# 并发

指在一个时间段内有多个进程在执行，只不过在人的角度看，因为这个计算机角度的时间实在是太短暂了，人根本就感受不到是多个进程，看起来像是同时进行，这种是并发。

# 并行

并行指的是在同一时刻有多个进程在同时执行。

# 并发和并行的区别

一个是时间段内发生的，一个是某一时刻发生的，如果是在只有一个CPU的情况下，是无法实现并行的，因为同一时刻只能有一个进程被调度执行，如果此时同时要执行其他进程则必须上下文切换，这种只能称之为并发，而如果是多个CPU的情况下，就可以同时调度多个进程，这种就可以称之为并行。从这里我们可以总结出并发和并行的差异，首先粒度不一样，并发针对的是时间段，并行针对的是时间点，其次行为也不一样，并发的动作侧重于处理行为，并行的动作侧重于执行行为，只不过人的视角和机器的视角差异导致看起来都是同时执行的。

# 高并发

高并发是指可以让软件系统在一段时间内能够处理大量的请求。比如每秒钟可以完成10万个请求，这是互联网系统的一个重要的特征。不像并发说的是“处理”，并行说的是“执行”，高并发说的是最终效果。只要能达到效果，不管怎么实现都行。因此极端一点高并发甚至并不一定需要并行，只要处理速度快的足够满足要求就可以。如启动一个nginx的OS进程，它只能用到一个CPU核心，也就不可能并行。但是他如果每秒能处理10万个请求，而业务需求只要求8万个请求就可以了，那么这个单进程的nginx本身就算高并发了。通过这段话我们能明白高并发是最终的结果，为了实现这个结果，技术上可能会用到非常多的技术方案，这些技术方案大量应用各种并发的集中人类智慧的各种方法，并尽可能的并行。

# 进程

定义：正在进行中的程序，其实进程就是一个应用程序运行时的内存分配空间。

# 线程

## 基本概念

进程中一个程序执行控制单元，一条执行路径。进程负责的是应用程序空间的标识，线程负责的是应用程序的执行顺序。一个进程至少有一个线程在运行，当一个进程中出现多个线程时，就称这个应用程序是多线程应用程序。每个线程在栈区中都有自己的执行空间，自己的方法区、自己的变量。jvm在启动时，首先有一个主线程，负责程序的执行，调用的是main函数。主线程执行的代码都在main方法中。当产生垃圾时，回收垃圾的动作是不需要主线程来完成的，因为这样会出现主线程中的代码停止执行，会去运行垃圾回收器代码，效率较低，所以会由单独一个线程来负责垃圾回收。

## 多线程的上下文切换

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 CPU 执行权的线程的过程。

## 调度策略

线程调度器选择优先级最高的线程运行，但是如果发生以下情况，就会终止线程的运行:

1、线程体中调用了 yield 方法让出了对 cpu 的占用权利。

2、线程体中调用了 sleep 方法使线程进入睡眠状态。

3、线程由于 IO 操作受到阻塞。

4、另外一个更高优先级线程的出现。

5、在支持时间片的系统中，该线程的时间片用完。

## 多线程的价值

### 1、发挥多核CPU的优势

多线程可以真正发挥出多核 CPU 的优势来，达到充分利用 CPU 的目的，采用多线程的方式去同时完成几件事情而不互相千扰。

### 2、防止阻塞

从程序运行效率的角度来看，单核 CPU 不但不会发挥出多线程的优势，反而会因为在单核 CPU 上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核 CPU 我们还是要应用多线程，就是为了防止阻塞。试想如果单核 CPU 使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

### 3、便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务 A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务 A 分解成几个小任务，任务 B、任务 C、任务 D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

## 多线程使用的目的

### 1、吞吐量

做WEB容器帮你做了多线程，但是它只能帮你做请求层面的，简单的说，就是一个请求一个线程(如struts2，是多线程的，每个客户端请求创建一个实例，保证线程安全)，或多个请求一个线程，如果是单线程，那只能是处理一个用户的请求。

### 2、伸缩性

通过增加CPU核数来提升性能。

## 使用场景

1、常见的浏览器、Web服务(现在写的web是中间件帮你完成了线程的控制)，web处理请求。

2、FTP下载，多线程操作文件

3、数据库用到多线程。

4、分布式计算

5、后台任务：如定时向大量(100W以上)的用户发送邮件。定期更新配置文件、任务调度(如quartz)，一些监控用于定期信息采集。

6、自动作业处理：比如定期备份日志、定期备份数据库。

7、异步处理：如发微博、记录日志。

8、多步骤的任务处理，可根据步骤特征选用不同个数和特征的线程来协作处理，多任务的分割，由一个主线程分割给多个线程完成。

# 并发编程的三要素

## 1、原子性

原子性指的是一个或者多个操作，要么全部执行并且在执行的过程中不被其他操作打断，要么就全部都不执行。

## 2、可见性

可见性指多个线程操作一个共享变量时，其中一个线程对变量进行修改后其他线程可以立即看到修改的结果。

## 3、有序性

有序性即程序的执行顺序按照代码的先后顺序来执行。

# 实现可见性的方法

synchronized 或者 Lock: 保证同一个时刻只有一个线程获取锁，执行代码，锁释放之前把最新的值刷新到主内存中，实现可见性。

# 线程的状态

## 1、新建状态 (New)

当线程对象创建后，即进入了新建状态，如下：

Thread t = new MyThread();

## 2、就绪状态 (Runnable)

当调用线程对象的 start()方法，线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待 CPU 调度执行，并不是说执行了start()方法后此线程立即就会执行。

## 3、运行状态 (Running)

当 CPU 开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注: 就绪状态是进入到运行状态的唯一入口，也就是说线程要想进入运行状态执行，首先必须处于就绪状态中。

## 4、阻塞状态 (Blocked)

处于运行状态中的线程由于某种原因，暂时放弃了对CPU 的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被 CPU 调用以进入到运行状态。

根据阻塞产生的原因不同，阻塞状态又可以分为三种

### 1、等待阻塞

运行状态中的线程执行了wait方法，使本线程进入到等待阻塞状态。

### 2、同步阻塞

线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)它会进入同步阻塞状态。

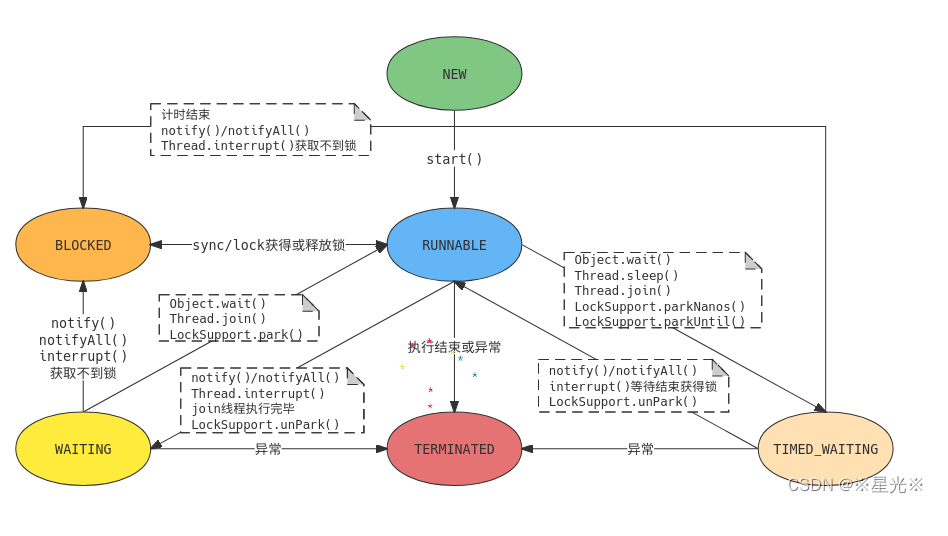
### 3、其他阻塞

通过调用线程的 sleep()或join()或发出了 I/O请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/0 处理完毕时，线程重新转入就绪状态。

## 5、死亡状态 (Dead)

线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

## **线程的生命周期以及状态转换图**



# 创建线程的方法

1、继承Thread类

2、实现Runnable接口

3、通过Callable和Future接口创建线程

4、通过线程池创建线程

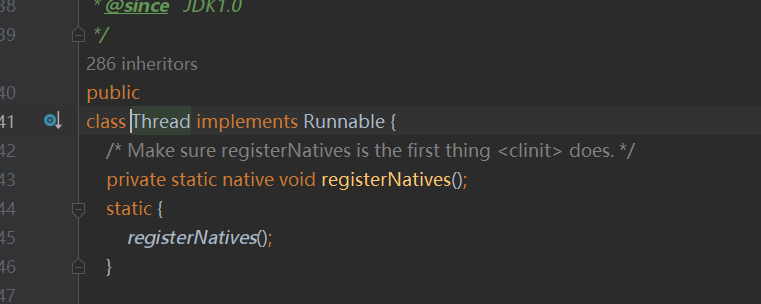
通过继承Thread类和实现Runnanle接口这两种方式都有一个缺陷就是在执行完任务之后无法获取到执行的结果。从Java 1.5开始，就提供了Callable和Future接口，通过它们可以在任务执行完毕之后得到任务的执行结果。

# 类

## Thread类

### 基本概念

### 源码



### 构造函数

#### public Thread()

#### public Thread(Runnable target)

#### public Thread(Runnable target, String name)

### 方法

#### public synchronized void start()

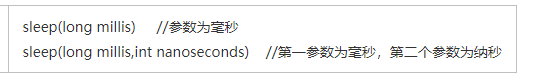
用来启动一个线程，当调用start方法后，系统才会开启一个新的线程来执行用户定义的子任务，在这个过程中会为相应的线程分配需要的资源。

#### public void run()

run()方法是不需要用户来调用的，当通过start方法启动一个线程之后，当线程获得了CPU的执行时间，便进入run方法体去执行具体的任务。注意继承Thread类必须重写run方法，在run方法中定义具体要执行的任务。

#### public static native void sleep(long millis)

sleep方法有两个重载版本：



sleep相当于让线程睡眠，交出CPU，让CPU去执行其他的任务，如果需要让当前正在执行的线程暂停一段时间并进入阻塞状态，则可以通过调用sleep()方法来实现。当前线程调用sleep()方法进入阻塞状态后，在其睡眠时间内，该线程不会获得执行机会，即使系统中没有其他可执行的线程，处于sleep()中的线程也不会执行，因此sleep()方法常用来暂停程序的执行，但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。

#### public static native void yield()

yield()方法和sleep()方法有点相似，它也是Thread类提供的一个静态方法，它也可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入到就绪状态。即让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是当某个线程调用了yield()方法暂停之后，线程调度器又将其调度出来重新执行。调用yield方法会让当前线程交出CPU的权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，当某个线程调用了yield()方法之后，只有优先级与当前线程相同或者比当前线程更高的处于就绪状态的线程才会获得执行机会。

注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU的执行时间，这一点是和sleep方法不一样的。

#### public final void join()

用于等待一个线程完成其执行。假如在main线程中，调用thread.join方法，则main方法会等待thread线程执行完毕或者等待一定的时间。如果调用的是无参join方法，则等待thread执行完毕，如果调用的是指定了时间参数的join方法，则等待一定的时间。

##### 作用

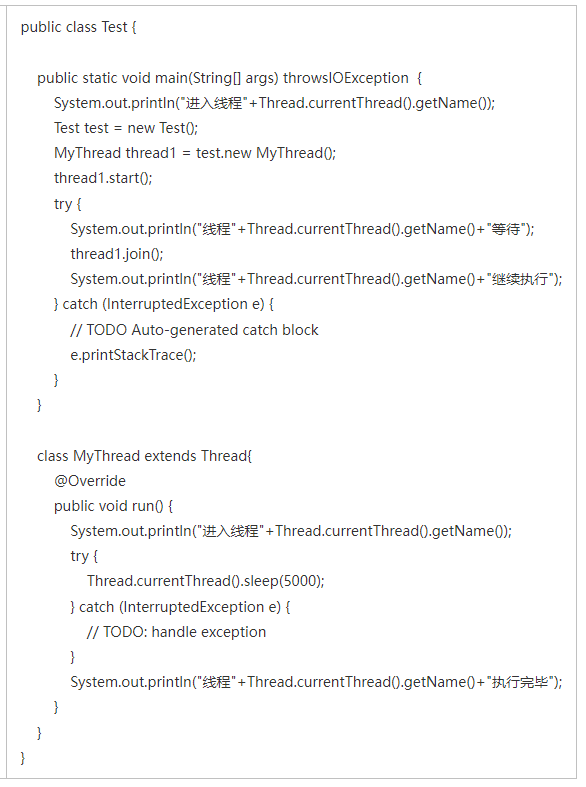
join() 方法用于使当前线程等待调用该方法的线程完成执行。当一个线程调用另一个线程的 join() 方法时，它会暂停自己的执行，直到被调用的线程执行完毕。

##### 使用方式

join() 方法通常通过调用一个线程对象的实例方法来使用，例如 thread.join()。在调用 join() 方法前，需要确保线程对象已经被创建并启动。

##### 示例代码

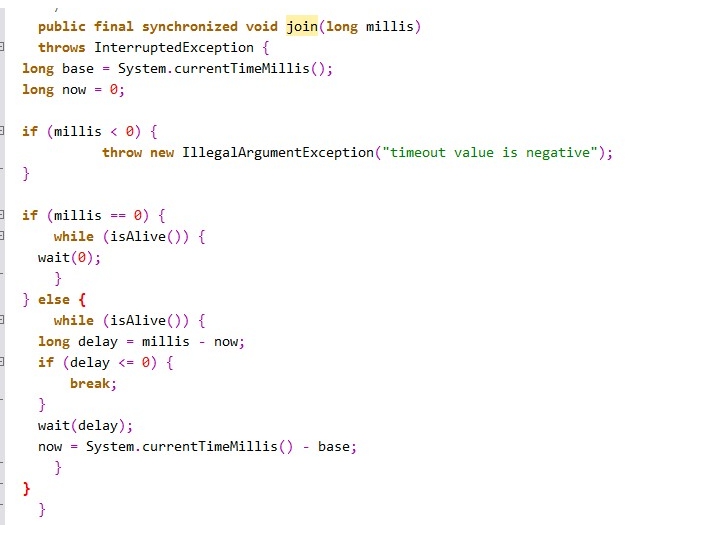
看下面一个例子：



输出结果：



可以看出当调用thread1.join()方法后，main线程会进入等待，然后等待thread1执行完之后再继续执行，实际上调用join方法是调用了Object的wait方法，这个可以通过查看源码得知：



wait方法会让线程进入阻塞状态，并且会释放线程占有的锁，并交出CPU的执行权限。由于wait方法会让线程释放对象锁，所以join方法同样会让线程释放对一个对象持有的锁。

##### 总结

join() 方法用于实现线程之间的协调和同步。它允许一个线程等待另一个线程执行完毕，从而确保多个线程按照预期顺序执行。通过适当地使用 join() 方法，可以实现线程之间的有序执行和任务依赖关系。

#### public final native boolean isAlive();

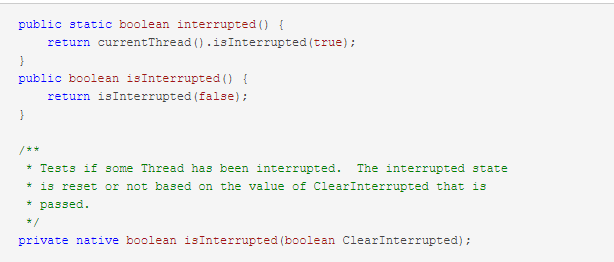
用来判断当前线程是否存活，如果当前线程存活，返回true,如果消亡，返回false。

#### public void interrupt()

单独调用interrupt方法可以使得处于阻塞状态的线程抛出一个异常，也就说它可以用来中断一个正处于阻塞状态的线程，另外通过interrupt方法和isInterrupted()方法来停止正在运行的线程。

#### public static boolean interrupted()

用来检测当前线程的interrupt状态，检测完成后，状态清空。通过下面的interrupted源码我们能够知道，此方法首先调用isInterrupted方法，而isInterrupted方法是一个重载的native方法private native boolean isInterrupted(boolean ClearInterrupted)， 通过方法的注释能够知道，用来测试线程是否已经中断，参数用来决定是否重置中断标志。



#### public boolean isInterrupted()

#### private native boolean isInterrupted(boolean ClearInterrupted)

#### public final void wait()

wait方法是Object对象的方法。线程与锁是分不开的，线程的同步、等待、唤醒都与对象锁是密不可分的。wait方法会将当前线程放入wait set，等待被唤醒，并放弃lock对象上的所有同步声明，当前线程会因为线程调度的原因处于休眠状态而不可用。只有通过以下四个方法可以主动唤醒：

1、notify

2、notifyAll

3、Thread.interrupt()

4、等待时间过完。

#### public final native void wait(long timeout)

#### public final void wait(long timeout, int nanos)

#### public final void stop()

这是废弃的方法

#### public void destroy()

这是废弃的方法

#### public final int getPriority()

获取当前线程的优先级

#### public final void setPriority(int newPriority)

设置当前线程的优先级

### 线程的优先级

#### ****3个优先级静态常量****

MAX\_PRIORITY:10（最高优先级）  
MIN\_PRIORITY:1（最低优先级）  
NORM\_PRIORITY:5（默认优先级）

### sleep方法和wait方法有什么区别

1、sleep 方法和 wait 方法都可以用来放弃 CPU 一定的时间，不同点在于如果线程持有某个对象的监视器，sleep 方法不会放弃这个对象的监视器，wait 方法会放弃这个对象的监视器。

2、wait()和sleep()都可以通过interrupt()方法打断线程的暂停状态，从而使线程立刻抛出InterruptedException（但不建议使用该方法）。

## Executors类

### 基本概念

## SimpleAsyncTaskExecutor类

## ThreadPoolTaskExecutor类

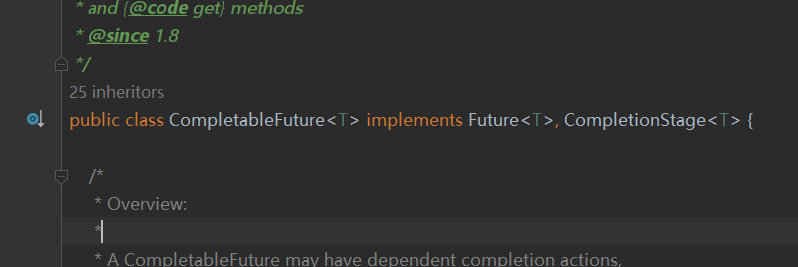
### 基本概念

## CompletableFuture类

### 基本概念

Java 8 引入的一个用于处理异步操作的工具类。它提供了一种更灵活、更强大的方式来处理异步任务，并可以方便地处理任务之间的依赖关系和组合操作。

### 源码



### 作用

#### 1、异步执行任务

CompletableFuture 允许你将一个耗时的任务提交给线程池进行异步执行，而不会阻塞当前线程。你可以使用 supplyAsync() 方法执行一个有返回值的任务，或者使用 runAsync() 方法执行一个没有返回值的任务。

#### 2、处理任务结果

通过 CompletableFuture 提供的一些方法，你可以对异步任务的结果进行处理。例如你可以使用 thenApply() 方法对任务的结果进行转换，或者使用 thenAccept() 方法消费任务的结果，还可以使用 thenRun() 方法在任务完成后执行一些操作。

#### 3、组合多个任务

CompletableFuture 提供了一系列的组合方法，如 thenCompose()、thenCombine() 和 allOf() 等，用于将多个任务进行组合。这些方法允许你按照一定的顺序或条件组合任务，实现任务之间的依赖关系。

#### 4、处理异常

使用 exceptionally() 方法，你可以指定一个回调函数来处理任务执行过程中发生的异常，从而有效地处理错误情况。

#### 5、等待多个任务完成

通过 CompletableFuture.allOf() 或 CompletableFuture.anyOf() 方法，你可以等待多个任务全部完成或任一任务完成，这在需要等待多个异步任务的结果时非常有用。

通过使用 CompletableFuture，你可以以更优雅和便捷的方式处理复杂的异步编程场景，避免了传统的回调地狱。它提供了更多的控制和灵活性，使得异步任务的处理代码更易于编写和维护。

### 方法

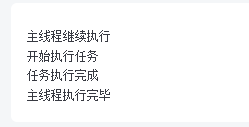
#### public static CompletableFuture<Void> runAsync(Runnable runnable)

允许你在后台线程或线程池中异步执行一个没有返回值的任务，而不必等待任务完成。



在上面的示例中，我们创建了一个 CompletableFuture 对象 future，并使用 runAsync() 方法执行一个任务。该任务是一个匿名函数，内部包含了我们希望在后台执行的代码。在任务执行过程中，我们使用 Thread.sleep() 方法模拟了一个耗时操作。然后在任务执行完成后，我们打印了一条消息。在主线程中，我们紧接着执行了一条提示消息，并使用 future.join() 方法等待任务执行完成，这样可以保证在任务执行完成之前主线程不会退出。最后，我们打印了一条消息表示主线程执行完毕。

当你运行这个示例时，你会看到以下输出：



可以看到在调用 runAsync() 方法后，主线程不需要等待任务执行完成，而是继续执行其他操作，而任务会在后台线程或线程池中异步执行。

#### public static CompletableFuture<Void> runAsync(Runnable runnable,Executor executor)

#### public T get()

#### public T join()

#### public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

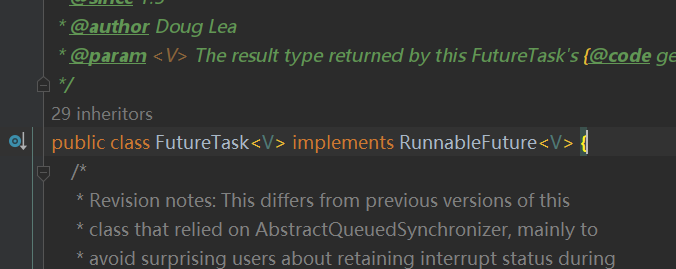
## ScheduledThreadPoolExecutor

## FutureTask类

### 基本概念

FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。由于FutureTask 也是Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

### 源码



### 构造函数

#### public FutureTask(Callable<V> callable)

#### public FutureTask(Runnable runnable, V result)

### 方法

#### public V get()

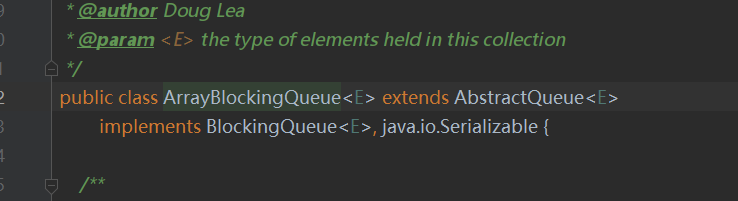
## ArrayBlockingQueue类

### 基本概念

ArrayBlockingQueue，一个由**数组**实现的**有界**阻塞队列。该队列采用先进先出（FIFO）的原则对元素进行排序添加的（有序可重复的）。ArrayBlockingQueue 为**有界且固定**，其大小在构造时由构造函数来决定，确认之后就不能再改变了。ArrayBlockingQueue 支持对等待的生产者线程和使用者线程进行排序的可选公平策略，但是在默认情况下不保证线程公平的访问，在构造时可以选择公平策略（fair = true）。公平性通常会降低吞吐量，但是减少了可变性和避免了“不平衡性”。（ArrayBlockingQueue 内部的阻塞队列是通过 ReentrantLock 和 Condition 条件队列实现的， 所以 ArrayBlockingQueue 中的元素存在公平和非公平访问的区别）

所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素，可以保证先进先出，避免饥饿现象。

### 源码



### 构造函数

#### public ArrayBlockingQueue(int capacity)

#### public ArrayBlockingQueue(int capacity, boolean fair)

#### public ArrayBlockingQueue(int capacity, boolean fair,Collection<? extends E> c)

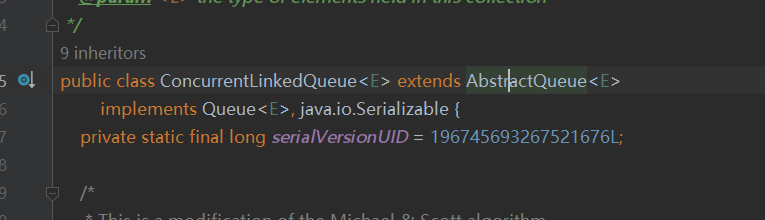
### 使用场景

## ConcurrentLinkedQueue类

### 基本概念

ConcurrentLinkedQueue是Java中的一个并发线程安全的队列实现，它实现了Queue接口，可以用于在多线程环境下实现高效的元素插入和删除操作。ConcurrentLinkedQueue内部使用无锁的链表数据结构来存储元素。它采用一种基于"wait-free"算法的无锁算法，通过CAS（Compare And Swap）等原子操作实现并发访问的线程安全性。这意味着多个线程可以并发地对队列进行操作，而不需要加锁等待。

### 源码



### 特点

1、无界队列：ConcurrentLinkedQueue没有指定容量的限制，可以根据需要动态增长。

2、高并发性能：由于使用无锁算法，多个线程可以并发地进行插入和删除操作，提供较好的吞吐量。

3、先进先出顺序：队列采用先进先出（FIFO）的顺序，保证了元素的顺序性。

4、支持弱一致性：ConcurrentLinkedQueue提供了弱一致性的迭代器，即在迭代过程中可能会遗漏一些新增的元素。

### 方法

#### add(E e)

往队列尾部插入一个元素，如果元素插入成功则返回true，否则抛出异常或者返回false。

#### offer(E e)

往队列尾部插入一个元素，如果元素插入成功则返回true，否则抛出异常或者返回false。

#### add和offer方法的区别

add方法：

（1）如果队列未满，直接将元素添加到队列的尾部，并返回true。

（2）如果队列已满，抛出IllegalStateException异常。

offer方法：

（1）如果队列未满，将元素添加到队列的尾部，并返回true。

（2）如果队列已满，不会抛出异常，而是返回false。

总结起来，add(E e)方法在队列已满时会抛出异常，而offer(E e)方法在队列已满时只会返回false。在实际使用中，建议优先使用offer(E e)方法，因为它避免了抛出异常的开销，并且能够更好地适应队列已满的情况。如果使用add(E e)方法，需要在使用前通过容量判断或者捕获异常来处理队列已满的情况。

#### poll()

删除头部元素并返回，如果队列为空则返回null。

#### pollFirst()

删除头部元素并返回，如果队列为空则返回null。

#### peek()

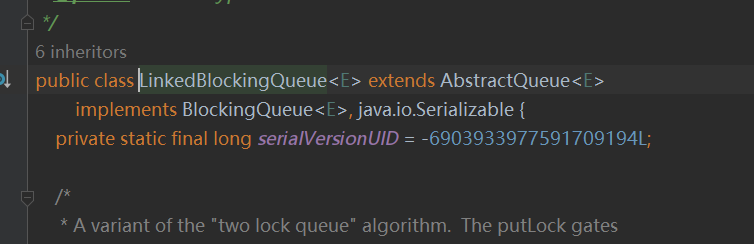
返回头部元素但不删除，如果队列为空则返回null。

#### peekFirst()

## LinkedBlockingQueue类

### 基本概念

### 源码

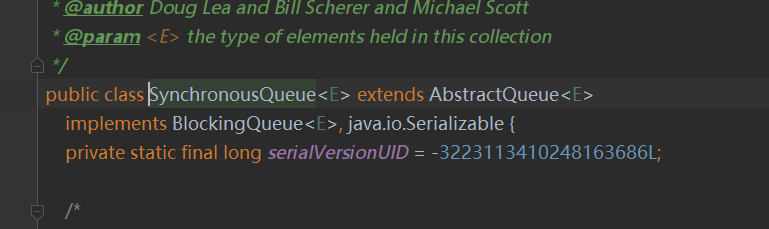


### 方法

## SynchronousQueue类

### 基本概念

### 源码



### 构造函数

#### public SynchronousQueue()

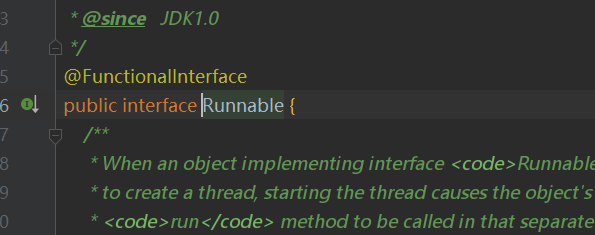
#### public SynchronousQueue(boolean fair)

# 接口

## Runnable接口

### 基本概念

### 源码



### 方法

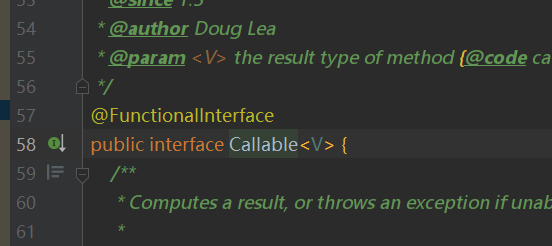
#### public abstract void run()

只有这一个方法

## Callable接口

### 基本概念

### 源码



### 方法

#### V call()

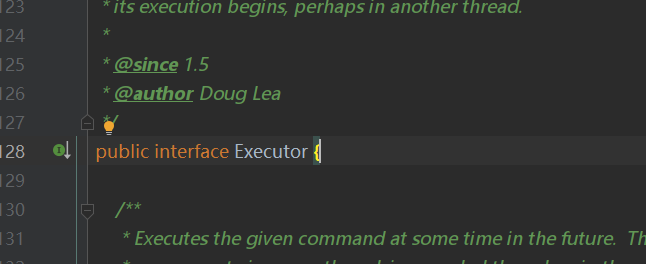
计算结果，如果无法计算结果，则引发异常。

## Executor接口

### 基本概念

Executor又称为执行器，用于执行提交的任务的简单接口，它提供了一种将任务提交与任务执行分离的机制。

### 源码



### 方法

#### void execute(Runnable command)

在将来的某个时间执行给定的命令。该命令可以在新线程、池线程或调用线程中执行，由 Executor 实现自行决定。

### 作用

将任务的提交和执行进行解耦。它允许开发人员专注于任务本身的实现，而不需要关注任务的执行细节。通过将任务的提交与执行分离，Executor 接口提供了一种灵活的机制来管理、调度和执行任务。

### 代码示例



在上述示例中，我们首先使用 Executors.newFixedThreadPool(3) 创建一个固定大小为 3 的线程池。然后，通过循环提交了 5 个任务，每个任务都打印任务编号和执行线程的名称。最后，我们通过判断 executor 是否是 ExecutorService 的实例来关闭线程池。

### 总结

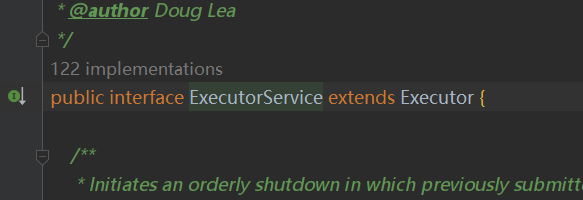
Executor 接口提供了一种简单而灵活的机制来管理和调度任务的执行。通过创建 Executor 实例，并调用其 execute() 方法来提交任务，可以实现对任务的异步执行和线程池的管理。

## ExecutorService接口

### 基本概念

继承了 Executor 接口，包含了大量的方法控制任务的进度和管理服务的终止。使用此接口，您可以提交要执行的任务，还可以使用返回的Future实例控制其执行。

### 源码



### 方法

#### <T> Future<T> submit(Callable<T> task)

提交执行一个有返回值的任务，并返回表示任务挂起结果的 Future 。Future 的 get 方法将在成功完成后返回任务的结果。

参数：

task：提交的任务

T：任务结果的类型

#### Future<?> submit(Runnable task)

#### <T> Future<T> submit(Runnable task, T result);

#### void shutdown()

## TaskExecutor接口

### 基本概念

TaskExecutor 是 Spring Framework 中定义的一个接口，它是用于执行异步任务的策略接口。该接口定义了一个方法 execute(Runnable task)，用于提交任务对象并异步执行。

### 作用

#### 异步任务

在应用程序中，有时候我们需要执行一些耗时的操作，例如调用远程服务、访问数据库或执行复杂的计算。这些操作可能会阻塞当前线程，导致应用程序响应变慢。为了避免这种情况，可以将这些操作封装到异步任务中，使其在独立的线程中执行。

## ScheduledExecutorService接口

## AsyncConfigurer接口

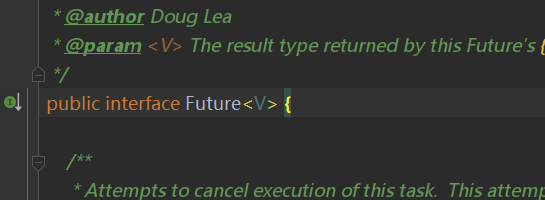
### 基本概念

## Future接口

### 基本概念

Future 表示一个可能还没有完成的异步任务的结果，针对这个结果可以添加Callback 以便在任务执行成功或失败后作出相应的操作。

### 源码



### 方法

#### boolean cancel(boolean mayInterruptIfRunning)

取消任务的执行。参数指定是否立即中断任务执行，或者等等任务结束。

#### boolean isCancelled()

任务是否已经取消，任务正常完成前将其取消，则返回 true

#### boolean isDone()

任务是否已经完成。需要注意的是如果任务正常终止、异常或取消，都将返回true

#### V get()

等待任务执行结束，然后获得V类型的结果。

#### V get (long timeout, TimeUnit unit)

同上面的get功能一样，多了设置超时时间。参数timeout指定超时时间，uint指定时间的单位，在枚举类TimeUnit中有相关的定义。

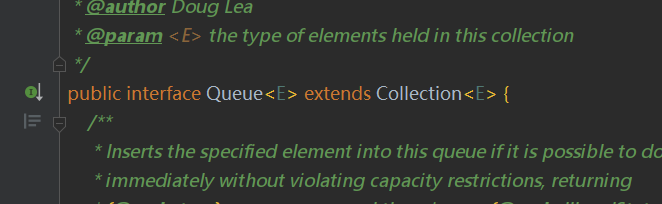
## Queue接口

### 基本概念

队列是一种特殊的线性表，遵循先入先出、后入后出的基本原则，一般来说，它只允许在表的前端进行删除操作，而在表的后端进行插入操作，但是java的某些队列允许在任何地方插入删除，比如我们常用的 LinkedList 集合，它实现了Queue 接口，因此我们可以理解为 LinkedList 就是一个队列。



### 源码



### 实现类

LinkedList

ArrayDeque

PriorityQueue

### 方法

#### boolean add(E e);

将指定的元素插入队列。如果任务成功，则返回true，否则将引发异常。

#### boolean offer(E e);

将指定的元素插入队列。如果任务成功，则返回true，否则返回false。

#### E remove();

返回并删除队列的头部。如果队列为空，则引发异常。

#### E poll();

返回并删除队列的开头。 如果队列为空，则返回null。

#### E element();

返回队列的开头。如果队列为空，则引发异常。

#### E peek()

返回队列的开头。 如果队列为空，则返回null。

### 分类

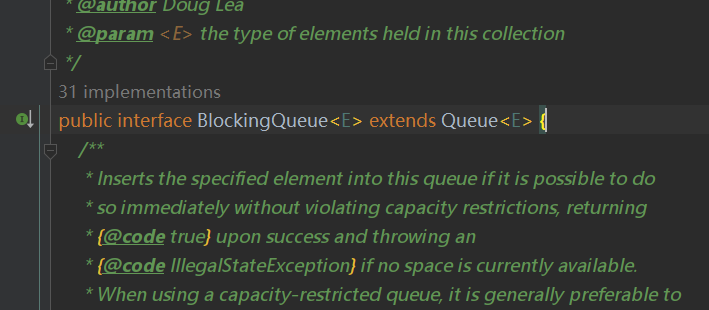
队列主要分为阻塞和非阻塞，有界和无界、单向链表和[双向链表](https://so.csdn.net/so/search?q=%E5%8F%8C%E5%90%91%E9%93%BE%E8%A1%A8&spm=1001.2101.3001.7020)之分。

## BlockingQueue接口

### 基本概念

BlcokingQueue继承了Queue接口，是队列的一种，Queue和BlockingQueue都是在Java5中加入的，BlockingQueue是线程安全的，我们在很多场景下都可以利用线程安全的队列来优雅地解决我们业务自身的线程安全问题。

### 源码



### 方法

#### boolean add(E e);

#### public boolean contains(Object o)

#### void put(E e)

将指定的元素插入到队列中，必要时等待使空间变得可用。

#### boolean remove(Object o)

从队列中删除指定元素的单个实例

#### E take()

检索并删除队列的头部，必要时等待直到元素可用。

# 阻塞队列

## 基本概念

阻塞队列 (BlockingQueue)是Java util.concurrent包下重要的数据结构，提供了线程安全的队列访问方式，当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。并发包下很多高级同步类的实现都是基于BlockingQueue实现的。

## 特点

阻塞队列的特点就是阻塞两个字，阻塞功能使得生产者和消费者两端的能力得以平衡，当有任何一端速度过快时，阻塞队列便会把过快的速度降下来。最重要的两个方法：take()和put()。

## 操作方法

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同，这些方法如下：



四组不同的行为方式解释：

（1）抛异常：如果试图的操作无法立即执行，抛一个异常。

（2）特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

（3）阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

（4）超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是true / false)。

无法向一个 BlockingQueue中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException。

# 线程池

## 基本概念

线程池（ThreadPool）是一种基于池化思想管理和使用线程的机制。它是将多个线程预先存储在一个“池子”内，当有任务出现时可以避免重新创建和销毁线程所带来的性能开销，只需要从“池子”内取出相应的线程执行对应的任务即可。

## 优势

### 1、降低资源消耗

通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。

### 2、提高响应速度

任务到达时，无需等待线程创建即可立即执行。

### 3、提高线程的可管理性

线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。

### 4、提供更多更强大的功能

线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

同时阿里巴巴在其《Java开发手册》中也强制规定：线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。

## 创建线程池的方法

### 基本概念

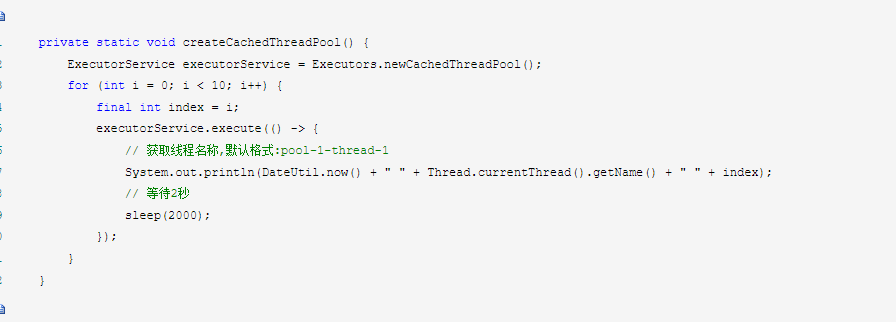
JAVA中创建线程池主要有两类方法，一类是通过Executors工厂类提供的方法，该类提供了4种不同的线程池可供使用。另一类是通过ThreadPoolExecutor类进行自定义创建。

### ****通过Executors类****

#### 1、newCachedThreadPool()

创建一个可缓存的线程池，若线程数超过处理所需，缓存一段时间后会回收，若线程数不够，则新建线程。

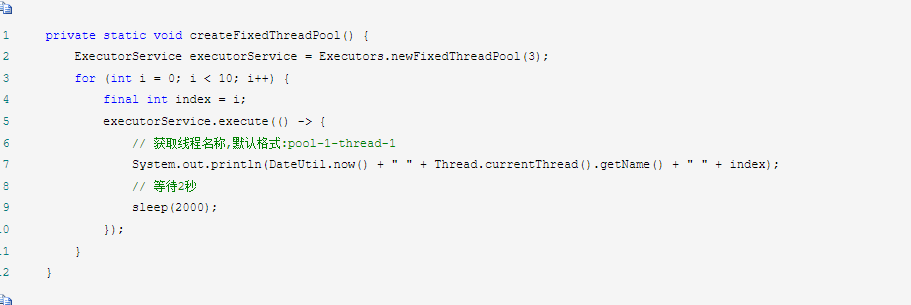
代码示例



效果：因为初始线程池没有线程，而线程不足会不断新建线程，所以线程名都是不一样的。

#### 2、newFixedThreadPool()

创建一个固定大小的线程池，可控制并发的线程数，超出的线程会在队列中等待。



效果：因为线程池大小是固定的，这里设置的是3个线程，所以线程名只有3个。因为线程不足会进入队列等待线程空闲，所以日志间隔2秒输出。

#### 3、newScheduledThreadPool()

创建一个周期性的线程池，支持定时及周期性执行任务。

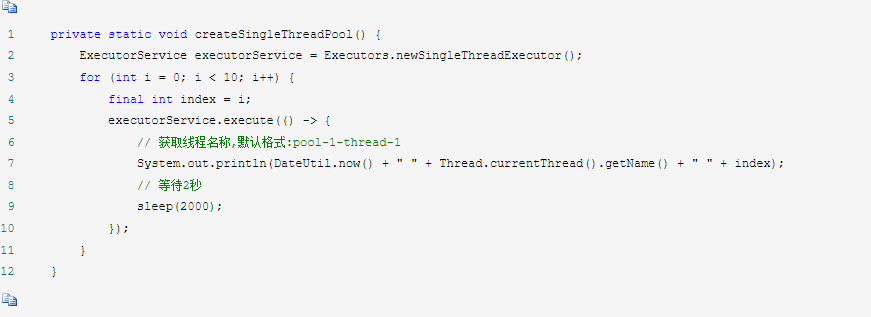


效果：因为设置了延迟3秒，所以提交后3秒才开始执行任务。因为这里设置核心线程数为3个，而线程不足会进入队列等待线程空闲，所以日志间隔2秒输出。

注意：这里用的是ScheduledExecutorService类的schedule()方法，不是ExecutorService类的execute()方法。

#### 4、newSingleThreadExecutor

创建一个单线程的线程池，可保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。



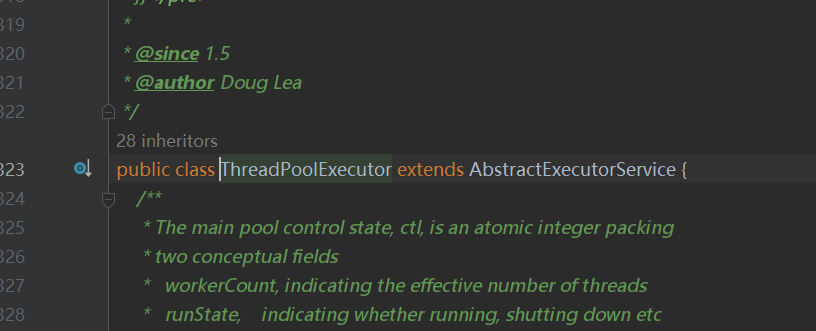
效果：因为只有一个线程，所以线程名均相同，且是每隔2秒按顺序输出的。

### ****通过ThreadPoolExecutor类自定义****

#### 基本概念

ThreadPoolExecutor类提供了4种构造方法，可根据需要来自定义一个线程池。

#### 源码



#### 构造函数

##### public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

#### 方法

#### 线程池的参数（7个）

##### 1、corePoolSize

核心线程数。

##### 2、maximumPoolSize

最大线程数。

##### 3、keepAliveTime

空闲线程的存活时间。

##### 4、TimeUnit

时间单位。

##### 5、BlockingQueue

一个阻塞队列，用来存储等待执行的任务，均为线程安全，7种可选。



较常用的是LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

##### 6、ThreadFactory

创建线程的工厂。

##### 7、RejectedExecutionHandler

拒绝策略。

最大线程数什么时候开始启用？当队列满的时候开始启用

#### 线程池的拒绝策略

##### 1、****AbortPolicy（默认）****

丢弃任务并抛出 RejectedExecutionException 异常。

##### **2、**CallerRunsPolicy****

由调用线程处理该任务

##### **3、**DiscardPolicy****

丢弃任务，但是不抛出异常。可以配合这种模式进行自定义的处理方式。

##### **4、**DiscardOldestPolicy****

丢弃队列最早的未处理任务，然后重新尝试执行任务。

#### 线程池的执行规则

1、当线程数小于核心线程数时，创建线程。

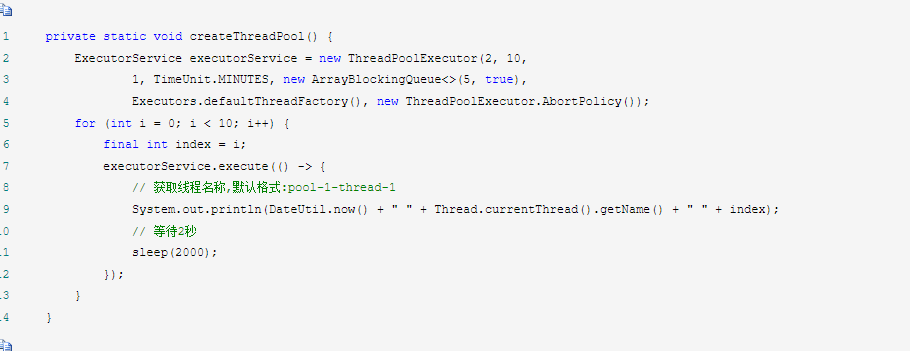
2、当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。

3、当线程数大于等于核心线程数，且任务队列已满：

若线程数小于最大线程数，创建线程。

若线程数等于最大线程数，抛出异常，拒绝任务。

#### 代码示例

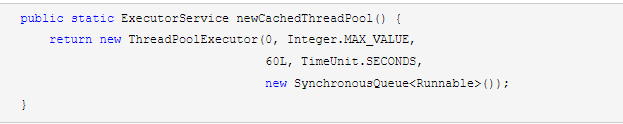


效果：因为核心线程数为2，队列大小为5，存活时间1分钟，所以流程是第0-1号任务来时，陆续创建2个线程，然后第2-6号任务来时，因为无线程可用，均进入了队列等待，第7-9号任务来时，没有空闲线程，队列也满了，所以陆续又创建了3个线程。所以你会发现7-9号任务反而是先执行的。又因为各任务只需要2秒，而线程存活时间有1分钟，所以线程进行了复用，所以总共只创建了5个线程。

### 这两种方式的比较

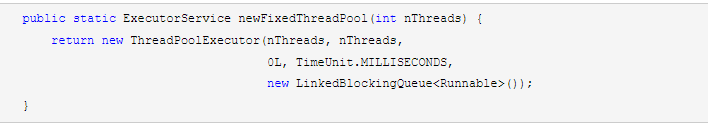
因为Executors类提供的4种方式，其底层其实都是通过ThreadPoolExecutor类来实现的。换句话说就是Executors类工厂通过参数的组合，组装出了上面提到的4种类型线程池供不同场景使用。我们可以通过查看Executors类的源码来看：

1、newCachedThreadPool



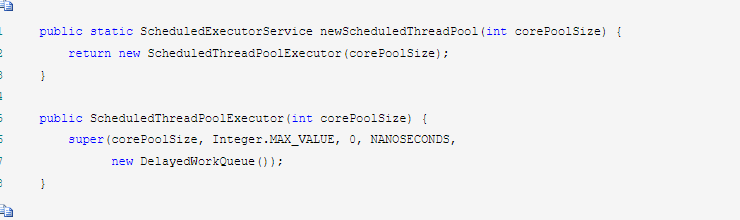
因为SynchronousQueue队列不保持它们，直接提交给线程，相当于队列大小为0，而最大线程数为Integer.MAX\_VALUE，所以线程不足时，会一直创建新线程，等到线程空闲时，又有60秒存活时间，从而实现了一个可缓存的线程池。

2、newFixedThreadPool



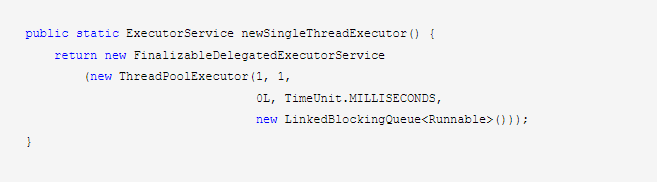
因为核心线程数与最大线程数相同，所以线程池的线程数是固定的，而且没有限制队列的大小，所以多余的任务均会被放到队列排队，从而实现一个固定大小，可控制并发数量的线程池。

3、newScheduledThreadPool



因为使用了延迟队列，只有在延迟期满时才能从中提取到元素，从而实现定时执行的线程池。而周期性执行是配合上层封装的其他类来实现的，可以看ScheduledExecutorService类的scheduleAtFixedRate方法。

4、newSingleThreadExecutor



因为核心线程数与最大线程数相同，均为1，所以线程池的线程数是固定的1个，而且没有限制队列的大小，所以多余的任务均会被放到队列排队，从而实现一个单线程按指定顺序执行的线程池。

虽然看上去Executors类的封装，可以简化我们的使用，但事实上阿里代码规范《阿里巴巴Java开发手册》中明确不建议使用Executors类提供的这4种方法：

【强制】线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

### Executors返回的线程池对象的弊端

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

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

所以我们应该使用ThreadPoolExecutor类来创建线程池，根据自己需要的场景来创建一个合适的线程池。

# 常用的并发工具类

1、CountDownLatch

2、CyclicBarrier

3、Semaphore

4、Exchanger

# CountDownLatch类

# CyclicBarrier类

# CyclicBarrier和CountDownLatch的区别

1、 CountDownLatch 简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用 countDown方法发出通知后，当前线程才可以继续执行。

2、cyclicBarrier 是所有线程都进行等待，直到所有线程都准备好进入 await()方法之后，所有线程同时开始执行。

3、 CountDownLatch 的计数器只能使用一次。而 CyclicBarrier的计数器可以使用 reset()方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

4、CyclicBarrier 还提供其他有用的方法，比如 getNumberWaiting 方法可以获得 CyclicBarrier 阻塞的线程数量。isBroken 方法用来知道阻塞的线程是否被中断，如果被中断返回 true，否则返回 false。

# synchronized关键字

## 作用

synchronized 关键字是用来控制线程同步的，在多线程的环境下控制 synchronized 代码段不被多个线程同时执行。synchronized 既可以加在一段代码上，也可以加在方法上。

## 注意事项

1、一把锁只能同时被一个线程获取，没有获得锁的线程只能等待。

2、每个实例都对应有自己的一把锁(this)，不同实例之间互不影响，所有对象公用同一把锁。

3、synchronized修饰的方法，无论方法正常执行完毕还是抛出异常，都会释放锁。

## 对象锁

包括方法锁(默认锁对象为this,当前实例对象)和同步代码块锁(自己指定锁对象。

### 代码块形式：手动指定锁定对象，可以是this,也可以是自定义的锁

#### 代码示例

示例一：



输出结果：

文本

描述已自动生成

示例二：

文本

描述已自动生成

输出结果：

文本

描述已自动生成

# volatile关键字

Java 提供了 volatile 关键字来保证可见性。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见iava.util.concurrent.atomic 包下的类，比如 Atomiclnteger。

# synchronized和volatile的区别

1、volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好，但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。

2、多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞。

3、volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

4、volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。

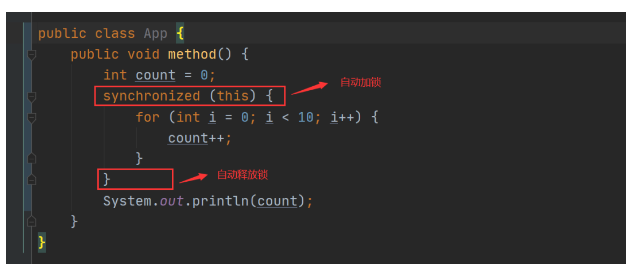
# synchronized与ReentrantLock的区别

## 1、用法不同

synchronized 可用来修饰普通方法、静态方法和代码块，而 ReentrantLock 只能用在代码块上。

## 2、获取锁和释放锁方式不同

synchronized 会自动加锁和释放锁，当进入 synchronized 修饰的代码块之后会自动加锁，当离开 synchronized 的代码段之后会自动释放锁，如下图所示：



而 ReentrantLock 需要手动加锁和释放锁，如下图所示：



PS：在使用 ReentrantLock 时要特别小心，unlock 释放锁的操作一定要放在 finally 中，否者有可能会出现锁一直被占用，从而导致其他线程一直阻塞的问题。

## 3、锁类型不同

synchronized 属于非公平锁，而 ReentrantLock 既可以是公平锁也可以是非公平锁。默认情况下 ReentrantLock 为非公平锁，这点查看源码可知：

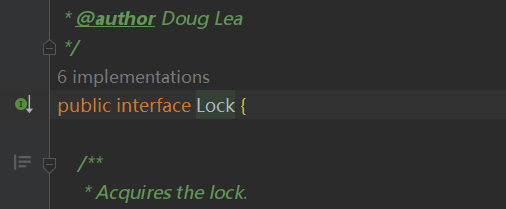
## 4、响应中断不同

## 5、底层实现不同

# Lock接口

## 基本概念

## 源码



## 方法

### void lock();

### void lockInterruptibly()

### boolean tryLock();

### boolean tryLock(long time, TimeUnit unit)

### void unlock();

### Condition newCondition();

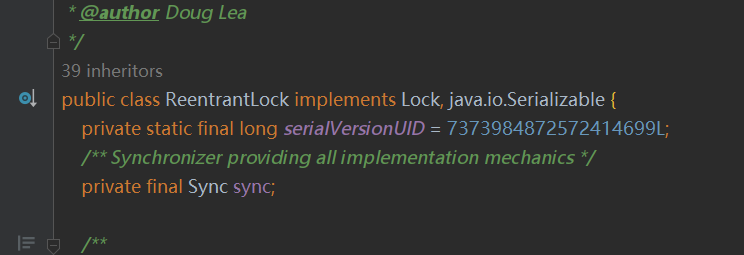
# ReentrantLock类

## 基本概念

ReentrantLock是Java并发包中[互斥锁](https://so.csdn.net/so/search?q=%E4%BA%92%E6%96%A5%E9%94%81&spm=1001.2101.3001.7020)，它有公平锁和非公平锁两种实现方式，以lock()为例，其使用方式为



## 源码



## 构造函数

### public ReentrantLock()

### public ReentrantLock(boolean fair)

## 方法

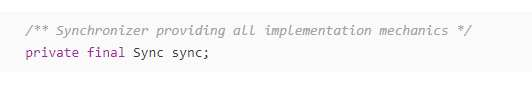
### public int getHoldCount()

### public void lock()

### public void unlock()

## 类的结构

ReentrantLock类实现了Lock和java.io.Serializable接口，其内部有一个实现锁功能的关键成员变量Sync类型的sync，定义如下：



而这个Sync是继承了AbstractQueuedSynchronizer的内部抽象类，主要由它负责实现锁的功能。关于AbstractQueuedSynchronizer我们会在以后详细介绍，你只要知道它内部存在一个获取锁的等待队列及其互斥锁状态下的int状态位（0当前没有线程持有该锁、n存在某线程重入锁n次）即可，该状态位也可用于其它诸如共享锁、信号量等功能。Sync在ReentrantLock中有两种实现类：NonfairSync、FairSync，正好对应了ReentrantLock的非公平锁、公平锁两大类型。

## 适用场景

### 1、同步执行（同synchronized）

### 2、尝试等待执行

### 3、尝试执行（同分布式锁）

### 4、可中断执行

### 5、可重入

（1）可以对同一资源多次加锁&解锁。

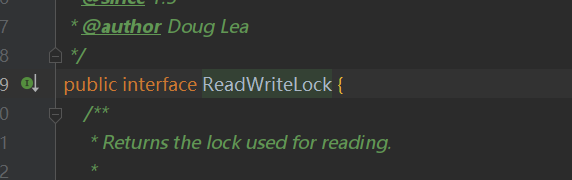
（2）lock(), unlock()配对使用，加一次就要解一次。

# ReadWriteLock接口

## 基本概念

ReadWriteLock 是一个读写锁接口，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

## 源码



## 方法

### Lock readLock();

### Lock writeLock();

## 适用场景

# CAS

## 基本概念

CAS 是 compare and swap 的缩写，即我们所说的比较交换。cas 是一种基于锁的操作，而且是乐观锁。在java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数内存位置 (V) 、预期原值 (A) 和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的( Atomiclnteger,AtomicBoolean,AtomicLong)。

## 问题

### 1、CAS容易造成 ABA 问题

一个线程 a 将数值改成了 b，接着又改成了 a，此时 CAS 认为是没有变化，其实是已经变化过了，而这个问题的解决方案可以使用版本号标识，每操作一次version 加 1。在 java5 中，已经提供了 AtomicStampedReference 来解决问题。

### 2、不能保证代码块的原子性

CAS 机制所保证的是一个变量的原子性操作，而不能保证整个代码块的原子性，比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 synchronized 了。

### 3、CAS 造成CPU利用率增加

之前说过了 CAS 里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu资源会一直被占用。

# synchronized、volatile、CAS 比较

1、synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

2、volatile 提供多线程共享变量可见性和禁止指令重排序优化。

3、CAS 是基于冲突检测的乐观锁 (非阻塞)

# 锁

## 基本概念

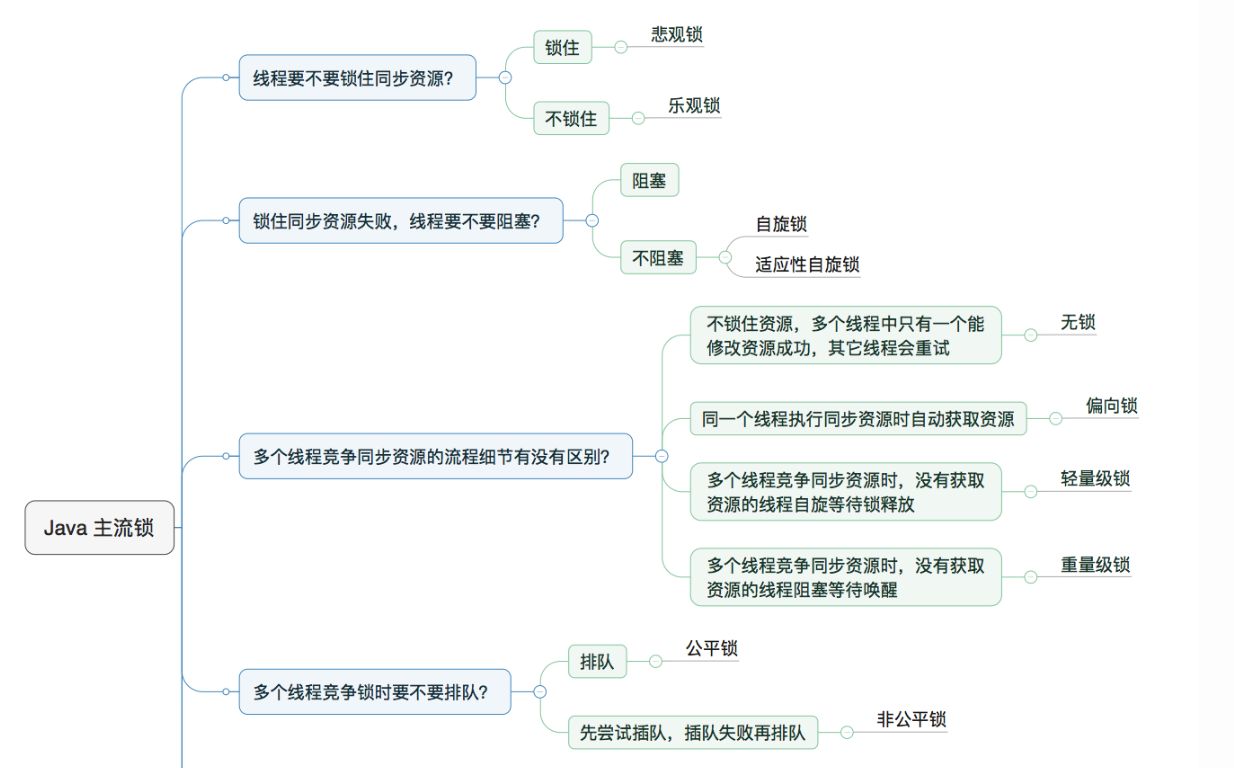
## 分类

公平锁、非公平锁

无锁、偏向锁、轻量级锁、重量级锁

独占锁、共享锁

乐观锁、悲观锁

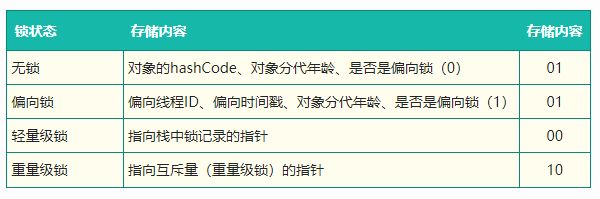


图示

描述已自动生成

## 锁的状态

目前锁一共有4种状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁。锁状态只能升级不能降级。

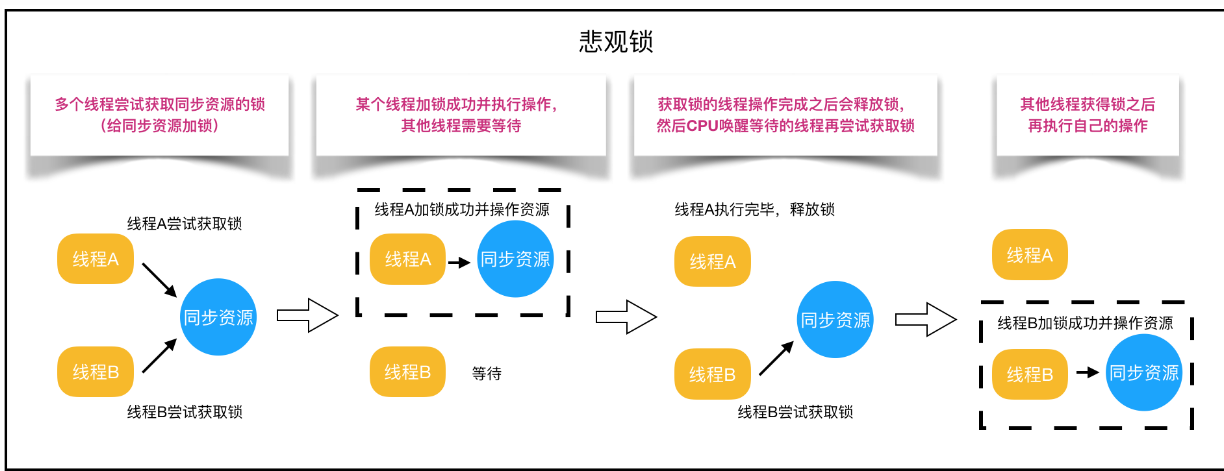


## 可重入锁

## 悲观锁

### 基本概念

对于同一个数据的并发操作，悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java中synchronized关键字和Lock的实现类都是悲观锁。



### 适用场景

悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。

### 代码示例



## 乐观锁

### 基本概念

乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试），乐观锁在Java中是通过使用无锁编程来实现，最常采用的是CAS算法，Java原子类中的递增操作就通过CAS自旋实现的。

图示

描述已自动生成

### 适用场景

乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

### 实现方式

1、版本号机制

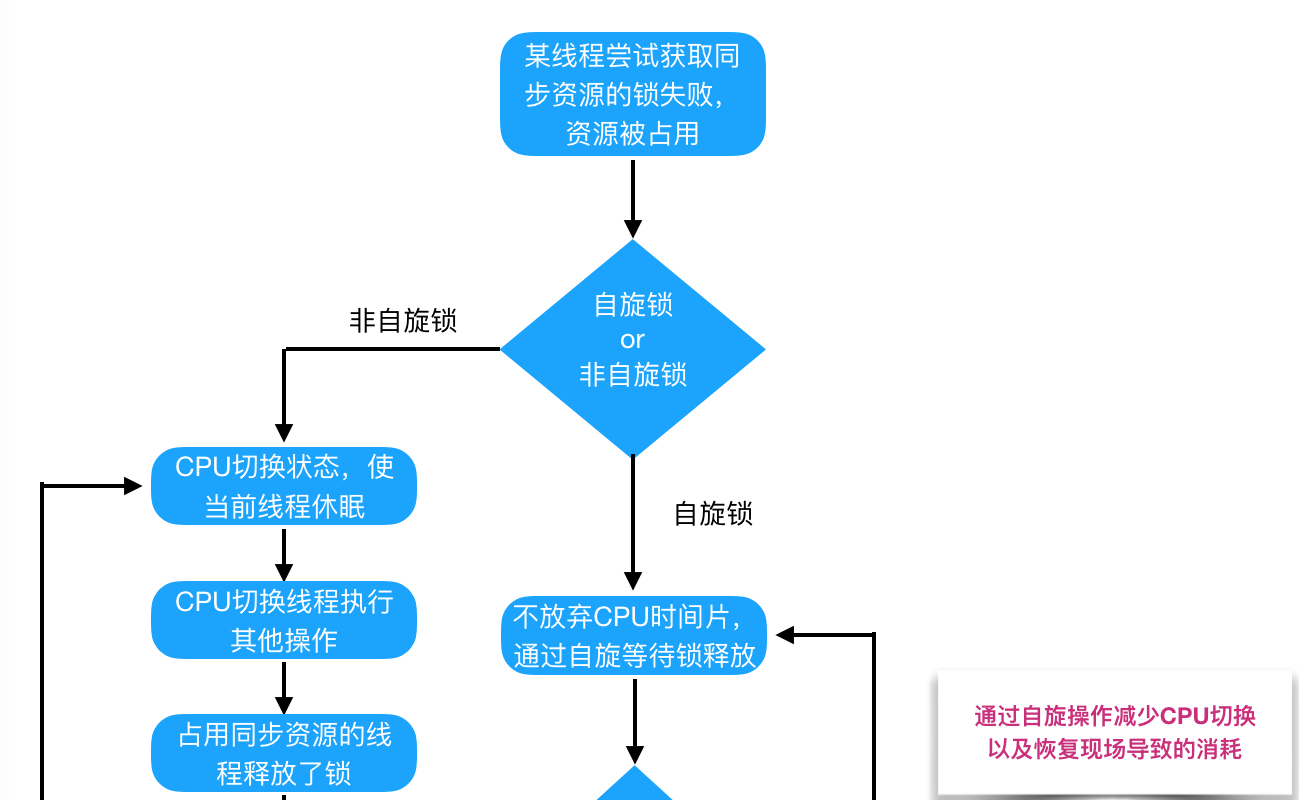
2、CAS算法

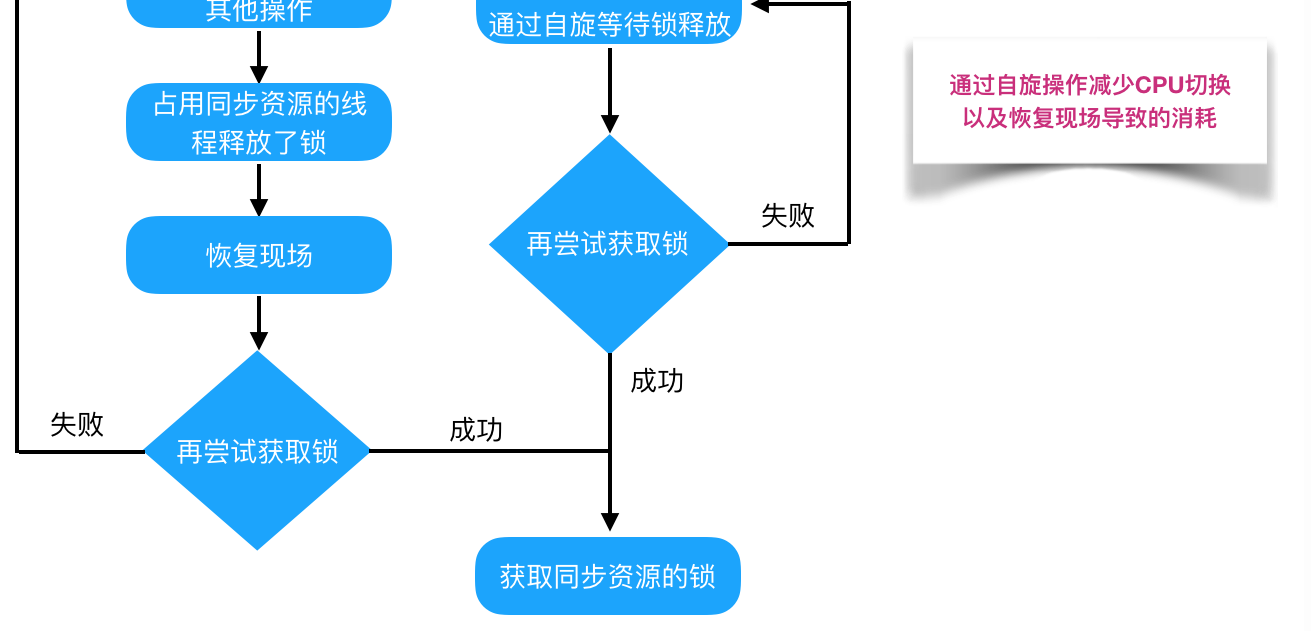
### 代码示例

## 自旋锁 VS 适应性自旋锁

### 基本概念

阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销，这就是自旋锁。

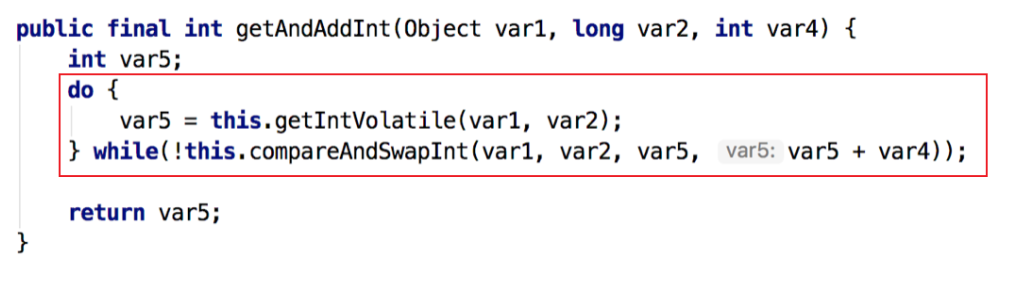




自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程。

### 原理

自旋锁的实现原理同样也是CAS，AtomicInteger中调用unsafe进行自增操作的源码中的do-while循环就是一个自旋操作，如果修改数值失败则通过循环来执行自旋，直至修改成功。



自旋锁在JDK1.4.2中引入，使用-XX:+UseSpinning来开启。JDK 6中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。自适应意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

## 无锁

# 线程B怎么知道线程 A 修改了变量

1、volatile 修饰变量

2、synchronized 修饰修改变量的方法

3、wait/notify

4、while 轮询

# ThreadLocal类

## 基本概念

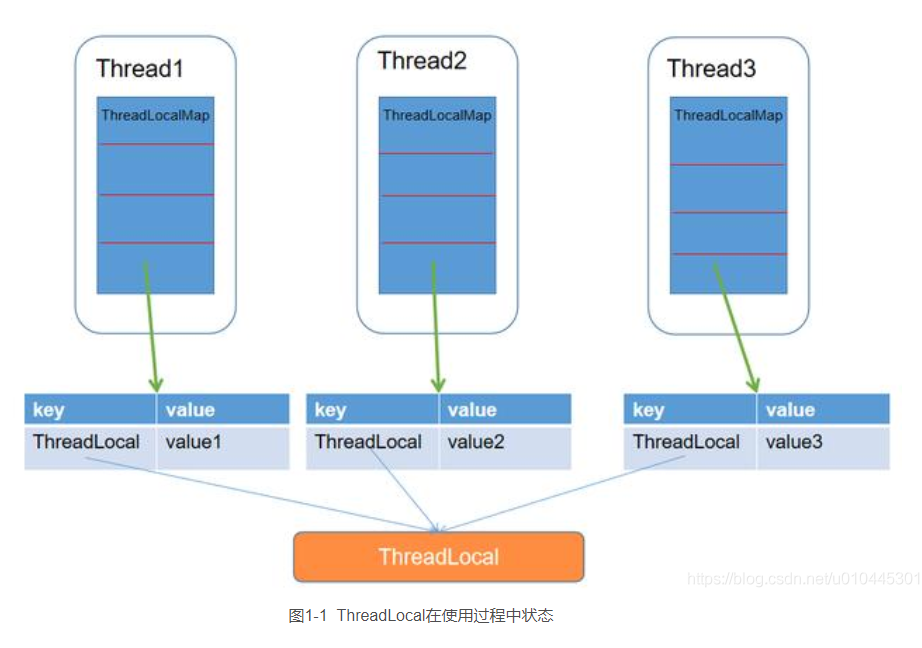
ThreadLocal叫做**线程变量**，意思是ThreadLocal中**填充的变量**属于当前线程，该变量对其他线程而言是隔离的，也就是说该变量是当前线程独有的变量。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

ThreadLoal 变量，线程局部变量，同一个 ThreadLocal 所包含的对象，在不同的 Thread 中有不同的副本。这里有几点需要注意：

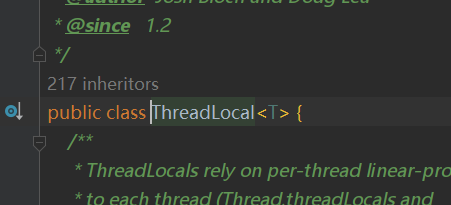
1、因为每个 Thread 内有自己的实例副本，且该副本只能由当前 Thread 使用。这是也是 ThreadLocal 命名的由来。

2、既然每个 Thread 有自己的实例副本，且其它 Thread 不可访问，那就不存在多线程间共享的问题。

ThreadLocal 提供了线程本地的实例。它与普通变量的区别在于，每个使用该变量的线程都会初始化一个完全独立的实例副本。ThreadLocal 变量通常被private static修饰。当一个线程结束时，它所使用的所有 ThreadLocal 相对的实例副本都可被回收。总的来说，ThreadLocal 适用于每个线程需要自己独立的实例且该实例需要在多个方法中被使用，也即变量在线程间隔离而在方法或类间共享的场景。



## 源码



## 构造函数

### public Thread()

### public Thread(Runnable target)

### public Thread(String name)

### Thread(Runnable target, AccessControlContext acc)

## 方法

### public T get()

返回此线程局部变量的当前线程副本中的值。

### public void set(T value)

将此线程局部变量的当前线程副本中的值设置为指定值。

### protected T initialValue()

返回此线程局部变量的当前线程的“初始值”。

### public void remove()

移除此线程局部变量当前线程的值。

除了这四个方法，ThreadLocal内部还有一个静态内部类ThreadLocalMap，该内部类才是实现线程隔离机制的关键，get()、set()、remove()都是基于该内部类操作。ThreadLocalMap提供了一种用键值对方式存储每一个线程的变量副本的方法，key为当前ThreadLocal对象，value则是对应线程的变量副本。

## 注意点

1、ThreadLocal实例本身是不存储值，它只是提供了一个在当前线程中找到副本值的key。

2、是ThreadLocal包含在Thread中，而不是Thread包含在ThreadLocal中，有些小伙伴会弄错他们的关系。

## ThreadLocal的原理

为每个线程提供独立的变量副本。

每个Thread对象内部都有一个ThreadLocalMap，这是一个类似HashMap的数据结构：

Key: ThreadLocal对象本身

Value: 该线程中存储的变量值

当你调用ThreadLocal.get()时，实际是：

1、获取当前线程的ThreadLocalMap

2、以当前ThreadLocal为key查找对应的value

3、返回该线程专属的变量副本

关键特点：

1、**线程隔离**: 每个线程看到的都是自己的变量副本，互不影响

**2、无锁设计**: 因为每个线程访问自己的数据，无需同步。

3、**弱引用**: ThreadLocalMap的key使用弱引用，避免内存泄漏。

简单示例：



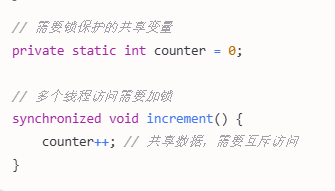
本质上是**用空间换隔离**，每个线程维护自己的变量存储空间，从而实现无锁的线程安全。

## 无锁的线程安全

### 基本概念

无锁的线程安全是指不使用传统的锁机制（如synchronized、ReentrantLock），但仍然保证多线程环境下数据的正确性。

### 传统加锁方式的问题



1、线程需要**排队等待**锁。

2、存在**性能开销**和**阻塞风险。**

### ThreadLocal的无锁方案



### 为什么无需加锁

核心原因：没有共享数据。

1、每个线程操作的是自己独有的数据副本

2、线程A修改counter不会影响线程B的counter

3、既然没有数据竞争，自然无需加锁保护

### 对比总结



本质：通过**数据隔离**避免了**数据竞争**，从而实现无锁的线程安全。

# 为什么wait()方法和 notify/notifyAll方法要在同步块中被调用

这是JDK 强制的,wait0)方法和 notify()/notifyAll0方法在调用前都必须先获得对象的锁。

# 多线程同步的方法

1、synchronized 关键字

2、Lock 锁实现

3、分布式锁

# Linux 环境下如何查找哪个线程使用 CPU 最长

1、获取项目的 pid，jps 或者 ps -efI grep java

2、top -H -p pid，顺序不能改变

# 死锁

## 基本概念

Java 中的死锁是一种编程情况，其中两个或多个线程被永久阻塞，Java 死锁情况出现至少两个线程和两个或更多资源。

## 发生死锁的根本原因

在申请锁时发生了交叉闭环申请。

# 怎么唤醒一个阻塞的线程

如果线程是因为调用了 wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出 InterruptedException 来唤醒它; 如果线程遇到了IO 阻塞无能为力，因为 lO是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

# 不可变对象对多线程有什么帮助

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行的效率。

# 如果你提交任务时，线程池队列已满，这时会发生什么

1、如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务。

2、如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来ArrayBlockingQueue 继续满，那么则会使用拒绝策略RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

# Java中用到的线程调度算法

抢占式，一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

# 线程调度器(Thread Schedule)和时间分片(TimeSlicing)

线程调度器是一个操作系统服务,它负责为 Runnable 状态的线程分配 CPU 时间一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。

时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择 (也就是说不要让你的程序依赖于线程的优先)。

# 单例模式的线程安全性

首先要说的是单例模式的线程安全意味着:某个类的实例在多线程环境下只会被创建一次出来。

单例模式有很多种的写法，我总结一下:

1、饿汉式单例模式的写法: 线程安全

2、懒汉式单例模式的写法: 非线程安全

3、双检索单例模式的写法:线程安全

# Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

# 线程类的构造方法、静态块是被哪个线程调用的

请记住: 线程类的构造方法、静态块是被 new这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了Thread1，main 函数中 new 了 Thread2，那么:

1、Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是Thread2 自己调用的。

2、 Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是Thread1 自己调用的。

# 同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。原则: 同步的范围越小越好。

# Java线程数过多会造成什么异常

## 1、线程的生命周期开销非常高

## 2、消耗过多的 CPU 资源

如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争 CPU资源时还将产生其他性能的开销。

## 3、降低稳定性

JVM 在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同并且承受着多个因素制约，包括JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出OutOfMemoryError 异常。

# 不可变类