JAVA 面试题

基础

1、String 编码 UTF-8 和 GBK 的区别?

UTF8 是国际编码，它的通用性比较好
GBK 是国家编码，通用性比 UTF8 差，不过 UTF8 占用的数据库比 GBK 大~

GBK 的文字编码是双字节来表示的，即不论中、英文字符均使用双字节来表示，
UTF-8 英文使用 8 位（即一个字节），中文使用 24 为（三个字节）来编码。

2、字节流与字符流的区别

InputStream 和 OutputStream,两个是为字节流设计的,主要用来处理**字节或二进制**对象,
Reader 和 Writer.两个是为字符流（**一个字符占两个字节**）设计的,主要用来处理**字符或字符串.**
实际上字节流在操作时本身不会用到缓冲区（内存），是文件本身直接操作的，而**字符流在操作时使用了缓冲区**，通过缓冲区再操作文件，

3、cookie 和 session 的区别

1、cookie 将数据存放在客户端，session 存放在服务器端
2、cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗，
考虑到安全应当使用 session
3、session 会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能
4、单个 cookie 保存的数据不能超过 4k，很多浏览器都限制一个站点最多保存 20 个 cookie

4、什么是 java 序列化，如何实现 java 序列化

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行**流化**。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。

5、为什么 String 是不可变对象

不可变对象是指在创建后其外部可见状态无法更改的对象，不可变对象可以提高 String Pool

的效率和安全性，不可变对象对于多线程是安全的。
当然也有其他方面原因，但是 Java 把 String 设成 immutable 最大的原因应该是**效率和安全**。

集合

1、ArrayList 和 Vector 的区别

1、ArrayList 和 Vector 都实现了 List 接口，底层都是基于 Java 数组来存储集合元素
2、ArrayList 使用 transient 修饰了 elementData 数组，而 Vector 则没有
3、Vector 是 ArrayList 的线程安全版本
4、容量扩充 ArrayList 为 0.5 倍+1，而 Vector 若指定了增长系数，则新的容量=”原始容量+增长系数”，否则增长为原来的 1 倍

参考：http://blog.itmyhome.com/2017/07/java-arraylist-vector

2、HashMap 和 Hashtable 的区别

相同点：
都是存储键值对(key-value)的散列表，通过 table 数组存储，数组的每一个元素都是一个 Entry 而一个 Entry 就是一个单向链表，Entry 链表中的每一个节点保存了 key-value 键值对数据

不同点：
1、继承和实现方式不同
2、线程安全不同
3、对 null 值处理不同
4、支持的遍历种类不同
5、容量的初始值和增加方式不同
6、添加 key-value 时的 hash 值算法不同

参考：http://blog.itmyhome.com/2017/08/hashmap-hashtable-difference

3、List 和 Map 区别

一个是存储单列数据的集合，另一个是存储键和值这样的双列数据的集合
List 中存储的数据是有顺序的，并且**允许重复**，Map 中存储的数据是没有顺序的，其键是不能重复的他的值是可以重复的。

4、List、Map、Set 三个接口，存取元素时，各有什么特点

1、List 和 Set 继承自 Collection 接口，而 Map 不是继承的 Collection 接口

Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 元素。
一些 Collection 允许相同的元素而另一些不行，一些能排序而另一些不行。JDK 中不提供此接口的任何直接实现，它提供更具体的子接口(如 List、Set)。Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射，不能包含相同的 key

2、List 接口

元素有放入顺序，元素可重复

List 接口有三个实现类：LinkedList, ArrayList，Vector

LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢。

ArrayList 和 Vector 的区别：ArrayList 是非线程安全的,效率高；Vector 是线程安全的，效率低

3、Set 接口

元素无放入顺序，元素不可重复

Set 接口有两个实现类：HashSet(底层由 HashMap 实现)，LinkedHashSet

SortedSet 接口有一个实现类：TreeSet(底层由平衡二叉树实现)

4、Map 接口

以键值对的方式出现的

Map 接口有三个实现类：HashMap，Hashtable, LinkedHashMap

HashMap 非线程安全，高效，支持 null；

HashTable 线程安全，低效，不支持 null

SortedMap 有一个实现类：TreeMap

5、说出 ArrayList、Vector、LinkedList 的存储性能和特性

ArrayList 和 Vector 都是使用**数组**方式存储数据，此数据元素大于实际存储的数据以便增加和插入元素，他们都运行直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以**索引数据快而插入数据慢**

Vector 由于使用了 **synchronized 方法(线程安全)** 通常性能上较 ArrayList 差

而 LinkedList 使用**双向链表**实现存储，按序号索引数据需要进行前后或后向遍历，但是插入数据时只需记录本项的前后项即可，所以**插入速度较快**。

LinkedList 也是线程不安全的，LinkedList 提供了一些方法(get、remove、insert)，使得 LinkedList 可以被当作堆栈和队列来使用

6、HashMap 底层

HashMap 采用"Hash 算法"来决定每个元素的存储位置，当程序执行 put 方法时，系统将调用 hashCode()方法得到其 hashCode 值，得到这个对象的 hashCode 值之后，系统会根据该 hashCode 值来决定该元素的存储位置

HashMap 是采用数组加链表的存储方式来实现的。

put 源码：

首先判断 key 是否为 null，如果为 null 调用 putForNullKey 方法进行处理
调用 hash()方法根据 key 计算 Hash 值
搜索指定 hash 值在对应 table 中的索引

根据源码可以看出，当程序试图将一个 key-value 对放入 HashMap 中时，首先根据 key 的 hashCode()返回值决定该 Entry 的存储位置，如果返回值相同，那它们的存储位置相同；如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value

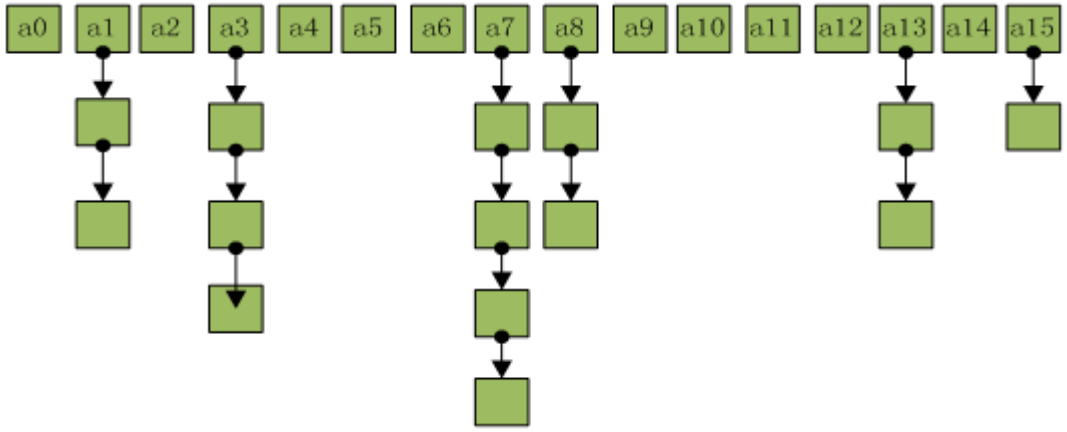
还调用了 addEntry()方法，涉及到两个变量: size(初始容量)、threshold(负载因子)

get 源码

当需要存储一个 Entry 对象时，会根据 Hash 算法来决定其存储位置；当需要取出一个 Entry 时，也会根据 Hash 算法找到其存储位置，直接取出该 Entry。

7、HashMap 存储结构

HashMap 采取数组加链表的存储方式来实现。亦即数组（散列桶）中的每一个元素都是链表，如下图：



8、HashMap、LinkedHashMap、TreeMap 的区别

HashMap 是一个最常用的 Map，它根据键的 hashCode 值存储数据，根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。HashMap 最多只允许一条记录的键为 null，不支持线程的同步。

LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，在用 Iterator 遍历

LinkedHashMap 时，先得到的记录肯定是先插入的。

TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器。

9、Collection 和 Collections 的区别

Collection 是集合类的上级接口，继承与它的接口主要有 Set 和 List
Collections 是针对集合类的一个帮助类，提供一系列静态方法(addAll、binarySearch、copy、fill、reverse，sort)实现对各种集合的搜索、排序、线程安全化等操作

线程、并发

1、Java 中有几种方法可以实现一个线程？

1、继承 Thread 类
2、实现 Runnable 接口
3、使用 ExecutorService、Callable、Future 实现有返回结果的多线程(JDK5.0 以后)

2、线程的几种状态

新建、就绪、运行、阻塞、死亡

3、如何停止一个正在运行的线程？

使用 interrupt

4、notify()和 notifyAll()有什么区别？

notify 表示当前的线程已经放弃对资源的占有，通知等待的线程来获得对资源的占有权。但是只有线程能够从 wait 状态中回复，然后继续运行 wait 方面的语句。

notifyAll 表示当前的线程已经放弃对资源的占有，通知所有的等待线程来 wait 方法后面的语句开始运行

5、sleep()和 wait()有什么区别?

sleep 是 Thread 类的静态方法，sleep 的作用是让线程休眠指定的时间，在时间到达时恢复，

也就是说 sleep 将在时间到达后恢复线程执行。调用 sleep 不会释放对象锁。
wait 是 Object 中的方法，也就是说可以对任意一个对象调用 wait 方法，调用 wait 方法将会将线程挂起直到使用 notify 方法才能重新唤醒。线程休眠的同时释放掉机锁

6、Java 如何实现多线程之间的通讯和协作

生产者和消费者模式，wait()、notify()方法

7、什么是线程安全？线程安全是怎么完成的

线程安全就是说多线程访问同一代码，不会产生不确定的结果。编写线程安全的代码是依靠线程同步，线程安全一般都涉及到 synchronized 就是一段代码同时只能有一个线程来操作

8、servlet 是线程安全的吗

不是。当 Tomcat 接收到 Client 的 HTTP 请求时，Tomcat 从线程池中取出一个线程，之后找到该请求对应的 Servlet 对象并初始化，
调用 service()方法，只有一个实例对象。如果在 servlet 中定义了实例变量或静态变量那么可能会发生线程安全问题

9、同步有几种实现方法

- wait() 使一个线程处于等待状态
- sleep()使一个正在运行的线程处于睡眠状态
- notify()唤醒一个处于等待状态的线程
- allnotify()唤醒所有处于等待状态的线程

10、volatile 有什么用？能否用一句话说明下 volatile 的应用场景？

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：

1、保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2、禁止进行指令重排序。

应用场景：
状态标记量、double check(双重检查)

11、HashMap 与 ConcurrentHashMap 的区别

我们知道 HashMap 不是线程安全的，Hashtable 是线程安全的，同步的，synchronized 是针对整张 Hash 表的，即每次锁住 整张表让线程独占，ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了**锁分离技术**。它使用了多个锁来控制对 hash 表的不同部分进行的修改。
ConcurrentHashMap 内部使用**段(Segment)**来表示这些不同的部分，每个段其实就是一个小的 hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

ConcurrentHashMap 是 Java5 中新增的一个线程安全的 Map 集合，可以用来代替 Hashtable，它使用了锁分离技术

12、ConcurrentHashMap 的实现原理

ConcurrentHashMap 是线程安全的哈希表，它是通过“锁分段”来实现的。ConcurrentHashMap 中包括了“Segment(锁分段)数组”，每个 Segment 就是一个哈希表，而且也是可重入的互斥锁。第一，Segment 是哈希表 表现在，Segment 包含了“HashEntry 数组”，而“HashEntry 数组”中的每一个 HashEntry 元素是一个单向链表。即 Segment 是通过链式哈希表。第二，Segment 是可重入的互斥锁表现在，Segment 继承于 ReentrantLock，而 ReentrantLock 就是可重入的互斥锁。
对于 ConcurrentHashMap 的添加，删除操作，在操作开始前，线程都会获取 Segment 的互斥锁；操作完毕之后，才会释放。而对于读取操作，它是通过 volatile 去实现的，HashEntry 数组是 volatile 类型的，而 volatile 能保证“即对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入”，即我们总能读到其它线程写入 HashEntry 之后的值。以上这些方式，就是 ConcurrentHashMap 线程安全的实现原理。

13、Lock 与 synchronized 的区别

ReentrantLock 与 synchronized 有相同的并发性和内存语义。由于 synchronized 是在 JVM 层面实现的，因此系统可以监控锁的释放与否，而 ReentrantLock 使用代码实现的，系统无法自动释放锁，需要在代码中 finally 子句中显式释放锁 lock.unlock();
在并发量比较小的情况下，使用 synchronized 是个不错的选择，但是在并发量比较高的情况下，其性能下降很严重，此时 ReentrantLock 是个不错的方案。

(使用 synchronized 获取锁的线程由于要等待 IO 或者其他原因被阻塞了，但是又没有释放锁，其他线程只能干巴巴的等待，而 Lock 可以只等待一定的时间或者能够响应中断)

---补充---
1）synchronized 是 Java 的关键字，因此是 Java 的内置特性，是基于 JVM 层面实现的。而 Lock 是一个 Java 接口，是基于 JDK 层面实现的，通过这个接口可以实现同步访问；
2）采用 synchronized 方式不需要用户去手动释放锁，当 synchronized 方法或者 synchronized 代码块执行完之后，系统会自动让线程释放对锁的占用；而 Lock 则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致死锁现象。

14、什么是死锁

死锁就是两个或两个以上的线程被无限的阻塞，线程之间相互等待所需资源。这种情况可能发生在当两个线程尝试获取其它资源的锁，而每个线程又陷入无限等待其它资源锁的释放，除非一个用户进程被终止。就 JavaAPI 而言，线程死锁可能发生在以下情况。
*当两个线程相互调用 **Thread.join ()**
*当两个线程使用嵌套的同步块，一个线程占用了另外一个线程必需的锁，互相等待时被阻塞就有可能出现死锁。

15、什么是线程饿死，什么是活锁？

线程饿死和活锁虽然不像是死锁一样的常见问题，但是对于并发编程的设计者来说就像一次邂逅一样。当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI 中线程活锁可能发生在以下情形：
*当所有线程在程序中执行 Object.wait (0)，参数为 0 的 wait 方法。程序将发生活锁直到在相应的对象上有线程调用 Object.notify ()或者 Object.notifyAll ()。
*当所有线程卡在无限循环中。

16、Java 中的队列都有哪些，有什么区别。

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列。

Queue 接口与 List、Set 同一级别，都是继承了 Collection 接口。LinkedList 实现了 Queue 接口。我们平时使用的一些常见队列都是非阻塞队列，比如 PriorityQueue、LinkedList(LinkedList 是双向链表，它实现了 Dequeue 接口)

ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、DelayQueue

17、线程和进程的区别

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。

18、Runnable 接口和 Callable 接口的区别

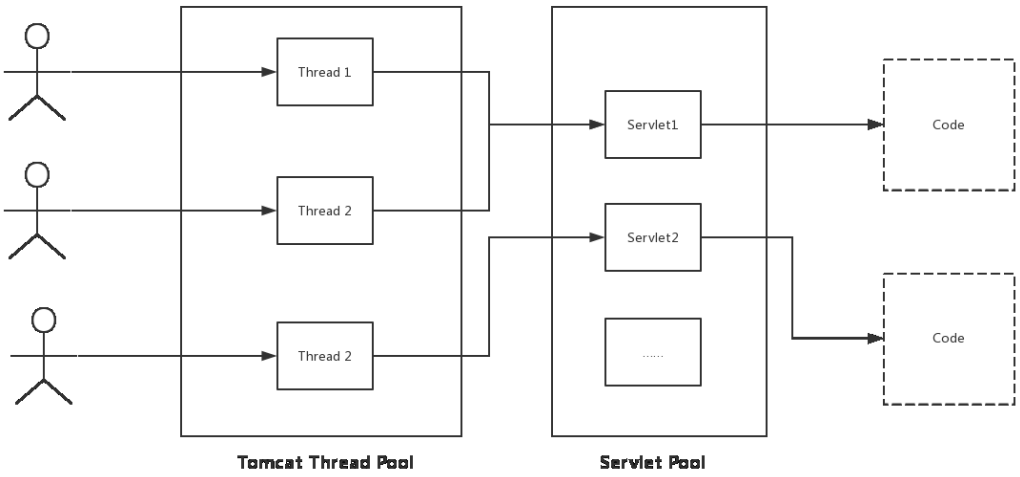
1、Callable 要实现 call 方法，Runnable 要实现 run 方法
2、call 方法可以返回值，run 方法不能
3、call 方法可以抛出 checked exception，run 方法不能
4、Runnable 接口在 jdk1.1 就有了，Callable 在 Jdk1.5 才有

19、什么是 ThreadLocal

待续

20、Servlet 是线程安全的吗

Servlet 不是线程安全的
当 Tomcat 接收到 Client 的 HTTP 请求时，Tomcat 从线程池中取出一个线程，之后找到该请求对应的 Servlet 对象并进行初始化，之后调用 service()方法。要注意的是每一个 Servlet 对象在 Tomcat 容器中只有一个实例对象，即是单例模式。如果多个 HTTP 请求请求的是同一个 Servlet，那么这两个 HTTP 请求对应的线程将并发调用 Servlet 的 service()方法。



上图中的 Thread1 和 Thread2 调用了同一个 Servlet1，所以**此时如果 Servlet1 中定义了实例变量或静态变量，那么可能会发生线程安全问题**（因为所有的线程都可能使用这些变量）。

21、什么是原子操作

原子（atom）本意是“不能被进一步分割的最小粒子”，而原子操作（atomic operation）意为"不可被中断的一个或一系列操作" 。

处理器如何实现原子操作：

32 位 IA-32 处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

JAVA 如何实现原子操作：

在 java 中可以通过锁和循环 CAS 的方式来实现原子操作。

22、 什么是竞争条件？你怎样发现和解决竞争？

在 Java 多线程中，当两个或以上的线程对同一个数据进行操作的时候，可能会产生“竞争条件”的现象。这种现象产生的根本原因是因为多个线程在对同一个数据进行操作，此时对该数据的操作是非“原子化”的，可能前一个线程对数据的操作还没有结束，后一个线程又开始对同样的数据开始进行操作，这就可能会造成数据结果的变化未知。

解决：

线程锁(synchroniezd, Lock)

线程同步

23、数据竞争

数据竞争是指，如果在访问共享的非 final 类型的域时没有采用同步来进行协同，那么就会出现数据竞争。当一个线程写入一个变量而另一个线程接下来读取这个变量，或者读取一个之前由另一个线程写入的变量时，并且在这两个线程之间没有使用同步，那么就可能出现数据竞争。在 Java 内存模型中，如果在代码中存在数据竞争，那么这段代码就没有确定的语义。

24、CyclicBarrier 和 CountDownLatch 的区别

1、CountDownLatch: 一个线程(或者多个)，等待另外 N 个线程完成某个事情之后才能执行。

2、CyclicBarrier: N 个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

3、CountDownLatch 的计数器只能使用一次。而 CyclicBarrier 的计数器可以使用 reset() 方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

4、CountDownLatch：减计数方式，CyclicBarrier：加计数方式

这两个的区别是 CyclicBarrier 可以重复使用已经通过的障碍，而 CountdownLatch 不能重复使用。

25、什么是上下文切换

即使是单核 CPU 也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片来实现这个机制。时间片是 CPU 分配给各个线程的时间，因为时间片非常短，所以 CPU 通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。

CPU 通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，从任务保存到再加载的过程就是一次上下文切换。

JVM 虚拟机

1、Java 内存区域划分

程序计数器：是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。
Java 虚拟机栈：线程私有的，生命周期与线程相同，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。
本地方法栈：和虚拟机栈类似，虚拟机栈执行 Java 方法，而本地栈执行 Native 方法
Java 堆：线程共享，在虚拟机启动时创建，存放对象实例，几乎所有的对象实例都在这里分配内存，是垃圾收集器管理的主要区域。
方法区：线程共享，用于存储已被虚拟机加载的类信息、常量、静态变量等
运行时常量池：方法区的一部分
直接内存

2、heap 和 stack 有什么区别

Java 的内存分为两类，一类是栈内存，一类是堆内存。栈内存是指程序进入一个方法时，会为这个方法单独分配一块私属存储空间，用于存储这个方法内部的存储变量，当这个方法结束时，分配给这个方法的栈会释放，这个栈中的变量也将随之释放。
堆是与栈作用不同的内存，一般用于存放不放在当前方法栈中的那些数据，例如，使用 new 创建的对象都放在堆里，所以，它不会随方法的结束而消失。方法中的局部变量使用 final 修饰后，放在堆中而不是栈中。

3、JVM 底层是如何实现 synchronized 的

synchronized 在 JVM 里的实现原理：**JVM 基于进入和退出 Monitor 对象来实现方法同步和代**

码块同步，但两者的实现细节不一样。代码块同步是使用 monitorenter 和 monitorexit 指令实现的，而方法同步是使用另外一种方式实现的，JVM 规范里并没有详细说明。但，方法的同步同样可以使用这两个指令来实现。

monitorenter 指令是在编译后插入到同步代码块的开始位置，而 monitorexit 是插入到方法结束处和异常处，JVM 要保证每个 monitorenter 必须有对应的 monitorexit 与之配对。任何对象都有一个 monitor 与之关联，当且一个 monitor 被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

4、volatile 的底层实现原理

如果把加入 volatile 关键字的代码和未加入 volatile 关键字的代码都生成汇编代码，会发现加入 volatile 关键字的代码会多出一个 lock 前缀指令。
lock 前缀指令实际相当于一个内存屏障，内存屏障提供了以下功能：
1、重排序时不能把后面的指令重排序到内存屏障之前的位置
2、使得本 CPU 的 Cache 写入内存
3、写入动作也会引起别的 CPU 或者别的内核无效化其 Cache，相当于让新写入的值对别的线程可见。

5、GC 是什么？为什么要有 GC？

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

6、如何判断一个对象是否存活

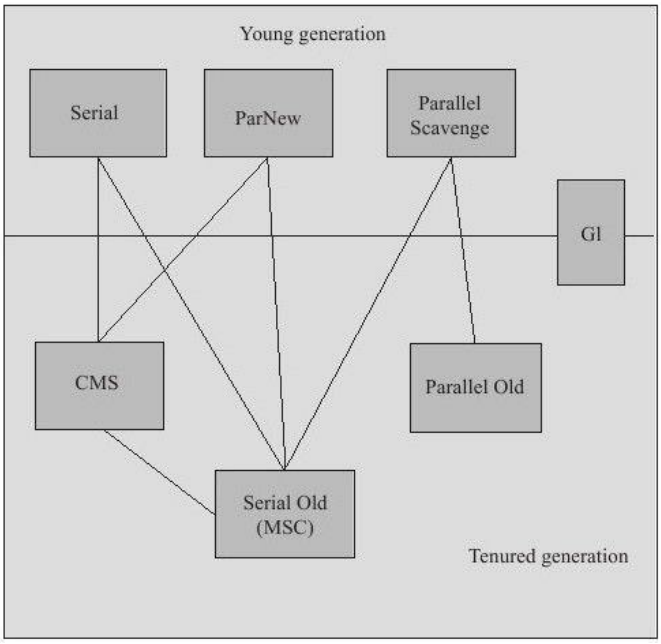
判断一个对象是否存活有两种方法：
1、引用计数算法
2、可达性分析算法

7、Java 中垃圾收集算法都有哪些

1、标记-清除算法(Mark-Sweep)
2、复制算法(Copring)
3、标记-整理算法(Mark-Compact)
4、分代收集算法(Generational Collection)

8、常见的垃圾收集器都有哪些

1、Serial 收集器
2、ParNew 收集器
3、Parallel Scavenge 收集器
4、Serial Old 收集器
5、Parallel Old 收集器
6、CMS 收集器
7、G1 收集器



9、垃圾回收机制的基本原理是什么

虚拟机的垃圾收集主要采用“分代收集”算法，一般把 Java 堆分为新生代和老年代。

新生代的内存空间划分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor，当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间

对象的内存分配，往大方向讲，就是在堆上分配，对象主要分配在新生代的 Eden 区上，少数情况下也可能会直接分配在老年代中。大多数情况下，对象在新生代 Eden 区中分配，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 MinorGC。

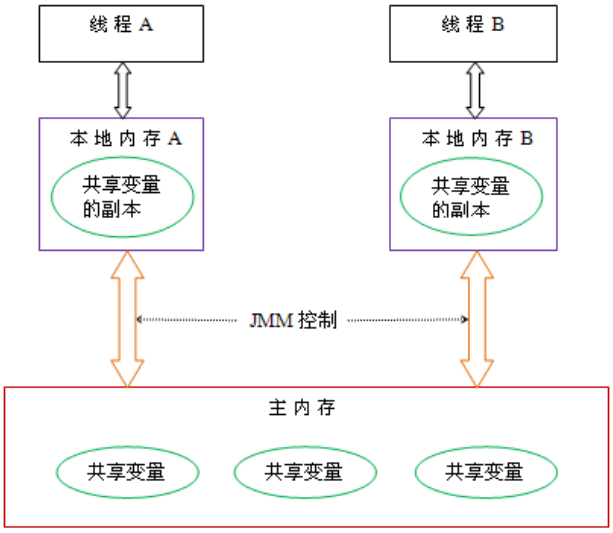
大对象和长期存活的对象直接进入老年代，新生代主要采用“复制算法”来回收内存。

参考：理解 Java 垃圾回收机制

10、Java 内存模型

Java 内存区分为堆内存和栈内存，所有实例域、静态域和数组元素存储在堆内存中，堆内存在线程之间共享，局部变量，方法定义参数和异常处理器参数不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

Java 线程之间的通信由 java 内存模型控制，JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系，线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，本地内存中存储了该线程以读/写共享变量的副本。



从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：
1、首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。
2、然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

参考：深入理解 Java 内存模型

11、Java 类加载机制

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

12、Java 类加载过程

Java 虚拟机中类加载的全过程，包括加载、验证、准备、解析和初始化这 5 个阶段

参考：Java 虚拟机类加载机制

13、类加载器双亲委派模型机制

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去加载。

14、什么是类加载器，类加载器有哪些？

通过一个类的全限定名来获取描述类的二进制字节流，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为"类加载器"
主要有以下四种类加载器：
1、启动类加载器(Bootstrap ClassLoader)
2、扩展类加载器
3、系统类加载器
4、用户自定义类加载器

MySQL

1、MySQL 事务隔离级别

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	X	V	V
可重复读（Repeatable read）	X	X	V
可串行化（Serializable）	X	X	X

2、MySQL 锁机制

MySQL 有三种锁的级别：页级、表级、行级
MyISAM 和 MEMORY 存储引擎采用的表级锁；InnoDB 存储引擎既支持行级锁，也支持表级锁，但默认情况下是采用行级锁。

3、MySQL 存储引擎

MyISAM
它不支持事务，也不支持外键，访问速度快，对事务完整性没有要求或者以 SEIECT INSERT 为主的应用基本都可以使用这个引擎来创建表。

InnoDB
InnoDB 存储引擎提供了具有提交，回滚和崩溃恢复能力的事务安全。但是对比 MyISAM 的存储引擎 InnoDB 写的处理效率差一些并且会占用更多的磁盘空间以保留数据和索引

MEMORY
如名字所指明的，MEMORY 表存储在内存中，且默认使用哈希索引。这使得它们非常快，并且对创建临时表非常有用。可是，当服务器关闭之时，所有存储在 MEMORY 表里的数据丢失。

4、MySQL Limit 分页优化

索引、子查询、表连接

5、共享锁与排他锁

http://blog.itmyhome.com/2017/06/mysql-share-lock

6、乐观锁与悲观锁

http://blog.itmyhome.com/2017/06/mysql-optimistic-lock-and-pessimistic-lock

算法

1、冒泡排序

```java
public class Test {
    public static void main(String[] args) {
        int temp[] = {13,52,3,8,5,16,41,29};
        //执行temp.length次
        for (int i = 0; i < temp.length; i++) {
            for (int j = 0; j < temp.length-i-1; j++) {
                if(temp[j]>temp[j+1]){ //前一个数和后一个数比较
                    int a = temp[j];
                    temp[j] = temp[j+1];
                    temp[j+1] = a;
                }
            }
        }
        for (int i = 0; i < temp.length; i++) {
            System.out.print(temp[i]+" ");
        }
    }
}
```

2、选择排序

原理：首先在未排序列中找到最小元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小元素，然后放到已排序序列的末尾。依次类推，直到所有元素均排序完毕。

3、快速排序

所谓的快速排序的思想就是，首先把数组的第一个数拿出来作为一个 key,在前后分别设置一个 i,j 作为标识，然后拿这个数组从后面往前遍历， 及 j- -,直到找到第一个小于这个 key 的那个数然后交换这两个值，交换完成后，我们拿着这个 key 要从 i 往后遍历了，及 i++ 一直循环到 i=j 结束，
当结束后，我们会发现大于这个 key 的值都会跑到这个 key 的后面，小于这个 key 的值就会跑到这个值的前面,然后我们对这个分段的数组再进行
递归调用就可以完成这个数组的排序。

Spring

1、事务传播属性和隔离级别

Spring 在 TransactionDefinition 接口中规定了 7 种类型的事务传播行为， 它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播：

事务传播行为类型	说明
PROPAGATION_REQUIRED	如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

在 Spring 中定义了四种不同的事务隔离级别：

事务隔离级别	说明
read_uncommitted	读未提交，一个事务可以操作另外一个未提交的事务，不能避免脏读，不可重复读，幻读，隔离级别最低，并发性能最高
read_committed	读已提交，一个事务不可以操作另外一个未提交的事务， 能防止脏读，不能避免不可重复读，幻读。
repeatable_read	能够避免脏读，不可重复读，不能避免幻读
serializable	隔离级别最高，消耗资源最低，代价最高，能够防止脏读， 不可重复读，幻读

2、Spring 注解@Resource 和@Autowired 区别对比

1、@Autowired 与@Resource 都可以用来装配 bean，都可以写在字段上或写在 setter 方法上
2、@Autowired 默认按类型装配(这个注解是属于 Spring 的)
@Resource(这个注解属于 J2EE)的，默认按照名称进行装配。

3、Spring 常用注解

@Component
是所有收 Spring 管理组件的通用形式，@Component 注解可以放在类的头上，不推荐使用。

@Controller
对应表现层的 Bean，也就是 Action

@Service
对应的是业务层 Bean

@Repository
对应数据访问层 Bean

@Resource 和@Autowired
都是做 bean 的注入时使用

@RequestMapping
是一个用来处理**请求地址映射**的注解，可用于类或方法上

4、BeanFactory 和 ApplicationContext 有什么区别?

BeanFactory 是 IoC 容器的核心接口，Spring 使用 BeanFactory 来实例化、配置和管理 Bean。它定义了 IoC 的基本功能，主要定义了 getBean 方法。getBean 方法是 IoC 容器获取 bean 对象和引发依赖注入的起点。方法的功能是返回特定的名称的 Bean

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。BeanFactory 接口提供了配置框架及基本功能，但是无法支持 spring 的 aop 功能和 web 应用。而 ApplicationContext 接口作为 BeanFactory 的派生，因为提供 BeanFactory 所有的功能。而且 ApplicationContext 还在功能上做了扩展，相较于 BeanFactorty，ApplicationContext 还提供了以下的功能：

（1）MessageSource, 提供国际化的消息访问
（2）资源访问，如 URL 和文件
（3）事件传播特性，即支持 aop 特性
（4）载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层

ApplicationContext：是 IOC 容器另一个重要接口， 它继承了 BeanFactory 的基本功能， 同时也继承了容器的高级功能，如：MessageSource（国际化资源接口）、ResourceLoader（资源加载接口）、ApplicationEventPublisher（应用事件发布接口）等。

链接：https://www.cnblogs.com/xiaoxi/p/5846416.html

SpringMVC

1、SpringMVC 工作原理

1、客户端请求提交到 DispatcherServlet
2、由 DispatcherServlet 控制器查询一个或多个 HandlerMapping，找到处理请求的 Controller
3、DispatcherServlet 将请求提交到 Controller
4、Controller 调用业务逻辑处理后，返回 ModelAndView
5、DispatcherServlet 查询一个或多个 ViewResoler 视图解析器，找到 ModelAndView 指定的视图
6、视图负责将结果显示到客户端 DispatcherServlet 是整个 Spring MVC 的核心。它负责接收 HTTP 请求组织协调 Spring MVC 的各个组成部分。

其主要工作有以下三项：
1. 截获符合特定格式的 URL 请求。
2. 初始化 DispatcherServlet 上下文对应的 WebApplicationContext，并将其与业务层、持久化层的 WebApplicationContext 建立关联。
3. 初始化 Spring MVC 的各个组成组件，并装配到 DispatcherServlet 中。

持续更新
2018-2-2