前端 MVVM 与 FRP 的升阶实践：ReRest 可视化编程

by Wayne Chan, 2017 年 10 月

ReRest (Reactive Resource State Transfer) 是前端开发领域新兴的方法论体系，它继承了 MVVM 与 FRP 编程理念，在技术上有不少创新。本文从专利稿修改而来，主要介绍 ReRest 原理与若干实践经验。

1. 前言

前阵子 React 附加专利条件的开源协议闹得沸沸扬扬，国内外有多家大公司开始弃用 React，我们也深感困惑，是否该将 shadow-widget 全盘改写，很犹豫。让底层脱离 React。但专利与开源协议是平行的两个世界，改底层也不大容易解决问题。Facebook 拥有虚拟 DOM 方面的专利，preact、vue 都可能涉嫌侵权，通过修改底层代码来规避还是挺难的。

后来我们决定自己申请专利，以便今后万一用到，手头有个专利可为 shadow-widget 增加话语权。当权利要求书完稿时，Facebook 宣布 React 回归真正的 MIT 开源协议了，真是大喜讯！我们不必担心专利的风险了，为自家申请专利不再必要 —— 我们创建 shadow-widget 技术平台，但无意借此盈利，源码开放出来让大家都受益。（PS：不必感谢，如果觉得这项目对您有用，上 github 为我们加星吧）

本文从专利申请稿改写而来，内容有压缩，要不文章太长了，另外还增加了可视化编程实践相关的若干内容。公布此文还有一个目的，防止他人偷偷拿我们的技术申请专利，如果以后真发现有人这么干了，本文是凭证，大家可以提请专利无效，把别人的保护条款废掉。

说明：本文完稿时，Shadow Widget 最新版本为 v1.1.2，产品用户手册对技术实现有更详细介绍。

2. 背景

近些年 Web 前端技术发展，可以说是框架横飞的时代，虽然十年前网页还正常能打开，IE 还是那个顽固的 IE，但前端开发却已经历翻天覆地的变化。近来比较抢眼的是 React 框架，Facebook 开创性的实践了两种技术：虚拟 DOM 与

Functional Reactive Programming（FRP，函数式响应型编程），这两种技术几乎已成现代前端框架的标准配置。

Facebook 在虚拟 DOM 上原创较多，钻研很深入，这项技术也可以说很成熟了。FRP 在 React 的实现就是那个 FLUX 框架，它不是 Facebook 首创，在 React 中用起来也有点磕磕碰碰，尤其在调和指令式风格与函数式风格方面，并不顺畅。

另外，尽管十年来 Web 开发技术发展很快，但在可视化开发方面仍然进展缓慢，所有主流框架都在界面的形式化描述上做文章，Angular 与 Vue 扩展了标签属性，增加不少控制指令，React 则全盘引入 JSX 描述方式，他们无一例外的都要求大家，一行行写脚本去定义界面，而不是 20 年前在 Delphi 与 VB 就已出现的可视化、所见即所得的开发方式。

本文所提的 ReRest 编程方法，是适应 Web 可视化开发要求，融合虚拟 DOM 与 FRP 技术，并克服它们应用于主流框架的若干不足，而提出的通用型解决方案。ReRest 方法在 shadow-widget 平台有一些实践，已取得良好效果。

3. ReRest 要点

ReRest 全称为 REactive REsource State Transfer，译为“响应式资源状态迁移”，与本概念相关的提法还有：

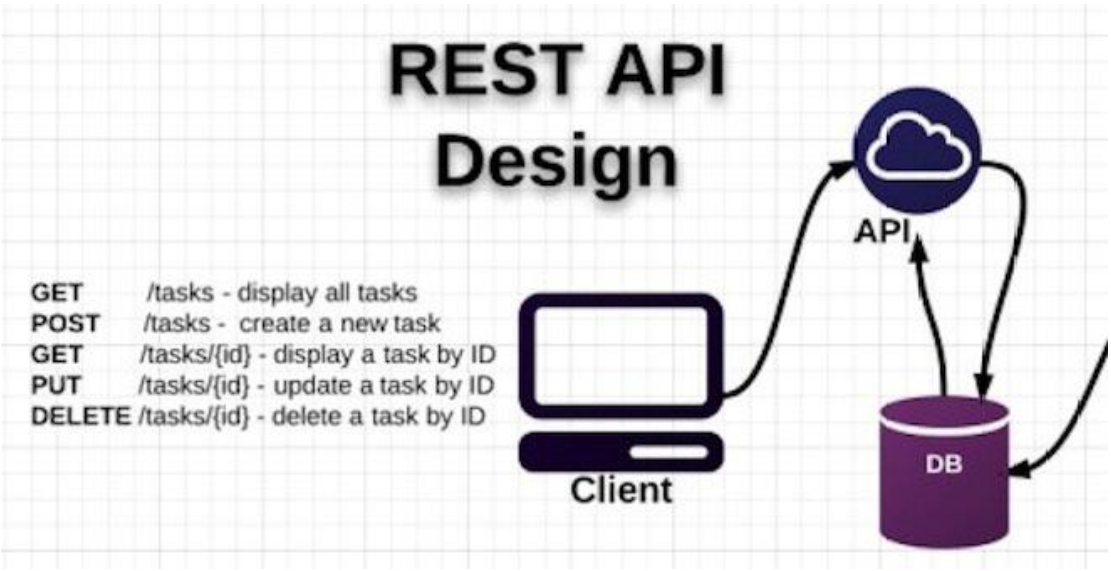
1. ReRest framework，ReRest 框架
2. ReRest based programming，基于 ReRest 的编程
3. ReRest-ful design，ReRest 风格设计

光从字面上看，“响应式资源状态迁移”不大好理解，就像缩写为 REST 的“Representational State Transfer”，表现层状态转移，只看文字，是不大容易搞清楚讲的是啥。

ReRest 提倡以“资源”的观点展开设计，将针对资源的操作规格化，统一抽象成 4 类操作，在程序开发过程中，可视界面的功能块分解设计是一个维度，基于资源状态变迁所带来的单向数据流，构成另一个维度，两个维度共同形成一个正交矩阵，这种开发方式有效平衡了指令式与函数式两种设计风格，集两者优势于一身。

ReRest 理念与 REST 有某种相似性。REST 核心含义是用 URL 定位资源，用 HTTP 动词描述操作，它要求服务侧提供的 RESTful API 中，只使用名词来指定资源，原则上不使用动词，“资源”概念可以说是 REST 架构的处理核心，针对资源的操作有 GET, POST, PUT, DELETE 等 HTTP 动词。在 ReRest

框架中，界面可视控件的属性数据视作资源，依据 shadow-widget 实践，“资源（Resource）”则指 React Component 的属性数据。



理解基于 ReRest 的编程，须把握两个重点：**Component 管界面呈现，Resource 管数据流**。前者适用静态思维，更偏指令式风格，后者适用动态思维，更偏函数式风格。

4. 两种思维模式

主流的前端框架一直并存静态与动态两种思维模式，举例来说，Vue 与 Angular 更多采用静态思维模式，**界面是可描述的**，React 更多的用动态思维，**界面是可编程的**，JSX 看上去也是一种表述形式，但它本质是一段 javascript 代码，你很难将它“去编程化”—— 把 JSX 从上下文环境抠出来独立使用，事情将变得毫无意义。

我们不必争论这两种模式孰优孰劣，两者都有显著优点。F.S.菲茨杰拉德曾说：检验一流智力的标准，就是头脑中能同时存在两种相反的想法，但仍保持行动能力。

况 且，在前端开发中，该采用静态思维或动态思维的条件还算清晰。比如开发一个网页，大块功能的界面设计应采用静态思维，比方，在顶部放一个工具条，左侧放导 航，中间放内容；简单界面设计应以静态思维为主，因为界面组件很少动态替换；而应对复杂功能，应以动态思维为主，既然 JS 代码可以控制一切，局部界面用 JSX 定义会很爽。越是动态变化的界面，应该越倾向于用动态的、编程性思维。

Angule 静态思维过重，React 动态思维过重，都不好，Vue 从静态走向动态，易用且适应复杂变化，应该说它正前进在正确道路上，只是，Vue 兼容两种风格并非一开始就统筹规划了，工具复杂性不容易降下来。

5. 从 FRP 到 FLUX，再到 ReRest

ReRest 在前人已有经验基础上，提出更优方法，然后验证，结合实践再调整、优化，React 生态链上系列工具的实践是其中最重要的经验基础。

如果只把 React 看作虚拟 DOM 库，它无疑是一项伟大的发明，作为 DOM 节点对应物，可按任意方式使用它。你完全可以在 React 基础上扩展出像 Vue 那样的指令式描述系统，甚到回退到 jQuery 方式也行（偷偷告诉你一个关键点，用 node.__reactInternalInstance$XXX 能反查 React Component），用 React 搭建 MVVM 也完全可能，React 团队在 SoC（关注度分离）方面分寸把握得很好。

React 工具链普遍遵从浓重的函数式编程风格，从函数式拓展命令式较为容易，但反过来就困难得多。就像许多编程语言，都从 LISP 普系吸收营养，相对来说，函数式编程更反映事物的本原，从此出发更容易理顺具有复杂关系的框架系统。

由于上面原因，ReRest 的实践性探索从 React 开始，而不是 Vue 或其它工具。

5.1 理解 React 的 FRP 机制

FRP 是响应式编程一种范式，由不断变化的数据驱动界面持续更新，界面更新中，或用户操作（如鼠标点击）中又产生新的数据流，再驱动界面更新，如此循环往复。触 发界面更新的数据流也称事件流，因为它的行为方式有一些限定，不是常规数据流动，它至少要求单流向、细粒度、按 tick 触发。

我们不妨把网页界面的更新过程，理解成众多“驱动更新的时间片”的集合，一个时间片称为一个 tick，各 tick 可能前后紧挨着，但两个 tick 之间至少都有“调度间隙”。就像下面 process2 函数紧随 process1 执行，用 setTimeout(process2,0) 延时 0 秒，这两函数之间就产生“调度间隙”了。

```
function process1() {
  console.log('in process1');

  setTimeout( function process2() {
    console.log('in process2');
  },0);
}
```

数据变化导致界面更新（即 React 的 render() 调用），界面更新又触发数据变化，如果没有调度间隙，系统可能陷入无限递归，递归结果必然爆栈。React 的 FLUX 框架首先要让数据单向流动，只要有“调度间隙”区隔，即使数据变化与界面更新无限制的互为触发，都算单向流动。

React 以两种机制保障数据单向流动，一是让 props 只读，二是 setState() 延后一个调度间隙执行。后者好理解，前者“props 只读”是间接生效的，因为 props 与 state 同时决定 Component 界面如何表现，但更改 props 属性只能在父节点的 render() 函数中进行，你得用 ownerComp.setState() 触发父节点再次 render()，所以，不管你怎么用，都会插入“调度间隙”的。

此外，React 要求在 shouldComponentUpdate() 中结合各属性的 immutable 是否变化，判断是否该触发 render 更新。总之，上述机制支持了 FRP 编程以下要求：按时间切片驱动界面更新，各切片保持细粒度，让每次更新最小化、无关联。

5.2 改造双源驱动

由父节点决定如何更新的 props.attr，与节点自己就能决定的 state.attr，两者共同定义 Component 的界面表现，所以 props 与 state 合称为“双源”，只是原生 React 是“隐式双源”，ReRest 框架要把它改造成“显式双源”。

实现原理大致如下：

1. 引入一个与 props.attr 及 state.attr 对等的集合：duals.attr
   该集合中的 attr 把 props.attr 自动记录到 state.attr，通过 duals.attr 读写接口，可等效实现对相应 state.attr 的存取，即：读 duals.attr 等效于读 state.attr，写操作 duals.attr = value 等效于执行 this.setState({attr:value})。
2. 提供 this.defineDual() 让用户手工注册 duals 属性
   系统还将传给标签内置属性（如 name,href,src 等）自动注册为 duals 属性，此举方便了编程，否则大量属性手工编码去注册很麻烦。

3. 由 defineDual() 实现 setter 回调的捆绑
   比如调用 this.defineDual('a',setter) 注册后，对它赋值 this.duals.a = value，将自动触发 setter(value,oldValue) 回调。

经上述改造，更改 Component 自身的 props 就不必绕转到父节点去做了，比如，用类似 comp.duals.name = 'new_name' 语句直接赋值就好。

这么变动将带来一个重大影响：**上层 FLUX 机制可以捋直了做**。 如何实现 FLUX，官方给出了框架建议，React 说我只管虚拟 DOM，如何搭 FLUX 是上层的事，Redux 说，我来管这事，增加 action，增加 reducer，增加 store，不过异步的事你自己解决。什么是 action 呢？就是事件化数据，什么是 reducer 呢？就是事件处理函数，什么是 store 呢？那个 Component 限制了数据读写，还搞不清关联子节点、父节点在哪，自个弄一数据集就是 stroe。结果，Redux 绕了很大一个弯，说把事情解决了，但用户仍报怨写异步很难受呀，这么绕的东西不难受就鬼了！

ReRest 的对策很简单，最直接。事件化数据就是可侦听的 duals 属性嘛，事件处理函数就是 duals 的 setter 回调，理不清父子从属关系，就弄一个 W 树吧，把各节点串起来，用 this.componentOf() 按相对路径（或绝对路径）直接找，至于 store，哪有必要，Component 自身就是 store 嘛！

5.3 资源化

ReRest 尝试让 Web 开发回归事物本原，网页开发主要处理两样东西：开发界面、与服务器交换数据，它与 Delphi、Qt 等 GUI 开发工具不该有太大差别，为什么 React 就不能支持 MVVM 呢？MVC 难以适应标签化的界面表达形式，但用 MVVM 是没问题的。

常规所见即所得开发工具，界面设计的主体过程是：拖入一个样板创建界面组件，选中它对修改某些属性，再拖入样板创建其它组件，设属性，重复操作直至组装出复杂界面。外观设计差不多就这些，剩下工作主要是功能实现，实现类似如何接收键盘输入，如何响应按钮点击等函数定义。

原 生 React 之所以离常规可视化设计很远，主要是 Component 属性成员级别的设计还不够好，少一层可静态依赖的锚点，过早套上高度动态变迁的事件流了，所有东西都动态变化，可视设计是无法支持的。在 ReRest 设计理念中，凡 Component 属性中公开供控制，或供配置的，都应视作 “资源”，“资源” 是静态化的概念，就像 RESTful 要求 URL 要用名词表达资源，动作统一由 HTTP 的 GET, POST, PUT 等表达一样，将 Component 属性 “资源化”，才是问题解决之道。

就 shadow-widget 已有实践而言，**ReRest 所谓的资源，专指 Component 的静态属性（即 comp.props.attr）与双源属性（即 comp.duals.attr）**。

React 对 Component 渲染组装在 render 函数中完成，组装过程是一段 JS 代码，因为 JS 代码可以任意书写，如何组装会非常灵活。而灵活是一把双刃剑，功能虽然强大了，但缺少稳定形态，对建立 MVVM 框架与可视化开发都不利。

ReRest 希望将渲染过程，改造成开发主体依赖于对“资源”的操作，当然，这里的“资源”是动作化了的，也就是，读写资源会自动触发预设的关联动作。换一句话来说，ReRest 想把 render 函数改造成一种固定格式，不必再通过写一段过程代码实施控制，而改成对若干 duals.attr 读写，以此驱动渲染过程的定制处理。

ReRest 对渲染的“资源化”改造过程，本质是将过程控制逻辑，挪到“资源”附属的动作函数中书写。

5.4 渲染临界区

如下示例：

```
01 render() {
02   // 进入渲染临界区
03   渲染临界区的过程处理 ...
04   // 退出渲染临界区
05
06   固定程式的其它 render 处理 ...
07 }
```

“渲染临界区” (Rendering Critical Section) 中的代码（上面 03 行）用来驱动本 Component 各个 duals.attr 附属动作。上面 06 行，让附属动作处理后的结果生效，完成渲染输出。经此改造，用户不必再定义各 Component 的 render() 函数。

在“渲染临界区”执行的代码有特别要求，其一，render 函数因为由 React 内核发起，使用有一些限制，比如 render 过程中再次触发更新、用 ReactDOM.findDOMNode 查找 node 节点等，不过，随着 React 版本优化，这些限制逐渐变少（比如目前版本在 render 函数中调用 findDOMNode 不再报错了）。其二，ReRest 资源的行为函数如何被调用，在临界区中与临界区外有差别，下文马上介绍。

5.5 资源的行为定义

ReRest 区分两类资源，只读资源（即 comp.props.attr）与可写资源（即 comp.duals.attr），对于前者，在 Component 生存周期内，只支持“读”行为，而对于后者，支持读、写、setter 处理、listen 处理共 4 种行为。

这 4 种行为含义如下：

1. 读
   从 props.attr 直接读取，或从 duals.attr 读取由系统返回 state.attr 的值。
2. 写
   对 duals.attr 赋值，系统除了把值赋给 state.attr 外，还触发相应的“setter 处理”与“listen 处理”。
3. setter 处理
   这个 setter 就是 defineDual(attr,setter) 的 setter 回调函数。对于同一 Component 的同一 attr，可以调用多次 defineDual() 注册多个回调函数，给 attr 赋值后，各回调函数依次被调，调用顺序与注册顺序相同。
4. listen 处理
   对一个已存在 comp.duals.attr，可调用 comp.listen(attr,fn) 登记一项侦听，当 attr 值发生变化后，系统会自动调用 fn(value,oldValue)。同一 comp.duals.attr 支持在多处侦听，我们可以为两个（或多个） duals.attr 建立侦听关联，一处更新，其它地方也联动更新。

setter 处理与 listen 处理的适用场合有明显差别，setter 函数只在渲染临界区的处理过程中被调用，listen 函数在触发后（即更改 duals 属性值）必定延后一个“调度间隙”才被执行，所以它必然不在任何节点的渲染临界区内执行。

在同一节点的渲染临界区内，setter 函数可被连续调用，当前节点中不同 duals.attr 的 setter，或同一 attr 的 setter 可串连执行，这意味着，临界区内对当前节点 duals.attr 赋值可能会引发递归重入，各次 setter 调用之间没有“调度间隙”区隔。比如对 comp1.duals.attr1 修改，导致 comp1.duals.attr2 与 comp2.duals.attr3 修改，而 comp1.duals.attr2 修改可能再导致 comp1.duals.attr1 修改，这时对 comp1.duals.attr1 赋值可能导致该属性的 setter 函数递归调用，而引发的 comp2.duals.attr3 更改却是延后一个“调度间隙”的，因为 comp2 的双源属性 setter 函数将在 comp2 的临界区被调用。

setter 与 listen 处理反映了两类资源联动的需求，常规情况下，隔一个“调度间隙”可确保数据单向流动，而特殊情况下，对于紧密相关的资源联动，如果总有“调度间隙”隔着，显然会影响运行效率，上述机制保留了重入式 setter 回调是有意义的。

6. 范式变换

Redux 是 React 生态链中提供 FLUX 框架的一个典型工具，有代表性，接下来介绍范式变换与它有关。

Redux 以“Action”的观点展开设计（其它 FLUX 工具也大都如此），ReRest 则要求以“Resource”的观点展开设计，Action 是动态的动作，Resource 是静态的资源，两者差别可用“非 RESTful”风格与“RESTful”的差别来类比。基于这两种观点的设计存在范式变换关系，下面我们用 Redux 与 shadow-widget 的 FLUX 实现差异为例，展开说明。

6.1 单 Store 变多 Store

拿 Redux 用户手册提到的 Todo 例子来说，增加一条 todo 记录，基于 Action 观点会先设计一个 Action 定义：

```
const ADD_TODO = 'ADD_TODO';

var actTodo = {
  type: ADD_TODO,
  text: 'Build my first Redux app'
};
```

然后，设计一个 reducer 响应这个 Action：

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [ ...state, {
        text: action.text,
        completed: false,
      }];
    // ...
  }
  // ...
}
```

Redux 采用单一的大 Store 结构，ReRest 要求的资源却是小数据，相当于把 Redux 的大 Store 分割成许多小块，一个小块就是一个资源。针对 todo 列表，资源项用 duals.todoList 表示，指定它的初值是空数组。

```
this.defineDual('todoList',null,[]);
```

然后如下代码添加一条 todo 记录，就对等实现了上述 reducer 功能：

```
utils.update(this,'todoList', {$push: [{
  text: 'Build my first Redux app',
  completed: false,
}]});
```

ReRest 的 Store 具备两个特点：

1. 采用多 Store（与 reflux 类似），Store 实体与 Component 重合。
2. 由于数据流动设计针对 Component 下的属性展开，为方便理解，ReRest 的 Store 也可视为双层结构，第一层是 Component 实体，第二层是 Component 下视作 resource 的属性定义，包括 props.attr 与 duals.attr。

Component 下的 resource，本质是数据，与 Store 同属一类，Redux 的 reducer 定义，对应 ReRest 变成 4 种资源行为定义（读、写、setter、listen），而 Redux 的 Action 则弱化成一条操作资源的常规语句。强调一句，Redux 设计用 Action 提纲挈领，ReRest 设计用 Resource 提纲挈领，弱化 Action 是很自然的事，因为相关操作可以随时添加，抓住数据定义才是核心本质。Redux 编程中，给 Action 指定一个常量名，再定义 Action 结构，然后用 switch..case 到处判断 action.type，就没人觉得烦吗？

6.2 数据定义用作事件

侦听一个 duals.attr 后，侦听函数就是事件处理函数，FLUX 框架要求的 Dispatcher 可以简化，比如我们用 duals.receivedData = data 表示接收到外部一条指令，对它赋值即触发侦听它的事件处理函数马上被调。

如果对 duals.receivedData 赋值时，新旧值没有变化，系统将忽略触发侦听函数。要是不想忽略，调整一下数据定义，比如用 duals.receivedData = [data,ex.time()]，加一个时间戳，就保证每次对 duals.receivedData 赋值，都能触发侦听函数了。

尽管 ReRest 聚焦于如何配置资源，duals.attr 的组织形式很简单，却完整支持事件流机制，包括多源头侦听，等全部事件来齐后再触发回调函数，例如：

```
utils.waitAll(comp1,'attr1',comp2,'attr2',
function(value1,value2) {
  // do something ...
});
```

6.3 渲染器

如果一个节点的结构比较稳定，比如它渲染输出的标签名不变，其子节点构成也不变，这时，对该 节点的属性做“资源化”改造很容易。但如果节点结构不稳定，比如，有时单节点，随时变为多层节点，甚至有时输出的标签名也在变。我们还得另寻方法实现资源化定义，解决对策便是“渲染器”。

在 React 中内容组装在 render() 函数进行，通常由 comp.setState() 驱动 render() 函数反复调用。render 是动作，按资源化方式理解，把它变名词，是 rendering，就是渲染器，我们假想 render() 由一个渲染器驱动，渲染器内部用一个计数器（记为 id__）控制渲染刷新，比如：comp.duals.id__ = 2 赋值导致 render() 被调用，运行 comp.setState({attr:value}) 也促使 render() 调用，而且 id__ 会自动取新值。也就是说，每次 render() 运行，渲染器的计数器都会自动取不同值，等效于执行 duals.id__ = value 语句。

按如下方式注册 duals.id__：

```
01 this.defineDual('id__', function(value,oldValue) {
02   // this.state['tagName.'] = 'div';
03   // this.state.attr1 = xxx;
04   // this.duals.attr2 = xxx;
05
06   // prepare jsx_list ...
07   // var jsx_list = [ <SomeTag ...props> ... </SomeTag> ];
08
09   // utils.setChildren(this,jsx_list);
10 });
```

上述渲染器 duals.id__ 的 setter 函数，我们称为 idSetter 函数，这种以“渲染器资源”指代 render() 渲染过程的定义形式，称为 **idSetter 定义**。

在 idSetter 函数中编写代码，等效于在 render() 编程，可以随意组装子节点，然后用 utils.setChildren() 设进去。还可修改当前节点的 state.attr, duals.attr，甚至节点的标签名也可以改，如上面 02 行代码。

借助 duals.attr 的资源化形式（包括 duals.id__ 渲染器），ReRest 实现了 render() 渲染过程的范式变换。现有实践表明，基于 ReRest 的编程与 React 原生方式等效，表达能力近乎等同。

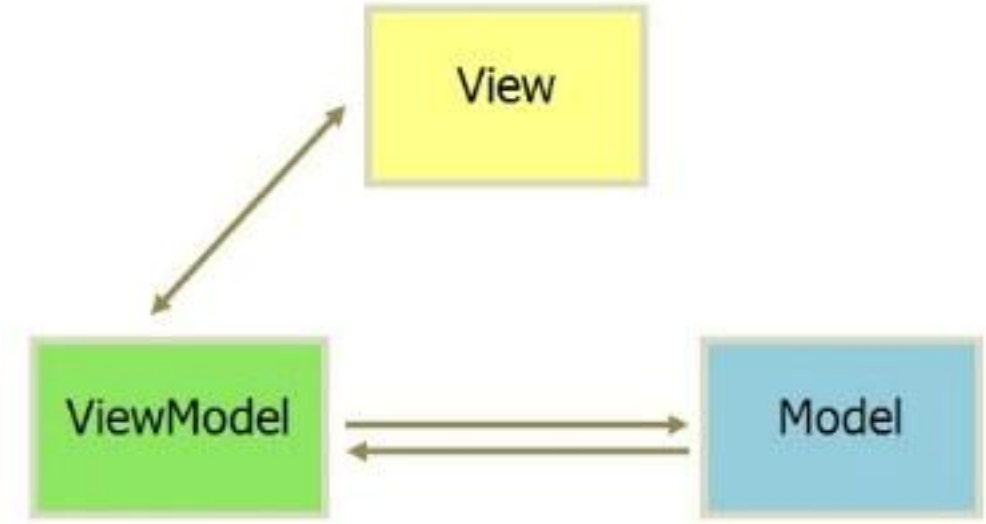
7. 可视化设计与 MVVM 框架

为了支持可视化编程，像 JSX 这种与 JS 代码混写的界面描述方式需要改进，因为界面设计应独立进行。在可视化设计器中，被设计的界面，不能像产品正常运行那样表现功能，鼠标点击在可视化设计器中 表示选择一个构件，接下来要配置它的属性，而对于正式运行的产品，可能是按钮点击、跳转链接点击等，所以，**基于 ReRest 的编程，要求我们改用一种“功能定义可选捆绑”的界面描述方式**。

shadow-widget 采用“转义标签”描述界面，界面的功能实现则在投影类或 idSetter 函数中实施，这两者分开定义。产品正常运行时，在页面导入初始化阶段，两者自动捆绑，让类似 onClick 在 JS 实现的功能定义，与用“转义标签”描述的界面结合。但在可视化设计状态，功能定义缺省被忽略（注：也可以不忽略，但要用特殊方式定义）。

上述“转义标签”，就是用类似 <div $=Ul>desc 的方式描述非行内标签 <Ul>desc</Ul>，或用类似 <span $=Button>title 描述行内标签 <Button>title</Button>。上述“投影类”，与“idSetter 定义”等效，都用来定义 Component 节点的行为。限于本文篇幅，这三项我们不展开介绍。

前面介绍的资源化改造，还支持了 MVVM 框架在 React 技术体系中得以实现，MVVM 要求数据属性能够双向绑定，duals.attr 的 getter/setter 支持了此项要求。如下图，ViewModel 就是投影定义与 idSetter 定义，View 是各 Component 从虚拟 DOM 反映到真实 DOM 的界面表现，而 Model 是数据模型，对于前端开发，Model 通常很简单，一般就是各 Component 的 props.attr 与 duals.attr 规格定义，只有少数需对数据做转换、存盘、备份等特殊处理的，才会额外设计一个 Model 实体。



MVVM 可视为 MVC 框架在前端环境的最佳适配，它也是可视设计的基础。可视化设计的主体过程是在创建 Component 构件后，在线设置它的 props.attr 与 duals.attr 属性值。正因为 MVVM 中 ViewModel 是双向绑定的，属性取值与界面表现才能自动保持一致，这也是 MVC 框架不能适应前端可视化开发，而 MVVM 适应得很好的主要原因。

多说一句，属性取值与界面表现并非简单的直接对应关系，而是属性取值变更要关联一系列变化，须有自动 setter 调用的机制才行。举例来说，设置一个按钮的 duals.disabled 为真，不止是设置 DOM 节点的 disabled 属性，还要让按钮外观变灰，再改换 cursor 配置为 "not-allowed"。

8. 函数式风格

相比 Angular 与 Vue，React 生态链上各工具普遍追求纯正的函数式开发，这既与 React 团队倾向性推动有关，也与 React 技术特征有关，越倾向函数式开发就越适应它的 FLUX 模型。

8.1 函数式是 FRP 编程的天然姻亲

FLUX 框架是 FRP 编程理念（Functional Reactive Programming）的一种实现，一个重要技术路径是，以 CPS 风格（Continuation-Passing Style）应对响应式接续处理。

函数式编程正是 CPS 变换的最佳载体。举一个简单例子，如下提供 Email 输入，当输入内容不合邮箱格式时，右侧图标出现告警图标，底部还有详细提示。

响 应式编程的做法是，用户持续输入文本，内容是否合规随即校验，校验与输入同时进行，校验结果并不打断用户输入。这么理解，手工输入形成持续的数据流，各次 数据都驱动一次校验处理，校验对于输入来说是异步推进的。假定用户输入合法的 Email 地址后，系统用它自动向服务器查询进一步信息，比如得到用户别名、上次登录时间等，这些信息用来辅助下一步表单填写。可以这么编码：

```
01 comp.listen('validation', function(value,oldValue) {
02   if (value == 'success') {
03     var sEmail = comp.duals.email;
04     utils.ajax( {
05       url: '/users/' + encodeURIComponent(sEmail),
06       success: function(data) {
07         // ...
08       },
09     });
10   }
11 });
```

这里 01 行与 06 行调用都是 CPS 风格，实际调用虽是异步，但代码写一起，上下文变量共享。这种代码风格在响应式编程中大量使用，不难看出，函数式是 FRP 编程的必然选择。

8.2 ReRest 中的函数式编程

虽然“资源”是静态化的概念，但 ReRest 对资源的动作定义，仍是可适用 CPS 的函数方式，并未破坏整体函数式风格。简单这么理解，前面所提 ReRest 资源化，实质是提供了 **带锚点的函数式编程**，锚点依附于 Component 实体而存在。所以，在可视设计器中，创建 Component 后，资源锚点（即 props.attr 与 duals.attr）就存在了，这让所见即所得的在线配置因此成为可能。换一种说法，相当于 ReRest 在原设计基础上，插入一排方便思考、易于可视设计的“抓手”。

用来实现 Component 功能定义的投影类，以对象方式编码，属于命令式风格。而与之对等提供功能的 idSetter 定义，是函数式的，如下举例：

```
this.defineDual('id__', function(value,oldValue) {
  if (oldValue == 1) {
    // init process just after all duals-attr registed
  }

  if (value <= 2) {
    if (value == 1) { // init process, same to getInitialState()
      // this.setEvent({$onClick:fn});
      // this.defineDual('attr',fn);
      // ...
    }
    else if (value == 2) { // same to componentDidMount()
      // ...
    }
    else if (value == 0) { // same to componentWillUnmount()
      // ...
    }
    return;
  }

  // other render process ...
});
```

前面已介绍 idSetter 如何组装渲染内容，既然渲染器每次计数变化代表一次渲染调用，那能不能留出几个特殊计数值表达 Component 状态变化呢？idSetter 确实这么做了，比如上面代码，计数值为 0 是初始状态，变为 1 是 Component 的双源属性尚未预备的初始化状态，相当于 getInitialState()，变为 2 是 componentDidMount() 状态，再变回 0 表示马上要返回初态，对应于 componentWillUnmount()。这样，一个完整的 React Class 定义，我们用一个 idSetter 函数就表达了，实现了命令式风格的函数式表达。

idSetter 函数既适应可视化设计时界面描述与功能定义分离，还适应函数式编程。比如当有多层 Component 嵌套时，你可以将里层 Component 的行为定义任意“Lifting State Up”到外层 Component 的函数空间。

8.3 Lifting State Up

采用 JSX 描述界面时，行为定义与虚拟 DOM 描述混在一起，这时仅依赖 props.attr 逐层传递实现数据共享方式，用起来很不方便。React 官方介绍提供一种“上举 State”的解决方案，以输入温度值判断是否达到沸点为例，参见 Lifting State Up。

将上举 State 用在 ReRest 编程中，除了收获 React 官方所提几个好处，还有两项特别收益。其一，原有 React 基于一个过程组织渲染内容，而 ReRest 主体是基于 duals.attr 资源驱动渲染，跨节点 listen 更容易，处理逻辑也更清晰；其二，定义节点行为的 idSetter 是函数，原生 React Class 定义要用 class MyClass extends React.Component {} 方式，层层嵌套使用时，肯定没有 idSetter 用得方便。

如果仔细琢磨“Lifting State Up”方案，大家不难发现，上举 State 解决了部分 Reflux 或 Redux 已支持的需求，被上举共享的 state 其实也是一种 Store 数据。

9. 可视化设计实践

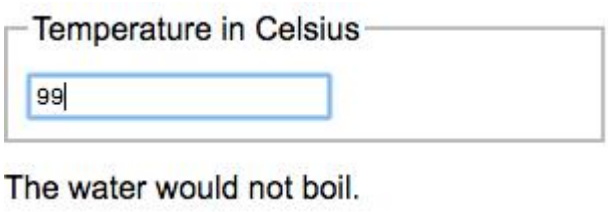
ReRest 编程在 shadow-widget 平台的实践已持续一年多时间，多个项目采用了 ReRest 编程，较典型的有 pinp-blog 与 shadow-bootstrap。在这一年多时间里，shadow-widget 底层库也在 ReRest 实践推动下不断完善，尤其是 idSetter 与可计算表达式方面，优化幅度较大。

在接下来几节，我们补充介绍前文尚未涉及的，与实践相关的若干知识与编程体验。

9.1 正交框架分析模式

先介绍“功能块”Functionarity Block（简称 FB）的概念。一组 Component 节点合起来提供某专项功能，称为一个 FB。以上面提到 Lifting State Up 判断温度是否达到沸点为背景，我们可以开发两个功能块，其一是配置温度格式（config FB），用来配置当前采用摄氏 Celsius 还是华氏 Fahrenheit 作计量单位，其二是计算沸点（calculator FB），提供输入框，判断输入温度是否达到沸点。

后一 FB 的界面如下：



编写 FB 代码块如下：

```
(function() { // functionarity block: calculator

var scaleNames = { c:'Celsius', f:'Fahrenheit' };
var selfComp = null, verdictComp = null;

idSetter['calculator'] = function(value,oldValue) {
  // ...
};

})();
```

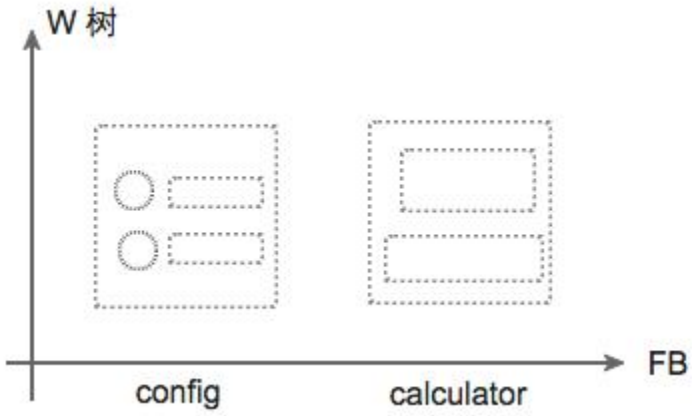
一个 FB 宜用一个函数包裹，主要为了构造独立的命名空间（Namespace），本功能块内共享的变量在这个地方定义，比如上面代码中 scaleNames, selfComp, verdictComp 变量，把命名空间独立出来，也防止 FB 内部使用的变量污染外部全局空间。

既然一个 FB 内某些 Component 很常用，把它定义成 FB 内共享的变量会更方便。

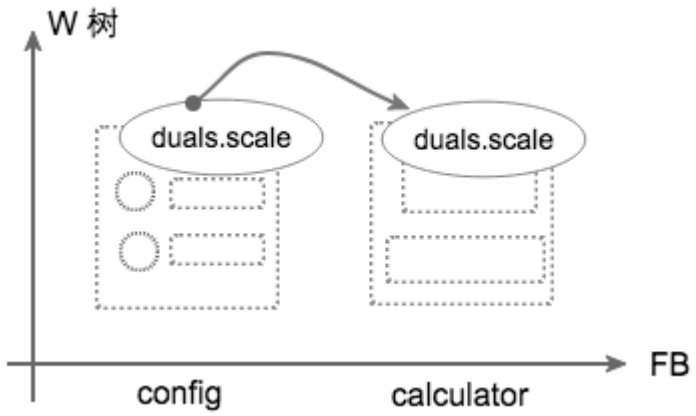
```
var selfComp = null, verdictComp = null;

idSetter['calculator'] = function(value,oldValue) {
  if (value <= 2) {
    if (value == 1) {     // init
      selfComp = this;
      // ...
    }
    else if (value == 2) { // mount
      verdictComp = this.componentOf('verdict');
      // ...
    }
    else if (value == 0) { // unmount
      selfComp = verdictComp = null;
    }
    return;
  }
};
```

产品开发明显可分两个阶段：界面可视化设计与功能实现，在前一阶段，应考虑有哪些 FB 功能块可分解，再针对各 FB 设计界面，按用户使用习惯逐级摆放各构件，各层构件都是 W 树中节点。以上述 config 与 calculator 功能块为例，我们画出 FB 分布为横轴，W 树为纵轴的示例图。



之后进入开发第二阶段：功能实现。这时要解决数据如何在 FB 之间流动，前一功能块 config 配置当前采用哪种温度格式，记录到 duals.scale，后一功能块 calculator 根据自身 duals.scale 配置指示界面如何显示，并决定用 100 度还是 212 度判断沸点，两个 scale 属性的数据流向如下图，我们只需让后一 duals.scale 侦听前一 duals.scale，即实现两者自动同步。



本处举例比较简单，复杂些产品的设计过程大致也是这几个步骤。

总结一下，整个 HTML 页面是一颗 DOM 树，是纵向的（上图纵轴），将这颗树划分为若干 FB 功能块（上图横轴），划分过程主要依据 MVVM 逐步拆解；而处理各功能块之间的横向联系，则以 FRP 思路为主导。这一纵一横的思考方式，我们称为“正交框架”分析模式。

可视化设计时，提供在线配置的最小单位是各 Component 的 props.attr 与 duals.attr，就是 ReRest 所说的“资源”项。而处理各 FB 之间数据如何流动的思考起点，也是这类“资源”项，MVVM 与 FRP 分析的交汇处正是 ReRest 资源化的落脚点。

9.2 Component 属性定位的变化

props.attr 是只读的，用来驱动本节点组织渲染数据，凡涉及状态变化的要用 state.attr，然后同样用 props 驱动子节点的内容更新。现有 React 生态链上各类工具对 props.attr 定位似乎只有两项：一是用作 Component 的入口驱动数据，二是以只读特性保障数据单向流动。

shadow-widget 对 props 与 state 的使用定位做了优化。其一，用 duals.attr 表达一个 Component 对外公开的控制接口，不再建议用 setState() 动态更新“非自身节点”的数据了，相应的 state.attr 也收缩到“只供 Component 内部编程”时使用，类似于用作私有变量。其二，props.attr 当入口驱动数据的定位没变，但刨去转换成 duals.attr 与事件函数，剩下的常规属性在生存周期内被看作常量，在节点 unmount 之前不会变化。

这 两点定位调整的背后有深刻原因，开发理念变了。在 React 支持的虚拟 DOM 库级别，各 Component 所有属性都是对等的，无差别，虚拟节点无需识别各项属性的语法含义，在底层这么处理没问题，因为作为底层库，只聚焦节点虚拟化。但对于上层应用，须区分各 属性的语义，**现实应用中，各节点总具备**

一定“性状”的。比如，你想表达一段文本就创建 <p> 节点；如果创建了 <ul> 节点，也意味着你将在它下面挂入 <li> 节点；如果创建 <input> 节点，通常连带 type 属性也算作“性状”一部分，type='text' 文本框，type='checkbox' 是选项框，两者形态差异巨大，文本框要用 node.value 取输入字串，选项框则用 node.checked。

所以，**上层应用宜将各节点的固有性状，视作生存期内不变的常量，动态变化的纳入 duals，用作控制量**。反之，如果不承认节点固有性状，就不会有 MVVM 框架形式，可视设计器也无法支持通过拖入样板来创建 Component。比如假设你创建的是 <table> 节点，改改属性就把它变成 <ul> 列表，可视设计就没法做了。

shadow-widget 还将 className 分裂成 props.className 与 duals.klass 两个属性，用 className 表达固有类定义，在构件的生存期内不变，用 klass 表达可变的状态量。

9.3 父子结节的单向依赖

我 们先看一个事实，Bootstrap 提供的 50 多个组件中，大部分由多层节点构成，或者使用时要求与其它组件搭配，一个节点表达完整功能的只是少数，而且都只提供简单功能，像 Label、Badge 等，这类组件约占总量十分之一。可以说，现实中的前端开发，父子 Component 组合是常态，是主流。

Shadow Widget 有很多机制让父子节点关联起来，主要有：

1. 把所有存活的构件（已挂载且未卸载）串接成一颗 W 树，树中各节点能方便的互相引用
2. 提供导航面板把多个构件封装起来，形成一组，组内构件用 "./" 相对路径索引
3. 上面提到 FB 功能块的编码，建立块内共用 Namespace，让功能紧密相关的父子节点共享变量
4. 用 $for, $if, $else 等指令描述动态节点，层层嵌套的 callspace 支持在下级节点直接引用上级各层节点的各种属性
5. 支持 $trigger 机制触发相邻节点的动作定义

React 让 props 属性只读的深刻根源是：**解决数据依赖性**。解决依赖性的同时，顺带保证数 据在父子节点之间要单向流动。节点创建有先有后，具有从属关系的两个节点，子节点必然在父节点之后创建，并且 unmount 必在父节点之前，也就是，子节点依赖于父节点而存在，子节点的数据也依赖于父节点的属性先行赋值。所以，React 设计了数据传递要借助 props 逐层进行，原则上属性数据跨层不可见（先撇开 context 不谈，那是补救性设计，官方并不推荐你用）。

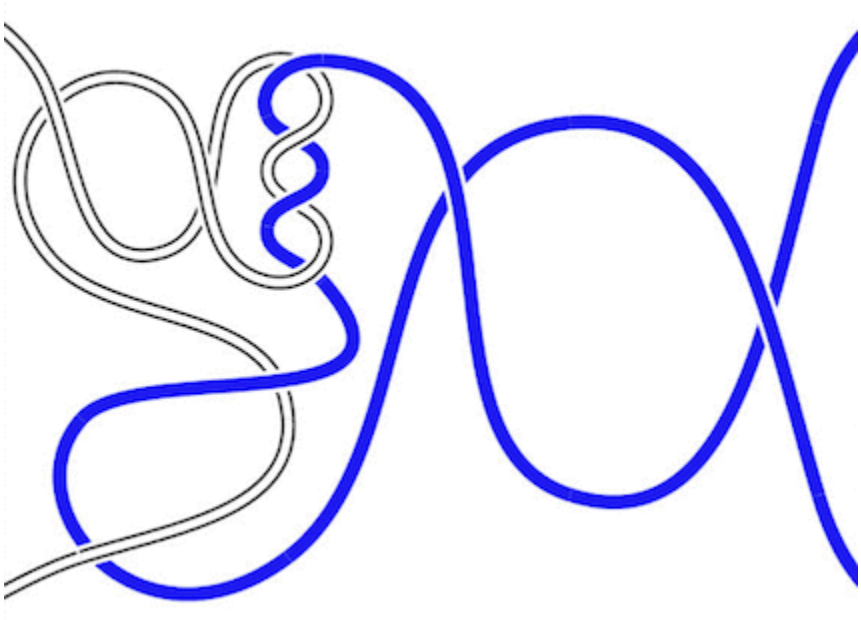
子节点依赖于父节点，但反过来 不是，依赖是单向的，但 React 生态链上诸多工具，都按“隔绝依赖”来处理了，相当于忽略了单向依赖存在。举例来说，比方我们要设计下图 DropdownBtn 与 SplitBtn 两种按钮，两者功能基本一样，外观有差别，怎么实现呢？

外层节点用 this.isSplitBtn 指示按钮是否为 SplitBtn，然后里层节点根据 isSplitBtn 取值，绘制不同外观的按钮。如果按“隔绝依赖”来处理，只能借助 props 属性层层传递 isSplitBtn，隔了几层就传几层；如果按“单向依赖”来处理，里层哪个节点需要要区分 isSplitBtn，就往上层查找，看看 props.isSplitBtn 取什么值。这两种处理方式差别很大，前者忽略了主从构件的天然关系，以暴露接口的代价实现功能，把无关节点都牵扯进当来传手，就像打排球的一传、二传、三传，当功能组合较多时，显得很绕。

从子节点向上查找，分析一级（或多级）父节点的属性特点，从而确定它自身所处的场景，进而让当前节点应对不同场景表现不同功能。我们管这种场景推导过程叫**“场景自省”**，如上介绍，向上追溯的“场景自省”是安全的，因为子节点若存活，父节点必然还存活，反过来从父节点查子节点则不行。

9.4 不绕弯也是生产力

现有 React 生态链上诸多主流工具都很绕，不像 shadow-widget 那么直接，主要表现以下几个方面。



其一，主流工具普遍忽视父子节点的主从关系是隐含丰富信息的，把所有 Component 摆同等位置来解决跨节点数据传递问题。

源头在于 Facebook 官方的 FLUX 框架有缺陷，FLUX 在虚拟 DOM 的上层实现，但它继续无视 Component 属性带语义特性，都无差别对待。借助 Dispatcher 分发 Action，构造独立的 Store，统一处理各 Action 消息。另设 Store 与 Action 另行驱动的过程，相当于换个地方重建各节点的场景信息。

其二，这些工具普遍过于依赖函数式风格，静态化概念只停留在 Component 层面，没往下探一层。各 Component 互相关联，形成网格，这网格直接用函数式编程去编织了。因为代码量没减，该做的事情一件不少，重建场景的各个处理环节又衍生不少概念，比较绕。基于 ReRest 的编程则将 Component 下的属性视作资源，把静态化概念深入一层，然后在“资源粒子”层面，用函数式风格编织网格。这样更直接了当，也符合开发者思考习惯。

shadow-bootstrap 项目按 ReRest 理念去实践的，该项目核心功能是将 Bootstrap 往 shadow-widget 平台适配。与之类似，业界还有一个知名项目 react-bootstrap， 把 Bootstrap 往 React 适配。这两项目的功能对等，封装的组件几乎能一一对应，如果对比两者源码，shadow-bootstrap 明显简洁许多，react-bootstrap 不容易读，绕来绕去的。最终代码 minify 后，前者 103 Kb，而后者 213 Kb，整整多出一倍。前者开发只用一个多月，后者远不止这个投入，当我们的框架没那么绕时，生产力是大幅提升的。

10. 总结

长期以来 GUI 开发工具与 Web 前端工具是两条独立主线，并行发展。MFC、Delphi、VB、WxWidget、Qt 等归入前者，没人将前端开发也视作 GUI 一类，不过，大概没人否认前端开发主要工作是设计图形用户界面（Graphical User Interface），就目的而言，前端开发无疑也是 GUI 开发。

这两条主线靠拢发展的时代已来临，虚拟 DOM 技术结合 FRP 理念，再结合 ReRest 资源化改造，基于 MVVM 框架 —— 对应主流 GUI 工具的 MVC —— 的可视化开发已经走通了。ReRest 方法论尝试让前端开发回归可视化 GUI 工具序列，其实践已在 shadow-widget 平台走出第一步，希望这一步对 Web APP 与 Native APP 逐步融合的发展提供有益经验。