Overview

The Flutter Engine in AOT mode requires four artifacts to run any given isolate. These are:

- **Dart VM Snapshot**
  - Represents the initial state of the Dart heap shared between isolates.
  - Helps launch Dart isolates faster.
  - Does not contain any isolate specific information.
  - Mostly predefined Dart strings used by the VM
  - Should live in the data segment.
    - From the VM's perspective, this just needs to be loaded in memory with READ permissions and does not need WRITE or EXECUTE permissions. Practically this means it should end up in rodata when putting the snapshot in a shared library.
- **Dart VM Instructions**
  - Contains AOT instructions for common routines shared between all Dart isolates in the VM.
  - This snapshot is typically extremely small and mostly contains stubs.
  - Must live in the text segment.
- **Isolate Snapshot**
  - Represents the initial state of the Dart heap and includes isolate specific information.
  - Along with the VM snapshot, helps in faster launches of the specific isolate.
  - Should live in the data segment.
- **Isolate Instructions**
  - Contains the AOT code that is executed by the Dart isolate.
  - Must live in the text segment.

The VM snapshot and instructions can be shared between all isolates. These must be available during the initialization of the Dart VM. The Dart VM is initialized when the first Flutter view initializes an instance of the Flutter shell. The Flutter shell manages thread safe initialization of the Dart VM. Once the Dart VM is initialized, multiple instances of the Flutter view reference the same VM to run their isolates. There can only be one VM running in the process at any given time.

Given this configuration, launching each root isolate (for a Flutter application) requires two artifacts along with two other artifacts that are shared between all the other isolates.

Isolates launched by Dart code (e.g., Isolate.spawn) inherit the snapshots of their parents.

Snapshot Generation

These artifacts on all platforms are generated by the same binary on the host. This binary is shipped with Flutter tools and is called `gen_snapshot`. The way these artifacts are packed and referenced on the device differs however.

iOS Configuration

`gen_snapshot` is invoked on the host by Xcode to generate the four binary blobs for each artifact.

Using the native toolchain in Xcode, these artifacts are compiled into a framework bundle that is packaged into the application. This framework is typically called `App.framework` and is present in the `Frameworks/` folder of the application bundle. The name of the framework is configurable and if the embedder wishes to name it anything

other than `App.framework`, the embedder can choose a custom name and specify the same in the `FLTLibraryPath` `Info.plist` key for the main application bundle.

The Flutter engine, itself in a Framework called `Flutter.framework` will dynamically open the resolved application framework and look for four specific symbols in that library. These symbols are called `kDartVmSnapshotData`, `kDartVmSnapshotInstructions`, `kDartIsolateSnapshotData` and `kDartIsolateSnapshotInstructions`. These refer to the four snapshots mentioned above.

Once snapshot resolution is finalized, the Flutter engine initializes the VM and launched the isolates.

This configuration was chosen because the Flutter engine is not able to mark pages as executable at runtime. Packaging the instruction in a binary makes sure all instructions are present in a separate dynamic library. This arrangement helps in isolate of code for a specific Flutter view's root isolate.

Android Configuration

`gen_snapshot` is invoked by Gradle to generate the four binary blobs for each artifact.

Gradle actually invokes `flutter build aot` under the hood to generate these artifacts. These binary artifacts are packaged into the APK directly. These artifacts are named `vm_snapshot_data`, `vm_snapshot_instr`, `isolate_snapshot_data` and `isolate_snapshot_instr`. They refer to the four snapshots mentioned above.

The embedder may choose to put these artifacts in custom locations. In such cases, the embedder must place these artifacts in a directory and specify the location using the following 5 flags:

- `aot-snapshot-path`: Path to the directory in the APK assets containing the snapshot.
- `vm-snapshot-data`: Path to the VM snapshot in that directory.
- `vm-snapshot-instr`: Path to the VM instructions in that directory.
- `isolate-snapshot-data`: Path to the isolate snapshot in that directory.
- `isolate-snapshot-instr`: Path to the isolate instructions in that directory.

Once the Flutter engine, itself packaged as a separate dynamic library called `libflutter.so`, resolves the locations of the required artifacts, it maps them in and makes sure the instructions are executable before launching the VM and isolate.

The tooling on Android does not require the presence of the Android NDK because the engine can mark pages as executable at runtime. If the embedder wishes to package the snapshots into a single dynamic library (and make the configuration look like the one used on iOS), it can do so and specify the library to the engine via the flag `aot-shared-library-path`. A native toolchain is necessary on the host to achieve this.

Notes

`gen_snapshot` can only generate AOT instructions for a specific architecture. For example, generating AOT instructions for `armv7`, `aarch64`, `i386` and `x86-64` requires four different `gen_snapshot` variants.

The VM and isolate snapshot data is specific to the `gen_snapshot` variant used and must also be generated along with the AOT instructions. Embedders may not mix and match the AOT instructions and data snapshots from different `gen_snapshot` variants. All four artifacts must typically be generated at the same time.

Using the current Flutter tools, only an `i386` `gen_snapshot` on the host can generate `armv7` AOT instructions and an `x86-64` `gen_snapshot` on the host can generate `aarch64` instructions. The mac builds take

advantage of this quirk and `lipo` the two `gen_snapshot` variants and select the executable architecture correctly when generating AOT instructions for the target architecture. This can be inspected with the `lipo` tool on the host while analyzing the `gen_snapshot` binary. In reality, the word sizes of `gen_snapshot` on host and the engine running on the target must match. Theoretically, an `armv7` `gen_snapshot` running on a Raspberry Pi host could generate an AOT snapshot for `armv7` target. The tools don't ship this configuration however.

Packaging multiple Flutter AOT application in the same application bundle will currently result in redundant copies of the VM data and instructions. However, these buffers are extremely small.

Flags to the Flutter engine are typically specified when the platform launches the underlying Flutter shell (platform agnostic way of interacting with the innards of the Flutter engine). This happens in `FlutterViewController` on iOS and `FlutterNativeView` in Java on Android.

To list all the flags currently supported by the Flutter engine, embedders may locate the `flutter_tester` binary in the tools distribution and pass the `--help` flag to the same. All currently supported flags along with a short help blob will be dumped to the console.