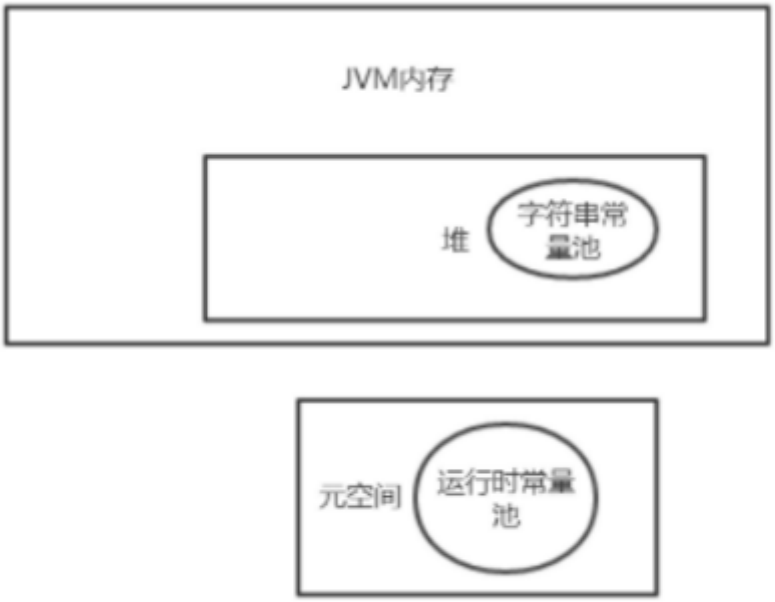
1、字符串String的内存结构

JDK 1.7：字符串常量放在了字符串常量池中，如果是new String则存在了堆中。

JDK 1.8：



方法区是一种规范层面的，元空间和永久带是方法区的实现。

2、基本数据类型与包装对象的区别

1、包装类是对象，拥有方法和字段，对象的调用都是通过引用对象的地址；基本类型不是
2、包装类型是引用的传递；基本类型是值的传递
3、声明方式不同：
    基本数据类型不需要new关键字；
    包装类型需要new在堆内存中进行new来分配内存空间
4、存储位置不同：
    基本数据类型直接将值保存在值栈中；
    包装类型是把对象放在堆中，然后通过对象的引用来调用他们
5、初始值不同：
    int的初始值为 0 、 boolean的初始值为false
    包装类型的初始值为null
6、使用方式不同：
    基本数据类型直接赋值使用就好；
    包装类型是在集合如 coolectionMap时使用

3、说一种你熟悉的设计模式

单例模式：饿汉式、懒汉式、内部静态类、枚举

4、LinkedHashMap,TreeHashMap,HashMap区别

1、有序性

HashMap　　无序

LinkedHashMap　有序(存值的顺序)

TreeMap　　有序(键值的顺序)

2、实现

HashMap　　是基于散列表，基于哈希表实现。O(1)。

LinkedHashMap　HashMap和双向链表合二为一。O(1)。

TreeMap　　基于红黑树实现的。O(log n)。

HashMap与TreeMap 比较，HashMap查询效率高但是无序，TreeMap有序但是查询效率低。LinkedHashMap保证有序和查询效率。

3、key是否可以为null

TreeMap 不可以

5、List中如果让你实现删除一个元素，具体代码（思路）

1、如果直接用remove在for循环中遍历删除的话，会出现问题，只能删除一个，因为list是连续的存储结构。每删除一个元素，后边的元素都会左移一位，也就是下标会减1，然后下一次循环时就会遗漏这个元素了。（**可以通过从后往前遍历删除**）

```
for (int i = list.size() -1 ; i >= 0; i--){
    if(list.get(i).equals("a")){
        list.remove(i);
    }
}
```

2、通过迭代器替代for循环来删除。

```
Iterator iterator = list.iterator();
while (iterator.hasNext()){
    if(iterator.next().equals("a")){
        iterator.remove();
    }
}
```

6、你知道的实现线程安全的几种方式

synchronized、volatile、Lock、原子类（Atomic）

7、数据库怎么去重

```
#查询时去重
SELECT DISTINCT s_id FROM student;

SELECT s_id FROM student GROUP BY (no);

#添加唯一索引来完成去重
ALTER TABLE `user` ADD unique(`username`);

#插入时去重
insert ignore into table_name(email,phone,user_id)
values('test9@163.com','99999','9999'),
```

8、数据库的ACID四大特性。

1. 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志（Undo Log）来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

2. 一致性（Consistency）

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对同一个数据的读取结果都是相同的。

3. 隔离性（Isolation）

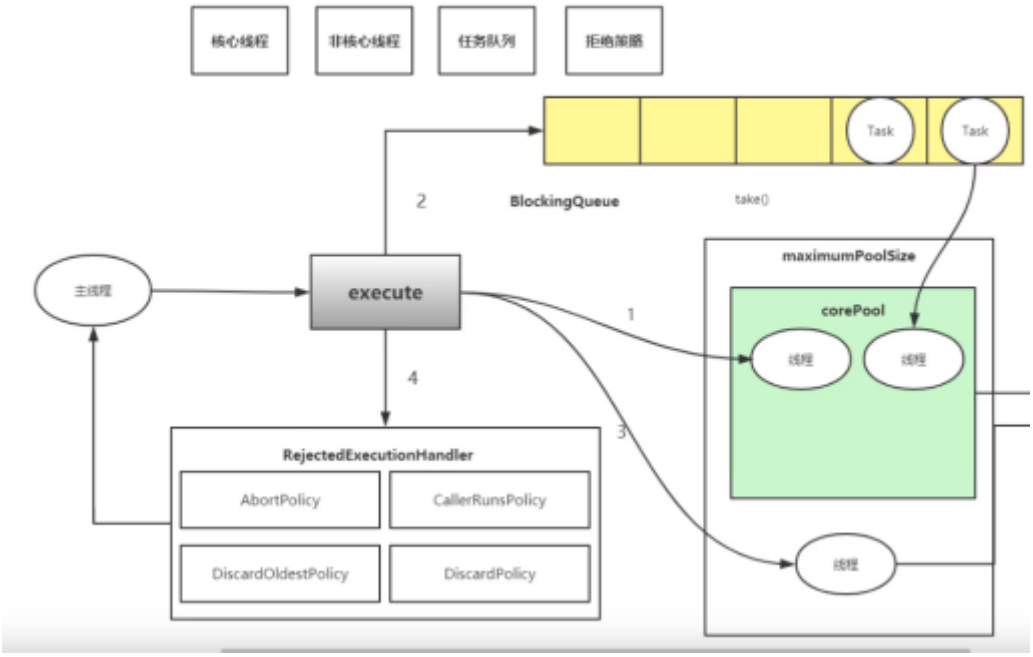
一个事务所做的修改在最终提交以前，对其它事务是不可见的。

4. 持久性（Durability）

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

9、线程池工作原理，常用的线程池实现类有哪些

流程是：一般阻塞队列中的任务会先放到核心线程中进行处理，如果核心线程都在工作且任务队列满了，就放到非核心线程中处理；否则就放到任务队列中，如果任务队列也满了就会创建非核心线程来处理。如果非核心线程也都在工作了，那么还有任务来请求处理就会被拒绝或者等待。



①. newFixedThreadPool(int nThreads)

创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

②. newCachedThreadPool()

可缓存的线程池，线程池无限大。如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

③. newSingleThreadExecutor()

这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

④. newScheduledThreadPool(int corePoolSize)

创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

10、ConcurrentHashMap原理

将数组分为多个段，每个段对应一个小的hashtable来加锁。

- java 1.7中：第一次先找到段，第二次到达段中的hashtable来遍历。put操作会对段加锁，若其他线程已获取锁会不断自旋（一直查询）。size操作通过每个段维护一个count来统计个数，首先不加锁统计所有count，若连续统计2次一致则正确，否则一致计算直到连续3次都不对则加锁统计。
- java1.8中：已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全。数据结构采用：数组+链表+红黑树。

11、流处理框架源码

12、数组与链表的区别，适用场景

最明显的区别是 ArrrayList底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。

数组适用于数据少、按序号快速访问元素的场景；

链表适用于数据多、频繁插入删除的场景；

13、Redis为什么那么快

- 数据方便扩展，因为是非关系型数据库，数据之间没有没关系。
- 存在内存中，读写速度快，一秒钟读11万次，写8万次。
- 数据类型多样，不需要提前设计复杂的数据表。

14、快排思路讲一下

```
int[] nums;

public void QuickSort(int left,int right){
        System.out.println("Quick Sort!");
        if(left>=right) return;
        int i = left,j = right;
        int temp = nums[left];
        while(i<j){
            while(i<j && nums[j]>=temp) j--;
            nums[i] = nums[j];
            while(i<j && nums[i]<=temp) i++;
            nums[j] = nums[i];
        }
        nums[i] = temp;
        QuickSort(left,i-1);
        QuickSort(i+1,right);
    }
```

15、线程间进行通信的方式

- synchronized+wait()、notify()

```
class Data{ // 数字 资源类
    private int number = 0;

    //+1
    public synchronized void increment() throws InterruptedException {
         while(number!=0){ //0
            // 等待
            this.wait();
        }
        number++;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我+1完毕了
        this.notifyAll();
```

```
    }

    //-1
    public synchronized void decrement() throws InterruptedException {
        while (number==0){ // 1
            // 等待
            this.wait(); }
        number--;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我-1完毕了
        this.notifyAll();
    }
}
```

- ReentrantLock + Condition

```
class Data2{
    // 数字 资源类
    private int number = 0;
    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();
    //condition.await(); // 等待
    //condition.signalAll(); // 唤醒全部
    //+1
    public void increment() throws InterruptedException {
        lock.lock();
        try {
            // 业务代码
            while (number!=0){ //0
                // 等待
                condition.await();
            }
            number++;
            System.out.println(Thread.currentThread().getName()+"=>"+number);
            // 通知其他线程，我+1完毕了
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    //-1
    public synchronized void decrement() throws InterruptedException {
        lock.lock();
        try {
            while (number==0){ // 1
                // 等待
                condition.await();
            }
            number--;
            System.out.println(Thread.currentThread().getName()+"=>"+number);
            // 通知其他线程，我-1完毕了
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
```

```
                lock.unlock();
            }
        }
    }
```

- volatile: 多个线程同时监听一个变量，当这个变量发生变化的时候 ，线程能够感知并执行相应的业务。这也是最简单的一种实现方式

16、Lock和synchronized的区别，ReentrantLock和Condition类的使用

- 首先synchronized是关键字，Lock是个java类；
- synchronized是JVM提供的，Lock是JDK提供的;
- synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；
- synchronized会自动加锁解锁，Lock需在手动加锁解锁；
- 用synchronized不能退出等待，而lock如果获取不到锁可以停止等待；
- Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

ReentrantLock和Condition类的使用见 15。

17、信号量，生产者-消费者模式如何实现

见15。

18、如何创建线程池，以及具体的ThreadPoolExcutor的方法参数、拒绝策略

```
ExecutorService threadPool = new ThreadPoolExecutor( 2,5,3,TimeUnit.SECONDS, new
LinkedBlockingDeque<>(3), Executors.defaultThreadFactory(), new
ThreadPoolExecutor.DiscardOldestPolicy());
```

1、corePoolSize：核心线程数
    * 核心线程会一直存活，及时没有任务需要执行
        * 当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理
        * 设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭

2、queueCapacity：任务队列容量（阻塞队列）
  * 当核心线程数达到最大时，新任务会放在队列中排队等待执行

3、maxPoolSize：最大线程数
  * 当线程数>=corePoolSize，且任务队列已满时。线程池会创建新线程来处理任务
    * 当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常

4、 keepAliveTime：线程空闲时间
  * 当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize
    * 如果allowCoreThreadTimeout=true，则会直到线程数量=0

5、allowCoreThreadTimeout：允许核心线程超时
6、rejectedExecutionHandler：任务拒绝处理器

```
* 两种情况会拒绝处理任务：
    - 当线程数已经达到maxPoolSize，切队列已满，会拒绝新任务
    - 当线程池被调用shutdown()后，会等待线程池里的任务执行完毕，再shutdown。如果在调用
shutdown()和线程池真正shutdown之间提交任务，会拒绝新任务
    * 线程池会调用rejectedExecutionHandler来处理这个任务。如果没有设置默认是
AbortPolicy，会抛出异常
    * ThreadPoolExecutor类有几个内部实现类来处理这类情况：
    - AbortPolicy 丢弃任务，抛运行时异常
    - CallerRunsPolicy 主线程执行任务
    - DiscardPolicy 忽视，什么都不会发生
    - DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务
    * 实现RejectedExecutionHandler接口，可自定义处理器
```

19、Atomic原子类及实现机制

```
public static void main(String[] args) {
    AtomicInteger atomicInteger = new AtomicInteger(2020);
    // 期望、更新
    // public final boolean compareAndSet(int expect, int update)
    // 如果我期望的值达到了，那么就更新，否则，就不更新，CAS 是CPU的并发原语
    System.out.println(atomicInteger.compareAndSet(2020, 2021));
    System.out.println(atomicInteger.get());

    atomicInteger.getAndIncrement();
    System.out.println(atomicInteger.compareAndSet(2020, 2021));
    System.out.println(atomicInteger.get());
}
```

底层是CAS(compare and swap)原理，即乐观锁。谨防ABA问题，使用AtomicStampedReference类

```
AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(num, version);
```

20、索引的实现机制

索引是一种数据结构，不是单单的一本书的目录而已，在查询数据时如果不用索引的话就是一条一条比对的查询。如果查询的数据就是千万级别的话，那么效率低下，就需要引入索引来帮助快速查询数据，索引是一条数据的一列或者多列，索引对应地是数据的地址。一般关系型数据库用B/B+树来构建索引（**一层可以存放更多的元**），不用红黑树或者平衡二叉树是因为树的高度还是太高了，磁盘IO读取次数过多。**B+树中叶子结点的指针是为了支持范围查找、模糊查找，所以不用每次查找一个元素都从根节点开始查找了**

存储引擎是修饰在数据库表的，不是数据库的。（InnoDB和MyISAM）

MyISAM：有三个文件，一个存放索引，一个存放数据，一个存放表结构。

查找数据时会先根据索引文件判断索引对应的行记录地址，然后再查找行记录文件找到所对应的行记录。叶子结点存放的数据是**一个行记录的地址指针。**

InnoDB：有两个文件，一个存放表结构，一个是数据和索引的合并文件。（主索引+辅助索引来查询数据）

表数据文件本身就是用B+树来组织的索引结构文件，所以必须要有主键，而且设置为自增的整型。叶子结点存放的数据是**一个完整的行记录数据。**

聚集索引：叶子结点存放的完整的记录数据，所以InnoDB是聚集而MyISAM不是。

21、InnoDB的一些特性

见20。

22、算法题：如何判断链表是否有环

23、算法题：逆序数

24、算法题：判断树的深度

25、手写单例模式

饿汉式

```
package com.kuang.single;

// 饿汉式单例

public class Hungry {
    private Hungry(){ }
    private final static Hungry HUNGRY = new Hungry();
    public static Hungry getInstance(){
        return HUNGRY;
    }
}
```

懒汉式

第一个 if 语句用来避免 uniqueInstance 已经被实例化之后的加锁操作，而第二个 if 语句进行了加锁，所以只能有一个线程进入，就不会出现 uniqueInstance == null 时两个线程同时进行实例化操作。

```java
public class LazyMan {

    private static boolean qinjiang = false;

    private LazyMan(){
        synchronized (LazyMan.class){
            if (qinjiang == false){
                qinjiang = true;
            }else {
                throw new RuntimeException("不要试图使用反射破坏异常");
            }
```

```java
        }
    }

    private volatile static LazyMan lazyMan;

    // 双重检测锁模式的 懒汉式单例  DCL懒汉式
    public static LazyMan getInstance(){
        if (lazyMan==null){
            synchronized (LazyMan.class){
                if (lazyMan==null){
                    lazyMan = new LazyMan(); // 不是一个原子性操作
                }
            }
        }
        return lazyMan;
    }
```

```
    // 反射!
    public static void main(String[] args) throws Exception {
//        LazyMan instance = LazyMan.getInstance();

        Field qinjiang = LazyMan.class.getDeclaredField("qinjiang");
        qinjiang.setAccessible(true);

        Constructor<LazyMan> declaredConstructor =
LazyMan.class.getDeclaredConstructor(null);
        declaredConstructor.setAccessible(true);
        LazyMan instance = declaredConstructor.newInstance();

        qinjiang.set(instance,false);

        LazyMan instance2 = declaredConstructor.newInstance();

        System.out.println(instance);
        System.out.println(instance2);
    }

}


/**
 * 1. 分配内存空间
 * 2、执行构造方法，初始化对象
 * 3、把这个对象指向这个空间
 *
 * 123
 * 132 A
 *     B // 此时lazyMan还没有完成构造
 */
```

静态内部类

```
package com.kuang.single;

// 静态内部类
public class Holder {
    private Holder(){

    }

    public static Holder getInstace(){
        return InnerClass.HOLDER;
    }

    public static class InnerClass{
        private static final Holder HOLDER = new Holder();
    }

}
```

枚举

```
public enum EnumSingle {

    INSTANCE;

    public EnumSingle getInstance(){
        return INSTANCE;
    }

}
```

26、Volatile，详细说了一下指令重排和内存屏障

指令重排：**你写的程序，计算机并不是按照你写的那样去执行的。**

处理器在进行指令重排的时候，考虑：数据之间的依赖性！

```
int x = 1; // 1
int y = 2; // 2
x = x + 5; // 3
y = x * x; // 4

我们所期望的：1234  但是可能执行的时候回变成 2134  1324
可不可能是  4123!
```

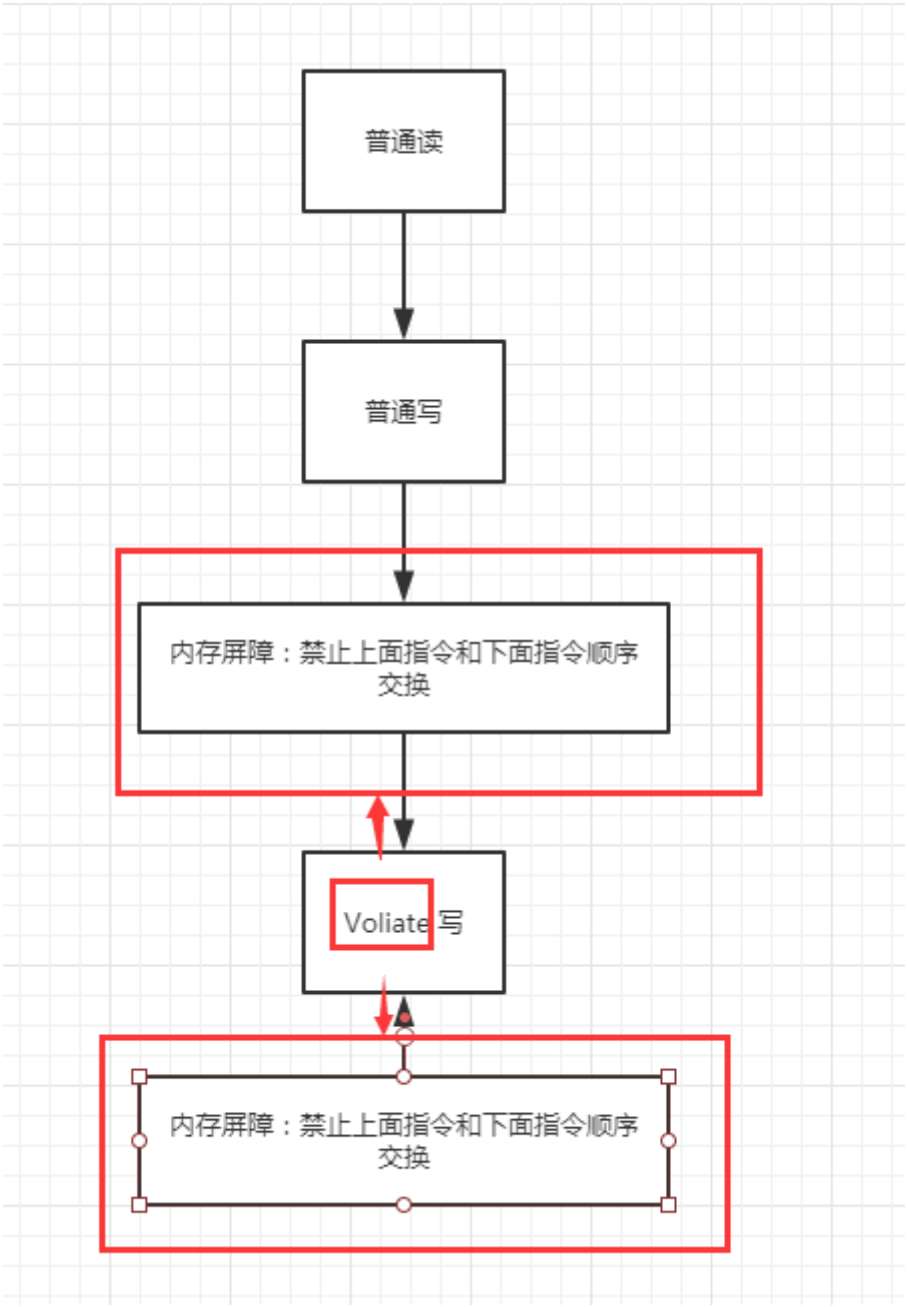
可能造成影响的结果： a b x y 这四个值默认都是 0；

线程A	线程B
x=a	y=b
b=1	a=2

正常的结果： x = 0；y = 0；但是可能由于指令重排

线程A	线程B
b=1	a=2
x=a	y=b

指令重排导致的诡异结果： x = 2；y = 1；



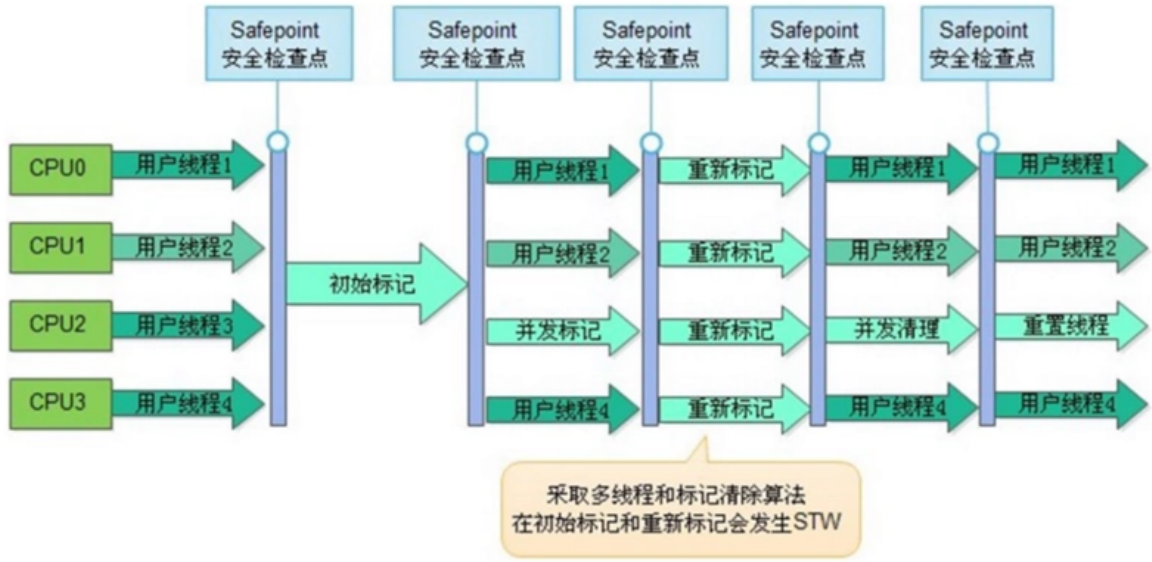
27、垃圾回收器，介绍一下cms和g1

	CMS	G1
JDK版本	1.6以上	1.7以上
回收算法	标记——清除	标记——整理
运行环境	针对70G以内的堆内存	可针好几百G的大内存
回收区域	老年代	新生代和老年代
内存布局	传统连续的新生代和老年代区域	Region(将新生代和老年代切分成Region，默认一个Region 1 M,默认2048块)
浮动垃圾	是	否
内存碎片	是	否
全堆扫描	是	否
回收时间可控	否	是
对象进入老年代的年龄	6	15
空间动态调整	否	是（新生代5%-60%动态调整，一般不需求指定）

1、CMS

CMS(Concurrent Mark Sweep)，我们可以轻易地从命名上看出，它是一个并发的，然后是基于标记——清理的垃圾回收器，它清理垃圾的步骤大致分为四步：

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清理



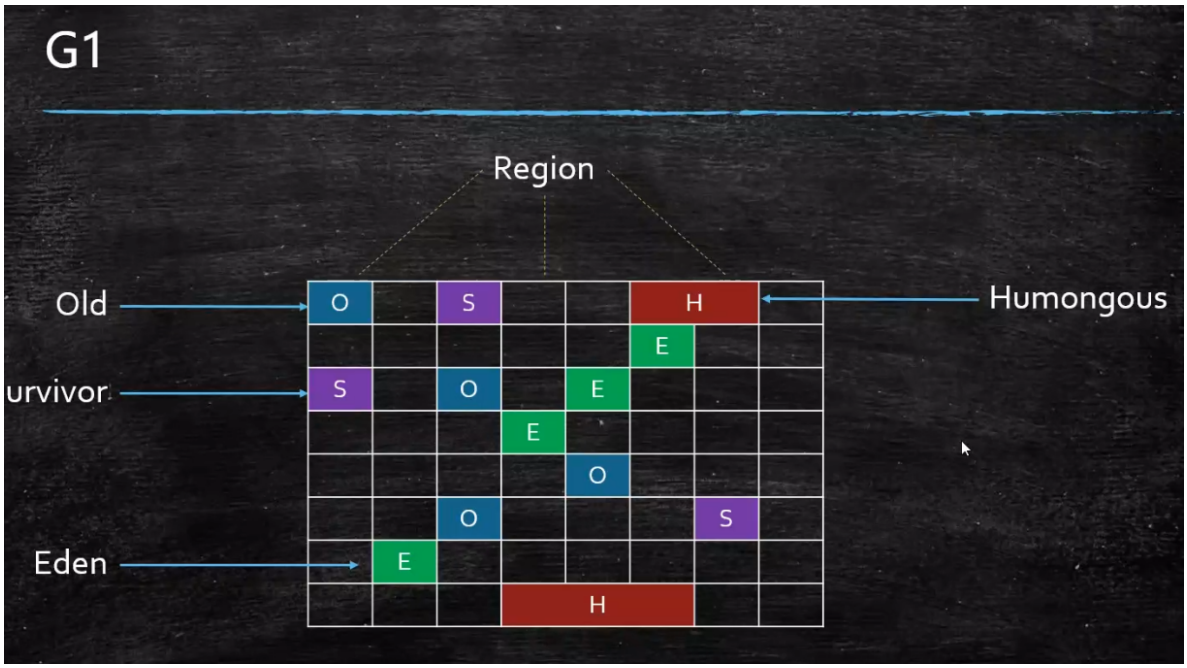
初始标记只要是找到GC Roots，所以是一个很快的过程，并发标记和用户线程一起，通过GC Roots找到存活的对象，重新标记主要是修复在并发标记阶段的发生了改变的对象，这个阶段会Stop the World；

并发清理则是保留上一步骤标记出的存活对象，清理掉其他对象，正因为采用并发清理，所以在清理的过程中用户线程又会产生垃圾，而导致浮动垃圾，只能通过下次垃圾回收进行处理；

因为cms采用的是标记清理，所以会导致内存空间不连续，从而产生内存碎片

2.G1

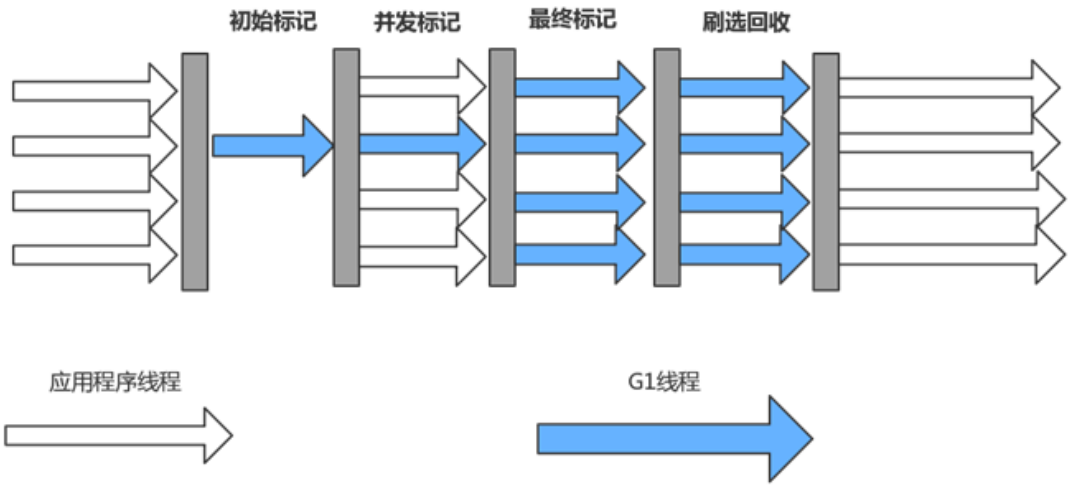
G1(Garbage-First)，以分而治之的思想将堆内存分为若干个等大的Region块，虽然还是保留了新生代，老年代的概念，但是G1主要是以Region为单位进行垃圾回收，G1的分块大体结果如下图所示：



G1垃圾回收器的它清理垃圾的步骤大致分为四步：

1. 初始标记
2. 并发标记

3. 最终标记
4. 复制回收



初始标记和并发标记和CMS的过程是差不多的，最后的筛选回收会首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划

因为采用的标记——整理的算法，所以不会产生内存碎片，最终的回收是STW的，所以也不会有浮动垃圾，Region的区域大小是固定的，所以回收Region的时间也是可控的

同时G1 使用了Remembered Set来避免全堆扫描，G1中每个Region都有一个与之对应的RememberedSet，在各个 Region 上记录自家的对象被外面对象引用的情况。当进行内存回收时，在GC根节点的枚举范围中加入RememberedSet 即可保证不对全堆扫描也不会有遗漏。

28、mysql引擎，行锁表锁

innodb行级锁是依赖于索引实现的，where条件后的字段只有是索引才可以应用于行级锁，否则是表级锁。

不通过索引就是表级锁。

29、算法题：两个链表怎样判断是否相交

30、linux命令，grep是干什么的,vi里面什么命令能跳到头部

grep 命令用于查找文件里符合条件的字符串。

```
#跳到头 是数字1不是字母l
:1

#跳到尾
:$
```

31、并发怎么实现，runnable优势是什么。

通过多线程、线程池。

①. 继承Thread类创建线程类

- 定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。
- 创建Thread子类的实例，即创建了线程对象。
- 调用线程对象的start()方法来启动该线程。

②. 通过Runnable接口创建线程类

- 定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
- 创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
- 调用线程对象的start()方法来启动该线程。

③. 通过Callable和Future创建线程

- 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
- 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
- 使用FutureTask对象作为Thread对象的target创建并启动新线程。
- 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

```
//SumTotal实现了Callable接口
SumTotal sumTotal = new SumTotal();
//创建FutureTask
FutureTask<Integer> futureTask = new FutureTask<Integer>(sumTotal);
//执行线程任务
Thread thread = new Thread(futureTask);
thread.setName("线程1");
thread.start();

//获取线程任务返回值
try {
    int sum = futureTask.get();
    System.out.println(sum);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

实现Runnable接口比继承Thread类所具有的优势：

（1）适合多个相同的程序代码的线程去处理同一个资源，线程任务和对象分离。

（2）可以避免java中的单继承的限制，类继承了Thread类就不能继承其他的类

（3）增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

32、java反射类载入函数，怎么调用方法，用反射的优缺点。

```
public class MethodDemo1 {

    public static void main(String[] args){

        // 获取 print（int, int） 方法
        // 1 要获取一个方法就是获取类的信息，获取类的信息


        A a1=new A();
        Class c=a1.getClass();

        /*
          2 获取方法 名称和参数列表决定
          getmethod 获取pulic 的方法
         */

        try {
            //Method m=c.getMethod("print",new Class[]{int.class,int.class});
            Method m=c.getMethod("print",int.class,int.class);

            // 方法的反射  作用是用m对象来进行方法调用
            // 以前 a1.print(10,20)
            // 方法如果没有返回值 返回null 有返回值返回具体的返回值
           // Object o = m.invoke(a1,new Object[]{10,20});

            Object o = m.invoke(a1,10,20);

            System.out.println("===============");
            Method m1=c.getMethod("print",String.class,String.class);

            o=m1.invoke(a1,"hello","world");



        } catch (Exception e) {
            e.printStackTrace();
        }



    }
}

class A{
    public void print(int a,int b){
        System.out.println(a+b);
    }

    public void print(String a, String b){
        System.out.println(a.toUpperCase()+" "+b.toLowerCase());
    }
}
```

反射的缺点：

1、使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码

2、使用反射会模糊程序内部逻辑，反射绕过了源代码的技术。

3、破坏了封装性，使得private没有意义了。

33、怎么写java注解

```
@Target(ElementType.TYPE)//TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR
@Retention(RetentionPolicy.RUNTIME) //SOURCE,RUNNTIME,CLASS
public @interface MyAnnotation1 {
}
```

34、重写与重载的区别，可以重写父类的构造函数吗；

重载和重写**都是多态**的一种体现方式。

1、重载是编译期间的活动，重写是运行期间的活动。

2、重载是在一个类中定义相同的名字的方法，方法的参数列表或者类型要互相不同，但是返回值类型不作为是否重载的标准，可以修改可见性；

　　重写是不同的，要求子类重写基类的方法时要与父类方法具有相同的参数类型和返回值，可见性需要大于等于基类的方法

关于可见性：

子类重写父类方法，返回值为基本类型时，必须相同；返回值为引用类型时，要小于等于父类。

同理，子类抛出的异常也要小于等于父类； 子类重写方法的修饰词要大于等于父类； 两小、一大、一同原则

java中子类无法继承父类构造函数，因此不能重写父类构造函数，但是可以在子类构造函数中通过super（）调用父类构造函数.

35、String可以被继承吗?

不能被继承，因为String类有final修饰符，而final修饰的类是不能被继承的。

36、线程创建方式，callable和其他两种的创建方式有什么不一样?

见31。

37、jvm内存分区，方法区除了静态方法，常量这些还放什么?

方法区存放着静态变量、字符串常量、运行时常量（即class类信息，一个类被加载时所有的符号引用都会存在这里）

38、可达性分析算法GCroots是哪些对象?

GC roots: 一般为方法区中常量引用/类中静态引用的对象、栈中局部变量引用的对象等。

39、应用层有哪些是TCP，哪些是udp?

1、基于TCP的应用层协议有：HTTP、FTP、SMTP、TELNET、SSH
2、基于UDP的应用层协议：DNS、TFTP（简单文件传输协议）、SNMP：简单网络管理协议

40、tcp的拥塞控制，窗口具体是怎么变化的?

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

- 接收方有足够大的接收缓存，因此不会发生流量控制；
- 虽然 TCP 的窗口基于字节，但是这里设窗口的大小单位为报文段。

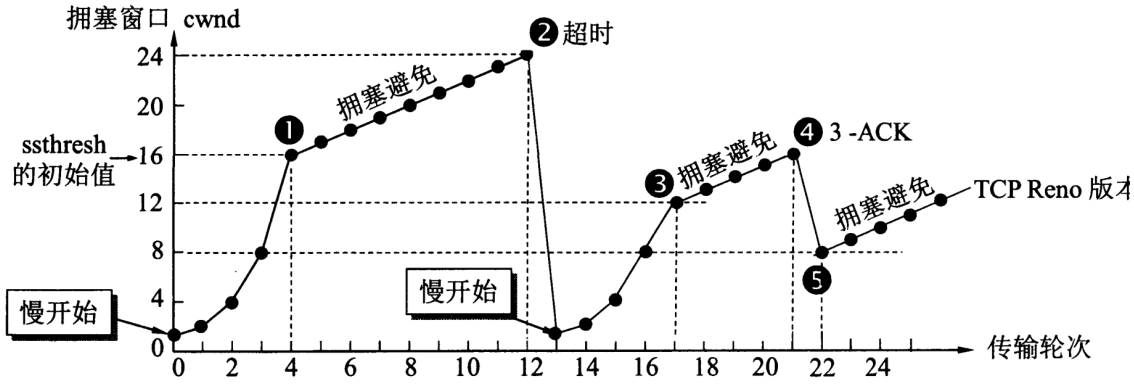


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

1. 慢开始与拥塞避免

发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。

如果出现了超时，则令 ssthresh = cwnd / 2，然后重新执行慢开始。

2. 快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2 ，cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

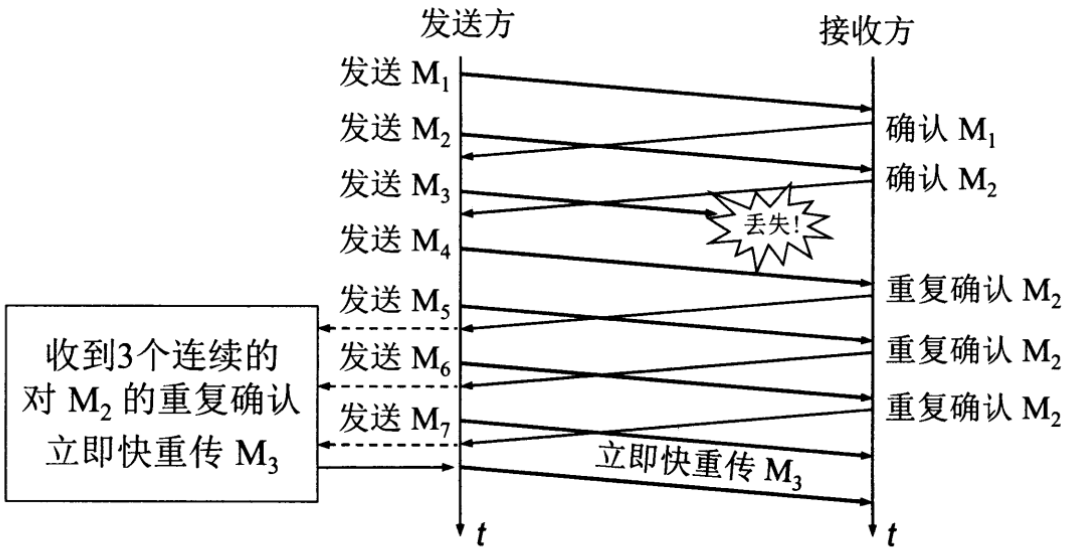


图 5-26　快重传的示意图

41、mysql索引结构？哪些字段适合创建索引，update多的适合创建索引吗?

b+树。

哪些情况适合建索引?

1. 主键自动建立唯一索引。
2. 频繁作为查询条件的字段应该创建索引。
3. 查询中与其他表关联的字段，外键关系建立索引。
4. 在高并发下倾向创建组合索引。
5. 查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度。（索引干两件事：检索和排序）。
6. 查询中统计或者分组字段。

哪些情况不适合建索引?

1. 表记录太少。
2. 经常增删改的表的字段。
3. 如果某个数据列包含许多重复的内容，为它建立索引就没太大的实际效果。

update多不适合。

42、MySQL语句中：where join limit group by和order by的执行顺序

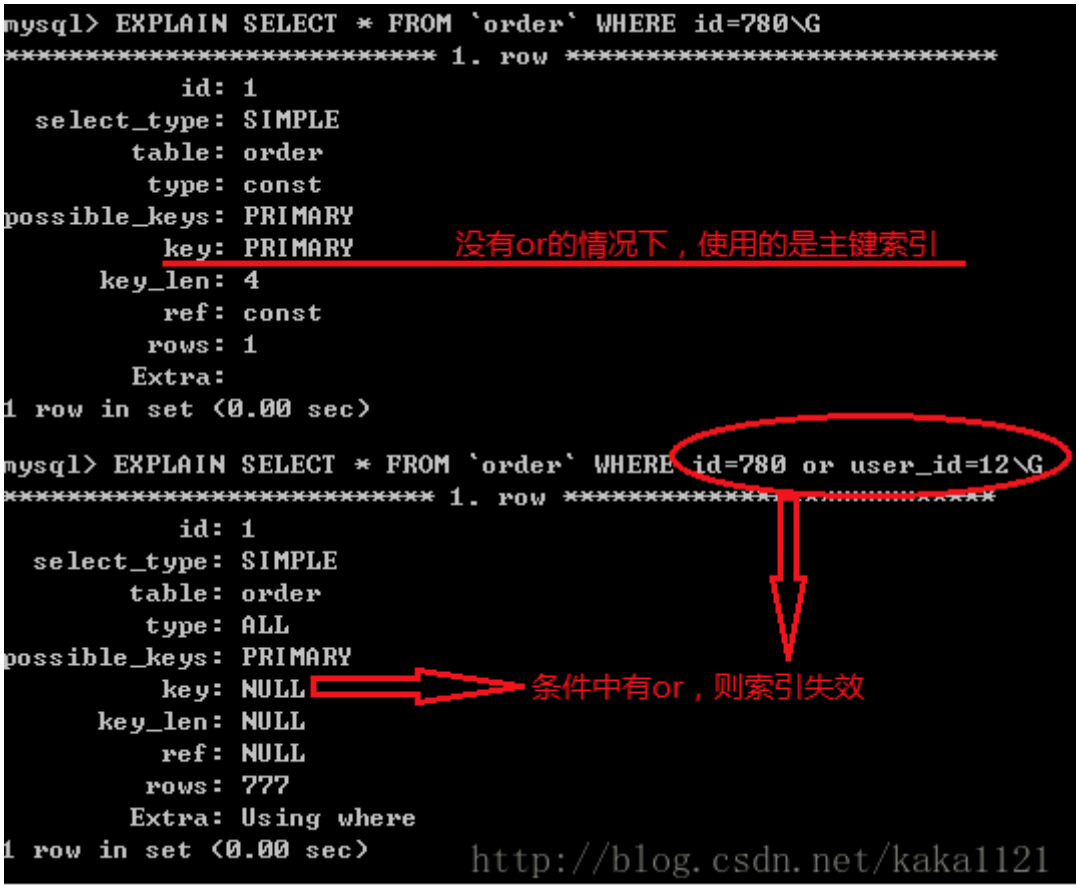
关键字或 解释	执行顺序
select 查询列表（字段）	第七步
from 表	第一步
连接类型 join 表2	第二步
on 连接条件	第三步
where 筛选条件	第四步
group by 分组列表	第五步
having 分组后的筛选条件	第六步
order by 排序列表	第八步
limit 偏移，条目数	第九步

43、索引失效的情况有哪些?

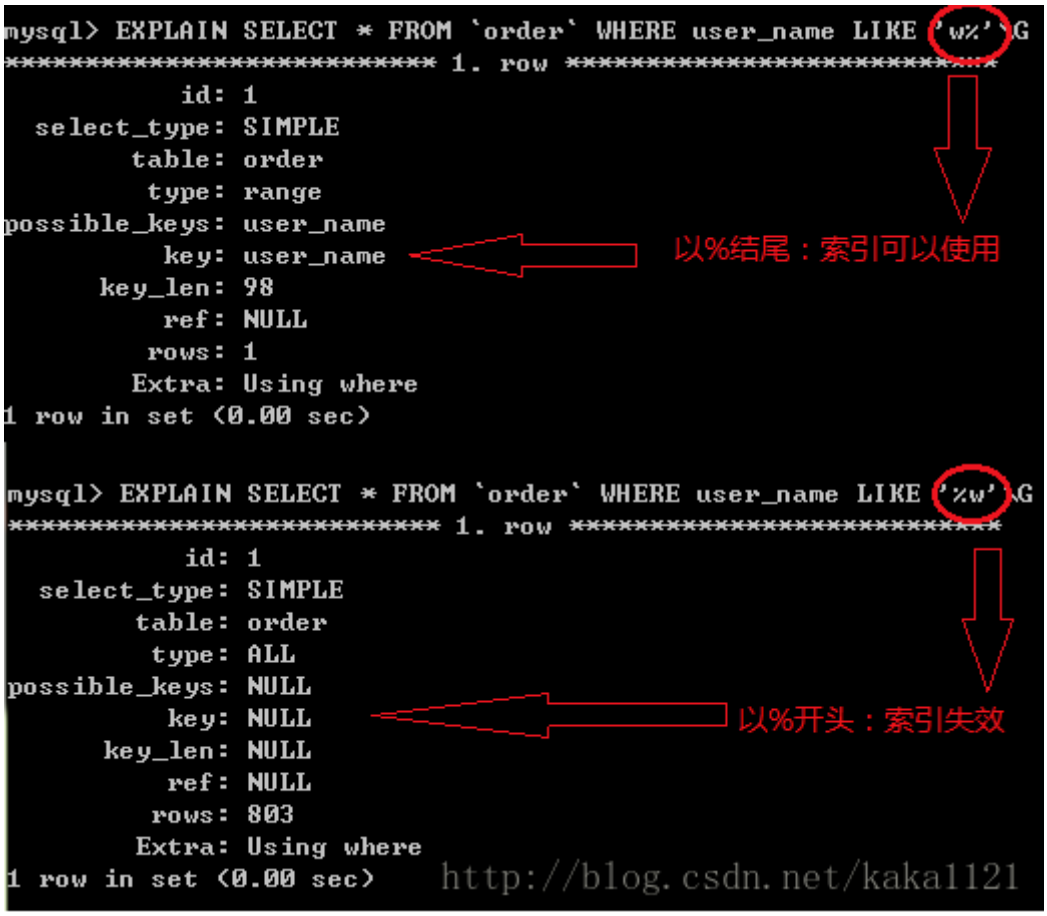
索引并不是时时都会生效的，比如以下几种情况，将导致索引失效：

- 如果条件中有or，即使其中有部分条件带索引也不会使用(这也是为什么尽量少用or的原因)，例子中user_id无索引

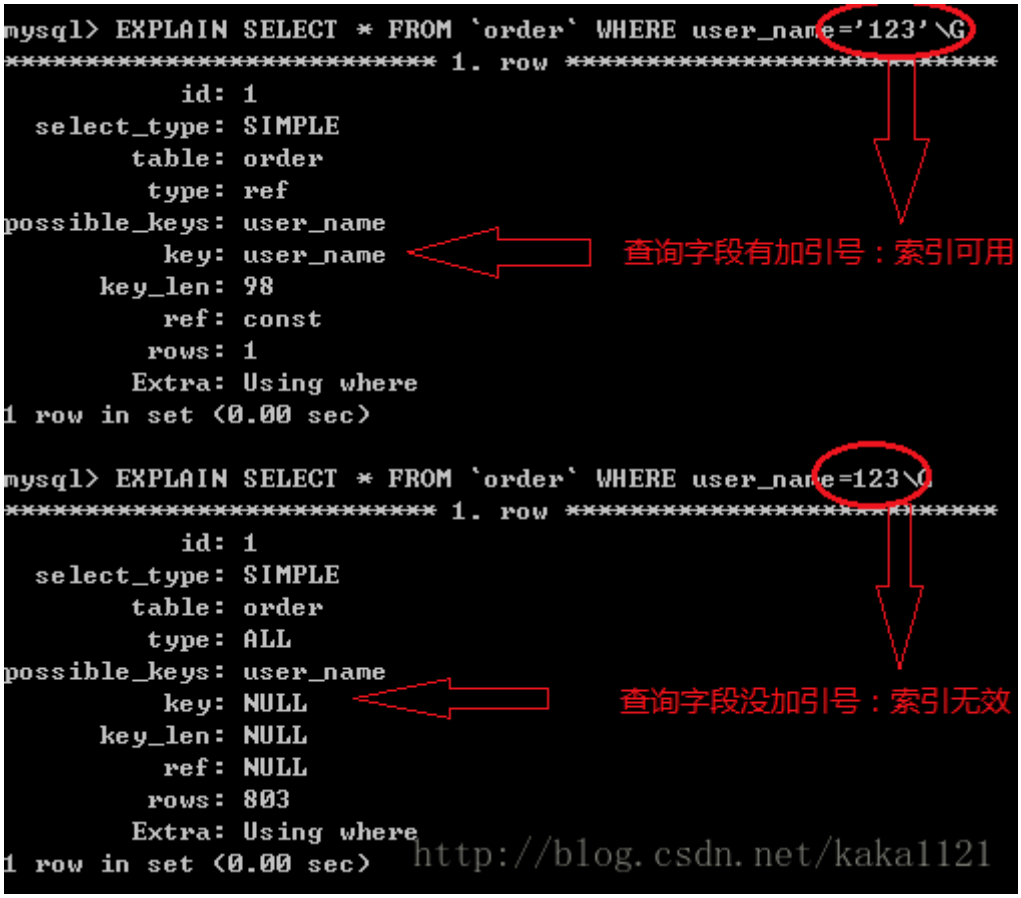
注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引



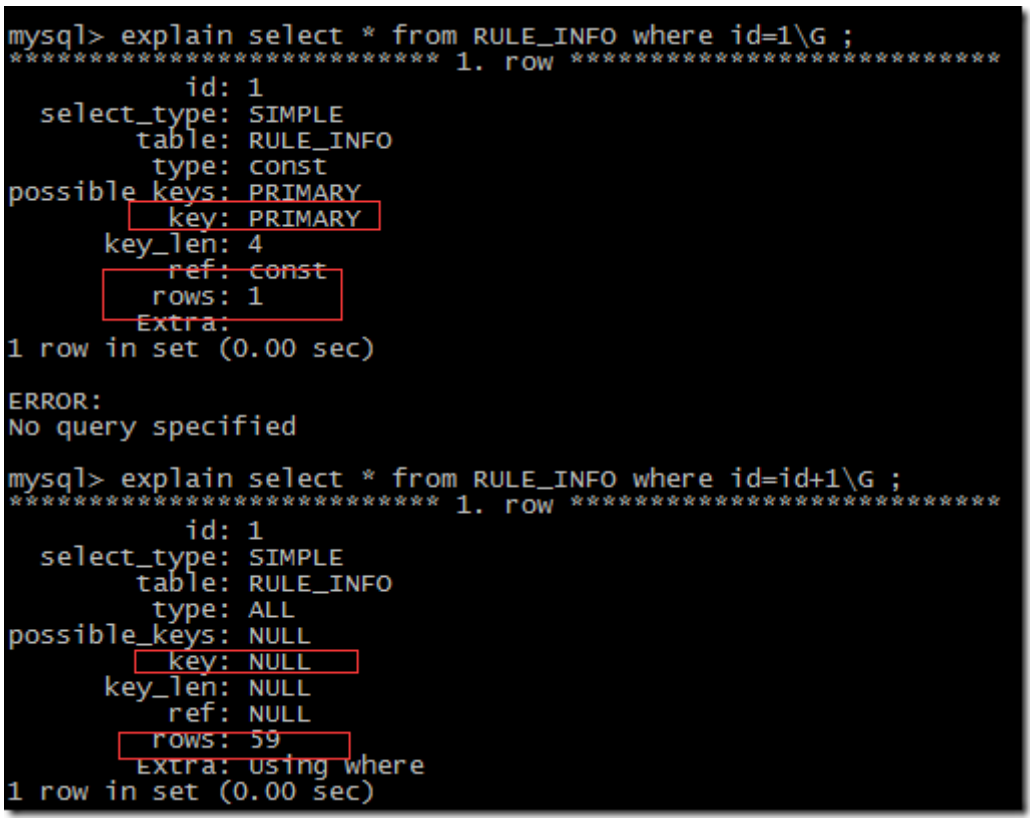
- 对于复合索引，如果不使用前列，后续列也将无法使用，类电话簿。
- like查询是以%开头



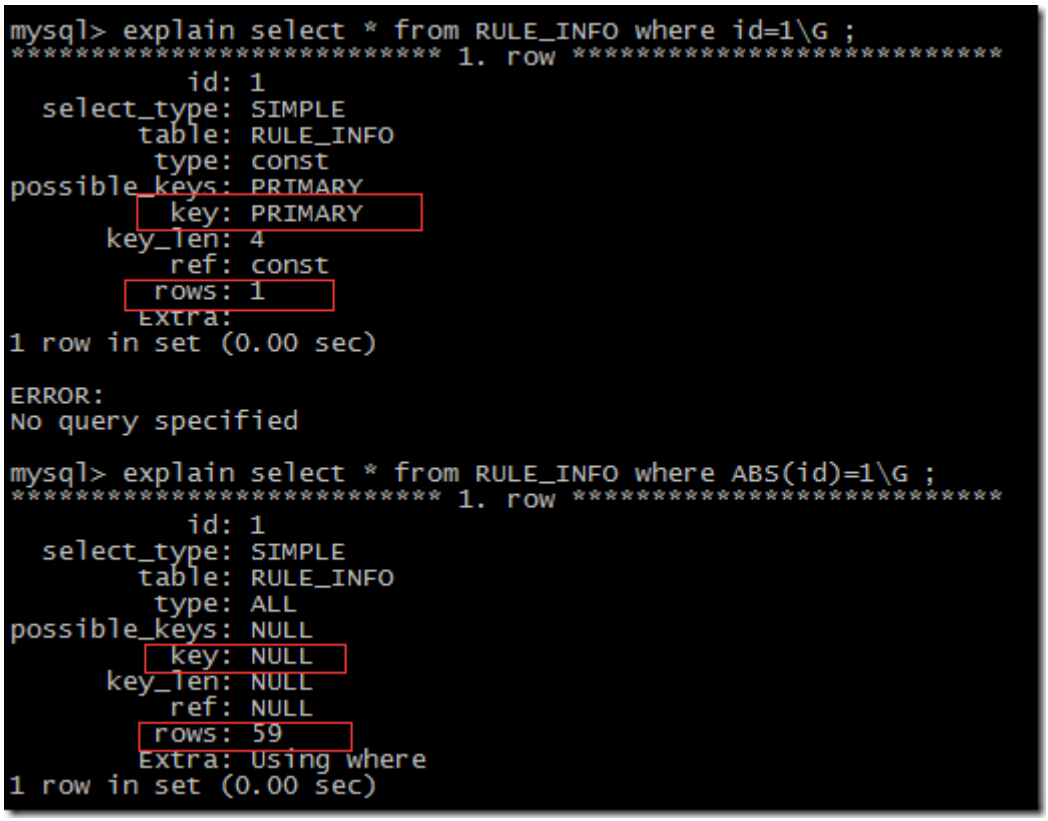
- 存在索引列的数据类型隐形转换，则用不上索引，比如列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引



- where 子句里对索引列上有数学运算，用不上索引



- where 子句里对有索引列使用函数，用不上索引



- 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

> 比如数据量极少的表

44、mysql锁机制，读取比较多适合用什么锁?

行锁和表锁。按照使用方式分为共享锁和排他锁。

如果只是频繁读，没有其他用户修改表中大量数据，行级锁就适合（即可重复读原理），如果频繁读的时候有用户会修改大量数据引发幻影读问题就用表锁。

45、循环队列能实现栈和队列吗?

可以。实现队列的话，删除元素从头指针开始，实现栈的话，删除元素从尾指针开始。

46、使用多线程的优势。线程数怎么设定？过大会怎么样？多线程有什么缺点？怎么实现线程安全?

优点：

1、进程之间不能共享内存，但线程之间可以，且十分容易

2、创建进程是需要为该进程重新分配系统资源，但创建线程则代价小得多。

3、多线程可以分别设置优先级以优化性能。

缺点：

1、线程管理要额外的 CPU开销，线程的使用会给系统带来上下文切换的额外负担。

2、对共享资源加锁实现同步的过程中可能会死锁。

3、对公有变量的同时读或写，可能对造成脏读等。

线程数设定：

1、CPU 密集型: 几核，就是几，可以保持CPU的效率最高!

2、IO 密集型：判断你程序中十分耗IO的线程数目，例如：如果有15个非常占用资源的IO线程，那么设置为30，这样可以保证还有其他线程正常执行，不易阻塞。

当我们的线程数量配置的过大，我们的线程与线程之间有会争取 CPU 资源，这就会导致上下文切换。

如果我们设置的线程池数量太小的话，如果同一时间有大量任务/请求需要处理，可能会导致大量的请求/任务在任务队列中排队等待执行，甚至会出现任务队列满了之后任务/请求无法处理的情况，或者大量任务堆积在任务队列导致 OOM。

线程安全：

ThreadLocal、volatile、synchronized、lock、Atomic

47、java面向对象四大特性

封装，多态，继承，抽象。

48、抽象类和接口的区别。abstract不能用什么关键字修饰

实现：抽象类的子类使用 extends 来继承；接口必须使用 implements 来实现接口。

构造函数：抽象类可以有构造函数；接口不能有。

main 方法：抽象类可以有 main 方法，并且我们能运行它；接口不能有 main 方法。

实现数量：类可以实现很多个接口；但是只能继承一个抽象类。

访问修饰符：接口中的方法默认使用 public 修饰；抽象类中的方法可以是任意访问修饰符。

不能用final

49、多态的概念，多态的具体表现形式和实例，假设父类某个方法是public情况下子类重写方法为protected，会出现什么问题和错误？父类有异常捕捉的情况下，子类对这些异常如何处理？

多态就是同一个接口，使用不同的实例而执行不同操作。重载和重写都是多态。

如果访问权限变小了，编译器会报错，无法编译。

只能抛出更小的异常或者不抛出。

50、JAVA捕获异常的顺序

对于try里面发生的异常，他会根据发生的异常和catch里面的进行匹配(怎么匹配，按照catch块从上往下匹配)，当它匹配某一个catch块的时候，他就直接进入到这个catch块里面去了，后面在再有catch块的话，它不做任何处理，直接跳过去，全部忽略掉。如果有finally的话进入到finally里面继续执行。换句话说，如果有匹配的catch，它就会忽略掉这个catch后面所有的catch。

51、线程池的概念，作用和目的。

作用：

1、降低资源的消耗

2、提高响应的速度

3、方便管理。

线程复用、可以控制最大并发数、管理线程

52、数据库 索引有什么优缺点

优点：

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
第二，可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。
第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
第四，在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

缺点：

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

53、数据库怎么优化

1、主从结点实现分离读写，采取主从复制把数据库的读操作和写操作分离出来。

2、尽量把字段设置为NOT NULL，这样在将来执行查询的时候，数据库不用去比较NULL值。

3、创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。

4、索引。**索引应建立在那些将用于JOIN,WHERE判断和ORDERBY排序的字段上。尽量不要对数据库中某个含有大量重复的值的字段建立索引。**

5、使用优化的SQL语句，避免索引失效。

6、用join替代子查询（子查询指把一个查询的结果在另一个查询中使用就叫做子查询）。连接（JOIN）之所以更有效率一些，是因为MySQL不需要在内存中创建临时表

54、如何保证不同数据库之间的数据一致性。不追求即时同步，只要两个数据库在事务操作完成之后最终数据是同步的就可以。

如果是redis和mysql：

1.第一种方案：采用延时双删策略

1. 先删除缓存
2. 再写数据库
3. 休眠500毫秒
4. 再次删除缓存

2、第二种方案：异步更新缓存(基于订阅binlog的同步机制)

一旦MySQL中产生了新的写入、更新、删除等操作，就可以把binlog相关的消息推送至Redis，Redis再根据binlog中的记录，对Redis进行更新。其实这种机制，很类似MySQL的主从备份机制，因为MySQL的主备也是通过binlog来实现的数据一致性。

如果是多个mysql:

- 从节点上的**I/O 线程**连接主节点，并请求日志文件的指定位置后的日志内容；
- 主节点接收到来自从节点的I/O请求后，主节点的**binary log dump 线程**读取指定日志指定位置之后的日志信息，返回给从节点。返回信息中包括bin log和bin log position；从节点的I/O进程接收到内容后，将接收到的日志内容更新到本机的relay log中，并将读取到的binary log文件名和位置保存到master-info 文件中；
- 从节点的 **SQL线程**检测到relay-log 中新增加了内容后，会将relay-log的内容解析成在从节点上实际执行过的操作，并在本数据库中执行。

55、Integer a = 128; Integer b = 128; a==b？ 为什么是false? Integer a = 127; Integer b = 127; a==b？ 为什么是true

当Integer的值为-128~127时，变量的值不会新建内存而是直接引用方法区常量池中的值。

56、float编码方式?

57、utf-8和GBK区别

编码的原因是需要让计算机理解我们所表达的符号。而计算机中最小的单位是字节，8bit，只能表示256个字符，远不够我们所拥有的，所以需要一个从字符到字节的编码（即char到byte）。通常在IO或者内存中操作时需要编码。Java中默认是UFT-16（双字节定长表示）

GBK和UTF-8：

长度：GBK为2字节，UTF-8为1-6字节。

用途：GBK主要用于中文，UTF-8包含了世界各地的字符。

占用空间：总的空间UTF-8更少，所以打开一个网页速度更快。

58、sychronized用在普通方法和静态方法的区别

普通方式锁对象，静态方法锁类名（即class类）

59、sychronized的底层

synchronized底层原理就是应用了信号量。

每个对象都会有一个monitor锁，如果monitor的进入数为0，则该线程进入monitor（**monitorenter**），然后将进入数设置为1，该线程即为monitor的所有者。如果其他线程要进入monitor时，已经有线程占有了该monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。线程完成了指定操作后会执行退出monitor操作(**monitorexit**)使得

monitor减一。

60、HashMap扩容问题（1.7和1.8的不同）

死锁原因（Java8之前）：多线程扩容导致的死锁

HashMap是非线程安全，死锁一般都是产生于并发情况下。我们假设有二个进程T1、T2，HashMap容量为2,T1线程放入key A、B、C、D、E。在T1线程中A、B、C Hash值相同，于是形成一个链接，假设为A->C->B，而D、E Hash值不同，于是容量不足，需要新建一个更大尺寸的hash表，然后把数据从老的Hash表中迁移到新的Hash表中(refresh)。这时T2进程闯进来了，T1暂时挂起，T2进程也准备放入新的key，这时也发现容量不足，也refresh一把。refresh之后原来的链表结构假设为**C->A（头插法）**，之后T1进程继续执行，链接结构为A->C,这时就形成A.next=C,C.next=A的环形链表。一旦取值进入这个环形链表就会陷入死循环。

JAVA8**取消了头插法解决了死锁**，并且也不用rehash来扩容。方法如下：首先假设扩容前容量为8，其中的一个链表假设为A->B->C->D，用四个指针来直接完成迁移扩容，先计算ABCD中的部分hashcode && 8是否为1，如果不为1则代表为低位元素，扩容后还是在指定的key处，可以通过低位Low指针和低位high指针直接迁移过去；如果为1则代表为高位元素，扩容后放到key + 8的位置处，可以通过高位Low指针和高位high指针迁移至key+8处。

61、MySQL默认隔离级别是哪种

提交读。

可重复读存在间隙，锁会使死锁的概率增大，在可重复读隔离级别下，条件列未命中索引会锁表！而在提交读隔离级别下，只锁行

62、如何看索引是否命中

通过语句：

```
explain select col_name from table_name;
```

63、什么时候触发FULL GC？如何手动触发。日常写代码时如何避免频繁的full gc？哪些情况会发生full gc?

老年代的区域满了以后就会执行Full GC对整个堆中的垃圾对象进行清理。当执行Full GC时会让所有线程进入STW状态（Stop the world）即停止状态，以便彻底清理垃圾对象。因为如果不这么设计的话，当从一个线程的局部变量出发判断非垃圾对象时，此时没垃圾对象就去判断下一个线程了，但是如果刚刚那个线程不久就结束了工作回收了局部变量，那么刚刚那个线程的GC roots可达的非垃圾对象就变成了垃圾对象了，此时已经不能回收了。所以JVM调优的目的就是要减少STW次数，STW次数过多会影响程序的用户体验。

System.gc()主动触发，只是建议，不一定会执行。

触发条件：

1、老年代（Tenured Gen）空间不足

2、元空间不足

3、老年代连续空间不足

4、堆中产生的对象过大，新生代放不下就放到老年代了。

避免方案：

1、缩短对象生命周期，避免对象被添加到old代。（避免大方法、避免一个变量跨方法调用，且被不同方法调用、对象不用了要及时的清空对象）

2、避免创建过大的对象，例如长字符串。

64、什么是对象序列化，如何实现

JAVA序列化就是将JAVA对象转化成字节流的过程，存储在磁盘或者网络中，以便再次读取或者传输。

```
public static void main(String[] args) throws Exception {
        File file = new File("person.out");
        ObjectOutputStream oout = new ObjectOutputStream(new
FileOutputStream(file)); // 注意这里使用的是 ObjectOutputStream 对象输出流封装其他的输出流
        Person person = new Person("John", 101, Gender.MALE);  //实现了
Serializable
        oout.writeObject(person);
        oout.close();

        ObjectInputStream oin = new ObjectInputStream(new
FileInputStream(file));  // 使用对象输入流读取序列化的对象
        Object newPerson = oin.readObject(); // 没有强制转换到Person类型
        oin.close();
        System.out.println(newPerson);
    }
```

65、关系型数据库和非关系型数据库区别

1、性能。

2、数据类型。

NOSQL是基于键值对的，不需要经过SQL层的解析。

SQL支持复杂查询，和事务。

3、数据存储结构（数据关联性）。

66、java的命名规则

命名规则:
1.由字母、数字、下划线、美元符号组成 $_
2.不能以数字开头
3.Java严格区分大小写
4.不能Java中的关键字

规范:
1.见名知意 (例如学生类: Student)
2.不允许使用中文和拼音
3.满足驼峰命名法

67、举例9个运行时异常

- NullPointerException：当应用程序试图访问空对象时，则抛出该异常。
- SQLException：提供关于数据库访问错误或其他错误信息的异常。
- IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
- FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。
- IOException：当发生某种I/O异常时，抛出此异常。此类是失败或中断的I/O操作生成的异常的通用类。
- ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。
- IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。
- RuntimeException：是那些可能在Java虚拟机正常运行期间抛出的异常的超类。
- ConcurrentModificationException：当方法检测到对象的并发修改。

68、讲一下spring中的设计模式?

1、工厂模式（简单工厂）：

BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

2、单例模式

Spring下默认的bean均为singleton

3、代理模式

AOP底层就是用的代理和反射。

69、说一下spring事务的原理

通过使用AOP来完成。

JDBC事务

```
<bean
id="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTran
sactionManager">
        <property name="dataSource" ref="dataSource" />
</bean>
```

配置好事务管理器后我们需要去配置事务的通知

```
<!--配置事务通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--配置哪些方法使用什么样的事务,配置事务的传播特性-->
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="search*" propagation="REQUIRED"/>
        <tx:method name="get" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

配置AOP

导入aop的头文件!

```
<!--配置aop织入事务-->
<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* com.kuang.dao.*.*
(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

70、为什么想来招银

71、插入1到1百万个数内存怎么申请

位图法。

72、java匿名内部类

通过创建一个接口或者抽象类，然后重写其中方法。

73、电脑突然关机了，开机后数据还在，属于ACID中哪个性质?

持久性。

74、java泛型

泛型其实就是在定义类、接口、方法的时候不局限地指定某一种特定类型，而让类、接口、方法的调用者来决定具体使用哪一种类型的参数。泛型信息存在于编译阶段，运行阶段就消失了——这种现象被称为“类型擦除”。

好处：

1、安全：在编译的时候检查类安全

2、消除强制类型转换。所有强制转换都是自动和隐式的，提高代码的重用率。

75、一万个数据，如何取前k个

用一个最大堆，堆大小设置为k。

76、子类能访问父类的哪些成员？父类和子类的初始化顺序。

1、能够访问标为public protected的成员变量和方法；
2、如果子类与父类在同一包内，还能访问默认（无修饰符）的成员变量与方法。
3、不能访问标为private的成员。

先初始化**父类的静态代码**--->初始化**子类的静态代码**-->
(创建实例时,如果不创建实例,则后面的不执行)初始化**父类的非静态代码**（变量定义等）--->初始化**父类构造函数**--->初始化**子类非静态代码**（变量定义等）--->初始化**子类构造函数**

77、静态方法怎么初始化。

静态方法在装载类的时候已经初始化，装载时和本类有关的所有符号引用都会存在运行时常量池中。

78、collection和map是一种吗?

不是。collection有三种，list、set和queue。

79、什么时候会用到map?

如果你需要通过一个名字去获取数据的时候就可以用Map。
如果你只是从数据库查询出来，很多条数据，那就放到List。

80、hashmap怎么实现key不重复？每个key的hashcode值不同吗？名字和年龄合并当做key要怎么实现?

首先通过hashcode获得一个int值，范围大小为40亿几乎不重复，然后通过hashcode & (n-1)使得均匀映射到每一个Key中。不一定，不同key的hashcode可能一样，和冲突策略有关。

封装成一个对象，分别有两个成员变量名字和年龄。

81、http和https的区别?

HTTP 有以下安全性问题：

使用明文进行通信，内容可能会被窃听；

不验证通信方的身份，通信方的身份有可能遭遇伪装；

无法证明报文的完整性，报文有可能遭篡改。

HTTPS 并不是新协议，而是让 HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信。

加密：使用非对称密钥加密方式，传输对称密钥加密方式所需要的 Secret Key，从而保证安全性;

获取到 Secret Key 后，再使用对称密钥加密方式进行通信，从而保证效率。

认证：通过使用 证书 来对通信方进行认证。数字证书认证机构（CA）是客户端与服务器双方都可信赖的第三方机构。服务器的运营人员向 CA 提出公开密钥的申请，CA 在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名，然后分配这个已签名的公开密钥，并将该公开密钥放入公开密钥证书后绑定在一起。

完整性保护：SSL 提供报文摘要功能来进行完整性保护。HTTP 也提供了 MD5 报文摘要功能。

82、https协议怎么实现?

https过程：数据加密、验证签名、保证报文完整性。

数字签名：

一个报文通过hash得到报文摘要，然后被私钥加密成为数字签名，接收方得到数字签名后用公钥进行解密得到报文摘要，将接受的报文hash后的摘要和解密得到的摘要比对来验证数字签名。

数字证书：

为了保证接收方的公钥确实是发送方的公钥，通过第三方机构证书中心来认证。证书中心用**自己的私钥**，对发送方的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate），随数字签名一起发送。接收方通过用发送发的公钥以及证书中心的公钥来解密。

83、为什么要先用非对称算法加密和对称加密算法?

对称：

- 优点：运算速度快；
- 缺点：无法安全地将密钥传输给通信方。

非对称：

- 优点：可以更安全地将公开密钥传输给通信发送方；
- 缺点：运算速度慢。

84、数据库怎么实现数据的并发?

设置隔离级别。

85、隔离级别怎么解决那些并发一致性问题的?

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。（更新数据时加入了共享锁即读锁，只能读不能写，解决丢失修改）

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。（读数据时加入了共享锁但是读操作完了以后就释放共享锁，使得不能重复读，更新数据时加入了排他锁即写锁，解决脏读的问题）

可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同一数据的结果是一样的。（读数据时加入了共享锁，一直到事务执行完毕才释放共享锁，这样就可以解决不可重复读问题，更新数据时加入了排他锁即写锁。）

可串行化（SERIALIZABLE）

强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题。

该隔离级别需要加锁实现，因为要使用加锁机制保证同一时间只有一个事务执行，也就是保证事务串行执行。（读数据时对**整个数据表**加入了共享锁，更新数据时对**整个数据表**加入了排他锁即写锁，解决幻影读问题）

86、联合索引不能命中是什么原因?

见43和101.

87、mysql怎么实现分页

```
#good_id为索引，pageNo为页号
select * from table where good_id > (pageNo-1)*pageSize limit pageSize;
```

88、算法题：多线程顺序打印（用Condition辅助）

```
public class C {

    public static void main(String[] args) {
        Data3 data = new Data3();

        new Thread(()->{
```

```
            for (int i = 0; i <10 ; i++) {
                data.printA();
            }
        },"A").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                data.printB();
            }
        },"B").start();

        new Thread(()->{
            for (int i = 0; i <10 ; i++) {
                data.printC();
            }
        },"C").start();
    }

}
```

```
class Data3{ // 资源类 Lock

    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();
    private int number = 1; // 1A  2B  3C

    public void printA(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            while (number!=1){
                // 等待
                condition1.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>AAAAAAA");
            // 唤醒，唤醒指定的人，B
            number = 2;
            condition2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printB(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            while (number!=2){
                condition2.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>BBBBBBBB");
            // 唤醒，唤醒指定的人，C
            number = 3;
            condition3.signal();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void printC(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            // 业务，判断-> 执行-> 通知
            while (number!=3){
                condition3.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>BBBBBBBBB");
            // 唤醒，唤醒指定的人，c
            number = 1;
            condition1.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}
```

89、一个线程要等待100个线程执行完之后再执行怎么办。是否可以直接使用count来控制?

```java
package com.kuang.add;

import java.util.concurrent.CountDownLatch;

// 计数器
public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        // 总数是6，必须要执行任务的时候，再使用！
        CountDownLatch countDownLatch = new CountDownLatch(6);

        for (int i = 1; i <=6 ; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+" Go out");
                countDownLatch.countDown(); // 数量-1
            },String.valueOf(i)).start();
        }

        countDownLatch.await(); // 等待计数器归零，然后再向下执行

        System.out.println("Close Door");

    }
}
```

原理：

不可以直接用count配合volatile来控制，不能保证原子性，最后可能不到100个线程执行。

90、手动垃圾回收怎么实现。了解finalize()么？是什么类的方法？

System.gc();

finalize()方法是Object类中提供的一个方法，在GC准备释放对象所占用的内存空间之前，它将首先调用finalize()方法。

91、拦截器和过滤器的区别？过滤器只能有一个么？

92、反射破坏了封装性如何避免？反射可以获取到私有的构造方法么？

反射可以获取到私有的构造方法。

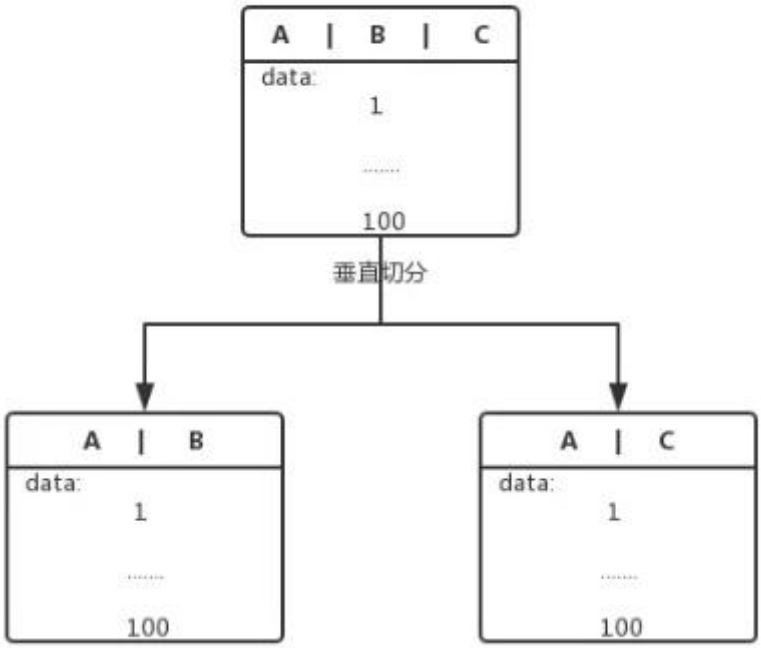
```java
Constructor<Person> con = Person.class.getDeclaredConstructor(null);
//暴力访问,值为true则指示反射的对象在使用时应该取消Java语言访问检查
con.setAccessible(true);
Object obj = con.newInstance("张三");

System.out.println(obj);
```

93、分库分表，可以怎么分?

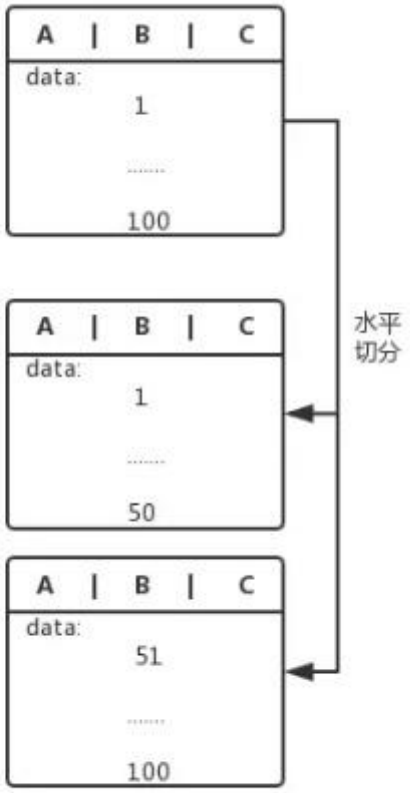
垂直切分：

根据业务的不同，将原先拥有很多字段的表拆分为两个或者多个表，这样的代价我个人觉得很大，原来对这应这个表的关系，开始细分，需要一定的重构，而且随着数据量的增多，极有可能还要增加水平切分；



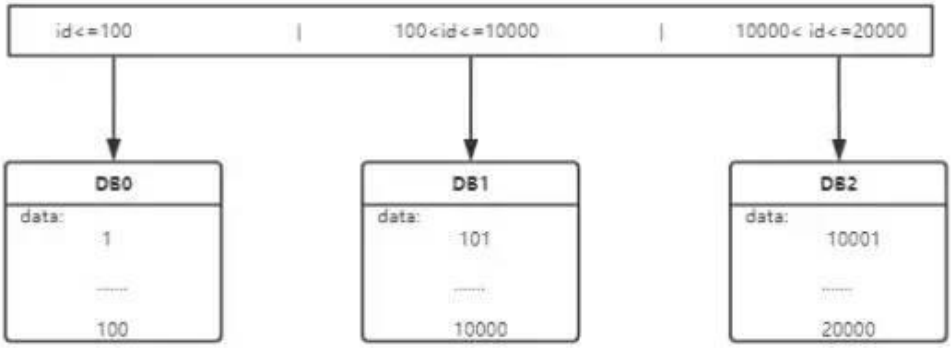
水平切分：

数据表结构，将数据分散在多个表中；



1.有瑕疵的简单分库分表（按id的大小分库分表）

按照分片字段（我们这里就用id表示了）的大小来进行分库分表，如果你的id是自增的，而且能保证在进行分库分表后也是自增的，那么能进行很好的改造，以id大小水平切分，而且极有可能不用迁移数据。

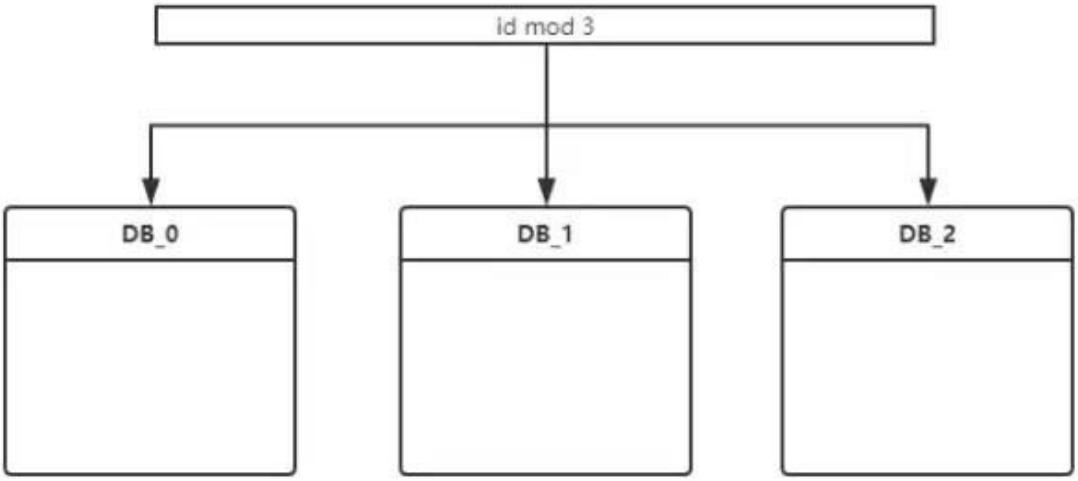


当然，这里只列举了比较小的数据量，实际情况的分库的界限还是要依据具体的情况而定。这样的分库分表，因为新的数据总在一个库里，很可能导致热点过于集中（读写可能集中在一个库中），这是采取这种方式需要考虑的事情。如果无法保证你的id是自增长的，那么你的数据就会凌乱的分散在各个数据库，这样热点确实就分散了，可是每当你增加一个数据库的时候，你就有可能进行大量的数据迁移，应对这种情况，就是尽量减少数据迁移的代价，所以这里运用一致性hash的方式进行分库分表比较好，可以尽可能的减少数据迁移，并且也能让解决热点过于集中的问题。一致性hash的分库策略去百度一下或者谷歌一下应该很容易搜到。这里按id的大小来分库，还可以发散到按照时间来分库，比如说一个月的数据放在一个库，这个使用mycat比较容易实现按时间分库，不过你需要思考的数据的离散性，数据集

中于一个两月，而剩下的几个月数据稀疏，这样的又可能需要按照数据的生产规律合并几个月到一个库中，使得数据分布均匀。

2.比较方便的取模分库

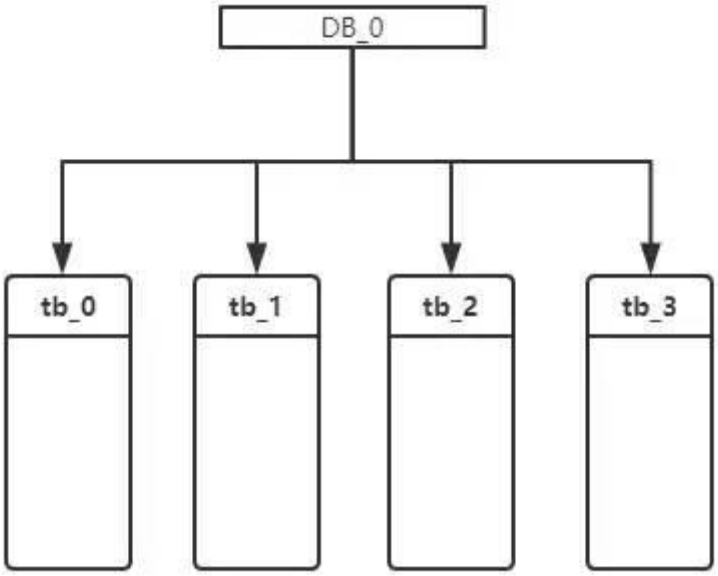
一般的取模分库分表是就是将id mod n，然后放入数据库中，这样能够使数据分散，不会有热点的问题，那么，剩下的是，在扩容的时候，是否会有数据迁移的问题，一般的扩容，当然是会有数据迁移的。



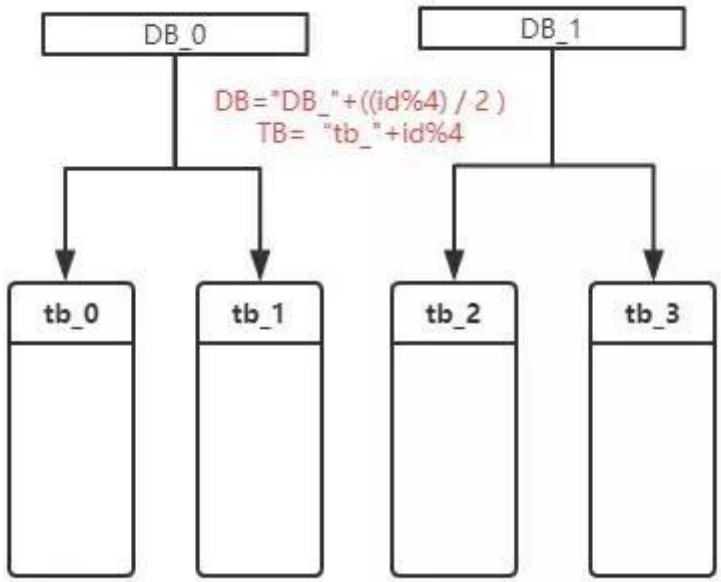
例子中，对3取模，当需要扩容的时候(假设增加两个库)，则对5取模，这样的结果必然是要进行数据迁移的，但是可以运用一些方法，让它不进行数据迁移，scale-out扩展方案能够避免在取模扩容的时候进行数据迁移。

(1)第一种扩容的方式：根据表的数据增加库的数量

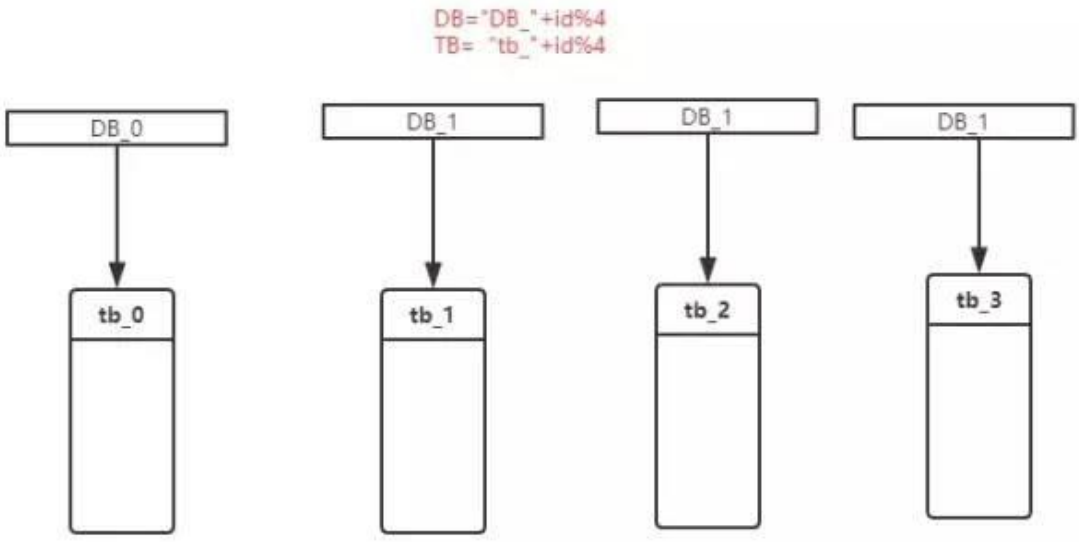
首先，我们有一个数据库——DB_0,四张表——tb_0,tb_1,tb_2,tb_3那么我们现在数据到数据库是这样的:DB="DB_0"TB="tb_"+id%4



当数据增加，需要进行扩容的时候，我增加一个数据库DB_1DB="DB"+*((id%4)/2)TB=“tb*"+id%4



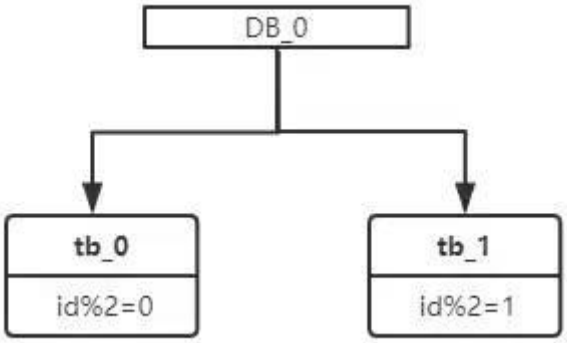
当我们的数据继续飙升，这个时候又需要我们增加库，这个时候进行加库操作的时候，就不是增加一个库，而必须是两个，这样才能保证不进行数据迁移。DB="DB"+*id%4TB=“tb*"+id%4



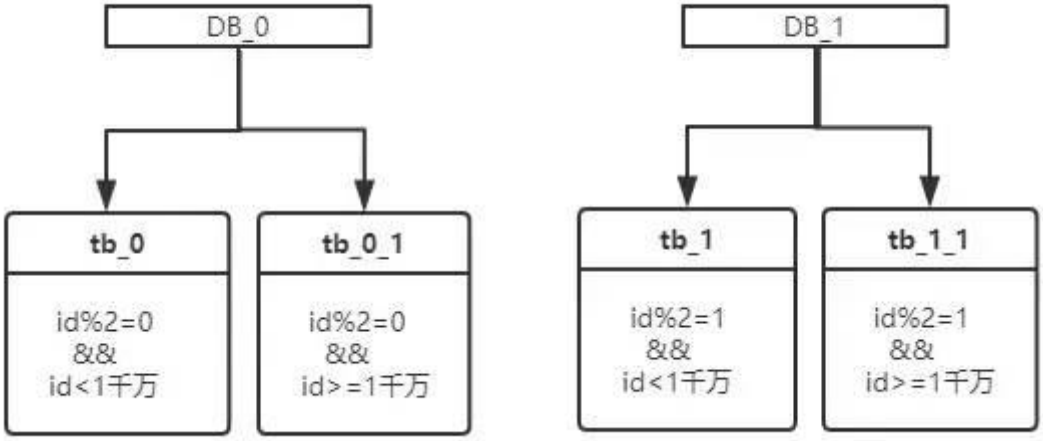
这个时候到了这个方案的加库上限，不能继续加库了，否则就要进行数据迁移，所以这个方案的弊端还是挺大了，这样的方式，也应该会造成单表的数据量挺大的。

(2)第二种扩容的方式：成倍的增加表

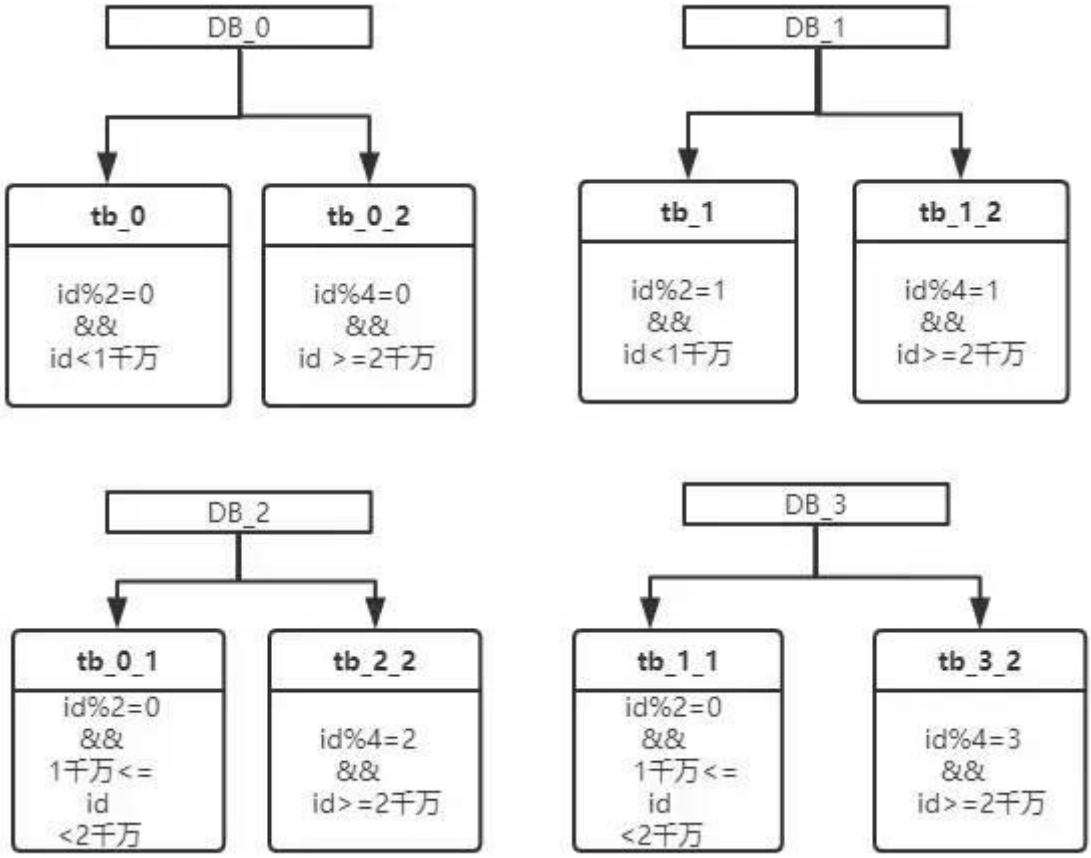
首先，我们还是一个数据库——DB_0,两张表——tb_0,tb_1那么我们现在数据到数据库是这样的:DB="DB_0"TB=“tb_"+id%2



假设当我们数据量打到一千万的时候，我们增加一个库，这时需要我们增加两张表tb_0_1,tb_1_1,并且原来的DB_0中库的表tb_1整表迁移到DB_1中，tb_0和tb_0_1放在DB_0中，tb_1和tb_1_1放到DB1中。DB="DB"+*id%2tb:if(id<1千万) { return "tb*" + id % 2 }else if(id>=1千万) { return "tb"+ *id % 2* + "1" }



数据的增长不可能到此为止，当增加到两千万的时候，我们需要加库，这个时候，按照这种做法，我们需要增加两个库(DB_2,DB_3)和四张表(tb_0_2,tb_1_2,tb_2_2,tb_3_2)，将上次新增的表整表分别放进两个新的库中，然后每个库里再生成一张新表。DB:if(id < 1千万) { return "DB" + *id % 2 }else if(1千万 <= id < 2千万) { return "DB*"+ id % 2 +2 }else if(2千万 <= id) { return "DB"+ *id % 4 }tb:if(id < 1千万) { return "tb*" + id % 2 }else if(1千万 <= id < 2千万) { return "tb"+ *id % 2 +"1" }else if(id >= 2千万) { return "tb"+ id % 4+"*2" }



值得注意的一点，在id超出范围的时候，该给怎么样的提示是值得思考的。

94、mysql A有10行记录，B有5行记录，请问 A left join B最后输出多少行?

最少10行。

95、手撕KMP

```
void Getnext(int next[],String t)
{
   int j=0,k=-1;
   next[0]=-1;
   while(j<t.length-1)
   {
      if(k == -1 || t[j] == t[k])
      {
         j++;k++;
         next[j] = k;
      }
      else k = next[k];
       /*这句回退语句是重点。
       如果t[j]!=t[k]，那么next[j] = h<k，且有t[j] = t[h]。
       又有t[j-1] = t[h-1] = t[k-1]，根据next数组的定义就有next[k]=h，即next[j] =
next[k]。
       所以k回退就设置为next[k]。
       */
```

```
    }
}

int KMP(String s,String t)
{
    int next[MaxSize],i=0;j=0;
    Getnext(t,next);
    while(i<s.length&&j<t.length)
    {
        if(j==-1 || s[i]==t[j])
        {
            i++;
            j++;
        }
        else j=next[j];                 //j回退。。。
    }
    if(j>=t.length)
         return (i-t.length);           //匹配成功，返回子串的位置
    else
        return (-1);                    //没找到
}

//改进后的next数组计算
void Getnext(int next[],String t)
{
    int j=0,k=-1;
    next[0]=-1;
    while(j<t.length-1)
    {
        if(k == -1 || t[j] == t[k])
        {
            j++;k++;
            if(t[j]==t[k])//当两个字符相同时，就跳过
                next[j] = next[k]; //next[j] = next[next[j]]=next[k];
            else
                next[j] = k;
        }
        else k = next[k];
    }
}
```

96、Redis 的分布式锁。

一般的sychronized加锁只能对同一个JVM中的进程下加锁，但是如果一个WEB项目是部署在不同的机器上，对于不同的机器，实际上是不同的JVM了，那么sychronized加锁就不起作用了，需要用分布式锁来满足开发需求。

通过对可能并发执行的代码加入分布式锁，利用redis中的setnx命令来加分布式锁，首先判断分布式锁是否有被加锁，没的话，这个进程可以执行接下来的代码，否则不可以执行。最终业务代码执行完毕以后必须要删除这个setnx锁。

```
Setnx lock true  ->  加分布式锁
Setnx lock false  -> 已经加了分布式锁，不能再加了
Get lock -> 显示lock状态为true

........ 执行代码..........

Del lock  ->  删除分布式锁
```

问题：

1、简易分布式锁可能会因为中间的业务代码抛出异常导致分布式锁没有被删除出错，可以用**try finally**解决。

2、但是如果机器重启的话，也会因为锁没有删除出错，**可以在加分布式锁时设置锁的存活时间（比如10s）**，过期会自动释放锁。

3、如果一个线程业务的执行时间比锁的存活时间更长，比如15s，那么这个线程在执行到一半时锁就会被释放，可是业务还没执行完，还需要执行5s。此时如果第二个线程申请加锁也能加成功，并且第一个线程业务结束时释放的锁会是第二个线程加的锁，那么redis锁就失效了，没有了锁的意义。**解决方法：可以将锁的值设置为一个UUID值，每次释放锁的时候判断锁的值是否是这个线程一开始设定的UUID值，如果是的话，代表是这个线程加的分布式锁，可以释放。**

4、如果业务执行的时间（15s）大于锁的存活时间（10s），除了会出现一个线程释放另一个线程的锁的问题（问题3），还会出现多个线程共同执行一部分同步代码的问题。**解决方法：可以在执行业务前设置一个后台分线程，每隔一段时间（3s）判断上述的锁是否还存活，如果存活的话，重新设置存的存活时间为10s。**

97、Redis 的分布式集群。

设置哨兵模式。

1、配置哨兵配置文件 sentinel.conf

```
# sentinel monitor 被监控的名称 host port 1
sentinel monitor myredis 127.0.0.1 6379 1
```

2、启动哨兵

```
 redis-sentinel config/sentinel.conf
```

如果Master 节点断开了，这个时候就会从从机中随机选择一个服务器！ （这里面有一个投票算法！）

如果主机此时回来了，只能归并到新的主机下，当做从机，这就是哨兵模式的规则！

98、同步和异步的理解

同步

所有的操作都做完，才返回给用户。这样用户在线等待的时间太长，给用户一种卡死了的感觉（就是系统迁移中，点击了迁移，界面就不动了，但是程序还在执行，卡死了的感觉）。这种情况下，用户不能关闭界面，如果关闭了，即迁移程序就中断了。

异步
将用户请求放入消息队列，并反馈给用户，系统迁移程序已经启动，你可以关闭浏览器了。然后程序再慢慢地去写入数据库去。这就是异步。但是用户没有卡死的感觉，会告诉你，你的请求系统已经响应了。你可以关闭界面

99、如果多个线程使用 ThreadLocal 的话，会不会有冲突的情况呢?

不会，每一个线程都会有一个工作内存副本。

100、快排的优化方案。

- 哨兵的位置通过三数取中法来确定，而不是只设置为头或者尾，因为最佳的划分是将待排序的序列分成等长的子序列。

  举例：待排序序列为：8 1 4 9 6 3 5 2 7 0

  左边为：8，右边为0，中间为6.

  我们这里取三个数排序后，中间那个数作为枢轴，则枢轴为6

- 当待排序序列的长度分割到一定大小后，使用插入排序
- 在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割

  具体过程：在处理过程中，会有两个步骤

  第一步，在划分过程中，把与key相等元素放入数组的两端

  第二步，划分结束后，把与key相等的元素移到枢轴周围

101、联合索引之最左匹配原则

联合索引是B+数原理，叶子节点中是按索引中第一列排序建立的，后面的列是相对有序的。

最左匹配原则：如果(a,b,c)三列建立索引，实际上会建立(a),(a,b)和(a,b,c)的索引。如果查找时遇到了范围查找（like,>,表达式）会停止后面的查找条件。

联合索引中如果where没有用到第一列会索引失效。

102、给学生表、成绩表。查询平均成绩大于等于 60 分的同学的学生编号和学生姓名和平均成绩

```
SELECT student.s_id, student.s_name,a.avg FROM student
INNER join
(SELECT s_id,avg(s_score) as avg
FROM score GROUP BY s_id
HAVING avg(s_score)>=60) a
on student.s_id=a.s_id
```

103、AOF文件过大怎么办

执行bgrewriteaof命令对AOF文件进行重写，将其中重复的命令或者可以合并的命令合并在一起。

104、hashmap中为什么要重写hashcode和equals方法

Object中的hashcode（）是直接根据对象在内存中的地址算的hashcode，而实际我们可能需要根据一个对象中的成员字段(id或者name)来计算Hashcode。

Objcet中的equals是用==来判断的，如果两个比较的key是引用类型的对象而不是基本数据类型，那么一定不会相等，因为引用数据类型存在堆中，对应的内存地址不一样，所以需要重写euqals方法，根据我们所需要比较的内容来重写，例如根据对象中的id或者name来判断key对象是否一样。

```
HashMap<Person,String> hashmap = new HashMap<>();
Person p1 = new Person("a");
Person p2 = new Person("a");
//如果不重写euqals，那么key会是不同的，因为是不一样的Person对象，但是他们名字一样，我们想让这两key一样。
hashmap.put(p1,"p1");
hashmap.put(p2,"p2");
```

105、http有哪些方法？put和post区别。

get, post, put, head, delete

put方法是上传资源，幂等方法

post方法是修改资源，非幂等方法

106、count(1) 、count(2)、count(*)、count(col名)区别

count(1)中的1指的不是第一个字段，而是每一行数据当成一个1，然后统计所有1的个数，所以count(2)和count(1)的意思一样。

count(*)也是统计所有行的数目，但是相比于count(1)来说，执行时多了一步，会把星号翻译成字段的具体名字。结果和count(1)一样。

count(col名)会统计指定列中**非空行**的数目。

执行效率上：

列名为主键，count(列名)会比count(1)快；

列名不为主键，count(1)会比count(列名)快；

如果表多个列并且没有主键，则 count（1）的执行效率优于 count（*）；

如果有主键，则 select count（主键）的执行效率是最优的

如果表只有一个字段，则 select count（*）最优。

107、迪杰斯特拉算法

```
 public static void dijstra(int[][] matrix, int source) {
        //最短路径长度
        int[] shortest = new int[matrix.length];
        //判断该点的最短路径是否求出
        int[] visited = new int[matrix.length];
        //存储输出路径
        String[] path = new String[matrix.length];

        //初始化输出路径
        for (int i = 0; i < matrix.length; i++) {
            path[i] = new String(source + "->" + i);
        }

        //初始化源节点
        shortest[source] = 0;
        visited[source] = 1;

        for (int i = 1; i < matrix.length; i++) {
            int min = Integer.MAX_VALUE;
            int index = -1;

            for (int j = 0; j < matrix.length; j++) {
                //已经求出最短路径的节点不需要再加入计算并判断加入节点后是否存在更短路径
                if (visited[j] == 0 && matrix[source][j] < min) {
                    min = matrix[source][j];
                    index = j;
                }
            }

            //更新最短路径
            shortest[index] = min;
            visited[index] = 1;

            //更新从index跳到其它节点的较短路径
            for (int m = 0; m < matrix.length; m++) {
                if (visited[m] == 0 && matrix[source][index] + matrix[index][m]
< matrix[source][m]) {
                    matrix[source][m] = matrix[source][index] + matrix[index]
[m];
                    path[m] = path[index] + "->" + m;
                }
            }

        }

    }
```

108、同步和异步的应用场景举例

同步：多线程互斥访问信号量。

异步：回调函数、多线程异步处理任务（互不干扰）

109、多种进程间通信方式的应用场景。

命名管道：用作客户端的汇聚点和服务端交互数据。

共享存储：一般在同一个机器内，共享内存中的一块区域，速度最快，不需要复制到输入/输出缓冲区。

消息队列：任务的异步处理、应用的解耦（例如通过消息队列 订单系统和库存系统可以分离，两个同时订阅消息队列即可，而不用因为库存无法访问导致下单失败）。

套接字：适用于不同机器间的网络通信（聊天室）。

110、包装类哪些是不可变的?

所有的包装类Integer、String、Float、Double都不可变，都被Final修饰了，在对包装类的状态（值）改变时是不可变的。

111、某些字段不想被序列化应该怎么处理?

用transient关键字修饰。

112、main能被重载吗

可以被重载

```
class Test {
    public static void main(String[] args) {
        main(1);
    }
    static void main(int i) {
        System.out.println("重载的main方法！");
    }
}
```

113、String类为什么是final

为了保证字符串为不可变的。

1.为了实现字符串池 。字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。

2.为了线程安全。因为字符串不可变，所以多线程可以共享同一个字符串而不用担心安全问题。

3.为了缓存Hashcode。字符串不可变的特性保证了hashcode永远是相同的。不用每次使用hashcode就需要计算hashcode。这样更有效率。很适合做map和set的key。

114、Class.forName 和 classLoader有什么区别

class.forName()前者除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static修饰的内容。
classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容,只有在实例化后才会去执行static内容。

115、为什么负数比正数多表示1个，比如-128-127?

因为表示第一位是符号位，0为正数，1为负数，而全零表示的是0不是正数，所以占据了正数的其中一个表示。

116、如果代码中if很多怎么办

用策略模式解耦。

假如A、B、C三种情况分别实行三种不同的func()方法。

那么写一个Strategy抽象类，其中只有一个func()方法，然后三个策略类SA, SB, SC实现了这个接口且重写了方法。

然后业务类中有一个成员变量就是Strategy抽象类，且也有一个func()方法，这个是调用了Strategy中的func()方法，当这个业务类传入不同的Strategy类就会执行它们自己的func()方法。

117、泛型擦除

Java 泛型在运行的时候是会进行类型擦除，泛型信息只存在于代码编译阶段。即对于List和List在运行阶段都是List.class,泛型信息被擦除了。

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
assert(l1.getClass() == l2.getClass()); //结果为真
```

```
//指定获得Bean中list属性, public List<String> list;
Field list = Bean.class.getField("list");
// 属性对应的Class如果是List或其子类
if (List.class.isAssignableFrom(list.getType())) {
    //获得 Type
    Type genericType = list.getGenericType();
    //ParameterizedType
    if (genericType instanceof ParameterizedType) {
        //获得泛型类型,如果是map<String,Integer>的话0代表Key,1代表value
        Type type = ((ParameterizedType) genericType).getActualTypeArguments()
[0];
    }
}
```

118、写一个生产者—消费者的实现

```
public class A {
    public static void main(String[] args) {
        Data data = new Data();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"B").start();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"C").start();


        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"D").start();
    }
}
```

```java
// 判断等待，业务，通知
class Data{ // 数字 资源类

    private int number = 0;

    //+1
    public synchronized void increment() throws InterruptedException {
        while (number!=0){  //0
            // 等待
            this.wait();
        }
        number++;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我+1完毕了
        this.notifyAll();
    }

    //-1
    public synchronized void decrement() throws InterruptedException {
        while (number==0){ // 1
            // 等待
            this.wait();
        }
        number--;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我-1完毕了
        this.notifyAll();
    }

}
```

119、Http1.0 1.1区别

1，HTTP/1.0协议使用非持久连接,即在非持久连接下,一个tcp连接只传输一个Web对象,;
2，HTTP/1.1默认使用持久连接。在持久连接下,不必为每个Web对象的传送建立一个新的连接,一个连接中可以传输多个对象!

120、InnoDB中的mvcc

多版本并发控制是**提交读和可重复读**的InnoDB实现。

基础原理：每一行数据都有一系列版本快照，读数据是读最老的版本，更新数据是更新最新的版本，更新数据都会往这个快照链表后写入新的快照。

维护原理：用一个叫ReadView结构维护，是一个未提交的事务列表，按照快照版本号大小串在一起。

在查询时，查询到的数据行快照版本号为id，如果id<min_id，那么该快照为这些未提交事务之前的，可以读，如果id>max_id，那么该快照为这些未提交事务修改之后的，不可读。如果id在min_id和max_id的范围内的话，若是提交读隔离级别，判断这个id在不在未提交的事务列表中，在的话可以读否则不能，若是可重复读级别，都不可以读。

121、InnoDB中怎么解决幻读问题?

用mvcc+ Next-key locks一起。

Record Locks

锁定一个记录上的索引，而不是记录本身。

如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚簇索引，因此 Record Locks 依然可以使用。

Gap Locks

锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就不能在 t.c 中插入 15。

```
SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE;
```

Next-Key Locks

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录上的索引，也锁定索引之间的间隙。它锁定一个前开后闭区间，例如一个索引包含以下值：10, 11, 13, and 20，那么就需要锁定以下区间：

```
(-∞, 10]
(10, 11]
(11, 13]
(13, 20]
(20, +∞)
```

122、hashMap的put过程

1. 求 Key 的 hash 值

想要存入 HashMap 的元素的 key 都必须要经过一个 hash 运算，首先调用 key 对象的 hashCode 方法获取到对象的 hash 值。然后将获得 hash 值向右位移 16 位，接着将位移后的值与原来的值做异或运算，即可获得到一个比较理想的 hash 值，之所以这样运算时为了尽可能降低 hash 碰撞（即使得元素均匀分布到各个槽中）。

2. 确定 hash 槽

获取到 hash 值后就是计算该元素所对应的 hash 槽，这一步操作比较简单。直接将上一步操作中获取到的 hash 值与底层数组长度取模（为了提高性能，源码采用了运算，这样等价于取模运算，但是性能更好）获取 index 位置。

3. 将元素放入槽中

在上一步中我们获得了 hash 槽的索引，接下来就是将元素放入到槽中，如果槽里当前没有任何元素则直接生成 Node 放入槽中，如果槽中已经有元素存在，则会有下面 2 种情况：

3.1 槽里是一颗平衡二叉树

当列表的元素长度超过 8 时，为了加快槽上元素的索引速度，HashMap 会将列表转换为平衡二叉树。所以在这种情况下，插入元素是在一颗平衡二叉树上做操作，查找和更新的时间复杂度都是 log(n)，HashMap 会将元素包装为一个 TreeNode 添加到树上。具体平衡二叉树的操作就不在此展开了，有兴趣的小伙伴可自行探索。

3.2 槽里是一个列表

初始情况下，槽里面的元素是以列表形式存在的，HashMap 遍历列表将元素 更新 / 追加 到列表尾部。元素添加后，HashMap 会判断当前列表元素个数，如达到 8 个元素则将列表转化为平衡二叉树，具体转换详情可参考 HashMap 中的方法 `final void treeifyBin(Node<K,V>[] tab, int hash)` 。

4. 扩容

到这里时候，我们元素已经完美的放到了 HashMap 的存储中。但是还有一个 HashMap 的自动扩容操作需要完成，默认情况下自动扩容因子是 0.75，即当容量为 16 时候，如果存储元素达到 16 * 0.75 = 12 个的时候，HashMap 会进行 resize 操作（俗称 rehash）。HashMap 会新建一个 2 倍容量与旧数组的数组，然后依次遍历旧的数组中的每个元素，将它们转移到新的数组当中。其中有 2 个需要注意的点：列表中元素依然会保持原来的顺序和加入二叉树上的元素少达 6 个时候会将二叉树再次转化为列表来存储。

123、怎么让主线程等待子线程完毕后再继续执行?

1、主线程等待，通过Thread.sleep(1000)

2、子线程通过join（）优先执行。

3、通过callable中的isDone()判断当前任务是否有完成。

124、什么时候用字节流什么时候用字符流?

其实底层都是字节，我们使用字符是为了处理一些文本和字符串提高性能
字符流处理的单元为2个字节的Unicode字符，分别操作字符、字符数组或字符串，而字节流处理单元为1个字节，操作字节和字节数组。所以字符流是由Java虚拟机将字节转化为2个字节的Unicode字符为单位的字符而成的，所以它对多国语言支持性比较好！如果是音频文件、图片、歌曲，就用字节流好点，如果是关系到中文（文本）的，用字符流好点

125、String 类的常用方法都有那些?

- indexOf()：返回指定字符的索引。
- charAt()：返回指定索引处的字符。
- replace()：字符串替换。
- trim()：去除字符串两端空白。
- split()：分割字符串，返回一个分割后的字符串数组。
- getBytes()：返回字符串的 byte 类型数组。
- length()：返回字符串长度。
- toLowerCase()：将字符串转成小写字母。
- toUpperCase()：将字符串转成大写字符。
- substring()：截取字符串。
- equals()：字符串比较。

126、缓存穿透、缓存击穿、缓存雪崩区别和解决方案

缓存穿透

描述：

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

缓存击穿

描述：

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

解决方案：

设置热点数据永远不过期。
加互斥锁，互斥锁参考代码如下：

```
10    public static String getData(String key) throws InterruptedException
11    {
12        //从缓存读取数据
13        String result = getDataFromRedis(key);
14        //缓存中不存在数据
15        if (result == null)
16        {
17            //去获取锁，获取成功，去数据库取数据
18            if (reenLock.tryLock())
19            {
20                //从数据获取数据
21                result = getDataFromMysql(key);
22                //更新缓存数据
23                if (result != null)
24                {
25                    setDataToCache(key,result);
26                }
27                //释放锁
28                reenLock.unlock();
29            }
30            //获取锁失败
31            else
32            {
33                //暂停100ms再重新去获取数据
34                Thread.sleep(100);
35                result =  getData(key);
36            }
37        }
38        return result;
39    }
```


说明：

1）缓存中有数据，直接走上述代码13行后就返回结果了

2）缓存中没有数据，第1个进入的线程，获取锁并从数据库去取数据，没释放锁之前，其他并行进入的线程会等待100ms，再重新去缓存取数据。这样就防止都去数据库重复取数据，重复往缓存中更新数据情况出现。

3）当然这是简化处理，理论上如果能根据key值加锁就更好了，就是线程A从数据库取key1的数据并不妨碍线程B取key2的数据，上面代码明显做不到这点。

缓存雪崩

描述：

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。
设置热点数据永远不过期。