# Java知识题

## 一、Java 基本API

### 1、String,StringBuffer, StringBuilder 的区别是什么？String为什么是不可变的？

答：（1）String是字符串常量，StringBuffer和StringBuilder都是字符串变量。后两者的字符内容可变，而前者创建后内容不可变。

（2）String不可变是因为在JDK中String类被声明为一个final类。

（3）StringBuffer是线程安全的，而StringBuilder是非线程安全的。

补充：String类是final的， 而且使用private final char value[];保存字符串内容

### 2、Vector, ArrayList, LinkedList的区别是什么？

答：（1）Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。

（2）List中的元素有序、允许有重复的元素，Set中的元素无序、不允许有重复元素。

（3）Vector线程同步，ArrayList、LinkedList线程不同步。

（4）LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。

（5）ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。（Vector可以传入增长大小，ArrayList没有增长大小）

补充：Vector是JDK1.0的api，所有方法都是synchronize方法，效率非常低。

Stack也是JDK1.0的API，继承Vector。

Vector和Stack被弃用

高并发下，使用java.util.concurrent.CopyOnWriteArrayList等替代Vector

使用java.util.concurrent.ConcurrentLinkedDeque或者java.util.concurrent.ConcurrentLinkedQueue替代Stack

### 3、HashTable, HashMap，TreeMap区别？

答：（1）HashTable线程同步，HashMap、TreeMap非线程同步。

（2）HashTable不允许<键,值>有空值，HashMap允许<键,值>有空值。

（3）HashTable使用Enumeration，继承Dictionary类，实现Map接口；HashMap使用Iterator，实现Map接口；TreeMap实现SortedMap接口

（4）HashTable中hash数组的默认大小是11，增加方式的old\*2+1，HashMap中hash数组的默认大小是16，增长方式一定是2的指数倍。

（5）TreeMap能够把它保存的记录根据键排序，默认是按升序排序。

（6）TreeMap采用红黑树（平衡二叉树）

补充：HashTable是JDK1.0的api，所有方法都是synchronize方法，效率非常低。可以用ConcurrentHashMap替代。但是并不能完全被取代，两者的迭代器的一致性不同的，hash table的迭代器是强一致性的，而concurrenthashmap是弱一致的。 ConcurrentHashMap的get，clear，iterator 都是弱一致性的。 Doug Lea 也将这个判断留给用户自己决定是否使用ConcurrentHashMap。

FastTreeMap是apache commons的api，有fast、slow模式，都是线程安全的

Fast模式下，写是复制对象，用synchronize， 读是从旧对象中读取

CopyOnWriteArrayList，set是复制对象，用ReentrantLock， 读是从旧对象中读取

### 4、Enumeration、Iterator、Iteratable区别？

答：（1）Iterator替代Enumeration，

（2）Iterator方法：boolean hasNext()，Object next()，void remove()

Enumeration方法：boolean hashMoreElements()，Object nextElement()

（2）Enumeration中没有删除方法，只有遍历。

（3）Enumeration是先进后出，而Iterator是先进先出。

（4）Iterator是迭代器，有boolean hasNext()，Object next()，void remove()

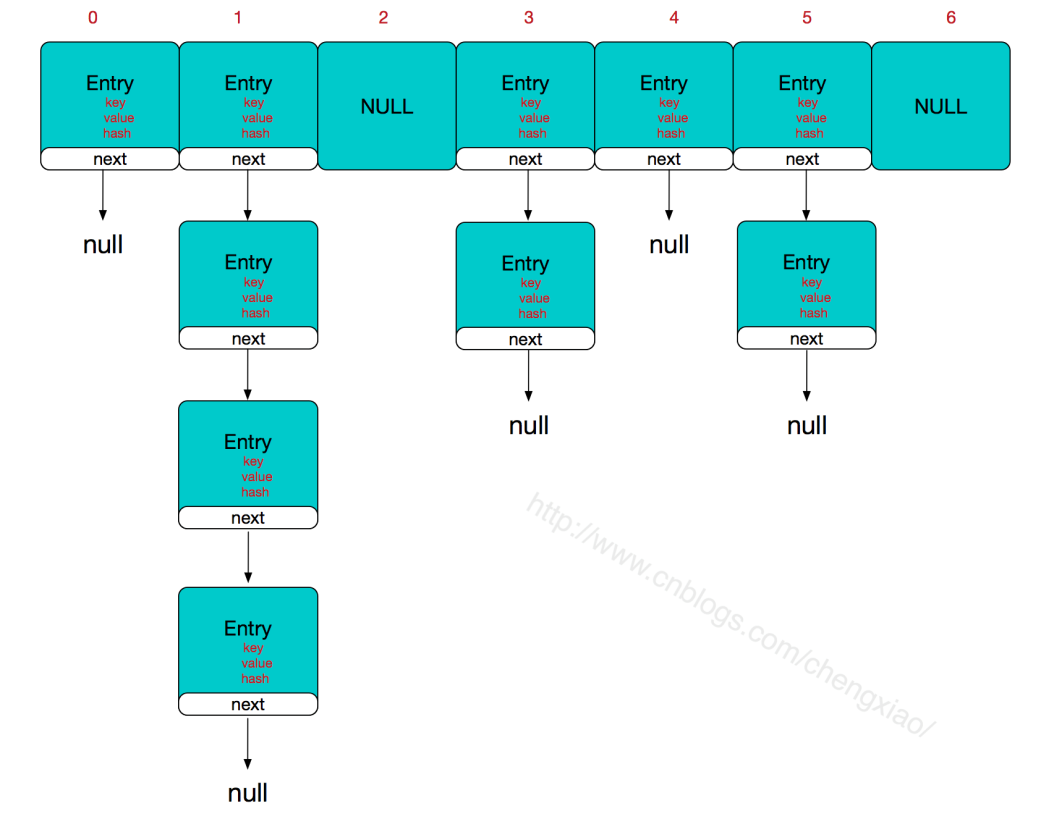
Iteratable是接口，有Iterator<T> iterator();

比如ArrayList实现Iteratable接口，iterator()方法返回的是Iterator迭代器实现

补充： 使用了Iteratable接口，可以使用foreach方法

### 5、JDK7.0下HashMap的内部实现机制，hash是怎么实现的，什么时候rehash

答：（1）HashMap内部是Entry数组（Hash表）， Entry是一个单向链表。HashMap的内部实现结合了链表和数组的优势



简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为O(1)，因为最新的Entry会插入链表头部，急需要简单改变引用链即可，而对于查找操作来讲，此时就需要遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

（2）源码分析https://www.cnblogs.com/chengxiao/p/6059914.html

默认值initialCapacity默认为16，loadFactory默认为0.75

public V put(K key, V value) {

//如果table数组为空数组{}，进行数组填充（为table分配实际内存空间），入参为threshold，此时threshold为initialCapacity 默认是1<<4(24=16)

if (table == EMPTY\_TABLE) {

inflateTable(threshold);

}

//如果key为null，存储位置为table[0]或table[0]的冲突链上

if (key == null)

return putForNullKey(value);

int hash = hash(key);//对key的hashcode进一步计算，确保散列均匀

int i = indexFor(hash, table.length);//获取在table中的实际位置

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

//如果该对应数据已存在，执行覆盖操作。用新value替换旧value，并返回旧value

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;//保证并发访问时，若HashMap内部结构发生变化，快速响应失败

addEntry(hash, key, value, i);//新增一个entry

return null;

}

在扩容的过程中需要进行ReHash操作，而这是非常耗时的，在实际中应该尽量避免。

（3）当size达到阈值时，需要进行数组扩容rehash， 数组长度一定是2的次幂， 因为是：为了hash的平均分布。如果不是2的次幂，会导致部分位置会永远分配不到。

（4）hash算法：直接定址法、求余法、数字分析法、平方取中法、折叠法、随机数法等。为了减少冲突



（5）如果两个键的hashcode相同，如何获取值对象?

当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

（6）重新调整HashMap大小存在什么问题吗？

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。

补充：关于HashMap其他几个问题

1. 为什么String, Interger这样的wrapper类适合作为键？

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。

1. 可以使用自定义的对象作为键吗？

使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之后将不会再改变了。定义为final类

1. 可以使用CocurrentHashMap来代替Hashtable吗？

Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性

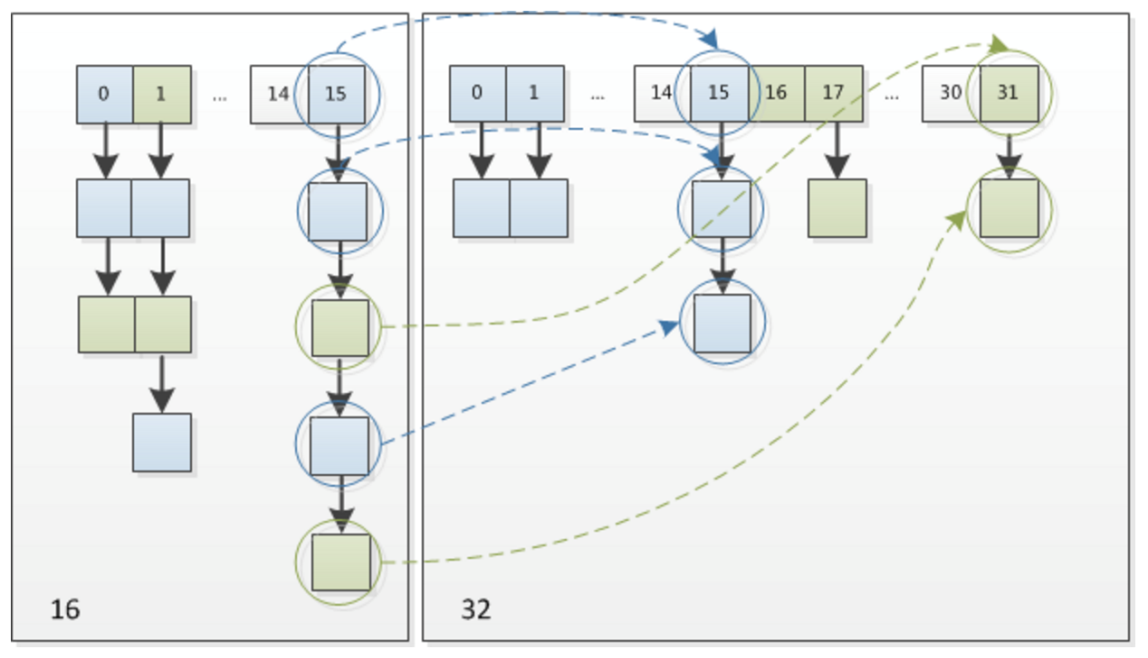
### 6、JDK8.0下HashMap的内部实现机制，hash是怎么实现的，什么时候rehash

答：（1）Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。JDK8优化高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)

（2）链表长度大于8，转换为红黑树

（3）rehash，不需要重新计算hash值，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”

JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。



（4）JDK1.8引入红黑树大程度优化了HashMap的性能。

### 7、JDK7.0下ConcurrentHashMap的内部实现机制，hash是怎么实现的，什么时候rehash

答：

### 8、JDK8.0下ConcurrentHashMap的内部实现机制，hash是怎么实现的，什么时候rehash

答：

### 9、foreach原理，如何边遍历边删除？

答：（1）集合类都实现了Iterable接口，该接口中定义了Iterator迭代器

（2）边遍历边删除，需要用Iterator

Iterator<String> iter = list.iterator();

while (iter.hasNext()) {

String v = iter.next();

if ("4".equals(v)) {

iter.remove();

}

}

### 10、java集合类

答：（1）List：有序、可重复

（2）Set：无序、不能重复

（3）Map（键值对、键唯一、值不唯一）

### 11、Java的队列都有哪些，有什么区别？

答：（1）并发队列：ConcurrentLinkedQueue

ConcurrentLinkedQueue是Queue的一个安全实现，Queue中元素按FIFO原则进行排序，采用CAS操作，来保证元素的一致性。适用于高并发场景下的队列

（2）阻塞队列：BlockingQueue

BlockingQueue的主要功能并不在于提高并发时的队列性能，简化多线程之间的数据共享。BlockingQueue的典型应用在于 生产者-消费者。才有ReentrantLock的Condition操作。

BlockingQueue提供了线程安全的队列访问方式：当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。



抛异常：如果试图的操作无法立即执行，抛一个异常。

特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是true / false)。

ArrayBlockingQueue：ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

DelayQueue：DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口。

LinkedBlockingQueue：LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。

PriorityBlockingQueue：PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。

SynchronousQueue：SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

### 12、Class.forName和ClassLoader.loadClass的区别

答：（1）class.forName()前者除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块和静态变量。

Class.forName(name, initialize, loader)带参函数也可控制是否加载static块。并且只有调用了newInstance()方法采用调用构造函数，创建类的对象。

（2）classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容和和静态变量,只有在newInstance才会去执行static块和静态变量

|  |
| --- |
| public class Line {  static{  System.out.println("执行静态块");  }  public static String aa = getString();  private static String getString() {  System.out.println("执行静态方法");  return "静态方法";  }  public Line() {  System.out.println("执行构造函数");  }  } |

|  |
| --- |
| public class ClassloaderAndFornameTest {  /\*\*  \* Class.forName默认会执行static块和静态变量  \*/  @Test  public void testClassForName1() {  try {  Class.forName(Line.class.getName());  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  /\*\*  \* Class.forName(name, initialize, loader)带参函数也可控制是否加载static块和静态变量。  \*/  @Test  public void testClassForName2() {  try {  Class.forName(Line.class.getName(), false, ClassLoader.getSystemClassLoader());  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  /\*\*  \* ClassLoader不会执行static块和和静态变量  \*/  @Test  public void testClassLoader() {  try {  ClassloaderAndFornameTest.class.getClassLoader().loadClass(Line.class.getName());  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  } |

## 二、Java并发API

### 1、什么是CAS

答：（1）CAS是一种系统原语（所谓原语属于操作系统用语范畴。原语由若干条指令组成的，用于完成一定功能的一个过程。primitive or atomic action 是由若干个机器指令构成的完成某种特定功能的一段程序，具有不可分割性·即原语的执行必须是连续的，在执行过程中不允许被中断）。

（2）CAS是Compare And Set的缩写。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

**直接使用字段偏移量取内存值**。

compareAndSwapInt就是借助C来调用CPU底层指令实现的。

private volatile int value;

public final int get() {

return value;

}

//实现++1

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

（3）CAS缺点：

ABA问题：因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。

循环时间长开销大：旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

只能保证一个共享变量的原子操作：当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

补充：AtomicReference是通过"volatile"和"Unsafe提供的CAS函数实现"原子操作。

(01) value是volatile类型。这保证了：当某线程修改value的值时，其他线程看到的value值都是最新的value值，即修改之后的volatile的值。**对象中的变量也需要volatile类型！**

(02) 通过CAS设置value。这保证了：当某线程池通过CAS函数(如compareAndSet函数)设置value时，它的操作是原子的，即线程在操作value时不会被中断。

for (int i = 0; i < 100; i++) {

new Thread() {

public void run() {

for (int j = 0; j < 10; j++) {

Person p = personRef.get();

p.setId(p.getId()+1);

if (!personRef.compareAndSet(null, p)) {

System.out.println(p.getId());

}

}

}

}.start();

}

### 2、自旋锁的原理

答：（1）获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。利用CAS原理

（2）自旋锁比较适用于锁使用者保持锁时间比较短的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

（3）缺点：

递归死锁，递归程序决不能在持有自旋锁时调用它自己，也决不能在递归调用时试图获得相同的自旋锁。

过多占用cpu资源，一般自旋锁实现会有一个参数限定最多持续尝试次数. 超出后, 自旋锁放弃当前time slice. 等下一次机会

### 3、UNSAFE的原理

答：Unsafe类提供了硬件级别的原子操作，主要提供了以下功能：

1. 通过Unsafe类可以分配内存，可以释放内存；提供的3个本地方法allocateMemory、reallocateMemory、freeMemory分别用于分配内存，扩充内存和释放内存
2. 可以定位对象某字段的内存位置，也可以修改对象的字段值，即使它是私有的；
3. 挂起与恢复：Unsafe.park()
4. CAS操作

补充：unsafe涉及到的偏移量，其实就是类成员变量在内存中的偏移量

stateOffset = unsafe.objectFieldOffset

(AbstractQueuedSynchronizer.class.getDeclaredField("state"));

### 4、ReectrantLock和AQS实现原理

### 5、ReentrantLock的Condition

答：Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set （wait-set）。

其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。

在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()，传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

Condition的强大之处在于它可以为多个线程间建立不同的Condition

|  |
| --- |
| public class MyBlockingQueue {  private final Lock lock = new ReentrantLock();  private Condition addCondition = lock.newCondition();  private Condition takeCondition = lock.newCondition();  private Object[] data;  private int count = 0;  private int position = -1;  public MyBlockingQueue(int size) {  this.data = new Object[size];  }  public void add(Object obj) {  lock.lock();  try {  while (count == this.data.length) {  try {  addCondition.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  this.data[++position] = obj;  count++;  takeCondition.signalAll();  } finally {  lock.unlock();  }  }  public Object take() {  lock.lock();  try {  while (count == 0) {  try {  takeCondition.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  Object obj = this.data[position--];  count--;  addCondition.signalAll();  return obj;  } finally {  lock.unlock();  }  }  } |

### 6、CountDownLatch的工作原理

答：CountDownLatch采用AQS（AbstractQueuedSynchronizer）队列实现，先初始化Count，再countDown，当计数器值到达0时，表示所有任务都执行完了。

|  |
| --- |
| /\*\*  \* 用CountDownLatch实现多个任务并发计算，并汇总结果  \* @author changtan.sun  \*  \*/  public class MyComputeTask {  static ConcurrentLinkedQueue<Long> sums = new ConcurrentLinkedQueue<Long>();  public MyComputeTask() {  }  public long compute(int number, int part) throws Exception {  number = number + 1;  int parts = (number + part - 1) / part;  CountDownLatch latch = new CountDownLatch(parts + 1);  for (int i = 0; i < parts; i++) {  long min = i \* part;  long max = (i + 1) \* part < number ? (i + 1) \* part : number;  new Thread(new MyTask(min, max, latch)).start();  }  latch.countDown();  latch.await();  int sum = 0;  for (Long s : sums) {  sum += s;  }  return sum;  }  private static class MyTask implements Runnable {  private long min;  private long max;  private CountDownLatch latch;  public MyTask(long min, long max, CountDownLatch latch) {  this.min = min;  this.max = max;  this.latch = latch;  }  @Override  public void run() {  long sum = 0;  for (long i = min; i < max; i++) {  sum += i;  }  sums.add(sum);  latch.countDown();  }  }  } |

7、

## 三、JVM

### 1、Java线程通信和同步

答：（1）并发编程涉及线程之间的通信和同步

（2）线程之间的通信：线程之间以何种机制来交换信息， 有两种**共享内存**和**消息传递**

**共享内存**：线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

**消息传递**：线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

（3）同步：指程序用于控制不同线程之间操作发生相对顺序的机制。

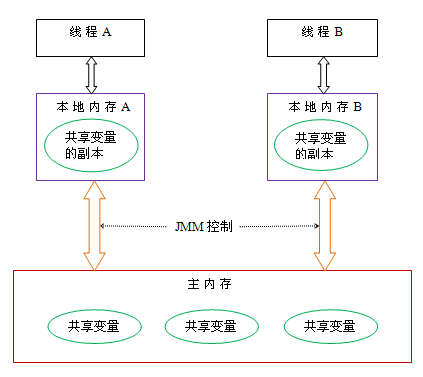
共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行。

（4）Java的并发采用的是共享内存模型

### 2、Java内存模型（并发）

答：（1）线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。



（2）支撑Java内存模型的基础原理

1）指令重排序：执行程序时，为了提高性能，编译器和处理器会对指令做重排序

2）数据依赖性：如果两个操作访问同一个变量，其中一个为写操作，此时这两个操作之间存在数据依赖性，即不会重排序。

3）as-if-serial：不管怎么重排序，单线程下的执行结果不能被改变，编译器、runtime和处理器都必须遵守as-if-serial语义。

4）内存屏障（Memory Barrier ）

通过内存屏障可以禁止特定类型处理器的重排序、强制刷各种CPU cache

5）happens-before

程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。

监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。

volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

（3）原子性问题，可见性问题，有序性问题

原子性问题：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

可见性问题：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性问题：程序执行的顺序按照代码的先后顺序执行

### 3、wait和notify

答：（1）wait()方法使得当前线程必须要等待，等到另外一个线程调用notify()或者notifyAll()方法。

当前的线程必须拥有当前对象的monitor，也即lock，就是锁。

线程调用wait()方法，释放它对锁的拥有权，然后等待另外的线程来通知它（通知的方式是notify()或者notifyAll()方法），这样它才能重新获得锁的拥有权和恢复执行。

要确保调用wait()方法的时候拥有锁，即，wait()方法的调用必须放在synchronized方法或synchronized块中。

（2）notify()方法会唤醒一个等待当前对象的锁的线程。

notify方法调用必须放在synchronized方法或synchronized块中。

### 4、用 wait-notify 写一段代码来解决生产者-消费者问题

### 5、什么是线程局部变量

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java 提供 ThreadLocal 类来支持线程局部变量，是一种实现线程安全的方式。

### 6、用 Java 写一个线程安全的单例模式（Singleton）

|  |
| --- |
| public class Test4 {  private static volatile Test4 instance; //懒汉模式  private Test4() {  }  public static Test4 getInstance() {  if (instance == null) { //双重否定加锁  synchronized (Test4.class) {  if (instance == null) {  instance = new Test4();  }  }  }  return instance;  }  }  这里为什么要使用volatile修饰instance？主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在JVM中这句话大概做了下面3件事情:  （1）给instance分配内存  （2）调用Singleton的构造函数来初始化成员变量  （3）将instance对象指向分配的内存空间（执行完这步instance就为非null了）。  但是在JVM的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在3执行完毕、2未执行之前，被线程二抢占了，这时instance已经是非null了（但却没有初始化），所以线程二会直接返回instance，然后使用，然后顺理成章地报错。  public class Test5 {  private static final Test5 instance = new Test5(); //  private Test5() {  }  public Test5 getInstance() {  return instance;  }  } |

### 7、Java内存逃逸

答：（1）当变量（或者对象）在方法中分配后，其指针被返回或者被全局引用（这样就会被其他过程或者线程所引用），这种现象称作指针（或者引用）的逃逸(Escape)。

（2）指针逃逸场景：全局变量赋值，方法返回值，实例引用传递

|  |
| --- |
| class A {  public static B b;    public void globalVariablePointerEscape() { // 给全局变量赋值，发生逃逸  b = new B();  }    public B methodPointerEscape() { // 方法返回值，发生逃逸  return new B();  }    public void instancePassPointerEscape() {  methodPointerEscape().printClassName(this); // 实例引用传递，发生逃逸  }  }    class B {  public void printClassName(A a) {  System.out.println(a.class.getName());  }  } |

（3）this逃逸

this逃逸是**指在构造函数返回之前其他线程就持有该对象的引用. 调用尚未构造完全的对象的方法可能引发令人疑惑的错误**, 因此应该避免this逃逸的发生.

|  |
| --- |
| public class ThisEscape {  public ThisEscape() {  new Thread(new EscapeRunnable()).start();  // ...  }    private class EscapeRunnable implements Runnable {  @Override  public void run() {  // 通过ThisEscape.this就可以引用外围类对象, 但是此时外围类对象可能还没有构造完成, 即发生了外围类的this引用的逃逸  }  }  }  解决方案：  public class ThisEscape {  private Thread t;  public ThisEscape() {  t = new Thread(new EscapeRunnable());  // ...  }    public void init() {  t.start();  }    private class EscapeRunnable implements Runnable {  @Override  public void run() {  // 通过ThisEscape.this就可以引用外围类对象, 此时可以保证外围类对象已经构造完成  }  }  } |

### 8、final关键词

答：（1）final的作用：

final关键字提高了性能。JVM和Java应用都会缓存final变量。

final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。

使用final关键字，JVM会对方法、变量及类进行优化。

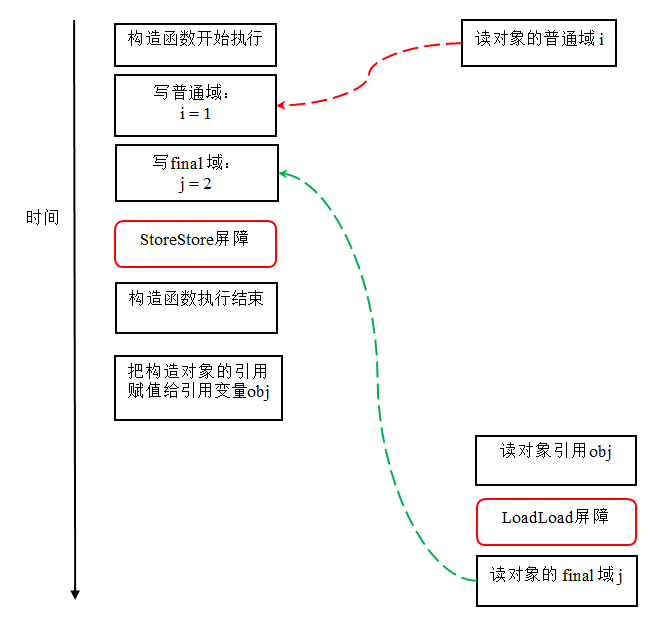
（2）final的内存模型

final域，编译器和处理器要遵守两个重排序规则：

1）在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2）初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

写final域的重排序规则会要求译编器在final域的写之后，构造函数return之前，插入一个StoreStore障屏。读final域的重排序规则要求编译器在读final域的操作前面插入一个LoadLoad屏障。



1. final引用不能从构造函数内“逸出”

### 9、volatile关键字

答：（1）volatile关键字来保证可见性

1）当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

2）禁止进行指令重排序

（2）volatile不能保证原子性

可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

（3）volatile一定程度上保证有序性

|  |
| --- |
| //x、y为非volatile变量  //flag为volatile变量    x = 2; //语句1  y = 0; //语句2  flag = true; //语句3  x = 4; //语句4  y = -1; //语句5 |

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

（4）使用volatile必须具备以下2个条件：

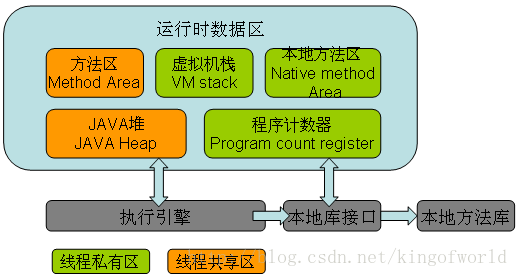
对变量的写操作不依赖于当前值

该变量没有包含在具有其他变量的不变式中

|  |
| --- |
| volatile boolean inited = false;  private volatile static Singleton instance = null;  也就是最简单的形式 |

### 10、JDK1.7的Jvm内存管理和GC算法

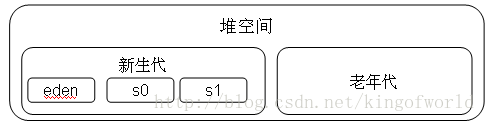
答：（1）内存结构



Java堆：线程共享的，存放所有对象实例和数组。垃圾回收的主要区域。可以分为新生代和老年代(tenured)。

新生代用于存放刚创建的对象以及年轻的对象，如果对象一直没有被回收，生存得足够长，老年对象就会被移入老年代。

新生代又可进一步细分为eden、survivorSpace0(s0,from space)、survivorSpace1(s1,to space)。刚创建的对象都放入eden,s0和s1都至少经过一次GC并幸存。如果幸存对象经过一定时间仍存在，则进入老年代(tenured)。



（2）GC算法：

**标记-清除算法（Mark-Sweep）**：最基础的GC算法，将需要进行回收的对象做标记，之后扫描，有标记的进行回收，这样就产生两个步骤：标记和清除。

**复制算法（Copying）**：将内存分成两块，每次只使用其中一块，垃圾回收时，将标记的对象拷贝到另外一块中，然后完全清除原来使用的那块内存。复制后的空间是连续的。复制算法适用于新生代，因为垃圾对象多于存活对象，复制算法更高效。在新生代串行垃圾回收算法中，将eden中标记存活的对象拷贝未使用的s1中，s0中的年轻对象也进入s1，如果s1空间已满，则进入老年代；这样交替使用s0和s1。这种改进的复制算法，既保证了空间的连续性，有避免了大量的内存空间浪费。

**标记-整理（或叫压缩）算法（Mark-Compact）**：标记后不复制，而是将存活对象压缩到内存的一端，然后清理边界外的所有对象。适合用于老年代的算法（存活对象多于垃圾对象）。

（3）垃圾收集器

串行GC（SerialGC）、并行回收GC（Parallel Scavenge）和并行GC（ParNew）

**串行GC（SerialGC）**：在进行垃圾回收的时候，需要将所有正在执行的线程暂停（Stop The World）

**并行ParNew GC**：加入了多线程机制，提高了效率

**Parallel Scavenge GC**：在整个扫描和复制过程采用多线程的方式来进行，适用于多CPU、对暂停时间要求较短的应用上，是server级别默认采用的GC方式，可用-XX:+UseParallelGC来强制指定，用-XX:ParallelGCThreads=4来指定线程数。



**CMS (Concurrent Mark Sweep)收集器**：解决Serial GC 的停顿问题，以达到最短回收时间，整个收集过程大致分为4个步骤：初始标记(CMS initial mark)、并发标记(CMS concurrenr mark)、重新标记(CMS remark)、并发清除(CMS concurrent sweep)。

优点：并发收集、低停顿，但是CMS还远远达不到完美

缺点：CMS是基于“标记-清除”算法实现的收集器，使用“标记-清除”算法收集后，会产生大量碎片

**G1收集器**：基于标记-整理算法，不会产生内存碎片问题，可以比较精确的控制停顿

**Serial Old**：Serial Old是Serial收集器的老年代版本，它同样使用一个单线程执行收集，使用“标记-整理”算法。主要使用在Client模式下的虚拟机。

**Parallel Old**：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。

### 11、JDK1.8的Jvm内存管理和GC算法

（）

### 12、OOM

答：（1）Java堆溢出：比如从数据库获取大量数据，List、MAP等集合对象是否有使用完后，未清除的问题

内存溢出：如果不存在泄漏，应当检查虚拟机的堆参数（-Xmx 与-Xms）

内存泄漏：通过工具查看泄漏对象到GC Roots 的引用链

测试：-verbose:gc –Xms10M –Xmx10M -XX:+PrintGCDetails

|  |
| --- |
| public static void main(String[] args) {  List<Person> cases = new ArrayList<Person>();  while (true) {  cases.add(new Person());  }  } |

使用jmap -histo pid>a.log 检测内存溢出

（2）Java栈溢出：常见于递归或死循环

（3）常量池溢出：常量

-XX:PermSize=64M -XX:MaxPermSize=128m

（4）方法区溢出：常见于反射生成大量对象实体

（5）直接内存溢出：常见于NIO内存分配

内存不足方面的异常见的是以下3种：

1、java.lang.OutOfMemoryError: Java heap space

2、java.lang.OutOfMemoryError: PermGen space

3、java.lang.StackOverflowError

### 13、jmap -histo pid 输出的[C [B [I [S methodKlass的含义

|  |
| --- |
| num #instances #bytes class name  ----------------------------------------------  1: 1169837 131659368 [C  2: 25945 38337824 [I  3: 31548 29407968 [B  4: 1164546 27949104 java.lang.String  6: 91313 12829072 <constMethodKlass>  7: 12395 12404880 [S  8: 91313 11700288 <methodKlass>  9: 7525 9303112 <constantPoolKlass>  10: 7525 5606808 <instanceKlassKlass>  11: 6043 5028288 <constantPoolCacheKlass>  12: 10048 2007888 [Ljava.lang.Object; |

C is a char[]

[S is a short[]

[I is a int[]

[B is a byte[]

[[I is a int[][]

上面的输出中[C对象占用Heap这么多，往往跟String有关，String其内部使用final char[]数组来保存数据的

constMethodKlass/ methodKlass/ constantPoolKlass/ constantPoolCacheKlass/ instanceKlassKlass/ methodDataKlass

与Classloader相关，常驻与Perm区。

其中最后一行(total行）

分别记录了实例总数、程序占用总内存数，本例显示的程序总占用内存约300M

### 14、VM性能调优监控工具jps、jstack、jmap、jhat、jstat、hprof使用详解

答：（1）jps主要用来输出JVM中运行的进程状态信息

（2）jstack主要用来查看某个Java进程内的线程堆栈信息

（3）jmap用来查看堆内存使用状况，一般结合jhat使用。

## 四、Servlet/JSP

1、GET，POST区别？

答：（1）Get是从服务器端获取数据，Post则是向服务器端发送数据。

（2）在客户端，Get方式通过URL提交数据，在URL地址栏可以看到请求消息，该消息被编码过；Post数据则是放在Html body内提交。

（3）对于Get方式，服务器端用Request.QueryString获取变量的值；对用Post方式，服务器端用Request.Form获取提交的数据值。

（4）Get方式提交的数据最多1024字节，而Post则没有限制。

（5）Get方式提交的参数及参数值会在地址栏显示，不安全，而Post不会，比较安全。

补充：Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE

GET是等幂操作， POST是非等幂操作

HTTP响应格式：

<status line>

<headers>

<blank line>

[<response-body>]

ttp header 消息通常被分为4个部分：general header, request header, response header, entity header

Accept 指定客户端能够接收的内容类型 Accept: text/plain, text/html

Accept-Language 浏览器可接受的语言 Accept-Language: en,zh

Accept-Encoding 指定浏览器可以支持的web服务器返回内容压缩编码类型。 Accept-Encoding: compr

Cookie HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器。 Cookie: $Version=1; Skin=new;

Content-Length 请求的内容长度 Content-Length: 348

Content-Type 请求的与实体对应的MIME信息 Content-Type: application/x-www-form-urlencoded

## 五、Spring IOC/AOP

## 六、SpringMVC

## 七、SpringBoot

## 八、Mybatis

## 九、Redis

## 十、MQ

## 十一、Dubbo

### 1、Dubbo的RPC

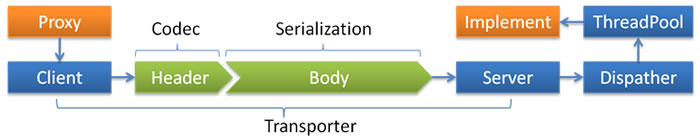
答：Dubbo不使用Http，因为Http面向连接的协议，需要3次握手，4次挥手，建立连接成本太高，而且协议复杂，内容很多

Dubbo优化：单一长连接，dubbo协议，简单，解析高效，字节少（header只有16字节）

Dubbo的Header，包括Response、Request。

1、发送请求 2、网络处理（默认是netty） 3、协议编解码 4、业务数据序列化（Hessian、FastJson等） 5、invoke

### 2、Dubbo的线程模型



<dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed" threads="100" />

**Dispatcher：**

all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。（默认）

direct 所有消息都不派发到线程池，全部在 IO 线程上直接执行。

message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。

execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。

connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

**ThreadPool：**

fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省) ，默认是100个线程，没有等待队列

cached 缓存线程池，空闲一分钟自动删除，需要时重建。

limited 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。

计算操作系统最大线程数：total\_memory/125k

Linux命令: ulimit –u

### 3、直连提供者

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService"

url="dubbo://localhost:20890" />

## 十二、数据库方面

## 十三、高并发问题处理

## 十四、分布式

1. 分布式服务
2. 分布式Session
3. 分布式Cache
4. 分布式事务
5. 分布式数据库

## 十五、其他问题

### 1、Tomcat，Apache，JBoss的区别

答：（1）Apache是Http服务器，Tomcat是web服务器，JBoss是应用服务器。

（2）Web服务器主要支持的是Http协议，通常说Http服务器和Web服务器是等同。

（3）以Java EE为例，Web服务器主要是处理静态页面处理和作为 Servlet容器，解释和执行servlet/JSP

应用服务器是运行业务逻辑的，主要是EJB、 JNDI和JMX API等J2EE API方面的，还包含事务处理、数据库连接等功能

IIS、Apache、Tomcat都可以属于Web服务器，Weblogic、WebSphere都属于应用服务器。

（4）Web服务器主要提供是Http协议，只是简单的通过响应(response)HTML页面来处理HTTP请求(request)。

应用服务器提供的商业逻辑，比如应用服务器使用Web Services，Web服务器接收WebService的数据，使用Http协议将数据展示处理。

### 2、Java的四个基本特性，对多态的理解以及在项目中那些地方用到多态

答：（1）四个基本特性：抽象、封装、继承，多态

（2）多态：方法重载（overload）、方法重写（override）

### 3、什么是RPC

答：客户端发起一个请求，由远程服务器进行接收并处理，这就是RPC过程

Http是RPC的一个具体实现，TCP

Http协议主要包括hander（头部信息，用于控制：Cookie、权限、协议编码）、body（业务数据json、xml、protobuf，序列化协议）