JAVA多线程并发编程

JUC常用类

锁

- 如ReentratLock，ReadWriteLock。ReentrantLock重入锁，可以替代synchronized使用，并且有更多强大的功能，比如说可以中断锁，trylock，超时等待，公平锁等。
- ReadWriteLock，读写锁，更是对读和写进行了锁分离，在读多写少的场景下，能极大的提高程序的性能。

原子类

- 基本类，数组类，引用类等。AtomicInteger比较常用。使用原子类，可以不需要手动加锁，实现线程安全。
- 原子变量，由CAS实现，也就是比较并交换，只有当当前值和修改之间记录的值一样时，才会修改，它比的是地址。CAS是原子操作的一种，由cpu指令执行。
- CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。 CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许java调用其他语言。而compareAndSwapInt就是借助C来调用CPU底层指令实现的。

线程同步

- 使得线程间的同步更加容易。如CountDownLatch计数器，CyclicBarrier可重置计数器，semaphore计数器信号量。
- Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore可以指定多个线程同时访问某个资源。
- CountDownLatch （倒计时器）： 这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
- CyclicBarrier(循环栅栏)： 比如我初始化一个Cyc类参数是5，那么每await()一次，就+1，当到5的时候，所有await阻塞的线程会一起被唤醒。初始化参数可以循环使用。

线程管理

- Executors，ExecutorService方便线程池管理，线程提交等操作。Future和CompletableFuture方便接收异步调用结果和对返回结果进行后续操作。

并发集合类

- 提供ConcurrentHashMap，ConcurrentLinkedQueue，CopyOnWriteArrayList等，方便并发场景下的集合类操作。

JAVA线程实现/创建方式

继承 Thread 类

- Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它将启动一个新线程，并执行 run()方法。

```
public class MyThread extends Thread {
 public void run() {
 System.out.println("MyThread.run()");
 }
}
MyThread myThread1 = new MyThread();
myThread1.start();
```

实现 Runnable接口

- 如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口。

```
public class MyThread extends OtherClass implements Runnable {
 public void run() {
 System.out.println("MyThread.run()");
 }
}

//启动 MyThread，需要首先实例化一个 Thread，并传入自己的 MyThread 实例：
MyThread myThread = new MyThread();
Thread thread = new Thread(myThread);
thread.start();

//事实上，当传入一个 Runnable target 参数给 Thread 后，Thread 的 run()方法就会调用
target.run()
public void run() {
 if (target != null) {
 target.run();
 }
}
```

ExecutorService、Callable、Future 有返回值线程

- 有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程了。

```
//创建一个线程池
ExecutorService pool = Executors.newFixedThreadPool(taskSize);

// 创建多个有返回值的任务
List<Future> list = new ArrayList<Future>();
for (int i = 0; i < taskSize; i++) {
Callable c = new MyCallable(i + " ");

// 执行任务并获取 Future 对象
Future f = pool.submit(c);
list.add(f);
}

// 关闭线程池
pool.shutdown();

// 获取所有并发任务的运行结果
```

```
for (Future f : list) {
// 从 Future 对象上获取任务的返回值，并输出到控制台
System.out.println("res: " + f.get().toString());
}
```

基于线程池的方式

- 线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

```
// 创建线程池
 ExecutorService threadPool = Executors.newFixedThreadPool(10);

 while(true) {
    threadPool.execute(new Runnable() {

    // 提交多个线程任务，并执行
    @Override
    public void run() {
 System.out.println(Thread.currentThread().getName() + " is running ..");
        try {
        Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        }
    });
 }
}
```

4 种线程池

- Java 里面线程池的顶级接口是 **Executor**，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。

newCachedThreadPool

- 创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并 从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

newFixedThreadPool

- 创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何 线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

newScheduledThreadPool

- 创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

```
ScheduledExecutorService scheduledThreadPool= Executors.newScheduledThreadPool(3);
    scheduledThreadPool.schedule(newRunnable(){
        @Override
        public void run() {
        System.out.println("延迟三秒");
        }
```

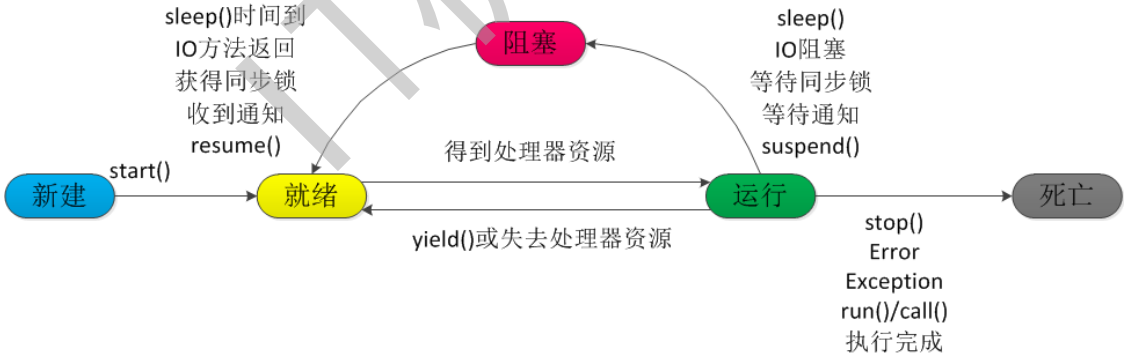
- ```
  }, 3, TimeUnit.SECONDS);
cheduledThreadPool.scheduleAtFixedRate(newRunnable(){
    @Override
    public void run() {
        System.out.println("延迟 1 秒后每三秒执行一次");
    }
},1,3,TimeUnit.SECONDS);
```

## newSingleThreadExecutor

- Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程）,这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去!

# 线程生命周期(状态)

- 当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换。

- 


## 新建状态（NEW）

- 当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值

## 就绪状态（RUNNABLE）

- 如果处于就绪状态的线程获得了 CPU，开始执行 run()方法的线程执行体，则该线程处于运行状态。

## 运行状态（RUNNING）
```

- 如果处于就绪状态的线程获得了 CPU，开始执行 run()方法的线程执行体，则该线程处于运行状态。

阻塞状态（BLOCKED）

- 阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状态。

线程死亡（DEAD）

线程会以下面三种方式结束，结束后就是死亡状态。

正常结束

- run()或 call()方法执行完成，线程正常结束。

异常结束

- 线程抛出一个未捕获的 Exception 或 Error。

调用 stop

- 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。
- 程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用 thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因此，并不推荐使用 stop 方法来终止线程。

sleep 与 wait 区别

- 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于Object 类中的。
- sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然 保持者，当指定的时间到了又会自动恢复运行状态。
- 在调用 sleep()方法的过程中，线程不会释放对象锁。
- 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

start 与 run 区别

- **start（）**方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码。
- 通过调用 Thread 类的 start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。
- 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

JAVA 后台线程

- 定义：守护线程--也称"服务线程"，他是后台线程，它有一个特性，即为用户线程 提供公共服务，在没有用户线程可服务时会自动离开。
- example: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread, 程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是 JVM 上仅剩的线 程时，

垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

- 生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周 期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。当 JVM 中所有的线程都是守护线程的时候，JVM 就可以退出了；如果还有一个或以上的非守护线程则 JVM 不会退出。

JAVA 锁

乐观锁

- 乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新）， 如果失败则要重复读-比较-写的操作。
- java 中的乐观锁基本都是通过 CAS 操作实现的，CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。

CAS

- CAS（Compare And Swap/Set）比较并交换，CAS 算法的过程是这样：它包含 3 个参数 CAS(V,E,N)。V 表示要更新的变量(内存值)，E 表示预期值(旧的)，N 表示新值。当且仅当 V 值等于 E 值时，才会将 V 的值设为 N，如果 V 值和 E 值不同，则说明已经有其他线程做了更新，则当 前线程什么都不做。最后，CAS 返回当前 V 的真实值。
- CAS 操作是抱着乐观的态度进行的(乐观锁)，它总是认为自己可以成功完成操作。当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

ABA 问题

- CAS 会导致“ABA 问题”。CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。
- 比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。
- 部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少

AQS

- AbstractQueuedSynchronizer 类如其名，抽象的队列式的同步器，AQS 定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的 ReentrantLock/Semaphore/CountDownLatch

悲观锁

- 悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会 block 直到拿到锁。

java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如 RetreenLock。

自旋锁

- 自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。
- 线程自旋是需要消耗 cup 的，说白了就是让 cup 在做无用功，如果一直获取不到锁，那线程也不能一直占用 cup 自旋做无用功，所以需要设定一个自旋等待的最大时间。
- 如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点

- 自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！
- 但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合 使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cup 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁。

Synchronized 同步锁

- synchronized 它可以把任意一个非 NULL 的对象当作锁。他属于独占式的悲观锁，同时属于可重入锁。

Synchronized 作用范围

- 作用于方法时，锁住的是对象的实例(this)；
- 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带 PermGen（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
- synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

ReentrantLock

- ReentantLock 继承接口 Lock 并实现了接口中定义的方法，他是一种可重入锁，除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。

ReentrantLock 与 synchronized

- ReentrantLock 通过方法 lock()与 unlock()来进行加锁与解锁操作，与 synchronized 会被 JVM 自动解锁机制不同，ReentrantLock 加锁后需要手动进行解锁。为了避免程序出现异常而无法正常解锁的情况，使用 ReentrantLock 必须在 finally 控制块中进行解锁操作。
- ReentrantLock 相比 synchronized 的优势是可中断、公平锁、多个锁。这种情况下需要使用 ReentrantLock。

synchronized 和 ReentrantLock 的区别

- ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁

- ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的 不可用性提供了更高的灵活性
- ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的
- ReentrantLock 可以实现公平锁
- ReentrantLock 通过 Condition 可以绑定多个条件
- 底层实现不一样， synchronized 是同步阻塞，使用的是悲观并发策略，lock 是同步非阻 塞，采用的是乐观并发策略
- Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现。
- synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生； 而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象， 因此使用 Lock 时需要在 finally 块中释放锁。
- Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。
- Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等

Semaphore 信号量

- Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做完自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore 可以用来构建一些对象池，资源池之类的，比如数据库连接池

实现互斥锁（计数器为 1）

- 我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。

可重入锁（递归锁）

- 本文里面讲的是广义上的可重入锁，而不是单指 JAVA 下的 ReentrantLock。可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。在 JAVA 环境下 ReentrantLock 和 synchronized 都是 可重入锁。

AtomicInteger

- 首先说明，此处 AtomicInteger ，一个提供原子操作的 Integer 的类，常见的还有 AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference 等，他们的实现原理相同，区别在与运算对象类型的不同。令人兴奋地，还可以通过 AtomicReference将一个对象的所有操作转化成原子操作。
- 我们知道，在多线程程序中，诸如++i 或 i++等运算不具有原子性，是不安全的线程操作之一。通常我们会使用 synchronized 将该操作变成一个原子操作，但 JVM 为此类操作特意提供了一些同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常AtomicInteger 的性能是 ReentantLock 的好几倍。

锁得类型

公平锁和非公平锁

- **非公平锁**
- JVM 按随机、就近原则分配锁的机制则称为不公平锁，ReentrantLock 在构造函数中提供了 是否公平锁的初始化方式，默认为非公平锁。非公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用非公平锁的分配机制。
- **公平锁**

- 公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式来定义公平锁。

共享锁和独占锁

- java 并发包提供的加锁模式分为独占锁和共享锁。
- 独占锁模式下，每次只能有一个线程能持有锁，ReentrantLock 就是以独占方式实现的互斥锁。独占锁是一种悲观保守的加锁策略，它避免了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。
- 共享锁则允许多个线程同时获取锁，并发访问 共享资源，如：ReadWriteLock。共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。

重量级锁（Mutex Lock）

- Synchronized 是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的 Mutex Lock 来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么 Synchronized 效率低的原因。因此，这种依赖于操作系统 Mutex Lock 所实现的锁我们称之为“重量级锁”。JDK 中对 Synchronized 做的种种优化，其核心都是为了减少这种重量级锁的使用。JDK1.6 以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和 “偏向锁”。

轻量级锁

- 锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。
- “轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

偏向锁

- 大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护。
- 引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次 CAS 原子指令，而偏向锁只需要在置换ThreadID 的时候依赖一次 CAS 原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所 以偏向锁的撤销操作的性能损耗必须小于节省下来的 CAS 原子指令的性能消耗）。
- 上面说过，轻 量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

分段锁

- 分段锁也并非一种实际的锁，而是一种思想 ConcurrentHashMap 是学习分段锁的最好实践。

锁优化

- **减少锁持有时间**
- 只用在有线程安全要求的程序上加锁
- **减小锁粒度**
- 将大对象（这个对象可能会被很多线程访问），拆成小对象，大大增加并行度，降低锁竞争。 降低了锁的竞争，偏向锁，轻量级锁成功率才会提高。最最典型的减小锁粒度的案例就是 ConcurrentHashMap。

- **锁分离**
- 最常见的锁分离就是读写锁 ReadWriteLock，根据功能进行分离成读锁和写锁，这样读读不互斥，读写互斥，写写互斥，即保证了线程安全，又提高了性能。读写分离思想可以延伸，只要操作互不影响，锁就可以分离。比如LinkedBlockingQueue 从头部取出，从尾部放数据
- **锁粗化**
- 通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。但是，凡事都有一个度，如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化 。
- **锁消除**
- 锁消除是在编译器级别的事情。在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作，多数是因为程序员编码不规范引起。

同步锁与死锁

同步锁

- 当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。 Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

死锁

- 何为死锁，就是多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

线程池原理

- 线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。他的主要特点为：线程复用；控制最大并发数；管理线程。

线程池的参数

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize, long
keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue) {
this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
Executors.defaultThreadFactory(), defaultHandler);
}
```

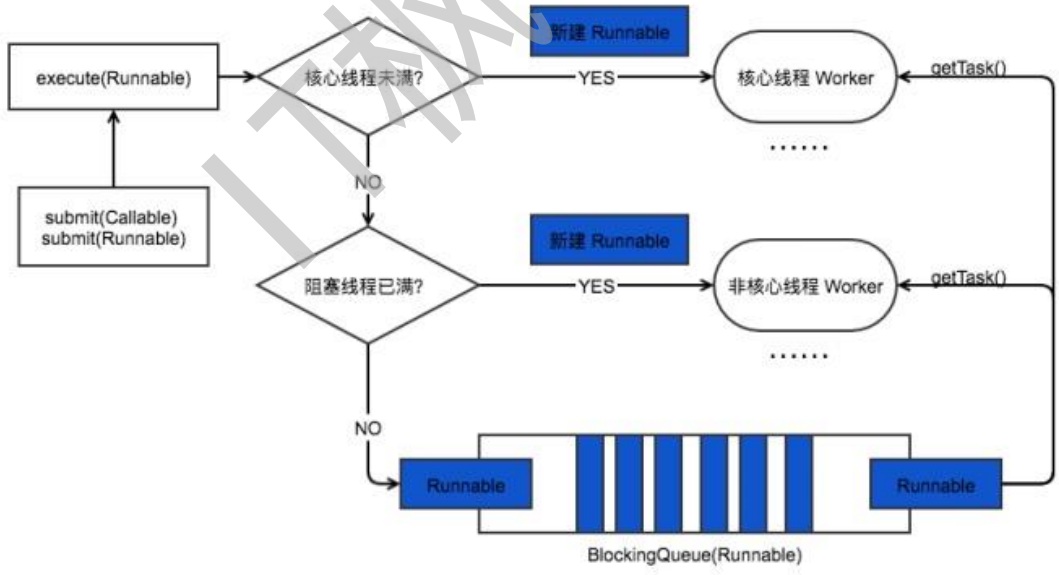
- corePoolSize：指定了线程池中的线程数量。
- maximumPoolSize：指定了线程池中的最大线程数量。
- keepAliveTime：当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多次时间内会被销毁。
- unit：keepAliveTime 的单位。
- workQueue：任务队列，被提交但尚未被执行的任务。
- threadFactory：线程工厂，用于创建线程，一般用默认的即可。
- handler：拒绝策略，当任务太多来不及处理，如何拒绝任务。

拒绝策略

- 线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。
- AbortPolicy： 直接抛出异常，阻止系统正常运行。
- CallerRunsPolicy： 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。
- DiscardOldestPolicy： 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
- DiscardPolicy： 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢 失，这是最好的一种方案。

Java 线程池工作过程

- 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。
- 当调用 execute() 方法添加一个任务时，线程池会做如下判断：
  - 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
  - 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
  - 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
  - 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。
- 当一个线程完成任务时，它会从队列中取下一个任务来执行。
- 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运 行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。
-



CyclicBarrier、CountDownLatch、Semaphore的用法

CountDownLatch（线程计数器）

- CountDownLatch 类位于 java.util.concurrent 包下，利用它可以实现类似计数器的功能。比如有一个任务 A，它要等待其他 4 个任务执行完毕之后才能执行，此时就可以利用 CountDownLatch 来实现这种功能了。

```
final CountDownLatch latch = new CountDownLatch(2);
    new Thread(){public void run() {
            System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");
            Thread.sleep(3000);
            System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");
            latch.countDown();
    };}.start();
    new Thread(){ public void run() {
            System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");
            Thread.sleep(3000);
            System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");
            latch.countDown();
    };}.start();
     System.out.println("等待 2 个子线程执行完毕...");
     latch.await();
     System.out.println("2 个子线程已经执行完毕");
     System.out.println("继续执行主线程");
    }
```

CyclicBarrier（回环栅栏-等待至 barrier 状态再全部同时执行）

- 字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后,CyclicBarrier 可以被重用。

Semaphore（信号量-控制同时访问的线程个数）

- Semaphore 翻译成字面意思为 信号量，Semaphore 可以控制同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。
- CountDownLatch 和 CyclicBarrier 都能够实现线程之间的等待，只不过它们侧重点不同；CountDownLatch 一般用于某个线程 A 等待若干个其他线程执行完任务之后执行；而 CyclicBarrier 一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；另外，CountDownLatch 是不能够重用的，而 CyclicBarrier 是可以重用的。
- Semaphore 其实和锁有点类似，它一般用于控制对某组资源的访问权限。

volatile 关键字的作用（变量可见性、禁止重排序）

- Java 语言提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。volatile 变量具备两种特性，volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取 volatile 类型的变量时总会返回最新写入的值
- **变量可见性**
- 其一是保证该变量对所有线程可见，这里的可见性指的是当一个线程修改了变量的值，那么新的值对于其他线程是可以立即获取的。
- **禁止重排序**
- volatile 禁止了指令重排.

ThreadLocal 作用（线程本地存储）

- ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，ThreadLocal 的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。

ThreadLocalMap（线程的一个属性）

- 每个线程中都有一个自己的 ThreadLocalMap 类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。
- 将一个共用的 ThreadLocal 静态实例作为 key，将不同对象的引用保存到不同线程的 ThreadLocalMap 中，然后在线程执行的各处通过这个静态 ThreadLocal 实例的 get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。
- ThreadLocalMap 其实就是线程里面的一个属性，它在 Thread 类中定义

  ThreadLocal.ThreadLocalMap threadLocals = null;
- 

- 最常见的 ThreadLocal 使用场景为 用来解决 数据库连接、Session 管理等。

ConcurrentHashMap 并发

减小锁粒度

- 减小锁粒度是指缩小锁定对象的范围，从而减小锁冲突的可能性，从而提高系统的并发能力。减小锁粒度是一种削弱多线程锁竞争的有效手段，这种技术典型的应用是 ConcurrentHashMap(高性能的 HashMap)类的实现。对于 HashMap 而言，最重要的两个方法是 get 与 set 方法，如果我们对整个 HashMap 加锁，可以得到线程安全的对象，但是加锁粒度太大。Segment 的大小也被称为 ConcurrentHashMap 的并发度。

ConcurrentHashMap 分段锁

- ConcurrentHashMap，它内部细分了若干个小的 HashMap，称之为段(Segment)。默认情况下一个 ConcurrentHashMap 被进一步细分为 16 个段，既就是锁的并发度。
- 如果需要在 ConcurrentHashMap 中添加一个新的表项，并不是将整个 HashMap 加锁，而是首先根据 hashcode 得到该表项应该存放在哪个段中，然后对该段加锁，并完成 put 操作。在多线程 环境中，如果多个线程同时进行 put操作，只要被加入的表项不存放在同一个段中，则线程间可以 做到真正的并行。