

垃圾回收

垃圾回收是计算机程序中一种自动管理内存的机制。它的主要目的是在程序运行时自动识别并回收不再被使用的内存空间，以便将这些空间释放出来供其他需要的部分使用，从而避免内存泄漏和内存溢出等问题，保证程序的稳定性和性能。

一、如何判断对象可以回收

1、引用计数法

引用计数法是一种简单的垃圾回收判断方法。

- 它给每个对象维护一个引用计数器，记录有多少个地方引用了该对象。
- 当引用计数为零时，表示没有任何引用指向该对象，因此可以被回收。
- 但是引用计数法存在循环引用的问题，无法解决循环引用导致的内存泄漏。

2、可达性分析算法

2.1 定义

可达性分析算法通过从根对象（如全局变量、活动栈、静态变量等）出发，通过引用链的方式遍历所有的对象，将可达的对象标记为活动对象，未被标记的对象即为垃圾对象，需要被回收。

- 这种方法可以解决循环引用的问题，是主流的垃圾回收判断方法。

2.2 GC Root

哪些对象可以作为GC Root呢？

GC Root是一组特殊的对象，它们被认为是不可回收的，作为垃圾回收的起始点。以下是可以作为GC Root的对象类型：

1. **虚拟机栈中的本地变量 (Local Variables)**：虚拟机栈中的本地变量指向的对象可以作为GC Root。这包括各个线程的栈帧中的本地变量，如方法参数和方法内部定义的局部变量。
2. **方法区中的类静态变量 (Static Variables)**：类静态变量存储在方法区中，它们属于类本身，被加载类的类对象所引用，因此也可以作为GC Root。
3. **方法区中的常量引用 (Constant References)**：方法区中的常量池存放着字符串常量和符号引用，这些常量引用可能被类的静态变量所引用，因此也算是GC Root。
4. **本地方法栈中JNI引用 (JNI Global Reference)**：如果Java代码通过JNI调用了本地方法，并且本地方法中保留了对Java对象的引用，这些JNI引用也被视为GC Root。

注意：除了GC Root对象，其它对象之间可能形成引用链，使得它们相互引用，但如果这些对象不可达到任何一个GC Root，那么它们将被判定为垃圾对象，将在垃圾回收时被清理。这种方式避免了引用计数算法的循环引用问题。

3、四种引用

3.1 定义

1. 强引用 (Strong Reference):

- 只有所有GC Roots对象都不通过**强引用**引用该对象，该对象才能被垃圾回收。
- 强引用是我们通常编码时使用的默认引用类型，只要有强引用指向一个对象，垃圾回收器不会回收该对象。

2. 软引用 (SoftReference):

- 仅有软引用引用该对象时，在垃圾回收后，内存仍不足时会再次触发垃圾回收，回收软引用对象。
- 可以配合引用队列 (ReferenceQueue) 来释放软引用自身。

3. 弱引用 (WeakReference):

- 仅有弱引用引用该对象时，在垃圾回收时，无论内存是否充足，都会回收弱引用对象。
- 可以配合引用队列 (ReferenceQueue) 来释放弱引用自身。

4. 虚引用 (PhantomReference):

- 必须配合引用队列 (ReferenceQueue) 使用，主要用于管理堆外内存或跟踪对象被回收的状态。
- 被虚引用引用的对象回收时，会将虚引用入队，由Reference Handler线程调用虚引用相关方法释放资源。

5. 终结器引用 (FinalReference)

- 无需手动编码，但其内部配合引用队列使用，在垃圾回收时，终结器引用入队（被引用对象暂时没有回收），再由Finalizer线程通过终结器引用找到被引用对象并调用它的finalize()方法，**第二次GC时才能回收被引用对象**
- 在现代Java中，终结器 (Finalizer) 机制已经不再推荐使用。finalize()方法的执行时间是不确定的，并且可能导致垃圾回收性能问题。取而代之，我们可以使用try-with-resources或手动管理资源来更可靠地释放资源。因此，在实际开发中，避免使用finalize()方法以及相关的Finalizer引用是推荐的做法。

3.2 示例

示例代码:

```

1  import java.lang.ref.PhantomReference;
2  import java.lang.ref.ReferenceQueue;
3  import java.lang.ref.SoftReference;
4  import java.lang.ref.WeakReference;
5
6  public class ReferenceTypesExample {
7      public static void main(String[] args) {
8          // 强引用示例
9          Object strongReference = new Object();
10         System.out.println("强引用示例 - 强引用对象存在，垃圾回收器不会回收该对象");
11         System.out.println("强引用对象: " + strongReference);
12         strongReference = null; // 取消对强引用对象的引用
13         System.gc(); // 主动触发垃圾回收


```

```

14      System.out.println("垃圾回收后强引用对象: " + strongReference);
15
16      // 软引用示例
17      SoftReference<Object> softReference = new SoftReference<>(new
Object());
18      System.out.println("\n软引用示例 - 弱引用对象存在, 但系统内存不足时会被回
收");
19      System.out.println("软引用对象: " + softReference.get());
20      //softReference = null; // 取消对软引用对象的引用
21      System.gc(); // 主动触发垃圾回收
22      System.out.println("垃圾回收后软引用对象: " + softReference.get());
23
24      // 弱引用示例
25      WeakReference<Object> weakReference = new WeakReference<>(new
Object());
26      System.out.println("\n弱引用示例 - 弱引用对象只能生存到下一次垃圾回收前");
27      System.out.println("弱引用对象: " + weakReference.get());
28      System.gc(); // 主动触发垃圾回收
29      System.out.println("垃圾回收后弱引用对象: " + weakReference.get());
30
31      // 虚引用示例
32      ReferenceQueue<Object> queue = new ReferenceQueue<>();
33      PhantomReference<Object> phantomReference = new PhantomReference<>
(new Object(), queue);
34      System.out.println("\n虚引用示例 - 虚引用对象不影响对象生命周期, 也无法通
过虚引用获取对象实例");
35      System.out.println("虚引用对象: " + phantomReference.get());
36      System.gc(); // 主动触发垃圾回收
37      System.out.println("垃圾回收后虚引用对象: " +
phantomReference.get());
38  }
39  }

```

输出:



```

1  强引用示例 - 强引用对象存在, 垃圾回收器不会回收该对象
2  强引用对象: java.lang.Object@1b6d3586
3  垃圾回收后强引用对象: null
4
5  软引用示例 - 弱引用对象存在, 但系统内存不足时会被回收
6  软引用对象: java.lang.Object@4554617c
7  垃圾回收后软引用对象: java.lang.Object@4554617c
8
9  弱引用示例 - 弱引用对象只能生存到下一次垃圾回收前
10 弱引用对象: java.lang.Object@74a14482
11垃圾回收后弱引用对象: null
12
13虚引用示例 - 虚引用对象不影响对象生命周期, 也无法通过虚引用获取对象实例
14虚引用对象: null
15垃圾回收后虚引用对象: null

```

3.3 应用场景

当谈到引用类型的具体场景时，请注意这些引用类型通常用于特定的内存管理需求。以下是每种引用类型的一些具体应用场景示例：

1. 强引用 (Strong Reference):

- 当你需要确保一个对象在程序的整个生命周期内都存在时，使用强引用。
- 保存全局配置信息或全局共享资源，以确保全局状态的一致性。
- 在实现单例模式时，使用强引用来保存单例实例，确保单例对象不会被提前回收。
- 在构造复杂对象时，使用强引用来确保相关对象不会在初始化完成前被回收。

2. 软引用 (Soft Reference):

- 实现内存敏感的高速缓存：对于某些数据，当系统内存足够时，保持其缓存，但在内存不足时允许垃圾回收器回收以释放内存。
- 图像缓存：在处理图像的过程中，一些图像可能不再频繁使用，使用软引用缓存图像可以节省内存，并在内存不足时回收不再使用的图像。
- 对象缓存：在需要频繁创建和销毁对象的情况下，使用软引用可以避免频繁的垃圾回收和内存分配，提高性能。

3. 弱引用 (Weak Reference):

- 实现关联数据结构：弱引用通常用于实现关联数据结构，如WeakHashMap，用于存储映射关系，并且在不再有强引用指向键时，允许垃圾回收器回收对应的键值对。
- 缓存管理：在缓存中，使用弱引用来保存缓存对象，以便在系统内存不足时，可以回收缓存中的对象，而不会导致内存溢出。

4. 虚引用 (Phantom Reference):

- 监控对象的回收状态：虚引用通常与ReferenceQueue一起使用，用于监控对象是否被垃圾回收器回收。可以在对象被回收时执行一些特定的操作。
- 管理堆外内存：虚引用在管理堆外内存时非常有用。当堆外内存不再被使用时，虚引用可以通知程序释放相应的资源。

5. 终结器引用 (Finalizer Reference):

- 终结器引用是一种特殊的引用类型，用于和对象的finalize()方法关联，当垃圾回收器准备回收一个对象时，如果该对象有关联的终结器引用，那么它将被加入到Finalization Queue（终结器队列）中，稍后将由Finalizer线程调用finalize()方法。然而，终结器引用在Java中已经被认为是不推荐的使用方式，因为finalize()方法的执行时间是不确定的，并且可能导致垃圾回收性能问题。

需要注意的是，软引用、弱引用、虚引用和终结器引用的使用需要谨慎。如果在错误的地方使用这些引用类型，可能导致程序出现内存泄漏或不稳定的行为。在使用这些引用类型时，请确保理解其内存管理特性，并根据实际需求进行合理的应用。在现代Java中，绝大多数情况下，建议使用弱引用和软引用，而虚引用和终结器引用的使用应该避免。

二、垃圾回收算法

垃圾回收算法是用来确定哪些内存块是垃圾（不再使用）的，以便将其回收并重新利用。下面分别介绍三种常见的垃圾回收算法：

1、标记清除 (Mark and Sweep)

标记清除是一种简单的垃圾回收算法，分为两个阶段：标记阶段和清除阶段。

1. **标记阶段**：从根对象开始，遍历所有可达对象，并将它们标记为活动对象，表示这些对象是可用的。
2. **清除阶段**：遍历整个堆，将未标记的对象（垃圾对象）进行回收，释放它们所占用的内存空间。

标记清除算法的优点是简单，容易实现，但它会造成内存碎片，可能导致后续内存分配时出现问题。

2、标记整理 (Mark and Compact)

标记整理是在标记清除的基础上进行优化的一种垃圾回收算法。

1. **标记阶段**：与标记清除算法相同，从根对象开始，遍历所有可达对象，并将它们标记为活动对象。
2. **整理阶段**：在清除阶段之前，将所有活动对象往一端移动，然后直接回收边界外的所有对象，从而将内存空间整理成连续的块，消除内存碎片。

标记整理算法解决了标记清除算法的内存碎片问题，但它需要额外的整理过程，因此可能会增加一定的回收时间。

3、分区复制 (Copying)

分区复制是一种将内存分为两个区域的垃圾回收算法：From区和To区。

1. **标记阶段**：从根对象开始，遍历所有可达对象，并将它们标记为活动对象。
2. **复制阶段**：将所有活动对象从From区复制到To区，同时对活动对象进行紧凑排列，消除内存碎片。
3. **角色交换**：交换From区和To区的角色，使得From区成为To区，反之亦然。

分区复制算法解决了内存碎片问题，并且避免了整理阶段的开销。不过，它要求堆空间的一半为From区，一半为To区，因此需要额外的内存。

每种垃圾回收算法都有其优缺点，不同的垃圾回收器会选择不同的算法或它们的组合，以满足特定的应用场景和性能要求。

三、分代垃圾回收

1、定义

分代垃圾回收是一种将堆内存划分为不同代（Generation）的垃圾回收策略。根据对象的生命周期特征，将对象分为不同的代别，然后根据不同代别的对象采用不同的垃圾回收算法和频率进行回收。

通常，Java堆内存被划分为三代：

1. **新生代 (Young Generation)**：
 - 新生代是存放新创建的对象区域，大部分对象在创建后很快就变成垃圾。
 - 使用较快的垃圾回收算法（如复制算法）进行频繁的垃圾回收，以尽早回收短生命周期对象。
 - 新生代又可以划分为Eden区、Survivor区From、Survivor区To。
2. **老年代 (Old Generation)**：

- 老年代用于存放生命周期较长的对象，大部分存活时间较长的对象存放在这里。
- 对老年代的垃圾回收相对较少，使用效率较高的垃圾回收算法（如标记整理算法）进行较少频率的回收。

3. 永久代 (Permanent Generation) (在 Java 8 后被元空间 (Metaspace) 取代) :

- 永久代用于存放静态对象、类信息、方法等数据。
- 这部分空间通常不会进行频繁的垃圾回收，而是在应用程序结束时或类加载器被回收时一次性地回收。

分代垃圾回收利用了大部分对象的生命周期特征：新创建的对象通常具有较短的生命周期，而已经存活较长时间的对象可能会继续存活较长时间。通过将对象划分为不同代别，分代垃圾回收器可以根据不同代别采用合适的垃圾回收算法和策略，从而提高垃圾回收的效率和性能。

在实际应用中，HotSpot虚拟机使用分代垃圾回收策略，并结合不同的垃圾回收器（如ParNew（新生代）、CMS（老年代）和G1（全堆））来实现高效的垃圾回收机制。

2、基本流程

好的，让我们来举一个从新生代到永久代（或元空间）的垃圾回收流程：

假设我们使用的是 HotSpot 虚拟机，并且采用了分代垃圾回收策略，其中新生代 (Young Generation) 使用复制算法，老年代 (Old Generation) 使用标记整理算法。

1. 初始状态:

- 新生代中的 Eden区、Survivor区From和Survivor区To都是空的，老年代中也是空的。
- 永久代（或元空间）中存放着类的元数据信息和其他静态数据。

2. 创建对象:

- 当我们在应用程序中创建新的对象时，这些对象首先被分配到新生代的 Eden区。

3. 新生代的垃圾回收 (Minor GC) :

- 当 Eden区满时，触发新生代的垃圾回收。
- 垃圾回收器会对Eden区、Survivor区From和Survivor区To进行垃圾回收，首先，它会从根对象（如全局变量、活动栈、静态变量等）出发，遍历所有的可达对象，并将它们标记为活动对象。
- 然后，所有活动对象将被复制到Survivor区To，并且为每个对象的年龄加1。年龄加1后，如果其中有对象的年龄达到一定阈值（通常为15岁）或者空间达到一定阈值时，则它们会被晋升到老年代。
- 清空Eden区和Survivor区From。

4. 停顿 (Stop-the-World) 现象:

- 在执行新生代的垃圾回收 (Minor GC) 时，会发生停顿现象。这意味着所有的应用程序线程都会被暂停，垃圾回收线程会独占CPU资源进行垃圾回收操作。
- 停顿时间的长短取决于垃圾回收器的实现和配置，通常在新生代垃圾回收时停顿时间比较短，但在老年代的垃圾回收 (Major GC或Full GC) 时，停顿时间可能会更长。

5. 老年代的垃圾回收 (Major GC或Full GC) :

- 当老年代中的空间满了，或者由于某些其他原因触发老年代的垃圾回收。
- 垃圾回收器会对老年代进行垃圾回收，首先，它会从根对象出发，遍历所有的可达对象，并将它们标记为活动对象。
- 然后，对老年代中的对象进行标记整理，将未被标记的垃圾对象进行回收，然后将存活对象紧凑排列，消除内存碎片。
- 对象晋升：从新生代晋升到老年代的对象和在老年代中存活的对象都会经过老年代的垃圾回收。

6. 永久代（或元空间）的垃圾回收:

- 永久代（或元空间）一般用于存放静态数据和类的元数据信息，它的垃圾回收通常在应用程序结束时或类加载器被回收时进行。
- 在 Java 8 之前，永久代是有限的，因此如果存在过多的类加载和卸载，可能导致永久代内存溢出。
- 在 Java 8 及以后的版本，永久代被元空间取代，它的内存由操作系统管理，不再有固定大小的限制。

请注意，以上流程中停顿现象的描述是垃圾回收过程中的一个重要细节。停顿现象会暂停应用程序的正常运行，因此在设计和配置垃圾回收时，需要根据应用程序的性能要求和实际场景，选择合适的垃圾回收器和调整垃圾回收的策略，以尽量减少停顿时间，提高应用程序的响应性能。

Minor GC和Full GC:

1. Minor GC（新生代垃圾回收）：

- Minor GC是针对新生代进行的垃圾回收，主要回收新创建的对象和短期存活的对象。
- Minor GC通常发生在新生代的Eden区满时触发。当Eden区满时，垃圾回收器会先对Eden区和Survivor区进行垃圾回收。
- 垃圾回收器通过可达性分析算法遍历所有的可达对象，将活动对象复制到Survivor区To，然后清空Eden区和Survivor区From，从而完成垃圾回收。
- 年龄加1：在垃圾回收过程中，每个对象的年龄加1，如果对象的年龄达到一定阈值（通常为15岁），则会晋升到老年代。

2. Full GC（老年代垃圾回收）：

- Full GC是针对老年代进行的垃圾回收，主要回收长期存活的对象和老年代中的垃圾对象。
- Full GC通常发生在老年代的空间满了，或者由于其他原因触发。
- 垃圾回收器首先会对老年代进行可达性分析，标记所有的可达对象。
- 然后，对老年代中的对象进行标记整理，将未被标记的垃圾对象进行回收，然后将存活对象紧凑排列，消除内存碎片。
- Full GC的执行频率通常较低，因为它涉及到整个堆内存的垃圾回收，会导致较长的停顿时间。

需要注意的是，Minor GC通常是并行执行的，因为它只涉及到新生代的垃圾回收，而新生代的空间通常比较小。而Full GC通常是串行执行的，因为它涉及到整个堆内存的垃圾回收，需要较长的处理时间，因此会引发较长的停顿时间，暂停所有的应用程序线程。为了减少Full GC的停顿时间，一些垃圾回收器使用了并发或并行的执行方式。

选择合适的垃圾回收器和调整垃圾回收的策略是优化应用程序性能的重要方面，特别是要避免Full GC频繁发生，以免影响应用程序的响应性能。不同的垃圾回收器有不同的特点和适用场景，需要根据实际应用的需求进行选择 and 配置。

3、VM参数部分

以下是一些常用的和重要的 JVM 参数，供您参考：

1. 通用参数：

- `-version` : 显示 JVM 版本信息。
- `-help` 或 `-h` : 显示 JVM 的帮助信息。
- `-server` : 使用服务器模式启动 JVM，适用于长时间运行的服务器应用。
- `-client` : 使用客户端模式启动 JVM，适用于快速启动和短时间运行的应用。
- `-X` : 显示非标准选项的帮助信息。
- `-XX:+PrintFlagsFinal` : 显示所有的 JVM 参数及其默认值。

- `-XX:+PrintCommandLineFlags` : 显示 JVM 启动时所有传递的参数。

2. 堆内存设置:

- `-Xmx<size>` : 设置 Java 堆的最大内存大小。
- `-Xms<size>` : 设置 Java 堆的初始内存大小。
- `-Xmn<size>` : 设置新生代的内存大小。
- `-XX:NewRatio=<ratio>` : 设置老年代和新生代的内存比例。
- `-XX:SurvivorRatio=<ratio>` : 设置 Eden 区和 Survivor 区的内存比例。

3. 垃圾回收设置:

- `-XX:+UseSerialGC` : 使用串行垃圾回收器。
- `-XX:+UseParallelGC` : 使用并行垃圾回收器。
- `-XX:+UseConcMarkSweepGC` : 使用并发标记清除垃圾回收器。
- `-XX:+UseG1GC` : 使用 G1 垃圾回收器。
- `-XX:MaxGCPauseMillis=<milliseconds>` : 设置最大垃圾回收暂停时间目标。

4. 调优参数:

- `-XX:MaxHeapFreeRatio=<percent>` : 设置最大空闲堆内存占比, 超过该比例会收缩堆。
- `-XX:MinHeapFreeRatio=<percent>` : 设置最小空闲堆内存占比, 低于该比例会扩展堆。
- `-XX:ParallelGCThreads=<num>` : 设置并行垃圾回收器的线程数。
- `-XX:ConcGCThreads=<num>` : 设置并发垃圾回收器的线程数。

5. 性能监控参数:

- `-XX:+PrintGCDetails` : 打印详细的垃圾回收信息。
- `-XX:+PrintGCDateStamps` : 打印垃圾回收的时间戳。
- `-XX:+PrintHeapAtGC` : 打印垃圾回收前后的堆内存信息。
- `-XX:+PrintTenuringDistribution` : 打印对象年龄分布信息。

6. 本章节: 下列是 JVM 常用的一些调优参数, 用于调整堆大小、新生代大小、幸存区比例、晋升年龄阈值等, 以优化垃圾回收的性能和内存利用率。

- `-Xms<size>` : 堆的初始大小。该参数设置 JVM 的初始堆大小, 即在 JVM 启动时分配给堆的内存量。
- `-Xmx<size>` : 堆的最大大小。该参数设置 JVM 的最大堆大小, 即堆可使用的最大内存量。
- `-Xmn(<size>)` : 新生代的大小。该参数设置 JVM 新生代的初始大小, 即新生代可使用的内存量。
- `-XX:MaxHeapSize(<size>)` : 堆的最大大小。与 `-Xmx` 相同, 用于设置 JVM 堆的最大大小。
- `-XX:InitialSurvivorRatio(<ratio>)` : 幸存区的初始比例。该参数用于设置 Eden 区和 Survivor 区的初始比例。
- `-XX:+UseAdaptiveSizePolicy` : 幸存区比例动态调整。该参数允许 JVM 动态调整幸存区的大小。
- `-XX:SurvivorRatio(<ratio>)` : 幸存区的比例。该参数设置 Eden 区和 Survivor 区的比例。
- `-XX:MaxTenuringThreshold(<threshold>)` : 晋升年龄阈值。该参数设置对象晋升到老年代的年龄阈值。
- `-XX:+PrintTenuringDistribution` : 打印晋升详情。启用该参数后, JVM 在每次垃圾回收时打印晋升情况。
- `-XX:+PrintGCDetails` : 打印 GC 详情。启用该参数后, JVM 在进行垃圾回收时打印详细的 GC 信息。
- `-verbose:gc` : 打印 GC 信息。该参数启用 JVM 输出简要的 GC 信息。
- `-XX:+ScavengeBeforeFullGC` : Full GC 前进行 Minor GC。该参数在进行 Full GC 之前先执行一次 Minor GC。

这些只是 JVM 参数中的一部分，实际上 JVM 提供了众多的参数选项，用于调整内存、垃圾回收、性能监控等方面的行为。您可以根据应用的需求和实际情况，选择合适的参数进行配置，以优化应用程序的性能和稳定性。若需要更详细的参数列表，可以参考 Oracle 官方文档或相关 JVM 实现的文档。

四、垃圾回收器

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

垃圾回收器是 JVM 中负责执行垃圾回收的组件，它们采用不同的算法和策略来回收无用对象并释放内存。以下是三种常见的垃圾回收器。

1、串行

1.1 定义

串行垃圾回收器：

- 串行垃圾回收器是最简单的垃圾回收器，也是 JVM 默认的垃圾回收器（在客户端模式下）。
- 它使用单个线程进行垃圾回收，因此也称为单线程垃圾回收器。
- 串行垃圾回收器适用于单核处理器或较小的应用程序，它的优势在于简单高效。

1.2 命令

开启串行垃圾回收器的命令：

`-XX:+UseSerialGC=Serial+SerialOld` 将会启用 Serial 收集器（新生代使用 Serial 收集器，老年代使用 Serial Old 收集器）。这是一种串行的垃圾回收方式。

- 新生代（Young Generation）使用 Serial 收集器 `xx:+UseSerialGC=Serial`：
 - Serial 收集器是一个单线程的收集器，使用**复制算法**来进行垃圾回收。
 - 在新生代的垃圾回收过程中，应用程序的所有用户线程都会被暂停，这叫做 STW（Stop-The-World）事件。
 - 它适合于小型或单核处理器的应用，由于只使用单个线程，对于较小的堆和简单的应用场景效果较好。
- 老年代（Old Generation）使用 Serial Old 收集器 `xx:+UseSerialGC=SerialOld`：
 - Serial Old 收集器也是一个单线程的收集器，使用**标记-整理算法**来进行垃圾回收。
 - 在老年代的垃圾回收过程中，同样会暂停所有用户线程，产生 STW 事件。
 - 适合单核处理器或对吞吐量要求不高的应用场景，但不适合大型应用程序和需要高吞吐量的后台任务。

2、吞吐量优先

2.1 定义

吞吐量优先垃圾回收器：

- 多线程垃圾回收器，适用于多核 CPU 和堆内存较大的服务器环境。
- 通过并行处理，尽可能减少 STW（Stop-The-World）的时间，以提高应用程序的吞吐量。
- 适合后台任务、数据处理等对吞吐量要求较高的应用。

- G1 (Garbage-First) 垃圾回收器是一个常见的吞吐量优先垃圾回收器。
- 让**单位时间内**，STW的时间最短。

2.2 命令

下列是详细解释一下每个参数的含义：

1. **-XX:+UseParallelGC**：启用 ParallelGC 垃圾回收器。ParallelGC 是一种并行的垃圾回收器，用于新生代，采用复制算法。它会使用多个线程来进行垃圾回收，以提高吞吐量，会同时开启old。
2. **-XX:+UseParallelOldGC**：启用 Parallel Old 垃圾回收器。Parallel Old 是一种并行的垃圾回收器，用于老年代，采用标记-整理算法。它同样会使用多个线程来进行垃圾回收，以提高吞吐量，会同时开启上一个。
3. **-XX:+UseAdaptiveSizePolicy**：启用自适应的内存分配策略。垃圾回收器会根据当前的系统资源和应用程序的行为动态调整堆内存的大小，以达到最佳的性能。
4. **-XX:GCTimeRatio=ratio**：设置吞吐量优先垃圾回收器的目标吞吐量。默认值为 99，表示垃圾回收时间占总执行时间的比例不超过 1%。您可以根据应用程序的性能需求调整这个值，[公式： \$1/\(1+ratio\)\$](#) 。
5. **-XX:MaxGCPauseMillis=ms**：设置吞吐量优先垃圾回收器的最大停顿时间。默认值为 200 毫秒。您可以根据应用程序对停顿时间的要求进行调整，以尽量减少 STW (Stop-The-World) 事件。
6. **-XX:ParallelGCThreads=n**：设置并行垃圾回收器的线程数。默认值为 CPU 核心数的一半。您可以根据系统的 CPU 核心数和性能需求进行调整。

吞吐量优先垃圾回收器 (ParallelGC 和 Parallel Old) 适用于多核 CPU 和堆内存较大的服务器环境，它通过并行处理，尽量减少 STW 的时间，从而提高应用程序的吞吐量。选择合适的垃圾回收器和调整相应的参数，是优化应用程序性能的关键因素。

3、响应时间优先

3.1 定义

响应时间优先垃圾回收器：

- 多线程垃圾回收器，适用于多核 CPU 和堆内存较大的服务器环境。
- 通过并发标记和重新标记，尽可能减少 STW 的时间，以提供更好的响应时间。
- 适合交互式应用、需要快速响应的应用场景。
 - CMS (Concurrent Mark-Sweep) 垃圾回收器是一个常见的响应时间优先垃圾回收器。
- 让**单次**STW的时间最短。

3.2 命令

CMS垃圾回收器

- **-XX:+UseConcMarkSweepGC**：启用 CMS 垃圾回收器，老年代使用的。
- **-XX:+UseParNewGC**：可选项，启用 ParNew 垃圾回收器。ParNew 是 CMS 的前提条件，在新生代使用 ParNew 垃圾回收器。

下列是 CMS 垃圾回收器的一些相关参数：

- `-XX:ParallelGCThreads=n` : 设置并行垃圾回收器的线程数。默认值为 CPU 核心数的一半。这个参数同时也适用于 CMS 垃圾回收器的并发标记阶段。
- `-XX:ConcGCThreads=threads` : 设置 CMS 垃圾回收器的并发线程数。默认值为 CPU 核心数的 1/4。并发线程主要用于执行并发标记和并发清除过程。
- `-XX:CMSInitiatingOccupancyFraction=percent` : 设置 CMS 垃圾回收器在老年代达到多少百分比的空间时触发垃圾回收。默认值为 68%。增加这个值可以让 CMS 更早地启动垃圾回收, 以减少 stop world 时间, 但会增加垃圾回收的频率。
- `-XX:+CMSScavengeBeforeRemark` : 在执行 CMS 的并发标记阶段之前, 先执行一次新生代的垃圾回收。这样做可以减少并发标记的工作量, 但会增加新生代垃圾回收的时间。如果并发标记时间较长, 可以考虑启用这个参数。

五、垃圾回收调优

垃圾回收调优是指通过调整垃圾回收器的参数和算法, 以提高程序的性能和稳