# Multithread

# 基础

## 什么是线程？

说道线程我们就需要拿其和进程进行比较，那么到底什么是线程，什么是进程呢？

进程与线程的一个简单解释[1]:

**计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。**

由于工厂电力有限，一次只能给一个车间使用。这里的车间就是进程，是工厂（CPU）调度的基本单位，CPU一次只能运行一个进程，即车间是轮流运行的

一个车间里，可以有很多工人。他们协同完成一个任务。线程就好比车间里的工人。一个进程可以包括多个线程。

车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。**他运算调度的最小单位**

## 什么是线程安全

当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替执行，并且不需要额外的同步及在调用方代码不必作其他的协调，这个类的行为任然是正确的，那么称这个类是线程安全的。

编写正确的并发程序的关键在于对共享的，可变的状态进行访问管理

一个机器只能同时被一个人操控，如果被多个人操控了，那么大家就会手忙脚乱，产生一些比较坏的结果，所以我们要保证一个人操纵一台机器，这就是线程安全。那么如何保证线程安全，通常有三种方法：

1. 不要跨线程共享变量（禁止两个人同时使用机器）
2. 使状态变量变为不可变的（机器执行过程被固定了，即使两个人操作机器也不影响机器的正常运行）
3. 在任何访问状态变量的时候使用同步（我们可以给这个机器加锁，只有有钥匙的人才能进入机器工作）

通常来说，我们应该尽量保证较少的机器可以被操控，即不要暴露过多的变量，面向对象技术可以帮助我们做到这一点（封装和数据隐藏），封装的越好，程序就越容易实现线程安全

## 原子性

一个操作如果是单独的，不可分割的，那么我们就可以称之具有原子性，不具有原子性的操作会产生风险，Java中++不具有原子性，他是需要读改写的复合操作，结果的状态衍生自它先前的状态。

那么是什么原因导致不具有原子性的操作产生错误的结果呢，我们再将其抽象一点，++需要一定的时序，需要检查再运行（check then act），就好比，一个机器是专门安装轮子的，例如你知道车子上有3个轮胎，这时候你需要再取一个轮胎，操作机器装上去，但是可能等你去取轮胎的时候，另一个人装了一个轮胎，等你回去再装的时候，你得知的三个轮胎信息已经失效。使用潜在的过期观察值来做决策或执行计算。这种竞争条件被称作检查在运行。

另一个很好的例子就是lazy initialization。

那么能不能给予一个复合操作原子性呢？

## 假设有操作A和B，如果从执行A的线程的角度看，当其他线程执行B时，要么B全部执行完成，要么一点都没有执行，这样A和B互为原子操作

## 什么是锁

Java中提供了synchronized关键字来使某一块具有原子性， Java中每个对象都可以扮演一个锁的角色，这些内置的锁被称为内部锁或监视器锁。线程进入synchronized块之前都会获得锁，正常退出或者异常退出都会释放锁。内部锁也是互斥锁，意味着至多只有一个线程可以拥有锁。 但是当线程尝试获取自己占用的锁时候，会成功。JVM会记录线程进入的次数

## 可见性

上面我们避免了一个线程修改其他线程正在使用对象的状态，现在我们需要保证一个状态被修改之后，其他线程能够正确的看见。

比如通过关灯来通知一个操作机器的工人下班，但是工人带着墨镜，所以他永远也看不到灯关了，这时候灯对于工人就不具有可见性。

所以对于一个通过设置变量flag的方式来改变另一个线程的状态，通常是不可行的，由于重排序的作用，会导致变量的值提前读入线程中保持不变。

同步是可以保证可见性的。当线程A执行一个同步块时，线程B也随后进入了被同一个锁监视的同步块中，这时可以保证，在锁释放之前对A可见的变量的值，B获得锁之后同样是可见的。

java语言提供了其他选择：volatile变量，当一个变量声明为volatile类型后，编译器运行时会监视这个变量：它是共享的，不会被重排序。通常我们只使用其当做完成，中断的状态标记，但不足以使得++这样的操作具有原子性。因此**加锁可以保证可见性与原子性，volatile变量只能保证可见性**

**数据的封装：**将数据封装在对象内部，对把数据的访问限制在对象的方法上，更易确保线程在访问数据时总能获得正确的锁。被限制的对象一定不能逸出到它的期望可用范围之外。

**数据的限制：**限制性使构造线程安全的类变得更容易，因为类的状态被限制后，分析它的线程安全性时，就不必检查完整的程序。

## 同步容器

尽管vector本身是线程安全的，但对于一些复合操作，vector又不是线程安全的了，例如：

**public static** Object removeLast(Vector list){  
 **int** lastIndex=list.size()-1;  
 **return** list.remove(lastIndex);  
}

多线程下，remove掉的可能是已经过期的lastIndex了

## 并发容器

Java 5.0 通过提供几种并发容器类来改进同步容器，同步容器通过对容器的所有状态进行串行访问，从而实现了它们的线程安全。这样做的代价是削弱了并发性，当多个线程共同竞争容器级的锁时，吞吐量就会降低。

ConcurrentHashMap和HashMap一样是一个哈希表，但是它使用完全不同的锁策略，可以提供更好的并发性和可伸缩性。在ConcurrentHashMap以前，程序使用一个公共锁同步每一个方法，并严格地限制只能有一个线程可以同时访问容器。而ConcurrentHashMap使用一个更加细化的锁机制，名叫**分离锁**。这个机制允许更深层次的共享访问。任意数量的读线程可以并发访问Map，读者和写者也可以并发访问Map，并且有限数量的写线程还可以并发修改Map。结果是，为并发访问带来更高的吞吐量，同时几乎没有损失单个线程访问的性能。

## 线程池

1. **newFixedThreadPool**

**return new** ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>());

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

2、**newCachedThreadPool**

**return new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,  
 60L, TimeUnit.***SECONDS***,  
 **new** SynchronousQueue<Runnable>());

创建一个可缓存的线程池。这种类型的线程池特点是：

1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

2).如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

这个线程池使用了**synchronousqueue**，适用于庞 大或者无限的池，将任务直接从生产者交给工作线程。Synchronous 并不是一个真正的队列，而是一种管理直接在线程间移交信息的机制。为了把一个元素放入到synchronousqueue中，必须有另一个线程正在等待接受移交的任务。如果没有这样一个线程，只要当前池的大小还小于最大值，ThreadPoolExcueter就会创建一个新的线程了；否则根据饱和策略，任务会被拒绝，这种方法更为高效，因为任务不必放置到队列中，就可以立即交由即将执行的线程处理

1. **newSingleThreadExecutor**

**return new** FinalizableDelegatedExecutorService  
 (**new** ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>()));

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它，保证顺序执行(我觉得这点是它的特色)。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的 。

1. **newScheduleThreadPool**

**super**(corePoolSize, Integer.***MAX\_VALUE***,  
 ***DEFAULT\_KEEPALIVE\_MILLIS***, ***MILLISECONDS***,  
 **new** DelayedWorkQueue());

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

总结： 一.FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

二．CachedThreadPool的特点就是在线程池空闲时，即线程池中没有可运行任务时，它会释放工作线程，从而释放工作线程所占用的资源。但是，但当出现新任务时，又要创建一新的工作线程，又要一定的系统开销。并且，在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

核心池大小（core pool size），最大池的大小（maximum pool size）和存活时间（keep alive time）共同管理着线程的创建与销毁。

**饱和策略**

如果线程池满了， 那么默认会执行**abort**策略，抛出未检查的异常

**discard**会默认放弃这个任务

**discard-oldest**策略选择丢弃接下来会执行的任务

**caller-runs**不会抛异常，而是直接把任务推回高调用者那里，然后调用在调用excutor的线程中直接执行，通常就是主线程

[1] <http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html>