目录

[第一章 简介 1](#_Toc6248540)

[第二章 线程安全性 1](#_Toc6248541)

[第三章 对象的共享 3](#_Toc6248542)

[第四章 对象的组合 4](#_Toc6248543)

[第五章 基础构建模块 5](#_Toc6248544)

[第六章 任务执行 7](#_Toc6248545)

[第七章 取消与关闭 8](#_Toc6248546)

[第八章 线程池的使用 10](#_Toc6248547)

[第九章 图形用户界面应用程序 11](#_Toc6248548)

[第十章 避免活跃性危险 11](#_Toc6248549)

[第十一章 性能与可伸缩性 12](#_Toc6248550)

[第十二章 并发程序的测试 13](#_Toc6248551)

[第十四章 构建自定义的同步工具 13](#_Toc6248552)

[第十五章 原子变量与非阻塞同步机制 14](#_Toc6248553)

[第十六章 Java内存模型 15](#_Toc6248554)

# 第一章 简介

1、操作系统为各个独立执行的进程分配各种资源，包括：内存、文件句柄、安全证书等。

2、不同进程间，粗粒度的通信机制：套接字、信号处理器、共享内存、信号量以及文件等。

3、在计算机中加入操作系统来实现多个程序的同时执行的原因：

1）资源利用率：IO阻塞时，可让其他程序利用CPU

2）公平性：时间分片来使每个程序都能运行

3）便利性：程序各司其职，比都放在一个任务里完成所有事情，更便利

4、线程的优势：

1）发挥多处理器的强大能力

2）建模的简单性

3）异步事件的简化处理

4）响应更灵敏的用户界面

5、线程带来的风险：

1）安全性问题：操作执行顺序不可预测，指令重排也可能会使实际情况更糟，即期望永远不发生糟糕的事情

2）活跃性问题：某件正确的事情最终会发生

3）性能问题：正确的事情尽快发生

6、线程无处不在：框架中可能会创建线程，例如：Timer、Servlet（JSP）、远程方法调用（RMI）、GUI

# 第二章 线程安全性

1、编写线程安全的代码，核心在于：对状态访问操作进行管理，特别是对共享的和可变的状态进行访问

注：对象的状态：指存储在状态变量中的数据

2、一个对象是否需要线程安全，取决于：在程序中访问对象的方式，而不是对象要实现的功能。

3、同步机制：

1）关键字synchronized，独占式加锁

2）volatile类型的变量

3）显示锁Lock

4）原子变量

4、线程同步的解决思路：

1）不在线程之间共享状态变量

2）将状态变量修改为不可变的变量

3）在访问状态变量时使用同步

5、正确的编程方法：优先保证正确性，然后提高性能

6、线程安全定义：当多个线程访问某个类时，这个类始终能表现出正确的行为，就称这个类时线程安全的。

注：1）在线程安全类中封装了必要的同步机制，因此客户端无需进一步采取同步措施。

2）无状态对象一定是线程安全的

7、竞态条件与数据竞争

1）竞态条件：由于不恰当的执行时序而出现不正确的结果，即某个计算的正确性取决于多个线程的交替执行时序时。其本质是：基于一种可能失效的观察结果来做出判断或执行某个计算，即先检查后执行。

2）数据竞争：在访问共享的非final类型的域时没有采用同步来进行协同，就会出现数据竞争。即当一个线程写入一个变量而另一个线程接下来读取这个变量，或者读取一个之前由另一个线程写入的变量时，并且在这两个线程之间没有使用同步。

8、复合操作：指对于访问同一个状态的所有操作（包括该操作本身）来说，这个操作是一个以原子方式执行的操作，以确保线程安全性。

9、java.util.concurrent.atomic里面为原子变量类。

注：在实际情况中，应尽可能地使用现有的线程安全对象来管理类的状态。

10、要保持状态的一致性，就需要在单个原子操作中更新所有相关的状态变量。

11、同步代码块，包括两部分：一个作为锁的对象引用，一个作为由这个锁保护的代码块。synchronized修饰方法，是一种横跨整个方法体的同步代码块，锁就是方法调用所在的对象，静态的synchronized方法以Class对象作为锁。

注：若所在对象不同则互相隔离，互不影响；若所在对象相同，则会相互影响；锁Class与锁实例对象，互相隔离，互不影响。

12、每个java对象都可以用作一个实现同步的锁，这些锁被称为内置锁或监视器锁。获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。

注：内置锁是互斥锁。

13、重入：某个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。意味着获取锁的操作粒度是线程而不是调用。

注：1）内置锁是可重入的

2）重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。

14、对于可能被多个线程同时访问的可变状态变量，在访问它时都需要持有同一个锁，称状态变量是由这个锁保护的。

注：某个线程在获得对象的锁之后，只能阻止其他线程获得同一个锁。

15、一种常见的加锁约定：将所有的可变状态都封装在对象内部，并通过对象的内置锁对所有访问可变状态的代码路径进行同步，使得在该对象上不会发生并发访问。

16、对于每个包含多个变量的不变性条件，其中涉及的所有变量都需要由同一个锁来保护。

17、串行访问：多个线程依次以独占的方式访问对象，而不是并发的访问。

18、当实现某个同步策略时，一定不要盲目地为了性能而牺牲简单性（这可能会破坏安全性）

19、当执行时间较长的计算或者可能无法快速完成的操作时（例如I/O）一定不要持有锁

# 第三章 对象的共享

1、在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。

2、只要有数据在多个线程之间共享，就应使用正确的同步。

3、失效数据：读未提交（其他线程已经更新该值但还未写入时）

4、最低安全性：读取一个失效值，至少这个值是由之前某个线程设置的值，而不是一个随机值。

5、对于非volatile类型的long和double变量，JVM允许将64位的读操作或写操作分解为两个32位的操作。很可能会读到某个值的高32位和另一个值的低32位。

注：可用关键字volatile声明它们或使用锁保护起来

6、加锁的含义不仅仅局限于互斥行为，还包括内存可见性。为了确保所有线程都能看到共享变量的最新值，所有执行读操作或写操作的线程都必须在同一个锁上同步。

7、volatile变量用来确保将变量的更新操作通知到其他线程，即：编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序，volatile变量不会被缓存在寄存器或者其他处理器不可见的地方。在写入volatile变量之前对A可见的所有变量的值，在B读取了volatile变量后，对B也是可见的。

注：访问volatile变量时不会执行加锁操作，是一种比synchronized关键字更轻量级的同步机制。

8、加锁机制既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性。

9、仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它们，包括：确保它们自身状态的可见性，确保它们所引用对象的状态的可见性，以及标识一些重要的程序声明周期事件的发生。使用volatile的条件（需同时满足）：

1）对变量的写入操作不依赖于变量的当前值，或者确保只有单个线程更新变量的值

2）该变量不会与其他变量一起纳入不可变性条件中

3）在访问变量时不需要加锁

10、当启动JVM时一定要指定 -server命令行选项，以server的模式启动

11、发布：指使对象能够在当前作用域之外的代码中使用。

12、逸出：指某个不应该被发布的对象被发布出去。

1）当发布一个对象时，在该对象的非私有域中引用的所有对象同样会被发布。

2）在构造方法中使用this引用逸出，当且仅当对象的构造函数返回时，对象才处于可预测和一致的状态

13、外部方法：指行为并不完全由该类来规定的方法，包括其他类中定义的方法以及该类中可以被改写的方法。

14、线程封闭：仅在单线程内访问数据，不需要同步。常见：Ad-hoc线程封闭、栈封闭、ThreadLocal类。

15、Connection非线程安全，通过使用连接池管理使其线程安全。

16、Ad-hoc线程封闭：指维护线程封闭性的职责完全由程序实现来承担。

17、栈封闭：是线程封闭的一种特例，在栈封闭中只能通过局部变量才能访问对象。也称线程内部使用或线程局部使用。

注：基本类型的局部变量始终封闭在线程内。

18、ThreadLocal类通常用于防止对可变的单实例变量或全局变量进行共享。其get、set方法为每个使用该变量的线程都存有一份独立的副本，即get总是返回由当前线程在调用set时设置的最新值。

注：1）某个线程初次调用ThreadLocal.get方法时，会调用initialValue来获取初始值。这些特定于线程的值保存在Thread对象中，当线程终止后，这些值会作为垃圾回收。

2）ThreadLocal变量类似于全局变量，会降低代码可重用性，在类之间引入隐含的耦合性，使用需注意。

19、不可变对象：在创建后其状态就不能被修改。不可变对象一定是线程安全的。

20、满足不可变对象的条件：

1）对象创建以后其状态就不能被修改

2）对象的所有域都是final类型

3）对象是正确创建的（构造函数中，无this引用逸出）

21、不可变对象与不可变对象的引用存在差异，不可变对象中的程序状态仍然可以更新，即通过一个保存新状态的实例来替换原有的不可变对象。

22、除非需要更高的可见性，否则应将所有的域都声明为私有域；除非需要某个域是可变的，否则应将其声明为final域。

23、使用Volatile类型来发布不可变对象：使用包含多个状态变量的容器对象来维持不变性条件，并使用一个volatile类型的引用来确保可见性，使在没有显式地使用锁的情况下仍然是线程安全的。

24、不正确的发布会导致正确的对象被破坏。

25、java内存模型为不可变对象的共享提供了一种特殊的初始化安全性保证。

26、要安全的发布一个对象，对象的引用以及对象的状态必须同时对其他线程可见，常用方式：

1）在静态初始化函数中初始化一个对象引用

2）将对象的引用保存到volatile类型的域或者AtomicReferance对象中

3）将对象的引用保存到某个正确构造对象的final类型域中

4）将对象的引用保存到一个由锁保护的域中

27、事实不可变对象：从技术上来看是可变的，但其状态在发布后不会再改变。

注：在没有额外同步的情况下，任何线程都可以安全地使用被安全发布的事实不可变对象

28、对象的发布与可变性的关系：

1）不可变对象可以通过任意机制来发布

2）事实不可变对象必须通过安全方式来发布

3）可变对象必须通过安全方式来发布，并且必须是线程安全的或者由某个锁保护起来。

29、并发程序中使用和共享对象的策略：

1）线程封闭

2）只读共享：包括不可变对象和事实不可变对象

3）线程安全共享：线程安全的对象在其内部实现同步

4）保护对象：被保护的对象只能通过持有特定的锁来访问

# 第四章 对象的组合

1、设计线程安全类的基本要素：

1）找出构成对象状态的所有变量

2）找出约束状态变量的不变性条件

3）建立对象状态的并发访问管理策略

2、先验/后验条件

1）先验条件：针对方法，规定了在调用方法之前必须为真的条件。依赖当前状态，等到先验条件为真在执行。

2）后验条件：针对方法，规定了在调用方法之后必须为真的条件。不一定依赖当前状态，但承诺方法调用后一定会有正确的结果。

3、等到为真在执行，java提供了内置机制（等待和通知等机制），与内置加锁机制紧密相关。

注：一种简单的方法是：使用阻塞队列、信号量来实现依赖状态的行为。

4、实例封闭机制，简称封闭：将数据封装在对象内部，可以将数据的访问限制在对象的方法上，从而更容易确保线程在访问数据时总能持有正确的锁。

5、钝化操作：指将状态保存到持久性存储。

6、遵循Java监视器模式的对象：会把对象的所有可变状态都封装起来，并由对象自己的内置锁来保护。

7、线程安全性的委托：通过多个线程安全类组合而成的类是线程安全的。

1）如果一个类是由多个独立且线程安全的状态变量组成，并且在所有的操作中都不包含无效状态转换，那么可以将线程安全性委托给底层的状态变量。

2）如果一个状态变量是线程安全的，并且没有任何不变性条件来约束它的值，在变量的操作上也不存在任何不允许的状态转换，那么就可以安全地发布这个变量。

8、在现有的线程安全的类中添加功能：

1）在原始类中添加一个方法

2）对类进行扩展（子类的方式）

3）客户端加锁机制（辅助类的方式）：指对于使用某个客户端X的代码，使用X本身用于保护其状态的锁（X的内置锁）来保护这段客户代码。

4）组合（委托的方式）

注：客户端加锁机制与扩展类机制都是将派生类的行为与基类的实现耦合在一起，破坏实现的封装性；组合的方式最好。

9、在文档中说明客户端代码需要了解的线程安全性保证，以及代码维护人员需要了解的同步策略。

# 第五章 基础构建模块

1、同步容器类包括Vector和Hashtable，将它们的状态封装起来，并对每个公有方法都进行同步，使得每次只有一个线程能访问容器的状态。

注：这些容器在迭代过程中被修改时，采用及时失败的策略。将容器的变化与容器关联起来实现。

2、封装对象的状态有助于维持不变性条件，封装对象的同步机制有助于确保实施同步策略。

3、并发容器包括ConcurrentHashMap和CopyOnWriteArrayList（遍历操作为主要操作时使用），通过并发容器来代替同步容器，可以极大地提高伸缩性并降低风险。

4、Queue用于临时保存一组等待处理的元素，底层使用LinkedList实现。BlockingQueue增加了可阻塞的插入和获取等操作。

5、ConcurrentHashMap：采用分段锁即一种粒度更细的加锁机制。读并发，写在一定程度上并发。

6、CopyOnWriteArrayList：写入时先复制一份，修改后，将引用指向新List地址。同步原理：正确地发布一个事实不可变的对象，在访问该对象时就不再需要进一步的同步。

注：存在一定性能开销，用于遍历多，写入少的情况

6、阻塞队列提供可阻塞的put、get方法，支持定时的offer（返回true/false）、poll（返回移除对象或null）方法，采用生产者-消费者模式。

1）阻塞队列：LinkedBlockingQueue、ArrayBlockingQueue是FIFO队列

2）优先级队列：PriorityBlockingQueue是一个按优先级排序的队列

3）同步队列SynchronousQueue不会为队列中元素维护存储空间、它维护一组线程，直接将任务交付给生产者。

注：仅当有足够多的消费者，并且总是有一个消费者准备好获取交付的工作时，才适合使用同步队列。

4）双端队列Deque：实现了在队列头部和队列尾部的高效插入和移除，实现包括ArrayDeque和LinkedBlockingDeque。适用于工作密取。

注：工作密取指：每个消费者都有自己的双端队列，当一个消费者完成了自己双端队列中的全部工作，可以从其他消费者双端队列末尾秘密地获取工作。

7、在构建高可靠的应用程序时，有界队列是一种强大的资源管理工具，它们能抑制并防止产生过多的工作项，使应用程序在负荷过载的情况下变得更加健壮。

8、线程阻塞与中断

1）阻塞：等待IO操作结束、等待获得一个锁、等待从Thread.sleep中清醒过来，等待另一个线程的计算结果。其状态有：BLOCKED、WAITING、TIMED\_WAITING

2）中断是一种协作机制，处理方式有：a、传递给调用者；b、调用当前线程上的interrupt方法恢复中断状态

9、同步工具类：封装了一些状态，这些状态将决定执行同步工具类的线程是继续执行还是等待，此处还提供了一些方法对状态进行操作，以及另一些方法对于高效地等待同步工具类进入到预期状态。包括：闭锁、Future、信号量、栅栏

10、闭锁：在闭锁达到结束状态之前，这扇门一直是关闭的，并且没有任何线程能通过，当达到结束状态时，这扇门会打开并允许所有的线程通过。CountDownLatch可以使一个或多个线程等待一组事件的发生。闭锁状态包括一个计数器，表示需要等待的事件数量，构造函数中初始化；countDown减1表示一个事件已经发生；await阻塞等待计数器到达零或等待中的线程中断或等待超时。

11、FutureTask通过Callable来实现计算，相当于一种可生成结果的Runnable。3种状态：等待运行、正在运行、运行完成。3种结束方式：正常结束、由于取消而结束、由于异常而结束。其get方法将阻塞直到任务进入完成状态，然后返回结果或抛出异常。可用于跨线程安全发布结果。

注：Future表示一种抽象的可生成结果的计算

12、计数信号量（Counting Semaphore）：用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。其管理着一组虚拟的许可，构造函数中指定初始化数量；acquire方法将阻塞直到获取到许可（或中断或超时）；release方法将返回一个许可。

注：初始值为1的二值信号量可以用做互斥体，作为不可重入加锁。

13、栅栏（Barrier）：能阻塞一组线程直到某个事件发生。CyclicBarrier可将一个问题拆分成一系列相互独立的子问题。当线程到达栅栏位置时将调用await方法，这个方法将阻塞直到所有线程到到达栅栏位置。

注：1）闭锁用于等待事件，而栅栏用于等待其他线程。

2）Exchange是一种两方栅栏，各方在栅栏位置上交换数据。

14、缓存污染：缓存中存储了错误结果

15、小结

1）所有的并发问题都可以归结为如何协调对并发状态的访问。可变状态越少，就越容易确保线程安全性。

2）尽量将域声明为final类型，除非需要它们是可变的

3）不可变对象一定是线程安全的

4）封装有助于管理复杂性

5）用锁保护每个可变变量

6）当保护同一个不变性条件中的所有变量时，要使用同一个锁

7）在执行复合操作期间，要持有锁

# 第六章 任务执行

1、任务：通常是一些抽象的且离散的工作单元，相互独立，不依赖于其他任务的状态、结果或边界。

2、一种自然的任务边界选择方式：以独立的客户请求为边界。

3、无限制创建线程的不足：

1）线程生命周期的开销非常高

2）资源消耗

3）稳定性：JVM和底层操作系统对线程数的限制

4、Executor框架：提供了一种标准的方法将任务的提交过程与执行过程解耦开来，并利用Runnable来表示任务。

5、执行策略：

1）在哪个线程中执行

2）执行顺序：FIFO、LIFO、优先级

3）并发数

4）任务队列容量

5）过载时拒绝任务：拒绝哪一个任务，如何通知应用程序

6）执行一个任务前后应进行哪些操作

6、线程池：指管理一组同构工作线程的资源池

1）工作队列：保存所有等待执行的任务

2）工作者线程：从工作队列中获取一个任务，执行任务，然后返回线程池并等待下一个任务

7、Executors自带线程池：

1）newFixedThreadPool：创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程池的规模不再变化

2）newCachedThreadPool：创建一个可缓存的线程池，如果线程池的当前规模超过了处理需求时，那么将回收空闲的线程，当需求增加时，可添加新的线程，线程池的规模不存在任何限制。

3）newSingleThreadExecutor：创建单个工作者线程来执行任务

4）newScheduledThreadPool：创建一个固定长度的线程池，以延迟或定时的方式来执行任务

注：1） FixedThreadPool 和 SingleThreadPool：允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。

2） CachedThreadPool 和 ScheduledThreadPool：允许的创建线程数量为 Integer.MAX\_VALUE， 可能会创建大量的线程，从而导致 OOM。

8、Executor生命周期：运行、关闭、已终止

1）shutdown方法：不再接受新的任务，正在执行和队列中的任务会执行完成

2）shutdownNow方法：尝试取消正在执行的任务，队列中的不会执行，返回队列中所有未执行的任务清单

3）awaitTermination方法：阻塞等待ExecutorService到达终止状态。调用后会自动调用shutdown

4）isTerminated方法：判断线程池是否已终止

9、Timer类：

1）基于绝对时间的调度机制

2）所有定时任务只会创建一个线程

3）不同任务执行时间冲突时，延迟或拒绝下一个任务

4）某个任务抛出异常，会终止所有的任务

# 第七章 取消与关闭

1、提前结束任务或线程：stop和取消，应采用后者

2、外部代码能够在某个操作正常完成之前将其置入“完成”状态，称这个操作为可取消的。

3、取消的原因：

1）用户请求取消

2）有时间限制的操作

3）应用程序事件

4）错误

5）关闭

4、取消的常用协作机制：标识法和中断。

注：一个可取消的任务必须拥有取消策略

5、标识法：设置某个“已请求取消”标志，任务将定期地查看该标识

6、中断：每个线程都有一个boolean类型的中断状态。当中断线程时，这个线程的中断状态将被设置为true。即：它并不会真正地中断一个正在运行的线程，而只是发出中断请求，然后由线程在下一个合适的时刻中断自己（这些时刻也称为取消点）

1）interrupt方法：中断目标线程

2）isInterrupted方法：返回目标线程的中断状态

3）静态的 interrupted方法：清除当前线程的中断状态，返回它之前的值

注：中断是实现取消的最合理方式。还可通过Future来实现取消

7、阻塞库方法：sleep、wait等响应中断时执行的操作有：清除中断状态，抛出InterruptedException，表示阻塞操作由于中断而提前结束

8、中断策略规定：当发现中断请求时，应该做哪些工作，哪些工作单元对于中断来说是原子操作，以多快的速度来响应中断。

1）最合理的中断策略：线程级或服务级的取消操作：尽快退出，在必要时进行清理，通知某个所有者该线程已经退出。

2）中断线程池中的某个工作者线程意味着：取消当前任务和关闭工作者线程

3）由于每个线程拥有各自的中断策略，因此除非知道中断对该线程的含义，否则不应该中断这个线程。

9、响应中断的策略：

1）传递异常：抛出

2）恢复中断状态：再次调用interrupt来恢复中断状态

注：只有实现了线程中断策略的代码才可以屏蔽中断请求，在常规的任务和库代码中都不应该屏蔽中断请求。

10、处理不可中断的阻塞：

1）Socket I/O：关闭底层套接字

2）同步I/O：中断

3）异步I/O：调用close或wakeup方法

4）获取某个锁：Lock类中提供了lockInterruptibly方法，允许在等待一个锁的同时仍能响应中断

11、采用newTaskFor来封装非标准的取消：它是一个工厂方法，创建Future来代表任务，通过定制表示任务的Future可以改变Future.cancel的行为。

12、正确的封装原则：除非拥有某个线程，否则不能对该线程进行操控

注：对于持有线程的服务，只要服务的存在时间大于创建线程的方法的存在时间，那么就应该提供生命周期方法。

13、log方法将日志消息放入某个队列中，并由其他线程来处理。

14、毒丸：指一个放在队列上的对象，其含义是：当得到这个对象时，立即停止。

15、RuntimeException不会在调用栈中逐层传递，而是默认的在控制台中输出栈追踪信息，并终止线程。

16、当一个线程由于未捕获异常而退出时，JVM会把这个事件报告给应用程序提供的UncaughtExceptionHandler异常处理器。

17、JVM关闭：

1）正常关闭：a、当最后一个普通线程结束时；b、调用System.exit时；c、通过其他特定于平台的方法关闭时

2）非正常关闭：a、调用Runtime.halt；b、杀死JVM进程

18、关闭钩子：指通过Runtime.addShutdownHook注册的但尚未开始的线程（属于普通线程）。注：不保证调用顺序

1）正常关闭时：

a、调用所有已注册的关闭钩子；

b、关闭进程与所有线程并发执行；

c、当关闭钩子执行完毕，且runFinalizersOnExit为true时，JVM运行终结器，然后停止；

d、JVM最终结束后，所有线程被强行结束

2）强行关闭：只关JVM，不会运行关闭钩子

注：关闭钩子应该是线程安全的，可以用于实现服务或应用程序的清理工作

19、线程分两种：普通线程和守护线程

1）在JVM启动时创建的所有线程中，除了主线程以外，其他线程都是守护线程（垃圾回收期、辅助线程等）

2）创建一个线程时，新线程将继承创建它的线程的守护状态，默认情况下，主线程创建的所有线程都是普通线程。

3）普通线程与守护线程的差异：当一个线程退出时，JVM会检查其他正在运行的线程，如果都是守护线程，则JVM正常退出。停止时，所有守护线程都被抛弃，即：既不会执行finally代码块，也不会执行回卷栈，JVM直接退出

20、终结器：垃圾回收器对定义了finalize方法的对象，在释放它们后，调用它们的finalize方法，从而保证一些持久化的资源被释放

注：避免使用终结器

# 第八章 线程池的使用

1、需要明确指定执行策略的任务，包括：

1）依赖性任务

2）使用线程封闭机制的任务

3）对响应时间敏感的任务

4）使用ThreadLocal的任务

2、线程饥饿死锁：工作线程数有限的情况下，正在执行的A任务依赖任务B的完成，而任务B在队列中获取不到工作线程。

3、运行时间较长的任务，应限定任务等待资源的时间，而不要无限制的等待。

4、设置线程池的大小：

1）计算密集型：N+1，之所以加1是为了保证有线程暂停时，仍然高CPU利用率

2）IO密集型：2N+1，实际应该根据等待时间与计算时间比值、内存、文件句柄、套接字句柄、数据库连接等去分析

5、ThreadPoolExecutor：为一些Executor提供了基本的实现，线程懒启动，任务提交时启动；可通过构造函数定制，也可以事后通过Setter定制。

6、线程的创建与销毁：基本大小、最大大小、存活时间

1）创建基本大小的线程数量

2）第一次有任务提交时，线程启动

3）当工作队列已满，且未超过线程最大数时，创建新线程

4）当某线程空闲时间超过存活时间，且线程池当前大小超过了基本大小，则该线程被终止

7、ThreadPoolExecutor允许提供BlockingQueue来保存等待执行的任务，排队方法有：无界队列、有界队列、同步移交

注：可用Semaphore将无界队列实现为有界队列；同步移交使用SynchronousQueue避免任务排队，并不是真的队列

8、饱和策略：有界队列已满，且线程数达到最大值后的拒绝策略。通过setRejectedExecutionHandler来设置。

1）终止策略（AbortPolicy）：默认的饱和策略，抛出未检查的RejectedExecutionException

2）抛弃策略（DiscardPolicy）：悄悄抛弃该任务

3）抛弃最旧的策略（DiscardOldestPolicy）：抛弃队列中下一个将被执行的任务，然后提交新的任务到队列

4）调用者运行策略（CallerRunsPolicy）：将任务回退到调用者，在调用者的线程中执行该任务

注：当服务器过载时，调用者运行策略可使系统性能平缓的降低。线程池–>工作队列–>应用程序–>TCP层

9、线程工厂（ThreadFactory）：只定义了一个方法newThread。线程池创建新线程时使用，若要创建定制的线程，则继承ThreadFactory接口，重写newThread即可。

10、通过重写ThreadPoolExecutor的方法可扩展其行为：

1）beforeExecute：一定会调用

2）afterExecute：在beforeExecute中抛异常，或者执行过程抛Error，则不会调用

3）terminated：线程池完成关闭操作时调用

11、递归算法并行化的场景：1）串行循环中的各个迭代操作之间彼此独立；2）每个迭代操作执行的工作量比管理一个新任务时带来的开销更多。

# 第九章 图形用户界面应用程序

略

# 第十章 避免活跃性危险

1、活跃性：一个并发的应用程序能及时执行的能力

2、死锁的原因：

1）每个线程都拥有其他线程需要的资源

2）同时等待其他线程已经拥有的资源

3）在获得所有需要的资源之前都不会放弃已经拥有的资源

3、数据库服务器，在检测到一组事务发生了死锁时，将选择一个牺牲者并放弃这个事务。而java应用程序无法从死锁中恢复过来。

注：数据库服务通过在表示等待关系的有向图中搜索循环来检测死锁

4、如果所有线程以固定的顺序来获得锁，在程序中就不会出现锁顺序死锁问题。通过实例对象的hash值来确定一致的加锁顺序，当哈希值相等时可借助某个对象作为加时赛锁，或者使用资源对象的键值作为唯一标识来确定顺序

5、如果在持有锁时调用某个外部方法，那么将出现活跃性问题。在这个外部方法中可能会获取其他锁（可能会产生死锁），或阻塞时间过长，导致其他线程无法及时获得当前持有的锁。

6、开放调用：在调用某个方法时不要有持有锁的调用。可通过开放调用来避免在相互协作的对象之间产生死锁。在程序中应尽量使用开放调用，与那些在持有锁时调用外部方法的程序相比，更易于依赖于开放调用的程序进行死锁分析。

7、资源死锁：在相同的资源集合上等待时，也会发生死锁；或者线程饥饿死锁。

注：1）资源池通常采用信号量来实现

2）有界线程池、资源池与相互依赖的任务不能一起使用

8、如何避免死锁：

1）每次至多只能获得一个锁

2）获取锁的顺序一致

3）尽量减少潜在的加锁交互数量，尽可能地使用开放调用

4）显式使用Lock类中的定时tryLock功能来代替内置锁机制，可指定一个超时时限

9、如何诊断死锁：

1）代码层面两阶段策略：首先找出在什么地方获取多个锁（这个集合应尽量小），然后对所有这些实例进行全局分析，从而保证它们在整个程序中获取锁的顺序都保持一致

2）Lock类可检测死锁并从死锁中恢复过来

3）通过线程转储信息来分析死锁，包括：各个运行中的线程的栈追踪信息，加锁信息（每个线程持有哪些锁，在哪些栈帧中获得的锁，被阻塞的线程正在等待获取哪一个锁）

注：在生成线程转储之前，JVM将在等待关系图中通过搜索循环来找出死锁，若发现死锁，则获取相应的死锁信息。

10、其他活跃性危险：饥饿、丢失信号、活锁

1）饥饿：线程由于无法访问它需要的资源而不能继续执行

注：Thread API 定义了10个优先级，JVM根据需要将它们映射到系统的调度优先级。通常要避免使用线程优先级，因为这会增加平台依赖性，并可能导致活跃性问题

2）活锁：线程将不断重复执行相同的操作，而且总会失败

注：可增加随机因子或限定重试次数

# 第十一章 性能与可伸缩性

1、资源密集型操作：性能由于某种特定的资源而受到限制的操作

2、线程开销包括：1）线程间协调；2）上下文切换；3）线程创建和销毁；4）线程调度

3、可伸缩性：指当增加计算资源时（CPU、内存、存储容量、IO带宽等），程序的吞吐量或者处理能力相应地增加。

1）运行速度：某个指定的任务单元需要多块才能处理完成

2）处理能力：在计算资源一定的情况下，能完成多少工作

4、Amdahl定律：在增加计算资源的情况下，程序在理论上能够实现最高加速比，这个值取决于程序中可并行组件与串行组件所占的比重。

注：在所有并发程序中都包含一些串行部分

5、上下文切换：

1）应用程序、操作系统、JVM使用一组相同的CPU

2）新线程所需要的数据，缓存缺失

3）线程被阻塞时挂起

6、内存同步：内存栅栏抑制编译器优化操作

注：JVM同步优化：1）锁消除优化；2）锁粒度粗化

7、JVM实现阻塞行为的方式：

1）自旋等待：指通过循环不断地尝试获取锁，直到成功

2）线程挂起：换进换出开销、通知被阻塞线程开销

8、在并发程序中，对可伸缩性最主要的威胁是：独占方式的资源锁

9、锁竞争的影响因素：锁的请求频率和每次持有该锁的时间

注：乘积关系

10、降低锁竞争的方式：

1）减小锁的持有时间，即缩小锁的范围

2）降低锁的请求频率，即减小锁的粒度

a、锁分解：一个锁维护多个对象状态，拆解成一个锁维护一个对象的状态

b、锁分段：对于一组独立对象上的锁进行分解

注：锁分段的劣势：要获取多个锁来实现独占访问将更加困难，并且开销更高。例如：ConcurrentHashMap扩展映射范围以及重新计算键值的散列值要分布到更大的桶集合中时，需要获取分段锁集合中的所有锁。要获取内置锁的一个集合，能采用的唯一方式是递归

3）避免热点域：往往会成为同步的瓶颈

4）放弃使用独占锁，使用并发容器、读写锁、不可变对象以及原子变量

注：ReadWriteLock：读时共享访问，写时独占访问

11、CPU得不到充分利用的原因：负载不充足、IO密集、外部限制、锁竞争

12、对象池技术：在对象池中，对象能被循环使用，而不是由垃圾收集器回收并在需要的时候重新分配

注：通常，对象分配操作的开销比同步的开销更低，应避免使用。

# 第十二章 并发程序的测试

略

第十三章 显示锁

1、Lock提供了一种无条件的、可轮询的、定时的、可中断的锁获取操作，所有加锁和解锁的方法都是显示的。ReentrantLock实现了Lock接口，并提供互斥性与内存可见性，可重入。

1）轮询通过while(true) lock.tryLock()

2）定时通过指定过期时间，可以在带有时限的操作中实现独占加锁的行为

3）lockInterruptibly方法能够在获得锁的同时保持对中断的响应

注：可轮询和可定时可以避免死锁

2、Lock与内置锁对比：

1）内置锁无法中断一个正在等待获取锁的线程

2）内置锁无法实现非阻塞结构的加锁规则

3）ReentrantLock必须在finally中释放锁，因为程序的执行控制离开被保护的代码块时，不会自动清除锁

3、竞争性能是可伸缩性的关键要素：如果有越多的资源被耗费在锁的管理和调度上，那应用程序得到的资源就越少

4、ReentrantLock构造函数中可指定创建一个非公平的锁（默认）还是一个公平的锁，独占锁，可重入

1）公平锁：线程按请求顺序获取

2）非公平锁：发出请求时若此刻锁可用，则跳过等待队列中的线程，直接获取到该锁

注：1）Semaphore中也可选择采用公平或非公平的方式获取资源

2）非公平锁的性能要高于公平锁，因为：在恢复一个被挂起的线程与该线程真正开始运行之间存在着严重的延迟。当持锁时间长或请求锁间隔长，应使用公平锁

5、当需要可定时的、可轮询的、可中断的锁获取操作，公平队列、非块结构的锁时使用ReentrantLock，否则优先使用synchronized。因为ReentrantLock危险性高，并且synchronized是JVM内置属性便于其优化

6、读写锁ReadWriteLock和ReentrantReadWriteLock：其readLock和writeLock是同一个锁，当且仅当并发量高且以读为主时读写锁才会提升性能，因为其实现比互斥锁更复杂

7、读取锁与写入锁交互方式：释放优先、读线程插队、重入性、降级、升级

注：由于升级可能导致死锁，大多数读写锁都不支持升级

# 第十四章 构建自定义的同步工具

1、条件队列：使一组线程（等待线程集合）能够通过某种方式来等待特定的条件变为真，其元素是正在等待的一个个线程

注：每个java对象都可以作为一个条件队列，Object的wait、notify、notifyAll就构成了内部条件队列的API。

2、条件队列实现方式：

1）不满足条件时sleep：响应性差

2）忙等待（自旋等待）：浪费CPU

3）Thread.yield：介于上述两者之间

4）wait和notifyAll配合：CPU效率、上下文切换、响应性上 性价比最高

注：上述四种方式均需要配合在while中使用

3、条件谓词：是使某个操作成为状态依赖操作的前提条件，是由类中各个状态变量构成的表达式。

注：要将与条件队列相关联的条件谓词以及在这些条件谓词上等待的操作都写入文档

4、条件等待中的三元关系：加锁，wait方法，条件谓词。每一次wait调用都会隐士地与特定的条件谓词关联起来。当调用某个特定条件谓词的wait时，调用者必须已经持有与条件队列相关的锁，并且这个锁必须保护着构成条件谓词的状态变量。

注：条件谓词中可能包含多个状态变量，一个条件队列可能与多个条件谓词相关

5、丢失的信号：线程必须等待一个已经为真的条件，但在开始等待之前没有等待条件谓词。

6、每当在等待一个条件时，一定要确保在条件谓词变为真时通过某种方式发出通知。

7、使用notify代替notifyAll的两个条件：

1）所有等待线程的类型都相同：即只有一个条件谓词与条件队列相关，并且每个线程在从wait返回后将执行相同的操作。

2）单进单出：在条件队列上的每次通知，最多只能唤醒一个线程来执行

8、使用显示的Lock和Condition实现一个带有多个条件谓词的并发对象。

1）Lock.newCondition创建一个condition

2）Condition的await、signal、signalAll类似于Object中的玩法

9、AbstractQueuedSynchronizer(AQS)：是一个用于构建锁和同步器的框架，许多同步器都可以通过AQS很容易并且高效地构造出来。其负责管理同步器类中的状态（一个整数状态信息）通过getState、setState、compareAndSetState操作。

注：java.util.concurrent下的同步工具类都是基于AQS实现的

10、AQS同步状态在同步工具类中的使用：

1）ReentrantLock：将同步状态用于保存锁获取操作的次数，并且维护一个owner变量来保存当前所有者线程的标识符

2）Semaphore：将同步状态用于保存当前可用许可的数量

3）CountDownLatch：在同步状态中保存当前计数值

4）FutureTask：将同步状态用于保存任务的状态（正在运行、已完成、已取消）

5）ReentrantReadWriteLock：使用一个16位的状态来表示写入锁的计数，使用另一个16位的状态来表示读取锁的计数。同时内部维护了一个等待线程队列

注：若实现优先级获取锁，需要两个队列

# 第十五章 原子变量与非阻塞同步机制

1、非阻塞算法利用底层的原子机器指令（如比较并交换指令）代替锁来确保数据在并发访问中的一致性。

2、锁的劣势：

1）线程调度开销

2）可伸缩性和活跃性问题

3、比较并交换指令（CAS）：包含3个操作数，内存值V、比较值A、修改值B。当且仅当V值等于A时，CAS才会通过原子方式用新值B来更新V值，否则不会执行任何操作。

注：无论操作是否成功、CAS都会返回内存值V

4、CAS缺点：

1）使调用者处理竞争问题（重试、回退、放弃），而在锁中能自动处理竞争问题（线程在获取锁之前将一直阻塞）

2）需要处理器之间的同步，CPU个数越多，CAS性能越差

3）ABA问题

5、原子变量：直接利用了硬件对并发的支持CAS，分为4组，即标量类、更新器类、数组类、复合变量类

1）标量类：AtomicInteger、AtomicLong、AtomicBoolean、AtomicReference

2）原子数组类：为数组元素提供了volatile类型的访问语义

6、原子变量是一种“更好的volatile类型变量”，相同的内存语义、而且支持原子的更新操作

7、原子变量与锁的性能比较：在中低程度的竞争下，原子变量能提供更高的可伸缩性，而在高强度的竞争下，锁能够更有效地避免竞争。

注：经验法则，在大多数处理器上，在无竞争的锁获取和释放的“快速代码路径”上的开销，大约是CAS开销的两倍。

8、一个线程的失败或挂起不会导致其他线程也失败或挂起，称为非阻塞算法。在算法的每个步骤中都存在某个线程能够执行下去，称为无锁算法。

注：非阻塞算法中不会出现死锁和优先级反转问题，但是可能出现饥饿和活锁问题。

9、创建非阻塞算法的关键：找出如何将原子修改的范围缩小到单个变量上，同时还要维护数据的一致性。

10、CAS的基本使用模式：在更新某个值时存在不确定性，在更新失败时重新尝试。

# 第十六章 Java内存模型

1、在共享内存的多处理器体系架构中，每个处理器拥有自己的缓存，并定期与主内存进行协调。

2、最小保证：允许不同的处理器在任意时刻从同一个存储位置上看到不同的值。

3、内存栅栏：是一些特殊的指令，当需要共享数据时，这些指令能实现额外的存储协调保证

注：java提供了自己的内存模型，通过在适当位置上插入内存栅栏来屏蔽在JMM与底层平台内存模型之间的差异

4、串行一致性：在程序中只存在唯一的操作执行顺序，而不考虑这些操作在何种处理器上执行，并且在每次读取变量时，都能获得在执行序列中（任何处理器）最近写入该变量的值。

注：不要假设它存在

5、各种使操作延迟活着看似乱序执行的不同原因，都可以归为重排序

注：同步将限制编译器、运行时、硬件对内存操作重排序的方式。

6、偏序关系Happens-Before：

1）程序顺序规则：如果程序中操作A在操作B之前，那么线程中操作A将在操作B之前执行

2）监视器锁规则：在同一个监视器锁上，解锁必须在加锁之前完成

3）volatile变量规则：对volatile变量的写入操作必须在对该变量的读操作之前执行

4）线程启动规则：在线程上对Thread.Start的调用必须在该线程中执行任何操作之前执行

5）线程结束规则：线程中的任何操作都必须在其他线程检测到该线程已经结束之前执行

6）中断规则：当一个线程在另一个线程上调用interrupt时，必须在被中断线程检测到interrupt调用之前执行

7）终结器规则：对象的构造函数必须在启动该对象的终结期之前执行完成

8）传递性：A在B前，B在C前，则A在C前

注：如果两个操作之间缺乏Happens-Before关系，那么JVM可以对它们任意地重排序。

7、造成不正确发布的真正原因：“发布一个共享对象”与“另一个线程访问该对象”之间缺少一种Happens-Before排序。如果无法确保发布共享引用的操作在另一个线程加载该共享引用之前执行，那么对新对象引用的写入操作将与对象中各个域的写入操作重排序

注：除了不可变对象以外，使用被另一个线程初始化的对象通常都是不安全的，除非对象的发布操作是在使用该对象的线程开始使用之前执行

8、Happens-Before排序是在内存访问级别上操作的，是一种“并发级汇编语言”；而安全发布的运行级别更接近程序设计

9、双重检查加锁：不能避免重排序。单例中使用，线程可能看到一个仅被部分构造的对象

10、初始化安全性确保，对于被正确构造的对象，所有线程都能看到由构造函数为对象给各个final域设置的正确值；以及通过final域可达的任意变量将同样对于其他线程是可见的。

---------------------

作者：猴子哥哥1024

来源：CSDN

原文：https://blog.csdn.net/qq\_21033663/article/details/79544597

版权声明：本文为博主原创文章，转载请附上博文链接！