



前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈

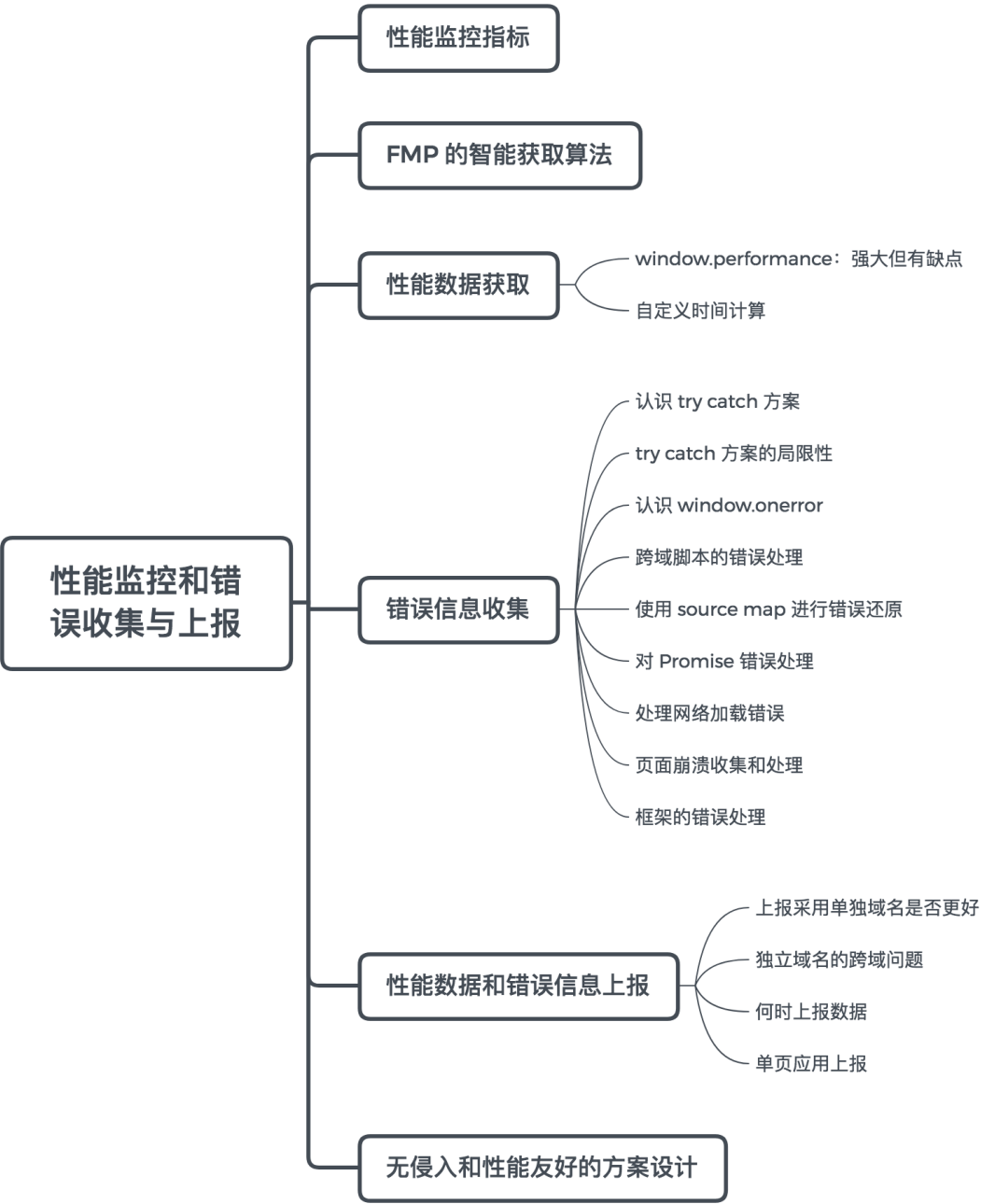
来自 Lucas ... · 盐选专栏

[查看详情 >](#)

## 性能监控和错误收集与上报（下）

上一节课我们学习了性能监控方面的知识。这一节来深入了解关于错误和异常的收集，并学习如何统一将这些信息进行上报。

在此之前，我们先回顾一下这个主题的知识点：



错误信息收集

提到错误收集方案，大家应该会首先想到两种：try catch 捕获错误和 window.onerror 监听。

认识 try catch 方案

我们先看一下 try catch 方案：

```
try {  
    // 代码块
```

```
} catch(e) {  
  // 错误处理  
  // 在这里，我们可以将错误信息发送给服务端  
}
```

这种方式需要开发者对预估有错误风险的代码进行包裹，这个包裹过程可以手动添加，也可以通过自动化工具或类库完成。自动化方案的基本原理是 AST 技术：比如 UglifyJS 就提供操作 AST 的 API，我们可以对每个函数添加 try catch，社区上 [foio](#) 的实现，就是一个很好的例子：

```
const fs = require('fs')  
const _ = require('lodash')  
const UglifyJS = require('uglify-js')  
  
const isASTFunctionNode = node => node instanceof  
UglifyJS.AST_Defun || node instanceof  
UglifyJS.AST_Function  
  
const globalFuncTryCatch = (source, errorHandler) => {  
  if (!_.isFunction(errorHandler)) {  
    throw 'errorHandler should be a valid function'  
  }  
  
  const errorHandlerSource = errorHandler.toString()  
  const errorHandlerAST = UglifyJS.parse('(' +  
errorHandlerSource + ')(error);')  
  var tryCatchAST = UglifyJS.parse('try{}catch(error){}')  
  const sourceAST = UglifyJS.parse(source)  
  var topFuncScope = []  
  
  tryCatchAST.body[0].catch.body[0] = errorHandlerAST  
  
  const walker = new UglifyJS.TreeWalker(function (node)  
{  
    if (isASTFunctionNode(node)) {  
      topFuncScope.push(node)  
    }  
  })
```

```

    })
    sourceAST.walk(walker)
    sourceAST.transform(transfer)

    const transfer = new UglifyJS.TreeTransformer(null,
        node => {
            if (isASTFunctionNode(node) &&
                _.includes(topFuncScope, node)) {
                var stream = UglifyJS.OutputStream()
                for (var i = 0; i < node.body.length; i++)
                {
                    node.body[i].print(stream)
                }
                var innerFuncCode = stream.toString()
                tryCatchAST.body[0].body.splice(0,
                tryCatchAST.body[0].body.length)
                var innerTyrCatchNode =
                UglifyJS.parse(innerFuncCode, {toplevel:
                tryCatchAST.body[0]})
                node.body.splice(0, node.body.length)
                return
                UglifyJS.parse(innerTyrCatchNode.print_to_string(),
                {toplevel: node});
            }
        })

    const outputCode = sourceAST.print_to_string({beautify:
    true})
    return outputCode
}

module.exports.globalFuncTryCatch = globalFuncTryCatch

```

我们从 `globalFuncTryCatch` 函数的第一个参数中获得目标代码 `source`，将其转换为 AST：

```
const sourceAST = UglifyJS.parse(source)
```

globalFuncTryCatch 函数的第二个参数为开发者定义的在出现错误时的响应函数，我们将其字符串化并转为 AST，并插入到 catch 块当中：

```
var tryCatchAST = UglifyJS.parse('try{}catch(error){}')
const errorHandlerSource = errorHandler.toString()
const errorHandlerAST = UglifyJS.parse('(' +
  errorHandlerSource + ')(error);')
tryCatchAST.body[0].catch.body[0] = errorHandlerAST
```

这样，借助于 globalFuncTryCatch，我们可以对每个函数添加 try catch 语句，并根据 globalFuncTryCatch 的第二个参数，传入自定义的错误处理函数（可以在该函数中进行错误上报）：

```
globalFuncTryCatch(inputCode, function (error) {
  // 此处是异常处理代码，可以上报并记录日志
  // ...
})
```

关键之处在于使用 UglifyJS 的能力，对 AST 语法树进行遍历，并转换：

```
const walker = new UglifyJS.TreeWalker(function (node) {
  if (isASTFunctionNode(node)) {
    topFuncScope.push(node)
  }
})
sourceAST.walk(walker)
sourceAST.transform(transfer)
```

最终再返回经过处理后的代码：

```
const outputCode = sourceAST.print_to_string({beautify:
true})
return outputCode
```

使用 try catch，我们可以保证页面不崩溃，并对错误进行兜底处理，这是一个非常好的习惯。

## try catch 方案的局限性

但是 try catch 处理异常的能力有限，对于运行时非异步错误，它并没有问题。但是对于：

语法错误

异步错误

try catch 就无法 cover 了。我们来看一个运行时非异步错误：

```
try {  
  a // 未定义变量  
} catch(e) {  
  console.log(e)  
}
```

可以被 try catch 处理。但是，将上述代码改动为语法错误：

```
try {  
  var a =\ 'a'  
} catch(e) {  
  console.log(e);  
}
```

就无法捕获。

我们再看一下异步的情况：

```
try {  
  setTimeout(() => {  
    a  
  })  
} catch(e) {  
  console.log(e)  
}
```

也无法捕获。

```
> try {
  setTimeout(() => {
    a
  })
} catch(e) {
  console.log(e)
}
```

2

✖ ▶ Uncaught ReferenceError: a is not defined  
at setTimeout (<anonymous>:3:6)

除非在 `setTimeout` 中再加一层 `try catch`：

```
> try {
  setTimeout(() => {
    try {
      a
    }
    catch (e) {
      console.log(e)
    }
  })
} catch(e) {
  console.log(e)
}
```

3

ReferenceError: a is not defined  
at setTimeout (<anonymous>:4:9)

总结一下，`try catch` 能力有限，且对于代码的侵入性较强。

## 认识 `window.onerror`

我们再看一下 `window.onerror` 对错误进行处理的方案：开发者只需要给 `window` 添加 `onerror` 事件监听，同时**注意需要将 `window.onerror` 放在所有脚本之前**，这样才能对语法异常和运行异常进行处理。

```
window.onerror = function (message, source, lineno, colno, error) {
  // ...
}
```

这里的参数较为重要，包含稍后需要上传的信息：

`message` 为错误信息提示

source 为错误脚本地址

lineno 为错误的代码所在行号

colno 为错误的代码所在列号

error 为错误的对象信息，比如 error.stack 获取错误的堆栈信息

window.onerror 这种方式对代码侵入性较小，也就不必涉及 AST 自动插入脚本。除了对语法错误和网络错误（因为**网络请求异常不会事件冒泡**）无能为力以外，无论是异步还是非异步，onerror 都能捕获到运行时错误。

但是需要注意的是，如果想使用 window.onerror 函数消化错误，需要显示返回 true，以保证错误不会向上抛出，控制台也就不会看到一堆错误提示。

## 跨域脚本的错误处理

千万不要以为掌握了这些，就万事大吉了。现实场景多种多样，比如**一种情况是：加载不同域的 JavaScript 脚本**，这样的场景较为常见，比如加载第三方内容，以展示广告，进行性能测试、错误统计，或者想用第三方服务等。

对于不同域的 JavaScript 文件，window.onerror 不能保证获取有效信息。由于安全原因，不同浏览器返回的错误信息参数可能并不一致。比如，跨域之后 window.onerror 在很多浏览器中是无法捕获异常信息的，要统一返回 Script error，这就需要 script 脚本设置为：

```
crossorigin="anonymous"
```

同时服务器添加 Access-Control-Allow-Origin 以指定允许哪些域的请求访问。

## 使用 source map 进行错误还原

到目前为止，我们已经学习了获取错误信息的「十八般武艺」。但是，如果错误脚本是经过压缩的，那么纵使你有千般本领，也无用武之地了，因为这样捕获到的错误信息的位置（行列号）就会出现较大偏差，错误代码也经过压缩而难以辨认。这时候就需要启用 source map。很多构建工具都支持 source map，比如



我们利用 webpack 打包压缩生成的一份对应脚本的 map 文件进行追踪，在 webpack 中开启 source map 功能：

```
module.exports = {  
  // ...  
  devtool: '#source-map',  
  // ...  
}
```

更多 source map 的内容，感兴趣的读者还可以参考以下资料：

[JavaScript Source Map 详解](#)

[Using source maps](#)

Webpack sourcemap 这里不是我们的重点，就不再展开。

## 对 Promise 错误处理

我们再来看一下针对 **Promise 的错误收集与处理**。我们都提倡养成写 Promise 的时候最后写上 catch 函数的习惯。ESLint 插件 eslint-plugin-promise 会帮我们完成这项工作，使用规则：catch-or-return 来保障代码中所有的 promise（被显式返回的除外）都有相应的 catch 处理。比如这样的写法：

```
var p = new Promise()  
p.then(fn1)  
p.then(fn1, fn2)  
function fn1() {  
  p.then(doSomething)  
}
```

是无法通过代码检查的。

这类 ESLint 插件基于 AST 实现，逻辑也很简单：

```
module.exports = {
  meta: {
    docs: {
      // ...
    },
    messages: {
      // ...
    }
  },
  create(context) {
    const options = context.options[0] || {}
    const allowThen = options.allowThen
    let terminationMethod = options.terminationMethod ||
'catch'

    if (typeof terminationMethod === 'string') {
      terminationMethod = [terminationMethod]
    }

    return {
      ExpressionStatement(node) {
        if (!isPromise(node.expression)) {
          return
        }

        if (
          allowThen &&
          node.expression.type === 'CallExpression' &&
          node.expression.callee.type ===
'MemberExpression' &&
          node.expression.callee.property.name === 'then'
&&
          node.expression.arguments.length === 2
        ) {
          return
        }
      }
    }
  }
}
```

```
    if (
      node.expression.type === 'CallExpression' &&
      node.expression.callee.type ===
'MemberExpression' &&
      terminationMethod.indexOf(node.expression.callee.
property.name) !== -1
    ) {
      return
    }

    if (
      node.expression.type === 'CallExpression' &&
      node.expression.callee.type ===
'MemberExpression' &&
      node.expression.callee.property.type ===
'Literal' &&
      node.expression.callee.property.value === 'catch'
    ) {
      return
    }

    context.report({
      node,
      messageId: 'terminationMethod',
      data: { terminationMethod }
    })
  }
}
}
```

如果读者对于 AST 和 ESLint 相关内容感兴趣，请关注课程《代码风格规范和背后技术设计》，会展开分析这方面的话题。

可能大家会想到，promise 实例的 then 方法中的第二个 onRejected 函数也能处理错误，这个和上面提到的 catch 方法有什么差别呢？事实上，我更加推荐 catch 方法，请看下面代码：

```
new Promise((resolve, reject) => {
  throw new Error()
}).then( () => {
  console.log('resolved')
}, err => {
  console.log('rejected')
  throw err
}).catch(err => {
  console.log(err, 'catch')
})
```

输出：rejected，在有 onRejected 的情况下，onRejected 发挥作用，catch 并未被调用。而当：

```
new Promise((resolve, reject) => {
  resolve()
}).then(() => {
  throw new Error()
  console.log('resolved')
}, err => {
  console.log('rejected')
  throw err
}).catch(err => {
  console.log(err, 'catch')
})
```

输出：VM705:10 Error at Promise.then (:4:9) "catch"，此时 onRejected 并不能捕获 then 方法中第一个参数 onResolved 函数中的错误。一经过对比，也许 catch 是进行错误处理更好的选择。但是，这两种方式各有特点，还是需要读者对 Promise 有较为深入的认识。

除此之外，对于 Promise 的错误处理，我们还可以注册对 Promise 全局异常的捕获事件 unhandledrejection：

```
window.addEventListener("unhandledrejection", e => {
  e.preventDefault()
})
```

```
    console.log(e.reason)
    return true
  })
```

这对于集中管理和错误收集更加友好。

## 处理网络加载错误

前面介绍的处理方式都是对已经在浏览器端的脚本逻辑错误进行的，我们设想用 `script` 标签，`link` 标签进行脚本或者其他资源加载时，由于某种原因（可能是服务器错误，也可能是网络不稳定），导致了脚本请求失败，网络加载错误。

为了捕获这些加载异常，我们可以：

除此之外，也可以使用 `window.addEventListener('error')` 方式对加载异常进行处理，注意这时候我们无法使用 `window.onerror` 进行处理，**因为 `window.onerror` 事件是通过事件冒泡获取 `error` 信息的，而网络加载错误是不会进行事件冒泡的。**

这里多提一下，**不支持冒泡的事件还有：**鼠标聚焦 / 失焦（`focus / blur`）、鼠标移动相关事件（`mouseleave / mouseenter`）、一些 UI 事件（如 `scroll`、`resize` 等）。

因此，我们也就知道 `window.addEventListener` 不同于 `window.onerror`，它通过事件捕获获取 `error` 信息，从而可以对网络资源的加载异常进行处理：

```
window.addEventListener('error', error => {
  console.log(error)
}, true)
```

那么，怎么区分网络资源加载错误和其他一般错误呢？这里有个小技巧，普通错误的 `error` 对象中会有一个 `error.message` 属性，表示错误信息，而资源加载错

误对应的 error 对象却没有，因此可以根据下面代码进行判断：

```
window.addEventListener('error', error => {
  if (!error.message) {
    // 网络资源加载错误
    console.log(error)
  }
}, true)
```

但是，也因为没有 error.message 属性，我们也就没有额外信息获取具体加载的错误细节，现阶段也无法具体区分加载的错误类别：比如是 404 资源不存在还是服务端错误等，只能配合后端日志进行排查。

到这里，我们简单做一个总结，分析 window.onerror 和 window.addEventListener('error') 的区别。

window.onerror 需要进行函数赋值：window.onerror = function() { //... }, 因此重复声明后会被替换，后续赋值会覆盖之前的值。这是一个弊端。

请看下图示例：

```
> window.onerror = function() {console.log(1)}
< f () {console.log(1)}
> window.onerror = function() {console.log(2)}
< f () {console.log(2)}
> window.dispatchEvent(new Event('error'))
2
web-8a293c1...js:1
< true
```

而 window.addEventListener('error') 可以绑定多个回调函数，按照绑定顺序依次执行，请看下图示例：

```
> window.addEventListener('error', () => {console.log(1)})
< undefined
> window.addEventListener('error', () => {console.log(2)})
< undefined
> window.dispatchEvent(new Event('error'))
1
web-8a293c1...js:1
2
web-8a293c1...js:1
< true
```

## 页面崩溃收集和處理

一个成熟的系统还需要收集崩溃和卡顿，对此我们可以监听 window 对象的 load 和 beforeunload 事件，并结合 sessionStorage 对网页崩溃实施监控：

```
window.addEventListener('load', () => {
  sessionStorage.setItem('good_exit', 'pending')
})

window.addEventListener('beforeunload', () => {
  sessionStorage.setItem('good_exit', 'true')
})

if(sessionStorage.getItem('good_exit') &&
  sessionStorage.getItem('good_exit') !== 'true') {
  // 捕获到页面崩溃
}
```

代码很简单，思路是首先在网页 load 事件的回调里：利用 sessionStorage 记录 good\_exit 值为 pending；接下来，在页面无异常退出前，即 beforeunload 事件回调中，修改 sessionStorage 记录的 good\_exit 值为 true。因此，如果页面没有崩溃的话，good\_exit 值都会在离开前设置为 true，否则就可以通过 sessionStorage.getItem('good\_exit') && sessionStorage.getItem('good\_exit') !== 'true' 判断出页面崩溃，并进行处理。

如果你的应用部署了 PWA，那么便可以享受 service worker 带来的福利！在这里，可以通过 service worker 来完成网页崩溃的处理工作。基本原理在于：service worker 和网页的主线程独立。因此，即便网页发生了崩溃现象，也不会影响 service worker 所在线程的工作。我们在监控网页的状态时，通过 navigator.serviceWorker.controller.postMessage API 来进行信息的获取和记录。

## 框架的错误处理

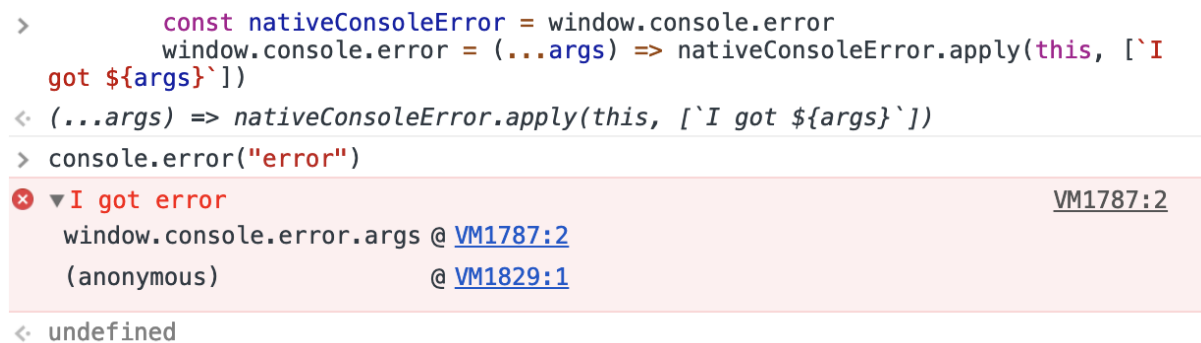
对于框架来说，React 16 版本之前，使用 unstable\_handleError 来处理捕获的错误；16 版本之后，使用著名的 componentDidCatch 来处理错误。Vue 中，提供了 Vue.config.errorHandler 来处理捕获到的错误，如果开发者没有配置

Vue.config.errorHandler，那么捕获到的错误会以 console.error 的方式输出。具体 API 的使用方式和框架特点，这里不再赘述。

上面提到框架会用 console.error 的方法抛出错误，因此可以劫持 console.error，捕获框架中的错误并做出处理：

```
const nativeConsoleError = window.console.error
window.console.error = (...args) =>
  nativeConsoleError.apply(this, [`I got ${args}`])
```

如下图：



```
> const nativeConsoleError = window.console.error
    window.console.error = (...args) => nativeConsoleError.apply(this, [`I
got ${args}`])
< (...args) => nativeConsoleError.apply(this, [`I got ${args}`])
> console.error("error")
✖ ▼ I got error VM1787:2
    window.console.error.args @ VM1787:2
    (anonymous) @ VM1829:1
< undefined
```

最后总结一下，我们大概处理了以下错误或者异常：

JavaScript 语法错误、代码异常

AJAX 请求异常 (xhr.addEventListener('error', function (e) { //... })))

静态资源加载异常

Promise 异常

跨域 Script error

页面崩溃

框架错误



在真实生产环境中，错误和异常多种多样，需要开发者格外留心，并对每一种情况进行覆盖。另外，除了性能和错误信息，一些额外信息，比如页面停留时间、长任务处理耗时等往往对分析网页表现非常重要。所有这些话题，欢迎大家在评论区展开讨论，也可以直接向我提问。对于错误信息采集和处理的介绍到此为止，接下来看一下数据的上报和系统设计。

## 性能数据和错误信息上报

数据都有了，我们该如何上报呢？可能有的开发者会想：「不就是一个 AJAX 请求吗？」，实际上还真没有这么简单，有一些细节需要考虑。

## 上报采用单独域名是否更好

我们发现，成熟的网站数据上报的域名往往与业务域名并不相同。这样做的好处主要有两点：

使用单独域名，可以防止对主业务服务器的压力，能够避免日志相关处理逻辑和数据在主业务服务器的堆积；

另外，很多浏览器对同一个域名的请求量有并发数的限制，单独域名能够充分利用现代浏览器的并发设置。

## 独立域名的跨域问题

对于单独的日志域名，肯定会涉及跨域问题。我们经常发现页面使用「构造空的 Image 对象的方式」进行数据上报。原因是请求图片并不涉及跨域的问题：

```
let url = 'xxx'
let img = new Image()
img.src = url
```

我们可以将数据进行序列化，作为 URL 参数传递：

```
let url = 'xxx?data=' + JSON.stringify(data)
let img = new Image()
img.src = url
```

## 何时上报数据

页面加载性能数据可以在页面稳定后进行上报。

一次上报就是一次访问，对于其他错误和异常数据的上报，假设我们的应用日志量很大，则有必要合并日志在统一时间，统一上报。那么什么情况下上报性能数据呢？一般合适的场景为：

页面加载和重新刷新

页面切换路由

页面所在的 Tab 标签重新变得可见

页面关闭

但是，对于越来越多的单页应用来说，需要格外注意数据上报时机，请看下文。

## 单页应用上报

如果切换路由是通过改变 hash 值来实现的，那么只需要监听 hashchange 事件，如果是通过 history API 来改变 URL，那么需要使用 pushState 和 replaceState 事件。当然一劳永逸的做法是进行 monkey patch，结合发布订阅模式，为相关事件的触发添加处理：

```
const patchMethod = type =>
  () => {
    const result = history[type].apply(this, arguments)
    const event = new Event(type)
    event.arguments = arguments
    window.dispatchEvent(event)
    return result
  }
```

```
history.pushState = patchMethod('pushState')
history.replaceState = patchMethod('replaceState')
```

我们通过重写 `history.pushState` 和 `history.replaceState` 方法，添加并触发 `pushState` 和 `replaceState` 事件。这样一来 `history.pushState` 和 `history.replaceState` 事件触发时，可以添加订阅函数，进行上报：

```
window.addEventListener('replaceState', e => {
  // report...
})
window.addEventListener('pushState', e => {
  // report...
})
```

## 何时以及如何上报

如果是在页面离开时进行数据发送，那么在页面卸载期间是否能够安全地发送完数据是一个难题：因为页面跳转，进入下一个页面，就难以保证异步数据的发送了。如果使用同步的 AJAX：

```
window.addEventListener('unload', logData, false);
const logData = () => {
  var client = new XMLHttpRequest()
  client.open("POST", "/log", false) // 第三个参数表明是同步
  的 XHR
  client.setRequestHeader("Content-Type",
    "text/plain;charset=UTF-8")
  client.send(data)
}
```

又会对页面跳转流畅程度和用户体验造成影响。

这时候给大家推荐一下 `sendBeacon` 方法：

```
window.addEventListener('unload', logData, false)

const logData = () => {
  navigator.sendBeacon("/log", data)
}
```

`navigator.sendBeacon` 就是天生来解决「页离开时的请求发送」问题的。它的几个特点决定了对应问题的解决方案：

它的行为是异步的，也就是说请求的发送不会阻塞向下一个页面的跳转，因此可以保证跳转的流畅度；

它在不受到极端「数据 size 和队列总数」的限制下，优先返回 `true` 以保证请求的发送成功。

目前 Google Analytics 使用 `navigator.sendBeacon` 来上报数据，请参考：[Google Analytics added sendBeacon functionality to Universal Analytics JavaScript API](#)。通过这篇文章，我们看到 Google Analytics 通过动态创建 `img` 标签，在 `img.src` 中拼接 URL 的方式发送请求，不存在跨域限制。如果 URL 太长，就会采用 `sendBeacon` 的方式发送请求，如果 `sendBeacon` 方法不兼容，则发送 AJAX post 同步请求。类似：

```
const reportData = url => {
  // ...
  if (urlLength < 2083) {
    imgReport(url, times)
  } else if (navigator.sendBeacon) {
    sendBeacon(url, times)
  } else {
    xmlhttpData(url, times)
  }
}
```

最后，如果网页访问量很大，那么一个错误发送的信息就非常多，我们可以给上报设置一个采集率：

```
const reportData = url => {
  // 只采集 30%
  if (Math.random() < 0.3) {
    send(data)
  }
}
```

这个采集率当然可以通过具体实际的情况来设定，方法多种多样。

## 无侵入和性能友好的方案设计

目前为止，我们已经了解了性能监控和错误收集的所有必要知识点。那么根据这些知识点，如何设计一个好的系统方案呢？

首先，这样的系统大致可分为四个阶段：



针对这几个阶段，我们聊一下关键方面的核心细节。

### 数据上报优化方面

借助 HTTP 2.0 带来的新特性，我们可以持续优化上报性能。比如：采用 HTTP 2.0 头部压缩，以减少数据传送大小；采用 HTTP 2.0 多路复用技术，以充分利用链接资源。

### 接口和智能化设计方面

我们可以考虑以下方面：

识别周高峰和节假日，动态设置上报采样率；

增强数据清洗能力，提高数据的可用性，对一些垃圾信息进行过滤；

通过配置化，减少业务接入成本；

如果用户一直触发错误，相同的错误内容会不停上报，这时可以考虑是否需要做一个短时间滤重。

### 实时性方面

目前我们对系统数据的分析都是后置的，如何做到实时提醒呢？这就要依赖后端服务，将超过阈值的情况进行邮件或短信发送。

在这个链路中，所有细节单独拿出来都是一个值得玩味的话题。打个比方，报警阈值如何设定。我们的应用可能在不同的时段和日期，流量差别很大，比如「点评」类应用，或「酒店预订」类应用，在节假日流量远远高于平时。如果报警阈值不做特殊处理，报警过于敏感，也许运维或开发者就要收到「骚扰」。业界上流行 3-sigma 的阈值设置，这是一个统计学概念。它表示对于一个正态分布或近似正态分布来说，数值分布在  $(\mu-3\sigma, \mu+3\sigma)$  中属于正常范围区间。这方面更多内容可以参考：

<https://www.investopedia.com/terms/t/three-sigma-limits.asp>

[What does a 1-sigma, a 3-sigma or a 5-sigma detection mean](#)

最后，我收集了业界几个性能监控和错误收集上报系统的分享，这些分享方案有的以 PPT 形式呈现，有的以源码分析实现，希望大家能够继续了解学习：

[前端异常监控解决方案研究](#)

[解密 ARMS 前端监控数据上报技术内幕](#)

[别再让你的 Web 页面在用户浏览器端裸奔](#)

[把前端监控做到极致](#)

[浏览器端 JS 异常监控探索与实践](#)

## 总结

本节梳理了性能监控和错误收集上报方方面面的内容。前端业务场景和浏览器的兼容性千差万别，因此数据监控上报系统要兼容多种情况。页面生命周期、业务逻辑复杂性也决定了成熟稳定的系统不是一蹴而就的。我们也要持续打磨，结合新技术和老经验，同时对比类似 Sentry 这样的巨型方案，探索更稳定高效的系统。

课程代码仓库：

<https://github.com/HOUCE/lucas-gitchat-courses>

点击查看下一节 

性能优化问题，老司机如何解决（上）