# 前端框架的一些介绍和思路

2016-03-09
焦向波

## 1. WLJ 框架现状介绍

### a) 背景

CRM 前端框架的最初的构建，是为了降解之前裸用 Ext 框架时带来的一系列负面影响，如，门槛高、BUG 率高、代码量大等问题；同时，期许于框架的重新构建，在之前的操作、UI 上，能做出一些突破。

截止到目前为止，在两三年框架的发展过程中，形成一些思路。

### b) 运行原理

虚拟机式的代码运行机制

- *看 java 的收获*

*在研发开发框架的过程中，我的 eclipse 恰好老是出问题。就翻阅参考资料，折腾 java 虚拟机，看 java 代码执行原理之类的，倒是受了一些影响。*

针对普通开发人员的业务逻辑开发，WLJ 框架将页面抽象成界面配置句柄和逻辑片段句柄，同时，以全局句柄的方式，给开发人员提供一些工具方法（API）。通过这些句柄，将框架运行与业务逻辑解耦。

开发人员负责填充这些句柄；运行框架负责解释这些句柄；换言之，就是将业务描述，与技术运行解耦。

- *框架取舍*

*当然，这样直接采用全局句柄的方式进行逻辑与技术的解耦，存在一定的句柄冲突风险；同时，也进行了必要的取舍，放弃了完全的 ONEPAGE 模式进行，变得有点胖瘦客户端模式掺杂；但是，避免了命名空间系统的设计与管理，降低了内存管理与回收的难度。*

类似 java 虚拟机的运行过程，框架的运行是一套封闭的，自运行的逻辑模式。

其运行方式为：

1、 首先加载框架运行所必须的支撑文件，构建运行所需的体系；
2、 根据菜单参数，加载业务代码；
3、 根据代码引用，加载第三方类库；
4、 检查业务代码句柄，排除必要的错误；
5、 构建整体调度对象以及下辖的子对象，渲染界面，绑定事件等等操作（业务代码的载入至此完结，后续就是针对业务输

入的响应了，完全由框架进行调配）；

6、 根据用户输入，调用必要的事件逻辑片段。

- *首页框架*

*以上描述，为单个功能点内部的逻辑运行原理。而在开发框架之上，首页框架上同样伴随着整体的任务窗口管理、瓷贴队列管理、版本管理等功能，这里不细述。*

## c) 开发接口

开发接口，在这里主要是指我们的句柄和 API；但是，在有些情况下，某些全局配置、某些内部类暴露的方法、以及 Ext 的一些源生对象也会成为业务逻辑开发的部分。

在开发接口的设计在这几年的更新中，逐步形成几个规则：

1、 **接口文件要独立、稳定**。句柄接口不归属于框架，也不归属于业务。通过句柄名称的固化，形成独立的句柄对象；同时，无论在框架发生怎样的变化，句柄接口名称、结构、以及调用方式不发生变化。

- *句柄带来解耦，解耦带来好处*

*这个规范形成，不仅仅是为了框架与功能（技术与业务）的解耦。更重要的一点在于遵循这个规范，产品框架在版本发展的持续性，兼容性上能够得到有效的控制与保证。框架的滚动更新，独立的句柄可以有效地摒除往期框架版本下业务代码的兼容性影响。*

- *看 C 的收获*

*关于 header 文件的形成，最初是学习了 C 语言中的头文件的设计方式。C 语言中，头文件申明结构、方法、以及各类变量句柄，当然，也并不强制变量一定在头文件中声明。我想这应该是 C 语言为了解决大规模 C 应用的协作问题，而给出的一个软性的协作方案。*

2、 **分化脚本语言的动静态特征，降低门槛。**框架本身要多使用脚本语言的动态特性，多动态的构建对象，动态的绑定事件，动态调用过程;这样,就可以让业务开发人员尽量静态的定义变量、静态地编写函数、静态地调用 API。

- *脚本的动态特性*

*脚本语言，一般语法规则简单，但调用极为灵活。其简单的安全模型，导致内存对象的动态特性极强，方法、过程的调用时机、顺序因素极为复杂；这些因素往往使开发人员的晕头转向。其代码的文本又是静态特性的，不需要经过编译链接，极容易将文本转化为代码以及内存对象。*

3、 **接口结构的设计，要尽量使用简单变量，简单函数。**开发人员代码尽量降解到与业务相关的必要配置,而不需要关心执行过

程，与触发时机。

4、 **结构化的全局配置对象。**通用的逻辑，抽象成全局配置对象，在所有功能点生效；全局配置对象只起到一个默认值的作用，不强制。开发人员可以在个性化的界面中订制配置。

5、 **分级过程化的补丁。**框架中提供了几种级别的补丁文件。对于一些需要逻辑过程的项目组个性需求，要提供一些可以编写这些代码的地方，这些代码可能会覆盖一些框架逻辑，可能会提供一些全局逻辑；一些不具备添加到产品框架功能的个性需求，或者一些需要紧急响应的过渡性需求响应，也可以在这种补丁文件中添加。这种补丁文件只能由项目组架构师来修改，或者产品部定向发送給指定项目组。

- *前段框架的补丁*

*目前，CRM 产品的前端框架中，包含几个这样的补丁文件：Crm-Ext-Extends-1.000-v1.0.js、Crm-Ext-Patch-1.000-v1.0.css 用于与底层支持库 EXT 框架的一些逻辑修改、BUG 修复、以及底层功能、浏览器版本适配等，它的加载时机是 Ext 加载之后，WLJ 框架加载之前；systemPatch.js 用于 WLJ 框架相关的一些逻辑覆盖等，它的加载时机是 WLJ 框架代码加载之后，框架构建之前；defaultPatches.js 用于为业务逻辑代码添加一些通用默认逻辑，不强制覆盖，它的加载时机是框架构建之后，业务代码加载之前。*

另外，还有一些小的规范，如：6、功能性需求由框架来做，业务代码只负责需求描述，质量高、风险小；7、句柄接口的订制，顺序的安排，尽量与需求文档的格式靠近；8、补丁、更新等，尽量以文件为单元进行设计编写；9、提供一些框架外的实用功能，如日志 API，日志控件等；10、配置信息不急于持久化，不急于存到数据库。

- *代码文档化的尝试*

*关于第七点，是代码文档化方面的一点尝试。Crockford 先生最早提出代码文档化的观点。他说大家写代码，变量名要取好，方法名要取好，注释要写好，要让别人能看懂。可惜的是，这只是一种提倡。python 做的好点，但也仅仅在代码排版上有些进步，而且，它自身也陷入了无限回调的泥潭，所做有限。WLJ 的开发框架中，最初的一些接口设计有些参考了咱们需求文档的写法。比如，先编写输入输出字段，然后增删改面板逐个描述；字段约束逐条编写；业务限制也是附在最后逐条编写。我总觉得，代码文档化这种大命题，可以在具体的业务框架中，通过框架的强制性，是能够有一些前进的。*

框架的设计，就是取舍；最难的也是取舍。说多了，就成哲学了。绍雷说，抓住痛点。痛点与前瞻相结合，就是取舍的准则。

## i. 面向架构师的全局配置

框架提供一个全局化的配置文件，该文件中定义了各类全局性的配置属性。一些无关于业务的默认值、开关定义在这里。同时，包含了界面上各级别右键菜单逻辑的定义，以及可扩展的前端数据类型配置。这

个文件中的内容，可由项目组架构师角色的人员上线前统一配置修改。Wlj-frame-function-app-cfg.js

**参考附：配置及补丁文件**

同时，在这些静态配置的基础上，框架同时还提供了 systemPatches.js 和 defaultPatches.js 文件，用于做一些动态的逻辑化的全局修改，前文已有介绍。这两个文件可由项目组架构师角色人员统一修改，或者由产品部协助修改下发。

## ii. 面向视图的属性句柄

这一部分的开发接口，特指在 Wlj-frame-function-header.js 文件中所定义的，除 listeners 对象之外的全局句柄。

这一类句柄绝大部分是定义界面的属性。但是，与其他的 js 框架不同，他并不直接定义或者接触 DOM 文档，也不直接定义或者创建 js 组件对象。

**参考附：接口文件**

这些句柄更多的从较为业务化的方面，对业务需求进行描述。比如，从哪里查数据，页面用了哪些字段，有哪些面板，有多少条校验规则，引用了哪些数据字典，有什么按钮之类的。

## iii. 面向响应的逻辑片段

这一部分开发接口，定义在 header 文件的 listeners 对象内。**参考附：接口文件**

这一类句柄大部分用于对用户输入做出逻辑响应。用户的所有输入，包括键盘点击，鼠标悬浮、移动、点击、右键、拖动等等，都会由框架捕获，并根据事件的实际情况，选择调用不同的片段逻辑进行响应。

这一类句柄，同样是以全局模式定义，但是以 function 的形式编写。这些函数在被调用的时候，会自动接收到与该输入响应有关的、逻辑片段编写所必要的参数。

- *Javascript 的函数调用方式很多，我们只提供一种*

*Js 函数的调用方式很多，常常导致函数内部逻辑出现不必要的问题。函数既可以作为方法独立存在，也可以作为对象类存在；函数的调用，既可以直接调用，也可以通过对象句柄调用，也可以通过指定作用域的方式进行调用。这三种调用方式，使得内部逻辑的作用域不同，说简单点，就是 this 指针所指代的东西不同，这就很有可能让函数的执行出现不可预期的问题。所以，在框架中，将所有逻辑片段句柄的执行作用域统一定义为全局统一调度对象（APP 对象）。*

这一类句柄，被定义为 listener，或者说“事件”。但是，它们并不是 DOM 事件，它们并不会接触 DOM，也不会成为对象监听。这些事件，都是由框架逻辑运算触发。有可能是几个源生事件的组合，也有可能是几个源生监听的组合。它们是更为业务化的。比如，它们可以定义页面加载以后要不要先做一些事情；选择一行数据后，要不要做一些事情；打开了新增面板要不要做些什么；要提交数据，在这提交之前和之后，

要不要自动做些什么。

## iv. 全局 API 的调用

API 系统，在 WLJ 框架中，特指在 Wlj-frame-function-api.js 文件中定义的 API 对象的所有键值。API 可以简化逻辑片段的编写难度。**参考附：接口文件**

这些 API 的调用域为 window，也就是说，直接根据名字和参数说明调用就可以了。

这些 API 通常可以独立发动一些效果，或者返回必要的结果。这些 API 同样是比较业务化的，无关 DOM，也无关 window 或者浏览器对象。

这些 API 作用域均为全局调度对象（APP 对象），也就是说，都是在 APP 对象上定义。但是，独立的 API 对象上的键和值（true 或者 false）用来定义该 API 是否装载。

这些 API 可以直接调用，但是有一定的约束条件。由于这些 API 的实现均在 APP 对象上，所以必须在 APP 对象构建之后才能调用。或者，简单点说，就是只能在其他的方法体内调用，不能在代码中顺序调用。当然，也只有在编写逻辑片段的时候，才需要调用这些 API。

## v. 其他一些子系统

### i. 日志监控系统

框架中的日志控制台中，框架内部的日志主要围绕两块进行：

1、代码加载时的代码检查日志

2、事件响应时的逻辑片段调用日志

另外，开发人员在开发调试过程中，也可以调用日志 API，在控制台中打印日志代码。

### ii. 异常捕获系统

异常捕获后，如果控制台打开的情况下，会在控制台中打印 js 错误信息。

目前，主要捕获的异常信息包括：

1、逻辑片段句柄中抛出的异常；

2、界面配置句柄中部分需要编写的函数，如字段校验，字段联动等。

### iii. 插件系统

界面加载完毕之后，会自动启动插件系统。插件文件中定义了插件编写接口。目前，插件系统带有查询方案保存插件和数据缩略图插件。

### iv. 设计预览系统

设计及预览系统包括界面设计功能，和预览系统。

界面设计功能中，可以完成对界面要素句柄的设计操作和代码的自动生成；设计功能待完善。

预览界面可以在新的窗口中直接预览设计的界面，并在代码控制台直接动态编写代码，或者动态绑定事件句柄。

### v. 测试加载系统（初版不甚满意，尚在完善）

测试系统目前有了自动测试的初步探索，但是，对于测试用例的编写和积累方案尚未认定。

主要问题也是集中在取舍。用例跟着代码版本走，比较便于管理和维护，但是这些用例会让产品工程变得臃肿；如果用例单独作为文档管理，不便于管理维护和积累。

# 2. 底层框架适配细则分析

## a) ExtJs 框架的使用现状

WLJ 框架以 ExtJs 为底层工具和类库来组织的。但是，WLJ 的整体运行机制则独立调用与实现。

- *Ext 的一点介绍*

*关于 ExtJs，通常人们使用这个框架，比较关注的就它提供的丰富的 UI 组件，当然，也有人诟病核心库太大。这些都对。*

*不过，我还是觉得这玩意儿真算是天才狂想式的作品，以至于我看到《架构师》杂志中一篇评论 js 框架的文章中，不得不将 ExtJs 单独作为一个分类，因为当前主流的分类都不合适。*

*主流的 js 框架（或者说库），一般有以下几个方面的功能：1、基本的 DOM 操作工具；2、事件处理工具；3、Ajax 的数据交互工具。随着 javascript 的应用发展，逐步出现了将以上功能对象化的封装，将常用的列表、表格封装为对象；再后来，WEB 架构师开始想着把 WEB 工程的传统的框架思路搬到 javascript 框架上来，（在我看来，这些人，应该是在互联网大规模点阵化架构应用逐步丰富起来时，落伍的一批架构师。）MVC，或者 MXXC 之类的。我觉得这并不是最好的出路，脚本语言的动态特性的灵活运用，匿名函数，匿名对象的应用，才是其未来的出路。这并不是本文的重点。*

*关于 ExtJs，除了具备以上的几点的基础工具和类库外，它提供了一些颇为有效的特性，值得思考。有些在 WLJ 框架中，有颇多应用。*

*ExtJs 从 JAVA 基础语法和 JAVA 设计模式中借鉴颇多。所有对象从 Observable 对象逐层派生，其间很多面向对象的设计模式得到应用，单态、工厂、观察者、门面之类的，以及匿名对象，向上转型等。*

*所有的 UI 组件和控制组件，在逐步派生的过程中，形成固定的、软性的派生语法，让他的实际可用组件变为无限。*

*ExtJs 对 JS 源生类对象做了一些强，尤其 Function 对象的增强，提供了足够模拟线程和切面的方法。*

*提供了颇多专门针对于数据结构的运算工具；*

*提供了一个 MixedCollection 的数据结构。该数据结构等同于 List 和 Map 的结合，运用灵活，效率颇高。在 WLJ 框架中使用颇多。*

这里的主要是指框架中使用到的内容。

ii. Ext UI 对象的引用：主要包括：表单、列表、布局、动画、拖动、菜单、窗口、各类输入框、按钮、树形面板等对象。这些对象绝大多数是由框架内部自行控制，由框架自行控制构建渲染；

iii. Ext 控制对象引用：数据源对象、Ajax 对象，MixedCollection 对象；这些在框架封装初始，也基本由框架控制代劳了；

iv. 其他，包括各类 DOM 增强 API，事件绑定机制，命名空间系统、派生系统；此类功能也大都由框架使用。

v. 各条线前端代码中，有大量使用 Ext 派生的公共组建，各类放大镜、上传下载导入导出等等功能。

vi. 部分业务逻辑代码中，由于业务操作较为复杂，开发人员采用源生的 Ext 对象做了部分开发。

## b) 框架层面 ExtJs 的可解耦性

WLJ 框架的发展，到目前为止的主要总结是：面向句柄的开发模式。而要做到这一点，主要的两个方面，包括：1、框架要提供运行环境，运行效果由框架执行，使代码静态化；2、稳定、固化业务抽象的独立的开发句柄对象。

通过以上的两点，基本上框架可以做到业务技术分离；框架的版本滚动更新；UI 框架逐步解耦；通过对底层的修改，支持多种平台。

那么，ExtJs 作为其底层支持的库，有多大的可解耦性。这是详细逐步分析的。

**1、吸收：**对于 ExtJs 框架的部分比较优秀，运用比较的频繁的 API，或者机制应该吸收进 WLJ 框架的底层；如：Ext.ns（命名空间定义），Ext.extend（对象继承机制），Ext.util.MixedCollection（集合数据结构），Observable（基于观察模式的基础对象），部分 JS 与源生对象的增强，部分 DOM 的增强等等。

**2、抽象父类：**目前框架主要的组件，大都是从 Ext 的组件继承派生出来的。这里应该逐步将父类抽象化，形成一套类似适配器的父类派生系统。逐步将当前的框架版本变成抽象父类的一个 Ext 版本的实现。之后，我们的自己编写底层支持库，或者更换其他的 UI 支持库，将会有一个实现标准。如：列表、表单、树形面板、窗口、普通面板的基础 UI 对象的抽象类；Ajax 对象、数据源对象等的向上抽象；动画特效、拖动功能对象的抽象；事件系统的抽象；等等。

**3、子系统的替换实现：**如插件系统、设计、控制台等系统的实现要适配新版本的支持库。子系统的抽象，可以以接口的形式抽象，且实现工作可以靠后，不影响主体功能的运行。

**4、样式资源文件系统：**如果替换新的支持库，如果该支持库是自带 UI 组件的，则需要研究拆分其样式资源体系，尽量做到符合当前的样式体系标准。对于涉及到内部事件判断的样式，必须与现有样式体系保持一致。

## c) 开发过程中，对于 EXT 基础对象的替换或者剥离

- *跑东方资产的收获*

*前两周跑了东方资产两次，项目上有个业务功能极其复杂。功能的代码，加上引用的 js 代码，行数上几乎是 WLJ 框架代码总量的数倍，这应该是需求设计的问题。*

*该功能中，有颇多地方需要实现表单、列表的动态嵌套使用。原本，框架并不提供这种功能的解决。因为 WLJ 框架提供运行环境和句柄，但不提供类库，因为 Ext 的组件库已经非常丰富了。*

*但是，在编写构建这些面板 UI 的时候，开发人员并不使用 Ext 源生组件，转而使用了 WLJ 框架内的私有组件。因为该组件提供了默认的排版、构建、以及一些按钮的默认功能等，编写要方便的多。由于该组件是有上下文关系的，所以在运行的时候出问题了。这边，我安排人，编写了几个类，基本上调用参数不变，但是，排除掉了上下文的影响。*

*可是，我在做这些的时候犹豫了。单纯的提供组件，是有悖于 WLJ 框架理念的。现在大部分的 JS 框架，只负责类库，不管理句柄；同时，对象都是直接调用，每一个对象都要有一个句柄去定位。这样，业务代码复杂，冗余，易出错。这是目前绝大多数开源 js 框架的问题。*

*所以，特意增加了预加载开发句柄和构建 API。这些对象的创建、检索，和移除，都使用 API，不使用句柄，以此来保证这些对象都受框架控制。*

开发过程中，面对复杂的功能点，很多时候，开发人员不得不选择源生组件库来满足一些个性化的复杂的体验要求。然而，我们的框架做的事情并不提供组件库。所以，在之前的很多时候面对这些需求的时候，我也只能无奈的建议开发人员使用源生组件。倒不是我懒于添加一些组件，而是一旦提供组件库，开发人员可以任意创建组件，必然导致句柄膨胀，且句柄不可管理。这与我们的框架理念背道而驰。

- *关于句柄膨胀和句柄的不可管理*

*任何一个系统，都应该是一个输入输出功能的有限集合。面向句柄编程的思路，最为关键的部分，就是句柄。句柄的膨胀会大大增加句柄冲突的可能性；句柄的不可管理，则隐患更大。如果，业务代码中，大量出现不可管理的对象句柄，那么框架的可控范围就大大减小的。很多对象的生命周期无法管理；版本滚动更新时，还得避免这些不可管理句柄的雷区；尤其，如果我们的框架需要更新、更换一个底层的 UI 库，或者更换一个平台执行的时候，这些句柄对象，就成为了拦路虎。一来，新的句柄接口必须跳过这些地雷；二来，我们的不得不为了这些不可控的句柄对象引入，或者编写与之对应的组建库，这是一个庞大的工程。*

所幸，东方资产运用的思路是我认为颇为有效的一个思路。既可以提供一系列的个性组件，又不至于让句柄膨胀，不可管理。

总结一下，方案大致如下：

1、 我们构建不同的个性 UI 组件，但不提供直接创建的入口；

2、 在全局 API 中，提供对这些组件的构建，检索，以及移除销毁的方法。虽然这种方法也有可能会产生多余句柄，但是，这些对象的生命周期完全受控于框架，这个句柄的存在与否，就不会成为太大的隐患。

这个大致的方案，应该在接下来的更新中，逐步在产品研发和项目组的开发使用中推广。让一线开发人员不再使用源生的这些组件对象，而是使用 API。这样，未来的如果更换平台，或者支持库，对于这些内部组件的适配，就与框架内部受控对象的适配方案一致。

## d) 不得不加的一节

锦哥周三下午一通电话，我在这两天的大风里凌乱得不要不要的。

作为一套开发框架，WLJ 框架想要做的，只是在底层的 UI 支持库之上，与开发人员的业务代码之间，构建一套无差异的开发运行环境。它可以适配各种底层的 UI 框架，也可以去适配 PC 或者移动浏览器，或者移动 APP；寄希望于保持开发接口的稳定，来让框架品屏蔽底层支持的不同，以及版本更新对业务代码的影响，甚或而能够统一代码，让同一篇代码在各种平台上，跑出相应的效果。

要总结一下有可能的替换底层平台的需要的思路，其实大体上的思路就是我们在 ExtJs 的解耦性那一节中提到的，解耦的过程。一旦框架的底层边界（抽象父类）确定，就是可以按照固定的标准来进行实现。

1、 实现所有的必要的抽象父类；

2、 对开发接口的支持，要覆盖到所有的句柄和 API。

# 3. 一些观点

## a) 架构

架构是一个很有趣的概念。因为大家做软件越来越复杂，发现更建筑学很像，就借了这个概念。于是，也就出现架构师这个说法。算是职位？算是称呼？我不很懂。

首先，我先申述一个观点。架构师和开发者只是分工不同而已，不存在谁比谁更应该有优越感。

架构的概念最早是建筑学的，那建筑学的定义就不说了。

软件领域的架构，也有很多的定义。听到过的一个最高深也最高大上的定义是：架构，就是把不同的东西分开来摆放。怎样，听起来牛逼吧。

那是一个非常资深的架构师的定义，我还不能真正体会这句话的高深含义。但也颇有些自己的体会。我虽然不能直击核心地给出如此精彩的定义，但也能从浅一些的层面或者侧面来给出一个我的定义。

架构，是带着严重的个人哲学理念的一套规则或者规范。

架构最核心的难点在于取舍。

所以，架构师最重要的品质也在于怎样面对取舍，或者说妥协，对于妥协两面的思考。

架构师嘛，好多人都可以这么叫。从通俗上的架构师定义来看，架构师也可以分为写框架的架构师和用框架的架构师。

核心意义上的架构一定是包含着技术成分也业务成分的。这两块很难分开，也很难区分。所以，很多时候啊，不要太过迷信有些主营业务不是 IT 服务的公司所开源出来的产品。他们开源出来的产品很可能阉割掉了最核心的技术，或者至少做了业务脱敏。那么，我们所真正能够看到的，也只是比较皮毛的东西了。

## b) 框架发展与含义：从 main 函数开始，逐步形成框架与产品

何为框架。我要写的东西有点执著。见谅。

从 JAVA 谈起。对于我们这些只是使用语言，而不开发语言的人员来说，绝大多数情况都是从 main 函数开始。这不仅仅是说我们的学习经历，还包括了设计理念、框架等的发展历程，也是程序运行的过程。

最初的框架，就是 JVM。它负责运行，你负责写 main 函数中的逻辑片段。

后来分化出 SE 和 EE（ME 不聊，聊起来有点崇洋媚外）。SE 做桌面程序，败的一塌糊涂；EE 做 WEB 以及网络，非常成功。

SE 做逻辑，但也需要做 UI 界面，后来出现 AWT、SWT、SWING 之类的组件库，但还是一塌糊涂。

EE 做网络程序；这里它们首先提出了自己的 main 函数，就是 severlet 接口，开发就是补充 severlet 里的逻辑片段；在这个方向上，又出现了 Struts 之流，它们把这个 main 函数丰满起来，让开发人员写的片段更少了。当然，这里还有一个关键，在于 UI 这块，让 HTML，或者说浏览器去做了，这让 EE 这块的 main 函数写起来容易多了。

单纯从框架这一块来聊一聊区别或者说成败。SE 给你提供了组件库，让你组织整体 UI；EE 这个方向，给你做好了整体，让你填写其中一部分。

一个让你造句；一个让你填空。

当然，还有另外一个问题，就是 UI（或者精确点说，是界面）这部分内容。界面太具体了，所以，抽象起来很难；还有一个点，界面是客户可以直接看到的。他们可以说三道四，指指点点，或者投入自己的想法。于是，想法越来越多，越来越混乱，新的要求和需求几乎每天都有。它们是如此的灵活，如此的“不合理”。架构师们不得不以组件的方式提供 API，把整体的设计交给能够直接接触需求的一线开发人员。

因为，没有取舍。这样的取舍太难了。取舍出了问题，不仅仅会被客户骂，还会被开发人员骂（内部都不团结咯）。

纵观 JAVA 历史上，各种号称框架的程序，其发展应用广泛与否，大部分是由我们的一线开发人员的开发体验决定的。他们用键盘投票，决定这些框架的生与死。（当然，运营支撑团队自身出现问题的原因除外。）而，生的这些框架，又大都是填空类型的，不是造句类型的。

不知道这样的总结对不对。但是我还是觉得，弄填空类型的东西，好一点。

我们现在聊的前端框架，就面临着 SE 的问题。用户对界面的 UI 体验要求越来越高，我们不得不在前端做大量的逻辑运算，尤其现在日益流行的移动前端。前端个性化逻辑的大量出现，就得有能支持逻辑的前端框架。

那么，我们做前端框架应该怎么玩？

## c) 框架来执行，代码静态化。

句柄化的框架代码，业务逻辑逐渐静态化，尤其是脚本语言的静态化天性；静态化的代码更容易格式化、文档化。

## d) 静态代码通过不同的解释运行机制，形成不同的产品效果，包括不同版本，或者在不同平台下运行。

## e) 基于句柄化的业务抽象可以有效地隔离业务需求与技术实现（提供环境，管理句柄）

- *先例*

*早前有一个版本的绩效产品，采用了 Ext2.2 做为前端框架开发。在该产品的前端代码中，做了一些简单的布局句柄，比如东南西北的面板的句柄。但是，可惜，该版本的产品做得不很彻底，也并不强制。该产品的代码有一些句柄化框架的小思路了，但是并没有人去真正做一个框架出来。从业务中抽象句柄，并独立形成可生长的句柄系统；当然，还有更为根本的缺陷，就是约定的几个句柄，仅仅是简单地教给 Ext 框架去执行，并没有提供完善的运行环境。*

## f) 自运行框架在自动化、测试、监控方向上都可以很好的支持

## 4. 附：

## a) 接口文件（API、HEADER）


Wlj-frame-function-api.js


Wlj-frame-function-header.js

## b) 配置及补丁文件(CFG、DEFAULTPATEH、SYSTEMPATCH)


Wlj-frame-function-app-cfg.js


defaultPatches.js


systemPatch.js

## c) 运行支撑文件(APP、BUILDER、BOOTER)


Wlj-frame-function-app.js


Wlj-frame-function-builder.js


WljFunctionBooter.js

## d) 私有组建库


Wlj.frame.functions.app.widgets.EdgeView.js


Wlj.frame.functions.app.widgets.ResultContainer.js


Wlj.frame.functions.app.widgets.SearchContainer.js


Wlj.frame.functions.app.widgets.SearchGrid.js


Wlj.frame.functions.app.widgets.TreeManager.js


Wlj.frame.functions.app.widgets.View.js