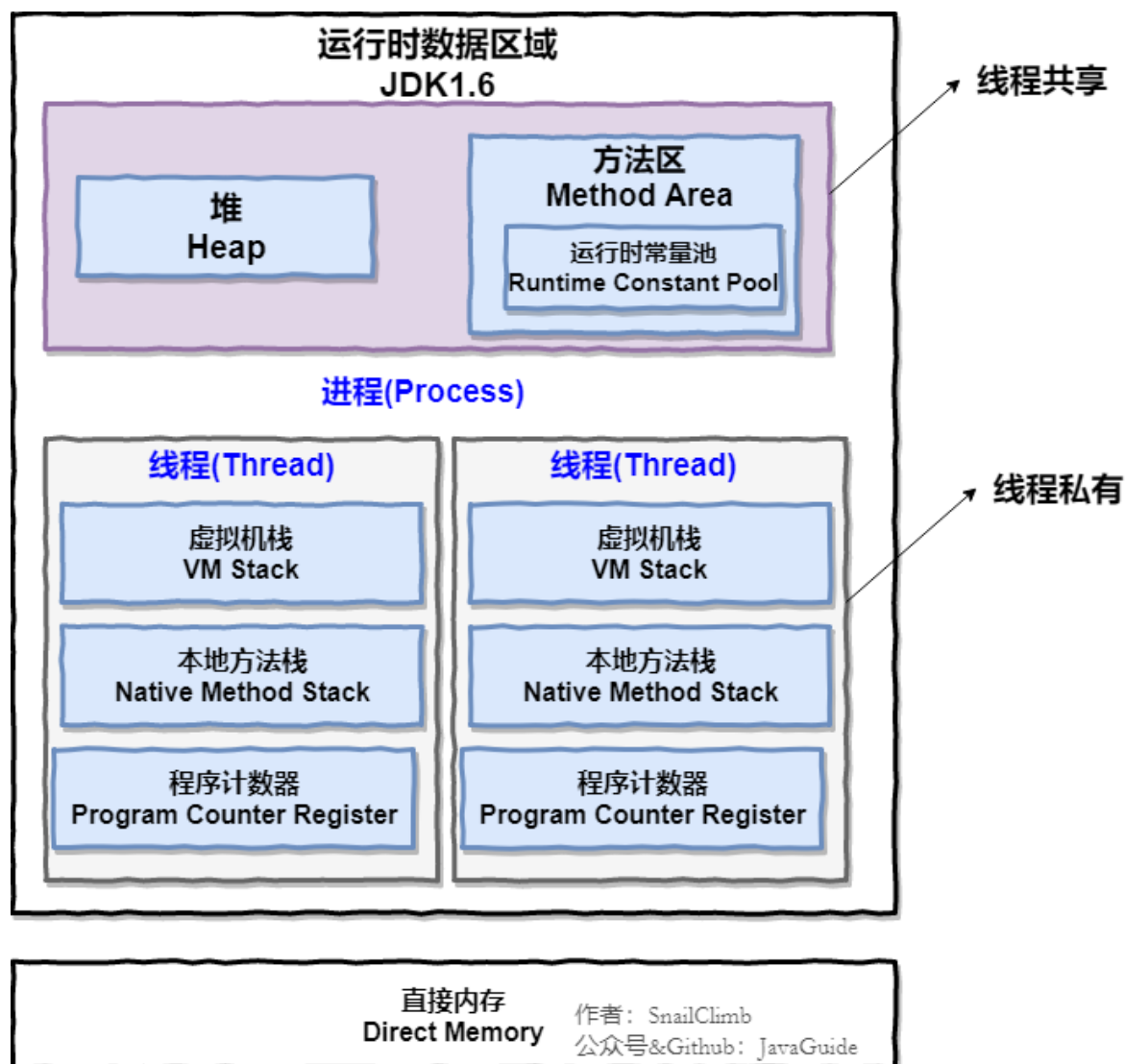


Java虚拟机

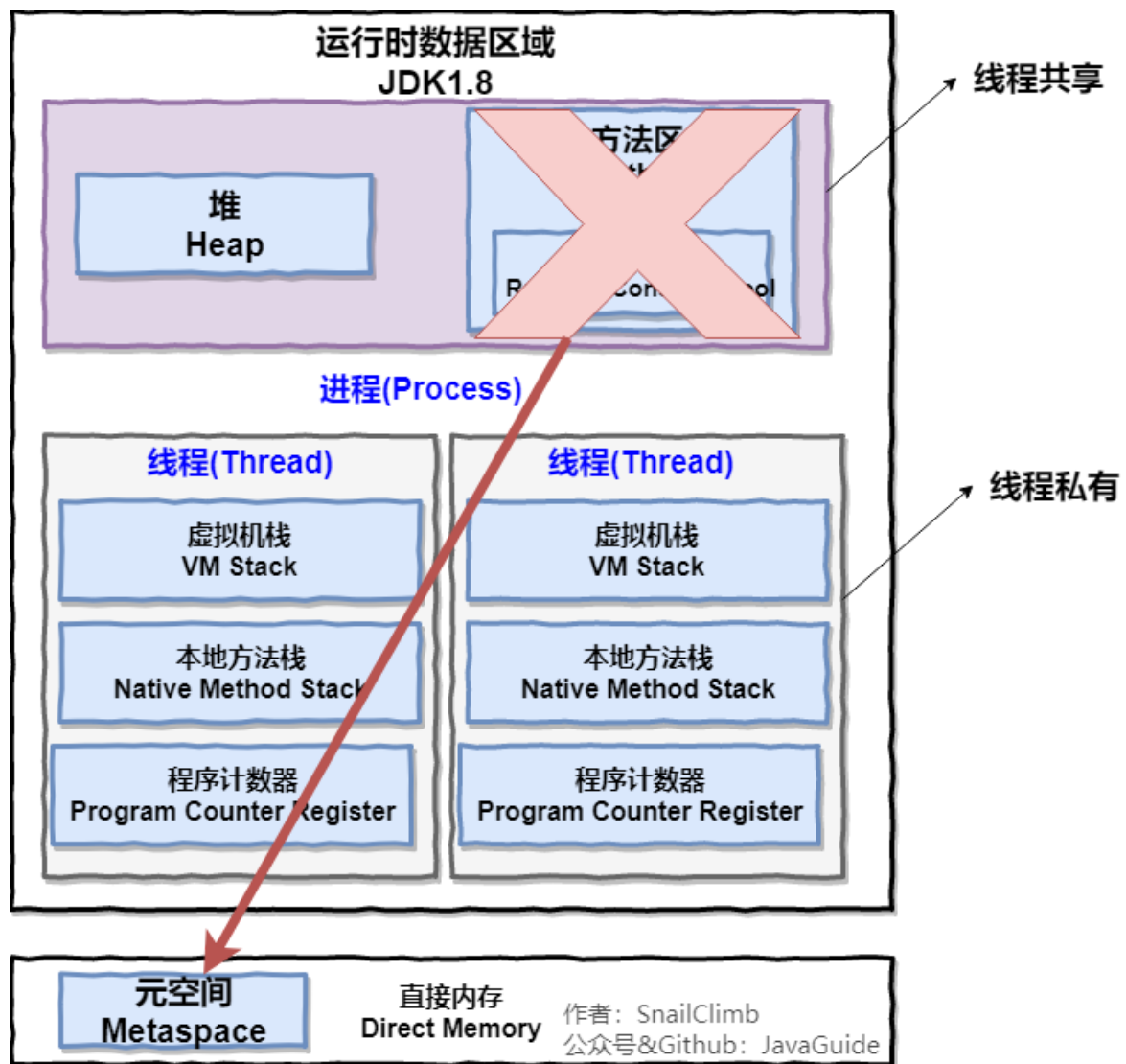
运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK. 1.8 和之前的版本略有不同，下面会介绍到。

JDK 1.8 之前：



JDK 1.8 :



线程私有的:

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的:

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)

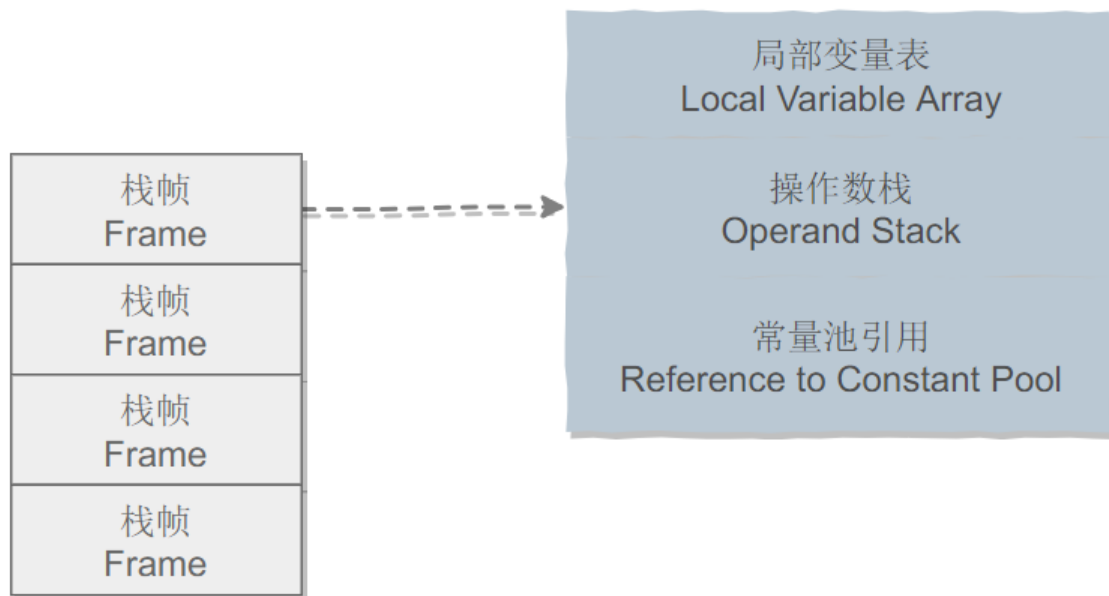
1.1 程序计数器

记录正在执行的虚拟机字节码指令的地址 (如果正在执行的是本地方法则为空)。

注意: 程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域, 它的生命周期随着线程的创建而创建, 随着线程的结束而死亡。

1.2 Java 虚拟机栈

每个 Java 方法在运行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程, 对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。



CyC2018

可以通过 -Xss 这个虚拟机参数来指定每个线程的 Java 虚拟机栈内存大小：

```
java -Xss512M HackTheJava
```

该区域可能抛出以下异常：

- 当线程请求的栈深度超过最大值，会抛出 `StackOverflowError` 异常；
- 栈进行动态扩展时如果无法申请到足够内存，会抛出 `OutOfMemoryError` 异常。

1.3 本地方法栈

本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。

本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统
的程序，对待这些方法需要特别处理。

1.4 堆

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。**

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆（Garbage Collected Heap）**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代（再细致一点有：Eden 空间、From Survivor、To Survivor 空间等）。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

eden 区、s0 区、s1 区都属于新生代，tenured 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

1.5 方法区

用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 `OutOfMemoryError` 异常。

对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。

HotSpot 虚拟机把它当成永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 `OutOfMemoryError` 异常。为了更容易管理方法区，从 JDK 1.8 开始，移除永久代，并把方法区移至元空间，它位于本地内存中，而不是虚拟机内存中。

方法区是一个 JVM 规范，永久代与元空间都是其一种实现方式。在 JDK 1.8 之后，原来永久代的数据被分到了堆和元空间中。元空间存储类的元信息，静态变量和常量池等放入堆中。

1.6 运行时常量池

运行时常量池是方法区的一部分。

Class 文件中的常量池（编译器生成的字面量和符号引用）会在类加载后被放入这个区域。

除了在编译期生成的常量，还允许动态生成，例如 `String` 类的 `intern()`。

1.7 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 `OutOfMemoryError` 异常出现。

在 JDK 1.4 中新引入了 NIO 类，它可以使用 Native 函数库直接分配堆外内存，然后通过 Java 堆里的 `DirectByteBuffer` 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在堆内存和堆外内存来回拷贝数据。

垃圾收集

垃圾收集主要是针对堆和方法区进行。程序计数器、虚拟机栈和本地方法栈这三个区域属于线程私有的，只存在于线程的生命周期内，线程结束之后就会消失，因此不需要对这三个区域进行垃圾回收。

判断一个对象是否可被回收

1. 引用计数算法

为对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收。

在两个对象出现循环引用的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收。正是因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法。

```
public class Test {

    public Object instance = null;

    public static void main(String[] args) {
        Test a = new Test();
        Test b = new Test();
        a.instance = b;
        b.instance = a;
        a = null;
        b = null;
        doSomething();
    }
}
```

```
}
```

在上述代码中，a 与 b 引用的对象实例互相持有了对象的引用，因此当我们把对 a 对象与 b 对象的引用去除之后，由于两个对象还存在互相之间的引用，导致两个 Test 对象无法被回收。

2. 可达性分析算法

以 GC Roots 为起始点进行搜索，可达的对象都是存活的，不可达的对象可被回收。

Java 虚拟机使用该算法来判断对象是否可被回收，GC Roots 一般包含以下内容：

- 虚拟机栈中局部变量表中引用的对象
- 本地方法栈中 JNI 中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中的常量引用的对象

新生代用广度搜索，老年代用深度搜索

深度优先DFS一般采用递归方式实现，处理tracing的时候，可能会导致栈空间溢出，所以一般采用广度优先来实现tracing（递归情况下容易爆栈）。

广度优先的拷贝顺序使得GC后对象的空间局部性（memory locality）变差（相关变量散开了）。

广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索算法法要快些。

深度优先搜索法占内存少但速度较慢，广度优先搜索算法占内存多但速度较快。

3. 方法区的回收

因为方法区主要存放永久代对象，而永久代对象的回收率比新生代低很多，所以在方法区上进行回收性价比不高。

主要是对常量池的回收和对类的卸载。

为了避免内存溢出，在大量使用反射和动态代理的场景都需要虚拟机具备类卸载功能。

类的卸载条件很多，需要满足以下三个条件，并且满足了条件也不一定会被卸载：

- 该类所有的实例都已经被回收，此时堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 Class 对象没有在任何地方被引用，也就无法在任何地方通过反射访问该类方法。

4. finalize()

类似 C++ 的析构函数，用于关闭外部资源。但是 try-finally 等方式可以做得更好，并且该方法运行代价很高，不确定性大，无法保证各个对象的调用顺序，因此最好不要使用。

当一个对象可被回收时，如果需要执行该对象的 finalize() 方法，那么就有可能在该方法中让对象重新被引用，从而实现自救。自救只能进行一次，如果回收的对象之前调用了 finalize() 方法自救，后面回收时不会再调用该方法。

5.彻底死亡条件

条件1：通过GC Roots作为起点的向下搜索形成引用链，没有搜到该对象，这是第一次标记。

条件2：在finalize方法中没有逃脱回收（将自身被其他对象引用），这是第一次标记的清理。

引用类型

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

Java 提供了四种强度不同的引用类型。

1. 强引用

被强引用关联的对象不会被回收。

使用 new 一个新对象的方式来创建强引用。

```
Object obj = new Object();
```

2. 软引用

被软引用关联的对象只有在内存不够的情况下才会被回收。

使用 SoftReference 类来创建软引用。

```
Object obj = new Object();  
SoftReference<Object> sf = new SoftReference<Object>(obj);  
obj = null; // 使对象只被软引用关联
```

3. 弱引用

被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。

使用 WeakReference 类来创建弱引用。

```
Object obj = new Object();  
WeakReference<Object> wf = new WeakReference<Object>(obj);  
obj = null;
```

4. 虚引用

又称为幽灵引用或者幻影引用，一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到一个对象。

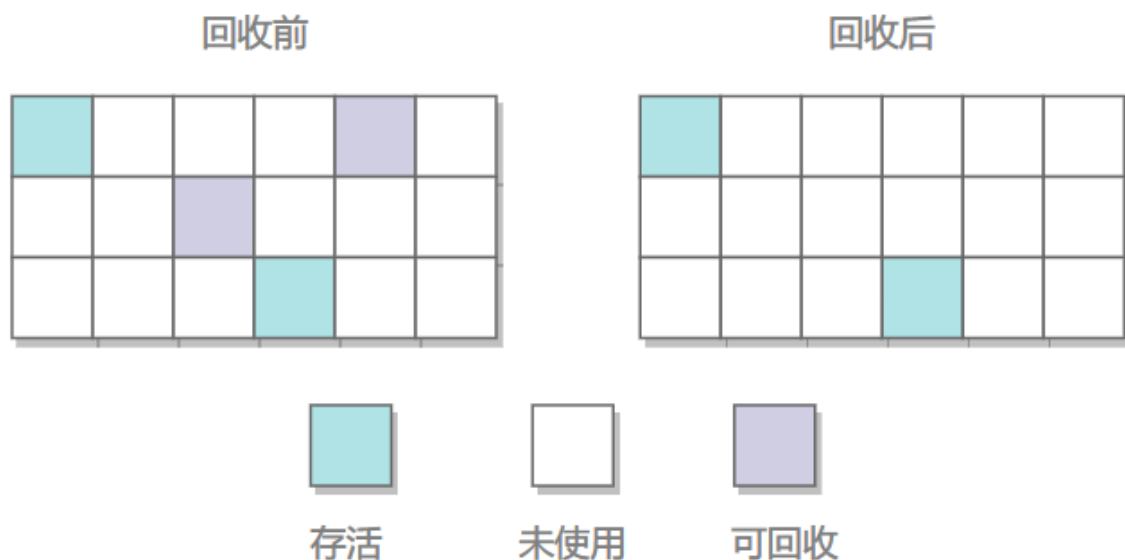
为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知。

使用 PhantomReference 来创建虚引用。

```
Object obj = new Object();  
PhantomReference<Object> pf = new PhantomReference<Object>(obj, null);  
obj = null;
```

垃圾收集算法

1. 标记 - 清除



在标记阶段，程序会检查每个对象是否为活动对象，如果是活动对象，则程序会在对象头部打上标记。

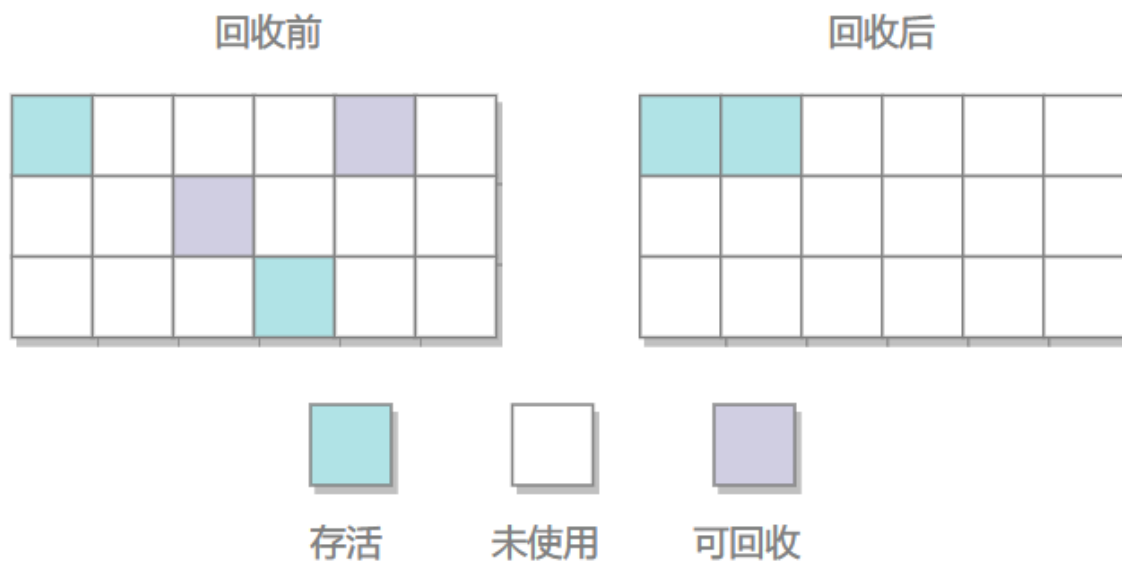
在清除阶段，会进行对象回收并取消标志位，另外，还会判断回收后的分块与前一个空闲分块是否连续，若连续，会合并这两个分块。回收对象就是把对象作为分块，连接到被称为“空闲链表”的单向链表，之后进行分配时只需要遍历这个空闲链表，就可以找到分块。

在分配时，程序会搜索空闲链表寻找空间大于等于新对象大小 `size` 的块 `block`。如果它找到的块等于 `size`，会直接返回这个分块；如果找到的块大于 `size`，会将块分割成大小为 `size` 与 `(block - size)` 的两部分，返回大小为 `size` 的分块，并把大小为 `(block - size)` 的块返回给空闲链表。

不足：

- 标记和清除过程效率都不高；
- 会产生大量不连续的内存碎片，导致无法给大对象分配内存。

2. 标记 - 整理



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。

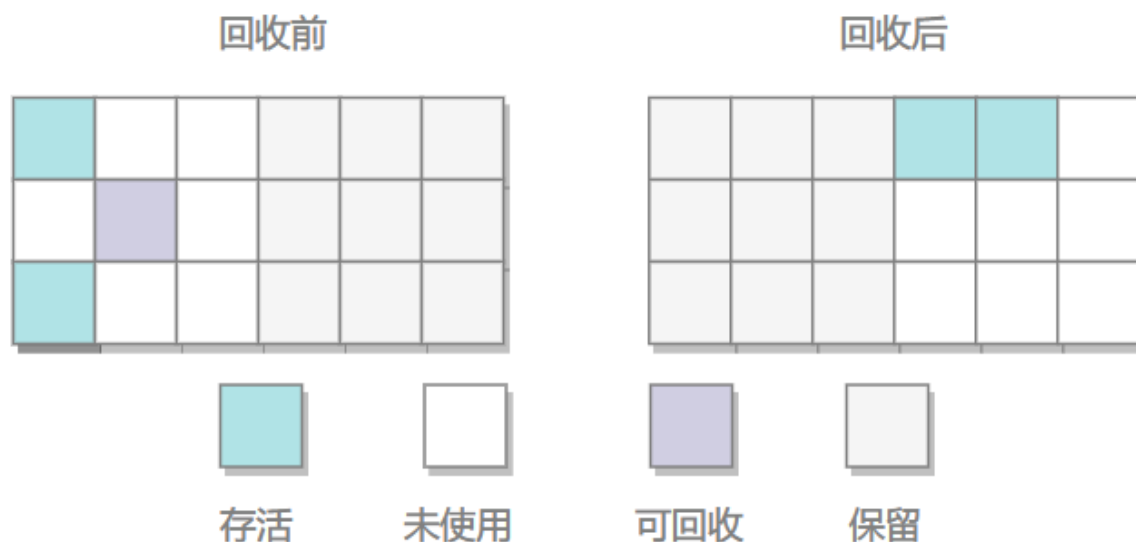
优点：

- 不会产生内存碎片

不足:

- 需要移动大量对象，处理效率比较低。

3. 复制



CyC2018

将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。

主要不足是只使用了内存的一半。

现在的商业虚拟机都采用这种收集算法回收新生代，但是并不是划分为大小相等的两块，而是一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。在回收时，将 Eden 和 Survivor 中还存活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的那一块 Survivor。

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1:1，保证了内存的利用率达到 90%。如果每次回收有多于 10% 的对象存活，那么一块 Survivor 就不够用了，此时需要依赖于老年代进行空间分配担保，也就是借用老年代的空间存储放不下的对象。

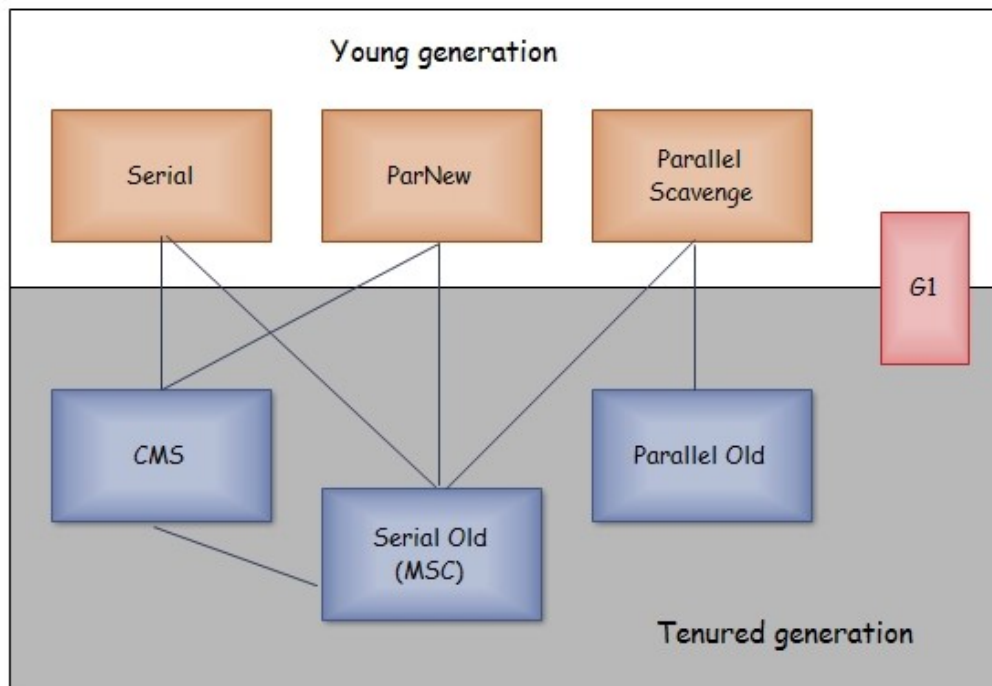
4. 分代收集

现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

一般将堆分为新生代和老年代。

- 新生代使用：复制算法
- 老年代使用：标记 - 清除 或者 标记 - 整理 算法

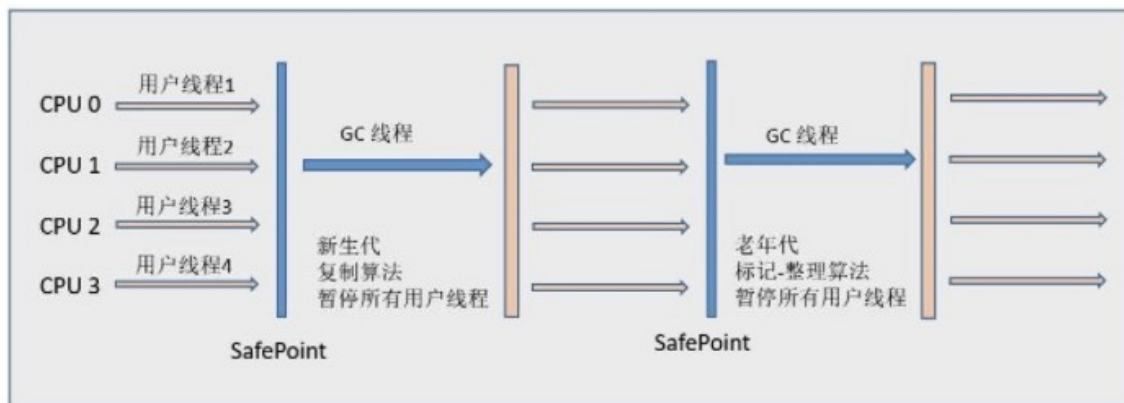
垃圾收集器



以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

- 单线程与多线程：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
- 串行与并行：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；并行指的是垃圾收集器和用户程序同时执行。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

1. Serial 收集器



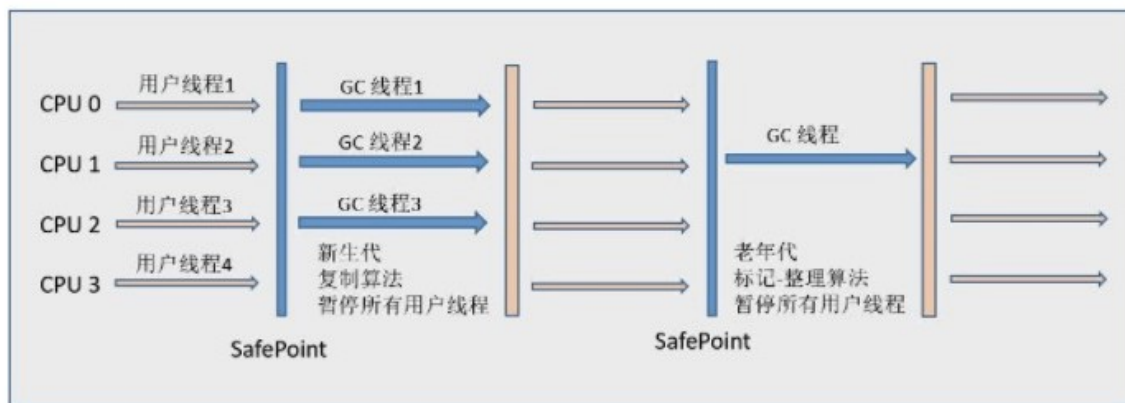
Serial 翻译为串行，也就是说它以串行的方式执行。

它是单线程的收集器，只会使用一个线程进行垃圾收集工作。

它的优点是简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

它是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

2. ParNew 收集器



它是 Serial 收集器的多线程版本。

它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用。

3. Parallel Scavenge 收集器

与 ParNew 一样是多线程收集器。

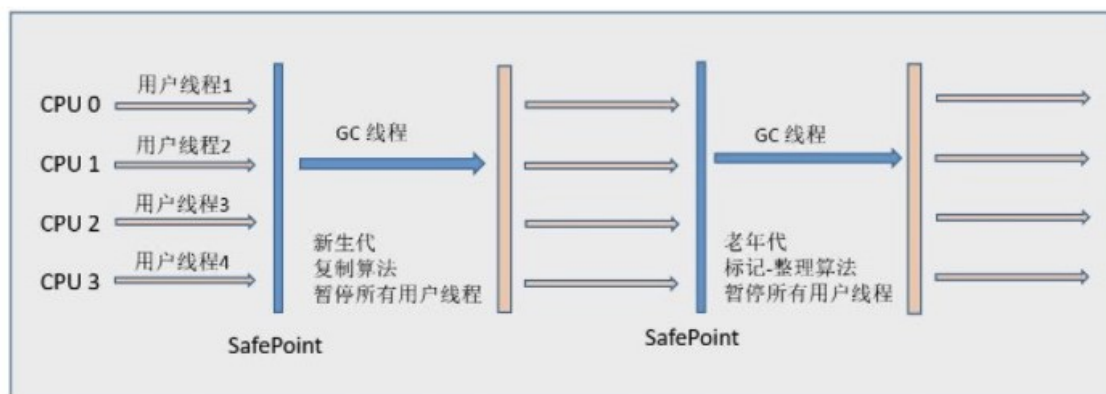
其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量，因此它被称为“吞吐量优先”收集器。这里的吞吐量指 CPU 用于运行用户程序的时间占总时间的比值。

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，适合在后台运算而不需要太多交互的任务。

缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。

可以通过一个开关参数打开 GC 自适应的调节策略（GC Ergonomics），就不需要手工指定新生代的大小（-Xmn）、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。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。

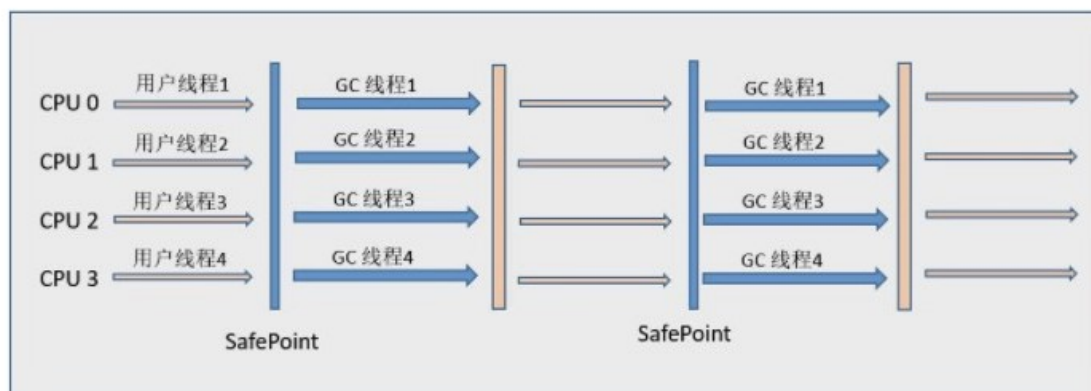
4. Serial Old 收集器



是 Serial 收集器的老年代版本，也是给 Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大大用途：

- 在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。
- 作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

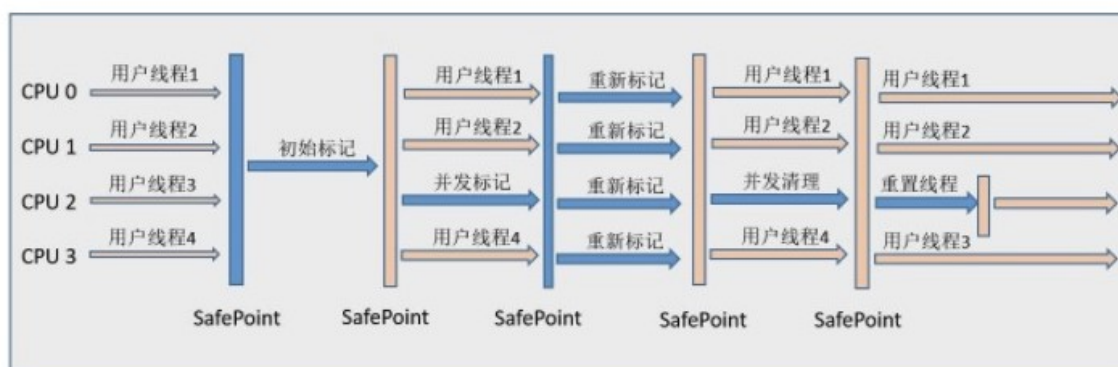
5. Parallel Old 收集器



是 Parallel Scavenge 收集器的老年代版本。

在注重吞吐量以及 CPU 资源敏感的场所，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

6. CMS 收集器



CMS (Concurrent Mark Sweep) , Mark Sweep 指的是标记 - 清除算法。

分为以下四个流程：

- 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
- 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
- 重新标记：为了修正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对象的标记记录，需要停顿。
- 并发清除：不需要停顿。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

具有以下缺点：

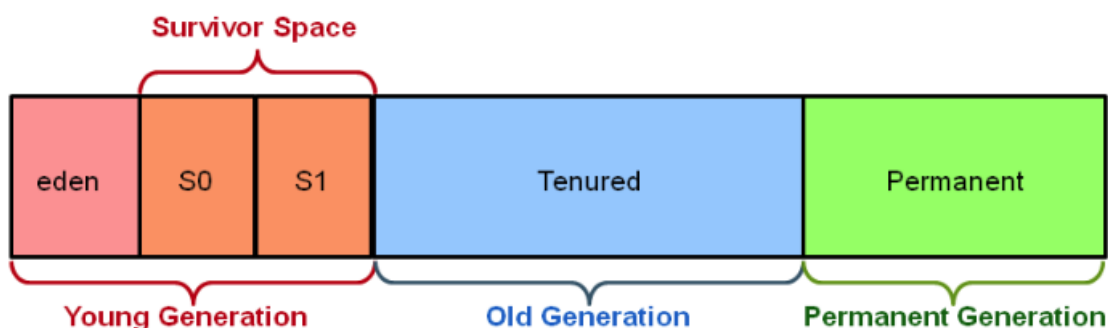
- 吞吐量低：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。
- 无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。
- 标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

7. G1 收集器

G1 (Garbage-First) , 它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

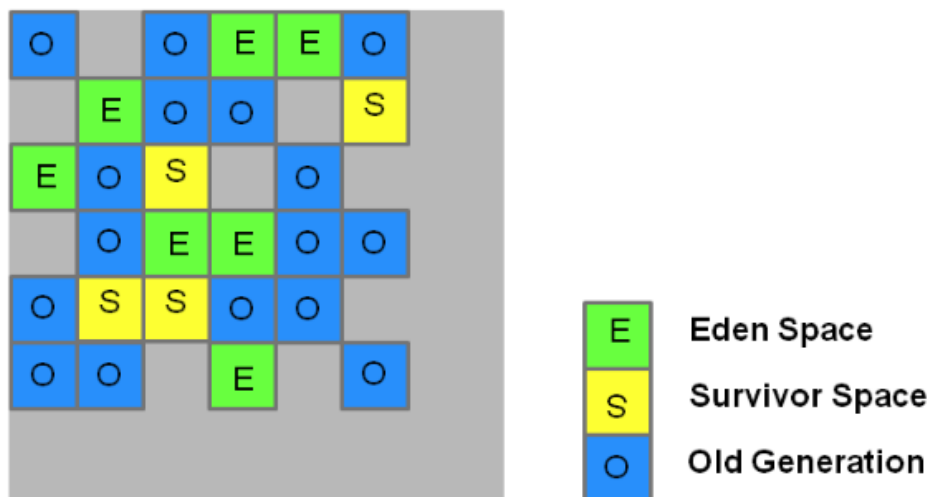
堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

Hotspot Heap Structure



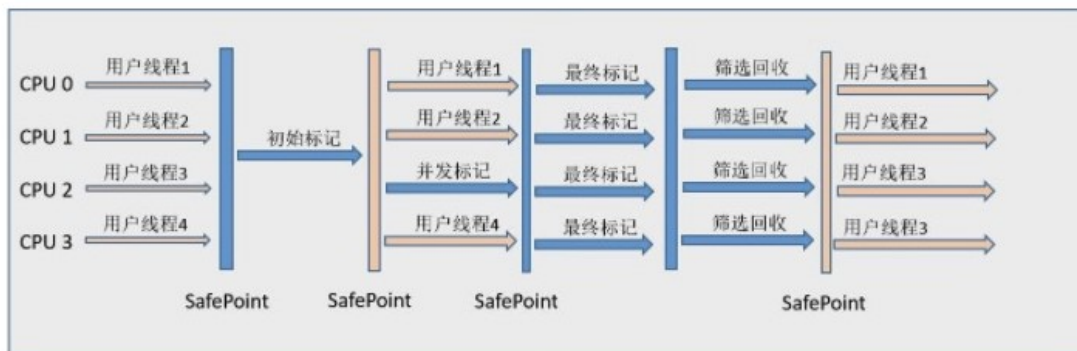
G1 把堆划分成多个大小相等的独立区域 (Region)，新生代和老年代不再物理隔离。

G1 Heap Allocation



通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记：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的 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。这阶段需要停顿线程，但是可并行执行。
- 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

具备如下特点：

- 空间整合：整体来看是基于“标记 - 整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。
- 可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒。

内存分配与回收策略

Minor GC 和 Full GC

- Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

内存分配策略

1. 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

2. 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

3. 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

4. 动态对象年龄判定

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

5. 空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

Full GC 的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

1. 调用 System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

2. 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。

为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

3. 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，如果担保失败会执行一次 Full GC。

4. JDK 1.7 及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 `java.lang.OutOfMemoryError`。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

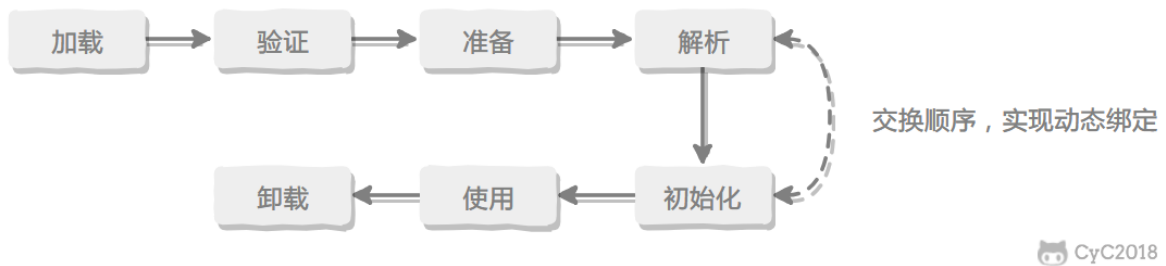
5. Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

类加载机制

类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。因为如果一次性加载，那么会占用很多的内存。

类的生命周期



包括以下 7 个阶段：

- 加载 (Loading)
- 验证 (Verification)
- 准备 (Preparation)
- 解析 (Resolution)
- 初始化 (Initialization)
- 使用 (Using)
- 卸载 (Unloading)

类加载过程

包含了加载、验证、准备、解析和初始化这 5 个阶段。

1. 加载

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

- 通过类的完全限定名称获取定义该类的二进制字节流。
- 将该字节流表示的静态存储结构转换为方法区的运行时存储结构。
- 在内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。

其中二进制字节流可以从以下方式中获取：

- 从 ZIP 包读取，成为 JAR、EAR、WAR 格式的基础。
- 从网络中获取，最典型的应用是 Applet。
- 运行时计算生成，例如动态代理技术，在 `java.lang.reflect.Proxy` 使用 `ProxyGenerator.generateProxyClass` 的代理类的二进制字节流。
- 由其他文件生成，例如由 JSP 文件生成对应的 Class 类。

2. 验证

确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

3. 准备

类变量是被 `static` 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存。

实例变量不会在这阶段分配内存，它会在对象实例化时随着对象一起被分配在堆中。应该注意到，实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

```
public static int value = 123;
```

如果类变量是常量，那么它将初始化为表达式所定义的值而不是 0。例如下面的常量 value 被初始化为 123 而不是 0。

```
public static final int value = 123;
```

4. 解析

将常量池的符号引用替换为直接引用的过程。

其中解析过程在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的动态绑定。

5. 初始化

初始化阶段才真正开始执行类中定义的 Java 程序代码。初始化阶段是虚拟机执行类构造器 () 方法的过程。在准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。

() 是由编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序由语句在源文件中出现的顺序决定。特别注意的是，静态语句块只能访问到定义在它之前的类变量，定义在它之后的类变量只能赋值，不能访问。例如以下代码：

```
public class Test {
    static {
        i = 0;           // 给变量赋值可以正常编译通过
        System.out.print(i); // 这句编译器会提示“非法向前引用”
    }
    static int i = 1;
}
```

由于父类的 () 方法先执行，也就意味着父类中定义的静态语句块的执行要优先于子类。例如以下代码：

```
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}

static class Sub extends Parent {
    public static int B = A;
}

public static void main(String[] args) {
    System.out.println(Sub.B); // 2
}
```


接口中不可以使用静态语句块，但仍然有类变量初始化的赋值操作，因此接口与类一样都会生成 () 方法。但接口与类不同的是，执行接口的 () 方法不需要先执行父接口的 () 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 () 方法。

虚拟机会保证一个类的 () 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 () 方法，其它线程都会阻塞等待，直到活动线程执行 () 方法完毕。如果在一个类的 () 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

类与类加载器

两个类相等，需要类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。

这里的相等，包括类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为 true，也包括使用 instanceof 关键字做对象所属关系判定结果为 true。

类加载器分类

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

- 启动类加载器 (Bootstrap ClassLoader)，使用 C++ 实现，是虚拟机自身的一部分；
- 所有其它类的加载器，使用 Java 实现，独立于虚拟机，继承自抽象类 java.lang.ClassLoader。

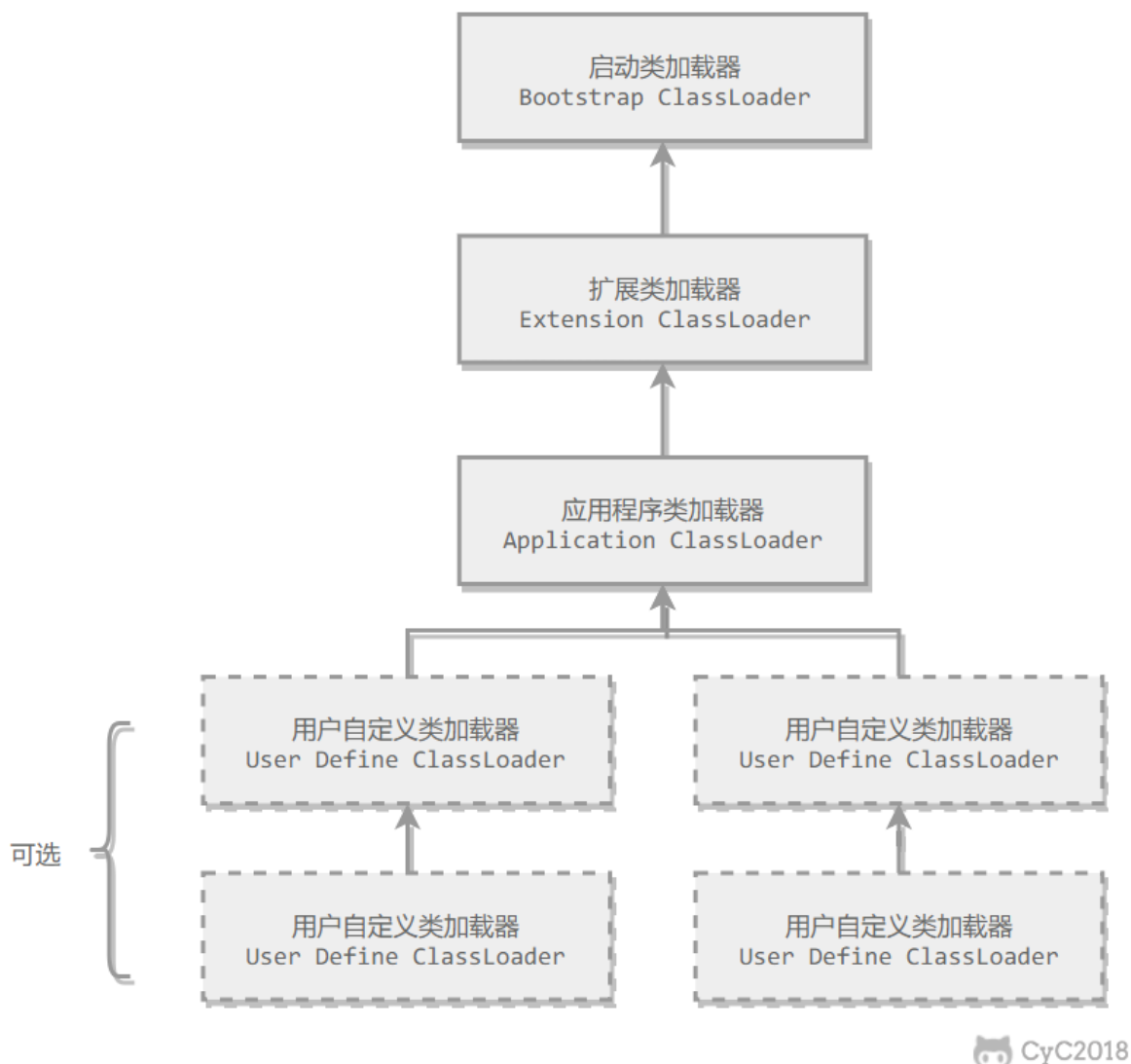
从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

- 启动类加载器 (Bootstrap ClassLoader) 此类加载器负责将存放在 <JRE_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可。
- 扩展类加载器 (Extension ClassLoader) 这个类加载器是由 ExtClassLoader (sun.misc.Launcher\$ExtClassLoader) 实现的。它负责将 <JAVA_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
- 应用程序类加载器 (Application ClassLoader) 这个类加载器是由 AppClassLoader (sun.misc.Launcher\$AppClassLoader) 实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径 (ClassPath) 上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认类加载器。

双亲委派模型

应用程序是由三种类加载器互相配合从而实现类加载，除此之外还可以加入自己定义类加载器。

下图展示了类加载器之间的层次关系，称为双亲委派模型 (Parents Delegation Model)。该模型要求除了顶层的启动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系 (Composition) 来实现，而不是继承关系 (Inheritance)。



1. 工作过程

一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。

2. 好处

使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。

例如 `java.lang.Object` 存放在 `rt.jar` 中，如果编写另外一个 `java.lang.Object` 并放到 `ClassPath` 中，程序可以编译通过。由于双亲委派模型的存在，所以在 `rt.jar` 中的 `Object` 比在 `ClassPath` 中的 `Object` 优先级更高，这是因为 `rt.jar` 中的 `Object` 使用的是启动类加载器，而 `ClassPath` 中的 `Object` 使用的是应用程序类加载器。`rt.jar` 中的 `Object` 优先级更高，那么程序中所有的 `Object` 都是这个 `Object`。

3. 实现

以下是抽象类 `java.lang.ClassLoader` 的代码片段，其中的 `loadClass()` 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 `ClassNotFoundException`，此时尝试自己去加载。

```
public abstract class ClassLoader {
    // The parent class loader for delegation
    private final ClassLoader parent;

    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }
}
```

```

    protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    c = findClass(name);
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
}

```

破坏双亲委派模型

双亲委派模型主要出现过3次较大的“被破坏”情况：

- **双亲委派模型出现之前——即JDK1.2发布之前**
 - 双亲委派模型在JDK1.2之后才被引入，类加载器和抽象类java.lang.ClassLoader在JDK1.0时代就已存在，面对已经存在的用户自定义类加载器的实现代码，Java设计者引入双亲委派模型时不得不做一些妥协。
 - 考虑向前兼容，JDK1.2之后的java.lang.ClassLoader中添加了一个新的方法findClass()。
 - JDK1.2之后不提倡用户覆盖loadClass()方法，而是把加载逻辑写到findClass()中，在loadClass()方法逻辑里如果父类加载失败，则调用findClass()方法加载，如此即符合双亲委派。
- **双亲委派模型自身缺陷**
 - 越基础的类越由上层加载器加载，若基础类需要调用回用户的代码，如JNI。
 - 线程上下文加载器（Thread Context ClassLoader）。这个类加载器可以通过java.lang.Thread类的setContextClassLoader()方法进行设置。如果创建线程时还未设置，将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置，这个类加载器默认就是应用程序类加载器。
- **用户对程序动态性的追求导致**

- OSGi环境下，类加载器不再是双亲委派模型的树状结构，而是进一步发展为网状结构。

自定义类加载器实现

以下代码中的 `FileSystemClassLoader` 是自定义类加载器，继承自 `java.lang.ClassLoader`，用于加载文件系统上的类。它首先根据类的全名在文件系统中查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 `defineClass()` 方法来把这些字节代码转换成 `java.lang.Class` 类的实例。

`java.lang.ClassLoader` 的 `loadClass()` 实现了双亲委派模型的逻辑，自定义类加载器一般不去重写它，但是需要重写 `findClass()` 方法。

```
public class FileSystemClassLoader extends ClassLoader {

    private String rootDir;

    public FileSystemClassLoader(String rootDir) {
        this.rootDir = rootDir;
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classData = getClassData(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
            return defineClass(name, classData, 0, classData.length);
        }
    }

    private byte[] getClassData(String className) {
        String path = classNameToPath(className);
        try {
            InputStream ins = new FileInputStream(path);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int bufferSize = 4096;
            byte[] buffer = new byte[bufferSize];
            int bytesNumRead;
            while ((bytesNumRead = ins.read(buffer)) != -1) {
                baos.write(buffer, 0, bytesNumRead);
            }
            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    private String classNameToPath(String className) {
        return rootDir + File.separatorChar
            + className.replace('.', File.separatorChar) + ".class";
    }
}
```

对象创建过程

Java创建对象的过程



微信公众号：Java面试通关手册

Step1:类加载检查

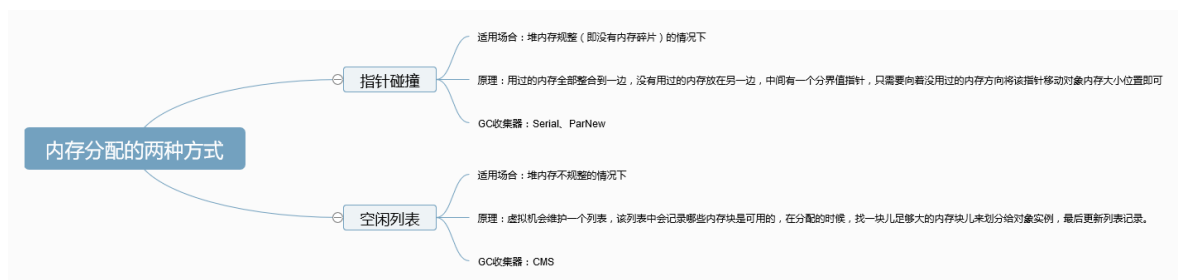
虚拟机遇遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

Step2:分配内存

在**类加载检查**通过后，接下来虚拟机将为新生对象**分配内存**。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。**分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。**

内存分配的两种方式：（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的



- 指针碰撞
 - 适用场合：堆内存规整（即没有垃圾碎片）的情况下
 - 原理：用过的内存全部整合到一边，没有用过的内存放到一边，中间有一个分界值指针，只需要想着没用过的内存方向将该指针移动到对象内存大小位置即可
 - GC收集器：Serial、ParNew
- 空闲列表
 - 适用场合：堆内存不规整的情况下
 - 原理：虚拟机会维护一个列表，该列表会记录哪些存储块是可用的，在分配的时候，找一块足够大的内存块划分给对象实例，最后更新列表记录
 - GC收集器：CMS

内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试**：CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**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。**

- **TLAB**：为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

Step3:初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

Step4:设置对象头

初始化零值完成之后，**虚拟机要对对象进行必要的设置**，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。**这些信息存放在对象头中**。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

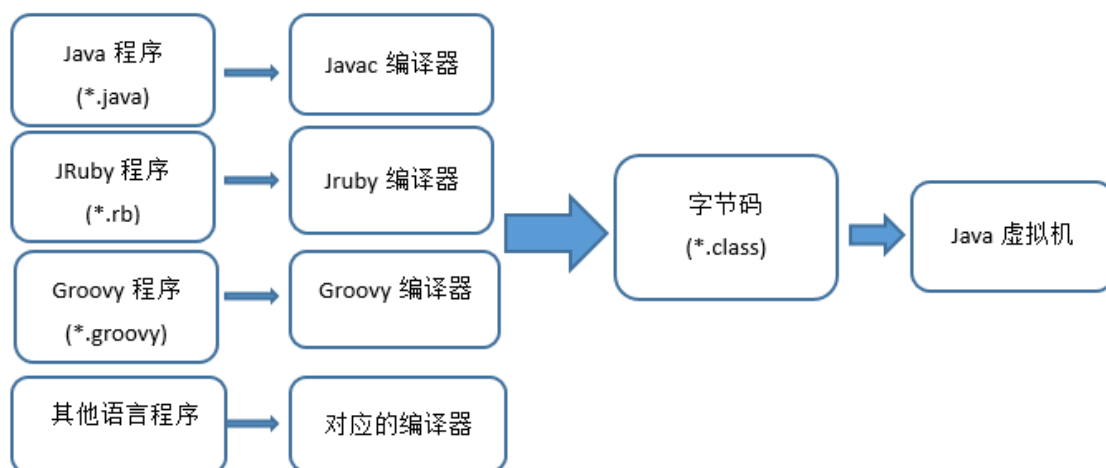
Step5:执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 `new` 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

字节码

在 Java 中，JVM 可以理解的代码就叫做 **字节码**（即扩展名为 `.class` 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Clojure（Lisp 语言的一种方言）、Groovy、Scala 等语言都是运行在 Java 虚拟机之上。下图展示了不同的语言被不同的编译器编译成 `.class` 文件最终运行在 Java 虚拟机之上。`.class` 文件的二进制格式可以使用 [WinHex](#) 查看。



可以说.class文件是不同的语言在 Java 虚拟机之间的重要桥梁，同时也是支持 Java 跨平台很重要的一个原因。

JDK监控和故障处理工具总结

JDK 命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

- **jps** (JVM Process Status) : 类似 UNIX 的 `ps` 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息;
- **jstat** (JVM Statistics Monitoring Tool) : 用于收集 HotSpot 虚拟机各方面的运行数据;
- **jinfo** (Configuration Info for Java) : Configuration Info for Java, 显示虚拟机配置信息;
- **jmap** (Memory Map for Java) : 生成堆转储快照;
- **jhat** (JVM Heap Dump Browser) : 用于分析 heapdump 文件, 它会建立一个 HTTP/HTML 服务器, 让用户可以在浏览器上查看分析结果;
- **jstack** (Stack Trace for Java) : 生成虚拟机当前时刻的线程快照, 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。

jps: 查看所有 Java 进程

`jps` (JVM Process Status) 命令类似 UNIX 的 `ps` 命令。

`jps` : 显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID (Local Virtual Machine Identifier, LVMID) 。 `jps -q` : 只输出进程的本地虚拟机唯一 ID。

```
C:\Users\SnailClimb>jps
7360 NettyClient2
17396
7972 Launcher
16504 Jps
17340 NettyServer
```

`jps -l` : 输出主类的全名, 如果进程执行的是 Jar 包, 输出 Jar 路径。

```
C:\Users\SnailClimb>jps -l
7360 firstNettyDemo.NettyClient2
17396
7972 org.jetbrains.jps.cmdline.Launcher
16492 sun.tools.jps.Jps
17340 firstNettyDemo.NettyServer
```

`jps -v` : 输出虚拟机进程启动时 JVM 参数。

`jps -m` : 输出传递给 Java 进程 `main()` 函数的参数。

jstat: 监视虚拟机各种运行状态信息

`jstat` (JVM Statistics Monitoring Tool) 使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 (需要远程主机提供 RMI 支持) 虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据, 在没有 GUI, 只提供了纯文本控制台环境的服务器上, 它将是运行期间定位虚拟机性能问题的首选工具。

jstat 命令使用格式:

```
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

比如 `jstat -gc -h3 31736 1000 10` 表示分析进程 id 为 31736 的 gc 情况, 每隔 1000ms 打印一次记录, 打印 10 次停止, 每 3 行后打印指标头部。

常见的 option 如下:

- `jstat -class vmid` : 显示 ClassLoader 的相关信息;
- `jstat -compiler vmid` : 显示 JIT 编译的相关信息;
- `jstat -gc vmid` : 显示与 GC 相关的堆信息;
- `jstat -gccapacity vmid` : 显示各个代的容量及使用情况;
- `jstat -gcnew vmid` : 显示新生代信息;
- `jstat -gcnewcapcapacity vmid` : 显示新生代大小与使用情况;
- `jstat -gcolld vmid` : 显示老年代和永久代的信息;
- `jstat -gcolldcapacity vmid` : 显示老年代的大小;
- `jstat -gcpermcapacity vmid` : 显示永久代大小;
- `jstat -gcutil vmid` : 显示垃圾收集信息;

另外, 加上 `-t` 参数可以在输出信息上加一个 Timestamp 列, 显示程序的运行时间。

jinfo: 实时地查看和调整虚拟机各项参数

`jinfo vmid` : 输出当前 jvm 进程的全部参数和系统属性 (第一部分是系统的属性, 第二部分是 JVM 的参数)。

`jinfo -flag name vmid` : 输出对应名称的参数的具体值。比如输出 MaxHeapSize、查看当前 jvm 进程是否开启打印 GC 日志 (`-XX:PrintGCDetails` : 详细 GC 日志模式, 这两个都是默认关闭的)。

```
C:\Users\SnailClimb>jinfo -flag MaxHeapSize 17340
-XX:MaxHeapSize=2124414976

C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:-PrintGC
```

使用 jinfo 可以在不重启虚拟机的情况下, 可以动态的修改 jvm 的参数。尤其在线上的环境特别有用, 请看下面的例子:

`jinfo -flag [+|-]name vmid` 开启或者关闭对应名称的参数。

```
C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:-PrintGC

C:\Users\SnailClimb>jinfo -flag +PrintGC 17340

C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:+PrintGC
```

jmap: 生成堆转储快照

`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, 可以使用 `"-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, Linux 命令下可以通过 `kill -3` 发送进程退出信号也能拿到 dump 文件。

`jmap` 的作用并不仅仅是为了获取 dump 文件, 它还可以查询 finalizer 执行队列、Java 堆和永久代的详细信息, 如空间使用率、当前使用的是哪种收集器等。和 `jinfo` 一样, `jmap` 有不少功能在 Windows 平台下也是受限制的。

示例: 将指定应用程序的堆快照输出到桌面。后面, 可以通过 jhat、Visual VM 等工具分析该堆文件。


```
C:\Users\SnailClimb>jmap -  
dump:format=b,file=C:\Users\SnailClimb\Desktop\heap.hprof 17340  
Dumping heap to C:\Users\SnailClimb\Desktop\heap.hprof ...  
Heap dump file created
```

jhat: 分析 heapdump 文件

jhat 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果。

```
C:\Users\SnailClimb>jhat C:\Users\SnailClimb\Desktop\heap.hprof  
Reading from C:\Users\SnailClimb\Desktop\heap.hprof...  
Dump file created Sat May 04 12:30:31 CST 2019  
Snapshot read, resolving...  
Resolving 131419 objects...  
Chasing references, expect 26 dots.....  
Eliminating duplicate references.....  
Snapshot resolved.  
Started HTTP server on port 7000  
Server is ready.
```

访问 <http://localhost:7000/>

jstack :生成虚拟机当前时刻的线程快照

jstack (Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照。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。

生成线程快照的目的主要是定位线程长时间出现停顿的原因，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，或者在等待些什么资源。

下面是一个线程死锁的代码。我们下面会通过 **jstack** 命令进行死锁检查，输出死锁信息，找到发生死锁的线程。

```
public class DeadLockDemo {  
    private static Object resource1 = new Object();//资源 1  
    private static Object resource2 = new Object();//资源 2  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            synchronized (resource1) {  
                System.out.println(Thread.currentThread() + "get resource1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + "waiting get  
resource2");  
                synchronized (resource2) {  
                    System.out.println(Thread.currentThread() + "get  
resource2");  
                }  
            }  
        })  
    }  
}
```

```

    }, "线程 1").start();

    new Thread() -> {
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get resource2");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread() + "waiting get
resource1");
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get
resource1");
            }
        }
    }, "线程 2").start();
}
}

```

Output

```

Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000); 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

通过 jstack 命令分析：

```

C:\Users\SnailClimb>jps
13792 KotlinCompileDaemon
7360 NettyClient2
17396
7972 Launcher
8932 Launcher
9256 DeadLockDemo
10764 Jps
17340 NettyServer

C:\Users\SnailClimb>jstack 9256

```

输出的部分内容如下：

```

Found one Java-level deadlock:
=====
"线程 2":
  waiting to lock monitor 0x00000000333e668 (object 0x00000000d5efe1c0, a
java.lang.Object),
  which is held by "线程 1"
"线程 1":

```

```
waiting to lock monitor 0x00000000333be88 (object 0x00000000d5efe1d0, a
java.lang.Object),
which is held by "线程 2"
```

Java stack information for the threads listed above:

```
=====
"线程 2":
    at DeadLockDemo.lambda$main$1(DeadLockDemo.java:31)
    - waiting to lock <0x00000000d5efe1c0> (a java.lang.Object)
    - locked <0x00000000d5efe1d0> (a java.lang.Object)
    at DeadLockDemo$$Lambda$2/1078694789.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
"线程 1":
    at DeadLockDemo.lambda$main$0(DeadLockDemo.java:16)
    - waiting to lock <0x00000000d5efe1d0> (a java.lang.Object)
    - locked <0x00000000d5efe1c0> (a java.lang.Object)
    at DeadLockDemo$$Lambda$1/1324119927.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
```

Found 1 deadlock.

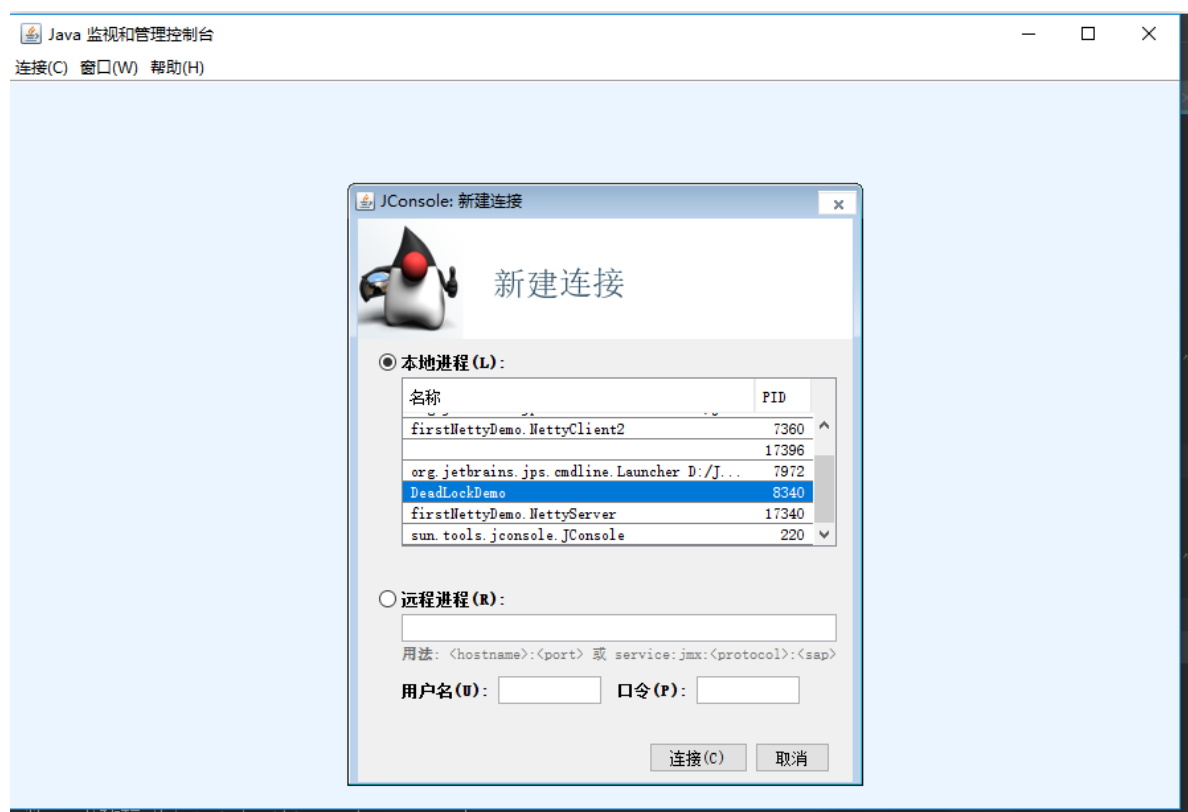
可以看到 `jstack` 命令已经帮我们找到发生死锁的线程的具体信息。

JDK 可视化分析工具

JConsole:Java 监视与管理控制台

JConsole 是基于 JMX 的可视化监视、管理工具。可以很方便的监视本地及远程服务器的 java 进程的内存使用情况。你可以在控制台输出 `console` 命令启动或者在 JDK 目录下的 `bin` 目录找到 `jconsole.exe` 然后双击启动。

连接 Jconsole



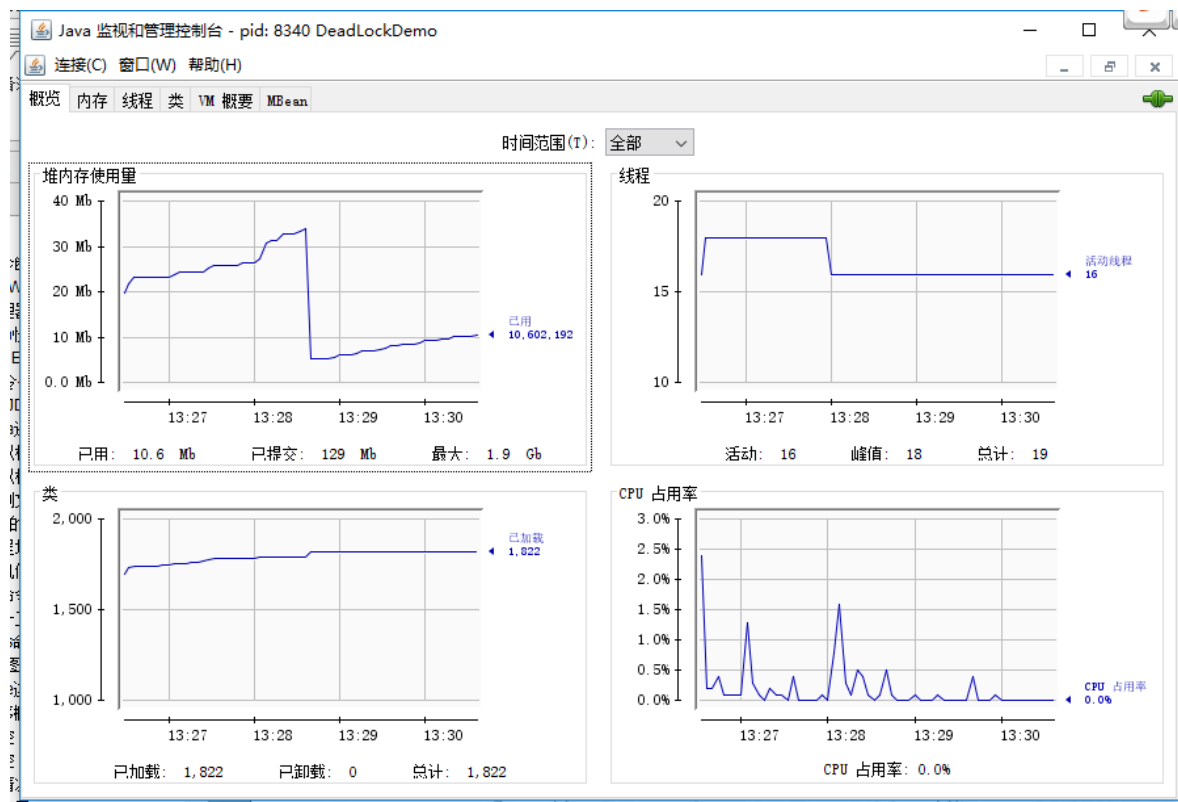
如果需要使用 JConsole 连接远程进程，可以在远程 Java 程序启动时加上下面这些参数：

```
-Djava.rmi.server.hostname=外网访问 ip 地址
-Dcom.sun.management.jmxremote.port=60001 //监控的端口号
-Dcom.sun.management.jmxremote.authenticate=false //关闭认证
-Dcom.sun.management.jmxremote.ssl=false
```

在使用 JConsole 连接时，远程进程地址如下：

外网访问 ip 地址:60001

查看 Java 程序概况

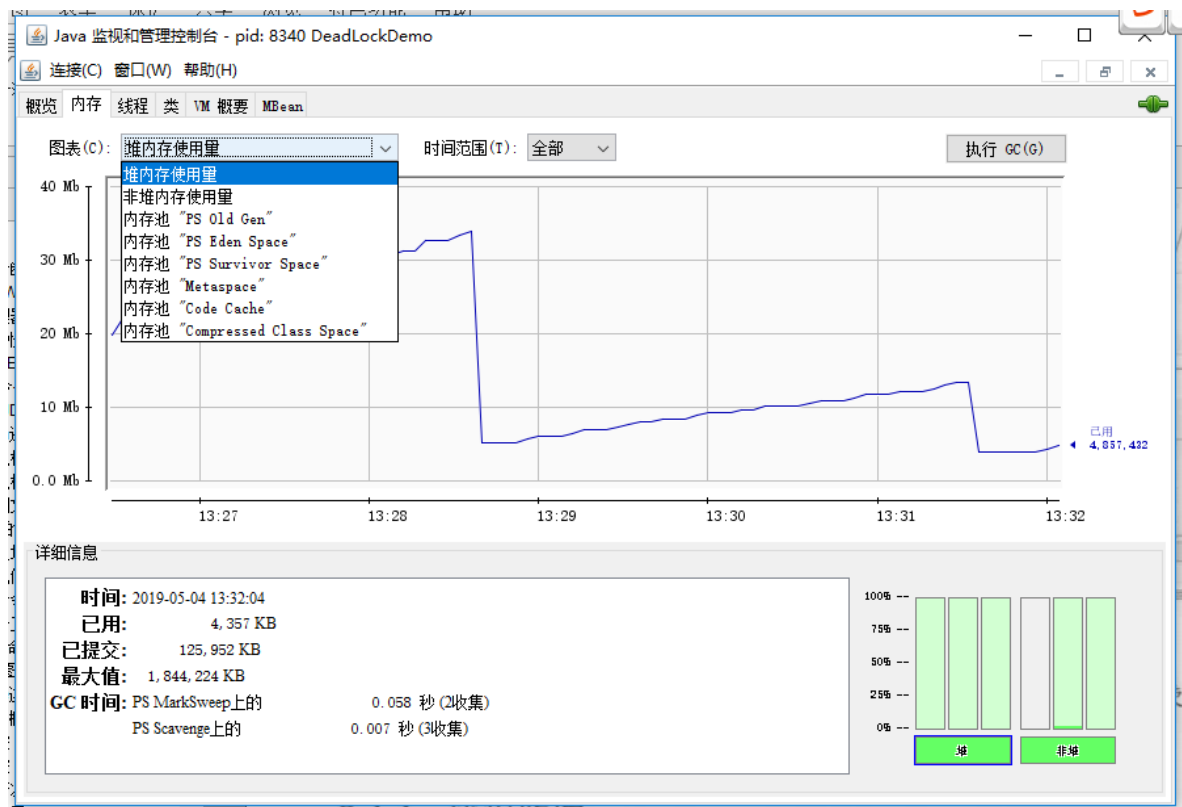


内存监控

JConsole 可以显示当前内存的详细信息。不仅包括堆内存/非堆内存的整体信息，还可以细化到 eden 区、survivor 区等的使用情况，如下图所示。

点击右边的“执行 GC(G)”按钮可以强制应用程序执行一个 Full GC。

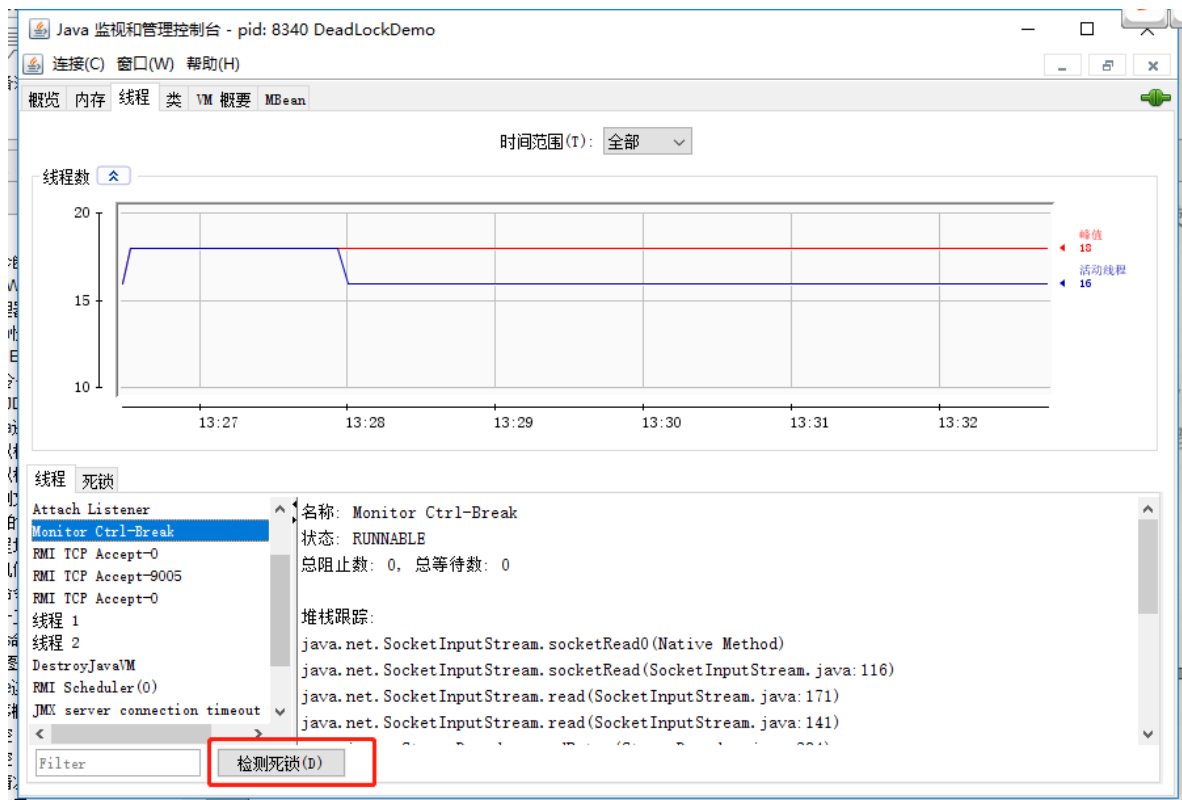
- **新生代 GC (Minor GC)** :指发生新生代的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- **老年代 GC (Major GC/Full GC)** :指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC (并非绝对)，Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。



线程监控

类似我们前面讲的 `jstack` 命令，不过这个是可视化的。

最下面有一个"检测死锁(D)"按钮，点击这个按钮可以自动为你找到发生死锁的线程以及它们的详细信息。



Visual VM:多合一故障处理工具

VisualVM 提供在 Java 虚拟机 (Java Virtual Machine, JVM) 上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中, 您可以方便、快捷地查看多个 Java 应用程序的相关信息。Visual VM 官网: <https://visualvm.github.io/>。Visual VM 中文文档:<https://visualvm.github.io/documentation.html>。

下面这段话摘自《深入理解 Java 虚拟机》。

VisualVM (All-in-One Java Troubleshooting Tool) 是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序, 官方在 VisualVM 的软件说明中写上了“All-in-One”的描述字样, 预示着他除了运行监视、故障处理外, 还提供了很多其他方面的功能, 如性能分析 (Profiling)。VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少, 而且 VisualVM 还有一个很大的优点: 不需要被监视的程序基于特殊 Agent 运行, 因此他对应用程序的实际性能的影响很小, 使得他可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

VisualVM 基于 NetBeans 平台开发, 因此他一开始就具备了插件扩展功能的特性, 通过插件扩展支持, VisualVM 可以做到:

- 显示虚拟机进程以及进程的配置、环境信息 (jps、jinfo)。
- 监视应用程序的 CPU、GC、堆、方法区以及线程的信息 (jstat、jstack)。
- dump 以及分析堆转储快照 (jmap、jhat)。
- 方法级的程序运行性能分析, 找到被调用最多、运行时间最长的方法。
- 离线程序快照: 收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照, 可以将快照发送开发者处进行 Bug 反馈。
- 其他 plugins 的无限的可能性.....

这里就不具体介绍 VisualVM 的使用, 如果想了解的话可以看:

- <https://visualvm.github.io/documentation.html>
- <https://www.ibm.com/developerworks/cn/java/j-lo-visualvm/index.html>

系统CPU飙高和GC频繁，如何排查

- 通过 `top` 命令查看CPU情况, 如果CPU比较高, 则通过 `top -Hp <pid>` 命令查看当前进程的各个线程运行情况, 找出CPU过高的线程之后, 将其线程id转换为十六进制的表现形式, 然后在jstack日志中查看该线程主要在进行的工作。这里又分为两种情况
- 如果是正常的用户线程, 则通过该线程的堆栈信息查看其具体是在哪处用户代码处运行比较消耗CPU;
- 如果该线程是 VM Thread, 则通过 `jstat -gcutil <pid> <period> <times>` 命令监控当前系统的GC状况, 然后通过 `jmap dump:format=b,file=<filepath> <pid>` 导出系统当前的内存数据。导出之后将内存情况放到eclipse的mat工具中进行分析即可得出内存中主要是什么对象比较消耗内存, 进而可以处理相关代码;
- 如果通过 `top` 命令看到CPU并不高, 并且系统内存占用率也比较低。此时就可以考虑是否是由于另外三种情况导致的问题。具体的可以根据具体情况分析:
- 如果是接口调用比较耗时, 并且是不定时出现, 则可以通过压测的方式加大阻塞点出现的频率, 从而通过 jstack 查看堆栈信息, 找到阻塞点;
- 如果是某个功能突然出现停滞的状况, 这种情况也无法复现, 此时可以通过多次导出 jstack 日志的方式对比哪些用户线程是一直都处于等待状态, 这些线程就是可能存在问题的线程;
- 如果通过 jstack 可以查看到死锁状态, 则可以检查产生死锁的两个线程的具体阻塞点, 从而处理相应的问题。