI 就可以将用户输入的数据发送到服务器端. 比如, 你可以向一个服务器端的 UI 扩展发起服务器 RPC 调用, 来实现从离线 UI 向服务器的数据发送, 详情请参见 第 16.6 节 “客户端与服务器端之间的 RPC 调用”.

20.9.5. 离线模式下的 Theme

通常，客户端模块有自己的样式表，保存在 public 文件夹中，并被编译到客户端目标模块中，详情请参见第 16.8 节“Widget 的样式控制”以及第 13.3.1 节“指定样式表”。但是，你也有可能会希望让离线模式使用与连线模式相同的样式。要使用与服务器端应用程序相同的 Theme，你需要在 Widget Set 定义文件中定义 Theme 路径，如下。

```
<set-configuration-property
    name='touchkit.manifestlinker.additionalCacheRoot'
    value='src/main/webapp/VAADIN/themes/mytheme:../../..../VAADIN/themes/mytheme
/>
```

在你的离线应用程序中，你需要遵循 Vaadin Theme 所要求的 CSS 样式结构。如果使用了任何的 Vaadin Widget，详情请参见第 15.3 节“Vaadin Widget”，它们都会使用 Vaadin Theme。

20.10. 构建最优化的 Widget Set

移动设备的网络通常会比 DSL Internet 连接要慢。当启动一个 Vaadin 应用程序时，Widget Set 是浏览器需要装载的资源中最大的。对于大多数应用程序来说，尤其是对于移动设备上的应用程序，大多数 Vaadin 组件其实不会被用到，因此创建优化版的 Widget Set 是很有益处的。

对于单独的 Widget，Vaadin 可以在需要时才延迟加载它。在 TouchKit 应用程序中使用的 **TouchKitWidgetSet** 对 Widget Set 进行了优化，最初只会下载最必要的 Widget，然后再延迟装载其他 Widget。对于大多数 TouchKit 应用程序，这是一种很好的这种方案。但是，由于大多数移动设备网络的速度很慢，将 Widget Set 分为小片来装载可能不是所有情况下的最佳方案。通过一些自定义优化，你可以去除所有不必要的 Widget，创建一个整体的 Widget Set。再配合适当的 GZip 压缩，对于移动设备浏览器来说，它将会非常轻量。

但是，如果应用程序中存在很大的组件，这些组件很少使用，或者并不存在于初期视图中，那么最好延迟装载这些组件的 Widget。

关于 Widget Set 优化，你可以在 Parking 源代码的 `ParkingWidgetset.gwt.xml` 和 `WidgetLoaderFactory.java` 文件中找到可以实际运行的示例。

20.10.1. 生成 Widget Map

你可以使用自定义的 **WidgetMapGenerator** 实现来对 Widget Set 进行调优。Generator 类应该继承 **TouchKitBundleLoaderFactory**，并重载 `getConnectorsForWidgetset()` 方法。这个方法返回 Widget Set 中使用的 Widget 的连接器类。

如果你知道你的应用程序中使用了哪些组件，那么可以手动构建使用的连接器（以及对应的 Widget）列表。你还可以，比如，使用调试器来研究一下 Vaadin 的 **CommunicationManager** 类，它会打开运行中的应用程序内的所有视图。这个类中会包含一个集合，其内容是到目前为止所用到的所有组件。

在 Parking 示例程序中，我们首先在构造函数中构建连接器类名的列表，如下：

```
public class WidgetLoaderFactory
    extends TouchKitBundleLoaderFactory {
    private final ArrayList<String> usedConnectors;

    public WidgetLoaderFactory() {
        usedConnectors = new ArrayList<String>();
        usedConnectors.add(ButtonConnector.class.getName());
        usedConnectors.add(ChartConnector.class.getName());
```

```
usedConnectors.add(CssLayoutConnector.class.getName());
...
```

然后我们在 `getConnectorsForWidgetset()` 方法中, 使用这个列表从所有类型定义中过滤出需要的类, 创建适当的类型定义列表. **JClassType** 用来表达类型定义.

```
@Override
protected Collection<JClassType> getConnectorsForWidgetset(
    TreeLogger logger, TypeOracle typeOracle)
    throws UnableToCompleteException {
    // The usedConnectors list should contain all the
    // connectors that we need in the app, so we
    // can leave all others away.

    // Get all connectors in the unoptimized widget set
    Collection<JClassType> connectorsForWidgetset = super
        .getConnectorsForWidgetset(logger, typeOracle);

    // Filter the connectors using the used list
    ArrayList<JClassType> arrayList =
        new ArrayList<JClassType>();
    for (JClassType jClassType : connectorsForWidgetset) {
        String qualifiedSourceName =
            jClassType.getQualifiedSourceName();
        if (usedConnectors.contains(qualifiedSourceName)) {
            arrayList.add(jClassType);
        }
    }
    return arrayList;
}
```

20.10.2. 定义 Widget 的装载方式

`getLoadStyle()` 方法返回 Widget 的装载方式, 为了实现整体的 Widget Set, 返回值应该是 **EAGER**.

```
@Override
protected LoadStyle getLoadStyle(JClassType connectorType) {
    return LoadStyle.EAGER;
}
```

20.10.3. 应用自定义的 Widget Map Generator

需要在 `.gwt.xml` Widget Set 定义文件中定义, 如下:

```
<generate-with class="com.myprj.WidgetLoaderFactory">
    <when-type-assignable
    class="com.vaadin.client.metadata.ConnectorBundleLoader" />
</generate-with>
```

20.10.4. 部署

注意, 如果你希望优化应用程序的启动时间, 并将数据传输量降到最低, 你需要为你的应用程序启用 GZip 压缩功能. 这个问题的最佳实现方法与你的服务器主机设置高度相关, 因此本书不讨论这个问题.

20.11. 在移动设备上测试和调试

对于移动设备来说，测试时一个特殊的挑战。移动设备浏览器可能并不带有很多调试功能，你可能无法安装第三方调试插件，比如 Chrome Developer Tools。

20.11.1. 调试

Debug 窗口(详情请参见 第 11.3 节 “Debug 模式和 Debug 窗口”), 在移动设备浏览器中也可以工作，虽然稍微有些难用。

虽然缺少浏览器内的分析工具，但可以通过简单的呵护端代码来补救。比如，你可以使用 HTML DOM 中的 `innerHTML` 属性来保存页面的 HTML 内容。为了实现这个目的，你需要从服务端执行一个 JavaScript 调用，并使用一个回调方法来处理它的应答，详情请参见 第 11.14.2 节 “处理 JavaScript 函数的回调”。

桌面调试

TouchKit 优先支持基于 WebKit 的浏览器，iOS 和 Android 设备使用的都是这类浏览器。因此你可以使用 WebKit 的桌面浏览器实现与移动设备浏览大致的兼容，比如使用 Google Chrome。桌面浏览器也支持地理位置信息之类的功能。如果你的应用程序需要自动适应手机/Tablet 环境的不同屏幕尺寸，以及不同的屏幕方向，你可以简单地拖动浏览器大小来模拟各种屏幕模式。此外，浏览器还带有特殊的开发设定，来模拟触摸设备上的某些功能。

远程调试

Safari 和 Chrome 都支持远程调试，这个功能允许你从桌面浏览器来远程调试移动设备浏览器。

Vaadin TestBench

21.1. 概述	563
21.2. 快速入门	567
21.3. 安装 Vaadin TestBench	570
21.4. 开发 JUnit 测试程序	575
21.5. 创建 Test Case	579
21.6. 查询页面元素	582
21.7. 元素选择器	584
21.8. 测试中的一些特殊问题	585
21.9. 创建可维护的测试程序	589
21.10. 屏幕截图的取得和比较	592
21.11. 运行测试	596
21.12. 在分布式环境中运行测试	598
21.13. 测试程序的并行执行	603
21.14. 无头(Headless)测试	604
21.15. 行为驱动开发	605
21.16. 已知的问题	606

本章介绍 Vaadin TestBench 的安装和使用。

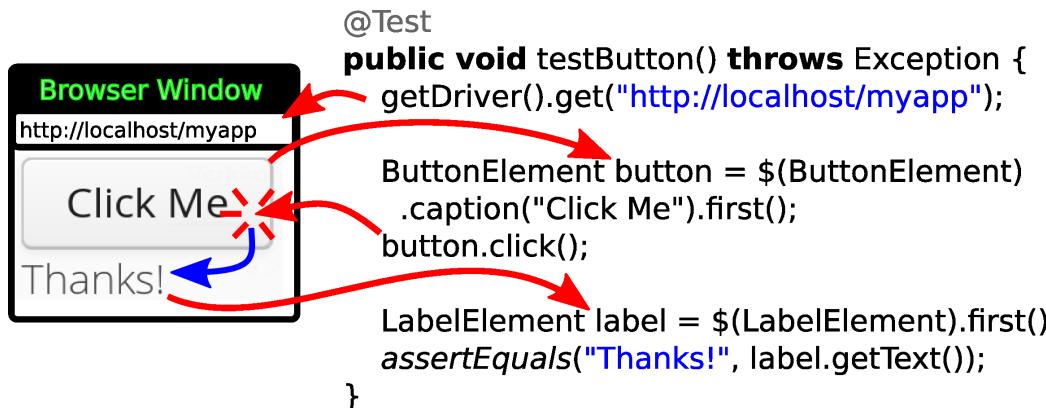
21.1. 概述

测试是现代软件开发的基石之一。测试贯穿软件开发的全过程，它是联系最终产品与客户需求的纽带。在敏捷软件开发，以及其他迭代式开发模式中，随着发布周期变短，以及持续集成的引入，集成的自动化，回归测试，压力测试，验收测试都变得非常重要。此外，为了集成测试的目的，以及针对残

障人士的辅助技术，还需要实现 UI 的自动化。由于 Web 应用程序存在的特殊性质，也就对测试和 UI 自动化都提出了一些特殊要求。

Vaadin TestBench 可以通过 Java 代码控制浏览器，见图 21.1 “使用 Testbench 控制浏览器”。它可以打开一个新的浏览器窗口，启动应用程序，与 UI 组件进行交互，比如点击组件，然后取得 HTML 元素的值。

图 21.1. 使用 **Testbench** 控制浏览器



在开始讨论更多的功能细节之前，你可能希望亲自试用一下 Vaadin TestBench。只需要使用 Eclipse plugin 或 Maven archetype 创建一个新的 Vaadin 工程就可以了。这两种方法都会创建一个简单的应用程序框架代码，其中包含 TestBench 的 test case，可以用来测试 UI。你还需要安装一个试用许可协议。具体方法请先阅读第 21.2 节“快速入门”，按照其中的指南动手试验一下，然后再回到本节。

Vaadin TestBench 在软件开发中的地位

Vaadin TestBench 可以成为整个软件开发过程的中心角色，它可以在开发周期的每个阶段，在所有的模块层次上，对应用程序进行测试：

- 自动化的验收测试
- 单元测试
- 端到端(End-to-end)的集成测试
- 回归测试

下面我们分别介绍以上各个问题。

不管是敏捷软件开发，还是其他的软件开发模式，首先都需要明确需求规范，需求规范决定软件应该做什么。验收测试(Acceptance test)确保产品符合需求规范。在敏捷软件开发模式中，自动化的验收测试可以用来持续地追踪最终目标的完成进度，还可以用来检测是否存在回归故障。在测试驱动开发(test-driven development，简称 TDD)中，非常强调需求规范的重要性，在这种开发模式下，还未编写真实代码之前，首先就应该编写测试程序。在第 21.15 节“行为驱动开发”中，我们介绍如何使用 Vaadin TestBench 来进行 行为驱动开发 (behaviour-driven development，简称 BDD)，它是 TDD 的一种形式，更集中关注需求规范中正式化的行为规范。

单元测试(Unit testing)用于测试软件组件中最小的功能单元；在 Vaadin 应用程序中，通常是独立的 UI 组件，或者视图类。你也可能希望为应用程序生成各种不同的输入，然后检查对应的输出是否符合预期。对于复杂的复合组件，比如视图，你可以使用页面对象模式(Page Object Pattern)，详情

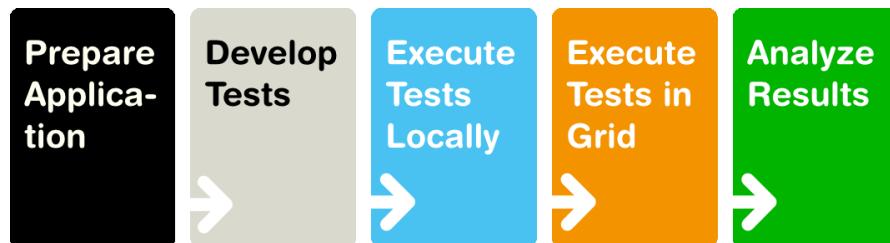
请参见第 21.9.2 节“页面对象模式(Page Object Pattern)”。这种模式将具体的底层实现细节从抽象的 UI 逻辑中分离出来，这样就可以使测试更加简化，更加模块化。除了用于单元测试之外，它还创造了一个抽象层，可以用于高级别的测试，比如验收测试和端到端测试。

集成测试 (Integration test) 负责确保软件的多个功能模块单元能够在不同的模块化层次上联合运转。在最大的层次上，端对端测试 (end-to-end test) 贯穿应用程序的整个生命周期，从开始到结束，覆盖很多(甚至全部)用户场景(user stories)。测试的目的不仅仅是验证是否达到了用户交互方面的功能需求，而且还要验证是否保证了数据的完整性。比如，在聊天应用程序中，一个用户应该先登录系统，然后发送和接收消息，最后退出系统。这样的测试流程可能包括配置，注册，用户间交互，管理性任务，删除用户帐号，等等。

在 回归测试 (regression testing) 中，你希望能够确保，在修改了代码之后，软件的行为只发生我们期待的变化。有两条防线可以阻止这类回归障碍。回归测试的主要源代码包括验收测试，单元测试，以及集成测试，负责验证 UI 的 HTML 表现中显示的值逻辑上是正确的。但是，这些测试还不足以检测出程序外观表现的回归故障，比如，无效的 UI 描绘，或 theme 问题都会导致外观的不正确。通过将屏幕截图与参考图片进行比较，这种方式实现了一种更合理的故障检测能力，但代价是当布局和 theme 变化时，会失去测试程序的健壮性。可以将这种代价降到最低，方法是将屏幕截图的比较范围局限到最关键的区域，以及使得对回归障碍的分析尽量简单。在第 21.10 节“屏幕截图的取得和比较”中我们将介绍，Vaadin TestBench 可以自动地将屏幕截图与正确图片之间的差异部分高亮度显示，还可以在图片比较时遮蔽掉无关区域。

你可以针对你的应用程序开发这样的测试程序，比如可以使用 JUnit，它是一种广泛使用的 Java 单元测试框架。你可以在你的工作站中，或者在一个分布式网格环境中，按照你的需求任意次地运行这些测试程序。

图 21.2. TestBench 的工作流程



功能

Vaadin TestBench 的主要功能包括：

- 通过 Java 代码控制浏览器
- 在调试窗口中生成组件选择器
- 通过断言(assertion)和比较屏幕截图来检验 UI 状态
- 比较屏幕截图，并将截图不同的部分高亮度显示
- 可用来运行测试的分布式测试网格
- 与单元测试集成
- 使用移动设备浏览器进行测试

测试的执行可以分散到多个测试节点组成的网格(grid)上, 这样可以加快测试速度。网格节点可以混行不同的操作系统, 安装不同的浏览器。在最小安装环境中, 比如开发测试程序的环境, 你可以在单台计算机上使用 Vaadin TestBench。

基于 Selenium

Vaadin TestBench 基于 Selenium Web 浏览器自动化库, 尤其是其中的 Selenium WebDriver, WebDriver 可以通过 Java 代码直接控制浏览器。

Vaadin TestBench 也通过 Vaadin 独有的扩展对 Selenium 进行了扩充, 比如:

- 正确地处理 Vaadin 的基于 AJAX 的通信
- 在高级编程模型中, 有一套静态类型的页面元素查询 API, 可用于选择 Vaadin 组件
- Vaadin 应用程序的性能测试
- 屏幕截图的比较
- 使用 Vaadin 选择器查找 HTML 元素

TestBench 的组件

TestBench 库包含 WebDriver, 它提供了 API, 可以象使用者一样控制浏览器。这些 API 可以用来开发测试程序, 比如, 使用 JUnit 开发单元测试。它还包括网格 hub 以及节点服务器, 你可以使用它们来在网格环境中运行测试。

Vaadin TestBench 库还提供了重要的控制逻辑, 可以用来:

- 使用 WebDriver 来运行测试
- 针对测试 Vaadin 应用程序的额外支持
- 将屏幕截图与参考图片进行比较
- 使用网格节点和 hub 服务进行分布式测试

需求

开发和运行测试程序, 需求如下:

- Java JDK 1.6 或更高版本
- 在测试节点上安装了 Selenium WebDriver 所支持的浏览器
 - Google Chrome
 - Internet Explorer
- Mozilla Firefox (推荐使用 ESR 版本)
- Opera
- 移动设备浏览器: Android, iPhone
- 一个构建系统, 比如 Ant 或 Maven, 以便在构建过程中自动化执行测试(推荐)

注意, 推荐在长期支持版(ESR, Extended Support Release)的 Firefox 上运行测试, 因为 Firefox 的发布非常频繁, 经常会导致测试失败. 可以在 <http://www.mozilla.org/en-US/firefox/organizations/all.html> 下载 ESR 版本的 Firefox. 请将 ESR 版与通常版 Firefox 同时安装(不要覆盖).

对于 Mac OS X, 请注意这里提到的问题: 第 21.16.1 节 “在 Mac OS X 上运行 Firefox 测试程序”.

与持续集成(Continuous Integration)的兼容

持续集成意味着对应用程序频繁地自动编译并自动测试, 通常至少一天一次, 理想状况下, 应该在源代码变更被提交到代码库时立即执行. 这种方式可以更早地捕捉到集成问题, 并找出最初导致这些问题的代码变更.

你可以使用 Vaadin TestBench 来开发单元测试程序, 就像你使用任何其他的 Java 单元测试一样, 因此这些单元测试程序可以与持续集成系统平滑地结合在一起. Vaadin TestBench 已被测试过, 至少可以与 TeamCity 以及 Hudson/Jenkins 构建管理和持续集成服务器共同工作, 这些工具都对 JUnit 单元测试框架有特殊的 support.

图 21.3. 持续集成(Continuous Integration)的工作流程

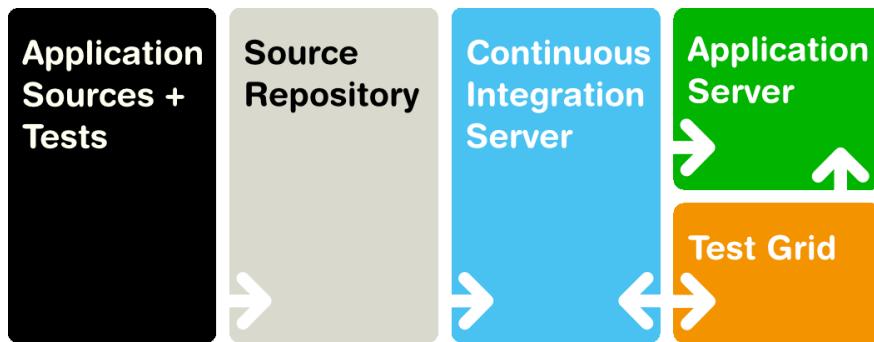


图 21.3 “持续集成(Continuous Integration)的工作流程” 描述了一种典型的开发配置. 应用程序和测试程序的源代码变更都会被 check in 到源代码仓库中, CIS 服务器再从源代码仓库 check out 这些源代码, 编译, 并将 Web 应用程序发布到服务器上. 然后, 它会运行测试程序, 并收集测试结果.

许可协议以及试用期限

你可以从 Vaadin Directory 下载 Vaadin TestBench, 并免费试用 30 天, 试用期结束后你需要购买许可. 你可以通过 Vaadin Directory 购买许可. 在 Vaadin Pro 订阅帐号内也包含了 Vaadin TestBench 的许可.

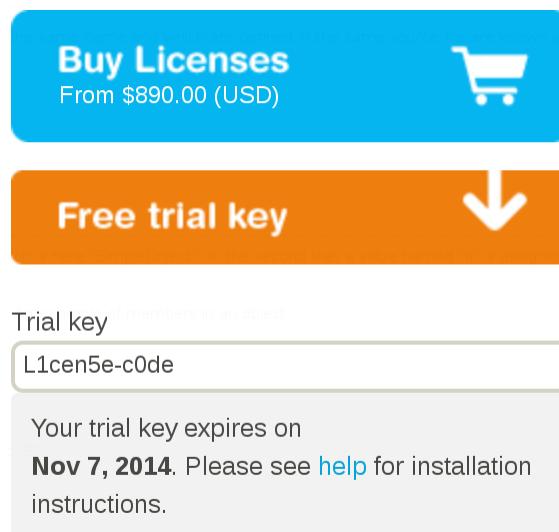
21.2. 快速入门

下面我们来介绍如何在几分钟之内让 Vaadin TestBench 运行起来. 你可以创建一个新的 Eclipse 工程, 或一个 Maven 工程. 这两种工程类型都需要安装 license key, 所以我们首先来介绍 license key 的安装.

21.2.1. 安装 License Key

你需要安装 license key 才能够运行测试程序. 你可以购买 Vaadin TestBench, 也可以从 Vaadin Directory 的 Vaadin TestBench 下载页面 得到一个免费试用的 key. 你需要在 Vaadin Directory 中注册才能得到 key.

图 21.4. 通过 **Vaadin Directory** 得到 License Key



要将 license key 安装到开发工作站上, 你可以将 license key 的文字内容复制并粘贴到你的 home 目录下的 `.vaadin.testbench.developer.license` 文件中. 比如, 在 Linux 和 OS X 环境中:

```
$ echo "L1cen5e-c0de" > ~/.vaadin.testbench.developer.license
```

你也可以将 license key 通过系统属性传递给运行测试的 Java 应用程序, 通常可以在命令行中使用 `-D` 选项:

```
$ java -Dvaadin.testbench.developer.license=L1cen5e-c0de ...
```

如何将参数传递给你的测试程序的运行器, 取决于实际的测试执行环境. 以下列举了一些典型的环境:

Eclipse IDE

要为所有的工程安装 license key, 请选择菜单项 **Window → Preferences**, 然后找到 **Java → Installed JREs** 部分. 选择你的应用程序所使用的 JRE 版本, 然后点击 **Edit**. 在 **Default VM arguments** 项目中, 指定上面介绍的 `-D` 表达式.

对于单个工程, 可以创建一个新的 JUnit 启动配置, 方法是选择菜单 **Run → Run configurations**. 选择 **JUnit**, 然后点击 **New launch configuration**. 如果你已经在这个工程内运行过 JUnit, 那么启动配置应该已经存在. 如果没有自动选中 JUnit 4 的话, 请选择它. 进入 **Arguments** tab 页, 并在 **VM arguments** 项目中指定 `-D` 表达式. 点击 **Run** 可以立即运行测试程序, 也可以点击 **Close**, 只保存相关设定.

Apache Ant

如果使用 Apache Ant 中的 `<junit>` task 来运行测试, (详情请参见 第 21.11.1 节“使用 Ant 运行测试”), 你可以使用以下方法来传递 license key:

```
<sysproperty key="vaadin.testbench.developer.license"
             value="L1cen5e-c0de"/>
```

但是, 你不应该将 license key 保存到源代码仓库中, 因此, 如果 Ant 脚本被保存在源代码仓库中, 那么你应该以属性的形式将 license key 传递给 Ant, 然后在 Ant 脚本中 `<sysproperty>` 的 `value` 参数中使用这个属性, 如下:

```
<sysproperty key="vaadin.testbench.developer.license"
    value="${vaadin.testbench.developer.license}" />
```

从命令行启动 Ant 时, 你可以使用 `-D` 参数来向 Ant 传递属性.

Apache Maven

如果使用 Apache Maven 来运行测试, 你可以使用 `-D` 参数来向 Maven 传递 license key:

```
$ mvn -Dvaadin.testbench.developer.license=L1cen5e-c0de verify
```

TeamCity

在 TeamCity 中, 你可以在构建配置中, 以系统属性的方式将 license key 传递给构建运行器. 但是, 这种方式只将它传递给运行器. 上面我们介绍过, Maven 还会将参数再传递给 JUnit, 但 Ant 不会默认地这样做, 因此你需要象前文中介绍过那样, 明确地转发这个参数.

更多详细信息请参见 AGPL license key 安装指南.

21.2.2. Eclipse 环境下的快速入门

安装 Vaadin Plugin for Eclipse 之后, 你可以使用它来创建新的 Vaadin 7 工程, 并激活 TestBench 测试功能, 详情请参见 第 2.5.1 节 “创建工程”. 在工程设置中, 你需要让 **Create TestBench test** 选项激活.

测试用例的框架代码创建在源代码文件夹 `test` 之下, 因此这些测试代码不会随应用程序一起发布. 工程和源代码文件夹的构成请参见 21.5.

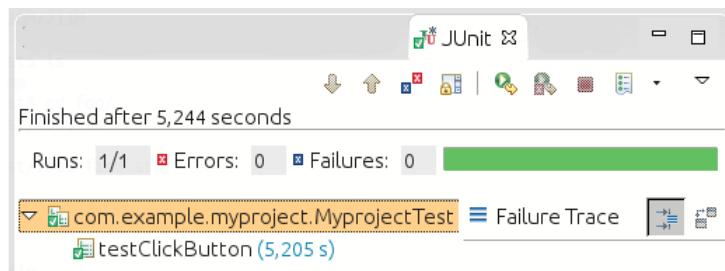
图 21.5. 带 Test Case 的 Eclipse 工程



你可以看到, UI 及测试用例与 图 21.1 “使用 Testbench 控制浏览器” 非常类似. 生成的测试用例框架代码的细节, 请参见 第 21.4.1 节 “测试用例(Test Case) 的基本结构”.

要运行测试, 请在编辑器中打开 `MyprojectTest.java` 文件, 然后按下快捷键 **Shift+Alt+X-T**. 浏览器将打开应用程序 UI, 然后 TestBench 将会运行测试. 测试的运行结果将显示在 Eclipse 的 **JUnit** 视图中, 见 图 21.6 “Eclipse 中的 JUnit 测试结果”.

图 21.6. Eclipse 中的 JUnit 测试结果



21.2.3. 使用 Maven 的快速入门

使用 Maven, 你需要使用 `vaadin-archetype-application archetype` 来创建新的 Vaadin 工程, 详情请参见 第 2.6 节 “通过 Maven 使用 Vaadin”.

工程中的 `src` 文件夹包含了应用程序源代码, 以及测试程序的源代码. 测试用例的框架代码在 `src/test` 文件夹下, 详情请参见 第 21.4.1 节 “测试用例(Test Case)的基本结构”.

需要安装 `license`, 或者对以下命令, 以参数的方式指定 `license`, 方法见上文的介绍. 要构建工程, 可以使用 `integration-test`, 或者使用构建周期(build lifecycle)中更晚一些的阶段(phase). 比如, 可以通过命令行:

```
$ mvn integration-test
```

这个命令将会执行所有需要的所有其他阶段, 包括编译, 打包应用程序, 启动 Jetty Web 服务器来运行应用程序, 以及运行 TestBench 测试程序. 测试结果报告将输出到控制台. Maven GUI, 比如 Eclipse 中的 Maven 插件, 会以更加可视化的方式显示测试结果.

21.3. 安装 Vaadin TestBench

和大多数 Vaadin add-on 一样, 你可以在你的项目中以 Maven 或 Ivy 依赖项目的形式安装 Vaadin TestBench, 或者使用安装包来安装. 安装包包含一些额外的内容, 比如文档, 以及独立的库文件, 你可以在网格环境的测试程序中使用这些库文件.

组件元素类是 Vaadin 独有的, 这些类打包在名为 `vaadin-testbench-api` 的 JAR 库文件中, 与执行测试时需要的运行库 `vaadin-testbench-core` 分离.

此外, 你可能需要为你使用的浏览器安装驱动程序.

21.3.1. 测试程序开发环境

在通常的测试程序开发环境中, 你会在 Java 工程内开发测试程序, 并在开发工作站上运行这些测试. 你可以在一个专用的测试服务器上运行同样的测试程序, 比如在持续集成系统内.

在测试程序开发环境中, 你不需要网格 hub 和节点. 但是, 如果你正在为网格环境开发测试程序, 你可以将测试程序, 网格 hub, 以及一个测试节点, 全部运行在你的开发工作站上. 关于分布式测试环境, 将在后文中介绍.

Maven 依赖项目

Vaadin TestBench 的 Maven 依赖项目如下:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-testbench</artifactId>
    <version>4.x.x</version>
    <scope>test</scope>
</dependency>
```

如果没有定义过 Vaadin add-on 的仓库, 你还需要定义它:

```
<repository>
    <id>vaadin-addons</id>
    <url>http://maven.vaadin.com/vaadin-addons</url>
</repository>
```

vaadin-archetype-application archetype (参见 第 21.2.3 节 “使用 Maven 的快速入门”), 包括了这些声明.

Ivy 依赖项目

Ivy 依赖项目, 定义在 ivy.xml 文件中, 内容应该如下:

```
<dependency org="com.vaadin" name="vaadin-testbench-api"
    rev="latest.release" conf="nodeploy->default"/>
```

nodeploy->default 是一个可选的配置, 它需要 Ivy 模块中的 nodeploy 配置; 新建 Vaadin 工程时会自动创建这个配置.

使用 Vaadin Plugin for Eclipse 新建的 Vaadin 工程 (第 21.2.2 节 “Eclipse 环境下的快速入门”), 将会包含这些依赖项目.

代码组织方式

我们通常建议将测试程序放在独立的工程或模块中, 与被测试的 Web 应用程序分离开, 以避免库版本问题. 如果测试程序是被测项目的一部分, 你至少应该管理好源代码和依赖项目, 使得测试类, TestBench 库, 以及它们的依赖项目不会随 Web 应用程序一起发布出去.

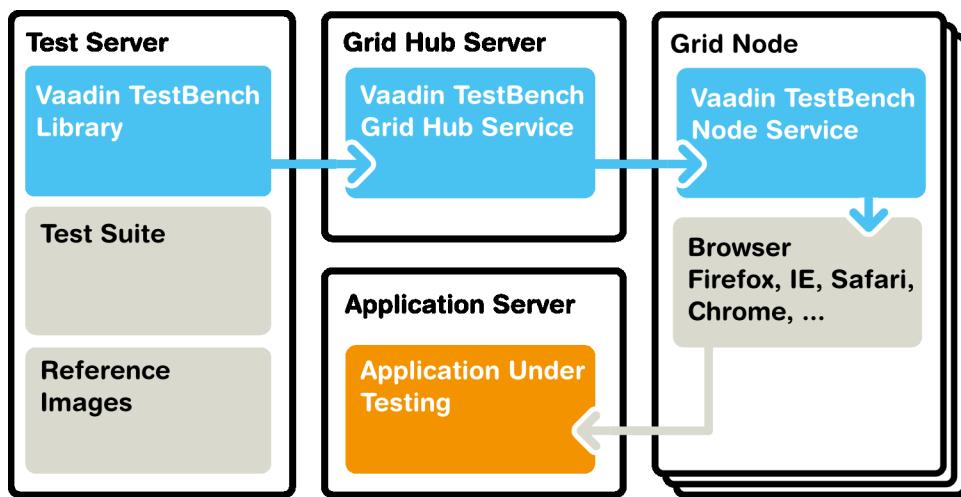
21.3.2. 分布式测试环境

Vaadin TestBench 支持在网格内分布式执行测试程序. 一个测试网格包括以下几种主机类别:

- 一个或多个测试服务器, 负责运行测试程序
- 一个网格 hub
- 网格节点

网格环境中的各种组件, 见 图 21.7 “Vaadin TestBench 网格环境”.

图 21.7. Vaadin TestBench 网格环境



网格 hub 是一个服务，它负责处理 JUnit 测试运行器与测试节点之间的通信。测试节点也是服务，它负责在浏览器内执行实际的测试命令。

hub 只需要极少的资源，因此通常可以将它运行在测试服务器上，或者运行在某一个测试节点上。你可以在同一台主机上同时运行测试程序，hub，以及一个测试节点，但在完整的分布式测试环境中，应该将 Vaadin TestBench 组件安装在不同的主机上。

通过分布式测试环境来控制浏览器，需要使用远程 WebDriver。关于测试网格的开发，以及 hub 和测试节点的使用，详情请参见 第 21.12 节“在分布式环境中运行测试”。

21.3.3. 安装包的内容

安装包中包含以下内容：



documentation 文件夹包含发布说明文件，本章的 PDF 摘录版，以及 license.

maven

Maven 文件夹包含 Vaadin TestBench 库的 JAR 文件（你也可以在非 Maven 工程中使用这些 JAR 文件）。这个文件夹还包含一个 POM 文件，因此你可以将它安装到你的本地 Maven 仓库。请参照 第 21.11.2 节“使用 Maven 运行测试”中的安装指南。

`vaadin-testbench-standalone-4.x.x.jar`

这是 Vaadin TestBench 库的 standalone 版本，主要用于运行网格 hub 及节点服务，详情请参见 第 21.12 节“在分布式环境中运行测试”。

21.3.4. TestBench 示例程序

TestBench 示例程序地址是: <https://github.com/vaadin/testbench-demo>。你可以在这个网站上查看源代码，也可以使用 Git 客户端将代码仓库 clone 到本地；在命令行中执行以下命令：

```
$ git clone https://github.com/vaadin/testbench-demo
```

在命令行中运行以下命令，可以执行测试程序：

```
$ mvn verify
```

被测应用程序是一个桌面计算器程序，源代码位于 `src/main/java` 文件夹下。

针对这个应用程序的 `TestBench` 测试程序位于 `src/test/java` 目录下，Java 包目录为 `com/vaadin/testbenchexample`。其中包含以下代码：

`SimpleCalculatorITCase.java`

演示了 WebDriver 的基本使用。与 UI 中的按钮交互，点击按钮，然后检查运行结果中的值。使用 ElementQuery API 来访问页面元素。

`LoopingCalculatorITCase.java`

另一个简单示例，演示了如何使用循环来产生程序化的循环，创建一个复杂的用例(use case)。

`ScreenshotITCase.java`

演示如何比较屏幕截图，详情请参见第 21.10.3 节“取得用于比较的屏幕截图”。有些测试用例包含了随机的输入，因此屏幕截图比较时需要排除随机数据区域。

这个示例使用了 `@Ignore` 注解，默认会被忽略，因为其中包含的图片是从一个特性平台的特定浏览器中截取的，因此如果你使用的是其他环境，测试将会失败。如果你启用这些测试程序，你需要运行它，将出错图片复制到参照屏幕截图的文件夹中，并使用 alpha 通道将错误区域遮蔽掉。关于如何启用这些示例测试程序，以及如何创建被遮蔽的参照图片，详情请参见 `example/Screenshot_Comparison_Tests.pdf`。

`SelectorExamplesITCase.java`

这个示例演示如何使用不同的方式查找元素；可以使用高级的 ElementQuery API，也可以使用低级的 `By.xpath()` 选择器。

`VerifyExecutionTimeITCase.java`

演示如何对一个测试用例计时，以及如何报告计时结果。

`AdvancedCommandsITCase.java`

演示如何测试上下文菜单（参见第 21.8.5 节“测试上下文菜单”）和提示信息(tooltip)（参见第 21.8.2 节“测试提示信息(Tooltip)”）。还演示了如何向组件发送键盘按键，以及如何读取表格中的值。

`pageobjectexample/PageObjectExampleITCase.java`

演示如何使用 页面对象模式(*Page Object Pattern*) 创建可维护的测试程序，这种模式将低层页面结构与业务逻辑代码分离开，详情请参见第 21.9 节“创建可维护的测试程序”。页面对象类处理与应用程序视图的低层交互，位于 `pageobjects` 包内。

`bdd/CalculatorSteps.java`, `bdd/SimpleCalculation.java`

演示如何使用 JBehave 框架，遵循 行为驱动开发 (behaviour-driven development，简称 BDD) 模式来开发测试程序。`SimpleCalculation.java` 定义了一个基于 JUnit 的用户场景(user story)，其中有一个情节(scenario)，在 `CalculatorSteps.java` 中定义。这个情节重用了页面对象示例(见上文)中定义的页面对象，以便实现应用程序视图的低级的访问和控制。这个示例将在 第 21.15 节“行为驱动开发”中介绍。

21.3.5. 安装浏览器驱动程序

无论是在工作站上使用 WebDriver 来开发测试程序，还是在网格内运行测试程序，要使用浏览器就需要安装浏览器驱动程序。

1. 下载最新的浏览器驱动程序

- Internet Explorer (只能用于 Windows) - 请安装 Selenium 最新发布版中的 IEDriverServer.exe, 地址是:

<http://selenium-release.storage.googleapis.com/index.html>

- Chrome - 请安装适合你的操作系统的 ChromeDriver 的最新版本(它是 Chromium 工程的一部分), 地址是:

<http://chromedriver.storage.googleapis.com/index.html>

2. 将驱动程序可执行文件的路径添加到 PATH 中. 在分布式测试环境中, 将其指定为网格节点服务的命令行参数, 详情请参见 第 21.12.4 节 “启动一个测试网格节点”.

为 Ubuntu Chromium 安装 ChromeDriver

虽然你可以在 Ubuntu 上安装 Google Chrome, 但它还带有自己的 Chromium 浏览器, 这个浏览器的代码是基于 Google Chrome 的. Chromium 也有它独自的 ChromeDriver, 使用这个驱动还需要一些额外的安装步骤.

安装 ChromeDriver:

```
$ sudo apt-get install chromium-chromedriver
```

将驱动程序的可执行文件添加到 path 中, 比如:

```
$ sudo ln -s /usr/lib/chromium-browser/chromedriver /usr/local/bin/chromedriver
```

Chromium 的库文件需要包含到系统的库文件查找路径中:

```
$ sudo sh -c 'echo "/usr/lib/chromium-browser/libs" > /etc/ld.so.conf.d/chrome_libs.conf'
```

```
$ sudo ldconfig
```

21.3.6. 测试节点的配置

如果你在网格环境中运行测试, 你需要对测试节点进行一些配置, 才能得到更稳定的测试结果.

更多的配置可以通过节点服务启动时的命令行参数指定, 详情请参见 第 21.12.4 节 “启动一个测试网格节点”.

操作系统设置

需要对操作系统进行一些设置, 这些设置可能影响到浏览器行为, 以及浏览器如何打开或关闭. 常见的问题包括浏览器崩溃时的错误对话框.

在 Windows 中, 需要禁用浏览器崩溃时的错误报告功能, 如下:

1. 打开 控制面板 → 系统
2. 选择 高级 Tab 页
3. 选择 错误报告
4. 选中 禁用错误报告
5. 去掉 但在发生严重错误时通知我

屏幕截图设置

屏幕截图比较功能要求浏览器的用户界面固定不变。影响测试效果的具体功能，取决于使用的操作系统和浏览器。

通常需要如下设定

- 禁用闪烁光标
- 在所有的测试主机上使用完全相同的操作系统 theme
- 关闭可能忽然弹出新窗口的所有软件
- 关闭屏幕保护程序

如果使用 Windows 和 Internet Explorer 浏览器，你还需要进行以下设定：

- 关闭 安全 中的 允许活动内容在我的计算机上运行 选项

21.4. 开发 JUnit 测试程序

JUnit 是一个非常流行的 Java 单元测试框架。大多数 Java IDE, 构建系统, 以及持续集成系统都提供对 JUnit 的支持。虽然我们在这一章中集中介绍 JUnit 测试程序的开发, 但 Vaadin TestBench 和 WebDriver 并不特别要求使用 JUnit, 你可以使用任何其他测试执行框架, 或者普通的 Java 应用程序, 来开发 TestBench 测试程序。

你可能会希望将测试程序类保存在你的应用程序工程中的独立的源代码树中, 或者完全保存在独立的工程中, 这样就可以不必将它们包含在 Web 应用程序的 WAR 文件中。将测试程序放在相同的工程内, 可能更利于源代码的版本管理。

21.4.1. 测试用例(**Test Case**) 的基本结构

JUnit 测试用例通过测试用例类的方法上的注解来定义。使用 TestBench 时, 测试用例类继承 **TestBenchTestCase** 类, 这个类提供了 WebDriver 和 ElementQuery API。

```
public class MyTestcase extends TestBenchTestCase {
```

在 TestBench 测试程序中使用的基本的 JUnit 注解如下:

@Rule

你可以使用 @Rule 注解来定义某些 TestBench 参数, 也可以定义其他 JUnit 规则。

比如, 为了启用测试失败时的屏幕截图(详情请参见第 21.10.2 节“测试失败时取得屏幕截图”), 你应该定义:

@Rule

```
public ScreenshotOnFailureRule screenshotOnFailureRule =
    new ScreenshotOnFailureRule(this, true);
```

注意, 如果使用这条规则, 在 @After 方法中, 一定不能调用 driver.quit() 方法, 因为这个方法会比屏幕截图更早执行, 而浏览器驱动必须保持打开才可以截取屏幕。

@Before

标注了这个注解的方法, 会在每个测试(通过 @Test 注解来标注)之前被执行。通常, 你可以在这里创建并设置浏览器驱动。

```
@Before
public void setUp() throws Exception {
    setDriver(new FirefoxDriver());
}
```

驱动类应该是 **FirefoxDriver**, **ChromeDriver**, **InternetExplorerDriver**, **SafariDriver**, 或 **PhantomJSDriver** 中的一个。关于目前的实现类一览, 请在 API 文档中查看 **RemoteWebDriver**。注意, 某些驱动需要安装浏览器驱动程序, 详情请参见第 21.3.5 节“安装浏览器驱动程序”。

驱动的示例保存在测试用例的 `driver` 属性中。你可以直接通过成员变量访问这个属性, 但应该使用 `setter` 方法设置这个属性。

`@Test`

这个注解标注一个测试方法。测试方法通常会打开页面, 然后执行命令, 最后对页面内容进行一些检查。

`@Test`

```
public void testClickButton() throws Exception {
    getDriver().get("http://localhost:8080/myproject");

    // Click the button
    ButtonElement button = $(ButtonElement.class).
        caption("Click Me").first();
    button.click();

    // Check that the label text is correct
    LabelElement label = $(LabelElement.class).first();
    assertEquals("Thanks!", label.getText());
}
```

通常, 对于所有测试中都相同的 URL, 应该使用变量来定义, 可能还需要将它与 URI 片段相加, 得到应用程序某一个状态的地址。

`@After`

每个测试执行结束后, 你可能会需要退出驱动, 关闭浏览器。

`@After`

```
public void tearDown() throws Exception {
    driver.quit();
}
```

但是, 如果你使用 **ScreenshotOnFailureRule**, 开启了测试失败时截取屏幕截图的功能, (详情请参见第 21.10.2 节“测试失败时取得屏幕截图”), 这段规则会在 `@After` 注解的方法之后执行, 但当着个规则截取屏幕时, 驱动需要处于打开状态。因此, 这种情况下你不应该退出驱动。这条规则会隐含地退出驱动。

你还可以使用 JUnit 的任何其他功能。但是, 要注意, 使用 TestBench 时要求驱动已被创建成功, 并且处于打开状态。

完整的测试用例的示例如下:

```
import com.vaadin.testbench.ScreenshotOnFailureRule;
import com.vaadin.testbench.TestBenchTestCase;
import com.vaadin.testbench.elements.ButtonElement;
import com.vaadin.testbench.elements.LabelElement;

import org.junit.Before;
```

```
import org.junit.Rule;
import org.junit.Test;
import org.openqa.selenium.firefox.FirefoxDriver;

import java.util.List;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

public class MyprojectTest extends TestBenchTestCase {
    @Rule
    public ScreenshotOnFailureRule screenshotOnFailureRule =
        new ScreenshotOnFailureRule(this, true);

    @Before
    public void setUp() throws Exception {
        setDriver(new FirefoxDriver()); // Firefox
    }

    /**
     * Opens the URL where the application is deployed.
     */
    private void openTestUrl() {
        getDriver().get("http://localhost:8080/myproject");
    }

    @Test
    public void testClickButton() throws Exception {
        openTestUrl();

        // At first there should be no labels
        assertFalse(${LabelElement.class}.exists());

        // Click the button
        ButtonElement clickMeButton = ${ButtonElement.class} .
            caption("Click Me").first();
        clickMeButton.click();

        // There should now be one label
        assertEquals(1, ${LabelElement.class}.all().size());

        // ... with the specified text
        assertEquals("Thank you for clicking",
                    ${LabelElement.class}.first().getText());

        // Click the button again
        clickMeButton.click();

        // There should now be two labels
        List<LabelElement> allLabels =
            ${LabelElement.class}.all();
        assertEquals(2, allLabels.size());

        // ... and the last label should have the correct text
        LabelElement lastLabel = allLabels.get(1);
        assertEquals("Thank you for clicking",
                    lastLabel.getText());
    }
}
```

这段测试程序框架代码，是由 Eclipse 的 Vaadin 工程创建向导，或 Maven archetype 创建的，详情请参见第 21.2 节“快速入门”。

21.4.2. 在 Eclipse 中运行 JUnit 测试程序

Eclipse IDE 集成了 JUnit，并添加了很好的控制功能，比如可以在当前测试程序源代码文件中运行测试。在 Eclipse 的 JUnit 视图区域中会生成可视化的测试结果报告。

使用 Vaadin Plugin for Eclipse 创建的新工程，已经包含了 TestBench API 的依赖项目，详情请参见第 21.2 节“快速入门”，因此你可以直接运行 TestBench 测试程序。

要对一个已经存在的工程，配置它的 TestBench 测试，你需要进行以下操作：

1. 在工程中添加 TestBench API 依赖项目。

- a. 如果使用的是 Vaadin Plugin for Eclipse 创建的工程，可以在 `ivy.xml` 中添加 TestBench API 库的依赖项目，如下：

```
<dependency org="com.vaadin"
            name="vaadin-testbench-api"
            rev="latest.release"
            conf="nodeploy->default"/>
```

TestBench API 库提供了针对 Vaadin 组件的元素类，因此它的版本号与它所支持的最新的 Vaadin 版本号一致。对于 Vaadin 的旧版本，你可以尝试使用上面给出的 `latest.release` 版本号。

工程中应该包含 `nodeploy` 配置，新的 Vaadin 工程会创建这个配置。详情请参见第 17.3 节“在 Eclipse 中使用 Ivy 安装 Add-on”。

- b. 否则的话，请从将安装包中的 `vaadin-testbench-api` 和 `vaadin-testbench-core` JAR 文件添加到工程的库文件夹中，比如 `lib` 目录。你不应该将库文件放到 `WEB-INF/lib` 目录下，因为部署到服务器上的 Vaadin Web 应用程序不会使用这个文件夹。请选中工程，然后按快捷键 **F5** 刷新工程。

2. 在 Project Explorer 中选中工程，按鼠标右键，并选择 **Properties**，然后打开 **Java Build Path** 中的 **Libraries** Tab 页。点击 **Add JARs** 按钮，找到库文件夹，选中库文件，然后点击 **OK** 按钮。

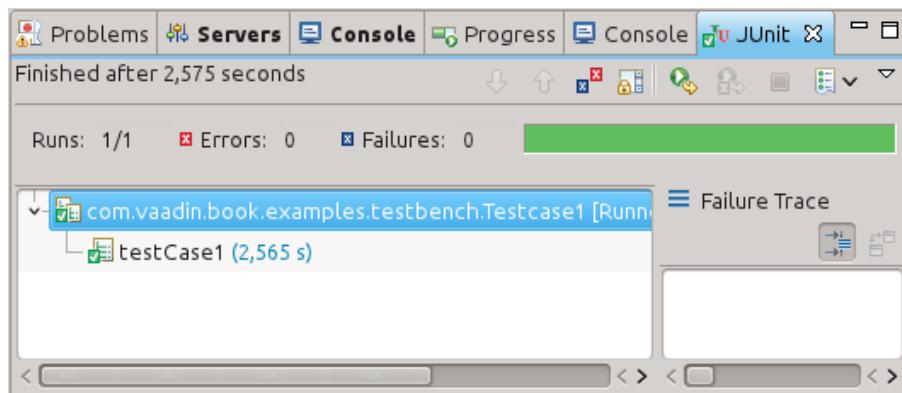
3. 在工程属性中切换到 **Order and Export** 页。确认 TestBench JAR 是否在 `gwt-dev.jar` 之上 (`gwt-dev.jar` 之内可能会包含一个老的 `httpclient` 包)，如果不是，可以选中它，并使用 **Up** 和 **Down** 按钮调整顺序。

4. 点击 **OK**，退出工程属性编辑界面。

5. 选中一个测试程序源文件，点击鼠标右键，选择 **Run As → JUnit Test**。

这时将出现一个 JUnit 视图，它应该会打开 Firefox 浏览器，启动应用程序，执行测试程序，然后关闭浏览器窗口。如果一切顺利，你的测试用例就执行通过了，结果会报告在 Eclipse 的 JUnit 视图区域中，见图 21.8 “在 Eclipse 中运行 JUnit 测试程序”。

图 21.8. 在 Eclipse 中运行 JUnit 测试程序



如果你使用的是其他 IDE，它可能也会支持 JUnit 测试程序。如果它不支持，你可以使用 Ant 或 Maven 来运行测试程序。

21.5. 创建 Test Case

21.5.1. 测试的启动

测试的配置可以在 `@Before` 注解所标记的方法之内进行。这个方法会在每个测试用例之前执行。

基本的配置内容包括：

- 设置 TestBench 参数
- 创建 Web Driver
- 其他初始化处理

TestBench 参数

TestBench 参数通过 `com.vaadin.testbench.Parameters` 类的静态方法来设置。参数主要与屏幕截图有关，详情请参见第 21.10 节“屏幕截图的取得和比较”。

21.5.2. 测试用例(Test Case)的基本结构

一个通常的测试用例的内容大致如下：

1. 打开 URL
2. 导航跳转到应用程序的某个状态
 - a. 找到一个 HTML 元素(**WebElement**)，作为后面的交互对象
 - b. 使用 `click()` 或其他命令与这个元素交互
 - c. 对其他元素重复以上步骤，直到应用程序到达期望的状态
3. 找到一个检查对象 HTML 元素(**WebElement**)
4. 取得并检查这个 HTML 元素的值

5. 取得屏幕截图

WebDriver 可以通过多种方式查找页面内的 HTML 元素, 比如, 使用 XPath 表达式. 元素元素的查找方法定义为 **By** 类中的静态方法.

以上测试内容对应的代码如下:

```
@Test
public void basic() throws Exception {
    getDriver().get("http://localhost:8080/tobetested");

    // Find an element to interact upon
    ButtonElement button =
        $(ButtonElement.class).id("mybutton");

    // Click the button
    button.click();

    // Check that the label text is correct
    LabelElement label = $(LabelElement.class).first();
    assertEquals("Thanks!", label.getText());
}
```

你也可以在 URL 中使用 URI 片段来访问应用程序的某个状态. 关于 URI 片段, 详情请参见第 11.11 节“管理 URI 片段”.

你应该使用 JUnit 断言语句来检查程序状态. 断言是定义在 org.junit.Assert 类中的静态方法, 举例来说, 你可以使用以下语句导入这个类:

```
import static org.junit.Assert.assertEquals;
```

关于页面元素查找方法的完整列表, 请参见 Selenium API 文档 中的 **WebDriver** 和 **By** 类, 关于与页面元素的交互命令, 请参见 **WebElement** 类.

TestBench 还有很多自己的命令, 定义在 TestBenchCommands 接口中. 你可以在测试用例中调用 testBench(driver), 得到一个 Command 对象, 并使用它.

虽然使用上述方法你可以很简单地编写测试用例, 但为了便于维护, 最好将测试代码进行更好的模块划分, 比如将业务逻辑层的测试与页面布局层的测试分离开. 为了这个目的可以使用页面对象, 详情请参见 第 21.9 节“创建可维护的测试程序”.

21.5.3. Web Driver 的创建和关闭

Vaadin TestBench 使用 Selenium WebDriver 在浏览器内执行测试. 可以使用 **TestBench** 类中的静态方法 createDriver() 来创建 **WebDriver** 实例. 这个方法接受一个浏览器驱动作为参数, 并注册这个浏览器驱动, 最后的返回值还是同一个浏览器驱动. 测试用例必须继承 **TestBenchTestCase** 类, 它会管理 TestBench 独有的功能. 你需要使用 setDriver() 方法, 将浏览器驱动保存在测试用例中.

通常的做法是, 在 JUnit @Before 注解标注的方法内创建浏览器驱动, 并在 @After 注解标注的方法内关闭浏览器驱动.

```
public class AdvancedTest extends TestBenchTestCase {
    @Before
    public void setUp() throws Exception {
        ...
        setDriver(TestBench.createDriver(new FirefoxDriver()));
    }
}
```

```

    }
    ...
    @After
    public void tearDown() throws Exception {
        driver.quit();
    }
}

```

以上代码会为测试用例之内的每一个测试创建一个浏览器驱动，导致每次测试之前都有新的浏览器实例被打开，测试结束后被关闭。如果你希望在整个测试期间保持浏览器实例不关闭，你可以使用 `@BeforeClass` 和 `@AfterClass` 注解标注的方法来创建和退出浏览器驱动。这种情况下，这些方法以及浏览器驱动的实例都必须是 static 的，而且你需要在 `@Before` 注解标注的方法中设置浏览器驱动。

```

public class AdvancedTest extends TestBenchTestCase {
    static private WebDriver driver;

    @BeforeClass
    static public void createDriver() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());
    }

    @Before
    public void setUp() throws Exception {
        setDriver(driver);
    }
    ...
    @AfterClass
    static public void tearDown() throws Exception {
        driver.quit();
    }
}

```

浏览器驱动

关于支持的浏览器驱动完整列表，请参见 `WebDriver` 接口的 API 文档，浏览器驱动类都实现了这个接口。

`Internet Explorer` 和 `Chrome` 都需要特殊的驱动程序，详情请参见 第 21.3.5 节“安装浏览器驱动程序”。驱动程序的可执行文件必须包含在操作系统的 PATH 路径内，还需要通过特殊的 Java 系统属性来指定：

- Chrome: `webdriver.chrome.driver`
- IE: `webdriver.ie.driver`

你可以在 Java 中使用 `System.setProperty(prop, key)` 方法来设置属性，也可以通过 Java 可执行文件的命令行参数来指定: `-Dwebdriver.chrome.driver=/path/to/driver`。

如果你使用 ESR 版本的 Firefox，(为保证测试的稳定性，我们推荐如此)，创建浏览器驱动时你需要指定可执行文件路径，如下：

```

FirefoxBinary binary =
    new FirefoxBinary(new File("/path/to/firefox_ESR_10"));
driver = TestBench.createDriver(
    new FirefoxDriver(binary, new FirefoxProfile()));

```

21.6. 查询页面元素

高级的 ElementQuery API 可以用于查询浏览器内的 Vaadin 组件，查询条件可以是组件类型，层级关系，标题，以及其他属性。找到一个或多个组件后，就可以与它们进行交互。查询 API 使用一种特定领域语言(Domain-Specific Language, DSL)，包含在 **TestBenchTestcase** 类中。

页面元素查询的基本思路是先匹配元素，再返回查询对象，返回的查询对象又可以继续进行查询，直到遇到一个终止查询才结束，终止查询返回一个或多个元素。

请看以下查询：

```
List<ButtonElement> buttons = $(ButtonElement.class).all();
```

这个查询返回 HTML 元素的列表，其中包含 UI 中所有的 **Button** 组件。每一个 Vaadin 组件都有对应的元素类，元素类都有方法可以与特定的组件进行交互。我们可以控制由这个查询找到的按钮，比如，可以点击按钮，如下：

```
for (ButtonElement b: buttons)
    b.click();
```

后面的小节中，我们介绍页面元素查询的细节。

21.6.1. 使用 Debug 窗口生成查询

你可以使用 Debug 窗口，非常简单地生成查询代码，生成的查询可以选择 UI 中的一个特定元素。刚开始使用 TestBench 时，这种方法非常有用，有助于理解查询的使用方法。

首先，使用 `&debug` 参数，打开应用程序的 Debug 窗口，详情请参见 第 11.3 节“Debug 模式和 Debug 窗口”。生成查询代码之前，你可以使用任何方法与 UI 交互，但我们建议你遵循用户在各个使用场景时正常的操作步骤，并在每一步都生成查询。

切换到 Debug 窗口的 TestBench Tab 页面，点击小按钮，启动 pick 模式。现在，当你将鼠标指针移动到页面元素之上时，页面元素将被高亮显示，当你点击一个元素时，将生成找到这个元素的 TestBench 的查询代码。Debug 窗口的使用，见图 21.9 “使用 Debug 窗口生成页面元素查询”。

图 21.9. 使用 Debug 窗口生成页面元素查询



你可以在 Debug 窗口中选中查询代码，并复制粘贴到你的编辑器中。要退出 pick 模式，请再次点击 pick 按钮。

Debug 窗口功能从 Vaadin 7.2 及以后的版本开始可用。

21.6.2. 使用组件类型(\$)查询页面元素

\$ 方法创建一个 **ElementQuery**, 查找指定的元素类型. 这个方法在 **TestBenchTestcase** 和 **ElementQuery** 类中都可用, 具体使用哪个类, 决定了查询的上下文. 查找会在上下文中递归进行.

```
// Find the first OK button in the UI
ButtonElement button = $(ButtonElement.class)
    .caption("OK").first();

// A nested query where the context of the latter
// component type query is the matching elements
// - matches the first Label inside the "content" layout.
LabelElement label = $(VerticalLayoutElement.class)
    .id("content").$(LabelElement.class).first();
```

21.6.3. 非递归的组件查询 (\$\$)

\$\$ 方法创建一个非递归的 **ElementQuery**. 它是一个快捷方法, 等同于首先使用 \$ 方法创建一个递归的查询, 然后对这个查询调用 recursive(false).

21.6.4. 页面元素类

每一个 Vaadin 组件在 TestBench 中都存在一个对应的元素类, 其中包含了与特定的组件进行交互的方法. 元素类继承自 **TestBenchElement**. 它实现了 Selenium 的 `WebElement` 接口, 因此 Selenium 的页面元素 API 也可以直接使用. 元素类通过 Vaadin 库发布, 而不是通过 TestBench, 因为它们必须与应用程序使用的 Vaadin 版本保持一致.

除组件外, 其他 Vaadin UI 元素, 比如通知(详情请参见 第 21.8.4 节 “测试通知信息”) 也存在对应的元素类. 其他的 Add-on 库也可以定义它们自己的页面元素类.

TestBenchElement 是一个 TestBench 命令执行器, 因此你可以使用一个页面元素, 在它的子元素树上创建查询. 比如, 下例中我们首先使用 ID 找到一个布局元素, 然后对它进行一次子查询找到布局之内的第一个 Label:

```
VerticalLayoutElement layout =
    $(VerticalLayoutElement.class).id("content");
LabelElement label = layout.$(LabelElement.class).first();
```

21.6.5. ElementQuery 对象

你可以使用一个 **ElementQuery** 对象, 创建子查询来完善查询结果, 或者使用一个结束符来结束查询, 并得到一个或多个匹配的页面元素.

21.6.6. 查询结束符

查询使用子查询来结束, 子查询返回单个元素, 或者多个元素的集合.

<code>first()</code>	返回第一个找到的元素.
<code>get()</code>	在匹配的元素集合中, 根据索引数值返回对用的元素.
<code>all()</code>	返回一个 <code>List</code> , 其中的内容是所有匹配的元素.

id()

返回拥有指定的 ID 的唯一元素. 元素 ID 在 Web 页面内必须始终是唯一的. 因此使用一个复杂的查询来匹配 ID 是无意义的, 只需要匹配元素类型就足够了.

页面元素

查询返回一个或多个页面元素, 页面元素继承自 Selenium 的 **WebElement** 类. 各元素类分别提供了独有的方法来操纵相关的 Vaadin 组件, 但你也可以使用 **WebElement** 类中定义的低级通用方法.

21.7. 元素选择器

除前一小节介绍的高级的 ElementQuery API 之外, Vaadin TestBench 还包括低级的 Selenium WebDriver API, 并做了一些 Vaadin 独有的扩展. 你也可以使用单纯的 XPath 表达式来查找元素, 还可以使用元素 ID, CSS 样式类, 等等. 你可以将这些选择器与元素查询组合起来使用. 与 ElementQuery API 类似, 这些 API 可以看作是一种特定领域语言(Domain-Specific Language, DSL), 包含在 **TestBenchTestcase** 类中.

可用的选择器定义为 **com.vaadin.testbench.By** 类中的静态方法. 这些方法创建并返回一个 **By** 实例, 你可以在 **WebDriver** 的 `findElement()` 方法中使用这个 **By** 实例.

ID, CSS 类, 以及 Vaadin 选择器将在下面的小节中介绍. 其他的选择器, 请参见 Selenium WebDriver API 文档.

有些选择器不能适用于所有的元素, 比如, 如果一个元素没有 ID, 或者它在 Vaadin 应用程序之外. 这种情况下, 会根据一个优先顺序使用其他选择器. 你可以修改选择器的优先顺序, 方法是选择菜单 **Options → Options → Locator Builders**, 然后拖动选择器(或定位器)的优先顺序. 通常, Vaadin selector 应该在优先顺序列表的最上方.

21.7.1. 通过 ID 查找

使用 HTML 元素的 `id` 属性来选择元素是一种比较健壮的方式, 这个问题我们会在 第 21.9.1 节 “增强选择器的健壮性” 中详细讨论. 这种方式要求你使用 `setId()` 方法为组件设置 ID.

```
Button button = new Button("Push Me!");
button.setId("pushmebutton");
```

按钮会被描绘为一个 HTML 元素: `<div id="pushmebutton" ...>...</div>`. 然后可以使用低级的 WebDriver 调用, 取得对应的 DOM 元素:

```
findElement(By.id("pushmebutton")).click();
```

这个选择器等价于静态类型的元素查询 `$(ButtonElement.class).id("pushmebutton")`.

21.7.2. 通过 CSS 类查找

带有特定 CSS 样式名称的元素可以使用 `By.className()` 方法来查找. 如果一个组件没有 ID, 也无法简单地通过组件层级关系查找, 但拥有一个特定的唯一的 CSS 样式, 那么 CSS 选择器是非常有用的. 提示信息(Tooltip)就是一个例子, 它们是浮动的 `div` 元素, 位于应用程序根元素之下. 由于它们带有 `v-tooltip` 样式, 所以可以使用以下方法选择:

```
// Verify that the tooltip contains the expected text
String tooltipText = findElement(
    By.className("v-tooltip")).getText();
```

完整的示例, 请参见 第 21.3.4 节 “TestBench 示例程序” 中介绍的 TestBench 示例程序中的 AdvancedCommandsITCase.java 文件.

21.8. 测试中的一些特殊问题

本节中, 我们介绍 TestBench 用来处理特殊情况的一些功能, 比如提示信息(Tooltip), 滚动, 通知信息, 上下文菜单, 以及测量应答性能. 最后, 我们将介绍页面对象模式(Page Object Pattern).

21.8.1. 等待 Vaadin 处理完毕

Vaadin TestBench 基于 Selenium, 而 Selenium 起初的目的是为了测试通常的 Web 应用程序, 这类应用程序装载的页面会立即由浏览器显示完毕. 在这类应用程序中, 你可以在页面装载后立即检查其中的元素. 在 Vaadin 以及其他 AJAX 应用程序中, 画面的显示是由异步的 JavaScript 代码完成的, 因此你需要等待服务器对 AJAX 请求给出应答, 以及 JavaScript 代码完成 UI 的显示工作. Selenium 支持 AJAX 应用程序, 可以使用一个特殊的等待方法, 查询 UI 状态, 直到画面显示完毕. 在纯 Selenium 程序中, 你需要明确地使用等待方法, 还需要知道什么时候使用, 以及使用哪一个方法. Vaadin TestBench 与 Vaadin 框架的客户端引擎配合工作, 可以实时检测画面显示是否结束. 等待处理是隐含进行的, 因此你通常不必自行添加任何等待命令.

等待功能是自动启用的, 但某些情况下也有可能会需要禁用等待. 你可以调用 `TestBenchCommands` 接口中的 `disableWaitForVaadin()` 方法来禁用等待. 你可以在测试用例中调用这个方法, 如下:

```
testBench(driver).disableWaitForVaadin();
```

当等待功能禁用时, 你可以明确地调用 `waitForVaadin()` 方法来等待画面显示结束.

```
testBench(driver).waitForVaadin();
```

你可以在同一个接口上调用 `enableWaitForVaadin()` 方法, 再次启用等待功能.

21.8.2. 测试提示信息(Tooltip)

当你将鼠标移动到组件上方时, 将会显示组件提示信息. 显示提示信息需要特殊的命令. 处理提示信息也需要特殊的方法, 因为提示信息是一个浮动的覆盖元素, 它不属于通常的组件层级关系.

假设你为组件设置了提示信息, 如下:

```
// Create a button with a component ID
Button button = new Button("Push Me!");
button.setId("main.button");

// Set the tooltip
button.setDescription("This is a tip");
```

组件的提示信息可以使用 `TestBenchElementCommands` 接口的 `showTooltip()` 方法来显示. 你应该等待一小段时间, 确保提示信息正确显示. 浮动的提示信息元素并不在组件的页面元素之下, 但你可以通过 XPath 表达式 `//div[@class='v-tooltip']` 来找到它.

```
@Test
public void testTooltip() throws Exception {
    driver.get(appUrl);

    ButtonElement button =
        $(ButtonElement.class).id("main.button");
```

```
button.showTooltip();

WebElement ttip = findElement(By.className("v-tooltip"));
assertEquals(ttip.getText(), "This is a tip");
}
```

21.8.3. 滚动

某些 Vaadin 组件带有滚动条, 比如 **Table** 和 **Panel**. 通常, 当你与这类滚动区域之内的元素进行交互时, TestBench 会隐含地试图滚动到对象元素, 使它变得可见. 某些情况下, 你可能希望显式地滚动一个滚动条. 你可以对可滚动的组件分别使用 scroll() 方法(垂直方向)和 scrollLeft() 方法(水平方向)来控制它的滚动条. 滚动位置参数的单位为像素.

```
// Scroll to a vertical position
PanelElement panel = $(PanelElement.class)
    .caption("Scrolling Panel").first();
panel.scroll(123);
```

21.8.4. 测试通知信息

你可以使用页面元素查询 API 中的 **NotificationElement** 类, 查找通知信息的页面元素. 这个页面元素类可以通过 getCaption() 方法取得标题, 通过 getDescription() 方法取得描述信息, 通过 getType() 方法取得通知类型.

假设你弹出了一个通知信息, 如下:

```
Button button = new Button("Pop It Up", e -> // Java 8
    Notification.show("The caption", "The description",
        Notification.Type.WARNING_MESSAGE));
```

你可以在测试程序中检查这个通知信息, 如下:

```
// Click the button to open the notification
ButtonElement button =
    $(ButtonElement.class).caption("Pop It Up").first();
button.click();

// Verify the notification
NotificationElement notification =
    $(NotificationElement.class).first();
assertEquals("The caption", notification.getCaption());
assertEquals("The description", notification.getDescription());
assertEquals("warning", notification.getType());
notification.close();
```

你需要使用 close() 方法关闭通知信息的对话框, 应用程序才能继续执行.

21.8.5. 测试上下文菜单

打开上下文菜单需要特殊的处理. 首先, 要打开一个菜单, 你需要在一个支持上下文菜单的组件的特定子元素之内, "点击鼠标右键"(context-click). 你可以通过 **Actions** 对象的 contextClick() 动作来实现这个任务.

上下文菜单会显示为一个浮动的元素, 这个元素包含在 HTML 页面中的一个特殊的覆盖元素之内, 而不是在弹出菜单的那个组件之内. 你可以使用这个元素的 CSS 类 v-contextmenu, 在页面内查找这个元素. 菜单项目表现为文本, 你可以使用下例中的 XPath 表达式查找这些文本元素.

下例中，我们在 **Table** 组件内打开上下文菜单，通过标题文字找到一个菜单项，然后点击这个菜单项。

```
// Get a table cell to work on
TableElement table = inExample(TableElement.class).first();
WebElement cell = table.getCell(3, 0); // A cell in the row

// Perform context click action to open the context menu
new Actions(getDriver()).contextClick(cell).perform();

// Find the opened menu
WebElement menu = findElement(By.className("v-contextmenu"));

// Find a specific menu item
WebElement menuitem = menu.findElement(
    By.xpath("//*[text() = 'Add Comment']]"));

// Select the menu item
menuitem.click();
```

21.8.6. 测量测试程序的执行时间

我们不仅需要测试程序是否正常工作，还关心它的执行时间。不断地测量测试程序的执行时间是很重要的，因为测试环境可能会有各种不同的延迟和干扰。比如，在一个分布式测试环境中，测试服务器上耗费的时间会包含测试服务器，网格 hub，运行浏览器的网格节点，和运行应用程序的 Web 服务器之间的网络传输延迟。在这样一个环境中，你还应该预计到多个测试节点之间会发生相互干扰，因为它们可能会向同一个应用程序服务器发起请求，也可能会共用虚拟机的资源。

而且，在 Vaadin 应用程序中，存在两个部分需要测量：服务器端，它执行应用程序的逻辑，以及客户端，它显示在浏览器内。Vaadin TestBench 针对服务器端和客户端，都包含了测量执行时间的方法。

`TestBenchCommands` 接口提供了以下方法，用于测量测试程序的执行时间：

`totalTimeSpentServicingRequests()`

返回应用程序服务器端响应请求时消耗的总时间(毫秒单位)。当你第一次跳转到应用程序之内，启动一个新 session 时，计时开始。只有在应用程序响应某个 session 的请求时，才会计时。计时器在 Servlet 的 Session 内共享，因此，比如说，如果在同一个应用程序(Session)之内，你有多个 portlet，那么它们的执行时间将被合计在一起。

注意，如果你还关心最后一次请求的客户端性能，你必须在调用本方法之前调用 `timeSpentRenderingLastRequest()` 方法。这是因为，这个方法会向服务器发起一次额外的请求，这个请求会导致一个不需要在画面上显示的应答。

`timeSpentServicingLastRequest()`

返回应用程序服务器端响应最后一次请求时消耗的时间(毫秒单位)。注意，通过 WebDriver 发起的用户交互，并不是全部都会导致服务器请求。

与前面所说的获取总时间的方法一样，如果你还关心最后一次请求的客户端性能，你必须在调用本方法之前调用 `timeSpentRenderingLastRequest()` 方法。

`totalTimeSpentRendering()`

返回应用程序客户端(也就是在浏览器中)描绘 UI 时消耗的总时间(毫秒单位)。只有在通过 WebDriver 与浏览器交互，并因此导致浏览器描绘 UI 界面时，这个时间才会计时。计时器在 Servlet 的 Session 内共享，因此，比如说，如果在同一个应用程序(Session)之内，你有多个 portlet，那么它们的执行时间将被合计在一起。

timeSpentRenderingLastRequest()
返回应用程序在最后一次服务器请求之后描绘 UI 时消耗的时间(毫秒单位). 注意, 通过 WebDriver 发起的用户交互, 并不是全部都会导致服务器请求.

如果你还调用 timeSpentServicingLastRequest() 方法或 totalTimeSpentServicingRequests() 方法, 那么应该在本方法之前调用这些方法(译注: 此处貌似写反了). 这些方法会导致一次服务器请求, 并导致本方法测量的画面描绘时间变为 0.

一般来说, 只有与 立即 模式的 Field 组件交互时才会导致服务器请求, 包括按钮的点击. 某些组件, 比如 **Table**, 也会导致服务器请求, 比如表格滚动导致需要装载数据时. 某些交互会导致多次请求, 比如, 作为用户交互的结果, 导致需要从服务器装载图片.

以下示例程序在 TestBench 示例程序的 VerifyExecutionTimeITCase.java 文件中.

```
@Test
public void verifyServerExecutionTime() throws Exception {
    // Get start time on the server-side
    long currentSessionTime = testBench(getDriver())
        .totalTimeSpentServicingRequests();

    // Interact with the application
    calculateOnePlusTwo();

    // Calculate the passed processing time on the serve-side
    long timeSpentByServerForSimpleCalculation =
        testBench().totalTimeSpentServicingRequests() -
        currentSessionTime;

    // Report the timing
    System.out.println("Calculating 1+2 took about "
        + timeSpentByServerForSimpleCalculation
        + "ms in servlets service method.");

    // Fail if the processing time was critically long
    if (timeSpentByServerForSimpleCalculation > 30) {
        fail("Simple calculation shouldn't take " +
            timeSpentByServerForSimpleCalculation + "ms!");
    }

    // Do the same with rendering time
    long totalTimeSpentRendering =
        testBench().totalTimeSpentRendering();
    System.out.println("Rendering UI took "
        + totalTimeSpentRendering + "ms");
    if (totalTimeSpentRendering > 400) {
        fail("Rendering UI shouldn't take " +
            totalTimeSpentRendering + "ms!");
    }

    // A normal assertion on the UI state
    assertEquals("3.0",
        $(TextFieldElement.class).first()
            .getAttribute("value"));
}
```

21.9. 创建可维护的测试程序

开发测试程序最重要的规则是，保持代码可读，可维护。否则，当测试失败时，比如应用程序代码重构之后，开发者会觉得难以理解测试程序，难以修复错误，所以很容易禁用这些测试程序。可以使用页面对象模式(Page Object Pattern)来改进可读性和可维护性，详情将在下文中介绍。

第二条规则是，应该经常运行测试程序。最好使用一个持续集成服务器运行测试程序，运行的频度至少每天一次，如果每次代码提交之后都运行测试则更好。

21.9.1. 增强选择器的健壮性

为了避免 HTML DOM 树结构中的无关变化导致测试失败，保证测试程序的健壮性是很重要的。不同的选择器的健壮性是不同的，也取决于如何使用这些选择器。

ElementQuery API 使用 Widget 的逻辑层次结构来查找对应的 HTML 元素，而不是使用实际的 HTML DOM 结构。这种方式使得它比较健壮，但仍然会受到 UI 中组件层级结构变化的影响。而且，如果你对应用程序进行了国际化，那么就无法通过组件的标题文字来查找它了。

低级的 XPath 选择器极易收到 DOM 路径变化的影响，尤其是查询路径从页面的 body 元素开始时。但是这个选择器非常灵活，也有一种健壮的使用方式，比如，可以使用 HTML 元素和 CSS 类名或属性名来进行选择。你可以象 CSS 选择器那样，以一种健壮的方式，使用 CSS 类来选择特定的组件。

使用组件 ID 增强测试程序健壮性

为了使 UI 在测试中更加健壮，你可以使用 `setId()` 方法，为特定的组件设置唯一的组件 ID，详情请参见第 21.7.1 节“通过 ID 查找”。

让我们来考虑一下以下应用程序，在这个应用程序中，我们为组件设置一种层级式的 ID，以保证 ID 唯一；在更加模块化的情况下，你也可以考虑使用其他方案。

```
public class UIToBeTested extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setId("myui");

        final VerticalLayout content = new VerticalLayout();
        content.setMargin(true);
        content.setId("myui.content");
        setContent(content);

        // Create a button
        Button button = new Button("Push Me!");

        // Optional: give the button a unique ID
        button.setId("myui.content.pushmebutton");

        content.addComponent(button);
    }
}

// Click the button
ButtonElement button =
```

用这种方式准备好应用程序之后，你就可以使用 `id()` 查询结束符，通过组件 ID 来查找页面元素了。

```
$ (ButtonElement.class).id("myui.content.pushmebutton");
button.click();
```

组件 ID 就是 HTML 元素的 id 属性, 在整个 UI 内必须是唯一的, 如果 HTML 页面除这个 UI 之外还包含其他内容, 那么 ID 在整个 HTML 页面之内也必须是唯一的. 如果存在多个 UI, 你可以在 ID 中包含 UI 名前缀, 上例中我们就是这样做的.

使用 CSS 类名增强测试程序健壮性

与使用组件 ID 的方法类似, 你可以使用 addStyleName() 方法为组件添加 CSS 类名. 然后就可以使用 findElement(By.className()) 选择器来查找这些组件, 详情请参见第 21.7.2 节“通过 CSS 类查找”. 你可以在元素查询中使用这种选择器. 与 ID 不同, CSS 类名不必是唯一的, 因此一个 HTML 页面内可以包含多个元素带有相同的 CSS 类.

你也可以在 XPath 选择器中使用 CSS 类名.

21.9.2. 页面对象模式(Page Object Pattern)

页面对象模型的目标是使应用程序视图测试更加简单化和模块化. 这个模式遵循关注点分离(separation of concerns)原则, 通过不同的模块来处理关注的不同问题, 并将其他测试不应该知道的信息封装起来.

定义一个页面对象

一个页面对象带有一些方法, 可以与视图或子视图交互, 还可以取得视图中的数值. 你还需要一个方法打开页面并跳转到适当的视图.

比如:

```
public class CalculatorPageObject
    extends TestBenchTestCase {
    @FindBy(id = "button_=")
    private WebElement equals;
    ...

    /**
     * Opens the URL where the calculator resides.
     */
    public void open() {
        getDriver().get(
            "http://localhost:8080/?restartApplication");
    }

    /**
     * Pushes buttons on the calculator
     *
     * @param buttons the buttons to push: "123+2", etc.
     * @return The same instance for method chaining.
     */
    public CalculatorPageObject enter(String buttons) {
        for (char numberChar : buttons.toCharArray()) {
            pushButton(numberChar);
        }
        return this;
    }

    /**
     * Pushes the specified button.
     *
```

```

    *
    * @param button The character of the button to push.
    */
    private void pushButton(char button) {
        getDriver().findElement(
            By.id("button_" + button)).click();
    }

    /**
     * Pushes the equals button and returns the contents
     * of the calculator "display".
     *
     * @return The string (number) shown in the "display"
     */
    public String getResult() {
        equals.click();
        return display.getText();
    }

    ...
}

```

通过 **ID** 查找页面对象中的成员元素

如果页面对象中存在 **WebElement** 类型的成员变量，并使用 **@FindBy** 注解标注它，那么这些成员变量会被自动地设置为与指定的组件 **ID** 匹配的 **HTML** 元素，和使用 `driver.findElement(By.id(fieldname))` 的结果一样。要执行这个页面元素的自动查找工作，你需要使用 **PageFactory** 来创建页面对象，如下例：

```

public class PageObjectExampleITCase {
    private CalculatorPageObject calculator;

    @Before
    public void setUp() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());

        // Use PageFactory to automatically initialize fields
        calculator = PageFactory.initElements(driver,
            CalculatorPageObject.class);
    }
    ...
}

```

页面对象中的成员元素必须定义为 **WebElement** 类型，但你可以使用 `wrap()` 方法将它转换为一个具体的元素类型：

```
ButtonElement equals = equalsElement.wrap(ButtonElement.class);
```

使用页面对象

在测试用例中可以在业务逻辑层使用页面对象的方法，而不必意识到视图的具体组成结构。

比如：

```

@Test
public void testAddCommentRowToLog() throws Exception {
    calculator.open();

    // Just do some math first
    calculator.enter("1+2");
}

```

```

    // Verify the result of the calculation
    assertEquals("3.0", calculator.getResult());
    ...
}

```

页面对象示例

页面对象模式的完整的例子可以在 `src/test/java/com/vaadin/testbenchexample/pageobjectexample/PageObjectExampleITCase.java` 文件夹中找到. `PageObjectExampleITCase.java` 会对 Calc UI (也包含在示例程序源代码中) 运行测试程序, 测试程序使用页面对象与 UI 中的不同部分进行交互, 并检查运行结果.

子文件夹 `pageobjects` 中包含的页面对象如下:

- **CalculatorPageObject** (前文的示例代码已经列举了它的大致内容), 其中存在一些方法用来点击计算器中的按钮, 以及取得 "display" 中显示的计算结果.
- **LogPageObject** 可以取得日志 Table 中日志条目的内容, 并在日志条目上点击鼠标右键, 打开注释子窗口.
- **AddComment** 可以在注释编辑子窗口中输入一个注释字符串, 并提交(点击 **Add** 按钮).

21.10. 屏幕截图的取得和比较

你可以取得屏幕截图, 并与之前取得的参照截图进行比较. 如果二者存在不同, 你可以将这个测试用例判定为失败.

21.10.1. 屏幕截图参数

屏幕截图参数使用 `com.vaadin.testbench.Parameters` 类中的静态方法来设置.

`screenshotErrorDirectory` (默认值: null)
指定一个目录, 用于保存测试失败或比较失败时的屏幕截图.

`screenshotReferenceDirectory` (默认值: null)
指定一个目录, 用于保存屏幕截图比较时使用的参照图片.

`screenshotComparisonTolerance` (默认值: 0.01)
屏幕截图通常并不使用完全一致的像素值进行比较, 因为在浏览器内显示时经常会存在一些微小的偏差. 而且图片的压缩也会导致一些变化.

`screenshotComparisonCursorDetection` (默认值: false)
有些 Field 组件获得输入焦点时会出现闪烁的光标. 光标可能会导致不必要的比较失败, 因为光标会闪烁, 使得截取屏幕时光标可能正好可见也可能正好不可见. 这个参数启动光标检测功能, 尽量减少光标导致的比较失败.

`maxScreenshotRetries` (默认值: 2)
有时屏幕截图的比较可能会失败, 因为屏幕描绘还未结束, 或者屏幕截图中存在一个闪烁的光标, 与参照的屏幕截图不一致. 由于这些原因, Vaadin TestBench 会重试屏幕截图比较, 重试次数由这个参数指定.

`screenshotRetryDelay` (默认值: 500)
当屏幕截图比较失败时, 会重试. 这个参数指定重试之前的时间延迟, 单位为毫秒.

比如:

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory(
        "screenshots/errors");
    Parameters.setScreenshotReferenceDirectory(
        "screenshots/reference");
    Parameters.setMaxScreenshotRetries(2);
    Parameters.setScreenshotComparisonTolerance(1.0);
    Parameters.setScreenshotRetryDelay(10);
    Parameters.setScreenshotComparisonCursorDetection(true);
    Parameters.setCaptureScreenshotOnFailure(true);
}
```

21.10.2. 测试失败时取得屏幕截图

Vaadin TestBench 可以在测试失败时自动取得屏幕截图。要启用这个功能，你需要在测试用例中，使用一个由 **@Rule** 注解标注的成员变量，来包含 **ScreenshotOnFailureRule** JUnit 规则，如下：

```
@Rule
public ScreenshotOnFailureRule screenshotOnFailureRule =
    new ScreenshotOnFailureRule(this, true);
```

注意，你一定不能在 **@After** 方法中对浏览器驱动调用 **quit()** 方法，因为这会导致在 JUnit 规则取得屏幕截图之前就关闭了浏览器驱动。

屏幕截图将被写入 **screenshotErrorDirectory** 参数指定的错误目录。你可以在测试用例的设置方法中配置这个参数，如下：

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory("screenshots/errors");
    ...
}
```

21.10.3. 取得用于比较的屏幕截图

Vaadin TestBench 可以使用 **TestBenchCommands** 接口中的 **compareScreen()** 命令取得 Web 浏览器窗口的屏幕截图。这个方法存在很多变体。

compareScreen(File) 方法的参数是一个 **File** 对象，指向比较用的参照图片。这种情况下，错误图片会被使用同样的文件名写入到错误目录中。

你可以使用静态的帮助方法 **ImageFileUtil.getReferenceScreenshotFile()**，得到指向参照图片的 **File** 对象。

```
assertTrue("Screenshots differ",
    testBench(driver).compareScreen(
        ImageFileUtil.getReferenceScreenshotFile(
            "myshot.png")));
```

compareScreen(String) 方法的参数是屏幕截图的基础文件名(base name)。这个基础文件名再加上浏览器标识符和文件扩展名，就是比较用的参照图片的实际文件名。

```
assertTrue(testBench(driver).compareScreen("tooltip"));
```

compareScreen(BufferedImage, String) 可以与保存在内存中的参照图片进行比较。错误图片会被写入到文件，文件名根据第二个参数指定的基础文件名自动决定。

使用 `compareScreen()` 方法取得的屏幕截图将与存储在参照图片文件夹内的参照图片进行比较。如果发现图片中存在差异(或者参照图片不存在), 比较方法将返回 `false`, 并将屏幕截图保存到错误文件夹中。比较方法还会产生一个 HTML 文件, 将图片中存在差异的区域高亮显示。

屏幕截图比较的错误图片

错误情况下的屏幕截图会被写入到错误文件夹中, 这个文件夹的路径由 `screenshotErrorDirectory` 参数指定, 详情请参见第 21.10.1 节“屏幕截图参数”。

比如, 由于参照图片不存在导致的错误可能被写入到 `screenshot/errors/tooltip_firefox_12.0.png` 文件中。图片见图 21.10 “测试程序运行中取得的屏幕截图”。

图 21.10. 测试程序运行中取得的屏幕截图



屏幕截图包括浏览器内页面的可见区域, 因此屏幕截图的比较也与浏览器的尺寸相关。浏览器通常会被调整为预定义的默认尺寸。你可以通过很多方法设置浏览器窗口的尺寸。比如, 可以在 `@Before` 方法内通过 `driver.manage().window().setSize(new Dimension(1024, 768))`; 来设置。这个尺寸包含浏览器本身的窗口框架, 因此实际取得的屏幕截图尺寸会更小一些。如果要设置实际视图区域的尺寸, 你可以使用 `TestBenchCommands.resizeViewPortTo(1024, 768)` 方法。

参照图片

参照图片是应该存在于参照图片文件夹内, 文件夹路径由 `screenshotReferenceDirectory` 参数指定, 详情请参见 第 21.10.1 节“屏幕截图参数”。要创建参照图片, 只需要将屏幕截图从 `errors/` 文件夹复制到 `reference/` 文件夹即可。

比如:

```
$ cp screenshot/errors/tooltip_firefox_12.0.png screenshot/reference/
```

现在正确的参照图片已经存在了, 所以再次运行测试程序会输出测试成功的信息:

```
$ java ...
JUnit version 4.5
.
Time: 18.222

OK (1 test)
```

屏幕截图的遮蔽(Mask)

屏幕截图比较时, 你可以使用带透明区域的参照图片来遮蔽(Mask)其中一部分. 参照图片中的透明像素(也就是 Alpha 通道值小于 1.0 的像素), 在屏幕截图比较中将被忽略.

关于使用带遮蔽的屏幕截图的示例, 请 TestBench 示例程序中的 `ScreenshotITCase.java` 示例程序 `example/Screenshot_Comparison_Tests.pdf` 文档介绍了如何启用这个示例程序, 以及如何使用图片编辑器来创建带遮蔽的屏幕截图.

通过高亮度来显示屏幕截图中的差异部分

Vaadin TestBench 支持一种高级的显示方式, 可以显示出抓取的屏幕截图与参照图片之间的差异. 差异报告会被写入到一个 HTML 文件中, 文件名与测试失败的屏幕截图一样, 但使用 `.html` 扩展名. 差异报告与测试失败的屏幕截图一样, 会写入到同一个 `errors/` 文件夹中.

图片中存在差异的部分会使用蓝色方框高亮标出. 将鼠标指针移动到某个差异区域, 将会显示出这个区域在参照图片中对应区域的内容. 点击图片会将整个图片切换到参照图片, 再次点击将切换回实际的屏幕截图. 在屏幕截图的左上角会显示会显示文字 "**Image for this run**", 以便与参照图片区别开.

图 21.11 “参照图片和被高亮显示的错误图片” 显示了一个差异报告, 其中存在一个差异区域, 屏幕截图显示在下方, 参照图片显示在上方.

图 21.11. 参照图片和被高亮显示的错误图片



21.10.4. 处理屏幕截图时的一些实际经验

屏幕截图的参照图片文件夹需要设置好读写权限, 以便开发者可以查看测试程序的运行结果, 并将正确的图片复制到参照图片文件夹中. 一种可行的方案是将参照图片保存到一个版本管理系统内, 再将它们 check-out 到 `reference/` 文件夹中.

在构建系统或持续集成系统中, 可以配置一个 Build Artifact, 自动地收集和保存屏幕截图.

21.10.5. 已知的兼容性问题

Internet Explorer 9 运行在兼容模式时的屏幕截图

Internet Explorer 在版本 9 之前会在内容区域周围加上 2 像素的边框. 版本 9 不会再加这个边框, 因此使用 Internet Explorer 9 运行在兼容模式时(IE7/IE8)得到的屏幕截图会包括这个 2 像素的边框, 这一点与旧版本的 Internet Explorer 不同(译注: 此段理解不能, 待校).

21.11. 运行测试

在测试程序的开发阶段, 你通常会在你的 IDE 内运行. 开发完毕后, 你会希望通过构建系统来运行测试, 有可能在持续集成系统中运行. 下面, 我们介绍如何使用 Ant 和 Maven 来运行测试程序.

21.11.1. 使用 Ant 运行测试

Apache Ant 内部支持运行 JUnit 测试程序, 你可以在 Ant 脚本中使用 `<junit>` task 来执行 JUnit 测试程序. 注意, 在较老的版本中, 要启用这个功能, 你需要将 JUnit 的库文件 `junit.jar` 以及它与 Ant 集成的库文件 `ant-junit.jar` 放在 Ant 的类路径中, 详情请参见 Ant 文档.

以下 Ant 脚本可以测试一个通过 Vaadin Plugin for Eclipse 创建的 Vaadin 应用程序. 这段脚本假设测试程序源代码位于当前目录之下的 `test` 目录内, 然后它编译测试程序, 输出到 `classes` 目录内. 类路径定义在参照 ID(reference ID) `classpath` 中, 其中应该包含 `TestBench` 以及其他必要的库.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="run-tests">
    <path id="classpath">
        <fileset dir="lib">
            <include name="vaadin-testbench-*.jar"/>
            <include name="junit-*.jar"/>
        </fileset>
    </path>

    <!-- This target compiles the JUnit tests. -->
    <target name="compile-tests">
        <mkdir dir="classes" />
        <javac srcdir="test" destdir="classes"
            debug="on" encoding="utf-8"
            includeantruntime="false">
            <classpath>
                <path refid="classpath" />
            </classpath>
        </javac>
    </target>

    <!-- This target calls JUnit -->
    <target name="run-tests" depends="compile-tests">
        <junit fork="yes">
            <classpath>
                <path refid="classpath" />
                <path element path="classes" />
            </classpath>

            <formatter type="brief" usefile="false" />

            <batchtest>
                <fileset dir="test">
                    <include name="**/**.java" />
                </fileset>
            </batchtest>
        </junit>
    </target>
</project>
```

你还需要将被测应用程序部署到服务器上, 可能需要为它启动一个专用的服务器.

通过 Ivy 获取 TestBench

要在 Ant 中使用 Ivy 来取得 TestBench 及其依赖项目，首先，如果需要的话，要在你的 Ant 环境内安装 Ivy。在构建脚本内，你需要使用 namespace 声明来启用 Ivy，并包含一个用来获取库文件的构建目标(target)，如下：

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
         default="run-tests">
    ...
    <!-- Retrieve dependencies with Ivy -->
    <target name="resolve">
        <ivy:retrieve conf="testing" type="jar,bundle"
                      pattern="lib/[artifact]-[revision].[ext]"/>
    </target>

    <!-- This target compiles the JUnit tests. -->
    <target name="compile-tests" depends="resolve">
        ...
    
```

以上脚本要求在你的 ivy.xml 文件中存在一个 "testing" 配置，而且在这个配置中定义了 TestBench 依赖项目。

```
<ivy-module>
    ...
    <configurations>
        ...
        <conf name="testing" />
    </configurations>

    <dependencies>
        ...
        <!-- TestBench 4 -->
        <dependency org="com.vaadin"
                    name="vaadin-testbench-api"
                    rev="latest.release"
                    conf="nodeploy,testing -> default" />
        ...
    
```

你还需要编译被测应用程序，并发布到服务器，此外还需要安装 TestBench 的 license key。

21.11.2. 使用 Maven 运行测试

在 Maven 中使用 Vaadin TestBench 来执行 JUnit 测试，需要在 POM 文件内将 TestBench 定义为一个依赖项目。

关于 Maven 测试环境的完整示例，可参见 TestBench 示例程序工程，地址是 github.com/vaadin/testbench-demo。详情请阅读其中的 README 文件。

将 TestBench 定义为依赖项目

你需要在你的工程的 Maven POM 文件中，将 TestBench 库定义为依赖项目，如下：

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-testbench</artifactId>
    <version>4.x.x</version>
</dependency>
```

关于如何使用 Maven 来创建新的 Vaadin 工程, 详情请参见 第 2.6 节 “通过 Maven 使用 Vaadin”.

运行测试程序

要编译并运行测试程序, 只需要使用 Maven 简单地执行 `test` 生命周期阶段(Lifecycle Phase), 如下:

```
$ mvn test
...
-----
T E S T S
-----
Running TestBenchExample
Tests run: 6, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 36.736 sec <<<
FAILURE!

Results :

Failed tests:
  testDemo(TestBenchExample):
    expected:<[5/17/]12> but was:<[17.6.20]12>

Tests run: 6, Failures: 1, Errors: 0, Skipped: 1
...
```

示例程序中的配置会启动 Jetty 来运行被测应用程序.

如果你启用了屏幕截图测试(详情请参见 第 21.3.4 节 “TestBench 示例程序”), 屏幕截图比较将会导致测试失败. 失败的屏幕截图会被写入到 `target/testbench/errors` 文件夹. 为了将它们与“期待的”屏幕截图进行比较, 你需要将这些屏幕截图复制到 `src/test/resources/screenshots/reference/` 文件夹中. 关于屏幕截图, 详情请参见 第 21.10 节 “屏幕截图的取得和比较”.

21.12. 在分布式环境中运行测试

分布式测试环境包括一个网格 hub, 以及多个测试节点. hub 监听从测试运行器发起的调用, 并将这些调用转发给网格节点. 不同的节点可以运行在不同的操作系统平台之上, 并安装不同的浏览器.

第 21.3.2 节 “分布式测试环境” 介绍了基本的分布式测试环境.

21.12.1. 远程运行测试

远程测试与本地运行的测试类似, 区别在于使用的不是浏览器驱动, 而是远程 web 驱动, 这个驱动可以连接到 hub. hub 将连接转发到一个浏览器能力符合需求的网格节点, 也就是说, 转发到安装在一个节点上的浏览器上.

下面我们介绍如何明确地创建并管理远程驱动程序, 但除了这种方式之外, 你还可以使用 **ParallelTest** 框架, 详情请参见 第 21.13 节 “测试程序的并行执行”.

关于远程运行测试的示例, 请参见 第 21.3.4 节 “TestBench 示例程序” 中的 TestBench 示例程序. 更多细节请阅读其中的 `README.md` 文件.

下例中, 我们创建并使用一个远程浏览器驱动, 它将在位于 `testingbot.com` 的 Selenium 云端服务中运行测试程序. 测试节点需要的浏览器能力通过 **DesiredCapabilities** 对象来描述.

```
public class UsingHubITCase extends TestBenchTestCase {
```

```

private String baseUrl;
private String clientKey = "INSERT-YOUR-CLIENT-KEY-HERE";
private String clientSecret = "INSERT-YOUR-CLIENT-KEY-HERE";

@Before
public void setUp() throws Exception {
    // Create a RemoteDriver against the hub.
    // In your local setup you don't need key and secret,
    // but if you use service like testingbot.com, they
    // can be used for authentication
    URL testingbotdotcom = new URL("http://" +
        clientKey + ":" + clientSecret +
        "@hub.testingbot.com:4444/wd/hub");
    setDriver(new RemoteWebDriver(testingbotdotcom,
        DesiredCapabilities.iphone()));
    baseUrl = "http://demo.vaadin.com/Calc/";
}

@Test
@Ignore("Requires testingbot.com credentials")
public void testOnePlusTwo() throws Exception {
    // run the test just as with "local bots"
    openCalculator();
    $(ButtonElement.class).caption("1").first().click();
    $(ButtonElement.class).caption("+").first().click();
    $(ButtonElement.class).caption("2").first().click();
    $(ButtonElement.class).caption("=").first().click();
    assertEquals("3.0", $(TextFieldElement.class)
        .first().getAttribute("value"));

    // That's it. Services may provide also some other goodies
    // like the video replay of your test in testingbot.com
}

private void openCalculator() {
    getDriver().get(baseUrl);
}

@After
public void tearDown() throws Exception {
    getDriver().quit();
}
}

```

关于支持的浏览器能力的完整列表, 请参见 **DesiredCapabilities** 类的 API 文档.

要运行上面的例子, 需要 hub 服务以及网格节点都在运行中. 它们的启动方法将在后面的小节中介绍. 详情请参见 Selenium 文档.

21.12.2. 启动 Hub

TestBench 网格 hub 监听测试运行器发起的调用, 并将这些调用转发给网格节点. 网格 hub 服务包含在 Vaadin TestBench 的 JAR 文件之内, 你可以使用以下命令启动它:

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar \
    -role hub
```

你也可以使用 Web 浏览器打开 hub 的控制界面。使用默认的端口时，打开 URL <http://localhost:4444/>。如果你启动了一个或多个网格节点(方法将在下一节中介绍)，"console" 页面将会列出所有的网格节点以及它们的浏览器能力。

21.12.3. 测试节点的服务配置

测试节点可以通过命令行选项来配置，详情见后文，也可以使用 JSON 格式的配置文件。如果没有配置文件，将使用默认配置。

节点配置文件通过节点服务程序的 `-nodeConfig` 参数来指定，如下例：

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar  
-role node -nodeConfig nodeConfig.json
```

关于节点服务程序的启动，详情请参见 第 21.12.4 节“启动一个测试网格节点”。

配置文件格式

测试节点配置文件遵循 JSON 格式，其中定义了一组嵌套的关联映射(Associative Map)。一个关联映射通过大括号({})括起的一个块来定义。一个映射就是以冒号(:)分隔的一个键-值对(key-value pair)。键是双引号("key")括起的一个文字列。值可以是文字列，列表，或者一个嵌套的关联映射。列表是以逗号分隔的多个值，以方括号([])括起。

最顶层的关联映射应该包含两个项目：`capabilities`（值应该是一组关联映射的列表）和 `configuration`（值是一个嵌套的关联映射）。

```
{  
  "capabilities":  
  [  
    {  
      "browserName": "firefox",  
      ...  
    },  
    ...  
  ],  
  "configuration":  
  {  
    "port": 5555,  
    ...  
  }  
}
```

后文将会给出完整的示例。

浏览器能力

浏览器能力通过一组关联映射的列表来定义，并设置为 `capabilities` 项目的值。浏览器能力也可以通过命令行参数 `-browser` 来指定，详情请参见 第 21.12.4 节“启动一个测试网格节点”。

映射内的键如下：

platform
测试节点的操作系统平台：WINDOWS, XP, VISTA, LINUX, 或 MAC.

browserName

浏览器标识符, 可以是以下浏览器之一: android, chrome, firefox, htmlunit, internet explorer, iphone, opera, 或 phantomjs (从 TestBench 3.1 开始可用).

maxInstances

并行测试时, 同时可以打开的浏览器实例的最大数目.

version

浏览器的主版本号.

seleniumProtocol

使用 WebDriver 时应该设置为 WebDriver.

firefox_binary

Firefox 可执行文件的完整路径及文件名. 当你使用 Firefox ESR 版, 并且安装路径不在系统的 PATH 路径中时, 通常会需要设置这个项目.

服务器配置

节点服务的配置以嵌套的关联映射, 并设置为 configuration 项目的值. 配置参数也可以通过节点服务程序的命令行参数来指定, 详情请参见第 21.12.4 节“启动一个测试网格节点”.

一个通常的服务器配置, 请见下例.

配置示例

```
{
  "capabilities": [
    [
      {
        "browserName": "firefox",
        "maxInstances": 5,
        "seleniumProtocol": "WebDriver",
        "version": "10",
        "firefox_binary": "/path/to/firefox10"
      },
      {
        "browserName": "firefox",
        "maxInstances": 5,
        "version": "16",
        "firefox_binary": "/path/to/firefox16"
      },
      {
        "browserName": "chrome",
        "maxInstances": 5,
        "seleniumProtocol": "WebDriver"
      },
      {
        "platform": "WINDOWS",
        "browserName": "internet explorer",
        "maxInstances": 1,
        "seleniumProtocol": "WebDriver"
      }
    ],
    "configuration": {
      "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
```

```

    "maxSession": 5,
    "port": 5555,
    "host": ip,
    "register": true,
    "registerCycle": 5000,
    "hubPort": 4444
}
}

```

21.12.4. 启动一个测试网格节点

TestBench 网格节点监听来自 hub 的调用, 并且可以打开浏览器. 网格节点服务程序包含在 Vaadin TestBench 的 JAR 文件内, 你可以使用以下命令启动它:

```
$ java -jar \
    vaadin-testbench-standalone-4.x.x.jar \
    -role node \
    -hub http://localhost:4444/grid/register
```

测试节点会将自己注册到网格 hub 中. 你需要通过 `-hub` 参数或节点配置文件, 来指定 hub 的地址, 节点配置文件请参见第 21.12.3 节“测试节点的服务配置”.

你可以在同一台主机上同时运行 hub 和网格节点, 前面的例子中我们就是这样做的, 使用的地址是 `localhost`.

浏览器能力

节点上安装的浏览器可以通过命令行参数来指定, 也可以使用 JSON 格式的配置文件, 详情请参见第 21.12.3 节“测试节点的服务配置”.

使用命令行参数时, 可以通过一个或多个 `-browser` 参数来定义浏览器性能. 参数后面跟着的必须是逗号分隔的一组属性-值定义列表, 如下例:

```
-browser "browserName=firefox,version=10,firefox_binary=/path/to/firefox10" \
-browser "browserName=firefox,version=16,firefox_binary=/path/to/firefox16" \
-browser "browserName=chrome,maxInstances=5" \
-browser "browserName=internet explorer,maxInstances=1,platform=WINDOWS"
```

关于配置中的属性项目名, 请参见第 21.12.3 节“测试节点的服务配置”.

浏览器驱动参数

如果使用 Chrome 或 Internet Explorer, 它们的远程驱动的可执行文件必须存在于系统路径内(在 PATH 环境变量中), 或者通过节点服务程序的命令行参数指向:

Internet Explorer
`-Dwebdriver.ie.driver=C:\path\to\IEDriverServer.exe`

Google Chrome
`-Dwebdriver.chrome.driver=/path/to/ChromeDriver`

21.12.5. 移动设备测试

Vaadin TestBench 包含了 iPhone 和 Android 驱动, 使用这些驱动可以在移动设备上进行测试. 测试程序可以运行在真实的设备内, 也可以运行在仿真器/模拟器内.

移动设备上的测试程序类似于其他任何类型的 WebDriver，可以使用 **iPhoneDriver** 或 **AndroidDriver**. Android 驱动假定 hub(android-server) 安装在仿真器中，并被转发到 localhost 的 8080 端口，iPhone 驱动则假定转发到端口 3001. 你也可以使用 **RemoteWebDriver**，并配合 `iphone()` 或 `android()` 的浏览器性能，并明确指定 hub 的 URI.

移动设备的测试环境在 Selenium 文档中有详细介绍，请参见关于 iOS driver 和 AndroidDriver 的部分.

21.13. 测试程序的并行执行

ParallelTest 类提供了一种简单的方式来并行地运行测试程序，可以在本地运行，也可以在测试网格环境中远程运行.

21.13.1. 本地并行测试

测试程序的并行执行通常用在开发阶段，要启用这个功能，你的测试用例类需要继承自 **ParallelTest**，而不是 **TestBenchTestCase**，还需要使用 `@RunLocally` 注解来标记测试用例类.

```
@RunLocally  
public class MyTest extends ParallelTest {  
    @Test  
    ...  
}
```

当你运行测试程序时，TestBench 会启动多个浏览器窗口来并行地运行各个测试.

测试程序的并行执行默认使用 Firefox. 你也可以通过注解的参数的方式指定其他浏览器，浏览器类型定义在 **Browser** 枚举型中：

```
@RunLocally(Browser.CHROME)
```

对于 Chrome 和 IE，你需要安装对应的浏览器驱动程序，详情请参见 第 21.3.5 节“安装浏览器驱动程序”.

21.13.2. 在网格环境中使用多个浏览器执行测试程序

要在多个不同的浏览器中运行测试程序，或者远程运行应用程序，你首先需要安装并启动一个网格 hub 以及一个以上的网格节点，详情请参见 第 21.12 节“在分布式环境中运行测试”. 然后就可以在测试网格中远程运行测试程序了，当然，你也可以在你的开发机器上同时运行网格 hub 和测试节点.

要在网格环境中运行一个测试用例，你只需要对测试用例类使用 `@RunOnHub` 注解. 这个注解的参数是网格 hub 的主机地址，默认主机是 localhost. 你需要在一个使用了 `@BrowserConfiguration` 注解的方法中定义测试程序需要的浏览器能力. 这个方法必须返回 **DesiredCapabilities** 的列表.

```
@RunOnHub("hub.testgrid.mydomain.com")  
public class MyTest extends ParallelTest {  
    @Test  
    ...  
  
    @BrowserConfiguration  
    public List<DesiredCapabilities> getBrowserConfiguration() {  
        List<DesiredCapabilities> browsers =  
            new ArrayList<DesiredCapabilities>();
```

```
// Add all the browsers you want to test
browsers.add(BrowserUtil.firefox());
browsers.add(BrowserUtil.chrome());
browsers.add(BrowserUtil.ie11());

return browsers;
}
}
```

测试运行中实际使用的浏览器，由各测试节点的浏览器能力决定。

如果你有很多测试类，你可以将上例中的配置信息放在一个共同的基类中，基类继承自 **ParallelTest**。

21.14. 无头(Headless)测试

TestBench (3.1 及以上版本) 支持使用 PhantomJS (<http://phantomjs.org>) 进行全功能的无头(headless)测试，PhantomJS 是一个基于 WebKit 的无头浏览器。它对很多种 Web 标准都带有高速的原生支持：JavaScript, DOM 处理, CSS 选择器, JSON, Canvas, 以及 SVG。

使用 PhantomJS 进行无头测试，测试的执行可变快大约 15%，它不必启动图形化的 Web 浏览器，即使在进行屏幕截图测试时也是如此！因此也可以直接在构建服务器上运行全量程的(full-scale)前端功能测试，而不必安装任何 Web 浏览器。

通常来说，最好使用图像化的浏览器来开发测试用例，因为可以直观地看到测试执行过程中发生了什么。一旦测试程序在图形化浏览器上可以正确工作了，你就可以将它移植到 PhantomJS 无头浏览器环境去运行。

21.14.1. 运行无头测试所需要的基本设置

所需要的唯一的设置就是安装 PhantomJS 的可执行文件。请访问 PhantomJS 下载页面，遵照你的操作系统的安装指示，并将 PhantomJS 的可执行文件放在系统路径中。

PhantomJSDriver 依赖项目已经包含在 Vaadin TestBench 中了。

创建一个无头的 **WebDriver** 实例

创建一个 **PhantomJSDriver** 实例，与创建 **FirefoxDriver** 实例一样简单。

```
setDriver(TestBench.createDriver(
    new PhantomJSDriver()));
```

某些测试程序可能会失败，因为 PhantomJS 默认的窗口尺寸很小。比如，当窗口太小时，弹出的页面元素可能出现在屏幕之外，相关的测试可能会因此失败。要让这类测试程序更好地运行，需要指定窗口尺寸，如下：

```
getDriver().manage().window().setSize(
    new Dimension(1024, 768));
```

除此之外，以无头模式运行测试程序，并不需要其他的设置。

21.14.2. 在分布式环境中运行无头测试

在分布式网格中运行 PhantomJS 也很简单。首先，遵照 第 21.14.1 节“运行无头测试所需要的基本设置”中的安装指示，在网格节点内安装 PhantomJS。然后，使用以下命令启动 PhantomJS：

```
phantomjs --webdriver=8080 \
--webdriver-selenium-grid-hub=http://127.0.0.1:4444
```

以上命令将以 WebDriver 模式启动 PhantomJS，并将它注册到网格 hub 中，hub 的运行地址是 127.0.0.1:4444。然后，在测试网格中运行测试程序很简单，只需要在 RemoteWebDriver 构造函数中使用 DesiredCapabilities.phantomjs() 作为参数。

```
setDriver(new RemoteWebDriver(
    DesiredCapabilities.phantomjs()));
```

21.15. 行为驱动开发

行为驱动开发(Behaviour-driven development, 简称 BDD) 是在测试驱动开发模式(Test-driven development, 简称 TDD)基础上发展的一种开发模式，在测试驱动开发模式中，首先需要针对所期待的软件正确行为编写测试程序。BDD 使用一种 通用语言(uniquitous language) 来表达业务目标 - 也就是所要求的行为 - 并对此进行测试，以确保软件完全满足了这些目标。

与敏捷开发模式一样，BDD 过程从收集业务需求开始，业务需求以 用户场景(user stories) 的形式表达。用户称为一个 角色(role)，角色要求一个 功能特性(feature) 来获得一种 利益(benefit)。

用户场景可以通过多个 情节(scenario)来表达，情节负责描述各种期望的行为。情节形式化地表达为以下三个阶段：

- *Given* (假定)我打开了计算器
- *When* (当)我按下计算器按钮
- *Then* (那么)屏幕应该显示出计算结果

在 Java 中，JBehave BDD 框架实现了上面这种形式化表达。TestBench 示例程序包含一个 JBehave 示例，在这个示例中，上例的场景表达为 以下测试用例类：

```
public class CalculatorSteps extends TestBenchTestCase {
    private WebDriver driver;
    private CalculatorPageObject calculator;

    @BeforeScenario
    public void setUpWebDriver() {
        driver = TestBench.createDriver(new FirefoxDriver());
        calculator = PageFactory.initElements(driver,
            CalculatorPageObject.class);
    }

    @AfterScenario
    public void tearDownWebDriver() {
        driver.quit();
    }

    @Given("I have the calculator open")
    public void theCalculatorIsOpen() {
        calculator.open();
    }

    @When("I push $buttons")
    public void enter(String buttons) {
        calculator.enter(buttons);
    }

    @Then("the display should show $result")
    public void displayShows(String result) {
        assertEquals(result, calculator.getResult());
    }
}
```

```
    }  
}
```

上面的示例程序使用了针对应用程序 UI 定义的页面对象，详情请参见 第 21.9.2 节“页面对象模式 (Page Object Pattern)”。

这样的场景出现在一个或多个用户场景中，需要继承 **JUnitStory** 或 **JUnitStories**，然后在子类中设置用户场景。在示例程序中，这个设置工作由 **SimpleCalculation** 类实现。这个类指定了类装载器如何动态查找 Story 类，以及如何报告用户场景的测试结果。

更详细的文档，请访问 JBehave 网站，地址是 jbehave.org。

21.16. 已知的问题

本节将介绍如何处理一部分比较难于使用的功能特性，以及需要一点调整才可以使用的功能特性。

21.16.1. 在 Mac OS X 上运行 Firefox 测试程序

Firefox 的主窗口需要获得焦点，才可以触发聚焦事件。如果别的什么程序干扰了焦点，这一点有时会导致一些问题。比如，**TextField** 的输入提示功能依赖于 JavaScript 的 `onFocus()` 事件，以便在 Field 得到焦点时清除提示信息。

使用 TestBench 的应用程序(或网格节点服务程序)存在对应的 Java 进程，OS X 会认为这些进程带有本地 UI 能力，就像 AWT 或 Swing 一样，但实际上它们并没有这个能力，此时就会发生问题。这个问题会导致焦点离开 Firefox，聚焦到使用 TestBench 的进程上，于是需要获得焦点的那些测试程序就会失败。为了解决这个问题，启动运行测试程序的 JVM 时，你需要使用 `-Djava.awt.headless=true` 参数，来禁用 Java 进程的 UI 能力。

注意，使用 Firefox 调试测试程序时，也会出现同样的问题。因此我们建议，除非必须使用 Firefox，否则在调试测试程序时请使用 Chrome。

索引

符号

@Connect, 421
@PreserveOnRefresh, 88
事件, 61
客户端引擎, 60, **63**
布局, 98
接口, 99
数据模型, 61
数据绑定, 61
文本变更事件, 133–134
服务器端 PUSH, 61
示例, 98
窗口, 98
组件, 60, 98

A

AbstractComponent, 98, 101
AbstractComponentContainer, 98
AbstractField, 98
addContainerFilter(), 302
addNestedContainerProperty(), 298
add-ons
 creating, **435–441**
AJAX, 26, **62**
Alignment, 239–240
And (filter), 303

B

BrowserWindowOpener, 87

C

caption 属性, 101
close(), 89
 UI, 88
closableSessions, 89, 95
compatibility, 254
ComponentState, 421
Component 接口
 description, 102
 enabled, 103
 icon, 104
 locale, 105
 read-only, 108
 style name, 109
 visible, 109
 标题, 101
Component 组件, 99, 101
connector, 420
Container, 294–304

Filterable, 162, 302–304
context menus, 586
CSS, 61, 62, 245–277
 compatibility, 254
 introduction, 247–254
CSS3, 62
CSS selections, 273
CustomComponent, 315–325

D

DefaultUIProvider, 87
description 属性, 102
display (CSS property), 276
DOM, 62
Drag and Drop, 362–370
 Accept Criteria, 365–368

E

Eclipse
 widget development, 422–425
enabled 属性, 103
Equal (filter), 302
extension, 273

F

field, 60
Field, 112–117
Filter (in Container), 302–304

G

getLocale(), 106
Google Web Toolkit, 22, 26, 60, 61, **62**, 203
 themeing, 248
 widgets, **419–445**
Greater (filter), 302
GreaterOrEqual (filter), 302

H

heartbeat, 90
HorizontalSplitPanel, 225–227
HTML, 61
HTML 5, 62
HTML 模板, 61
HTTP, 60

I

icon 属性, 104
IndexedContainer, 302
init(), 87
IPC add-on, 398–403
IsNull (filter), 302
IT Mill Toolkit, 26

J

JavaDoc , 98
 JavaScript , 22 , 61 , 62 , 422
 execute() , 347
 print() , 347–348
 JavaScript integration , 441–445
 JPAContainer , 499–522

L

Less (filter) , 302
 LessOrEqual (filter) , 302
 Liferay
 display descriptor , 395–396
 plugin properties , 396–397
 portlet descriptor , 394–395
 liferay-display.xml , 395–396
 liferay-plugin-package.xml , 396–397
 locale 属性
 in Component , 105
 Log4j , 370
 logout , 89

M

Maven
 using add-ons , 450–452
 使用插件 , 46
 创建工程 , 45–46
 编译 , 46
 memory leak , 371
 MethodProperty , 300

N

nested bean properties , 298–300 , 301
 NestedMethodProperty , 300
 Not (filter) , 303
 Notification
 testing , 586
 Null 值的表现 , 132–133

O

Or (filter) , 303
 Out of Sync , 95
 overflow , 224

P

Page
 setLocation() , 89
 Paintable , 99
 Parking demo , 277
 PDF , 348
 PermGen , 371
 popup windows , 328

portal integration , 383–403
 print() , 347–348
 printing , 347–349
 push , 90

R

read-only 属性 , 108
 redirection , 89 , 90
 responsive 扩展 , 273–277

S

Sass , 61 , 62
 Scrollable , 221 , 224
 scroll bars , 204 , 220–221 , 224
 scrolling , 586
 SCSS , 62
 Serializable , 99
 server push , 90
 servletInitialized() , 87
 session , 86
 closing , 89
 expiration , 80 , 89 , 90
 timeout , 80 , 89
 SessionDestroyListener , 87 , 89
 SessionInitListener , 87
 session-timeout , 95
 setComponentAlignment() , 239–240
 setLocation() , 89
 setNullRepresentation() , 132
 setNullSettingAllowed() , 132
 setVisibleColumns() , 299
 SimpleStringFilter , 302
 Sizeable 接口 , 110
 SLF4J , 370
 SQL , 61
 state object , 421
 static , 371
 style name 属性 , 109
 system messages , 90

T

Table , 155–174 , 302
 TestBenchElement , 583
 TextField , 130–134
 theme , 61 , 62 , 245–277
 ThreadLocal pattern , 375–376
 tooltips , 102
 TouchKit , 523–562
 Parking demo , 277

U

UI

closing , 88
expiration , 88
heartbeat , 90
loading , 87
preserving on refresh , 88
UIProvider , 87
 custom , 88

V

Vaadin 6 Migration
 add-ons , **441**
Vaadin Data Model , 279–304
Vaadin Plugin for Eclipse
 visual editor , 315–325
VaadinRequest , 87
VaadinService , 87
VaadinServlet , 60 , 87
VariableOwner , 99
VerticalSplitPanel , 225–227
visible 属性 , 109

W

widget , 60
widget, definition , 405
widgets , 419
windows
 popup , 328

X

XML , 62
XMLHttpRequest , 62

