**1. 并发问题**

**1.1减少上下文切换的措施**

1.无锁并发编程。多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以用一

些办法来避免使用锁，如将数据的ID按照Hash算法取模分段，不同的线程处理不同段的数据。

2.CAS算法。Java的Atomic包使用CAS算法来更新数据，而不需要加锁。

3.使用最少线程。避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这

样会造成大量线程都处于等待状态。

4.协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

**1.2避免死锁的措施**

1.避免一个线程同时获取多个锁。

2.避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。

3.尝试使用定时锁，使用lock.tryLock（timeout）来替代使用内部锁机制。

4.对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。

2. **volatile与synchronize**

**2.1 volatile**

特性：轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”， 不会引起线程上下文的切换和调度。

对volatile进行写操作时，会产生lock前缀的汇编指令

**2.1.1 volatile的实现原则**

1. Lock前缀指令会引起处理器缓存回写到内存。
2. 一个处理器的缓存回写到内存会导致其他处理器的缓存无效。

**2.1.2 总线锁定与缓存锁定（处理器如何保证原子写）**

1. 总线锁定 在多处理器环境中，LOCK#信号确保在声言该信号期间，处理器可以

独占任何共享内存。当一个处理器在总线上输此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。

2. 缓存锁定 如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反，它会锁定这内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性

**2.1.3 缓存一致性协议（MESI）**

缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据。

如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，这个变量所在缓存行的数据写回到系统内存。在多处理器下，为了保证各个处理器的缓存一致，每个处理器通过嗅探在总线上传播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，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

volatile的使用优化（LinkedTransferQueue\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*）

**2.2 synchronized**

**2.2.1 表现形式**

1.对于普通同步方法，锁是当前实例对象。

2.对于静态同步方法，锁是当前类的Class对象。

3.对于同步方法块，锁是Synchonized括号里配置的对象。

**2.2.2 原理**

基于进入和退出Monitor对象来实现方法同步和代码块同步，代码块同步是使用monitorenter和monitorexit指令实现的，方法同步使用另外一种方式。

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

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。

**2.2.3 对象头**

synchronized用的锁是存在Java对象头里的，对象头里的Mark Word里默认存储对象的HashCode、分代年龄和锁标记位，在运行期间，Mark Word里存储的数据会随着锁标志位的变化而变化。

Java SE 1.6为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”，在Java SE 1.6中，锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，目的是为了提高获得锁和释放锁的效率。

* + 1. **偏向锁、轻量级锁和重量级锁的优缺点及应用场景**



* + 1. **Java如何实现原子操作**

1. 使用循环CAS实现原子操作；

CAS使用中的问题

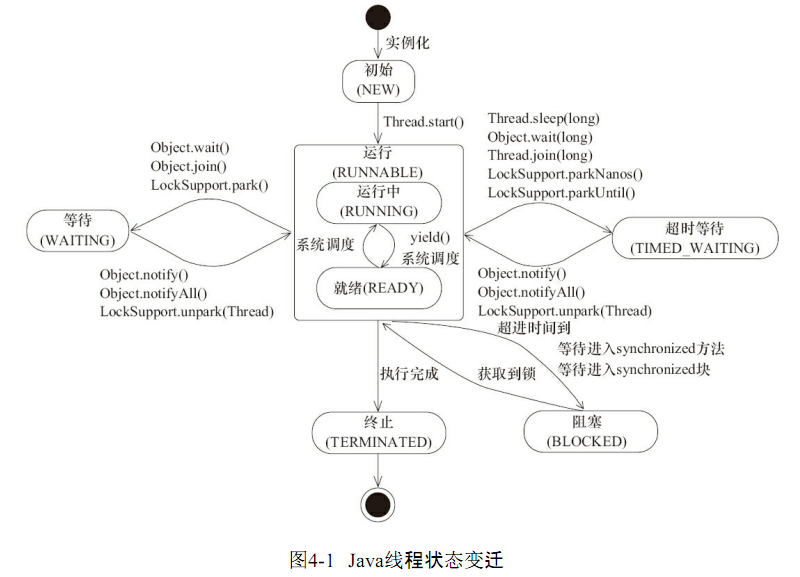
1).ABA问题；

2).循环时间长开销大；

3).只能保证一个共享变量的原子操作

2.使用锁机制实现原子操作。

**线程状态**



**阻塞**状态是线程阻塞在进入synchronized关键字修饰的方法或代码块（获取锁）时的状态，但是阻塞在java.concurrent包中Lock接口的线程状态却是等待状态，因为java.concurrent包中Lock接口对于阻塞的实现均使用了LockSupport类中的相关方法。

**Daemon线程**

主要被用作程序中后台调度以及支持性工作，，当一个Java虚拟机中不存在非Daemon线程的时候，Java虚拟机将会退出。

setDaemon(true)需要在启动线程之前设置，不能在启动线程之后设置（Daemon线程中的finally块并不一定会执行）。

**中断**

**方法**

boolean isInterrupted() 是否已中断；

void interrupt() 中断线程；

boolean interrupted() 返回并清除中断状态，如果连续两次调用，第二次将返回false（当前线程再次中断除外）。

**线程不同状态下对中断（interrupt()）的响应**

NEW和TERMINATED：无响应，无意义；

RUNNABLE：设置该线程的中断标志位，不会让线程实际中断，需要用程序去判断；

BLOCKED：设置该线程的中断标志位，不会让线程实际中断，需要用程序去判断；

WAITING/TIMED\_WAITING：终止线程，清除标志位并抛出InterruptedException异常。

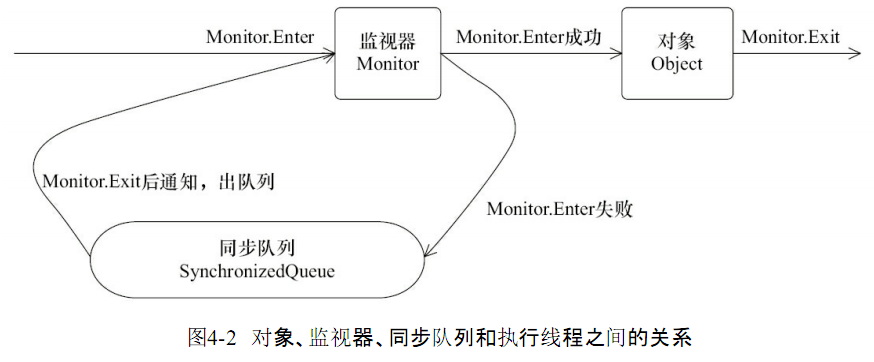
**安全终止线程的方法：**

使用中断，或在循环中使用boolean变量终止。

**线程间通讯**

**synchronized实现细节**

任意对象都拥有自己的监视器，当该对象由同步块或同步方法调用时，要先获取该对象的监视器才能进入同步块或同步方法；如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问该对象的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。



**等待/通知机制**

**wait、notify和notifyAll用法**

1）使用wait()、notify()和notifyAll()时需要先对调用对象加锁。

2）调用wait()方法后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的

等待队列。

3）notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或

notifAll()的线程释放锁之后，等待线程才有机会从wait()返回。

4）notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll()

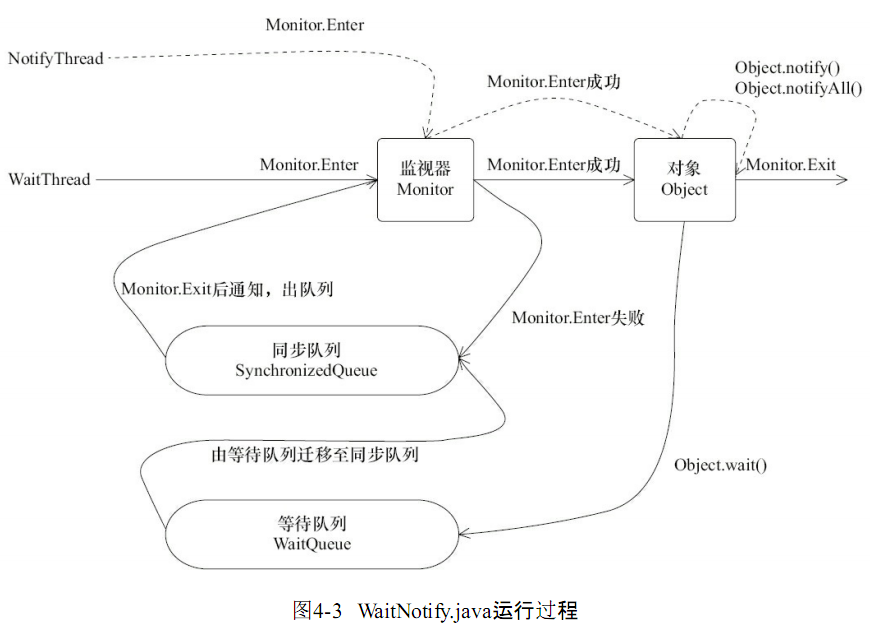
方法则是将等待队列中所有的线程全部移到同步队列，被移动的线程状态由WAITING变为

BLOCKED。

5）从wait()方法返回的前提是获得了调用对象的锁。

从上述细节中可以看到，等待/通知机制依托于同步机制，其目的就是确保等待线程从

wait()方法返回时能够感知到通知线程对变量做出的修改。



**等待/通知（生产者与消费者）范式**

通知方

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。

synchronized(对象) {

改变条件

对象.notifyAll();

}

等待方

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

synchronized(对象) {

while(条件不满足) {

对象.wait();

}

对应的处理逻辑

}

**join()方法**

线程A执行了thread.join()语句，其含义是：当前线程A等待thread线程终止之后才

从thread.join()返回。

join(long millis)和join(longmillis,int nanos)：

表示如果线程thread在给定的超时时间里没有终止，那么将会从该超时方法中返回。

源码（即使用等待通知范式）：

// 加锁当前线程对象

public final synchronized void join() throws InterruptedException {

// 条件不满足，继续等待

while (isAlive()) {

wait(0);

}

// 条件符合，方法返回

}

**ThreadLocal线程变量**

以ThreadLocal对象为键、任意对象为值的存储结构。该结构被附带在线程上，一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。

可以通过set(T)方法来设置一个值，在当前线程下再通过get()方法获取到原先设置的值。

应用：

在AOP中，可以在方法调用前的切入点执行begin()方法，而在方法调用后的切入点执行

end()方法，用以统计方法的执行耗时。

超时等待范式（在等待/通知范式基础上增加超时控制）

// 对当前对象加锁（等待方方法）

public synchronized Object get(long mills) throws InterruptedException {

long future = System.currentTimeMillis() + mills;

long remaining = mills;

// 当超时大于0并且result返回值不满足要求

while ((result == null) && remaining > 0) {

wait(remaining);

remaining = future - System.currentTimeMillis();

}

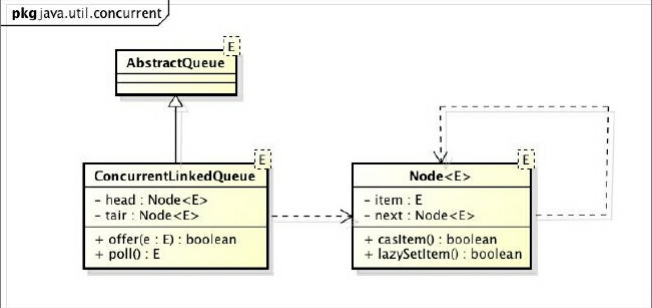
return result;

}

**并发容器与框架**

1. **ConcurrentLinkedQueue** 非阻塞线程安全的队列。

1.1定义：基于链接节点的无界线程安全队列，采用先进先出的规则对节点进行排序，当添加一个元素的时候，会添加到队列的尾部；当获取一个元素时，会返回队列头部的元素。



1.2入队列

1.3出队列

1.4使用场景

1. **阻塞队列**

2.1特点与公共方法

支持两个附加的操作，阻塞插入和阻塞移除。

支持阻塞插入：当队列满时，队列会阻塞插入元素的线程，直到队列不满。

支持阻塞移除：在队列为空时，获取元素的线程会等待队列变为非空。



1）抛出异常：队列满时，再添加数据会抛出IllegalStateException（"Queue

full"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。

2）返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。

3）一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。

4）超时退出：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

\* 如果是无界阻塞队列，队列不可能会出现满的情况，所以使用put或offer方法永远不会被阻塞，而且使用offer方法时，该方法永远返回true。

2.2七种阻塞队列

2.2.1 ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

按照先进先出（FIFO）的原则对元素进行排序。

默认情况下不保证线程公平的访问队列。若要保证公平性构造时使用

new ArrayBlockingQueue(1000,true);

访问者的公平性是使用可重入锁实现

* + 1. LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

默认和最大长度为Integer.MAX\_V ALUE。此队列按照先进先出的原则对元素进行排序。

* + 1. PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

默认情况下元素采取自然顺序升序排列。也可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

* + 1. DelayQueue：一个使用优先级队列实现的无界阻塞队列。

支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。

只有在延迟期满时才能从队列中提取元素。

* + 1. SynchronousQueue：一个不存储元素的阻塞队列。

不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。

支持公平访问队列。默认情况下线程采用非公平性策略访问队列。使用该构造方法可以创建公平性访问：SynchronousQueue(true)，如果设置为true，则等待的线程会采用先进先出的顺序访问队列。

适合负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

* + 1. LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
    2. LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

可以

从队列的两端插入和移出元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。

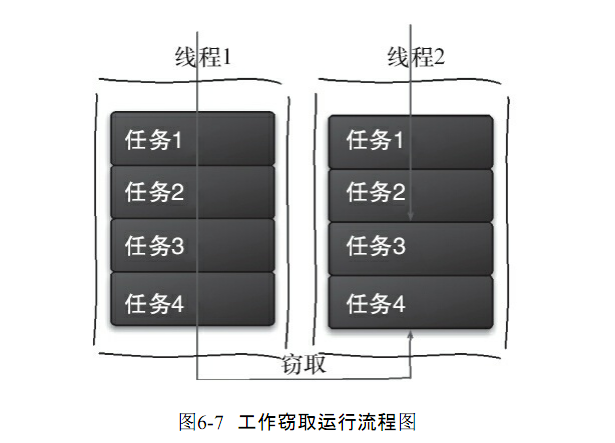
在初始化LinkedBlockingDeque时可以设置容量防止其过度膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中（整合Fork/Join）。

* 1. 原理

**3. Fork/Join框架**

定义：把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果。

工作窃取算法：



子任务线程1执行完成队列中的任务后，获取线程2队尾的任务来执行。

优点：充分利用线程进行并行计算，减少了线程间的竞争。

缺点：在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且该算法会消耗了更多的系统资源，比如创建多个线程和多个双端队列。