目录

[**第一部分 多线程编程基础** 2](#_Toc52206508)

[**第1章 走进Java世界中的线程** 2](#_Toc52206509)

[**1.1 进程、线程与任务** 2](#_Toc52206510)

[**1.3 Java线程API简介** 2](#_Toc52206511)

[**1.5 线程的层次关系** 3](#_Toc52206512)

[**1.6 线程的生命周期状态** 3](#_Toc52206513)

[**1.9 多线程编程的优势和风险** 4](#_Toc52206514)

[**第2章 多线程编程的目标与挑战** 5](#_Toc52206515)

[**2.1 串行、并发与并行** 5](#_Toc52206516)

[**2.2 竞态** 5](#_Toc52206517)

[**2.3 线程安全性** 5](#_Toc52206518)

[**2.4 原子性** 6](#_Toc52206519)

[**2.5 可见性** 6](#_Toc52206520)

[**2.6 有序性** 8](#_Toc52206521)

[**2.7 上下文切换** 11](#_Toc52206522)

[**2.8 线程的活性故障** 13](#_Toc52206523)

[**2.9 资源争用与调度** 13](#_Toc52206524)

**第一部分 多线程编程基础**

**第1章 走进Java世界中的线程**

**1.1 进程、线程与任务**

进程是程序的运行实例，运行中的一个Java程序实质上就是启动了一个Java虚拟机进程。进程是程序向操作系统申请资源（内存空间和文件句柄）的基本单位。

线程是进程中可独立执行的最小单位。一个进程可以 包含多个线程，同一个进程中的所有线程共享该进程中的资源（内存空间、文件句柄）。Java虚拟机启动的时候会创建一个main线程来执行Java程序的入口方法main方法。Java虚拟机中的垃圾回收线程是由Java虚拟机自行创建的。Java虚拟机中的JIT即时编译器会动态地将Java字节码编译为Java虚拟机宿主机处理器可以直接执行的机器码，这个动态编译的过程是由Java虚拟机创建的专门的线程来执行的。

任务是线程所要完成的计算/工作。

**1.3 Java线程API简介**

**1.3.1 线程创建、启动与运行**

线程的任务处理逻辑可以再Thread类的run实例方法中直接实现或者通过该方法调用，run方法相当于线程的任务处理逻辑入口，由Java虚拟机直接调用。

运行一个线程就是让Java虚拟机运行线程的run方法，前提是需要先启动线程。Thread类的start方法的作用就是启动线程，启动线程的实质是请求Java虚拟机运行相应的线程，而线程何时才能运行由线程调度器决定。

start方法只能调用一次，多次调用同一个Thread的start方法会抛出IllegalThreadStateException异常。

在Java平台，一个线程Thread就是一个对象，对象的创建离不开内存空间的分配。创建一个线程与创建其它类型的Java对象所不同的是，除了堆内存空间分配，Java虚拟机还会为每个线程分配调用栈所需的内存空间。调用栈用于跟踪Java方法间的调用关系以及Java本地方法（Native Code）的调用。创建线程对象比创建其它类型的对象的成本要更高一些。

**1.3.2 Runnable接口**

Runnable接口只定义了一个run方法，可以被看作对任务进行的抽象，任务的处理逻辑就体现在run方法之中。Thread类实际上是Runnable接口的一个实现类。Thread中有target来引用通过Thread(Runnable target)构造方法传入的Runnable实现类对象，如果target不为null，则Thread的run方法就调用target.run()方法，否则什么也不做。因此创建线程有两种方法：一个是重写Thread的run方法，在其中直接实现任务处理逻辑（继承）；另一个是在Runnable实例中实现任务处理逻辑，并由Thread的run方法调用（组合）。

通过组合的方式创建线程比起继承，类与类之间的耦合性更低，更加灵活。但是组合的方式意味着多个线程实例可以共享同一个Runnable实例，可能会产生竞态并最终导致线程安全问题。

**1.3.3 线程属性**

线程属性：线程编号ID、名称Name、线程类别Daemon、优先级Priority。除了编号之外，其它属性都是可Get/Set的属性。其中setDaemon方法的调用必须在start调用之前，否则抛出IllegalThreadStateException异常。

Java线程的优先级属性本质上只是一个给线程调度器的提示信息，以便于线程调度器优先调度哪些线程运行，它并不能保证线程按照其优先级高低的顺序运行。Java线程的优先级使用不当可能导致某些线程永远得不到运行（线程饥饿）。

按照线程是否会阻止Java虚拟机正常停止，将Java中的线程分为守护线程和用户线程。用户线程会阻止Java虚拟机的正常停止，Java虚拟机只有在所有用户线程都运行结束的情况下才能正常停止。而守护线程则不会影响Java虚拟机的正常停止。守护线程通常用于执行一些重要性不很高的任务，如用于监视其它线程的运行情况。

**1.3.4 Thread类的常用方法**

void join()：若线程A调用线程B的join方法，则线程A的运行会被暂停，知道线程B运行结束。

static void yield()：使但概念线程主动放弃其对处理器的占用，可能导致当前线程被暂停。这个方法是不可靠的，该方法被调用时当前线程可能仍然继续运行。

**1.5 线程的层次关系**

若线程A创建了线程B，则称线程B为线程A的子线程，线程A为线程B的父线程。

一个线程是否是守护线程默认取决于其父线程。

一个线程的优先级默认与该线程的父线程相同。

**1.6 线程的生命周期状态**

NEW：已创建而为启动的线程处于该状态。

RUNNABLE：包括两个子状态：READY和RUNNING。前者表示线程可以被线程调度器进行调度而使之处于RUNNING状态。后者表示线程正在运行，即run方法的指令正在由处理器执行。执行Thread.yield()方法的线程其状态可能会由RUNNING转换为READY。处于READY子状态的线程也被称为活跃线程。

BLOCKED：线程发起了阻塞式的操作（阻塞式的IO操作），或者申请其它线程持有的独占资源（如锁）时，线程处于该状态。处于BLOCKED状态的线程并不会占用处理器资源。当阻塞式的操作或者获得了申请的资源时可转换为RUNNABLE状态。

WAITING：一个线程执行了某些特定方法（Object.wait()、Thread.join()、LockSupport.park(Object)）之后就会处于等待其它线程执行另外一些特定操作（Object.notify()/notifyAll()、LockSupport.unpark(Object)）的状态。

TIMED\_WAITING：类似于WAITING状态，但并非无限制地等待其它线程执行特定操作，而是处于带有时间限制的等待状态。当其它线程没有在指定时间内执行特定操作时，该线程自动转换为RUNNABLE状态。

TERMINATED：已经执行结束的线程，无论是run方法正常返回或是因抛出异常而提前终止，都会处于该状态。

一个线程的整个生命周期中只能有一次处于NEW和TERMIANTED状态。

**1.9 多线程编程的优势和风险**

优势：

* 提高系统的吞吐率。一个进程内可以有多个并发操作，一个线程等待时其它线程仍然可以执行其它操作。
* 提高响应性。某些处理时间较长执行较慢的线程并不会影响其它线程的任务处理。
* 充分利用多核处理器资源。
* 最小化系统资源的使用。一个进程中的多个线程可以共享其所在进程的资源，相比于多进程节约了对系统资源的使用。
* 简化复杂程序的结构。

问题：

* 线程安全问题。多个线程共享数据时，如果没有并发访问控制，可能会产生数据一致性问题（读取脏数据、丢失更新）。
* 线程活性问题。死锁、锁死（线程执行需要满足某种条件，但是这个条件却永远无法满足）、活锁（线程一直在尝试某个操作但始终没有进展）、线程饥饿（线程永远无法获取处理器执行的机会而永远处于RUNNABLE的子状态READY）
* 上下文切换。处理器从执行一个线程转向执行另一个线程的过程就称为上下文切换。
* 可靠性。多线程编程一方面有利于可靠性，例如某个线程意外终止并不影响其他线程的执行。另一方面，线程是进程的一个组件，如果进程意外终止，例如Java虚拟机内存泄露导致进程终止，那么该进程内的所有线程也都将终止。

**第2章 多线程编程的目标与挑战**

**2.1 串行、并发与并行**

串行：任务依序执行，执行完一个任务之后才能开始执行下一个任务。

并发：**一段时间内**可以交替处理多个任务。

并行：一种更为严格、理想的并发，**一个时间点**可以同时执行多个任务。

从硬件角度来说，一个处理器可以使用时间片分配技术来实现一段时间内运行多个线程，一个处理器即可实现并发。而并行需要靠多个处理器在同一个时刻各自运行一个线程来实现。

多线程编程的实质就是将任务的处理方式由串行改为并发。

**2.2 竞态**

多线程的计算结果的正确性不唯一确定的现象就成为竞态。

**2.2.1 二维表分析法：解释竞态的结果**

状态变量：类的实例变量、静态变量

共享变量：可以被多个线程共同访问的变量。“共享”强调的是“可以被共享”的可能性，而并不表示该变量一定会被多个线程访问。

状态变量可以被多个线程共享。

导致竞态的常见原因是多个线程在没有采取任何并发控制措施的情况下并发地更新、读取共享变量。

通过二维表分析可以得出，竞态是指计算的正确性依赖于相对时间顺序或者线程的交错。竞态不一定就导致计算结果的不正确性。竞态往往伴随着读取脏数据、丢失更新的问题。

**2.2.2 竞态的模式与竞态产生的条件**

产生竞态的两种模式/操作：read-modify-write和check-then-act（if-else）。

局部变量（形式参数和方法体内定义的变量）的使用不会导致竞态。

**2.3 线程安全性**

如果一个类在单线程下能够正常运行，在多线程下，使用者不需要进行额外的改变或者操作下仍能正常运行，则称其是线程安全的，即这个类具有线程安全性。如果一个类能够导致竞态，那么它就是非线程安全的，否则就是线程安全的。一个线程安全的类在单线程下肯定能够正常运行。

线程安全问题概括来说表现在3个方面：原子性、可见性和有序性。

**2.4 原子性**

对于涉及共享变量访问的操作，若该操作从其执行线程以外的线程看来是不可分割的，那么该操作就是原子操作，称该操作具有原子性。

所谓“不可分割”，指读写某个共享变量的操作在其它线程看来，该操作要么已经执行结束，要么尚未发生，而不会看到该操作执行了部分的中间效果。

访问同一组共享变量的原子操作是不能够被交错的，一个线程执行一个操作期间其它线程不会读取或更新共享变量，否则会读取到脏数据或者丢失更新。因此，操作的原子性可以消除该操作导致竞态的可能性。

注意点：

* 原子操作是针对访问共享变量的操作而言的。涉及局部变量访问的操作是无所谓是否原子的，或者把这一类操作都看成原子操作。
* 原子操作是从该操作的执行线程以外的线程来描述的，它只有在多线程环境下才有意义。

Java中有两种方式来实现原子性：一种是使用锁。锁具有排他性，保障共享变量任意时刻只能被一个线程访问，排除多个线程同时访问同一个共享变量导致的干扰和冲突。另一种是利用处理器提供的CAS指令，在硬件（处理器和内存）层面上实现的原子操作。

在Java中，除了32位Java虚拟机中对long型和double型**变量的写操作**之外，其它任何类型的**变量的写操作**都是原子操作。但是，Java语言规范规定对于volatile关键字修饰的long/double**变量的写操作**具有原子性。volatile关键字只能保障**变量写操作**的原子性，并不能保障其它如read-modify-write和check-then-act等操作的原子性。

Java语言中针对任何变量的读操作都是原子操作。

从原子操作的不可分割的特性可知，是一个操作具有原子性就可以消除该操作导致竞态的可能性。可以通过将read-modify-write和check-then-act操作转换为原子操作来消除竞态。

**2.5 可见性**

可见性指一个线程对共享变量的更新对于读取该共享变量的线程是否可见的问题。多线程的可见性问题可能会导致线程读取到旧数据。

可见性问题与计算机的存储系统有关。程序中的变量可能会被分配到寄存器而不是主内存中进行存储。每个处理器都有其寄存器，而一个处理器无法读取到另一个处理器上的寄存器的内容。即便某个共享变量被分配到主内存中进行存储，由于处理器对主内存的访问并不是直接访问，而是通过其高速缓存子系统。一个处理器上运行的线程对变量的更新可能只是更新到该处理器的写缓冲器中，还没有到达该处理器的高速缓存，更不用说到主内存。而一个处理器的写缓冲器中的内容无法被另一个处理器读取，因此运行在另外一个处理器上的线程无法看到这个线程对某个共享变量的更新。即便一个处理器上运行的线程对共享变量的更新写入该处理器的高速缓存，由于该处理器将这个变量的更新结果通知给其它处理器的时候，其它处理器可能仅仅将更新通知的内容存入无效化队列而没有直接更新其高速缓存的相应内容，导致其它处理器上运行的其它线程读取相应共享变量时从其处理器的高速缓存中读取到的变量是一个过时的值。

处理器并不是直接访问主内存，而是通过寄存器、高速缓存、写缓冲器和无效化队列等执行主内存的读、写操作。这些中间部件相当于主内存的副本，统称为处理器对主内存的缓存，简称**处理器缓存**。

虽然一个处理器的高速缓存中的内容不能被另一个处理器直接读取，但是可以通过缓存一致性协议来读取其它处理器的高速缓存中的数据并更新到自己的高速缓存中，称之为**缓存同步**。可同步的存储部件包括处理器的高速缓存和主内存。为了保障可见性，必须使处理器对共享变量的更新最终被写入高速缓存或主内存中，而不是一直停留在写缓冲器中，该过程称为**冲刷处理器缓存**。并且，处理器在读取共享变量的时候，如果其它处理器在此之前已经更新了该变量，那么必须从其它处理器的高速缓存或者主内存中对相应的变量进行缓存同步，称为**刷新处理器缓存**。因此，可见性的保障是通过使更新共享变量的处理器执行冲刷处理器缓存的动作，并使读取共享变量的处理器执行刷新处理器缓存的动作来实现的。

volatile关键字所起到的作用是，提示JIT编译器被该关键字修饰的变量**可能**会被多线程共享，阻止JIT编译器做出可能会导致程序运行不正常的优化，另外就是读取volatile变量会使相应的处理器执行刷新处理器缓存的动作，写一个volatile变量会使相应的处理器执行冲刷处理器缓存的动作，从而保障了可见性。

对于一个共享变量，一个线程更新了该变量，其它线程能够读取到这个更新后的值，该新值就成为该变量的**相对新值**。如果线程在读取并使用该变量的时候其它线程无法更新该变量，则该线程读取到的值就成为**最新值**。

保障可见性仅仅意味者线程能够读取到共享变量的相对新值，而不是最新值。

扩展阅读：单处理器系统是否存在可见性问题？

可见性问题是多线程衍生出来的问题，与处理器的数目无关。单处理器系统中实现的多线程编程也可能出现可见性问题。多线程通过分配时间片的方式在同一处理器上并发运行时，发生上下文切换时，一个线程对寄存器变量的修改会被作为该线程的上下文保存起来，导致其它线程无法看到该线程对变量的修改。

扩展阅读：可见性与原子性的联系与区别

原子性保证一个线程对共享变量的更新在其它线程看来，肯定是变量的初始值**或者**更新之后的相对值，而不是更新中间的一种状态/半成品。而可见性保证一个线程对共享变量的更新对其它线程可见，即保证其它线程可以读取到共享变量的相对新值。因此，从保障线程安全的角度来看，必须同时保障原子性和可见性才能确保一个线程能够正确地看到其它线程对共享变量的**完整**的更新。

线程启动/停止与可见性：

Java语言规范保证，父线程在启动子线程之前对共享变量的更新对于子线程来说是可见的。

类似的，Java语言规范保证，一个线程终止后该线程对共享变量的更新对于调用该线程的join方法的线程而言是可见的。

**2.6 有序性**

**2.6.1 重排序的概念**

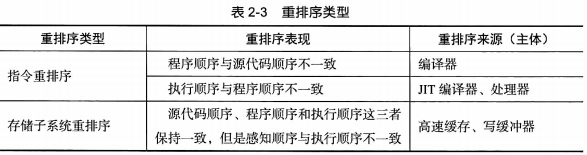
编译器可能改变两个操作的先后顺序；处理器可能不是完全按照程序的目标代码所指定的顺序执行指令；另外，一个处理器上执行的多个操作，从其它处理器的角度来看其顺序可能与目标代码所指定的顺序不一致。这种现象就叫重排序。

重排序是对内存访问操作（读/写）所作的一种优化，它可以在不影响单线程程序正确性的情况下提升程序的性能。虽然重排序并不是必然出现，但是可能对多线程程序的正确性产生影响，即导致线程安全问题。

与内存操作顺序有关的几个术语：

* **源代码顺序**：源代码中所指定的内存访问操作顺序。
* **程序顺序**：在给定处理器上运行的目标代码所指定的内存访问操作顺序。Java虚拟机执行Java代码有两种方式：解释执行（执行字节码）和编译执行（执行机器码）。这里仅将目标代码定义为字节码。
* **执行顺序**：内存访问操作在给定处理器上的实际执行顺序。
* **感知顺序**：给定处理器所感知到的该处理器及其它处理器的内存访问操作发生的顺序。

在此基础上，将重排序划分为**指令重排序**和**存储子系统重排序**：



**2.6.2 指令重排序**

指令重排序是一种动作，它确确实实地对指令的顺序做了调整，其重排序的对象是指令。

提示：

Java平台包括两种编译器：**静态编译器**（javac）和**动态编译器**（JIT编译器）。前者的作用是将Java源代码编译为字节码，在代码编译阶段介入。后者的作用是将字节码动态编译为Java虚拟机宿主机的本地代码（机器码），在Java程序运行过程中介入。

在Java平台中静态编译器基本上不会执行指令重排序，而JIT编译器则可能会执行指令重排序。其中一个典型的重排序就是在new一个对象时，分配了对象所需的内存空间并获得引用之后，JIT编译器在完成对象初始化之前就已经将对象的引用写入变量，这就导致其它线程看到该变量的时候该变量不为null但是却引用了一个还未初始化或初始化未完成的对象。

处理器也可能执行指令重排序使得执行顺序与程序顺序不一致。处理器对指令进行重排序也称为**乱序执行**。在乱序执行的处理器中，指令是一条一条按照程序顺序被处理器读取（**顺序读取**），然后这些指令哪条就绪了（例如指令所需的操作数都已准备好）哪条就先被执行，而不是完全按照程序顺序执行。这些指令执行的结果会被先存入重排序缓冲器，而不是直接被写入寄存器或者主内存。重排序缓冲器会将各个指令的执行结果按照相应指令被处理器读取的顺序提交（即写入）到寄存器或者主内存中（**顺序提交**）。

处理器的乱序执行还采用一种被称为猜测执行的技术，即先执行某种可能可能性，如果行不通再进行回溯。猜测执行会导致if语句的语句体现于其条件语句被执行。

处理器的指令重排序并不会对单线程程序的正确性产生影响，但是会影响多线程程序的正确性。

**2.6.3 存储子系统重排序**

主内存相对于处理器是一个慢速设备。处理器并不是直接访问**主内存**，而是通过**高速缓存**访问主内存的。在此基础上，现代处理器还引入了**写缓冲器**，通过写缓冲器写入高速缓存，再由高速缓存实现写主内存。有的处理器都是直接通过写缓冲器进行写主内存的操作。写缓冲器和高速缓存被统称为**存储子系统**，是处理器的子系统。

在处理器严格按照程序顺序执行两个内存访问操作的情况下，在存储子系统的作用下其它处理器对两个操作的**感知顺序**可能仍然与程序顺序不一致，这种现象称为**存储子系统重排序**，或**内存重排序**。

指令重排序的重排序对象是**指令**，它确实对指令的顺序进行了调整。而存储子系统重排序是一种现象而不是一种动作，它并没有对指令执行顺序进行调整，只是造成了指令执行顺序被调整过一样的现象，其重排序的对象是**内存操作的结果**。

从处理器的角度来说，读内存操作的实质是从指定的RAM地址加载数据到寄存器（Load），写内存操作的实质是将数据存储到指定地址的RAM单元（Store）。内存重排序只有L-L，S-S， L-S和S-L重排序。比如，当两个指令的操作结果需要写入主内存时，会首先将两个操作结果先后写入写缓冲器中。但是为了提高将写缓冲器的内容写入高速缓存的效率，写缓冲器并不保证写操作是先入先出的，即较晚写入写缓冲器的写操作结果可能更早地被写入高速缓存。这就可能导致S-S重排序。

**2.6.4 貌似串行语义**

重排序并未随意地对指令、内存操作的结果进行随意调整，而是必须保证单线程程序的正确性在重排序后不受影响，就好像指令就是按照源代码顺序执行的，这称为**貌似串行语义**。但这并不保证多线程程序的正确性。

如果两个操作（指令）访问同一个变量（地址），且其中一个操作为写操作，那么这两个操作之间就存在**数据依赖关系**。为了保证貌似串行语义，存在数据依赖关系的语句不会被重排序。

如果一条语句的执行结果会决定另一条语句能否执行，那么这两条语句之间就存在**控制依赖关系**。存在控制依赖关系的语句是可以被重排序的。存在控制依赖关系的最典型的语句就是if语句的条件表达式与语句体。这就意味着处理器可能先执行if语句体再执行条件判断。

扩展阅读：单处理器系统是否会受到重排序的影响？

编译器重排序，即静态编译器javac造成的重排序**会对运行在单线程处理器上的多个线程产生影响**。比如data=1; ready=true;这两个语句如果在编译的时候调换了顺序，那么当一个线程运行了ready=true;之后发生了上下文切换，另一个线程运行的时候看到的ready值为true但是data的值还不是1。

运行期重排序，包括JIT编译器造成的重排序、处理器乱序执行重排序以及存储子系统重排序**并不会对单处理器上运行的多线程产生影响**。这些重排序是运行期发生的，相关指令还没完全执行完毕，即它们的执行结果还没提交到主内存，此时的单处理器通常不会立即进行上下文切换，而是等这些指令执行完毕之后。所以当前线程被切出的时候，另一个切入的线程已经看不到重排序的指令了。

**2.6.5 保证内存访问的顺序性**

貌似串行语义只能保障重排序不影响单线程程序的正确性。

那么如何保障重排序不影响多线程程序的正确性？

也即如何避免重排序导致的多线程安全问题？

也即保障感知顺序与源代码顺序一致？

也即保障有序性？

有序性的保障可以理解为通过某些措施使得貌似串行语义扩展到多线程程序。有序性的保障也可以理解为从**逻辑**上**部分**禁止重排序。

从底层的角度来说，禁止重排序是通过调用处理器提供的相应指令（内存屏障）来实现的。对于Java语言，只需使用语言本身提供的机制即可，如volatile和synchronized关键字。

扩展阅读：可见性与有序性的联系与区别：

可见性是有序性的基础。可见性描述的是一个线程对共享变量的更新对另一个线程是否可见。有序性描述的是一个处理器上运行的线程对共享变量的更新，在其它处理器上运行的线程看来是以什么样的顺序**观察到这些更新**。

有序性影响可见性。由于重排序的作用，一个线程对共享变量的更新对于另一个线程而言可能变得不可见。

**2.7 上下文切换**

**2.7.1 上下文切换及其产生原因**

一个线程由于时间片用完或者自身的原因被迫或主动暂停运行，另一个线程被操作系统（线程调度器）选中开始或者继续运行的过程称为**线程上下文切换**。一个线程被剥夺处理器的使用权而被暂停运行就称为**切出**；一个线程被操作系统选中占用处理器开始或者继续运行就成为**切入**。在切入切出的时候操作系统需要保存和恢复相应线程的进度信息，包括计算的中间结果以及执行到哪条指令等等。这个进度信息就称为上下文，一般包括寄存器和程序计数器的内容。在切出时，操作系统需要将上下文保存到内存中；在切入时，操作系统需要从内存中加载恢复选中线程的上下文。

从Java应用的角度来看，一个线程的生命周期状态在RUNNABLE状态与非RUNNABLE状态（BLOCKED、WAITING和TIMED\_WAITING中的任意一个状态）之间切换的过程就是一个上下文切换的过程。

**2.7.2 上下文切换的分类及具体诱因**

按照导致上下文切换的因素划分，可以将上下文切换分为**自发性上下文切换**和**非自发性上下文切换**。

自发性上下文切换指线程由于自身因素导致的切出。一个线程在其运行过程中执行了能使线程进入WAITING、TIMED\_WAITING或者BLOCKED状态的方法或操作都会引发自发性上下文切换，如：

* Thread.*sleep*(long millis)
* Object.wait()/wait(long timeout)/wait(long timeout, int nanos)
* Thread.*yield*()
* Thread.join()/Thread.join(long timeout)
* LockSupport.park()
* 以及线程发起了阻塞式的I/O操作或者等待其它线程持有的锁。

非自发性上下文切换指线程由于线程调度器的原因被迫切出。包括时间片用完或者有一个比被切出线程优先级更高的线程需要执行。Java虚拟机的垃圾回收过程可能也需要暂停所有的应用线程。

**2.7.3 上下文切换的开销和测量**

上下文切换的开销包括**直接开销**和**间接开销**。

直接开销包括：

* 操作系统保存和恢复上下文所需的处理器时间开销
* 线程调度器进行线程调度的开销

间接开销包括：

* 处理器高速缓存重新加载的（时间）开销。一个被切出的线程可能稍后会在另一个处理器上被切入，并且切入时的处理器可能之前并未运行过该线程，那么该线程继续运行所需的变量仍然需要被该处理器重新从主内存或者通过缓存一致性协议从其它处理器加载到高速缓存之中。
* 上下文切换可能导致整个一级高速缓存中的内容被冲刷，即一级高速缓存中的内容会被写入下一级高速缓存（如二级高速缓存）或主内存中。

线程数量越多，可能导致的上下文切换的开销也就可能越大。因此，**多线程程序中线程数量并不是越多计算效率就越高，可能反而越低**。另外，**多线程程序由于额外的上下文切换开销，并不一定就比单线程程序效率更高**。

**2.8 线程的活性故障**

由于资源稀缺性（处理器资源有限导致上下文切换）或者程序自身的问题导致线程一直处于非RUNNABLE状态，或者线程虽处于RUNNABLE状态但是执行的任务却一直无法进展的现象就称为**线程活性故障**。

常见的活性故障包括：

* 死锁（Deadlock）。多个线程互相持有对方所需的资源而导致的矛盾。
* 锁死（Lockout）。线程执行任务必须满足一定的前提，但是这个前提条件却永远无法满足。
* 活锁（Livelock）。活锁的外在表现是线程可能处于RUNNABLE状态，但是线程执行的任务却丝毫没有进展。
* 饥饿。线程无法获得其所需的资源而使得任务无法进展。

**2.9 资源争用与调度**

一次只能够被一个线程占用的资源被称为**排他性资源**，如处理器、数据库连接、文件等。

一个线程占用一个排他性资源而未释放时，其它线程试图访问该资源的现象就称为**资源争用**。

多个线程申请同一个排他性资源的情况下，决定哪个线程会被授予该资源的独占权的过程就是**资源调度**。

资源调度策略的**公平性**指的是，资源的申请线程是否按照其申请资源的顺序而被授予资源的独占权。

资源调度的常见策略是使用队列。资源调度器维护一个等待队列，资源争用时申请失败的线程存入队列并暂停。当资源释放时，等待队列中的一个线程被唤醒并获得再次申请资源的机会，申请成功则移出队列，否则继续等待。等待队列中的线程可能需经历多次暂停和唤醒才能获得资源，这就导致了上下文切换。

**公平调度策略**中的资源申请者总是按照先来后到的顺序来获得资源的独占权。而**非公平调度策略**则允许插队，即资源被释放后，等待队列中的一个线程被唤醒再次申请资源，如果这个时候另一个活跃线程（RUNNABLE状态）也在申请该资源，那么这两个线程可以共同参与资源的抢占。因此，非公平调度策略中，被唤醒的线程不一定就能成功申请到资源。在极端情况下，非公平调度策略可能导致饥饿现象，而公平调度策略则可以避免饥饿现象。

一般来说，非公平调度策略吞吐率较高，单位时间可以为更多的线程调配资源。但是，线程获得资源所需时间的偏差可能较大。

公平调度策略吞吐率较低，因为其维护资源独占权的授予顺序的开销较大。但是线程获得资源所需时间的偏差较小。

在非公平调度策略中，资源被释放时，等待队列中的一个线程会被唤醒，在线程被唤醒到继续运行的这段时间内，如果其它的活跃线程（RUNNABLE）可以先被授予资源的独占权。如果该活跃线程占用资源时间不长，可能并不影响被唤醒线程申请该资源。这种情况下，非公平调度策略可以减少上下文切换。但是，如果多数线程占用资源的时间很长，那么会导致被唤醒的线程需要再次经历暂停和唤醒，从而增加了上下文切换，因此就不适用非公平的调度策略。

默认情况下，非公平调度策略是首选的。当线程占用资源时间较长，或资源的平均申请时间间隔较长，或要求资源申请所需的时间偏差较小的情况下，可以使用公平调度策略。

**第3章 Java线程同步机制**

**3.1 线程同步机制简介**