愿景："让编程不在难学，让技术与生活更加有趣"

第1章 JVM课程学习说明书

第1集 互联网架构之JAVA虚拟机课程介绍

- JVM学习的必要性、优先级
- 团队介绍
- 知识介绍

愿景："让编程不在难学，让技术与生活更加有趣"

第2章 开门见山大厂面试题之运行时数据区

第1集 大厂高频面试题Java内存区域分布与概述

简介：JVM内存模型的概述

- Java语言为甚么优势巨大，一处编译随处运行
- Java的另外一个优势
    - 自从内存管理机制之下，不再需要为没一个new操作去写配对的内存分配和回收等代码， 不容易出现内存泄漏和内存溢出等问题
- JVM运行时数据区分布图讲解
    - 线程共享数据区：方法区、堆
    - 线程隔离数据区：虚拟机栈、本地方法栈、堆、程序计数器

第2集 动手实战Java内存区域程序计算器

简介：JVM内存模型之程序计算器

- 是什么?
    - 程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器
    - 线程是一个独立的执行单元，是由CPU控制执行的
    - 字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分

支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成

- 为什么?
  - 为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存
- 特点?
  - 内存区域中唯一一个没有规定任何 OutOfMemoryError 情况的区域

第3集 动手实战Java内存区域JAVA虚拟机栈

简介：JVM内存模型之java虚拟机栈讲解

- 是什么?
- 用于作用于方法执行的一块Java内存区域
- 为什么?
  - 每个方法在执行的同时都会创建一个栈帧（Stack Framel）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程
- 特点?
  - 局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）以及对象引用（reference 类型）
  - 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常

```
import java.util.concurrent.atomic.AtomicInteger;

public class A {

    public native static void c();
    public static void a(){
        System.out.println("enter method a");
    }

    public static void b(){
        a();
        System.out.println("enter method b");
    }

    public static void main(String[] args) {
        b();
        System.out.println("enter method main");
        AtomicInteger atomicInteger = new AtomicInteger(1);
        atomicInteger.compareAndSet(1,2);
    }
}
```

第4集 动手实战Java内存区域本地方法栈

简介：JVM内存模型之本地方法栈讲解

- 是什么?
  - 用于作用域本地方法执行的一块Java内存区域
  - 什么是本地方法?
- 为什么?
  - 与Java虚拟机栈相同，每个方法在执行的同时都会创建一个栈帧（Stack Framel）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程
- 特点?
  - Hotshot将Java虚拟机栈和本地方法栈合二为一

第5集 动手实战Java内存区域Java堆

简介：JVM内存模型之JAVA堆

- 是什么?
  - 是Java内存区域中一块用来存放对象实例的区域，【几乎所有的对象实例都在这里分配内存】
- 为什么?
  - 此内存区域的唯一目的就是存放对象实例
  - Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块 Java 堆是被所有线程共享的一块内存区域
- 特点?
  - Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”（Garbage
  - -Xmx -Xms
  - Java堆可以分成新生代和老年代 新生代可分为To Space、From Space、Eden

第6集 动手实战Java内存区域方法区

简介：JVM内存模型之方法区

- 是什么?

    - 是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据
    - 什么是类信息：类版本号、方法、接口
- 为什么?
    - 内存中存放类信息、静态变量等数据，属于线程共享的一块区域
    - Hotspot使用永久代来实现方法区 JRockit、IBM J9VM Java堆一样管理这部分内存
- 特点?
    - 并非数据进入了方法区就如永久代的名字一样“永久”存在了。这区域的内存回收目标主要是针对常量池的回收和对类型的卸载
    - 方法区也会抛出OutofMemoryError，当它无法满足内存分配需求时

第7集 带你用上神秘的运行时常量池

简介：JVM内存模型之方法区运行时常量池

- 是什么?
    - 运行时常量池是方法区的一部分，Class文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。
- 运行时常量池的模拟
- 特点：
    - 运行时常量池是方法区的一部分，受到方法区内存的限制，当常量池再申请到内存时会抛出OutOfMemoryError异常

```
public class A {

    public static void main(String[] args) {
        String a = "abc";
        String b = "abc";
        System.out.println(a==b);
        String c = new String("abc");
        System.out.println(a==c);
        System.out.println(a==c.intern());
    }
}
```

愿景："让编程不在难学，让技术与生活更加有趣"

第3章 你真的了解对象吗?

第1集 你的对象是怎么来的?

简介：Java程序员不可不知的对象创建底层步骤细节

- 对象创建的流程步骤包括哪些：
    - 虚拟机遇到一条new指令时，首先检查这个对应的类能否在常量池中定位到一个类的符号引用
    - 判断这个类是否已被加载、解析和初始化
    - 为这个新生对象在Java堆中分配内存空间，其中Java堆分配内存空间的方式主要有以下两种
        - 指针碰撞
            - 分配内存空间包括开辟一块内存和移动指针两个步骤
            - 非原子步骤可能出现并发问题，Java虚拟机采用CAS配上失败重试的方式保证更新操作的原子性
        - 空闲列表
            - 分配内存空间包括开辟一块内存和修改空闲列表两个步骤
            - 非原子步骤可能出现并发问题，Java虚拟机采用CAS配上失败重试的方式保证更新操作的原子性
    - 将分配到的内存空间都初始化为零值
    - 设置对象头相关数据
        - GC分代年龄
        - 对象的哈希码 hashCode
        - 元数据信息
    - 执行对象方法

- 代码分析对象执行的过程

第2集 new这么多对象，你知道它们的结构吗?

简介：Java程序员不可不知的对象结构

- 对象头用于存储对象的元数据信息：
    - Mark Word 部分数据的长度在32位和64位虚拟机（未开启压缩指针）中分别为32bit和64bit，存储对象自身的运行时数据如哈希值等。Mark Word一般被设计为非固定的数据结构，以便存储更多的数据信息和复用自己的存储空间。
    - 类型指针 指向它的类元数据的指针，用于判断对象属于哪个类的实例。
- 实例数据存储的是真正有效数据，如各种字段内容，各字段的分配策略为longs/doubles、ints、shorts/chars、bytes/boolean、oops(ordinary object pointers)，相同宽度的字段总是被分配到一起，便于之后取数据。父类定义的变量会出现在子类定义的变量的前面。
- 对齐填充部分仅仅起到占位符的作用

第3集 你是怎么访问你的对象的

简介：Java程序员不可不知的对象访问定位方式

- 当我们在堆上创建一个对象实例后，就要通过虚拟机栈中的reference类型数据来操作堆上的对象。现在主流的访问方式有两种（HotSpot虚拟机采用的是第二种）：
  1. 使用句柄访问对象。即reference中存储的是对象句柄的地址，而句柄中包含了对象实例数据与类型数据的具体地址信息，相当于二级指针。
  2. 直接指针访问对象。即reference中存储的就是对象地址，相当于一级指针。
- 对比
  - 垃圾回收分析：方式1️⃣当垃圾回收移动对象时，reference中存储的地址是稳定的地址，不需要修改，仅需要修改对象句柄的地址；方式2️⃣垃圾回收时需要修改reference中存储的地址。
  - 访问效率分析，方式二优于方式一，因为方式二只进行了一次指针定位，节省了时间开销，而这也是HotSpot采用的实现方式。

愿景："让编程不在难学，让技术与生活更加有趣"

第4章 内功深厚招数才易懂垃圾回收算法

第1集 你不得不懂的GC垃圾回收?

简介：GC垃圾回收讲解

- 战略意义 能做出一个需求的同时也要懂得其对应的战略意义
- 为什么要垃圾回收?
  - 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。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，有效的使用空闲的内存
- 垃圾回收的过程是怎样的?
- 如果让你考虑垃圾回收算法你会怎么设计
  - 完成哪些对象回收哪些对象不回收的功能需求
- 对象是否存活判断
  - 堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1

第2集 对象存活算法引用计数法

简介：判断对象是否存活算法，讲解对象垃圾回收对象是否回收判断

- 引用计数法存在的特点分析
- 优缺点
  - 引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。
  - 无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.
- 代码分析JVM是否用引用计数法
- run configurations—vm options—加配置项

```
-verbose:gc  -XX:+PrintGCDetails
```

第3集 对象存活算法可达性分析

简介：判断对象是否存活算法，讲解对象垃圾回收对象是否回收判断

- 可达性分析算法的概念(又叫跟搜索法)
    - 根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点
- java中可作为GC Root的对象有
    - 虚拟机栈中引用的对象（本地变量表）
    - 本地方法栈中引用的对象
    - 方法区中静态属性引用的对象
    - 方法区中常量引用的对象

第4集 算法内功之剖析标记清除

简介：标记清除算法讲解

- 大厂的标准就是原理
- 最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：
    - 首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，它的标记过程其实在前- -节讲述对象标记判定时已经介绍过了。
    - 它的主要不足有两个：
        - 一个是效率问题，标记和清除两个过程的效率都不高；
        - 另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作

第5集 算法内功之剖析复制算法

简介：标记复制算法讲解

- 为甚么出现复制算法?
    - 为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按量划分为大小相等的两块，每次只使用其中的一块
    - 当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效
- 现在的商业虚拟机都采用这种收集算法来回收新生代，研究表明，新生代中的对象 98%是“朝生夕死”的，所以并不需要按照 1:1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。 Survivor from 和Survivor to ，内存比例 8：1：1
- 当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1, 也就是每次新生代中可用内存空间为整个新生代容量的 90% (80%+10%），只有 10% 的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10%的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

第6集 剖析标记整理算法与分代收集算法

简介：标记整理算法讲解以及分代收集算法讲解

- 标记整理算法解决了什么问题
    - 复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费 50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都 100%存活的极端情况，所以在老年代一般不能直接选用这种算法
- 标记-整理
    - 根据老年代的特点，有人提出了另外一种“标记-整理（Mark- Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存
- 分代收集
    - 一般把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法
    - 在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有

额外空间对它进行分配担保，就必须使用“标记-清理”或者“标记一整理”算法来进行回收

愿景："让编程不在难学，让技术与生活更加有趣"

第5章 偷偷在干活的JVM垃圾收集器

第1集 Serial收集器内容精讲

简介：serial垃圾收集器讲解

- hashMap list，hashcode
- 是什么?
  - 收集算法是内存收到的方法论，垃圾回收器是内存回收的具体实现。
  - Serial是一个单线程的垃圾收集器

- serial垃圾收集器的特点
  - “Stop The World”，它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。在用户不可见的情况下把用户正常工作的线程全部停掉
  - 使用场景：多用于桌面应用，Client端的垃圾回收器
  - 桌面应用内存小，进行垃圾回收的时间比较短，只要不频繁发生停顿就可以接受

第2集 ParNew收集器内容精讲

简介：parnew垃圾收集器图解分析特点

- 是什么?

    - ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：-XX: SurvivorRatio、-XX: PretenureSize' Threshold、-XX: HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，在实现上，这两种收集器也共用了相当多的代码
- parnew垃圾收集器的特点?
    - ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。
    - 使用-XX: ParallelGCThreads 参数来限制垃圾收集的线程数
    - 多线程操作存在上下文切换的问题，所以建议将-XX: ParallelGCThreads设置成和CPU核数相同，如果设置太多的话就会产生上下文切换消耗
- 并发与并行的概念讲解 CMS垃圾回收器
    - 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
    - 并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个 CPU 上

第3集 Parallel Scavenge收集器

简介：Parallel Scavenge收集器讲解

- 是什么?
    - Parallel Scavenge 收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器
    - 由于与吞吐量关系密切，Parallel Scavenge 收集器也经常称为“吞吐量优先”收集器
    - 吞吐量是什么？CPU用于运行用户代码的时间与CPU总时间的比值，99%时间执行用户线程，1%时间回收垃圾 ，这时候吞吐量就是99%
- 特点：
    - Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到个可控制的吞吐（Throughput）。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99% 停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。
    - 虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停

顿时间或者最大的吞吐量，这种调节方式称为 GC自适应调节策略
    - -XX:MaxGCPauseMillis参数GC停顿时间，500MB ——>300MB,这个参数配置太小的话会发生频繁GC
    - -XX:GCTimeRatio参数，99%
- Serial old收集器，它是一个单线程收集器，使用"标记--整理"算法
- Parallel old收集器Parallel Scavenge收集器的老年代版本，使用多线程+标记整理算法

第4集 你不得不懂的CMS收集器

简介：标记整理算法讲解以及分代收集算法讲解

- 是什么?
    - CMS (Concurrent Mark Sweep）收集器是-种以获取最短回收停顿时间为目标的收集器。
    - 目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上,这类应用尤其重 视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。
    - CMS 收集器是基于“标记-清除”算法实现的

- 步骤流程：
    - 初始标记（CMS initial mark) -----标记一下 GC Roots 能直接关联到的对象，速度很快
    - 并发标记（CMS concurrent mark --------并发标记阶段就是进行 GC RootsTracing 的过程
    - 重新标记（CMS remark) ----------为了修正并发标记期间因用户程序导致标记产生变动的标记记录
    - 并发清除（CMS concurrent sweep)

- CMS垃圾收集器缺点
    - 对CPU资源非常敏感
    - 无法处理浮动垃圾，程序在进行并发清除阶段用户线程所产生的新垃圾
    - 标记-清除暂时空间碎片

第5集 你不得不懂的G1收集器

简介：G1垃圾收集器

- 是什么
  - G1是一款面向服务端应用的垃圾收集器
- 特点：
  - G1 中每个 Region 都有一个与之对应的 Remembered Set，当进行内存回收时，在 GC 根节点的枚举范围中加入 Remembered Set 即可保证不对全堆扫描也不会有遗漏 检查Reference引用的对象是否处于不同的Region
- G1 收集器的运作大致可划分为以下几个步骤
  - 初始标记（Initial Marking） --标记一下 GC Roots 能直接关联到的对象
  - 并发标记（Concurrent Marking）---从GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行
  - 最终标记（Final Marking) ---为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录。虚拟机将这段时间对象变化记录在线程 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs的数据合并到 Remembered Set 中
  - 筛选回收（Live Data Counting and Evacuation)
- G1的优势有哪些
  - 空间整合：基于“标记一整理”算法实现为主和Region之间采用复制算法实现的垃圾收集
  - 可预测的停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型
  - 在 G1 之前的其他收集器进行收集的范围都是整个新生代或者老年代，而 G1 不再是这样。使用 G1 收集器时，Java 堆的内存布局就与其他收集器有很大差别，它将整个 Java 雄划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔高的了，它们都是一部分 Region（不需要连续）的集合。
  - G1 收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。G1 跟踪各个 Regions 里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region（这也就是 Garbage- Firsti 名称的来由）。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限的时间内可以获取尽可能高

愿景："让编程不在难学，让技术与生活更加有趣"

第6章 简明扼要内存分配

第1集 探索堆内存到底是怎么分配

简介：堆内存分配方式，分配规则讲解

- Java堆内存区域的划分以及作用讲解
- 对象分配的规则有哪些
    - 对象主要分配在新生代的 Eden 区上
    - 如果启动了本地线程分配缓冲，将按线程优先在 TLAB 上分配
    - 少数情况下也可能会直接分配在老年代中
-
- GC参数指定垃圾回收
    - -Xms20 M、-Xmx20 M、-Xmn1 0 M 这 3 个参数限制了 Java 堆大小为 20 MB，不可扩展，其中 10 MB 分配给新生代，剩下的 10 MB 分配给老年代。-Xx: SurvivorRatio= 8 决定了新生代中 Eden 区与两个 Survivor 区的空间比例是 8:1
- 新生代与老年代
    - 新生代 GC (Minor GC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度老年代 GC (Major GC/ Full GC）：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。Major GC 的速度一般会比 Minor GC 慢 10 倍以上。

第2集 大对象的分配和基本分配一样吗?

简介：堆内存JVM参数讲解，大对象分配原则讲解和代码验证

- 是什么?
    - 所谓的大对象是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串以及数组
    - 虚拟机提供了一个-XX: PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制
- 实战代码演练大对象配置
    - -verbose:gc -XX:+PrintGCDetails 开启GC日志打印
    - -Xms20 M 设置JVM初始内存为20M
    - -Xmx20 M 设置JVM最大内存为20M
    - -Xmn10 M 设置年轻代内存大小为10M

```
-verbose:gc -XX:+PrintGCDetails -XX:+UseSerialGC -Xms20M -Xmx20M -Xmn10M -
XX:PretenureSizeThreshold=3145728
```

第3集 逃逸分析和栈上分配

简介：堆内存分配方式，分配规则讲解

- 逃逸分析
  - 逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，称为方法逃逸。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为线程逃逸
- 栈上分配
  - 栈上分配就是把方法中的变量和对象分配到栈上，方法执行完后自动销毁，而不需要垃圾回收的介入，从而提高系统性能

```
-XX:+DoEscapeAnalysis开启逃逸分析（jdk1.8默认开启，其它版本未测试）
-XX:-DoEscapeAnalysis 关闭逃逸分析
```

愿景："让编程不在难学，让技术与生活更加有趣"

第7章 中高级工程师必备线上虚拟机工具

第1集 实战应用之使用虚拟机工具jps

简介：虚拟机工具的意义以及JPS讲解

- 虚拟机工具的意义
  - 给一个系统定位问题的时候，知识、经验是关键基础，数据是依据，工具是运用知识处理数据的手段
  - 数据包括：运行日志、异常堆栈、GC 日志、线程快照

    (threaddump/javacore文件）、堆转储快照（ heapdump/hprof文件)等。使用适当的虚拟机监控和分析的工具可以加快我们分析数据、定位解决问题的速度

- ps -ef|grep java
  - grep命令是查找
  - 中间的|是管道命令 是指ps命令与grep同时执行
  - PS是LINUX下最常用的也是非常强大的进程查看命令
  - grep命令是查找，是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来
- JPS是什么?
  - jps (JVM Process Status Tool）是其中的典型jvm工具。除了名字像 UNIX 的 ps 命令之外，它的功能也和 ps 命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类（Main Class, main（）函数所在的类）名称以及这些进程的本地虚拟机唯- ID (Local Virtual Machine Identifier, LVMID），虽然功能比较单一，但它是使用频率最高的 JDK 命令行工具
- 实战使用
  - jps -l 输出主类的全名，如果进程执行的是Jar包则输出Jar路径
  - jps -v 输出虚拟机进程启动时JVM参数

第2集 实战应用之使用虚拟机工具jstat与jinfo

简介：jstat命令和jinfo命令实战讲解

- jstat是什么
  - Jstat (JVM Statistics Monitoring Tool）是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程-虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GU 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具
- jstat命令使用

```
 jstat -gc 2764 250 20   //2764表示进程id ， 250表示250毫秒打印一次 ， 20表示一共打印20次
 S0C：第一个幸存区的大小
 S1C：第二个幸存区的大小
 S0U：第一个幸存区的使用大小
 S1U：第二个幸存区的使用大小
 EC：伊甸园区的大小
 EU：伊甸园区的使用大小
 OC：老年代大小
 OU：老年代使用大小
 MC：方法区大小
 MU：方法区使用大小
 CCSC:压缩类空间大小
 CCSU:压缩类空间使用大小
 YGC：年轻代垃圾回收次数
 YGCT：年轻代垃圾回收消耗时间
```

```
FGC：老年代垃圾回收次数
FGCT：老年代垃圾回收消耗时间
GCT：垃圾回收消耗总时间
```

- jinfo是什么?
  - jinfo (Configuration Info for Java）的作用是实时地查看和调整虚拟机各项参数。使用 jps 命令的-v 参数可以查看虚拟机启动时显式指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，除了去找资料外，就只能使用 info 的-flag 选项进行查询了
- jinfo命令使用

```
jinfo -flag CMSInititingOccupancyFraction 1444
```

第3集 实战应用之使用虚拟机工具jmap

简介：jmap常用命令讲解，linux命令说明书怎么看?

- jmap是什么?
  - Jmap (Memory Map for Java）命令用于生成堆转储快照。如果不使用 jmap 命令，要想获取 Java 堆转储快照，还有一些比较"暴力"的手段：-XX: +HeapDumpOnOutOfMemoryError 参数，可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，用于系统复盘环节
  - 和 info 命令一样，jmap 有不少功能在 Windows 平台下都是受限的，除了生成 dump 文件的- dump 选项和用于查看每个类的实例、空间占用统计的-histo选项在所有操作系统都提供之外，其余选项都只能在Linux/Solaris 下使用。
- jmap常用命令
  - -dump
    - 生成 Java 堆转储快照。格式为：-dump: format=b, file=

```
windows:  jmap -dump:format=b,file=d:\a.bin 1234
mac:      jmap -dump:format=b,file=/Users/daniel/deskTop
```

  - -histo more分页去查看
    - 显示堆中对象统计信息，包括类、实例数量、合计容量

    B： byte

    C : char

    I： Int

第4集 实战应用之使用虚拟机工具jhat

简介：jhat命令的是什么？jhat命令的使用

- 实战OOM场景dump下内存快照
- jhat是什么?
- - Sun JDK 提供 jhat (JVM Heap Analysis Tool）命令常与 jmap 搭配使用，来分析 jmap 生成的堆 转储快照。jhat内置了一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以在浏览器中查看
- 特点：
  - jhat分析工作是一个耗时而且消耗硬件资源的过程
  - jhat 的分析功能相对来说比较简陋

第5集 实战应用之使用虚拟机工具jstack

简介：堆内存分配方式，分配规则讲解

- Jstack是什么?
  - Jstack (Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照（-般称为threaddump 或者 javacore 文件）
  - 线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者等待着什么资源
- Jstack怎么做
  - 常用命令jstack -l 3500
  - jstack -F 当正常输出的请求不被响应时，强制输出线程堆栈 Force
  - 经典面试题之 【jstack怎么进行死锁问题定位？】
- 线上程序一般不能kill进程pid的方式直接关闭
  - shutdownHook :在关闭之前执行的任务

第6集 经典面试题之死锁

简介：线程死锁是什么？怎么模拟一个线程死锁？怎么用jstack定位到线程死锁

- 线程死锁是什么?
  - 线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行
- 死锁的模拟过程，代码模拟死锁

第7集 经典面试题之线程状态

简介：线程状态有哪些？他们之间是怎么切换的

- 线程状态分类
  - NEW
  - RUNNABLE
  - BLOCKED 一个正在阻塞等待一个监视器的线程处于这一状态
  - WAITING 一个正在无限期等待另一个线程执行一个特别的动作的线程处于这一状态
  - TIMED_WAITING 一个正在限时等待另一个线程执行一个动作的线程处于这一状态
  - TERMINATED

- Blocked状态与Waiting状态的区别
  - WAITING 状态属于主动地显式地申请的阻塞，BLOCKED 则属于被动的阻塞

愿景："让编程不在难学，让技术与生活更加有趣"

第8章 中高级工程师必备虚拟机图形化工具

第1集 可视化虚拟机工具Jconsole介绍

简介：jconsole是什么？怎么连接与使用介绍

- Jconsole是什么?
  - JConsole (Java Monitoring and Management Console）是一种基于 JMX 的可视化监视、管理工具，它管理部分的功能是针对 JMXMBean 进行管理，由于 MBean 可以使用代码、中间件服务器的管理控制台或者所有符合 JMX 规范的软件进行访问
  - 特点：jconsole集成了线程与内存的可视化展示
- Jconsole怎么连接使用?

- 本地连接
  - 通过JDK/bin目录下的“jconsole.exe”启动JConsole 后，将自动搜索出本机运行的所有虚拟机进程，不需要用户自己再使用 jps 来查询了
- 远程连接
  - mvn install 生成项目jar
  - scp将jar包进行远程复制

    ```
    sudo scp /Users/daniel/Desktop/jvm-demo-0.0.1-SNAPSHOT.jar
    daniel@172.16.244.151:/usr/local
    ```

  - 将jar包启动

    ```
    nohup java -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m
    -XX:MaxNewSize=512m -Dcom.sun.management.jmxremote.port=9999
    -Djava.rmi.server.hostname=172.16.244.151
    -Dcom.sun.management.jmxremote.ssl=false
    -Dcom.sun.management.jmxremote.authenticate=false -jar
    /Users/daniel/Desktop/jvm-demo-0.0.1-SNAPSHOT.jar &
    ```

    http与https的区别：https=http+ssl

第2集 可视化虚拟机工具Jconsole内存监控验证

简介：怎么设计程序验证jconsole内存监控功能

- 设计jconsole内存监控功能需求分析
  - 掌握需求分析能力的重要性
  - 作为一个有追求的程序员我们应该怎么分析需求
- 实战演练jconsole内存增长过程
  - 编写jconsole内存分配代码
  - 分析jconsole各个指标的增长情况

第3集 可视化虚拟机工具Jconsole线程验证

简介：怎么设计程序验证jconsole线程监控功能

- 设计jconsole线程监控功能需求分析
- 实战演练jconsole线程状态过程
    - 线程状态waiting、timeWaiting模拟

第4集 可视化虚拟机工具VisualVM

简介：VisualVM插件安装讲解以及基础线程功能/内存功能讲解

- VisualVM是什么?
    - VisualVM是一个集成命令行JDK工具和轻量级分析功能的可视化工具
- VisualVM怎么用?
    - 在IDEA安装VisualVM插件，File-> Setting-> Plugins -> Browers Repositrories 搜索VisualVM Launcher安装并重启IDEA
    - 点击配置VisualVM executable执行路径

      如： /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/jvisualvm
    - eclipse配置教程原理相同

第5集 可视化虚拟机工具VisualVM实战

简介：VisualVM实战

- VisualVM大对象模拟，HashMap和Integer大对象对应的在堆内存信息的展示
    - dump内存实时情况
- 学员需自己实战VisualVM部分内容
    - 怎么查看线程堆栈信息?
    - VisualVM怎么查看内存信息，堆信息，堆使用量大小

第9章 20W年薪面试必备实战篇

第1集 教你如何剖析传统项目JVM问题

简介：20W年薪应该掌握哪些实战知识

- 关于JAVA项目分类
  - 传统项目容易遇到哪些JVM问题
  - 互联网项目容易遇到哪些JVM问题
- 实战案例讲解，传统公司遇到的上传下载问题
  - 项目介绍
  - 项目特点分析
  - 瓶颈分析
    - 是否是单接口性能问题?

      如果是的是，先考虑是否有sql慢查询，定位慢查询的方法一般是用explain查看sql的执行计划

      select * from user u where u.id=1; ==> explain select * from user u where u.id=1;

      扫描行数是特别占CPU的，举个例子一个sql的扫描行数达到100w

第2集 FullGC与Minor的区别频繁FullGC问题分析

简介：FullGC与MinorGC讲解

- Minor GC触发条件：当Eden区满时，触发Minor GC
- FullGC触发条件
  - 调用 System.gc() 此方法的调用是建议 JVM 进行 Full GC，虽然只是建议而非一定，但很多情况下它会触发 Full GC。因此强烈建议能不使用此方法就不要使用，让虚拟机自己去管理它的内存。可通过 -XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc()
  - 老年代空间不足 老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等，当执行 Full GC 后空间仍然不足，则抛出 Java.lang.OutOfMemoryError。为避免以上原因引起的 Full GC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间以及不要创建过大的对象及数组
  - 空间分配担保失败 使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果出现了 HandlePromotionFailure 担保失败，则会触发 Full GC
- 此项目中出现频繁FullGC，也就是系统空间分配不足导致的系统堆内存强制回收
- 问题解决方法分析

    - 由于本机单服务内存过大导致，此场景下Full GC，而且需要回收的内存很大，持续时间过长
    - 解决停顿时间过长问题，缩短GC时间

第3集 下载导致的频繁FullGC问题演练与解决

简介：FullGC演练

- 下载问题是什么问题?
    - 用户线程访问所导致的大对象问题
- 怎么设计一个用户线程所导致的FullGC问题
- 解决这个问题的关键是什么?
    - 32G内存-xmx30G，系统每次进行FullGC时长太长
    - 可以减少-xmx大小成4G，从而缩短Full GC
    - 最终解决方案：集群部署，第一个节点4G 第二个节点4G 第三个节点4G 用nginx配置转发upstream

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
```

第4集 互联网项目常见的JVM问题剖析

简介：互联网项目常见JVM问题讲述

- 为什么访问量大就越容易出问题

    - 判断一个用户是否在白名单 List.contain(用户) true，==》Set.contain(用户) 通过hash比较==》布隆过滤器
    - 结论==》用户量大和用户量小的项目遇到的问题和解决方案也就不一样
- 案例1，关于死锁问题
    - 解决方案==》jstack -m命令查看帮我们检测是否有死锁，或者jconsole、jvisualVM也能检查
    - new Thread的时候最好带上名称
- 案例2，堆内存泄漏问题
    - 现象：出现OOM或者Full GC，heap使用率明显上升，经常达到Xmx
    - Full GC出现的正常频率是大概一天一到两次
    - 解决方案==》jmap dump下内存/heap dump on OOM/heap dump on FullGC+jhat本地分析/mat（eclipse），或者jconsole、jvisualVM也能检查
- 案例3，堆外内存泄漏
    - 现象：heap使用率很低，但是出现了OOM或者Full GC
    - 解决方案==》可以用btrace跟踪DirectByteBuffer的构造函数来定位

第5集 互联网20W年薪学习秘笈分享

- 互联网学习秘笈
- 掌握学习秘笈提高效率巩固知识