61. 数据库sql关键字执行顺序？

from--->.where,---->group by--->,having,--->select,--->distinct---->,union----->,order by

62.守护线程和僵尸线程

```
任何线程都可以设置为守护线程和用户线程，通过方法Thread.setDaemon(bool on) 设置，true则是将该线程设置为守护线程，false则是将该线程设置为用户线程。
同时，Thread.setDaemon()必须在Thread.start()之前调用，否则运行时会抛出异常。

用户线程：平时使用到的线程均为用户线程。
守护线程：用来服务用户线程的线程，例如垃圾回收线程。

用户线程：当任何一个用户线程未结束，Java虚拟机是不会结束的。
守护线程：如何只剩守护线程未结束，Java虚拟机结束。
```

63.redis的数据结构底层？

string：

我们知道Redis是由C语言编写的。Redis由于各种原因，并没有直接使用了C语言的字符串结构，而是对其做了一些封装，

得到了自己的简单动态字符串(simple dynamic string, SDS)的抽象类型。Redis中，默认以SDS作为自己的字符串表示。只有在一些字符串不可能出现变化的地方使用C字符串。

```
    SDS与C字符串的区别
    1、常数复杂度获取字符串长度
        而SDS结构中本身就有记录字符串长度的len属性，所有复杂度为O(1)。
        Redis将获取字符串长度所需的复杂度从O(N)降到了O(1)，确保获取字符串长度的工作不会成为Redis的性能瓶颈
    2、杜绝缓冲区溢出，减少修改字符串时带来的内存重分配次数
        SDS实现了空间预分配和惰性空间释放两种优化的空间分配策略，解决了字符串拼接和截取的空间问题**，修改字符串长度N次最多会需要执行N次内存重分配

hash：
    ziplist+hashtable
    这两种数据结构我们之前都有讲解，hash对象只有同时满足以下条件，才会采用ziplist编码：
       1 hash对象保存的键和值字符串长度都小于64字节
       2 hash对象保存的键值对数量小于512 ziplist存储的结构如下

list：
    压缩列表ziplist+双向链表linkedlist
        ziplist 是一个特殊的双向链表
        特殊之处在于：没有维护双向指针:prev next；而是存储上一个 entry的长度和 当前entry的长度，通过长度推算下一个元素在什么地方。
        牺牲读取的性能，获得高效的存储空间，因为(简短字符串的情况)存储指针比存储entry长度 更费内存。这是典型的“时间换空间”。

set：
    Set底层用两种数据结构存储，一个是hashtable，一个是inset。
    其中hashtable的key为set中元素的值，而value为null

    inset为可以理解为数组，使用inset数据结构需要满足下述两个条件：
        元素个数不少于默认值512
        set-max-inset-entries 512
        元素可以用整型表示
```

```
    inset结构体定义如下：
        typedef struct intset {
            uint32_t encoding;  // 编码方式，一个元素所需要的内存大小
            uint32_t length;    // 集合长度
            int8_t contents[];  // 集合数组，整数集合的每个元素在数组中按值的大小从小到大排序，且不包含重复项
        } intset;
    inset查询方式一般采用二分查找法，实际查询复杂度也就在log(n)，你会发现到redis是在数据量少的情况下才会用到这个数据结构，
        插入删除O(n)在数据量少的情况下移动数据可以用到CPU向量化执行的特性，特别快，在数据量大的话性能退化选用其他数据结构

zset：ziplist+跳表
```

64.线程创建方式？

1.继承Thread类创建线程，首先继承Thread类，重写run()方法，在main()函数中调用子类实实例的start()方法。

```
public class ThreadDemo extends Thread {
  @Override
  public void run() {
    System.out.println(Thread.currentThread().getName() + " run()方法正在执行");
  }
}
public class TheadTest {
  public static void main(String[] args) {
    ThreadDemo threadDemo = new ThreadDemo();
    threadDemo.start();
    System.out.println(Thread.currentThread().getName() + " main()方法执行结束");
  }
}
```

2.实现Runnable接口创建线程：首先创建实现Runnable接口的类RunnableDemo，重写run()方法；

创建类RunnableDemo的实例对象runnableDemo，以runnableDemo作为参数创建Thread对象，调用Thread对象的start()方法。

```
public class RunnableDemo implements Runnable {
  @Override
  public void run() {
    System.out.println(Thread.currentThread().getName() + " run()方法执行中");
  }
}
public class RunnableTest {
  public static void main(String[] args) {
    RunnableDemo  runnableDemo = new RunnableDemo ();
    Thread thread = new Thread(runnableDemo);
    thread.start();
    System.out.println(Thread.currentThread().getName() + " main()方法执行完成");
}
```

3.使用Callable和Future创建线程：

1. 创建Callable接口的实现类CallableDemo，重写call()方法。
2. 以类CallableDemo的实例化对象作为参数创建FutureTask对象。
3. 以FutureTask对象作为参数创建Thread对象。
4. 调用Thread对象的start()方法。

```
class CallableDemo implements Callable {
    @Override
    public Integer call() {
        System.out.println(Thread.currentThread().getName() + " call()方法执行中");
        return 0;
    }
}
 class CallableTest {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        FutureTask futureTask = new FutureTask(new CallableDemo());
        Thread thread = new Thread(futureTask);
        thread.start();
        System.out.println("返回结果 " + futureTask.get());
        System.out.println(Thread.currentThread().getName() + " main()方法执行完成");
    }
}
```

4.使用线程池例如用Executor框架： Executors可提供四种线程池.

```
class ThreadDemo extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "正在执行");
    }
}
class TestFixedThreadPool {
    public static void main(String[] args) {
        //创建一个可重用固定线程数的线程池
        ExecutorService pool = Executors.newFixedThreadPool(2);
        //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口
        Thread t1 = new ThreadDemo();
        Thread t2 = new ThreadDemo();
        //将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        //关闭线程池
        pool.shutdown();
    }
}
```

65.同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。

同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

66.重量级锁的底部Monitor实现原理？

Monitor数据结构的 *owner*、WaitSet和*EntryList 字段比较重要，它们之间的转换关系如下图*

```
_owner       = NULL; //指向持有ObjectMonitor对象的线程地址
_count       = 0; //锁的计数器，获取锁时count数值加1，释放锁时count值减1，直到
_WaitSet     = NULL; //处于wait状态的线程，会被加入到WaitSet
_EntryList   = NULL ; //处于等待锁block状态的线程，会被加入到该列表
_recursions  = 0; //锁的重入次数
```

从上图可以总结获取Monitor和释放Monitor的流程如下：
1当多个线程同时访问同步代码块时，首先会进入到EntryList中，然后通过CAS的方式尝试将Monitor中的owner字段设置为当前线程，
　　同时count加1，若发现之前的owner的值就是指向当前线程的，recursions也需要加1。如果CAS尝试获取锁失败，则进入到EntryList中。
2当获取锁的线程调用wait()方法，则会将owner设置为null，同时count减1，recursions减1，当前线程加入到WaitSet中，等待被唤醒。
3当前线程执行完同步代码块时，则会释放锁，count减1，recursions减1。当recursions的值为0时，说明线程已经释放了锁。

之前提到过一个常见面试题，为什么wait()、notify()等方法要在同步方法或同步代码块中来执行呢，
　　是因为wait()、notify()方法需要借助ObjectMonitor对象内部方法来完成。

67.synchronized关键字的底层原理？
同步代码块:
是由monitorenter 和 monitorexit 指令完成的，其中monitorenter指令所在的位置是同步代码块开始的位置，
第一个monitorexit 指令是用于正常结束同步代码块的指令，第二个monitorexit 指令是用于异常结束时所执行的释放Monitor指令。
同步方法原理:
没有monitorenter 和 monitorexit 这两个指令了，而在查看该方法的class文件的结构信息时发现了Access flags后边的synchronized标识，
该标识表明了该方法是一个同步方法。Java虚拟机通过该标识可以来辨别一个方法是否为同步方法，如果有该标识，线程将持有Monitor，在执行方法，最后释放Monitor。

总结：Java虚拟机是通过进入和退出Monitor对象来实现代码块同步和方法同步的，代码块同步使用的是monitorenter 和 monitorexit 指令实现的，
　　而方法同步是通过Access flags后面的标识来确定该方法是否为同步方法。

68.ConCurrentHashMap的put()方法？
JDK1.7中的put()方法：
先计算出key的hash值，利用hash值对segment数组取余找到对应的segment对象。
尝试获取锁，失败则自旋直至成功，获取到锁，通过计算的hash值对hashentry数组进行取余，找到对应的entry对象。
遍历链表，查找对应的key值，如果找到则将旧的value直接覆盖，如果没有找到，则添加到链表中。（JDK1.7是插入到链表头部，JDK1.8是插入到链表尾部，这里可以思考一下为什么这样）
JDK1.8中的put()方法:

计算key值的hash值，找到对应的Node，如果当前位置为空则可以直接写入数据。
利用CAS尝试写入，如果失败则自旋直至成功，如果都不满足，则利用synchronized锁写入数据。

69.ConcurrentHashMap迭代器是强一致性还是弱一致性？
与HashMap不同的是，ConcurrentHashMap迭代器是弱一致性。

这里解释一下弱一致性是什么意思，当ConcurrentHashMap的迭代器创建后，会遍历哈希表中的元素，在遍历的过程中，哈希表中的元素可能发生变化，

如果这部分变化发生在已经遍历过的地方，迭代器则不会反映出来，如果这部分变化发生在未遍历过的地方，迭代器则会反映出来。

换种说法就是put()方法将一个元素加入到底层数据结构后，get()可能在某段时间内还看不到这个元素。

这样的设计主要是为ConcurrenthashMap的性能考虑，如果想做到强一致性，就要到处加锁，性能会下降很多。

所以ConcurrentHashMap是支持在迭代过程中，向map中添加元素的，而HashMap这样操作则会抛出异常。

70.ConCurrentHashMap 的key，value是否可以为null?

不能。ConCurrentHashMap中的key和value为null会出现空指针异常，而HashMap中的key和value值是可以为null的。

原因如下：ConCurrentHashMap是在多线程场景下使用的，如果ConcurrentHashMap.get(key)的值为null，那么无法判断到底是key对应的value的值为null还是不存在对应的key值。

而在单线程场景下的HashMap中，可以使用containsKey(key)来判断到底是不存在这个key还是key对应的value的值为null。

在多线程的情况下使用containsKey(key)来做这个判断是存在问题的，因为在containsKey(key)和ConcurrentHashMap.get(key)两次调用的过程中，key的值可能已经发生了改变。

即你一开始get方法获取到null之后，再去调用containsKey方法，没法确保get方法和containsKey方法之间，没有别的线程来捣乱，刚好把你要查询的对象设置了进去或者删除掉了。