

## Q&A

1. 为什么要埋点？
2. 埋点方案？
  - a) 百度统计、谷歌统计、友盟、talkingdata
  - b) 小程序数据分析工具
    - i. 小程序数据助手（官方小程序工具）
    - ii. 腾讯移动分析 - 微信小程序统计 mta
    - iii. GrowingIO 小程序统计分析 ? 诸葛 IO
    - iv. 阿拉丁小程序统计工具
    - v. TalkingData 小程序统计分析
    - vi. 及策微信小程序统计
3. 不同埋点方案差异，优势与缺点？
  - a) 不同埋点类型差异
  - b) 不同埋点产品差异
4. 小程序和 web，app 端埋点异同？
5. 小程序埋点目标、架构、流程
6. 小程序埋点主要实现细节
7. 小程序埋点注意点

# 1. 为什么要埋点？

## 数据采集的重要性

### 1. 数据对于线上问题排查的作用：

- 用户行为数据还原“现场”，帮助分析和定位问题，提高问题定位效率
- 对于问题分析提供有力证据

### 2. 数据对于性能优化的作用：

- 帮助发现和监控在线业务的关键成功指标
- 帮助发现最需要优化环境及其优先顺序
- 帮助发现所面临的挑战的信息和改进决策
- 帮助提供对应用测试和优化更好的分析

### 3. 数据对于业务增长的作用：

- 帮助衡量市场营销效果
- 帮助发现激活转化效果的策略
- 帮助发现用户留存和用户活跃分析
- 帮助产品营收变现分析

# 2. 埋点方案？

a) 百度统计、谷歌统计、友盟、talkingdata

b) 小程序数据分析工具

- i. 小程序数据助手（官方小程序工具）
- ii. 腾讯移动分析 - 微信小程序统计 mta
- iii. GrowingIO 小程序统计分析 ? 诸葛 IO
- iv. 阿拉丁小程序统计工具
- v. TalkingData 小程序统计分析
- vi. 及策微信小程序统计
- vii. 自建埋点

● （仅针对小程序）小程序目前还不能使用百度统计、谷歌统计、友盟、talkingdata 这些主流统计工具；

● “小程序数据助手”当前功能模块包括数据概况、访问基础分析（用户趋势、来源分析、留存分析、时长分析、页面详情）、实时统计和用户画像（年龄性别、省份城市、终端机型），数据与小程序后台常规分析一致

1. 无法获知支付是否成功，只知道点击了支付
2. 不埋点不能进行指定过滤条件(有些页面需要埋点)
3. 可以统计次数、人数、点击量、但是不能统计总额
4. 转发数是基于小程序内转发的可以统计
5. 无法统计授权

- 腾讯移动分析推出的微信小程序分析工具，可以帮助开发者实时统计分析微信小程序流量概况、用户属性和行为数据等，辅助产品优化以及运营推广。
- 硅谷新一代数据分析产品 GrowingIO，上线国内首款无埋点小程序数据统计分析功能。帮助企业快速、低成本获取全量、实时用户行为数据，小程序推广事半功倍。
- 阿拉丁除了能显示小程序数据助手之外的数据，还可以追踪到某个页面的 pv / uv，转发率，打开率，更有渠道二维码，添加后就可以看到这个二维码的 pv 和 uv（颗粒度是人数 / 小时）。
- TalkingData 小程序统计分析，依赖微信小程序公开接口并打通自有数据，补充用户地区分布、机型分布、联网分布等多种用户属性。更多角度认识用户，更了解用户。TalkingData 小程序统计分析拓展了事件及转化分析，能准确了解用户在具体事件上的使用情况，掌握在流程的关键节点上的转化率；支持页面访问分析，掌握用户在页面的停留时长等指标，了解用户偏好及用户离开应用前的停留页面，便于针对性优化 App。
- 实时分析用户在微信小程序中的每一个互动行为，了解用户行为轨迹，结合用户设置的微信属性全面分析用户价值，结合场景的统计分析，提升小程序的使用效率，通过数据驱动用户的成长。

### 3. 不同埋点方案差异，优势与缺点？

- a) 不同埋点类型差异
- b) 不同埋点产品差异

#### 3.1 数据上报方式大体上可以归为三类：

1. 第一类是代码埋点，即在需要埋点的节点调用接口直接上传埋点数据，友盟、百度统计等第三方数据统计服务商大都采用这种方案；
2. 第二类是可视化埋点，即通过可视化工具配置采集节点，在前端自动解析配置并上报埋点数据，从而实现所谓的“无痕埋点”，代表方案是已经开源的 Mixpanel；国内较早支持可视化埋点的有 TalkingData、诸葛 IO，2017 年腾讯的 MTA 也宣布支持可视化埋点；
3. 第三类是“无埋点”，它并不是真正的不需要埋点，而是前端自动采集全部事件并上报埋点数据，在后端数据计算时过滤出有用数据，代表方案是国内的 GrowingIO。

- 可视化埋点其实可以算是无埋点的一个衍生，故这里主要对比代码埋点与无埋点。

#### 1 代码埋点或 Capture 模式的埋点缺点

对于数据产品来说：

1. 依赖人的经验和直觉判断。  
业务相关的埋点位置需要数据产品或者业务产品主观判断，技术相关的埋点则需要技术人员主观判断。

## 2. 沟通成本高

数据产品确定所需要的数据，需要提出需求与开发沟通，且数据人员对技术不是特别熟悉，还需与开发人员明确相关信息否能上报的可行性。

## 3. 存在数据清洗成本

随着业务更迭变化，之前主观判断所需的数据会存在更改变化，此时对之前打点的数据就需要手动清洗，且清洗的工作量占比并不小。

对于开发来说：

### 1. 开发人员精力消耗

埋点对于业务团队来说，常常被相关开发人员所诟病。开发技术人员不能只关注技术，还需分散精力做埋点这样高度重复且机械性的任务。

### 2. 埋点相关代码侵入性强，对系统设计和代码可维护性造成负面影响

大部分的业务相关数据点都需要手动埋点完成，埋点代码不得不与业务代码强耦合在一起。即使业界已有无埋点 sdk，数据产品关注的业务特殊点也逃离不了手动埋点。

在业务不断变化下对数据的需求变更，埋点相关代码也需要跟着变化。进一步增加开发和代码维护成本。

### 3. 易错、漏

由于人工打点存在主观意识差异，打点位置的准确度难控，且易漏数据

### 4. 存在打点流程成本

当数据漏采或错采时，又要经历一遍开发流程和上线流程，效率低下。

## 2 无埋点优势（略）

手动埋点让使用者可以方便地设置自定义属性、自定义事件。所以当你需要深入下钻，并精细化自定义分析时，比较适合使用手动埋点。

## 3 无埋点优势

与手动埋点相比较，无埋点优势便不用多解释。

### 1. 提高效率

### 2. 数据更全面，按需提取

### 3. 减少代码侵入

## 4 无埋点缺点

1. 对于业务特殊关注点的粒度小于 SDK 粒度时无法单纯靠 SDK 无埋点完全解决，可采用无埋点和埋点相结合，故我们的小程序无埋点 SDK 也提供手动埋点的 API 接口，完善数据的完整程度，进而解决更多的问题。

2. 无埋点的劣势是自定义属性不灵活，

3. 传输时效性差，数据可靠性欠佳，耗费网络流量，

4. 还会增加服务器负载，

5. 而且兼容性也不佳。

## 3.2 不同埋点方案上的差异性：

	官方工具	第三方工具	自建平台
沟通成本	中	高	低

可扩展性	N	N	高
数据挖掘	N	N	高
开发成本	低	中	高

### 第三方统计平台基本解决初步目标：

流量监控，用户行为日志收集，漏斗分析，用户画像...

痛点：

1. 非实时反馈
2. 数据波动难以定位，与第三方沟通成本高
3. 没有自己的核心数据存储

## 4. 小程序和 web 端埋点异同？

在 Web 中有 BOM 对象及 DOM 对象可以给我们进行全方位的操作，特别是 BOM 对象，可以进行全局事件拦截。而小程序类似 BOM 对象的主要有下面三个对象：App、Page、Component。

- **App**：App 对象有且只有一个，属于小程序的总控对象，App 下面包含着 n 个 Page 对象。
- **Page**：Page 则通过字面理解就是页面的总控对象，一个小程序可以有多个 Page，每一个 Page 对象对应一个页面，Page 下面可以包含 n 个 Component 及其他普通的小程序组件。
- **Component**：Component 则是自定义组件对象，类似于 Page，只明生命周期及一些特性上有所区别，自定义组件对象在实际项目中比较常用，主要用来解决小程序页面堆栈太小的问题。

小程序的事件机制也是采用与 JavaScript 一样的事件机制，捕获->执行->冒泡的机制，JavaScript 在 BOM 及 DOM 对象中都提供了 addEventListener 这种订阅/发布机制，使我们可以轻松的拦截所有的事件，又不影响现有的流程及代码，从而实现松耦合埋点方案。但是小程序没有提供 addEventListener 的订阅/发布机制，没有办法通过这样子的方案来实现。

### 1. 拦截器方式

翻了官网，发现没有办法统一的处理，那么只能退而求其次，减少对现有的代码的侵入，于是想到了拦截器的机制来实现。

看看下面小程序原生的代码

```

1. App({
2.   onLaunch(options) {
3.     // Do something initial when launch.
4.   },

```

```

5.   onShow(options) {
6.     // Do something when show.
7.   },
8.   onHide() {
9.     // Do something when hide.
10.  },
11.  onError(msg) {
12.    console.log(msg)
13.  },
14.  globalData: 'I am global data'
15. })

```

提供拦截器，这里只拿 App 的 onLaunch 事件进行示例，其他事件都差不多这样处理即可。

```

1. var appFilter = function(config){
2.   if(config.onLaunch){
3.     let _onLaunch = config.onLaunch;
4.     config.onLaunch = function(ops){
5.       // 在这里干埋点的事
6.       // 例如存储数据、上送数据
7.       _onLaunch.call(this); // 调用原来的执行逻辑
8.     }
9.   }
10.  return config;
11. }
12.
13.
14. //App 使用拦截器示例
15. App(appFilter({
16.   onLaunch(options) {
17.     // Do something initial when launch.
18.   },
19.   onShow(options) {
20.     // Do something when show.
21.   },
22.   onHide() {
23.     // Do something when hide.
24.   },
25.   onError(msg) {
26.     console.log(msg)
27.   },
28.   globalData: 'I am global data'
29. })
30. )

```

这个也是 JS 好玩的地方，一切皆为对象。通过新增一个 Function 对象，增强方法之后，把旧的 Function 对象进行替换。上面的 App 的 onLaunch 方法，经过 appFilter 过滤器的处理后，就替换成过滤器里面加了埋点事件的新方法了，对于开发人员来讲，不需要去改动原来 App.onLaunch 事件里面的代码，目的达到了，但是不够完美，那是否可以用这种方式来实现不侵入代码的埋点？答案是可以的。

## 2.增强扩展方式

利用新增一个 Function 对象，增强方法之后，把旧的 Function 对象进行替换这样的原理，依次将 App、Page、Component 这三个对象进行增强扩展，示例代码如下。

```
1. //先把原生的三个对象保存起来
2. const originalApp = App,
3.     originalPage = Page,
4.     originalComponent = Component;
5. //在原生的事件函数里面，添加数据埋点，并替换成新的事件函数
6. const _extendsApp = function (conf, method) {
7.     const _o_method = conf[method];
8.     conf[method] = function (ops) {
9.         //在此处进行数据埋点
10.         if (typeof _o_method === 'function') {
11.             _o_method.call(this, ops);
12.         }
13.     }
14. }
15.
16. //重新定义 App 这个对象，将原来的 App 对象覆盖
17. App = function(conf){
18.     //定义需要增强的方法
19.     const methods = ['onLaunch', 'onShow', 'onHide', 'onError']
20.
21.     methods.map(function (method) {
22.         _extendsApp(conf, method);
23.     })
24.     //另外增强扩展埋点上送的方法
25.     conf.william = {
26.         addActionData: function (ops) {
27.             console.log('addActionData');
28.         },
29.         addVisitLog: function (ops) {
30.             console.log('addVisitLog');
31.         }
32.     }
33.     return originalApp(conf);
34. }
```

35. *//Page 及 Component 对象类似 App 的处理即可*

至此，整个小程序的生命周期都在掌握之中了，可以按需采集对应的数据，并且对于开发人员来讲，还不需去修改及调整代码，松耦合埋点方案搞定。

### 3.增强扩展+订阅/发布

上面的实现方案虽然实现了松耦合，但是觉得还不够完美，埋点的动作必须要写在增加的方法里面，这样子可维护性较差，也不够灵活。解耦，引出 EventHub(基于订阅/发布模式实现的消息总线)。

```
1. //引用 EventHub
2. import EventHub from '../utils/eventhub.min';
3. //先把原生的三个对象保存起来
4. const originalApp = App,
5.     originalPage = Page,
6.     originalComponent = Component;
7. //在原生的事件函数里面，添加数据埋点，并替换成新的事件函数
8. const _extendsApp = function (conf, method) {
9.     const _o_method = conf[method];
10.    conf[method] = function (ops) {
11.        //在此处进行数据埋点
12.        //此处改成消息发布
13.        if (typeof EventHub !== "undefined") {
14.            EventHub.emit('app' + method, ops);
15.        }
16.        if (typeof _o_method === 'function') {
17.            _o_method.call(this, ops);
18.        }
19.    }
20. }
21.
22. //重新定义 App 这个对象，将原来的 App 对象覆盖
23. App = function(conf){
24.     //定义需要增强的方法
25.     const methods = ['onLaunch', 'onShow', 'onHide', 'onError']
26.
27.     methods.map(function (method) {
28.         _extendsApp(conf, method);
29.     })
30.     //另外增强扩展埋点上送的方法
```



```

31.   conf.william = {
32.     addActionData: function (ops) {
33.       console.log('addActionData');
34.     },
35.     addVisitLog: function (ops) {
36.       console.log('addVisitLog');
37.     }
38.   }
39.   return originalApp(conf);
40. }
41. //Page 及 Component 对象类似 App 的处理即可
这样子就可以把埋点处理的逻辑抽离到另外一个 JS 文件中去实现。
1.   EventHub.on('apponLaunch',function(ops){
2.     //在这里可以处理数据埋点的事
3.   })

```

## 延展性思考

基于上述的实现方案，我们除了增强生命周期，也可以像上面那样去增加公用的方法，例如 App.william.addActionData，更可以通过增强 setData 方法来进行数据溯源或者进行增量比较来提升性能等方式。

## 5. 小程序埋点目标、架构、流程

- a) 埋点目标，埋点分析方法、模型
  - 流量监控 ( pv,uv )
  - 用户行为日志
  - 用户画像
  - ab 测试
  - 漏斗分析
  - 留存分析
  - 错误分析
- b) 整体架构

## I 整体架构



### c) 前端 SDK 工作流程

## I 前端SDK工作流程



## 6. 小程序埋点主要实现细节

### 6.1 要做什么？

1. 生命周期监控
  - a) 参考 session 实现
2. 点击事件监控
  - a) 循环一遍 Page 里的参数，筛选出用户触发事件，函数外套一层装饰器
3. 数据拼接
  - a) 获取当前页面的 name，事件类型，描述信息
4. 数据上报
  - a) 数据过滤

### 6.2 一个简单的映射关系：

OnLanuch ——》 计入小程序

OnShow ——》 计入页面

Event ——》 页面内触发事件  
OnHide ——》 离开页面  
OnUpload  
OnHide ——》 结束 session

- 如果要完成一个完整的 session 极度依赖对于小程序生命周期的回调,这就需要我们对生命周期进行监听(扩展);完成上面所述的生命周期的监听基本完成了 90%的工作。
- 可以通过页面名称,类型,描述信息,三个维度确定一个具体的点

## 7. 小程序埋点注意点

- 预览图片,分享会触发页面级别的 onShow, onHide
- navigateTo/navigateBack 两种模式都会导致页面切换
- app 的 onHide 的时候并非退出了小程序
- 全量无埋点的数据量庞大,灰度上线时遇到过服务器过载导致服务器可用性下降的问题。后续对于数据上报的量有所控制,只自动上报关键节点数据,其他业务关注节点可通过接入初始化时针对性配置再上报,避免上报过多冗余数据。此外对于上报数据结构的设计也需要尤为注意,结构目标是要清晰、简洁、便于数据检索(区分)。
- 在 SDK 内部做好 try、catch,避免对业务可用性造成影响。
- SDK 本身会对业务性能造成一定成都影响,数据暂存放在了小程序的 localStorage 中,多次较频繁的存/取小程序的 localStorage 在业务方本身较耗费性能的情况下会暴露出操作卡顿问题。减少 localStorage 的存/取操作,只在页面关闭时未上传的数据才存入 localStorage

全埋点存在的问题:

大量的无效数据,难以归纳

难以携带数据信息,一般情况下只能记录点击的 pv, uv

力度不够,无法区分页面哪一个元素所触发的事件

一种实现思路:

