BUI源码解析

# 背景

目前公司所有产品线的轻量化正在逐步迈进，而研发人员对WEB端的轻量化框架BUI止步于初浅掌握使用方法的程度上，对其背后的实现机制及原理一知半解。这就导致了如下问题：

* 在进行页面设计时，没有明确的依据论证设计是否能够实现，无法评估设计在实现上的风险；
* 在进行页面实现时，无法评估或者预估实现上的功能漏洞或者性能风险；
* 在脱离官方示例的业务场景中，无法及时有效地完成开发；
* 在开发或者bug修复的过程中遇到问题时，无法准确及时地定位并解决问题。

以上问题都会导致开发效率降低、bug数量增加、可维护性差，引起项目的进度及质量风险难以预计和正确评估。

# 目标

本文档的主要目标是加深研发人员对BUI原理及实现机制的理解，增强研发人员使用BUI进行设计、实现、问题处理的能力，以达到控制页面设计风险、提升开发效率和产品可维护性，最终保障项目进度和质量的目标。

次要目标是通过对BUI源码的解析过程，展示源码解析的方法及步骤，提供源码解析的思路，提升研发人员的自主学习能力。

# 源码解析的思路及方法

## 明确定义和范围

**思路：**

在正式展开源码解析前，首先明确解析对象的**精确**定义和适用范围，不能带着模糊的印象和不明的适用边界对源码进行解析。否则，无法正确理解解析对象的适用场景和设计意图。

**方法：**

通过阅读官方文档及社区资料建立解析对象的初步印象，通过尝试撰写代码对解析对象进行实践性分析，最后总结得出精确的定义和适用范围。

## 确定环境及配置

**思路：**

源码并非独立事物，它的开发和运行都有相应的环境和配置依赖。如果我们不能明确其环境和配置，即使能够明白其设计和实现机制，也无法达到实际应用的目标。

**方法：**

通过阅读官方文档及社区资料确定开发环境（包含开发工具、依赖的其它框架和技术等）、运行环境（比如：浏览器环境、服务器环境等）和配置文件或方法（明确配置文件的位置和文件中各配置项的作用）。

## 明确整体结构

**思路：**

源码的 整体结构代表了其设计思想，体现在类别、模块的划分以及各自的交互上面。要充分理解解析对象的设计意图，必须明确其整体结构。

源码的整体结构分为文件结构和逻辑结构两种类型。文件结构体现了源码的物理划分，逻辑结构体现了逻辑调用链（比如：类图、UML图等）。我们需要从这两方面来对整体结构进行明确分类，并了解其设计思想和意图。

**方法：**

文件结构是其物理结构，直接通过人工查看即可；逻辑结构则需要通过对源码的查看和分析才能得出。

## 分解样例代码

**思路：**

从整体结构不同的分类中选取具备代表性（类库型源码）或者全部（工具或框架型）的样例代码（或文件），对其内部的结构和实现进行详细解析。

**方法：**

针对类库型源码，阅读整体结构下相同分类的所有源码，找到结构最完整或者实现最复杂的源码（文件），作为样例代码进行解析。

针对工具或框架型源码，依据模块划分进行分类解析，根据调用链进行逻辑解析。

## 机制分析

**思路：**

完成了样例代码解析和调用链解析后，就能建立起对源码结构及实现的初步认识来。基于此认识，展开对源码的整体阅读，在了解其功能实现的基础上抽象出其实现机制。

**方法：**

首先，采用广度优先方式，将源码全部阅读一遍，一方面建立整体印象，明白哪些是核心代码，哪些是理解困难的点；另一方面印证之前的初步认识是否正确。

然后，采用深度优先方式，对核心代码或困难代码进行精读，先明确其功能实现，再思考其设计，最后抽象出机制来。

最后，将抽象出的机制和其它源码进行对比印证，以确认其正确性。

# BUI源码解析

## 定义

BUI是一个基于JQUERY、兼容kissy和sea.js的模块化富客户端UI控件库。语法规则遵照ES5标准，不包含ES6及以上版本特性。

它适用企业管理软件、文档库软件等业务复杂的企业级开发场景。

本次解析的BUI源码大版本号为1.0。

## 环境

* + 1. 依赖技术

BUI依赖于jQuery,兼容kissy和sea.js。

其中jQuery是必需项，使用BUI时需要在页面上首先静态引入jQuery，然后再静态引入BUI的js文件。

1. <script src="http://g.tbcdn.cn/fi/bui/jquery-1.8.1.min.js"></script>
2. <script src="http://g.tbcdn.cn/fi/bui/bui-min.js"></script>
3. <script type="text/javascript">
4. BUI.use('bui/calendar',**function**(Calendar){
5. **var** datepicker = **new** Calendar.DatePicker({
6. trigger:'.calendar',
7. autoRender : **true**
8. });
9. });
10. </script>

BUI对kissy的兼容是通过bui/adapter来实现的，在使用kissy时，需要先静态引入kissy，再静态引入adapter，才能正常使用。

1. <script src="http://a.tbcdn.cn/s/kissy/1.3.0/kissy-min.js"></script>
2. <script src="http://g.tbcdn.cn/fi/bui/adapter.js"></script>
3. <script type="text/javascript">
5. KISSY.use('bui/calendar',**function**(S,Calendar){
6. **var** datepicker = **new** Calendar.DatePicker({
7. trigger:'.calendar',
8. autoRender : **true**
9. });
10. });
11. </script>

BUI和sea.js的关系比较复杂。兼容sea.js的用法是，在页面上静态引入sea.js相关库文件，再写好sea.js的配置即可使用。

1. <!-- seajs以及插件 -->
2. <script charset="utf-8" src="https://a.alipayobjects.com/??seajs/1.3.1/sea.js,jquery/jquery/1.7.2/jquery.js"></script>
3. <!-- seajs config 配置 -->
4. <script>
5. seajs.config({
6. vars: {
7. locale: 'zh-cn'
8. },
9. alias : { //2.0后使用paths
10. bui : 'http://g.alicdn.com/bui/bui/1.1.21' //'https://s.tbcdn.cn/g/fi/bui'
11. },
12. map : [
13. [/bui\/(.\*).js/,'bui/$1-min.js'] //['.js', '-min.js'] ,仅bui目录下使用-min.js,
14. ]
15. });
16. seajs.use("jquery/jquery/1.7.2/jquery", **function**(jQuery){
17. window.$ = window.jQuery = jQuery;
18. define('jquery',**function**(){
19. **return** jQuery;
20. });
21. });
22. seajs.use('bui/calendar',**function**(Calendar){
23. **var** datepicker = **new** Calendar.DatePicker({
24. trigger:'.calendar',
25. autoRender : **true**
26. });
27. });
28. </script>

而实际上，在独立使用bui.js时，此文件中会包含sea.js中模块化管理代码的定义。

1. /\*\*
2. \* Sea.js 2.1.1 | seajs.org/LICENSE.md
3. \*/
4. (**function**(global, undefined) {
6. // Avoid conflicting when `sea.js` is loaded multiple times
7. **if** (global.seajs) {
8. **return**
9. }
11. **var** seajs = global.seajs = {
12. // The current version of Sea.js being used
13. version: "2.1.1"
14. }
16. **var** data = seajs.data = {}
17. ... ...
19. // Merge object config such as alias, vars
20. **if** (prev && isObject(prev)) {
21. **for** (**var** k **in** curr) {
22. prev[k] = curr[k]
23. }
24. }
25. **else** {
26. // Concat array config such as map, preload
27. **if** (isArray(prev)) {
28. curr = prev.concat(curr)
29. }
30. // Make sure that `data.base` is an absolute path
31. **else** **if** (key === "base") {
32. (curr.slice(-1) === "/") || (curr += "/")
33. curr = addBase(curr)
34. }
35. // Set config
36. data[key] = curr
37. }
38. }
39. emit("config", configData)
40. **return** seajs
41. }
42. })(this);
43. ... ...
44. BUI.use = seajs.use;

基于以上分析，我们可以看出，BUI实际上依赖于jQuery（官方文档指出）、sea.js和kissy，但对sea.js的依赖关系是以源码中直接嵌入的方式体现的，对kissy的依赖关系是以源码复制及少量修改的方式体现的。在模块的整体定义加载配置上面，BUI完全是使用的sea.js的模块化功能；而对模块内的属性、方法定义是根据kissy来的。换句话说，我们在实际开发中直接使用bui.js即可。在进行模块定义及调用时，需参照sea.js的语法；而模块的内部定义及属性，则参照kissy的语法。

* + 1. 开发环境

从官方文档以及源码目录中可以看出，BUI的源码是分散开发、集中构建的形式。它起初使用ant进行自动化构建，后来换为gulp来进行自动化构建。

* + - 1. 构建过程

1. **ant**

ant构建时读取源码根目录下build.xml，在其中首先读取各控件目录下的build.xml，进行js和css控件的独立构建：js控件构建时会将控件目录下的js模块文件进行合并，将合并后的js控件文件拷贝到目标文件夹（build）；css构建时会调用assets/css目录下的build.xml，使用less生成css文件，使用YUICompressor 压缩 css文件。

然后ant会将目标文件夹中所有的js控件文件合并为bui.js文件；然后使用uglify 压缩目标文件夹中所有的js文件（包括js控件文件和bui.js文件，生成\*-min.js文件）；最后将资源文件（css、图片等）拷贝到目标文件夹（build）。

1. **gulp**

gulp的构建过程和ant类似，读取源码根目录下gulpfile.js，在其中首先读取各控件目录下的gulpfile.js，进行控件中js模块文件的合并（目前BUI 1.0版本只有uploader控件使用了gulp来进行合并），并将合并后的js文件拷贝到目标文件夹（build）；然后在目标文件夹中将模块js文件合并为bui.js文件，然后使用uglify压缩目标文件夹中所有的js文件（包括模块js文件和bui.js文件，生成\*-min.js文件）；然后使用minify进行css文件的压缩。

由于bs3中的less样式只能采用gulp进行构建，在此简单讲述一下如何使用gulp：

1、首先，你的电脑应具备nodejs（>= v10.15.2）及npm（>= v6.4.1）。

2、全局安装gulp，本文发布时，gulp的对应版本为4.0.2。在命令行中输入并执行：

npm install --global gulp

3、卸载bui本地的gulp，并重新安装。在命令行中进入bui源码根目录，输入并执行：

npm rm gulp --save-dev

npm i gulp --save-dev

4、进行/bui/assets/css/bs3目录，从enovell分支上将按4.0版本修改过的gulpfile更新到本地：

git pull

5、将bs3中的less编译为css文件并压缩，然后把生成的css及压缩文件拷贝到build目录中去。在命令行中进入/bui/assets/css/bs3目录，输入并执行：

gulp

至此，就完成了使用gulp构建bs3中的less文件的操作。

从上述两种构建过程中我们可以发现，BUI的开发首先是分控件开发，然后在控件中再分模块开发，源码中每个js文件对应一个具体的模块，构建过程中会先将控件目录中所有的js模块文件合并为一个js控件文件，再将所有的js控件文件合并为统一的bui.js文件。

* + - 1. 其它工具

BUI使用了jasmine进行单元测试，所有的测试用例和测试数据在/test目录下，采用php页面来进行测试。

BUI使用了jsduck来生成文档（DPL，Design Pattern Library），调用方法为/bui/tools/jsduck/run.bat，执行完成后生成的文档在/bui/docs下面。jsduck能够将js源文件中对应的注释及注解提取为一份API文档说明。

* + 1. 运行环境

BUI运行在客户的浏览器上，浏览器的兼容性如下：

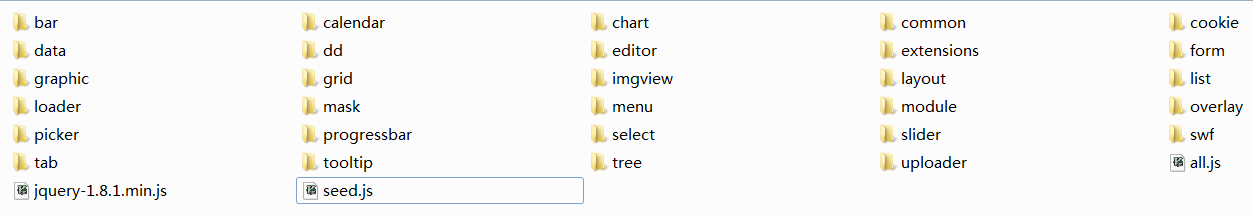
ie6(大部分兼容)及以上版本ie浏览器；firefox；chrome；360等。

## 结构

我们从BUI源码的文件结构、逻辑结构、调用链和实现机制四个方面来解析它的结构。

* + 1. 文件结构

BUI源码（src）的文件整体结构如下：



基本上它的每个目录代表一个命名空间，目录下的每个文件代表一个模块。而jquery-1.8.1.min.js是所有控件依赖的文件，all.js是使用所有控件的声明，seed.js是通过kissy使用BUI时的入口文件。

* + 1. 逻辑结构

BUI源码从逻辑上可以划分为两大类：框架类、模块类。

框架类定义了框架提供的API和功能，简单来说，就是直接定义在BUI对象上的方法和属性。

模块类定义了模块的基类和各个具体的模块。在构建时，会将多个模块组合在一起形成控件，但在源码层级上，并没有独立的控件源码。每个模块都是按照面向对象思想设计，具备其独立的职责。

BUI的源码需要通过构建工具进行构建，则我们以构建过程的顺序来分析其逻辑结构。

* + - 1. 框架类

框架类可分为定义类和工具类两种。

定义类是模块定义、模块使用等在BUI实际开发过程中必然会用到的相应接口。

工具类是在BUI实际开发过程中根据业务需求选用的接口。

* + - * 1. 定义类

定义类中包含模块的定义、配置、加载及基础函数的定义还有事件的定义等，主要集中在loader和common目录下。

src/loader目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| sea.js | 无 | 定义了模块的定义（define）、加载（use）、配置（config）等模块管理的函数，是模块化开发的基础。 |
| seajs-cfg.js | /loader/  sea.js | 定义了模块管理的默认配置项，包括默认字符集为UTF-8、在引入bui.js时script标签上加上data-debug=true可开启debug模式、开启debug模式时根据模块id加载\*.js而不是\*-min.js等 |

src/common目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| common.js | /common/\* | 将common目录下的框架及工具类挂载到BUI上；如果使用kissy加载，将adapter也挂载到BUI上。common控件的入口文件 |
| util.js | 无 | bui基础函数定义（extend、mixin、augment、merge、each、log等），类型判断函数定义（isString、isNumber、isBoolean等），表格帮助类FormHelper。从函数的用途上来讲，util.js既是定义类，也属于工具类。此处收录，工具类中就不再赘述。 |
| observable.js | /common/  util.js | 通过观察者模式来实现事件，包括事件的添加、删除、绑定、回调、冒泡等 |

* + - * 1. 工具类

工具类包含日期处理、JSON、kissy适配、浏览器检测等等，主要集中在common目录中。

src目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| all.js | bui.js | 在引入bui.js时script标签上加上data-auto-use=true，则会自动加载所有业务模块。 |
| seed.js | /common/\*,  /cookie/\* | 用于只包含核心的按需加载模式(AMD) |

src/common目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| ua.js | 无 | 浏览器版本检测工具 |
| keycode.js | 无 | 键盘值常量类 |
| date.js | 无 | 日期工具类，包括日期格式化、加法、当前日期、判断字符串是否有效日期等 |
| adapter.js | 无 | 兼容kissy 和 jQuery 的适配器 |
| array.js | /common/  util.js | 数组帮助类，提供遍历、定位、判断数组是否相等、过滤、判断是否为空等 |
| json.js | /common/  ua.js | JSON格式化 |

src/cookie目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| cookie.js | 无 | cookie工具类 |

* + - 1. 模块类

模块类可分为模块基类和功能模块。

模块基类为功能模块提供了公用属性和函数的定义。

功能模块通过继承和组合实现具体的功能。

* + - * 1. 模块基类及扩展类

src/common目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /common/ observable.js | bui中ui控件的最基础类，提供设置获取属性、事件支持、属性变化触发相应的事件、将配置项自动转化为属性 |
| component.js | /common/component/\* | Component命名空间的入口文件，提供了根据Xclass创建对象的方法 |

src/common/component目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| manage.js | 无 | 模块管理器，负责维护模块在管理器上的注册、移除、遍历、获取等，管理模块的元数据 |
| loader.js | /common/util.js  /common/base.js | 模块加载器定义 |
| uibase.js | /common/component/uibase/\* | uibase的入口文件，引入uibase目录下所有模块。 |
| view.js | /common/component/manage.js  /common/component/uibase.js | 模块的视图基类。定义了所有视图的共用属性，包括样式、根节点、内容、可视性等等。  继承自UIBase.Tpl.View |
| controller.js | /common/component/manage.js  /common/component/uibase.js  /common/component/view.js  /common/component/loader.js | 模块可实例化的基类，定义了所有模块的共用属性和函数，包括样式、可视性、容器、事件处理等。  继承自  UIBase.Decorate,UIBase.Tpl,UIBase.ChildCfg,UIBase.KeyNav,UIBase.Depends |

src/common/component/uibase目录：

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /common/component/manage.js, /common/base.js | 可视化模块的基类，管理了在继承机制下可视化模块的生命周期以及公用属性和方法。  继承自BUI.Base |
| align.js | /common/ua.js | 和指定元素对齐方式扩展类 |
| autohide.js | 无 | 点击或移出控件外部，控件隐藏扩展类 |
| autoshow.js | 无 | click，focus,hover等引起控件显示，并提供定位的扩展类 |
| bindable.js | 无 | 使控件绑定store，处理store事件的扩展类 |
| childcfg.js | 无 | 子控件的默认配置项扩展类 |
| close.js | 无 | 关闭或隐藏控件扩展类 |
| collapsable.js | 无 | 控件展开折叠的扩展类 |
| decorate.js | /common/array.js,  /common/json.js,  /common/component/manage.js | 装饰类，用于将DOM对象封装成控件 |
| depends.js | /commmon/component/manage.js | 控件依赖事件源的扩展类 |
| drag.js | 无 | 拖拽控件的扩展类 |
| keynav.js | /common/keycode.js | 键盘导航的封装扩展类 |
| list.js | /common/component/uibase/selection.js | 列表一类的控件的基类，simpleList,menu,grid都是从此类继承 |
| listitem.js | 无 | 列表项的定义 |
| mask.js | 无 | 遮罩层扩展类 |
| position.js | 无 | 控件定位扩展类 |
| selection.js | 无 | 可选择控件中的选择项扩展类，选择项可为子控件或DOM元素；  抽象类； |
| stdmod.js | 无 | 上中下布局容器扩展类，header、body、foot |
| tpl.js | 无 | 控件模板扩展类的渲染类(view) |

* + - * 1. 功能模块

功能模块基本上都会对模块基类产生依赖，因此，下方列表中的依赖一栏将不会注明对基类的依赖关系。

src中的chart（图表）目录不做说明，因为我们使用echarts作为图表工具，不会使用此功能。

src中的dd（拖拽）目录不做说明，因为里面仅有空文件，并无具体实现。

src中的graphic（图形化）目录不做说明，因为里面内容是针对h4的canvas操作进行的封装，支持svg和vml，一方面这属于落后技术，另一方面我们的图形绘制要么采用qunee，要么采用zrender，也不会使用。但需要注意的是，imgview对graphic有依赖，构建时仍然不能落下它。

src中的swf（swf操作）目录不做说明，因为我们基本不会操作swf了。

另外，如果某个目录中的js文件为空文件，也不会进行说明。

src/bar目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| bar.js |  | 工具栏的基类，定义了工具栏的容器（ul），定义了工具栏上子项的默认样式（bar-item） |
| barItem.js |  | 工具栏的子项，扩展了ListItem的功能，定义了子项的容器（li），定义了子项容器样式默认内联 |
| base.js | /toolbar/\* | 工具栏命名空间入口，将本目录内所有模块整合为一个模块对外发布 |
| numberpagingbar.js | /toolbar/paginbar.js | 数字分页栏，继承自分页栏。覆写了分页栏的按钮，以a标签包裹的数字替换，定义了默认的省略号模板 |
| pagingbar.js | /toolbar/bar.js | 分页栏，继承自工具栏，扩展了数据绑定功能。定义了首页、上一页、下一页、末页、跳转某页、总条数功能 |

src/calendar目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /calendar/\* | 日历命名空间入口，将本目录内所有模块整合为一个模块对外发布 |
| calendar.js | /picker/\*,/toolbar/\*,/calendar/monthpicker.js,/calendar/header.js,/calendar/panel.js | 日期控件，由头部（header）、面板（panel）和底部按钮栏组成，并支持时间选择。 |
| datepicker.js | /picker/\*,/calendar/ calendar.js | 日期选择器，可由输入框等触发。继承自Picker，内部生成Calendar，触发后悬浮显示。 |
| header.js |  | 日期控件的头部，用来选择年月。定义了头部的模板及样式，年份选择及上月下月的交互响应。 |
| monthpicker.js | /list/simplelist.js,/toolbar/\*,/overlay/overlay.js | 独立的年月选择器，内部定义了月份面板、年份面板和月份选择器。面板继承自List，选择器继承自Picker。 |
| panel.js |  | 日期控件的面板，显示日期的容器。定义了周模板、日模板，定义了选择日期的交互响应。 |

src/data目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| abstractstore.js | /data/proxy.js | 抽象数据缓冲类，定义了数据缓冲类的基本属性和方法声明，定义了自动加载数据行为辨别、查询参数初始化、数据代理类初始化、加载数据行为、加载数据的回调、触发过滤行为、数据过滤行为、数据错误辨识、处理错误函数，以这些定义构成了数据缓冲类的内部机制。 |
| base.js | /data/\* | 命名空间入口，将本目录内所有模块整合为一个模块对外发布。跟之上的入口不同，此处采用了匿名函数即时执行的方式来发布，而并未采用seajs的模块定义机制。 |
| node.js |  | 树形数据结构的节点类，无法直接使用数据作为节点，所以进行一层封装，可以直接作为TreeNode控件的配置项。  这是一个最简单的非展示类型模块，由于其功能单一，直接采用了一个js对象作为模块，扩展了属性就对外发布了。 |
| proxy.js | /data/sortable.js | 数据代理对象，加载数据, 一般不直接使用，在store里面决定使用什么类型的数据代理对象。  定义了ajax代理和内存代理两种方式。  这是一个典型的非展示类型模块，采用了BUI的继承、定义常量、定义属性、定义参数机制。 |
| sortable.js |  | 数据排序扩展类，定义了排序字段、排序方法以及正序倒序的选择。  注意，这不是工具类，而是扩展组件类，仅为继承了BUI.Base的类提供扩展功能。 |
| store.js | /data/sortable.js，/data/proxy.js，/data/ abstractstore.js | 数据缓冲组件，用于加载远程或本地数据。定义了加载数据的格式、针对缓存数据的CRUD方法，并且支持数据的双向绑定。 |
| treestore.js | /data/node.js，/data/proxy.js，/data/ abstractstore.js | 树形对象数据缓冲类，定义了加载数据的树形结构，支持本地和远程两种方式。 |

src/editor目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /editor/\* | 命名空间入口，将本目录内所有模块整合为一个模块对外发布。 |
| dialog.js | /overlay/dialog.js,/editor/mixin.js | 弹出框编辑器，继承自dialog |
| editor.js | /overlay/ overlay.js,/editor/mixin.js | 弹出层编辑器，继承自overlay，通过输出框等触发弹出 |
| mixin.js |  | 编辑器扩展类，引入这个扩展，控件可以支持编辑器功能 |
| record.js | /editor/editor.js | 对象编辑器，继承自editor |

src/extensions目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| multiselect/multilist.js | /list/listbox.js,/list/simplelist.js | 穿梭框，左右选择 |
| multiselect/multilistpicker.js | /overlay/dialog.js,/picker/mixin.js | 多选弹出框 |
| multiselect/multiselect.js | /extensions/multiselect/multilist, /extensions/multiselect/multilistpicker, /extensions/search | 输入框触发弹出穿梭框 |
| search.js | /form/group.js | 搜索组件，由一个输入框带确定按钮组成 |
| treegrid.js | /grid/gird.js,/tree/mixin.js | 树形列表 |
| treepicker.js | /picker/listpicker.js,/tree/\* | 多选或单选树型弹出框 |

src/form目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| field/base.js |  | 表单字段基类，定义了错误信息、内部控件的容器、验证机制、数据转换等 |
| field/check.js | /form/field/base.js | 可勾选字段，提供单选及多选能力 |
| field/checkbox.js | /form/field/check.js | 复选框表单域 |
| field/checklist.js | /form/field/list.js | 可勾选的列表，模拟多个checkbox |
| field/combox.js |  | 空的，等我们自己完善吧 |
| field/date.js | /form/field/base.js, /calendar/datepicker.js | 表单日历域 |
| field/hidden.js | /form/field/base.js | 隐藏字段 |
| field/list.js | /form/field/base.js,  /list/simplelist.js | 表单中的列表，每个列表后有个隐藏域用来存储数据 |
| field/number.js | /form/field/base.js | 表单数字域 |
| field/plain.js | /form/field/base.js | 仅仅用于显示文本，不能编辑的字段 |
| field/radio.js | /form/field/check.js | 单选框表单域 |
| field/radiolist.js | /form/field/list.js | 可勾选的列表，模拟多个radio |
| field/readonly.js | /form/field/base.js | 只读字段 |
| field/select.js | /form/field/base.js | 模拟选择框在表单中 |
| field/text.js | /form/field/base.js | 表单文本域 |
| field/textarea.js | /form/field/base.js | 表单多行文本域 |
| field/uploader.js | /form/field/base.js | 表单上传域 |
| group/base.js | /form/fieldcontainer.js | 表单文本域组，可以包含一个至多个字段 |
| group/check.js | /form/group/base.js | 选择分组，包含:checkbox,radio。能够配置选择的个数 |
| group/range.js | /form/group/base.js | 范围的字段组，比如日期范围、数字范围等场景 |
| group/select.js | /form/group/base.js | 级联选择框分组 |
| base.js | /form/\* | 命名空间入口，将本目录内所有模块整合为一个模块对外发布 |
| field.js | /form/field/\* | 表单域的入口文件 |
| fieldcontainer.js | /form/ field.js,/form/ groupvalid.js | 表单字段的容器扩展 |
| fieldgroup.js | /form/group/\* | 表单组的入口文件 |
| form.js | /form/ fieldcontainer.js | 表单控件 |
| groupvalid.js | /form/valid.js | 表单分组验证 |
| hform.js | /form/form.js | 垂直表单 |
| remote.js |  | 表单异步请求，异步校验、远程获取数据 |
| row.js | /form/ fieldcontainer.js | 表单里的一行元素 |
| rule.js |  | 验证规则 |
| rules.js | /form/rule.js | 验证集合 |
| tips.js | /overlay/overlay.js | 输入提示信息 |
| valid.js | /form/rules.js | 表单验证 |

src/grid目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| plugins/autofit.js |  | 自动适应表格宽度的扩展 |
| plugins/base.js | /grid/plugins/\* | 表格插件的入口 |
| plugins/cascade.js |  | 级联表格 |
| plugins/cellediting.js | /grid/plugins/ editing.js | 表格单元格编辑 |
| plugins/columngroup.js |  | 表头列分组功能 |
| plugins/columnresize.js |  | 拖拽改变列的宽度 |
| plugins/dialog.js | /editor/dialog.js | 表格跟表单联用。配置中支持是否自动保存数据到数据源，通过store的save方法实现 |
| plugins/editing.js |  | 表格编辑插件 |
| plugins/gridmenu.js | /menu/contextmenu.js | 表格菜单插件 |
| plugins/rowediting.js | /grid/plugins/editing.js | 表格行编辑 |
| plugins/rowgroup.js |  | 表头列分组功能，仅处理数据展示，排序，不处理这个过程中的增删改，添加删除列 |
| plugins/rownumber.js |  | 表格显示行序号的插件 |
| plugins/selection.js |  | 选择的插件，包含单选和多选行 |
| plugins/summary.js |  | 表格数据汇总 |
| base.js | /grid/\* | 表格命名空间入口 |
| column.js |  | 表格的列对象，存储列信息，此对象不会由用户创建，而是配置在Grid中 |
| grid.js | /mask/loadmask.js,/toolbar/bar.js,/list/simplelist.js,/grid/header.js,/grid/column.js | 表格主体 |
| header.js | /grid/column.js | 表格的头部 |
| simplegrid.js | /list/simplelist.js | 简单表格,仅用于展示数据 |
| util.js |  | 表格工具类，包含：  日期格式化函数，表格中用于渲染列，输出yyyy-MM-dd;  日期时间格式化函数，表格中用于渲染列，输出yyyy-MM-dd hh:mm:ss;  文本截取函数，当文本超出一定数字时，会截取文本，添加...;  枚举格式化函数，接收键值对的枚举对象，返回指定枚举的格式化函数；  将多个值转换成一个字符串，接收键值对的枚举对象，返回指定枚举的格式化函数；  将财务数据分转换成元，接收键值对的枚举对象，返回将分转换成元的数字； |

src/imgview目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /imgview/\* | 命名空间入口 |
| imgview.js | /imgview/viewcontent.js, /imgview/previewlist.js,/toolbar/bar.js | 自带小图预览及动作条的图片预览 |
| previewlist.js | /list/simplelist.js | 图片预览控件 |
| viewcontent.js | /graphic/canvas.js | 单纯的图片展示控件,兼容IE6,支持旋转、拖动、缩放 |

src/layout目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| item/absolute.js | /layout/ item/base.js | 绝对布局的布局项 |
| item/anchor.js | /layout/ item/base.js | 锚定容器的布局项 |
| item/base.js |  | 布局插件的子项，用于封装控件，控制控件的位置 |
| item/border.js | /layout/ item/base.js | 边框布局选项 |
| item/cell.js | /layout/ item/base.js | 表格布局的项，是一个个的单元格 |
| item/tab.js | /layout/ item/base.js | 可以收缩的选项 |
| absolute.js | /layout/abstract.js, /layout/item/absolute.js | 绝对位置布局 |
| abstract.js | /layout/item/ base.js | 布局控件的基类。控件布局插件的抽象类，此插件不要直接使用 |
| accordion.js | /layout/abstract.js, /layout/item/tab.js,  /layout/collapsable.js | 可折叠的布局，只能展开一个选项内容 |
| anchor.js | /layout/abstract.js, /layout/item/anchor.js | 锚定布局，根据容器的边缘自动计算宽高 |
| base.js | /layout/\* | 布局模块的入口文件 |
| border.js | /layout/abstract.js, /layout/item/border.js,  /layout/collapsable.js | 经典的边框布局,上中下，中间分为左中右 |
| collapsable.js |  | 布局内部存在可折叠的项 |
| columns.js | /layout/abstract.js | 列模式布局 |
| flow.js | /layout/abstract.js, /layout/item/base.js | 浮动布局，所有的元素float:left |
| table.js | /layout/abstract.js, /layout/item/cell.js | 表格布局 |
| viewport.js |  | 全屏容器，一般情况下用于布局 |

src/list目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /list/\* | 列表模块入口文件 |
| domlist.js |  | 使用DOM元素作为选项的扩展类 |
| keynav.js |  | 列表导航扩展类 |
| list.js |  | 列表 |
| listbox.js | /list/simplelist.js | 列表选择框 |
| listitem.js |  | 列表子项 |
| simplelist.js | /list/domlist.js, /list/keynav.js,/list/sortable.js | 简单列表，直接使用DOM作为列表项 |
| sortable.js | /data/sortable.js | 列表排序的扩展 |

src/mask目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /mask/\* | Mask的入口文件 |
| loadMask.js | /mask/mask.js | 加载数据时屏蔽层 |
| mask.js |  | 屏蔽层，能够屏蔽指定元素和解除屏蔽 |

src/menu目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /menu/\* | 菜单命名空间入口文件 |
| contextmenu.js | /menu/menuitem.js,/menu/popmenu.js | 弹出菜单，一般用于右键菜单 |
| menu.js |  | 菜单基类，定义了相应事件绑定 |
| menuitem.js |  | 菜单子项 |
| popmenu.js |  | 下拉菜单，一般用于下拉显示菜单 |
| sidemenu.js |  | 侧边栏菜单 |

src/overlay目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /overlay/\* | 命名空间入口 |
| dialog.js | /overlay/overlay.js | 弹出框 |
| message.js | /overlay/dialog.js | 消息框，警告、确认；构建时会将它直接挂载在BUI.Message命名空间上，如BUI.Message.Alert |
| overlay.js |  | 悬浮层，显示悬浮信息，Message、Dialog的基类 |

src/picker目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /picker/\* | 命名空间入口 |
| listpicker.js | /picker/picker.js | 列表项的选择器 |
| mixin.js |  | picker的扩展 |
| picker.js | /overlay/overlay.js | 选择器控件的基类，弹出一个层来选择数据，不要使用此类创建控件，仅用于继承实现控件 |

src/progressbar目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /progressbar/\* | 进度条命名空间入口 |
| loadprogressbar.js | /progressbar/progressbar.js | 异步加载进度条 |
| progressbar.js |  | 基础进度条，用于显示进度 |

src/select目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /select/\* | 选择框命名空间入口 |
| combox.js | /select/select.js, /select/tag.js | 组合框，用于选择输入文本 |
| select.js | /picker/listpicker.js | 选择控件 |
| suggest.js | /select/combox.js | 组合框 用于提示输入 |
| tag.js | /list/simplelist.js | 输入、选择完毕后显示tag |

src/slider目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /slider/slider.js | 命名空间入口 |
| keynav.js |  | 滑块的键盘导航 |
| scroller.js | /slider/slider.js | 模拟滚动条 |
| slider.js |  | 滑块 |

src/tab目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /tab/\* | 命名空间入口 |
| navtab.js | /menu/contentmenu.js | 导航标签，包含导航内容的容器 |
| navtabitem.js |  | 导航标签项 |
| panelitem.js |  | 拥有内容的标签项的扩展类，每个标签项都有一个分离的容器作为面板 |
| panels.js |  | 拥有多个面板的容器 |
| tab.js |  | 切换标签 |
| tabitem.js |  | 切换标签项 |
| tabpanel.js | /tab/tab.js,/tab/panels.js | 带有面板的切换标签 |
| tabpanelitem.js | /tab/tabitem.js,/tab/panelitem.js | 存在面板的标签项 |

src/tooltip目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /tooltip/\* | 提示的入口 |
| tip.js | /overlay/overlay.js | 简单易用的提示信息，定义触发元素、默认方向、样式、模板等配置项 |
| tips.js | /tooltip/tip.js | 批量显示提示信息 |

src/tree目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| base.js | /tree/\* | 命名空间入口 |
| selection.js | /list/simplelist.js | 树的选中，跟列表的选中有所差异 |
| treelist.js | /list/simplelist.js,/tree/treemixin.js,/tree/selection.js | 树形列表控件 |
| treemenu.js | /list/simplelist.js,/tree/treemixin.js,/tree/selection.js | 树形菜单 |
| treemixin.js | /data/treestore.js | 树形扩展，基于list扩展，可以组合出tree list,tree grid ,tree menu |

src/uploader目录

|  |  |  |
| --- | --- | --- |
| **名称** | **依赖** | **说明** |
| button/ajbridge.js | /swf/swf.js | swf桥接组件，用于调用swf接口 |
| button/base.js | /uploader/button/filter.js | 文件上传按钮的基类 |
| button/filter.js |  | 上传文件类型过滤器，默认支持图片类型（gif、png、jpg、bmp）和office部分类型（doc、docx、xls、xlsx）； |
| button/htmlButton.js | /uploader/button/base.js,/uploader/file.js | 文件上传按钮, ajax和iframe上传方式使用,使用的是input[type=file] |
| button/swfButton.js | /uploader/button/ajbridge.js,/uploader/button/base.js,/uploader/file.js | flash上传按钮 |
| type/ajax.js |  | ajax方案上传 |
| type/base.js | /uploader/type/base.js | 上传方式类的基类，定义通用的事件和方法，一般不直接监听此类的事件 |
| type/flash.js | /uploader/type/base.js | flash上传方案 |
| type/iframe.js | /uploader/type/base.js | iframe方案上传，全浏览器支持 |

* + 1. 调用链

BUI的调用链是使用BUI.use加载组件或模块实现业务时的调用过程，所以首先我们来看BUI.use进行了怎样的操作。

* + - 1. BUI.use

从之前的逻辑结构分析中，我们了解到，BUI.use实际上是sea.use，它的语法是：

1. BUI.use(id, callback?)

其中的参数id是指模块定义时的模块id，callback则是一个匿名函数，其参数是第一个参数对应的模块。同时第一个参数也可以是一个模块id数组，而callback匿名函数的形参则按顺序对应模块id数组。如下所示：

1. BUI.use(['bui/grid','bui/data','bui/overlay','bui/calendar'],**function**(Grid,Data,Overlay,Calendar){
2. ... ...
3. }

在callback函数中，我们就可以对use的模块进行实例化、属性设置、事件监听及渲染等操作来进行业务开发。那么use是通过怎样的机制加载模块的呢？

从之前的逻辑结构分析中，我们发现use的定义在src/loader/sea.js文件中：

1. seajs.use = **function**(ids, callback) {
2. Module.preload(function() {
3. Module.use(ids, callback, data.cwd + "\_use\_" + cid())
4. })
5. return seajs
6. }

这里面做了两件事，第一步在加载模块之前，委托调用了Module的preload进行所需的预加载：

1. // Modules that are needed to load before all other modules
2. data.preload = (**function**() {
3. **var** plugins = []
5. // Convert `seajs-xxx` to `seajs-xxx=1`
6. // NOTE: use `seajs-xxx=1` flag in uri or cookie to preload `seajs-xxx`
7. **var** str = loc.search.replace(/(seajs-\w+)(&|$)/g, "$1=1$2")
9. // Add cookie string
10. str += " " + doc.cookie
12. // Exclude seajs-xxx=0
13. str.replace(/(seajs-\w+)=1/g, **function**(m, name) {
14. plugins.push(name)
15. })
17. **return** plugins
18. **})()**

预加载是检测uri或cookie中是否存在对seajs的请求（seajs-xxx=1形式），如果存在，则加载对应的seajs版本。

第二步再委托调用了Module的use函数来加载模块：

1. // Use function is equal to load a anonymous module
2. Module.use = **function** (ids, callback, uri) {
3. **var** mod = Module.get(uri, isArray(ids) ? ids : [ids])
5. mod.callback = **function**() {
6. **var** exports = []
7. **var** uris = mod.resolve()
9. **for** (**var** i = 0, len = uris.length; i < len; i++) {
10. exports[i] = cachedMods[uris[i]].exec()
11. }
13. **if** (callback) {
14. callback.apply(global, exports)
15. }
17. **delete** mod.callback
18. }
20. mod.load()
21. }

use函数中包含三部分，第一部分是采用Module.get根据模块的id获取对应的模块实例（如果在缓存中没有则新建一个实例并放入缓存）；第二部分是模块的回调函数注册，在回调函数中先通过mod.resolve获取模块的路径（uri），然后根据路径找到缓存中对应的模块并执行cachedMods[uris[i]].exec()，最后将使用模块的回调函数注册到全局作用域中callback.apply(global, exports)以备调用；最后加载模块mod.load()。

从上面三部分的过程来看，我们发现Module执行存在三个关键点，一是Module的生命周期管理，包括新建、缓存、加载、执行等；二是Module的加载细节，方法是load；三是Module的执行细节，方法是exec。接下来我们详细对这三个关键点进行分析。

* + - 1. Module的生命周期管理

在Module的源码中，它的生命周期被分为6个状态：

* 抓取中（1-FETCHING）：抓取到Module的uri;
* 已保存（2-SAVED）：将Module的元数据保存在缓存（cachedMods）中；
* 加载中（3-LOADING）：Module的依赖项加载完成；
* 加载完成（4-LOADED）：Module加载完成，等待执行；
* 执行中（5-EXECUTING）：Module执行中；
* 执行完成（6-EXECUTED）：Module的export已经就绪。

此外，除了上述6种生命周期状态外，Module还具备错误（7-ERROR）状态。

从上面的6个状态，我们可以看出模块的整个生命周期过程，结合use函数的调用链，我们跟踪一下Module的状态变化情况。

* + - 1. Module的加载

module的加载主要由Module的load方法实现，具体分为三个步骤：一是解析module并将其依赖模块置入等待队列中：

1. **var** mod = **this**
3. // If the module is being loaded, just wait it onload call
4. **if** (mod.status >= STATUS.LOADING) {
5. **return**
6. }
8. mod.status = STATUS.LOADING
10. // Emit `load` event for plugins such as combo plugin
11. **var** uris = mod.resolve()
12. emit("load", uris)
14. **var** len = mod.\_remain = uris.length
15. **var** m
17. // Initialize modules and register waitings
18. **for** (**var** i = 0; i < len; i++) {
19. m = Module.get(uris[i])
21. **if** (m.status < STATUS.LOADED) {
22. // Maybe duplicate
23. m.\_waitings[mod.uri] = (m.\_waitings[mod.uri] || 0) + 1
24. }
25. **else** {
26. mod.\_remain--
27. }
28. }
30. **if** (mod.\_remain === 0) {
31. mod.onload()
32. **return**
33. }

二是抓取module的所有依赖模块生成请求函数并将其置入请求缓存中，注意，此时请求函数并未调用：

1. // Begin parallel loading
2. var requestCache = {}
4. **for** (i = 0; i < len; i++) {
5. m = cachedMods[uris[i]]
7. **if** (m.status < STATUS.FETCHING) {
8. m.fetch(requestCache)
9. }
10. **else** **if** (m.status === STATUS.SAVED) {
11. m.load()
12. }
13. }

三是针对所有抓取上来的模块发起请求：

1. // Send all requests at last to avoid cache bug in IE6-9. Issues#808
2. for (var requestUri in requestCache) {
3. **if** (requestCache.hasOwnProperty(requestUri)) {
4. requestCache[requestUri]()
5. }
6. }

module的状态变化如下：在第一步解析并将依赖模块置入等待队列完成时，module的状态是加载中（3-LOADING）；第二步中，module本身的状态没有变化，而module的依赖模块的状态变成抓取中（1-FETCHING）；第三步中，针对依赖模块发起请求时，依赖模块的状态变成已保存（2-SAVED）。

当所有依赖模块都已保存，module调用onload方法，首先将自身的状态变成加载完成（4-LOADED）；然后调用依赖模块的onload方法，将依赖模块的状态变成加载完成（4-LOADED）。

至此，模块的加载全部完成，接下来进行module的执行。

* + - 1. Module的执行

module的执行分为两个侧面进行分析，一是module本身及其依赖模块的执行顺序及状态变化，二是module内部的执行顺序。

module本身及其依赖模块的执行顺序及状态变化如下：

首先，将module自身的状态变成执行中（5-EXECUTING）；然后，生成require函数；最后，执行module的factory函数生成module类，并将module自身的状态改为执行完成（6-EXECUTED）。

其中需要注意的是，require函数中封装了依赖模块的执行，当require函数在factory函数实际执行过程中被调用时，会执行对应的依赖模块。

附注：此处的require函数和factory函数是指module定义语法define(moduleId,[dependencies],factory(require))其中的factory和require；require在此处定义并传入factory，factory的内容由module定义时具体指定。

下面以UIBase为例讲述下module在执行过程中内部的具体生命周期和执行顺序。

* + - * 1. UIBase对可视化模块的生命周期管理

UIBase中定义了可视化模块的抽象方法和执行顺序。

可视化模块的抽象函数定义按执行顺序排列如下：initializer、createDom、renderUI、bindUI和syncUI。比如：Controller模块在实例化时，首先执行initializer函数，然后依次执行createDom、renderUI、bindUI和syncUI函数（实际上Controller模块并未实现syncUI函数，此处仅为原理说明）。

针对以上每个函数，继承链上的执行顺序都是先执行父类的函数，再执行子类的函数。其执行顺序为：父类的所有扩展类函数 -> 父类对应函数 -> 子类的所有扩展函数 -> 子类对应函数。（在BUI的标准中，扩展类是指没有initializer函数，而拥有构造函数的类，一般提供功能扩展。）

比如，某业务模块继承了Controller模块，那么，在业务模块实例化的过程中，必定是先执行Controller的initializer函数，然后再执行业务模块的initializer函数。

而插件的执行则是在继承链上所有的函数执行完毕后才会执行。某业务模块继承了Controller模块并使用了分页插件，那么，在业务模块实例化的过程中，必定是先执行Controller的initializer函数，然后执行业务模块的initializer函数，最后再执行分页插件的initializer函数。

下面是对模块内部生命周期函数的说明：

initializer

initializer函数相当于模块的构造函数，其作用是组织并初始化可视化模块的**BUI对象结构**。注意，在initializer函数中仅做结构层面的组织，不涉及任何页面渲染、事件处理的逻辑。甚至，某些BUI对象的结构和渲染综合在一起时，结构组织的工作会放入renderUI函数中去进行，比如grid对象。

initializer阶段不触发任何事件。

createDom

createDom函数的作用是创建模块**DOM结构**。需要注意的是，此方法在框架中仅用于根据模块的elTagName建立对应的dom，再根据contentEl建立对应的子dom，但此时建好的dom结构**并没有**挂载到实际页面上。

进入createDom函数时，触发beforeCreateDom事件；模块及其继承层次和扩展类createDom完成后，触发afterCreateDom事件。此后再进行插件plugins的createDom。

renderUI

renderUI函数的作用是渲染模块。简单来说，就是将createDom中创建好的模块DOM结构挂载到页面上对应的render中去。所以，所有BUI模块render函数的执行，以及对页面的调整都应在此函数中进行。

进入renderUI函数时，触发beforeRenderUI事件；模块及其继承层次和扩展类renderUI完成后，触发afterRenderUI事件。此后再进行插件plugins的renderUI。

bindUI

bindUI函数的作用是针对模块的BUI对象或者DOM对象的事件进行绑定。注意，在此函数中不仅仅是对事件绑定处理逻辑，而重要的是封装模块的自定义事件。

进入bindUI函数时，触发beforeBindUI事件；模块及其继承层次和扩展类bindUI完成后，触发afterBindUI事件。此后再进行插件plugins的bindUI。

syncUI

syncUI函数的作用是根据初始值来同步UI的状态，即读取ATTR中设置的默认值，并根据这个默认值对UI的状态进行设置。

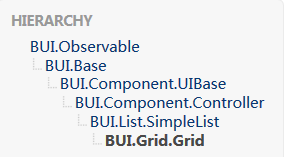
进入syncUI函数时，触发beforeSyncUI事件；模块及其继承层次和扩展类syncUI完成后，触发afterSyncUI事件。此后再进行插件plugins的syncUI。

* + 1. 实现机制及样例解析

通过对逻辑结构和调用链的分析，结合源码的阅读，我们可以总结出BUI当中使用了以下实现机制：

* + - 1. 继承机制

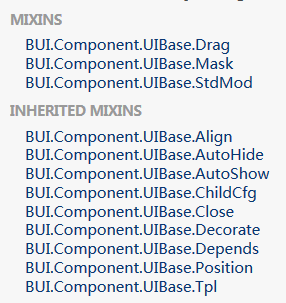
BUI中采用了原型链继承方式，实例化父类作为子类的原型。继承机制在绝大部分模块中都有相应的体现，通过功能职责的划分，形成了继承链。比如，grid的继承链如下图：



Observable提供了事件机制，Base提供了模块的基本功能和定义，UIBase提供了可视化模块的基本功能和定义，Controller提供了可实例化可视模块的基本功能和定义，SimpleList提供了简单列表的基本功能和定义，最后Grid具备继承链上所有功能，并自定义了业务列表的功能。

* + - 1. 扩展机制

BUI的扩展机制主要体现在将其他类作为mixin集成到指定类上面，以达到功能扩展的目标。扩展机制在绝大部分模块中也有相应的体现，为模块提供了很多增强功能。比如，dialog的扩展类如下图：



dialog本身具备三个扩展类：Drag、Mask和StdMod，分别提供了拖拽、遮罩和上中下布局三项功能，而它在继承链上还获得了相对定位、自动隐藏显示、子配置、关闭等一系列扩展功能。

* + - 1. 插件机制

BUI的插件机制由UIBase提供，但在框架中仅grid拥有插件。插件机制的实现是将使用插件的模块实例传入到插件类中，由插件类实现增强功能。

插件机制和扩展机制都是为模块实现增强功能，其区别有二：一、插件是模块实例化时可配置选用的；二、插件的实例化过程在模块及其扩展类之后。

* + - 1. 属性机制

BUI的属性机制由Base提供，其作用在于，提供设置获取属性、提供事件支持、属性变化时会触发对应的事件、将配置项转化为属性、支持自定义属性、支持属性的静默设值即设置属性值不触发事件。

* + - 1. 事件机制

BUI的事件机制主要由Observable提供，实现了事件的绑定、处理、暂停、中止等。

* + - 1. 模板机制

BUI的模板机制主要由TplView提供，支持带占位符的模板解析、根据模板和属性值生成DOM。需要注意的是，如果设置了content属性，模板是不会生效的。

* + - 1. 渲染机制

BUI的渲染机制主要由UIBase实现，控制了模块渲染的生命周期和状态变化，详情参见4.3.3.4.1UIBase对可视化模块的生命周期管理。

* + - 1. 依赖机制

依赖机制在很多地方都得以体现，比如插件必须依赖模块才能生效，模块必须依赖DOM才能渲染等等。

* + - 1. 父子控件机制

BUI的父子控件机制也在很多地方有所体现，比如Menu和MenuItem、Bar和BarItem、Form和Field等等。

* + - 1. 视图控制器机制

BUI的视图控制器机制由View和Controller联合提供，View封装了视图层，Controller封装了控制层。

* + - 1. 自定义状态机制

BUI的自定义状态机制由UIBase提供，支持自定义状态并覆写syncUI方法来实现自定义状态对页面的影响。

# 总结

BUI框架在js框架发展中是一个承前启后的框架，设计上前承mvc后接mvvm，技术上前承Jquery后接WebModule。通过对这个框架的学习和掌握，能够一窥js框架的发展和革新，起到认清过去、面向未来的作用。

我们采用BUI的最大原因是铁路用户现场仍然拥有大量的XP系统乃至ie6浏览器，但web模块化潮流不可阻挡，而大量开发人员具备的仍是Jquery技能。因此，在实现上可以通过jQuery进行hack，在设计上可以采用模块化原则进行设计的bui目前仍是我们的最佳选择。

# 附录

## bui相关github地址

源码地址：<https://github.com/roysong/bui.git>。本源码库是经由官方库fork后修改的版本，我们自己增加的公用组件将合并到本库中进行发布。

API地址：<https://github.com/roysong/bui-docs>。本API库是经由官方库fork后修改的版本，我们自己增加的公用组件API将合并到本库中进行发布。

教程地址：<https://github.com/roysong/bui-document>。本教程库是经由官方库fork后修改的版本，首先会补全原版中缺失的教程，其次我们自己增加的公用组件教程将合并到本库中进行发布。