**进程线程区别:**

**进程**：是程序的一次执行过程，是系统运行程序的基本单位，系统运行一个程序即是一个进程从创建，运行到消⽅的过程。

在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

**线程：**是一个比进程更小的执行单位。一个进程可以有多个线程，多个线程共享进程的**堆**和**方法区**，但是每个线程有自己的**程序计数器、虚拟机栈**和**本地方法栈**。

**Java内存区域**：

堆和方法区是所有线程共享，**堆**是用于存放新创建的对象 ，**方法区**主要用于存放巳被加载的类信息、常量、静态变量等。

**程序计数器：**1、字节码解释器通过改变程序计数器来读取指令，从而实现代码的流程控制。2、在多线程的情况下，程序计数器用于记录当前线程执行的位置。

**虚拟机栈：**每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用。方法调用就对应着一个栈帧在Java 虚拟机栈中入栈和出栈的过程。

**本地方法栈：**虚拟机栈为虚拟机执行 Java方法服务，本地方法栈是为虚拟机使用到的 Native 本地方法服务。

sleep方法没有释放锁，而wait方法释放了锁。Wait通常被用于线程间通信，sleep通常被用于暂停执行。

**锁升级：**

JDK1.6对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁等来减少锁操作的开销。

**偏向锁**：在没有多线程竞争的时候，为了减少传统的重量级锁使用操作系统互斥量产生的性能消耗。偏向锁在无竞争的情况下会****把整个同步都消除掉****。

**轻量级锁：**若偏向锁失败，也一样减少互斥量，轻量级锁的加锁和解锁都****用到了CAS操作。****

****自旋锁：****轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。因为挂起线程/恢复线程的操作都需要用户态转入内核态中完成。****让一个线程等待，让线程执行一个忙循环，就是自旋。****

****自适应的自旋锁：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定。****

**synchronized **依赖于 JVM 而 ReenTrantLock 依赖于 API。****ReenTrantLock增加了一些高级功能。****①等待可中断；②可实现公平锁；③锁可以绑定多个条件。（synchronized只能是非公平锁）****

**Java内存模型：**

线程可以把主内存中的变量保存每个线程的本地内存中进行读取和写入。

**volatile**，关键字修饰变量，每次使用它都到主存中进行读取。volatile 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

并发编程的三个重要特性：**原子性，可见性，有序性**。

一般创建的变量是可以被任何一个线程访问并修改的。**ThreadLocal**可以实现每一个线程都有自己的专属本地变量。就是让每个线程绑定自己的值，如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的本地副本，可以使用 get和set方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

**原子类：**主要利用CAS + volatile 和native方法来保证原子操作，从而避免 synchronized 的高开销。CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。

**AQS：**

一个用来构建锁和同步器的框架，比如ReentrantLock，Semaphore。**1、如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是队列锁实现的，即将暂时获取不到锁的线程加入到队列中。**AQS使用一个int成员变量state来表示同步状态，通过队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作。

**两种资源共享方式：**

**Exclusive(独占)：**只有一个线程能执行，如ReentrantLock分为公平锁和非公平锁。

**Share(共享锁）**：多个线程可同时执行，如Semaphore、CountDownLatch、 CyclicBarrier。

**Semaphore(信号量)**-允许多个线程同时访问，Semaphore(信可以指定多个线程同时访问某个资源。值被获取到后是可以释放，可以被用来**限制流量**。****

**CountDownLatch (倒计时器）**： CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，直到其他线程执行的操作完成，再开始执行。典型的应用如并行计算，等待所有的子任务都完成之后，父任务再拿到所有子任务的运算结果汇总。

**CyclicBarrier(循环柄栏)：** CyclicBarrier和CountDownLatch非常类似，它也可以实现线程间的等待，但是它的功能比 CountDownLatch 更加复杂和强大。是可循环使用。让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，才会继续千活。每个线程调用await()方法告诉 CyclicBarrier 我巳经到达了屏障，然后当前线程被阻塞。通过它可以完成多个线程之间相互等待，只有当每个线程都准备就绪后，才能各自继续往下执行后面的操作。

**Java对象的创建过程：**

1、**类加载检查**： 虚拟机遇到一条new指令时，首先将去检查这个类是否巳被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

2、类加载检查通过后，接下来虚拟机将为新生对象**分配内存**。

3、**初始化零值**： 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（保证了对象在 Java 代码中可以不赋初始值就直接使用）。

4、**设置对象头**： 初始化零值完成之后，虚拟机要对对象进行必要的设置，比如这个对象是那个类的实例、对象的哈希吗、对象的 GC 分代年龄等信息。这些信息存放在对象头中。

5、**执行init方法**： 完成以后才开始执行<init>方法。把对象按照代码里的设置进行初始化，这样对象才创建成功。

**如何判断对象是否死亡：**

**引⽤计数法**：给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1传当引用失效，计数器就减1；计数器为0的对象就是不可能再被使用的。

**可达性分析算法：**通过 “GC Roots”对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。

**垃圾回收：**

**Serial收集器：**新生代采用复制算法，老年代采用标记-整理算法，单线程收集器。

**ParNew收集器**：新生代采用复制算法，老年代采用标记-整理算法，多线程版本。

**Parallel Scavenge收集器：**新生代采用复制算法，老年代采用标记-整理算法。关注的是吞吐量(高效率的利用CPU)。

#### **CMS收集器：**关注点更多的是用户线程的停顿时间，使用的是标记-清除算法。

初始标记（暂停线程，记录可达性）、并发标记（开启GC和用户线程，记录更新）、重新标记（记录并发标记阶段变更，stw）、并发清除。

**类加载过程：**

加载、验证、准备、解析、初始化、其中准备、验证、解析3个部分统称为连接（Linking）。 、**加载：** 1. 通过一个类的全限定名来获取定义此类的二进制字节流；2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构； 3. 在内存中生成一个代表这个类的Class对象，作为方法区这个类的各种数据的访问入口；

**验证：**确保Class文件的字节流中包含的信息符合当前虚拟机的要求，

1. 验证字节流是否符合Class文件格式的规范；例如：是否以魔术开头、主次版本号是否在当前虚拟机的处理范围之内。 2、字节码验证：通过数据流分析，确定程序语义是合法的。

**准备阶段：**正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量）

**解析阶段：**虚拟机将常量池内的符号引用替换为直接引用。

**类初始化阶段：**才真正开始执行类中定义的java程序代码。在初始化阶段，则根据程序员通过程序设计去初始化类变量和其他资源，就是执行类构造器<clinit>()方法的过程.<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的。

**计算机网络：5应⽤层：**（DNS域名系统：将域名和IP地址相互映射，更方便访问）、（HTTP协议：超文本传输协议，是互联网上应用最为广泛的一种网络协议。所有的万维网文件都必须遵守这个标准。提供一种发布和接收 HTML 页面的方法。

**4运输层：（提供数据传输服务）**1.传输控制协议**TCP(字节流)**提供面向连接的，可靠的数据传输服务。2、用户数据协议 **UDP（数据报文段）**提供无连接的，(不保证数据传输的可靠性）。

**3网络层：IP协议，**选择合适的网络路由连接，确保数据及时传送**。面向无连接**

**2数据链路层、1物理层**

**TCP保证可靠：**1.应用数据被**分割成** TCP认为最适合发送的**数据块**。2.TCP 给发送的每一个包**进行编号**，接收方对数据包进行排序，把有序数据传送给应用层。3**.校验和**：TCP 将保持它首部和数据的校验和。这是一个端到端的校验和，目的是检测数据在传输过程中的任何变化。如果收到段的校验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。4.TCP的接收端会**丢弃重复的数据。**5.**流量控制：**TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。(**TCP利用滑动窗口实现流量控制**）

**6.拥塞控制：**当网络拥塞时，减少数据的发送。**7.ARQ协议：**为了实现可靠传输的，它的原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。

**8.超时重传：**当 TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

**Url访问过程：1.DNS解析（**递归查询，浏览器缓存->本地域名服务器->根域名->顶级域名**）2.TCP连接3.发送HTTP请求(** 请求行, 请求报头和请求正文**)4.服务器处理请求并返回HTTP报文5.浏览器解析渲染页面6.连接结束**

**三次握手：**建立可靠的通信信道，就是双方确认自己与对方的发送与接收是正常的。

1. 客户端–发送带有 SYN 标志2、服务端–发送带有 SYN/ACK 标志的数据包3、客户端–发送带有带有 ACK 标志的数据包。

**四次挥手：1、**客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送2、服务器-收到这个 FIN，它发回一个ACK，确认序号为收到的序号加1 。3、然后等待数据传输完，发送一个FIN给客户端4、客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加1。

**2ms：**服务器已经发送了FIN+ACK报文请求断开了，客户端还没有给回应，发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个2MSL时间段内收到这个重传的报文，接着给出回应报文，并且会重启2MSL计时器。防止新的连接中出现旧连接的请求报文

**红黑树：**1.每个节点非红即黑；2.根节点总是黑色的；3.每个叶子节点都是黑色的空节点；4.如果节点是红色的，则它的子节点必须是黑色；5.从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点(即相同的黑色高度）。

TreeMap、TreeSet以及JDK1.8的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

B树是一种平衡的多路查找(又称排序）树，主要用作文件的索引。B+树的叶子节点之间是链表结构相比于 B树便于扫库，和范围索引。B+树支持(区间查询）非常方便。这是数据库选用B+树的最主要原因。1、单一节点存储更多的元素，使得查询的IO次数更少。2、所有查询都要查找到叶子节点，查询性能稳定。3、所有叶子节点形成有序链表，便于范围查询。1、**b树依靠的是中序遍历**。2、**b+树只需要在链表上做遍历**

**进程间通信：**

1. **管道/匿名管道**：****1.具有亲缘关系的父子进程间或者兄弟进程之间的通信。双方通信时，需要建立起两个管道。数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
2. **信号量**：****是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。
3. **套接字 :**可以用于在客户端和服务器之间通过网络进行通信。支持 TCP/IP 的网络通信的基本操作单元，不同主机之间的进程进行双向通信，用套接字中的相关函数来完成通信过程。****

****进程间同步：1、互斥量****，只有拥有互斥对象的线程才有访问公共资源的权限。****2、信号量2.允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量，****

**索引：**索引是一种特殊的文件，它包含着数据表里所有记录的引用指。数据库索引就像是一本书前面的目录，能加快数据库的查询速度

**InnoDB:数据文件本身就是索引文件**。而MyISAM，索引文件和数据文件是分离的，其**数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录**。**索引的key就是数据表的主键**，因此InnoDB表数据文件本身就是主索引。这被称为“聚簇索引”。而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值。然后再根据主索引搜索时，直接找到key所在的节点，取出数据。

**区别：**

1. MyISAM 只有表级锁，而InnoDB 支持行级锁和表级锁,默认为行级锁。

2.支持事务和崩溃后的安全恢复。3. MyISAM支持外键。4.支持MVCC （多版本并发控制）。

**InnoDB**存储引擎在数据库每行数据的后面添加了三个字段

1、**事务ID**(DB\_TRX\_ID)字段: 用来标识最近一次对本行记录做修改的事务id。

2、**回滚指针**(DB\_ROLL\_PTR)字段: 指向写入回滚段的 undo log record (撤销日志记录)。

3、**DB\_ROW\_ID字段**：如果没有自己的主键或者合适的唯一索引，InnoDB会帮我们自动生成聚集索引, 使用DB\_ROW\_ID的值来作为主键; 如果我们有自己的主键或者合适的唯一索引, 那么聚簇索引中也就不会包含 DB\_ROW\_ID了。

**InnoDB实现MVCC的方式是**:

1. 事务以排他锁的形式修改原始数据2、把修改前的数据存放于undo log，通过回滚指针与主数据关联3、修改成功（commit）啥都不做，失败则恢复undo log中的数据（rollback）

**大表优化：限定数据的范围，读写分离，垂直分区，水平分区（**保持数据表结构不变，通过某种策略存储数据分片。把每一片数据分散到不同的表或者库中，达到了分布式的目的**）**

可以使用中间件代理： 在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。

**#{}是预编译处理，$ {}是字符串替换**。mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；mybatis在处理 $ { } 时，就是把 ${ } 替换成变量的值。使用 #{} 可以有效的防止SQL注入，提高系统安全性。

**分布式锁**：分布式环境下，基于本地单机的锁无法控制分布式系统中分开部署客户端的并发行为。在分布式的应用服搭建务器外，一个存储服务器，存储锁信息，使用Redis。

**加锁**使用jedis，*SET lock\_key random\_value NX PX 5000*。基于*setnx(key, value)：“set if not exits”*，**若该key-value不存在，则成功加入缓存并且返回1，否则返回0。**将value关联到key，并将key生成时间设置为seconds，这是一个原子性操作，关联值和生存时间会同一时间完成。**释放锁**的操作必须使用Lua脚本来实现。释放锁其实包含三步操作：GET、判断和DEL，用Lua脚本来实现能保证这三步的原子性。

**Redis主从同步**：使用主从结构一来是备份数据，二来也可以实现读写分离作用，master写数据，salve读数据，**实现高可用**，配置多台 Redis 服务器，一台主服务器和多台从服务器。主服务器进行写操作，从服务器进行读操作。SLAVEOF 指令。

**同步实现过程：**master配置了一个slave，不管这个slave是否是第一次连接上Master，它都会发送一个SYNC命 令给master请求复制数据。

**同步：**将从服务器的数据库状态更新成主服务器当前的数据库状态

1、从服务器向主服务器发送 sync 命令2、收到 sync 命令后，主服务器执行 bgsave 命令，用来生成 rdb 文件，并在一个缓冲区中记录从现在开始执行的写命令。3、bgsave 执行完成后，将生成的 rdb 文件发送给从服务器，用来给从服务器更新数据4、主服务器再将缓冲区记录的写命令发送给从服务器，从服务器执行完这些写命令后，此时的数据库状态便和主服务器一致了。

**命令传播：**当主服务器数据库状态被修改后，导致主从服务器数据库状态不一致，此时需要让主从数据同步到一致的过程。为了再次让主从数据库状态一致，**主服务器就会向从服务器执行命令传播操作** ，即把刚才造成不一致的写命令，发送给从服务器去执行。

哨兵模式（sentinel），如果master宕机了，自动切换，并且还会切换客户端的连接数据源。这就达到高可用。哨兵是特殊的redis服务，**不提供读写服务，主要用来监控redis节点**。 哨兵架构下client端第一次从哨兵找出redis的主节点，后续就直接访问redis的主节点，当redis的主节点发生变化，哨兵会第一时间感知到，并且哨兵会会从主从模式的从节点中重新选出来一个新的master，**并且将新的master信息通知给client端**。这里面redis的client端一般都实现了订阅功能，订阅sentinel发布的节点变动消息。Redis服务是通过配置文件启动的，比如上面的从节点设置了只读模式，它被选举成了master之后就是可读写的了，重新选举之后的各redis服务的配置文件会被哨兵修改。

**redis过期删除机制**：**1、定期删除**：redis默认每隔100ms随机抽取一些设置了过期时间的key，过期就删除。**2、惰性删除:**不管过期的键，当键被取出时，首先检查取出的键是否过期，若过期删除该键，否则，返回该键。两种方式并用，大量内存堆积，采用内存淘汰机制：**redis的LRU机制就会启动，老的数据将会被删除**，我们通过redis的配置文件来设置淘汰策略。1.volatile-lru 使用LRU算法删除一个键(只针对设置了过期时间的key)2.allkeys-lru 使用LRU算法删除一个键3.volatile-lfu 使用LFU算法删除一个键(只针对设置了过期时间的键)4.allkeys-lfu 使用LFU算法删除一个键5.volatile-random 随机删除一个键(只针对设置了过期时间的键)6.allkeys-random 随机删除一个键7.volatile-ttl 删除最早过期的一个键8.noeviction 不删除键，返回错误信息(redis默认选项)

**LRU,最近最少使用算法，较为注重于时间；LFU,最近最不常用算法，较为注重于被访问频率。**

**缓存一致性：对于读请求**（1）先从cache中尝试get数据，结果miss了（2）再从db中读取数据，从库，读写分离（3）最后把数据set回cache，方便下次读命中

**对于写请求，**先操作数据库，再删除缓存（淘汰缓存，而不是更新缓存）更新缓存的话在多线程情况下可能会写入脏数据。

**MySQL主从同步：**当master(主)库的数据发生变化的时候，变化会实时的同步到slave(从)库。

**好处：1、**扩展数据库的负载能力2、容错，高可用。Failover(失败切换)3、数据备份。

**机制：**主服务器和每一个从服务器都必须配置一个唯一的ID号。从服务器还需要通过CHANGE MASTER TO语句来配置它要连接的主服务器的ip地址。

1. 当master上的数据发生改变的时候，该事件(insert、update、delete)变化会按照顺序写入到binlog
2. 当slave连接到master的时候，master机器会**为slave开启binlog dump线程**。当master 的 binlog发生变化的时候，binlog dump线程会通知slave，并将相应的binlog内容发送给slave。
3. 当主从同步开启的时候，slave上会创建2个线程。

**I/O线程**：当连接到master机器，master机器上的binlog dump线程会将binlog的内容发送给该I/O线程。该I/O线程接收到binlog内容后，再将内容写入到本地的relay log。

**SQL线程**：该线程读取I/O线程写入的relay log。并且根据relay log的内容对slave数据库做相应的操作。

**主从同步复制模式：1、**异步复制-主库在执行完客户端提交的事务后会立即将结果返给客户端,并不关心从库是否已经接收并处理。2、全同步复制–当主库执行完一个事务,所有的从库都执行了该事务才返回给客户端。

**分布式事务**：为了保证不同的数据库的数据一致性。**C (一致性)：**对某个指定的客户端来说，读操作能返回最新的写操作。**A (可用性)：**非故障的节点在合理的时间内返回合理的响应。**P (分区容错性)**：当出现网络分区后，系统能够继续工作。

Zookeeper保证CP:因为服务注册有master之分，如果down掉了会有一个选举节点过程，这样可能会有概率无法正常相应。

spring cloud保证AP：Eureka各个节点都是平等的，但是查到的信息可能不是最新的。因为当15分钟内85%节点没有正常心跳，就会认为注册中心出故障，不会移除过期服务，并且暂停同步服务。

二阶段提交（2PC）：**一个事务协调者**：负责协调多个参与者进行事务投票及提交(回滚)

**多个事务参与者**：即本地事务执行者

**处理步骤**（1）投票阶段：协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。参与者将告知协调者自己的决策：同意（事务参与者本地事务执行成功，但未提交）或取消（本地事务执行故障）；（2）提交阶段（commit phase）：收到参与者的通知后，协调者再向参与者**发出通知**，根据反馈情况决定各参与者**是否要提交还是回滚**；

Seata：**TC：** 事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。

TM：控制全局事务的边界，负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。

**RM：**控制分支事务，并接收事务协调器的指令，驱动本地事务的提交和回滚。

**第一阶段：**通过代理数据源对业务Sql的解析，提取到元数据，然后将元数据保存到undolog中，将**业务数据的更新和回滚日志的写入**在同一个本地事务中完成并提交。任何提交的业务数据的更新一定有相应的回滚日志存在。本地事务可以在全局事务的第一阶段提交，并马上释放本地事务锁定的资源

**第二阶段：**如果决议是**全局提交**：此时分支事务已经完成提交，不需要同步协调处理（只需要异步清理回滚日志），可以非常快速地完成。**全局回滚**：RM 收到协调器发来的回滚请求，通过XID（全局事务的唯一标识，由 ip:port:sequence 组成）和Branch ID找到相应的回滚日志记录，通过回滚记录生成反向的更新 SQL 并执行，以完成分支的回滚。

案例：用户下单->订单添加->用户支付->商品库存消减

使用：1、配置文件导入2、创建配置类3、给数据库导入undolog结构表

要想实现全局事务管理器，需要**添加一个@GlobalTransactional注解**,该注解需要创建一个解析器，**GlobalTransactionScanner,它是一个全局事务扫描器**，用来解析带有@GlobalTransactional注解的方法，然后采用AOP的机制控制事务。

每次微服务和微服务之间相互调用,要想控制全局事务，每次TM都会请求TC生成一个XID，每次执行下一个事务，也就是调用其他微服务的时候都需要将该XID传递过去,所以我们可以每次请求的时候，都获取头中的XID，并将XID传递到下一个微服务。

**索引：1、**哈希索引适合等值查询，但是无法进行范围查询 2、哈希索引没办法利用索引完成排序

1. 哈希索引不支持多列联合索引的最左匹配规则 5、如果有大量重复键值的情况下，哈希索引的效率会很低，因为存在哈希碰撞问题

加上索引后查询数据会直接在索引中定位。（注：一般数据库默认都会为主键生成索引）聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

直接创建索引：**CREATE** **INDEX** index\_name **ON** **table**(**column**(length))

修改表结构的方式添加索引：**ALTER** **TABLE** table\_name **ADD** **INDEX** index\_name **ON** (**column**(length))

**联合索引、最左前缀匹配**：在创建多列索引时，我们根据业务需求，where子句中使用最频繁的一列放在最左边，因为MySQL索引查询会遵循最左前缀匹配的原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。所以当我们创建一个联合索引的时候，如(key1,key2,key3)，相当于创建了（key1）、(key1,key2)和(key1,key2,key3)三个索引，这就是最左匹配原则

查看索引是否失效：遇到慢SQL的时候我们才会去排查，可以通过explain查看sql语句的执行计划，通过执行计划来分析索引使用情况。

优化：**1、索引不会包含有NULL值的列2、使用短索引（应该指定一个前缀长度）3、不要在列上进行运算4、不适用索引列排序，使用默认数据库排序**

**Redis为什么快：**1、C语言开发，与unix系统语言一样。2、所有数据放在内存中，非同步工作时，不需要磁盘IO，内存响应时间为100纳秒。3、单线程，避免了线程切换以及加锁带来的性能消耗。

使用多路复用IO模型，非阻塞IO。select，poll，epoll

**传统多进程并发模型：** 每监听到一个Socket连接就会分配一个线程处理

**多路复用模型：**通过记录跟踪每一个Socket连接的I/O的状态来同时管理多个网络I/O。

redis内部使用文件时间处理器file event handler,是单线程的，它采用多路复用机制同时监听多个socket。可能会产生不同的操作，每个操作对应不同的事件处理器。

select/epoll模型，在于单个进程就可以同时处理多个网络IO。它的基本原理就是select/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

1. Redis中的I/O多路复用程序会监听多个客户端连接的Socket
2. 每当有客户端通过Socket流向Redis发送请求进行操作时，I/O多路复用程序会将其放入一个队列中。
3. 事件分派器每次从队列中取出一个事件，把改事件交给对应的事件处理器处理。
4. I/O多路复用程序会在上一个请求处理完毕后再继续分派下一个任务。（同步）

单例模式：双重检查锁 懒汉式优化版

执行双重检查是因为，如果多个线程同时了通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么剩余通过了第一次检查的线程就不会再去实例化对象。

public class Singleton {

private volatile static Singleton uniqueSingleton; 加入volatile防止实例化对象重排序

private Singleton() {

}

public Singleton getInstance() {

if (null == uniqueSingleton) {

synchronized (Singleton.class) {

if (null == uniqueSingleton) {

uniqueSingleton = new Singleton();1、分配内存空间2、将对象指向分配的内存空间3、初始化对象

} } }

return uniqueSingleton; } }