ALIBABA CLOUD

Alibaba Cloud

Application Real-time Monitoring Service

Browser monitoring

Document Version: 20210304

Alibaba Cloud

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 because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should 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 this document based on the "status quo", "being defective", and "existing functions" of its products and services. 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 take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, 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 directly contact Alibaba Cloud for any errors of this document.

Document conventions

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

Table of Contents

1.What is ARMS Browser Monitoring?

Browser Monitoring of Application Real-Time Monitoring Service (ARMS) is applicable to scenarios such as web page monitoring, Weex monitoring, and mini-program monitoring. You can monitor web pages and mini-programs based on the following metrics: the page loading speed (speed test), page stability (JavaScript errors), and success rate of calls to external services that are APIs.

Why is Browser Monitoring necessary?

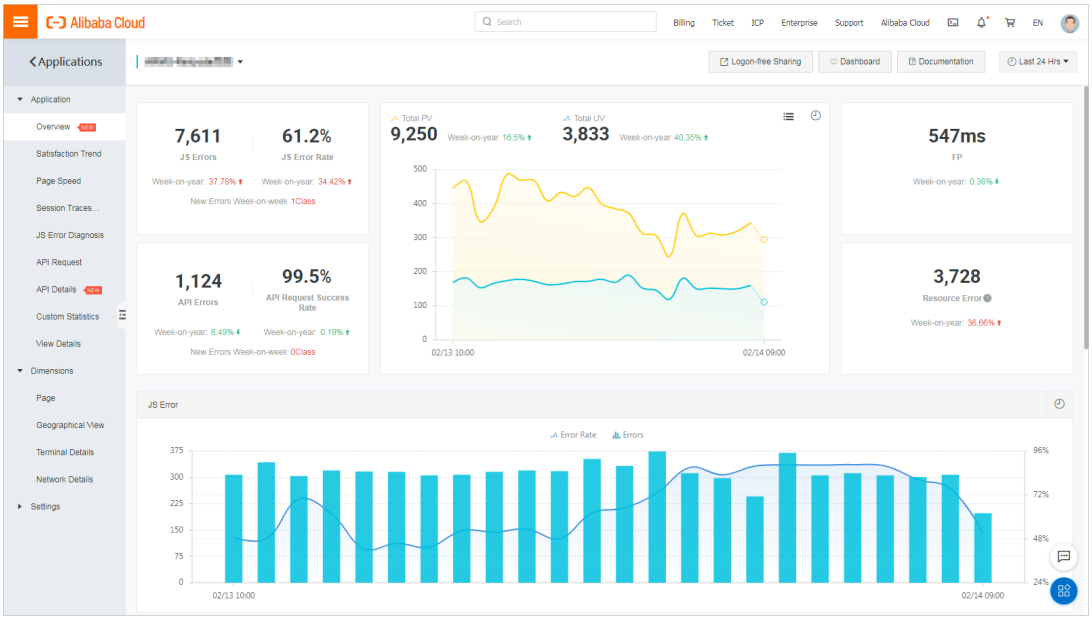
When a user accesses a service, the whole process can be divided into three phases: page generation (server status), page loading, and page runtime.

To ensure stable online services, the server monitors the status of these services. Existing server monitoring systems are quite mature, but the monitoring of page loading and runtime is far from satisfactory. For example:

- You cannot immediately capture the errors that users encounter when they access your website.
- 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

ARMS Browser Monitoring monitors the status of page loading and runtime, and reports data to the logger. The data includes the page load performance, runtime exceptions, and API call status and consumed time. Then, the platform monitors the access of all real online users based on the rich real-time log analytics and processing services provided by ARMS. Finally, the platform presents visual reports to help you detect and diagnose problems at the earliest opportunity.



Scenarios

ARMS Browser Monitoring supports the following scenarios:

- Web/H5

  - Install the browser monitoring probe by using CDN
  - Implement browser monitoring by using npm
- Weex
  Implement browser monitoring in the Weex environment
- Mini-programs
  - Monitor DingTalk mini programs
  - Monitor Alipay mini programs
  - Monitor WeChat mini programs
  - Monitor other mini programs

Capabilities

ARMS Browser Monitoring provides the following capabilities:

- Page speed
- JS errors
- API request
- Front-to-back tracking

Browser and platform compatibility

Browser or platform	Supported version	Automatic report by the SDK	Manual report
Safari	Safari 9+	✔	✔
Chrome	Chrome 49+	✔	✔
IE	IE 9+	✔	✔
Edge	Edge 12+	✔	✔
Firefox	Firefox 36+	✔	✔
Opera	Opera 43+	✔	✔
Safari for iOS	Safari for iOS 9.2+	✔	✔
Android Browser	android_webkit 4.4.2+	✔	✔
Weex	Weex 0.16.0+	✖	✔

2.Quick start

2.1. Browser monitoring overview

The browser monitoring feature of Application Real-Time Monitoring Service (ARMS) allows you to monitor websites, Weex frameworks, and mini programs. Read the corresponding documents to get started with browser monitoring.

Web scenarios

- Install the browser monitoring probe by using CDN
- Implement browser monitoring by using npm

Weex scenarios

Implement browser monitoring in the Weex environment

Mini-program scenarios

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

2.2. For Web applications

2.2.1. Implement browser monitoring by using npm

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

Install the npm package

Install the npm package named `alife-logger`.

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

Initialize the SDK

Use `BrowserLogger.singleton` to initialize the SDK.

```
const BrowserLogger = require('alife-logger');
        // Use BrowserLogger.singleton(conf) config to load the configurations specified by config.
        const __bl = BrowserLogger.singleton({
        pid: 'your-project-id',
        // Specify the path to which the logs are uploaded.
        // If you want to deploy the SDK in the Singapore region, set the path to https://arms-retcode-sg.aliyuncs.com/r.png?.
        // If you want to deploy the SDK in the US (Silicon Valley) region, set the path to http://arms-us-west-1.console.aliyun.com/r.png?.
        imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?',
        // Set other configurations specified by config.
        });
```

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

```
uid: 'xxx', // The UID is used to identify a user. Set its value based on your businesses.
```

Example:

```
const BrowserLogger = require('alife-logger');
        // Use BrowserLogger.singleton(conf) config to load the configurations specified by config.
        const __bl = BrowserLogger.singleton({
          pid: 'your-project-id',
            // Specify the path to which the logs are uploaded.
            // If you want to deploy the SDK in the Singapore region, set the path to https://arms-retcode-sg.aliyuncs.com/r.png?.
            // If you want to deploy the SDK in the US (Silicon Valley) region, set the path to http://arms-us-west-1.console.aliyun.com/r.png?.
           uid: 'xxx', // The UID is used to identify a user. Set its value based on your businesses.
            imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?',
          // Set other configurations specified by config.
        });
```

API operation

> **Note** This method applies only to the implementation of browser monitoring by using npm.

The following table describes the parameters you can configure for `BrowserLogger.singleton(config,prePipe)`.

This method is a static method that returns a single-instance object. The loaded config and prePipe parameters take effect only when the method is called for the first time. Only generated instances are returned for subsequent calls.

Parameter	Type	Description	Required	Default value

Parameter	Type	Description	Required	Default value
config	Object	The site configurations. For more information about other parameters you can configure in config, see SDK reference.	Yes	None
prePipe	Array	The content that must be reported in advance.	No	None

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

You can use `BrowerLogger.singleton` to obtain instances.

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

For more information about how to use other methods of `__bl`, see API reference.

The configurations of config is the same as those when you use CDN to install the ARMS browser monitoring agent. For more information, see SDK reference.

If some data must be reported based on the logic of the code executed before `BrowserLogger.singleton()` is called, you must pre-report the data. For more information, see Pre-report data.

```
const BrowserLogger = require('alife-logger');
        // The structure of pipe is the same as the that when you use CDN to install the ARMS browser monitoring agent.
        const pipe = [
        // Report the current HTML page as an API request.
        ['api', '/index.html', true, performance.now, 'SUCCESS'], // This is equivalent to __bl.api(api, success, time, code, msg).
        // After the SDK is initialized, enable automatic Single Page Application (SPA) resolution.
        ['setConfig', {enableSPA: true}]
        ];
        const __bl = BrowserLogger.singleton({pid:'Unique site ID'},pipe);
```

Method used to return a single instance: @static singleton()

Other reporting methods

Config configurations

Data pre-report

Common SDK parameters

The browser monitoring feature of ARMS allows you to configure a variety of SDK parameters to meet more requirements. The following table describes the parameters that you can configure in the scenarios described in this topic.

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	No	Automatically generated by SDK
tag	String	The input tag. Each log carries a tag.	No	None
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	`undefined`
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	`prod`
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in `1/sample` The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	`1`
behavior	Boolean	Whether to record the error user behavior to facilitate troubleshooting.	No	`true`
enableSPA	Boolean	Listen to the hashchange event on the page and report the PV again. This method is applicable to single-page application scenarios.	No	`false`

Parameter	Type	Description	Required	Default Value
enableLinkTrace	Boolean	For more information about tracing frontend and backend links, see Use the front-to-back tracing feature to diagnose causes of API errors.	No	false

For more information about the SDK parameters that you can configure, see SDK reference.

Related information

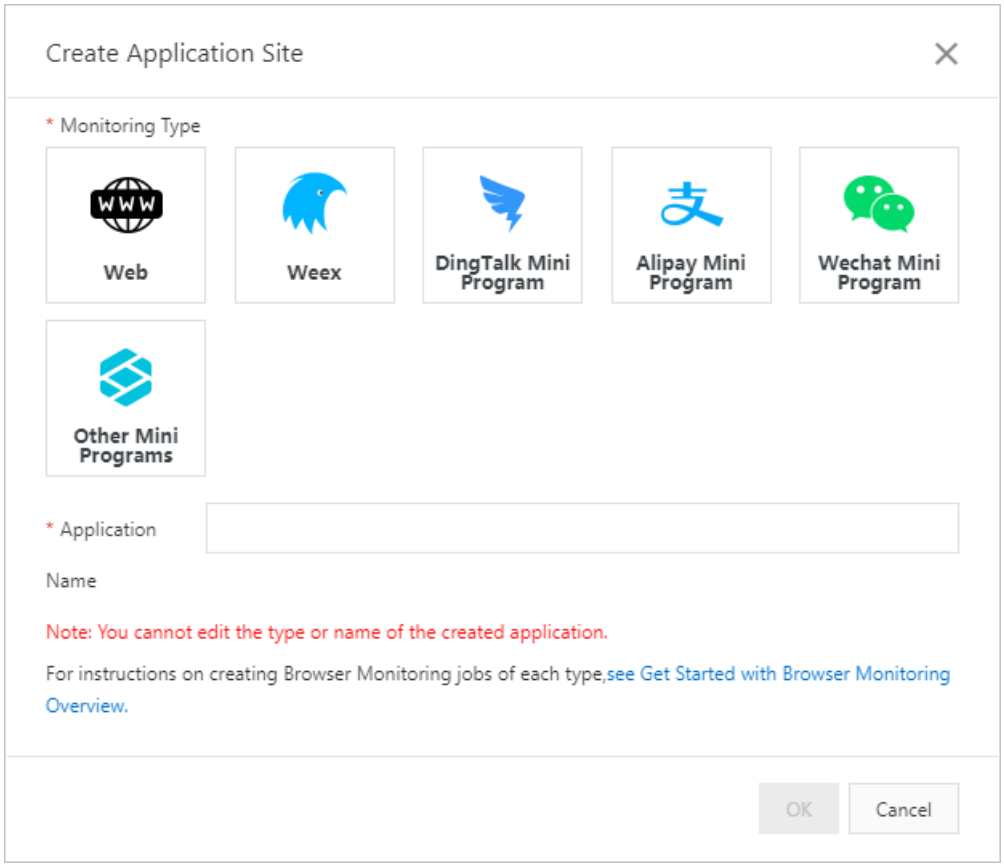
- Browser monitoring overview
- SDK reference
- Install the browser monitoring probe by using CDN
- Implement browser monitoring in the Weex environment

2.2.2. Install the browser monitoring probe by using CDN

To use the browser monitoring feature of Application Real-Time Monitoring Service (ARMS) to monitor web applications, you must install the probe by using Alibaba Cloud Content Delivery Network (CDN) or node package manager (npm). This topic shows you how to install the ARMS browser monitoring probe on web applications by using CDN.

Install the browser monitoring probe

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click **Create Application Site** in the upper-right corner.
3. In the **Create Application Site** dialog box, select web as the monitoring type, enter the application name, and then click **OK**.



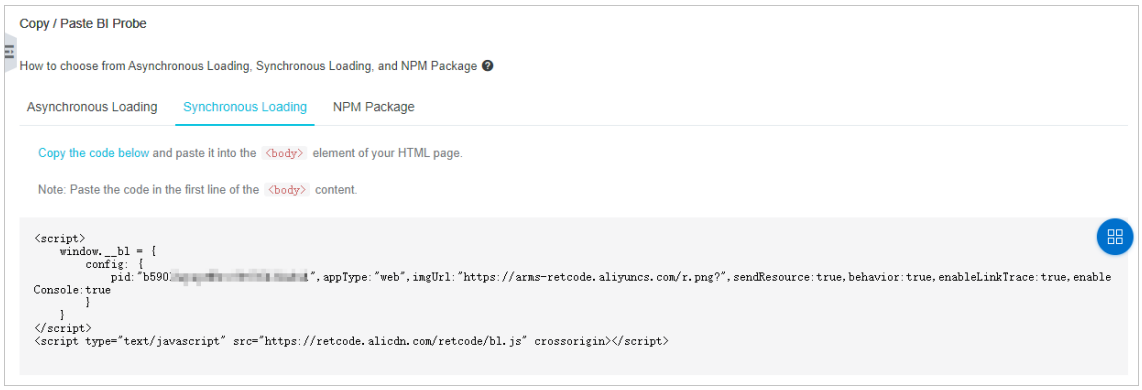
4. On the **settings** page of the application, select the required options in the **SDK Extension Configuration** section. The code of the BI probe to be pasted to the page is then generated based on the selected options.
   - **Disable Automatic API Reporting**: If you select this option, you must manually call the `__bl.api()` method to report the API success rate.
   - **Enable Automatic SPA Parsing**: If you select this option, ARMS monitors the `hashchange` event of the page and automatically reports page views (PVs). This option is applied to single-page applications (SPAs).
   - **Enable Data Collection of First Meaningful Paint**: If you select this option, ARMS collects data of First Meaningful Paint (FMP).
   - **Enable Page Resources Reporting**: If you select this option, static resources loaded on the page are reported when the onload event is triggered.
   - **Associate with Application Monitoring**: If you select this option, API requests are in end-to-end association with application monitoring.
   - **Enable User Behavior Trace**: If you select this option, you can view user behavior trace in JS Error Diagnosis.
   - **Enable Console Tracing**: If you select this option, user behavior is traced in the console. The behavior can be `error`, `warn`, `log`, or `info`.

     > **Notice** This feature affects the path of the Console panel.

5. Install the probe by using one of the following methods:
   - Asynchronous loading: Copy the provided code, paste it to the first line of the `<body>` element

of the HTML page, and then restart the application.

- Synchronous loading: Copy the provided code, paste it to the first line of the `<body>` element of the HTML page, and then restart the application.



- npm package:
  a. Run the following command to install the npm package:

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

  b. Copy and run the following command from the console to initialize the npm package:

     ```
     const BrowserLogger = require('alife-logger');
     const __bl = BrowserLogger.singleton({pid:"b590lhguqs@8cc3f63543d****",appType:"web",imgUrl:"https://arms-retcode.aliyuncs.com/r.png?",sendResource:true,behavior:true,enableLinkTrace:true,enableConsole:true});
     ```

Differences between asynchronous loading and synchronous loading

- Asynchronous loading: It is also known as non-blocking loading. In asynchronous loading, the browser continues to process subsequent pages no matter whether JavaScript loading is complete. We recommend that you use this method if you require high page performance.

  > **Notice** If you use asynchronous loading, ARMS cannot capture JavaScript errors or resource loading errors before the monitoring SDK completes the initialization.

- Synchronous loading: It is also known as blocking loading. In synchronous loading, the browser does not continue to process subsequent pages until the JavaScript loading is complete. We recommend that you use synchronous loading if you need to capture JavaScript errors and resource loading errors during the whole process.

Custom UID

When the ARMS browser monitoring probe is installed by using synchronous or asynchronous loading, the web SDK automatically generates a user ID (UID) to collect information such as the number of unique visitors (UVs). The UID can be used to identify a user but does not have business attributes. If you want to customize a UID, add the following content to `config` in the code:

```
uid: 'xxx', // The UID is used to identify a user. You can specify the UID based on your business requirements.
```

Sample code:

```
<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 options in the SDK Extension Configuration section, the code changes. Copy and paste the code again.

Common SDK parameters

The browser monitoring feature of ARMS allows you to configure a variety of SDK parameters to meet additional requirements. The following table describes the common parameters that you can specify in the scenarios described in this topic.

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	No	Automatically generated by SDK
tag	String	The input tag. Each log carries a tag.	No	None
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	`undefined`

Parameter	Type	Description	Required	Default Value
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	prod
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in 1/sample The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	1
behavior	Boolean	Whether to record the error user behavior to facilitate troubleshooting.	No	true
enableSPA	Boolean	Listen to the hashchange event on the page and report the PV again. This method is applicable to single-page application scenarios.	No	false
enableLinkTrace	Boolean	For more information about tracing frontend and backend links, see Use the front-to-back tracing feature to diagnose causes of API errors.	No	false

The browser monitoring feature of ARMS also provides more SDK parameters to further meet your requirements. For more information, see SDK reference.

Related information

- Browser monitoring overview
- SDK reference
- Implement browser monitoring by using npm
- Implement browser monitoring in the Weex environment

2.3. For Weex

2.3.1. Implement browser monitoring in the Weex environment

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

Import the NPM package

To use the browser monitoring feature of ARMS in the Weex environment, run the following command in your 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 based on the following sample code.

> **Note** To call the singleton(props) method at the entry point of a Weex application to obtain instances, configure parameters for the imported props. For more information, see the following sections:
>
> - General method: @static singleton()
> - General method: setPage()
> - General method: setConfig()

```
// Add the following content to 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',
  // Configure the UID to generate unique visitor (UV) reports.
  page: 'Lazada | Home',
  // Configure the name of the initial page. The SDK sends page view (PV) reports after the initialization is co
mplete.
  imgUrl: 'https://arms-retcode.aliyuncs.com/r.png?',
  // Specify the path to which the reports are sent. If you want to deploy the SDK in the Singapore region, se
t the path to 'https://arms-retcode-sg.aliyuncs.com/r.png?'.
  // The following code provides an example on how to use the GET method to send reports:
  sendRequest: (data, imgUrl) =>{
    const url = imgUrl + serialize(data);
    fetch({
      method: 'GET',
      url
    });
  },
  // The following code provides an example on how to use the POST method to send reports:
  postRequest: (data, imgUrl) =>{
    fetch({
      method: 'POST',
      type: 'json',
      url: imgUrl,
      body: JSON.stringify(data)
    });
  }
});
export
default wxLogger;
```

Report logs

Call the corresponding 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 when the method is called for the first time. The following table describes the parameters you can configure when you call the method.

This method can be used to initialize the SDK at the application entry point. For more information, see Initialize the SDK.

Parameters of WeexLogger.singleton(props)

Parameter	Type	Description	Required	Default value
pid	String	The site ID.	Yes	None
page	String	The page name after initialization.	No	None
uid	String	The ID of the user.	Yes	None
imgUrl	String	The path to which the logs are uploaded. The path ends with a question mark (?).	No	None

General method: setPage()

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

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

Parameters of wxLogger.setPage(nextPage)

Parameter	Type	Description	Required	Default value
nextPage	String	Page Name	Yes	None

General method: setConfig()

setConfig() is used to modify configurations after the SDK is initialized. The configuration method is the same as that of singleton(). For more information about the parameters that you can configure for the method, see setConfig().

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

Parameters of wxLogger.setConfig(config)

Parameter	Type	Description	Required	Default value
config	Object	The configuration items you want to modify.	Yes	None
uid	String	The user ID used to collect the unique visitor (UV) data.	Yes	Storage settings

Log reporting methods

For more information, see the log reporting methods in API reference.

Common SDK parameters

The browser monitoring feature of ARMS allows you to configure a variety of SDK parameters to meet more requirements. The following table describes the parameters that you can configure in the scenarios described in this topic.

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	Yes	N/A
tag	String	The input tag. Each log carries a tag.	No	None
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	`undefined`
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	`prod`

Parameter	Type	Description	Required	Default Value
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in `1/sample` The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	`1`

For more information about the SDK parameters that you can configure, see SDK reference.

2.4. For mini programs

2.4.1. Monitor DingTalk mini programs

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

Background information

For background information about the DingTalk mini program, see DingTalk mini program Migrate an instance across regions based on Global Replica.

Procedure

To monitor a DingTalk mini program, install and initialize the Node Package Manager (NPM) package, report monitoring data, and then configure security domains.

1. Install and initialize the NPM package.
   i. Install the NPM package named alife-logger in the DingTalk mini program. This module can then be used to report logs.

   ```
   npm install alife-logger
   ```

   ii. Add the following to */utils* Directory in the monitor.js file to complete the initialization.

   > **Note** You can specify the name and storage path of the JavaScript (JS) file.

   ```
   import EAppLogger from 'alife-logger/eapp';
   const Monitor = EAppLogger.init({
     pid: 'xxx',
     region: "cn",
   });
   export default Monitor;
   ```

   > **Note** For detailed configuration of parameters, see Common parameters of SDK.

2. You can use the following methods to automatically collect and report page views (PVs), error, API requests, performance, and health data.

i. In app.js, use the Monitor.hookApp(options) Method silently captures the Error class log. Of which options That is, the corresponding Object configuration for the App layer.

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

ii. In the JS file of the Page by Monitor.hookPage(options) Method to silently report the API request, PV, and Health data.

```
import Monitor from '/util/monitor';
Page(Monitor.hookPage({
  data: {},
  onLoad(query) {
  },
  onReady() {
  },
  onShow() {
  },
  onLoad(query) {
  },
  onHide() {
  },
  onUnload() {
  }
}));
```

3. Set security domains.
   - If region Set as `cn` , add the arms-retcode.aliyuncs.com to the HTTP security Domains.
   - If region Set as `sg` , add the arms-retcode-sg.aliyuncs.com to the HTTP security Domains.

Basic API methods for automatic tracking

Method	Parameter	Description	Scenario
hookApp	{}	Enter the parameters of the application.	API lifecycle management automatically tracks the application.

Method	Parameter	Description	Scenario
hookPage	{}	Enter the parameters of the page.	API lifecycle management automatically tracks the page.

Note For mini program monitoring items that need to use hookApp and hookPage to embed lifecycle hitting points, they must comply with the standard Mini program specifications on apps and pages. That is, the App layer has **onError** , the Page layer has **onShow** , **onHide** , **onUnload** . For usage examples, see Procedure.

Other API methods

Method	Parameter	Description
setCommonInfo	{[key: string]: string;}	Set basic log fields for scenarios such as phased release.
setConfig	{[key: string]: string;}	Set the config field. For more information about the operation, see SDK configuration items parameters .
pageShow	{}	Report the PV data.
pageHide	{}	Report the health data.
error	String/Object	Report error logs.
api	For more information, API reference	Report the API request logs.
sum/avg	String	Report custom sum and average logs.

Advanced scenarios

If the basic methods cannot meet your needs, see the following advanced scenarios:

- Manually report the API operations. Automatic reporting is disabled.
  i. Soon disableHook Set as `true` , not reported silently dd.httpRequest The requested log.
  ii. Manually called api() Method to report API-related information.
- Disable automatic reporting and enable manual tracking.
  i. No longer used in App and Page JS files. hookApp , hookPage Method.
  ii. To send the PV data of the current Page, on the Page onShow Call under method pageShow() Method.

Note Do not interact with hookPage() Method uses this method at the same time. Otherwise, PV logs may be reported repeatedly.

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

iii. To send Health-class data for the current Page, statistics on the Health of the current Page and Page stay time, on the Page onHide and onUnload Call under method pageHide() Method.

> **Note** Do not interact with hookPage() METHOD. This method is used at the same time. Otherwise, repeated log reporting may occur.

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

Common parameters of SDK

ARMS browser monitoring provides a series of SDK parameters. You can configure the parameters to meet additional requirements. The following table describes the common parameters suitable for the scenarios described in this topic.

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	No	Automatically generated by SDK
tag	String	The input tag. Each log carries a tag.	No	None
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	`undefined`

Parameter	Type	Description	Required	Default Value
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	`prod`
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in `1/sample` The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	`1`
behavior	Boolean	Whether to record the error user behavior to facilitate troubleshooting.	No	`false`
enableLinkTrace	Boolean	For more information about tracing frontend and backend links, see Use the front-to-back tracing feature to diagnose causes of API errors.	No	`false`

ARMS browser monitoring also provides other SDK parameters. For more information, see SDK reference.

Related information

- What is ARMS Browser Monitoring?
- 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.

i. Introduce the npm package named `alife-logger` in the Alipay mini program project to facilitate log reporting.

```
npm install alife-logger
```

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

> **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.

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

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

```
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.

Parameter	Type	Description	Required	Default value
pid	String	The ID of the site.	Yes	null
uid	String	The ID of the user, which is used to collect the unique visitor (UV) data.	No	Storage setting
tag	String	The input tag. Each log carries a tag.	No	None
disabled	Boolean	Specifies whether the log reporting function is disabled.	No	false

Parameter	Type	Description	Required	Default value
sample	Integer	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.	No	1
enableLinkTrace	Boolean	Specifies whether front-to-back tracing is supported.	No	false
disableHook	Boolean	Specifies whether to disable monitoring my.httpRequest. By default, the request is monitored, and the API request success rate is reported.	No	false
sendRequest	Function	The method for sending logs. If this parameter is not configured, the default value my.httpRequest is used.	No	my.httpRequest
getCurrentPage	Function	The method for obtaining the current page.	No	getCurrentPage

Basic methods for automatic instrumentation

Method	Parameter	Remarks	Scenario
hookApp	{}	Enter the source app parameters.	The app lifecycle API automatically starts instrumentation.
hookPage	{}	Enter the source page parameters.	The page lifecycle API automatically starts instrumentation.

> **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

Method	Parameter	Remarks
setCommonInfo	{[key: string]: string;}	Set basic log fields for the scenarios such as phased release.
setConfig	{[key: string]: string;}	Set the config field.
pageShow	{}	Report the PV data.
pageHide	{}	Report the health data.
error	String/Object	Report error logs.
api	See also API reference	Report the API request logs.
sum/avg	String	Report the custom sum and average logs.

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).
  i. Set **disableHook** to `true`. The logs of the **my.httpRequest** request are not reported automatically.
  ii. Manually call **api()** to report the API request results.
- Disable automatic reporting and enable manual instrumentation.
  i. No longer use the hookApp and hookPage methods in the app.js and page.js files.
  ii. 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.

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

  iii. 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.

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

More information

- Browser monitoring overview
- Monitor DingTalk mini programs
- Monitor WeChat mini programs
- Monitor other mini programs

2.4.3. Monitor WeChat mini programs

This topic describes how to use the browser monitoring feature of Application Real-Time Monitoring Service (ARMS) to monitor WeChat mini programs and introduces the common SDK configurations, API operations, and advanced scenarios of the browser monitoring feature.

Background information

For more information about WeChat mini programs, visit WeChat mini programs.

Basic usage

To monitor a WeChat mini program, you must perform the following basic steps:

1. Obtain and initialize the monitoring SDK for WeChat mini programs.
   i. Copy the content of the JS file and paste the content to the *wxLogger.js* file in the */utils* folder of the WeChat mini program that you want to monitor.

ii. Add the following content to the monitor.js file in the */utils* folder of the WeChat mini program to initialize the monitoring SDK.

> **Note** You can specify the name of the JS file and the path used to store the file.

- If the project is integrated by using the node module (require) method, add the following content to the monitor.js file:

```
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 monitor.js file:

```
import WXLogger from './wxLogger.js';
  const Monitor = WXLogger.init({
    pid: 'xxx',
    region: 'cn'
  });
  export default Monitor;
```

> **Note** For more information about parameter configurations, see Common SDK parameters.

2. Use the following methods to automatically collect the PV, error, API, performance, and health data of the WeChat mini program:

i. In app.js, call **Monitor.hookApp(options)** to automatically capture error logs. The **options** parameter is the object settings in the app.

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

ii. In page.js, call **Monitor.hookPage(options)** to automatically report the API, PV, and health data of the WeChat mini program.

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

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

Basic methods for automatic instrumentation

Method	Parameter	Description	Scenario
hookApp	{}	Uses the parameters of App.	Perform automatic instrumentation during the lifecyle of App.
hookPage	{}	Uses the parameters of Page.	Perform automatic instrumentation during the lifecyle of Page.

Note If you want to use the hookApp and hookPage methods in the monitoring project of your WeChat mini program for instrumentation during the lifecycle of App and Page, the hookApp method must include **onError** and the hookPage method must include **onShow**, **onHide**, and **onUnload**. For more information about how to use these methods, see Basic usage

Methods for other settings

Method	Parameter	Description
setCommonInfo	{[key: string]: string;}	Sets basic log fields in scenarios such as canary release.

Method	Parameter	Description
setConfig	{[key: string]: string;}	Set the config field. For more information about the operation, see SDK configuration items parameters .
pageShow	{}	Reports the PV data.
pageHide	{}	Reports the health data.
error	String/Object	Reports error logs.
api	For more information, see API reference.	Reports API request logs.
sum/avg	String	Reports the custom sum and average logs.

Advanced scenarios

When the basic usage of application monitoring cannot meet your requirements, try the usage in the following advanced scenarios:

- Manually report the API request results.
  i. Set **disableHook** to `true`. The logs of the **wx.request** request are not automatically reported.
  ii. Manually call **api()** to report the API request results.
- Disable automatic reporting and enable manual instrumentation.
  i. Do not use the hookApp and hookPage methods in the app.js and page.js files.
  ii. To send the PV data of the current page, call **pageShow()** in the **onShow** method of Page.

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

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

  iii. To send the health data including the health and browsing time of the current page, call **pageHide()** under the **onHide** and **onUnload** methods of Page.

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

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

Common SDK parameters

The browser monitoring feature of ARMS allows you to configure a variety of SDK parameters to meet more requirements. The following table describes the parameters that you can configure in the scenarios described in this topic.

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	No	Automatically generated by SDK
tag	String	The input tag. Each log carries a tag.	No	None
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	`undefined`
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	`prod`

Parameter	Type	Description	Required	Default Value
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in `1/sample` The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	`1`
behavior	Boolean	Whether to record the error user behavior to facilitate troubleshooting.	No	`false`
enableLinkTrace	Boolean	For more information about tracing frontend and backend links, see Use the front-to-back tracing feature to diagnose causes of API errors.	No	`false`

For more information about the SDK parameters that you can configure, see SDK reference.

Related information

- Browser monitoring overview
- 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.
   i. In your mini program project, introduce the npm package named `alife-logger` to facilitate log reporting.

```
npm install alife-logger
```

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

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

```
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;
    }
  }
});
export default Monitor;
```

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

2. Report logs.

i. 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.

```
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.

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

ii. 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.

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

```
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.

> **Note** For more information about the methods, see Methods.

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

Parameter	Type	Description	Required	Default value
pid	String	The ID of the site.	Yes	null
uid	String	The ID of the user, which is used to collect the UV data.	No	Storage setting
tag	String	The input tag. Each log carries a tag.	No	None
disabled	Boolean	Specifies whether the log reporting function is disabled.	No	false

Parameter	Type	Description	Required	Default value
sample	Integer	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.	No	1
disableHook	Boolean	Specifies whether to disable monitoring my.httpRequest. By default, the request is monitored, and the API request success rate is reported.	No	false
sendRequest	Function	The method for sending logs. If this parameter is not configured, the logs cannot be sent.	Yes	None
getCurrentPage	Function	The method for obtaining the current page.	Yes	None

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.

Parameter	Type	Description
url	String	The URL to which the log is reported.
resData	Object	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.

`sendRequest` configuration 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) {
        //...
      }
    });
  }
```

Methods

Method	Parameter	Remarks
hookApp	{}	Enter the source app parameters. The app lifecycle API automatically starts instrumentation.
hookPage	{}	Enter the source page parameters. The page lifecycle API automatically starts instrumentation.
setCommonInfo	{[key: string]: string;}	Set basic log fields for the scenarios such as phased release.
setConfig	{[key: string]: string;}	Set the config field.
pageShow	{}	Report the PV data.
pageHide	{}	Report the health data.
error	String/Object	Report error logs.
api	See also API reference	Report the API request logs.
sum/avg	String	Report the custom sum and average logs.

> **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.

  ```
  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.

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

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

- Browser monitoring overview
- Monitor DingTalk mini programs
- Monitor Alipay mini programs
- Monitor WeChat mini programs

3.Console functions

3.1. Browser monitoring dashboard

Application Real-Time Monitoring Service (ARMS) provides the dashboard of the browser monitoring feature. This allows you to view all the critical monitoring data of the monitored application in real time.

Procedure

1. Log on to the ARMS console. In the left-side navigation pane, click **Browser Monitoring**.
2. On the **Browser Monitoring** page, click the name of the application.
3. On the **Overview** page, click **Dashboard** in the upper-right corner.
4. On the Dashboard page, you can view the browser monitoring data of the application. The monitoring data includes the JavaScript (JS) error rate and API request success rate.

Features

The dashboard of ARMS browser monitoring provides all the critical monitoring data of the monitored application in real time. We recommend that you display the dashboard on a big screen.

> **Note** The monitoring data can be updated once a minute.

Related operations

- View the monitoring data on the dashboard.
- Move the pointer over the curve in a chart. The statistics at the point in time appears.
- To show the dashboard in full screen, click **Full Screen** in the upper-right corner.

Parameters

The dashboard of ARMS browser monitoring includes the following parameters.

- JS error rate
  - JS Error Rate: the JS error rate.
  - Compared with Yesterday's Average: the increase or decrease ratio of the average JS error rate compared with the average JS error rate in the previous day.
  - Statistical Chart: the curve of the JS error rate in the last hour.
  - Page Ranked by Error Rate: the JS error rate ranking.
- Alerts in the last 24 hours
  - Alarms: the number of alerts.
  - Recent 3 Alarms: the alerts from browser monitoring in the last 24 hours.
- PV/UV
  - Today's PV: the number of page views (PVs) of the monitored application on the current day.
  - Today's UV: the number of unique visitors (UVs) of the monitored application on the current day.
  - Compared with Yesterday: the increase or decrease ratio of PVs and UVs compared with the PVs and UVs in the previous day.

  - Statistical Chart: the curve of PVs or UVs in the last hour.
  - Statistical Table: the top five regions with the most PVs and UVs and their corresponding PVs and UVs.
  - High Page View TOP5: the top five services with the most PVs.
- API request success rate
  - API Request Success Rate: the success rate of API requests.
  - Compared with Yesterday's Average: the increase or decrease ratio of the API request success rate compared with the API request success rate in the previous day.
  - Statistical Chart: the curve of the API request success rate in the last hour.
  - Service Ranked by API Success Rate: the API request success rate ranking.
- Page speed
  - Page Speed: the First Paint Time (FPT). Unit: milliseconds.
  - Compared with Yesterday's Average: the increase or decrease ratio of the page speed compared with the average page speed in the previous day.
  - Statistical Chart: the curve of the page speed in the last hour.
  - Low Page Speed Top 5: the top five services with the lowest page speeds.

Related information

- Page speed
- JS error diagnostics
- API request
- Create ARMS alerts

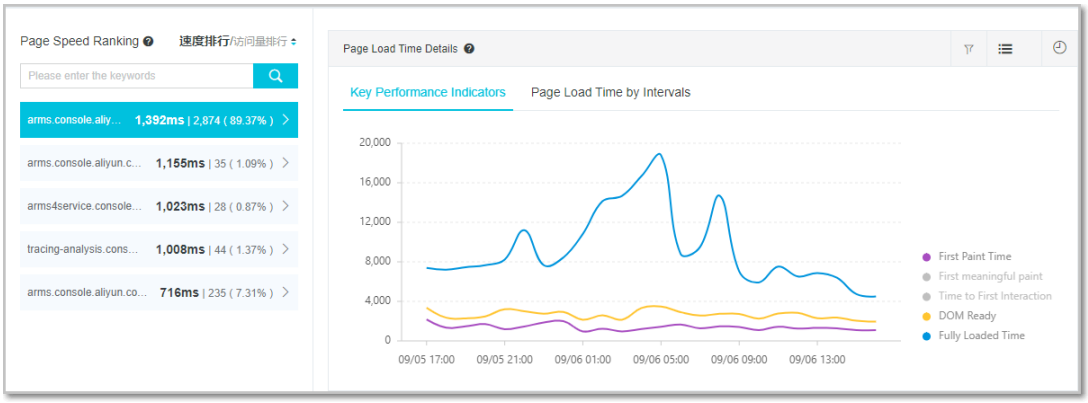
3.2. Page speed

This topic describes the page speed feature that is provided by Browser Monitoring of Application Real-Time Monitoring Service (ARMS).
After your application is connected to ARMS Browser Monitoring, you can view the following performance data of the application on the **Page Speed** page:

> **Note** You can manually report custom performance metrics. You can customize first paint time, time to interact, or other performance metrics. For more information, see performance().

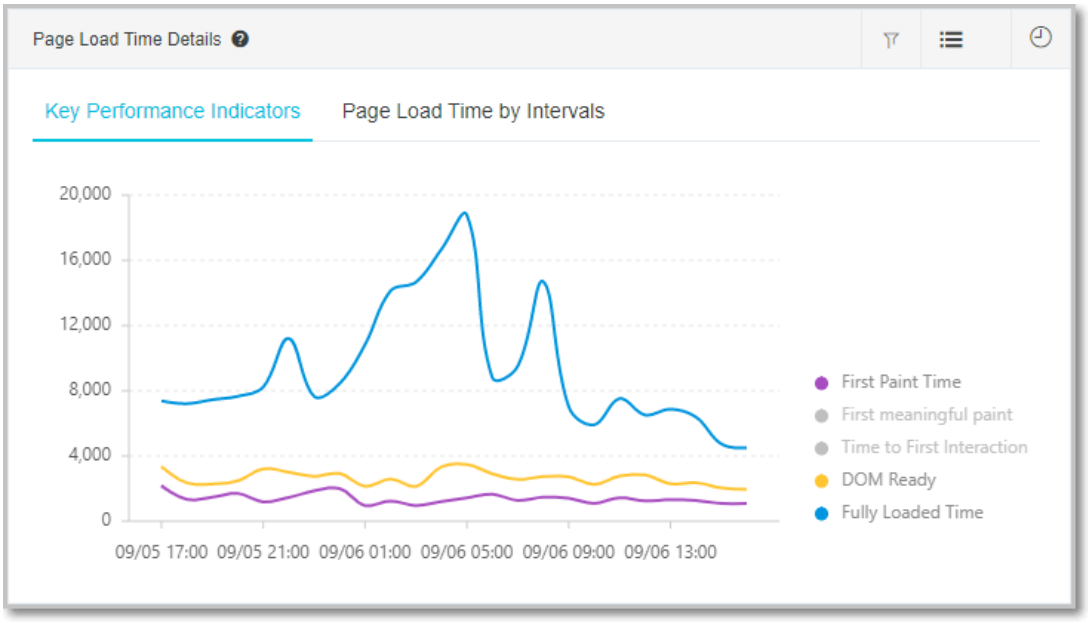
- Page Load Time Details section
- Page Load Waterfall Plot section
- Performance Distribution section
- Slow Page Session Trace section
- Page loading distribution
- Custom performance metrics that are manually reported

In the **Page Speed** section, you can rank the pages by first paint time or page view (PV), and change the display order by clicking the Up or Down arrow.



Page Load Time Details section

The following figure shows the metrics in this section.

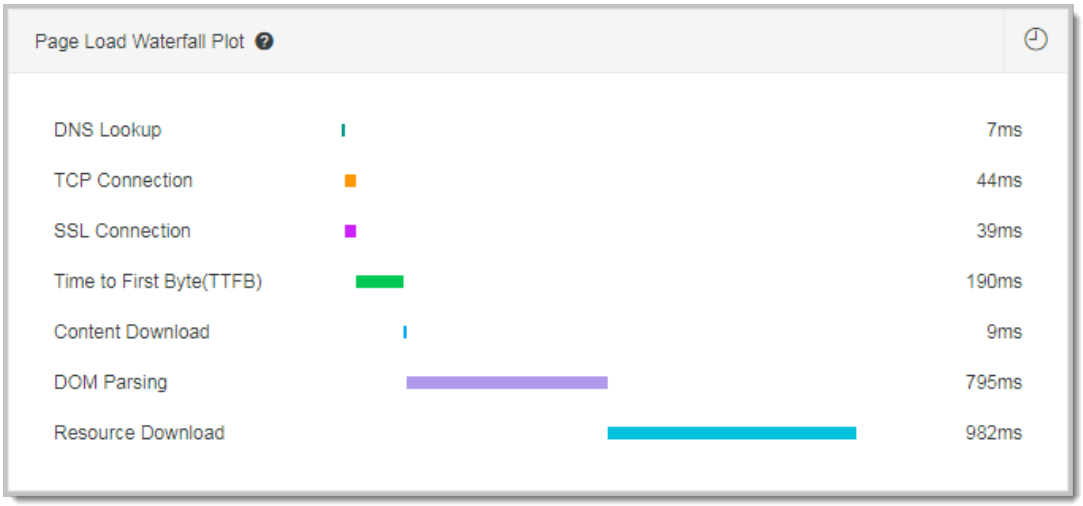


Note

- The data in the line chart is displayed based on the average values of the metrics within the specified time range. The average value reflects the average performance over a period of time. However, this value is sensitive to extreme values and large fluctuations. For example, if the overall page loading speed is slow due to weak network connection during an access request, the average response time is high. You can click the icon in the upper-right corner to remove extreme values. In this way, the impact of extreme values on the overall performance trend is avoided.
- If the data in the line chart increases sharply, you can use Performance Sample Distribution and Slow Page Session Trace sections to identify the issue.

Page Load Waterfall Plot section

The waterfall chart 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, we recommend that you implement the related countermeasures at specific stages.



Performance Distribution section

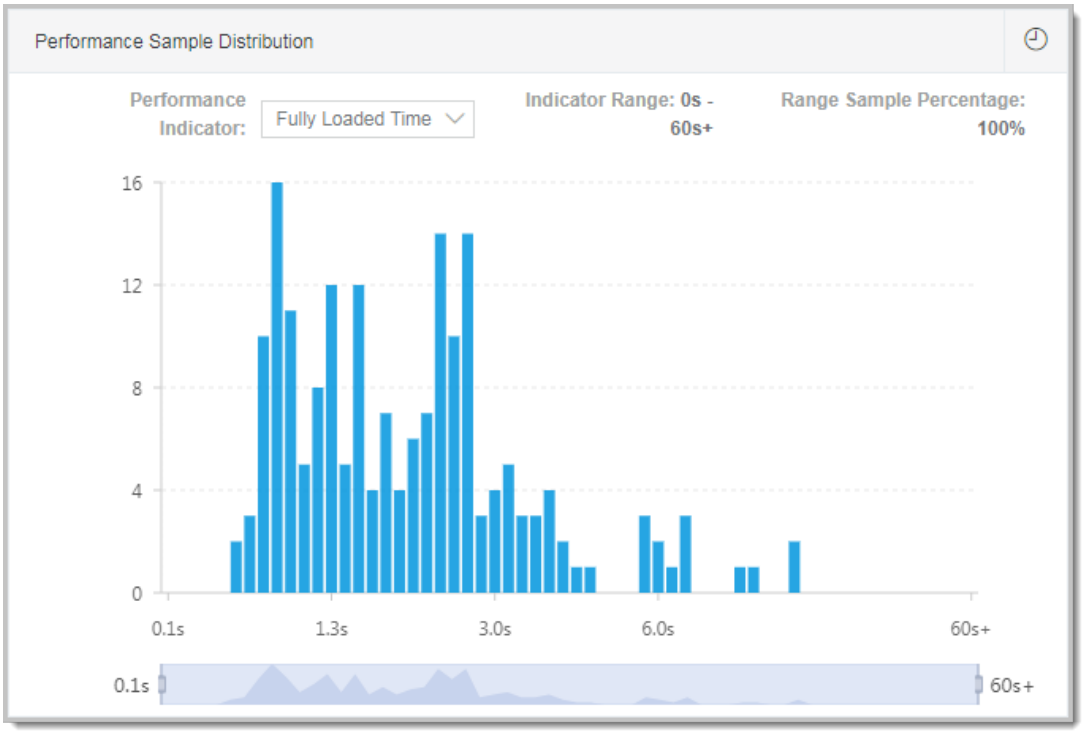
This section shows the distribution of the page performance.

On the **Performance Stacked Area Chart** tab, a stack line chart is displayed. This chart uses time as the horizontal axis. You can view the distribution of performance metrics at different points in time.

Performance Stacked Area Chart tab

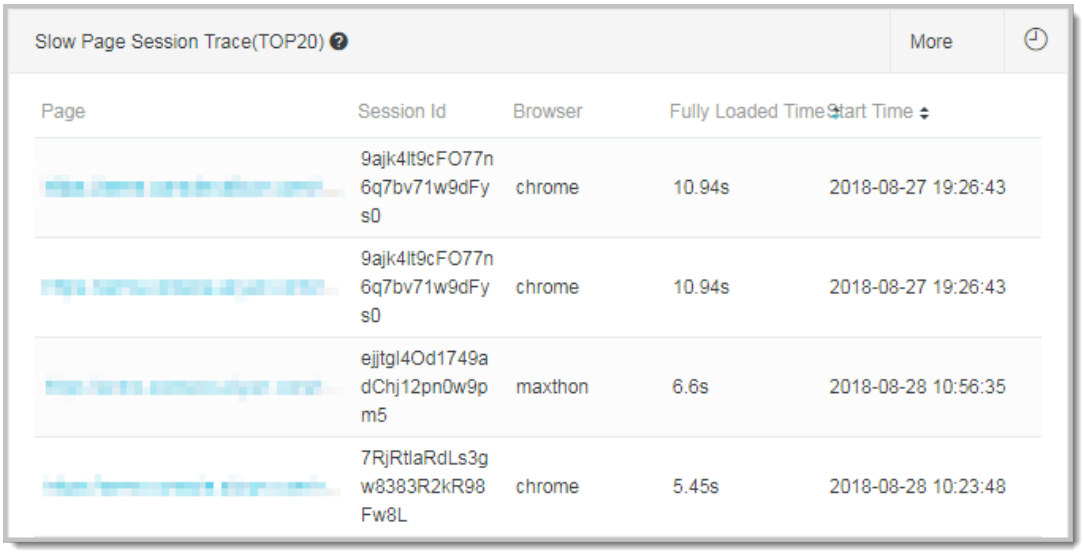
On the **Performance Sample Distribution** tab, you can view the sample proportion of the page loading time in the specified time range. For example, how many pages can be opened within 1 second, or the proportion of samples of long-tail users.

Performance Sample Distribution tab



Slow Page Session Trace section

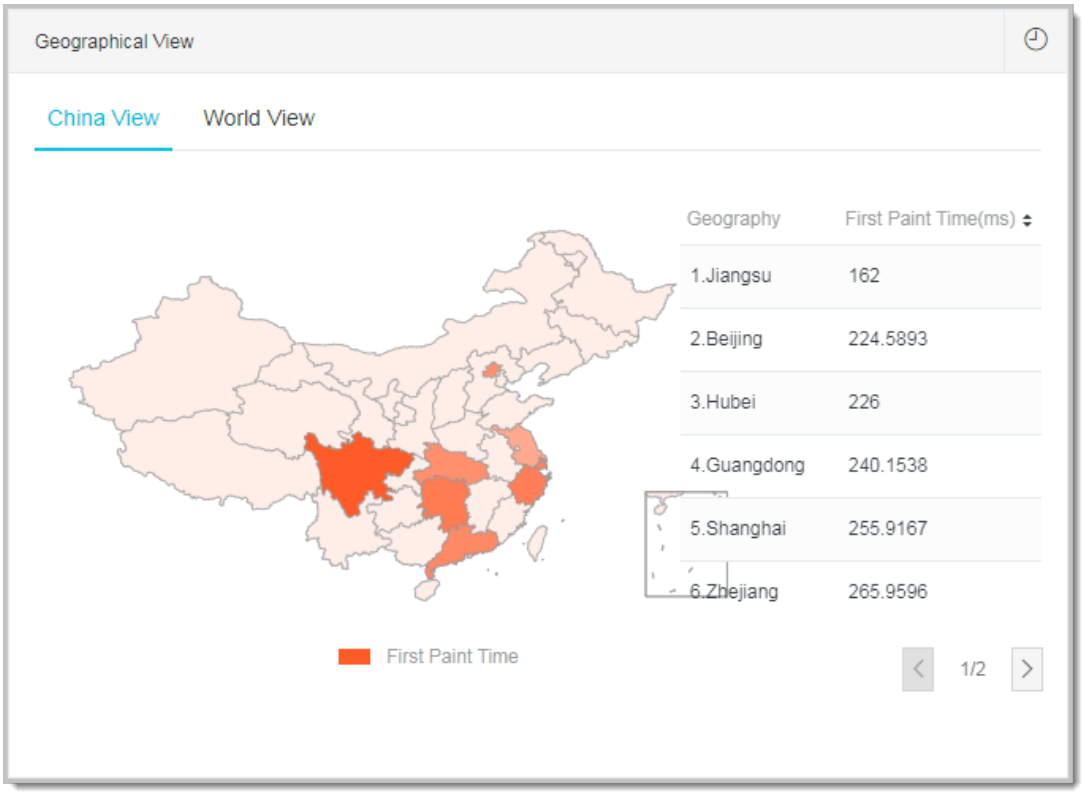
The Slow Page Session Trace section can provide a performance waterfall chart to display static resource loading during the page loading process. This section allows you to view the status of page resource loading based on page performance data. This allows you to identify and handle the performance bottlenecks at the earliest opportunity. For more information, see Session tracing.



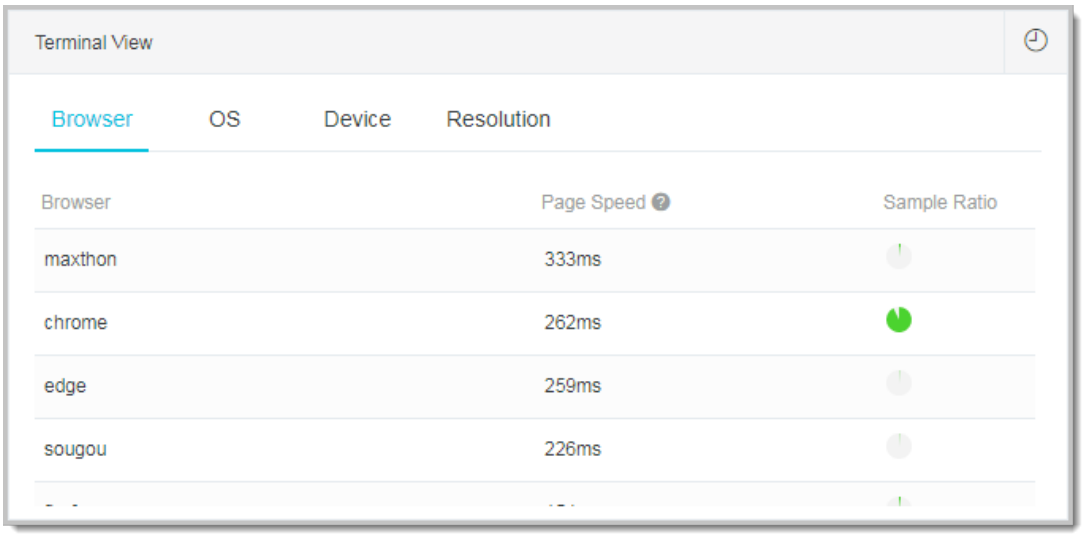
Page loading distribution

A page is loaded on the browser of a user. The loading time is related to specific factors. These factors include the geographical location, network condition, browser, and carrier. Therefore, the page speed feature provides the statistics of geographical distribution, terminal distribution, network distribution, and version distribution. This allows you to identify performance bottlenecks.

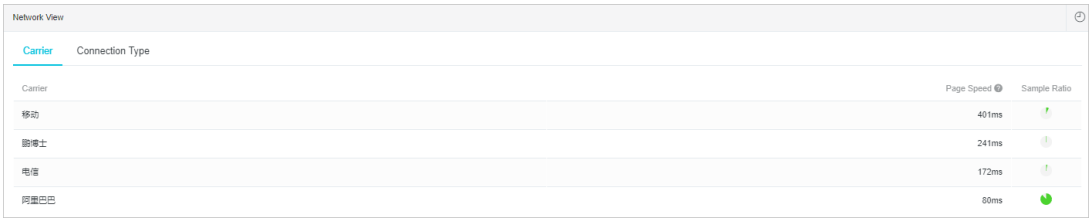
Geographical View section



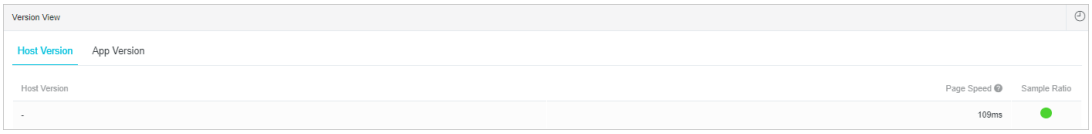
Terminal View section



Network View section



Version View section



Description of performance metrics

Key performance metrics of web pages

Reported field	Description	Calculation method	Remarks
First Meaningful Paint (FMP)	The First Meaningful Paint.	For more information, visit Technical solution of FMP.	None
fpt	The FPT.	responseEnd - fetchStart	This field indicates the duration from the time when a request is initiated to the time when the browser begins to parse the bytes of the first batch of HTML documents.
tti	The time to interact (TTI).	domInteractive - fetchStart	This field indicates the time when the browser starts to load resources after it resolves all HTML documents and constructs the DOM.
ready	The time it takes to complete HTML loading, which is the time it takes to construct the DOM.	domContentLoadEventEnd - fetchStart	If a JavaScript (JS) script is executed synchronously on the page, the execution time of the JS script can be calculated based on the following formula: Execution time of the JS script = ready - tti.
load	The time it takes to completely load the page.	loadEventStart - fetchStart	This field can be calculated based on the following formula: load = fpt + dom + (ready - tti) + res.
firstbyte	The time it takes to generate the first response packet.	responseStart - domainLookupStart	None

Fields that describe the amount of time consumed in each phase

Reported field	Description	Calculation method	Remarks

Reported field	Description	Calculation method	Remarks
dns	The amount of time consumed for DNS query.	domainLookupEnd - domainLookupStart	None
tcp	The amount of time consumed for TCP connection.	connectEnd - connectStart	None
ttfb	The time to first byte (TTFB), which indicates the amount of time it takes to respond to a request.	responseStart - requestStart	TTFB can be calculated in multiple ways. For more information about how TTFB is calculated in ARMS, visit Google development definition.
trans	The amount of time consumed for data transmission.	responseEnd - responseStart	None
dom	The amount of time consumed for document object model (DOM) resolution.	domInteractive - responseEnd	None
res	The amount of time consumed for resource loading.	loadEventStart - domContentLoadedEventEnd	This field indicates the amount of time consumed for synchronously loading resources on the page.
ssl	The amount of time consumed for SSL connection.	connectEnd - secureConnectionStart	This field is valid only when HTTPS is used to transmit data.

3.3. Session tracing

If you want to reproduce the situation where an error occurred based on the context to identify the cause of the error, you can use the session tracing feature of Application Real-Time Monitoring Service (ARMS) Browser Monitoring. This feature implements distributed tracing based on a username or user ID to show a comprehensive list of user behavior traces, including page loading, API calls, JS errors, and user operations. This helps identify and analyze the cause of an error.

Procedure

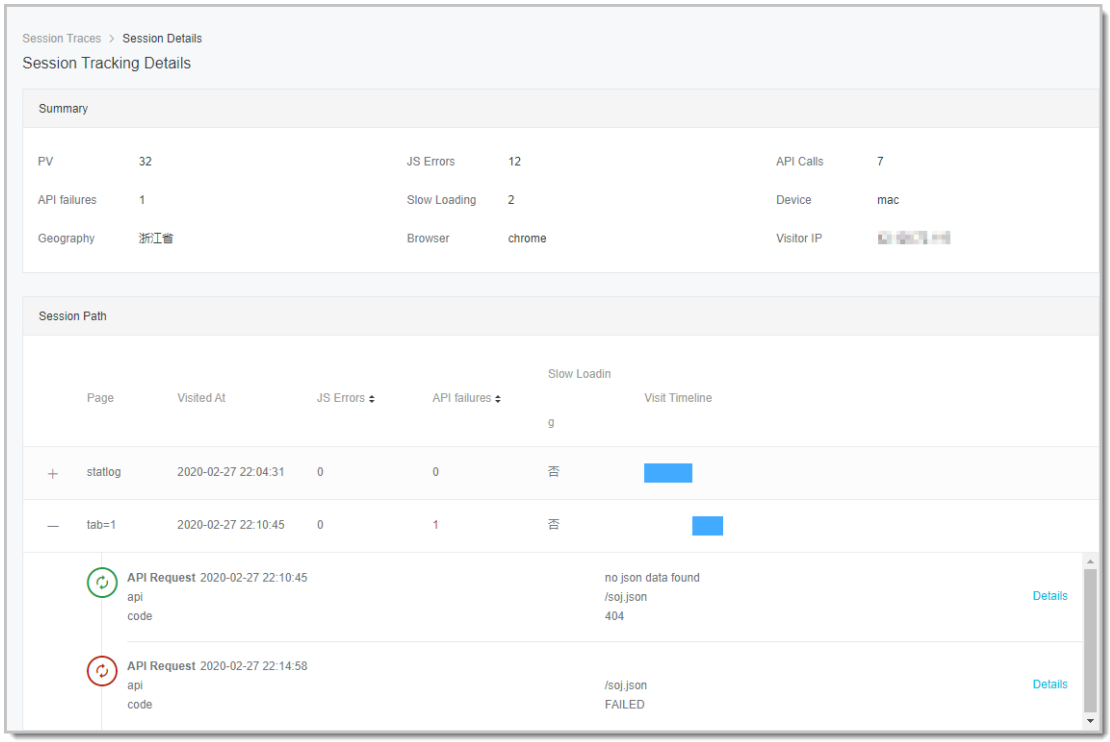
1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the application.
3. In the left-side navigation pane, choose **Application > Session Traces**.

View session details

1. (Optional)On the **Session Traces** page, set **User ID** or **Username** and click **Search** to search for the corresponding session.

> **Note** For information about how to set a username, see SDK reference.

2. In the Session List section, find the session and click its ID in the **Session ID** column. The **Session Details** page appears.
3. In the **Overview** section, view basic information of the session such as the username, user ID, session ID, number of page views (PVs), number of JS errors, number of API calls, number of API failures, number of slow loads, device, region, browser, IP address, and network system.
4. In the **Session Path** section, view the access path of the user.
   i. Click the **+** icon on the left of the page to show the user behavior traces.



   ii. Click **Details** on the right of a user behavior to view the details such as details about API calls, slow loads, and JS errors.

Other methods to view session details

- In the left-side navigation pane, choose **Application > Page Speed**. On the page that appears, click a session ID in the **Session ID** column of the **Slow Page Session Trace (TOP20)** section.
- In the left-side navigation pane, choose **Application > JS Error Diagnosis**. On the **Frequent Errors** tab, click **Diagnose** in the **Actions** column that corresponds to an error. On the page that appears, click **View Session** in the **User behavior backtracking** section.
- In the left-side navigation pane, choose **Application > API Details**. On the **API Requests** tab, click the number of errors in the **Error Number** column. On the page that appears, click **View Session** in the **Network Request Information** section.
- In the left-side navigation pane, choose **Application > View Details**. On the **All Logs** tab, click **View Session** in the **Actions** column in the Log List section.

Related information

- Slow session tracing

3.4. JS error diagnostics

The Browser Monitoring module of Application Real-Time Monitoring Service (ARMS) provides the JavaScript (JS) error diagnostics feature. You can view the basic information and distribution of JS errors and backtrack user behaviors. This feature helps you identify and fix errors.

Portal

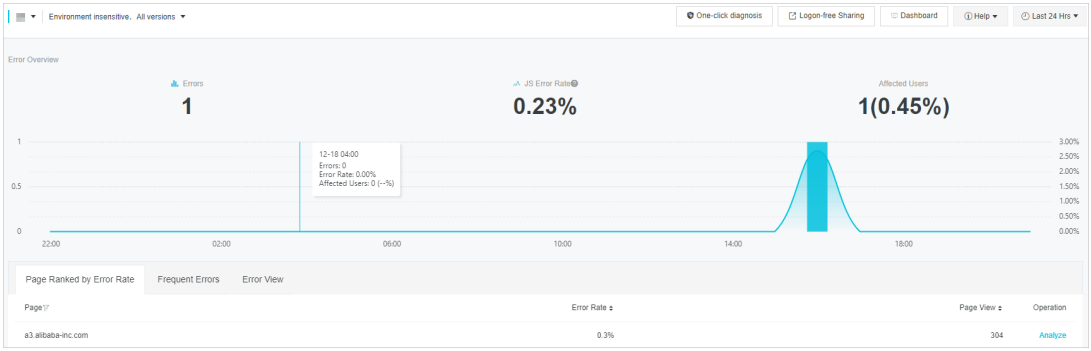
1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the application that you want to view.
3. On the page that appears, choose **Application > JS Error Diagnosis** in the left-side navigation pane.
   On the **JS error diagnosis** page, you can set the required period in the upper-right corner.

View the error overview of the application

The **Error Overview** section shows the statistics and trends of JS errors within the selected period, including the following metrics:

- **Errors**: the total number of JS errors that occurred within the selected period.
- **JS Error Rate**: the number of page views (PVs) with JS errors to the total PVs within the selected period.
- **Affected Users**: the quantity and percentage of users that are affected by JS errors.

Error overview of the application



In the **Error Overview** section, perform the following operations:

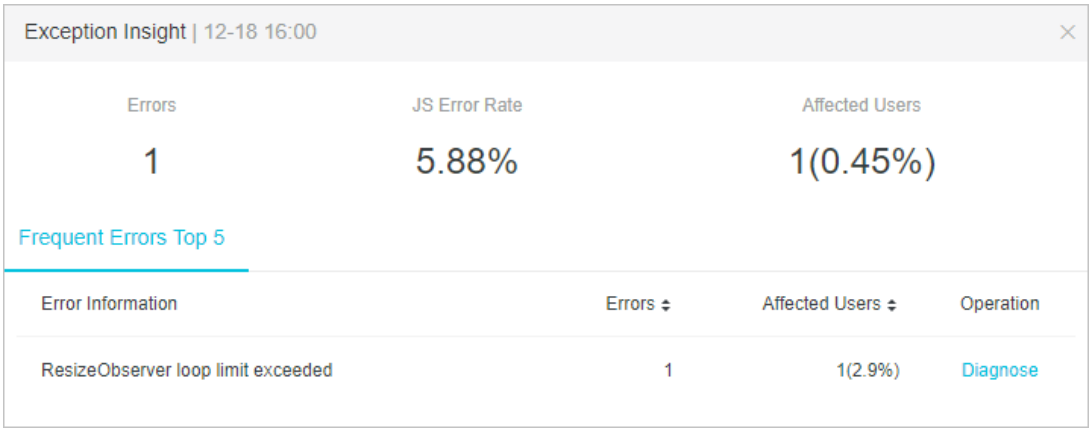
- Place the pointer over the curve. The number of errors, error rate, and number of affected users at the time point that correspond to a turning point of the curve appear in a floating manner.
- Place the pointer over a turning point of the curve. When the pointer changes to a hand shape, click the turning point. The **Exception Insight** dialog box appears. For more information, see View exception insight.
- In the curve section, hold down the left mouse button, drag the mouse to select an area, and then zoom in and view the selected part of the curve. Click **Reset Zoom** in the upper-right corner to restore the view.

> **Note** On the **JS error diagnosis** page, the application-specific error overview is displayed by default in the **Error Overview** section. On the **Page Ranked by Error Rate** tab or the **Page Error Rate Top 5** tab in the **Exception Insight** dialog box, click **Analyze**. The overview information of the corresponding tab appears.

View exception insight

In the **Exception Insight** dialog box, information about the JS errors at a specific time point appears, including the following metrics:

- **Errors**: the total number of JS errors at the specific time point.
- **JS Error Rate**: the number of PVs with JS errors to the total PVs at the specific time point.
- **Affected Users**: the quantity and percentage of users that are affected by JS errors.
- **Frequent Errors Top 5**: the top 5 JS errors that occur the most frequently at the specific time point, including the information about the JS errors, the number of errors, and the number of affected users that are captured by ARMS.
- **Page Error Rate Top 5**: the top 5 pages that have the five highest JS error rates at the specific time point, including the names of pages with JS errors, the JS error rates of the pages, and the number of PVs.



In the **Exception Insight** dialog box, you can perform the following operations:

- Click the **Frequent Errors Top 5** tab, and then click **Diagnose** in the **Operation** column. The **JS error details** page appears. For more information, see View error details.
- Click the **Page Error Rate Top 5** tab. On the tab that appears, click **Analyze** in the **Operation** column that corresponds to the page you want to view. The error overview of the page appears.

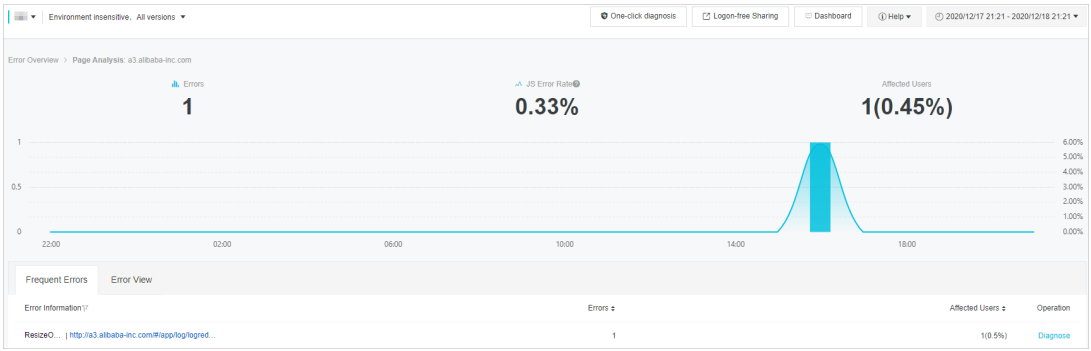
View pages ranked by error rate

On the **Page Ranked by Error Rate** tab, pages are ranked by JS error rate within the selected period in descending order, including the following metrics:

- **Page**: the page on which JS errors occur.
- **Error Rate**: the number of PVs with JS errors on the page to the total PVs within the selected period.
- **Page View**: the number of views of the page.

Click **Analyze** in the **Operation** column. The error overview of the page appears.

Error overview of the page



View frequent errors

On the **Frequent Errors** tab, JS errors are ranked by the number of occurrences within the selected period in descending order, including the following metrics:

- **Error Information**: the JS error information captured by ARMS.
- **Page**: the page on which the JS error occurs.
- **Errors**: the number of occurrences of the JS error.
- **Affected Users**: the quantity and percentage of users that are affected by the JS error.

Click **Diagnose** in the **Operation** column to go to the **Error Detail** tab. For more information, see View error details.

> **Note** On the **JS error diagnosis** page, the application-specific JS errors are displayed by default on the **Frequent Errors** tab. On the **Page Ranked by Error Rate** tab or the **Page Error Rate Top 5** tab in the **Exception Insight** dialog box, click **Analyze**. The information about the JS errors on the corresponding pages appears.

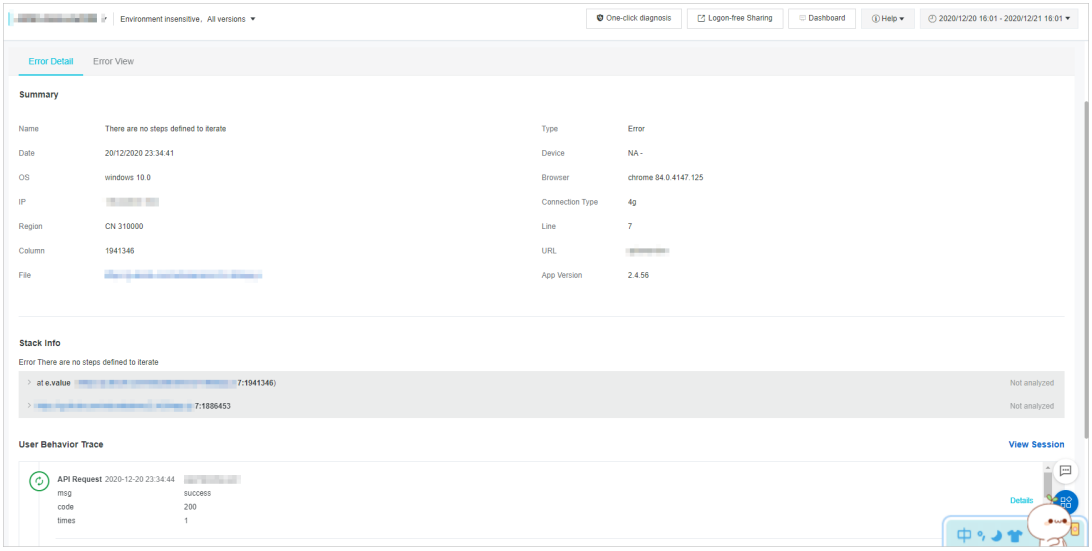
View error details

The following information is displayed on the **Error Detail** tab:

- **Summary**
  - **Name**
  - **Type**
  - **Date**: the time when the JS error is detected.
  - **Device**
  - **OS**
  - **Browser**
  - **IP**
  - **Connection Type**
  - **Region**
  - **Line**
  - **Column**
  - **URL**
  - **File**: the path of the file where the JS error occurs.
  - **App Version**
- **Stack Info**: the information related to the location of the JS error.

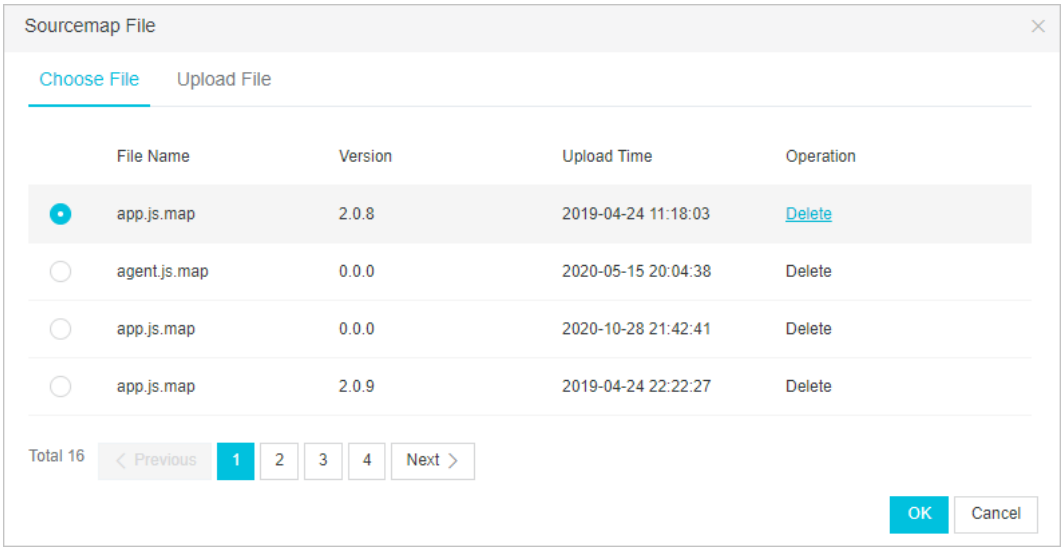
- **User Behavior Trace**: the user behavior trace, which is used to backtrack the error.

  JS error details page



On the **Error Detail** tab, you can perform the following operations:

- To specify the exact location of the JS error, click the triangle icon on the left side of a stack in the **Stack Info** section to show the line. Click **Choose Sourcemap**. In the **Sourcemap File** dialog box, select an existing source map file or upload a new source map file, and then click **OK**.

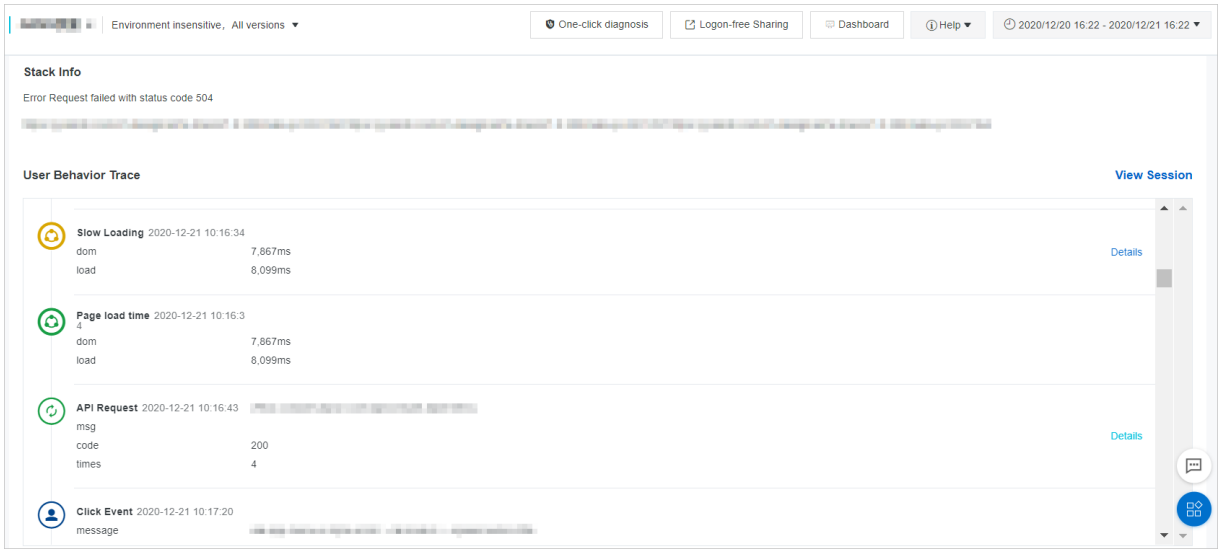


  ARMS uses the source map file to retrieve the exact location of the JS error.

- To view the user behavior trace, go to the Backtrack user behaviors section.
- To view the distribution of the JS error, click the **Error View** tab.

Backtrack user behaviors

On the **Error Detail** tab, the **User Behavior Trace** section displays the user behavior trace to help backtrack the error.

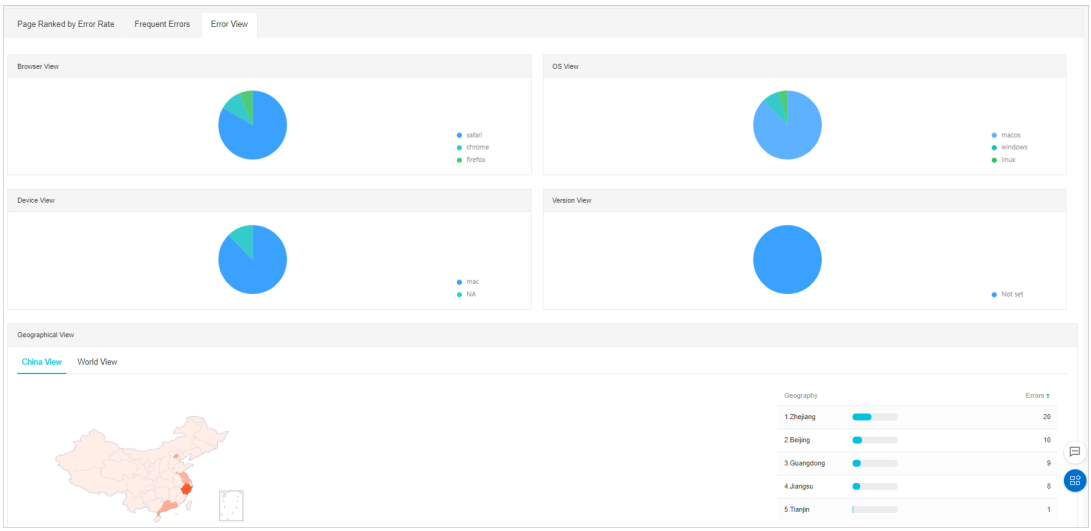


View error distribution

The **Error View** tab of the JS error diagnosis page displays the distribution of a specific JS error, including the following metrics:

- **Time View**: Only page-specific error distribution is shown in this section.
- **Browser View**
- **OS View**
- **Device View**
- **Version View**
- **Geographical View**: Statistics are collected by province, municipality, or autonomous region on the China View tab, and are collected by country or region on the World View tab.

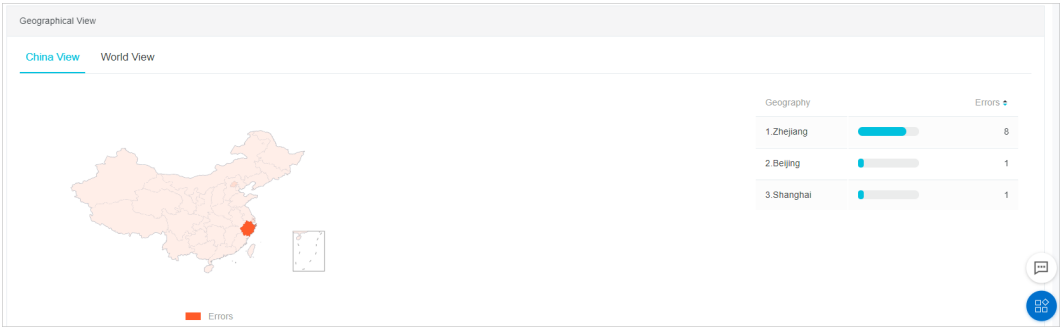
JS error view page



On the **Error View** tab, you can perform the following operations:

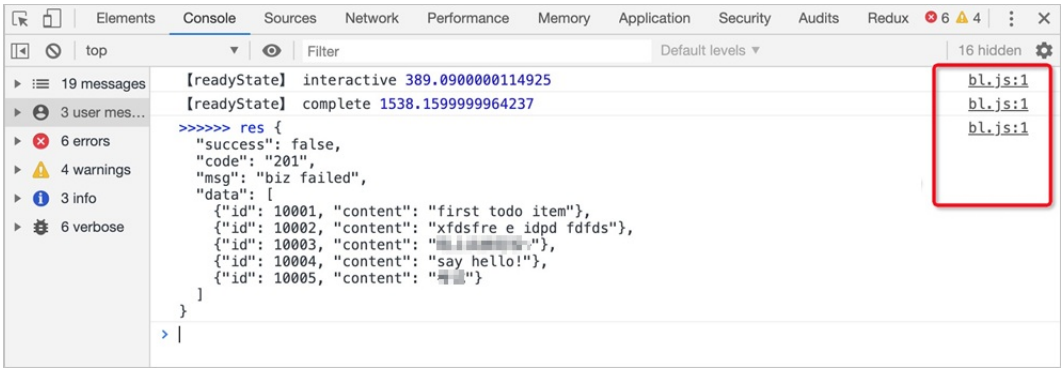
- In the **Time View** section, move the pointer over the distribution map to view the number of errors.
- In the **Browser View**, **OS View**, **Device View**, and **Version View** sections, move the pointer over the distribution map to view the number and percentage of errors.
- On the **China View** or **World View** tab in the **Geographical View** section, click the arrows next to

the **Errors** column name in the table on the right side to switch between ascending and descending orders.



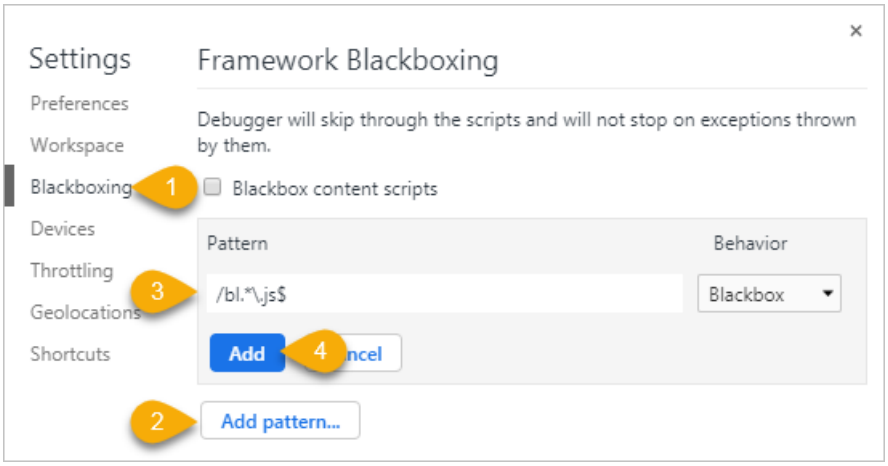
FAQ

- How do I enable or disable user behavior backtracking?
  By default, this feature is enabled. To disable the feature, add the behavior: false SDK configuration item to config. For more information about SDK configuration items, see SDK reference.
- After user behavior backtracking is enabled, the error in the ARMS SDK code bl.js instead of the source code is located based on the printed information in console.log. How do I solve this problem?



The reason is that ARMS rewrites log of the console object to monitor the content printed by the browser console. Solutions:

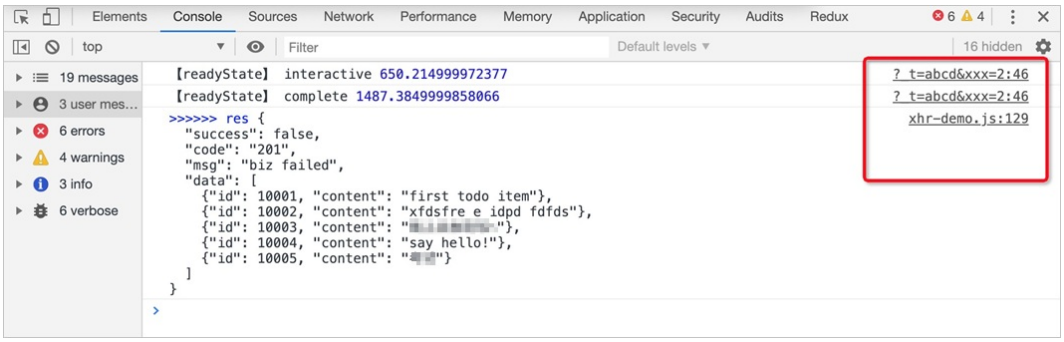
- Method 1 (recommended): Set blackboxing for the Chrome browser.
  a. Open the Chrome browser, press Ctrl+Shift+I to access the developer tools panel, and then click the Settings icon.
  b. In the left-side navigation pane of the **Settings** panel, click **Blackboxing**. On the page that appears, click **Add pattern**. In the **Pattern** field, enter */bl. *\.js$*. Then, click **Add**.



- Method 2: Use the behavior: false SDK configuration item to disable user behavior backtracking.

```
<script>
  ! (function ( c , b, d, a ) {
    c [a] || ( c[a] = {});
    c [a].config = {
      pid: "xxxxx",
      imgUrl: "https://arms-retcode.aliyuncs.com/r.png?",
      sendResource: true,
      enableLinkTrace: true,
      behavior: false
    };
    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 the preceding handling, the error in the source code can be located based on the printed information in console.log.



Related information

- SDK reference

3.5. API request

The API request feature provides the call information about each API operation in an application, which includes the call success rate, response, and average response time of successful or failed API calls.

Feature description

The API request feature of ARMS browser monitoring shows the following information:

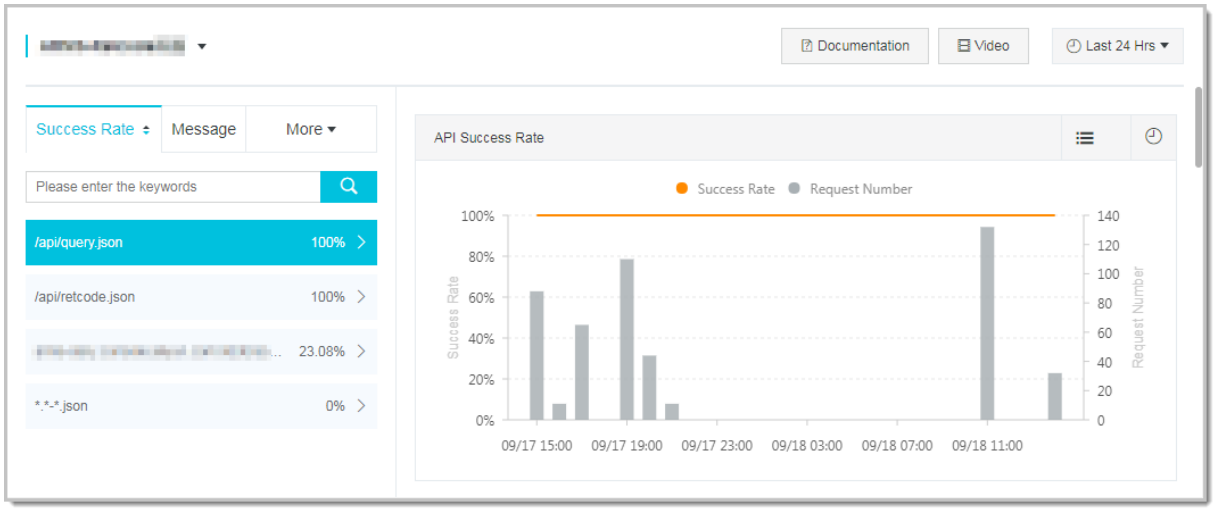
- Call success rate of each API operation
- API response
- Average response time of successful API calls
- Average response time of failed API calls

This feature also shows the distribution of the preceding statistics in the following dimensions:

- Geographical location
- Browser
- Operating system
- Device
- Resolution
- Version

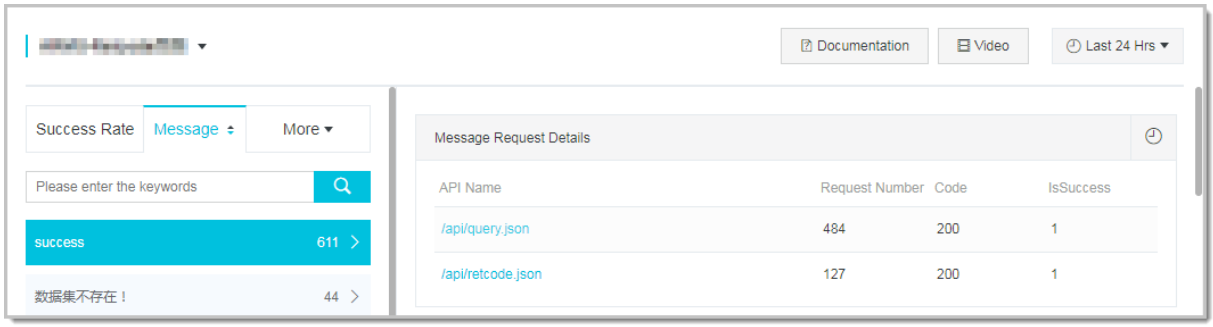
API success rate

The **Success Rate** tab on the left ranks the success rates of API calls. The **API Success Rate** section on the right shows the curve of the call volume and call success rate of the API operation selected on the left within a specified time range. The **API Link Trace (TOP 20)** section on the right shows the traces of the top 20 failed and successful calls to the API operation. You can click **Link Trace** and **View Details** in the **Actions** column to check the trace and view details.



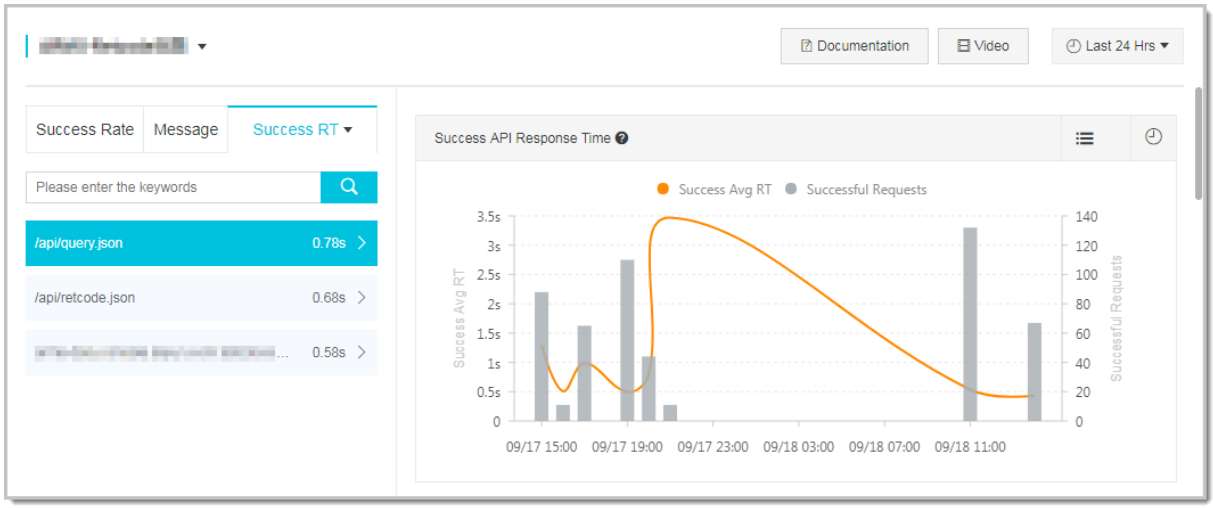
API response

The **Message** tab on the left ranks API responses. The **Message Request Details** section on the right shows the API calls of the API response selected on the left. These API calls are listed by call volume in descending order.



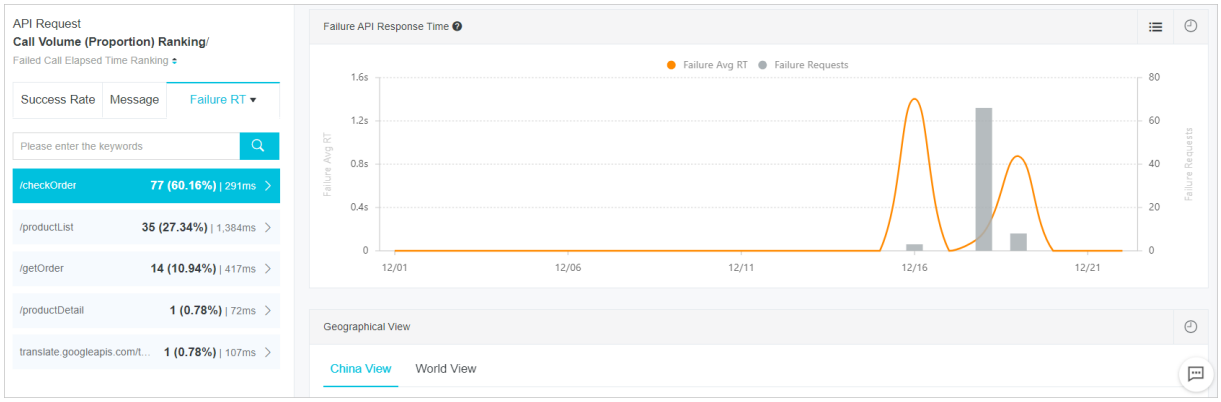
Response time for successful API calls

The **Success RT** tab on the left shows the average response time for successful API calls. The **Success API Response Time** section on the right shows two items: the average response time for successful API calls that is indicated by the curve and the number of successful calls that is shown in the bar chart.



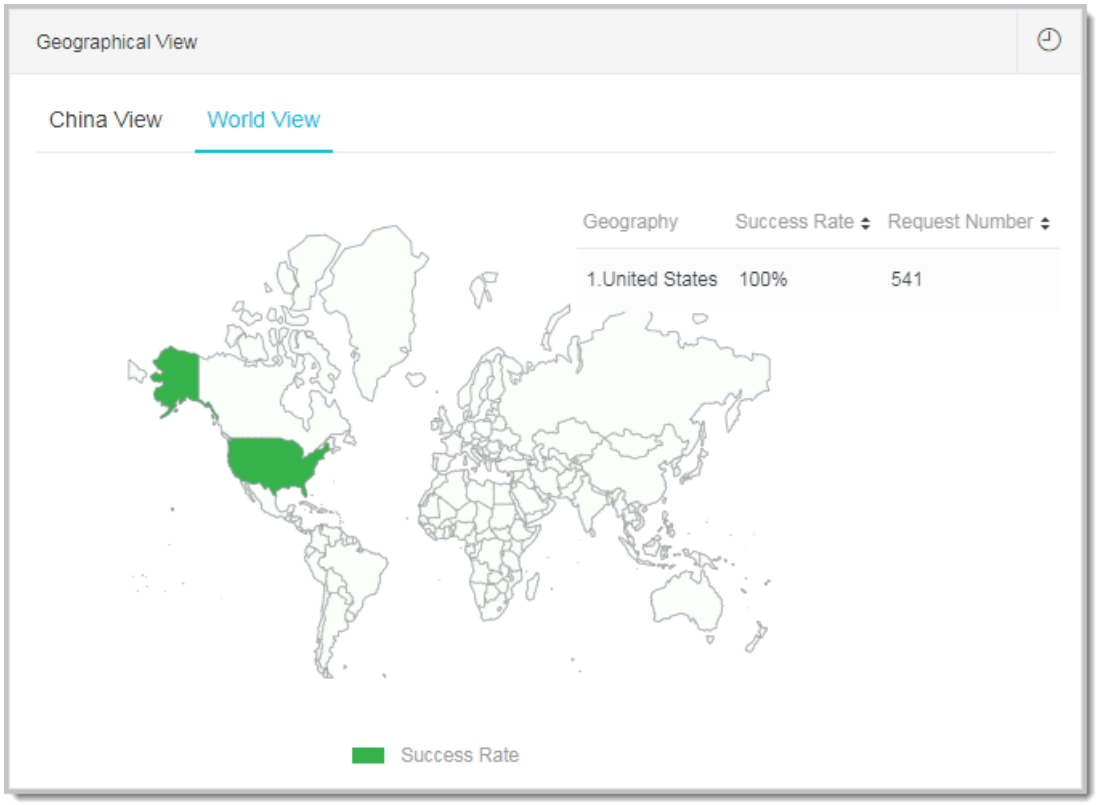
Response time for failed API calls

The **Failure RT** tab on the left shows the average response time for failed API calls. The **Failure API Response Time** section on the right shows two items: the average response time for failed API calls that is indicated by the curve and the number of failed calls that is shown in the bar chart.



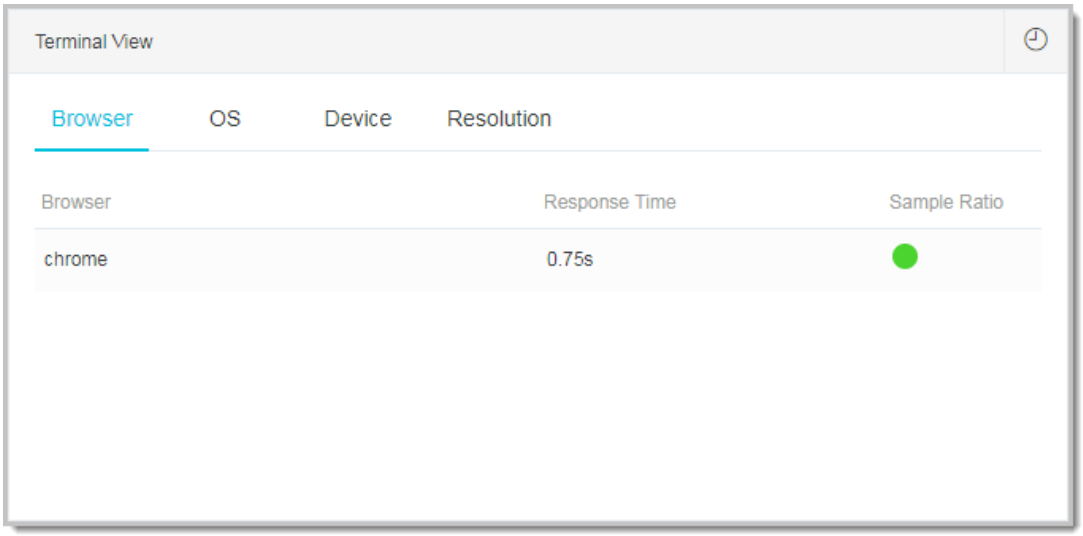
Geographical view

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 shows statistics by province in China, whereas World View shows 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.



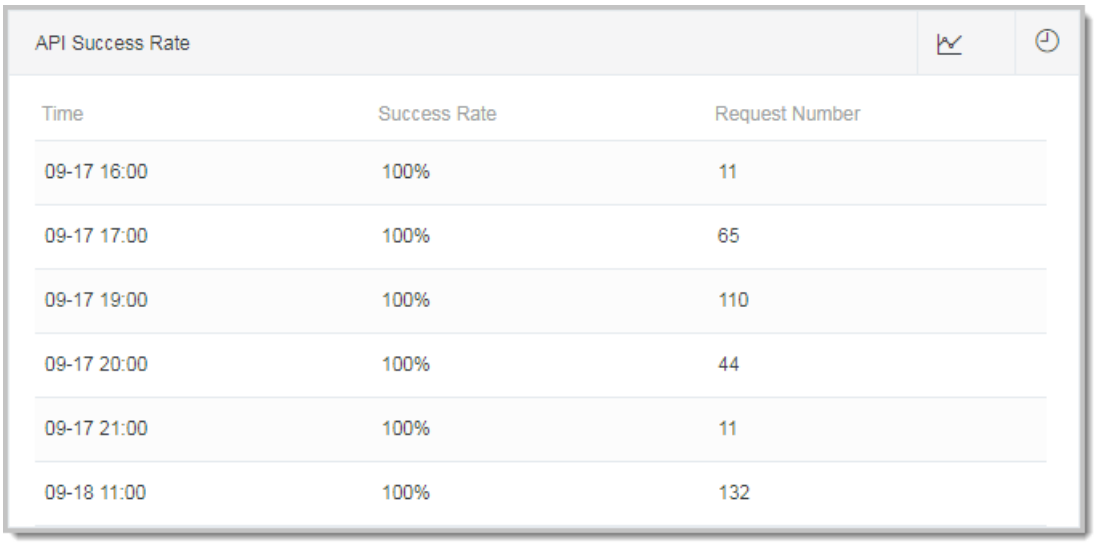
Version view

On the **Version View** page, you can view the host version and application version of each application.

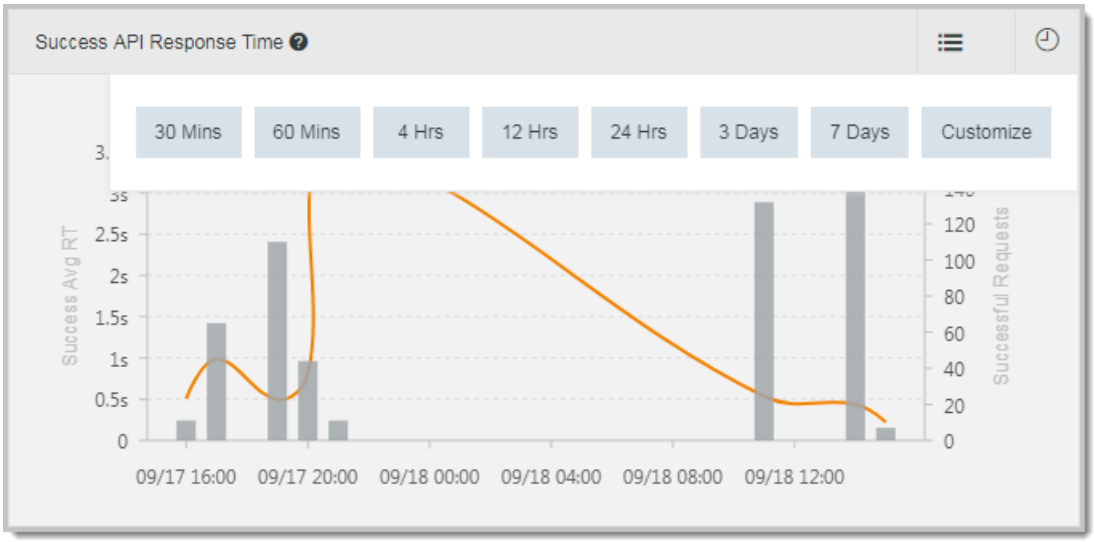
Common operations

In the API Request 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 chart and table views.



- In the right-side details section, click the clock icon in the upper-right corner to specify a time range.



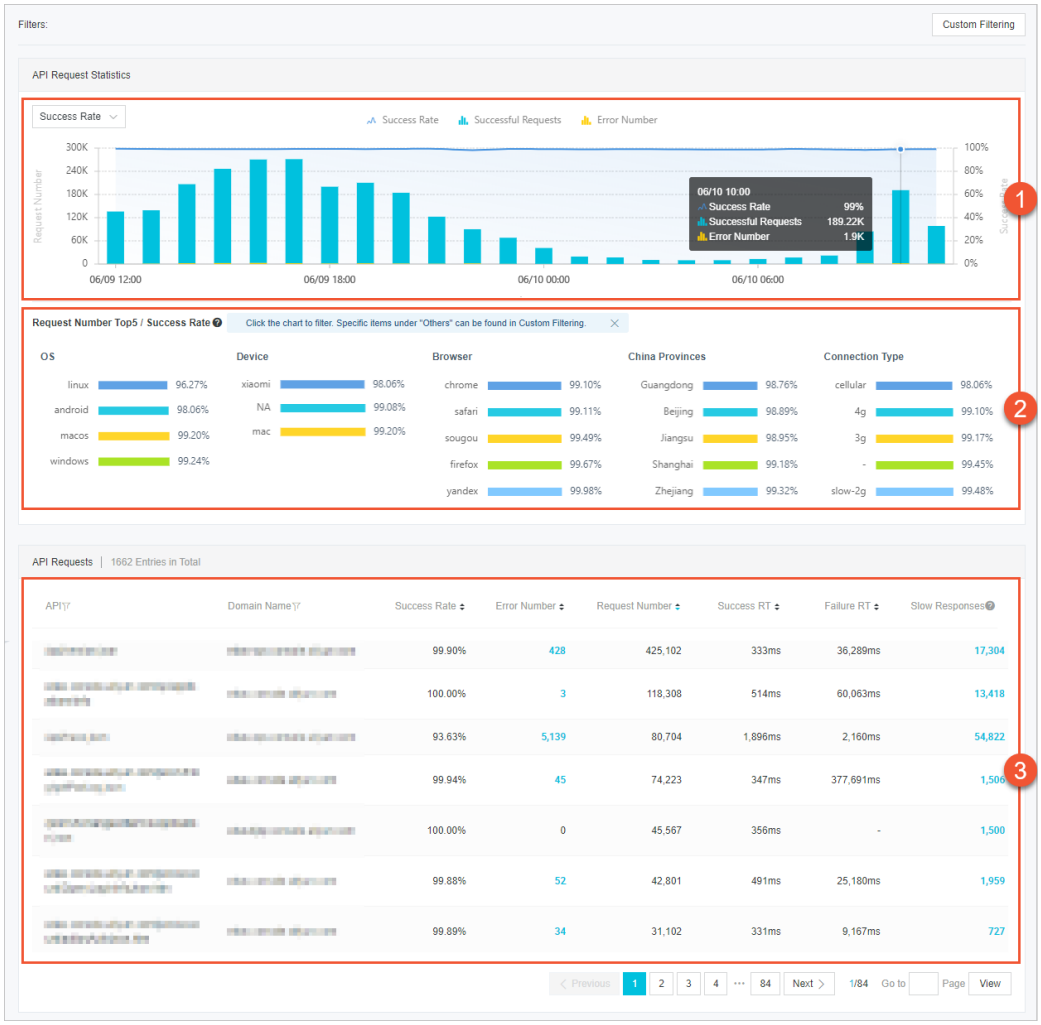
3.6. API details

The API details page of Application Real-Time Monitoring Service (ARMS) Browser Monitoring shows the success rate of all API requests, average response time (RT) of successful calls, average RT of failed calls, number of slow responses, and number of errors in a specified period. This page also shows statistics for API requests initiated in five dimensions such as regions, domain names, and networks.

Portal

1. Log on to the ARMS console.
2. In the left-side navigation pane, click **Browser Monitoring**.
3. On the **Browser Monitoring** page, click the name of the required application.
4. In the left-side navigation pane, choose **Application > API Details** to go to the API Details page. The page is divided into three sections.

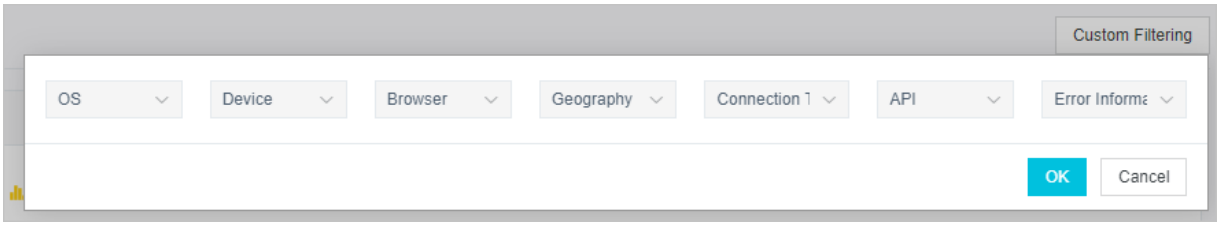
   API details



Filters

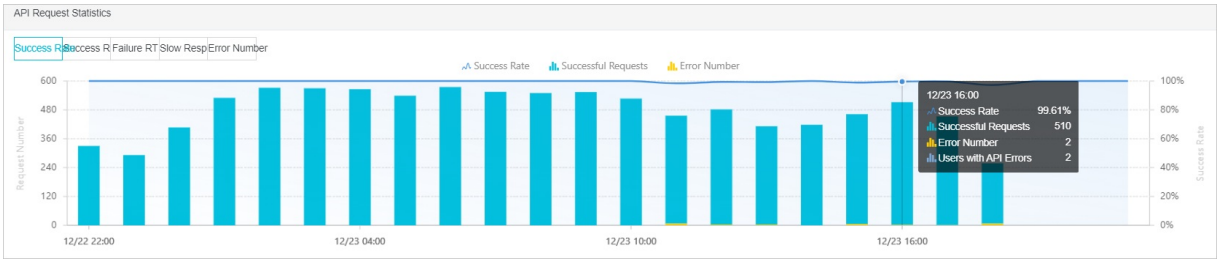
Set filters in different dimensions to view the corresponding API information. Valid values: **API**, **Return Message**, **HTTP Status Code**, **Page**, **Geography**, **OS**, **Domain Name**, **Connection Type**, **Device**, and **Browser**.

To delete a filter, move the pointer over the filter and click the icon. Click Reset to clear all filters.



Success rate

In the **API Request** section, select **API Success Rate** to view the success rate, successful requests, error number, and users with API errors.

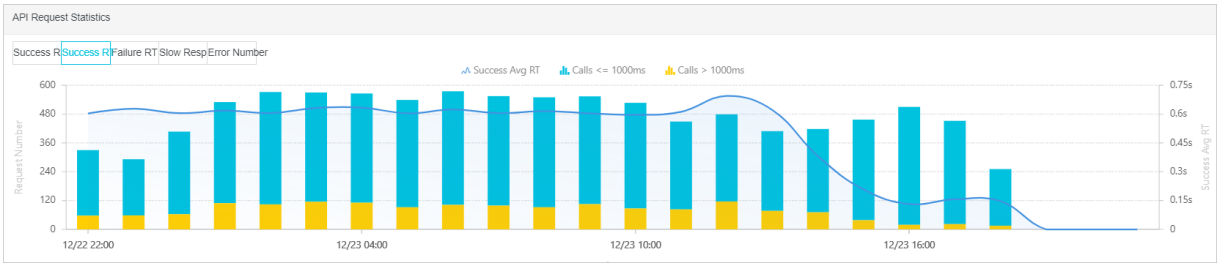


- Success rate: The success rate of all API requests to the specified application within a specified period of time is displayed in a line chart, which corresponds to the y-axis on the right side.
- Successful requests: The number of calls per hour within a specified period of time is displayed in a blue bar chart, which corresponds to the y-axis on the left side.
- Error number: The number of calls per hour within a specified period of time is displayed in a yellow bar chart, which corresponds to the y-axis on the left side.

Move the pointer over the bar chart to view the success rate, successful requests, error number, and users with API errors within a specified period of time.

Success RT or failure RT

In the **API Request** section, select **Success RT** or **Failure RT** to view the average RT of successful calls, average RT of failure calls, count of calls whose RT is smaller than or equal to 1,000ms, and count of calls whose RT is greater than 1,000ms.

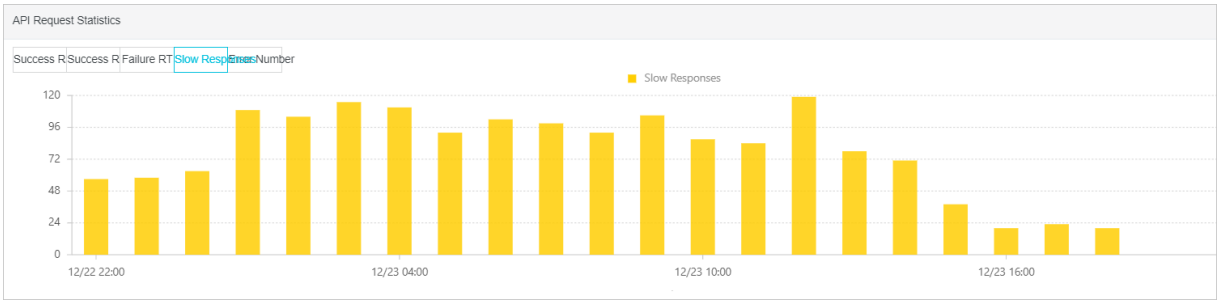


- Success Avg RT or failure Avg RT: The average success RT or failure RT of all API calls in an application within a specified period of time is displayed in a line chart, which corresponds to the y-axis on the right side.
- Calls <= 1000ms: The total number of successful calls or failed calls smaller than or equal to 1000 ms per hour is displayed in a blue bar chart, which corresponds to the y-axis on the left side.
- Calls > 1000ms: The total number of successful calls or failed calls greater than 1000 ms per hour is displayed in a yellow bar chart, which corresponds to the y-axis on the left side.

Slow responses

> **Note** Slow responses refer to the number of calls that take more than 1000ms.

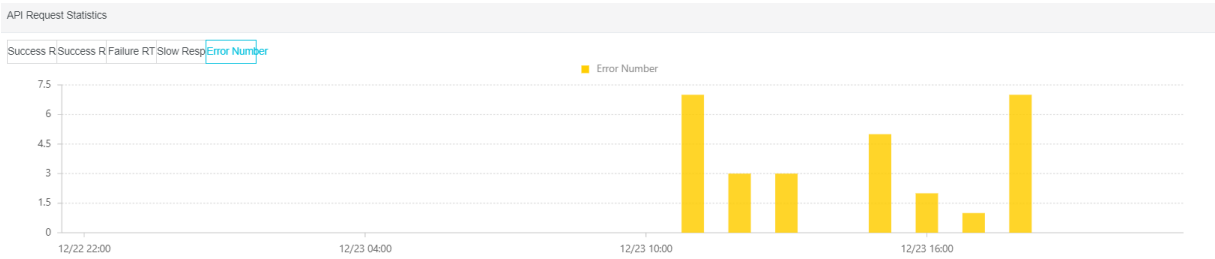
In the **API Request** section, select **Slow Responses** to view the slow responses.



Slow responses: The number of all API requests within a specified period of time is displayed in a bar chart, which corresponds to the y-axis on the left side.

Error number

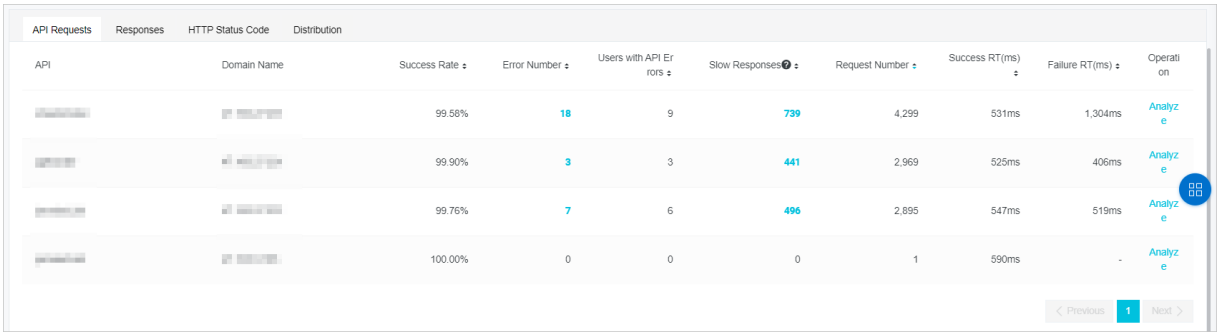
In the **API Request** section, select **Error Number** to view the number of failed requests.



Error Number: The number of all API errors within a specified period of time is displayed in a bar chart, which corresponds to the y-axis on the left side.

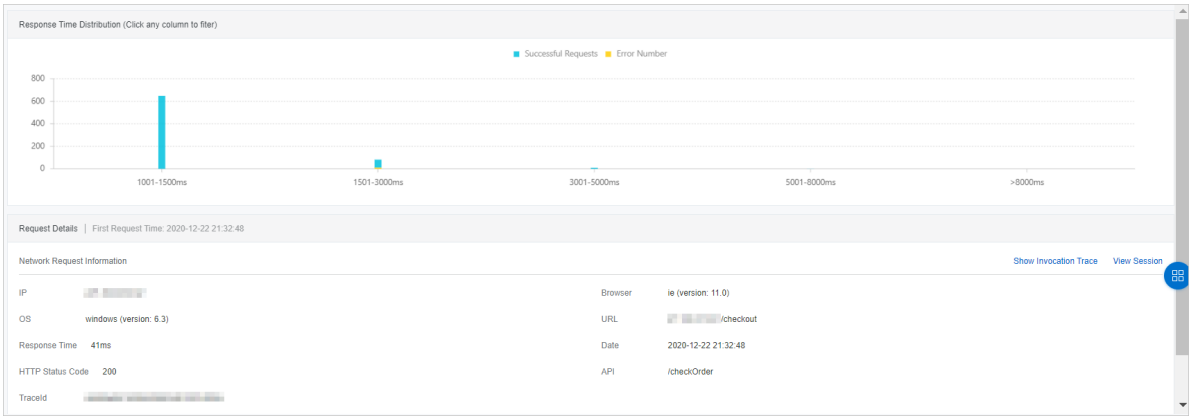
API requests

Select **API Requests** in the section ③ to view the requests of each API within a specified period of time.

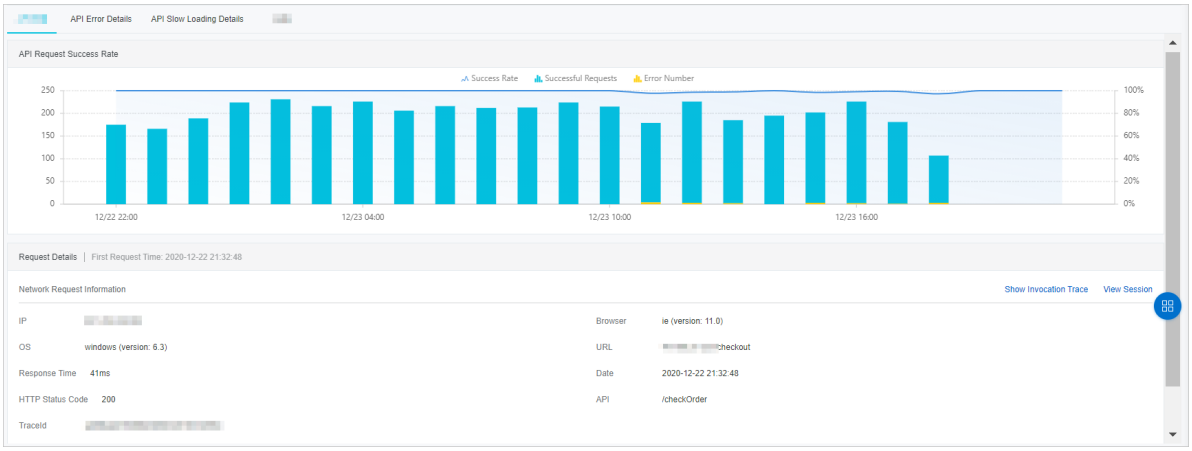


- Click the ▲▼ icon to the right of a property in the first row of the list to sort the APIs.
- Move the pointer over an API alias in the **API** column. Click the icon to modify the alias of the API.

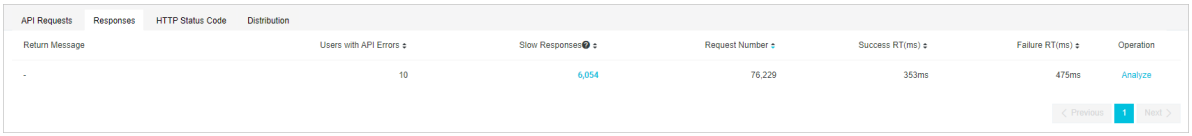
  All APIs are displayed by alias. Click **Set To Search Value** to set the API to a filter.
- Click a number in the **Error Number** column to view the details and distribution of errors within a specified period of time. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of failed requests. Click **View Session** to view the session trace of failed requests.



- Click a number in the **Slow Responses** column to display the details and distribution of slow responses within a specified period of time. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of slow responses. Click **View Session** to view the session trace of slow responses.



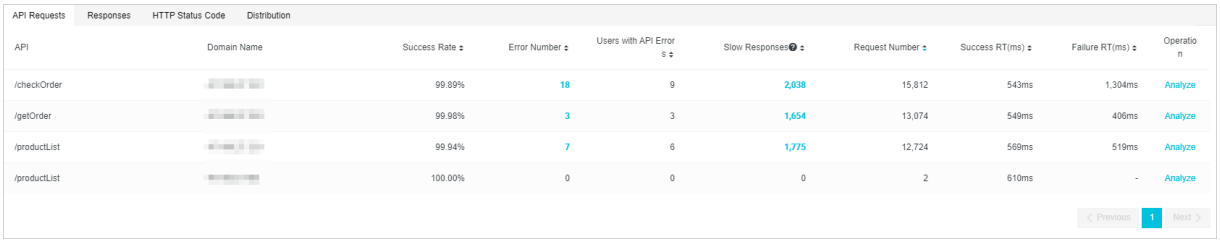
- Click **Analyze** in the **Operation** column to view the API details, API error details, API slow loading details, and distribution.



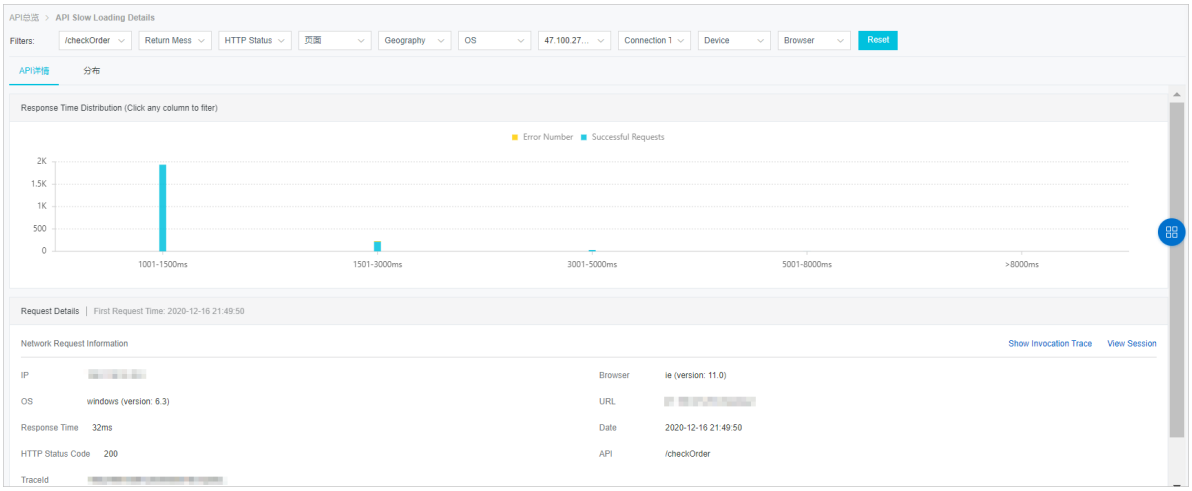
  - On the **API details** page, you can view the API success rate and request details. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of the API. Click **View Session**, you can view the session trace of the API.
  - On the **API Error Details** page, you can view the distribution of the error number and request details.
  - On the **API Slow Loading Details** page, you can view the response time distribution and network request information.
  - On the **Distribution** page, you can view the return message, HTTP status code, page, domain name, and geography. You can also view the proportions by OS, browser, device, or connection type.

Responses

Select **Responses** in the section ③ to view all response information of each API.

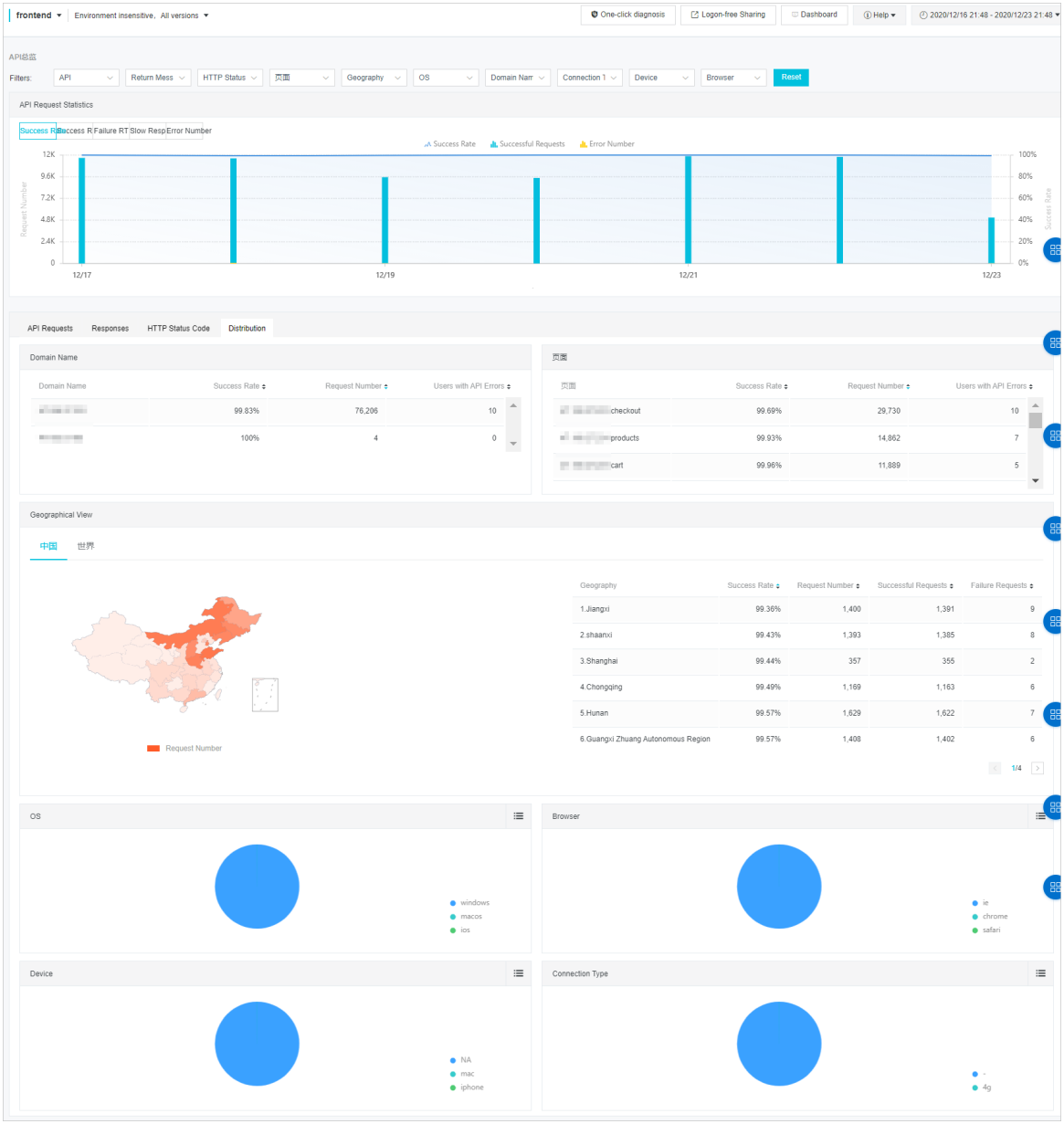


- Click the ⇕ icon to the right of a property in the first row of the list to sort the APIs.
- Move the pointer over a response in the **Responses** column. Click **Set To Search Value** to set the response to a filter.
- Click a number in the **Slow Responses** column to display the details and distribution of slow responses within a specified period of time. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of slow responses. Click **View Session** to view the session trace of slow responses.



- Click **Analyze** in the **Operation** column to view the API details, API error details, API slow loading details, and distribution.
  - On the **API details** page, you can view the API success rate and request details. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of the API. Click **View Session**, you can view the session trace of the API.
  - On the **API Error Details** page, you can view the distribution of the error number and request details.
  - On the **API Slow Loading Details** page, you can view the response time distribution and network request information.

- On the **Distribution** page, you can view the return message, HTTP status code, page, domain name, and geography. You can also view the proportions of the OS, browser, device, and connection type.

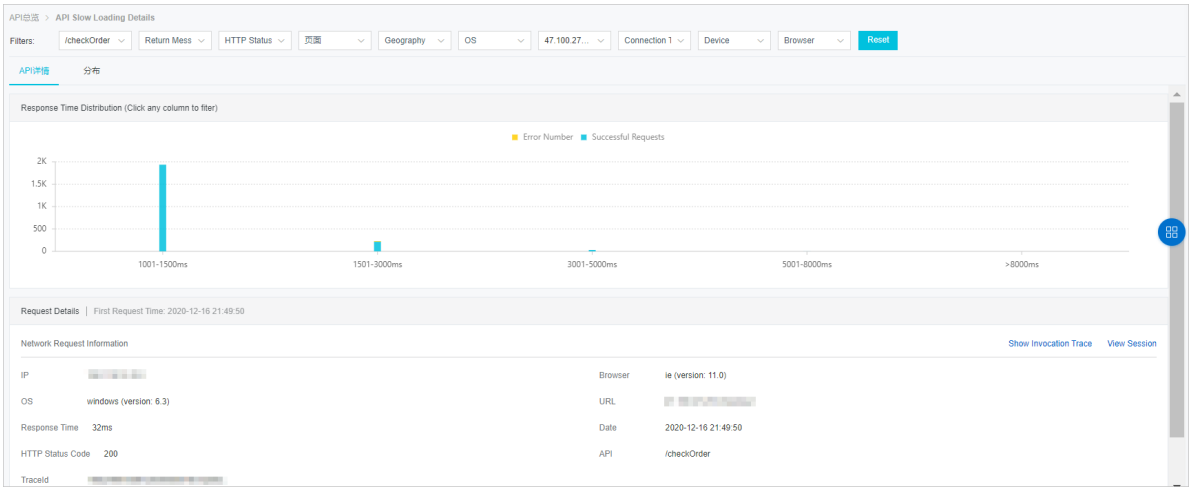


HTTP status code

Select an **HTTP Status Code** in section ③ to view all HTTP status codes.

- Click the ⇕ icon to the right of a property in the first row of the list to sort the APIs.
- Move the pointer over an HTTP status code in the **HTTP Status Code** column. Click **Set To Search Value** to set the HTTP status code to a filter.

- Click a number in the **Slow Responses** column to display the details and distribution of slow responses within a specified period of time. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of slow responses. Click **View Session** to view the session trace of slow responses.

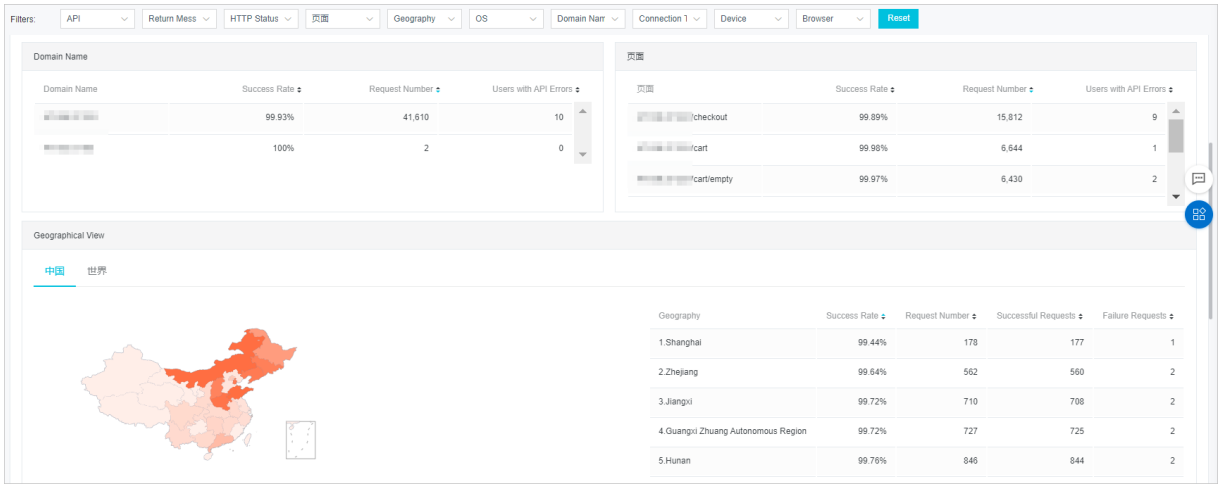


- Click **Analyze** in the **Operation** column to view the API details, API error details, API slow loading details, and distribution.
  - On the **API details** page, you can view the API success rate and request details. In the **Request Details** section, click **Show Invocation Trace** to view the call and business traces of the API. Click **View Session**, you can view the session trace of the API.
  - On the **API Error Details** page, you can view the distribution of the error number and request details.
  - On the **API Slow Lading Details** page, you can view the response time distribution and network request information.
  - On the **Distribution** page, you can view the return message, HTTP status code, page, domain name, and geography. You can also view the proportions of the OS, browser, device, and connection type.

Distribution

Select **Distribution** in the section ③. You can view the return message, HTTP status code, page, domain name, and geography. You can also view the proportions of the OS, browser, device, and connection type. Move the pointer over the **Domain Name** or **Page** column. Click **Set To Search Value** to set the domain name or page to a filter.

> **Note**
>
> - Versions numbers can be displayed only by OS or browser.
> - The pie chart of OS, browser, device, and connection type displays only the dimensional distribution of the first five items. You can click Toggle View in the upper-right corner of each section to view the proportion of all filters in this dimension.



3.7. Custom statistics

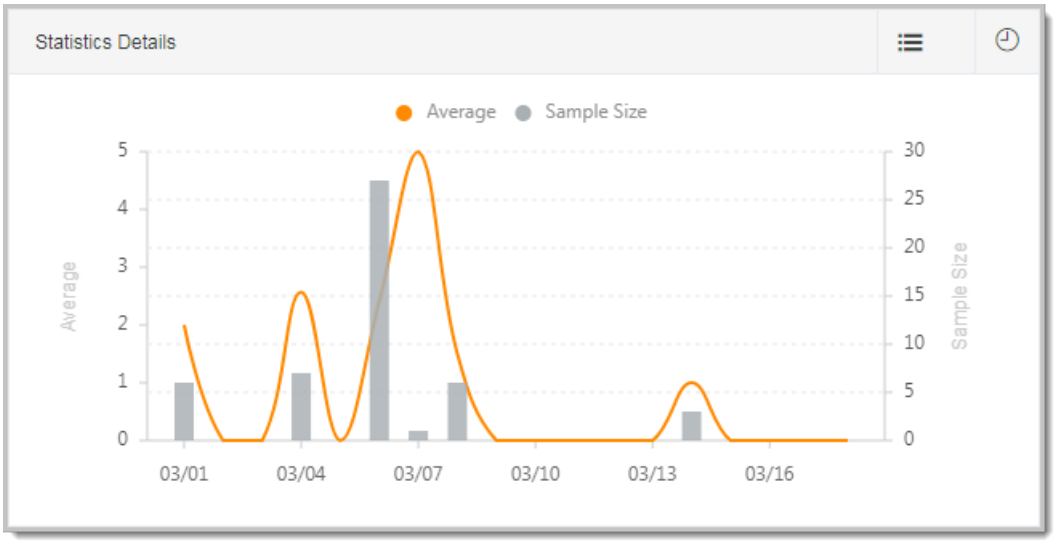
This topic describes the custom statistics feature provided by browser monitoring of Application Real-Time Monitoring Service (ARMS).

To help you monitor and collect lightweight business interaction behaviors, browser monitoring provides the following types of custom statistics:

- Sum statistics: calculates the total number of occurrences for some 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: calculates the average values for some events in the business, such as the average loading time of a module.

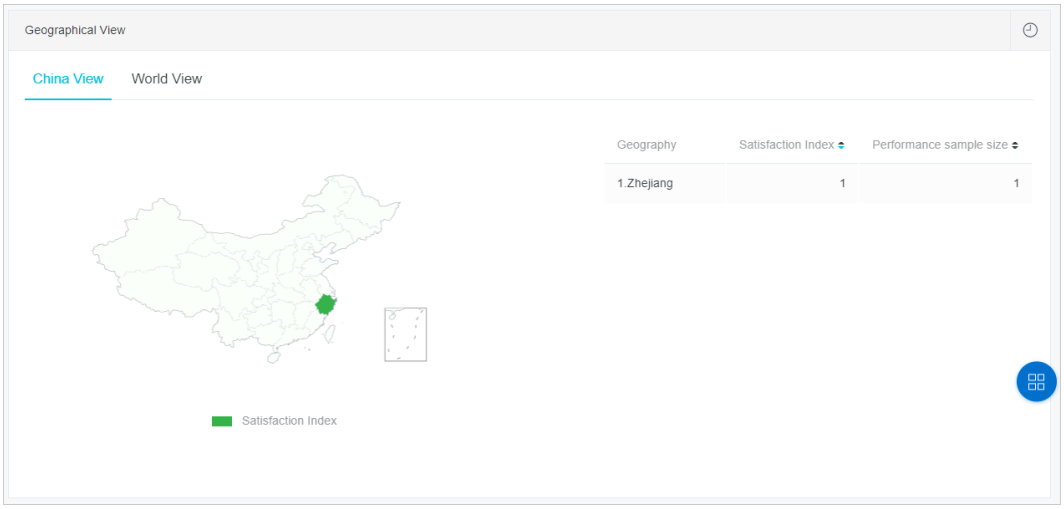
ARMS provides the following dimensions for the preceding types of custom statistics. Average statistics are used in this example.

- Statistics details
  The statistics details line chart shows trends of the average values and the size of samples for 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 view the average time consumption data and the size of sent samples in the corresponding period of time.



- 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. This helps the business side understand the differences between events in different regions and make decisions.



- Terminal view
  Browsers, devices, operating systems, and resolutions may affect the performance, compatibility, and display of a page. ARMS browser monitoring provides the average values and sample sizes in these dimensions. This helps the business side understand the distribution of the event in different browsers, devices, operating systems, and resolutions.

API method for sum statistics

After the browser monitoring SDK is introduced on the page, use the following log reporting API method in the business JavaScript file for sum statistics:

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

Call parameter description:

Parameter	Type	Description	Required	Default value
key	String	The name of the event.	Yes	None
value	Number	The cumulative number in each report. Default value: 1.	No	1

Example:

```
__bl.sum('event-a');
__bl.sum('event-b', 3);
```

API 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)`

Call parameter description:

Parameter	Type	Description	Required	Default value
key	String	The name of the event.	Yes	None
value	Number	The number of reported items. Default value: 0.	No	0

Example:

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

3.8. Quick diagnosis

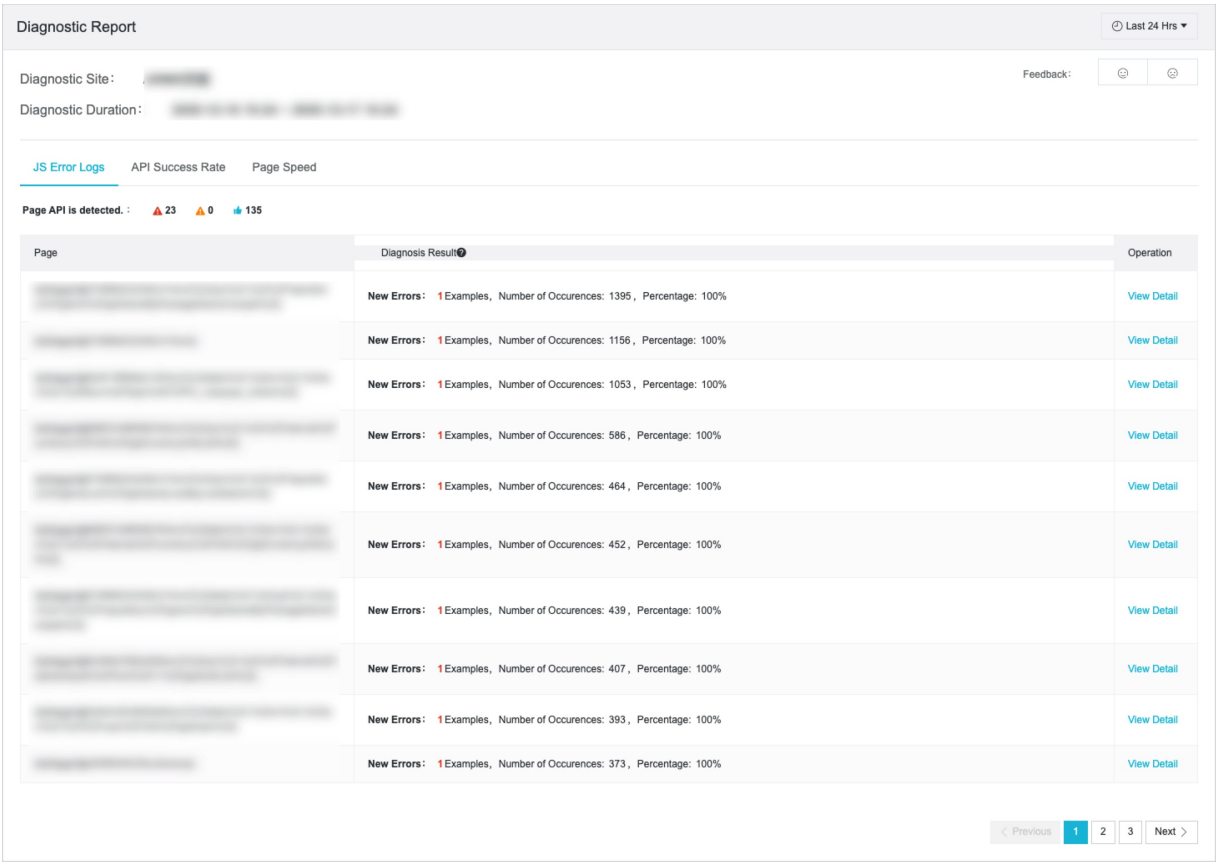
A diagnostic report summarizes the data collected by browser monitoring during a past period of time. Such a report analyzes and displays the website performance based on JS error logs, the success rate of API requests, and the page speed. This helps you understand the key monitoring metrics for websites and troubleshoot the issues.

Enter the page for the feature

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the application that you want to view.
3. In the upper-right corner of the page, click **One-click diagnosis**.
4. On the **Diagnostic Report** page, view the diagnostic report.

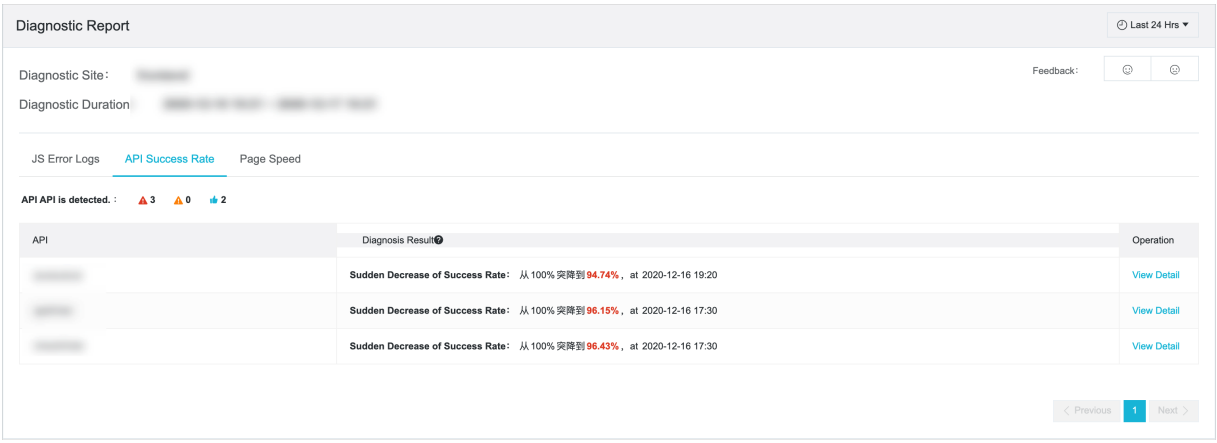
JS error logs

JS Error Logs uses the number of JS errors as a metric, and collects information about the error page views (PVs), exception PVs, and normal PVs. This feature displays the diagnostic result of each page, including the number of error PVs, the JS error rate, and the number of new error PVs. The following formula is used to calculate the JS error rate: JS error rate = Number of error PVs/Total number of PVs.



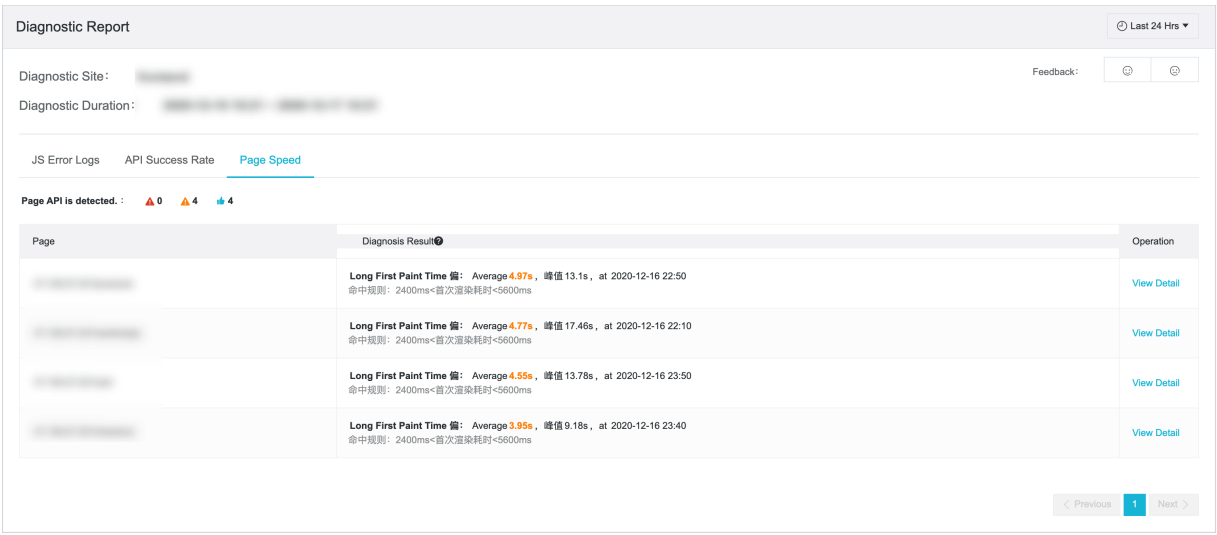
Success rate of API requests

API Success Rate uses the number of API requests as a metric, and counts the failed, abnormal, and successful API requests. This feature displays the diagnostic result of each API. For example, the result may indicate that the success rate of requests on an API is low or the success rate encounters an abrupt decrease. The result also contains the average and lowest success rates of API requests. The following formula is used: Success rate of API requests = Number of successful API requests/Total number of API requests.



Page speed

Page Speed provides the PV information of each page, including the numbers of failed, exception, and successful PVs. This feature calculates the page speed based on the PV information, and provides the diagnostic result. The information about the first paint time (FTP) of PVs is displayed, including the PVs that have a long FTP. The result also displays the average FTP, maximum FTP, and the time point when the maximum FTP occurs. The following formula is used: FTP = responseEnd - fetchStart.



4.Tutorials

4.1. Diagnose slow page loading by using ARMS Browser Monitoring

Slow page loading extremely affects the user experience, which determines the user loyalty. Therefore, the browser performance monitoring and analysis is particularly important. This topic describes how to use Application Real-Time Monitoring Service (ARMS) Browser Monitoring to diagnose slow page loading.

Prerequisites

Your browser application has been connected to ARMS. For more information, see Browser monitoring overview.

Issues that can be solved by ARMS Browser Monitoring

- Multidimensional performance analysis based on real user data
  - Fully restores real user operations without simulation and logon.
  - Uses real user data to reflect the user experience, which reduces statistical errors and helps you determine the error location, such as the region, device, and browser.
- Page performance analysis
  - First Paint Time: specifies whether the Domain Name System (DNS) resolution time and the TCP connection time are long.
  - DOM Ready: specifies whether the response time (RT) is long.
  - Fully Loaded Time: specifies whether resource loading is slow.
- JS error analysis
  Monitors and diagnoses JS errors.
- Custom statistics
  Uses the software development kit (SDK) to report custom events for custom statistics.
- Access details
  Queries all monitored user access information.
- Integration with data on Application Monitoring
  Diagnoses slow page loading with Application Monitoring.

Diagnose slow page loading

The following example shows the troubleshooting procedure:

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the target application. In the left-side navigation pane, choose **Application > Page Speed**.
3. In the **Page Load Time Details** and **Page Load Waterfall Plot** sections, check whether key performance metrics are normal.
   - If the value of First Paint Time in the **Page Load Time Details** section or the values of DNS Lookup, TCP Connection, and SSL Connection in the **Page Load Waterfall Plot** section are large, slow page loading may be caused by the network. In this case, check the network.

- If the value of DOM Ready in the **Page Load Time Details** section or the values of Time to First Byte(TTFB) and Content Download in the **Page Load Waterfall Plot** section are large, slow page loading may be caused by slow API responses. In this case, perform Step 4 to diagnose the issue.
- If the value of Fully Loaded Time in the **Page Load Time Details** section or the values of DOM Ready and Resource Download in the **Page Load Waterfall Plot** section are large, slow page loading may be caused by slow loading of browser resources. In this case, perform Step 5 to diagnose the issue.

4. In the left-side navigation pane, choose **Application > API Details**.
   i. In the **API Requests** section, find the target API and click the number in the **Slow Responses** column.
   ii. In the **API Slow Calls Details** dialog box, click **Show Invocation Trace** in the upper-right corner of the **Network Request Information** section to view the overall RT of the browser and the call sequence diagram of the backend application.
      - If the processing time of the backend application is short but the overall RT is long, the time consumed from API request sending to the server to data returning to the browser is long. In this case, choose **Application > View Details** in the left-side navigation pane. On the page that appears, click the **API Logs** tab, set the preceding API as the search value, and then click **Search** to view the access information, such as the network, region, browser, device, and operating system.
      - If the backend application takes a long time to process, the backend processing performance is poor. Click the magnifier icon in the **Method Stack** column. In the **Local Method Stack** dialog box, check which part of backend tracing is time-consuming to locate the problem.
5. In the **Slow Page Session Trace(TOP20)** section, click the page name of a slow session.
   In the **Page Resource Loading Waterfall** section on the **Slow Loading Details** page, you can view the details of the resources that cause slow page loading.

References

- Page speed
- Diagnose JS errors by using ARMS browser monitoring

4.2. Diagnose JS errors by using ARMS browser monitoring

JavaScript (JS) errors directly affect the quality of front-end applications. Therefore, it is particularly important to locate and diagnose JS errors. The JS error diagnosis feature provided by browser monitoring of Application Real-Time Monitoring Service (ARMS) can quickly and accurately locate and diagnose JS errors.

Prerequisites

Developers have generated a source map by using the builder.

Context

In practice, you may encounter the following difficulties during JS error diagnosis:

- Difficult to replicate

For example, when User A accesses a page, the page is loaded on the local browser of User A. Because the loading duration of the page is affected by factors such as the region, network, browser, or carrier, the actual situation occurred when User A accessed the page cannot be replicated.

- Lack of monitoring information for in-depth investigation
  Most browser monitoring services use the *PerformanceTiming* object to obtain the full page loading time, which does not include loading information of static resources. As a result, the performance bottleneck cannot be located.

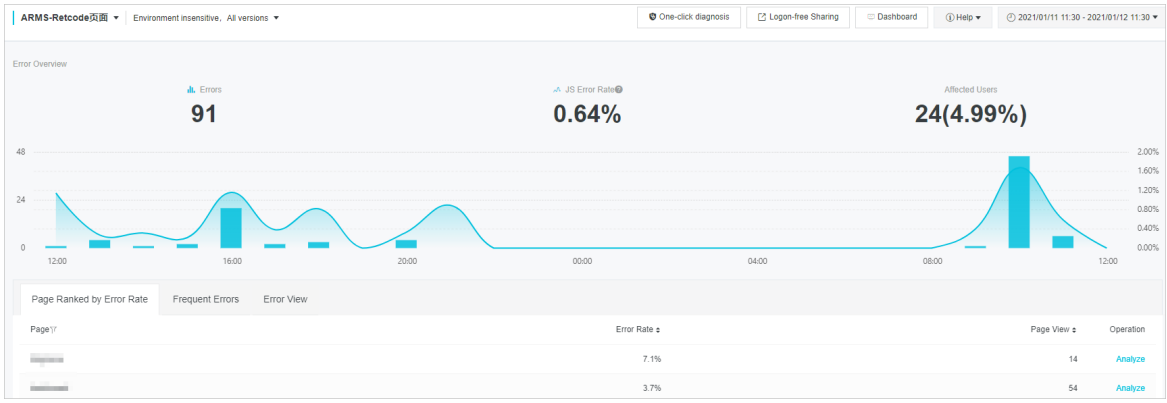
ARMS browser monitoring can use the source map to find the real location of the error in code and use the user behavior backtracking feature to replicate the JS error. This allows developers to quickly locate errors in source code and corresponding code blocks.

Step 1: Install the ARMS agent

To comprehensively monitor frontend applications, you must install the ARMS agent for your frontend applications. Select a method to install the ARMS agent as needed. For more information, see Browser monitoring overview.

Step 2: View the error overview

1. Log on to the ARMS console .
2. On the **Browser Monitoring** page, click the name of the application that you want to manage.
3. In the left-side navigation pane, click **JS Error Diagnosis**.



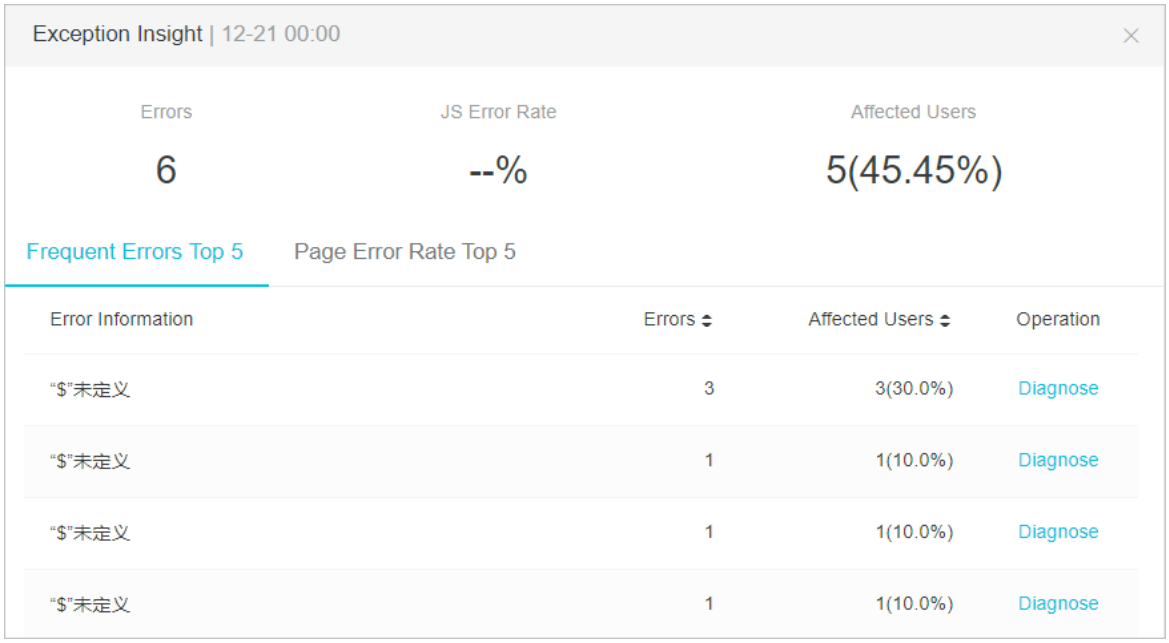
   - View the total number of errors, error rate, number of affected users, and percentage of affected users in **Error Overview**.
   - Determine the JS error trend based on the line chart.
   - Filter frequent errors in **Frequent Errors**.
   - Determine the global JS errors based on the information in **Page Ranked by Error Rate** and **Error View**.

Step 3: Diagnose a specific error

After you have determined any global errors, you must troubleshoot specific errors. You can use the following two methods to diagnose JS errors in ARMS browser monitoring. The second method is used as an example in this topic.

- On the **Frequent Errors** tab, click **Diagnose** to access the diagnosis page.
- In the error chart, go to the **Exception Insight** dialog box at a specific time point and then access the diagnosis page.

1. When you find that the error rate at a specific time point on the error chart suddenly increases, place the pointer over the turning point of the curve. When the pointer is displayed as a hand shape, click the turning point to display the Exception Insight dialog box for that time point. For more information, see JS error diagnosis.

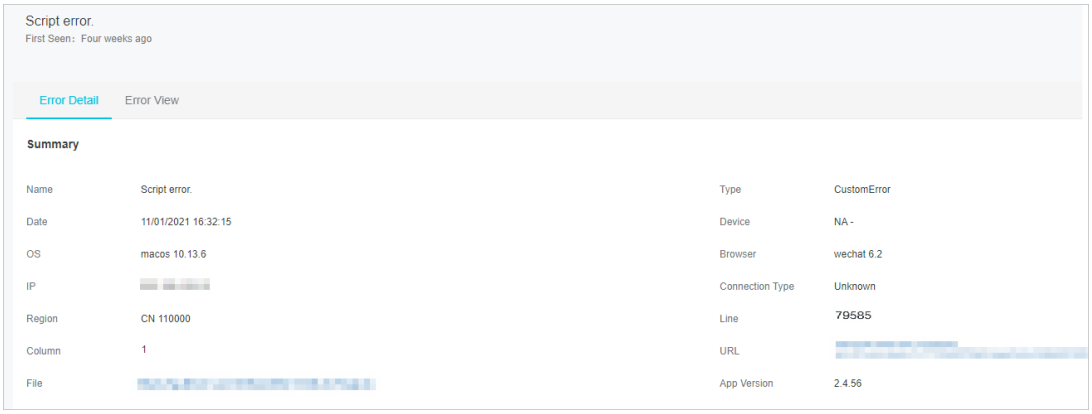


2. Click the **Frequent Errors Top 5** tab. On the tab that appears, select an error and click **Diagnose** in the **Actions** column. The **JS Error Diagnosis** page appears.

Step 4: View error details

Onsite JS error information includes the first occurrence time, first occurrence version (optional), error name, error type, device, operating system, browser, and the page, file, row, and column where the error resides. As shown in Error details page, the map module on the real-time dashboard reported an invalid data error during the update, and the error occurred in line 1 and column 79585.

Error details page



Step 5: Locate the error code

The information acquired from the JS error details is insufficient for problem diagnosis, because online code is unreadable after being compiled, compressed, and obfuscated. Although you can see the error occurred in line 1 and column 79585, this is not the actual location in the source code. In this case, you must use the source map for source code mapping.

1. In the **Stack Info** section, click the triangle icon on the left of a stack to expand the row, and click

Choose Sourcemap.

2. In the **Sourcemap File** dialog box, select an existing source map or upload a new one, and click **OK**.
3. ARMS browser monitoring uses the source map to locate the exact JS error location. If the selected source map matches the error in the source code, the original error location is marked in red in the **Source Code** section.

Step 6: View the backtracked user behaviors

The source code error occurred when invalid data was loaded during the creation of the map component. However, invalid data may appear in a variety of forms. Null judgment and fault tolerance have been performed for data in the preceding code, but the invalid data exception is still triggered. If a snapshot of the data at the time of the error can be taken, you can locate the problem more accurately. However, onsite data is usually unavailable in global exception capture, and troubleshooting depends on only the data information reported with the exception.

Is there any other way available to help troubleshooting? The best way is to replicate the problem. ARMS browser monitoring defines each event node on the page as user behavior, such as page loading, route jumps, page clicks, API requests, and console output. User behavior is connected in chronological order to form a behavior link. Behavior backtracking upon the error can help developers replicate the problem.

The user behavior backtracking upon the error shows that there is an API request before the error. This API request tries to request real-time update of the map module. However, the returned data is ConsoleNeedLogin, indicating logout from the page. This is the root cause of invalid data.

Related information

- JS error diagnostics
- Causes and solutions for script errors

4.3. Diagnose slow page loading

Difficulties exist when you identify and troubleshoot the problem of slow page loading. To solve this problem, the slow session tracing feature in browser monitoring of Application Real-Time Monitoring Service (ARMS) provides a performance waterfall chart for static resource loading on pages. This allows you to check the resource loading status on the pages, identify the root cause, and troubleshoot the problem.

Problem description

Slow page loading is one of the common and noted problems for users. You may encounter the following difficulties when you identify and troubleshoot the problems:

- Difficult to reproduce problems
  For example, when user A accesses a page, the page is loaded in the local browser of user A. The actual situation where user A accesses the page cannot be reproduced because the page loading time is affected by factors such as the region, network, browser, or carrier.
- Lack of monitoring information for troubleshooting
  Most browser monitoring services use the PerformanceTiming object to obtain all information about page loading time, which does not include static resource loading time. Therefore, performance bottlenecks cannot be identified.

Solution

You can connect the web application to ARMS browser monitoring, enable the page resource reporting feature, and use the slow session tracing feature of ARMS browser monitoring to identify the performance bottleneck.

Step 1: Connect to ARMS browser monitoring

By default, the ARMS browser monitoring SDK does not report static resource loading information. However, if you want to use the slow session tracing feature to identify the slow page loading problem, the static resource loading information is required.

Connect your web application to ARMS browser monitoring. For more information, see Install the browser monitoring probe by using CDN.

> **Notice** When you connect the web application to ARMS browser monitoring, you must select **Enable Page Resources Reporting**.

Step 2: Identify the problem

You can identify the problem by using one of the following methods:

Method 1: Check the access speed

1. Log on to the ARMS console.In the left-side navigation pane, click **Browser Monitoring**.
2. On the **Browser Monitoring** page, click the name of your application.
3. In the left-side navigation pane, click **Page Speed**.
   For more information about the **Page Speed** page, see Page speed. In this example, the page performance is poor. The loading time of the full page reaches 36.7s at about 11:00.
4. On the **Page Speed** page, drag the right-side scroll bar to view 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 causes a sharp increase of the page loading time.
5. In the **Slow Page Session Trace (TOP20)** section, click the page name in the **Page** column to access the **Session Details** page. Then, identify the cause and troubleshoot the problem based on the information on the **Session Details** page.
   The **Page Information** section in the upper part of the **Session Details** page shows the access information such as the client IP address, browser, and operating system. This helps you further identify the cause of the problem.
   The **Page Resource Loading Waterfall** section on the **Session Details** page shows the waterfall chart of static resource loading. This helps you identify the performance bottleneck.
   For more information about the **Session Traces** page, see Slow session tracing.

Method 2: Trace the session

1. Log on to the ARMS console.In the left-side navigation pane, click **Browser Monitoring**.
2. On the **Browser Monitoring** page, click the name of your application.
3. In the left-side navigation pane, click **Session Traces** to access the **Session List** page.
   The **Session List** page shows the top 100 sessions of the application by loading time. You can filter sessions by page, session ID, browser, or browser version to identify time-consuming sessions.
4. On the **Session List** page, find the session and click **Details** in the **Operation** column. Then, identify the cause and troubleshoot the problem based on the information on the **Session Details** page.
   For more information about the **Session Traces** page, see Slow session tracing.

The troubleshooting is complete by using the slow session tracing feature. This feature can help you reproduce the page resource loading status and identify the performance bottleneck.

What to do next

To avoid passive diagnostics after an exception occurs, you can also use the alert feature of ARMS to create an alert for one or all API operations. This ensures that the O&M team immediately receives a notification when an exception occurs.

For information about how to create an alert, see Create ARMS alerts.

References

- Page speed
- Slow session tracing
- Create ARMS alerts

4.4. Diagnose JS errors by backtracking user actions

In the JS error diagnosis process, Application Real-Time Monitoring Service (ARMS) Browser Monitoring provides the user action backtracking feature to shows you a comprehensive list of the user actions leading up to the error.

Context

ARMS Browser Monitoring defines all events on the page as user actions, including console actions, page jumps, user clicks, user inputs, and API calls. You can connect user actions in chronological order to form an action trace. Then, you can backtrack and analyze the action trace to reproduce the situation in which the error occurred.

Step 1: Install the ARMS agent

After you install the ARMS agent, your frontend applications can be fully monitored. Select a method to install the ARMS agent as needed. For more information, see Browser monitoring overview.

Step 2: View frequent errors

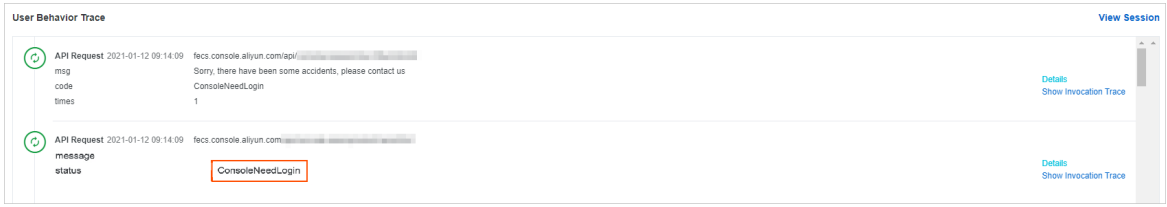
1. Log on to the ARMS console. In the left-side navigation pane, click **Browser Monitoring**.
2. On the **Browser Monitoring** page, click the name of the target application.
3. In the left-side navigation pane, click **JS Error Diagnosis**.
4. On the **JS Error Diagnosis** page, click the **Frequent Errors** tab.

Step 3: Diagnose causes of the error

Use the user action backtracking feature to reproduce the situation in which the JS error occurred and diagnose causes of the error.

1. Click **Diagnose** in the **Operation** column on the right.
2. On the **Error Detail** tab, analyze user behaviors in the **User Behavior Trace** section to determine

which operation causes the error.



Related information

- Causes and solutions for script errors

4.5. Slow session tracing

The slow session tracing feature of Application Real-Time Monitoring Service (ARMS) provides a performance waterfall chart for static resource loading during page loading. This helps you learn more about the loading status of page resources based on page performance data and locate performance bottlenecks.

Prerequisites

> **Notice** The reporting of static resource loading information is triggered during page loading, and a large amount of information is reported. When the loading time is greater than 8s, full data is reported. If the loading time is within the range from 2s to 8s, 5% of the information is sampled and reported. If the loading time is less than 2s, no information is reported. We recommend that you do not set this parameter if your application requires high page performance.

By default, ARMS Browser Monitoring SDK does not report information about static resources loaded on a page. To obtain information about static resources loaded on a page and enable slow session tracing, set sendResource to *true* in the `config` file of the SDK. After the application is redeployed, when an onload event on the page is triggered, the information about static resources loaded on the current page is reported. Then, you can locate slow page loading problems in ARMS Browser Monitoring.

The following code shows an example of the SDK configuration:

```
<script>
!( function(c,b,d,a){c[a]||(c[a]={});c[a].config={pid:"atc889zkcf@8cc3f6354******",imgUrl:"https://arms-retcode.aliyuncs.com/r.png?",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>
```

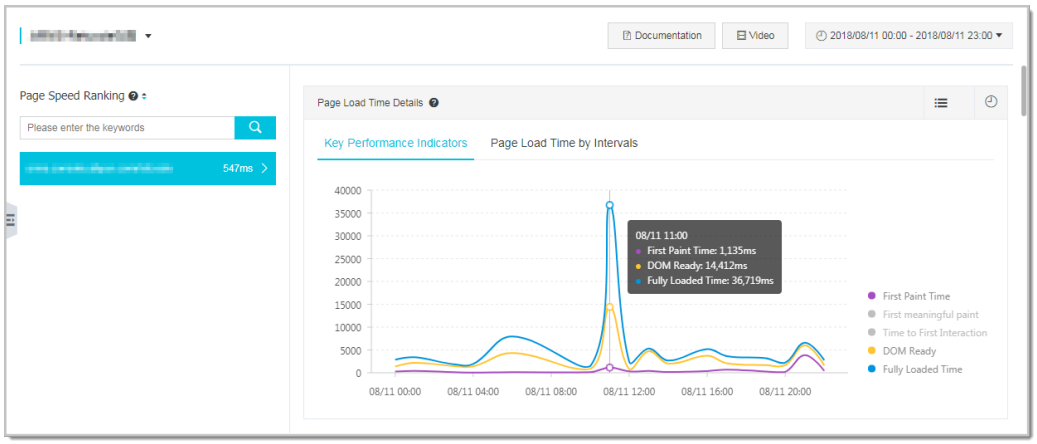
Portal

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

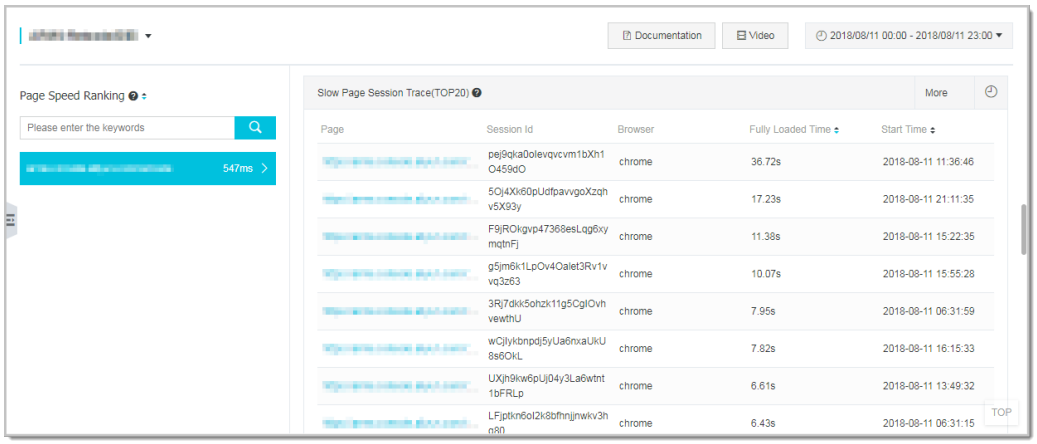
Use case: Locate page performance bottlenecks

The following example shows how to locate page performance bottlenecks.

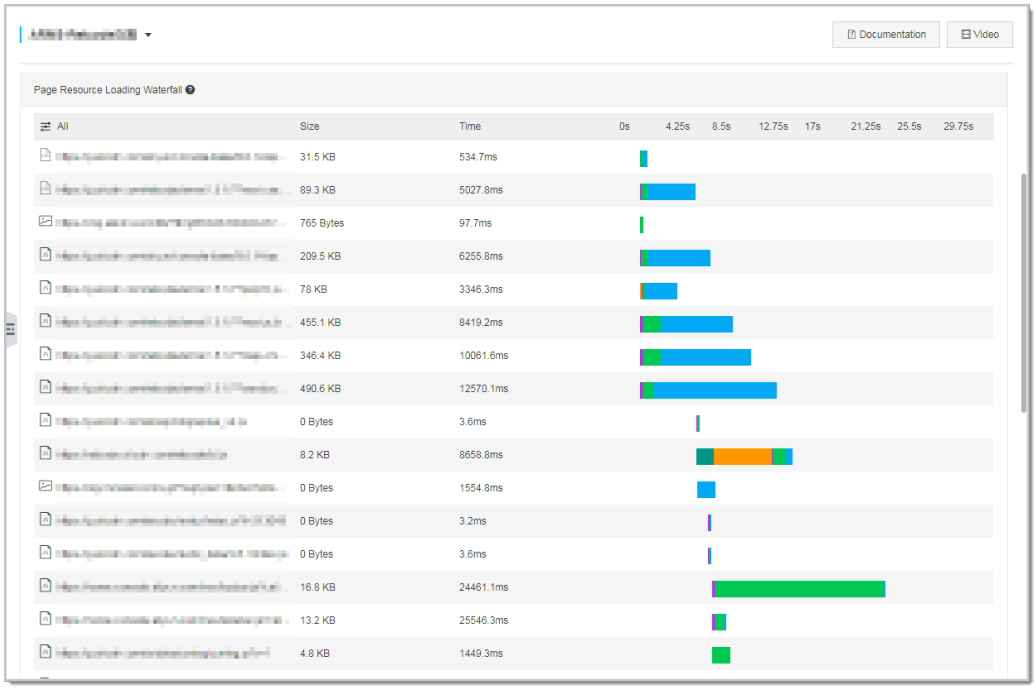
1. In the left-side navigation pane, choose **Application > Page Speed**. The result is shown in the following figure. It took 36.7s to completely load the page at 11:00.



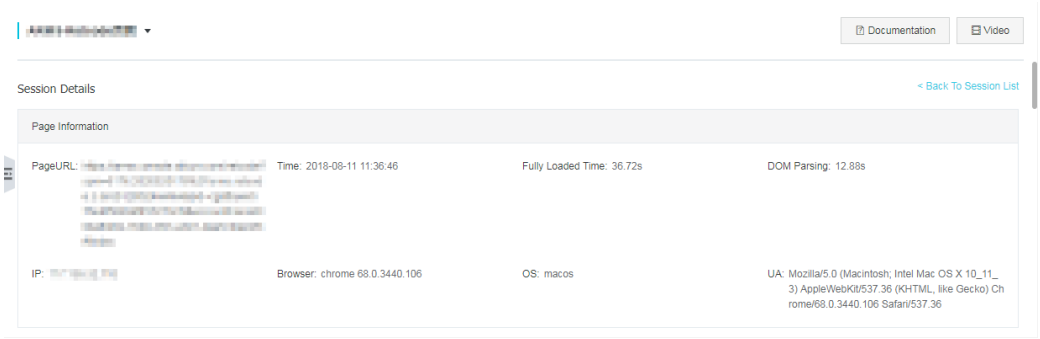
2. On the **Page Speed** page, scroll down the page to the **Slow Page Session Trace(TOP20)** section. This section lists the top 20 sessions with the lowest page loading speed within the specified period of time.
   To change the specified time period, you can click the clock icon in the upper-right corner of this section and select a value.
   In the following figure, the page loading speed of a session at 11:36:46 is 36.72s. This access is the direct cause of the sharp increase in the page loading time.



3. In the **Slow Page Session Trace(TOP20)** section, click a page name in the **Page** column to go to the **Trace Sessions > Session Details** page. This page provides an intuitive view of the waterfall chart for the loading status of static page resources and allows you to locate the performance bottleneck for resource loading.



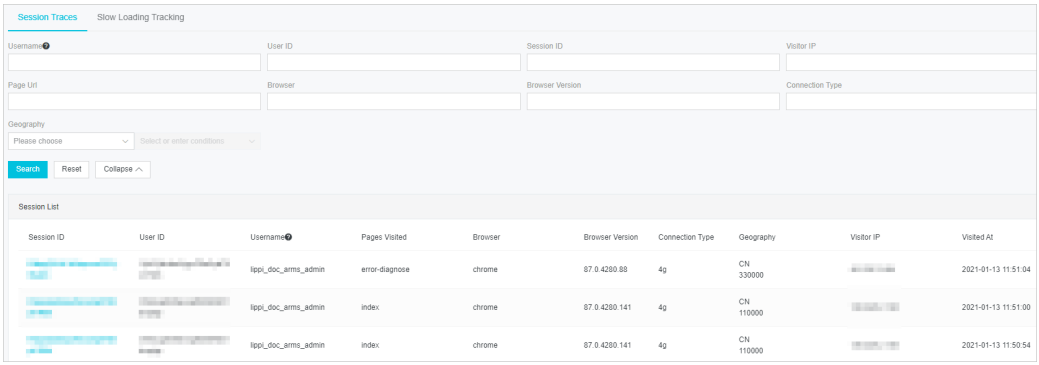
4. Choose **Session Traces > Session Details**. On the page that appears, you can view the client IP address, browser, operating system, and other information of the current access in the **Page Information** section. Then, you can locate the causes of slow sessions and optimize the page accordingly.



Other channels for detecting performance problems

In addition to the **Page Speed** page, you can also locate performance problems on the **Session Traces** page.

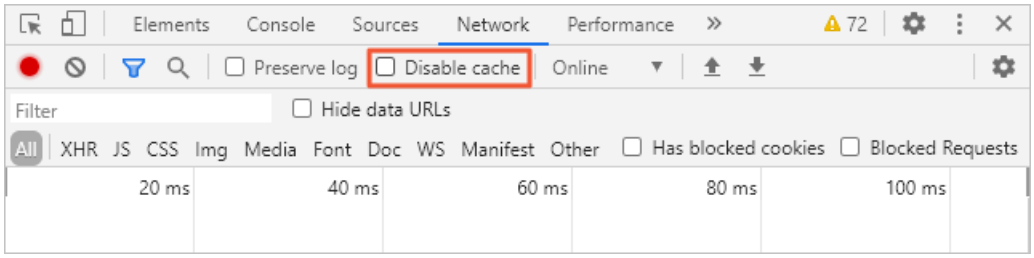
1. In the left-side navigation pane, choose **Session Traces**. On the **Session Traces** page, you can view the list of all sessions of your application. You can filter sessions based on conditions such as the username, user ID, page URL, session ID, browser, and browser version.

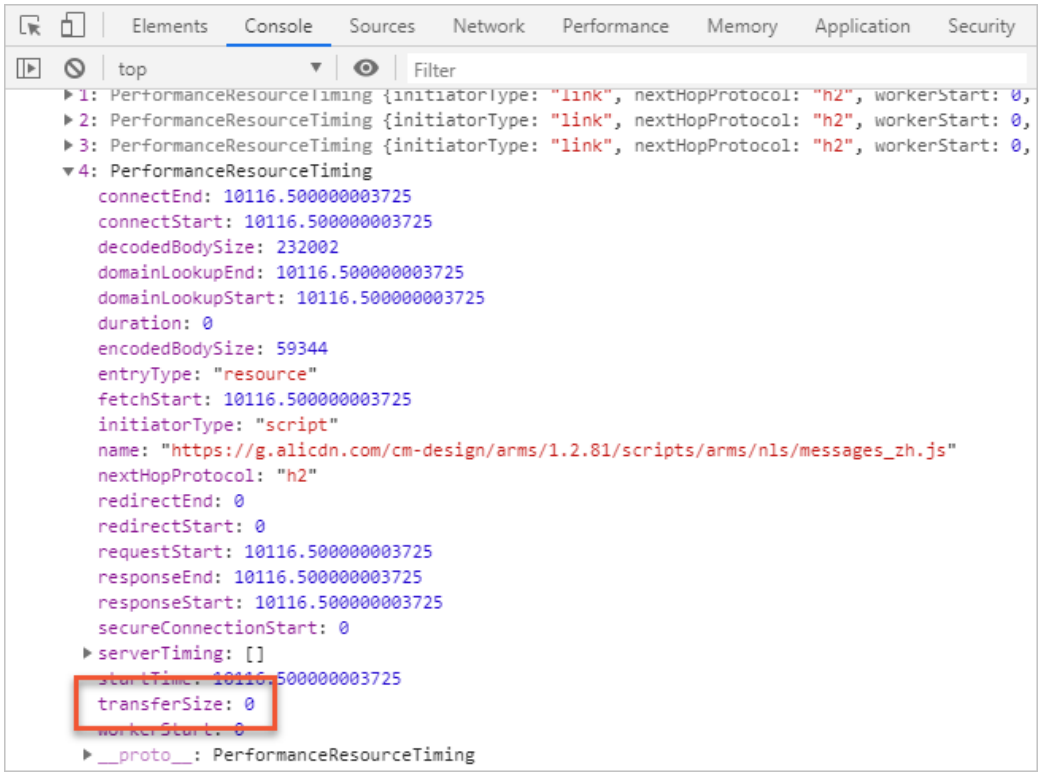


2. In the **Session ID** column, click the ID of the session whose details you want to view. On the **Session Details** page, you can view the session overview and session traces. For more information, see Session tracing.

FAQ

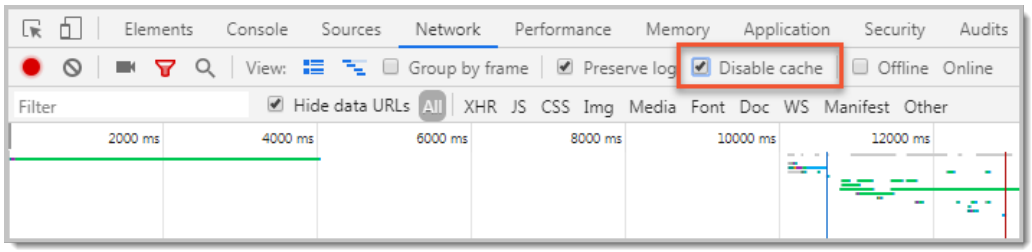
1. **Why is `Size` 0 on the waterfall chart for resource loading?**
   `Size` is obtained by PerformanceResourceTiming.transferSize. The read-only attribute of transferSize indicates the size of the extracted resource in eight bits. If resources are obtained from the local cache or are cross-origin resources, the returned value of this parameter is 0. Open the Chrome browser, press **F12** to open the developer tool panel. When **Disable cache** on the **Network** tab is not selected, transferSize is 0.

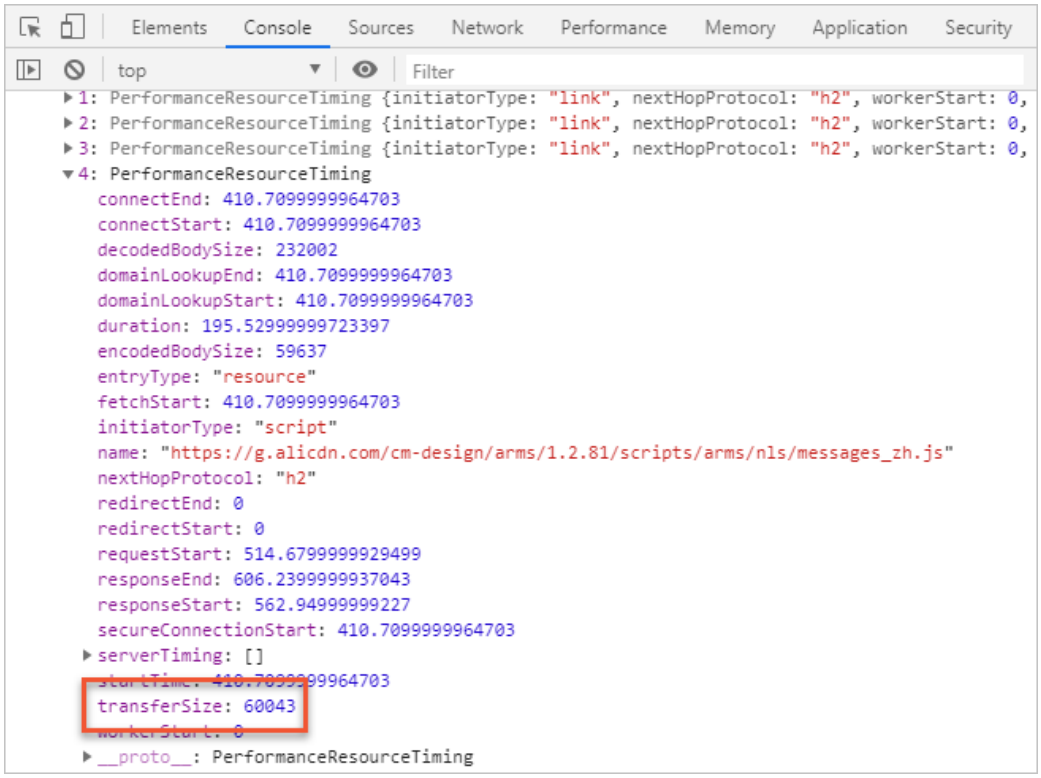




Solution

Select **Disable cache**. Then, the actual value of `transferSize` is displayed.





2. **Why is `Time` 0 on the waterfall chart for resource loading?**
   `Time` is obtained by PerformanceResourceTiming.duration. On the waterfall chart for static resource loading, `Time` is sometimes 0. The reason is that the request hits a long cache that is controlled by `max-age`.
   Solution
   Open the Chrome browser and press **F12** to open the developer tool panel. Clear **Disable cache** on the **Network** tab and refresh the page. Then, the network access time is displayed.
3. **Why is the returned value for Time 0?**
   The value of many time-related parameters returned by an API is 0. The reason is that the time point for obtaining cross-origin resources is 0 due to the same-origin policy. The following attributes are involved:
   - redirectStart
   - redirectEnd
   - domainLookupStart
   - domainLookupEnd
   - connectStart
   - connectEnd
   - secureConnectionStart
   - requestStart
   - responseStart

   Solution
   Add `Timing-Allow-Origin`, such as `Timing-Allow-Origin:*`, to the resource response header.
4. **In which time range does the waterfall chart for API loading show the API loading status?**

The following information shows the start time and end time of the time range on the waterfall chart for API loading:

- Start time: the time when the page starts to load
- End time: full page loading time plus 1 minute

The waterfall chart for API loading shows the overall status of the requested API during page loading.

5. **Why is the response time on the waterfall chart for API loading different from that on the waterfall chart for page resource loading?**
   The response time on the waterfall chart for API loading is several milliseconds longer than that on the waterfall chart for page resource loading, because the two values are obtained in different ways. The response time on the waterfall chart for API loading is calculated from the time when the API sends a request to the time when the API returns data. The API response time on the waterfall chart for page resource loading is obtained by calling performance.getEntriesByType('resource') that is provided by the browser.
   The several-millisecond difference does not affect the troubleshooting of performance bottlenecks.
6. **What is the start time of the timeline on the waterfall chart for API loading?**
   The start time of the timeline on the waterfall chart for API loading is the difference between the time when the API sends a request and the time when the fetchStart value of the page is returned. This timeline displays when the API sends the request during page loading and its response time.

References

- Page speed
- Browser monitoring FAQ

4.6. Use the front-to-back tracing feature to diagnose causes of API errors

In browser monitoring, you do not know the performance of the network transmission or the trace and performance of backend services even if you know the API response time. Therefore, you cannot immediately troubleshoot API errors in applications. The front-to-back tracing feature can resolve this issue. This feature connects the frontend and backend traces of an API request to reproduce the code execution process.

Prerequisites

The browser monitoring and application monitoring features of Application Real-Time Monitoring Service (ARMS) are activated. For more information, see Activate and upgrade ARMS. The version of application monitoring is 2.4.5 or later. For more information, see Overview.

Context

Application monitoring provides the backend processing performance and traces of API requests. However, the data does not reflect real user experience. Browser monitoring can monitor only the overall response time and statuses of API requests, but cannot provide the traces or performance of backend services. In this case, the front-to-back tracing feature connects the frontend and backend to provide you with a one-stop troubleshooting experience.

Configure ARMS browser monitoring

The API and the domain name of the application have the same origin.

1. Check whether the frontend connects to the backend.
2. Do not select the Disable Automatic API Reporting option. Enable the automatic API reporting.
3. Set enableLinkTrace to `true` to enable the front-to-back tracing feature. The following sample code shows how to enable the front-to-back tracing feature:

```
<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))setAttribute("crossorigin","",src=d)
})(window,document,"https://retcode.alicdn.com/retcode/bl.js","__bl");
</script>
```

The API and the domain name of the application do not have the same origin.

1. Check whether the frontend connects to the backend.
2. Set enableLinkTrace and enableApiCors to `true`.

```
<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, enableApiCors: 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>
```

Notice If enableApiCors is set to `true`, the backend service must also support cross-domain requests and custom header values. Make sure that all requests can be called in combinations as expected. Otherwise, these requests may fail. The following sample code shows how to configure NGINX:

```
upstream test {
    server 192.168.220.123:9099;
    server 192.168.220.123:58080;
  }
  server {
    listen   5800;
    server_name  192.168.220.123;
    root      /usr/share/nginx/html;
    include /etc/nginx/default.d/*.conf;
    location / {
      proxy_pass http://test;
      proxy_set_header Host $host:$server_port;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Real-PORT $remote_port;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header  EagleEye-TraceID $eagleeye_traceid;
      proxy_set_header  EagleEye-SessionID $eagleEye_sessionid;
      proxy_set_header  EagleEye-pAppName $eagleeye_pappname;
    }
```

3. Configure the Ignore method. For more information, see . The following sample code shows how to configure the Ignore method:

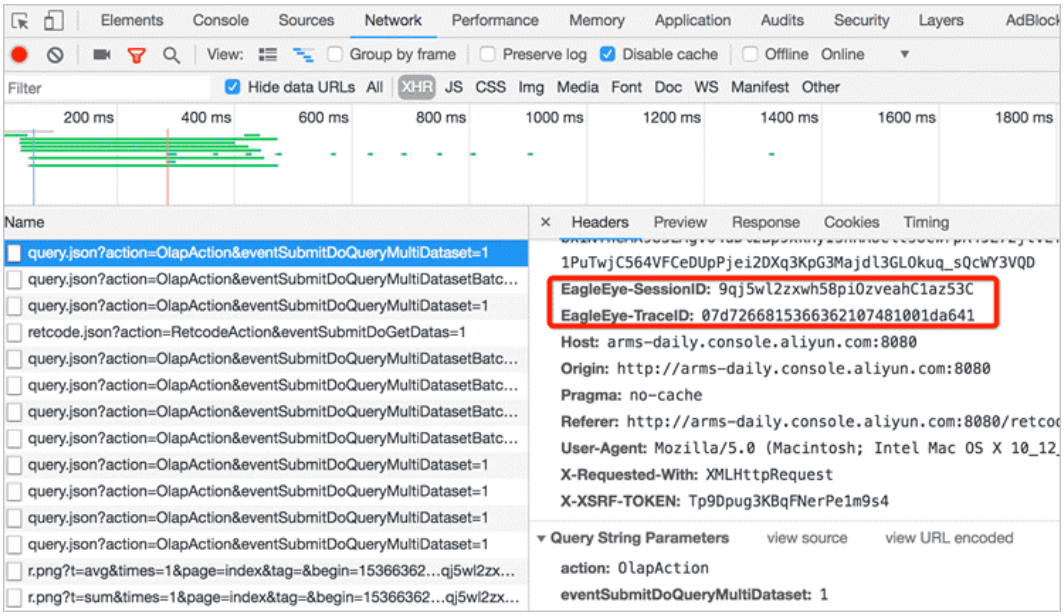
```
let whitelist = ['api.xxx','source3']; // The whitelist.
let blacklist = ['source2','source6']; // The blacklist.
// You can select a blacklist and whitelist based on your business requirements. The whitelist and blacklist are controlled by the returned true and false values of the method.
ignore: {
      ignoreApis: [
        function(str) {  // The method.
          if (whitelist.includes(str)) {
            return false;
          }
          return true; // Ignore if true is returned.
        }]
    }
```

> **Note** The Ignore method is similar to blacklists and whitelists. This method prevents header modifications against unauthorized third-party resource requests and errors of resource requests.

Principle

- If automatic API reporting is enabled and the API and the domain name of the application have the same origin, 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 frontend and backend.
- If the API and the domain name of the application do not have the same origin, no custom headers are added to the API request headers. This ensures that the application can send requests as expected.
- To check whether the front-to-back tracing configurations have taken effect, go to the console and view the API request headers. If the EagleEye-TraceID and EagleEye-SessionID headers are included in the API request headers, the configurations have taken effect.

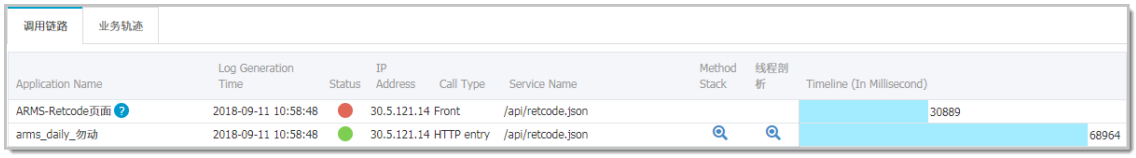
> **Warning** The values of EagleEye-TraceID and EagleEye-SessionID are automatically generated. We recommend that you do not manually set the values.



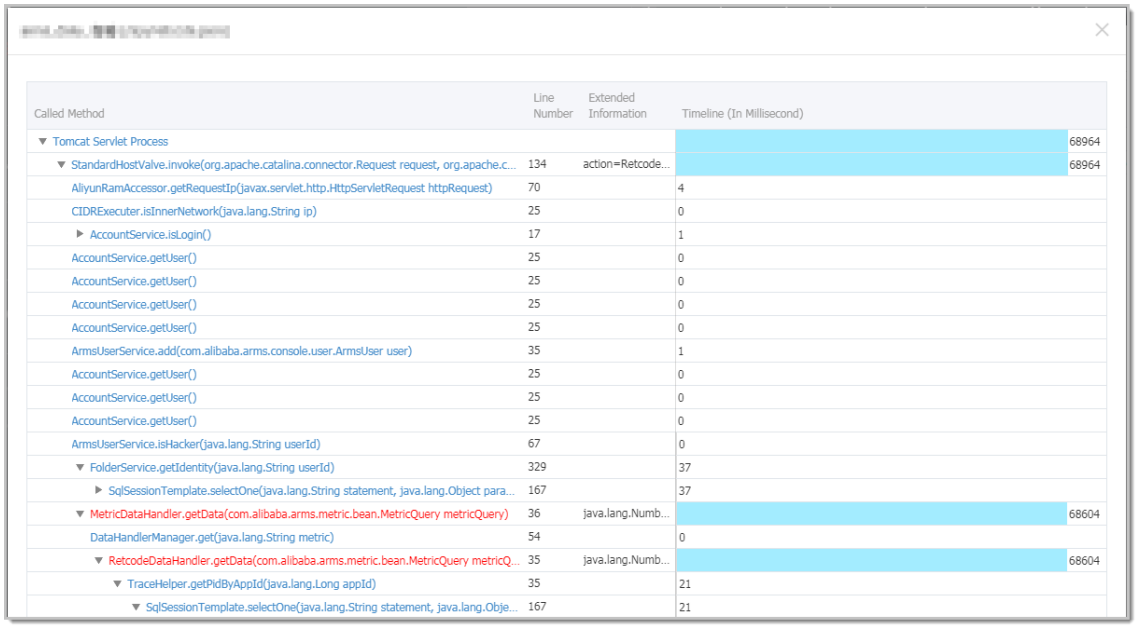
Scenarios and cases

If a request takes a longer time than expected, you can determine whether the cause is network transmission or backend call process based on the call timeline. You can also click thread profiling of the backend application to view the complete backend trace of the request. In this case, you can identify the cause of an API error based on the business.

- Locate the error if the API returns an error code or a business logic error occurs.
  i. Log on to the ARMS console .
  ii. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the application. In the left-side navigation pane, click **API Request**.
  iii. On the **API Failure List** tab of the **API Link Trace(Top20)** section, find the API or trace ID, and click **Link Trace**. You can then view the overall response time of browser monitoring and the call sequence chart of the backend application.



  iv. Determine whether the network transmission or backend call process takes a longer time based on the call timeline.
  v. Click the Magnifier icon in the **Method Stack** column of the backend application to view the overall backend trace of the request. You can then locate the cause of the API error based on the business.



Called Method	Line Number	Extended Information	Timeline (In Millisecond)
Tomcat Servlet Process			68964
StandardHostValve.invoke(org.apache.catalina.connector.Request request, org.apache.c...	134	action=Retcode...	68964
AliyunRamAccessor.getRequestIp(javax.servlet.http.HttpServletRequest httpRequest)	70		4
CIDRExecuter.isInnerNetwork(java.lang.String ip)	25		0
AccountService.isLogin()	17		1
AccountService.getUser()	25		0
AccountService.getUser()	25		0
AccountService.getUser()	25		0
AccountService.getUser()	25		0
ArmsUserService.add(com.alibaba.arms.console.user.ArmsUser user)	35		1
AccountService.getUser()	25		0
AccountService.getUser()	25		0
AccountService.getUser()	25		0
ArmsUserService.isHacker(java.lang.String userId)	67		0
FolderService.getIdentity(java.lang.String userId)	329		37
SqlSessionTemplate.selectOne(java.lang.String statement, java.lang.Object para...	167		37
MetricDataHandler.getData(com.alibaba.arms.metric.bean.MetricQuery metricQuery)	36	java.lang.Numb...	68604
DataHandlerManager.get(java.lang.String metric)	54		0
RetcodeDataHandler.getData(com.alibaba.arms.metric.bean.MetricQuery metricQ...	35	java.lang.Numb...	68604
TraceHelper.getPidByAppId(java.lang.Long appId)	35		21
SqlSessionTemplate.selectOne(java.lang.String statement, java.lang.Obje...	167		21

- Locate the error if the response time of the API is longer than expected.
  i. Log on to the ARMS console .
  ii. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the application. In the left-side navigation pane, click **API Request**.
  iii. In the **API Link Trace(Top20)** section, the APIs are sorted in descending order by response time. or trace ID. You can find the API or trace ID whose response time is longer than expected,
  iv. and click **Link Trace**. You can then view the overall response time of browser monitoring and the call sequence chart of the backend application.
    - If the processing time of the backend application is short but the overall response time is long, the network transmission takes a longer time. Click **View Details**. On the details page, view the network, region, browser, device, and operating system of the access request.
    - If the backend application takes a long time to process the access request, the backend processing performance is poor. Click the Magnifier icon in the **Method Stack** column. In the dialog box that appears, check which part of backend tracing is time-consuming to locate the problem.

5.Advanced options

5.1. Page data reporting of SPAs

In a single page application (SPA), a page is refreshed only once. Traditionally, page view (PV) data is reported once only after the page is loaded. However, PV data of sub-pages cannot be collected, and logs of other types cannot be aggregated based on sub-pages. This topic describes how to use Application Real-Time Monitoring Service (ARMS) Browser Monitoring SDK to resolve issues about page data reporting of SPAs.
ARMS Browser Monitoring SDK provides two methods for processing SPA pages.

- Enable automatic SPA page resolution
- Manually report data

Enable automatic SPA page resolution

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

In initial configuration items, set enableSPA to `true`. This way, hashchange events can be listened to on the page and PV data can be automatically reported again. URL hash is used as the page field for reporting other data.

enableSPA can also be used with parseHash. For more information, see enableSPA and parseHash.

Manually report data

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

ARMS Browser Monitoring SDK provides the setPage method for you to manually update the value of page name. You can use the new value when you report data. When this method is called, PV data will be reported again by default. For more information, see setPage().

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

Related information

- SDK reference
- API reference

5.2. Pre-report data

Data reporting may fail in specific cases, for example, when the software development kit (SDK) initialization has not been completed. This topic describes how to use the SDK of Application Real-Time Monitoring Service (ARMS) Browser Monitoring to pre-report data.

Data reporting failure scenarios

Data reporting errors may occur in the following scenarios:

- Some data needs to be reported when a page is being loaded but the SDK initialization is not complete 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 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. Example:

```
__bl.pipe = [
  // Report HTML of the current page as an API.
  ['api', '/index.html', true, performance.now, 'SUCCESS'], // This is equivalent to __bl.api(api, success, time, code, msg).
  // After SDK initialization is complete, enable automatic Single Page Application (SPA) resolution.
  ['setConfig', {enableSPA: true}]
];
```

To report a single data record, use:

```
__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 that is set 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, page view (PV) is reported for the first time after application initialization. However, it is not clear if the SDK initialization is complete.

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

Related information

- SDK reference
- API reference

6.SDK reference

The browser monitoring feature of Application Real-time Monitoring Service (ARMS) allows you to configure a variety of SDK parameters to meet additional requirements, such as ignoring the reports of specific URLs, API operations, and JS errors, aggregating pages by filtering non-key characters from URLs, and reducing reports and loads by using random sampling.

Index

Configure browser monitoring SDK parameters

You can configure browser monitoring SDK parameters in one of the following ways:

- When you install an ARMS browser monitoring agent to 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.

```
<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 the page is initialized, call the setConfig method in the IS code to modify the parameters. The following table describes the parameters that the `__bl.setConfig(next)` method calls.

Parameter	Type	Description	Required	Default value
next	Object	The configuration item and its value that you want to modify.	Yes	None

The following code provides an example on how to modify the disableHook parameter to disable automatic API reporting:

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

pid

Parameter	Type	Description	Required	Default Value
pid	String	The unique ID of the project, which is automatically generated by ARMS when it creates the site.	Yes	N/A

[Back to the top]

uid

Parameter	Type	Description	Required	Default Value
uid	String	The user ID, which identifies the user and can be manually configured to be retrieved based on the user ID. If you do not configure the settings, they are automatically generated by the SDK and updated semi-annually.	• Weex scenario: Required • Other scenarios: not required	• Weex scenario: None • Other scenarios: Automatically generated by SDKs

[Back to the top]

tag

Parameter	Type	Description	Required	Default Value
tag	String	The input tag. Each log carries a tag.	No	None

[Back to the top]

page

Parameter	Type	Description	Required	Default Value
page	String	The page name.	No	The key part of the current page URL is taken by default: `host + pathname`.

[Back to the top]

setUsername

Parameter	Type	Description	Required	Default Value
setUsername	Function	Lets you set up a method that needs to return a user name of type String.	No	None

This parameter is used to configure a method that returns a user name as a string. After you configure this parameter, you can use the returned user name to perform full link tracing on the user and query the sessions related to the user.

> **Note** If the user name cannot be obtained when the page starts to be loaded, you can set the return value to null rather than a temporary user name. In general, the SDK calls the setUsername method to obtain the user name when it sends logs. However, if you set the return value to a temporary user name, the SDK uses the temporary user name and does not call setUsername to obtain the user name.

The following code provides an example on how to call the setConfig method to modify SDK parameters:

```
// Call the setConfig method to modify the SDK parameters.
__bl.setConfig({
  setUsername: function () {
    return "username_xxx";
  }
});
```

[Back to the top]

enableSPA

Parameter	Type	Description	Required	Default Value
enableSPA	Boolean	Listen to the hashchange event on the page and report the PV again. This method is applicable to single-page application scenarios.	No (supported by Web scenarios only)	`false`

[Back to the top]

parseHash

Parameter	Type	Description	Required	Default Value
parseHash	Function	And enableSPA Use with.	No	See below

In SPA scenarios (see Page data reporting of SPAs), if 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 using the following string processing method:

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

You do not need to modify this parameter in most cases. However, if you want to use a custom page name to report data or the URL hash is complex, you can modify this parameter. Example:

```
// Define the mapping between the hash values and pages.
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 the SDK parameters.
  __bl.setConfig({
    parseHash: function (hash) {
      key = hash.replace(/\?.*$/, '');
      return PAGE_MAP[key] || 'Unknown page';
    }
  });
});
```

[Back to the top]

disableHook

Parameter	Type	Description	Required	Default Value
disableHook	Boolean	Disables the listener for AJAX requests.	No	`false`: By default, the listener is listened to and used for reporting the API call success rate.

[Back to the top]

ignoreUrlCase

Parameter	Type	Description	Required	Default Value
ignoreUrlCase	Boolean	Page URL case is ignored.	No	`true`: defaults to ignored.

[Back to the top]

urlHelper

Parameter	Type	Description	Required	Default Value
urlHelper	*	Replace the old parameter ignoreUrlPath , which is used to configure URL filtering rules.	No	See below

When a page URL is in a similar format as `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 aggregated during data browsing. To implement page aggregation, set the urlHelper parameter to filter non-key characters such as the project ID in this example.

> **Notice**
>
> - The parameter ignoreUrlPath was used to configure URL filtering rules and is replaced by the urlHelper parameter. If you use the ignoreUrlPath parameter, the configuration still takes effect. If you use the ignoreUrlPath and urlHelper parameters at the same time, the configuration specified by urlHelper takes effect.
> - This parameter is valid only when the page URL is automatically obtained and used as Page. This parameter is invalid if you manually modify Page by calling the setPage or setConfig method (see API reference) or the enableSPA parameter is set to `true` .

Default value

The default value of this parameter is the following array and does not need to be modified.

```
[
  // Replace all numbers in the URL with asterisks (*).
  {rule: /\/([a-z\-_]+)? \d{2,20}/g, target: '/$1**'},
  // Remove the forward slash (/) at the end of the URL.
  /\/$/
]
```

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

Value type

The value of urlHelper can be one of the following types:

- `String` or `RegExp` (regular expression): The matched strings are removed.
- `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 the String::replace method described in related JS tutorials.
- `Function` : The original string is used as the input parameter for method execution, and the execution result is used as Page.
- `Array` : This type is used to set multiple rules. The type of each rule can be one of the preceding types.

[Back to the top]

apiHelper

Parameter	Type	Description	Required	Default Value
apiHelper	*	Replace the old parameter ignoreApiPath To configure API filtering rules.	No	See below

This parameter filters out non-key characters in API URLs when the results of API operations are automatically reported. The usage and function of this parameter are the same as those of urlHelper.

> **Notice** The parameter ignoreApiPath was used to configure API filtering rules and is replaced by the apiHelper parameter. If you use the ignoreApiPath parameter, the configuration still takes effect. If you use the ignoreApiPath and apiHelper parameters at the same time, the configuration specified by apiHelper takes effect.

Default value

The default value of this parameter is an object and does not need to be modified.

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

The default value of this parameter filters out the numbers in strings such as `xxxx/123456`.

[Back to the top]

parseResponse

Parameter	Type	Description	Required	Default Value
parseResponse	Function	It is used to parse the data returned when the API is automatically reported.	No	See below

This parameter parses the data returned when the results of API operations are automatically reported.

Default value

The following code shows the default value of this parameter:

```
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 common applications, the default value of this parameter does not need to be modified. If the default value cannot meet your business requirements, you can set a new value.

[Back to the top]

ignore

Parameter	Type	Description	Required	Default Value
ignore	Object	Ignores Errors of the specified URL/API/JS. Logs that conform to rules are ignored and not reported, including sub-configuration items ignoreUrls , ignoreApis , ignoreErrors and ignoreResErrors .	No	See below

The value of ignore is an object that contains four attributes: ignoreUrls, ignoreApis, ignoreErrors, and ignoreResErrors. You can set one or more attributes of the parameter value.

Default value

The following code shows the default value of this parameter:

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

ignoreUrls

The ignoreUrls attribute is used to ignore reports from specific URLs. Logs from the URLs that comply with the specified rule are not reported. The value of this attribute can be a string (the `String` type), a regular expression (the `RegExp` type), a method (the `Function` type), or an array that includes the preceding three types. Example:

```
__bl.setConfig({
        ignore: {
          ignoreUrls: [
          'http://host1/',  // Character string
          /. +? host2. +/,    // Regular expression
          function(str) {  // Method
            if (str && str.indexOf('host3') >= 0) {
              return true; // The data is not reported.
            }
            return false; // The data is reported.
          }]
        }
      });
```

ignoreApis

The ignoreApis attribute is used to ignore the reports from APIs that comply with a specified rule. The value of this attribute can be a string (the `String` type), a regular expression (the `RegExp` type), a method (the `Function` type), or an array that includes the preceding three types. Example:

```
__bl.setConfig({
        ignore: {
          ignoreApis: [
          'api1','api2','api3', // Character string
          /^random/,  // Regular expression
          function(str) { // Method
            if (str && str.indexOf('api3') >= 0) return true; // The data is not reported.
            return false; // The data is reported.
          }]
        }
      });
```

ignoreErrors

The ignoreErrors attribute is used to ignore JS errors that comply with a specified rule. The value of this attribute can be a string (the `String` type), a regular expression (the `RegExp` type), a method (the `Function` type), or an array that includes the preceding three types. Example:

```
__bl.setConfig({
        ignore: {
          ignoreErrors: [
          'test error', // Character string
          /^Script error\.? $/, // Regular expression
          function(str) { // Method
            if (str && str.indexOf('Unknown error') >= 0) return true;   // The error is not reported.
            return false;   // The error is reported.
          }]
        }
      });
```

ignoreResErrors

The ignoreErrors attribute is used to ignore resource errors that comply with a specified rule. The value of this attribute can be a string (the `String` type), a regular expression (the `RegExp` type), a method (the `Function` type), or an array that includes the preceding three types. Example:

```
__bl.setConfig({
        ignore: {
          ignoreResErrors: [
          ' http://xx/picture.jpg ', // String
          /jpg$/, // Regular expression
          function(str) { // Method
            if (str && str.indexOf('xx.jpg') >= 0) return true;   // The error is not reported.
            return false;   // The error is reported.
          }]
        }
      });
```

[Back to the top]

disabled

Parameter	Type	Description	Required	Default Value
disabled	Boolean	Specifies whether to disable the log reporting function.	No	false

[Back to the top]

sample

Parameter	Type	Description	Required	Default Value
sample	Integer	The log sampling configuration. The value is an integer ranging from 1 to 100. For performance logs and successful API logs, follow the steps in 1/sample The proportional sampling of. For more information about metrics descriptions of performance logs and successful API logs, see Statistical metrics.	No	1

You can configure this parameter to randomly sample and report performance logs and successful API logs to reduce the number of reports and the load. ARMS restores the data based on the sampling configuration when ARMS processes the reported logs in the background. Therefore, this parameter does not affect the reporting of other types of logs such as JS error logs and failed API logs.

The default value of `sample` is `1`. The valid values of this parameter are `1`, `10`, and `100`, which separately indicate the sampling rates of 100%, 10%, and 1%. The sampling rate can be calculated using the following formula: `1/Value of sample`.

Warning Sampling is performed randomly. If the number of reported items is small, this parameter may cause large statistical result errors. We recommend that you use this parameter for the websites whose daily average PV is more than 1 million.

[Back to the top]

sendResource

Parameter	Type	Description	Required	Default Value
sendResource	Boolean	Reports static resources on a page.	No	false

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

The following code provides an example on how to configure the sendResource parameter:

```
<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 value of this parameter is checked when the load event of the page is triggered. Therefore, configure sendResource in config as shown in the preceding example. Do not use the setConfig method because this method may check the value of this parameter after the load event of the page is complete. In this case, the configuration of this parameter does not take effect.

[Back to the top]

useFmp

Parameter	Type	Description	Required	Default Value
useFmp	Boolean	Collect FMP(First Meaningful Paint, First valid rendering) data from the First screen.	No	false

[Back to the top]

enableLinkTrace

Parameter	Type	Description	Required	Default Value
enableLinkTrace	Boolean	For more information about tracing frontend and backend links, see Use the front-to-back tracing feature to diagnose causes of API errors.	No (supported only in Web scenarios, Alipay applets, WeChat applets, and DingTalk applets.)	false

[Back to the top]

release

Parameter	Type	Description	Required	Default Value
release	String	The version of the application. We recommend that you configure to view the reports of different versions.	No	undefined

[Back to the top]

environment

Parameter	Type	Description	Required	Default Value
environment	String	The environment field. Valid values: prod, gray, pre, daily, and local, where: • prod indicates an online environment. • gray indicates a phased-release environment. • pre indicates a staging environment. • daily indicates a daily environment. • local indicates a local environment.	No	`prod`

[Back to the top]

behavior

Parameter	Type	Description	Required	Default Value
behavior	Boolean	Whether to record the error user behavior to facilitate troubleshooting.	No (supported only in Web scenarios and mini program scenarios)	The Browser defaults to `true`. The mini program defaults to `false`.

[Back to the top]

autoSendPerf

Parameter	Type	Description	Required	Default Value
autoSendPerf	Boolean	Specifies whether to allow automatic sending of performance logs.	No	`true`

[Back to the top]

c1\c2\c3

In addition to the preceding parameters, you may need to specify more information to handle business problems. Therefore, ARMS SDK provides three parameters for which you can customize fields. After you configure a custom parameter, the fields of the parameter are included in each reported log.

Parameter	Type	Description	Required	Default Value
c1	String	Custom service field. Each log carries the field.	No	None
c2	String	Custom service field. Each log carries the field.	No	None
c3	String	Custom service field. Each log carries the field.	No	None

[Back to the top]

Related information

- API reference
- Page data reporting of SPAs
- Pre-report data

7.API reference

The monitoring SDK provides API operations such as data reporting operations and method operations that are used to modify SDK configurations.

Operations in this topic

- Data reporting operations: api() | error() | sum() | avg() | reportBehavior() | performance()
- Method operations: setConfig() | setPage() | addBehavior()

api()

You can call the api() operation to report the success rate of API calls on the page.

By default, the SDK listens for AJAX requests on the page and calls this operation to report. If data on the page is requested by 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 operation, we recommend you set disabledHook to *true* in SDK configuration items. For more information, see disableHook.

api() syntax:

```
__bl.api(api, success, time, code, msg)
```

api() request parameters

Parameter	Type	Description	Required	Default value
api	String	The name of the operation.	Yes	None
success	Boolean	Specifies whether the call is successful.	Yes	None
time	Number	The time consumed by the operation call.	Yes	None
code	String/Number	The returned code.	No	''
msg	String	The response information.	No	''

Example of api()

```
var begin = Date.now(),
  url = '/data/getTodoList.json';
$.ajax({
  url: url,
  data: {id: 123456}
}).done(function (result) {
  var time = Date.now() - begin;
  // Reports that the operation call is successful.
  window.__bl && __bl.api(url, true, time, result.code, result.msg);
  // do something ....
}).fail(function (error) {
  var time = Date.now() - begin;
  // Reports that the operation call fails.
  window.__bl && __bl.api(url, false, time, 'ERROR', error.message);
  // do something ...
});
```

[Back to the top]

error()

You can call the error() operation to report 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 operation 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.

error() syntax:

```
__bl.error(error, pos)
```

error() request parameters

Parameter	Type	Description	Required	Default value
error	Error	The JS error object.	Yes	None
pos	Object	The location where the error occurs. It contains the following three attributes.	No	None
pos.filename	String	The name of the file where the error occurs.	No	None
pos.lineno	Number	The number of the row where the error occurs.	No	None

Parameter	Type	Description	Required	Default value
pos.colno	Number	The number of the column where the error occurs.	No	None

Example 1 of error(): Listen to and report JS errors on the page.

```
window.addEventListener('error', function (ex) {
  // Event parameters usually contain position information.
  window.__bl && __bl.error(ex.error, ex);
});
```

Example 2 of error(): Report a custom error.

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

[Back to the top]

sum()

You can call the sum() operation to customize the reported logs. The logs are used to count how many times a specific business event occurs. You can view the following data reported by sum() on the **Custom Statistics** page:

- The trend graph of the custom events.
- The page views (PVs) and unique visitors (UVs) of an event.
- The dimension distribution information.

sum() syntax:

```
__bl.sum(key, value)
```

sum() request parameters

Parameter	Type	Description	Required	Default value
key	String	The name of the event.	Yes	None
value	Number	The number of reports at a time.	No	1

Example of sum():

```
__bl.sum('event-a');
__bl.sum('event-b', 3);
```

[Back to the top]

avg()

You can call the avg() operation to customize the reported logs. The logs are used to count how many times a specific event occurs on average in a business scenario. You can view the following data reported by avg() on the **Custom Statistics** page:

- The trend graph of the custom events.
- The PVs and UVs of an event.
- The dimension distribution information.

avg() syntax:

```
__bl.avg(key, value)
```

avg() request parameters

Parameter	Type	Description	Required	Default value
key	String	The name of the event.	Yes	None
value	Number	The number of reported items.	No	0

Example of avg():

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

[Back to the top]

addBehavior()

You can call the addBehavior() operation to add 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. You can call the addBehavior() operation to add a custom user behavior to the end of the current behavior queue. When a JS error occurs, the operation reports the current behavior queue and clears the queue.

> **Note** To call this operation, you must set behavior to *true* in SDK configuration.

addBehavior() syntax:

```
__bl.addBehavior(behavior)
```

addBehavior() request parameters

Parameter	Type	Description	Required	Default value

data	Object	The behavior data. Valid values: • name: the behavior name of the string type. The name can be up to 20 characters in length. This parameter is required. • message: the behavior content of the string type. The value can be up to 200 characters in length. This parameter is required.	Yes	None
page	String	The page where the behavior happens.	No	Value of location.pathname

Example of addBehavior():

```
_bl.addBehavior({
 data:{name:'string',message:'sting'},
 page:'string'
})
```

[Back to the top]

reportBehavior()

You can call the reportBehavior() operation to report the current behavior queue immediately.

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

> **Note** To call this operation, you must set behavior to *true* in SDK configuration.

reportBehavior() syntax:

```
__bl.reportBehavior()
```

reportBehavior() does not have request parameters.

[Back to the top]

performance()

> **Notice** This operation is applicable only to web clients.

After the onLoad operation is performed, performance() is called to report custom performance metrics other than the default performance metrics.

> **Note** You must call this operation after the onLoad operation is performed. Otherwise, the call fails because the collection of default performance metrics is incomplete. The operation can be called only once during each PV.

Usage of performance():

1. Set the SDK configuration item autoSendPerf to *false* to disable automatic reporting of performance metrics and wait for manual reporting.
2. Call the __bl.performance(Object) method to manually report custom performance metrics. In this process, the default performance metrics are automatically reported.

Example 1 of performance() (accessed by using CDN):

```
window.onload = () => {
 setTimeout(()=>{
  __bl.performance({cfpt:100, ctti:200, t1:300, ···}) ;
 }, 1000); // Set a delay to ensure that the default performance data is collected.
};
```

Example 2 of performance() (accessed by using npm packages):

```
const BrowserLogger = require('alife-logger');
const __bl = BrowserLogger.singleton({pid:'Unique site ID'});
window.onload = () => {
 setTimeout(()=>{
  __bl.performance({cfpt:100, ctti:200, t1:300, ···}) ;
 }, 1000); // Set a delay to ensure that the default performance data is collected.
};
```

> **Note** Descriptions of custom performance metrics:
>
> - cfpt: the custom first paint time
> - ctti: the first custom time to interact
> - t1 to t10: ten custom performance metrics

[Back to the top]

setConfig()

You can call the setConfig() operation to modify some configuration items after SDK initialization. For more information, see SDK reference.

setConfig() syntax:

```
__bl.setConfig(next)
```

setConfig() request parameters

Parameter	Type	Description	Required	Default value
next	Object	The configuration items to be modified and their values.	Yes	None

Example of setConfig(): Modify the disableHook value to disable automatic API reporting.

```
__bl.setConfig({
  disableHook: true,
  setUserGroup:function () {
    return 'YourUserGroup'
  }
});
```

[Back to the top]

setPage()

You can call the setPage() operation to reset the page name of a page. PV is reported again by default. This method is typically used for single-page applications. For more information, see Page data reporting of SPAs.

setPage() syntax:

```
__bl.setPage(page, sendPv)
```

setPage() request parameters

Parameter	Type	Description	Required	Default value
page	String	The new page name.	Yes	None
sendPv	Boolean	Specifies whether to report PV. PV is reported by default.	No	`true`

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

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

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

```
__bl.setPage('homepage', false);
```

[Back to the top]

FAQ

Q: What do I do if I am not sure whether the SDK has been loaded when I call the __bl.performance() operation?

A: For more information, see Page data reporting of SPAs.

[Back to the top]

Related information

- SDK reference
- Page data reporting of SPAs

8.Statistical metrics

This topic describes the key statistical metrics on each page in application monitoring of Application Real-Time Monitoring Service (ARMS) and the fields in ARMS application monitoring logs.

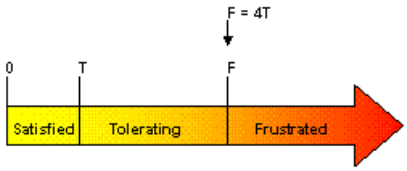
Satisfaction

Application Performance Index (APDEX) is a global standard for evaluating application performance. The user experience of an application can be evaluated based on APDEX by three levels:

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

The following formula is used to calculate the APDEX score:

```
APDEX score = (Number of satisfied samples + Number of tolerating samples/2)/Number of total samples
```



The preceding image is obtained from apdex.org.

The First Paint Time (FPT) is used as the metric to evaluate the APDEX of ARMS. The default value of T is 2 seconds.

JS stability

In ARMS, JS stability is evaluated by the JS error rate of pages.

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

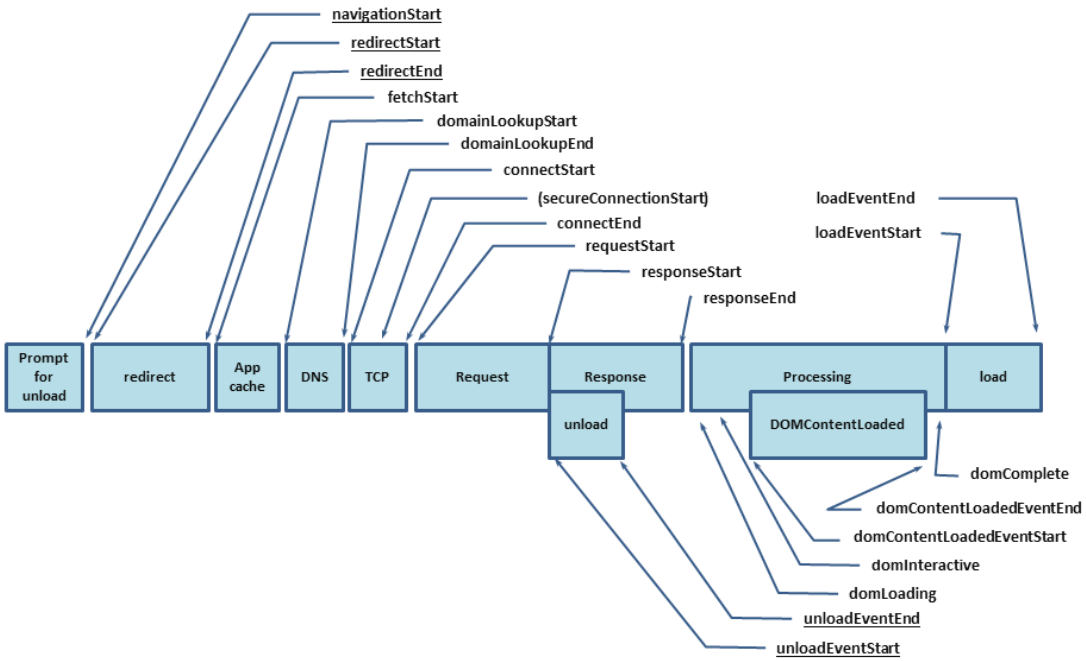
```
Error rate = Number of error samples/Number of total samples
```

In addition to the JS errors that are automatically reported, page exceptions also include errors reported when you manually call methods described in API reference.

Access speed

In ARMS, access speed is evaluated by the *FPT* of a page.

All performance or speed statistics are collected by using the Navigation Timing API defined by the World Wide Web Consortium (W3C).



The preceding image is obtained from www.w3.org.

Key performance metrics of web pages

Reported field	Description	Calculation method	Remarks
First Meaningful Paint (FMP)	The First Meaningful Paint.	For more information, visit Technical solution of FMP.	None
fpt	The FPT.	responseEnd - fetchStart	This field indicates the duration from the time when a request is initiated to the time when the browser begins to parse the bytes of the first batch of HTML documents.
tti	The time to interact (TTI).	domInteractive - fetchStart	This field indicates the time when the browser starts to load resources after it resolves all HTML documents and constructs the DOM.

Reported field	Description	Calculation method	Remarks
ready	The time it takes to complete HTML loading, which is the time it takes to construct the DOM.	domContentLoadEventEnd - fetchStart	If a JavaScript (JS) script is executed synchronously on the page, the execution time of the JS script can be calculated based on the following formula: Execution time of the JS script = ready - tti.
load	The time it takes to completely load the page.	loadEventStart - fetchStart	This field can be calculated based on the following formula: load = fpt + dom + (ready - tti) + res.
firstbyte	The time it takes to generate the first response packet.	responseStart - domainLookupStart	None

Fields that describe the amount of time consumed in each phase

Reported field	Description	Calculation method	Remarks
dns	The amount of time consumed for DNS query.	domainLookupEnd - domainLookupStart	None
tcp	The amount of time consumed for TCP connection.	connectEnd - connectStart	None
ttfb	The time to first byte (TTFB), which indicates the amount of time it takes to respond to a request.	responseStart - requestStart	TTFB can be calculated in multiple ways. For more information about how TTFB is calculated in ARMS, visit Google development definition.
trans	The amount of time consumed for data transmission.	responseEnd - responseStart	None
dom	The amount of time consumed for document object model (DOM) resolution.	domInteractive - responseEnd	None
res	The amount of time consumed for resource loading.	loadEventStart - domContentLoadedEventEnd	This field indicates the amount of time consumed for synchronously loading resources on the page.

Reported field	Description	Calculation method	Remarks
ssl	The amount of time consumed for SSL connection.	connectEnd - secureConnectionStart	This field is valid only when HTTPS is used to transmit data.

Key performance metrics of mini programs

Reported field	Description	Calculation method	Remarks
fpt	The FPT.	onShow (first page) - onLaunch (app)	This field indicates the duration from the time when the mini program calls the onLaunch method to the time when the mini program calls the onShow method to display the first page.

API call success rate

API call success rate = Number of successful API calling samples/Number of total API calling samples

In addition to the AJAX requests that are automatically reported, samples for API call success rate statistics also include the data reported when you manually call the methods described in API reference.

Log fields

The following tables describe the fields included in ARMS logs.

Common fields

Field	Description
uid	The user ID.
username	The user name.
release	The version of the application.
environment	The production environment.
page	The page.
sampling	The sampling rate.
tag	The custom tag.

API

Field	Description
api	The URL of the API request without parameters.
msg	The responseText, which indicates the response string to the API request.
code	The HTTP status code.
time	The time consumed by the API operation.
success	Indicates whether the API request is successful.

JS error

Field	Description
msg	The error message.
stack	The stack where the error occurs.
cate	The error type.
file	The file where the error occurs.
line	The line where the error occurs.
col	The column where the error occurs.
times	The number for which the error occurs.

Log overview

Logs for multi-dimensional analysis

Log type	Type	Query field (Common metric fields can be used to query or filter all types of logs)
PV log	PV	• PV • UV
Performance log	Perf	Performance metrics
Slow loading log (Performance logs in which the loading time is longer than 8 seconds)	RES	Performance metrics

Log type	Type	Query field (Common metric fields can be used to query or filter all types of logs)
JS error log	Error	• The JS error message • The URL of the JS error file • The JS error type
API log	API	• The API name • The API message • The HTTP status code • The time consumed by the API operation • The domain name of the API operation • Whether the API operation is successful • TraceID
SUM log	SUM	Custom key: event name (example: scroll-count)
AVG log	AVG	Custom key: event name (example: scroll-time)
Resource error log	ResourceError	The resource error
None	Custom	None

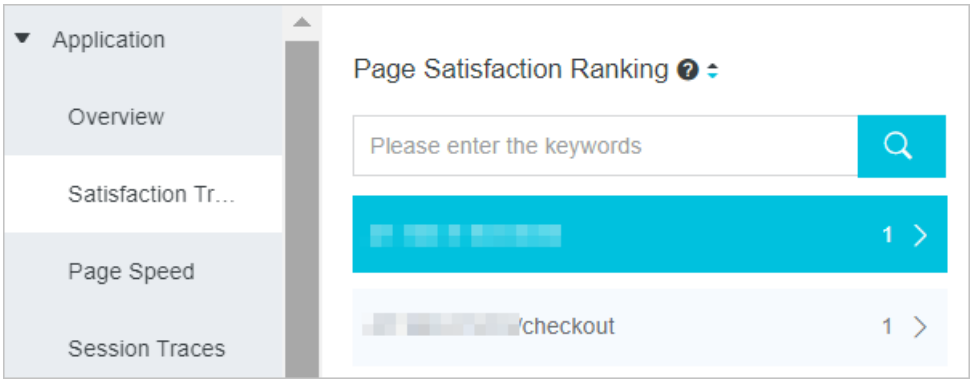
9.Browser monitoring FAQ

This topic provides answers to commonly asked questions about browser monitoring.

Overview

Why do I see asterisks (*) in the names of some monitored pages or API operations?

The page statistics of Application Real-Time Monitoring Service (ARMS) browser monitoring are retrieved based on actual page URLs and calculated by dimension. The asterisks (*) included in the names of monitored pages or API operations are not a part of the page URLs. The asterisks (*) indicate the result of URL convergence. Therefore, a name that includes an asterisk (*) is not a specific URL, but a group of similar URLs.



How URL convergence works

- Problem: Variables make it difficult to monitor or analyze similar URLs.
- Objective: Group similar URLs by replacing variables with asterisks (*).

- Solution: Use the Alibaba Cloud proprietary URL convergence algorithm to group similar URLs and decrease the number of URLs. This way, you can keep as much semantic information as possible and decrease the number of URLs. This is done in two steps:
  - Aggregation: Aggregate similar URLs into one group.
  - Variable identification: Extract the variables from the URLs in the same group, and replace the variables with asterisks (*).

The following figure shows the URL convergence process.

Solution

For information about how to disable URL convergence, see urlHelper.

[Back to the top]

Why is the page view list different from the page speed list?

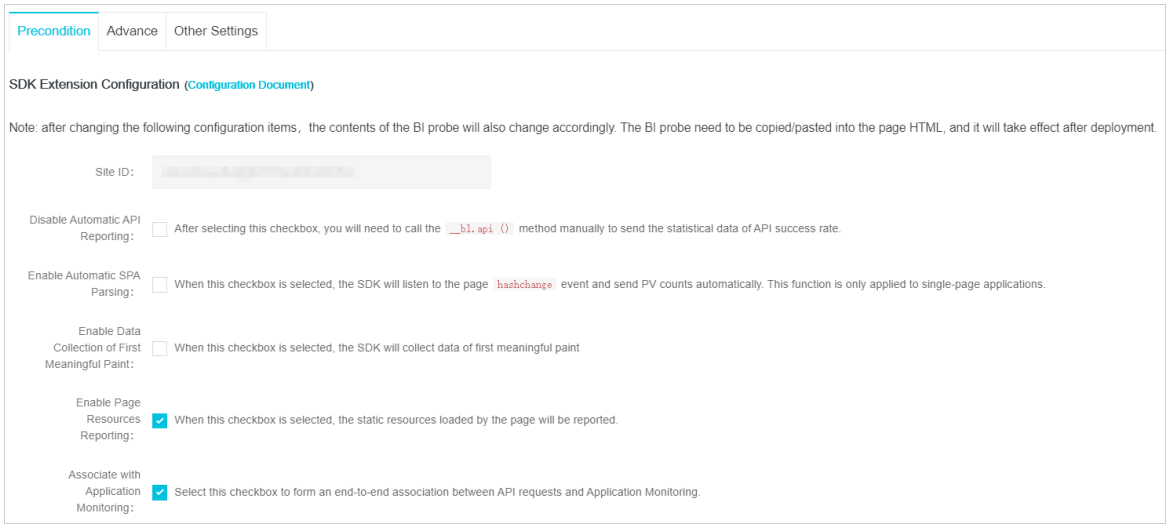
This is because your application is a single-page application (SPA), and the SPA auto-resolution is enabled. In the SPA application scenario, page views and page speed are measured by using the following methods:

- Page views: When a hashchange event is triggered, the page views are automatically reported to record the page views of the page based on the hash value. Therefore, when you view the page view list of an SPA, you can view the exact page views of the hash pages.
- Page speed: When the hash value of an SPA application changes, the page speed does not change. Therefore, the page speed is not recorded based on the hash. This way, unnecessary reports are avoided and page performance is clear.

[Back to the top]

TraceId is not found in the API logs, and as a result, the system cannot redirect me to the application monitoring page. Why?

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the target application.
3. In the left-side navigation pane, choose **Settings > Application Settings**.
4. On the **Precondition** tab, check whether **Associate with Application Monitoring** is selected for the ARMS agent configuration items. If Associate with Application Monitoring is not selected, select it. Then, connect the ARMS agent to the frontend application again.



Check whether the TraceId parameter is generated in the API logs. If no TraceId is generated, perform the operations described in this section.

5. Check whether the domain name used in your page request is the same as the domain name used in your API request. If the domain name used in your page request is different from the domain name in your API request, this is a cross-domain access. In this case, TraceId cannot be generated to prevent API request failure caused by cross-domain authentication.For information about how to solve this problem, see Front-to-back tracing.

[Back to the top]

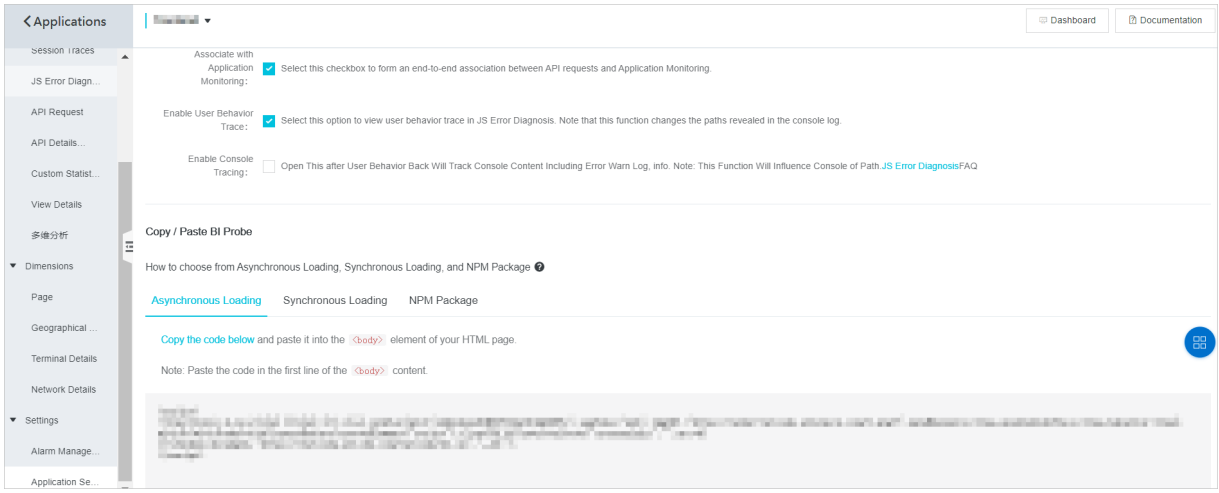
Why is the Source Map file error displayed when I am troubleshooting JS errors?

1. Make sure that the file suffix is *.js.map*.
2. Make sure that your account has the permissions to write data to ARMS. If your account has no write permissions, contact your administrator.

[Back to the top]

What are the differences between console setting and SetConfig?

Console setting can accelerate only the generation of configuration code, and the generated code takes effect only after the code is published. However, modifications to SetConfig immediately take effect.



In addition, console setting is valid only when a project is connected to SDK code. After the project is connected to SDK code, you must use SetConfig to modify configurations.

[Back to the top]

How do I configure the environment and version in an SDK?

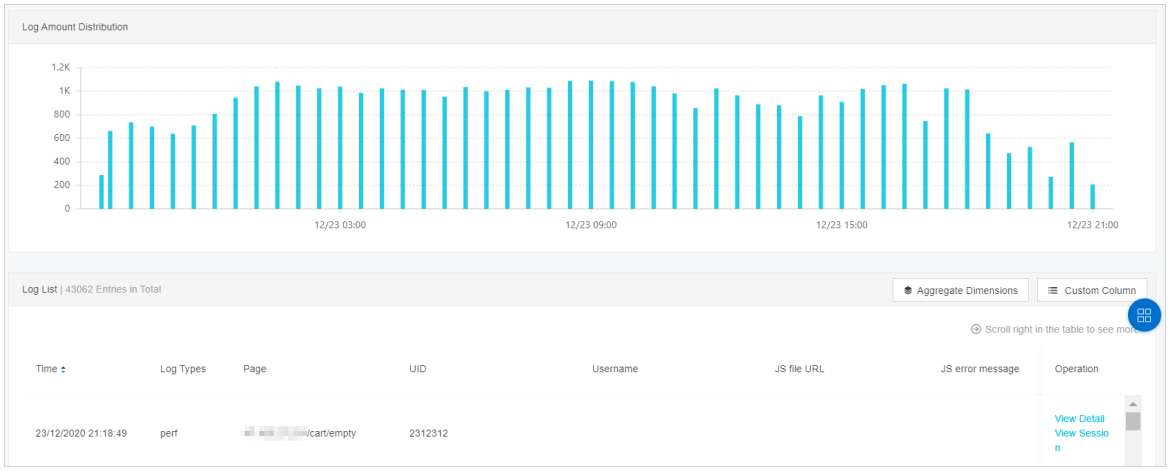
You can set the release parameter to compare versions. For more information, see release. You can also set environment to distinguish different environments. For more information, see .

- prod indicates an online environment.
- gray indicates a phased-release environment.
- pre indicates a staging environment.
- daily indicates a daily environment.
- local indicates a local environment.

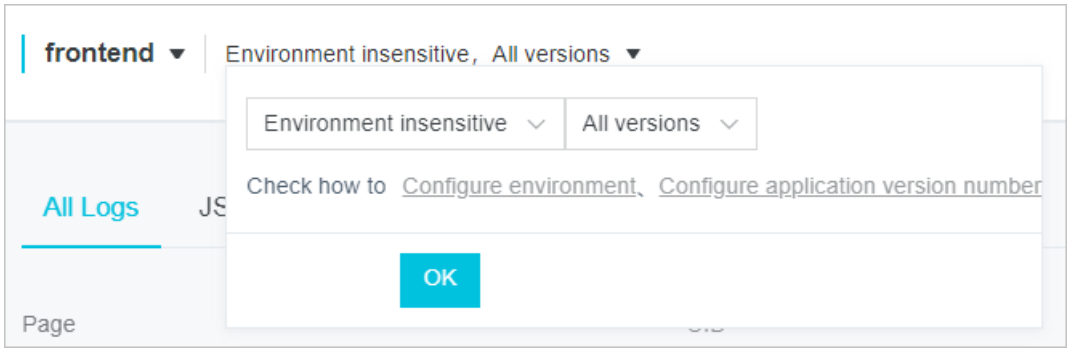
[Back to the top]

How do I view the version number of the configuration?

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the target application.
3. In the left-side navigation pane, choose **Application > View Details**.
   The version number of a log is displayed in the **Version Number** column in the **Log List** section.



4. You can also filter logs by environment and version in the menu bar. You can filter logs only after PV logs are set with the version number.



[Back to the top]

How do I view the time on page (TP) of a user for a page?

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the target application.
3. In the left-side navigation pane, choose **Application > Session Traces**.
4. In the Session List section, find the target session and click its ID in the **Session ID** column. The **Session Details** page appears.
5. Move the pointer over the **Timeline** area in the Visit Timeline column to view the time on page of a user for a page.

[Back to the top]

How do I view the browser custom performance metrics of ARMS?

1. Log on to the ARMS console .
2. In the left-side navigation pane, click **Browser Monitoring**. On the **Browser Monitoring** page, click the name of the target application.
3. In the left-side navigation pane, choose **Application > Page Speed**.
4. Custom performance metrics are displayed in the **Page Speed** section.

[Back to the top]

What do I do if the ARMS configuration does not take effect?

The possible reason is that the browser cache is not updated. In the left-side navigation pane, click View Details. Switch to the latest version to view the trend chart. If the version number is not configured, you can configure the release parameter in the SDK. For more information, see release. After the release parameter is configured, check whether the value that you specified is displayed as the latest version.

[Back to the top]

Why are JS errors on mini programs not reported?

This may be because error messages are captured by try catch at the underlying layer of the mini programs, and the error messages fail to upload. You can try to manually report error messages. For more information, see API reference.

[Back to the top]

Can JS errors of console.error be listened to?

- Yes, JS errors of console.error can be listened to. Web browsers report the error messages that meet the format of JS errors.
- On the mini program side, you can try to manually report error messages. For more information, see API reference.

[Back to the top]

In the Weex environment, why does a UID set in a mini program not take effect?

- If you have not called SetConfig, check whether you specified a UID during initialization configuration. If you did not specify a UID, specify one.

- If you have called SetConfig, re-specify a UID in SetConfig.

[Back to the top]

How long can logs be stored?

Logs can be stored for up to one month.

[Back to the top]

After I activate the expert edition, can I continue to use sites that are created during the trial period?

- No, you cannot continue to use the sites. Within 15 days after the trial period expires, the sites are suspended due to overdue payments. You can try to restart the application.
- If the expert edition is not activated for the sites within 15 days after the trial period expires, the sites are deleted to save your computing and storage resources. Relevant resources are released and the data on the resources cannot be recovered.

[Back to the top]

What version numbers are used by application edition and host edition of ARMS?

- The application edition uses the version number of the current online project. You can configure the release parameter of the SDK to specify the version number of application edition. For more information, see release.
- The host edition uses the version number of the app where the current project resides. The version number of the host edition is automatically obtained by the SDK. The version of a hosted app cannot be resolved. Only versions of Taobao, Alipay, or WeChat are resolved.

[Back to the top]

Why are the page views on the same page different among modules in the console?

On the Page Speed page, the number of page views is equal to the product of entries in the performance log and sample rate.

On the Page page under Dimensions, the number of page views is equal to the value shown in the PV log.

Automatically reported performance logs are reported only after pages are loaded. A performance log is reported each time the page is refreshed.

After the SPA mode is enabled, PV logs are reported every time routes are switched. In the SPA mode, the number of performance logs is less than the number of PV logs, which results in a large difference in the page views for different modules.

[Back to the top]

Why is the value of duration less than the value of connect download?

The performance data loaded to ARMS resources is obtained from `performance.getEntriesByType('resource')`. The domain name of a third party ARMS resource must be the same as the domain name of the site of the current requested resource. By default, the value of 0 is obtained from `performance.getEntriesByType('resource')` for the following parameters of the performance data of cross-domain resources:

```
redirectStart
redirectEnd
domainLookupStart
domainLookupEnd
connectStart
connectEnd
secureConnectionStart
requestStart
responseStart
```

Some time properties may be inaccurate or abnormal because the preceding parameters are used in the calculation. For example, in `connect download: responseEnd - responseStart`, the value of duration is less than the value of connect download because the timestamp of responseStart is 0.

- To solve this problem, for the CDN resources, you can configure the response header Timing-Allow-Origin to specify the time to obtain the resource time.
- For third-party resources, we recommend that you take the value of the duration parameter as the major reference.

[Back to the top]

10.Causes and solutions for script errors

Script errors are common errors. However, no complete error message or error stack is available for errors of this type. Therefore, it is difficult to troubleshoot these errors. This topic analyzes the causes, provides solutions, and shows you how to ignore this type of error in Application Real-Time Monitoring Service (ARMS).

Causes of script errors

Script errors are also called cross-origin errors. When a website requests and executes a script hosted under a third-party domain name, a script error may occur. The most common scenario is to use a content delivery network (CDN) to host JavaScript resources.

For better understanding, assume that the following HTML page is deployed in the *http://test.com* domain:

```
<! doctype html>
<html>
<head>
 <title>Test page in http://test.com</title>
</head>
<body>
 <script src="http://another-domain.com/app.js"></script>
 <script>
 window.onerror = function (message, url, line, column, error) {
  console.log(message, url, line, column, error);
 }
 foo(); // Invokes the foo method defined in app.js.
 </script>
</body>
</html>
```

Assume that the `foo` method invokes an undefined `bar` method:

```
// another-domain.com/app.js
function foo() {
 bar(); // ReferenceError: bar is not a function
}
```

After the page is run, the following error is captured:

```
"Script error.", "", 0, 0, undefined
```

In fact, this is not a JavaScript bug. For security reasons, browsers deliberately hide specific errors thrown by JavaScript files of other origins. This way, sensitive information can be effectively prevented from being inadvertently captured by uncontrolled third-party scripts. Therefore, only scripts from the same origin can capture specific error messages, whereas scripts from other origins can detect errors but cannot capture specific error messages. For more information, see Webkit source code.

```
bool ScriptExecutionContext::sanitizeScriptError(String& errorMessage, int& lineNumber, String& sourceURL)
  {
    KURL targetURL = completeURL(sourceURL);
    if (securityOrigin()->canRequest(targetURL))
      return false;
    errorMessage = "Script error.";
    sourceURL = String();
    lineNumber = 0;
    return true;
  }
```

The following sections describe how to resolve script errors.

Solution 1: Enable cross-origin resource sharing

To capture JavaScript errors across origins, perform the following steps:

1. Add the `crossorigin="anonymous"` attribute.

   ```
   <script src="http://another-domain.com/app.js" crossorigin="anonymous"></script>
   ```

   This step tells the browser to anonymously obtain third-party scripts. This means that the browser does not send potential user identity information, such as cookies and HTTP certificates, to the server when the browser requests scripts.

2. Add the cross-origin HTTP response header.

   ```
   Access-Control-Allow-Origin: *
   ```

   or

   ```
   Access-Control-Allow-Origin: http://test.com
   ```

   > **Note** By default, most of mainstream CDNs have the `Access-Control-Allow-Origin` attribute. The following example is from Alibaba Cloud CDN:

   ```
   $ curl --head https://retcode.alicdn.com/retcode/bl.js | grep -i "access-control-allow-origin"
   => access-control-allow-origin: *
   ```

After the preceding two steps are performed, you can capture cross-origin errors by using the `window.onerror` handler. In the preceding example, after the page is run again, the following error is captured:

```
=> "ReferenceError: bar is not defined", "http://another-domain.com/app.js", 2, 1, [Object Error]
```

Solution 2: Add the `try catch` statement (Optional)

When it is difficult to add a cross-origin attribute to the HTTP request or response header, add the `try catch` statement.

Add the `try catch` statement to the preceding HTML page:

```
<! doctype html>
<html>
<head>
 <title>Test page in http://test.com</title>
</head>
<body>
 <script src="http://another-domain.com/app.js"></script>
 <script>
 window.onerror = function (message, url, line, column, error) {
  console.log(message, url, line, column, error);
 }
 try {
  foo(); // Invokes the foo method defined in app.js.
 } catch (e) {
  console.log(e);
  throw e;
 }
 </script>
</body>
</html>
```

Run the page again. The following information is returned:

```
=> ReferenceError: bar is not defined
    at foo (http://another-domain.com/app.js:2:3)
    at http://test.com/:15:3
=> "Script error.", "", 0, 0, undefined
```

The `try catch` statement outputs complete error information in the console log, but the `window.onerror` handler outputs only script errors. You can manually report captured exceptions in the catch statement. For more information, see API reference.

```
__bl.error(error, pos);
```

Note Although the `try catch` statement can be used to capture some exceptions, we recommend that you use Solution 1.

How can I ignore script errors in ARMS?

Script errors usually come from scripts under third-party domain names. If script errors does not affect the running of your application, you can ignore them by setting the ignore parameter of ARMS Browser Monitoring SDK.

The `ignore` parameter allows you to ignore the reporting of specified JavaScript errors. The value of this parameter is an object that contains three properties: ignoreUrls, ignoreApis, and ignoreErrors. The following code shows the default value of this parameter:

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

Use the ignoreErrors property to ignore script errors. The ignoreErrors property is used to ignore JavaScript errors that comply with a specified rule. The value of this property can be a string (the *String* type), a regular expression (the *RegExp* type), a method (the *Function* type), or an array that includes the preceding three types. The following code provides an example on how to ignore script errors:

```
__bl.setConfig({
  ignore: {
    ignoreErrors: /^Script error\.? $/
  }
});
```

References

- API reference
- SDK reference