BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

BÁO CÁO BÀI TẬP LỚN

Đề tài:

Nghiên cứu android memory optimization

Môn học: Tối ưu phần mềm di động

Nhóm sinh viên thực hiện:

Nguyễn Khánh Duy CT040113

Vũ Trọng Chương CT040107

Trần Đức Toàn CT040148

Người hướng dẫn: **Lê Bá Cường**

Hà Nội, 2021

Mục lục

MỤC LỤC ẢNH

Lời mở đầu

Cùng với sự phát triển của khoa học kỹ thuật công nghệ dần đi vào đời sống của mọi người. Hiện nay mỗi người đều sở hữu điện thoại thông minh để phục vụ các nhu cầu khác nhau như học tập, giải trí, lưu trữ dữ liệu, … Do đó việc xây dựng các ứng dụng để phục vụ nhu cầu của mọi người là ngành vô cùng tiềm năng và sẽ phát triển mạnh hơn nữa trong tương lai.

Phần mềm điện thoại thông minh hiện nay được phát triển cho hai nền tảng chính là ANDROID và IOS, nhưng với lợi thế là hệ điều hành mở Android đã vươn lên là một hệ điều hành được sử dụng phổ biến nhất của các điện thoại thông minh.

Cùng với tiềm năng phát triển như vậy thì các phần mềm cũng ngày càng cạnh tranh dẫn đến chất lượng các phần mềm dần được nâng cao và cải thiện hơn về hiệu năng, độ tin cậy, tính an toàn, … Làm thế nào để làm ra được các ứng dụng có hiệu suất cao tiết kiệm năng lượng? Mục tiêu của báo cáo này là cung cấp cái nhìn tổng quát về tối ưu hóa bộ nhớ, giới thiệu các kỹ thuật tối ưu hóa bộ nhớ, giới thiệu một số phần mềm giúp tối ưu hóa bộ nhớ nhằm tạo ra một ứng dụng tối ưu về bộ nhớ, tiết kiệm năng lượng.

Chương 1: Giới thiệu tổng quan về tối ưu hóa bộ nhớ

Ngày nay, điện thoại thông minh đã trở thành một nhu cầu thiết yếu đối với tất cả mọi người. Hầu hết chúng ta đều phụ thuộc quá nhiều vào điện thoại thông minh để hoàn thành công việc hàng ngày của chúng ta. Ở một số khía cạnh, chúng ta có đã thay thế máy tính của họ bằng chiếc điện thoại thông minh. Cùng với thời gian, điện thoại thông minh ngày càng phát triển và các thiết bị ngày càng trở nên mạnh mẽ.

Android là hệ điều hành điện thoại thông minh được sử dụng phổ biến nhất hiện nay bởi vì nó là Mã nguồn mở và các nhà sản xuất có thể dễ dàng tích hợp nó vào phần cứng của họ, làm cho các thiết bị Android rẻ hơn đối thủ cạnh tranh iOS của họ. Ngoài những mặt tích cực, nó cũng có những tiêu cực khác. Một trong những vấn đề lớn mà người dùng Android phải đối mặt là gặp các sự cố không mong muốn nó làm chậm thiết bị của họ. Những vấn đề như vậy chủ yếu gây ra khi thiết bị hết bộ nhớ. Các nhà sản xuất điện thoại tiếp tục tăng bộ nhớ chính để bù đắp nhưng đây không phải là giải pháp hiệu quả nhất.

Hiện tại, hệ điều hành Android sử dụng Garbage Collector để quản lý bộ nhớ. Garbage Collector là một bộ nhớ Java Công cụ quản lý như Android được xây dựng trên Máy ảo Java (JVM). Công cụ Garbage Collector theo dõi và xác định các đối tượng và lấy lại không gian khi chúng không cần thiết nữa. Về cơ bản nó có thể giải phóng bộ nhớ theo hai cách. Đầu tiên theo định kỳ kiểm tra các đối tượng không cần thiết và giải phóng bộ nhớ của chúng, thứ hai ngay lập tức giải phóng bộ nhớ khi hệ thống cần bộ nhớ lớn hơn bộ nhớ trống có sẵn. Giả sử, chúng tôi có một điện thoại thông minh Android với bộ nhớ 1 GB. Bây giờ, có ba đối tượng là a, b và c. Khi chúng được tạo, bộ nhớ sẽ được phân bổ cho các đối tượng này theo yêu cầu của nó. Khi những đối tượng này bị hủy sau khi phục vụ mục đích, nó được đánh dấu là không cần thiết nhưng bộ nhớ vẫn không được giải phóng hoàn toàn. Trong tình huống này, nếu một số đối tượng mới sẽ được cấp phát một số bộ nhớ và kích thước phân bổ lớn hơn bộ nhớ trống khả dụng thì một sự cố sẽ xảy ra. Điều này xảy ra do nguyên nhân là Garbage Collector đã

không xác nhận không gian chết. khi đối tượng mới có nhu cầu phân bổ bộ nhớ cao hơn bộ nhớ hiện có nó sẽ gây ra hiện tượng giật và trải nghiệm người dùng kém. Hơn nữa, JVM sao chép các đối tượng từ nơi này sang nơi khác trong quá trình Garbage Collector và không ghi đè lên dữ liệu cũ đã thu thập. Cái này có thể dẫn đến rò rỉ quyền riêng tư và dữ liệu cá nhân bị xâm phạm.

Để giải quyết những vấn đề như vậy và cải thiện quản lý bộ nhớ, các giải pháp khác nhau đã được đề xuất bằng cách giữ thêm giải phóng bộ nhớ bằng cách sử dụng Bộ đệm GPU, Nền thích ứng Quản lý ứng dụng, áp dụng tối ưu hóa vi mô, tránh sao chép bộ nhớ, phát hiện những tiến trình không tối ưu và sửa chữa những điều đó, v.v. Tất cả những điều này kỹ thuật được cung cấp bởi Garbage Collector. Vấn đề chính vẫn là cách Trình thu gom rác giải phóng bộ nhớ của các đối tượng đã chết.

Mục đích của nghiên cứu là đề xuất một cách tiếp cận hoàn toàn khác để xử lý vấn đề này. Sử dụng đếm tham chiếu tự động (ARC) trong thiết bị Android thay vì có thể dùng Bộ thu gom rác để cải thiện các vấn đề về memory.

Chương 2: Nội dung về tối ưu hóa bộ nhớ

2.1. Bộ nhớ android

Bộ nhớ android được chia thành 3 loại:

2.1.1. RAM

RAM được sử dụng để lưu trữ tạm thời thông tin trong khi chạy ứng dụng. Bao gồm cả thông tin từ ứng dụng hiện đang chạy và các ứng dụng đang chạy trong nền. Vì nó là tạm thời, thông tin sẽ bị mất khi ứng dụng bị dừng hoặc khi thiết bị được khởi động lại.

Ngày nay, một thiết bị Android thường có 4GB RAM, nhưng một số thiết bị gần đây có 6GB trở lên. Nhiều RAM hơn có nghĩa là nhiều bộ nhớ tạm thời hơn, cho phép người dùng thực hiện nhiều tác vụ hơn - tuy nhiên đối với hầu hết mọi người, 4GB là rất nhiều RAM để thực hiện các tác vụ mong muốn và các tác vụ hệ thống, một ứng dụng chỉ có thể truy cập một phần nhỏ trong tổng số RAM.

2.1.2. zRAM

zRAM là một phần của RAM và hoạt động thông qua việc nén các tài nguyên không sử dụng và chuyển chúng đến một khu vực dành riêng để giải phóng dung lượng, làm tăng bộ nhớ khả dụng. Cần lưu ý rằng quá trình nén và giải nén bộ nhớ đòi hỏi CPU phải làm việc nhiều hơn và có thể làm chậm hoạt động của thiết bị.

2.1.3. Storage

Bộ nhớ lưu giữ tất cả dữ liệu liên tục, bao gồm ảnh, video, nhạc và tài liệu, sẽ vẫn còn nếu điện thoại được khởi động lại. Ví dụ về lưu trữ liên tục trên Android bao gồm các cặp khóa / giá trị được lưu trữ trong các tùy chọn chia sẻ và lưu thông tin vào cơ sở dữ liệu, chẳng hạn như sử dụng SQLite, Realm hoặc Room. Dung lượng lưu trữ trên thiết bị Android có thể khác nhau nhưng dung lượng lưu trữ ngày nay thường là 32GB hoặc 64GB. Trên một số thiết bị, bộ nhớ có thể được mở rộng thông qua bộ nhớ ngoài như thẻ nhớ microSD.

Android Runtime (ART) và Dalvik sử dụng phân trang (paging) và ánh xạ bộ nhớ (memory-mapping hay mmapping) để quản lý bộ nhớ. Điều này có nghĩa là bất kỳ bộ nhớ nào mà một ứng dụng thay đổi, cho dù bởi việc cấp phát các đối tượng mới hoặc thay đổi các trang bộ nhớ bị ánh xạ, thì vẫn tồn tại trong bộ nhớ và không thể được loại bỏ. Cách duy nhất để giải phóng bộ nhớ từ một ứng dụng là giải phóng các đối tượng tham chiếu mà ứng dụng lưu giữ, làm cho bộ nhớ sẵn sàng cho trình thu gom rác (garbage collector). Có một ngoại lệ: bất kỳ file nào được ghi vào mà không sửa đổi, chẳng hạn như mã nguồn, có thể được loại bỏ khỏi bộ nhớ nếu hệ thống muốn sử dụng bộ nhớ đó ở nơi khác.

2.2. Memory Leaks (Rò rỉ bộ nhớ)

Trước tiên, cần hiểu về 2 loại phân cấp bộ nhớ: Stack và Heap

2.2.1. Stack

Stack (Ngăn xếp): Ngăn xếp được sử dụng để cấp phát bộ nhớ tĩnh. Điều này có nghĩa là các đối tượng ngăn xếp là tạm thời và một khi chức năng hoàn tất, mỗi đối tượng sẽ được giải phóng khỏi bộ nhớ.

2.2.2. Heap

Heap (Đống): Heap được sử dụng để cấp phát bộ nhớ động. Điều này hoạt động khác với ngăn xếp trong đó một khi chức năng hoàn tất, các đối tượng sẽ không được lấy lại hoặc giải phóng khỏi bộ nhớ.

2.2.3. Memory Leaks

Để giải phóng các đối tượng khỏi heap, Android sử dụng Garbage Collector (GC) để thu hồi các đối tượng không sử dụng để giải phóng bộ nhớ. Nó chỉ đơn giản là theo dõi các đối tượng trong heap, xác định khi nào nó không còn được sử dụng nữa và tại thời điểm đó, nó giải phóng các đối tượng khỏi heap. Nếu chúng ta vô tình giữ một tham chiếu đến một đối tượng không sử dụng trong heap, Garbage Collector cho rằng nó vẫn đang được sử dụng và không nhận nó là rác - có nghĩa là nó không thể giải phóng đối tượng khỏi bộ nhớ.

Nếu chúng ta tiếp tục lưu trữ nhiều đối tượng không sử dụng hơn trong heap và Bộ thu gom rác không thể lấy lại bộ nhớ, các đối tượng sẽ tiếp tục hình

thành cho đến khi chúng ta hết dung lượng. Điều này có thể khiến ứng dụng của chúng ta OutOfMemory bị treo, đóng băng giao diện người dùng và cuối cùng dẫn đến một ngoại lệ OutofMemory làm ứng dụng bị treo.

Android chạy trên Nhân Linux được tùy chỉnh điều khiển phần cứng như Bộ nhớ và CPU và cho phép giao tiếp với phần cứng và người dùng. Hai thành phần chính của các ứng dụng Android là Activity và Fragment. Code không được tối ưu hóa cho các thành phần này mà hầu hết các trường hợp quản lý tài nguyên yếu kém, có thể dẫn đến rò rỉ bộ nhớ và các vấn đề về bộ nhớ. Chi tiết về các thành phần này được thảo luận dưới đây:

Activity là thành phần cơ bản của bất kỳ ứng dụng Android nào. Nó về cơ bản đại diện cho một View Controller. Một chế độ xem duy nhất trong một ứng dụng là một activity. Bất cứ khi nào, một chế độ xem mới được hiển thị, đó là một activity mới, khi chế độ xem đó bị loại bỏ, activity sẽ bị phá hủy.

Fragment là các phần của giao diện người dùng. Chúng là các thành phần có thể tái sử dụng có thể được sử dụng trong các hoạt động khác nhau. Vòng đời của các fragmet là phụ thuộc vào vòng đời của activity. Khi một activity được tạo, các fragment được tạo ra. Khi một activity bị phá hủy, các fragment cũng bị phá hủy.

Hầu hết các vấn đề rò rỉ bộ nhớ là do code không được tối ưu hóa trong số các activity và fragment, chẳng hạn như quên tái chế các phiên bản bitmap, sự kiện xử lý nhấp chuột hủy đăng ký, đóng các phiên bản con trỏ sau khi truy cập cơ sở dữ liệu và tham chiếu đến các đối tượng từ các lớp tĩnh hoặc đánh dấu các đối tượng là tĩnh.

2.3. Detecting memory leaks

Một số công cụ giúp chúng ta phát hiện rò rỉ bộ nhớ:

2.3.1. Memory Profiler

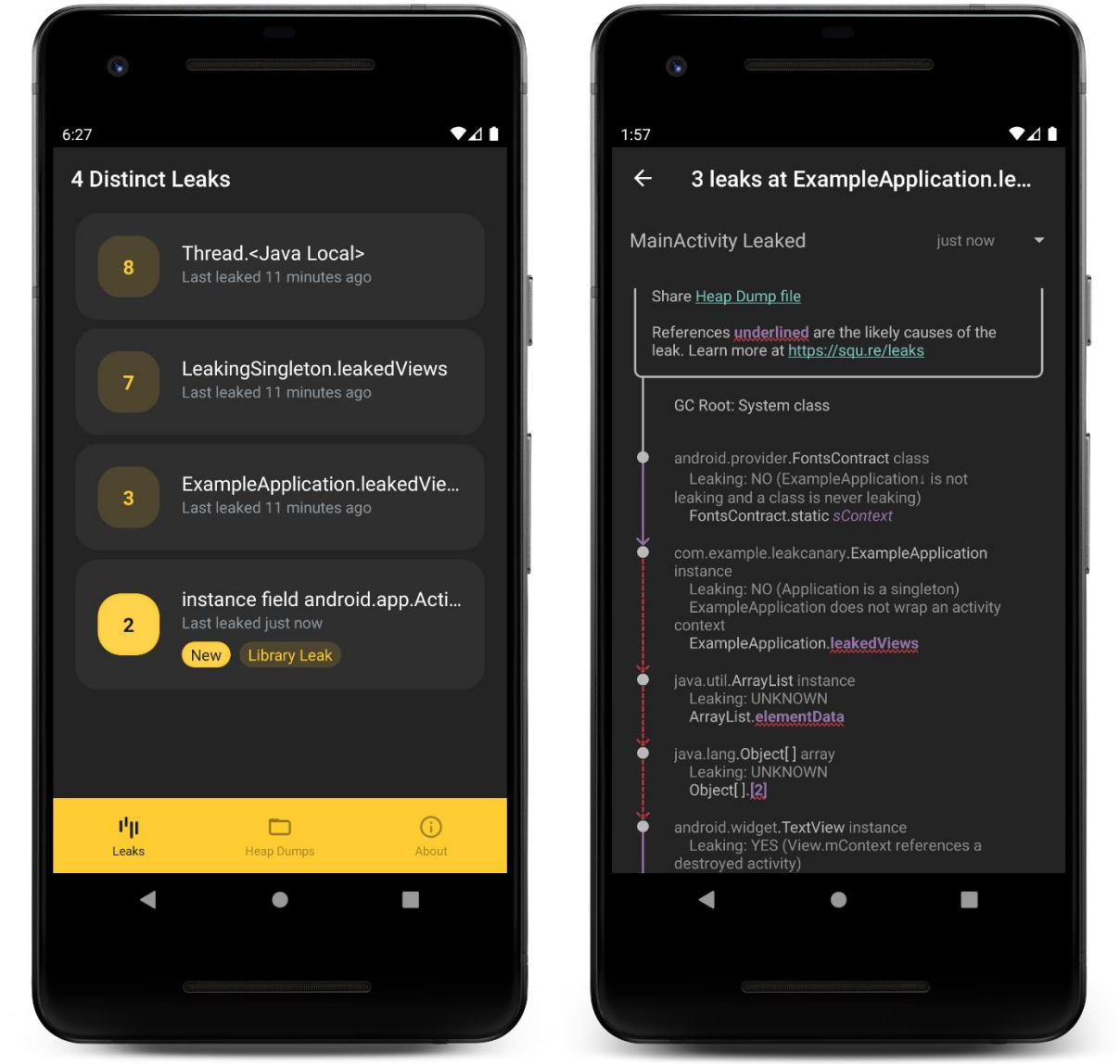
Android Studio cung cấp một công cụ có tên là Memory Profiler để phát hiện rò rỉ bộ nhớ. Công cụ này chia bộ nhớ thành các phân đoạn, bao gồm mã Java, Đồ họa, Ngăn xếp và số lượng các loại bộ nhớ khác để cung cấp cái nhìn

tổng quan về những gì đang sử dụng hết bộ nhớ khi bạn điều hướng qua ứng dụng.

Hình 1: Giao diện Memory Profiler

2.3.2. LeakCanary

Một cách khác để tìm rò rỉ bộ nhớ là sử dụng Leak Canary, một thư viện do Square tạo ra để giúp bạn phát hiện rò rỉ bộ nhớ. Nó hoạt động thông qua việc sử dụng một lớp ObjectWatcher chứa các tham chiếu đến các đối tượng bị phá hủy trong heap. Nếu tham chiếu không được xóa trong năm giây sau khi Trình thu gom rác đã chạy, đối tượng được coi là được giữ lại và ghi lại điều này như là một sự cố rò rỉ bộ nhớ tiềm ẩn trong logcat.



Hình 2: LeakCanary

Để sử dụng LeakCanary, hãy thêm phần dependency leakcanary-android vào file build.gradle ứng dụng:

```
dependencies {
  // debugImplementation because LeakCanary should only run in debug builds.
  debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.7'
}
```

Hình 3: Thêm thư viện LeakCanary vào file build.gradle

Xác nhận rằng LeakCanary đang chạy khi khởi động bằng cách lọc LeakCanary trong Logcat:

```
D LeakCanary: LeakCanary is running and ready to detect leaks
```

Hình 4: Lọc LeakCanary trong Logcat

LeakCanary tự động phát hiện các đối tượng sau:

- Acticity bị phá hủy
- Fragment bị phá hủy
- Các View trường hợp phân mảnh bị phá hủy
- Các ViewModel trường hợp đã xóa

Khi LeakCanary được cài đặt, nó sẽ tự động phát hiện và báo cáo rò rỉ bộ nhớ, theo 4 bước:

- Detecting retained objects (Phát hiện các đối tượng được giữ lại)
- Dumping the heap (Thu gom các heap)
- Analyzing the heap (Phân tích các heap)
- Categorizing leaks (Phân loại rò rỉ)

❖ Detecting retained object

LeakCanary kết nối vào vòng đời của Android để tự động phát hiện khi nào các hoạt động và phân đoạn bị phá hủy và cần được thu gom rác. Các đối tượng bị phá hủy này được chuyển đến ObjectWatcher, và giữ các tham chiếu đến chúng. LeakCanary tự động phát hiện rò rỉ cho các đối tượng sau:

- Activity bị phá hủy
- Fragment bị phá hủy
- Các fragment View bị phá hủy
- Các ViewModel đã xóa

Bạn có thể xem bất kỳ đối tượng nào không còn cần thiết, ví dụ như một detched view hoặc một destroyed presenter:

```
AppWatcher.objectWatcher.watch(myDetachedView, "View was detached")
```

Hình 5: Xem 1 đối tượng không cần thiết

Nếu tham chiếu được giữ bởi ObjectWatcher không được xóa sau khi đợi 5 giây và GC đang chạy, thì đối tượng đã theo dõi được coi là được giữ lại và có khả năng bị rò rỉ. LeakCanary ghi lại điều này vào Logcat:

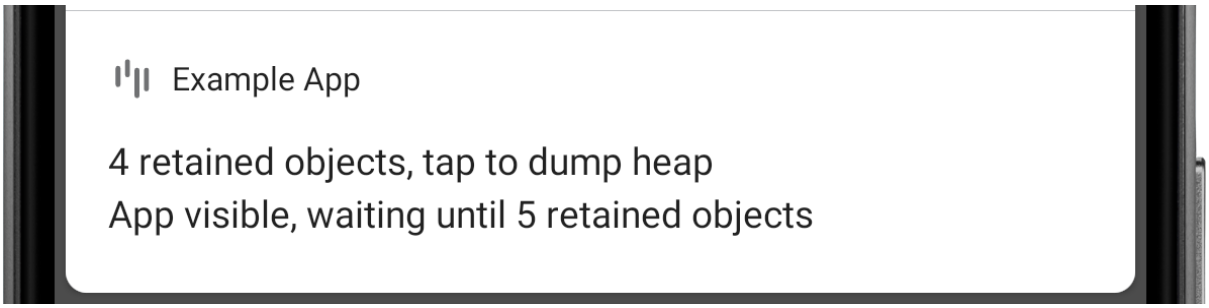
```
D LeakCanary: Watching instance of com.example.leakcanary.MainActivity
  (Activity received Activity#onDestroy() callback)

... 5 seconds later ...

D LeakCanary: Scheduling check for retained objects because found new object
  retained
```

Hình 6: Logcat đối tượng có khả năng bị rò rỉ

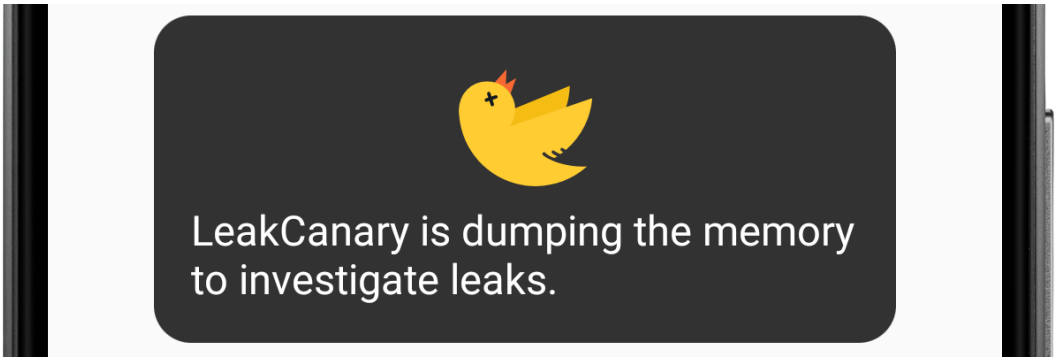
LeakCanary đợi số lượng đối tượng được giữ lại đạt đến ngưỡng trước khi kết xuất heap và hiển thị thông báo với số lượng mới nhất.



Hình 7: LeakCanary hiển thị thông báo tìm thấy 4

❖ Dumping the heap

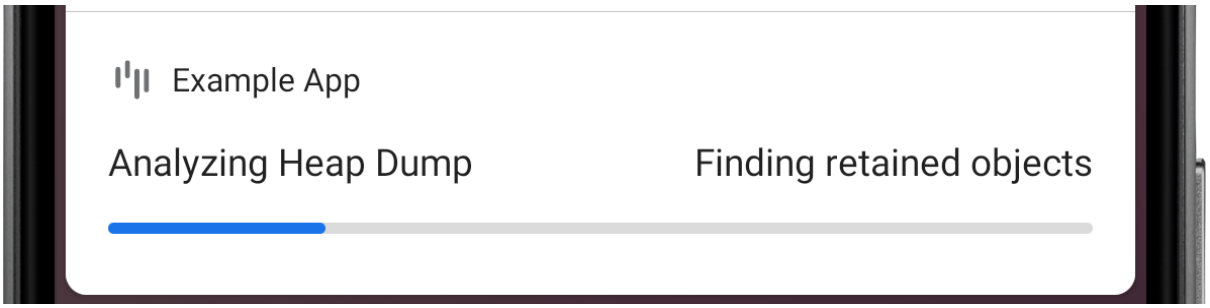
Khi số lượng các đối tượng được giữ lại đạt đến một ngưỡng, LeakCanary kết xuất heap vào một tệp .hprof (một kết xuất heap) được lưu trữ trên hệ thống tệp Android. Việc Dumping the heap sẽ đóng băng ứng dụng trong một khoảng thời gian ngắn, trong đó LeakCanary hiển thị toast sau:



Hình 8: LeakCanary hiển thị Toast thông báo khi kết xuất

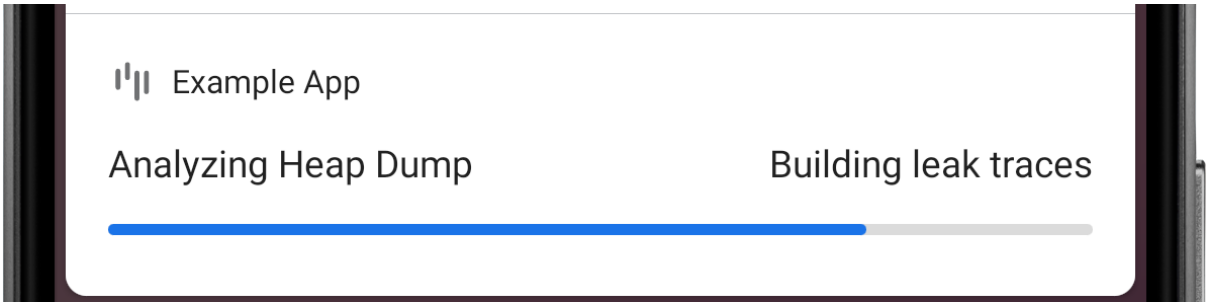
❖ Analyzing the heap

LeakCanary phân tích cú pháp tệp .hprof bằng cách sử dụng Shark và định vị các đối tượng được giữ lại trong kết xuất đống đó.



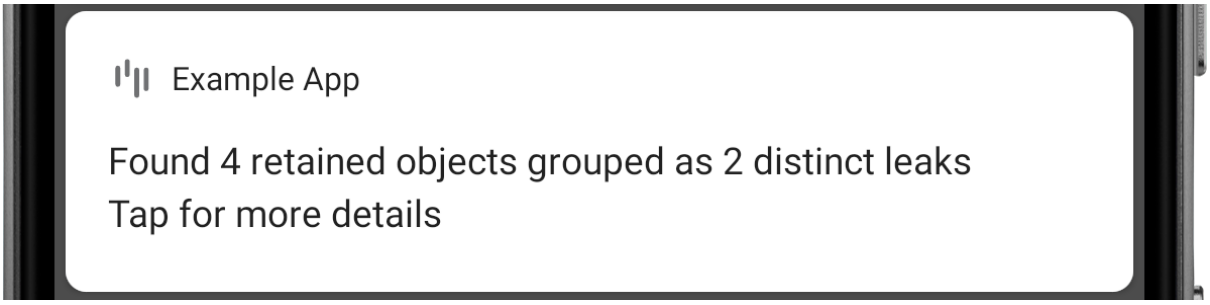
Hình 9: Tìm các đối tượng được giữ lại trong kết xuất heap

Đối với mỗi đối tượng được giữ lại, LeakCanary tìm đường dẫn của các tham chiếu ngăn đối tượng được giữ lại đó và thu thập dấu vết rò rỉ của nó.



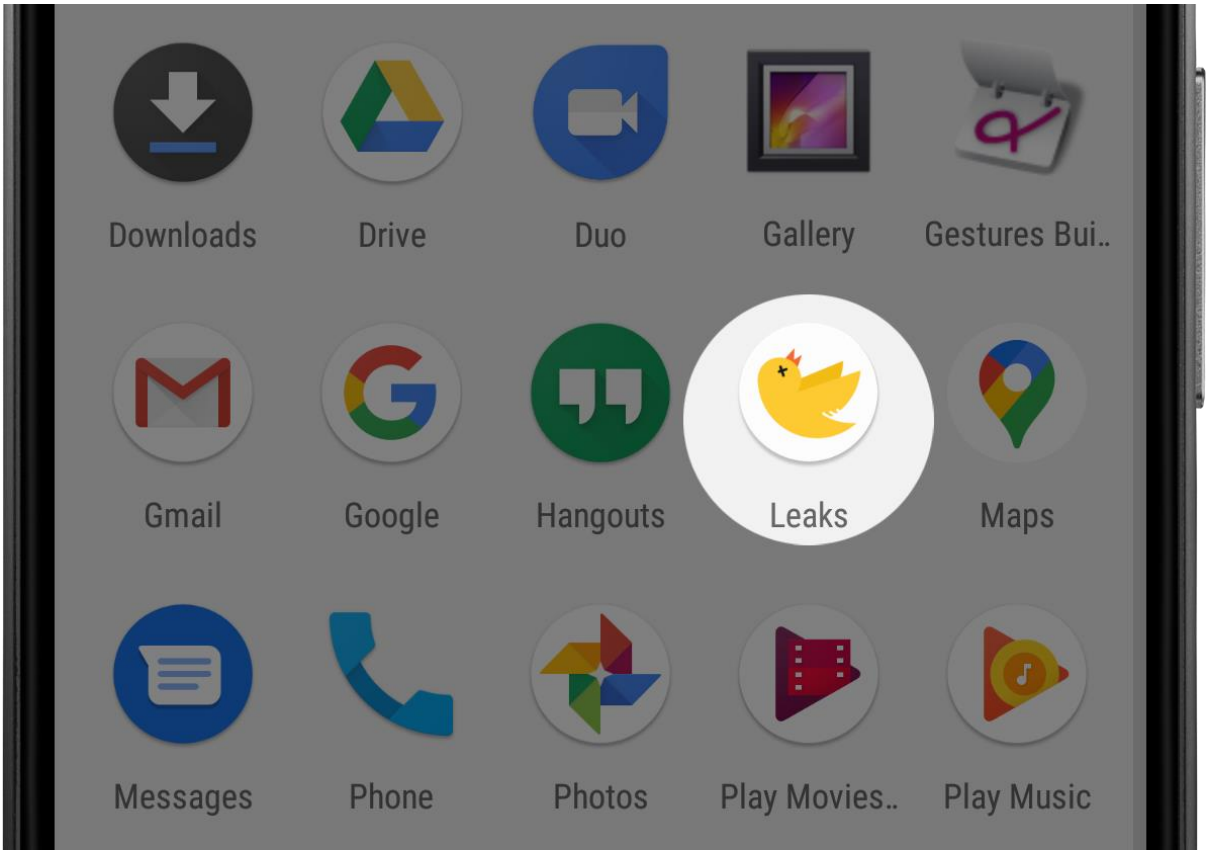
Hình 10: LeakCanary tính toán rò rỉ cho từng đối tượng

Khi phân tích xong, LeakCanary hiển thị thông báo kèm theo bản tóm tắt, đồng thời in kết quả trong Logcat. Lưu ý bên dưới cách 4 đối tượng được giữ lại được nhóm thành 2 rò rỉ riêng biệt. LeakCanary tạo ra một chữ ký cho mỗi dấu vết rò rỉ và nhóm lại các rò rỉ có cùng một chữ ký, tức là các rò rỉ do cùng một lỗi gây ra.



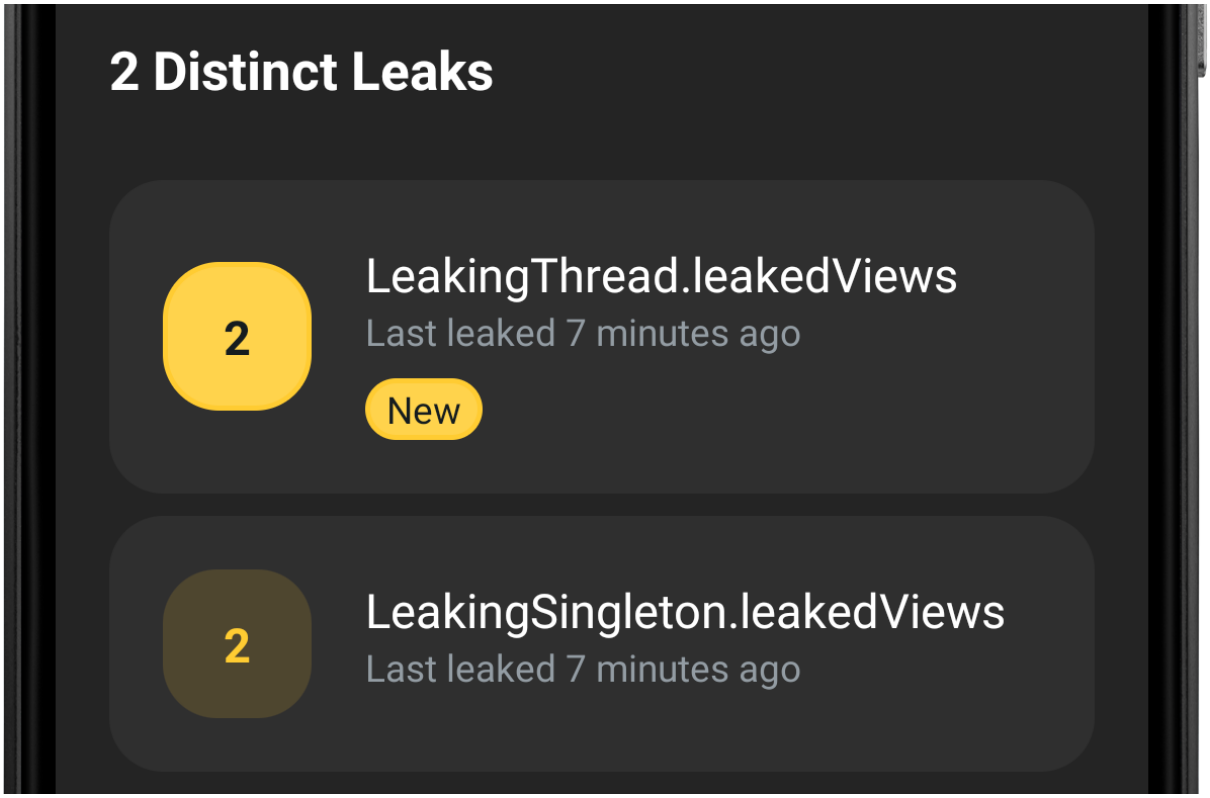
Hình 11: Phân loại 4 dấu vết rò rỉ thành 2 loại dấu vết rò rỉ

Nhấn vào thông báo sẽ bắt đầu một hoạt động cung cấp thêm thông tin chi tiết. Quay lại lần nữa bằng cách nhấn vào biểu tượng trình khởi chạy LeakCanary:



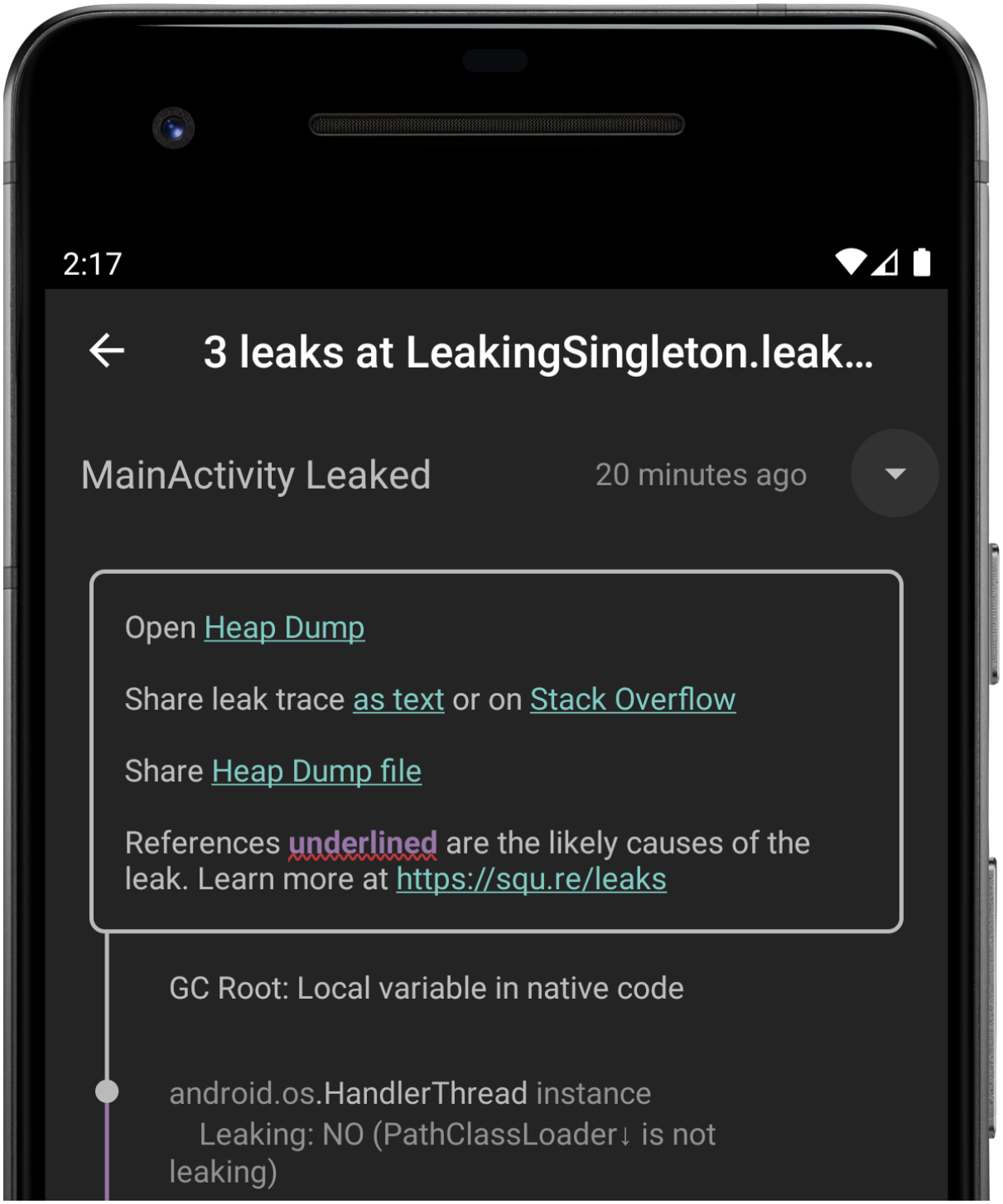
Hình 12: LeakCanary được cài cho mỗi ứng dụng

Mỗi hàng tương ứng với một nhóm rò rỉ có cùng một chữ ký. LeakCanary đánh dấu một hàng là lần đầu tiên ứng dụng kích hoạt rò rỉ với chữ ký đó.



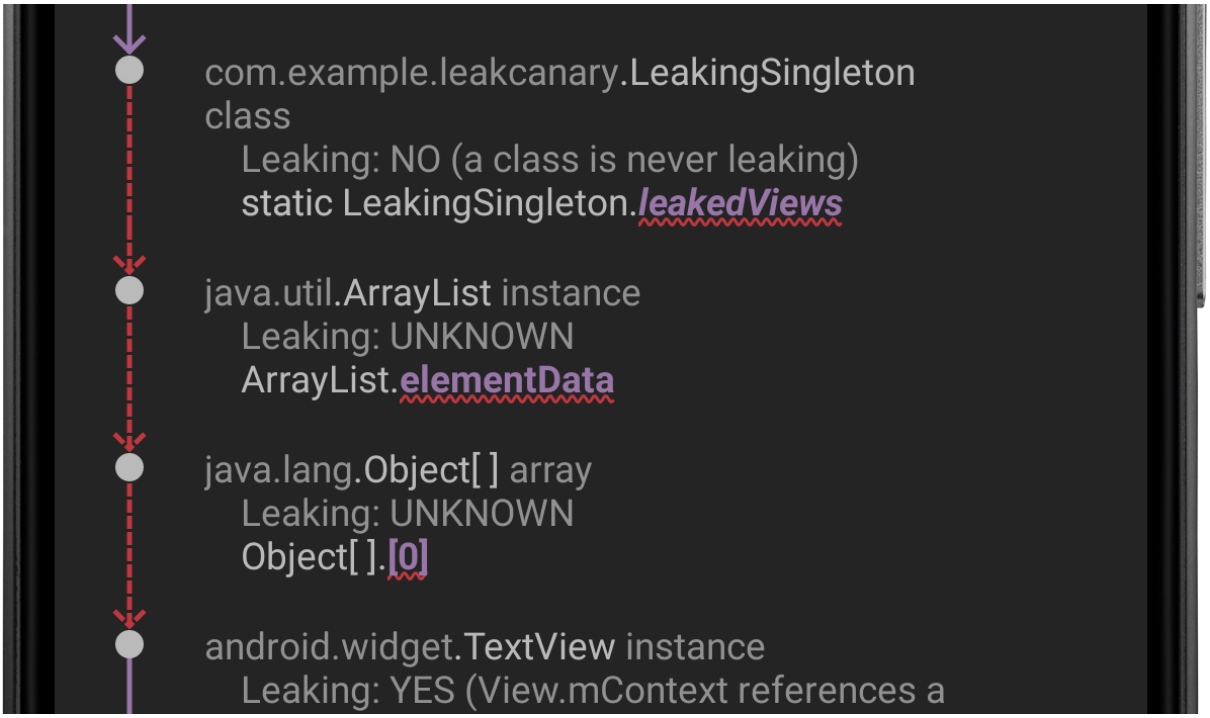
Hình 13: 4 rò rỉ được nhóm thành 2 hàng có dấu hiệu rò rỉ riêng biệt

Chạm vào vết rò rỉ để mở màn hình có dấu vết rò rỉ. Bạn có thể chuyển đổi giữa các đối tượng được giữ lại và dấu vết rò rỉ của chúng thông qua một menu thả xuống.



Hình 14: Một màn hình hiển thị 3 rò rỉ được nhóm theo chữ ký rò rỉ

Các chữ ký rò rỉ là hash của mỗi tham chiếu nghi ngờ gây ra rò rỉ, tức là mỗi tham chiếu hiển thị với một màu đỏ gạch dưới:



Hình 15: Một dấu vết rò rỉ với 3 tham chiếu đáng ngờ

Các tham chiếu đáng ngờ này được gạch dưới ~~~ khi dấu vết rò rỉ được chia sẻ dưới dạng văn bản:

```
...
|
├─ com.example.leakcanary.LeakingSingleton class
|    Leaking: NO (a class is never leaking)
|    ↓ static LeakingSingleton.leakedViews
|                              ~~~~~~~~~~~
├─ java.util.ArrayList instance
|    Leaking: UNKNOWN
|    ↓ ArrayList.elementData
|                 ~~~~~~~~~~
├─ java.lang.Object[] array
|    Leaking: UNKNOWN
|    ↓ Object[].[0]
|                ~~~
├─ android.widget.TextView instance
|    Leaking: YES (View.mContext references a destroyed activity)
...
```

Hình 16: Một dấu vết rò rỉ với 3 tham chiếu đáng ngờ

Trong ví dụ trên, chữ ký của rò rỉ sẽ được tính là:

```
val leakSignature = sha1Hash(
    "com.example.leakcanary.LeakingSingleton.leakedView" +
    "java.util.ArrayList.elementData" +
    "java.lang.Object[].[x]"
)
println(leakSignature)
// dbfa277d7e5624792e8b60bc950cd164190a11aa
```

Hình 17: Cách chữ ký rò rỉ được tính

❖ Categorizing leaks

LeakCanary phân chia các rò rỉ mà nó tìm thấy trong ứng dụng của bạn thành hai loại: Rò rỉ ứng dụng và Rò rỉ thư viện. Một thư viện rò rỉ là một sự rò rỉ gây ra bởi một lỗi do bên thứ 3 mà bạn không có quyền kiểm soát. Rò rỉ này đang ảnh hưởng đến ứng dụng của bạn, nhưng rất tiếc, việc sửa chữa nó có thể không nằm trong tầm kiểm soát của bạn nên LeakCanary sẽ tách nó ra.

Hai danh mục được phân tách trong kết quả được in trong Logcat:

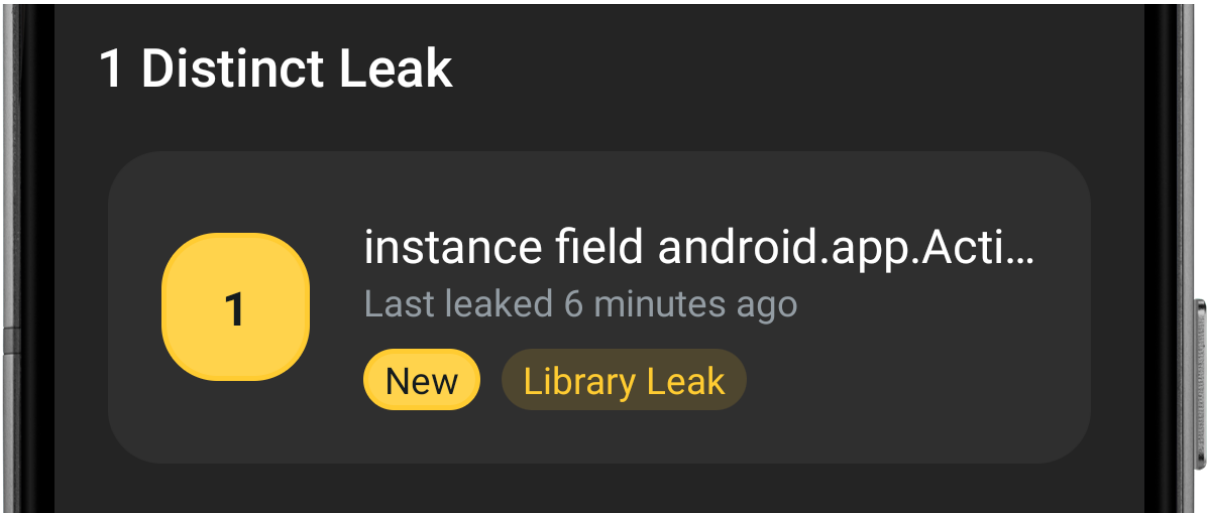
```
====================================
HEAP ANALYSIS RESULT
====================================
0 APPLICATION LEAKS

====================================
1 LIBRARY LEAK

...
┬───
│ GC Root: Local variable in native code
│
...
```

Hình 18: Logcat phân loại rò rỉ

LeakCanary đánh dấu một hàng là thư viện rò rỉ trong danh sách rò rỉ:



Hình 19: LeakCanary tìm thấy một thư viện bị rò rỉ

LeakCanary cung cấp một cơ sở dữ liệu về các rò rỉ đã biết, mà nó nhận ra bằng cách đối sánh mẫu trên các tên tham chiếu. Ví dụ:

```
Leak pattern: instance field android.app.Activity$1#this$0
Description: Android Q added a new IRequestFinishCallback$Stub class [...]
┬───
│ GC Root: Global variable in native code
│
├─ android.app.Activity$1 instance
│    Leaking: UNKNOWN
│    Anonymous subclass of android.app.IRequestFinishCallback$Stub
│    ↓ Activity$1.this$0
│                 ~~~~~~
╰→ com.example.MainActivity instance
```

Hình 20: Leak cung cấp dữ liệu về các rò rỉ đã biết

2.3.3. Khắc phục rò rỉ bộ nhớ

Rò rỉ bộ nhớ là một lỗi lập trình khiến ứng dụng giữ một tham chiếu đến một đối tượng không còn cần thiết. Ở đâu đó trong mã, có một tham chiếu đáng lẽ phải được xóa khi không cần thiết.

4 bước để sửa lỗi rò rỉ bộ nhớ:

1. Tìm dấu vết rò rỉ.
2. Thu hẹp các tham chiếu đáng ngờ.
3. Tìm tham chiếu gây ra rò rỉ.
4. Khắc phục sự cố rò rỉ.

LeakCanary giúp bạn thực hiện hai bước đầu tiên. Hai bước cuối cùng là tùy thuộc vào bạn!

❖ Tìm dấu vết rò rỉ

Một dấu vết rò rỉ là một cái tên ngắn hơn cho path tham chiếu mạnh nhất từ GC (Garbage Collector) root cho các đối tượng giữ lại, tức là path của tham chiếu mà đang nắm giữ một đối tượng trong bộ nhớ, do đó ngăn ngừa nó khỏi bị GC.

Ví dụ: hãy lưu trữ một singleton trợ giúp trong một trường tĩnh:

```
class Helper {
}

class Utils {
  public static Helper helper = new Helper();
}
```

```
AppWatcher.objectWatcher.watch(Utils.helper)
```

Dấu vết rò rỉ cho singleton đó trông như thế này:

```
┬───
│ GC Root: Local variable in native code
│
├─ dalvik.system.PathClassLoader instance
│    ↓ PathClassLoader.runtimeInternalObjects
├─ java.lang.Object[] array
│    ↓ Object[].[43]
├─ com.example.Utils class
│    ↓ static Utils.helper
╰→ java.example.Helper
```

Hình 21: Dấu vết rò rỉ

Ở trên cùng, một PathClassLoader cá thể được giữ bởi gốc Garbage Collector (GC), cụ thể hơn là một biến cục bộ trong mã gốc. GC root là những đối tượng đặc biệt luôn có thể tiếp cận được, tức là chúng không thể được thu gom. Có 4 loại gốc GC chính:

- Các biến cục bộ, thuộc về ngăn xếp của một luồng.
- Các phiên bản của các luồng Java đang hoạt động.
- Các lớp hệ thống, không bao giờ tải xuống.
- Tham chiếu gốc, được kiểm soát bởi mã gốc.

```
┬───
│ GC Root: Local variable in native code
│
├─ dalvik.system.PathClassLoader instance
```

Một dòng bắt đầu bằng ├─ đại diện cho một đối tượng Java (một lớp, một mảng đối tượng hoặc một thể hiện) và một dòng bắt đầu bằng │ ↓đại diện cho một tham chiếu đến đối tượng Java trên dòng tiếp theo.

PathClassLoader có một runtimeInternalObjects (Thời gian chạy đối tượng) là tham chiếu đến một mảng Object:

```
├─ dalvik.system.PathClassLoader instance
│    ↓ PathClassLoader.runtimeInternalObjects
├─ java.lang.Object[] array
```

Phần tử ở vị trí 43 trong mảng đó Object là một tham chiếu đến Utils lớp.

```
├─ java.lang.Object[] array
│    ↓ Object[].[43]
├─ com.example.Utils class
```

Một dòng bắt đầu bằng └→đại diện cho đối tượng bị rò rỉ, tức là đối tượng được chuyển đến AppWatcher.objectWatcher.watch () .

Các lớp Utils có một helper là một tham chiếu đến đối tượng rò rỉ, đó là Helper singleton ví dụ:

```
├─ com.example.Utils class
│    ↓ static Utils.helper
└─ java.example.Helper instance
```

❖ Thu hẹp các tham chiếu đáng ngờ

Dấu vết rò rỉ là một đường dẫn tham chiếu. Ban đầu, tất cả các tham chiếu trong đường dẫn đó bị nghi ngờ là nguyên nhân gây ra rò rỉ, nhưng LeakCanary có thể tự động thu hẹp các tham chiếu đáng ngờ. Để hiểu điều đó có nghĩa là gì, chúng ta hãy thực hiện quy trình đó theo cách thủ công.

Đây là một ví dụ về mã Android không tối ưu:

```
class ExampleApplication : Application() {
  val leakedViews = mutableLis<View>()
}

class MainActivity : Activity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_activity)

    val textView = findViewById<View>(R.id.helper_text)

    val app = application as ExampleApplication
    // This creates a leak, What a Terrible Failure!
    app.leakedViews.add(textView)
  }
}
```

Hình 22: Ví dụ về mã không tối ưu

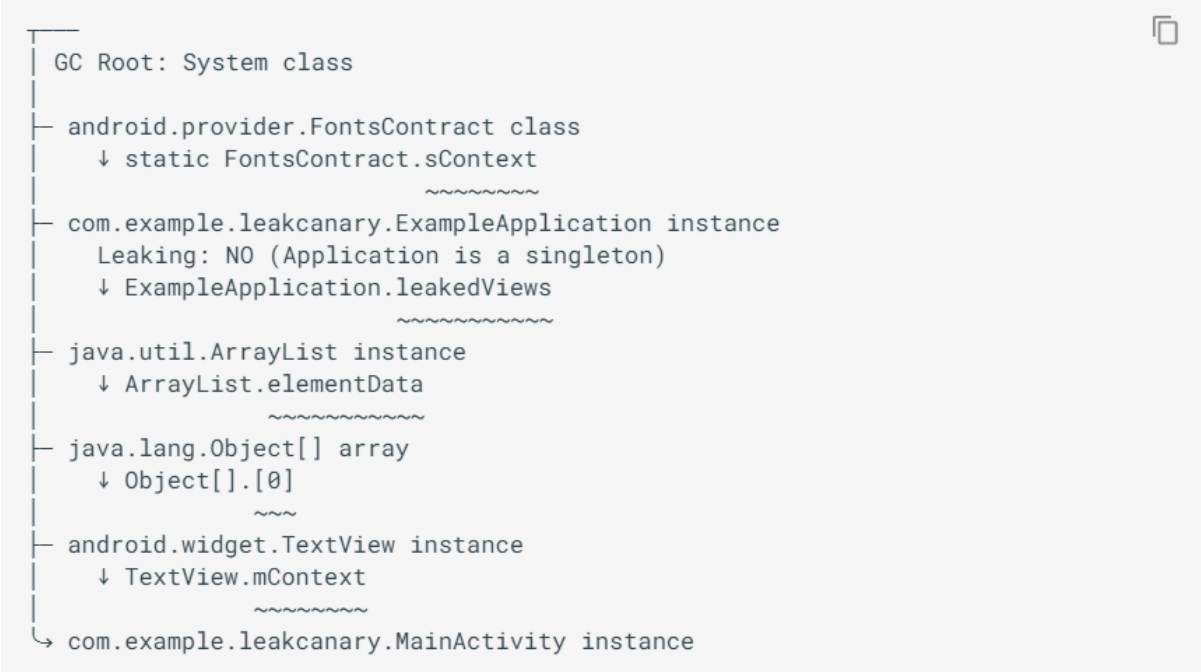
LeakCanary tạo ra một dấu vết rò rỉ trông giống như sau:

```
┬───
│ GC Root: System class
│
├─ android.provider.FontsContract class
│    ↓ static FontsContract.sContext
├─ com.example.leakcanary.ExampleApplication instance
│    ↓ ExampleApplication.leakedViews
├─ java.util.ArrayList instance
│    ↓ ArrayList.elementData
├─ java.lang.Object[] array
│    ↓ Object[].[0]
├─ android.widget.TextView instance
│    ↓ TextView.mContext
╰→ com.example.leakcanary.MainActivity instance
```

Hình 23: LeakCanary thông báo dấu vết rò rỉ

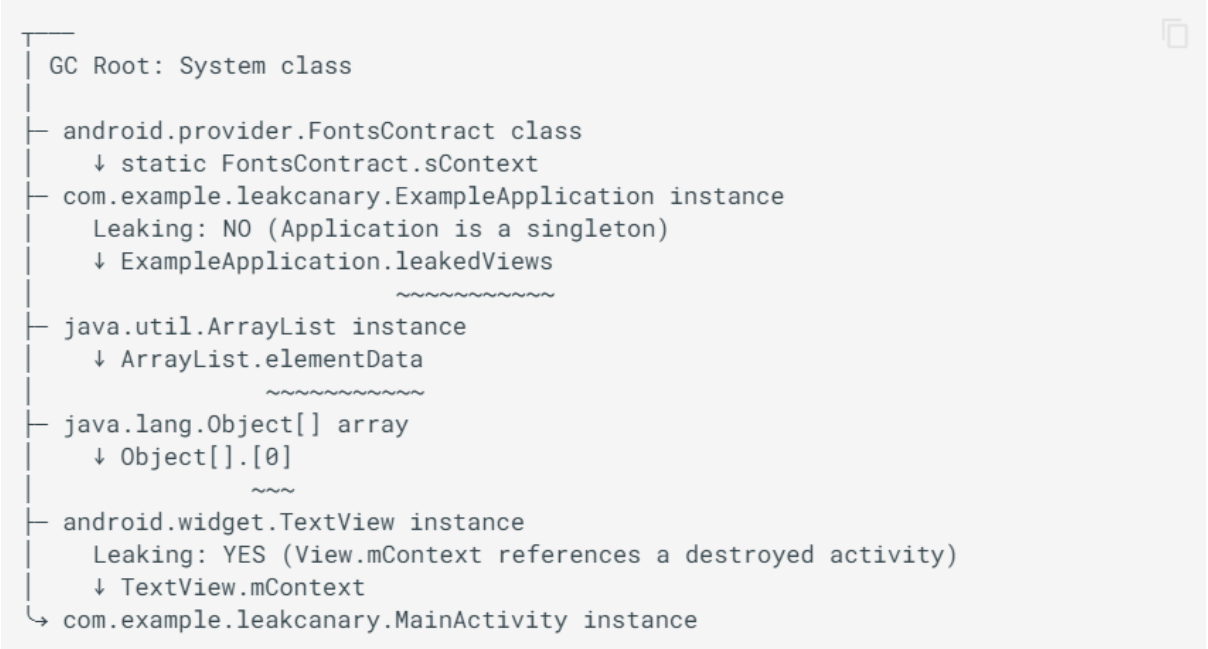
LeakCanary đánh dấu tất cả các tham chiếu bị nghi ngờ gây ra rò rỉ này bằng cách sử dụng gạch dưới ~~~. Ban đầu, tất cả các tham chiếu đều bị nghi ngờ:

Sau đó, LeakCanary thực hiện các suy luận về trạng thái và vòng đời của các đối tượng trong dấu vết rò rỉ. Trong một ứng dụng Android, Application ví dụ là một singleton không bao giờ được thu gom rác, vì vậy nó không bao giờ bị rò rỉ (Leaking: NO (Application is a singleton)). Từ đó, LeakCanary kết luận rằng rò rỉ không phải do FontsContract.sContext(loại bỏ tương ứng ~~~). Đây là dấu vết rò rỉ được cập nhật:



Hình 24: LeakCanary cập nhật dấu vết rò rỉ

Các TextView tham chiếu đến MainActivity bị phá hủy thông qua trường mContext của nó. Các khung nhìn không được tồn tại trong vòng đời của Activity của chúng, vì vậy LeakCanary biết rằng TextView này đang bị rò rỉ (Leaking: YES (View.mContext tham chiếu đến một đối tượng bị hủy)), và do đó việc rò rỉ không phải do TextView.mContext gây ra (loại bỏ tương ứng ~~~). Đây là dấu vết rò rỉ được cập nhật:



Hình 25: LeakCanary cập nhật dấu vết rò rỉ

Tóm lại, LeakCanary kiểm tra trạng thái của các đối tượng trong dấu vết rò rỉ để tìm ra liệu các đối tượng này có bị rò rỉ hay không (Leaking: YES so với Leaking: NO) và tận dụng thông tin đó để thu hẹp các tham chiếu đáng ngờ. Bạn có thể cung cấp các ObjectInspector triển khai tùy chỉnh để cải thiện cách LeakCanary hoạt động trong cơ sở mã của bạn.

❖ Tìm tham chiếu gây ra rò rỉ

Trong ví dụ trước, LeakCanary thu hẹp các tham chiếu bị nghi ngờ đến ExampleApplication.leakedViews, ArrayList.elementDatavà Object[].[0]:

```
┬───
│ GC Root: System class
│
├─ android.provider.FontsContract class
│    ↓ static FontsContract.sContext
├─ com.example.leakcanary.ExampleApplication instance
│    Leaking: NO (Application is a singleton)
│    ↓ ExampleApplication.leakedViews
│                         ~~~~~~~~~~~
├─ java.util.ArrayList instance
│    ↓ ArrayList.elementData
│                ~~~~~~~~~~~
├─ java.lang.Object[] array
│    ↓ Object[].[0]
│               ~~~
├─ android.widget.TextView instance
│    Leaking: YES (View.mContext references a destroyed activity)
│    ↓ TextView.mContext
╰→ com.example.leakcanary.MainActivity instance
```

Hình 26: LeakCanary thu hẹp tham chiếu rò rỉ

ArrayList.elementDatavà Object[].[0] những chi tiết thực hiện ArrayList, và nó không chắc rằng có một lỗi trong ArrayList, vì vậy các tham chiếu gây ra rò rỉ là tham chiếu duy nhất còn lại: ExampleApplication.leakedViews.

❖ Khắc phục sự cố rò rỉ

Khi tìm thấy tham chiếu gây ra rò rỉ, cần phải tìm hiểu tham chiếu đó nói về cái gì, khi nào đáng lẽ nó phải được xóa và tại sao nó không bị rò rỉ. Đôi khi nó hiển nhiên, giống như trong ví dụ trước. Đôi khi cần thêm thông tin để tìm ra nó.

2.4. Controlling Memory Leaks

Nhiều phương pháp khác nhau đã được đề xuất để giảm thiểu và loại bỏ rò rỉ bộ nhớ chẳng hạn như sử dụng LeakDAF, tạo các trường hợp thử nghiệm để xác định và sửa lỗi rò rỉ bộ nhớ bằng cách làm theo hướng dẫn. LeakDAF sử dụng các kỹ thuật kiểm tra giao diện người dùng để chạy ứng dụng và phân tích các tệp kết xuất bộ nhớ để xác định các activity và fragment bị rò rỉ. Sử dụng kỹ thuật Test Case để xác định và sửa lỗi rò rỉ bộ nhớ có được các nhà nghiên cứu đề xuất theo hai cách riêng biệt như sau:

2.4.1. Ưu tiên các Test case

Sử dụng một cách tiếp cận để ưu tiên các trường hợp thử nghiệm và chạy các trường hợp đó trong thứ tự cụ thể thay vì chạy tất cả các trường hợp thử nghiệm khi họ có thể tốn kém và dẫn đến việc đặt nhiều tải lên CPU. Mức độ ưu tiên của các trường hợp kiểm thử được xác định bằng cách triển khai các thuật toán học máy dự đoán độ chính xác của một bài kiểm tra trường hợp để xác định rò rỉ bộ nhớ.

2.4.2. Tạo Test case để phát hiện rò rỉ bộ nhớ

Đề xuất một kỹ thuật xác định một chuỗi tự nhiên của Các sự kiện GUI như khởi chạy và đóng ứng dụng. Sự lặp lại như vậy các sự kiện sẽ không làm tăng mức sử dụng bộ nhớ nếu không có rò rỉ bộ nhớ. Nếu có rò rỉ, bộ nhớ sẽ tiếp tục ngày càng tăng.

2.5. Managing GPU Buffers

GPU là Đơn vị xử lý giống như CPU nhưng được sử dụng để xử lý đồ họa. Cùng với thời gian, các ứng dụng và trò chơi phát triển và trở nên đồ họa mạnh mẽ, vì vậy cần một GPU chuyên dụng là bắt buộc. Các thiết bị điện thoại thông minh được trang bị với GPU giống như máy tính để bàn nhưng có một sự khác biệt giữa các GPU của họ. GPU lớp dành cho máy tính để bàn có bộ nhớ chuyên dụng nhưng do kích thước nhỏ và tính di động của thiết bị điện thoại thông minh, GPU điện thoại thông minh không có bộ nhớ nhưng chia sẻ bộ nhớ với bộ nhớ chính cách này làm giảm bộ nhớ chính khả dụng. Android lưu trữ các ứng dụng

để chúng nhanh chóng được khởi chạy lần sau khi được gọi. Khi được lưu vào bộ nhớ đệm, bộ đệm GPU sẽ cũng được lưu vào bộ nhớ đệm vì các ứng dụng chứa đồ họa. Khi ứng dụng kết thúc, dữ liệu đã lưu trong bộ nhớ cache của nó vẫn còn bên trong bộ nhớ chính, do đó tăng mức sử dụng bộ nhớ và giảm khả năng sử dụng lại. Một trong những giải pháp hiện tại là nén bộ đệm GPU khi ứng dụng được chạy nền và giải nén khi ứng dụng được khởi chạy. Bộ nhớ nén sẽ mất một ít dung lượng hơn và sẽ cải thiện hiệu suất tổng thể của thiết bị. Nén và giải nén này sẽ tăng thời gian truy cập khiến giải pháp này kém tối ưu hơn.

2.6. Avoiding Memory Duplication

Nhân bản bộ nhớ là một quá trình trong đó cùng một bộ nhớ các trang được đặt trong bộ nhớ chính nhiều hơn một lần. Qua sử dụng kỹ thuật sao chép này, khi Bộ thu gom rác giải phóng dung lượng, cần nhiều chu kỳ CPU gây ra độ trễ và thậm chí đôi khi ứng dụng gặp sự cố. Vì vậy, trùng lặp trang phải được tránh để Bộ thu gom rác không phá hủy cùng nội dung nhiều lần. Để khắc phục những sự cố như vậy, Android có đã giới thiệu một số cơ chế như zRAM và Kernel.

Hợp nhất cùng trang (KSM). KSM hợp nhất các trang trùng lặp vào một. zRAM sử dụng một vùng được cấp phát đặc biệt trong bộ nhớ chính được gọi là vùng hoán đổi, nơi nó nén các trang được lưu trữ cần được hoán đổi. Cả hai cơ chế này giảm mức sử dụng bộ nhớ nhưng tiêu thụ số lượng CPU lớn hơn chu kỳ và quyền lực. Memscope là một công cụ có thể được sử dụng để tránh các vấn đề. Nó chụp nhanh bộ nhớ vào những khoảng thời gian cố định. Nó xác định các khung trùng lặp có thể tồn tại trong một thời gian và chúng sẽ thay đổi như thế nào trong vòng đời của ứng dụng. Nó phân tích các ảnh chụp nhanh và tìm ra các khung hình có cơ hội sao chép và tập trung vào các khung đó để tránh trùng lặp. Một phương pháp khác hiện được sử dụng để giảm thiểu và loại bỏ nhân bản bộ nhớ là sao chép bộ nhớ có chọn lọc. Thay vì quét toàn bộ bộ nhớ, nó sẽ quét một số trang bộ nhớ để trùng lặp làm giảm mức sử dụng CPU. Cơ chế này quét các trang bộ nhớ của các ứng dụng trong nền chỉ một lần cho đến khi chúng được đưa

vào nền một lần nữa vì rất khó có khả năng dấu chân bộ nhớ sẽ thay đổi cho các ứng dụng trong nền.

2.7. Adaptive Background App Management

Tối ưu hóa vi mô phải được áp dụng trên các ứng dụng điện thoại thông minh vì họ dễ gặp các vấn đề về hiệu suất hơn. Vi mô tối ưu hóa nên được áp dụng khi bắt đầu vòng đời của ứng dụng. Phát hiện, loại bỏ các biến không sử dụng và các phương thức riêng tư cải thiện hiệu suất vì nó giảm dung lượng bộ nhớ.

Thông thường các nhà phát triển không áp dụng các tối ưu hóa vi mô tại vì:

1. Hầu hết trong số họ không biết về điều này.
2. Họ nghĩ rằng ứng dụng quá nhỏ để áp dụng bất kỳ tối ưu hóa.
3. Họ không nghĩ dành thời gian cho vi sự lạc quan là xứng đáng.
4. Họ không tin rằng những tối ưu hóa vi mô sẽ trợ giúp các ứng dụng của họ.

Kết quả được tìm thấy bằng cách sử dụng phân tích tĩnh được thực hiện bởi sử dụng các công cụ như FindBugs, PMD và LINT cung cấp cảnh báo để cải thiện mã. Một trong những điều hữu ích nhất tối ưu hóa được tìm thấy là loại bỏ mã không sử dụng. Nhưng như vậy các công cụ và kỹ thuật dễ bị lỗi và không thể tin cậy được hoàn toàn.

2.8. Detecting Bad Programming Practices

Một giải pháp khác được các nhà nghiên cứu đề xuất là công cụ CheckDroid được sử dụng để xác định các phương pháp lập trình không hợp lệ. Sửa chữa những điều này có thể cải thiện hiệu suất ứng dụng và bộ nhớ tổng thể của hệ điều hành Android. Một công cụ như vậy là cần thiết bởi vì các phương pháp xấu thường không được IDE xác định và báo cáo. Hầu hết các các công cụ và kỹ thuật được áp dụng là để phát hiện bộ nhớ và rò rỉ hiệu suất nhưng các hoạt động xấu vẫn chưa được khắc phục.

Một số khuyến nghị về Hiệu suất sau khi phân tích là:

1. Ghi nhật ký chi tiết không nên để trực tiếp các ứng dụng.

2. Các nhiệm vụ chạy dài nên được chia thành các nhiệm vụ phụ.
3. Các nhiệm vụ phụ này nên có mức độ ưu tiên thấp hơn luồng chính.

Một số cách tối ưu chính được đề xuất là:

1. Tham chiếu đến ngữ cảnh không nên được lưu trữ trong tĩnh biến.
2. Các luồng được tạo nên bị hủy.

2.9. Partitioning Memory

Một đề xuất khác là phân vùng bộ nhớ chính vì Android sử dụng LMK (Low Memory Killer) và OOMK (Out Of Memory Killer) để giải phóng bộ nhớ bằng cách phá hủy các quy trình không hiệu quả lắm. Vì vậy, một kỹ thuật phân vùng bộ nhớ được đề xuất để phân chia bộ nhớ trong hai nút ảo.

1. Virtual Node 0 được sử dụng cho các ứng dụng đáng tin cậy.

2. Virtual Node 1 được sử dụng cho các ứng dụng không đáng tin cậy. Nếu bộ nhớ hết một nút, chỉ bộ nhớ của nút đó sẽ được giải phóng. Thông thường, các ứng dụng không đáng tin cậy chiếm nhiều hơn bộ nhớ hơn các ứng dụng đáng tin cậy. Bằng cách tuân theo phương pháp luận này, bộ nhớ có thể được tiết kiệm phần nào khi hết.

2.10. Handling Low Memory Using Logs

Kỹ thuật này đề xuất sử dụng các bản ghi đã tạo để xác định tương tác của người dùng và thời gian dành cho ứng dụng để động thay đổi mức độ ưu tiên của các ứng dụng. Điều này sẽ cho phép giữ mức cao các ứng dụng ưu tiên trong bộ nhớ cache và chỉ loại bỏ các ứng dụng có mức độ ưu tiên thấp khi sự cố bộ nhớ thấp xảy ra.

2.11. Detecting Anti-Patterns

Để cố gắng phát triển ứng dụng nhanh hơn, các nhà phát triển ngày nay có xu hướng đi chệch khỏi các mẫu lập trình được gọi là phản các mẫu dẫn đến thiết kế kém và do đó hiệu suất ứng dụng kém. Paprika là một công cụ được đề xuất phân tích mã để xác định các mô hình chống và đề xuất các giải pháp để khắc phục nó. OOP là khối xây dựng cơ bản củaphát triển bất kỳ loại ứng dụng nào, nó cung cấp khả năng tái sử dụng và cáccác chức năng mà trước đây không thể

thực hiện được. Ứng dụng Android chứa tệp .dex chứa java đã biên dịch các lớp học. Android chạy trên Máy ảo Darvik và bytecode khác với Java. Nhiều công cụ có sẵnđể đảo ngược các tệp .dex. Paprika lần đầu tiên trích xuất siêu dữ liệu từ apk chẳng hạn như ứng dụngtên, số nhận dạng gói và đánh giá của người dùng rồi mãcác thực thể như lớp, phương thức và tên biến cũng được trích xuất. Bằng cách sử dụng các thực thể được trích xuất, một mô hình mã làđược tính dưới dạng đồ thị với các giá trị thô. Mô hình này được lưu trữ trongcơ sở dữ liệu đồ thị và sau đó cơ sở dữ liệu này được truy vấn để phát hiệnchống các mẫu. Những mô hình chống này khi cố định có thể dẫn đến khámột chút bộ nhớ cần được giải phóng.

2.12. Limiting Software Aging

Lão hóa phần mềm là một quá trình trong đó hiệu suất của hệ điều hành và ứng dụng xuống cấp theo thời gian. Hầu hết tất cả các vấn đề là bộ nhớ bị rò rỉ. Với sự lão hóa, bộ nhớ trống bị giảm nên ít ứng dụng hơn được lưu trong bộ nhớ cache và khi ứng dụng mới được khởi chạy, bộ nhớ cần phải giải phóng sẽ làm tốn chu kỳ CPU và pin. Để phát hiện và xác định sự lão hóa, số liệu thống kê về sử dụng tài nguyên cần được lưu giữ. Để điều tra rò rỉ bộ nhớ, thiết bị cần được kiểm tra theo điều kiện nghiêm trọng, nơi nó dễ bị hỏng. Đối với điều này, Các bài kiểm tra được chia thành các bài kiểm tra phụ với mỗi bài kiểm tra phụ thuộc vào sản lượng của cái trước. Nếu quá trình lão hóa được phát hiện, thử nghiệm tiếp theo sẽ được chạy trong một thời gian dài hơn. Các bài kiểm tra này đã giúp xác định mức độ rò rỉ bộ nhớ trong các ứng dụng.

2.13. Non-Blocking Garbage Collector

Bộ thu gom rác hoạt động trên nguyên tắc “dừng dòng điện thực hiện, đánh dấu các đối tượng để xóa và quét chúng ra khỏi bộ nhớ” có thể dẫn đến các hành vi không phản hồi và thậm chí sự cố. Để khắc phục những sự cố này, một bộ thu gom Rác theo thời gian thực được đề xuất bởi các nhà nghiên cứu có hai đặc trưng:

- GC này sẽ hoạt động tăng dần với thời gian ngắn giai đoạn ngăn chặn.
- Tốc độ của GC phải phù hợp với rác do HĐH tạo ra để tránh Out of Memory situation. Điều này sẽ cho phép giải phóng bộ nhớ nhanh chóng so với Bộ thu gom rác không chặn và sẽ cải thiện tổng thể hiệu suất của hệ điều hành Android.

2.14. Use of Non-Volatile Memory

Theo một nghiên cứu, hầu hết người dùng có ít phiên ứng dụng hơn 10 giây và trong trường hợp đó, thời gian tải ứng dụng ngắn hơn là cần thiết. Nhiều giải pháp đã được đề xuất để cải thiện thời gian khởi chạy ứng dụng nhưng hầu hết các giải pháp không phải là phần cứng dựa trên. Giải pháp được các nghiên cứu đề xuất là sử dụng Non-Volatile Memory (NVM) hoặc cụ thể hơn là Phase Change.

Phase Change Memory (PCM) của bộ nhớ chính. Bộ nhớ NVM mới là phổ biến vì nó tiêu thụ ít điện năng hơn. PCM nhanh, rất tiết kiệm năng lượng cho các hoạt động đọc nhưng tiêu thụ nhiều quyền lực trong các hoạt động ghi và rất chậm. Do đó, một DRAM- PCM giải pháp lai được đề xuất để cải thiện ứng dụng thời gian khởi động. Một nghiên cứu đã được thực hiện để phân tích các ứng dụng dựa trên bộ nhớ và người ta thấy rằng các ứng dụng thuộc hai loại danh mục:

- Ứng dụng ổn định trong đó bộ nhớ tăng lên đến 10 đầu tiên giây sau đó sẽ ổn định.
- Các ứng dụng không ổn định trong đó bộ nhớ không ngừng tăng lênvới thời gian và không bao giờ ổn định. Lưu ý đến cả hai loại ứng dụng, giải pháp được đề xuất là việc sử dụng NVM làm bản sao lưu của bộ nhớ chính. Khi GC đátrong và xóa dữ liệu khỏi bộ nhớ chính, dữ liệu làđã chuyển sang NVM. Các vùng đặc biệt được chỉ định cho NVMnơi lưu trữ dữ liệu của các ứng dụng được sử dụng nhiều nhất và được chia sẻ chungcác thư viện. Điều này sẽ cho phép thời gian khởi chạy tốt hơn ngay cả khi ứng dụng

không được lưu trong bộ nhớ chính, nhưng chúng sẽ hiện diện trong bộ nhớ dự phòng.

Tất cả các giải pháp này đều ít nhiều hướng đến việc tối ưu hóa hệ thống quản lý bộ nhớ đệm và bộ nhớ đã có trongAndroid nhưng không ai trong số họ thực sự xử lý những vấn đề này với giải pháp thích hợp và được xác định rõ ràng.

Chương 3: Giải pháp tối ưu hóa bộ nhớ

Một số tính năng Android, lớp Java và cấu trúc mã có xu hướng sử dụng nhiều bộ nhớ hơn những tính năng khác. Bạn có thể giảm thiểu dung lượng bộ nhớ mà ứng dụng sử dụng bằng cách chọn các giải pháp thay thế hiệu quả hơn trong mã của bạn.

3.1. Sử dụng các services một cách tiết kiệm

Để một service chạy khi không cần thiết là một trong những lỗi quản lý bộ nhớ tồi tệ nhất mà ứng dụng Android có thể mắc phải. Nếu ứng dụng của bạn cần một dịch vụ để thực hiện công việc trong nền, đừng để ứng dụng tiếp tục chạy trừ khi ứng dụng cần chạy một công việc. Hãy nhớ dừng dịch vụ của bạn khi nó đã hoàn thành nhiệm vụ. Nếu không, bạn có thể vô tình gây ra rò rỉ bộ nhớ.

Khi bạn bắt đầu một dịch vụ, hệ thống ưu tiên luôn duy trì quá trình cho dịch vụ đó hoạt động. Hành vi này làm cho các quá trình dịch vụ trở nên rất tốn kém vì RAM được sử dụng bởi một dịch vụ vẫn không khả dụng cho các quá trình khác. Điều này làm giảm số lượng các quy trình được lưu trong bộ nhớ cache mà hệ thống có thể giữ trong bộ nhớ cache LRU, làm cho việc chuyển đổi ứng dụng kém hiệu quả hơn. Nó thậm chí có thể dẫn đến sự cố trong hệ thống khi bộ nhớ bị thắt chặt và hệ thống không thể duy trì đủ quy trình để lưu trữ tất cả các dịch vụ hiện đang chạy.

Nói chung, bạn nên tránh sử dụng các dịch vụ liên tục vì các yêu cầu liên tục mà chúng đặt ra trên bộ nhớ khả dụng. Thay vào đó, chúng tôi khuyên bạn nên sử dụng một triển khai thay thế như JobScheduler. Để biết thêm thông tin về cách sử dụng JobScheduler để lập lịch các quy trình nền, hãy xem Tối ưu hóa nền.

Nếu bạn phải sử dụng một dịch vụ, cách tốt nhất để giới hạn tuổi thọ của dịch vụ là sử dụng IntentService, IntentService sẽ tự hoàn thiện ngay sau khi xử lý xong ý định bắt đầu.

3.2. Sử dụng các vùng chứa dữ liệu được tối ưu hóa

Một số lớp được cung cấp bởi ngôn ngữ lập trình không được tối ưu hóa để sử dụng trên thiết bị di động. Ví dụ, việc triển khai HashMap chung có thể khá kém hiệu quả về bộ nhớ vì nó cần một đối tượng mục nhập riêng biệt cho mọi ánh xạ.

Android framework bao gồm một số vùng chứa dữ liệu được tối ưu hóa, bao gồm SparseArray, SparseBooleanArray và LongSparseArray. Ví dụ, các lớp SparseArray hiệu quả hơn vì chúng tránh được việc hệ thống phải tự động đóng hộp khóa và đôi khi giá trị (tạo ra một hoặc hai đối tượng khác cho mỗi mục nhập).

Nếu cần, bạn luôn có thể chuyển sang mảng thô để có cấu trúc dữ liệu thực sự tinh gọn.

3.3. Cẩn thận với các code abstractions

Các nhà phát triển thường sử dụng trừu tượng đơn giản như một phương pháp lập trình tốt, bởi vì trừu tượng có thể cải thiện tính linh hoạt và bảo trì của mã. Tuy nhiên, sự trừu tượng đi kèm với một cái giá đáng kể: nói chung chúng yêu cầu một lượng lớn mã cần được thực thi hơn, đòi hỏi nhiều thời gian hơn và nhiều RAM hơn để mã đó được ánh xạ vào bộ nhớ. Vì vậy, nếu các món ăn kiêng của bạn không mang lại lợi ích đáng kể, bạn nên tránh chúng.

3.4. Sử dụng protobufs lite cho dữ liệu được tuần tự hóa

Bộ đệm giao thức là một cơ chế trung lập về ngôn ngữ, trung lập về nền tảng, có thể mở rộng được Google thiết kế để tuần tự hóa dữ liệu có cấu trúc — tương tự như XML, nhưng nhỏ hơn, nhanh hơn và đơn giản hơn. Nếu bạn quyết định sử dụng protobufs cho dữ liệu của mình, bạn nên luôn sử dụng protobufs lite trong mã phía máy khách của mình. Protobufs thông thường tạo ra mã cực kỳ dài dòng, có thể gây ra nhiều loại vấn đề trong ứng dụng của bạn như tăng mức sử dụng RAM, tăng kích thước APK đáng kể và thực thi chậm hơn.

3.5. Tránh xáo trộn bộ nhớ

Như đã đề cập trước đây, các sự kiện thu gom rác không ảnh hưởng đến hiệu suất ứng dụng của bạn. Tuy nhiên, nhiều sự kiện thu gom rác xảy ra trong một khoảng thời gian ngắn có thể nhanh chóng làm tiêu hao pin cũng như tăng nhẹ thời gian thiết lập khung hình do các tương tác cần thiết giữa trình thu gom rác và chủ đề ứng dụng. Hệ thống dành càng nhiều thời gian cho việc thu gom rác, pin càng cạn kiệt nhanh hơn.

Thông thường, xáo trộn bộ nhớ có thể khiến một số lượng lớn các sự kiện thu gom rác xảy ra. Trong thực tế, xáo trộn bộ nhớ mô tả số lượng các đối tượng tạm thời được cấp phát xảy ra trong một khoảng thời gian nhất định.

Ví dụ: bạn có thể cấp phát nhiều đối tượng tạm thời trong vòng lặp for. Hoặc bạn có thể tạo các đối tượng Paint hoặc Bitmap mới bên trong hàm onDraw () của một dạng xem. Trong cả hai trường hợp, ứng dụng tạo ra nhiều đối tượng một cách nhanh chóng với âm lượng lớn. Những thứ này có thể nhanh chóng tiêu thụ tất cả bộ nhớ có sẵn trong thế hệ trẻ, buộc phải xảy ra sự kiện thu gom rác.

Tất nhiên, bạn cần phải tìm những vị trí trong mã của mình mà bộ nhớ bị xáo trộn cao trước khi có thể sửa chúng. Đối với điều đó, bạn nên sử dụng Hồ sơ bộ nhớ trong Android Studio.

Khi bạn xác định các khu vực có vấn đề trong mã của mình, hãy cố gắng giảm số lượng phân bổ trong các khu vực quan trọng về hiệu suất. Cân nhắc chuyển mọi thứ ra khỏi vòng lặp bên trong hoặc có thể chuyển chúng vào cấu trúc phân bổ dựa trên Factory.

Một khả năng khác là đánh giá xem các nhóm đối tượng có mang lại lợi ích cho trường hợp sử dụng hay không. Với một nhóm đối tượng, thay vì thả một cá thể đối tượng xuống sàn, bạn giải phóng nó vào một nhóm khi nó không còn cần thiết nữa. Khi cần đến một cá thể đối tượng của kiểu đó vào lần tiếp theo, nó có thể được lấy từ pool, thay vì phân bổ nó.

Đánh giá hiệu suất kỹ lưỡng là điều cần thiết để xác định xem một nhóm đối tượng có phù hợp trong một tình huống nhất định hay không. Có những trường

hợp các nhóm đối tượng có thể làm cho hiệu suất kém hơn. Mặc dù các nhóm tránh phân bổ, họ đưa ra các khoản chi phí khác. Ví dụ: duy trì pool thường liên quan đến việc đồng bộ hóa có chi phí không đáng kể. Ngoài ra, việc xóa cá thể đối tượng gộp (để tránh rò rỉ bộ nhớ) trong quá trình phát hành và sau đó việc khởi tạo nó trong quá trình thu có thể có chi phí khác không. Cuối cùng, việc giữ lại nhiều cá thể đối tượng trong nhóm hơn mong muốn cũng đặt ra gánh nặng cho GC. Trong khi các nhóm đối tượng làm giảm số lượng lệnh gọi GC, chúng kết thúc bằng việc tăng khối lượng công việc cần thực hiện trên mỗi lần gọi, vì nó tỷ lệ với số byte trực tiếp (có thể truy cập).

3.6. Xóa các tài nguyên và thư viện sử dụng nhiều bộ nhớ

Một số tài nguyên và thư viện trong mã của bạn có thể chiếm bộ nhớ mà bạn không biết. Kích thước tổng thể của ứng dụng, bao gồm cả thư viện của bên thứ ba hoặc tài nguyên được nhúng, có thể ảnh hưởng đến lượng bộ nhớ mà ứng dụng của bạn sử dụng. Bạn có thể cải thiện mức tiêu thụ bộ nhớ của ứng dụng bằng cách xóa mọi thành phần, tài nguyên hoặc thư viện dư thừa, không cần thiết hoặc cồng kềnh khỏi mã của bạn.

3.7. Giảm kích thước APK tổng thể

Bạn có thể giảm đáng kể mức sử dụng bộ nhớ của ứng dụng bằng cách giảm kích thước tổng thể của ứng dụng. Kích thước bitmap, tài nguyên, khung hoạt ảnh và thư viện của bên thứ ba đều có thể góp phần vào kích thước ứng dụng của bạn. Android Studio và Android SDK cung cấp nhiều công cụ để giúp bạn giảm quy mô tài nguyên và các yếu tố phụ thuộc bên ngoài. Những công cụ này hỗ trợ các phương pháp thu nhỏ mã hiện đại, chẳng hạn như biên dịch R8. (Android Studio 3.3 trở xuống sử dụng ProGuard thay vì biên dịch R8.)

3.8. Dependency injection frameworks

Dependency injection frameworks có thể đơn giản hóa mã bạn viết và cung cấp một môi trường thích ứng hữu ích cho việc thử nghiệm và các thay đổi cấu hình khác. Nếu bạn định sử dụng dependency injection frameworks trong ứng

dụng của mình, hãy cân nhắc sử dụng Dagger 2. Dagger không sử dụng phản chiếu để quét mã ứng dụng của bạn. Triển khai tĩnh, thời gian biên dịch của Dagger có nghĩa là nó có thể được sử dụng trong các ứng dụng Android mà không cần tốn thời gian chạy hoặc sử dụng bộ nhớ. Các dependency injection frameworks khác sử dụng phản chiếu có xu hướng khởi tạo các quy trình bằng cách quét mã của bạn để tìm các chú thích. Quá trình này có thể yêu cầu nhiều chu kỳ CPU và RAM hơn đáng kể và có thể gây ra độ trễ đáng kể khi ứng dụng khởi chạy.

3.9. Hãy cẩn thận khi sử dụng các thư viện bên ngoài

Mã thư viện bên ngoài thường không được viết cho môi trường di động và có thể không hiệu quả khi được sử dụng cho công việc trên máy khách di động. Khi bạn quyết định sử dụng thư viện bên ngoài, bạn có thể cần phải tối ưu hóa thư viện đó cho các thiết bị di động. Lập kế hoạch cho công việc đó từ trước và phân tích thư viện về kích thước mã và dung lượng RAM trước khi quyết định sử dụng nó.

Ngay cả một số thư viện được tối ưu hóa cho thiết bị di động cũng có thể gây ra sự cố do các cách triển khai khác nhau. Ví dụ: một thư viện có thể sử dụng các protobuf nhẹ trong khi một thư viện khác sử dụng các protobuf vi mô, dẫn đến hai cách triển khai protobuf khác nhau trong ứng dụng của bạn. Điều này có thể xảy ra với các triển khai khác nhau của ghi nhật ký, phân tích, khung tải hình ảnh, bộ nhớ đệm và nhiều thứ khác mà bạn không mong đợi.

Mặc dù ProGuard có thể giúp loại bỏ các API và tài nguyên có cờ phù hợp, nhưng nó không thể loại bỏ các phần phụ thuộc nội bộ lớn của thư viện. Các tính năng mà bạn muốn trong các thư viện này có thể yêu cầu các phụ thuộc cấp thấp hơn. Điều này trở nên đặc biệt có vấn đề khi bạn sử dụng lớp con Activity từ một thư viện (sẽ có xu hướng có nhiều phụ thuộc), khi các thư viện sử dụng phản chiếu (điều này phổ biến và có nghĩa là bạn cần phải dành nhiều thời gian để tinh chỉnh ProGuard theo cách thủ công để làm cho nó làm việc), và như vậy.

Cũng tránh sử dụng thư viện được chia sẻ chỉ cho một hoặc hai tính năng trong số hàng chục tính năng. Bạn không muốn lấy một lượng lớn mã và chi phí mà bạn thậm chí không sử dụng. Khi bạn cân nhắc có nên sử dụng thư viện hay không, hãy tìm cách triển khai phù hợp nhất với những gì bạn cần. Nếu không, bạn có thể quyết định tạo triển khai của riêng mình.

Chương 4: Tiến hành thực nghiệm tối ưu hóa bộ nhớ

4.1. AsyncTaskActivity

Khi nhấn vào button AsyncTask ứng dụng sẽ chuyển từ activity_main sang activity_hello_world.

Ứng dụng sẽ bị rò rỉ khi chúng ta xoay hoặc đóng activity_hello_world trong vòng 10 giây khi activity_hello_world được tạo ra.

Nếu chúng ta xoay hoặc đóng activity_hello_world sau 10 giây khi activity_hello_world được khởi tạo thì ứng dụng sẽ không bị rò rỉ.

Phía bên dưới là code của class AsyncTaskActivity gây ra rò rỉ.

```
public class AsyncTaskActivity extends Activity {
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world);
        textView = findViewById(R.id.text_view);

        new DownloadTask().execute();
    }

    public static void start(Context context) {
        Intent starter = new Intent(context, AsyncTaskActivity.class);
        context.startActivity(starter);
    }

    public void updateText() {
        textView.setText(R.string.hello);
    }

    @SuppressLint("StaticFieldLeak")
    private class DownloadTask extends AsyncTask<Void, Void, Void> {

        @Override
        protected Void doInBackground(Void... params) {
            SystemClock.sleep(1000 * 10);
            return null;
        }

        @Override
        protected void onPostExecute(Void aVoid) {
            super.onPostExecute(aVoid);

            try {
                updateText();
            } catch (Exception e) {
                //doNothing
            }
        }
    }
}
```

Để khắc phục rò rỉ này chúng ta cần phải biết nguyên nhân dẫn đến rò rỉ, nguyên nhân ở đây là class DowloadTask là lớp inner của class AsyncTaskActivity rò rỉ xảy ra khi class DowloadTask cố gắng để vòng đời dài hơn vòng đời của AsyncTaskActivity chứa nó, khi AsyncTaskActivity bị phá hủy nhưng inner class DowloadTask vẫn còn giữ tham chiếu tới đối tượng ở Activity cũ dẫn tới GC không thể dọn bộ nhớ gây ra rò rỉ bộ nhớ.

Để khắc phục thay vì sử dụng inner class thì ta sẽ tại ra 1 static class.

Bên dưới là code đã khắc phục lỗi

```
public class StaticAsyncTaskActivity extends Activity implements
DownloadListener {

    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world);
        textView = findViewById(R.id.text_view);

        new DownloadTask(this).execute();
    }

    public static void start(Context context) {
        Intent starter = new Intent(context,
StaticAsyncTaskActivity.class);
        context.startActivity(starter);
    }

    @Override
    public void onDownloadTaskDone() {
        updateText();
    }

    public void updateText() {
        textView.setText(R.string.hello);
    }

    private static class DownloadTask extends AsyncTask<Void, Void, Void> {
        private WeakReference<DownloadListener> listener;

        private DownloadTask(DownloadListener activity) {
            this.listener = new WeakReference<>(activity);
        }

        @Override
        protected Void doInBackground(Void... params) {
            SystemClock.sleep(1000 * 10);
            return null;
        }
```

```
        @Override
        protected void onPostExecute(Void aVoid) {
            super.onPostExecute(aVoid);
            if (listener.get() != null) {
                listener.get().onDownloadTaskDone();
            }
        }
    }
}
```

4.2. StaticAsyncTaskActivity

Là trường hợp lưu một strong reference của listener, điều này làm cho listener không đủ điều kiện để thu gom rác.

Bên dưới là code MemoryLeak khi sử dụng Static Class DownloadTask

```
public void onDownloadTaskDone() {
    updateText();
}

private static class DownloadTask extends AsyncTask<Void, Void, Void> {

    @SuppressLint("StaticFieldLeak")
    private DownloadListener listener;

    public DownloadTask(DownloadListener listener) {
        this.listener = listener;
    }

    @Override
    protected Void doInBackground(Void... params) {
        SystemClock.sleep(1000 * 10);
        return null;
    }

    @Override
    protected void onPostExecute(Void aVoid) {
        super.onPostExecute(aVoid);
        try {
            listener.onDownloadTaskDone();
        } catch (Exception e) {
            //doNothing
        }
    }
}
```

Để khắc phục sử dụng Weak Reference cho phép Activity được GC phá hủy. Rác được thu thập, Weak Reference bắt đầu được thu hồi.

Mô tả Weak Reference:

- Weak reference objects, không ngăn cản đối tượng tham chiếu của chúng được hoàn thiện, sau khi hoàn thiện sau đó thu hồi.
- Giả sử rằng GC xác định tại một thời điểm nhất định rằng một đối tượng có thể tiếp cận yếu. Tại thời điểm đó, nó sẽ tự động xóa tất

cả các weak references đến đối tượng đó và tất cả các Weak Reference đến bất kỳ đối tượng nào có khả năng tiếp cận yếu khác mà từ đó đối tượng đó có thể truy cập được thông qua một chuỗi các tham chiếu mạnh và mềm.

- Đồng thời, nó sẽ khai báo tất cả các đối tượng có thể truy cập yếu trước đây là có thể hoàn thiện. Đồng thời một thời gian sau, nó sẽ xếp hàng đợi những tham chiếu yếu mới được xóa đã được đăng ký với hàng đợi tham chiếu.

Bên dưới là code đã khắc phục lỗi

```
public void onDownloadTaskDone() {
    updateText();
}

public void updateText() {
    textView.setText(R.string.hello);
}

private static class DownloadTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<DownloadListener> listener;

    private DownloadTask(DownloadListener activity) {
        this.listener = new WeakReference<>(activity);
    }

    @Override
    protected Void doInBackground(Void... params) {
        SystemClock.sleep(1000 * 10);
        return null;
    }

    @Override
    protected void onPostExecute(Void aVoid) {
        super.onPostExecute(aVoid);
        if (listener.get() != null) {
            listener.get().onDownloadTaskDone();
        }
    }
}
```

4.3. ThreadActivity

Là việc Task được khai báo là non-static class nên nó sẽ giữ tham chiếu của activity khiến activity không đủ điều kiện để GC thu gom. Nếu Task được thực hiện trước khi xoay hoặc đóng Activity, mọi thứ sẽ ổn mà không bị leak.

Bên dưới là code Memory Leak khi sử dụng ThreadActivity

```
public class ThreadActivity extends Activity {

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

        new DownloadTask().start();
    }

    public static void start(Context context) {
        Intent starter = new Intent(context, ThreadActivity.class);
        context.startActivity(starter);
    }

    private class DownloadTask extends Thread {
        @Override
        public void run() {
                SystemClock.sleep(1000 * 10);
        }
    }
}
```

Để khắc phục trường hợp này ta cho task được thực hiện trước đó để đóng hoặc xoay Activity. Từ đó mọi thứ sẽ hoạt động tốt mà không bị leak.

Bên dưới là code sau khi đã khắc phục lỗi:

```
public class ThreadActivity extends Activity {

    private DownloadTask thread = new DownloadTask();

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

        thread.start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        thread.interrupt();
    }

    public static void start(Context context) {
        Intent starter = new Intent(context, ThreadActivity.class);
        context.startActivity(starter);
    }

    private static class DownloadTask extends Thread {
        @Override
        public void run() {
            while (!isInterrupted()) {
                SystemClock.sleep(2000 * 10);
            }
        }
```

```
        }
    }
```

4.4. HandlerActivity

Đây là trường hợp handler được khai báo là một lớp bên trong, nó có thể ngăn lớp bên ngoài được GC thu gom.

Bên dưới là code là code Memory Leak khi sử dụng Handler

```
public class HandlerActivity extends Activity {

    @SuppressLint("HandlerLeak")
    private final Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            Log.e("HandlerActivity", "new message");
        }
    };

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

        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                Log.e("HandlerActivity", "task start");
            }
        }, 1000 * 60 * 10);
    }

    public static void start(Context context) {
        Intent starter = new Intent(context, HandlerActivity.class);
        context.startActivity(starter);
    }

}
```

Để khác phục vấn để này ta sẽ thêm handler vào main. Đồng thời xóa mọi post đang chờ xử lý của Runnable @downloadTask nằm trong message queue, tránh bị rò rỉ.

Bên dưới là code sau khi đã khắc phục lỗi

```
public class HandlerActivity extends Activity {

    private final DownloadTask downloadTask = new DownloadTask();
    private final Handler handler = new TaskHandler();

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

        handler.postDelayed(downloadTask, 1000 * 60 * 10);
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();

        handler.removeCallbacks(downloadTask);
    }

    public static void start(Context context) {
        Intent starter = new Intent(context, HandlerActivity.class);
        context.startActivity(starter);
    }

    private static class DownloadTask implements Runnable {
        @Override
        public void run() {
            Log.e("HandlerActivity", "in run()");
        }
    }

    private static class TaskHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            Log.e("HandlerActivity", "handle message");
        }
    }

}
```

4.5. SingletonActivity

Đây là trường hợp singleton được khởi tạo bằng context khiến cho Activity sẽ xếp chồng trong bộ nhớ cho đến khi kết thúc ứng dụng.

Bên dưới là code Memory Leak khi sử dụng Singleton

```
public class SingletonManager {

    private static SingletonManager singleton;
    private Context context;

    private SingletonManager(Context context) {
        this.context = context;
    }

    public synchronized static SingletonManager getInstance(Context
context) {
        if (singleton == null) {

            singleton = new
SingletonManager(context.getApplicationContext());
        }
        return singleton;
    }

}
```

Để khắc phục vấn đề này ta sẽ khởi tạo singleton bằng context.getApplicationContext() để ngăn rò rỉ ứng dụng.

Bên dưới là code sau khi đã khắc phục lỗi

```
public class SingletonManager {

    private static SingletonManager singleton;
    private Context context;

    private SingletonManager(Context context) {
        this.context = context;
    }

    public synchronized static SingletonManager getInstance(Context
context) {
        if (singleton == null) {

            singleton = new
SingletonManager(context.getApplicationContext());
        }
        return singleton;
    }

}
```

4.6. Kết luận

Việc tối ưu hóa bộ nhớ trong lập trình android là vô cùng quan trọng. Nó khiến trải nghiệm người dùng được tốt hơn dẫn đến có nhiều người dùng sử dụng hơn và tạo nên một ứng dụng được người dùng tin tưởng và cài đặt. Là một lập trình viên song song với việc học các kỹ năng chuyên môn thì kỹ năng tối ưu hóa bộ nhớ là một kỹ năng không thể thiếu…

Tài liệu tham khảo

1.https://developer.android.com/

2.Android memory optimization book

3.Medium.com

4.https://square.github.io/leakcanary/fundamentals-how-leakcanary-works/

5. https://stackoverflow.com/questions/12084382/what-is-handlerleak

6.https://proandroiddev.com/everything-you-need-to-know-about-memory-leaks-in-android-d7a59faaf46a

7.https://stackoverflow.com/questions/26882594/static-singleton-class-memory-leak-in-android