1.连接池

为什么使用连接池：

比较常见的就是数据库的连接池。连接池的目的是避免频繁的创建、关闭连接，重复的利用一个连接，减少资源消耗。比如jdbc创建一个连接的时候，就要先注册驱动，根据ip,创建socket连接，再得到连接结果。

bonecp的实现原理：

Bonecp中有分区，默认是1个，建议是3-4个；每个分区有最大连接数（默认是2可以设置为150）和最小连接数（默认是0，可以设置为5）；未使用的连接的最大存活时间（默认60分，设置为30分）；检查连接池中空闲连接的存活时间（默认240分，可设置为60分）。也就是每到一定的时间，就会去检查连接池中的连接的空闲时间，如果超过最大空闲时间，那么就会重新建立这个连接。这么做的原因是因为mysql本身也有个时间校验一个连接的空闲时间，如果大于空闲时间（默认是8个小时）就会把这个连接取消掉，这样的话连接池如果不去检验空闲时间的话，当数据库把这个连接给kill了，连接池还是不知道，还把这个连接返回给应用使用的话就会报错了。

每个分区的最大连接数乘以分区个数要小于mysql的最大连接数。

实现连接池的思路：

连接池的创建要同步，可以用一个LinkedList或者Vector来存连接，需要设置最大连接数，最小连接数，要有获取连接，归还连接的方法。还需要建立线程来检测每个连接的使用情况。

1. 同步方法（能够实现大范围的原子性，java中只有基本的赋值操作等才具有原子性）

Synchronized可以修饰方法或者是代码块，synchronized锁住的是调用的对象（静态方法的调用对象是类本身），也是后面括号里面跟着的内容，如果同步方法是非静态的，那么不同的对象实例是可以自由调用同步方法的，这个同步只针对相同的一个实例；如果同步方法是静态的，因为这个方法全局也就只有一个，那么对于不同的实例都是同步的

Synchronized与lock的区别

Synchronized是java内置特性，lock是一个类

采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。所以使用lock需要try catch finally,把释放锁的操作放在finally中

Lock可以手动控制等待，不必因为线程阻塞而一直等待下去

lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

　tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

　　tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

ockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

1）Lock是一个接口，主要的实现类有ReentrantLock，ReentrantReadWriteLock而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　5）Lock可以提高多个线程进行读操作的效率。

1. 线程相关问题

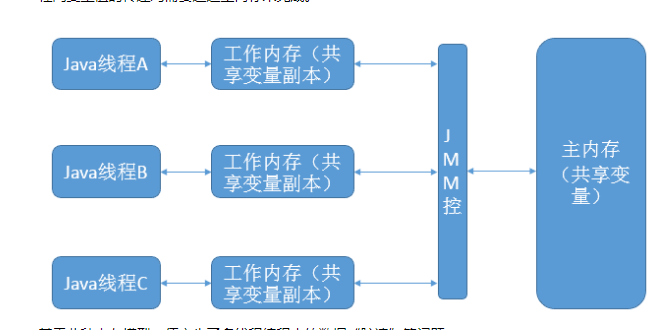
创建线程的3种方式

1. 继承Thread类 （继承了Thread类就不能再继承其他的类了）
2. 实现Runnable接口
3. 实现Callable接口（这个可以有返回值的）

调用线程要用start()方法，这样会启动一个线程

Volatile关键字

首先要理解java的内存模式



这样的模式会导致数据的脏读

Volatie是修饰成员变量的

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

多线程要解决的问题就是保证程序的原子性，可见性和有序性（编译器或者是jvm会进行指令重排）。这些方面利用Synchronized能够很好的解决（就是保证单一线程执行，但是效率比较低），利用volatie可以实现可见性，保证修饰变量的有序性

**public** **class** Test {

**public** **volatile** **int** inc = 0;

**public** **void** increase() {

        inc++;

    }

**public** **static** **void** main(String[] args) {

**final** Test test = **new** Test();

**for**(**int** i=0;i<10;i++){

**new** Thread(){

**public** **void** run() {

**for**(**int** j=0;j<1000;j++)

                        test.increase();

                };

            }.start();

        }

**while**(Thread.activeCount()>1)  //保证前面的线程都执行完

            Thread.yield();

        System.out.println(test.inc);

    }

}

自增的操作不是原子性的，他的输出结果是小于1000\*10的

Volatile应用场景，如单例模式中修饰静态属性

**class** Singleton{

**private** **volatile** **static** Singleton instance = **null**;

**private** Singleton() {

    }

**public** **static** Singleton getInstance() {

**if**(instance==**null**) {

**synchronized** (Singleton.**class**) {

**if**(instance==**null**)

                    instance = **new** Singleton();

            }

        }

**return** instance;

    }

}

用volatile修饰的原因是因为instance = **new** Singleton();是非原子性的，他进行了3个操作，

1. 给 instance 分配内存 2. 调用 Singleton 的构造函数来初始化成员变量 3. 将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

想象一下，第一个线程先执行3再执行2，这样在3执行完之后2还没有执行的时候第二个线程进来了，这是他获取到的instance是非null的，但是确实没有进行初始化的，可是第二个线程是会直接返回这个instance的，那么就不合理了。

1. 锁(这样简单的理解还是不够的，我们还需要知道他们内部的实现原理以及如何自己去实现一种数据结构，把这些锁的功能加上去)

公平锁/非公平锁

公平锁是指多个线程在等待同一个锁，必须按照申请的先后顺序来依次获取锁，好处是等待的线程不会饿死，坏处是效率较低，他可以用new ReentrantLock(true)实现。

自旋锁

多线程下，当一个线程遇到一个同步的方法时，需要等待，这时CPU会将这个线程挂起，转向去执行另一个线程，但是有些时候同步方法的执行时间是很短的，一遇到同步方法就挂起线程转向另一个线程，CPU这样操作反而浪费的时间比等待同步方法执行完还要长，所以我们就用自旋锁，让CPU再等待一段时间，而不是马上就去转换线程。

偏向锁、轻量级锁和重量级锁（等以后再研究）

锁消除

就是对象不是共享的，但是对象中的方法是同步的，这时虚拟机会进行判断，即使是同步的方法也不会生效的，这个锁就自然消除了。所以说锁是对于共享数据而言的，对于局部变量，根本不会生效的。

可重入锁

可重入锁，也叫做递归锁，指的是同一线程外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。在JAVA环境下 ReentrantLock 和synchronized 都是可重入锁。可重入锁最大的作用是避免死锁。

类锁和对象锁

类锁：在方法上加上static synchronized的锁，或者synchronized(xxx.class)的锁。

对象锁：加锁的方法是非静态方法，或者是锁括号里面的是this或者是一个Object的具体对象

悲观锁和乐观锁

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。   
乐观锁：假定不会发生并发冲突，只在提交操作时检测是否违反数据完整性。（使用版本号或者时间戳来配合实现）

Synchronized就是一个悲观锁，乐观锁简单的实现就是对于写操作设置同步，利用双重锁机制（类似于实现单例模式），设置成员变量文本号，文件对象，每次写之前要检查版本是否一致，如果一致进入到同步写的代码块，再次比较版本是否一致，如果一致就写，不一致就告诉操作者，然后重新执行操作。

1. **public** **class** OptimThread **extends** Thread {
3. // 文件版本号
4. **public** **int** version;
5. // 文件
6. **public** String file;
8. **public** **void** run() {
9. // 1. 读取文件
10. String text = read(file);
11. println("线程"+ getName() + "，文件版本号为：" + OptimLockMain.getVersion());
12. println("线程"+ getName() + "，版本号为：" + getVersion());
13. // 2. 写入文件
14. **if**(OptimLockMain.getVersion() == getVersion()){
15. println("线程" + getName() + "，版本号为：" + version + "，正在执行");
16. // 文件操作，这里用synchronized就相当于文件锁
17. // 如果是数据库，相当于表锁或者行锁
18. **synchronized**(OptimThread.**class**){
19. **if**(OptimLockMain.getVersion() == **this**.version){
20. // 写入操作
21. write(file, text);
22. // 更新文件版本号
23. OptimLockMain.updateVersion();
24. **return** ;
25. }
26. }
27. }
28. // 3. 版本号不正确的线程，需要重新读取，重新执行
29. println("线程"+ getName() + "，文件版本号为：" + OptimLockMain.getVersion());
30. println("线程"+ getName() + "，版本号为：" + getVersion());
31. System.err.println("线程"+ getName() + "，需要重新执行。");
32. }

读写锁

读写锁是一个资源能够被多个读线程访问，或者被一个写线程访问但不能同时存在读线程。Java当中的读写锁通过ReentrantReadWriteLock实现。

private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

 public void get(Thread thread) {

        rwl.readLock().lock();

        try {

            long start = System.currentTimeMillis();

            while(System.currentTimeMillis() - start <= 1) {

                System.out.println(thread.getName()+"正在进行读操作");

            }

            System.out.println(thread.getName()+"读操作完毕");

        } finally {

            rwl.readLock().unlock();

        }

    }

如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。

如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

互斥锁

所谓互斥锁就是指一次最多只能有一个线程持有的锁。在JDK中synchronized和JUC的Lock就是互斥锁。

死锁

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，他们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足一下4个条件：

互斥条件：一个资源每次只能被一个进程使用。

请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。

循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序做操作来避免死锁。

思路是创建两个字符串a和b，再创建两个线程A和B，让每个线程都用synchronized锁住字符串（A先锁a，再去锁b；B先锁b，再锁a），如果A锁住a，B锁住b，A就没办法锁住b，B也没办法锁住a，这时就陷入了死锁

synchronized(DeadLock.obj1){

System.out.println("Lock1 lock obj1");

Thread.sleep(3000);//获取obj1后先等一会儿，让Lock2有足够的时间锁住obj2

synchronized(DeadLock.obj2){

System.out.println("Lock1 lock obj2");

}

}

synchronized(DeadLock.obj2){

System.out.println("Lock2 lock obj2");

Thread.sleep(3000);

synchronized(DeadLock.obj1){

System.out.println("Lock2 lock obj1");

}

}

数据库中常用的锁

1. Threadlocal类
2. String StringBuild StringBuffer
3. 反射\
4. Spring框架如何扩展，有哪些常用的扩展功能（需要理解spring框架的实现原理，bean，aop,代理，反射）
5. Cdn相关知识（缓存问题）
6. 常见的加密算法

微信小程序、顺丰、支付宝、微信

1. 面试中常见的小算法题目
2. 数组去重

一般情况下可以先声明一个集合去装非重复的数字，先将原数组中的第一个元素放入到集合中，从第二个开始遍历原先的数组，判断原先数组中的元素在当前集合中是否存在，如果不存在就加入进去

1. String[] array = {"a","b","c","c","d","e","e","e","a"};
2. List<String> list = **new** ArrayList<>();
3. list.add(array[0]);
4. **for**(**int** i=1;i<array.length;i++){
5. **if**(list.toString().indexOf(array[i]) == -1){
6. list.add(array[i]);
7. }
8. }

或者利用set是不重复的，直接将数组加入到set中，就能自动去重了，或者是利用hashset

1. String[] array = {"a","b","c","c","d","e","e","e","a"};
2. Set<String> set = **new** HashSet<>();
3. **for**(**int** i=0;i<array.length;i++){
4. set.add(array[i]);
5. }

如何将一个唯一的数字转换为另一个唯一的数字？

1. app接口设计规则（主要从请求路径，请求方式，参数，返回值（返回值该如何设计））

Pojo的方法中最好不要出现is...() 因为这样会把is后面的参数当做pojo的一个属性输出的，返回给前端的时候就会多了一个属性

1. pojo设计小技巧（就是为了满足前端某些返回值的格式需要，我们需要返回某些特定的参数）

可以增加get...()方法，get后面的参数会作为一个属性输出，也可以是is...()效果应该是一样的

有些属性我们不是在数据库表中存在的属性，但是需要返回给前端，这样可以增加@Transient注解

在转换为json格式的时候，有些属性需要改为其他的属性名称输出，我们可以添加@jsonProperty注解，再赋值其他的属性名称，这样可以不用破坏pojo，耦合性低

1. 调用第三方接口的开发（支付宝，微信，七牛，极光，顺丰，网易，管易）,总结主要包括发送请求的方式，参数分装，如何加密，如何处理返回值
2. 数据库表设计（频繁修改的表，分库分表。。。这是一个很庞大的问题，感觉需要请教专业的DBA）
3. 解决springmvc传参乱码问题

17.二进制与16进制相互转换