

Android内存泄漏

讲师：宋征轩

通过本次课程，大家将收获到：

1. Java内存管理中的垃圾回收机制
2. Android应用程序中常见的内存泄漏原因和检查方法
3. 用于检查Android应用程序内存泄漏工具

1. Java程序中的内存泄漏

- Java中的垃圾回收机制
- 一个对象不被GC的原因
- Java内存泄漏的定义

2. 通过MAT工具内存泄漏检测的实例演示

- Hprof内存dump文件如何生成，如何使用MAT去分析

3. Android应用程序中常见的内存泄漏原因和检查方法

4. 进程级别内存占用分析方法

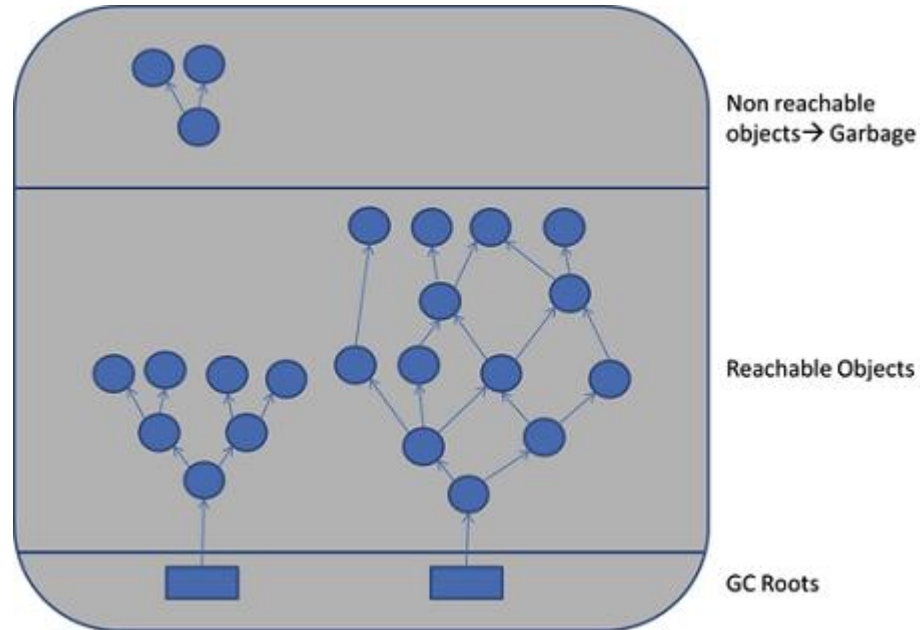
1. Java程序中的内存泄漏



Java基于垃圾回收的内存机制

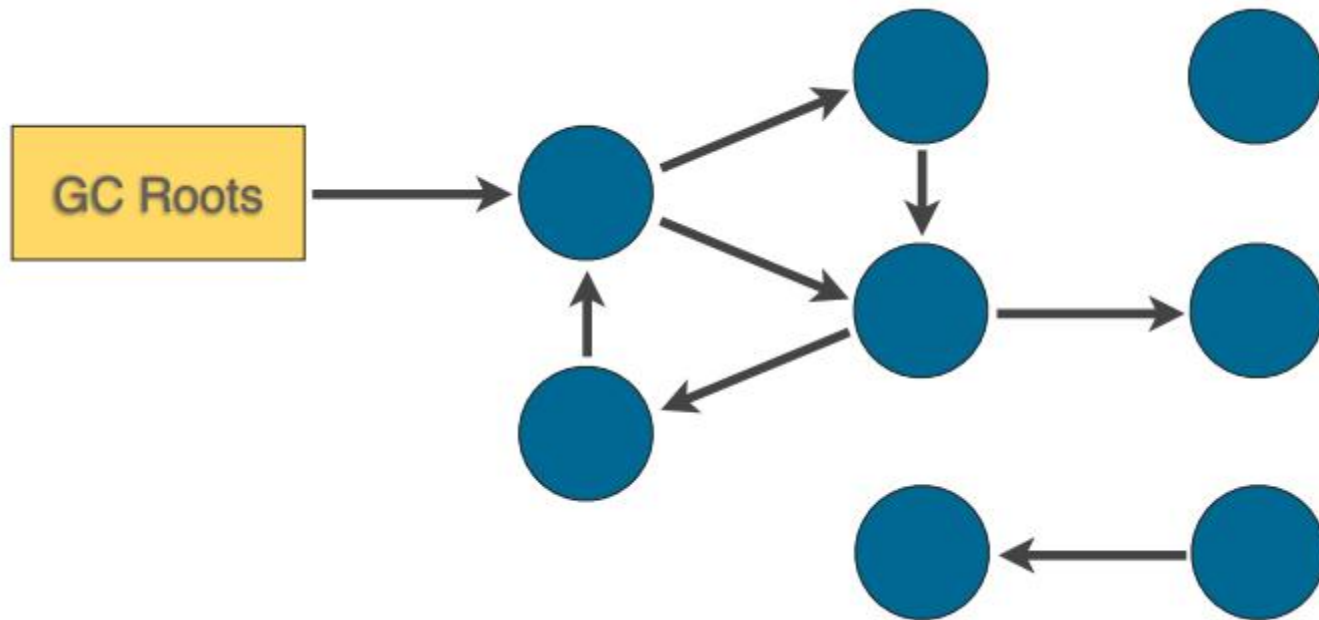
- Java的内存管理机制会自动回收无用对象所占用的内存，减轻手工管理内存的负担
 - C/C++: 从申请、使用、释放都需要手工管理
 - Java: 无用的对象的内存会被自动回收
- 什么样的对象是无用的对象
 - Java通过引用来操作一个具体的对象，引用类似于C 中的指针。一个对象可以持有其他对象的引用。
 - 从一组根对象(GC Roots)开始,按对象之前的引用关系遍历所有对象，在遍历过程中标记所有的可达对象。如果一个对象由根对象出发不可达，则将它作为垃圾收集。

Java程序中的内存泄漏

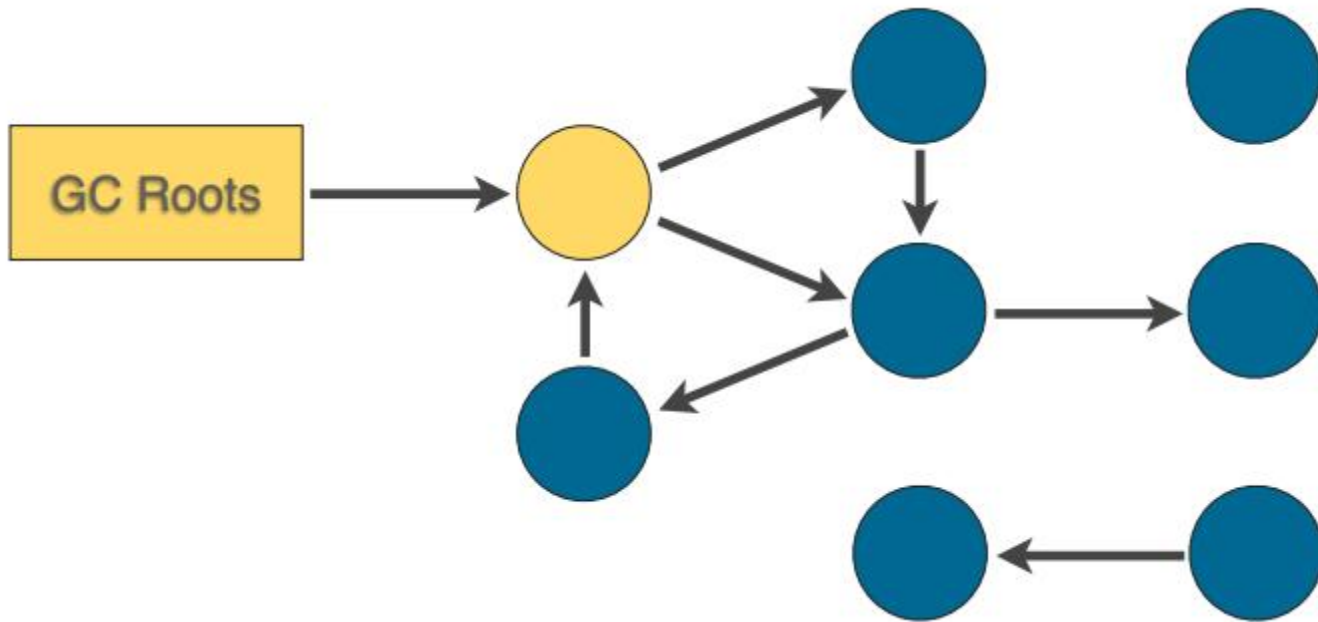


Java基于垃圾回收的内存管理机制

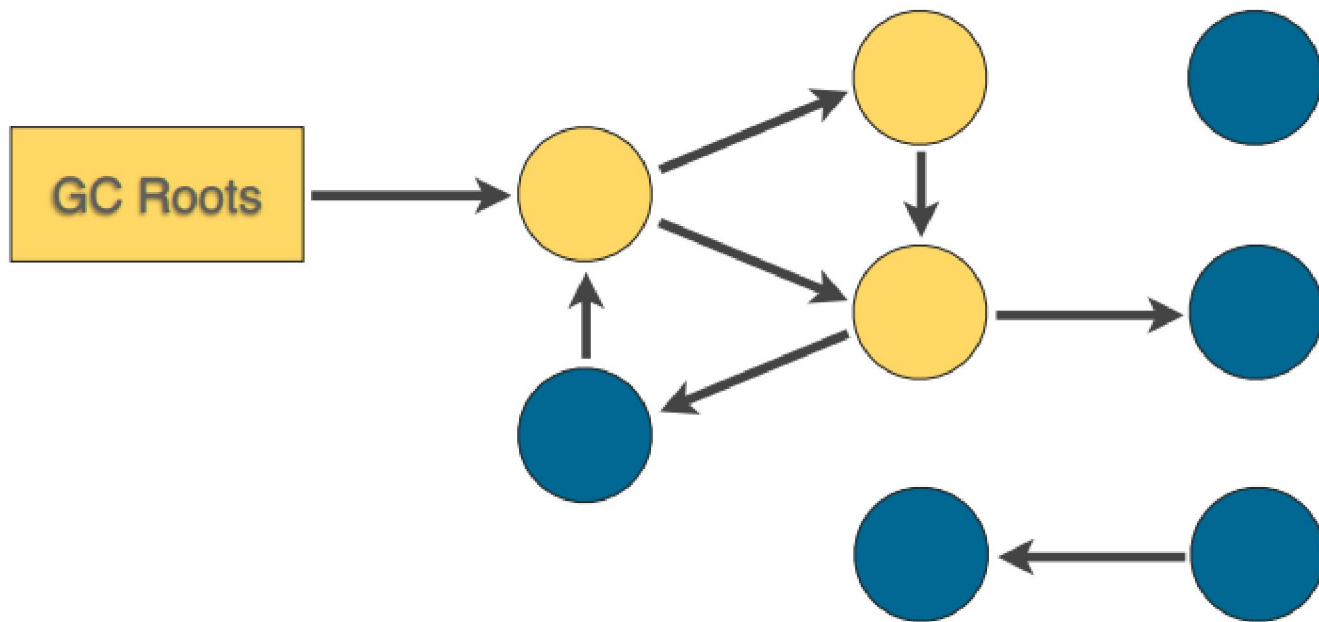
- GC Root 都有哪些？
 - Class: 由系统的类加载器加载的类对象
 - Static Fields
 - Thread: 活着的线程
 - Stack Local: java方法的局部变量或参数
 - JNI Local: JNI方法中的
 - JNI Global: 全局的JNI引用
 - Monitor used: 用于同步的监控对象
 - Help by VM: 用于JVM特殊目的由GC保留的对象



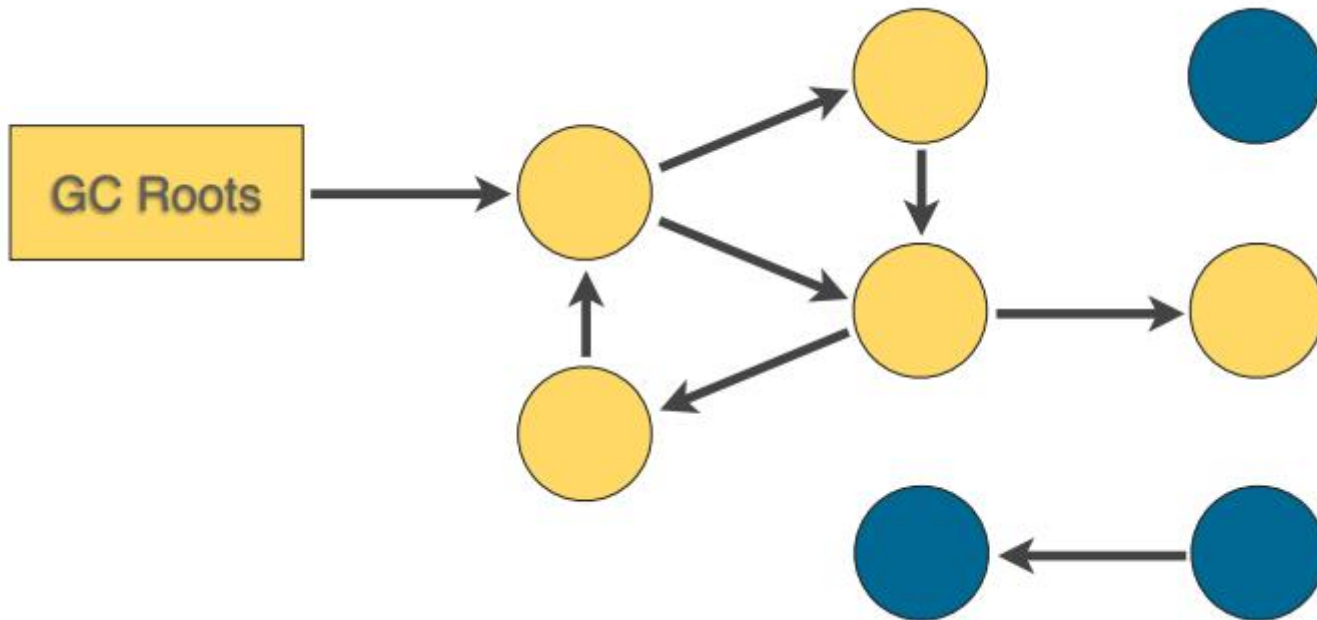
Java程序中的内存泄漏



Java程序中的内存泄漏



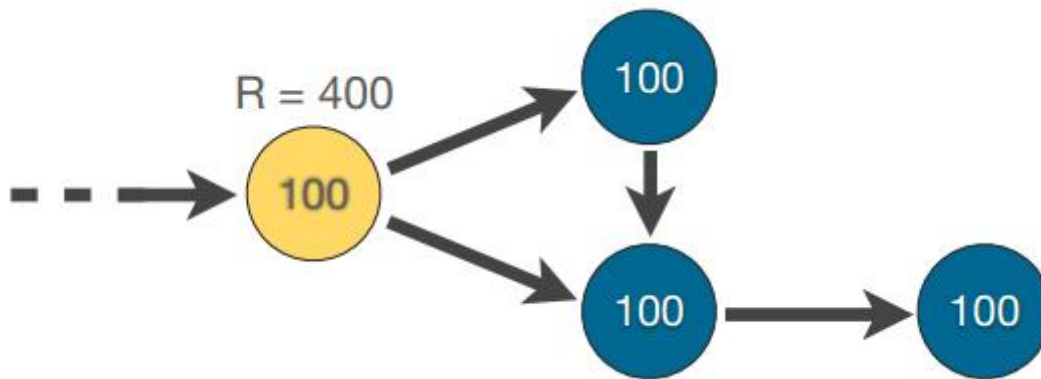
Java程序中的内存泄漏



Shallow Heap 和 Retained Heap

Shallow Heap: 对象本身所占用的内存

Retained Heap: 对象被GC时，能够连带被GC的内存总和



Java程序中的内存泄漏

- 对象的内存在分配之后无法通过程序的执行逻辑释放对该对象的引用，不能被回收该对象所占内存
- 内存泄漏的危害
 - 引起OutOfMemoryError
 - 内存占用高时JVM虚拟机会频繁触发GC, 影响程序响应速度
 - 内存占用大的程序容易被各种清理优化程序中止，用户也更倾向于卸载这些程序

Android应用可用内存限制

Android应用的开发语言为Java，每个应用最大可使用的堆内存受到Android系统的限制

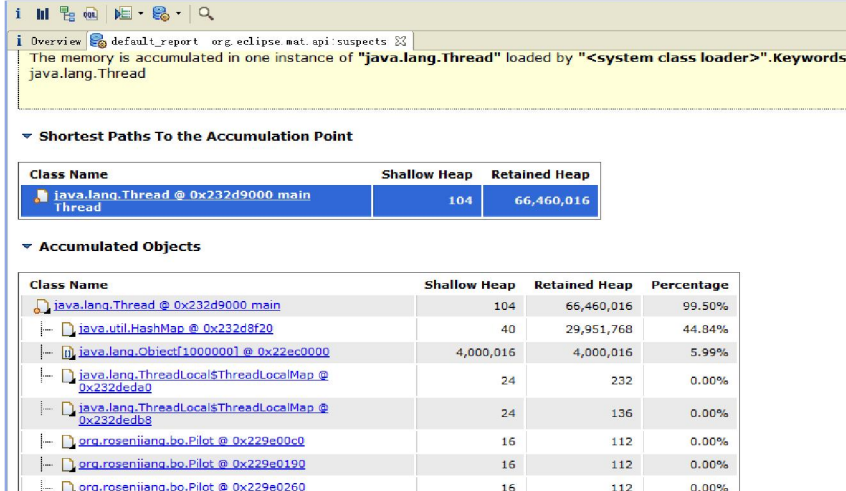
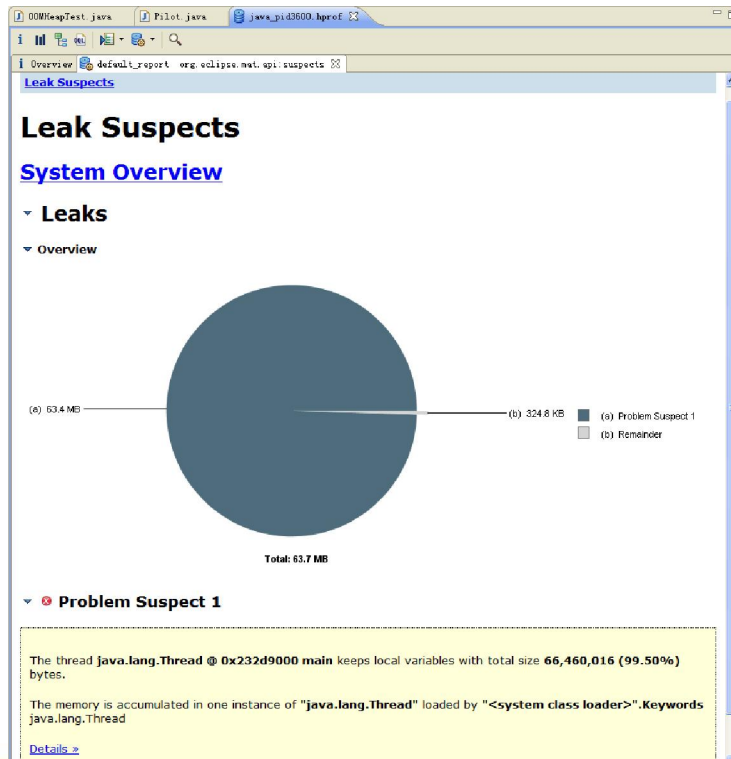
- Android每一个应用的堆内存大小有限
 - 通常的情况为16M-48M
 - 通过ActivityManager的getMemoryClass()来查询可用堆内存限制
 - 3.0(HoneyComb)以上的版本可以通过largeHeap="true"来申请更多的堆内存
 - Nexus S(4.2.1):normal 192, largeHeap 512
- 如果试图申请的内存大于当前余下的堆内存就会引发OutOfMemoryError()
- 应用程序由于各方面的限制，需要注意减少内存占用，避免出现内存泄漏。

2. 通过MAT工具 内存泄漏检测的实例演示

Heap dump

- 包含了触发Heap dump生成的时刻Java进程的内存快照，主要内容为各个Java类和对象在堆内存中的分配情况

Memory Analyzer Tool (MAT)



The figure shows the 'Shortest Paths To the Accumulation Point' section of the Memory Analyzer Tool (MAT) interface. It displays a table with two columns: 'Class Name' and 'Shallow Heap'. The table lists the following classes and their shallow heap sizes:

Class Name	Shallow Heap	Retained Heap
java.lang.Thread @ 0x232d9000 main Thread	104	66,460,016

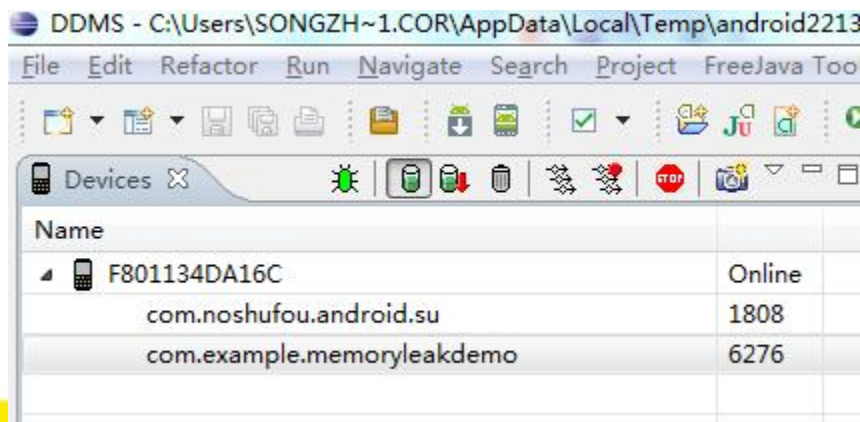
Below the table, the 'Accumulated Objects' section displays a table with four columns: 'Class Name', 'Shallow Heap', 'Retained Heap', and 'Percentage'. The table lists the following classes and their accumulated object sizes:

Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0x232d9000 main	104	66,460,016	99.50%
java.util.HashMap @ 0x232d8f20	40	29,951,768	44.84%
java.lang.Object[1000000] @ 0x22ec0000	4,000,016	4,000,016	5.99%
java.lang.ThreadLocal\$ThreadLocalMap @ 0x232d8da0	24	232	0.00%
java.lang.ThreadLocal\$ThreadLocalMap @ 0x232d8db8	24	136	0.00%
org.rosenjiang.bo.Pilot @ 0x229a00c0	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229a0190	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229a0260	16	112	0.00%

MAT工具的作用

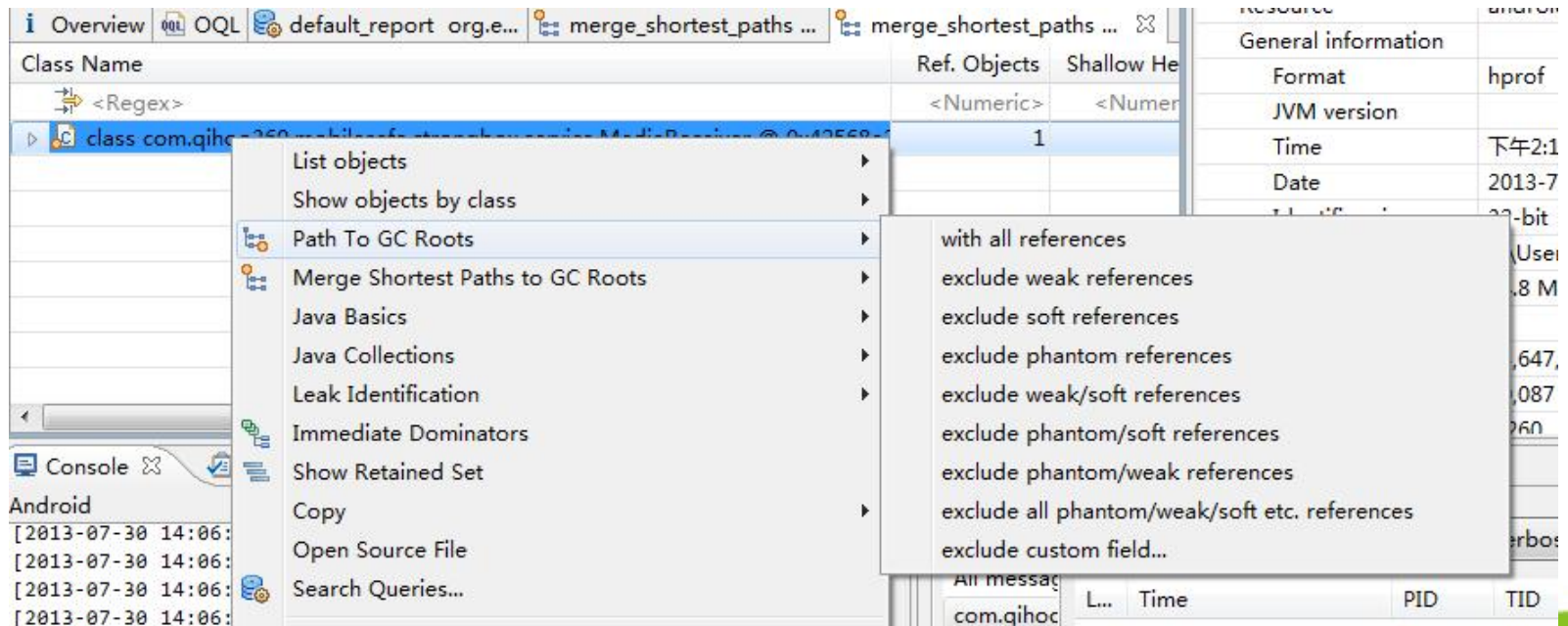
- 通过MAT工具，可以解析heap dump内存映象文件，列出各个类、对象所占用的内存大小。
- MAT工具也会列出可能的发生内存泄漏的对象，并且可以查看每一个对象的GC Root，定位一个对象不被GC的原因。
- 支持OQL对象查询语言
- <http://download.eclipse.org/mat/1.3/update-site/>

Android程序HPROF内存映象的生成



MAT工具的使用

- 了解各个类、对象都占用的多少内存
 - Shallow Heap. 对象自身占用内存的大小
 - Retained Heap 对象自身的大小和对象可直接或间接引用到的对象大小总和。
- 各个对象不被GC的原因



MAT中的OQL

- 查询一个class的所有实例对象
 - `SELECT * FROM INSTANCEOF java.lang.ref.Reference`
- 查询类名符合一正则表达式的所有对象
 - `SELECT * FROM "java\lang\..*"`
- 子查询
 - `SELECT * FROM (SELECT * FROM java.lang.Class c WHERE c implements org.eclipse.mat.snapshot.model.IClass)`
- 指定具体列名
 - `SELECT toString(s), s.@usedHeapSize, s.@retainedHeapSize FROM java.lang.String s.`

3. Android应用中的内存泄漏



1.1 Context对象泄漏

- 如果一个类持有Context对象的强引用，就需要检查其生存周期是否比Context对象更长。否则就可能发生Context泄漏。
- View持有其创建所在Context对象的引用，如果将View对象传递给其它生存周期比View所在Context更长的强引用，就可能会引起内存泄漏。
 - 例如View#setTag(int, Object)的内存泄漏
<https://code.google.com/p/android/issues/detail?id=18273>
- 把Context对象赋给static变量。

Android应用中的内存泄漏

```
7724     private static void ↴ setTagInternal(View view, int key, Object tag) {  
7725         SparseArray<Object> tags = null;  
7726         synchronized (sTagsLock) {  
7727             if (sTags == null) {  
7728                 sTags = new WeakHashMap<View, SparseArray<Object>>();  
7729             } else {  
7730                 tags = sTags.get(view);  
7731             }  
7732         }  
7733  
7734         if (tags == null) {  
7735             tags = new SparseArray<Object>(2);  
7736             synchronized (sTagsLock) {  
7737                 sTags.put(view, tags);  
7738             }  
7739         }  
7740  
7741         tags.put(key, tag);  
7742     }
```

1.2 避免Context对象泄漏Checklist

- 检查所有持有对Context对象强引用的对象的生命周期是否超出其所持有的Context对象的生命周期。
- 检查有没有把View传出到View所在Context之外的地方，如果有的话就需要检查生命周期。
- 工具类中最好不要有Context成员变量，尽量在调用函数时直接通过调用参数传入。如果必须有Context成员变量时，可以考虑使用WeakReference来引用Context对象。
- View持有其创建所在Context对象的引用，如果将View对象传递给其它生存周期比View所在Context更长的强引用，就可能会引起内存泄漏。
- 检查把Context或者View对象赋给static变量的地方，看是否有Context泄漏。
- 检查所有把View放入容器类的地方（特别是static容器类），看是否有内存泄漏。
 - 使用WeakHashMap也需要注意有没有value-key的引用。
- 尽量使用ApplicationContext。

2.1 Handler对象泄漏

```
1 public class SampleActivity extends Activity {  
2  
3     private final Handler mLeakyHandler = new Handler() {  
4         @Override  
5         public void handleMessage(Message msg) {  
6             // ...  
7         }  
8     }  
9 }
```

- 发送到Handler的Message实际上是加入到了主线程的消息队列等待处理，每一个Message持有其目标Handler的强引用。
- Non-static inner class 和anonymous class持有其outer class的引用。

Android应用中的内存泄漏

```
1 public class SampleActivity extends Activity {
2
3     private final Handler mLeakyHandler = new Handler() {
4         @Override
5         public void handleMessage(Message msg) {
6             // ...
7         }
8     }
9
10    @Override
11    protected void onCreate(Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13
14        // Post a message and delay its execution for 10 minutes.
15        mLeakyHandler.postDelayed(new Runnable() {
16            @Override
17            public void run() { }
18        }, 600000);
19
20        // Go back to the previous Activity.
21        finish();
22    }
23 }
```

2.2 避免Handler对象泄漏的方法

- 注意编译时的警告，把non-static Handler改为static
- 如果Handler.handlerMessage()中必须调用其所在Context的非静态方法时。可以在Handler中通过WeakReference保存其所在Context，通过该WeakReference中保存的Context来调用。

3.1 Drawable.Callback引起的内存泄漏

- Drawable对象持有Drawable.callback的引用。当把一个Drawable对象设置到一个View时，Drawable对象会持有该View的引用作为Drawable.Callback

```
7203     public void setBackgroundDrawable(Drawable d) {  
7204         boolean requestLayout = false;  
7205  
7206         mBackgroundResource = 0;  
7207  
  
7233  
7234         d.setCallback(this);  
7235         if (d.isStateful()) {  
7236             d.setState(getDrawableState());  
7237         }
```

3.2 避免Drawable.Callback引起内存泄漏

- 尽量不要在static成员中保存Drawable对象
- 对于需要保存的Drawable对象，在需要时调用Drawable#setCallback(null).

4. 其他内存泄漏

- Android DigitalClock引起的内存泄漏
<http://code.google.com/p/android/issues/detail?id=17015>
- 使用Map容器类时，作为Key 的类没有正确的实现 hashCode和equal函数

```
class BadKeyClass {  
    public final String memberA;  
  
    public BadKeyClass(String memberA) {  
        this.memberA = memberA;  
    }  
}  
  
private static final HashMap<BadKeyClass, Object> sMaps  
    = new HashMap<BadKeyClass, Object>();
```

4. 其他内存泄漏

- JNI程序中的内存泄漏
 - Malloc/free。
 - JNI Global reference
- Thread-Local Variable
 - 相当于Thread对象的成员变量，可以存储线程相关的状态
 - 如果thread是alive状态，那么Thread-Local中的对象就无法被GC。

4. 进程内存占用监测工具

5.1 Dumpsys

- \$ dumsys meminfo [pid]

```
** MEMINFO in pid 2715 [com.miui.mihome.lockscreen] **
```

	Pss	Shared Dirty	Private Dirty	Heap Size	Heap Alloc	Heap Free
	-----	-----	-----	-----	-----	-----
Native	53	48	52	4596	4385	134
Dalvik	10953	11492	10556	21255	15088	6167
Cursor	0	0	0			
Ashmem	0	0	0			
Other dev	5748	52	5744			
.so mmap	1132	2324	884			
.jar mmap	0	0	0			
.apk mmap	58	0	0			
.ttf mmap	0	0	0			
.dex mmap	436	4	0			
Other mmap	125	20	112			
Unknown	2849	360	2844			
TOTAL	21354	14300	20192	25851	19473	6301

Objects

Views:	15	ViewRootImpl:	5
AppContexts:	4	Activities:	3
Assets:	3	AssetManagers:	3
Local Binders:	63	Proxy Binders:	33

5.2 Procrank + Shell脚本

- #procrank
 - VSS - Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
 - RSS - Resident Set Size 实际使用物理内存（包含共享库占用的内存）
 - PSS - Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
 - USS - Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）

```
# procrank
procrank
PID      Vss      Rss      Pss      Uss      cmdline
 52    44896K    43396K   23927K   19704K   system_server
 97    29568K    29568K   10836K    7464K   android.process.acore
 30    27052K    27052K    8559K    5468K   zygote
 94    23500K    23500K    6120K    3548K   com.android.phone
 92    21052K    21052K    4653K    2776K   com.android.inputmethod.pinyin
190    20736K    20736K    4211K    2272K   com.android.email
152    20372K    20372K    3937K    2144K   android.process.media
116    20156K    20156K    3871K    1800K   com.android.settings
176    20340K    20340K    3866K    1964K   com.android.mms
128    19748K    19748K    3325K    1492K   com.android.alarmclock
```

- Shell 脚本

```
#!/bin/bash
```

```
while true; do  
    adb shell procrank | grep  
    "com.qihoo360.mobilesafe"  
    sleep 1  
done
```

小结

1. 保存对象前要三思
 - I. 对象本身有无隐含的引用
 - II. 保存后何时能够回收
2. 要了解常见的隐含引用
 - I. non-static anonymous inner class to outer
 - II. View to context
3. 要通过各种工具检查内存占用是否有异常
4. 创建大对象时，要检查它的生命周期

推荐网站，论坛，书籍：

1. <http://stackoverflow.com/> 开发问题
2. <http://greppcode.com> 阅读Android源代码
3. <http://code.google.com/p/android> Android bugs
4. <http://www.yourkit.com/docs/kb/sizes.jsp> java
虚拟机的内存结构

Q&A



谢谢！

