20180418 JVM 内存泄漏分析和解决

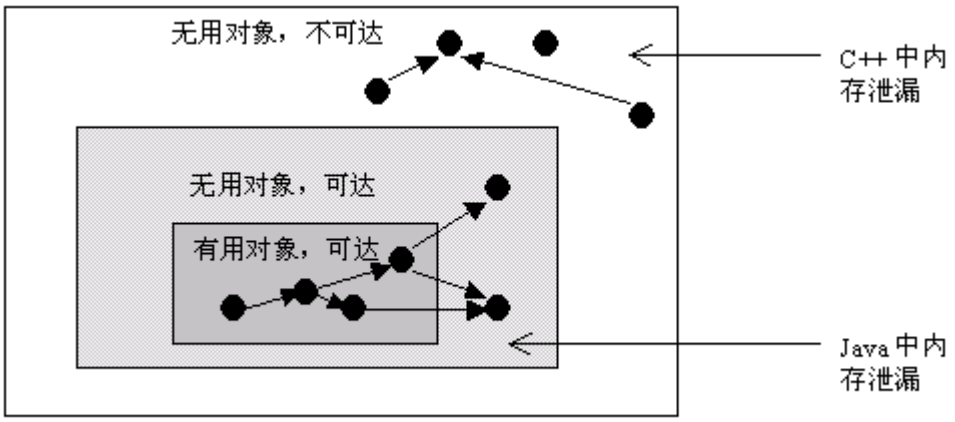
原文： https://www.jianshu.com/p/54b5da7c6816

1. 什么是内存泄漏？

> **内存泄漏**：对象已经没有被应用程序使用，但是垃圾回收器没办法移除它们，因为还在被引用着。 在Java中，**内存泄漏**就是存在一些被分配的对象，这些对象有下面两个特点，**首先**，这些对象是可达的，即**在有向图中，存在通路可以与其相连**；**其次**，**这些对象是无用的，即程序以后不会再使用这些对象**。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC（**Garbage Collection垃圾回收**），这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于C++，程序员需要自己管理边和顶点，而对于Java程序员只需要管理边就可以了(不需要管理顶点的释放)。通过这种方式，Java提高了编程的效率。



C++与Java当中的内存泄漏

因此，通过以上分析，我们知道在Java中也有内存泄漏，但范围比C++要小一些。因为Java从语言上保证，任何对象都是可达的，所有的不可达对象都由GC管理。

对于程序员来说，GC基本是透明的，不可见的。虽然，我们只有几个函数可以访问GC，例如运行GC的函数System.gc()，但是根据Java语言规范定义， 该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级别较低。JVM调用GC的策略也有很多种，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，我们不需要关心这些。除非在一些特定的场合，GC的执行影响应用程序的性能，例如对于基于Web的实时系统，如网络游戏等，用户不希望GC突然中断应用程序执行而进行垃圾回收，那么我们需要调整GC的参数，让GC能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun提供的HotSpot JVM就支持这一特性。

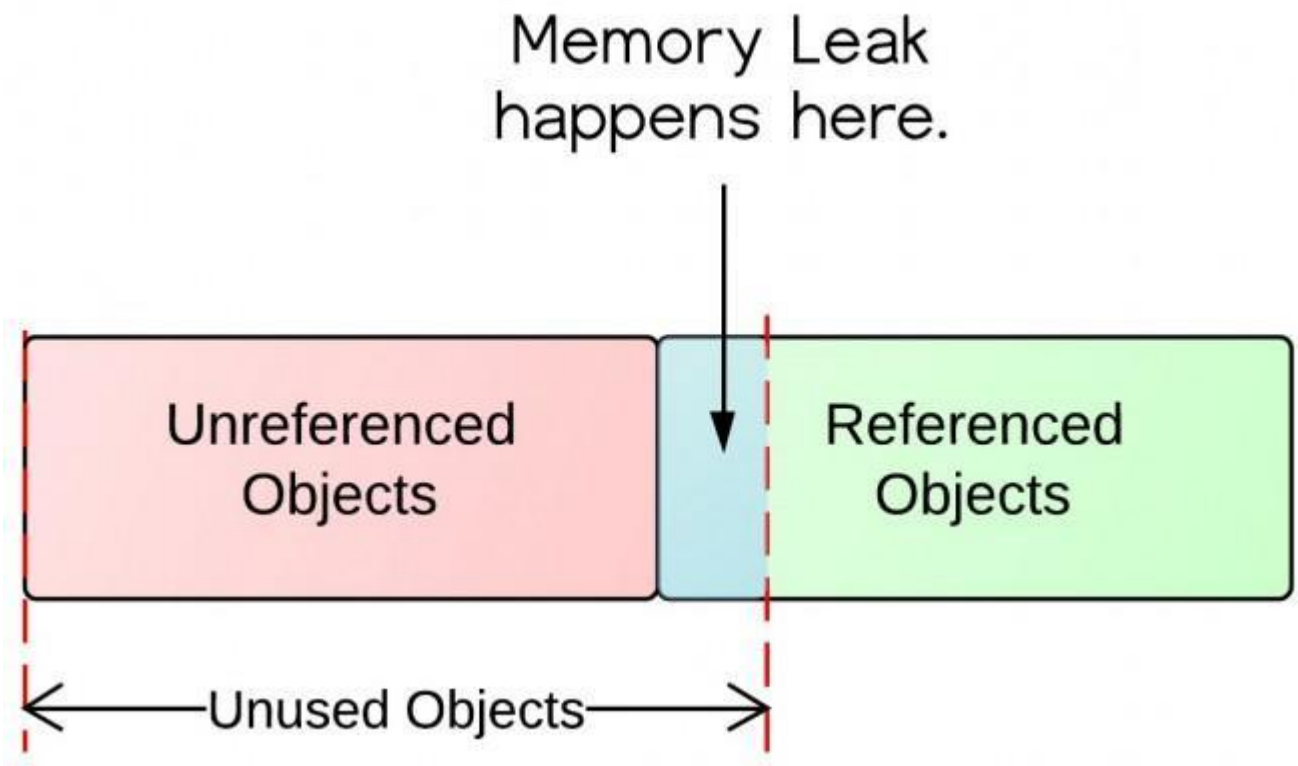
下面给出一个 Java 内存泄漏的典型例子，

```
Vector v = new Vector(10);

for (int i = 0; i < 100; i++) {
    Object o = new Object();
    v.add(o);
    o = null;
}

```

在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个 Vector 中，如果我们仅仅释放引用本身，那么 Vector 仍然引用该对象，所以这个对象对 GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从 Vector 中删除，最简单的方法就是将 Vector 对象设置为 null。

```
v = null

```

要想理解这个定义，我们需要先了解一下对象在内存中的状态。下面的这张图就解释了什么是**无用对象**以及什么是**未被引用对象**。



内存泄漏示意图

引用对象

未被引用对象

2. 详细Java中的内存泄漏

2.1 Java内存回收机制

不论哪种语言的内存分配方式，都需要返回所分配内存的真实地址，也就是返回一个指针到内存块的首地址。Java中对象是采用new或者反射的方法创建的，这些对象的创建都是在堆（Heap）中分配的，所有对象的回收都是由Java虚拟机通过垃圾回收机制完成的。GC为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控，Java会使用有向图的方法进行管理内存，实时监控对象是否可以达到，如果不可到达，则就将其回收，这样也可以消除引用循环的问题。在Java语言中，判断一个内存空间是否符合垃圾收集的标准有两个：**一个是给对象赋予了空值null，以下再没有调用过另一个是给对象赋予了新值，这样重新分配了内存空间。**

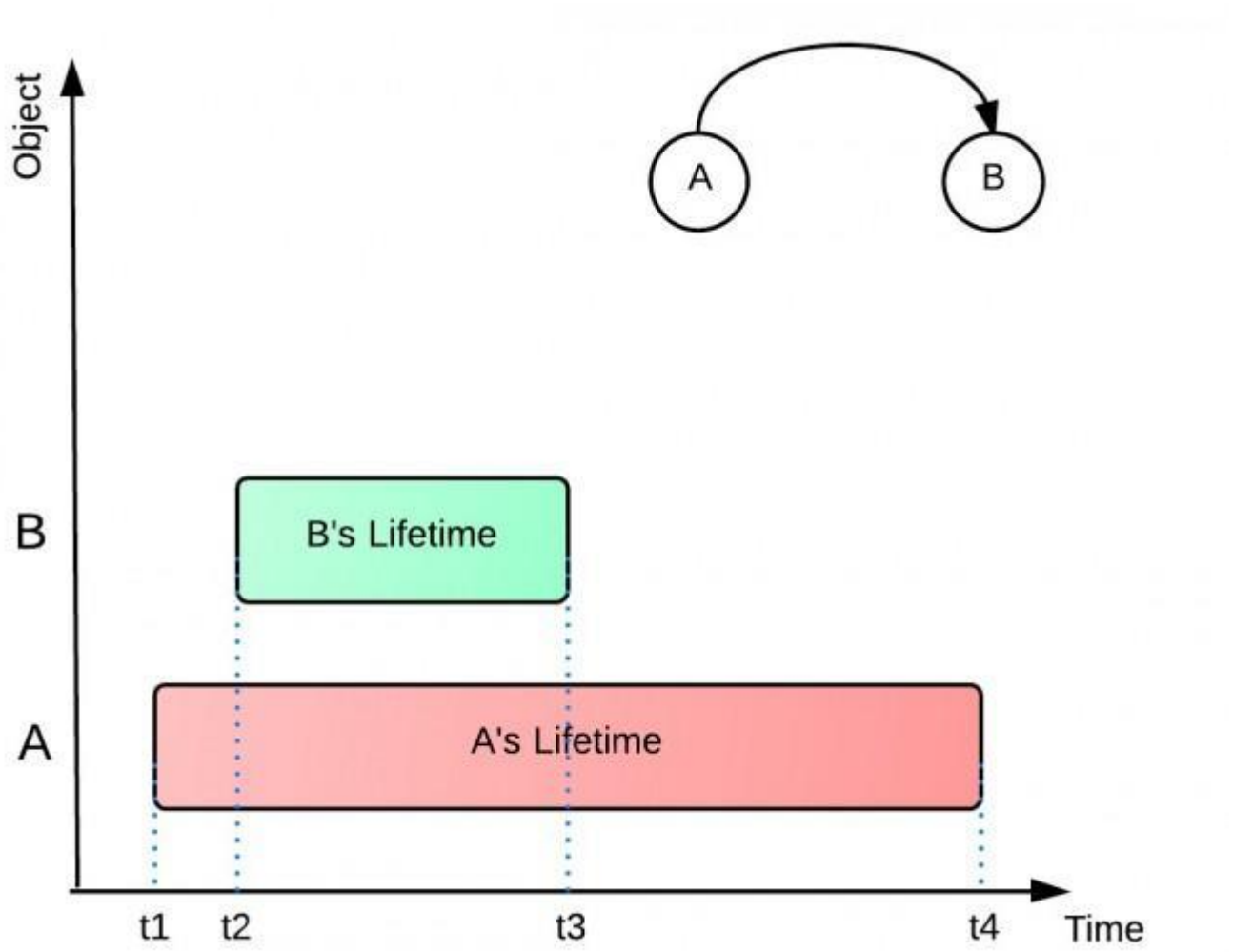
2.2 Java内存泄漏引起的原因

> 内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。

Java内存泄漏的根本原因是什么呢？长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是**因为长生命周期持有它的引用而导致不能被回收，这就是Java中内存泄漏的发生场景。**

来先看看下面的例子，为什么会发生内存泄漏。下面这个例子中，A对象引用B对象，A对象的生命周期（t1-t4）比B对象的生命周期（t2-t3）长的多。当B对象没有被应用程序使用之后，A对象仍然在引用着B对象。这样，垃圾回收器就没办法将B对象从内存中移除，从而导致内存问题，因为如果A引用更多这样的对象，那将有更多的未被引用对象存在，并消耗内存空间。

B对象也可能会持有许多其他的对象，那这些对象同样也不会被垃圾回收器回收。所有这些没在使用的对象将持续的消耗之前分配的内存空间。



生命周期图

具体主要有如下几大类：

2.2.1 静态集合类引起内存泄漏

像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。

例如：

```
Static Vector v = new Vector(10);

for (int i = 0; i < 100; i++) {
    Object o = new Object();
    v.add(o);
    o = null;
}

```

在这个例子中，循环申请Object 对象，并将所申请的对象放入一个Vector 中，如果仅仅释放引用本身（o=null），那么Vector 仍然引用该对象，所以这个对象对GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从Vector 中删除，最简单的方法就是将Vector对象设置为null。

2.2.2 监听器

在 **java** 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如 **addXXXListener()** 等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

2.2.3 各种连接

比如数据库连接（**dataSourse.getConnection()**），网络连接(**socket**)和**io**连接，除非其显式的调用了其**close()** 方法将其连接关闭，否则是不会自动被**GC** 回收的。对于**Resultset** 和**Statement** 对象可以不进行显式回收，但**Connection** 一定要显式回收，因为**Connection** 在任何时候都无法自动回收，而**Connection**一旦回收，**Resultset** 和**Statement** 对象就会立即为**NULL**。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭**Resultset Statement** 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的**Statement** 对象无法释放，从而引起内存泄漏。这种情况下一般都会在**try** 里面去的连接，在**finally**里面释放连接。

2.2.4 内部类和外部模块的引用

内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：

```
public void registerMsg(Object b);

```

这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B是否提供相应的操作去除引用。

2.2.5 单例模式

不正确使用单例模式是引起内存泄漏的一个常见问题，单例对象在初始化后将在 **JVM** 的整个生命周期中存在（**以静态变量的方式**），如果单例对象持有外部的引用，那么这个对象将不能被 **JVM** 正常回收，导致内存泄漏，考虑下面的例子：

```
public class A {
    public A() {
        B.getInstance().setA(this);
    }

    ...
```

```
}

//B类采用单例模式
class B{
    private A a;
    private static B instance = new B();

    public B(){}

    public static B getInstance() {
        return instance;
    }

    public void setA(A a) {
        this.a = a;
    }

    public A getA() {
        return a;
    }
}

```

3. Java 内存分配策略

Java 程序运行时的内存分配策略有三种,分别是**静态分配,栈式分配**,和**堆式分配**，对应的，三种存储策略使用的内存空间主要分别是**静态存储区（也称方法区）、栈区和堆区**。

静态存储区（方法区）：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。

栈区 ：当方法被执行时，方法体内的局部变量（其中包括**基础数据类型、对象的引用**）都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆区 ： 又称**动态内存分配**，通常就是指在程序运行时直接 new 出来的内存，也就是对象的实例。这部分内存在不使用时将会由 **Java 垃圾回收器**来负责回收。

3.1 栈与堆的区别

在方法体内定义的（**局部变量**）一些基本类型的变量和对象的引用变量都是在方法的**栈内存**中分配的。当在一段方法块中定义一个变量时，Java 就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。

堆内存用来存放所有由 new 创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由 **Java 垃圾回收器**来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变量。我们可以通过这个引用变量来访问堆中的对象或者数组。

举个栗子：

```
public class Sample {
    int s1 = 0;
    Sample mSample1 = new Sample();

    public void method() {
        int s2 = 1;
        Sample mSample2 = new Sample();
    }
}
Sample mSample3 = new Sample();

```

Sample 类的局部变量 s2 和引用变量 mSample2 都是存在于栈中，但 mSample2 指向的对象是存在于堆上的。

mSample3 指向的对象实体存放在堆上，包括这个对象的所有成员变量 s1 和 mSample1，而它自己存在于栈中。

> **结论**： **局部变量**的基本数据类型和引用存储于**栈**中，引用的对象实体存储于**堆**中。—— 因为它们属于方法中的变量，生命周期随方法而结束。 **成员变量**全部存储于**堆**中（*包括基本数据类型，引用和引用的对象实体*）—— 因为它们属于类，类对象终究是要被new出来使用的。

了解了 Java 的内存分配之后，我们再来看看 Java 是怎么管理内存的。

3.2 Java如何管理内存

Java的内存管理就是**对象的分配和释放问题**。在 Java 中，程序员需要通过关键字 new 为每个对象申请内存空间(基本类型除外)，所有的对象都在**堆 (Heap)**中分配空间。另外，对象的释放是由 GC 决定和执行的。在 Java 中，内存的分配是由程序完成的，而内存的释放是由 GC 完成的，这种收支两条线的方法确实简化了程序员的工作。但同时，它也加重了JVM的工作。这也是 **Java 程序运行速度较慢的原因之一**。因为**GC 为了能够正确释放对象，GC 必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控**。

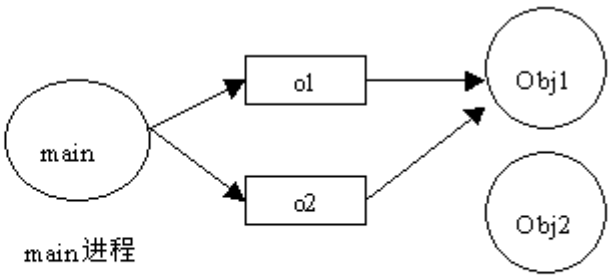
监视对象状态是为了更加准确地、及时地释放对象，而**释放对象的根本原则就是该对象不再被引用**。

为了更好理解 GC 的工作原理，我们可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引对象。另外，每个线程对象可以作为一个图的起始顶点，例如大多程序从 main 进程开始执行，那么该图就是以 main 进程顶点开始的一棵根树。在这个有向图中，根顶点可达的对象都是有效对象，GC将不回收这些对象。如果某个对象 (连通子图)与这个根顶点不可达(注意，该图为有向图)，那么我们认为这个(这些)对象不再被引用，可以被 GC 回收。

以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每一个时刻，我们都有一个有向图表示JVM的内存分配情况。以下右图，就是左边程序运行到第6行的示意图。

```
public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Object o1 = new Object();
        Object o2 = new Object();
        o2 = o1;//此行为第6行
    }
}

```

```
class test{
 Public static void  main(String a[]){
          Object  o1  =new Object();
          Object  o2  =new Object();
          o2=o1;
          //此行为第 6 行
          }
}
```



该图描述了第 6 行的内存管理的有向图，
Obj2 是第二次申请的对象，此时为可回收对象

有向图

4. 如何防止内存泄漏的发生？

在了解了引起内存泄漏的一些原因后，应该尽可能地避免和发现内存泄漏。

4.1 好的编码习惯

最基本的建议就是尽早释放无用对象的引用，大多数程序员在使用临时变量的时候，都是让引用变量在退出活动域后，自动设置为 **null** 。在使用这种方式时候，必须特别注意一些复杂的对象图，例如数组、列、树、图等，这些对象之间有相互引用关系较为复杂。对于这类对象，**GC** 回收它们一般效率较低。如果程序允许，尽早将不用的引用对象赋为null。另外建议几点：

在确认一个对象无用后，将其所有引用显式的置为null；

当类从 **Jpanel** 或 **Jdialog** 或其它容器类继承的时候，删除该对象之前不妨调用它的 **removeall()** 方法；在设一个引用变量为 **null** 值之前，应注意该引用变量指向的对象是否被监听，若有，要首先除去监听器，然后才可以赋空值；当对象是一个 **Thread** 的时候，删除该对象之前不妨调用它的 **interrupt()** 方法；内存检测过程中不仅要关注自己编写的类对象，同时也要关注一些基本类型的对象，例如：**int[]、String、char[]** 等等；如果有数据库连接，使用 **try...finally** 结构，在 **finally** 中关闭 **Statement** 对象和连接。

4.2 好的测试工具

在开发中不能完全避免内存泄漏，关键要在发现有内存泄漏的时候能用好的测试工具迅速定位问题的所在。市场上已有几种专业检查 **Java** 内存泄漏的工具，它们的基本工作原理大同小异，都是通过监测 **Java** 程序运行时，所有对象的申请、释放等动作，将内存管理的所有信息进行统计、分析、可视化。开发人员将根据这些信息判断程序是否有内存泄漏问题。这些工具包括 **Optimizeit Profiler、JProbe Profiler、JinSight、Rational** 公司的 **Purify** 等。

4.3 注意像 HashMap 、ArrayList 的集合对象

特别注意一些像 **HashMap** 、**ArrayList** 的集合对象，它们经常会引发内存泄漏。当它们被声明为 **static** 时，它们的生命周期就会和应用程序一样长。

4.4 注意 事件监听 和 回调函数

特别注意 **事件监听** 和 **回调函数** 。当一个监听器在使用的时候被注册，但不再使用之后却未被反注册。

“如果一个类自己管理内存，那开发人员就得小心内存泄漏问题了。” 通常一些成员变量引用其他对象，初始化的时候需要置空。

> 参考文章： **1.介绍Java中的内存泄漏 2.Java的内存泄漏 3.Java中关于内存泄漏出现的原因汇总及如何避免内存泄漏 4.Java内存泄漏的几大原因及预防检测**