1. View.js是什么

View.js是一个专门用于开发移动端H5应用的底层框架。其核心理念是“视图”，并提供而且仅提供视图相关的API和事件监听。

View.js将“视图”定义为：用户在设备上看到的一页内容。出于复杂类型数据（如：回调方法，json对象和DOM元素等）传递，以及动画效果的开发等需要，View.js将视图以“一个视图对应HTML中的一个片段，多个视图共存于同一个HTML”的方式来组织。如下所示：

*index.html*

<!DOCTYPE HTML>

<html>

<head>...</head>

<body>

<section id = "view1" data-view = "true" data-view-title=”View title 1”>

...

</section>

<section id = "view2" data-view = "true" data-view-title = “View title 2”>

...

</section>

</body>

</html>

由于View.js只专注于视图相关的操作，并不涉及其它领域，因此View.js能够很好地和其它优秀的框架一起协同工作，如Hammer、Swiper、 jQuery和zepto等。因为这些不同的框架或库解决的是不同范畴的问题，所以不会出现冲突，因而可以放心使用。[[1]](#footnote-1)

1. 如何使用View.js

在html中引入view-{{version}}.js和view-{{version}}.css即可。（其中{{version}}为版本号）

1. View.js的适用范围

View.js适用于偏交互性质的网页应用，如：商城、物业管理、股票分析等。对于宣传、广告性质等页面较少，动画较多的H5，使用View.js可能就显得略微笨重了。

1. View.js特性 – 事件驱动

所有视图均具备事件驱动特性。

视图原生支持的事件包括：

1. View事件：beforechange - 视图将要切换

监听样例：

View.on(“beforechange”, function(e){});

1. View实例事件：ready - 视图就绪

监听样例：

View.ofId(“myView”).on(“ready”, function(e){});

注：特定视图的ready事件只会在视图第一次进入时触发一次。视图第二次进入后不会再触发

1. View实例事件：beforeenter - 视图将要进入

监听样例：

View.ofId(“myView”).on(“beforeenter”, function(e){});

1. View实例事件：enter - 视图进入

监听样例：

View.ofId(“myView”).on(“enter”, function(e){});

1. View实例事件：afterenter - 视图进入完成

监听样例：

View.ofId(“myView”).on(“afterenter”, function(e){});

1. View事件：afterchange - 视图切换完成

监听样例：

View.on(“afterchange”, function(e){});

除了视图原生支持的事件外，开发者还可以根据自己需要，发起自定义事件并为这些事件添加监听器。如：

var view = View.ofId(“myView”);

view.on(“myevent”, function(e){

view.logger.debug(“Event name: {}, event data: {}”, e.name, e.data);

});

//…

view.fire(“myevent”, {a: 1});//-> 0713 10:20:54 [View#myView]: Event name: null, event data: {"a":1}

开发者在创建自定义事件时，需注意事件名的可读性以及见名知意的直观性。虽然任何形式的命名都能驱动程序的正常工作，但处于工程的可维护性，并不建议这样做。

开发者在发起自定义事件时，可以为事件附加任意类型的数据。对应的事件监听器在捕获对应的事件时，可以通过“data”属性获取附加的数据。

1. View.js特性 – 设备操作系统检测

View组件在加载完成后，会自动识别当前的移动设备类型，并将识别结果以名称为：“data-view-os”DOM属性的方式附加至DOM树中的html结点上。如：

<html data-view-os=”android”> <!—代表安卓设备 -->

<html data-view-os=”ios”> <!—代表苹果设备 -->

<html data-view-os=”wp”><!—代表windows phone设备 -->

开发者可以根据该属性撰写不同的CSS样式。

1. View.js特性 – 日志输出

View.js内置了格式化的日志输出组件，以供程序调测使用。所有视图实例均含有日志实例，如：

var view = View.ofId(“myView”);

view.logger.debug(“Hello, view.js”);//-> 0713 11:42:02 [View#myView]: Hello, view.js

view.logger.warn(“Hello, {}, {}”, 123, true, false); //-> 0713 11:42:44 [View#SC\_home-page]: Hello, 123, true

console.log(view.logger.getName();)//->View#myView

其中“{}”为占位符，一一对应于第二个参数开始的多个参数。可以使用“\\”转义占位符，以输出“{}”。如：

view.logger.log(“\\{}”);//-> 0713 11:43:54 [View#myView]: {}

View日志组件共支持4个日志级别：debug, info, warn, error，以及1个普通的log。

开发者可以通过调用API：View.Logger.ofName(loggerName)来创建一个自定义的日志输出器。如：

var logger = View.Logger.ofName(“WindowUtil”);

logger.debug(“test”); //-> 0713 11:56:24 [WindowUtil]: test

1. View.js特性 – 视图配置

通常情况下，一个视图的功能表现是固定的。但在部分场景下，视图可能需要同时包含多种功能表现，并需要以配置的方式指定要启用的功能表现。在这种情况下，就可以借助视图配置完成。

视图配置是一个集合，可以根据程序需要包含多个配置项。每个配置项由“配置项名称”和“配置项取值”两部分组成。除此之外，为方便开发者，视图配置项还提供有apply()及reflectToDom()等方法。如：

var view = View.ofId(“myView”);

var a = view.config.get(“config-item”);

var b = view.config.get(“config-item”);

console.log(a === b);//-> true

a.setValue(123);//-> set value to 123

console.log(b.getValue());//->123

b.apply();//-> nothing happes

b.setApplication(function(v){

console.log(123, v);

});

b.setValue(“asdf”);//-> nothing happens

console.log(b.getValue());//-> 123

b.setValue(“asdf”, true);//-> set value to ‘asdf’

b.apply();//-> 123, 123

开发者可以通过reflectToDom()方法将配置项取值体现到DOM中，从而可以在CSS层面控制元素的表现。reflectToDom()方法被调用时，将配置项以“data-viewconfig\_[configName]=[configValue]”的形式添加至视图的DOM结点上。如：

View.ofId(“myView”).config.get(“show-header”).setValue(true).reflectToDom();

1. View.js特性 – 视图上下文

为了满足工程多个脚本文件之间共享变量的需要，以及降低全局环境下变量被污染的可能，View.js为每个视图提供了数据上下文，以供开发者存取数据。如：

*a.js*

var view = View.ofId(“myView”);

var getOrderId = function(){

return “ORD001”;

};

view.context.set(“getOrderId”, getOrderId);

*b.js*

var view = View.ofId(“myView”);

var getOrderId = view.context.get(“getOrderId”);

console.log(getOrderId());//-> ORD001

view.context.clear();

console.log(view.context.get(“getOrderId”));//->undefined

不同视图拥有不同的上下文，不同视图的上下文中可以存储相同名称的数据。

1. View.js特性 – 自定义初始化器

默认情况下，引用了View.js的页面会在网页加载就绪（DOMContentLoaded）后自动执行初始化动作。但开发者可以通过调用API： View.setInitializer(initializerFunc[, execTime])提供自定义的初始化器延迟执行视图的初始化动作。如：

View.setInitializer(function(init){

//…

init();//->执行初始化动作

}, “rightnow”);

视图的初始化动作包括：

1. 调用视图初始化监听器
2. 在DOM中标识识别的操作系统
3. 扫描文档，遍历视图定义
4. 确定默认视图
5. 添加视图标题自动设置支持
6. 使能data-view-rel属性
7. 调用视图就绪监听器
8. 呈现指定视图
9. View.js特性 – 布局

由于不同视图呈现的内容不同，因此所执行的布局动作也不同。大部分情况下，视图可以在没有脚本的情况下，通过CSS完成页面布局。但少数情况下，需要借助脚本完成页面的动态布局，如元素高度的动态计算等。

View.js假定所有视图都需要执行布局动作，且为简化开发，将在视图每次进入前自动执行开发者指定的布局动作（如果视图第一次进入，则在ready事件触发前执行）。

开发者可以通过API：view.setLayoutAction(actionFunc, ifLayoutWhenLayoutChanges) 设定布局动作。如：

var viewId = "myView";

var view = View.ofId(viewId);

var headerObj = view.find("header");

var bodyObj = view.find(".body"),

btnObj = view.find(".btn");

view.setLayoutAction(function(){

var totalHeight = View.layout.getLayoutHeight();

var height = totalHeight - headerObj.offsetHeight - btnObj.offsetHeight;

bodyObj.style.height = height + "px";

}, true);

在布局功能的设计上，View.js假定不同分辨率下所需要执行的布局动作是不同的。

View.js支持分别为：移动设备的竖屏模式、移动设备的横屏模式、平板设备的竖屏模式、平板设备的横屏模式、PC设备的竖屏模式，PC设备的横屏模式几种场景执行不同的布局动作。开发者只需为不同场景提供不同的布局动作即可，View.js自动完成设备类型及设备方向的识别并调用对应的布局动作。

默认情况下，View.js假定移动设备的竖屏模式、移动设备的横屏模式、平板设备的竖屏模式和平板设备的横屏模式表现一致，均为：“宽度渲染为浏览器宽度，高度自动”。

当在PC上浏览时，View.js默认将页面以iPhone5的320 \* 568分辨率渲染。亦即，PC横屏浏览时，根据浏览器高度动态计算可用高度，并根据iPhone5的分辨率计算宽度，然后将界面水平居中呈现；PC纵屏浏览时，将其以移动设备的竖屏模式对待。

开发者可以通过调用API：View.layout.setExpectedWidthHeightRatio(ratio)设定PC横屏浏览时，渲染的纵向效果的宽高比，如：

/\* Layout as iPhone6+ \*/

View.layout.setExpectedWidthHeightRatio(414 / 736).init({

autoReLayoutWhenResize: true,

layoutAsPcLandscape: function(width, height){

document.body.style.cssText = "width: " + width + "px; height: " + height + "px; margin: 0 auto;";

},

layoutAsMobilePortrait: null,

layoutAsMobileLandscape: null,

layoutAsTabletLandscape: null,

layoutAsTabletPortrait: null,

layoutAsPcPortrait: null,

}).doLayout();

值得注意的时，在这种渲染模式下，如果界面含有position: fixed绝对定位的样式表，表现结果可能与期望并不相符。

1. View.js特性 – 浏览器标题自动设置

开发者可以为视图的HTML结点设置单独的视图标题，从而达到“视图进入后，浏览器标题自动更新为当前视图标题；视图离开后，浏览器标题自动恢复为默认标题”的目的。如：

<section id = “myView” data-view = “true” data-view-title = “My view title”>

<header></header>

<div class = “body”></div>

<footer></footer>

</section>

注：默认标题在View.js加载时通过自动检测浏览器标题获取。如果浏览器标题通过脚本延迟设置，则会出现默认标题为空的情况。此时开发者可以通过调用API：View.setDocumentTitle通知View使用给定的字符串赋值默认标题。

1. View.js特性 – 设定视图是否允许直接访问

在技术上，View.js利用HTML5的History API，借助地址栏hash完成视图之间的导向和路由。由此造就的现象，是在同一html下的不同视图之间进行切换时，浏览器地址栏的hash部分会发生变化。如：

切换前的地址栏：

<http://www.mydomain.com/html/index.html#myView>

执行如下切换动作后：

View.navTo(“anotherView”)

<http://www.mydomain.com/html/index.html#anotherView>

其中hash部分代表的是当前呈现的视图ID。

通常情况下，这没什么问题。但对于操作步骤有先后顺序要求的应用而言，当用户在操作时把地址分享给他人，或借助其它手段传播用户当前的URL地址时，就会出现“上一步内容尚未填写或校验就打开了后续步骤界面”的现象。

此时，开发者可以通过如下手段解决该问题：

1. 使用data-view-directly-accessible属性，并设置取值为”false”设定依赖上一步骤界面的视图不能直接访问；

对应的JS API：

View.setIsDirectlyAccessible(isDirectlyAccessible) – 配置全局；

View.ofId(“myView”).setAsDirectlyAccessible(isDirectlyAccessible) – 配置单个视图；

1. 使用data-view-fallback属性设定依赖上一步骤界面的视图的回退视图（可选）；

对应的JS API：View.ofId(“myView”).setFallbackViewId(viewId)

1. 使用data-view-default属性设置第一步的界面为默认视图

对应的JS API：View.setAsDefault(viewId)

如：

*index.html*

<!DOCTYPE HTML>

<html>

<head>

<meta name = "viewport" content = "user-scalable = no, initial-scale = 1, maximum-scale = 1, minimum-scale = 1, width = device-width" />

</head>

<body>

<section id = “step1” data-view = “true” data-view-title = “Step 1” data-view-default></section>

<section id = “step2” data-view = “true” data-view-title = “Step 2” data-view-fallback = “step1”></section>

</body>

</html>

如此一来，当用户打开的地址中指定的视图不能直接访问时，View.js将自动查找该视图的回退视图（多层次查找，直到找到的视图是可以直接访问的）。如果回退视图不存在，则最终使用默认视图呈现界面，同时更新地址栏中的hash为最终呈现的视图的视图ID。

同时，为简化开发，View.js支持以“设定所有视图是否可以直接访问 + 设定单个视图是否可以直接访问”的方式设定视图的表现。如果特定视图没有设置是否可以直接访问，则使用全局配置代替。如果全局配置也没有设置，则以“视图不允许直接访问”方式对待。

1. View.js特性 – 回退视图

当设置了特定视图不能直接访问时，开发者可以通过设定该视图的回退视图达到呈现该视图的入口视图等效果。例如：开发者可以设定“个人中心”视图：profile可以直接访问，而“个人中心”下的“账户设置”：setting等视图不能直接访问，并设定“个人资料”视图的回退视图为“个人中心”。这样，即时用户打开的界面地址体现的是“账户设置”界面，如：

http://www.mydomain.com/html/index.html#setting

，浏览器在页面加载完毕后呈现的也将是个人中心界面，且页面地址会更新为：

http://www.mydomain.com/html/index.html#profile

在技术实现上，当用户打开的地址中指定的视图不能直接访问时， View.js将自动查找该视图的回退视图（多层次查找，直到找到的视图是可以直接访问的）。如果回退视图不存在，则最终使用默认视图呈现界面，同时更新地址栏中的hash为最终呈现的视图的视图ID。

开发者可以通过JS API：setFallbackViewId(viewId)，也可以通过视图HTML节点上的data-view-fallback属性设置回退视图ID。如：

*JS*

View.ofId(“setting”).setFallbackViewId(“profile”)

*HTML*

<section id = “setting” data-view = “true” data-view-fallback = “profile”>

</section>

1. View.js特性 – 多种方式进行视图跳转

View.js支持如下几种方式进行视图跳转：

1. A链接，如：

<a href = “http://www.mydomain.com/html/index.html#goods-detail”>商品详情</a>

此时，要求视图：goods-detail为默认视图或可以直接访问。

1. data-view-rel属性，如：

<span data-view-rel = “@index.html#goods-detail”>商品详情</span>

<span data-view-rel = “@http://www.mydomain.com/index.html#goods-detail”>商品详情</span>

<div data-view-rel = “profile” data-view-rel-type = “change”>个人中心</div>

其中”@”符号用于告诉View.js进行页面跳转访问。

此时，要求视图：goods-detail为默认视图或可以直接访问。

1. JS API：View.navTo，如：

View.navTo(“setting”, {params: {a: 1}})

1. JS API：View.changeTo，如：

View.changeTo(“setting”, {params: {a: 1}})

其中，1)和2)两种方式无法传参；3)和4)两种方式可以传参。

View.navTo和View.changeTo的差别之处，在于是否向浏览堆栈中添加记录。使用View.navTo执行视图A向视图B的跳转时，用户可以在视图B中通过物理返回键返回A；使用View.changeTo执行视图A向视图B的跳转时，A的浏览历史将被B所替代，用户在视图B中无法通过物理按键返回A。

1. View.js特性 – 识别浏览器的前进和后退
2. View.js特性 – 视图跳转多种方式传参
3. View.js特性 – 视图切换动画
4. View.js特性 – DOM元素获取

1. View.js会在window上附加名称为”View”的对象。如果您引用的其它框架或库也有这样的名称，那可能会有冲突出现。 [↑](#footnote-ref-1)