JVM体系结构 & 内存管理机制

TM技术支持五部 徐文卓

资料整理
火龙果软件
荐

S1

印度国王哈里什和独生子打猎途经一个城镇，空地上有三个泥瓦匠正在工作。哈里什国王问那几个匠人在做什么。

第一个人粗暴地说：“我在垒砖头。”

第二个人有气无力地说：“我在砌一堵墙。”

但第三个泥瓦匠热情洋溢、充满自豪地回答说：“我在建一座宏伟的寺庙。”

哈里什国王回到皇宫后，立刻召见了第三个泥瓦匠，并给了他一个总督的职位。国王的儿子问：“父王，我不明白，为什么你那么欣赏第三个工匠呢？”

“一个人有多成功，最终是由他做事时的**态度**决定的，”哈里什国王回答说：“有智慧的人可以看到事业最后的结果，而不是手头的任务，因为只有这样，才会有克服困难的动力。”

资料整理
火龙果软件
荐

目录

1 基本定义

2 JVM体系结构

3 JVM内存管理

4 内存调优

资料整理
火龙果软件
荐

1、基本定义

资料整理
火龙果软件
荐

JVM、JRE、JDK基本定义

-JVM: Java Virtual Machine 负责执行符合Java规范的Class文件，不同厂家会有不同的实现：IBM J9, Sun Hotspot, JRockit, etc

-JRE: Java Runtime Environment 包括JVM和基础类库(常见的标准API，比如java.util.*, etc)。 一般来说基础类库是那些JCP大牛呕心沥血之作。但有些厂家也会针对自己的硬件修改一部分基础类库的实现。最典型的是IBM的JRE。

-JDK: Java Development Kit 包括JRE和开发/编译工具(javac, javadoc, etc)。

资料整理
火龙果软件
荐

1 JVM生命周期

区别**两个概念**：JVM实例和JVM执行引擎实例

■JVM实例 对应了一个独立运行的java程序，它是**进程**级别

■JVM执行引擎实例 则对应了属于用户运行程序的线程，它是**线程**级别的

每一个Java线程就是一个执行引擎的实例，对应一个JVM实例中同时有多个执行引擎在工作。执行引擎有的执行用户程序，有的执行JVM内部程序（如垃圾回收器）。

资料整理
火龙果软件
荐

1 JVM生命周期

JVM的生命周期

（1）JVM实例的诞生

当启动一个Java程序时，一个JVM实例就产生了，任何一个拥有public static void main(String[] args)函数的class都可以作为JVM实例运行的起点

（2）JVM实例的运行

main()函数作为该程序初始线程的起点，任何其他线程均由该线程启动。JVM内部有两种线程：守护线程和非守护线程，main()属于非守护线程，守护线程通常由JVM自己使用，java程序也可以标明自己创建的线程是守护线程。

（3）JVM实例的消亡

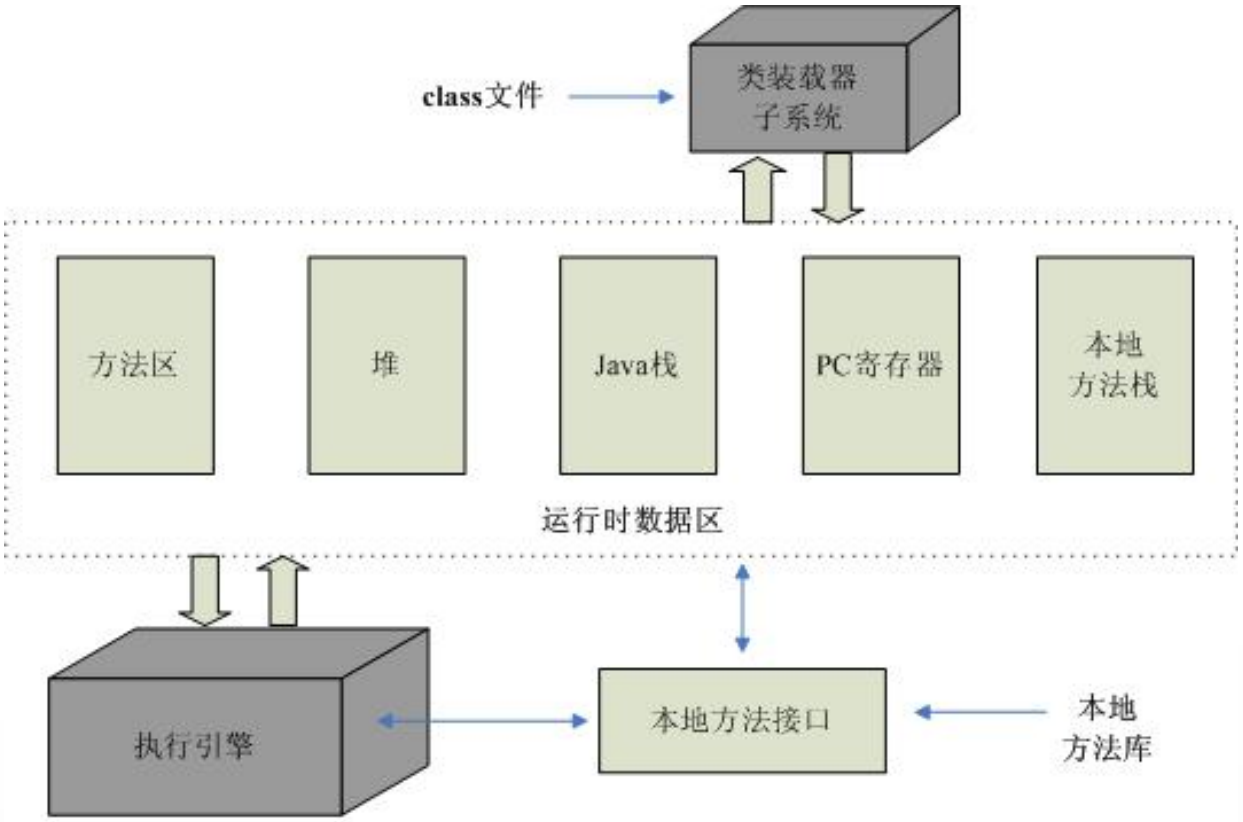
当程序中的所有非守护线程都终止时，JVM才退出；若安全管理器允许，程 序也可以使用Runtime类或者System.exit()来退出。

资料整理
火龙果软件
荐

2、JVM体系结构

资料整理
火龙果软件
荐

2 体系结构



Java虚拟机的内部体系结构

资料整理
火龙果软件
荐

2 体系结构

JVM的内部体系结构分为四部分：

- 类加载器（ClassLoader）

  作用: 用来装载.class文件，JVM启动时或者类运行时将需要的class加载到JVM中

- 执行引擎

  JVM的核心部分，作用:执行字节码指令，或者执行本地方法，相当于CPU

- 运行时数据区（内存区）

  将内存划分为若干个区域，模拟实际机器上的存储、记录和调度模块。方法区，堆，java栈，PC寄存器，本地方法区等

- 本地方法调用

  调用C或者C++实现的本地方法代码，并返回结果

资料整理
火龙果软件
荐

2 体系结构 类加载器

JVM将整个类加载过程划分为了三个步骤：

① 装载

装载过程负责找到二进制字节码并加载至JVM中，JVM通过类名、类所在的包名通过ClassLoader来完成类的加载，同样，也采用以上三个元素来标识一个被加载了的类：类名+包名+ClassLoader实例ID。

② 链接

链接过程负责对二进制字节码的格式进行校验、初始化装载类中的静态变量以及解析类中调用的接口、类。

在完成了校验后，JVM初始化类中的静态变量，并将其值赋为默认值。

最后一步为对类中的所有属性、方法进行验证，以确保其需要调用的属性、方法存在，以及具备应的权限（例如public、private域权限等），会造成NoSuchMethodError、NoSuchFieldError等错误信息。

③ 初始化

初始化过程即为执行类中的静态初始化代码、构造器代码以及静态属性的初始化，在四种情况下初始化过程会被触发执行：

调用了new；反射调用了类中的方法；子类调用了初始化；JVM启动过程中指定的初始化类。

资料整理
火龙果软件
荐

2 体系结构 类加载器

ClassLoader抽象类提供了几个关键的方法：

- loadClass

  此方法负责加载指定名字的类，ClassLoader的实现方法为：先从已经加载的类中寻找，如没有则继续从parent ClassLoader中寻找，如仍然没找到，则从System ClassLoader中寻找，最后再调用findClass方法来寻找，如要改变类的加载顺序，则可覆盖此方法

- findLoadedClass

  此方法负责从当前ClassLoader实例对象的缓存中寻找已加载的类，调用的为native的方法。

- findClass

  此方法直接抛出ClassNotFoundException，因此需要通过覆盖loadClass或此方法来以自定义的方式加载相应的类。

- findSystemClass

  此方法负责从System ClassLoader中寻找类，如未找到，则继续从Bootstrap ClassLoader中寻找，如仍然为找到，则返回null。

- defineClass

  此方法负责将二进制的字节码转换为Class对象

- resolveClass

  此方法负责完成Class对象的链接，如已链接过，则会直接返回。

资料整理
火龙果软件
荐

简单的classLoader例子

```
/*
 * 重写ClassLoader类的findClass方法，将一个字节数组转换为 Class 类的实例
 */
public Class<?> findClass(String name) throws ClassNotFoundException {
    byte[] b = null;
    try {
        b = loadClassData(AutoClassLoader.FormatClassName(name));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return defineClass(name, b, 0, b.length);
}

/*
 * 将指定路径的.class文件转换成字节数组
 */
private byte[] loadClassData(String filepath) throws Exception {
    int n =0;
    BufferedInputStream br = new BufferedInputStream(new FileInputStream(new File(filepath)));
    ByteArrayOutputStream bos= new ByteArrayOutputStream();
        while((n=br.read())!=-1){
            bos.write(n);
        }
        br.close();
    return bos.toByteArray();
}
```

资料整理
火龙果软件
荐

简单的classLoader例子

```
/*
 * 格式化文件所对应的路径
 */
public static String FormatClassName(String name){

    FILEPATH= DEAFAULTDIR + name+".class";
    return FILEPATH;
}

/*
* main方法测试
*/
public static void main(String[] args) throws Exception {

    AutoClassLoader acl = new AutoClassLoader();
    Class c = acl.findClass("testClass");
    Object obj = c.newInstance();
    Method m = c.getMethod("getName",new Class[]{String.class ,int.class});
    m.invoke(obj,"你好",123);
    System.out.println(c.getName());
    System.out.println(c.getClassLoader());
    System.out.println(c.getClassLoader().getParent());
}
```

资料整理 火龙果软件 荐

2 体系结构 执行引擎

JVM通过执行引擎来完成字节码的执行，在执行过程中JVM采用的是自己的一套指令系统，每个线程在创建后，都会产生一个程序计数器（pc）和Java栈（Stack），其中程序计数器中存放了下一条将要执行的指令，Stack中存放Stack Frame，表示的为当前正在执行的方法，每个方法的执行都会产生Stack Frame。

Stack Frame中存放了传递给方法的参数、方法内的局部变量以及操作数栈，操作数栈用于存放指令运算的中间结果，指令负责从操作数栈中弹出参与运算的操作数，指令执行完毕后再将计算结果压回到操作数栈，当方法执行完毕后则从Stack中弹出，继续其他方法的执行。

在执行方法时JVM提供了invokestatic、invokevirtual、invokeinterface和invokespecial四种指令来执行

（1）invokestatic：调用类的static方法

（2） invokevirtual： 调用对象实例的方法

（3） invokeinterface：将属性定义为接口来进行调用

（4） invokespecial： JVM对于初始化对象（Java构造器的方法为：<init>）以及调用对象实例中的私有方法时。

资料整理
火龙果软件
荐

2 体系结构 执行引擎

反射机制是Java的亮点之一，基于反射可动态调用某对象实例中对应的方法、访问查看对象的属性等，而无需在编写代码时就确定需要创建的对象，这使得Java可以实现很灵活的实现对象的调用，代码：

```
Class actionClass=Class.forName(外部实现类);
Method method=actionClass.getMethod(“execute”,null);
Object action=actionClass.newInstance();
method.invoke(action,null);
```

反射的关键：要实现动态的调用，最明显的方法就是动态的生成字节码，加载到JVM中并执行

资料整理
火龙果软件
荐

2 体系结构 执行引擎

① Class actionClass=Class.forName(外部实现类);

调用本地方法，使用调用者所在的ClassLoader来加载创建出Class对象；

② Method method=actionClass.getMethod(“execute” ,null);

- 校验此Class是否为public类型的，以确定类的执行权限，如不是public类型的，则直接抛SecurityException；
- 调用privateGetDeclaredMethods来获取到此Class中所有的方法，在privateGetDeclaredMethods对此Class中所有的方法的集合做了缓存，在第一次时会调用本地方法去获取；
- 扫描方法集合列表中是否有相同方法名以及参数类型的方法，如有则复制生成一个新的Method对象返回；
- 如没有则继续扫描父类、父接口中是否有此方法，如仍然没找到方法则抛出NoSuchMethodException；

资料整理
火龙果软件
荐

2 体系结构 执行引擎

③ Object action=actionClass.newInstance();

第一步：校验此Class是否为public类型，如权限不足则直接抛出SecurityException；

第二步：如没有缓存的构造器对象，则调用本地方法获取到构造器，并复制生成一个新的构造器对象，放入缓存，如没有空构造器则抛出InstantiationException；

第三步：校验构造器对象的权限；

第四步：执行构造器对象的newInstance方法；构造器对象的newInstance方法判断是否有缓存的ConstructorAccessor对象，如果没有则调用sun.reflect.ReflectionFactory生成新的ConstructorAccessor对象；

第五步：sun.reflect.ReflectionFactory判断是否需要调用本地代码，可通过sun.reflect.noInflation=true来设置为不调用本地代码，在不调用本地代码的情况下，就转交给MethodAccessorGenerator来处理了；

资料整理 火龙果软件 荐

2 体系结构 执行引擎

第六步：MethodAccessorGenerator中的generate方法根据Java Class格式规范生成字节码，字节码中包括了ConstructorAccessor对象需要的newInstance方法，此newInstance方法对应的指令为invokespecial，所需的参数则从外部压入，生成的Constructor类的名字以：sun/reflect/GeneratedSerializationConstructorAccessor或sun/reflect/GeneratedConstructorAccessor开头，后面跟随一个累计创建的对象的次数；

第七步：在生成了字节码后将其加载到当前的ClassLoader中，并实例化，完成ConstructorAccessor对象的创建过程，并将此对象放入构造器对象的缓存中；

最后一步：执行获取的ConstructorAccessor.newInstance，这步和标准的方法调用没有任何区别。

资料整理
火龙果软件
荐

④ method.invoke(action,null);

这步执行的过程和上一步基本类似，只是在生成字节码时生成的方法改为了invoke，其调用的目标改为了传入的对象的方法，同时生成的类名改为了：sun/reflect/GeneratedMethodAccessor。

注：但是getMethod是非常耗性能的，一方面是权限的校验，另外一方面所有方法的扫描以及Method对象的复制，因此在使用反射调用多的系统中应缓存getMethod返回的Method对象

资料整理 火龙果软件 荐

2 体系结构 执行引擎

JVM主要的**执行技术**有:解释、即时编译、自适应优化、芯片级直接执行

（1）**解释**：属于第一代JVM，将每个Java指令都转译成对等的微处理器指令，并根据转译后的指令先后次序依序执行，由于一个Java指令可能被转译成十几或数十几个对等的微处理器指令，这种模式执行的速度相当缓慢。

（2）**即时编译（JIT）**：属于第二代JVM，每遇到一个新的类,JIT编译器在此时就会针对这个类进行编译（compile）作业。经过编译后的程序，被优化成相当精简的二进制，这种程序的执行速度相当快。某些极少执行到的Java指令在编译时所额外花费的时间可能比编译器在执行时的时间还长，针对这些指令而言，整体花费的时间并没有减少。

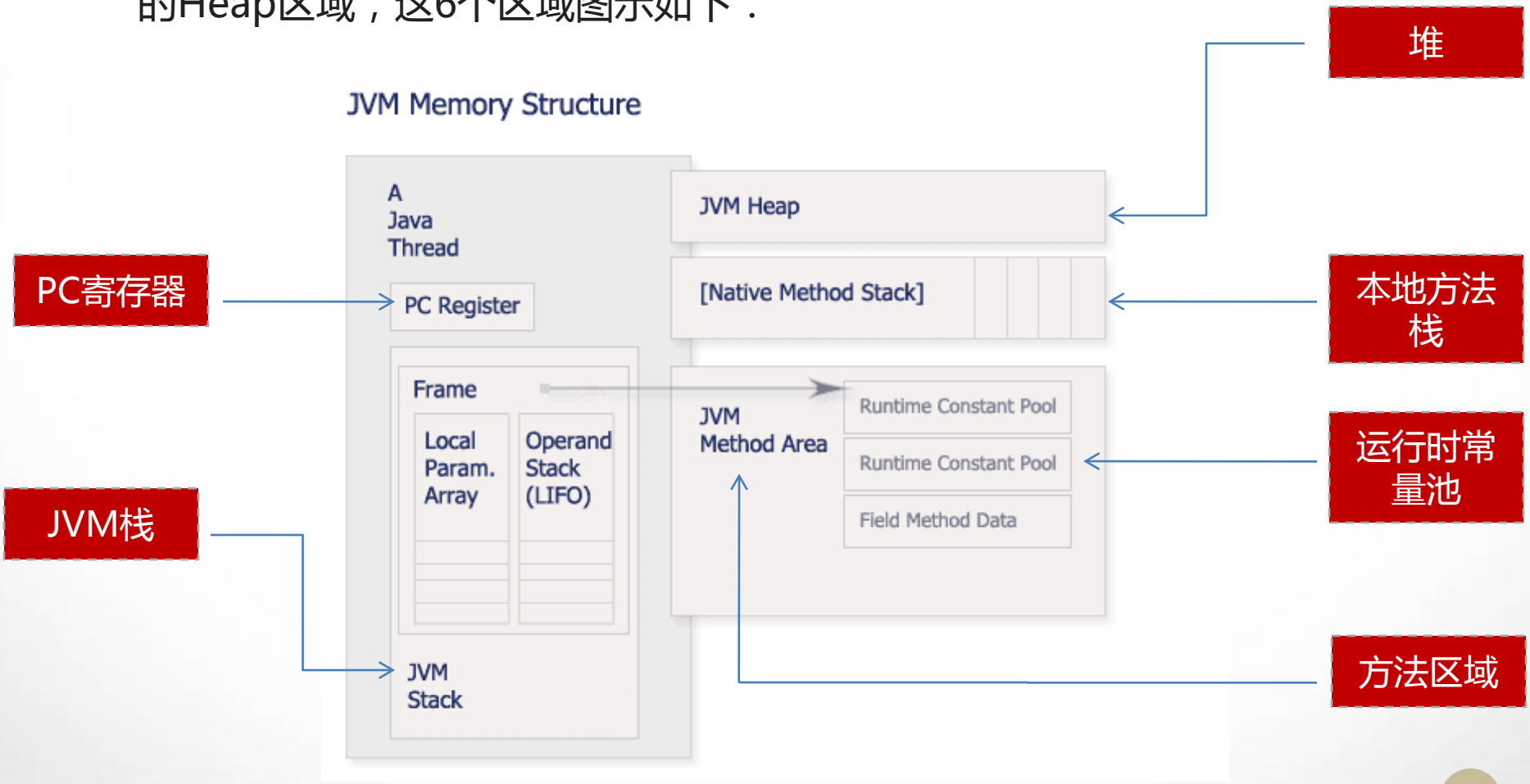
（3）**自适应优化**：（目前Sun的Hotspot JVM采用这种技术）则吸取第一代JVM和第二代JVM的经验，采用两者结合的方式。

开始对所有的代码都采取解释执行的方式，并监视代码执行情况，然后对那些经常调用的方法启动一个后台线程，将其编译为本地代码，并进行仔细优化。若方法不再频繁使用，则取消编译过的代码，仍对其进行解释执行。

资料整理
火龙果软件
荐

2 体系结构 运行时数据区

JVM在运行时将数据划分为了6个区域来存储，而不仅仅是大家熟知的Heap区域，这6个区域图示如下：



资料整理
火龙果软件
荐

2 体系结构 运行时数据区

第一块： PC寄存器

PC寄存器是用于存储每个线程下一步将执行的JVM指令，如该方法为native的，则PC寄存器中不存储任何信息。

第二块：JVM栈

JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈，JVM栈中存放的为当前线程中局部基本类型的变量（java中定义的八种基本类型：boolean、char、byte、short、int、long、float、double）、部分的返回结果以及Stack Frame，非基本类型的对象在JVM栈上仅存放一个指向堆上的地址

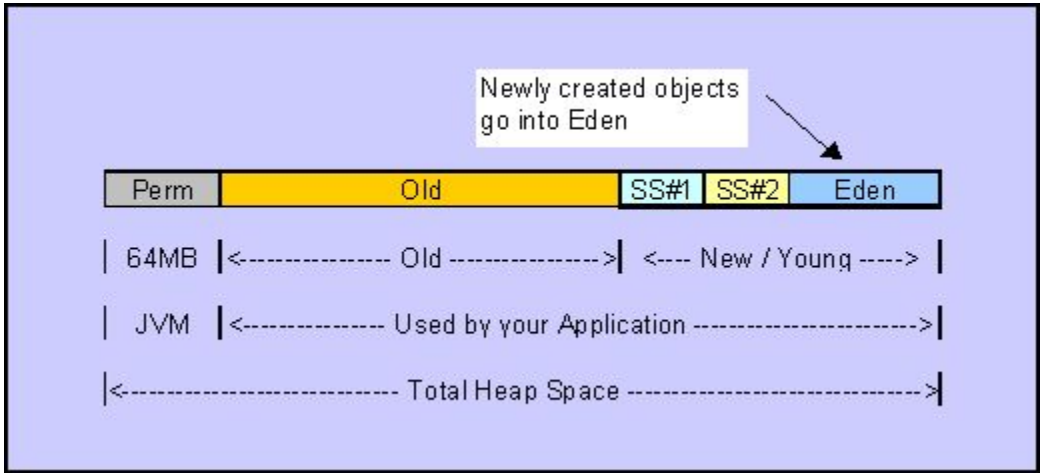
第三块：堆（Heap）

Heap是大家最为熟悉的区域，它是JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收。

资料整理
火龙果软件
荐

2 体系结构 运行时数据区

JVM将Heap分为New Generation和Old Generation（或Tenured Generation）两块来进行管理：



（1）New Generation

又称为新生代，程序中新建的对象都将分配到新生代中，新生代又由Eden Space和两块Survivor Space构成，可通过-Xmn参数来指定其大小

（2） Old Generation

又称为旧生代，用于存放程序中经过几次垃圾回收还存活的对象，例如缓存的对象等，旧生代所占用的内存大小即为-Xmx指定的大小减去-Xmn指定的大小。

资料整理 火龙果软件 荐

堆（Heap）

（1）堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的

（2）鉴于上面的原因，Sun Hotspot JVM为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间，这块空间又称为TLAB（Thread Local Allocation Buffer），其大小由JVM根据运行的情况计算而得，在TLAB上分配对象时不需要加锁，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配

（3）TLAB仅作用于新生代的Eden Space，因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效，

但这种方法同时也带来了两个问题，一是空间的浪费，二是对象内存的回收上仍然没法做到像Stack那么高效，同时也会增加回收时的资源的消耗，可通过在启动参数上增加-XX:+PrintTLAB来查看TLAB这块的使用情况。

资料整理 火龙果软件 荐

2 体系结构 运行时数据区

第四块：方法区域（Method Area）

（1）方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息。

当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，可见方法区域的重要性，同样，方法区域也是全局共享的，在一定的条件下它也会被GC，当方法区域需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

（2）在Sun JDK中这块区域对应的为Permanet Generation，又称为**持久代**，默认为64M，可通过-XX:PermSize以及-XX:MaxPermSize来指定其大小。

第五块：运行时常量池（Runtime Constant Pool）

类似PC中的符号表，存放的为类中的固定的常量信息、方法和Field的引用信息等，其空间从方法区域中分配。

第六块：本地方法堆栈（Native Method Stacks）

JVM采用本地方法堆栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

资料整理 火龙果软件 荐

S2

位于日本南部宫崎县的幸岛是短尾猴的故乡。日本科学家对幸岛短尾猴的研究已有半个世纪之久，研究过程中最著名的发现是猴子也会清洗红薯。科学家将这种行为看作是非人类种群表现出的一种文化现象。

1952 年，日本京都大学的一位教授带着几名学生对短尾猴进行了观察研究，在研究的过程中，他们在沙土里种植了一些红薯，走的时候就把这些红薯留下了。后来，猴子 发现了红薯，就开始作为食物来吃。由于是在沙土里生长的，红薯上经常粘着一些沙子，比较磕牙。后来，有一个聪明的猴子发现，把红薯放到水里洗一下，然后再 吃，就不会磕牙了，于是他高兴地把这个发现告诉了身边的小猴子，这些猴子也开始用水洗红薯吃，再后来，这些猴子又把这个秘密告诉了其他的猴子，甚至告诉了 其他岛上的猴子。于是，一天，一个令人震撼的场景出现了，在皎洁的月光下，100 多只猴子排着队在水里洗红薯，这就像预示着一个新纪元的出现。

资料整理
火龙果软件
荐

3、JVM内存管理

资料整理
火龙果软件
荐

堆(Heap)和非堆(Non-heap)内存

堆：就是Java代码可及的内存，是留给开发人员使用的。

非堆：就是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

资料整理 火龙果软件 荐

堆内存分配

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。因此服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小。

非堆内存分配

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。

资料整理 火龙果软件 荐

3 内存管理 内存分配

JVM基于堆(Heap)和栈(Stack)的内存分配优缺点：

- 栈：Java栈的分配和线程绑定在一起，创建线程时，JVM创建一个新的Java栈。栈中主要存放基本类型的变量（int、short、long等）和对象引用。优点：存取速度较快，数据能够共享。缺点：生存周期固定，缺乏灵活性。

- 堆：Java的堆是一个运行时数据区域，通过new 、newarray、anewarray和multianewarray等建立，由GC负责回收。优势：动态分配内存大小，生存周期不用预先告知编译器，缺点：运行时分配，存储速度较慢

资料整理
火龙果软件
荐

JVM内存限制(最大值)

首先JVM内存限制于实际的最大物理内存，假设物理内存无限大的话，JVM内存的最大值跟操作系统有很大的关系。

简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制，这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，Linux系统下为2G-3G），而64bit以上的处理器就不会有限制了。

资料整理
火龙果软件
荐

3 内存管理 回收算法

几种经典的内存回收算法

- 引用计数算法
- 标记-清除算法（Mark-Sweep）
- 标记-缩并算法（Mark-Compact）
- 节点拷贝算法（Copying）

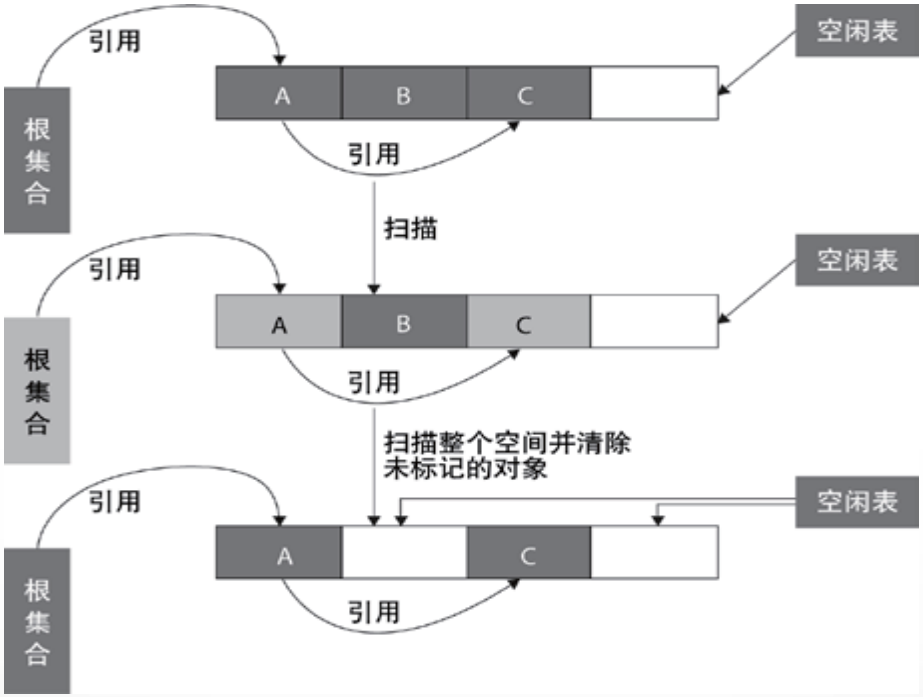
Sun JDK的GC

- 新生代可用的GC——Minor GC
- 旧生代和持久代可用的GC
- Full GC

资料整理
火龙果软件
荐

3 内存管理 回收算法

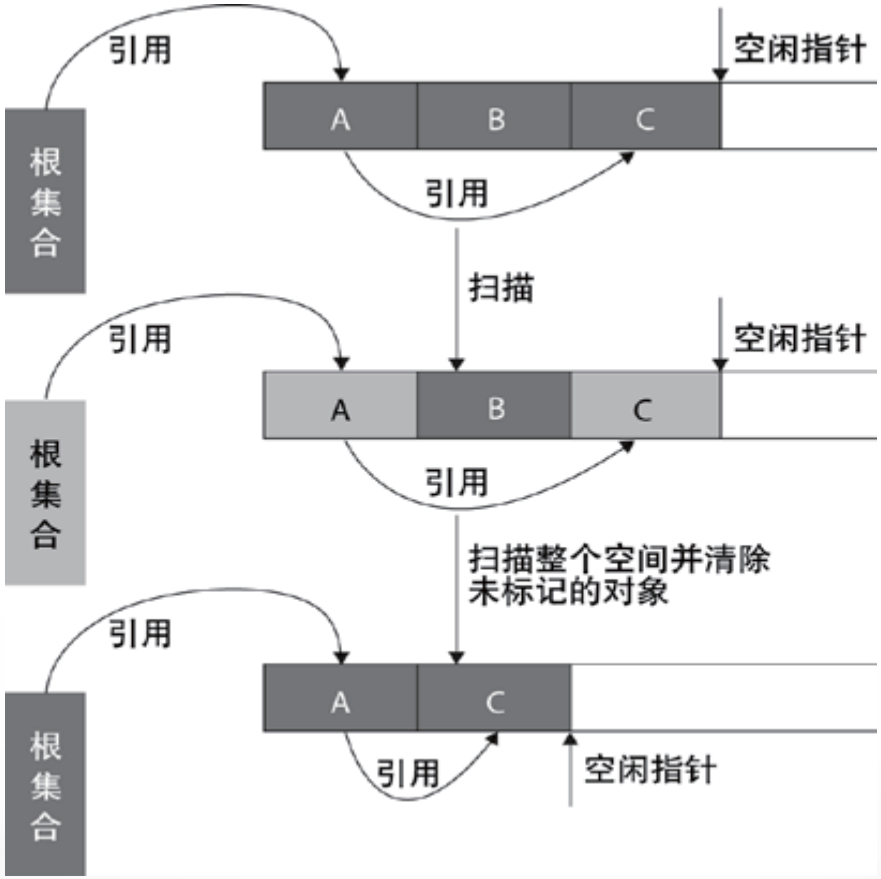
- **引用计数算法**：每个对象计算指向它的指针的数量，当有一个指针指向自己时计数值加1；当删除一个指向自己的指针时，计数值减1，如果计数值减为0，说明已经不存在指向该对象的指针了，所以它可以被安全的销毁了
- **标记-清除算法（Mark-Sweep）**：整个过程分为两个阶段：标记阶段：找到所有存活对象（遍历可达的对象）；清除阶段：清除所有垃圾对象（遍历不可达的对象）。



资料整理
火龙果软件
荐

3 内存管理 回收算法

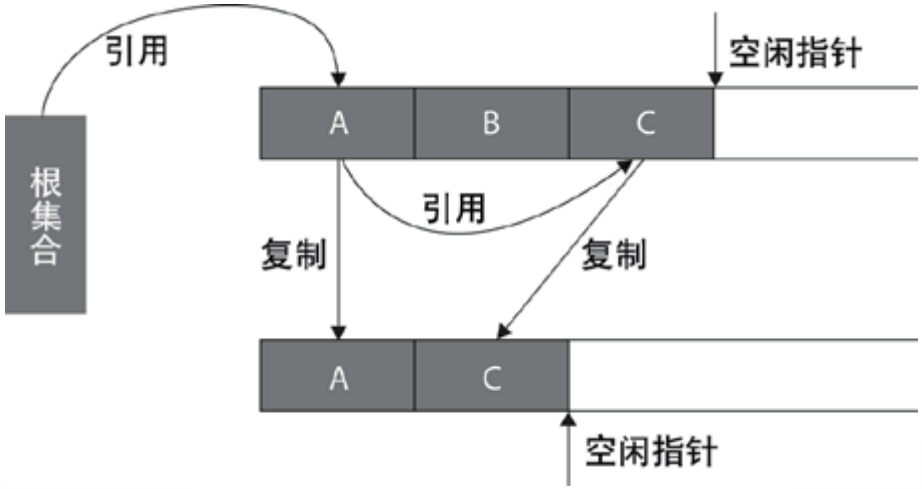
- **标记-缩并算法（Mark-Compact）**：为了解决内存碎片问题而产生的一种算法。 它的整个过程可以描述为：标记所有的存活对象；通过重新调整存活对象位置来缩并对象图；更新指向被移动了位置的对象的指针。



资料整理
火龙果软件
荐

3 内存管理 回收算法

- **节点拷贝算法（Copying）：** 把整个堆分成两个半区（From，To），GC的过程其实就是把存活对象从一个半区From拷贝到另外一个半区To的过程，而在下一次回收时，两个半区再互换角色。在移动结束后，再更新对象的指针引用。



资料整理
火龙果软件
荐

3 内存管理 回收机制

JVM中自动的对象内存回收机制称为：**GC**（Garbage Collection）

GC的基本原理：

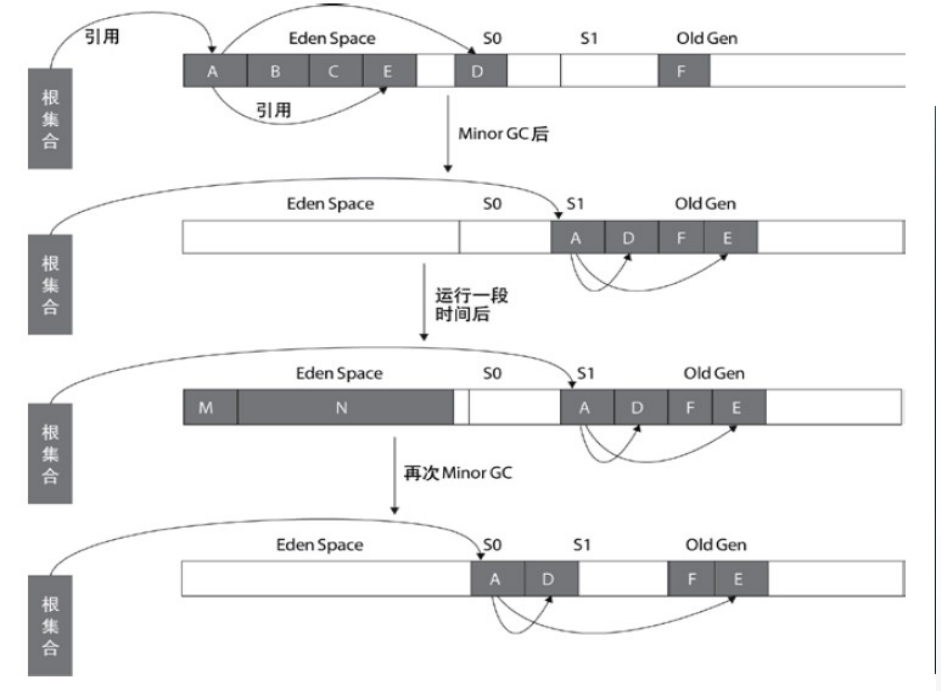
为将内存中不再被使用的对象进行回收，GC中用于回收内存中不被使用的对象的方法称为**收集器**，由于GC需要消耗一些资源和时间的，Java在对对象的生命周期特征进行分析后，在V 1.2以上的版本采用了**分代**的方式来进行对象的收集，即按照新生代、旧生代的方式来对对象进行收集，以尽可能的**缩短GC对应用造成的暂停**

Sun JDK的GC： Sun JDK认为java程序中大部分对象的存活时间都较短，少部分对象长期存活，基于此分析，Sun JDK将JVM堆划为新生代和旧生代，并根据对象存活时间提供了不同的GC实现。

资料整理
火龙果软件
荐

1、新生代可用的GC——Minor GC ：

新生代对象的特点是存活时间较短，基于节点拷贝算法实现对新生代对象的回收。在执行复制时，需要一块未使用的空间来存放存活的对象，这是新生代又被划为Eden、S0和S1的原因。Eden Space存放新创建的对象，S0或S1的其中一块用于GC触发时作为复制的目标空间，当其中一块为复制的目标空间时，另一块中的内容则会被清空。因此通常又将S0、S1称为From Space和To Space。对新生代所占用内存进行的GC又通常被称为Minor GC 。



资料整理 火龙果软件 荐

2、旧生代和持久代可用的GC：

JDK提供了串行、并行及并发三种GC来对旧生代及持久代对象所占内存进行回收

- 串行GC：基于Mark-Sweep-Compact实现，它结合Mark-Sweep和Mark-Compact做了些改进。
- 并行标记压缩GC——并行实现Mark-Compact GC
- 并发标记清除GC——并发实现Mark-Sweep GC（CMS GC）

资料整理 火龙果软件 荐

3、Full GC：

对新生代、旧生代、持久代都进行GC时，称为Full GC，当触发Full GC时，首先按照新生代所配置的GC方式进行GC，然后按照旧生代的GC方式对旧生代、持久代进行GC。

触发Full GC的条件：

- 旧生代空间不足，如果Full GC仍然空间不足，则报错：java.lang.OutOfMemoryError: Java heap space
- Permanet Generation 空间满，如果Full GC仍然空间不足，则报错：java.lang.OutOfMemoryError: PermGen space
- CMS GC时出现promotion failed 和concurrent mode failure，均是因为旧生代空间不足造成
- 统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间
- 内存在RMI调用时，默认每小时调用一次Full GC，可以通过参数配置时间间隔
- 直接调用System.gc，可以DisableExplicitGC来禁止

资料整理 火龙果软件 荐

垃圾收集器的“并行”和“并发”

- 并行（Parallel）：指多个收集器的线程同时工作，但是用户线程处于等待状态
- 并发（Concurrent)：指收集器在工作的同时，可以允许用户线程工作。

并发不代表解决了GC停顿的问题，在关键的步骤还是要停顿。比如在收集器标记垃圾的时候。但在清除垃圾的时候，用户线程可以和GC线程并发执行。

资料整理
火龙果软件
荐

Serial收集器

- 最早的收集器，单线程进行GC
- New和Old Generation都可以使用
- 在新生代，采用复制算法；在老生代，采用Mark-Compact算法
- 因为是单线程GC，没有多线程切换的额外开销，简单实用
- Hotspot Client模式缺省的收集器

资料整理 火龙果软件 荐

CMS收集器(Concurrent Low Pause Collector)

- 追求最短停顿时间，非常适合Web应用
- 只针对老年区，一般结合ParNew使用
- Concurrent，GC线程和用户线程并发工作（尽量并发）
- 只有在多CPU环境下才有意义
- 使用-XX:+UseConcMarkSweepGC打开

资料整理 火龙果软件 荐

3 内存管理 GC回收器

此外还有：

Serial收集器

ParNew收集器

G1收集器(Garbage First)

Parallel Old

Parallel Scavenge

……

资料整理
火龙果软件
荐

3 内存管理 GC实现

Sun JDK为高效实现内存的分配及内存的回收，提供了多种GC的实现，在client和server模式下的**默认选择并不相同**。

client、server模式的默认GC

	新生代GC方式	旧生代和持久代GC方式
Client	串行GC	串行GC
Server	并行回收GC	并行GC（JDK 5.0 Update 6以后）

Sun JDK GC的组合方式

	新生代GC方式	旧生代和持久代GC方式
-XX:+UseSerialGC	串行GC	串行GC
-XX:+UseParallelGC	并行回收GC	并行GC
-XX:+UseConcMarkSweepGC	并行GC	并发GC 当出现concurrent Mode failure时采用串行GC
-XX:+UseParNewGC	并行GC	串行GC
-XX:+UseParallelOldGC	并行回收GC	并行GC
-XX:+UseConcMarkSweepGC -XX:-UseParNewGC	串行GC	并发GC 当出现Concurrent Mode failure或promotion failed时则采用串行GC
不支持的组合方式	1. -XX:+UseParNewGC –XX:+UseParallelOldGC 2. -XX:+UseParNewGC –XX:+UseSerialGC	

资料整理 火龙果软件 荐

4、内存调优

资料整理
火龙果软件
荐

4 内存调优

首先需要注意：在对JVM内存调优的时候不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

① 旧生代空间不足调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象
② Permanet Generation空间不足增大Perm Gen空间，避免太多静态对象
③ 统计得到的GC后晋升到旧生代的平均大小大于旧生代剩余空间，控制好新生代和旧生代的比例
④ System.gc()被显示调用垃圾回收不要手动触发，尽量依靠JVM自身的机制

资料整理
火龙果软件
荐

4 内存调优

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现

几种比例设置不当的后果：

1）新生代设置过小

一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2）新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加。一般说来新生代占整个堆1/3比较合适

3）Survivor设置过小

导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4）Survivor设置过大

导致eden过小，增加了GC频率

另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题。

JVM提供两种较为简单的GC策略的设置方式

1）吞吐量优先

JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n来设置

2）暂停时间优先

JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n来设置

资料整理
火龙果软件
荐

4 内存调优　GC参数定义

参　数	描　述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden：Survivor=8：1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配

资料整理
火龙果软件
荐

4 内存调优　GC参数定义

MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

资料整理
火龙果软件
荐

Thanks!

资料整理
火龙果软件
荐