

UBC Android 多进程场景时长不自洽修复方案

目录

- 一、背景
- 二、数据分析&归因
 - 2.1 18点位时长不自洽统计数据
 - 2.2 日志回捞
- 三、原因分析-多进程场景时长不自洽
- 四、修复方案
 - 4.1 方案1: 将in标记修改为inout
 - 4.2 方案2: flowEnd数据库持久化endtime不修改
- 五、附录
 - 5.1 竞品调研-神策
 - 神策跨进程处理时长点位逻辑
 - 关键点
 - 结论

一、背景

近期排查时长点位时长不自洽问题——即 `duration != endtime - starttime`。以18点位为切入点，通过日志回捞，对18点位时长不自洽问题做了归因：

1	日志完整性	日志表现/日志特征	原因&场景
	日志不完整	1. <code>endtime = 0</code>	a. 跨版本上报；日志在旧版本产生，在新版本上报


2	<ul style="list-style-type: none">• 字段不全• 值为0• 时长不自洽			b. 极端场景 SQLite 异常，endFlow 失败，日志丢弃不上报
3		2. duration = 0	2.1 duration = '0' <ul style="list-style-type: none">• '0' 是由业务方传入	a. 极端场景下 SQLite 异常，fSVWD 失败，endFlow 成功——是接口调用失败，还是本来就要基于旧格式上报，暂无法修复，基于类异常日志 b. 多进程场景 fSVWD 未调用或调用失败，endFlow 成功
4	日志完整		2.2 duration = '0.000'	a. 时间跳变，endtime < starttime；日志完整
5	<ul style="list-style-type: none">• 字段完整• 值不为0• 时长不自洽	3. 多进程场景 <ul style="list-style-type: none">• 业务方线程 pid!=tid• 都是跨进程调用	3.1 跨进程调用行为无异常	a. 进程间通信，Flow 实例不同步，时长入库后 endtime 取新时间
6			3.2 跨进程调用行为异常 <ul style="list-style-type: none">• duration = '0'	a. 多进程场景 fSVWD 未调用或调用失败，endFlow 成功 --> 日志
7			3.3 跨进程调用行为异常 <ul style="list-style-type: none">duration != 0	a. 多进程场景 fSVWD 成功，endFlow 成功
8		4. 其他	4.1 handleID 重复	a. 极端场景下，handleID 持久化失败，跨生命周期导致同点位拿修改了旧日志并上报。

文档方案2用于解决3.1 & 3.3日志问题，通过统计日志回捞数据，3.1 & 3.3 占时长不自洽点位pv的97%

二、数据分析&归因

2.1 18点位时长不自洽统计数据

查询20240507 v13.40+ 18点位日志数据：



v13.40+ 0507 18点位时长一致性结果统计，双端.xlsx

9.5KB

os_name	total	error_count
iphone	285,554,026	5,362
android	942,817,914	29,328,047
		3.11%

Android端18点位总PV = 942,817,914，其中有时长不自洽PV = 29,328,047，时长不自洽点位pv占比 3.11%

查询部分日志明细，发现一些日志具有明显的异常特征，如 `endtime = 0` `pv = 8856`或 `duration = 0` `pv=12776` 占比都很小，说明不是造成时长不自洽的主要问题

其中 `endtime = 0` 经排查全部都是旧版本产生并打包的日志在新版本上报导致的。归因过程不再分析该情况

2.2 日志回捞

对500名用户下发日志回捞任务，获取了20240420~20240427七日18点位有效日志19911条，其中时长不自洽日志共2431条

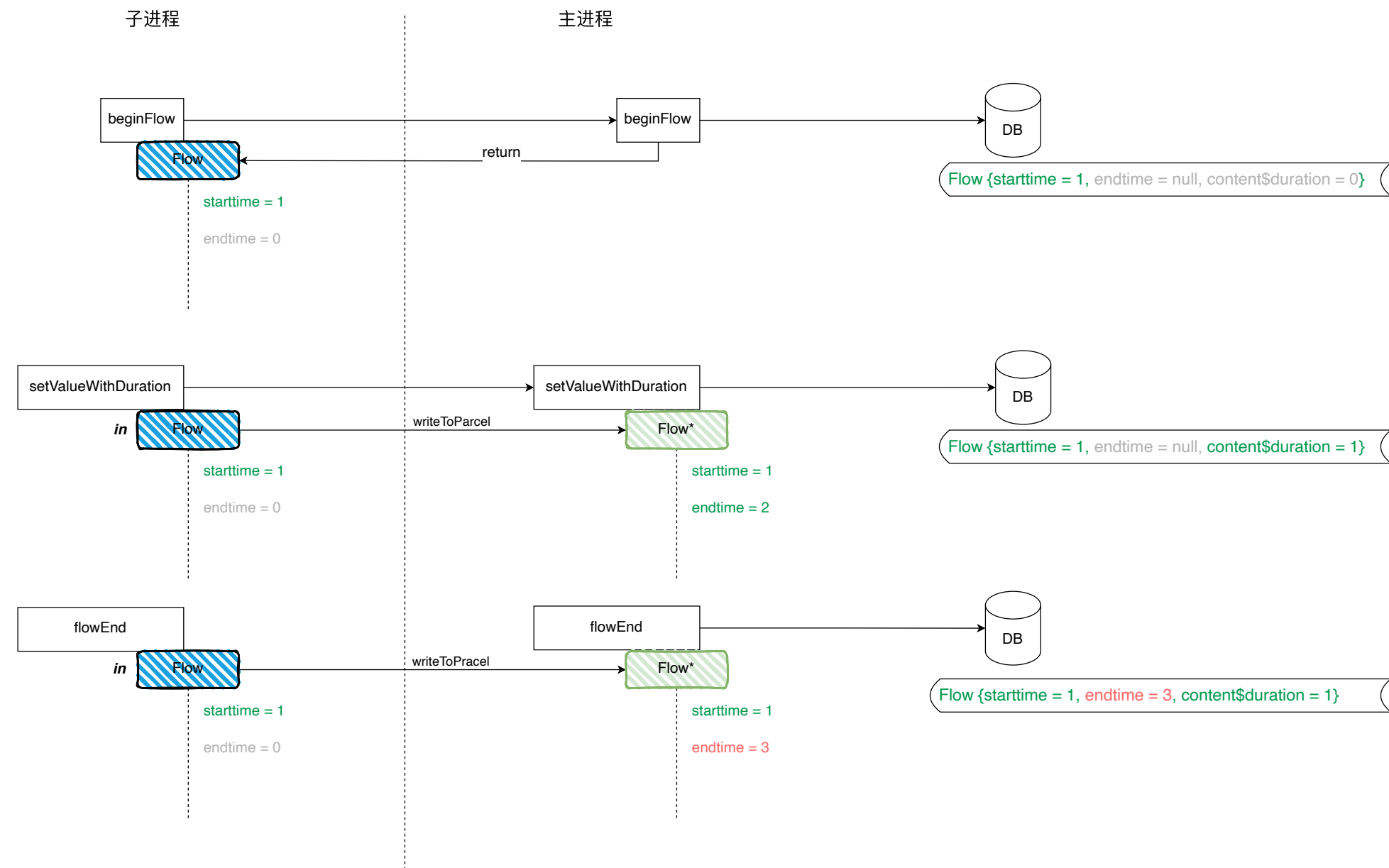
对这2431条异常日志做归因：



三、原因分析-多进程场景时长不自洽

UBC时长打点逻辑主要涉及到的三个UBC方法

- UBCServiceManager#beginFlow(): Flow
- UBCServiceManager#flowSetValueWithDuration()
- UBCServiceManager#flowEnd()



- 子进程开启时长打点
 - `beginFlow`

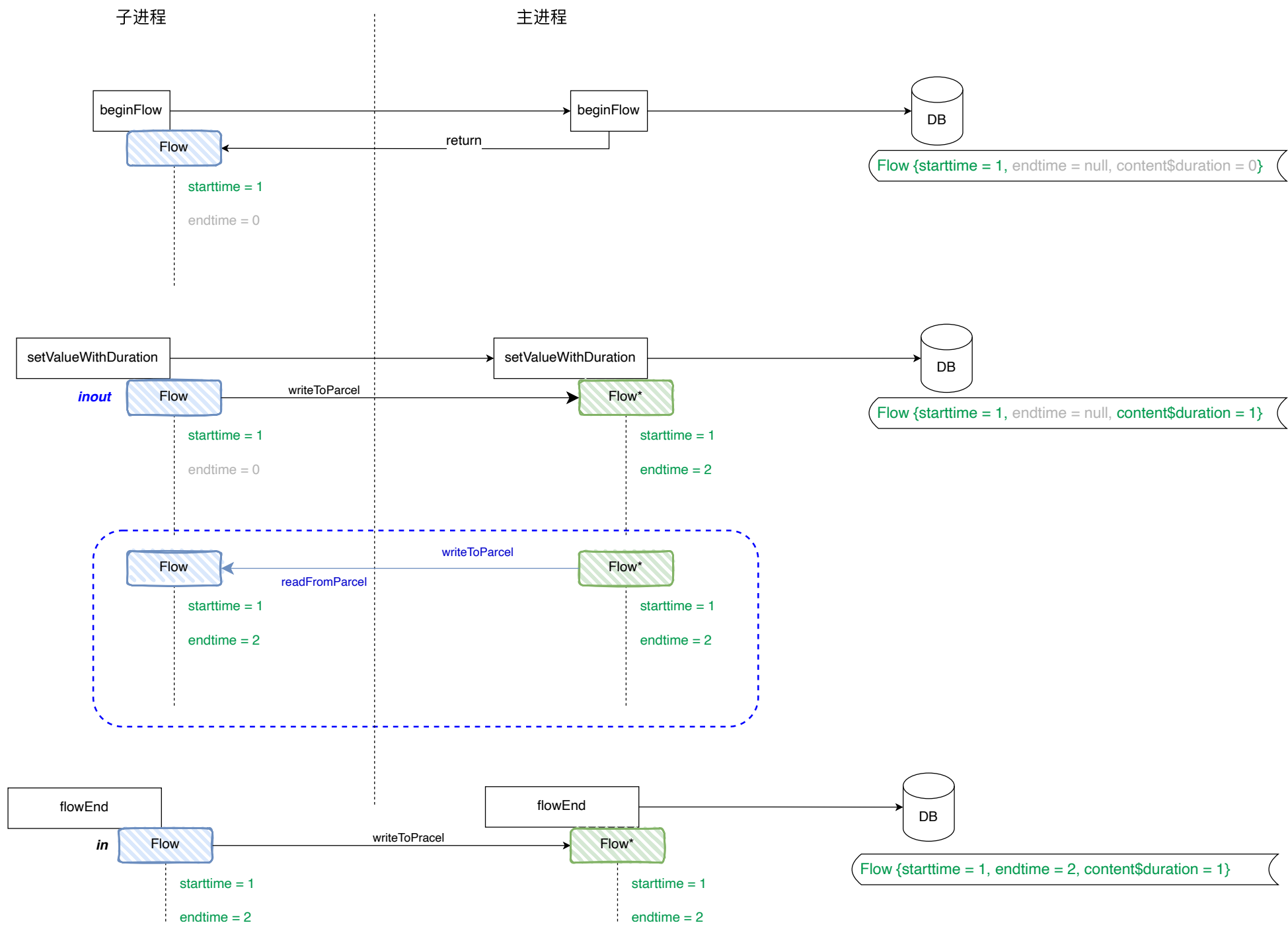
- IPC 调用主进程方法，获得一个 Flow 实例，子进程持有该 Flow 实例
- 子进程结束时长打点
 - setValueWithDuration
 - IPC 调用主进程方法，子进程 Flow 实例作为方法的参数，标记为 in ，主进程获得一个序列化的 Flow* 实例
 - Flow* 的 endtime 取时间戳，同时计算时长 duration 并持久化到数据库中
 - endFlow 方法，通过 IPC 调用主进程方法
 - IPC 调用主进程方法，子进程 Flow 实例以 in 模式传入方法列表，主进程获得一个序列化的 Flow* 实例
 - Flow* 的 endtime 值仍为 0 ，为保证 endtime 不为 0 ，又取了一次时间戳，并把数据持久化到数据库

setValueWithDuration 的 IPC 方法以 in 来标记 Flow 参数，数据流向是 子进程 -> 主进程 ，主进程对 Flow* 赋值了 endtime 并不能回写到子进程的 Flow 实例。之后调用 flowEnd 时得到的 Flow* 实例 endtime 仍然未赋值，导致了 endtime 二次赋值，持久化到数据库时覆盖了原本的 endtime ，而 content\$*** 内容保持不变，导致了 **endtime - starttime != duration**

四、修复方案

4.1 方案1: 将 in 标记修改为 inout

保证数据双向流动——主进程修改 Flow* 后，主进程多一步 writeToParcel ，子进程会多一步 readFromParcel 来同步主进程对 Flow* 实例的修改



【风险点1】性能损耗

理论上因为多了一次读写(序列化-反序列化), 性能上会有损耗。线下调试发现损耗几乎可以忽略不计, IPC方法耗时都是在 2~5ms

【风险点2】是否因数据双向流动, 暴露并引入新问题

这里的数据双向流动有作用域, flowSetValueWithDuration 方法区间内的变化回被回写

经过排查, 在 flowSetValueWithDuration 方法中主进程对于 Flow 实例仅对 endtime 做了修改, 不会有额外的影响

代码示例

</>Java

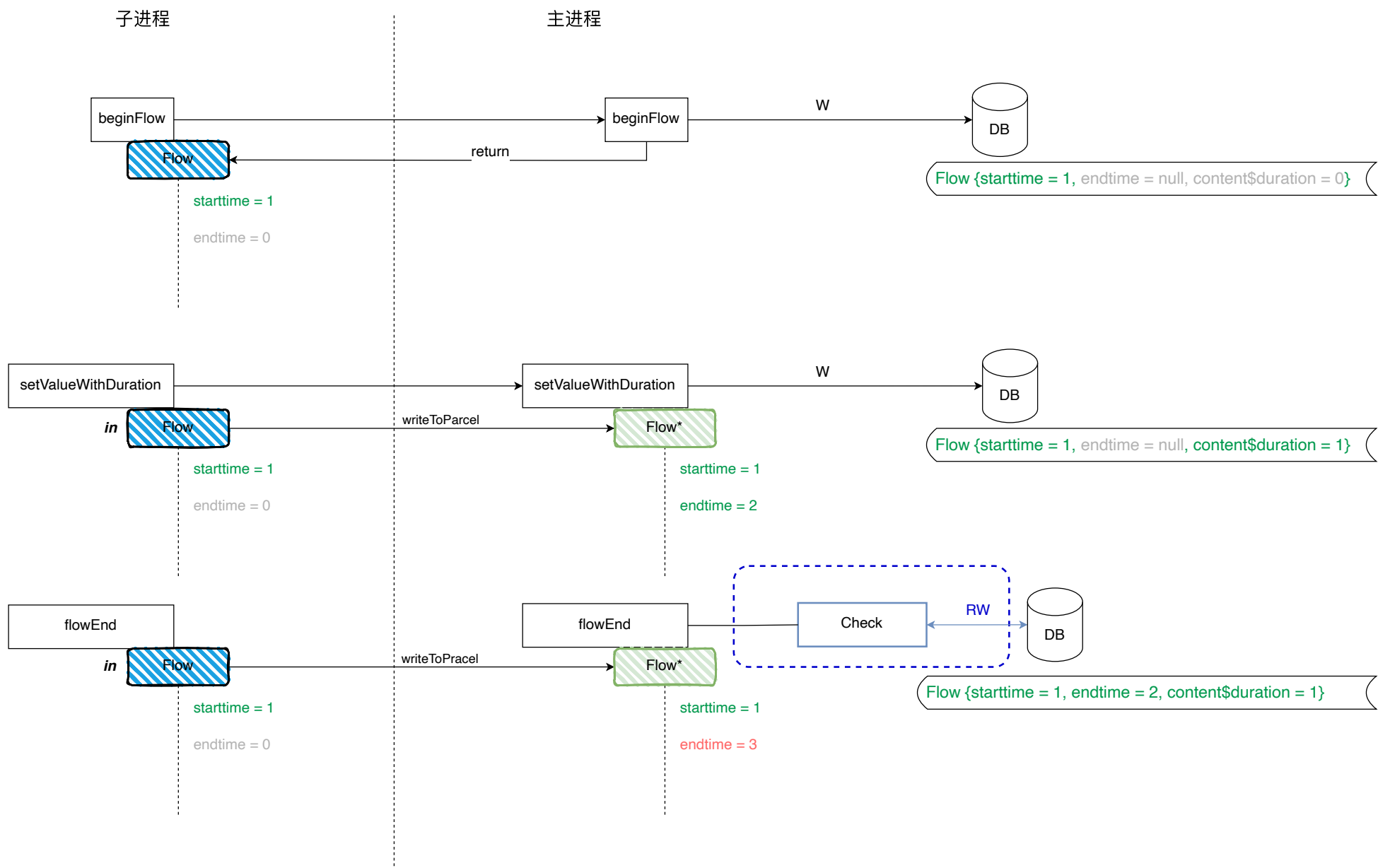
```
1  /**
2   * 流对象时长统计的流更新内容,同时记录时长
3   *
4   * @param flow 流对象
5   * @param value json串, 流的具体内容, 没有则传空, 只会设置时
  长
6   */
7   void flowSetValueWithDuration(in Flow flow, String
  value);
```

</>Java

```
1  /**
2   * 流对象时长统计的流更新内容,同时记录时长
3   *
4   * @param flow 流对象
5   * @param value json串, 流的具体内容, 没有则传空, 只会设置时
  长
6   */
7   void flowSetValueWithDuration(inout Flow flow, String
  value);
```

4.2 方案2: flowEnd数据库持久化endtime不修改

- endtime 在 setValueWithDuration 阶段随 duration 入库
- flowEnd 方法数据库持久化过程中检查该条日志的 endtime 是否已经有值, 若有值则不修改 endtime



【风险点】不引入额外的数据库IO操作，需要替换endFlow入库API

CASE WHEN 语法能够保证在 update 操作时，依据条件更新部分字段。endFlow 入库使用 db.update() 更新日志记录，该 API 不支持 CASE WHEN 且 SQL 注入无效。

需要使用 db.execSQL(sql) ，执行一条拼接的 SQL

【SQL对比】

</> endFlow现行SQLSQL

```
1 UPDATE TABLE_FLOW
2 SET
3     COLUMN_STATE = FLOW_STATE_END, # Flow状态为END
4     COLUMN_END_TIME = endtime, # endtime
5     COLUMN_LOGID = logID, # logId
6     COLUMN_SLOT = slotArray.toString() # slot
7 WHERE COLUMN_FLOW_ID = flowID AND COLUMN_FLOW_HANDLE_ID =
   flowHandle
```

</> endFlow目标SQLSQL

```
1 # endFlow目标SQL
2 UPDATE TABLE_FLOW
3 SET
4     COLUMN_STATE = FLOW_STATE_END, # Flow状态为END
5     COLUMN_END_TIME = CASE WHEN COLUMN_END_TIME IS NULL
   THEN endTime ELSE COLUMN_END_TIME END, # endtime
6     COLUMN_LOGID = logID, # logId
7     COLUMN_SLOT = slotArray.toString() # slot
8 WHERE COLUMN_FLOW_ID = flowID AND COLUMN_FLOW_HANDLE_ID =
   flowHandle
```

【代码对比】

</>Java

```
1 void endFlow(String flowId, int flowHandle, long endTime,
   JSONArray slotArray, String logId) {
2     // ...
3     final ContentValues cv = new ContentValues();
4     cv.put(COLUMN_STATE, Constants.FLOW_STATE_END);
5     cv.put(COLUMN_END_TIME, endTime);
```

</>Java

```
1 void endFlow(String flowId, int flowHandle, long endTime,
   JSONArray slotArray, String logId) {
2     // ...
3     StringBuilder updateSql = new StringBuilder();
4     updateSql.append("UPDATE " + TABLE_FLOW + " SET ")
```

```
6     if (!TextUtils.isEmpty(logId)) {
7         cv.put(COLUMN_LOGID, logId);
8     }
9     if (slotArray != null && slotArray.length() > 0) {
10        cv.put(COLUMN_SLOT, slotArray.toString());
11    }
12    StringBuilder sb = new StringBuilder();
13    sb.append(COLUMN_FLOW_ID)
14        .append("=\")
15        .append(flowId)
16        .append("\")
17        .append(" AND ")
18        .append(COLUMN_FLOW_HANDLE_ID)
19        .append(" = ")
20        .append(flowHandle);
21    final String where = sb.toString();
22    count = db.update(TABLE_FLOW, cv, where, null);
23    // ...
24 }
```

```
5        .append(COLUMN_STATE + " = " +
6        Constants.FLOW_STATE_END + ", ")
7        .append(COLUMN_END_TIME + " = CASE WHEN " +
8        COLUMN_END_TIME + " IS NULL THEN
9        ").append(endTime).append(" ELSE " + COLUMN_END_TIME + "
10        END");
11    if (!TextUtils.isEmpty(logId)) {
12        updateSql.append(", " + COLUMN_LOGID + " =
13        ").append(logId);
14    }
15    if (slotArray != null && slotArray.length() > 0) {
16        updateSql.append(", ").append(COLUMN_LOGID + " =
17        ").append(slotArray);
18    }
19    updateSql.append(" WHERE ")
20        .append(COLUMN_FLOW_ID)
21        .append("=\")
22        .append(flowId)
23        .append("\")
24        .append(" AND ")
25        .append(COLUMN_FLOW_HANDLE_ID)
26        .append(" = ")
27        .append(flowHandle);
28    db.execSQL(updateSql.toString());
29    // ...
30 }
```

五、附录

5.1 竞品调研-神策

针对UBC SDK多进程场景下时长不自洽的问题，分析了神策SDK的时长打点代码逻辑和多进程处理方式

神策跨进程处理时长点位逻辑

- 神策文档
- Github 神策Android

核心方法

</>Java

```
1 SensorsDataAPI#trackTimerStart(String eventName)
2 SensorsDataAPI#trackTimerPause(final String eventName)
3 SensorsDataAPI#trackTimerResume(final String eventName)
4 SensorsDataAPI#trackTimerEnd(final String eventName, final JSONObject properties)
```

SensorsDataAPI#trackTimerStart(String eventName)

给 `eventName` 生成 `EventTimer` 实例并缓存到 `Map` 中

</> com.sensorsdata.analytics.android.sdk.SensorsDataAPI#trackTimerStartJava

```
1 @Override
2 public String trackTimerStart(String eventName) {
3     try {
4         final String eventNameRegex = String.format("%s_%s_%s", eventName, UUID.randomUUID().toString().replace("-",
5             "_"), "SATimer");
6         trackTimer(eventNameRegex, TimeUnit.SECONDS);
6         trackTimer(eventName, TimeUnit.SECONDS);
```

```
7         return eventNameRegex;
8     } catch (Exception ex) {
9         SLog.printStackTrace(ex);
10    }
11    return "";
12 }
```

</> com.sensorsdata.analytics.android.sdk.SensorsDataAPI#trackTimer

Java

```
1  @Deprecated
2  @Override
3  public void trackTimer(final String eventName, final TimeUnit timeUnit) {
4      final long startTime = SystemClock.elapsedRealtime();
5      mTrackTaskManager.addTrackEventTask(new Runnable() {
6          @Override
7          public void run() {
8              try {
9                  SDataHelper.assertEventName(eventName);
10                 EventTimerManager.getInstance().addEventTimer(eventName, new EventTimer(timeUnit, startTime));
11             } catch (Exception e) {
12                 SLog.printStackTrace(e);
13             }
14         }
15     });
16 }
```

</> com.sensorsdata.analytics.android.sdk.core.business.timer.EventTimerManager#addEventTimer

Java

```
1  public void addEventTimer(String eventName, EventTimer eventTimer) {
2      synchronized (mTrackTimer) {
3          // remind: update startTime before runnable queue
```

```
4         mTrackTimer.put(eventName, eventTimer);
5     }
6 }
```

SensorsDataAPI#trackTimerEnd(final String eventName, final JSONObject properties)

</> com.sensorsdata.analytics.android.sdk.SensorsDataAPI#trackTimerEnd()

Java

```
1 @Override
2     public void trackTimerEnd(final String eventName, final JSONObject properties) {
3         final long endTime = SystemClock.elapsedRealtime();
4         try {
5             final JSONObject cloneProperties = JSONUtils.cloneJsonObject(properties);
6             mTrackTaskManager.addTrackEventTask(new Runnable() {
7                 @Override
8                 public void run() {
9                     if (eventName != null) {
10                         EventTimerManager.getInstance().updateEndTime(eventName, endTime);
11                     }
12                     try {
13                         JSONObject _properties =
14 SAModuleManager.getInstance().invokeModuleFunction(Modules.Advert.MODULE_NAME,
15 Modules.Advert.METHOD_MERGE_CHANNEL_EVENT_PROPERTIES, eventName, cloneProperties);
16                         if (_properties == null) {
17                             _properties = cloneProperties;
18                         }
19                         mSAContextManager.trackEvent(new
20 InputData().setEventName(eventName).setEventType(EventType.TRACK).setProperties(_properties));
21                     } catch (Exception e) {
22                         SALog.printStackTrace(e);
23                     }
24                 }
25             });
26         } catch (Exception e) {
27             SALog.printStackTrace(e);
28         }
29     }
```

```
21         }
22     }
23 });
24 } catch (Exception e) {
25     SALog.printStackTrace(e);
26 }
27 }
```

</> com.sensorsdata.analytics.android.sdk.core.event.EventProcessor#process

Java

```
1  /**
2   * data process
3   *
4   * @param input DataInput
5   */
6  protected synchronized void process(InputData input) {
7      try {
8          // 1. assemble data
9          Event event = assembleData(input);
10         // 2. store data
11         int errorCode = storeData(event);
12         // 3. send data
13         sendData(input, errorCode);
14     } catch (Exception e) {
```

关键点

数据库持久化方案：

SQLite

- 时长打点的业务逻辑不直接操作数据库，而是通过 `ContentProvider` 组件对 `SQLite` 读写

从时长打点的业务逻辑上看：

- 时长打点流程中只有 `trackTimerEnd` 阶段才入库。`starttime` , `endtime` 不入库，只用于计算 `duration`
 - `starttime` , `endtime` 取 `SystemClock` API的时间，计算的 `duration` 相当于 UBC SDK 的 `cduration`，不受系统时间跳变影响；
- 当开启一个时长打点时，SDK 内部使用 `Map` 对 `eventName` 和 `EventTimer` 做映射(`eventName` 使用 `UUID` 拼出唯一标识字符串作为 `key`)，调用方可以获取到 `key`；`EventTimer` 由 SDK 内部持有，并通过 `key` 映射访问，只处理计时逻辑

处理多进程场景的方式：

- 时长打点的业务逻辑不直接操作数据库，而是通过 `ContentProvider` 组件对 `SQLite` 读写
- 在多进程场景中，`ContentProvider` 可以做到在多进程场景下仍是单实例，避免了进程间通信逻辑可能带来的并发问题

结论

结论1：

【神策SDK】采用`ContentProvider`组件间接操作`SQLite`数据库。`ContentProvider`是Android中的一个组件，可以作为跨进程通信的一种方式，默认在多进程中保持单实例，通过这种方式处理多进程场景下并发读写数据库的问题

【UBC SDK】采用AIDL的IPC方式，业务逻辑也切换到主进程执行

结论2：

【神策SDK】`starttime`,`endtime`仅用于计算`duration`，不会追加到日志中；所以对于时长是否自洽是不做解释的

【UBC SDK】`starttime`,`endtime`是具有可读性的时间戳，用于计算`duration`日志中也需要保证时长自洽

结论3:

【神策SDK】结束时长打点时才会通过ContentProvider组件写库，会有一次IPC跨进程通信。此前的其他阶段数据都在内存缓存中，没有其他IPC过程

【UBC SDK】starttime,duration,endtime会在三个阶段分别写库；若在子进程调用也会有三次IPC，Flow实例作为方法参数在进程间传递(进程间内存不共享，实例对象需要序列化)