**JAVA基础:**

1. **JDK 和 JRE 有什么区别？**

JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。

JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。

具体来说 JDK 其实包含了 JRE，同时还包含了编译 Java 源码的编译器 Javac，还包含了很多 Java 程序调试和分析的工具。简单来说：如果你需要运行 Java 程序，只需安装 JRE 就可以了，如果你需要编写 Java 程序，需要安装 JDK。

1. **Java 内存区域**
2. **== 和 equals 的区别是什么？**

==是运算符，equal是方法

1）对于==，如果作用于基本数据类型的变量，则直接比较其存储的 “值”是否相等；如果作用于引用类型的变量，则比较的是所指向的对象的地址

2）对于equals方法，注意：equals方法不能作用于基本数据类型的变量;如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址；

1. **final 在 Java 中有什么作用？**

final 修饰的类不能被继承。  
 final 修饰的方法不能被重写。  
 final 修饰的变量叫常量，必须初始化，不能被修改。

1. **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。

1. **Java 容器都有哪些？**

Java 容器分为 Collection 和 Map 两大类，其下又有很多子类，如下所示：

Collection

List、ArrayList、LinkedList、Vector、Stack、Set、HashSet、LinkedHashSet、TreeSet

Map

HashMap、LinkedHashMap、TreeMap、ConcurrentHashMap、Hashtable

1. **数组与链表的区别**

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(n)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：   
　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；   
　　链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

1. **Map集合几种遍历方式?** keySet()、entrySet()、values()
2. entrySet

for(Map.Entry<String, String> entry : map.entrySet()){

String mapKey = entry.getKey();

String mapValue = entry.getValue();

System.out.println(mapKey+":"+mapValue);

}

1. 在for循环中遍历key或者values，区别是可以通过键找值，不能通过值找键

for(String key : map.keySet()){

System.out.println(key);

}

//value

for(String value : map.values()){

System.out.println(value);

}

1. 迭代器

Iterator<Entry<String, String>> entries = map.entrySet().iterator();

while(entries.hasNext()){

Entry<String, String> entry = entries.next();

String key = entry.getKey();

String value = entry.getValue();

System.out.println(key+":"+value);

}

1. Lambda

map.forEach((key, value) -> {

System.out.println("Key: " + key + ", Value: " + value);

});

1. **List集合的特性**

ArrayList

1.ArrayList是List接口可变数组的实现,允许存放null

2.底层是使用数组实现,无参构造函数默认初始化长度为10,数组扩容是会将原数组中的元素重新拷贝到新数组中,长度为原来的1.5+1(代扩容价高)

3.线程非同步.

LinkedList

1.LinkedList是List接口双向链表的非同步实现,允许存放null

2.底层的数据结构是基于双向链表,数据结构是节点

3.双向链表中每个节点分为prev,next,item,其中prev中存放上一个节点的信息,next存放下一个节点的信息,item存放该节点的值

1. **如何实现数组和 List 之间的转换？**

数组转 List：使用 Arrays. asList(array) 进行转换。  
 List 转数组：使用 List 自带的 toArray() 方法。

1. **Hashtable和**ConcurrentHashMap

Hashtable通过使用synchronized修饰方法的方式来实现多线程同步，因此，Hashtable的同步会锁住整个数组。在高并发的情况下，性能会非常差，Java5中引入了java.util.concurrent.ConcurrentHashMap作为高吞吐量的线程安全HashMap实现，它采用了锁分离的技术允许多个修改操作并发进行。

1. **ArrayList 和 LinkedList 有什么区别**

ArrayList和LinkedList都实现了List接口，有以下的不同点：  
 1、ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。  
 2、相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。  
3、LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

1. **List、set和map的区别**

list和set是实现了collection接口的。

List：1.可以允许重复的对象。

　　 2.可以插入多个null元素。

    3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Set： 1.不允许重复对象

　　 2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator  或者 Comparable 维护了一个排序顺序。

    3. 只允许一个 null 元素

4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

Map不是collection的子接口或者实现类。Map是一个接口。

1.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

2. TreeMap 也通过 Comparator  或者 Comparable 维护了一个排序顺序。

3.Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

4.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

1. **HashMap的特性**

* Map集合的特点

1. Map是一个双列集合,将键映射到值的对象.
2. Map集合的数据结构,只针对键有效,跟值没关系.
3. 一个映射不能包含重复的键,每个键最多只能映射一个值.

* HashMap的数据结构

1. 哈希表结构:数组+链表
2. 通过哈希表结构配合对象的hashCode和equals方法就可以确保键的唯一性.

* HashMap和Hashtable的区别

1. HashMap是jdk1.2版本出现的,允许存储null键和null值

不同步(线程不安全):效率高

1. Hashtable是jdk1.0版本出现的,不允许存储null键和null值

同步(线程安全):效率低

1. **HashMap 的实现原理？**

HashMap 基于 Hash 算法实现的，我们通过 put(key,value)存储，get(key)来获取。当传入 key 时，HashMap 会根据 key. hashCode() 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash 冲突的个数比较少时，使用链表否则使用红黑树

1. **HashSet 的实现原理**

HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。

1. **CurrentHashMap**

HashMap是线程不安全的 , 但是效率高 , HashTable是线程安全的 , 但是效率低.有没有一种对象是即是线程安全的 , 同时执行效率可以达到HashMap呢?

CurrentHashMap可以做到。底层实现通过分段加锁进行实现 , hashmap底层是数组加上链表实现的 , 那么一个线程来操作数据,只是操作数组中一个索引的数据. 如果此时对整个数组加锁,其他线程操作不了这个数组,所以效率低.其实线程也就操作数组的一个索引,对这个索引进行加锁 , 而锁对象就是这个索引所对应的值,其他线程来修改其他索引数据时,拿到的是其他索引的锁对象,从而提高了效率.

1. **String,StringBuilder,StringBuffer三者的区别**

* String和StringBuilder的本质区别

String是一个不可改变的字符序列.

StringBuilder是一个可以改变的字符序列.

* 常见的字符拼接,该选择谁

推荐使用StringBuilder,因为拼接的效率高

* StringBuilder和StringBuffer的区别

1. StringBuilder和StringBuffer的功能是完全一致的.
2. 不同点

StringBuffer是jdk1.0出现的,线程安全(同步):效率低.

StringBuilder是jdk1.5出现的,线程不安全(不同步):效率高.

1. **抽象类与接口区别**

* 一个类可以实现多个接口，但只能继承一个抽象类。
* 抽象类中可以有普通成员变量，接口中没有普通成员变量
* 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型
* 抽象类可以有构造方法，接口中不能有构造方法
* 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的
* 抽象类中的抽象方法的访问类型可以是public，protected，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型
* 抽象类中可以包含静态方法，接口中不能包含静态方法
* 接口关注的是功能，抽象类关注的抽象现实中事物

1. **类的实例化方法调用顺序**

此题考察的是类加载器实例化时进行的操作步骤（加载–>连接->初始化）。  
 父类静态变量、  
 父类静态代码块、  
 子类静态变量、  
 子类静态代码块、  
 父类非静态变量（父类实例成员变量）、  
 父类构造函数、  
 子类非静态变量（子类实例成员变量）、  
 子类构造函数。

1. **JAVA反射机制提供了什么功能**

在运行时判断一个对象所属的类

在运行时构造一个类的对象

在运行时判断一个类所具有的成员变量和方法

在运行时调用一个对象的方法

1. **Stream常用方法：**

* map: 用作类型转换 如把集合里面的字符串转为大写,或者一个对象的集合取几个字段转为新的对象集合
* filter: 过滤 符合条件的集合元素保存下来,不符合条件的去掉
* flatMap:合并集合,比如List<Album> Album里面有一LIst<Track> 对象,这个时候就能不通过循环的方式把 List<Album> 里的每一个元素的 trasks 对象组装成一个新的集合
* reduce: reduce可以做累加运算, .reduce(0, (a,b)-> a+b);
* count: count和size一样返回的是元素的个数
* max,min: 求最大值和最小值,这两个方法需要传入一个comparator比较器,Comparator比较器有一个comparing() 方法
* anyMatch表示，判断的条件里，任意一个元素成功，返回true
* allMatch表示，判断条件里的元素，所有的都是，返回true

1. **常见的异常类有哪些？**

* NullPointerException 空指针异常
* ClassNotFoundException 指定类不存在
* NumberFormatException 字符串转换为数字异常
* IndexOutOfBoundsException 数组下标越界异常
* ClassCastException 数据类型转换异常
* FileNotFoundException 文件未找到异常
* NoSuchMethodException 方法不存在异常
* IOException IO 异常
* SocketException Socket 异常

1. **Java中异常处理机制**

* 什么是异常

异常指的就是程序的不正常,简单理解就是程序所发生的错误.

* 异常的体系结构&分类

1. 分类
2. 编译时异常:指的就是编译期间,编译器检测到某段代码可能会发生某些问题,需要程序员提前给代码做出错误的解决方案,否则编译是不通过的.(例如FileReader)
3. 运行时异常:指的是编译通过了,但运行时出现的错误.
4. 体系结构

Throwable

Error：严重性错误

Exception：

RuntimeException:运行时异常

!RuntimeException:编译时异常

* 异常产生的原理

java对异常默认的处理方式,是将问题抛出给上一级

抛出之前,java会根据错误产生的异常类,创建出该类的对象,底层并通过throw关键字将异常抛出给上一级,不断向上抛出,直到抛给了JVM虚拟机,虚拟机拿到问题之后,就会将错误的原因和所在的位置,打印在控制台.

* 异常的处理方式

1. 问题可以自己处理掉的

try...catch处理方式:自己将问题处理掉,不会影响到后续代码的继续执行.

1. 问题自己处理不掉的

throws抛出处理方式：如果发现问题自己无法完美结局,就可以通过throw关键字,将异常对象抛出给调用者,但如果使用throw抛出异常对象,则方法上面必须进行throws的声明,告知调用者此方法存在异常.

细节:如果抛出的对象是RuntimeException,则方法上面无需throws声明.

1. **线程和进程的区别？**

* 进程：进程指正在运行的程序。确切的来说，当一个程序进入内存运行，即变成一个进程，进程是处于运行过程中的程序，并且具有一定独立功能。
* 线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。
* 简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程

1. **创建线程的方式**

* 继承Thread类

优点:代码简单

缺点:该类无法继承别的类

* 实现Runnable接口

优点:继承其他类.统一实现该接口的实例可以共享资源.

缺点:代码复杂.

* 实现Callable接口

Callable中的call()方法有返回值,其他和Runnable的run()方法一样.

* 线程池方式

优点:实现自动化装配,易于管理,循环利用资源.

1. **线程的状态**

* 就绪(Runnable):线程准备运行，不一定立马就能开始执行。
* 运行中(Running)：进程正在执行线程的代码。
* 等待中(Waiting):线程处于阻塞的状态，等待外部的处理结束。
* 睡眠中(Sleeping)：线程被强制睡眠。
* I/O阻塞(Blocked on I/O)：等待I/O操作完成。
* 同步阻塞(Blocked on Synchronization)：等待获取锁。
* 死亡(Dead)：线程完成了执行。

1. **Java中wait方法和sleep方法的不同之处**

* wait和sleep的基本使用:

wait:此方法来自于Object类,必须由锁对象进行调用

sleep:此方法来自于Thread类,是Thread类的静态方法,可以类名点调用

* wait方法和sleep方法的原理对比

sleep:让当前程序休眠xxx毫秒,休眠之后,程序继续执行.

wait:如果使用的wait方法是传入毫秒值参数的,产生的效果和sleep类似,但是wait方法较为麻烦一点,wait方法必须由锁对象调用,锁对象还必须放在同步当中.

* 总结:

wait方法:

1. 空参数:会让线程进入无限等待状态,进入无限等待状态后,必须由notify方法对其进行唤醒.

重点:

wait方法在等待的过程中,释放锁对象.

sleep方法在休眠的过程中,不会释放锁对象.

1. 有参数的:效果和sleep方法类似.
2. **线程池的种类**

* 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。
* newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
* newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
* newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行

因为在项目中，线程的创建和销毁非常消耗资源，所以使用在多线程场景的时候会使用线程池，根据自己业务逻辑的需求，使用不同的线程池。

1. **线程池都有哪些状态？**

* RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
* SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
* STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
* TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
* TERMINATED：terminated()方法结束后，线程池的状态就会变成这个

1. **线程同步**

* 线程同步：实现共享数据的一致性，让多个线程有序的访问共享资源，而不是同时操作共享资源
* 方法同步：用关键字 synchonized 可将方法声明为同步
* 同步代码块：synchornized 获取的是参数中的对象锁，synchornized(obj){}
* 同步类的属性：在类中声明多个object对象，对每个代码块分别操作类的不同属性加锁的时候就分别使用object对象来作为锁，这样能保证多个线程同时运行，分别操作不同的对象
* synchronized 静态方法与非静态方法：非静态同步方法的锁对象是this，静态的同步方法的锁对象是当前类的字节码对象
* 当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本

1. **多线程中 synchronized 锁升级的原理是什么？**

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

1. **volatile与synchronized的区别，底层实现**

* 首先两者都是用来确保数据的一致性的，volatile它能够使变量在值发生改变时尽快让其他线程知道，为什么要这么做呢？编译器为了加快程序运行速度，对一些变量的写操作会现在寄存器或者是cpu缓存上进行，最后才写入内存，这个过程，变量的新值对其他线程是不可见的，而volatile的作用就是使它修饰的变量的读写操作都必须在内存中进行

区别：

* Volatile本质是告诉jvm当前变量在寄存器中的值是不安全的需要从内存中读取，sychronized则是锁定当前变量，只有当前线程可以访问到该变量其他线程被阻塞
* Volatile只能作用于变量，synchronized则是可以使用在变量和方法上
* Volatile仅能实现变量的修改可见性,但不具备原子特性,而synchronized则可以保证变量的修改可见性和原子性
* volatile不会造成线程的阻塞,而synchronized可能会造成线程的阻塞
* volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化

注意事项：

* 在使用volatile关键字时要慎重，并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原来操作，当变量的值由自身的上一个决定时，如n=n+1、n++ 等，volatile关键字将失效，只有当变量的值和自身上一个值无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原子级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile

1. **什么是死锁？**

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

1. **怎么防止死锁？**

* 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用 Java. util. concurrent 并发类代替自己手写锁。
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
* 尽量减少同步的代码块。

1. **IO对比总结**

* IO 的方式通常分为几种：同步阻塞的 BIO、同步非阻塞的 NIO、异步非阻塞的 AIO。
* BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。
* NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。
* AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

举个例子：

* 同步阻塞：你到饭馆点餐，然后在那等着，啥都干不了，饭馆没做好，你就必须等着！
* 同步非阻塞：你在饭馆点完餐，就去玩儿了。不过玩一会儿，就回饭馆问一声：好了没 啊！
* 异步非阻塞：饭馆打电话说，我们知道您的位置，一会给你送过来，安心玩儿就可以了， 类似于现在的外卖。

1. **反射，泛型项目中有没有使用**

* 反射：spring框架，底层使用大量反射，但是我们直接使用IOC注入对象，所以并不直接使用。除非特殊情况下，自己封装一些工具类的时候使用到（最后一句可以不说）。
* 泛型：底层肯定也是大量使用的，写的工具类，返回给前端对象中，对外的参数List Map等可以使用泛型。

1. **Java虚拟机有哪几块内存空间**

* 程序计数器：可以看作是当前线程所执行的字节码文件（class）的行号指示器，它会记录执行痕迹，是每个线程私有的；
* 栈：栈是运行时创建的，是线程私有的，生命周期与线程相同，存储声明的变量的
* 本地方法栈：为native方法服务，native方法是一种由非java语言实现的java方法，为什么使用这种方法呢，与java环境外交互，或者与操作系统交互
* 堆：堆是所有线程共享的一块内存，是在java虚拟机启动时创建的，几乎所有对象实例都在此创建，所以经常发生垃圾回收操作；
* 方法区：主要存储已被虚拟机加载的类的信息，常量，静态变量和即时编译器编译后的代码等数据，该区域是被线程共享的，很少发生垃圾回收

1. **JAVA中垃圾回收机制**

* 什么样的对象会被当做垃圾回收

当一个对象的引用(地址)没有变量去记录的时候,该对象就会成为垃圾对象,并在垃圾回收器空闲的时候对其进行清扫.

* 如何检验对象是否被回收

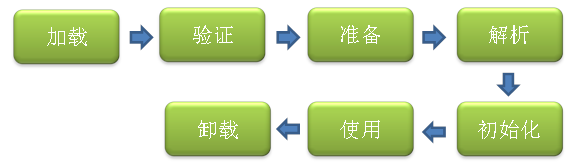
可以重写Object类中的finalize方法

这个方法在垃圾回收器执行的时候,被回收器自动调用执行的.

* 怎样通知垃圾回收器回收对象

可以调用system类的静态方法gc().通知垃圾回收器去清理垃圾.

1. **类加载过程**



* 加载 -> 验证 -> 准备 -> 解析 -> 初始化
* 加载:1.获取类的二进制字节流

2.将字节流代表的静态存储结构转化为方法区运行时数据结构

3.在堆中生成class字节码对象

* 验证:连接过程的第一步,确保class文件的字节流中的信息符合当前虚拟机的要求,不会危害虚拟机的安全
* 准备:为类的静态变量分配内存并将其初始化为默认值
* 解析:虚拟机将常量池内符号引用替换成直接引用的过程
* 初始化:执行类构造器的init的过程

1. **对象创建过程**

* JVM会先去方法区下找有没有所创建对象的类存在，有就可以创建对象了，没有则把该类加载到方法区
* 在创建类的对象时，首先会先去堆内存中分配空间
* 当空间分配完后，加载对象中所有的非静态成员变量到该空间下
* 所有的非静态成员变量加载完成之后，对所有的非静态成员进行默认初始化
* 所有的非静态成员默认初始化完成之后，调用相应的构造方法到栈中
* 在栈中执行构造函数时，先执行隐式，再执行构造方法中书写的代码
* 执行顺序：静态代码库，构造代码块，构造方法
* 当整个构造方法全部执行完，此对象创建完成，并把堆内存中分配的空间地址赋给对象名（此时对象名就指向了该空间）

1. **方法区堆栈溢出怎么处理**

jdk1.7之前字符串常量池是方法区的一部分，方法区叫做“永久代”，在1.7之前无限的创建对象就会造成内存溢出

用jdk1.7之后，取消了永久代，添加了元数据区，就不会产生内存溢出。

在jdk1.7时出现内存溢出：

1. 需要查看代码中是否出现死循环。
2. 是否出现死锁现象。
3. 在jvm运行时，提高堆内存的大小。
4. **单例设计模式中懒汉式和饿汉式的区别**

* 饿汉式：

//饿汉式单例类.在类初始化时，已经自行实例化

public class Singleton1 {

private Singleton1() {}

private static final Singleton1 single = new Singleton1();

//静态工厂方法

public static Singleton1 getInstance() {

return single;

}

}

* 懒汉式：

//懒汉式单例类.在第一次调用的时候实例化自己

public class Singleton {

private Singleton() {}

private static Singleton single=null;

//静态工厂方法

public static Singleton getInstance() {

if (single == null) {

single = new Singleton();

}

return single;

}

}

* 饿汉式就是类一旦加载，就把单例初始化完成，保证getInstance()的时候，单例就已经存在。
* 懒汉式比较懒，只有当调用getInstance的时候，才会去初始化这个单例

区别：

* 饿汉式是线程安全的，懒汉式是线程不安全的（即一个进程内有多个线程在在同时使用时可能会产生多个实例，可创建个静态内部类，产生一个单例对象，通过静态内部类返回获取这个对象）

1. **常见的基本排序**

* 冒泡排序

//N个数字冒泡排序，总共要进行N-1趟比较，每趟的排序次数为(N-i)次比较。下面示例虽然每趟是n-i-1，但i是从0开始的。   
 public static void bubbleSort(int[] arr){  
 //一定要记住判断边界条件，很多人不注意这些细节，面试官看到你的代码的时候都懒得往下看，你的代码哪个项目敢往里面加？  
 if(arr==null||arr.length<2){  
 return;  
 }  
 //需要进行arr.length趟比较  
 for(int i = 0 ;i<arr.length-1;i++){  
 //第i趟比较  
 for(int j = 0 ;j<arr.length-i-1;j++){  
 //开始进行比较，如果arr[j]比arr[j+1]的值大，那就交换位置  
 if(arr[j]>arr[j+1]){  
 int temp=arr[j];  
 arr[j]=arr[j+1];  
 arr[j+1]=temp;  
 }  
 }  
  
 }  
 }

* 选择排序：

**void** **select\_sort**(**int** a[], **int** n){

**int** i;

**int** j;

**int** min;

**for**(i**=**0; i**<**n; i**++**){

min**=**i;

*//找"a[i+1]..a[n]"之间最小元素，并赋给min*

**for**(j**=**i**+**1; j**<**n; j**++**){

**if**(a[j] **<** a[min])

min**=**j;

}

*//若min!=i，则交换 a[i] 和 a[min]。*

*//交换后，保证了a[0]..a[i]之间元素有序。*

**if**(min **!=** i)

swap(a[i], a[min]);

}}

1. **常见设计模式**

* 工厂模式：一个抽象接口的实现，多个抽象接口的实现类，spring的beanFactory就是工厂模式
* 单例模式：在内存中，保证对象的实例只有一个。
* 装饰者模式：对一个类进行装饰，增强其方法行为，如Java中的IO流就使用了装饰者模式
* 代理模式：比如sping AOP使用动态代理
* 适配器模式：io流，通过继承实现将一个接口适配到另一个接口，InputStreamReader类继承Reader接口，但要创建它们必须在构造函数中传入一个InputStream的实例，InputStreamReader的作用也就是将InputStream适配到Reader
* 状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

**数据库:**

1. **事务的特性和隔离级别**

事务的特性：

* 原子性（Atomicity）

原子性指事务是不可分割的工作单位，事务中的操作要么都发生，要么都不发生

* 一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态

* 隔离性（Isolation）

事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

* 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

隔离级别：

* 未提交读read uncommitted

会发生 脏读、不可重复读、幻读

* 已提交读read committed //Oracle SQL Server（系统事务）

解决脏读，但是不可重复读和幻读有可能发生

* 重复读repeatable read //Mysql

解决脏读和不可重复读，但是幻读有可能发生.

* 串行化serializable

避免脏读，不可重复读，幻读的发生

* 脏读：指一个线程中的事务读取到了另外一个线程中未提交的数据。
* 不可重复读：由于另一个事务对数据的更改所造成的
* 幻读：由于另一个事务插入或删除引起的。

1. **Mysql的体系结构**

* 连接层
* 最上层是一些客户端和链接服务，包含本地sock 通信和大多数基于客户端/服务端工具实现的类似于 TCP/IP的通 信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证 它所具有的操作权限。
* 服务层
* 第二层架构主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部分内置函数的执行。所有跨存储引擎的功能也在这一层实现，如 过程、函数等。在该层，服务器会解析查询并创建相应的内部 解析树，并对其完成相应的优化如确定表的查询的顺序，是否利用索引等， 最后生成相应的执行操作。如果是 select语句，服务器还会查询内部的缓存，如果缓存空间足够大，这样在解决大量读操作的环境中能够很好的提升系统的性能。
* 引擎层
* 存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API和存储引擎进行通信。不同的存储引擎具有不同的功能，这样我们可以根据自己的需要，来选取合适的存储引擎。
* 存储层
* 数据存储层，主要是将数据存储在文件系统之上，并完成与存储引擎的交互。和其他数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎上，插件式的存储引擎架构，将查询处理和其他的系统任务以及数据的存储提取分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。

1. **InnoDB，MyISAM存储引擎特性**

* InnoDB
* InnoDB存储引擎是Mysql的默认存储引擎。InnoDB存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全。 但是对比MyISAM的存储引擎，InnoDB写的处理效率差一些，并且会占用更多的磁盘空间以保留数据和索引。
* MyISAM
* MyISAM 不支持事务、也不支持外键，其优势是访问的速度快，对事务的完整性没有要求或者以SELECT、INSERT 为主的应用基本上都可以使用这个引擎来创建表 。

1. **CHAR和VARCHAR的区别：**

* CHAR和VARCHAR类型在存储和检索方面有所不同
* CHAR列长度固定为创建表时声明的长度，长度值范围是1到255
* 当CHAR值被存储时，它们被用空格填充到特定长度，检索CHAR值时需删除尾随空格。

1. **delete、drop、truncate区别**

* truncate 和 delete只删除数据，不删除表结构 ,drop删除表结构，并且释放所占的空间。
* 删除数据的速度，drop> truncate > delete
* delete属于DML语言，需要事务管理，commit之后才能生效。drop和truncate属于DDL语言，操作立刻生效，不可回滚。
* 使用场合：
* 当你不再需要该表时， 用 drop;
* 当你仍要保留该表，但要删除所有记录时， 用 truncate;
* 当你要删除部分记录时（always with a where clause), 用 delete.

1. **mysql 中 in 和 exists 区别**

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。  
 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in：  
not in 和not exists如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。  
 1.EXISTS只返回TRUE或FALSE，不会返回UNKNOWN。

2.IN当遇到包含NULL的情况，那么就会返回UNKNOWN。

1) select \* from T1 where exists(select 1 from T2 where T1.a=T2.a) ;

    T1数据量小而T2数据量非常大时，T1<<T2 时，1) 的查询效率高。Exists用的是内表的索引，即T2的索引

2) select \* from T1 where T1.a in (select T2.a from T2) ;

     T1数据量非常大而T2数据量小时，T1>>T2 时，2) 的查询效率高。In用的是外表的索引，即T1的索引

 简而言之，一般式：外表大，用IN；内表大，用EXISTS。

1. **数据库的三范式是什么？**

* 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。
* 第二范式：有主键，非主键元素依赖于主键。
* 第三范式：任何非主属性不依赖于其它非主属性。

1. **Sql优化**

* 通过慢查询日志去寻找，哪些sql执行效率低
* 使用explain分析低效率的sql执行计划
* 针对低效率的sql执行计划分析

没有索引

索引失效

数据量太大

没有索引：

* 针对查询的列创建索引，提高查询效率，但是索引太多，mysql也会出现选择困难，所以建立索引要有效，无效的索引需要删除。

索引失效：

* 尽量选择较小的列
* 将where中用的比较频繁的字段建立索引
* select子句中避免使用‘\*’
* 避免在索引列上使用计算，not in和<>等操作
* 当只需要一行数据的时候使用limit 1
* 保证表单数据不超过200w，适时分割表
* 针对查询较慢的语句，可以使用explain来分析该语句具体的执行情况
* 避免查询时判断null，否则可能会导致全表扫描，无法使用索引;
* 避免like查询，否则可能导致全表扫描，可以考虑使用全文索引
* 能用union all的时候就不用union，union过滤重复数据要耗费更多的CPU资源

数据量太大：

* 分页查询优化
* 在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容。
* Select \* from table t where t.id in（select id from table order by id limit 100000，10） ;
* 对于主键自增的表，可以把limit查询缓存某个位置的查询
* Select \* from table where id > 100000 limit 10;

分库分表：

* 使用MyCat中间件实现。

全文索引技术：

* ElasticSearch ， solr

非关系型数据库：

* 不需要像关系型数据库一样维护表于表之间的关系，而是使用json这种灵活多变的形式，效率比Mysql提高很多

1. **分库分表方案**

* 一开始上来就是32个库，每个库32个表，1024张表这个分法，

基本上国内的互联网肯定都是够用了

无论是并发支撑还是数据量支撑都没问题

* 如果每个库正常承载的写入并发量是1000，那么32个库就可以承载32 \* 1000 = 32000的写并发，如果每个库承载1500的写并发，32 \* 1500 = 48000的写并发，接近5万/s的写入并发，前面再加一个MQ，削峰，每秒写入MQ 8万条数据，每秒消费5万条数据。1024张表，假设每个表放500万数据，在MySQL里可以放50亿条数据。每秒的5万写并发，总共50亿条数据，对于国内大部分的互联网公司来说都够了。
* 此方案最多可以扩展到32个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到1024个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是1024个表么。

服务器升级流程：

* 设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是32库 \* 32表，对于大部分公司来说，可能几年都够了
* 路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表
* 扩容的时候，申请增加更多的数据库服务器，装好mysql，倍数扩容，4台服务器，扩到8台服务器，16台服务器
* 由dba负责将原先数据库服务器的库，迁移到新的数据库服务器上去，很多工具，库迁移，比较便捷
* 我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址
* 重新发布系统，上线，原先的路由规则变都不用变，直接可以基于2倍的数据库服务器的资源，继续进行线上系统的提供服务

1. **MySQL 索引是怎么实现的？**

索引是满足某种特定查找算法的数据结构，而这些数据结构会以某种方式指向数据，从而实现高效查找数据。

具体来说 MySQL 中的索引，不同的数据引擎实现有所不同，但目前主流的数据库引擎的索引都是 B+ 树实现的，B+ 树的搜索效率，可以到达二分法的性能，找到数据区域之后就找到了完整的数据结构了，所有索引的性能也是更好的。

1. **Mysql支持的索引类型**

* index  ----  普通索引,数据可以重复，没有任何限制。
* unique   ---- 唯一索引,要求索引列的值必须唯一，但允许有空值；如果是组合索引，那么列值的组合必须唯一。
* primary key ---- 主键索引,是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值，一般是在创建表的同时创建主键索引。
* 组合索引 ----  在多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。
* fulltext ---- 全文索引,是对于大表的文本域：char，varchar，text列才能创建全文索引，主要用于查找文本中的关键字，并不是直接与索引中的值进行比较。

1. **索引失效问题**

7种引起索引失效的情境

* 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

1. **左连接 ，右连接，内连接和全外连接的4者区别**

* left join （左连接）：返回包括左表中的所有记录和右表中连接字段相等的记录。
* right join （右连接）：返回包括右表中的所有记录和左表中连接字段相等的记录。
* inner join （等值连接或者叫内连接）：只返回两个表中连接字段相等的行。
* full join （全外连接）：返回左右表中所有的记录和左右表中连接字段相等的记录。

1. **一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 MySQL 数据库，又插入了一条数据，此时 id 是几？**

* 表类型如果是 MyISAM ，那 id 就是 8。
* 表类型如果是 InnoDB，那 id 就是 6。  
  InnoDB 表只会把自增主键的最大 id 记录在内存中，所以重启之后会导致最大 id 丢失。

1. **说一下 MySQL 的行锁和表锁？**

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。  
 行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

1. **说一下乐观锁和悲观锁？**

乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。

悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻止，直到这个锁被释放。

数据库的乐观锁需要自己实现，在表里面添加一个 version 字段，每次修改成功值加 1，这样每次修改的时候先对比一下，自己拥有的 version 和数据库现在的 version 是否一致，如果不一致就不修改，这样就实现了乐观锁。

1. **高并发下，如何做到安全的修改同一行数据。**

使用悲观锁 悲观锁本质是当前只有一个线程执行操作，结束了唤醒其他线程进行处理。也可以缓存队列中锁定主键。

1. **数据库会死锁吗，举一个死锁的例子**

产生死锁的原因主要是：

（1）系统资源不足。  
 （2） 进程运行推进的顺序不合适。  
 （3）资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。  
 （2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。  
 （3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  
 （4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

1. **聚集索引和非聚集索引的区别**

聚簇索引就是索引和记录紧密在一起。  
 非聚簇索引 索引文件和数据文件分开存放，索引文件的叶子页只保存了主键值，要定位记录还要去查找相应的数据块。（非聚集索引，应该就是二级索引吧，比如原表是按abcd...xyz顺序排列的，那么这个字母就是聚集索引，然后非聚集索引就是1-a,2-c,3-b，通过1找到a行，2找到c行，1-2是连续的，但ac不是）

每个表只能有一个聚集索引，因为目录只能按照一种方法进行排序 。

**WEB:**

1. **cookie和session的区别与联系**
2. cookie数据存放在客户的浏览器上,session数据存放在服务器上.
3. 很多浏览器限制站点最多保存20个cookie,单个cookie保存的数据不能超过4k.
4. cookie不是很安全,考虑安全应当使用session.
5. 可以考虑将登录信息等重要信息存放为session,其它信息如果需要保留,可以放在cookie中.
6. session会在一定时间内保存在服务器上.
7. session会在浏览器关闭或者一段时间内销毁,也可以通过setMaxInactiveInterval(int)方法进行设置,或是通过invalidate()方法强制结束当前会话.cookie可以通过setMaxAge(int)方法设置缓存在客户端的时间.
8. 一般情况下,session生成的sessionid都是保存在cookie中.

总结:

cookie:在客户端保存数据,不安全.只能保存字符串,且是少量数据.

session:在服务器端保存数据,安全.可以保存对象数据,数据无限制.

1. **如果客户端禁止 cookie 能实现 session 还能用吗？**

可以用，session 只是依赖 cookie 存储 sessionid，如果 cookie 被禁用了，可以使用 url 中添加 sessionid 的方式保证 session 能正常使用。

1. **Get方法和Post方法区别**

get:查询、请求参数 不安全,提交的数据量小

post:添加或更新数据、提交数据比较安全、提交的数据量无限制.

1. get方法用于信息获取,他是安全的,而post方法是用于修改服务器上资源的请求.
2. get请求的数据会附在url之后,而post方法提交的数据则放置在http报文实体的主体里,所以post方法的安全性比get方法要高.
3. get方法传输的数据量一般限制在2kb,post方法对于数据大小是无限制的.
4. **servlet的生命周期及常用方法**

* init()方法:在servlet的生命周期中,仅执行一次init()方法.
* service()方法:它是servlet的核心,每当客户请求一个httpservlet对象,该对象的service()方法就要调用,而且传递给这个方法一个”请求”对象和一个”响应”对象作为参数.
* destory()方法:仅执行一次,在服务器端停止且卸载servlet时执行该方法.

解决servlet线程安全

1. 继承SingleThreadModel,消耗服务器内存,降低性能.并且过时,不推荐.
2. 尽量避免使用全局变量,推荐.
3. 通过使用ThreadLocal.
4. **过滤器有哪些作用，以及过滤器的生命周期？**

生命周期：每个Filter在tomcat启动时进行初始化，每个Filter只有一个实例对象

* Init:在服务器启动时会创建Filter实例
* doFilto:这个方法会在用户每次访问“目标资源”时执行
* destroy():服务器关闭时销毁Filter对象

作用：

* 验证客户是否来自可信网络
* 对客户提交的数据进行重新编码
* 过滤掉客户的某些不应该出现的词汇
* 验证用户是否可以登录
* 验证客户的浏览器是否支持当前的应用
* 记录系统日志

1. **转发和重定向的区别**
2. 重定向是浏览器发送请求并收到响应以后再次向一个新地址发请求;转发是服务器收到请求后为了完成响应转到另一个资源.
3. 重定向中有两次请求对象,不共享数据;转发只产生一次请求对象且在组件间共享数据.
4. 重定向后地址栏地址改变,而转发不会.
5. 重定向的新地址可以是任意地址;转发必须是同一个应用内的某个资源.

获取servlet的转发和响应重定向的方式

* 转发的方法:

通过HttpServletRequest的getRequestDispatcher()方法获得

通过ServletContext的getRequestDispatcher()方法获得

* 重定向的方法:

HttpServletResponse的sendRedirect()方法.

1. **ajax书写方式及内部主要参数**

主要参数:

1. url:要求为String类型的参数,发送请求的地址.
2. data:要求为Object或String类型,发送到服务器的数据.
3. type:要求为String类型,请求方式get或post.
4. datatype:要求为String类型,预期服务器返回的类型.
5. timeout:要求为number类型,设置请求超时时间.
6. async:要求为boolean类型,异步为true(默认),同步为false.
7. cache:要求为boolean类型,默认为true,是否从浏览器缓存中加载信息.
8. beforesend:要求为Function类型的参数.例如添加自定义http头.

ajax的优缺点:

* 优点:减轻服务器的负担,按需取数据,最大程度的减少冗余请求,局部刷新页面,减少用户心理和实际的等待时间,带来更好的用户体验.
* 缺点:ajax大量的使用了JavaScript和ajax引擎,这些取决于浏览器的支持,在编写的时候考虑对浏览器的兼容性.ajax只是局部刷新,所以页面的后退按钮是没有用的.

1. **Jquery常用选择器**

Jquery选择器总共有四大类:

基本选择器、层级选择器、过滤选择器和表单选择器.

* 基本选择器是Jquery最常用的选择器,也是最简单的选择器,他通过元素id,class和标签名来查找dom元素.
* 层级选择器是通过dom元素间的层次关系来获取元素,主要层次关系包括父子、后代、相邻、兄弟关系.
* 过滤选择器主要是通过特定的过滤规则筛选出所需的dom元素,过滤规则与css中的伪类选择器语法相同,即选择器都以一个冒号开头.

1. Jquery基本过滤选择器
2. Jquery内容过滤选择器
3. Jquery可见性过滤选择器
4. Jquery属性过滤选择器
5. Jquery子元素过滤选择器
6. Jquery表单对象属性过滤选择器

* 表单选择器:我们可以极其方便的获取表单的某个或某类型的元素.

1. **JSP和Servlet的区别**

* 什么是JSP:Java Server Pages(Java服务器端页面)动态页面

它和servlet技术一样,都是SUN公司定义的一种用于开发动态web资源的技术.

jsp=html+java

* jap执行原理

.jsp(翻译)-->.java(编译)-->.class(执行)

servlet:服务器端的小应用程序.适合编写java逻辑代码.

jsp:适合编写输出动态内容,但不适合编写java逻辑.

1. **JSP常用的标签**

* 请求转发：<jsp:forward>
* 页面传递数据：<jsp:param>
* 输出标签：<c:out>
* 判读标签<c:if>
* 迭代标签<c:foreach>
* 多重判断标签<c:choose>

静态包含包含的是内容,动态包含包含的是结果.

静态包含不可以传递参数,而动态包含可以传递参数.

1. **jsp九大内置对象及作用**

* PageContext JSP的页面容器
* request 获取用户的请求信息
* response 服务器向客户端的回应信息
* session 用来保存每一个用户的信息
* application 表示所有用户的共享信息
* config 服务器配置信息，可以取得初始化参数
* out 页面输出
* page 但前页面对象，可以获取其他对象
* exception 异常对象

1. **JSP四大作用域及请求范围**

JSP四大作用域从小到大分别为:page,request,session,application.

* 第一个作用域是page,他只在当前页面有效,也就是用户的请求页面有效.
* 第二个作用域是request,他在当前第一次请求中有效.
* 第三个作用域是session,他在当前整个会话中有效.
* 第四个作用域是application,他在整个应用都有效.

PageContext:pageContext 存放的数据在当前页面有效.开发时使用较少.

ServletRequest:request 存放的数据在第一次请求(转发)中有效.使用非常多.

HttpSession:session 存放的数据在第一次会话中有效.使用的比较多.如存放登录信息,购物车

ServletContext:application 存放的数据在整个应用范围都有效.因为范围太大,应尽量少用.

1. **如何防止表单重复提交**

网络延迟时,重复点击提交按钮,有可能发生重复提交表单问题.

解决方案:

1. 数据库主键唯一.
2. 提交成功后重定向.
3. 使用JavaScript解决,使用标记位,提交后隐藏或不可用提交按钮.

使用session解决:

生成唯一的Token(uuid)给客户端,客户端第一次提交时带着这个Token,后台与session中的进行对比.

1. **常见的http返回状态码**

100:告诉客户端应继续发送请求

200:请求响应成功

202:请求已被受理还未做出响应

301：永久重定向

302：暂时重定向

400:请求无效,常见的情况是请求参数有误,http头构建错误

404:访问不到资源

500:服务器后端错误

1开头的状态码是消息类型的.

2开头的状态码表示成功.

3开头的状态码表示需要重定向.

4开头的状态码表示请求错误.

5开头的状态码表示服务器错误.

1. **TCP和UDP的区别,HTTP协议**

* TCP协议提供安全可靠的网络传输服务,它是一种面向连接的服务.类似于打电话,必须先拨号.双方建立一个传递信息的通道传输.
* UDP协议是一种数据报协议,它传输的数据是分组报文,它是无连接的,不需要和目标通信方建立连接,类似于写信,所以它的传输不保证安全可靠.但适合大数据量的传输.
* HTTP协议是超文本传输协议,是一种相对于TCP来说更细致的协议,TCP以及UDP协议规范的是网络设备之间的通信规范,HTTP实在TCP协议的基础上针对用户的协议,用户服务具体体现在应用程序之间的交互,比如javaweb中客户端服务端体系就要用http协议来规范通信.

TCP和UDP在开发中很少见到,但是网络底层都有他们的影子,正常的会话级别的服务:如客户端服务器体系底层就说基于TCP协议.而邮件发送,短信发送等底层使用的是UDP协议.

HTTP协议,客户端/服务器体系的程序都使用HTTP协议来规范通信.

1. **tcp为什么要三次握手，两次不行吗？为什么？**

如果采用两次握手，那么只要服务器发出确认数据包就会建立连接，但由于客户端此时并未响应服务器端的请求，那此时服务器端就会一直在等待客户端，这样服务器端就白白浪费了一定的资源。若采用三次握手，服务器端没有收到来自客户端的再此确认，则就会知道客户端并没有要求建立请求，就不会浪费服务器的资源。

1. **说一下 tcp 粘包是怎么产生的？**

tcp 粘包可能发生在发送端或者接收端，分别来看两端各种产生粘包的原因：

发送端粘包：发送端需要等缓冲区满才发送出去，造成粘包；  
 接收方粘包：接收方不及时接收缓冲区的包，造成多个包接收。

1. **json的数据格式**

* Json的最初出现是专门为JavaScript准备的,它是一种轻量级的数据交换格式.

1. 数据在名称/值对中;
2. 数据由逗号隔开;
3. 花括号保存对象

{”属性名”:”值”,”属性名”:”值”}

* Json对象数组:

[{”属性名”:”值”,”属性名”:”值”},{”属性名”:”值”,”属性名”:”值”}]

* 复杂格式:

{”属性名”:”值”,”属性名”:{”属性名”:”值”}}

json中的值是有限制的,对于简单类型来说,只能是字符串,数值(必须是十进制)、布尔值和null;对于复合类型来说,只能放数组或者对象,不能是正则、函数或者日期;

json在网络开发中有非常广泛的用途,但可以归纳为一句:可以用于接口开发及调用中使用的数据格式.一来用于服务端和JavaScript之间的数据交互,二来可以用于跨域传输数据的数据格式.

在项目中,前后端交互、接口开发中很多都使用json来作为数据传输格式.

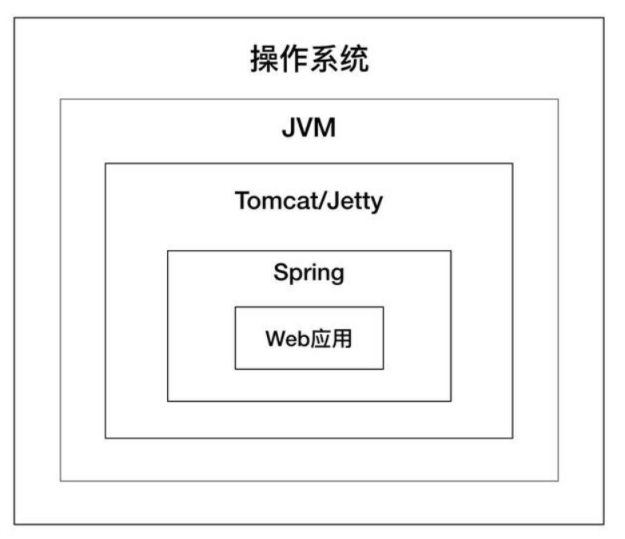
1. **如何实现跨域？**
   * 服务器端运行跨域 设置 CORS 等于 \*；
   * 在单个接口使用注解 @CrossOrigin 运行跨域；

* 使用 jsonp 跨域,JSONP 实现原理:

jsonp：JSON with Padding，它是利用script标签的 src 连接可以访问不同源的特性，加载远程返回的“JS 函数”来执行的。

1. **Tomcat体系结构讲解**

Tomcat 或者 Jetty 就是一个“HTTP 服务器 + Servlet 容器”，我们也叫它们 Web 容器。

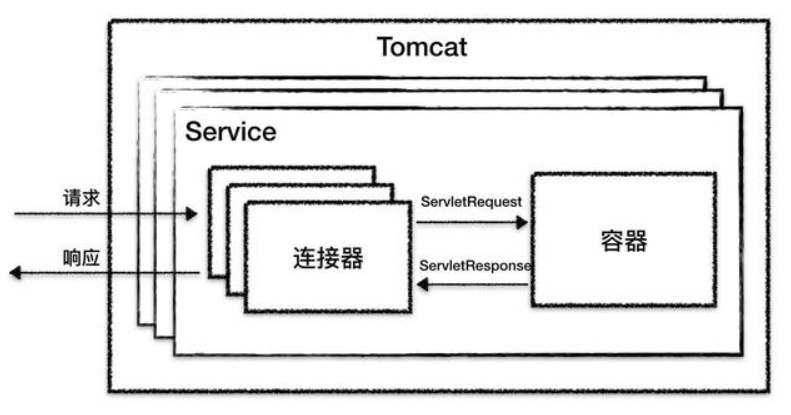


Web服务器需要实现两个核心功能：

* 处理 Socket 连接，负责网络字节流与 Request 和 Response 对象的转化。
* 加载和管理 Servlet，以及具体处理 Request 请求。

因此 Tomcat 设计了两个核心组件连接器（Connector）和容器（Container）来分别做这两件事情。连接器负责对外交流，容器负责内部处理。

可参考：http://www.sohu.com/a/321678611\_120176035



(1) Server  
 Server表示整个的Catalina Servlet容器。Tomcat提供了Server接口的一个默认实现，这通常不需要用户自己去实现。在Server容器中，可以包含一个或多个Service组件。  
(2) Service  
 Service是存活在Server内部的中间组件，它将一个或多个连接器（Connector）组件绑定到一个单独的引擎（Engine）上。在Server中，可以包含一个或多个Service组件。Service也很少由用户定制，Tomcat提供了Service接口的默认实现，而这种实现既简单又能满应用。  
(3) Connector  
 连接器（Connector）处理与客户端的通信，它负责接收客户请求，以及向客户返回响应结果。在Tomcat中，有多个连接器可以使用。  
(4) Engine  
 在Tomcat中，每个Service只能包含一个Servlet引擎（Engine）。引擎表示一个特定的Service的请求处理流水线。作为一个Service可以有多个连接器，引擎从连接器接收和处理所有的请求，将响应返回给适合的连接器，通过连接器传输给用户。用户允许通过实现Engine接口提供自定义的引擎，但通常不需要这么做。  
(5) Host  
 Host表示一个虚拟主机，一个引擎可以包含多个Host。用户通常不需要创建自定义的  
Host，因为Tomcat给出的Host接口的实现（类StandardHost）提供了重要的附加功能。  
(6) Context  
 一个Context表示了一个Web应用程序，运行在特定的虚拟主机中。什么是Web应用程序呢？在Sun公司发布的Java Servlet规范中，对Web应用程序做出了如下的定义：“一个Web应用程序是由一组Servlet、HTML页面、类，以及其他的资源组成的运行在Web服务器上的完整的应用程序。它可以在多个供应商提供的实现了Servlet规范的Web容器中运行”。一个Host可以包含多个Context（代表Web应用程序），每一个Context都有一个唯一的路径。用户 通 常 不 需 要 创 建 自 定 义 的 Context ， 因 为 Tomcat 给 出 的 Context 接 口 的 实 （ 类StandardContext）提供了重要的附加功能。凡是实现了Servlet规范的都可以成为Servlet容器

1. **tomcat 如何调优，涉及哪些参数**

硬件上选择，操作系统选择，版本选择，jdk选择，配置jvm参数，配置connector的线程数量，开启gzip压缩，trimSpaces，集群等  
可参考：[http://blog.csdn.net/lifetragedy/article/details/7708724](https://link.jianshu.com?t=http://blog.csdn.net/lifetragedy/article/details/7708724" \t "_blank)

作者：消失er  
链接：https://www.jianshu.com/p/97e210a6d95a  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**框架:**

1. **Spring**

* spring是一个轻量级的,开源的,模块化的,一站式业务层框架,它能够整合其他主流框架
* spring的实质就是实现了工厂模式的工厂类,在配置文件中,通过添加<bean>标签,来创建实例对象
* spring的核心分别是Ioc和AOP

IoC是控制反转,即使将常见实力对象的权力交给spring框架去管理,通过依赖注入的方式去,去注入给调用者,解除bean与bean之间的耦合性

依赖注入的三种方式分别是:接口注入,构造注入,setter方法注入(<property name="" value或ref="">

AOP是指面向切面编程,就是在不修改代码的前体下,对程序进行加强,例如加入日志,权限判断,异常处理等等

AOP的底层使用的是代理技术,分别为jdkProxy动态代理和CGLIB动态代理,jdk动态代理是需要被代理的类实现接口的,而CGLIB则不需要被代理的类实现接口

* 关于动态代理,它的底层实现是利用java反射来实现.

Proxy主要使用 Proxy.newProxyInstance方法

CGLIB主要使用 Enhancer.create方法

MethodHandler主要使用 MethodHandles.lookup().findVirtual().bindTO()方法

静态代理:是实现已经编写好,在程序运行前代理类的.class文件就已经存在

动态代理:在程序运行时运用反射机制动态创建而成

静态代理虽然执行效率要比动态代理高,但需要实现编写好代理类的java文件,工程量大,由于代理类和委托类实现了相同接口,会出现大量重复代码,而且后期不易于维护;

而动态代理则是随着程序的运行而创建代理对象,比较灵活,但是动态代理的底层是反射机制,比较消耗资源

1. **spring 有哪些主要模块？**

* spring core：框架的最基础部分，提供 ioc 和依赖注入特性。
* spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
* spring dao：Data Access Object 提供了JDBC的抽象层。
* spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
* spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
* spring Web mvc：spring 中的 mvc 封装包提供了 Web 应用的 Model-View-Controller（MVC）的实现。

1. **Spring AOP**

Spring AOP的面向切面编程,是面向对象编程的一种补充,用于处理系统中分布的各个模块的横切关注点,比如事务管理、日志、缓存等.它是使用动态代理实现的,在内存中临时为增强某个方法生成一个AOP对象,这个对象包含目标对象的所有方法,在特定的切入点做了增强处理,并回调原来的方法.

Spring AOP的动态代理主要有两种方式实现,JDK动态代理和cglib动态代理.JDK动态代理通过反射来接受被代理的类,但是被代理的类必须实现接口,核心是InvacationHandler和Proxy类.cglib动态代理的类一般是没有实现接口的类,cglib是一个代码生成的类库,可以在运行时动态生成某个类的子类.所以,cglib是通过继承的方式做的动态代理,因此如果某个类被标记为final,那么它是无法使用cglib做动态代理的.

AOP能做什么:

* 降低模块的耦合度.
* 使系统容易扩展
* 避免修改业务代码,避免引入重复代码,更好的代码复用.

AOP怎么用:

* 前置通知:某方法调用前发出通知.
* 后置通知:某方法完成之后发出通知.
* 返回后通知:方法正常返回后,调用通知.在方法,正常退出发出通知.
* 异常通知:抛出异常后通知:在方法抛出异常退出时执行的通知.在方法调用时,异常退出发出通知.
* 环绕通知:通知包裹在被通知的方法的周围.

1. **IOC和DI**

* IOC(控制反转).将对在自身对象中的一个内置对象的控制反转,反转后不再由自己本身的对象进行控制这个内置对象的创建,而是由第三方系统去控制这个内置对象的创建.简单来说就是把本来在类内部控制的对象,反转到类外部进行创建后注入,不再由类本身进行控制,这就是IOC本质.
* DI(依赖注入).自身对象中的内置对象是通过注入的方式进行创建.
* IOC和DI的关系.

ioc就是容器,di就是注入这一行为,那么di确实就是ioc的具体功能的实现.而ioc则是di发挥的平台和空间.所以说,ioc和di即是相辅相成的搭档.最重要的是,他们都是为了实现解耦而服务的.

* DI是如何实现的.

依赖注入可以通过setter方法注入、构造注入、接口注入三种方法来实现,Spring支持setter注入和构造器注入,通常使用构造器注入来注入必须的依赖关系,对于可选的依赖关系,则setter注入是更好的选择,setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象.

1. **Spring中Bean的作用域**

Spring IOC容器在根据配置创建一个Bean对象实例时,可以为Bean指定实例的作用范围.

* singleton(单例模式)

IOC容器仅创建一个Bean实例,IOC容器每次返回的是同一个Bean实例.

* prototype(原型模式)

IOC容器仅创建多个Bean实例,IOC容器每次返回的是一个新的实例.

* request(HTTP请求)

该属性仅对HTTP请求产生作用,每次HTTP请求都会创建一个新的Bean,适用于WebApplicationContext.

* session(会话)

该属性仅用于HTTP Session,同一个session共享一个Bean实例.不同session使用不同的实例.

* global-session(全局会话,在spring5.x中已移除)

该属性仅用于HTTP Session,同session作用域不同的时候,所有session共享一个Bean实例.

Bean的生命周期

在实际开发中,我们一般常用的就是单例模式和原型模式

单例模式生命周期和容器相同.

原型模式生命周期,是每次使用时创建新的对象,用完等待垃圾回收器回收.

1. **Spring框架实现实例化和依赖注入的方式**

* 实例化:

1. 构造器实例化Bean
2. 静态工厂方式实例化Bean
3. 实例工厂方式实例化Bean

* 依赖注入:

1. 基于构造函数的注入
2. 基于set方法的注入
3. 基于自动装配的注入
4. 基于注解的依赖注入
5. **springmvc执行流程**

* 用户发送请求至前端控制器DispatcherServlet
* DispatcherServlet收到请求调用HandlerMapping处理器映射器。
* 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
* DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
* HandlerAdapter执行处理器(handler，也叫后端控制器)。
* Controller执行完成返回ModelAndView
* HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
* DispatcherServlet将ModelAndView传给ViewReslover视图解析器
* ViewReslover解析后返回具体View对象
* DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
* DispatcherServlet响应用户

1. **Spring以及SpringMVC常用注解**

* @Component:用于标记在一个类上,表示当前类是spring的一个组件,是ioc的一个容器.他有三个衍生注解:@Controller、@Service、@Repository
* @Controller：用于标记在一个类上,代表这个类是控制层组件.
* @Service:用于标记在一个类上,代表这个类是业务层组件.
* @Repository:用于标记在一个类上,代表这个类是数据访问层组件.
* @RequestMapping:是一个用于处理请求地址映射的注解,可用于类或方法上.用于类上,表示类中所有响应请求的方法都是以该地址作为父路径
* @RequestParam:用于将指定的请求参数赋给方法中的形参.
* @PathVariable:可以获取URL中的动态参数.
* @RequestBody:用于读取request请求的body部分数据.
* @ResponseBody:用于将controller方法返回的对象,用流响应给客户端.
* @RestController:@Controller+@ResponseBody,用于标记在一个类上.
* @Transactional:写在类上用于指定当前类中的方法支持事务,写在方法上表示当前的方法支持事务

1. **springmvc获取表单的几种方式**

* 借助原始ServletAPI的HttpServletRequest对象.

Controller的方法,

添加HttpServletRequest类型入参,

通过HttpServletRequest.getParameter()获取请求数据

* 借助控制器方法的形参

controller的方法,添加参数来接收表单的数据.

接收的类型包括:

1. 基本类型或String
2. 实体类类型
3. 实体类关联对象

* Controller方法,接收实体类类型参数,而表单提交的是json数据时,可以使用@RequestBody注解获取全部请求体,配合jackson开源组件可以实现转换成实体类.要求json数据的key必须和实体类属性保持一致.

1. **Spring MVC的异常处理 ？**

在 Spring MVC的Web应用程序中，可以存在多个实现了HandlerExceptionResolver的异常处理类，他们的执行顺序，由其order属性决定, order值越小，越是优先执行, 在执行到第一个返回不是null的ModelAndView 的Resolver时，不再执行后续的尚未执行的Resolver的异常处理方法。

1. **为什么要用 spring boot？**

配置简单  
独立运行  
自动装配  
无代码生成和 xml 配置  
提供应用监控  
易上手  
提升开发效率

1. **Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？**

1. 配置变更

2. JDK 版本升级

3. 第三方类库升级

4. 响应式 Spring 编程支持

5. HTTP/2 支持

6. 配置属性绑定

1. **spring boot 有哪些方式可以实现热部署？**

使用 devtools 启动热部署，添加 devtools 库，在配置文件中把 spring. devtools. restart. enabled 设置为 true；  
 使用 Intellij Idea 编辑器，勾上自动编译或手动重新编译。

1. **SpringBoot 的常用注解有哪些？**

@SpringBootApplication:

包含@Configuration、@EnableAutoConfiguration、@ComponentScan通常用在主类上；

@ComponentScan:组件扫描。个人理解相当于，如果扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean\*；  
@Configuration:指出该类是 Bean 配置的信息源，相当于XML中的，一般加在主类上；

@Bean:相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理；

@Import：用来导入其他配置类。

@ImportResource：用来加载xml配置文件。

@ControllerAdvice：包含@Component。可以被扫描到。统一处理异常。

@ExceptionHandler（Exception.class）：用在方法上面表示遇到这个异常就执行以下方法。

1. **SpringBoot 有哪几种读取配置的方式？**

方式一： 通过@Value("${spring.datasource.url}")这样的方式读取  
 方式二：通过@ConfigurationProperties(prefix = "spring.datasource")这种写法系统会依据prefix前缀自动注入配置数据到数据实体变量,这种方式不错,但是存在缺陷,我们编写的PropertyPlaceholderConfigurer扩展字段会无效,所以如果只是单纯的读取配置而不需要额外操作时可使用这方式最简单

方式三： 我们可以直接注入Environment对象示例并读取properties对象属性environment.getProperty("spring.datasource.database");与方式一的本质差不多,我们不需要编写对应字段的模型对象,但是对于程序可阅读性不好友,复用率不高

方式四： 通过系统启动时候初始化Listener,使用PropertiesLoaderUtils工具类读取指定配置文件并获得Properties配置对象,我们可以随时随地使用该对象的属性,这种方式比较少用,针对比较自定义的配置数据可使用该方式

1. **SpringBoot 配置加载顺序？**

在不指定要被加载文件时，默认的加载顺序：由里向外加载，所以最外层的最后被加载，会覆盖里层的属性，加载顺序依次为：

* 位于与jar包同级目录下的config文件夹，
* 位于与jar包同级目录下
* idea 环境下，resource文件夹下的config文件夹
* idea 环境下，resource文件夹下  （1->4, 外->里）

1. **Spring Boot 如何定义多套不同环境配置**

一、Spring Boot 环境设置机制

spring.profiles.active 属性可以为我们指定当前设置的环境，以此来选择我们的配置文件。例如我们有配置文件

application.yml

application-dev.yml

application-test.yml

application-prod.yml

当执行 java -jar xxx.jar --spring.profiles.actvie=test 此时，系统将启用 application.yml 和 application-test.yml配置文件。

当执行 java -jar xxx.jar --spring.profiles.actvie=prod 此时，系统将启用 application.yml 和 application-prod.yml 配置文件。

二、配置多环境

正如 第一 点所述，我们配置不同的配置文件

application.yml

application-dev.yml（开发环境）

application-test.yml（测试环境）

application-uat.yml（预发布环境）

application-prod.yml（生产环境）

三、指定环境

1 在 cmd 命令中指定

java -jar xxx.jar --spring.profiles.actvie=dev

2 在 application.yml 中指定

spring: profiles: active: dev

1. **Mybatis中使用#和$书写占位符有什么区别**

#{}传参能防止sql注入.

${}传参是字符串拼接.

1. **动态SQL**

* 所谓SQL的动态和静态,是指SQL语句在何时被编译和执行,二者都是用在SQL嵌入式编程中的.
* SQL语句的主体结构,在编译时尚无法确定,只有等到程序运行起来,在执行的过程中才能确定,这种SQL叫做动态SQL.
* 静态SQL语句的编译是在应用程序运行前进行的,编译的结果会存储在数据库内部.
* 程序运行时,数据库将直接执行编译好的SQL语句,降低运行时的开销.
* MyBatis中用于实现动态SQL的元素主要有:if、where、foreach.

1. **Mapper动态代理规范**

* xml映射文件中的namespace与mapper接口的全类名相同.
* mapper接口方法名和xml映射文件中定义的每个statement的id相同.
* mapper接口方法的输入参数类型和xml映射文件中定义的每个sql的parameterType的类型相同.
* mapper接口方法的输出参数类型和xml映射文件中定义的每个sql的resultType的类型相同.
* mybatis中的mapper动态代理是不支持方法重载的dao接口里的方法,因为是全类名+方法名的保存和寻找策略.
* mapper接口的工作原理是JDK动态代理,mybatis运行时就是用JDK动态代理为mapper接口生成代理proxy对象,代理对象proxy会拦截接口方法,转而执行mappedStatement所代表的sql,然后将sql执行结果返回.

1. **Mybatis常用注解**

@Insert ： 插入sql , 和xml insert sql语法完全一样

@Select ： 查询sql, 和xml select sql语法完全一样

@Update ： 更新sql, 和xml update sql语法完全一样

@Delete ： 删除sql, 和xml delete sql语法完全一样

@Param ： 当映射器方法需要多个参数时，指定参数的名字

@Results ： 结果集合

@Result ： 结果

@One ：复杂类型的单独属性值映射

@Many ：复杂类型的己合属性映射

1. **springcloud如何实现服务的注册和发现**

服务在发布时 指定对应的服务名（服务名包括了IP地址和端口） 将服务注册到注册中心（eureka或者zookeeper）

这一过程是springcloud自动实现 只需要在main方法添加@EnableDisscoveryClient  同一个服务修改端口就可以启动多个实例

调用方法：传递服务名称通过注册中心获取所有的可用实例 通过负载均衡策略调用（ribbon和feign）对应的服务

1. **ribbon和feign区别**

Ribbon添加maven依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用RestTemplate调用远程服务对应的方法

feign添加maven依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用 在接口上使用@FeignClient("指定服务名")

Ribbon和Feign的区别：Ribbon和Feign都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

1. **springcloud断路器的作用（Hystrix）**

当一个服务调用另一个服务由于网络原因或者自身原因出现问题时 调用者就会等待被调用者的响应 当更多的服务请求到这些资源时

导致更多的请求等待，这样就会发生连锁效应（雪崩效应）断路器就是解决这一问题

断路器有：

  完全打开：一定时间内达到一定的次数无法调用，并且多次检测没有恢复的迹象，断路器完全打开，那么下次请求就不会请求到该服务

  半开：短时间内有恢复迹象断路器会将部分请求发给该服务，当能正常调用时断路器关闭

  关闭：当服务一直处于正常状态能正常调用断路器关闭

**项目技术点:**

**MQ：**

**1.为什么使用MQ?**

* 使用MQ的好处:解耦，异步，削峰平谷;
* 解耦：
* 当A系统生产关键数据,而且B,C,D系统需要A系统给它们发送数据,来进行下一步操作,此时A系统和BCD系统产生了严重的耦合,所有的操作和维护都要在A系统中进行,如果将A系统产生的数据放到MQ当中,让BCD系统需要的时候去消费,此时就解放了A系统,不用考虑调用成功,失败超时等情况,同时ABCD系统独立运行,后续新添加系统需要A系统的数据,也不需要去修改A系统的代码,达到了解耦的效果
* 异步：
* 一般互联网类企业,对用户的直接操作,一般要求每个请求在200ms以内完成。对于一个系统调用多个系统,在不适用mq的情况下,它执行完返回的耗时,是执行完所有系统所需时间的总和;使用mq进行优化后,执行的耗时,则是执行主系统的耗时,以及加上主系统发送数据到消息队列的耗时,大幅度提升高延时接口的性能,提升了用户体验.
* 削峰平谷：
* 一般MySQL的每秒请求最高在2000左右，用户访问量高峰期的时候涌入的大量请求,很可能将MySQL给打死,然后系统就挂掉,但是高峰期过了,请求量可能远远低于2000,所以这种情况去增加服务器就不值得,如果使用mq的情况,将用户的请求全部放到mq中,然后让系统去消费用户的请求,不要超过系统所能承受的最大请求数量,保证系统不会再高峰期挂掉,但此时可能有几十万或几百万请求积压在mq中,但是高峰期一过,系统还是按照最大请求数量进行处理请求,很快就能将积压请求处理完

* 使用MQ的缺陷:
* 系统可用性降低:以前只要担心系统的问题,现在还要考虑mq如果挂掉的问题,因为mq一旦挂掉,所关联的系统,就会统统挂掉
* 系统复杂性变高:考虑的问题变多,要考虑消息丢失,消息重复消费的情况
* 一致性问题:比如A系统调用BCD系统,BCD同时成功才能执行成功,返回数据,现在BC执行成功,D发生异常的情况,并没有执行成功,但是A给用户返回的是成功

**2.你了解哪些MQ技术，ActiveMQ,RabbitMQ,RocketMQ,kafka**

* ActiveMQ它可以支持万级的吞吐量,它是一个比较成熟完善的中间件,但是它存在有少量信息丢失的情况,而且目前官方对他的更新迭代不是很即时,社区的活跃度不是很高
* RabbitMQ是一款用Erlang语言开发的消息中间件,它延时低,唯一一款达到微妙级延时的消息中间件,而且社区活跃度高,对于bug问题的修复很及时,而且提供了很友善的后台界面,唯一的劣势是就是我们搞java开发的很少有人能够阅读它的源码,对于问题的修复仅仅依靠社区。
* RocketMQ是阿里旗下的一款品质优秀的MQ,它可以达到十万级的吞吐量,而且它还是支持分布式事务,应用于分布式架构,它的维护基本上都是靠阿里,如果哪天阿里宣布放弃这个项目的维护,除非你们公司有能力继续来维护RocketMQ的使用。
* Kafka也是一款分布式的中间件,最大优点就是其吞吐量高,一般运用于大数据系统的实时运算和日志采集的场景,功能简单,可靠性高,扩展性高,唯一的缺点是可能导致重复消费,但是这点轻微的影响在大数据场景下可以忽略。

**3.RabbitMQ 的使用场景有哪些？**

抢购活动，削峰填谷，防止系统崩塌。

延迟信息处理，比如 10 分钟之后给下单未付款的用户发送邮件提醒。

解耦系统，对于新增的功能可以单独写模块扩展，比如用户确认评价之后，新增了给用户返积分的功能，这个时候不用在业务代码里添加新增积分的功能，只需要把新增积分的接口订阅确认评价的消息队列即可，后面再添加任何功能只需要订阅对应的消息队列即可。

**4.如何保证MQ的高可用?**

* 非分布式的MQ的高可用:
* 普通集群模式:实际queue保存在一个节点上,其他节点保存该queue的元数据,消费者连接某个节点,该节点去调用实际存放queue的节点,返回数据,说白了只是简单的提高了吞吐量而已.

缺陷在于:在集群内部产生大量的数据传输,高可用性没有,queue所在的节点宕机,服务就没法继续使用

* 镜像集群模式:每个节点上都保存了实际的queue的全部数据,消费者无论连接哪个节点,都能直接获取到数据.任何一个节点宕机,其他节点还存在queue的全部数据,消费者换个节点继续消费,这是高可用的模式
* 分布式MQ的高可用:
* 假设生产者往topic里写了三条数据,每条数据放在一个partition中,每个partition部署在一台机器中,每个partition都有一个或多个replica副本,其中有一个是leader,其他的是follower,只有leader能够向外提供数据的读写,生产者写入消息时,leader负责向follower同步消息,此时高可用架构就已经体现,如果一台leader宕机,会从它的follower中选举一个leader,继续向外提供服务

**5.如何保证消息不被重复消费(如何保证消息消费时的幂等性)?**

* 重复消费:
* 每个中间件它能够保证的是消息不丢失,但不能保证消息不被重复发送，所以接收方就要做幂等性的判断，防止消息重复消费，例如：支付金额的消息被消费了两次，那么所得到的钱数就是错误的，这种情况在系统中肯定是不允许存在的。
* 消费的幂等性（同样的操作，在一段时间内，只执行一次）:
* 如果是往数据库里写入数据,就根据主键查一下,如果数据已经存在,就update
* 可以在把数据先存在set或者redis中,消费前,先去里面查看,数据是否已存在,已存在就丢弃这数据,比如说我们的订单系统,生成订单也会用到mq,此时订单id就是全局唯一的id,在写入数据库之前,就可以先把数据去redis中查询,如果redis中已经存在,则不进行消费操作,如果redis中不存在的话,就存在redis中,然后进行下一步操作.
* 在数据库中设置唯一约束,就不会导致重复数据的多次插入

**6.如何保证消息不丢失**

* rabbitMQ:
* 消息丢失的情况:生产者写的消息在到中间件的网络传输过程中就丢了,或者是消息到了中间件,但是内部出错,消息没保存下来
* 中间件将消息保存下来,还没等到消费者消费完,就自己挂掉,导致消息丢失
* 消费者取到消息还没来得及消费就自己挂掉了,因为rabbitMQ消费者开启了autoAck,在消费数据还没成功时,就已经向中间件发送完成的信息,此时消费者挂掉,就会消息丢失
* 解决方案:
* 生产者消息丢失,可以通过开启事务功能,如果消息没有发送成功,发生异常就回滚,然后重试发送消息,发送成功就提交事务,这个的缺陷就是阻塞式的,降低吞吐量,耗费性能;如果是rabbitMQ可以开启confirm模式,它能给每次写的消息都分配一个唯一的id,如果写入到rabbitMQ中,rabbitMQ就会回传一个ack消息,如果没有就会会挑一个nack接口,告诉你消息接收失败,你可以重试,confrim机制是异步的,效率会高很多
* 关于中间件的数据丢失,可以开启中间件的持久化,将消息持久化磁盘中,中间件挂了恢复之后自动读取之前存储的数据.
* 消费者数据丢失,关闭rabbitMQ的autoACK机制,自己手动提交完成信息

**7.如何保证消息的顺序性**

* rabbitMQ为多个消费者开辟多个queue队列（先进先出）,将保证操作顺序的消息发布到同一个队列中去，操作这个队列的消费者会一个一个消息去处理，因为队列这种结构是先进先出的类型，所以保证的数据的顺序性。

**8.消息大量积压怎么解决？**

* 临时启动多个消息者，并发处理消息
* 临时启动多个消息者，接受消息之后，不处理。暂时把消息写到文件中。消息中间件中的消息处理的完了。关闭临时消费者。单独写个离线程序，处理文件中的消息
* 临时启动多个消息者，接受消息之后，直接丢弃。 可以让生产者的源头恢复数据

**9.RabbitMQ 节点的类型有哪些？**

* 磁盘节点：消息会存储到磁盘。
* 内存节点：消息都存储在内存中，重启服务器消息丢失，性能高于磁盘类型。

**10.rabbitmq如何确认消息一定发送到了消息中间件中呢?**

消息发送到server是先通过交换机,再到消息队列,只要数据到队列,那么此消息肯定是发送成功的.

第一步确定消息发送到了交换机.

使用发送方确认机制来判断消息是否发送成功.

第二步确认交换机把消息路由到消息队列.

使用失败回调来判断消息是否发送成功.

只有两步都成功,此消息才是发送成功的.

**11.rabbitmq的集群**

参考网站:docker中搭建rabbitmq集群-> https://www.cnblogs.com/vipstone/p/9362388.html

集群分为普通集群和镜像集群.

普通集群: 普通集群,他会把所有节点的交换机信息和队列的元数据分为两种(队列数据分为两种: 一种为队列里面的消息,另一种是队列本身的信息, 后者被称为元数据.)进行复制,确保所有的节点都有一份.

镜像集群: 在普通集群的基础上,把所有的队列数据完全同步(对性能有一定的影响)当对数据可靠性要求高时,可以使用镜像模式.

镜像集群实现由两种方式: 一种是直接在管理台控制,

一种是在声明队列的时候控制.

配置集群还设计到节点信息, 有内存节点和磁盘节点,如果对队列有修改的情况下,必须有磁盘节点,用来保存信息,内存节点断电后,信息就消失,无法保存. 默认就是磁盘节点, 设置 --ram为内存节点.

# Redis：

1. **使用redis缓存的好处**

* 提高并发量，不需要每次都访问数据库，提高并发量。
* 提高性能，redis运行在内存中，内存的执行效率，远远超过数据库。

1. **Redis的key和value的存储大小有限制吗？**

* Redis的key和value的存储大小都是有限制的，都是512M。

1. **redis存储什么数据类型**
2. 字符串（String）
3. 字符串列表（lists）
4. 字符串集合（sets）
5. 有序字符串集合（sorted sets）
6. 哈希（hash）
7. **使用redis缓存的弊端**

* 缓存穿透，大量的请求访问,查询的是数据库中不存在的数据,就会在缓存中无法命中,直接去访问数据库,数据库中也查询不到,自然无法将结果写入缓存,下一秒又有大量请求查询不存在的数据,导致数据库最终挂掉(要么系统出bug,要么黑客恶意攻击)

解决方案:

* 在数据库中没有查询到的数据,存储一个特定值到缓存中,这样下次恶意访问就可以去缓存中查询到数据,不会访问到数据库.注意的是这些特定的key,需要设置过期时间，避免黑客攻击时，大量的无效key把redis存满。
* 缓存雪崩，大规模的key失效，大量的请求通过key访问到数据库，导致数据库直接崩溃,然后系统直接瘫痪.

解决方案:

* 事前:保证redis集群的高可用性,redis cluster,主从机制;
* 事中:hystrix（熔断器）限流+降级,避免数据库被打死.高并发的访问量走hystrix限流组件,让访问量保持在数据库最大负载范围内,保证数据库不被打死,系统可以继续使用.剩余的请求走降级组件,返回一些默认的值或者是友情提示等
* 事后:redis持久化机制,尽快回复缓存集群,一旦重启,自动从磁盘上加载数据,回复内存中的数据
* 缓存与数据库不一致，修改数据时，数据库和缓存数据不一致。

最初级的缓存不一致:

出现场景：

* 先修改数据库,再删除缓存,如果修改缓存失败,就会导致数据库中式新数据,缓存中是旧数据,数据不一致

解决思路:

* 先删除缓存,再修改数据库,如果删除缓存成功,修改数据库失败,那么数据库中是旧数据,缓存中是空的,读的时候缓存中没有,去数据库中读旧数据,然后更新到缓存中去

高并发场景下数据库与缓存的数据不一致:

出现场景：

* 读写并发请求,导致数据库和缓存中的数据不一致,在写请求删除缓存,修改数据库库存还未成功时,查询库存的请求就发来,先去缓存中查询,发现是空,然后去数据库中查,然后将结果放入缓存中,然后修改库存的操作成功,导致数据库中是新数据,缓存中是旧数据

解决思路:

* 相同的商品id,进行哈希取值,再加上对内存队列的数量进行取模,每个商品都可以路由到某一个内存队列中,然后将去请求和写请求串行化,这样就可保证一定不会出现不一致的情况,但是会导致系统的吞吐量会大幅降低
* 缓存并发竞争，同一时间，多个线程来执行，操作同一个key。

解决思路:

* 方式一：分布式锁,（zookeeper分布式锁），确保同一时间,只能由一个系统实例在操作某个key,别的实例不被允许读和写
* 方式二：乐观锁 ，每次要写之前,先判断这个value的时间戳是否比缓存里的更新,如果新就允许写.

1. **为什么redis是单线程的但是还可以支撑高并发?**

* redis线程模型:redis里有个处理器叫做文件事件处理器(file event handler),这个文件事件处理器是单线程的执行的,redis所以是单线程的模型。

执行流程:

* redis启动初始化时,redis会将连接应答器和AE\_READABLE事件关联
* 客户端跟redis连接,会在serverSocket产生一个AE\_READABLE事件,然后由IO多路复用程序监听到事件,将serverSocket压入队列,然后连接应答处理器来处理事件,和客户端建立连接,创建对应的socket,同时将这个socket的AE\_READABLE事件跟命令请求处理器关联起来
* 客户端向redis发起请求时,首先会在socket中产生一个AE\_READABLE事件,然后被监听,被压入队列,然后由对应的命令请求处理器来处理
* redis准备好给客户端响应数据之后,就会将socket的AE\_WRITABLE命令跟回复处理器关联起来,当客户端准备好读取响应数据时,就会在socket上产生一个AE\_WRITABLE事件,会由对应的命令回复处理器来处理
* 命令回复处理器处理完之后,就会删除这个socket的AE\_WRITABLE事件和命令回复处理器的关联关系

1. **Redis为什么效率高?**

* 核心是基于非阻塞的IO多路复用机制,由监听程序轮询等待的事件,然后压入队列,可以达到一个线程同时处理多个io请求的目的
* 纯内存操作
* 单线程避免了多线程的频繁切换问题

1. **Redis由哪些数据类型,分别在哪些场景下使用?**

* String,存储基本的类型
* List,有序列表,比如之前项目中的广告缓存,用的就是list集合进行缓存的,它可以用于比如存储粉丝列表,文章的评论列表等等
* Set,无序列表,它最主要的特点就是去重,比如微博上查询两个明星的共同粉丝,就可以用set进行去重
* SortSet,它主要是在set的基础上加上了排序功能
* Hash,类似map的集合,一般可以用来存储对象,我们系统中的购物车对象就是以hash类型存储在redis中,key就是用户的id,value就是购物车对象

1. **Redis过期策略**

* 定期删除配合惰性删除
* 定期删除,指的是redis默认每个100ms就抽取一些设置了过期事件的key,检查是否过期,如果过期就删除.如果太多的key都设置了过期时间,每隔几百毫秒,就全部检查删除,cpu的负载会很高.实际上redis是每隔100ms随机抽取一些key来检查和删除.
* 惰性删除,就是当你获取key时,redis就会检查一下,如果key过期了,就删除,不会返回,如果没过期就返回
* 因为redis的过期策略会导致你设置了过期key,过期了很多过期key没有被删除,还堆积在内存中,此时就要设置内存淘汰机制来解决了
* 内存淘汰机制
* noevication:当内存不足以容纳写入新数据时,新写入操作就会报错
* allkeys-lru:当内存不足以容纳新写入数据时,在键空间中,移除最近最少使用的key
* allkeys-random:在键空间中,随机移除某个key
* volatile-lru:在设置了过期时间的键空间中,移除最近最少使用的key
* volatile-random:在设置了过期时间的键空间中,随机移除某个key
* volatile-ttl:在设置了过期时间的键空间中,由更早过期时间的key优先移除

1. **怎样做redis支撑高并发(读多写少用缓存,读少写多用队列)**

* 搭建redis集群,做成主从架构,一主多从,通过redis replication机制实现读写分离,主负责写,并且将数据同步复制到其他slave（从）节点上,从节点负责读,所有的读请求走从节点.好处,可支持水平扩容的读高并发架构
* redis replication的核心机制
* redis采用异步方式复制数据都slave节点上,slave节点会周期性的确认自己每次复制的数据量
* 一个master（主）节点可以配置多了slave节点
* slave节点也可以连接其他slave节点
* slave节点做复制的时候,也不会阻塞对自己的查询操作,它会用旧数据向外提供服务;复制完后,它就删除旧数据,加载新数据,这个时候向外停止服务(毫秒级别)
* slave节点主要用来横向扩容,做读写分离,提高吞吐量
* Master（主节点）持久化对于主从架构的安全保障的意义
* 如果仅仅用slave节点作为master节点的数据热备份,当master节点宕机后重启,数据清空,本地没有数据可以恢复,它会任为自己的数据就是空的,然后同步到slave节点上,就会导致数据百分百丢失
* 其次,如果开启了哨兵机制,在master节点挂掉之后,会迅速推选出一个slave节点去做master节点,但是当哨兵还没检测到master节点挂掉,master节点就自动重启了,还是会出现上面的情况

* redis主从架构的核心原理
* 当启动一个slave节点时,会发送一个PSYNC命令给master节点
* 如果这个slave节点第一次连接master节点,就会出发full resynchronization
* 开始full resynchronization的时候,master会开启一个后台线程,开始生成一份rdb快照文件,同时还还会继续接收客户端的写命令,存在内存中.RDB文件生成完毕后,master会将这个RDB发送给slave,slave先将RDB写入本地自盘,然后再加载到内存.然后master会将内存中缓存的写命令发送给slave,slave同步这些数据
* redis的高可用
* 不可用:虽然主从架构,读写分离可以保证缓存应对高并发场景,但是如果master节点挂掉,整个缓存架构就会挂掉,大量的流量涌入数据库,超过数据库的最大承载压力,数据库挂掉,整个系统就会挂掉不可用
* 高可用:故障转移,主备切换,当master故障时,自动检测(通过sentinal node 哨兵),将某个slave自动切换成master的过程交主备切换.这个机制实现了redis主从架构下的高可用性

1. **redis的持久化**

* 如果没有持久化,遇到灾难性故障时,就会丢失所有数据，如果开启持久化到磁盘,定期备份到云服务上,就能保证遇到灾难性故障,就不会丢失全部数据
* RDB
* 当redis需要做持久化时,redis会fork一个子进程,将数据写到磁盘写上一个临时RDB文件中,当子进程完成写临时文件后,将原来的RDB换掉
* 优势:

适合做冷备份,在最坏的情况下,恢复数据要比AOF快

对redis对外提供读写服务,影响服务非常小

RDB数据快照文件更少

* AOF
* 可以做到更精细的持久化,redis每执行一个修改数据的命令,都会将它添加到os cache中,一般会每个一秒执行一次fsync操作,保证将os cache中的数据写入磁盘中AOF日志文件中,AOF日志文件以append-only模式写入,文件不容易破损,在AOF日志过大时,就会出现后台重写
* 优势:

AOF可以更好的保护数据不丢失,丢失的数据更少

AOF日志文件以append-only模式写入,没有磁盘寻址的开销,写入性能非常高

AOF日志过大时,也不会影响客户端的读写

* RDB和AOF到底如何抉择
* 综合使用两种持久化机制,用AOF保证数据不丢失,作为数据恢复的第一选择,用RDB来做不同程度的冷备份,在AOF文件丢失或者损坏不可用的情况下,用RDB来进行快速的数据回复

1. **redis怎么设置缓存大小**

打开redis配置文件

示例：maxmemory 100mb

单位：mb,gb。

默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一点要设置。

设置maxmemory之后，配合的要设置缓存数据回收策略, 可以通过设置LRU算法来删除部分key，释放空间(LRU是Least Recently Used 近期最少使用算法。)。

就可以修改缓存大小为16gb.

1. **redis集群主从数据如何同步**

答1:

 Redis主从同步有两种方式：全同步和部分同步。主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。

答2:

1. 全量同步：  
   Redis全量复制一般发生在S-ave初始化阶段，这时S-ave需要将Master上的所有数据都复制一份。具体步骤如下：  
   　　1）从服务器连接主服务器，发送SYNC命令；  
   　　2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；  
   　　3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；  
   　　4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；  
   　　5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；  
   　　6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；  
   (2)增量同步：  
   Redis增量复制是指S-ave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。  
   增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令  
   (3)Redis主从同步策略：  
   主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，s-ave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。
2. **Redis缓存穿透和redis雪崩处理**

**(1)缓存穿透**

​ 一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，应该去后端系统查询（比如数据库）。如果key对应的value不存在，并且对key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

**解决方案(晦涩)**  
对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。  
也可以采用一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

**(2)缓存雪崩**

​ 当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统（比如数据库）带来很大的压力。

​ 缓存雪崩是指在我们设置缓存是采用了相同的过期时间，导致缓存在某一时期同时失效，请求全部转发到DB，DB瞬间压力过大雪崩。

**​ 解决方案：**

​ 把缓存时间分散开，比如在原来的时间基础上增加一个随机值，比如1~5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

**Elasticsearch**

**1.Elasticsearch是如何实现Master选举的**

* Elasticsearch的选主是ZenDiscovery模块负责的，主要包含Ping（节点之间通过这个RPC来发现彼此）和Unicast（单播模块包含一个主机列表以控制哪些节点需要ping通）这两部分；
* 对所有可以成为master的节点（node.master: true）根据nodeId字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第0位）节点，暂且认为它是master节点。
* 如果对某个节点的投票数达到一定的值（可以成为master节点数n/2+1）并且该节点自己也选举自己，那这个节点就是master。否则重新选举一直到满足上述条件。

**2.在并发情况下，Elasticsearch如果保证读写一致？**

* 可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；
* 另外对于写操作，一致性级别支持quorum/one/all，默认为quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。
* 对于读操作，可以设置replication为sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置replication为async时，也可以通过设置搜索请求参数\_preference为primary来查询主分片，确保文档是最新版本。

**3.Elasticsearch索引文档的过程**

协调节点默认使用文档ID参与计算（也支持通过routing），以便为路由提供合适的分片。  
　　shard = hash(document\_id) % (num\_of\_primary\_shards)  
　　当分片所在的节点接收到来自协调节点的请求后，会将请求写入到Memory Buffer，然后定时（默认是每隔1秒）写入到Filesystem Cache，这个从Momery Buffer到Filesystem 　　Cache的过程就叫做refresh；  
　　当然在某些情况下，存在Momery Buffer和Filesystem Cache的数据可能会丢失，ES是通过translog的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到translog中，当Filesystem cache中的数据写入到磁盘中时，才会清除掉，这个过程叫做flush；  
　　在flush过程中，内存中的缓冲将被清除，内容被写入一个新段，段的fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的translog将被删除并开始一个新的translog。  
　　flush触发的时机是定时触发（默认30分钟）或者translog变得太大（默认为512M）时；

**4.Elasticsearch更新和删除文档的过程**

删除和更新也都是写操作，但是Elasticsearch中的文档是不可变的，因此不能被删除或者改动以展示其变更；  
　　磁盘上的每个段都有一个相应的.del文件。当删除请求发送后，文档并没有真的被删除，而是在.del文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del文件中被标记为删除的文档将不会被写入新段。  
　　在新的文档被创建时，Elasticsearch会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

**5.Elasticsearch搜索的过程**

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；  
　　在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。 每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询Filesystem Cache的，但是有部分数据还在Memory Buffer，所以搜索是近实时的。  
　　每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。  
　　接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。  
　　补充：Query Then Fetch的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch增加了一个预查询的处理，询问Term和Document frequency，这个评分更准确，但是性能会变差。

**6.ElasticSearch中的集群、节点、索引、文档、类型是什么？**

　　群集是一个或多个节点（服务器）的集合，它们共同保存您的整个数据，并提供跨所有节点的联合索引和搜索功能。群集由唯一名称标识，默认情况下为“elasticsearch”。此名称很重要，因为如果节点设置为按名称加入群集，则该节点只能是群集的一部分。

　　节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。

　　索引就像关系数据库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片，并且可以有零个或多个副本分片。 MySQL =>数据库ElasticSearch =>索引

　　文档类似于关系数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构（字段），但是对于通用字段应该具有相同的数据类型。 MySQL => Databases =>Tables => Columns / Rows ElasticSearch => Indices => Types =>具有属性的文档

类型是索引的逻辑类别/分区，其语义完全取决于用户。

**7.elasticsearch 索引数据多了怎么办，如何调优**

使用bulk API  
 初次索引的时候，把 replica 设置为 0  
 增大 threadpool.index.queue\_size  
 增大 indices.memory.index\_buffer\_size  
 增大 index.translog.flush\_threshold\_ops  
 增大 index.translog.sync\_interval  
 增大 index.engine.robin.refresh\_interval

**Dubbo和Zookeeper：**

1. **dubbo的工作流程**

* provider向注册中心去注册自己为一个服务
* consumer去注册中心订阅服务,注册中心会通知consumer注册好的服务,consumer会将provider的地址等信息拉取到本地缓存
* consumer去调用provider
* consumer和provider都异步的通知监控中心

1. **Dubbo的通信原理？**

* Dubbo底层使用 hessain2进行二进制序列化进行远程调用
* Dubbo底层使用 netty框架进行异步通信。NIO

1. **dubbo负载均衡策略有哪些?**

* random loadbalance
* 默认情况下,dubbo是random loadbalance随机调用实现负载均衡,可以对provider不同实例设置不同的权重,会按照权重来负载均衡,权重大分配的流量高
* roundrobin loadbalance
* 默认就是均匀地将流量达到各个机器上,值得注意的是,要根据机器的性能,调整权重。
* leastactive loadbalance
* dubbo自动感知,如果某个机器性能越差,那么接收的请求就越少,给不活跃的机器更少的请求
* consistanthash loadbalance
* 一致性hash算法,相同参数的请求一定分发到同一个provider上,provider挂掉的时候,会基于虚拟机节点均匀分配甚于的流量,抖动不会太大.适用于订单,同一个订单的操作,分配到同一个机器上,这样就可以避免高并发场景下,数据库和缓存中数据不一致的情况

**3.zookeeper注册中心挂了，dubbo可以继续使用吗？**

可以,因为刚开始dubbo初始化的时候,消费者会将提供者的地址等信息缓存到dubbo,注册中心挂了dubbo可以继续通信

**4.zookeeper 都有哪些功能？**

* 集群管理：监控节点存活状态、运行请求等。
* 主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 zookeeper 可以协助完成这个过程。
* 分布式锁：zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。zookeeper可以对分布式锁进行控制。
* 命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

**5.Zookeeper的watch机制（通知机制）**

Zookeeper系统中其实类似于window系统中的文件夹，zookeeper可以建立普通节点和临时节点，但是每个节点只能有一个，例如：/a/b/c节点已经存在，那么再来申请/a/b/c的节点就会失败。此时可以设置watch此节点，当/a/b/c节点删除后，会通知watch此节点的所有线程，线程再来创建节点，完成设定的任务。（分布式锁，高可用性）

**6.zookeeper 怎么保证主从节点的状态同步？**

zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 zab 协议。 zab 协议有两种模式，分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

**7.什么是 zab 协议**

ZAB 是 Zookeeper 原子广播协议的简称

整个ZAB协议主要包括消息广播和崩溃恢复两个过程，进一步可以分为三个阶段，分别是：

发现 Discovery  
 同步 Synchronization  
 广播 Broadcast

组成ZAB协议的每一个分布式进程，都会循环执行这三个阶段，将这样一个循环称为一个主进程周期。

**8.如果zookeeper服务挂了怎么办？**

注册中心对等集群，任意一台宕掉后，会自动切换到另一台

注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯

服务提供者无状态，任一台宕机后，不影响使用

服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

**分布式系统：**

1. **怎么实现远程通信**

什么是远程通信:简单来说，就是一个系统去调用另一个系统中的数据.

常见的有三种方式：

(1)Webservice的方式：

    1)优点：跨语言跨平台

    2)缺点：它是基于soap协议的，使用http+xml的方式进行数据传输，http是应用层协议，传输效率不是很高，而且xml的解析也比 较费时，所以项目内部进行通信的时候，不建议使用Websservice的方式

(2)restful形式的服务：

   1)优点：restful本身就是http，使用的是http+json的方式进行数据传输，因为json数据本身是非常简洁的，所以它比webservice的 传输效率更高；手机app端一般都使用该方法，其他很多项目也是用这种方式

    2)缺点：如果服务太多的话，会出现服务之间调用关系混乱，此时就需要治理服务

(3)Dubbo：

     使用的是RPC协议进行远程调用，RPC协议是一个二进制协议，直接使用的socket进行通信，传输效率高，并且可以统计出系统 之间的调用关系和调用次数系统分布式SAO系统的内部通信推荐使用dubbo

1. **分布式事务：**

TCC方案(Try,Confirm,Cancel)

* Try:此阶段是对各个服务的资源做检测以及对资源进行锁定或者预留
* Confirm:此阶段是在各个服务中执行实际的操作
* Cancel:如果任何一个服务的业务执行出错,就要进行补偿,将已经执行成功的业务逻辑回滚操作
* 这种方案的事务回滚实际上是依赖于自己写代码来回滚和补偿,补偿代码巨大,除了一些严格保证分布式事务的场景,比如和资金,订单相关的业务,尽量少使用

本地消息表(ebay)

* A系统在自己本地一个事务里操作同时,插入一条数据到消息表
* A系统将这个消息发送到MQ中
* B系统接收到消息之后,在一个事务里,往自己本地消息表中插入一条数据,同时执行其他业务操作,如果这个消息已经被处理了,那么此时这个消息会回滚,这样保证不会重复处理消息
* B系统执行成功后,就会更新本地消息表的状态以及A系统消息表的状态
* 如果B系统处理失败了,就不会更新消息表状态,A系统会定时扫描自己的消息表,没有处理的消息,会再次发送到MQ中,让B处理
* 这个方案为了保证最终一致性,哪怕B事务失败了,A会不断重发消息,直到B成功为止;此外最大的问题是严重依赖数据库的消息表来管理事务,不适用高并发的场景,扩展性能差

可靠消息最终一致性方案

* 这个方案是基于阿里的rocketMQ(3.3.6之前)来实现,是目前比较国内比较常用的方案
* A系统发送一个prepared消息到mq,如果prepared消息发送失败,就直接取消操作
* 如果消息发送成功,就接着执行本地事务,如果成功就给mq发送确认消息,如果失败就告诉mq回滚消息
* 如果发送了确认消息,B系统就会接收到,然后执行本地的事务
* mq会定时轮询所有prepared消息,回调接口,所有没发送确认消息的,问你继续重试还是回滚,然后A系统就去查看本地事务的状态,如果回滚,告诉mq回滚消息,如果成功了就重新发送确认消息
* 如果B系统的事务失败,自动不断重试,直到成功,如果实在不行就让B系统本地回滚,然后想办法通知A系统也回滚,或者是发送警报由人工手动回滚和补偿

最大努力通知方案

* A系统本地事务执行完后,发送消息到MQ
* 有个服务会消费MQ中的消息,然后写入数据库记录下来,或者放入内存队列中,接着调用B系统的接口
* 要是B系统执行成功就ok,执行失败,就最大努力通知服务定时尝试重新调用B系统,反复N次,实在不行就放弃

1. **系统的高并发问题是怎么解决的**

并发问题高，这个问题的解决方案是一个系统性的，系统的每一层面都需要做优化：

1） 数据层

a) 集群

b) 分表分库

c) 开启索引

d) 开启缓存

e) 表设计优化

f) Sql语句优化

g) 缓存服务器（提高查询效率，减轻数据库压力）

h) 搜索服务器（提高查询效率，减轻数据库压力）

2）项目层

a) 采用面向服务分布式架构（分担服务器压力，提高并发能力）

b) 采用并发访问较高的详情系统采用静态页面

c) 使用页面缓存

d) 用 ActiveMQ使得业务进一步进行解耦，提高业务处理能力

e) 使用分布式文件系统存储海量文件

3） 应用层

a) Nginx服务器来做负载均衡

b) Lvs做二层负载

1. **分布式锁的问题**

针对分布式锁的实现，目前比较常用的有以下几种方案：

1.基于数据库实现分布式锁

2.基于缓存（redis，memcached，tair）实现分布式锁

3.基于 zookeeper实现分布式锁