Juc补充日志

Lock锁

Lock是一个接口有三个实现类，可重入锁，读锁，写锁，还可以通过lock对象去获取candition对象，通过这个对象可以实现wait和notify等功能，除此之外还可以实现精准监控每一个线程，他的实现类如ReentrantLock采用的是cas+volatile来进行实现。

Lock的实现原理

Lock是基于同步队列来进行实现（AbstractQueueSynchronizer），也就是Aqs

它提供了三个方法

getStatus ：获取同步状态

setStatus：设置同步状态

CompareAndSetStatus：使用cas原语设置状态保证原子性

除此之外Aqs还使用双向列表维持一个队列，来存放没有获取锁而等待的线程节点

对于一个线程获取锁就是要使用CompareAndSetStatus修改标志位，如果成功则获取锁，如果不成功则进行cas继续获取锁，或者超时退出。

锁的到底是谁

对于普通的synchronized方法，或者lock，我们所的是调用者，即当前对象，对象不同不影响调用，并且如果一个方法因为获取不到锁而阻塞在某个带锁的方法上，不会影响这个对象调用无锁方法，

对于静态的方法加锁，那么我们锁的是类对象，类对象只有一个，所以多个对象是无法同时调用带锁的静态方法。

Juc安全的list，安全的set 安全的map

List<String> list = new CopyOnWriteArrayList<>()；

Set<String> set = new CopyOnWriteArraySet<>();

Currenthashmap

普通版本

Set<String> set = Collections.synchronizedSet(new HashSet<>());

List<String> list = Collections.synchronizedList(new ArrayList<> ());

CopyOnWriteArrayList，CopyOnWriteArraySet采用的是cas+volatile，在读取的时候不加锁，在写入的时候会复制一个新的数组，然后写入，然后覆盖，并且写入的时候不会影响读取数据，适合读多写少的场景，他的缺点就是，占用内存大，还有就是不能保证数据实时一致性，只能保证最终一致性

Set的本质就是hashmap，实现方法如下

Set代码如下

public HashSet() {

map = new HashMap<>();

}

// add set 本质就是 map key是无法重复的！

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

private static final Object PRESENT = new Object(); // 不变得值！

Callable接口

Callable接口也可以产生线程，与runnable类似

他们之间的区别

1. callable可以拿到返回值
2. Callable可以抛出异常
3. Runnable采用run来写线程方法，callable采用call（）来写线程方法

如何使用callable接口，

首先我们看看启动线程的方法，会发现它只能传入runnable的实现类

New Thead（rubnnable（））.start();

因此，callable先要启动只能采用rubnnable的实现类futerTask来封装callable接口的实现类，从而达到启用callable，并且可以从futerTask上得到返回值

即

New Thead（FuterTask(callable)).start(); FuterTask.get() //拿到返回值

Callable注意事项

1、callable的call方法执行很长时间，那么FuterTask.get()方法，这个方法如果可能会阻塞，

2、get方法会有缓存，效率较高，详细看下面代码



ReadWriteLock

读写锁，好处就是不阻塞读取，但是阻塞写

ReadWriteLock

读-读 可以共存！

读-写 不能共存！

写-写 不能共存！

阻塞队列



阻塞队列的四组api：队满抛出异常，不抛出异常返回true或者flase，队满阻塞等待，队满阻塞等待超时退出。



SnchronousQueue 同步队

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！ put、take

线程调度工具类

Coundowmlunch（减法计算器）cyclicbaric（加法计算器）semophgore（信号量）：这三个类是线程调度调节的工具类，可以协调线程进行操作，如在cvte计算所有快递公司这个月的平均单价，总价，总体单价，总体价格，那么就可以用countdownlunch协调每一个线程去调用接口查询价格，然后统计

线程池：自己回忆

线程池线程数目如何设置

Cpu密集型：几个核心就是几，可以保证线程快速运行

Io密集型：线程数 = CPU 核心数 \* (1 + IO 耗时/ CPU 耗时)

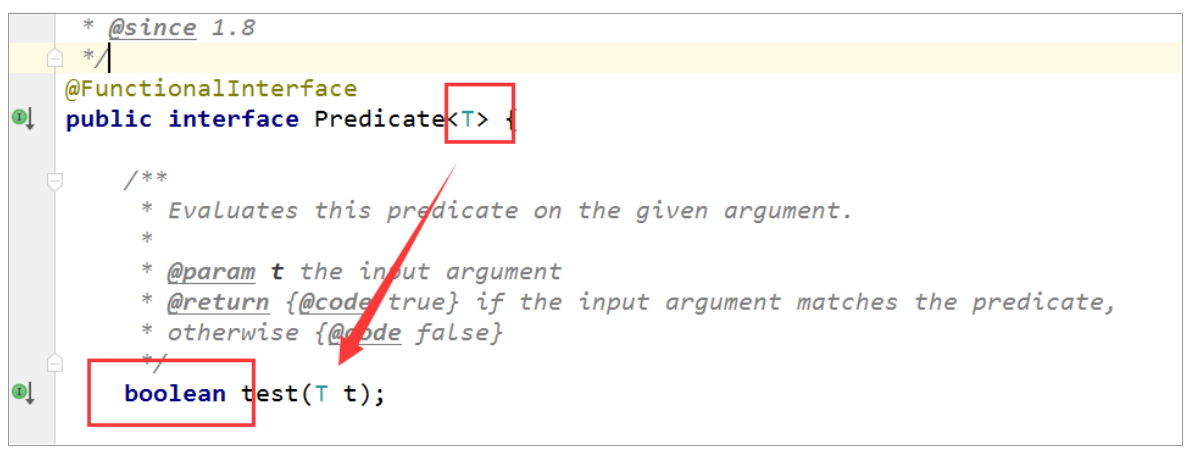
重点：四大函数试接口

函数试接口的一个特点就是函数式接口里面都只有一个方法，并且方法需要自己实现，并且都可以用lamda来进行替代

1、Function函数式接口



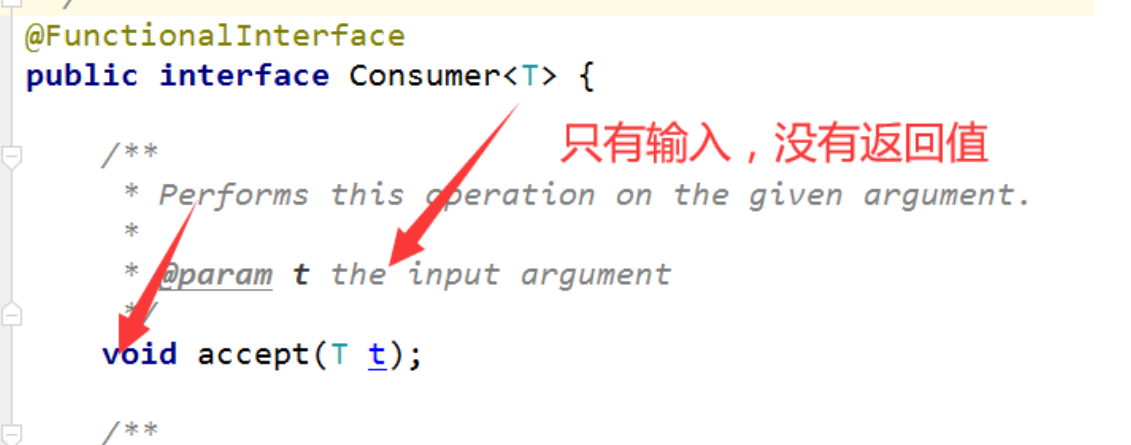
2predicate判定型接口



3supplir供给型接口，就是提供服务的接口



4Cummer消费型 接口



Stream流式计算

程序大部分都是存储+计算，存储我们可以交给集合，而计算可以用流

ForkJoin 任务拆解与合并

它可以将大人物拆解为小任务，然后将结果进行合并得出结果，典型的例子就是计算1加到10亿，他的核心其实是递归计算

除此之外，并行流计算的做法也是采用forkjion的原理。

任务窃取：frokJoin采用的任务存储方式是双端队列，当一个线程完成任务的时候会从双端队列里面获取别的线程的任务来执行，从而提高效率

**关于JMM的一些同步的约定：**

1. 线程解锁前，必须把共享变量**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

1. 加锁和解锁是同一把锁

单例模式

使用valitale的原因，  
 因为创建singleModle的过程分为三步，1开辟空间，2构造方法初始化，3将引用赋给 singleModle 这三步是没有关系的，可以进行指令冲排序，如果是132的执行顺序，一旦3执行完就认为对象已经存在 可能会导致一个没有初始化的对象就被返回出去， 使用volatile可以避免这种情况  
  
为什么需要双重检（两次if）测锁，

因为在第一次初始化对象的时候可能会有多个对象同时执行“singleModle == null”  
导致多个线程进入，如果不再多一次if判断，可能会创建多个对象

反编译代码

Jd-ui

Transist关键字

换句话来说就是，用transient关键字标记的成员变量不参与序列化过程。也就是说这个变量在传输的时不参与二进制序列化。

Cas原语

Cas的原理在于它的底层是利用java的unsafe类的本地方法调用c++然后直接操作内存进行执行操作系统本身就存在的cas原语，这个原语是原子操作，所以满足原子性

可重入锁

可重入锁就是当一个对象已经获取到了外层的锁的时候他也自动会获取到内层的锁，可以大大防止死锁

解决死锁

1日志

2堆栈信息

1. 使用 jps -l 定位进程号



2、使用 jstack 进程号 找到死锁问题

