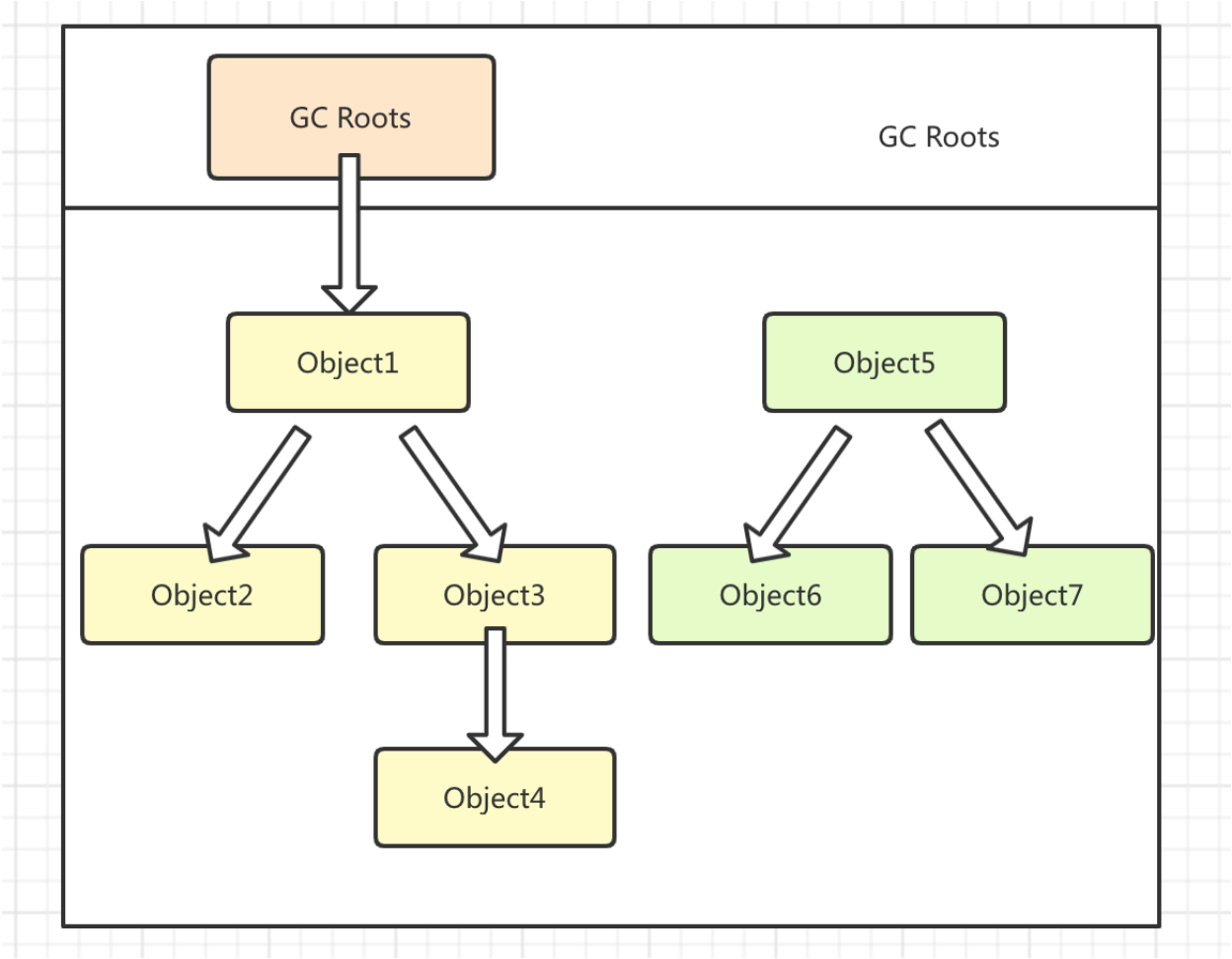
Java中的内存泄漏总结

可达性分析算法

JVM使用**可达性分析算法**判断对象是否存活。

GC Root

通过一系列名为“GC Roots”的对象作为起点，从这些结点开始向下搜索，搜索所走过的路径称为“引用链（Reference Chain）”，当一个对象到GC Roots没有任何饮用链相连时，则证明此对象是不可用的。



object4、object5、object6虽然有互相判断，但是它们到GC Rootd是不可达的，所以它们将会判定为是可回收对象。

可以作为GC Roots的对象有：

- 虚拟机栈（栈帧中的本地变量表）中的引用的对象；
- 方法区中的类静态属性引用的对象；
- 方法区中的常量引用的对象；
- 本地方法栈中JNI的引用的对象

虽然Java有垃圾收集器帮组实现内存自动管理，虽然GC有效的处理了大部分内存，但是并不能完全保证内存的不泄漏。

内存泄漏

内存泄漏就是堆内存中不再使用的对象无法被垃圾收集器清除掉，因此它们会不必要地存在。这样就导致了内存消耗，降低了系统的性能，最终导致OOM使得进程终止。

内存泄漏的表现：

- 应用程序长时间连续运行时性能严重下降；
- 应用程序中的OutOfMemoryError堆错误；
- 自发且奇怪的应用程序崩溃；
- 应用程序偶尔会耗尽连接对象；

可能导致内存泄漏的原因：

1. static字段引起的内存泄漏

大量使用static字段会潜在的导致内存泄漏，在Java中，静态字段通常拥有与整个应用程序相匹配的生命周期。

解决办法：最大限度的减少静态变量的使用；单例模式时，依赖于延迟加载对象而不是立即加载的方式（即采用懒汉模式，而不是饿汉模式）

2. 未关闭的资源导致内存泄漏

每当创建连接或者打开流时，JVM都会为这些资源分配内存。如果没有关闭连接，会导致持续占有内存。在任意情况下，资源留下的开放连接都会消耗内存，如果不处理，就会降低性能，甚至OOM。

解决办法：使用finally块关闭资源；关闭资源的代码，不应该有异常；JDK1.7之后，可以使用太try-with-resource块。

3. 不正确的equals()和hashCode()

在HashMap和HashSet这种集合中，常常用到equal()和hashCode()来比较对象，如果重写不合理，将会成为潜在的内存泄漏问题。

解决办法：用最佳的方式重写equals()和hashCode().

4. 引用了外部类的内部类

非静态内部类的初始化，总是需要外部类的实例；默认情况下，每个非静态内部类都包含对其外部类的隐式引用，如果我们在应用程序中使用这个内部类对象，那么即使在我们的外部类对象超出范围后，它也不会被垃圾收集器清除掉。

解决办法：如果内部类不需要访问外部类包含的类成员，可以转换为静态类。

5. finalize方法导致的内存泄漏

重写finalize()方法时，该类的对象不会立即被垃圾收集器收集，如果finalize()方法的代码有问题，那么会潜在的印发OOM；

解决办法：避免重写finalize()方法。

6. 常量字符串造成的内存泄漏

如果我们读取一个很大的String对象，并调用了intern()，那么它将放到字符串池中，位于PermGen中，只要应用程序运行，该字符串就会保留，这就会占用内存，可能造成OOM。（针对JDK1.6及以前，常量池在PermGen永久代中）

解决办法：增加PermGen的大小，-XX:MaxPermSize=512M；JDK1.7以后字符串池转移到了堆中。

intern()方法详解：

```
String str1 = "abc";
String str2 = "abc";
String str3 = new String("abc");
String str4 = str3.intern();

System.out.println(str1 == str2);
System.out.println(str2 == str3);

System.out.println(str1 == str4);
System.out.println(str3 == str4);

true, false, true, false
```

intern()方法搜索字符串常量池，如果存在指定的字符串，就返回之；

否则，就将该字符串放入常量池并返回之。

换言之，intern()方法保证每次返回的都是 “**同一个字符串对象**”。

```
String str1 = "abc";
String str2 = "abc";
String str3 = new String("abcd");
String str4 = str3.intern();
String str5 = "abcd";


System.out.println(str1 == str2);
System.out.println(str2 == str3);

System.out.println(str1 == str4);
System.out.println(str3 == str4);

System.out.println(str4 == str5);

true
false
false
false
true
```

为何要使用intern()方法？看看equals方法的源码：

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
```

```
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

可以看到，比较两个字符串的时候，首先比较两个字符串对象是否地址相同，不同再挨个比较字符。这样就大大加快了比较的速度。否则若每次都挨个比较将是非常耗时的。

7. 使用ThreadLocal造成内存泄漏

使用ThreadLocal时，每个线程只要处于存活状态就可保留对其ThreadLocal变量副本的隐式调用，且将保留其自己的副本。使用不当，就会引起内存泄漏。

一旦线程不再存在，该线程的threadLocal对象就应该被垃圾收集，而现在线程的创建都是使用线程池，线程池有线程重用的功能，因此线程就不会被垃圾回收器回收。所以使用到ThreadLocal来保留线程池中的线程的变量副本时，ThreadLocal没有显式地删除时，就会一直保留在内存中，不会被垃圾回收。

解决办法：不再使用ThreadLocal时，调用remove()方法，该方法删除了此变量的当前线程值。不要使用ThreadLocal.set(null)，它只是查找与当前线程关联的Map并将键值中这个threadLocal对象所对应的值为null，并没有清除这个键值对。