1.监视器是什么

在java中，每个对象都有一个内置的监视器用于控制对象的同步访问，可以说监视器是实现锁的基础

- 监视器与对象：当线程进入一个被syn修饰的方法或代码块时，他会尝试获取对象的监视器锁，如果锁可用，线程获取锁并执行代码，否则线程阻塞
- 监视器的作用：
  - 确保同一时间只有一个线程可以执行与对象关联的同步代码块和方法
  - 通过锁机制实现线程同步，防止多个线程访问共享资源，从而避免数据不一致问题

2.java中创建线程的方式

1.继承Thread类

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // 启动线程
    }
}
```

2.实现runnable接口

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // 启动线程
    }
}
```

3.使用callable和future

这种方法相对比与runnable是可以获取返回值的

```
import java.util.concurrent.*;

class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
```

```
        return "Thread is running";
    }
}

public class Main {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(new MyCallable());
        System.out.println(future.get()); // 获取线程执行结果
        executor.shutdown();
    }
}
```

4.线程池

3.为什么用ThreadLocal而不是线程的成员变量

如果使用线程的成员变量，就像下面这样

```
class MyThread extends Thread {
    private int myVar; // 线程成员变量
    public void run() {
        // 使用myVar
    }
}
```

那么如果使用实现runnable来实现，runnable的实现类实无法访问Thread的成员变量的，如果要访问只能想办法把Thread作为参数穿进去，从而造成代码耦合！

4.JMM内存模型

1.核心概念

- 主内存：所有线程的共享变量存储的内存区域
- 工作内存：每个线程都有自己私有的工作内存，存储了线程对应主内存中共享变量的副本，线程所有的操作都在自己的工作内存执行，不直接操作主内存

2.关键规则

可见性：通过volatile（happens-before），syn，lock等实现

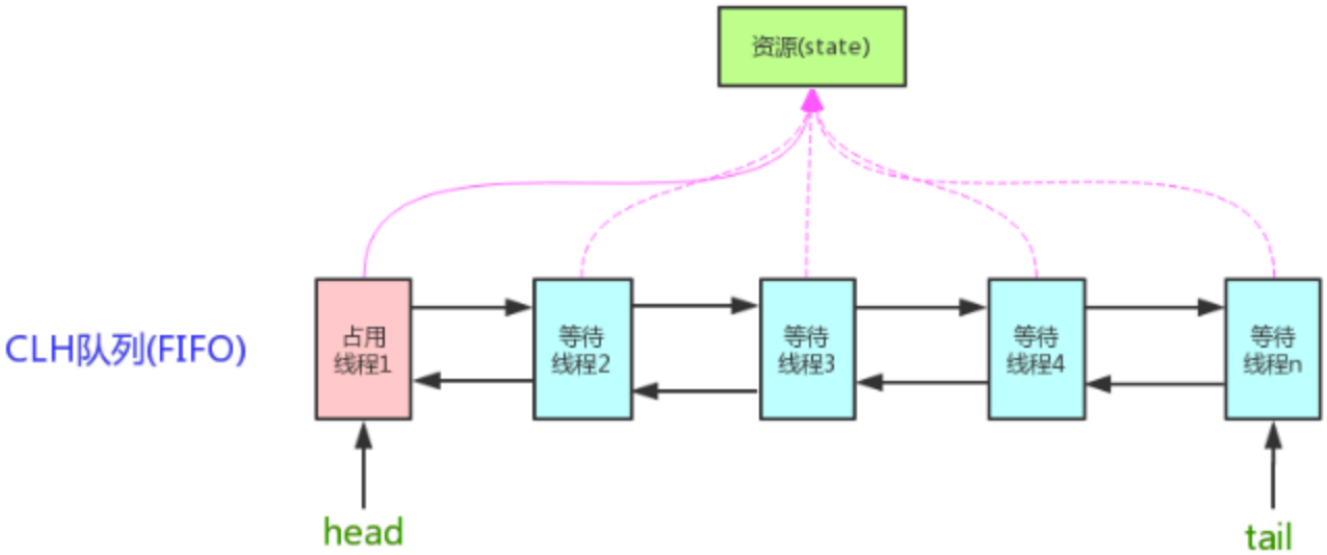
有序性：指令重排可以用于提升性能，但是多线程环境下可能出错，JMM使用happens-before保证有序性

原子性：对基本数据类型的读写操作是原子的

5.AQS

AQS依赖一个双向链表和一个violate的state变量实现，每有一个线程获取锁，state++

AQS实现的关键：一个先进先出的等待队列，通过将等待线程加入等待队列中，然后再释放同步时从等待队列中唤醒等待线程从而实现同步



AQS的分类：

- 独占式：只有一个线程可以占有资源（ReenTrantLock）tryAquire
  - 可重入的实现：
    - 1.SetExclusiveOwnerThread(currentThread)
    - 2.getExclusiveOwnerThread()==currentThread
- 共享式：多个线程可以同时占用同步资源（CountDownLatch）tryAquireShare

AQS的核心方法：

- 3个和修改同步状态有关的方法（state 变量）
- 5个和等待队列有关的可重写方法
- 9个模板方法（tryAcquire，tryRelease，tryAcquireShared等），这些都是定义好框架但是由子类去实现

6.写锁和读锁的共存问题

1.持有读锁：不可以获取写锁，因为读锁时共享锁，不能确定还有没有其他线程获取了读锁正在读数据

2.持有写锁：可以获取读锁，前提是同一线程，因为写锁是独占锁，不可能有其他线程获取，占有者自己读自己写的数据是合理的

7.读写锁的升降级

1.写锁可以降级为读锁：直接释放写锁资源即可

2.读锁不能升级为写锁：写锁是独占锁，两个读锁都想升级为写锁，他们都需要等待对方释放资源，从而可能导致死锁

8.ReentrantLock是怎么实现公平锁和非公平锁的

1.非公平锁：在tryAcquire方法中，直接调用了父类Sync的nonfairTryAcquire方法

```java
abstract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = -5179523762034025860L;

    Performs Lock.lock. The main reason for subclassing is to allow fast path for nonfair version.
    abstract void lock();

    Performs non-fair tryLock. tryAcquire is implemented in subclasses, but both need nonfair
    try for trylock method.
    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;
    }
```

2.公平锁：唯一的区别就是判断条件了多了hasQueuedPredecessors这个函数，判断当前同步队列里是否有前驱节点，如果有，那么当前线程就要等待前驱节点获取并释放资源后才能执行