第二版：JVM 20 道

目录

我们的网站：https://tech.souyunku.com

关注我们的公众号：搜云库技术团队，回复以下关键字

回复:【进群】邀请您进「技术架构分享群」

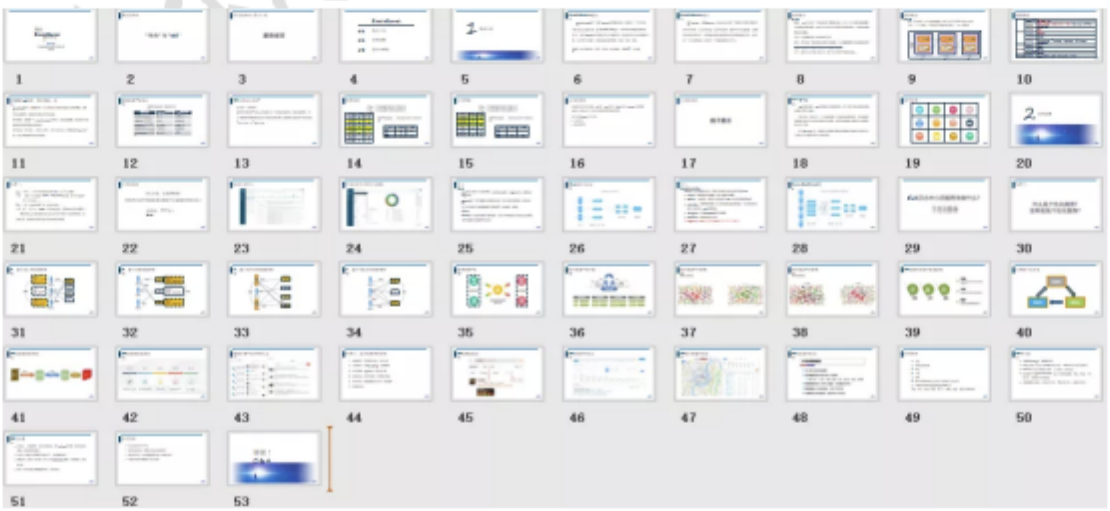
回复:【内推】即可进：北京，上海，广周，深圳，杭州，成都，武汉，南京，郑州，西安，长沙「程序员工作内推群」

回复 【1024】送 4000G 最新架构师视频

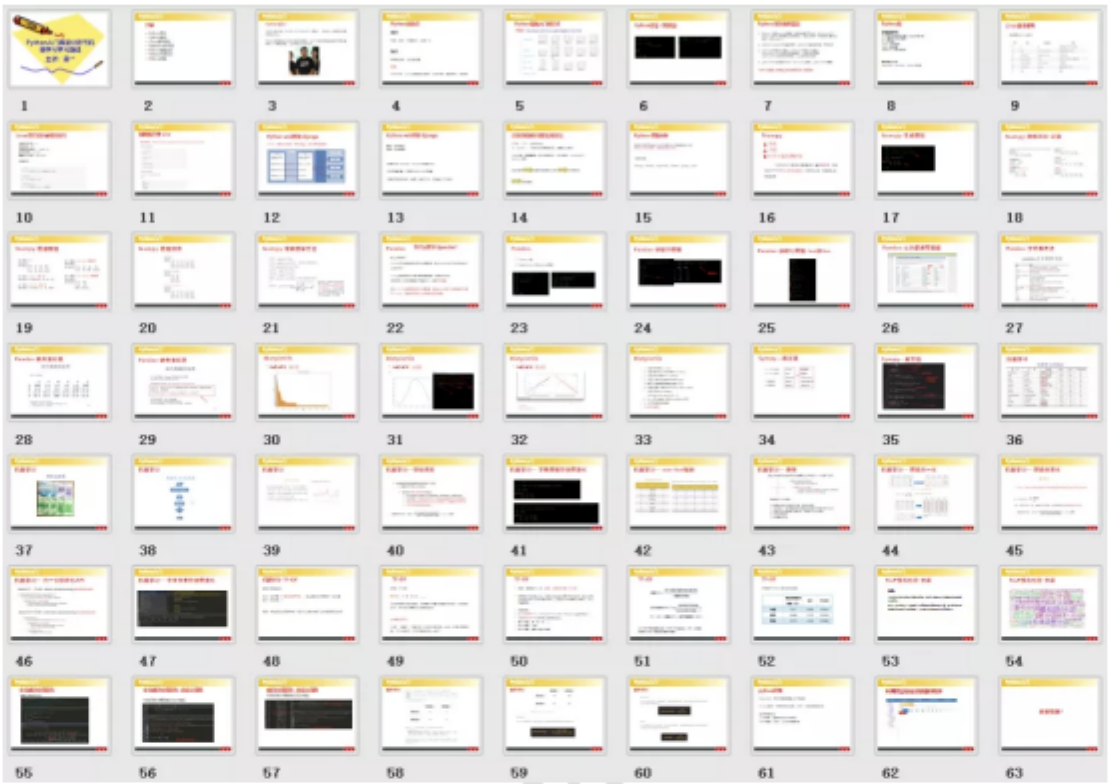
回复 【PPT】即可无套路获取，以下最新整理调优 PPT!

46 页《JVM 深度调优，演讲 PPT》

53 页《Elasticsearch 调优演讲 PPT》



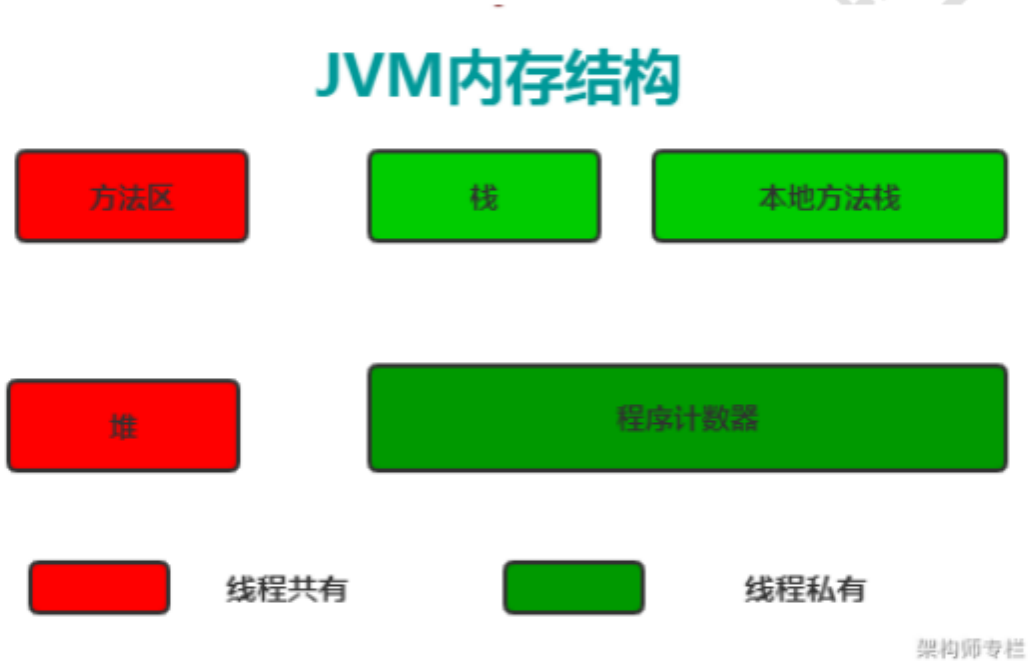
63 页《Python 数据分析入门 PPT》



一、JVM 内存区域相关

1.JVM 的内存模型以及分区情况和作用

如下图所示：



黄色部分为线程共有，蓝色部分为线程私有。

方法区

用于存储虚拟机加载的类信息，常量，静态变量等数据。

堆

存放对象实例，所有的对象和数组都要在堆上分配。 是 JVM 所管理的内存中最大的一块区域。

栈

Java 方法执行的内存模型：存储局部变量表，操作数栈，动态链接，方法出口等信息。生命周期与线程相同。

本地方法栈

作用与虚拟机栈类似，不同点本地方法栈为 native 方法执行服务，虚拟机栈为虚拟机执行的 Java 方法服务。

程序计数器

当前线程所执行的行号指示器。是 JVM 内存区域最小的一块区域。执行字节码工作时就是利用程序计数器来选取下一条需要执行的字节码指令。

2.Java 内存分配。

寄存器：我们无法控制。
静态域：static 定义的静态成员。
常量池：编译时被确定并保存在 .class 文件中的（final）常量值和一些文本修饰的符号引用（类和接口的全限定名，字段的名称和描述符，方法和名称和描述符）。
非 RAM 存储：硬盘等永久存储空间。
堆内存：new 创建的对象和数组，由 Java 虚拟机自动垃圾回收器管理,存取速度慢。
栈内存：基本类型的变量和对象的引用变量（堆内存空间的访问地址），速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

3.Java 堆的结构是什么样子的？什么是堆中的永久代（Perm Gen space）？

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些 对象回收掉之前，他们会一直占据堆内存空间。

3.Java 中会存在内存泄漏?简述一下

所谓内存泄露就是指一个不再被程序使用的对象或变量一直被占据在内存中。Java 中有垃圾回收机制，它可以保证一对象不再被引用的时候，即对象变成了孤儿的时候，对象将自动被垃圾回收器从内存中清除掉。由于 Java 使用有向图的方式进行垃圾回收管理，可以消除引用循环的问题，例如有两个对象，相互引用，只要它们和根进程不可达的，那么 GC 也是可以回收它们的，例如下面的代码可以看到这种情况的内存回收：

```
import java.io.IOException;
public class GarbageTest {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        try {
            gcTest();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

```
        System.out.println("has exited gcTest!");
        System.in.read();
        System.in.read();
        System.out.println("out begin gc!");
        for(int i=0;i<100;i++)
        {
            System.gc();
            System.in.read();
            System.in.read();
        }
    }
    private static void gcTest() throws IOException {
        System.in.read();
        System.in.read();
        Person p1 = new Person();
        System.in.read();
        System.in.read();
        Person p2 = new Person();
        p1.setMate(p2);
        p2.setMate(p1);
        System.out.println("before exit gctest!");
        System.in.read();
        System.in.read();
        System.gc();
        System.out.println("exit gctest!");
    }
    private static class Person
    {
        byte[] data = new byte[20000000];
        Person mate = null;
        public void setMate(Person other)
```

```
        {
            mate = other;
        }
    }
}
```

Java 中的内存泄露的情况：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露，尽管短生命周期对象已经不再需要，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，这就是 Java 中内存泄露的发生场景，通俗地说，就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的，这就是 java 中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中（例如放在一个全局 map 对象中），然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。

检查 Java 中的内存泄露，一定要让程序将各种分支情况都完整执行到程序结束，然后看某个对象是否被使用过，如果没有，则才能判定这个对象属于内存泄露。

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，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

下面内容来自于网上（主要特点就是清空堆栈中的某个元素，并不是彻底把它从数组中拿掉，而是把存储的总数减少，本人写得可以比这个好，在拿掉某个元素时，顺便也让它从数组中消失，将那个元素所在的位置的值设置为 null 即可）：

我实在想不到比那个堆栈更经典的例子了，以致于我还要引用别人的例子，下面的例子不是我想到的，是书上看到的，当然如果没有在书上看到，可能过一段时间我自己也想的到，可是那时我说是我自己想到的也没有人相信的。

```
public class Stack {
```

```
    private Object[] elements=new Object[10];
    private int size = 0;
    public void push(Object e){
        ensureCapacity();
        elements[size++] = e;
    }

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

    private void ensureCapacity(){
        if(elements.length == size){
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length+1];
            System.arraycopy(oldElements,0, elements, 0, size);
            }
    }
}
```

上面的原理应该很简单，假如堆栈加了 10 个元素，然后全部弹出来，虽然堆栈是空的，没有我们要的东西，但是这是个对象是无法回收的，这个才符合了内存泄露的两个条件：无用，无法回收。但是就是存在这样的东西也不一定会导致什么样的后果，如果这个堆栈用的比较少，也就浪费了几个 K 内存而已，反正我们的内存都上 G 了，哪里会有什么影响，再说这个东西很快就会被回收的，有什么关系。下面看两个例子。

```
public class Bad{
    public static Stack s=Stack();
    static{
```

```
            s.push(new Object());
            s.pop(); //这里有一个对象发生内存泄露
            s.push(new Object()); //上面的对象可以被回收了，等于是自愈了
        }
}
```

因为是 static，就一直存在到程序退出，但是我们也可以看到它有自愈功能，就是说如果你的 Stack 最多有 100 个对象，那么最多也就只有 100 个对象无法被回收其实这个应该很容易理解，Stack 内部持有 100 个引用，最坏的情况就是他们都是无用的，因为我们一旦放新的进取，以前的引用自然消失！

内存泄露的另外一种情况：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，造成内存泄露。

二、垃圾回收相关

1.GC 是什么？为什么要有 GC？

GC 是垃圾收集的意思（GabageCollection），内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

2.说说 Java 垃圾回收机制。

在 Java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

3.如何判断一个对象是否存活?

判断一个对象是否存活有两种方法：

1、 引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A、B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2、 可达性算法（引用链法）

该算法的思想是：从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。

在 Java 中可以作为 GC Roots 的对象有以下几种：

1、虚拟机栈中引用的对象

2、方法区类静态属性引用的对象

3、方法区常量池引用的对象

4、本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比不一定会被回收。当一个对象不可达 GC Root 时，这个对象并不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记.

如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize() 方法。当对象没有覆盖 finalize() 方法或者已被虚拟机调用过，那么就认为是没必要的。 如果该对象有必要执行 finalize() 方法，那么这个对象将会放在一个称为 F-Queue 的对队列中，虚拟机会触发一个 Finalize() 线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize() 执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行第二次被标记，这时，该对象将被移除” 即将回收” 集合，等待回收。

4. 垃圾回收的优点和原理。说说 2 种回收机制。

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 C++ 程序员最头疼的内存管理的问题迎刃而解，它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有“作用域”的概念，只有对象的引用才有"作用域"。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低级别的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清楚和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

回收机制有分代复制垃圾回收和标记垃圾回收，增量垃圾回收。

5.垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和管理堆（heap）中的所有对象。通过这种方式确定哪些对象是”可达的”，哪些对象是”不可达的”。当 GC 确定一些对象为“不可达”时，GC 就有责任回收这些内存空间。可以。程序员可以

手动执行 System.gc()，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

6.什么是分布式垃圾回收（DGC）？它是如何工作的?

DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

7. 串行（serial）收集器和吞吐量（throughput）收集器的区别是什么?

吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。 而串行收集器对大多数的小应用（在现代处理器上需要大概 100M 左右的内存）就足够了。

8.在 Java 中，对象什么时候可以被垃圾回收?

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

9.简述 Java 内存分配与回收策率以及 Minor GC 和 Major GC。

- 对象优先在堆的 Eden 区分配
- 大对象直接进入老年代
- 长期存活的对象将直接进入老年代

当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC。Minor GC 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快；Full GC/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC，但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

10. 介绍一下 JVM 中垃圾收集器有哪些？他们特点分别是什么？

新生代垃圾收集器

Serial 收集器

特点： Serial 收集器只能使用一条线程进行垃圾收集工作，并且在进行垃圾收集的时候，所有的工作线程都需要停止工作，等待垃圾收集线程完成以后，其他线程才可以继续工作。

使用算法：复制算法

ParNew 收集器

特点： ParNew 垃圾收集器是 Serial 收集器的多线程版本。为了利用 CPU 多核多线程的优势，ParNew 收集器可以运行多个收集线程来进行垃圾收集工作。这样可以提高垃圾收集过程的效率。

使用算法：复制算法

Parallel Scavenge 收集器

特点： Parallel Scavenge 收集器是一款多线程的垃圾收集器，但是它又和 ParNew 有很大的不同点。

Parallel Scavenge 收集器和其他收集器的关注点不同。其他收集器，比如 ParNew 和 CMS 这些收集器，它们主要关注的是如何缩短垃圾收集的时间。而 Parallel Scavenge 收集器关注的是如何控制系统运行的吞吐量。这里说的吞吐量，指的是 CPU 用于运行应用程序的时间和 CPU 总时间的占比，吞吐量 = 代码运行时间 /（代码运行时间 + 垃圾收集时间）。如果虚拟机运行的总的 CPU 时间是 100 分钟，而用于执行垃圾收集的时间为 1 分钟，那么吞吐量就是 99%。

使用算法：复制算法

老年代垃圾收集器

Serial Old 收集器

特点： Serial Old 收集器是 Serial 收集器的老年代版本。这款收集器主要用于客户端应用程序中作为老年代的垃圾收集器，也可以作为服务端应用程序的垃圾收集器。

使用算法：标记-整理

Parallel Old 收集器

特点： Parallel Old 收集器是 Parallel Scavenge 收集器的老年代版本这个收集器是在 JDK1.6 版本中出现的，所以在 JDK1.6 之前，新生代的 Parallel Scavenge 只能和 Serial Old 这款单线程的老年代收集器配合使用。Parallel Old 垃圾收集器和 Parallel Scavenge 收集器一样，也是一款关注吞吐量的垃圾

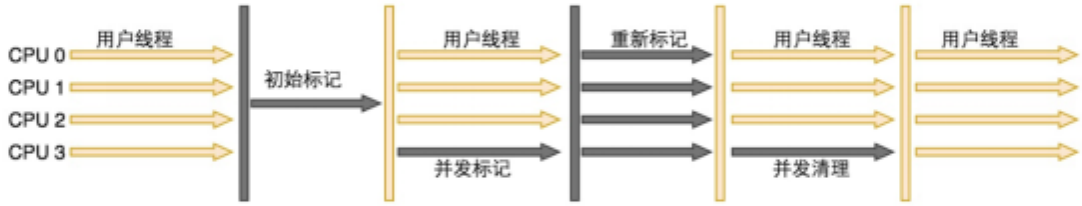
收集器，和 Parallel Scavenge 收集器一起配合，可以实现对 Java 堆内存的吞吐量优先的垃圾收集策略。

使用算法：标记-整理

CMS 收集器

特点： CMS 收集器是目前老年代收集器中比较优秀的垃圾收集器。CMS 是 Concurrent Mark Sweep，从名字可以看出，这是一款使用"标记-清除"算法的并发收集器。

CMS 垃圾收集器是一款以获取最短停顿时间为目标的收集器。如下图所示：



从图中可以看出，CMS 收集器的工作过程可以分为 4 个阶段：

- 初始标记（CMS initial mark）阶段
- 并发标记（CMS concurrent mark）阶段
- 重新标记（CMS remark）阶段
- 并发清除(（CMS concurrent sweep）阶段

使用算法：复制+标记清除

其他

G1 垃圾收集器

特点： 主要步骤：初始标记，并发标记，重新标记，复制清除。

使用算法：复制 + 标记整理

11.Java 中垃圾收集的方法有哪些?

标记 - 清除：这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：

1、 效率不高，标记和清除的效率都很低；
2、 会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作。

复制算法：为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。
于是将该算法进行了改进，内存区域不再是按照 1:1 去划分，而是将内存划分为 8:1:1 三部分，较大那份内存交 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。（java 堆又分为新生代和老年代）

标记 - 整理：该算法主要是为了解决标记 - 清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

分代收集：现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保。

三、类加载相关

1.什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器：

1.启动类加载器（Bootstrap ClassLoader）用来加载 Java 核心类库，无法被 Java 程序直接引用。

2.扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3.系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。

4.用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

2.类加载器双亲委派模型机制?

基本定义： 双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器没有找到所需的类时，子加载器才会尝试去加载该类。

双亲委派机制：

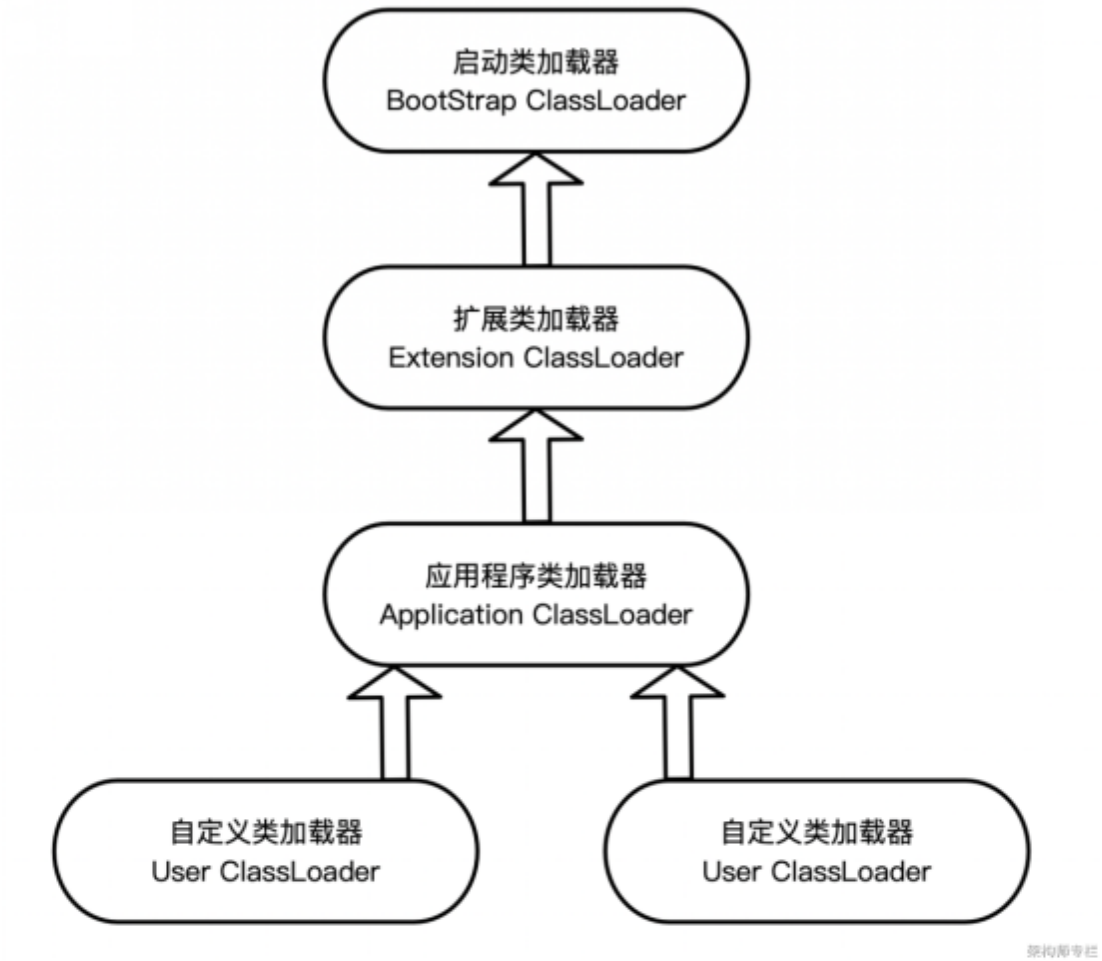
1、 当 AppClassLoader 加载一个 class 时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器 ExtClassLoader 去完成。

2、 当 ExtClassLoader 加载一个 class 时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给 BootStrapClassLoader 去完成。

3、 如果 BootStrapClassLoader 加载失败，会使用 ExtClassLoader 来尝试加载；

4、 若 ExtClassLoader 也加载失败，则会使用 AppClassLoader 来加载，如果 AppClassLoader 也加载失败，则会报出异常 ClassNotFoundException。

如下图所示：



双亲委派作用：

- 通过带有优先级的层级关可以避免类的重复加载；
- 保证 Java 程序安全稳定运行，Java 核心 API 定义类型不会被随意替换。

3.什么是 Class 文件？ Class 文件主要的信息结构有哪些？

Class 文件是一组以 8 位字节为基础单位的二进制流。各个数据项严格按顺序排列。

Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据。这样的伪结构仅仅有两种数据类型：无符号数和表。

无符号数：是基本数据类型。以 u1、u2、u4、u8 分别代表 1 个字节、2 个字节、4 个字节、8 个字节的无符号数，能够用来描写叙述数字、索引引用、数量值或者依照 UTF-8 编码构成的字符串值。

表：由多个无符号数或者其它表作为数据项构成的复合数据类型。全部表都习惯性地以 _info 结尾。

四、JVM 调优

1.JVM 数据运行区，哪些会造成 OOM 的情况?

除了数据运行区，其他区域均有可能造成 OOM 的情况。

堆溢出： java.lang.OutOfMemoryError: Java heap space

栈溢出： java.lang.StackOverflowError

永久代溢出： java.lang.OutOfMemoryError: PermGen space

2.线上常用的 JVM 参数有哪些?

数据区设置

Xms：初始堆大小

Xmx：最大堆大小

Xss:Java 每个线程的 Stack 大小

XX:NewSize=n：设置年轻代大小

XX:NewRatio=n：设置年轻代和年老代的比值。如：为 3，表示年轻代与年老代比值为 1:3，年轻代占整个年轻代年老代和的 1/4。

XX：SurvivorRatio=n：年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor 区占整个年轻代的 1/5。

XX：MaxPermSize=n：设置持久代大小。

收集器设置

XX:+UseSerialGC：设置串行收集器

XX:+UseParallelGC:：设置并行收集器

XX:+UseParalledlOldGC：设置并行年老代收集器

XX:+UseConcMarkSweepGC：设置并发收集器

GC 日志打印设置

XX:+PrintGC：打印 GC 的简要信息

XX:+PrintGCDetails：打印 GC 详细信息

XX:+PrintGCTimeStamps：输出 GC 的时间戳

3.JVM 提供的常用工具

jps：用来显示本地的 Java 进程，可以查看本地运行着几个 Java 程序，并显示他们的进程号。 命令格式：jps

jinfo：运行环境参数：Java System 属性和 JVM 命令行参数，Java class path 等信息。 命令格式：jinfo 进程 pid

jstat：监视虚拟机各种运行状态信息的命令行工具。 命令格式：jstat -gc 123 250 20

jstack：可以观察到 JVM 中当前所有线程的运行情况和线程当前状态。 命令格式：jstack 进程 pid

jmap：观察运行中的 JVM 物理内存的占用情况（如：产生哪些对象，及其数量）。命令格式：jmap [option] pid