JAVA 内存管理总结

1. java 是如何管理内存的

Java 的内存管理就是对象的分配和释放问题。（两部分）

分配 ：内存的分配是由程序完成的，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。
释放 ：对象的释放是由垃圾回收机制决定和执行的，这样做确实简化了程序员的工作。但同时，它也加重了 JVM 的工作。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。

2. 什么叫 java 的内存泄露

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（也就是说仍存在该内存对象的引用）；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。

3. JVM 的内存区域组成

java 把内存分两种：一种是栈内存，另一种是堆内存 1。在函数中定义的基本类型变量和对象的引用变量都在函数的栈内存中分配；2。堆内存用来存放由 new 创建的对象和数组以及对象的实例变量 在函数（代码块）中定义一个变量时，java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，java 会自动释放掉为该变量所分配的内存空间；在堆中分配的内存由 java 虚拟机的自动垃圾回收器来管理
堆和栈的优缺点

堆的优势是可以动态分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的。

缺点就是要在运行时动态分配内存，存取速度较慢； 栈的优势是，存取速度比堆要快，仅次于直接位于 CPU 中的寄存器。

另外，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。

4. Java 中数据在内存中是如何存储的

a) 基本数据类型

Java 的基本数据类型共有 8 种，即 **int, short, long, byte, float, double, boolean, char**(注意，并没有 string 的基本类型)。这种类型的定义是通过诸如 int a = 3； long b = 255L；的形式来定义的。如 int a = 3；这里的 a 是一个指向 int 类型的引用，指向 3 这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值定义在某个程序块里面，程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。

另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。比如：我们同时定义：

```
int a=3;
int b =3;
```

编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为 3 的地址，没找到，就开辟一个存放 3 这个字面值的地址，然后将 a 指向 3 的地址。接着处理 int b = 3；在创建完 b 这个引用变量后，由于在栈中已经有 3 这个字面值，便将 b 直接指向 3 的地址。这样，就出现了 a 与 b 同时均指向 3 的情况。 定义完 a 与 b 的值后，再令 a = 4；那么，b 不会等于 4，还是等于 3。在编译器内部，遇到时，它就会重新搜索栈中是否有 4 的字面值，如果没有，重新开辟地址存放 4 的值；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响到 b 的值。

b) 对象

在 Java 中，创建一个对象包括对象的声明和实例化两步，下面用一个例题来说明对象的内存模型。 假设有类 Rectangle 定义如下：

```
public class Rectangle {
double width;
double height;
public Rectangle(double w,double h){
w = width;
h = height;
}
}
```

(1)声明对象时的内存模型

用 Rectangle rect；声明一个对象 rect 时，将在栈内存为对象的引用变量 rect 分配内存空间，但 Rectangle 的值为空，称 rect 是一个空对象。空对象不能使用，因为它还没有引用任何"实体"。

(2)对象实例化时的内存模型

当执行 rect=new Rectangle(3,5)；时，会做两件事： 在堆内存中为类的成员变量 width,height 分配内存，并将其初始化为各数据类型的默认值；接着进行显式初始化（类定义时的初始化值）；最后调用构造方法，为成员变量赋值。 返回堆内存中对象的引用（相当于首地址）给引用变量 rect,以后就可以通过 rect 来引用堆内存中的对象了。

c) 创建多个不同的对象实例

一个类通过使用 new 运算符可以创建多个不同的对象实例，这些对象实例将在堆中被分配不同的内存空间，改变其中一个对象的状态不会影响其他对象的状态。例如：

```
Rectangle r1= new Rectangle(3,5);
Rectangle r2= new Rectangle(4,6);
```

此时，将在堆内存中分别为两个对象的成员变量 width、height 分配内存空间，两个对象在堆内存中占据的空间是互不相同的。如果有：

```
Rectangle r1= new Rectangle(3,5);
Rectangle r2=r1;
```

则在堆内存中只创建了一个对象实例，在栈内存中创建了两个对象引用，两个对象引用同时指向一个对象实例。

d) 包装类

基本型别都有对应的包装类：如 int 对应 Integer 类，double 对应 Double 类等，基本类型的定义都是直接在栈中，如果用包装类来创建对象，就和普通对象一样了。例如：int i=0；i 直接存储在栈中。 Integer i（i 此时是对象） = new Integer(5)；这样，i 对象数据存储在堆中，i 的引用存储在栈中，通过栈中的引用来操作对象。

e) String

String 是一个特殊的包装类数据。可以用用以下两种方式创建：String str = new String("abc")；String str = "abc";

第一种创建方式，和普通对象的的创建过程一样；

第二种创建方式，Java 内部将此语句转化为以下几个步骤：

(1) 先定义一个名为 str 的对 String 类的对象引用变量：String str；

(2) 在栈中查找有没有存放值为"abc"的地址，如果没有，则开辟一个存放字面值为"abc"地址，接着创建一个新的 String 类的对象 o，并将 o 的字符串值指向这个地址，而且在栈这个地址旁边记下这个引用的对象 o。如果已经有了值为"abc"的地址，则查找对象 o，并回 o 的地址。

(3) 将 str 指向对象 o 的地址。

值得注意的是，一般 String 类中字符串值都是直接存值的。但像 String str = "abc"；这种合下，其字符串值却是保存了一个指向存在栈中数据的引用。

为了更好地说明这个问题，我们可以通过以下的几个代码进行验证。

```
String str1="abc"；
String str2="abc"；
System.out.println(s1==s2)；//true
```

注意，这里并不用 str1.equals(str2)；的方式，因为这将比较两个字符串的值是否相等。==号，根据 JDK 的说明，只有在两个引用都指向了同一个对象时才返回真值。而我们在这里要看的是，str1 与 str2 是否都指向了同一个对象。

我们再接着看以下的代码。

```
String str1= new String("abc")；
String str2="abc"；
System.out.println(str1==str2)；//false
```

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。 以上两段代码

说明，只要是用 new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

f) 数组

当定义一个数组，int x[]；或 int []x；时，在栈内存中创建一个数组引用，通过该引用（即数组名）来引用数组。x=new int[3]；将在堆内存中分配 3 个保存 int 型数据的空间，堆内存的首地址放到栈内存中，每个数组元素被初始化为 0。

g) 静态变量

用 static 的修饰的变量和方法，实际上是指定了这些变量和方法在内存中的"固定位置"—static storage，可以理解为所有实例对象共有的内存空间。static 变量有点类似于 C 中的全局变量的概念；静态表示的是内存的共享，就是它的每一个实例都指向同一个内存地址。把 static 拿来，就是告诉 JVM 它是静态的，它的引用（含间接引用）都是指向同一个位置，在那个地方，你把它改了，它就不会变成原样，你把它清理了，它就不会回来了。　　那静态变量与方法是在什么时候初始化的呢？对于两种不同的类属性，static 属性与 instance 属性，初始化的时机是不同的。instance 属性在创建实例的时候初始化，static 属性在类加载，也就是第一次用到这个类的时候初始化，对于后来的实例的创建，不再次进行初始化。　　我们常可看到类似以下的例子来说明这个问题：

```
class Student{
static int numberOfStudents=0;
Student()
{
numberOfStudents++;
}
}
```

每一次创建一个新的 Student 实例时,成员 *numberOfStudents* 都会不断的递增,并且所有的 Student 实例都访问同一个 *numberOfStudents* 变量,实际上 int *numberOfStudents* 变量在内存中只存储在一个位置上。

5. Java 的内存管理实例

Java 程序的多个部分(方法，变量，对象)驻留在内存中以下两个位置：即堆和栈，现在我们只关心 3 类事物：实例变量，局部变量和对象：

实例变量和对象驻留在堆上

局部变量驻留在栈上

让我们查看一个 java 程序，看看他的各部分如何创建并且映射到栈和堆中：

```
public class Dog {
Collar c;
String name;
```

```
//1. main()方法位于栈上
public static void main(String[] args) {
//2. 在栈上创建引用变量 d,但 Dog 对象尚未存在
Dog d;
//3. 创建新的 Dog 对象，并将其赋予 d 引用变量
d = new Dog();
//4. 将引用变量的一个副本传递给 go()方法
d.go(d);
}
//5. 将 go()方法置于栈上，并将 dog 参数作为局部变量
void go(Dog dog){
//6. 在堆上创建新的 Collar 对象，并将其赋予 Dog 的实例变量
c =new Collar();
}
//7.将 setName()添加到栈上，并将 dogName 参数作为其局部变量
void setName(String dogName){
//8. name 的实例对象也引用 String 对象
name=dogName;
}
//9. 程序执行完成后，setName()将会完成并从栈中清除，此时，局部变量 dogName 也会消失，
尽管它所引用的 String 仍在堆上
}
```

6. 垃圾回收机制：

（问题一：什么叫垃圾回收机制？） 垃圾回收是一种动态存储管理技术，它自动地释放不再被程序引用的对象，按照特定的垃圾收集算法来实现资源自动回收的功能。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用，以免造成内存泄露。 **（问题二：java 的垃圾回收有什么特点？）** JAVA 语言不允许程序员直接控制内存空间的使用。内存空间的分配和回收都是由 JRE 负责在后台自动进行的，尤其是无用内存空间的回收操作(garbagecollection,也称垃圾回收)，只能由运行环境提供的一个超级线程进行监测和控制。 **（问题三：垃圾回收器什么时候会运行？）** 一般是在 CPU 空闲或空间不足时自动进行垃圾回收，而程序员无法精确控制垃圾回收的时机和顺序等。 **（问题四：什么样的对象符合垃圾回收条件？）** 当没有任何获得线程能访问一个对象时，该对象就符合垃圾回收条件。 **（问题五：垃圾回收器是怎样工作的？）** 垃圾回收器如发现一个对象不能被任何活线程访问时，他将认为该对象符合删除条件，就将其加入回收队列，但不是立即销毁对象，何时销毁并释放内存是无法预知的。垃圾回收不能强制执行，然而 Java 提供了一些方法（如：System.gc()方法），允许你请求 JVM 执行垃圾回收，而不是要求，虚拟机会尽其所能满足请求，但是不能保证 JVM 从内存中删除所有不用的对象。 **（问题六：一个 java 程序能够耗尽内存吗？）** 可以。垃圾收集系统尝试在对象不被使用时把他们从内存中删除。然而，如果保持太多活的对象，系统则可能会耗尽内存。垃圾回收器不能保证有足够的内存，只能保证可用内存尽可能的得到高效的管理。 **（问题七：如何显示的使对象符合垃圾回收条件？）** （1） 空引用 ：当对象没有对他可到达引用时，他就符合垃圾回收的条件。也就是说如果没有对他的引用，删除对象的引用就可以达到目的，因此我们可以把引用变量设置为 null，来符合垃圾回收的条件。

```
StringBuffer sb = new StringBuffer("hello");
System.out.println(sb);
sb=null;
```

（2） 重新为引用变量赋值：可以通过设置引用变量引用另一个对象来解除该引用变量与一个对象间的引用关系。

```
StringBuffer sb1 = new StringBuffer("hello");
StringBuffer sb2 = new StringBuffer("goodbye");
System.out.println(sb1);
sb1=sb2;//此时"hello"符合回收条件
```

（3） 方法内创建的对象：所创建的局部变量仅在该方法的作用期间内存在。一旦该方法返回，在这个方法内创建的对象就符合垃圾收集条件。有一种明显的例外情况，就是方法的返回对象。

```
public static void main(String[] args) {
Date d = getDate();
System.out.println("d = " + d);
}
private static Date getDate() {
Date d2 = new Date();
StringBuffer now = new StringBuffer(d2.toString());
System.out.println(now);
return d2;
}
```

（4） 隔离引用：这种情况中，被回收的对象仍具有引用，这种情况称作隔离岛。若存在这两个实例，他们互相引用，并且这两个对象的所有其他引用都删除，其他任何线程无法访问这两个对象中的任意一个。也可以符合垃圾回收条件。

```
public class Island {
Island i;
public static void main(String[] args) {
Island i2 = new Island();
Island i3 = new Island();
Island i4 = new Island();
i2.i=i3;
i3.i=i4;
i4.i=i2;
i2=null;
i3=null;
i4=null;
}
}
```

（问题八：垃圾收集前进行清理------finalize()方法） java 提供了一种机制，使你能够在对象刚要被垃圾回收之前运行一些代码。这段代码位于名为 finalize()的方法内，所有类从 Object 类继承这个方法。由于不能保证垃圾回收器会删除某个对象。因此放在 finalize()中的代码无法保证运行。因此建议不要重写 finalize();

7. final 问题：

final 使得被修饰的变量"不变"，但是由于对象型变量的本质是"引用"，使得"不变"也有了两种

含义：引用本身的不变?，和引用指向的对象不变。?　　引用本身的不变：

```
final StringBuffer a=new StringBuffer("immutable");
final StringBuffer b=new StringBuffer("not immutable");
a=b;//编译期错误
final StringBuffer a=new StringBuffer("immutable");
final StringBuffer b=new StringBuffer("not immutable");
a=b;//编译期错误
```

引用指向的对象不变：

```
final StringBuffer a=new StringBuffer("immutable");
a.append(" broken!"); //编译通过
final StringBuffer a=new StringBuffer("immutable");
a.append(" broken!"); //编译通过
```

可见，final 只对引用的"值"(也即它所指向的那个对象的内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。至于它所指向的对象的变化，final 是不负责的。这很类似==操作符：==操作符只负责引用的"值"相等，至于这个地址所指向的对象内容是否相等，==操作符是不管的。在举一个例子：

```
public class Name {
private String firstname;
private String lastname;
public String getFirstname() {
return firstname;
}
public void setFirstname(String firstname) {
this.firstname = firstname;
}
public String getLastname() {
return lastname;
}
public void setLastname(String lastname) {
this.lastname = lastname;
}
}

public class Name {
private String firstname;
private String lastname;
public String getFirstname() {
return firstname;
}
public void setFirstname(String firstname) {
this.firstname = firstname;
}
public String getLastname() {
return lastname;
```

```
}
public void setLastname(String lastname) {
this.lastname = lastname;
}
}
```

编写测试方法：

```
public static void main(String[] args) {
final Name name = new Name();
name.setFirstname("JIM");
name.setLastname("Green");
System.out.println(name.getFirstname()+" "+name.getLastname());
}
public static void main(String[] args) {
final Name name = new Name();
name.setFirstname("JIM");
name.setLastname("Green");
System.out.println(name.getFirstname()+" "+name.getLastname());
}
```

理解 final 问题有很重要的含义。许多程序漏洞都基于此----final 只能保证引用永远指向固定对象，不能保证那个对象的状态不变。在多线程的操作中,一个对象会被多个线程共享或修改，一个线程对对象无意识的修改可能会导致另一个使用此对象的线程崩溃。一个错误的解决方法就是在此对象新建的时候把它声明为 final，意图使得它"永远不变"。其实那是徒劳的。 Final 还有一个值得注意的地方： 先看以下示例程序：

```
class Something {
final int i;
public void doSomething() {
System.out.println("i = " + i);
}
}
class Something {
final int i;
public void doSomething() {
System.out.println("i = " + i);
}
}
```

对于类变量，Java 虚拟机会自动进行初始化。如果给出了初始值，则初始化为该初始值。如果没有给出，则把它初始化为该类型变量的默认初始值。但是对于用 final 修饰的类变量，虚拟机不会为其赋予初值，必须在 constructor (构造器)结束之前被赋予一个明确的值。可以修改为"final int i = 0;"。

8. 如何把程序写得更健壮：

1、尽早释放无用对象的引用。 好的办法是使用临时变量的时候，让引用变量在退出活动域

后，自动设置为 null，暗示垃圾收集器来收集该对象，防止发生内存泄露。对于仍然有指针指向的实例，jvm 就不会回收该资源,因为垃圾回收会将值为 null 的对象作为垃圾，提高 GC 回收机制效率；

2、**定义字符串应该尽量使用 String str="hello"; 的形式** ，避免使用 String str = new String("hello"); 的形式。因为要使用内容相同的字符串，不必每次都 new 一个 String。例如我们要在构造器中对一个名叫 s 的 String 引用变量进行初始化，把它设置为初始值，应当这样做：

```
public class Demo {
private String s;
public Demo() {
s = "Initial Value";
}
}
```

```
public class Demo {
private String s;
...
public Demo {
s = "Initial Value";
}
...
}
```

而非

```
s = new String("Initial Value");
s = new String("Initial Value");
```

后者每次都会调用构造器，生成新对象，性能低下且内存开销大，并且没有意义，因为 String 对象不可改变，所以对于内容相同的字符串，只要一个 String 对象来表示就可以了。也就说，多次调用上面的构造器创建多个对象，他们的 String 类型属性 s 都指向同一个对象。

3、我们的程序里不可避免大量使用字符串处理，避免使用 String，应**大量使用 StringBuffer** ，因为 String 被设计成不可变(immutable)类，所以它的所有对象都是不可变对象，请看下列代码：

```
String s = "Hello";
s = s + " world!";
String s = "Hello";
s = s + " world!";
```

在这段代码中，s 原先指向一个 String 对象，内容是 "Hello"，然后我们对 s 进行了+操作，那么 s 所指向的那个对象是否发生了改变呢？答案是没有。这时，s 不指向原来那个对象了，而指向了另一个 String 对象，内容为"Hello world!"，原来那个对象还存在于内存之中，只是 s 这个引用变量不再指向它了。 通过上面的说明，我们很容易导出另一个结论，如果经常对字符串进行各种各样的修改，或者说，不可预见的修改，那么使用 String 来代表字符串的话会引起很大的内存开销。因为 String 对象建立之后不能再改变，所以对于每一个不同的字符串，都需要一个 String 对象来表示。这时，应该考虑使用 StringBuffer 类，它允许修改，而不是每个不同的字符串都要生成一个新的对象。并且，这两种类的对象转换十分容易。

4、**尽量少用静态变量** ，因为静态变量是全局的，GC 不会回收的；

5、**尽量避免在类的构造函数里创建、初始化大量的对象** ，防止在调用其自身类的构造器时造成不必要的内存资源浪费，尤其是大对象，JVM 会突然需要大量内存，这时必然会触发 GC

优化系统内存环境；显示的声明数组空间，而且申请数量还极大。　　以下是初始化不同类型的对象需要消耗的时间：

运算操作	示例	标准化时间
本地赋值	i = n	1.0
实例赋值	this.i = n	1.2
方法调用	Funct()	5.9
新建对象	New Object()	980
新建数组	New int[10]	3100

从表 1 可以看出，新建一个对象需要 980 个单位的时间，是本地赋值时间的 980 倍，是方法调用时间的 166 倍，而新建一个数组所花费的时间就更多了。

6、尽量在合适的场景下使用**对象池技术** 以提高系统性能，缩减缩减开销，但是要注意对象池的尺寸不宜过大，及时清除无效对象释放内存资源，综合考虑应用运行环境的内存资源限制，避免过高估计运行环境所提供内存资源的数量。

7、大集合对象拥有大数据量的业务对象的时候，可以考虑**分块进行处理** ，然后解决一块释放一块的策略。

8、**不要在经常调用的方法中创建对象** ，尤其是忌讳在循环中创建对象。可以适当的使用 hashtable，vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次 new 之后又丢弃。

9、一般都是发生在开启大型文件或跟数据库一次拿了太多的数据，造成 Out Of Memory Error 的状况，这时就大概要计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。

10、尽量少用 **finalize 函数** ，因为 finalize()会加大 GC 的工作量，而 GC 相当于耗费系统的计算能力。

11、**不要过滥使用哈希表** ，有一定开发经验的开发人员经常会使用 hash 表（hash 表在 JDK 中的一个实现就是 HashMap）来缓存一些数据，从而提高系统的运行速度。比如使用 HashMap 缓存一些物料信息、人员信息等基础资料，这在提高系统速度的同时也加大了系统的内存占用，特别是当缓存的资料比较多的时候。其实我们可以使用操作系统中的缓存的概念来解决这个问题，也就是给被缓存的分配一个一定大小的缓存容器，按照一定的算法淘汰不需要继续缓存的对象，这样一方面会因为进行了对象缓存而提高了系统的运行效率，同时由于缓存容器不是无限制扩大，从而也减少了系统的内存占用。现在有很多开源的缓存实现项目，比如 ehcache、oscache 等，这些项目都实现了 FIFO、MRU 等常见的缓存算法