Guava and Android

Background

Once upon a time, use of Guava was generally frowned upon in Android apps developed at Google, as Guava was designed, tested, benchmarked, and optimized for the conditions found in server Java development. This situation was suboptimal: Android developers found themselves reinventing wheels, awkwardly working around API gaps, and their best practices diverged from best practices for other Java development.

Over 2017, we worked extensively to change this situation: to optimize a branch of Guava for suitability for Android, in collaboration with Google's Android experts. We converged on the following set of priorities:

- Do not compromise on API design principles. Maintain API compatibility between the Android and mainline branches.
- Optimize for code size *after applying ProGuard shrinking.* Most Android apps need to prioritize APK size, but for apps where this is a concern, they should already be using ProGuard. For simplicity, we measured shrunk JAR size in bytes, classes, and methods, after applying Google's internal version of ProGuard, but most code shrinking tools should show similar results.
- Optimize for minimal allocation of objects and bytes. Garbage collection is often more expensive on Android.
- While some sacrifice to constant factors in CPU performance is tolerable, avoid asymptotic slowdowns.

As a result of this work, Google's internal Android best practices have gone from forbidding the use of Guava to actively recommending key parts of Guava.

Specifics

Even after this work, not all of Guava is recommended for use on Android. We focused our optimization work on some key data structures and APIs. Loosely, we recommend against the following:

- `cache` (prefer `LruCache` instead)
- `eventbus` (uses reflection)
- `reflect` (uses reflection)
- `graph` (uses expensive data structures, usually not worth it)
- `collect.Table` (uses expensive data structures, usually not worth it)

Details

This work consisted of lots of nitty gritty optimization work, but we can pick out some high points. First, we made significant strides on reducing code size footprint of many types, with most of that effort focused on `common.collect`.

The below chart shows the before and after JAR sizes of several key APIs, with all measurements taken after appropriate ProGuard stripping.

JAR sizes

We found many tricks for reducing code size, but the most important were

- removing unnecessary use of skeleton classes
- making it easier for ProGuard to eliminate dead code
- reducing static constants that could not be inlined, or moving them to helper classes

This work often went hand in hand with rewriting of underlying data structures, especially in `common.collect`. Our primary goals for our data structures were:

- O(1) total objects in the "steady state" data structure (retained if you keep a reference to the collection itself)
- Asymptotics equivalent to the appropriate equivalent mutable data structures (`LinkedHashMap`, etc.)
- Constant factors for memory consumption competitive with best-of-breed Android data structures (`ArrayMap`, `ArraySet`, even though those data structures gave up O(1) asymptotics)
- Share objects where possible with the builder, to reduce garbage allocation

This required rewriting many of our core data structures in the Android fork, especially requiring us to avoid entry objects wherever possible. For example, `ImmutableMap` on Android is now backed by a flattened hash table built of parallel arrays, with the hash table storing indexes into another array rather than storing pointers to entry objects. Similarly, we developed algorithmic tweaks allowing `ImmutableSortedMap` to sort the keys and values together without combining them into `Entry` objects as would typically be required.

Generally speaking, we did not achieve data structures that were *as* compact as Android's `ArrayMap` and `ArraySet`, but we came close while still providing the O(1) asymptotics we expect from these structures.

Set memory consumption

Map memory consumption

Other data structures, unique to Guava, showed similar improvements even if they did not have competition. For example:

Data structure	Bytes/entry (before)	Bytes/entry (after)
`HashBiMap`	64	40
`HashMultiset`	59	24
`LinkedHashMultiset`	67	32

We did, in some cases, accept asymptotic slowdowns to view collections where workarounds were easily available. For example, few users know – or care –

that `ImmutableSet.asList()` returned a "magical" `ImmutableList` with an O(1) `contains` implementation. Eliminating the specialized implementation in the Android branch allowed us to make significant code size savings, and the workaround of just calling `contains` on the `ImmutableSet` directly is easy enough.