# 第二章 线程安全性

线程安全的核心就是对共享的状态的访问的管理；这些共享的对象可能还存在于其他模块中；java中对于此的管理是同步，主要的同步机制的关键字是synchronized，还包括volatile、显示锁、原子变量等。

线程安全性的定义：当线程访问某个类时，这个类始终都能表现出正确的行为，这个类时线程安全的。

无状态对象是线程安全的，因为没有共享变量，变量都是线程私有的。

当具有共享状态时，不恰当的执行顺序可能导致错误的结果，这种现象叫做竞态条件。

常见的竞态条件现象是先检查后执行或者是延迟初始化现象，这种多个操作在一起要避免发生错误，就要弄成原子操作；java.util.concurrent.atomic下面有很多原子变量类，它的所有类内部的方法都是原子操作的。可以代替i++等这种非原子操作。

如果需要协同多个状态，那么线程安全状态的简单组合不能满足要求，因为状态之间的关系并没有线程安全的保证，此时需要用到同步代码块（注意是块、块），synchronized关键字修饰代码块，或者方法。

Synchronized(lock){

代码

}

锁的对象的引用，静态下是Class对象，以及要锁保护的代码块。每个java对象内部都有一个内置锁，进入时获得锁，退出时释放锁；synchronized代码块内重新访问这块代码，并不会产生互斥，因为是在同一个线程内，此时叫做重入；里面有一个计数器，计数重入的个数，就是递归操作算重入。如果不这样的话，递归操作就会产生死锁。

共享变量的保护机制就是在它被访问的所有的地方使用同一个锁来保护，不能是不同的锁，否则就破坏了保护机制。

由于可能在多个地方出现，为了方便管理，将这些共享状态封装在一个单独的对象中，这些状态要是private的，否则，线程可以直接访问这些状态，而不经过锁，所有的操作都进行该对象锁保护；新增方法时，就要都加上synchronized。

代码块可以用同一个锁来保护多个共享状态。

粗粒度的Synchronized会带来性能下降，就需要缩小Synchronized的范围，一个原则就是，原子操作使用锁来保护，将不影响共享状态的操作拆分出去，多个原子操作拆分出去。

# 第三章 对象的共享

同步代码块可以保证互斥与原子性执行，这是对当前线程来说，对于线程交互来说，它确保了内存可见性，内存可见性就是保证内存值一直是有效的，不存在脏数据。

## 3.1 可见性

同步机制保证在每个线程访问共享变量时，值都是有效的，隐晦的说就是修改等操作对其他线程是可见的，直白点说就是，其他线程修改后的有效值能反映到其他线程中，在这个时间点来说，这个值是对其线程有效的，而且当有多个共享状态时，可能发生重排序，就是执行顺序并不是代码中所表现出来的那样，在多线程情况下，就会出现问题，失效值会带来错误的结果。

概念：最低安全性，读取变量时，可能是前面某个线程设置的失效值，但是至少也是符合要求的，这个就是最低安全性，Java的大多数基本数据类型都是最低安全性的，只有64的double与long，不是，因为它俩个的读写是分开成2个32位的操作，从而有可能数据遭到破坏，而不是一个完全值。

加锁，就带表告诉了其他同步线程我的操作，所以保证了内存可见性，前提，这个锁要是同一个锁。

Volatile是一个较弱的同步机制，保证变量不会被重排序（寄存器的原因），不会被缓存在寄存器中，读取时总是返回最新值；加锁机制可以确保可见性与原子性，volatile只保证可见性；volatile只用在某个操作完成、发生中断或者状态的标志。

## 3.2 发布与逸出

发布一个对象的意思就是对象在当前作用域之外使用，最常见的就是返回对象引用，使外部代码持有对象，如果有多处发布，或者外部修改，就需要同步，因为不知道此时发布的对象是否是有效的状态对象，简而言之就是要保证对象的一致性；如果不该发布的对象被发布，称为逸出。

所以返回私有对象就是发布了对象，这种就是逸出，不论什么方式，只要对象能被外部代码访问到就是发布了。

外部方法是指类的接口，因为这种接口方法时公开的，发布的对象不知道外部代码如何调用本对象的方法。这种方法是private或者是final的。

对象逸出，容易造成误用，所以需要封装，封装加入对象引用的规则，对引用对象的行为加以规范化。

发布内部类对象时，外部类对象也会被发布，因为内部类对象含有指向外部类对象的指针。此时，外部类对象在内部类构造的过程中逸出，如果有其他的代码修改外部类对象，那么就是不正确构造；因为此时外部类对象无法保证一致性。

## 3.3 线程封闭

线程只访问自己的数据，这种就是线程封闭，一直是线程安全的；swing大量使用了这种技术；jdbc也是这种技术的。

Ad-hoc线程封闭是指维护线程封闭的职责完全由程序实现来承担。

栈封闭是指线程用到的变量都是线程的局部变量。

ThreadLocal类实现线程封闭性，ThreadLocal用来包装共享变量，里面有set、get等方法，执行get方法时，会第一次执行initialValue方法；这些都是线程安全封装的。

## 3.4 不变性

不可变对象一定是线程安全的；对象创建以后所有状态不能更改，所有域都是final的，对象是正确创建的，this引用没逸出；不可以变对象内部的属性可以是可变属性，只要扩展出来的接口行为完全不能改变对象状态，对象就是不可变对象。

Final关键字，禁止引用被修改；基本类型是值不能被修改，但是引用的对象的状态还是可能被修改。

不可变对象你鞥狗提供一种较弱形式的原子性。

## 3.5 安全发布

安全发布对象，确保对象在多处地方具有一致性。

不可以变对象只要正确构建了，就可以随意发布，不需要同步，就可以保证一致性；

安全发布对象的4种方式：静态初始化函数初始化一个函数引用；volatile;final;或者锁机制；

事实不可变对象也是线程安全的；

可变对象的发布要用锁来保护。

# 第四章 对象的组合

## 4.1 设计线程安全的类

线程安全的类需要包含的要素：构成所有对象状态的变量；找出约束变量不变行的约束条件；建立对象状态的并发访问策略。

分析对象的状态，先从域开始，所有的域的状态极其子状态都是对象的状态；

同步策略定义了并发访问策略，要写为正式文档。保证类的线程安全性，就是保证在并发访问情况下数据的有效性，不能脏读或者覆写等；其状态范围越少，就越容易判断对象的状态；类中属性必然有一些不可变条件；还会包含一些后验条件来判断状态更改是否有效，这些约束需要额外的同步与封装，否则，客户端代码可能会绕过约束使代码处于无效状态；封装多个变量共同组成的不变性条件时，需要加锁。

对象中的方法可能要依赖于对象的状态选择做一些操作，这种选择叫做先验条件，这种方法叫做依赖状态的操作；在多线程情况下，先验条件验完后，可能条件已经改变，所以此时条件已经不满足，所以对先验条件与操作进行原子操作，同时需要一些阻塞机制，让操作等待条件为真，比如生产者与消费者程序。

对象的状态时对象的属性及孩子属性等构成的状态对象树的一部分；在定义那些变量将构成对象的状态时，只考虑对象拥有的数据；对象的很多属性对象都是从客户端代码传进来的，这样，控制权就是共享的，此时就很难保证并发操作，所以这些共享的对象要进行封装。所有的操作都封装成线程安全的。

## 4.2 实例封闭

如果对象不是线程安全的，那么就需要封装对象，再加入加锁机制，通常是为涉及到的方法加锁，那么就可以用一种线程安全的方式来访问非线程安全的对象。

Java类库中有很多线程封闭的例子，就是封装了非线程安全的类，比如很多容器类；Collection.synchronizedList等方法。

这种方式就是Java监视器模式，一般使用私有的专用锁对象而不是对象内置锁；主要原因是：所有锁，可以将锁封装，外面看不到锁。

## 4.3 线程安全性委托

类中的所有属性都是线程安全的，那么整体是不是线程安全的要视情况而定，通常来说，如果整体是线程安全的，就是每个方法的操作都是线程安全的，发布的对象都是线程安全的，那么类的安全性被委托给了其某个属性。

通常，发布final对象不会破坏整体的线程安全性；如果类的某个方法操作的属性之间没有任何必要的约束条件，就是不相关，那么整体的安全性也是可以保障的；如果是相关的，就要进行加锁控制，如果类对其属性没有什么约束条件，那么线程的安全性可以委托给底层的状态变量。

线程中的对象如果没有任何约束，那么发布对象就是线程安全的。

## 4.4 在现有的线程安全类中添加功能

假如现有的线程安全的类不能满足我们的操作需求，如果要新增操作，同时不破坏线程安全性，可以的解决办法有以下3种：

1. 修改源代码，有时没有源代码就不行了；
2. 扩展类，在扩展类中添加新的方法，这种方式侵入性太强；
3. 组合而不是继承，属性是线程安全类，并且是public的，用到新增的操作时，使用新类中的方法，并用synchronized标记；缺点，属性的锁与新类方法的锁不是一个锁，造成了不安全；
4. 3的改进，新增的方法的锁使用属性的锁；保持一致性；
5. 4的改进，新类完全代理所有的属性的操作；并用自己的锁来达到线程安全性，相当于新开发了一个类。

## 4.5 将同步策略文档化

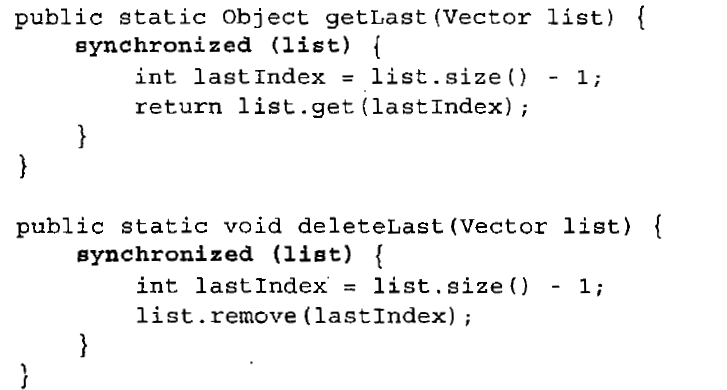
# 第五章 基础构建模块

主要是java1.5与java1.6引入的并发构建新模块。

## 5.1 同步容器类

Vector与Hashtable，这些类是由Collections.synchronizedXxx等工厂方法创建的；实现线程安全的方式是：将底层实现封装起来，对每个公有的方法都进行同步。

虽然这些容器类都是线程安全的，但是在这些容器类上的一些复合操作就不是了，比如遍历、若没有就添加、跳转等；这些复合操作是由多个原子操作组成，需要客户端进行加锁控制，而且这个锁是要与容器类对象的锁相同，保证在同一个锁之下的控制，比如客户端代码：



需要注意的一点是：当使用Iterator时，遍历时没有进行同步控制的，如果容器在迭代中被修改（迭代器维持一个计数器），会抛出一个ConcurrentModificationException异常；如果每次迭代的操作都要进行很长时间，那么同步控制有可能导致竞争激烈并产生死锁；如果希望避免，可以采用容器克隆的方法，在克隆的容器上迭代，但是受限于容器的大小等开销。

有时候迭代器操作不是客户端代码，认识底层代码，此时就是隐藏迭代器，比如容器的toString方法，此时还是会在多线程下抛出异常。

## 5.2 并发容器

同步容器影响性能，所以设计了并发容器；ConcurrentHashMap，CopyonWriteArrayList、ConcurrentSkipListMap、ConcurrentSkipListSet等都是新增的并发容器，ConcurrentMap接口也有复合操作的并发支持；Queue与BlockingQueue是新增的类型，Queue的实现有ConcurrentLinkedQueue与PriorityQueue，这些容器在操作上不会阻塞，如果不满足会返回空；BlockingQueue是阻塞类，在并发下，会阻塞操作直到可以进行操作；

相比于HashMap等严格的同步容器不同，ConcurrentHashMap放开了一些不重要方法的同步锁，使得多个方法同时操作成为可能，使用了一种更细的加锁机制，称为分段锁；这是一种权衡，能适用大部分的场景。

ConcurrentMap接口提供了常见的复合操作接口，这些都是原子性的。

CopyonWriteArrayList也放开了一些方法的并发访问，获得了一定的并发性；每次修改时，复制一个副本，所有的读取操作都会获得读取时生成的一个快照副本，这个副本没有改变。数据较多时，需要考虑复制的开销。

## 5.3 阻塞队列和生产者-消费者模式

生产者与消费者就是阻塞队列，生产者与消费者不知道对方，只关注自己的任务；Executor就是这个模式的体现。

BlockingQueue有多种实现，LinkedBlockingQueue与ArrayBlockingQueue是FIFO队列，PriorityBlockingQueue是优先级队列，还有SynchronousQueue队列。

阻塞队列传递的对象是线程安全的，因为每次对象的所有权只是进行串行的转移，在对象层面没有并发的访问，并发发生在队列层。

Deque与BlockingDeque。

## 5.4 阻塞方法与中断方法

线程被阻塞，等待某事件结束，重回Runnable状态；BlockingQueue的一些方法都会抛出InterruptedException，主要是阻塞的方法在阻塞时发出中断请求才会出现这个异常，所以发出这种异常的方法都是阻塞方法；interrupt方法中断线程或者检测线程的状态；中断某线程只是告诉这个线程在其可以停止的地方停下来，这个线程完全可以根据系统的状态不停下来；客户端代码调用阻塞方法，客户端代码也变成阻塞方法；当抛出InterruptedException时，客户端代码可以选择抛出或者捕获（在单独的线程中），调用interrupt恢复interrupt的状态；因为阻塞时中断无法设置interrupt的状态。

## 5.5 同步工具类

同步工具类是根据自身的状态协调线程的控制流的类，比如阻塞队列；还有信号量、栅栏、闭锁等。

同步工具类也是普通的java对象，里面有表示持有同步工具类对象的线程的控制流的状态以及对状态操作的一些方法。

闭锁可以延迟线程执行时间，是一种简单的同步方式，里面有个计数器，await()方法循环检测计数器值，为0跳出（此处阻塞），countDown递减计数器；CountDownLatch就是一个闭锁类。

闭锁适用的场景是等待所有事件的发生后再进行下一步的操作，比如资源初始化、依赖服务都得到启动等。

FutureTask也可以作为闭锁，FutureTask实现了Runnable, Future2个接口，表示一种可生成结果的计算，Future本身是用来包装计算的，比如Callable、Runnable，里面有一个状态state表示FutureTask的状态（NEW、COMPLETING、NORMAL、EXCEPTIONAL、CANCELLED….）等，可以简单理解为3种状态：等待运行、运行、运行完成。Future.get()可以获得计算的结果，如果还没结果，将会阻塞等待结果。Callable等会抛出受检查异常，或者未检查异常，都会封装到一个ExecutionException异常中。

计数信号量（Counting Semaphore）是阻塞队列的另一种版本，对象数据量的存储于同步机制是分开的，信号量表示剩余的队列空间，acquire()是否有剩余的空间，release()释放，二值信号量完成了互斥操作；简而言之，信号量用来控制某个特定资源的操作数量，比如资源池或者对容器施加边界。

栅栏跟闭锁有些类似，CycliciBarrier类是栅栏；跟闭锁的区别：

1. 调用await()增加计数器；
2. 栅栏是多个线程之间等待的机制，所有线程都到达了这个位置，才能继续向下执行，闭锁是等待某个事件发生，这个事件的发生看的是countDown的操作；作用的形式是不同的。

## 5.6 构建高效且可伸缩的结果缓存

缓存是普遍存在的，并且缓存可以降低延迟，但是会消耗较多的内存。代码主要是书上，可以看书。

# 第六章 任务执行

并发应用程序都是围绕任务的，任务是一个离散的工作单元。

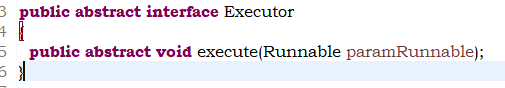
## 6.1 在线程中执行任务

任务就是一小块的处理单元，围绕处理单元编程时，首要的是要找出任务的边界，理想情况是任务之间相互独立；这样就可以并发进行；web服务器等选择已客户请求为一个任务边界。

串行执行任务比较简单，但是吞吐率比较低；接受一个请求就创建一个线程比较合理，但是这样也有问题，当创建大量线程时也会带来较低的处理效率；线程的生命周期的开销很高，创建与销毁都需要时间与资源，超过CPU数量的多余线程会占用内存，竞争产生性能开销，所以需要对线程的数量进行限制。

## 6.2 Executor框架

使用有界队列防止线程无限增长，Java的类库提供了Executor框架可以完成类似的功能，也叫做线程池；Excutor是一个简单的接口。



这种设计，可以在executor方法内部创建线程，执行任务；可以实现异步任务机制；使得任务的提交过程与执行解耦；基于生产者与消费者模式。

如果想要改变任务的执行策略，就改变线程池的实现方式就可以了，在一个程序中，一处改变就使得所有任务的执行方式都获得了改变，比前面的编码更为方便；线程池的执行策略定义了任务执行的What、Where、how、when；在什么线程中执行任务，按照什么顺序执行，有多少能并发实行，队列中有多少等待等等。

线程池（工作队列与一组工作线程）与工作队列有关；工作队列保存了所有待执行的线程，工作者线程从里面取出执行线程；线程池管理了线程的生命周期，往往线程不需要实时创建，而是之前已经存在，节省了创建线程的开销。Java中使用Executors静态类的一些方法创建线程池：newFixedThreadPool，定义了最大线程规模，每来一个任务就创建线程，直到最大，不变；newCachedThreadPool，规模不定，超出的回收，不够补线程；newSingleThreadPool，创建一个线程顺序执行；newScheduledThreadPool，创建固定长度线程池，以延时或者定时的方式执行任务。

Executor的扩展接口ExecutorService提供了一些用于管理Executor与其中的任务的生命周期的一些关闭的方法，shutdown()会等待所有的任务执行完再挂你Executor，shutdownNow()会取消执行中的任务，然后关闭。

Timer类管理延迟与周期任务，存在缺陷（所有定时任务只会创建一个线程），使用sheduledThreadPoolExecutor代替。

## 6.3 找出可利用的并行性

Callable有返回结果的任务，Future代表Callable任务的生命周期，可以获得结果并根据线程所处的状态抛出异常等。ExecutorService的submit方法返回Future，还可以用FutureTask得到Future。异构任务并行化获得性能提升比较难，取决于速度最慢的任务速度，同构任务的并行执行能获得较大的性能提升；CompletionService融合了Executor与BlockingQueue，执行的结果会放到队列中，取队列中的结果就可以了，适合批量任务的获取结果。Future的get方法可以执行等待时间，超出后会抛出异常，可以在异常处理中取消任务。

# 第七章 取消与关闭

任务或者线程需要提前关闭，此时Java采用协商制，即中断，怎么停止由线程决定，因为线程会在停止前达到一个共享数据的一致性状态，确保程序完善的处理取消与关闭。

## 7.1 任务取消

Java任务取消不能立即执行，需要协商；可以在线程中设置一个标志，由外部程序控制这个标志，线程循环检测标志，置位就退出；这种方法的缺点是线程内操作阻塞时，没办法及时退出；一个可取消的任务必须拥有取消策略，这个取消策略定义了How（外部代码怎么取消任务）、When（任务在何时检查是否有取消请求）、What（响应取消请求前后，应该执行哪些操作）。Java API提供了中断操作，就是一种取消任务的方式，中断会告诉线程取消当前执行的任务，如果任务中没有对中断的检测，那么任务将不响应中断，在线程层面来说，中断可解决线程阻塞小的任务取消下的问题，sleep、wait等阻塞方法执行时会不断检测中断标志，然后抛出中断异常；中断是取消任务的最合理的方式；任务有取消策略，线程有中断策略（规定如何解释某个中断请求，英爱做哪些工作，哪些工作是原子操作，多块的速度响应中断），最合理的中断策略是某种形式的线程级取消操作或者服务级取消操作：尽快退出、完成清理，通知线程所有者线程结束；任务与线程对中断的反应是不同的，但是中断意味着取消当前任务与关闭工作者线程2个操作；任务往往不是在我们自己创建的线程中执行，有时是托管给线程池，此时的线程中断策略由线程池决定，任务中的阻塞代码往往只是向外抛出中断异常，中断只能有线程所有者进行，比如线程池，所有者可以将线程的关闭策略封装到某个方法中，如果是自己构建的线程，自己操作即可；任务响应中断的2种方式：抛出InterruptedException或者重新设置中断状态；只有实现了线程中断策略的代码才可以屏蔽中断请求，常规情况下不能屏蔽中断请求；线程池中单一任务的取消，由Future对象的cancel方法处理，线程池有线程池的关闭策略，此时会关闭所有正在执行的任务；有些阻塞方法并不能响应中断，及时设置了中断，也不会抛出InterruptedException；比如Socket I/O操作（可以关闭底层的套接字中断）、同步I/O、异步I/O，获取某个内置锁阻塞。newTaskFor封装非标准的取消。

## 7.2 停止基于线程的服务（线程池）

拥有线程的服务生命周期往往比程序长，程序终止后，线程要终止执行，但是只能有线程的所有者，即线程池终止，ExecutorService提供了shutdown与shutdownNow2个方法来实现线程池的终止。实例程序为日志服务，采用可阻塞队列存储消息；使用一些同步机制确保原子操作，解决生产者与消费者之间存在的一些问题。

Shutdown方法正常关闭，shutdownNow强行关闭，中断任务，返回还未执行的任务；所有权链从上到下管理下层的生命周期；简单来说就是封装，简化。

毒丸方式终止线程服务生命周期，也就是在队列中放入一个标志对象，标志线程要终止。

## 7.3 处理非正常的线程终止

线程异常终止，不会通知程序，可能只是打印一些栈追踪信息，如果是runtimeException异常，更加难以追踪；对调用的代码越不熟悉，对代码的行为越要保持怀疑线程应该在try块中调用任务，可以实现一个UncaughtExceptionHandler异常处理器，当线程发生未捕获异常退出时，JVM会把它交给异常处理器处理，异常处理器可以做一些打印日志的操作；为线程池提供统一的异常处理器，ThreadPoolExecutor提供了Threadfactory参数的构造方法，异常处理器封装在ThreadFactory中；只有execute提交的任务的异常能被异常处理器处理，submit的异常作为返回结果返回。

## 7.4 JVM关闭

System.exit与Runtime.halt关闭JVM。在正常关闭中，JVM首先调用关闭钩子线程；用于实现关闭时的清理工作；普通线程与守护线程。

# 第八章 线程池的使用

## 8.1 任务与执行策略之间的隐性耦合

Executor框架将任务的提交与任务的执行策略解耦，但是一些特殊的任务不能使用通用的执行策略，比如依赖性任务，使用线程封闭机制的任务，对响应时间敏感的任务，ThreadLocal任务；只有任务是同类型的且相互独立，线程池性能达到最佳；依赖性任务可能会产生线程死锁。

## 8.2 设置线程池的大小

Runtime类可以得到CPU的数量；线程池过大，线程切换会比较频繁，过小，硬件资源得不到合理利用；线程池的大小依赖计算环境、资源预算、与任务的特性，CPU、内存、计算密集型、IO密集；对于计算密集型线程池大小为Ncpu+1；I/O密集型，线程池应该更大。稀缺资源的数量也制约线程池的大小。

## 8.3 配置ThreadPoolExecutor

可以通过ThreadPoolExecutor的构造函数定制线程池。

* + - 1. 基本大小（队列满，才会扩），最大大小，存活时间：
      2. 无限制的创建线程会导致不稳定，所以线程池内部保有存储任务的队列，BlockingQueue，排队的方法：无界队列、有界队列、同步移交；固定大小的线程池使用无界队列，但是会无限制增加，最好使用有界队列，但是队列填满也无法处理（饱和策略），对于很大的线程池，可以使用同步移交（synchronousQueue）避免排队；直接将任务移交线程，前提条件是必须有一个工作者线程等待，如果还没到最大大小，那么创建一个线程处理任务，否则，拒绝该任务的执行。
      3. 队列填满，饱和策略发挥作用，setRejectedExecutionHandler来修改，JDK提供的饱和策略：AbortPolicy（抛出异常）、CallerRunsPolicy、DiscardPolicy（抛弃任务）、DiscardOldestPolicy（抛弃下一个将被执行的任务，新的加进去），CallerRunsPolicy实现一种调解机制，将任务退回到提交任务的主线程中去执行，防止主线程再提交新的任务，也可以用信号量来控制控制任务的提交。也能实现同样的功能。
      4. 线程池创建线程时，采用的是一个线程工厂方法，如果想要定制化的生成线程，可以实现线程工厂ThreadFactory。
      5. 调用构造函数后，ThreadPoolExecutor接口可以实现对一些属性的更改，Executor与ExecutorService都不行，Executor有个unconfigurableExecutorService方法，会对线程池进行包装，只返回ExecutorService接口，这样会防止对线程池策略的修改。

## 8.4 扩展ThreadPoolExecutor

ThreadPoolExecutor提供了：beforeExecute、AfterExecute、terminated3个方法可以在子类中改写，beforeExecute、AfterExecute分别在任务执行前，执行后由任务线程调用，线程池关闭时，调用terminated方法；

## 8.5 递归算法的并行化

循环中包含了密集计算或者可能阻塞的IO操作，只要每次迭代都是独立的，那么串行的迭代操作可以转化为并行的；将迭代操作封装到任务中，送入线程池中去执行即可；如果要等待所有任务的结果，使用invokeAll方法并用CompletionService获得结果。

递归也可以并行化，只要递归的方法不需要返回值来决定下一次的操作就可以，比如深度遍历二叉树；遍历过程是串行的，保证得到数据的顺序不会乱掉，但是执行的计算的过程是放到另外一个线程中去的。

# 第十章 避免活跃性危险

安全性与活跃性之间存在某种制衡，过度加锁会导致顺序死锁，线程池信号量等限制资源的使用会导致资源死锁。

## 10.1 死锁

哲学家进餐问题，每个人拥有其他人所需要的资源，有需要其他人占有的资源，同时在获得资源前，不会释放占有的资源，那么会产生死锁，多个线程存在环路的锁依赖关系就会产生死锁。数据库系统处理事务死锁的方法是选择其中一个放弃，打破环路。

多线程的锁的环路依赖导致死锁，实际是多线程的乱序执行导致的获取锁的顺序是不一致的，当如果指定一个获取锁的顺序，那么不会产生死锁。

指定获取锁顺序的方法，比较处理资源的hash值，比如System.identityHashCode，如果散列冲突，可以设置一个单独的额外的锁，作为顶层锁，获取它后才能执行有效的里面的操作，但这样相当于只有一个锁，限制了并发性，如果资源有唯一性的键值，那么肯定可以比较锁的顺序。有时候锁的环路依赖并不十分明显，比如锁在不同的方法中，方法调用形成依赖关系，此时也可能发生死锁问题。

调用外部方法发生的死锁，可以通过开放调用的方式来解决，开放调用就是调用方法时不在持有锁的状态，与封装机制类似，其实就是缩小同步区域的范围，把外部方法调用放到临界区外面，但这有可能会带来原子性问题，此时可以采用让单线程执行开放调用，其他线程都碰到有锁的版本。

## 10.2 死锁的避免与诊断

避免的方法就是定义锁的获取顺序并严格遵循；2阶段策略检查死锁，找出哪些地方将会获取多个锁，分析顺序；转变为开放调用。

Lock类中的tryLock是一个定时锁，可以代替内置锁机制，可以指定超时时间，超时后会返回一个失败信息；线程转储可以帮助识别死锁；线程转储包含线程的各种信息，有加锁信息；所以，JVM在生成转储文件时，会分析加锁信息，找出死锁及相关的线程及在程序中的位置；SIGQUIT信号生成转储文件。

## 10.3 其他活跃性危险

饥饿：资源一直被别人持有，自己得不到执行；线程优先级只能作为线程优先的参考，不一定准确，10个优先级会映射到操作系统系统的调度优先级；与平台相关；尽量不改变线程的优先级，仿制与平台相关。

糟糕的响应性：提高前台程序的优先级，降低后台线程的优先级执行；

活锁：

# 第11章 性能与可伸缩性

线程能够提高程序运行效率，增加响应性；提升性能可能造成安全性与活跃性上风险；收腰任务是要保证程序能够正常运行。

## 11.1 对性能的思考

当操作性能由于某种特定的资源受到限制时，该操作称为该资源密集型操作；使用多线程会带来多线程方面的开销；如果设计不合理，会比单线程的性能差，为了获得更好的性能：高效利用现有资源、尽可能的利用新的资源；哪块资源称为瓶颈就添加哪块资源。

应用程序的性能指标：运行速度类（服务时间、等待时间）、处理能力类（）吞吐量；可伸缩性是指增加资源时，程序的处理能力线性增加。可伸缩性的设计与传统性能调优相悖，可伸缩性设计是增加程序的并行处理，而不是单一的调整处理时间；单一程序处理速度会达到极限，进一步提高处理能力很难；此时就需要拆分，增加更多的资源以方便大批量的数据处理。只有有明确的优化需求的时候才去优化程序，否则会有风险。

## 11.2 Amdahl定律

程序都是由并行与串行部分组成的，Amdhl定律就是：程序的加速比取决于程序中可并行组件与串行组件所占的比重：

其中F是串行执行的部分，N是核数。

## 11.3 线程引入的开销

多个线程的调度协调需要一定的开销，使用对线程的前提是提升的性能比开销大：

1. 上下文切换的开销，主要是因为线程调度，可能会引起缓存缺失，经常阻塞也会影响吞吐量，上下文啊开销的相当于几微秒；vmstat命令报告上下文切换次数，如果内核占用率较高，表明切换频繁；
2. 内存同步开销，synchronized与volatile步骤中，可能会使用到内存栅栏，内存栅栏会禁止硬件缓存等，而且栅栏里面的步骤都是不能重排序的，JVM会通过优化去除掉不会发生竞争的锁，JVM可以进行逸出分析，线程里面的引用的对象没在外部使用时，都算线程本地变量，可以除掉锁，比如集合类的锁；也可以进行粗粒度优化，将多个锁操作合并为一个。
3. 阻塞，非竞争的同步在JVM中处理；竞争的同步需要操作系统的家介入；当在锁上竞争时，失败的线程转入阻塞状态，此时可以选择1.自旋等待（循环尝试获得锁）；2.交给操作系统挂起；大多数JVM选择将线程挂起，这包含了2次上下文切换的开销。

## 11.4 减少锁的竞争

减少锁的竞争能够提高性能与可伸缩性；访问独占锁保护的资源时，实际采用的是串行的方式，锁上发生竞争的2个因素：锁的请求频率、每次持有该锁的时间；乘积小，不会发生竞争，乘积大，就非常容易竞争；降低锁的竞争程度的3种方式：

1.减少锁的持有时间；2.降低锁的请求频率；3.使用带有协调机制的独占锁，这些机制允许更高的并发性。

1. 缩小锁的范围，降低发生竞争的可能性的方法是尽可能缩短锁的持有时间，将与锁无关的代码移除同步代码块；
2. 另一种减少锁的持有时间的方式是降低线程请求锁的频率；可以通过锁分解或者锁分段的方式实现；采用相互独立的锁来保护独立的状态变量；减少锁操作的粒度；
3. 锁分段，锁分解技术在单个对象上的进一步分解，通常都是应用在集合上，每个锁保护对象内的某一部分区域，比如ConcurrentHashMap；实现中的数组上有16个锁；对于锁的请求能够减少到1/16；能支持多达16个并发的写入器；锁分段技术的劣势就是独占访问对象很困难，因为要获得所有的分段锁；
4. 避免热点域，原子操作很难降低锁的粒度；缓存中间的结果往往陷入热点域问题；
5. 一些替代独占域的方法，放弃使用独占锁，使用并发容器、读写锁、不可变对象或者原子变量；ReadWriteLock支持多读一写的并发；原子变量可以解决热点域问题；
6. 检测CPU的利用率，CPU利用率低时，可能需要增加负载，是否是I/O密集型应用；是否依赖外部限制，比如数据库或者网络；是否存在过度的锁竞争，比如通过转储文件分析；
7. 对象池技术因为涉及到并发需要加索控制多线程的访问，所以伸缩性不如直接在线程本地创建对象来的快。

## 11.5 比较Map的性能

ConcurrentHashMap的性能比同步额HashMap要好；同步的HashMap有一个独占锁，ConcurrentHashMap对读操作不加锁；写入操作域其他的一些不要同步的读操作上加了锁分段技术，所以在多线程的情况下，性能更高，避免了竞争性。

## 11.6 减少上下文切换的开销

# 第12章 并发程序的测试

串行程序于并发程序有着相同的设计原则与设计模式；并发测试分为安全性测试与活跃性测试与性能测试。

## 12.1 正确性测试

为并发类设计单元测试时，需要执行与测试串行类相同的分析，找出需要检查的不变性条件与后验条件；下面的示例创建了一个有界缓存。定义了可阻塞的put与take方法。



1. 基本的单元测试，测试一些在串行情况下的必要条件也称为不变性条件；
2. 对阻塞操作的测试，使用Thread继承开启新的线程，组合使用InterruptException异常与join方法来做测试阻塞情况。
3. 安全性测试：测试竞争的并发性，模拟多客户端。
4. 资源管理的测试，使用内存检查工具测试类的资源占用的情况，是否有内存泄漏；
5. 使用回调，
6. 产生更多的交替操作，

## 12.2 性能测试

性能测试与功能测试的目标不同：1.性能测试测试典型的端到端性能，2.根据经验值调整不同的限值。

## 12.3 避免性能测试的陷阱

## 12.4 其他的测试方法

# 第13章 显示锁

协调共享对象状态访问的3种方法：synchronized、volatile、ReentrantLock。

## 13.1 Lock与ReentrantLock