> 笔记来源：尚硅谷 JVM 全套教程，百万播放，全网巅峰（宋红康详解 java 虚拟机）
>
> 同步更新：https://gitee.com/vectorx/NOTE_JVM
>
> https://codechina.csdn.net/qq_35925558/NOTE_JVM
>
> https://github.com/uxiahnan/NOTE_JVM

[toc]

补充：浅堆深堆与内存泄露

1. 浅堆（Shallow Heap）

浅堆是指一个对象所消耗的内存。在 32 位系统中，一个对象引用会占据 4 个字节，一个 int 类型会占据 4 个字节，long 型变量会占据 8 个字节，每个对象头需要占用 8 个字节。根据堆快照格式不同，对象的大小可能会同 8 字节进行对齐。

以 String 为例：2 个 int 值共占 8 字节，对象引用占用 4 字节，对象头 8 字节，合计 20 字节，向 8 字节对齐，故占 24 字节。 (jdk7 中)

int	**hash32**	**0**
int	**hash**	**0**
ref	**value**	**C:\Users\Administrat**

这 24 字节为 String 对象的浅堆大小。它与 String 的 value 实际取值无关，无论字符串长度如何，浅堆大小始终是 24 字节。

2. 保留集（Retained Set）

对象 A 的保留集指当对象 A 被垃圾回收后，可以被释放的所有的对象集合（包括对象 A 本身），即对象 A 的保留集可以被认为是只能通过对象 A 被直接或间接访问到的所有对象的集合。通俗地说，就是指仅被对象 A 所持有的对象的集合。

3. 深堆（Retained Heap）

深堆是指对象的保留集中所有的对象的浅堆大小之和。

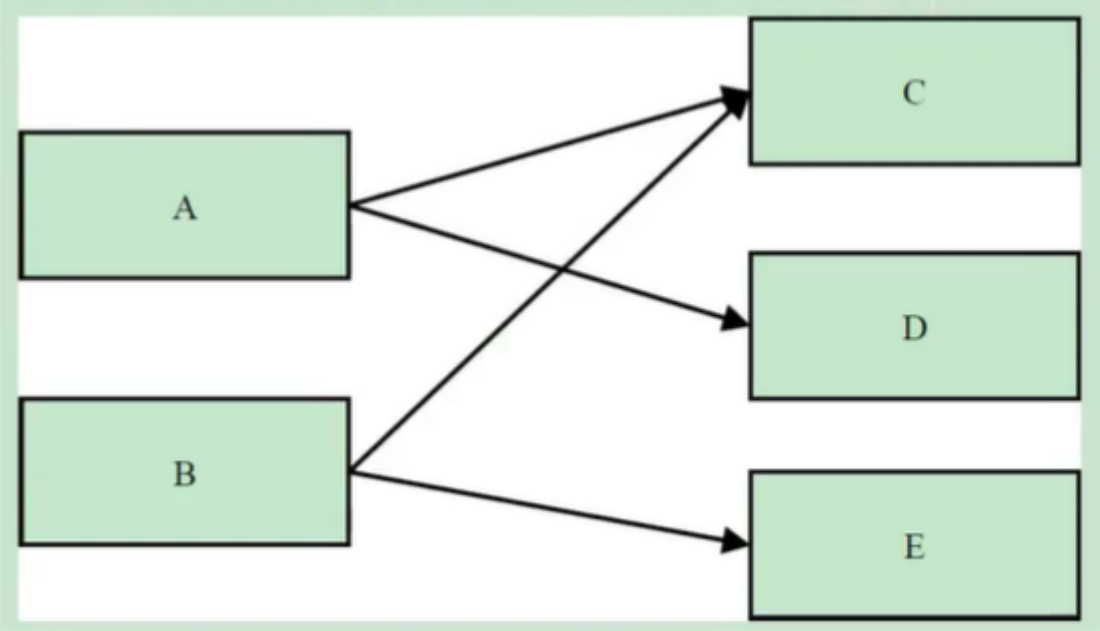
注意：浅堆指对象本身占用的内存，不包括其内部引用对象的大小。一个对象的深堆指只能通过该对象访问到的（直接或间接）所有对象的浅堆之和，即对象被回收后，可以释放的真实空间。

4. 对象的实际大小

这里，对象的实际大小定义为一个对象所能触及的所有对象的浅堆大小之和，也就是通常意义上我们说的对象大小。与深堆相比，似乎这个在日常开发中更为直观和被人接受，但实际上，这个概念和垃圾回收无关。

下图显示了一个简单的对象引用关系图，对象 A 引用了 C 和 D，对象 B 引用了 C 和 E。那么对象 A 的浅堆大小只是 A 本身，不含 C 和 D，而 A 的实际大小为 A、C、D 三者之和。而 A 的深堆大小为 A 与 D 之和，由于

对象 C 还可以通过对象 B 访问到，因此不在对象 A 的深堆范围内。

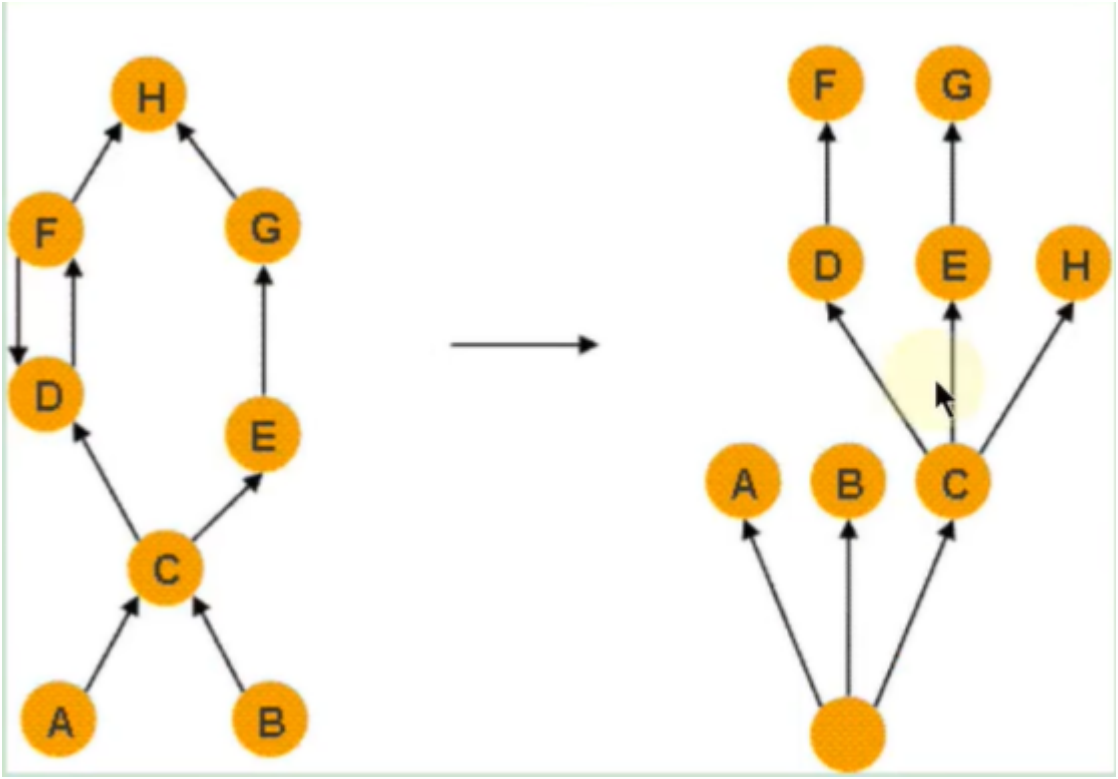


5. 支配树（Dominator Tree）

支配树的概念源自图论。MAT 提供了一个称为支配树（Dominator Tree）的对象图。支配树体现了对象实例间的支配关系。在对象引用图中，所有指向对象 B 的路径都经过对象 A，则认为对象 A 支配对象 B。如果对象 A 是离对象 B 最近的一个支配对象，则认为对象 A 为对象 B 的直接支配者。支配树是基于对象间的引用图所建立的，它有以下基本性质：

- 对象 A 的子树（所有被对象 A 支配的对象集合）表示对象 A 的保留集（retained set），即深堆。
- 如果对象 A 支配对象 B，那么对象 A 的直接支配者也支配对象 B。
- 支配树的边与对象引用图的边不直接对应。

如下图所示：左图表示对象引用图，右图表示左图所对应的支配树。对象 A 和 B 由根对象直接支配，由于在到对象 C 的路径中，可以经过 A，也可以经过 B，因此对象 C 的直接支配者也是根对象。对象 F 与对象 D 相互引用，因为到对象 F 的所有路径必然经过对象 D，因此，对象 D 是对象 F 的直接支配者。而到对象 D 的所有路径中，必然经过对象 C，即使是从对象 F 到对象 D 的引用，从根节点出发，也是经过对象 C 的，所以，对象 D 的直接支配者为对象 C。同理，对象 E 支配对象 G。到达对象 H 的可以通过对象 D，也可以通过对象 E，因此对象 D 和 E 都不能支配对象 H，而经过对象 C 既可以到达 D 也可以到达 E，因此对象 C 为对象 H 的直接支配者。

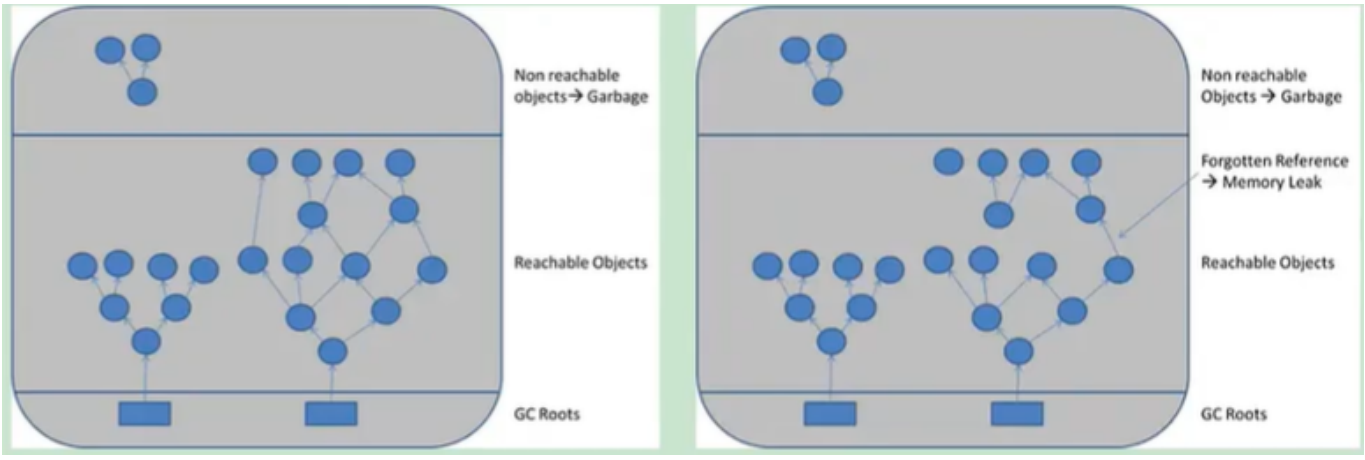


6. 内存泄漏（memory leak）

可达性分析算法来判断对象是否是不再使用的对象，本质都是判断一个对象是否还被引用。那么对于这种情况下，由于代码的实现不同就会出现很多种内存泄漏问题（让 JVM 误以为此对象还在引用中，无法回收，造成内存泄漏）。

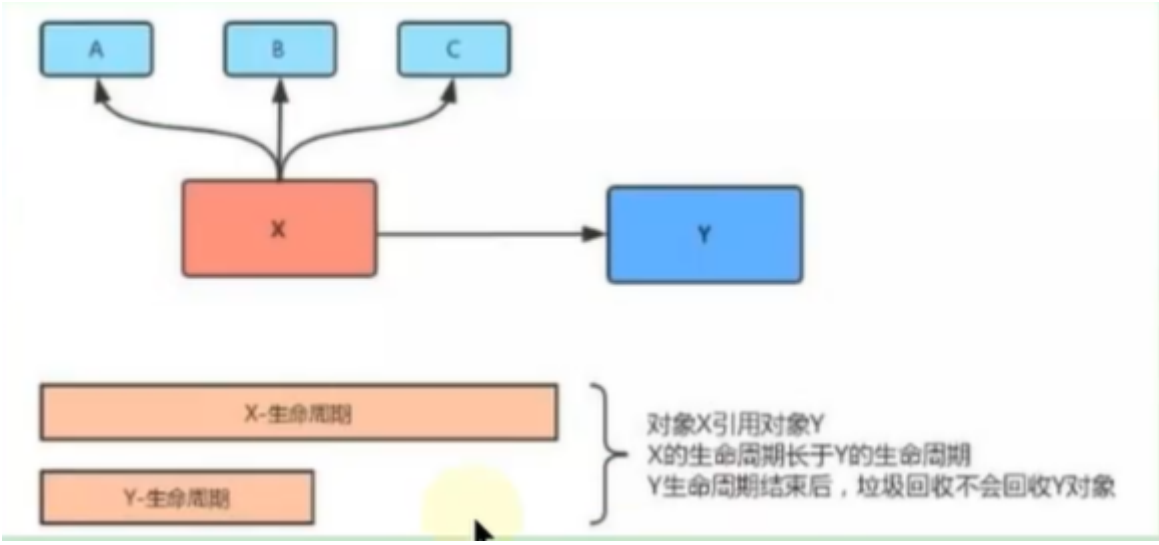
> 是否还被使用？是

> 是否还被需要？否



严格来说，只有对象不会再被程序用到了，但是 GC 又不能回收他们的情况，才叫内存泄漏。但实际情况很多时候一些不太好的实践（或疏忽）会导致对象的生命周期变得很长甚至导致 00M，也可以叫做宽泛意义上的“内存泄漏”。

如下图，当 Y 生命周期结束的时候，X 依然引用着 Y，这时候，垃圾回收期是不会回收对象 Y 的；如果对象 X 还引用着生命周期比较短的 A、B、C，对象 A 又引用着对象 a、b、c，这样就可能造成大量无用的对象不能被回收，进而占据了内存资源，造成内存泄漏，直到内存溢出。



申请了内存用完了不释放，比如一共有 1024M 的内存，分配了 512M 的内存一直不回收，那么可以用的内存只有 512M 了，仿佛泄露掉了一部分；通俗一点讲的话，内存泄漏就是【占着茅坑不拉 shi】

7. 内存溢出（out of memory）

申请内存时，没有足够的内存可以使用；通俗一点儿讲，一个厕所就三个坑，有两个站着茅坑不走的（内存泄漏），剩下最后一个坑，厕所表示接待压力很大，这时候一下子来了两个人，坑位（内存）就不够了，内存泄漏变成内存溢出了。可见，内存泄漏和内存溢出的关系：内存泄漏的增多，最终会导致内存溢出。

泄漏的分类

- 经常发生：发生内存泄露的代码会被多次执行，每次执行，泄露一块内存；
- 偶然发生：在某些特定情况下才会发生
- 一次性：发生内存泄露的方法只会执行一次；
- 隐式泄漏：一直占着内存不释放，直到执行结束；严格的说这个不算内存泄漏，因为最终释放掉了，但是如果执行时间特别长，也可能会导致内存耗尽。

8. Java 中内存泄露的 8 种情况

8.1. 静态集合类

静态集合类，如 HashMap、LinkedList 等等。如果这些容器为静态的，那么它们的生命周期与 JVM 程序一致，则容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有短生命周期对象的引用，尽管短生命周期的对象不再使用，但是因为长生命周期对象持有它的引用而导致不能被回收。

```
public class MemoryLeak {
    static List list = new ArrayList();
    public void oomTests(){
        Object obj = new Object();//局部变量
        list.add(obj);
    }
}
```

8.2. 单例模式

单例模式，和静态集合导致内存泄露的原因类似，因为单例的静态特性，它的生命周期和 JVM 的生命周期一样长，所以如果单例对象如果持有外部对象的引用，那么这个外部对象也不会被回收，那么就会造成内存泄漏。

8.3. 内部类持有外部类

内部类持有外部类，如果一个外部类的实例对象的方法返回了一个内部类的实例对象。这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄漏。

8.4. 各种连接，如数据库连接、网络连接和 IO 连接等

在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用 close 方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对 Connection、Statement 或 ResultSet 不显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。

```
public static void main(String[] args) {
    try{
        Connection conn =null;
        Class.forName("com.mysql.jdbc.Driver");
        conn =DriverManager.getConnection("url","","");
        Statement stmt =conn.createStatement();
        ResultSet rs =stmt.executeQuery("....");
    } catch (Exception e) {//异常日志
    } finally {
        // 1. 关闭结果集 Statement
        // 2. 关闭声明的对象 ResultSet
        // 3. 关闭连接 Connection
    }
}
```

8.5. 变量不合理的作用域

变量不合理的作用域。一般而言，一个变量的定义的作用范围大于其使用范围，很有可能会造成内存泄漏。另一方面，如果没有及时地把对象设置为 null，很有可能导致内存泄漏的发生。

```
public class UsingRandom {
    private String msg;
    public void receiveMsg(){
        readFromNet();//从网络中接受数据保存到msg中
        saveDB();//把msg保存到数据库中
    }
}
```

如上面这个伪代码，通过 readFromNet 方法把接受的消息保存在变量 msg 中，然后调用 saveDB 方法把 msg 的内容保存到数据库中，此时 msg 已经就没用了，由于 msg 的生命周期与对象的生命周期相同，此时 msg 还不能回收，因此造成了内存泄漏。实际上这个 msg 变量可以放在 receiveMsg 方法内部，当方法使用完，那么

msg 的生命周期也就结束，此时就可以回收了。还有一种方法，在使用完 msg 后，把 msg 设置为 null，这样垃圾回收器也会回收 msg 的内存空间。

8.6. 改变哈希值

改变哈希值，当一个对象被存储进 HashSet 集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了。

否则，对象修改后的哈希值与最初存储进 HashSet 集合中时的哈希值就不同了，在这种情况下，即使在 contains 方法使用该对象的当前引用作为的参数去 HashSet 集合中检索对象，也将返回找不到对象的结果，这也会导致无法从 HashSet 集合中单独删除当前对象，造成内存泄漏。

这也是 String 为什么被设置成了不可变类型，我们可以放心地把 String 存入 HashSet，或者把 String 当做 HashMap 的 key 值；

当我们想把自己定义的类保存到散列表的时候，需要保证对象的 hashCode 不可变。

```
/**
 * 例1
 */
public class ChangeHashCode {
    public static void main(String[] args) {
        HashSet set = new HashSet();
        Person p1 = new Person(1001, "AA");
        Person p2 = new Person(1002, "BB");

        set.add(p1);
        set.add(p2);

        p1.name = "CC";//导致了内存的泄漏
        set.remove(p1); //删除失败

        System.out.println(set);

        set.add(new Person(1001, "CC"));
        System.out.println(set);

        set.add(new Person(1001, "AA"));
        System.out.println(set);

    }
}

class Person {
    int id;
    String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;

        Person person = (Person) o;

        if (id != person.id) return false;
        return name != null ? name.equals(person.name) : person.name == null;
    }

    @Override
    public int hashCode() {
        int result = id;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

```
/**
 * 例2
 */
public class ChangeHashCode1 {
    public static void main(String[] args) {
        HashSet<Point> hs = new HashSet<Point>();
        Point cc = new Point();
        cc.setX(10);//hashCode = 41
        hs.add(cc);

        cc.setX(20);//hashCode = 51  此行为导致了内存的泄漏

        System.out.println("hs.remove = " + hs.remove(cc));//false
        hs.add(cc);
        System.out.println("hs.size = " + hs.size());//size = 2

        System.out.println(hs);
    }

}

class Point {
    int x;
```

```
    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        if (x != other.x) return false;
        return true;
    }

    @Override
    public String toString() {
        return "Point{" +
                "x=" + x +
                '}';
    }
}
```

8.7. 缓存泄露

内存泄漏的另一个常见来源是缓存，一旦你把对象引用放入到缓存中，他就很容易遗忘。比如：之前项目在一次上线的时候，应用启动奇慢直到夯死，就是因为代码中会加载一个表中的数据到缓存（内存）中，测试环境只有几百条数据，但是生产环境有几百万的数据。

对于这个问题，可以使用 WeakHashMap 代表缓存，此种 Map 的特点是，当除了自身有对 key 的引用外，此 key 没有其他引用那么此 map 会自动丢弃此值。

```
public class MapTest {
    static Map wMap = new WeakHashMap();
    static Map map = new HashMap();

    public static void main(String[] args) {
        init();
        testWeakHashMap();
```

```
        testHashMap();
    }

    public static void init() {
        String ref1 = new String("obejct1");
        String ref2 = new String("obejct2");
        String ref3 = new String("obejct3");
        String ref4 = new String("obejct4");
        wMap.put(ref1, "cacheObject1");
        wMap.put(ref2, "cacheObject2");
        map.put(ref3, "cacheObject3");
        map.put(ref4, "cacheObject4");
        System.out.println("String引用ref1, ref2, ref3, ref4 消失");

    }

    public static void testWeakHashMap() {
        System.out.println("WeakHashMap GC之前");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("WeakHashMap GC之后");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
    }

    public static void testHashMap() {
        System.out.println("HashMap GC之前");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("HashMap GC之后");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
    }

}
```

上面代码和图示主演演示 WeakHashMap 如何自动释放缓存对象，当 init 函数执行完成后，局部变量字符串引用 weakd1，weakd2，d1，d2 都会消失，此时只有静态 map 中保存中对字符串对象的引用，可以看到，调用 gc 之后，HashMap 的没有被回收，而 WeakHashMap 里面的缓存被回收了。

8.8. 监听器和其他回调

内存泄漏第三个常见来源是监听器和其他回调，如果客户端在你实现的 API 中注册回调，却没有显示的取消，那么就会积聚。

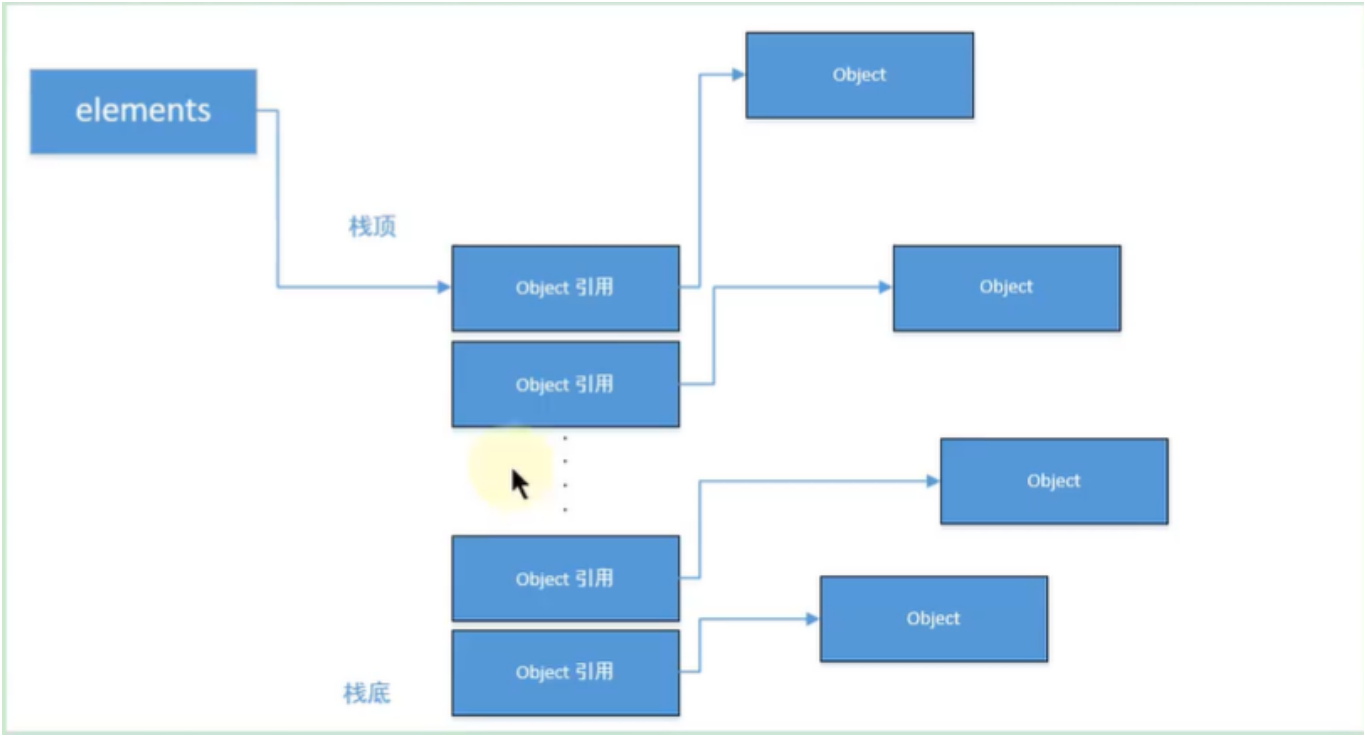
需要确保回调立即被当作垃圾回收的最佳方法是只保存它的弱引用，例如将他们保存成为 WeakHashMap 中的键。

9. 内存泄露案例分析

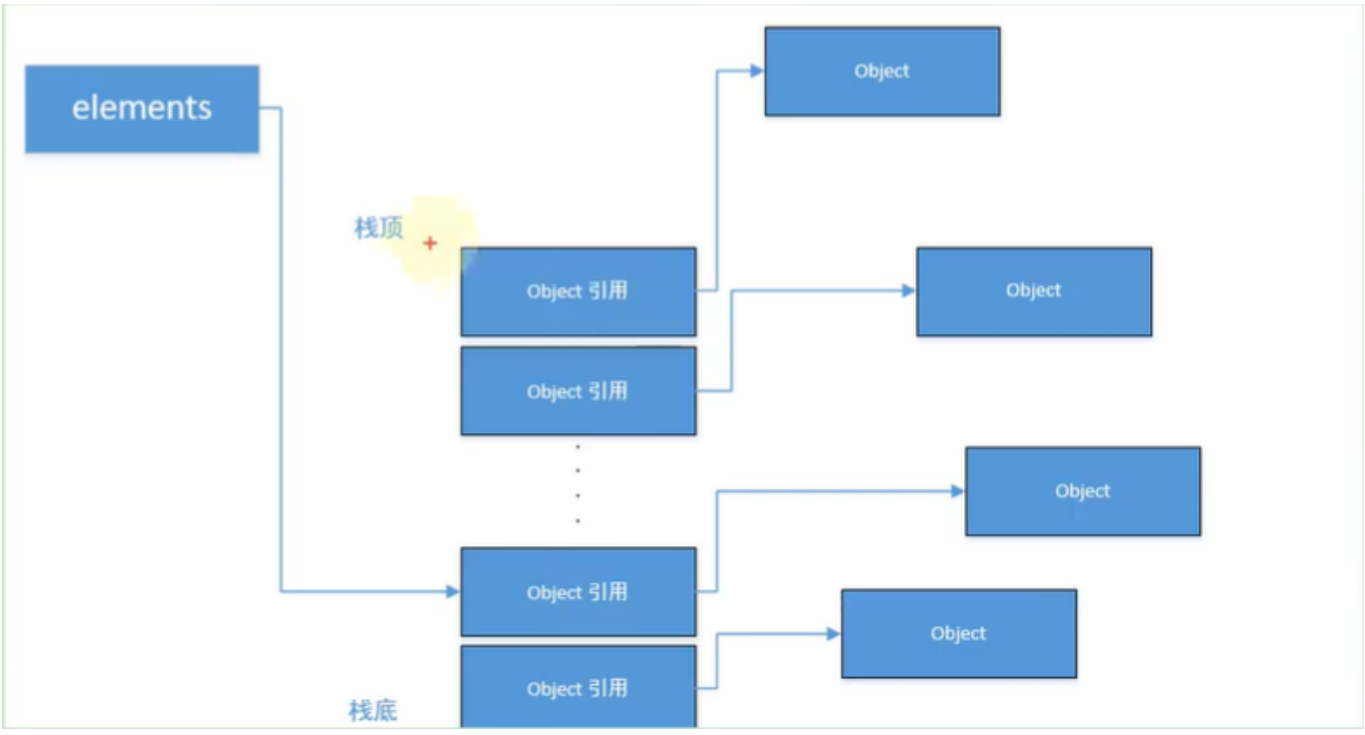
```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) { //入栈
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() { //出栈
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

上述程序并没有明显的错误，但是这段程序有一个内存泄漏，随着 GC 活动的增加，或者内存占用的不断增加，程序性能的降低就会表现出来，严重时可导致内存泄漏，但是这种失败情况相对较少。

代码的主要问题在 pop 函数，下面通过这张图示展现。假设这个栈一直增长，增长后如下图所示



当进行大量的 pop 操作时，由于引用未进行置空，gc 是不会释放的，如下图所示



从上图中看以看出，如果栈先增长，再收缩，那么从栈中弹出的对象将不会被当作垃圾回收，即使程序不再使用栈中的这些队象，他们也不会回收，因为栈中仍然保存这对象的引用，俗称过期引用，这个内存泄露很隐蔽。

将代码中的 pop()方法变成如下方法：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
```

```
        elements[size] = null;
        return result;
    }
```

一旦引用过期，清空这些引用，将引用置空。