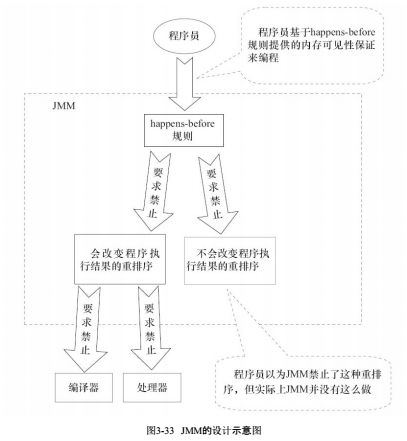
第三章

3.7 happens-before

3.7.1。JMM的设计

程序员希望有一个强内存模型来保证内存的可见性，而编译器和处理器则希望有一个若内存模型，从而可以尽可能多地来做优化，所以需要从中取个平衡点。

JMM把happens-before要求禁止的重排序分为了会改变程序结果的重排序和不会改变结果的重排序。



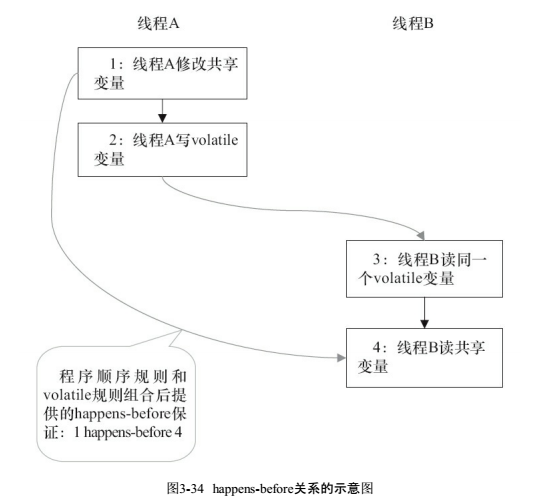
从图上我们可知JMM向程序员提供的happens-before规则能够满足程序员的需求,并且尽可能少地减少对编译器和处理器的束缚。

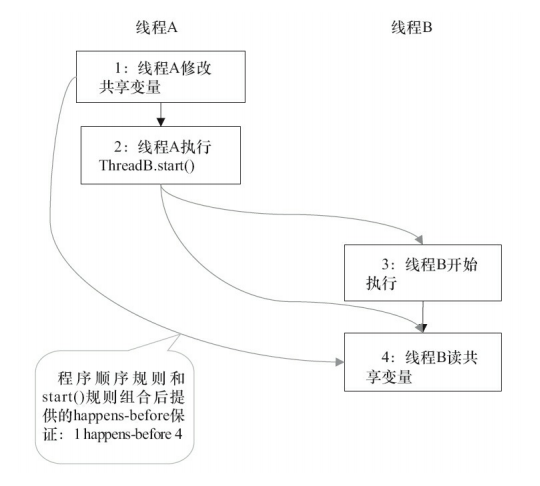
3.7.2 happens-before的定义

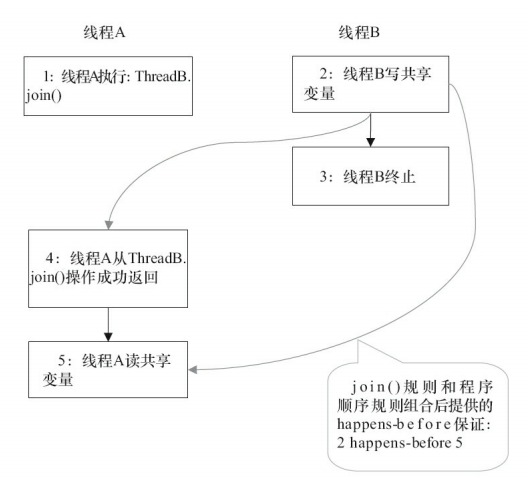
1. 如果一个操作happens-before另一个操作，那么这第一个操作的执行结果将对的第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前，这是JMM对程序员的承诺。
2. 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行，如果重排序之后的执行结果与按照happens-before关系执行的结果一至那么允许这种重排序，这是JMM对编译器和处理器重排序的约束原则。
3. as-if-serial保证单线程内程序的执行结果不会被改变。
4. Happens-before保证正确同步的多线程程序的执行结果不会别改变。

3.7.3 happens-before规则

1. 程序顺序规则
2. 监视器锁规则
3. Volatile变量规则
4. 传递性
5. Start()规则
6. Join()规则







3.8双重检查锁定与延迟初始化

3.8.1 有时我们想要对高开销的对象在使用它时再初始化，所以有了延迟初始化。

下面来看一个线程不安全的延迟初始化代码

public class UnsafeLazyInitialization {  
 private static Instance instance;  
 public static Instance getInstance() {  
 if (instance == null) // 1：A线程执行  
 instance = new Instance(); // 2：B线程执行  
 return instance;  
 }  
}

再看一个线程安全的代码

public class SafeLazyInitialization {  
 private static Instance instance;  
 public synchronized static Instance getInstance() {  
 if (instance == null) // 1：A线程执行  
 instance = new Instance(); // 2：B线程执行  
 return instance;  
 }  
}

但是这种方法在频繁调用的时候性能会下降。

于是就又有了双检测锁定方法来降低同步开销。

public class DoubleCheckedLocking { // 1

private static Instance instance; // 2

public static Instance getInstance() { // 3

if (instance == null) { // 4:第一次检查

synchronized (DoubleCheckedLocking.class) { // 5:加锁

if (instance == null) // 6:第二次检查

instance = new Instance(); // 7:问题的根源出在这里

} // 8

} // 9

return instance; // 10

} // 11

}

多个线程试图在同一时间创建对象时，会通过加锁来保证只有一个线程能创建对象。  
 在对象创建好之后，执行getInstance()方法将不需要获取锁，直接返回已创建好的对象。

3.8.2 问题根源

创建一个对象要经过以下几个步骤：

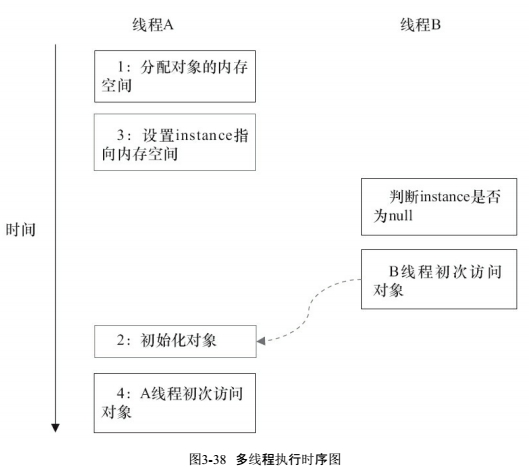
memory = allocate(); // 1：分配对象的内存空间

ctorInstance(memory); // 2：初始化对象

instance = memory; // 3：设置instance指向刚分配的内存地址

在某些编译器上2，3会重排序，遵守了intra-thread规则，保证重排序不会改变单线程内程序的执行结果。

所以会出现下面这种情况：



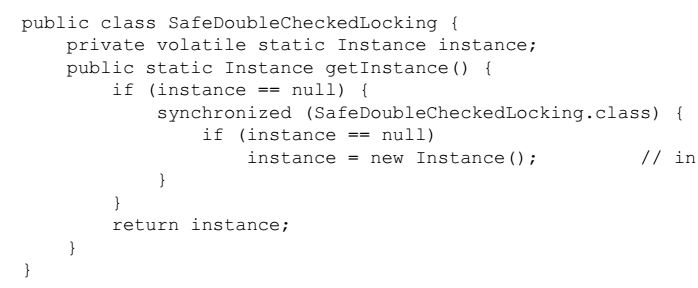
B线程将会看到一个没有被初始化的对象。

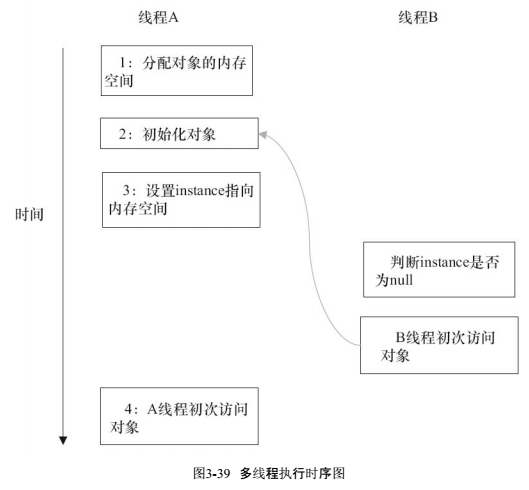
解决方案：

1）不允许2和3重排序。

2）允许2和3重排序，但不允许其他线程“看到”这个重排序。

3.8.3 基于volatile的解决方案





3.8.4 基于类初始化的解决方案：

在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

public class InstanceFactory {

private static class InstanceHolder {

public static Instance instance = new Instance();

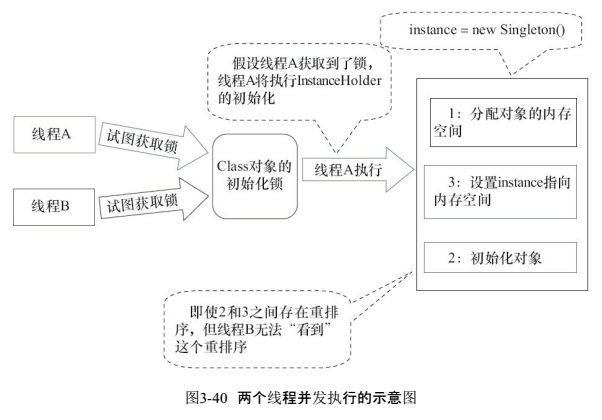
}p

ublic static Instance getInstance() {

return InstanceHolder.instance ; // 这里将导致InstanceHolder类被初始化

}

}



初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。根

据Java语言规范，在首次发生下列任意一种情况时，一个类或接口类型T将被立即初始化。

1）T是一个类，而且一个T类型的实例被创建。

2）T是一个类，且T中声明的一个静态方法被调用。

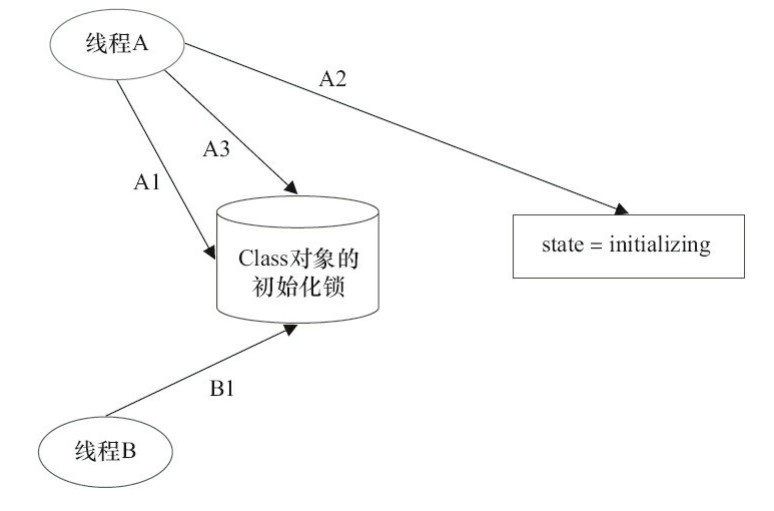
3）T中声明的一个静态字段被赋值。

4）T中声明的一个静态字段被使用，而且这个字段不是一个常量字段。

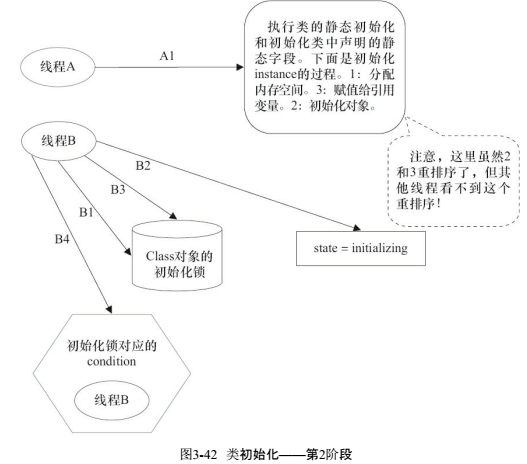
5）T是一个顶级类（Top Level Class，见Java语言规范的§7.6），而且一个断言语句嵌套在T内部被执行。

类初始化的过程：

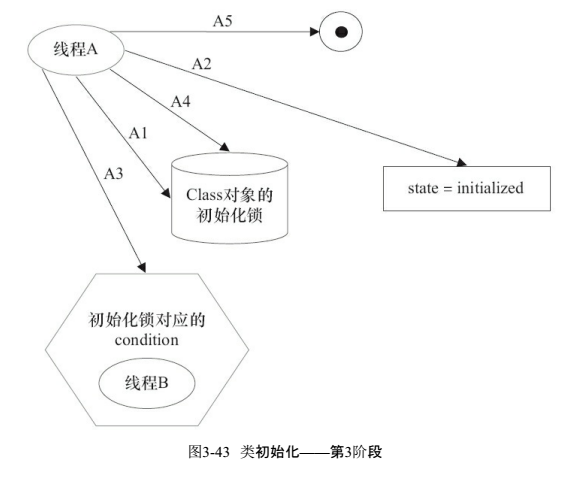
第1阶段：通过在Class对象上同步（即获取Class对象的初始化锁），来控制类或接口的初始化。这个获取锁的线程会一直等待，直到当前线程能够获取到这个初始化锁。



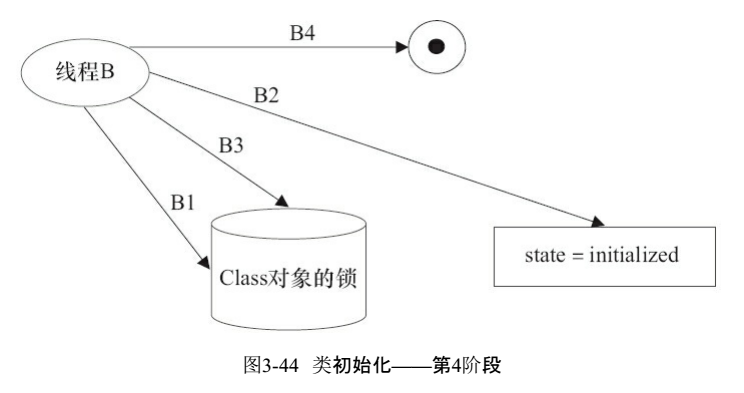
第2阶段：线程A执行类的初始化，同时线程B在初始化锁对应的condition上等待。

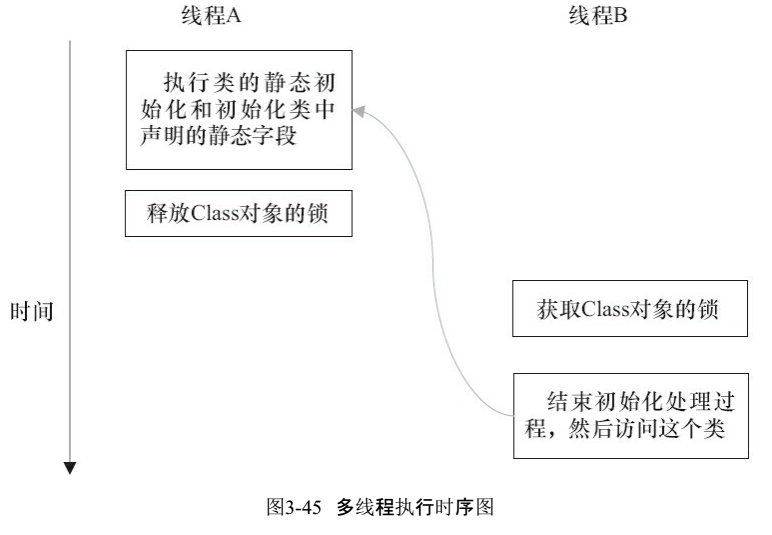


第3阶段：线程A设置state=initialized，然后唤醒在condition中等待的所有线程。

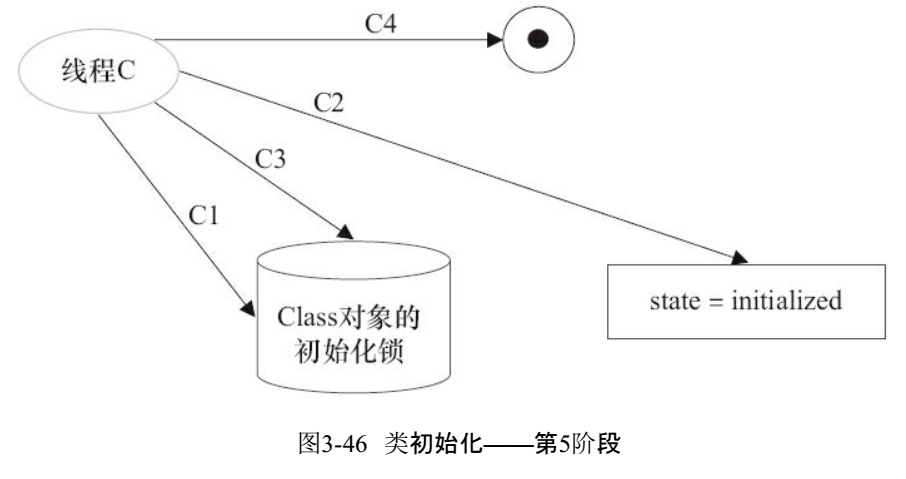


第4阶段：线程B结束类的初始化处理。





第5阶段：线程C执行类的初始化的处理。



这个happens-before关系将保证：线程A执行类的初始化时的写入操作，线程C一定能看。

如果确实需要对实例字段使用线程安全的延迟初始化，请使用上面介绍的基于volatile的延迟初始化的方案；如果确实需要对静态字段使用线程安全的延迟初始化，请使用上面介绍的基于类初始化的方案。

第四章

Java并发编程基础

4.1 线程简介

4.1.1 什么是线程

现代操作系统在运行一个程序时，会为其创建一个进程。线程作为操作系统调度的最小单元，多个线程能够同时执行，这将显著提升程序性能，在多核环境中表现得更加明显。在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

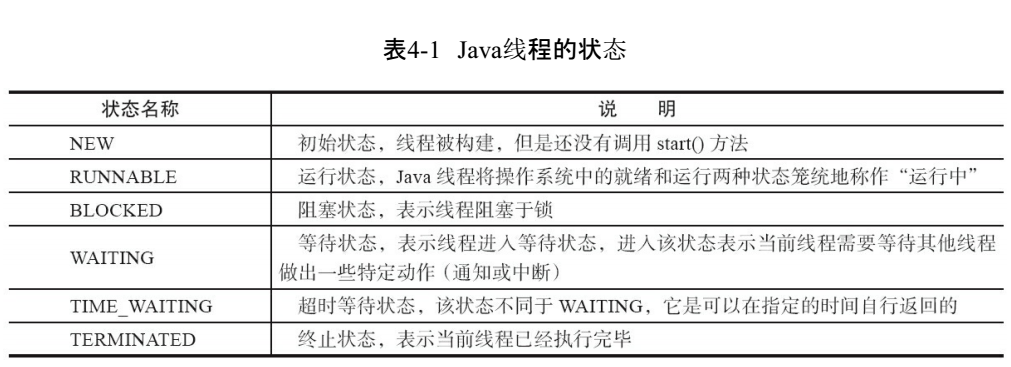
4.1.2 为什么要使用线程

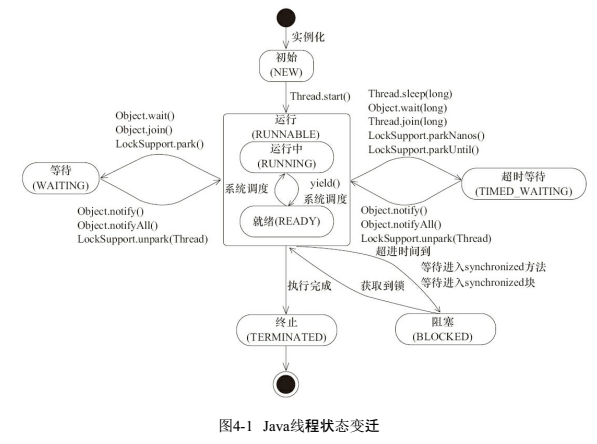
更多的处理器核心、更快的响应时间、更好的编程模型

4.1.3 线程的优先级

现代操作系统基本采用时分的形式调度运行的线程，操作系统会分出一个个时间片，线程会分配到若干时间片，当线程的时间片用完了就会发生线程调度，并等待着下次分配。而线程优先级就是决定线程需要多或者少分配一些处理器资源的线程属性。

4.1.4 线程的状态





注： 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。

4.1.5 Daemon后台线程

Daemon属性需要在启动线程之前设置，不能在启动线程之后设置。，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。在构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

4.2 启动和终止线程

4.2.1 构造线程

由父线程完成

4.2.2 启动线程线程对象在初始化完成之后，调用start()方法就可以启动这个线程。当前线程（即parent线程）同步告知Java虚拟机，只要线程规划器空闲，应立即启动调用start()方法的线程。

4.2.3 理解中断

中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。Java虚拟机会先将该线程的中断标识位清除，然后抛出InterruptedException，此时调用isInterrupted()方法将会返回false。

4.2.4 过期的suspend()、resume()、stop()

以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。

4.2.5 安全地终止线程

public class Shutdown {

public static void main(String[] args) throws Exception {

Runner one = new Runner();

Thread countThread = new Thread(one, "CountThread");

countThread.start();

// 睡眠1秒，main线程对CountThread进行中断，使CountThread能够感知中断而结束

TimeUnit.SECONDS.sleep(1);

countThread.interrupt();

Runner two = new Runner();

countThread = new Thread(two, "CountThread");

countThread.start();

// 睡眠1秒，main线程对Runner two进行取消，使CountThread能够感知on为false而结束

TimeUnit.SECONDS.sleep(1);

two.cancel();

}

private static class Runner implements Runnable {

private long i;

private volatile boolean on = true;

@Override

public void run() {

while (on && !Thread.currentThread().isInterrupted()){

i++;

}

System.out.println("Count i = " + i);

}

public void cancel() {

on = false;

}

}

}

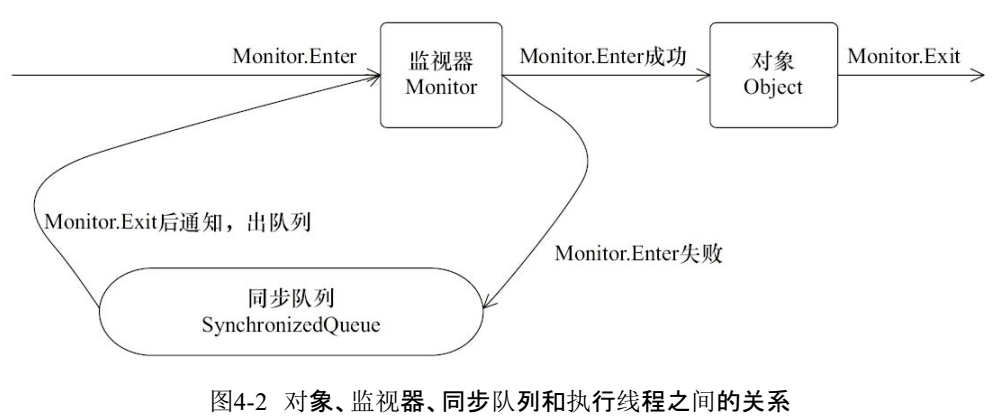
这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

4.3 线程间通信

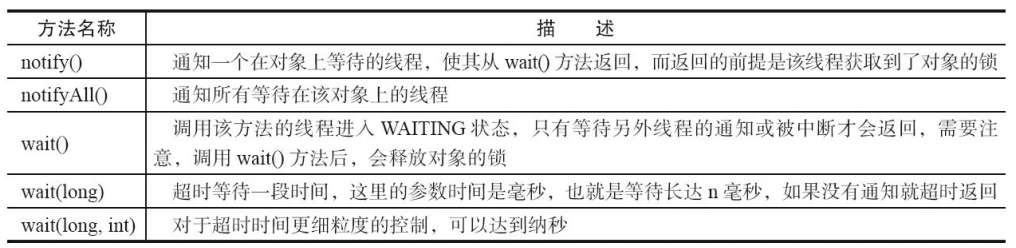
4.3.1 volatile和synchronized关键字

Java支持多个线程同时访问一个对象或者对象的成员变量。关键字volatile可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法，而没有获取到监视器（执行该方法）的线程将会被阻塞在同步块和同步方法的入口处，进入BLOCKED状态。



4.3.2 等待/通知机制



等待/通知机制，是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。

注意： 获取lock的锁，然后进行通知，通知时不会释放lock的锁，直到当前线程释放了lock后，WaitThread才能从wait方法中返回。

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。

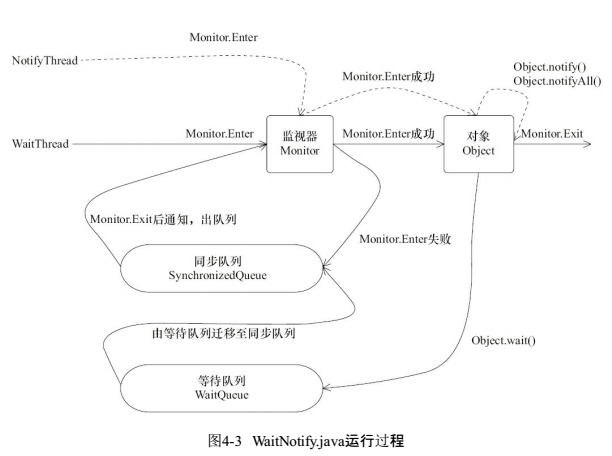
4）notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll()

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。

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。



4.3.3 从等待/通知机制中可以提炼出生产者和消费者模式

等待方遵循如下原则

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

}

通知方遵循如下原则。

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();

}

4.3.4 管道输入/输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

对于Piped类型的流，必须先要进行绑定，也就是调用connect()方法，如果没有将输入/输出流绑定起来，对于该流的访问将会抛出异常。