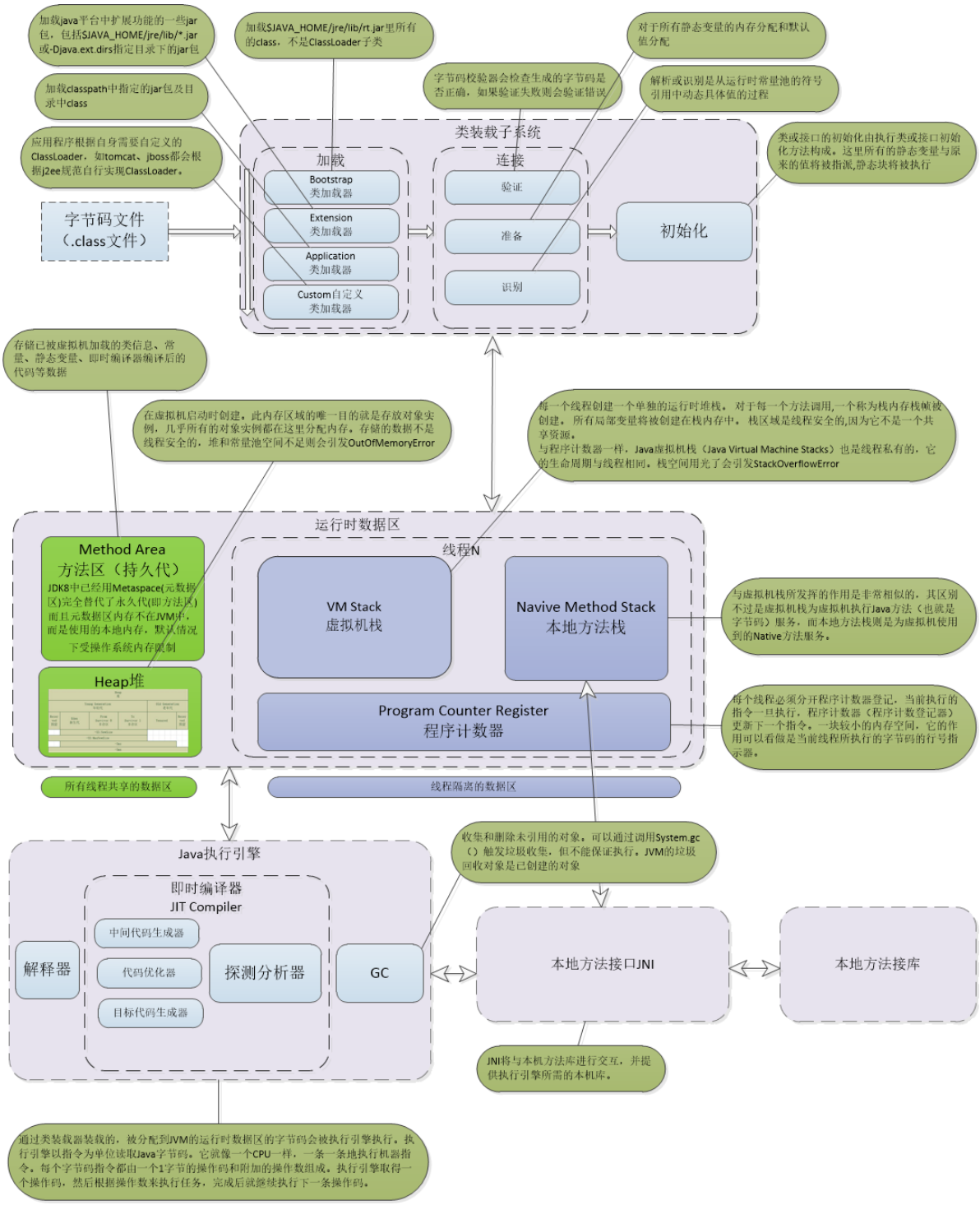
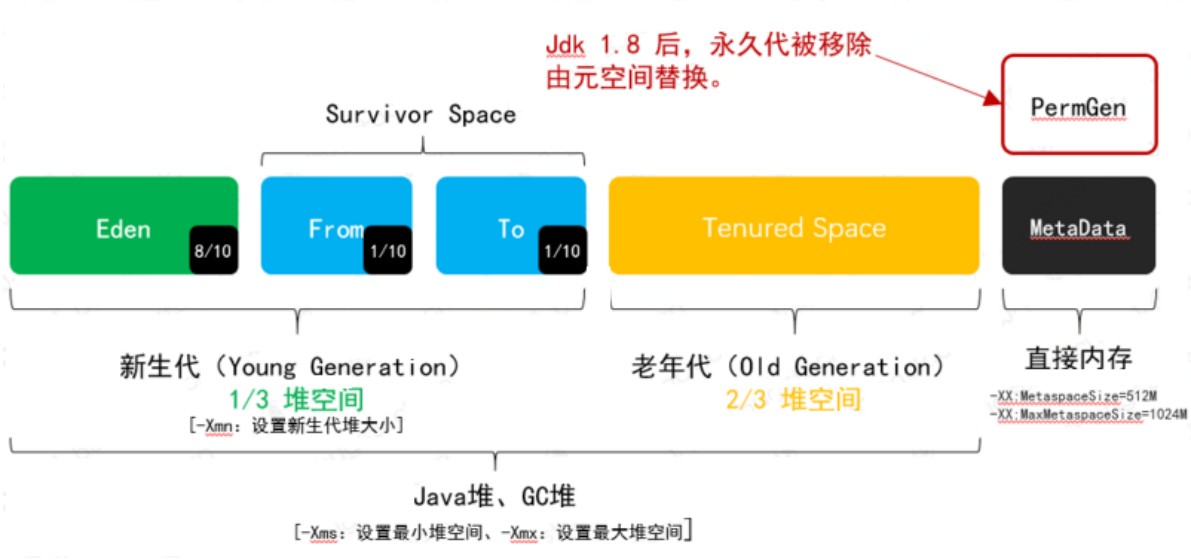
JVM（马老师，黄老师）

1. 描述一下jvm内存模型，以及这些空间的存放的内容？



2. 堆内存划分的空间，如何回收这些内存对象，有哪些回收算法？



垃圾回收算法：标记清除、复制（多为新生代垃圾回收使用）、标记整理

3. 如何解决线上gc频繁的问题?

1. 查看监控，以了解出现问题的时间点以及当前FGC的频率（可对比正常情况看频率是否正常）
2. 了解该时间点之前有没有程序上线、基础组件升级等情况。
3. 了解JVM的参数设置，包括：堆空间各个区域的大小设置，新生代和老年代分别采用了哪些垃圾收集器，然后分析JVM参数设置是否合理。
4. 再对步骤1中列出的可能原因做排除法，其中元空间被打满、内存泄漏、代码显式调用gc方法比较容易排查。
5. 针对大对象或者长生命周期对象导致的FGC，可通过 jmap -histo 命令并结合dump堆内存文件作进一步分析，需要先定位到可疑对象。
6. 通过可疑对象定位到具体代码再次分析，这时候要结合GC原理和JVM参数设置，弄清楚可疑对象是否满足了进入到老年代的条件才能下结论。

4. 描述一下class初始化过程?

一个类初始化就是执行clinit()方法，过程如下

- 父类初始化
- static变量初始化/static块（按照文本顺序执行）

*Java Language Specification*中，类初始化详细过程如下（最重要的是类初始化是线程安全的）：

1. 每个类都有一个初始化锁LC，进程获取LC（如果没有获取到，就一直等待）
2. 如果C正在被其他线程初始化，释放LC并等待C初始化完成
3. 如果C正在被本线程初始化，即**递归初始化**，释放LC
4. 如果C已经被初始化了，释放LC
5. 如果C处于erroneous状态，释放LC并抛出异常NoClassDefFoundError
6. 否则，将C标记为正在被本线程初始化，释放LC；然后， 初始化那些final且为基础类型的类成员变量
7. 初始化C的父类SC和各个接口SI_n（按照implements子句中的顺序来）；如果SC或SIn初始化过程中抛出异常，则获取LC，将C标记为erroneous，并通知所有线程，然后释放LC，然后再抛出同样的异常。
8. 从classloader处获取assertion是否被打开
9. 接下来， 按照文本顺序执行类变量初始化和静态代码块，或接口的字段初始化，把它们当作是一个个单独的代码块。
10. 如果执行正常，获取LC，标记C为已初始化，并通知所有线程，然后释放LC

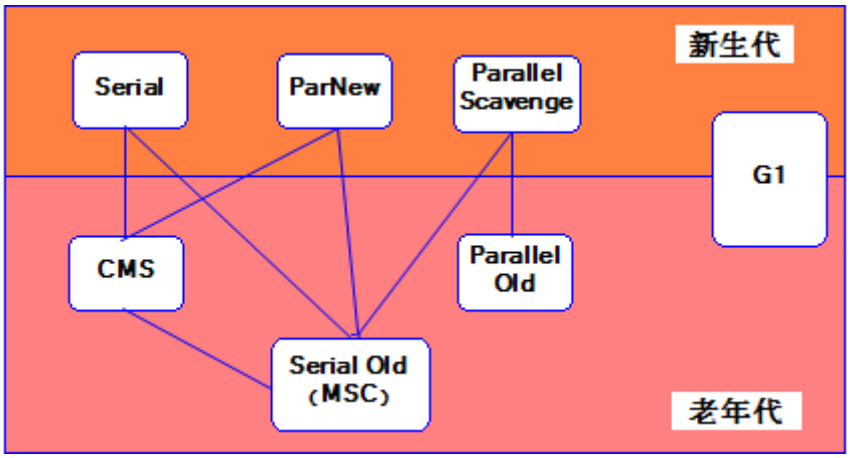
11. 否则，如果抛出了异常E。若E不是Error，则以E为参数创建新的异常ExceptionInInitializerError作为E。如果因为OutOfMemoryError导致无法创建ExceptionInInitializerError，则将OutOfMemoryError作为E。
12. 获取LC，将C标记为erroneous，通知所有等待的线程，释放LC，并抛出异常E。

5. 简述一下内存溢出的原因，如何排查线上问题?

内存溢出的原因

- java.lang.OutOfMemoryError: ……java heap space…… 堆栈溢出，代码问题的可能性极大
- java.lang.OutOfMemoryError: GC over head limit exceeded 系统处于高频的GC状态，而且回收的效果依然不佳的情况，就会开始报这个错误，这种情况一般是产生了很多不可以被释放的对象，有可能是引用使用不当导致，或申请大对象导致，但是java heap space的内存溢出有可能提前不会报这个错误，也就是可能内存就直接不够导致，而不是高频GC.
- java.lang.OutOfMemoryError: PermGen space jdk1.7之前才会出现的问题，原因是系统的代码非常多或引用的第三方包非常多、或代码中使用了大量的常量、或通过intern注入常量、或者通过动态代码加载等方法，导致常量池的膨胀
- java.lang.OutOfMemoryError: Direct buffer memory 直接内存不足，因为jvm垃圾回收不会回收掉直接内存这部分的内存，所以可能原因是直接或间接使用了ByteBuffer中的allocateDirect方法的时候，而没有做clear
- java.lang.StackOverflowError - Xss设置的太小了
- java.lang.OutOfMemoryError: unable to create new native thread 堆外内存不足，无法为线程分配内存区域
- java.lang.OutOfMemoryError: request {} byte for {}out of swap 地址空间不够

6. jvm有哪些垃圾回收器，实际中如何选择?



图中展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，则说明它们可以搭配使用。虚拟机所处的区域则表示它是属于新生代还是老年代收集器。
新生代收集器（全部的都是复制算法）：Serial、ParNew、Parallel Scavenge
老年代收集器：CMS（标记-清理）、Serial Old（标记-整理）、Parallel Old（标记整理）
整堆收集器： G1（一个Region中是标记-清除算法，2个Region之间是复制算法）

同时，先解释几个名词：

1，**并行（Parallel）**：多个垃圾收集线程并行工作，此时用户线程处于等待状态

2，**并发（Concurrent）**：用户线程和垃圾收集线程同时执行

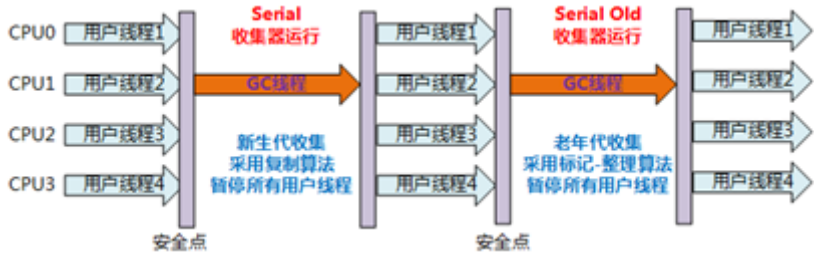
3，**吞吐量**：运行用户代码时间 /（运行用户代码时间 + 垃圾回收时间）

1.Serial收集器是最基本的、发展历史最悠久的收集器。**

特点：单线程、简单高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程手机效率。收集器进行垃圾回收时，必须暂停其他所有的工作线程，直到它结束（Stop The World）。

应用场景：适用于Client模式下的虚拟机。

Serial / Serial Old收集器运行示意图



2.ParNew收集器其实就是Serial收集器的多线程版本。**

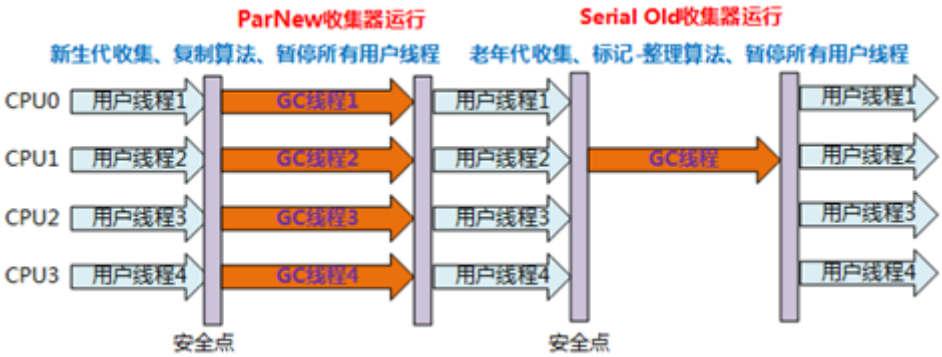
除了使用多线程外其余行为均和Serial收集器一模一样（参数控制、收集算法、Stop The World、对象分配规则、回收策略等）。

特点：多线程、ParNew收集器默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境中，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

和Serial收集器一样存在Stop The World问题

应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为它是除了Serial收集器外，唯一一个能与CMS收集器配合工作的。

ParNew/Serial Old组合收集器运行示意图如下：



3.Parallel Scavenge 收集器与吞吐量关系密切，故也称为吞吐量优先收集器。**

特点：属于新生代收集器也是采用复制算法的收集器，又是并行的多线程收集器（与ParNew收集器类似）。

该收集器的目标是达到一个可控制的吞吐量。还有一个值得关注的点是：GC自适应调节策略（与ParNew收集器最重要的一个区别）

GC自适应调节策略：Parallel Scavenge收集器可设置-XX:+UseAdptiveSizePolicy参数。当开关打开时不需要手动指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThreshold）等，虚拟机会根据系统的运行状况收集性能监控信息，动态设置这些参数以提供最优的停顿时间和最高的吞吐量，这种调节方式称为GC的自适应调节策略。

Parallel Scavenge收集器使用两个参数控制吞吐量：

- XX:MaxGCPauseMillis 控制最大的垃圾收集停顿时间
- XX:GCRatio 直接设置吞吐量的大小。

4.Serial Old是Serial收集器的老年代版本。**

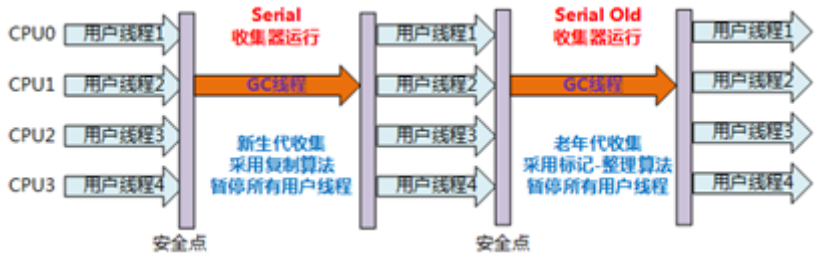
特点：同样是单线程收集器，采用标记-整理算法。

应用场景：主要也是使用在Client模式下的虚拟机中。也可在Server模式下使用。

Server模式下主要的两大用途（在后续中详细讲解…）：

1. 在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用。
2. 作为CMS收集器的后备方案，在并发收集Concurent Mode Failure时使用。

Serial / Serial Old收集器工作过程图（Serial收集器图示相同）：



5.Parallel Old是Parallel Scavenge收集器的老年代版本。**

特点：多线程，采用标记-整理算法。

应用场景：注重高吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge+Parallel Old 收集器。

Parallel Scavenge/Parallel Old收集器工作过程图:

6.CMS收集器是一种以获取最短回收停顿时间为目标的收集器。**

特点：基于标记-清除算法实现。并发收集、低停顿。

应用场景：适用于注重服务的响应速度，希望系统停顿时间最短，给用户带来更好的体验等场景下。如web程序、b/s服务。

CMS收集器的运行过程分为下列4步：

初始标记：标记GC Roots能直接到的对象。速度很快但是仍存在Stop The World问题。

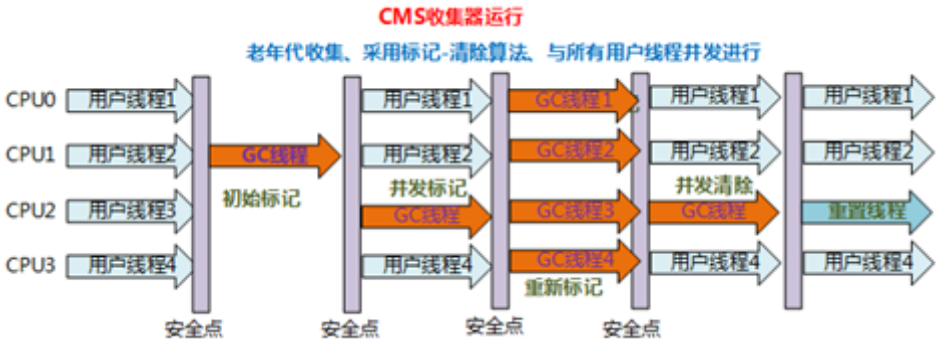
并发标记：进行GC Roots Tracing 的过程，找出存活对象且用户线程可并发执行。

重新标记：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。仍然存在Stop The World问题。

并发清除：对标记的对象进行清除回收。

CMS收集器的内存回收过程是与用户线程一起并发执行的。

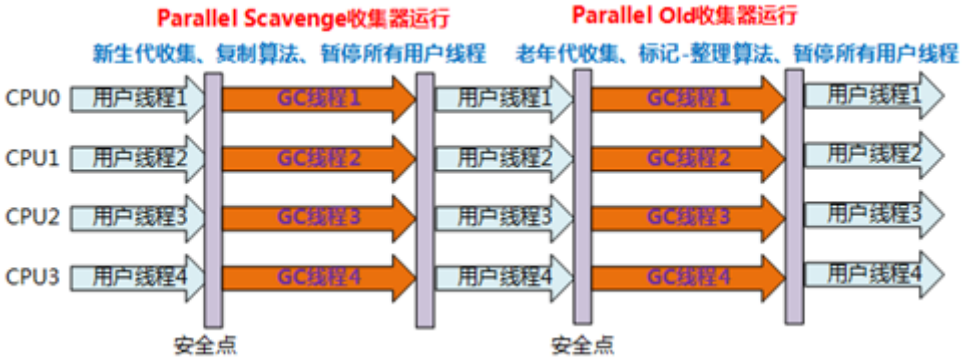
CMS收集器的工作过程图：



CMS收集器的缺点：

- 对CPU资源非常敏感。

- 无法处理浮动垃圾，可能出现Concurrent Model Failure失败而导致另一次Full GC的产生。
- 因为采用标记-清除算法所以会存在空间碎片的问题，导致大对象无法分配空间，不得不提前触发一次Full GC。



**

**

7.G1收集器一款面向服务端应用的垃圾收集器。**

特点如下：

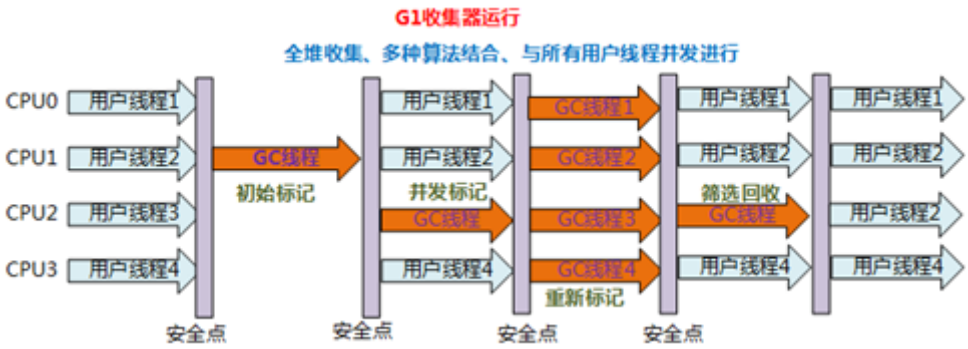
并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让Java程序继续运行。

分代收集：G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整合：G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。

可预测的停顿：G1除了追求低停顿外，还能建立可预测的停顿时间模型。能让使用者明确指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

G1收集器运行示意图：



**

**

关于gc的选择
除非应用程序有非常严格的暂停时间要求，否则请先运行应用程序并允许VM选择收集器（如果没有特别要求。使用VM提供给的默认GC就好）。
如有必要，调整堆大小以提高性能。 如果性能仍然不能满足目标，请使用以下准则作为选择收集器的起点：

- 如果应用程序的数据集较小（最大约100 MB），则选择带有选项-XX：+ UseSerialGC的串行收集器。
- 如果应用程序将在单个处理器上运行，并且没有暂停时间要求，则选择带有选项-XX：+ UseSerialGC的串行收集器。

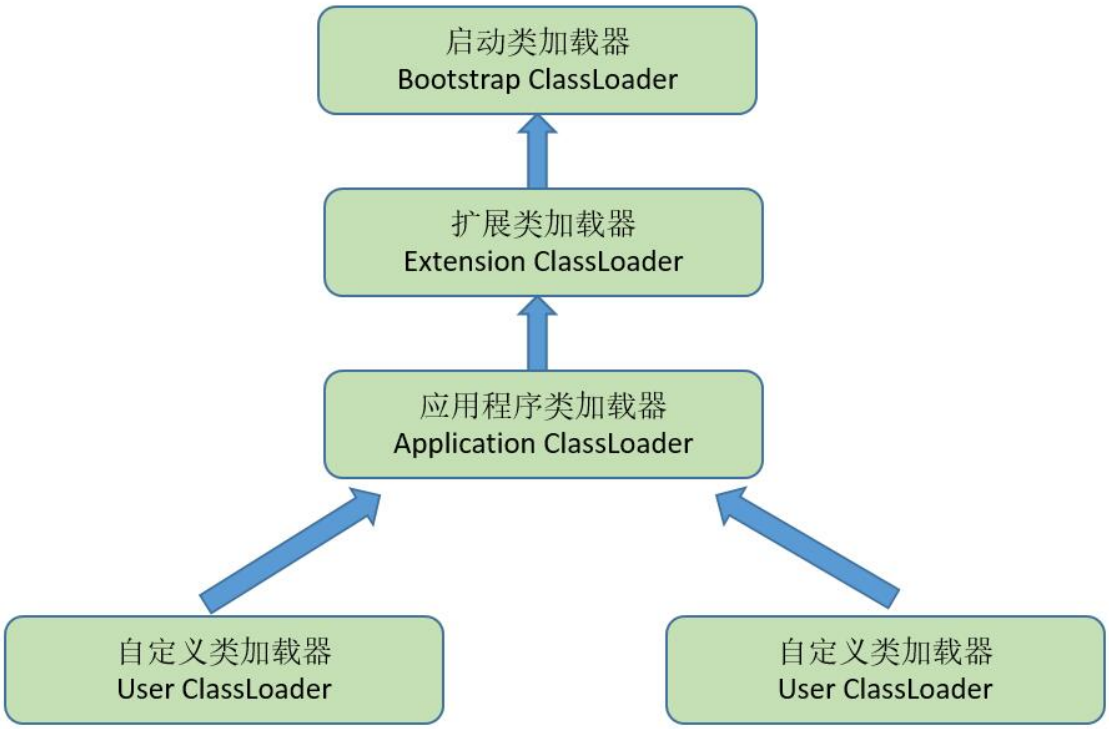
- 如果（a）峰值应用程序性能是第一要务，并且（b）没有暂停时间要求或可接受一秒或更长时间的暂停，则让VM选择收集器或使用-XX：+ UseParallelGC选择并行收集器 。
- 如果响应时间比整体吞吐量更重要，并且垃圾收集暂停时间必须保持在大约一秒钟以内，则选择具有-XX：+ UseG1GC。（值得注意的是JDK9中CMS已经被Deprecated，不可使用！移除该选项）
- 如果使用的是jdk8，并且堆内存达到了16G，那么推荐使用G1收集器，来控制每次垃圾收集的时间。
- 如果响应时间是高优先级，或使用的堆非常大，请使用-XX：UseZGC选择完全并发的收集器。（值得注意的是JDK11开始可以启动ZGC，但是此时ZGC具有实验性质，在JDK15中[202009发布]才取消实验性质的标签，可以直接显示启用，但是JDK15默认GC仍然是G1）

这些准则仅提供选择收集器的起点，因为性能取决于堆的大小，应用程序维护的实时数据量以及可用处理器的数量和速度。

如果推荐的收集器没有达到所需的性能，则首先尝试调整堆和新生代大小以达到所需的目标。 如果性能仍然不足，尝试使用其他收集器

总体原则：减少STOP THE WORD时间，使用并发收集器（比如CMS+ParNew，G1）来减少暂停时间，加快响应时间，并使用并行收集器来增加多处理器硬件上的总体吞吐量。

7. 简述一下Java类加载模型?



双亲委派模型
在某个类加载器加载class文件时，它首先委托父加载器去加载这个类，依次传递到顶层类加载器(Bootstrap)。如果顶层加载不了（它的搜索范围中找不到此类），子加载器才会尝试加载这个类。
双亲委派的好处

- 每一个类都只会被加载一次，避免了重复加载
- 每一个类都会被尽可能的加载（从引导类加载器往下，每个加载器都可能会根据优先次序尝试加载它）

- 有效避免了某些恶意类的加载（比如自定义了Java.lang.Object类，一般而言在双亲委派模型下会加载系统的Object类而不是自定义的Object类）

9. JVM8为什么要增加元空间，带来什么好处?

原因：
1、字符串存在永久代中，容易出现性能问题和内存溢出。
2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。
元空间的特点：
1，每个加载器有专门的存储空间。
2，不会单独回收某个类。
3，元空间里的对象的位置是固定的。
4，如果发现某个加载器不再存货了，会把相关的空间整个回收。

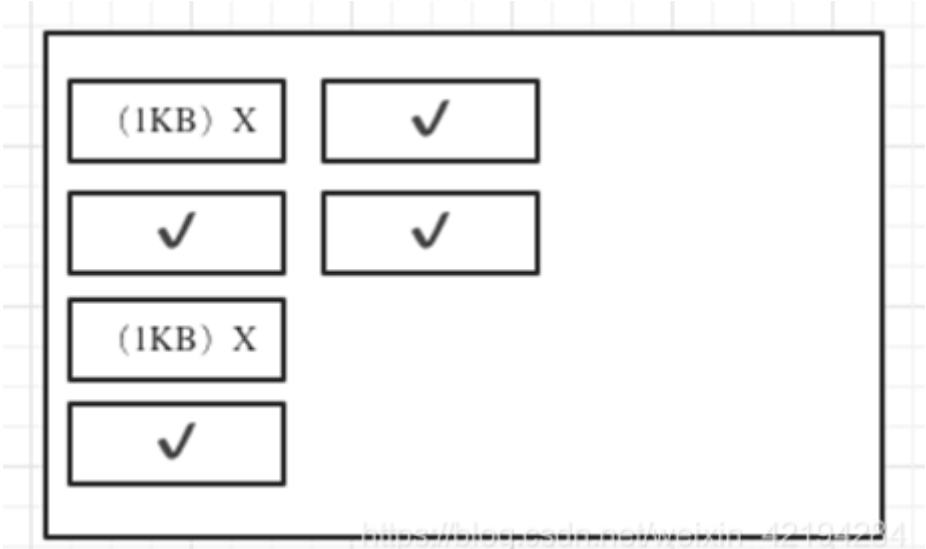
10. 堆G1垃圾收集器有了解么，有什么特点（可以结合第6题内容）？

G1的特点：

1. G1的设计原则是"首先收集尽可能多的垃圾(Garbage First)"。因此，G1并不会等内存耗尽(串行、并行)或者快耗尽(CMS)的时候开始垃圾收集，而是在内部采用了启发式算法，在老年代找出具有高收集收益的分区进行收集。同时G1可以根据用户设置的暂停时间目标自动调整年轻代和总堆大小，暂停目标越短年轻代空间越小、总空间就越大；
2. G1采用内存分区(Region)的思路，将内存划分为一个个相等大小的内存分区，回收时则以分区为单位进行回收，存活的对象复制到另一个空闲分区中。由于都是以相等大小的分区为单位进行操作，因此G1天然就是一种压缩方案(局部压缩)；
3. G1虽然也是分代收集器，但整个内存分区不存在物理上的年轻代与老年代的区别，也不需要完全独立的survivor(to space)堆做复制准备。G1只有逻辑上的分代概念，或者说每个分区都可能随G1的运行在不同代之间前后切换；
4. G1的收集都是STW的，但年轻代和老年代的收集界限比较模糊，采用了混合(mixed)收集的方式。即每次收集既可能只收集年轻代分区(年轻代收集)，也可能在收集年轻代的同时，包含部分老年代分区(混合收集)，这样即使堆内存很大时，也可以限制收集范围，从而降低停顿。
5. 因为G1建立可预测的停顿时间模型，所以每一次的垃圾回收时间都可控，那么对于大堆（16G左右）的垃圾收集会有明显优势

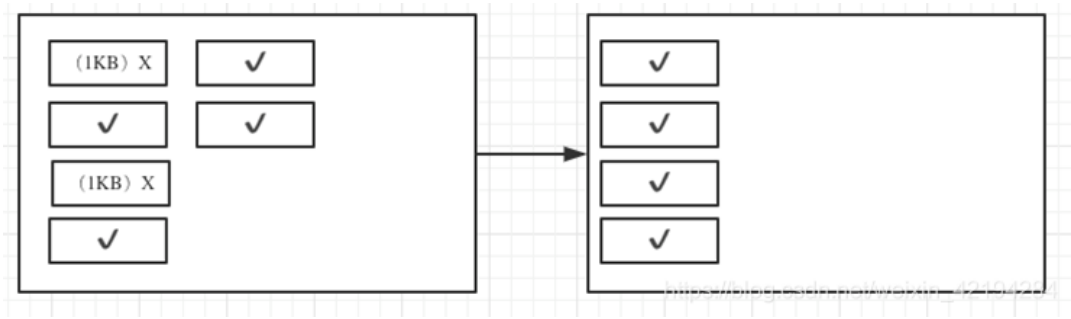
11. 介绍一下垃圾回收算法?

- 标记-清除



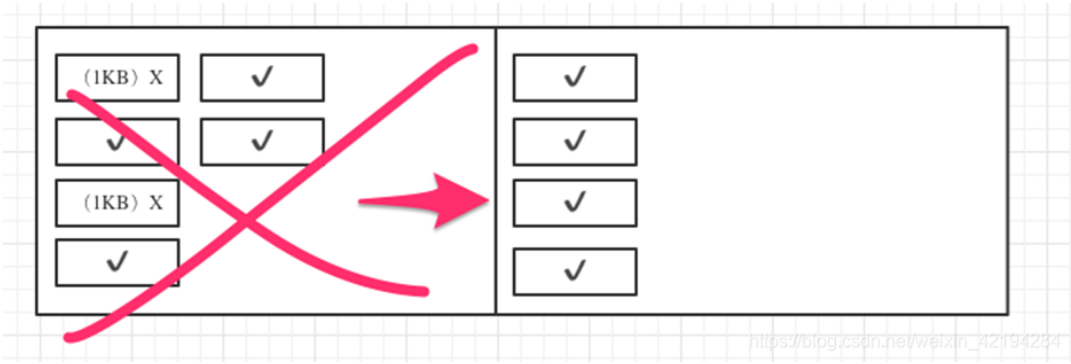
缺点：产生内存碎片，如上图，如果清理了两个1kb的对象，再添加一个2kb的对象，无法放入这两个位置

- 标记-整理（老年代）



缺点：移动对象开销较大

- 复制（新生代）



缺点：浪费空间并且移动对象开销大

12. Happens-Before规则?

先行发生原则（Happens-Before）是判断数据是否存在竞争、线程是否安全的主要依据。
先行发生是Java内存，模型中定义的两项操作之间的偏序关系，如果操作A先行发生于操作B，那么**操作A产生的影响能够被操作B观察到。**
口诀：如果两个操作之间具有happen-before关系，**那么前一个操作的结果就会对后面的一个操作可见。是Java内存模型中定义的两个操作之间的偏序关系。**
常见的happen-before规则：
1.程序顺序规则：
一个线程中的每个操作，happen-before在该线程中的任意后续操作。(注解：如果只有一个线程的操作，那么前一个操作的结果肯定会对后续的操作可见。)
程序顺序规则中所说的每个操作happen-before于该线程中的任意后续操作并不是说前一个操作必须要在后一个操作之前执行，而是指前一个操作的执行结果必须对后一个操作可见，如果不满足这个要求那

就不允许这两个操作进行重排序

2.锁规则：

对一个锁的解锁，happen-before在随后对这个锁加锁。(注解：这个最常见的就是synchronized方法和syncronized块)

3.volatile变量规则：

对一个volatile域的写，happen-before在任意后续对这个volatile域的读。该规则在CurrentHashMap的读操作中不需要加锁有很好的体现。

4.传递性：

如果A happen-before B，且B happen-before C，那么A happen - before C.

5.线程启动规则：

Thread对象的start()方法happen-before此线程的每一个动作。

6.线程终止规则：

线程的所有操作都happen-before对此线程的终止检测，可以通过Thread.join()方法结束，Thread.isAlive()的返回值等手段检测到线程已经终止执行。

7.线程中断规则：

对线程interrupt()方法的调用happen-before发生于被中断线程的代码检测到中断时事件的发生。

13. 描述一下java类加载和初始化的过程?

JAVA类的加载机制

Java类加载分为5个过程,分别为：加载，链接(验证，准备，解析)，初始化，使用，卸载。

加载

加载主要是将.class文件通过二进制字节流读入到JVM中。 在加载阶段，JVM需要完成3件事：

1）通过classloader在classpath中获取XXX.class文件，将其以二进制流的形式读入内存。

2）将字节流所代表的静态存储结构转化为方法区的运行时数据结构；

3）在内存中生成一个该类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

1. 链接
   2.1. 验证
   主要确保加载进来的字节流符合JVM规范。验证阶段会完成以下4个阶段的检验动作：
   1）文件格式验证
   2）元数据验证(是否符合Java语言规范)
   3）字节码验证（确定程序语义合法，符合逻辑）
   4）符号引用验证（确保下一步的解析能正常执行）
   2.2. 准备
   准备是连接阶段的第二步，主要为静态变量在方法区分配内存，并设置默认初始值。
   2.3. 解析
   解析是连接阶段的第三步，是虚拟机将常量池内的符号引用替换为直接引用的过程。
2. 初始化
   初始化阶段是类加载过程的最后一步，主要是根据程序中的赋值语句主动为类变量赋值。
   当有继承关系时，先初始化父类再初始化子类，所以创建一个子类时其实内存中存在两个对象实例。
3. 使用
   程序之间的相互调用。
4. 卸载
   即销毁一个对象，一般情况下中有JVM垃圾回收器完成。代码层面的销毁只是将引用置为null。

14. JVM线上出OOM问题了如何定位？（和问题5类似）

15. 吞吐量优先和响应时间优先的回收器是哪些?

- 吞吐量优先：Parallel Scavenge+Parallel Old（多线程并行）
- 响应时间优先：cms+par new（并发回收垃圾）

16. 什么叫做阻塞队列的有界和无界，实际中有用过吗?

阻塞队列

- ArrayBlockingQueue：一个由**数组**结构组成的**有界**阻塞队列，线程池，生产者消费者
- LinkedBlockingQueue：一个由**链表**结构组成的**无界**阻塞队列，线程池，生产者消费者
- PriorityBlockingQueue：一个**支持优先级排序**的**无界**阻塞队列，可以实现精确的定时任务
- DelayQueue：一个**使用优先级队列**实现的**无界**阻塞队列，可以实现精确的定时任务
- SynchronousQueue：一个不存储元素的阻塞队列，线程池
- LinkedTransferQueue：一个由**链表**结构组成的**无界**阻塞队列
- LinkedBlockingDeque：一个由**链表**结构组成的**双向无界**阻塞队列，可以用在“工作窃取”模式中

17. jvm监控系统是通过jmx做的么?

一般都是，但是要是记录比较详细的性能定位指标，都会导致进入 safepoint，从而降低了线上应用性能
例如 jstack，jmap打印堆栈，打印内存使用情况，都会让 jvm 进入safepoint，才能获取线程稳定状态从而采集信息。
同时，JMX暴露向外的接口采集信息，例如使用jvisualvm，还会涉及rpc和网络消耗，以及JVM忙时，无法采集到信息从而有指标断点。这些都是基于 JMX 的外部监控很难解决的问题。
所以，推荐使用JVM内部采集 JFR，这样即使在JVM很忙时，也能采集到有用的信息

18. 内存屏障的汇编指令是啥?

1.硬件内存屏障 X86
sfence: store| 在sfence指令前的写操作当必须在sfence指令后的写操作前完成。
lfence：load | 在lfence指令前的读操作当必须在lfence指令后的读操作前完成。
mfence：modify/mix | 在mfence指令前的读写操作当必须在mfence指令后的读写操作前完成。
2.原子指令，如x86上的”lock ...” 指令是一个Full Barrier，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。Software Locks通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序。
3.JVM级别如何规范（JSR133）
LoadLoad屏障：
对于这样的语句Load1; LoadLoad; Load2，
在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
StoreStore屏障：
对于这样的语句Store1; StoreStore; Store2，
在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
LoadStore屏障：
对于这样的语句Load1; LoadStore; Store2，
在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
StoreLoad屏障：
对于这样的语句Store1; StoreLoad; Load2，
在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。

19. 怎么提前避免内存泄漏？（类似问题5）

多线程（马老师，黄老师）

1. 如何预防死锁?

1. 首先需要将死锁发生的是个必要条件讲出来:
    1. 互斥条件 同一时间只能有一个线程获取资源。
    2. 不可剥夺条件 一个线程已经占有的资源，在释放之前不会被其它线程抢占
    3. 请求和保持条件 线程等待过程中不会释放已占有的资源
    4. 循环等待条件 多个线程互相等待对方释放资源
1. 死锁预防，那么就是需要破坏这四个必要条件
    1. 由于资源互斥是资源使用的固有特性，无法改变，我们不讨论
    2. 破坏不可剥夺条件
        1. 一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行
    1. 破坏请求与保持条件
        1. 第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源,
        2. 第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源
    1. 破坏循环等待条件
        1. 采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

2. 多线程有哪几种创建方式?

1. 实现Runnable，Runnable规定的方法是run()，无返回值，无法抛出异常
2. 实现Callable，Callable规定的方法是call()，任务执行后有返回值，可以抛出异常
3. 继承Thread类创建多线程：继承java.lang.Thread类，重写Thread类的run()方法，在run()方法中实现运行在线程上的代码，调用start()方法开启线程。
   Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它将启动一个新线程，并执行 run()方法
4. 通过线程池创建线程. 线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

3. 描述一下线程安全活跃态问题，竞态条件?

1. 线程安全的活跃性问题可以分为 死锁、活锁、饥饿
    1. 活锁 就是有时线程虽然没有发生阻塞，但是仍然会存在执行不下去的情况，活锁不会阻塞线程，线程会一直重复执行某个相同的操作，并且一直失败重试

1. 我们开发中使用的异步消息队列就有可能造成活锁的问题，在消息队列的消费端如果没有正确的ack消息，并且执行过程中报错了，就会再次放回消息头，然后再拿出来执行，一直循环往复的失败。这个问题除了正确的ack之外，往往是通过将失败的消息放入到延时队列中，等到一定的延时再进行重试来解决。
2. 解决活锁的方案很简单，尝试等待一个随机的时间就可以，会按时间轮去重试

1. 饥饿 就是 线程因无法访问所需资源而无法执行下去的情况，

    1. 饥饿 分为两种情况：

        1. 一种是其他的线程在临界区做了无限循环或无限制等待资源的操作，让其他的线程一直不能拿到锁进入临界区，对其他线程来说，就进入了饥饿状态
        2. 另一种是因为线程优先级不合理的分配，导致部分线程始终无法获取到CPU资源而一直无法执行

    1. 解决饥饿的问题有几种方案:

        1. 保证资源充足，很多场景下，资源的稀缺性无法解决
        2. 公平分配资源，在并发编程里使用公平锁，例如FIFO策略，线程等待是有顺序的，排在等待队列前面的线程会优先获得资源
        3. 避免持有锁的线程长时间执行，很多场景下，持有锁的线程的执行时间也很难缩短

1. 死锁 线程在对同一把锁进行竞争的时候，未抢占到锁的线程会等待持有锁的线程释放锁后继续抢占，如果两个或两个以上的线程互相持有对方将要抢占的锁，互相等待对方先行释放锁就会进入到一个循环等待的过程，这个过程就叫做死锁

1. 线程安全的竞态条件问题

    1. 同一个程序多线程访问同一个资源，如果对资源的访问顺序敏感，就称存在竞态条件，代码区成为临界区。 大多数并发错误一样，竞态条件不总是会产生问题，还需要不恰当的执行时序
    2. 最常见的竞态条件为

        1. 先检测后执行执行依赖于检测的结果，而检测结果依赖于多个线程的执行时序，而多个线程的执行时序通常情况下是不固定不可判断的，从而导致执行结果出现各种问题，见一种可能 的解决办法就是：在一个线程修改访问一个状态时，要防止其他线程访问修改，也就是加锁机制，保证原子性
        2. 延迟初始化（典型为单例）

4. Java中的wait和sleep的区别与联系?

1. 所属类: 首先，这两个方法来自不同的类分别是Thread和Object ，wait是Object的方法，sleep是Thread的方法

    1. sleep方法属于Thread类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了sleep方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。但在sleep的过程中过程中有可能被其他对象调用它的interrupt(),产生InterruptedException异常，如果你的程序不捕获这个异常，线程就会异常终止，进入TERMINATED状态，如果你的程序捕获了这个异常，那么程序就会继续执行catch语句块(可能还有finally语句块)以及以后的代码

1. 作用范围: sleep方法没有释放锁，只是休眠，而wait释放了锁，使得其他线程可以使用同步控制块或方法
2. 使用范围: wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

3. 异常范围： sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

5. 描述一下进程与线程区别?

1. 进程（Process）

    1. 是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。程序是指令、数据及其组织形式的描述，进程是程序的实体。总结: j进程是指在系统中正在运行的一个应用程序；程序一旦运行就是进程；进程——资源分配的最小单位

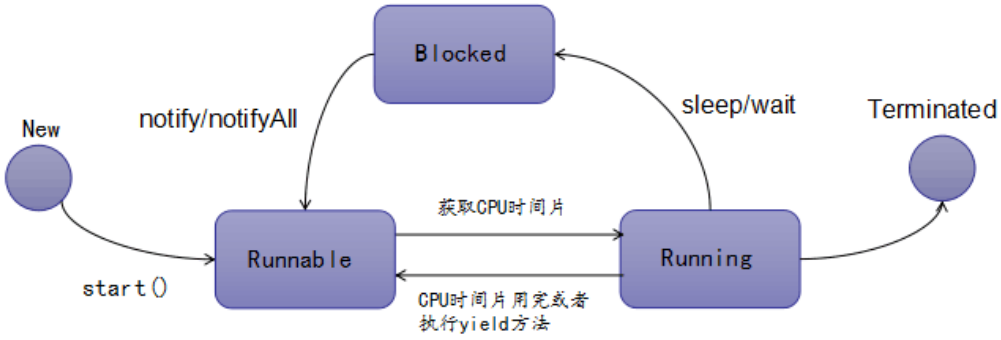
1. 线程

    1. 操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。总结: 系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元执行流。线程——程序执行的最小单位

6. 描述一下Java线程的生命周期?

1. 大致包括5个阶段

    1. 新建 就是刚使用new方法，new出来的线程；
    2. 就绪 就是调用的线程的start()方法后，这时候线程处于等待CPU分配资源阶段，谁先抢的CPU资源，谁开始执行;
    3. 运行 当就绪的线程被调度并获得CPU资源时，便进入运行状态，run方法定义了线程的操作和功能;
    4. 阻塞 在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态，比如sleep()、wait()之后线程就处于了阻塞状态，这个时候需要其他机制将处于阻塞状态的线程唤醒，比如调用notify或者notifyAll()方法。唤醒的线程不会立刻执行run方法，它们要再次等待CPU分配资源进入运行状态;
    5. 销毁 如果线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束，那么线程就要被销毁，释放资源

    ;

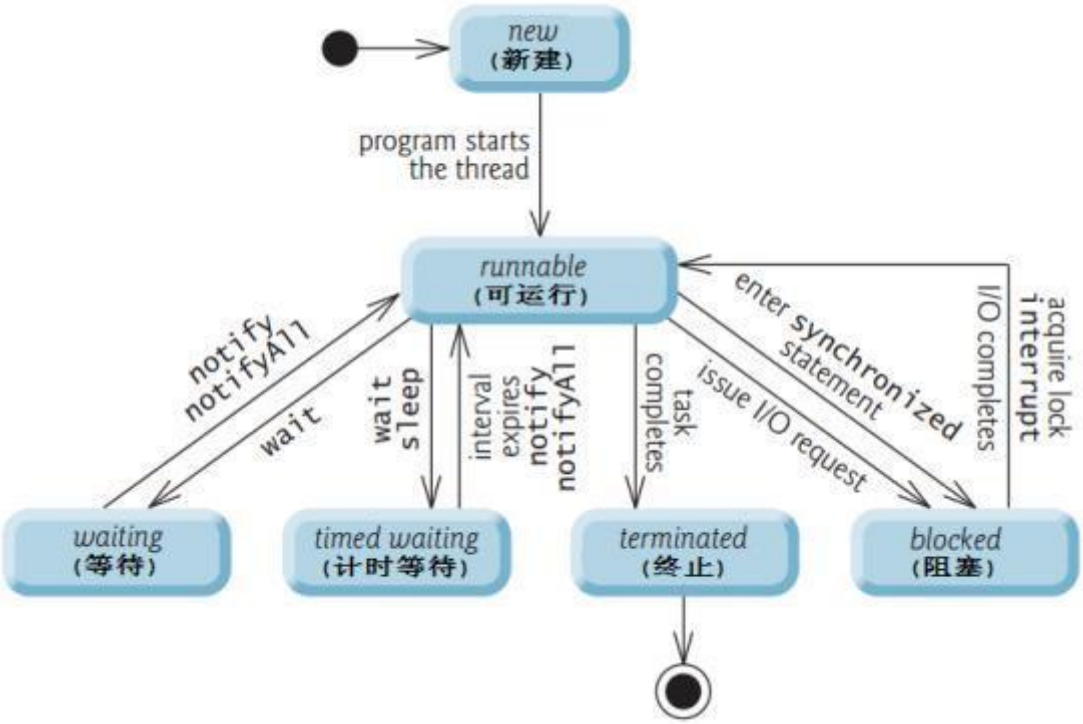
1. 按JDK的源码分析来看，Thread的状态分为：

    1. NEW： 尚未启动的线程的线程状态
    2. RUNNABLE： 处于可运行状态的线程正在Java虚拟机中执行，但它可能正在等待来自操作系统（例如处理器）的其他资源

3. BLOCKED： 线程的线程状态被阻塞，等待监视器锁定。处于阻塞状态的线程正在等待监视器锁定以输入同步的块方法或在调用后重新输入同步的块方法，通过 Object#wait() 进入阻塞
4. WAITING：处于等待状态的线程正在等待另一个线程执行特定操作：例如: 在对象上调用了Object.wait（）的线程正在等待另一个线程调用Object.notify（） 或者 Object.notifyAll(), 调用了 Thread.join（）的线程正在等待指定的线程终止
5. TIMED_WAITING： 具有指定等待时间的等待线程的线程状态。由于以指定的正等待时间调用以下方法之一，因此线程处于定时等待状态：

    1. Thread.sleep（long）
    2. Object#wait(long)
    3. Thread.join（long）
    4. LockSupport.parkNanos（long...）
    5. LockSupport.parkUntil（long...）

1. TERMINATED: 终止线程的线程状态。线程已完成执行



7. 程序开多少线程合适?

1. 这里需要区别下应用是什么样的程序：

    1. CPU 密集型程序， 一个完整请求，I/O操作可以在很短时间内完成， CPU还有很多运算要处理，也就是说 CPU 计算的比例占很大一部分，线程等待时间接近0

        1. 单核CPU： 一个完整请求，I/O操作可以在很短时间内完成， CPU还有很多运算要处理，也就是说 CPU 计算的比例占很大一部分，线程等待时间接近0。单核CPU处理CPU密集型程序，这种情况并不太适合使用多线程，
        2. 多核： 如果是多核CPU 处理 CPU 密集型程序，我们完全可以最大化的利用 CPU 核心数，应用并发编程来提高效率。CPU 密集型程序的最佳线程数就是：因此对于 CPU 密集型来说，理论上 线程数量 = CPU 核数（逻辑），但是实际上，数量一般会设置为 CPU 核数（逻辑）+ 1（经验值）
           计算(CPU)密集型的线程恰好在某时因为发生一个页错误或者因其他原因而暂停，刚好有一个“额外”的线程，可以确保在这种情况下CPU周期不会中断工作

    1. I/O 密集型程序，与 CPU 密集型程序相对，一个完整请求，CPU运算操作完成之后还有很多 I/O 操作要做，也就是说 I/O 操作占比很大部分，等待时间较长，线程等待时间所

占比例越高，需要越多线程；线程CPU时间所占比例越高，需要越少线程

1. I/O 密集型程序的最佳线程数就是： 最佳线程数 = CPU核心数 *(1/CPU利用率)* = *CPU核心数* (1 + (I/O耗时/CPU耗时))
2. 如果几乎全是 I/O耗时，那么CPU耗时就无限趋近于0，所以纯理论你就可以说是2N（N=CPU核数），当然也有说 2N + 1的，1应该是backup
3. 一般我们说 2N + 1 就即可

8. 描述一下notify和notifyAll区别?

1. 首先最好说一下 锁池 和 等待池 的概念
    1. 锁池:假设线程A已经拥有了某个对象(注意:不是类)的锁，而其它的线程想要调用这个对象的某个synchronized方法(或者synchronized块)，由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权，但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中。
    2. 等待池:假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁(因为wait()方法必须出现在synchronized中，这样自然在执行wait()方法之前线程A就已经拥有了该对象的锁)，同时线程A就进入到了该对象的等待池中。如果另外的一个线程调用了相同对象的notifyAll()方法，那么处于该对象的等待池中的线程就会全部进入该对象的锁池中，准备争夺锁的拥有权。如果另外的一个线程调用了相同对象的notify()方法，那么仅仅有一个处于该对象的等待池中的线程(随机)会进入该对象的锁池.

1. 如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁
2. 当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争
3. 所谓唤醒线程，另一种解释可以说是将线程由等待池移动到锁池，notifyAll调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而notify只会唤醒一个线程。

9. 描述一下synchronized和lock区别？

1. 如下表

区别类型	synchronized	Lock
存在层次	Java的关键字，在jvm层面上	是JVM的一个接口
锁的获取	假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待	情况而定，Lock有多个锁获取的方式，大致就是可以尝试获得锁，线程可以不用一直等待(可以通过tryLock判断有没有锁)
锁的释放	1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放	在finally中必须释放锁，不然容易造成线程死锁
锁类型	锁可重入、不可中断、非公平	可重入、可判断 可公平（两者皆可）
性能	少量同步	适用于大量同步
支持锁的场景	1. 独占锁	1. 公平锁与非公平锁

1. 可以再多说下 synchronized的 加锁流程
    1. 由于HotSpot的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低从而引入偏向锁。偏向锁在获取资源的时候会在锁对象头上记录当前线程ID，偏向锁并不会主动释放，这样每次偏向锁进入的时候都会判断锁对象头中线程ID是否为自己，如果是当前线程重入，直接进入同步操作，不需要额外的操作。默认在开启偏向锁和轻量锁的情况下，当线程进来时，首先会加上偏向锁，其实这里只是用一个状态来控制，会记录加锁的线程，如果是线程重入，则不会进行锁升级。获取偏向锁流程：
        1. 判断是否为可偏向状态--MarkWord中锁标志是否为‘01’，是否偏向锁是否为‘1’
        2. 如果是可偏向状态，则查看线程ID是否为当前线程，如果是，则进入步骤 'V'，否则进入步骤‘III’
        3. 通过CAS操作竞争锁，如果竞争成功，则将MarkWord中线程ID设置为当前线程ID，然后执行‘V’；竞争失败，则执行‘IV’
        4. CAS获取偏向锁失败表示有竞争。当达到safepoint时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块
        5. 执行同步代码

1. 轻量级锁是相对于重量级锁需要阻塞/唤醒涉及上下文切换而言，主要针对多个线程在不同时间请求同一把锁的场景。轻量级锁获取过程：

    1. 进行加锁操作时，jvm会判断是否已经是重量级锁，如果不是，则会在当前线程栈帧中划出一块空间，作为该锁的锁记录，并且将锁对象MarkWord复制到该锁记录中
    2. 复制成功之后，jvm使用CAS操作将对象头MarkWord更新为指向锁记录的指针，并将锁记录里的owner指针指向对象头的MarkWord。如果成功，则执行'III'，否则执行'IV'
    3. 更新成功，则当前线程持有该对象锁，并且对象MarkWord锁标志设置为'00'，即表示此对象处于轻量级锁状态
    4. 更新失败，jvm先检查对象MarkWord是否指向当前线程栈帧中的锁记录，如果是则执行'V'，否则执行'VI'
    5. 表示锁重入；然后当前线程栈帧中增加一个锁记录第一部分（Displaced Mark Word）为null，并指向Mark Word的锁对象，起到一个重入计数器的作用
    6. 表示该锁对象已经被其他线程抢占，则进行自旋等待（默认10次），等待次数达到阈值仍未获取到锁，则升级为重量级锁

1. 当有多个锁竞争轻量级锁则会升级为重量级锁，重量级锁正常会进入一个cxq的队列，在调用wait方法之后，则会进入一个waitSet的队列park等待，而当调用notify方法唤醒之后，则有可能进入EntryList。重量级锁加锁过程:

    1. 分配一个ObjectMonitor对象，把Mark Word锁标志置为'10'，然后Mark Word存储指向ObjectMonitor对象的指针。ObjectMonitor对象有两个队列和一个指针，每个需要获取锁的线程都包装成ObjectWaiter对象
    2. 多个线程同时执行同一段同步代码时，ObjectWaiter先进入EntryList队列，当某个线程获取到对象的monitor以后进入Owner区域，并把monitor中的owner变量设置为当前线程同时monitor中的计数器count+1；

10.简单描述一下ABA问题?

1. 有两个线程同时去修改一个变量的值，比如线程1、线程2，都更新变量值，将变量值从A更新成B。
2. 首先线程1、获取到CPU的时间片，线程2由于某些原因发生阻塞进行等待，此时线程1进行比较更新（CompareAndSwap），成功将变量的值从A更新成B。
3. 更新完毕之后，恰好又有线程3进来想要把变量的值从B更新成A，线程3进行比较更新，成功将变量的值从B更新成A。
4. 线程2获取到CPU的时间片，然后进行比较更新，发现值是预期的A，然后有更新成了B。但是线程1并不知道，该值已经有了
   A->B->A这个过程，这也就是我们常说的ABA问题。
5. 可以通过加版本号或者加时间戳解决，或者保证单向递增或者递减就不会存在此类问题。

11.实现一下DCL?

```
public class Singleton {
    //volatile是防止指令重排
    private static volatile Singleton singleton;
    private Singleton() {}
    public static Singleton getInstance() {
        //第一层判断singleton是不是为null
        //如果不为null直接返回，这样就不必加锁了
        if (singleton == null) {
            //现在再加锁
            synchronized (Singleton.class) {
                //第二层判断
```

```
                //如果A,B两个线程都在synchronized等待
                //A创建完对象之后，B还会再进入，如果不再检查一遍，B又会创建一个对象
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

12.实现一个阻塞队列（用Condition写生产者与消费者就）？

```
public class ProviderConsumer<T> {

    private int length;

    private Queue<T> queue;

    private ReentrantLock lock = new ReentrantLock();

    private Condition provideCondition = lock.newCondition();

    private Condition consumeCondition = lock.newCondition();

    public ProviderConsumer(int length){
        this.length = length;
        this.queue = new LinkedList<T>();
    }

    public void provide(T product){
        lock.lock();
        try {
            while (queue.size() >= length) {
                provideCondition.await();
            }
            queue.add(product);
            consumeCondition.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public T consume() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                consumeCondition.await();
            }
            T product = queue.remove();
            provideCondition.signal();
            return product;
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
```

```
            lock.unlock();
        }
        return null;
    }
}
```

13.实现多个线程顺序打印abc?

```
public class PrintABC {

    ReentrantLock lock = new ReentrantLock();
    Condition conditionA = lock.newCondition();
    Condition conditionB = lock.newCondition();
    Condition conditionC = lock.newCondition();
    volatile int value = 0;
    //打印多少遍
    private int count;

    public PrintABC (int count) {
        this.count = count;
    }

    public void printABC() {
        new Thread(new ThreadA()).start();
        new Thread(new ThreadB()).start();
        new Thread(new ThreadC()).start();
    }

    class ThreadA implements Runnable{

        @Override
        public void run() {
            lock.lock();
            try {
                for (int i = 0; i < count; i++) {
                    while (value % 3 != 0) {
                        conditionA.await();
                    }
                    System.out.print("A");
                    conditionB.signal();
                    value ++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }

    class ThreadB implements Runnable{


        @Override
        public void run() {
            lock.lock();
            try {
```

```
                for (int i = 0; i < count; i++) {
                    while (value % 3 != 1) {
                        conditionB.await();
                    }
                    System.out.print("B");
                    conditionC.signal();
                    value ++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }

    class ThreadC implements Runnable{

        @Override
        public void run() {
            lock.lock();
            try {
                for (int i = 0; i < count; i++) {
                    while ( value % 3 != 2) {
                        conditionC.await();
                    }
                    System.out.println("C");
                    conditionA.signal();
                    value ++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) {
        PrintABC printABC = new PrintABC(15);
        printABC.printABC();
    }

}
```

14.服务器CPU数量及线程池线程数量的关系?

首先确认业务是CPU密集型还是IO密集型的，

如果是CPU密集型的，那么就应该尽量少的线程数量，一般为CPU的核数+1；

如果是IO密集型：所以可多分配一点 cpu核数*2 也可以使用公式：CPU 核数 / (1 - 阻塞系数)；其中阻塞系数　在 0.8 ~ 0.9 之间。

15.多线程之间是如何通信的?

1、通过共享变量，变量需要volatile 修饰

2、使用wait()和notifyAll()方法，但是由于需要使用同一把锁，所以必须通知线程释放锁，被通知线程才能获取到锁，这样导致通知不及时。

3、使用CountDownLatch实现，通知线程到指定条件，调用countDownLatch.countDown()，被通知线程进行countDownLatch.await()。

4、使用Condition的await()和signalAll()方法。

16.描述一下synchronized底层实现，以及和lock的区别?

详见第九题

17.synchronized关键字加在静态方法和实例方法的区别?

修饰静态方法，是对类进行加锁，如果该类中有methodA 和methodB都是被synchronized修饰的静态方法，此时有两个线程T1、T2分别调用methodA()和methodB()，则T2会阻塞等待直到T1执行完成之后才能执行。

修饰实例方法时，是对实例进行加锁，锁的是实例对象的对象头，如果调用同一个对象的两个不同的被synchronized修饰的实例方法时，看到的效果和上面的一样，如果调用不同对象的两个不同的被synchronized修饰的实例方法时，则不会阻塞。

18.countdownlatch的用法?

两种用法：

1、让主线程await，业务线程进行业务处理，处理完成时调用countdownLatch.countDown()，CountDownLatch实例化的时候需要根据业务去选择CountDownLatch的count;

2、让业务线程await，主线程处理完数据之后进行countdownLatch.countDown()，此时业务线程被唤醒，然后去主线程拿数据，或者执行自己的业务逻辑。

19.线程池问题：

（1）Executor提供了几种线程池

1、newCachedThreadPool()（工作队列使用的是 SynchronousQueue）

创建一个线程池，如果线程池中的线程数量过大，它可以有效的回收多余的线程，如果线程数不足，那么它可以创建新的线程。

不足：这种方式虽然可以根据业务场景自动的扩展线程数来处理我们的业务，但是最多需要多少个线程同时处理却是我们无法控制的。

优点：如果当第二个任务开始，第一个任务已经执行结束，那么第二个任务会复用第一个任务创建的线程，并不会重新创建新的线程，提高了线程的复用率。

作用：该方法返回一个可以根据实际情况调整线程池中线程的数量的线程池。即该线程池中的线程数量不确定，是根据实际情况动态调整的。

2、newFixedThreadPool()（工作队列使用的是 LinkedBlockingQueue）

这种方式可以指定线程池中的线程数。如果满了后又来了新任务，此时只能排队等待。

优点：newFixedThreadPool 的线程数是可以进行控制的，因此我们可以通过控制最大线程来使我们的服务器达到最大的使用率，同时又可以保证即使流量突然增大也不会占用服务器过多的资源。

作用：该方法返回一个固定线程数量的线程池，该线程池中的线程数量始终不变，即不会再创建新的线程，也不会销毁已经创建好的线程，自始自终都是那几个固定的线程在工作，所以该线程池可以控制线程的最大并发数。

3、newScheduledThreadPool()

该线程池支持定时，以及周期性的任务执行，我们可以延迟任务的执行时间，也可以设置一个周期性的时间让任务重复执行。该线程池中有以下两种延迟的方法。

scheduleAtFixedRate 不同的地方是任务的执行时间，如果间隔时间大于任务的执行时间，任务不受执行时间的影响。如果间隔时间小于任务的执行时间，那么任务执行结束之后，会立马执行，至此间隔时间就会被打乱。

scheduleWithFixedDelay 的间隔时间不会受任务执行时间长短的影响。

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。

4、newSingleThreadExecutor()

这是一个单线程池，至始至终都由一个线程来执行。

作用：该方法返回一个只有一个线程的线程池，即每次只能执行一个线程任务，多余的任务会保存到一个任务队列中，等待这一个线程空闲，当这个线程空闲了再按 FIFO 方式顺序执行任务队列中的任务。

5、newSingleThreadScheduledExecutor()

只有一个线程，用来调度任务在指定时间执行。

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。只不过和上面的区别是该线程池大小为 1，而上面的可以指定线程池的大小。

(2) 线程池的参数

```
int corePoolSize,//线程池核心线程大小
int maximumPoolSize,//线程池最大线程数量
long keepAliveTime,//空闲线程存活时间
TimeUnit unit,//空闲线程存活时间单位，一共有七种静态属性(TimeUnit.DAYS天,TimeUnit.HOURS小时,TimeUnit.MINUTES分钟,TimeUnit.SECONDS秒,TimeUnit.MILLISECONDS毫秒,TimeUnit.MICROSECONDS微妙,TimeUnit.NANOSECONDS纳秒)
BlockingQueue<Runnable> workQueue,//工作队列
ThreadFactory threadFactory,//线程工厂，主要用来创建线程(默认的工厂方法是:Executors.defaultThreadFactory()对线程进行安全检查并命名)
RejectedExecutionHandler handler//拒绝策略(默认是：ThreadPoolExecutor.AbortPolicy不执行并抛出异常)
```

(3) 拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，就会执行拒绝策略。

jdk中提供了4中拒绝策略：

①ThreadPoolExecutor.CallerRunsPolicy

该策略下，在调用者线程中直接执行被拒绝任务的 run 方法，除非线程池已经 shutdown，则直接抛弃任务。

②ThreadPoolExecutor.AbortPolicy

该策略下，直接丢弃任务，并抛出 RejectedExecutionException 异常。

③ThreadPoolExecutor.DiscardPolicy

该策略下，直接丢弃任务，什么都不做。

④ThreadPoolExecutor.DiscardOldestPolicy

该策略下，抛弃进入队列最早的那个任务，然后尝试把这次拒绝的任务放入队列。

除此之外，还可以根据应用场景需要来实现 RejectedExecutionHandler 接口自定义策略。

（4）任务放置的顺序过程

任务调度是线程池的主要入口，当用户提交了一个任务，接下来这个任务将如何执行都是由这个阶段决定的。了解这部分就相当于了解了线程池的核心运行机制。

首先，所有任务的调度都是由execute方法完成的，这部分完成的工作是：检查现在线程池的运行状态、运行线程数、运行策略，决定接下来执行的流程，是直接申请线程执行，或是缓冲到队列中执行，亦或是直接拒绝该任务。其执行过程如下：

首先检测线程池运行状态，如果不是RUNNING，则直接拒绝，线程池要保证在RUNNING的状态下执行任务。
如果workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务。
如果workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中。
如果workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务。
如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。
其执行流程如下图所示：

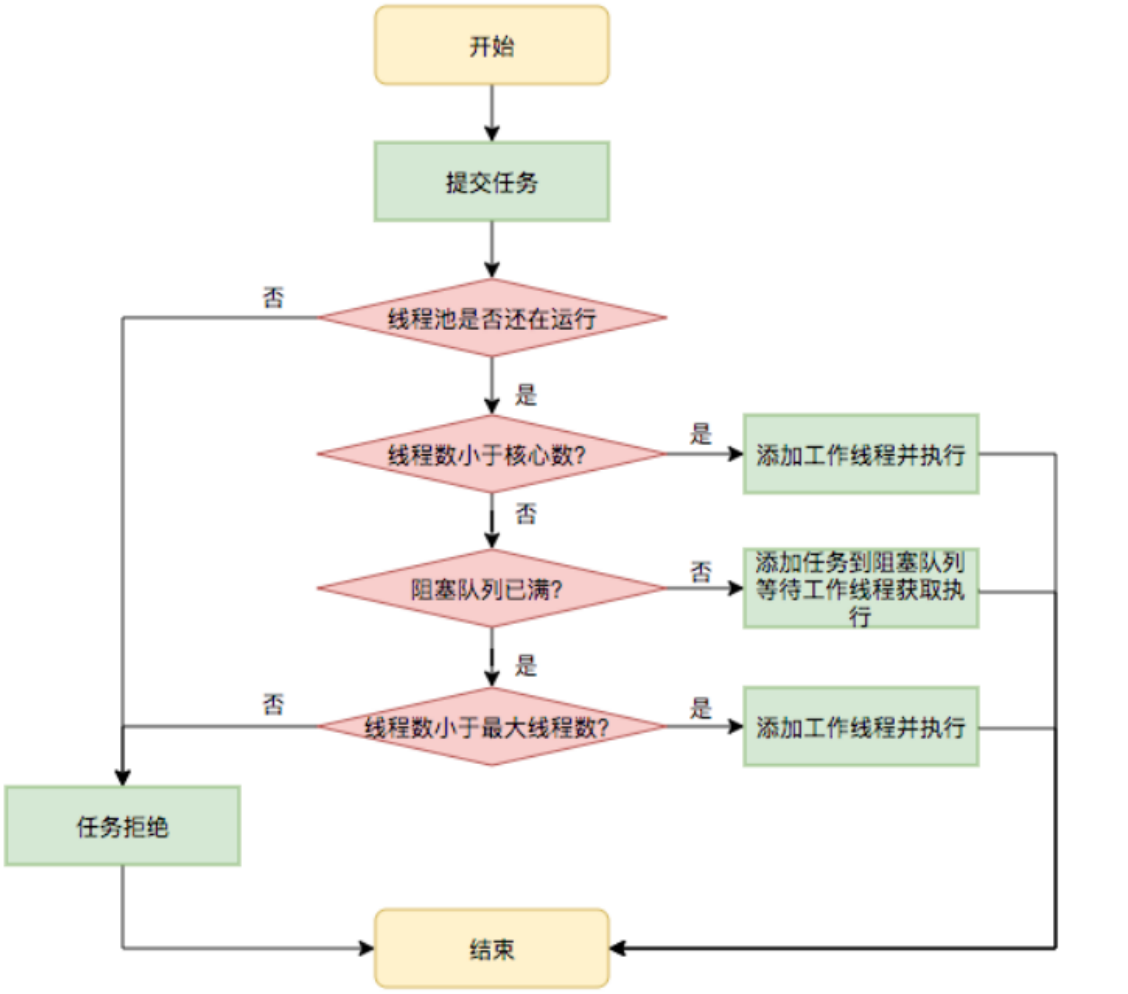


图4 任务调度流程

(5) 任务结束后会不会回收线程

根据情况.

```
/java/util/concurrent/ThreadPoolExecutor.java:1127
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            while (task != null || (task = getTask()) != null) {...执行任务...}
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
        }
    }
```

首先线程池内的线程都被包装成了一个个的java.util.concurrent.ThreadPoolExecutor.Worker,然后这个worker会马不停蹄的执行任务,执行完任务之后就会在while循环中去取任务,取到任务就继续执行,取不到任务就跳出while循环(这个时候worker就不能再执行任务了)执行 processWorkerExit方法,这个方法呢就是做清场处理,将当前woker线程从线程池中移除,并且判断是否是异常进入processWorkerExit方法,如果是非异常情况,就对当前线程池状态(RUNNING,shutdown)和当前工作线程数和当前任务数做判断,是否要加入一个新的线程去完成最后的任务.

那么什么时候会退出while循环呢?取不到任务的时候.下面看一下getTask方法

```
private Runnable getTask() {
```

```
        boolean timedOut = false; // Did the last poll() time out?

        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);

            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                decrementWorkerCount();
                return null;
            }

            int wc = workerCountOf(c);

            // Are workers subject to culling?
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

            try {
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

（6）未使用的线程池中的线程放在哪里

```
    private final HashSet<Worker> workers = new HashSet<Worker>();
```

（7）线程池线程存在哪

```
    private final HashSet<Worker> workers = new HashSet<Worker>();
```

（8）cache线程池会不会销毁核心线程

20.Java多线程的几种状态及线程各个状态之间是如何切换的?

运行状态	状态描述
RUNNING	能接受新提交的任务,并且也能处理阻塞队列中的任务
SHUTDOWN	关闭状态,不再接受新提交的任务.但却可以继续处理阻塞队列中已经保存的任务
STOP	不能接收新任务,也不处理队列中的任务,会中断正在处理的线程
TIDYING	所有的任务已经终止,wokerCount = 0
TERMINATED	在terminated()方法执行后进入此状态

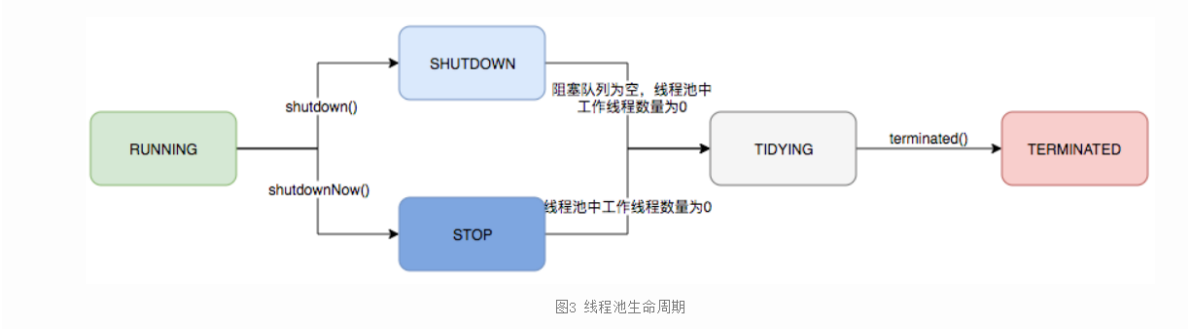


图3 线程池生命周期

21.如何在方法栈中进行数据传递?

通过方法参数传递;通过共享变量;如果在用一个线程中,还可以使用ThreadLocal进行传递.

22.描述一下ThreadLocal的底层实现形式及实现的数据结构?

Thread类中有两个变量threadLocals和inheritableThreadLocals，二者都是ThreadLocal内部类ThreadLocalMap类型的变量，我们通过查看内部内ThreadLocalMap可以发现实际上它类似于一个HashMap。在默认情况下，每个线程中的这两个变量都为null:

```
ThreadLocal.ThreadLocalMap threadLocals = null;
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

只有当线程第一次调用ThreadLocal的set或者get方法的时候才会创建他们。

```
public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
}

ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
}
```

除此之外，和我所想的不同的是，每个线程的本地变量不是存放在ThreadLocal实例中，而是放在调用线程的ThreadLocals变量里面。也就是说，ThreadLocal类型的本地变量是存放在具体的线程空间上，其本身相当于一个装载本地变量的工具壳，通过set方法将value添加到调用线程的threadLocals中，当调用线程调用get方法时候能够从它的threadLocals中取出变量。如果调用线程一直不终止，那么这个本地变量将会一直存放在他的threadLocals中，所以不使用本地变量的时候需要调用remove方法将threadLocals中删除不用的本地变量,防止出现内存泄漏。

```
public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
}
public void remove() {
         ThreadLocalMap m = getMap(Thread.currentThread());
         if (m != null)
             m.remove(this);
}
```

23.Sychornized是否是公平锁?

不是公平锁

24.Sychronized和ReentryLock的区别?

请看第9题

25.描述一下线程池的创建方式、分类、应用场景、拒绝策略的场景?

请看第19题.

26.描述一下锁的四种状态及升级过程?

以下是32位的对象头描述

锁状态	25 bit	4bit	1bit	2bit	
23bit	2bit	是否是偏向锁	锁标志位		
轻量级锁	指向栈中锁记录的指针	00			
重量级锁	指向互斥量（重量级锁）的指针	10			
GC标记	空	11			
偏向锁	线程ID	Epoch	对象分代年龄	1	01

synchronized锁的膨胀过程：
当线程访问同步代码块。首先查看当前锁状态是否是偏向锁(可偏向状态)
1、如果是偏向锁：
1.1、检查当前mark word中记录是否是当前线程id，如果是当前线程id，则获得偏向锁执行同步代码块。
1.2、如果不是当前线程id，cas操作替换线程id，替换成功获得偏向锁(线程复用)，替换失败锁撤销升级轻量锁(同一类对象多次撤销升级达到阈值20，则批量重偏向,这个点可以稍微提一下,详见下面的注意)
2、升级轻量锁

升级轻量锁对于当前线程，分配栈帧锁记录lock_record(包含mark word和object-指向锁记录首地址)，对象头mark word复制到线程栈帧的锁记录 mark word存储的是无锁的hashcode(里面有重入次数问题)。
3、重量级锁(纯理论可结合源码)
CAS自旋达到一定次数升级为重量级锁(多个线程同时竞争锁时)
存储在ObjectMonitor对象，里面有很多属性ContentionList、EntryList 、WaitSet、owner。当一个线程尝试获取锁时，如果该锁已经被占用，则该线程封装成ObjectWaiter对象插到ContentionList队列的对首，然后调用park挂起。该线程锁时方式会从ContentionList或EntryList挑一个唤醒。线程获得锁后调用Object的wait方法，则会加入到WaitSet集合中(当前锁或膨胀为重量级锁)

注意:
1.偏向锁在JDK1.6以上默认开启，开启后程序启动几秒后才会被激活
2.偏向锁撤销是需要在safe_point,也就是安全点的时候进行,这个时候是stop the word的,所以说偏向锁的撤销是开销很大的,如果明确了项目里的竞争情况比较多,那么关闭偏向锁可以减少一些偏向锁撤销的开销
3.以class为单位，为每个class维护一个偏向锁撤销计数器。每一次该class的对象发生偏向撤销操作时(这个时候进入轻量级锁)，该计数器+1，当这个值达到重偏向阈值(默认20,也就是说前19次进行加锁的时候,都是假的轻量级锁,当第20次加锁的时候,就会走批量冲偏向的逻辑)时，JVM就认为该class的偏向锁有问题，因此会进行批量重偏向。每个class对象也会有一个对应的epoch字段，每个处于偏向锁状态对象的mark word中也有该字段，其初始值为创建该对象时，class中的epoch值。每次发生批量重偏向时，就将该值+1，同时遍历JVM中所有线程的站，找到该class所有正处于加锁状态的偏向锁，将其epoch字段改为新值。下次获取锁时，发现当前对象的epoch值和class不相等，那就算当前已经偏向了其他线程，也不会执行撤销操作，而是直接通过CAS操作将其mark word的Thread Id改为当前线程ID
4.需要看源码的同学:https://github.com/farmerjohngit/myblog/issues/12

27.描述一下CMS和G1的异同?

CMS只对老年代进行收集，采用“**标记-清除**”算法，会出现内存碎片，但是可以设置；而G1使用了独立区域（Region）概念，G1从整体来看是基于“**标记-整理**”算法实现收集，从局部（两个Region）上来看是基于“**复制**”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片尤其是当 Java 堆非常大的时候，G1 的优势更加明显，并且G1建立了可预测的停顿时间模型， 可以直观的设定停顿时间的目标，减少每一次的垃圾收集时间，相比于 CMS GC，G1 未必能做到 CMS 在最好情况下的延时停顿，但是最差情况要好很多。

28. G1什么时候引发Full GC?

1. Evacuation的时候没有足够的to-space来存放晋升的对象;
2. 并发处理过程完成之前空间耗尽。

29. 除了CAS，原子类，syn，Lock还有什么线程安全的方式?

park() 、信号量semaphore

30. 描述一下HashMap和Hashtable的异同。

1.两者最主要的区别在于Hashtable是线程安全，而HashMap则非线程安全。
2.key、value都是对象，但是不能拥有重复key值，value值可以重复出现。
1.Hashtable中，key和value都不允许出现null值。
2.HashMap允许null值(key和value都可以)，因为在HashMap中null可以作为健，而它对应的值可以有多个null。
3.Hashtable 是线程安全的，每个方法都要阻塞其他线程，所以 Hashtable 性能较差，HashMap 性能较好，使用更广。
4.Hashtable 继承了 Dictionary类，而 HashMap 继承的是 AbstractMap 类

31. CAS的ABA问题怎么解决的?

通过加版本号控制，只要有变更，就更新版本号

32. 描述一下AQS?

状态变量state
AQS中定义了一个状态变量state，它有以下两种使用方法：
（1）互斥锁
当AQS只实现为互斥锁的时候，每次只要原子更新state的值从0变为1成功了就获取了锁，可重入是通过不断把state原子更新加1实现的。
（2）互斥锁 + 共享锁
当AQS需要同时实现为互斥锁+共享锁的时候，低16位存储互斥锁的状态，高16位存储共享锁的状态，主要用于实现读写锁。
互斥锁是一种独占锁，每次只允许一个线程独占，且当一个线程独占时，其它线程将无法再获取互斥锁及共享锁，但是它自己可以获取共享锁。
共享锁同时允许多个线程占有，只要有一个线程占有了共享锁，所有线程（包括自己）都将无法再获取互斥锁，但是可以获取共享锁。
AQS队列
AQS中维护了一个队列，获取锁失败（非tryLock()）的线程都将进入这个队列中排队，等待锁释放后唤醒下一个排队的线程（互斥锁模式下）。
condition队列
AQS中还有另一个非常重要的内部类ConditionObject，它实现了Condition接口，主要用于实现条件锁。
ConditionObject中也维护了一个队列，这个队列主要用于等待条件的成立，当条件成立时，其它线程将signal这个队列中的元素，将其移动到AQS的队列中，等待占有锁的线程释放锁后被唤醒。
Condition典型的运用场景是在BlockingQueue中的实现，当队列为空时，获取元素的线程阻塞在notEmpty条件上，一旦队列中添加了一个元素，将通知notEmpty条件，将其队列中的元素移动到AQS队列中等待被唤醒。
模板方法
AQS这个抽象类把模板方法设计模式运用地炉火纯青，它里面定义了一系列的模板方法，比如下面这些：

```
// 获取互斥锁
public final void acquire(int arg) {
    // tryAcquire(arg)需要子类实现
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
// 获取互斥锁可中断
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // tryAcquire(arg)需要子类实现
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}
// 获取共享锁
public final void acquireShared(int arg) {
    // tryAcquireShared(arg)需要子类实现
    if (tryAcquireShared(arg) < 0)
     doAcquireShared(arg);
```

```
}
// 获取共享锁可中断
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // tryAcquireShared(arg)需要子类实现
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
// 释放互斥锁
public final boolean release(int arg) {
    // tryRelease(arg)需要子类实现
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
// 释放共享锁
public final boolean releaseShared(int arg) {
    // tryReleaseShared(arg)需要子类实现
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

获取锁、释放锁的这些方法基本上都穿插在ReentrantLock、ReentrantReadWriteLock、Semaphore、CountDownLatch的源码解析中

需要子类实现的方法

上面一起学习了AQS中几个重要的模板方法，下面我们再一起学习下几个需要子类实现的方法：

```
// 互斥模式下使用：尝试获取锁
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
// 互斥模式下使用：尝试释放锁
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
// 共享模式下使用：尝试获取锁
protected int tryAcquireShared(int arg) {
    throw new UnsupportedOperationException();
}
// 共享模式下使用：尝试释放锁
protected boolean tryReleaseShared(int arg) {
    throw new UnsupportedOperationException();
}
// 如果当前线程独占着锁，返回true
protected boolean isHeldExclusively() {
    throw new UnsupportedOperationException();
}
```

这几个方法为什么不直接定义成抽象方法呢?
因为子类只要实现这几个方法中的一部分就可以实现一个同步器了，所以不需要定义成抽象方法。

33. JUC包里的同步组件主要实现了AQS的哪些主要方法？（见问题32)

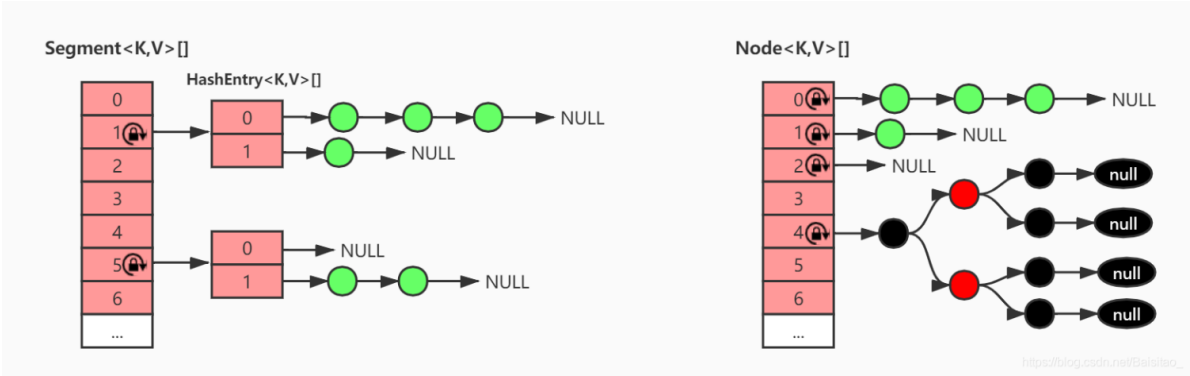
34. 介绍一下volatile的功能?

- 保证线程可见性
- 防止指令重排序

35. volatile的可见性和禁止指令重排序怎么实现的?

- 可见性:
  volatile的功能就是被修饰的变量在被修改后可以立即同步到主内存，被修饰的变量在每次是用之前都从主内存刷新。本质也是通过内存屏障来实现可见性
  写内存屏障（Store Memory Barrier）可以促使处理器将当前store buffer（存储缓存）的值写回主存。读内存屏障（Load Memory Barrier）可以促使处理器处理invalidate queue（失效队列）。进而避免由于Store Buffer和Invalidate Queue的非实时性带来的问题。
- 禁止指令重排序:
  volatile是通过**内存屏障**来禁止指令重排序
  JMM内存屏障的策略
  - 在每个 volatile 写操作的前面插入一个 StoreStore 屏障。
  - 在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。
  - 在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。
  - 在每个 volatile 读操作的后面插入一个 LoadStore 屏障。

36. 简要描述一下ConcurrentHashMap底层原理?



JDK1.7中的ConcurrentHashMap
内部主要是一个Segment数组，而数组的每一项又是一个HashEntry数组，元素都存在HashEntry数组里。因为每次锁定的是Segment对象，也就是整个HashEntry数组，所以又叫分段锁。
JDK1.8中的ConcurrentHashMap
舍弃了分段锁的实现方式，元素都存在Node数组中，每次锁住的是一个Node对象，而不是某一段数组，所以支持的写的并发度更高。
再者它引入了红黑树，在hash冲突严重时，读操作的效率更高。