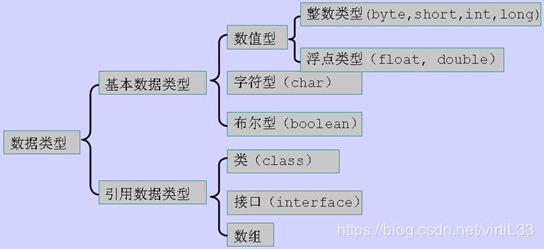
JAVA面试 题**重点**

# 源码，算法，JVM,设计与模式，线程。（重点准备）

# 1 JAVA基础

## （线程同步，线程池，jvm常见异常与处理，红黑树，ks8与docker,分布式，memchace,zookpeer,kafka,网络协议，linux性能查询，BIO、NIO、AIO）

## Java常用基本类型与String。



取值范围：如byte取 ｛-2^7，2^7-1｝ ;1B（byte，字节）= 8 bit

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 简单类型 | boolean | byte | char | short | Int | long | float | double |
| 二进制位数 | 1 | 8 | 16 | 16 | 32 | 64 | 32 | 64 |
| 封装器类 | Boolean | Byte | Character | Short | Integer | Long | Float | Double |

其中基本类型为值传递，引用类型为副本传递（String例外，其为不可变类型，所以类似于值传递）,String类用final修饰类，底层是一个 char[];

String s =”123” 创建一个或者0个对象。New String(“123”) 创建一个或者两个对象。

## String、Stringbuffer和stringbuilder区别。

String：不可变字符串；每次操作都会产生新的String对象，浪费内存。

StringBuffer：可变字符串、效率低、线程安全（采用了 synchronized 关键字进行修饰）；

初始默认为16的容量，ensureCapacityInternal方法判断是否需要扩容，需要扩容就调用expandCapacity方法进行扩容，扩容为2n+2，直到扩容足够;

StringBuilder：可变字符序列、效率高、线程不安全；



## Object 常用方法。

toString(),equeals(),clone(),hashCode(),notify(),notifyAll(),wait(),getClass(),finalize();

## 1.4==与equals()区别。

==比较对象的引用是否指向同一块内存地址（栈）。equals方法默认也是用来比较两个对象的引用是否指向同一块内存地址（栈），如果要对象的内容是否相等（堆内），需要重写对象的equals方法。

## 1.5各种类型转int

Double d = 1.0;  
String s = "123";  
Integer a = Integer.*valueOf*(s);//转为Integer，大于127不能使用==比较  
int b = Integer.*parseInt*(s);//转为int，大于127能使用==比较  
Integer c = d.intValue();

## 1.6 final、finally、finalize

1、final可以修饰类,不能被继承； 修饰方法,不能被重写；修饰变量,只能赋值一次  
    2、finally是try语句中的一个语句体,不能单独使用,用来释放资源  
    3、finalize是一个方法,当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

## 1.7 arraylist和linkedlist（包含结构与线程安全，选型）。

Arraylist基于数组实现，方便查询，不好做新增与删除（需要移位）。

Linkedlist基于双向循环链表实现，方便删除与新增，不方便查询（需要从前往后，或者从后往前遍历）。

## 1.8 HashMap底层结构。线程安全map。

HashMap底层结构:数组+单链表（<8个）+红黑树（>8个）。

线程安全的map:hashtable, synchronizedMap,这两个加synchronized实现，性能差。

ConcurrentHashMap是cas+synchronized+volatile来实现的，其中也可看出它的锁是分段锁，所以它的性能相对来说是比较好的。整体实现还是比较复杂的

## 1.9 集合中Set与List和Map区别。以及线程安全。

以下基于java1.8

Collection 接口的接口 对象的集合（单列集合）

├——-List 接口：元素按进入先后有序保存，可重复

│—————-├ LinkedList 接口实现类，双向循环链表，线程不安全。查询效率差，删除与新增快。插入有 序（不是大小有序），能存多个null.

│—————-├ ArrayList 接口实现类，数组，线程不安全。 查询快，删除与新增差。插入无序（不是大小 有序），能存多个null.

│—————-└ Vector 接口实现类 数组，线程安全， 同步，

└——-Set 接口： 仅接收一次，不可重复，并做内部排序

├—————-└HashSet 使用hash表（数组）存储元素

│————————└ LinkedHashSet 链表维护元素的插入次序

└ —————-TreeSet 底层实现为二叉树，元素排好序

Map 接口 键值对的集合 （双列集合）

├———Hashtable 接口实现类， 同步， 线程安全

├———HashMap 接口实现类 ，没有同步， HashMap底层结构:数组+单链表（<8个）+红黑 树（>8个）。线程不安全-

│—————–├ LinkedHashMap 双向链表和哈希表实现

│—————–└ WeakHashMap

├ ——–TreeMap 红黑树对所有的key进行排序

└———IdentifyHashMap

├———ConcurrentHashMap 线程安全 ，cas+synchronized+volatile来实现的，其中也可看出它的锁是分段锁

├———synchronizedMap 线程安全

## 1.10 线程安全的Map。

**数组+链表（单链表）+红黑树的存储方式**，在性能上进一步得到提升。

数组存放不冲突的map,get()时key和hashcode同时标识

Node(int hash, K key, V value, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = next;//hash冲突时使用，链表的node数量大于8，则将链表转为红黑树  
}

数组里面存放值与链表（单链表）/红黑树

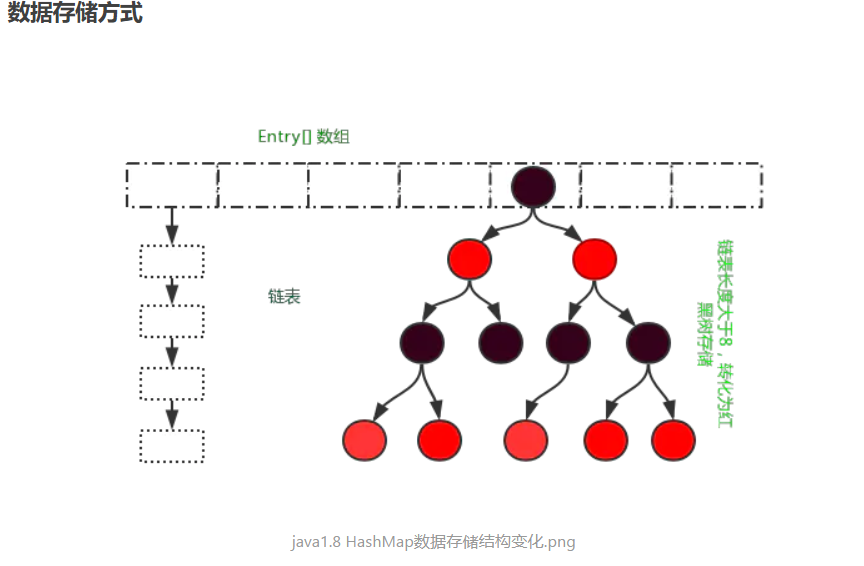
· 如果当前map中没有数据，执行resize方法

· 如果要插入的键值对要存放的位置上刚好没有元素，那么就把它封装成Node对象，并放在这个位置上。

· 如果发生碰撞，判断node的类型是红黑树还是链表：  
3.1 如果为红黑树，则将K-V对插在红黑树对应的位置。  
3.2 如果为链表，遍历链表：  
　a.如果为链表最后一个node ,则将新的node节点插入到链表尾  
　b.插入完，如果链表的node数量大于8，则将链表转为红黑树的操作；如果当前哈希表为空或数组长度小于64，会扩容，否则转化为红黑树。转化的过程：先遍历链表 ，将链表的节点转化为红黑树的节点；然后将链表转化为红黑树。  
c.遍历链表时，如果key已存在，则直接bredk循环。

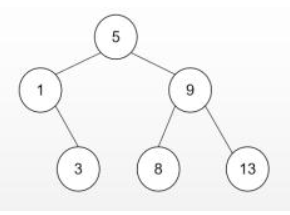
· 判断是否要扩容

· 返回



**注：**

平衡二叉树，又称AVL树，指的是左子树上的所有节点的值都比根节点的值小，而右子树上的所有节点的值都比根节点的值大，且左子树与右子树的高度差最大为1。因此，平衡二叉树满足所有二叉排序（搜索）树的性质。



**红黑树**<https://blog.csdn.net/tanrui519521/article/details/80980135>

祖宗根节点必黑，允许黑连黑，不允许红连红；新增红色，爸叔通红就变色，爸红叔黑就旋转，那黑往那旋。

## 1.11 Collections工具类与Collection接口。

Collections是一个操作Set、Map和List的工具类，提供了大量方法对集合元素进行排序、查询和修改等操作。

Collection 是集合的顶级父类接口（set与list）

## 1.12 线程创建的方式。

1）继承Thread类创建线程

public class MyThread extends Thread{//继承Thread类  
 　　public void run(){  
 　　//重写run方法  
 　　}  
 }  
 public class Main {  
 　　public static void main(String[] args){  
 　　　　new MyThread().start();//创建并启动线程  
 　　}  
 }

1. 实现Runnable接口创建线程

public class MyThread2 implements Runnable {//实现Runnable接口  
　　public void run(){  
　　//重写run方法  
　　}  
}  
public class Main {  
　　public static void main(String[] args){  
　　　　//创建并启动线程  
　　　　MyThread2 myThread=new MyThread2();  
　　　　Thread thread=new Thread(myThread);  
　　　　thread().start();  
　　　　//或者 new Thread(new MyThread2()).start();  
　　}  
}

1. 使用Callable和Future创建线程

public static class MyThread3 implements Callable{  
 @Override  
 public Object call() throws Exception {  
 return 5;  
 }  
}  
public class Main {  
　　public static void main(String[] args){  
　　　MyThread3 th=new MyThread3();  
　　　//也可以直接使用Lambda表达式创建Callable对象  
　　 //使用FutureTask类来包装Callable对象  
　　　FutureTask<Integer> future=new FutureTask<Integer>(  
 　　　　(Callable<Integer>)()->{  
　　　　　　return 5;  
　　　　}  
　　 );  
　　　new Thread(future,"有返回值的线程").start();//实质上还是以Callable对象来创建并启动线程  
　　 try{  
　　　　System.*out*.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回  
 　　 }catch(Exception e){  
　　　　ex.printStackTrace();  
　　　}  
　　}  
}

1. 使用线程池例如用Executor框架

// ExecutorService executorService = Executors.newCachedThreadPool();   
// ExecutorService executorService = Executors.newFixedThreadPool(5);   
// ExecutorService executorService = Executors.newSingleThreadExecutor();  
// Executors.newScheduledThreadPool(int)  
自定义： new ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,BlockingQueue<Runnable> workQueue)自定义创建

## 1.13 线程sleep()与wait()区别。

sleep()方法的过程中，线程不会释放对象锁。是线程的方法。

wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程

## 1.14 线程同步的方式与volatile。

<https://www.cnblogs.com/dolphin0520/p/3920373.html>

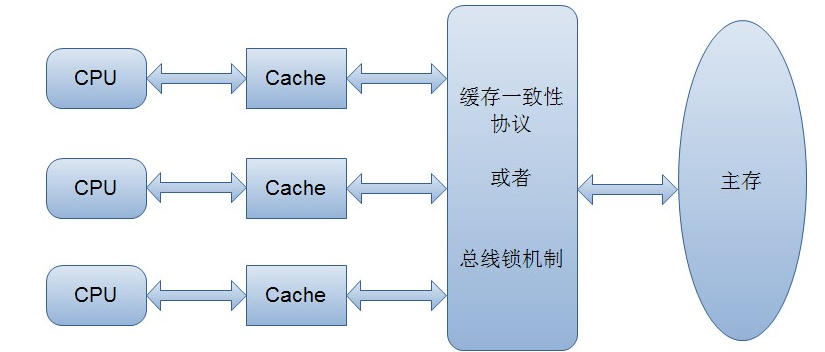
1.volatile只能保证可见性，无法保证原子性

先写入高速缓存，再刷新到内存(主存)中。

单线程中运行是没有任何问题的，但是在多线程中运行就会有问题了。在多核CPU中，每条线程可能运行于不同的CPU中，因此每个线程运行时有自己的高速缓存（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的），

初始时，两个线程分别读取i的值存入各自所在的CPU的高速缓存当中，然后线程1进行加1操作，然后把i的最新值1写入到内存。此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存。

　　最终结果i的值是1，而不是2。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。--**共享变量，缓存一致性问题**



解决方法有：1java单机锁LOCK，synchronized

2分布式锁：redis,zookpeer,mysql等实现

3乐观锁CAS等

并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

可见性：是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性：即程序执行的顺序按照代码的先后顺序执行。

Java语言 本身对 原子性、可见性以及有序性提供了哪些保证呢？

Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。

对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。如x=1;

volatile关键字来保证可见性。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

**volatile关键字的两层语义**

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

　1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

　2）禁止进行指令重排序。

　synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

　　1）对变量的写操作不依赖于当前值

　　2）该变量没有包含在具有其他变量的不变式中

适合使用在状态标记已经单列返回对象。

## 1.15 线程的状态与什么时候达成。

https://blog.csdn.net/qq\_22771739/article/details/82529874

线程5个状态：新建、就绪、运行、阻塞、死亡。

1. 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。  
   2. 运行(RUNNABLE)：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。（就绪状态与运行中）  
   线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。  
   3. 阻塞(BLOCKED)：表示线程阻塞于锁。--（加锁）  
   4. 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。--（锁释放）  
   5. 超时等待(TIMED\_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回。

6. 终止(TERMINATED)：表示该线程已经执行完毕。

## 1.16 线程池原理与类型区别。

**(1) 介绍线程池原理**  
  
ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器，前三个基于第四个实现。  
参数：核心线程数，最大线程数，线程空闲时间，空闲时间的单位，阻塞队列（核心线程满时，任务进队列），线程工厂，拒绝策略。

**corePoolSize**：核心线程池大小，创建线程池时，线程池默认没有线程，当任务来时才会创建线程。除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目**达到corePoolSize后，就会把到达的任务放到缓存队列当中**；  
**maximumPoolSize**：线程池最大线程数，线程池能创建的最大线程数。  
**keepAliveTime**：线程空闲时间，表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；  
**unit**：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：  
      
    TimeUnit.DAYS;               //天  
    TimeUnit.HOURS;             //小时  
    TimeUnit.MINUTES;           //分钟  
    TimeUnit.SECONDS;           //秒  
    TimeUnit.MILLISECONDS;      //毫秒  
    TimeUnit.MICROSECONDS;      //微妙  
    TimeUnit.NANOSECONDS;       //纳秒  
**workQueue**：一个阻塞队列，用来存储等待执行的任务。  
    ArrayBlockingQueue; LinkedBlockingQueue; SynchronousQueue;  
ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。  
**threadFactory**：线程工厂，主要用来创建线程；  
**handler**：表示当拒绝处理任务时的策略，有以下四种取值：  
    ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。   
    ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。   
    ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）  
    ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务   
  
ThreadPoolExecutor、AbstractExecutorService、ExecutorService和Executor  
在ThreadPoolExecutor类中有几个非常重要的方法：  
execute();submit();shutdown();shutdownNow()  
 **execute()**方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。  
　　**submit()**方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够**返回任务执行的结果**，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果（Future相关内容将在下一篇讲述）。  
　　**shutdown()和shutdownNow()**是用来关闭线程池的。  
　　还有很多其他的方法：  
　　比如：getQueue() 、getPoolSize() 、getActiveCount()、getCompletedTaskCount()等获取与线程池相关属性的方法，有兴趣的朋友可以自行查阅API。  
  
**创建流程**  
 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；  
 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；  
 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；  
 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

1. **常用线程池**

**newCachedThreadPool：**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

* 底层：返回ThreadPoolExecutor实例，corePoolSize为0；maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为60L；unit为TimeUnit.SECONDS；workQueue为SynchronousQueue(同步队列)
* 通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。
* 适用：执行很多短期异步的小程序或者负载较轻的服务器

**newFixedThreadPool：常用**

* 底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，corePoolSize为nThread，maximumPoolSize为nThread；keepAliveTime为0L(不限时)；unit为：TimeUnit.MILLISECONDS；WorkQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列
* 通俗：创建可容纳固定数量线程的池子，每隔线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)
* 适用：执行长期的任务，性能好很多

**newSingleThreadExecutor:**

* 底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；unit为：TimeUnit.MILLISECONDS；workQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列
* 通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)
* 适用：一个任务一个任务执行的场景

**NewScheduledThreadPool:**

* 底层：创建ScheduledThreadPoolExecutor实例，corePoolSize为传递来的参数，maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为0；unit为：TimeUnit.NANOSECONDS；workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列
* 通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构
* 适用：周期性执行任务的场景

线程池任务执行流程：

1. 当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。
2. 当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行
3. 当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务
4. 当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理
5. 当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程
6. 当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

备注：

一般如果线程池任务队列采用LinkedBlockingQueue队列的话，那么不会拒绝任何任务（因为队列大小没有限制），这种情况下，ThreadPoolExecutor最多仅会按照最小线程数来创建线程，也就是说线程池大小被忽略了。

如果线程池任务队列采用ArrayBlockingQueue队列的话，那么ThreadPoolExecutor将会采取一个非常负责的算法，比如假定线程池的最小线程数为4，最大为8所用的ArrayBlockingQueue最大为10。随着任务到达并被放到队列中，线程池中最多运行4个线程（即最小线程数）。即使队列完全填满，也就是说有10个处于等待状态的任务，ThreadPoolExecutor也只会利用4个线程。如果队列已满，而又有新任务进来，此时才会启动一个新线程，这里不会因为队列已满而拒接该任务，相反会启动一个新线程。新线程会运行队列中的第一个任务，为新来的任务腾出空间。

这个算法背后的理念是：该池大部分时间仅使用核心线程（4个），即使有适量的任务在队列中等待运行。这时线程池就可以用作节流阀。如果挤压的请求变得非常多，这时该池就会尝试运行更多的线程来清理；这时第二个节流阀—最大线程数就起作用了。

**使用线程池风险：**

1. **死锁**：线程A持有对象X的锁等待对象Y的锁。而线程B持有对象Y的锁却等待对象X的锁。线程池中体现为池线程都在执行已阻塞的等待队列中另一任务的执行结果的任务，但这一任务却因为没有未被占用的线程而不能运行。
2. **资源不足**：如内存与JDBC连接不够。
3. **并发错误**：线程池和其它排队机制依靠使用 wait() 和 notify() 方法，这两个方法都难于使用。如果编码不正确，那么可能丢失通知，导致线程保持空闲状态，尽管队列中有工作要处理。使用这些方法时，必须格外小心。而最好使用现有的、已经知道能工作的实现，例如 util.concurrent 包。
4. **线程泄漏**：当从池中除去一个线程以执行一项任务，而在任务完成后该线程却没有返回池时，会发生这种情况。发生线程泄漏的一种情形出现在任务抛出一个 RuntimeException 或一个 Error 时。如果池类没有捕捉到它们，那么线程只会退出而线程池的大小将会永久减少一个。当这种情况发生的次数足够多时，线程池最终就为空，而且系统将停止，因为没有可用的线程来处理任务。
5. **请求过载**：请求就压垮了服务器，这种情况是可能的。在这种情形下，我们可能不想将每个到来的请求都排队到我们的工作队列，因为排在队列中等待执行的任务可能会消耗太多的系统资源并引起资源缺乏。在这种情形下决定如何做取决于您自己；在某些情况下，您可以简单地抛弃请求，依靠更高级别的协议稍后重试请求，您也可以用一个指出服务器暂时很忙的响应来拒绝请求。

线程风险解决：

## 1.17 Jdk8新特性

1. 代码更少（增加了新语法：Lambda 表达式）

2. 强大的 Stream API（集合数据的操作）

3. 最大化的减少空指针 异常：Optional 类 的使用

4. 接口的新特性;默认方法实现。

5. 注解的新特性 ;Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。

6. 集合的底层 源码实现

7. 新日期时间的 api LocalDateTime

## 1.18 JAVA中的4种访问权限



## 1.19 红黑树

## 1.20 一亿数据查询特定用户

http://blog.itpub.net/69959420/viewspace-2704837/

.数据孤岛：电商部门的数据存在数仓A、门店经营收入数据存在数仓B，如何便捷的进行多仓联合分析？

.PB级数据量：多电商平台+全国线下门店每天将产生TB级数据量，年数据量高达PB级！

## 1.21 GC

Minor GC(年轻代)触发机制：

当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。

Full GC（老年代）触发机制：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、survivor space1（From Space）区向survivor space2（To Space）区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

 Major GC（老年代） 是清理永久代

## 1.22 一个简单的链表结构实现。

## 1.23 死锁产生与避免

**死锁**：线程A持有对象X的锁等待对象Y的锁。而线程B持有对象Y的锁却等待对象X的锁。

**死锁避免**

1. 加锁顺序：确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。
2. 加锁时限：尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(译者注：加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑)。
3. **死锁检测：**是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。将线程和锁相关信息记录下来。产生死锁进行回退并释放所有锁。过一段时间再随机重试。

## 1.23 servlet与webservice协议

## 1.24 分布式锁（包含java单机锁）

**单机锁（多环境没有效果）**

1 synchronized关键字修饰的方法或者代码块。

2 使用特殊域变量(volatile)实现线程同步，但是不是原子性的，可能不同步。

3 ReenreantLock类的常用方法有：  
         ReentrantLock() : 创建一个ReentrantLock实例   
         lock() : 获得锁   
         unlock() : 释放锁

4 ThreadLocal 类的常用方法

     ThreadLocal() : 创建一个线程本地变量

    get() : 返回此线程局部变量的当前线程副本中的值

    initialValue() : 返回此线程局部变量的当前线程的"初始值"

    set(T value) : 将此线程局部变量的当前线程副本中的值设置为value

注：ThreadLocal与同步机制

        a.ThreadLocal与同步机制都是为了解决多线程中相同变量的访问冲突问题。

        b.前者采用以"空间换时间"的方法，后者采用以"时间换空间"的方式

**分布式锁：**

**Setnx 没有就写入key。key存在就不再写入，返回（已存在返回0 false，不更新。不存在返回1 true，做写入）**

**127.0.0.1:6379> setnx lockkey 666**

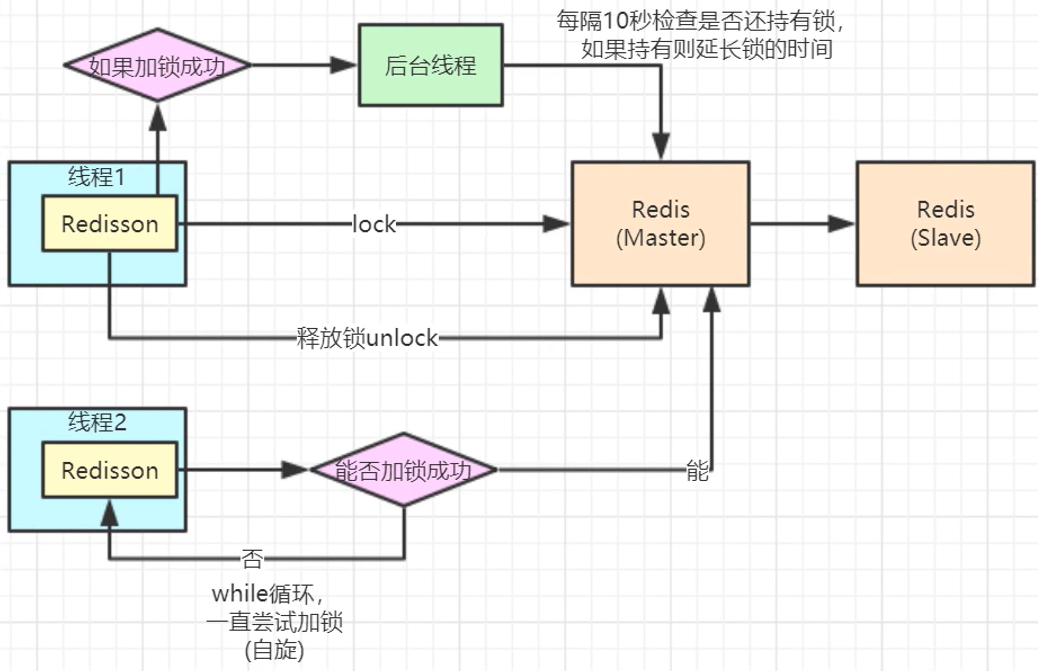
**(integer) 1**

**127.0.0.1:6379> setnx lockkey 777**

**(integer) 0**

**127.0.0.1:6379> get lockkey**

**"666"**



package com.zlk.redis.redisson;  
  
import org.redisson.Redisson;  
import org.redisson.api.RLock;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.StringRedisTemplate;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.UUID;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 测试分布式锁  
 \*/*@RestController  
@RequestMapping("/redisson")  
public class RedissonController {  
 @Autowired  
 private StringRedisTemplate redisTemplate;  
 @Autowired  
 private Redisson redisson;  
  
  
 */\*\*  
 \* synchronized非单系统会出现问题  
 \*/* @RequestMapping("/lock")  
 public String lock(){  
 //不加synchronized，并发大时stock会出现线程问题。数量不对。  
 // 但是synchronized针对单服务有效。对分布式系统由于JVM有多个。此时将不能使用  
 synchronized (this){  
 //库存  
 int stock = Integer.*parseInt*(redisTemplate.opsForValue().get("stock"));  
 if(stock>0){  
 redisTemplate.opsForValue().set("stock",--stock+"");  
 System.*out*.println("扣除成功，剩余库存："+stock);  
 }else{  
 System.*out*.println("库存不足！");  
 }  
 }  
 return null;  
 }  
  
 */\*\*  
 \* 分布式锁  
 \*/* @RequestMapping("/lock1")  
 public String lock1(){  
 String lockKey = "product\_001";  
 String uuid = UUID.*randomUUID*().toString();  
 try{  
 //setnx 不存在能添加成功返回true。已存在key不能修改，返回false。  
 // 不设置失效时间可能导致锁不能释放（如刚好重启而锁没有释放）.  
 // 但是可能方法没执行完导致锁先释放出问题。极端情况会刚加锁就被的线释放掉。加一个唯一标识UUID,自己的允许删除  
 Boolean result = redisTemplate.opsForValue().setIfAbsent(lockKey, uuid,10, TimeUnit.*SECONDS*);  
 if (!result){  
 //锁存在，别的线程在此返回。等锁解开才能往下执行  
 //为了使锁不在方法没结束失效。可以定时器延迟时间 。但是很复杂。不推荐  
 return "error";  
 }  
 //查询库存  
 int stock = Integer.*parseInt*(redisTemplate.opsForValue().get("stock"));  
 if(stock>0){  
 redisTemplate.opsForValue().set("stock",--stock+"");  
 System.*out*.println("扣除成功，剩余库存："+stock);  
 }else{  
 System.*out*.println("库存不足！");  
 }  
 }finally {  
 //当前线程的锁只允许当前删除  
 if(uuid.equals(redisTemplate.opsForValue().get(lockKey)){  
 //业务完成或者 报异常需要释放分布式锁。  
 redisTemplate.delete(lockKey);  
 }  
 }  
 return null;  
 }  
  
 */\*\*  
 \* 分布式锁redisson  
 \* 改进可以使用分段锁  
 \*/* @RequestMapping("/lock2")  
 public String lock2(){  
 String lockKey = "product\_001";  
 //String uuid = UUID.randomUUID().toString();  
 //redisson 每过10秒（1/3）会重置锁时间  
 RLock lock = redisson.getLock(lockKey);  
 try{  
 /\* //setnx 不存在能添加成功返回true。已存在key不能修改，返回false。  
 // 不设置失效时间可能导致锁不能释放（如刚好重启而锁没有释放）.  
 // 但是可能方法没执行完导致锁先释放出问题。极端情况会刚加锁就被的线释放掉。加一个唯一标识UUID,自己的允许删除  
 Boolean result = redisTemplate.opsForValue().setIfAbsent(lockKey, uuid,10, TimeUnit.SECONDS);  
 if (!result){  
 //锁存在，别的线程在此返回。等锁解开才能往下执行  
 //为了使锁不在方法没结束失效。可以定时器延迟时间 。但是很复杂。不推荐  
 return "error";  
 }\*/  
 //除当前加锁成功的锁，别的锁会阻塞。会导致性能问题  
 //可以考虑使用库存按一定量分成多段实现。（分段锁-hashmap形式）。  
 // 主从来不及复制会出现一些问题（可以考虑zookpeer）  
 lock.lock(30,TimeUnit.*SECONDS*);  
 //查询库存  
 int stock = Integer.*parseInt*(redisTemplate.opsForValue().get("stock"));  
 if(stock>0){  
 redisTemplate.opsForValue().set("stock",--stock+"");  
 System.*out*.println("扣除成功，剩余库存："+stock);  
 }else{  
 System.*out*.println("库存不足！");  
 }  
 }finally {  
 /\* //当前线程的锁只允许当前删除  
 if(uuid.equals(redisTemplate.opsForValue().get(lockKey)){  
 //业务完成或者 报异常需要释放分布式锁。  
 redisTemplate.delete(lockKey);  
 }\*/  
 //业务完成或者 报异常需要释放分布式锁。  
 lock.unlock();  
 }  
 return null;  
 }  
  
}

## 1.24 分布式事务

## 1.25 类加载器与双亲委派模型

1）类加载7个阶段：

加载：通过一个类的全限定名来获取定义此类的二进制字节流，生成class对象。

验证：确保Class文件字节流中包含的信息符合当前虚拟机的要求，并且不会威胁虚拟机自身的安全。

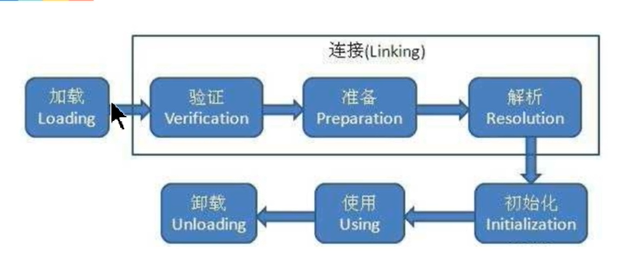
准备：正式为类实例变量分配内存并且设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配

解析：虚拟机将常量池内的符号引用替换为直接引用的过程。

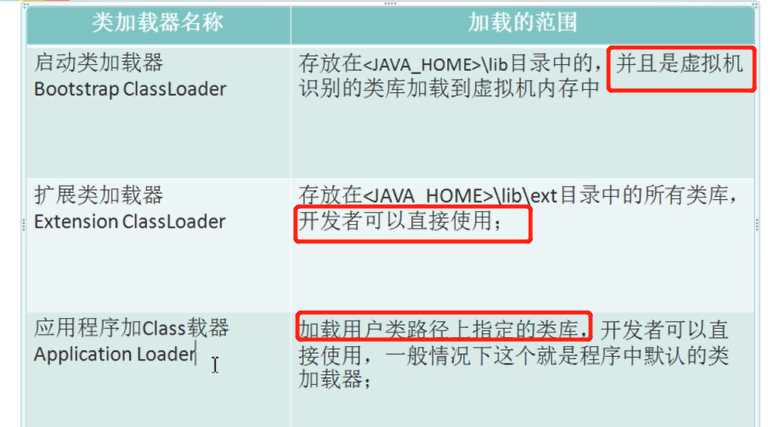
初始化：初始化init与static方法。

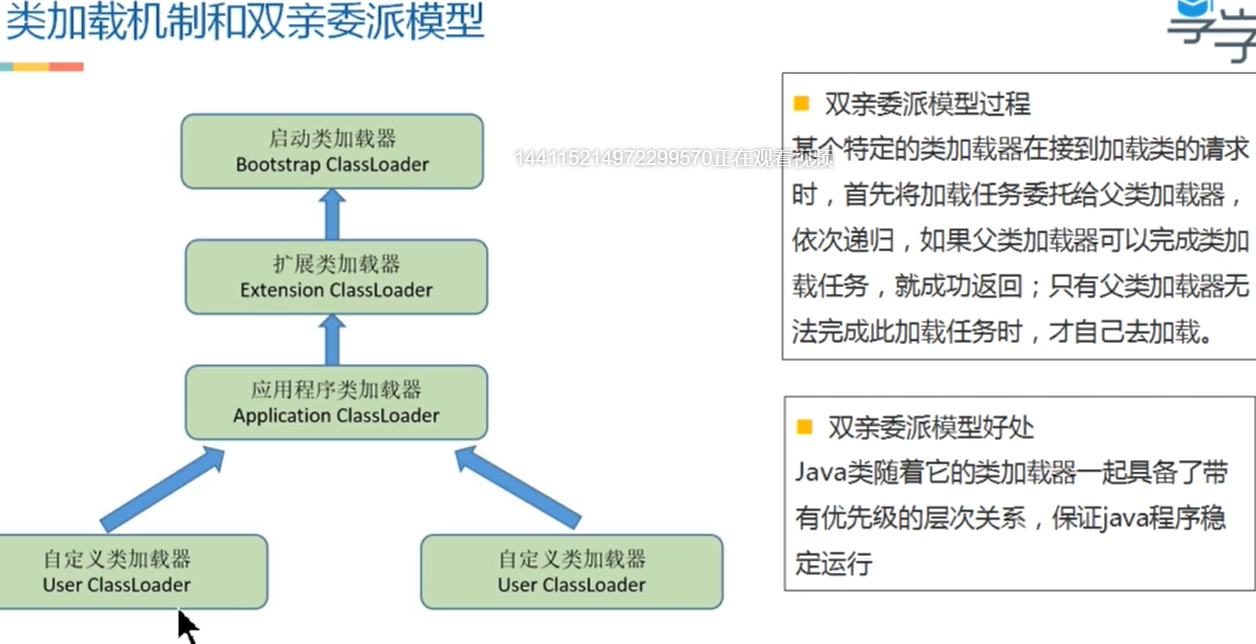
使用：new对象的使用。

卸载：



2）常用加载器





都统一先交给父类加载器加载。保证稳定。

## 1.26 事务隔离级别和传播属性

第一个为默认属性：事务会传递（如果没事务，会新建）。



## 1.27 高并发下安全修改同一行

1）JVM

上锁synchronized ()或者lock()锁，但是lock需要释放

但是不能解决分布式环境下的锁

1. 分布式锁
2. 数据库方式： 存在性能问题
3. Redis: redission 性能好，但是会出现死锁
4. Zookeeper:

基于异常：实现简单，性能也不错（使用的临时节点，不带序号）

基于相互监听：实现难，性能高，占用资源（使用临时带序号节点）；序号类似于排队，只需要监听前一个节点。节点会较多。

## 1.28 synchronized与lock区别

## 1.29 线程加入、停止、等待。

1.30 锁（单机锁，分布式锁，乐观锁，悲观锁等）

# 2 设计模式

1 单例

2工厂

............

# 3 JAVA框架

## 什么是spring

## 3.1 SpringMVC原理。

第1步：浏览器发送指定的请求都会交给DispatcherServlet,他会委托其他模块进行真正的业务和数据处理

第2步：DispatcherServlet会查找到HandleMapping,根据浏览器的请求找到对应的Controller，并将请求交给目标Controller

第3步：目标Controller处理完业务后，返回一个ModelAndView给DispatcherServlet

第4步：DispatcherServlet通过ViewResolver视图解析器找到对应的视图对象View

第5步：视图对象View负责渲染，并返回到浏览器

## 3.2 Spring常用注解与区别。

<https://www.jianshu.com/p/06e3dd0ff74b>

@Configuration把一个类作为一个IoC容器，它的某个方法头上如果注册了@Bean，就会作为这个Spring容器中的Bean。

@ComponentScan 用于对Component进行扫描，相当于xml中的（类上）

@Repository、@Service、@Controller 和 @Component 将类标识为Bean。

 Spring2.5为我们引入了组件自动扫描机制，他可以在类路径底下寻找标注了@Component,@Service,@Controller,@Repository注解的类，并把这些类纳入进spring容器中管理。它的作用和在xml文件中使用bean节点配置组件时一样的。

@Component 是一个泛化的概念，仅仅表示一个组件 (Bean) ，可以作用在任何层次。

@Service 通常作用在业务层，但是目前该功能与 @Component 相同。

@Constroller 通常作用在控制层，但是目前该功能与 @Component 相同。

通过在类上使用 @Repository、@Component、@Service 和 @Constroller 注解，Spring会自动创建相应的 BeanDefinition 对象

@Autowried和@Resource

1、@Autowried

用来装配bean, 可作用于字段上, 也可以作用于setter方法上.

是Spring的注解.

默认情况下要求对象必须存在, 它要求依赖对象必须存在. 若允许null值, 可以设置它的required为false.

默认**按照类型byType**进行装配注入. 如果想按照名称进行装配的话, 需要与Qualifer注解搭配使用

2、@Resource

用来装配bean, 可作用于字段上, 也可以作用于setter方法上.

是J2EE的注解.

默认按照名称来装配注入, 只有找不到与名称匹配的bean才会按照类型来注入.

它有两个属性是比较重要的:

name: Spring将name的属性值解析为bean的名称, 使用byName的自动注入策略

type: Spring将type的属性值解析为bean的类型, 使用byType的自动注入策略，如果既不指定name属性又不指定type属性, Spring这时通过反射机制默认使用byName自动注入策略

 以下这三种方式都可以成功注入：

//通过name属性注入

@Resource(name = "RedisTemplate")

private RedisTemplate redisTemplate;

//通过type属性注入

@Resource(type = RedisTemplate.class)

private RedisTemplate redisTemplate;

//通过name，type属性注入

@Resource(name = "RedisTemplate",type = RedisTemplate.class)

## 3.3 Springboot优缺点。

## 3.4 IOC与Aop(包含实现原理)。

**IOC（Inverse of Contro）控制反转**，有时候也被称为DI依赖注入，它是一种降低对象耦合关系的一种设计思想。（如工厂方法模式，在xml中配置，指定名称注入，对象组件初始化和调用都由容器负责。）

优点：

1）通过IOC容器，开发人员不需要关注对象是如何创建的，同时增加新类也非常方便，只需要修改配置文件即可实现对象的热插拔。

2）IOC容器可以通过配置文件来确定需要注入的实例化对象，因此非常便于进行单元测试。

缺点：

3）对象是通过反射机制实例化出来的，因此对系统的性能有一定的影响。

4）创建对象的流程变得复杂。

**AOP面向切面编程（动态代理加反射）**<https://www.jianshu.com/p/22cf46235d75>，是面向对象开发的一种补充，它允许开发人员在不改变原来模型的以满足新的需求，例如，开发人员可以在不改变原来业务逻辑模型的基础可以进行动态的增加日志，权限或异常处理功能。

**AOP应用场景**

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交关闭事务 ）

事实上AOP也确实是按照这个类似的思路去实现的，先来了解下AOP包含的几个概念

* Jointpoint（连接点）：具体的切面点点抽象概念，可以是在字段、方法上，Spring中具体表现形式是PointCut（切入点），仅作用在方法上。
* Advice（通知）: 在连接点进行的具体操作，如何进行增强处理的，分为前置、后置、异常、最终、环绕五种情况。
* 目标对象：被AOP框架进行增强处理的对象，也被称为被增强的对象。
* AOP代理：AOP框架创建的对象，简单的说，代理就是对目标对象的加强。Spring中的AOP代理可以是JDK动态代理，也可以是CGLIB代理。
* Weaving（织入）：将增强处理添加到目标对象中，创建一个被增强的对象的过程

**总结为一句话就是：在目标对象（target object）的某些方法（jointpoint）添加不同种类的操作（通知、增强操处理），最后通过某些方法（weaving、织入操作）实现一个新的代理目标对象。**

## 3.5 spring事务级别

Spring事务隔离级别比数据库事务隔离级别多一个default，具体由数据库决定，一般数据库默认的事务隔离级别为可重复读。

1. DEFAULT （默认）  
   这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与JDBC的隔离级别相对应。  
   2) READ\_UNCOMMITTED （读未提交）--  
   这是事务最低的隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。  
   3) READ\_COMMITTED （读已提交）  
   保证一个事务修改的数据提交后才能被另外一个事务读取，另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。  
   4) REPEATABLE\_READ （可重复读）  
   这种事务隔离级别可以防止脏读、不可重复读，但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了不可重复读。

--幻读：指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的数据行  
5) SERIALIZABLE（串行化）  
这是花费最高代价但是最可靠的事务隔离级别，事务被处理为顺序执行。除了防止脏读、不可重复读外，还避免了幻像读。

## 3.6 Mybatis一级缓存与二级缓存

<https://www.cnblogs.com/happyflyingpig/p/7739749.html>

## **一级缓存**

Mybatis对缓存提供支持，但是在没有配置的默认情况下，它只开启一级缓存，一级缓存只是相对于同一个SqlSession而言。所以在参数和SQL完全一样的情况下，我们使用同一个SqlSession对象调用一个Mapper方法，往往只执行一次SQL，因为使用SelSession第一次查询后，MyBatis会将其放在缓存中，以后再查询的时候，如果没有声明需要刷新，并且缓存没有超时的情况下，SqlSession都会取出当前缓存的数据，而不会再次发送SQL到数据库。

但sqlSession执行commit，即增删改操作时会清空缓存。这么做的目的是避免脏读。

既然有了一级缓存，那么为什么要提供二级缓存呢？

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。二级缓存的作用范围更大。

还有一个原因，实际开发中，MyBatis通常和Spring进行整合开发。Spring将事务放到Service中管理，对于每一个service中的sqlsession是不同的，这是通过mybatis-spring中的org.mybatis.spring.mapper.MapperScannerConfigurer创建sqlsession自动注入到service中的。 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的。

## **二级缓存：**

　　MyBatis的二级缓存是Application级别的缓存，它可以提高对数据库查询的效率，以提高应用的性能。

SqlSessionFactory层面上的二级缓存默认是不开启的，二级缓存的开席需要进行配置，实现二级缓存的时候，MyBatis要求返回的POJO必须是可序列化的。 也就是要求实现Serializable接口，配置方法很简单，只需要在映射XML文件配置就可以

开启缓存了<cache/>，如果我们配置了二级缓存就意味着

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

前面我们说到，Spring和MyBatis整合时， 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的。那么如果开启二级缓存，关闭sqlsession后，会把该sqlsession一级缓存中的数据添加到namespace的二级缓存中。这样，缓存在sqlsession关闭之后依然存在。

**总结：**

对于查询多commit少且用户对查询结果实时性要求不高，此时采用mybatis二级缓存技术降低数据库访问量，提高访问速度。

但不能滥用二级缓存，二级缓存也有很多弊端，从MyBatis默认二级缓存是关闭的就可以看出来。

二级缓存是建立在同一个namespace下的，如果对表的操作查询可能有多个namespace，那么得到的数据就是错误的。

3.7 Mybatis #与$

#{} 占位符，对应变量会加入单引号。能防止sql注入。

${} 拼接符，对应变量不会加入单引号，不能防止sql注入。

3.7 Mybatis 实现

# 4 关系型数据库

## 4.1 数据库优化（主要为数据库设计与sql优化）

## 4.2 底层存储。

## 4.3 索引创建 注意与优化。

CREATE INDEX index\_name ON table\_name (column\_list)

在mysql中使用索引的原则有以下几点：

1、 对于查询频率高的字段创建索引；

2、 对排序、分组、联合查询频率高的字段创建索引；

3、 索引的数目不宜太多

原因：a、每创建一个索引都会占用相应的物理控件；

　　 b、过多的索引会导致insert、update、delete语句的执行效率降低；

4、若在实际中，需要将多个列设置索引时，可以采用多列索引

如：某个表(假设表名为Student)，存在多个字段(StudentNo, StudentName, Sex, Address, Phone, BirthDate)，其中需要对StudentNo,StudentName字段进行查询，对Sex字段进行分组，对BirthDate字段进行排序，此时可以创建多列索引

index index\_name (StudentNo, StudentName, Sex, BirthDate); #index\_name为索引名

在上面的语句中只创建了一个索引，但是对4个字段都赋予了索引的功能。

创建多列索引，需要遵循BTree类型，

即第一列使用时，才启用索引。

在上面的创建语句中，只有mysql语句在使用到StudentNo字段时，索引才会被启用。

如: select \* from Student where StudentNo = 1000; #使用到了StudentNo字段，索引被启用。

可以使用explain检测索引是否被启用

如:explain select \* from Student where StudentNo = 1000;

5、选择唯一性索引

唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。例如，学生表中学号是具有唯一性的字段。为该字段建立唯一性索引可以很快的确定某个学生的信息。如果使用姓名的话，可能存在同名现象，从而降低查询速度。

6、尽量使用数据量少的索引

如果索引的值很长，那么查询的速度会受到影响。例如，对一个CHAR(100)类型的字段进行全文检索需要的时间肯定要比对CHAR(10)类型的字段需要的时间要多。

7、尽量使用前缀来索引--ALTER table 表名 add index title\_pre(列名(100)) 100个前缀

如果索引字段的值很长，最好使用值的前缀来索引。例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间。如果只检索字段的前面的若干个字符，这样可以提高检索速度。

8、删除不再使用或者很少使用的索引.

表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。数据库管理员应当定期找出这些索引，将它们删除，从而减少索引对更新操作的影响

## 4.4 索引失效情况。

1.如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)

　　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

　　2.对于多列索引，不是使用的第一部分，则不会使用索引

3.like查询是以%开头

****4.如果列类型是字符串，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****

****5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引****

****此外，查看索引的使用情况  
show status like ‘Handler\_read%';  
大家可以注意：  
handler\_read\_key:这个值越高越好，越高表示使用索引查询到的次数  
handler\_read\_rnd\_next:这个值越高，说明查询低效****

****1) 没有查询条件，或者查询条件没有建立索引****

****2) 在查询条件上没有使用引导列****

****3) 查询的数量是大表的大部分，应该是30％以上。****

****4) 索引本身失效****

****5) 查询条件使用函数在索引列上，或者对索引列进行运算，运算包括(+，-，\*，/，! 等) 错误的例子：select \* from test where id-1=9; 正确的例子：select \* from test where id=10;****

****6) 对小表查询****

****7) 提示不使用索引****

****8) 统计数据不真实****

****9) CBO计算走索引花费过大的情况。其实也包含了上面的情况，这里指的是表占有的block要比索引小。****

****10)隐式转换导致索引失效.这一点应当引起重视.也是开发中经常会犯的错误. 由于表的字段tu\_mdn定义为varchar2(20),但在查询时把该字段作为number类型以where条件传给Oracle,这样会导致索引失效. 错误的例子：select \* from test where tu\_mdn=13333333333; 正确的例子：select \* from test where tu\_mdn='13333333333';****

****12)** 在MYSQL使用不等于（<,>,!=）的时候无法使用索引，会导致索引失效。**

****13）like "%\_" 百分号在前.****

****4,表没分析.****

****15,单独引用复合索引里非第一位置的索引列.****

****16,字符型字段为数字时在where条件里不添加引号.****

****17,对索引列进行运算.需要建立函数索引.****

运算如+，-，\*，/等，如下：

|  |  |
| --- | --- |
| 1 | **select** \* **from** `user` **where** age - 1 = 10; |

优化的话，要把运算放在值上，或者在应用程序中直接算好，比如：

|  |  |
| --- | --- |
| 1 | **select** \* **from** `user` **where** age = 10 - 1; |

****18,not in ,not exist.****

****19,当变量采用的是times变量，而表的字段采用的是date变量时.或相反情况。****

****20,B-tree索引 is null不会走,is not null会走,位图索引 is null,is not null 都会走****

****21,联合索引 is not null 只要在建立的索引列（不分先后）都会走, in null时 必须要和建立索引第一列一起使用,当建立索引第一位置条件是is null 时,其他建立索引的列可以是is null（但必须在所有列 都满足is null的时候）,或者=一个值； 当建立索引的第一位置是=一个值时,其他索引列可以是任何情况（包括is null =一个值）,以上两种情况索引都会走。其他情况不会走。****

## 4.5 索引类型区别。

<https://www.cnblogs.com/s-b-b/p/8334593.html>

普通索引、唯一索引、主键索引、组合索引、全文索引。

-- 主键（聚集）索引，非聚集索引（非主键索引）

聚集（clustered）索引，也叫聚簇索引。

定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

非聚集（unclustered）索引。

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。（查询中出现未索引列需要二次查询）

## 4.6 事务特性

原子性 （atomicity）:强调事务的不可分割.

一致性 （consistency）:事务的执行的前后数据的完整性保持一致.

隔离性 （isolation）:一个事务执行的过程中,不应该受到其他事务的干扰

持久性（durability） :事务一旦结束,数据就持久到数据库

**、事务的并发问题**

**1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据**

**2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。**

可重复读：重复读的时候，当前会话可以重复读，就是每次读取的结果集都相同，而不管其他事务有没有提交。（乐观锁实现）。

**3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。**

**小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表**

三、MySQL事务隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 不可重复读（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

Spring事务隔离级别比数据库事务隔离级别多一个default，具体由数据库决定，一般数据库默认的事务隔离级别为可重复读。

1. DEFAULT （默认）  
   这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与JDBC的隔离级别相对应。  
   2) READ\_UNCOMMITTED （读未提交）--  
   这是事务最低的隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。  
   3) READ\_COMMITTED （读已提交）  
   保证一个事务修改的数据提交后才能被另外一个事务读取，另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。  
   4) REPEATABLE\_READ （可重复读）

<https://blog.csdn.net/u014801432/article/details/81219205>  
这种事务隔离级别可以防止脏读、不可重复读，但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了不可重复读。

--幻读：指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的数据行  
5) SERIALIZABLE（串行化）  
这是花费最高代价但是最可靠的事务隔离级别，事务被处理为顺序执行。除了防止脏读、不可重复读外，还避免了幻像读。

## 4.7 char、varchar、varchar2区别

Mysql 5中  
非空CHAR的最大总长度是255【字节】；非空VARCHAR的最大总长度是65533【字节】。  
可空CHAR的最大总长度是254【字节】；可空VARCHAR的最大总长度是65532【字节】。  
原因：非空标记需要占据一个字节，VARCHAR超过255需要用2个字节标记字段长度，不超过255用1个字节标记字段长度.

char是定长的，varchar是变长的。varchar2（在ORACLE varchar对于汉字占两个字节，对于英文是一个字节，占的内存小，varchar2都是占两个字节。varchar对空串不处理，varchar2将空串当做null来处理。）应该是varchar的升级，只有ORACLE才有，这里不作讨论。

char 定长存储，速度快，但是存在一定的空间浪费，适用于字段不是很大，对速度要求高的场合。速度快是因为其在物理上是按定长存储的，这样，就可以根据偏移址一次取出固定长度的字符。

varchar 变长存储，效率不如char。varchar在存储时，在物理上要先存储该字段的实际长度，然后才是内容。这样读取的时候，就要读取两次，一次读它的长度，然后才是内容。所以它的访问速度会比char慢一些。但它可以节省空间。

## 4.8 执行计划

explain显示了MySQL如何使用索引来处理select语句以及连接表。他可以帮助选择更好的索引和写出更优化的查询语句

explain显示了很多列，各个关键字的含义如下：

table：顾名思义，显示这一行的数据是关于哪张表的；

type：这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为：const、eq\_reg、ref、range、indexhe和ALL；

possible\_keys：显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从where语句中选择一个合适的语句；

key： 实际使用的索引。如果为NULL，则没有使用索引。很少的情况下，MySQL会选择优化不足的索引。这种情况下，可以在Select语句中使用USE INDEX（indexname）来强制使用一个索引或者用IGNORE INDEX（indexname）来强制MySQL忽略索引；

key\_len：使用的索引的长度。在不损失精确性的情况下，长度越短越好；

ref：显示索引的哪一列被使用了，如果可能的话，是一个常数；

rows：MySQL认为必须检查的用来返回请求数据的行数；

Extra：关于MySQL如何解析查询的额外信息。

## 4.9 mysql 存储引擎

****InnoDB存储引擎：****InnoDB是**事务型数据库的首选引擎**，支持事务安全表（ACID），支持行锁定和外键，InnoDB是默认的MySQL引擎

****MyISAM存储引擎：****MyISAM基于ISAM存储引擎，并对其进行扩展。它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。MyISAM拥有较高的插入、查询速度，但****不支持事物****。

**Memory：**将数据存在内存，为了提高数据的访问速度，每一个表实际上和一个磁盘文件关联。文件是frm。

## 4.10 mysql 索引结构

<https://www.cnblogs.com/s-b-b/p/8334593.html>

SQL的主流索引结构有B+树以及Hash结构，聚集索引以及非聚集索引用的是B+树索引。

## 4.11 数据库范式

**第一范式：每个字段已经不可再分解**  
      
    第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。  
    第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，  
  
第二范式(确保表中的每列都和主键相关)  
 组合主键，数据应该与组合主键直接相关。拆分数据，让数据不冗余。  
   第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。  
    比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键  
    这样就产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。  
    而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。  
第三范式(确保每列都和主键列直接相关,而不是间接相关)  
 外键，主从表，需要直接与主键相关。  
    第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。  
    比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。

## 4.12悲观锁与乐观锁

**悲观锁**

每次拿数据都以为别人会修改，所以每次拿数据时都会上锁。  
实现：开启事务，启用锁机制

**乐观锁**

每次拿数据时候都认为别人不会修改，所以不会上锁，但是在更新数据时候会判断在此期间是否有人更新过。  
 实现：1.使用版本号2.使用时间戳

## 4.13 B树与hash

索引大多采用B/B+树来作为存储结构，而全文搜索引擎的索引则主要采用hash的存储结构。

hash结构的特点：检索效率非常高，索引的检索可以一次到位，O(1)。B树需要从根节点到枝节点，最后才能到叶节点进行多次I/O操作，所以hash的效率远远高于B树的效率。

**那么为什么数据库索引还是用B树结构呢？**

1、hash索引仅满足“=”、“IN”和“<=>”查询，不能使用范围查询

因为hash索引比较的是经常hash运算之后的hash值，因此只能进行等值的过滤，不能基于范围的查找，因为经过hash算法处理后的hash值的大小关系，并不能保证与处理前的hash大小关系对应。

2、hash索引无法被用来进行数据的排序操作

由于hash索引中存放的都是经过hash计算之后的值，而hash值的大小关系不一定与hash计算之前的值一样，所以数据库无法利用hash索引中的值进行排序操作。

3、对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

4、Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。

对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

（因此：键值重复率低的适合用B树索引）

hash相当于把key通过hash函数计算，得到key的hash值,再用这个hash值做指针，查找hash表中是否存在key，如果存在就返回 key所对应的value，选定一个好的hash函数很重要，好的hash函数可以使计算出的hash值分布均匀，降低冲突，只有冲突减小了，才会降低 hash表的查找时间。

b-tree完全基于key的比较，和二叉树相同的道理，相当于建个排序后的数据集，使用二分法查找算法，实际上也非常快，而且受数据量增长影响非常小。

哈希索引一般适用于：不需要做**排序**、**范围查询的需求。就是说只能做等值查询。**

## 4.14索引结构的底层实现

**索引**（Index）是帮助MySQL高效获取数据的数据结构。

**索引作用：**

创建索引可以大大提高系统的性能。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。

**实现：**

  数据在磁盘上是以块的形式存储的，磁盘上的这些数据块与链表类似。直接查找需要遍历所有空间。而建立索引会排序。此时可以使用二分查找调高效率（B树就是一个排序的二叉树）。

## 4.15 分库分表

## 4.16 MVCC

## 4.17 各种优化

## 4.18 集群

# 5 非关系型数据库

## 5.1 Redis

String底层是一个位数组

### 5.1.1 Redis数据类型与区别。

<https://www.cnblogs.com/superfj/p/9232482.html>

### 5.1.2 Redis 缓存穿透、击穿、雪崩。

(1) **缓存穿透**：数据在缓存与数据库中都不存在；使用本来不存在的id去做攻击

查询会压垮数据库。

解决方案1：使用布隆过滤器 ,将所有可能存在的key值哈希到一个足够大的bitmap中(setbit)，一个一定不存在的值直接拦截掉。（将所有key持久到hash表，查询时先查key是否存在）可以考虑redis（setbit）+定时任务。不能删除，因为共享。只能重新初始话。

解决方案2：返回为空直接给你缓存一个空值（不管系统故障还是 数据不存在），但是过期时间要短，一般就5分钟。（可能造成很多无用的key）

1. **数据击穿**：数据在缓存中过期。此时若大量并发请求（一般是非常热点的数据），这些请求发现缓存失效都会从db中加载数据并设到缓存。这时候大 并发请求会把数据库压垮。

解决方案1:分布式锁做双向检索，加锁块中要先查询缓存，再查数据库。第一个入缓存成功后释放掉锁。（用**synchronized会全部锁住**）

1. **缓存雪崩**：当缓存服务器重启或者大量缓存在同一时间段失效。这样会给数据库带来很大的压力。

解决方案：1、加锁或者队列控制不会有大量线程同时读写库来缓存（重启时，由于有性能问题，高并发环境很少使用）；降级、熔断。

1. 将失效时间随机设置，减低每个缓存过期时间的重复率（使用中）。集群。

package com.zlk.redis.penetration;  
  
import com.google.common.hash.Funnels;  
import com.zlk.redis.dao.User;  
import com.zlk.redis.dao.UserDao;  
import javafx.scene.effect.BloomBuilder;  
import org.redisson.api.RLock;  
import org.redisson.api.RedissonClient;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.stereotype.Service;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 缓存穿透（数据库与缓存中都查不到）  
 \* 方式2：使用布隆过滤器的方式解决。牺牲准确率换取效率。击穿也放在这里  
 \*/*@Service  
public class CachePenetrationService1 {  
 @Autowired  
 private UserDao userDao;  
 @Autowired  
 private RedisTemplate redisTemplate;  
 @Autowired  
 private RedissonClient redisson;  
 @Autowired  
 private BloomFilterHelper bloomFilterHelper;  
 @Autowired  
 private RedisBloomFilter redisBloomFilter;  
  
 */\*\*  
 \* 使用返布隆过滤器解决缓存穿透  
 \* 查询所有（测试缓存击穿，数据库存在，缓存失效）  
 \** ***@return*** *用户列表  
 \*/* public User getUserList(String id){  
 //布隆过滤器判断key不存在直接返回（缓存穿透）  
 boolean bloomFilter = redisBloomFilter.includeByBloomFilter(bloomFilterHelper, "abc", id);  
 if(!bloomFilter){  
 //key不存在，也没有数据库对应数据。缓存穿透，直接返回。  
 System.*out*.println("缓存穿透,直接返回");  
 return null;  
 }  
 //key设置为user\_list,测试代码不考虑分页  
 User user = (User)redisTemplate.opsForValue().get(id);  
 if (user!=null){  
 System.*out*.println("1--缓存中有对应值");  
 return user;  
 }  
  
 //分布式锁，redisson保证缓存的分布式锁不过期。（解决缓存击穿）  
 RLock lock = redisson.getLock(id);  
 try{  
 //阻塞，设置时间。保证redis中可以解锁。  
 lock.lock(30,TimeUnit.*SECONDS*);  
 //再查询一次缓存，当第一次入缓存成功后，第一个线程以缓存。不再查询数据库。  
 user = (User)redisTemplate.opsForValue().get(id);  
 if (user!=null){  
 System.*out*.println("2--缓存中有对应值");  
 return user;  
 }  
 //缓存中不存在，查寻数据库  
 user = userDao.getUserById1();  
 System.*out*.println("数据库中有对应值，缓存中过期，查询数据库");  
 //设置时间30分钟失效，将数据写到redis。  
 redisTemplate.opsForValue().set(id,user,20, TimeUnit.*SECONDS*);  
 }finally {  
 //保证解锁成功  
 lock.unlock();  
 }  
 return user;  
 }  
  
// 已过时，会有容错误判（和数组长度与hash个数有关，每个hash定位到的位置标记为1）。hash会出现碰撞，内存有共享。就是牺牲准确率换取速度与内存。  
// private static BloomBuilder<Integer> blooFilter = BloomBuilder.create(Funnels.integerFunnel(),10000000,0.01);  
  
}

### 5.1.3 Redis 与数据库双写问题。

### 5.1.4 Redis主从。

### 5.1.5 Redis集群。

### 5.1.6 Redis优化。

### 5.1.7 Redis数据一致性。

(1)数据库成功,缓存失败。

(2)网络问题，先的线程后更新。

解决方法：先删除出缓存，再更新数据库。

1. 延时双删，先删除出缓存，再更新数据库。再删除缓存（使用mq重试保证成功）。
2. 串行话

## 5.2 mongodb

# 6 消息队列 （主要为kafka与RabbitMQ）

### [Kafka、ActiveMQ、RabbitMQ、RocketMQ](http://www.baidu.com/link?url=ywbAK4dHY2LtYohKmGLOA66QjTFxdFemr0OlNWG_v0j9BRTjiA5ITLFA9YgUyDMb39G8Up1wIlq81o2Z8HYzua" \t "https://www.baidu.com/_blank)

## 6.1 RabbitMQ

### 6.1.1RabbitMQ常用概念。

### 6.1.2 RabbitMQ数据持久化。

### 6.1.3 RabbitMQ集群。

### 6.1.4 RabbitMQ怎么保证数据的发送与消费成功。

### 6.1.5 RabbitMQ怎么保证数据的有序性与防治重复消费 。

### 6.1.6 RabbitMQ优化。

**6.2 kafka常用概念。**

# 7 搜索引擎

1 ES

# 8 Spring cloud

<https://blog.csdn.net/jerryDzan/article/details/89137818>

## 怎么理解微服务架构。

## 8.1 常用组件

### Spring  Cloud  **Eureka**（服务治理）:

       服务治理： 服务治理是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的自动化注册和发现。

### Spring  Cloud  **Ribbon**（客户端负载均衡）

Spring  Cloud  **Hystrix**（服务容错保护

Hystrix的断路器就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力.

Spring  Cloud  **Feign**（声明式服务调用）

    Feign是一种声明式、模板化的HTTP客户端。在Spring Cloud中使用Feign, 我们可以做到使用HTTP请求远程服务时能与调用本地方法一样的编码体验，开发者完全感知不到这      是远程方法，更感知不到这是个HTTP请求

### Spring Cloud   **Zuul**（API网关服务）:   过滤:安全、监控、限流、路由

### Spring Cloud   **Config**（分布式配置中心）

### Spring Cloud    **Bus**（消息总线）

### Spring Cloud Bus 将分布式的节点用轻量的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。

### 消息总线是一种通信工具，可以在机器之间互相传输消息、文件等。消息总线扮演着一种消息路由的角色，拥有一套完备的路由机制来决定消息传输方向。发送段只需要向消息总线发出消息而不用管消息被如何转发。Spring cloud bus 通过轻量消息代理连接各个分布的节点。管理和传播所有分布式项目中的消息，本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。

Spring Cloud    **Stream**（消息驱动微服务）:

消息驱动理解： Windows消息是，当一个窗体或者控件需要让另外一个窗体或者消息执行某个动作，就向那个窗体发一个消息，放到对方的消息队列中，然后对方有一个消息循环不停的读取消息队列，并执行相应的动作。

简单的讲就是一种生产者，消费者模式。发布者是生产，将输出发布到数据中心，订阅者是消费者，订阅自己感兴趣的数据。当有数据到达数据中心时，就把数据发送给对应的订阅者。

Spring Cloud    **Sleuth**（分布式服务跟踪）:

在微服务框架中，一个由客户端发起的请求在后端系统中会经过多个不同的的服务节点调用来协同产生最后的请求结果，每一个前段请求都会形成一条复杂的分布式服务调用链路，链路中的任何一环出现高延时或错误都会引起整个请求最后的失败。

Spring Cloud **Sleuth**提供了一套完整的服务跟踪的解决方案。

Spring  Cloud  **Shiro**（安全权限控制）:    这个是进入系统了， 然后进行处理。 Zuul这个是路由器的管理，在系统未进入之前管理。

## 8.2 分布式事务

<https://zhuanlan.zhihu.com/p/93029043>

<https://www.cnblogs.com/bluemiaomiao/p/11216380.html>

## 8.3 分布式锁

使用redis或者zk

8.4 Spring cloud与Doubbo

### 8.5 zk与Eureka

# 9 JVM

# 10 linux 常用命令

登录：ssh [root@122.112.246.108](mailto:root@122.112.246.108)

切换root用户: root su sudo -

查看目录：1.包含文件属性 ll

1. 仅仅目录与文件名称 ls

切换目录redis：cd redis

查看当前目录:**pwd**

创建目录redis：mkdir redis

强制删除目录redis：rm -rf redis

移动redis到redis1：mv redis/ redis1/

复制redis到redis1(不存在)：cp -r redis/ redis1/

解压tar.gz包 tar -zxvf jdk-8u211-linux-x64.tar.gz

解压war包jar -xvf game.war

查询jdk安装目录：which java

ls -l /usr/bin/java

ls -l /etc/alternatives/java

find 目录 -name '\*nginx\*' 查找

sudo ln -s /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el7\_8.x86\_64/jre/bin/java

/usr/bin/java

Vim

退出：Ctrl+C

上传：rz （本地上传 sudo scp 本地目录 [用户名@i](mailto:root@122.112.246.108)p: 远程目录）

sudo scp G:\git\exercise\_jvm\target\jvm-0.0.1-SNAPSHOT.jar [root@122.112.246.108:/](mailto:root@122.112.246.108/)data/ 下载：sz

查看进程

ps -ef|grep redis

ps aux|grep redis

Vim：

进入编辑:i

退出编辑:esc然后:

：q！强制退出，所有改动不生效

：wq  保存退出

查看日志：

grep "2020" -i catalina.out

tail -f catalina.out

tail -n 500 catalina.out

df -hl

du -sh \*

telnet 47.93.191.144 6378

文件

touch catalina.out

chmod 777 catalina.out

du -sh \*

> catalina.out

**linux性能查看命令：**

<https://blog.csdn.net/qq_19272431/article/details/88616236>

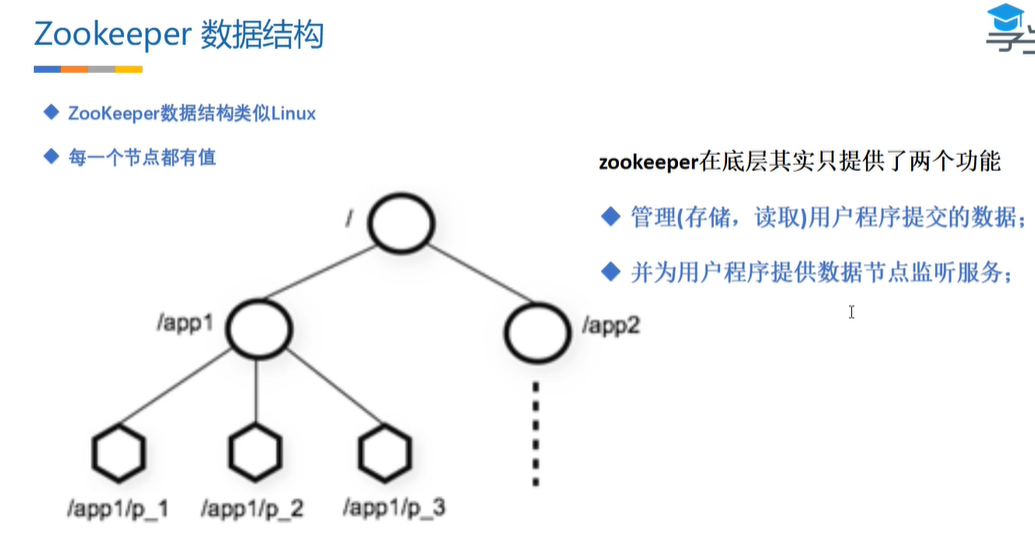
1. top

显示了CPU的使用率、内存使用率、交换内存使用大小、高速缓存使用大小、缓冲区使用大小，进程PID、所使用命令以及其他。它还可以显示正在运行进程的内存和CPU占用多的情况。对系统管理员来说，top命令式是一个非常有用的，它可用于监控系统并在需要。

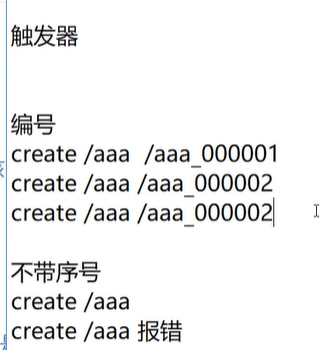
(2)

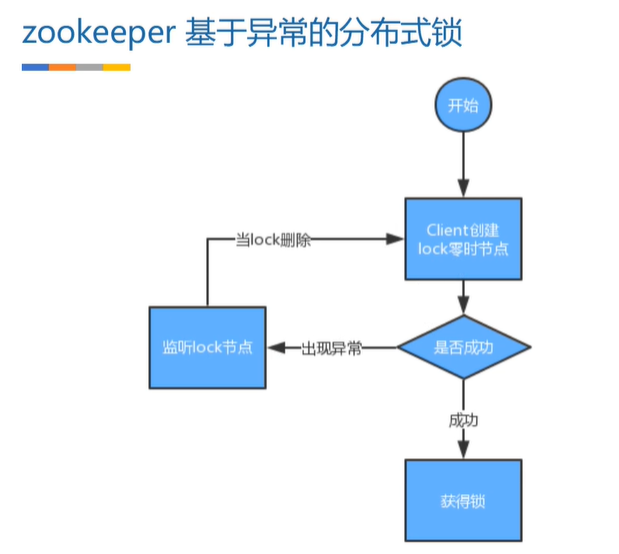
# 11 nginx

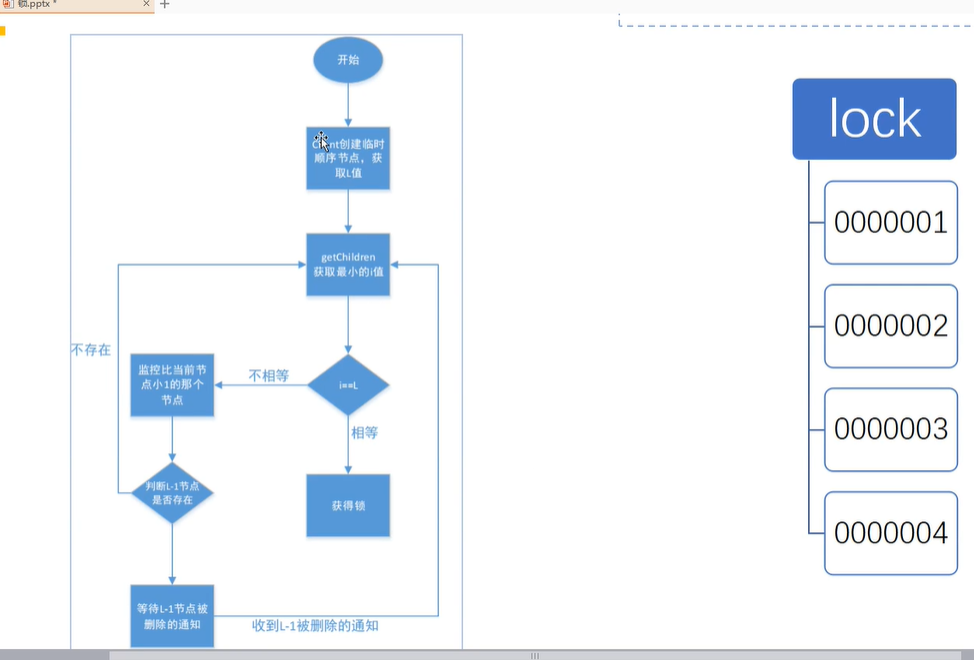
# 12 zookeeper











# 12 擅长什么，缺点，学习快的举例

# 13 项目中的难点与解决方案。

# 14 学习知识的方式。

# 15 前端

Vue的优点。

# 16 数据结构。