MS

# Java基础

## String、StringBuffer、StringBuilder的区别

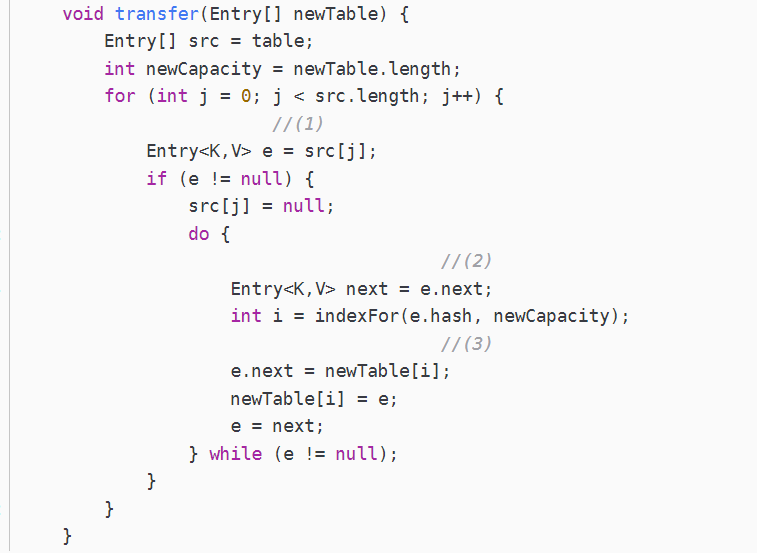
1. String是final类型，不允许继承，通过char数组来保存字符串
2. StringBuffer是线程安全的，但效率低
3. StringBuilder是非线程安全的，但效率高
4. 通过javap –c查看反编译，可以看到+=操作实际上就是一个个的StringBuilder，大量的循环+=操作会造成内存的浪费。需要注意的是如果是常量间的直接+，那么效率会更高。
5. String声明的字符串和StringBuffer和StringBuilder!=，主要是因为Buffer和Builder最后都是通过new String()去实现，而String声明是在编译期生成了字面常量和符号引用，值存在于运行时常量池，JVM会在运行时先检查运行时常量池是否存在，而new String()是在堆上生成一个对象，因此引用地址是对不上的

## ArrayList和LinkedList有什么区别

1. ArrayList是底层是数组，访问复杂度是O(1)，而LinkedList是使用链表，访问复杂度是O(n)
2. ArrayList 添加操作是很快的，但是del操作需要更新索引，因为array是连续的内存空间
3. ArrayList需要考虑扩容问题，而LinkedList不需要
4. LinkedList比ArrayList更耗费空间

## Map部分

1. LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在遍历的时候，得到的记录是有序的
2. TreeMap实现了SortedMap接口，能够将保存的记录根据键排序，默认是升序，使用的时候key需要实现Comparable接口或者在初始化的时候传入Comparable
3. HashMap原理：数组+链表+红黑树(链表长度大于8的时候转换为红黑树)
4. CocurrentHashmap锁分段技术：首先将数据分为一段一段存储，然后给每一段数据配一把锁，由Segment数据结构，它是一种可重入锁ReentrankLock，包含HashEntry存储数据。而jdk1.8之后则是采用cas和synchronized(Node)了
5. 之所以链表长度大于8的时候要转换为红黑树，是由于链表的查找性能是O(n)，而红黑树查找性能提高到O(log(n))

* HashMap使用链地址法(每个数组元素都加上一个链表结构)来解决哈希冲突问题，Node的默认初始化长度length是16，默认负载因子0.75，由于threshold(超过就扩容)=length\*负载因子，所以默认负载因子越大，可以存储的键值对就越多。
* Resize()扩容后的HashMap的数量是之前容量的两倍，用一个更大容量的数组然后把原有的数组复制到新数组中
* Hashcode的本质就是，取key的hashCode值，然后进行高位运算、取模运算
* Jdk8以前链表采用的是头插法，jdk8以后采用的是尾插法
* 头插法死循环的原因：
* 假如两个线程同时执行到(1)这一步，这个时候有一个链表
* OldTable[i]=a1-a2-null，线程2先执行完了，由于是使用e.next ,e=a1,e.next=a2，所以执行完之后就变成netTable[i]=a2-a1-null
* 这个时候cpu切回线程1，线程1看到的e=a2,e.next=a1。所以执行完之后newTable[i]=a1-a2-a1，所以再get到这个的时候就死循环了

## IO和NIO和reactor模型

JavaIO和NIO的一个最大区别是，IO是面向流的，NIO是面向缓冲区的，IO是阻塞的，而NIO是非阻塞的

## 浅拷贝和深拷贝

1. 浅拷贝：会对主对象进行拷贝，但不会复制主对象里面的对象，里面的对象会再原来的对象和它的副本之间共享
2. 深拷贝：会对主对象及其对象进行拷贝

## Error和Exception

1. Error和Exception有一个共同的父类Throwable，Error代表错误，而Exception代表异常。Exception一般分为Checked异常和Runtime异常。
2. Error指的是发生不可控的错误，通常是需要让用户终止程序的执行。由JVM生成并抛出，包括动态链接错误，虚拟机错误等。

## 抽象类和接口的区别

1. 抽象类允许存在抽象方法与默认方法，而接口全是抽象的方法
2. 如果抽象类添加了新方法，且是非抽象的，子类可以不实现，接口必须实现。Jdk8之后出现了default方法，接口子类也可以不实现
3. 抽象类继承一个抽象类，可以不实现抽象方法

## hashcode()和equals()的关系

1. hashcode()主要是用来定位，而equals是用来比较内存地址的引用
2. equals()相等时，返回的hashcode()一定相同
3. equals()不等时，返回的hashcode()可能相同
4. hashcode()相同，equals()可能不等
5. hashcode()不相同时，equals()一定不等
6. 为什么说重写equals()要重写hashcode()，因为hashmap中判断key的唯一性，先判断hashcode()是否相等，如果相同，再定位到对应的数据，再用equals()对比，如果相等则认为已存在

## Object常用的一些方法

Equals()、toString()、线程通信可能会用到notify()、wait()

## 反射及其用到的场景

1. 反射是在运行状态中，对于任意一个类都能知道它的所有变量和方法，对于任意一个对象，都可以调用它的方法
2. 使用场景：实际工作中，基本没怎么用到，但是看过的源码中可以发现，一般rpc调用最后都会用到反射去执行对应的方法

## 怎么新开线程

继承Thread、重写run方法

实现Runnable接口

## Hashmap如何实现线程安全

可以使用Collections.SynchronizedMap去实现线程安全，使用this做锁，用synchronzied(this)锁住了所有的操作。或者使用concurrentHashMap，使用分段锁，提高并发性能

## Java代理机制

1. 静态代理
2. 动态代理，又分java动态代理，cglib动态代理

* 动态代理主要代表实现有spring aop、mybatis的interface映射
* jdk动态代理只能针对接口的类生成代理，而不能针对类。
* Cglib是生成类生成代理，主要是对指定的类生成一个子类，继承覆盖其中的方法。Cglib比jdk动态代理快，采用asm字节码生成技术，但是不能代理final方法

## 写出5个运行时异常

1. IndexOutOfBoundsException
2. NullPointerException
3. ClassCastException
4. illegalArgumentException
5. ArrayStoreException

## 什么是happens-before

Happens-before用来阐述操作之间的内存可见性，在JMM中，一个操作执行结果需要对另一个线程可见，那么这两个操作必须要存在happens-before关系，这两个操作可以是一个线程，也可以是多线程。需要注意的是，如果不改变程序的执行结果，那么是可以重排序的

## 什么是上下文切换

1. CPU通过时间片分配算法循环执行任务，当前任务执行一个时间片之后切换到下一个任务。但是，在切换前会保存上一个任务的状态，切回来的时候可以继续执行。这个任务从保存到再加载的过程就是一次上下文切换
2. 减少上下文切换，可以看看是否空闲线程太多，尝试减少maxThread

## 什么是volatile

1. Volatile在多处理器开发中保证共享变量的可见性，禁止指令的重排序。
2. 对一个volatile变量的读，JMM会把线程本地变量置为无效，接下来从主内存中读取。当写一个volatile变量时，JVM会把该线程对应的本地内存中的共享变量刷新到主内存。
3. 为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障(一组处理器指令，用于实现对内存操作的顺序限制)来禁止特定的处理器重排序
4. Volatile仅仅保证对单个volatile变量的读/写具有原子性
5. Volatile两条实现原则，Lock前缀会引起处理器缓存写回内存，一个处理器的缓存写会内存，会导致其他处理器的缓存无效

## 什么是synchronized

1. Synchronzied是java提供的一个并发控制的关键字
2. 主要用法有同步方法和同步代码块
3. 同步块原理是在开始位置插入moniterenter和结束或异常处插入monitorexit指令。
4. 同步方法使用的是ACC\_SYNCHRONIZED指令来完成，本质也是enter和exit
5. 对一个对象的监视器进行获取，这个过程是排他的，获取不到的进入BLOCKED状态，进入SynchronizedQueue队列

## 什么是偏向锁

为了使线程获取锁的代价更低，当一个线程访问同步块并获取锁时，会在对象头和栈帧中存储锁偏向的线程id，以后该线程进入和推出的时候，不需要进行cas操作来加锁和解锁，当其他线程尝试争取偏向锁时，持有的线程才会释放锁

## 什么是轻量级锁

* 轻量级锁加锁：执行同步块之前，JVM会在当前线程的栈帧中创建用于锁记录的空间，并将对象头的mark word复制到这个空间中，然后线程尝试使用cas将对象头mark word替换指向锁空间的指针，如果成功，当前对象获取锁，失败则自旋。
* 解锁：使用cas操作将这个空间替换回对象头，如果成功，则表明没有竞争的发生，如果失败，则膨胀成重量级锁

## 什么是自旋锁

不让出cpu，执行空循环

## ReentrantLock

1. ReentrantLock的实现依赖于java同步器框架(AQS)AbstractQueuedSynchronizer，使用state(volatile)来维护同步状态
2. 公平锁：加锁前检查是否有排队等待线程，先来先得
3. 非公平锁：加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待
4. 吞吐量：公平锁比较低，因为当一个线程释放锁时，可以快速的通过tryAcquire()方法再次获取锁，其他排队线程还是继续等待，会造成饥饿现象。
5. 可重入：单线程可以重复进入，但是需要重复推出，采用的是state+1
6. 可中断：lockInterruptibly()
7. 可限时：超时不能获取锁，就返回false，不会一直等待

## Synchronized和reentrankLock区别，性能差异

reentrankLock提供了多样化的同步，比如时间限制的同步、可以被interrupt的同步，当竞争激烈的时候，synchronized会下降的比较多，而reentrankLock可以维持

## 什么是Condition

类似与wait()和notify()，主要和ReentrantLock一起使用，使用await()等待并unlock释放锁，使用signal()/singnalAll()唤醒

## 什么是Semaphore信号量

信号量是一个可以同时容纳N人的房间，如果人不满就可以进去，人满了就需要等人出来

## 什么是CountDownLatch

倒数计算器，所有的线程都完成后再执行，使用new CountDownLatch(count)，当一个工作完成则执行countDown()，使用await()等待工作完成

## 线程池

**为什么要使用线程池？**

减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，执行多个任务。防止开太多线程，浪费太多内存

**自定义线程池，线程数量如何确定？**

IO密集型任务：由于线程并一直在执行，所以可以尽可能的多配置线程，比如CPU个数\*2，由于cpu利用率低，程序中会出现大量I/O操作占据时间，导致线程空余时间出来，所以通常就需要开cpu核数的两倍的线程，当线程进行I/O操作cpu空暇时启动其他线程继续使用cpu，提高cpu使用率

CPU密集型任务：应当分配较少的线程，比如CPU个数相当的大小，cpu使用率高，也就是一些复杂运算，逻辑处理等。

总结一下就是线程等待时间所占比例越高，需要越多线程。CPU时间所占比例越高，需要越少线程

**线程池包括了四个基本组成：**

1. 线程池管理器：ThreadPool，用于创建并管理线程池，包括创建、销毁、添加新任务
2. 工作线程：PoolWorder，线程池中队列，在没有任务时处于等待状态
3. 任务接口：Task，每个任务必须实现的接口，以供工作线程调度任务的执行
4. 任务队列：taskQueue，用于存放没有处理的任务，提供一种缓冲机制

## 什么是ThreadPoolExecutor

**主要参数有哪些：**

1. CorePoolSize：线程池的基本大小，一般不会退出，除非设置了allowCoreThreadTimeOut
2. MaximumPoolSize：线程池中允许的最大线程数
3. KeepAliveTime：空闲线程退出时间
4. WorkQueue：线程池使用的缓冲队列
5. Handler：对拒绝任务的处理策略

**运行机制：**

1. 如果线程数小于corePoolSize，那么马上创建线程运行这个任务。
2. 如果正在运行的线程数大于等于corePoolSize，那么就把任务放入队列。
3. 如果队列满了，且运行中的线程小于maximunPoolSize那么就继续创建线程运行。
4. 如果队列满了，且大于maximunPoolSize，那么会执行拒绝策略

**拒绝策略：**

1. AbortPolicy()：抛出异常
2. CallerRunsPolicy()：重试添加当前的任务，会自动调用execute方法
3. DiscardOldestPolicy()：抛弃旧任务
4. DiscardPolicy()：抛弃当前任务

## 阻塞队列

**阻塞队列和非阻塞队列的区别：**

1. 阻塞队列：当队列空，要获取元素的队列会阻塞。当队列满，插入元素的队列会阻塞
2. 非阻塞队列：不具备阻塞特点，如常见的LinkedList
3. 使用非阻塞队列的问题在于不会阻塞线程，在面对类似消费者-生产者模型时，就必须额外的实现同步策略与唤醒策略

**阻塞队列种类：**

1. 有界队列：ArrayBlockingQueue，LinkedBlockingQueue(capacity)基于数组实现的一个阻塞队列，可以指定公平或非公平，默认不公平
2. 无界队列：LinkedBlockingQueue，基于链表实现的一个阻塞队列，默认大小为MAX\_VALUE
3. 优先队列：PriorityBlockingQueue，会按照元素的优先级进行排序，按顺序出队，无界
4. 延时阻塞队列：DelayQueue，只有当其指定的延迟时间到了，才能从队列中获取到，无界
5. 无容量队列：SynchronousQueue，无容量，每个put必须等待一个take()，类似一手交钱一手交货的场景

**主要方法：**

1. 非阻塞队列：offer(E e)元素插入队尾，成功返回true，否则返回false、poll()：移除并获取队首元素，失败返回null、peek()：获取队首元素，失败返回null
2. 阻塞队列：put(E e)元素插入队尾，队列满则等待。take()方法用来从队首取元素，如果队列空，则等待。offer()元素插入队尾，队列满，则等待一定的时间，期限到返回true/false。poll()用来从队首取元素，如果队列空，则等待一定的时间，返回null/value

**实现原理：**

拿ArrayBlockingQueue来说，使用Object[]存储数据，用ReentrantLock做锁，并存在变量名为notEmpty和notFull的Condition，使用await()和singal()做阻塞等待。事实上和我们自己用wait()和notify()对非阻塞队列做阻塞，思路类似

**看过的使用场景：**

在xxl-jobb的执行器中，工作线程会把执行任务放入阻塞队列LinkedBlockedQueue，并使用poll(seconds)进行超时操作

# JVM

## Java内存区域(运行时数据区)

1. 堆：所有的对象实例和数组都在堆上做分配，堆又分新生代、老年代。新生代又分Eden、From Survivor、To Survivor(8:1:1)
2. 栈：描述java方法的内存模型，栈是由一个个栈帧组成，每个栈帧又拥有局部变量表、操作数栈、动态链接、方法出口等
3. 本地方法栈：虚拟机使用到的native方法服务
4. 程序计数器：当前线程所执行字节码的行号指示器
5. MetaSpace：由Klass MetaSpace(klass指的是运行时数据结构)，NoKlass MetaSpace(存储klass相关其他内容，比如method、常量池等)，它存在于本地内存。默认情况下，仅受本地内存大小限制。
6. 运行时常量池由1.7的存在方法区中，移动到了堆中

## 为什么要把方法区替换成MetaSpace

1. 大小在启动的时候就确定，比较难调整。
2. 简化FULL GC
3. 可以在GC不进行暂停的情况下并发地释放类数据
4. 还有就是改进受限的方法区

## Java对象创建过程

类加载检查、分配内存、初始化零值、设置对象头、执行init方法

## 对象的内存布局

对象头、实例数据、对齐填充

## 对象访问定位的两种方式

句柄、直接指针

## 虚拟机是如何加载java类的

加载：查找字节流，并且据此创建类。需要借助类加载器(使用的是双亲委派模型)

链接：创建的类合并到Java虚拟机中，使之能够执行。(验证、准备、解析)

初始化：为标记为常量的字段赋值，以及执行方法的过程。

## 类加载器的种类

从下至上：自定义ClassLoader=>Application ClassLoader=>Extension ClassLoader=>Boostrap ClassLoader

## ClassLoader加载一个类的过程/如何自定义classLoader

1. loadClass()为入口，首先通过native方法findLoadedClass()查找.class是否被加载过
2. 没被加载过，会尝试使用parent classloader loadClass()，执行1的流程，知道找不到(Bootstrap classLoader)为止。
3. 如果还是找不到，那么就尝试findClass()，而这个findClass()是protected的，也就是可以继承ClassLoader，并重写
4. 自定义的findClass()，可以加载自定义的class，接着转换成byte[]，然后调用classloader的defineClass将其转换成对象。
5. 使用的时候，可以使用Class.forName()

## Class.forName()和classLoader().loadClass的区别

1. Class.forName()会将类加载到jvm中，还会对类进行解析，执行类中的static等等
2. ClassLoader().loadClass()只会将类加载到jvm中，只有在newInstance()才会去执行static

## 什么是双亲委派模型

非顶级的类加载器都要有自己的父类加载器，加载一个类的时候，会把加载器交给父加载去执行，由下到上、找不到再由上到下。目的是防止内存中存在多份一样的字节码，保证类在JVM中的唯一性

## GC在什么时候，对什么东西，做了什么事情

**Minor gc(young gc)：**

* 当eden区满了，
* 或者对象大于eden区的时候触发，
* fullgc触发

**Major gc：(old gc)**

**清理Old**

**Full gc：(all gc)**

* Old区空间不足。
* MetaSpace空间不足。
* Ygc的对象到达一定的年龄后，promotion failure(晋升到old失败)。
* 显示调用System.gc()

**对什么东西：**

从root开始搜索，搜索不到的对象。并且经过第一次标记、清理后，没有复活的对象

**做了什么事情：**

Minor gc：一般分为Eden、survivor、to，使用复制算法进行清理

Full gc：一般是标记清理

## 什么是G1垃圾回收器

G1的设计原则：简单可行的性能调优

G1将整个堆划分为一个个大小相等的region，region的内存是连续的，也会充当Eden、Survivor、Old三个角色，但是它们是不固定的。

执行垃圾收集时，收集线程和应用线程并发执行(与CMS一样)。当JVM分配Eden Region失败后就触发一个年轻代回收，会移动所有的Eden Region存储对象到Survivor Region，也就是Copy To Survivor过程。当堆空间耗尽，会进行初始化标记=》

# 网络

## 一个http请求的过程

1. 域名校验
2. DNS域名解析，找到服务ip
3. 建立TCP连接(3次握手)
4. 发起http请求
5. 服务器响应http请求

# 数据结构与算法

# Spring boot/cloud

## 什么是Spring

Spring是个企业级应用的开源框架，目标是简化java企业级开发

## 使用Spring有什么好处

轻量、提供AOP、MVC、事务管理、异常管理等

## 为什么要使用Spring Boot

主要是简化配置，自动配置

## 核心配置文件都有哪些

Application和bootstrap文件，application文件主要用于自动化配置。Bootstrap主要是用于固定的不能被覆盖的场景，加载顺序再application之前

## 自动化配置的原理

主要是使用@EnableAutoConfiguration，通过ImportSelector，找到对应的spring.factories文件，加载里面需要自动配置的类，一般配置的类会添加@Configuration注解和@Condition注解

## Spring事务隔离级别

* **脏读：一个事务读取了被另一个事务改写但尚未提交的数据。**
* **不可重复读：一个事务执行相同两次或两次以上的查询，但查询结果都不同，通常是因为另一个并发事务在两次查询之间更新了数据**
* **幻读：当一个事务读取几行记录后，另一个事务插入一些数据，后来的查询，会发现一些原来没有的数据**

1. **DEFAULT：PlatfromTransactionManager默认的隔离级别，使用数据库默认事务隔离级别**
2. **READ UNCOMMITED(未提交读)：脏读，不可重复读，幻读都有可能发生**
3. **READ COMMITTED(已提交读)：避免脏读，但是不可重复读和幻读都有可能发生**
4. **REPEATABLE READ(可重复读)：避免脏读和不可重复读，但是幻读有可能发生**
5. **SERIALIZABLE(串行化)：避免以上所有读问题**

## Spring事务传播行为

* 1. REQUIRED：当前方法必须在一个事务中运行，如果有了那么就复用，否则新开
  2. SUPPORTS：不需要事务，但是如果有事务，就复用
  3. MANDATORY：必须有事务，没有则抛出异常
  4. REQUIRES\_NEW：必须新开事务，挂起现有事务
  5. NOT\_SUPPORTED：没有事务，会挂起现有事务
  6. NEVER：没有事务，有则抛出异常
  7. NESTED：嵌套事务，可以自己提交或回滚

## Spring事务在什么时候回滚、什么时候失效

Spring事务只在发生runtime exception时回滚，在checked exception不会滚，也可以自己指定rollback类型。

Spring事务使用要小心this调用，因为Spring事务是基于aop的动态代理去实现的，this调用会导致方法不被代理，导致失效，记得使用AopUtil获取bean也是可以的

# 数据库相关

## 数据库的事务隔离级别

1. READ\_UNCOMMITED(未提交读)：最低的隔离级别，允许读取尚未提交。比如说A向B转账，A还没提交事务，B读取数据就说自己收到钱了，A回滚事务，B发现没钱了。
2. READ\_COMMITTED(提交读)：允许读取并发事务已经提交的数据，可以阻塞脏读，但是幻读或不可重复读仍有可能发生。本质是把释放锁的位置调教到事务提交后，这样子其他数据就无法对其修改
3. REPEATABLE\_READ(可重复读)：对同一字段的多次读取结果是一致的，除非数据是被本身事务自己修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。每次读取的都是当前事务的版本，即使被修改了，也只会读取当前事务版本的数据。Mysql这个级别加上GAP间隙锁已经处理了幻读
4. SERIALIZABLE(串行)：最高的隔离级别，完全服从ACID。所有事务依次逐个执行，这样事务之间就完成不可能产生干扰

## 多事务并行问题

脏读：一个事务修改了，没提交又rollback，另外一个事务又拿去用了，导致操作是错误的

丢失修改：两个事务同时读取，又同时修改，导致update丢失。A=20，事务1做了A=A-1，事务2也做A=A-1，最后结果=19 是错的

## Mysql调优

1. 擅用explain，需要了解explain之后各列的意义
2. 使用短索引
3. Like后缀
4. 不需要考虑=、in等的顺序，mysql会自动优化。匹配尽可能多的索引

## 索引

1. 最左前缀匹配原则。Mysql会一直向右匹配直到遇到范围查询
2. 尽量选择分区度高的列做为索引，COUNT(DISTINCT col)/COUNT(\*)
3. 索引列不能参与计算
4. 尽可能的扩展索引

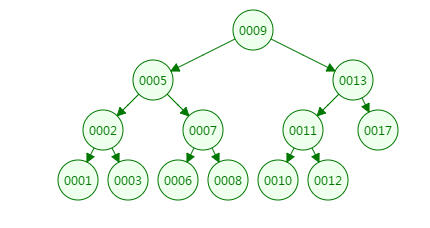
## 行级锁

INNODB实现了两种类型的行锁

1. 共享锁(S锁)：允许一个事务去读一行，阻止其他事务获得相同数据集的写锁，也叫做读锁。读锁是共享的，多个客户可以同时读取同一资源，但不允许修改
2. 排它锁(X锁)：允许获得排它锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。也叫做写锁，排他且阻塞其他读写锁

## B+TREE

1. 为什么要使用B+TREE，先看看什么是B-TREE
2. 文件很大，不可能全部存储在内存中，故要存储到磁盘上
3. 索引的结构组织要尽量减少查找过程中磁盘I/0的次数，因为一次磁盘I/0大概需要10ms，速度是内存的1/10W
4. 比如这个从0009找到0010 需要经历4次磁盘IO。所以最坏的情况下磁盘IO的次数由树的高度来决定，要减少IO次数，就必须压缩树的高度



# Mybatis

## Mybatis定义的接口，如何找到实现的

# Redis

## Redis雪崩

描述：缓存同一时间大面积的失效，后面的请求全部落到数据库上，造成数据库短时间内承受大量请求而崩掉

解决：尽量保证整个redis集群的高可用，选择合适的淘汰策略。

本地加缓存，还有做限流/降级

如果做了持久化，尽快恢复

## Redis击穿

热点key可能在某些时间点被超高并发的访问，缓存过期后，恰好很多key一致性打到db

## Redis穿透

简介：故意请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉

解决：采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中。或者如果查询key或者value为空，也把null结果缓存到redis中，但是过期时间不要设置太长

# Mq

## ActiveMq的几种通信模式

1. Topic(发布订阅模式)
2. Queue(p2p模式)

## 如何解决消息重复的问题

一般是消费方自己做业务处理

## 如何持久化数据

1. Jdbc持久化到数据库
2. AMQ(日志文件)
3. KahaDb和LevelDb

## Failover连接导致卡死

如果所需要的activemq宕机，那么程序会一直等待，不超时不报错。解决方案是不采用failover

## Rabbitmq如何处理消息丢失问题

1. 生产者：可以开启事务，如果没有被rabbitmq收到，会异常报错。但吞吐量会降低。也可以开启confirm模式，每个消息会分配一个唯一id，然后被rabbitmq收到，会传一个ack消息，确实是否成功。这两种区别在于，事务是同步的，会阻塞。而confirm是异步的
2. Rabbitmq：如果是rabbitmq丢数据，那么需要考虑消息持久化

## Rabbitmq基于什么传输

由于TCP连接的创建和销毁开销比较大，且并发数收系统资源限制，会有性能瓶颈。所以rabbitmq是使用信道的方式来传输的，这是一种建立再TCP连接内的虚拟连接，没有限制条数

# 方案性

## 分布式锁实现方式

1. 数据库

* 乐观锁：乐观锁一般是基于添加类似version去做，可以避免ABA问题。例如Update table set status=1,version=2 where id={} and version=1
* 悲观锁：悲观锁一般基于数据库自带的锁，当然需要注意，where column要带索引，也就是使用行锁。记得提交事务！！例如：select id form table where id={} for upadte

1. Redis

* Redis一般网上的文章都会说使用SETNX+expire，然后又会说如果SETNX成功，出现某些问题，expire失败了，会死锁。本来使用SETNX+expire就是为了避免死锁，又会推荐你写lua。
* 其实分版本，如果redis版本在2.8以上，redis本身支持set key value ex [] nx命令，setnx+expire一步到位。
* 如果需要手动del锁，还需要注意value用uuid等唯一id。当get value=内存中的uuid的时候，再去del
* Spring Boot下推荐做法是封装成注解，使用aop再拿到Transaction commit通知的时候，再del。另外也可以自己研究下redisTemplate源码，底层也是jedis，改造下就可以用了

1. ZK