JVM 调优

主要内容

JVM 介绍

JAVA 字节码文件结构

类加载机制

JVM 内存模型

JVM 垃圾回收机制

JVM 常用命令和常用工具

JVM 调优技巧

JVM 面试技巧

学习目标

知识点	要求
JVM 介绍	了解
JAVA 字节码文件结构	了解
类加载机制	掌握
JVM 内存模型	掌握
JVM 垃圾回收机制	掌握
JVM 常用命令和常用工具	掌握
JVM 调优技巧	掌握
JVM 面试技巧	掌握

一、 为什么要学习 JVM 调优?

- 面试 :随着互联网门槛越来越高,JVM 知识成为中高级程序员阶段必问的一个话题。
- 调优：网站规模逐渐扩大时，可以从底层对项目进行性能调优。
- 可以更好的排查生产环境的问题，更深入理解 JAVA 语言。
- 可以知道你的头发是怎么没有的。

二、 JVM 介绍

1 什么是 JVM

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写。一台执行 Java 程序的机器。

2 JAVA 语言的执行原理

计算机语言：

计算机能够直接执行的指令。这种指令和系统及硬件有关。

计算机高级语言：

在遵循语法的前提下，写一个文本文件，之后利用某种方式，把文本转换为计算机指令执行。

A. 编译型语言(C 语言）:文本文件(.c ） --> 编译器 --> 可执行文件(.exe) --> 执行机器指令。特点：运行速度快，但不能跨平台

B. 解释型语言（JavaScript）：文本文件 --> 解释器 --> 翻译成机器指令并执行。特点：运行速度较慢，但能跨平台

JAVA 语言：先编译，后解释执行

文本文件(java) --> 编译器 --> class 文件(虚拟指令) --> JAVA 虚拟机(JVM)--> 解释为指令执行。

3 JDK + JRE + JVM

（1）JDK（JAVA 开发环境）：JRE + 工具（编译器、调试器、其他工具...）+ 类库

编译器：将 JAVA 文件编译为 JVM 能够看懂的文件（Class 文件）。

（2）JRE（JAVA 运行环境）：JVM + JAVA 解释器

Java 解释器：将虚拟指令解释为机器指令执行。

（3）JVM（JAVA 虚拟机）

4 JAVA 字节码文件结构

```
ClassFile {
    u4              magic;
    u2              minor_version;          副版本号
    u2              major_version;      主版本号
    u2              constant_pool_count;
    cp_info         constant_pool[constant_pool_count-1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
```

```
    method_info      methods[methods_count];
    u2               attributes_count;
    attribute_info   attributes[attributes_count];
}
```

什么是 u2，u4？

u2：代表数据占两个字节　　u4：代表数据占四个字节

JDK 编译对应的版本号：

JDK7 --> 51 JDK8 --> 52　JDK9 -->53 ...　JDK15 --> 59

结论：

编译的本质就是将 java 源文件转为 JVM 能够认识的 16 进制 class 文件格式

三、 类加载机制

1 类加载过程

1.1 装载

（1）获取类的全限定类名，把 class 文件转为二进制流

（2）将二进制流中类的描述信息存入方法区中。如：创建时间、版本等...

（3）将 java.lang.Class 对象存入堆中。

1.2 链接

（1）验证：验证被加载类的正确性：如文件的格式，元数据等。

（2）准备：在方法区中为静态变量分配空间，并设置初始值。

（3）解析：把类的符号引用转为直接引用。

符号引用：class 文件定义的内容

直接引用：JAVA 进程中真实的地址

1.3 初始化

为类的静态变量设置默认值、执行静态代码块。

2 类加载器

2.1 分类

不同的类加载器加载不同的类：

启动类加载器(Bootstrap classLoader)：主要负责加载 JAVA 中的 一些核心类库，主要是位于<JAVA_HOME>/lib/rt.jar 中。

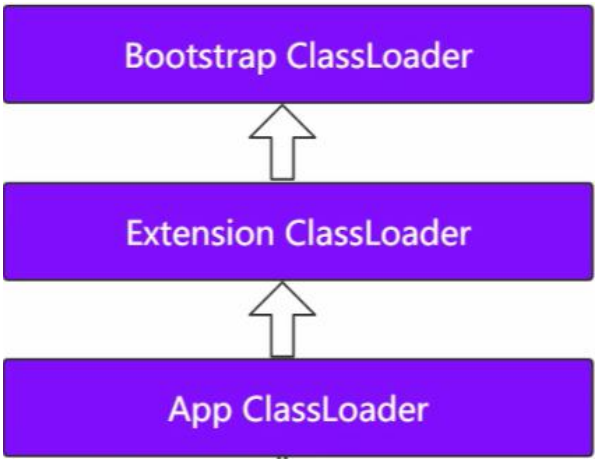
拓展类加载器(Extension classLoader):主要加载 JAVA 中的一些拓展类，位于<JAVA_HOME>/lib/ext 中,是启动类加载器的子类。

应用类加载器(System classLoader): 主要用于加载 CLASSPATH 路径下我们自己写的类，是拓展类加载器的子类

2.2 双亲委派模型

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，

请求最终将到达顶层的启动类加载器。如果父类加载器可以完成类加载任务，就成功返回。倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载。



面试题：如何打破双亲委派机制？ tomcat

自定义类加载器类，继承 ClassLoader 类，重写 loadClass 方法

四、 JVM 内存模型

1 什么是 JVM 内存模型

JVM 需要使用计算机的内存，Java 程序运行中所处理的对象或者算法都会使用 JVM 的内存空间，JVM 将内存区划分为 5 块，这样的结构称之为 JVM 内存模型。

2 JVM 为什么进行内存区域划分

随着对象数量的增加，JVM 内存使用率也在增加，如果 JVM 内存使用率达到 100%，则无法继续运行程序。为了让 JVM 内存可以被重复使用，我们需要进行垃圾回收。为了提高垃圾回收的效率，JVM 将内存区域进行了划分。

3 JVM 内存划分

JVM 按照线程是否共享将内存首先分成两大类

线程独享区

只有当前线程能访问数据的区域，线程之间不能共享

线程独享区随线程的创建而创建，随线程的销毁而被回收

线程共享区

所有线程都可以访问的区域，

当线程被销毁的时候，共享区的数据不会立即回收，需要等待达到垃圾回收的阈（yu）值之后才会进行回收。

4 程序计数器

程序计数器会记录当前线程要执行指令的内存地址，只占用一小部分内存区域，只记录一个地址，所以我们认为程序计数器是不会出现内存溢出问题的分区。

5 本地方法栈

Java 中有些代码的实现是依赖于其他非 Java 语言的（C++），本地方法栈存储的是维护非 Java 语句执行过程中产生的数据，一般我们认为本地方法栈不会出现内存的问题。

6 虚拟机栈

6.1 虚拟机栈的作用

存放当前线程中所声明的变量，包括基本数据类型的数据和引用数据类型的引用。

基本数据类型和引用数据类型划分的标准：

- 基本数据类型：

变量在声明的时候，能够确认占用内存的大小。

- 引用数据类型：

变量在声明的时候，不能确认占用内存的大小。

引用数据类型将值的引用存放到虚拟机栈中，而对象存放在堆内存中，引用数据类型占用 4 个字节存放地址。

6.2 栈帧

每一个线程都会对应一个虚拟机栈，线程中的每个方法都会创建一个栈帧，存放本次方法执行过程中所需要的所有数据。

如果我们一个线程中有多个方法的嵌套调用，虚拟机栈会对栈帧进行压栈和出栈操作。正在执行的方法一定在栈顶，我们只能获取栈顶的栈帧，栈帧在虚拟机栈中先进后出。

6.3 栈帧的数据结构

局部变量

存放当前方法的局部变量，基本数据类型存值，引用数据类型存堆内存地址。

操作数栈

对方法中的变量提供计算的区域。

常量数据的引用

常量数据会存放到方法区的常量池中 ,不管是基本数据类型还是引用数据类型都会存放常量池的地址

方法返回值的地址

方法返回数据会存到计算机内存的寄存器中。

6.4 虚拟机栈溢出异常

由于栈帧调用的深度太深，会出现虚拟机栈溢出异常（SOF 异常）。一般手动方法的调用是不会出现这个异常的，如果出现这个异常 ，99%是由于递归。

可以通过修改虚拟机栈的内存大小设置栈帧的最大深度，指令为：

-Xss 虚拟机栈内存大小

一般栈帧深度达到 3000~5000 即可

太小：虚拟机栈容易溢出。

太大：每个线程占据的内存过大，影响线程数量。

7 方法区

在 java8 之后，我们把方法区称之为元空间（MetaSpace），方法区在逻辑上属于堆的一部分，但一些具体机制和堆有所区别，如：一些 JVM 的方法区是可以不进行垃圾回收的，关闭 JVM 时才会释放方法区内存。所以方法区还有一个别名叫非堆，目的是和堆分开。

方法区会存储类信息、静态变量、常量（JDK8 之后不存放字符串常量）、本地机器指令。

如果加载大量 class 文件，也会造成方法区内存溢出，如一个 tomcat 运行 20~30 个项目。

五、 JVM 执行引擎

1 什么是 JVM 执行引擎

执行引擎是 Java 虚拟机核心的组成部分之一。JVM 的将字节码装载到内存，但字节码并不能够直接运行在操作系统之上。为了执行内存中的字节码文件指令,执行引擎(Execution Engine)就要将**字节码指令**解释/编译为对应平台上的**本地机器**指令。

执行引擎的翻译过程有两种：1、通过解释器将字节码文件转为机器指令执行；2、使

用即时编译器(JIT)将字节码文件的二进制流编译成机器指令执行。

目前市面的主流 JVM 采用解释器与即时编译器并存的架构。在 Java 虚拟机运行时，**解释器**和**即时编译器**相互协作，取长补短。在今天，Java 程序的运行性能早已脱胎换骨,已经达到了可以和 C/C++程序一较高下的地步。

2 解释器与即时编译器

解释器每次解释都会将字节码文件解释为机器指令。整体效率较低，但当程序启动后,解释器可以马上发挥作用，省去编译的时间，立即执行。

即时编译器则会将字节码文件编译为机器指令，存在方法区中，编译完成后直接执行本地机器指令即可。编译器把代码编译成本地代码需要一定的执行时间，但编译为本地代码后执行效率高。

当 Java 虚拟器启动时，解释器首先发挥作用，不必等待即时编译器全部编译完成后再执行。随着时间的推移，编译器把越来越多的代码编译成本地代码，此时运行本地机器指令，获得更高的执行效率。

六、 堆内存模型

JVM 将对象存放在堆内存中，堆内存所需要的空间是比较大的。我们对于 JVM 的调优也主要是针对堆内存的调优，比如分配堆内存的空间，那么我们如何能确定堆内存需要分配多少空间呢？我们需要大概计算每个对象所占的空间大小。

1 JAVA 对象内存布局

JAVA 对象在内存中主要有以下几部分：

对象头

MarkWord：一系列标记位（哈希码、分代年龄、锁状态标记等），在 64 位系统中占 8 字节。

ClassPoint：对象对应的类信息的内存地址，在 64 位系统中占 8 字节。

Length：数组对象特有，表示数组长度，占 4 字节。

实例数据

包含了对象的所有成员变量，大小由变量类型决定。

byte、boolean：1 字节

short：2 字节

char：2~3 字节

int、float：4 字节

long、double、引用数据类型：8 字节

对其填充

将对象大小填充为 8 字节的整数倍

2 JVM 内存溢出和垃圾回收机制

为什么要进行垃圾回收：

如果对象只创建不回收，会造成堆内存溢出(OOM)异常。

为什么要进行堆内存分区：

1. 提高搜索垃圾的效率。
2. 垃圾回收后可以更好的利用内存空间，存放大对象。
3. 尽可能减少 GC 次数。

3 JVM 堆内存的划分

老年代：

对象会优先分配到新生代内存中，每次 GC 后没有回收的对象年龄加 1，年龄到 15 还没有被回收，对象会存放到老年代内存中；如果对象较大，超过新生代内存的一半，对象也会存放到老年代区域。

新生代：

为了减少 young 区垃圾回收后的空间碎片，新生代又分为 Eden 区和两个 Survivor 区，且始终有一个 Suvivor 区保持闲置。对象会先存放到 Eden 区当中，Eden 区空间满了之后会进行 young 区的垃圾回收，之后将 young 区所有存活的对象复制到闲置的 Suvivor 区中，并清空 Eden 区和正在使用的 Survivor 区。

4 YoungGC 和 OldGC

YoungGC

新生代区域的垃圾回收称之为 YoungGC，也叫 MinorGC，Eden 区满后会触发 YoungGC

OldGC

老年代区域的垃圾回收称之为 OldGC，也叫 MajorGC，OldGC 非常浪费性能，所以我们的 JVM 调优要尽可能减少 OldGC 的次数， OldGC 往往伴随着 YoungGC。YoungGC+OldGC = FullGC

问题：

1. Survivor 区空间并不大，如果满了怎么办？

（1）一般情况下 GC 会回收 95%的对象，且超过 15 次 GC 的对象会存放到 old 去，所以 Survivor 区不容易满。

（2）如果 Survivor 区满了，会触发担保机制，提前将对象存入 Old 区。

2. 为什么需要 Survivor 区？

为了减少垃圾回收带来的空间碎片，空间碎片过多会频繁触发 YoungGC。

3. 为什么需要两块 Survivor 区？

为了减少 Survivor 区的空间碎片。

5 JVM 运行监控工具 VisualVM

打开 JAVA 安装目录/bin/jvisualvm.exe，安装 VisualGC 插件。

注：JDK9 之后需自行下载该工具

修改堆内存大小的指令

```
-Xms10M -Xmx10M
```

七、 垃圾回收机制

堆内存模型设计是为了提高垃圾回收的效率，那么如何判断一个对象是垃圾？

1 如何判断一个对象是垃圾

引用计数法：

如果要操作对象，必须通过引用来进行。如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到。那么这个对象就成为可被回收的对象了。这种方式实现简单，效率较高，但是它无法解决循环引用的问题，因此在 Java 中并没有采用这种方式（Python 采用的是引用计数法）。

可达性分析：

以一个 GC Root 对象作为起点进行搜索，如果在 GC Roots 和对象之间没有可达路径，则称该对象是不可达的。

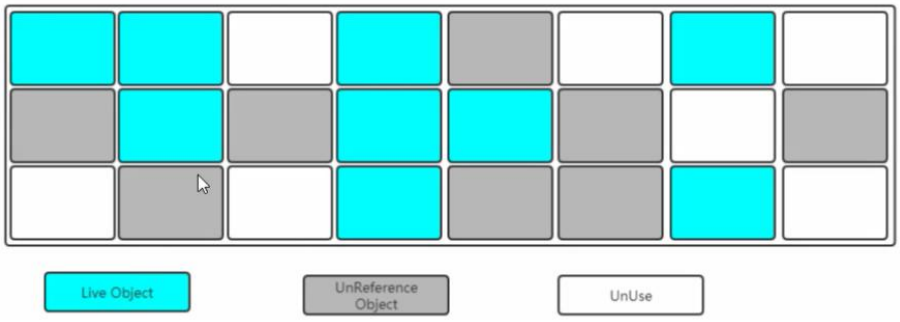
GC ROOT 对象：

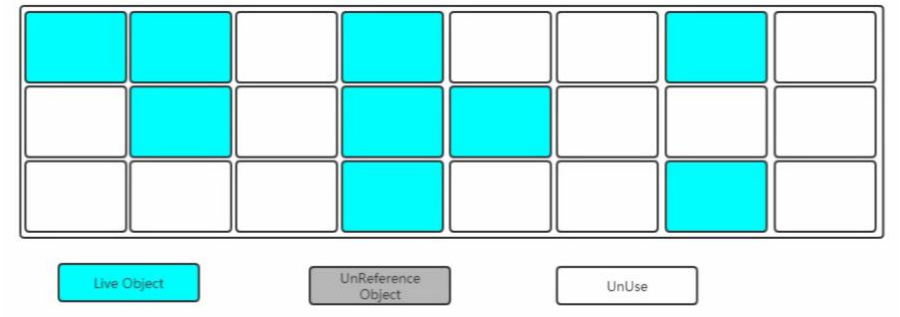
- 栈帧中的本地变量表中引用的对象。

- 方法区中静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中引用的对象。

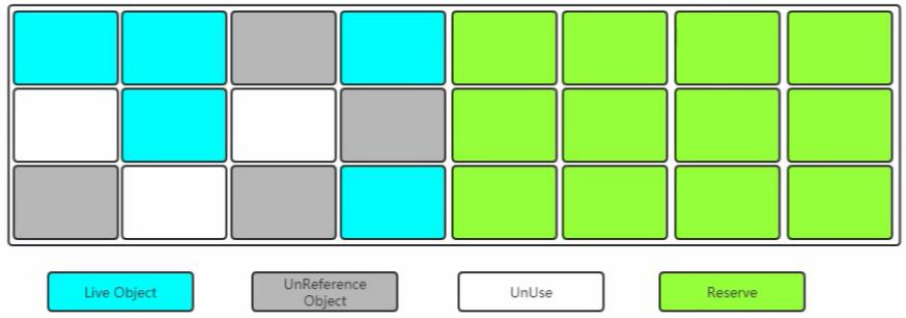
2 垃圾回收算法

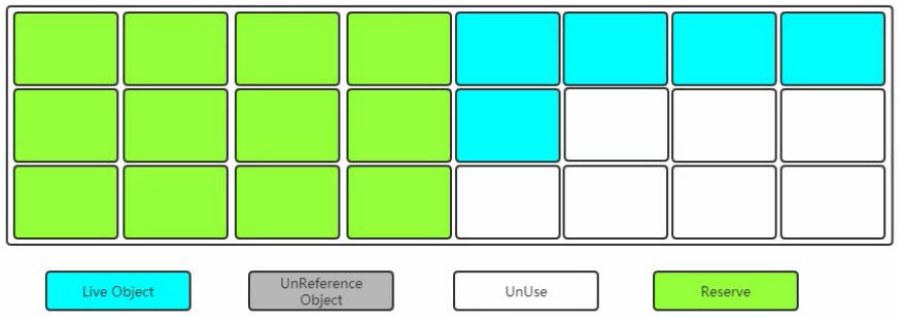
（1）标记——清除算法：效率较低，有空间碎片。Old 区使用的算法



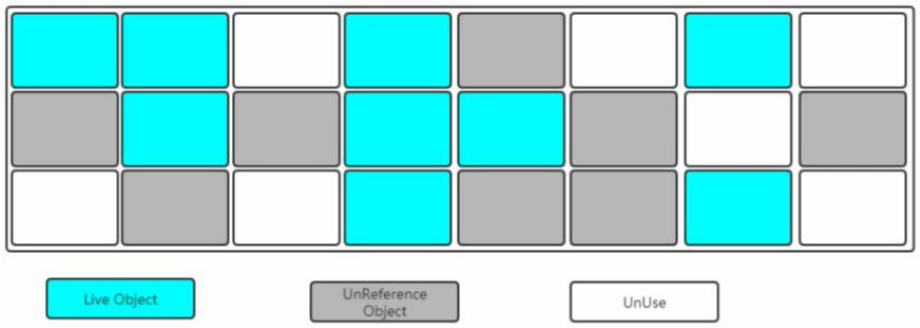


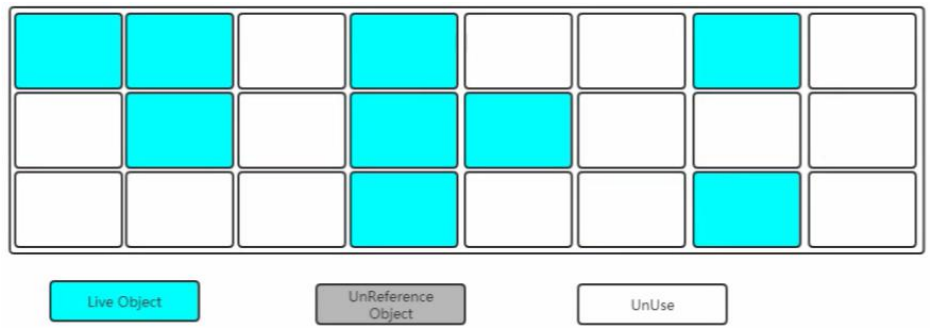
（2）复制算法：空间碎片少，但会浪费空间。存活对象较少才会使用的算法。young 区使用的算法。

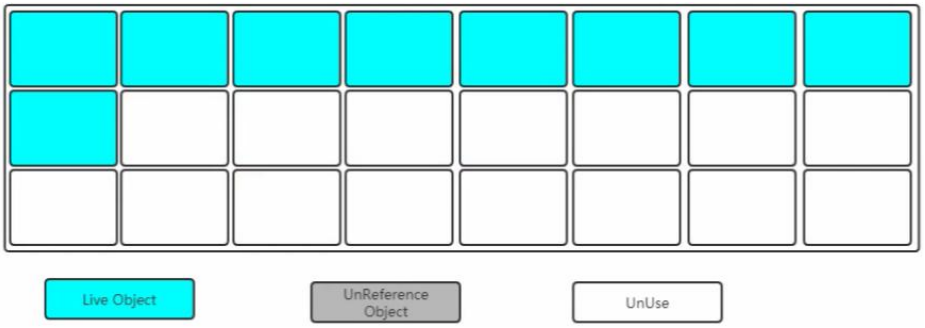




（3）标记——整理算法：空间碎片少，效率较低。Old 区使用的算法。







3 垃圾收集器的评判标准

垃圾收集器是对垃圾回收算法的实现，JVM 中提供了很多垃圾收集器，我们如何评判一个垃圾收集器的好坏呢？

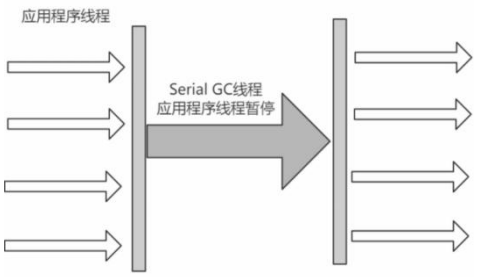
垃圾收集器的执行效率 = 吞吐量 / 停顿时间

吞吐量 = 用户代码执行时间/(用户代码执行时间+停顿时间)

4 垃圾收集器的类型

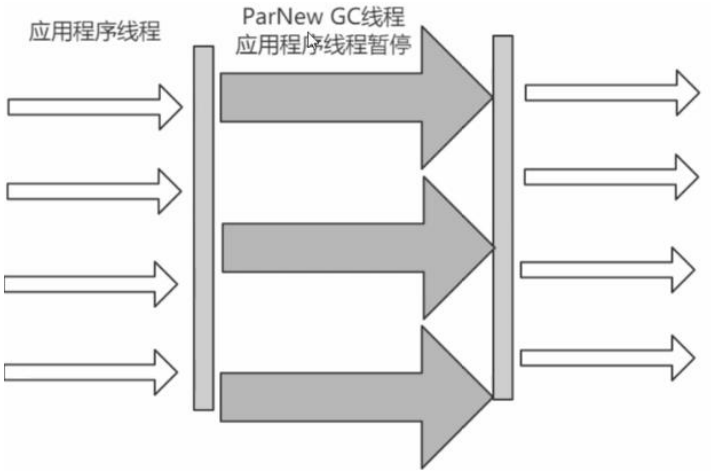
- **串行收集器：**

  只有一个垃圾回收线程，在垃圾回收时暂停用户代码线程，如 Serial 和 Serial Old 收集器。



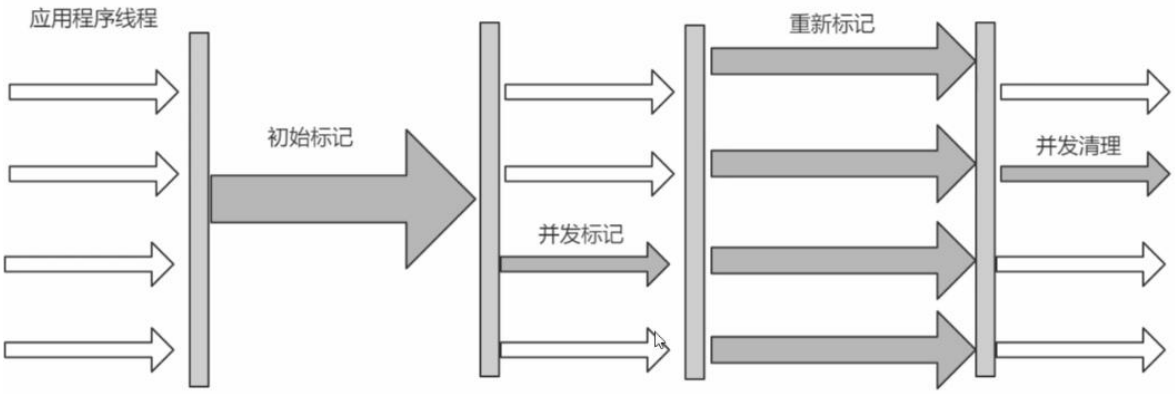
- **并行收集器**

吞吐量优先，多个垃圾收集器线程共同工作，尽快完成垃圾收集。如 ParNew，Parallel Scanvenge, Parallel Old 收集器。



- **并发收集器**

停顿时间优先，用户线程和垃圾回收线程一同工作，用户代码线程也会完全停止一小段时间，如 CMS，G1 收集器。



5 CMS 收集器

CMS（concurrent mark sweep，并发标记扫描）收集器是并发收集器，是基于标记——清理的算法进行垃圾回收，用于 OldGC。

优点：并发收集、低停顿

缺点：会产生大量空间碎片，停顿时间虽然短但是不可控。

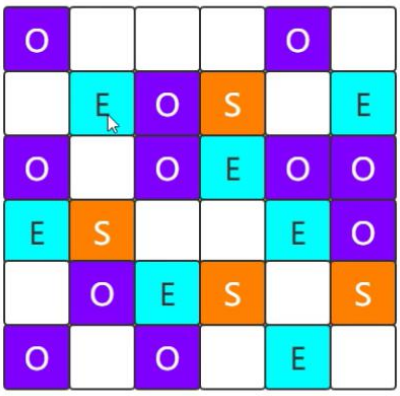
问：CMS 收集器为什么不进行并发的初始标记？

因为初始标记速度很快，不值得多开线程，开线程也是需要耗费资源的。

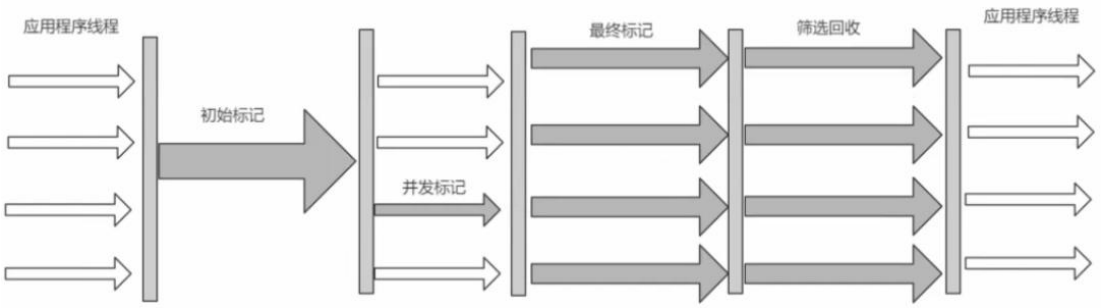
6 G1 收集器

G1（garbage first，垃圾优先）收集器是并发收集器，从 JDK1.7 开始支持，能进行 OldGC 和 YoungGC。Old 区采用标记整理算法，Young 区采用复制算法。

G1 收集器没有固定的 Old、Young、Eden、Survivor 区，而是将内存分为一个个大小相等的 Region（格子，1Mb~32Mb)。每次垃圾回收后，Region 的用途可以发生改变，提高了内存的灵活性和利用率。



G1 收集器可以根据开发者设置的参数，停顿时间的期望值，优先筛选回收存活的对象比较少，垃圾对象比较大的区域 Region，可以把更多空余的空间释放出来。



7 ZGC 收集器

ZGC 从 JDK11 开始支持，目前还是一个实验性版本，原理类似 G1。是目前收集效率最高的垃圾收集器，平均暂停时间为 0.05 毫秒。

8 如何选择垃圾收集器？

- 优先让服务器自己来选择
- 如果内存小于 100M，使用串行收集器
- 如果服务器是单核，并且没有停顿时间要求，使用串行收集器
- 如果允许停顿时间超过 1 秒，选择并行收集器
- 如果停顿时间不能超过 1 秒，使用并发收集器

八、 JVM 参数设置

1 JVM 参数设置方式

Intellij idea：在运行设置的 VM Option 中设置。

tomcat：进入 Tomcat 的 bin 目录下，打开文件 catalina.bat/catalina.sh，修改如下参数。

```
set "JAVA_OPTS=参数"
```

2 JVM 参数类型

（1）标准参数：不随 jdk 版本的变化而变化的参数，如：-version

（2）-X 参数：不能保证所有的 JVM 都支持。

如：-Xcomp：使用即时编译器执行字节码文件

-Xint：使用解释器执行字节码文件

-Xmixed :混合模式 ,先使用解释器 ,即时编译器编译好后执行机器指令。

（3）-XX 参数：不能保证所有的 JVM 都支持。

A. Boolean 类型参数：

-XX:+UseG1GC：使用 G1 收集器

-XX:-UseG1GC：不使用 G1 收集器

B. Key-Value 类型参数：

-XX:MaxTenuringThreshold=15：对象年龄达到 15 就会进入老年代

3 常用参数

参数	含义	说明
-XX:CICompilerCount=3	最大并行编译数	如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加JVM崩溃的可能
-XX:InitialHeapSize=100M	初始化堆大小	简写-Xms100M
-XX:MaxHeapSize=100M	最大堆大小	简写-Xmx100M
-XX:NewSize=20M	设置年轻代的大小	
-XX:MaxNewSize=50M	年轻代最大大小	
-XX:OldSize=50M	设置老年代大小	
-XX:MetaspaceSize=50M	设置方法区大小	
-XX:MaxMetaspaceSize=50M	方法区最大大小	
-XX:+UseParallelGC	使用UseParallelGC	新生代，吞吐量优先
-XX:+UseParallelOldGC	使用UseParallelOldGC	老年代，吞吐量优先
-XX:+UseConcMarkSweepGC	使用CMS	老年代，停顿时间优先
-XX:+UseG1GC	使用G1GC	新生代，老年代，停顿时间优先
-XX:NewRatio	新老生代的比值	比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5
-XX:SurvivorRatio	两个S区和Eden区的比值	比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8，也就是一个S占整个新生代的1/10
-XX:+HeapDumpOnOutOfMemoryError	启动堆内存溢出打印	当JVM堆内存发生溢出时，也就是OOM，自动生成dump文件
-XX:HeapDumpPath=heap.hprof	指定堆内存溢出打印目录	表示在当前目录生成一个heap.hprof文件
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps Xloggc:$CATALINA_HOME/logs/gc.log	打印出GC日志	可以使用不同的垃圾收集器，对比查看GC情况
-Xss128k	设置每个线程的堆栈大小	经验值是3000-5000最佳
-XX:MaxTenuringThreshold=6	提升年老代的最大临界值	默认值为 15
-XX:InitiatingHeapOccupancyPercent	启动并发GC周期时堆内存使用占比	G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示"一直执行GC循环". 默认值为 45.
-XX:G1HeapWastePercent	允许的浪费堆空间的占比	默认是10%，如果并发标记可回收的空间小于10%,则不会触发MixedGC。
-XX:MaxGCPauseMillis=200ms	G1最大停顿时间	暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。
-XX:ConcGCThreads=n	并发垃圾收集器使用的线程数量	默认值随JVM运行的平台不同而不同
-XX:G1MixedGCLiveThresholdPercent=65	混合垃圾回收周期中要包括的旧区域设置占用率阈值	默认占用率为 65%
-XX:G1MixedGCCountTarget=8	设置标记周期完成后，对存活数据上限为G1MixedGCLiveThresholdPercent 的旧区域执行混合垃圾回收的目标次数	默认8次混合垃圾回收，混合回收的目标是要控制在此目标次数以内
-XX:G1OldCSetRegionThresholdPercent=1	描述Mixed GC时，Old Region被加入到CSet中	默认情况下，G1只把10%的Old Region加入到CSet中

九、 JVM 常用命令和常用工具

1 JVM 常用命令

jps：查看当前执行的所有 JAVA 进程

jinfo：实时查看 JVM 参数

jinfo -flag InitialHeapSize PID：JAVA 进程堆内存大小

jinfo -flag UseG1GC PID：JAVA 进程是否使用 G1GC

jinfo -flag UseParallelGC PID：JAVA 进程是否使用 ParallelGC

jstat：虚拟机性能信息

jstat -class PID 1000：每秒查看一次虚拟机中类加载信息

jmap：打印快照

jmap -heap PID：查看堆存储快照

jmap -dump:format=b,file=heap.hprof PID：在出现内存溢出异常时，将堆内存的信息下载到文件中

2 JVM 常用工具

一般情况下，我们会让项目在发生 OOM 异常时自动下载堆内存信息，进行错误的排查。此时我们需要给 JVM 配置如下代码：

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof
```

JVM 堆内存文件查看工具：

MemoryAnalyzer（Mat）、PerfMa

如果我们要进行垃圾回收调优，首先需要将 GC 信息打印出来，此时我们需要给 JVM 配置如下代码：

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
-Xloggc:gc.log
```

垃圾收集器 log 文件查看工具：

GCViewer

运行 GCViewer：输入命令 java -jar gcviewer-1.36-SNAPSHOT.jar

JVM 监控工具：

JVisualVM、Jconsole

十、 垃圾收集器调优

1 对比各个垃圾收集器的指标

-XX:+UseConcMarkSweepGC：使用 CMSGC -XX:+UseG1GC：使用 G1GC

名称	吞吐量	平均停顿时间	GC 次数
ParallelGC	93.25%	25ms	33
CMSGC	98.37%	7.9ms	24
G1GC	98.83%	7.3ms	26

2 G1GC 调优

没有调优时的数据:

吞吐量	平均停顿时间	GC 次数
98.83%	7.3ms	26

内存增加后的数据：

吞吐量	平均停顿时间	GC 次数

99.3%	10.5ms	5

停顿时间变小后的数据

设置停顿时间：-XX:MaxGCPauseMillis=5

吞吐量	平均停顿时间	GC 次数
99.09%	9.1ms	11

3 G1GC 调优指南

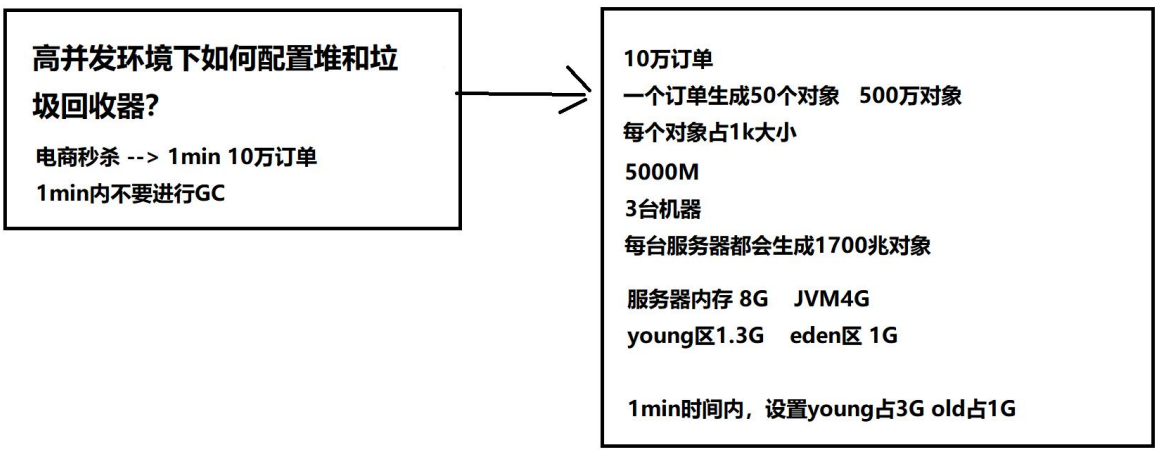
- **不要手动设置新生代和老年代的大小，只设置堆的大小。**
- **不断调优暂停时间目标**

一般情况设置到 100ms 或者 200ms 都是可以的，但如果设置成 50ms 就不太合理。暂停时间太短，会导致 GC 跟不上垃圾产生的速度。

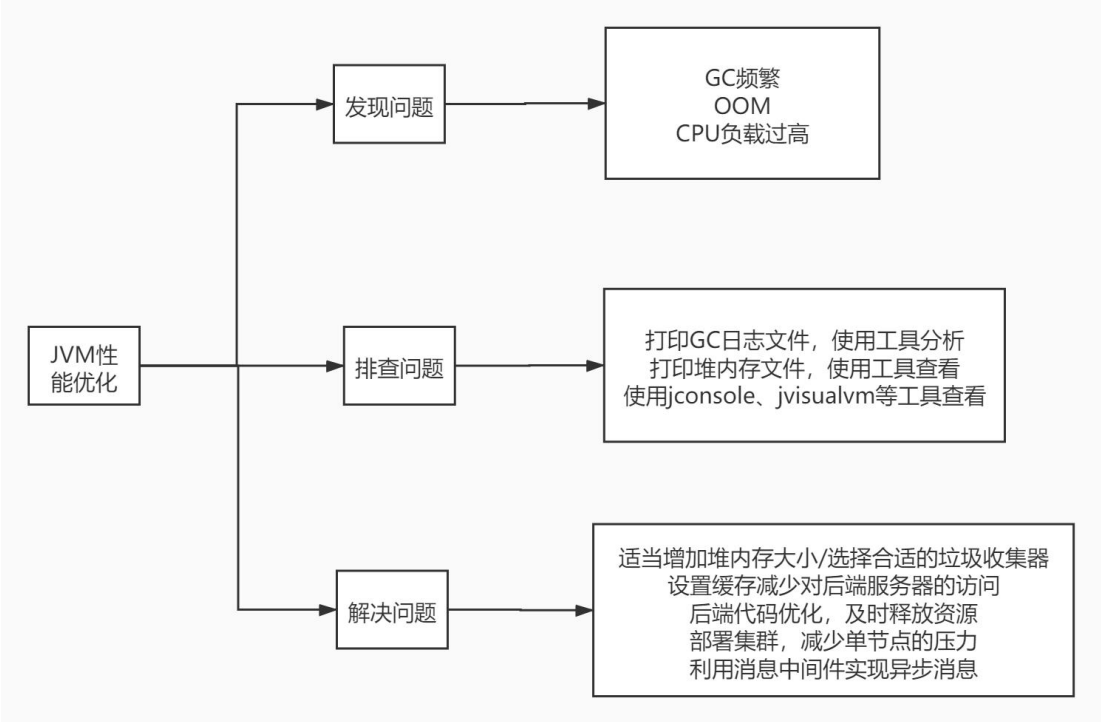
- **适当增加堆内存大小**

十一、 JVM 调优思路

1 高并发环境下如何配置堆和垃圾回收器？



2 生产环境 JVM 问题的排查



3 常见面试题补充

内存泄漏和内存溢出是一样的概念吗?

不一样，内存泄漏指不再使用的对象无法得到及时的回收，持续占用内存空间，造成内存空间的浪费。内存溢出指程序运行要用到的内存大于能提供的最大内存。内存无法内存泄漏很容易导致内存溢出，内存溢出不一定是内存泄漏导致的。

GC Root 不可达的对象一定会被回收吗？

不一定，不可达的对象也不是非死不可的，G1 收集器最终就会筛选要回收的对象。

方法区中的类会被回收吗?

有可能。需要满足以下三个条件：（1）该类的所有的实例都已经被回收了。（2）该类的 ClassLoader 已经被回收了。（3）java.lang.class 对象没有任何地方使用。

CMS 和 G1 的区别

cms 只能适用于老年代，g1 可以用于老年代和新生代。

cms 使用了标记——清除算法，会产生大量碎片，g1 使用标记——整理算法，减少了碎片的产生。

g1 更加灵活，它将内存分为了一块块 region，并且停顿时间可控。