二、并发编程

1. JUC简介

...

2. 原子类与CAS

...

3. Lock锁与AQS

3.1 Java锁简介

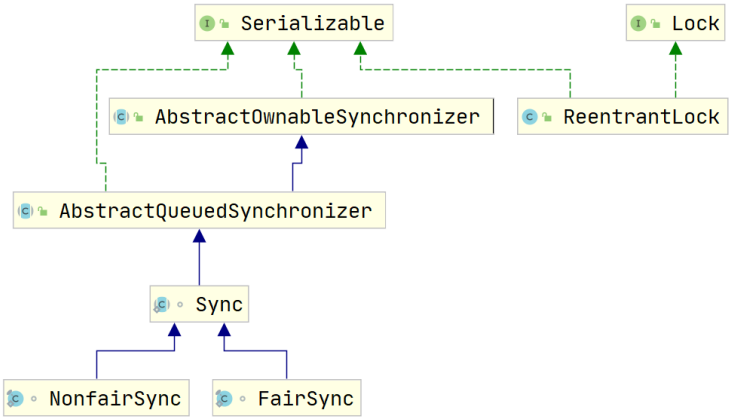
。。。

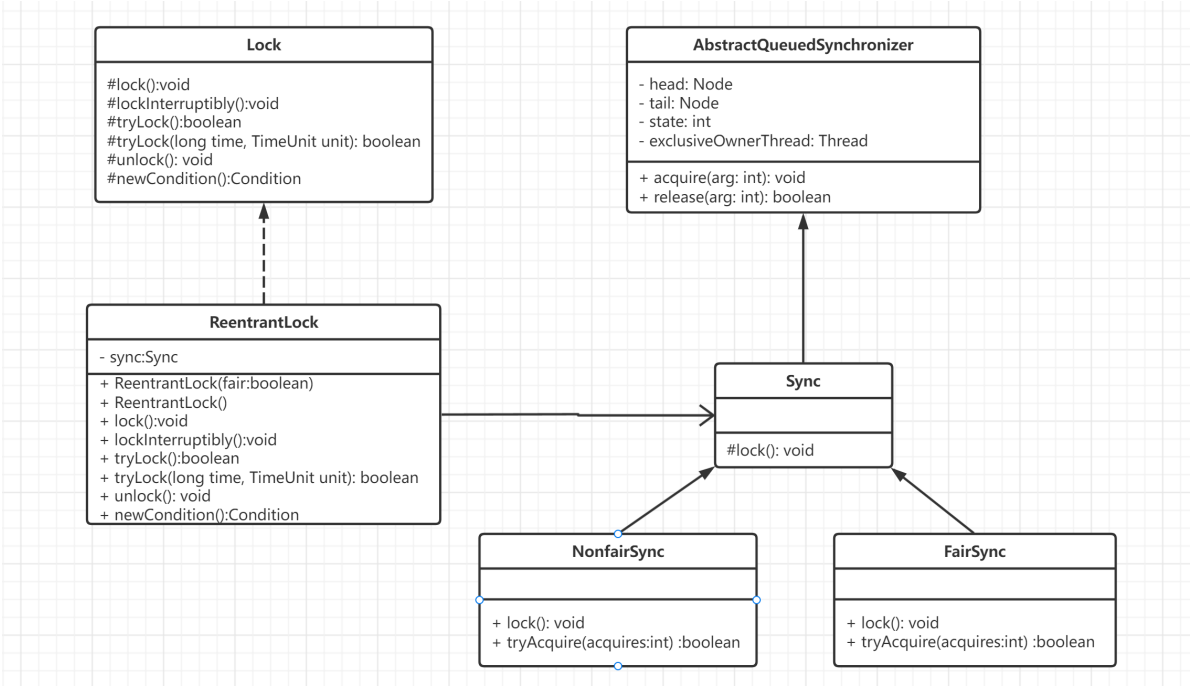
3.2 synchronized和JUC的锁对比

。。。

3.3 ReentrantLock源码分析

ReentrantLock类图



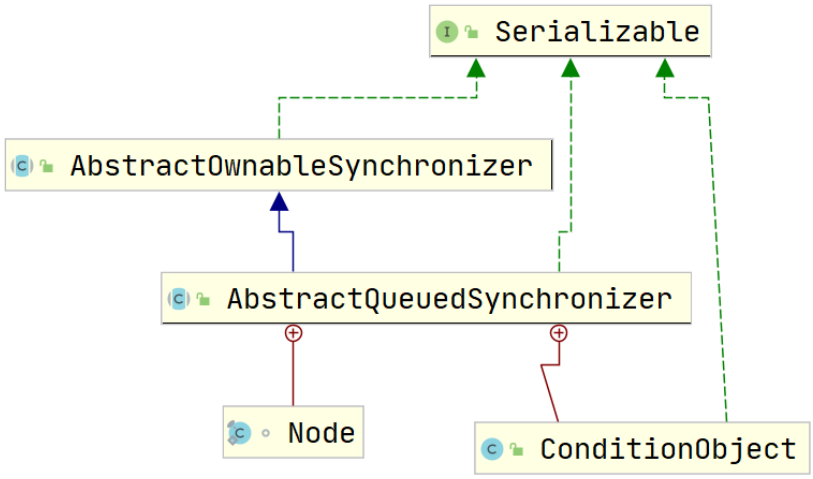


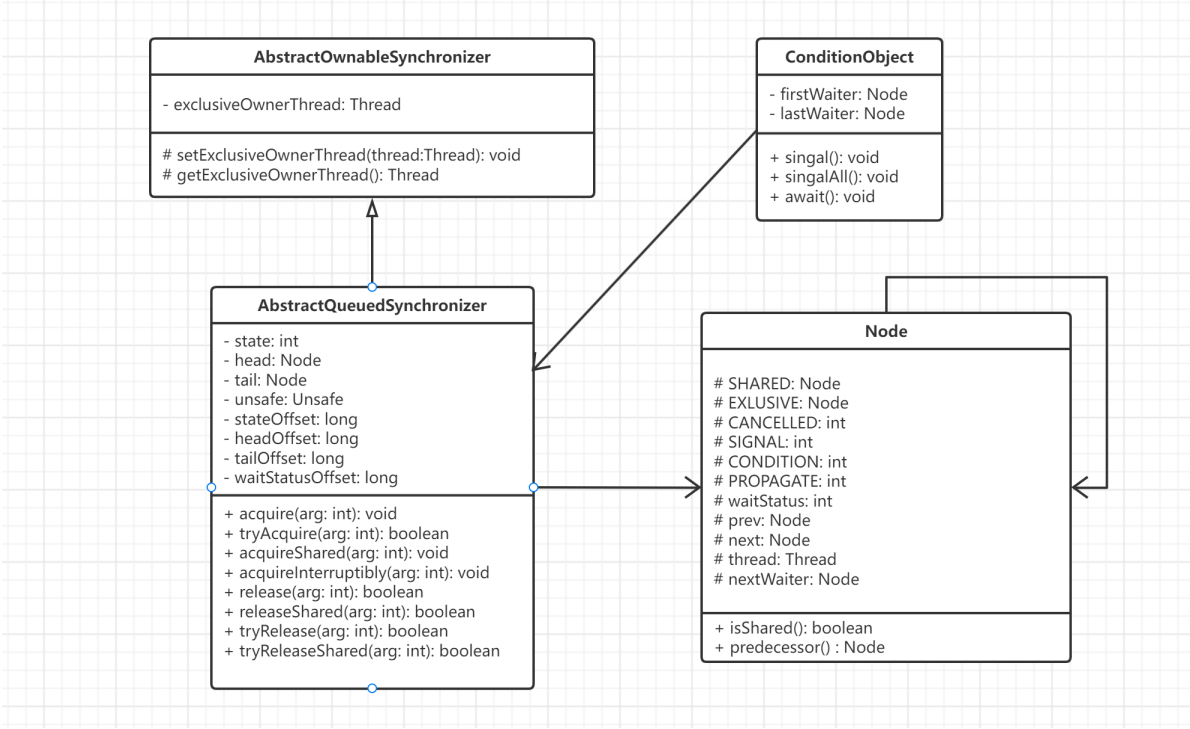
我们看一下重入锁ReentrantLock类关系图，它是实现了Lock接口的类。NonfairSync和FairSync都继承自抽象类Sync，在ReentrantLock中有非公平锁**NonfairSync**和公平锁**FairSync**的实现。

在重入锁ReentrantLock类关系图中，我们可以看到NonfairSync和FairSync都继承自抽象类Sync，而Sync类继承自抽象类AbstractQueuedSynchronizer（简称**AQS**）。如果我们看过JUC的源代码，会发现不仅重入锁用到了AQS， JUC 中绝大部分的同步工具也都是基于AQS构建的。**那AQS是什么作用呢?**

3.4 AQS简介

AQS（全称AbstractQueuedSynchronizer）即队列同步器。它是**构建锁**或者其他**同步组件的基础框架**（如ReentrantLock、ReentrantReadWriteLock、Semaphore等）。AQS是JUC并发包中的核心基础组件，其本身是一个抽象类。理论上还是利用管程实现的，在AQS中，有一个volatile修饰的state，获取锁的时候，会读写state的值，解锁的时候，也会读写state的值。所以AQS就拥有了volatile的happens-before规则。加锁与解锁的效果上与synchronized是相同的。





由类图可以看到，AQS是一个FIFO的双向队列，其内部通过节点head和tail记录队首和队尾元素，队列元素的类型为Node。

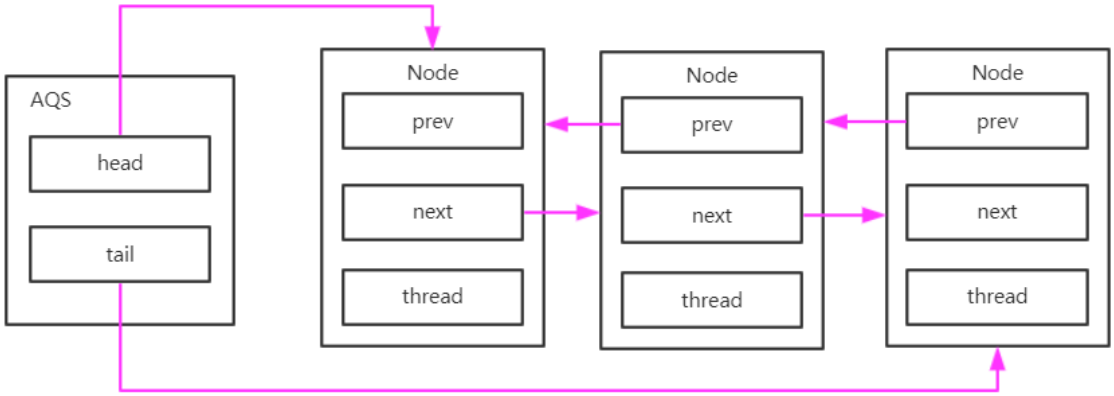
- Node中的thread变量用来存放进入AQS队列里面的线程，Node节点内部:
  - prev记录当前节点的前驱节点
  - next 记录当前节点的后继节点
- SHARED用来标记该线程是获取共享资源时被阻塞挂起后放入AQS队列的
- EXCLUSIVE用来标记线程是获取独占资源时被挂起后放入AQS队列的
- waitStatus 记录当前线程等待状态，可以为①CANCELLED (线程被取消了)、②SIGNAL(线程需要被唤醒)、③CONDITION(线程在CONDITION条件队列里面等待)、④PROPAGATE(释放共享资源时需要通知其他节点);

在AQS中维持了一个单一的状态信息state，对于ReentrantLock的实现来说，state 可以用来表示当前线程获取锁的可重入次数；AQS继承自AbstractOwnableSynchronizer，其中的exclusiveOwnerThread变量表示当前共享资源的持有线程。

3.5 AQS实现原理

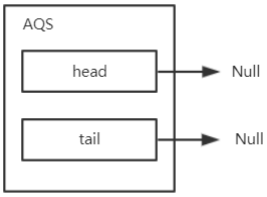
AQS是一个同步队列，内部使用一个FIFO的双向链表，管理线程同步时的所有被阻塞线程。双向链表这种数据结构，它的每个数据节点中都有两个指针，分别指向直接后继节点和直接前驱节点。所以，从双向链表中的任意一个节点开始，都可以很方便地访问它的前驱节点和后继节点。

我们看下面的AQS的数据结构，AQS有两个节点head，tail分别是头节点和尾节点指针，默认为null。AQS中的内部静态类Node为链表节点，AQS会在线程获取锁失败后，线程会被阻塞并被封装成Node加入到AQS队列中；**当获取锁的线程释放锁后，会从AQS队列中的唤醒一个线程（节点）。**

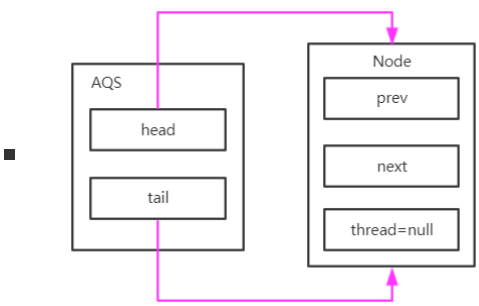


场景01-线程抢夺锁失败时，AQS队列的变化

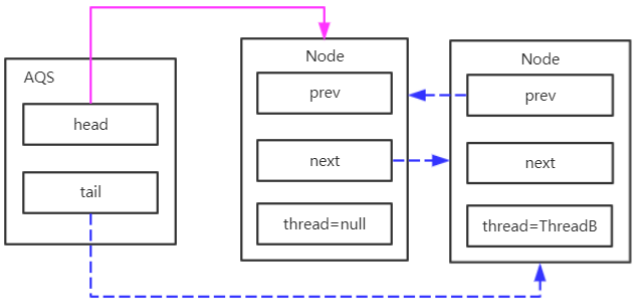
1. AQS的head，tail分别代表同步队列头节点和尾节点指针，默认为null。



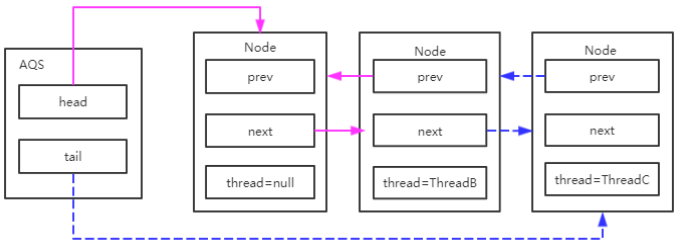
2. 当第一个线程抢夺锁失败，同步队列会先初始化，随后线程会被封装成Node节点追加到AQS队列中。假设当前独占锁的的线程为ThreadA，抢占锁失败的线程为ThreadB。
    - （1）同步队列初始化，首先会在队列中添加一个空Node，这个节点中的thread=null，代表当前获取锁成功的线程。随后，AQS的head和tail会同时指向这个节点。



- （2）接下来将ThreadB封装成Node节点，追加到AQS队列。设置新节点的prev指向AQS队尾节点；将队尾节点的next指向新节点；最后将AQS尾节点指针指向新节点。此时AQS变化，如下图：



3. 当下一个线程抢夺锁失败时，重复上面步骤即可。将线程封装成Node，追加到AQS队列。假设此次抢占锁失败的线程为ThreadC，此时AQS变化，如下图：



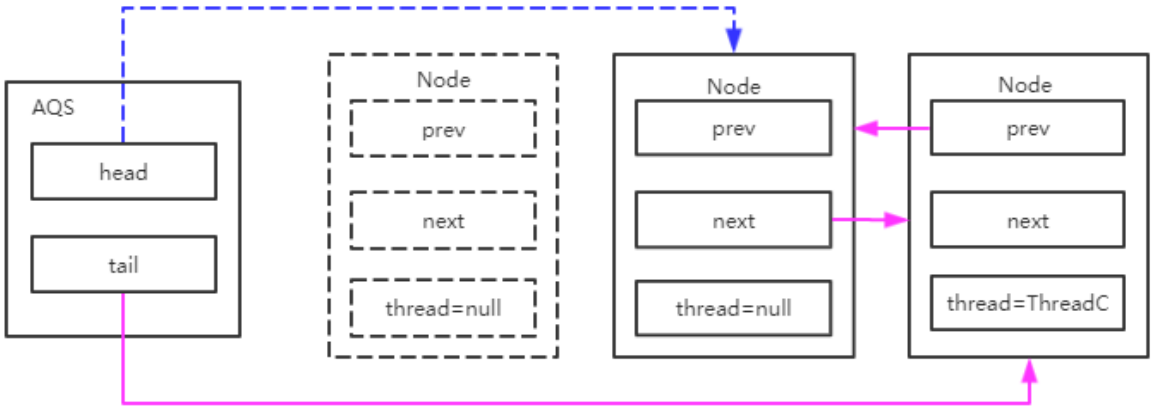
场景02-线程被唤醒时，AQS队列的变化

ReentrantLock唤醒阻塞线程时，会按照**FIFO的原则从AQS中head头部开始唤醒首个节点中线程。**

head节点表示当前获取锁成功的线程ThreadA节点。

当ThreadA释放锁时，它会唤醒后继节点线程ThreadB，ThreadB开始尝试获得锁，如果ThreadB获得锁成功，会将自己设置为AQS的头节点。ThreadB获取锁成功后，AQS变化如下：

1. head指针指向ThreadB节点。
2. 将原来头节点的next指向Null，从AQS中删除。
3. 将ThreadB节点的prev指向Null，设置节点的thread=null。



上面是线程在竞争锁时，线程被阻塞和被唤醒时AQS同步队列的基本实现过程。

3.6 ReentrantLock源码分析：锁的获取

研究任何框架或工具都就要一个入口，我们以重入锁为切入点来理解AQS的作用及实现。下面我们深入ReentrantLock源码来分析AQS是如何实现线程同步的。

AQS其实使用了一种典型的设计模式：模板方法。我们如果查看AQS的源码可以看到，AQS为一个抽象类，AQS中大多数方法都是final或private的，也就是说AQS并不希望用户覆盖或直接使用这些方法，而是只能重写AQS规定的部分方法。

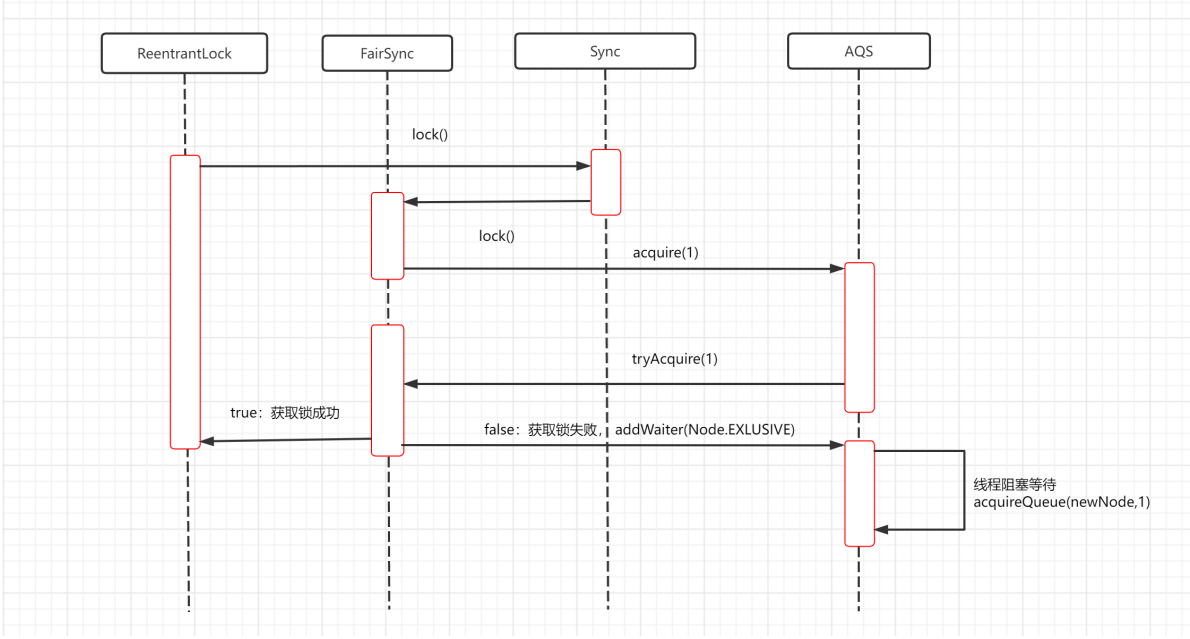
```
//AQS内部维护这一个双向链表，AQS主要属性
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {
        private transient volatile Node head;//头节点指针
        private transient volatile Node tail;//尾节点指针
        private volatile int state;//同步状态，0无锁；大于0，有锁，state的值代表重入次数。

        //AQS链表节点结构
        static final class Node {
            static final Node SHARED = new Node();//共享模式
            static final Node EXCLUSIVE = null;//独占模式
            /**
              *等待状态：取消。表明线程已取消争抢锁并从队列中删除。
              *取消动作：获取锁超时或者被其他线程中断。
              */
            static final int CANCELLED =  1;
            /**
              *等待状态：通知。表明线程为竞争锁的候选者。
              *只要持有锁的线程释放锁，会通知该线程。
              */
            static final int SIGNAL    = -1;
              /**
              *等待状态：条件等待
              *表明线程当前线程在condition队列中。
              */
            static final int CONDITION = -2;
              /**
              *等待状态：传播。
              *用于将唤醒后继线程传递下去，这个状态的引入是为了完善和增强共享锁的唤醒机制。
              *在一个节点成为头节点之前，是不会跃迁为此状态的
              */
            static final int PROPAGATE = -3;
            volatile int waitStatus;
            volatile Node prev;//直接前驱节点指针
            volatile Node next;//直接后继节点指针
            volatile Thread thread;//线程
            Node nextWaiter;//condition队列中的后继节点

            final boolean isShared() {//是否是共享
                return nextWaiter == SHARED;
            }
```

```
            final Node predecessor() throws NullPointerException {
                Node p = prev;
                if (p == null)
                    throw new NullPointerException();
                else
                    return p;
            }
            Node() {//默认构造器
            }
            //在重入锁中用于addWaiter方法中，用于将阻塞的线程封装成一个Node
            Node(Thread thread, Node mode) {
                this.nextWaiter = mode;
                this.thread = thread;
            }
            Node(Thread thread, int waitStatus) {//用于Condition中
                this.waitStatus = waitStatus;
                this.thread = thread;
            }
        }
    }
```

我们以重入锁中相对简单的公平锁为例，以获取锁的 lock 方法为入口，一直深入到AQS，来分析多线程是如何同步获取锁的。

```
Lock lock = new ReentrantLock(true);//创建ReentrantLock公平锁实例
lock.lock();//获取锁
try {
    //业务代码
} finally {
    //释放锁，在finally语句块确保锁最终会释放。
    lock.unlock();
}
```

获取锁时源码的调用过程，时序图如下：



第一步：ReentrantLock.lock()

ReentrantLock获取锁调用了 lock 方法，我们看下该方法的内部：调用了sync.lock()。

```
public void lock() {
    sync.lock();
}
```

sync是Sync类的一个实例，Sync类实际上是ReentrantLock的抽象静态内部类，它集成了AQS来实现重入锁的具体业务逻辑。AQS是一个同步队列，实现了线程的阻塞和唤醒，没有实现具体的业务功能。在不同的同步场景中，需要用户继承AQS来实现对应的功能。

我们查看ReentrantLock源码，可以看到，Sync有两个实现类公平锁FairSync和非公平锁NoFairSync。

重入锁实例化时，根据参数fair为属性sync创建对应锁的实例。以公平锁为例，调用sync.lock事实上调用的是FairSync的lock方法。

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

第二步：FairSync.lock()

我们看下该方法的内部，执行了方法acquire(1)，acquire为AQS中的final方法，用于竞争锁。

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

第三步：AQS.acquire(1)

线程进入AQS中的acquire方法，arg=1。

这个方法逻辑：先尝试抢占锁，抢占成功，直接返回；

抢占失败，将线程封装成Node节点追加到AQS队列中并使线程阻塞等待。

（1）首先会执行tryAcquire(1)尝试抢占锁，成功返回true，失败返回false。抢占成功了，就不会执行下面的代码了

（2）抢占锁失败后，执行addWaiter(Node.EXCLUSIVE)将x线程封装成Node节点追加到AQS队列。

（3）然后调用acquireQueued将线程阻塞，线程阻塞。

线程阻塞后，接下来就只需等待其他线程唤醒它，线程被唤醒后会重新竞争锁的使用。

接下来，我们看看这个三个方法具体是如何实现的。

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

第四步：FairSync.tryAcquire(1)

尝试获取锁：若获取锁成功，返回true；获取锁失败，返回false。

这个方法逻辑：获取当前的锁状态，如果为无锁状态，当前线程会执行CAS操作尝试获取锁；若当前线程是重入获取锁，只需增加锁的重入次数即可。

```
//尝试以独占模式获取锁
//若锁是未锁定状态state=0，CAS修改state=1，修改成功说明当前线程获取锁成功，设置当前线程为锁持有者，然后返回true。
protected final boolean tryAcquire(int acquires) {

    final Thread current = Thread.currentThread();
    int c = getState();//状态：0未锁定，大于0已被其他线程独占。
    if (c == 0) {//未锁定，可以获取锁
        if (!hasQueuedPredecessors() && compareAndSetState(0, acquires))
{//CAS设置state为1
            setExclusiveOwnerThread(current);//设置当前线程为独占资源持有者
            return true;
        }
    }
    //如果当前线程已经是为锁持有者，设置重入次数，state + 1
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;//设置重入次数+1
        //重入次数，超过int最大值，溢出。
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);//设置重入次数
        return true;
    }
    return false;
}
```

第五步：AQS.addWaiter(Node.EXCLUSIVE)

线程抢占锁失败后，执行addWaiter(Node.EXCLUSIVE)将线程封装成Node节点追加到AQS队列。

addWaiter(Node mode)的mode表示节点的类型，Node.EXCLUSIVE表示是独占排他锁，也就是说重入锁是独占锁，用到了AQS的独占模式。

Node定义了两种节点类型：

- 共享模式：Node.SHARED。共享锁，可以被多个线程同时持有，如读写锁的读锁。
- 独占模式：Node.EXCLUSIVE。独占很好理解，是自己独占资源，独占排他锁同时只能由一个线程持有。

```
static final Node SHARED = new Node();//共享模式
static final Node EXCLUSIVE = null;//独占模式
```

相应的AQS支持两种模式：支持独占模式和共享模式。

```
/*
 * 模式有两种：共享模式和独占模式
 */
private Node addWaiter(Node mode) {
    //当前线程封装为Node准备排队获取锁
    Node node = new Node(Thread.currentThread(), mode);
    //先尝试快速插入同步队列。如果失败，再使用完整的排队策略。
    Node pred = tail;
    if (pred != null) {//如果双向链表不为空链表（有节点），追加节点到尾部
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);//链表为空，将节点追加到同步队列队尾
    return node;
}

//通过自旋插入节点到同步队列AQS中，如果队列为空时，需先初始化队列。
private Node enq(final Node node) {
    for (;;) {//自旋，至少会有两次循环。
        Node t = tail;
        if (t == null) { //队列为空，先初始化队列
            if (compareAndSetHead(new Node()))//CAS插入节点
                tail = head;
        } else {//插入节点，追加节点到尾部
            node.prev = t;
            if (compareAndSetTail(t, node)) {//CAS插入节点
                t.next = node;
                return t;
            }
        }
    }
}
```

第六步：AQS.acquireQueued(newNode,1)

这个方法的主要作用就是将线程阻塞。

1. 若同步队列中，若当前节点为队列第一个线程，则有资格竞争锁，再次尝试获得锁。
    - 尝试获得锁成功，移除链表head节点，并将当前线程节点设置为head节点。
    - 尝试获得锁失败，判断是否需要阻塞当前线程。
2. 若发生异常，取消当前线程获得锁的资格。

```
    /**
     *等待队列中的线程以独占的模式获取锁
```

```
     * @param node 新加入等待队列线程节点
     * @param arg 获取参数
     * @return {@code true} 在等待中是否被中断
     */
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;//获取锁是否失败，一般是发生异常
    try {
        boolean interrupted = false;//是否中断
        for (;;) {//无限循环，线程获得锁或者线程被阻塞
            final Node p = node.predecessor();//获取此节点的前一个节点
            //若此节点的前个节点为头节点，说明当前线程可以获取锁，阻塞前尝试获取锁，若获取锁成功，将当前线程从同步队列中删除。
            if (p == head && tryAcquire(arg)) {//获取锁成功
               /**
                * 将当前线程从同步队列中删除。
                * 将当前节点置为空节点，节点的prev，next和thread都为null。
                * 将等待列表头节点指向当前节点
                */
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }

            if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
                interrupted = true;//当前线程被中断
        }
    } finally {
        //如果出现异常，取消线程获取锁请求
        if (failed)
            cancelAcquire(node);
    }
}

private void setHead(Node node) {
     head = node;
     node.thread = null;
     node.prev = null;
 }
```

AQS.shouldParkAfterFailedAcquire

这个方法的主要作用是：线程竞争锁失败以后，通过Node的前驱节点的waitStatus状态来判断， 线程是否需要被阻塞。

1. 如果前驱节点状态为 SIGNAL，当前线程可以被放心的阻塞，返回true。
2. 若前驱节点状态为CANCELLED，向前扫描链表把 CANCELLED 状态的节点从同步队列中移除，返回false。
3. 若前驱节点状态为默认状态或PROPAGATE，修改前驱节点的状态为 SIGNAL，返回 false。
4. 若返回false，会退回到acquireQueued方法，重新执行自旋操作。自旋会重复执行acquireQueued和shouldParkAfterFailedAcquire，会有两个结果：

（1）线程尝试获得锁成功或者线程异常，退出acquireQueued，直接返回。

（2）执行shouldParkAfterFailedAcquire成功，当前线程可以被阻塞。

5. 若返回true，调用parkAndCheckInterrupt阻塞当前线程。

Node 有 5 种状态，分别是：

- 0：默认状态。
- 1：CANCELLED，取消/结束状态。表明线程已取消争抢锁。线程等待超时或者被中断，节点的waitStatus为CANCELLED，线程取消获取锁请求。需要从同步队列中删除该节点
- -1：SIGNAL，通知。状态为SIGNAL节点中的线程释放锁时，就会通知后续节点的线程。
- -2：CONDITION，条件等待。表明节点当前线程在condition队列中。
- -3：PROPAGATE，传播。在一个节点成为头节点之前，是不会跃迁为PROPAGATE状态的。用于将唤醒后继线程传递下去，这个状态的引入是为了完善和增强共享锁的唤醒机制。

```
 /**
     *是否需要阻塞当前线程，根据前驱节点中的waitStatus来判断是否需要阻塞当前线程。如果线程需要被阻塞，返回true，这是自旋中的主要的信号量。
     * @return {@code true} 如果线程需要被阻塞，返回true。
     */
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;//上一个节点的waitStatus的状态
        if (ws == Node.SIGNAL)
            //前驱节点为SIGNAL状态，在释放锁的时候会唤醒后继节点， 当前节点可以阻塞自己。
            return true;
        if (ws > 0) {
         /**
          * 向前扫描链表把 CANCELLED 状态的节点从同步队列中移除。
          * 前驱节点状态为取消CANCELLED（1）时,向前遍历，更新当前节点的前驱节点为第一个非取消状态节点。
          * 之后，
          * （1）当前线程会再次返回方法acquireQueued，再次循环，尝试获取锁；
          * （2）再次执行shouldParkAfterFailedAcquire判断是否需要阻塞。
          */
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            /*前驱节点状态<=0,此时还未判断的状态有 默认状态(0)/CONDITION(-2)/PROPAGATE(-3)。
             *此时，不可能是CONDITION(-2)，所以只能是默认状态(0)/PROPAGATE(-3)。
             *CAS设置前驱节点的等待状态waitStatus为SIGNAL状态。
             *此次，当前线程先暂时不阻塞。
             *之后，
             * （1）当前线程会再次返回方法acquireQueued，再次循环，尝试获取锁；
             * （2）再次执行shouldParkAfterFailedAcquire判断是否需要阻塞。
             * （3）前驱节点为SIGNAL状态，可以被阻塞。
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

AQS.parkAndCheckInterrupt

将当前线程阻塞挂起。

LockSupport.park(this)会阻塞当前线程，会使当前线程（如ThreadB）处于等待状态，不再往下执行。

```
    /**
     * 将当前线程阻塞，并且在被唤醒时检查是否被中断
     * @return {@code true} 如果被中断，返回true
     */
    private final boolean parkAndCheckInterrupt() {
        //阻塞当前线程
        LockSupport.park(this);
        //检测当前线程是否已被中断（若被中断，并清除中断标志），中断返回 true，否则返回false。
        return Thread.interrupted();
    }
```

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

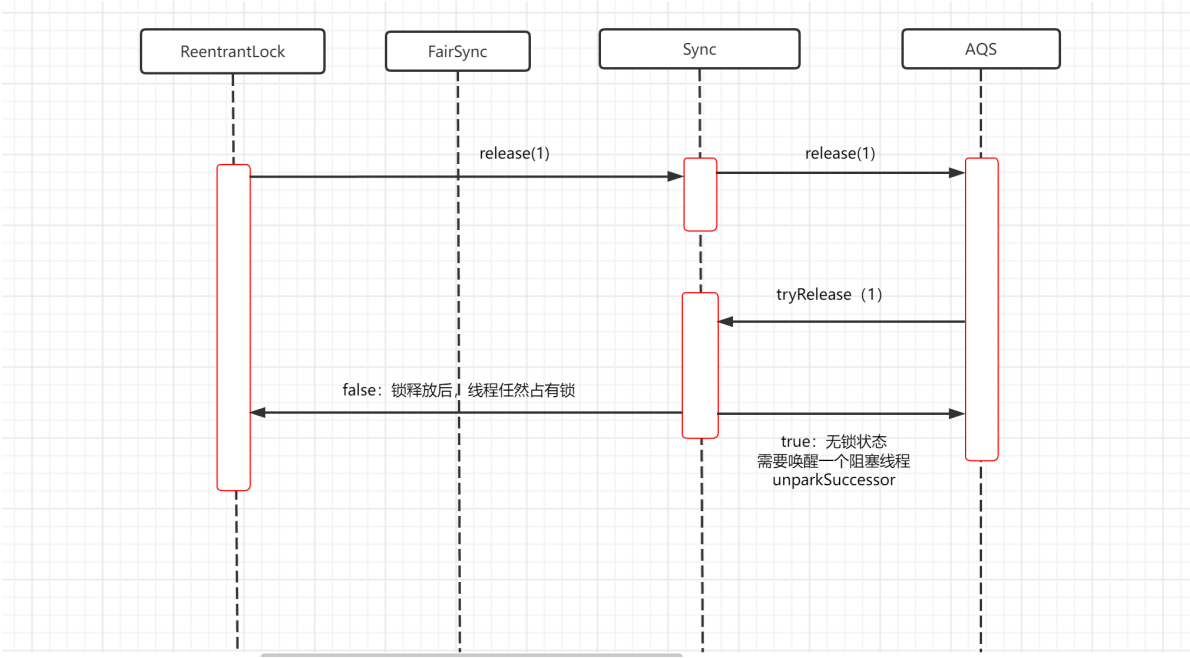
LockSupport类

LockSupport类是Java1.6引入的一个类，所有的方法都是静态方法。它提供了基本的线程同步原语，提供了可以使线程阻塞和唤醒的方法。LockSupport实际上是调用了Unsafe类里的函数，调用了Unsafe的两个函数。

```
//取消阻塞（唤醒）线程
public native void unpark(Object thread);
/**阻塞（挂起）线程。当前线程被阻塞后，当前线程就会被挂起，直到其他线程unpark此线程。
  *isAbsolute是否为绝对时间，true绝对时间，false相对时间。
  *park(false,0)：阻塞线程，直至被唤醒。
  *park(true,time)：暂停当前线程，增加了相对时间的限制，如
  *park(true,time)：暂停当前线程，增加了绝对时间的限制，如2020-12-01 21:00:00的long值
  */
public native void park(boolean isAbsolute, long time);
```

3.7 ReentrantLock源码分析：锁的释放

公平锁的释放，源码调用链路图



第一步：ReentrantLock.unlock

释放锁时，需调用ReentrantLock的unlock方法。这个方法内部，会调用sync.release(1)，release方法为AQS类的final方法。

```
public void unlock() {
    sync.release(1);
}
```

第二步：AQS.release(1)

首先执行方法tryRelease(1)，tryRelease方法为ReentrantLock中Sync类的final方法，用于释放锁。

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {//释放锁。若释放后锁状态为无锁状态，需唤醒后继线程
        Node h = head;//同步队列头节点
        if (h != null && h.waitStatus != 0)//若head不为null,说明链表中有节点。其状态不为0，唤醒后继线程。
            unparkSuccessor(h);
        return true;
    }
    return false;
}

```

第三步：Sync.tryRelease(1)

1. 判断当前线程是否为锁持有者，若不是持有者，不能释放锁，直接抛出异常。
2. 若当前线程是锁的持有者，将重入次数减1，并判断当前线程是否完全释放了锁。
    - 若重入次数为0，则当前新线程完全释放了锁，将锁拥有线程设置为null，并将锁状态置为无锁状态(state=0)，返回true。
    - 若重入次数>0,则当前新线程仍然持有锁，设置重入次数=重入次数-1，返回false。
3. 返回true说明，当前锁被释放，需要唤醒同步队列中的一个线程，执行unparkSuccessor唤醒同步队列中节点线程。

```
/**
 * 释放锁返回值：true释放成功；false释放失败
 */
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;//重入次数减去1
    //如果当前线程不是锁的独占线程，抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        //如果线程将锁完全释放，将锁初始化未无锁状态
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);//修改锁重入次数
    return free;
}

```

第四步：AQS.unparkSuccessor

```
//唤醒后继线程
private void unparkSuccessor(Node node) {
    /*
     * 头节点waitStatus状态 SIGNAL或PROPAGATE
     */
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    //查找需要唤醒的节点:正常情况下，它应该是下一个节点。但是如果下一个节点为null或者它的waitStatus为取消时，则需要从同步队列tail节点向前遍历，查找到队列中首个不是取消状态的节点。
    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    //将下一个节点中的线程unpark唤醒

```

```
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

第五步：LockSupport.unpark(s.thread)

会唤醒挂起的线程，使被阻塞的线程继续执行。

3.8 公平锁和非公平锁源码实现区别

公平锁和非公平锁在获取锁和释放锁时有什么区别呢?

- **非公平锁与非公平锁释放锁是没有差异**，释放锁时调用方法都是AQS的方法。
- **非公平锁与非公平锁获取锁的差异**
  - 我们可以看到上面在公平锁中，线程获得锁的顺序按照请求锁的顺序，按照**先来后到**的规则获取锁。如果线程竞争公平锁失败后，都会到AQS同步队列队尾排队，将自己阻塞等待锁的使用资格，锁被释放后，会从队首开始查找可以获得锁的线程并唤醒。
  - 而非公平锁，允许新线程请求锁时，可以**插队**，新线程先尝试获取锁，如果获取锁失败，才会AQS同步队列队尾排队。

我们对比下两种锁的源码，非公平锁与非公平锁获取锁的差异有两处：

1. lock方法差异：

FairSync.lock：公平锁获取锁

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

```

NoFairSync.lock：非公平锁获取锁，lock方法中新线程会先通过CAS操作compareAndSetState(0, 1)，尝试获得锁。

```
final void lock() {
    if (compareAndSetState(0, 1))//新线程，第一次插队
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

lock方法中的acquire为AQS的final方法，公平锁和非公平锁，执行代码没有差别。差别之处在于公平锁和非公平锁对tryAcquire方法的实现。

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

2. tryAcquire差异

FairSync.tryAcquire：公平锁获取锁，若锁为无锁状态时，本着公平原则，新线程在尝试获得锁前，需先判断AQS同步队列中是否有线程在等待，若有线程在等待，当前线程只能进入同步队列等待。若AQS同步无线程等待，则通过CAS抢占锁。而非公平锁，不管AQS是否有线程在等待，则都会先通过CAS抢占锁。

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //公平锁，先判断同步队列中是否有线程在等待
        if (!hasQueuedPredecessors() && compareAndSetState(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;
}
```

NoFairSync.tryAcquire和NoFairSync.nonfairTryAcquire：

```
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) {
        //非公平锁，入队前，二次插队
        if (compareAndSetState(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;
    }
```

公平锁和非公平锁获取锁时，其他方法都是调用AQS的final方法，所以没有不同之处。

3.9 读写锁ReentrantReadWriteLock

可重入锁ReentrantLock是互斥锁，互斥锁在同一时刻仅有一个线程可以进行访问，但是在大多数场景下，大部分时间都是提供读服务，而写服务占有的时间较少。然而读服务不存在数据竞争问题，如果一个线程在读时禁止其他线程读势必会导致性能降低，所以就出现了读写锁。

读写锁维护着一对锁，一个读锁和一个写锁。通过分离读锁和写锁，使得并发性比一般的互斥锁有了较大的提升：在同一时间可以允许多个读线程同时访问，但是在写线程访问时，所有读线程和写线程都会被阻塞。

读写锁的主要特性：

- **公平性：**支持公平性和非公平性。
- **重入性：**支持重入。读写锁最多支持65535个递归写入锁和65535个递归读取锁。
- **锁降级：**写锁能够降级成为读锁，但读锁不能升级为写锁。遵循获取写锁、获取读锁在释放写锁的次序

读写锁ReentrantReadWriteLock实现接口ReadWriteLock，该接口维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

ReadWriteLock定义了两个方法。readLock()返回用于读操作的锁，writeLock()返回用于写操作的锁。ReentrantReadWriteLock定义如下：

```
/** 内部类  读锁 */
private final ReentrantReadWriteLock.ReadLock readerLock;
/** 内部类  写锁 */
private final ReentrantReadWriteLock.WriteLock writerLock;

final Sync sync;

/** 使用默认（非公平）的排序属性创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock() {
    this(false);
}

/** 使用给定的公平策略创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
```

```
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}

/** 返回用于写入操作的锁 */
public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
/** 返回用于读取操作的锁 */
public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }

abstract static class Sync extends AbstractQueuedSynchronizer {
    //省略其余源代码
}
public static class WriteLock implements Lock, java.io.Serializable{
    //省略其余源代码
}

public static class ReadLock implements Lock, java.io.Serializable {
    //省略其余源代码
}
```

ReentrantReadWriteLock与ReentrantLock一样，其锁主体依然是Sync，它的读锁、写锁都是依靠Sync来实现的。所以ReentrantReadWriteLock实际上只有一个锁，只是在获取读取锁和写入锁的方式上不一样而已，它的读写锁其实就是两个类：ReadLock、writeLock，这两个类都是lock的实现。

案例

```
package com.hero.multithreading;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Demo10ReentrantReadWriteLock {
    private static volatile int count = 0;

    public static void main(String[] args) {
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        WriteDemo writeDemo = new WriteDemo(lock);
        ReadDemo readDemo = new ReadDemo(lock);

        for (int i = 0; i < 3; i++) {
            new Thread(writeDemo).start();
        }
        for (int i = 0; i < 5; i++) {
            new Thread(readDemo).start();
        }
    }

    static class WriteDemo implements Runnable {
        ReentrantReadWriteLock lock;
        public WriteDemo(ReentrantReadWriteLock lock) {
            this.lock = lock;
        }

```

```
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                lock.writeLock().lock();
                count++;
                System.out.println("写锁: "+count);
                lock.writeLock().unlock();
            }
        }
    }

    static class ReadDemo implements Runnable {
        ReentrantReadWriteLock lock;
        public ReadDemo(ReentrantReadWriteLock lock) {
            this.lock = lock;
        }

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                lock.readLock().lock();
                System.out.println("读锁: "+count);
                lock.readLock().unlock();
            }
        }
    }
}

```

```
C:\develop\java\jdk1.8.0_171\bin\java.exe ...
读锁：0
读锁：0
读锁：0
读锁：0
写锁：1
写锁：2
写锁：3
读锁：3
写锁：4
读锁：4
```

3.10 锁优化

减少锁持有时间

```
public synchronized void syncMethodBefore(){
    otherCode1();
    mutextMethod();
    otherCode2();
}
```

```
public void syncMethodAfter(){
    otherCode1();
    synchronized (this){
        mutextMethod();
    }
    otherCode2();
}
```

减少锁粒度

- 将大对象拆分成小对象，增加并行度，降低锁竞争。
- ConcurrentHashMap允许多个线程同时进入

锁分离

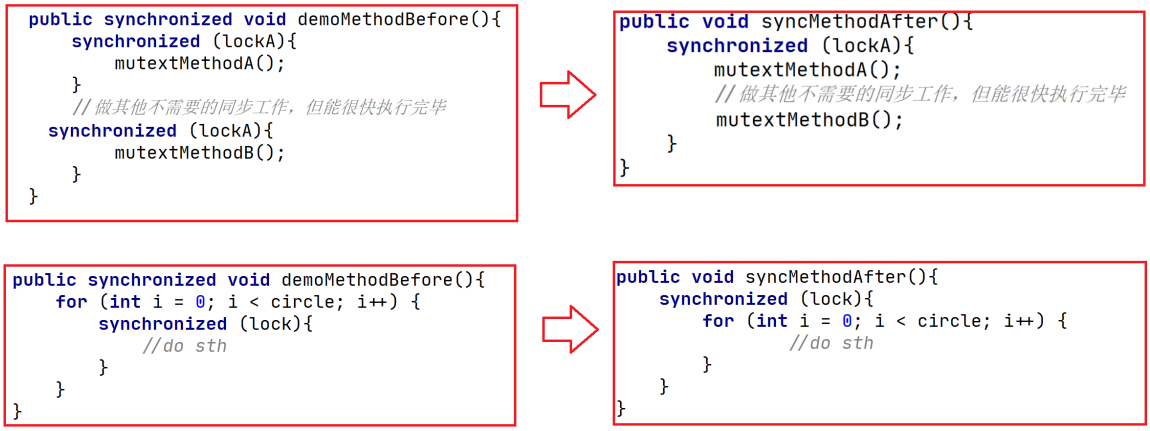
- 根据功能进行锁分离
- ReadWriteLock在读多写少时，可以提高性能。

锁消除

- 锁消除是发生在编译器级别的一种锁优化方式。
- 有时候我们写的代码完全不需要加锁，却执行了加锁操作。

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可能短，但是在某些情况下，一个程序对同一个锁不间断、高频地请求、同步与释放，会消耗掉一定的系统资源，因为锁的请求、同步与释放本身会带来性能损耗，这样高频的锁请求就反而不利于系统性能的优化了，虽然单次同步操作的时间可能很短。锁粗化就是告诉我们任何事情都有个度，有些情况下我们反而希望把很多次锁的请求合并成一个请求，以降低短时间内大量锁请求、同步、释放带来的性能损耗。



上面我们讲解了线程之间的互斥，接下来我们看一下线程之间如何进行合作

4. 线程协作工具类

线程协作工具类就是帮助程序员更容易的让线程之间进行协作，来完成某个业务功能。

类	作用	说明
Semaphore	信号量，通过控制 **许可** 的数量来保证线程之间的配合	**场景：限流**，只有拿到**许可**才可运行
CyclicBarrier	线程会等待，直到线程到了事先规定的数目，然后触发执行条件进行下一步动作	**场景：并行计算**（线程之间相互等待处理结果就绪的场景）
CountDownLatch	线程处于等待状态，指导计数减为0，等待线程才继续执行	适用场景：购物拼团
Condition	控制线程的**等待/唤醒**	Object.wait()和notify()的升级版

4.1 CountDownLatch倒数门闩

倒数结束之前，一直处于等待状态，直到数到0结束，此线程才继续工作。

场景：购物拼团，大巴人满发车，分布式锁

主要方法：

- 构造函数：CountDownLatch(int count)：只有一个构造函数，参数count为需要倒数的数值。
- await()：当一个或多个线程调用await()时，这些线程会阻塞。
- countDown()：其他线程调用countDown()会将计数器减1，调用countDown方法的线程不会阻塞。当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行

用法：

一个线程等待多个线程都执行完，再继续自己的工作。

```
package com.hero.multithreading;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
/**
 * CountDownLatch案例：6个程序猿加班
 * 当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行
 */
public class Demo11CountDownLatch {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(6);

        for (int i = 1; i <= 6; i++) {
            new Thread(()->{
                try { TimeUnit.SECONDS.sleep(5); } catch
(InterruptedException e) {e.printStackTrace(); }
                System.out.println(Thread.currentThread().getName() + "\t上
完班，离开公司");
                countDownLatch.countDown();
            }, String.valueOf(i)).start();
        }
        new Thread(()->{
            try {
                countDownLatch.await();//卷王也是有极限的，设置超时时间
                System.out.println(Thread.currentThread().getName()+"\t卷王最
后关灯走人");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "7").start();
    }
}
```

4.2 Semaphore信号量

用来限制或管理数量有限资源的使用情况。

信号量的作用就是维护一个"许可证"的计数，线程可以"获取"许可证，那信号量剩余的许可证就减少一个，线程也可以"释放"一个许可证，那信号量剩余的许可证就可以加一个。当信号量拥有的许可证数为0时，下一个还要要获取许可证的线程就需要等待，直到有另外的线程释放了许可证。

主要方法：

- 构造函数：Semaphore(int permits,Boolean fair)：可以设置是否使用公平策略，如果传入true,则Semaphore会把之前等待的线程放到FIFO队列里，以便有了新许可证可以分给之前等待时间最长的线程。
- acquire()：获取许可证，当一个线程调用acquire操作时，他要么通过成功获取信号量（信号量减1），要么一直等待下去，直到有线程释放信号量，或超时。
- release()：释放许可证，会将信号量加1，然后唤醒等待的线程。

```
1  package com.hero.multithreading;
2
3  import java.util.concurrent.Semaphore;
4  import java.util.concurrent.TimeUnit;
5
6  /**
7   * Semaphore案例：三辆小汽车抢车位
8   * Semaphore信号量主要作用：1.用于多个共享资源的互斥使用，2.用于并发线程数的控制
9   */
10 public class Demo12Semaphore {
11     public static void main(String[] args) {
12         //模拟资源类，有3个空车位
13         Semaphore semaphore = new Semaphore(3);
14
15         for (int i = 1; i <= 6; i++) {
16             new Thread(()->{
17                 try{
18                     //占有资源
19                     semaphore.acquire();
20                     System.out.println(Thread.currentThread().getName()+"\t
   抢到车位");
21
22                     try { TimeUnit.SECONDS.sleep(3); } catch
   (InterruptedException e) {e.printStackTrace(); }
23
24                     System.out.println(Thread.currentThread().getName()+"\t
   停车3秒后离开车位");
25                 } catch (Exception e) {
26                     e.printStackTrace();
27                 } finally {
28                     //释放资源
29                     semaphore.release();
30                 }
31             }, "Thread-Car-"+String.valueOf(i)).start();
32         }
33     }
34 }
35
```

4.3 CyclicBarrier循环栅栏

线程会等待，直到线程到了事先规定的数目，然后触发执行条件进行下一步动作

当有大量线程互相配合，分别计算不同任务，并且需要最后统一汇总时，就可以用CyclicBarrier，它可以构造一个集结点，当某一个线程执行完，它就会到集结点等待，直到所有线程都到集结点，则该栅栏就被撤销，所有线程统一出再，继续执行剩下的任务。

主要方法：

- 构造函数：CyclicBarrier(int parties, Runnable barrierAction)，设置聚集的线程数量和集齐线程数的结果之后要执行的动作。
- await()：阻塞当前线程，待凑齐线程数量之后继续执行

```
package com.hero.multithreading;

import java.util.concurrent.CyclicBarrier;

/**
 * 案例：集齐7龙珠召唤神龙
 */
public class Demo13CyclicBarrier {
    public static void main(String[] args) {

        CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
            System.out.println("======召唤神龙");
        });

        for (int i = 1; i <= 7; i++) {
            final int tempInt = i;
            new Thread(()->{
                try {
                    System.out.println(Thread.currentThread().getName() +
"\t收集到第" + tempInt + "颗龙珠");
                    cyclicBarrier.await();
                    System.out.println(Thread.currentThread().getName() +
"\t第" + tempInt + "颗龙珠飞走了");
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }, "Thread-"+String.valueOf(i)).start();
        }
    }
}

```

CyclicBarrier和CountDownLatch区别：

- 作用不同：CyclicBarrier要等固定数量的线程都到达了栅栏位置才能继续执行，而CountDownLatch只需要等待数字到0，也就是说，CountDownLatch用于事件，而CyclicBarrier用于线程。

- 可重用性不同：CountDownLatch在倒数到0并触发门闩打开后，就不能再次使用，而CyclicBarrier可以重复使用。

4.4 Condition接口（条件对象）

当线程1需要等待某个条件时就去执行condition.await()方法，一旦执行await()方法，线程就会进入阻塞状态。通常会有另一个线程2去执行对应条件，直到这个条件达成时，线程2就会执行condition.signal()方法，此时JVM就会从被阻塞的线程里找到那些等待该condition的线程，当线程1收到可执行信号时，它的线程状态就会变成Runnable可执行状态。

- signalAll()会唤起所有正在等待的线程。
- signal()是公平的，只会唤起那个等待时间最长的线程。

注意点：

- Condition用来代替Object.wait/notify两者用法一样
- Condition的await()会自动释放持有的Lock锁这点也和Object.wait一样
- 调用await时必须持有锁，否则会抛出异常。

```
package com.hero.multithreading;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 案例：Tony仨小哥洗剪吹
 * 演示多线程之间按顺序调用，实现A->B->C
 * 三个线程Tony要求如下：
 *   tony雄雄-洗头，tony超超-理发，tony麦麦-吹干
 *   。。。
 *   tony雄雄-洗头，tony超超-理发，tony麦麦-吹干
 *  依次来10轮
 */
public class Demo14ConditionDemo {
    public static void main(String[] args) {
        ShareData shareData = new ShareData();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                shareData.wash();
            }
        }, "tony-雄雄").start();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                shareData.cut();
            }
        }, "tony-超超").start();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                shareData.cook();
            }
```

```
        }, "tony-麦麦").start();

    }
}
class ShareData {
    private volatile int number = 1; //tony-雄雄:1, tony-超超:2, tony-麦麦:3

    private Lock lock = new ReentrantLock();
    private Condition c1 = lock.newCondition(); //number == 1
    private Condition c2 = lock.newCondition(); //number == 2
    private Condition c3 = lock.newCondition(); //number == 3

    /**
     * A线程每一轮要执行的操作
     */
    public void wash() {
        lock.lock();
        try{
            //判断
            while(number != 1){
                c1.await();
            }
            //模拟线程执行的任务
            System.out.println(Thread.currentThread().getName()+"-洗头");
            //通知
            number = 2;
            c2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    /**
     * B线程每一轮要执行的操作
     */
    public void cut() {
        lock.lock();
        try{
            //判断
            while(number != 2){
                c2.await();
            }
            //模拟线程执行的任务
            System.out.println(Thread.currentThread().getName()+"-理发");
            //通知
            number = 3;
            c3.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
```

```
91
92      public void cook() {
93          lock.lock();
94          try{
95              //判断
96              while(number != 3){
97                  c3.await();
98              }
99              //模拟线程执行的任务
100             System.out.println(Thread.currentThread().getName()+"-吹干");
101             //通知
102             number = 1;
103             c1.signal();
104         } catch (Exception e) {
105             e.printStackTrace();
106         } finally {
107             lock.unlock();
108         }
109     }
110 }
```

5. 并发容器

5.1 什么是并发容器?

在JUC包中，有一大部分是关于并发容器的，如ConcurrentHashMap，ConcurrentSkipListMap，CopyOnWriteArrayList及阻塞队列。这里将介绍使用频率、面试中出现频繁的最高的ConcurrentHashMap和阻塞队列。

> 注意：这里说到的容器概念，相当于我们理解中的集合的概念。

同步容器:

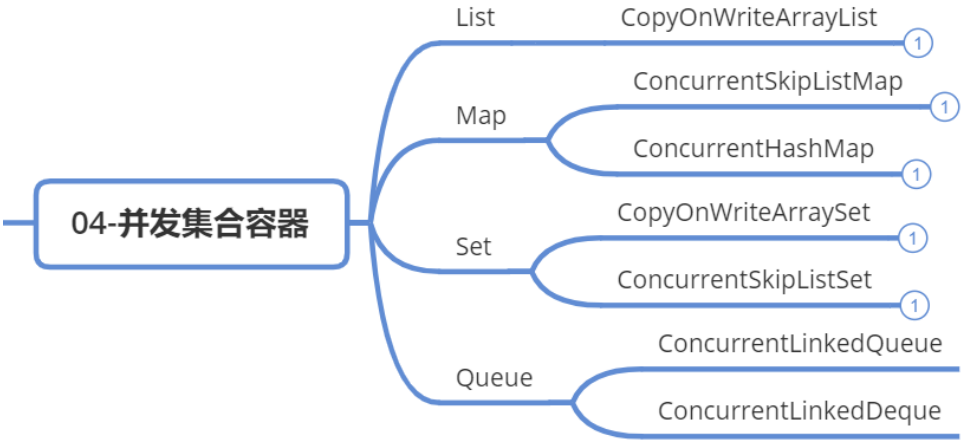
Java中的集合主要分为四大类：List、Map、Set和Queue，但是并不是所有集合都是线程安全的。比如，我们经常使用的ArrayList，HashMap，HashSet就不是线程安全的。

早期的JDK1.0中的就提供了线程安全的集合，包括Vector，Stack和Hashtable。此外还有在JDK1.2中增加的Collections中内部SynchronizedXxx类，它们也是线程安全的集合，可以由对应Collections.synchronizedXxx工厂方法创建。这些类实现线程安全的方式都是一样的：都是基于synchronized这个同步关键字实现的，对每个公有方法都进行了同步，保证每次只有一个线程能访问集合，所以它们被称为线程安全的集合（同步容器）。

并发容器:

在JDK1.5之前，JDK提供的线程安全的类都是同步集合容器。同步容器都是线程安全的，但是所有线程对容器只能串行访问，性能很差。在JDK1.5之后引入的JUC并发包，提供的更多类型的并发容器，在性能上做了很多改进优化，可以用来替代同步容器。它们都是针对多线程并发访问来进行设计的，我们称它们为并发容器。

并发容器依然可以归属到我们提到的四大类：List、Map、Set 和 Queue。



这里我总结了一下它们特性和使用场景：

1. List容器：
    - Vector：使用synchronized同步锁，数据具有强一致性。适合于对数据有强一致性要求的场景，但性能较差。
    - **CopyOnWriteArrayList**：底层使用数组存储数据，使用复制副本实现有锁写操作，不能保证强一致性。适合于读多写少，允许读写数据短暂不一致的高并发场景。
2. Map容器
    - Hashtable：使用synchronized同步锁，数据具有强一致性。适合于对数据有强一致性要求的场景，但性能较差。
    - **ConcurrentHashMap**：基于数组+链表+红黑树实现，写操作时通过synchronized同步锁将HashEntry作为锁的粒度支持一定程度的并发写，具有弱一致性。适合于存储数据量较小，读多写少且不要求强一致性的高并发场景。
    - ConcurrentSkipListMap：基于跳表实现的有序Map，使用CAS实现无锁化读写，具有弱一致性。适合于存储数据量大，读写都比较频繁，对数据不要求强一致性的高并发场景。
3. Set容器
    - CopyOnWriteArraySet：底层使用数组存储数据，使用复制副本实现有锁写操作，不能保证强一致性。适合于读多写少，允许读写数据短暂不一致的场景。
    - ConcurrentSkipListSet：基于跳表实现的有序Set，使用CAS实现无锁化读写，具有弱一致性。适合于存储数据量大，读写都比较频繁，对数据不要求强一致性的高并发场景。

5.2 ConcurrentHashMap

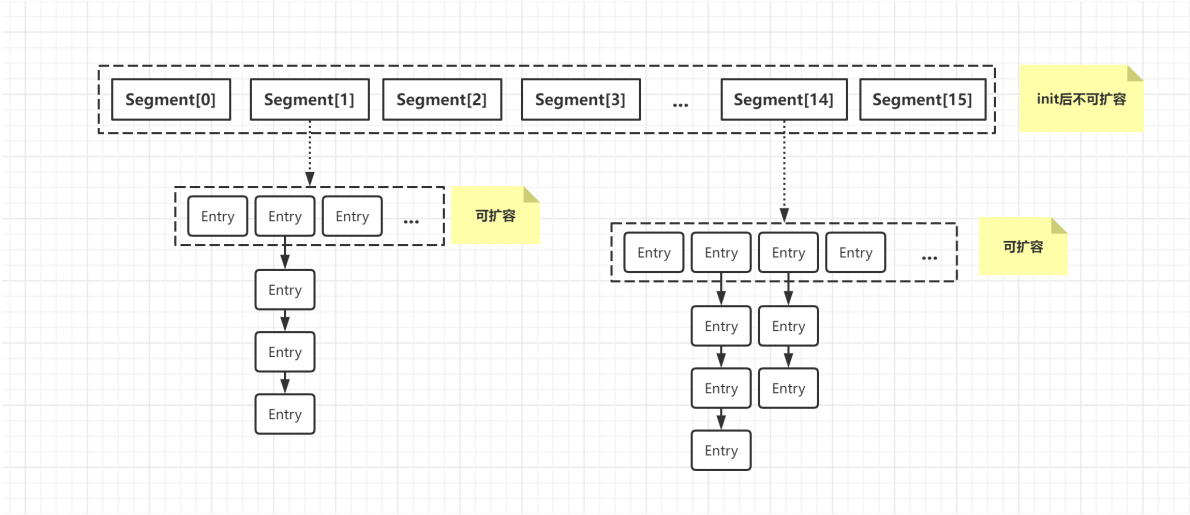
结构图

JDK1.7结构图

Java7中的ConcurrentHashMap最外层是多个segment，每个segment的底层数据结构与HashMap类似，仍然是数组和链表组成。

每个segment独立上ReentrantLock锁，每个segment之间互不影响，提高并发效率。

默认有16个segment，最多可以同时支持16个线程并发写（操作分别分布在不同的Segment上）。这个默认值可以在初始化时设置，但一旦初始化以后，就不可以再扩容了。

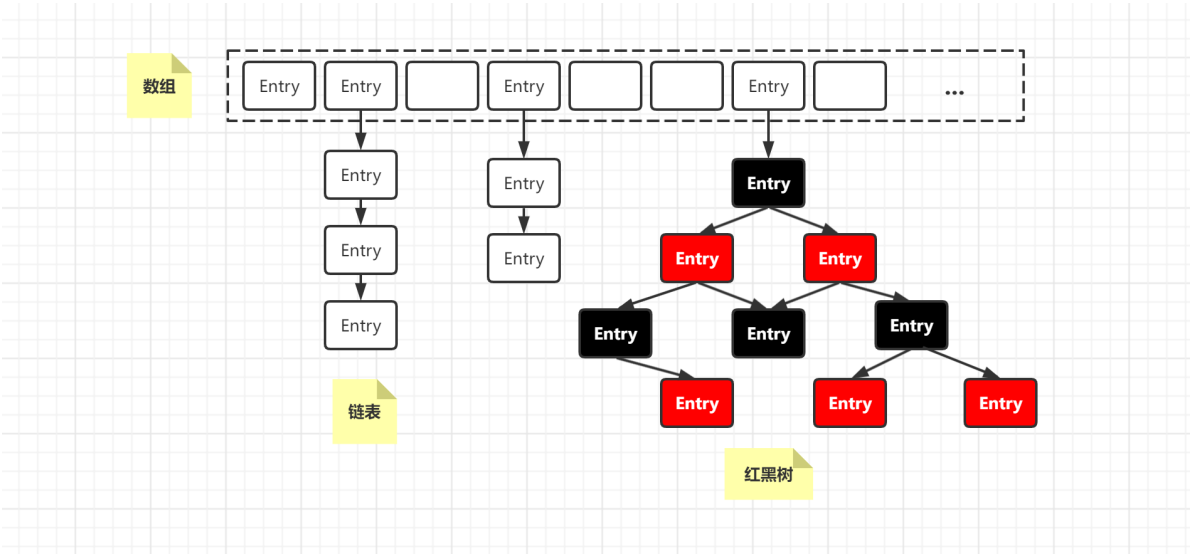


JDK1.8结构图

ConcurrentHashMap是一个存储 key/value 对的容器，并且是线程安全的。

改进一： 取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

改进二： 将原先table数组 + 单向链表的数据结构，变更为table数组 + 单向链表 + 红黑树的结构。查询更快



这是经典的数组加链表的形式。 并且在链表长度过长时转化为红黑树存储（ Java 8 的优化） ， 加快查找速度。

小结：

- ConcurrentHashMap 采用数组 + 链表 + 红黑树的存储结构
- 存入的Key值通过自己的 hashCode 映射到数组的相应位置
- ConcurrentHashMap 为保障查询效率， 在特定的时候会对数据增加长度【扩容】
- 当链表长度增加到 8 时， 可能会触发链表转为红黑树

5.3 CopyOnWriteArrayList

实现原理

CopyOnWrite 思想：是平时查询的时候，都不需要加锁，随便访问，只有在更新的时候，才会从原来的数据复制一个副本出来，然后修改这个副本，最后把原数据替换成当前的副本。修改操作的同时，读操作不会被阻塞，而是继续读取旧的数据。这点要跟读写锁区分一下。

```
package com.hero.multithreading;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Demo15CopyOnWriteArrayList {

    public static void main(String[] args) {
        //1、初始化CopyOnWriteArrayList
        List<Integer> tempList = Arrays.asList(new Integer [] {1,2});
        CopyOnWriteArrayList<Integer> copyList = new CopyOnWriteArrayList<>
(tempList);


        //2、模拟多线程对list进行读和写
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("copyList size:"+copyList.size());
        executorService.shutdown();

    }
}

class ReadThread implements Runnable {
    private List<Integer> list;

    public ReadThread(List<Integer> list) {
        this.list = list;
    }

```

```
    @Override
    public void run() {
        System.out.print("size:="+list.size()+",::");
        for (Integer ele : list) {
            System.out.print(ele + ",");
        }
        System.out.println();
    }
}

class WriteThread implements Runnable {
    private List<Integer> list;

    public WriteThread(List<Integer> list) {
        this.list = list;
    }

    @Override
    public void run() {
        this.list.add(9);
    }
}
```

优缺点

优点

- 对于一些读多写少的数据，写入时复制的做法就很不错，例如：配置、黑名单、物流地址等变化非常少的数据，这是一种无锁的实现。可以帮我们实现程序更高的并发。
- CopyOnWriteArrayList 并发安全且性能比 Vector 好。Vector 是增删改查方法都加了 synchronized 来保证同步，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而 CopyOnWriteArrayList 只是在增删改上加锁，但是读不加锁，在读方面的性能就好于 Vector。

缺点

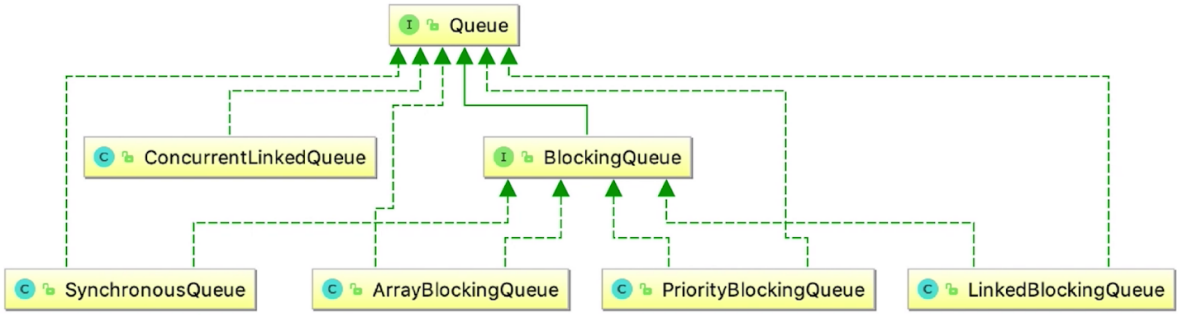
- 数据一致性问题。这种实现只是保证数据的最终一致性，在添加到拷贝数据而还没进行替换的时候，读到的仍然是旧数据。
- 内存占用问题。如果对象比较大，频繁地进行替换会消耗内存，从而引发 Java 的 GC 问题，这个时候，我们应该考虑其他的容器，例如 ConcurrentHashMap。

5.4 并发队列

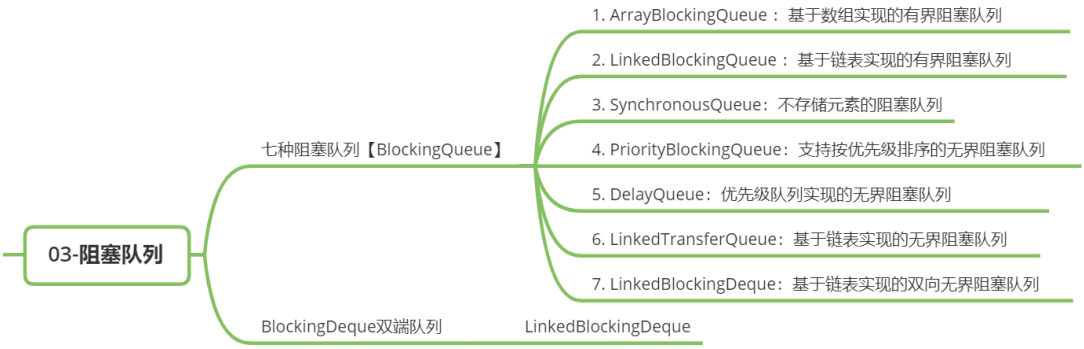
为什么要用队列

- 通过队列可以很容易的实现数据共享，并且解决上下游处理速度不匹配的问题，典型的**生产者消费者模式**
- 队列中的读写等线程安全问题由队列负责处理。

常用并发队列



JUC提供了7种适合与不同应用场景的阻塞队列。



1. **ArrayBlockingQueue： 基于数组实现的有界阻塞队列**
2. **LinkedBlockingQueue： 基于链表实现的有界阻塞队列**
3. **SynchronousQueue：不存储元素的阻塞队列**
4. **PriorityBlockingQueue**：支持按优先级排序的无界阻塞队列
5. **DelayQueue**：优先级队列实现的无界阻塞队列
6. **LinkedTransferQueue**：基于链表实现的无界阻塞队列
7. **LinkedBlockingDeque**：基于链表实现的双向无界阻塞队列

阻塞队列

- 阻塞队列的一端是给生产者放数据用，另一端给消费者拿数据用。阻塞队列是线程安全的，所以生产者和消费者都可以是多线程的。
- take()方法获取并移除队列的头结点，一旦执行take时，队列里无数据则阻塞，直到队列里有数据。
- put()方法是插入元素，但是如何队列已满，则无法继续插入，则阻塞，直到队列中有空闲空间。
- 是否有界（容量多大），这是非常重要的属性，无界队列Integer.MAX_VALUE，认为是无限容量。

ArrayBlockingQueue

有界，可以指定容量

公平：可以指定是否需要保证公平，如果想要保证公平，则等待最长时间的线程会被优先处理，不过会带来一定的性能损耗。

场景：有10个面试者，只有1个面试官，大厅有3个位子让面试者休息，每个人面试时间10秒，模拟所有人面试的场景。

```java
package com.hero.multithreading;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;

/**
 * 案例：有10个面试者，只有1个面试官，大厅有3个位子让面试者休息，每个人面试时间10秒，模拟所有人面试的场景。
 */
public class Demo16ArrayBlockingQueue {

    static ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(3);

    public static void main(String[] args) {


        Interviewer r1 = new Interviewer(queue);//面试官
        Engineers e2 = new Engineers(queue);//程序员们
        new Thread(r1).start();
        new Thread(e2).start();
    }
}

class Interviewer implements Runnable {

    BlockingQueue<String> queue;

    public Interviewer(BlockingQueue queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        System.out.println("面试官：我准备好了，可以开始面试");
        String msg;
        try {
            while(!(msg = queue.take()).equals("stop")){
                System.out.println(msg + " 面试+开始...");
                TimeUnit.SECONDS.sleep(10);//面试10s
                System.out.println(msg + " 面试-结束...");
            }
            System.out.println("所有候选人都结束了");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Engineers implements Runnable {

    BlockingQueue<String> queue;

    public Engineers(BlockingQueue queue) {
```

```
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            String candidate = "程序员" + i;
            try {
                queue.put(candidate);
                System.out.println(candidate+" 就坐=等待面试~");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        try {
            queue.put("stop");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}

```

6. 线程池

6.1 线程池介绍

线程池（ThreadPool） 是一种基于池化思想管理线程的工具，看过new Thread源码之后我们发现，频繁创建线程销毁线程的开销很大，会降低系统整体性能。线程池维护多个线程，等待监督和管理分配可并发执行的任务。

优点:

- **降低资源消耗**：通过线程池复用线程，降低创建线程和释放线程的损耗
- **提高响应速度**：任务到达时，无需等待即刻运行
- **提高线程的可管理性**：使用线程池可以进行统一的分片、调优和监控线程
- **提供可扩展性**：线程池具备可扩展性，研发人员可以向其中增加各种功能，比如延时，定时，监控等

适用场景:

- **连接池**：预先申请数据库连接，提升申请连接的速度，降低系统的开销
- **快速响应用户请求**：服务器接受到大量请求时，使用线程池是很适合的，它可以大大减少线程的创建和销毁的次数，提高服务器的工作效率。
- 在实际开发中，如果需要创建5个以上的线程，就可以用线程池来管理。

6.2 线程池参数

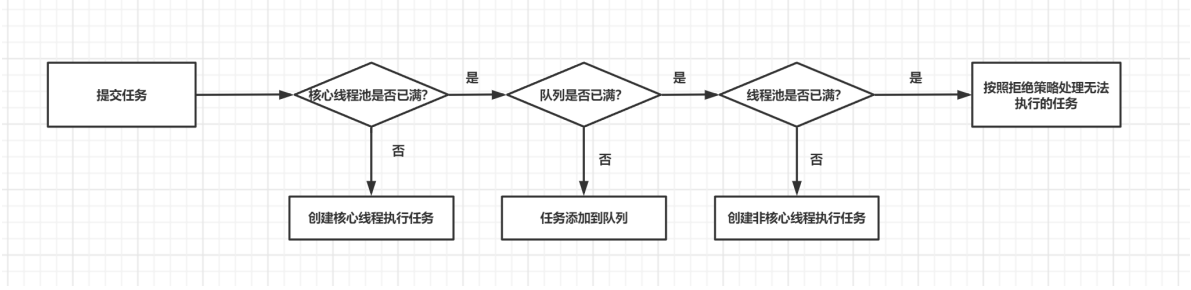
参数名	类型	含义
corePoolSize	int	核心线程数
maxPoolSize	int	最大线程数
keepAliveTime	long	保持存活时间
workQueue	BlockingQueue	任务存储队列
threadFactory	ThreadFactory	线程池创建新线程的线程工厂类
Handler	RejectedExecutionHandler	线程无法接收任务时的拒绝策略

参数详解：

1. corePoolSize核心线程数：即便线程空闲（Idle）也不会回收
2. maxPoolSize最大线程数：线程池可能会在核心线程数的基础上，额外增加一些线程，但是这些新增加的线程数有一个上限，就是maxPoolSize
3. ThreadFactory新的线程是由ThreadFactory创建，默认使用Executors.defaultThreadFactory()，创建出来的线程都在同一个线程组，拥有同样的NORM_PRIORITY优先级并且都不是守护线程。如果自己指定ThreadFactory,则可以改变线程名、线程组、优先级、是否是守护线程
4. workQueue工作队列类型
    - 直接交换：SynchronousQueue，这个队列没有容量，无法保存工作任务。
    - 无界队列：LinkedBlockingQueue无界队列
    - 有界队列：ArrayBlockingQueue有界队列

6.3 线程池原理

- 如果线程数小于corePoolSize，即使其它工作线程处于空闲状态，也会创建一个新线程来运行新任务。
- 如果线程数大于等于corePoolSize，但少于maxPoolSize，将任务放入队列。
- 如果队列已满，并且线程数小于maxPoolSize，则创建一个新线程来运行任务。
- 如果队列已满，并且线程数大于或等于maxPoolSize，则拒绝该任务。



例子：核心池大小为5，最大池大小为10，队列为100

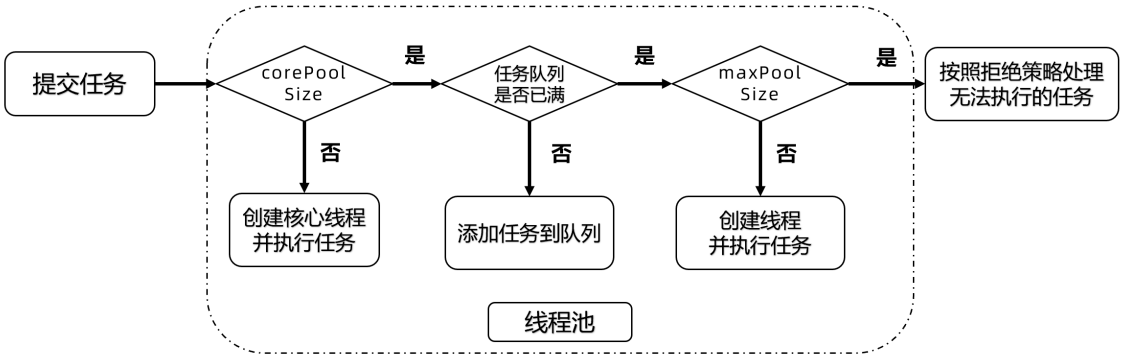
因为线程中的请求最多会创建5个，然后任务将被添加到队列中，直到达到100。

当队列已满时，将创建最新的线程maxPoolSize，最多到10个线程，如果再来任务，则拒绝。

增减线程的特点

- **固定大小线程池**：通过设置corePoolSize和maxPoolSize相同，可以创建固定大小的线程池。
- **动态线程池**：线程池希望保持较少的线程数，并且只有在负载变得很大时才会增加。可以设置corePoolSize比maxPoolSize大一些
- 通过设置maxPoolSize为很高的值，例如Integer.MAX_VALUE，可以允许线程池容纳任意数量的并发任务。
- 只有在队列填满时才创建多于corePoolSize的线程，所以如果用的是无界队列（LinkedBlockingQueue），则线程数就一直不会超过corePoolSize

6.4 自动创建线程池



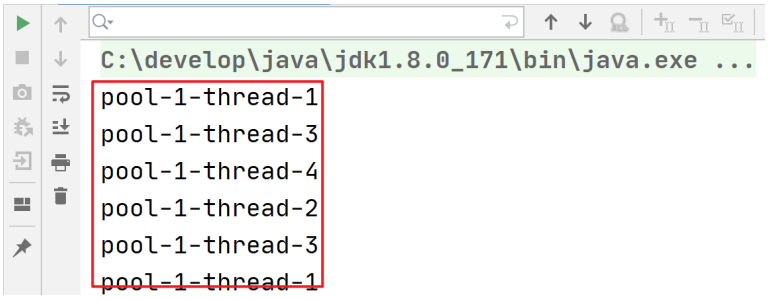
newFixedThreadPool

手动创建更好，因为这样可以更明确线程池的运行规则，避免资源耗尽的风险。

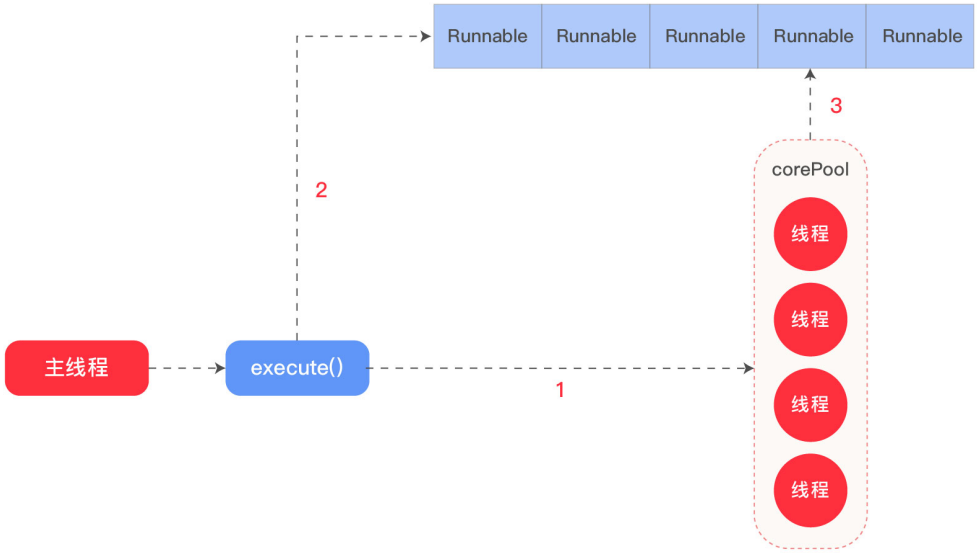
```java
package com.hero.multithreading;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 案例：演示newFixedThreadPool
 */
public class Demo17FixedThreadPool {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 1000; i++) {
            executorService.execute(new Task());
        }
    }
}

class Task implements Runnable {

    @Override
    public void run() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName());
```

```
    }
}

```

不管有多少任务，始终只有4个线程在执行。



源码解读




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

- 参数01是核心线程数量
- 参数02是最大线程数量，它设置的与核心线程数量一样，不会有超过核心线程数量的线程出现，所以第三个参数存活时间设置为0，
- 参数04是**无界队列**，有再多的任务也都会放在队列中，不会创建新的线程。

如果线程处理任务的速度慢，越来越多的任务就会放在无界队列中，会占用大量内存，这样就会导致内存溢出（OOM）的错误。

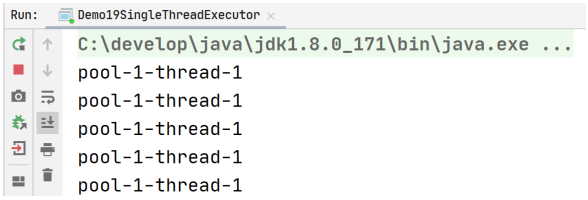
```java
package com.hero.multithreading;
```

```
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  /**
7   * 案例：演示newFixedThreadPool出错的情况
8   * -Xmx8m -Xms8m
9   */
10 public class Demo18FixedThreadPoolOOM {
11
12     private static ExecutorService executorService =
   Executors.newFixedThreadPool(1);
13
14     public static void main(String[] args) {
15         for (int i = 0; i < Integer.MAX_VALUE; i++) {
16             executorService.execute(new SubThread());
17         }
18     }
19
20 }
21
22 class SubThread implements Runnable {
23     @Override
24     public void run() {
25         try {
26             //处理越慢越好，演示内存溢出
27             Thread.sleep(1000000000);
28         } catch (InterruptedException e) {
29             e.printStackTrace();
30         }
31     }
32 }
33
```

newSingleThreadExecutor

只有一个线程

```
1  package com.hero.multithreading;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class Demo19SingleThreadExecutor {
7      public static void main(String[] args) {
8          ExecutorService executorService =
   Executors.newSingleThreadExecutor();
9          for (int i = 0; i < 1000; i++) {
10             executorService.execute(new Task());
11         }
12     }
13 }
14
```

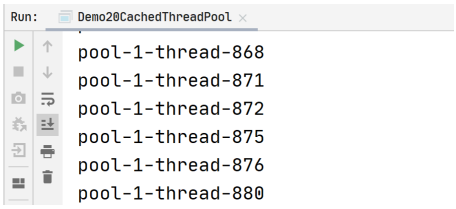


它和newFixedThreadPool的原理相同，只不过把线程数直接设置为1，当请求堆积时，也会占用大量内存。

newCachedThreadPool

可缓存线程池，它是无界线程池，并具有自动回收多余线程的功能。

```
package com.hero.multithreading;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 案例：演示CachedThreadPool
 */
public class Demo20CachedThreadPool {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 1000; i++) {
            executorService.execute(new Task());
        }
    }
}

```



源码解读

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

- 参数01是核心线程数，为0
- 参数02线程池的最大线程数量是整数的最大值，可以认为没有上限，有多个任务过来就创建多个线程去执行。
- 参数03一定时间后（默认60秒），会将多余的线程进行回收。
- 参数04SynchronousQueue是直接交换队列，容量为0，所以不能将任务放在队列中。任务过来后直接交给线程去执行。

弊端：第二个参数maxPoolSize设置为Integer.MAX_VALUE，可能会创建非常多的线程，导致OOM

newScheduledThreadPool

支持定时及周期性任务执行的线程池

```
package com.hero.multithreading;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
/**
 * 案例：演示newScheduledThreadPool
 */
public class Demo21ScheduledThreadPool {

    public static void main(String[] args) {
        ScheduledExecutorService threadPool =
Executors.newScheduledThreadPool(10);
        //延迟5秒运行
        //threadPool.schedule(new Task(), 5, TimeUnit.SECONDS);
        //先延迟1秒运行，然后每隔3秒运行一次
        threadPool.scheduleAtFixedRate(new Task(), 1, 3, TimeUnit.SECONDS);
    }
}

```

6.5 手动创建线程池

部分企业的开发规范中会禁止使用快捷线程池，要求通过标准构造器 ThreadPoolExecutor 去构造工作线程池。

```
// 使用标准构造器，构造一个普通的线程池
public ThreadPoolExecutor(
    int corePoolSize, // 核心线程数，即使线程空闲（Idle），也不会回收；
    int maximumPoolSize, // 线程数的上限；
    long keepAliveTime, TimeUnit unit, // 线程最大空闲（Idle）时长
    BlockingQueue workQueue, // 任务的排队队列
    ThreadFactory threadFactory, // 新线程的产生方式
    RejectedExecutionHandler handler) // 拒绝策略
```

根据不同的业务场景，自己设置线程池的参数、线程名、任务被拒绝后如何记录日志等。

设置线程池数量

- CPU密集型线程池：CPU 密集型任务也叫计算密集型任务，其特点是要进行大量计算而需要消耗 CPU 资源，比如计算圆周率、对视频进行高清解码等等。 CPU 密集型任务虽然也可以并行完成，但是并行的任务越多，花在任务切换的时间就越多， CPU 执行任务的效率就越低，所以，要最高效地利用 CPU，CPU 密集型任务的并行执行的数量应当等于 CPU 的核心数。

- IO密集型线程池：由于 IO 密集型任务的 CPU 使用率较低，导致线程空余时间很多，所以通常就需要开 CPU核心数两倍的线程。当 IO 线程空闲时，可以启用其他线程继续使用 CPU，以提高 CPU 的使用率。

拒绝策略

- **拒绝时机**
  - 当Executor关闭时，提交新任务会被拒绝。
  - 当Executor对最大线程和工作队列容量使用有限边界并且已经饱和时
- **AbortPolicy**：直接抛出异常，说明任务没有提交成功
- **DiscardPolicy**：线程池会默默的丢弃任务，不会发出通知
- **DiscardOldestPolicy**：队列中存有很多任务，将队列中存在时间最久的任务给丢弃。
- **CallerRunsPolicy**：当线程池无法处理任务时，那个线程提交任务由那个线程负责运行。好处在于避免丢弃任务和降低提交任务的速度，给线程池一个缓冲时间。

6.6 线程池案例：手写网站服务器案例

需求：模拟基于Http协议的网站服务器，使用浏览器访问自己编写的服务端程序。然后压测看一看

案例分析:

- 准备测试页面及图片，存放在web文件夹
- 模拟服务器端（ServerSocket）使用浏览器访问，查看页面效果
- 本案例涉及并发编程与网络编程，我们先来观察并发编程部分

```
/**
 * 线程池版本
 */
public class BsServer03 {
    public static void main(String[] args) throws IOException {
        System.out.println("服务器 启动.....  ");
        System.out.println("开启端口 : 9999.....  ");
        // 创建服务端ServerSocket
        ServerSocket serverSocket = new ServerSocket(9999);
        //创建10个线程的线程池
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        while (true) {
            Socket accept = serverSocket.accept();
            //提交线程执行的任务
            executorService.submit(()->{
                try{
                    /*
                     *socket对象进行读写操作
                     */
                    //转换流,读取浏览器请求第一行
                    BufferedReader readWb = new BufferedReader(new
InputStreamReader(accept.getInputStream()));
                    String requst = readWb.readLine();
                    //取出请求资源的路径
                    String[] strArr = requst.split(" ");
                    System.out.println(Arrays.toString(strArr));
```

```
                    String path = strArr[1].substring(1);
                    System.out.println(path);

                    //----
                    FileInputStream fis = new FileInputStream(path);
                    System.out.println(fis);
                    byte[] bytes= new byte[1024];
                    int len = 0 ;

                    //向浏览器 回写数据
                    OutputStream out = accept.getOutputStream();
                    out.write("HTTP/1.1 200 OK\r\n".getBytes());
                    out.write("Content-Type:text/html\r\n".getBytes());
                    out.write("\r\n".getBytes());
                    while((len = fis.read(bytes))!=-1){
                        out.write(bytes,0,len);
                    }

                    fis.close();
                    out.close();
                    readWb.close();
                    accept.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            });
        }
    }
}
```

7. ThreadLocal

> 说一下ThreadLocalMap的key为什么是弱类型？使用完ThreadLocal为什么必须要remove？

7.1 什么是ThreadLocal?

ThreadLocal 被译为**“线程本地变量”**类，在 Java 的多线程并发执行过程中，为保证多个线程对变量的安全访问，可以将变量放到ThreadLocal 类型的对象中，使变量在每个线程中都有独立值，不会出现一个线程读取变量时而被另一个线程修改的现象。

ThreadLocal 是解决线程安全问题一个较好方案，它通过为每个线程提供一个独立的本地值，去解决并发访问的冲突问题。很多情况下，使用 ThreadLocal 比直接使用同步机制（如 synchronized）解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

举例：

- ThreadLocal在Spring中作用巨大，在管理Request作用域中的Bean、事务、任务调度、AOP等模块都有它。
- Spring中绝大部分Bean都可以声明成Singleton作用域，采用ThreadLocal进行封装，因此有状态的Bean就能够以singleton的方式在多线程中正常工作了。

7.2 ThreadLocal 使用场景

ThreadLocal 使用场景大致可以分为以下两类：

1. 解决线程安全问题

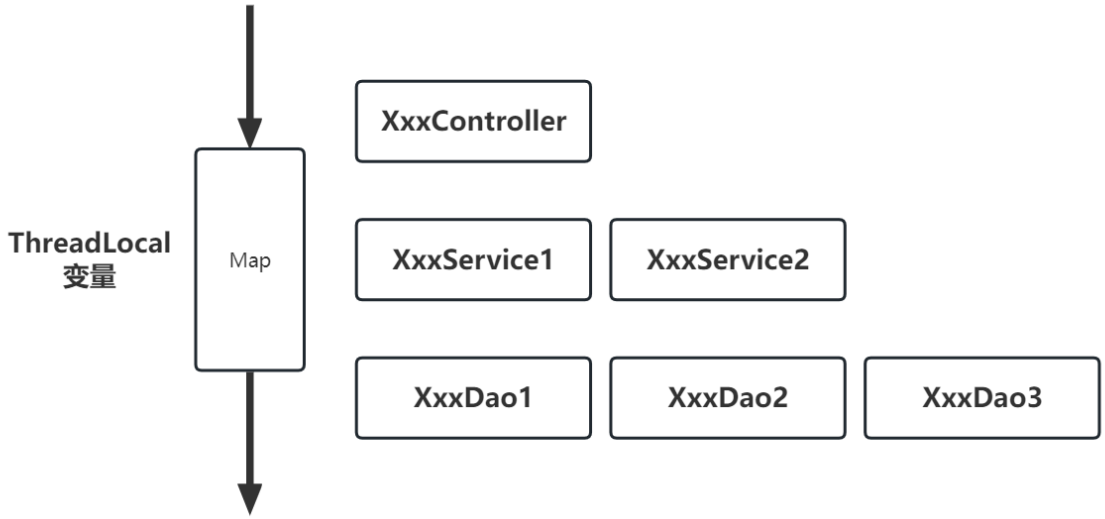
ThreadLocal 的主要价值在于解决线程安全问题， ThreadLocal 中数据只属于当前线程，其本地值对别的线程是不可见的，在多线程环境下，可以防止自己的变量被其他线程篡改。另外，由于各个线程之间的数据相互隔离，避免同步加锁带来的性能损失，大大提升了并发性的性能。

典型案例：可以每个线程绑定一个数据库连接，是的这个数据库连接为线程所独享，从而避免数据库连接被混用而导致操作异常问题。

```
//伪代码
private static final ThreadLocal localSqlSession = new ThreadLocal();

public void startManagedSession() {
    this.localSqlSession.set(openSession());
}
@Override
public Connection getConnection() {
    final SqlSession sqlSession = localSqlSession.get();
    if (sqlSession == null) {
        throw new SqlSessionException("Error:  Cannot get connection.  No
managed session is started.");
    }
    return sqlSession.getConnection();
}
```

2. 跨函数传递数据

通常用于同一个线程内，跨类、跨方法传递数据时，如果不用 ThreadLocal，那么相互之间的数据传递势必要靠返回值和参数，这样无形之中增加了这些类或者方法之间的耦合度。



“跨函数传递数据”场景典型案例：可以每个线程绑定一个 Session（用户会话）信息，这样一个线程的所有调用到的代码，都可以非常方便地访问这个本地会话，而不需要通过参数传递。

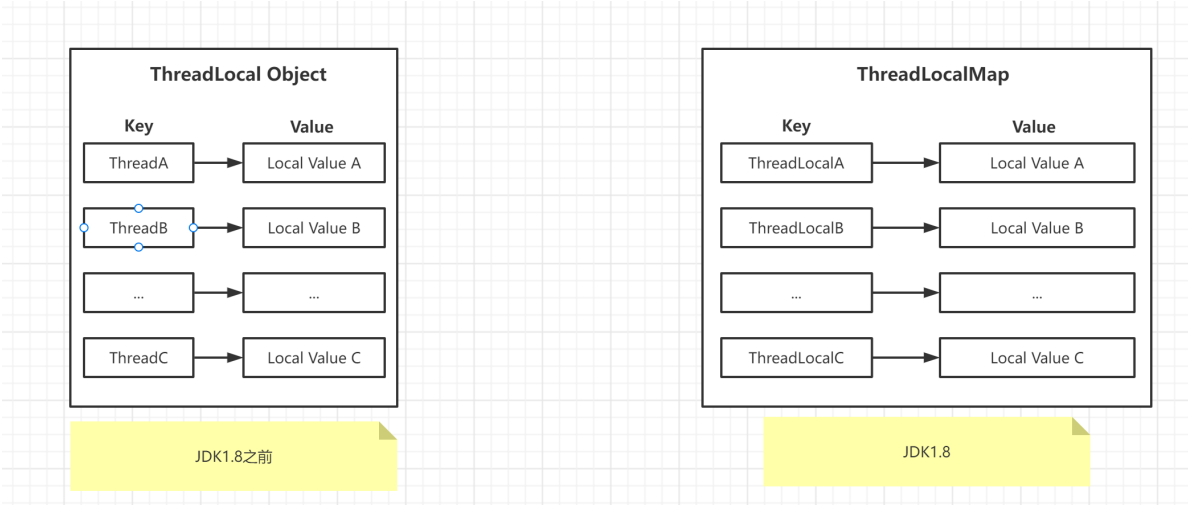
```
//伪代码
public class SessionHolder{
```

```
    // 用户信息 线程本地变量
    private static final ThreadLocal<UserDTO> sessionUserLocal = new
ThreadLocal<>("sessionUserLocal");
    // session 线程本地变量
    private static final ThreadLocal<HttpSession> sessionLocal = new
ThreadLocal<>("sessionLocal");
    //...省略其他
    /**
      *保存 session 在线程本地变量中
    */
    public static void setSession(HttpSession session){
        sessionLocal.set(session);
    }
    /**
    * 取得绑定在线程本地变量中的 session
    */
    public static HttpSession getSession() {
        HttpSession session = sessionLocal.get();
        Assert.notNull(session, "session 未设置");
        return session;
    }
    //...省略其他
}
```

7.3 底层原理

ThreadLocal内部结构演进:

早期ThreadLocal为一个 Map。当工作线程 Thread 实例向本地变量保持某个值时，会以“Key-Value 对”的形式保存在 ThreadLocal 内部的 Map 中，其中 Key为线程 Thread 实例， Value 为待保存的值。当工作线程 Thread 实例从 ThreadLocal 本地变量取值时，会以 Thread 实例为 Key，获取其绑定的 Value。



在 JDK8 版本中， ThreadLocal 的内部结构依然是Map结构，但是其拥有者为Thread线程对象，每一个 Thread 实例拥有一个ThreadLocalMap对象。Key 为 ThreadLocal 实例。

与早期版本的 ThreadLocalMap 实现相比，新版本的主要的变化为:

- 拥有者发生了变化：新版本的 ThreadLocalMap 拥有者为 Thread，早期版本的ThreadLocalMap 拥有者为 ThreadLocal。

- Key 发生了变化：新版本的 Key 为 ThreadLocal 实例，早期版本的 Key 为 Thread 实例。

与早期版本的 ThreadLocalMap 实现相比，新版本的主要优势为：

- ThreadLocalMap 存储的“Key-Value 对”数量变少
- Thread 实例销毁后， ThreadLocalMap 也会随之销毁，在一定程度上能减少内存的消耗。

Thread、ThreadLocal、ThreadLocalMap关系

```
Thread --> ThreadLocalMap --> Entry(ThreadLocalN, LocalValueN)*n
```

7.4 Entry 的 Key 为什么需要使用弱引用?

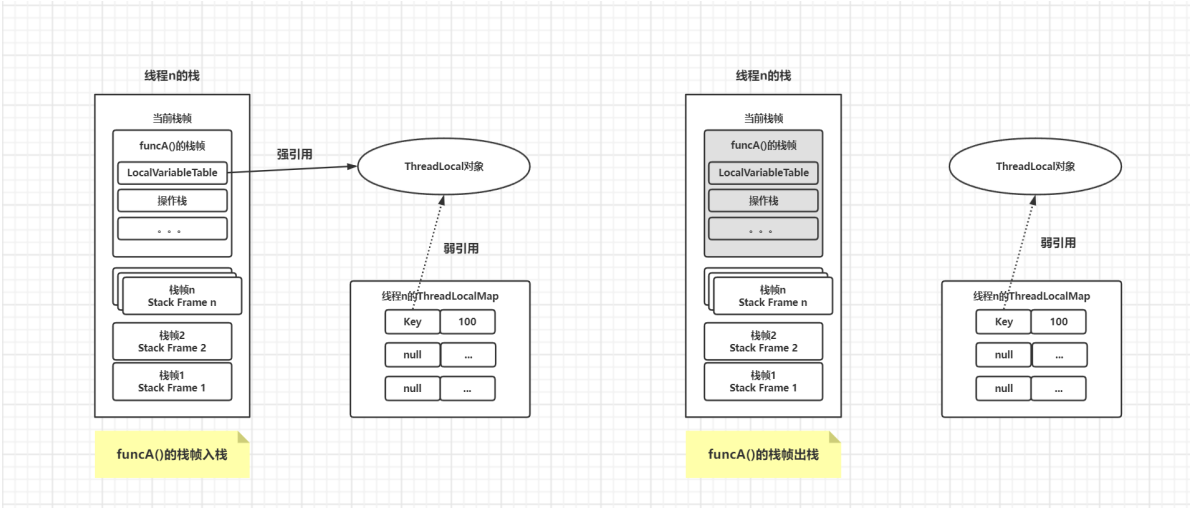
Entry 用于保存 ThreadLocalMap 的“Key-Value”条目，但是 Entry 使用了对 Threadlocal 实例进行包装之后的弱引用（WeakReference）作为 Key，其代码如下：

```
// Entry 继承了 WeakReference,并使用 WeakReference 对 Key 进行包装
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value; //值
    Entry(ThreadLocal<?> k, Object v) {
        super(k); //使用 WeakReference 对 Key 值进行包装
        value = v;
    }
}
```

为什么 Entry 需要使用弱引用对 Key 进行包装，而不是直接使用 Threadlocal 实例作为 Key呢? 比如如下代码

```
//伪代码
public void funcA() {
    //创建一个线程本地变量
    ThreadLocal local = new ThreadLocal();
    //设置值
    local.set(100);
    //获取值
    local.get();
    //函数末尾
}
```

当线程n 执行 funcA 方法到其末尾时，线程n 相关的 JVM 栈内存以及内部 ThreadLocalMap成员的结构，大致如图所示。

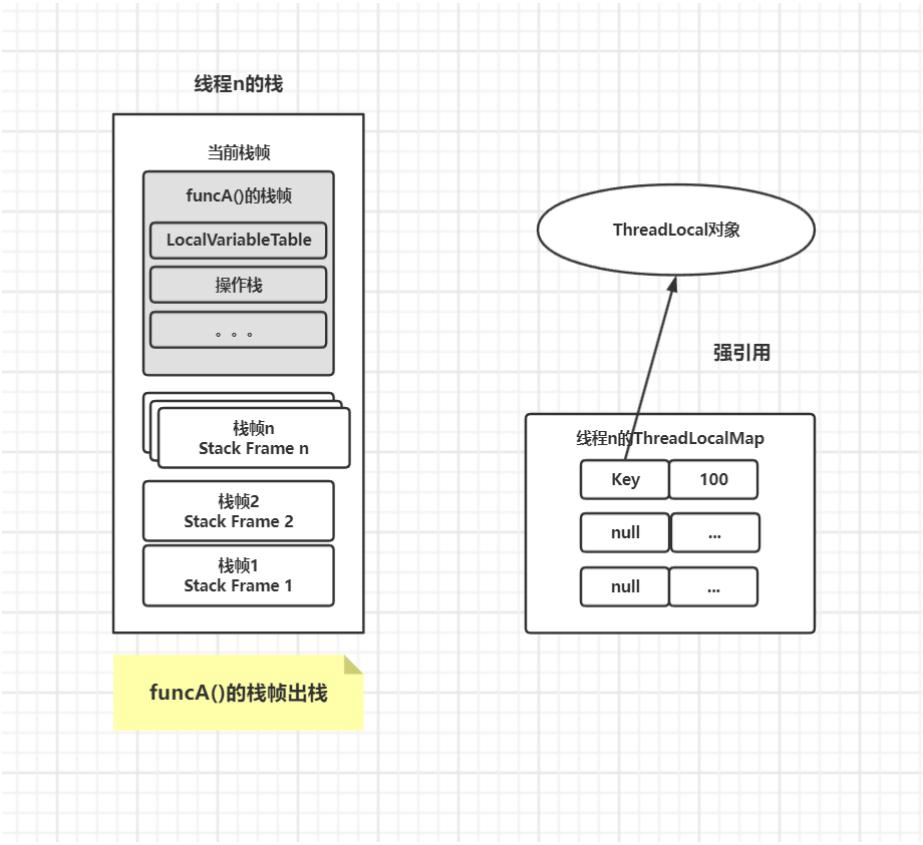


线程n 调用 funcA（）方法新建了一个 ThreadLocal 实例，并使用 local 局部变量指向这个实例，并且此 local 是强引用；

在调用 local .set（100）之后，线程n 的 ThreadLocalMap 成员内部会新建一个 Entry 实例，其 Key 以弱引用包装的方式指向 ThreadLocal 实例。

当线程n 执行完 funcA 方法后， funcA 的方法栈帧将被销毁，强引用 local 的值也就没有了，但此时线程的 ThreadLocalMap 里的对应的 Entry 的 Key 引用还指向了 ThreadLocal 实例。

若 Entry的 Key 引用是强引用，就会导致 Key 引用指向的 ThreadLocal 实例、及其 Value 值都不能被GC回收，这将造成严重的内存泄露，具体如图所示。

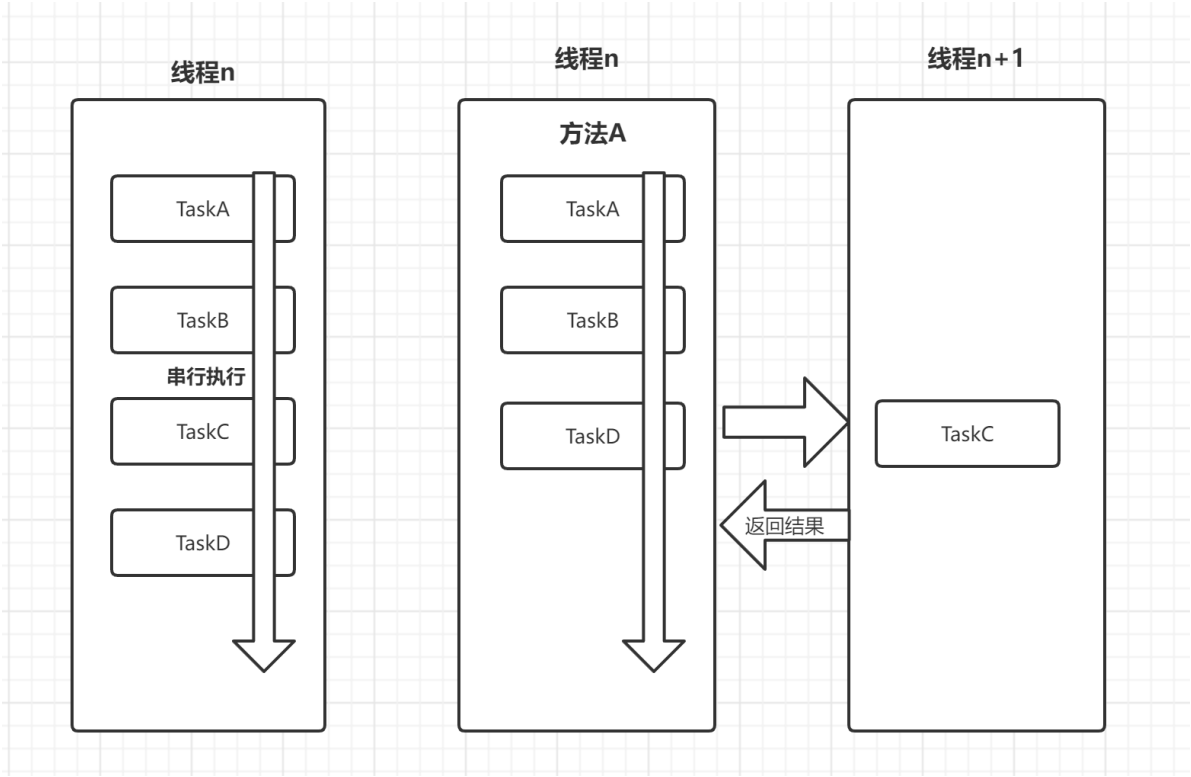


由于 ThreadLocalMap 中 Entry 的 Key 使用了弱引用，在下次 GC 发生时，就可以使那些没有被其他强引用指向、仅被 Entry 的 Key 所指向的 ThreadLocal 实例能被顺利回收。并且，在 Entry的 Key 引用被回收之后，其 Entry 的 Key 值变为 null。后续当 ThreadLocal 的 get、 set 或 remove 被调用时，通过 expungeStaleEntry方法， ThreadLocalMap 的内部代码会清除这些 Key 为 null 的 Entry，从而完成相应的内存释放。

8. Future和FutureTask

8.1 Future类

FutureTask叫未来任务，可以将一个复杂的任务剔除出去交给另外一个线程来完成



图示为main方法开始，依次调用多个方法。但是第3个方法复杂，为避免其是瓶颈影响整个程序效率，所以将其剔除出去交给FutureTask去完成，这样避免阻塞主线程。

8.3 Future主要方法

get()

get方法的行为取决于Callable任务的状态，只有以下5种情况：

1. 任务正常完成：get方法会立刻返回结果
2. 任务尚未完成：任务还没有开始或进行中，get将阻塞并直到任务完成。
3. 任务执行过程中抛出Exception：get方法会抛出ExecutionException，这里抛出异常，是call()执行时产生的那个异常
4. 任务被取消：get方法会抛出CancellationException
5. 任务超时：get方法有一个重写方法，是传入一个延迟时间的，如果时间到了还没有获得结果，get方法会抛出TimeoutException

get(long timeout,TimeUnit unit)

如果call()在规定时间内完成任务，那么就会正常获取到返回值，而如果在指定时间内没有计算出结果，则会抛出TimeoutException

cancel()

- 如果这个任务还没有开始执行，任务会被正常取消，未来也不会被执行，返回true
- 如果任务已经完成或已经取消，则cancel()方法会执行失败，方法返回false
- 如果这个任务已经开始，这个取消方法将不会直接取消该任务，而是会根据参数mayInterruptIfRunningg来做判断。如果是true,就会发出中断信号给这个任务。

isDone()

- 判断线程是否执行完，执行完并不代表执行成功。

isCancelled()

- 判断是否被取消

8.4 用线程池的submit方法返回Future对象

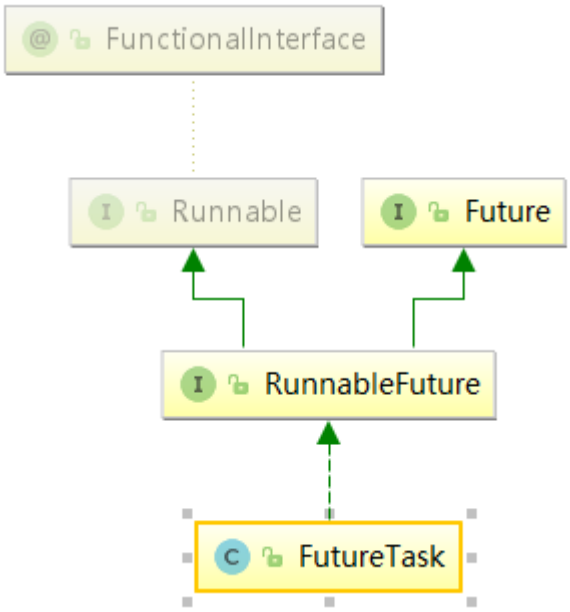
首先要给线程池提交任务，提交时线程池会立刻返回一个空的Future容器。当线程的任务一旦执行完，也就是当我们可以获取结果时，线程池会把该结果填入到之前给我们的那个Future中去（而不是创建一个新的Future），我们此时可以从该Future中获得任务执行的结果。

```
package com.hero.multithreading;

import java.util.Random;
import java.util.concurrent.*;

/**
 * 案例：演示一个Future的使用方法
 */
public class Demo22Future {
    public static void main(String[] args) {

        ExecutorService service = Executors.newFixedThreadPool(10);
        Future<Integer> future = service.submit(new CallableTask());

        try {
            //等待3秒后打印
            System.out.println(future.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        service.shutdown();
    }

    static class CallableTask implements Callable<Integer> {
        @Override
        public Integer call() throws Exception {
            Thread.sleep(3000);
            return new Random().nextInt();
```

```
        }
    }
}
```

8.5 用FutureTask来创建Future

用FutureTask来获取Future和任务的结果。FutureTask实现Runnable和Future接口



把Callable实例当作参数，生成FutureTask的对象，然后把这个对象当作一个Runnable对象，用线程池去执行这个Runnable对象，最后通过FutureTask获取刚才执行的结果。

```
package future;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.FutureTask;

/**
 * 描述：演示FutureTask的用法
 */
public class Demo23FutureTask {

    public static void main(String[] args) {
        Task task = new Task();
        //FutureTask继承Future和Runnalbe接口
        FutureTask<Integer> integerFutureTask = new FutureTask<>(task);
//        new Thread(integerFutureTask).start();
        ExecutorService service = Executors.newCachedThreadPool();
        service.submit(integerFutureTask);

        try {
            System.out.println("task运行结果："+integerFutureTask.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class Task implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        System.out.println("子线程正在计算");
        Thread.sleep(3000);
        //模拟子线程处理业务逻辑
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        return sum;
    }
}

```

扩展01-锁升级

> **多线程锁的升级原理是什么?**

在Java中，synchronized共有4种状态，级别从低到高依次为：无状态锁，偏向锁，轻量级锁和重量级锁状态，这几个状态会随着竞争情况逐渐升级，**锁可以升级但不能降级。**

- **偏向锁**：是指当一段同步代码一直被同一个线程所访问时，即**不存在多个线程的竞争时**，那么该线程在后续访问时便会自动获得锁，从而降低获取锁带来的消耗，即提高性能。
- **轻量级锁**：（自旋锁）是指**当锁是偏向锁的时候，却被另外的线程所访问**，此时偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，线程同样不会阻塞。长时间的自旋操作是非常消耗资源的，一个线程持有锁，其他线程就只能在原地空耗CPU，执行不了任何有效的任务，这种现象叫做忙等（busy-waiting）
- **重量级锁**：此忙等是有限度的。如果锁竞争情况严重，某个达到最大自旋次数的线程，会将轻量级锁升级为重量级锁。当后续线程尝试获取锁时，发现被占用的锁是重量级锁，则直接将自己挂起（而不是忙等），等待将来被唤醒。，有个计数器记录自旋次数，默认允许循环10次，可以通过虚拟机参数更改