Ch 1. Dart & Flutter 컴파일의 시작(JIT와 AOT)

우리는 Dart라는 언어로 Flutter 프레임워크를 활용해 앱 코드를 작성하고, 디버깅하거나 빌드하여 각 환경에 맞게 실행합니다. 막연히 Flutter는 크로스 플랫폼을 지원하는 프레임워크라 알고 있을텐데, 어떻게 Dart 코드가 각 네이티브 환경에서 실행되는지 제대로 알지 못하고 있습니다. 이번 글에서는 Dart 코드가 어떻게 여러 플랫폼에서 실행되는지 전반적인 이해를 다루고, Dart VM의 핵심 개념인 JIT과 AOT에 대해 알아보겠습니다.

Dart VM이란?

Dart 언어는 다른 언어들과 비슷하게 컴파일 과정을 거쳐 우리의 코드를 기계가 이해할 수준의 언어로 변환합니다. 그렇다고 단순히 컴파일 과정만을 거쳐서 우리가 원하는 앱이 되진 않습니다. 크게 보면 Dart 언어로 작성된 코드를 컴파일하는 단계, 그리고 컴파일된 기계어를 바탕으로 각 플랫폼 별 엔진을 거쳐 빌드되는 단계로 나눌 수 있습니다.

Dart VM은 Dart 코드를 각 플랫폼에서 네이티브로 실행하는 하나의 큰 시스템입니다. 여기엔 다양한 런타임 시스템, 객체 모델, 가비지 컬렉션, 스냅샷, 컴파일 파이프라인 등 여러 요소들이 있습니다.

Dart VM은 여러 방식으로 코드 실행을 지원합니다.

> 참고 문헌 https://mrale.ph/dartvm/ >
> https://medium.com/@howyoujini/%EC%BD%94%EB%93%9C%EA%B0%80-%EC%96%B4%EB%96%BB%EA%B2%8C-%ED%99%94%EB%A9%B4%EC%9D%B4-%EB%90%98%EB%82%98-1-aot-vs-jit-compilation-with-flutter-19af2b5f8d51

====

일단 모든 시작에 앞서 Dart의 영역과 Flutter의 영역을 구분할 수 있어야 합니다.

Dart는 기본적으로 **컴파일 언어**이며, Dart 코드를 실행하는 **Dart VM(Virtual Machine)**은 런타임 시스템, 객체 모델, 가비지 컬렉션, 스냅샷, JIT(Just-in-Time) 및 AOT(Ahead-of-Time) 컴파일 파이프라인, 인터프리터, 그리고 ARM 시뮬레이터와 같은 다양한 구성 요소를 포함합니다. 예를 들어, Dart 코드는 Dart VM의 AOT 파이프라인을 사용하여 기계 코드로 컴파일된 다음, 컴파일러 구성 요소가 없고 Dart 소스 코드를 동적으로 로드할 수 없는 축소된 버전의 Dart VM인 **'precompiled runtime'** 내에서 실행될 수 있습니다.

이렇게 컴파일된 Dart 코드는 **Flutter 엔진**에 전달되며, 엔진은 이를 렌더링하여 앱으로 보여줍니다. Flutter 아키텍처는 크게 세 개의 레이어로 구성됩니다. Dart를 기반으로 하는 **프레임워크(Framework)**, C++를 기반으로 하는 **엔진(Engine)**, 그리고 각 플랫폼에 Flutter를 통합하는 **임베더(Embedder)**입니다. 엔진은 Skia/Impeller, Dart 런타임, 가비지 컬렉션(GC), 그리고 JIT/AOT 컴파일러와 같은 핵심 구성 요소를 포함합니다. 임베더는 iOS와 같은 플랫폼에 "캔버스"를 제공하여, Flutter가 플랫폼 관련성이 낮은 레이어에서 대부분의 렌더링 작업을 수행하게 하여 뛰어난 크로스-터미널 일관성을 제공합니다.

이 중에서 컴파일 절차에 대해 먼저 알아보겠습니다.

컴파일 방식은 크게 두 가지로 나뉩니다: **AOT(Ahead-of-Time) 컴파일**과 **JIT(Just-in-Time) 컴파일**. **AOT는 주로 릴리즈 모드(Release Mode)**에서 동작하는 컴파일 방식이며, **JIT는 디버깅 모드(Debug Mode)**에서 동작하는 컴파일 방식입니다. 이 둘의 차이는 컴파일러와 인터프리터 정도의 차이를 만들어냅니다.

JIT (Just-in-Time) 컴파일

JIT는 `frontend_server`라는 프로세스를 통해 Dart 컴파일러가 Dart 코드를 Kernel AST로 변환합니다. 이 과정은 개발자 PC(호스트)에서 이루어집니다. 이렇게 변환된 **Kernel 바이너리는 기기에 전송되어 스크립트 스냅샷(`snapshot_blob.bin`) 형태로 변환되고, 런타임에 실행 가능한 머신 코드로 컴파일됩니다**.

이 덕분에 **핫 리로드(Hot Reload)**가 가능합니다. `frontend_server` 프로세스가 지속적으로 실행되면서 이전 컴파일의 CFE(Common Front-End) 상태를 재사용하고, 변경된 라이브러리만 재컴파일하여 빠른 개발 경험을 제공합니다.

JIT 컴파일은 개발 모드에 최적화되어 있으며, assertion 및 모든 디버깅 정보, 서비스 확장, Observatory와 같은 디버깅 도구를 포함합니다. 실행 속도나 패킷 크기에 최적화되어 있지 않습니다.

JIT 컴파일러는 **동적 타입 피드백**을 사용하여 최적화를 수행합니다. 실행 중인 프로그램의 프로필을 기반으로 최적화 결정을 내리는데, 이는 인라인 캐싱(inline caching)을 통해 콜사이트에서 관찰된 리시버 타입 정보를 수집함으로써 이루어집니다. 함수의 실행 카운터가 특정 임계값에 도달하면, 해당 함수는 **백그라운드 최적화 컴파일러**로 전송되어 최적화됩니다.

초기에는 함수 본문에 실행 가능한 코드 대신 **`LazyCompileStub`라는 플레이스홀더**가 있습니다. 함수가 처음 호출될 때, `LazyCompileStub`가 런타임 시스템에 해당 함수에 대한 실행 가능한 코드를 생성하도록 요청하고, 생성된 코드를 호출합니다. 이 초기 컴파일은 **비최적화 컴파일러(unoptimizing compiler)**에 의해 수행됩니다. 이는 빠르지만 최적화는 이루어지지 않습니다.

JIT는 **'AppJIT 스냅샷'**을 통해 대규모 Dart 애플리케이션(예: `dartanalyzer` 또는 `dart2js`)의 **워밍업 시간을 단축**할 수 있습니다. 애플리케이션을 특정 '훈련 데이터'로 실행한 후 생성된 모든 코드와 VM 내부 데이터 구조를 AppJIT 스냅샷으로 직렬화하여 배포할 수 있습니다. 이를 통해 VM은 사용자 코드를 더 빨리 실행할 수 있습니다. 일반적으로 **JIT는 AOT보다 더 나은 최고 성능을 달성할 수 있습니다**, 특히 훈련 데이터가 좋거나 런타임 타입 정보가 최적화에 유리할 때 그렇습니다.

만약 최적화된 코드가 가정한 추측이 런타임에 위반되면, **역최적화(deoptimization)** 과정을 거쳐 실행을 최적화되지 않은 함수의 해당 지점으로 전환하고, 나중에 업데이트된 타입 피드백을 사용하여 다시 최적화합니다.

AOT (Ahead-of-Time) 컴파일

AOT에서는 `gen_snapshot`이라는 Dart 컴파일러가 Kernel AST를 네이티브 ARM 머신 코드로 직접 컴파일합니다. 이 과정에서 **트리 쉐이킹(Tree Shaking)**과 같은 최적화 프로세스가 동작합니다. 트리 쉐이킹은 앱에 의해 임포트되었지만 사용되지 않는 모든 코드를 컴파일된 앱 바이너리에서 제거하여 앱의 크기를 줄이는 역할을 합니다. 이 `gen_snapshot` 도구는 최소한의 패킷을 생성하고 Dart의 리플렉션 기능을 비활성화합니다.

AOT 컴파일은 약 1년 반 전에 Dart를 **iOS에서 효율적으로 실행하기 위해 시작**되었습니다. 당시 JIT 컴파일러는 Android에서만 작동했고, iOS의 제약 조건 때문에 JIT 컴파일을 직접 사용할 수 없었기 때문입니다. Flutter는 처음에는 iOS에서 ARM 시뮬레이터를 사용했지만, 이는 너무 느려서 앱을 배포하는 옵션이 아니었습니다. 따라서 AOT 컴파일러 개발이 시작되었고, 처음에는 Intel 용으로 프로토타입을 만든 후 ARM 및 ARM64로 포팅하여 모바일 지원을 가능하게 했습니다.

AOT 컴파일된 코드는 **iOS뿐만 아니라 Android에서도 훨씬 빠르게 시작됩니다**. 이는 워밍업(warmup) 비용이 없기 때문입니다. 코드가 바이너리에서 메모리로 매핑되므로 JIT에서 시작을 늦추는 코드 생성이 필요 없습니다.

AOT 컴파일러는 JIT 컴파일러와 **많은 구성 요소를 공유**합니다. 예를 들어, 파서, 많은 컴파일 최적화(레지스터 할당 등), 코드 생성기, 가비지 컬렉터 등이 그렇습니다. 하지만 AOT 컴파일에만 독점적으로 필요한 새로운 구

성 요소들도 추가되었습니다. 여기에는 **트리 쉐이킹**, **AOT-특정 컴파일러 패스** (코드 속도 향상), **새로운 스냅샷 읽기/쓰기** (앱 시작 시 데이터 초기화 속도 향상), 그리고 iOS에서 정규 표현식을 실행하기 위한 **인터프리트 버전** 등이 포함됩니다.

AOT 컴파일의 주요 과제 중 하나는 **원본 Dart VM이 AOT 컴파일된 환경에서 실행되도록 설계되지 않았다는 점**입니다. 따라서 컴파일된 코드가 **재배치 가능(relocatable)**하고 **격리 독립적(isolate-independent)**이 되도록 만들어야 했습니다. AOT 환경에서는 **힙의 코드 부분이 모든 Isolate 간에 공유됩니다**. 이는 코드가 바이너리로 컴파일되어 하나만 존재하기 때문입니다. 반면 JIT 환경에서는 각 Isolate가 고유한 코드 부분을 가집니다.

AOT 컴파일은 **'닫힌 세계(closed world)' 가정을 기반으로 합니다**. 이는 특정 언어 기능에 제한을 둠으로써 달성됩니다. 예를 들어, 실제 **지연 로딩(deferred loading)**이나 `Isolate.spawnUri`**를 통한 새로운 코드 스폰이 지원되지 않습니다**. 또한 **미러(mirrors) 기능도 지원되지 않습니다**. 이러한 제한은 주로 코드 크기 최적화 때문입니다. AOT는 JIT처럼 동적 타입 피드백(dynamic type feedback)을 활용할 수 없으므로, 컴파일러가 스스로 타입을 추론해야 합니다. 모든 함수는 **전역 정적 분석(Type Flow Analysis, TFA)**을 통해 도달 가능성이 판단되며, 추측적(speculative) 최적화 없이 네이티브 코드로 컴파일됩니다.

iOS에서 `App.framework`는 `kDartVmSnapshotdData`, `kDartVmSnapshotInstructions`, `kDartIsolateSnapshotData`, `kDartIsolateSnapshotInstructions`의 네 부분으로 구성됩니다. iOS 시스템 제한으로 인해 런타임에 메모리 페이지를 실행 가능으로 표시할 수 없기 때문에 이러한 구조를 사용합니다. 반면 Android의 릴리즈 모드에서는 `vm/isolate_snapshot_data/instr`와 같은 ARM 명령어가 런타임에 엔진에 의해 로드되고 실행 가능으로 표시됩니다.

===

주제 : Flutter의 매력, Hot Reload에 대한 정확한 이해

(https://theflutterist.medium.com/how-does-hot-reload-or-hot-restart-actually-works-an-insight-under-the-hood-da7c1a44bed2) (https://medium.com/podiihq/understanding-hot-reload-in-flutter-2dc28b317036)

- 요즘엔 핫 리로드라는 기능이 보편적이지만, Flutter가 처음 나온 2018년 쯤엔 혁신이었다.
- 이게 어떻게 되는걸까? 심지어 다양한 플랫폼에서 말이다.
- Dart VM의 컴파일 단계, Flutter 엔진이 각 플랫폼 별 앱으로 만들어주는 단계, 그리고 기기에 전달되는 과정까지를 정리해보자.

Dart VM의 컴파일 단계(JIT, AOT)

(https://medium.com/@howyoujini/%EC%BD%94%EB%93%9C%EA%B0%80-%EC%96%B4%EB%96%BB%EA%B2%8C-%ED%99%94%EB%A9%B4%EC%9D%B4-%EB%90%98%EB%82%98-1-aot-vs-jit-compilation-with-flutter-19af2b5f8d51)

- Dart에는 두개의 컴파일 방식이 있음 => JIT, AOT
- JIT는 debug 모드, AOT는 release/profile 모드에서 동작하는 컴파일 방식임 (https://docs.flutter.dev/perf/ui-performance#connect-to-a-physical-device)
- JIT : 개발중, 인터프리터 동시에 활용, 핫 리로드 기능 제공
- AOT : 릴리즈용, 최적화 프로세스 진행, 최종 바이너리 코드 컴파일

JIT (Just-In-Time)	AOT (Ahead-Of-Time)

	JIT (Just-In-Time)	AOT (Ahead-Of-Time)
언제 컴파일?	앱 **실행 중**에 필요한 함수만 즉석에서 기계어로 변환	**빌드 시** 전체 코드를 기계어로 변환
장점	- 컴파일이 부분적으로 일어나므로 **코드 변경 즉시 반영** 가능 - 디버거·DevTools에서 모든 메타데이터 사용 가능	- VM 없이 바로 실행 ⇒ **빠른 시작** - 코드 난독화·사이즈 최적화 용이 - **App Store 정책**(동적 코드 불가) 충족
단점	- 실행 중 코드 생성이 필요 ➜ iOS App Store 금지 - 첫 실행/스크롤 시 JIT 패널티("잰크")	- 코드 바뀌면 전체 재빌드 필요 - Hot Reload 불가
Flutter 모드	`flutter run` (Debug)	`flutter build ios --release`, TestFlight, App Store

핫 리로드가 동작하는 방식

1. flutter run

- 호스트(맥)에서 ▸ frontend_server(Dart incremental compiler) 프로세스 시작
- 디바이스(시뮬레이터/실기기)에서 ▸ Flutter Engine + Dart VM(JIT) 실행

2. 코드 저장

- IDE가 바뀐 .dart 파일 목록을 flutter tool에 알림

3. 증분 컴파일

- frontend_server가 Kernel IR (.dill) 패치를 수 ms 안에 생성

4. 패치 전송

- flutter tool이 DDK/DDS 프로토콜로 기기의 Dart VM에 전송

5. JIT 재컴파일 (디바이스)

- VM이 이미 로드된 클래스의 필드·메서드 정의만 교체 후 필요한 함수만 즉시 JIT

6. 프레임워크 재조립

- VM이 "코드 바뀜" 이벤트 발행 → Flutter Framework이 reassemble() 호출
- 현재 State는 유지한 채 위젯 트리만 다시 build
- 평균 200-400 ms 안에 UI 반영 → Hot Reload 완성

Hot Reload는 상태를 보존하지만, main() 시그니처 변경·const 제거 등 구조가 크게 바뀌면 Hot Restart가 필요합니다. Flutter 공식문서도 "VM에 코드를 주입해 클래스 정의만 갱신한다"고 설명합니다.

iOS에서는 원래 JIT이 안되어야 했는데 Flutter 팀이 취약점을 찾음

개발용(디버그) 프로비저닝 프로파일에는 get-task-allow 라는 디버그용 엔타이틀먼트가 들어갑니다. 이 플래그가 켜진 프로세스는 디버거가 붙어 있을 때 코드 서명 제한을 약간 풀어 줍니다. Flutter Debug 빌드가 포함한

Dart VM은 그 틈을 이용해 메모리 페이지 권한을 RW→RX로 바꿔 가며 JIT 코드를 생성합니다. 그렇기 때문에 시뮬레이터뿐 아니라 실제 기기에서도 Hot Reload가 가능했습니다.

근데 패치됨..

(https://github.com/flutter/flutter/issues/163984) (https://forum.itsallwidgets.com/t/ios-26-jit-and-flutter/3534)