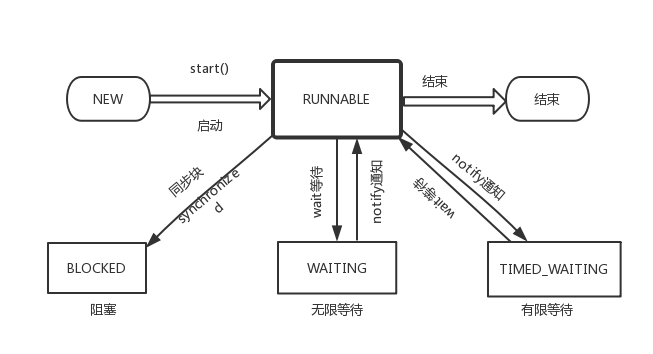
# 高并发编程

## 线程基础

### 线程的基本操作

#### 线程的生命周期



NEW： NEW 状态表示刚刚创建的线程，这种线程还没开始执行。当调用start()方法时,线程才开始执行。

RUNNABLE: 当线程执行时，就处于RUNNABLE 状态,表示线程所需要的一切资源都已经准备好了。

BLOCKED: 当线程在执行的过程中遇到synchronized同步块，就会处于BLOCKED阻塞状态,这是线程会暂停执行，知道获得请求的锁。

WAITING和TIMED\_WAITING: 这个两个都表示等待状态,一般TIMED\_WAITIING会进行有限的等待而WAITING的线程正在等待进行一个特殊的事件。比如，通过wait()方法等待的线程在等待notify()方法，而通过join()方法等待的线程会等待目标线程的终止。

TERMINATED: 表示线程结束。

#### 新建线程

1. **怎么新建一个线程？**

通过new Thread().start();这样就新建了一个线程。

注意: 我们必须调用Thread对象实例的start()方可创建新线程，

而不是直接调用run()方法。

1. **怎么让新线程执行我们自己的代码？**
2. 通过继承Thread类，重写run()方法。
3. 通过实现Runnable接口，实现run()方法。

然后调用start()方法启动线程。

3. 使用匿名内部类创建线程

#### 终止线程

1. **一般来说，当线程执行完程序之后就会自行结束，无需手工关闭。但是有些程序是常驻后台提供服务的线程。那么怎么安全的关闭这些线程？**

Thread类提供了一个stop()方法，如果调用了stop()方法就会立即停止线程。但是这个方法如果使用不当，会造成线程不安全。因为，stop()方法调用后，会立即停止线程，并释放这个线程所持有的锁,而这个锁恰好就是用来维持对象一致性的。比如线程A持有Person对象锁，线程A准备修改Person对象的name和age属性，但是修改了name，再修改age之前调用了stop()方法。那么这个对象的一致性就被破坏了，如果再来一个线程B，它读取的数据，就被修改过的数据。

我们可以自定义一个终止标签。当标签设置为true时，安全退出。

#### 线程中断

在java中,线程中断是一种很重要的线程协作机制。不同于终止线程，

线程中断不会立即停止线程，而是一种平滑中断的手段，当我们想通知某个线程中断时，这个线程要不要中断是由该线程决断。

关于线程中断的三个方法：

public void Thread.interrupt(); //中断线程

public void Thread.isInterrupted(); //判断是否被中断

public void Thread.interrupted();//判断是否被中断，并清除当前中断状态

我们可以通过Thread.interrupte();方法通知线程中断,然后该线程通过Thread.isInterrupted();判断是否被中断,然后安全的退出线程。

注意： Thread.sleep()方法会抛出一个InterruptedException()中断异常。当线程在sleep()休眠时,如果被中断，就会产生这个异常。此时，它会将中断标记清除。



我们需要在catch代码块中，重新标记中断。

#### wait()和notify()方法

wait()和notify()方法时Object类中的方法。

当一个obj对象调用wait()方法时会释放obj监视器。由其他线程获得，其他线程调用obj.notify()方式，会随机唤醒一个线程，被唤醒的线程不会立即获得obj监视器,必须要等待获取。notifyAll()会唤醒所有线程。

#### 线程优先级

现代操作系统基本采用时分的形式调度运行的线程, 线程分配得到的时间片的多少决定了线程使用处理器资源的多少，也对应了线程优先级这个概念。

在Java线程中，通过一个init priority来控制优先级，范围为1-19，其中

10最高，默认值为5.通过Thread中的

**setPriority(10)** 来设置

#### 7. join()和yield()方法

**yield()** 方法:

* yield是一个本地的静态方法
* yield告诉当前正在执行的线程把运行机会交给线程池中相同优先级的线程
* yield不能保证当前线程迅速切换到可运行状态
* yield仅能使一个线程从运行状态转到可运行状态，而不是阻塞等待状态

**join()** 方法:

join方法可以使得一个线程等待另一个线程执行完毕后再执行。

例如：在A线程中调用B线程的join方法，那么A线程会等待B线程执行完毕之后再执行。

### 线程安全

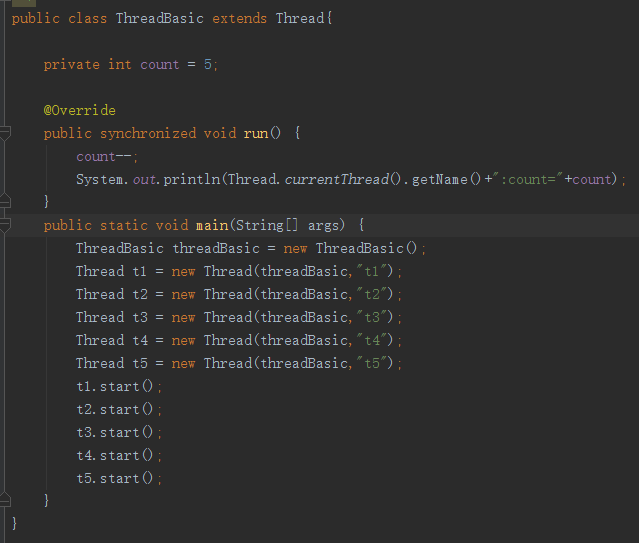
#### 什么是线程安全

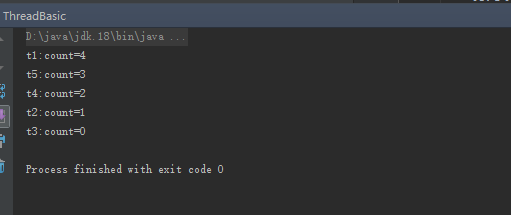
当多个线程访问同一个类(对象或方法)时,这个类（对象或方法）都能表现出正确的行为。那么就说这个类(对象或方法)是线程安全的。

#### Synchronized

可以在任意对象及方法上加锁，而加锁的这段代码称为“互斥区”或“临界区”。

#### 实例





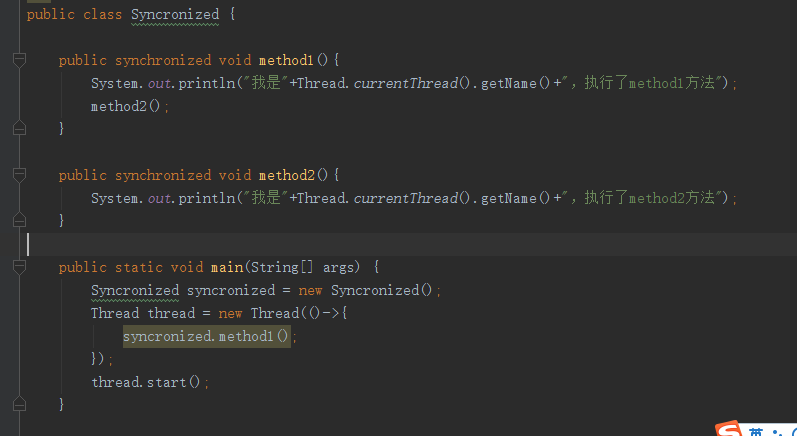
**总结：”**

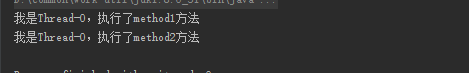
当多个线程访问theadBasic对象的run方法时,会以排队的方式进行处理(这里的排队是按照CPU分配的先后顺序而定的)，一个线程想要执行synchronized关键字修饰的代码，首先要尝试获得锁,如果拿到锁,执行synchronized里面的代码；拿不到锁，这个线程就会不断尝试获得锁，直到拿到为止,而且是多个线程同时去竞争这把锁(也就是会有锁竞争的问题)。

#### Synchronized锁重入

关键字Synchronized具有重入的功能，也就是使用synchronizd时，当一个线程获得了一个对象的锁,再次请求此对象时是可以再次获得该对象的锁。

**实例：**





#### Synchronized异常

我们的代码在执行过程中肯定会出现异常的情况，那么在Synchronized中出现异常怎么处理？

1. 将异常信息和异常数据记录日志，然后跳过该异常继续执行。
2. 如果是原子性操作则直接回滚,或将操作作废，然后邮件报警。

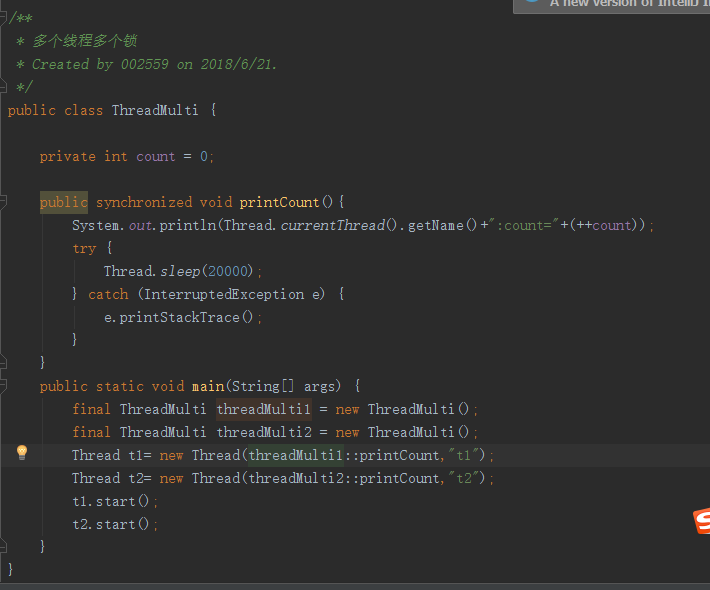
#### Synchronized使用的锁类型

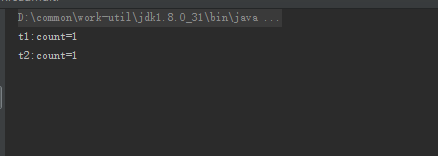
Synchroized 可以使用各种对象或者类作为锁。在使用的过程中需要注意的是。

1. 不要使用字符串常量作为锁，会出现一些奇怪的问题。
2. 如果使用了字符串对象作为锁，那么在执行过程中，不能修改字符串对象的引用，其他对象也一样。

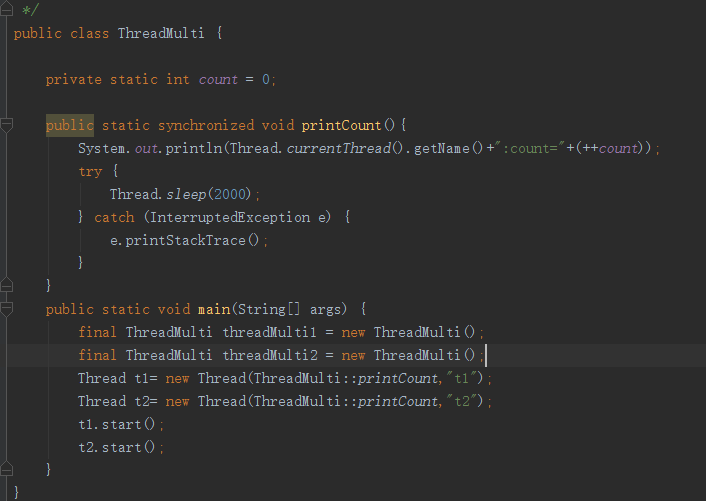
### 多个线程多个锁

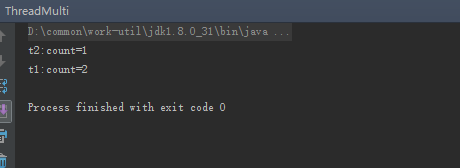
多个线程，每个线程都能获取自己指定的锁，分别获取锁后执行synchronized里面的代码。





静态方法





**总结：**

关键字synchronized取得的锁都是对象锁，而不是把一段代码当做锁,执行哪个对象的方法，那么就持有该对象的锁。两个线程分别执行两个对象里面的方法时，获取的是两个对象各自的锁，所以互不影响。

有一种情况是相同的锁，那就是synchronized修饰的静态方法，表示锁定class类。+

### 对象锁的同步和异步

#### 同步(synchronized)

同步的概念是共享,如果不是共享的资源，就没必要同步。

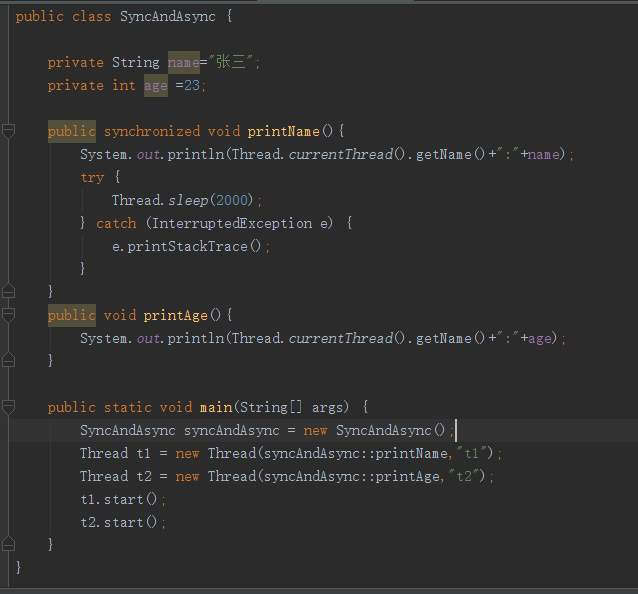
同步的目的是为了线程安全，对于线程安全来说,需要满足两个特性：

原子性、可见性。同步在代码之中是顺序执行

#### 异步(asynchronized)

异步的概念是独立，线程相互之间不受任何制约。

#### 实例



**总结：**

如果t1，t2线程都调用 syncAndAsync对象的printName方法，那么这就是同步。假设t1线程先进入printName执行，那么t2线程必须等待t1线程释放锁。

如果t1线程调用printName方法，而t2线程调用printAge方法

那么这就是异步。t1,t2线程互不影响。

### 5. 有关线程的问题

#### 1. 脏读

当我们设计程序的时候，一定要考虑程序的整体性和一致性。也就是共享资源的时候。

#### 2. 死锁

### 6. 守护线程和非守护线程

**守护线程**是一些特殊的线程，在后台默默的为系统服务的线程，例如GC线程。守护线程和主线程有直接关系，主线程终止了守护线程也会跟着终止。通过 Thread类中

的**setDaemon(true)** 将线程设置为守护线程;

**非守护线程**是只和主线程无关的、独立的线程

### 7. volatile关键字

#### 1. 概念

volatile的作用主要就是使变量在多个线程间可见

#### 2. 原理

在Java中，每一个线程都会有一块工作内存区，其中存放着所有线程共享的主内存中的变量的拷贝。当线程执行时，他在自己的工作内存区中操作这些变量。为了存取一个共享的变量，一个线程通常获取锁定并去清除它的内存工作区，把这些共享变量从所有线程的共享内存区中的正确装入到他自己所在的工作内存中，当线程解锁时保证该工作内存区中变量的值写回到共享内存中。

一个线程可以执行的操作有使用(use)、赋值(assign)、装载(load)、存储(store)、锁定(lock)、解锁(unlock)

而主内存中可以执行的操作有 读(read)、写(write)、锁定(lock)、解锁(unlock)，每个操作都是原子操作。

volatile的作用就是强制线程到主内存(共享内存)里去读取变量，而不去线程工作内存区里取读取，从而实现了多个线程间的变量可见。也就是满足线程安全的可见性。

#### 3. volatile执行流程

#### 4. volatile的特性

**volatile只能保证变量在多个线程之间的可见性，但是不能保证变量的原子性。**

**如果需要保证变量的原子性，可以使用Atomic 系列的相关类。**

### 8. 线程间通信

#### 1. 概念

**线程是操作系统中独立的个体，但这些个体如果不经过特殊处理就不能成为一个整体，线程间的通信就成为整体的必用方式之一。当线程存在通信指挥，系统间的交互性会更强大，在提高CPU利用率的同时还会使开发人员对线程任务在处理的过程中进行有效的把控与监督.**

**1. 使用wait / notify 方法实现线程间的通信:**

1. **wait和notify必须配合synchronized关键字使用**
2. **wait方法释放锁，notify方法不释放锁**

**2. 使用CountDownLatch**

#### 2. 使用wait/notify 方法实现线程间通信

**package** com.rxl.concurrent;  
  
**import** java.util.ArrayList;  
**import** java.util.Iterator;  
**import** java.util.List;  
**import** java.util.Scanner;  
  
*/\*\*  
 \* 使用notify和wait方法实现线程间的通信  
 \*/***public class** NotifyWithWaitTest {  
  
 **public static void** main(String[] args) {  
  
 **final** List<String> container = **new** ArrayList<>(20);  
 **final** Object lock = **new** Object();  
  
 *// 生产者* Thread producer = **new** Thread(() -> {  
 Scanner scanner = **new** Scanner(System.***in***);  
 System.***out***.println(**"请在下方输入消息："**);  
 **while** (scanner.hasNext()){  
 **synchronized** (lock){  
 String msg = scanner.nextLine();  
 container.add(msg);  
 System.***out***.println(Thread.*currentThread*().getName() + **"【生产消息】："** + msg);  
 lock.notify();  
 }  
 }  
 }, **"producer-service"**);  
  
 *// 消费者* Thread consumer = **new** Thread(() -> {  
 **while** (**true**){  
 **synchronized** (lock){  
 **if** (container.isEmpty()){  
 **try** {  
 lock.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 Iterator<String> iterator = container.iterator();  
 **while** (iterator.hasNext()){  
 System.***out***.print(Thread.*currentThread*().getName());  
 System.***out***.println(**"【消费消息】： "** + iterator.next());  
 iterator.remove();  
 }  
 }  
 }  
 }, **"consumer-service"**);  
  
 producer.start();  
 consumer.start();  
 }  
  
  
  
  
  
}

#### CountDownLatch实现线程通信

较wait/notify方法，CountDownLatch类更为强大，它不会有notify延迟的缺点。

### 9. Queue队列

#### 自定义阻塞Queue

**package** com.rxl.concurrent;  
  
  
**import** java.util.LinkedList;  
**import** java.util.concurrent.atomic.AtomicInteger;  
  
**public class** BlockQueue {  
  
 **private volatile** LinkedList<String> **container** = **new** LinkedList<>();  
  
 **private final int max**; *// 队列可以容纳的最大数量* **private final** Object **lock** = **new** Object();  
 **private final** AtomicInteger **size** = **new** AtomicInteger(0);  
  
 **public** BlockQueue(**int** max) {  
 **this**.**max** = max;  
 }  
  
 */\*\*  
 \* 如果当前队列已经到达最大值时, 线程等待消费(阻塞)  
 \** ***@param* msg** *\*/* **public void** put(String msg){  
 **synchronized** (**lock**){  
 **if** (**size**.get() == **max**){  
 **try** {  
 System.***out***.println(**"当前队列已满，等待消费"**);  
 **lock**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **container**.add(msg);  
 System.***out***.println(**"存放数据："** + msg);  
 **size**.incrementAndGet();  
 **lock**.notify();  
 }  
 }  
  
 */\*\*  
 \* 获取数据  
 \** ***@return*** *\*/* **public** String take(){  
 String ret;  
 **synchronized** (**lock**){  
 **if** (**size**.get() == 0){  
 **try** {  
 **lock**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ret = **container**.getFirst();  
 System.***out***.println(**"获取数据："** + ret);  
 **container**.removeFirst();  
 **size**.decrementAndGet();  
 **lock**.notify();  
 }  
 **return** ret;  
 }  
}

#### JDK提供的并发Queue

**在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队列，一个是以BlockingQueue接口为代表的阻塞队列，无论哪种都继承自Queue**

**阻塞式队列：**

**在存的时候，如果超出队列总数，这时候会进行等待。**

**在取的时候，如果队列为空的情况下，这时候也会进行等待。**

**阻塞式队列最大的好处就是防止容器溢出，防止数据丢失。**

##### ConcurrentLinkedQueue高性能队列

**是一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能，通常ConcurrentLinkedQueue性能好于BlockingQueue。它是一个基于链接节点的无界线程安全队列。改队列的元素遵循先进先出的规则。头是最先加入的，尾是最近加入的，该队列不允许null元素**

**方法：**

**add()和offer()都是加入元素的方法(在ConcurrentLinkedQueue中，这个两个方法没有任何区别。)**

**poll()和peek()都是取头部元素节点，区别在于前者会删除元素，后者不会**

##### BlockQueue阻塞队列接口

**ArrayBlockingQueue：**

**基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护一个定长数组，以便缓存队列中的数据对象，其内部没实现读写分离，也就意味着生产和消费不能完全并行，长度是需要定义的，可以指定先进先出或者先进后出，也叫有界队列。**

**LinkedBlockingQueue:**

**基于链表的阻塞队列，同ArrayBlockingQueue类似，其内部也维持一个数据缓冲队列(该队列由一个链表构成)，LinkedBlockingQueue之所以能够高效的处理并发数据，是因为其内部实现采用分离锁(读写分离两个锁)，从而实现生产这和消费这操作的完全并行运行。是一个无界队列。**

**SynchronousQueue：**

**一种没有缓冲的队列，生产者产生的数据直接会被消费者获取并消费。**

**是一种按需添加的队列，如果有线程调用take()方法, 那么add()方法调用后则直接推送到该线程。**

**PriorityBlockingQueue:**

**基于优先级的阻塞队列(优先级判断通过构造函数传入的Compator对象决定，也即是说传入队列的对象必须实现Comparable接口).,在试下PriorityBlockingQueue时，内部控制现场同步的锁采用的是公平锁，他也是一个无界的队列**

**必须调用take()方法后才会排序，而不是在添加的时候排序**

**DelayQueue:**

**带有延迟时间的Queue，其中的元素只有当指定的延迟时间到了，才能够从队列中获取到该元素，DelayQueue中的元素必须实现Delayed接口，DelayQueue是一个没有大小限制的队列，应用场景很多，比如对缓存超时的数据进行移除，任务超时处理、空闲连接的关闭等等。**

### 10. 单例与多线程-ThreadLocal

#### 1. 概念

**线程局部变量，是一种多线程间并发访问变量的解决方案。与其synchronized等枷锁的方式不同，ThreadLocal完成不提供锁,而使用以空间换时间的手段，为每个线程提供变量的独立副本，以保障线程安全。**

#### 2. 性能

**从性能上说，ThreadLocal不具有绝对的优势，在并发不是很高的时候，加锁的性能会更好，但作为一套与锁完全无关的线程安全解决方案，在高并发量或者竞争激烈的场景，使用ThreadLocal可以在一定程度上减少锁竞争。**

#### ThreadLocal的工作原理

**set：**

1. **首先获取当前线程对象**
2. **从当前线程中取出threadLocals成员变量**
3. **将值绑定到map中，其中key是threadLocal对象this**

**值是用户需要存储的值。**

**所以说，实现线程间的局部变量的原理是，将****ThreadLocal绑定到当前线程ThreadLocalMap成员变量中，其中将ThreadLocal对象作为key, 用户需存储的值作为value。而不是将当前线程绑定到ThreadLocal中.**

**public void** set(T value) {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**)  
 map.set(**this**, value);  
 **else** createMap(t, value);  
}

// 在Thread类中有一个ThreadLocalMapd的成员变量:

// ThreadLocal.ThreadLocalMap **threadLocals** = **null**;

// 在第一次调用ThreadLocal的set方法之前创建该对象

ThreadLocalMap getMap(Thread t) {  
 **return** t.**threadLocals**;  
}

#### 内存泄露

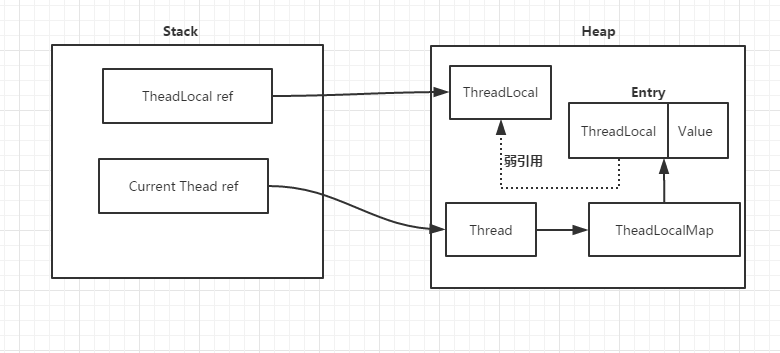
**在Java里面存在强引用、软引用、弱引用、虚引用的区别。**

**强引用：类似于**

**A a = new A();**

**B b = new B(a);S**

**考虑到GC的情况下，如果将a = null; 但是b还是存在a的引用，所以a的内存并不会被释放，如果b一直存在，则就造成了内存泄露。**



**ThreadLocal里面ThreadLocalMap的Entry是一个虚引用。**

### 11. 并发类容器

#### ConcurrentMap

**ConcurrentMap**有两个重要的实现:

ConcurrentHashMap

ConcurrentSkipListMap (支持并发排序功能，弥补ConcurrentHashMap)

##### C:\Users\002559.XHH\Downloads\未命名文件 (1).jpg1. ConcurrentHashMap原理

ConcurrentHashMap内部使用段来标识这些不同的部分，每个段其实就是一个HashTable,他们有自己的锁。只要多个修改操作发生在不同的段上，他们就可以并发进行访问。把一个整体分成了16个段，也就是最高支持16个线程的并发修改操作。这也是在多线程场景时减小锁的粒度从而降低锁竞争的一种方案。并且代码中大多共享变量使用volatile关键字声明，目的是第一时间获取修改的内容，性能非常好。

#### Copy-On-Write容器

**Copy-On-Write简称COW，是一种用于程序设计中的优化策略。**

**JDK里面的COW容器有两种：CopyOnWriteArrayList和CopyOnWriteArraySet,Cow容器非常有用。**

##### 1. CopyOnWrite原理

CopyOnWrite容器即写时复制的容器。通俗的理解就是当我们往一个容器添加元素的时候，不直接网当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

## 线程池

### 什么是线程池

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序，都可以使用线程池。

### 线程池的作用

**线程池的好处：**

* **降低资源消耗，不需要重复的创建线程，减少线程创建和销毁带来的消耗**
* **提高响应速度，不需要等待线程创建，可以立即执行**
* **提高线程的可管理性，使用线程池可以进行统一的分配、调优和监控。**

### 线程池的分类

**ThreadPoolExecutor**

### 线程池四种创建方式

Java通过Executors(JDK1.5并发包)提供四种线程池，分别为：

**newCachedThreadPool :**

**创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。**

**newFixedThreadPool**

**创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待**

**newScheduledThreadPool**

**创建一个定长线程池，支持定时及周期性任务执行。**

**newSingleThreadExecutor**

**创建一个单线程化的线程池，它会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行**

#### *newCachedThreadPool*

**package** com.rxl.concurrent.threadPool;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.ThreadFactory;  
**import** java.util.concurrent.atomic.AtomicInteger;  
  
*/\*\*  
 \* 创建一个缓存的线程池，如果线程池数量大于任务数量，则回收空闲线程。  
 \*/***public class** NewCachedThreadPoolDemo {  
  
  
 **public static void** main(String[] args) {  
  
 ThreadFactory factory = **new** ThreadFactory() {  
 **private** AtomicInteger **count** = **new** AtomicInteger(0);  
  
 @Override  
 **public** Thread newThread(Runnable r) {  
 Thread thread = **new** Thread(r);  
 thread.setName(**"print-service-"** + **count**.incrementAndGet());  
 thread.setPriority(10);  
 **return** thread;  
 }  
 };  
  
 ExecutorService executorService = Executors.*newCachedThreadPool*(factory);  
 *//* Runnable run = () -> {  
 System.***out***.println(Thread.*currentThread*().getName() + **"我是猪"**);  
 };  
  
 **for** (**int** i = 0; i < 100; i++) {  
 executorService.execute(run);  
 }  
  
 }  
}

#### *newFixedThreadPool*

**package** com.rxl.concurrent.threadPool;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \* 创建一个定长的线程池  
 \*/***public class** NewFixedThreadPoolDemo {  
  
  
 **public static void** main(String[] args) {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(5);  
  
 **for** (**int** i = 1; i <= 100; i++) {  
 **final int** temp = i;  
 executorService.execute(() ->{  
 System.***out***.println(Thread.*currentThread*().getName() + **"我是傻逼"** + temp);  
 });  
 }  
 executorService.shutdown();  
 }  
}

#### *newScheduledThreadPool*

**package** com.rxl.concurrent.threadPool;  
  
**import** java.util.Timer;  
**import** java.util.TimerTask;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.ScheduledExecutorService;  
**import** java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 创建一个定长线程池，支持延迟及周期性执行任务  
 \*/***public class** NewScheduledThreadPoolDemo {  
  
  
 **public static void** main(String[] args) {  
  
 ScheduledExecutorService scheduledExecutorService = Executors.*newScheduledThreadPool*(5);  
  
 **for** (**int** i = 1; i < 6; i++) {  
 **final int** temp = i;  
 scheduledExecutorService.scheduleWithFixedDelay( () -> {  
 System.***out***.println(Thread.*currentThread*().getName() + **": 我是猪"** + temp);  
 }, 5, 2, TimeUnit.***SECONDS***);  
 }  
  
 **long** time = System.*currentTimeMillis*();  
 TimerTask task = **new** TimerTask() {  
 @Override  
 **public void** run() {  
 String str = String.*format*(**"%1$tM:%1$tS:%1$1tL"**, System.*currentTimeMillis*() - time);  
 System.***out***.println(str.substring(0, 7));  
 }  
 };  
 **new** Timer().schedule(task, 1, 100);  
 }  
  
}

**scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) ：**

创建并执行在给定的初始延迟之后，随后以给定的时间段首先启用的周期性动作; 那就是执行将在initialDelay之后开始，然后是initialDelay+period ，然后是initialDelay + 2 \* period ，等等。

**scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) ：**

创建并执行在给定的初始延迟之后首先启用的定期动作，随后在一个执行的终止和下一个执行的开始之间给定的延迟。

**上面两个方法是有区别的，scheduleAtFixedRate方法在执行任务时，不管上一次任务有没有被执行完成，都会间隔指定时间执行下一次任务，而scheduleWithFixedDelay方法则是需要要等待上一次任务执行完成，然后已上一次任务执行完成的时间为起点 + delay再执行下一次任务。**

#### *newSingleThreadPool*

**package** com.rxl.concurrent.threadPool;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \* 创建一个单化线程，可以将一堆任务按照顺序执行  
 \*/***public class** NewSingleThreadPoolDemo {  
  
 **public static void** main(String[] args) {  
 ExecutorService executorService = Executors.*newSingleThreadExecutor*();  
 **for** (**int** i = 1; i <= 100; i++) {  
 **final int** temp = i;  
 executorService.execute(() ->{  
 System.***out***.println(Thread.*currentThread*().getName() + **"我是傻逼"** + temp);  
 });  
 }  
 executorService.shutdown();  
 }  
}

### 线程池原理解析

**研究JDK线程池的创建过程，发现newCachedThreadPool、newFixedThreadPool、**

**newSingleThreadPool三个方法底层都是依赖ThreadPoolExecutor类创建的**

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

#### ThreadPoolExecutor构造器解析

**corePoolSize：**

**corePoolSize参数定义线程池的核心线程数。**

**maximumPoolSize**：

**指定了线程池中的最大线程数量。**

**keepAliveTime：**

**当线程池线程数量超过corePoolSize时，多余的空闲线程的存活时间，即，超过corePoolSize的空闲线程，在多长时间内，会被销毁。**

**workQueue：**

**存放未执行任务的队列。**

**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。**

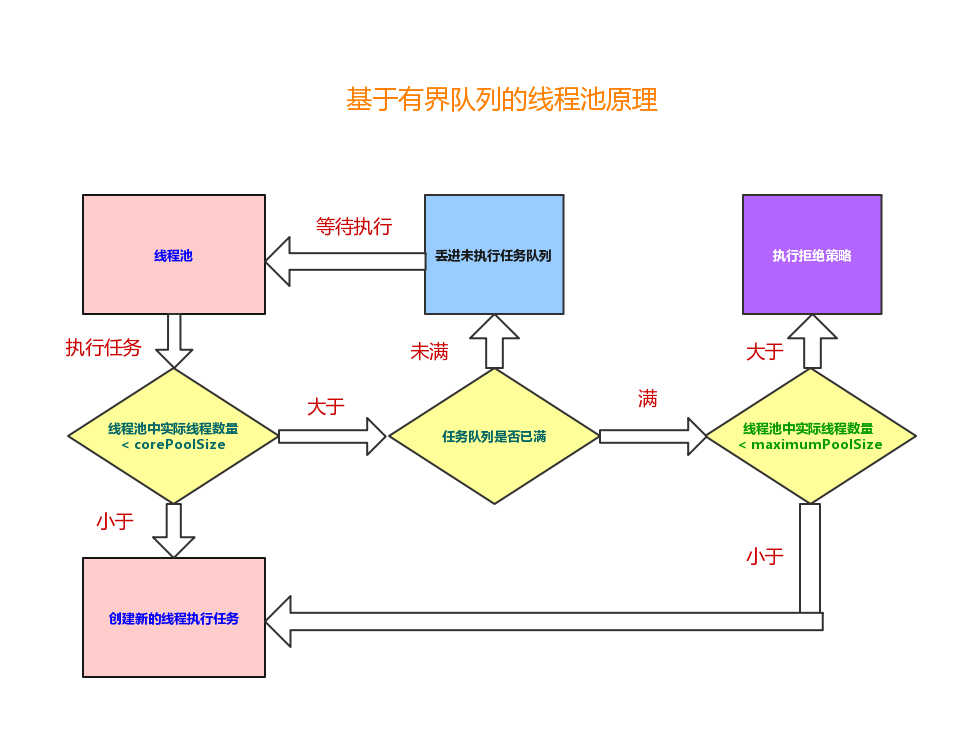
#### 线程池执行逻辑

JDK的线程池根据workQueue参数的特性，执行的逻辑也有细微的区别.

**有界队列：**

使用ArrayBlockingQueue或者带LinkedBlockingQueue可以创建有界队列

这种队列存放任务是有限制的。在这种情况下，线程执行逻辑如图;

 有界队列是有大小限制的，如果队列已满同时线程数量已达最大值，则执行拒绝测策略。

**无界队列：**

无界队列当线程池中的实际线程数量大于核心线程数量之后，会直接丢进任务队列直到内存占满。

**直接提交队列：**

由SynchronousQueue对象提供。这个队列没有容量，无法保存任务，每当任务执行时，都会创建新的线程来执行，所以线程数量很容易到达最大值，而执行拒绝策略。

#### 拒绝策略

ThreadPoolExecutor的最后一个参数指定了拒绝策略。也即是当任务数量超过系统实际承担能力时，会执行的拒绝策略。

### 自定义线程池