a Java 并发编程 - 01

Java 并发编程 - 01

一、 同步

1 synchronized 关键字

synchronized 锁什么？锁对象。
可能锁对象包括： *this*， 临界资源对象，*Class* 类对象。

1.1 同步方法

synchronized T methodName(){}
同步方法锁定的是当前对象。当多线程通过同一个对象引用多次调用当前同步方法时，需同步执行。

1.2 同步代码块

同步代码块的同步粒度更加细致，是商业开发中推荐的编程方式。可以定位到具体的同步位置，而不是简单的将方法整体实现同步逻辑。在效率上，相对更高。

1.2.1 锁定临界对象

```
T methodName(){
    synchronized(object){}
}
```

同步代码块在执行时，是锁定 *object* 对象。当多个线程调用同一个方法时，锁定对象不变的情况下，需同步执行。

1.2.2 锁定当前对象

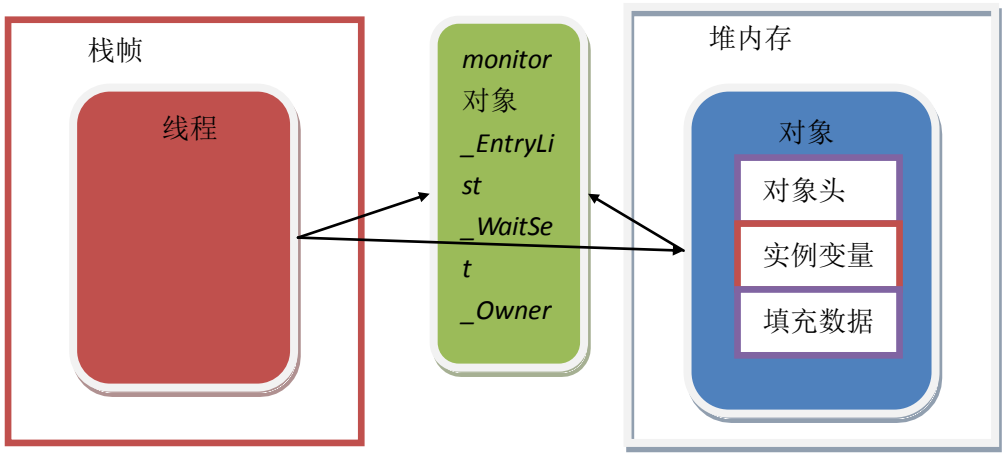
```
T methodName(){
    synchronized(this){}
}
```

当锁定对象为 *this* 时，相当于 *1.1* 同步方法。

1.3 锁的底层实现

Java 虚拟机中的同步*(Synchronization)*基于进入和退出管程*(Monitor)*对象实现。同步方法 并不是由 *monitor enter* 和 *monitor exit* 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 *ACC_SYNCHRONIZED* 标志来隐式实现的。

1.3.1 对象内存简图



对象头：存储对象的 *hashCode*、锁信息或分代年龄或 *GC* 标志，类型指针指向对象的类元数据，*JVM* 通过这个指针确定该对象是哪个类的实例等信息。

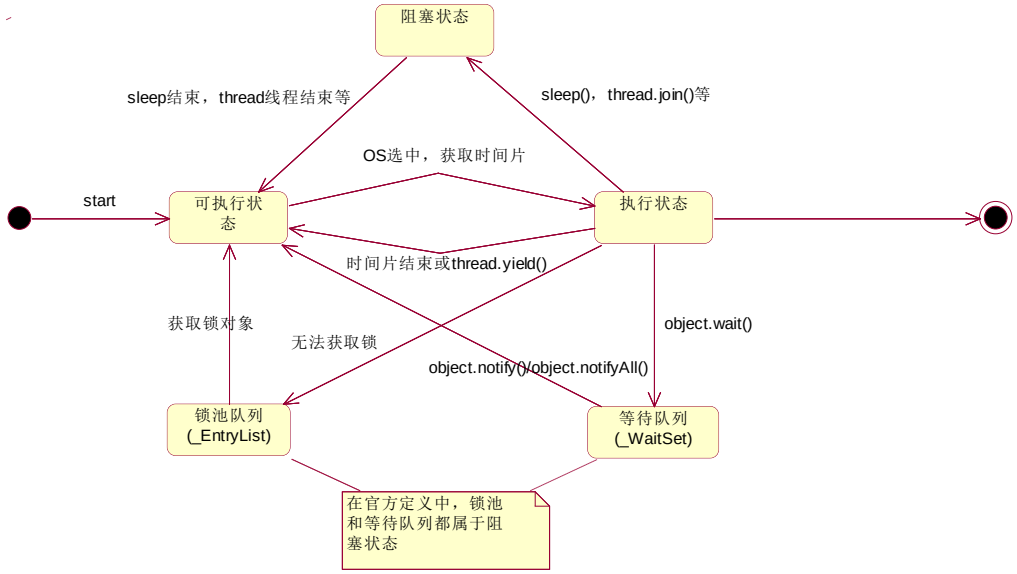
实例变量：存放类的属性数据信息，包括父类的属性信息

填充数据：由于虚拟机要求对象起始地址必须是 *8* 字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐

当在对象上加锁时，数据是记录在对象头中。当执行 *synchronized* 同步方法或同步代码块时，会在对象头中记录锁标记，锁标记指向的是 *monitor* 对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 *monitor* 与之关联，对象与其 *monitor* 之间的关系有存在多种实现方式，如 *monitor* 可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 *monitor* 被某个线程持有后，它便处于锁定状态。

在 *Java* 虚拟机*(HotSpot)*中，*monitor* 是由 *ObjectMonitor* 实现的。

ObjectMonitor 中有两个队列，*_WaitSet* 和 *_EntryList*，以及*_Owner* 标记。其中*_WaitSet* 是用于管理等待队列*(wait)*线程的，*_EntryList* 是用于管理锁池阻塞线程的，*_Owner* 标记用于记录当前执行线程。线程状态图如下：



当多线程并发访问*同一个同步代码*时，首先会进入*_EntryList*，当线程获取锁标记后，*monitor* 中的*_Owner* 记录此线程，并在 *monitor* 中的计数器执行递增计算（*+1*），代表锁定，其他线程在*_EntryList* 中继续阻塞。若执行线程调用 *wait* 方法，则 *monitor* 中的计数器执行赋值为 *0* 计算，并将*_Owner* 标记赋值为 *null*，代表放弃锁，执行线程进如*_WaitSet* 中阻塞。若执行线程调用 *notify/notifyAll* 方法，*_WaitSet* 中的线程被唤醒，进入*_EntryList* 中阻塞，等待获取锁标记。若执行线程的同步代码执行结束，同样会释放锁标记，*monitor* 中的*_Owner* 标记赋值为 *null*，且计数器赋值为 *0* 计算。

1.4 锁的种类

Java 中锁的种类大致分为偏向锁，自旋锁，轻量级锁，重量级锁。

锁的使用方式为：先提供偏向锁，如果不满足的时候，升级为轻量级锁，再不满足，升级为重量级锁。自旋锁是一个过渡的锁状态，不是一种实际的锁类型。

锁只能升级，不能降级。

1.4.1 重量级锁

在 *1.3* 中解释的就是重量级锁。

1.4.2 偏向锁

是一种编译解释锁。如果代码中不可能出现多线程并发争抢同一个锁的时候，*JVM* 编译代码，解释执行的时候，会自动的放弃同步信息。消除 *synchronized* 的同步代码结果。使用锁标记的形式记录锁状态。在 *Monitor* 中有变量 *ACC_SYNCHRONIZED*。当变量值使用的时候，代表偏向锁锁定。可以避免锁的争抢和锁池状态的维护。提高效率。

1.4.3 轻量级锁

过渡锁。当偏向锁不满足，也就是有多线程并发访问，锁定同一个对象的时候，先提升为轻量级锁。也是使用标记 *ACC_SYNCHRONIZED* 标记记录的。*ACC_UNSYNCHRONIZED* 标记记录未获取到锁信息的线程。就是只有两个线程争抢锁标记的时候，优先使用轻量级锁。

两个线程也可能出现重量级锁。

1.4.4 自旋锁

是一个过渡锁，是偏向锁和轻量级锁的过渡。

当获取锁的过程中，未获取到。为了提高效率，*JVM* 自动执行若干次空循环，再次申请锁，而不是进入阻塞状态的情况。称为自旋锁。自旋锁提高效率就是避免线程状态的变更。

2 volatile 关键字

变量的线程可见性。在 *CPU* 计算过程中，会将计算过程需要的数据加载到 *CPU* 计算缓存中，当 *CPU* 计算中断时，有可能刷新缓存，重新读取内存中的数据。在线程运行的过程中，如果某变量被其他线程修改，可能造成数据不一致的情况，从而导致结果错误。而 *volatile* 修饰的变量是线程可见的，当 *JVM* 解释 *volatile* 修饰的变量时，会通知 *CPU*，在计算过程中，每次使用变量参与计算时，都会检查内存中的数据是否发生变化，而不是一直使用 *CPU* 缓存中的数据，可以保证计算结果的正确。

volatile 只是通知底层计算时，*CPU* 检查内存数据，而不是让一个变量在多个线程中同步。

3 wait&notify

4 AtomicXxx 类型组

原子类型。

在 *concurrent.atomic* 包中定义了若干原子类型，这些类型中的每个方法都是保证了原子操作的。多线程并发访问原子类型对象中的方法，不会出现数据错误。在多线程开发中，如果某数据需要多个线程同时操作，且要求计算原子性，可以考虑使用原子类型对象。

注意：原子类型中的方法是保证了原子操作，但多个方法之间是没有原子性的。如：

```
AtomicInteger i = new AtomicInteger(0);
if(i.get() != 5)  i.incrementAndGet();
```

在上述代码中，get 方法和 incrementAndGet 方法都是原子操作，但复合使用时，无法保证原子性，仍旧可能出现数据错误。

5 CountDownLatch 门闩

门闩是 *concurrent* 包中定义的一个类型，是用于多线程通讯的一个辅助类型。

门闩相当于在一个门上加多个锁，当线程调用 *await* 方法时，会检查门闩数量，如果门

闩数量大于 *0*，线程会阻塞等待。当线程调用 *countDown* 时，会递减门闩的数量，当门闩数量为 *0* 时，*await* 阻塞线程可执行。

6 锁的重入

在 *Java* 中，同步锁是可以重入的。只有同一线程调用同步方法或执行同步代码块，对同一个对象加锁时才可重入。

当线程持有锁时，会在 *monitor* 的计数器中执行递增计算，若当前线程调用其他同步代码，且同步代码的锁对象相同时，*monitor* 中的计数器继续递增。每个同步代码执行结束，*monitor* 中的计数器都会递减，直至所有同步代码执行结束，*monitor* 中的计数器为 *0* 时，释放锁标记，*_Owner* 标记赋值为 *null*。

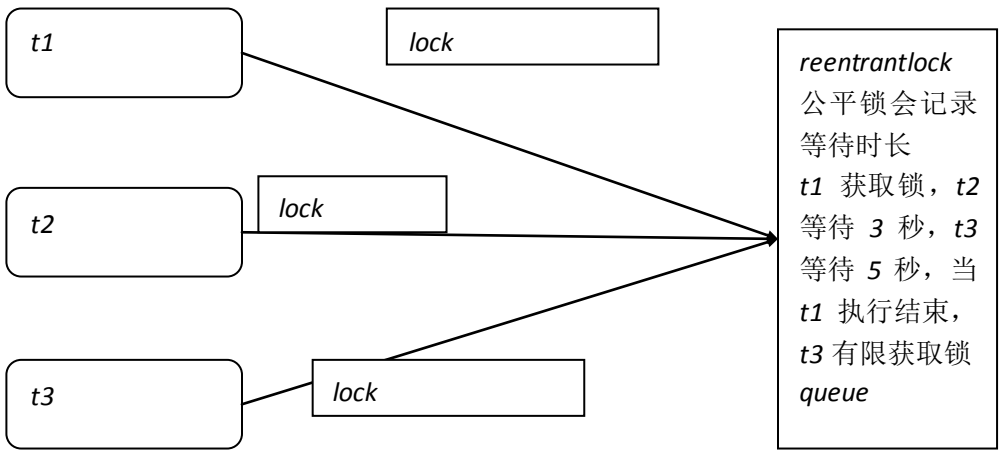
7 ReentrantLock

重入锁，建议应用的同步方式。相对效率比 *synchronized* 高。量级较轻。

synchronized 在 *JDK1.5* 版本开始，尝试优化。到 *JDK1.7* 版本后，优化效率已经非常好了。在绝对效率上，不比 *reentrantLock* 差多少。

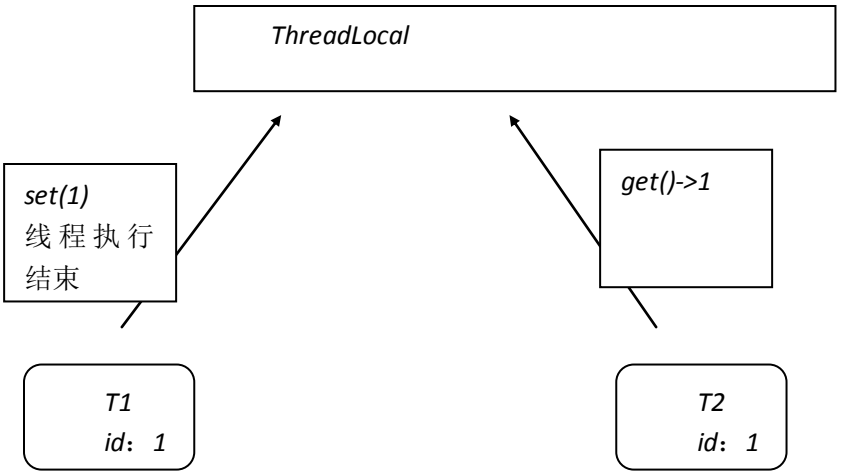
使用重入锁，***必须必须必须***手工释放锁标记。一般都是在 *finally* 代码块中定义释放锁标记的 *unlock* 方法。

7.1 公平锁



8 ThreadLocal

remove 问题



在一个操作系统中，线程和进程是有数量上限的。在操作系统中，确定线程和进程唯一性的唯一条件就是线程或进程 *ID*。

操作系统在回收线程或进程的时候，不是一定杀死线程或进程，在繁忙的时候，只会做情况线程或进程栈数据的操作，重复使用线程或进程

二、 同步容器

解决并发情况下的容器线程安全问题的。给多线程环境准备一个线程安全的容器对象。

线程安全的容器对象： *Vector, Hashtable*。线程安全容器对象，都是使用 *synchronized* 方法实现的。

concurrent 包中的同步容器，大多数是使用系统底层技术实现的线程安全。类似 *native*。*Java8* 中使用 *CAS*。

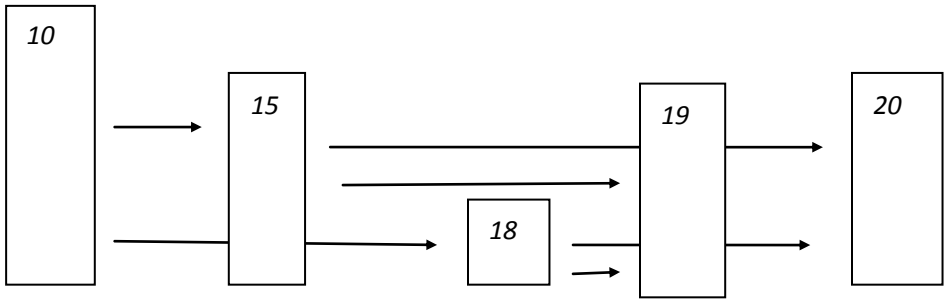
1 Map/Set

1.1 ConcurrentHashMap/ConcurrentHashSet

底层哈希实现的同步 *Map(Set)*。效率高，线程安全。使用系统底层技术实现线程安全。量级较 *synchronized* 低。*key* 和 *value* 不能为 *null*。

1.2 ConcurrentSkipListMap/ConcurrentSkipListSet

底层跳表（*SkipList*）实现的同步 *Map(Set)*。有序，效率比 *ConcurrentHashMap* 稍低。



2 List

2.1 CopyOnWriteArrayList

写时复制集合。写入效率低，读取效率高。每次写入数据，都会创建一个新的底层数组。

3 Queue

3.1 ConcurrentLinkedQueue

基础链表同步队列。

3.2 LinkedBlockingQueue

阻塞队列，队列容量不足自动阻塞，队列容量为 *0* 自动阻塞。

3.3 ArrayBlockingQueue

底层数组实现的有界队列。自动阻塞。根据调用 *API*（*add/put/offer*）不同，有不同特性。

当容量不足的时候，有阻塞能力。

add 方法在容量不足的时候，抛出异常。

put 方法在容量不足的时候，阻塞等待。

offer 方法，

单参数 *offer* 方法，不阻塞。容量不足的时候，返回 *false*。当前新增数据操作放弃。

三参数 *offer* 方法（*offer(value,times,timeunit)*），容量不足的时候，阻塞 *times* 时长（单位为 *timeunit*），如果在阻塞时长内，有容量空闲，新增数据返回 *true*。如果阻塞时长范围内，无容量空闲，放弃新增数据，返回 *false*。

3.4 DelayQueue

延时队列。根据比较机制，实现自定义处理顺序的队列。常用于定时任务。
如：定时关机。

3.5 LinkedTransferQueue

转移队列，使用 *transfer* 方法，实现数据的即时处理。没有消费者，就阻塞。

3.6 SynchronusQueue

同步队列，是一个容量为 *0* 的队列。是一个特殊的 *TransferQueue*。
必须现有消费线程等待，才能使用的队列。
add 方法，无阻塞。若没有消费线程阻塞等待数据，则抛出异常。
put 方法，有阻塞。若没有消费线程阻塞等待数据，则阻塞。

三、 ThreadPool&Executor

1 Executor

线程池顶级接口。定义方法，*void execute(Runnable)*。方法是用于处理任务的一个服务方法。调用者提供 *Runnable* 接口的实现，线程池通过线程执行这个 *Runnable*。服务方法无返回值的。是 *Runnable* 接口中的 *run* 方法无返回值。

常用方法 *- void execute(Runnable)*
作用是： 启动线程任务的。

2 ExecutorService

Executor 接口的子接口。提供了一个新的服务方法，*submit*。有返回值（*Future* 类型）。*submit* 方法提供了 *overload* 方法。其中有参数类型为 *Runnable* 的，不需要提供返回值的；有参数类型为 *Callable*，可以提供线程执行后的返回值。

Future，是 *submit* 方法的返回值。代表未来，也就是线程执行结束后的一种结果。如返回值。

常见方法 *- void execute(Runnable)*， *Future submit(Callable)*， *Future submit(Runnable)*
线程池状态： *Running*， *ShuttingDown*， *Termitnaed*
Running - 线程池正在执行中。活动状态。
ShuttingDown - 线程池正在关闭过程中。优雅关闭。一旦进入这个状态，线程池不再接收新的任务，处理所有已接收的任务，处理完毕后，关闭线程池。
Terminated - 线程池已经关闭。

3 Future

未来结果，代表线程任务执行结束后的结果。获取线程执行结果的方式是通过 *get* 方法

获取的。*get* 无参，阻塞等待线程执行结束，并得到结果。*get* 有参，阻塞固定时长，等待线程执行结束后的结果，如果在阻塞时长范围内，线程未执行结束，抛出异常。

常用方法： *T get()* *T get(long, TimeUnit)*

4 Callable

可执行接口。 类似 *Runnable* 接口。也是可以启动一个线程的接口。其中定义的方法是 *call*。*call* 方法的作用和 *Runnable* 中的 *run* 方法完全一致。*call* 方法有返回值。

接口方法 ： *Object call();*相当于 *Runnable* 接口中的 *run* 方法。区别为此方法有返回值。不能抛出已检查异常。

和 *Runnable* 接口的选择 - 需要返回值或需要抛出异常时，使用 *Callable*，其他情况可任意选择。

5 Executors

工具类型。为 *Executor* 线程池提供工具方法。可以快速的提供若干种线程池。如：固定容量的，无限容量的，容量为 *1* 等各种线程池。

线程池是一个进程级的重量级资源。默认的生命周期和 *JVM* 一致。当开启线程池后，直到 *JVM* 关闭为止，是线程池的默认生命周期。如果手工调用 *shutdown* 方法，那么线程池执行所有的任务后，自动关闭。

开始 - 创建线程池。

结束 -*JVM* 关闭或调用 *shutdown* 并处理完所有的任务。

类似 *Arrays*，*Collections* 等工具类型的功用。

6 FixedThreadPool

容量固定的线程池。活动状态和线程池容量是有上限的线程池。所有的线程池中，都有一个任务队列。使用的是 *BlockingQueue<Runnable>*作为任务的载体。当任务数量大于线程池容量的时候，没有运行的任务保存在任务队列中，当线程有空闲的，自动从队列中取出任务执行。

使用场景： 大多数情况下，使用的线程池，首选推荐 *FixedThreadPool*。*OS* 系统和硬件是有线程支持上限。不能随意的无限制提供线程池。

线程池默认的容量上限是 *Integer.MAX_VALUE*。

常见的线程池容量： *PC - 200*。 服务器 *- 1000~10000*

queued tasks - 任务队列

completed tasks - 结束任务队列

7 CachedThreadPool

缓存的线程池。容量不限（*Integer.MAX_VALUE*）。自动扩容。容量管理策略：如果线程池中的线程数量不满足任务执行，创建新的线程。每次有新任务无法即时处理的时候，都会创建新的线程。当线程池中的线程空闲时长达到一定的临界值（默认 *60* 秒），自动释放线程。

默认线程空闲 *60* 秒，自动销毁。

应用场景： 内部应用或测试应用。 内部应用，有条件的内部数据瞬间处理时应用，如：

电信平台夜间执行数据整理（有把握在短时间内处理完所有工作，且对硬件和软件有足够的信心）。 测试应用，在测试的时候，尝试得到硬件或软件的最高负载量，用于提供 *FixedThreadPool* 容量的指导。

8 ScheduledThreadPool

计划任务线程池。可以根据计划自动执行任务的线程池。

scheduleAtFixedRate(Runnable, start_limit, limit, timeunit)

runnable - 要执行的任务。

start_limit - 第一次任务执行的间隔。

limit - 多次任务执行的间隔。

timeunit - 多次任务执行间隔的时间单位。

使用场景： 计划任务时选用（*DelaydQueue*），如：电信行业中的数据整理，没分钟整理，没消失整理，每天整理等。

9 SingleThreadExceutor

单一容量的线程池。

使用场景： 保证任务顺序时使用。如： 游戏大厅中的公共频道聊天。秒杀。

10 ForkJoinPool

分支合并线程池（*mapduce* 类似的设计思想）。适合用于处理复杂任务。

初始化线程容量与 *CPU* 核心数相关。

线程池中运行的内容必须是 *ForkJoinTask* 的子类型（*RecursiveTask,RecursiveAction*）。

ForkJoinPool - 分支合并线程池。 可以递归完成复杂任务。要求可分支合并的任务必须是 *ForkJoinTask* 类型的子类型。其中提供了分支和合并的能力。*ForkJoinTask* 类型提供了两个抽象子类型，*RecursiveTask* 有返回结果的分支合并任务,*RecursiveAction* 无返回结果的分支合并任务。（*Callable/Runnable*）*compute* 方法：就是任务的执行逻辑。

ForkJoinPool 没有所谓的容量。默认都是 *1* 个线程。根据任务自动的分支新的子线程。当子线程任务结束后，自动合并。所谓自动是根据 *fork* 和 *join* 两个方法实现的。

应用： 主要是做科学计算或天文计算的。数据分析的。

11 WorkStealingPool

JDK1.8 新增的线程池。工作窃取线程池。当线程池中有空闲连接时，自动到等待队列中窃取未完成任务，自动执行。

初始化线程容量与 CPU 核心数相关。此线程池中维护的是精灵线程。

ExecutorService.newWorkStealingPool();

12 ThreadPoolExecutor

线程池底层实现。除 *ForkJoinPool* 外，其他常用线程池底层都是使用 *ThreadPoolExecutor*

实现的。

public ThreadPoolExecutor

(int corePoolSize, // 核心容量，创建线程池的时候，默认有多少线程。也是线程池保持的最少线程数

int maximumPoolSize, // 最大容量，线程池最多有多少线程

long keepAliveTime, // 生命周期，*0* 为永久。当线程空闲多久后，自动回收。

TimeUnit unit, // 生命周期单位，为生命周期提供单位，如：秒，毫秒

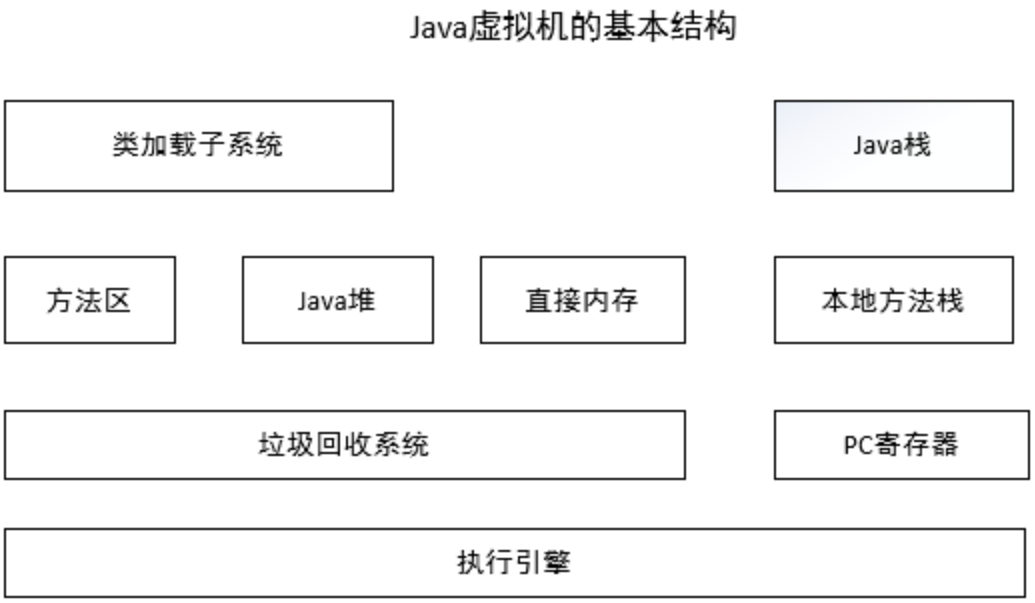
BlockingQueue<Runnable> workQueue // 任务队列，阻塞队列。注意，泛型必须是 *Runnable*

);

使用场景： 默认提供的线程池不满足条件时使用。如：初始线程数据 *4*，最大线程数 *200*，线程空闲周期 *30* 秒。

四、 JVM 优化

1 JVM 简单结构图



1.1 类加载子系统与方法区：

类加载子系统负责从文件系统或者网络中加载 *Class* 信息，加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中可能还会存放运行时常量池信息，包括字符串字面量和数字常量（这部分常量信息是 *Class* 文件中常量池部分的内存映射）。

1.2 Java 堆

java 堆在虚拟机启动的时候建立，它是 *java* 程序最主要的内存工作区域。几乎所有的 *java* 对象实例都存放在 *java* 堆中。堆空间是所有线程共享的，这是一块与 *java* 应用密切相

关的内存空间。

1.3 直接内存

java 的 *NIO* 库允许 *java* 程序使用直接内存。直接内存是在 *java* 堆外的、直接向系统申请的内存空间。通常访问直接内存的速度会优于 *java* 堆。因此出于性能的考虑，读写频繁的场合可能会考虑使用直接内存。由于直接内存在 *java* 堆外，因此它的大小不会直接受限于 *Xmx* 指定的最大堆大小，但是系统内存是有限的，*java* 堆和直接内存的总和依然受限于操作系统能给出的最大内存。

1.4 垃圾回收系统

垃圾回收系统是 *java* 虚拟机的重要组成部分，垃圾回收器可以对方法区、*java* 堆和直接内存进行回收。其中，*java* 堆是垃圾收集器的工作重点。和 *C/C++*不同，*java* 中所有的对象空间释放都是隐式的，也就是说，*java* 中没有类似 *free()*或者 *delete()*这样的函数释放指定的内存区域。对于不再使用的垃圾对象，垃圾回收系统会在后台默默工作，默默查找、标识并释放垃圾对象，完成包括 *java* 堆、方法区和直接内存中的全自动化管理。

1.5 Java 栈

每一个 *java* 虚拟机线程都有一个私有的 *java* 栈，一个线程的 *java* 栈在线程创建的时候被创建，*java* 栈中保存着帧信息，*java* 栈中保存着局部变量、方法参数，同时和 *java* 方法的调用、返回密切相关。

1.6 本地方法栈

本地方法栈和 *java* 栈非常类似，最大的不同在于 *java* 栈用于方法的调用，而本地方法栈则用于本地方法的调用，作为对 *java* 虚拟机的重要扩展，*java* 虚拟机允许 *java* 直接调用本地方法（通常使用 *C* 编写）

1.7 PC 寄存器

PC（*Program Counter*）寄存器也是每一个线程私有的空间，*java* 虚拟机会为每一个 *java* 线程创建 *PC* 寄存器。在任意时刻，一个 *java* 线程总是在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，*PC* 寄存器就会指向当前正在被执行的指令。如果当前方法是本地方法，那么 *PC* 寄存器的值就是 *undefined*

1.8 执行引擎

执行引擎是 *java* 虚拟机的最核心组件之一，它负责执行虚拟机的字节码，现代虚拟机为了提高执行效率，会使用即时编译*(just in time)*技术将方法编译成机器码后再执行。

Java HotSpot Client VM(-client)，为在客户端环境中减少启动时间而优化的执行引擎；

Java HotSpot Server VM(-server)，为在服务器环境中最大化程序执行速度而设计的执行引擎。

Java HotSpot Client 模式和 *Server* 模式的区别

当虚拟机运行在*-client* 模式的时候,使用的是一个代号为 *C1* 的轻量级编译器, 而*-server* 模式启动的虚拟机采用相对重量级,代号为 *C2* 的编译器. *C2* 比 *C1* 编译器编译的相对彻底,服务起来之后,性能更高

注意：在部分 JDK1.6 版本和后续的 JDK 版本(64 位系统)中，-client 参数已经不起作用了，Server 模式成为唯一

2 堆结构及对象分代

2.1 什么是分代，分代的必要性是什么

Java 虚拟机根据对象存活的周期不同，把堆内存划分为几块，一般分为新生代、老年代和永久代（对 HotSpot 虚拟机而言），这就是 JVM 的内存分代策略。

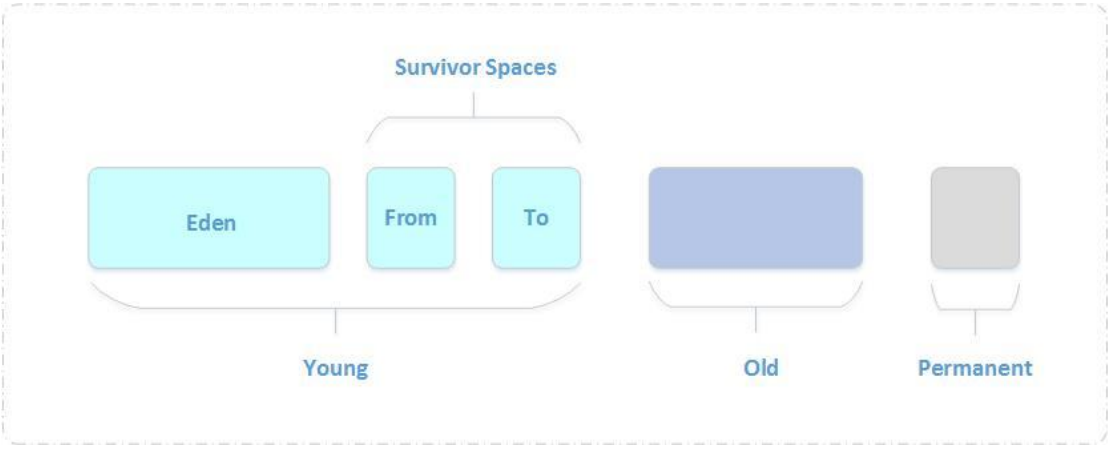
堆内存是虚拟机管理的内存中最大的一块，也是垃圾回收最频繁的一块区域，我们程序所有的对象实例都存放在堆内存中。给堆内存分代是为了提高对象内存分配和垃圾回收的效率。试想一下，如果堆内存没有区域划分，所有的新创建的对象和生命周期很长的对象放在一起，随着程序的执行，堆内存需要频繁进行垃圾收集，而每次回收都要遍历所有的对象，遍历这些对象所花费的时间代价是巨大的，会严重影响我们的 GC 效率。

有了内存分代，情况就不同了，新创建的对象会在新生代中分配内存，经过多次回收仍然存活下来的对象存放在老年代中，静态属性、类信息等存放在永久代中，新生代中的对象存活时间短，只需要在新生代区域中频繁进行 GC，老年代中对象生命周期长，内存回收的频率相对较低，不需要频繁进行回收，永久代中回收效果太差，一般不进行垃圾回收，还可以根据不同年代的特点采用合适的垃圾收集算法。分代收集大大提升了收集效率，这些都是内存分代带来的好处。

2.2 分代的划分

ava 虚拟机将堆内存划分为***新生代、老年代和永久代***，永久代是 *HotSpot* 虚拟机特有的概念，它采用永久代的方式来实现方法区，其他的虚拟机实现没有这一概念，而且 *HotSpot* 也有取消永久代的趋势，在 *JDK 1.7* 中 *HotSpot* 已经开始了“去永久化”，把原本放在永久代的字符串常量池移出。永久代主要存放常量、类信息、静态变量等数据，与垃圾回收关系不大，新生代和老年代是垃圾回收的主要区域。

内存简图如下：



2.2.1 新生代（Young Generation）

新生成的对象优先存放在新生代中，新生代对象朝生夕死，存活率很低，在新生代中，常规应用进行一次垃圾收集一般可以回收 *70%～95%* 的空间，回收效率很高。

HotSpot 将新生代划分为三块，一块较大的 *Eden*（伊甸）空间和两块较小的 *Survivor*（幸存者）空间，默认比例为 *8*：*1*：*1*。划分的目的是因为 *HotSpot* 采用复制算法来回收新生代，设置这个比例是为了充分利用内存空间，减少浪费。新生成的对象在 *Eden* 区分配（大对象除外，大对象直接进入老年代），当 *Eden* 区没有足够的空间进行分配时，虚拟机将发起一次 *Minor GC*。

GC 开始时，对象只会存在于 *Eden* 区和 *From Survivor* 区，*To Survivor* 区是空的（作为保留区域）。*GC* 进行时，*Eden* 区中所有存活的对象都会被复制到 *To Survivor* 区，而在 *From Survivor* 区中，仍存活的对象会根据它们的年龄值决定去向，年龄值达到年龄阀值（默认为 *15*，新生代中的对象每熬过一轮垃圾回收，年龄值就加 *1*，*GC* 分代年龄存储在对象的 *header* 中）的对象会被移到老年代中，没有达到阀值的对象会被复制到 *To Survivor* 区。接着清空 *Eden* 区和 *From Survivor* 区，新生代中存活的对象都在 *To Survivor* 区。接着， *From Survivor* 区和 *To Survivor* 区会交换它们的角色，也就是新的 *To Survivor* 区就是上次 *GC* 清空的 *From Survivor* 区，新的 *From Survivor* 区就是上次 *GC* 的 *To Survivor* 区，总之，不管怎样都会保证 *To Survivor* 区在一轮 *GC* 后是空的。*GC* 时当 *To Survivor* 区没有足够的空间存放上一次新生代收集下来的存活对象时，需要依赖老年代进行分配担保，将这些对象存放在老年代中。

2.2.2 老年代（Old Generationn）

在新生代中经历了多次（具体看虚拟机配置的阀值）*GC* 后仍然存活下来的对象会进入老年代中。老年代中的对象生命周期较长，存活率比较高，在老年代中进行 *GC* 的频率相对而言较低，而且回收的速度也比较慢。

2.2.3 永久代（Permanent Generationn）

永久代存储类信息、常量、静态变量、即时编译器编译后的代码等数据，对这一区域而言，Java 虚拟机规范指出可以不进行垃圾收集，一般而言不会进行垃圾回收。

3 垃圾回收算法及分代垃圾收集器

3.1 垃圾收集器的分类

3.1.1 次收集器

Scavenge GC，指发生在新生代的 *GC*，因为新生代的 *Java* 对象大多都是朝生夕死，所以 *Scavenge GC* 非常频繁，一般回收速度也比较快。当 *Eden* 空间不足以为对象分配内存时，会触发 *Scavenge GC*。

一般情况下，当新对象生成，并且在 *Eden* 申请空间失败时，就会触发 *Scavenge GC*，对 *Eden* 区域进行 *GC*，清除非存活对象，并且把尚且存活的对象移动到 *Survivor* 区。然后整理 *Survivor* 的两个区。这种方式的 *GC* 是对年轻代的 *Eden* 区进行，不会影响到年老代。因为大部分对象都是从 *Eden* 区开始的，同时 *Eden* 区不会分配的很大，所以 *Eden* 区的 *GC* 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 *Eden* 去能尽快空闲出来。

当年轻代堆空间紧张时会被触发

相对于全收集而言，收集间隔较短

3.1.2 全收集器

Full GC，指发生在老年代的 *GC*，出现了 *Full GC* 一般会伴随着至少一次的 *Minor GC*（老年代的对象大部分是 *Minor GC* 过程中从新生代进入老年代），比如：分配担保失败。*Full GC* 的速度一般会比 *Minor GC* 慢 *10* 倍以上。当老年代内存不足或者显式调用 *System.gc()*方法时，会触发 *Full GC*。

当老年代或者持久代堆空间满了，会触发全收集操作

可以使用 *System.gc()*方法来显式的启动全收集

全收集一般根据堆大小的不同，需要的时间不尽相同，但一般会比较长。

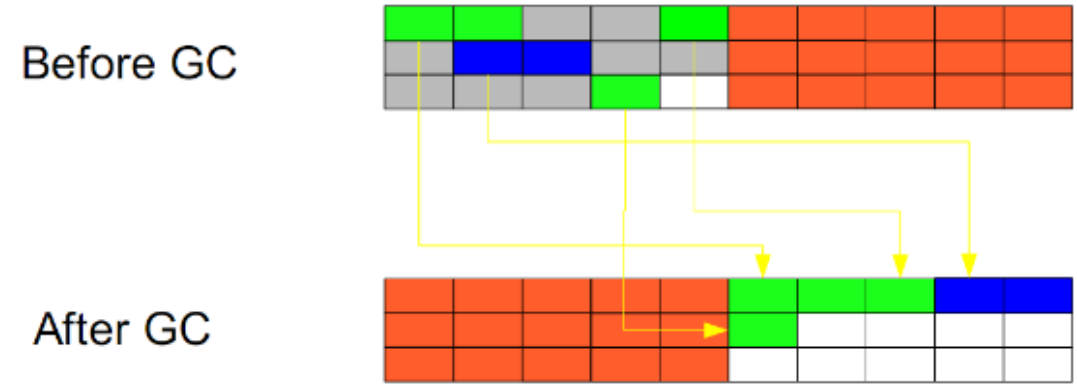
3.2 常见垃圾回收算法

3.2.1 引用计数（Reference Counting）

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 *0* 的对象。此算法最致命的是无法处理循环引用的问题。

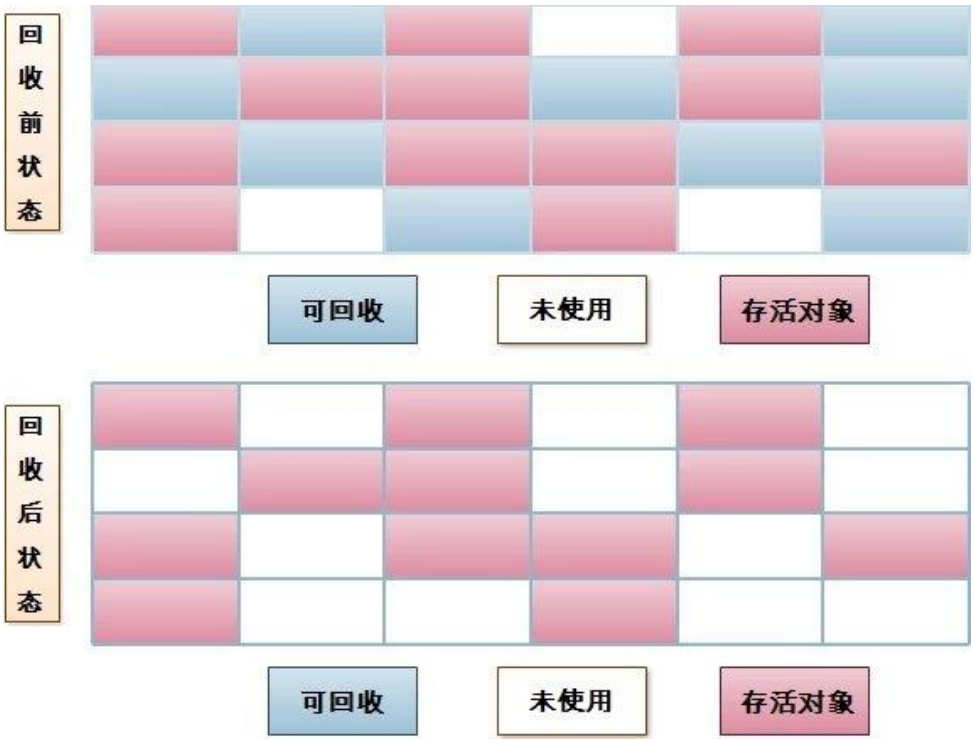
3.2.2 复制（Copying）

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。简图如下：



3.2.3 标记-清除（Mark-Sweep）

此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。简图如下：



3.2.4 标记-整理（Mark-Compact）

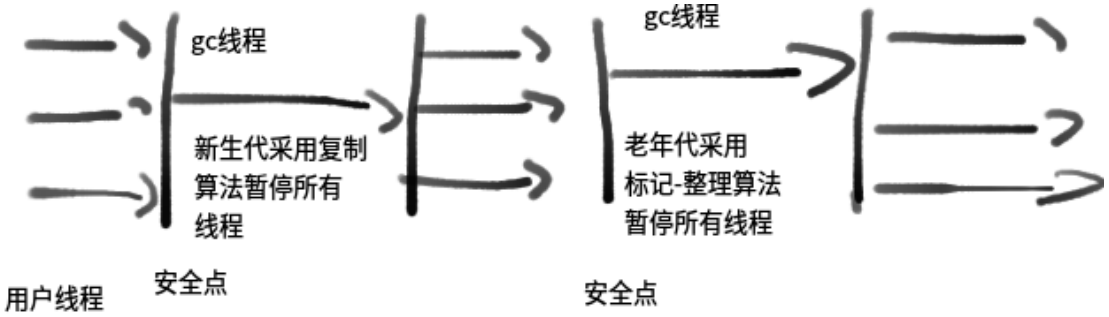
此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。简图如下：

3.3 分代垃圾收集器

3.3.1 串行收集器（Serial）

Serial 收集器是 *Hotspot* 运行在 *Client* 模式下的默认新生代收集器，它的特点是：只用一个 *CPU*（计算核心）/一条收集线程去完成 *GC* 工作，且在进行垃圾收集时必须暂停其他所有的工作线程(“*Stop The World*” -后面简称 *STW*)。可以使用*-XX:+UseSerialGC* 打开。

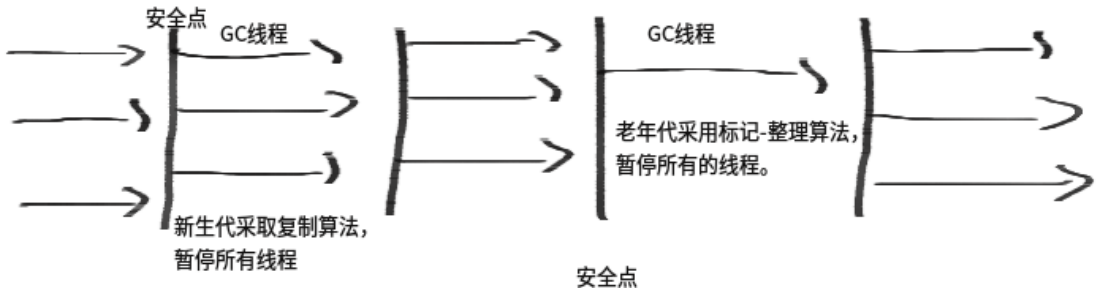
虽然是单线程收集，但它却简单而高效，在 *VM* 管理内存不大的情况下(收集几十 *M*~一两百 *M* 的新生代)，停顿时间完全可以控制在几十毫秒~一百多毫秒内。



3.3.2 并行收集器（ParNew）

ParNew 收集器其实是前面 *Serial* 的多线程版本，除使用多条线程进行 *GC* 外，包括 *Serial* 可用的所有控制参数、收集算法、*STW*、对象分配规则、回收策略等都与 *Serial* 完全一样(也是 *VM* 启用 *CMS* 收集器*-XX: +UseConcMarkSweepGC* 的默认新生代收集器)。

由于存在线程切换的开销，*ParNew* 在单 *CPU* 的环境中比不上 *Serial*，且在通过超线程技术实现的两个 *CPU* 的环境中也不能 *100%*保证能超越 *Serial*。但随着可用的 *CPU* 数量的增加，收集效率肯定也会大大增加*(ParNew* 收集线程数与 *CPU* 的数量相同，因此在 *CPU* 数量过大的环境中，可用*-XX:ParallelGCThreads=<N>*参数控制 *GC* 线程数*)*。



3.3.3 Parallel Scavenge 收集器

与 *ParNew* 类似，*Parallel Scavenge* 也是使用复制算法，也是并行多线程收集器。但与其他收集器关注尽可能缩短垃圾收集时间不同，*Parallel Scavenge* 更关注系统吞吐量*:*

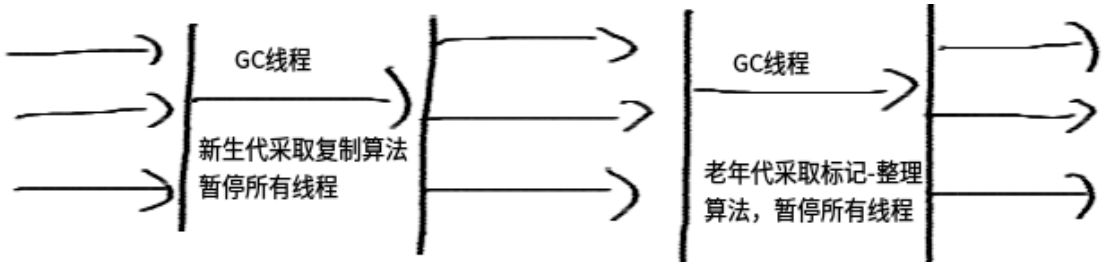
系统吞吐量=运行用户代码时间*/(*运行用户代码时间+垃圾收集时间*)*

停顿时间越短就越适用于用户交互的程序-良好的响应速度能提升用户的体验*;*而高吞吐量则适用于后台运算而不需要太多交互的任务-可以最高效率地利用 *CPU* 时间*,*尽快地完成程序的运算任务*.* *Parallel Scavenge* 提供了如下参数设置系统吞吐量*:*

Parallel Scavenge 参数	描述
-XX:MaxGCPauseMillis	*(*毫秒数*)* 收集器将尽力保证内存回收花费的时间不超过设定值，但如果太小将会导致 *GC* 的频率增加*.*
-XX:GCTimeRatio	*(*整数*:0 < GCTimeRatio < 100)* 是垃圾收集时间占总时间的比率
XX:+UseAdaptiveSizePolicy	启用 *GC* 自适应的调节策略*:* 不再需要手工指定*-Xmn*、*-XX:SurvivorRatio*、*-XX:PretenureSizeThreshold* 等细节参数，*VM* 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最大的吞吐量

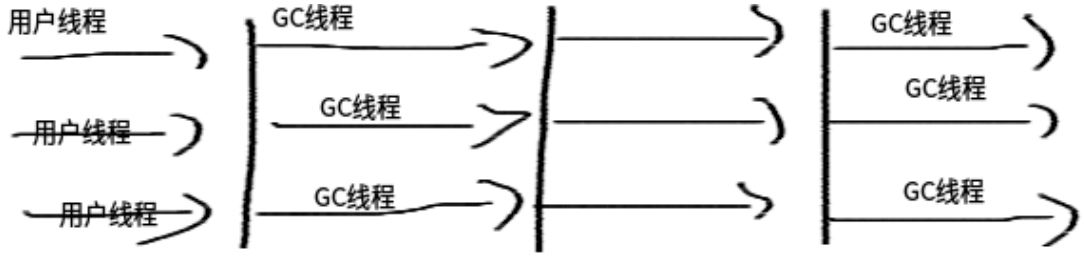
3.3.4 Serial Old 收集器

Serial Old 是 *Serial* 收集器的老年代版本，同样是单线程收集器*,*使用“标记-整理”算法



3.3.5 Parallel Old 收集器

Parallel Old 是 *Parallel Scavenge* 收集器的老年代版本，使用多线程和“标记－整理”算法，吞吐量优先，主要与 *Parallel Scavenge* 配合在注重吞吐量及 *CPU* 资源敏感系统内使用；



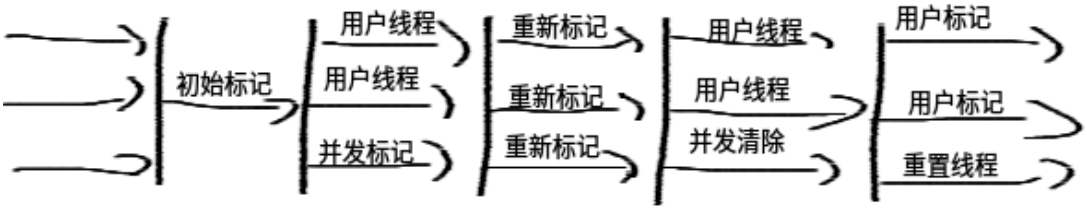
3.3.6 CMS 收集器（Concurrent Mark Sweep）

*CMS(Concurrent Mark Sweep)*收集器是一款具有划时代意义的收集器，一款真正意义上的并发收集器，虽然现在已经有了理论意义上表现更好的 *G1* 收集器，但现在主流互联网企业线上选用的仍是 *CMS(*如 *Taobao*、微店*).*

CMS 是一种以获取最短回收停顿时间为目标的收集器*(CMS* 又称多并发低暂停的收集器*)*，基于”标记-清除”算法实现，整个 *GC* 过程分为以下 *4* 个步骤*:*

1. 初始标记*(CMS initial mark)*
2. 并发标记*(CMS concurrent mark: GC Roots Tracing* 过程*)*
3. 重新标记*(CMS remark)*
4. 并发清除*(CMS concurrent sweep:* 已死对象将会就地释放，注意*:*此处没有压缩*)*

其中 *1*，*3* 两个步骤*(*初始标记、重新标记*)*仍需 *STW.* 但初始标记仅只标记一下 *GC Roots* 能直接关联到的对象，速度很快*;* 而重新标记则是为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，虽然一般比初始标记阶段稍长，但要远小于并发标记时间*.*



CMS 特点：

1. CMS 默认启动的回收线程数*=(CPU* 数目*+3)4*

当 *CPU* 数*>4* 时，*GC* 线程一般占用不超过 *25%*的 *CPU* 资源，但是当 *CPU* 数*<=4* 时，*GC* 线程可能就会过多的占用用户 *CPU* 资源，从而导致应用程序变慢，总吞吐量降低*.*

*2.*无法处理浮动垃圾，可能出现 *Promotion Failure*、*Concurrent Mode Failure* 而导致另一次 *Full GC* 的产生*:* 浮动垃圾是指在 *CMS* 并发清理阶段用户线程运行而产生的新垃圾*.* 由于在 *GC* 阶段用户线程还需运行，因此还需要预留足够的内存空间给用户线程使用，导致 *CMS* 不能像其他收集器那样等到老年代几乎填满了再进行收集*.* 因此 *CMS* 提供了 *-XX:CMSInitiatingOccupancyFraction* 参数来设置 *GC* 的触发百分比*(*以及

-XX:+UseCMSInitiatingOccupancyOnly 来启用该触发百分比*)*，当老年代的使用空间超过该比例后 *CMS* 就会被触发*(JDK 1.6* 之后默认 *92%)*. 但当 *CMS* 运行期间预留的内存无法满足程序需要，就会出现上述 *Promotion Failure* 等失败，这时 *VM* 将启动后备预案*:* 临时启用 *Serial Old* 收集器来重新执行 *Full GC(CMS* 通常配合大内存使用，一旦大内存转入串行的 *Serial GC*，那停顿的时间就是大家都不愿看到的了*)*.

*3.*最后，由于 *CMS* 采用”标记-清除”算法实现，可能会产生大量内存碎片. 内存碎片过多可能会导致无法分配大对象而提前触发 *Full GC*. 因此 *CMS* 提供了 *-XX:+UseCMSCompactAtFullCollection* 开关参数，用于在 *Full GC* 后再执行一个碎片整理过程. 但内存整理是无法并发的，内存碎片问题虽然没有了，但停顿时间也因此变长了，因此 *CMS* 还提供了另外一个参数*-XX:CMSFullGCsBeforeCompaction* 用于设置在执行 *N* 次不进行内存整理的 *Full GC* 后，跟着来一次带整理的*(*默认为 *0:* 每次进入 *Full GC* 时都进行碎片整理*)*.

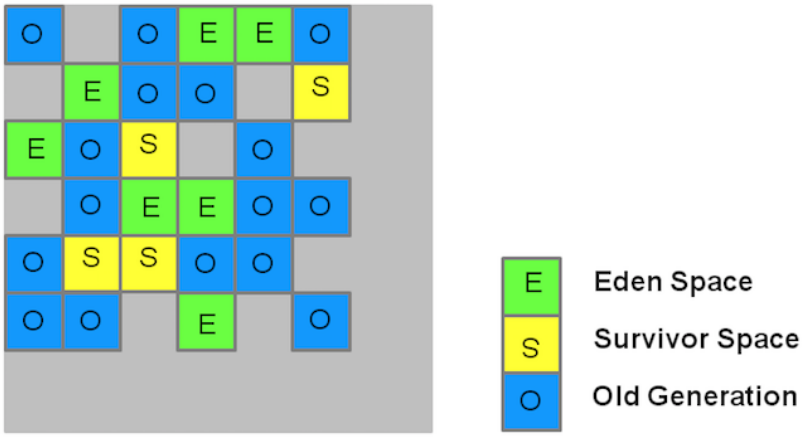
3.3.7 分区收集- G1 收集器

*G1(Garbage-First)*是一款面向服务端应用的收集器，主要目标用于配备多颗 *CPU* 的服务器治理大内存.

- G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS).

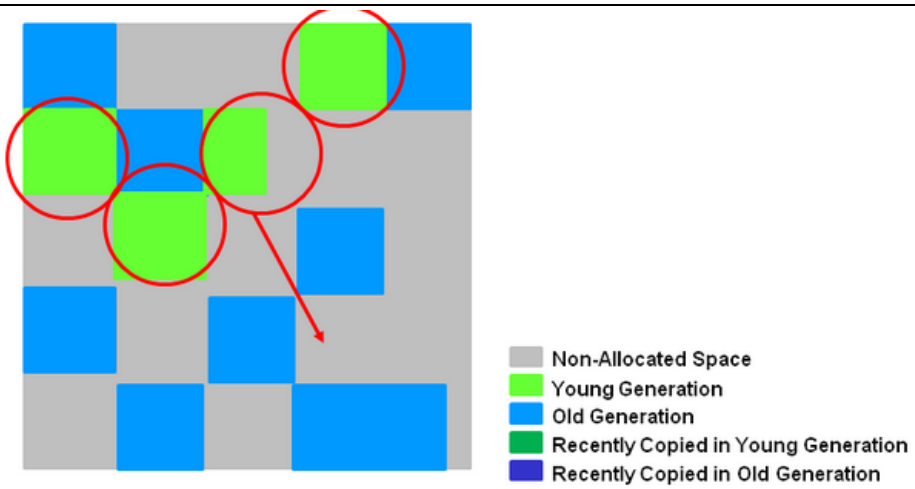
-XX:+UseG1GC 启用 *G1* 收集器.

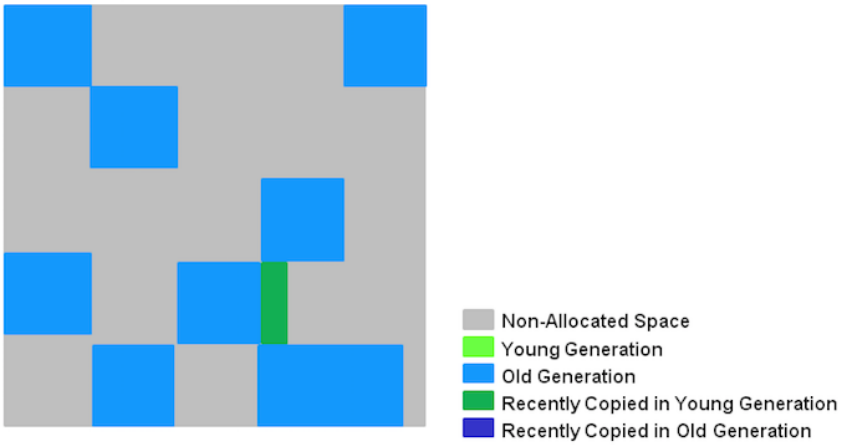
与其他基于分代的收集器不同，*G1* 将整个 *Java* 堆划分为多个大小相等的独立区域*(Region)*，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 *Region(*不需要连续*)*的集合.如：



每块区域既有可能属于 *O* 区、也有可能是 *Y* 区，因此不需要一次就对整个老年代/新生代回收. 而是当线程并发寻找可回收的对象时，有些区块包含可回收的对象要比其他区块多很多. 虽然在清理这些区块时 *G1* 仍然需要暂停应用线程，但可以用相对较少的时间优先回收垃圾较多的 *Region*. 这种方式保证了 *G1* 可以在有限的时间内获取尽可能高的收集效率.

G1 的新生代收集跟 *ParNew* 类似*:* 存活的对象被转移到一个/多个 *Survivor Regions*. 如果存活时间达到阀值，这部分对象就会被提升到老年代.如图：





其特定是：
一整块堆内存被分为多个 *Regions.*
存活对象被拷贝到新的 *Survivor* 区或老年代.
年轻代内存由一组不连续的 *heap* 区组成，这种方法使得可以动态调整各代区域尺寸.
Young GC 会有 *STW* 事件，进行时所有应用程序线程都会被暂停.
多线程并发 *GC.*

G1 老年代 GC 特点如下:
并发标记阶段
1 在与应用程序并发执行的过程中会计算活跃度信息.
2 这些活跃度信息标识出那些 regions 最适合在 STW 期间回收(which regions will be best to reclaim during an evacuation pause).
3 不像 CMS 有清理阶段.
再次标记阶段
1 使用 Snapshot-at-the-Beginning(SATB)算法比 CMS 快得多.
2 空 region 直接被回收.
拷贝/清理阶段(Copying/Cleanup Phase)
1 年轻代与老年代同时回收.
2 老年代内存回收会基于他的活跃度信息.

4 JVM 优化

4.1 JDK 常用 JVM 优化相关命令

bin	描述	功能
jps	打印 Hotspot VM 进程	VMID、JVM 参数、main()函数参数、主类名/Jar 路径
jstat	查看 Hotspot VM 运行时信息	类加载、内存、GC[可分代查看]、JIT 编译 命令格式：jstat -gc 10340 250 20
jinfo	查看和修改虚拟机各项配置	-flag name=value
jmap	heapdump: 生成 VM 堆转储快照、查询 finalize 执行队列、Java 堆和永久代详细信息	jmap -dump:live,format=b,file=heap.bin [VMID]
jstack	查看 VM 当前时刻的线程快照: 当前 VM 内每一条线程正在执行的方法堆栈集合	Thread.getAllStackTraces() 提供了类似的功能
javap	查看经 javac 之后产生的 JVM 字节码代码	自动解析.class 文件, 避免了去理解 class 文件格式以及手动解析 class 文件内容
jcmd	一个多功能工具, 可以用来导出堆, 查看 Java 进程、导出线程信息、 执行 GC、查看性能相关数据等	几乎集合了 jps、jstat、jinfo、jmap、jstack 所有功能
jconsole	基于 JMX 的可视化监视、管理工具	可以查看内存、线程、类、CPU 信息, 以及对 JMX MBean 进行管理
jvisualvm	JDK 中最强大运行监视和故障处理工具	可以监控内存泄露、跟踪垃圾回收、执行时内存分析、CPU 分析、线程分析…

4.1.1 jps

jps - l

显示线程 *id* 和执行线程的主类名

jps -v

显示线程 *id* 和执行线程的主类名和 *JVM* 配置信息

```
C:\Users\Administrator>jps
4488 TestContainer02
2400
5788 Jps

C:\Users\Administrator>jps -l
5856 sun.tools.jps.Jps
4488 concurrent.t04.TestContainer02
2400

C:\Users\Administrator>jps -v
4488 TestContainer02 -Dfile.encoding=UTF-8
2400  -Dosgi.requiredJavaVersion=1.7 -Xms256m -Xmx1024m -Dhttps.protocols=TLSv1.1,TLSv1.2 -XX:MaxPermSize=256m
5920 Jps -Denv.class.path=.;D:\jdk1.7.0.80\java7\lib\rt.jar;D:\jdk1.7.0.80\java7\lib\tools.jar -Dapplication.home=D:\jdk
1.7.0.80\java7 -Xms8m
```

4.1.2 jstat

jstat -参数 线程 *id* 执行时间（单位毫秒） 执行次数

jstat -gc 4488 30 10

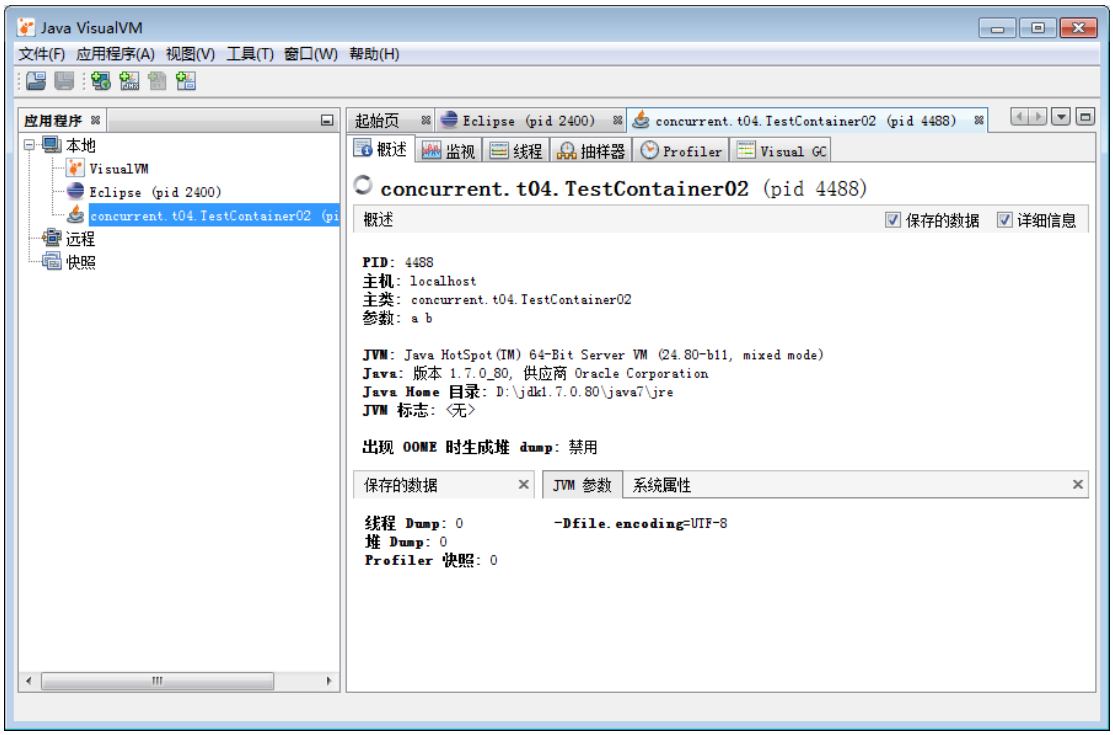
S0C	*S1C*	*S0U*	*S1U*	*EC*	*EU*
10752.0	*10752.0*	*0.0*	*0.0*	*65536.0*	*18351.3*

OC	*OU*	*PC*	*PU*	*YGC*	*YGCT*	*FGC*	*FGCT*	*GCT*
173568.0	*0.0*	*21504.0*	*2659.6*	*0*	*0.000*	*0*	*0.000*	*0.000*

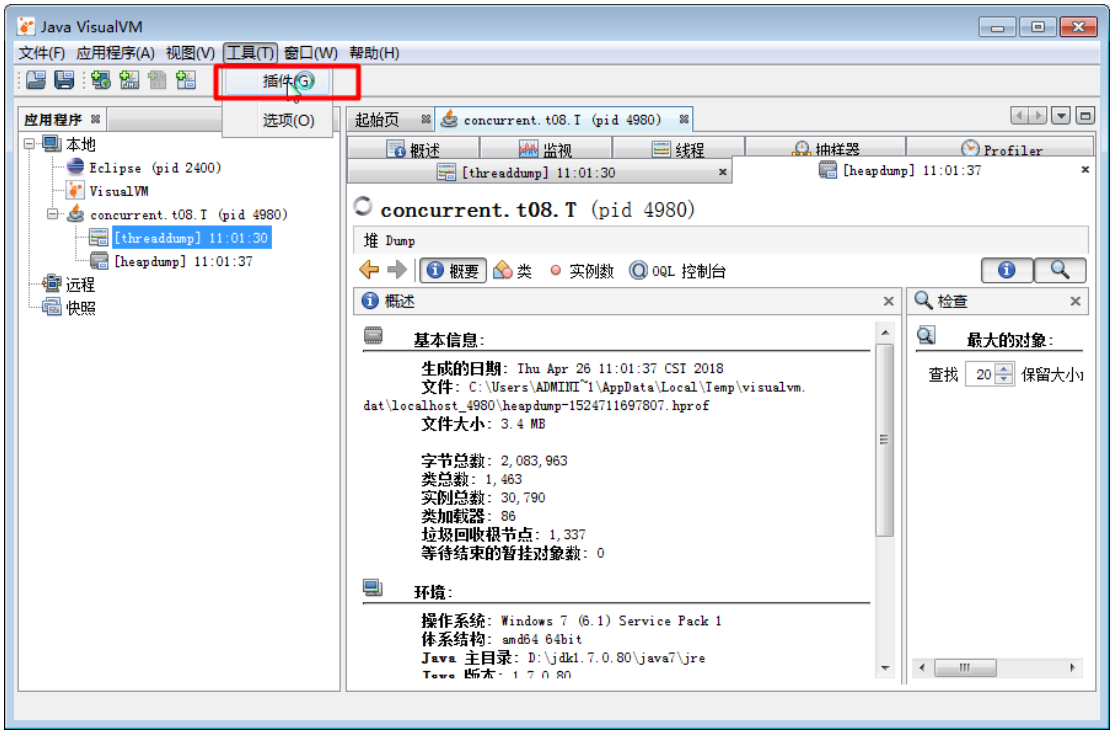
SXC - survivor 初始空间大小，单位字节。
SXU - survivor 使用空间大小， 单位字节。
EC - eden 初始空间大小
EU - eden 使用空间大小
OC - old 初始空间大小
OU - old 使用空间大小
PC - permanent 初始空间大小
PU - permanent 使用空间大小
YGC - youngGC 收集次数
YGCT - youngGC 收集使用时长， 单位秒
FGC - fullGC 收集次数
FGCT - fullGC 收集使用时长
GCT - 总计收集使用总时长

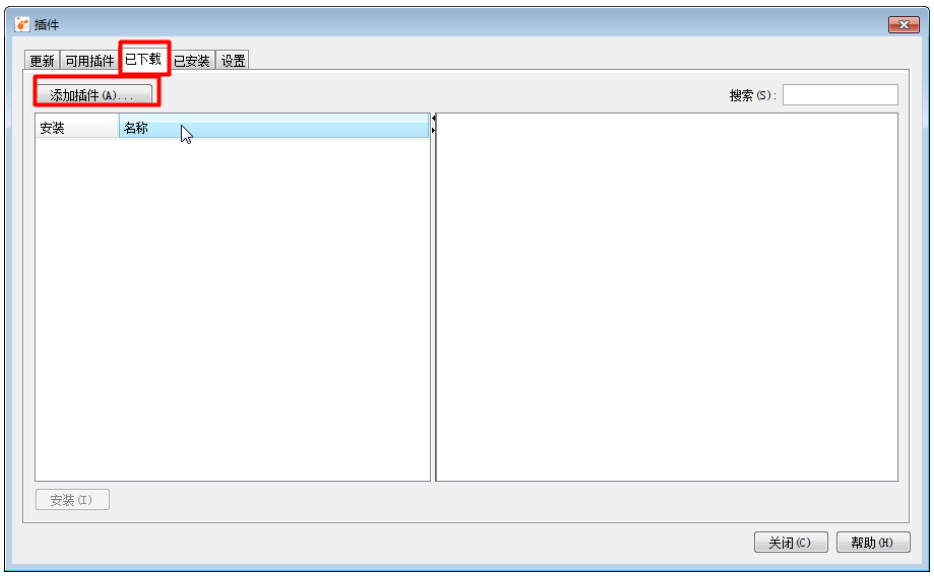
4.1.3 jvisualvm

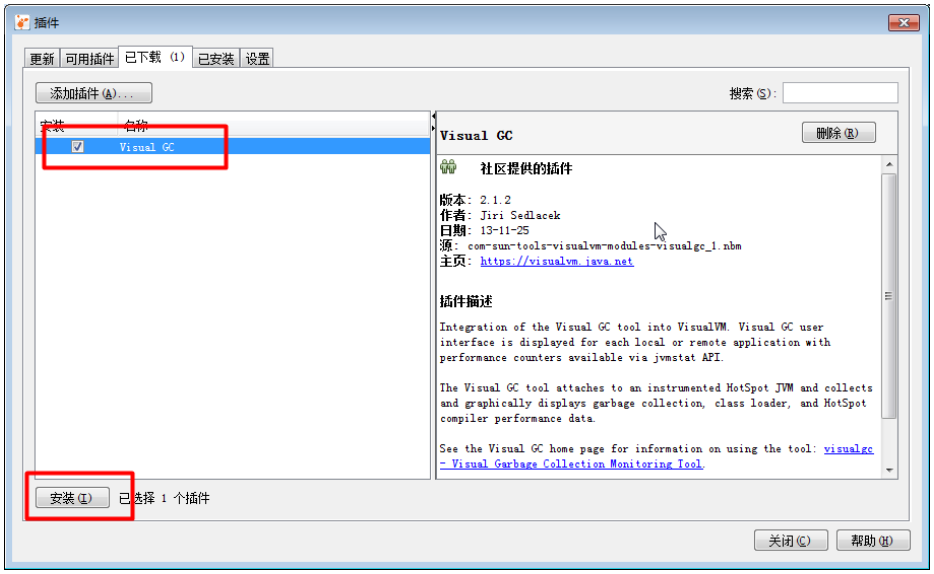
一个 *JDK* 内置的图形化 *VM* 监视管理工具

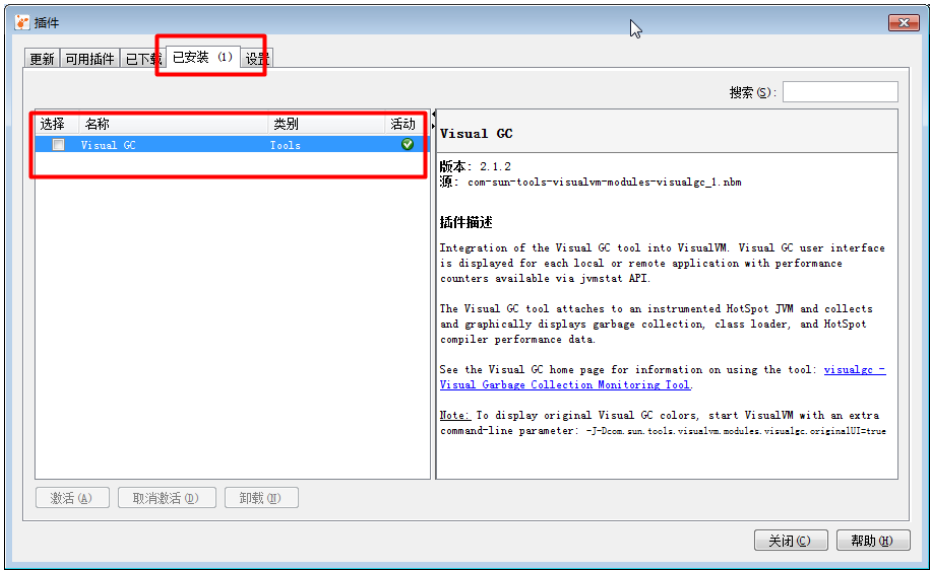


4.1.4 visualgc 插件

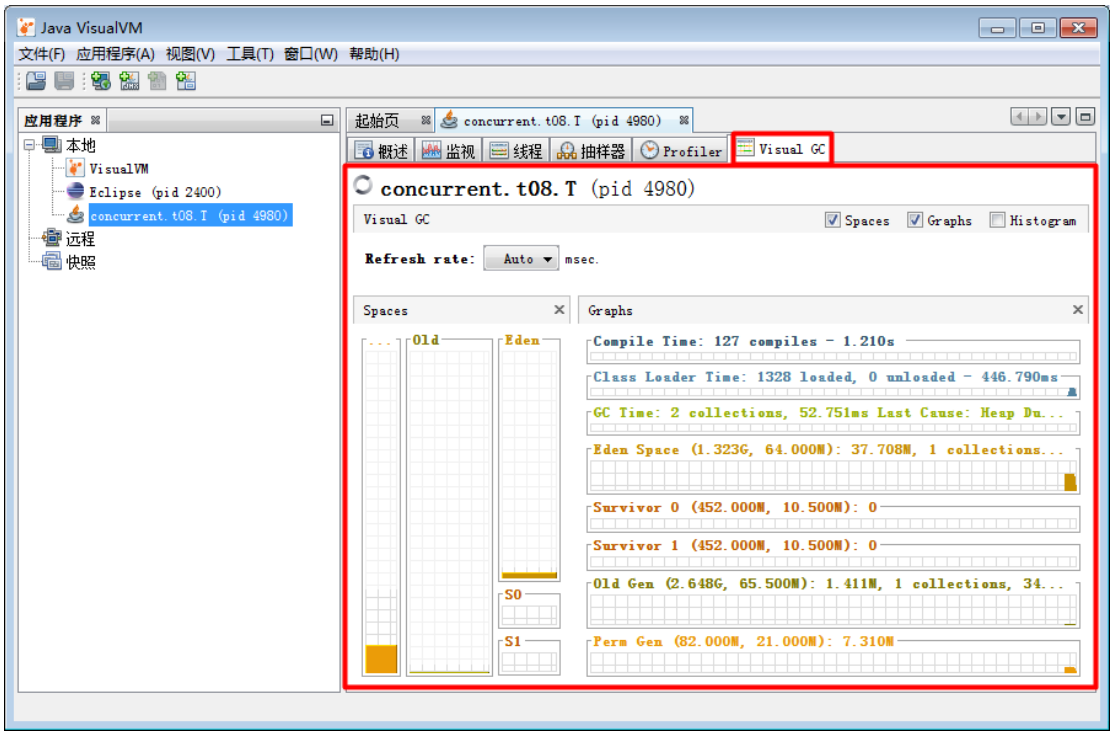








重启 *jvisualvm* 工具



4.2 JVM 常见参数

4.2.1 内存设置

*-Xms:*初始堆大小

*-Xmx:*最大堆大小

-Xmn：设置年轻代大小为 *2G*。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为 *64m*，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，*Sun* 官方推荐配置为整个堆的 *3/8*。

-Xss： 设置每个线程的堆栈大小。*JDK5.0* 以后每个线程堆栈大小为 *1M*，以前每个线程堆栈大小为 *256K*。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 *3000~5000* 左右。

*-XX:NewSize=n:*设置年轻代大小

*-XX:NewRatio=n:*设置年轻代和年老代的比值。如:为 *3*，表示年轻代与年老代比值为 *3*：*1*，年轻代占整个年轻代年老代和的 *3/4*

*-XX:SurvivorRatio=n:*年轻代中 *Eden* 区与两个 *Survivor* 区的比值。注意 *Survivor* 区有两个。如：*3*，表示 *Eden*：*Survivor=3*：*2*，一个 *Survivor* 区占整个年轻代的 *1/5*

*-XX:MaxPermSize=n:*设置持久代大小

-XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为 *0* 的话，则年轻代对象不经过 *Survivor* 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 *Survivor* 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概率。

4.2.2 内存设置经验分享

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（*32-bt* 还是 *64-bit*）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。*32* 位系统 下，一般限制在 *1.5G~2G*；*64* 为操作系统对内存无限制。

常见设置：

-Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

4.2.3 收集器设置

*-XX:+UseSerialGC:*设置串行收集器

*-XX:+UseParallelGC:*设置并行收集器

*-XX:+UseParNewGC:*设置年轻代为并行收集。可与 *CMS* 收集同时使用。*JDK5.0* 以上，*JVM* 会根据系统配置自行设置，所以无需再设置此值。

*-XX:+UseParalledlOldGC:*设置并行年老代收集器，*JDK6.0* 支持对年老代并行收集。

*-XX:+UseConcMarkSweepGC:*设置年老代并发收集器，测试中配置这个以后，*-XX:NewRatio* 的配置失效，原因不明。所以，此时年轻代大小最好用*-Xmn* 设置。

*-XX:+UseG1GC:*设置 *G1* 收集器

4.2.4 垃圾回收统计信息

-XX:+PrintGC

-XX:+Printetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

4.2.5 并行收集器设置

*-XX:ParallelGCThreads=n:*设置并行收集器收集时使用的 *CPU* 数。并行收集线程数。

*-XX:MaxGCPauseMillis=n:*设置并行收集最大暂停时间，单位毫秒

*-XX:GCTimeRatio=n:*设置垃圾回收时间占程序运行时间的百分比。公式为 *1/(1+n)*并发收集器设置

*-XX:+CMSIncrementalMode:*设置为增量模式。适用于单 *CPU* 情况。

*-XX:ParallelGCThreads=n:*设置并发收集器年轻代收集方式为并行收集时，使用的 *CPU* 数。并行收集线程数

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的 *Survivor* 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 *GC* 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

4.2.6 收集器设置经验分享

关于收集器的选择 *JVM* 给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，*JDK5.0* 以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。*JDK5.0* 以后，*JVM* 会根据当前系统配置进行判断。

常见配置：

并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

-Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

4.2.7 简单总结

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达 *Gbit* 的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合 *8CPU* 以上的应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

并发垃圾收集信息

持久代并发收集次数

传统 *GC* 信息

花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代存放长期存活对象。

较小堆引起的碎片问题，因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、整理方式进行回收。如果出现“碎片”，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

-XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次 *Full GC* 后，对年老代进行压缩