Please visit website: http://cxyroad.com

走进Java虚拟机（JVM）堆内存

开头：

在Java的世界里，理解Java虚拟机（JVM）的堆内存对于开发高效、稳定的应用程序至关重要。JVM堆内存是所有线程共享的一块内存区域，它负责存放Java对象和数组，是垃圾回收的主要舞台。本文将带你走进JVM堆内存的深层结构，揭示其运作原理，以及它如何影响你的Java应用。我们将从堆内存的组成部分讲起，探索年轻代和老年代的角色，以及在Java 8中引入的元空间。

堆内存空间大小：

在Java虚拟机（JVM）中，堆内存是用于存储Java对象实例的主要区域。堆内存被分为几个部分，主要包括年轻代（Young Generation）、老年代（Old Generation）或称为老生代，以及在某些情况下的永久代（PermGen，已在Java 8中被元空间Metaspace取代）。默认的堆内存分配比例主要取决于JVM的版本和厂商实现，以及可能的启动参数。对于HotSpot JVM，年轻代和老年代的默认比例大约如下：

- **年轻代：** 默认情况下，年轻代占整个堆的1/3。
- **老年代：** 默认情况下，老年代占整个堆的2/3。

年轻代进一步分为三个部分：一个Eden区和两个幸存者区（Survivor spaces），通常称为S0和S1。默认情况下，Eden区和两个幸存者区的比例大约是8:1:1，这意味着每个幸存者区大约是整个年轻代的10%。

请注意，这些比例可以通过JVM启动参数进行调整。例如，可以使用以下参数来改变年轻代和老年代的比例：

- `-XX:NewRatio`：设置老年代与年轻代的比例。例如，`-XX:NewRatio=3` 表示老年代将是年轻代大小的三倍。
- `-XX:SurvivorRatio`：设置Eden区与一个幸存者区的比例。例如，`-XX:SurvivorRatio=8` 表示Eden区将是一个幸存者区大小的八倍。

比例图：

```
graph TD
    subgraph Heap
        subgraph YoungGen Young Generation 1/3 of Heap
            Eden[Eden 8/10 of Young Gen]
            S0[Survivor S0 1/10 of Young Gen]
            S1[Survivor S1 1/10 of Young Gen]
        end
        OldGen[Old Generation 2/3 of Heap]
    end

    Heap --> YoungGen
    Heap --> OldGen
    YoungGen --> Eden
    YoungGen --> S0
    YoungGen --> S1
```

调整这些比例可能会影响垃圾收集的性能，因此在调整之前应该进行充分的测试。在实际应用中，适当的堆内存分配比例取决于应用程序的特定工作负载和垃圾收集行为。

对象流转过程：

在Java中，对象的生命周期和内存分配是由Java虚拟机（JVM）中的垃圾收集器（Garbage Collector，GC）管理的。JVM的堆内存主要分为两个区域：年轻代（Young Generation）和老年代（Old Generation）。以下是对象在这些区域中的流程和流转过程的概述：

1. **对象创建：**

- 当在Java应用程序中创建一个新对象时，它通常首先被分配到堆内存的年轻代的Eden空间中。

2. **初次垃圾回收：**

– 当Eden空间填满时，触发一次名为Minor GC的垃圾收集事件。

– 存活的对象（即在当前运行的应用程序中仍被引用的对象）从Eden空间被移动到一个幸存者空间（Survivor space，通常称为S0）。

– 如果对象在Eden区存活过一次垃圾收集，它的年龄就会增加。JVM中通常有一个年龄计数器来追踪对象在年轻代中存活的次数。

3. **在幸存者空间之间移动：**

– 随着更多的Minor GC发生，存活的对象会在两个幸存者空间（S0和S1）之间移动。每次Minor GC后，存活对象的年龄会增加，并被移动到另一个幸存者空间，而当前幸存者空间被清空。 存活阈值默认是15 这意味着一个对象在Survivor区（即年轻代中的Survivor空间）中移动了15次垃圾回收（GC）后，如果它仍然存活，它将被晋升到老年代（Old Generation）
  – 当对象在年轻代中存活了足够次数（这个阈值可以通过`-XX:MaxTenuringThreshold`参数设置）后，它们会被晋升（Promotion）到老年代。

4. **晋升到老年代：**

– 老年代是用于存储长期存活的对象的区域。对象在年轻代中存活了足够的垃圾收集周期后，或者如果它们太大而无法在年轻代中分配时，会被移动到老年代。

5. **老年代的垃圾回收：**

– 当老年代接近满时，会触发Major GC或Full GC，这是一次更彻底的垃圾收集过程。

– Major GC通常比Minor GC慢得多，因为它涉及整个老年代，有时甚至包括年轻代和其他内存区域。

6. **垃圾回收和对象销毁：**

– 在任何GC过程中，不再被应用程序引用的对象将被识别并回收，释放内存供新对象使用。

实际的垃圾收集和内存管理过程可能会根据使用的JVM版本、选择的垃圾收集器以及JVM的配置参数而有所不同。从Java 8开始，永久代（PermGen）被元空间（Metaspace）替代，它存储了类元数据，并且位于本地内存中而不是堆内存中。这些改变也会影响内存管理的具体细节。

流程图：

```
graph TD
    A[创建对象] -->|分配到Eden区| B(Eden区)
    B --> C{Eden区满了吗?}
    C -->|是| D[Minor GC]
    C -->|否| B
    D --> E[存活对象移到Survivor区 S0 ]
    E --> F{Survivor区 S0 满了吗?}
    F -->|是| G[Minor GC]
    F -->|否| E
    G --> H[存活对象在S0和S1间移动]
    H --> I{对象年龄超过阈值?}
    I -->|是| J[晋升到老年代]
    I -->|否| H
    J --> K{老年代满了吗?}
    K -->|是| L[Major GC/Full GC]
    K -->|否| J
    L --> M[清理老年代中无引用对象]
    M --> N[老年代空间释放]
    N --> J
```

流程图中，可以看到以下流程：

1. 对象被创建并分配到Eden区。

2. 当Eden区满了，会触发Minor GC。

3. Minor GC期间，存活的对象会被移到一个Survivor区（比如S0）。

4. 随着更多Minor GC的发生，存活的对象会在Survivor区S0和S1之间移动，并且它们的年龄会增加。

5. 当对象的年龄超过设定的阈值时，它们会被晋升到老年代。

6. 当老年代满了，会触发Major GC或Full GC。

7. Major GC或Full GC会清理老年代中无引用的对象，释放空间。

时序图：

```
sequenceDiagram
    participant 创建对象
    participant Eden区
    participant Survivor区S0
    participant Survivor区S1
    participant 老年代
    participant 垃圾回收器

    创建对象->>Eden区: 对象分配到Eden
    Eden区->>垃圾回收器: Eden满了
    垃圾回收器->>Eden区: 触发Minor GC
    垃圾回收器->>Survivor区S0: 存活对象移动到S0
    Survivor区S0->>垃圾回收器: S0满了
    垃圾回收器->>Survivor区S0: 触发Minor GC
    垃圾回收器->>Survivor区S1: 存活对象移动到S1
    Survivor区S1->>垃圾回收器: S1满了
    垃圾回收器->>Survivor区S1: 触发Minor GC
    垃圾回收器->>Survivor区S0: 存活对象移回S0
    loop 对象在Survivor区间移动
        垃圾回收器->>Survivor区S0: 对象年龄增加
        垃圾回收器->>Survivor区S1: 对象年龄增加
        Note over 垃圾回收器: 对象在Survivor区间移动\n并且年龄逐渐增加
    end
    垃圾回收器->>老年代: 对象年龄达到阈值\n晋升到老年代
    老年代->>垃圾回收器: 老年代满了
    垃圾回收器->>老年代: 触发Major GC
    垃圾回收器->>老年代: 清理无引用对象
```

在这个时序图中，可以看到以下步骤：

1. 对象在创建时首先被分配到Eden区。

2. 当Eden区填满时，会触发Minor GC，存活的对象会被移动到Survivor区S0。

3. 随着更多的Minor GC事件，存活的对象可能会在Survivor区S0和S1之间移动，并且它们的年龄会逐渐增加。

4. 当对象的年龄达到一定阈值或Survivor区无法容纳更多对象时，它们会被晋升到老年代。

5. 当老年代填满时，会触发Major GC，清理掉无引用的对象。

参数介绍：

在Java虚拟机（JVM）中，堆内存分配和管理是通过一系列的参数来配置的。这些参数允许开发者和系统管理员调整JVM以最佳地适应应用程序的需求
以下是一些常用的JVM堆内存分配相关参数及其详细解释：

1. **-Xms**

- 设置JVM启动时堆的初始大小。

- 例如：`-Xms512m` 设置初始堆大小为512兆字节。

2. **-Xmx**

- 设置JVM允许的堆的最大大小。

- 例如：`-Xmx1024m` 设置最大堆大小为1024兆字节。

3. **-Xmn**

- 设置年轻代的大小。

- 例如：`-Xmn256m` 设置年轻代大小为256兆字节。

- 这个参数会影响Eden区和两个Survivor区的总和大小。

4. **-XX:NewRatio**

- 设置老年代（Old Generation）和年轻代（Young Generation）的比例。

- 例如：`-XX:NewRatio=3` 表示老年代将是年轻代大小的三倍。

5. **-XX:SurvivorRatio**

- 设置Eden区与Survivor区的大小比例。

- 例如：`-XX:SurvivorRatio=8` 表示Eden区是每个Survivor区大小的8倍。

6. **-XX:PermSize**

- 设置永久代（PermGen）的初始大小（Java 8之前的版本）。

- 例如：`-XX:PermSize=64m` 设置永久代的初始大小为64兆字节。

7. **-XX:MaxPermSize**

- 设置永久代（PermGen）的最大大小（Java 8之前的版本）。

- 例如：`-XX:MaxPermSize=128m` 设置永久代的最大大小为128兆字节。

8. **-XX:MetaspaceSize**

- 设置元空间（Metaspace）的初始大小（Java 8及以后的版本）。

- 例如：`-XX:MetaspaceSize=128m` 设置元空间的初始大小为128兆字节。

9. **-XX:MaxMetaspaceSize**

- 设置元空间（Metaspace）的最大大小（Java 8及以后的版本）。

- 例如：`-XX:MaxMetaspaceSize=256m` 设置元空间的最大大小为256兆字节。

10. **-XX:+UseG1GC**

- 启用G1垃圾收集器。

- G1（Garbage-First）是一种服务器风格的垃圾收集器，适用于多核处理器和大内存服务器。

11. **-XX:+UseConcMarkSweepGC**

- 启用CMS（Concurrent Mark Sweep）垃圾收集器。

- CMS收集器旨在减少应用程序暂停时间。

12. **-XX:+UseParallelGC**

- 启用并行垃圾收集器。

– 这个收集器在垃圾收集阶段使用多个线程来加速垃圾回收过程。

13. **–XX:+UseParallelOldGC**

– 启用并行老年代垃圾收集器。

– 这是并行收集器的老年代版本，它也使用多线程来回收老年代中的垃圾。

14. **–XX:+PrintGCDetails**

– 打印垃圾回收的详细信息。

– 这对于调试和优化垃圾收集性能非常有用。

总结：
===

通过本文的探讨，我们对JVM堆内存有了更深入的理解。从年轻代的快速对象周转到老年代的长期存储，再到元空间的类元数据管理，JVM堆内存的结构和垃圾回收机制共同确保了Java应用的性能和稳定性。

原文链接: https://juejin.cn/post/7393740927171444799