多线程并发

Part——多线程和并发专题

1.什么是缓存一致性问题？如何解决呢？

当程序在运行过程中，会将运算需要的数据从主存复制一份到 **CPU 的高速缓存**当中，那么 CPU 进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。举个简单的例子，比如下面的这段代码：

```
i = i+1
```

当线程执行这个语句时，会先从主存当中读取 i 的值，然后复制一份到高速缓存当中，然后 CPU 执行指令对 i 进行加 1 操作，然后将数据写入高速缓存，最后将高速缓存中 i 最新的值刷新到主存当中。

这个代码在单线程中运行是没有任何问题的，但是在多线程中运行就会有问题了。在多核 CPU 中，每条线程可能运行于不同的 CPU 中，因此每个线程运行时有自己的高速缓存（对单核 CPU 来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的）。**本文我们以多核 CPU 为例**。

比如同时有 2 个线程执行这段代码，假如初始时 i 的值为 0，那么我们希望两个线程执行完之后 i 的值变为 2。但是事实会是这样吗？

可能存在下面一种情况：初始时，两个线程分别读取 i 的值存入各自所在的 CPU 的高速缓存当中，然后线程 1 进行加 1 操作，然后把 i 的最新值 1 写入到内存。此时线程 2 的高速缓存当中 i 的值还是 0，进行加 1 操作之后，i 的值为 1，然后线程 2 把 i 的值写入内存。

最终结果 i 的值是 1，而不是 2。这就是著名的**缓存一致性问题**。通常称这种被多个线程访问的变量为共享变量。

也就是说，如果一个变量在多个 CPU 中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题。

为了解决缓存不一致性问题，通常来说有以下 **2 种解决方法**：

1） 通过**在总线加 LOCK#锁**的方式

2） 通过**缓存一致性协议**

在早期的 CPU 当中，是通过在总线上加 LOCK#锁的形式来解决缓存不一致的问题。**因为 CPU 和其他部件进行通信都是通过总线来进行的，如果对总线加 LOCK#锁的话，也就是说阻塞了其他 CPU 对其他部件访问（如内存）**，从而使得只能有一个 CPU 能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了 LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他 CPU 才能从变量 i 所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。

但是上面的方式会有一个问题，由于在锁住总线期间，其他 CPU 无法访问内存，导致效率低下。

所以就出现了缓存一致性协议。该协议保证了每个缓存中使用的**共享变量的副本是一致的**。它核心的思想是：**当 CPU 向内存写入数据**时，如果发现操作的变量是共享变量，即在其他 CPU 中也存在该变量的副本，会发出信号**通知其他 CPU 将该变量的**缓存行**置为无效状态**，因此当其他 CPU 需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它**就会从内存重新读取**。

2.简述volatile关键字（或volatile的内存语义或volatile的2个特性）。

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读取时的可见性，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。(volatile 解决了线程间共享变量的可见性问题)。

第一：使用 volatile 关键字会强制**将修改的值立即写入主存**；

第二：使用 volatile 关键字的话，当线程 2 进行修改时，会导致线程 1 的**工作内存中缓存变量 stop 的缓存行无效**（反映到硬件层的话，就是 CPU 的 L1 或者 L2 缓存中对应的缓存行无效）；

第三：由于线程 1 的工作内存中缓存变量 stop 的缓存行无效，所以线程 1 再次读取变量 stop 的值时**会去主存读取**。

那么，在线程 2 修改 stop 值时（当然这里包括 2 个操作，修改线程 2 工作内存中的值，然后将修改后的值写入内存），会使得线程 1 的工作内存中缓存变量 stop 的缓存行无效，然后线程 1 读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程 1 读取到的就是最新的正确的值。

2）禁止进行指令重排序，阻止编译器对代码的优化。

volatile 关键字禁止指令重排序有两层意思：

I)当程序执行到 **volatile 变量的读操作或者写操作**时，在其前面的操作的更改肯定全部已经进行，且**结果已经对后面的操作**可见；在其后面的操作肯定还没有进行；

II） **在进行指令优化时**，不能把 volatile 变量前面的语句放在其后面执行，也不能把 volatile 变量后面的语句放到其前面执行。

为了实现 volatile 的内存语义，加入 volatile 关键字时，编译器在生成字节码时，会在指令序列中插入内存屏障，会多出一个 lock 前缀指令。内存屏障是一组处理器指令，解决禁止指令重排序和内存可见性的问题。编译器和 CPU 可以在保证输出结果一样的情况下对指令重排序，使性能得到优化。处理器在进行重排序时是会考虑指令之间 的数据依赖性。

内存屏障，有 2 个作用：1.先于这个内存屏障的指令必须先执行，后于这个内存屏障的指令必须后执行。2.使得内存可见性。所以，**如果你的字段是 volatile，**在读指令前插入读屏障，可以让高速缓存中的数据失效，重新从主内存加载数据。在写指令之后插入写屏障，能让写入缓存的最新数据写回到主内存。

lock 前缀指令在多核处理器下会引发了两件事情：

1.将当前处理器中这个变量所在缓存行的数据会写回到系统内存。这个写回内存的操作会引起在其他 CPU 里缓存了该内存地址的数据无效。但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了 保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总 线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内 存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进 行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

2.它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面 的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全 部完成。

内存屏障可以被分为以下几种类型：

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。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

扩展：使用 volatile 关键字的场景。

双重校验锁 DCL(double checked locking)--使用 volatile 的场景之一。

3.简述JAVA 的内存模型。

区别于“**JVM 的内存模型**”。

Java 内存模型规定所有的变量都是存在**主存**当中（类似于前面说的物理内存），每个线程都有自己的**工作内存**（类似于前面的高速缓存）。线程对变量

的所有操作都必须在工作内存中进行，而不能直接对主存进行操作，并且每个线程不能访问其他线程的工作内存。

Java 内存模型的 Volatile 关键字和 原子性、可见性、有序性和 happens-before 关系等。

一．Volatile 关键字 解析见上面。

二．要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。

只要有一个没有被保证，就有可能会导致程序运行不正确。

1.原子性

原子性：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。

可以通过 Synchronized 和 Lock 实现“原子性”。

例题：请分析以下哪些操作是原子性操作。

```
x = 10;         //语句 1
y = x;          //语句 2
x++;            //语句 3
x = x + 1;      //语句 4
```

特别注意，在 java 中，只有对除 long 和 double 外的基本类型进行简单的赋值（如 int a=1）或读取操作，才是原子的。只要给 long 或 double 加上 volatile，操作就是原子的了。

语句 1 是原子性操作，其他三个语句都不是原子性操作。

语句 2 实际上包含 2 个操作，它先要去读取 x 的值，再将 x 的值写入工作内存，虽然读取 x 的值以及将 x 的值写入工作内存这 2 个操作都是原子性操作，但是合起来就不是原子性操作了。

同样的，x++和 x = x+1 包括 3 个操作：读取 x 的值，进行加 1 操作，写入新的值。

2.可见性

可见性是指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。

通过 Synchronized 和 Lock 和 volatile 实现“可见性”。

3.有序性

有序性：即程序执行的顺序按照代码的先后顺序执行。

我的理解就是一段程序代码的执行在单个线程中看起来是有序的。这个应该是程序看起来执行的顺序是按照代码顺序执行的，因为虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

三．**happens-before 原则（先行发生原则）：**

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个 unLock 操作先行发生于后面对同一个锁的 lock 操作

volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

传递规则：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出操作 A 先行发生于操作 C

线程启动规则：Thread 对象的 start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的 finalize()方法的开始

4.ava 中的同步容器类和缺陷。

在 Java 中，同步容器主要包括 2 类：

1）Vector、HashTable。

2）Collections 类中提供的**静态工厂方法创建的类**。Collections.synchronizedXXX()。

缺陷：

1.性能问题。

在有多个线程进行访问时，如果**多个线程都只是进行读取操作**，那么每个时刻就只能有一个线程进行读取，其他线程便只能等待，这些线程必须竞争同一把锁。

2.ConcurrentModificationException 异常。

在对 Vector 等容器进行迭代修改时，会报 ConcurrentModificationException 异常。但是在**并发容器中**（如 **ConcurrentHashMap，CopyOnWriteArrayList** 等）不会出现这个问题。

5.为什么说 ConcurrentHashMap 是弱一致性的？以及为何多个线程并发修改 ConcurrentHashMap 时不会报 ConcurrentModificationException？

1.ConcurrentHashMap#get()

正是因为 get 操作几乎所有时候都是一个无锁操作（get 中有一个 readValueUnderLock 调用，不过这句执行到的几率极小），使得同一个 Segment 实例上的 **put 和 get 可以同时进行，这就是 get 操作是弱一致的根本原因。**

2.ConcurrentHashMap#clear()

clear 方法很简单，看下代码即知。

```
public void clear() {
    for (int i = 0; i < segments.length; ++i)
        segments[i].clear();
}
```

因为没有全局的锁，在清除完一个 segment 之后，正在清理下一个 segment 的时候，**已经清理的 segment 可能又被加入了数据，因此 clear 返回的时候，ConcurrentHashMap 中是可能存在数据的。因此，clear 方法是弱一致的。**

ConcurrentHashMap 中的迭代器

在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出 ConcurrentModificationException 异常。如果未遍历的数组上的内容发生了

变化，则有可能反映到迭代过程中。这就是 ConcurrentHashMap **迭代器弱一致**的表现。

在这种迭代方式中，当 iterator 被创建后，集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变，更重要的，这保证了多个线程并发执行的连续性和扩展性，是性能提升的关键。

总结，ConcurrentHashMap 的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与 Hashtable 和同步的 HashMap 一样了。

6.CopyOnWriteArrayList 的实现原理。

CopyOnWrite 容器即**写时复制**的容器，也就是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，**再将原容器的引用指向新的容器（改变引用的指向）**。这样做的好处是我们可以对 CopyOnWrite 容器进行**并发的读，而不需要加锁**，因为当前容器不会添加任何元素。所以 **CopyOnWrite 容器也是一种读写分离的思想，读和写在不同的容器上进行，注意，写的时候需要加锁。**

1.以下代码是向 CopyOnWriteArrayList 中 **add 方法**的实现，可以发现在**添加的时候是需要加锁的**，否则多线程写的时候会 Copy 出 N 个副本出来。

```
public boolean add(E e) {
final ReentrantLock lock = this.lock;//加的是lock 锁
lock.lock();
try {
    Object[] elements = getArray();
    int len = elements.length;
    Object[] newElements = Arrays.copyOf(elements, len + 1);
    newElements[len] = e;
    setArray(newElements);//将原容器的引用指向新的容器;
    return true;
} finally {
    lock.unlock();
}
```

}

在 CopyOnWriteArrayList 里处理写操作（包括 add、remove、set 等）是**先将原始的数据通过 Arrays.copyof()来生成一份新的数组，然后在新的数据对象上进行写，写完后再将原来的引用指向到当前这个数据对象，这样保证了每次写都是在新的对象上。**然后读的时候就是**在引用的当前对象**上进行读（包括 get，iterator 等），不存在加锁和阻塞。

CopyOnWriteArrayList **中写操作需要大面积复制数组，所以性能肯定很差，**但是**读操作因为操作的对象和写操作不是同一个对象，读之间也不需要加锁，读和写之间的同步处理只是在写完后通过一个简单的“=”将引用指向新的数组对象上来，这个几乎不需要时间 ，这样读操作就很快很安全，适合在多线程里使用。**

2. **读的时候不需要加锁**，如果读的时候有线程正在向 CopyOnWriteArrayList 添加数据，读还是会读到旧的数据（**在原容器中进行读**。

```
public E get(int index) {
    return get(getArray(), index);
}
```

CopyOnWriteArrayList 在**读上效率很高**，由于，写的时候每次都要将源数组复制到一个新组数中，所以**写的效率不高**。

CopyOnWrite 容器有很多优点，但是同时也存在两个问题，即**内存占用问题**和**数据一致性问题**。

1. **内存占用问题**。因为 CopyOnWrite 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象。针对内存占用问题，可以

1）通过**压缩容器中的元素**的方法来减少大对象的内存消耗，比如，如果元素全是 10 进制的数字，可以考虑把它压缩成 36 进制或 64 进制。

2) 不使用 CopyOnWrite 容器，而使用其他的并发容器，如 **ConcurrentHashMap**。

2. **数据一致性问题**。CopyOnWrite 容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的数据，马上能读到，请不要使用 CopyOnWrite 容器！！

7. Java 中堆和栈有什么不同？

栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而**堆是所有线程共享的一片公用内存区域。**对象都在堆里创建，**为了提**

升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

8.Java 中的活锁,死锁,饥饿有什么区别？

死锁：是指两个或两个以上的进程在执行过程中，**因争夺资源**而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去，此时称系统处于死锁状态或系统产生了死锁。

饥饿：考虑一台打印机分配的例子，当有多个进程需要打印文件时，系统**按照短文件优先的策略排序**，该策略具有平均等待时间短的优点，似乎非常合理，但当短文件打印任务源源不断时，长文件的打印任务将被无限期地推迟，导致**饥饿**以至饿
死。

活锁：与饥饿相关的另外一个概念称为**活锁** ，在忙式等待条件下发生的饥饿，称为活锁。

不进入等待状态的等待称为忙式等待。另一种等待方式是**阻塞式等待**，进程得不到共享资源时将进入阻塞状态，让出 CPU 给其他进程使用。忙等待和阻塞式等待的相同之处在于进程都不具备继续向前推进的条件，不同之处在于**处于忙等待的进程不主动放弃 CPU** ，尽管 CPU 可能被剥夺，因而是低效的；而处于**阻塞状态的进程主动放弃 CPU** ，因而是高效的。

活锁的例子：如果事务 T1 封锁了数据 R,事务 T2 又请求封锁 R，于是 T2 等待。T3 也请求封锁 R，当 T1 释放了R 上的封锁后，**系统首先批准了 T3 的请求**，T2 仍然等待。然后 T4 又请求封锁 R，当 T3 释放了 R 上的封锁之后，**系统又批准了 T4 的请求** T2 可能永远等待（**在整个过程中，事务 T2 在不断的重复尝试获取锁 R**）。

活锁的时候，进程是不会阻塞的，这会导致耗尽 CPU 资源，**这是与死锁最明显的区别。**

活锁指的是任务或者执行者**没有被阻塞**，由于某些条件没有满足，**导致一直重复尝试，失败，尝试，失败**。 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；**活锁有一定几率解开**，而死锁是无法解开的。

避免活锁的简单方法是采**用先来先服务的策略**。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得锁。

线程池的问题

1.什么是线程池？线程池的工作原理和使用线程池的好处？

一个线程池管理了一组**工作线程**，同时它还包括了一个用于放置等待执行任务的**任务队列（阻塞队列）**。

默认情况下，在创建了线程池后，线程池中的线程数为 0。当**任务提交给线程池之后的处理策略如下：**

1. 如果此时线程池中的数量**小于 corePoolSize（核心池的大小）**，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务（**也就是每来一个任务，就要创建一个线程来执行任务**）。

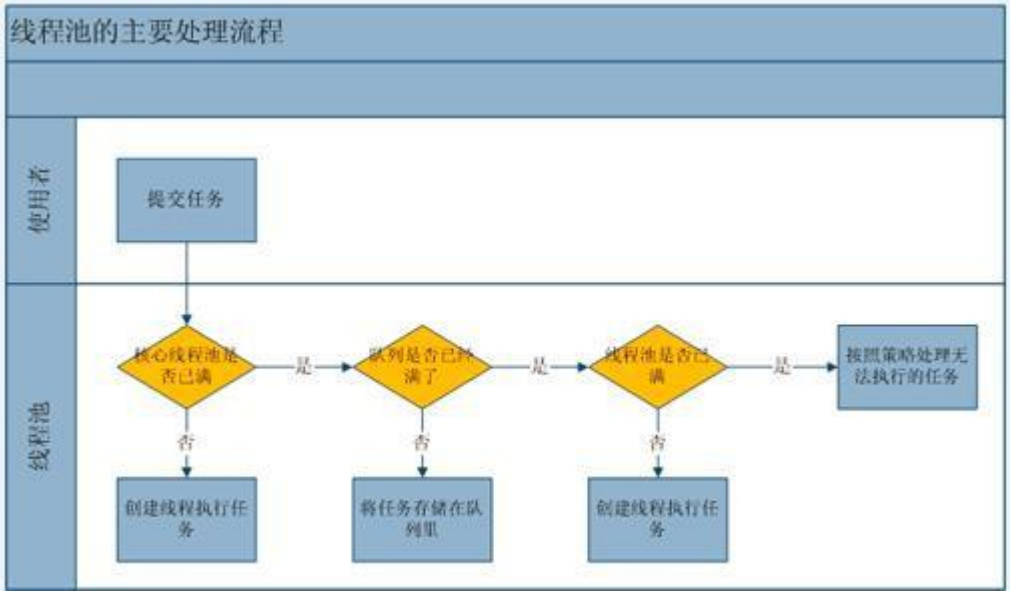
2. 如果此时线程池中的数量**大于等于 corePoolSize**，但是**缓冲队列 workQueue 未满**，那么任务被放入缓冲队列，则该任务会等待空闲线程将其取出去执行。

3. 如果此时线程池中的数量 **大于等于 corePoolSize**，**缓冲队列 workQueue 满**，并且线程池中的数量**小于 maximumPoolSize（线程池最大线程数）**，建新的线程来处理被添加的任务。

4. 如果此时线程池中的数量**大于等于 corePoolSize**，**缓冲队列 workQueue 满**，并且线程池中的数量**等于 maximumPoolSize**，那么通过 RejectedExecutionHandler 所指定的策略**(任务拒绝策略)**来处理此任务。

也就是处理任务的优先级为： **核心线程 corePoolSize、任务队列 workQueue、最大线程 maximumPoolSize，如果三者都满了，使用 handler 处理被拒绝的任务。**

5. 特别注意，在 **corePoolSize** 和 **maximumPoolSize** 之间的线程数会被自动释放。当线程池中线程数量大于 **corePoolSize** 时，如果某线程空闲时间超过 **keepAliveTime**，线程将被终止，直至线程池中的线程数目不大于 **corePoolSize**。这样，线程池可以动态的调整池中的线程数。



使用线程池的好处:

1.通过重复利用已创建的线程，**减少在创建和销毁线程上**所花的时间以及**系统资源的开销**。

2.提高响应速度。当任务到达时，任务可以不需要等到线程创建就可以立即执行。

3.提高线程的可管理性。使用线程池可以对线程进行统一的分配和监控。

4.**如果不使用线程池**，有可能造成系统创建大量线程而**导致消耗完系统内存**。

对于原理，有几个接口和类值得我们关注：

Executor 接口
Executors 类
ExecutorService 接口
AbstractExecutorService 抽象类
ThreadPoolExecutor 类

Executor 是一个顶层接口，在它里面只声明了一个方法 execute(Runnable)，返回值为 void，参数为 Runnable 类型，从字面意思可以理解，就是用来执行传进去的任务的；

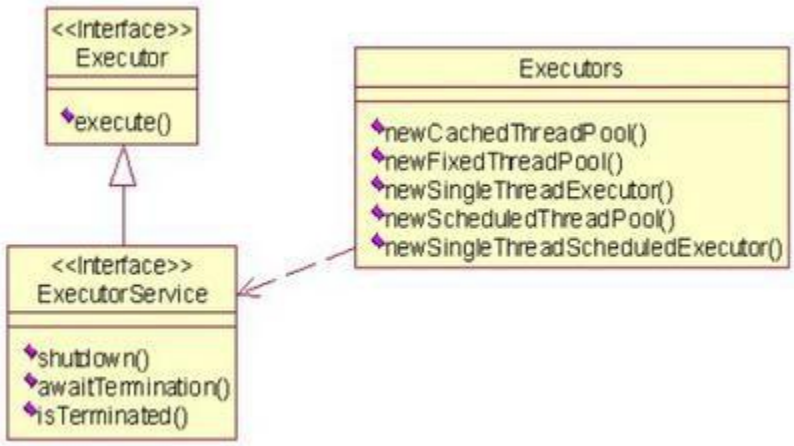
然后 ExecutorService 接口继承了 Executor 接口，并声明了一些方法：submit、invokeAll、invokeAny 以及 shutDown 等；

抽象类 AbstractExecutorService 实现了 ExecutorService 接口，基本实现了 ExecutorService 中声明的所有方法；

然后 ThreadPoolExecutor 继承了类 AbstractExecutorService。

public static ScheduledExecutorService **newScheduledThreadPool**(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代 Timer 类。



Executor 接口

```
public interface Executor {
    void execute(Runnable command);
}
```

Executor 接口只有一个方法 execute()，并且需要传入一个 Runnable 类型的参数。那么它的作用自然是 具体的执行参数传入的任务。

ExecutorService 接口

```
public interface ExecutorService extends Executor
    { void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);
     <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;
    .....
```

}

Executors 类：

它主要用来创建线程池。

```
Executors.newSingleThreadExecutor();    //创建容量为 1 的缓冲池
Executors.newFixedThreadPool(int n);    //创建固定容量大小的缓冲池
Executors.newCachedThreadPool();        //创建一个缓冲池，缓冲池最大容量为
Integer.MAX_VALUE（无界线程池）
```

下面是这三个静态方法的具体实现：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串 行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。 此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2. newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，**如果某个线程因为执行异常而结束，那么线程池会补充一个新线程**。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

ThreadPoolExecutor 类

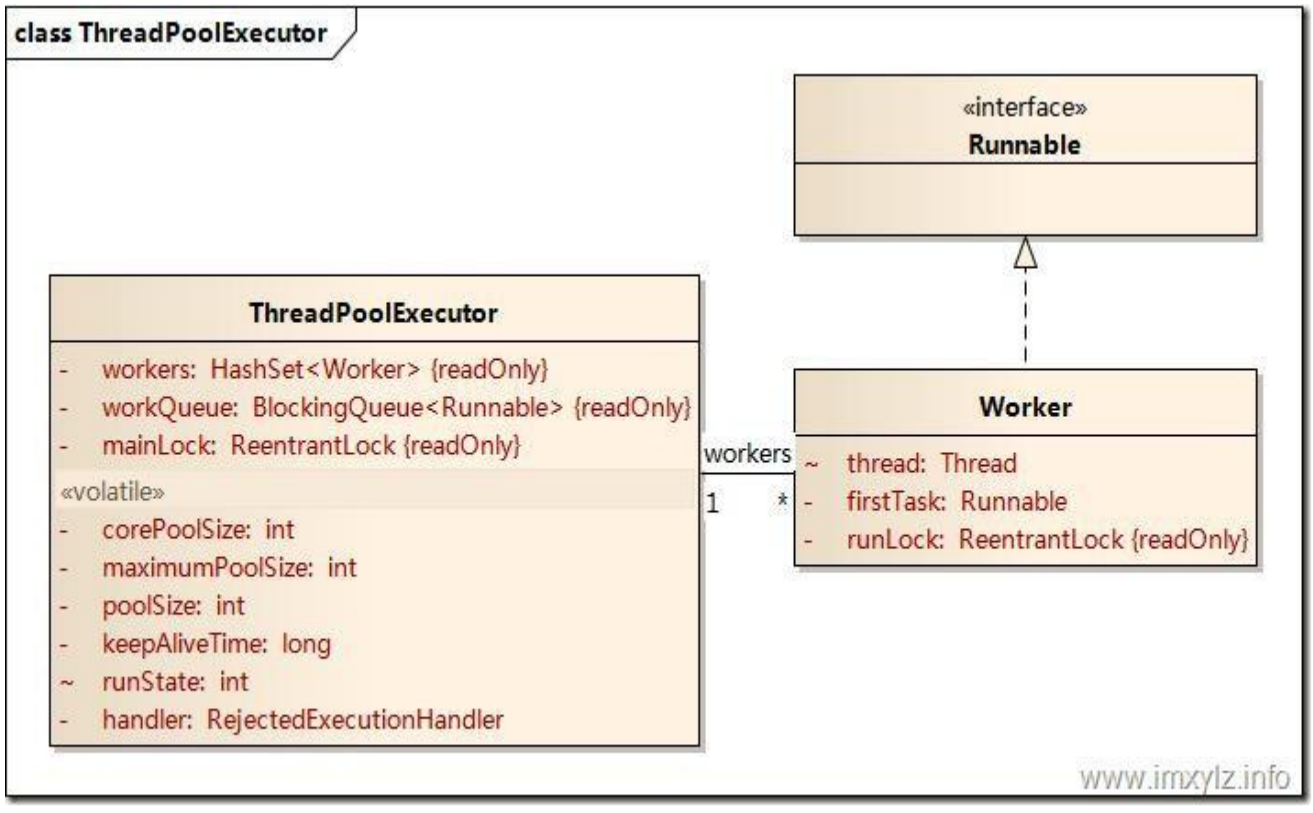


图1 ThreadPoolExecutor 数据结构

在 ThreadPoolExecutor 类中提供了四个构造方法：

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
        BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);
```

不过在 java doc 中，并**不提倡我们直接使用 ThreadPoolExecutor 创建线程池，而是使用 Executors 类中提供的几个静态方法来创建线程池**。下面解释一下构造器中各个参数的含义：

corePoolSize：核心池的大小。默认情况下，在创建了线程池后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到缓存队列当中。

maximumPoolSize：线程池最大线程数，它表示在线程池中最多能创建多少个线程。

keepAliveTime：默认情况下，只有当线程池中的线程数大于 corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于 corePoolSize 时，如果一个线程空闲的时间达到 keepAliveTime，则会终止，直到线程池中的线程数不超过 corePoolSize。

unit：参数 keepAliveTime 的时间单位。

workQueue：一个阻塞队列，任务缓存队列，即 workQueue，它用来存放等待执行的任务。

workQueue 的类型为 BlockingQueue<Runnable>，通常可以取下面三种类型：

1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为 Integer.MAX_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。在某次添加元素后必须等待其他线程取走后才能继续添加。

ThreadFactory：线程工厂，主要用来创建线程。

handler：表示当拒绝处理任务时的策略，有以下**四种取值**：

1. **AbortPolicy**：直接抛出异常（默认的）
2. **DiscardPolicy**：直接丢弃任务
3. **DiscardOldestPolicy**：丢弃队列中最旧（队头）的任务，并执行当前任务
4. **CallerRunsPolicy**：不用线程池中的线程执行，用调用者所在线程执行。

在 ThreadPoolExecutor 类中有几个非常重要的方法：

execute()

submit()

shutdown()

shutdownNow()

execute 和 submit 区别：

submit 有返回值，execute 没有返回值。 所以说可以根据任务有无返回值选择对应的方法。

submit 方便异常的处理。 如果任务可能会抛出异常，而且希望外面的调用者能够感知这些异常，那么就需要调用 submit 方法，通过捕获 Future.get 抛出的异常。

shutdown()和 shutdownNow()的区别：

shutdown()和 shutdownNow()是用来关闭线程池的。

shutdown 方法：此方法执行后不得向线程池再提交任务，如果有空闲线程则销毁空闲线程，等待所有正在执行的任务及位于阻塞队列中的任务执行结束，然后销毁所有线程。

shutdownNow 方法：此方法执行后不得向线程池再提交任务，如果有空闲线程则销毁空闲线程，取消所有位于阻塞队列中的任务，并将其放入 List<Runnable>容器，作为返回值。取消正在执行的线程（实际上仅仅是设置正在执行线程的中断标志位，调用线程的 interrupt 方法来中断线程）。

线程池的注意事项

虽然线程池是构建多线程应用程序的强大机制，但使用它并不是没有风险的。

（1） **线程池的大小**。多线程应用并非线程越多越好，需要根据系统运行的软硬件环境以及应用本身的特点决定线程池的大小。一般来说，如果代码结构合理的话，线程数目与 CPU 数量相适合即可。如果线程运行时可能出现阻塞现象，可相应增加池的大小；如有必要可采用自适应算法来动态调整线程池的大小，以提高 CPU 的有效利用率和系统的整体性能。

（2） **并发错误**。多线程应用要特别注意并发错误，要从逻辑上保证程序的正确性，注意**避 免死锁现象**的发生。

（3） **线程泄漏**。这是线程池应用中一个严重的问题，当任务执行完毕而线程没能返回池中就会发生线程泄漏现象。

简单线程池的设计

一个典型的线程池，应该包括如下几个部分：

1、线程池管理器（ThreadPool），用于启动、停用，管理线程池

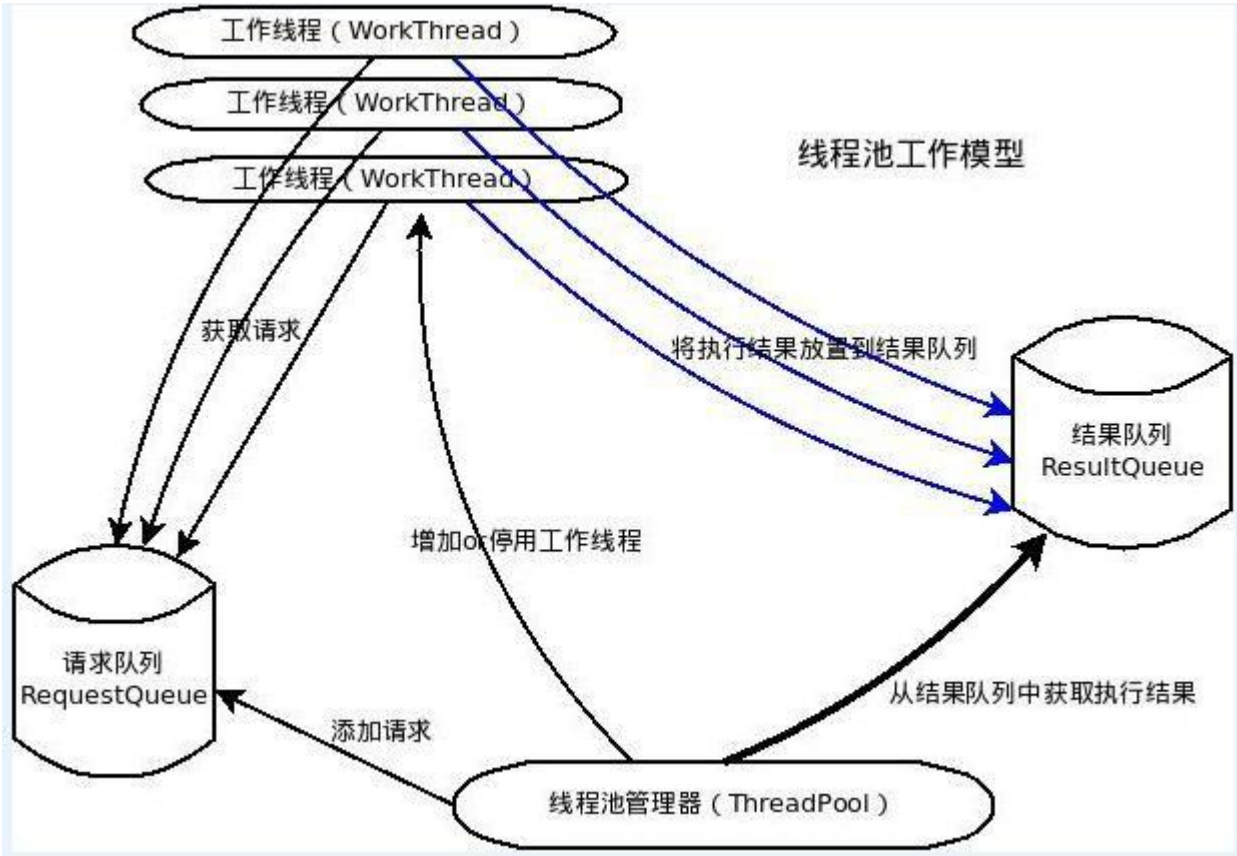
2、工作线程（WorkThread），线程池中的线程

3、**请求接口**（WorkRequest），**创建请求对象**，以供工作线程调度任务的执行

4、**请求队列**（RequestQueue）,用于存放和提取请求

5、**结果队列**（ResultQueue）,用于存储请求执行后返回的结果

线程池管理器，通过添加请求的方法（putRequest）向请求队列（RequestQueue）添加请求，**这些请求事先需要实现请求接口**，即传递工作函数、参数、结果处理函数、以及异常处理函数。之后初始化一定数量的工作线程，这些线程通过轮询的方式不断查看请求队列（RequestQueue），只要有请求存在，则会提取出请求，进行执行。然后，线程池 管理器调用方法（poll）查看结果队列（resultQueue）是否有值，如果有值，则取出，调用结果处理函数执行。通过以上讲述，不难发现，这个系统的核心资源在于请求队列和结果队列，工作线程通过轮询 requestQueue 获得任务，主线程通过查看结果队列，获得执行结果。因此，**对这个队列的设计，要实现线程同步**，以及一定阻塞和超时机制的设计，以防止因为不断轮询而导致的过多 cpu 开销。



如何合理的配置 **java** 线程池？如 **CPU** 密集型的任务，基本线程池应该配置多大？**IO** 密集型的任务，基本线程池应该配置多大？用有界队列好还是无界队列好？任务非常多的时候 使用什么阻塞队列能获取最好的吞吐量？

答： 1）配置线程池时，CPU 密集型任务可以少配置线程数，大概和机器的 cpu 核数相当，可以使得每个线程都在执行任务。

2） IO 密集型任务则由于需要等待 IO 操作，线程并不是一直在执行任务，则配置尽可能多的线程，2*cpu 核数。

3） 有界队列和无界队列的配置需区分业务场景，一般情况下配置有界队列，在一些可能会有爆发性增长的情况下使用无界队列。

4） 任务非常多时，使用非阻塞队列使用 CAS 操作替代锁可以获得好的吞吐量。synchronousQueue 吞吐率最高。

并发的问题

网站的高并发，大流量访问怎么解决？

1. HTML 页面静态化

访问频率较高但内容变动较小，使用网站 HTML 静态化方案来优化访问速度。将社区内的帖子、文章进行实时的静态化，有更新的时候再重新静态化也是大量使用的策略。

优势：

一、减轻服务器负担。

二、加快页面打开速度，**静态页面无需访问数据库**，打开速度较动态页面有明显提高；

三、很多搜索引擎都会优先收录静态页面，不仅被收录的快，还收录的全，容易被搜索引擎找到；

四、HTML 静态页面不会受程序相关漏洞的影响，减少攻击 ，提高安全性。

2. 图片服务器和应用服务器相分离

现在很多的网站上都会用到大量的图片，而图片是网页传输中占主要的数据量,也是影 响网站性能的主要因素。因此很多网站都会将图片存储从网站中分离出来，另外架构一个或多个服务器来存储图片，将图片放到一个虚拟目录中，而网页上的图片都用一个 URL 地址来指向这些服务器上的图片的地址，这样的话网站的性能就明显提高了。

优势：

一、 分担 Web 服务器的 I/O 负载-将耗费资源的图片服务分离出来，提高服务器的性能和稳定性。

二、 能够专门对图片服务器进行优化-为图片服务设置有针对性的**缓存**方案，减少带宽成本，提高访问速度。

三、 提高网站的可扩展性-通过增加图片服务器，提高图片吞吐能力。

3. 数据库

见“数据库部分的---如果有一个特别大的访问量到数据库上，怎么做优化？”。

4. 缓存

尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与 数据库的交互，提高性能。

假如我们能减少数据库频繁的访问，那对系统肯定大大有利的。比如一个电子商务系 统的商品搜索，如果某个关键字的商品经常被搜，那就可以考虑这部分商品列表存放到缓 存（内存中去），这样不用每次访问数据库，性能大大增加。

5. 镜像

镜像是冗余的一种类型，一个磁盘上的数据在另一个磁盘上存在一个完全相同的副本 即为镜像。

6. 负载均衡

在网站高并发访问的场景下，使用负载均衡技术（负载均衡服务器）为一个应用构建 一个由多台服务器组成的服务器集群，将并发访问请求分发到多台服务器上处理，避免单一服务器因负载压力过大而响应缓慢，使用户请求具有更好的响应延迟特性。

7. 并发控制

加锁，如乐观锁和悲观锁。

8.消息队列

通过 mq 一个一个排队方式，跟 12306 一样。

订票系统，某车次只有一张火车票，假定有 1w 个人同时打开 12306 网站来订票，如何解决并发问题？（可扩展到任何高并发网站要考虑的并发读写问题）。

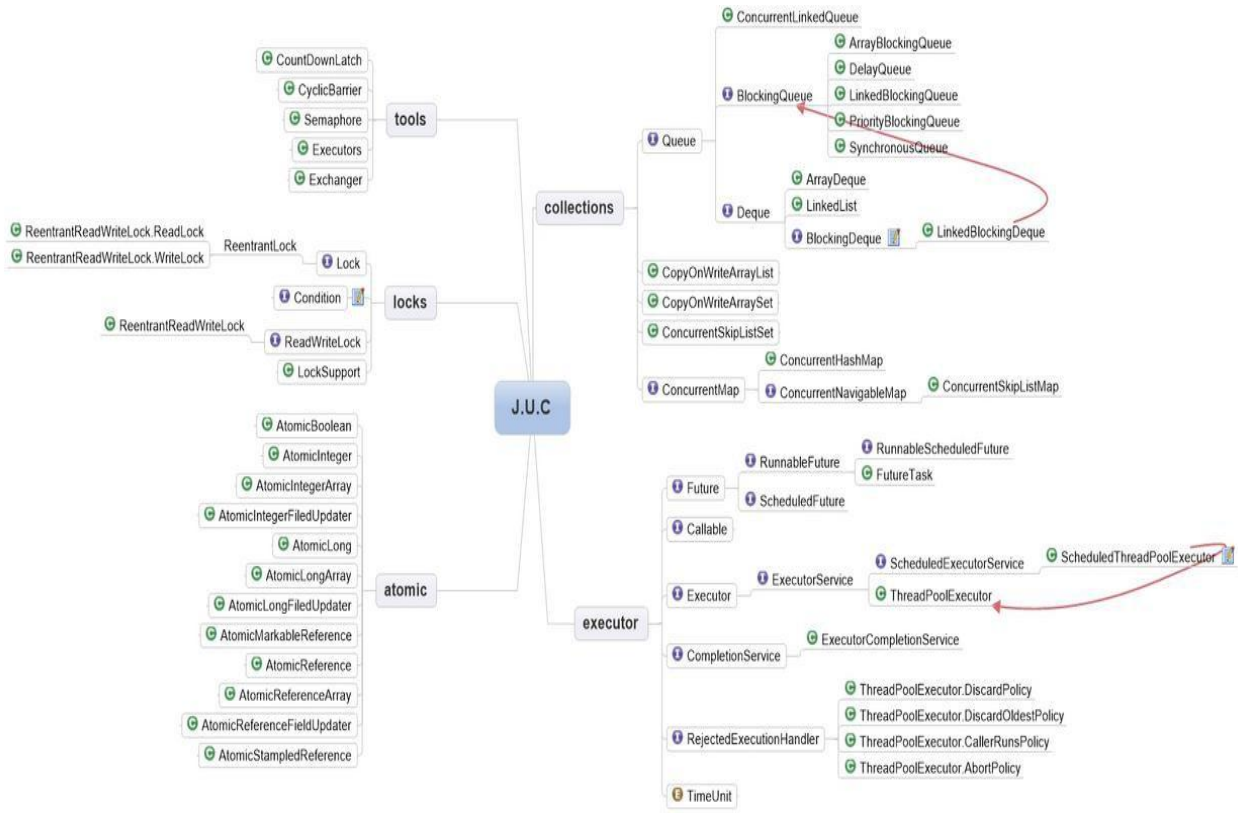
不但要保证 1w 个人能同时看到有票（数据的可读性），还要保证最终只能由一个人买到票（数据的排他性）。

使用数据库层面的并发访问控制机制。采用乐观锁即可解决此问题。乐观锁意思是不锁定表的情况下，利用业务的控制来解决并发问题，这样既保证数据的并发**可读性**，又保证保存数据的**排他性**，保证性能的同时解决了并发带来的脏数据问题。hibernate 中实现乐观锁。

银行两操作员同时操作同一账户就是典型的例子。比如 A、B 操作员同时读取一余额为 1000 元的账户，A操作员为该账户增加 100 元，B操作员同时为该账户减去 50 元，A先提交，B后提交。最后实际账户余额为 1000-50=950 元，但本该为 1000+100-50=1050。这就是典型的并发问题。如何解决？可以用锁。

并发包的问题

并发包（Concurrent 包）中含有的类：



ConcurrentHashMap / CopyOnWriteArrayList。

阻塞队列。

同步辅助类。

和线程池相关的类。

Lock 接口。

原子类。

同步辅助类的介绍如下：

1.CountDownLatch:

举例：

模拟一个场景，只有三个程序都干完活了，才算项目完成。实例如下：

```
import java.util.concurrent.CountDownLatch;
public class CountDownLatchDemo{
        public static void main(String args[]) throws Exception{
                CountDownLatch latch = new CountDownLatch(3);
```

```
        Worker worker1 = new Worker("Jack 程序员1" ,latch);
        Worker worker2 = new Worker("Rose 程序员2" ,latch);
        Worker worker3 = new Worker("Json 程序员3" ,latch);
        worker1.start();
        worker2.start();
        worker3.start();
        latch.await();
        System.out.println("Main thread end!");
    }
    static class Worker extends Thread {
        private String workerName;
        private CountDownLatch latch;
        public Worker(String workerName,CountDownLatch latch) {
            this.workerName = workerName;
            this.latch = latch;
        }
        @Override
        public void run() {
            try {
                System.out.println("Worker:" +workerName +" is begin.");
                Thread.sleep(1000L);
                System.out.println("Worker:" +workerName +" is end.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } //模拟下班.
            latch.countDown();
        }
    }
}
```

运行结果如下：

```
Worker:Jack 程序员1 is begin.
Worker:Rose 程序员2 is begin.
Worker:Json 程序员3 is begin.
Worker:Jack 程序员1 is end.
Worker:Json 程序员3 is end.
Worker:Rose 程序员2 is end.
Main thread end!
```

它相当于一个计数器。用一个给定的数值初始化 CountDownLatch，之后计

数器就从这个值开始倒计数，直到计数值达到零。

CountDownLatch 是通过“共享锁”实现的。在创建 CountDownLatch 时，会传递一个 int 类型参数，该参数是“锁计数器”的初始状态，表示该“共享锁”最多能被 count 个线程同时获取，这个值只能被设置一次，而且 CountDownLatch 没有提供任何机制去重新设置这个计数值。**主线程必须在启动其他线程后立即调用 await()方法**。这样**主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务**。当某线程调用该 CountDownLatch 对象的
await()方法时，该线程会等待“共享锁”可用时，才能获取“共享锁”进而继续运行。而“共享锁”可用的条件，就是“锁计数器”的值为 0！而“**锁计数器**”的初始值为 count，每当一个线程调用该 CountDownLatch 对象的 countDown()方法时， 才将“ 锁计数器”-1； 通过这种方式，必须有 count 个线程调用 countDown()之后，“锁计数器”才为 0，而前面提到的等待线程才能继续运行！

2 个重要的函数：

await()函数的作用是让线程阻塞等待其他线程，直到 CountDownLatch 的计数值变为 0，才继续执行之后的操作。

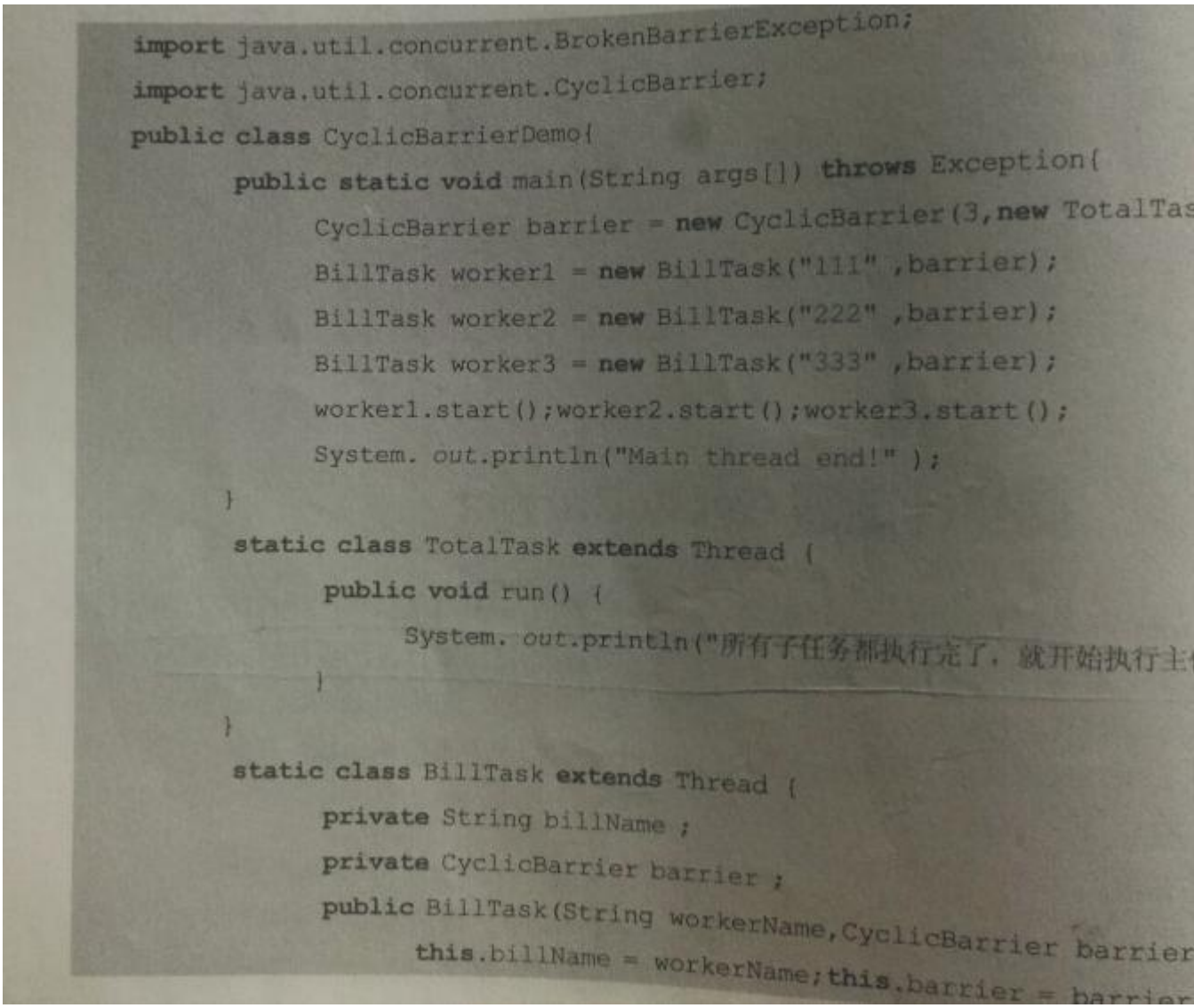
countDown()函数:这个函数用来将 CountDownLatch 的计数值减 1，如果计数达到 0，则释放所有等待的线程。

它的应用场景：

一个任务，它需要等待其他的一些任务都执行完毕之后它才能继续执行。

比如：开 5 个多线程去下载，当 5 个线程都执行完了才算下载成功。

2.CyclicBarrier



```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierDemo{
    public static void main(String args[]) throws Exception{
        CyclicBarrier barrier = new CyclicBarrier(3,new TotalTas
        BillTask worker1 = new BillTask("111" ,barrier);
        BillTask worker2 = new BillTask("222" ,barrier);
        BillTask worker3 = new BillTask("333" ,barrier);
        worker1.start();worker2.start();worker3.start();
        System. out.println("Main thread end!" );
    }
    static class TotalTask extends Thread {
        public void run() {
            System. out.println("所有子任务都执行完了，就开始执行主
        }
    }
    static class BillTask extends Thread {
        private String billName ;
        private CyclicBarrier barrier ;
        public BillTask(String workerName,CyclicBarrier barrier
            this.billName = workerName;this.barrier = barrier
```

```
            public void run() {
                try {
                    System.out.println("市区:" +billName +"运算开始: " );
                    Thread.sleep(1000L);//模仿第一次运算
                    System.out.println("市区:" +billName +"运算完成，等待
中..." );

                    barrier.await();//假设一次运算不完，第二次要依赖第一次的运算
结果，都到达这个节点之后后面才会继续执行。  ↓等待所有线程都调用过此函数才能进行后续动作.
                    System.out.println("全部都结束，市区" +billName +"才开始后
面的工作。" );

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

运行结果如下：

```
Main thread end!
市区:111运算开始:
市区:333运算开始:
市区:222运算开始:
市区:222运算完成，等待中...
市区:111运算完成，等待中...
市区:333运算完成，等待中...
所有子任务都执行完了，就开始执行主任务了。
全部都结束，市区333才开始后面的工作。
全部都结束，市区222才开始后面的工作。
全部都结束，市区111才开始后面的工作。
```

CyclicBarrier 翻译过来就是：循环的屏障，这个类是一个可以重复利用的屏障类。它允许一组线程相互等待，直到全部到达某个公共屏障点，然后所有的这组线程再同步往后执行。

1 个重要的函数：

await()函数每被调用一次，计数便会减少1（CyclicBarrier 设置了初始值），并阻塞住当前线程。当计数减至 0 时，阻塞解除，所有在此 CyclicBarrier 上面阻塞的线程开始运行。

CountDownLatch 和 CyclicBarrier 的区别？

(1) CountDownLatch 的作用是允许 1 个线程等待其他线程执行完成之后，它才执行；而 CyclicBarrier 则是允许 N 个线程相互等待到某个公共屏障点，然后这一组线程再同时执行。

(2) CountDownLatch 的**计数器的值无法被重置**，这个初始值只能被设置一

次，是不能够重用的；CyclicBarrier 是可以重用的。

3.Semaphore

可以**控制某个资源可被同时访问的个数**，通过构造函数设定一定数量的许可，通过 **acquire()** 获取一个许可，如果没有就等待，而 **release()** 释放一个许可。

下面的例子只允许 5 个线程同时进入执行acquire()和 release()之间的代码：

```
public class SemaphoreTest {
    public static void main(String[] args) {
        // 线程池
        ExecutorService exec = Executors.newCachedThreadPool();
        // 只能 5 个线程同时访问
        final Semaphore semp = new Semaphore(5);
        // 模拟 20 个客户端访问
        for (int index = 0; index < 20; index++) {
            final int NO = index;
            Runnable run = new Runnable() {
                public void run() {
                    try {
                        // 获取许可
                        semp.acquire(); System.out.println("Accessing:
                        " + NO); Thread.sleep((long) (Math.random() *
                        10000));
                        // 访问完后，释放 ，如果屏蔽下面的语句，则在控制台只能
打印 5 条记录，之后线程一直阻塞
                        semp.release();
                    } catch (InterruptedException e) {
                    }
                }
            };
            exec.execute(run);
        }
        // 退出线程池
        exec.shutdown();
    }
}
```

阻塞队列相关的问题

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是： 在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程， 消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

1. ArrayBlockingQueue

ArrayBlockingQueue 是一个由数组支持的有界缓存的阻塞队列。在**读写操作上都需要锁住整个容器**，因此吞吐量与一般的实现是相似的，适合于实现“生产者消费者”模式。ArrayBlockingQueue 内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置。这个类是线程安全的。**生产者和消费者共用一把锁**。

源码：

```
/** The queued items */
final Object[] items;
/** items index for next take, poll, peek or remove */
int takeIndex;
/** items index for next put, offer, or add */
int putIndex;
/** Number of elements in the queue */
int count;
/** Main lock guarding all access */
final ReentrantLock lock;
/** Condition for waiting takes */
private final Condition notEmpty;
/** Condition for waiting puts */
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

```
private void enqueue(E x) {
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal();
 }

private E dequeue() {
    // assert lock.getHoldCount() == 1;
    // assert items[takeIndex] != null;
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length)
        takeIndex = 0;   //循环队列
    count--;
    if (itrs != null)
        itrs.elementDequeued();
    notFull.signal();
    return x;
  }

public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
```

```
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
    }

    public boolean offer(E e) {
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            if (count == items.length)
                return false;
            else {
                enqueue(e);
                return true;
            }
        } finally {
            lock.unlock();
        }
    }

    public E poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return (count == 0) ? null : dequeue();
        } finally {
            lock.unlock();
        }
    }
```

2. LinkedBlockingQueue

基于链表的阻塞队列，内部维持着一个数据缓冲队列（该队列由链表构成）。

只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue 可以通过构造函数指定该值），才会阻塞生产者线程，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。

LinkedBlockingQueue 之所以能够高效的处理并发数据，还因为其对于**生产者端和消费者端分别采用了独立的锁来控制数据同步**，这也意味着在高并发的情况下生产者和消费者可以**并行地**操作队列中的数据，**以此来提高整个队列的并发性能**。

源码：

```
/** The capacity bound, or Integer.MAX_VALUE if none */
    private final int capacity;
    /** Current number of elements */
    private final AtomicInteger count = new AtomicInteger();
    /**
     * Head of linked list.
     * Invariant: head.item == null
     */
    transient Node<E> head;
    /**
     * Tail of linked list.
     * Invariant: last.next == null
     */
    private transient Node<E> last;
    /** Lock held by take, poll, etc */
    private final ReentrantLock takeLock = new ReentrantLock();
    /** Wait queue for waiting takes */
    private final Condition notEmpty = takeLock.newCondition();
    /** Lock held by put, offer, etc */
    private final ReentrantLock putLock = new ReentrantLock();
    /** Wait queue for waiting puts */
    private final Condition notFull = putLock.newCondition();

    private void enqueue(Node<E> node) {
        // assert putLock.isHeldByCurrentThread();
        // assert last.next == null;
        last = last.next = node;
    }

    private E dequeue() {
        // assert takeLock.isHeldByCurrentThread();
        // assert head.item == null;
        Node<E> h = head;
        Node<E> first = h.next;
        h.next = h; // help GC
        head = first;
        E x = first.item;
        first.item = null;
```

```
        return x;
    }

  public void put(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        // Note: convention in all put/take/etc is to preset local
var
        // holding count negative to indicate failure unless set.
        int c = -1;
        Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();
        try {
          //当队列满时，调用notFull.await()方法释放锁，陷入等待状态。
         //有两种情况会激活该线程
         //第一、 某个put 线程添加元素后，发现队列有空余，就调用 notFull.signal()方法激
活阻塞线程
         //第二、take 线程取元素时，发现队列已满。则其取出元素后，也会调用 notFull.signal()
方法激活阻塞线程
            while (count.get() == capacity)
                { notFull.await();
            }
            enqueue(node);
            c = count.getAndIncrement();
  //发现队列未满，调用 notFull.signal()激活阻塞的put 线程（可能存在）
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
     }

    public E take() throws InterruptedException
        { E x;
        int c = -1;
        final AtomicInteger count = this.count;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lockInterruptibly();
        try {
```

```
            while (count.get() == 0)
                { notEmpty.await();
            }
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1)
                notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
        if (c == capacity)
            signalNotFull();
        return x;
}

    public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        final AtomicInteger count = this.count;
        if (count.get() == capacity)
            return false;
        int c = -1;
        Node<E> node = new Node<E>(e);
        final ReentrantLock putLock = this.putLock;
        putLock.lock();
        try {
            if (count.get() < capacity)
                { enqueue(node);
                c = count.getAndIncrement();
                if (c + 1 < capacity)
                    notFull.signal();
            }
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
        return c >= 0;
 }

    public E poll() {
        final AtomicInteger count = this.count;
```

```
        if (count.get() == 0)
            return null;
        E x = null;
        int c = -1;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
            if (count.get() > 0)
                { x = dequeue();
                c = count.getAndDecrement();
                if (c > 1)
                    notEmpty.signal();
            }
        } finally {
            takeLock.unlock();
        }
        if (c == capacity)
            signalNotFull();
        return x;
    }
```

ArrayBlockingQueue 和 LinkedBlockingQueue 的区别：

1. 队列大小的初始化方式不同

ArrayBlockingQueue 是有界的，必须指定队列的大小；

LinkedBlockingQueue 是分情况的，指定队列的大小时，就是有界的；不指定队列的大小时，默认是 Integer.MAX_VALUE，看成无界队列，但当生产速度大于消费速度时候，有可能会内存溢出。

2. 队列中锁的实现不同

ArrayBlockingQueue 实现的队列中的**锁是没有分离的**，即生产和消费用的是同一个锁；进行 put 和 take 操作，共用同一个锁对象。也即是说，put 和 take 无法并行执行！

LinkedBlockingQueue 实现的队列中的**锁是分离的**， 即生产用的是 putLock， 消费是 takeLock。也就是说，生成端和消费端各自独立拥有一把锁，避免了读（take）写（put）时互 相竞争锁的情况，可并行执行。

3. 在生产或消费时操作不同

ArrayBlockingQueue 基于数组，在插入或删除元素时，是直接将枚举对象插入或移除的，**不会产生或销毁任何额外的对象实例**；

LinkedBlockingQueue 基于链表，在插入或删除元素时，**需要把枚举对象转换为 Node<E> 进行插入或移除，会生成一个额外的 Node 对象**，这在长时间内需要高效并发地处理大批量数据的系统中，其对于 GC 的影响还是存在一定的区别，会影响性能。

Put()和 take()方法。

都可以实现**阻塞**的功能。

Put()方法：把元素加入到阻塞队列中，如果阻塞队列没有空间，则调用此方法的线程被阻塞，直到有空间的时候再继续。

take()方法：取出排在阻塞队列首位的对象，若阻塞队列为空，则调用此方法的线程被阻塞，直到有新的对象被加入的时候再继续。

offer()和 poll()方法。

不具有阻塞的功能。

offer()方法：把元素加入到阻塞队列中，如果可以容纳，则**返回 true**。如果不可以容纳，则**返回 false**。

poll()方法：取出排在阻塞队列首位的对象，若阻塞队列为空，则**返回 null**,如果不为空，则**返回取出来的那个元素**。

3. PriorityBlockingQueue VS PriorityQueue

此阻塞队列为**基于数组**的**无界**阻塞队列。它会按照元素的优先级对元素进行排序，按照优先级顺序出队，**每次出队的元素都是优先级最高的元素**。注意，**不会阻塞生产者，但会阻塞消费者。**PriorityBlockingQueue 里面**存储的对象必须是实现 Comparable 接口**，队列通过这个接口的 compare 方法确定对象的 priority。

队列的元素并不是全部按优先级排序的，但是**队头的优先级肯定是最高的**。每取一个头元素时候，都会对剩余的元素做一次调整，这样就能保证每次队头的元素都是优先级最高的元素。

4. DelayQueue

DelayQueue 是一个**无界**阻塞队列，用于放置实现了 Delayed 接口的对象，**只有在延迟期满时才能从中提取元素**。该队列的**头部是延迟期满后保存时间最长的 Delayed 元素**。 这个队列里面所存储的对象都带有一个时间参数，**采用 take 获取数据的时候，如果时间没有到，取不出来任何数据。而加入数据的时候，是不会阻塞的（不会阻塞生产者，但会阻塞消费者）。**

DelayQueue 内部使用 PriorityQueue 实现的。DelayQueue 是一个使用 PriorityQueue 实现的 BlockingQueue，**优先队列的比较基准值是时间**。本质上即： DelayQueue = BlockingQueue +PriorityQueue + Delayed。

优势：

如果不使用 DelayQueue，那么常规的解决办法就是：使用一个后台线程，遍历所有对象，挨个检查。这种笨笨的办法简单好用，但是对象数量过多时，可能存在性能问题，检查间隔时间不好设置，间隔时间过大，影响精确度，过小则存在效率问题。而且做不到按超时的时间顺序处理。

应用场景：

缓存系统的设计。缓存中的对象，超过了有效时间，需要从缓存中移出。使用一个线程循 环查询 DelayQueue，**一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。**

```
class Wangming implements Delayed {

    private String name;
    // 身份证
    private String id;
    // 截止时间
    private long endTime;

    public Wangming(String name, String id, long endTime) {
        this.name = name;
        this.id = id;
        this.endTime = endTime;
    }

    public String getName() {
        return this.name;
    }

    public String getId() {
        return this.id;
    }

    /**
     * 用来判断是否到了截止时间
     */
    @Override
    public long getDelay(TimeUnit unit) {
        // TODO Auto-generated method stub
        return endTime - System.currentTimeMillis();
    }

    /**
```

```
     * 相互比较排序用
     */
    @Override
    public int compareTo(Delayed o) {
        // TODO Auto-generated method stub
        Wangming jia = (Wangming) o;
        return endTime - jia.endTime > 0 ? 1 : 0;
    }
}

public class WangBa implements Runnable {

private DelayQueue<Wangming> queue = new DelayQueue<Wangming>();
    public boolean yinye = true;

    public void shangji(String name, String id, int money)
        { Wangming man = new Wangming(name, id, 1000 * 60 *
        money +
System.currentTimeMillis());
        System.out.println("网名" + man.getName() + " 身份证" +
man.getId() + "交钱" + money + "块,开始上机...");
        this.queue.add(man);
    }

    public void xiaji(Wangming man) {
        System.out.println("网名" + man.getName() + " 身份证" +
man.getId() + "时间到下机...");
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        while (yinye) {
            try {
                System.out.println("检查 ing");
                Wangming man = queue.take();
                xiaji(man);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
```

```
    public static void main(String args[]) {
        try {
            System.out.println("网吧开始营业");
            WangBa siyu = new WangBa();
            Thread shangwang = new Thread(siyu);
            shangwang.start();

            siyu.shangji("路人甲", "123", 1);
            siyu.shangji("路人乙", "234", 2);
            siyu.shangji("路人丙", "345", 3);
        } catch (Exception ex) {

        }

    }
}
```

5. SynchronousQueue

同步队列是一个不存储元素的队列，**它的 size()方法总是返回 0**。每个线程的插入操作必须等待另一个线程的移除操作，同样任何一个线程的移除操作都必须等待另一个线程的插入操作。可以认为 SynchronousQueue 是一个缓存值为 1 的阻塞队列。

生产者/消费者问题的多种实现方式

1. 使用阻塞队列实现

```
// Producer Class in java
class Producer implements Runnable {

    private final BlockingQueue sharedQueue;

    public Producer(BlockingQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
```

```
            try {
                System.out.println("Produced: " + i);
                sharedQueue.put(i);
            } catch (InterruptedException ex)
                { System.out.println(ex);
            }
        }
    }
}

// Consumer Class in Java
class Consumer implements Runnable {

    private final BlockingQueue sharedQueue;

    public Consumer(BlockingQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }

    public void run() {
        while (true) {
            try {
                int i = (Integer) sharedQueue.take();
                System.out.println("Consumed: " + i);
            } catch (InterruptedException ex)
                { System.out.println(ex);
            }
        }
    }
}

public class ProducerConsumerPattern {

    public static void main(String args[]) {

        // Creating shared object
        BlockingQueue sharedQueue = new LinkedBlockingQueue();

        // Creating Producer and Consumer Thread
        Thread prodThread = new Thread(new Producer(sharedQueue));
        Thread consThread = new Thread(new Consumer(sharedQueue));
```

```
        // Starting producer and Consumer thread
        prodThread.start();
        consThread.start();
    }
}
```

2. 使用Object 的wait()和 notify()实现

```
PriorityQueue<Integer> queue = new PriorityQueue<Integer>(10);//
充当缓冲区

class Consumer extends Thread {

        public void run() {
            while (true) {
                synchronized (queue) {
                    while (queue.size() == 0) {//队列空的条件下阻塞
                        try
                            { queue.wait()
                            ;
                        } catch (InterruptedException e)
                            { e.printStackTrace();
                            queue.notify();
                        }
                    }
                    queue.poll(); // 每次移走队首元素
                    queue.notify();
                }
            }
        }
    }

    class Producer extends Thread {

        public void run() {
            while (true) {
                synchronized (queue) {
                    while (queue.size() == 10) {//队列满了的条件下阻
塞
                        try
```

```
{ queue.wait()
;
```

```
                    } catch (InterruptedException e)
                        { e.printStackTrace();
                        queue.notify();
                    }
                }
                queue.offer(1); // 每次插入一个元素
                queue.notify();
            }
        }
    }
```

3. 使用Condition 实现

```
    private PriorityQueue<Integer> queue = new
PriorityQueue<Integer>(10);
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

class Consumer extends Thread {

        public void run() {
            while (true)
                { lock.lock()
                ; try {
                    while (queue.size() == 0) {
                        try
                            { notEmpty.await()
                            ;
                        } catch (InterruptedException e)
                            { e.printStackTrace();
                        }
                    }
                    queue.poll(); // 每次移走队首元素
                    notFull.signal();
                } finally
                    { lock.unlock()
                    ;
                }
            }
        }
```

```
}
```

```
class Producer extends Thread {

   public void run() {
      while (true)
         { lock.lock()
         ; try {
            while (queue.size() == 10) {
               try
                  { notFull.await()
                  ;
               } catch (InterruptedException e)
                  { e.printStackTrace();
               }
            }
            queue.offer(1); // 每次插入一个元素
            notEmpty.signal();
         } finally
            { lock.unlock()
            ;
         }
      }
   }

}
```

编程实现一个最大元素为 100 的阻塞队列。

```
 Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();

Object[] items = new Object[100];
int putptr, takeptr, count;

public void put(Object x) throws InterruptedException
   { lock.lock();
   try {
      while (count == items.length)
         notFull.await();
      items[putptr] = x;
```

```
if (++putptr == items.length)
```

```
            putptr = 0;
        ++count;
        notEmpty.signal();
    } finally
        { lock.unlock()
        ;
    }
}

public Object take() throws InterruptedException
    { lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length)
            takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally
        { lock.unlock()
        ;
    }
}
```

设计一个双缓冲阻塞队列，写代码。

在服务器开发中，通常的做法是**把逻辑处理线程和 I/O 处理线程分离**。

逻辑处理线程：对接收的包进行逻辑处理。

I/O 处理线程：网络数据的发送和接收，连接的建立和维护。

通常逻辑处理线程和 I/O 处理线程是通过**数据队列**来交换数据，就是**生产者--消费者模型**。

这个数据队列是多个线程在共享，每次访问都需要加锁，因此如何减少互斥/同步的开销就显得尤为重要。**解决方案：双缓冲队列**。

两个队列，将读写分离，一个给逻辑线程读，一个给 IO 线程用来写，当逻辑线程读完队列后会将自己的队列与 IO 线程的队列相调换。这里需要加锁的地方有两个，一个是 IO 线程每次写队列时都要加锁，另一个是逻辑线程在调换队列时也需要加锁，但逻辑线程在读队列时是不需要加锁的。如果是一块缓冲

区，读、写操作是不分离的，双缓冲区起码节省了单缓冲区时读部分操作互斥/同步的开销。本质是采用空间换时间的优化思路。

Java 中的队列都有哪些，有什么区别。

队列都实现了 Queue 接口。

阻塞队列和非阻塞队列。

阻塞队列：见上面的讲解。

非阻塞队列：LinkedList，PriorityQueue。

多线程相关的问题

如果不用锁机制如何实现共享数据访问。（不要用锁，不要用 sychronized 块或者方法，也不要直接使用 jdk 提供的线程安全的数据结构，需要自己实现一个类来保证多个线程同时读写这个类中的共享数据是线程安全的，怎么办？）

无锁化编程的常用方法：硬件CPU 同步原语 CAS（Compare and Swap），如无锁栈，无锁队列（ConcurrentLinkedQueue）等等。现在几乎所有的CPU 指令都支持CAS 的原子操作，X86 下对应的是CMPXCHG 汇编指令，处理器执行 CMPXCHG 指令是一个原子性操作。有了这个原子操作，我们就可以用其来实现各种无锁（lock free）的数据结构。

CAS 实现了区别于 sychronized 同步锁的一种乐观锁，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改后的新值 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。其实 CAS 也算是有锁操作，只不过是由 CPU 来触发，比 synchronized 性能好的多。CAS 的关键点在于，系统**在硬件层面保证了比较并交换操作的原子性**， 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。CAS 是非阻塞算法的一种常见实现。

一个线程间共享的变量，首先在主存中会保留一份，然后每个线程的工作内存也会保留一份副本。这里说的预期值，就是线程保留的副本。当该线程从主存中获取该变量的值后，主存中该变量可能已经被其他线程刷新了，但是该线程工作内存中该变量却还是原来的值，这就是所谓的预期值了。当你要用 CAS 刷新该值的时候，如果发现线程工作内存和主存中不一致了，就会失败，如果一致，就可以更新成功。

Atomic 包提供了一系列原子类。这些类可以保证多线程环境下，当某个线程在执行 atomic 的方法时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个线程执行。Atomic 类在软件层面上是非阻塞的，它的原子性其实是在硬件层面上借助相关的指令来保证的。

AtomicInteger 是一个支持原子操作的 Integer 类，就是保证对 AtomicInteger 类型变量的增加和减少操作是原子性的，不会出现多个线程下的数据不一致问题。如果不使用 AtomicInteger，要实现一个按顺序获取的

ID，就必须在每次获取时进行加锁操作，以避免出现并发时获取到同样的 ID 的现象。Java 并发库中的 AtomicXXX 类均是基于这个原语的实现，拿出 AtomicInteger 来研究在没有锁的情况下是如何做到数据正确性的：

来看看++i 是怎么做到的。

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

在这里采用了 CAS 操作，每次从内存中读取数据然后将此数据和+1 后的结果进行 CAS 操作，如果成功就返回结果，否则重试直到成功为止。

而 compareAndSet 利用 JNI 来完成 CPU 指令的操作，非阻塞算法。

```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect,
update);

}
```

其中，unsafe.compareAndSwapInt（）是一个 native 方法，正是调用 CAS 原语完成该操作。

首先假设有一个变量 i，i的初始值为 0。每个线程都对 i 进行+1 操作。CAS 是这样保证同步的：
假设有两个线程，线程 1 读取内存中的值为 0，current = 0，next = 1，然后挂起，然后线程 2 对 i 进行操作，将 i 的值变成了 1。线程 2 执行完，回到线程 1，进入 if 里的 compareAndSet 方法，该方法进行的操作的逻辑是，（1）如果操作数的值在内存中没有被修改，返回 true，然后 compareAndSet 方法返回 next 的值（2）如果操作数的值在内存中被修改了，则返回 false，重新进入下一次循环，重新得到 current 的值为 1，next 的值为 2，然后再比较，由于这次没有被修改，所以直接返回 2。

那么，为什么自增操作要通过 CAS 来完成呢？仔细观察 incrementAndGet()方法，发现自增操作其实拆成了两步完成的：

```
int current = get();
int next = current + 1;
```

由于 volatile 只能保证读取或写入的是最新值，那么可能出现以下情况：

1.A 线程执行 get()操作，获取 current 值（假设为 1）

2.B 线程执行 get()操作，获取 current 值（为 1）

3.B 线程执行 next = current + 1 操作，next = 2

4.A 线程执行 next = current + 1 操作，next = 2

这样的结果明显不是我们想要的，所以，自增操作必须采用 CAS 来完成。

CAS 的优缺点

CAS 由于是在硬件层面保证的原子性，不会锁住当前线程，它的效率是很高的。

CAS 虽然很高效的实现了原子操作，但是它依然存在三个问题。

1、ABA 问题。CAS 在操作值的时候检查值是否已经变化，没有变化的情况下才会进行更新。但是如果一个值原来是 A，变成 B，又变成 A，那么 CAS 进行检查时会认为这个值没有变化，操作成功。ABA 问题的解决方法是使用版本号。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。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。这个类的 compareAndSet 方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

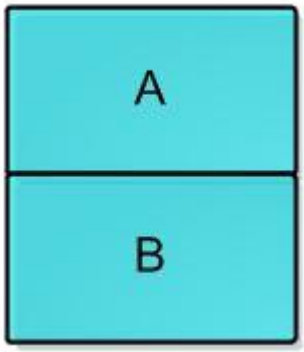
CAS 算法实现一个重要前提是需要取出内存中某时刻的数据，而在下一时刻把取出后的数据和内存中原始数据比较并替换，那么在这个时间差内会导致数据的变化。

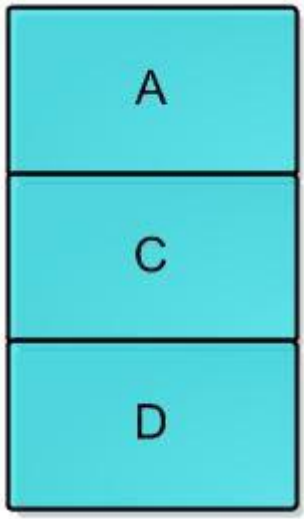
比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。如果链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。因此前面提到的原子操作 AtomicStampedReference/AtomicMarkableReference 就很有用了。这允许一对变化的元素进行原子操作。

现有一个用单向链表实现的堆栈，栈顶为 A，这时线程 T1 已经知道 A.next 为 B，然后希望用 CAS 将栈顶替换为 B：

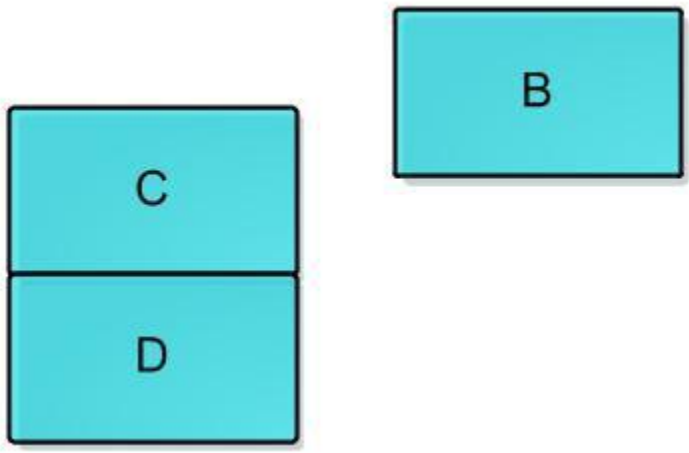
head.compareAndSet(A,B);

在 T1 执行上面这条指令之前，线程 T2 介入，将 A、B 出栈，再 pushD、C、A，此时堆栈结构如下图，而对象 B 此时处于游离状态：





此时轮到线程 T1 执行 CAS 操作，检测发现栈顶仍为 A，所以 CAS 成功，栈顶变为 B，但实际上 B.next 为 null，所以此时的情况变为：



其中堆栈中只有 B 一个元素，C 和 D 组成的链表不再存在于堆栈中，平白无故就把 C、D 丢掉了。

以上就是由于 ABA 问题带来的隐患，各种乐观锁的实现中通常都会用版本号 version 来对记录或对象标记，避免并发操作带来的问题。在 Java 中，AtomicStampedReference<E>也实现了这个作用，它通过包装[E,Integer]的元组来对对象标记版本戳 stamp，从而避免 ABA 问题。

2、循环时间长开销大。自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。因此 CAS 不适合竞争十分频繁的场景。

3. 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。

这里粘贴一个，模拟 CAS 实现的计数器：

```
public class CASCount implements Runnable {

    private SimilatedCAS counter = new SimilatedCAS();

    public void run() {
         for (int i = 0; i < 10000; i++)
                { System.out.println(this.increment())
                ;
         }
    }
```

```
    public int increment() {
        int oldValue = counter.getValue();
        int newValue = oldValue + 1;

        while (!counter.compareAndSwap(oldValue, newValue)) { //
如果 CAS 失败,就去拿新值继续执行 CAS
              oldValue = counter.getValue();
              newValue = oldValue + 1;
        }

        return newValue;
    }

    public static void main(String[] args)
        { Runnable run = new CASCount();

        new Thread(run).start();
        new Thread(run).start();
        new Thread(run).start();
        new Thread(run).start();
        new Thread(run).start();
    }
}
class SimilatedCAS {
    private int value;

    public int getValue() {
        return value;
    }

    // 这里只能用 synchronized 了,毕竟无法调用操作系统的 CAS
    public synchronized boolean compareAndSwap(int expectedValue,
int newValue) {
        if (value == expectedValue)
            { value = newValue;
            return true;
        }
        return false;
    }

}
```

说一下java 中的多线程。

1.Java 中实现多线程的四种方式（创建多线程的四种方式）？

一、继承 Thread 类创建线程类

⑴ 定义 Thread 类的子类，并重写该类的 run 方法，该 run 方法的方法体就代表了线程要完成的任务。因此把 run()方法称为执行体。

⑵ 创建 Thread 子类的实例，即创建了线程对象。

⑶ 调用线程对象的 start()方法来启动该线程。

二、通过 Runnable 接口创建线程类

⑴ 定义 Runnable 接口的实现类，并重写该接口的 run()方法，该 run()方法的方法体同样是该线程的线程执行体。

⑵ 创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。

⑶ 调用线程对象的 start()方法来启动该线程。

三、通过 Callable 和 Future 创建线程

⑴ 创建 Callable 接口的实现类，并实现 call()方法，该 call()方法将作为线程执行体，并且**有返回值**。

⑵ 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 **FutureTask** 对象**封装了该 Callable 对象的 call()方法的返回值**。

⑶ 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

⑷ 调用 FutureTask 对象的 get()方法来获得子线程执行结束后的返回值。

四、通过线程池创建线程

利用线程池不用 new 就可以创建线程，线程可复用，利用 Executors 创建线程池。

扩展 1：Java 中 Runnable 和 Callable 有什么不同？

Ⅰ.Callable 定义的方法是 call()，而 Runnable 定义的方法是 run()。

II. Callable 的 call 方法可以有返回值，而 Runnable 的 run 方法不能有返回值。

III. Callable 的 call 方法可抛出异常，而 Runnable 的 run 方法不能抛出异常。

扩展 2：一个类是否可以同时继承 Thread 和实现 Runnable 接口？

可以。比如下面的程序可以通过编译。因为 Test 类从 Thread 类中继承了 run()方法，这个 run()方法可以被当作对 Runnable 接口的实现。

```
public class Test extends Thread implements Runnable {

   public static void main(String[] args)
      { Thread t = new Thread(new Test());
      t.start();
   }

}
```

2.实现多线程的同步。

在多线程的环境中，经常会遇到**数据的共享问题**，即当多个线程需要访问同一资源时，他们需要以某种顺序来确保该资源在某一时刻只能被一个线程使用，否则，程序的运行结果将会是不可预料的，在这种情况下，就必须对数据进行同步。

在 Java 中，提供了四种方式来实现同步互斥访问：synchronized 和 Lock 和 wait()/notify()/notifyAll()方法和 CAS。

一. synchronized 的用法。

1.同步代码块

synchronized 块写法：

synchronized(object)

{

}

表示线程在执行的时候会将 object 对象上锁。（注意这个对象可以是任意类的对象，也可以使用 this 关键字或者是 class 对象）。

可能一个方法中只有几行代码会涉及到线程同步问题，所以 synchronized 块比 synchronized 方法更加 **细粒度地** 控制了多个线程的访问， 只有 synchronized 块中的内容不能同时被多个线程所访问，方法中的其他语句仍然可以同时被多个线程所访问（包括 synchronized 块之前的和之后的）。

2.修饰非静态的方法

当 synchronized 关键字修饰一个方法的时候，该方法叫做同步方法。

Java 中的每个对象都有一个锁（lock），或者叫做监视器（monitor），当一个线程访问某个对象的 synchronized 方法时，**将该对象上锁**，其他任何线程都无法再去访问该对象的 synchronized 方法了（这里是指所有的同步方法，而不仅仅是同一个方法），直到之前的那个线程执行方法完毕后（**或者是抛出了异常**），才将该对象的锁释放掉，其他线程才有可能再去访问该对象的 synchronized 方法。

注意这时候是**给对象上锁**，如果是不同的对象，则各个对象之间没有限制关系。

注意，如果一个对象有多个 synchronized 方法，某一时刻某个线程已经进入到了某个 synchronized 方法，那么在该方法没有执行完毕前，其他线程是无法访问该对象的任何 synchronized 方法的。

3.修饰静态的方法

当一个 synchronized 关键字修饰的方法同时又被 static 修饰，之前说过，非静态的同步方法会将对象上锁，但是静态方法不属于对象，而是属于类，它会**将这个方法所在的类的 Class 对象上锁**。一个类不管生成多少个对象，它们所对应的是同一个 Class 对象。

因此，当线程分别访问同一个类的两个对象的两个 static，synchronized 方法时，它们的执行顺序也是顺序的，也就是说一个线程先去执行方法，执行完毕后另一个线程才开始。

结论：

synchronized 方法是一种**粗粒度的并发控制**，某一时刻，只能有一个线程执行该 synchronized 方法。

synchronized 块则是一种**细粒度的并发控制**，只会将块中的代码同步，位于方法内，synchronized 块之外的其他代码是可以被多个线程同时访问到的。

二．Lock 的用法。

使用 Lock 必须在 try-catch-finally 块中进行，并且将释放锁的操作放在 finally 块中进行，以保证锁一定被释放，**防止死锁的发生**。通常使用 Lock 来进行同步的话，是以下面这种形式去使用的：

```
Lock lock = ...;
lock.lock();
try{
   //处理任务
}catch(Exception ex){

}finally{
   lock.unlock();   //释放锁
}
```

Lock 和 synchronized 的区别和 Lock 的优势。你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？

1）Lock 是一个**接口**，而 synchronized 是 Java 中的**关键字**，synchronized 是**内置**的语言实现；

2）synchronized 在**发生异常时**，会**自动释放**线程占有的锁，因此**不会导致死锁现象发生**；而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很**可能造成死锁现象**，因此**使用 Lock 时需要在 finally 块中释放锁**；

3）Lock 可以让**等待锁的线程响应中断（可中断锁）**，而 synchronized **却不行**，使用 synchronized 时，等待的线程会一直等待下去，**不能够响应中断（不可中断锁）**；

4）通过 Lock 可以知道**有没有成功获取锁（tryLock（）方法：如果获取了锁，则返回** `true`**；否则返回** `false`**，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。）**，而 synchronized 却无法办到。

5）Lock 可以提高**多个线程进行读操作**的效率（**读写锁**）。

6）Lock 可以实现**公平锁**，synchronized **不保证公平性**。

在性能上来说，如果线程竞争资源不激烈时，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时 Lock 的性能要远远优于 synchronized。所以说，在具体使用时要根据适当情况选择。

扩展 1：volatile 和 synchronized 区别。

1.volatile 是变量修饰符，而synchronized 则作用于代码块或方法。

2.volatile 不会对变量加锁，不会造成线程的阻塞；synchronized 会对变量加锁，可能会造成线程的阻塞。

3.volatile 仅能实现变量的修改可见性， 并不能保证原子性； 而 synchronized 则可以保证变量的修改可见性和原子性。（synchronized 有两个重要含义：它确保了一次只有一个线程可以执行代码的受保护部分（互斥），而且它确保了一个线程更改的数据对于其它线程是可见的（更改的可见性），在释放锁之前会将对变量的修改 刷新到主存中）。

4.volatile 标记的变量不会被编译器优化，禁止指令重排序；synchronized 标记的变量可以被编译器优化。

扩展 2：什么场景下可以使用 volatile 替换synchronized？

只需要保证共享资源的可见性的时候可以使用 volatile 替代，synchronized 保证可操作的原子性，一致性和可见性。

三．wait()\notify()\notifyAll()的用法（Java 中怎样唤醒一个阻塞的线程？）。

在 Java 发展史上曾经使用 suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用 Object 类的 wait()和 notify()方法实现线程阻塞。

首先，wait、notify 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行；其次，wait、notify 方法必须在 synchronized 块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

扩展 1：为什么 wait(),notify(),notifyAll()等方法都定义在 Object 类中？

因为这三个方法都需要定义在同步代码块或同步方法中，这些方法的调用是依赖锁对象的，而同步代码块或同步方法中的锁对象可以是任意对象，那么能被任意对象调用的方法一定定义在 Object 类中。

扩展 2：notify()和 notifyAll()有什么区别？

notify()和 notifyAll()都是 Object 对象用于通知处在等待该对象的线程的方法。

void notify(): 唤醒一个正在等待该对象的线程，进入就绪队列等待 CPU 的调度。

void notifyAll(): 唤醒所有正在等待该对象的线程，进入就绪队列等待 CPU 的调度。

两者的最大区别在于：

notifyAll 使所有原来在该对象上等待被 notify 的线程统统退出 wait 的状态，**变成等待该对象上的锁**，一旦该对象被解锁，他们就会去竞争。

notify 他只是选择一个 wait 状态线程进行通知，并使它获得该对象上的锁，但不惊动其他同样在等待被该对象 notify 的线程们，当第一个线程运行完毕以后释放对象上的锁，此时如果该对象没有再次使用 notify 语句，即便该对象已经空闲，其他 wait 状态等待的线程由

于没有得到该对象的通知，继续处在 wait 状态，直到这个对象发出一个 notify 或 notifyAll，它们等待的是被 notify 或 notifyAll，而不是锁。

四．CAS

它是一种非阻塞的同步方式。具体参见上面的部分。

扩展一：同步锁的分类？

Synchronized 和 Lock 都是悲观锁。

乐观锁，CAS 同步原语，如原子类，非阻塞同步方式。

扩展二：锁的分类？

一种是代码层次上的，如 java 中的同步锁，可重入锁,公平锁,读写锁。

另外一种是数据库层次上的，比较典型的就是悲观锁和乐观锁，表锁，行锁，页锁。

扩展三：java 中的悲观锁和乐观锁？

悲观锁：悲观锁是认为肯定有其他线程来争夺资源，因此不管到底会不会发生争夺，悲观锁总是会先去锁住资源，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。Synchronized 和 Lock 都是悲观锁。

乐观锁：每次不加锁，假设没有冲突去完成某项操作，如果因为冲突失败就重试，直到成功为止。就是当去做某个修改或其他操作的时候它认为不会有其他线程来做同样的操作（竞争），这是一种乐观的态度，通常是基于 CAS 原子指令来实现的。CAS 通常不会将线程挂起，因此有时性能会好一些。乐观锁的一种实现方式——CAS。

3.实现线程之间的通信？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。

1.Object 类中 **wait()\notify()\notifyAll()**方法。

2.用 **Condition** 接口。

Condition 是被绑定到 Lock 上的，要创建一个 Lock 的 Condition 对象必须用 newCondition()方法。在一个 Lock 对象里面可以创建多个 Condition 对象，线程可以注册在指定的 Condition 对象中，从而可以有选择性地进行线程通知，在线程调度上更加灵活。

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。调用 Condition 对象中的方法时，需要被包含在 lock()和 unlock()之间。

3.**管道**实现线程间的通信。

实现方式：一个线程发送数据到输出管道流，另一个线程从输入管道流中读取数据。

基本流程：

1）创建管道输出流 PipedOutputStream pos 和管道输入流 PipedInputStream pis。

2）将 pos 和 pis 匹配，pos.connect(pis)。

3）将 pos 赋给信息输入信息的线程，pis 赋给获取信息的线程，就可以实现线程间的通讯了。

```
consumer1:0
consumer1:1
consumer1:2
consumer1:3
consumer2:4
consumer2:5
consumer1:6
```

缺点：

1）管道流只能在两个线程之间传递数据。

线程 consumer1 和 consumer2 同时从 pis 中 read 数据，当线程 producer 往管道流中写入一段数据（1,2,3,4,5,6）后，**每一个时刻**只有一个线程能获取到数据，并不是两个线程都能获取到 producer 发送来的数据，因此一个管道流只能用于两个线程间的通讯。

2）管道流只能实现单向发送，如果要两个线程之间互通讯，则需要两个管道流。

线程 producer 通过管道流向线程 consumer 发送数据，如果线程 consumer 想给线程 producer 发送数据，则需要新建另一个管道流 pos1 和 pis1，将 pos1 赋给 consumer1，将 pis1 赋给 producer1。

4. 使用 **volatile** 关键字

见上面部分。

4. 如何确保线程安全？

如果多个线程同时运行某段代码，**如果每次运行结果和单线程运行的结果是一样的**，而且**其他变量的值也和预期的是一样的**，就是**线程安全的**。

Synchronized，Lock，原子类（如 atomicinteger 等），同步容器、并发容器、阻塞队列、同步辅助类（比如 CountDownLatch, Semaphore, CyclicBarrier）。

5.多线程的优点和缺点？

优点：

1.充分利用 cpu，避免 cpu 空转。

2.程序响应更快。

缺点：

1.上下文切换的开销

当 CPU 从执行一个线程切换到执行另外一个线程的时候，它需要先存储当前线程的本地的数据，程序指针等，然后载入另一个线程的本地数据，程序指针等，最后才开始执行。这种切换称为“上下文切换”。CPU 会在一个上下文中执行一个线程，然后切换到另外一个上下文中执行另外一个线程。上下文切换并不廉价。如果没有必要，应该减少上下文切换的发生。

2.增加资源消耗

线程在运行的时候需要从计算机里面得到一些资源。 除了 CPU，线程还需要一些**内存**来维持它本地的堆栈。它也需要**占用操作系统中一些资源**来管理线程。

3.编程更复杂

在多线程访问共享数据的时候，要考虑**线程安全问题。**

6.写出 3 条你遵循的多线程最佳实践。

1.给线程起个有意义的名字。

2.避免锁定和缩小同步的范围 。

相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。

3.多用同步辅助类，少用 wait 和 notify 。

首先，CountDownLatch, Semaphore, CyclicBarrier 这些同步辅助类简化了编码操作，而用 wait 和 notify 很难实现对复杂控制流的控制。其次，

3.多线程中锁的种类。

1.可重入锁

ReentrantLock 和 synchronized 都是可重入锁。

如果当前线程已经获得了某个监视器对象所持有的锁，那么该线程在该方法中调用另外一个同步方法也同样持有该锁。

这些类是由最好的企业编写和维护在后续的 JDK 中它们还会不断优化和完善，比如：

使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

4.多用并发容器，少用同步容器。

如果下一次你需要用到 map，你应该首先想到用 ConcurrentHashMap。

```
public synchronized void test()
{ xxxxxx;
  test2();
}
public synchronized void test2()
{ yyyyy;
}
```

在上面代码段中，执行 test 方法需要获得当前对象作为监视器的对象锁，但方法中又调用了 test2 的同步方法。

7.多线程的性能一定就优于单线程吗？

对于**单核 CPU**，如果是 **CPU 密集型任务**，如解压文件，多线程的性能反而不如单线程性能，因为解压文件需要一直占用 CPU 资源，如果采用多线程，**线程切换**导致的开销反而会让性能下降。如果是**交互类型的任务**，肯定是需要使用多线程的。

对于**多核 CPU**，对于解压文件来说，多线程肯定优于单线程，因为多个线程能够更加充分利用每个核的资源。

如果锁是具有可重入性的话，那么该线程在调用 test2 时并不需要再次获得当前对象的锁，可以直接进入 test2 方法进行操作。

可重入锁最大的作用是避免死锁。如果锁是不具有可重入性的话，那么该线程在调用 test2 前会等待当前对象锁的释放，实际上该对象锁已被当前线程所持有，不可能再次获得，那么线程在调用同步方法、含有锁的方法时就会**产生死锁**。

2.可中断锁

顾名思义，就是可以响应中断的锁。

在 Java 中，synchronized 不是可中断锁，而 Lock 是可中断锁。lockInterruptibly()的用法已经体现了 Lock 的可中断性。如果某一线程 A 正在执行锁中的代码，另一线程 B 正在等待获取该锁，可能由于等待时间过长，线程 B 不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中断它，这种就是可中断锁。

3.公平锁

在 Java 中，synchronized 就是非公平锁，它无法保证等待的线程获取锁的顺序。而对于 ReentrantLock 和 ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

公平锁即**尽量**以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该锁，这种就是公平锁。

4.读写锁

正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。ReadWriteLock 就是读写锁，它是一个接口，ReentrantReadWriteLock 实现了这个接口。可以通过 readLock()获取读锁，通过 writeLock()获取写锁。

9.锁优化

1.自旋锁

为了让线程等待，让线程执行一个忙循环(自旋)。需要物理机器有一个以上的处理器。自旋等待虽然避免了线程切换的开销，但它是要占用处理器时间的，所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之自旋的线程只会白白消耗处理器资源。自旋次数的默认值是 10 次，可以使用参数 -XX:PreBlockSpin 来更改。

自适应自旋锁：自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

2.锁清除

指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行清除(逃逸分析技术：在堆上的所有数据都不会逃逸出去被其它线程访问到，可以把它们当成栈上数据对待)。

3.锁粗化

如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部。

4.轻量级锁

在代码进入同步块时，如果此同步对象没有被锁定，虚拟机首先将在当前线程的栈帧中建立一个名为锁记录(Lock Record)的空间，用于存储所对象目前的 Mark Word 的拷贝。然后虚拟机将使用 CAS 操作尝试将对象的 Mark Word 更新为执行 Lock Record 的指针。如果成功，那么这个线程就拥有了该对象的锁。如果更新操作失败，虚拟机首先会检查对象的 Mark Word 是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，否则说明这个对象已经被其它线程抢占。如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁。

解锁过程：如果对象的 Mark Word 仍然指向着线程的锁记录，那就用 CAS 操作把对象当前的 Mark Word 和和线程中复制的 Displaced Mark Word 替换回来，如果替换成功，整个过程就完成。如果失败，说明有其他线程尝试过获取该锁，那就要在释放锁的同时，唤醒被挂起的线程。

轻量级锁的依据：对于绝大部分的锁，在整个同步周期内都是不存在竞争的。

传统锁(重量级锁)使用操作系统互斥量来实现的。

HotSpot 虚拟机的对象的内存布局：对象头(Object Header)分为两部分信息吗，第一部分(**Mark Word**)用于存储对象自身的运行时数据，另一个部分用于存储指向方法区对象数据类型的指针，如果是数组的话，还会由一个额外的部分用于存储数组的长度。

32 位 HotSpot 虚拟机中对象未被锁定的状态下 Mark Word 的 32 个 Bits 空间中 25 位用于存储对象哈希码，4 位存储对象分代年龄，2 位存储锁标志位，1 位固定为 0。

HotSpot 虚拟机对象头 Mark Word

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀(重量级锁)
空，不记录信息	11	GC 标记
偏向线程 ID，偏向时间戳、对象分代年龄	01	可偏向

5.偏向锁

目的是消除在无竞争情况下的同步原语，进一步提高程序的运行性能。锁会偏向第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其它线程获取，则持有锁的线程将永远不需要再进行同步。

当锁第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为 01，同时使用 CAS 操作把获取到这个锁的线程的 ID 记录在对象的 Mark Word 之中，如果成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，都可以不进行任何同步操作。

当有另一个线程去尝试获取这个锁时，偏向模式就宣告结束。根据锁对象目前是否处于被锁定的状态，撤销偏向后恢复到未锁定或轻量级锁定状态。

10. wait()和 sleep()的区别。

1、这两个方法来自**不同的类**，sleep()来自 Thread 类，是静态方法；wait()是 Object 类里面的方法，和 notify()或者 notifyAll()方法配套使用，来实现线程间的通信。

2、**最主要**是 sleep 是将当前线程挂起指定的时间，没有**释放锁**；而 wait 方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3、**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 。

```
synchronized(x){
    x.notify()
  //或者 wait()
}
```

特别注意：sleep 和 wait **必须捕获异常**(Thread.sleep()和 Object.wait()都会抛出 InterruptedException)，notify 和 notifyAll 不需要捕获异常。

11. Java 中 interrupted() 和 isInterrupted()方法的区别？

二个方法都是判断线程是否停止的方法。

1.前者是**静态方法**，后者是**非静态方法**。interrupted 是作用于**当前正在运行的线程**，isInterrupted 是作用于**调用该方法的线程对象所对应的线程**。（线程对象对应的线程不一定是当前运行的线程。例如我们可以在 A 线程中去调用 B 线程对象的 isInterrupted 方法，此时，当前正在运行的线程就是 A 线程。）

2.前者会将中断状态清除而后者不会。

12. Java 创建线程之后，直接调用start()方法和run()的区别 ？

1.start（）方法来启动线程，并在新线程中运行 run()方法，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码；通过调用 Thread 类的 **start()方法来启动一个线程，这时此线程是处于就绪状态，并没有运行，**然后通过此 Thread 类调用方法 run()来完成其运行操作，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，run（）方法运行结束，此线程终止。然后 CPU 再调度其它线程。

2.直接调用 run()方法的话，会把 run（）方法当作普通方法来调用，会在当前线程中执行 run()方法，而不会启动新线程来运行 run()方法。程序还

是要顺序执行，要等待 run 方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到多线程的目的。

13.什么是线程的上下文切换？

对于单核 CPU，CPU 在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

线程上下文切换过程中会记录**程序计数器**、**CPU 寄存器的状态**等数据。

虽然多线程可以使得**任务执行的效率得到提升**，但是由于在**线程切换时同样会带来一定的开销代价**，并且多个线程会导致**系统资源占用的增加**，所以在进行多线程编程时要注意这些因素。

14.怎么检测一个线程是否拥有锁？

在 java.lang.Thread 中有一个方法叫 holdsLock(Object obj)，它返回 true，如果当且仅当当前线程拥有某个具体对象的锁。

15.用户线程和守护线程有什么区别？

当我们在 Java 程序中创建一个线程，它就被称为用户线程。将一个用户线程设置为守护线程的方法就是**在调用 start()方法之前**， 调用对象的 setDamon(true)方法。一个守护线程是在后台执行并且不会阻止 JVM 终止的线程，守护线程的作用是为其他线程的运行提供便利服务。**当没有用户线程在运行的时候，JVM 关闭程序并且退出。**一个守护线程创建的子线程依然是守护线程。

守护线程的一个典型例子就是**垃圾回收器**。

16.什么是线程调度器？

线程调度器是一个操作系统服务，它负责为 **Runnable 状态的线程**分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于**线程调度器**的实现。

17.线程的状态。

版本 1.

在 Java 当中，**线程通常都有五种状态**，创建、就绪、运行、阻塞和死亡。

第一是创建状态。在生成线程对象，并没有调用该对象的 start 方法，这是线程处于创建状态。

第二是就绪状态。当调用了线程对象的 start 方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

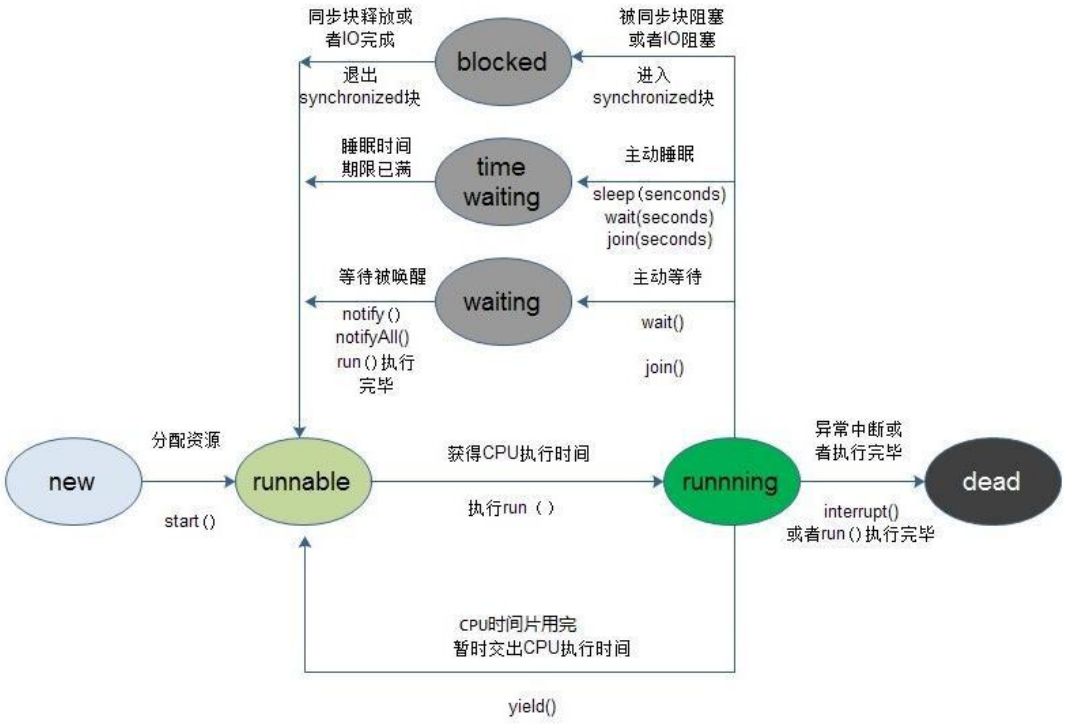
第三是运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行 run 函数当中的代码。

第四是阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个事件的发生(比如说某项资源就绪)之后再继续运行。sleep,wait 等方法都可以导致线程阻塞。

第五是死亡状态。如果一个线程的 run 方法执行结束或者异常中断后，该线程就会死亡。对于已经死亡的线程，无法再使用 start 方法令其进入就绪。

版本 2.

一般来说，线程包括以下这几个状态：创建(new)、就绪(runnable)、运行(running)、阻塞(blocked)、timed_waiting、waiting、消亡（dead）。



18.有三个线程T1，T2，T3，怎么确保它们按顺序执行？

join()方法。

19.在一个主线程中，要求有大量子线程执行完之后，主线程才执行完成。多种方式，考虑效率。

1.在主函数中使用 join()方法。

```
        t1.start();
        t2.start();
        t3.start();
        t1.join();//不会导致 t1 和 t2 和 t3 的顺序执行
        t2.join();
        t3.join();
System.out.println("Main finished");
```

2.CountDownLatch，一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

```
public class WithLatch {

    public static void main(String[] args)
        { CountDownLatch latch = new
        CountDownLatch(3); for (int i = 0; i < 3; i++)
        {
            new ChildThead(i, latch).start();
        }
        try
            { latch.await()
            ;
        } catch (InterruptedException e)
            { e.printStackTrace();
        }
        System.out.println("Main finished");
    }

    static class ChildThead extends Thread {
        private int id = -1;
        private CountDownLatch latch = null;

        public ChildThead(int id, CountDownLatch latch) {
            this.id = id;
            this.latch = latch;
        }
```

```
public void run() {
```

```
            try
                { Thread.sleep(Math.abs(n
                ew
Random().nextInt(5000)));
                System.out.println(String.format("Child Thread %d
finished", id));
            } catch (InterruptedException e)
                { e.printStackTrace();
            } finally
                { latch.countDown()
                ;
            }
        }
    }
}
```

3.使用线程池。

```
public class WithExecutor {
    public static void main(String[] args) throws
InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool(3);
        List<Callable<Void>> list = new
ArrayList<Callable<Void>>();
        for (int i = 0; i < 3; i++)
            { list.add(new
            ChildThead(i));
        }
        try
            { pool.invokeAll(list)
            ;
        } finally
            { pool.shutdown()
            ;
        }
        System.out.println("Main finished");
    }

    static class ChildThead implements Callable<Void> {
        private int id = -1;

        public ChildThead(int id) {
            this.id = id;
        }
```

```
        public Void call() throws Exception {
            try
                { Thread.sleep(Math.abs(n
                ew
Random().nextInt(5000)));
```

```
            System.out.println(String.format("Child Thread %d
finished", id));
         } catch (InterruptedException e)
            { e.printStackTrace();
         }
         return null;
      }
   }
}
```

20.java 程序如何停止一个线程？

建议使用”异常法”来终止线程的继续运行。在想要被中断执行的线程中，调用 interrupted()方法，该方法用来检验当前线程是否已经被中断，即该线程是否被打上了中断的标记，并不会使得线程立即停止运行，如果返回 true，则抛出异常，停止线程的运行。在线程外，调用 interrupt()方法,使得该线程打上中断的标记。

其他问题

ThreadLocal 的原理。

ThreadLocal 相当于一个容器， 用于存放每个线程的局部变量。ThreadLocal 实例通常来说都是 private static 类型的。 ThreadLocal 可以给一个初始值，而每个线程都会获得这个初始化值的一个副本，这样才能保证不同的线程都有一份拷贝。

一般情况下，通过 ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是访问不到的，各个线程中访问的是不同的对象。如果 ThreadLocal.set()进去的东西本来就是多个线程共享的同一个对象，那么多个线程的 ThreadLocal.get()取得的还是这个共享对象本身，还是有并发访问问题。

向 ThreadLocal 中 set 的变量是由 Thread 线程对象自身保存的，当用户调用 ThreadLocal 对象的 set(Object o) 时，该方法则通过 Thread.currentThread() 获取当前线程，将变量存入线程中的 ThreadLocalMap 类的对象内，Map 中元素的键为当前的 threadlocal 对象，而值对应线程的变量副本。

```
public T get() {
    Thread t = Thread.currentThread();  //每个 Thread 对象内都保存
```

一个 ThreadLocalMap 对象。

```
    ThreadLocalMap map = getMap(t); //map 中元素的键为共用的
threadlocal 对象，而值为对应线程的变量副本。
    if (map != null)
        return (T)map.get(this);

    T value = initialValue();
    createMap(t, value);
    return value;
}


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

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

void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

Collections.synchronizedXX 方法的原理。

```
        public E get(int index) {
            synchronized (mutex) {return list.get(index);}
        }
        public E set(int index, E element) {
            synchronized (mutex) {return list.set(index,
element);}
        }
        public void add(int index, E element) {
```

```
            synchronized (mutex) {list.add(index, element);}
        }
        public E remove(int index) {
            synchronized (mutex) {return list.remove(index);}
        }
```

在返回的列表上进行迭代时，用户必须手工在返回的列表上进行同步：

```
List list = Collections.synchronizedList(new ArrayList());
   ...
   synchronized(list) {
   Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
     foo(i.next());
}
```

如何在两个线程间共享数据？

一、每个线程执行的代码相同

若每个线程执行的代码相同，共享数据就比较方便。可以使用同一个 Runnable 对象，这个 Runnable 对象中就有那个共享数据。

```
public class MultiThreadShareData1 {
   public static void main(String[] args)
      {    SaleTickets    sale    =    new
      SaleTickets();                  new
      Thread(sale).start();
      new Thread(sale).start();
   }
}

class SaleTickets implements Runnable {
   public int allTicketCount = 20;

   public void run() {
      while (allTicketCount > 0)
         { sale();
      }
   }

   public synchronized void sale() {
      System.out.println("剩下" + allTicketCount);
      allTicketCount--;
```

```
    }
}
```

二、每个线程执行的代码不相同

如果每个线程执行的代码不同，这时候需要用不同的 **Runnable** 对象，将需要共享的数据封装成一个对象，将该对象传给执行不同代码的 **Runnable** 对象。