多线程笔记整理

一、概述

1) 网络应用之间的通信一般需要明确 IP 和端口，因为一个网络应用往往占用一个或多个端口，然后采取某种网络协议来完成通信。
2) 网络编程一般使用 socket 类又名套接字，以 BIO 或 NIO 的形式，常用 BIO，NIO 常用于线程并发的编程中
3) 常见的协议及区别

1). UDP 协议

传递数据时不关心连接状态直接发送,他的传输效率高,传递的数据容易丢包

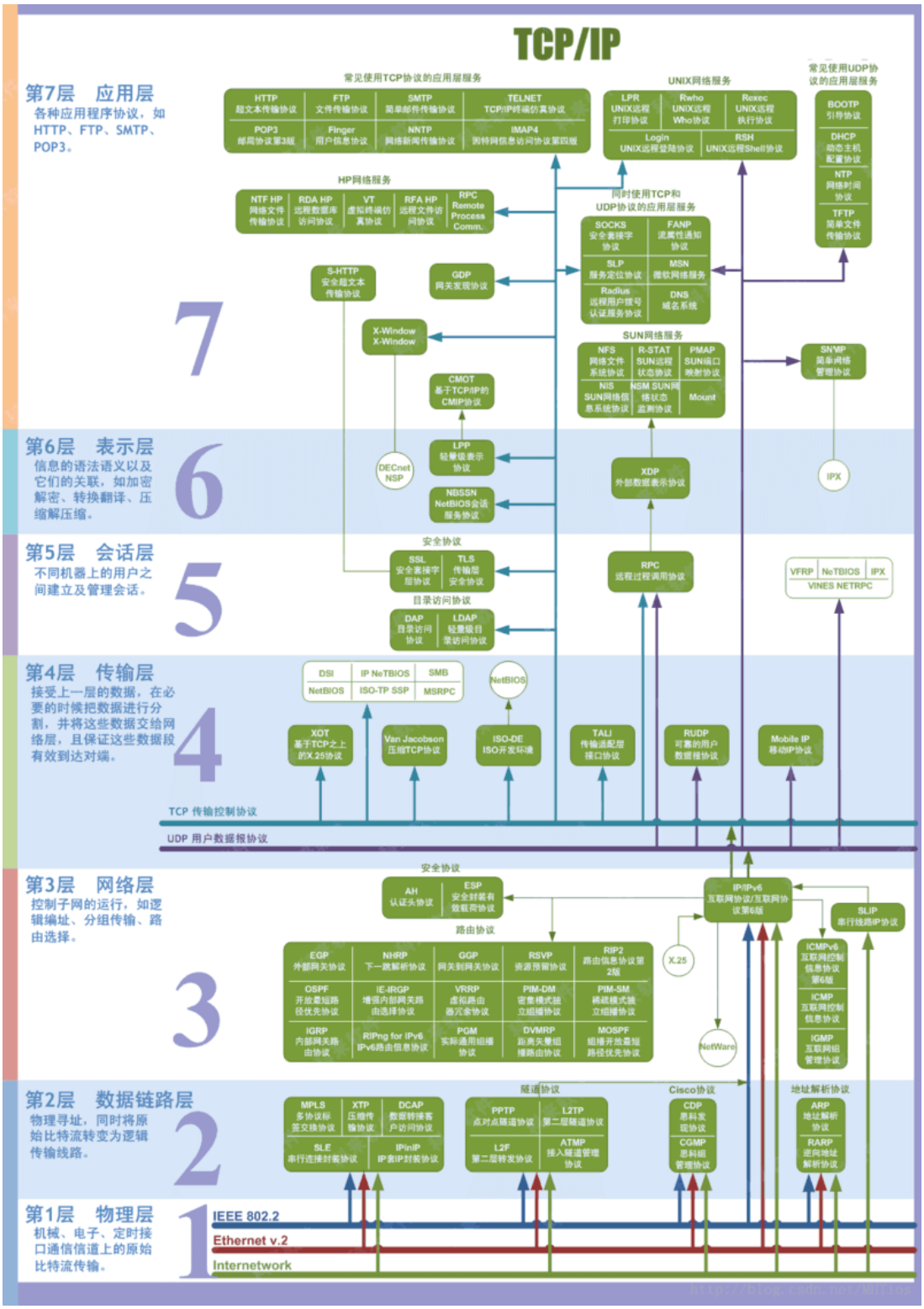
2). TCP/IP 协议

传递数据时关系连接的状态,连接成功才会发送数据,他的传输效率相对于 UDP 协议要低,会通过三次握手机制保证数据的完整性，java 编程默认使用的协议

3). Http 协议

相对 TCP 协议效率较低，是对 TCP 协议的封装

4) OSI 7 层模型
   a) 层数越高，效率越低
   b) UDP 在三层，TCP 在四层，HTTP 在 7 层



5) 我们可以通过 new ServerSocket + 端口的形式创建一个服务对象，通过 new Socket + 服务 IP,端口的形式创建一个连接对象，并通过输出流 OutputStream 向服务对象发送信息，通过捕获输入流 InputStream 来获取服务器的信息；
服务对象通过 accept 方法来获取连接对象；通过捕获连接对象中的输入流 InputStream 来获取，客户端的信息，通过输出流 OutputStream 向客户端发送消息。

```java
public static void main(String[] args) throws IOException {

    //创建socket对象
    Socket socket = new Socket( host: "192.168.43.81",  port: 8989);

    //向服务器发送数据
    OutputStream outputStream = socket.getOutputStream();
    outputStream.write("hello Server".getBytes());
    //代表客户端发送的数据完成
    socket.shutdownOutput();

    //获取服务器端响应数据
    InputStream inputStream = socket.getInputStream();
    int len = 0;
    byte[] b = new byte[1024];
    StringBuilder builder = new StringBuilder();
    while(true){
        len = inputStream.read(b);
        if(len==-1) {break;}
        builder.append(new String(b, offset: 0,len));
    }
    System.out.println(builder.toString());

    socket.shutdownInput();//确定读取响应数据结束
```

创建连接对象

向服务器发送消息

接受服务器消息

6) 在网络编程中，ServerSocket 如果没有并发操作，那么每次只有一个线程，必须等获取的套接字对象操作完，才能对下一个请求进行操作，这样大量的请求都会处于等待状态，很不利于生产开发。

多线程可以并行异步执行请求，主要用来解决网络编程中的问题。所以，开发方式有三种 :1.Thread 类对象以匿名内部类的方式实现 Runnable 接口，重写 run 方法，里面添加要并发的操作，并调用 Thread 类的 start 方法启动线程。2.创建一个类继承 Thread 类，重写 run 方法，run 方法里面为并发内容，以 start 方法启动线程。3.创建线程池。4.spring 的 @Async 注解

7) 但是由于每个计算机系统所能负载的线程数量是有上限的，当线程超过一定数量的时候，内存开销太大，同时 cpu 轮询寻找可操作线程也会导致卡顿，甚至出现死机的情况。

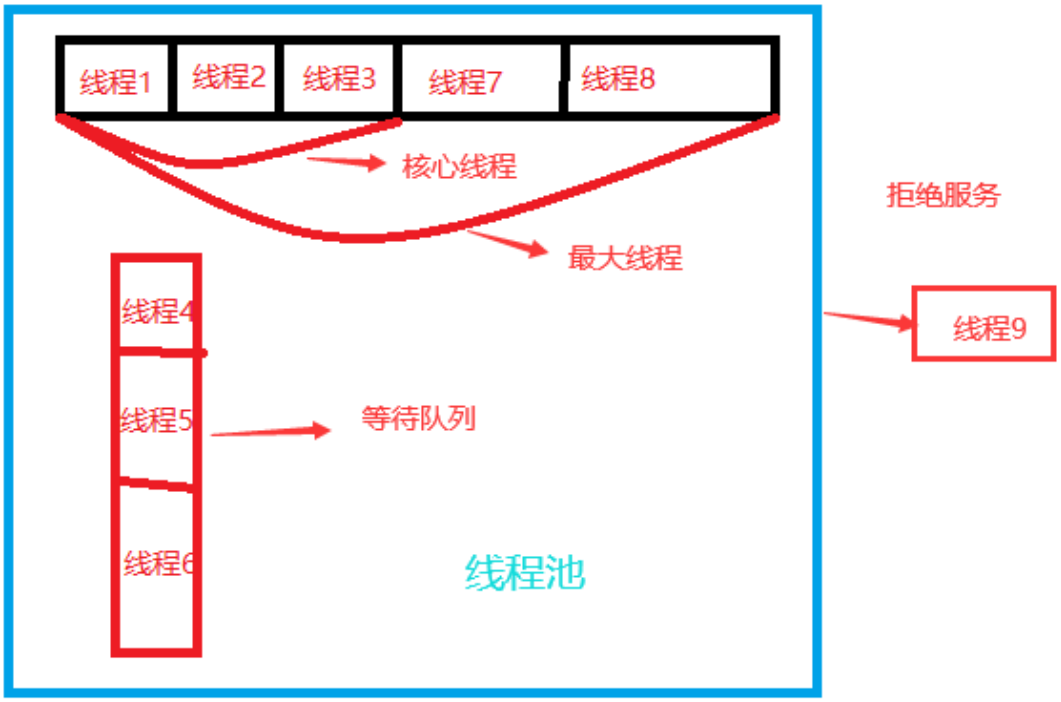
8) 匿名内部类说明

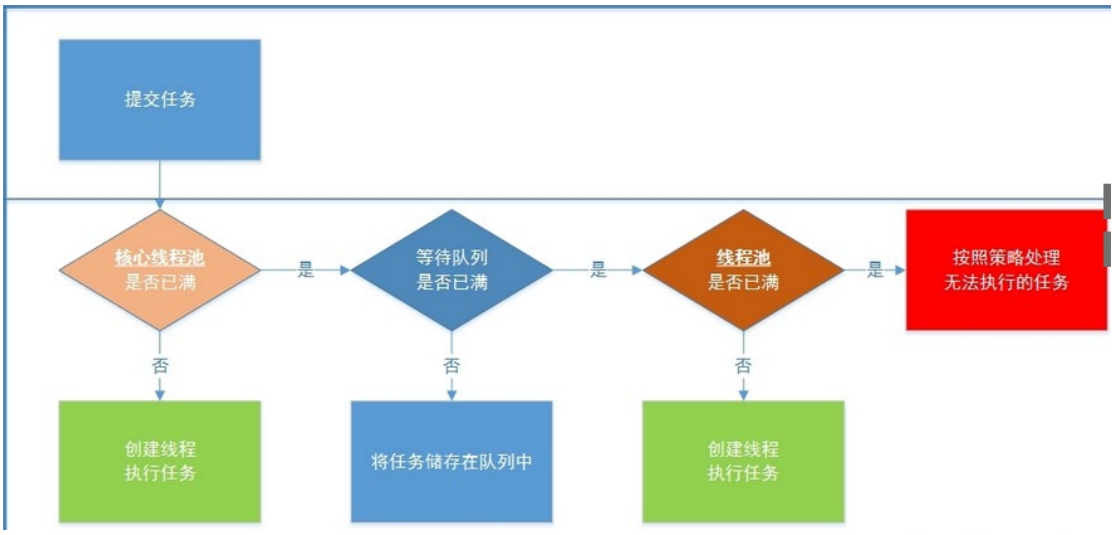
9) 当我们在进行并发编程时，如果没有任何限制的话，已创建的线程是不会被复用的，使用过的线程依旧会占线程资源，随着请求的增多，线程会越来越多，直至达到上限，以至死机。

线程池可以规定线程的上限且提高线程的复用率。

使用方法对 ThreadPoolExecutor 类进行实例化即可，同时规定核心线程数、最大线程数，等待队列的数量，然后向线程池中加入实现 runnable 接口的类，run 方法中来执行并发内容。

当执行并发时，会优先在核心线程数执行，当线程数量超过核心线程数时，会将其他线程放在等待队列中，当等待队列也满了，就会将其他线程放到最大线程的非核心线程上，当最大线程已满，就会抛出异常，拒绝执行。所以如果并发量特别高的时候，可以采用 MQ 来代替多线程。





10) 由于 ThreadPoolExecutor 使用比较复杂，JDK 用 ThreadPoolExecutor 封装了三个三个使用较为方便的类

newCachedThreadPool	**无界线程池，可以进行自动线程回收　线程数目没有限制**
newFixedThreadPool	固定线程池　线程池中线程数固定，开发中一般使用这个
newSingleThreadExecutor	单一线程池　所有任务均在一个线程中执行

我们只需要将其实例化，用实现 runnable 接口类的 run 方法携带并发任务，然后将该任务放入线程池中即可，线程池就会开启多线程进行并发操作

11) @Async 注解是对线程池的封装，需要单独创建一个类，注入 spring 中，修饰的方法可以对方法内调用的其他方法进行并行异步操作。

12) 什么是线程安全

当多个线程同时对一共享数据进行写操作时，会发生脏读现象，我们称这个数据不是线程安全的。

13) 乐观锁和悲观锁

为了解决线程安全问题，就产生了乐观锁和悲观锁

A. 乐观锁：

a) 概念：认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发生了冲突，则返回用户错误的信息，让用户决定如何去做或者程序自动去试。

b) 乐观锁又叫无锁，自旋锁、轻量级锁

c) 一般的实现方式：为共享数据添加版本或时间戳，cas 原子实现

d) 添加版本或时间戳的实现原理：

1.先获取数据 A 以及数据的版本α；

2.对获取的数据进行业务处理数据变为 B

3.获取处理后数据的版本γ与α比较，如果一致，更新共享数据为 B，同时更新版本为α+1；

```
ic static void addTo100() throws InterruptedException {
while (data1.get("value-1") < 100){
    Integer value = data1.get("value-1");
    Integer version = data1.get("version-1");
    System.out.println(Thread.currentThread().getName()+"
    //阻塞1秒  模拟处理业务
    Thread.sleep( millis: 1000);
    value++;
    //然后去更新
    if(version.equals(data1.get("version-1"))){
        data1.put("value-1",value);
        //版本号+1
        data1.put("version-1",version+1);
        System.out.println("======"+Thread.currentThread()
    }else{
        System.out.println("******"+Thread.currentThread()
    }
}
```

获取数据的版本和值

业务处理

和数据中的版本进行比对，如果一直更新数据并版本+1

e) 但是这种方法从比对版本到更新数据并不是原子操作，对于特别高的并发来说，很有可能多个线程同时在进行比对，多个线程发现版本号都一致，然后多个线程对该数据进行了不同的更新，版本号都加一，所以不建议

f) Cas 原子实现

g) JDK 提供了原子类来便于进行乐观锁的实现

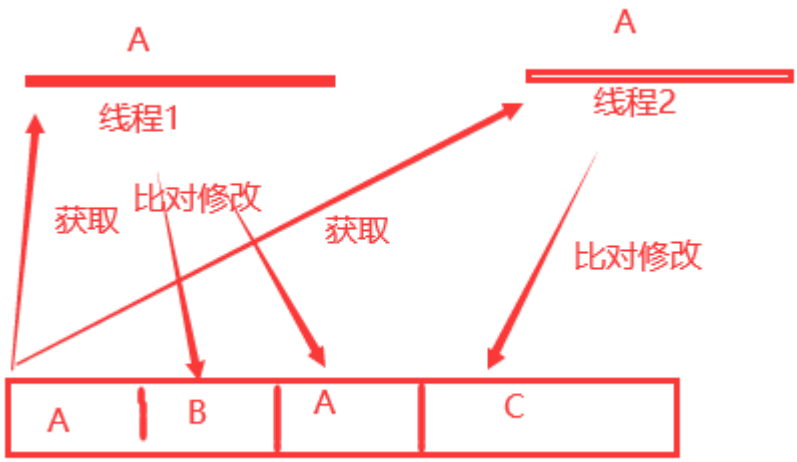
h) 具体原理：每一种原子类都提供了 compareAndSet 方法来进行数据的比对和更新，底层封装了汇编语言的线程锁，保证了比较和数据更新只能由一个线程来操作，这也叫原子性。

```
//创建int类型的原子类，设置初始值为0
static AtomicInteger ato = new AtomicInteger( initialValue: 0);
//创建多线程下的并发操作   将   原子值加至100
public static void addTo100() throws InterruptedException {
    while(ato.get() <100){
        int oldValue = ato.get();
        int newValue = oldValue + 1;
        Thread.sleep( millis: 1000);
        if(ato.compareAndSet(oldValue,newValue)){
            System.out.println("======"+Thread.currentThread().g
        }else{
            System.out.println("******"+Thread.currentThread().g
        }
    }
}
```

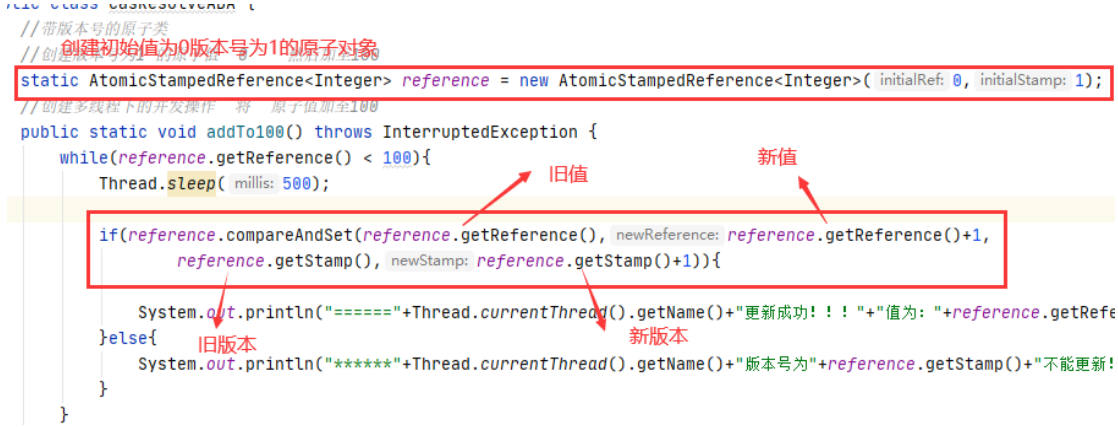
获取旧值和创建新

通过对原子类中老值的比较，如果一致就更换为新值，如果不一致，就放弃操作

i) ABA 问题：上述操作中虽然保证了比较和更新过程不被其他线程打扰，但是只是通过比较旧值来修改新值，会存在 ABA 问题，比如有两个线程同时获取了原子类的值 A，线程 1 速度比较快，率先将其修改为 B，然后又修改为 A 之后，线程 2 才拿着开始获取到的旧值 A，去原子类中进行比对修改，修改为 C，但是这个 A 已经不是旧值 A 了，虽然值一样，但是是被别的线程操作过后的，一般情况不会出问题



j) 解决 ABA 问题，JDK 提供了带有版本的原子类 AtomicStampedReference，可以对数据进行版本管理。首先版本比对以及数据更新是不会被其他线程干扰的，通过比对来将旧值替换为新值。



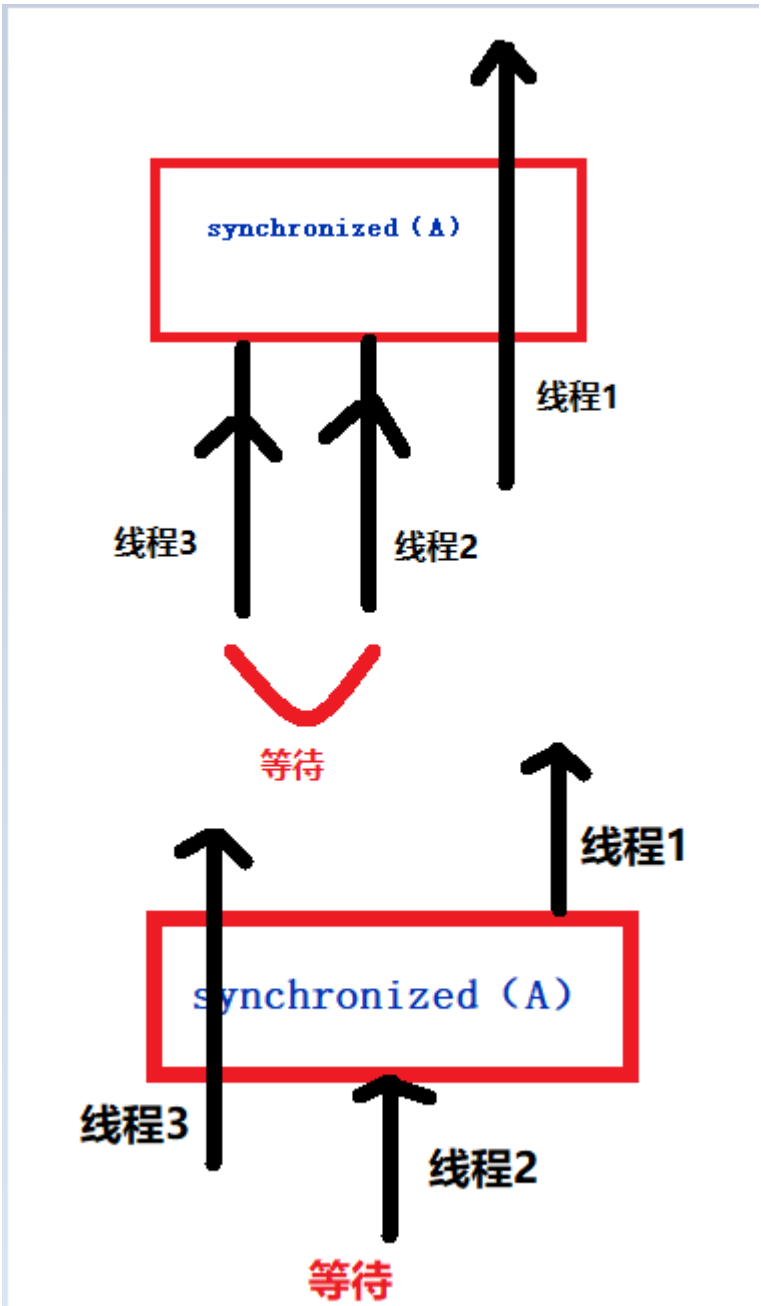
B. 悲观锁

a) 概念：对数据被外界修改持保守态度，因此在整个数据处理过程中将数据处于锁定状态，被一个线程释放后，才允许另一个线程访问。悲观锁相对于乐观锁效率会变低。

b) 悲观锁又叫互斥锁、同步锁、重量级锁。

c) 悲观锁的实现：synchronized，Lock

d) 当线程 1 经过被互斥锁修饰的方法 A 时，我们说，线程 1 获取到 A 的锁，其他线程就无法获取到 A 的锁，不能对 A 进行访问。当线程 1 执行完 A 方法后，线程 1 释放 A 锁，其他线程会随机的得到 A 锁。



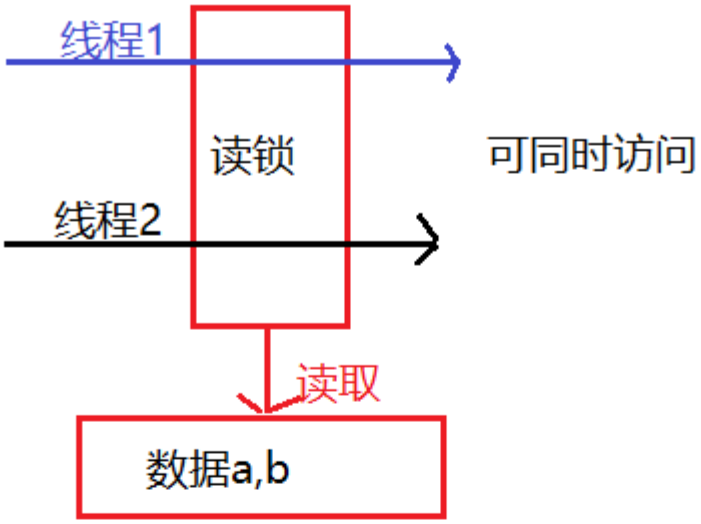
e) Synchronized 是封装好可以自动释放锁的同步锁，但是使用比较局限，只能对某变量或某方法整体进行加锁。

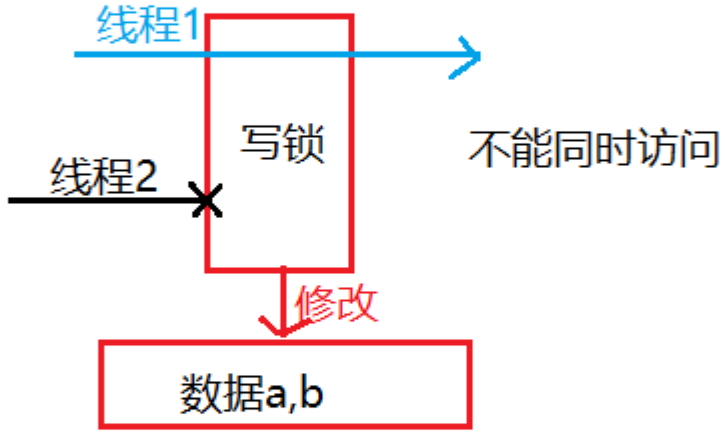
```
public static void main(String[] args) {
    synchronized (num){          对变量进行加锁
        System.out.println("对num变量加了同步锁！！！");
    }
    synchronized (c()){          对方法进行加锁
        System.out.println("对C方法加了同步锁！！！");
    }
}
```

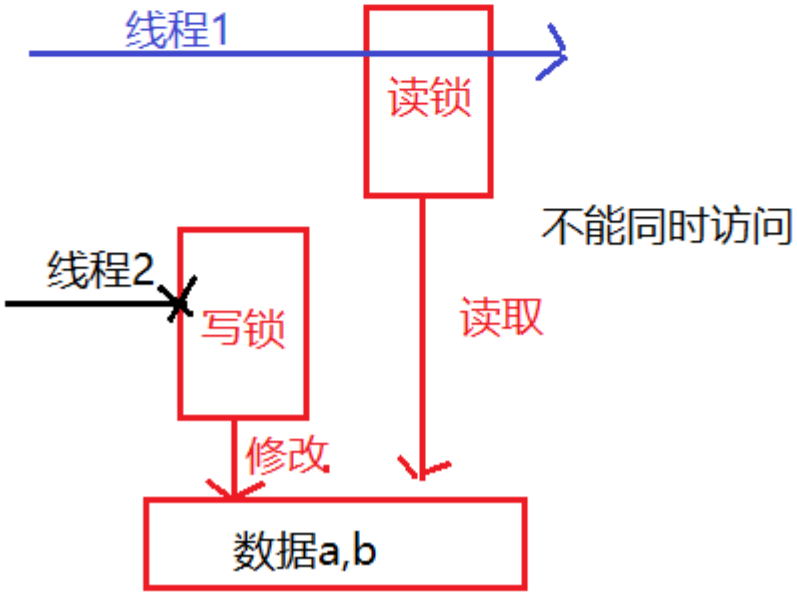
f) Lock 相对 synchronized 使用更灵活，可以对方法中的某片段代码进行加锁，但是要手动释放锁。

```
//创建锁
public static void letLock() {

    System.out.println(Thread.currentThread().getName() + "已准备进入锁！！");

    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + "已进入锁，并将数
        Thread.sleep( millis: 500);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        System.out.println(Thread.currentThread().getName() + "释放锁！！");
        lock.unlock();
    }
}
```


g) 下面进入难点，也是重点，Lock 还提供了读写锁，原则是读读共享，读写、写写互斥，是说读锁与读锁可以共享锁里的操作，但是读锁与写锁，写锁与写锁，只能有一个线程先去执行，另一个线程等待。

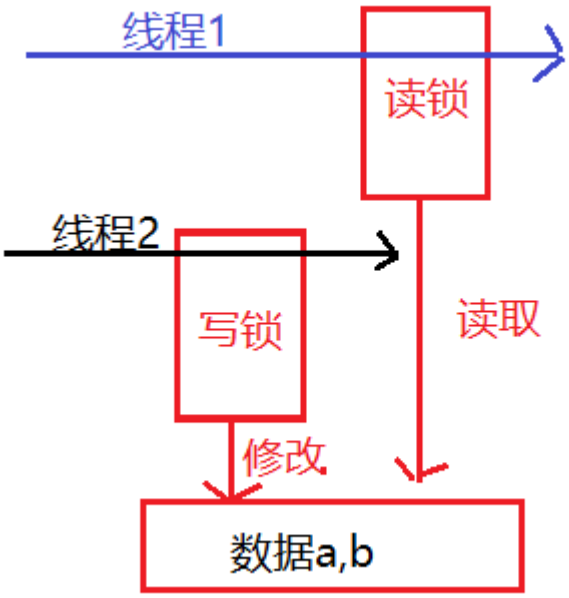

线程1
读锁
可同时访问
线程2
读取
数据a,b


线程1
写锁
不能同时访问
线程2
修改
数据a,b


线程1
读锁
不能同时访问
线程2
写锁
读取
修改
数据a,b

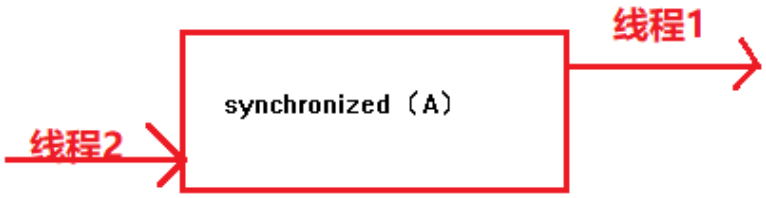
```java
public static void readLock(){
    //开启读锁
    lock.readLock().lock();
    try {...}catch (Exception e){...}finally {
        //解锁
        System.out.println(Thread.currentThread().getName()
        lock.readLock().unlock();
    }
}
//创建解锁
public static void writeLock(){
    //开启写锁
    lock.writeLock().lock();
    try {...}catch (Exception e){...}finally {
        //解锁
        System.out.println(Thread.currentThread().getName()
        lock.writeLock().unlock();
    }
}

//创建多个线程
public static void main(String[] args) {
    //创建10个线程去读写读
    for(int i = 0; i < 10 ;i++){
        new Thread(()->{
            readLock();
            writeLock();
            readLock();
            }, name: "线程==="+i).start();
    }
    //创建10个线程去写读写
    for(int i = 0; i < 10 ;i++){
        new Thread(()->{
            readLock();
            readLock();
            writeLock();
            }, name: "线程==="+i).start();
    }
}
```


h) 读锁的存在是为了防止正在修改的数据被读到，造成脏读，如果读锁失去作用，即读读、读写共享，那么在线程 1 写到一半的时候线程 2 去读，线程 1 写完的时候，线程 2 再去读，就会造成两次读到的数据不一致。

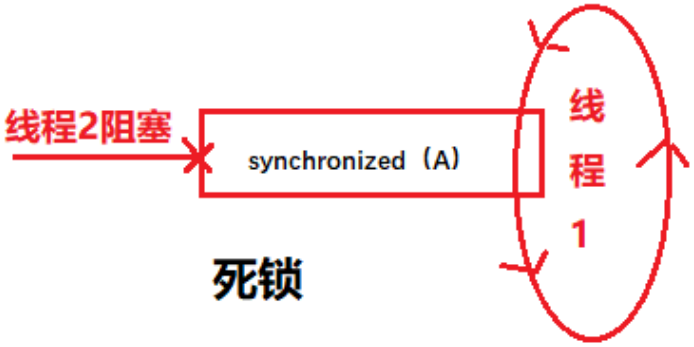


14）死锁：

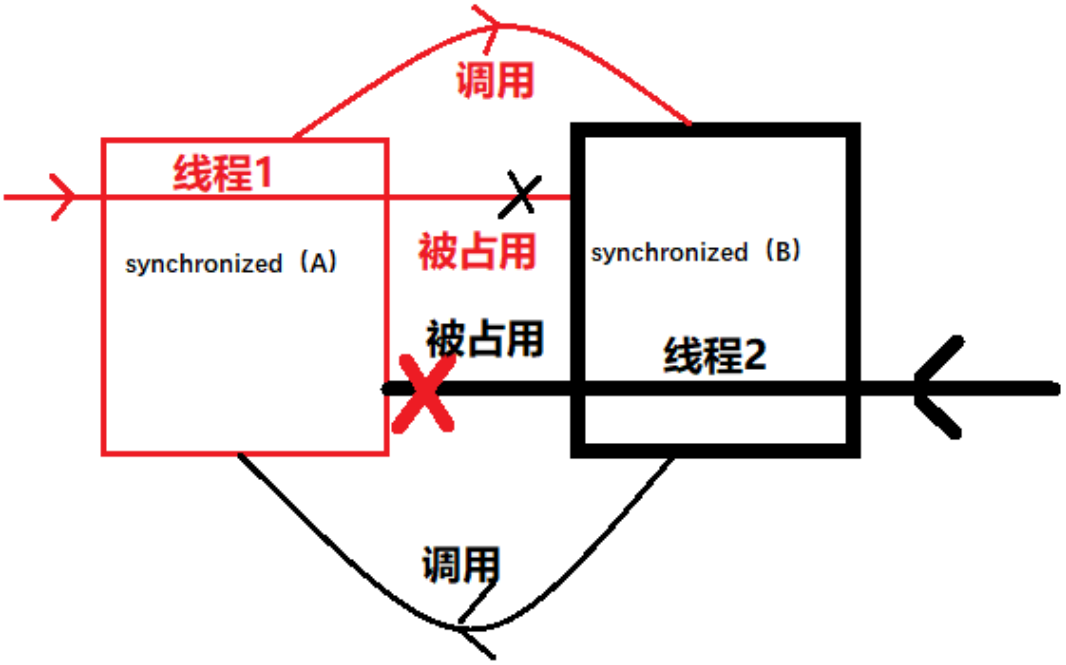
- 同步锁（synchronized）只有在某线程上执行完后，才能被其他线程调用
- 一个同步锁可以调用另一个同步锁，这叫同步锁的重入性，且多个同步锁组成的调用链叫做重入锁。
- 当某线程 A 要去调用某同步锁 a，但是该同步锁 a 被其他线程长时间占用,导致该线程 A 一直处于等待状态，叫做死锁。
- 死锁最经典的就是两个相反的重入锁，同时被调用导致。例如重入锁 A 是同步锁 a 去调用同步锁 b，重入锁 B 是同步锁 b 去调用同步锁 a，当线程 1 和线程 2 分别同时调用重入锁 A 和重入锁 B，同步锁 a 被线程 1 占用，同步锁 b 被线程 2 占用，然后线程 1 去调用同步锁 b，线程 2 去调用同步锁 a，但是他们都被占用了，造成了死锁。注意：一定是同步锁 a 去调用同步锁 b，或者同步锁 b 去调用同步锁 a，而不是同步锁 a 执行完后再去调用 b，因为不相互调用的话，被线程 1 用完后就释放掉了，线程 2 仍然可以使用，造不成死锁。详见 demo。



同步锁一次只允许一个线程调用

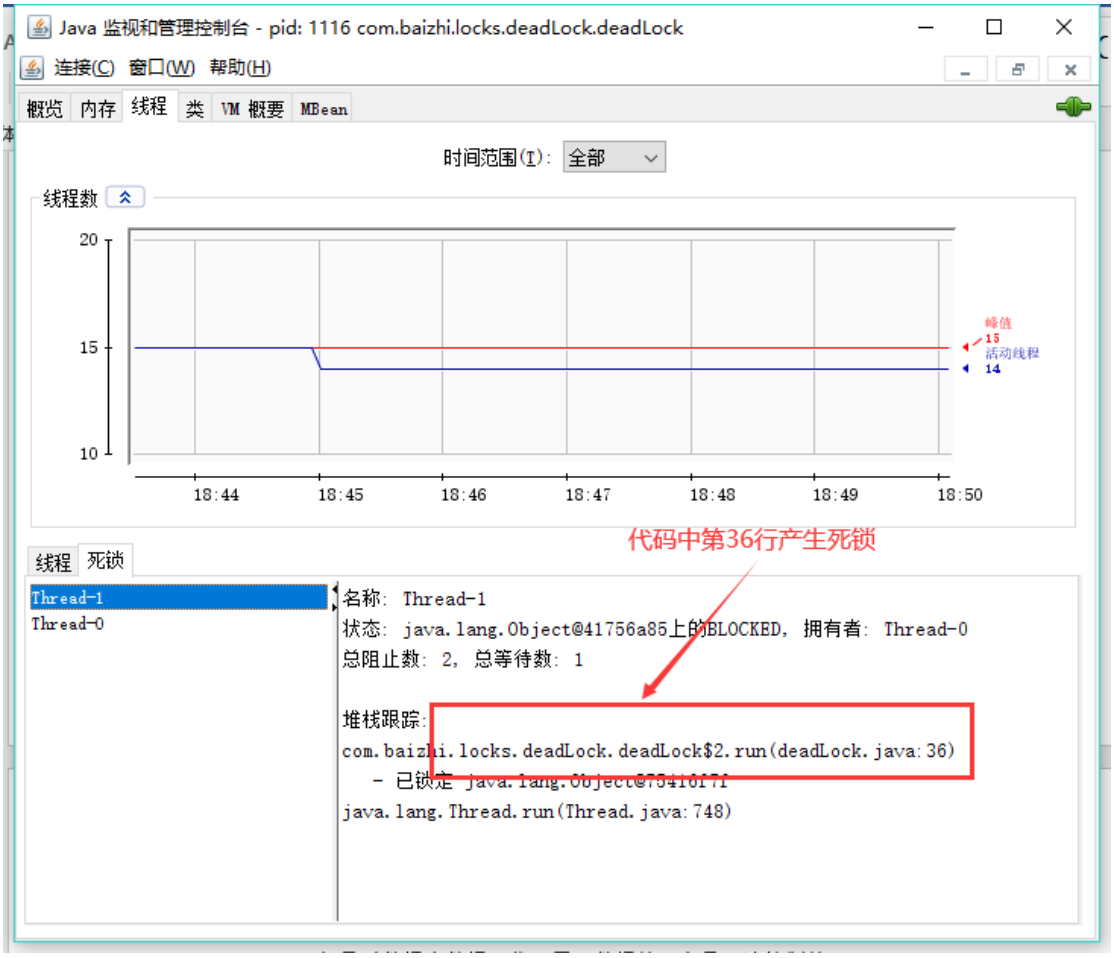


死锁



死锁的经典案例

- 死锁的定位：用 jconsole 检测死锁，即可定位到产生死锁代码的具体位置。



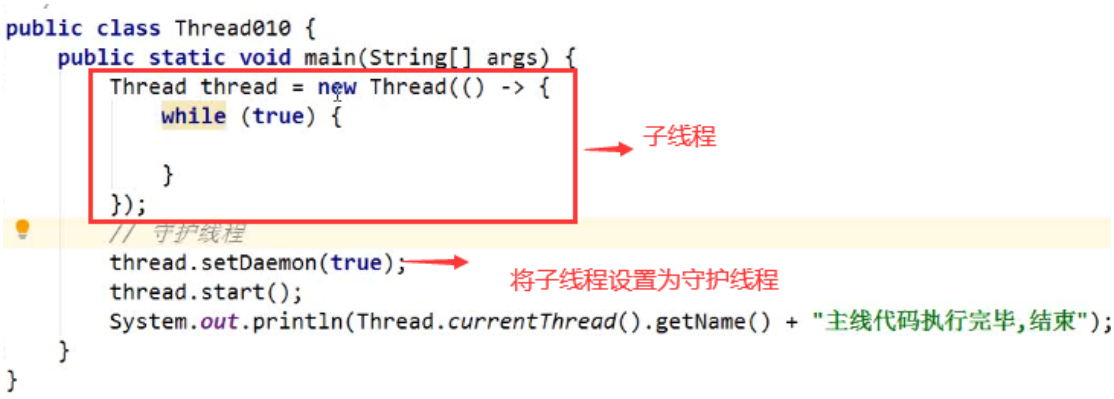
15) 事务、线程和锁

- 事务解决的是并发的数据库的整体读写操作之间共享数据的隔离问题，注意仅仅是对数据库数据，代码层面数据的共享是无法控制的。
- 被事务修饰的方法依旧可以被多个并发线程访问，且会产生多个事务。
- 而被悲观锁修饰的方法，同一时间只能有一个线程来访问，如果有事务注解，只能产生一个事务，但是可能会和系统中的其他事务产生并发。
- 容易出现的误区是事务的原子性和隔离性，让人误以为事务只能被单个线程访问，事务的原子性指的是被事务修饰的一系列数据库操作是一个整体，隔离性指的是一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。注意是事务级别的隔离，是数据库层面的数据，而不是线程，而且人家允许并发。
- 总结：事务是解决数据库层面的并发，锁解决的是代码层面的并发。

16) 用户线程和守护线程

用户线程：java 默认情况下创建的线程都是用户线程，用户线程不会随着主线程的停止而停止，主线程结束用户线程还可以执行

守护线程：守护线程会随着主线程的结束而结束，线程有一个守护线程属性，默认为 false，将其设置为 true 即为守护线程。例如我们的垃圾回收 GC 就是守护线程



17) 面试的时候，可以说曾经用线程池写过操作日志，记住，千万不要说是用的 Thread 写的操作日志，因为这样线程是无限创建且不可复用的，系统会越来越卡，直至死机

18) Wait、notify、sleep、join 区别

Wait	阻塞当前线程，并释放锁且线程中必须存在锁，一般用在 synchronized 的代码块中
Sleep	阻塞当前线程，和锁没有关系
notify	唤醒进程中一个阻塞的线程，同样需要锁的配合，一般用在 synchronized 的代码块中
Join	实现线程先后执行的顺序，阻塞当前线程，直到指定线程执行完后，再执行当前线程。

```
@Override
public void run() {
    while (true) {
        try {
            synchronized (res) {
                if (!res.flag) {
                    res.wait();
                }
                System.out.println(res.userName +
res.userSex);
                res.flag = false;
                res.notify();
            }
        } catch (Exception e) {

        }

    }
```

锁

释放指定的锁res，并线程阻塞

唤醒一个阻塞的线程

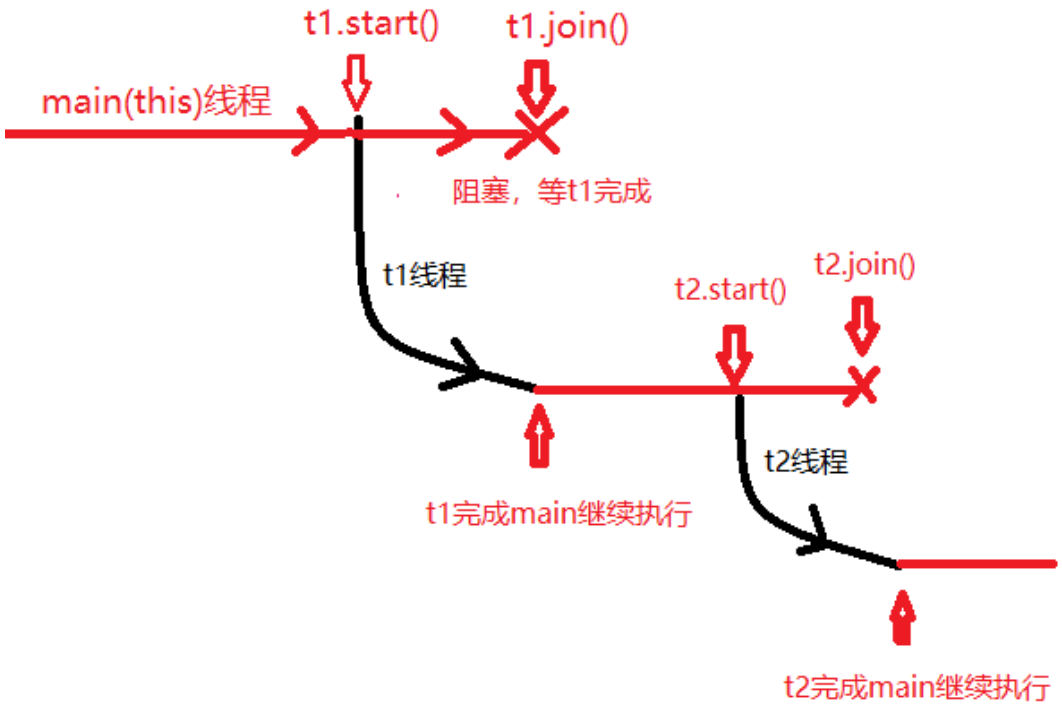
Join 具象理解

```
public static void main(String[] args) {
    try {
        Thread t1 = new Thread(new Thread009(), name: "t1");
        Thread t2 = new Thread(new Thread009(), name: "t2");
        Thread t3 = new Thread(new Thread009(), name: "t3");
        t1.start();
        t1.join();//t1.join
        t2.start();
        t2.join();
        t3.start();
        t3.join();
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName() + ","
        }
    } catch (Exception e) {
```

相当于this.t1.join

this当前线程main,t1线程执行完后，再执行main线程

图形表示


t1.start()
t1.join()
main(this)线程
阻塞，等t1完成
t1线程
t2.start()
t2.join()
t2线程
t1完成main继续执行
t2完成main继续执行