# 안드로이드의 전반적인 개념

## -계층구조에 대하여

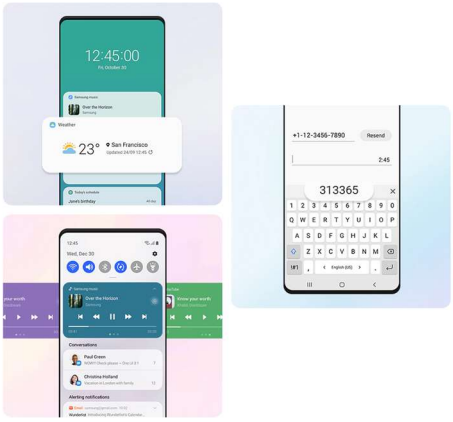
12171742 최진우

목차



## 기본 개념

안드로이드는 터치스크린 디바이스를 겨냥하여 리눅스 커널과 그 외 오픈소스들을 활용해서 개발된 모바일 운영체제이다. 무료 오픈소스로서 Android Open Source Project로 알려진 Apache License로 배포되고 있고 스마트폰 제조사들은 이를 커스터마이징 하여 사용자들에게 배포하게 된다.



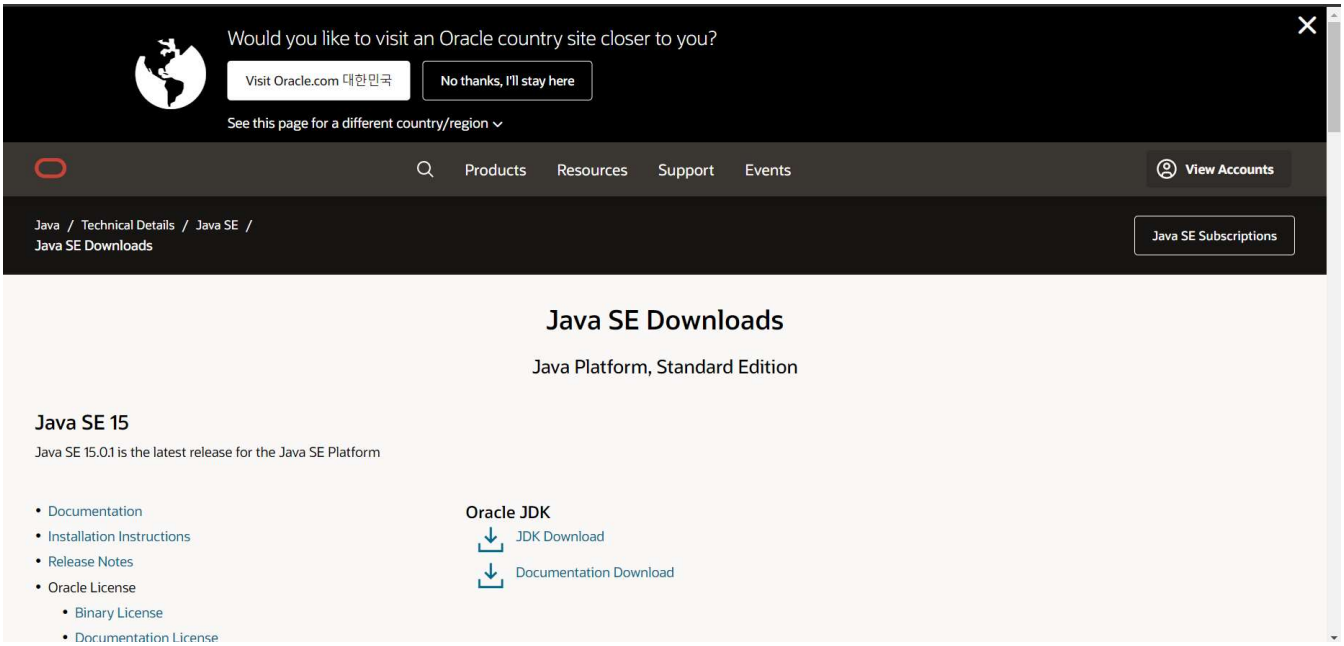

**삼성의 ONE UI (출처: samsung.com)**


**LG의 UX (출처: reddit.com)**

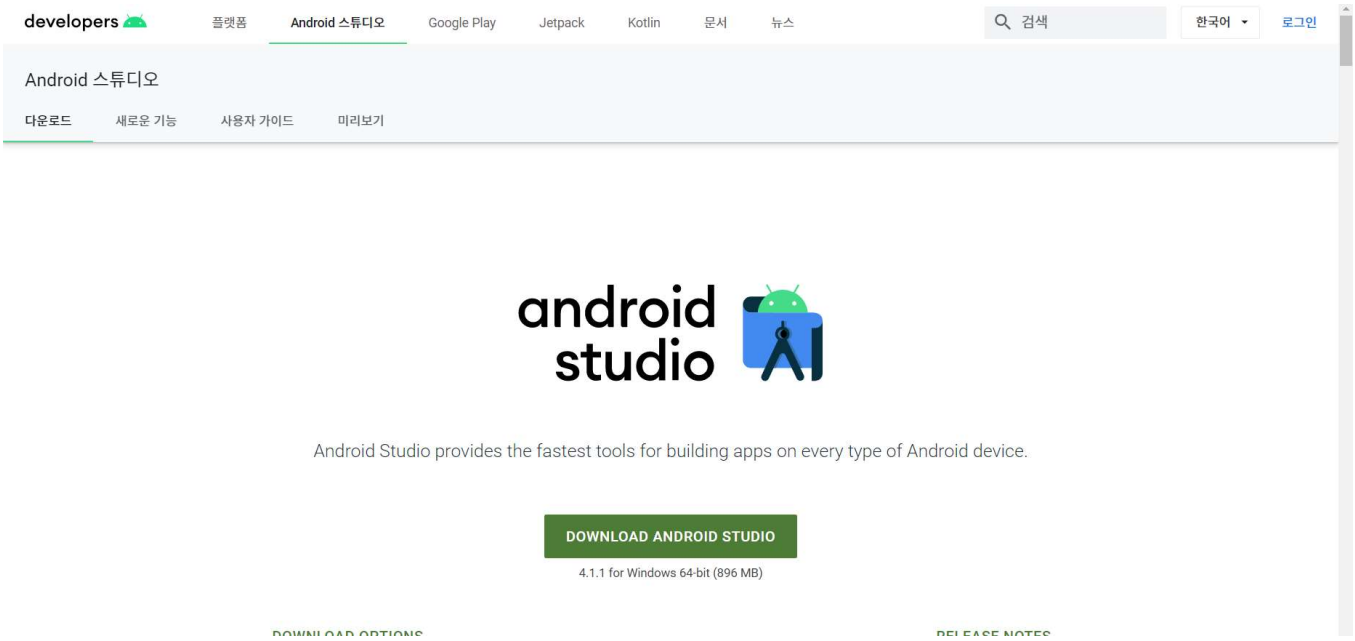
위의 예시들이 앞서 언급한 제조사들에 의해 커스터마이징 된 안드로이드이다. 이렇게 안드로이드 스마트폰에 적용되고 있는 소스 코드만 보았지만 이외에도 콘솔 게임 장치, 디지털카메라, PMP, 컴퓨터, 안드로이드TV, 스마트웨어 등등 많은 장치에 해당 용도에 특화된 사용자 인터페이스가 적용되어 탑재되고 있다. 이러한 범용성에 힘입어 안드로이드는 2011년부터 최고로 많이 팔리는 스마트폰 OS로 자리 잡았으며, 2013년부터는 태블릿 시장에서도 1위OS로 자리매김하였다. 안드로이드는 오픈소스이나 내부적으로는 구글 플레이, 크롬 등 구글 생태계의 모바일 서비스를 사용하게 되어있는데 이 덕분에 약 70% 이상의 안드로이드 탑재 제품에서 구글 서비스를 사용하고 있다. 안드로이드는 월간 보안 업데이트 및 연간 메이저 업데이트를 제공하는데 숫자 카운팅으로 메이저 업데이트 버전을 구분한다. 대외적으로는 안드로이드 2.3부터 시작하여 현재는 안드로이드 11버전까지 나온 상태이다.

# 개발환경 구성 방법

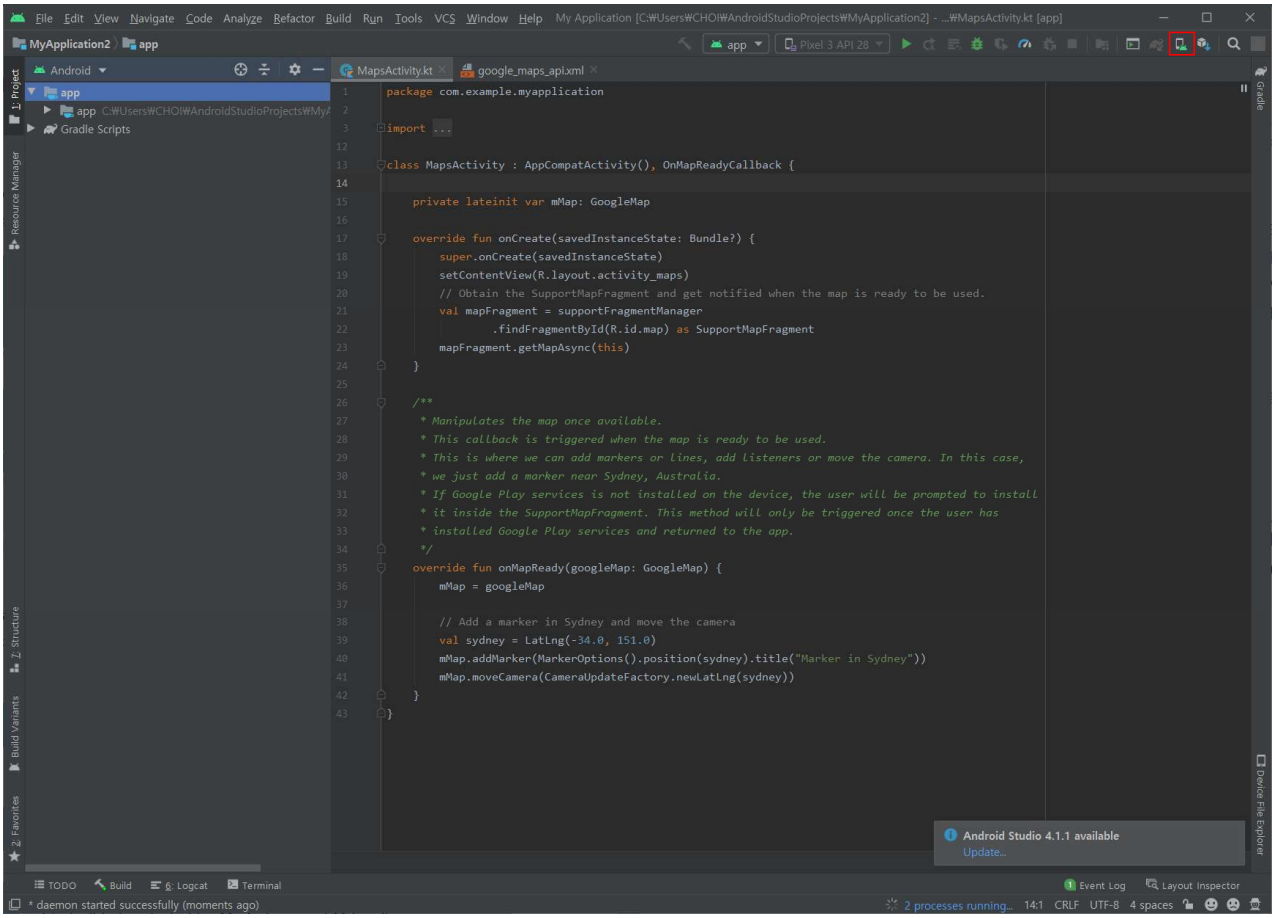
안드로이드는 Android Studio 라는 IDE를 통해 개발됩니다. 이때 java를 사용하여 개발하는데 이를 위해 JDK또한 필요하다. 아래는 간단한 개발환경 구성 방법을 단계별로 나타낸 것이다.



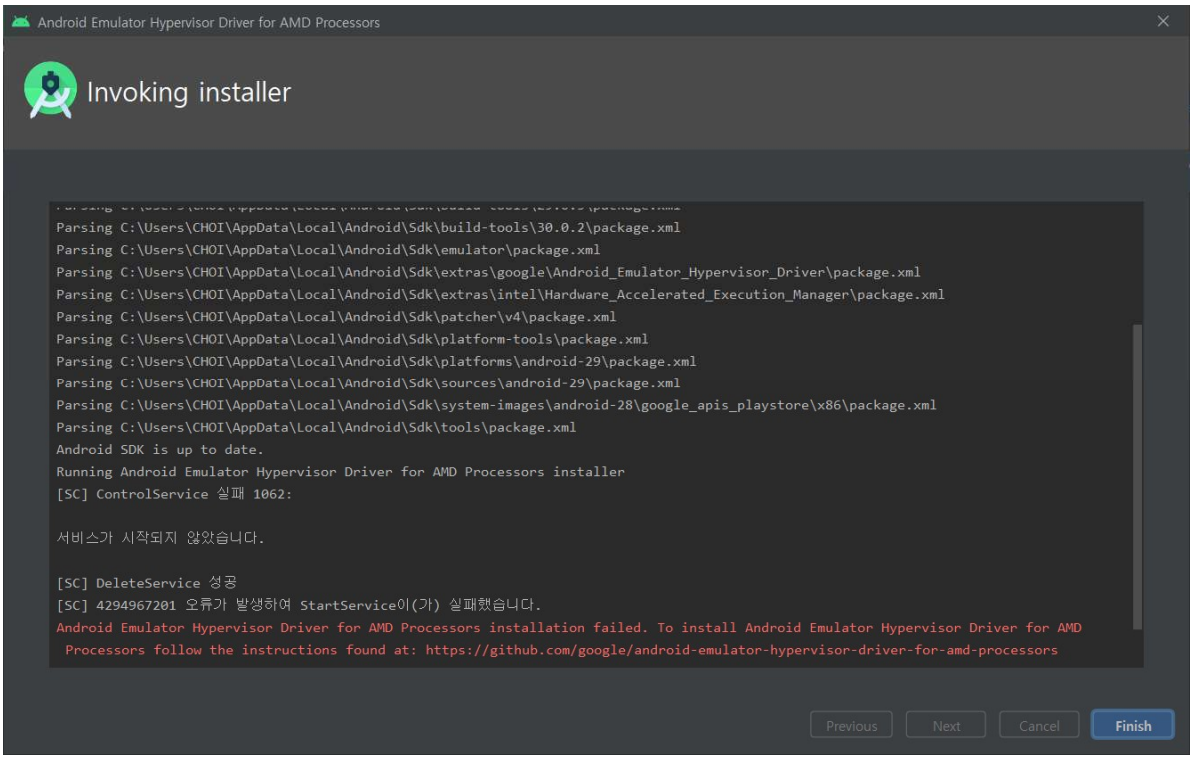

1. 앞서 설명한 대로 java를 사용하기 위해 oracle에서 JDK를 다운 후 설치한다.




2. 이후 android studio를 (https://developer.android.com/studio) 링크에서 다운 후 설치한다.




3. Android studio 실행 후 사용할 가상 디바이스 설정을 한다.

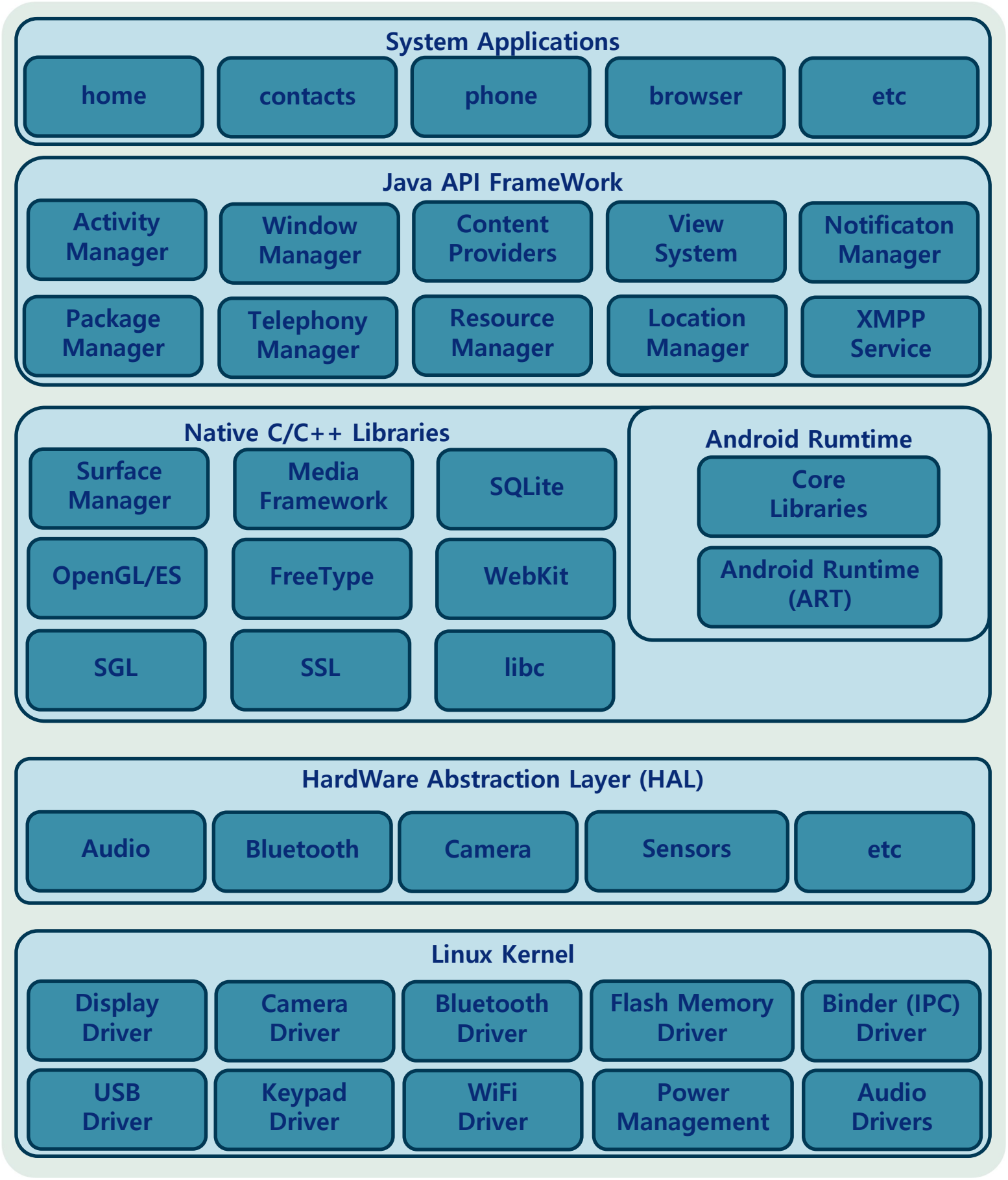



☞ 이때 AMD CPU의 경우 가상화 드라이버 오류가 발생하는데 bios에서 가상화 설정을 잡아 주어야 한다. (참고 https://friendcom.tistory.com/585)

이로서 간단한 개발 환경 설정이 끝나게 된다.

## 동작 방식

안드로이드 플랫폼은 다음과 같은 5계층으로 나뉘게 된다.



해당 다이어그램은 https://developer.android.com/guide/platform(안드로이드 공식 홈페이지)을 참조하였다. 이를 보면 리눅스 커널 계층, 네이티브 라이브러리 계층, 안드로이드 런타임 계층. 애플리케이션 프레임워크 계층, 애플리케이션 계층이 존재한다.

## 커널계층

안드로이드의 최하위 계층에 위치한다. 이러한 커널은 리눅스 기반으로 안드로이드 플랫폼의 기초를 담당하고 있다고 할 수 있다. 커널은 하드웨어와 소프트웨어를 연결해주는 역할을 수행한다. 예를 들어 Android Runtime 계층의 ART는 하위수준의 메모리 관리 등의 기본기능에 리눅스 커널을 사용한다. 이러한 특성에 따라 리눅스를 지원하는 다른 기기들 또한 안드로이드로 구동이 잠재적으로 가능하다고 할 수 있다. 하지만 리눅스에다가 디바이스 구동에 필요한 몇 가지 기능을 추가한 안드로이드는 리눅스와 약간씩 차이를 보인다. 이러한 차이는 커널 계층의 Power Management, Binder Driver에서 나타난다. Power Management는 항상 전원이 인가되어있는 기존의 리눅스 기기들과 다르게 배터리 기반으로 동작하는 안드로이드 기기들의 지원을 위해 새로 추가된 기능이다. 여기서는 대표적으로 두 가지 기능이 있는데 앱 대기 버킷, 배터리 세이버이다.


첫째로 앱 대기 버킷은 안드로이드 파이(9) 버전부터 추가된 배터리 관리기능으로 앱의 리소스 요청에 대해 우선순위를 부여하는 기능이다. 이러한 순위는 좌측의 그림과 같이 5가지의 버킷으로 나뉘는데 해당 애플리케이션이 특정 조건을 만족할 시 5개의 버킷 중 하나의 버킷으로 배치된다. 다음은 해당 버킷으로 들어갈 조건들을 보여준다.

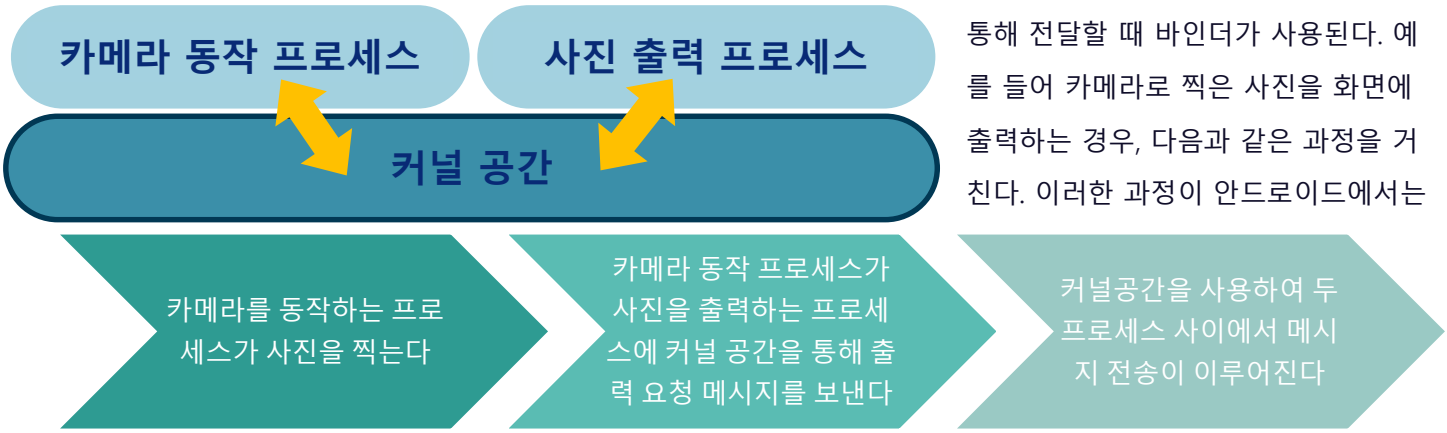
- **Active**
  - 앱이 작동을 시작할 때
  - 앱이 포그라운드 서비스를 실행 중 일 때
  - 앱에 포그라운드 앱과 동기화 되어있을 때
  - 사용자가 앱의 알림을 클릭할 때
- **Working Set**
  - 해당 앱을 사용자가 거의 매일 사용할 때
  - 앱이 간접적으로 사용될 때
- **Frequent**
  - 앱이 매일은 아니지만 정기적으로 사용될 때
- **Rare**
  - 앱이 자주 사용되지 않을 때
- **never**
  - 설치되었지만 한 번도 실행 되지 않을 때

Active bucket은 쉽게 말해 사용자가 해당 앱을 사용 중인 경우이다. 이 버킷에서는 앱의 작업, 알람 등의 모든 작업에 대해 시스템의 제한을 걸지 않다. Working Set은 자주 실행되지만, 현재는 활성 상태가 아닌 경우에 해당한다. 앱이 간접적으로 실행될 때 또한 Working Set으로 승격된다. 이 버킷에서는 작업의 경우 최대 2시간 연기될 수 있고, 알람의 경우 최대 6분 연기될 수 있다. Frequent는 사용자가 운동 시간에만 사용하는 운동 추적 앱과 같이 정기적으로 사용되는 경우이다. 여기서는 작업이 최대 8시간 연기될 수 있고 알람 또한 최대 30분 연기될 수 있다. Rare는 사용자가 숙박할 때만 사용하는 숙박 앱과 같이 특정 조건에서만 실행되고 자주 사용되지 않는 상황에 해당된다. 이때는 작업은 최대 24시간 연기되고 알람은 최대 2시간 연기될 수 있다. 또한 추가로 네트워크 연결 또한 최대 1일 제한될 수 있다. 마지막으로 never는 설치 이후 한 번도 실행하지 않은 경우이다. 이 경우 시스템에서 매우 엄격한 제한을 걸게 된다. 이러한 우선순위 버킷에는 시스템에 의해 앱이 동적으로 담기게 된다. 이때 제한은 기기가 배터리를 전원으로 사용 중일 때만 적용되며 충전기를 사용 중일 때는 시스템이 제한을 걸지 않는다. 둘째로 배터리 세이버는 기본적으로 시스템이 배터리 관리를 위해 앱을 특정 상황에서 제한하는 정책으로 순정 안드로이드인 AOSP에서는 대표적으로 다음과 같은 4가지의 전원 제한 정책을 가지고 있다.

- 시스템은 앱이 유휴 상태가 되는 것을 기다리는 것보다 적극적으로 앱을 대기모드로 만들어준다.
- 백그라운드 실행을 막는 것은 API 레벨에 상관없이 모든 앱에 적용된다.
- 화면이 꺼지면 위치 서비스를 비활성화하게 만들 수 있다.
- 백그라운드 앱은 네트워크 접근 권한이 없다.

이러한 조건들을 통해 배터리를 사용하는 기기들의 배터리 소모를 최적화하고 동작을 원활하게 만들 수 있다.

이제 Binder Driver(이하 바인더)에 대해 알아보자. 바인더는 기본적으로 IPC(네트워크상 각 프로세스들이 통신하는 형태 *Inter Process Communication)도구다. 하지만 안드로이드에서는 이런 IPC의 한 종류인 RPC(Remote Procedure Call)를 바인더로 사용한다. 여기서 리눅스 메모리 공간에 관해 이야기해야 한다. 안드로이드 프로세스는 해당 프로세스 고유의 주소에서 실행되고 이렇게 여러 프로세스들은 독립된 주소 공간에서 각자 별개로 동작하게 된다. 이러한 주소 공간은 사용자 공간과 커널 공간으로 나뉘게 된다. 사용자 공간에서는 사용자 코드와 관련 라이브러리가, 커널 공간에서는 커널 코드가 동작하게 된다. 이때 어떤 프로세스가 다른 프로세스로 데이터를 전달하기 위해서는 커널 공간을 통해야 한다. 왜냐하면 커널 공간은 프로세스들끼리 공유되기 때문이다. 이때 커널 입장에서 프로세스는 하나의 스레드일 뿐이고 커널 공간 내부의 데이터 등은 전역변수와 같이 모두에게 공유된다. 이렇게 데이터를 커널 공간을 통해 전달할 때 바인더가 사용된다. 예를 들어 카메라로 찍은 사진을 화면에 출력하는 경우, 다음과 같은 과정을 거친다. 이러한 과정이 안드로이드에서는




단순히 메시지를 전달하는 것이 아니라 상대방 프로세스에 존재하는 함수까지 호출하는 바인더를 사용하여 이루어진다. 이를 RPC방식 또는 메커니즘이라고 한다. 이러한 방식을 사용하여 앞에서 본 카메라 예시를 진행하면 카메라를 동작하는 프로세스가 사진을 출력하는 프로세스의 사진 출력함수를 커널 공간을 통해 호출하여 사진을 출력하게 된다. 이러한 동작이 Binder IPC Driver라는 추상화된 드라이버를 통해 커널 공간에서 이루어지는 것이다. 이러한 방식

은 하위계층인 커널 부분을 통해 데이터를 전달하기 때문에 데이터의 신뢰성과 보안성이 매우 우수하다. 이렇게 커널 계층에서 기본적인 리눅스 커널과 다른 부분을 일부 살펴보았다. 이외의 Display driver, WiFi driver 등은 기기에 맞는 드라이버를 커널 코드를 통해 구동시키는 리눅스의 기본적인 커널 부분과 같다.

## 하드웨어 추상화 계층

하드웨어 추상화 계층은 HAL로 불리기도 하는데 이는 Hardware Abstraction Layer의 약자로서, 컴퓨터의 하드웨어와 소프트웨어를 이어주는 역할을 하기 위해 그사이에 존재하는 추상화 계층이다. 이는 안드로이드에만 국한되는 것이 아니라 이외의 다른 OS나 기기에서 모두 사용되는 계층이다. 세상에는 수많은 하드웨어들이 있고 같은 성능의 제품들일지라도 안에 들어간 하드웨어의 종류가 다를 수 있다. 이런 상황에서 하드웨어를 구동시키기 위해서는 코드에서 하드웨어를 동작시키는 명령을 보내야 할 텐데 이때 하드웨어가 다르다면 같은 동작을 위해 여러 가지 종류의 하드웨어별로 코드를 따로 써줘야 한다. 이러한 작업은 프로그래머 입장에서 개발이 난해하고 코드 정리가 안되는 등의 불편함이 존재한다. 이때 필요한 것이 하드웨어 추상화인데 이는 같은 종류의 하드웨어들을 공통된 명령어 집합으로 묶어내는 과정이다. 좌측의 그림과 같이 기기에는 여러 회사의 오디오 부품이 들어가 있다. 이때 회사별 오디오 코드를 전부 제작한다면 몇백 개 회사들의 오디오마다 전부 작성을 하여야 함은 물론 개선품이나 신제품이 나올 때마다 코드를 수정하고 추가해줘야 한다. 이는 매우 힘든 일로 하나라도 지원을 하지 않는 부속이 들어갔다면 기기가 작동을 하지 않을 것이다. 이를 해결하기 위해 여러 오디오 제품들을 하나의 "오디오"로 인식하고 오디오에 대한 코드만 작성할 수 있는 방식을 찾았는데 이것이 하드웨어 추상화인 것이다. 하드웨어 추상화 계층은 이렇게 수많은 종류의 하드웨어들이 어느 코드에서든지 정상적으로 동작할 수 있게 해주고 사용자 코드 부분에서 하드웨어로의 직접적인 접근을 막아주어 오류를 줄여준다. 이를 통해 프로그래머 또한 하드웨어 종류에 구애받지 않고 하나의 코드를 통해 일관된 작업을 진행 할 수 있다. 프레임워크에서 기기의 하드웨어에 접근하기 위해 기기를 호출하면 하드웨어 추상화 계층에서 해당 하드웨어에 대한 라이브러리 모듈을 로드해준다. 안드로이드에서는 C와 C++로 작성이 되어있고 상위 수준의 Java와 하드웨어와의 통신을 담당하게 된다.


## Native Library / 안드로이드 런타임 계층

해당 계층에서는 안드로이드 런타임과 Native Library가 존재한다. 먼저 안드로이드 런타임부터 살펴보자. 안드로이드는 자바 기반 언어이다. 자바의 특성은 하드웨어에 구애받지 않고 코딩이 된다는 점이다. 일반적으로 다른 언어는 컴파일 시 기계어로 컴파일되어 직접 전달된다. 하지만 자바는 바이트 코드라는 것으로 컴파일 되는데 이는 JVM(Java Virtual Machine)을 통해 이루어진다. 이러한 바이트 코드로 컴파일된 결과에 JVM이 개입하여 해당 기기에 맞는 기계어로 번역시켜주므로 플랫폼에 구애를 받지 않고 실행이 되는 것이다. 안드로이드는 이러한 JVM을 라이선스 문제로 사용하지 못하고 대신 구글에서 개발한 Dalvik VM(Virtual Machine)을 사용하게 된다. 이러한 DVM은 JIT(Just In Time)컴파일러를 기반으로 하는데 JIT는 프로그램을 실행하는 시점에서 필요한 부분을 그때그때 가져와 바이트코드에서 기계어로 변환하여 캐싱 후 램에 올리는 방식으로 작동한다. 이러한 방식은 필요할 때마다 컴파일을 하게 되므로 화면 전환 등의 많은 데이터 처리가 필요해 컴파일이 자주 실행된다면 실행 전에 해당 부분을 RAM에 올려야 하기

때문에 RAM 점유율이 증가 되고 그에 따른 배터리 소모도 증가해버리는 문제가 발생한다. 또한 실시간으로 컴파일되므로 기기의 성능에 따라 지연이 생기고 이 차이 또한 달라지게 된다. 하지만 이후 설명할 AOT에 비해 설치시점에서는 컴파일이 진행되지 않음으로 설치시간이 빠르고, 또한 컴파일한 코드를 따로 저장하지 않기 때문에 앱의 용량이 AOT에 비해 적다. 그러나 이러한 장점에도 불구하고, 앞서 설명한 하드웨어적 부하가 심각한 문제로 대두되어 이를 해결하기 위해 새로운 가상머신이 요구되어졌다. 이에 구글에서 안드로이드 4.4버전 공개 당시 함께 AOT(Ahead Of Time)방식의 ART(Android Run Time)를 공개한다. 이러한 방식은 앱을 설치하는 시점에서 모든 코드를 기계어로 변환 후 저장해놓는다. 이로 인해 앱 실행 시의 컴파일이 필요 없어지면서 지연이 사라지게 되었다. 또한 RAM 위에 코드를 올리지 않아도 되므로 RAM 사용성 부분에서도 개선이 되었다. 하지만 설치시점의 변환 및 저장으로 인해 설치 시간이 JIT에 비해 매우 느리고, 기계어 또한 따로 저장하므로 JIT에 비해 각 앱의 크기가 최대 2배 정도 차이가 나게 되었다. 이후 안드로이드 버전 5.0에서 이 ART가 기본 런타임으로 지정되었다. 이 당시 뜻하지 않은 호환성 문제가 생겼는데 ART 이전 앱들은 모두 JIT방식의 DVM에 초점을 두고 개발하였기 때문이다. 이러한 DVM에서는 동작하지만, ART에서 동작하지 않는 호환 문제는 현재 안드로이드 11까지 업데이트되며 대부분의 앱들이 ART를 상정하고 개발되어 없어졌다. 또한 ART에서 동작한다면 DVM에서는 정상 동작하므로 이러한 호환성 문제는 시간이 갈수록 점점 사라질 것이다. 안드로이드 5.0에서 기본 런타임으로 지정된 이후 안드로이드 7.0에서는 ART에 JIT방식또한 조합하여 최종적으로 ART는 JIT와 AOT를 함께 사용하게 되었다. 예를 들어 최초에는 JIT 방식으로 작동하다가 기기를 사용하지 않는 충전할 때와 같은 시점에서 컴파일을 진행하여 AOT 방식으로 전환하는 시도가 있었다. 이러한 방식은 기기 하드웨어의 비약적인 성능 향상과 함께 안드로이드 11 현재 두 방식의 장점을 극대화 하고 있다. Core Library는 아래 계층의 리눅스 커널에 추가기능을 제공하는 역할을 한다. 두 번째로 Native Library가 있다. 아래는 네이티브 라이브러리 계층의 일부 대표적인 라이브러리들에 대한 설명이다. 이 계층에서는 ART 및 HAL 등의 Android의 시스템의 핵심구성요소와 이외의 서비스들이 필요로 하는 라이브러리를 C나 C++로 작성하여 빌드한 계층이다. 이는 다시 말해 Android

| | |
|---|---|
| Surface Manager | •디스플레이의 서브시스템을 관리<br>•여러 앱으로부터 출력된 2D, 3D 그래픽 레이어를 오류, 지연 없이 합성하여 출력 |
| Media Framework | •OpenCORE에 기반하여 오디오와 비디오, 이미지 포맷 제공 |
| SQLite | •모바일전용의 경량화된 데이터베이스 |
| OpenGL/ES | •그래픽 처리 하드웨어의 표준 소프트웨어 인터페이스를 지정 |
| FreeType | •문자열을 비트맵으로 처리 |
| WebKit | •HTML, SVG, JS등을 지원 |
| SGL | •보안소켓레이어(Secure Socket Layer) 프로토콜 |
| SSL | •OpenGL/ES와 달리 2D 그래픽 지원 |
| libc | •시스템 C라이브러리 |

내부의 하위계층의 기능들이 사용할 수 있는 라이브러리들을 저장해놓았다는 말이다. High level의 java에서 쓰인다기보다 Low level의 하드웨어들이나 db 관리, 메모리 관리 등에서 많이 사용된다. 이러한 네이티브 라이브러리는 일부 기능을 노출해주는데, 나중에 설명할 Framework 계층에서는 이것을 활용하여 선언을 통해 간접적으로 실행 할 수 있다. FrameWork 계층(개발자 입장)에서 이러한 네이티브 라이브러리에 직접 접근하여 사용하기 위해서는 Android NDK(Native Development Kit)를 통하여 엑세스 할 수 있다.

## 자바 API 프레임 워크 계층

이 계층부터는 자바로 작성된 코드를 볼 수 있다. 실제로 개발자가 코드를 작성할 때 많이 접하게 되는 API (Application Program Interface)들을 제공하고 앱의 기본이 되는 계층이라고 할 수 있다. 해당 계층에서 하위계층과 상위계층의 통신이 실질적으로 이루어지며 중간다리 역할을 하게 된다. 이를 통해 개발자가 기기의 직접적인 기능에 접근 할 수 있다. 여기서 API라 함은 사용자가 응용프로그램에서 기기의 기능을 사용할 수 있게 할 수 있도록 만든 인터페이스로 앞서 보았던 HAL과 비슷한 종류이다. 이러한 API의 틀을 잡아주는 프레임워크들을 지원하는 계층이라고 할 수 있겠다. 보통 우리가 import 또는 extend 하여 사용해왔던 것과 유사하다고 생각하면 될 것이다. 이러한 API 프레임워크는 시스템 구성요소 및 서비스의 재사용을 단순하게 해준다. 아래는 자바 API 프레임 워크 계층의 항목들을 보여준다.

| | |
|---|---|
| Activity Manager | •앱의 출력주기를 관리<br>•공통 탐색 백 제공 (작업관리위한 일종의 스택) |
| Window Manager | •화면정보, 배치등을 관리 (화면의 투명도 설정, 크기반환 등에 사용) |
| Content Providers | •어떤 앱이 다른앱으로부터의 데이터에 엑세스하거나 전송하는 것을 관리 |
| View System | •리스트. 그리드, 웹 뷰 등 레이아웃들 지원 |
| Notification Manager | •모든 앱의 푸쉬알림을 관리 |
| Package Manager | •현재 디바이스에 설치된 앱들과 관련된 정보 관리 |
| Telephony Manager | •전화 및 통신, 단말기의 상태 등을 관리 |
| Resource Manager | •현지화된 문자열, 그래픽. 레이아웃 파일의 리소스에 대한 엑세스 제공(코드가 아님) |
| Location Manager | •GPS, 기지국 등과의 통신을 통해 현재 기기의 위치를 파악 |
| XMPP Service | •기기간 양방향 통신 지원 |

## 시스템 애플리케이션 계층


이 계층은 일반 사용자들에게 친숙한 계층이다. 사용자가 앱을 설치하게 되면 아이콘이 보이는데 이 상태가 애플리케이션 계층에 있다고 할 수 있을 것이다. 기본적으로 제공되는 시스템 앱들(전화, 문자 등)은 높은 시스템 우선권을 가지고 있어 잘 종료되지 않게 설정되어있다. 하지만 이러한 기본 앱들은 일부 시스템UI 앱 등을 제외하고서는 사용자가 apk파일을 통해 좌측 사진과 같이 설치한 서드파티 앱이 대신 작동하도록 변경할 수 있다.

## 마치며

지금까지 안드로이드의 구조에 대해 정리해보았다. 정리하고 나니 이렇게 구조를 알고 안드로이드를 접하는 것과 아무것도 모른 채로 교재를 따라 하는 것은 엄청난 차이가 날 것임이 자명해 보인다. 왜 안드로이드가 자바 기반이었는지, 우리 기억에 남아있는 초창기 안드로이드는 IOS에 비해 왜 그렇게 성능이 뒤처졌었는지 등등 많은 의문이 해결되는 유익한 시간이었다. 최근의 AOSP에는 앞서 알아보았던 OS의 기본 기능 이외에도 다크모드, 화면 녹화 등 사용자가 직접 체감할 수 있는 여러 가지 사용자 친화적인 기능들이 추가되어가고 있다. 이러한 안드로이드의 발전이 앞으로 스마트폰 시장의 경쟁 OS인 IOS와의 대결 구도에 어떤 변수로 작용할지 기대되는 부분이다.