juc并发编程

1.多线程的概念，目的，困难

1.1 多线程的基本概念

1. 线程与进程：进程是系统调度的基本单位，一个进程就相当于电脑的应用，对于线程而言，线程是cpu资源分配的基本单位，并发执行，宏观上并发执行，微观上实际是串行的。
2. 上下文切换：实际上说的是当前线程cpu时间片结束后，调用另外一个线程所消耗的资源，叫上下文切换，举个例子：我们只有一核cpu,cpu可以支持多个线程，cpu为每隔线程分配时间片，当一个线程的时间片结束时，调用另外一个线程所消耗的资源。
3. 资源受限：说的是多线程并不是越多越好，执行效率就越高，实际上这个和你的硬件资源也有关系，受限与别的资源。

1.2 多线程的目的

1. 提高代码的执行效率

1.3 多线程带来的问题

- 死锁问题: 就相当于多个线程来争抢多个临界资源，由于顺序推进不当，形成了环路等待的条件：什么意思：举例：两个人去吃西餐：一个人用刀，一个人用叉，结果都吃不到西餐，互不相让，最终两人只能在这等着，谁也别吃。
- 数据不准确：由于多线程改变数据，可能导致数据错乱更改，与我们的结果不一致问题。

2. 线程的创建，状态，通信

2.1 线程的创建方式

- 继承Thread这个类，重写run()方法；
- 实现Runnable接口，实现run()方法;
- 实现Callable接口，实现run()方法；这个接口的方法有返回值，可以抛异常，但是他在创建线程的时候需要配合FultureTast使用；
- 使用线程池：线程池呢，他最大父类就是Executor 可以通过Executors工具类获取三种类型的线程池，固定类型，cache线程池，单线程池；实际上这三个线程池都实现了一个叫ThreadpoolExecutor的类，他有五个核心参数，核心线程数，最大线程数，队列类型，线程的存活时间，拒绝处理；我们的单线程池和固定类型的线程池核心线程数是，一个或者固定数量，队列为无限队列，当有1000000个任务提交给这个线程池，首先核心线程数会先去执行任务，剩下大量的任务都会放在无限队列中，这会导致oom，而cache线程池，无核心线程数，来一个任务创建一个线程去处理，这就意味这来百万个任务，我得开启百万个线程去处理，不仅cpu会100%;也可能会导致oom;所以实际上我们是自己new ThreadpoolExecutor()线程池对象去处理任务，根据自己的业务指定一个固定大小的队列，既不会oom，也不会cpu100%;

2.2 线程的状态

- 线程的状态：分为新生，就绪，执行，终止，阻塞
- 新生就是我们new Thread（）的时候为新生态，当我们调用start()方法的时候会成为一个就绪的状态，争抢到cpu资源会变为执行态，正常执行结束叫终止态，阻塞分为两种 sleep()睡眠阻塞，还有就是wait()这种加锁原因导致阻塞。

2.3 线程间如何通信

- 线程通信，指的是线程与线程之间可以进行通信，从而避免一些线程带来的死锁，或者数据不一致问题，1.5版本之前，sychronized关键字解决线程安全问题，sychronized需要一个锁对象，我们的任何对象都可以是一把锁，对象**有wait()方法，等待，notify notifyall唤醒的方法，可以**用于线程之间的一个通信.

- Interface Condition,可以通过lock锁的获取，Condition这个类
- 调用await()和signal()方法阻塞和唤醒线程
- 必须结合lock一起使用

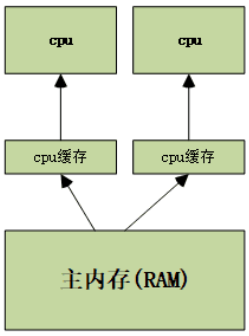
park unpark

3. 线程安全问题的根源

3.1 jmm内存模型

- 从硬件的角度而言，我们经常会存在cpu和磁盘IO速度不匹配问题，可以通过高速缓冲区解决问题，cache1,cache2等，jvm是模拟真实计算机的，所以他的结构同样有主内存，线程,工作内存副本来解决速度不匹配问题；

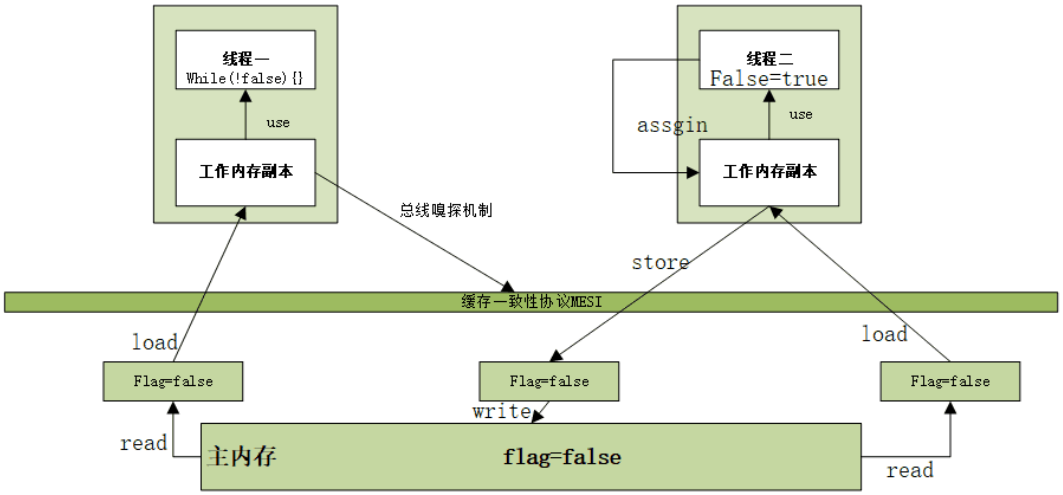
显示计算机内存模型



voitail第二大作用为有序性，可防止指令重排，由于cpu为了提高运算效率，
有时候会对我们的代码进行指令重排，指令重排有两个原则，第一as-if-serial
原则,在单线程环境下不影响结果的前提下进行指令重排，第二个原则就是
happens-before原则happerr-before原则有很多规则，
例同步代码不能指令重排，被voitail修饰的不能进行指令重排，这边是保证有序性。

voitail为了保证可见性会将工作内
存副本中的数据立即同步到主内存，
并开启缓存一致性协议，让其他线程
工作内存中的数据失效，重新加载

java多线程内存工作机制



3.2 volatile的可见性和有序性

- volatile的可见性：volatile关键字加上有这么几个作用，能够将工作内存副本的数据快速同步回主内存，开启MESI缓存一致性协议，让其他工作内存副本的数据失效。最好介绍jmm内存模型的流程；
- volatile的有序性：cpu为了提高效率经常对我们的程序，在单线程情况下，不影响执行结果的时候会对其进行指令重排，单线程环境下不会影响结果，但是多线程就有可能不同；而volatile可以防止指令重排，不指令重排的条件有两个，一个叫happen-before原则，和as-if-as原则，而在happen-before中就有很多规则，比如volatile关键字规则，锁标记规则等；举个例子，我们在写单例模式的时候会写一个双重检测的单例模式，由于我们的变量是静态私有的，我们创建对象有一个过程，先类

加载---验证---准备，准备阶段就会定义类变量，分配内存，初始化类变量，给一个默认值，最后进行实际的初始化；如果我们不加volation关键字，cpu对我们的代码程序进行指令重排，如果返回值先返回在new对象赋值之前返回，我们会得到一个默认值为null的对象，这变会导致空指针异常，所以对其加上volation关键字。

4. 如何保证线程安全

4.1 synchronized控制线程安全

1. synchronized关键字jdk1.5版本之前

sychronized在5以前，他的实现比较简单，我们synchronized一般有三种用法，同步代码块，实例方法上（锁的对象是当前对象），对于静态方法而言（锁的对象是类对象）；实际的原理是如何的，加上这个关键字，我们的多个线程来访问临界资源的时候，不管怎样，都会做这么一大堆操作，需不需要加锁，加锁，没有得到锁，切换上下文，去阻塞队列等待，这些大多数都是操作系统层面的操作，非常耗费系统资源；

2. jdk1.6之后对锁升级过程

jdk1.6对锁有了一个升级过程，先是一个无锁的状态，偏向锁，轻量级锁，重量级锁，这些锁是如何实现的，实际上我们对象在生成的时候不仅要生成实例数据，还要生成对象头，数据部分，对其填充，对其填充没有什么作用，数据部分就是我们要初始化的数据部分，对象头是关键在64位的虚拟地址中，31位代表hash值，25位为空闲的高位，4位表示分代年龄，2位代表锁标记，1位代表偏向锁，1位空闲，锁标记位：00 01 10 11刚好能代表四种锁标记的状态，下面说一下锁的升级过程；当没有任何线程来访问的时候，是一个无锁的状态，有一个线程来访问的时候，会将这个线程的线程id放在对象头的markword中，如果下次还是当前线程，就直接放行，如果不是会升级成一个轻量级锁，轻量级锁：就是我们的线程是公平的，没有加锁，都去争抢临界资源，谁抢到谁去执行，没抢到就在自旋加锁，如果有高并发的线程都来访问，这个时候，如果使用自旋加锁，可能会导致大量空旋；cpu100%这个时候还不如升级为重量级锁，直接去阻塞队列等着；

4.2 lock锁保证线程安全

1. AQS

- 原理：维护两个东西，一个叫阻塞队列，结点的特点，另一个叫state,所谓的加锁，就是改变state的状态，如果为0，空闲状态，为1被占用了。
- 公平锁的源码刨析

```
abstract void lock();
```

```
static final class FairSync extends Sync {
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() {
        acquire( arg: 1);
    }
```

1 公平锁的实现，进方法

```
 */
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}

/**
```

1 情况一：先去尝试加锁，没有加到锁，就会进行将结点添加到队列中去

```
     */
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (!hasQueuedPredecessors() &&
                compareAndSetState( expect: 0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0)
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
}
```

1 获取锁的时候会先看阻塞队列有没有，没有，返回false才会去通过cas获取锁

2 这里代表可重入

```java
public final boolean hasQueuedPredecessors() {
    // The correctness of this depends on head being initialized
    // before tail and on head.next being accurate if the current
    // thread is first in queue.
    Node t = tail; // Read fields in reverse initialization order
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

1 阻塞队列有没有

链表不为Null

2 false,进一步判断后面有没有

```java
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

1 回到此处，没加到锁，走acquireQueued()

```java
volatile int waitStatus;

/** Link to predecessor node that current node/thread relies on ...*/
volatile Node prev;

/** Link to the successor node that the current node/thread ...*/
volatile Node next;

/** The thread that enqueued this node.  Initialized on ...*/
volatile Thread thread;
```

初始值为0，当有线程添加队列的时候被改为1

前指针

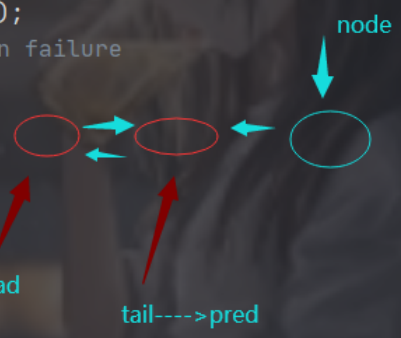
后指针

value值就是当前线程

```java
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

1 cas

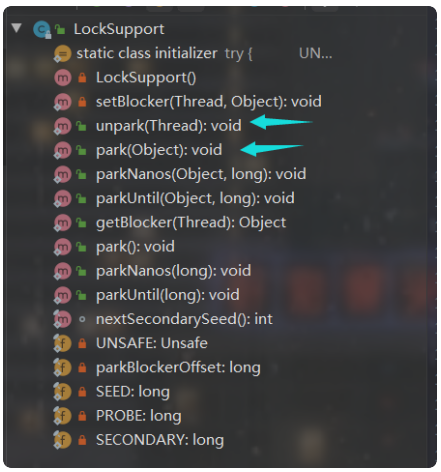
2 没有进入阻塞队列的线程一定要保证自旋添加成功



```
    */
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

1 将线程使用park进行阻塞住

lockSupport方法中有 park()或者unpark方法用于阻塞的，线程放弃当前所有资源，阻塞线程。



- 非公平锁的源码刨析

```
 */
final void lock() {
    if (compareAndSetState( expect: 0,  update: 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire( arg: 1);
}
```

1 上来就去争抢一次锁

2 没抢到，[redacted]，尝试加锁

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

1 非公平锁这里的实现方式不一样了

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

与公平锁的区别，在尝试获取一次

2. 独占锁可重入锁

可重入锁也叫独占锁，实际也是基于AQS实现的，他只是在AQS的判断基础之上，加锁成功后会将锁加锁的线程存起来，下次线程如果是当前存入的，直接返回加锁成功，达到一个可重入的作用；

3. 共享锁读写锁

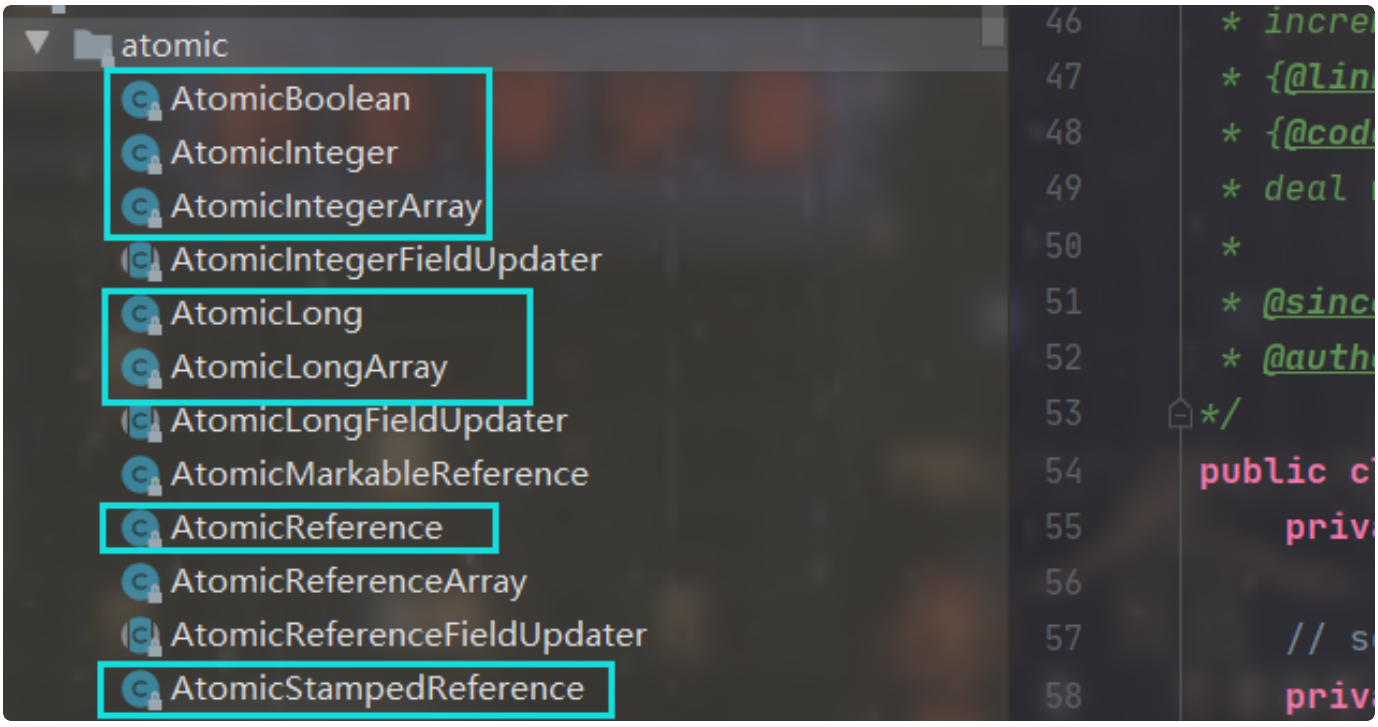
保证线程安全，其实不仅可以使得多线程实际上按照单线程的方式去执行，其实也可多线程去访问，比如读写锁，读锁：读与读可以并发执行，读与写，写和写互斥，提高我们的读写效率。

4.3 原子类保证线程安全

- cas概念----->我们一定要提到unsafe这个类

```
Class<Unsafe> unsafeClass = Unsafe.class;
Constructor<Unsafe> constructor = unsafeClass.getConstructor();
Unsafe unsafe = constructor.newInstance();
```

cas:compareAndSwap比较在交换：什么意思举个列子：当前有连个线程都来修改某一个值，我们cas一般有三个关键的变量，如果原来的值和副本不一样，ok，代表被线程篡改了，我就不在修改了，如果没有，就可以通过修改值；所带来的缺点：ABA问题 ，只能针对某一个数进行修改。分别都有响应的解决方案。



原子类操作，是原子性的，可以解决并发安全问题，也是基于unsafe.java去实现 13种原子类，列：long int boolean intergerArray LongArray reference 等。看上图.

原子类：分为三个类

- unsafe 里面的基本实现 interger long object

- 数组的引出：intergerArray longArray referenceArray =====>用于对数组里的某个元素进行操作
- 引用类型的：带版本号的StampedReference 带标记的 AtomicMarkableReference =====>对象本省的cas
- 操作字段的: LongFieldUpdater IntergerFieldUpdater RefenceFieldUpdater===>对象种的某一个字段

4.4 并发容器和框架

1. list: CopyOnWriteArrayList的原理

- copyonWriteArrayList是通过读写锁实现的，若对list是 一个读操作，怎不会加锁，当我们要改变list集合的元素的时候copyonwriteArraylist会先复制出一份来，加上锁，做到读写分离，互不影响，修改成功后，将修改后的数组赋值给原来的数组的引用地址即可；

2. set: CopyOnWriteArraySet的原理

- copyOnwriteArraySet是基于copyOnwriteArrayList集合实现的，大致原理和他相同，唯一不同的是我们在添加元素的时候，会先遍历一下集合种的元素，因为是set集合，做到不可重复原则;

3. queue:

- currentLinkedQueue 高并发场景下高效安全队列: currentLinkedQueue用于解决LinkedList线程安全问题；
- BlockQueue 生产者消费者问题：队列满的时候：阻塞插入线程，唤醒消费线程；反之，与之相反；举例：阻塞队列，当我们的阻塞队列为null的时候可以唤醒生产者线程来生产，阻塞消费者线程去消费队列；相反：我们的阻塞队列满的时候，可以唤醒消费者线程来消费，阻塞生产者线程生产；
- 固定队列，无限队列，同步队列：分别为固定大小，无限大小，大小为1的------>只有被线程消费了才可以继续入队；

4. map: currentHashMap的原理

略----->请看我后续笔记（较为复杂，我会单独出一篇去介绍）。

4.5 并发工具包

1. 都是基于AQS实现的，各自的与原理说清除.

并发工具包 Java 复制代码

```java
CountDownLatch countDownLatch = new CountDownLatch(4);//设置计数的时间countdown-1
 Semaphore semaphore = new Semaphore(5); //资源个数
 CyclicBarrier cyclicBarrier = new CyclicBarrier(3,new Runnable());//设置到达的线程数，优先执行的线程
```

2. CountDownLatch: 用于让一个线程或者多个线程等待其他线程同时出发；只能触发一次，构造器里面的参数是用来设置要等几个线程，来一个countDown就会减一，到达0的时候同时出发；相当于一个计数器，当多个线程执行时，他能控制多个线程同时去执行，里面有一些计数的方法：就好比跑步一样：时间到，大家都同时出发；**AQS实现**
3. CyclicBarrier：循环屏障；这个并发工具类的构造函数提供了两个参数，一个是需要到达线程的个数，一个是优先执行的线程；啥意思，到达线程个数为3--ok，只有当最后一个，也就是第三个线程到达，三个才会一起执行，假如你启动两个：代表这第三个永远不会到达，意味着三个线程都不会执行了；当然循环体现在它里面有一个重置的方法，可以再次使用，循环使用（这也是与countDownLatch的区别）**可重入锁实现**
4. Semaphore：相当于信号量；多个线程过来争夺信号量，假如我现在通过构造函数设置了10个信号量，一个线程需要6，一个需要5个，另一个需要1个，ok执行顺序就是6，1，5类似操作系统的银行家算法。**AQS实现**