1. 线程的停止
2. 设置退出标志，正常退出

设置一个退出的标志变量，来控制线程的退出。如使用boolean类型的变量，通过设置该变量为true或者false来控制while循环是否退出。

1. interrupt()中断线程

使用interrupt()方法中断线程有两种情况：

1. 线程处于阻塞状态

当线程阻塞时，可能会经过很长的时间才能获得对象锁，这时，我们可以主动地调用interrupt()方法来中断该线程，阻塞中的线程会抛出InterruptException异常，我们可以捕获这个异常，然后通过break跳出循环，结束这个线程。

1. 线程未处于阻塞状态

若线程未阻塞，则通过isInterrupted()方法来判断线程是否中断来退出循环，当调用interrupt()方法时，中断标志就会置为true，isInterrupt()方法就会返回true，因此通过该返回结果来跳出循环，结束线程。

1. stop()终止线程（不推荐）

极其不安全，废弃的方法。

1. 线程优先级
2. 优先级priority

操作系统采用分时的形式调度运行的线程，线程能够得到多少时间片决定了线程使用处理器资源的多少。Java中，通过priority来控制优先级，范围为1-10,10是最高的，5是最低的。

1. join方法

thread.join()方法将thread线程加入到当前的线程中，当前的线程变为等待状态，thread线程开始执行，等待thread线程执行结束后，当前线程继续执行。

1. yield方法

Thread.yield()方法暂停当前的线程，回到可运行的状态，将CPU资源让给其他具有相同优先级的线程执行。但是，yield方法并不一定能够达到目的，因为当前的线程可能会再次被线程调度选中，继续执行。

1. 多线程的三个特性

多线程必须保证原子性，可见性与有序性，若有一点没有保证，就会导致程序的错误。

1. 原子性

即一个操作或者多个操作，要么全部执行且执行过程不会被任何因素打断，要么都不执行。这就是操作的同步性，只有操作是同步的，才能够保证数据的正常。

1. 可见性

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，其他的线程能够立即看到修改的值，不会出现两个线程得到的数据不一致的情况。

1. 有序性

程序执行的顺序按照代码的先后顺序执行。JVM在真正执行代码的时候，并不一定会保证按照代码的顺序执行，可能会发生指令重排序。处理器为了提高程序运行效率，会优化代码，但是会保证最终的执行结果与代码顺序执行结果一致。处理器在进行重排序时会考虑到指令之间的数据依赖性，如果一个指令2必须用到指令1的结果，那么处理器会保证指令1会在指令2之前执行，在单线程中重排序不会有问题，但是在多线程中就会由于上下文环境的不同造成并发执行的错误。

1. Java内存可见性

JVM内存结构和Java虚拟机的运行时区域有关；

Java对象模型和Java对象在虚拟机中的表现形式有关；

Java内存模型和Java的并发编程有关；

1. JVM内存结构

JVM在执行Java程序时会将所管理的内存划分为若干个不同的区域，Java堆和方法区是所有线程共享的区域，程序计数器，Java虚拟机栈，本地方法栈是用户线程私有的，每个区域有不同的功能。

1. Java对象模型

Java是一种面向对象的语言，Java对象在JVM中的存储也是有一定结构的，Java对象自身的存储模型称为Java对象模型。

1. Java内存模型

Java内存模型是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都是一样的效果。Java内存模型是根据Java Memory Model(JMM)翻译过来的，JMM并不是像JVM内存结构一样是真实存在的，它只是一个抽象的概念。JMM与多线程是相关的，他描述了一组规则或规范，这个规范定义了一个线程对共享变量的写入是对另一个线程可见的。

Java的多线程是通过共享内存通信的，通信过程中会存在可见性，原子性，有序性等问题，JMM就是围绕多线程通信建立的模型，JMM定义了一些语法集，这些语法集映射到Java语言就是volatile, synchronized等关键字。

主内存： 存储所有线程创建的Java实例对象，不论该实例变量是成员变量还是方法中的本地变量（局部变量），也包括了共享的类信息，常量，静态变量，共享数据区域，多线程对同一个变量的访问可能有线程安全问题。

工作内存：存储当前方法的所有本地变量信息，还有主内存中的变量副本，存储在工作内存中的数据不存在线程安全问题。

每个线程都有自己的工作内存，线程对于共享变量的所有操作都是通过将共享变量复制到自己的工作内存中进行的，不能直接在主内存中操作。不同线程之间无法直接访问其他线程的工作内存中的变量，线程间变量值的传递需要通过主内存来完成。

1. 内存可见性问题
2. 概述

一个线程对共享变量值的修改，能够及时地被其他线程看到。当该线程修改完工作内存中的共享变量的副本后，要及时地将该变量更新到主内存中，然后其他线程也要及时地通过从主内存中更新工作内存中的副本。

1. synchronized解决内存可见性

synchronized可以保证方法或者代码块在执行时，同一时刻只有一个线程执行synchronized声明的代码块；还可以保证共享变量的内存可见性；且同一时刻只有一个线程执行，这部分代码块的重排序也不会影响其执行结果。因此，synchronized可保证并发的原子性，可见性，有序性。

JMM关于synchronized的两条规定：

线程解锁前（退出同步代码块时） ： 将该线程工作内存中的共享变量的最新值刷新到主内存中

线程加锁时（进入同步代码块时） ：将清空工作内存中共享变量的值，从主内存中读取最新的值。

只需要在线程的代码中加锁即可更新工作内存的共享变量。

1. votalite关键字解决可见性问题

用votalite关键字声明变量，那么该变量对于每个工作内存就是最新的值，解决了可见性问题。

public votalite int a = 2;

1. 同步锁
2. synchronized

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础，有以下三种同步锁：

1. 普通同步方法，锁是当前的实例对象this
2. 静态同步方法，锁是当前类的Class对象
3. 同步代码块，锁是当前的this对象或者指定的锁对象

当一个线程访问同步代码时，首先需要得到锁才能够执行同步代码，当退出或者抛出异常时需要释放锁。synchronized的同步操作主要是通过monitorenter和moniterexit这两个JVM指令实现的，可在Class文件中查看。

1. 锁优化
2. 锁的状态

synchronized是重量级锁，效率不高，JDK1.6对synchronized的实现进行了优化，如自旋锁，自适应自旋锁，锁消除，锁粗化，偏向锁，轻量级锁等技术来减少锁操作的开销。同时，锁的状态主要是：无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态，随着锁的竞争的激烈性会逐渐升级。锁只可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

1. 自旋锁

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁地阻塞和唤醒对CPU来说负担很重。而且很多应用上，对象锁的锁状态只会持续很短的时间，在这种情况下阻塞和唤醒线程性价比很低，因此，引入自旋锁。自旋锁会让该线程通过一段无意义的循环代码等待一段时间，不会立即被挂起，看持有锁的线程是否会很快释放锁。如果其他线程很快释放了锁，那么自旋的效率很高；若自旋锁持续等待，那么会浪费处理器的资源，因此需要设定自旋等待的次数时间，如果超过这个时间，该线程就要被挂起来进入阻塞状态。

JDK1.4.2中引入，默认是关闭的，可使用-XX:+UserSpinning开启，JDK1.6中默认开启，次数10次，通过-XX:PreBlockSpin调整。但是手动设置自旋次数不太合理，无法预测真实的等待情况，因此引入了自适应自旋锁。

1. 自适应自旋锁

自旋的次数不再是固定的，而是由上一次在同一个锁上的自旋时间以及锁的拥有者的状态决定的。如果上一个线程自旋成功了，那么下一次自旋的次数会增多，因为成功的可能性增大了，允许等待更长时间；反之，如果对于某个锁，很少自旋成功，则之后对于这个锁的自旋次数都会减少甚至省略掉自旋时间，节省CPU资源。

1. 锁消除

当JVM检测到不存在共享数据的竞争时，就不需要这些同步锁了，JVM就会消除这些同步锁，节省毫无意义的请求锁的开销。开发者对于是否需要添加synchronized关键字是清楚的，但是即使没有显示地使用锁，在使用一些JDK的内置API时，如StringBuffer, Vector, HashTable，都会隐形地加上锁。锁消除的依据是逃逸分析的数据支持，JVM通过检测变量是否会逃逸出某个代码段来分析是否需要取消锁。

1. 锁粗化

虽然说在使用同步锁的时候，要让同步代码块的作用范围越小越好，尽在共享数据的实际作用域中才进行同步，这样做使需要同步的操作量尽可能地小，减少资源消耗。。

但是如果频繁地加锁，解锁操作，会耗费很多的资源，所以需要将锁粗化处理，将多个连续的加锁，解锁操作连接在一起，扩展成一个范围更大的锁，这样会减少性能消耗。

1. 偏向锁

轻量级锁的加锁解锁操作是需要依赖多次CAS原子指令的。而偏向锁只需要检查是否为偏向锁、锁标识为以及ThreadID即可，可以减少不必要的CAS操作。

1. 轻量级锁

在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁。

轻量级锁主要使用CAS进行原子操作，但是在多线程竞争的情况下，轻量级锁比重量级锁更慢。

1. 重量级锁

重量级锁通过对象内部的监视器实现，监视器依赖于底层操作系统的Mutex Lock（互斥锁）实现，操作系统实现线程之间的切换需要从用户态切换为内核态，成本很高。