**首先检查jdk版本是否正确：**

·保证Project Structure=》Project里的language level=》8

·保证Project Structure=》Modules里的language level=》8

·保证Settings=》Java Compiler里的Target bytecode version=》8

**目录**

1. 什么是JUC
2. 进程和线程回顾

3、Lock锁

4、生产者和消费者

5、8锁的现象

1. 集合类不安全

7、Callable

8、CountDownLatch、CyclicBarrier、Semaphore

9、读写锁

10、阻塞队列

11、线程池

12、四大函数式接口

13、Stream流式计算

14、分支合并

15、异步回调

16、JMM

17、volatile

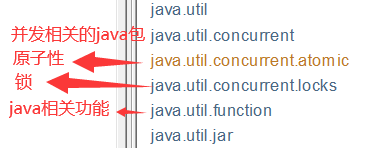
18、深入单例模式

19、深入理解CAS

20、原子引用

21、可重入锁、公平锁非公平锁、自旋锁

1. 什么是juc（学习方法：官方文档+源码）



java.util：java的工具包，包，分类

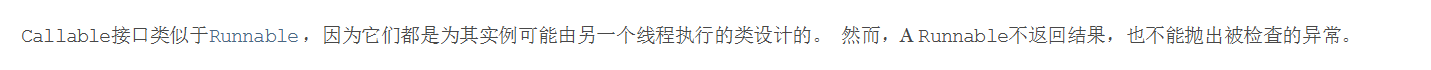
什么是juc：[java.util.concurrent](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/concurrent/package-frame.html" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/packageFrame)，java工具类下的一个并发功能的包。

Runnable 没有返回值、效率相比入 Callable相对较低!

·两者都属于juc里面



·但callable可以返回结果，也可以抛出异常

所以比起runable来说，callable有以下特点：  
1、有返回值  
2、可以抛出异常  
3、方法不同，run（），call（）

案例：<https://blog.csdn.net/qq_42388853/article/details/111769444>

1. **线程和进程**

·进程:一个程序，QQ.exe Music.exe程序的线程的集合;

·一个进程往往可以包含多个线程，至少包含一个!

·Java默认有几个线程?最少2个，mian、GC

·举例说明：开了一个进程Typora（开启进程），写字（一个线程），自动保存（又一个线程线程)

·对于Java而言调用多线程的方法:Thread、Runnable（静态代理）、Callable（静态代理）

#java真的可以开启线程嘛？(面试题)

不行，因为开启线程需要调用start0（），这是个本地方法，使用的是底层c++。

**并发、并行：**

**并发(多线程操作同一个资源)**

·CPU一核，模拟出来多条线程，天下武功，唯快不破，快速交替

**并行(多个人一起行走)**

·CPU多核，多个线程可以同时执行;线程池

public class Test1 {  
 public static void main(String[] args) {  
 //获取cpu核数  
 System.out.println(Runtime.getRuntime().availableProcessors());  
 }  
}

#并发编程的本质:充分利用CPU的资源

线程有几个状态：java有6个状态，操作系统有5个

public enum State {  
 //新生  
 NEW,  
 //运行

RUNNABLE,  
 //阻塞  
 BLOCKED,  
 //无限期等待  
 WAITING,  
 //限时等待  
 TIMED\_WAITING,  
 //结束  
 TERMINATED;  
}

操作系统的五个状态：初始状态（NEW) ，可运行状态（READY），运行状态（RUNNING) ，等待状态（WAITING) ，终止状态（TERMINATED）。

案例：<https://blog.csdn.net/qq_42388853/article/details/111597105>

**wait/sleep区别：**

1. 来自不同的类

wait=》object

sleep=》Thread

常用的休眠方法：使用java.util.concurrent里的TimeUnit

TimeUnit.SECONDS.sleep(1);  
TimeUnit.DAYS.sleep(2);

2、关于锁的释放

wait 会释放锁，sleep睡觉了，抱着锁睡觉，不会释放!

3、使用的范围不同

wait：必须在同步代码块。

sleep：可在任何地方睡。

#即有synchronized修饰符修饰的语句块，被该关键词修饰的语句块，将加上内置锁。实现同步。  
例：synchronized(Object o ){}

4、是否需要捕获异常，两者都需要捕获异常

wait：需要捕获异常（中断异常）

sleep：必须要捕获异常

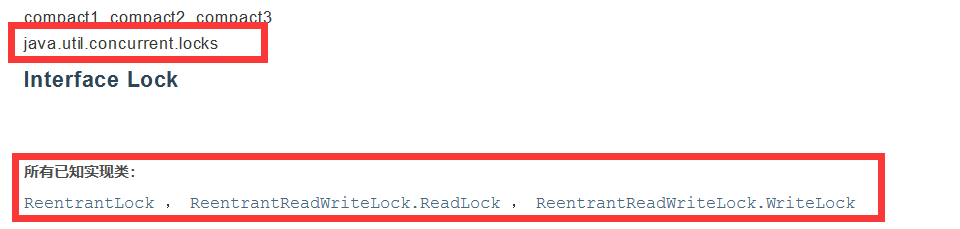
**Lock锁（重点）：**

1. 传统的Synchronized

使用过程：<https://blog.csdn.net/qq_42388853/article/details/111599262>

1. Lock锁

·属于JUC包下，是一个接口，实现类有读写锁和可重入锁（reentrantlock）



·观看文档的使用方法



使用过程：<https://blog.csdn.net/qq_42388853/article/details/111599754>

Synchronized和Lock区别：

1、Synchronized 内置的Java关键字，Lock是一个Java类

2、Synchronized无法判断获取锁的状态，Lock 可以判断是否获取到了锁

3、Synchronized 会自动释放锁，lock 必须要手动释放锁!如果不释放锁，死锁

4、Synchronized线程1(获得锁，阻塞)、线程2(等待，傻傻的等） ; Lock锁就不一定会等待下去;

5、Synchronized 可重入锁，不可以中断的，非公平;Lock，可重入锁，可判断锁，默认非公平(可以自己设置）;

6、Synchronized适合锁少量的代码同步问题，Lock适合锁大量的同步代码!

什么是CAS、非公平锁和公平锁：

<https://blog.csdn.net/qq_42388853/article/details/111663629>

**锁是什么，如何判断锁的是谁？**

8锁案例：<https://blog.csdn.net/qq_42388853/article/details/111701790>

生产者和消费者问题（使用synchronized和Lock进行尝试）

线程之间的通信问题:生产者和消费者问题!等待唤醒，通知唤醒线程交替执行A B操作同一个变量num = 0

A num+1

B num-1

·使用synchronized演示生产者和消费者问题

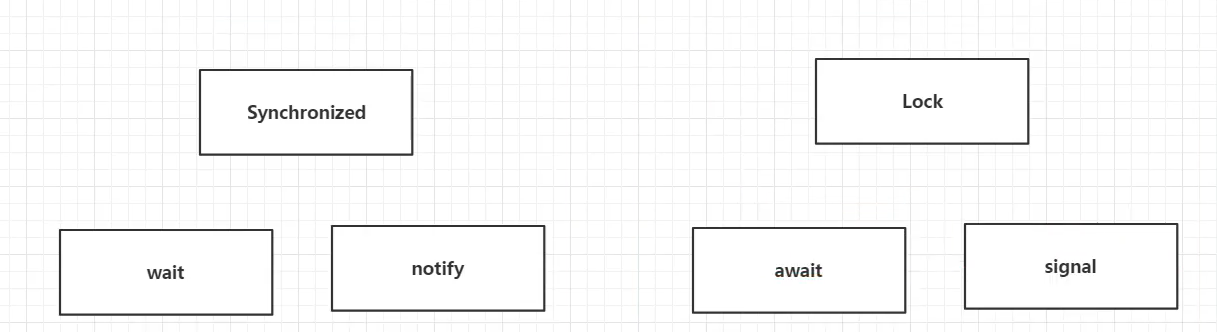
案例：<https://blog.csdn.net/qq_42388853/article/details/111679873>



#虚假唤醒问题：while能解决虚假唤醒的原因是因为，拿消费者举例，假如存在虚假唤醒，那么此时的num==0，就进入等待，一直等到num不是虚假唤醒而是真实唤醒变成了1。（解决方法，用while）

·JUC版本的生产者和消费者

二者的区别：



案例：<https://blog.csdn.net/qq_42388853/article/details/111683589>

任何一个新的技术，绝对不是仅仅只是覆盖了原来的技术，优势和补充!接下去看一下新功能。

Condition（同步监视器）精准的通知和唤醒线程：顺序执行。

案例：<https://blog.csdn.net/qq_42388853/article/details/111686101>

**8锁现象**

---如何判断锁的是谁!永远的知道什么锁，锁到底锁的是谁!深刻理解我们的锁。

8锁案例：<https://blog.csdn.net/qq_42388853/article/details/111701790>

总结：  
普通带锁方法：锁对象，同一对象下的才按顺序执行，如果是同一个类下的不同对象则不受影响。

普通不带锁方法：不受任何影响。

静态带锁方法：锁类，同一个类下的所有对象的所有带锁方法都得按顺序执行。

**集合安全类：COW**

# CopyOnWriteArrayList源码解析：<https://blog.csdn.net/qq_42388853/article/details/111738495>

CopyOnWriteArraySet的构造方法源码上是这样写的，所以直接参考COW即可：

public CopyOnWriteArraySet() {  
 al = new CopyOnWriteArrayList<E>();  
}

hashSet 底层是什么?是HashMap

public HashSet() {  
 map = new HashMap<>();  
}

他的add方法：其中PRESENT是一个常量

public boolean add(E e) {  
 return map.put(e, PRESENT)==null;  
}

**MAP安全类：ConcurrentHashMap**

回顾map：

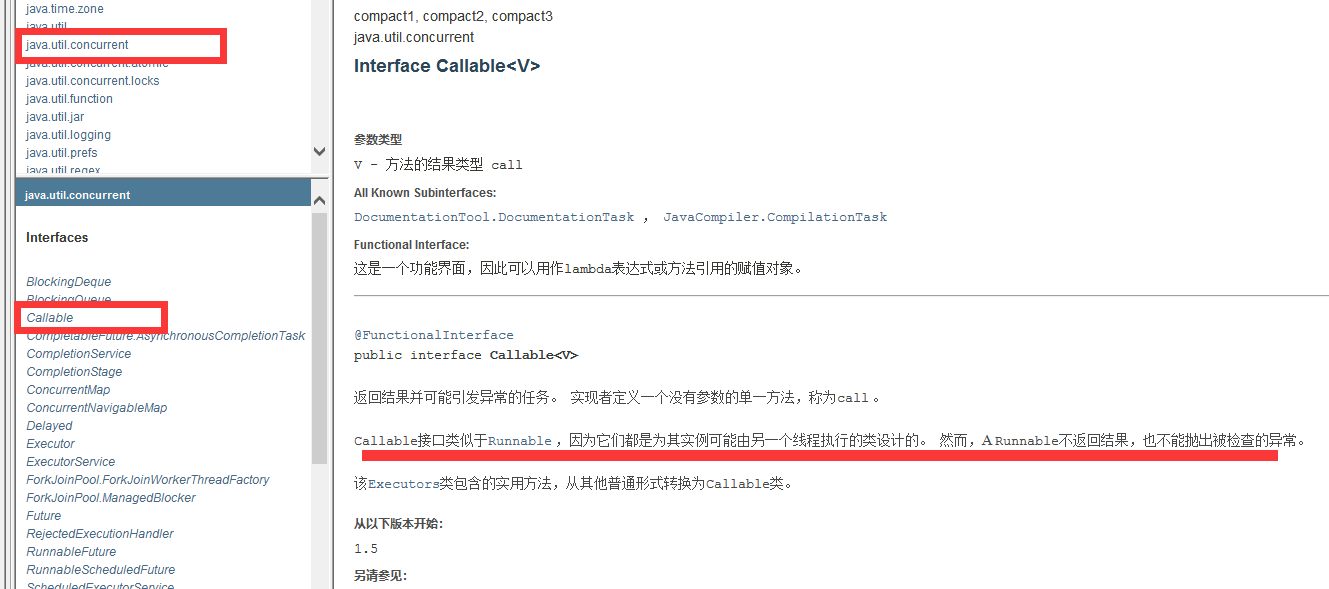
HashMap<String, String> map = new HashMap<>();  
//等价于 HashMap<String, String> map1 = new HashMap<>(16, 0.75f);  
//0.75f是加载因子，16是初始化容量，最大容量是 1 << 30，即2^30

ConcurrentHashMap的案例：<https://blog.csdn.net/qq_42388853/article/details/111768427>

作业：研究ConcurrentHashMap的原理。

**Callable**

在jdk的api解释里：

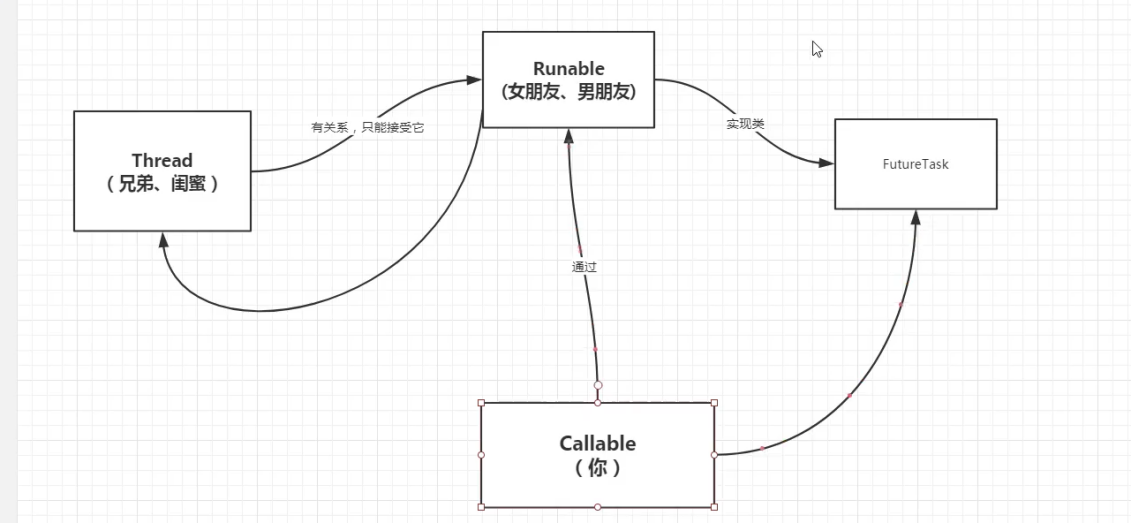


1、有返回值

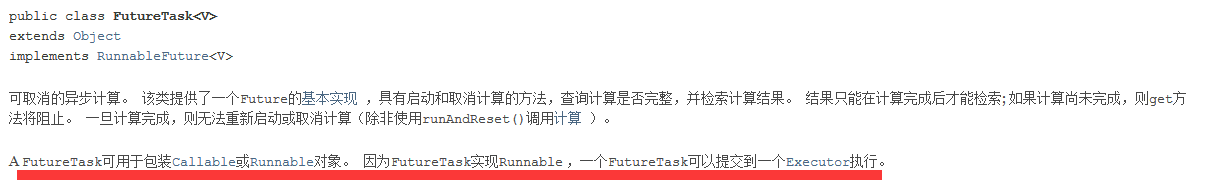
2、可以抛出异常

3、方法不同，run（），call（）

callable并不能像runable一样直接执行：需要借助FutureTask来执行



FutureTask:可以包装callable或runable



细节:

1、有缓存

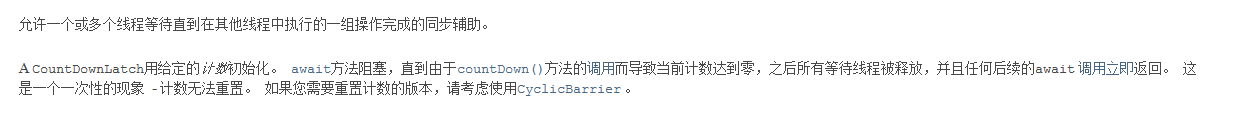
2、结果可能需要等待，会阻塞!

使用案例：<https://blog.csdn.net/qq_42388853/article/details/111769444>

**三大常用辅助类：**

1. **CountDownLatch（计数器）**

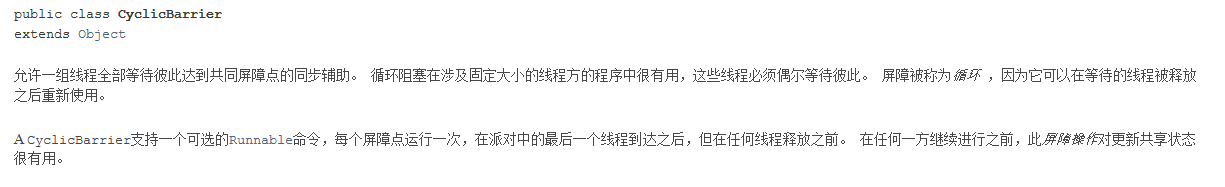
·api里的解释：就是一个计数器，辅助全部线程按顺序执行。



原理:  
countDownLatch.countDown（）;数量-1  
countDownLatch.await（）；//等待计数器归零，然后再向下执行

使用案例：<https://blog.csdn.net/qq_42388853/article/details/111769840>

1. **CyclicBarrier（栅栏）**

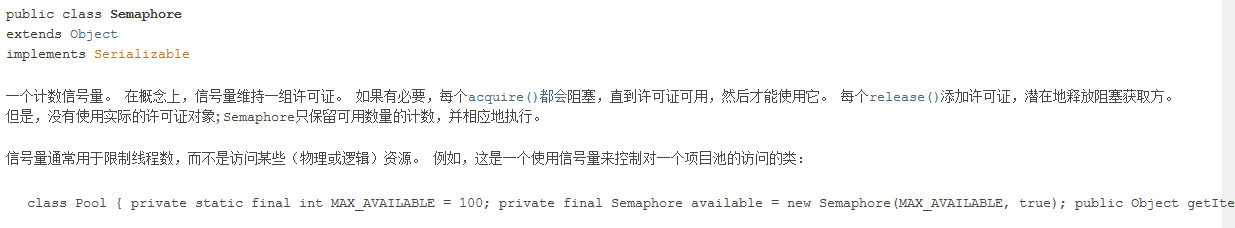
·api解释：类似CountDownLatch，但是是一个加法计数器。从0开始计数

·具体方法：cyclicBarrier.await();等待到CyclicBarrier内规定数量，才会执行await内的语句。

案例：<https://blog.csdn.net/qq_42388853/article/details/111855974>

1. **Semaphore**(信号量)

·api解释限流作用的一个工具类，规定了多少数量，同一时间就只能走多少数量，直到有释放出来。



案例：<https://blog.csdn.net/qq_42388853/article/details/111862992>

**读写锁：ReadWriteLock**



独占锁（写锁)：—次只能被一个线程占有

共享锁（读锁)：多个线程可以同时占有

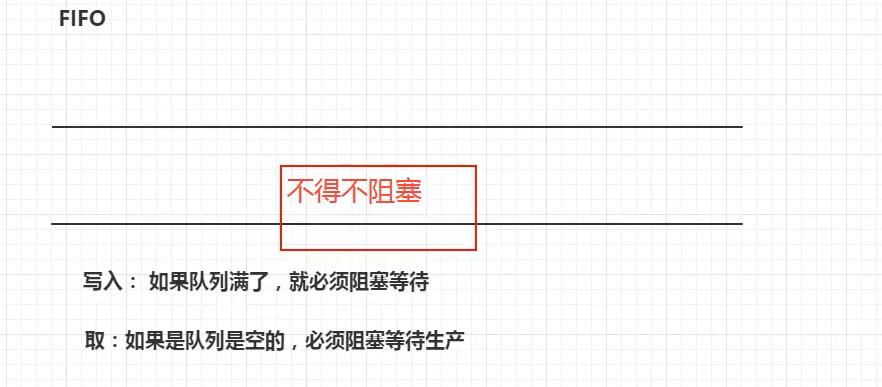
ReadwriteLock

读-读可以共存!

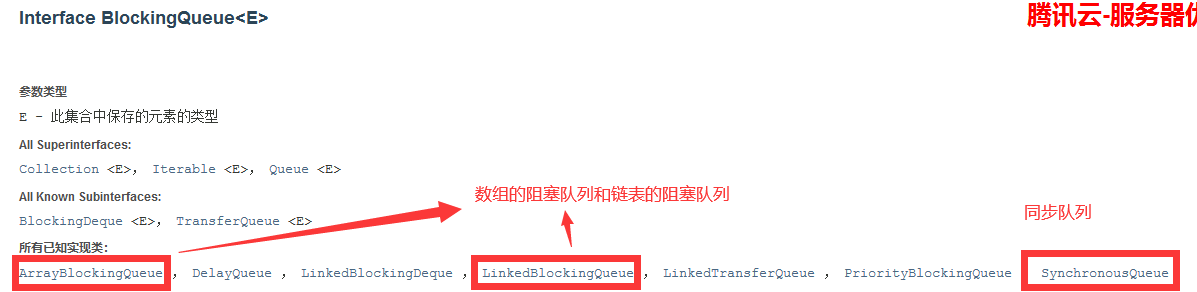
读-写不能共存!

写-写不能共存!

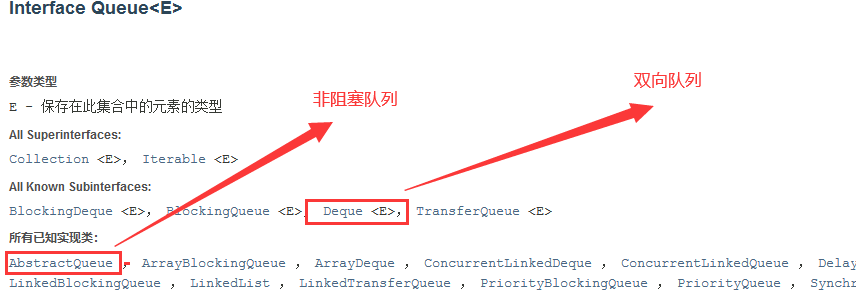
使用案例：<https://blog.csdn.net/qq_42388853/article/details/111863795>

阻塞队列：

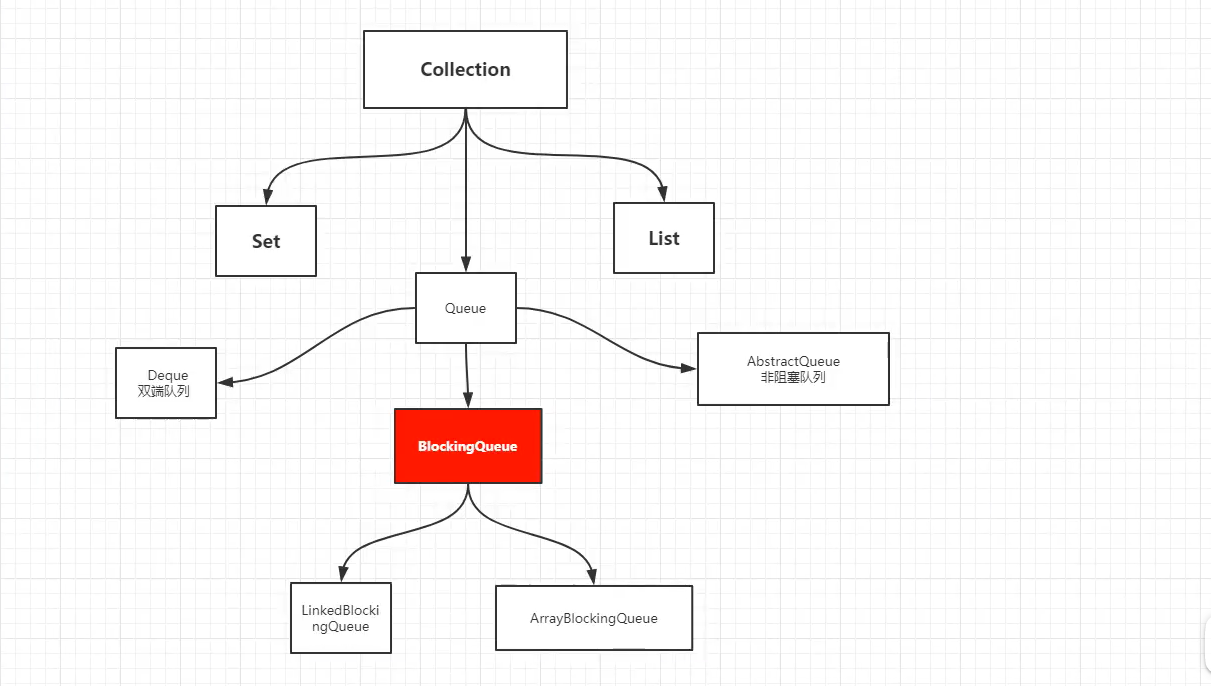
队列的分级：



## Queue的分类：



加入队列后的示意图：

 什么情况下我们会使用阻塞队列:多线程并发处理，线程池!

学会使用队列添加、移除四组API：



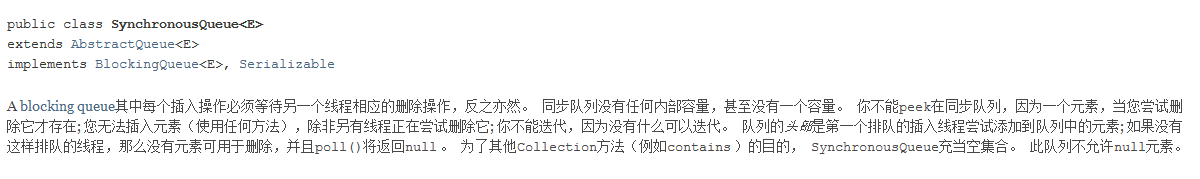
使用地址:https://blog.csdn.net/qq\_42388853/article/details/111875653

**SynchronousQueue同步队列**

·没有容量，进去一个元素，必须等待取出来之后，才能再往里面放一个元素!

·具体方法：put,take

api中的描述：



使用案例：<https://blog.csdn.net/qq_42388853/article/details/111917573>

**线程池：三大方法、7大参数、4种拒绝策略**

程序的运行，本质:占用系统的资源! 优化资源的使用!=>池化技术

线程池、连接池、内存池、对象池/l.....创建、销毁。十分浪费资源

·池化技术:事先准备好一些资源，有人要用，就来我这里拿，用完之后还给我。

·线程池的好处:

1、降低资源的消耗

2、提高响应的速度

1. 方便管理。

·线程复用、可以控制最大并发数、管理线程

线程池：三大方法

1、executors

// ExecutorService service = Executors.newSingleThreadExecutor();//单个线层  
// ExecutorService service = Executors.newFixedThreadPool(5); //五个线程  
 ExecutorService service = Executors.newCachedThreadPool();//弹性线程，会自己去判断创造几个

// 使用方法

service.execute(()->{  
 System.out.println(Thread.currentThread().getName());  
});

//关闭方法

service.shutdown();

三大方法的底层源码：观察后可以发现，这三个底层源码都是传入了参数，例如单线程的就是（1，1）多线程的则传入对应线程数，变化线程的就传入（0，最大数）。

public static ExecutorService newSingleThreadExecutor() {  
 return new FinalizableDelegatedExecutorService  
 (new ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>()));  
}

public static ExecutorService newFixedThreadPool(int nThreads) {  
 return new ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>());  
}

public static ExecutorService newCachedThreadPool() {  
 return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,  
 60L, TimeUnit.SECONDS,  
 new SynchronousQueue<Runnable>());  
}

进入最底层原码发现：

public ThreadPoolExecutor(int corePoolSize, //线程池初始大小，最少开几个线程  
 int maximumPoolSize, //最大线程池大小 ，最多开多少  
 long keepAliveTime,//最大线程池空闲时间，空闲太久级把大于 初始大小的线程释放  
 TimeUnit unit, //超时的单位  
 BlockingQueue<Runnable> workQueue) { //阻塞队列  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, Executors.defaultThreadFactory(), //线程工厂，默认不动

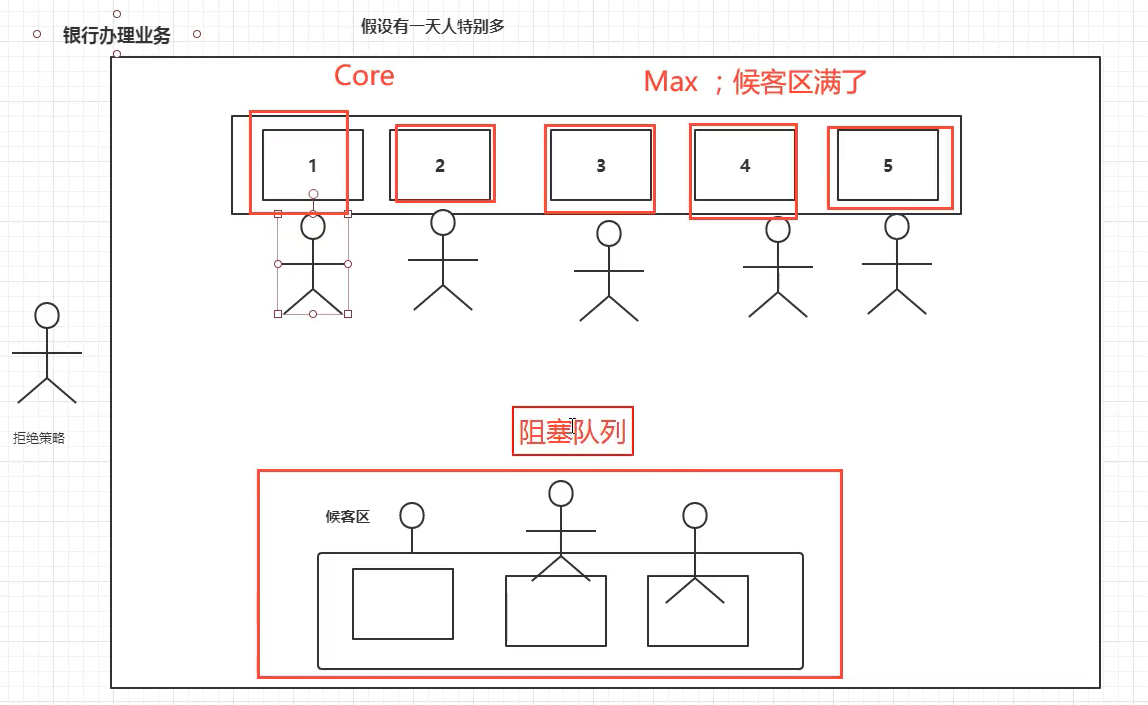
defaultHandler); //拒绝策略  
  
}

几个后拒绝是看LinkedBlockingDeque+max

四个拒绝策略：

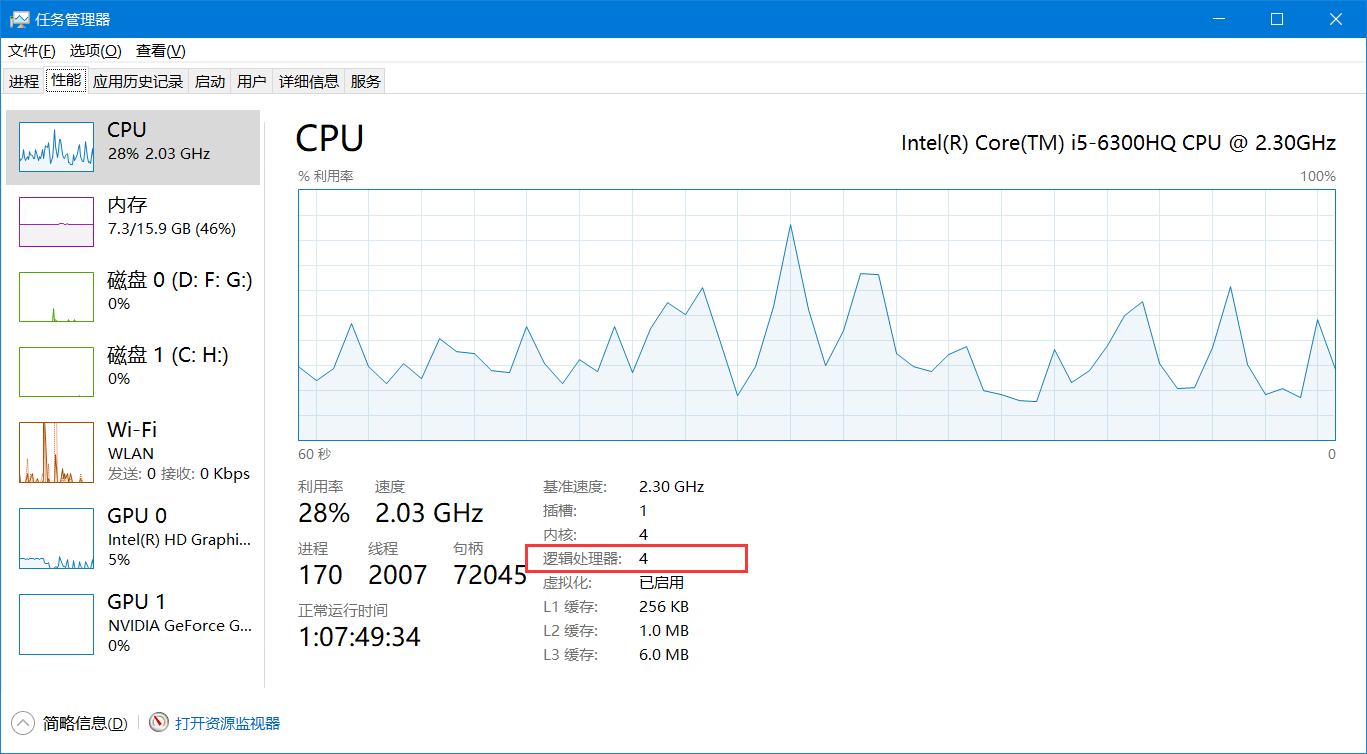
// new ThreadPoolExecutor.AbortPolicy() //超出后抛出异常  
// new ThreadPoolExecutor.CallerRunsPolicy()//超出后，超出的线程原本是谁的，谁就去执行  
// new ThreadPoolExecutor.DiscardOldestPolicy()//新方法，会去尝试获取线程，获取不到则直接拒绝不返回错误  
// new ThreadPoolExecutor.DiscardPolicy() //直接拒绝不返回错误

7大参数示意图：一开始仅开放核心线程1-2区域，等候客区（阻塞队列满了）开放3-5最大线程。此后，如果其他的还满了，还有客人进来，则触发拒绝策略。而超时（keepAliveTime)表示的是当max区空闲了那么多时间时，则再次关闭。



最大线程到底该如何定义：分为CPU密集型和IO密集型

1. CPU密集型：几核，就是几，可以保持CPU的效率最高!



Runtime.getRuntime().availableProcessors() //获取最大cpu核数

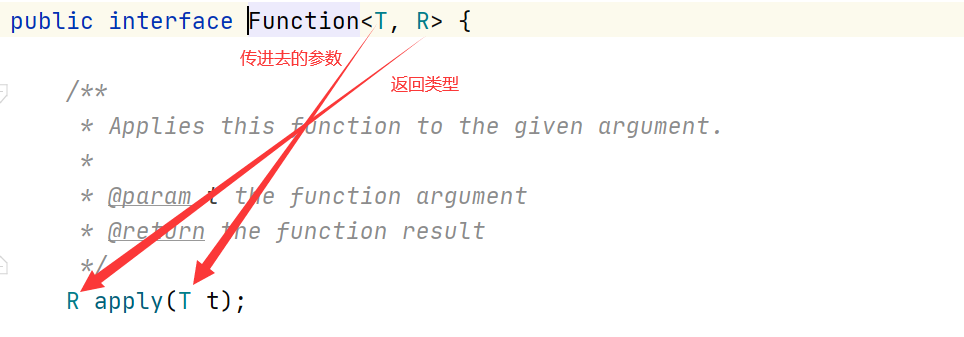
1. IO密集型：判断哪些程序十分占用IO线程，则设置大于她们的最大线程，一般2倍。

**四大函数式接口（必须掌握）**

新时代的程序员:lambda表达式、链式编程、函数式接口、Stream流式计算。

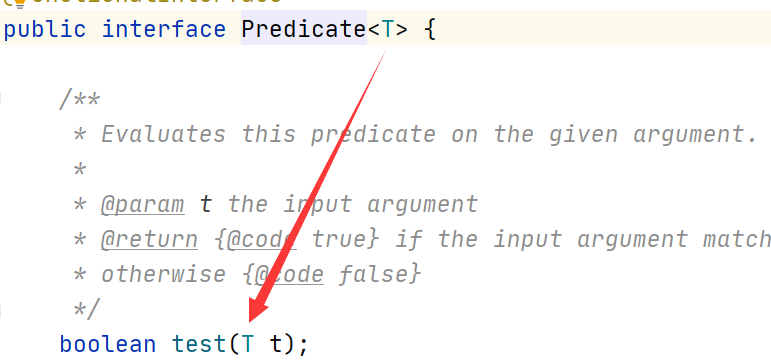
四大原生的函数式接口：[Consumer](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Consumer.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame)（消费型接口） 、[Function](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Function.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame)（函数式接口） 、[Predicate](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Predicate.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame) （判断型接口）、[Supplier](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Supplier.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame)（供给型接口） 。

查看[Function](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Function.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame) 的源码使用：传入进行一系列操作，返回一个类型的值



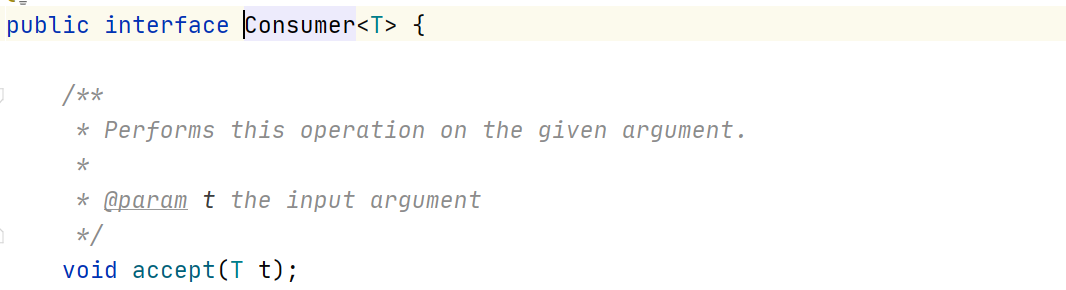
使用方法，调用function里的apply方法，传入参数。

查看[Predicate](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Predicate.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame) 的源码使用：传入进行判断，然后返回boolean值。



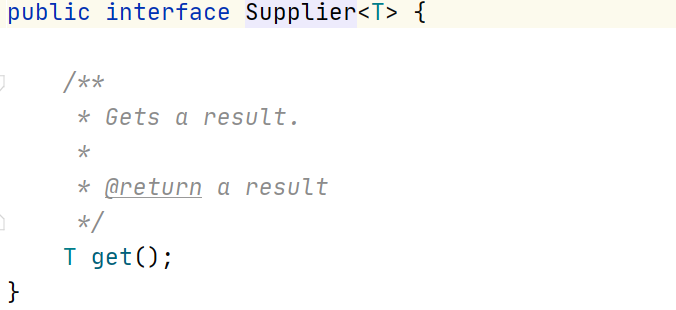
传入泛型即可。

[Consumer](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Consumer.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame) 的使用：



传入参数无返回值

[Supplier](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/Supplier.html" \o "interface in java.util.function" \t "mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/function/classFrame)供给型接口的使用：



**Stream流式计算：**

大数据：存储+计算

集合、MySQL本质就是存储东西的；计算都应该交给流！

常见使用方法：map（对对象进行操作）filter（对对象进行过滤）sorted（排序）limit（限制输出数量）

题目要求:一分钟内完成此题，只能用一行代码实现!

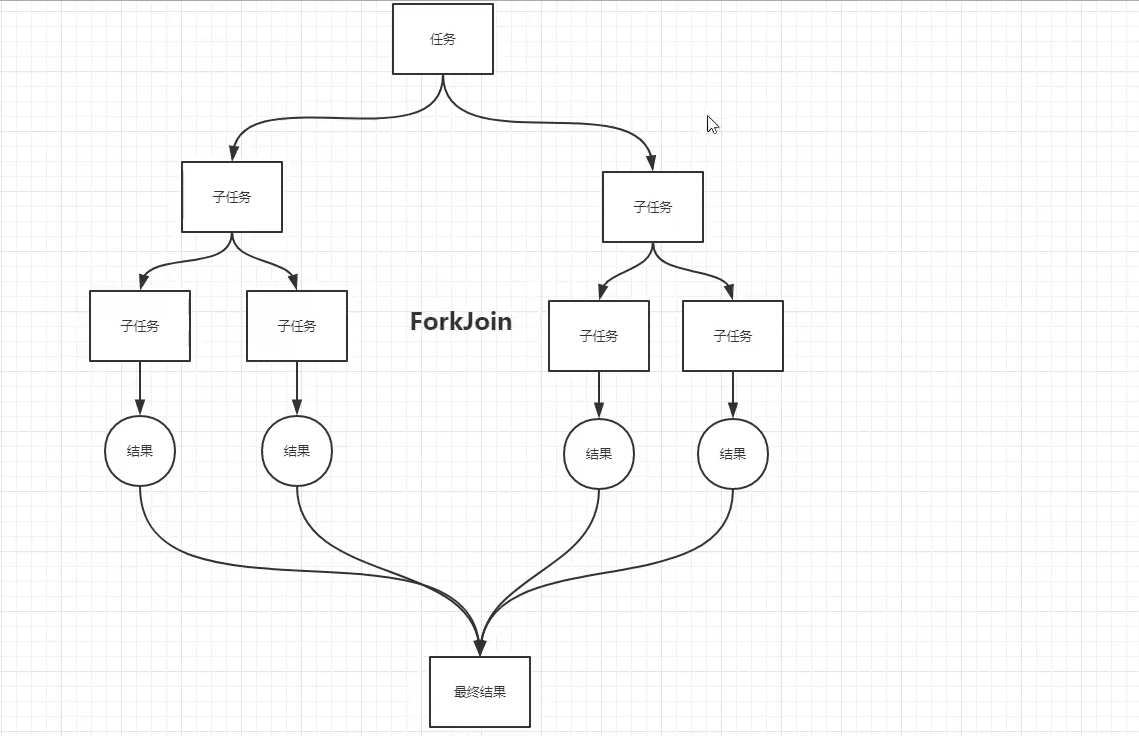
现在有5个用户!筛选：

1. ID 必须是偶数
2. 龄必须大于23岁
3. 用户名转为大写字母
4. 用户名字母倒着排序
5. 只输出一个用户!

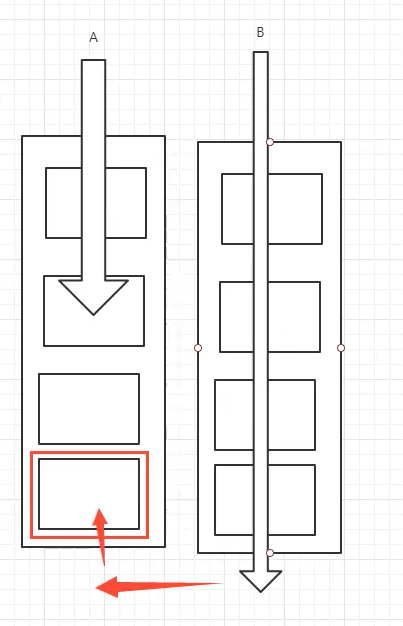
public class StreamDemo {  
 public static void main(String[] args) {  
// 常见使用方法：  
// 题目要求:一分钟内完成此题，只能用一行代码实现!  
// 现在有5个用户!筛选：  
// 1、ID 必须是偶数  
// 2、龄必须大于23岁  
// 3、用户名转为大写字母  
// 4、用户名字母倒着排序  
// 5、只输出一个用户!  
 User u1 = new User( 1, "a" ,21);  
 User u2 = new User( 2, "b", 22);  
 User u3 = new User( 3, "c", 23);  
 User u4 = new User( 4, "d", 24);  
 User u5 = new User( 6, "e", 25);  
 List<User> list = Arrays.asList(u1, u2, u3, u4, u5);  
  
 list.stream()  
 .filter((user)->{return user.getId()%2==0;})  
 .filter((user)->{return user.getAge()>23;})  
 .map((user)->{return user.getName().toUpperCase();})  
 .sorted((user1,user2)->{return user2.compareTo(user1);})  
 .limit(1)  
 .forEach(System.out::println);  
  
 }  
}

**ForJoin：**

ForkJoin在JDK 1.7 ，并行执行任务!提高效率。大数据量!大数据:Map Reduce（把大任务拆分为小任务)



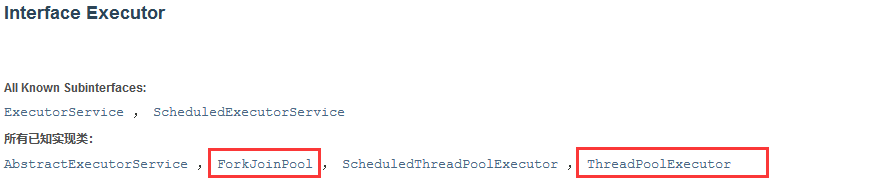
ForJoin特点：工作窃取 。



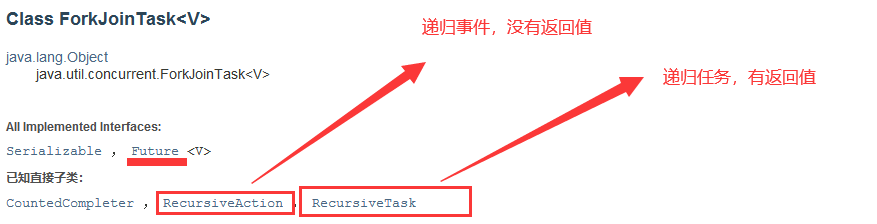
工作窃取：维护了一个双端队列。

A,B线程同时进行工作，B线程执行任务完毕后，会去执行A的任务。

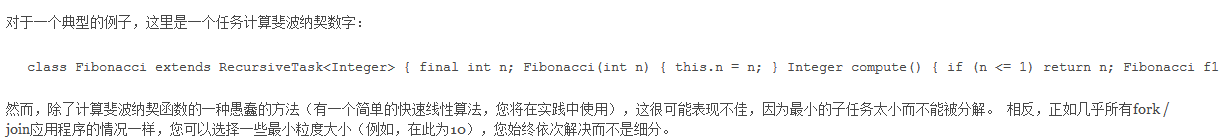
ForkJoinPool和[ThreadPoolExecutor](mk:@MSITStore:D:\\java项目\\API\\jdk%20api%201.8_google.CHM::/java/util/concurrent/../../../java/util/concurrent/ThreadPoolExecutor.html" \o "java.util.concurrent中的类)同属于Executor：



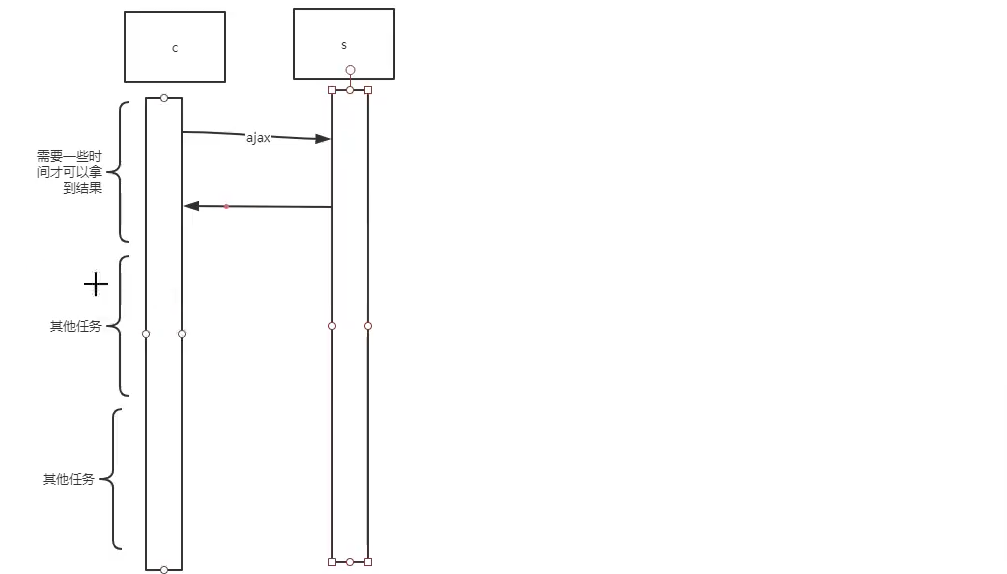
并且ForkJoinTask是实现了future接口的：



使用方法：创造一个类继承它



**异步回调：类似于ajax**



**异步回调：Future**

Future设计的初衷:对将来的某个事件的结果进行建模

**Volatile是Java虚拟机提供轻量级的同步机制**

1、保证可见性

2、不保证原子性

3、禁止指令重排

**什么是JMM：**

JMM : Java内存模型，不存在的东西，概念!约定!

关于JMM的一些同步的约定︰

1、线程解锁前，必须把工作内存立刻刷回主存。

2、线程加锁前，必须读取主存中的最新值到工作内存中!

3、加锁和解锁是同一把锁！

#每个线程都有自己的工作内存，主存只有一个。

**8种操作：**

过程解析：1、线程会去主存里读取信息然后加载到工作内存，然后加锁

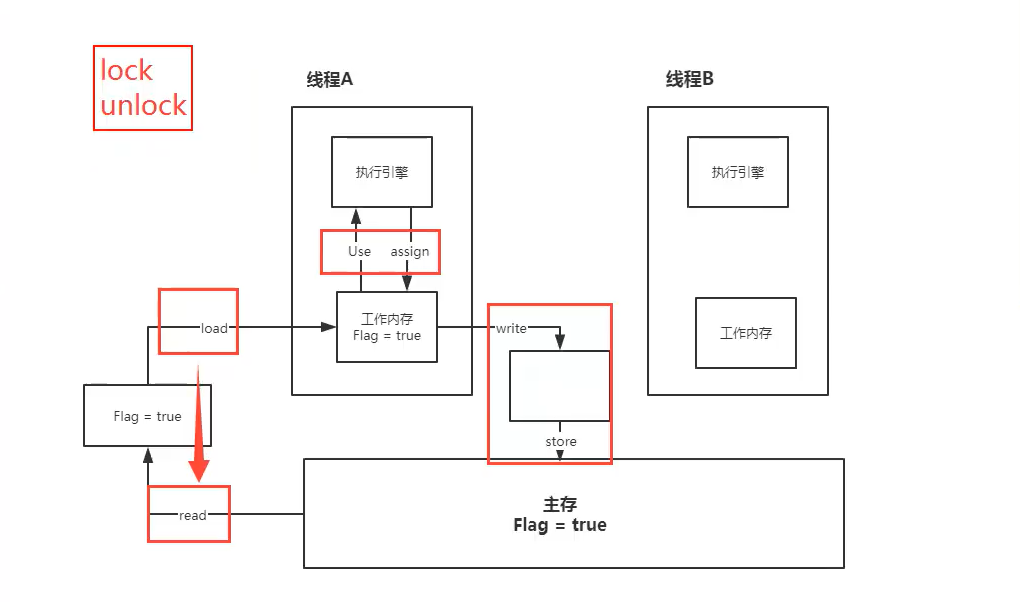
（read和load，带上lock）

2、线程的执行引擎会使用工作内存里的信息，更新完毕后**赋值**给工作内

（use和assign）

3、当线程执行完毕后会将工作内存的信息写入并且存储在主存，然后解锁

（write和store，带上unlock）



## **内存交互操作**

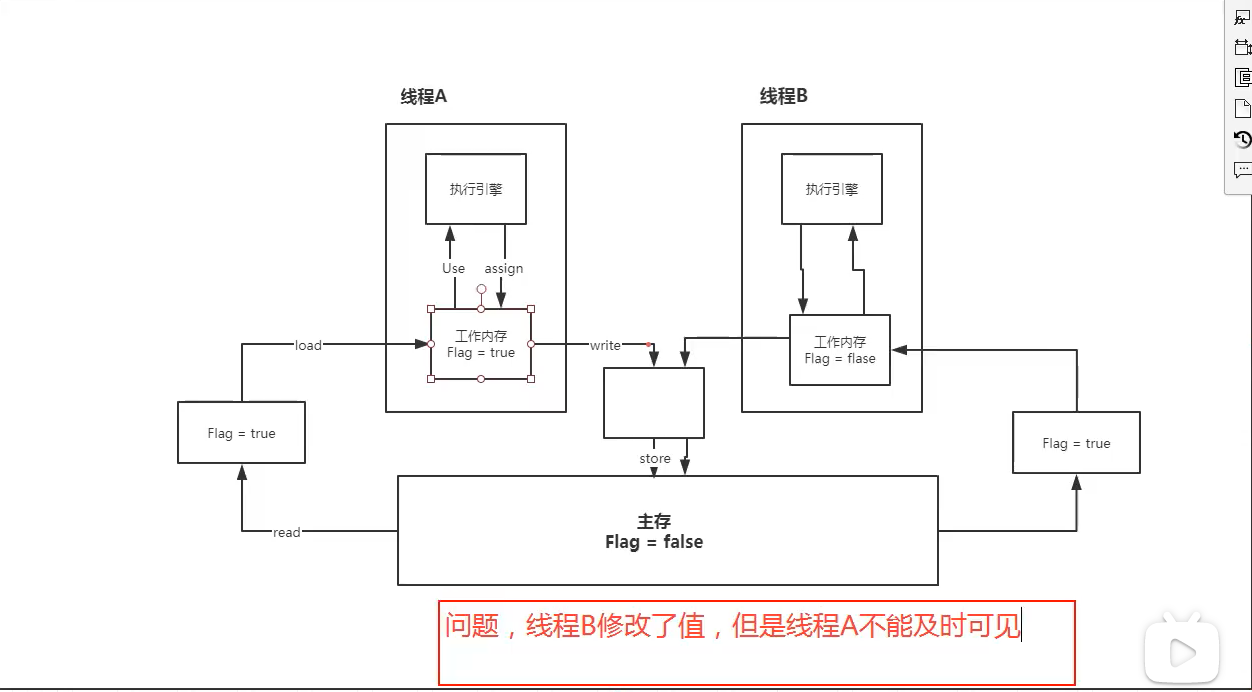
 　　内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

* + lock     （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
  + unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
  + read    （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用
  + load     （载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
  + use      （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
  + assign  （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
  + store    （存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
  + write 　（写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中

　　JMM对这八种指令的使用，制定了如下规则：

* + 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write(**不许单一出现**)
  + 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
  + 不允许一个线程将没有assign的数据从工作内存同步回主内存
  + 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼变量实施use、store操作之前，必须经过assign和load操作
  + 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
  + 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值
  + 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
  + 对一个变量进行unlock操作之前，必须把此变量同步回主内存

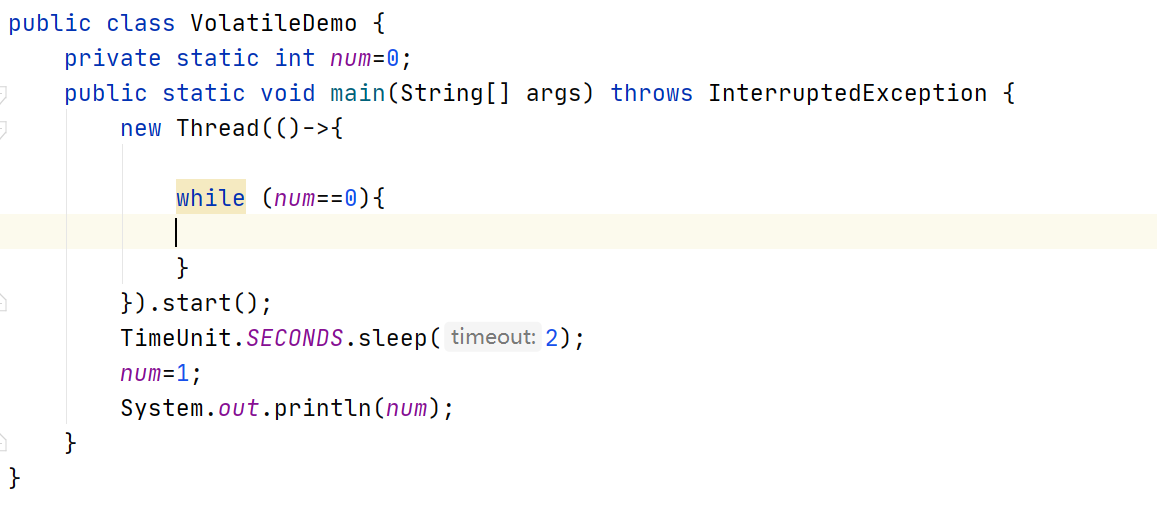
但这样存在一定的问题:当线程AB同时读取了一个数据，线程A还在处理的时候，线程B就先写入改变了主内存中的数据的时候，此时A是看不到数据发生改变的，会有覆盖的风险，所以此时要引入volatile。



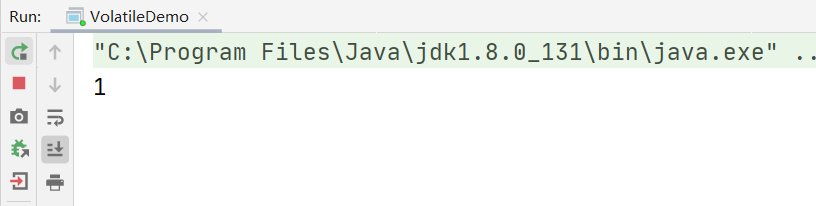
**Volatile：保证可见性，不保证原子性，禁止指令重排**

**1、可见性：**

举例上面的情况：

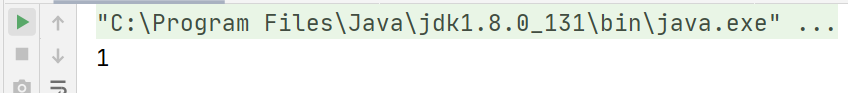


此时B线程陷入了死循环：



解决办法：在变量中加入volatile



当主线程值被修改后，其他线程也因为num不为0变回了原本的值。

**2、不保证原子性：**

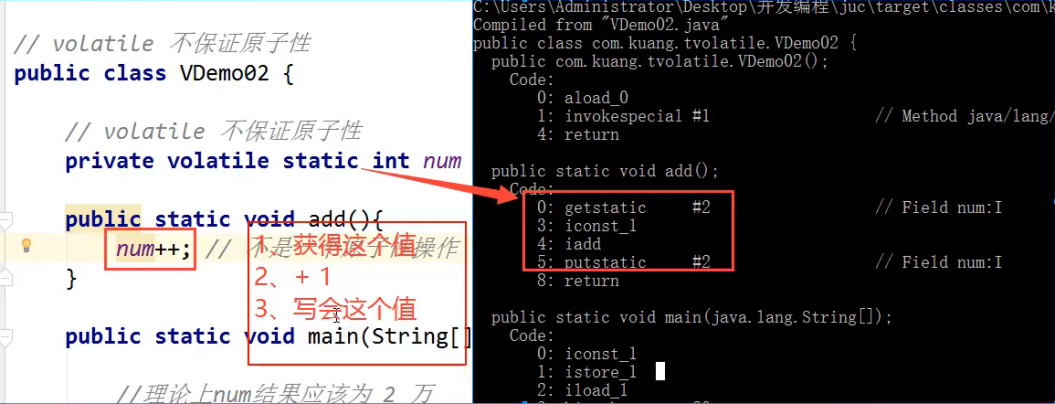
package JMM;  
public class VDemo2 {  
 private volatile static int num=0;  
   
 public static void add(){  
 num++;  
 }  
 public static void main(String[] args) {  
 for (int i = 0; i <20 ; i++) {  
 new Thread(()->{  
 for (int j = 0; j <1000 ; j++) {  
 add();  
 }  
 }).start();  
 }  
 while(Thread.activeCount()>2)  
 {  
 Thread.yield();  
 //让主线程变成就绪状态，直到线程只有gc线程和主线程才继续执行下面操作  
 }  
 System.out.println(num);  
 }  
}

输出结果是不确定的，因为不保证原子性，大致在19679左右。

解决办法：

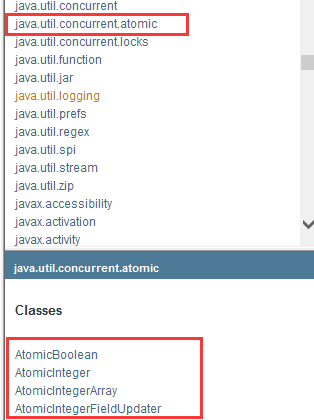
1. 加锁，不论是synchronized还是lock都可以（可行）
2. 使用JUC下的原子类

反编译查看一下：使用javap -c VDemo2.class



从虚拟机上来看，add方法执行了获取值，+1，写入这个值三步操作。

**juc原子类：**



使用原子类进行操作：

import java.util.concurrent.atomic.AtomicInteger;  
  
public class VDemo2 {  
// private volatile static int num=0;  
 private volatile static AtomicInteger num =new AtomicInteger();  
 public static void add(){  
 num.getAndIncrement();  
 }  
 public static void main(String[] args) {  
 for (int i = 0; i <20 ; i++) {  
 new Thread(()->{  
 for (int j = 0; j <1000 ; j++) {  
 add();  
 }  
 }).start();  
 }  
 while(Thread.activeCount()>2)  
 {  
 Thread.yield();  
 //让主线程变成就绪状态，直到线程只有gc线程和主线程才继续执行下面操作  
 }  
 System.out.println(num);  
 }  
}

#发现答案正确，说明加锁和原子类操作都可行。

这些类的底层都直接和操作系统挂钩!在内存中修改值 ! Unsafe类是一个很特殊的存在!

**3、禁止指令重排：**

什么是指令重排:你写的程序，计算机并不是按照你写的那样去执行的。

源代码-->编译器优化的重排-->指令并行也可能会重排-->内存系统也会重排--->执行

**处理器在进行指令重排的时候，考虑:数据之间的依赖性!**



指令重排可能会出现的问题：



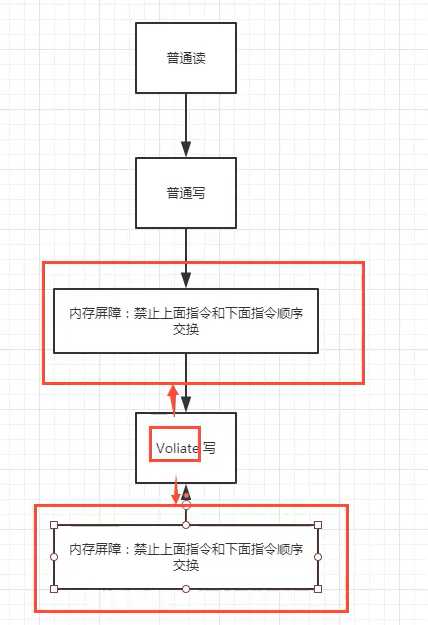
volatile可以避免指令重排:操作系统内存在内存屏障。

作用:

1、保证特定的操作的执行顺序!

2、可以保证某些变量的内存可见性（利用这些特性volatile实现了可见性)

示意图：



Volatile是可以保持可见性。不能保证原子性，由于内存屏障，可以保证避免指令重排的现象产生!

**彻底玩转单例模式**

饿汉式，DCL懒汉式，深究!

饿汉式：饿汉式，一开始就把对象创建好，然后返回对象给调用方法的人，缺点浪费空间，并且多线程下会存在问题。

public class Hungry {  
 public Hungry() {  
 }  
 private final static Hungry HUNGRY =new Hungry();  
  
 public static Hungry getInstance(){  
 return HUNGRY;  
 }  
}

DCL懒汉式：懒汉式加锁两次，并且给参数加上volatile防止重排

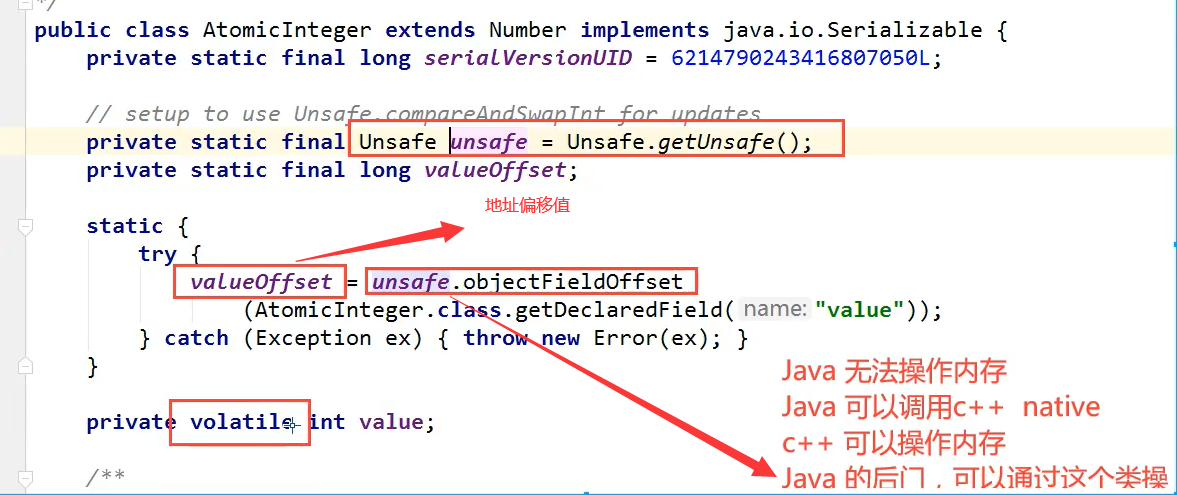
破解单例模式和使用枚举解决反射破解单例模式问题：

<https://blog.csdn.net/qq_42388853/article/details/112063892>

**深入理解CAS：**

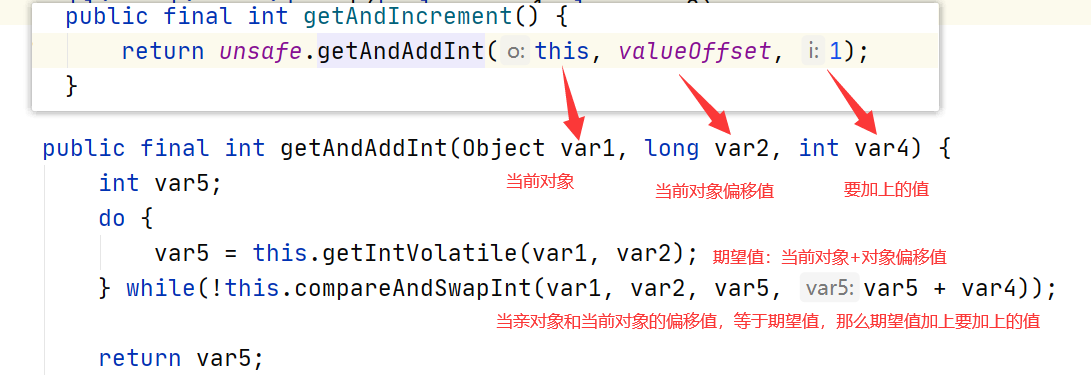
大厂你必须要深入研究底层!有所突破!**修内功，操作系统，计算机网络原理**

**unsafe类：java通过这个类去操作c++方法，从而操作内存**



什么是CAS？

深入源码理解：这里用到了自旋锁



CAS∶比较当前工作内存中的值和主内存中的值，如果这个值是期望的，那么则执行操作!如果不是就一直循环!

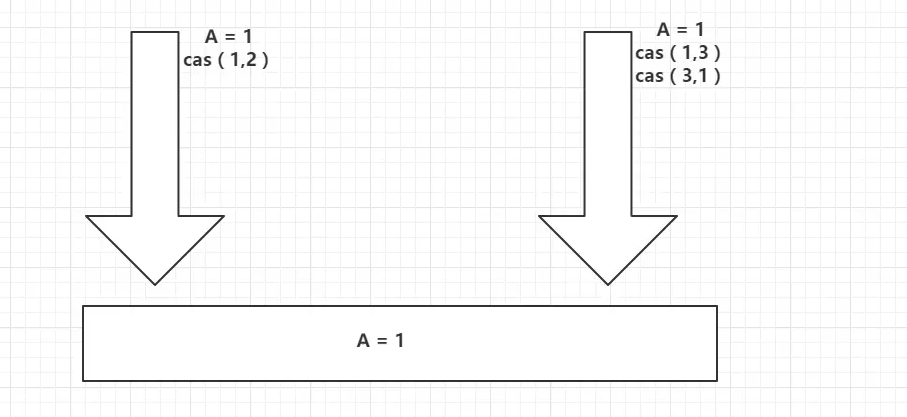
缺点∶

1、循环会耗时

2、一次性只能保证一个共享变量的原子性

3、ABA问题

CAS问题：ABA问题（狸猫换太子）：



线程B改变了线程1为3，然后再改回1，线程A却无法知道，仍旧是把1改成了2。

A线程做的操作，我们并不知道。

**原子引用:解决ABA问题**

类似于乐观锁：加入了版本号机制

ABA问题详解包括解决办法：

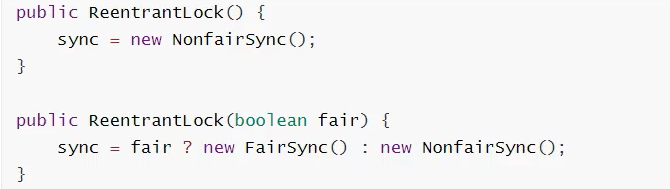
<https://blog.csdn.net/qq_42388853/article/details/112070834>

**各种锁的理解：**

1. 公平锁和非公平锁：

公平锁:非常公平，不能够插队，必须先来后到!FIFO

非公平锁:非常不公平，可以插队（默认都是非公平)



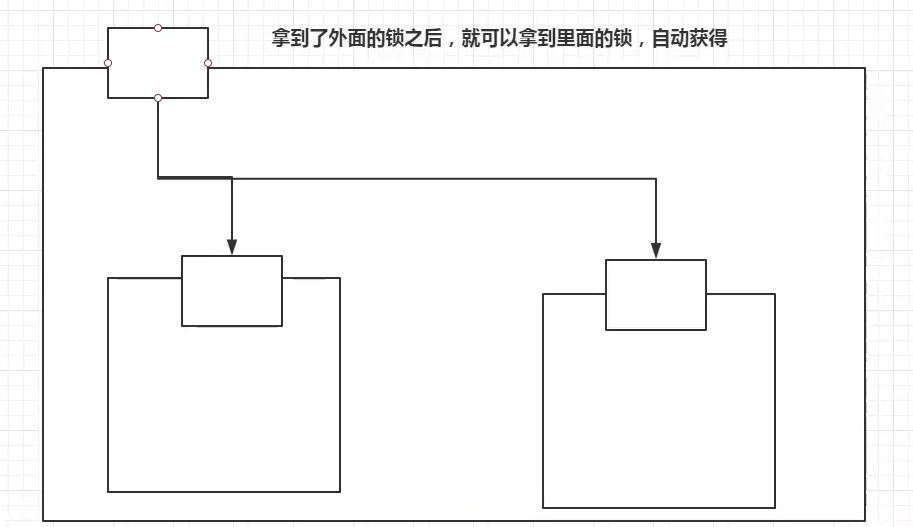
1. 可重入锁：可重入就是说某个线程已经获得某个锁，可以再次获取锁而不会出现死锁。

·可重入锁有

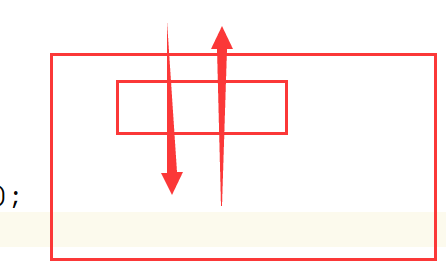
synchronized

ReentrantLock

synchronized的示意图如下：自动获取自动释放



ReentrantLock的示意图如下:ReentrantLock 和 synchronized 不一样，需要手动释放锁，所以使用 ReentrantLock的时候一定要手动释放锁，并且加锁次数和释放次数要一样



示例代码如下：<https://blog.csdn.net/qq_42388853/article/details/112102145>

1. 自旋锁：

**自旋锁**（spinlock）：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

优点：

自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快。

自己实现一个自旋锁：<https://blog.csdn.net/qq_42388853/article/details/112105359>

缺点：

1. 如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。

2. 上面Java实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

“不患寡，而患不均”，如果线程优先级“不均”，在CPU繁忙的情况下，优先级低的线程得到执行的机会很小，就可能发生线程“饥饿”；持有锁的线程，如果执行的时间过长，也可能导致“饥饿”问题。

解决“饥饿”问题的方案很简单，有三种方案：一是保证资源充足，二是公平地分配资源，三就是避免持有锁的线程长时间执行。这三个方案中，方案一和方案三的适用场景比较有限，因为很多场景下，资源的稀缺性是没办法解决的，持有锁的线程执行的时间也很难缩短。倒是方案二的适用场景相对来说更多一些。

1. 死锁：

死锁的四个必要条件：

****（1） 互斥条件：一个资源每次只能被一个进程使用。****

****（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。****

****（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。****

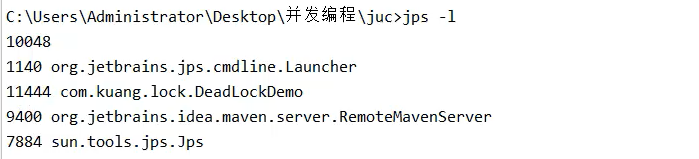
****（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。****

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁排查：

1. 使用jps -l定位进程号

例如：jps -l



1. 使用jstack 进程号 找到死锁问题

例如：jstack 110



面试，工作中，排查问题：

1. 日志
2. 堆栈信息，jps -l