Android JSONObject运行时异常修复方案

目录

1. 背景

`v13.58` 灰度版本中收集到 `JOSNObject` 堆栈崩溃，排查定位到 `v13.58` 随版需求的优先级队列功能会暴露历史代码设计缺陷：[BAIDUSEARCH-BUG-420709] 【新增】【Sprint 13.58.0】【Android端】Top3.0 崩溃类型=java Crash 版本=13.58.0.1 崩溃率=13.64% 崩溃次数=3 影响用户数=1

2. 问题排查

2.1 崩溃原因

从堆栈信息来看，对JSONObject的同步修改抛出了未能处理的异常

UBC SDK提供的接口方法中支持业务方传入 `JSONObject` 类型的参数；SDK内部直接使用了同一份 `JSONObject` 实例，在日志入库、上报等场景会调用 `JSONObject#toString()` 方法获取字符串对象，对 `JSONObject` 底层容器 `LinkedHashMap` 做遍历操作，若同一时刻 `JSONObject` 实例做了增减操作，会导致遍历时校验不通过，抛出运行时异常 `java.util.ConcurrentModificationException`

> 目前在v12+ 版本没有看到相关崩溃聚类数据，推测是部分逻辑被上层捕获，具体的代码调试后续跟进排查

从线程异常处理机制来看，实验组线程池不具备异常捕获能力

Java

`Thread` 中包含一个名为 *`uncaughtExceptionHandler`* 的成员变量和 *`defaultUncaughtExceptionHandler`* 的静态变量；

线程任务抛出异常未被 `try-catch` 时，线程被杀， *`uncaughtExceptionHandler.uncaughtException()`* 方法被回调；

线程池中线程任务抛出异常未被 `try-catch` 会导致该线程被杀，之后会创建新线程处理后续任务；

特别的：Java的线程池模型中有 **`ScheduledThreadPoolExecutor`** 的线程池实现，比如 `Executors.newSingleThreadScheduledExecutor()` 。该线程池在调用 `execute` 方法时会将 `Runnable` 对象封装成 `ScheduledFutureTask` (父类 `FutureTask`)。任务执行时 `FutureTask.run` 会对 `Runnable` 做异常捕获，**不再抛出异常和打印堆栈**。

Android

`Android` 平台， `Android` 初始化时设置了 `Thread` 全局的 `uncaughtExceptionHandler` ，当线程任务抛出异常未被 `try-catch` 在 *`uncaughtExceptionHandler. uncaughtException()`* 中**会主动杀死当前进程并退出，见下图**；

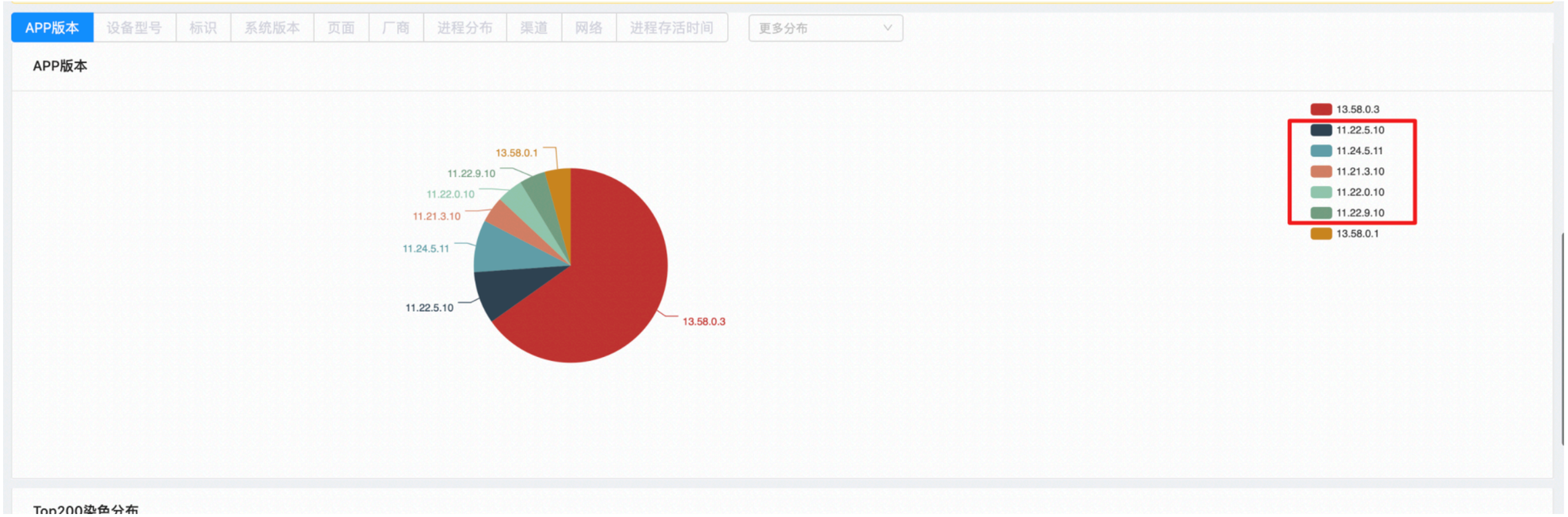
同样的，当使用 `ScheduledThreadPoolExecutor` 时，异常在 `FutureTask` 捕获，也不会打印异常堆栈。此时线程也没有死掉。

RuntimeInit.java

```
224
225     @UnsupportedAppUsage
226     protected static final void commonInit() {
227         if (DEBUG) Slog.d(TAG, "Entered RuntimeInit!");
228
229         /*
230          * set handlers; these apply to all threads in the VM. Apps can replace
231          * the default handler, but not the pre handler.
232          */
233         LoggingHandler loggingHandler = new LoggingHandler();
234         RuntimeHooks.setUncaughtExceptionPreHandler(loggingHandler);
235         Thread.setDefaultUncaughtExceptionHandler(new KillApplicationHandler(loggingHandler));
236
237         /*
238          * Install a time zone supplier that uses the Android persistent time zone system property.
239          */
240         RuntimeHooks.setTimeZoneIdSupplier(() -> SystemProperties.get("persist.sys.timezone"));
241
242         /*
243          * Sets handler for java.util.logging to use Android log facilities.
244          * The odd "new instance-and-then-throw-away" is a mirror of how
245          * the "java.util.logging.config.class" system property works. We
246          * can't use the system property here since the logger has almost
```

```
117
        Handle application death from an uncaught exception. The framework catches these for the main
        threads, so this should only matter for threads created by applications. Before this method runs, the
        given instance of RuntimeInit.LoggingHandler should already have logged details (and if not it is
        run first).
125     private static class KillApplicationHandler implements Thread.UncaughtExceptionHandler {
126         private final LoggingHandler mLoggingHandler;
127
            Create a new KillApplicationHandler that follows the given LoggingHandler. If
            uncaughtException is called on the created instance without loggingHandler having been
            triggered, loggingHandler.uncaughtException will be called first.
            Params: loggingHandler – the RuntimeInit.LoggingHandler expected to have run before
                    this instance's uncaughtException is being called.
139         public KillApplicationHandler(LoggingHandler loggingHandler) {
140             this.mLoggingHandler = Objects.requireNonNull(loggingHandler);
141         }
142
143         @Override
144         public void uncaughtException(Thread t, Throwable e) {
145             try {...} catch (Throwable t2) {...} finally {
173                 // Try everything to make sure this process goes away.
174                 Process.killProcess(Process.myPid());
175                 System.exit( status: 10);
176             }
177         }
178
```

2.2 崩溃数据版本分布



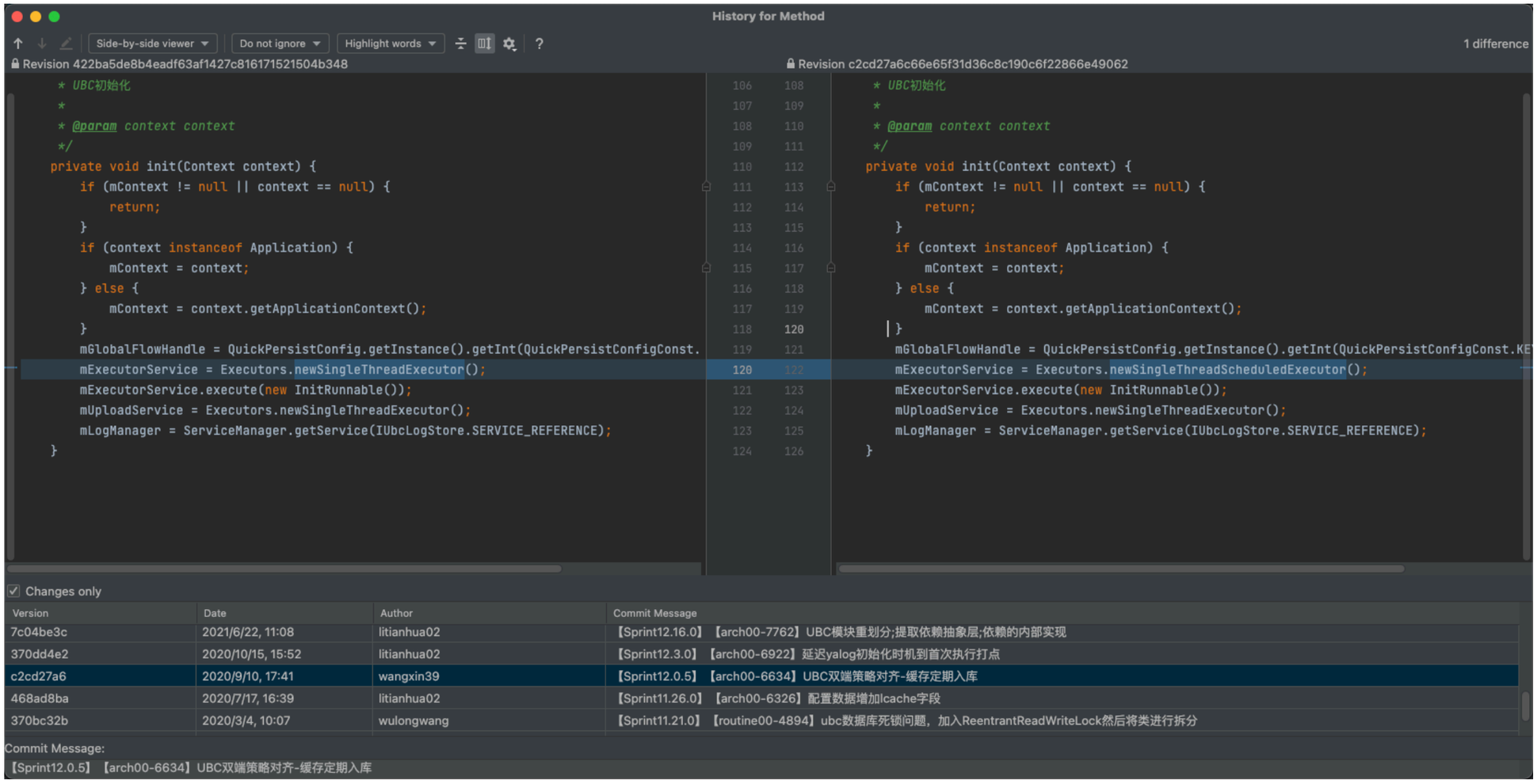
崩溃集中在 `v13.58.0` 以及 `v11.xx` 版本，但是 `v12.xx` 版本没有崩溃数据

线下调试

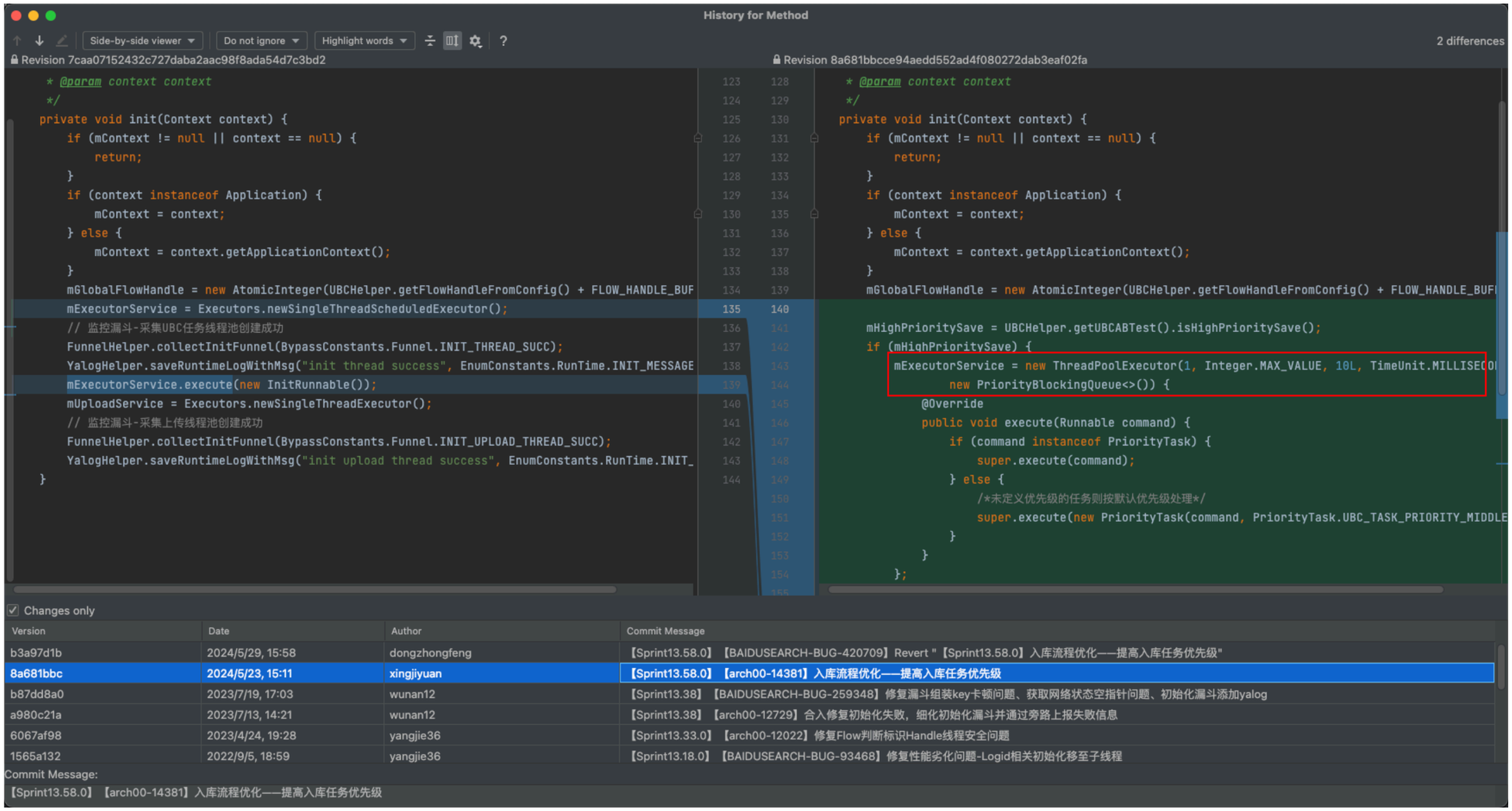
- 实验组：抛出运行时异常，线程被杀，进程崩溃，异常堆栈打印
- 对照组：抛出运行时异常，线程存活，进程不崩溃，没有异常堆栈（手动try-catch可以捕获到异常并打印堆栈）

从 `UBC` 线程池实现的记录中可以找到答案：

- 从 `v12.0.5` 版本开始， `UBC` 线程池的实现从 `Executors.newSingleThreadExecutor()` 更改为 `Executors.newSingleThreadScheduledExecutor()`
  - 前者对于未捕获的异常会崩溃，后者不会



- 从 `v13.58.0` 版本开始，新增的优先级队列方案，实验组使用 `newThreadPoolExecutor()`
  - 实验组对于未捕获的异常会崩溃；对照组不会



相关线程池实现及异常处理机制整理

1			线程任务抛出未处理的异常	异常堆栈打印
2	Android	`Executors.newSingleThreadExecutor()`	线程被杀，新建线程处理后续任务；进程崩溃，应用退出	打印堆栈
3		`Executors.newSingleThreadScheduledExecutor()`	线程存活；进程不崩溃	不打印堆栈

3. 解决方案

方案1：`SDK`使用备份的`JSONObject`评审意见：不建议

适用于`JSONObject`的拷贝方式有两种：

- **手动遍历：**`JSONObject`

- 【缺点】因为 `JSONObject` 没有实现 `Cloneable`，不能直接调用 `clone` 方法拷贝；同时对于 `value` 也并不适合做深拷贝

```java
JSONObject copy = new JSONObject();
Iterator<String> keys = json.keys();
while (keys.hasNext()){
    String next = keys.next();
    Object obj = json.get(next);
    copy.put(next, obj);
}
```

> 一般的，Java深拷贝是针对自定义Class做深拷贝；JSONObject本身没有提供深拷贝相关的实现方式

- **JSON序列化：** `JSONObject` 转字符串

```java
public final void onEvent(String eventId, JSONObject value) {
        JSONObject jsonObject;
        try {
            jsonObject = new JSONObject(value.toString());
        } catch (JSONException ignore) {
            jsonObject = new JSONObject();
        } catch (ConcurrentModificationException e){
            throw new RuntimeException("eventId=" + eventId,  e);
        }
        // ...
}
```

【风险点】：仍需对底层容器做遍历操作；时机尽量前置，最好是在业务方线程处理

【风险点】：在业务方线程中做序列化会有性能损耗，需要做线下评估。序列化耗时和JSONObject长度正相关，1ms per 10 KB （100KB 10ms）

方案2：JSONObject.toString()失败重试

【操作】UBC SDK内部取消对 `JSONObject.toString()` 方法的直接调用，改为封装方法对 `toString()` 方法做失败重试调用。

```java
public static final int MAX_RETRY = 10; // 失败重试上限

public static String safeGetJSONContent(JSONObject jsonObject) {
    return safeGetJson(jsonObject, 0, null);
}

public static String safeGetJson(JSONObject jsonObject, int count, Throwable t) {
```

```
    if (count >= MAX_RETRY) {
        return "";
    }
    try {
        return jsonObject.toString();
    } catch (Exception e) {
        try {
            Thread.sleep(0);
        } catch (InterruptedException ignored) {
        }
        return safeGetJson(jsonObject, count + 1, e);
    }
}
```

【缺点】维护成本较高，后续UBC代码逻辑都需要调用该方法避免并发读写异常；

【缺点】序列化重试耗时也较高

方案3：完善异常捕获

在 `JSONObject.toString()` 方法调用的位置补充相应的异常捕获，上传异常点位，供业务方自查

方案4：完善异常捕获

目标：实验组线程异常处理逻辑和对照组一致，异常发生时，两个组的处理逻辑是一样的

优化：参照 `FutureTask` ，对 `PriorityTask` 做异常捕获，保证线程任务不引起崩溃

缺点：所有在 `UBC` 线程池提交的任务都需要做封装

PriorityTask Java

```java
public class PriorityTask implements Runnable, Comparable<PriorityTask> {
    @Override
    public void run() {
        if (DEBUG) {
            Log.d(TAG, "run() called with priority=" + mPriority + ", index=" + mIndex);
        }
        try {
            mTask.run();
        } catch (Throwable t){
            // 捕获到异常
```

```
        }finally {
        }
    }
}
```

> 1. 优先补充监控漏斗，确定量级；
> 2. 堆栈上报和崩溃同学确认下现有的上报路径；否则我们SDK通过1876点位自行上报，后续对相关上报堆栈信息自行分析

评审意见：完善方案4，保证实验组的异常处理逻辑和对照组一致，数据指标上后续可以关注线程处理率及相关数据漏斗