StringBuffer，StringBuilder区别是啥?

```
String str="hello";
str=str+"world";
内存变化：首先开辟空间存储hello，再开辟空间存储world，再开辟内存空间，helloworld，最后将地址的
因此创建了StringBuffer和StringBuilder，多次修改不会产生新的引用对象
StringBuffer：可变序列，效率低，线程安全，使用了很多synchornized修饰符
StringBuilder：可变字符序列，效率高，线程不安全，有很对非原子操作
```

什么是线程安全?

```
线程同步：就是当一个程序对一个线程安全的方法或者语句进行访问的时候，其他的不能他进行操作了，必须等
    这次访问结束后才能对这个线程安全的方法进行访问
线程安全：代码所在的进行中有多个线程再同时运行，而这些线程可能会同时运行这段代码，如果每次运行的结
    而且其他的变量的值和预期的是一样的，那么就是线程安全的，当然这个接口对于线程来说是原子操作或
    不会导致执行结果存在二义性，也就是说不用考虑同步问题
    线程安全都是因为全局变量，静态变量存在写操作而引起的

```

如何保证线程安全

```
线程安全等级：1.不可变（对象不可变）2.绝对线程安全（无论何时调用者都不需要额外的同步措施）
    3.相对线程安全（对于一些特定顺序的连续调用）4.线程兼容（线程不安全需要使用同步手段）
    5.线程对立（无论怎样使用这些代码，都无法保证线程安全）
线程安全的实现方法：
    1.互斥同步（阻塞同步）：保证共享数据同一时刻只被一个线程使用，reentrantLock与synchroniz
    2.非阻塞同步：CAS算法（compare and Swap），cas算法会导致ABA问题，可以通过加入版本号来解
    3.无需同步方案
        1）可重入代码：可以在程序执行的任何时候去中断他，转而去执行另一端程序，但当控制权返回后
        2）线程本地存储：将线程存储在本地，如果一段的代码的所需数据与其他代码是共享的，那就看这
            如果可以就把共享数据的可见范围控制在同一个线程内，这样也就无需同步
      符合这个特点的应用并不常见，大部分的消费队列的架构模式都会将产品的消费过程尽量在一个线程中
      经典的例子就是：web交互模式中：一个请求对应一个服务器线程，这使得很多web服务器应用都可以
```

注意：Spring中的controller service Dao都是单例的，也就是请求其实访问的是服务器端的同一个实例对象，那么这些Bean的对象会被不同的线程调用，容易出现线程不安全问题，解决：Service的方法是有事物控制的，因此一旦执行就不会被中断处理

但是：其中就不应该存在共享的实例对象，多个线程可能会修改他的内容

```
Spring Dao是单例的，connection是多例的，那是怎么回事 。Spring利用了ThreadLocal
public class TopicDao {

  //①使用ThreadLocal保存Connection变量
private static ThreadLocal<Connection> connThreadLocal = new ThreadLocal<Connecti
public static Connection getConnection(){

        //②如果connThreadLocal没有本线程对应的Connection创建一个新的Connection，
        //并将其保存到线程本地变量中。
if (connThreadLocal.get() == null) {
            Connection conn = ConnectionManager.getConnection();
            connThreadLocal.set(conn);
              return conn;
        }else{
              //③直接返回线程本地变量
            return connThreadLocal.get();
        }
    }
    public void addTopic() {

        //④从ThreadLocal中获取线程对应的
         Statement stat = getConnection().createStatement();
    }
}
```

如上代码，每个线程调用时都会先去ThreadLoca中去判断又没有当前线程的conn，如有则调用，没有则创建并调用，从而实现了单例的Dao使用多个Connection的方法，

实际上上面的threadLocal方法应该是在service的事物处理中实现的，

什么是死锁以及避免死锁

线程死锁定义：指由于两个或多个线程相互持有对方需要的资源，导致这些线程处于等待状态，无法前往执行，当线程进入对象的synchronized代码块时，便占有了资源，直到它推出该代码块或者调用wat方法，才释放资源，在此期间其他线程不能进入该代码块

当线程互相持有对象对方需要的资源时，会互相等待对方释放资源，如果线程都不主动释放占有的资源，将产生死锁

```
--线程死锁必须满足的一些特定条件
1.互斥条件：进程对与所分配的资源具有排他性，一个资源只能被一个进程占用，直到该进程释放
2.请求和保持条件：一个进程因请求被占用资源发生阻塞时对以获得的资源保持不放
3.不剥夺条件：任何一个资源在没被该进程释放前，任何其他进程都无法对他剥夺占用
```

```
4.循环等待：当发生死锁时，所等待的进程必定会形成环路（类似于死循环），造成永久阻塞
```

避免死锁：

加锁顺序：

```
Thread 1:
  lock A
  lock B

Thread 2:
   wait for A
   lock C (when A locked)

Thread 3:
   wait for A
   wait for B
   wait for C
线程2和3只有在获取了锁A后才能尝试获取锁，按照顺序加锁是一种有效的死锁预防机制，但是总有时候我们无
```

加锁时限：

```
为尝试加锁的时候加一个超时时间，若超过了这个时限该线程则放弃对该锁进行请求，若是一个线程没有在给的
成功获取所有需要的锁，则会进行回退，并释放所有已经获得的锁，等待一段随机事件再重试
注意：存在锁超时时，不一定就是出现了死锁，也可能是因为获得了锁的线程，需要很长的时间去完成他的任务
    也有可能多个线程的超时时间很接近，到时重试的时机也很接近，新一轮的竞争也可能会失败
```

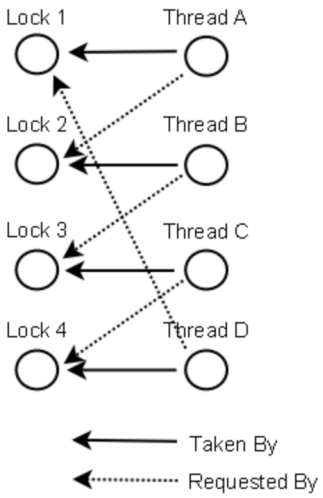
死锁检测：

```
主要是针对那些不可能实现按序加锁，并加锁超时也不可行的场景
每当一个请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生，当发生时候：释放锁，回退，等待
    他是死锁发生了才回退，不同于简单的加锁超时（超时就回退和等待），但是大量线程竞争同一批锁时还
    解决方案是：设置线程的优先级，让一个或者几个线程回退，剩下的线程就像没发生死锁一样继续保持，
    同一批的线程总是会拥有更高的优先级，为避免这个问题，可以在死锁发生时设置随机的优先级
```

银行家算法（资源分配算法）

模拟分配当前的剩余资源给需要的进程，判断是否全部的进程能够执行结束，不发生死等待

下面是一幅关于四个线程（A,B,C和D）之间锁占有和请求的关系图。像这样的数据结构就可以被用来检测死锁。



- 检查已占用资源的进程是否有正在等待的资源，如有则去占用该资源的进程中寻找他等待的资源，如此循环，如果最后发现找到了自己，那就说明这是一个死锁等待

Synchronized的实现原理

```
public synchronized void doSth(){
        System.out.println("Hello World");
    }

    public void doSth1(){
        synchronized (SynchronizedTest.class){
            System.out.println("Hello World");
        }
同步方法，JVM采用ACC_SYNCHRONIZED标记符来实现同步，对于同步代码块，JVM采用minitorenter和mo
1.同步方法通过ACC_SYNCHRONIZED关键字隐式的对方法进行加锁。当线程要执行的方法时，先判断是都有这
    执行结束后释放监视器锁，如果其他方法执行该方法，就因为监视器锁而被阻挡
2.同步代码块通过monitorenter和monitorexit执行来进行加锁。当线程执行到monitorenter的时候要先
3.每个对象自身维护这一个被加锁次数的计数器，当计数器数字为0时表示可以被任意线程获得锁。当计数器不
```

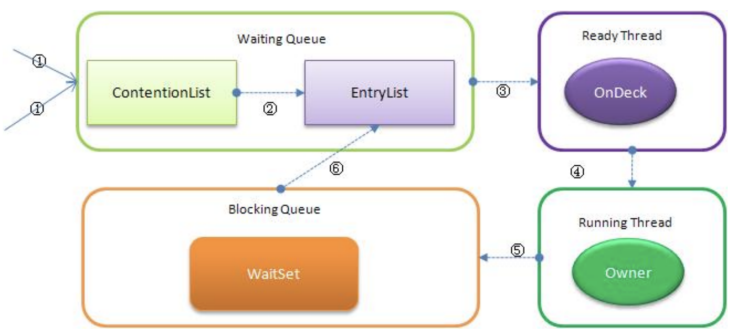
Monitor

```
无论是同步方法还是同步代码块，ACC_SYNCHORONIZED，monitorentor，monitorexit都是基于Monito
Monitor是一种同步工具，通常被描述为一个对象
    1.任何线程进入任何一个方法都需要获取Monitor的许可，离开时归还许可
ObjectMonitor:
五个内部成员:
    1._owner：指向持有ObjectMonitor对象的线程
    2._WaitSet：存储处于wait状态的线程队列
    3._EntryList:存放处于等待锁block状态的线程队列
    4._recursions：锁重入次数
```

```
    5._count：用来记录该线程获取锁的次数
    当多个线程同步访问一段同步代码时候，首先会进入Contension List中，当有线程执行结束需要unloc
    Owner线程并不是把锁传递给ondeck线程，而是把竞争锁的权利交给他，当某个线程获取到对象的monit
将monitor中的owner设为当前线程，同时count+1，即获得对象锁
    持有monitor的线程调用wait方法，将释放当前持有的monitor，owner变量恢复为null，count-1，
    waitSet集合中等待被唤醒，或当前线程执行完毕后也将释放monitor并复位变量的值，以便其他线程进
```

synchronized和volatile

```
volatile被誉为轻量级的synchronized，volatile可以保证可见性和有序性，实现原理是通过内存屏障实
volatile有一个重要的作用，是synchronized不具备的，那就是禁止指令重排序
指令重排序
    Java语言规范JVM线程内部维持顺序花语义，即只要程序的最终结果与它顺序化情况的结果相等，那么指
    指令重排序的意义：使指令更加符合CPU的执行特性，最大限度的发挥机器的性能，提高程序的执行效率。
volatile关键字提供了一个功能：
    那就是被其修饰的变量在被修改后可以立即同步到主内存（缓存一致性原理），被其修饰的变量在每次是
    因此，可以使用volatile来保证多线程操作时变量的可见性。
缓存一致性协议：
    每个处理器通过嗅探在总线上传播的数据来检查自己的缓存是不是过期了，当处理器发现自己的缓存行对
    当处理器要对这个数据进行修改操作是，会强制重新从系统内存中吧数据读取到处理器缓存中
应用：如果一个变量被volatile所修饰的话，在每次数据变化之后，其值都会被强制刷入主存，二其他处理器
    这就保证了一个volatile在并发编程中，其值在多个缓存中是可见的
```

> 当有多个线程同时请求某个对象监视器时，对象监视器会设置几种状态来区分请求的线程：
>
> 1. Contention List：所有请求锁的线程被首先放置在该竞争队列中
> 2. Entry List：Contention List 中有机会获得锁的线程被放置到Entry List
> 3. Wait Set：调用wait()方法被阻塞的线程被放置到Wait Set中
> 4. OnDeck：任何一个时候只能有一个线程竞争锁 该线程称作OnDeck
> 5. Owner：获得锁的线程成为Owner
> 6. !Owner：释放锁的线程
>
> 转换关系如下图：
>
> 
>
>
> 新请求锁的线程被首先加入到Contention List中，当某个拥有锁定线程(Owner状态)调用unlock之后，如果发现Entry List为空就从ContentionList中移动线程到Entry List中

synchornized升级过程

```
偏向锁（Biased Lock）：
  按照之前HotSpot设计，每次加锁和解锁都会涉及cas操作，这会导致本地调用的延迟，因此偏向锁的想法是
    之后让这个监视对象偏向这个线程，之后的多次调用只需判断该锁是否偏向这个线程而无需CAS操作
    流程锁对象的对象头有一个ThreadId字段，如果字段为空，第一次获取锁的时候就把自身的ThreadId写
    下次获取锁的时候，直接查看ThreadId是否和自身线程线程id一致，如果一致就认为该线程已经取得了
自旋锁 spin lock
处于Contetion List，Entry List和Wait Set中的线程均属于阻塞状态，阻塞操作由操作系统完成，线程
这会导致用户态和内核态之间来回切换，严重影响锁的性能，这时可以让争用的线程稍微等待，执行几个空的f
等待锁被释放（自旋锁），这种适合执行时间比较短的代码块，如果时间较长则会导致自旋的时间太长，浪费c
因此，可以根据cpu的负载情况设置自旋时间（平均负载小于cpu个数则一直自旋，）
轻量级锁（利用到了自旋锁）：
    轻量级锁认为大多数情况下不会出现锁竞争，即使出现了，获取锁的线程也能很快释放锁，获取不到锁的线
    因此流程：线程在执行同步块之前，JVM会在当前线程的栈桢中存储锁记录的空间，并把对象头中的Mark
    然后线程尝试持有CAS吧对象头中的Mark Word指向锁记录的指针，如果成功
    当前线程就获得锁，如果失败表示有其他线程竞争锁，
    当前线程尝试使用自旋来获取锁，获取失败就升级成重量级锁
重量级锁：
    就是synchornized的实现原理，会将等待的线程阻塞，被阻塞的线程不会消耗cpu，但是阻塞或者唤醒线
        状态很消耗时间，使用了ACC_SYNCHRONIZED标记和monitor
各种锁的比较：
    偏向锁：加锁不需要额外消耗，只需判断ThreadId，但是当线程间存在锁竞争，会带来额外的锁撤销的消
            适用于只有有一个线程访问同步块的场景
    轻量级锁：竞争的线程不会阻塞，提高程序响应速度，但是始终得不到锁竞争的线程使用自旋会消耗cpu资
            适用于执行锁的时间较短的情况，这样响应速度快
    重量级锁：线程不会自旋不会额外消耗cpu，但是线程阻塞，响应时间长
            适用于追求吞吐量，执行同步块时间的长的场景
ps:偏向锁和轻量级锁理念上的区别：
轻量级锁：在无竞争的情况下使用CAS操作去消除同步使用的互斥量
偏向锁：在无竞争的情况下把整个同步都消除掉，连CAS操作都不做了
CAS锁：比如一个变量的初始值是0，想对他加一，加一的时候先判断还是不是0，如果是就加1，不是的话则ca

```

源码注意：synchornized内部的等待队列其实是一个node串联的链表，当锁被释放之后，会唤醒头结点的元素，当这个线程释放之后会在唤醒下一个线程

```
对于synchronized加锁的完整过程描述：
检查Mark Word里存放的是否是自身的ThreadId，如果是，表示当前线程处于偏向锁，无需加锁就可获取临界
如果不是自身的ThreadId，锁升级，使用CAS来进行切换，新的线程根据MarkWord里现有的ThreadId，通知
```

```
4 两个线程都把对象的HashCode复制到自己新建的用于存储锁的记录空间，接着开始通过CAS操作把共享对象的
5 成功执行CAS的获得资源，失败的进入自旋
6 自旋在线程在自旋过程中，成功获得资源则整个状态依然处于轻量级的锁状态
7 如果自旋失败进入重量级锁的状态，自旋的线程进行阻塞，等待之前的线程完成并唤醒自己
```

reetrantLock原理

reetrantLock内部

state参数：无锁状态：0，有线程加锁，通过cas对其赋值为1，如果发现已被锁同时锁线程是自己则将state直接加1

定义了一个抽象类：Sync，这个Sync继承了AbstractQueuedSynchornizer接口也即是AQS，拥有了队列的链表node结构

定义了两个内部类：fair，nonFair，这两个类都继承了上面的Sync，同时新增了自己的tryAcquire方法，在reetrantLock对象创建时会根据参数决定内部的Sync是哪个实现类默认是unfair的

源码逻辑

AQS的acquire方法：

先tryAcquire一次成功则获取成功不阻塞当前线程，失败则添加到等待队列，并中断当前线程

tryAcquire逻辑3个流程

1、先判断state是否为0，如是则state设置成功并且没有排队任务或排队任务不是自己则任务可设置锁线程是自己，这里相当于加锁成功后会判断一次所线程是不是自己，如果不是则不需要修改加锁的线程

2、如果发现锁线程是自己则对state加一即可

3、否则认为加锁失败

AQS的acquire方法：

Fair的lock：

直接调用AQS的acquire方法

Fair的unlock：

直接调用sync实现的AQS的release方法：

1、获取state-1的值，如果是0则可以直接释放即清空设置锁线程为null且state为0返回true，如果不是0则直接对state减一并返回false

2、如果发现头结点不为空，则需要遍历node链表，找到等待的线程并唤醒下一个线程

NonFaire的lock：

先通过cas设置state为1成功则加锁成功，失败则调用acquire方法

NonFaire的unlock

与Fair的一样直接调用sync实现的AQS的release方法