一、摘要

1. 多线程开发有两个核心问题，一个是竞争，另一个是协作。竞争会出现线程安全问题，所以首先总结线程安全的机制，然后是协作的机制。管理竞争和协作是复杂的，所以Java提供了更高层次的服务，比如并发容器类和异步任务执行服务。

(1) 线程安全的机制

(2) 线程的协作机制

(3) 容器类

(4) 任务执行服务

二、具体内容

1. 线程安全的机制

(1) 线程表示一条单独的执行流，每个线程有自己的执行计数器，有自己的栈，但可以共享内存，共享内存是实现线程协作的基础，但共享内存有两个问题，竞态条件和内存可见性，解决方法：

① 使用synchronized

② 使用显式锁

③ 使用volatile

④ 使用原子变量和CAS

⑤ 写时复制

⑥ 使用ThreadLocal

1.1 synchronized – 锁住对象实现线程安全 – 只能修饰代码块或者方法

(1) 只是一个关键字，大部分情况下，放到类的方法声明上就可以了，既可以解决竞态条件问题，也可以解决内存可见性问题。

(2) 需要理解的是，它保护的是对象，而不是代码，只有对同一个对象的synchronized方法调用，synchronized才能保证它们被顺序调用。对于实例方法，这个对象是this，对于静态方法，这个对象是类对象，对于代码块，需要指定哪个对象。

(3) 需要注意，它不能尝试获取锁，也不响应中断，还可能会死锁。不过，相比显式锁，synchronized简单易用，JVM也可以不断优化它的实现，应该被优先使用。

1.2 显式锁

(1) 显式锁是相对于synchronized隐式锁而言的，它可以实现synchronzied同样的功能，但需要程序员自己创建锁，调用锁相关的接口，主要接口是Lock，主要实现类是ReentrantLock。

(2) 相比synchronized，显式锁支持以非阻塞方式获取锁、可以响应中断、可以限时、可以指定公平性、可以解决死锁问题，这使得它灵活的多。

(3) 在读多写少、读操作可以完全并行的场景中，可以使用读写锁以提高并发度，读写锁的接口是ReadWriteLock，实现类是ReentrantReadWriteLock。

1.3 volatile – 保证内存可见性

synchronized和显式锁都是锁，使用锁可以实现安全，但使用锁是有成本的，获取不到锁的线程还需要等待，会有线程的上下文切换开销等。保证安全不一定需要锁。如果共享的对象只有一个，操作也只是进行最简单的get/set操作，set也不依赖于之前的值，那就不存在竞态条件问题，而只有内存可见性问题，这时，在变量的声明上加上volatile就可以了。

1.4 原子变量和CAS

(1) 使用volatile，set的新值不能依赖于旧值，但很多时候，set的新值与原来的值有关，这时，也不一定需要锁，如果需要同步的代码比较简单，可以考虑原子变量，它们包含了一些以原子方式实现组合操作的方法，对于并发环境中的计数、产生序列号等需求，考虑使用原子变量而非锁。

(2) 原子变量的基础是CAS，比较并设置，一般的计算机系统都在硬件层次上直接支持CAS指令。通过循环CAS的方式实现原子更新是一种重要的思维，相比synchronized，它是乐观的，而synchronized是悲观的，它是非阻塞式的，而synchronized是阻塞式的。CAS是Java并发包的基础，基于它可以实现高效的、乐观、非阻塞式数据结构和算法，它也是并发包中锁、同步工具和各种容器的基础。\

1.5 写时复制

写时复制就是将共享访问的对象变为只读的，写的时候，再使用锁，保证只有一个线程写，写的线程不是直接修改原对象，而是新创建一个对象，对该对象修改完毕后，再原子性地修改共享访问的变量，让它指向新的对象。

1.6 ThreadLocal

ThreadLocal就是让每个线程，对同一个变量，都有自己的独有拷贝，每个线程实际访问的对象都是自己的，自然也就不存在线程安全问题了。

2. 线程的协作机制

(1) 多线程之间的核心问题，除了竞争，就是协作。我们在67节和68节介绍了多种协作场景，比如生产者/消费者协作模式、主从协作模式、同时开始、集合点等。之前章节探讨了协作的多种机制：

① wait/notify

② 显式条件

③ 线程的中断

④ 协作工具类

⑤ 阻塞队列

⑥ Future/FutureTask

2.1 wait/notify

(1) wait/notify与synchronized配合一起使用，是线程的基本协作机制，每个对象都有一把锁和两个等待队列，一个是锁等待队列，放的是等待获取锁的线程，另一个是条件等待队列，放的是等待条件的线程，wait将自己加入条件等待队列，notify从条件等待队列上移除一个线程并唤醒，notifyAll移除所有线程并唤醒。

(2) 需要注意的是，wait/notify方法只能在synchronized代码块内被调用，调用wait时，线程会释放对象锁，被notify/notifyAll唤醒后，要重新竞争对象锁，获取到锁后才会从wait调用中返回，返回后，不代表其等待的条件就一定成立了，需要重新检查其等待的条件。

(3) wait/notify方法看上去很简单，但往往难以理解wait等的到底是什么，而notify通知的又是什么，只能有一个条件等待队列，这也是wait/notify机制的局限性，这使得对于等待条件的分析变得复杂。

2.2 显式条件

显式条件与显式锁配合使用，与wait/notify相比，可以支持多个条件队列，代码更为易读，效率更高，使用时注意不要将signal/signalAll误写为notify/notifyAll。

2.3 中断

Java中取消/关闭一个线程的方式是中断，中断并不是强迫终止一个线程，它是一种协作机制，是给线程传递一个取消信号，但是由线程来决定如何以及何时退出，线程在不同状态和IO操作时对中断有不同的反应，作为线程的实现者，应该提供明确的取消/关闭方法，并用文档清楚描述其行为，作为线程的调用者，应该使用其取消/关闭方法，而不是贸然调用interrupt。

2.4 协作工具类（同步器）

(1) 信号量类Semaphore用于限制对资源的并发访问数。

(2) 倒计时门栓CountDownLatch主要用于不同角色线程间的同步，比如在"裁判"-"运动员"模式中，"裁判"线程让多个"运动员"线程同时开始，也可以用于协调主从线程，让主线程等待多个从线程的结果。

(3) 循环栅栏CyclicBarrier用于同一角色线程间的协调一致，所有线程在到达栅栏后都需要等待其他线程，等所有线程都到达后再一起通过，它是循环的，可以用作重复的同步。

2.5 阻塞队列

(1) 对于最常见的生产者/消费者协作模式，可以使用阻塞队列，阻塞队列封装了锁和条件，生产者线程和消费者线程只需要调用队列的入队/出队方法就可以了，不需要考虑同步和协作问题。

(2) 阻塞队列有普通的先进先出队列，包括基于数组的ArrayBlockingQueue和基于链表的LinkedBlockingQueue/LinkedBlockingDeque，也有基于堆的优先级阻塞队列PriorityBlockingQueue，还有可用于定时任务的延时阻塞队列DelayQueue，以及用于特殊场景的阻塞队列SynchronousQueue和LinkedTransferQueue。

2.6 Future/FutureTask（线程执行结果）

(1) 在常见的主从协作模式中，主线程往往是让子线程异步执行一项任务，获取其结果，手工创建子线程的写法往往比较麻烦，常见的模式是使用异步任务执行服务，不再手工创建线程，而只是提交任务，提交后马上得到一个结果，但这个结果不是最终结果，而是一个Future，Future是一个接口，主要实现类是FutureTask。

(2) Future封装了主线程和执行线程关于执行状态和结果的同步，对于主线程而言，它只需要通过Future就可以查询异步任务的状态、获取最终结果、取消任务等，不需要再考虑同步和协作问题。

3. 容器类

(1) 线程安全的容器有两类，一类是同步容器，另一类是并发容器。

(2) 并发容器有：

① 写时拷贝的List和Set

② ConcurrentHashMap

③ 基于SkipList的Map和Set

④ 各种队列

3.1 同步容器

(1) Collections类中有一些静态方法，可以基于普通容器返回线程安全的同步容器，比如：

public static <T> Collection<T> synchronizedCollection(Collection<T> c)

public static <T> List<T> synchronizedList(List<T> list)

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

(2) 是给所有容器方法都加上synchronized来实现安全的。同步容器的性能比较低，另外，还需要注意一些问题，比如复合操作和迭代，需要调用方手工使用synchronized同步，并注意不要同步错对象。

(3) 并发容器是专为并发而设计的，线程安全、并发度更高、性能更高、迭代不会抛出ConcurrentModificationException、很多容器以原子方式支持一些复合操作。

3.2 并发容器之写时拷贝的List和Set

CopyOnWriteArrayList基于数组实现了List接口，CopyOnWriteArraySet基于CopyOnWriteArrayList实现了Set接口，它们采用了写时拷贝，适用于读远多于写，集合不太大的场合。不适用于数组很大，且修改频繁的场景。它们是以优化读操作为目标的，读不需要同步，性能很高，但在优化读的同时就牺牲了写的性能。.

3.3 并发容器之ConcurrentHashMap

HashMap不是线程安全的，在并发更新的情况下，HashMap的链表结构可能形成环，出现死循环，占满CPU。ConcurrentHashMap是并发版的HashMap，通过分段锁和其他技术实现了高并发，读操作完全并行，写操作支持一定程度的并行，以原子方式支持一些复合操作，迭代不用加锁，不会抛出ConcurrentModificationException。

3.4 并发容器之基于SkipList的Map和Set

(1) Java并发包中与TreeMap/TreeSet对应的并发版本是ConcurrentSkipListMap和ConcurrentSkipListSet。ConcurrentSkipListMap是基于SkipList实现的，SkipList称为跳跃表或跳表，是一种数据结构，主要操作复杂度为O(log(N))，并发版本采用跳表而不是树，是因为跳表更易于实现高效并发算法。

(2) ConcurrentSkipListMap没有使用锁，所有操作都是无阻塞的，所有操作都可以并行，包括写。与ConcurrentHashMap类似，迭代器不会抛出ConcurrentModificationException，是弱一致的，也直接支持一些原子复合操作。

3.5 并发容器之队列

各种阻塞队列主要用于协作，非阻塞队列适用于多个线程并发使用一个队列的场合，有两个非阻塞队列，ConcurrentLinkedQueue和ConcurrentLinkedDeque，ConcurrentLinkedQueue实现了Queue接口，表示一个先进先出的队列，ConcurrentLinkedDeque实现了Deque接口，表示一个双端队列。它们都是基于链表实现的，都没有限制大小，是无界的，这两个类最基础的实现原理是循环CAS，没有使用锁。

4. 任务执行服务（执行器）

(1) 各个部分：

任务执行服务的基本概念

主要实现方式 - 线程池

方便处理结果的CompletionService

定时任务

4.1 基本概念

(1) 任务执行服务大大简化了执行异步任务所需的开发，它引入了一个"执行服务"的概念，将"任务的提交"和"任务的执行"相分离，"执行服务"封装了任务执行的细节，对于任务提交者而言，它可以关注于任务本身，如提交任务、获取结果、取消任务，而不需要关注任务执行的细节，如线程创建、任务调度、线程关闭等。

(2) 任务执行服务主要涉及以下接口：

① Runnable和Callable：表示要执行的异步任务

② Executor和ExecutorService：表示执行服务

③ Future：表示异步任务的结果

④ 使用者只需要通过ExecutorService提交任务，通过Future操作任务和结果即可，不需要关注线程创建和协调的细节。

4.2 线程池

(1) 任务执行服务的主要实现机制是线程池，实现类是ThreadPoolExecutor，线程池主要由两个概念组成，一个是任务队列，另一个是工作者线程。任务队列是一个阻塞队列，保存待执行的任务。工作者线程主体就是一个循环，循环从队列中接受任务并执行。

(2) ThreadPoolExecutor实现了生产者/消费者模式，工作者线程就是消费者，任务提交者就是生产者，线程池自己维护任务队列。当我们碰到类似生产者/消费者问题时，应该优先考虑直接使用线程池，而非重新发明轮子，自己管理和维护消费者线程及任务队列。

4.3 CompletionService

在异步任务程序中，一种场景是，主线程提交多个异步任务，然后希望有任务完成就处理结果，并且按任务完成顺序逐个处理，对于这种场景，Java并发包提供了一个方便的方法，使用CompletionService，这是一个接口，它的实现类是ExecutorCompletionService，它通过一个额外的结果队列，方便了对于多个异步任务结果的处理。

4.4 定时任务

(1) 异步任务中，常见的任务是定时任务。在Java中，有两种方式实现定时任务：

使用java.util包中的Timer和TimerTask

使用Java并发包中的ScheduledExecutorService

(2) Timer有一些需要特别注意的事项：

① 一个Timer对象背后只有一个Timer线程，这意味着，定时任务不能耗时太长，更不能是无限循环

② 在执行任何一个任务的run方法时，一旦run抛出异常，Timer线程就会退出，从而所有定时任务都会被取消

(3) ScheduledExecutorService的主要实现类是ScheduledThreadPoolExecutor，它没有Timer的问题：（相当于是定时任务的执行器）

① 它的背后是线程池，可以有多个线程执行任务

② 任务执行线程会捕获任务执行过程中的所有异常，一个定时任务的异常不会影响其他定时任务