异步编程模型：t1线程执行t1的，t2线程执行t2的，两个线程之间谁也不等谁。

同步编程模型：t1线程和t2线程执行，当t1线程必须等t2线程执行结束之后，t1才可以执行，这就是同步编程模型。

1.dom4j解析xml文件

public class Test01 {  
 */\*\*  
 \* dom4j解析xml文件  
 \*/* public static void main(String[] args) throws DocumentException {  
 //获取dom4j的读取流对象  
 SAXReader reader = new SAXReader();  
 //拿到文档对象  
 Document document = reader.read("src/main/resources/application.xml");  
 //拿到根节点(书架）  
 Element root = document.getRootElement();  
 //拿到所有的书节点  
 List<Element> book = root.elements("书");  
 //拿到某个节点的内容  
 Element element1 = book.get(1);  
 //拿出一个数据  
 System.*out*.println(element1.elementText("书名"));  
 }  
}

1. 用正则表达式判断一个输入的身份证号

public class Test01 {  
 */\*\*  
 \* 判断输入的数据为正确的身份证号码，正确打出出生日期  
 \*/* public static void main(String[] args) {  
 System.*out*.println("请输入身份证号：");  
 String date = new Scanner(System.*in*).nextLine().replace(" ", "");  
 Pattern pattern = Pattern.*compile*("^(\\d{15}|\\d{17}[\\dx])$");  
 Matcher matcher = pattern.matcher(date);  
 //符合为true，否则为false  
 boolean flag = matcher.matches();  
 if (flag) {  
 System.*out*.println("输入合法，生日是：" + date.substring(6, 14));  
 } else {  
 System.*out*.println("输入不合法！");  
 *main*(args);  
 }  
  
 }  
}

3.jdk1.8的新特性

1.lambda表达式解决匿名内部类问题

2.接口中可增加default修饰的方法

3.HashMap底层为数组+链表，当链表的长度超过8之后，会转换为红黑树。

4.匿名内部类使用的局部变量自动加final修饰。

4.volatile关键字

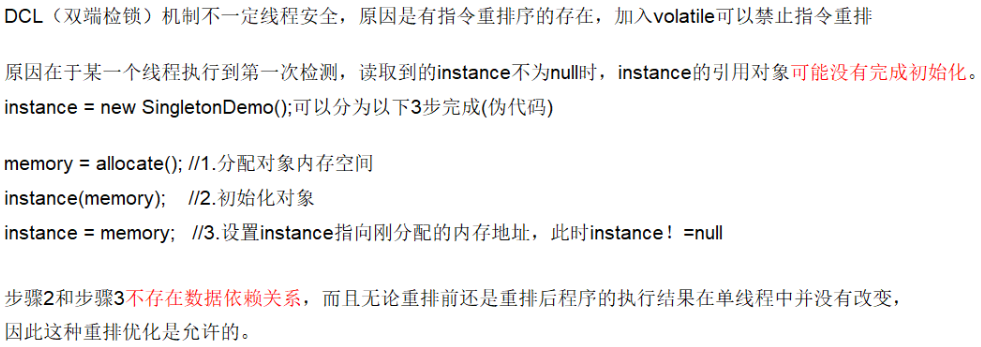
1.保证可见性（一个线程修改有某个对象中的属性值后，另一个线程中可以立刻获取到这个改变后的值）

package com.test;  
  
  
class MyData {  
 */\*\*\*  
 \* 测试的数据data  
 \*/* private volatile int data = 0;  
  
 public void addData() {  
 data = 60;  
 }  
  
 public int getData() {  
 return data;  
 }  
}  
  
public class Test01 {  
 public static void main(String[] args) {  
 */\*\*  
 \* 这个位置只能放个对象，直接放值的话会自动被  
 \* final修饰，导致下面新的线程无法改变它，即无法  
 \* 测试  
 \*/* MyData myData = new MyData();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 */\*\*  
 \* 让睡一秒，以保证主线程先走到下面的while  
 \*/* try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 myData.addData();  
 }  
 }).start();  
 */\*\*  
 \* 当新开的线程改变了值被主线程看见了，这个循坏才会跳出  
 \*打出最下面的一句话  
 \*/* while (myData.getData() == 0) {  
 }  
 */\*\*  
 \* 如果变量data没有加volatile关键字，主线程看不见新线程所  
 \* 改变后的值，会一直循环上面的while，打不过结果。  
 \* 如果加了data加了volatile关键字，主线程课可见，则打出一下的  
 \* 值为60  
 \*/* System.*out*.println("data=======" + myData.getData());  
 }  
  
  
}

2.不保证原子性（对一个值进行多线程并发修改的时候，有可能导致数据丢失，本来加到100，实际小于100，可以采用原子性的包装类避免这种情况的产生）

class Person {  
 int number = 0;  
 AtomicInteger atomicInteger = new AtomicInteger();  
  
 public void add() {  
 number++;  
 }  
  
 public void atomicAdd() {  
 atomicInteger.incrementAndGet();  
 }  
}  
public class Demo01 {  
 public static void main(String[] args) {  
 Person person = new Person();  
 for (int i = 0; i < 20; i++) {  
 new Thread(() -> {  
 for (int j = 0; j < 1000; j++) {  
 try {  
 Thread.*sleep*(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 person.add();  
 person.atomicAdd();  
 }  
 }).start();  
 }  
 //Thread.activeCount()判断有几个线程在跑，一般是jvm和gc  
 while (Thread.*activeCount*() > 2) {  
 //大于2个线程就礼让线程，不让主方法跑下去  
 Thread.*yield*();  
 }  
 */\*\*  
 \* 1.第一个没有保证原子性导致出来的结果小于20000  
 \* 2.第二个采用的原子操作包类AtomicInteger解决了这一问题的结果为20000  
 \*/* System.*out*.println("number=" + person.number);//number=19989  
 System.*out*.println("atomicInteger=" + person.atomicInteger);//atomicInteger=20000  
 }  
}

3.禁止指令重排（单线程中没有依赖的可以调整java语句的执行顺序，但是在多线程中不行）



volatile的单例模式（懒汉式）

class Plant {  
 //volatile采用了禁止指令重排  
 private static volatile Plant *plant* = null;  
  
 private Plant() {  
 System.*out*.println("Plant的构造方法执行了");  
 }  
 public static Plant getInstance() {  
 //双端检锁  
 if (*plant* == null) {  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (Plant.class) {  
 //双端检锁  
 if (*plant* == null) {  
 *plant* = new Plant();  
 }  
 }  
 }  
 return *plant*;  
 }  
}  
public class Test01 {  
 public static void main(String[] args) {  
 for (int j = 0; j < 20; j++) {  
 new Thread(() -> {  
 Plant plant = Plant.*getInstance*();  
 }).start();  
 }  
 }  
}

1. 什么是CAS（乐观锁）

CAS的全称是Compare-And-Swap，它是一条CPU并发原语。

它的功能是判断内存中某个位置的值是否为预期值，如果是则更改为新值，这个过程是原子的。（原子操作类底层调用了unsafe类中的CAS方法，是基于硬件的功能，在执行的过程中不能被中断，也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题）

原理：A线程过来拿走主内存中的数据3，给加1，往回写的时候发现别的线程在他拿走后的时间段改变了它为4了，它就会在拿走4去加1，直到他加完回来往回写的时候，还是原来自己拿走时的样子，才会写回去，然后结束，要不然会一直在循环的跑。（底层有个do...while的方法）（这个unsafe.compareAndSwapInt()方法是native修饰的是汇编语言，操作的硬件，只能同时一个并发。）（自旋）

自定义对象的CAS操作

package com.test;  
  
import lombok.Getter;  
import lombok.Setter;  
import lombok.ToString;  
  
import java.util.concurrent.atomic.AtomicReference;  
  
@Getter  
@Setter  
@ToString  
class User {  
 private String name;  
 private Integer age;  
  
 public User(String name, Integer age) {  
 this.name = name;  
 this.age = age;  
 }  
}  
  
*/\*\*  
 \* 自定义对象的原子CAS操作  
 \*/*public class AtomicReferenceDemo {  
 public static void main(String[] args) {  
 User zhangsan = new User("zhangsan", 22);  
 User lisi = new User("lisi", 22);  
 AtomicReference<User> ar = new AtomicReference<>();  
 ar.set(zhangsan);  
 System.*out*.println(ar.compareAndSet(zhangsan, lisi) + "==" + ar.get());  
 System.*out*.println(ar.compareAndSet(zhangsan, lisi) + "==" + ar.get());  
 }  
}

CAS的缺点：

1. 循环时间长CPU开销大。
2. 只能保证一个共享变量的原子操作。
3. ABA问题（两个线程t1和t2，t1和t2拿到主物理内存中的数据A,然后去改变，t1线程的执行时间长于t2线程的两倍以上，在t1线程拿到A到改变为C的中间时刻，t2线程先将A改为B，然后在拿到B改为A，t1线程过来的时候，发现还是A，比较并修改，改为C。期间t1线程不知道t2线程操作了2次，还以为没人操作呢。这就是ABA问题）

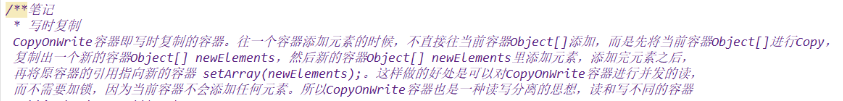
解决ABA问题：使用一个修改版本号的机制（类似于时间戳）

package com.test;  
  
import java.util.concurrent.atomic.AtomicStampedReference;  
  
public class AtomicStampedReferenceDemo {  
 */\*\*  
 \* 构造方法中的第一个参数是数值，第二个是版本号  
 \*/* static AtomicStampedReference<Integer> *atomicStampedReference* = new AtomicStampedReference<>(100, 1);  
  
 public static void main(String[] args) {  
 new Thread(() -> {  
 */\*\*  
 \* 先让t1拿到第一次的版本号  
 \*/* System.*out*.println(Thread.*currentThread*().getName() + "线程的第一次版本号" + *atomicStampedReference*.getStamp());  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *atomicStampedReference*.compareAndSet(100, 101, *atomicStampedReference*.getStamp(), *atomicStampedReference*.getStamp() + 1);  
 System.*out*.println(Thread.*currentThread*().getName() + "线程的第二次版本号" + *atomicStampedReference*.getStamp());  
 *atomicStampedReference*.compareAndSet(101, 100, *atomicStampedReference*.getStamp(), *atomicStampedReference*.getStamp() + 1);  
 */\*\*  
 \* 改变后，值为原来的100了，但是版本号已经为3了  
 \*/* System.*out*.println(Thread.*currentThread*().getName() + "线程的第三次版本号" + *atomicStampedReference*.getStamp());  
 }, "t1").start();  
  
 new Thread(() -> {  
 int stamp = *atomicStampedReference*.getStamp();  
 */\*\*  
 \* 拿到第一次的版本号，前面的t1线程等待一秒的作用就是这  
 \*/* System.*out*.println(Thread.*currentThread*().getName() + "线程的第一次版本号" + stamp);  
 */\*\*  
 \* 等待3秒钟，让t1线程跑完，把值改回来了为100，但是版本号为3了  
 \*/* try {  
 Thread.*sleep*(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 */\*\*  
 \* 在此处的比较并交换的时候，原来的值是100比较到了，但是当前的版本号和原来的不一样了，所以无法写入实际的数据2019  
 \*/  
 atomicStampedReference*.compareAndSet(100,2019,stamp,stamp+1);  
 */\*\*  
 \* 此处的值为100，版本号为3  
 \*/* System.*out*.println("值为="+*atomicStampedReference*.getReference()+"当前的版本号为="+*atomicStampedReference*.getStamp());  
 }, "t2").start();  
 }  
}

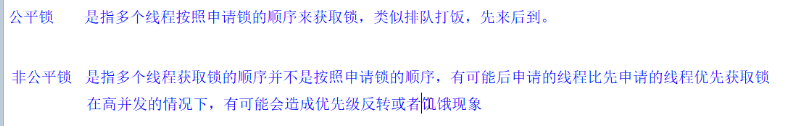
1. 集合修改的时候不安全

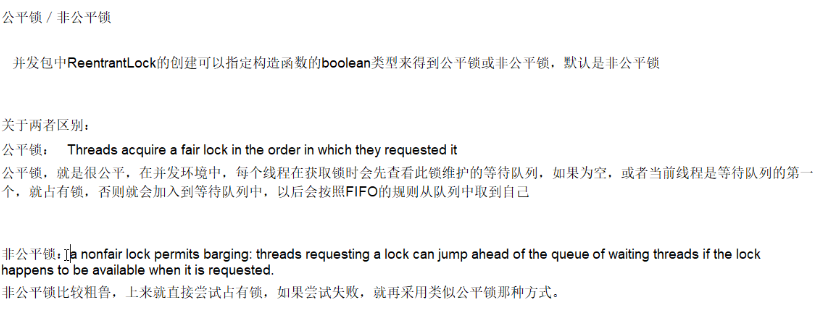
读写分离，写时复制，每个集合都有对应的CopyOnWriteArrayList和CopyOnWriteArraySet类，其中CopyOnWriteArraySet底层也是CopyOnWriteArrayList这个类。

对于Map集合有个 ConcurrentHashMap分段式锁机制，保证并发安全。



1. 公平锁和非公平锁

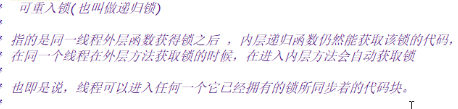




synchronized默认是非公平锁，ReentrantLock默认也是非公平锁，如果在构造中加true参数就为公平锁。new ReentrantLock(true);

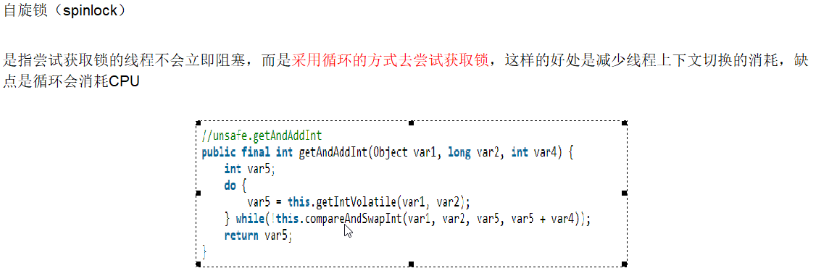
1. 可重入锁（递归锁）

在一个同步方法中在执行同步方法可以获得锁。（synchronized和ReentrantLock都是可重入锁）



package com.zr.test;  
  
  
public class Demo {  
 public static void main(String[] args) {  
 new Thread(new MyRunable(), "t1").start();  
 new Thread(new MyRunable(), "t2").start();  
 }  
}  
  
class MyRunable implements Runnable {  
 @Override  
 public void run() {  
 method1();  
 }  
  
 public synchronized void method1() {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t method1方法执行了！");  
 */\*\*  
 \* 在同步代码块中继续调用同步方法，可以运行。（证明synchronized为可重入锁）  
 \*/* method2();  
 }  
  
 public synchronized void method2() {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t method2方法执行了！");  
 }  
}

1. 自旋锁



package com.zr.test;  
  
  
import java.util.concurrent.atomic.AtomicReference;  
  
public class Demo {  
 //原子引用线程  
 AtomicReference<Thread> atomicReference = new AtomicReference<>();  
 public void myLock() {  
 Thread thread = Thread.*currentThread*();  
 System.*out*.println(thread.getName() + "\t myLock方法执行了");  
 */\*\*  
 \* t1线程先到此处，执行完下面的代码，把线程对象放入到AtomicReference  
 \* 中，然后t2线程到达时，只能在下面的代码中死循环，等到t1执行完myUnLock()  
 \* 方法将AtomicReference里面的线程换为null，然后t1结束死循环，将自己放入到  
 \* AtomicReference中，走完下面的调用的方法  
 \*/* while (!atomicReference.compareAndSet(null, thread)) {  
 }  
 }  
  
 public void myUnLock() {  
 Thread thread = Thread.*currentThread*();  
 System.*out*.println(thread.getName() + "\t myUnLock方法执行了");  
 atomicReference.compareAndSet(thread, null);  
 }  
  
 public static void main(String[] args) {  
 Demo demo = new Demo();  
 new Thread(() -> {  
 demo.myLock();  
 */\*\*  
 \* 线程在中间睡眠5秒，以保证t2线程到  
 \* 自旋锁处自旋  
 \*/* try {  
 Thread.*sleep*(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 demo.myUnLock();  
 }, "t1").start();  
 new Thread(() -> {  
 */\*\*  
 \* 线程先睡眠1秒，以保证t1线程先走  
 \*/* try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 demo.myLock();  
 demo.myUnLock();  
 }, "t2").start();  
 }  
}

1. CountDownLatch代码(减法)

package com.zr.test;  
  
import java.util.concurrent.CountDownLatch;  
  
*/\*\*  
 \* 目的是让一个或者多个线程等待，直到其他线程的一系列操作完成。  
 \* 保证其他线程完全结束操作，主（main）线程才执行。  
 \*/*public class Demo {  
 public static void main(String[] args) throws Exception {  
 */\*\*  
 \* CountDownLatch是具有synchronized机制的一个工具，  
 \* 计数预设值为6个。  
 \*/* CountDownLatch countDownLatch = new CountDownLatch(6);  
 for (int i = 0; i < 6; i++) {  
 new Thread(() -> {  
 System.*out*.println(Thread.*currentThread*().getName() + "线程 执行完了");  
 //当一个线程执行完了上述的代码，就把计数值减1  
 countDownLatch.countDown();  
 }, String.*valueOf*(i)).start();  
 }  
 //main线程在此等待，直到计数值为0的时候才能执行以下的代码  
 countDownLatch.await();  
 System.*out*.println(Thread.*currentThread*().getName() + "#####线程 执行完了");  
 }  
}

11.CyclicBarrier代码（加法）

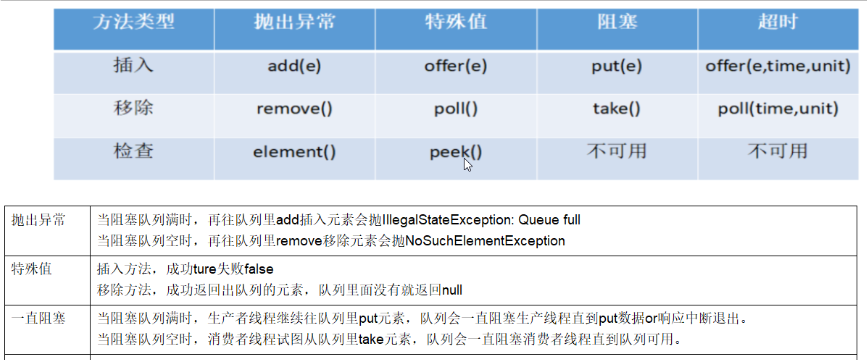
package com.zr.test;  
  
  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CyclicBarrier;  
  
public class Demo {  
 public static void main(String[] args) throws Exception {  
 //预设置一个数字和任务  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(7, () -> {  
 System.*out*.println("召唤神龙！！！");  
 });  
 for (int i = 1; i <=7; i++) {  
 final int j = i;  
 new Thread(() -> {  
 System.*out*.println("集齐了" + j + "龙珠");  
 try {  
 //其他线程走到这里加1，等到加到预设值后，执行预设的方法  
 cyclicBarrier.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }, String.*valueOf*(i)).start();  
 }  
 }  
}

1. Semaphore作用：解决多个线程操作多个资源的问题

package com.zr.test;  
  
import java.util.Random;  
import java.util.concurrent.Semaphore;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 作用：解决多个线程操作多个资源的问题  
 \* 可以预设一个数字值3，当来的请求为6个时，先进入3个，当其中有一个离开时  
 \* 在进去一个，最后直到全部完成。（类似于景点停车）  
 \*/*public class Demo {  
 public static void main(String[] args) throws Exception {  
 Semaphore semaphore = new Semaphore(3);//模拟三个车位  
 //有6个车需要车位  
 for (int i = 1; i <= 6; i++) {  
 new Thread(() -> {  
 try {  
 semaphore.acquire();//有一个车占了一个车位  
 System.*out*.println(Thread.*currentThread*().getName() + "\t 抢到了车位");  
 int seconds = new Random().nextInt(5) + 1;  
 TimeUnit.*SECONDS*.sleep(seconds);//这个车待的一个随机的时间(1到6秒内的一个时间段)  
 System.*out*.println(Thread.*currentThread*().getName() + "\t 离开了车位，用来" + seconds + "秒");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 semaphore.release();//这个车走了离开了车位  
 }  
 }, String.*valueOf*(i)).start();  
 }  
 }  
}

1. 阻塞队列



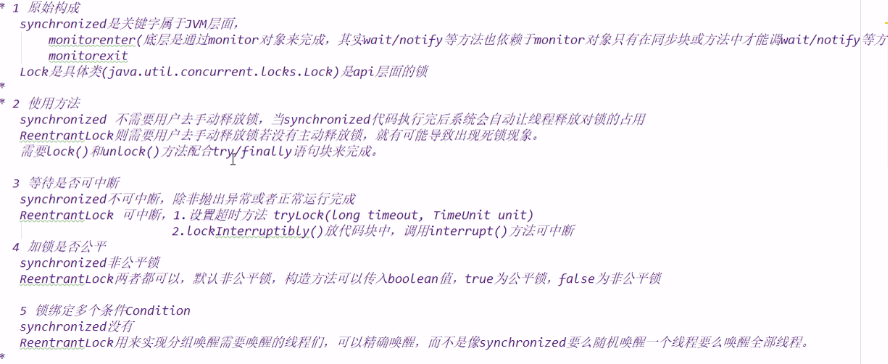


ArrayBlockingQueue对应ArrayList， BlockingQueue和List都是Collection的子接口。（不平级List和Queue平级）

同步队列SynchronousQueue

synchronized和Lock的区别

1. synchronized关键字是JVM层面的，Lock是一个类中api层面的。
2. synchronized不需要主动释放锁（代码执行完后会自动释放锁），Lock需要用户主动释放锁，不释放会导致死锁。
3. synchronized不可中断，除非抛出异常和运行完成。Lock可中断tryLock设置超时方法。
4. synchronized默认是非公平锁不可更改。Lock默认也是非公平锁，但是在构造方法通过传布尔值true可以修改为公平锁。
5. synchronized不能精确唤醒，要么唤醒一个，要么全部唤醒。Lock用Condition的定点唤醒可以精确唤醒。



开启线程的第三个方法实现Callable接口

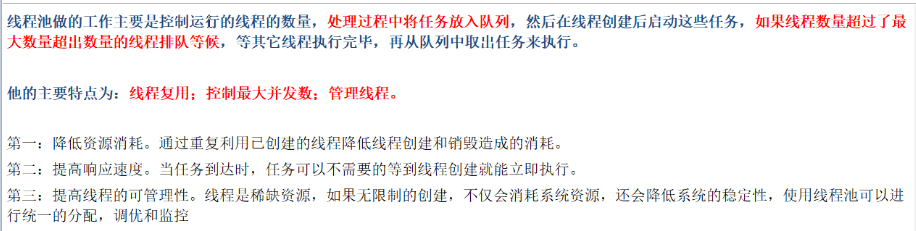
实现Callable<>接口，是带泛型的，泛型就是call方法的返回值类型，要new一个FutureTask在构造中放入实现这个接口的类，然后再将这个FutureTask放入到Thread的构造中。

用途：在一个线程中可以中途开一个分支线程去计算一个耗时慢的任务，最后汇总。

有返回值就可以清晰的看到新开的线程那个有异常。（便于监控）

package com.zr.test;  
  
  
import java.util.concurrent.Callable;  
import java.util.concurrent.FutureTask;  
  
class MyCallable implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
 System.*out*.println(Thread.*currentThread*().getName() + "MyCallable执行了！");  
 Thread.*sleep*(2000);//等待2秒，模拟新开的线程没有算完  
 return 2;  
 }  
}  
  
*/\*\*  
 \* Callable线程的使用  
 \*/*public class Demo {  
 public static void main(String[] args) throws Exception {  
 FutureTask<Integer> integerFutureTask = new FutureTask<>(new MyCallable());  
 new Thread(integerFutureTask, "AAA").start();  
 */\*\*  
 \* 如果在开一个线程都用的是一个FutureTask那么call()方法只会执行一次  
 \* 例：new Thread(integerFutureTask, "BBB").start();  
 \*/  
 /\*\*  
 \* 有可能上面的线程还没有计算完返回i，所以有可能导致mian线程阻塞，  
 \*/* Integer i = integerFutureTask.get();  
 */\*\*  
 \* 也可以在此处写一个自旋锁防止main线程阻塞，main线程会在此处死循环，  
 \* 但是没有意义，阻塞和死循环没区别  
 \*/* /\*while (!integerFutureTask.isDone()) {  
 }\*/  
 Integer j = 1;  
 System.*out*.println(i + j);  
  
 }  
}

线程池

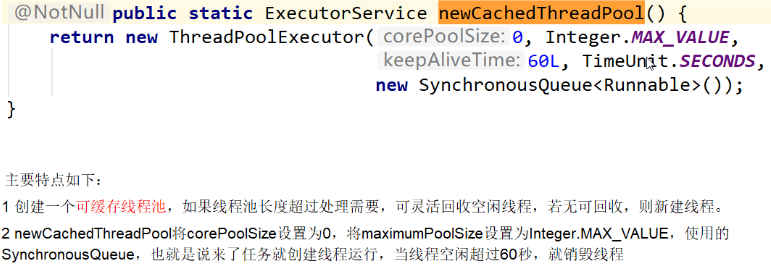


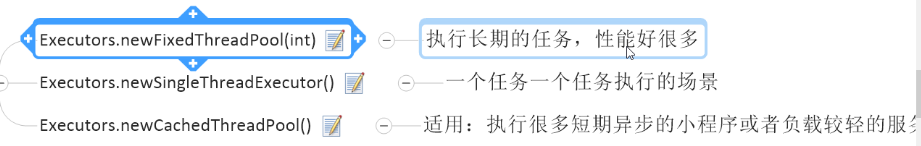
1.

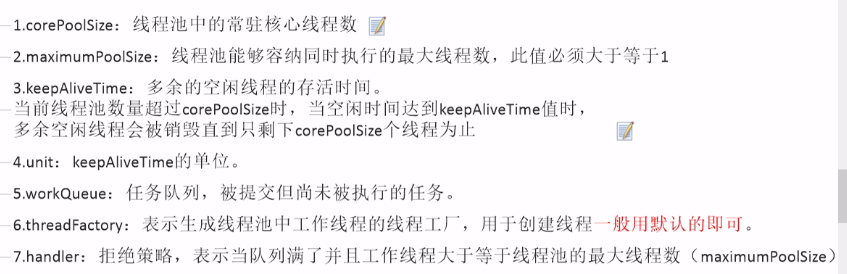
2.



3.



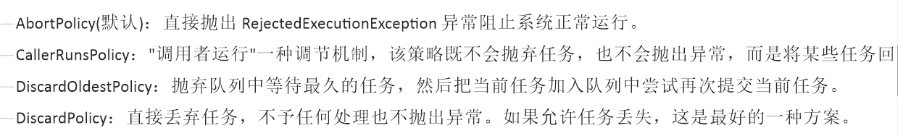
  
线程池的7大参数



线程池的工作流程



线程池的拒绝策略



自定义线程池的代码

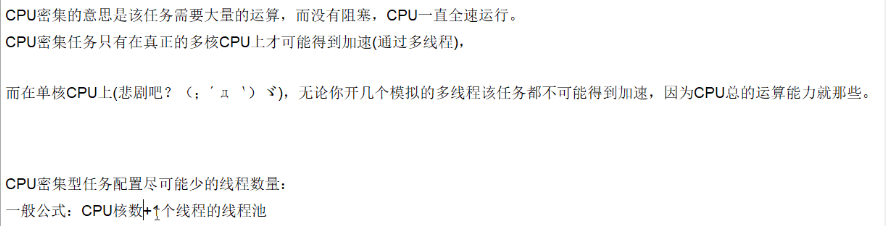
package com.zr.test;  
  
import java.util.concurrent.Executors;  
import java.util.concurrent.LinkedBlockingQueue;  
import java.util.concurrent.ThreadPoolExecutor;  
import java.util.concurrent.TimeUnit;  
  
public class Demo {  
 public static void main(String[] args) throws Exception {  
 */\*\*  
 \* 自定义线程池  
 \*/* ThreadPoolExecutor pool = new ThreadPoolExecutor(  
 2,//核心线程数  
 5,//最大线程数  
 1L,//空闲线程最大等待时间  
 TimeUnit.*MILLISECONDS*,///空闲线程最大等待时间单位  
 new LinkedBlockingQueue<Runnable>(3),//阻塞队列，参数为阻塞队列的容量  
 Executors.*defaultThreadFactory*(),//默认的线程创建工厂  
 */\*\*  
 \* 拒绝策略 AbortPolicy:抛出异常  
 \* CallerRunsPolicy:将任务返回给调用者（main线程）  
 \* DiscardOldestPolicy:抛弃等待最久的线程  
 \* DiscardPolicy:直接丢弃任务  
 \*/* new ThreadPoolExecutor.DiscardPolicy());  
 try {  
 for (int i = 1; i <= 10; i++) {  
 pool.execute(() -> {  
 System.*out*.println(Thread.*currentThread*().getName() + "线程执行了");  
 });  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 pool.shutdown();  
 }  
 }  
}

定义线程池的线程数（分两种情况）

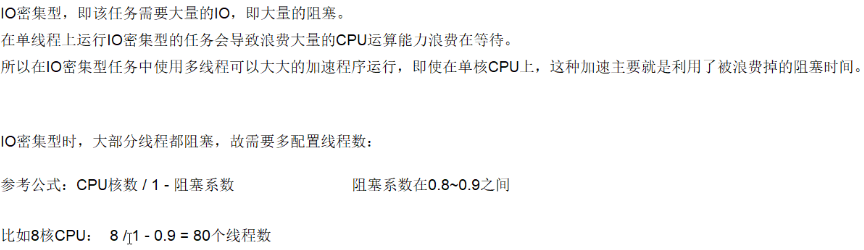
CPU密集型（减少切换）

CPU的核心数查看

Runtime.*getRuntime*().availableProcessors()



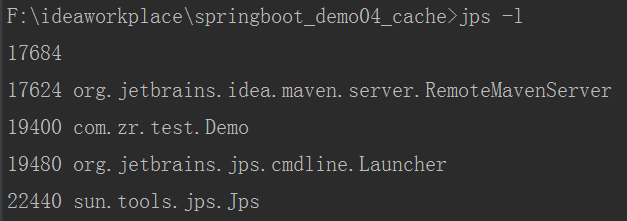
IO密集型



死锁代码

死锁的故障排查

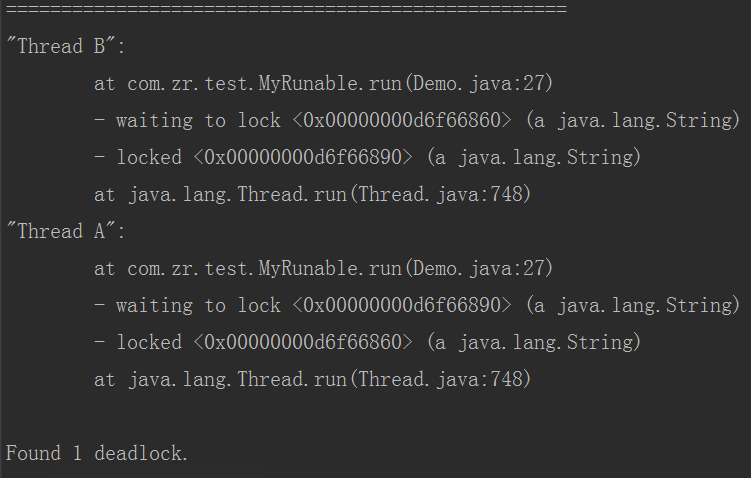
在idea的dos窗口输入jps -l 查出你运行的类的进程编号。



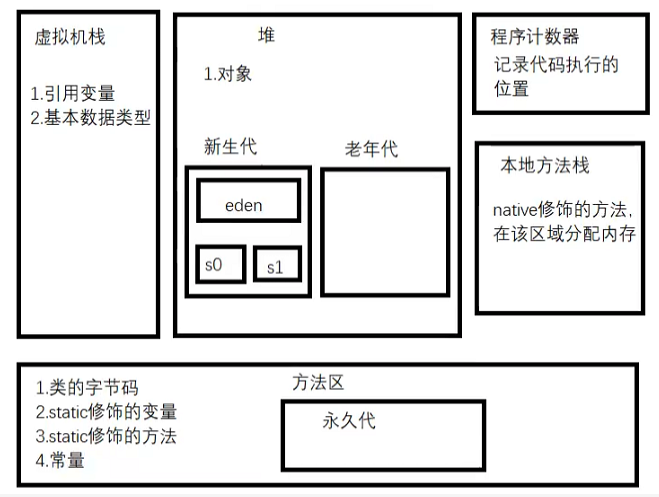
然后在用jstack +刚才查出的进程号，查询故障结果。



在查询故障结果的最下边又具体的故障信息。



GC垃圾回收机制



GC工作的地方（线程公用的内存）

方法区和堆内存

1. GC的垃圾收集算法。

引用计数法：一般不采用，无法解决循环引用的问题。

可达性分析算法：JVM采用，涉及GCRoots。

GCRoots：

虚拟机栈中的对象，静态变量对象，常量，native方法中引用的对象。

复制算法：新生代S0和S1区，缺点：浪费空间

标记清除：先标记需要回收的对象，再清除这些对象。缺点：产生了内存碎片

标记整理：老生代先标记，后压缩。（默认新生代存活15次进入老生代）

垃圾回收算法：

JVM的参数类型

标配参数：-version -help

X参数：-Xint -Xcomp -Xmixed

XX参数-布尔类型: -XX:+PrintGCDetails（开启GC打印细节）

jsp -l ：查看进程

jinfo -flag+配置项+进程编号：查看某个属性的状态

jinfo -flags+进程编号：查所有JVM的参数

例

jinfo -flag PrintGCDetails 20508 查看打印GC收集细节

jinfo -flag MetaspaceSize 14108 查看原空间大小

jinfo -flag MaxTenuringThreshold 24240 查看新生代多少次变为老生代

XX参数-K-V类型:

-XX:MetaspaceSize=1024m（设置原空间大小） -XX:MaxTenuringThreshold=15 （设置新生代15次变为老生代）



查看JVM全局的参数配置：

初始化：java -XX:+PrintFlagsInitial

修改后：java -XX:+PrintFlagsFinal

= 没有修改过的值 ：= 人为修改过的值

在运行java文件的时候打印出来JVM的信息（在Main.class文件夹中的cmd）

看默认的垃圾回收信息：java -XX:+PrintCommandLineFlags -version

-XX:+UseParallelGC

例：java -XX:+PrintFlagsFinal -XX:MetaspaceSize=512m Mian

程序获取堆内存的大小：

long totalMemory = Runtime.*getRuntime*().totalMemory();//初始化堆内存  
 long maxMemory = Runtime.*getRuntime*().maxMemory();//最大堆内存  
 System.*out*.println(totalMemory / 1024 / 1024 \* 64);//默认物理内存的1/64  
 System.*out*.println(maxMemory / 1024 / 1024 \* 4);//默认物理内存的1/4  
//两者输出为7000左右，单位为M，本机的电脑是8G，因为有部分的计算问题

新生代内存占堆内存的1/3,老生代占堆内存的2/3

堆内存初始值：-Xms 等价于 -XX:InitialHeapSize

堆内存最大值：-Xmx 等价于 -XX:MaxHeapSize

栈内存：-Xss 等价于 -XX:ThreadStackSize （一般大小为512k-1024k）

新生代：-Xmn （一般不调）

元空间：-XX:MetaspaceSize

打印GC回收细节：-XX:+PrintGCDetails

垃圾回收器：1.-XX:+UseSerialGC 串行

2..-XX:+UseParalleGC 并行（默认）

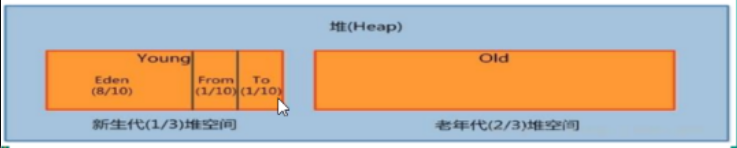
eden区的占比: -XX:SurvivorRatio 新生代Eden与两个Survivor的比例

老年代的占比：-XX:NewRatio

设置进入养老区的次数：-XX:MaxTenuringThreshold 默认是15次。如果改成0，那么就不在新生代的S区分配，直接进入老年代。

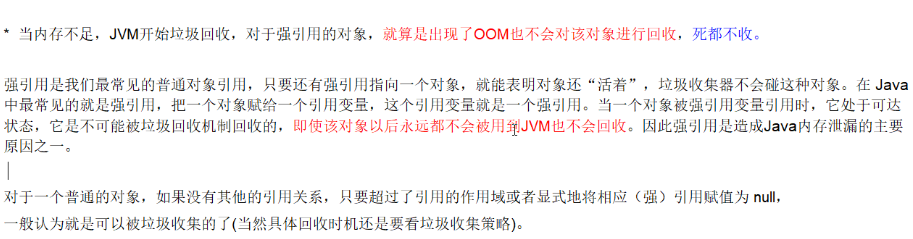
例：

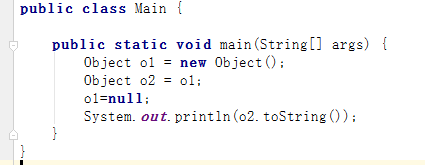
-Xms128m -Xmx2048m -XX:ThreadStackSize=1024k -XX:+PrintGCDetails -XX:+UseSerialGC -XX:SurvivorRatio=8 -XX:NewRatio=2 -XX:MaxTenuringThreshold=15



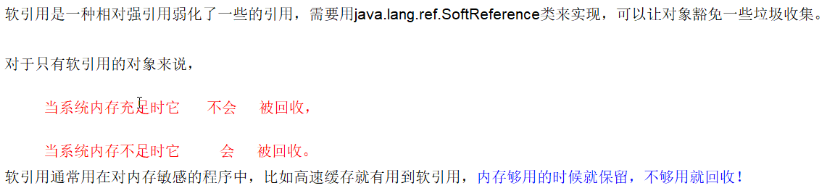


强引用





软引用



1. sql语句中的union 和union all的区别

这两个关键字放在两条查询语句中间会把查询结果制成一张表，单前提是两个查询结果的字段数要一致才可以，union会把查询的结果集去重，而union all不会去重。

例：查同表去重：SELECT \* FROM dept UNION SELECT \* FROM dept;

查同表不去重：SELECT \* FROM dept UNION All SELECT \* FROM dept;

查不同表：SELECT \* FROM dept UNION SELECT \* FROM salgrade;

1. Mybatis中的#{}和${}的区别

#{}是防注入的占位符，它会将输入的数据自动加单引号，有预编译功能。（主要针对一些条件参数）

例：select \* from user where name = #{name};

${}是没有防注入的占位符，它将输入的数据直接拼装在sql语句中。（主要用于sql中关键字的使用，比如order by动态参数）

1. 事务的特征和隔离级别

事务的四大特性：（ACID）

A：原子性：事务是最小的工作单元，不可再分。

C: 一致性：事务必须保证多条DML语句同时成功或者同时失败。

I：隔离性：事务A和事务B之间具有隔离。

D: 持久性：最终的数据必须持久化到硬盘文件中，事务才算成功的结束。

事务之间的隔离性

\*有4个级别

1. 读未提交（read uncommitted）（A事务不用结束比如改变数据后不commit 。B个事务就可以看到A事务改变后的数据）

2.读已提交（read committed）（A事务改变数据commit后，B事务才可以看到A事务改变后的数据）

3.可重复读（repeatable read）（A事务改变数据commit后，B事务进来看到的数据还是之前它进来时的数据（幻想数据），看不到A事务改变后的数据）

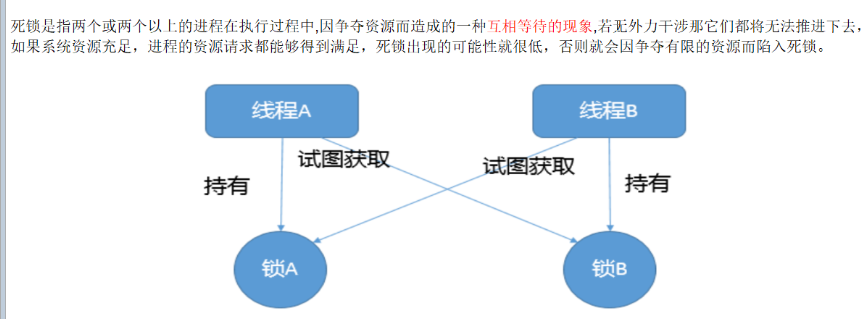
4.序列化读（serializable）（A事务没有commit，B事务根本查不了关闭一直在这里闪，等A事务提交后，B事务立马就可以查出来）

Oracle 数据库默认的隔离级别是：读已提交。

MySQL数据库默认的隔离级别是：可重复读。

Mysql事务默认是自动提交的。（执行一条DML语句就自动提交事务）（关闭自动提交start transaction）

1. 死锁



package com.atguigu.springboot;  
  
*/\*\*  
 \* 手写死锁  
 \* 假设t1先拿到o1锁,等待一秒后，去拿o2锁（这一秒钟内t2线程早  
 \* 已在下面的代码中已经拿到了o2锁，也在等待一秒后拿o1锁）两个  
 \* 线程都必须拿到对方持有的锁才可以输出后归还锁，两个线程一直  
 \* 在此等待，所以就导致了死锁的产生。  
 \*/*public class Test01 {  
 public static void main(String[] args) {  
 Object o1 = new Object();  
 Object o2 = new Object();  
 */\*\*  
 \* 两个线程公用两个Object对象  
 \*/* Thread t1 = new Thread(new MyRunnable1(o1, o2));  
 Thread t2 = new Thread(new MyRunnable2(o1, o2));  
 t1.start();  
 t2.start();  
 }  
}  
class MyRunnable1 implements Runnable {  
 Object o1;  
 Object o2;  
  
 MyRunnable1(Object o1, Object o2) {  
 this.o1 = o1;  
 this.o2 = o2;  
 }  
  
 @Override  
 public void run() {  
  
 synchronized (o1) {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (o2) {  
 System.*out*.println("MyRunnable1执行了...");  
 }  
 }  
 }  
}  
  
class MyRunnable2 implements Runnable {  
 Object o1;  
 Object o2;  
  
 MyRunnable2(Object o1, Object o2) {  
 this.o1 = o1;  
 this.o2 = o2;  
 }  
  
 @Override  
 public void run() {  
 synchronized (o2) {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 synchronized (o1) {  
 System.*out*.println("MyRunnable2执行了...");  
 }  
 }  
 }  
}

10.跨域

协议不同算跨域，例如：http和https。  
端口号不同算跨域，例如：80和8080。  
域名不同算跨域，例如：baidu.com和jingdong.com  
以上出现一种就算是跨域。

11sql优化（原则：防止全表扫描）（索引底层采用的数据结构是：B + Tree）

1. 在where后面少用判断null或者not null；
2. 查询的时候不要select \*，把要查的字段写出来。
3. 在where和order by 经常查的字段上添加索引。
4. 尽可能减少临时表的使用。
5. 模糊查询的时候尽量不要把%放在数据的最前面。（会导致索引失效）
6. 索引不能太多，太多了会导致insert和update的速度慢。
7. 查看sql语句的执行时间（注意:直接show profiles也可以，下面的操作不用）

1 show profiles;   
2 show variables;查看profiling 是否是on状态；   
3 如果是off，则 set profiling = 1；   
4 执行自己的sql语句；   
5 show profiles；就可以查到sql语句的执行时间；

1. ArrayList遍历

用普通的for循环：

输出的错误，因为这是按照下标去遍历删除的，在下标为1的位置有需要删除，当删除完这个的时候，集合的size变化，导致每个元素的下标也变化，原来下标为2位置的元素顺位到了下标为1，此时遍历的i已经到了2，导致这个元素没有遍历判断过，亦未删除。

List<Integer> list = new ArrayList();  
list.add(1);  
list.add(0);  
list.add(0);  
list.add(12);  
list.add(0);  
list.add(0);  
for (int i = 0; i < list.size(); i++) {  
 if (list.get(i) == 0) {  
 list.remove(i);  
 }  
}  
//输出的结果[1, 0, 12, 0]  
System.*out*.println(list);

普通的for循坏可以倒着删除（不会产生问题）

List<String> list = new ArrayList<>();  
list.add("a");  
list.add("a");  
list.add("d");  
list.add("a");  
list.add("a");  
list.add("f");  
list.add("c");  
list.add("q");  
  
for (int i = list.size() - 1; i >= 0; i--) {  
 if (list.get(i).equals("a")) {  
 list.remove(i);  
 }  
}  
System.*out*.println(list);

用迭代器删除：

可以正确的删除元素，因为每个Iterator的hasNext()方法会判断边界是否可以被迭代，next()

方法会拿出指针指向的对象，所以调用remove()方法删除元素（注意：用remove()方法之前必须调用next()方法）

List<Integer> list = new ArrayList();  
 list.add(1);  
 list.add(0);  
 list.add(0);  
 list.add(12);  
 list.add(0);  
 list.add(0);  
 Iterator<Integer> it = list.iterator();  
 while (it.hasNext()) {  
 if (it.next() == 0) {  
 it.remove();  
 }  
 }  
 //输出的结果：[1, 12]  
 System.*out*.println(list);

ArrayList:

                   底层数据结构是数组，查询快，增删慢线程不安全，效率高

Vector:

                   底层数据结构是数组，查询快，增删慢线程安全，效率低（方法都有synchronized关键字修饰）

LinkedList:

                   底层数据结构是（双向）链表，查询慢，增删快线程不安全，效率高

LinkedList是一个由相互引用的节点组成的双向链表，那么当把数据插入至该链表某个位置时，该数据就会被组装成一个新的节点，随后只需改变链表中对应的两个节点之间的引用关系，使它们指向新节点，即可完成插入。同样的道理，删除数据时，只需删除对应节点的引用即可。

ArrayList是一个可变长数组，插入数据时，则需要先将原始数组中的数据复制到一个新的数组，随后再将数据赋值到新数组的指定位置（如下图）；删除数据时，也是将原始数组中要保留的数据复制到一个新的数组。

Mybatis的常用注解：

### ****@Insert @Update @Delete @Select @Result @Results @ResultMap @One @Many****

Mybatis的常用标签：

<resultMap id="BaseResultMap" type="com.online.charge.platform.student.model.Student">

<id property="id" column="id" />

<result column="NAME" property="name" />

<result column="HOBBY" property="hobby" />

<result column="MAJOR" property="major" />

<result column="BIRTHDAY" property="birthday" />

<result column="AGE" property="age" />

</resultMap>

<if test="name != null and name != ''">

and NAME = #{name}

</if>

<select id="getStudentListWhere" parameterType="Object" resultMap="BaseResultMap">

SELECT \* from STUDENT

<where>

<if test="name!=null and name!='' ">

NAME LIKE CONCAT(CONCAT('%', #{name}),'%')

</if>

<if test="hobby!= null and hobby!= '' ">

AND hobby = #{hobby}

</if>

</where>

</select>

<include refid="Base\_Column\_List" />

Spring的常用注解：

****@Component @Service  @Repository  @Autowired****

****@Resource @Configuration @Bean  @Value****

springmvc的常用注解

****@Controller @RequestMapping  @ResponseBody****

****@PathVariable（路径参数） @RestController****

springboot的常用注解

****@SpringBootApplication =@Configuration +@EnableAutoConfiguration+@ComponentScan****

****@MapperScan @GetMapping @PostMapping @ConfigurationProperties @ConditionalOnClass****

Linux的常用命令

ll 查看当前文件夹下的目录。

mkdir +文件名 创建文件夹

cd + 文件夹名 进入文件夹

cd .. 退回上层目录

cd / 返回根目录

sudo touch 123.txt 创建123.txt文件

sudo vi 123.txt 编辑123.txt文件

按esc 后输入:wq 保存文件

rm -f 123.txt 删除123.txt

rm -rf one 删除名为one的文件夹（也可以删文件）

find / -name 文件（或文件夹） 搜索文件或文件夹

uname -r  查看内核版本

yum install docke 安装docker

systemctl start docker 运行docker

systemctl stop docker 停止 docker

ps -ef|grep xxxx 查看进程

ls -l

面向对象：继承，封装，多态

UDP和TCP

是 TCP/IP 协议栈中传输层的两个协议。

1.TCP 是面向连接的传输。UDP 是无连接的传输

2.UDP 因为少了很多控制信息，所以传输速度比 TCP 速度快

3.TCP 适合用于传输大量数据，UDP 适合用于传输小量数据

HTTPS和HTTP的区别主要如下：（应用层）

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

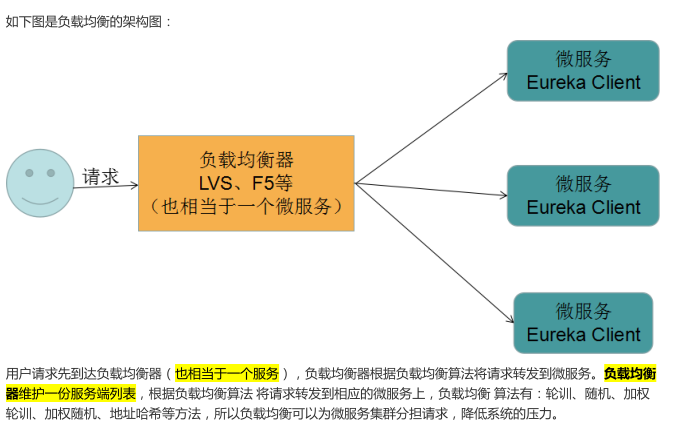
3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

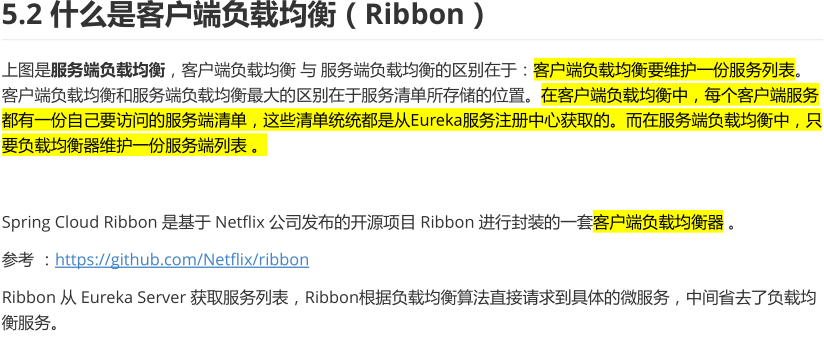
4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

SpringCloud的五大组件

****Eureka（注册中心）****:微服务启动时,Eureka会将服务注册到EurekaService,并且EurakeClient还可以返回过来从EurekaService拉去注册表,从而知道服务在哪里

****Ribbon（负载均衡）****:服务间发起请求的时候,基于Ribbon服务做到负载均衡,从一个服务的对台机器中选择一台（默认轮训算法）（服务端的负载均衡）





****Feign（客户端服务调用）****:

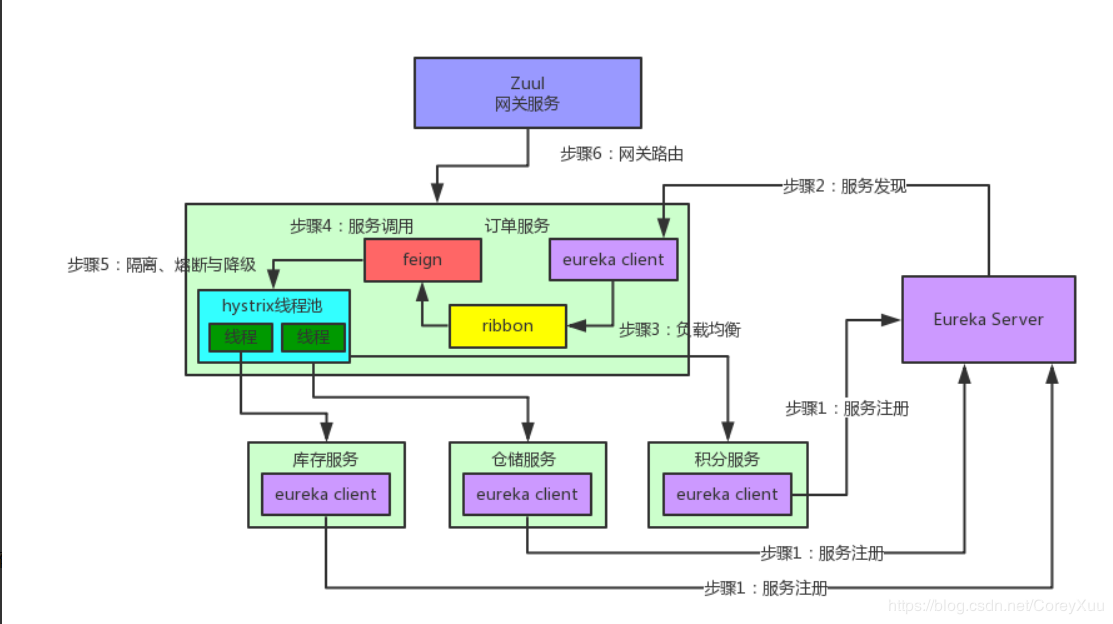


****Hystrix（服务熔断）:****发起的请求是通过Hystrix的线程池来走,不同的服走不同的线程池,实现了不同的服务调度隔离,避免服务雪崩的问题 （阈值（缺省是5秒内20次调用失败)）

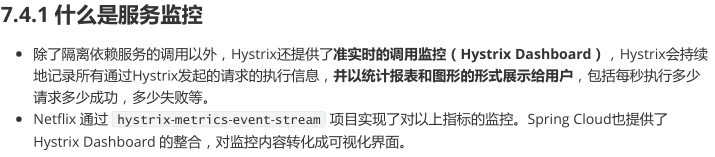
分为：服务端的熔断和客户端熔断，服务端熔断会在发生错误的时候返回一个设置好的定值，不让线程在其等待（防止雪崩问题的出现），客户端的熔断主要用于服务端的宕机，无法起到熔断效果，两者共同使用效果好。

****Zuul（路由网关）****:如果前端后端移动端调用后台系统,统一走zull网关进入,有zull网关转发请求给对应的服务

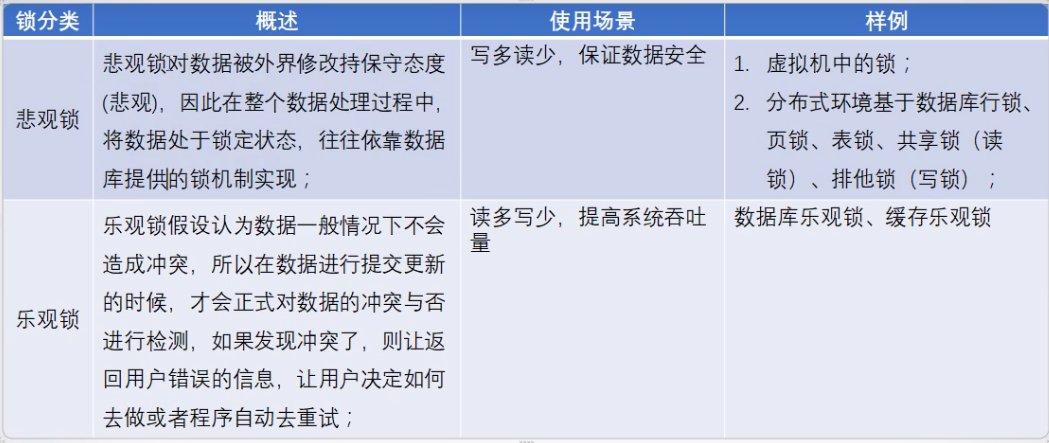
路由网关可以处理一些权限问题，里面可以配置过滤器



****Hystrix Dashboard 服务监控****



乐观锁和悲观锁





Map集合的遍历的方法

package com.zr.test;  
  
import java.util.\*;  
  
  
*/\*\*\*  
 \* Map集合的三种遍历方法  
 \*/*public class Demo {  
 public static void main(String[] args){  
 Map<Integer, String> map = new HashMap<>();  
 map.put(1, "a");  
 map.put(2, "b");  
 map.put(4, "d");  
 map.put(9, "c");  
 map.put(6, "g");  
 //第一种：拿到键的集合，然后遍历键的时候对应取出值  
 Set<Integer> set = map.keySet();  
 for (Integer key : set) {  
 System.*out*.println("key: " + key + "\t value: " + map.get(key));  
 }  
 //第二种 ：遍历值  
 Collection<String> values = map.values();  
 for (String value : values) {  
 System.*out*.println("\t value: " + value);  
 }  
 //第三种：迭代器  
 Iterator it = map.entrySet().iterator();  
 // 键和值  
 Integer key = null;  
 String value = null;  
 while (it.hasNext()) {  
 Map.Entry entry = (Map.Entry) it.next();  
 key = (Integer) entry.getKey();  
 value = (String) entry.getValue();  
 System.*out*.println("key:" + key + "---" + "value:" + value);  
  
 }  
 }  
}

MyBatis的一级缓存和二级缓存

一级缓存：是Mybatis默认开启的缓存机制，在同一个SqlSession下是一样的，当sql语句和参数没有改变的情况下，会先去缓存查找，当执行完update insert delete conmmit 或者sqlsession的close的方法等就会自动刷新一下缓存。

二级缓存：是基于SqlSessionFactory下的缓存，需要在SqlMapperConfig.xml中配置，IUserDao中配置，它是将查询的结果放置在cache中，但是放的结果不是对象只是数据，所以二次的查询结果对象不同，但是只走了一次数据库。