

HỌC VIỆN KỸ THUẬT MẬT MÃ
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO MÔN HỌC
TỐI ƯU PHẦN MỀM DI ĐỘNG

Đề tài:

**NGHIÊN CỨU BÀI BÁO ANDROID MEMORY
OPTIMIZATION**

Giảng viên hướng dẫn: ThS. Lê Bá Cường

Sinh viên thực hiện: Trương Quang Nghĩa - CT040335

Nguyễn Phúc Sơn - CT040343

Hồ Minh Thông - CT040346

Nhóm 9

Hà Nội, 2023

MỤC LỤC

LỜI MỞ ĐẦU	4
CHƯƠNG I: GIỚI THIỆU VỀ TỐI ƯU HÓA BỘ NHỚ	5
CHƯƠNG II: CÁC VẤN ĐỀ VỀ TỐI ƯU HÓA BỘ NHỚ	7
2.1 Memory Leaks (Rò rỉ bộ nhớ)	7
2.2 Quản lý rò rỉ bộ nhớ	8
2.2.1 Ưu tiên Trường Hợp Thử Nghiệm	8
2.2.2 Tạo Thử Nghiệm để Phát Hiện Rò Rỉ	8
2.3 Quản lý bộ đệm GPU	8
2.4 Tránh trùng lặp bộ nhớ	9
2.5 Quản lý ứng dụng nền linh hoạt	10
2.6 Áp dụng những tối ưu hóa nhỏ	11
2.7 Phát hiện việc lập trình kém	11
2.8 Phân vùng bộ nhớ	12
2.9 Xử lý bộ nhớ bằng cách sử dụng Logs	12
2.10 Phát hiện Anti-Patterns	13
2.11 Giảm thiểu quá trình lão hóa phần mềm	13
2.12 Non-Blocking Garbage Collector	14
2.13 Sử dụng bộ nhớ Non-Volatile	14
CHƯƠNG III: GIẢI PHÁP TỐI ƯU HÓA BỘ NHỚ	16
Đề xuất giải pháp	16
3.1 Sử dụng các services một cách tiết kiệm	20
3.2 Sử dụng các vùng chứa dữ liệu được tối ưu hóa	20
3.3 Cẩn thận với các code abstractions	21
3.4 Sử dụng protobufs lite cho dữ liệu được tuần tự hóa	21
3.5 Tránh xáo trộn bộ nhớ	21
3.6 Xóa các tài nguyên và thư viện sử dụng nhiều bộ nhớ	22
3.7 Giảm kích thước APK tổng thể	23
3.8 Dependency injection frameworks	23
3.9 Hãy cẩn thận khi sử dụng các thư viện bên ngoài	23
CHƯƠNG IV: TIẾN HÀNH THỰC NGHIỆM	25
4.1. Sử dụng tham chiếu yếu	26
4.2. Sử dụng ProGuard	30
4.3 Sử dụng cache	30
4.4 Sử dụng ARC	31
4.5 Tối ưu hóa ảnh	37
KẾT LUẬN	39

DANH MỤC ẢNH

Hình 1: Cách hoạt động của ARC	16
Hình 2: Android ImageView loading	17
Hình 3: iOS ImageView loading	17
Hình 4: Tạo động một Đối tượng tham chiếu trong iOS	18
Hình 5: Android Button Registering and Unregistering event	18
Hình 6: iOS Button Registering event	19
Hình 7: Memory Profiler	25
Hình 8: LeakCanary	25
Hình 9: Kích thước của ứng dụng trước và sau khi dùng Proguard	30
Hình 10: Code tích hợp cache vào ứng dụng phát video	30
Hình 11: Kết quả khi sử dụng cache	31
Hình 12: Glide	38

LỜI MỞ ĐẦU

Về đề tài

Trong bối cảnh hiện nay, khi công nghệ di động phát triển với tốc độ nhanh chóng, hệ điều hành Android đóng vai trò quan trọng trong việc đáp ứng nhu cầu ngày càng cao của người sử dụng. Với sự phổ biến của các ứng dụng di động và đa dạng của các thiết bị Android, việc tối ưu hóa bộ nhớ trở thành một thách thức đối với những nhà phát triển và người quản lý hệ thống.

Phần mềm di động 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 cho smartphone.

Với tiềm năng và sự phát triển mạnh thì việc các sản phẩm phần mềm cũng ngày càng cạnh tranh hơn, từ đó chất lượng các phần mềm dần được nâng cao và cải thiện hơn về hiệu năng, độ tin cậy, tính an toàn,... Bản báo cáo với nhiệm vụ nghiên cứu về tối ưu hóa bộ nhớ trên nền tảng Android, nhằm cải thiện hiệu suất và trải nghiệm người dùng. Mục tiêu của đề tài báo cáo 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 tốt về bộ nhớ.

Cấu trúc báo cáo

Ngoài phần mở đầu và kết luận, bản báo cáo gồm 4 phần:

- Chương 1: Giới thiệu về tối ưu hóa bộ nhớ
- Chương 2: Các vấn đề về tối ưu hóa bộ nhớ
- Chương 3: Giải pháp tối ưu hóa bộ nhớ
- Chương 4: Thực nghiệm

Cuối cùng, nhóm chúng em xin gửi lời cảm ơn sâu sắc tới thầy. Người đã dành nhiều thời gian của mình hướng dẫn tận tình, chỉnh sửa và tạo điều kiện để nhóm có thể hoàn thiện đề tài một cách tốt nhất. Chúng em xin chân thành cảm ơn!

CHƯƠNG I: GIỚI THIỆU 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. Ở một số khía cạnh, chúng đã thay thế được máy tính của chúng ta. 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 vì nó là mã nguồn mở và các nhà sản xuất có thể dễ dàng tích hợp nó vào phần cứng của họ, giúp cho các thiết bị Android có giá thành rẻ hơn đối thủ cạnh tranh là iOS. Ngoài những mặt tích cực, nó cũng có những mặt tiêu cực. Một trong những vấn đề lớn mà người dùng Android phải đối mặt là gặp các sự cố không mong muốn 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 chạy trên cấu trúc bộ nhớ được biết đến là Bộ thu gom rác (Garbage Collector). Bộ thu gom rác là một công cụ Quản lý Bộ nhớ của Java do Android được xây dựng trên Máy ảo Java (JVM). Bộ thu gom rác theo dõi và xác định các đối tượng đã chết và giải phóng không gian khi chúng không còn cần thiết nữa. Nó có thể giải phóng bộ nhớ theo hai cách cơ bản. Đầu tiên là bằng cách kiểm tra định kỳ các đối tượng đã chết và giải phóng bộ nhớ của chúng, thứ hai, ngay lập tức giải phóng bộ nhớ khi có một phân bổ nào đó lớn hơn bộ nhớ trống có sẵn. Hãy giả sử chúng ta có một điện thoại thông minh Android với bộ nhớ chính là 1 GB. Bây giờ, có ba đối tượng là a, b và c, và khi chúng được tạo ra, bộ nhớ được cấp phát cho mỗi đối tượng theo nhu cầu của chúng. Khi những đối tượng này bị hủy sau khi phục vụ mục đích của chúng, chúng được đánh dấu là đã chết nhưng bộ nhớ vẫn chưa được giải phóng hoàn toàn. Trong tình huống này, nếu một đối tượng mới cần được cấp phát bộ nhớ và kích thước phân bổ của nó lớn hơn bộ nhớ trống có sẵn, thì sẽ xảy ra sự cố. Điều này xảy ra do Bộ thu gom rác chưa giải phóng không gian đã chết qua chu kỳ định kỳ của nó cho đến bây giờ. Khi nó giải phóng

không gian đã chết ngay sau khi nhận được một đối tượng mới có yêu cầu cao hơn, điều này sẽ gây ra một trục trặc và trải nghiệm người dùng kém chất lượng. Hơn nữa, JVM sao chép đối tượng từ một nơi sang nơi khác trong quá trình Thu gom rác và không ghi đè lên dữ liệu đã được thu gom cũ. Điều này có thể dẫn đến rò rỉ thông tin và thông tin cá nhân có thể bị đánh cắp.

Để giải quyết những vấn đề này và cải thiện quản lý bộ nhớ, đã có nhiều giải pháp được đề xuất bằng cách giữ bộ nhớ phụ trông bằng cách sử dụng GPU Buffers, Quản lý Ứng dụng Nền Linh hoạt, áp dụng các tối ưu hóa nhỏ, tránh Sự trùng lặp Bộ nhớ, phát hiện các thói quen lập trình xấu và khắc phục chúng v.v. Tất cả những kỹ thuật này đều cung cấp các tối ưu hóa trong hệ thống quản lý bộ nhớ hiện tại, đó là Bộ thu gom rác. Tuy nhiên, vấn đề chính vẫn tồn tại, đó là làm thế nào Bộ thu gom rác giải phóng bộ nhớ của các đối tượng đã chết.

Mục đích của nghiên cứu là đề xuất một phương pháp hoàn toàn khác để giải quyết vấn đề này. Việc sử dụng Đếm Tham Chiếu Tự động (Automatic Reference Counting - ARC) thay vì Bộ Thu Gom Rác trong thiết bị Android có thể là một hướng tiếp cận hướng tới việc cải thiện các vấn đề về bộ nhớ.

CHƯƠNG II: CÁC VẤN ĐỀ VỀ TỐI ƯU HÓA BỘ NHỚ

Tất cả các kỹ thuật được đề xuất cho đến nay đều xoay quanh Bộ Thu Gom Rác. Do đó, trước hết, các vấn đề liên quan đến bộ nhớ Android được thảo luận do đó Android thể hiện hiệu suất kém. Sau đó, các kỹ thuật đề xuất để cải thiện những vấn đề đó sẽ được thảo luận chi tiết.

2.1 Memory Leaks (Rò rỉ bộ nhớ)

Android chạy trên một nhân Linux được tùy chỉnh, điều này điều khiển phần cứng như Bộ nhớ và CPU và cho phép giao tiếp giữa phần cứng và người dùng. Hai thành phần chính của ứng dụng Android là Activities và Fragments. Mã nguồn không tối ưu cho những thành phần này, thường là quản lý không hiệu quả tài nguyên, có thể dẫn đến nhiều rò rỉ bộ nhớ và do đó gây ra vấn đề về bộ nhớ. Chi tiết về những thành phần này được thảo luận dưới đây:

- Activities (Hoạt động) là các thành phần cơ bản của bất kỳ Hệ điều hành Android nào. Chúng thực sự đại diện cho một Bộ điều khiển Xem. Mỗi lần một chế độ xem mới được hiển thị, đó là một hoạt động mới, và khi chế độ xem đó bị đóng, hoạt động cũng bị đóng.
- Fragments (Đoạn mã) là các thành phần của Giao diện người dùng. Chúng là các thành phần có thể tái sử dụng có thể được sử dụng trong nhiều hoạt động khác nhau. Tuổi thọ của chúng phụ thuộc vào tuổi thọ của một hoạt động. Khi một hoạt động được tạo, các đoạn mã cũng được tạo. Khi một hoạt động bị hủy, các đoạn mã cũng bị hủy.

Hầu hết các rò rỉ bộ nhớ xảy ra trong mã nguồn không tối ưu của Activities và Fragments, chẳng hạn như quên tái chế các thẻ hiển thị bitmap, hủy đăng ký sự kiện xử lý click, đóng các thẻ hiển thị con trỏ sau khi truy cập cơ sở dữ liệu và tham chiếu đối tượng từ các lớp tĩnh hoặc đánh dấu đối tượng làm tĩnh.

2.2 Quản lý rò rỉ bộ nhớ

Đã có nhiều phương pháp được đề xuất để giảm và loại bỏ rò rỉ bộ nhớ như sử dụng LeakDAF, tạo các trường hợp thử nghiệm để xác định và sửa chữa rò rỉ bộ nhớ bằng cách tuân theo các hướng dẫn lập trình cụ thể.

LeakDAF sử dụng kỹ thuật thử nghiệm UI để chạy ứng dụng và phân tích các tệp bảng xếp hạng bộ nhớ để xác định các hoạt động và đoạn mã đã rò rỉ.

Phương pháp tạo trường hợp thử nghiệm để xác định và sửa chữa rò rỉ bộ nhớ đã được các nhà nghiên cứu đề xuất theo hai cách riêng biệt như sau:

2.2.1 Ưu tiên Trường Hợp Thử Nghiệm

Sử dụng một phương pháp để ưu tiên các trường hợp thử nghiệm và chạy những trường hợp đó theo một thứ tự cụ thể thay vì chạy tất cả các trường hợp thử nghiệm vì chúng có thể tốn kém và dẫn đến tăng cường tải trên CPU. Ưu tiên của các trường hợp thử nghiệm được xác định bằng cách triển khai các thuật toán học máy dự đoán độ chính xác của một trường hợp thử nghiệm để xác định một rò rỉ bộ nhớ.

2.2.2 Tạo Thử Nghiệm để Phát Hiện Rò Rỉ

Đề xuất một kỹ thuật xác định một chuỗi sự kiện GUI tự nhiên như khởi chạy và đóng ứng dụng. Lặp lại các sự kiện như vậy không nên tăng sử dụng bộ nhớ nếu không có rò rỉ bộ nhớ. Nếu có rò rỉ, bộ nhớ sẽ tiếp tục tăng lên.

2.3 Quản lý bộ đệm GPU

GPU (Đơn vị Xử lý Đồ họa) tương tự như CPU nhưng được sử dụng để xử lý đồ họa. Theo thời gian, ứng dụng và trò chơi ngày càng phức tạp và yêu cầu xử lý đồ họa mạnh mẽ, vì vậy cần có một GPU riêng biệt. Thiết bị di động được trang bị GPU giống như máy tính để bàn, nhưng có sự khác biệt lớn giữa các GPU của chúng. GPU dành cho máy tính để bàn có bộ nhớ riêng biệt, nhưng do kích thước nhỏ và tính di động của thiết bị di động, GPU của điện thoại thông minh không có bộ nhớ riêng mà chia sẻ bộ nhớ chính. Việc chia sẻ này giảm bớt bộ nhớ chính có sẵn.

Android lưu trữ ứng dụng để chúng có thể được khởi chạy nhanh chóng lần sau khi chúng được gọi. Khi được lưu trữ, bộ đệm GPU cũng được lưu trữ vì ứng dụng chứa đồ họa. Khi ứng dụng bị kết thúc, dữ liệu đã được lưu trữ của nó vẫn còn trong bộ nhớ chính, do đó làm tăng sử dụng bộ nhớ và giảm bớt bộ nhớ có sẵn.

Một trong những giải pháp hiện tại là nén bộ nhớ của GPU khi ứng dụng được đưa vào nền và giải nén khi quay lại mặt trước. Bộ nhớ nén sẽ chiếm rất ít không gian và sẽ cải thiện hiệu suất tổng của thiết bị. Tuy nhiên, quá trình nén và giải nén này sẽ tăng thời gian truy cập, làm cho giải pháp này ít tối ưu hóa hơn.

2.4 Tránh trùng lặp bộ nhớ

Trùng lặp bộ nhớ là quá trình trong đó các trang bộ nhớ giống nhau được đặt trong bộ nhớ chính nhiều lần hơn một lần. Bằng cách sử dụng kỹ thuật trùng lặp này, khi Bộ Thu Gom Rác giải phóng không gian, nó sử dụng rất nhiều chu kỳ CPU, gây giật và đôi khi thậm chí là crash ứng dụng. Do đó, tránh trùng lặp trang là cần thiết để Bộ Thu Gom Rác không phá hủy nội dung giống nhau nhiều lần. Để khắc phục những vấn đề như vậy, Android đã giới thiệu một số cơ chế như zRAM và Kernel Same-Page Merging (KSM). KSM hợp nhất các trang bộ nhớ giống nhau thành một. zRAM sử dụng một khu vực được cấp phát đặc biệt trong bộ nhớ chính được biết đến là khu vực đổi, nơi nó nén các trang đã lưu trữ cần được đổi chỗ. Cả hai cơ chế này giảm sử dụng bộ nhớ nhưng tiêu tốn số lượng lớn chu kỳ CPU và năng lượng.

Memscope là một công cụ có thể được sử dụng để tránh những vấn đề như vậy. Nó lấy bản chụp bộ nhớ ở các khoảng thời gian cố định. Nó xác định các khung trùng lặp có thể tồn tại trong một khoảng thời gian cụ thể và làm thế nào chúng sẽ thay đổi qua vòng đời của ứng dụng. Nó phân tích các bản chụp và xác định các khung có khả năng trùng lặp và tập trung vào những khung đó để tránh trùng lặp.

Một Phương pháp khác hiện đang được sử dụng để giảm và loại bỏ Trùng Lặp Bộ Nhớ là Trùng Lặp Bộ Nhớ Chọn Lọc. Thay vì quét toàn bộ bộ nhớ, nó quét một số trang bộ nhớ cụ thể để giảm sử dụng CPU và năng lượng. Cơ chế này quét các trang bộ nhớ của các ứng dụng chỉ một lần trong nền cho đến khi chúng được đưa lên phần trước vì khả năng rất ít khi kích thước bộ nhớ sẽ thay đổi cho các ứng dụng đang chạy ở nền.

2.5 Quản lý ứng dụng nền linh hoạt

Hệ điều hành Android duy trì các ứng dụng được sử dụng gần đây trong bộ nhớ cache thông qua hệ thống Tang để tăng tốc độ nạp lại, giảm thời gian khởi chạy và tiêu thụ năng lượng. Trạng thái của một ứng dụng khác nhau khi nó được khởi chạy và khi nó bị dừng. Trong Hệ điều hành Android, quá trình Zygote tích hợp sẵn đóng vai trò khởi chạy ứng dụng, cũng như nhận và phản hồi sự kiện của người dùng. Quá trình này cũng được sử dụng để quản lý bộ nhớ trong Android. Khi thiết bị trải qua giai đoạn bộ nhớ thấp, Low Memory Killer (LMK) bắt đầu hoạt động và kết thúc ứng dụng ít được sử dụng gần đây nhất (LRU). Nếu ứng dụng cần bị kết thúc có kích thước lớn, việc giải phóng bộ nhớ sẽ mất thời gian khi nó được khởi chạy lần tiếp theo. Một kỹ thuật khác là Out of Memory Killer (OOMK), giết chết các ứng dụng có ưu tiên thấp, giết chết nhiều ứng dụng cùng một lúc để giải phóng bộ nhớ, điều này có thể ảnh hưởng xấu đối với các ứng dụng này. Tóm lại, những cơ chế này gặp phải những vấn đề cơ bản sau:

1. Ứng dụng sử dụng bộ nhớ cao được ưu tiên để kết thúc.
2. Thời gian khởi chạy ứng dụng không được xem xét khi chọn người chết.
3. Việc thu hồi bộ nhớ là một quy trình theo yêu cầu và phải đưa ra quyết định nhanh chóng, vì vậy một thuật toán tối ưu hóa có thể dẫn đến việc kết thúc các ứng dụng không đúng.

Dịch vụ Quản lý Hoạt động (AMS) có trách nhiệm xử lý yêu cầu của người dùng. Nó duy trì các ứng dụng trong bộ nhớ cache dựa trên ưu tiên, được xác định theo thứ tự sau:

1. Ứng dụng đang chạy ở phía trước.
2. Một quy trình được ràng buộc với ứng dụng đang chạy ở phía trước.
3. Một quy trình được ràng buộc với ứng dụng đang chạy ở phía sau.
4. Các quy trình ẩn có trên thiết bị nhưng không hiển thị.
5. Các ứng dụng cung cấp nội dung như lịch và email.
6. Các quy trình trống rỗng được lưu trữ để tăng tốc độ nạp lại.

Các chính sách lưu trữ hiện tại có phần tĩnh lập và không thay đổi tùy thuộc vào sở thích sử dụng ứng dụng của người dùng. Các nhà nghiên cứu đã đề xuất một giải pháp tốt

hơn, lấy into account tính tái sử dụng của ứng dụng để xác định số lượng ứng dụng động cần được lưu trữ. Kỹ thuật này cho phép quản lý bộ nhớ chính một cách hiệu quả để có cách lưu trữ tốt hơn và do đó cải thiện hiệu suất tổng thể của thiết bị.

Một đề xuất là sử dụng một công cụ được gọi là SmartLMK. Đây là một quy trình động, về cơ bản là một ứng dụng bộ nhớ thấp đang chạy ở nền. Nó theo dõi thời gian khởi chạy của ứng dụng, dữ liệu sử dụng và các đặc điểm khác. Sử dụng các thống kê này, nó tính toán một phạt thời gian và sử dụng nó để kết thúc một ứng dụng.

2.6 Áp dụng những tối ưu hóa nhỏ

Tối ưu hóa cần được áp dụng cho ứng dụng di động vì chúng thường xuyên gặp vấn đề về hiệu suất. Micro-tối ưu hóa nên được áp dụng ở đầu vòng đời của ứng dụng vì dễ tối ưu hóa khi cấu trúc đang được viết so với khi nó đã được viết hoàn toàn. Theo các kết quả nghiên cứu, việc loại bỏ biến và phương thức riêng tư không sử dụng cải thiện hiệu suất vì nó giảm bớt kích thước bộ nhớ.

Thường thì các nhà phát triển không áp dụng micro-tối ưu hóa vì:

1. Hầu hết họ không biết về điều này.
2. Họ nghĩ rằng ứng dụng của họ quá nhỏ để áp dụng bất kỳ tối ưu hóa nào.
3. Họ không nghĩ rằng việc dành thời gian cho micro-tối ưu hóa là đáng giá.
4. Họ không tin rằng micro-tối ưu hóa sẽ giúp ích cho ứng dụng của họ.

Các kết quả được thu được thông qua phân tích tĩnh được thực hiện bằng cách sử dụng các công cụ như FindBugs, PMD và LINT cung cấp cảnh báo để cải thiện mã nguồn. Một trong những tối ưu hóa hữu ích nhất được tìm thấy là loại bỏ mã không sử dụng. Nhưng các công cụ và kỹ thuật như vậy có thể dễ mắc lỗi và không thể tin tưởng hoàn toàn.

2.7 Phát hiện việc lập trình kém

Một giải pháp khác được các nhà nghiên cứu đề xuất là công cụ CheckDroid được sử dụng để xác định thực hành lập trình kém. Việc 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 vì thực hành kém thường không được xác định và báo cáo bởi môi trường

phát triển tích hợp (IDE). Hầu hết các công cụ và kỹ thuật được áp dụng để phát hiện rò rỉ bộ nhớ và hiệu suất, nhưng ít chú ý đến thực hành lập trình kém.

Một số đề xuất về hiệu suất sau khi phân tích là:

1. Logging chi tiết không nên được giữ lại trong ứng dụng hoạt động.
2. Các công việc chạy lâu dài nên được chia thành các luồng con.
3. Những luồng con này nên có ưu tiên thấp hơn so với luồng chính.

Một số tối ưu hóa chính được đề xuất là:

1. Tham chiếu đến ngữ cảnh không nên được lưu trữ trong biến tĩnh.
2. Các luồng được tạo ra nên được hủy bỏ khi không cần thiết.

2.8 Phân vùng bộ nhớ

Một đề xuất khác là phân vùng bộ nhớ chính vì Android sử dụng LMK và OOMK để giải phóng bộ nhớ bằng cách kết thúc các quy trình nạn nhân, điều này không hiệu quả lắm. Vì vậy, đề xuất một kỹ thuật phân vùng bộ nhớ chia bộ nhớ thành hai nút ảo.

1. Nút ảo 0 được sử dụng cho các ứng dụng đáng tin cậy.
2. Nút ảo 1 được sử dụng cho các ứng dụng không đáng tin cậy.

Nếu bộ nhớ cạn kiệt từ một nút, chỉ bộ nhớ của nút đó sẽ được giải phóng. Thông thường, các ứng dụng không đáng tin cậy chiếm nhiều bộ nhớ hơn so với các ứng dụng đáng tin cậy. Bằng cách tuân thủ phương pháp này, bộ nhớ có thể được tiết kiệm đến một mức độ từ việc cạn kiệt.

2.9 Xử lý bộ nhớ bằng cách sử dụng Logs

Kỹ thuật này đề xuất sử dụng các logs được tạo ra để xác định tương tác của người dùng và thời gian mà họ sử dụng các ứng dụng để động động thay đổi ưu tiên của các ứng dụng. Điều này sẽ cho phép giữ các ứng dụng ưu tiên cao trong bộ nhớ cache và chỉ kết thúc các ứng dụng ưu tiên thấp khi có vấn đề về bộ nhớ thấp.

2.10 Phát hiện Anti-Patterns

Trong việc phát triển ứng dụng một cách nhanh chóng, các nhà phát triển hiện nay thường lạc quan khỏi các mô hình lập trình được biết đến là các mẫu chống, dẫn đến thiết kế kém và do đó hiệu suất kém của ứng dụng. Paprika là một công cụ được đề xuất để phân tích mã nguồn và xác định các mẫu chống, đồng thời đề xuất giải pháp để sửa chúng. OOP (Lập trình hướng đối tượng) là khối xây dựng cơ bản của việc phát triển bất kỳ loại ứng dụng nào, nó cung cấp khả năng tái sử dụng và các chức năng khác mà trước đây là không thể.

Các file apk của Android chứa file .dex chứa các lớp Java đã được biên dịch. Android chạy trên Máy ảo Dalvik và bytecode của nó khác biệt so với Java. Có nhiều công cụ có sẵn để đảo ngược các file .dex.

Paprika trước tiên trích xuất siêu dữ liệu từ apk như tên ứng dụng, định danh gói và đánh giá của người dùng, sau đó các đối tượng mã nguồn như lớp, phương thức và tên biến cũng được trích xuất. Bằng cách sử dụng các đối tượng đã được trích xuất, một mô hình mã nguồn được tính toán dưới dạng đồ thị với các giá trị raw. Mô hình này được lưu trữ trong một cơ sở dữ liệu đồ thị và sau đó cơ sở dữ liệu này được truy vấn để phát hiện các mẫu chống. Những mẫu chống này khi được sửa chữa có thể giải phóng một lượng lớn bộ nhớ.

2.11 Giảm thiểu quá trình lão hóa phần mềm

Lão hóa phần mềm là quá trình mà hiệu suất của hệ điều hành và ứng dụng giảm đi theo thời gian. Vấn đề chủ yếu là rò rỉ bộ nhớ. Khi lão hóa, bộ nhớ trống giảm đi nên ít ứng dụng được lưu trữ và khi ứng dụng mới được khởi chạy, bộ nhớ cần được giải phóng sẽ tốn CPU và pin. Để phát hiện và xác định lão hóa, cần giữ các thống kê sử dụng tài nguyên. Để điều tra rò rỉ bộ nhớ, thiết bị cần được kiểm tra dưới điều kiện nghiêm trọng nơi nó có nguy cơ gặp sự cố. Đối với điều này, Exerciser Monkey được sử dụng bởi các nhà nghiên cứu, mô phỏng sự kiện chạm, nhấp chuột và các sự kiện Android phổ biến khác. Các bài kiểm tra được chia thành các bài kiểm tra phụ, với mỗi bài kiểm tra phụ phụ thuộc vào kết quả của bài kiểm tra trước đó. Nếu phát hiện lão hóa, bài kiểm tra tiếp theo sẽ được thực hiện trong thời gian dài hơn.

Những bài kiểm tra này giúp xác định phạm vi của rò rỉ bộ nhớ trong các ứng dụng khác nhau.

2.12 Non-Blocking Garbage Collector

Bộ thu gom rác (Garbage Collector) hoạt động dựa trên nguyên tắc "ngừng thực thi hiện tại, đánh dấu các đối tượng để xóa và làm sạch chúng khỏi bộ nhớ," điều này có thể dẫn đến các hành vi không phản ứng và thậm chí là sự cố. Để khắc phục những vấn đề này, các nhà nghiên cứu đã đề xuất một Bộ thu gom rác thời gian thực có hai đặc điểm sau đây:

1. GC này sẽ hoạt động một cách tăng dần với giai đoạn chặn ngắn.
2. Tốc độ của GC phải phù hợp với rác được tạo ra bởi hệ điều hành để tránh tình trạng Hết Bộ Nhớ.

Điều này sẽ cho phép giải phóng bộ nhớ nhanh chóng so với Bộ thu gom rác không chặn và sẽ cải thiện hiệu suất tổng thể của hệ điều hành Android.

2.13 Sử dụng bộ nhớ Non-Volatile

Theo một nghiên cứu, hầu hết người dùng có các phiên ứng dụng ít hơn 10 giây và, trong trường hợp như vậy, việc giảm thời gian tải ứng dụng là cần thiết. Nhiều giải pháp đã được đề xuất để cải thiện thời gian khởi chạy ứng dụng nhưng hầu hết đều không phải là dựa trên phần cứng. Giải pháp được đề xuất bởi các nghiên cứu là sử dụng Bộ Nhớ Không Biến Động (NVM), cụ thể là Bộ Nhớ Thay Đổi Pha (PCM) như là bản sao lưu của bộ nhớ chính. Bộ Nhớ NVM phổ biến vì nó tiêu thụ ít năng lượng. PCM nhanh, rất hiệu quả năng lượng cho các hoạt động đọc nhưng tiêu thụ nhiều năng lượng trong các hoạt động ghi và rất chậm. Do đó, một giải pháp kết hợp DRAM-PCM được đề xuất để cải thiện thời gian khởi chạy ứng dụng.

Một nghiên cứu đã được tiến hành để phân tích ứng dụng dựa trên bộ nhớ và phát hiện rằng ứng dụng thuộc hai loại hạng mục:

Ứng dụng ổn định, nơi bộ nhớ tăng lên ban đầu trong 10 giây đầu tiên rồi ổn định.

Ứng dụng không ổn định, nơi bộ nhớ tiếp tục tăng lên theo thời gian và không bao giờ ổn định.

Giữ cả hai loại ứng dụng này trong tâm trí, giải pháp đề xuất là sử dụng NVM như bản sao lưu của bộ nhớ chính. Khi Bộ thu gom rác (GC) bắt đầu và loại bỏ dữ liệu khỏi bộ nhớ chính, dữ liệu được chuyển giao sang NVM. Các khu vực cụ thể được chỉ định trong NVM để lưu trữ dữ liệu từ các ứng dụng sử dụng thường xuyên và thư viện chia sẻ phổ biến. Phương pháp này nhằm mục đích cải thiện thời gian khởi chạy, ngay cả khi các ứng dụng không được lưu trữ trong bộ nhớ chính, nhưng chúng sẽ có mặt trong bộ nhớ dự phòng.

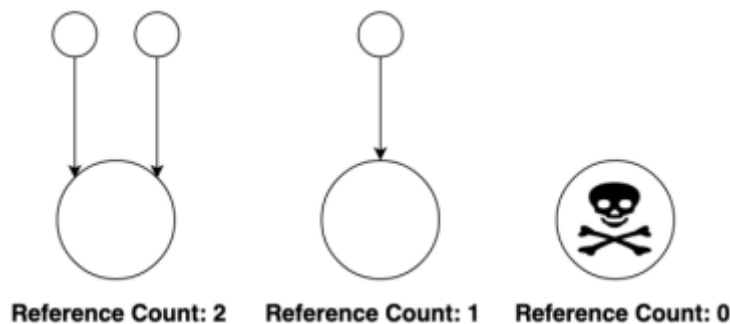
Trong khi những giải pháp này tập trung vào việc tối ưu hóa các hệ thống bộ nhớ và quản lý bộ nhớ hiện có trong Android, không có giải pháp nào cung cấp một giải pháp toàn diện và rõ ràng để giải quyết những vấn đề này.

CHƯƠNG III: GIẢI PHÁP TỐI ƯU HÓA BỘ NHỚ

Đề xuất giải pháp

Đề xuất một phương pháp hoàn toàn khác nhau để thay thế hệ thống Garbage Collector của Android bằng hệ thống Automatic Reference Counting (ARC) của iOS.

Đề xuất một phương pháp hoàn toàn khác để thay thế hệ thống Garbage Collector của Android bằng hệ thống Automatic Reference Counting (ARC) của iOS. Trong hệ thống ARC, đối tượng sẽ tự động giải phóng bộ nhớ khi không còn đối tượng nào giữ tham chiếu đến nó, và do đó, giải phóng bộ nhớ một cách tự động. Cơ chế này của ARC được minh họa trong hình 1.



Hình 1: Cách hoạt động của ARC

Sự khác biệt lớn nhất giữa ARC và Garbage Collector là ARC giải phóng bộ nhớ ngay lập tức khi đối tượng được giải phóng. Trong khi đó, Garbage Collector đánh dấu các đối tượng đã giải phóng và làm sạch bộ nhớ sau các khoảng thời gian đều đặn hoặc khi hệ điều hành cạn kiệt bộ nhớ.

```
public class TestActivity extends AppCompatActivity {  
    ImageView myImageView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_test);  
  
        myImageView = (ImageView) findViewById(R.id.image_view);  
        myImageView.setImageResource(R.drawable.test_image);  
    }  
  
    @Override  
    protected void onDestroy() {  
        if (myImageView != null) {  
            ((BitmapDrawable) myImageView.getDrawable()).getBitmap().recycle();  
            myImageView = null;  
        }  
        super.onDestroy();  
    }  
}
```


Hình 2: Android ImageView loading

Giờ hãy thảo luận về một số ví dụ mã nguồn về quản lý bộ nhớ trong Garbage Collector và ARC để làm rõ sự khác biệt giữa hai phương pháp này. Đối với các ví dụ mã nguồn, ARC được sử dụng trong ngôn ngữ Swift trên Xcode.

```
class TestViewController: UIViewController {  
    @IBOutlet weak var mainImageView_: UIImageView!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let image = UIImage(named: "eagle")  
        mainImageView_.image = image  
    }  
  
    deinit {  
    }  
}
```

Hình 3: iOS ImageView loading

IDE và Bộ Thu Gom Rác được sử dụng trong Ngôn Ngữ Java trên Android Studio IDE. Trước hết, cả trong iOS (ARC) và Android (Bộ Thu Gom Rác), hãy gán tham chiếu hình ảnh cho đối tượng ImageView được tạo trong các tệp tài nguyên giao diện người dùng tương ứng của họ trong các hình 2 và 3.

Trong hình 2 và 3, các đối tượng ImageView với một số hình ảnh thử nghiệm trong cả hai hệ thống bộ nhớ được khởi tạo. Android tái sử dụng tham chiếu imageView của imageResource khi lớp cha onDestroy được gọi và cũng đặt tham chiếu imageView về null để tránh gây ra rò rỉ bộ nhớ. Điều này có nghĩa là ngay cả khi bạn không tạo đối tượng tham chiếu một cách động, bạn vẫn cần gán giá trị null để đánh dấu nó có thể gỡ bỏ được cho Bộ Thu Gom Rác. Trong khi đó, iOS không cần thực hiện bất kỳ điều gì tương tự.

Khi lớp cha bị hủy, referenceCount của đối tượng myImageView giảm và đến khi nó đạt đến số không, nó sẽ bị hủy và bộ nhớ sẽ được giải phóng tự động. Ở đây, một ví dụ khác được hiển thị trong hình 4.

```

class Test2ViewController: UIViewController {

    var secondImageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let image = UIImage(named: "eagle")
        secondImageView = UIImageView(image: image)

        self.view.addSubview(secondImageView)
    }

    deinit {

    }
}

```

Hình 4: Tạo động một Đối tượng tham chiếu trong iOS

Trong Hình 4, một đối tượng tham chiếu (secondImageView kiểu UIImageView) được tạo động và thêm vào view cha. Khi view cha bị hủy, referenceCount của secondImageView trở thành số không và tự động bị hủy. Điều này cho thấy rằng ngay cả đối với các đối tượng được tạo động, không cần phải gán tham chiếu null khi view cha bị giải phóng. Điều này mang lại hiệu suất lớn so với Bộ Thu Gom Rác, đối với nó, bạn cần gán null cho tham chiếu để đánh dấu chúng để được thu gom rác.

```

public class Test2Activity extends AppCompatActivity {

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

        findViewById(R.id.button).setOnClickListener (
            new View.OnClickListener() {
                public void onClick(View v) { }
            });
    }

    @Override
    protected void onDestroy() {
        findViewById(R.id.button).setOnClickListener(null);
        super.onDestroy();
    }
}

```

Hình 5: Android Button Registering and Unregistering event

Bây giờ, hãy xem xét một ví dụ phổ biến khác về việc đăng ký sự kiện cho nút trong cả hai mô hình bộ nhớ, như được thể hiện trong các hình 5 và 6.

```
class Test3ViewController: UIViewController {  
    @IBOutlet weak var myButton_: UIButton!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        myButton_.addTarget(self, action:  
            #selector(onButtonClick(_:)), for: .touchUpInside)  
    }  
  
    @objc func onButtonClick (_ sender: AnyObject) {  
    }  
  
    deinit {  
    }  
}
```

Hình 6: iOS Button Registering event

Trong trường hợp của Android (hình 5), sau khi đăng ký sự kiện nhấn nút, việc hủy đăng ký là bắt buộc, nếu không sẽ gây rò rỉ bộ nhớ. Tuy nhiên, không cần điều này trong iOS. Trong iOS (hình 6), khi lớp cha bị hủy, referenceCount của đối tượng myButton_ giảm và nó đạt đến số không, do đó nó bị hủy và bộ nhớ được giải phóng tự động.

Sau khi đi qua các ví dụ, rõ ràng rằng Garbage Collector có khả năng gây rò rỉ bộ nhớ hơn trong khi ARC xử lý bộ nhớ một cách tự động và hiệu quả hơn. Tóm tắt về các ưu điểm chính của ARC so với Garbage Collection như sau:

1. ARC giải phóng bộ nhớ ngay lập tức khi đối tượng bị hủy, khác với phương pháp quét và đánh dấu của Garbage Collector.
2. Không cần phải đánh dấu biến là null hoặc tái sử dụng, và sự kiện nhấn không cần được loại bỏ để tránh rò rỉ bộ nhớ.
3. Đối tượng được tự động hủy khi ra khỏi phạm vi.

Chính vì những sự thật này mà các thiết bị iOS hoạt động hiệu quả hơn so với các đối tác Android của chúng và vì lý do này, các thiết bị iOS sở hữu khoảng 1/3 bộ nhớ ít hơn so với một thiết bị Android có hiệu suất tương tự.

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 Cần 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 thể 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 công kênh khởi 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.

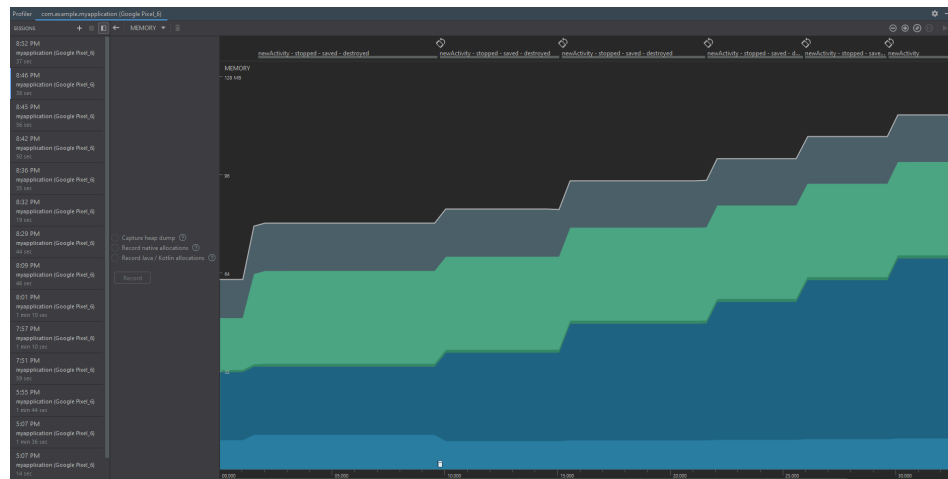
Mặc dù ProGuard có thể giúp loại bỏ các API và tài nguyên có cồng kềnh, nhưng nó không thể loại bỏ các phụ thuộc nội bộ lớn của thư viện. Các tính năng mà bạn muốn trong các thư viện này có thể yêu cầu các phụ thuộc cấp thấp hơn. Điều này trở nên đặc biệt có vấn đề khi bạn sử dụng lớp con Activity từ 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 IV: TIẾN HÀNH THỰC NGHIỆM

Memory Profiler

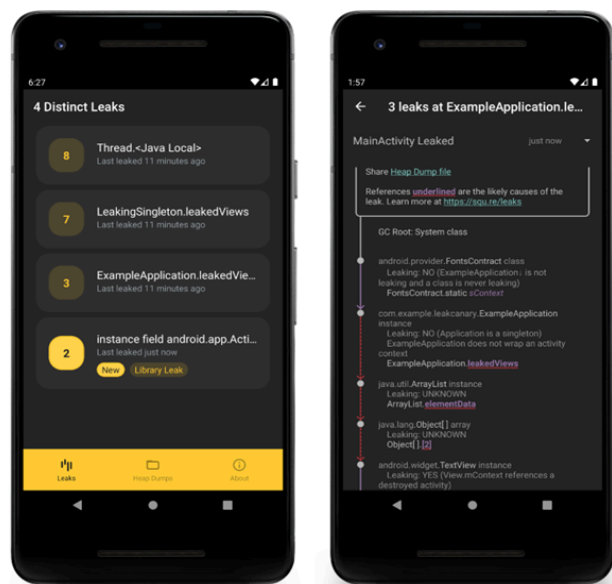
Android Studio cung cấp công cụ là Memory Profiler để phát hiện rò rỉ bộ nhớ.



Hình 7: Memory Profiler

LeakCanary

Một cách khác để tìm rò rỉ bộ nhớ là sử dụng LeakCanary, một thư viện do Square tạo ra để giúp phát hiện rò rỉ bộ nhớ.



Hình 8: LeakCanary

4.1. Sử dụng tham chiếu yếu

- Tham chiếu mạnh (Strong reference):

Mọi khi bạn tạo một biến để lưu trữ một đối tượng, bạn đang sử dụng một tham chiếu mạnh. Tham chiếu mạnh đảm bảo rằng đối tượng sẽ không bao giờ bị thu hồi bởi garbage collector cho đến khi không còn tham chiếu mạnh nào trỏ đến nó.

- Tham chiếu yếu (Weak reference):

Tham chiếu yếu cho phép một đối tượng được thu hồi bởi garbage collector ngay cả khi vẫn còn tham chiếu đến nó, nếu chỉ có tham chiếu yếu trỏ đến nó.

Khi sử dụng tham chiếu yếu, bạn có thể tránh được một số vấn đề liên quan đến rò rỉ bộ nhớ, đặc biệt là khi bạn làm việc với cấu trúc dữ liệu phức tạp hoặc các tương tác giữa các đối tượng lớn

Đoạn code phía dưới có thể gây ra rò rỉ bộ nhớ nếu đóng Activity trước khi hàm WaitTask chạy xong

```
public class AsyncTaskActivity extends Activity {  
  
    private TextView textView;  
  
    private WaitTask waitTask;  
  
    @Override  
  
    protected void onCreate(Bundle savedInstanceState) {  
  
        super.onCreate(savedInstanceState);  
  
        setContentView(R.layout.activity_async_task);  
  
        textView = findViewById(R.id.text_view);  
  
        waitTask = new WaitTask();  
  
        waitTask.execute();  
  
    }  
  
    public static void start(Context context) {  
  
        Intent starter = new Intent(context, AsyncTaskActivity.class);  
  
        context.startActivity(starter);  
  
    }  
}
```

```

    }

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

        if (waitTask != null) {
            waitTask.cancel(true);
        }
    }

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

    @SuppressWarnings("StaticFieldLeak")
    private class WaitTask 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 bộ nhớ ta sử dụng Weak Reference để cho phép Activity được GC phá hủy. Rác được thu thập, Weak Reference bắt đầu được thu hồi. Bên dưới là code khắc phục

```
public class AsyncTaskWeakRefActivity extends Activity {

    private TextView textView;

    private WaitTask waitTask;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_async_task_weak_ref);

        textView = findViewById(R.id.textView1);

        waitTask = new WaitTask(AsyncTaskWeakRefActivity.this);

        waitTask.execute();

    }

    public static void start(Context context) {

        Intent starter = new Intent(context, AsyncTaskWeakRefActivity.class);

        context.startActivity(starter);

    }

    public void updateText() {

        textView.setText(R.string.string);

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        if (waitTask != null) {

            waitTask.cancel(true);

        }

    }

}
```

```

    }

    @SuppressWarnings("StaticFieldLeak")
    private static class WaitTask extends AsyncTask<Void, Void, Void> {

        private WeakReference<AsyncTaskWeakRefActivity> listener;

        WaitTask(AsyncTaskWeakRefActivity 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);

            AsyncTaskWeakRefActivity activity = listener.get();

            if (activity != null) {

                try {

                    activity.updateText();

                } catch (Exception e) {

                    // Handle exceptions
                }

            }



        }

    }
}

```

4.2. Sử dụng ProGuard

ProGuard là một công cụ tối ưu mã nguồn Java, giảm kích thước ứng dụng bằng cách loại bỏ mã nguồn không sử dụng, thu nhỏ tên biến và phương thức, nén mã nguồn qua đó làm giảm kích thước của ứng dụng

 app-release-unsigned.apk	12/6/2023 12:16 AM	ldmnq.apk	4,498 KB
 app-release-unsigned.apk	12/6/2023 12:11 AM	ldmnq.apk	1,932 KB

Hình 9: Kích thước của ứng dụng trước và sau khi dùng Proguard

4.3 Sử dụng cache

Bộ nhớ cache trong Android là một phần của bộ nhớ của thiết bị để lưu trữ một số dữ liệu tạm thời từ các ứng dụng và trình duyệt. Sử dụng cache để tăng cường hiệu suất hệ thống bằng cách lưu trữ các dữ liệu ứng dụng có thể truy cập nhanh chóng mà không cần phải tải lại từ nguồn. Việc sử dụng bộ nhớ cache để tối ưu bộ nhớ đặc biệt cần thiết với các ứng dụng có load ảnh hay load video.

```
// Create a SimpleCache with an eviction policy
File cacheDir = new File(getCacheDir(), "exoplayer_cache");
Cache cache = new SimpleCache(cacheDir, new LeastRecentlyUsedCacheEvictor( (maxBytes: 1024 * 1024 * 100), new ExoDatabaseProvider( context: this)));

// Create a CacheDataSource.Factory with the cache
CacheDataSource.Factory dataSourceFactory = new CacheDataSource.Factory()
    .setCache(cache)
    .setUpstreamDataSourceFactory(new DefaultHttpDataSource.Factory());

// Create an ExoPlayer with the MediaSourceFactory using the cache
player = new ExoPlayer.Builder( context: this).setMediaSourceFactory(new DefaultMediaSourceFactory(dataSourceFactory)).build();
```

Hình 10: Code tích hợp cache vào ứng dụng phát video

```
67 Cache com.example.videostreaming D Cache size: 71932022
92 Cache com.example.videostreaming D Bytes read from cache: 15019512, Bytes cached: 71932022
92 Cache com.example.videostreaming D Bytes read from cache: 3539668, Bytes cached: 83195286
92 Cache com.example.videostreaming D Bytes read from cache: 9864223, Bytes cached: 84858727
92 Cache com.example.videostreaming D Bytes read from cache: 186590, Bytes cached: 84858727
92 Cache com.example.videostreaming D Bytes read from cache: 14839849, Bytes cached: 84858727
92 Cache com.example.videostreaming D Bytes read from cache: 7161715, Bytes cached: 84858727
92 Cache com.example.videostreaming D Bytes read from cache: 186590, Bytes cached: 85491152
```

Hình 11: Kết quả khi sử dụng cache

4.4 Sử dụng ARC

Toàn bộ những giải pháp thường sẽ phần nào liên quan đến việc tối ưu cache và hệ thống quản lý bộ nhớ hiện có trong Android nhưng không giải pháp nào thực sự xử lý được những vấn đề này một cách thích hợp và được xác định rõ ràng.

Một cách tiếp cận hoàn toàn khác đó chính là việc thay thế Android's Garbage Collector (GC) bằng một hệ thống khác là Automatic Reference Counting (ARC) của iOS's

ARC là gì:

Automatic Reference Counting (ARC, tạm dịch: Bộ tự động đếm tham chiếu) là một hệ thống mà mỗi đối tượng được tạo ra sẽ được gán thêm 1 biến dữ liệu dạng số nguyên có tên là "referenceCount" hay "số lượng tham chiếu".

Khi tham chiếu đến đối tượng được tạo "referenceCount" sẽ tăng lên 1 và ngược lại, khi tham chiếu đến đối tượng bị hủy bỏ "referenceCount" sẽ giảm 1.

Khi referenceCount về 0, đối tượng sẽ bị loại bỏ và bộ nhớ được giải phóng. Chính vì thế, việc tái sử dụng tham chiếu hay việc hủy đăng ký các sự kiện là không cần thiết như trong khi sử dụng GC.

So sánh ARC và GC:

Điểm khác biệt lớn nhất giữa GC và ARC là: ARC sẽ ngay lập tức giải phóng bộ nhớ khi đối tượng không còn được sử dụng. Trong khi đó GC sẽ đánh dấu các đối tượng không còn được sử dụng và xóa nó khi ART (Android Runtime) cảm nhận bộ nhớ đã bị lấp đầy và không thể cấp phát bộ nhớ.

Để làm rõ hơn về phương thức hoạt động của GC, ta có một ví dụ về việc thêm 100,000 person vào 1 list people và xóa toàn bộ list people ngay lập tức.

```
addBtn.setOnClickListener(new View.OnClickListener() {  
  
    @Override  
  
    public void onClick(View view) {
```

```

for (int i = 0; i < 100000; i++) {

    Person person = new Person("new" + i);

    people.add(person);

}

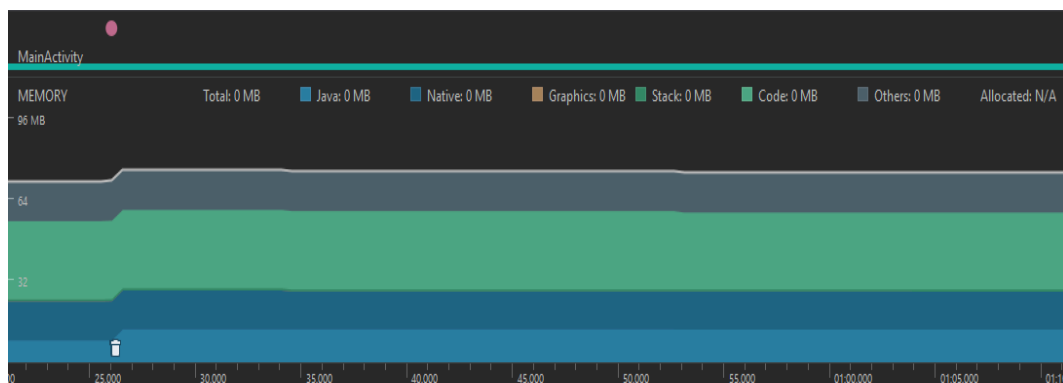
people.clear();

}

});

```

Khi chạy đoạn code và sử dụng memory profiler để kiểm tra, thu được kết quả như sau:

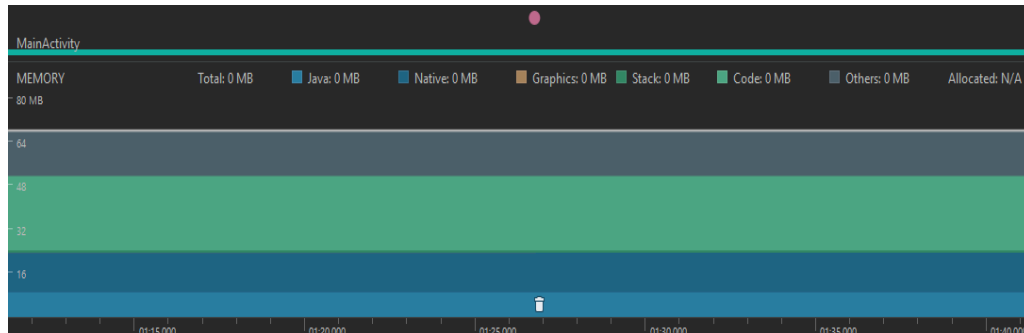


Trong ví dụ trên, có thể thấy rằng GC được gọi 1 lần trong quá trình cấp phát bộ nhớ cho 100,000 person nhưng không thực sự hiệu quả vì các person đều được people tham chiếu đến. Tuy nhiên, sau khi thực hiện people.clear() 1 khoảng thời gian dài, không có bất kỳ GC nào được gọi để loại bỏ 100,000 person đã được tạo trước đó.

Lý do dẫn đến hiện tượng này là do Auto Runtime (ART) sẽ gọi GC khi nó cảm thấy cần thiết. Khi này ART đã đánh giá việc phải giải phóng bộ nhớ để cấp phát 100,000 person là cần thiết nên GC được gọi trong lúc cấp phát. Sau khi xóa bỏ 100,000 person, ART lại đánh giá việc giải phóng bộ nhớ lúc này là không cần thiết nên không thực hiện gọi GC. Việc này dẫn đến việc bộ nhớ không được dọn dẹp sau một khoảng thời gian dài sau đó.

Trong Java có thể sử dụng: `Runtime.getRuntime().gc()` hoặc `System.gc()` để gọi GC nhưng việc này là không được khuyến khích bởi vì việc gọi GC 1 cách chủ động có thể gây ra hiệu suất kém và ảnh hưởng đến trình quản lý bộ nhớ tự động của ART. Một mặt khác, các hàm trên chỉ đưa ra gợi ý về việc sử dụng GC, việc GC có được chạy hay không là do ART quyết định.

Dưới đây là kết quả sau khi sử dụng `Runtime.getRuntime.gc()`:



Một ví dụ khác về cách hoạt động của ARC. Ví dụ này được viết bằng ngôn ngữ Swift, một ngôn ngữ được sử dụng để phát triển phần mềm trên iOS:

```
import Foundation

class Person {

    let name: String

    init(name: String) {

        self.name = name

        print("\(name) is initialized")

    }

    deinit {

        print("\(name) is being deallocated")

    }

}

var reference1: Person?
```

```

var reference2: Person?

var reference3: Person?

reference1 = Person(name: "Alice")

reference2 = reference1

reference3 = reference1

print("Wait for 2 second")

Foundation.sleep(2)

reference1 = nil

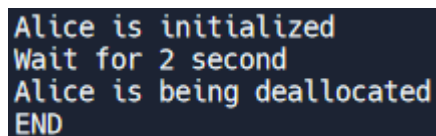
reference2 = nil

reference3 = nil

print("END")

```

Đây là kết quả sau khi chạy đoạn code trên:



```

Alice is initialized
Wait for 2 second
Alice is being deallocated
END

```

Ở ví dụ trên đối tượng Person Alice được tạo, sau đó tạo 3 tham chiếu đến đối tượng. Sau 2 giây, 3 tham chiếu đang có đến Alice được xóa bỏ. Lúc này ART ngay lập tức dọn dẹp Alice vì nó không còn bất kỳ tham chiếu nào đến nó trước khi kết thúc chương trình.

2 ví dụ trên đã làm rõ được phương thức hoạt động của GC và ARC. ARC thực hiện việc dọn rác khi không còn tham chiếu nào đến đối tượng, còn GC chỉ được gọi khi ART nhận thấy rằng việc dọn dẹp là cần thiết.

Với sự khác nhau giữa cách hoạt động, sự ưu việt của ARC được mô tả qua những ví dụ sau:

- Ví dụ về sử dụng dữ liệu lớn như bitmap:

```

public class BitmapTest extends Activity {

    ImageView imageView;

    Bitmap bitmap;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.bitmap_activity);

        imageView = findViewById(R.id.imgView);

        bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.img);

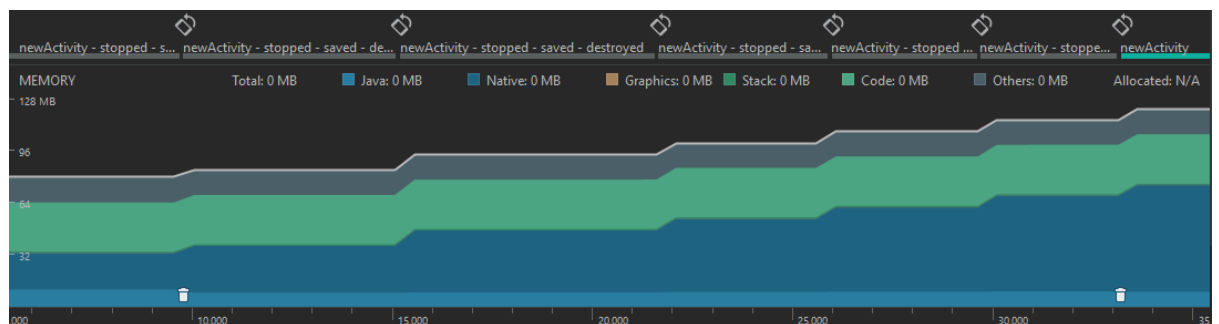
        imageView.setImageBitmap(bitmap);

    }

}

```

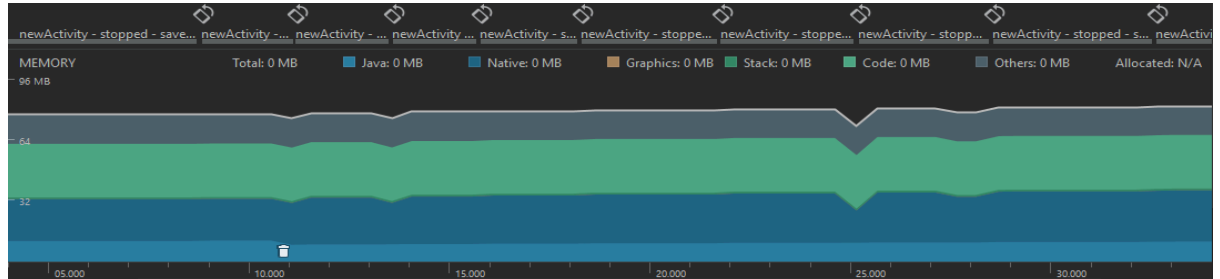
Ví dụ trên là 1 Activity sử dụng bitmap để hiển thị hình ảnh nhưng không thực hiện hiệu tái chế bitmap khi Activity bị hủy. Điều này dẫn đến kết quả như hình bên dưới khi thực hiện việc xoay màn hình liên tục để hủy Activity.



GC sẽ gặp khó khăn trong việc dọn dẹp các Bitmap nếu nó không được tái chế, có thể phải mất một vài lần gọi GC để hoàn toàn dọn dẹp được 1 Bitmap. Vì GC vẫn sẽ dọn dẹp được Bitmap sau một vài lần, nên việc không tái chế Bitmap sẽ không gây ra MemoryLeak. Tuy nhiên, việc không tái chế Bitmap sẽ làm cho bộ nhớ tăng lên liên tục

khi xoay màn hình (như ở hình trên), việc này sẽ khiến cho xảy ra lỗi Out Of Memory khiến cho chương trình bị crash.

Ở dưới đây là kết quả khi thực hiện việc tái chế Bitmap:



- Ví dụ về sử dụng hình ảnh trong Swift:

```
class TestViewController: UIViewController {  
  
    @IBOutlet weak var mainImageView_: UIImageView!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let image = UIImage(named: "eagle")  
        mainImageView_.image = image  
    }  
  
    deinit {  
    }  
}
```

Trong Swift thì việc giải phóng bộ nhớ như trong Android là không cần thiết. Khi class cha bị hủy bỏ, tham chiếu đến mainImageView bị hủy bỏ, referenceCount của mainImageView bị giảm 1 và bằng 0, dẫn đến mainImageView cũng bị hủy bỏ. Kéo theo đó image cũng được hủy bỏ. Do đó, bộ nhớ được giải phóng một cách hoàn toàn tự động.

- Ví dụ về biến local trong Swift

```
import Foundation  
  
class Person {  
  
    let name: String  
  
    init(name: String) {
```

```

        self.name = name

        print("\n(name) is initialized")

    }

    deinit {

        print("\n(name) is being deallocated")

    }

}

do{

    let a = Person(name:"Alice")

    print("Wait")

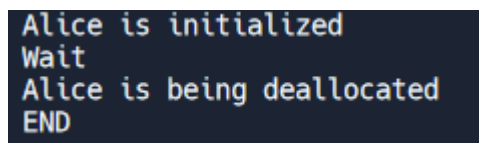
    Foundation.sleep(2)

}

print("END")

```

Kết quả của đoạn code trên :



```

Alice is initialized
Wait
Alice is being deallocated
END

```

Trong ARC, các tham chiếu bị giới hạn trong phạm vi. Phạm vi này có thể là global hay local. Chính vì vậy khi thoát khỏi phạm vi, tham chiếu của các biến bên trong phạm vi sẽ bị hủy bỏ. Trong ví dụ trên Person Alice bị hủy bỏ ngay khi ra khỏi phạm vi của do{ }.

4.5 Tối ưu hóa ảnh

Glide:

Việc sử dụng Glide trong việc tối ưu bộ nhớ của ứng dụng Android có một số ưu điểm quan trọng:

1. Cache hiệu quả: Glide sử dụng cache mạnh mẽ để lưu trữ ảnh đã tải về, giúp giảm băng thông và thời gian tải khi người dùng quay lại xem lại các ảnh đã được tải trước đó. Điều này giúp giảm lượng dữ liệu mạng tiêu tốn và giúp tăng tốc độ tải ảnh.
2. Quản lý bộ nhớ: Glide cung cấp cơ chế quản lý bộ nhớ thông minh, tự động giảm bớt việc sử dụng bộ nhớ khi cần thiết. Nó có khả năng xóa cache không cần thiết để giải phóng bộ nhớ khi bộ nhớ thiếu hụt hoặc cần tái phân phối cho các hoạt động khác.
3. Tối ưu hóa kích thước ảnh: Glide cho phép thiết lập kích thước cụ thể của ảnh để tải về và hiển thị, giúp tránh việc tải về ảnh có kích thước lớn không cần thiết và tiết kiệm bộ nhớ.
4. Hiệu suất cao: Glide được thiết kế để tối ưu hóa việc xử lý và hiển thị ảnh, giúp giảm áp lực lên bộ nhớ và CPU của thiết bị Android. Điều này có ích đặc biệt trên các thiết bị có tài nguyên hạn chế.
5. Giảm thiểu rủi ro memory leaks: Glide giúp giảm thiểu rủi ro memory leaks (rò rỉ bộ nhớ) thông qua việc quản lý cache và việc tải ảnh một cách hiệu quả, giúp ứng dụng duy trì hiệu suất tốt hơn và tránh các vấn đề liên quan đến bộ nhớ.
6. Tùy chọn tùy chỉnh: Glide cung cấp các tùy chọn tùy chỉnh linh hoạt cho việc quản lý bộ nhớ, cho phép người phát triển điều chỉnh các tham số để phù hợp với nhu cầu cụ thể của ứng dụng.



Hình 12: Glide

KẾT LUẬN

Báo cáo đề cập về những chiến lược và phương pháp quan trọng để tối ưu hóa hiệu suất bộ nhớ trên hệ điều hành Android. Việc quản lý bộ nhớ là một thách thức quan trọng đối với các nhà phát triển Android, đặc biệt là trong ngữ cảnh các ứng dụng ngày càng phức tạp và đa nhiệm.

Thông qua các kỹ thuật, chúng ta cũng nhận thức được rằng việc tối ưu hóa bộ nhớ không chỉ giúp cải thiện hiệu suất mà còn giúp tăng trải nghiệm người dùng và giảm tiêu tốn pin. Điều này là quan trọng đặc biệt đối với các thiết bị di động, nơi tài nguyên hạn chế và người dùng đánh giá cao hiệu suất và sự mượt mà.

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 để sử dụng.

Cuối cùng, báo cáo đã trình bày các biện pháp tối ưu hóa bộ nhớ cho ứng dụng Android, và nhấn mạnh tầm quan trọng của việc tích hợp những chiến lược này vào quá trình phát triển để đảm bảo ứng dụng hoạt động mượt mà, hiệu quả và tiết kiệm tài nguyên hệ thống, nâng cao trải nghiệm người dùng.

TÀI LIỆU THAM KHẢO

- [1] Giáo trình môn học
- [2] Bài báo Android Memory Optimization
- [3] <https://developer.android.com/>
- [4] <https://developer.android.com/topic/performance/graphics/cache-bitmap>
- [5] <https://stackoverflow.com/questions/33654503/how-to-use-leak-canary>
- [6] <https://square.github.io/leakcanary/fundamentals-how-leakcanary-works/>
- [7] <https://developer.android.com/reference/androidx/constraintlayout/core/Cache>
- [8] <https://proandroiddev.com/everything-you-need-to-know-about-memory-leaks-in-android-d7a59faaf46a?gi=21c36dd4a0c6>