

20 | Sentry：线上错误与性能监控怎么处理？

2022-05-13 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 25:43 大小 23.55M



你好，我是蒋宏伟。

今天这一讲是我们社区生态篇的最后一讲，我们来聊聊 App 上线之后，如果遇到线上异常，或者是线上性能问题应该怎么办。

这是我们每个人都会遇到的问题。即便我们的代码在本地测试时没有问题，也有各种上线流程的保障，但由于线上环境的复杂性，也难免遇到各种奇奇怪怪的线上 Bug。我们既然不能完全避免线上 Bug，那么就需要尽可能地减少线上 Bug 对用户的影响，这就要用到线上监控系统了。

我曾经遇到过好几次老板甩来的 Bug，那时候我开发的 React Native 应用也没有接入线上监控，遇到问题只能绞尽脑汁在本地复现和解决。经历过那几次痛苦的 Debug 后，我就打算搞个 React Native 监控系统，从 2020 年开始至今，我一直在参与 58 大前端监控系统的设计和研发，其中 React Native 的监控也是由我负责的。

但是，从头搭建和迭代一个监控系统的成本非常高。如果你有线上错误和性能的监控需求，但公司内部没有现成的监控系统，那我的建议是直接用 Sentry。Sentry 提供了一个 [🔗 演示 Demo](#)，你可以直接打开看看它具体都有哪些功能。

而且 Sentry 的代码是开源的，它既支持你自己搭建，也支持付费直接使用。

如果想自己搭建的话，Sentry 后端服务是基于 Python 和 ClickHouse 创建的，需要自己使用物理机进行搭建，我们的兄弟团队，[🔗 转转团队](#)就是这么做的。如果想付费使用的话，可以参考 [🔗 Sentry 官方文档](#)先试用一下，如果老板也觉得不错，愿意付费使用，那就省去了自己搭建和维护 Python 服务的麻烦事了。

我也参考了 Sentry 官方文档，把我放在 Github 的实战课 Demo 也接入试用了一下，发现 Sentry 的接入过程很简单，文档也非常详细。如果你有接入需求，可以看看 [🔗 Sentry 的接入文档](#)。

因此这一讲，我不会和你介绍 Sentry 接入和使用方法，我想深入 Sentry 的底层和你聊聊实现线上错误和性能监控的原理。

这样，无论是你打算直接使用 Sentry，还是打算自研线上监控，都能有所收获。

基本信息的收集

首先，我们要明确的是，解决线上问题和解决本地问题的思路是不一样的。

在解决本地问题时，你不仅可以不断修改代码，反复尝试寻找解决方案，你还可以使用调试工具，比如 Flipper，它有打日志、打断点、查看性能火焰图等功能。然而在解决线上问题时，我们并不能反复尝试和使用调试工具。

你能借助的只有类似 Sentry 这样的线上监控工具。这些线上监控工具帮你记录了用户是谁，用户又是在什么情况下，出现了什么问题。你有了这些线上信息之后，才能确定问题的影响范围和紧急程度，也能更方便修复线上问题。

如果我们深入 Sentry 线上监控 SDK 的底层原理，你会发现它主要收集了三类线上数据：

- 用户是谁；

- 用户报错；
- 用户性能。

而深入学习原理的最好方式就是自己写一个简易版本。所以接下来，我们要先一起实现一个简易监控 SDK，把这些信息都收集上去，这样你就能够明白 Sentry 线上监控 SDK 的底层原理了。

当然，以上信息的收集必须遵守网信办的 [《网络数据安全条例（征求意见稿）》](#)，像设备唯一标示 IMEI、用户地理位置、运营商编号这些信息，我们是不能收集的。

你可能会问，不能收集设备唯一标示 IMEI，那我们怎么知道用户是谁啊？替代 IMEI 方案就是 **UUID**。

UUID 的全称是 Universally Unique Identifier，翻译过来就是通用唯一识别码，它是通过一个随机算法生成的 128 位的标识。生成两个重复 UUID 概率接近零，可以忽略不计，因此我们可以使用 UUID 代替与用户设备绑定的 IMEI 作为唯一标示符，该方法也是业内的通用方案之一。

在这一讲要实现的简易监控 SDK 中，你可以使用 UUID 算法配合 AsyncStorage 或 MMKV 生成一个用户 ID，具体实现方法如下：

 复制代码

```
1 import uuid from 'react-native-uuid';
2 import { MMKV } from 'react-native-mmkv'
3
4 // 用户唯一标示
5 let userId = ''
6
7 const storage = new MMKV()
8 const hasUserId = storage.contains('userId')
9
10 // 用户曾经打开过 App
11 if(hasUserId) {
12   userId = storage.get('userId')
13 } else {
14   // 用户第一次打开 App
15   userId = uuid.v4(); // ⇒ '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
16   storage.set('userId', userId)
17 }
```


如上代码中的 `react-native-uuid` 是 UUID 算法的 React Native 版本。`react-native-mmkv` 是持久化键值存储工具，MMKV 的性能比 `AsyncStorage` 更好，所以我这里就用它代替了 `AsyncStorage`。

生成用户唯一标示 `userId` 的思路是这样的。每次开打 App 时，先使用 `storage.contains('userId')`，判断一下在 MMKV 持久化键值存储中心是否存在 `userId`。如果 `userId` 的键值对不存在，那么该用户是第一次打开 App，这时使用 `uuid.v4` 算法生成一个 `uuid` 作为用户的唯一标示，并使用 `userId` 作为键名，调用 `storage.get` 方法将该键值对存在 MMKV 中。

如果存在 `userId` 的键值对，那么该用户是就不是第一次打开 App 了，这时直接使用 `userId` 这个键名，将第一次打开 App 生成的用户唯一标示，从 MMKV 中读出来就可以了。

有了 `userId` 这个用户唯一标示后，后台分析收集上来的线上信息时，就可以把线上报错、性能等信息和某个具体的用户挂上钩了，比如你可以通过对 `userId` 字段进行去重，来确定它影响了多少用户。

那么，有了 `userId` 就算解决了“用户是谁”这个问题了吗？

还不够，光有 `userId`，用户画像还是不够清晰，你还得知道他设备信息，这样用户画像才更立体。在 React Native 中，你可以通过 `react-native-device-info` 来获取设备信息，示例代码如下：

 复制代码

```
1 import DeviceInfo from 'react-native-device-info';
2
3 // API 提供了获取的能力，但根据 《网络数据安全条例（征求意见稿）》 是不能上报的，所以推荐使用
4
5 const androidIdPromise = DeviceInfo.getAndroidId()
6
7 // 将设备信息收集到一个 deviceInfo 对象中，统一上报。
8 const deviceInfo = {}
9 deviceInfo.systemName = DeviceInfo.getSystemName(); // iOS: "iOS" Android: "And
10 deviceInfo.systemVersion = DeviceInfo.getSystemVersion(); // iOS: "11.0" Androi
11 deviceInfo.brand = getBrand(); // iOS: "Apple" Android: "xiaomi"
12 deviceInfo.appName = DeviceInfo.getApplicationName(); // AwesomeApp
13 deviceInfo.appVersion = DeviceInfo.getVersion(); // iOS: "1.0" Android: "1.0"
```

在这个示例中，我使用了 `react-native-device-info` 库来获取设备信息，包括系统名字 `systemName`、系统版本 `systemVersion`、手机品牌 `brand`、应用名字 `appName`、应用版本 `appVersion`，并将它们放到了 `deviceInfo` 对象上，方便统一上报。

有了这些设备信息后，你定位特定机型、特定版本的问题就会容易很多。

普通 JavaScript 报错的收集

在回答完“用户是谁”这个问题后，下一个要回答的问题是“用户报了什么错”。用户应用报错了，报错信息我们是直接看不到的，要通过监控 **SDK** 收集上来才能看到。

那监控 **SDK** 如何收集这些报错信息呢？主要有三种方案：

- `ErrorUtils.setGlobalHandler`;
- `PromiseRejectionTracking`;
- `Error Boundaries`。

我们先来看 `ErrorUtils.setGlobalHandler`，它是用来处理 JavaScript 的全局异常的。如果某个 JavaScript 函数报错，并且该报错没有被捕获，该报错就会抛到全局中，示例代码如下：

 复制代码

```
1 function throwError(errorName){
2     throw new Error(errorName)
3 }
4
5 // 1. 被捕获的错误
6 try {
7     throwError('该错误会被 try catch 捕获')
8 } catch({})
9
10 // 2. 未被捕获的错误
11 throwError('该错误没有捕获，会抛到全局')
```

在这个示例中，第一个错误是被 `try catch` 捕获的错误，由于开发者已经对错误进行了处理，错误就不会再往外抛了，本地调试时也不会有红屏。第二个错误，开发者并没有 `try catch` 处理，该错误就会一层层往外抛，最终抛向全局作用域。

本地调试时，如果一个报错抛到了全局作用域，就会出现红屏。

本地调试的红屏其实是，React Native 框架在内部使用 `ErrorUtils.setGlobalHandler` 捕获到全局错误后，调用 `LogBox` 显示的红屏。红屏报错逻辑涉及框架源码的两个文件，分别是 [🔗 `setUpErrorHandling.js`](#) 和 [🔗 `ExceptionsManager.js`](#)，我省去了其中的处理细节，关键示例代码如下：

 复制代码

```
1 ErrorUtils.setGlobalHandler( (error: Error, isFatal?: boolean) => {
2   if (__DEV__) {
3     const LogBox = require('../LogBox/LogBox');
4     LogBox.addException({
5       message: error.message,
6       name: error.name,
7       componentStack: error.componentStack,
8       stack: error.stack,
9       isFatal
10    });
11  }
12 });
```

从这段代码可以看出，没有被 `try catch` 住的报错，会触发 `setGlobalHandler` 的回调，在该回调中会判断，如果是 `DEV` 环境，那么就用 `LogBox` 组件把报错的 `message`、`name`、`componentStack`、`stack`、`isFatal` 等信息展示出来，这样一来就可以在本地报错时，看到红屏的报错信息了。

看到这儿，你可能会问：既然 React Native 框架在本地调试时使用的是 `ErrorUtils.setGlobalHandler`，那么是否可以把这段逻辑改改用于线上错误监控呢？

这条思路很好。沿着这条思路想下去，我们有两个方案可以实现线上全局错误信息的上报，一种是使用 [🔗 `patch-package`](#) 修改 React Native 源码，另一种使用 `ErrorUtils.setGlobalHandler` 重写回调函数。显然，重写回调函数比直接修改源码侵入性更小，更利于后续维护，因此我选择了重写回调函数的方式，重写回调函数的示例代码如下：

 复制代码

```
1 const defaultHandler = ErrorUtils.getGlobalHandler && ErrorUtils.getGlobalHand
2
3 ErrorUtils.setGlobalHandler( (error: Error, isFatal?: boolean) => {
4   console.log(
```

```

5      `Global Error Handled: ${JSON.stringify(
6          {
7              isFatal,
8              errorName: error.name,
9              errorMessage: error.message,
10             componentStack: error.componentStack,
11             errorStack: error.stack,
12         },
13         null,
14         2,
15     )}` ,
16 );
17
18     defaultHandler(error, isFatal);
19 }):

```

在这段代码中，React Native 框架的代码会比我的代码先执行，所以它会先调用一次 `ErrorUtils.setGlobalHandler` 设置回调函数，而我的代码会在 React Native 框架代码执行之后再执行，并通过 `ErrorUtils.getGlobalHandler` 获取 React Native 框架设置的回调函数 `defaultHandler`，也就是在上个示例中演示的红屏报错的函数代码。

接着，我再次调用 `ErrorUtils.setGlobalHandler` 重新设置回调函数。在重置的回调函数中，我可以先处理自己的错误上报逻辑，这里用的是 `console.log` 代替的，然后再调用 React Native 框架的 `defaultHandler` 处理红屏报错。

Promise 报错的收集

以上是普通 JavaScript 报错的处理逻辑，但 Promise 报错的逻辑不一样。普通 JavaScript 错误，可以使用 `try catch` 捕获，但 promise 错误，`try catch` 是捕获不到的，需要用 `promise.catch` 来捕获。因此，二者全局的捕获机制也不一样。

React Native 提供了两种 Promise 捕获机制，一种是由新架构的 Hermes 引擎提供的捕获机制，另一种是老架构非 Hermes 引擎提供的捕获机制。这两种捕获机制，你都可以在 React Native 源码中找到，它涉及 [polyfillPromise.js](#)、[Promise.js](#)、[promiseRejectionTrackingOptions.js](#) 三个文件，我把其中关键代码摘出来了：

 复制代码

```

1  const defaultRejectionTrackingOptions = {
2      allRejections: true,
3      onUnhandled: (id: string, error: Error) => {},
4      onHandled : (id: string) => {}

```

```

5 }
6
7 if (global?.HermesInternal?.hasPromise?.()) {
8   if (__DEV__) {
9     global.HermesInternal?.enablePromiseRejectionTracker?.(
10       defaultRejectionTrackingOptions,
11     );
12   }
13 } else {
14   if (__DEV__) {
15     require('promise/setimmediate/rejection-tracking').enable(
16       defaultRejectionTrackingOptions,
17     );
18   }
19 }

```

在上面这个示例中，我们先声明了一个配置项 `defaultRejectionTrackingOptions`。这个配置项中最重要的就是 `onUnhandled` 回调函数，该回调函数是专门用来处理未被 `catch` 的 `Promise` 错误的。

接着我们再通过 `HermesInternal.hasPromise` 判断该 `React Native` 应用是否用的是 `Hermes` 引擎，如果返回 `true` 则为 `Hermes` 引擎，否则为其他引擎。如果是 `Hermes` 引擎，我们就使用 `Hermes` 引擎提供的 [enablePromiseRejectionTracker](#) 方法来捕获未被 `catch` 的 `Promise` 错误，如果不是 `Hermes` 引擎，则使用 [第三方 promise 库](#) 中 `rejection-tracking` 文件暴露的 `enable` 方法来捕获未被 `catch` 的 `Promise` 错误。

以上，就是 `React Native` 内部处理 `Promise` 的逻辑。

那么，如何将未被捕获的 `Promise` 错误上报呢？

答案就是再调用上一次 `Hermes` 引擎提供的 `enablePromiseRejectionTracker` 方法，或者再调用一次 `rejection-tracking` 文件暴露的 `enable` 方法，将框架的默认处理逻辑覆盖。示例代码如下：

 复制代码

```

1 const cusotomtRejectionTrackingOptions = {
2   allRejections: true,
3   onUnhandled: (id: string, error: Error) => {
4     console.log(
5       `Possible Unhandled Promise Rejection: ${JSON.stringify({
6         id,
7         errorMessage: error.message,

```



```

8       errorStack: error.stack,
9     },null,2)}\`,
10   },
11   onHandled : (id: string) => {}
12 }
13
14 if (global?.HermesInternal?.hasPromise?.()) {
15   if (__DEV__) {
16     global.HermesInternal?.enablePromiseRejectionTracker?.(
17       cusotomtRejectionTrackingOptions,
18     );
19   }
20 } else {
21   if (__DEV__) {
22     require('promise/setimmediate/rejection-tracking').enable(
23       cusotomtRejectionTrackingOptions,
24     );
25   }
26 }

```

开发者自定义的未捕获的 **Promise** 报错处理逻辑就是这样，和 **React Native** 框架内部的调用方法几乎一样。唯一不同的是，开发者可以在 **onUnhandled** 和 **onHandled** 回调中自定义错误的上报方法。在上述代码中，我用 **console** 代替了错误上报的逻辑。

组件 render 报错的收集

在 **React/React Native** 应用中，除了全局 **JavaScript** 报错和未捕获的 **Promise** 报错以外，还有一类报错可以统一处理，就是 **React/React Native** 的 **render** 报错。

在类组件中，**render** 报错指的是类的 **render** 方法执行报错；在函数组件中，**render** 报错指的就是函数本身执行报错了。

这里我展示了两类组件的报错形式，你可以看下：

 复制代码

```

1 function FunctionComponent() {
2   const [renderError, setRenderError] = useState(false)
3
4   if(renderError) throw Error('render 报错')
5
6   return <View></View>
7 }
8
9 function ClassComponent() {

```

```

10     state = {
11         renderError: false
12     }
13
14     render(){
15         return (
16             <View>
17                 {this.state.renderError && <span></span>}
18             </View>
19         )
20     }
21 }

```

你可以看到，第一个 **FunctionComponent** 示例是，当 **renderError** 状态由 **false** 变为 **true** 时，函数组件执行了到一半就会被 **throw Error** 报错打断。第二个 **ClassComponent** 示例是，当 **this.state.renderError** 状态由 **false** 变为 **true** 时，**render** 方法执行时发现了一个 **React Native** 中不存在的组件 **span**，整个渲染过程被中断。

类似这两种组件的 **render** 执行报错，在本地会抛红屏，在线上可能就是没有任何反应或者白屏。

那如何解决整个页面无响应或者白屏的问题呢？**React/React Native** 也提供了类似 **try catch** 的方法，叫做 **Error Boundaries**。**Error Boundaries** 是专门用于捕获组件 **render** 错误的。

不过，**React/React Native** 只提供了类组件捕获 **render** 错误的方法，如果是函数组件，必须将其嵌套在类组件中才能捕获其 **render** 错误。业内通常的做法是将其封装成一个通用方法给其他组件使用，比如 **Sentry** 就提供了 [ErrorBoundary 组件](#) 和 [withErrorBoundary 方法](#) 来帮助其他类组件或函数组件捕获 **render** 错误。

这里我提供了一个简易的 **ErrorBoundary** 组件的示例代码，你可以看看：

 复制代码

```

1  class ErrorBoundary extends React.Component {
2      constructor(props) {
3          super(props);
4          this.state = { hasError: false };
5      }
6
7      static getDerivedStateFromError(error) {
8          // 更新 state 使下一次渲染能够显示降级后的 UI
9          return { hasError: true };

```

```

10   }
11
12   componentDidCatch(error, errorInfo) {
13     // 你同样可以将错误日志上报给服务器
14     logErrorToMyService(error, errorInfo);
15   }
16
17   render() {
18     if (this.state.hasError) {
19       // 你可以自定义降级后的 UI 并渲染
20       return <View>404页面</View>;
21     }
22
23     return this.props.children;
24   }
25 }
26 // 使用方法
27 <ErrorBoundary>
28   <App/>
29 </ErrorBoundary>

```

这段代码中的 **ErrorBoundary** 是用于捕获 **App** 组件 **render** 执行报错的组件。

如果 **App** 组件 **render** 没有报错，那么会走 **return this.props.children** 的逻辑正常渲染；如果 **App** 组件 **render** 报错了，那么会触发 **getDerivedStateFromError** 回调，在 **getDerivedStateFromError** 回调中将控制是否有报错的开关状态 **hasError** 打开，并重新执行 **render** 渲染降级后的 **404** 页面，同时还会触发 **componentDidCatch** 回调。你可以在 **componentDidCatch** 回调中将组件的 **render** 错误上报。

在这个示例中，我用 **ErrorBoundary** 包裹的是 **App** 组件，也就是通常意义上的根组件，只要页面中出现任意组件的 **render** 错误，就会渲染一个“**404** 页面”。实际上，你也可以使用 **ErrorBoundary** 包裹局部组件，当某个局部组件出现错误时，使用其他局部组件将其替换。

到此，**JavaScript** 全局错误、**Promise** 未捕获错误和 **React Native** 组件的 **render** 错误，就都收集完成了。接下来我们开始进行性能收集。

性能收集

相对于错误收集，性能收集的优先级会低一些，因为错误影响的是能不能操作的问题，性能影响的是操作快点或慢点的体验问题。

早期的 Sentry 也是只收集错误不收集性能的，但现在也开始重视性能收集了。Sentry 主要收集的性能包括：

- App 启动耗时；
- 页面跳转耗时；
- 请求耗时。

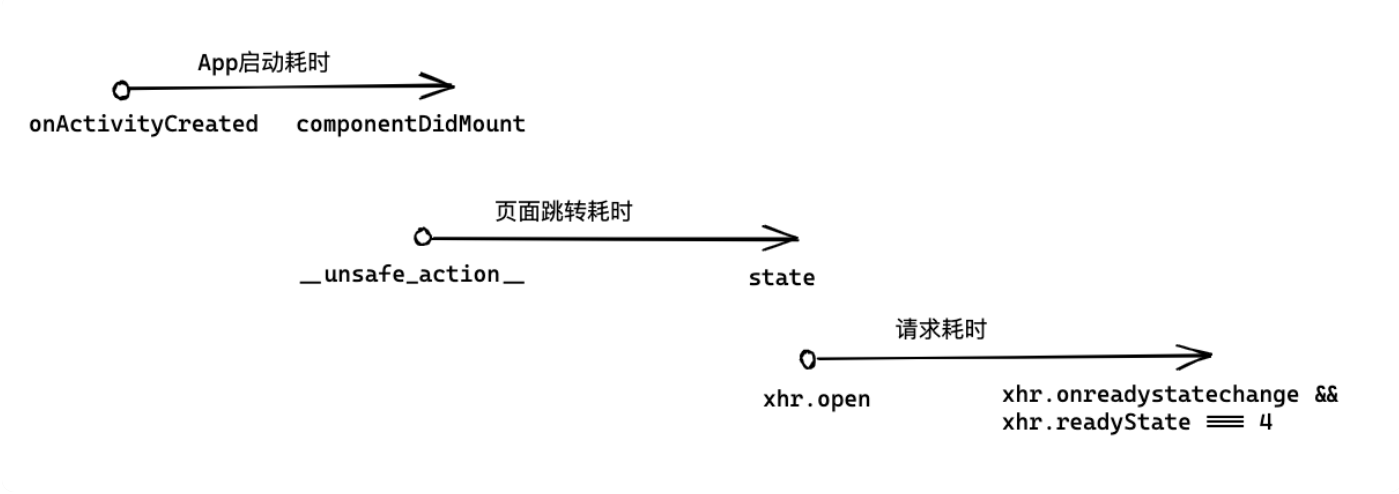
像 App 启动耗时、页面跳转耗时和请求耗时这些耗时类的统计原理，都是通过两个时间点的间隔计算出来的，原理示意如下：

复制代码

```
1 开始时间点 = Date.now()  
2 结束时间点 = Date.now()  
3 总耗时 = 结束时间点 - 起始时间点
```

在原理示例中，**总耗时等于结束时间点减去开始时间点的差值**，开始时间和结束时间点都是通过 `Date.now()` 获取的当前系统时间，单位是 `ms`。

因此，**耗时类统计的关键是找准开始时间点和结束时间点**。对于 App 启动耗时、页面跳转耗时和请求耗时的时间点，我画了一张示意图：



首先我们来看 **App 启动耗时**。App 启动的开始时间点是在 Native 组件的生命周期里面的。例如，在 Android 上就是 Fragment 所在的 Activity 启动完成后的 `onActivityCreated` 回调发生的时间点。App 启动的结束时间点是在 React/React Native 应用的生命周期里，也就是组件挂载完成 `componentDidMount` 回调发生的时间点。

虽然 App 只有一个，但页面、请求有很多个。统计 App 启动耗时可以在 Native 根组件或 React 根组件的生命周期里面统计，只需统计一次就行。但你不可能在每个页面的开始挂载和结束挂载的生命周期回调里面添加统计，也不可能在每个请求开始之前和回来之后添加统计。

那么，我们如何统计 App 中所有的页面跳转和请求耗时呢？

我们先来看**页面跳转耗时**怎么统计。如果你使用的是 React Navigation，那在每次页面跳转之前都需要下达跳转命令。在下达跳转命令的时候会触发 `__unsafe_action__` 事件，你可以在 `__unsafe_action__` 事件的回调中添加页面跳转耗时的开始时间点。在页面跳转完成后，页面的状态会发生改变，此时会触发 `state` 改变事件，此时添加结束时间点。

示例代码如下：

 复制代码

```
1 function App({navigation}) {
2
3   useEffect(()=>{
4     let startTime = 0
5
6     navigation.addListener('__unsafe_action__', (e) => {
7       startTime = Date.now()
8     });
9
10    navigation.addListener('state', (e) => {
11      const totalTime = Date.now() - startTime
12      console.log(`totalTime:${totalTime}`)
13    });
14  },[])
15
16  return <></>
17 }
```

从代码中你可以看到，我们无须在每个组件的声明周期里面都添加回调，只用在 App 根组件挂载后，直接监听导航命令触发的 `__unsafe_action__` 和 `state` 事件就可以完成页面跳转耗时的统计。

当然上面的示例代码只是列举了原理，还有些边界情况没有考虑到，如果你对其中细节感兴趣你可以查看一下 [@Sentry 的 ReactNavigation 部分的源码](#)。

最后，我们再一起来看下**请求耗时**如何统计，示例代码如下：

 复制代码

```
1 let startTime = 0
2
3 const originalOpen = XMLHttpRequest.prototype.open
4
5 XMLHttpRequest.prototype.open(function(...args){
6     startTime = Date.now()
7     const xhr = this;
8
9     const originalOnready = xhr.prototype.onreadystatechange
10
11     xhr.prototype.onreadystatechange = function(...readyStateArgs) {
12         if (xhr.readyState === 4) {
13             const totalTime = Date.now() - startTime
14             console.log(`totalTime:${totalTime}`)
15         }
16         originalOnready(...readyStateArgs)
17     }
18
19     originalOpen.apply(xhr, args)
20 })
```

因为 React Native 中的 `fetch` 或 `axios` 请求都是基于 [XMLHttpRequest](#) 包装的，所以要统计请求耗时，就要监听 `XMLHttpRequest` 的 `open` 事件，以及其实例 `xhr` 的 `onreadystatechange` 事件。在 `open` 事件中，记录请求开始的时间点，在 `onreadystatechange` 事件触发时且 `xhr.readyState` 等于 4 时记录请求的结束时间点。这里 `xhr.readyState` 等于 4 就代表下载操作已完成。

为了不破坏请求默认的 `open` 和 `onreadystatechange` 事件，我又保留了这些事件的默认回调，并在相应事件中继续调用和传参。

在我刚刚做前端开发时，如何使用 `XMLHttpRequest` 实现 `ajax` 异步请求是一道必考题，后来随着 `fetch` 和 `axios` 这些上层的 API 的普及，就很少有人直接操作底层的 `XMLHttpRequest` 了。但如果你想实现一些稍微底层的库，比如这一讲的线上监控，你就必须深入底层，把这些底层的 API 搞懂才行。

总结

今天这一讲，我和你介绍了实现一个简易监控 SDK 的思路。

你需要先知道用户是谁，在计算机的视角它会用一个唯一标识符 **uuid** 来代表用户，并且会记录该用户的设备信息。

能够统一收集的线上报错主要分为三类，**JavaScript** 全局报错、**Promise** 未捕获报错、组件 **Render** 报错，这些报错信息会和 **uuid**、设备信息一起上报到服务端。

同样，**App** 启动耗时、页面跳转耗时、请求耗时这类性能信息也会和 **uuid**、设备信息一起上报到服务端。

当服务端接收到这些从用户手机发来的错误和性能数据后，它会将这些数据进行处理、存储和展示，这就是线上监控的基本原理。


作业

1. 我在 **GitHub** 上的 [🔗 Demo](#) 中实现了一个简易的线上错误监控 **SDK**。请你根据这一讲中的性能监控的代码片段，为该简易 **SDK** 添加性能监控能力；
2. 如果要你来实现一个监控 **SDK**，除了文中提到了设备信息、报错信息、性能信息，你还会收集哪些维度的信息来帮助你排查问题？这些收集上来的信息，你又会通过什么方式将它们用起来呢？

欢迎在评论区和我们分享你的想法。我是蒋宏伟，咱们下一讲见。

分享给需要的人，Ta 订阅超级会员，你最高得 **50** 元

Ta 单独购买本课程，你将得 **20** 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | **Redux**：大型应用应该如何管理状态？

下一篇 21 | 混合应用：如何从零开始集成 **React Native**？

精选留言 (3)

写留言



python4

2022-05-14

实例代码中用同步的log来简单代替上报, 真实上报是个异步过程, 能讲一下异步上报需要注意的点么? 比如会不会阻塞业务代码运行, 需不需要空闲时间集中上报

作者回复: 因为报错通知的事件本身是异步的, 上报请求本身也是异步的, 所以不会阻塞, 也无需考虑集中上报。



1



郭智强

2022-05-18

老师您好, 您上面提到的那一段设置 `ErrorUtils.setGlobalHandler` 的代码我放到一个tsx 文件后, 那这个文件应该在哪个地方 `import` 进去? 我想在初始化的时候, 就把这个 `handler` 设置好

作者回复: 单独起个文件, 在该文件中执行, 然后把该文件在入口 `index.js` 文件中优先引入即可。



魑魅魍魉

2022-05-13

老师您好,
我们也在使用Sentry 的 `performance` 来监控产线应用的性能, 项目中使用的是React Navigation V5. 同时也是直接使用Sentry提供的 `ReactNavigationInstrumentation` 作为`routingInstrumentation`. 但是在Sentry 的 Dashboard上会有很多的Route Change的Transaction. 这是什么原因?

作者回复: 每次 `Route`, Sentry 都当作性能 `Transaction` 统计的, 可以看下 `Tracing` 相关的文档。

共 2 条评论 >

