[前端监控和前端埋点方案设计](https://segmentfault.com/a/1190000015864670)

在线上项目中,需要统计产品中用户行为和使用情况，从而可以从用户和产品的角度去了解用户群体，从而升级和迭代产品，使其更加贴近用户。用户行为数据可以通过前端数据监控的方式获得，除此之外，前端还需要实现性能监控和异常监控。性能监控包括首屏加载时间、白屏时间、http请求时间和http响应时间。异常监控包括前端脚本执行报错等。

实现前端监控有三个步骤：前端埋点和上报、数据处理和数据分析。本文针对整个前端监控，设计适用的方案。本文的主要内容分为：

* 为什么需要前端监控
* 常用前端埋点方案总结
* 前端埋点方案选型和前端上报方案设计
* 前端监控结果可视化展示系统的设计

原文的地址，在我的博客中：[https://github.com/forthealll...](https://github.com/forthealllight/blog/issues/23)

如有帮助，您的star是对我最好的鼓励～

一、为什么需要前端监控

前端监控的目的是：

***获取用户行为以及跟踪产品在用户端的使用情况，并以监控数据为基础，指明产品优化的方向***。

前端监控可以分为三类：数据监控、性能监控和异常监控。下面我们来一一的了解。

(1)数据监控

数据监控，顾名思义就是监听用户的行为。常见的数据监控包括：

* PV/UV:PV(page view)，即页面浏览量或点击量。UV:指访问某个站点或点击某条新闻的不同IP地址的人数
* 用户在每一个页面的停留时间
* 用户通过什么入口来访问该网页
* 用户在相应的页面中触发的行为

统计这些数据是有意义的，比如我们知道了用户来源的渠道，可以促进产品的推广，知道用户在每一个页面停留的时间，可以针对停留较长的页面，增加广告推送等等。

(2)性能监控

性能监控指的是监听前端的性能，主要包括监听网页或者说产品在用户端的体验。常见的性能监控数据包括：

* 不同用户，不同机型和不同系统下的首屏加载时间
* 白屏时间
* http等请求的响应时间
* 静态资源整体下载时间
* 页面渲染时间
* 页面交互动画完成时间

这些性能监控的结果，可以展示前端性能的好坏，根据性能监测的结果可以进一步的去优化前端性能，比如兼容低版本浏览器的动画效果，加快首屏加载等等。

(3)异常监控

此外，产品的前端代码在执行过程中也会发生异常，因此需要引入异常监控。及时的上报异常情况，可以避免线上故障的发上。虽然大部分异常可以通过try catch的方式捕获，但是比如内存泄漏以及其他偶现的异常难以捕获。常见的需要监控的异常包括：

* Javascript的异常监控
* 样式丢失的异常监控

二、常用前端埋点方案总结

在上一节中介绍了前端监控的作用，那么如何实现前端监控呢，实现前端监控的步骤为：前端埋点和上报、数据处理和数据分析。首要的步骤就是前端埋点和上报，也就是数据的收集阶段。数据收集的丰富性和准确性会影响对产品线上效果的判别结果。

目前常见的前端埋点方法分为三种：代码埋点、可视化埋点和无痕埋点。下面一一介绍每一种埋点的方法。

(1) 代码埋点

代码埋点，就是以嵌入代码的形式进行埋点，比如需要监控用户的点击事件，会选择在用户点击时，插入一段代码，保存这个监听行为或者直接将监听行为以某一种数据格式直接传递给server端。此外比如需要统计产品的PV和UV的时候，需要在网页的初始化时，发送用户的访问信息等。

代码埋点的优点：

* 可以在任意时刻，精确的发送或保存所需要的数据信息。

缺点：

* 工作量较大，每一个组件的埋点都需要添加相应的代码

(2)可视化埋点

通过可视化交互的手段，代替代码埋点。将业务代码和埋点代码分离，提供一个可视化交互的页面，输入为业务代码，通过这个可视化系统，可以在业务代码中自定义的增加埋点事件等等，最后输出的代码耦合了业务代码和埋点代码。

可视化埋点听起来比较高大上，实际上跟代码埋点还是区别不大。也就是用一个系统来实现手动插入代码埋点的过程。

缺点：

* 可视化埋点可以埋点的控件有限，不能手动定制。

(3)无埋点

无埋点并不是说不需要埋点，而是全部埋点，前端的任意一个事件都被绑定一个标识，所有的事件都别记录下来。通过定期上传记录文件，配合文件解析，解析出来我们想要的数据，并生成可视化报告供专业人员分析因此实现“无埋点”统计。

从语言层面实现无埋点也很简单，比如从页面的js代码中，找出dom上被绑定的事件，然后进行全埋点。

无埋点的优点：

* 由于采集的是全量数据，所以产品迭代过程中是不需要关注埋点逻辑的，也不会出现漏埋、误埋等现象

缺点：

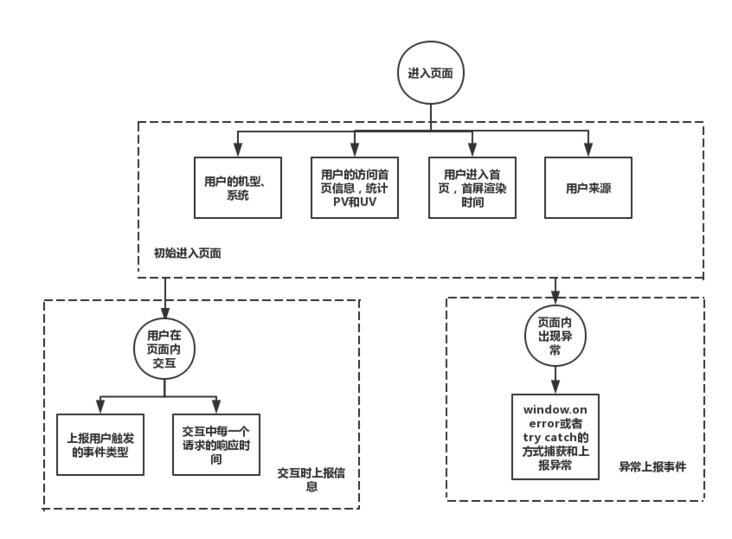
* 无埋点采集全量数据，给数据传输和服务器增加压力
* 无法灵活的定制各个事件所需要上传的数据

三、前端埋点方案选型和前端上报方案设计

第一章中介绍了前端所需要监听的信息，在第二章中介绍了前端埋点的常见方式，本文来根据需求，来定制我们的埋点和上报方案。

(1)监控数据

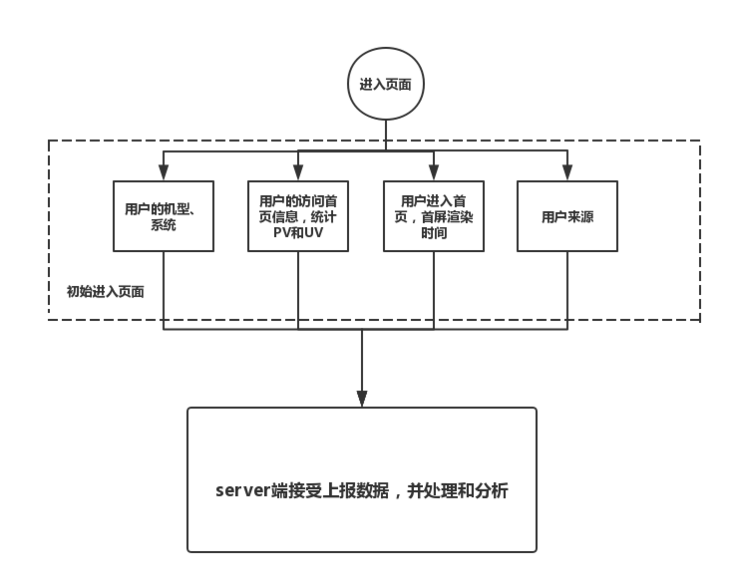
首先我们需要明确一个产品或者网页，普遍需要监控和上报的数据。监控的分为三个阶段：用户进入网页首页、用户在网页内部交互和交互中报错。每一个阶段需要监控和上报的数据如下图所示：



(2)埋点方案

在实际项目中考虑到上报数据的灵活定制，以及减少数据传输和服务器的压力，在所需埋点处不多的情况下，常用的方式是代码埋点。

以用户进入首页为例，我们在首页渲染完成后会发送事件类型和类型相关的数据给server端，告知首页的监控信息。



(3)上报周期和上报数据类型

如果埋点的事件不是很多，上报可以时时进行，比如监控用户的交互事件，可以在用户触发事件后，立刻上报用户所触发的事件类型。如果埋点的事件较多，或者说网页内部交互频繁，可以通过本地存储的方式先缓存上报信息，然后定期上报。

接着来确定需要埋点上报的数据，上报的信息包括用户个人信息以及用户行为，主要数据可以分为：

* who: appid(系统或者应用的id),userAgent(用户的系统、网络等信息)
* when: timestamp(上报的时间戳)
* from where: currentUrl(用户当前url)，fromUrl(从哪一个页面跳转到当前页面)，type(上报的事件类型),element(触发上报事件的元素）
* what: 上报的自定义扩展数据data:{},扩展数据中可以按需求定制，比如包含uid等信息

上报数据的对象为：

{

----------------上报接口本身提供--------------------

currentUrl,

fromUrl,

timestamp,

**userAgent:**{

os,

netWord,

}

----------------业务代码配置和自定义上报数据------------

type,

appid,

element,

**data:**{

uid,

uname

}

}

(4)埋点和上报举例

我们以上报首屏加载事件为例，DOM提供了document的DOMContentLoaded事件来监听dom挂载，提供了window的load事件来监听页面所有资源加载渲染完毕。

<script **type**="text/javascript">

var start=**Date**.now();

document.addEventListener('DOMContentLoaded', **function**() {

**fetch**('some api',{

**type**:'dom complete',

data:{

domCompletedTime:**Date**.now()-**start**

}

})

});

**window**.addEventListener('load', **function**() {

**fetch**('some api',{

**type**:'load complete',

data:{

LoadCompletedTime:**Date**.now()-**start**

}

})

});

</script>

(5)前端埋点系统的前后端通信加密

在上报数据的前后端通信中，需要和server端协商加密机制，利用 OpenSSL库来实现的加密，OpenSSL已经是一个广泛被采用的加密算法。前端可以采用node的crypto模块。

首先来看hash算法，crypto.createHash() 来创建一个Hash实例，可利用的hash算法如下：

* md5
* sha1
* sha256
* sha512
* ripemd160

以sha256算法加密为例：

**const** str="123445";*//需要加密的字段*

**const** hash=crypto.createHash('sha256');*//指定加密算法*

hash.update(str); *//通过算法加密相应的字段*

**const** result=hash.digest('hex');*//转化成十六进制*

四、前端监控结果可视化展示系统的设计

当后端得到前端上报的信息之后，经过数据分析和处理，需要前端可视化的展示数据分析后的结果。

可以在开源中后台系统ant-design-pro的基础上进行二次开发，首先要明确展示信息。展示的信息包括单个用户和整体应用。

对于单个用户来说需要展示的监控信息为：

* 单个用户，在交互过程中触发各个埋点事件的次数
* 单个用户，在某个时间周期内，访问本网页的入口来源
* 单个用户，在每一个子页面的停留时间

对于全体用户需要展示的信息为：

* 某一个时间段内网页的PV和UV
* 全体用户访问网页的设备和操作系统分析
* 某一个时间段内访问本网页的入口来源分析
* 全体用户在访问本网页时，在交互过程中触发各个埋点事件的总次数
* 全体用户在访问本网页时，网页上报异常的集合

删选功能集合：

* 时间筛选：提供今日（00点到当前时间）、本周、本月和全年
* 用户删选：提供根据用户id删选出用户行为的统计信息
* 设备删选：删选不同系统的整体展示信息

埋点即监控用户在应用表现层的行为，于产品迭代而言至关重要。埋点数据分析是产品需求的 来源，检验功能是否达预期的 佐证。前端较服务端更接近用户，本小白将在此对前端埋点统计方案述说一二。

采集埋点数据可做如下分析（以百度统计为例）：

将 用户属性、用户行为 转化各类可视化图表：

不同产品对数据的关注角度不同，可按需采集。如信息流产品对停留时长的关注度更高（统计页面访问 & 跳出时间），商城类较注重“复购率”（统计新老用户），广告类更追求最大限度。

埋点统计通常分两类：

页面访问量统计

功能点击量统计

页面访问量统计

页面访问量统计通常分两类：

PV：页面访问人次

UV：页面访问人数

页面访问量，并非仅仅取决于其内容吸引力，影响因素包含入口 外观、位置、深度 等等（在此不考虑刚需）。入口外观属 UI 设计范畴，入口位置可通过分析用户点击热力图调整，入口深度可通过分析用户访问路径调整。

用户点击 热力图 形如：

将核心页面入口置于热力图红色区域？

采集页面加载 from、to 以获知用户访问路径：

分析可知用户普遍 访问深度、每一深度 & 每一页面的 流失率 等，依照结果调整核心页面入口源、入口深度？

页面访问量，也并非仅仅取决于产品设计。假若 PV 稳定的页面访问量 爆跌，便需考虑其加载成功率了（或许是枚技术 bug）。

前端如何实现全局 PV 统计，以 Vue 应用为例。

方案一

通过在入口文件 index.js 全局定义 Router.beforeEach：

import App from './app'

import Router from './router'

Router.beforeEach((to, from, next) => {

    App.logEvent({

        type: 'visit',

        name: to.name,

        time: new Date().valueOf(),

        params: {

            from: {

                name: from.name,

                path: from.path,

                query: from.query

            },

            to: {

                name: to.name,

                path: to.path,

                query: to.query

            }

        }

    })

    next()

})

停留时长可通过 （跳转页 time - 当前页 time） 获知，但关闭应用时如何统计？监听应用关闭 onbeforeunload: https://developer.mozilla.org/zh-CN/docs/Web/API/Window/onbeforeunload 和 onunload: https://developer.mozilla.org/zh-CN/docs/Web/API/Window/onunload ？

其中 App.logEvent 为自定义 Vue 插件 App 中的 method，用于向服务器发起 埋点上报请求：

import Request from './utils/request'

const App = {

    // ...

    logEvent (opts) {

        Request({

            url: '/log/event',

            method: 'POST',

            data: {

                type: opts.type,

                name: opts.name,

                time: opts.time,

                params: opts.params || {}

            }

        })

    }

}

App.install = (Vue, options) => {

    Vue.prototype.$app = {

        // ...

        logEvent: App.logEvent

    }

}

export default App

方案二

通过在入口文件 index.js 全局注册混入 beforeRouteEnter、beforeRouteLeave 对象：

import Vue from 'vue'

Vue.mixin({

    beforeRouteEnter (to, from, next) {

        next(vm => {

            vm.$app.logEvent({

                type: 'visit',

                name: to.name,

                time: new Date().valueOf(),

                params: {

                    from: {

                        name: from.name,

                        path: from.path,

                        query: from.query

                    },

                    to: {

                        name: to.name,

                        path: to.path,

                        query: to.query

                    }

                }

            })

        })

    },

    beforeRouteLeave (to, from, next) {

        this.$app.logEvent({

            type: 'visit',

            name: to.name,

            time: new Date().valueOf(),

            params: {

                from: {

                    name: from.name,

                    path: from.path,

                    query: from.query

                },

                to: {

                    name: to.name,

                    path: to.path,

                    query: to.query

                }

            }

        })

        next()

    }

})

关闭应用时 beforeRouteLeave 是否触发？

上述方案存在明显缺陷：

官方曰慎用全局混入对象！！！

对于页面同名钩子函数 beforeRouteEnter、beforeRouteLeave，如何 merge？如何 next？

含子路由的页面将调用 2 次 beforeRouteEnter、beforeRouteLeave，PV 无形翻倍...

我猜此刻有打全局混入 created、destroyed 并通过 this.$route 获知访问对象主意的人了，试试看？

令人不知所措的输出，打印次数与 路由表 长度一致嗷~

其中 this.$app.logEvent（vm.$app.logEvent） 等同方案一中 App.logEvent，不再赘述。

如何恰当选取全局 PV 统计方案？

SPA 应用：仅单入口，在入口文件全局定义 Router.beforeEach 方便可行。

MPA 应用：多入口，在每个入口文件定义 Router.beforeEach？可封装公用逻辑（伪装单入口），免去重复构造 entry 的成本。

SPA + MPA 混合应用：emmmmmm...采用 MPA 应用的统计方案。

SSR 应用：为追求更好的 SEO 而采用服务端渲染（SSR）。以 Jinja（Python 模板）为例，调用 TemplateView 则为渲染页面（不同于前后端分离项目，服务端无法获知接口调用与页面渲染的对应关系），统计其调用次数及 TemplateName 可知页面 PV。

至于 UV，统计 PV 时采集 userId 去重即可。若应用无用户管理体系，采集 IP、deviceId 也可粗略得知 UV（不精准）。

功能点击量统计

不同功能的点击量不同，同一功能不同区域的点击量也不同：

按钮点击量，影响因素包含按钮 外观、位置、入口 等等（在此不考虑刚需）。按钮外观属 UI 设计范畴，按钮位置可通过分析用户点击热力图调整，按钮入口可通过分析触发源分布调整。

举一实例：

运营同学会将一张图片裁切成 n 个区域，点击每一区域所推荐商品不同。统计区域点击坐标，据热力图调整商品排序以求 利益最大化。

前端如何实现功能点击量统计？

本人将功能点击分两类：

带业务接口请求

无业务接口请求

方案一

将埋点上报混入业务接口请求，无接口请求的点击采用自定义上报：

其中 param keys 指代需上报的业务请求参数 key list（并非全部参数均需随埋点上报）。

上述方案大大节约请求数，但存在明显缺陷：

将埋点上报混入业务接口，上报 crash 不仅丢失统计数据，还将影响主功能。

统计与业务 高耦合，两者尽量不混于同一服务。

方案二

将所有点击事件视为同一类，走统一上报接口：

logEvent (opts) {

    Request({

        url: '/log/event',

        method: 'POST',

        data: {

            type: opts.type,

            name: opts.name,

            time: opts.time,

            params: opts.params || {}

        }

    })

}

上述方案也存在明显缺陷：

请求量翻倍：但统计与业务服务拆分后，请求并非同一组服务器承担。

待上报的点击事件函数均需调用 logEvent：封装一枚附带埋点上报的 组件，以 Vue 为例。

<template>

    <div class="vc-trace" @click="triggerClick">

        <slot></slot>

    </div>

</template>

<script>

import Request from './utils/request'

export default {

    name: 'Trace',

    props: {

        type: {

            type: String,

            default: ''

        },

        name: {

            type: String,

            default: ''

        },

        from: {

            type: String,

            default: ''

        },

        params: {

            type: Object,

            default: () => ({})

        }

    },

    methods: {

        triggerClick () {

            Request({

                url: 'XXX/log/event',

                method: 'POST',

                data: {

                    type: this.type,

                    name: this.name,

                    from: this.from,

                    time: new Date().valueOf(),

                    params: this.params

                }

            })

        }

    }

}

</script>

方案本无优劣，适合才更重要，需综合考虑 产品设计、产品使用度、服务利用率 等等。例使用度较低应用可将统计与业务混于同一服务以节约成本，使用度较高应用可采取 本地缓存、批量上报 以降低服务压力，但批量上报是否加大统计 误差？

————————————————

版权声明：本文为CSDN博主「1024小神」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_44786530/article/details/90286376>

### 埋点方案的确定

业界的埋点方案主要分为以下三类：

* 代码埋点：在需要埋点的节点调用接口，携带数据上传。如百度统计等；
* 可视化埋点：使用可视化工具进行配置化的埋点，即所谓的「无痕埋点」，前端在页面加载时，可以读取配置数据，自动调用接口进行埋点。如开源的Mixpanel;
* 无埋点：前端自动采集全部事件并上报埋点数据。如国内的神策数据等；

在当时排期紧凑，人力紧缺的情况下，显然不允许我们去开发可视化埋点方案和无埋点方案，所以只能采取代码埋点方案。

### 代码埋点

代码埋点分为 命令式埋点 与 声明式埋点 。

#### 命令式埋点

命令式埋点，顾名思义，开发者需要手动在需要埋点的节点处进行埋点。如点击按钮或链接后的回调函数、页面ready时进行请求的发送。大家肯定都很熟悉这样的代码：

// 页面加载时发送埋点请求

$(document).ready(function(){

// ... 这里存在一些业务逻辑

sendRequest(params);

});

// 按钮点击时发送埋点请求

$('button').click(function(){

// ... 这里存在一些业务逻辑

sendRequest(params);

});

可以很容易发现，这样的做法很有可能会将埋点代码侵入业务代码，这使整体业务代码变得繁琐，容易出错，且后续代码会愈加膨胀，难以维护。所以，我们需要让埋点的代码与具体的业务逻辑解耦，即 声明式埋点 ，从而提高埋点的效率和代码的可维护性。

#### 声明式埋点

理论上，声明式埋点只需要关注两个问题：

* 需要埋点的DOM节点；
* 所需携带的数据

因此，可以很快想出一个声明式埋点的方法：

// key表示埋点的唯一标识；act表示埋点方式

<button data-stat="{key:'111', act: 'click'}">埋点</button>

那么可以去遍历DOM树，找到 [data-stat] 的节点，给这个button绑上click事件，把这些参数在回调函数中通过请求发出去。

在DOM节点(html)上声明埋点，与业务逻辑（通常在Javascript文件中）就解耦了。调用也很方便。

看起来很美，但这样就能解决问题了吗？显然是不够的。还需要解决以下问题：

* 遍历DOM树的时机问题，一个简单的例子，一个表格的行数据是通过异步加载，而表格行中的操作按钮需要埋点，那么在DOM ready的时候去遍历，显然是无法找到的
* 绑定埋点事件次数的问题，怎样保证埋点事件不会被重复绑定到元素上，一次操作发了N个埋点请求?
* 如何处理特有的埋点行为，如页面展现埋点，区域展现埋点?
* 如何在解绑时，销毁已绑定的事件?

**通用的解决方案**

回顾一下，我们需要解决的问题是：

* 通过声明式埋点来解耦业务代码
* 埋点方案需要兼容Vue应用和jquery应用（甚至所有应用）
* 需要支持页面展现埋点、区域展现埋点、点击埋点等多种埋点方式
* 极端情况下需要支持命令式埋点

我们最终提出了一个基于Vue指令(Directive)和混合(Mixin)的解决方案：

#### 基于Vue指令的声明式埋点

由于在埋点的需求中有部分项目使用了Vue作为基础框架，结合上面声明式埋点的例子，很容易就联想到 Vue自定义指令。

这样的特性可以很好的解决以上的一些问题。我们只需要像这样：

Vue.directive('stat', {

bind: function () {

// 准备工作

},

update: function (newValue, oldValue) {

// 值更新时的工作

// 也会以初始值为参数调用一次, 此时可以根据传值类型来进行相应埋点行为的请求处理

},

unbind: function () {

// 清理工作

}

})

在一个Vue应用中，不需要再去遍历DOM树，因为在Vue应用中基本所有DOM操作都是使用数据的变更结合Vue的内置指令实现，Vue可以感知到这些变更。在指令从元素上解绑时我们也可以去销毁已经绑定的事件。

还有一些项目基于 jquery + widget 的老方案实现,那么在这些项目中的DOM操作是jquery甚至原生DOM API来实现，Vue的自定义指令就无法工作。

<div id="container">

<button id="btn">click</button>

</div>

<script>

new Vue({

el: '#container',

directives: {stat}

})

$('#btn').click(function() {

$('#container').append('<button v-stat="{key: '3', act: 'click'}">click</button>')

})

</script>

在上面例子中，虽然Vue已经挂载到 container 容器上，引入了自定义指令stat， #btn 这个按钮点击时插入了一段带有指令v-stat的按钮，因为Vue无法感知这个DOM变更，所以该指令不能被解析。这样的方式就会失效。

之前在外卖运营平台方向有基于 jquery 的DOM劫持操作的实现，在所有DOM操作中加入埋点相关的逻辑；因为无法保证所有的DOM操作都使用 jquery ， 且不能保证所有埋点逻辑完全一致，所以也无法通用。

那么，怎样保证在任意库，包括原生API的DOM操作下都感知到DOM的变更并且通知Vue重新解析指令呢？这里就需要引入 MutationObserver。

#### 基于MutationObserver API的Mixin

MutationObserver是在DOM3标准中提出的标准API，提供让开发者感知到在某一个DOM节点变更的能力。可以监听以下场景：

* childList: 目标节点的子节点插入删除引起的变更
* attributes: 目标节点属性改变引起的变更
* characterData: 目标节点的文本节点改变引起的变更，如通过appendData()等
* subtree: 目标节点的子孙节点改变引起的变更
* attributeOldValue：当attribute监听被设定为true时，可以记录改变前的属性值
* characterDataOldValue：当characterData监听被设定为true时，可以记录改变前的属性值
* attributeFilter：可以设定需要监听的属性列表

但为了保证MutationObserver可以在所有浏览器上正常工作，我们仍然引入了这个API的polyfill,[详情可见这里](https://link.zhihu.com/?target=https%3A//github.com/megawac/MutationObserver.js)。

在此能力的前提下，我们就可以在任意的DOM操作下触发Vue进行重新解析指令。

我们将 MutationObserver 封装进一个 Vue mixin , 非Vue应用的业务代码只需要引入这个mixin，这样也可以很好地解耦。

详细的实现原理可以见以下伪代码：

let observer;

export default {

ready() {

// 开启监听

observer = new MutationObserver(mutations => {

this.$compile(this.$el);

});

observer.observe(this.$el, config);

},

destroyed() {

// 清理工作

observer.disconnect();

observer.takeRecords();

}

}

### 埋点行为的处理

埋点库另一部分主要的逻辑是处理埋点行为。

### 页面展现埋点(ready)

Ready事件的处理，在页面根元素绑定指令后，在指令第一次update钩子调用时即可认为该元素ready, 直接发起请求埋点即可；

### 点击统计埋点(click)

click事件的处理，在该节点上绑定click事件，在指令解绑时销毁该事件。

### 区域展现埋点(show)

区域展现埋点即：当区域为可见状态变更时进行埋点。

那么，我们同样需要监听节点的可见状态变更。

理论上，DOM可见状态的变更也在MutationObserver的监听范围内，最初的一种思路是：

* 先设定MutationObserver的配置，开启attributeFilter和attributeOldValue，监听style的改变
* 看oldValue的值是否包含display: none, 和新状态比对
* 如成立，发送埋点

**let** observer **=** **new** MutationObserver((mutations) **=>** {

**if** (mutations[0].oldValue.indexOf('display: none') **>** **-**1

**&&** mutations[0].target.style.display **!==** 'none') {

sendRequest();

}

})

**let** config **=** {

attributes: **true**,

attributeOldValue: **true**,

attributeFilter: ['style']

};

observer.observe(el, config);

但是这种思路很快被否决，因为很显然，可见状态还有可能是被节点类名class控制的。而具体节点上的类名是无法预期的，因此这种方案行不通。

最终我们使用了开源库 VisSense。VisSense提供了监听可见状态变更的能力，[具体请见这里](https://link.zhihu.com/?target=https%3A//github.com/vissense/vissense)，本文不进行详细描述。

VisSense 实际使用了消息订阅模式和setInterval来进行周期性的节点状态检查，感兴趣的同学可以看看它的源码。

**function** handleShow(el) {

**var** visMonitor **=** VisSense(el).monitor({

visible: **function**() {

sendRequest();

}

});

visMonitor.start();

}

### 眼球曝光埋点(collect)

眼球曝光埋点标识用户是否「看到」了某个区域，那么用前端的方式来解释就是：

* 该区域是可见状态
* 用户页面的滚动条位置与该区域的实际位置相匹配

主要的实现思路就是监听scroll事件，与当前节点的scrollTop进行对比。 由于本次需求未涉及眼球曝光，本部分不再赘述。

### 极端情况下的命令式埋点支持

上面的声明式埋点方案已经可以解决大多数问题。

但是，不是100%的情况都适用声明式埋点，主要发生在 DOM操作不受开发者完全控制 的情况。

举个例子，在使用百度地图API时，在地图上打一些POI点（markPoint）, 或者一些蒙层（如Polygon), 再在点击这些覆盖物时埋点，由于这些DOM操作是百度地图API完成的，无法预期插入了哪些DOM，自然就不能在这些DOM上插入指令。所以只能在调用API时进行命令式埋点。需要我们也提供命令式埋点支持。

命令式埋点的大部分逻辑实际已经包含在指令中，于是我们在指令中提供了这样的接口方式：

**export** **default** {

bind() {...},

update() {...},

unbind() {...},

sendStat(val) {

*// 命令式埋点接口*

}

}

引入此模块后，即可以当作Vue指令使用，也可以当做一个API来使用。

### 其他的一些处理

此外，埋点方案还提供了可配置能力，可以设定测试环境还是生产环境的规则（根据URL匹配），设定埋点请求的URL地址，是否开启debug模式等。

在测试环境下，埋点请求的时机只会在浏览器中进行console.log并打印出触发埋点的节点，不会实际发送请求，可以支持测试环境下的正常开发，又可以避免埋点出现脏数据。

#### 使用方式

* 在Vue项目中，直接使用自定义指令即可
* 在非Vue项目中，需要引入Mixin。如下

new Vue({

el: '#app', // 根节点

directives: {stat},

mixins: [observerMixin] // 非Vue项目需要引入

})

然后在页面相应节点进行声明式埋点即可:

<div id="app" v-stat="{'act':'ready',' key':'samplepg'}"></div> // 页面展现埋点

<button v-stat="{'act':'click', 'key':'samplebtn'}"></button> // 点击统计埋点

<div id="container" v-stat="{'act':'show',' key':'samplepn'}"></div> // 区域展现埋点

这样的埋点方式十分简便快捷。

## 总结

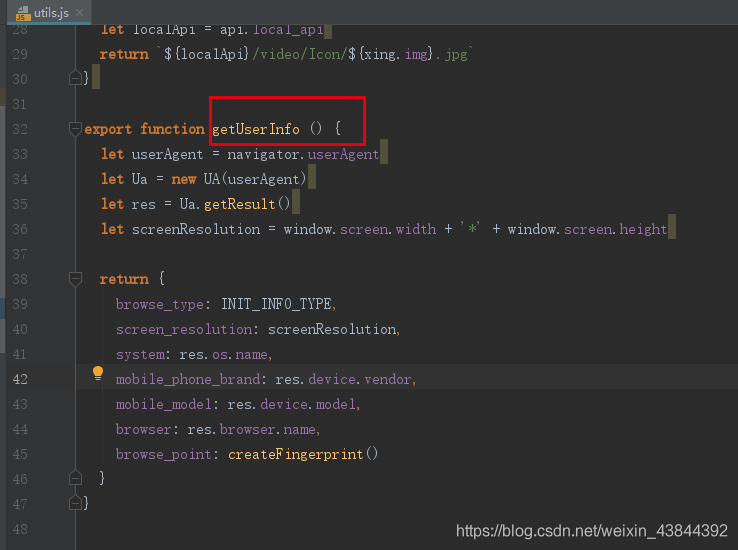
前端的数据采集和上报是构建数据平台的重要环节，而前端如何进行埋点也是值得深究的。为了快速满足业务的大量埋点需求，我们使用了本文的埋点方案，而且已经大量在商业平台部开发中使用，无论从FE同学的开发反馈、实际产出数据的结果来看都达到我们的预期，后续会继续在一些业务上进行持续迭代和优化。

# 前端埋点的简单实现

要记录用户的浏览轨迹

知道用户都浏览了哪些页面，在每个页面都停留了多长时间，还有 比如用户所用的手机型号 等个人信息

个人信息获取的接口如下：



在用户刚进入时获取用户信息



3.安装 js-cookie 插件

npm install js-cookie --save

 4.在main.js中引入

5.利用路由导航实现对不同页面的浏览的时间的记录

// 用于记录每次路由跳转时候的事件，只记录两个

let time = []

// console.log(Cookie.get('webPoint'))

router.beforeEach((to, from, next) => {

let startTime = new Date().getTime()

if (time.length < 2) {

// 模拟队列，保证队列中每次只有两个元素

time.push(startTime)

}

if (time.length === 2) {

let stopTime = time[1] - time[0]

let baseRes = {

browse\_time: stopTime,

start\_time: time[0],

end\_time: time[1],

browse\_point: Cookie.get('webPoint')

}

if (from.name === 'videoPlay') {

let path = from.path.split('/')[2]

let pathJson = JSON.parse(path)

let res = {

browse\_type: 1,

movie\_id: pathJson.show\_name,

...baseRes

}

console.log('res1', res)

axios.post('http://192.168.0.188:8006/movie-info/', {

...res

}).then(res => { console.log(res) })

} else if (from.name === 'goods') {

let res = {

browse\_type: 2,

goods\_id: parseInt(from.params.id),

...baseRes

}

console.log('res2', res)

axios.post('http://192.168.0.188:8006/goods-info/', {

...res

}).then(res => { console.log(res) })

} else if (from.name === 'infoDetail') {

let res = {

browse\_type: 3,

msg\_id: parseInt(from.params.id),

...baseRes

}

console.log('res3', res)

axios.post('http://192.168.0.188:8006/msg-info/', {

...res

}).then(res => { console.log(res) })

} else {

let res = {

browse\_type: 4,

url: from.path,

...baseRes

}

console.log('res4', res)

axios.post('http://192.168.0.188:8006/additional-info/', {

...res

}).then(res => { console.log(res) })

}

time.shift()

}

next()

})

大概就是这个样子

具体的因为涉及到隐私不能够透露哦！

————————————————

版权声明：本文为CSDN博主「weixin\_43844392」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_43844392/article/details/89478756>

# JS埋点技术分析

 发表于 2017-12-24 |  分类于 [技术](http://unclechen.github.io/categories/%E6%8A%80%E6%9C%AF/)|

## 一、背景

[上一篇博客](http://unclechen.github.io/2017/12/18/Android%E5%9F%8B%E7%82%B9SDK%E6%8A%80%E6%9C%AF%E5%88%86%E6%9E%90/)分析了Android上的埋点SDK技术原理，这次我看看Web页面上的埋点。Web页面上的埋点主要通过JS完成，在JS里面同样有代码埋点、全埋点、可视化埋点三种方案，如果对这几种方案的概念不了解可以看下上一篇博客。由于[mixpanel-js](https://github.com/mixpanel/mixpanel-js)和[Sensors Analytics JavaScript SDK](https://github.com/sensorsdata/sa-sdk-javascript)都开源了自己的SDK，就以它们为例进行分析。

## 二、代码埋点

以Mixpanel为例（源码位于/src/mixpanel-core.js），看一下里面的实现。

### 2.1 基本用法

埋点之前，需要在head部分嵌入SDK，并调用SDK的初始化接口。以Mixpanel为例，官方介入文档提供的加载、初始化SDK代码如下：

|  |  |
| --- | --- |
| 1  2  3  4 | <!-- start Mixpanel --><script type="text/javascript">(function(e,a){if(!a.\_\_SV){var b=window;try{var c,l,i,j=b.location,g=j.hash;c=function(a,b){return(l=a.match(RegExp(b+"=([^&]\*)")))?l[1]:null};g&&c(g,"state")&&(i=JSON.parse(decodeURIComponent(c(g,"state"))),"mpeditor"===i.action&&(b.sessionStorage.setItem("\_mpcehash",g),history.replaceState(i.desiredHash||"",e.title,j.pathname+j.search)))}catch(m){}var k,h;window.mixpanel=a;a.\_i=[];a.init=function(b,c,f){function e(b,a){var c=a.split(".");2==c.length&&(b=b[c[0]],a=c[1]);b[a]=function(){b.push([a].concat(Array.prototype.slice.call(arguments,  0)))}}var d=a;"undefined"!==typeof f?d=a[f]=[]:f="mixpanel";d.people=d.people||[];d.toString=function(b){var a="mixpanel";"mixpanel"!==f&&(a+="."+f);b||(a+=" (stub)");return a};d.people.toString=function(){return d.toString(1)+".people (stub)"};k="disable time\_event track track\_pageview track\_links track\_forms register register\_once alias unregister identify name\_tag set\_config reset people.set people.set\_once people.unset people.increment people.append people.union people.track\_charge people.clear\_charges people.delete\_user".split(" ");  for(h=0;h<k.length;h++)e(d,k[h]);a.\_i.push([b,c,f])};a.\_\_SV=1.2;b=e.createElement("script");b.type="text/javascript";b.async=!0;b.src="undefined"!==typeof MIXPANEL\_CUSTOM\_LIB\_URL?MIXPANEL\_CUSTOM\_LIB\_URL:"file:"===e.location.protocol&&"//cdn.mxpnl.com/libs/mixpanel-2-latest.min.js".match(/^\/\//)?"https://cdn.mxpnl.com/libs/mixpanel-2-latest.min.js":"//cdn.mxpnl.com/libs/mixpanel-2-latest.min.js";c=e.getElementsByTagName("script")[0];c.parentNode.insertBefore(b,c)}})(document,window.mixpanel||[]);  mixpanel.init("YOUR TOKEN");</script><!-- end Mixpanel --> |

这是一段立即执行的js代码，作用通常是去异步加载真正的JS SDK，然后调用SDK的初始化接口init方法，完成初始化的操作。

初始化的核心代码为

|  |  |
| --- | --- |
| 1 | mixpanel.init('YOUR TOKEN', { your: 'config' }, 'library\_name') |

也可以简写为

|  |  |
| --- | --- |
| 1 | mixpanel.init("YOUR TOKEN") |

看下init方里面几个参数的含义：

* 第一个参数是你在后台注册的app token
* 第二个参数是SDK的配置，传入了一堆key-value，如果不传，SDK内部也有个默认配置，长下面这样：

|  |  |
| --- | --- |
| 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 | var DEFAULT\_CONFIG = {  'api\_host': HTTP\_PROTOCOL + 'api.mixpanel.com',  'app\_host': HTTP\_PROTOCOL + 'mixpanel.com',  'autotrack': true, // 是否打开全埋点监测  'cdn': HTTP\_PROTOCOL + 'cdn.mxpnl.com',  'cross\_subdomain\_cookie': true,  'persistence': 'cookie',  'persistence\_name': '',  'cookie\_name': '',  'loaded': function() {},  'store\_google': true,  'save\_referrer': true,  'test': false,  'verbose': false,  'img': false,  'track\_pageview': true,  'debug': false,  'track\_links\_timeout': 300,  'cookie\_expiration': 365,  'upgrade': false,  'disable\_persistence': false,  'disable\_cookie': false,  'secure\_cookie': false,  'ip': true,  'property\_blacklist': []  }; |

* 第三个参数是SDK全局变量名

Mixpanel接入文档：<https://mixpanel.com/help/reference/javascript>

### 2.2 上报的基本实现

代码埋点的方式通常都会被封装成类似track(eventName, properties)的接口，例如在Mixpanel中，可以用mixpanel.track("Played song", {"genre": "hip-hop"});来上报事件。

这里是整个SDK中最重要的地方，使用频率也是最高的。代码位于/src/mixpanel-core.js里面，先撇开复杂的逻辑和条件控制，看一下track的基本实现，我稍微加了点注释：

|  |  |
| --- | --- |
| 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  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | // track方法实现  MixpanelLib.prototype.track = function(event\_name, properties, callback) {  // 各种边界判断  ...  // 获取一些公共参数，和用户传入的properties一起encode  var truncated\_data = \_.truncate(data, 255);  var json\_data = \_.JSONEncode(truncated\_data);  var encoded\_data = \_.base64Encode(json\_data);  console.log('MIXPANEL REQUEST:');  console.log(truncated\_data);  // 调用\_send\_request发送请求  this.\_send\_request(  this.get\_config('api\_host') + '/track/',  { 'data': encoded\_data },  this.\_prepare\_callback(callback, truncated\_data)  );  return truncated\_data;  };  // 发送请求的实现，主要用的是XMLHttpRequest，如果浏览器不支持XMLHttpRequest，那么用动态添加img/script标签的方式  MixpanelLib.prototype.\_send\_request = function(url, data, callback) {  // 一些特殊情况的处理  ...  if ('img' in data) {  var img = document.createElement('img');  img.src = url;  document.body.appendChild(img);  } else if (USE\_XHR) {  try {  var req = new XMLHttpRequest();  req.open('GET', url, true);  // send the mp\_optout cookie  // withCredentials cannot be modified until after calling .open on Android and Mobile Safari  req.withCredentials = true;  req.onreadystatechange = function () {  if (req.readyState === 4) { // XMLHttpRequest.DONE == 4, except in safari 4  if (req.status === 200) {  if (callback) {  if (verbose\_mode) {  callback(\_.JSONDecode(req.responseText));  } else {  callback(Number(req.responseText));  }  }  } else {  var error = 'Bad HTTP status: ' + req.status + ' ' + req.statusText;  console.error(error);  if (callback) {  if (verbose\_mode) {  callback({status: 0, error: error});  } else {  callback(0);  }  }  }  }  };  req.send(null); // 发送异步请求  } catch (e) {  console.error(e);  }  } else {  var script = document.createElement('script');  script.type = 'text/javascript';  script.async = true;  script.defer = true;  script.src = url;  var s = document.getElementsByTagName('script')[0];  s.parentNode.insertBefore(script, s);  }  }; |

上面就是事件上报代码的核心实现。但是由于Web应用自身的一些特性，比如在追踪页面跳转行为（链接的点击、表单的提交等）时，为了防止数据发送不及时导致的数据丢失，SDK中提供一些诸如track\_links和track\_forms特殊方法，这些方法内部用的其实是setTimeout或者等待服务器返回结果之后再让页面跳转。

## 三、全埋点

Mixpanel和神策都提供了名为**“AutoTrack”**的方案，只需要在初始化SDK的时候，传入一个参数即可打开这个功能。JS SDK可以自动监测网页中所有的点击、表单submit等事件，这和AndroidSDK里面监听所有按钮的点击有些类似。

### 3.1 自动监测的元素、事件类型

* **神策JS：**设置AutoTrack之后，SDK就会自动追踪页面上的按钮(a、button、input) 这种html标签类型的点击情况，一旦页面某一个按钮发生了点击行为，SDK就会去采集此按钮的一些信息，例如: 这个按钮的标签类型，这个按钮的文本内容，这个按钮的name，这个按钮的id、class名，还有一些按钮特有的属性如href等。
* **MixpanelJS：**设置AutoTrack之后，SDK会监测页面上的所有form表单、input标签、select和textarea标签产生的submit、change、click事件，并采集这些标签上的属性一起上报。

### 3.2 全埋点监测的实现

以Mixpanel为例，在/src/autotrack.js代码中，把几个关键的方法扣出来看一下（不要问我为什么以Mixpanel为例，因为代码少一些。。。）。

当SDK初始化的时候，会执行autotrack里面的\_addDomEventHandlers方法，给整个document的submit、change、click事件设置监听器。当监听到这几类事件时，会执行\_trackEvent方法。

直接看代码，我给代码里面加了一点注释，来说明自动监测上报的过程。

|  |  |
| --- | --- |
| 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  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126 | // SDK初始化时，通过register\_event设置需要监听了submit、change、click这3类事件  // Mixpanel的js sdk代码里面自己封装了一个underscore模块，里面有一些工具方法  \_addDomEventHandlers: function(instance) {  var handler = \_.bind(function(e) {  e = e || window.event;  this.\_trackEvent(e, instance);  }, this);  \_.register\_event(document, 'submit', handler, false, true);  \_.register\_event(document, 'change', handler, false, true);  \_.register\_event(document, 'click', handler, false, true);  },  // register\_event的实现，优先采用addEventListener的方式，如果浏览器不支持会尝试使用onXXX的方式  var register\_event = function(element, type, handler, oldSchool, useCapture) {  if (!element) {  console.error('No valid element provided to register\_event');  return;  }  if (element.addEventListener && !oldSchool) {  element.addEventListener(type, handler, !!useCapture);  } else {  var ontype = 'on' + type;  var old\_handler = element[ontype]; // can be undefined  element[ontype] = makeHandler(element, handler, old\_handler);  }  };  // 监听到事件发生后，调用\_trackEvent方法来上报  \_trackEvent: function(e, instance) {  // 首先找到这个事件的target  var target = this.\_getEventTarget(e);  if (target.nodeType === TEXT\_NODE) {  target = target.parentNode;  }  // 然后判断是不是autotrack要监测的事件，如果不是的话，啥也不干直接返回。  if (this.\_shouldTrackDomEvent(target, e)) {  // 如果满足监测条件，那么从当前标签开始，向上追溯到body标签，并记录这条路径上所有的元素到一个数组中  var targetElementList = [target];  var curEl = target;  while (curEl.parentNode && !this.\_isTag(curEl, 'body')) {  targetElementList.push(curEl.parentNode);  curEl = curEl.parentNode;  }  // 按照刚才记录的路径开始遍历（相当于自底向上）  var elementsJson = [];  var href, elementText, form, explicitNoTrack = false;  \_.each(targetElementList, function(el, idx) {  // if the element or a parent element is an anchor tag  // include the href as a property  // 读取到a标签或者form标签时，记录它们的属性。  if (el.tagName.toLowerCase() === 'a') {  href = el.getAttribute('href');  } else if (el.tagName.toLowerCase() === 'form') {  form = el;  }  // crawl up to max of 5 nodes to populate text content  // 读取节点的文本内容，最多往上读个5层  if (!elementText && idx < 5 && el.textContent) {  var textContent = \_.trim(el.textContent);  if (textContent) {  elementText = textContent.replace(/[\r\n]/g, ' ').replace(/[ ]+/g, ' ').substring(0, 255);  }  }  // allow users to programatically prevent tracking of elements by adding class 'mp-no-track'  // 如果不希望某个节点被监测，开发者可以设置一个名为`mp-no-track`的css class  var classes = this.\_getClassName(el).split(' ');  if (\_.includes(classes, 'mp-no-track')) {  explicitNoTrack = true;  }  // 读取每个标签的属性，最后这条路径上所有的标签都会被记录下来保存在elementsJson数组中  elementsJson.push(this.\_getPropertiesFromElement(el));  }, this);  // 如果是一个开发者设置了不需要监测的标签，那么直接返回，不上报了  if (explicitNoTrack) {  return false;  }  // 处理采集到的属性，这里面有几个getXXXProperties(element/elements)方法（\_getPropertiesFromElement、\_getDefaultProperties、\_getCustomProperties、\_getFormFieldProperties），就是在读取各种属性  var props = \_.extend(  this.\_getDefaultProperties(e.type), // 事件的基本属性，包含事件名称、window.location.host、window.location.pathname  {  '$elements': elementsJson, // target标签到body标签这条路径上的所有标签及其属性  '$el\_attr\_\_href': href, // 采集到的href链接  '$el\_text': elementText // target标签的文本内容  },  this.\_getCustomProperties(targetElementList) // 读取自定义属性，这里应该是指用户在后台管理界面配置的属性  );  if (form && (e.type === 'submit' || e.type === 'click')) {  \_.extend(props, this.\_getFormFieldProperties(form)); // 读取表单的一些属性  }  // 调用了代码埋点中介绍的track方法上报一个名为`$web\_event`的事件，并带上采集的到的属性  instance.track('$web\_event', props);  return true;  }  },  // \_trackEvent之前，需要判断标签上的发生的事件是不是应该被autotrack监测上报  \_shouldTrackDomEvent: function(element, event) {  // html根节点下面的事件不需要监测  if (!element || this.\_isTag(element, 'html') || element.nodeType !== ELEMENT\_NODE) {  return false;  }  var tag = element.tagName.toLowerCase();  // 查看标签的名字  // 如果是html则不监听  // 如果是form标签下的submit事件，或者是input->button、input->submit标签的change、click事件，或者是select、textarea标签下的change、click事件，可以监听  // 如果是其他标签，监听click事件  switch (tag) {  case 'html':  return false;  case 'form':  return event.type === 'submit';  case 'input':  if (['button', 'submit'].indexOf(element.getAttribute('type')) === -1) {  return event.type === 'change';  } else {  return event.type === 'click';  }  case 'select':  case 'textarea':  return event.type === 'change';  default:  return event.type === 'click';  }  }, |

### 3.3 全埋点小结

可以看到全埋点还是有点暴力的，会采集的数据量也挺大，并且采集到的属性也比较多，可以看到在MixpanelSDK中，如果页面结构比较深，那么数据报过去分析起来可能还是需要花点时间的，同时也会产生大量可能不会使用的数据，对资源也是一种浪费。在神策SDK的接入文档中也提到，建议那些按钮不是很多的，相对简单的页面可以采用这个方法。一般情况下，如果网页上的按钮比较多的话，因为每次按钮的点击都会发数据，数据量很大。

## 四、可视化埋点

Mixpanel和神策等平台，都提供了JS可视化埋点功能，与全埋点相比，这种方式可以指定自己想要监测的元素和属性（所有可以点击的元素），既可以做到动态配置，又不会像全埋点那样产生大量的数据（但也有例外，比如Mixpanel的可视化埋点仍然上报了全量点击数据，只是在后台根据可视化配置过滤出实际的数据）。

可视化埋点首先需要进入埋点模式，以Mixpanel为例，可视化埋点的入口在后台管理界面，需要在后台输入需要埋点的页面url，然后再进入我的Web页面，此时就会加载可视化标记的编辑器（代码见autotrack.js中的\_maybeLoadEditor方法，需要注意的是这个页面必须已经嵌入了JS SDK）。

这里一定要从平台登录才可以进入可视化编辑状态，这实际上是对安全性的一个保证，试想如果只要嵌入SDK就可以做可视化埋点，那岂不是我的Web应用随便就可以被别人埋点，对我的数据产生干扰了。在Mixpanel JS SDK内部，通常会**判断当前页面的sessionStorge/localStorage中是否有一个开启可视化编辑器标志字段（例如Mixpanel是\_mpcehash字段）**，读取这个字段，解析到其中的打开可视化编辑器的开关开启之后，就会加载可视化编辑器。由此可见其实从SDK后台管理界面跳转到可视化标记页面时，就是向SessionStorage中写入了相应的标志。

**可视化埋点的两个关键点是：**

* **标记元素，保存配置**：这一步要保存好需要追踪的元素的element\_path，以及需要追踪的元素。
* **下发配置，查找元素，监听点击，上报行为**：这一步要通过element\_path找到元素，给它添加一个点击监听器，当点击事件发生时SDK上报事件。

这里最重要的就是：**元素的标记和查找**，不同的SDK就是实现标记和查找的时候稍微有一些差异。

### 4.1 标记元素，保存配置

MixpanelJS加载可视化编辑器时，需要从//mixpanel.com/js-bundle/reports/collect-everything/editor.js?\_ts={$timestamp}去加载一个js文件，**这个js差不多可以看成一个独立的标记SDK，**最后这个请求会被重定向到一个cdn地址（https://cdn4.mxpnl.com/static/asset-cache/3fc4abfdcebcb5121f1ebf143415b232/compiled/reports/collect-everything/editor.min.js），随便打开这个js看下就有两万多行，因此单独做成了一个按需加载的模块。

由于Mixpanel就没有提供标记SDK的源码，不过从体验和**抓包分析后台下发的配置**，我仍然可以推测出技术实现的细节。

**从体验的角度来讲，**当进入可视化编辑状态时，在开发者web页面上，用户的鼠标经过**可以被点击**的元素（例如a、button标签等）时，这个元素会被一种颜色高亮提示，此时点击一下这个元素，就会弹出一个浮窗，用户填写信息，设置一个事件和一些属性，保存之后就算完成对这个元素的标记操作了，当标记过的元素的配置保存好了以后，这个元素会用另外一种特殊的颜色高亮标识起来。

**从技术的角度来讲，**我看下神策JS SDK中的vtrack.sdk.js这个文件，当神策SDK进入可视化标记模式的时候，会去加载vendor.js和vendor.css，这两个文件可以看作一个标记SDK。那么vendor.js代码里是**如何标记需要追踪的元素的**呢？

在vendor.js中，有一个EventDefine模块，这个模块负责把一个标签处理成我要保存的selector。

**EventDefine**有三个方法：

* getSelfAttr：获取一个标签内的文本内容，举例来说，一个<p>This is another paragraph.</p>得到的内容是This is another paragraph.。
* toSelector：把一个标签的tagName、id、classNames解析出来，拼成一个串。举例来说，一个<div id="test" class="uncle chen"></div>标签，它的selector是div#id.uncle.chen，这个selector是可以直接给jQuery用来查找元素的。
* toAllSelector：选择一个需要追踪的标签，并给这个标签定义点击时上报的事件（EventDefine），最后将这个事件转成一个selector保存下来，selector就是用于给jQuery来查找元素的选择器，这里需要注意，如果一个元素是在iFrame里面的，那么SDK保存的选择器路径是相对iFrame内部的，而不是最外层的document。

前两个方法都是给toAllSelector方法调用的，toAllSelector方法是神策的标记SDK的重点，这个方法的实现如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | toAllSelector: function($target, outDocuemnt) {  outDocuemnt = outDocuemnt ? $(outDocuemnt) : $(document);  var $parent, newSelSize, newSelector, parts, selSize, selector, targetSel;  selector = this.toSelector($target, outDocuemnt);  $parent = $target.parent();  selSize = outDocuemnt.find(selector).length;  while ($parent.prop('tagName') !== 'BODY' && selSize !== 1) {  newSelector = '' + (this.toSelector($parent)) + ' ' + selector; // 如果向上回溯的话，selector会用空格分开保存  newSelSize = outDocuemnt.find(newSelector).length;  if (newSelSize < selSize) {  selector = newSelector;  selSize = newSelSize;  }  $parent = $parent.parent();  }  var nthEle = selector;  var selfAttr = this.getSelfAttr($target);  return {  nthEle: nthEle,  selfAttr: selfAttr  };  } |

当选中一个标签时，SDK会提取出这个标签的selector，然后用jQuery选择器查找这个selector指向的元素，如果这个selector指向的元素有多个（selSize !== 1，也就是说这个元素有着多个兄弟标签），那么还需要进一步去提取其父标签的selector，直到找出可以**唯一**标识这个元素的selector为止，最后将需要追踪的这个元素以{nthEle: nthEle, selfAttr: selfAttr}`，nthEle是selector，selfAttr是文本内容。

**简单总结一下元素的标记，在Web页面中，一个元素的唯一css选择器生成算法，应该记录了从body到这个元素的完整路径，并记录每一个节点是其父亲节点的第几个孩子节点，即这个元素在整个Dom Tree中的深度和下标。此外，为了在一定程度上抵抗Dom Tree的变化，下标应该记录的是这个元素在父节点中相同类型元素的index（nth-of-type），而不是其父节点下面所有孩子节点的index（nth-child）。**

### 4.2 查找元素，监听上报

标记元素，保存配置之后，SDK如何根据配置来监测配置好的元素，并进行上报呢？前面我说到Mixpanel在可视化埋点的上报实现里，仍然保持了全量点击事件上报，并在每个上报中把元素在Dom Tree的完整路径一起上报到了后台，由后台去过滤出可视化事件。

所以这里我看下神策js的代码，在可视化模块vtrack.sdk.js中，正常模式下，会去解析后台下发的配置，找到标记过的元素，绑定事件。

**1.下发配置**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // 进入普通模式时，会从后台的一个接口去拉去标记过的元素（这里也叫“部署”过的元素）的关键信息，然后进行解析  enterNormalMode: function() {  sd.vtrack\_mode = 'normalMode';  var me = this;  this.getDeployFile().then(function() {  me.parseDeployFile(); // 解析配置  });  }, |

由于神策的后台代码是走私有化部署的，我没有办法体验，这里看一份诸葛IO平台可视化配置：

|  |  |
| --- | --- |
| 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  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | {  "code": 10001,  "msg": "Request success",  "visual\_events": [  {  "event\_name": "button3333",  "url": "http://localhost:63343/sa-sdk-javascript/zhuge.html?\_ijt=kc0vtnal8qahrd2tq5obomukh8",  "element": [  "#b"  ],  "attr": [  {  "name": "ppp",  "selector": "body>p:eq(0)"  }  ],  "app\_id": 56070,  "platform": 3,  "create\_date\_time": "2018-01-23 17:45:53",  "hidden": null,  "stop": null,  "alias\_name": null,  "edit\_url": "http://localhost:63343/sa-sdk-javascript/zhuge.html?\_ijt=kc0vtnal8qahrd2tq5obomukh8"  },  {  "event\_name": "button点击2222",  "url": "",  "element": [  "#b"  ],  "attr": [],  "app\_id": 56070,  "platform": 3,  "create\_date\_time": "2018-01-23 17:39:16",  "hidden": null,  "stop": null,  "alias\_name": null,  "edit\_url": "http://localhost:63343/sa-sdk-javascript/zhuge.html?\_ijt=k9d5hjmkqt1ethu1ao51cf4t9e"  },  {  "event\_name": "20180123",  "url": "",  "element": [  "#b"  ],  "attr": [],  "app\_id": 56070,  "platform": 3,  "create\_date\_time": "2018-01-23 17:08:04",  "hidden": null,  "stop": null,  "alias\_name": null,  "edit\_url": "http://localhost:63343/sa-sdk-javascript/zhuge.html?\_ijt=ir7e9n0scm88b0k2uvotnc4ntd"  },  {  "event\_name": "hello链接点击",  "url": "http://localhost:63342/sa-sdk-javascript/zhuge.html?\_ijt=hg3r25mmicg2icf0jtndtaskh7#",  "element": [  "body>a:eq(0)"  ],  "attr": [  {  "name": "自定义属性",  "selector": "body>p:eq(0)"  }  ],  "app\_id": 56070,  "platform": 3,  "create\_date\_time": "2017-12-22 16:25:29",  "hidden": false,  "stop": false,  "alias\_name": null,  "edit\_url": "http://localhost:63342/sa-sdk-javascript/zhuge.html?\_ijt=hg3r25mmicg2icf0jtndtaskh7#"  }  ]  } |

注意，上面这份配置有两个字段，一个是url，另一个是editUrl，editUrl表示标记元素的时候，是在哪一个页面里操作的。url表示应该去哪个url下面查找标记的元素。因为有些情况下，虽然我们是在某一个页面标记的元素，但是我们有很多其他页面和这个页面长得类似，比如商品详情类的页面，所以我们其实希望在所有的商品详情页都可以上报某些事件。所以，如果url为一个具体的值，例如”[http://localhost:63342/sa-sdk-javascript/zhuge.html?\_ijt=hg3r25mmicg2icf0jtndtaskh7#](http://localhost:63342/sa-sdk-javascript/zhuge.html?_ijt=hg3r25mmicg2icf0jtndtaskh7)“，说明只应该在这个url对应的页面中查找元素，上报行为即可；如果url=””，说明我们应该在整个Web应用中的所有页面都去依据路径查找元素，上报事件。

**2.解析配置，监测元素**

还是直接看神策代码实现，里面加了点注释：

|  |  |
| --- | --- |
| 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  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | // 解析配置，查看当前页面中是否有元素需要被追踪，把需要追踪的元素的配置保存到requiredData变量中  parseDeployFile: function() {  this.requireData = this.checkUrl(this.deployData);  this.listenEvents();  },  // 找到元素，绑定点击事件的处理，当元素被点击时，上报事件  listenEvents: function() {  var data = this.requireData;  var me = this;  for (var i = 0; i < data.length; i++) {  this.getEle(data[i]).on('click', function(ev) {  return function() {  me.doVTrackAction(ev);  }  }(data[i]));  }  },  doVTrackAction: function(data) {  sd.track(  data.eventName,  {  $from\_vtrack: String(data.trigger\_id)  },  {  $lib\_method: 'vtrack',  $lib\_detail: String(data.trigger\_id)  }  );  },  // 通过jQuery的选择器来找到元素，我在前一节的标记操作中知道，标记SDK会把一个定义好的事件eventDefine转化成一个{nthEle: nthEle, selfAttr: selfAttr}结构保存起来，这里去寻找元素的时候和标记那里的逻辑其实是一个逆操作。  // 这里要注意，和标记时一样元素，碰到iframe时要特殊处理一下。  getEle: function(data) {  var ele;  if ($(data.nthEle[0]) && $(data.nthEle[0]).prop('tagName') === 'IFRAME') {  try {  ele = $(data.nthEle[0]).contents().find(data.nthEle.slice(1).join(' '));  } catch (e) {  }  ;  } else {  ele = $(data.nthEle.join(' '));  }  if (data.selfAttr && data.selfAttr.text !== void 0) {  ele = ele.filter(':contains(' + data.selfAttr.text + ')');  }  return ele;  }, |

神策查找元素的时候用到了jQuery，而在Mixpanel中没有用jQuery，而是用的Document.querySelectorAll这个API。毕竟有很多移动页面为了优化加载速度，不会用jQuery这么重的库。

此外，当追踪一些特殊的标签时，可以考虑用[XPath](http://www.w3school.com.cn/xpath/)去定位，今日头条的广告监测插件其实就用到了XPath。

**3.给事件上报添加属性**

单独把添加属性拿出来讲，是因为它的原理是类似的。前面我们只提到标记一个元素，当它被点击的时候上报事件，但是这样没有在上报事件的同时带上自定义的一些属性。

其实只要是Web页面上出现了的元素，我们都可以把它记录下来，然后在事件发生的时候，查找到这些元素，并把它们的内容作为事件的属性上报上来。我们可以在标记了一个元素的时候，再去标记其他的一些元素（例如一些文本标签），并设置其他这些元素各自所对应的key，当事件发生时，我们可以找到其他这些元素，并获取到其中显示的文本内容，作为各自参数的值，上报到后台。

### 4.3 可视化埋点小结

可以看出，在JS上实现可视化埋点不是一件太麻烦的事情，我认为最关键的两件事就是标记元素和查找元素。不过它缺点是只会读取页面上的标签元素的展示出来的属性，也不会像代码埋点的方案那样去理解业务场景，获取上下文（通常在内存里）的一些属性；另外，当页面的结构发生变化的时候，可能要重新进行一次标记操作。有些平台是通过对事件监测的告警来提醒用户的，当事件数量同比大幅减少的时候，大概率是因为某次改版导致页面Dom Tree产生了变化，通过配置下发里面的元素路径找不到之前标记的元素了，这时就应该提醒用户重新标记。

## 五、总结

本文从代码埋点、全埋点、可视化埋点三个角度，以Mixpanel、神策数据的JS SDK的源代码，分析了Web页面埋点的实现方案的实现。在流量红利逐渐消失的现在，数据的采集、分析和精细化的运营显得更加重要，下面简单列一个表格对以上三种方式的埋点方案进行对比，还是那句话，三种埋点方式相辅相成，结合业务需求搭配使用，适合自己的才是最好的。

|  |  |  |  |
| --- | --- | --- | --- |
| **埋点方案** | **优点** | **缺点** | **适用场景** |
| 代码埋点 | 可以按照业务上报详细、定制化的数据 | 需要开发人员参与，更新维护成本高，无法获得历史数据 | 对上下文理解要求较高的业务数据 |
| 全埋点 | 对发人员依赖低，仅需嵌入一次SDK，可以全量上报通用数据，可以拿到历史数据 | 数量量太大，占用更多资源，且无法收集业务上下文数据，给后续数据筛选和分析带来一定的难度 | 上下文相对独立的、通用的数据 |
| 可视化埋点 | 对开发人员依赖低，可以按照业务需求上报数据，对上下文数据有一定收集能力 | 标记事件有一定的操作难度，事件需要被更新时无法获得历史数据，界面变化时标记的元素可能失效 | 业务上下文数据相对简单，操作交互比较固定的界面 |