Android Developers

Launch-Time Performance

Users expect apps to be responsive and fast to load. An app with a slow startup time doesn't meet this expectation, and can be disappointing to users. This sort of poor experience may cause a user to rate your app poorly on the Play store, or even abandon your app altogether.

This document provides information to help you optimize your app's launch time. It begins by explaining the internals of the launch process. Next, it discusses how to profile startup performance. Last, it describes some common startup-time issues, and gives some hints on how to address them.

In this document

Launch Internals

App launch can take place in one of three states, each affecting how long it takes for your app to become visible to the user: cold start, warm start, and lukewarm start. In a cold start, your app starts from scratch. In the other states, the system needs to bring the app from the background to the foreground. We recommend that you always optimize based on an assumption of a cold start. Doing so can improve the performance of warm and lukewarm starts, as well.

To optimize your app for fast startup, it's useful to understand what's happening at the system and app levels, and how they interact, in each of these states.

Cold start

A cold start refers to an app's starting from scratch: the system's process has not, until this start, created the app's process. Cold starts happen in cases such as your app's being launched for the first time since the device booted, or since the system killed the app. This type of start presents the greatest challenge in terms of minimizing startup time, because the system and app have more work to do than in the other launch states.

At the beginning of a cold start, the system has three tasks. These tasks are:

1. Loading and launching the app.
2. Displaying a blank starting window for the app immediately after launch.
3. Creating the app process. (https://developer.android.com/guide/components/processes-and-threads.html#Processes)

As soon as the system creates the app process, the app process is responsible for the next stages. These stages are:

1. Creating the app object.
2. Launching the main thread.
3. Creating the main activity.
4. Inflating views.
5. Laying out the screen.
6. Performing the initial draw.

Once the app process has completed the first draw, the system process swaps out the currently displayed background window, replacing it with the main activity. At this point, the user can start using the app.

Figure 1 shows how the system and app processes hand off work between each other.

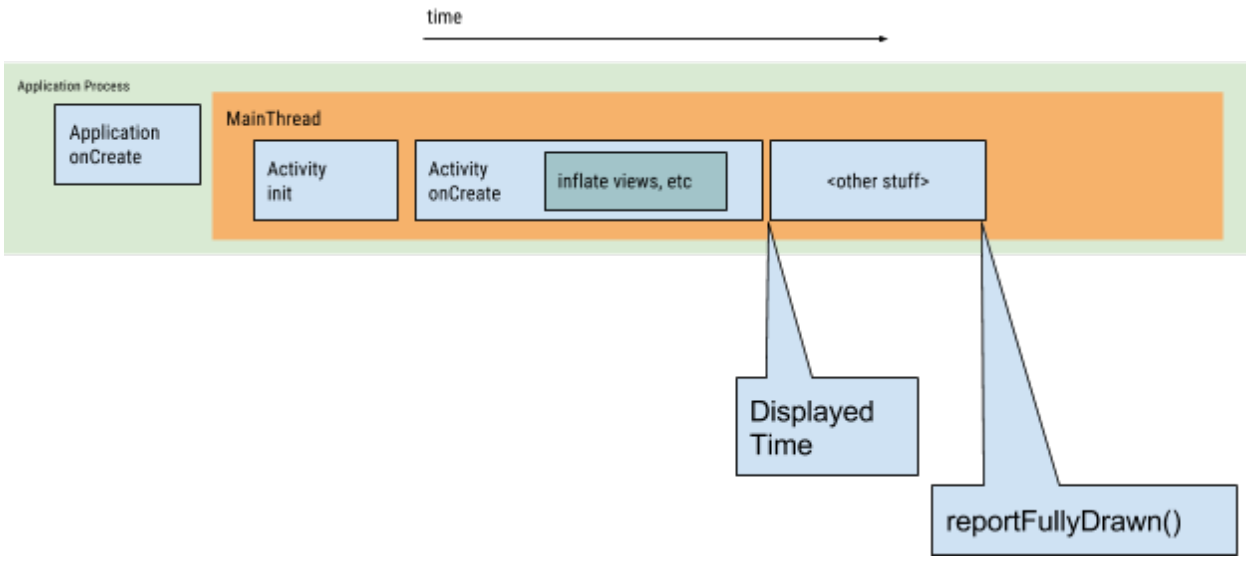

Figure 1. A visual representation of the important parts of a cold application launch.

Performance issues can arise during creation of the app and creation of the activity.

Application creation

When your application launches, the blank starting window remains on the screen until the system finishes drawing the app for the first time. At that point, the system process swaps out the starting window for your app, allowing the user to start interacting with the app.

If you've overloaded `Application.oncreate()` (https://developer.android.com/reference/android/app/Application.html#onCreate()) in your own app, the system invokes the `onCreate()` method on your app object. Afterwards, the app spawns the main thread, also known as the UI thread, and tasks it with creating your main activity.

From this point, system- and app-level processes proceed in accordance with the app lifecycle stages (https://developer.android.com/guide/topics/processes/process-lifecycle.html).

Activity creation

After the app process creates your activity, the activity performs the following operations:

1. Initializes values.
2. Calls constructors.
3. Calls the callback method, such as `Activity.onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)), appropriate to the current lifecycle state of the activity.

Typically, the `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method has the greatest impact on load time, because it performs the work with the highest overhead: loading and inflating views, and initializing the objects needed for the activity to run.

Warm start

A warm start of your application is much simpler and lower-overhead than a cold start. In a warm start, all the system does is bring your activity to the foreground. If all of your application's activities are still resident in memory, then the app can avoid having to repeat object initialization, layout inflation, and rendering.

However, if some memory has been purged in response to memory trimming events, such as `onTrimMemory()` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html#onTrimMemory(int)), then those objects will need to be recreated in response to the warm start event.

A warm start displays the same on-screen behavior as a cold start scenario: The system process displays a blank screen until the app has finished rendering the activity.

Lukewarm start

A lukewarm start encompasses some subset of the operations that take place during a cold start; at the same time, it represents less overhead than a warm start. There are many potential states that could be considered lukewarm starts. For instance:

- The user backs out of your app, but then re-launches it. The process may have continued to run, but the app must recreate the activity from scratch via a call to `onCreate()` (https://developer.android.com/reference/android

/app/Activity.html#onCreate(android.os.Bundle)).

- The system evicts your app from memory, and then the user re-launches it. The process and the Activity need to be restarted, but the task can benefit somewhat from the saved instance state bundle passed into `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)).

Profiling Launch Performance

In order to properly diagnose start time performance, you can track metrics that show how long it takes your application to start.

Time to initial display

From Android 4.4 (API level 19), logcat includes an output line containing a value called `Displayed`. This value represents the amount of time elapsed between launching the process and finishing drawing the corresponding activity on the screen. The elapsed time encompasses the following sequence of events:

1. Launch the process.
2. Initialize the objects.
3. Create and initialize the activity.
4. Inflate the layout.
5. Draw your application for the first time.

The reported log line looks similar to the following example:

```
ActivityManager: Displayed com.android.myexample/.StartupTiming: +3s534ms
```

If you're tracking logcat output from the command line, or in a terminal, finding the elapsed time is straightforward. To find elapsed time in Android Studio, you must disable filters in your logcat view. Disabling the filters is necessary because the system server, not the app itself, serves this log.

Once you've made the appropriate settings, you can easily search for the correct term to see the time. Figure 2 shows how to disable filters, and, in the second line of output from the bottom, an example of logcat output of the `Displayed` time.

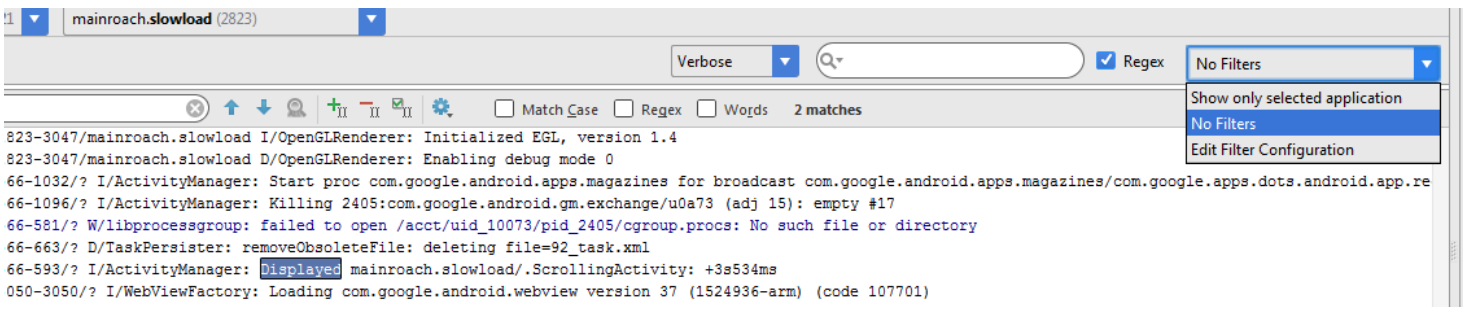

Figure 2. Disabling filters, and finding the `Displayed` value in logcat.

The `Displayed` metric in the logcat output does not necessarily capture the amount of time until all resources are loaded and displayed: it leaves out resources that are not referenced in the layout file or that the app creates as part of object initialization. It excludes these resources because loading them is an inline process, and does not block the app's initial display.

You can also measure the time to initial display by running your app with the ADB Shell Activity Manager (https://developer.android.com/studio/command-line/shell.html#am) command. Here's an example:

```
adb [-d|-e|-s <serialNumber>] shell am start -S -W
com.example.app/.MainActivity
-c android.intent.category.LAUNCHER
-a android.intent.action.MAIN
```

The `Displayed` metric appears in the logcat output as before. Your terminal window should also display the following:

```
Starting: Intent
```

```
Activity: com.example.app/.MainActivity
ThisTime: 2044
TotalTime: 2044
WaitTime: 2054
Complete
```

The `-c` and `-a` arguments are optional and let you specify `<category>` (https://developer.android.com/guide/topics/manifest/category-element.html) and `<action>` (https://developer.android.com/guide/topics/manifest/action-element.html) for the intent.

Time to full display

You can use the `reportFullyDrawn()` (https://developer.android.com/reference/android/app/Activity.html#reportFullyDrawn()) method to measure the elapsed time between application launch and complete display of all resources and view hierarchies. This can be valuable in cases where an app performs lazy loading. In lazy loading, an app does not block the initial drawing of the window, but instead asynchronously loads resources and updates the view hierarchy.

If, due to lazy loading, an app's initial display does not include all resources, you might consider the completed loading and display of all resources and views as a separate metric: For example, your UI might be fully loaded, with some text drawn, but not yet display images that the app must fetch from the network.

To address this concern, you can manually call `reportFullyDrawn()` (https://developer.android.com/reference/android/app/Activity.html#reportFullyDrawn()) to let the system know that your activity is finished with its lazy loading. When you use this method, the value that logcat displays is the time elapsed from the creation of the application object to the moment `reportFullyDrawn()` (https://developer.android.com/reference/android/app/Activity.html#reportFullyDrawn()) is called. Here's an example of the logcat output:

```
system_process I/ActivityManager: Fully drawn {package}/.MainActivity: +1s54ms
```

If you learn that your display times are slower than you'd like, you can go on to try to identify the bottlenecks in the startup process.

Identifying bottlenecks

Two good ways to look for bottlenecks are Android Studio's Method Tracer tool and inline tracing. To learn about Method Tracer, see that tool's documentation (https://developer.android.com/studio/profile/am-methodtrace.html).

If you do not have access to the Method Tracer tool, or cannot start the tool at the correct time to gain log information, you can gain similar insight through inline tracing inside of your apps' and activities' `onCreate()` methods. To learn about inline tracing, see the reference documentation for the `Trace` (https://developer.android.com/reference/android/os/Trace.html) functions, and for the Systrace (https://developer.android.com/studio/profile/systrace-commandline.html) tool.

Common Issues

This section discusses several issues that often affect apps' startup performance. These issues chiefly concern initializing app and activity objects, as well as the loading of screens.

Heavy app initialization

Launch performance can suffer when your code overrides the `Application` object, and executes heavy work or complex logic when initializing that object. Your app may waste time during startup if your Application subclasses perform initializations that don't need to be done yet. Some initializations may be completely unnecessary: for example, initializing state information for the main activity, when the app has actually started up in response to an intent. With an intent, the app uses only a subset of the previously initialized state data.

Other challenges during app initialization include garbage-collection events that are impactful or numerous, or disk I/O happening concurrently with initialization, further blocking the initialization process. Garbage collection is especially a consideration with the Dalvik runtime; the Art runtime performs garbage collection concurrently, minimizing that operation's impact.

Diagnosing the problem

You can use method tracing or inline tracing to try to diagnose the problem.

Method tracing

Running the Method Tracer tool reveals that the `callApplicationOnCreate()` (https://developer.android.com/reference/android/app/Instrumentation.html#callApplicationOnCreate(android.app.Application)) method eventually calls your `com.example.customApplication.onCreate` method. If the tool shows that these methods are taking a long time to finish executing, you should explore further to see what work is occurring there.

Inline tracing

Use inline tracing to investigate likely culprits including:

- Your app's initial `onCreate()` (https://developer.android.com/reference/android/app/Application.html#onCreate()) function.
- Any global singleton objects your app initializes.
- Any disk I/O, deserialization, or tight loops that might be occurring during the bottleneck.

Solutions to the problem

Whether the problem lies with unnecessary initializations or disk I/O, the solution calls for lazy-initializing objects: initializing only those objects that are immediately needed. For example, rather than creating global static objects, instead, move to a singleton pattern, where the app initalizes objects only the first time it accesses them. Also, consider using a dependency injection framework like Dagger (http://google.github.io/dagger/) that creates objects and dependencies are when they are injected for the first time.

Heavy activity initialization

Activity creation often entails a lot of high-overhead work. Often, there are opportunities to optimize this work to achieve performance improvements. Such common issues include:

- Inflating large or complex layouts.
- Blocking screen drawing on disk, or network I/O.
- Loading and decoding bitmaps.
- Rasterizing `VectorDrawable` (https://developer.android.com/reference/android/graphics/drawable/VectorDrawable.html) objects.
- Initialization of other subsystems of the activity.

Diagnosing the problem

In this case, as well, both method tracing and inline tracing can prove useful.

Method tracing

When running the Method Tracer tool, the particular areas to focus on your your app's `Application` (https://developer.android.com/reference/android/app/Application.html) subclass constructors and `com.example.customApplication.onCreate()` methods.

If the tool shows that these methods are taking a long time to finish executing, you should explore further to see what work is occurring there.

Inline tracing

Use inline tracing to investigate likely culprits including:

- Your app's initial `onCreate()` (https://developer.android.com/reference/android/app/Application.html#onCreate()) function.
- Any global singleton objects it initializes.
- Any disk I/O, deserialization, or tight loops that might be occurring during the bottleneck.

Solutions to the problem

There are many potential bottlenecks, but two common problems and remedies are as follows:

- The larger your view hierarchy, the more time the app takes to inflate it. Two steps you can take to address this issue are:
  - Flattening your view hierarchy by reducing redundant or nested layouts.

  - Not inflating parts of the UI that do not need to be visible during launch. Instead, use use a `ViewStub` (https://developer.android.com/reference/android/view/ViewStub.html) object as a placeholder for sub-hierarchies that the app can inflate at a more appropriate time.

- Having all of your resource initialization on the main thread can also slow down startup. You can address this issue as follows:
  - Move all resource initialization so that the app can perform it lazily on a different thread.
  - Allow the app to load and display your views, and then later update visual properties that are dependent on bitmaps and other resources.

Themed launch screens

You may wish to theme your app's loading experience, so that the app's launch screen is thematically consistent with the rest of the app, instead of with the system theming. Doing so can hide a slow activity launch.

A common way to implement a themed launch screen is to use the `windowDisablePreview` (https://developer.android.com/reference/android/R.attr.html#windowDisablePreview) theme attribute to turn off the initial blank screen that the system process draws when launching the app. However, this approach can result in a longer startup time than apps that don't suppress the preview window. Also, it forces the user to wait with no feedback while the activity launches, making them wonder if the app is functioning properly.

Diagnosing the problem

You can often diagnose this problem by observing a slow response when a user launches your app. In such a case, the screen may seem to be frozen, or to have stopped responding to input.

Solutions to the problem

We recommend that, rather than disabling the preview window, you follow the common Material Design (http://www.google.com/design/spec/patterns/launch-screens.html#) patterns. You can use the activity's `windowBackground` theme attribute to provide a simple custom drawable for the starting activity.

For example, you might create a new drawable file and reference it from the layout XML and app manifest file as follows:

Layout XML file:

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android" android:opacity="opa
  <!-- The background color, preferably the same as your normal theme -->
  <item android:drawable="@android:color/white"/>
  <!-- Your product logo - 144dp color version of your app icon -->
  <item>
    <bitmap
      android:src="@drawable/product_logo_144dp"
      android:gravity="center"/>
  </item>
</layer-list>
```

Manifest file:

```
<activity ...
android:theme="@style/AppTheme.Launcher" />
```

The easiest way to transition back to your normal theme is to call `setTheme(R.style.AppTheme)` (https://developer.android.com/reference/android/view/ContextThemeWrapper.html#setTheme(int)) before calling `super.onCreate()` and `setContentView()`:

```
public class MyMainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // Make sure this is before calling super.onCreate
    setTheme(R.style.Theme_MyApp);
    super.onCreate(savedInstanceState);
    // ...
  }
}
```