数据埋点方案、监控方案

代码埋点

代码埋点是最灵活，同时也是最耗时的一种方式。

一般大厂内部会封装自己的一套埋点上报的npm包, 提供给各业务线使用。

一般我们需要上报什么信息呢?

1. 埋点的标识信息, 比如eventId, eventType
2. 业务自定义的信息, 比如教育行业, 点击一个按钮, 我们要上报用户点击的是哪个年级
3. 通用的设备信息, 比如用户的userId, useragent, deviceId, timestamp, locationUrl等等

一般怎么上报?

1. 实时上报, 业务方调用发送埋点的api后, 立即发出上报请求
2. 延时上报, sdk内部收集业务方要上报的信息, 在浏览器空闲时间或者页面卸载前统一上报，上报失败会做补偿措施。

实现

代码

无埋点

概念

无埋点并不是真正的字面意思，其真实含义其实是，不需要研发去手动埋点。

一般会有一个 sdk 封装好各种逻辑, 然后业务方直接引用即可。

sdk中做的事情一般是监听所有页面事件, 上报所有点击事件以及对应的事件所在的元素，然后通过后台去分析这些数据。

业界有GrowingIO, 神策, 诸葛IO, Heap, Mixpanel等等商业产品

实现

1. 监听window元素

```
window.addEventListener("click", function(event){
    let e = window.event || event;
    let target = e.srcElement || e.target;
}, false);
```

2. 获取元素唯一标识 xPath

```
function getXPath(element) {
    // 如果元素有id属性，直接返回//*[@id="xPath"]
    if (element.id) {
        return '//*[@id=\"' + element.id + '\"]';
    }
    // 向上查找到body，结束查找, 返回结果
    if (element == document.body) {
        return '/html/' + element.tagName.toLowerCase();
    }
    let currentIndex = 1, // 默认第一个元素的索引为1
        siblings = element.parentNode.childNodes;


    for (let sibling of siblings) {
        if (sibling == element) {
            // 确定了当前元素在兄弟节点中的索引后, 向上查找
            return getXPath(element.parentNode) + '/' + element.tagName.toLowerCase() + '[' + (currentIndex
                ']';
        } else if (sibling.nodeType == 1 && sibling.tagName == element.tagName) {
            // 继续寻找当前元素在兄弟节点中的索引
            currentIndex++;
        }
    }
};
```

获取元素的位置

```
function getOffset(event) {
    const rect = getBoundingClientRect(event.target);
    if (rect.width == 0 || rect.height == 0) {
        return;
    }
    let doc = document.documentElement || document.body.parentNode;
    const scrollX = doc.scrollLeft;
    const scrollY = doc.scrollTop;
    const pageX = event.pageX || event.clientX + scrollX;
    const pageY = event.pageY || event.clientY + scrollY;

    const data = {
        offsetX: ((pageX - rect.left - scrollX) / rect.width).toFixed(4),
        offsetY: ((pageY - rect.top - scrollY) / rect.height).toFixed(4),
    };

    return data;
}
```

上报

```
window.addEventListener("click", function(event){
    const e = window.event || event;
    const target = e.srcElement || e.target;
    const xPath = getXPath(target);
    const offsetData = getOffset(event);

    report({ xPath,  ...offsetData});
}, false);
```

列表无限滚动方案

无限滚动, 首先应该想到两点：

1. 下拉到底, 继续加载数据并拼接
2. 数据太多, 要做虚拟列表展示

虚拟列表

虚拟列表的实现，实际上就是在首屏加载的时候，只加载可视区域内需要的列表项，当滚动发生时，动态通过计算获得可视区域内的列表项，并将非可视区域内存在的列表项删除。

1. 计算当前可视区域起始数据索引(startIndex)
2. 计算当前可视区域结束数据索引(endIndex)

3. 计算当前可视区域的数据，并渲染到页面中
4. 计算startIndex对应的数据在整个列表中的偏移位置startOffset并设置到列表上

滚动

由于只是对可视区域内的列表项进行渲染，为了保持列表容器的高度并可正常的触发滚动, 我们需要有一个元素展示真正渲染的数据, 一个元素撑开高度保证滚动, 一个容器

1. infinite-list-container 为可视区域的容器
2. infinite-list-phantom 为容器内的占位，高度为总列表高度，用于形成滚动条
3. infinite-list 为列表项的渲染区域

监听滚动

监听infinite-list-container的scroll事件，获取滚动位置scrollTop

可视区域高度：screenHeight
列表每项高度：itemSize
列表数据：listData
当前滚动位置：scrollTop

得出最终想要的数据

列表总高度listHeight = listData.length * itemSize
可显示的列表项数visibleCount = Math.ceil(screenHeight / itemSize)
数据的起始索引startIndex = Math.floor(scrollTop / itemSize)
数据的结束索引endIndex = startIndex + visibleCount
列表显示数据为visibleData = listData.slice(startIndex,endIndex)

当滚动后，由于渲染区域相对于可视区域已经发生了偏移，此时我需要获取一个偏移量startOffset，通过样式控制将渲染区域偏移至可视区域中。

偏移量startOffset = scrollTop - (scrollTop % itemSize);

无限滚动

当滚动触底, 就加载新一批数据, 拼接到原来的数据上

代码

代码