Alibaba Cloud
Application Real-time Monitoring Service

Browser monitoring

Issue: 20200615
Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company, or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed due to product version upgrades, adjustments, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and the updated versions of this document will be occasionally released through Alibaba Cloud-authorized channels. You shall pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides the document in the context that Alibaba Cloud products and services are provided on an "as is", "with all faults" and "as available" basis. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not bear any liability for any errors or financial losses incurred by any organizations, companies, or individuals arising from their download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, bear responsibility for any indirect, consequential, exemplary, incidental, special, or punitive damages, including lost profits arising from the use or trust in this document, even if Alibaba Cloud has been notified of the possibility of such a loss.
5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please contact Alibaba Cloud directly if you discover any errors in this document.
# Document conventions

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚠️</td>
<td>A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.</td>
<td>⚠️ <strong>Danger:</strong> Resetting will result in the loss of user configuration data.</td>
</tr>
<tr>
<td>⚠️</td>
<td>A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.</td>
<td>⚠️ <strong>Warning:</strong> Restarting will cause business interruption. About 10 minutes are required to restart an instance.</td>
</tr>
<tr>
<td>📝</td>
<td>A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.</td>
<td>📝 <strong>Notice:</strong> If the weight is set to 0, the server no longer receives new requests.</td>
</tr>
<tr>
<td>📝</td>
<td>A note indicates supplemental instructions, best practices, tips, and other content.</td>
<td>📝 <strong>Note:</strong> You can use Ctrl + A to select all files.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Closing angle brackets are used to indicate a multi-level menu cascade.</td>
<td>Click <strong>Settings</strong> &gt; <strong>Network</strong> &gt; <strong>Set network type</strong>.</td>
</tr>
<tr>
<td><strong>Bold</strong></td>
<td>Bold formatting is used for buttons, menus, page names, and other UI elements.</td>
<td>Click <strong>OK</strong>.</td>
</tr>
<tr>
<td><strong>Courier font</strong></td>
<td>Courier font is used for commands.</td>
<td>Run the <code>cd /d C:/windows</code> command to enter the Windows system folder.</td>
</tr>
<tr>
<td><strong>Italic</strong></td>
<td>Italic formatting is used for parameters and variables.</td>
<td><code>bae log list --instanceid Instance_ID</code></td>
</tr>
<tr>
<td>[] or [a</td>
<td>b]</td>
<td>This format is used for an optional value, where only one item can be selected.</td>
</tr>
<tr>
<td>Style</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>{} or {a</td>
<td>b}</td>
<td>This format is used for a required value, where only one item can be selected.</td>
</tr>
</tbody>
</table>
## Contents

- **Legal disclaimer**
- **Document conventions**
- **1 Overview**
- **2 Get started with Browser Monitoring**
  - 2.1 Overview of Getting Started with Browser Monitoring
  - 2.2 For Web applications
    - 2.2.1 Install the ARMS agent through npm
    - 2.2.2 Install the ARMS agent through CDN
  - 2.3 For Weex
    - 2.3.1 Perform browser monitoring in the Weex environment
  - 2.4 For mini programs
    - 2.4.1 Monitor DingTalk mini programs
    - 2.4.2 Monitor Alipay mini programs
    - 2.4.3 Monitor WeChat mini programs
    - 2.4.4 Monitor other mini programs
- **3 Console functions**
  - 3.1 Page speed
  - 3.2 Browser monitoring dashboard
  - 3.3 API request monitoring
  - 3.4 Custom statistics
- **4 Tutorials**
  - 4.2 Diagnose slow loading of pages
  - 4.4 Front-to-back tracing
- **5 Advanced options**
  - 5.1 Advanced browser monitoring scenarios
- **6 Configuration items of the browser monitoring SDK**
- **7 Methods user guide**
- **8 Statistical metrics**
1 Overview

The browser monitoring function of Application Real-Time Monitoring Service (ARMS) monitors experience data for Web pages, including the page loading speed (speed test), page stability (JS errors), and external service calling success rate (API), to check the healthiness of Web pages.

In addition, based on the Application Performance Index (APDEX) algorithm, ARMS calculates the satisfaction rating of application sites and pages, allowing you to learn user experience on the sites and pages.

Why is browser monitoring necessary?

When a user accesses a service, the whole process can be divided into three phases: page production time (server status), page load time, and page run time.

To make sure stable online services, the running status of services is monitored on the server. Currently, the server monitoring system is quite mature, but the monitoring of page load time and run time is far from satisfactory. For example:

- You cannot capture the access errors of users immediately.
- You do not know the actual response time for users from different countries or regions to access your website.
- You have no idea about the performance and success rate of asynchronous data calls of each application.
Solution

The ARMS browser monitoring platform monitors the status of page load time and run time, and reports data, such as the page load performance, runtime exceptions, and API call status and consumed time, to the logger. Then, the platform monitors the access of all real online users based on the rich real-time log analysis and processing services provided by ARMS. Finally, the platform displays results on reports to help you detect and diagnose problems promptly.
### Browser and platform compatibility

<table>
<thead>
<tr>
<th>Browser/Platform</th>
<th>Supported version</th>
<th>Automatic reporting</th>
<th>Active reporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>Safari 9+</td>
<td>###</td>
<td>###</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>Google Chrome 49+</td>
<td>###</td>
<td>###</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>IE 9+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Microsoft Edge</td>
<td>Microsoft Edge 12+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Firefox</td>
<td>Firefox 36+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Opera</td>
<td>Opera 43+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Safari for iOS</td>
<td>Safari for iOS 9.2+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Android browser</td>
<td>android_webkit 4.4.2+</td>
<td></td>
<td>###</td>
</tr>
<tr>
<td>Weex</td>
<td>Weex 0.16.0+</td>
<td></td>
<td>###</td>
</tr>
</tbody>
</table>
2 Get started with Browser Monitoring

2.1 Overview of Getting Started with Browser Monitoring

ARMS Browser Monitoring works in Web, Weex, and Mini Programs use cases. Read the corresponding documents to get started with Browser Monitoring.

For Web

- Install the ARMS agent through CDN
- Install the ARMS agent through npm

For Weex

Perform browser monitoring in the Weex environment

For Mini Programs

- Monitor DingTalk mini programs
- Monitor Alipay mini programs
- Monitor WeChat mini programs
- Monitor other mini programs

2.2 For Web applications

2.2.1 Install the ARMS agent through npm

To use the browser monitoring function of Application Real-time Monitoring Service (ARMS) to monitor Web applications, you must install the ARMS agent through CDN (Content Delivery Network) or npm (Node Package Manager). This topic describes how to install the ARMS browser monitoring agent for Web applications through npm.

Install the npm package

Install the npm package named alife-logger.

```
npm install alife-logger --save
```

Use npm to install the agent

Initialize the SDK
Initialize the SDK in the BrowserLogger.singleton mode.

```javascript
const BrowserLogger = require('alife-logger');
// Use BrowserLogger.singleton(conf) config to load config configurations.
const __bl = BrowserLogger.singleton(
  pid: 'your-project-id',
  imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?', // Set the URL to which logs are uploaded. If the application is deployed in Singapore, set it to 'https://arms-retcode-sg.aliyuncs.com/r.png?'.
  // The other config configurations.
);
```

When the ARMS browser monitoring agent is connected through npm, the Web SDK automatically generates a user ID (UID) to collect information such as the number of unique visitors (UVs). The automatically generated UID can be used to identify a user but does not have service attributes. If you want to customize a UID, add the following content to the code:

```javascript
uid: 'xxx', // The UID used to identify a user, which is set based on businesses.
```

An example is as follows:

```javascript
const BrowserLogger = require('alife-logger');
// Use BrowserLogger.singleton(conf) config to load config configurations.
const __bl = BrowserLogger.singleton(
  pid: 'your-project-id',
  uid: 'xxx', // The UID used to identify a user, which is set based on businesses.
  imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?', // Set the URL to which logs are uploaded. If the application is deployed in Singapore, set it to 'https://arms-retcode-sg.aliyuncs.com/r.png?'.
  // The other config configurations.
);
```

**Method description**

@static singleton() for a single-instance object

**Note:**

This method applies only to npm.

Call parameter description: BrowserLogger.singleton(config,prePipe)

It is a static method that returns a single-instance object. The loaded config and prePipe parameters take effect only for the first call. Only generated instances are returned for subsequent calls.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>Object</td>
<td>The site configuration. For other configurations, see configurations of #config.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>prePipe</td>
<td>Array</td>
<td>The pre-reported content.</td>
<td>No</td>
<td>None</td>
</tr>
</tbody>
</table>

This method can be used to initialize the SDK at the application entry point and obtain an instance during each call.

**Other reporting methods**

Use BrowserLogger.singleton to obtain instances.

```javascript
const __bl = BrowserLogger.singleton();
```

For the usage of other methods of __bl, see [Methods user guide](#).

**Config configurations**

Config configurations are the same as the CDN mode. For more information, see [Configuration items of the browser monitoring SDK](#).

**Pre-report data**

Some data must be reported for part of the logic that is executed before BrowserLogger.singleton() is called.

```javascript
const BrowserLogger = require('alife-logger');
// Same as the pipe structure of the CDN mode.
const pipe = [
  // Report the HTML content of the current page as an API.
  ['api', '/index.html', true, performance.now, 'SUCCESS'],
  // After SDK initialization is complete, enable automatic single-page application (SPA) resolution.
  ['setConfig', {enableSPA: true}]
];
const __bl = BrowserLogger.singleton({pid:'Unique site ID'},pipe);
```

### 2.2.2 Install the ARMS agent through CDN

To use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor Web applications, you must install the ARMS agent through Content Delivery
Network (CDN) or Node Package Manager (NPM). This topic describes how to install the ARMS browser monitoring agent for Web applications through CDN.

Procedure

1. Log on to the ARMS console. In the left-side navigation pane, choose Application Monitoring > Applications.

2. In the left-side navigation pane, choose Browser Monitoring. On the Browser Monitoring page, click Create New Site in the upper-right corner.

3. In the Create New Site dialog box, set the site type to Web, enter the site name, and click OK.

4. On the Settings page, select required options in the SDK Extension Configuration section.
   - Disable Automatic API Reporting: If you select this option, you need to manually call the __bl.api() method to report the API request success rate.
   - Enable Automatic SPA Parsing: If you select this option, ARMS monitors the hashchange event on the page and automatically reports page views (PVs). It is applicable to single-page application (SPA) scenarios.
   - Enable Data Collection of First Meaningful Paint: If you select this option, ARMS collects First Meaningful Paint (FMP) data.
   - Enable Page Resources Reporting: If you select this option, static resources loaded on the page are reported upon the triggering of the onload event.
   - Associate with Application Monitoring: If you select this option, API requests are associated with back-end application monitoring in end-to-end mode.

5. In the Copy / Paste BI Probe section, copy and paste the provided code to the first line in the <body> element on the HTML page, and restart the application.

When the ARMS browser monitoring agent is installed through CDN, the Web SDK automatically generates a user ID (UID) to collect information such as the number of
unique visitors (UVs). The automatically generated UID can be used to identify a user but does not have service attributes. If you want to customize a UID, add the following content to `config` in the code:

```javascript
uid: 'xxx', // The UID used to identify a user, which is set based on businesses.
```

An example is as follows:

```html
<script>
!(function(c,b,d,a){c[a]||(c[a]={});c[a].config={pid:"xxx",appType:undefined,imgUrl:"https://arms-retcode.aliyuncs.com/r.png?", uid: "xxxx");
with(b)with(body)with(insertBefore(createElement("script"),firstChild))setAttribute("crossorigin",""._src=d)
})(window,document,"https://retcode.alicdn.com/retcode/bl.js","__bl");
</script>
```

**Notice:**
If you modify any information in the SDK Extension Configuration section, the code changes. Copy and paste the code again.

**What's next**

The ARMS browser monitoring function also provides other SDK configuration items to meet your requirements. For more information, see Configuration items of the browser monitoring SDK.

**Overview of Getting Started with Browser Monitoring**

ARMS Browser Monitoring works in Web, Weex, and Mini Programs use cases. Read the corresponding documents to get started with Browser Monitoring.

**Configuration items of the browser monitoring SDK**

**Install the ARMS agent through npm**

To use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor Web applications, you must install the ARMS agent through CDN (Content Delivery Network) or npm (Node Package Manager). This topic describes how to install the ARMS browser monitoring agent for Web applications through npm.

**Perform browser monitoring in the Weex environment**

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) in the Weex environment.

**2.3 For Weex**
2.3.1 Perform browser monitoring in the Weex environment

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) in the Weex environment.

Import the NPM package

To use ARMS browser monitoring in the Weex environment, run the following command in the project to import the Node Package Manager (NPM) package named alife-logger and use the dedicated WeexLogger module to report logs:

```
npm install alife-logger --save
```

Initialize the SDK

Create the monitor.js file in the /utils directory and initialize the SDK according to the following sample code.

---

Note:

To call the `singleton(props)` method at the Weex application entry point to obtain instances, set the configurations in the input `props`. For more information, see `General method: @static singleton()`, `General method: setPage()`, and `General method: setConfig()`.

---

```javascript
// in monitor.js
import WeexLogger from 'alife-logger/weex';

const fetch = weex.requireModule('stream').fetch;
const serialize = (data) => {
  data = data || {};
  var arr = [];
  for (var k in data) {
    if (Object.prototype.hasOwnProperty.call(data, k) && data[k] !== undefined) {
      arr.push(k + '=' + encodeURIComponent(data[k]).replace(/\(/g, '%28').replace(/\)/g, '%29'));
    }
  }
  return arr.join('&');
}

// Initialize the SDK.
const wxLogger = WeexLogger.singleton({
  pid: 'your-project-id',
  uid: 'zhangsan', // Login uid, for UV report
  page: 'Lazada | Home', // Initial page name, if passed, SDK will send a PV log after initialization completed
  imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?', // Set the URL to which logs are uploaded. If you want to deploy the SDK in Singapore, change it to 'https://arms-retcode-sg.aliyuncs.com/r.png?'.
  // Set the GET method for log reporting as follows:
  sendRequest: (data, imgUrl) => {
    const url = imgUrl + serialize(data);
    fetch({
      method: 'GET',
      url
```

```
Report logs

Call the corresponding reporting methods to report logs based on instances.

```
// in some biz module
import wxLogger from '/utils/monitor';
wxLogger.api('/search.do', true, 233, 'SUCCESS');
```

General method: `@static singleton()`

`@static singleton()` is a static method used to return a single instance. `props` takes effect only upon the initial call. The following table describes the common parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>String</td>
<td>The ID of the site.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>page</td>
<td>String</td>
<td>The page name after initialization.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>uid</td>
<td>String</td>
<td>The ID of the user.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>imgUrl</td>
<td>String</td>
<td>The URL to which logs are uploaded, which ends with a question mark (?). [DO NOT TRANSLATE]</td>
<td>No</td>
<td>None</td>
</tr>
</tbody>
</table>

This method can be used to initialize the SDK at the application entry point. See Initialize the SDK for example code.
General method: setPage()

setPage() is used to set the name of the current page and report the PV logs once (default).

```javascript
import wxLogger from '/utils/monitor';
// ...
wxLogger.setPage(nextPage);
```

Table 2-2: wxLogger.setPage(nextPage) call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextPage</td>
<td>String</td>
<td>The page name.</td>
<td>Yes</td>
<td>None</td>
</tr>
</tbody>
</table>

General method: setConfig()

setConfig() is used to modify some configurations after SDK initialization. The configuration method is the same as that of singleton().

```javascript
import wxLogger from '/utils/monitor';
// ...
wxLogger.setConfig(next);
```

Table 2-3: wxLogger.setConfig(next) call parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>Object</td>
<td>The configuration item to be modified and its value.</td>
<td>Yes</td>
<td>None</td>
</tr>
</tbody>
</table>

Log reporting methods

For more information, see the log reporting methods in Methods user guide.

2.4 For mini programs
2.4.1 Monitor DingTalk mini programs

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor DingTalk mini programs. It also demonstrates the general configurations, methods, and advanced scenarios.

Procedure

To monitor the mini programs, you need to perform at least the three steps: introducing and initializing the Node Package Manager (NPM) package, reporting logs, and setting security domains.

1. Introduce and initialize the NPM package.
   a. Introduce the NPM package named alife-logger in the DingTalk mini program project to facilitate log reporting.

   ```
   npm install alife-logger
   ```
   
   b. Add the following information to the monitor.js file in the /utils directory to initialize the NPM package.

   ```
   import EAppLogger from 'alife-logger/eapp';
   const Monitor = EAppLogger.init({
     pid: 'xxx',
     region: 'cn', // The region where the application is deployed. Set it to cn if the application is deployed in China, to sg if the application is deployed near Singapore outside China, and to us if the application is deployed near the United States outside China.
   });
   
   export default Monitor;
   ```

   Note:
   You can specify the name and storage path of the JS file.

   ```
   import EAppLogger from 'alife-logger/eapp';
   const Monitor = EAppLogger.init({
     pid: 'xxx',
     region: 'cn', // The region where the application is deployed. Set it to cn if the application is deployed in China, to sg if the application is deployed near Singapore outside China, and to us if the application is deployed near the United States outside China.
   });
   
   export default Monitor;
   ```

   Note:
   For more information about parameter configurations, see Common parameters.

2. Call the following methods to automatically collect the page view (PV), error, API request, performance, and health data.

   a. In app.js, call `Monitor.hookApp(options)` to automatically capture error logs. The `options` parameter is the app-specific object.

   ```
   import Monitor from '/util/monitor';
   
   App(Monitor.hookApp({
     onError(err) {
   ```
b. In page.js, call `Monitor.hookPage(options)` to automatically report the API request, PV, and health data.

```javascript
import Monitor from '/util/monitor';
// After hookPage is called, the lifecycle API automatically starts instrumentation.
Page(Monitor.hookPage({
  data: {},
  onLoad(query) {
  },
  onReady() {
    // Page loaded.
  },
  onShow() {
  },
  onLoad(query) {
  },
  onHide() {
  },
  onUnload() {
  }
})
```

3. Set security domains.

- If the `region` is set to `cn`, add arms-retcode.aliyuncs.com to the HTTP security domain of the request.
- If the `region` is set to `sg`, add arms-retcode-sg.aliyuncs.com to the HTTP security domain of the request.

**Common parameters**

The following table lists the common parameters that are used for initializing the NPM package.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>String</td>
<td>The ID of the site.</td>
<td>Yes</td>
<td>null</td>
</tr>
</tbody>
</table>
Application Real-time Monitoring Service

Browser monitoring / 2 Get started with Browser Monitoring

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>String</td>
<td>The ID of the user, which is used to collect the unique visitor (UV) data.</td>
<td>No</td>
<td>Storage setting</td>
</tr>
<tr>
<td>tag</td>
<td>String</td>
<td>The input tag. Each log carries a tag.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>disabled</td>
<td>Boolean</td>
<td>Specifies whether the log reporting function is disabled.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sample</td>
<td>Integer</td>
<td>The log sampling rate. Valid values: 1, 10, and 100. Performance logs and successful API request logs are reported in a 1/number of samples ratio.</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>enableLinkTrace</td>
<td>Boolean</td>
<td>Specifies whether front-to-back tracing is supported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>disableHook</td>
<td>Boolean</td>
<td>Specifies whether to disable monitoring dd.httpRequest. By default, the request is monitored, and the request success rate is reported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sendRequest</td>
<td>Function</td>
<td>The method for sending logs. If this parameter is not configured, the default value dd.httpRequest is used.</td>
<td>No</td>
<td>dd.httpRequest</td>
</tr>
<tr>
<td>getCurrentPage</td>
<td>Function</td>
<td>The method for obtaining the current page.</td>
<td>No</td>
<td>getCurrent Page</td>
</tr>
</tbody>
</table>

### Basic methods for automatic instrumentation

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>hookApp</td>
<td>{}</td>
<td>Enter the source app parameters.</td>
<td>The app lifecycle API automatically starts instrumentation.</td>
</tr>
<tr>
<td>hookPage</td>
<td>{}</td>
<td>Enter the source page parameters.</td>
<td>The page lifecycle API automatically starts instrumentation.</td>
</tr>
</tbody>
</table>

Note:
If the lifecycle API calls the hookApp or hookPage method for instrumentation in mini program monitoring projects, the projects must conform to the app and page regulations of standard mini programs. In other words, the projects must support **onError** under App, and **onShow**, **onHide**, and **onUnload** under Page. For usage examples, see Procedure.

### Methods for other settings

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCommonInfo</td>
<td><code>{[key: string]: string;}</code></td>
<td>Set basic log fields for the scenarios such as phased release.</td>
</tr>
<tr>
<td>setConfig</td>
<td><code>{[key: string]: string;}</code></td>
<td>Set the config field.</td>
</tr>
<tr>
<td>pageShow</td>
<td>{}</td>
<td>Report the PV data.</td>
</tr>
<tr>
<td>pageHide</td>
<td>{}</td>
<td>Report the health data.</td>
</tr>
<tr>
<td>error</td>
<td>String/Object</td>
<td>Report error logs.</td>
</tr>
<tr>
<td>api</td>
<td>See also api() for reporting the API call success rate</td>
<td>Report the API request logs.</td>
</tr>
<tr>
<td>sum/avg</td>
<td>String</td>
<td>Report the custom sum and average logs.</td>
</tr>
</tbody>
</table>

### Advanced scenarios

When the basic usage cannot meet your needs, see the following advanced scenarios.

- Manually report the API request results (automatic reporting is disabled).

  1. Set `disableHook` to `true`. The logs of the `dd.httpRequest` request are not reported automatically.

  2. Manually call `api()` to report the API request results.

- Disable automatic reporting and enable manual instrumentation.

  1. No longer use the `hookApp` and `hookPage` methods in the app.js and page.js files.

  2. To send the PV data of the current page, call `pageShow()` under `onShow` of Page.

---

**Note:**
3. To send the health data (health and browsing time) of the current page, call `pageHide()` under `onHide` and `onUnload` of Page.

**Note:**
Do not call `pageHide()` together with `hookPage()`. Otherwise, the logs are reported repeatedly.

```javascript
import Monitor from '/util/monitor';
Page({
  onHide: function() {
    Monitor.pageHide();
  },
  onUnload: function() {
    Monitor.pageHide();
  }
});
```

**More information**

- Overview
- Monitor Alipay mini programs
- Monitor WeChat mini programs
- Monitor other mini programs

### 2.4.2 Monitor Alipay mini programs

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor Alipay mini programs. It also demonstrates the general configurations, methods, and advanced scenarios.

**Basic usage**

To monitor the mini programs, you must perform at least the three steps: introducing and initializing the npm (Node Package Manager) package, reporting logs, and setting security domains.
1. Introduce and initialize the npm package.
   
   a. Introduce the npm package named alife-logger in the Alipay mini program project to facilitate log reporting.

   ```javascript
   npm install alife-logger
   ```

   b. Add the following information to the monitor.js file in the /utils directory to initialize the npm package.

   ```javascript
   Note:
   You can specify the name and storage path of the JS file.

   import AlipayLogger from 'alife-logger/alipay';
   const Monitor = AlipayLogger.init({
     pid: 'xxx',
     region: "cn", // The region where the application is deployed. Set it to cn if the application is deployed in China and to sg if the application is deployed outside China.
   });

   export default Monitor;
   ```

   ```
   Note:
   For more information about parameter configurations, see Common parameters.
   ```

2. Call the following methods to automatically collect the page view (PV), error, API request, performance, and health data.

   a. In app.js, call `Monitor.hookApp(options)` to automatically capture error logs. The `options` parameter is the app-specific object.

   ```javascript
   import Monitor from '/util/monitor';

   App(Monitor.hookApp({
     onError(err) {
       console.log('Trigger onError:', err);
     },
     onLaunch() {
       console.log('Trigger onLaunch');
     },
     onHide() {
     },
   }}));
   ```
b. In page.js, call `Monitor.hookPage(options)` to automatically report the API request, PV, and health data.

```javascript
import Monitor from '/util/monitor';
// After hookPage is called, the lifecycle API automatically starts instrumentation.
Page(Monitor.hookPage({
  data: {},
  onLoad(query) {
  },
  onReady() {
    // Page loaded.
  },
  onShow() {
  },
  onLoad(query) {
  },
  onHide() {
  },
  onUnload() {
    }
}))
```

3. Set security domains.

- If the `region` is set to `cn`, add `arms-retcode.aliyuncs.com` to the HTTP security domain of the request.
- If the `region` is set to `sg`, add `arms-retcode-sg.aliyuncs.com` to the HTTP security domain of the request.

**Common parameters**

The following table lists the common parameters that are used for initializing the npm package.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>String</td>
<td>The ID of the site</td>
<td>Yes</td>
<td>null</td>
</tr>
<tr>
<td>uid</td>
<td>String</td>
<td>The ID of the user, which is used to collect the unique visitor (UV) data.</td>
<td>No</td>
<td>Storage setting</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>tag</td>
<td>String</td>
<td>The input tag. Each log carries a tag.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>disabled</td>
<td>Boolean</td>
<td>Specifies whether the log reporting function is disabled.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sample</td>
<td>Integer</td>
<td>The log sampling rate. Valid values: 1, 10, and 100. Performance logs and</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>successful API request logs are reported in a 1/number of samples ratio.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enableLinkTrace</td>
<td>Boolean</td>
<td>Specifies whether front-to-back tracing is supported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>disableHook</td>
<td>Boolean</td>
<td>Specifies whether to disable monitoring my.httpRequest. By default, the</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>request is monitored, and the API request success rate is reported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendRequest</td>
<td>Function</td>
<td>The method for sending logs. If this parameter is not configured, the</td>
<td>No</td>
<td>my.httpRequest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default value my.httpRequest is used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>----------------------------------------------------</td>
<td>----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>getCurrentPage</td>
<td>Function</td>
<td>The method for obtaining the current page.</td>
<td>No</td>
<td>getCurrentPage</td>
</tr>
</tbody>
</table>

**Basic methods for automatic instrumentation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>hookApp</td>
<td>{}</td>
<td>Enter the source app parameters.</td>
<td>The app lifecycle API automatically starts instrumentation.</td>
</tr>
<tr>
<td>hookPage</td>
<td>{}</td>
<td>Enter the source page parameters.</td>
<td>The page lifecycle API automatically starts instrumentation.</td>
</tr>
</tbody>
</table>

**Note:**
If the lifecycle API calls the hookApp or hookPage method for instrumentation in mini program monitoring projects, the projects must conform to the app and page regulations of standard mini programs. In other words, the projects must support `onError` under App, and `onShow`, `onHide`, and `onUnload` under Page. For usage examples, see [Basic usage](#).

**Methods for other settings**

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCommonInfo</td>
<td>{{key: string}: string;}</td>
<td>Set basic log fields for the scenarios such as phased release.</td>
</tr>
<tr>
<td>setConfig</td>
<td>{{key: string}: string;}</td>
<td>Set the config field.</td>
</tr>
<tr>
<td>pageShow</td>
<td>{}</td>
<td>Report the PV data.</td>
</tr>
<tr>
<td>pageHide</td>
<td>{}</td>
<td>Report the health data.</td>
</tr>
<tr>
<td>error</td>
<td>String/Object</td>
<td>Report error logs.</td>
</tr>
<tr>
<td>api</td>
<td>See also <a href="#">api() for reporting the API call success rate</a></td>
<td>Report the API request logs.</td>
</tr>
<tr>
<td>sum/avg</td>
<td>String</td>
<td>Report the custom sum and average logs.</td>
</tr>
</tbody>
</table>
Advanced scenarios

When the basic usage cannot meet your needs, see the following advanced scenarios.

- Manually report the API request results (automatic reporting is disabled).
  1. Set `disableHook` to true. The logs of the `my.httpRequest` request are not reported automatically.
  2. Manually call `api()` to report the API request results.
- Disable automatic reporting and enable manual instrumentation.
  1. No longer use the `hookApp` and `hookPage` methods in the `app.js` and `page.js` files.
  2. To send the PV data of the current page, call `pageShow()` under `onShow` of `Page`.

  **Note:**
  Do not call `pageShow()` together with `hookPage()`. Otherwise, the PV logs are reported repeatedly.

  ```javascript
  import Monitor from '/util/monitor';
  Page({
    onShow: function() {
      Monitor.pageShow();
    }
  })
  ```

  3. To send the health data (health and browsing time) of the current page, call `pageHide()` under `onHide` and `onUnload` of `Page`.

  **Note:**
  Do not call `pageHide()` together with `hookPage()`. Otherwise, the logs are reported repeatedly.

  ```javascript
  import Monitor from '/util/monitor';
  Page({
    onHide: function() {
      Monitor.pageHide();
    },
    onUnload: function() {
      Monitor.pageHide();
    }
  })
  ```

More information

- **Overview of Getting Started with Browser Monitoring**
2.4.3 Monitor WeChat mini programs

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor WeChat mini programs. It also demonstrates the general configurations, methods, and advanced scenarios.

Basic usage

To monitor the mini programs, you need to perform at least the three steps: obtaining and initializing the monitoring SDK for WeChat mini programs, reporting logs, and setting the security domain.

1. Obtain and initialize the monitoring SDK for WeChat mini programs.
   a. Copy and paste the content of this JS file to the wxLogger.js file in the /utils directory of WeChat mini programs.
   b. Add the following information to the monitor.js file in the /utils directory to initialize the SDK.

   ```javascript
   Note:
   You can specify the name and storage path of the JS file.
   
   - If the project is integrated by using the node module (require) method, add the following content to the wxLogger.js file:
     ```javascript
     const WXLogger = require('./wxLogger.js');
     const Monitor = WXLogger.init(
       pid: 'xxx',
       region: 'cn'
     );
     export default Monitor;
     ```
   
   - If the project is integrated by using the ES module (import) method, add the following content to the wxLogger.js file:
     ```javascript
     import WXLogger from './wxLogger.js';
     const Monitor = WXLogger.init(
       pid: 'xxx',
       region: 'cn'
     );
     export default Monitor;
     ```
2. Call the following methods to automatically collect the page view (PV), error, API request, performance, and health data.

   a. In app.js, call `Monitor.hookApp(options)` to automatically capture error logs. The `options` parameter is the app-specific object.

   ```javascript
   import Monitor from '/util/monitor';
   App(Monitor.hookApp({
       onError(err) {
           console.log('Trigger onError:', err);
       },
       onLaunch() {
           console.log('Trigger onLaunch');
       },
       onShow(options) {
       },
       onHide() {
       })
   });
   ```

   b. In page.js, call `Monitor.hookPage(options)` to automatically report the API request, PV, and health data.

   ```javascript
   import Monitor from '/util/monitor';
   // After hookPage is called, the lifecycle API automatically starts instrumentation.
   Page(Monitor.hookPage({
       data: {},
       onLoad(query) {
       },
       onReady() {
           // Page loaded.
       },
       onShow() {
       },
       onLoad(query) {
       },
       onHide() {
       },
       onUnload() {
       }
   }));
   ```
3. Set security domains.

- If the region is set to cn, add https://arms-retcode.aliyuncs.com to the valid domain of the request.
- If the region is set to sg, add https://arms-retcode-sg.aliyuncs.com to the valid domain of the request.

Common parameters

The following table lists the common parameters that are used for initializing the monitoring SDK.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>String</td>
<td>The ID of the site.</td>
<td>Yes</td>
<td>null</td>
</tr>
<tr>
<td>uid</td>
<td>String</td>
<td>The ID of the user, which is used to collect the unique visitor (UV) data.</td>
<td>No</td>
<td>Storage setting</td>
</tr>
<tr>
<td>tag</td>
<td>String</td>
<td>The input tag. Each log carries a tag.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>disabled</td>
<td>Boolean</td>
<td>Specifies whether the log reporting function is disabled.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sample</td>
<td>Integer</td>
<td>The log sampling rate. Valid values: 1, 10, and 100. Performance logs and successful API request logs are reported in a 1/number of samples ratio.</td>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>
### Application Real-time Monitoring Service

#### Browser monitoring / 2 Get started with Browser Monitoring

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableLinkTrace</td>
<td>Boolean</td>
<td>Specifies whether front-to-back tracing is supported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>disableHook</td>
<td>Boolean</td>
<td>Specifies whether to disable monitoring the request. By default, the request is monitored, and the API request success rate is reported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sendRequest</td>
<td>Function</td>
<td>The method for sending logs. If this parameter is not configured, the default value wx.request is used.</td>
<td>No</td>
<td>wx.request</td>
</tr>
<tr>
<td>getCurrentPage</td>
<td>Function</td>
<td>The method for obtaining the current page.</td>
<td>No</td>
<td>getCurrentPage</td>
</tr>
</tbody>
</table>

### Basic methods for automatic instrumentation

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>hookApp</td>
<td>{}</td>
<td>Enter the source app parameters.</td>
<td>The app lifecycle API automatically starts instrumentation.</td>
</tr>
<tr>
<td>hookPage</td>
<td>{}</td>
<td>Enter the source page parameters.</td>
<td>The page lifecycle API automatically starts instrumentation.</td>
</tr>
</tbody>
</table>

### Note:

Issue: 20200615
If the lifecycle API calls the hookApp or hookPage method for instrumentation in mini program monitoring projects, the projects must conform to the app and page regulations of standard mini programs. In other words, the projects must support `onError` under App, and `onShow`, `onHide`, and `onUnload` under Page. For usage examples, see Basic usage.

### Methods for other settings

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCommonInfo</td>
<td><code>{[key: string]: string;}</code></td>
<td>Set basic log fields for the scenarios such as phased release.</td>
</tr>
<tr>
<td>setConfig</td>
<td><code>{[key: string]: string;}</code></td>
<td>Set the config field.</td>
</tr>
<tr>
<td>pageShow</td>
<td>{}</td>
<td>Report the PV data.</td>
</tr>
<tr>
<td>pageHide</td>
<td>{}</td>
<td>Report the health data.</td>
</tr>
<tr>
<td>error</td>
<td>String/Object</td>
<td>Report error logs.</td>
</tr>
<tr>
<td>api</td>
<td>See also api() for reporting the API call success rate</td>
<td>Report the API request logs.</td>
</tr>
<tr>
<td>sum/avg</td>
<td>String</td>
<td>Report the custom sum and average logs.</td>
</tr>
</tbody>
</table>

### Advanced scenarios

When the basic usage cannot meet your needs, see the following advanced scenarios.

- Manually report the API request results (automatic reporting is disabled).
  1. Set `disableHook` to true. The logs of the `wx.request` request are not reported automatically.
  2. Manually call `api()` to report the API request results.
- Disable automatic reporting and enable manual instrumentation.
  1. No longer use the hookApp and hookPage methods in the app.js and page.js files.
  2. To send the PV data of the current page, call `pageShow()` under `onShow` of Page.

**Note:**

Do not call `pageShow()` together with `hookPage()`. Otherwise, the PV logs are reported repeatedly.

```javascript
import Monitor from '/util/monitor';
Page({
onShow: function() {
    Monitor.pageShow();
})
```
3. To send the health data (health and browsing time) of the current page, call `pageHide()` under `onHide` and `onUnload` of Page.

**Note:**
Do not call `pageHide()` together with `hookPage()`. Otherwise, the logs are reported repeatedly.

```javascript
import Monitor from '/util/monitor';
Page({
  onHide: function() {
    Monitor.pageHide();
  },
  onUnload: function() {
    Monitor.pageHide();
  }
})
```

More information

- Overview of Getting Started with Browser Monitoring
- Monitor DingTalk mini programs
- Monitor Alipay mini programs
- Monitor other mini programs

### 2.4.4 Monitor other mini programs

This topic describes how to use the browser monitoring function of Application Real-Time Monitoring Service (ARMS) to monitor the standards-compliant mini programs, except for DingTalk, Alipay, and WeChat mini programs. It also demonstrates the general configurations, methods, and advanced scenarios.

**Basic usage**

To monitor the mini programs, you need to perform at least the three steps: introducing and initialize the npm (Node Package Manager) package, reporting logs, and setting security domains.
1. Introduce and initialize the npm package.

   **a.** In your mini program project, introduce the npm package named `alife-logger` to facilitate log reporting.

   ```shell
   npm install alife-logger
   ```

   **b.** Add the following information to the monitor.js file in the /utils directory to initialize the npm package.

   ```javascript
   import MiniProgramLogger from 'alife-logger/miniprogram';
   const Monitor = MiniProgramLogger.init({
     pid: 'xxx',
     uid: 'userxxx', // The ID of the user, which is used to collect the unique visitor (UV) data.
     region: 'cn', // The region where the application is deployed. Set it to cn if the application is deployed in China and to sg if the application is deployed outside China. The default value is cn.
     // You need to specify the remote procedure call (RPC) method to perform browser monitoring of mini programs. Write the implementation method as needed. The following example takes the method of DingTalk E-App as an example.
     sendRequest: (url, resData) => {
       // This parameter must be configured by the business side. The GET or POST method is supported.
       // demo in dingding
       var method = 'GET';
       var data;
       if (resData) {
         method = 'POST';
         data = JSON.stringify(resData);
       }
       dd.httpRequest({
         url: url,
         method: method,
         data: data,
         fail: function (error) {
           //...
         }
       });
    },
    // Manually enter the method to get the path of the current page. Write the implementation method as needed. The following example takes the method of DingTalk E-App as an example.
    getCurrentPage: () => {
      // This parameter must be configured by the business side.
      if (typeof getCurrentPages !== 'undefined' && typeof getCurrentPages === 'function') {
        var pages = (getCurrentPages() || []);
        var pageLength = pages.length;
        var currPage = pages[pageLength - 1];
        return (currPage && currPage.route) || null;
      }
    });
   ```

   **Note:**
   You can specify the name and storage path of the JS file.
export default Monitor;

Note:
For more information about parameter configurations, see Common parameters.

2. Report logs.

a. In app.js, use either of the following two methods to report logs:

- Use the Monitor.hookApp(options) method to automatically capture error logs.
The options parameter is the app-specific object.

```javascript
import Monitor from '/utils/monitor';

App(Monitor.hookApp({
  onError(err) {
    console.log('Trigger onError:', err);
  },
  onLaunch() {
    console.log('Trigger onLaunch');
  }
  onShow(options) {
  },
  onHide() {
  }
}));
```

- Use the Monitor.error(err) method to manually report error logs.

```javascript
import Monitor from '/utils/monitor';

App({
  onError(err) {
    Monitor.error(err);
    console.log('Trigger onError:', err);
  },
  onLaunch() {
    console.log('Trigger onLaunch');
  }
  onShow(options) {
  },
  onHide() {
  }
}));
```
b. In `page.js`, use either of the following two methods to report logs:

- Use the `Monitor.hookPage(options)` method to automatically report the PV and health data.

  ```javascript
  import Monitor from '/utils/monitor';
  Page(Monitor.hookPage({
    data: {},
    onLoad(query) {
    },
    onReady() {
      // Page loaded.
    },
    onShow() {
    },
    onLoad(query) {
    },
    onHide() {
    },
    onUnload() {
    },
    onTitleClick() {
      /**
        * Collects instrumentation data and performs instrumentation in a custom manner.
        * @desc
        */
      Monitor.sum('titleClick');
    }));
  }
  ```

- Call a method to start instrumentation actively.

  ```javascript
  import Monitor from './util/monitor';
  Page({
    data: {},
    onShow() {
      Monitor.pageShow();
    },
    onHide() {
      Monitor.pageHide();
    },
  });
  ```

**Note:**
Automatic report of API request results is not supported in this method.

**Note:**
For more information about the methods, see [Methods](#).
```javascript
onUnload() {
  Monitor.pageHide();
},
onTitleClick() {
  /**
   * Collects instrumentation data and performs instrumentation in a custom manner.
   * @desc
   */
  Monitor.sum('titleClick');
}
});
```

3. Set security domains.

- If the **region** is set to `cn`, add `https://arms-retcode.aliyuncs.com` to the valid domain of the request.
- If the **region** is set to `sg`, add `https://arms-retcode-sg.aliyuncs.com` to the valid domain of the request.

**Common parameters**

The following table lists the common parameters that are used for initializing the NPM package.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>String</td>
<td>The ID of the site.</td>
<td>Yes</td>
<td>null</td>
</tr>
<tr>
<td>uid</td>
<td>String</td>
<td>The ID of the user, which is used to collect the UV data.</td>
<td>No</td>
<td>Storage setting</td>
</tr>
<tr>
<td>tag</td>
<td>String</td>
<td>The input tag. Each log carries a tag.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>disabled</td>
<td>Boolean</td>
<td>Specifies whether the log reporting function is disabled.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>sample</td>
<td>Integer</td>
<td>The log sampling rate. Valid values: 1, 10, and 100. Performance logs and successful API request logs are reported in a 1/number of samples ratio.</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>disableHook</td>
<td>Boolean</td>
<td>Specifies whether to disable monitoring my.httpRequest. By default, the request is monitored, and the API request success rate is reported.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sendRequest</td>
<td>Function</td>
<td>The method for sending logs. If this parameter is not configured, the logs cannot be sent.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>getCurrentPage</td>
<td>Function</td>
<td>The method for obtaining the current page.</td>
<td>Yes</td>
<td>None</td>
</tr>
</tbody>
</table>

**Fields in sendRequest**

The sendRequest parameter is used to send logs and must support the GET or POST method. The POST method is used to report error logs. The method parameters are described as follows.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>String</td>
<td>The URL to which the log is reported.</td>
</tr>
</tbody>
</table>
### Application Real-time Monitoring Service

#### Browser monitoring / 2 Get started with Browser Monitoring

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resData</td>
<td>Object</td>
<td>The content that you want to report in the POST method. When a value is set for this parameter, the log must be reported in the POST method. Otherwise, the log must be reported in the GET method.</td>
</tr>
</tbody>
</table>

#### sendRequest configuration example

```javascript
sendRequest: (url, resData) => {
  // This parameter must be configured by the business side. The GET or POST method is supported.
  // demo in dingding
  var method = 'GET';
  var data;
  if (resData) {
    method = 'POST';
    data = JSON.stringify(resData);
  }
  dd.httpRequest({
    url: url,
    method: method,
    data: data,
    fail: function (error) {
      //...
    }
  });
}
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>hookApp</td>
<td>{}</td>
<td>Enter the source app parameters. The app lifecycle API automatically starts instrumentation.</td>
</tr>
<tr>
<td>hookPage</td>
<td>{}</td>
<td>Enter the source page parameters. The page lifecycle API automatically starts instrumentation.</td>
</tr>
<tr>
<td>setCommonInfo</td>
<td>{{key: string}: string;}</td>
<td>Set basic log fields for the scenarios such as phased release.</td>
</tr>
<tr>
<td>setConfig</td>
<td>{{key: string}: string;}</td>
<td>Set the config field.</td>
</tr>
<tr>
<td>pageShow</td>
<td>{}</td>
<td>Report the PV data.</td>
</tr>
<tr>
<td>Method</td>
<td>Parameter</td>
<td>Remarks</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>pageHide</td>
<td>{}</td>
<td>Report the health data.</td>
</tr>
<tr>
<td>error</td>
<td>String/Object</td>
<td>Report error logs.</td>
</tr>
<tr>
<td>api</td>
<td>See also api() for reporting the API call success rate</td>
<td>Report the API request logs.</td>
</tr>
<tr>
<td>sum/avg</td>
<td>String</td>
<td>Report the custom sum and average logs.</td>
</tr>
</tbody>
</table>

**Note:**

If the lifecycle API calls the hookApp or hookPage method for instrumentation in monitoring mini program projects, the projects must conform to the app and page regulations of standard mini programs. In other words, the projects must support `onError` under App, and `onShow`, `onHide`, and `onUnload` under Page. For usage examples, see `Basic usage`.

Most log report methods achieve the same purposes as the browser monitoring SDKs. The following section describes how to call other methods:

- To send the PV data of the current page, call `pageShow()` under `onShow` of Page.

  **Note:**

  Do not call `pageShow()` together with `hookPage()`. Otherwise, the PV logs are reported repeatedly.

```javascript
import Monitor from './util/monitor';
Page({
  onShow: function() {
    Monitor.pageShow();
  }
});
```

- To send the health data (health and browsing time) of the current page, call `pageHide()` under `onHide` and `onUnload` of Page.

  **Note:**

  Do not call `pageHide()` together with `hookPage()`. Otherwise, the logs are reported repeatedly.

```javascript
import Monitor from './util/monitor';
Page({
  onHide: function() {
```
Advanced scenarios

When the basic usage cannot meet your needs, see the following advanced scenarios.

- **Set the uid**, which is used to collect the UV data.
  - If you can obtain the user information before initializing the monitoring SDK, you can directly set the uid.
  - Otherwise, you can obtain the user information before onShow is triggered for the application and then call `setCommonInfo({uid: 'xxx'})` to set the uid.
- **Set common information for mini programs.**

Call `setCommonInfo` to set common information for mini programs. ARMS browser monitoring performs statistical analysis of the following fields:

- sr: the size of the screen.
- vp: the visible section in the browser window.
- dpr: the pixel ratio of the screen.
- ul: the language of the document.
- dr: the reference of the document.
- ct: the network connection type, for example, Wi-Fi or 3G.

**Warning:**
Do not call `setCommonInfo` to set too many fields at a time. Otherwise, the request length may exceed the constraints, causing request failures.

More information

- **Overview of Getting Started with Browser Monitoring**
- **Monitor DingTalk mini programs**
- **Monitor Alipay mini programs**
- **Monitor WeChat mini programs**
3 Console functions

3.1 Page speed

This topic describes the functions on the Page Speed page of browser monitoring of Application Real-Time Monitoring Service (ARMS).

After connecting your application to ARMS browser monitoring, you can view the following performance data of your application on the Page Speed page:

- Page loading details
- Page loading waterfall chart
- Performance distribution
- Slow page session tracing
- Page loading distribution

In the Page Speed Ranking section, you can rank the pages by first paint time or PV and click the upward or downward arrow to change the display order.

Page loading details

This module displays the following metrics.
• Key performance metrics
  - First Paint Time
  - First Meaningful Paint
  - Time to First Interaction
  - DOM Ready
  - Fully Loaded Time

• Page load time by intervals:
  - DNS Lookup
  - TCP Connection Time
  - Time to First Byte (TTFB)
  - Content Download
  - DOM Parsing
  - Resource Download

For more information about how these metrics are calculated, see Statistical metrics.

[Line chart showing page load time details]

**Note:**

• The data in the line chart is displayed based on the average value of the metric within the specified time range. The average value reflects the average performance over a period of time, but is vulnerable to extreme values. For example, the overall page
loading is very slow due to poor network connection during an access, and therefore the average response time is high. You can click the Filter icon in the upper-right corner to remove extreme values. In this way, the impact of extreme values on the overall performance trend is removed.

- If the data in the line chart increases sharply, you can use the performance sample distribution and slow page session trace functions to locate the problem.

Page loading waterfall chart

The waterfall chart visually shows the response time in each stage based on the page loading order. The data in the figure is the average value of a specified metric within a specified time range. To optimize performance, implement targeted measures at specific stages.

Performance distribution

This module visually shows the page performance distribution.

On the Performance Stacked Area Chart tab, a stack line chart with time as the horizontal axis appears. You can view the distribution of performance at different time points.
On the **Performance Sample Distribution** tab, you can view the sample proportion of the page loading time in the specified time range. For example, how many pages can be opened within 1 second, or the proportion of samples of users with long tails.

**Figure 3-1: Performance sample distribution**

![Performance Sample Distribution](image)

**Slow page session tracing**

The slow session tracing function provides a performance waterfall chart for static resource loading during page loading. This helps you learn more about page resource loading status based on page performance data, and quickly locate performance bottlenecks. For more information, see [#unique_21](#).
Page loading distribution

A page is loaded on the browser of the user. The loading time is related to factors such as the geographical location, network condition, browser, or carrier. Therefore, we provide
statistics by factors such as geographical distribution, terminal distribution, network distribution, and version distribution to help you better locate performance bottlenecks.

**Figure 3-2: Geographical view**

![Geographical View](image1)

**Figure 3-3: Terminal view**

![Terminal View](image2)
3.2 Browser monitoring dashboard

The dashboard provided by browser monitoring of Application Real-Time Monitoring Service (ARMS) allows you to view all the key real-time monitoring data of the monitored application at one time.

**Portal**

1. Log on to the ARMS console and choose **Browser Monitoring** from the left-side navigation pane.
2. On the **Browser Monitoring** page, click the name of your application.
3. On the **Overview** page, click **Dashboard** on the top.
4. On the Dashboard page, you can view the browser monitoring data of your application, including the JavaScript (JS) error rate and API request success rate. The data is updated in real time.

**Description**

The ARMS browser monitoring dashboard displays all key real-time monitoring data of the monitored application that is suitable for display in a dashboard.

**Note:**

The monitoring data is updated at least once every minute.

**Available actions**

- View the monitoring data on the dashboard.
- Move the pointer over a curve in the statistical chart. The statistics at the time point appears.
- Click **Full Screen** in the upper-right corner to display the dashboard in full screen.

**Field description**

The following describes the fields in the ARMS browser monitoring dashboard.

- **JS error rate**
  - JS Error Rate: the JS error rate.
  - Compared with Yesterday's Average: the increase or decrease ratio of JS errors compared to the average JS error rate of the previous day
  - Statistical Chart: the JS error rate curve of the last hour.
  - High Error Rate Top 5: the top five services with the highest JS error rates.
• Today’s alerts
  - Alarms: the number of alerts.
  - Recent Alarms: alerts from browser monitoring on the current day.
• PV and UV
  - PV Today: the number of page views (PVs) of the monitored application on the current day.
  - UV Today: the number of unique visitors (UVs) of the monitored application on the current day.
  - Compared with Yesterday: the increase or decrease ratio of PVs compared with that of PVs in the previous day.
  - Statistical Chart: the PV or UV curve of the last hour.
  - Statistical Table: the top five regions with the highest PV and UV and their corresponding PV and UV.
  - High Page View Top 5: the top five services with the most page views.
• API request success rate
  - API Request Success Rate: the proportion of successful API requests.
  - Compared with Yesterday’s Average: the increase or decrease ratio of API request success rate compared to that of the previous day.
  - Statistical Chart: the API request success rate curve of the previous hour.
  - Low Success Rate Top 5: the top five services with the lowest API request success rates.
• Page speed
  - Page Speed: the First Paint Time (FPT), in milliseconds.
  - Compared with Yesterday’s Average: the increase or decrease ratio of page speed compared with that of the previous day.
  - Statistical Chart: the page speed curve of the last hour.
  - Low Page Speed Top 5: the top five services with the lowest page speeds.

More information
• Page speed
• #unique_23
• API request monitoring
• #unique_25
3.4 API request monitoring

This topic introduces the API request monitoring function provided by browser monitoring of Alibaba Cloud Application Real-Time Monitoring Service (ARMS).

Description

The API request monitoring function of ARMS browser monitoring clearly displays the following information:

- Success rate of each API
- API response information
- Average response time of successful API calls
- Average response time of failed API calls

In addition, this function displays the distribution of the preceding statistics in the following dimensions:

- Location
- Browser
- OS
- Device
- Resolution

API call success rate

The Success Rate tab on the left ranks the API success rates. The chart on the right shows the call volume and success rate curves of the API selected on the left within a specified time range.
API response information

The Message tab on the left displays the API response information. The Message Request Details section on the right displays the API calls of the API response information selected on the left, which are listed in descending order by call volume.

Response time for successful API calls

The Success RT tab on the left displays the average response time of successful API calls. The Success API Response Time section on the right shows the average response time of successful API calls selected on the left by curves and the numbers of successful calls in a bar chart.

Response time for failed API calls

The Failure RT tab on the left displays the average response time of failed API calls. The Failure API Response Time section on the right shows the average response time of failed API calls selected on the left by curves and the numbers of failed calls in a bar chart.
Geographical view

On the Geographical View page, you can view the statistics of each geographical location. The Geographical View page consists of the China View and World View tabs. China View displays statistics by province in China, while World View displays statistics by country or region worldwide.

Terminal view

On the Terminal View page, you can view the statistics of each terminal. The Terminal View page consists of the Browser View, OS View, Device View, and Resolution View tabs.
Common operations

In the JS error statistics module, you can perform the following common operations.

- On the left-side tab, click the upward or downward arrow to change the order of the list. The upward arrow indicates the ascending order and the downward arrow indicates the descending order.

- In the right-side details section, click the list icon in the upper-right corner to switch between charts and table views.
3.5 Custom statistics

This topic introduces the custom statistics function provided by browser monitoring of Application Real-Time Monitoring Service (ARMS).

To help you monitor and measure lightweight business interaction behavior, browser monitoring provides the following two types of custom statistics:

- **Sum statistics**: counts the total number of occurrences of certain events in the business, such as the number of times that a button is clicked and the number of times that a module is loaded.
- **Average statistics**: counts the average value of certain events in the business, such as the average loading time of a module.

ARMS provides the preceding two types of statistics in the following three dimensions (taking average statistics for example):

- **Statistics details**

  The statistics details line chart shows the trends of Average and Sample Size of an event within a specified period of time. Assume that the time consumption data of a module is
collected. In the statistics details, you can see the average time consumption data and the number of sent samples in the corresponding time interval.

![Statistics Details](image)

- **Geographical view**

  Statistics on the event in corresponding regions are collected by province or city in China or by country or region worldwide. ARMS browser monitoring provides the reported amount, average value, and unique visitor (UV) data of each region. It helps the business side quickly understand the differences between events in different regions and make decisions.

- **Terminal view**

  Browsers, devices, operating systems, and resolutions affect the performance, compatibility, and display of a page. Therefore, ARMS browser monitoring provides the average values and sample sizes in these dimensions, helping the business side know the distribution of the event in different browsers, devices, operating systems, and resolutions.

**Method for sum statistics**

After the browser monitoring SDK is introduced on the page, use the following log reporting method in the business JavaScript (JS) file for sum statistics.

Call parameters: `__bl.sum(key, value)`

Call parameter description:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The name of the event.</td>
<td>Yes</td>
<td>None</td>
</tr>
</tbody>
</table>
### Method for average value statistics

After the browser monitoring SDK is introduced on the page, use the following log reporting method in the business JavaScript file for average value statistics.

Call parameter: `__bl.avg(key, value)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The name of the event.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>value</td>
<td>Number</td>
<td>The number of reported items. Default value: 0.</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:

```javascript
__bl.avg('event-a', 1);
__bl.avg('event-b', 3);
```
4 Tutorials

4.2 Diagnose slow loading of pages

It is much difficult to locate the problem of slow page loading. To solve this problem, the slow session tracing function in browser monitoring of Application Real-Time Monitoring Service (ARMS) provides a performance waterfall chart for static resource loading on pages, which allows you to check the resource loading status on the pages and identify the root cause of the problem.

Problem description

Slow page loading is one of the common concerns for users. You may encounter the following difficulties when locating and solving such problems:

• Difficult to recur
  
  For example, when user A accesses a page, the page is loaded on user A’s local browser. As the page loading duration is affected by factors such as the region, network, browser, or carrier, the actual situation when user A accesses the page cannot recur.

• Lacking monitoring information for deep investigation
  
  Most browser monitoring services use the `PerformanceTiming` object to obtain the full page loading time, which does not include the static resource loading information. As a result, the performance bottleneck cannot be located.

Solution

You only need to connect the web application to ARMS browser monitoring, enable the page resource reporting function, and use the slow session tracing function of ARMS browser monitoring to quickly locate the performance bottleneck.

Step 1: Connect to ARMS browser monitoring

The ARMS browser monitoring SDK does not report static resource loading information by default. However, the static resource loading information is required for quickly locating the slow page loading problem with the slow session tracing function.

In this case, connect your web application to ARMS browser monitoring. For more information, see Install the ARMS agent through CDN.
Step 2: Locate the fault

You can locate the fault by using either of the following two methods.

Method 1: Check the access speed

1. Log on to the ARMS console. In the left-side navigation pane, choose Browser Monitoring.
2. On the Browser Monitoring page, click the name of your application.
3. In the left-side navigation pane, choose Page Speed.
   
   For more information on the Page Speed page, see Page speed. In this example, the page performance is poor. The full loading time of the page reaches 36.7s at about 11:00.
4. On the Page Speed page, drag the right-side scroll bar to display the Slow Page Session Trace (TOP20) section. This section lists top 20 sessions with the lowest page loading speed within the specified time range.
   
   In this example, the page loading time at 11:36:46 is 36.72s, which is the cause of the sharp increase of the page loading time.
5. In the Slow Page Session Trace (TOP20) section, click the target page name in the Page column to access the Session Details page. Then, determine the cause and troubleshoot the fault according to the information on the Session Details page.
   
   On the top of the Session Details page, the Page Information section displays the access information such as the client IP address, browser, and operating system, helping you further identify the cause of the fault.
   
   On the Session Details page, the Page Resource Loading Waterfall section displays the waterfall chart of static resource loading on the page, helping you quickly locate the performance bottleneck.
   
   For more information on the Session Traces page, see #unique_21.

Method 2: Trace the session

1. Log on to the ARMS console. In the left-side navigation pane, choose Browser Monitoring.
2. On the **Browser Monitoring** page, click the name of your application.

3. In the left-side navigation pane, choose **Session Traces** to access the **Session List** page.

   The **Session List** page displays the top 100 sessions of the application by loading duration. You can filter sessions by page, session ID, browser, and browser version to quickly identify time-consuming sessions.

4. On the **Session List** page, find the target session and click **Details** in the **Actions** column. Then, determine the cause and troubleshoot the fault according to the information on the **Session Details** page.

   For more information on the **Session Traces** page, see #unique_21.

Up to now, you have completed troubleshooting with the slow session tracing function. This function can help you recur the page resource loading status and quickly locate the performance bottleneck.

**More operations**

To avoid passive diagnosis after an exception occurs, you can also use the alert function of ARMS to create an alert for one or all APIs. This ensures that the O&M team immediately gets a notification after an exception occurs.

To create an alert, see #unique_25.

**More information**

- Page speed
- #unique_21
- #unique_25

### 4.4 Front-to-back tracing

In the browser monitoring module of Alibaba Cloud Application Real-Time Monitoring Service (ARMS), you are unable to accurately know about the network transmission performance and the back-end service traces and performance even if you know the API response time. Therefore, you cannot quickly and accurately troubleshoot application API problems. The front-to-back tracing function can solve such problems. It connects the front end and back end of an API request to make the complete code execution recur.

To obtain the complete front-to-back trace, you must activate ARMS browser monitoring and application monitoring and use them together.
• ARMS browser monitoring
  - **SDK configuration**
    - Enable automatic API reporting.
    - Enable front-to-back tracing, that is, set `enableLinkTrace` to true. The configuration is as follows:
      ```xml
      <script>
        !(function(c,b,d,a){c[a]||(c[a]={});c[a].config={pid:"xxx",imgUrl:"https://arms-retcode.aliyuncs.com/r.png?", enableLinkTrace: true};
        with(b)with(body)with(insertBefore(createElement("script"),firstChild))setAttribu
te("crossorigin","",src=d)
        })(window,document,"https://retcode.alicdn.com/retcode/bl.js","__bl");
      </script>
      ```
  - **Principle**
    - When automatic API reporting is enabled, if the API origin is the same as that of the domain of the current application, the EagleEye-TraceID and EagleEye-SessionID custom headers are added to the API request headers. EagleEye-TraceID is the identifier of the trace that connects the front end and back end.
    - If the API origin is different from that of the domain of the current application, no custom header is added to the API request header to ensure that the application request is sent normally.
    - To check whether the front-to-back tracing configuration takes effect, access the console and check whether the EagleEye-TraceID and EagleEye-SessionID headers are included in the API request header, as shown in the following figure.

    ![Notice:](image)
    *The values of EagleEye-TraceID and EagleEye-SessionID have corresponding meanings. Do not set the values manually.*

• ARMS application monitoring
  - For more application monitoring configurations, see #unique_30.
  - This feature is only supported by ARMS agent 2.4.5 or later.

**Use method**

In the ARMS browser monitoring module, the Overview, Page, and API Request pages contain the API Link Trace section.
You can perform the following operations in the section:

- Sort the APIs in ascending or descending order by the report time or response time.
- Click **Link Trace**. On the tab that appears, view the overall response time of the browser monitoring and the call sequence diagram of the back-end application.
- Click **View Details**. On the details page that appears, view details of the API request.

**Scenarios and cases**

Application monitoring provides the back-end processing performance and trace of an API request. However, the data may not accurately reflect real user experience. Browser monitoring can only monitor the overall response and statuses of an API from request to response but cannot provide the traces and performance of back-end services. In this case, the front-to-back tracing function connects the front end and back end, providing you with one-stop troubleshooting experience.

You can know whether the network transmission or back-end call takes a longer time based on the call timeline. You can also click thread profiling in the back-end application to view the complete back-end trace of this request. In this way, you can identify the cause of the error based on the business.
• Locate the error when the API returns an error code or a business logic error occurs.

1. In API Failure List of the API Link Trace section, locate the API or trace ID, and click **Link Trace** to view the overall response time of browser monitoring and the call sequence diagram of the back-end application.

2. Determine whether the network transmission or back-end call process takes a longer time based on the call timeline.

3. Click the magnifier icon in the **Method Stack** column of the back-end application to view the overall back-end trace of the request and locate the cause of the API error based on the business.

• Locate the error when the API response time is long

1. In the API Link Trace section, sort the APIs in descending order by response time and find the time-consuming API or trace ID.

2. In the row of a time-consuming API, click **Link Trace** to view the overall response time of the browser monitoring and the call sequence diagram of the back-end application.

   - If the back-end application processing time is short but the overall response time is long, the time consumed from API request sending to the server to data returning
to the browser is long. Click **View Details**. On the details page that appears, view the network, region, browser, device, and operating system of the access.

- If the back-end application takes a long time to process, the back-end processing performance is poor. Click the magnifier icon in the **Method Stack** column. In the dialog box that appears, check which part of back-end tracing is time-consuming to locate the problem.
5 Advanced options

5.1 Advanced browser monitoring scenarios

The browser monitoring SDK of Application Real-Time Monitoring Service (ARMS) provides several advanced options for browser monitoring in advanced scenarios, such as SPA page reporting, data pre-reporting, and custom performance metric reporting.

SPA page reporting

In a single page application (SPA), a page will only be refreshed once. Traditionally, the page view (PV) data is only reported once after the page is loaded. However, PVs of sub-pages cannot be collected, and logs of other types cannot be aggregated according to sub-pages.

The ARMS browser monitoring SDK provides two methods for processing SPA pages.

- Enable automatic SPA resolution

  This method is applicable to most SPAs that use URL hash.

  In initial configuration items, setting `enableSPA` to true can enable hashchange event listening (triggering repeated PV reporting) on the page and use URL hash as the `page` field for reporting other data.

  The `<Function>parseHash` parameter can also be used together with `enableSPA` (for more information, see Configuration items of the browser monitoring SDK).

- Manually report data

  This method is applicable to all SPA scenarios. Use this method if the first method is ineffective.

  The SDK provides the `setPage` method for manually updating `page name` during data reporting. When this method is called, page PV will be reported again by default.

  ```javascript
  // Listen to application route change event
  app.on('routeChange', function (next) {
    __bl.setPage(next.name);
  });
  ```

Pre-report data

Data reporting errors may occur in the following situations:
• Some data needs to be reported when a page is being loaded but the SDK initialization is not complete yet (or it is not clear if the SDK initialization is complete).

• The `setConfig` method is called in the application initialization logic. However, the loading may not be complete yet because the SDK is asynchronously loaded.

Solution: The SDK adds the `pipe` attribute to the `__bl` object to cache pre-called information into this variable. The following shows an example.

```javascript
__bl.pipe = [  // Report HTML of the current page as an API.  ['api', '/index.html', true, performance.now, 'SUCCESS'],  // After SDK initialization is complete, enable automatic SPA resolution.  ['setConfig', {enableSPA: true}]
];
```

To report a single data record, use:

```javascript
__bl.pipe = ['msg', 'I\'m another generic message'];
```

The zeroth number in the array is the method name, followed by input parameters. After the SDK initialization is complete, it calls methods and parameters attached to `window.__bl.pipe` one by one.

**Note:**

Before the SDK initialization is complete, if the value of `__bl.pipe` is set multiple times, the last value takes effect.

If you are not sure whether the SDK initialization is complete and do not want to add too much judgment logic, you can call `pipe` after SDK initialization is complete (this is supported by Internet Explorer 9 and later).

For example, in an SPA, after `autoSend: false` is set, PV is reported for the first time after application initialization. However, it is not clear if the SDK initialization is complete.

```javascript
// Set the name of the page to 'homepage' and report PV.
__bl.pipe = ['setPage', 'homepage'];
```

**Report custom performance metrics**

You can use the `__bl.pipe` method to report the following custom performance metrics:

- Custom First Meaningful Paint `cfpt`
- Custom First Time to Interact `ctti`
- Other custom performance metrics `t1 - t10` (10 metrics in total)
The preceding custom performance metrics are reported simultaneously before the page load event is triggered.

For CDN-based access, the custom performance metrics to be reported are as follows:

```javascript
__bl.pipe=['performance',
  {
    cfpt: 100,
    ctti: 20,
    t1: 300,
    t2: 300,
    t3: 60,
    t4: 1300,
    t5: 300,
    t6: 1300,
    t7: 40,
    t8: 300,
    t9: 300,
    t10: 4100
  }]
```

For Weex access, the custom performance metrics to be reported are as follows:

```javascript
const BrowserLogger = require('alife-logger');
// Same as the pipe structure of CDN.
const pipe = ['performance', {cfpt:100, ctti:20, t1:300, t2:300, t3:60, t4:1300, t5:300, t6:1300, t7:40, t8:300, t9:300, t10:4100}];
const __bl = BrowserLogger.singleton({pid:'Unique site ID'},pipe);
```
6 Configuration items of the browser monitoring SDK

Application Real-Time Monitoring Service (ARMS) provides a series of configuration items for the browser monitoring SDK, meeting your additional requirements, for example, ignoring the reported errors related to the specified URLs, APIs, and JavaScript (JS) scripts, filtering non-key characters in URLs for page clustering, and reducing the number of reported items and load by random sampling.

Use configuration items of the browser monitoring SDK

You can use the configuration items of the browser monitoring SDK in either of the following ways:

- When installing an ARMS browser monitoring agent in a page, add extra parameters to `config`.

  For example, in the following sample code, in addition to the default `pid` parameter, the `enableSPA` parameter for single page application (SPA) is added to `config`.

  ```html
  <script>
  !( function(c,b,d,a){c[a]||(c[a]={});c[a].config={pid:"xxxxxx",enableSPA:true};
  with(b)with(body)with(insertBefore(createElement("script"),firstChild))setAttribute("crossorigin","".,src=d)
  })(window,document,https://retcode.alicdn.com/retcode/bl.js","__bl");
  </script>
  ```

- After page initialization is complete, call the `SetConfig` method in JS code to modify the configuration item. For the modification method, see Methods user guide.

SDK configuration parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid</code></td>
<td>String</td>
<td>The unique ID of the project, which is automatically generated when ARMS creates the site.</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>String</td>
<td>The input tag. Each log carries a tag.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>page</td>
<td>String</td>
<td>The page name.</td>
<td>No</td>
<td>The key part of the current page’s URL is used by default: host + pathname.</td>
</tr>
<tr>
<td>enableSPA</td>
<td>Boolean</td>
<td>Specifies whether to monitor the hashchange event on the page and report page view (PV) again. It is applicable to SPA scenarios.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>parseHash</td>
<td>Function</td>
<td>This parameter is used together with enableSPA. For more information, see parseHash: resolves URL hash into a page.</td>
<td>No</td>
<td>For more information, see parseHash: resolves URL hash into a page.</td>
</tr>
<tr>
<td>disableHook</td>
<td>Boolean</td>
<td>Specifies whether to disable the monitoring of AJAX requests.</td>
<td>No</td>
<td>false: Requests are listened to and used for reporting API call success rate.</td>
</tr>
<tr>
<td>ignoreUrlCase</td>
<td>Boolean</td>
<td>Specifies whether the page URL is case insensitive.</td>
<td>No</td>
<td>true: Case insensitive by default.</td>
</tr>
<tr>
<td>urlHelper</td>
<td>*</td>
<td>A replacement of the old parameter ignoreUrlPath to configure URL filtering rules. For more information, see UrlHelper: performs URL filtering.</td>
<td>No</td>
<td>For more information, see UrlHelper: performs URL filtering.</td>
</tr>
<tr>
<td>apiHelper</td>
<td>*</td>
<td>A replacement of the old parameter ignoreApiPath to configure API filtering rules. For more information, see apiHelper: performs API filtering.</td>
<td>No</td>
<td>For more information, see apiHelper: performs API filtering.</td>
</tr>
<tr>
<td>parseResponse</td>
<td>Function</td>
<td>Parses the data returned when the API data is automatically reported. For more information, see config.parseResponse: parses responses.</td>
<td>No</td>
<td>For more information, see config.parseResponse: parses responses.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ignore</td>
<td>Object</td>
<td>Ignores the errors of the specified URL, API, or JS error. Logs that match the rules are ignored and not reported, including sub-items <code>ignoreUrls</code>, <code>ignoreApis</code>, and <code>ignoreErrors</code>. For more information, see ignore: ignores specified URL, API, and JS errors.</td>
<td>No</td>
<td>For more information, see ignore: ignores specified URL, API, and JS errors.</td>
</tr>
<tr>
<td>disabled</td>
<td>Boolean</td>
<td>Specifies whether to disable the log reporting function.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>sample</td>
<td>Integer</td>
<td>The log sampling rate. Valid values: 1, 10, and 100. Performance logs and successful API request logs are sampled in the ratio of 1/sample. For more information, see config.sample: configures sample reporting of logs.</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>sendResource</td>
<td>Boolean</td>
<td>Specifies whether to report static resources on the page. For more information, see sendResource: reports static resources.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>useFmp</td>
<td>Boolean</td>
<td>Specifies whether to collect the First Meaningful Paint (FMP) data.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>enableLinkTrace</td>
<td>Boolean</td>
<td>Specifies whether to enable front-to-back tracing. For more information, see Front-to-back tracing.</td>
<td>No</td>
<td>false</td>
</tr>
<tr>
<td>release</td>
<td>String</td>
<td>The version of the application.</td>
<td>No</td>
<td>undefined</td>
</tr>
<tr>
<td>environment</td>
<td>String</td>
<td>The environment for front-to-back tracing analysis.</td>
<td>No</td>
<td>production</td>
</tr>
<tr>
<td>behavior</td>
<td>Boolean</td>
<td>Performs user behavior backtracking when an error occurs.</td>
<td>No</td>
<td>true for the browser and false for the applet by default.</td>
</tr>
</tbody>
</table>
### Application Real-time Monitoring Service

#### Browser monitoring / 6 Configuration items of the browser monitoring SDK

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>String</td>
<td>Custom service field. Each log carries the field.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>c2</td>
<td>String</td>
<td>Custom service field. Each log carries the field.</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>c3</td>
<td>String</td>
<td>Custom service field. Each log carries the field.</td>
<td>No</td>
<td>None</td>
</tr>
</tbody>
</table>

**parseHash: resolves URL hash into a page**

In the SPA scenario (see Advanced browser monitoring scenarios), when `enableSPA` is set to true and the page triggers a hashchange event, the `parseHash` parameter is used to resolve the URL hash into **Page**.

**Default value**

The default value is obtained by a simple string processing method:

```javascript
function (hash) {
  var page = hash ? hash.replace(/\#/, '').replace(/\?.*$/, '') : '';
  return page || '[index]';
}
```

Normally this item does not need to be modified. You need to modify this item if a custom page name is required during reporting or if the URL hash is complex. The following shows an example.

```javascript
// Define the mapping between Hash and Page.
var PAGE_MAP = {
  '/': 'Homepage',
  '/contact': 'Contact us',
  '/list': 'Data list',
  // ...
};
// Call the SDK method after the page is loaded.
window.addEventListener('load', function (e) {
  // Call the setConfig method to modify SDK configuration items.
  __bl.setConfig({
    parseHash: function (hash) {
      key = hash.replace(/\?.*$/, '')
      return PAGE_MAP[key] || 'Unknown page';
    }
  });
});
```

**UrlHelper: performs URL filtering**

When a page URL is similar to `http://xxx.com/projects/123456` (**projects** is followed by the project ID) and `xxx.com/projects/123456` is reported as a page, the page cannot be
clustered during data browsing. To implement page clustering, set the `urlHelper` parameter to filter non-key characters, for example, the project ID in this example.

**Notice:**
The old parameter `ignoreUrlPath` that configures URL filtering has been replaced by the `urlHelper` parameter. If you use the old parameter, the configuration will still take effect. If both the old and new parameters are used, the new parameter takes effect.

**Note:**
This configuration item is valid only when the page URL is automatically obtained and used as Page. This configuration item will be invalid if you have modified Page by manually calling the `setPage` or `setConfig` method (see Methods user guide) or `enableSPA` has been set to true.

**Default value**
The array is the default value, which does not need to be modified.

```javascript
[ // Replace all numbers in the URL with asterisks (*).
  {rule: /\d+/g, target: '/\$1**'},
  // Remove the trailing slash (/) of the URL.
  '/\$/'
]
```

The default value of this configuration item filters out the numbers in strings such as `xxxx/123456`. For example, `xxxx/00001` and `xxxx/00002` change to `xxxx/**`.

**Value type**
The value of `urlHelper` can be any of the following types:

- **String or RegExp (regular expression):** removes matched strings.
- **Object<rule, target>:** The object contains two keys (rule and target, which are the input parameters of the replace method of the JS string. For more information, see JS related tutorials String::replace method.
- **Function:** uses the original string as the input parameter for method execution, and the execution result as Page.
- **Array:** sets multiple rules. Each rule can be of the preceding types.
apiHelper: performs API filtering

This configuration item filters out non-key characters in API URLs when API results are automatically reported. The usage and function are the same as those of UrlHelper: performs URL filtering.

⚠️ Notice:
The old parameter `ignoreApiPath` that configures API filtering rules has been replaced by the `apiHelper` parameter. If you use the old parameter, the configuration will still take effect. If both the old and new parameters are used, the new parameter takes effect.

Default value

The default value is an object, which does not need to be modified.

```
{rule: /(^w+)\d{2,}/g, target: '$1'}
```

The default value of this configuration item filters out the numbers in strings such as `xxxx/123456` in the API URL.

config.parseResponse: parses responses

This configuration item parses the data returned when the API results are automatically reported.

Default value

The default value of this configuration item is:

```javascript
function (res) {
    if (! res || typeof res !== 'object') return {};
    var code = res.code;
    var msg = res.msg || res.message || res.subMsg || res.errorMsg || res.ret || res.errorResponse || '';
    if (typeof msg === 'object') {
        code = code || msg.code;
        msg = msg.msg || msg.message || msg.info || msg.ret || JSON.stringify(msg);
    }
    return {msg: msg, code: code, success: true};
}
```

The preceding code parses the returned data and tries to extract `msg` and `code`. For most applications, this configuration item does not need to be modified. If the default value cannot meet service requirements, you can reset it.
ignore: ignores specified URL, API, and JS errors

The value of **ignore** is an object, which contains three attributes: **ignoreUrls**, **ignoreApis**, and **ignoreErrors**. You can set one or more attributes.

**Default value**

The default value of this configuration item is:

```json
ignore: {
  ignoreUrls: [],
  ignoreApis: [],
  ignoreErrors: []
},
```

**ignoreUrls**

The **ignoreUrls** attribute is used to ignore certain URLs. Logs at the URLs that comply with a specified rule are not reported. The value can be String, RegExp, Function, or an array of the preceding three types. Example:

```javascript
__bl.setConfig({
  ignore: {
    ignoreUrls: [
      'http://host1/', // Character string
      /.+? host2. +/, // Regular expression
      function(str) { // Method
        if (str && str.indexOf('host3') >= 0) {
          return true;
        }
        return false;
      }
    ]
  }
});
```

**ignoreApis**

The **ignoreApis** attribute is used to ignore certain APIs that comply with a specified rule. The value can be String, RegExp, Function, or an array of the preceding three types. Example:

```javascript
__bl.setConfig({
  ignore: {
    ignoreApis: [
      'api1','api2','api3', // Character string
      /^random/, // Regular expression
      function(str) { // Method
        if (str && str.indexOf('api3') >= 0) return true;
        return false;
      }
    ]
  }
});
```

**ignoreErrors**
The `ignoreErrors` attribute is used to ignore certain JS errors that comply with a specified rule. The value can be `String`, `RegExp`, `Function`, or an array of the preceding three types. Example:

```javascript
__bl.setConfig({
  ignore: {
    ignoreErrors: [
      'test error', // Character string
      /^Script error/.? $/, // Regular expression
      function(str) { // Method
        if (str && str.indexOf('Unknown error') >= 0) return true;
        return false;
      }
    ]
  }
});
```

**config.sample:** configures sample reporting of logs

To reduce the number of reported items and load, this configuration item randomly samples and reports performance logs and successful API call logs. When processing logs, the background will restore the data based on the sampling configuration. Therefore, the configuration does not affect the reporting of other types of logs, such as JS error logs and failed API call logs.

The value of `sample` can be 1, 10, or 100, corresponding to the sampling ratio `1/sample`. That is, 1 indicates 100% sampling, 10 indicates 10% sampling, and 100 indicates 1% sampling. The default value is 1.

⚠️ **Notice:**

Sampling is performed randomly. If the number of reported items is small, this configuration item may cause large statistical result errors. This configuration item is recommended for the sites with daily average PVs of more than 1 million.

**sendResource:** reports static resources

If `sendResource` is set to `true`, the static resources loaded to the current page are reported when the page `load` event is triggered. If page loading is slow, you can view the static resource waterfall chart on the Session Traces page to determine the reason.

Sample code of `SendResource`

```html
<script>
!(function(c,b,d,a){c[a]||(c[a]={});c[a].config={pid:"xxxxxx",sendResource:true};
  with(b)with(body)with(insertBefore(createElement("script"),firstChild))setAttribute("crossorigin","",src=d)
})(window,document,"https://retcode.alicdn.com/retcode/bl.js","__bl");
</script>
```
Note:
The determination is made when the page `load` event is triggered. Therefore, configure `sendResource` in `config` (as shown in the preceding example). Do not use the `setConfig` method because this method may trigger the determination after the page `load` event is complete. In this situation, this configuration item is invalid.

c1 \ c2 \ c3: configuration items with custom fields.

In addition to the configuration items listed above, you may need more information when handling business problems. Therefore, the ARMS SDK provides three configuration items that allow you to customize fields. After the configuration, the fields are included in each reported log.

More information

- Advanced browser monitoring scenarios
- Methods user guide
7 Methods user guide

This topic describes some methods of the browser monitoring SDK and their usage.

The SDK opens some data reporting methods, which can be called on the page to report more data.

**api() for reporting the API call success rate**

This method reports the API call success rate on the page. The SDK listens for AJAX requests on the page and calls this method for reporting by default. If data on the page is requested using JSONP or other custom methods (such as the client SDK), you can call `api()` in the data request method for manual reporting.

**Note:**
To call this method, we recommend you set disabledHook to true in SDK configuration items. For detailed configuration, see Configuration items of the browser monitoring SDK.

**Call parameter description:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>api</td>
<td>String</td>
<td>The method name.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>success</td>
<td>Boolean</td>
<td>Specifies whether the call is successful.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>time</td>
<td>Number</td>
<td>The time consumed by the method call.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>code</td>
<td>String or number</td>
<td>The return code.</td>
<td>No</td>
<td>The null string (&quot;&quot;),</td>
</tr>
<tr>
<td>msg</td>
<td>String</td>
<td>The response information.</td>
<td>No</td>
<td>The null string (&quot;&quot;).</td>
</tr>
</tbody>
</table>

**Example**

```javascript
var begin = Date.now(),
    url = '/data/getTodoList.json';
$.ajax({
  url: url,
  data: {id: 123456}
})
```

error() for reporting the error information

This method reports JavaScript (JS) errors or exceptions you want to capture on the page.

Generally, the SDK listens for global errors on the page and calls this method to report exceptions. However, due to the same-origin policy of the browser, error details are usually out of reach. In this case, you must manually report such errors.

**Call parameter description:** __bl.error(error, pos)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>Error</td>
<td>The JS error object.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>pos</td>
<td>Object</td>
<td>The location where the error occurs. It contains the following three attributes:</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>pos.filename</td>
<td>String</td>
<td>The name of the file where the error occurs.</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>pos.lineno</td>
<td>Number</td>
<td>The number of the row where the error occurs.</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>pos.colno</td>
<td>Number</td>
<td>The number of the column where the error occurs.</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Example 1: Listen to and report JS errors on the page.**

```javascript
window.addEventListener('error', function (ex) {
    // Event parameters usually contain position information.
    window.__bl && __bl.error(ex.error, ex);
});
```
Example 2: Report a custom error.

```javascript
window.__bl && __bl.error(new Error('A custom error occurs.'), {
  filename: 'app.js',
  lineno: 10,
  colno: 15
});
```

sum() for summation

This method counts how many times a specific business event occurs.

**Call parameter description:** __bl.sum(key, value)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The name of the event.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>value</td>
<td>Number</td>
<td>The cumulative number in each report. Default value: 1.</td>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example**

```javascript
__bl.sum('event-a');
__bl.sum('event-b', 3);
__bl.sum('group-x::event-c', 2);
```

avg() for average value calculation

This method counts how many times a specific event occurs on average in a business scenario.

**Call parameter description:** __bl.avg(key, value)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The name of the event.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>value</td>
<td>Number</td>
<td>The number of reported items. Default value: 0.</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

```javascript
__bl.avg('event-a', 1);
__bl.avg('event-b', 3);
__bl.avg('events::event-c', 10);
__bl.avg('speed::event-d', 142.42);
```

addBehavior() for adding a user behavior

This method adds a custom user behavior to the end of the current behavior queue.

The SDK maintains a user behavior queue with a maximum of 100 records. When a JS error occurs, the SDK reports the current behavior queue and clears the queue. You can call this method to add a custom behavior to the end of the current behavior queue.

Note:
To call this method, you need to set `behavior` to true in SDK configuration.

Call parameter description: `__bl.addBehavior(behavior)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Object</td>
<td>Behavior data. Valid values:</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>name</strong>: behavior name of the string type. It is required, with a maximum length of 20 characters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>message</strong>: behavior content of the string type. It is required, with a maximum length of 200 characters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Required</td>
<td>Default value</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>--------------------------------------</td>
<td>----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>page</td>
<td>String</td>
<td>The page where the behavior happens.</td>
<td>No</td>
<td>Value of Location.pathname</td>
</tr>
</tbody>
</table>

**Example**

```json
{
  data: {
    name: 'string',
    message: 'string',
  },
  page: 'string'
}
```

**reportBehavior() for user behavior reporting**

When you call this method, the current behavior queue is reported immediately. This method is optional.

If you do not manually call this method, when a JS error occurs, the current behavior queue is automatically reported, with a maximum size of 100. If the queue contains more than 100 behavior records, behavior records are discarded from the header of the queue.

**Note:**
To call this method, you need to set `behavior` to true in SDK configuration.

**Call parameter description:** `__bl.reportBehavior()`

No request parameters.

**Other methods**

Non-log reporting methods are usually used for modifying some configuration items of the SDK.

**setConfig() for modifying configuration items**

This method allows you to modify some of the configuration items after SDK initialization.

For detailed configuration, see [Configuration items of the browser monitoring SDK](#).

**Call parameter description:** `__bl.setConfig(next)`
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>Object</td>
<td>The configuration items to be modified and their values.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Example: Change the value disableHook to true to disable automatic API reporting.**

```javascript
__bl.setConfig({
  disableHook: true
});
```

**setPage() for setting the name of the current page**

This method resets the page name of a page. PV will be reported again by default. This method is generally used for SPAs. For more information, see Advanced browser monitoring scenarios.

**Call parameter description:** `__bl setPage(next, sendPv)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>String</td>
<td>The new page name.</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>sendPv</td>
<td>Boolean</td>
<td>Specifies whether to report PV. PV is reported by default.</td>
<td>No</td>
<td>true</td>
</tr>
</tbody>
</table>

Example 1: Set the name of the current page to the current URL hash, and report PV again.

```javascript
__bl.setPage(location.hash);
```

Example 2: Set the name of the current page to 'homepage' without triggering PV reporting.

```javascript
__bl.setPage('homepage', false);
```
8 Statistical metrics

This topic explains the meanings of key statistical metrics on each page in application monitoring of Application Real-Time Monitoring Service (ARMS).

Satisfaction

Application Performance Index (APDEX) is an internationally accepted standard for evaluating application performance. According to APDEX, user experience of an application can be divided into three levels:

- Satisfied (0 to T)
- Tolerating (T to 4T)
- Frustrated (greater than 4T)

The following formula is used to calculate the APDEX score:

\[
\text{APDEX score} = \frac{\text{Satisfied samples} + \frac{\text{Tolerating samples}}{2}}{\text{Total samples}}
\]

For ARMS, First Paint Time (FPT) is taken into account. The default value of T is 2s.

JS stability

In ARMS, JS stability refers to the JS error rate of pages.

If any JS error occurs in a page view (PV) cycle, this PV cycle is an error sample.

Error rate = Error samples/Total samples

In addition to automatically reported JS errors, page exceptions include errors reported by manually calling methods in Methods user guide.

Page speed

In ARMS, the page speed indicates the FPT of a page.
When performance or speed statistics are collected, all data is collected by the Navigation Timing API defined by the World Wide Web Consortium (W3C).

Image source: www.w3.org

Field description

Table 8-1: Time consumed in each phase

<table>
<thead>
<tr>
<th>Reported field</th>
<th>Description</th>
<th>Formula</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dns</td>
<td>The time consumed for DNS resolution.</td>
<td>domainLookupEnd - domainLookupStart</td>
<td>None</td>
</tr>
<tr>
<td>tcp</td>
<td>The time consumed for TCP connection.</td>
<td>connectEnd - connectStart</td>
<td>None</td>
</tr>
<tr>
<td>ssl</td>
<td>The time consumed for SSL connection.</td>
<td>connectEnd - secureConnectionStart</td>
<td>This field is valid only when HTTPS is used.</td>
</tr>
<tr>
<td>ttfb</td>
<td>The time to first byte (TTFB), indicating the time consumed for a network request.</td>
<td>responseStart - requestStart</td>
<td>TTFB can be calculated in multiple ways. For TTFB calculation in ARMS, see Google development definition.</td>
</tr>
<tr>
<td>trans</td>
<td>The time consumed for data transmission.</td>
<td>responseEnd - responseStart</td>
<td>None</td>
</tr>
<tr>
<td>Reported field</td>
<td>Description</td>
<td>Formula</td>
<td>Remarks</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>dom</td>
<td>The time consumed for document object model (DOM) resolution.</td>
<td>domInteractive - responseEnd</td>
<td>None</td>
</tr>
<tr>
<td>res</td>
<td>The time consumed for resource loading.</td>
<td>loadEventStart - domContentLoadedEventEnd</td>
<td>This field indicates the time consumed for synchronously loading resources on the page.</td>
</tr>
</tbody>
</table>

Table 8-2: Key performance metrics

<table>
<thead>
<tr>
<th>Reported field</th>
<th>Description</th>
<th>Formula</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstbyte</td>
<td>The time used for generating the first response packet.</td>
<td>responseStart - domainLookupStart</td>
<td>None</td>
</tr>
<tr>
<td>fpt</td>
<td>The FPT.</td>
<td>responseEnd - fetchStart</td>
<td>This field indicates the duration from the time when a request is initiated to the time when the browser starts to parse the bytes of the first batch of HTML documents.</td>
</tr>
<tr>
<td>tti</td>
<td>The Time to Interact (TTI).</td>
<td>domInteractive - fetchStart</td>
<td>After the browser resolves all HTML documents and constructs the DOM, the browser starts to load resources.</td>
</tr>
<tr>
<td>ready</td>
<td>The time used for HTML loading, or the time when the DOM is ready.</td>
<td>domContentLoadedEventEnd - fetchStart</td>
<td>If a JavaScript (JS) script is executed synchronously on the page, the execution time of the JS script is equal to ready minus tti.</td>
</tr>
<tr>
<td>load</td>
<td>The time for completely loading the page.</td>
<td>loadEventStart - fetchStart</td>
<td>load = fpt + dom + (ready - tti) + res</td>
</tr>
</tbody>
</table>
API call success rate

API call success rate = Successful API calling samples/Total samples

In addition to the automatically reported AJAX requests, samples for API call success rate statistics also include the data reported by manually calling methods in Methods user guide.