Android Meomory Leaks

https://github.com/yongtaii/yongapps

1 Memory Leak

안드로이드 메모리 동작방식

Android Memory

- 안드로이드에서 대부분의 앱들은 Android Runtime(ART) 위에서 구동된다.
- ART 와 Dalvik 은 Java Virtual Machine(JVM)과 비슷하며 구동되는 어플리케이션과 사용되는 데이터를 저장하기 위해 Stack과 Heap 메모리 공간을 사용한다.
- Stack과 Heap 메모리 공간은 RAM 메모리에 속해있다

Android 4.4 버전 이하는 ART 대신 DVM(Dalvik Virtual Machine) 을 사용한다.

Stack 메모리

Stack Memory

- Stack 메모리는 정적 메모리 할당 방식으로 사용된다
- Stack 메모리는 지역변수 저장을 위해 사용된다 (primitive type 변수나 객체참조변수 모두)
- LIFO (Last In First Out) 방식으로 참조된다
- 사이즈
  - Dalvik : Java Code 32KB , native(C++/JNI) 1MB
  - ART : Java Code + native(C++/JNI) 1MB
- 제한된 스택메모리에 접근하면 StackOverflowError가 발생한다
- 메서드가 호출될 때 마다 메서드 지역변수를 갖는 블록(Stack frame) 이 Push된다.
  메서드가 완료되면 이 Stack frame은 Pop 되고 결과값이 스택으로 다시 Push된다

Heap 메모리 #1

Heap Memory

- Heap 메모리는 동적 메모리 할당 방식으로 사용된다
- Heap 메모리는 객체를 할당하기 위해 사용된다
- 사용자를 위해 Android는 실행중인 각 어플리케이션의 힙 크기를 엄격하게 제한한다. 이 때의 Heap size는 디바이스와 RAM에 따라 다르게 제한한다.
- 사이즈 : Android 2.3 이상 단말에서 getMemoryClass() 메서드를 통해 확인이 가능하다. 36MB 정도의 크기로 제한을 둔다.
- 앱이 제한된 최대 힙 사이즈를 넘는 메모리 할당을 요구 할 경우 OutOfMemoryError 가 발생하고 앱이 종료된다.

Heap 메모리 #2

Heap Memory

- Heap 은 Stack 메모리와 달리 함수가 완료될 때 객체가 자동으로 회수되지 않는다.
- JVM,DVM, ART와 같은 가상머신에는 사용하지 않는 객체를 감지하고 회수하는 Garbage Collector가 존재한다.
- Garbage Collector는 unreachable한 상태의 객체를 찾는다. 힙 메모리에 참조되지 않는 객체가 존재할 경우 할당이 해제된다.
- 사용하지 않는 객체가 힙에 있는데, 해당 객체를 Stack에서 계속 참조하는 경우가 있다. 이에 따라 Garbage Collector는 힙으로부터 해당 객체를 할당해제 하는데 실패한다. 따라서 메모리는 증가하게 되고, 이를 Memory Leak 이라고 한다.

Android largeHeap 설정

largeHeap

- Heap Memory 크기를 동적으로 늘릴 수는 없다
- large memory 가 필요하다면, 앱에 large Dalvik heap을 요청할 수있다
- Honeycomb 이전 디바이스에서는 지원되지 않는다
- AndroidManifest.xml의 〈appliation〉 태그에 android:largeHeap="true"를 통해 설정이 가능하다
- 힙 사이즈가 얼마나 커지는 지는 보증이 안된다

Android largeHeap 설정

Attention !

- 단순히 Memory 사용량을 증가시키려고 largeHeap 사용하는 것은 좋지않다
- 힙 사이즈를 늘리면, GC 수집에 더 많은 시간이 걸린다. 로그를 확인하면 GC Pause 시간이 더 길어지는 것을 확인할 수 있다.
- 따라서 메모리가 부족하다는 이유로 largeHeap="true"를 설정하면 안되고, 최후의 단계로 남겨두자.
- 이보다 메모리 최적화가 더 우선되어야 한다.

Memory Leak ?

...그래서 메모리 릭이란 ?

힙 메모리 에서 사용되지 않는 객체를 할당 해제(release)에 실패하는 것

메모리 릭 관리 필요성 #1

short Garbage Collector

- 메모리가 할당되었지만 해제되지 않은 경우 Momory Leak이 발생한다. 이 것은 GC(Garbage Collector)가 쓰레기를 처리할 수 없다는 것을 의미한다.
  사용자가 앱을 계속 이용할 때, 힙메모리도 지속적으로 증가한다. short GC는 죽은 객체를 즉시 처리하려고 할 것이다. 현재 이러한 short GC들은 동시에 동작하며(각자의 쓰레드에서) 앱 속도를 크게 저하시키기 않는다(2ms ~ 5ms pause).
  그러나, 적은 GC 수행이 앱 성능 향상에 도움이 된다는 것은 기억해야 한다!

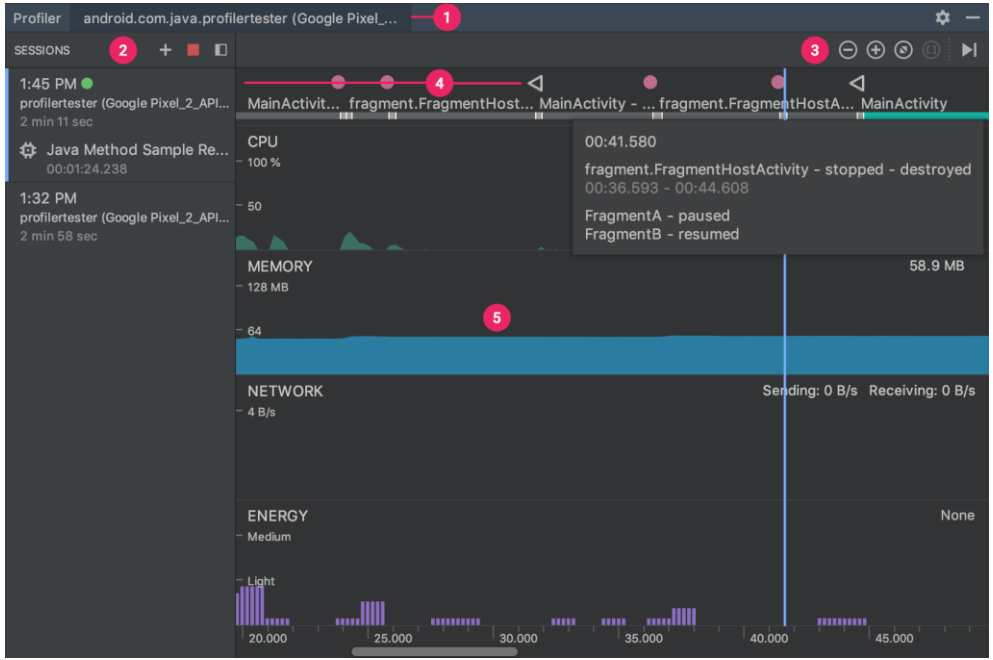
메모리 릭 관리 필요성 #2

Larger GC(stop-the-world)

- 만약 앱에 심각한 Memory Leak이 있다면, short GC들은 메모리 회수를 할 수 없다. 그리고 힙 메모리는 계속적으로 증가한다. 이것은 더 큰 GC 가 시작되도록 한다.
  Larger GC (called “stop-the-world”) 는 앱 메인스레드를 50ms~100ms 시간동안 중지시킨다. 이 때 앱은 심각하게 렉(lags)이 걸리고 거의 사용할 수없게 된다.
- 해당 Memory Leak을 해결하지 않는다면, 앱에서 더 이상 메모리를 할당할 수 없는 지경에 이를 수 있다. 그리고 앱 크래시를 일으키는 OutOfMemoryError가 발생한다.

메모리 릭 감지 방법 #1

Android Profiler



Android Studio 3.0 이후부터

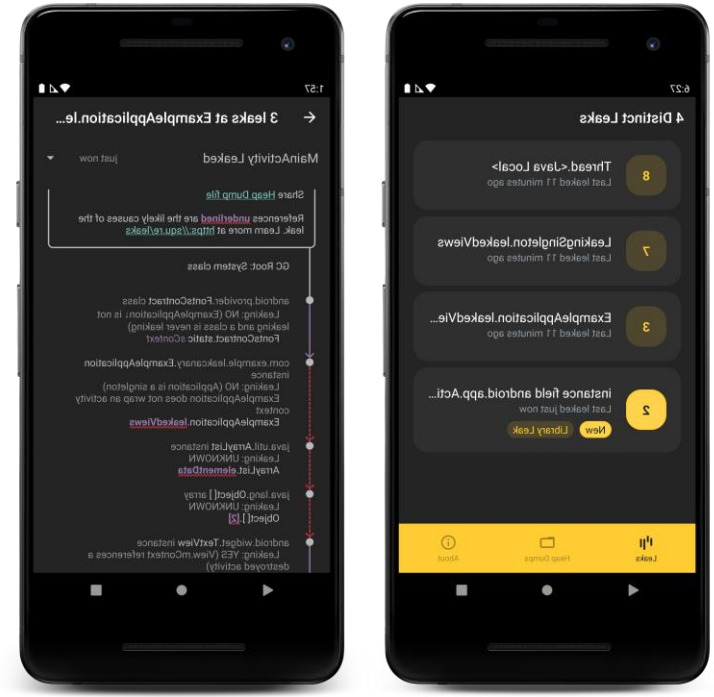
제공되는 Profiler 를 이용하면

CPU/MEMORY/ NETWORK/

ENERGY 사용 정보를 볼 수있다

메모리 릭 감지 방법 #2

LeakCanary



Square에서 제공하는 LeakCanary 라이브러리를 이용하면 Memory Leak 발생시 Stack Trace 정보와 함께 확인할 수 있다.

노티피케이션 알람을 통해 Memory Leak 이 발생했음을 알려준다.

2 Memory Leak Senarios

Context

Application/Activity Context

- Context의 잘못된 사용으로 Memory Leak을 발생시킬 수 있다
- activity-level Context와 application-level Context를 어떤 상황에서 써야 하는지 이해하는 것이 중요하다
- 잘못된 위치에서 Activity-Context를 사용하는 것은 해당 Activityd 데한 참조가 계속 유지되는 것이며, 잠재적인 Memory Leak을 발생시킬 수 있다

#CASE 01 : Singleton Class Reference

Singleton Class Reference

```
public class SampleActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sample);
        SingletonClass singletonClass = SingletonClass.getInstance(this);
    }
}
```

액티비티에서 싱클톤 클래스를 초기화할때, 위와 같이 activity-context 를 넣을 수있다.

이렇게 되면, 싱글톤은 어플리케이션이 종료될때까지 해당 액티비를 유지하게 된다

#Solution01 : Singleton Class Reference

solution01

```
public class SampleActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sample);
        SingletonClass singletonClass = SingletonClass.getInstance(getApplicationContext());
    }
}
```

Activity-context 대신 Application-context를 넣는다

#Solution02 : Singleton Class Reference

solution02

```
public synchronized static SingletonClass getInstance(Context context) {
    if (instance == null) {
        instance = new SingletonClass(context.getApplicationContext());
    }
    return instance;
}
```

싱글톤 클래스에서 객체를 생성할 때 Application-Context를 사용하게 만들어 준다

#Solution03 : Singleton Class Reference

solution03

```
public void onDestroy() {
    if(instance != null) instance = null;
}
```

Activity-Context를 사용해야 한다면 액티비티가 Destroy()될때 싱글톤 클래스를 null 로 만들어준다

Static Activity or View Reference

Static View Reference

- Activity나 View를 Static으로 참조한다면, 해당 액티비티는 Destory() 된 이후에 GC에 의해 수집되지 않을 수있다

#CASE 02 : Static Activity or View Reference

Static Activity or View Reference

```
public class StaticViewActivity extends AppCompatActivity {

    private static TextView textView;
    private static Activity activity;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_static_view);

        textView = findViewById(R.id.tv);
        activity = this;
    }

}
```

TextView, Activity 변수를 Static으로 선언해서 사용하고 있다

#Solution01 : Static Activity or View Reference

solution01

View나 Activity,Context 관련 변수는 **절대** static 변수로 사용하지 않아야 한다

Unregistered Listeners

Unregistered Listeners

- Activity나 Fragment에 Listener를 등록하고 해제하지 않은 경우들이많다.
  이것은 매우 큰 Memory Leak을 유발할 수 있다

#CASE 01 : LocationManager Reference

LocationManager Reference

```
public class LocationActivity extends AppCompatActivity {

    private LocationManager mLocManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_location);

        mLocManager = (LocationManager) getSystemService(LOCATION_SERVICE);
        mLocManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
           TimeUnit.MINUTES.toMillis(5), 100, locationListener);

    }
}
```

앱에서 위치 업데이트를 받기 위해 LocationManahger를 이용해 Listener를 등록한다.

따라서 Activity가 LocationManager에 대한 Strong Reference를 갖게 된다.

이대로 액티비티가 종료되면 GC는 해당 인스턴스를 할당해제하지 못한다.

#Solution01 : LocationManager Reference

solution01

```
@Override
protected void onDestroy() {
    if(mLocManager != null){
        mLocManager.removeUpdates(this);
    }
    super.onDestroy();

}
```

액티비티가 종료될 때 (onDestroy()) 해당 리스너를 해제한다

#CASE 02 : Broadcast receiver Reference

LocationManager Reference

```
@Override
protected void onStart() {
    super.onStart();
    registerReceiver(mBroadcastReceiver, new IntentFilter("IntentFilter.intent.MAIN"));
}
```

위 코드는 액티비티에서 브로드캐스트 리시버를 등록하는 샘플코드다.

이때 해당 리시버를 등록해제 안한다면 액티비티가 종료 되어도 액티비티에 대한 참조가 계속 유지된다

#Solution01 : Broadcast receiver Reference

solution01

```
@Override
protected void onStop() {
    super.onStop();
    if(mBroadcastReceiver != null) {
        unregisterReceiver(mBroadcastReceiver);
    }
}
```

브로드캐스트 등록해제 코드를 추가해준다.

브로드캐스트는 보통 onStop()에서 해준다.

Inner classes

Inner classes

- 내부 클래스는 Java뿐만 아니라 안드로이드에서도 많이 쓰인다.
- 그러나 적절하지 않은 사용법으로 인해 잠재적인 Memory Leak을 유발할 수 있다.

#CASE 01 : AsyncTask Reference

AsyncTask Reference

```
public class SampleActivity extends AppCompatActivity {
    TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        //…
        new BackgroundTask().execute();
    }

    private class BackgroundTask extends AsyncTask<Void, Void, String> {
        @Override
        protected String doInBackground(Void... params) {
            return "some string";
        }
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result);
        }
    }
}
```

AsynTask를 통해 백그라운드에서 작업 후 결과를 액티비티의 TextView에 표시해주는 예제이다.

Task가 종료되지 않고 액티비티를 떠나면 Memory Leak을 발생시킨다. non-static 내부클래스가 이를 감싸는 클래스(액티비티) 에 대한 implicit reference를 계속 갖고 있기 때문이다.

#Solution01 : AsyncTask Reference

solution01

1) non-static 내부클래스를 static 내부클래스로 변경한다. static 내부클래스는 이를 감싸고 있는 외부 클래스에 대한 implicit 참조를 하지 않는다.

2) 요구되는 참조객체를 내부클래스로 전달해라. static 클래스는 non-static 참조변수에접근할 수 없기 때문이다.

3) onDestory()에서 AsyncTask를 취소해라

4) TextView에 대한 참조가 여전히 강력하고 GC 수집을 방지하게되므로 추가적인 Memory Leak을 막기 위해 전달된 객체를 WeakReference로 감싸라.

#Solution01 : AsyncTask Reference

```
public class SampleActivity extends AppCompatActivity {

    TextView textView;
    AsyncTask task;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // …
        textView = (TextView) findViewById(R.id.tv);
        task = new BackgroundTask(textView).execute();
    }
    @Override
    protected void onDestroy() {
        task.cancel(true);
        super.onDestroy();
    }
    private static class BackgroundTask extends AsyncTask<Void, Void, String> {

        private final WeakReference<TextView> textViewReference;
        public BackgroundTask(TextView resultTextView) {
            this.textViewReference = new WeakReference<>(resultTextView);
        }
        @Override
        protected void onCancelled() {
            // Cancel task. Code omitted.
        }
        @Override
        protected String doInBackground(Void... params) {
                        return "some string";
        }
        @Override
        protected void onPostExecute(String result) {
            TextView view = textViewReference.get();
            if (view != null) view.setText(result);
}}}
```

onPostExecute에서 null체크를 해준다. (객체가 회수되었는지 아닌지 확인)

Anonymous classes

Anonymous classes

- Anonymous 클래스는 개발자들이 많이 사용한다. 그러나 Anonymous 클래스들은 non-static 클래스일 뿐이며 이는 Momory Leak을 유발할 수 있다

#CASE 01 : Hnadler Reference

Handler Reference

```
new Handler().postDelayed(new Runnable() {
    @Override
    public void run() {
        // do something
    }
}, 10000);
```

Runnable 작업이 끝나기 전에 Activity가 종료되면 Memory Leak이 발생할 수 있다.

모든 Anonymous 클래스들은 그들의 부모(액티비티)를 참조하고 있기 때문이다.

#CASE 02 : Retrofit Reference

Retrofit Reference

```
MoviesRepository repository = ((MoviesApp) getApplication()).getRepository();
repository.getMoviesThisWeek()
        .enqueue(new Callback<List<Movie>>() {
            @Override
            public void onResponse(Call<List<Movie>> call,
                                   Response<List<Movie>> response) {
                int numberOfMovies = response.body().size();
                textview.setText("result: " + String.valueOf(numberOfMovies));
            }
            @Override
            public void onFailure(Call<List<Movie>> call, Throwable t) {
                // Oops.
            }
        });
```

위는 네트워크 작업이 완료되면TextView에 표시하는 Retrofit 사용 예제를 나타낸다.

여기서 Callable 객체는 이를 감싸는 액티비티를 참조한다.

네트워크 콜이 끝나기 전에 액티비티가 종료되거나 회전되면 액티비티 객체 메모리는 Leak된다.

#Solution : Anonymous classes

solution01

1) Anonymous 클래스에서는 긴동작을 하지마라.
2) 또한, 액티비티나 뷰에 WeakReference를 전달하기 위해서는 static class가 필요하다
3) Thread 또한 Anonymous 클래스다.

solution02

1) 액티비티가 onDestroy() 될 때, 핸들러/타이머/네트워크 콜을 취소한다.

📌 참고

- https://medium.com/programming-lite/memory-leaks-in-details-in-android-8b2832905f4f

Thanks!

Any questions ?

You can find me at

- jeonyt89@gmail.com