腾讯云可观测平台

终端性能监控 Pro

腾讯云

文档目录

终端性能监控 Pro
终端性能监控 Pro 简介
产品概述

最近更新时间：2025-08-05 16:18:11

终端性能监控 Pro 是一站式客户端应用质量监控与管理平台，专注于为您的客户端应用（Android、iOS、鸿蒙、Windows、Flutter 等）提供全面的崩溃分析、性能监控、异常告警能力，致力于保障应用的稳定性和流畅性，大幅降低用户流失风险，提升应用体验和品牌口碑。

终端性能监控 Pro 主要功能

多维崩溃诊断

实时捕获移动应用原生崩溃（Native）、卡顿无响应（ANR）、异常错误等关键问题，自动采集堆栈信息、线程状态、设备信息、内存快照等数据。

卡顿智能分析

深入追踪主线程阻塞问题，识别耗时函数与操作路径，定位导致应用界面卡顿、操作迟缓的根本原因。

网络性能分析

详细监控网络请求详情，包括请求耗时、成功率、数据量大小、域名解析时间、连接建立时间等，辅助优化网络交互体验。

智能根因分析

基于海量数据智能聚合崩溃点与堆栈调用路径，精准定位崩溃发生的代码文件、行号及具体方法，并关联相关系统日志、应用版本、设备型号、网络环境等信息。

趋势分析与版本对比

清晰展示崩溃率的实时趋势及变化，支持对比不同应用版本的崩溃指标，快速评估新版本稳定性。

全量日志采集

支持用户自定义日志记录上传，在崩溃发生时一并收集现场上下文信息，大幅提升问题定位效率。

核心性能指标监控

实时监控关键性能指标（KPI），包括应用启动时间、页面加载时长、帧率（FPS）、内存占用、流量消耗、网络请求耗时与成功率、电池消耗等。

符号表自动管理

集成上传与解析符号文件功能，确保崩溃堆栈的准确符号化，清晰还原可读代码信息。

应用版本管理

便捷管理各版本应用的运行数据和问题反馈。

主流平台支持

持续良好兼容 Android、iOS、Windows、Unity、Flutter、React Native 等主流开发平台与应用框架。

精细化数据筛选

支持按应用版本、设备型号、操作系统、地域、网络环境、用户标识等多维度筛选分析崩溃、卡顿、性能异常数据。

实时告警推送

支持对崩溃率激增、ANR 率超标、关键性能指标异常等设置阈值告警规则，通过邮件、短信、微信、企业微信等渠道即时推送，确保团队第一时间响应处理。

产品优势

最近更新时间：2025-08-05 16:18:11

移动端质量专家

终端性能监控 Pro 深耕移动应用领域多年，深度理解移动端独特的性能问题（如卡顿、ANR）和稳定性挑战（如崩溃、OOM）。提供专业的崩溃堆栈智能分析、卡顿监控、ANR 检测、资源监控等能力，精准定位移动应用特有的问题根源。

全面崩溃监控与分析

提供业界领先的实时崩溃监测与分析能力，支持捕获 Java、Native（C/C++）及 Unity 等引擎的崩溃信息。通过强大的错误聚合、堆栈反混淆、根因智能诊断功能，帮助开发者快速精准定位和修复崩溃问题，显著降低崩溃率。

深入性能剖析

实时监控应用的启动时间、页面渲染耗时、网络请求性能、卡顿帧率、ANR 等关键性能指标。提供耗时分析、多维度性能聚合以及用户会话轨迹追踪，清晰呈现性能瓶颈，助力应用流畅运行，提升用户体验。

实时告警与闭环管理

支持基于崩溃率、ANR 率等关键指标的智能阈值告警，支持针对新增 Issue、Top Issue 的问题粒度告警追踪，实时通过多种渠道（邮件、短信、微信、企业微信、DingTalk、WebHook 等）通知到人。提供问题指派、状态跟踪、修复验证等闭环管理功能，提升团队协作修复效率。

低成本接入与稳定服务

基于腾讯海量业务实践经验，底层 SDK 技术由持续经营十几年的 TDS-Bugly 支持，SDK 轻量级、可用性高、接入简单，对应用性能影响极小。依托腾讯云强大的基础设施，提供高可用、高吞吐的数据处理服务，开发者无需担忧运维负担即可享受稳定可靠的服务体验。提供灵活的付费模式（用量套餐 + 月活套餐），性价比突出。

应用场景

最近更新时间：2025-08-05 16:18:11

崩溃治理与稳定性保障

- 精准捕获移动端全量崩溃场景（Native 崩溃、ANR、FOOM），通过优化异常捕获链路和异常上报时机前置，尽最大程度解决「异常无上报」痛点。
- 支持 iOS/Android 多维度崩溃分析（内存分配、VMMap、进程状态），结合 minidump/tombstone 现场还原技术，可高效定位如「图片内存暴增导致闪退」等隐蔽问题。

内存疑难问题攻坚

针对 OOM 率突增、内存泄漏、指针垂悬等复杂场景，提供内存详情分析等深度分析能力：

- 自动捕获 Java 堆转储文件，分析泄漏对象引用链。
- 支持 FOOM 与内存泄漏监控联动，精准定位如 Hippy 框架升级引发的 OOM 劣化问题。

分钟级告警响应

基于云原生实时流处理架构，实现：

- 崩溃率突增/ANR 超标等关键指标分钟级告警。
- 支持 Issue 新增、TOP Issue 劣化等复杂告警场景。
- 支持多维过滤（版本/机型/系统）和防误报机制。
- 告警系统支持电话、短信、微信、企业微信、钉钉、飞书等主流告警通道，还支持 WebHook 接口，能够无缝对接客户公司内部平台，助力企业一体化响应体系构建。

全链路性能优化

- 卡顿治理：主线程/工作线程卡顿监控，结合 ANR Trace+GC 详情定位阻塞根源。
- 资源监控：流量/电量异常消耗实时预警，杜绝资源滥用。
- 启动速度优化：页面加载性能分析，识别渲染瓶颈点。

多平台发布验证

- 版本质量对比：支持 AB 测试数据下钻分析（二十余种自定义字段），精准评估新版本稳定性。
- 跨端支持：覆盖 Android/iOS/鸿蒙 NEXT/Flutter/Windows，确保多端发布质量一致性。

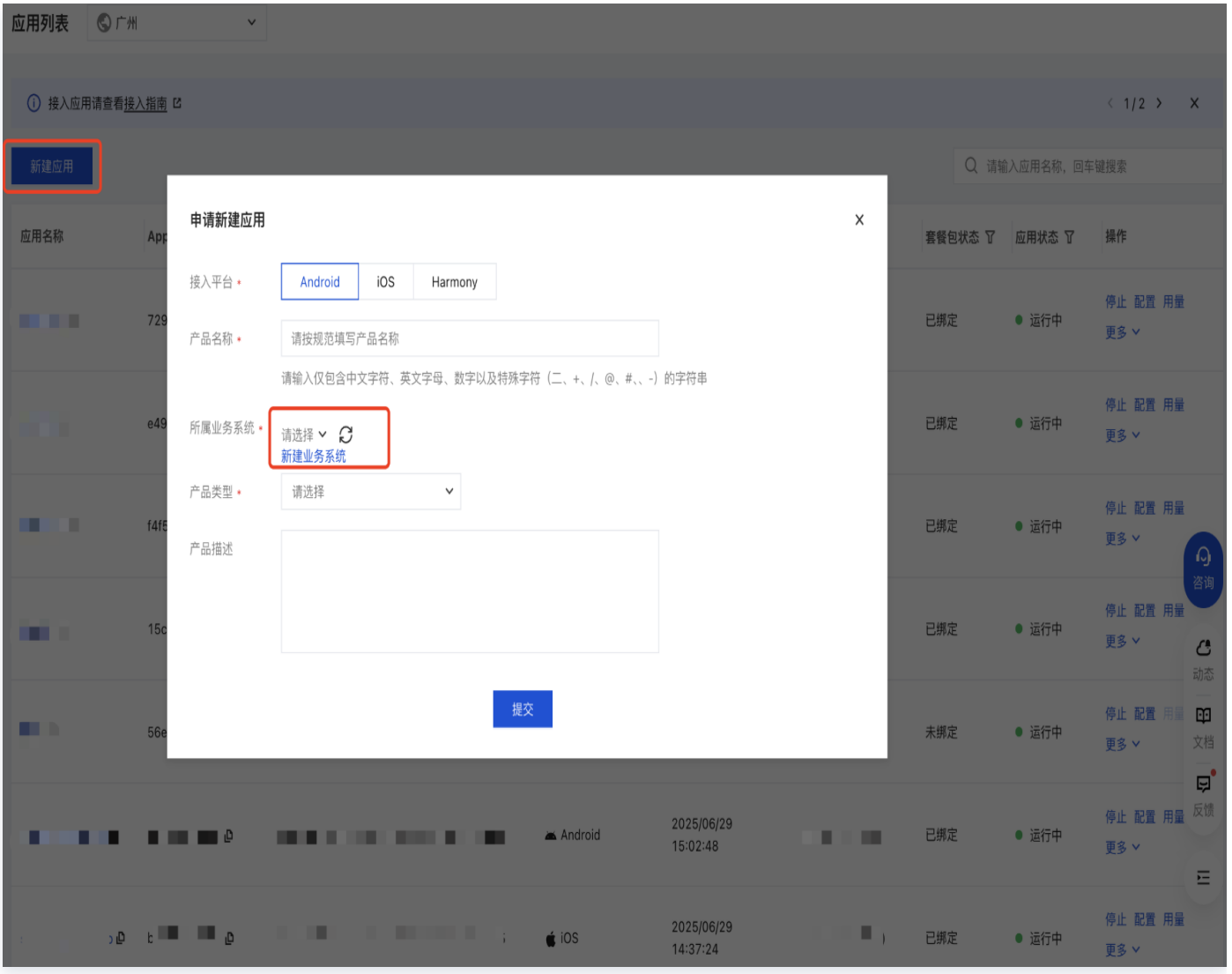
接入指南
快速接入

最近更新时间：2025-08-20 17:54:22

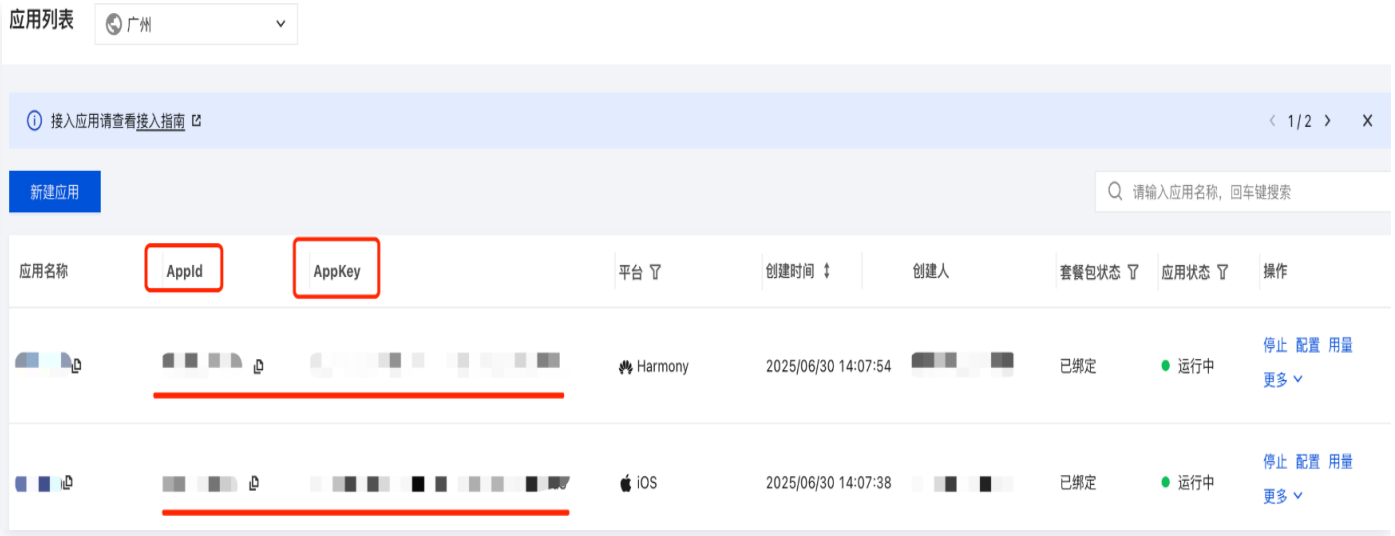
本文主要帮助您快速完成终端性能监控 Pro SDK 的接入、数据上报及验证工作。

创建应用

1. 登录 终端性能监控 Pro，进入**应用列表**页面。
2. 在应用列表页单击**新建应用**，按要求完成业务系统选择。若您当前没有业务系统，单击**新建业务系统**可前往 业务系统列表 页面完成创建，填写相关信息。



3. 成功创建应用后，可以看到平台分配的 AppID 及 AppKey，作为该产品在平台的凭证。

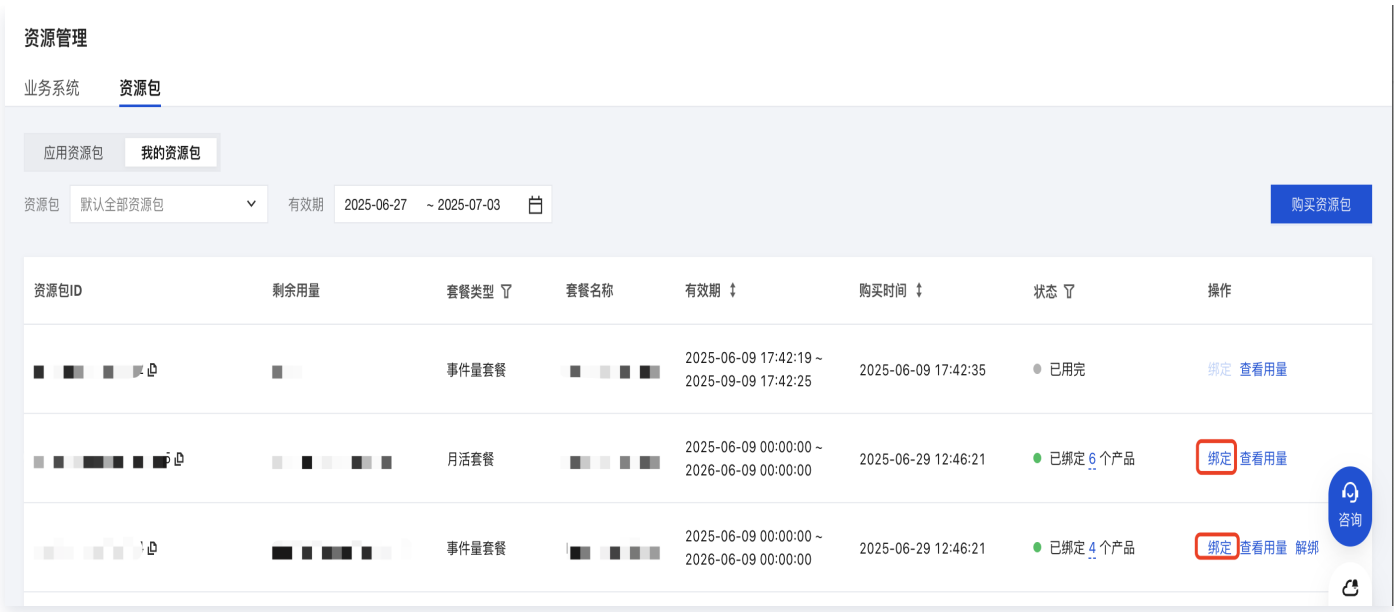


购买资源包

在终端性能监控 Pro 中，成功创建应用后，还需要为其购买并绑定资源包，资源包详情见 计费概述，才能使用新创建的应用上报数据。如果应用没有绑定生效中的资源包，或者绑定的资源包已经用完，则该应用无法上报数据。

1. 前往 资源管理 页面，选择资源包，单击**购买资源包**，资源包的详细说明请参见 计费概述 。

2. 购买完成后，单击操作列的**绑定**，绑定应用。



接入SDK

创建应用成功后，请参见 Android SDK 接入指引 、iOS SDK 接入指引 、鸿蒙 SDK 接入指引 在应用中集成终端性能监控 Pro SDK。

检查上报

应用按指引接入终端性能监控 Pro SDK 后，模拟/触发相关的事件，检查上报流程。终端性能监控 Pro 当前支持 崩溃监控 、ANR 监控 、OOM 监控 、错误监控 、性能监控（启动监控、卡顿监控、内存监控） 等功能。
不同的监控功能，数据上报逻辑有些差异。对于异常退出类的监控，默认全开启，SDK 捕获异常后，将立即上报，或者应用下次启动再上报。对于性能类的监控，默认采样开启，应用可以根据自身情况，创建配置任务，选择恰当的采样率。

崩溃监控

默认全开启，暂不支持采样。用户可以模拟崩溃，并在崩溃发生后重启应用，在 崩溃/问题列表 检查是否包含相关上报。

- Android 平台示例代码：

```
public void testNPECrash() {
    String value = null;
    System.out.println("Length of str is: " + value.length());
}
```

- iOS 平台示例代码：

```
- (void)testNPECrash {
    NSString *str = nil;
```

```
    NSInteger length = [str length];
    NSLog(@"Length of str is: %ld", length);
}
```

ANR 监控

Android 的 ANR 监控，默认全开启，暂不支持采样；iOS 的 ANR 监控，默认全开启，支持采样。
下文通过在 UI 线程执行一个长耗时任务（例如直接 sleep 20s），来模拟 ANR。因应用无响应被系统杀死后，重启应用，在 ANR/问题列表 检查是否包含相关上报。

- Android 平台示例代码：

```
public void testANR() {
    try {
        Thread.sleep(20000); // 模拟耗时操作
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- iOS 平台示例代码：

```
- (void)startLongRunningTask {
    dispatch_async(dispatch_get_main_queue(), ^{
        // 模拟长时间运行的任务
        [NSThread sleepForTimeInterval:20.0];
    });
}
```

OOM 监控

Android 的 OOM 监控，默认全开启，暂不支持采样；iOS 的 OOM 监控，默认全开启，支持采样。
下文模拟了一个内存溢出的场景，不断分配内存而且不释放，最终导致应用因 OOM 被系统杀死。重启应用，可以在 OOM/问题列表 检查是否包含相关上报。

- Android 平台示例代码：

```
public void simulateOOM() {
    List<byte[]> list = new ArrayList<>();
    while (true) {
        byte[] bytes = new byte[1024 * 1024]; // 分配1MB的内存
        list.add(bytes);
    }
```

```
}
```

- iOS 平台示例代码：

```
- (void)simulateFOOM {
    NSMutableArray *array = [NSMutableArray array];
    while (true) {
        @autoreleasepool {
            NSString *string = [NSString stringWithFormat:@"%d",
arc4random_uniform(1000000)];
            [array addObject:string];
        }
    }
}
```

错误监控

错误监控默认全开启，暂不支持采样。终端性能监控 Pro SDK 提供了一组接口，支持应用上报自己定义的异常及捕获的异常，这些异常被称为错误。
应用主动调用错误上报接口，上报数据后，无需重启应用，直接在 错误/问题列表 检查是否包含相关上报。

性能监控

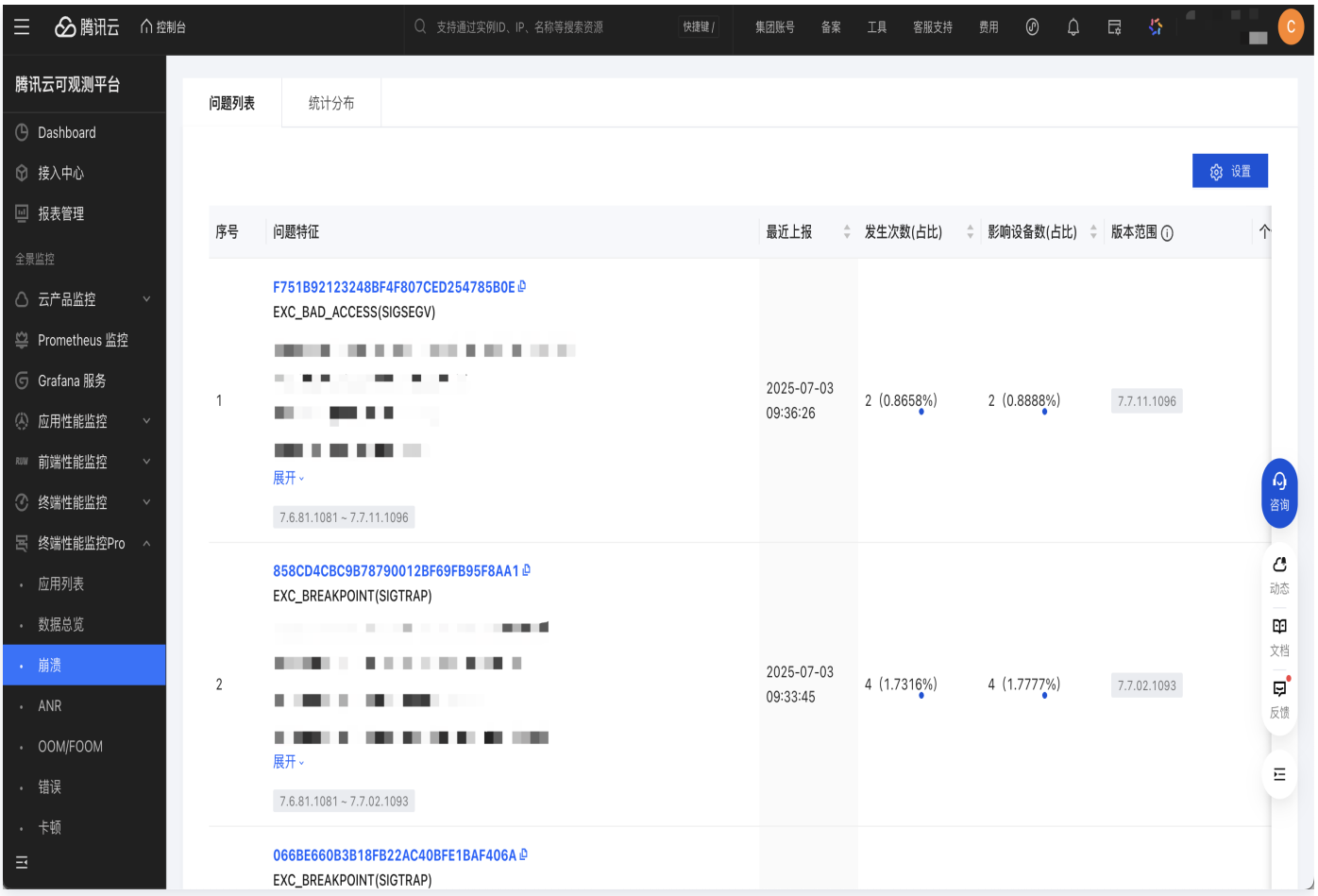
终端性能监控 Pro 当前支持的性能监控包含卡顿监控、启动监控、内存监控。性能监控都支持采样，默认不开启，需要用户在 设置/SDK 配置 中创建配置任务，指定各性能监控项的采样率。

1. 在测试阶段，为了方便测试，建议创建专门的白名单测试任务，所有监控功能设置为全采样，即"sample_ratio"以及"event_sample_ratio"设置为1。
2. 配置调整后，一般需要等待10分钟才能生效。为了确保配置生效，建议在修改配置10分钟后，开启应用，等待几分钟后再重启应用。
3. 允许开启应用启动监控后，在应用启动时，启动监控会自动开启，并在 SDK 初始化完成后，进行启动数据上报。
4. 卡顿和内存指标，开启监控后，收集应用本次运行的数据，下次启动时再上报数据。
5. 卡顿和内存的问题监控，开启监控后，当监控到相关异常个例时，wifi 条件下立即上报，非wifi条件下，会先本地缓存数据，在应用下次启动后再上报数据。

查看数据

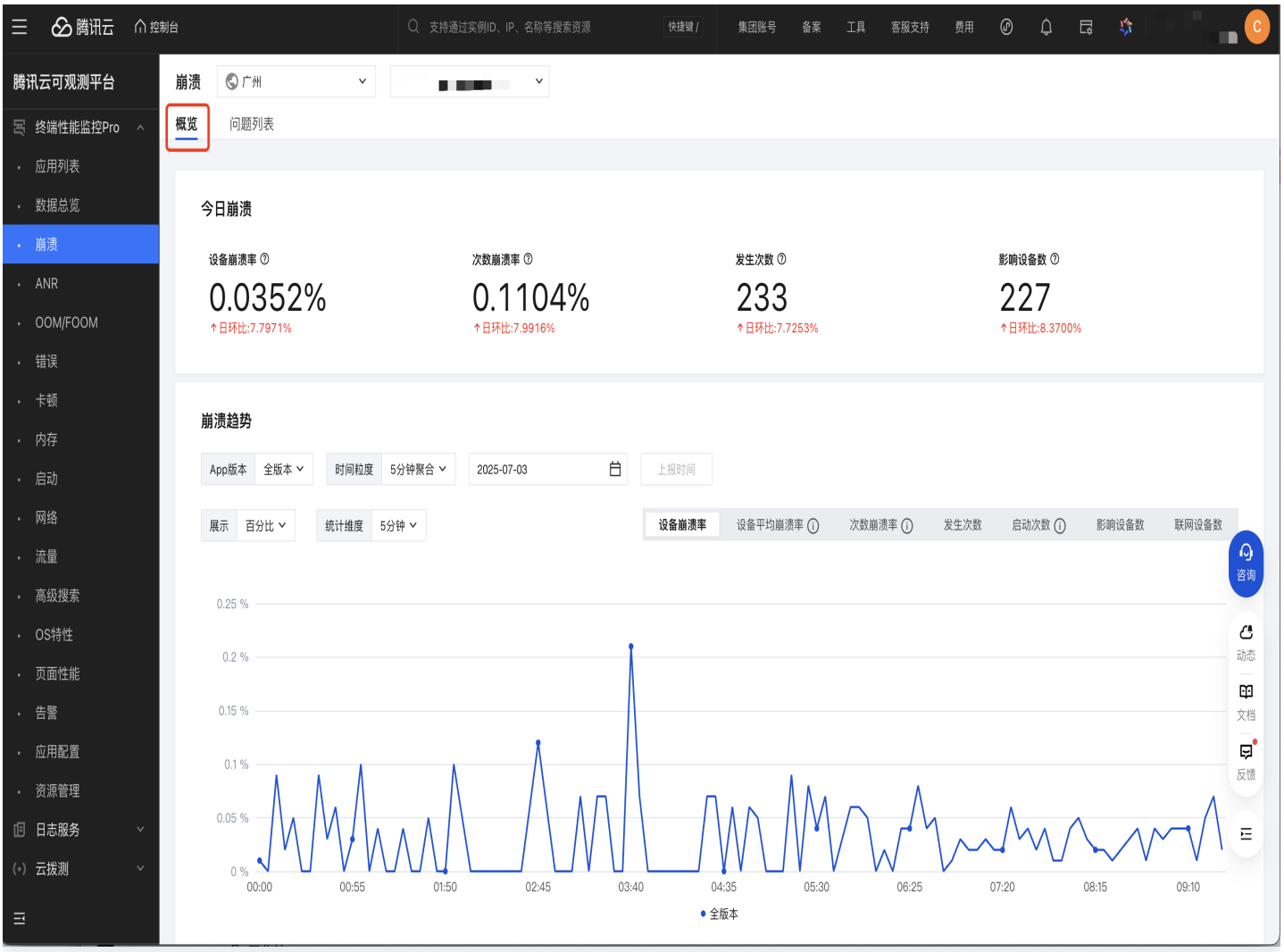
数据上报成功后，可以在 终端性能监控 Pro 的控制管理台看到相关数据。如下图所示：

1. 左边的是菜单栏，包含了各监控功能的入口。
2. 每类监控功能都包含概览、指标分析、问题列表，以及问题详情。



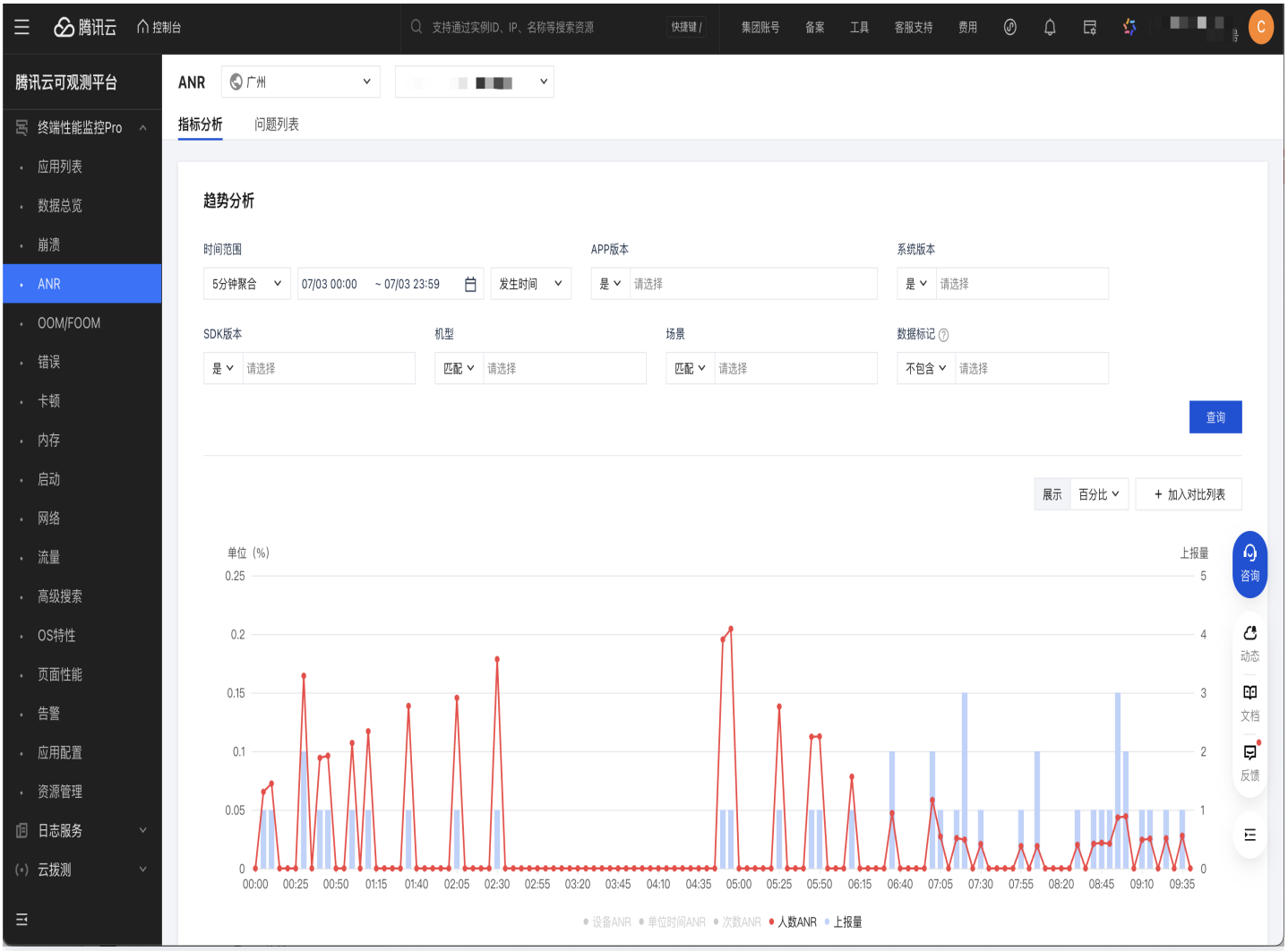
异常概览

异常概览中包含崩溃、错误、ANR 以及 OOM 等质量监控的指标分析数据。当前支持实时数据分析，异常趋势分析，以及异常的统计分布。



指标分析

指标分析包含卡顿，内存以及启动的指标分析数据。当前支持指标趋势分析、对比分析和多维分析。

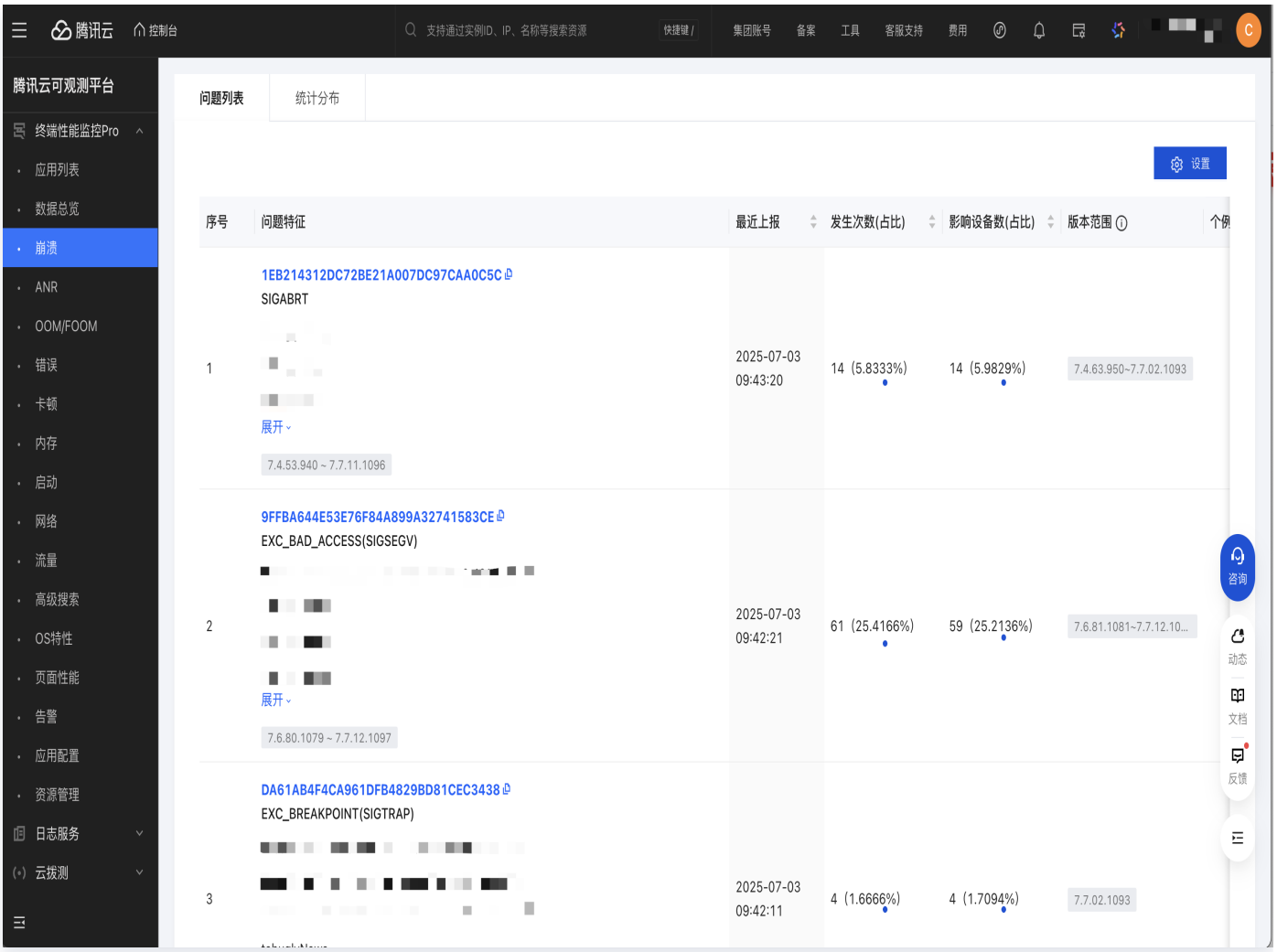


问题列表

无论是崩溃、错误、ANR 以及 OOM 等质量监控，还是卡顿、内存、启动等的性能监控，都包含问题列表。监控到异常后，SDK 会采集现场信息并上报给服务器。服务器收到数据后，处理链路会提取上报个例的特征，将相同特征的个例聚合在一起，形成 Issue（即问题）。问题列表展示这些 Issue，并提供丰富的搜索能力，方便用户分析以及查询。

问题列表的常用使用步骤：

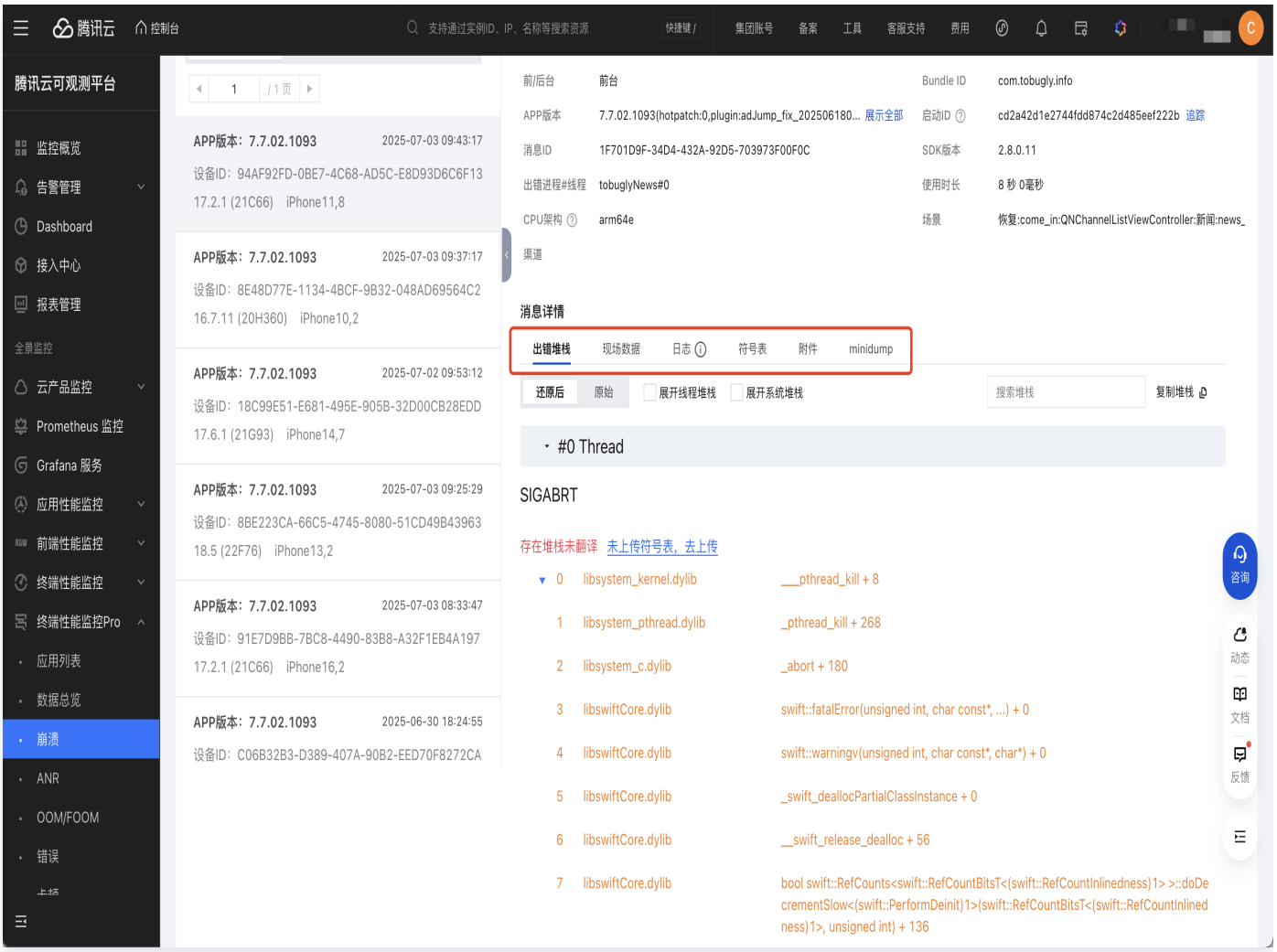
1. 根据分析需要，编辑搜索条件，用户还可以选择只显示常用的搜索条件，将不常用的条件隐藏。
2. 编辑完搜索条件后，单击查询，提交查询任务。
3. 完成搜索后，页面自动刷新，更新搜索结果。
4. 单击查看某个问题的详情。



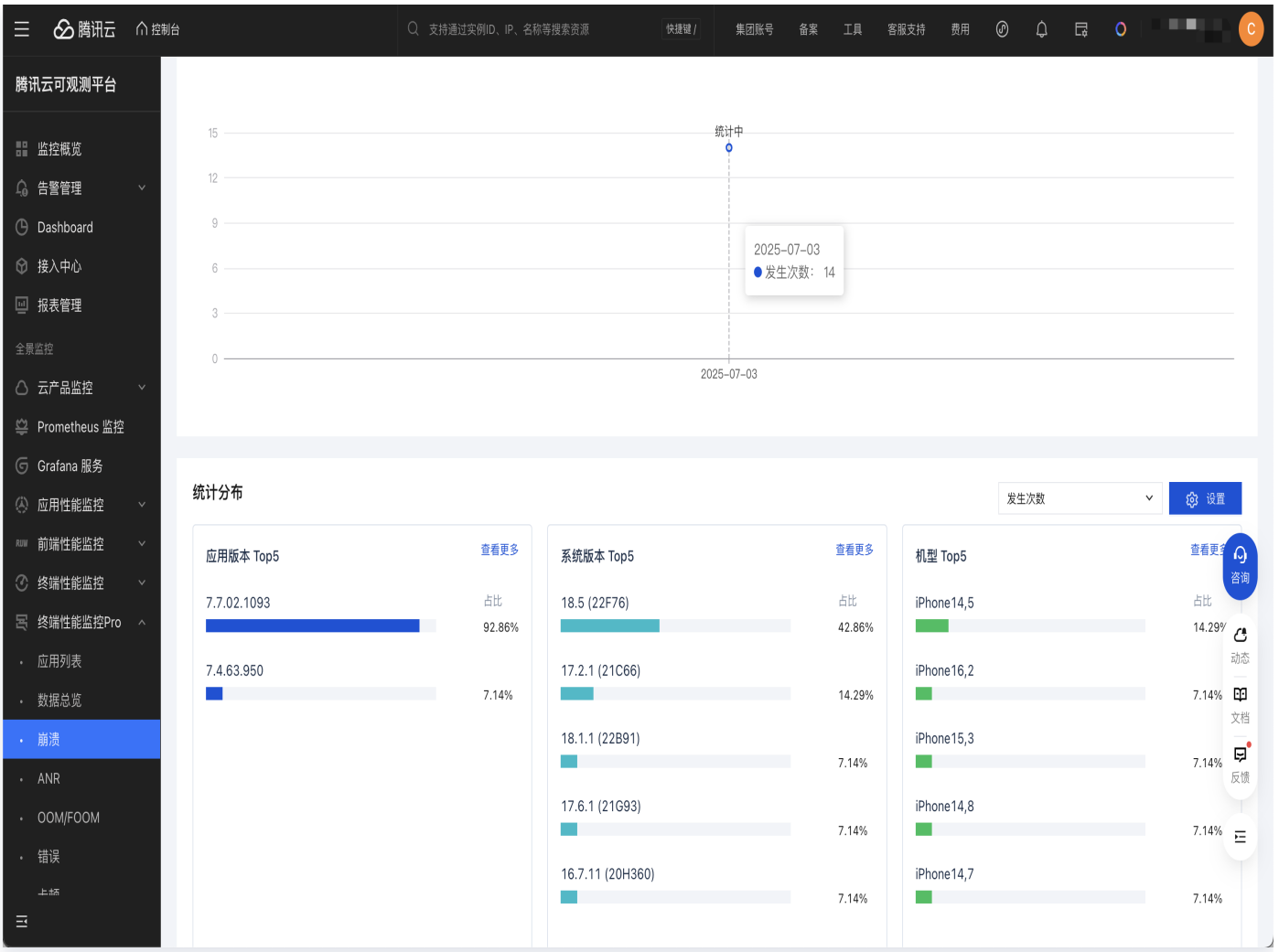
问题详情

问题详情展示一个具体 Issue 的详细情况。一般而言，我们查看问题详情是为了了解导致异常的根本原因。

1. 首先我们会分析出错堆栈。在个例详情，选择某个个例，分析其出错堆栈。
2. 如果出错堆栈还不足以帮忙我们定义问题，我们还有可能要借助日志、FD 信息、进程信息或者甚至附件中的信息。



3. 如果对于这个问题的出现还心存疑惑，我们可以通过下钻分析功能，分析其上报的趋势，以及分布特性。



Android SDK 接入指引
SDK 接入简介

最近更新时间：2025-08-14 15:29:52

终端性能监控 Pro SDK 由腾讯 Bugly 团队提供技术支持。本文介绍如何在 Android 平台上接入终端性能监控 Pro SDK。接入 SDK 后，验证数据上报成功，即可在控制管理台使用相关分析功能。

SDK 简介

- **SDK 版本**：4.4.5.6
- **SDK功能**：专业的应用质量监控工具，提供异常数据的采集和分析服务，帮助开发者及时发现并解决异常问题，打造高质量 App。
- **服务提供方**：腾讯云计算（北京）有限责任公司
- 终端性能监控 Pro SDK 合规使用指南
- 终端性能监控 Pro SDK 个人信息保护规则

接入步骤

1. 在接入 SDK之 前，请务必认真阅读 终端性能监控 Pro SDK 合规使用指南 。
2. 完成 SDK 集成 。
3. 完成 SDK 初始化 。SDK 在初始化过程中，可能会采集部分用户信息，请在用户同意 终端性能监控 Pro SDK 个人信息保护规则 之后，再进行 SDK 的初始化。在初始化 SDK 前，不会收集任何信息。
4. 验证 数据上报 。
5. 详细了解其他 SDK 功能，请参见 API 说明 。

SDK 集成

最近更新时间：2025-08-14 15:29:52

终端性能监控 Pro SDK 由腾讯 Bugly 团队提供技术支持，与 Bugly 专业版共用 SDK。支持自动集成和手动集成两种方式。

前提条件

终端性能监控 Pro 需要使用 Bugly SDK 4.4.5.6及以上版本。

自动集成（推荐）

1. 在 project 级别的 build.gradle 添加 maven 地址。

```
buildscript {
    repositories {
        maven { url 'https://repo1.maven.org/maven2/' }
    }
}

allprojects {
    repositories {
        maven { url 'https://repo1.maven.org/maven2/' }
    }
}
```

2. 在 Module 的 build.gradle 文件中添加依赖和属性配置。

```
android {
    defaultConfig {
        ndk {
            abiFilters 'armeabi-v7a', 'arm64-v8a'
        }
    }
}

dependencies {
    implementation "com.tencent.bugly:bugly-pro:4.4.5.6"
}
```

手工集成（不推荐）

如果您因特殊业务诉求、安全性要求、版本控制等原因无法采用自动集成方式集成 SDK，您也可以手动集成终端性能监控 Pro SDK，操作步骤如下。

1. 下载终端性能监控 Pro Android SDK 的 aar 文件。

   终端性能监控 Pro Android SDK 下载地址： Central Repository: com/tencent/bugly/bugly-pro 。选择合适的版本，如下内容以 SDK bugly-pro-4.4.5.6.aar 为例进行介绍。

2. Android Studio 工程中引入依赖。

   将 aar 文件复制到 module 的 libs 目录下，在 module 的 gradle 文件中添加依赖。

```
dependencies {
    // bugly sdk 的依赖
    implementation 'org.jetbrains.kotlin:kotlin-android-extensions-
runtime:1.3.41'
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.3.41'
    implementation 'com.squareup.leakcanary:shark:2.7'
    implementation 'androidx.annotation:annotation:1.2.0'
    // 手动集成
    implementation files('libs/bugly-pro-4.4.5.6.aar')
}
```

SDK 初始化

最近更新时间：2025-08-14 15:29:52

本文将为您介绍如何初始化 SDK。

示例代码

参考以下代码初始化 SDK，我们推荐尽可能早初始化，这样才能及时捕获异常。

```
public static void initBugly(Context context) {
    // 1. 初始化参数预构建，必需设置初始化参数
    String appID = "a278f01047"; // 【必需设置】产品的 APPID
    String appKey = "1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86"; // 【必需设置】产品的 APPKEY
    BuglyBuilder builder = new BuglyBuilder(appID, appKey);

    // 2. 【必需设置】设置上报域名，因为终端性能监控 Pro 与 Bugly 共用一个 SDK，需要通过上报域名来区分。

buglyBuilder.setServerHostType(BuglyBuilder.ServerHostTypeBuglyCloud);

    // 3. 基本初始化参数，推荐设置初始化参数
    builder.uniqueId = "unique_id"; // 【推荐设置】设置设备唯一 ID，必须保证唯一性，不设置则由终端性能监控 Pro 生成唯一 ID，影响设备异常率的统计以及联网设备数的统计,建议 sp 保存复用;
    builder.userId = "user_id"; // 【推荐设置】设置用户 ID，影响用户异常率的统计,建议 sp 保存复用，同一进程生命周期里面，暂不支持多次设置;
    builder.deviceModel = Build.MODEL; // 【推荐设置】设置设备类型，设置机型后，终端性能监控 Pro SDK 不再读取系统的机型
    builder.appVersion = "1.0.0"; // 【推荐设置】设置 App 版本号，不设置则从 packageManager 中读取。建议按应用的规范，主动设置，需要跟上传符号表的应用版本参数保持一致。
    builder.buildNumber = "builderNum"; // 【推荐设置】设置 App 版本的构建号，用于 Java 堆栈翻译关联版本，跟上传符号表的构建号参数保持一致。
    builder.appVersionType = BuglyAppVersionMode.Debug; // 【推荐设置】设置版本类型

    // 4. 更多初始化参数，按需设置初始化参数
    builder.appChannel = "appChannel"; // 设置 App 的渠道
    builder.logLevel = BuglyLogLevel.LEVEL_DEBUG; // 设置日志打印级别，级别可从 Bugly LogLevel 中获取
```

```
    builder.enableAllThreadStackCrash = true; // 设置 Crash 时是否抓取全部线程堆栈，默认开启
    builder.enableAllThreadStackAnr = true; // 设置 Anr 时是否抓取全部线程堆栈，默认开启
    builder.enableCrashProtect = true; // 设置性能监控时开启 Crash 保护模式，默认开启
    builder.debugMode = false; // 设置 debug 模式，可在调试阶段开启
    builder.initAppState = BuglyBuilder.APP_STATE_FOREGROUND; // 自 4.4.3.7版本起支持。该参数为非必选项，可在初始化 Bugly SDK 时指定应用的前后台状态。若未指定，SDK 将在初始化时通过 getRunningAppProcesses 判断应用的前后台状态；若已指定，SDK 将直接采用指定状态，不再调用 getRunningAppProcesses 进行判断。

    // 5. 设置回调方法，按需设置初始化参数
    builder.setCrashHandleListener(crashHandleListener); // 设置 Crash 处理回调接口，详情见回调接口
    builder.setUploadHandleListener(uploadhandleListener); // 设置 Crash 上报回调接口，详情见回调接口

    // 6. 初始化，必需调用
    Bugly.init(context, builder);
}
```

⚠ 注意事项：

- Context 需要传递 ApplicationContext。
- 设备 ID 非常重要，终端性能监控 Pro 使用设备 ID 来计算设备异常率，强烈建议应用设置正确的设备 ID，以确保设备的唯一性。
- BuglyBuilder 需在 init 方法前创建，且应避免重复调用 init 方法。
- 需要调用 Bugly.init 接口进行初始化，完成初始化后，再调用其他接口进行定制化设置，否则设置不生效。
- 性能监控项可在 应用配置 > SDK 配置 中进行采样调整，通过调整设备采样率来开启或者关闭性能监控项。
- 建议在用户授权《终端性能监控 Pro SDK 个人信息保护规则》 后再初始化终端性能监控 Pro SDK。
- 初始化时，一定要通过 BuglyBuilder.setServerHostType 来设置上报域名。

数据上报验证

最近更新时间：2025-08-14 15:29:52

前提条件

- 崩溃、ANR、OOM 是100%上报，默认开启，但不支持采样。除崩溃、ANR、OOM 外，其他监控项默认关闭，且支持采样。一般情况下，性能指标监控默认开启，性能异常监控默认关闭，用户可以通过 SDK 配置调整开启情况。在接入时，请务必确认功能的开启情况符合您的预期。
- 需要在 终端性能监控 Pro > 应用配置 > SDK 配置 页面创建配置任务。有关配置功能的详细说明，请参见 SDK 配置 。
- 在接入期间，建议将所有监控项的采样率调整为1.0（即 sample_ratio 和 event_sample_ratio），方便验证数据上报。接入完成后，可再根据需要调整采样率。或者创建多个配置任务，根据版本类型（对应初始化的 BuglyBuilder#appVersionType），配置开发任务、灰度任务，以及正式任务。开发任务打开所有的监控项，灰度阶段按需采样，正式发布根据需要降低采样。
- 检查待接入的产品是否已经 购买并绑定 生效中的资源包。如果产品没有绑定生效中的资源包，该产品将无法上报数据。

崩溃监控

崩溃监控是通过捕获应用运行时的异常和错误（如 Java 异常或 native 崩溃），收集崩溃日志和堆栈信息，帮助开发者定位和修复问题，提高应用稳定性。

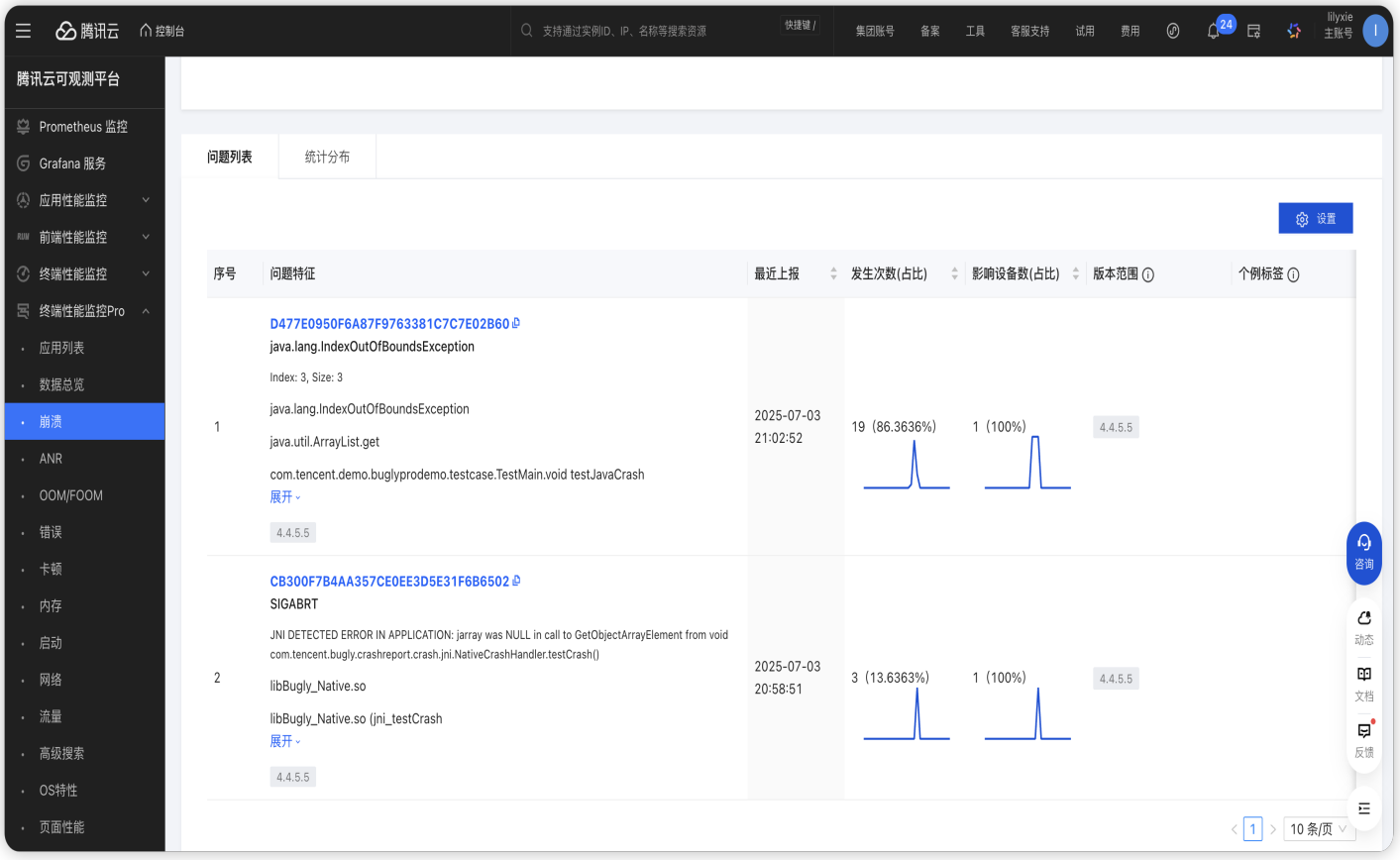
模拟异常

初始化完成后，可以模拟 Java 异常，或 Native 异常来验证 SDK 的上报情况。
崩溃发生后，部分异常问题需要在第二次启动才能完成上报，为保证当次崩溃处理完成，二次启动的时间与崩溃发生时间最好间隔10s以上。上报可能存在延时，测试时建议手动多触发数次后，再刷新展示界面。

```
Bugly.testCrash(Bugly.JAVA_CRASH); // 模拟 Java 异常

Bugly.testCrash(Bugly.NATIVE_CRASH); // 模拟 Native 异常
```

检查数据上报

崩溃上报后，可在 崩溃 > 问题列表 查看上报问题。



ANR 监控

Android 的 ANR 监控是通过检测应用主线程长时间无响应（如5秒内未处理输入事件），记录和上报相关堆栈信息，帮助开发者定位和优化卡顿问题。

模拟异常

跟崩溃类似，可以通过终端性能监控 Pro 提供的测试接口来模拟 ANR，也可以自行在 UI 线程执行异常耗时任务。

- 接口模拟 ANR。

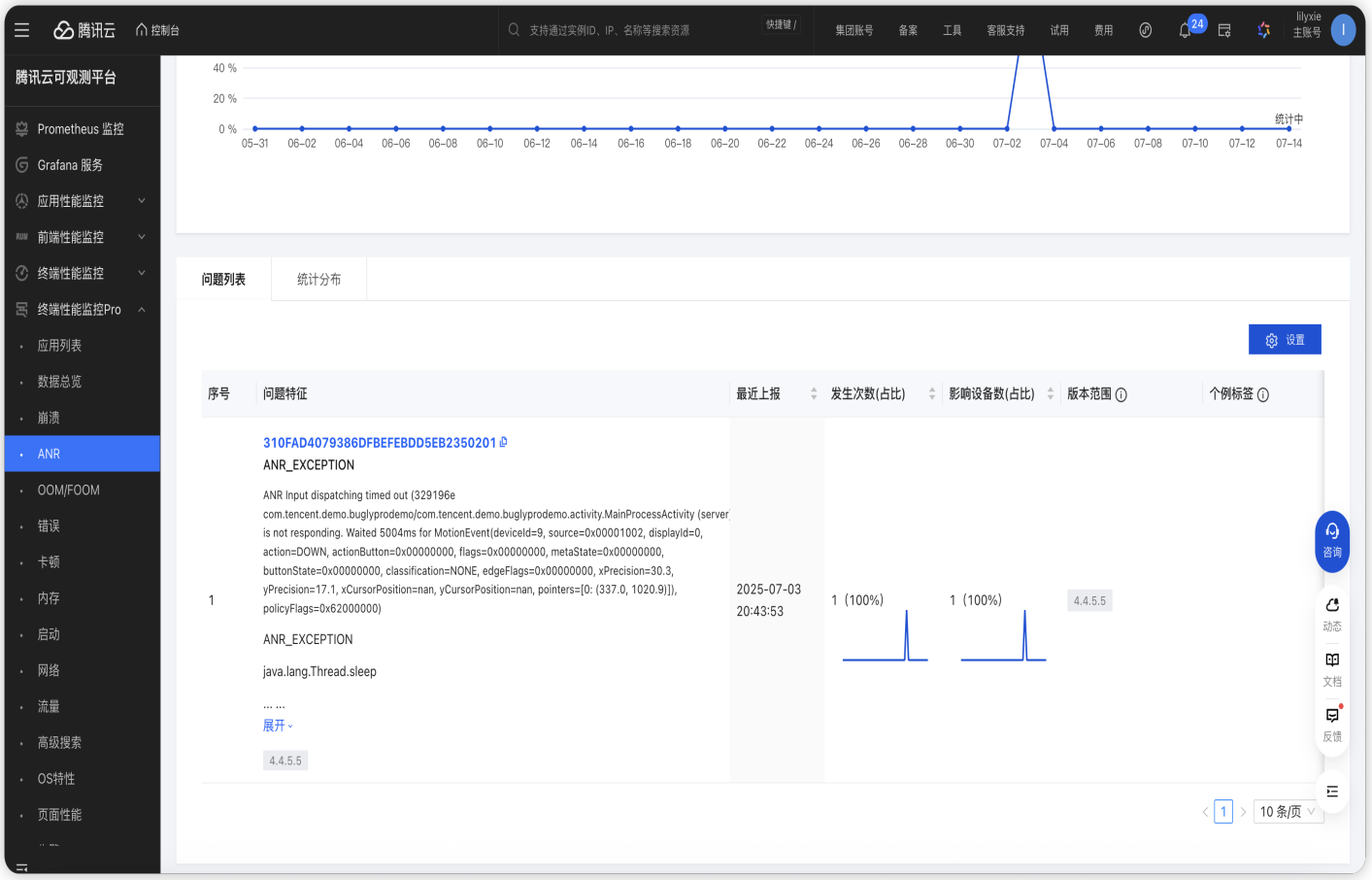
```
Bugly.testCrash(Bugly.ANR_CRASH); // 模拟 ANR
```

- 在 UI 线程执行以下耗时逻辑，触发 ANR。

```
private void testANR(){
    try {
        Thread.sleep(30000);
    } catch (Exception e){
        // do nothing
    }
}
```

检查数据上报

ANR 上报后，可在 ANR > 问题列表 查看上报问题。

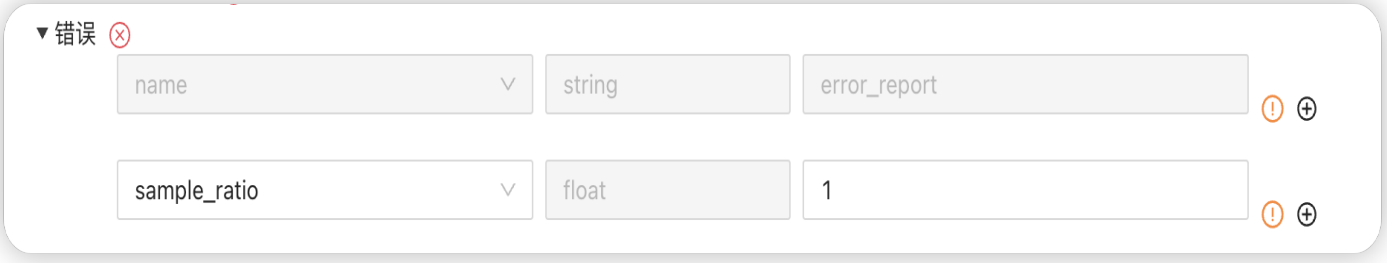


错误监控

错误一般是指用户自定义的异常，如已经捕获的 Java 异常，或者 C#异常、JS 异常、lua 异常等。一般通过终端性能监控 Pro 的 handleCatchException 或者 postException 来上报（详细参考其他接口/上报自定义异常或者上报 Java catch 异常部分）。

检查 SDK 配置

验证数据上报前，请在 应用配置 > SDK 配置 中检查错误的采样情况。



SDK 接入

应用可以通过终端性能监控 Pro 提供的 handleCatchException 和 postException 接口上报自定义错。

- 上报 Java 捕获异常，可以参考以下示例代码。

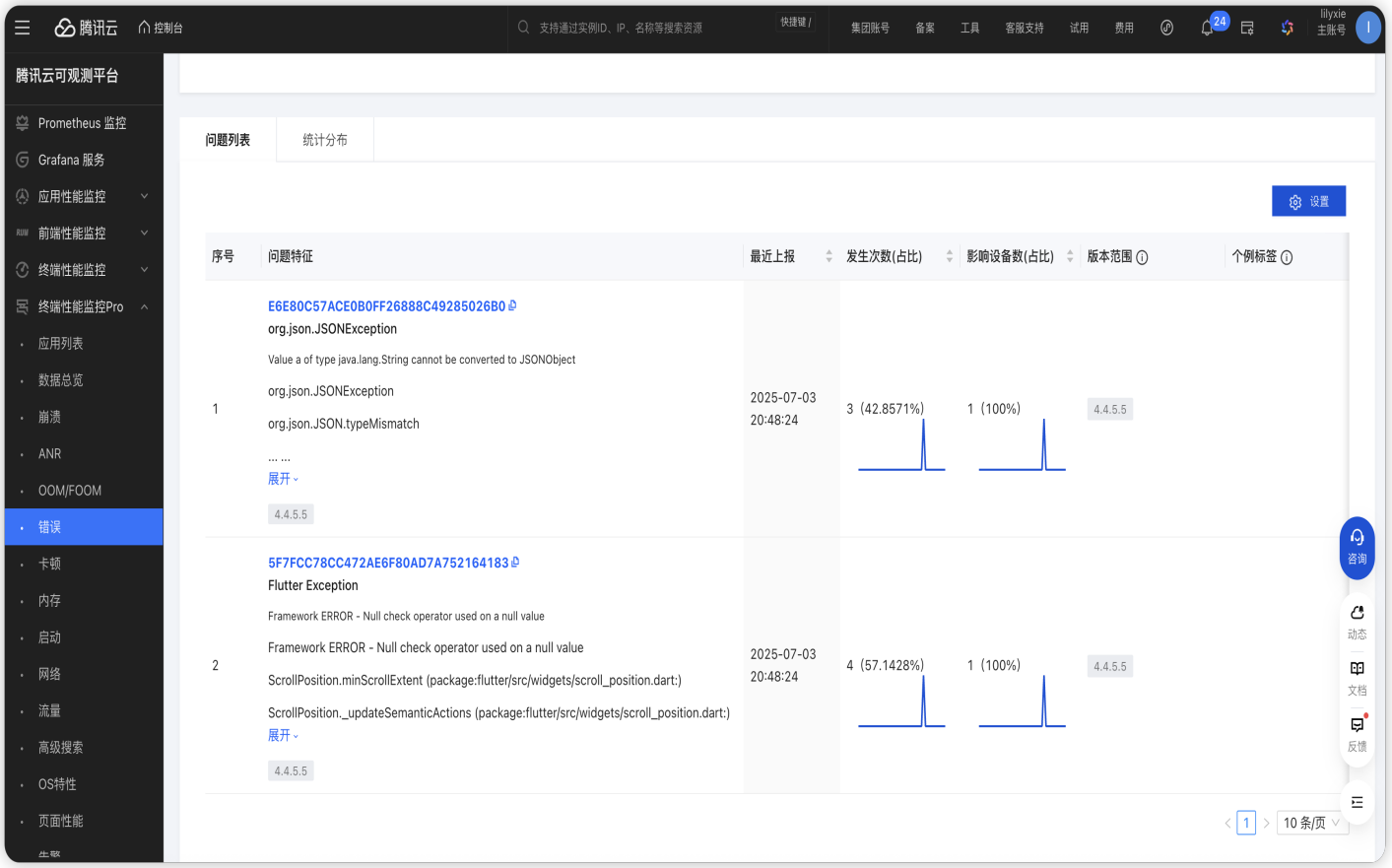
```
private void testJavaCatchError(){
      String content = "a illegal string.";
      try {
          JSONObject jsonObject = new JSONObject(content);
          double price  = jsonObject.getDouble("price");
      } catch (Throwable t) {
          Bugly.handleCatchException(Thread.currentThread(), t,
                                  "in test code for json parse fail.",
content.getBytes(), true);
      }
}
```

- 上报自定义异常，参考以下示例代码。

```
Bugly.postException(4, "testErrorType", "testErrorMsg", "testStack",
null);
```

检查数据上报

错误上报后，可在 错误 > 问题列表 查看上报问题。

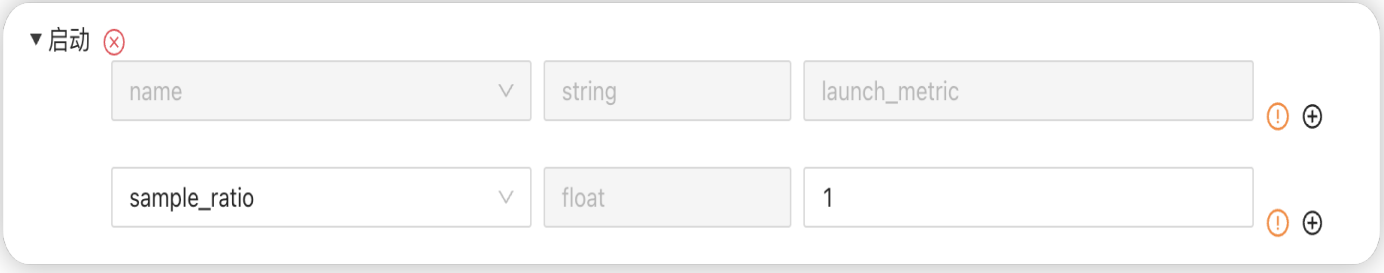


启动监控

在开启启动监控的情况下，启动模块会自动安装监控，并且在 SDK 初始化后，进行启动数据上报。用户可以通过 Bugly.reportAppFullLaunch 接口自定义启动结束点，也支持自定义标签以及通过打点接口，监控启动过程中的耗时任务。

检查 SDK 配置

验证数据上报前，请在 应用配置 > SDK 配置 中检查启动的采样情况。



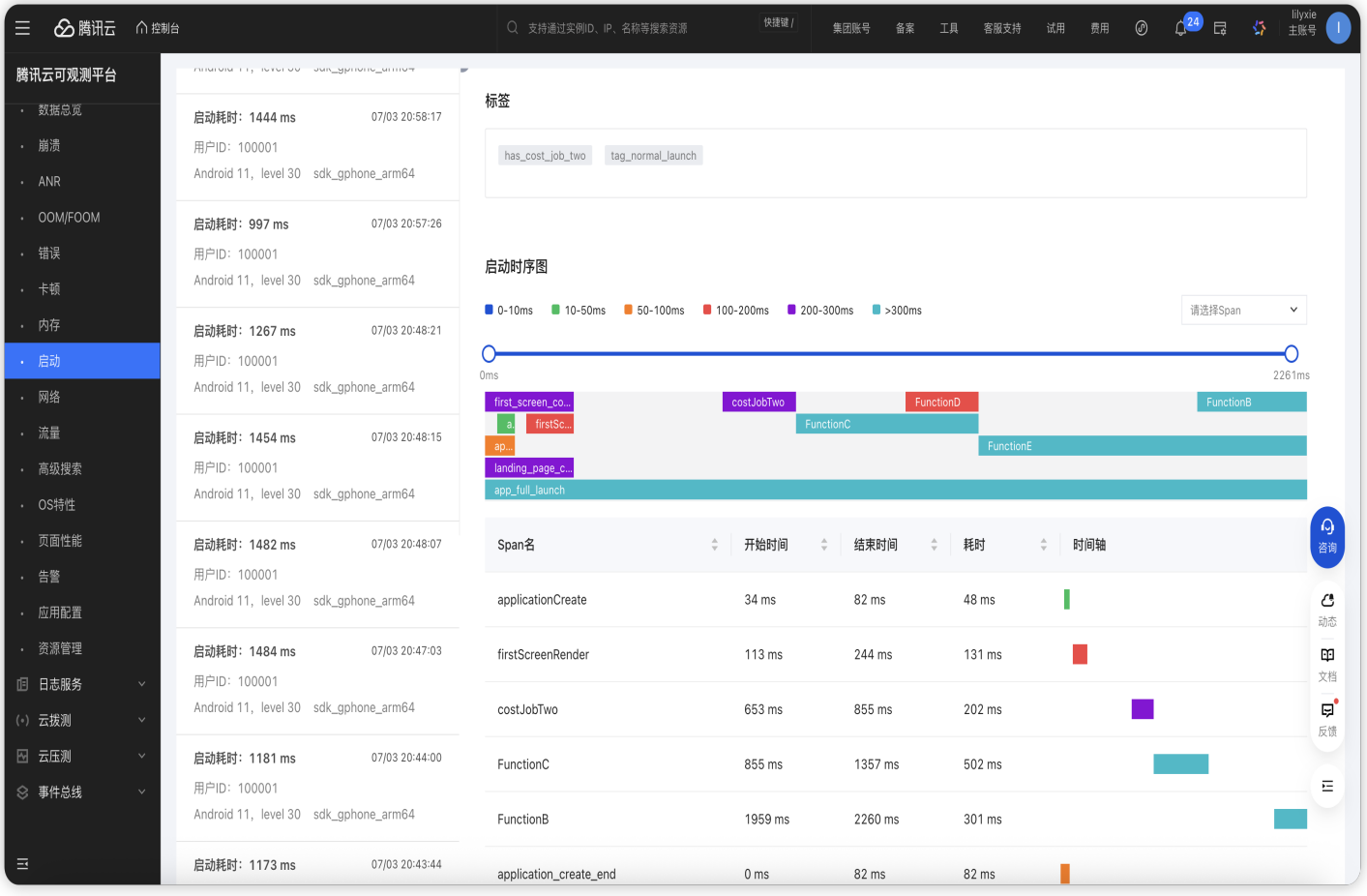
SDK 接入

请查看 API 说明–启动监控 部分了解更多启动接口的使用。

```
private void costJobOne() {
    AppLaunchProxy.getAppLaunch().addTag("has_cost_job_one");
    AppLaunchProxy.getAppLaunch().spanStart("costJobOne", null);
    try {
        Thread.sleep(300);
    } catch (Throwable t) {
        Bugly.handleCatchException(Thread.currentThread(), t,
"costJobOne sleep fail", null, true);
    }
    AppLaunchProxy.getAppLaunch().spanEnd("costJobOne");
}

private void finalJob() {
    AppLaunchProxy.getAppLaunch().reportAppFullLaunch();
}
```

检查数据上报

数据上报后，可在 启动 > 启动个例 查看上报详情。



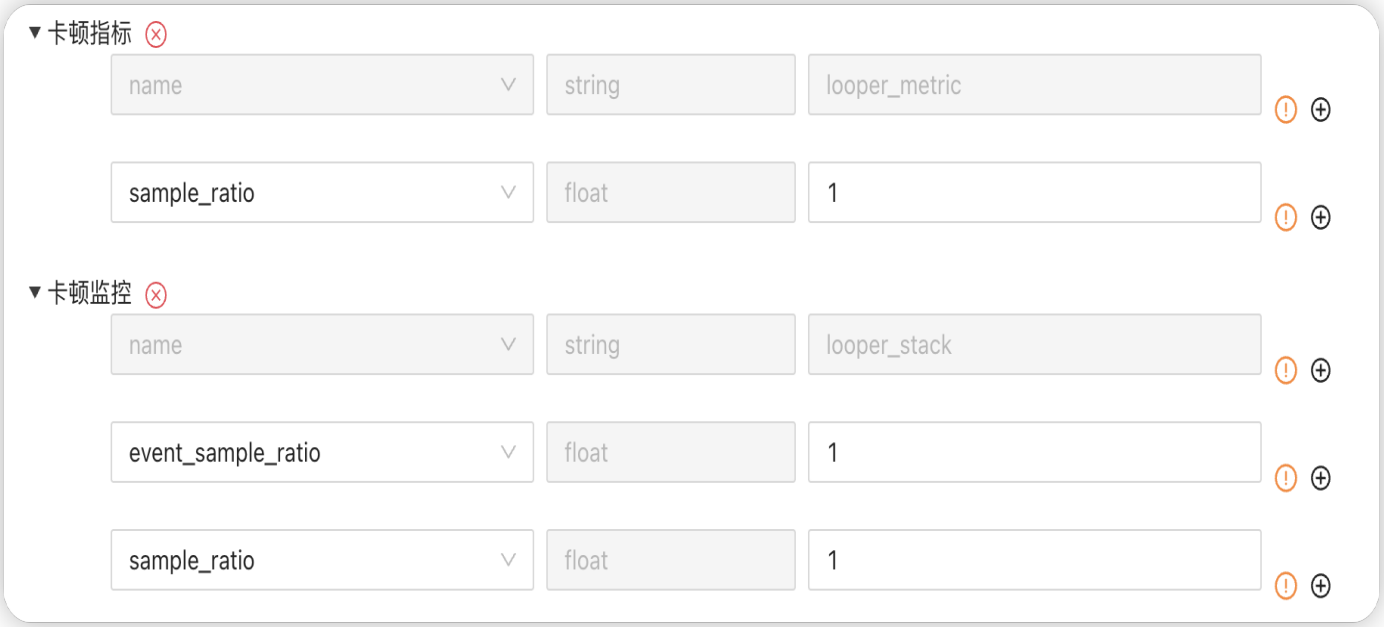
卡顿监控

应用卡顿是导致用户流失的重要原因之一，卡顿监控模块致力于协助用户治理卡顿问题。卡顿治理的一个重要思路是，通过提取有效的指标衡量应用当前的状况，抓取足够的信息提供优化方向。卡顿监控模块通过 FPS 及挂起率两个指标来衡量应用的流畅度情况，通过卡顿问题监控，直接抓取卡顿堆栈，协助用户定位卡顿原因，为用户提供优化方向。

- 卡顿指标通过稳定、轻量的采集技术，收集应用在整个运行过程中的流畅度数据。
- 卡顿问题监控，通过高频抓栈技术，提供丰富的现场信息，实现准确归因。
- Android 平台自研快速抓栈技术，对比传统抓栈实现，性能提升6倍。
- 无论是卡顿指标，还是卡顿问题，都支持用户自定义采样率，同时提供了丰富的数据分析能力。

检查 SDK 配置

在验证数据上报前，请先检查 应用配置 > SDK 配置 中卡顿的配置情况。



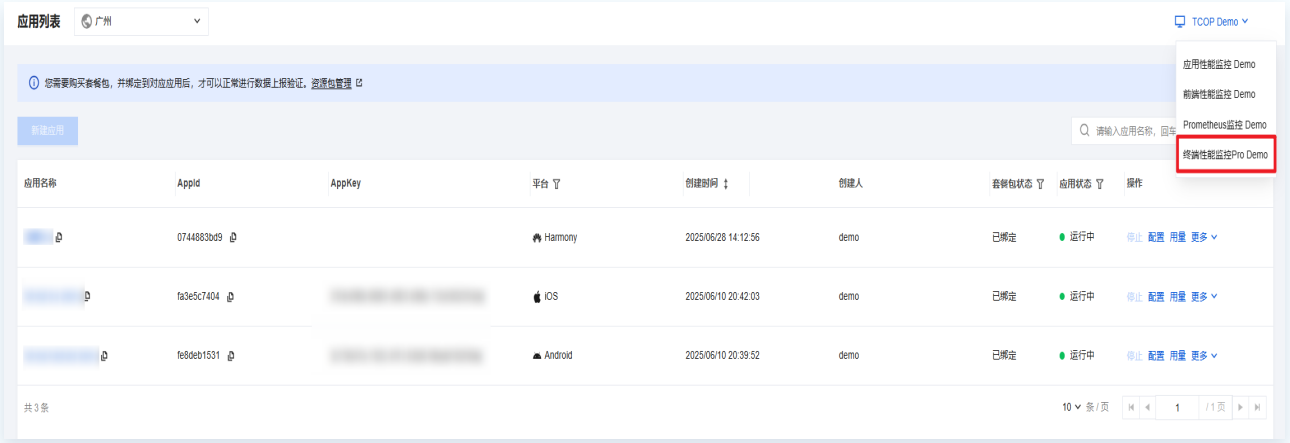
- **卡顿指标：FPS 及挂起率**

  在开启卡顿指标（"looper_metric"）的情况下，SDK 会在初始化完成后，监控应用的流畅度情况。在应用运行一次期间，会先收集数据，并保存至本地，下次启动时再聚合上报。为了避免影响应用的启动，SDK 会在初始化5分钟后，再将缓存的数据聚合上报后台。

  - FPS：帧率，在应用运行时，GPU 和 CPU 合作可产生的图像的数量，计量单位是帧/秒（FramePerSecond，FPS），通常是评估硬件性能与应用体验流畅度的指标。
  - 挂起率：如果应用两帧之间的刷新延时超过200ms，则认为此时应用不能很好地响应用户的交互，累加到挂起时间中。一个设备的挂起率，指这个设备在一天中，总的挂起时间除以其前台总时长，单位是秒/小时。

> **说明：**

可以通过在 Demo 中测试滑动卡顿来观察列表滑动情况下的 FPS 以及挂起率。您可在 终端性能监控 Pro > 应用列表 页面右上角选择**终端性能监控 Pro Demo**。



- **卡顿监控**

  开启卡顿监控（"looper_stack"）后，在 UI 线程执行耗时逻辑，耗时超过500ms的情况下，会触发卡顿上报。卡顿监控通过监控 UI 线程的消息执行来判断当前 UI 线程是否发生卡顿。
  在验证阶段，建议将卡顿监控的"event_sample_ratio"（消息采样率）以及"sample_ratio"（设备采样率）都设置为1。这样只要满足卡顿的耗时阈值，即可触发卡顿上报。

模拟异常

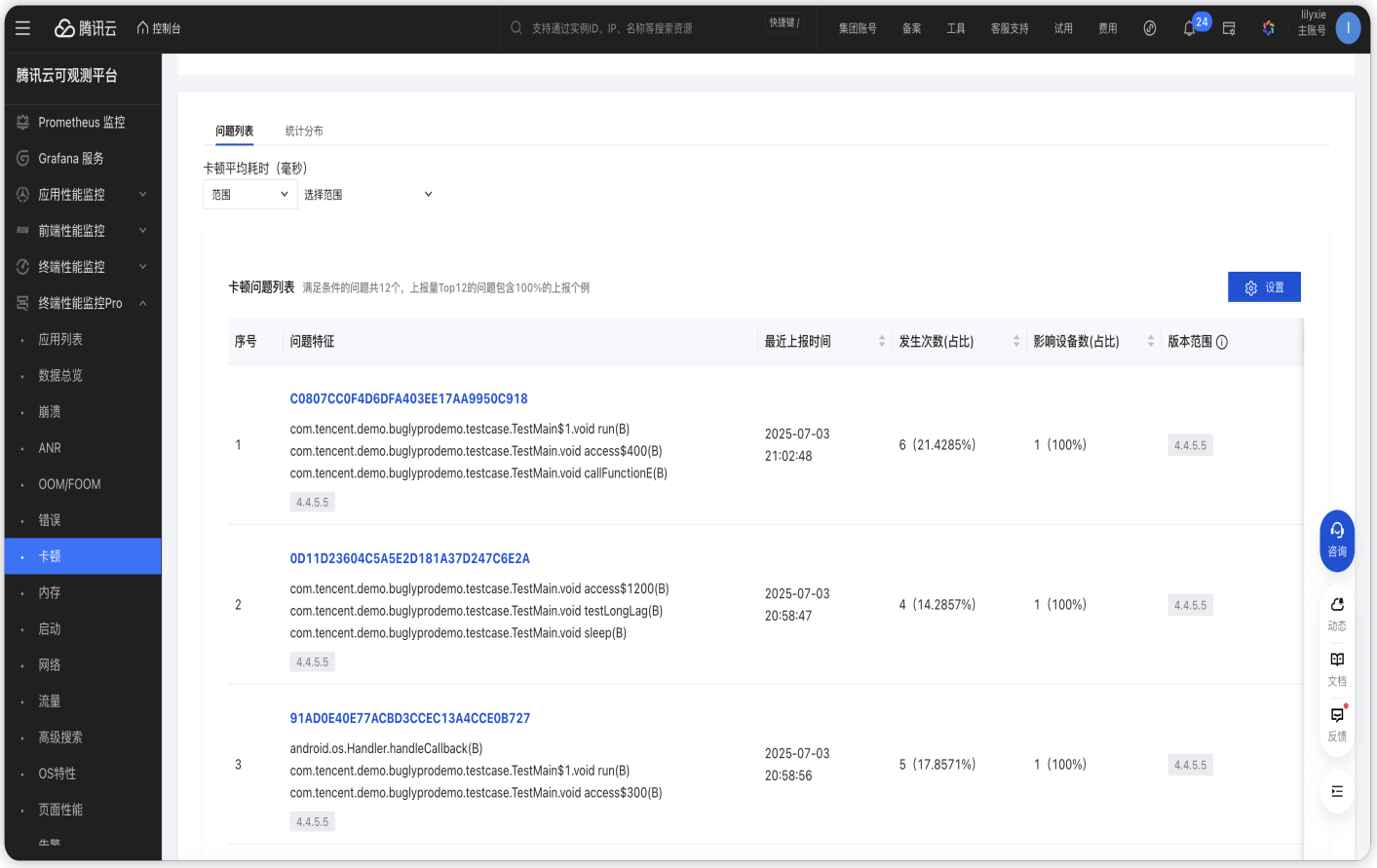
参考以下示例代码模拟一个卡顿上报。

```
private void testLongLag() {
    showToast("测试长卡顿");
    sleep(200);
    callFunctionA();
    callFunctionB();
}

private void callFunctionA() {
    sleep(400);
    callFunctionB();
}

private void callFunctionB() {
    sleep(300);
}

private static void sleep(long timeInMs) {
    try {
        Thread.sleep(timeInMs);
```

```
        } catch (Throwable t) {
            Bugly.handleCatchException(Thread.currentThread(), t,
"sleep", null, false);
        }
    }
```

检查数据上报

卡顿上报后，可在 **卡顿 > 问题列表** 查看上报问题。



内存监控

内存监控当前支持内存峰值和内存详情功能，更多内存监控功能在建设中。

内存峰值

内存峰值是指，应用在整个运行期间的内存使用峰值。终端性能监控 Pro 通过定时任务，查询应用的内存使用情况，并且保存整个运行期间的最大使用量，应用下次启动后再上报。

- 物理内存峰值：一次运行期间，应用使用的物理内存峰值，即 PSS 峰值。
- 虚拟内存峰值：一次运行期间，应用使用的虚拟内存峰值，即 VSS 峰值。
- Java 堆内存峰值：一次运行期间，应用所使用的 Java 堆内存峰值。

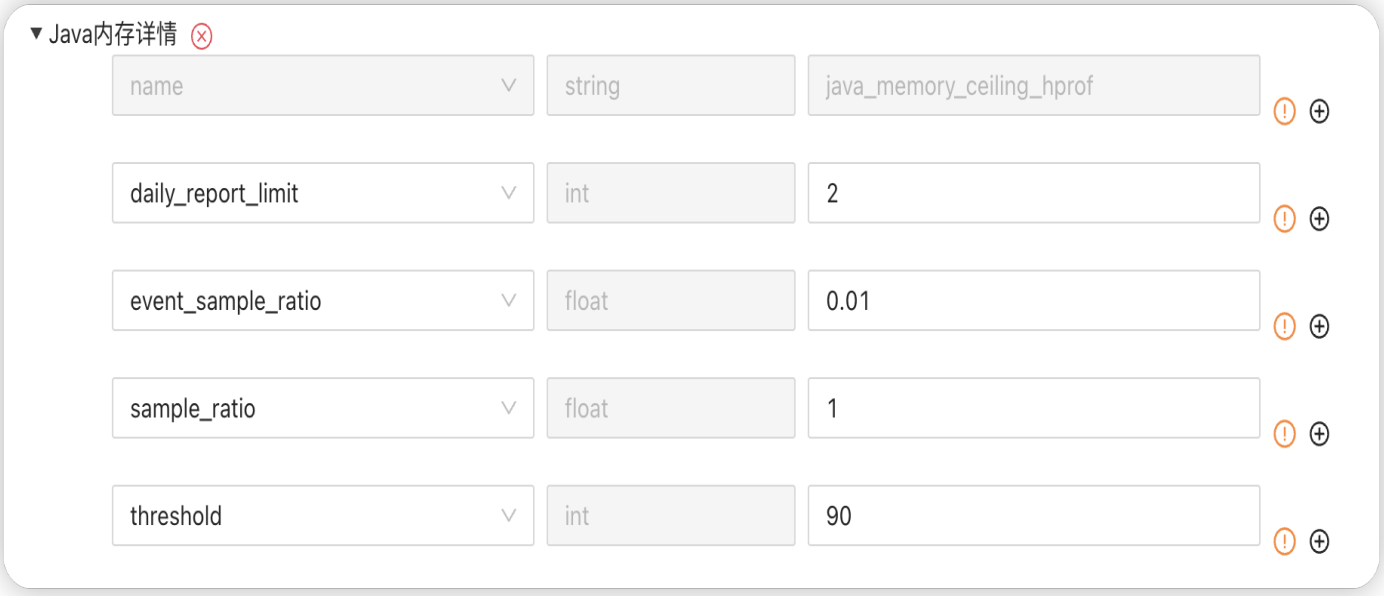
内存详情

在初始化 SDK 时，需要在 BuglyBuilder 中主动添加监控项 BuglyMonitorName.MEMORY_JAVA_CEILING。

```
public static void initBugly(Context context) {
    // 1. 初始化参数预构建，必需设置初始化参数
    String appID = "xxxxxxx"; // 【必需设置】在终端性能监控Pro 创建产品的appID
    String appKey = "xxxxxxxxxxx"; // 【必需设置】在终端性能监控Pro 创建产品的
appKey
    BuglyBuilder builder = new BuglyBuilder(appID, appKey);

    ......

    // 2. 开启Java内存详情
    build.addMonitor(BuglyMonitorName.MEMORY_JAVA_CEILING);

    // 3. 初始化，必需调用
    Bugly.init(context, builder);
}
```

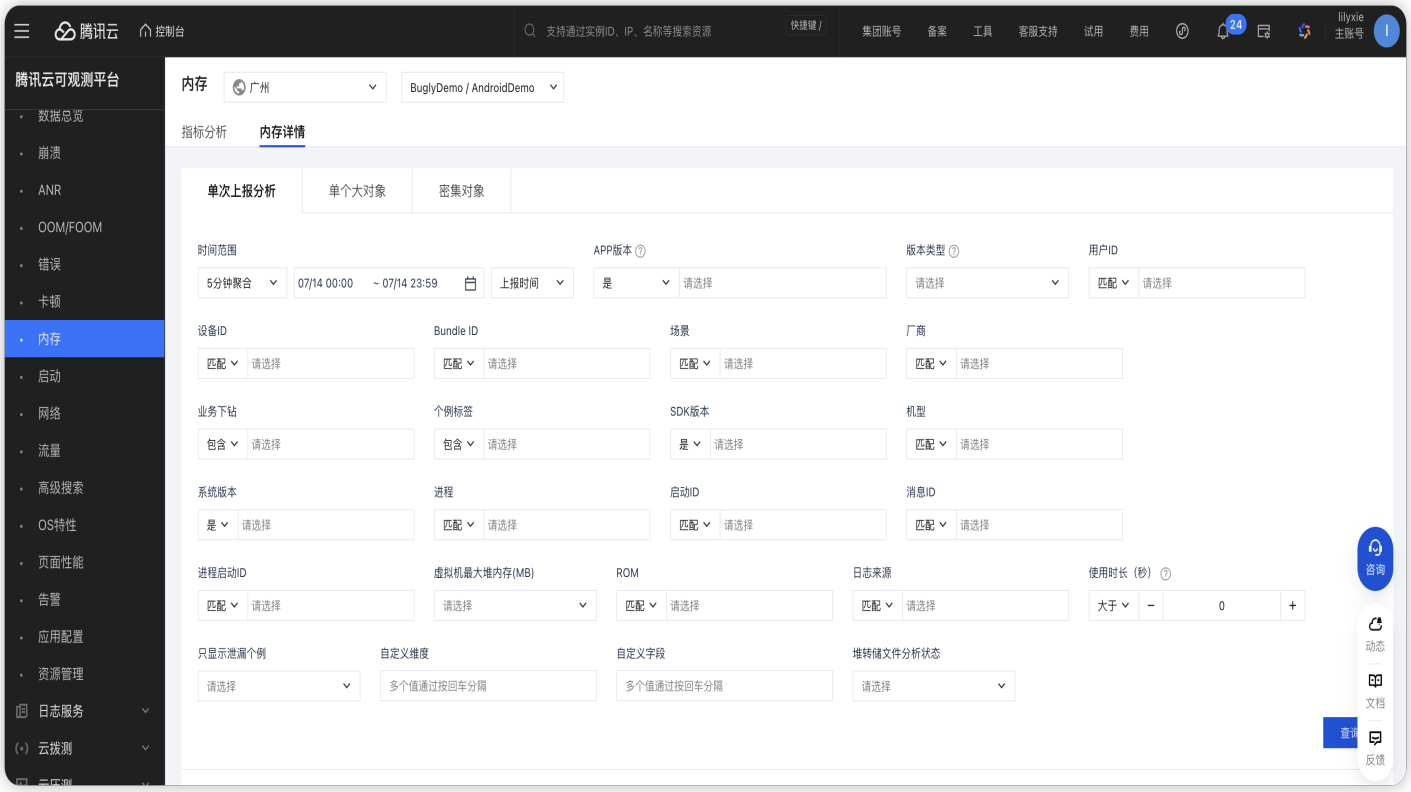
在 应用配置 > SDK 配置 中修改 SDK 配置任务， 调整 **Java 内存详情**的配置属性。

- sample_ratio: 控制用户采样率，即多少设备会开启这个功能;
- event_sample_ratio: 控制事件采样率，即发生内存触顶之后，是否需要上报;
- threshold：设置堆转储文件的dump时机，90 代表的是在达到最大值的90%的时候开始dump，堆内存最大值对应 Runtime.getRuntime().maxMemory();



检查数据上报

触发上报后，可在 内存/内存详情 中查看上报问题。



网络监控

网络监控是通过 okhttp3的 EventListener 来监控 HTTP 请求的质量，包含请求耗时、成功率、传输数据量，以及请求过程中重要阶段的耗时。当前通过无插桩的方案实现以下能力：

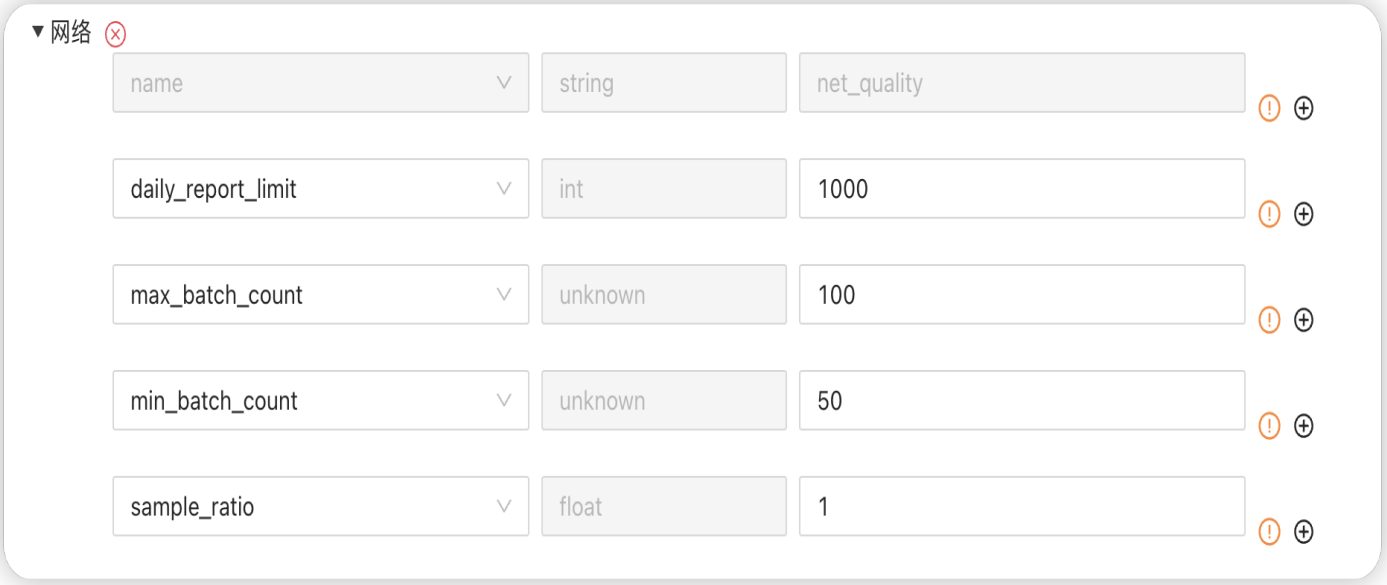
1. 业务按 SDK 接入指引，使用 BuglyListenerFactory 创建的 OkHttpClient，支持监控 http/https 请求的质量。
2. 提供 BuglyURLStreamHandlerFactory，代理系统原生的 HttpURLConnection，从而支持监控使用系统原生接口实现的 http/https 请求。

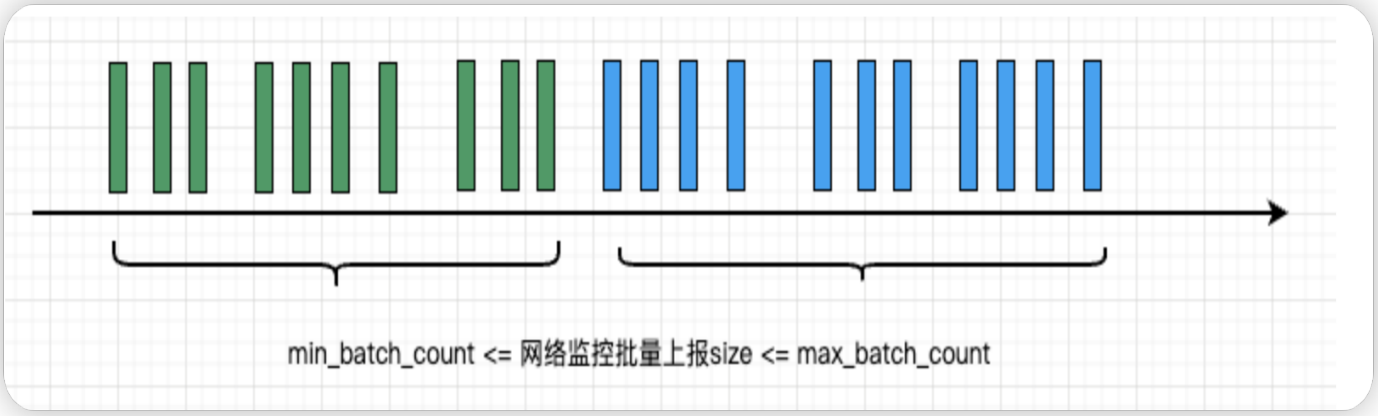
> **说明：**
> 当前暂不支持如下能力：
> - 不使用原生接口或者 okhttp3实现的 HTTP 请求。
> - 组件使用了 okhttp3库实现 HTTP 请求，但是没有调整 OkHttpClient 过程，即没有使用 BuglyListenerFactory 作为 OkHttpClient 的 EventListener.Factory。

检查 SDK 配置

验证数据上报前，请先检查 应用配置 > SDK 配置 中网络监控的配置情况。
网络监控包含以下配置属性：

- sample_ratio：设备采样率，表示允许多少设备开启网络监控。
- daily_report_limit：表示网络监控的数据上报，每天最多允许上报多少次。网络监控的数据是合并上报的，具体合并的情况由 max_batch_count 和 min_batch_cout 来决定。
- max_batch_count：表示一条网络监控的数据上报中，最多包含多少条 HTTP 请求质量的明细数据，默认是100。
- min_batch_count：表示一条网络监控的数据上报中，最少包含多少条 HTTP 请求质量的明细数据，默认是50 。





注意：

- 在自测阶段，推荐修改 sample_ratio 为1，自测结束后，根据实际情况调整设备采样率。
- 在自测阶段，推荐修改 min_batch_count 为5或者10，这样可以将数据尽快上报到后台。自测结束后，推荐根据业务的实际情况来调整，或者还原为默认值。

SDK 接入

1. 代理系统原生的 HttpURLConnection。如果不想代理**使用系统原生接口**的 HTTP 请求，则无需执行此步骤。

   如果想代理「使用系统原生接口」的Http请求，推荐在 Application#onCreate 时尽早开启代理。

```
// 开启代理
BuglyURLStreamHandlerFactory.init(okHttpClient);
```

   开启代理后，也可以通过以下代码关闭。

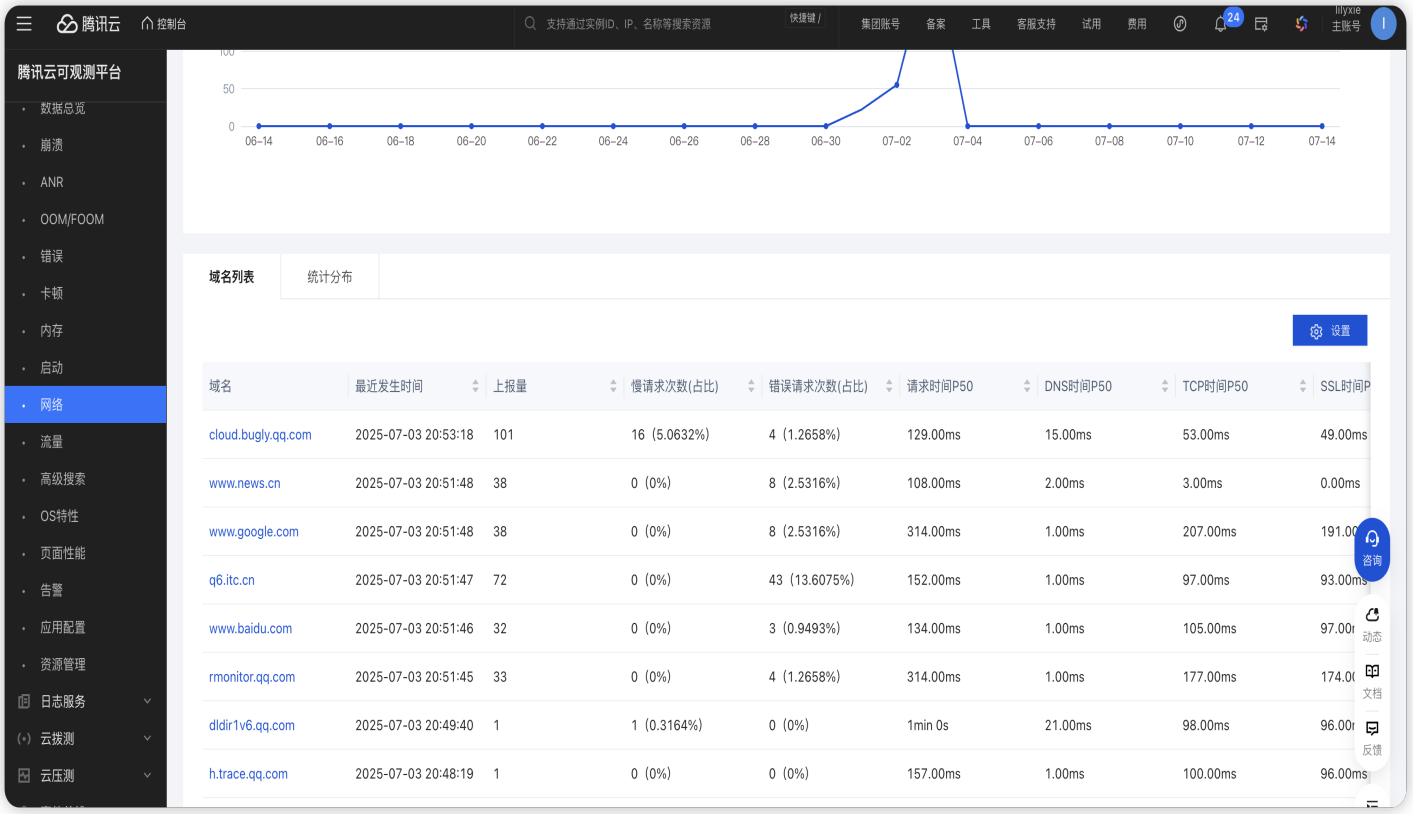
```
// 关闭代理
BuglyURLStreamHandlerFactory.reset();
```

2. 创建 OkHttpClient 时，使用 BuglyListenerFactory。

```
OkHttpClient client = new OkHttpClient.Builder()

.eventListenerFactory(BuglyListenerFactory.getInstance())
                .build();
```

检查数据上报

完成接入后，可在 网络 > HTTP 或者 网络 > 网络错误 查看上报数据。

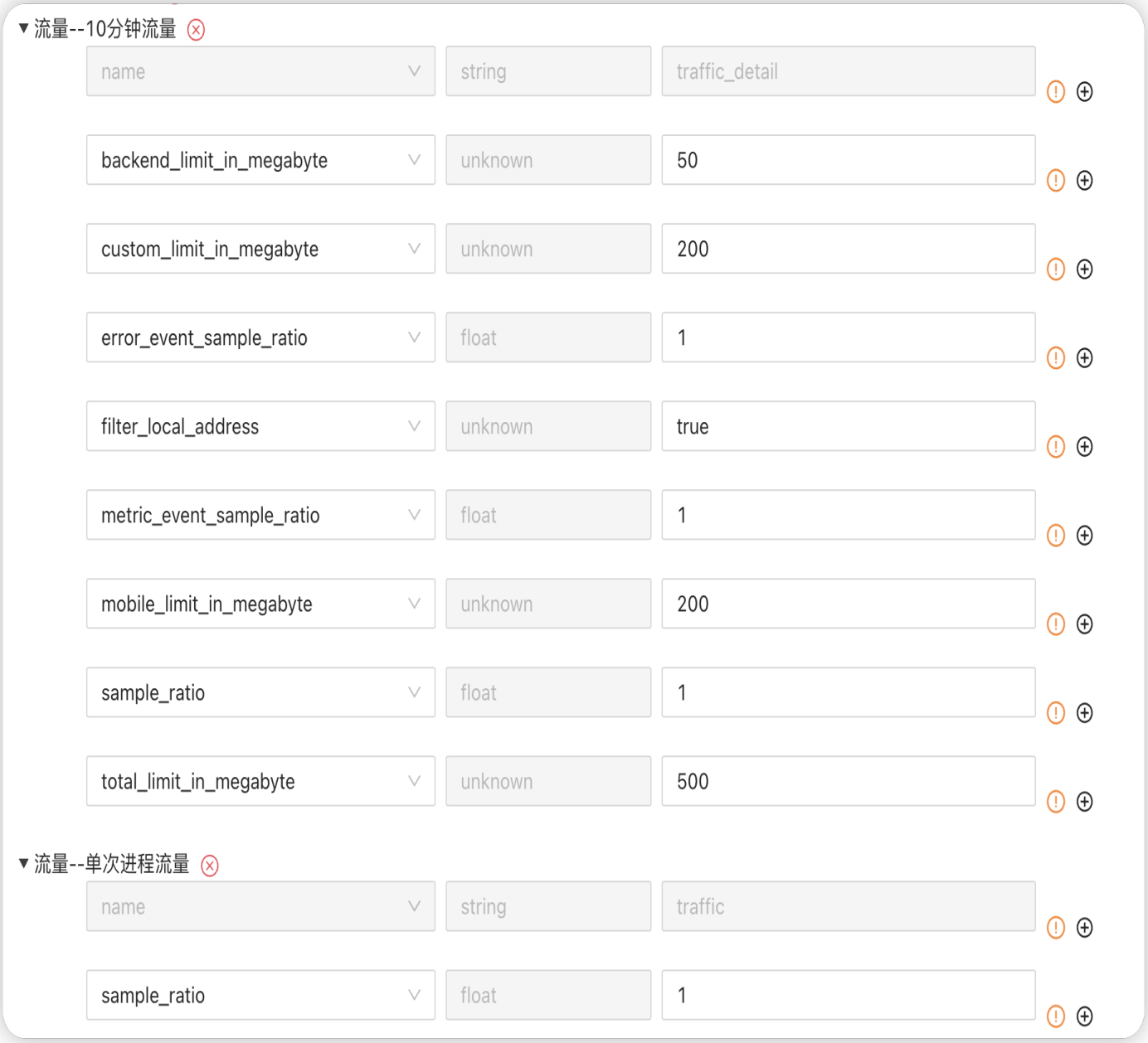


流量监控

流量监控的核心功能在于它能够实时监控应用程序的数据使用情况。通过实时监控，可以随时了解应用的数据消耗情况，及时发现流量使用异常等问题。这有助于用户避免超出数据计划或产生额外费用。此外，流量监控还能帮助用户优化应用程序的功耗，减少不必要的数据请求，提供更高效的用户体验，并延长设备的电池寿命。流量监控还具备下钻到域名级别的能力，用户可以了解哪些域名或服务消耗了大量的流量，从而采取相应的措施来优化数据传输。

检查 SDK 配置

验证数据上报前，请先检查 应用配置 > SDK 配置 中流量监控的配置情况。流量监控包含**流量--10分钟流量**和**流量--单次进程流量**两个配置项。



SDK 接入

在客户端，SDK 需要在初始化的时候执行以下语句，同时在后台配置开启流量监控功能，才会真正开启。

```
...
buglyBuilder.addMonitor(BuglyMonitorName.TRAFFIC);
buglyBuilder.addMonitor(BuglyMonitorName.TRAFFIC_DETAIL);
...
// 初始化Bugly SDK
Bugly.init(application, buglyBuilder);
```

SDK 还提供了自定义场景流量上报接口。

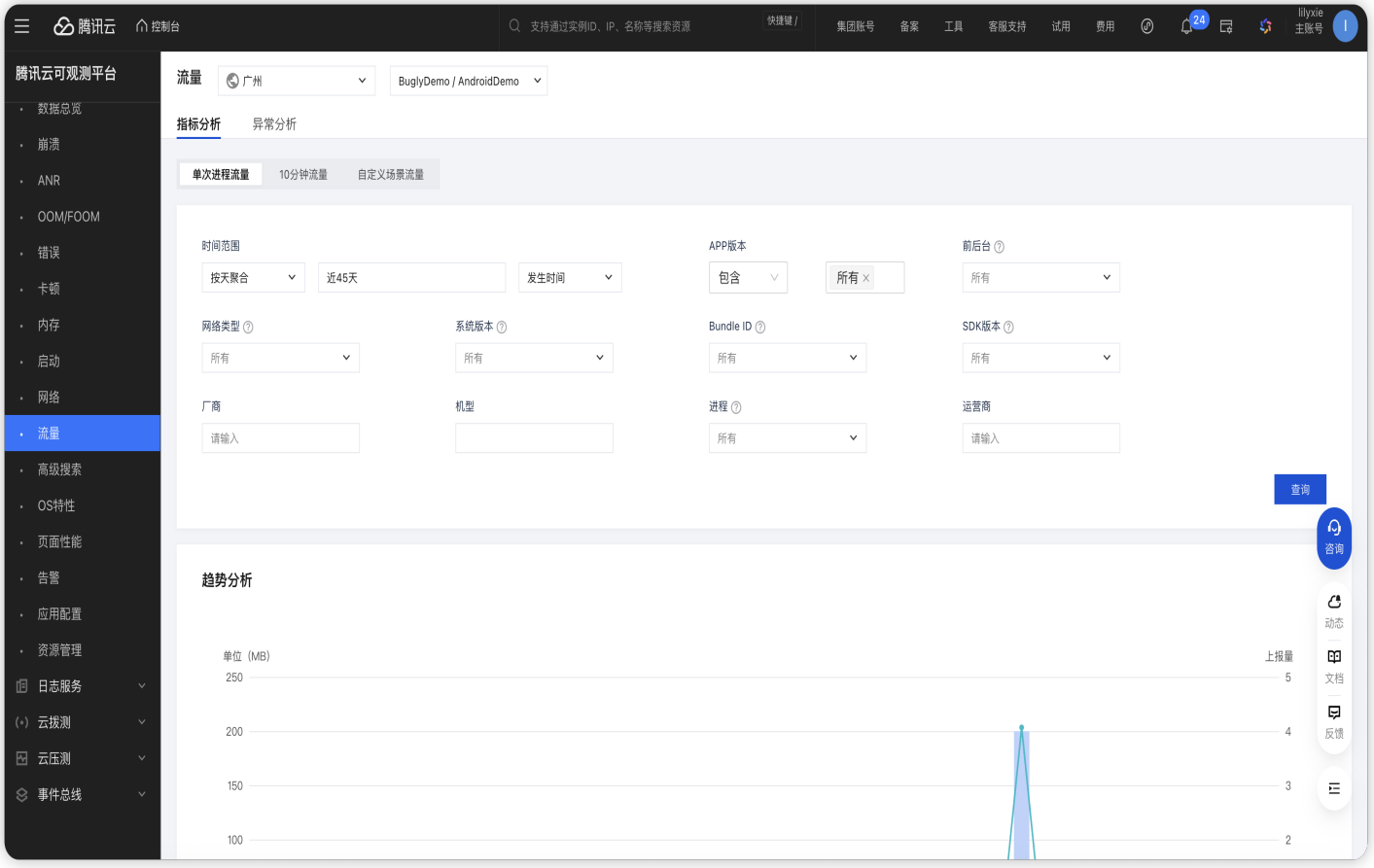
```
CustomTrafficStatistic.getInstance().addHttpToQueue(SocketInfo
socketInfo);
```

socketInfo 还有很多其他字段，统计流量只需要给以下字段来赋值，其他成员变量可以暂时忽略。

```
class SocketInfo {
    receivedBytes;    // 收到多少字节
    sendBytes;        // 发送了多少字节
    networkType;      // 网络类型，WiFi流量填1， 5G流量填2， 无网络情况填3
    frontState;       // 前后台状态，前台填1，后台填2
    host;             // 域名，例如www.baidu.com
    startTimeStamp;   // 网络请求开始时间戳,精确到ms
    ......
}
```

检查数据上报

数据上报后，可在 流量 > 指标分析 或者 流量 > 异常分析 中查看数据。



页面性能

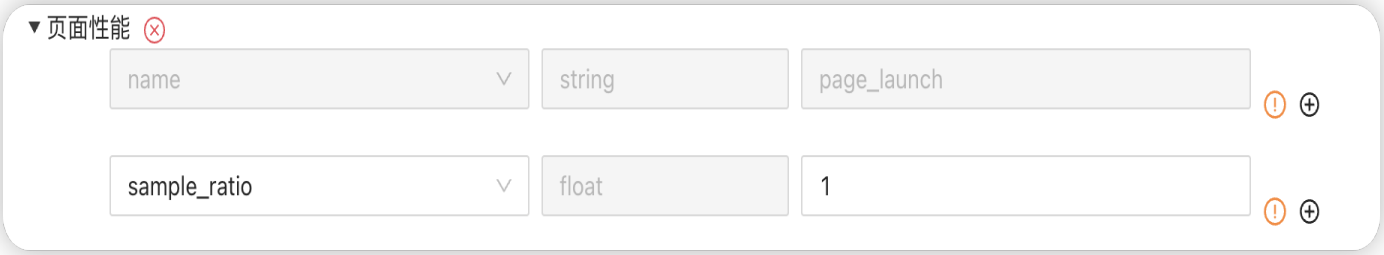
在移动应用开发中，用户体验（User Experience，UX）是一个至关重要的因素。随着智能手机硬件性能的提升和用户需求的不断增加，用户对应用的响应速度和流畅度有了更高的期望。特别是在移动端应用中，页面启动时间是用户体验的一个关键指标。页面启动时间过长会导致用户感到不耐烦，甚至可能导致用户流失。

终端性能监控 Pro 的页面启动耗时默认会监控 Activity 的冷启动耗时，耗时信息也分为两类：页面渲染耗时和页面加载耗时。

- 页面渲染耗时：从 Activity 的 onCreate 调用到第一帧画面渲染完成的时间。
- 页面加载耗时：其值默认与页面渲染耗时相同，但用户可以通过 reportActivityFullLaunch 接口来自定义 Activity 的加载结束时间。

检查 SDK 配置

验证数据上报前，请先检查 应用配置 > SDK 配置 中**页面性能**的配置情况。
sample_ratio：控制用户采样率，即多少设备会开启这个功能，1代表所有设备开启，0代表所有设备都不开。



SDK 接入

客户端在调用 Bugly.init 接口之前，需添加如下代码。

```
...
buglyBuilder.addMonitor(BuglyMonitorName.PAGE_LAUNCH); // 添加此语句
...
Bugly.init(context, builder);
```

除了监控页面的整体耗时外，用户还可以通过调用以下接口，监控页面启动过程中各个 span 的耗时。span 可以理解为页面启动过程中各个子阶段，以下为 com.tencent.rmonitor.pagelaunch.PageLaunchMonitor 类的接口，该类是单例实现，可以通过 PageLaunchMonitor.getInstance() 获取该单例对象。

```
/**
  * 标记 activity 启动过程中某个span的开始
  * @param activity
  * @param name,  span名字
  * @param parentName,  父span的名字
  */
```

```
public void startSpan(Activity activity, String name, String
parentName);


/**
 * 标记 activity 启动过程中某个span的结束
 * @param activity
 * @param name span名字，需要匹配startSpan的参数
 */
public void endSpan(Activity activity, String name);


/**
 * 用户自定义Activity启动结束时间点，会以调用该接口的时间作为Activity的启动结束时间
 * @param activity
 */
public void reportActivityFullLaunch(Activity activity);
```

> ⚠ **注意：**
> startSpan 和 endSpan 需要成对匹配调用，同一 span 名字的只会保存一条，即后面同名的 span 会覆盖前面的。

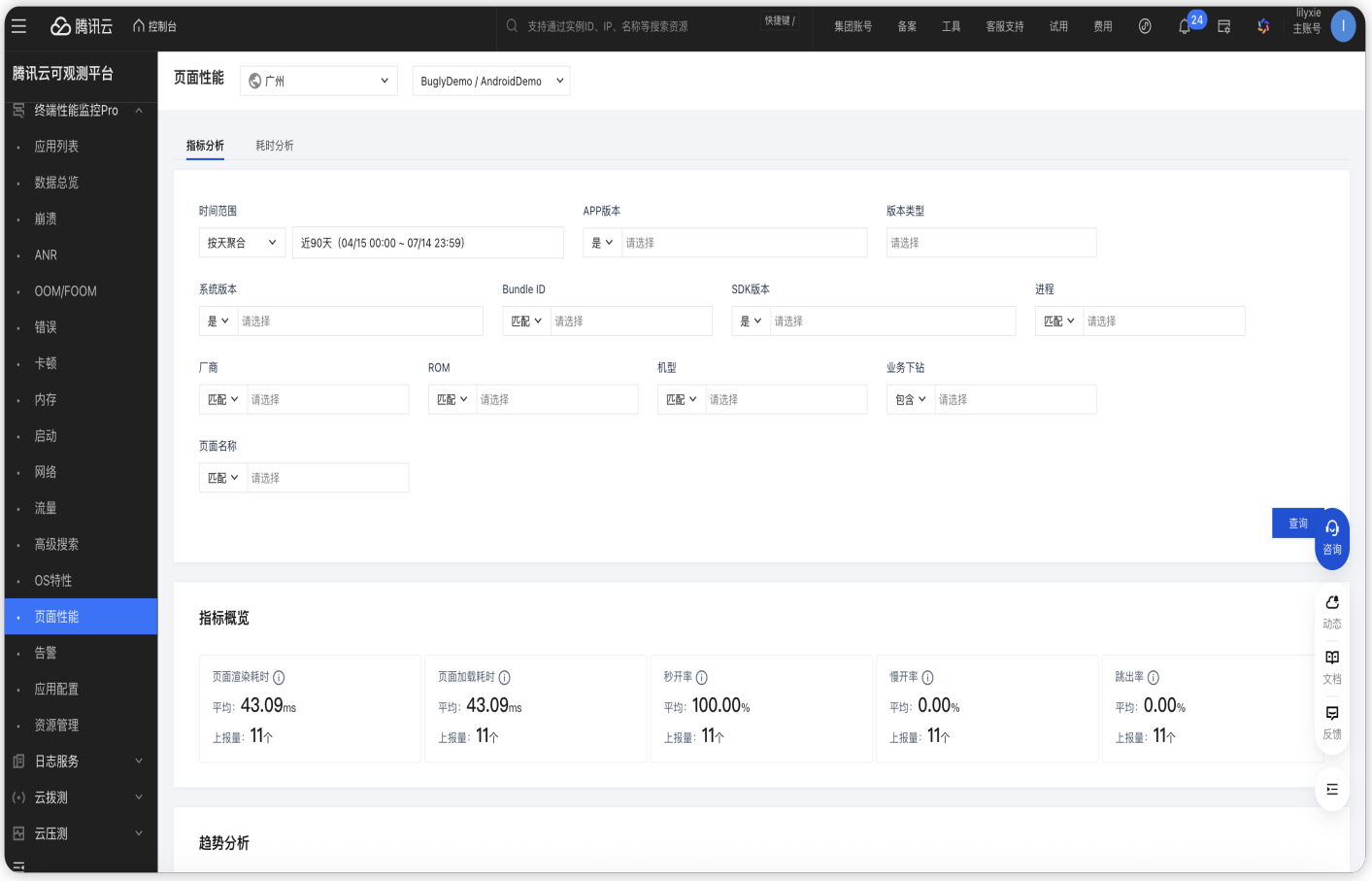
可参考以下示例代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PageLaunchMonitor.getInstance().startSpan(this, "MemoryTracerTest",
"");
    PageLaunchMonitor.getInstance().startSpan(this, "onCreate",
"MemoryTracerTest");
    startMemoryTrace();
    PageLaunchMonitor.getInstance().endSpan(this, "onCreate");
}

@Override
public void onResume() {
    super.onResume();

    PageLaunchMonitor.getInstance().startSpan(this, "onResume",
"MemoryTracerTest");
```

```
        reStartMemoryTrace();
        PageLaunchMonitor.getInstance().endSpan(this, "onResume");
        PageLaunchMonitor.getInstance().endSpan(this, "MemoryTracerTest");

        // 等页面加载完成之后，才调用 reportLaunchFinished 来定义Activity启动结束
        new Thread (new Runnable() {
            reportLaunchFinished();
        }).start();
    }

    private void reportLaunchFinished() {
        PageLaunchMonitor.getInstance().reportActivityFullLaunch(this);
    }
```

检查数据上报

数据上报后，可在 **页面性能 > 指标分析** 查看上报数据。



API 说明

最近更新时间：2025-08-14 15:29:52

本文详细介绍终端性能监控 Pro SDK 的各功能接口，帮助您更加灵活、深度的使用 SDK。

回调接口

设置 Crash 处理回调

终端性能监控 Pro 处理 Crash 异常时可以回调业务，业务可以在回调中加入自定义的逻辑，或提供额外信息跟随异常一起上报。

```
public interface CrashHandleListener {

    /**
     * Crash处理回调时，此接口返回的数据以附件的形式上报，附件名userExtraByteData
     * @param isNativeCrashed 是否Native异常
     * @param crashType 异常的类型
     * @param crashStack 异常堆栈
     * @param nativeSiCode native异常时的SI_CODE，非Native异常此数据无效
     * @param crashTime native异常的发生时间
     * @return 上报的附件字节流
     */
    byte[] getCrashExtraData(boolean isNativeCrashed, String crashType,
String crashAddress,
            String crashStack, int nativeSiCode, long crashTime);

    /**
     * Crash处理回调时，此接口返回的数据在附件extraMessage.txt中展示
     * @param isNativeCrashed 是否Native异常
     * @param crashType 异常的类型
     * @param crashStack 异常堆栈
     * @param nativeSiCode native异常时的SI_CODE，非Native异常此数据无效
     * @param crashTime native异常的发生时间
     * @return 上报的数据
     */
    String getCrashExtraMessage(boolean isNativeCrashed, String
crashType, String crashAddress,
            String crashStack, int nativeSiCode, long crashTime);

    /**
     * Crash处理回调前，执行此接口
```

```
     * @param isNativeCrashed 是否Native异常
     */
    void onCrashHandleStart(boolean isNativeCrashed);

    /**
     * Crash处理回调后，执行此接口
     * @param isNativeCrashed 是否Native异常
     * @return 返回值没有实际作用，不影响方法正常使用，可忽略
     */
    boolean onCrashHandleEnd(boolean isNativeCrashed);

    /**
     * Crash处理回调时，执行此接口
     * @param isNativeCrashed 是否NativeCrash
     * @param crashType Crash类型
     * @param crashMsg Crash消息， 例如 “Attempt to invoke virtual method
'int java.lang.String.length()' on a null object reference” （4.4.1.2新
增）
     * @param crashAddress Crash地址
     * @param crashStack Crash堆栈
     * @param nativeSiCode native异常时有效，SI_CODE
     * @param crashTime crash时间
     * @param userId crash时用户ID
     * @param deviceId crash时的设备ID
     * @param crashUuid 这条异常的唯一标识
     * @return 返回值没有实际作用，不影响方法正常使用，可忽略
     */
    boolean onCrashSaving(boolean isNativeCrashed, String crashType,
String crashMsg, String crashAddress,
        String crashStack, int nativeSiCode, long crashTime, String
userId, String deviceId,
        String crashUuid, String processName);
}
```

在初始化阶段，通过 BuglyBuilder 设置，如下所示：

```
CrashHandleListener crashHandleListener = xxx;
buglyBuilder.setCrashHandleListener(crashHandleListener);
```

设置 Crash 上报回调

终端性能监控 Pro 上报 Crash 异常时可以回调业务，业务可以在回调中加入自定义的逻辑。

```
public interface UploadHandleListener {

        /**
         * 上报开始时回调
         * int TYPE_JAVA_CRASH = 0;  // Java 崩溃
         * int TYPE_NATIVE_CRASH = 1; // Native崩溃
         * int TYPE_JAVA_CATCHED = 2; // Java错误上报
         * int TYPE_ANR_CRASH = 3; // ANR
         * int TYPE_U3D_CRASH = 4; // U3D错误上报
         * @param requestKey 上报关键字（4.4.3.4版本开始，这个字段表示上报异常个例类型，取值如上）
         */
        public void onUploadStart(int requestKey);

        /**
         * 上报结束后回调
         * int TYPE_JAVA_CRASH = 0;  // Java 崩溃
         * int TYPE_NATIVE_CRASH = 1; // Native崩溃
         * int TYPE_JAVA_CATCHED = 2; // Java错误上报
         * int TYPE_ANR_CRASH = 3; // ANR
         * int TYPE_U3D_CRASH = 4; // U3D错误上报
         * @param requestKey 上报关键字（4.4.3.4版本开始，这个字段表示上报异常个例类型，取值如上）
         * @param responseKey 回包的关键字
         * @param sended 发送的字节流
         * @param recevied 接受的字节流
         * @param result true则上报成功，否则失败
         * @param exMsg 额外信息
         */
        public void onUploadEnd(int requestKey, int responseKey, long sended, long recevied, boolean result, String exMsg);
}
```

在初始化阶段，通过 BuglyBuilder 设置，如下所示：

```
UploadHandleListener uploadHandleListener = null;
buglyBuilder.setUploadHandleListener(uploadHandleListener);
```

> **注意：**
> UploadHandleListener 的 requestKey 参数，从4.4.3.4版本开始，这个字段用于描述上报个例类型，取值如下：

- int TYPE_JAVA_CRASH = 0; // Java 崩溃
- int TYPE_NATIVE_CRASH = 1; // Native 崩溃
- int TYPE_JAVA_CATCHED = 2; // Java 错误上报
- int TYPE_ANR_CRASH = 3; // ANR
- int TYPE_U3D_CRASH = 4; // U3D 错误上报

其他接口

其他接口的调用均需在 SDK 初始化之后，初始化之前调用无效，SDK 提供其他接口功能如下。

更新设备 ID

```
/**
 * 更新设备id（初始化后）
 * @param context context
 * @param deviceId 设备id
 */
public static void updateUniqueId(Context context, String deviceId);
```

更新用户 ID

```
/**
 * 更新用户id（初始化后）
 * @param context context
 * @param userId 用户id
 */
public static void updateUserId(Context context, String userId);
```

更新设备型号

SDK 默认不获取手机型号，业务可以在合规获取到手机型号后通过该接口更新至终端性能监控 Pro SDK。

```
/**
 * 更新设备型号（初始化后）
 * @param context context
 * @param model 设备型号
 */
public static void updateDeviceModel(Context context, String model);
```

更新日志等级

```
/**
 * 更新日志等级（初始化后）
 * @param logLevel 日志等级，级别可从BuglyLogLevel中获取
 */
public static void updateLogLevel(int logLevel);
```

上报自定义异常

如业务需要上报除 Java Crash、Native Crash、ANR 之外的错误，如 C#异常等，可以通过自定义异常接口上报至终端性能监控 Pro，上报的数据在 终端性能监控 Pro > 错误 页面展示。

```
/**
 * 上报自定义异常
 * @param thread 出错线程，默认当前线程
 * @param category 异常类型 u3d c# : 4 | js : 8 | cocos2d lua : 6
 * @param errorType 错误类型
 * @param errorMsg 错误信息
 * @param stack 出错堆栈
 * @param extraInfo 额外信息
 */
public static void postException(Thread thread, int category, String
errorType, String errorMsg,
                                 String stack, Map<String, String>
extraInfo);
public static void postException(int category, String errorType, String
errorMsg,
                                 String stack, Map<String, String>
extraInfo);
```

上报 Java catch 异常

如业务在 Java 中通过 try catch 方式捕获到异常，希望上报到终端性能监控 Pro，可以使用上报 Java catch 异常接口，上报的数据在 终端性能监控 Pro > 错误 页面展示。

```
/**
 * 处理catch异常并上报.
 * @param thread 出错线程
 * @param exception 异常
 * @param extraMsg 额外信息
 * @param extraData 额外数据
 * @param enableAllThreadStack 开启全部线程抓栈，默认开启
 * @return 上报结果
```

```
 */
public static boolean handleCatchException(Thread thread, Throwable
exception, String extraMsg,
                                           byte[] extraData, boolean
enableAllThreadStack);
public static boolean handleCatchException(Thread thread, Throwable
exception, String extraMsg,
                                           byte[] extraData);
```

设置个例标签

```
/**
 * 非线程安全，推荐切到相同线程操作
 * 设置个例标签，来自终端性能监控Pro管理台的标签ID
 * 1. 先在终端性能监控Pro管理台添加标签，得到标签ID；
 * 2. 通过setCaseLabels设置标签ID，需要设置多个时，通过英文竖线分割；
 * 3. 异常个例上报时会同时上报这些标签ID；
 * 示例："123|456|789"
 * SDK版本：4.3.2.9+
 * @param labels 个例标签
 */
public static void setCaseLabels(String labels);
```

设置业务下钻

```
/**
 * 非线程安全，推荐切到相同线程操作
 * 设置「业务下钻」标签，设置多个标签时通过英文的竖线分割
 * 最多只允许设置30个标签，超出时，只取前面30个
 * 每个标签最长不超过1024个字符，超出时会添加失败
 * 示例："test_one|test_two|test_three"
 * SDK版本：4.4.1+
 * @param labels 业务下钻标签
 */
public static void setTestLabels(String labels);
```

设置 Java 内存泄漏嫌疑对象（性能监控）

在开启终端性能监控 Pro Java 内存泄漏监控后，可以通过如下接口设置疑似泄漏的对象给 SDK 进行监控，若不设置该接口，SDK 只对 Activity/Fragment 对象的内存泄漏进行监控。

```
/**
 * 设置Java内存泄漏嫌疑对象
 * @param leakObj 嫌疑对象
 */
public static void startInspectLeakObj(Object leakObj);
```

测试接口

SDK 提供接口测试崩溃异常，业务在调试时可以使用该接口进行崩溃采集上报的测试。

```
/**
 * 测试crash
 * @param crashType crash类型，可填入Bugly.JAVA_CRASH、Bugly.NATIVE_CRASH、
Bugly.ANR_CRASH
 */
public static void testCrash(@CrashTypeKey int crashType);
```

业务自定义日志接口

SDK 提供异常时业务自定义日志携带能力，业务可以在崩溃发生前（不含回调）调用以下接口写入自定义日志。

```
public class BuglyCustomLog {
    public static void v(String tag, String msg) {
        BuglyLog.v(tag, msg);
    }

    public static void d(String tag, String msg) {
        BuglyLog.d(tag, msg);
    }

    public static void i(String tag, String msg) {
        BuglyLog.i(tag, msg);
    }

    public static void w(String tag, String msg) {
        BuglyLog.w(tag, msg);
    }

    public static void e(String tag, String msg) {
        BuglyLog.e(tag, msg);
    }
```

```
    public static void e(String tag, String msg, Throwable throwable) {
        BuglyLog.e(tag, msg, throwable);
    }

    /**
     * 设置写入日志的缓存大小，日志将根据设置的缓存大小循环写入
     * @param size 日志缓存大小
     */
    public static void setCache(int size) {
        BuglyLog.setCache(size);
    }
}
```

在写入自定义日志后，如果进程发生异常，会将自定义日志进行上报展示，展示为 **附件** 中的 `BuglyLog.zip` 。

自定义数据

添加/移除自定义数据（Crash 监控）

SDK 支持业务添加自定义数据，可以保存在 Crash 发生时、或发生前的一些自定义环境信息。自定义数据会随异常一起上报，并在异常个例详情界面的**自定义字段**中展示。自定义数据最多支持50对 key-value。

```
/**
 * 添加自定义数据
 * @param context context
 * @param key key
 * @param value value
 */
public static void putUserData(Context context, String key, String
value);
```

如需移除自定义数据，可通过如下接口进行移除操作。

```
/**
 * 移除用户自定义数据.
 * @param key key
 * @return 如果存在，则返回对应key的值
 */
public static String removeUserData(Context context, String key);
```

设置/更新自定义文件上传路径（Crash 监控）

SDK 支持业务上传自定义大文件，并关联至异常个例。自定义文件的上传采用独立于异常上报的通道，对异常处理流程无干扰，且支持配置多个文件。业务可以在发生异常之前的任意时间段，通过如下接口设置/更新文件路径的数组。进程发生异常重启后，会上报这些自定义文件，在异常个例的**附件**中，展示为custom_log.zip。

```
/**
 * 设置或更新自定义文件路径数组
 * @param files 文件路径数组，最多10个文件，且压缩后大小不能超过10MB
 * @return 设置结果
 */
public static boolean setAdditionalAttachmentPaths(String[] files);
```

⚠ 重要提醒：

- 该接口最多设置10个文件路径。
- 需上传文件压缩后的大小限制为10MB，若超过该大小，则不会被上传。
- 该接口可被多次调用，更新的文件路径会覆盖之前的值。
- 自定义文件的上传时机在进程二次启动时，与异常上报间存在一定延迟。
- 频繁发生崩溃异常会触发文件清理逻辑，可能会导致文件漏传。
- 可以通过配置设置自定义文件上传的采样率。

进入/退出自定义场景（性能监控）

在性能监控中，业务可以自定义场景，并根据场景来执行特定的监控或回调操作。使用如下接口来进入、退出自定义场景。

```
/**
 * 进入自定义场景
 * @param sceneName 场景名称
 */
public static void enterScene(String sceneName);

/**
 * 退出自定义场景
 * @param sceneName 场景名称
 */
public static void exitScene(String sceneName);
```

⚠ 注意：

- enterScene --- exitScene 期间，各监控能力监控到问题时，上报的场景取[自定义场景]。

- exitScene 后，各监控能力监控到问题时，上报的场景取 [simpleName Of lastResumedActivity]。
- 推荐使用 enterScene/exitScene 成对调用，形如：enterScene(A) -> exitScene(A) -> enterScene(B) -> exitScene(B)。如果没有使用成对调用，如下示例：
  - enterScene(A) --> 自定义场景为 A
  - enterScene(B) --> 自定义场景为 B
  - enterScene(C) --> 自定义场景为 C
  - exitScene(B) --> 自定义场景为 C
  - exitScene(C) --> 自定义场景为空
  - exitScene(A) --> 自定义场景为空
- 如果 sceneName 为空，直接返回。

添加/移除自定义数据采集器（性能监控）

SDK 性能监控模块的自定义设置方式不同于 Crash 监控模块，性能监控模块的自定义数据支持两类：**自定义维度**与**自定义字段**。

- **自定义维度**：指由 SDK 指定了三组 Key（详情请参见 ICustomDataEditor），每组10个。应用可以根据需要，选择合适的 Key 上报数据。在控制管理台，用户可以给这些 Key 设置别名，方便查看和分析。对于自定义维度，服务器存储时是一个字段，一个字段分开存储的。后续可以提供丰富的查询和分析能力。当前支持全局设置，或者数据上报前的回调设置。当前支持通过 `Bugly.getGlobalCustomDataEditor` 获取全局接口，设置自定义维度，也可以通过指标类回调（ `ICustomDataCollector` ）或问题类回调（ `ICustomDataCollectorForIssue` ），设置或者更新自定义维度。
- **自定义字段**：指 Key 和 Value 都由应用自由设置，SDK 不理解字段的类型，统一作为字符串存储。用户可以在控制管理台，在问题列表和问题详情，通过自定义字段查询包含指定内容的上报。当前只支持通过问题类回调 `ICustomDataCollectorForIssue` ，设置自定义字段。

全局设置

全局设置指应用可以随时调用全局设置接口，设置自定义维度，相同 Key 的数据，重复设置表示更新。性能上报在需要上报时，会获取全局缓存的自定义维度数据。
可以通过如下接口来获取默认的全局自定义维度收集回调，并设置全局自定义维度。

```
/**
 * 获取全局自定义维度设置接口
 * @return 全局自定义维度设置类
 */
public static ICustomDataEditor getGlobalCustomDataEditor();

// 设置示例
ICustomDataEditor customDataEditor = Bugly.getGlobalCustomDataEditor();
```

```
customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());
customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
Debug.getPss());
```

回调设置

回调设置指应用设置回调，在监控到相关问题上报前，会回调应用的接口，应用根据具体的问题及场景，分别采集额外的现场信息。回调设置分为，指标类回调以及问题类回调。

```
// 添加回调，指标类回调和问题类回调需要分别设置。
Bugly.addCustomDataCollector(customDataEditor);

// 移除回调，指标类回调和问题类回调需要分别移除。
Bugly.removeCustomDataCollector(customDataDditor);
```

- **指标类回调：** `ICustomDataCollector` ，如卡顿指标，内存峰值属于指标类回调。会在信息采集，缓存数据前，回调应用是否需要补充额外信息。

```
/**
 * 设置指标数据回调，对于不同监控项、不同场景可设置多个
 * @param dataCollector 全局自定义数据回调类
 */
public static void addCustomDataCollector(ICustomDataCollector
dataCollector);
/**
 * 移除全局自定义数据回调
 * @param dataCollector 全局自定义数据回调类
 */
public static void removeCustomDataCollector(ICustomDataCollector
dataCollector);

// 设置示例
ICustomDataEditor customDataEditor = new ICustomDataCollector() {
    @Override
    public void collectCustomData(String pluginName, String scene,
ICustomDataEditor customDataEditor) {

customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());

customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
```

```
Debug.getPss());
    }
};

// 设置指标类回调
Bugly.addCustomDataCollector(customDataEditor);
// 移除指标类回调
Bugly.removeCustomDataCollector(customDataDditor);
```

- **问题类回调：** `ICustomDataCollectorForIssue` **，如卡顿问题监控，Java内存泄露，大图分析属于问题类回调，会在监控到异常问题时，回调用户是否需要补充额外信息。**

```
/**
 * 设置单个指标、问题自定义数据回调，对于不同监控项、不同场景可设置多个
 * @param issueDataCollector 单个指标、问题自定义数据回调类
 */
public static void addCustomDataCollector(ICustomDataCollectorForIssue
issueDataCollector);
/**
 * 移除单个指标、问题自定义数据回调
 * @param issueDataCollector 单个指标、问题自定义数据回调类
 */
public static void
removeCustomDataCollector(ICustomDataCollectorForIssue
issueDataCollector);

// 设置示例
ICustomDataCollectorForIssue issueDataCollector = new
ICustomDataCollectorForIssue() {
    @Override
    public void collectCustomData(String pluginName, String scene,
ICustomDataEditorForIssue customDataEditor) {
        // 允许覆盖全局设置的自定义维度

customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());

customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
Debug.getPss());
    }
}
// 设置问题类回调
Bugly.addCustomDataCollector(issueDataCollector);
```

```
// 移除问题类回调
Bugly.removeCustomDataCollector(issueDataCollector);
```

动态开关

如业务需在指定场景中，动态开关终端性能监控 Pro Crash 监控项或性能监控项，可以通过如下接口进行动态开关。但需注意，崩溃异常的动态开关非必要不应使用，会导致异常的漏报。

```
/**
 * crash监控动态开关
 * @param crashType crash类型，可填入Bugly.JAVA_CRASH、Bugly.NATIVE_CRASH、
Bugly.ANR_CRASH
 * @param isAble true 打开，false 关闭
 */
public static void setCrashMonitorAble(@CrashTypeKey int crashType,
boolean isAble);

/**
 * 性能监控动态开关（开关一组监控项）
 * @param monitorList 监控项list，可从BuglyMonitorName中获取
 * @param isAble true 打开，false 关闭
 */
public static void setPerformanceMonitorsAble(List<String> monitorList,
boolean isAble);

/**
 * 性能监控动态开关（开关单个监控项）
 * @param monitorName 监控项
 * @param isAble true 打开，false 关闭
 */
public static void setPerformanceMonitorAble(String monitorName, boolean
isAble);

/**
 * 停止所有的性能监控项
 */
public static void abolishPerformanceMonitors();
```

启动监控

默认情况下，启动监控取首个 Activity 的首帧渲染结束作为启动结束点，应用也可以通过 `reportAppFullLaunch` 接口自定义启动结束点。除此之外，AppLaunch 还提供添加自定义标签接口，以及自

定义打点接口。

```
AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
appLaunch.reportAppFullLaunch();
```

- **自定义标签**：为本次启动数据添加自定义标签，方便后续做下钻分析，如下所示：

```
AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
appLaunch.addTag("show_splash");
```

- **自定义打点**：即在启动过程中，针对启动流程，监控一些耗时任务。

```
AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
appLaunch.spanStart("login", null); // 登陆任务开始
....
appLaunch.spanStart("verification_code", "login"); // 验证码任务开始
....
appLaunch.spanEnd("verification_code"); // 验证码结束
....
appLaunch.spanEnd("login"); // 登陆结束
```

AppLaunch 接口的详细说明：

```
public interface AppLaunch {
    /**
     * 开始启动监控
     * @param context Context
     */
    void install(Context context);

    /**
     * 添加自定义标签
     * @param tag 自定义标签
     */
    void addTag(String tag);

    /**
     * Span开始打点
     * @param spanName 开始打点的span名称
     * @param parentSpanName 当前要打点的span的父span的名称
     */
```

```
    void spanStart(String spanName, String parentSpanName);

    /**
     * Span结束打点
     * @param spanName 结束打点的span名称
     */
    void spanEnd(String spanName);

    /**
     * 标记启动完成
     */
    void reportAppFullLaunch();
}
```

iOS SDK 接入指引
SDK 接入简介

最近更新时间：2025-08-20 17:25:54

终端性能监控 Pro SDK 由腾讯 Bugly 团队提供技术支持。本文介绍如何在 iOS 平台接入终端性能监控 Pro SDK。接入 SDK 后，验证数据上报成功，即可在控制管理台使用相关分析功能。

SDK 简介

- **SDK 版本**：2.8.1.5
- **SDK 功能**：专业的应用质量监控工具，提供异常数据的采集和分析服务，帮助开发者及时发现并解决异常问题，打造高质量 App。
- **服务提供方**：腾讯云计算（北京）有限责任公司
- 终端性能监控 Pro SDK 合规使用指南
- 终端性能监控 Pro SDK 个人信息保护规则

接入前准备

1. 在接入 SDK 之前，请务必认真阅读 终端性能监控 Pro SDK 合规使用指南 。
2. 完成 SDK 集成 。
3. 完成 SDK 初始化 。SDK 在初始化过程中，可能会采集部分用户信息，请在用户同意 终端性能监控 Pro SDK 个人信息保护规则 之后，再进行 SDK 的初始化。在初始化 SDK 前，不会收集任何信息。
4. 验证 数据上报 。
5. 详细了解其他 SDK 功能，请参见 API 说明 。

SDK 集成

最近更新时间：2025-08-20 17:25:54

终端性能监控 Pro SDK 由腾讯 Bugly 团队提供技术支持，与 Bugly 专业版共用 SDK。支持自动集成和手动集成两种方式。

版本升级提醒

因 SDK 中存在缓存数据等，旧版本无法做到对新版本缓存数据的全兼容，**故一般情况下升级后的 SDK 强烈不推荐进行降级处理**，可能存在缓存读取异常的情况。若存在降级的需求，请 联系我们 。

前提条件

终端性能监控 Pro 需要使用 Bugly iOS SDK 2.8.1.5及以上版本。

自动集成（推荐）

1. 在工程的 Podfile 里面添加以下代码。

```
pod 'BuglyPro','2.8.1.5' // 此为示例版本，大家参考 SDK 更新日志，获取最新版本信息
```

2. 保存并执行 `pod install` ，然后用后缀为 .xcworkspace 的文件打开工程。

```
pod install
```

手动集成（不推荐）

如果您因特殊业务诉求、安全性要求、版本控制等原因无法采用自动集成方式集成 SDK，也可以手动集成 SDK。

1. 下载终端性能监控 Pro iOS SDK 的 xcframework 文件。

   终端性能监控 Pro iOS SDK 下载地址： Central Repository: com/tencent/bugly
2. 拖拽 BuglyPro.xcframework 文件到 Xcode 工程内（请勾选 **Copy items if needed** 选项）。
3. 添加依赖库。
   - SystemConfiguration.framework
   - Security.framework
   - Network.framework
4. 检查工程配置。

   在 Xcode 工程 -Build Settings 中，Other Linker Flags 检查是否有添加 **-ObjC**，若没有该配置项，需要手动添加。

SDK 初始化

最近更新时间：2025-08-20 17:25:54

本文将为您介绍如何初始化 SDK。

操作步骤

参考以下代码初始化终端性能监控 Pro SDK，在 App 启动后初始化监控框架，一般推荐在 `application:didFinishLaunchingWithOptions:delegate` 中进行初始化。

1. 在 AppDelegate 实现文件中引入对应的头文件。

```
#import <BuglyPro/Bugly.h>
#import <BuglyPro/BuglyConfig.h>
#import <BuglyPro/BuglyDefine.h>
```

2. 在 didFinishLaunchingWithOptions 中启动框架。

```
// 使用产品对应的 APP_ID 和 APP_KEY 创建一个 config 对象
BuglyConfig *config = [[BuglyConfig alloc] initWithAppId:APP_ID
appKey:APP_KEY];
// 必选，指定上报域名类型
config.serverHostType = BuglyServerHostTypeCloud;
// 可选，建议设置，指定当前构建版本的类型，会在后续配置中用到该字段
config.buildConfig = BuglyBuildConfigGray;
// 可选，Bugly 回调
config.delegate = self;
// 设置 device id 和 user id，如果可以尽量在此时设置
config.deviceIdentifier = @"device_id";
config.userIdentifier = @"user_id";

NSArray *modules = @[BUGLY_MODULE_CRASH, RM_MODULE_LOOPER,
RM_MODULE_MEMORY];
// NSArray *modules = RM_MODULE_ALL;
[Bugly start:modules config:config completeHandler:^{
    // SDK 初始化完成后的回调
}];
```

> ⚠ **注意：**
>
> - 设备 ID 非常重要，终端性能监控 Pro 使用设备 ID 来计算设备异常率，强烈建议应用设置正确的设备 ID，以确保设备的唯一性。

- 性能监控项可在 应用配置 > SDK 配置 中进行采样调整，通过调整设备采样率来开启或者关闭性能监控项。
- 建议在用户授权 《终端性能监控 Pro SDK 个人信息保护规则》 后再初始化 SDK。

数据上报验证

最近更新时间：2025-09-02 11:50:02

前提条件

- 崩溃是100%上报，不支持采样。除崩溃外，其他监控能力都支持采样，默认是全部关闭。
- 需要在 终端性能监控 Pro > 应用配置 > SDK 配置 页面创建配置任务。有关配置功能的详细说明请参见 SDK 配置 。
- 在接入期间，建议参考 SDK 配置 设置**测试号码白名单**，即将所有采样率调整为1.0，方便验证数据上报。接入完成后，再根据需要调整采样率。或者创建多个配置任务，根据版本类型（对应初始化的 `BuglyConfig` 中的 `buildConfig` ），配置开发任务、灰度任务、以及正式任务。开发任务打开所有的监控项，灰度阶段按需采样，正式发布根据需要降低采样。
- 检查待接入的产品是否已经 购买并绑定 生效中的资源包。如果产品没有绑定生效中的资源包，该产品将无法上报数据。

崩溃监控

> **说明：**
> 由于崩溃捕获依赖系统的 signal 监听接口，若 App 中有接入其他类似能力的 SDK 或有相关逻辑存在监听 signal 的逻辑，可能会影响 SDK 对崩溃的捕获成功率。这种情况下，可确保 SDK 初始化在其他监听逻辑注册之后，可减少对 Crash 捕获的影响。

模拟异常

初始化完成后，可以模拟 OC 异常、C++异常、信号异常来验证 SDK 的上报情况，详细可以参考 Demo 。
崩溃发生后，部分异常问题需要在第二次启动才能完成上报。上报可能存在延时，测试时建议触发异常后，重新启动App，再刷新展示界面。

```
// 模拟 OC 异常
id data = [NSArray arrayWithObject:@"Hello World"];
[(NSDictionary *)data objectForKey:@""];

// 模拟信号异常
abort();

// 模拟 C++异常
throw "Something went wrong";
```

检查数据上报

崩溃上报后，在 崩溃 > 问题列表 中可以看到上报的问题，点击进入问题详情，即可查看上报详情。

ANR 监控

跟崩溃类似，可以通过如下的测试接口来模拟 ANR，也可以自行在 UI 线程执行异常耗时任务。

```
// 模拟ANR
while(YES)
{
    ;
}
```

ANR 上报后，在 ANR > 问题列表 可以看到上报的问题，点击进入问题详情，即可查看上报详情。

FOOM 监控

FOOM 率：FOOM（Foreground Out Of Memory）一般指 App 因为内存使用过高，被系统意外杀死的情况。由于 FOOM 判断原理的原因，因此也会包含其他原因导致的 App 被系统意外杀死的情况。可以通过如下的测试接口来模拟 FOOM：

```
- (void)foomButtonClicked {
    self.timer = [NSTimer scheduledTimerWithTimeInterval:0.01
target:self selector:@selector(timerFired) userInfo:nil repeats:YES];
}

- (void)timerFired {
    CGSize size = CGSizeMake(1024 , 1024);
    const size_t bitsPerComponent = 8;
    const size_t bytesPerRow = size.width * 4;
    CGContextRef ctx = CGBitmapContextCreate(calloc(sizeof(unsigned
char), bytesPerRow * size.height), size.width, size.height,
                                             bitsPerComponent,
bytesPerRow,

CGColorSpaceCreateDeviceRGB(),

kCGImageAlphaPremultipliedLast);
    CGContextSetRGBFillColor(ctx, 1.0, 1.0, 1.0, 1.0);
    CGContextFillRect(ctx, CGRectMake(0, 0, size.width, size.height));
}
```

检查数据上报

OOM 上报后，在 OOM > 问题列表 可以看到上报的问题，点击进入问题详情，即可查看上报详情。

错误监控

错误一般是指用户自定义的异常，如已经 Catch 的 OC/C++异常，或者 C#异常、JS 异常、lua 异常等。一般通过 `BuglyCrashMonitorPlugin.h` 中的 `reportException:`、`reportError:` 或 `reportExceptionWithCategory:name:reason:callStack:extraInfo:terminateApp:` 来上报（详细参考其他接口/上报自定义异常部分）。

模拟异常

- 上报自定义错误，可以参考以下示例代码：

```
- (void)userDefinedCrash {
    NSError *error = [NSError errorWithDomain:@"Error Test" code:-2
userInfo:nil];
    [BuglyCrashMonitorPlugin reportError:error];
}
```

- 上报 OC/C++异常，可以参考以下示例代码：

```
- (void)userDefinedCrash {
    @try {
        NSString *value = @"This is a string";
        [NSException raise:@"TurboEncabulatorException"
                    format:@"Spurving bearing failure: Barescent skor
motion non-sinusoidal for %p", value];
    } @catch (NSException *exception) {
        [BuglyCrashMonitorPlugin reportException:exception];
    }
}
```

- 上报自定义异常，参考以下示例代码：

```
    NSArray * arr = [NSThread callStackSymbols];
    [BuglyCrashMonitorPlugin reportExceptionWithCategory:3
name:@"reportTest" reason:@"test" callStack:arr extraInfo:
[NSDictionary dictionary] terminateApp:false];
```

检查数据上报

错误上报后，在 错误 > 问题列表 看到上报的问题，点击进入问题详情，即可查看上报详情。

> **说明：**
> `reportExceptionWithCategory:name:reason:callStack:extraInfo:terminateApp:` 的参数中，callStack 设置的出错堆栈，extraInfo 展示在附件中。

启动监控

在开启启动监控的情况下，启动模块会自动安装监控，并且在 SDK 初始化后，进行启动数据上报。用户可以通过 `[BuglyLaunchMonitorPlugin endColdLaunch]` 接口自定义启动结束点，也支持自定义标签以及通过打点接口，监控启动过程中的耗时任务。如需了解更多关于启动接口的使用，请参见 API 说明–启动监控 部分。

```
[BuglyLaunchMonitorPlugin startSpan:@"parentSpan" parentSpanName:nil];
[BuglyLaunchMonitorPlugin startSpan:@"spanTest"
parentSpanName:@"parentSpan"];

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
    [BuglyLaunchMonitorPlugin endSpan:@"spanTest"];
    [BuglyLaunchMonitorPlugin endSpanFromLaunch:@"spanFromLaunch"];
});

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(4.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
    [BuglyLaunchMonitorPlugin endSpan:@"parentSpan"];
});

[BuglyLaunchMonitorPlugin addTag:@"tagTest1"];
[BuglyLaunchMonitorPlugin addTag:@"tagTest2"];
```

检查数据上报

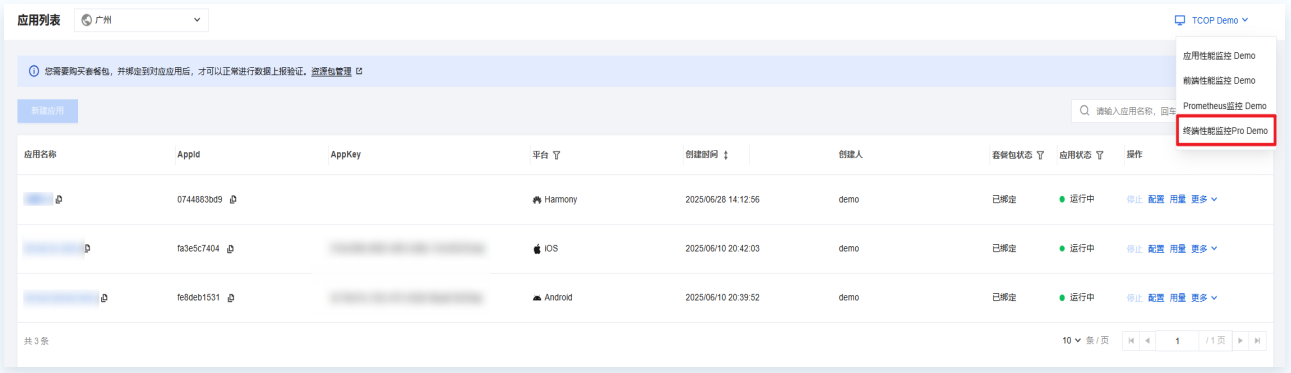
可在 启动 > 启动个例 查看上报详情。看到上报的问题，点击进入问题详情，即可查看上报详情。

卡顿监控

- **卡顿指标：FPS 及挂起率**

  在开启卡顿指标（ `"looper.fluency"` ）的情况下，SDK 会在初始化完成后，监控应用的流畅度情况。在应用运行一次的期间，会先收集数据，保存至本地，下次启动时再聚合上报。为了避免影响应用的启动，SDK 会在初始化5分钟后，再将缓存的数据聚合上报后台。
  - FPS：帧率，即单位时间内刷新次数 / 单位时间。因为 iPhone 13的高刷屏的出现，FPS 采用一种归一化的计算方案，即将大于60 FPS的数据归一化为 60。
  - 挂起率：挂起率定义为单位时间内回调间隔超过200ms的时间和除以单位总时间 * 3600，即单位是 s/h。

说明：
可以通过在 Demo 中测试滑动卡顿来观察列表滑动情况下的 FPS 以及挂起率。您可在 终端性能监控 Pro > 应用列表 页面右上角选择**终端性能监控 Pro Demo**。



- **卡顿监控**

  开启卡顿监控（ `"looper.hitches"` ）后，在 UI 线程执行耗时逻辑，耗时超过500ms的情况下，会触发卡顿上报。卡顿监控通过监控 UI 线程的消息执行来判断当前 UI 线程是否发生卡顿。
  在验证阶段，建议将卡顿监控的 `"event_sample_ratio"` （消息采样率）以及 `"sample_ratio"` （设备采样率）都设置为1。这样只要满足卡顿的耗时阈值，即可触发卡顿上报。

模拟异常

可以在 tableView 的回调方法中添加一个耗时操作，参考以下示例代码模拟一个卡顿上报：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:@"defaultcell"];
    cell.textLabel.text = [self.dataList objectAtIndex:indexPath.row];
    cell.textLabel.font = [UIFont systemFontOfSize:80];

    BOOL hitch = arc4random() % 5;
    if (hitch) {
        [NSThread sleepForTimeInterval:0.02];
    }

    return cell;
}
```

检查数据上报

卡顿上报后，在 卡顿 > 问题列表 可以看到上报的问题，点击进入问题详情，即可查看上报详情。

内存监控

峰值内存

峰值内存表示在 App 的整个生命周期中，最大的 phys_footprint 的值。峰值内存越高，意味着极端情况下 App 使用内存越多。
数据上报后，在 内存 > 指标分析 可以看到上报的问题。

VC 内存泄漏

VC 内存泄漏表示监控 iOS App 开发框架中的 UIViewController 的泄漏情况。可以通过以下示例代码来验证：

```
    UIStoryboard *storyboard = [UIStoryboard
storyboardWithName:@"RMVCLeakExample"
                                                        bundle:
[NSBundle mainBundle]];

    UIViewController * subVC = [storyboard
instantiateViewControllerWithIdentifier:@"rm.vcleak.subvc"];

    [self.navigationController pushViewController:subVC animated:YES];

    self.holdVC = subVC;
```

泄漏发生并且上报后，在 内存 > VC 内存泄漏 > 问题列表 可以看到上报的问题，点击进入问题详情，即可查看上报详情。

大块内存分配监控

大内存分配监控的目的是针对 App 中使用大块内存的情况进行监控，让业务感知 App 中使用内存较为严重的业务逻辑。大内存监控在 App 触发大于一定阈值的内存分配时，收集触发内存分配的堆栈、当前的业务场景、内存使用情况等数据并上报。数据上报后，在 内存 > 大块内存分配 可以看到上报的问题。

大内存分配不一定是一个明确问题，但是可以让业务得知对应的业务逻辑中有较大的内存消耗，需要关注和优化，避免内存快速消耗，导致发生 FOOM 的情况。

API 说明

最近更新时间：2025-09-02 11:50:02

本文详细介绍终端性能监控 Pro SDK 的各功能接口，帮助您更加灵活、深度地使用 SDK。

初始化接口

SDK 在初始化时，BuglyConfig 除了设置 appId 和 appKey 以外，还可以设置一系列用户参数。

```
/// 设置自定义版本号
@property (nonatomic, copy) NSString *appVersion;
/// 设置自定义设备唯一标识
@property (nonatomic, copy) NSString *deviceIdentifier;
/// 自定义用户id
@property (nonatomic, copy) NSString *userIdentifier;
/// build config，用于拉取不同的配置
@property (nonatomic, assign) BuglyBuildConfig buildConfig;

/// Bugly Delegate
@property (nonatomic, assign) id<BuglyDelegate> delegate;
```

回调接口

设置 Crash 附件信息回调

SDK 处理 Crash 异常时可以回调业务，业务可以在回调中加入自定义的逻辑，或提供额外信息跟随异常一起上报。

```
/// 允许业务带附件到个例上报中
/// @param eventType 个例事件类型
/// @param customData 业务自定义字段
/// @param eventInfo 个例信息，暂为空
- (NSDictionary<NSString *, NSString *> *)attachmentForEvent:(NSString
*)eventType
                                                   customData:
(BuglyCustomData *)customData
                                                    eventInfo:
(NSDictionary *)eventInfo;

#### 2. 回调 Crash 信息供设置信息
```

Bugly上报Crash异常时可以回调业务，业务可以在回调中加入自定义的逻辑。

``` objective-c
/**
 *  发生异常时回调
 *  @param exception 异常信息
 *  @return 返回需上报记录，随异常上报一起上报
 */
- (NSString * __nullable)attachmentForException:(NSException *
__nullable)exception;
```

设置日志回调接口

1. 实现一个函数回调方法。

```
/// 自定义 log 函数
static void logger_func(RMLoggerLevel level, const char *log) {
    //handle sdk log.
}
```

2. 并注册该方法。

```
// 注册 Bugly 日志输出
[Bugly registerLogCallback:logger_func];
```

其他接口

其他接口的调用均需在 SDK 初始化之后，初始化之前调用无效，SDK 提供其他接口功能如下。

更新设备 ID

```
/**
* 更新设备 id
* @param deviceId 设备 id
*/
+ (void)updateDeviceIdentifier:(NSString *)deviceId;
```

更新用户 ID

```
/**
* 更新userId
```

```
 * @param userId 用户id
 */
+ (void)updateUserIdentifier:(NSString *)userId;
```

获取 SDK 版本号

```
/**
 *  获取 SDK 版本信息
 *  @return SDK版本号
 */
+ (NSString *)sdkVersion;
```

上报自定义异常

如业务需要上报除 OC/C++ 之外的错误，如 C#/js 等异常，可以通过自定义异常接口上报至终端性能监控 Pro，上报的数据将在 错误 页面中展示。

```
/**
 *     @brief 上报自定义错误
 *
 *     @param category     类型(Cocoa=3,CSharp=4,JS=5,Lua=6)
 *     @param aName        名称
 *     @param aReason      错误原因
 *     @param aStackArray 堆栈
 *     @param info         附加数据
 *     @param terminate    上报后是否退出应用进程
 */
+ (void)reportExceptionWithCategory:(NSUInteger)category
                               name:(NSString *)aName
                             reason:(NSString *)aReason
                          callStack:(NSArray *)aStackArray
                          extraInfo:(NSDictionary *)info
                       terminateApp:(BOOL)terminate;
```

上报自定义错误

如业务希望把 NSError 信息上报到终端性能监控 Pro，可以使用上报错误的接口，上报的数据将在 错误 页面中展示。

```
/**
 *  上报错误
```

```
 *
 *  @param error 错误信息
 */
+ (void)reportError:(NSError *)error;
```

上报 OC 异常

如业务通过 try catch 方式捕获到异常，希望上报到终端性能监控 Pro，可以使用上报异常接口，上报的数据将在 错误 页面中展示。

```
/**
 *  上报自定义Objective-C异常
 *
 *  @param exception 异常信息
 */
+ (void)reportException:(NSException *)exception;
```

设置个例标签

```
/**
 * 更新个例标签，需要在 bugly sdk 完成初始化（Bugly setup completeHandler 回调）后调用，否则可能导致数据丢失
 * SDK版本: 2.7.53.3+
 * @params tagArr 字符串数组，字符串限长 1024 字节，数组限长 30
 */
+ (void)updateCaseTags:(NSArray<NSString *> *)tagArr;
```

设置业务下钻

```
/**
 * 更新业务下钻，需要在 bugly sdk 完成初始化（Bugly setup completeHandler 回调）后调用，否则可能导致数据丢失
 * SDK版本: 2.7.53.3+
 * @params tagArr 字符串数组，字符串限长 1024 字节，数组限长 30
 */
+ (void)updateTestTags:(NSArray<NSString *> *)tagArr;
```

自定义数据

添加自定义数据（Crash 监控）

终端性能监控 Pro 支持业务添加自定义数据，可以保存在 Crash 发生时，或发生前的一些自定义环境信息。自定义数据会随异常一起上报，并在异常个例详情界面的**自定义字段**中展示。自定义数据最多支持50对 key-value。

```
/**
 *  设置关键数据，随崩溃信息上报
 *
 *  @param value KEY
 *  @param key VALUE
 */
+ (void)setUserValue:(NSString *)value forKey:(NSString *)key;
```

获取已设置自定义数据，可通过如下接口进行操作。

```
/**
 *  获取自定义数据
 *
 *  @return 关键数据
 */
+ (NSDictionary * _Nullable)allUserValues;
```

设置/更新自定义文件上传路径（Crash 监控）

终端性能监控 Pro 支持业务上传自定义大文件，并关联至异常个例。自定义文件的上传采用独立于异常上报的通道，对异常处理流程无干扰，且支持配置多个文件。业务可以在发生异常之前的任意时间段，通过如下接口设置/更新文件路径的数组。进程发生异常重启后，会上报这些自定义文件，将上报至异常个例的**附件**中，展示为 custom_log.zip。

```
/**
 * 设置自定义附件的绝对路径的集合。
 * 文件压缩后的大小不大于10M，二次启动时上报。
 */
+ (void)setAdditionalAttachmentPaths:(NSArray<NSString *>*)pathArray;
```

> **注意：**
> - 该接口最多设置10个文件路径。
> - 需上传文件压缩后的大小限制为10MB，若超过该大小，则不会被上传。
> - 该接口可被多次调用，更新的文件路径会覆盖之前的值。
> - 自定义文件的上传时机在进程二次启动时，与异常上报间存在一定延迟。
> - 频繁发生崩溃异常会触发文件清理逻辑，可能会导致文件漏传。

- 可以通过配置设置自定义文件上传的采样率，配置方式可以参见 SDK 配置 。
- **在崩溃发生时调用此接口无法生效**，因此需要业务提前设置路径。

进入/退出自定义场景（性能监控）

在性能监控中，业务可以自定义场景，并根据场景来执行特定的监控或回调操作。使用如下接口来进入、重置自定义场景。

- **设定自定义场景**

```
/// 更新场景信息
/// @param key 场景 id，卡顿、内存、资源监控等功能会根据该值进行聚类
+ (void)setScene:(NSString *)key;
```

- **重置自定义场景**

```
/// 结束自定义场景信息，继续使用 RMonitor 自动获取的场景信息
+ (void)resetScene;
```

添加/移除自定义数据采集器（性能监控）

终端性能监控 Pro 性能监控模块的自定义设置方式不同于 Crash 监控模块，性能监控模块的自定义数据支持两类：自定义维度和自定义字段。

- **自定义维度**：指由 SDK 指定了三组 Key（详细见 `BuglyCustomData.h` ），每组10个。应用可以根据需要，选择合适的 Key 上报数据。在控制管理台，用户可以给这些 Key 设置别名，方便查看和分析。对于自定义维度，服务器存储时是每个字段分开存储的。后续可以提供丰富的查询和分析能力。当前支持全局设置，或者数据上报前的回调设置。
- **自定义字段**：指 Key 和 Value 都由应用自由设置，SDK 不理解字段的类型，统一作为字符串存储。用户可以在控制管理台，在问题列表和问题详情，通过自定义字段查询包含指定内容的上报。

说明：
全局设置指应用可以随时调用全局设置接口，设置自定义维度，相同key的数据，重复设置表示更新。性能上报在需要上报时，会获取全局缓存的自定义维度数据。

可以通过如下接口来获取默认的全局自定义维度收集回调，并设置全局自定义维度。

```
/**
 * 添加自定义上报的 tag, SDK 初始化前调用无效
 * @param data 需要更新的自定义字段
 */
+ (void)updateCustomData:(BuglyCustomData *)data;
```

```
/**
 * 获取当前自定标签的副本信息，若未设置，返回空
 * @return 已设置的用户自定义字段
 */
+ (nullable BuglyCustomData *)currentCustomData;

/**
 * 添加自定义数据为特定的事件上报
 * @param data 需要更新的自定义字段
 * @param eventType 对应的事件类型
 */
+ (void)updateCustomData:(BuglyCustomData *)data forEvent:
(BuglyEventTypeName)eventType;

/**
 * 获取特定事件的当前自定义数据
 * @param eventType 需要获取的事件类型
 * @return 对应已设置的自定义字段
*/
+ (nullable BuglyCustomData *)currentCustomDataForEvent:
(BuglyEventTypeName)eventType;

/**
 * 使用新的自定义字段更新此自定义数据
 * @param dict 符合 BuglyCustomData 存储格式的字典信息
 * @return 返回在次数据基础上更新后的新数据
 */
- (BuglyCustomData *)customDataByUpdateDict:(NSDictionary *)dict;

/**
  * 获取设置的数字类型的自定义数据
  * @param key 如果 KEY 不在 NumberParamKey 定义中，直接返回0.0
  * @return 如果有设置，则返回当前值，如果没有设置，返回0.0
  */
- (nullable NSNumber *)getNumberParam:(BuglyCustomNumberDataKey)key;

/**
 * 设置数字类型的自定义数据
 * @param key 如果 KEY 不在 NumberParamKey 定义中，则添加失败
 * @param param NUMBER VALUE
 */
```

```
- (BOOL)putNumberParam:(nullable NSNumber *)param forKey:
(BuglyCustomNumberDataKey)key;

/**
 * 获取设置的字符串类型的自定义数据
 * @param key 如果 KEY 不在 StringParamKey 定义中，直接返回""
 * @return 如果有设置，则返回当前值，如果没有设置，返回""
 */
- (nullable NSString *)getStringParam:(BuglyCustomStringDataKey)key;

/**
 * 设置字符串类型的自定义数据
 * @param key 如果 KEY 不在 StringParamKey 定义中，则添加失败
 * @param param 长度不能超过 BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH
 *      value = null, 则会设置空串
 *      value = 长度超标字符串，则会截取前
BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH 个字符设置
 */
- (BOOL)putStringParam:(nullable NSString *)param forKey:
(BuglyCustomStringDataKey)key;

/**
 * 获取字符串数组类型的自定义数据
 * @param key 如果 KEY 不在 StringArrayParamKey 定义中，则直接返回空列表
 * @return 如果有设置，则返回相关设置数据，如果没有设置，则直接返回空列表, 返回的
List 不可以修改。
 */
- (nullable NSArray<NSString *> *)getStringArrayParam:
(BuglyCustomArrayDataKey)key;

/**
 * 添加顺序是: A1 -> A2 -> A2 -> A3 -> A4, 这个接口产生的结果类似: [A1, A2,
A3, A4]
 * 为字符串数组类型的自定义数据，增加VALUE
 * @param key 如果 KEY 不在 StringArrayParamKey 定义中，则添加失败
 * @param param 非空值，非空串，如果 value 不存在，则添加，如果存在，则直接返回，
长度不能超过 BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH
 */
- (BOOL)addStringToStringArrayParam:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;

/**
 * 从字符串数组类型的自定义数据中，移除 VALUE
```

```
 * @param key 如果 KEY 不在 StringArrayParamKey 定义中，则移除失败
 * @param param 如果 value 存在，则移除，如果不存在，则直接返回
 */
- (BOOL)removeStringFromStringArrayParam:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;

/**
 * 添加顺序是：A1 -> A2 -> A2 -> A3 -> A4，这个接口产生的结果类似：[A1, A2,
A2, A3, A4]
 * 为字符串数组类型的自定义数据，增加VALUE
 * @param key 如果 KEY 不在 StringArrayParamKey 定义中，则添加失败
 * @param param 长度不能超过 BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH，无
论是否存在，都会添加，即可以存在重复的 value
 * @return value 是空值，空串，或者超长，或者已经达字符串数组最大值，返回失败，否则
还回成功
 */
- (BOOL)addStringToSequence:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;
```

启动监控

定义

冷启动耗时计算规则分为 iOS 15之前的版本和 iOS 15之后的版本两种。

- iOS 15以前的版本：**冷启动耗时 = 第一帧 UI 上屏时间 – App 进程创建时间**。
- iOS 15及以上版本：

  苹果在 iOS 15上增加了 Prewarming 的新特性，可以预启动进程，导致从“App 进程初始化结束时间”到“`main` 函数执行时间”这段时间变得非常不确定。因此冷启动耗时减去中间这段不确定的时间，调整为：**冷启动耗时 = 第一帧 UI 上屏时间 – main 函数执行时间 + App 进程初始化结束时间 – App 进程创建时间**。

API 介绍

自定义场景区间耗时统计

- 调用 `BuglyLaunchMonitorPlugin` 的

  `+ (void)startSpan:(NSString *)spanName parentSpanName:(nullable NSString *)parentSpanName`

  ，记录自定义场景开始时间戳。

```
[BuglyLaunchMonitorPlugin startSpan:@"testSpan" parentSpanName:nil];
```

- 调用 `BuglyLaunchMonitorPlugin` 的 `+ (void)endSpan:(NSString *)spanName`，记录自定义场景区间结束，请务必跟开启打点的spanName相同，否则记为无效数据。

```
[BuglyLaunchMonitorPlugin endSpan:@"testSpan"];
```

- 调用 `BuglyLaunchMonitorPlugin` 的 `+ (void)endSpanFromLaunch:(NSString *)spanName`，记录从进程创建开始统计的 span 耗时。

```
[BuglyLaunchMonitorPlugin endSpanFromLaunch:@"testSpan"];
```

设置启动 tag

- 调用 `BuglyLaunchMonitorPlugin` 的 `- (void)addTag:(NSString *)tagName`，添加一个 tag。
- 调用 `BuglyLaunchMonitorPlugin` 的 `- (void)removeTag:(NSString *)tagName`，删除一个 tag。

```
[BuglyLaunchMonitorPlugin addTag:@"tagTest1"];
[BuglyLaunchMonitorPlugin removeTag:@"tagTest1"];
```

自定义启动结束时间

因为业务特性，业务定义启动结束时间点并不仅是到 `CA::Transaction::commit` 结束。所以增加 `- (void)endColdLaunch` 接口用于业务主动调用声明冷启动已经结束，调用该接口会触发启动数据上报，后续更新tag或者span信息将不会生效。
如果接口未被调用，退后台、闪退、或者超过一定时间后（默认3分钟）也会触发启动数据上报。

```
[BuglyLaunchMonitorPlugin endColdLaunch];
```

鸿蒙 SDK 接入指引
SDK 接入简介

最近更新时间：2025-08-20 17:51:32

终端性能监控 Pro SDK 由腾讯 Bugly 团队提供技术支持。终端性能监控 Pro Harmony 版本，支持 Harmony OS Next 平台基础异常问题的捕获上报，包含 Js 异常、Cpp 异常、AppFreeze、错误登。SDK 基于 Harmony 平台 arkTs 开发，可通过 Har 包集成方式接入，提供 arkTs 接口。

SDK 简介

- **SDK 版本**：0.4.1
- **SDK 功能**：专业的应用质量监控工具，提供异常数据的采集和分析服务，帮助开发者及时发现并解决异常问题，打造高质量 App。
- **服务提供方**：腾讯云计算（北京）有限责任公司
- 终端性能监控 Pro SDK 合规使用指南
- 终端性能监控 Pro SDK 个人信息保护规则

接入步骤

1. 在接入 SDK 之前，请务必认真阅读 终端性能监控 Pro SDK 合规使用指南 。
2. 完成 SDK 集成 。
3. 完成 SDK 初始化 。SDK 在初始化过程中，可能会采集部分用户信息。请在用户同意 终端性能监控 Pro SDK 个人信息保护规则 之后，再进行 SDK 的初始化。在初始化 SDK 前，不会收集任何信息。
4. 验证 数据上报 。
5. 详细了解其他 SDK 功能，请参见 API 说明 。

SDK 集成

最近更新时间：2025-08-20 17:51:32

本文将为您介绍通过自动集成和手动集成两种方式集成 SDK。

前提条件

终端性能监控 Pro 需要使用 Bugly SDK 0.4.1及以上版本。

自动集成（推荐）

1. 配置内网鸿蒙三方库，执行以下命令。(设置默认存在该三方库，则无需配置)

```
ohpm config set registry https://ohpm.openharmony.cn/ohpm/
```

> **说明：**
> 设置默认原始只有鸿蒙官方三方库，如添加了其他三方库，需通过 `ohpm config list` 查看设置的三方库，手动将 `https://ohpm.openharmony.cn/ohpm/` 追加后重新设置。

2. 通过 `ohpm` 安装 Bugly 库。

```
ohpm install bugly@0.4.1
```

3. 安装完成后可直接在 arkTs 中通过 `import` 导入引用。

手动集成

1. 通过三方仓库或其他渠道下载 `Bugly.har` 三方 SDK 包。
2. 在需要集成的模块下创建 `libs` 目录，将 `Bugly.har` 放入目录。
3. 在模块的 `oh-package.json5` 文件中添加对应 `dependencies`，如下所示。

```
"dependencies": {
    "bugly": "file:./libs/Bugly.har"
}
```

配置混淆规则

业务二次混淆 Har 包代码，可能导致部分 Napi 符号或字符串变量不可见，引入运行问题，建议业务直接 keep Bugly Har 包。在 hap 工程目录下 `obfuscation-rules.txt` 混淆规则配置文件中直接添加如下规则即可：

```
-keep
../oh_modules/bugly
```

SDK 初始化

最近更新时间：2025-08-20 17:51:32

本文将为您介绍如何初始化终端性能监控 Pro SDK。

示例代码

参考以下代码初始化 SDK，我们推荐尽早可能初始化SDK，如将初始化逻辑放在 `Ability` 的 `onCreate` 生命周期中。可参考如下代码进行初始化。

```
import { Bugly, BuglyBuilder, AppVersionMode, ModuleName } from "bugly";

initBugly(context: Context): void {
    let builder = new BuglyBuilder();

    builder.appId = 'xxxxxxx';    // 必填，终端性能监控 Pro 应用列表中的 APP ID
    builder.appKey = 'xxx-xxxx-xxxx-xxxx-xxxx';     // 必填，终端性能监控 Pro 应用列表中的 APP KEY
    builder.deviceId = "12345";      // 必填，设备 ID，应保证设备 ID 对不同设备唯一
    builder.platform = BuglyBuilder.PLATFORM_PRO;     // 必填，设置上报平台，专业版本需设置为[BuglyBuilder.PLATFORM_PRO]

    builder.appVersion = '1.0.0';    // 选填，业务的 App 版本
    builder.appVersionMode = AppVersionMode.DEBUG;  // 选填，当前 App 的版本类型，支持根据不同的版本类型下发配置
    builder.buildNum = '0';          // 选填，业务 App 版本的构建号
    builder.appChannel = 'website'; // 选填，业务 App 渠道
    builder.userId = "12345";        // 选填，用户 ID，如不设置则为空
    builder.deviceModel = "huawei"; // 选填，机型，如不设置则为空
    builder.debugMode = true;        // 选填，默认开启，开启后 SDK 会打印更多调试日志，线上版本可关闭
    builder.sdkLogMode = true;       // 选填，设置 debugMode 或 sdkLogMode 均可开启 Bugly sdk 日志，适用于线上关闭 debug 模式但又希望打印 bugly 日志的场景
    builder.initDelay = 0;           // 选填，延迟初始化时间，单位ms
    builder.enableJsCrashProtect = false;    // 选填，是否开启 Js 异常崩溃保护，设置为 true 后发生未捕获 Js 异常，进程不会退出
    builder.enablePerfModules = ModuleName.AllModules;  // 选填，开启性能监控项，可传入单个性能模块名称或一组性能模块名称，此处初始化后，还需配置开启对应模块采样率，模块才会真正开启
```

```
    Bugly.init(context, builder);    // 如果需等待 Bugly 完全初始化完成，使用
await Bugly.init(context, builder);
}
```

⚠ 注意：

- Context 需要传递 ApplicationContext。
- 设备 ID 非常重要，终端性能监控 Pro 使用设备 ID 来计算设备异常率，强烈建议应用设置正确的设备 ID，以确保设备的唯一性。
- BuglyBuilder 需在 init 方法前创建，且应避免重复调用 init 方法。
- 需要调用 Bugly.init 接口进行初始化，完成初始化后，再调用其他接口进行定制化设置，否则设置不生效。
- 在同一进程中，应只初始化一次 SDK。仅建议在主线程中初始化，`0.4.1` 及之后的 SDK 版本仅支持在主线程中初始化。

数据上报验证

最近更新时间：2025-08-20 17:51:32

崩溃监控

初始化完成后，可以模拟崩溃进行上报，如执行以下调用。

```
Bugly.testCrash(Bugly.JS_CRASH); // 模拟Js异常

Bugly.testCrash(Bugly.CPP_CRASH); // 模拟native异常
```

说明：

- 异常问题发生后，需要二次启动 Hap 应用，即可完成上报。
- Crash 异常会上报 FaultLog 信息，可在 附件 tab 的 `crashInfos.txt` 文件中查看。

异常上报后，可在 崩溃 > 问题列表 中查看上报问题，单击即可进入**问题详情**，查看上报内容。

Freeze 监控

与崩溃类似，可以通过终端性能监控 Pro 提供的测试接口来模拟 Freeze，调用后适当滑动屏幕，系统将判定为当前进程 Freeze。

```
Bugly.testCrash(Bugly.APP_FREEZE); // 模拟Freeze异常
```

或是在 UI 线程中执行耗时逻辑，触发 ANR。

```
let index = 0
while (true) {
    index++
    index = index % 2
}
```

Freeze 问题触发上报后，在 Freeze > 问题列表 中查看，单击即可进入**问题详情**，查看上报内容。

说明：

- Freeze 异常上报的是 Native、arkTs 混合堆栈。
- Freeze 异常会上报 FaultLog 信息及主线程消息信息，可在 **附件** tab 的 `crashInfos.txt` 文件中查看。

- `event_handler` 表示主线程未处理消息。
- `event_handler_size_3s` 表示 THREAD_BLOCK 事件3s时任务栈中任务数。
- `event_handler_size_6s` 表示 THREAD_BLOCK 事件6s时任务栈中任务数。
- `peer_binder` 表示 binder 调用信息。

卡顿监控

卡顿指标（FPS、挂起率）

注意：

- 确保卡顿指标模块开启，需在 buglybuilder 初始化参数 enablePerfModules 中添加卡顿指标模块（looper_metric），同时在配置中设置卡顿指标的 sample_ratio 设备采样率，二者缺一不可。
- 建议测试时，配置设备采样率设为1，设置方式请参见 SDK 配置 。

指标说明

- **FPS：** 帧率，在应用运行时，GPU 和 CPU 合作可产生的图像数量，反映了 UI 图像刷新的频率，计量单位帧/秒（FramePerSecond，FPS），通常是评估硬件性能与应用流畅度的指标。
- 鸿蒙 FPS 的计算借鉴了 Android、iOS 实现方式，采用归一化的方式进行统计，以60HZ为基准，突出反映应用在影响用户体验（低于60HZ）运行时的状态，即应用越流畅，归一化 FPS 越接近60HZ，反之越低。
- **挂起率：** 如果应用在主线程中的单条消息执行时间过长，则可能导致 UI 卡顿，此时认为应用不能很好地响应用户的交互，累计执行的时间到挂起时间中。一个设备的挂起率，指这个设备在一天中，总的挂起时间除以其前台总时长，单位是秒/小时。

验证上报

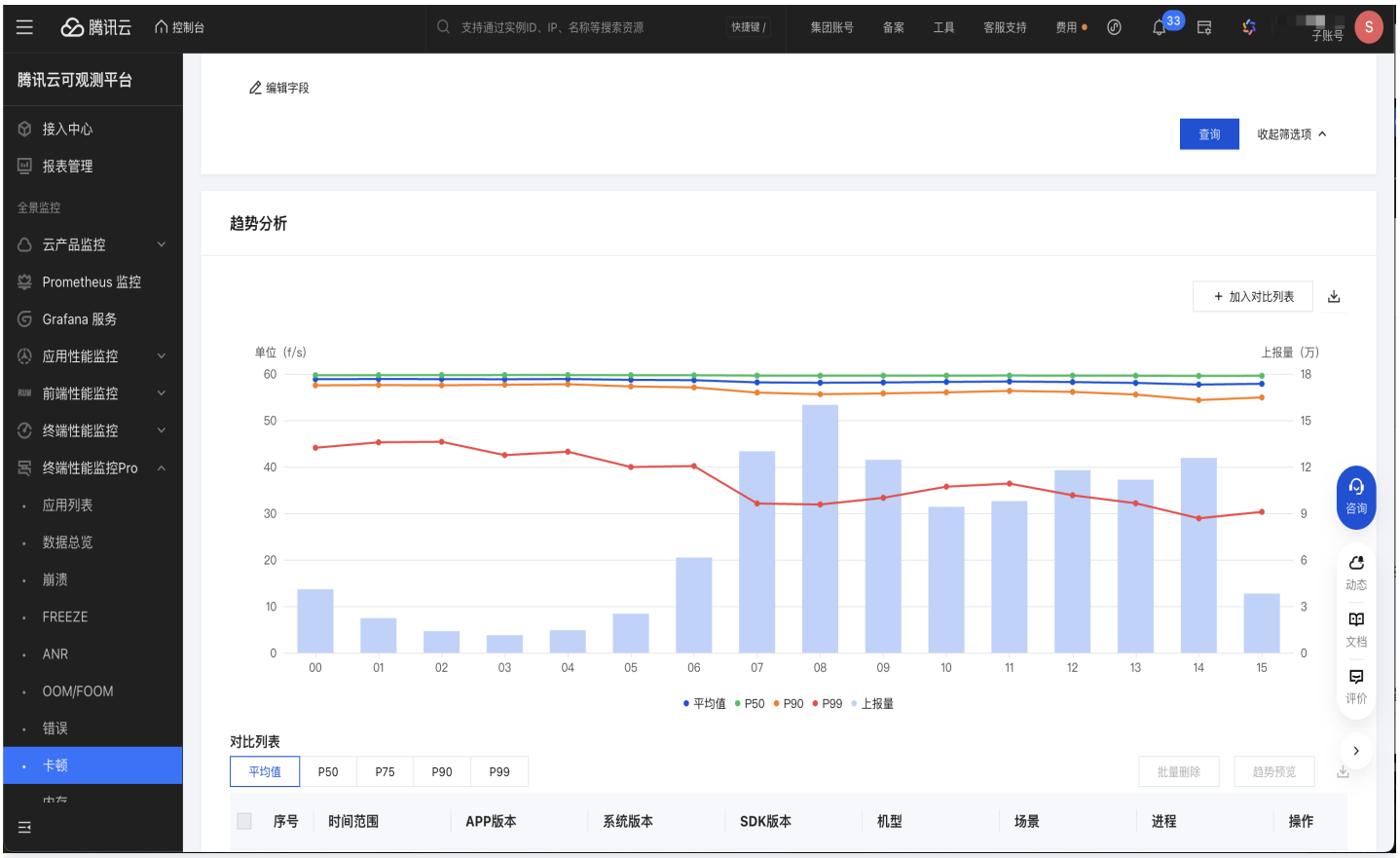
初始化卡顿指标能力并命中采样后，在 `debug` 模式下，可以看到如下日志，以判断卡顿指标已被开启。

```
03-27 20:05:31.683   14562-14562   A00000/com.ten...looper_metric
com.tencent.bugly      I      Jank metric module start!
```

在有多个页面切换，并切换前后台时，可以看到卡顿指标上报日志。

```
04-02 15:12:34.938   61473-61473   A00000/com.tencent.bugly/Bugly
com.tencent.bugly      I      [Upload] upload Success, upload event:
looper_metric, record id: a84630f6-ffb4-4d9b-8033-8c288507d58f.
```

卡顿指标上报后，可在 指标分析 页面中查看对应数据统计，并支持 App 版本、页面场景等维度下钻分析。



> **说明：**
>
> - 卡顿指标以页面场景为统计维度，在应用有页面切换时记录指标数据，退至后台时上报数据。
> - 应尽早初始化 SDK，才能及时开启卡顿指标的统计。如果 SDK 初始化时间较晚，将在用户下一次切前后台操作后开启采集。

卡顿个例

> **注意：**
>
> - 确保卡顿个例模块开启，需在 `buglybuilder` 初始化参数 `enablePerfModules` 中添加卡顿指标模块（ `looper_stack` ），同时在配置中设置卡顿指标的 `sample_ratio` 设备采样率，二者缺一不可。
> - 建议测试时，配置设备采样率设为1，设置方式请参见 SDK 配置 。

个例说明

卡顿堆栈监控能力基于系统 API 5.0.0(12) HiAppEvent 主线程超时事件开发，兼容 API 12及以上版本，**默认超时时间150ms，抓栈次数固定10次，在进程生命周期中，仅会触发一次**。详情请参见 鸿蒙主线程超时事件 。

验证上报

初始化卡顿个例监控并命中采样后，在 `debug` 模式下，可以看到如下日志，以判断卡顿个例监控已被开启。

```
03-27 20:05:31.683   14562-14562   A00000/com.ten...-looper_stack
com.tencent.bugly      I      Jank monitor module start!
```

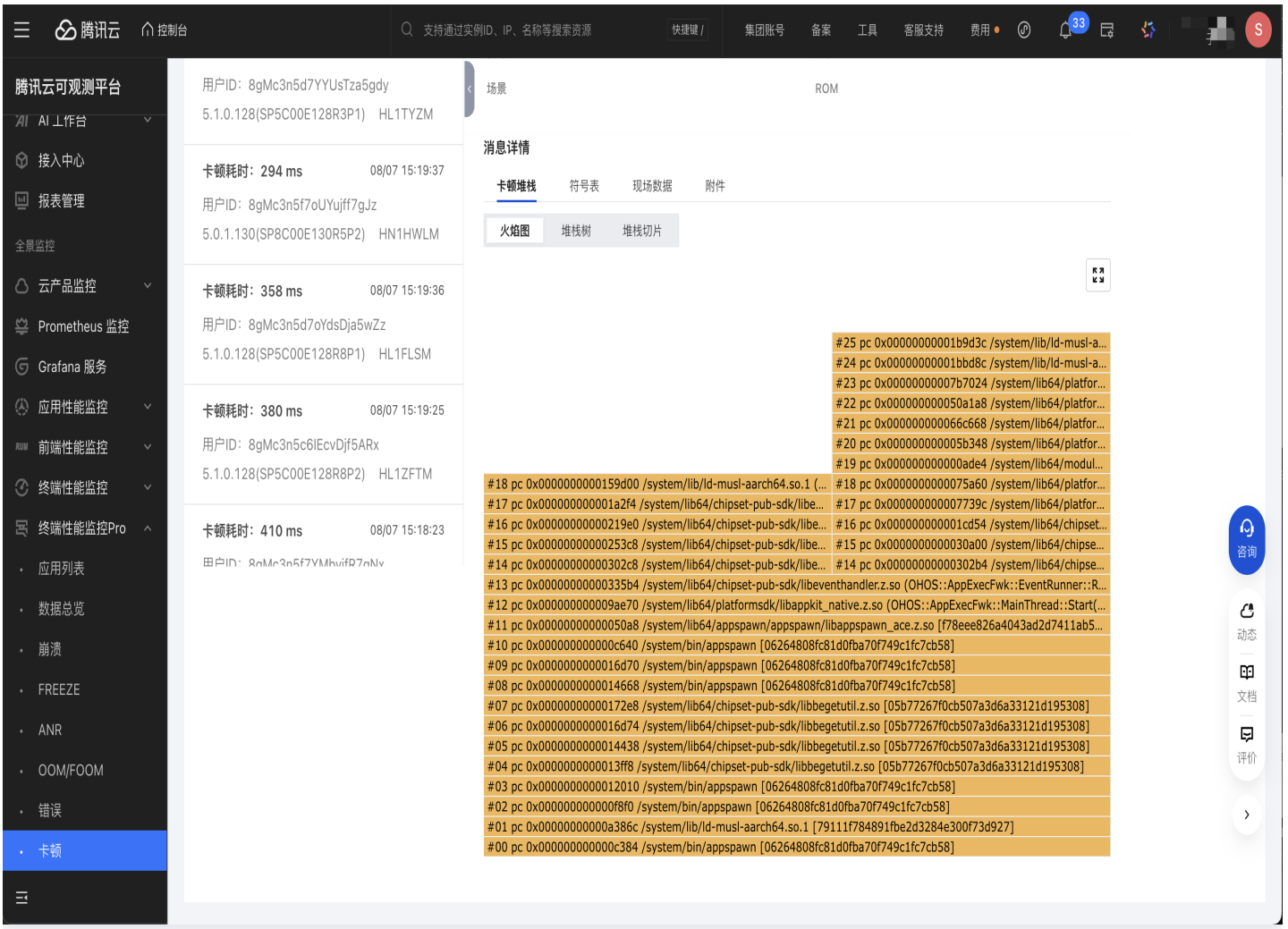
可通过在点击事件中连续触发以下代码，来构造触发卡顿事件。

```
let t = Date.now();
while (Date.now() - t <= 350) {}
```

成功触发后，可看到如下上报日志，表明卡顿个例已成功上报。

```
04-02 16:06:24.405   61473-61473   A00000/com.ten...-looper_stack
com.tencent.bugly      I      [Jank] jank event received!!! domain: OS
04-02 16:06:24.406   61473-61473   A00000/com.ten...-looper_stack
com.tencent.bugly      I      [Jank] event group name: MAIN_THREAD_JANK
04-02 16:06:24.406   61473-61473   A00000/com.ten...-looper_stack
com.tencent.bugly      I      [Jank] jank begin time: 1743581180178, end
time: 1743581180530, log paths:
["/data/storage/el2/log/watchdog/MAIN_THREAD_JANK_20250402160622_61473.t
xt"]
04-02 16:06:26.170   61473-61473   A00000/com.tencent.bugly/Bugly
com.tencent.bugly      I      bugly do upload, type: 3, event:
looper_stack
04-02 16:06:26.453   61473-61473   A00000/com.tencent.bugly/Bugly
com.tencent.bugly      I      [Upload] upload Success, upload event:
looper_stack, record id: cd6623ba-e5e5-474d-b70b-bf087120aa44.
```

卡顿个例上报后，可在 卡顿问题详情 页面中查看抓取全部堆栈聚合而成的火焰图、堆栈树数据，堆栈支持上传符号表翻译。


腾讯云
控制台
支持通过实例ID、IP、名称等搜索资源
快捷键 /
集团账号
备案
工具
客服支持
费用
33
S
腾讯云可观测平台
AI 工作台
接入中心
报表管理
全景监控
云产品监控
Prometheus 监控
Grafana 服务
应用性能监控
前端性能监控
终端性能监控
终端性能监控Pro
应用列表
数据总览
崩溃
FREEZE
ANR
OOM/FOOM
错误
卡顿
用户ID：8gMc3n5d7YYUsTza5gdy
5.1.0.128(SP5C00E128R3P1) HL1TYZM
卡顿耗时：294 ms
08/07 15:19:37
用户ID：8gMc3n5f7oUYujff7gJz
5.0.1.130(SP8C00E130R5P2) HN1HWLM
卡顿耗时：358 ms
08/07 15:19:36
用户ID：8gMc3n5d7oYdsDja5wZz
5.1.0.128(SP5C00E128R8P1) HL1FLSM
卡顿耗时：380 ms
08/07 15:19:25
用户ID：8gMc3n5c6lEcvDjf5ARx
5.1.0.128(SP5C00E128R8P2) HL1ZFTM
卡顿耗时：410 ms
08/07 15:18:23
场景
ROM
消息详情
卡顿堆栈
符号表
现场数据
附件
火焰图
堆栈树
堆栈切片
#25 pc 0x00000000001b9d3c /system/lib/ld-musl-a...
#24 pc 0x00000000001bbd8c /system/lib/ld-musl-a...
#23 pc 0x00000000007b7024 /system/lib64/platfor...
#22 pc 0x000000000050a1a8 /system/lib64/platfor...
#21 pc 0x000000000066c668 /system/lib64/platfor...
#20 pc 0x0000000000005b348 /system/lib64/platfor...
#19 pc 0x0000000000000ade4 /system/lib64/modul...
#18 pc 0x0000000000159d00 /system/lib/ld-musl-aarch64.so.1 (...
#18 pc 0x0000000000075a60 /system/lib64/platfor...
#17 pc 0x000000000001a2f4 /system/lib64/chipset-pub-sdk/libe...
#17 pc 0x000000000007739c /system/lib64/platfor...
#16 pc 0x00000000000219e0 /system/lib64/chipset-pub-sdk/libe...
#16 pc 0x000000000001cd54 /system/lib64/chipset...
#15 pc 0x0000000000253c8 /system/lib64/chipset-pub-sdk/libe...
#15 pc 0x0000000000030a00 /system/lib64/chipse...
#14 pc 0x00000000000302c8 /system/lib64/chipset-pub-sdk/libe...
#14 pc 0x00000000000302b4 /system/lib64/chipse...
#13 pc 0x00000000000335b4 /system/lib64/chipset-pub-sdk/libeventhandler.z.so (OHOS::AppExecFwk::EventRunner::R...
#12 pc 0x000000000009ae70 /system/lib64/platformsdk/libappkit_native.z.so (OHOS::AppExecFwk::MainThread::Start(...
#11 pc 0x00000000000050a8 /system/lib64/appspawn/appspawn/libappspawn_ace.z.so [f78eee826a4043ad2d7411ab5...
#10 pc 0x000000000000c640 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
#09 pc 0x0000000000016d70 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
#08 pc 0x0000000000014668 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
#07 pc 0x00000000000172e8 /system/lib64/chipset-pub-sdk/libbegetutil.z.so [05b77267f0cb507a3d6a33121d195308]
#06 pc 0x0000000000016d74 /system/lib64/chipset-pub-sdk/libbegetutil.z.so [05b77267f0cb507a3d6a33121d195308]
#05 pc 0x0000000000014438 /system/lib64/chipset-pub-sdk/libbegetutil.z.so [05b77267f0cb507a3d6a33121d195308]
#04 pc 0x0000000000013ff8 /system/lib64/chipset-pub-sdk/libbegetutil.z.so [05b77267f0cb507a3d6a33121d195308]
#03 pc 0x0000000000012010 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
#02 pc 0x000000000000f8f0 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
#01 pc 0x00000000000a386c /system/lib/ld-musl-aarch64.so.1 [79111f784891fbe2d3284e300f73d927]
#00 pc 0x000000000000c384 /system/bin/appspawn [06264808fc81d0fba70f749c1fc7cb58]
咨询
动态
文档
评价

API 说明

最近更新时间：2025-08-20 17:51:32

本文详细介绍终端性能监控 Pro SDK 的各功能接口，帮助您更加灵活、深度的使用 SDK。

信息更新接口

SDK 初始化后，如需更新必要字段，接口如下。

```
/**
 * 更新device id
 * @param deviceId device id
 */
public static updateDeviceId(deviceId: string);

/**
 * 更新user id
 * @param userId user id
 */
public static async updateUserId(userId: string);

/**
 * 更新device model
 * @param deviceModel
 */
public static updateDeviceModel(deviceModel: string);
```

错误上报接口

SDK 支持通过如下两个接口上报 catch Error 或自定义错误。

```
/**
 * 上报catch Error
 * @param e Error
 * @param extraData [可选] 随错误上报的附加信息，以userExtraByteData附件展示
 */
public static postError(e: Error, extraData?: string): void;


/**
 * 上报自定义错误
```

```
 * @param name 异常名
 * @param message 异常msg
 * @param stack 异常堆栈
 * @param storeFirst [可选] 是否优先存储db不立即上报，默认为false，仅在异常现场时采集的错误可设置为true
 * @param context [可选] 如果未初始化Bugly调用接口，传入context可记录错误
 * @param extraData [可选] 随错误上报的附加信息，以userExtraByteData附件展示
 */
public static postCustomError(name: string, message: string, stack: string, storeFirst: boolean = false, context?: Context, extraData?: string): void;
```

如即时上报一条自定义错误，且添加附加信息，示例如下：

```
Bugly.postCustomError("testErrorName", "testErrorMsg", "at testErrorStack 1\nat testErrorStack 2", false, undefined, "This is extra data file content");
```

说明：

- `postError` 接口适用于 arkTs 中 catch 异常的捕获，直接传入 catch 错误的 Error 实例，会自动从 Error 实例中解析出对应的异常名称及堆栈信息。
- `postCustomError` 接口适用于上报自定义异常，如采用跨端框架、游戏框架的异常。如果在正常环境下，`storeFirst` 参数无需设置，如果在进程可能被系统杀死环境下，建议设置 `storeFirst` 参数为 true，当前只以同步方式保存异常到本地文件，二次启动进程时上报。
- `postCustomError` 在 `0.4.1` 及之后的 SDK 版本支持跨线程上报错误，但需在主线程初始化 Bugly，或设置 `storeFirst` 参数设置为 true。
- 上述错误上报接口在 `0.4.1` 及之后的 SDK 版本中支持携带附加信息，如有设置 `extraData` 附加信息参数，错误上报后，将在错误详情中以 `userExtraByteData` 附件的形式提供下载。
- 错误上报在 `0.4.1` 及之后的 SDK 版本中支持配置控制采样率，设置方式请参见 SDK 配置 。

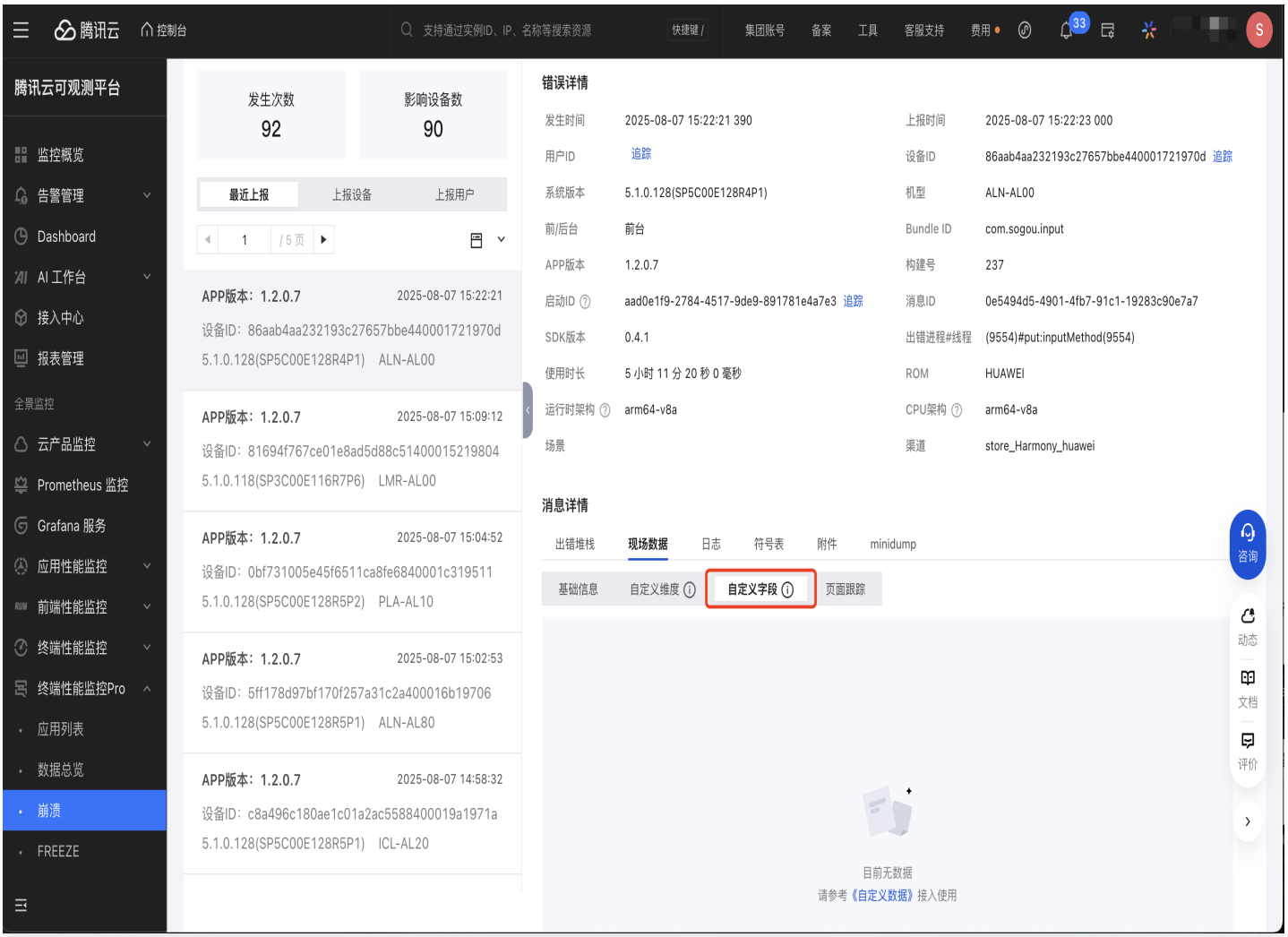
自定义数据接口

SDK `0.2.0` 开始支持设置自定义数据，自定义数据需在 Crash 、Freeze 或错误发生前进行设置，可以进行更新和移除操作，异常上报时会携带设置的自定义数据。

```
/**
 * 添加或更新自定义数据
 * @param key 自定义key
 * @param value 自定义value
```

```
 */
public static putUserData(key: string, value: string);

/**
 * 移除自定义数据
 * @param key 自定义key
 */
public static removeUserData(key: string);

/**
 * 清空自定义数据
 */
public static clearUserData();
```

上报的自定义数据将在问题详情 > 现场数据 > 自定义字段中进行展示，如下图所示。



自定义附件接口

SDK `0.2.0` 开始支持设置业务自定义附件，自定义附件上报当前只支持 **Crash** 和 **Freeze**。在发生异常前，通过如下接口设置自定义附件的路径。

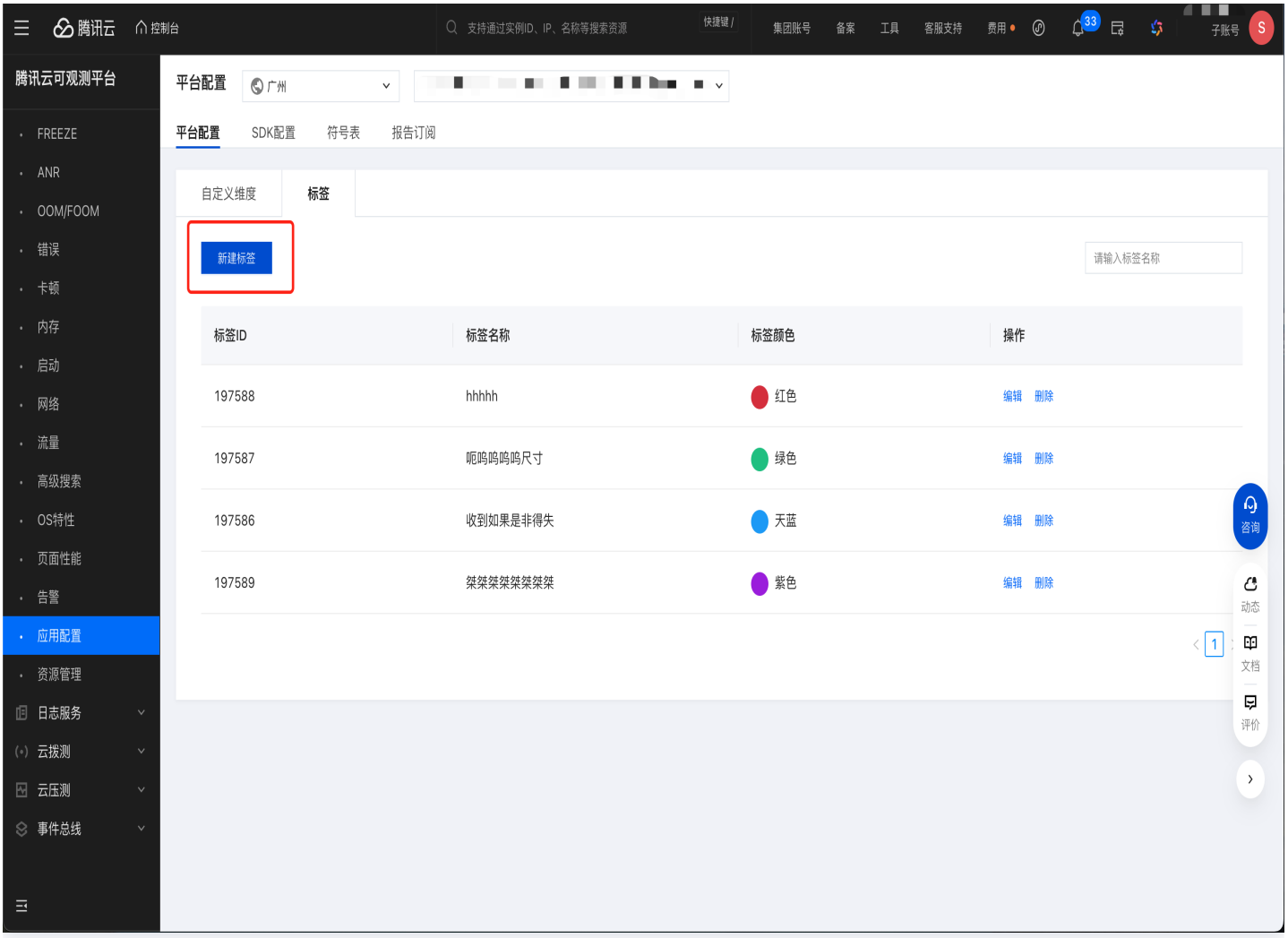
```
/**
 * 设置自定义文件路径
 * 传入参数为string数组，受限于AppEvent参数限制，数组值长度需在1024个字符以内
 * @param paths 自定义文件路径
 */
public static async setCustomFilePaths(paths: Array<string>);
```

发生异常重启进程后，自定义附件会进行打包上传，在附件中展示为 `custom_log.gzip` 。

> ⚠ **注意：**
>
> - 自定义数据、自定义附件、自定义标签及现场信息的关联均会使用到系统 `hiAppEvent.setEventParam` 接口，该接口设置参数值长度需在1024个字符以内，且至多设置64个键值对。因此**自定义附件 Array 的所有值长度和需在1024个字符以内**，否则可能设置失败。
> - 如果业务有自行设置系统 `hiAppEvent.setEventParam` 接口，请务必预留一定数量的键值对供 Bugly 设置（5个以上），否则可能影响自定义数据、附件或现场信息关联。

自定义标签接口

SDK `0.4.1` 开始支持业务自定义个例标签设置，标签随异常个例上报。个例标签需先在控制台的 应用配置 > 平台配置 > 标签 中进行新建，如下所示。



接着通过如下 SDK 接口，在代码中设置自定义标签 ID，异常个例上报时会同时上报这些标签 ID。

```
/**
    * 设置个例标签
    * @param caseLabel 个例标签数组
    */
public static async setCaseLabel(caseLabel: Array<string>);
```

如设置如下自定义标签。

```
await Bugly.setCaseLabel(["22096", "22103", "22244"]);
```

日志导出接口

SDK `0.2.0` 开始支持导出 Bugly 日志打印到业务日志系统中（不设置则默认打印到系统 HiLog 日志系统中），**在初始化前调用如下接口进行设置。**

```
/**
 * 设置Bugly日志适配器
 * @param adapter 适配器
 */
public static setLogAdapter(adapter: BuglyLogAdapter);

/**
 * Bugly日志适配接口
 */
export interface BuglyLogAdapter {
    // debug级别日志
    debug(tag: string, arg: string): void;
    // info级别日志
    info(tag: string, arg: string): void;
    // warn级别日志
    warn(tag: string, arg: string): void;
    // error级别日志
    error(tag: string, arg: string): void;
    // fatal级别日志
    fatal(tag: string, arg: string): void;
}
```

> ⚠ **注意：**
> SDK `0.3.7` 及之后的版本新增了日志导出接口 `tag` 参数，如有导出 Bugly 日志，请进行适配调整。

FaultLog 附件管理接口

Bugly 注册系统异常回调后，会自动管理系统异常 `FaultLog` 文件，默认上报完后会进行删除，否则缓存区满会影响后续 `FaultLog` 文件生成。如不希望 Bugly 对 `FaultLog` 文件进行删除，而是自行管理，可通过以下接口进行设置。

```
/**
 * 设置是否在bugly上报完fault log附件后删除
 * @param shouldDelete 是否需要bugly删除
 */
public static setDeleteFaultLogFileAfterUpload(shouldDelete: boolean);
```

异常回调接口

SDK `0.3.3` 开始支持 Js Crash、Cpp Crash 的异常回调，方便在发生异常时，能回调业务进行一些自定义操作。

SDK `0.3.5` 提供 HiAppEvent 收到异常信息时的回调，简化业务注册流程，设置接口如下。

```
/**
 * 异常回调接口
 */
export interface ICrashListener {
  /**
   * 异常发生时回调
   * @param crashType 异常类型，当前仅支持 JsCrash、CppCrash
   * @param crashName 异常名称，具体异常名称
   * @param crashMsg 异常信息，CppCrash不支持
   * @param crashStack 异常堆栈，CppCrash不支持
   */
  onCrash(crashType: string, crashName: string, crashMsg: string,
crashStack: string): void;
  /**
   * HiAppEvent收到异常信息时回调，支持Crash和Freeze
   * @param crashType 异常类型
   * @param crashName 异常名称
   * @param crashMsg 异常信息
   * @param crashStack 异常堆栈
   */
  onHiAppEventReceive(crashType: string, crashName: string, crashMsg:
string, crashStack: string): void;
}
```

设置异常回调接口示例如下。

```
// 在 Bugly 初始化前定义接口实现
class DemoCrashListener implements ICrashListener {
  onCrash(crashType: string, crashName: string, crashMsg: string,
crashStack: string): void {
    console.info('receive callback in demo.');
    console.info(`Crash Type: ${crashType}`);
    console.info(`Crash Name: ${crashName}`);
    console.info(`Crash Message: ${crashMsg}`);
    console.info(`Crash Stack: ${crashStack}`);
  }
  onHiAppEventReceive(crashType: string, crashName: string, crashMsg:
string, crashStack: string): void {
    console.info('[demo] receive hiAppEvent crash in demo.');
    console.info(`[demo] Crash Type: ${crashType}`);
```

```
    console.info(`[demo] Crash Name: ${crashName}`);
    console.info(`[demo] Crash Message: ${crashMsg}`);
    console.info(`Crash Stack: ${crashStack}`);
  }
}

// Bugly 初始化逻辑
let builder = new BuglyBuilder();
...
// 初始化时将接口示例传递给 builder.crashListener 参数
builder.crashListener = new DemoCrashListener();
...
await Bugly.init(context, builder);
```

> **注意：**
>
> - 暂不支持 App Freeze 异常回调。
> - Cpp Crash 回调暂不支持提供堆栈信息。
> - 请勿在回调中进行复杂操作，异常现场回调暂时无法保障稳定性，可能引入未知风险，请谨慎开启。

动态开关

SDK 支持质量和性能异常监控模块动态开关，可通过以下接口在业务需要的场景中动态开启或关闭异常监控上报。

```
/**
 * 质量异常监听动态开关
 * @param isFreeze true为Freeze监控，false为Crash监控(包括Js Crash和Cpp
Crash)
 * @param isAble 打开或关闭
 */
public static setCrashMonitorAble(isFreeze: boolean, isAble: boolean):
void;

/**
 * 性能模块动态开关
 * @param modules 性能模块项或列表
 * @param isAble 打开或关闭
 */
public static setPerfMonitorsAble(modules: string | Array<string>,
isAble: boolean): void;
```