前端异常监控系统的落地

背景

在后端服务中，错误或者业务日志的记录是非常常见的一种习惯，它可以帮助开发者快速定位系统的状态、追查 bug，了解异常情况等。但是反观前端开发领域，成型的日志或者监控系统就非常的少见。

那么如何在自己公司自主搭建一款适合自己的业务需要的前端异常监控系统呢？本文就是一个实际项目落地的过程总结。

问题

之前百姓网Team的刘小杰同学出了一篇《浏览器端 JavaScript 异常监控》前端的分享和文章。描述了整个监控体系的实现方案，以及未来可能需要解决的一些问题。没有读过的小伙伴强烈推荐看完我这篇文章再去看一下这个分享。其实很多公司都有这个方面的尝试，包括一些开源的解决方案，比如腾讯的badjs，淘宝的JSTracker，阿里巴巴的FdSafe，支付宝的saijs，国外的sentry和对应的前端sdk ravenjs，包括对应的TraceKit，当你真正开始要动手做的时候，一大堆已有的解决方案其实早在3-4年前就已经被人提出，实现和应用了。

但是他们真的是每一个人的解决方案吗，答案肯定并不是，目前主流的前端监控系统，包括一些收费服务，国内比如fundebug，产品功能和sentry还真像，只不过他只关注前端，sentry关注所有的Error收集场景，他们解决的问题其实都是本文要说到的一些通用问题，但是针对到具体项目适合不适合，就要看业务方自己选择了。

到这里其实大家不难看出，前端异常监控系统的开发其实并不复杂，开源实现方案也颇多，技术实现成本并不难，痛点有但是并不是都不能解决，总结一下：

1. 前端SDK需要实现，主要是错误拦截，代理监控，上报策略，API设计，以及日志接口。
2. 上报的日志实现实时查询。
3. 监控日志可视化管理后台的开发。
4. 压缩后的单行文件如何定位源码错误。

以上5点是基本，甚至后期可以再自己扩展加入短信提醒，邮件提醒等成熟的产品功能。当然这其中会充斥着一些暗坑，明坑，以及你想自己定制的业务场景，那么下面就通过这篇文章，把我们的项目经验给大家一一道来。

SDK的实现

先说一些题外话，如果先不考虑他是一个report加拦截错误的SDK，当你要写一个前端SDK的时候，我们要做些什么？

最好的一个思路就是先写使用文档或者入门技巧的教程，这是我百试不爽的招数，因为只有当使用教程和入门文档写到完美的时候，你再去通过编码实现对应的API，那简直是事半功倍，代码实现完都会美滋滋的。

其次就是如果是多人合作开发，一定要先订好团队的代码规范，书写风格，项目的脚手架，测试用例以及前端项目必须要有的跨浏览器兼容自动化测试。

这里简单介绍一下，开始前的准备工作。

```
 "scripts": {
    "pretest": "jshint -c .jshintrc src",
    "test": "mocha --compilers js:babel-core/register -r jsdom-
global/register",
    "easysauce": "easy-sauce",
    "devbuild": "rollup -c rollup.config.js --environment
entry:src/index.js,dest:dist/ger.js",
    "testbuild": "rollup -c rollup.config.js --environment
entry:test/index.js,dest:test/index.build.js",
    "build": "rollup -c rollup.config.js --environment
entry:src/index.js,dest:dist/ger.min.js,uglify",
    "beautify": "node ./scripts/beatufyjs.js src test",
    "watch": "rollup -c  rollup.config.js -w",
    "start": "http-server -a 0.0.0.0 -p 8080 -s",
    "dev": "npm-run-all --parallel start watch"
  }
```


pretest是保证代码格式没有错误的，使用jshint进行校验。

test是跑mocha本地测试的，因为项目是使用ES6编写的，所以还需要用babel编译测试文件，因为是面向浏览器的，所以还需要加入jsdom-global插件让命令行支持dom，window，document等浏览器端的全局属性。

easysauce是用来进行连接saucelabs进行兼容性测试的。

devbuild，testbuild，build，分别是对应rollup三个编译入口生成开发版，生产版，以及提供给saucelabs测试的test文件的。

beautify是格式化统一用的，他和pre-git配合使用的，比较方便，commit之前做format处理。

watch，start和dev都是开发时构建环境用的命令，其实用npm-run-all就可以了，一键启动2个命令。

具体配置可以最后去看开源后的项目中的package.json文件，这里只是简单介绍下准备工作，目录结构设计原则也比较简单，根据文件夹名字就能理解意思了，主要是src源码目录下，入口是index.js，lib下是按照功能模块划分的几个类，最后通过混合继承的方式把GER构造器外置到全局属性下，因为rollup配置的是umd的格式，所以最后是可以兼容各种引用方式的。

下面说一下核心部分：

错误拦截

常见的方案就是对 `window.onerror` 进行拦截，这里需要注意几个点，一个是对当前页面的onerror做保存，拦截后再去回调，避免页面自身的onerror不触发，还一个是通过广播的形式，可以在外部对一些error做过滤不去上报，这里可以用自定义事件执行的返回值做控制。还有就是在error对象没有stack的时候，自己对错误调用栈进行递归记录，主要是用 `arguments.callee.caller` 来做栈递归，最后是对错误消息格式做统一封装。

最后再把这一次错误消息扔进队列等待上报即可。

跨域的问题无法拦截错误，在载入的时候对标签做特殊限制就行了，这个不强制，很多人会遇到这个坑，这里也不细说了，解决办法就是加入crossorigin属性，大部分的公司CDN也都支持配置Access-Control-Allow-Origin，所以问题并不大，默认配置跨域获取不到详细错误信息的错误，会直接忽略。

代理监控

onerror的上报方式很多很多教程都有写了，但是其实还有一种代理监控的方式可以替代错误拦截，那就是对已知的一些系统方法进行代理，或者一些框架的方法进行代理，这里我们参考了badjs的几个实现方案，简单讲一下。

首先可以对 `console` 对象下的所有方法进行分级，然后可以直接使用log等方法进行日志上报，最后再统一处理成错误日志的msg。

然后我们可以对 `setTimeout` , `setInterval` 做代理，检测定时器中的报错。

如果你的网站使用了jquery或者zepto，requirejs或者seajs，那么对对应的事件监听方法做拦截，或者define方法做拦截，也可以实现大部分的业务错误拦截。

仅仅这样就够了么？当然不是，通过函数拦截，函数中传入的参数，比如callbakc或者object，或者array，我们也会自动对代理的函数中的参数做检测，对应的函数增加 `try catch` 上报。

这些方法到底是怎么实现的呢？代码也很简单，我们先保存已有函数，然后对函数的参数做检测，再递归调用拦截方法，最后执行的时候，try catch住这个函数执行的过程，

再自动上报，当然这里还有一个需要注意的地方，就是在上报上要做一个延迟处理，避免上报过程中触发页面的onerror，造成重复上报，先对onerror置空，然后再throw，throw之后，重置回onerror。

上报策略

错误的拦截和监控的思路说完，上报的策略有人可能觉得非常简单，我们参考了badjs和ravenjs的上报策略，总结出了一些通用的配置和我们额外想到的配置。

比如代理功能的开关，有些用户可能默认只想拦截onerror，因为毕竟try catch浪费性能，再之后就是上报时同时触发多个错误时，把错误进行合并上报，或者延迟上报（避免线程抢占用户操作），错误日志接口url配置，过滤一些无法排错的错误类型，比如跨域的错误类型 `Script error` ，上报抽样的概率（大型网站一定要有），重复错误重复多少次之后就忽略上报（垃圾信息）。

额外的情况我们这里还扩展了几个，主要是平时在工作中，我们会遇到一些bug是用户的环境遇到了，但是我们遇不到的情况。说白了就是无法重现，然后等你联系用户的时候，可能用户莫名其妙的就好了。

说白了，本质问题就是如何保留异常现场，所以我们额外加了几个配置来设置本地存储，把你最近的N条错误记录保存在客户端，避免上报丢失或者因为概率问题无法上报丢错误。

这样在遭遇用户投诉时可以针对性的从客户端本地信息中获取以前的历史错误日志，避免你再去查询或者因为时间过期而日志失效的问题。

大家都见过一些大厂，对一些投诉客户端的异常收集页面，会给客户发一个网站URL，里面就会显示出客户端的所有信息以及cookie，ls等，这时这个本地错误存储的功能就可以帮上你定位错误了。

日志接口的设计

感觉上面说完配置项的一些功能后，那么上传日志接口基本格式也就确定了。

字段	类型	含义
userAgent	String	浏览器信息
currentUrl	String	错误发生页面URL
host	String	错误发生页面host
timestamp	Date	发生错误时间戳
projectType	String	客户端类型PC/Mobile
flashVer	Number	flash版本
title	String	错误页面标题
screenSize	String	分辨率
referer	String	页面来源
colNum	Number	错误列
rowNum	Number	错误行
msg	String	错误信息
level	level	错误级别
targetUrl	String	错误js文件
ext	Object	扩展信息可自定义，手工上报时可用

基本上涵盖用户的客户端信息，报错信息，页面信息，以及如果你是主动上报的话，也会提供一些额外你想加的一些当前上下文环境的信息，来辅助你调试错误。

说出来你可能不信，我就遇到过一次做大促活动，全公司的测试机都没问题，但是非常多的投诉说用户客户端出现错误的情况，我还真就是在实时的日志系统里通过log真实用户数据来定位到底是什么导致的问题。

比如 token值，比如userid，比如一些环境变量是否被正确初始化了，这些都可以加入到ext中，方便你定位无法重现的bug。

API设计

这一部分其实就和错误上报关系不大了，主要是考虑一个上报SDK的使用场景适合什么样的API。

首先我理解的API要包含几个方面，一个是获取当前属性状态，一个是可以调用的方法，还有就是方便解耦扩展的接口。

那么属性肯定要包含目前SDK的配置型状态，广播事件队列，错误信息栈，以及阻拦下来的错误队列。

然后方法的话就好说了，在别人已经初始化过SDK后，其他人或者部门可能需要调整你的默认参数， `set`，`get` 方法肯定是必须有了，他允许你在初始化后改变配置项。然后就是上报的方法，这里参考了console的所有方法，比如error，debug，info等方法都是可以调用上报的。

再然后就是如果用户主动上报也想只做收集，在特殊时间节点上报信息的话，可以提供send方法或者catchError方法，前者是发送方法，后者是收集方法，当然收集后你可以选择不发送。

最后就是对本地存储和事件的一些处理通用方法了。比如 `on`,`off`,`trigger`,`clear` 等。

关于事件，我这里设计了4个广播， `beforeReport`，`afterReport`，`tryError`，`error` ，看名字就明白了，你可以在上报前后和拦截代理的时候对错误做加工或者其他你想做的扩展。

上报的日志实现实时查询

说完了SDK的设计和实现思路，我们可以进行下一步环节，我们都知道，一个PV很高的网站，某一处的前端页面报错，那么上报的日志量就是非常大的，所以在SDK层面做了抽样和过滤，但是哪怕是这样，如果你的网站质量不高，那么上报的日志量也会非常大，传统的写入数据库方式或者写缓存方式肯定不行。上报服务一定是要做日志服务的。

而日志服务有一个很大的缺陷就是很难做实时处理，因为日志的保存和量都非常的大，如果我要对日志做一些排序，汇总，聚合，查询等操作，会非常耗时。

我们选择的架构目前是，SDK上传错误信息到前端机（前端机指的是服务层，这一层有负载均衡策略），然后前端机比如php或者java，对日志进行本机日志的IO的写入，然后有一台总的日志服务器通过拉取的方式把不同前端机收集到的日志做汇总写入ElasticSearch，最后使用ElasticSearch的API进行聚合，查询等操作，完成日志的实时可看，实时可查，以及统计和汇总分析。

ElasticSearch我们选择的是nodejs的API，直接安装npm的包就可以使用了，配置好ElasticSearch的地址和端口就可以使用了，主要用的几个方法其实就是 `search`，`msearch` 分别是查询和多条结果查询。还有就是search的时候elasticsearch支持的DSL语法，其中的aggregations是一个比较重要的点，他提供了聚合汇总日志的能力，打个比方，我要做一个7日某错误触发总数，就要用到了，再或者我要按照错误文件进行数量汇总甚至条件分组，也是需要用到的，这也是为什么我们不选择后端成型的那一套日志查询界面系统kibana而选择自己做的问题了。

因为这个系统除了关注错误日志，还更关注如何解决错误，以及对错误进行梳理和观测。

监控日志可视化管理后台的开发

大厂的朋友们每天除了做业务之外，可能业余精力会去自己开发一些管理平台，针对开发人员使用的，比如运维的机器配置，比如上线的版本管理，比如自动化工具管理界面，比如机器性能的监控报警等。

那么当然我们也可以自己开发前端使用的错误日志可视化平台咯！

本文不是教大家如何使用express和vue来开发web应用的，篇幅有限，我只能说我们的系统是使用这2个框架来进行的开发，配合elasticsearch的API来做接口，完成的。

这里重点讲一下开发个工具系统一般的套路：

1. 用户管理。
2. 权限管理。
3. 错误聚合以及展示
4. 错误详情展示

其实功能非常简单，任何后端语言都可以快速实现，为了避免申请大量公司资源，你可以选择ldap的登陆方式也可以选择纯文本的保存用户信息，反正内部使用也没多大的量，这样就不需要操作数据库了，因为数据接口都来自日志服务。

然后就是对错误，让我们用产品的维度来思考，开发人员怎么用着舒服了。

首先按照业务功能来区分，那么业务功能区分最主要的就是按照域名来区分了，以二级域名为分解，对错误进行汇总，给出错误的汇总聚合，比如某个域名下的错误状态，曲线图等。

查看详情时，再根据你的查询条件，比如时间，错误msg，错误页面，文件名等来进行列表的筛选。

最后进入到每一个错误信息展示页面，把错误信息，客户端信息等全部列出即可。

简单实用的一个小的sentry的代替系统就开发好了。

压缩后的单行文件如何定位源码错误

这个问题可能是太多人关注的一个点了，百姓网的团队给出的解决方案是生成不同的文件再配合sourceMap进行错误上报或者是在本地构建时，对函数进行try catch 上报代码时带上源码错误行号，无论是哪一种，其实都不适合我们现在的场景，就是侵入性太强，部门多，没人用，严重影响性能和效率，增加代码质量。

使用函数代理的方式可能能够缓解问题，但是一旦遇到不开函数代理功能的业务团队时，定位的错误就是1行，3000列的错误，并且报错信息是经过各种压缩后的，比如 `af is not defined` 这种报错，就非常的尴尬了。

其实解决办法我们换一个思路就好了，我们手上的条件，有报错的压缩的js文件，有错误行和列，有错误信息。

那么我们是否可以在可视化平台的后台增加这样一个功能：

先定位这个错误是否是一个压缩后的报错，再看这个报错能否根据报错信息快速定位问题，如果都不行。

那么你作为开发者，你一定拥有这个压缩后的js的源代码，然后这个js肯定也是你压缩的，对应你肯定也有他的sourceMap文件。

ok，到这里就思路很清晰了，通过在后台上传你的sourceMap，甚至上传你的源代码，选择压缩方式，平台本身就可以帮你产生对应的sourceMap，再通过转换，把单行的行和列转换成源码的行和列就可以了。而这些都可以做成全自动的，你只需要把源代码文件拖进web界面即可~！

关键代码在这里：

```
var fs = require('fs');
var sourcemap = require('source-map');
var smc = new
sourcemap.SourceMapConsumer(fs.readFileSync('./test.js.map','utf8
'));
var ret = smc.originalPositionFor({line:1,column:105});

console.log(ret);
```


只需用originalPositionFor方法把日志中的line和column传入，就可以反推定位出源码错误位置了，其他的步骤就不需要细说了。

总结

基本到这里本文就结束了，其中涉及到的SDK具体代码以及可视化平台的实现，前端VUE的实现，还有包括ElasticSearch的查询代码示例等，我们在内部完善成型之后会及时开源，虽然现在已经开源，但是文档和安装示例以及报表展示效果等还没有最后验证，预计在4月底之前就可以整体测试通过并上线了。

如果你迫不及待的想使用，或者想根据本文定制你司自己的前端js错误监控系统，欢迎参加后续我的chat，大家一起交流！

Chat实录：《小爝：前端监控实战解析》
