Logging to disk reactively on Android

Karn Saheb Feb 24, 2020 · 9 min read

Logging is not a new concept, `Log.v` here, `Log.d` there, and a `Log.wtf` for good measure. For anyone working on a large project, logging is often a utility that is taken for granted — generally the plumbing has already been done to ensure that logs are captured, stored, rotated, and eventually uploaded to some external service to aid with debugging.

This past year I've been fortunate enough to have a project see enough traffic and by proxy a set of more nuanced bugs that I can no longer just *find* the obvious issue. Partial logs that are bundled with the Firebase Crashlytics tool no longer capture enough valuable information to deal with non-crash related events, or that pesky unexpected state that the user is in that for the life in you, you can't seem to be able to reproduce. The point is, its no longer feasible to not have a clearer picture of what's happening in your application when you get one of these reports. And so, it's time to bring in the big guns.

Logging on android is rather straightforward, you can use the regular `android.util.Log` class to do just about everything, the Android Studio (or terminal) Logcat utility is great at debugging information, but out of the box, the logs are not saved to disk and as a result, cannot be uploaded for remote triaging. I use Timber for logging, it's a lightweight utility for logging which wraps the `android.util.Log` class for its console output but also provides abstractions of logging so that you can bake in your own custom solution.

```
// Somewhere in your application's onCreate(...).
Timber.plant(Timber.DebugTree(),
        CrashlyticsTree())
```

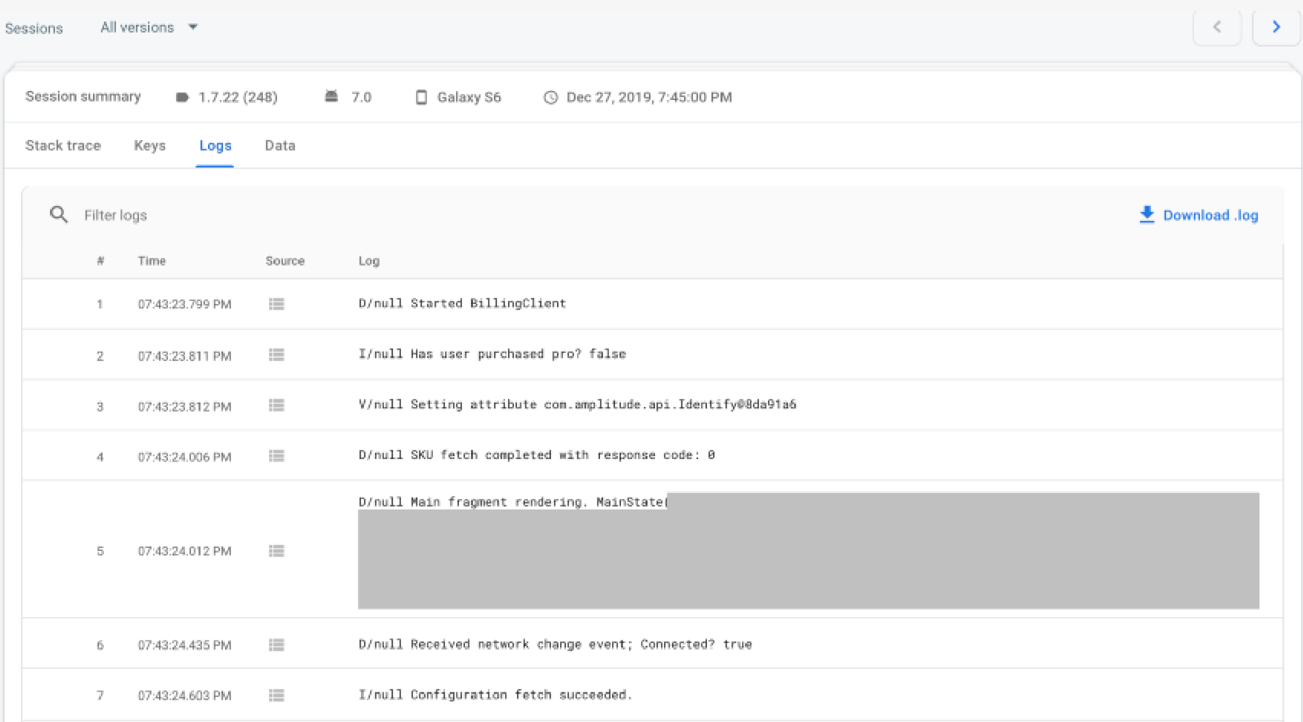
Prior to requiring the logs to be viewed remotely, I had two log "Trees" configured. The first was the `Timber.DebugTree` which performed the necessary logging to Android's Logcat, and a custom one to log to Crashlytics, unoriginally named

`CrashlyticsTree`. The implementation for the latter, and how the logs are viewed can be seen below:

```
class CrashlyticsTree : Timber.Tree() {

    /**
     * Write a log message to its destination. Called for all level-
specific methods by default.
     *
     * @param priority Log level. See [Log] for constants.
     * @param tag Explicit or inferred tag. May be `null`.
     * @param message Formatted log message. May be `null`, but then
`t` will not be.
     * @param t Accompanying exceptions. May be `null`, but then
`message` will not be.
     */
    override fun log(priority: Int, tag: String?, message: String,
t: Throwable?) {
        if (!Fabric.isInitialized()) {
            return
        }

        Crashlytics.log(priority, tag, message)
    }
}
```



An example crash report which contains a subset of the logs up until the crash was handled.

Writing to disk seems like a rather practical thing to do. The naive approach is to keep a reference to the filehandle and `FileWriter#append(...)` all the logs, but as we perhaps all know, disk IO operations can be rather taxing to the device, not to mention that synchronous logging to disk is already problematic since it would be, by definition, a blocking operation.

The question then becomes how do you handle logging asynchronously to disk.

Since I was already using RxJava I decided to make use of the schedulers to defer logs to a background thread and buffer the logs before writing them to disk as a batch operation. The implementation of the `Tree` is as follows.

The log collector and emitter

The process began by first creating the `Tree` class as well as a PublishSubject to allow the logs to be written to and later processed from.

```
/**
 * The LogElement triple provides an easy wrapper for the Date
 * (as a string), the priority (log level), and the log message.
 */
typealias LogElement = Triple<String, Int, String?>

/**
 * The FileTree is the additional log handler which we plant.
 * It's role is to buffer logs and periodically write them to disk.
 */
class FileTree : Timber.Tree() {
     private val LOG_LINE_TIME_FORMAT = SimpleDateFormat("yyyy-MM-dd
HH:mm:ss", Locale.US)

    /**
     * The Observable which will receive the log messages which are
     * to be written to disk.
     */
    private val logBuffer = PublishSubject.create<LogElement>()

    init {
        logBuffer.subscribe { it : LogElement ->
            // TODO: Handle incoming log lines
        }
    }

    /**
     * Schedule this log to be written to disk.
     */
    override fun log(priority: Int, tag: String?, message: String,
t: Throwable?) {
    // For the sake of simplicity we skip logging the exception,
    // but you can parse the exception and and emit it as needed.
    logBuffer.onNext(
        LogElement(LOG_LINE_TIME_FORMAT.format(Date()),
```

```
        priority,
        message)
    )
}
```

Once the tree was created it was trivial to "plant" it.

```
// Somewhere in your application's onCreate(...).
Timber.plant(Timber.DebugTree(),
        CrashlyticsTree(),
        FileTree())
```

The next step was to set up a buffering technique, which Rx has built-in using the buffer. My buffering criteria was every 5 minutes or 20 log lines.

```
logBuffer.observeOn(Schedulers.computation())
         .buffer(5 /* Timespan */, TimeUnit.MINUTES, 20 /* count */)
         .subscribeOn(Schedulers.io())
         .subscribe { // it : LogElement ->
             // TODO: Write to disk
         }
```

The issue here was that I would eventually need a method of manually flushing the logs to disk once certain actions were taken, for example when the user submitted a report manually or when the application was closed.

The solution was not quite as elegant as I would have liked. The default *buffer* operators did not provide an external signal argument. I opted for a simpler solution — use a manual signal to indicate that the buffer needed to be released.

```
// In the FileTree Class
companion object {
    /**
     * Flush sends a signal which allows the buffer to release it's
     * contents downstream.
     */
    private var flush = BehaviorSubject.create<Long>()
}
```

This meant that I would need to modify the buffer to accept this signal.

```
// Maintain a count of the processed LogElements
val processedCount = 0

logBuffer.observeOn(Schedulers.computation())
    // Increment the counter after each item is processed
    // and perform a flush if the criteria is met.
    .doAfterEach {
       processedCount++

       if (processedCount % 20 == 0) {
           flush.onNext(1L)
        }
    }
    // Merge the signal from flush and the signal from
    // the interval observer to create a dual signal.
    .buffer(flush.mergeWith(Observable.interval(5,
TimeUnit.MINUTES)))
    .subscribeOn(Schedulers.io())
    .subscribe { // it : LogElement ->
        // TODO: Write to disk
    }
```

Once I had the buffering ready, the next step was to write the logs to disk. This involved opening the file and appending all the buffered LogElements and then closing (and flushing) the `FileWriter`.

```
logBuffer.observeOn(Schedulers.computation())
    // ...
    .subscribe { // it: LogElement
        try {
            // Validate the existence of the file and grab it in
            // preperation to write.
            val f = getFile(filePath, LOG_FILE_NAME)

            // Open the file and write.
            FileWriter(f, true).use { fw ->
                // Write LogElements to the file
                it.forEach { (date, priority, message) ->
                    fw.append("$date\t" +
                              "${LOG_LEVELS[priority]}\t" +
                              "$message\n")
                }

                fw.flush() // Flush the FileWriter.
            }
        } catch (e: Exception) {
            // Handle the any IO exceptions here
        }
    }
```

Now that the logs were being written to disk, the next step was to build a mechanism to rotate the logs as needed. In my case, I chose to prune log files older than 14 days and rotate the logs as soon as they were larger than about 1.66 MB so that when viewing log files using gzcat, for example, they'd be fast to scroll through.

Log rotation methodology

Log rotation is a simple concept. Once your log file gets large enough it is "rotated" and the log files are pruned.

I chose to keep the individual files pretty small and 1.66MB seemed like a reasonable size. The logic was modified to validate file size and rotate if required. This rotation step involved gzipping the file to reduce the file size for upload.

The trick here was that since the writing to disk was asynchronous, I'd need to wait the operation was complete to decide to rotate the logs. This lent itself to the use of another observer which would receive a signal with the size of the current log file when the log writing operation was complete.

```
// In the FileTree Class
companion object {
    /**
     * Flush sends a signal which allows the buffer to release it's
     * contents downstream.
     */
    private var flush = BehaviorSubject.create<Long>()
    /**
     * Signal that the flush has completed
     */
    private var flushCompleted = BehaviorSubject.create<Long>()
}
```

I updated the `logBuffer`'s subscribe to emit the filesize to this observer.

```
logBuffer.observeOn(Schedulers.computation())
    // ...
    .subscribe { // it: LogElement
        try {
            // Validate the existence of the file and grab it in
            // preperation to write.
            val f = getFile(filePath, LOG_FILE_NAME)

            // Open the file and write.
            FileWriter(f, true).use { fw ->
```

```
                    // Write LogElement to the file
                    it.forEach { (date, priority, message) ->
                        fw.append("$date\t" +
                                  "${LOG_LEVELS[priority]}\t" +
                                  "$message\n")
                    }

                    fw.flush() // Flush the FileWriter.
                }

                flushCompleted.onNext(f.length())
            } catch (e: Exception) {
                // Handle the any IO exceptions here
            }
        }

    flushCompleted
            .subscribeOn(Schedulers.io())
            .filter { size -> size > LOG_FILE_MAX_SIZE_THRESHOLD }
            .subscribe { rotateLogs(filePath, LOG_FILE_NAME) }
```

Next, I implemented the `rotateLogs` function and a helper to compress the files and clear the log file.

```
/**
 * A utility function to rotate application logs. This function
 * operates in three steps.
 * 1. Compresses the existing log file into a gzip format.
 * 2. Truncate the existing log file to size zero to reset it.
 * 3. Grab all the compressed files that are outside the retention
 *    period and delete them.
 */
private fun rotateLogs(path: String, name: String) {
    val file = getFile(path, name)

    if (!compress(file)) {
        // Unable to compress file
        throw IOException("Failed to compress files for rotation")
    }

    // Truncate the log file to zero.
    PrintWriter(file).close()

    // Iterate over the gzipped files in the directory and delete
    // the files outside the retention period.
    val currentTime = System.currentTimeMillis()
    file.parentFile.listFiles()
            ?.filter {
                it.extension.toLowerCase(Locale.ROOT) == "gz"
                        && it.lastModified() + LOG_FILE_RETENTION <
currentTime
            }?.map { it.delete() }
}
```

and finally the logic for compressing the file:

```
// Define the DateTime format which will be used in the file names.
private val LOG_FILE_TIME_FORMAT = SimpleDateFormat("yyyy-MM-dd_HH-
mm-ss", Locale.US)

private fun compress(file: File): Boolean {
    try {
        // Create a new file for the compressed logs.
        val c = File(file.parentFile.absolutePath,
                     "${file.name.substringBeforeLast(".")}" +
                     "_${LOG_FILE_TIME_FORMAT.format(Date())}.gz")

        FileInputStream(file).use { input ->
            FileOutputStream(compressed).use { output ->
                GZIPOutputStream(output).use { zipped ->

                    val buffer = ByteArray(1024)
                    var length = input.read(buffer)

                    while (length > 0) {
                        zipped.write(buffer, 0, length)

                        length = input.read(buffer)
                    }

                    // Finish file compressing and close all
                    // streams.
                    zipped.finish()
                }
            }
        }
    } catch (e: IOException) {
        // TODO: Handle exception
        return false
    }

    return true
}
```

And that is it! Once the implementation was complete the logs were being written to disk and being rotated as expected.

logs	drwx------	2020-02-21 17:16	3.4 KB
insights.log	-rw-------	2020-02-22 10:57	1.3 MB
insights_2020-02-10_01-16-01.gz	-rw-------	2020-02-10 01:16	83.9 KB
insights_2020-02-10_01-20-39.gz	-rw-------	2020-02-10 01:20	1.2 KB
insights_2020-02-10_21-05-24.gz	-rw-------	2020-02-10 21:05	740.3 KB
insights_2020-02-13_03-05-41.gz	-rw-------	2020-02-13 03:05	120.7 KB

Viewing the logs directory

Future work

I’ve had this implementation live for almost two months now and have been able to track down a number of issues with the help of logs submitted by users — spoiler: I’ll cover my approach to uploading to Firebase CloudStorage in another post. However, there are still a few things that I’d like to improve upon in the future, below are some of these items:

- I’d like to consider implementing a backpressure strategy by using a PublishProcessor instead of a PublishSubject for the `logBuffer`. Sometimes large JSON objects are being written to logs and the buffer can quickly fill up.
- On a related note to the above, I’d like to change the size of the Strings to smaller 1024 byte sizes to prevent possible memory-related exceptions on less powerful devices that could be holding a few hundred KBs in logs in memory.
- Consider using Kotlin Flow in place of RxJava — as we all perhaps know, RxJava is often abused to do concurrency/asynchronous work, moving the logic to a paradigm that is better suited for the kind of work being done here might be valuable.
- Implementing post-processing for log lines to strip out sensitive data before writing logging them to console and to disk.

. . .

Thanks for reading and be sure to check out the example source code along with a followup post on how I handle uploading to Firebase CloudStorage in the near future.

If you have any more feedback, comments, or if there is a glaring mistake, let me know below!

Android Logging Rxjava Kotlin