# **一、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。

## 2.== 和 equals 的区别是什么？

**== 解读**

对于基本类型和引用类型 == 的作用效果是不同的，如下所示：

1. 基本类型：比较的是值是否相同；
2. 引用类型：比较的是引用是否相同；

String x = "string";

String y = "string";

String z = new String("string");

System.out.println(x==y); // true

System.out.println(x==z); // false

System.out.println(x.equals(y)); // true

System.out.println(x.equals(z)); // true

代码解读：因为 x 和 y 指向的是同一个引用，所以 == 也是 true，而 new String()方法则重写开辟了内存空间，所以 == 结果为 false，而 equals 比较的一直是值，所以结果都为 true。

**equals 解读**

equals 本质上就是 ==，只不过 String 和 Integer 等重写了 equals 方法，把它变成了值比较。看下面的代码就明白了。

首先来看默认情况下 equals 比较一个有相同值的对象，代码如下：

class Cat {

    public Cat(String name) {

        this.name = name;

    }

    private String name;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}

Cat c1 = new Cat("王磊");

Cat c2 = new Cat("王磊");

System.out.println(c1.equals(c2)); // false

**总结** ：== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；而 equals 默认情况下是引用比较，只是很多类重写了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

## **3.** **两个对象的 hashCode()相同，则 equals()也一定为 true，对吗？**

不对，两个对象的 hashCode()相同，equals()不一定 true。

String str1 = "通话";

String str2 = "重地";

System.out.println(String.format("str1：%d | str2：%d",  str1.hashCode(),str2.hashCode()));

System.out.println(str1.equals(str2));

代码解读：很显然“通话”和“重地”的 hashCode() 相同，然而 equals() 则为 false，因为在散列表中，hashCode()相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能得出键值对相等。

## **4.**       **final 在 java 中有什么作用？**

final 修饰的类叫最终类，该类不能被继承。

final 修饰的方法不能被重写。

final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

## **5.java 中的 Math.round(-1.5) 等于多少？**

等于 -1

## **6.String 属于基础的数据类型吗？**

String 不属于基础类型，基础类型有 8 种：byte、boolean、char、short、int、float、long、double，而 String 属于对象。

## **7.java 中操作字符串都有哪些类？它们之间有什么区别？**

操作字符串的类有：String、StringBuffer、StringBuilder。

String 和 StringBuffer、StringBuilder 的区别在于 String 声明的是不可变的对象，每次操作都会生成新的 String 对象，然后将指针指向新的 String 对象，而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 String。

StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

## **8.String str="i"与 String str=new String(“i”)一样吗？**

不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

## **9.如何将字符串反转？**

使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。

## **10.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。

## **11.抽象类必须要有抽象方法吗？**

不需要，抽象类不一定非要有抽象方法

abstract class Cat {

    public static void sayHi() {

        System.out.println("hi~");

    }

}

## **12.普通类和抽象类有哪些区别？**

1. 普通类不能包含抽象方法，抽象类可以包含抽象方法。
2. 抽象类不能直接实例化，普通类可以直接实例化。

## **13.抽象类能使用 final 修饰吗？**

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼此就会产生矛盾，所以 final 不能修饰抽象类

## **14.接口和抽象类有什么区别？**

实现：抽象类的子类使用 extends 来继承；接口必须使用 implements 来实现接口。

构造函数：抽象类可以有构造函数；接口不能有。

main 方法：抽象类可以有 main 方法，并且我们能运行它；接口不能有 main 方法。

实现数量：类可以实现很多个接口；但是只能继承一个抽象类。

访问修饰符：接口中的方法默认使用 public 修饰；抽象类中的方法可以是任意访问修饰符。

## **15.java 中 IO 流分为几种？**

按功能来分：输入流（input）、输出流（output）。

按类型来分：字节流和字符流。

字节流和字符流的区别是：字节流按 8 位传输以字节为单位输入输出数据，字符流按 16 位传输以字符为单位输入输出数据。

## **16.BIO、NIO、AIO 有什么区别？**

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。

NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

## **17.Files的常用方法都有哪些？**

Files.exists()：检测文件路径是否存在。

Files.createFile()：创建文件。

Files.createDirectory()：创建文件夹。

Files.delete()：删除一个文件或目录。

Files.copy()：复制文件。

Files.move()：移动文件。

Files.size()：查看文件个数。

Files.read()：读取文件。

Files.write()：写入文件。

## Java中的值传递和引用传递有什么区别？

按值传递：指的是在方法调用时，传递的参数是按值的拷贝传递。传递的是值的拷贝，也就是说传递后就互不相关了。

引用传递：指的是在方法调用时，传递的是引用的地址，也就是变量所对应的内存空间的地址。，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。

# **Java 容器**

## **18.java 容器都有哪些？**

数组,String,java.util下的集合容器

数组长度限制为 Integer.Integer.MAX\_VALUE;

String的长度限制: 底层是char 数组 长度 Integer.MAX\_VALUE 线程安全的

List:存放有序,列表存储,元素可重复

Set:无序,元素不可重复

Map:无序,元素可重复

分别从以下点进行对比

1.有无顺序

2.元素是否可重复

3.可存放元素数量

4.底层实现

5.线程安全性

6.通常用来做什么

7.优点，特性

8.线程安全框架包

9.在现有哪些框架上用到了这些集合

Set底层HashMap实现

HashMap底层用数组+链表实现

LinkedHashMap,TreeHashMap底层用额外的链表和树进行维护

HashMap的线程安全包:ConcurrentHashMap

ConcurrentHashMap1.7和1.8的区别

JUC包各种知识点。

## **19.Collection 和 Collections 有什么区别？**

Collcetion是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法。实现该接口的类主要有List和Set，该接口的设计目标是为各种具体的集合提供最大化的统一操作方式。

Collections是针对集合类的一个包装类，它提供了一系列的静态方法以实现对各种集合的搜索、排序、线程安全化等操作，其中大多数方法都是用来处理线性表。 Collections类不能实例化，如同一个工具类，服务于Collection。若是在使用Collections类的方法时候，对应的collection的对象为null，则这些方法都会抛出 NullPointerException 。之前我们见到的包装类还有Arrays，它是为数组提供服务的。

## **20.List、Set、Map 之间的区别是什么？**

总结list：

　　1.值可重复、允许多个null元素。

　　2.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。

　　3.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

总结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

总结map：

　　1.Map不是collection的子接口，Map是独立的一个接口。

　　2.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。

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

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

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

## **21.HashMap 和 Hashtable 有什么区别？**

**1、继承的父类不同**

Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口

**2、线程安全性不同**

Hashtable线程是安全的，hashmap线程是不安全的；

**3、是否提供contains方法**

      HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey，因为contains方法容易让人引起误解。

      Hashtable则保留了contains，containsValue和containsKey三个方法，其中contains和containsValue功能相同。

我们看一下Hashtable的ContainsKey方法和ContainsValue的源码：

**4、key和value是否允许null值**

      其中key和value都是对象，并且不能包含重复key，但可以包含重复的value。

      通过上面的ContainsKey方法和ContainsValue的源码我们可以很明显的看出：

      Hashtable中，key和value都不允许出现null值。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，可能是 HashMap中没有该键，也可能使该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键， 而应该用containsKey()方法来判断。

**5、两个遍历方式的内部实现上不同**

      Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

**6、hash值不同**

      哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

      hashCode是jdk根据对象的地址或者字符串或者数字算出来的int类型的数值。

      Hashtable计算hash值，直接用key的hashCode()，而HashMap重新计算了key的hash值，Hashtable在求hash值对应的位置索引时，用取模运算，而HashMap在求位置索引时，则用与运算，且这里一般先用hash&0x7FFFFFFF后，再对length取模，&0x7FFFFFFF的目的是为了将负的hash值转化为正值，因为hash值有可能为负数，而&0x7FFFFFFF后，只有符号外改变，而后面的位都不变。

**7、内部实现使用的数组初始化和扩容方式不同**

      HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

      Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

      Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。

## **22.如何决定使用 HashMap 还是 TreeMap？**

TreeMap<K,V>的Key值是要求实现java.lang.Comparable，所以迭代的时候TreeMap默认是按照Key值升序排序的；TreeMap的实现是基于红黑树结构。适用于按自然顺序或自定义顺序遍历键（key）。

HashMap<K,V>的Key值实现散列hashCode()，分布是散列的、均匀的，不支持排序；数据结构主要是桶(数组)，链表或红黑树。适用于在Map中插入、删除和定位元素。

如果你需要得到一个有序的结果时就应该使用TreeMap（因为HashMap中元素的排列顺序是不固定的）。除此之外，由于HashMap有更好的性能，所以大多不需要排序的时候我们会使用HashMap。

**1、HashMap 和 TreeMap 的实现**

**HashMap：**基于哈希表实现。使用HashMap要求添加的键类明确定义了hashCode()和equals()[可以重写hashCode()和equals()]，为了优化HashMap空间的使用，您可以调优初始容量和负载因子。

1. HashMap(): 构建一个空的哈希映像
2. HashMap(Map m): 构建一个哈希映像，并且添加映像m的所有映射
3. HashMap(int initialCapacity): 构建一个拥有特定容量的空的哈希映像
4. HashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的哈希映像

**TreeMap：**基于红黑树实现。TreeMap没有调优选项，因为该树总处于平衡状态。

1. TreeMap()：构建一个空的映像树
2. TreeMap(Map m): 构建一个映像树，并且添加映像m中所有元素
3. TreeMap(Comparator c): 构建一个映像树，并且使用特定的比较器对关键字进行排序
4. TreeMap(SortedMap s): 构建一个映像树，添加映像树s中所有映射，并且使用与有序映像s相同的比较器排序

**2、HashMap 和 TreeMap 都是非线程安全**

HashMap继承AbstractMap抽象类，TreeMap继承自SortedMap接口。

**AbstractMap抽象类：**覆盖了equals()和hashCode()方法以确保两个相等映射返回相同的哈希码。如果两个映射大小相等、包含同样的键且每个键在这两个映射中对应的值都相同，则这两个映射相等。映射的哈希码是映射元素哈希码的总和，其中每个元素是Map.Entry接口的一个实现。因此，不论映射内部顺序如何，两个相等映射会报告相同的哈希码。

**SortedMap接口：**它用来保持键的有序顺序。SortedMap接口为映像的视图(子集)，包括两个端点提供了访问方法。除了排序是作用于映射的键以外，处理SortedMap和处理SortedSet一样。添加到SortedMap实现类的元素必须实现Comparable接口，否则您必须给它的构造函数提供一个Comparator接口的实现。TreeMap类是它的唯一一个实现。

**3、TreeMap中默认是按照升序进行排序的，如何让他降序**

通过自定义的比较器来实现

定义一个比较器类，实现Comparator接口，重写compare方法，有两个参数，这两个参数通过调用compareTo进行比较，而compareTo默认规则是：

1. 如果参数字符串等于此字符串，则返回 0 值；
2. 如果此字符串小于字符串参数，则返回一个小于 0 的值；
3. 如果此字符串大于字符串参数，则返回一个大于 0 的值。

## **23.说一下 HashMap 的实现原理？**

HashMap 的实现原理：

HashMap是基于Hash算法实现的，

我们通过put（key，value）存储数据，通过get（key）来获取数据

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 ，。

当计算出相同的Hash值时，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，使用链表存储，

否则使用红黑树。

## **24.说一下 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不允许有重复的值，并且元素是无序的

## **25.ArrayList 和 LinkedList 的区别是什么？**

ArrayList和LinkedList都实现了List接口，他们有以下的不同点：

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)。

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。

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。

也可以参考ArrayList vs. LinkedList。

1) 因为 Array 是基于索引 (index) 的数据结构，它使用索引在数组中搜索和读取数据是很快的。 Array 获取数据的时间复杂度是 O(1), 但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于 ArrayList ， LinkedList 插入是更快的。因为 LinkedList 不像 ArrayList 一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是 ArrayList 最坏的一种情况，时间复杂度是 O(n) ，而 LinkedList 中插入或删除的时间复杂度仅为 O(1) 。 ArrayList 在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时， LinkedList 也优于 ArrayList 。

4) LinkedList 需要更多的内存，因为 ArrayList 的每个索引的位置是实际的数据，而 LinkedList 中的每个节点中存储的是实际的数据和前后节点的位置 ( 一个 LinkedList 实例存储了两个值： Node<E> first 和 Node<E> last 分别表示链表的其实节点和尾节点，每个 Node 实例存储了三个值： E item,Node next,Node pre) 。

什么场景下更适宜使用 LinkedList，而不用ArrayList

1) 你的应用不会随机访问数据 。因为如果你需要LinkedList中的第n个元素的时候，你需要从第一个元素顺序数到第n个数据，然后读取数据。

2) 你的应用更多的插入和删除元素，更少的读取数据 。因为插入和删除元素不涉及重排数据，所以它要比ArrayList要快。

换句话说，ArrayList的实现用的是数组，LinkedList是基于链表，ArrayList适合查找，LinkedList适合增删

以上就是关于 ArrayList和LinkedList的差别。你需要一个不同步的基于索引的数据访问时，请尽量使用ArrayList。ArrayList很快，也很容易使用。但是要记得要给定一个合适的初始大小，尽可能的减少更改数组的大小

## **26.如何实现数组和 List 之间的转换？**

List转数组：toArray(arraylist.size()方法

数组转List：Arrays的asList(a)方法

**示例：**

List<String> arrayList = new ArrayList<String>();

arrayList.add("s");

arrayList.add("e");

arrayList.add("n");

/\*\*

 \* ArrayList转数组

 \*/

int size=arrayList.size();

String[] a = arrayList.toArray(new String[size]);

//输出第二个元素

System.out.println(a[1]);//结果：e

//输出整个数组

System.out.println(Arrays.toString(a));//结果：[s, e, n]

/\*\*

 \* 数组转list

 \*/

List<String> list=Arrays.asList(a);

/\*\*

\* list转Arraylist

\*/

List<String> arrayList2 = new ArrayList<String>();

arrayList2.addAll(list);

System.out.println(list)

## **27.ArrayList 和 Vector 的区别是什么？**

1. ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。
2. Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。
3. LinkedList是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了List接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。
4. vector是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而Arraylist是线程异步（ASynchronized）的，是不安全的。如果不考虑到线程的安全因素，一般用Arraylist效率比较高。
5. 如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度

的50%.如过在集合中使用数据量比较大的数据，用vector有一定的优势。

1. 如果查找一个指定位置的数据，vector和arraylist使用的时间是相同的，都是0(1),这个时候使用vector和arraylist都可以。而

如果移动一个指定位置的数据花费的时间为0(n-i)n为总长度，这个时候就应该考虑到使用Linkedlist,因为它移动一个指定位置的数据

所花费的时间为0(1),而查询一个指定位置的数据时花费的时间为0(i)。

ArrayList 和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，

都允许直接序号索引元素，但是插入数据要设计到数组元素移动 等内存操作，所以索引数据快插入数据慢，

Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差

，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快！

1. 1）  Vector的方法都是同步的(Synchronized),是线程安全的(thread-safe)，而ArrayList的方法不是，由于线程的同步必然要影响性能，因此,ArrayList的性能比Vector好。

2） 当Vector或ArrayList中的元素超过它的初始大小时,Vector会将它的容量翻倍,而ArrayList只增加50%的大小，这样,ArrayList就有利于节约内存空间。

## **28.Array 和 ArrayList 有何区别？**

Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。

Array大小是固定的，ArrayList的大小是动态变化的。

ArrayList提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

ArrayList可以算是Array的加强版，（对array有所取舍的加强）。

另附分类比较：

**存储内容比较：**

1. Array数组可以包含基本类型和对象类型，
2. ArrayList却只能包含对象类型。

但是需要注意的是：Array数组在存放的时候一定是同种类型的元素。ArrayList就不一定了，因为ArrayList可以存储Object。

**空间大小比较：**

1. 它的空间大小是固定的，空间不够时也不能再次申请，所以需要事前确定合适的空间大小。
2. ArrayList的空间是动态增长的，如果空间不够，它会创建一个空间比原空间大约0.5倍的新数组，然后将所有元素复制到新数组中，接着抛弃旧数组。而且，每次添加新的元素的时候都会检查内部数组的空间是否足够

## **29.在 Queue 中 poll()和 remove()有什么区别？**

1、offer()和add()的区别

add()和offer()都是向队列中添加一个元素。但是如果想在一个满的队列中加入一个新元素，调用 add() 方法就会抛出一个 unchecked 异常，而调用 offer() 方法会返回 false。可以据此在程序中进行有效的判断！

2、peek()和element()的区别

peek()和element()都将在不移除的情况下返回队头，但是peek()方法在队列为空时返回null，调用element()方法会抛出NoSuchElementException异常。

3、poll()和remove()的区别

poll()和remove()都将移除并且返回对头，但是在poll()在队列为空时返回null，而remove()会抛出NoSuchElementException异常

## 30.哪些集合类是线程安全的？

1. Vector 线程安全：
2. HashTable 线程安全：
3. StringBuffer 线程安全：

线程不安全的：

1. ArrayList ：
2. LinkedList：
3. HashMap：
4. HashSet：
5. TreeMap：
6. TreeSet：
7. StringBulider：

## **31.迭代器 Iterator 是什么？**

为了方便的处理集合中的元素,Java中出现了一个对象,该对象提供了一些方法专门处理集合中的元素.例如删除和获取集合中的元素.该对象就叫做迭代器(Iterator).

对 Collection 进行迭代的类，称其为迭代器。还是面向对象的思想，专业对象做专业的事情，迭代器就是专门取出集合元素的对象。但是该对象比较特殊，不能直接创建对象（通过new），该对象是以内部类的形式存在于每个集合类的内部。

如何获取迭代器？Collection接口中定义了获取集合类迭代器的方法（iterator（）），所以所有的Collection体系集合都可以获取自身的迭代器。

## **32.Iterator 怎么使用？有什么特点？**

public class TestIterator {

         static List<String> list = new ArrayList<String>();

         static {

                   list.add("111");

                   list.add("222");

                   list.add("333");

         }

         public static void main(String[] args) {

                   testIteratorNext();

                   System.out.println();

                   testForEachRemaining();

                   System.out.println();

                   testIteratorRemove();

         }

         //使用 hasNext 和 next遍历

         public static void testIteratorNext() {

                   Iterator<String> iterator = list.iterator();

                   while (iterator.hasNext()) {

                            String str = iterator.next();

                            System.out.println(str);

                   }

         }

         //使用 Iterator 删除元素

         public static void testIteratorRemove() {

                   Iterator<String> iterator = list.iterator();

                   while (iterator.hasNext()) {

                            String str = iterator.next();

                            if ("222".equals(str)) {

                                     iterator.remove();

                            }

                   }

                   System.out.println(list);

         }

         //使用 forEachRemaining 遍历

         public static void testForEachRemaining() {

                   final Iterator<String> iterator = list.iterator();

                   iterator.forEachRemaining(new Consumer<String>() {

                            public void accept(String t) {

                                     System.out.println(t);

                            }

                   });

         }

}

## **33.Iterator 和 ListIterator 有什么区别？**

我们需要知道的第一个则是：

（1）所属关系，ListIterator是一个Iterator的子类型。

（2）局限：只能应用于各种List类的访问。

（3）优势：Iterator只能向前移动，而ListIterator可以双向移动。

还可以产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引

nextIndex()、previousIndex()方法。

还可以通过set（）方法替换它访问过的最后一个元素。

还可以通过调用listIterator（）方法产生一个指向List开始处的ListIterator，当然也可以有参数，即指向索引为参数处的ListIterator。

（4）ListIterator 有 add() 方法，可以向 List 中添加对象，而 Iterator 不能**。**

## **34.怎么确保一个集合不能被修改？**

我们可以采用Collections包下的unmodifiableMap方法，通过这个方法返回的map,是不可以修改的。他会报 java.lang.UnsupportedOperationException错。

同理：Collections包也提供了对list和set集合的方法。

Collections.unmodifiableList(List)

Collections.unmodifiableSet(Set)

# 三、多线程

## synchronized在静态方法和普通方法的异同

所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。  
 而所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

## **35.并行和并发有什么区别？**

并发(concurrent)指的是多个程序可以同时运行的现象，更细化的是多进程可以同时运行或者多指令可以同时运行。但这不是重点，在描述并发的时候也不会去扣这种字眼是否精确，并发的重点在于它是一种现象。并发描述的是多进程同时运行的现象。但实际上，对于单核心CPU来说，同一时刻只能运行一个进程。所以，这里的"同时运行"表示的不是真的同一时刻有多个进程运行的现象，这是并行的概念，而是提供一种功能让用户看来多个程序同时运行起来了，但实际上这些程序中的进程不是一直霸占CPU的，而是执行一会停一会。

所以，并发和并行的区别就很明显了。**它们虽然都说是"多个进程同时运行"，但是它们的"同时"不是一个概念。并行的"同时"是同一时刻可以多个进程在运行(处于running)，并发的"同时"是经过上下文快速切换，使得看上去多个进程同时都在运行的现象，是一种OS欺骗用户的现象**。

实际上，**当程序中写下多进程或多线程代码时，这意味着的是并发而不是并行**。并发是因为多进程/多线程都是需要去完成的任务，不并行是因为并行与否由操作系统的调度器决定，可能会让多个进程/线程被调度到同一个CPU核心上。只不过调度算法会尽量让不同进程/线程使用不同的CPU核心，所以在实际使用中几乎总是会并行，但却不能以100%的角度去保证会并行。也就是说，**并行与否程序员无法控制，只能让操作系统决定**。

## **36.线程和进程的区别？**

进程和线程*就是这样的背景出来的，两个名词不过是对应的CPU时间段的描述，名词就是这样的功能。*

1. 进程就是包换上下文切换的程序执行时间总和 = CPU加载上下文+CPU执行+CPU保存上下文

*线程是什么呢？*

进程的颗粒度太大，每次都要有上下的调入，保存，调出。如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序A，实际分成 a，b，c等多个块组合而成。那么这里具体的执行就可能变成：

程序A得到CPU =》CPU加载上下文，开始执行程序A的a小段，然后执行A的b小段，然后再执行A的c小段，最后CPU保存A的上下文。

这里a，b，c的执行是共享了A的上下文，CPU在执行的时候没有进行上下文切换的。这里的a，b，c就是线程，也就是说线程是共享了进程的上下文环境，的更为细小的CPU时间段。

到此全文结束，再一个总结：

进程和线程都是一个时间段的描述，是CPU工作时间段的描述，不过是颗粒大小不同。

## **37.守护线程是什么？**

守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程，这是它的作用——而其他的线程只有一种，那就是用户线程。所以java里线程分2种，

1、守护线程，比如垃圾回收线程，就是最典型的守护线程。

2、用户线程，就是应用程序里的自定义线程

1、守护线程，专门用于服务其他的线程，如果其他的线程（即用户自定义线程）都执行完毕，连main线程也执行完毕，那么jvm就会退出（即停止运行）——此时，连jvm都停止运行了，守护线程当然也就停止执行了。

2、再换一种说法，如果有用户自定义线程存在的话，jvm就不会退出——此时，守护线程也不能退出，也就是它还要运行，干嘛呢，就是为了执行垃圾回收的任务啊。

3、守护线程又被称为“服务进程”“精灵线程”“后台线程”，是指在程序运行是在后台提供一种通用的线程，这种线程并不属于程序不可或缺的部分。 通俗点讲，任何一个守护线程都是整个JVM中所有非守护线程的“保姆”。

## **38.创建线程有哪几种方式？**

**一、继承Thread类创建线程类**

（1）定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。

（2）创建Thread子类的实例，即创建了线程对象。

（3）调用线程对象的start()方法来启动该线程

**二、通过Runnable接口创建线程类**

（1）定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

（3）调用线程对象的start()方法来启动该线程。

**三、通过Callable和Future创建线程**

（1）创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。

（2）创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。

（3）使用FutureTask对象作为Thread对象的target创建并启动新线程。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

**二、创建线程的三种方式的对比**

采用实现Runnable、Callable接口的方式创见多线程时，优势是：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承Thread类的方式创建多线程时优势是：

编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

 劣势是：

线程类已经继承了Thread类，所以不能再继承其他父类。

## **39.说一下 runnable 和 callable 有什么区别？**

1）Runnable提供run方法，无法通过throws抛出异常，所有CheckedException必须在run方法内部处理。Callable提供call方法，直接抛出Exception异常。

2）Runnable的run方法无返回值，Callable的call方法提供返回值用来表示任务运行的结果

3）Runnable可以作为Thread构造器的参数，通过开启新的线程来执行，也可以通过线程池来执行。而Callable只能通过线程池执行。

## **40.线程有哪些状态？**

线程状态有 5 种，新建，就绪，运行，阻塞，死亡。

1. 线程 start 方法执行后，并不表示该线程运行了，而是进入就绪状态，意思是随时准备运行，但是真正何时运行，是由操作系统决定的，代码并不能控制，

2. 同样的，从运行状态的线程，也可能由于失去了 CPU 资源，回到就绪状态，也是由操作系统决定的。这一步中，也可以由程序主动失去 CPU 资源，只需调用 yield 方法。

3. 线程运行完毕，或者运行了一半异常了，或者主动调用线程的 stop 方法，那么就进入死亡。死亡的线程不可逆转。

4. 下面几个行为，会引起线程阻塞。

主动调用 sleep 方法。时间到了会进入就绪状态

主动调用 suspend 方法。主动调用 resume 方法，会进入就绪状态

调用了阻塞式 IO 方法。调用完成后，会进入就绪状态。

试图获取锁。成功的获取锁之后，会进入就绪状态。

线程在等待某个通知。其它线程发出通知后，会进入就绪状态

## **41.sleep() 和 wait() 有什么区别？**

1、这两个方法来自不同的类分别是Thread和Object，sleep方法属于Thread类中的静态方法，wait属于Object的成员方法。

2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用（使用范围）。

sleep方法属于Thread类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了sleep方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。但在sleep的过程中有可能被其他对象调用它的interrupt(),产生InterruptedException异常，如果你的程序不捕获这个异常，线程就会异常终止，进入TERMINATED结束状态，如果你的程序捕获了这个异常，那么程序就会继续执行catch语句块（可能还有finally语句块）以及以后的代码。

wait属于Object的成员方法，一旦一个对象调用了wait方法，必须要采用notify()和notifyAll()方法唤醒该线程;如果线程拥有某个或某些对象的同步锁，那么在调用了wait()后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了wait()方法的对象。wait()方法也同样会在wait的过程中有可能被其他对象调用interrupt()方法而产生InterruptedException，效果以及处理方式同sleep()方法.

## **42.notify()和 notifyAll()有什么区别？**

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

## **43.线程的 run()和 start()有什么区别？**

调用 start() 方法是用来启动线程的，轮到该线程执行时，会自动调用 run()；直接调用 run() 方法，无法达到启动多线程的目的，相当于主线程线性执行 Thread 对象的 run() 方法。

一个线程对线的 start() 方法只能调用一次，多次调用会抛出 java.lang.IllegalThreadStateException 异常；run() 方法没有限制。

run()和start()的区别可以用一句话概括：**单独调用run()方法，是同步执行；通过start()调用run()，是异步执行。**

## **44.创建线程池有哪几种方式？**

newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

newSingleThreadScheduledExecutor

单线程可执行周期性任务的线程池

newWorkStealingPool

任务窃取线程池，不保证执行顺序，适合任务耗时差异较大。

线程池中有多个线程队列，有的线程队列中有大量的比较耗时的任务堆积，而有的线程队列却是空的，就存在有的线程处于饥饿状态，当一个线程处于饥饿状态时，它就会去其它的线程队列中窃取任务。解决饥饿导致的效率问题。

默认创建的并行 level 是 CPU 的核数。主线程结束，即使线程池有任务也会立即停止。

## **45.线程池都有哪些状态？**

线程池的5种状态：RUNNING、SHUTDOWN、STOP、TIDYING、TERMINATED。

## **46.线程池中 submit()和 execute()方法有什么区别？**

线程池中的execute方法大家都不陌生，即开启线程执行池中的任务。还有一个方法submit也可以做到，它的功能是提交指定的任务去执行并且返回Future对象，即执行的结果。下面简要介绍一下两者的三个区别：

1、接收的参数不一样

2、submit有返回值，而execute没有

用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。

然后我就可以把所有失败的原因综合起来发给调用者。

个人觉得cancel execution这个用处不大，很少有需要去取消执行的。

而最大的用处应该是第二点。

3、submit方便Exception处理

意思就是如果你在你的task里会抛出checked或者unchecked exception，

而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

## **47.在 java 程序中怎么保证多线程的运行安全？**

线程的安全性问题体现在：

原子性：一个或者多个操作在 CPU 执行的过程中不被中断的特性

可见性：一个线程对共享变量的修改，另外一个线程能够立刻看到

有序性：程序执行的顺序按照代码的先后顺序执行

导致原因：

缓存导致的可见性问题

线程切换带来的原子性问题

编译优化带来的有序性问题

解决办法：

JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题

synchronized、volatile、LOCK，可以解决可见性问题

Happens-Before 规则可以解决有序性问题

Happens-Before 规则如下：

程序次序规则：在一个线程内，按照程序控制流顺序，书写在前面的操作先行发生于书写在后面的操作

管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作

volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作

线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始

## **48.多线程锁的升级原理是什么？**

锁的级别从低到高：

无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁

锁分级别原因：

没有优化以前，sychronized是重量级锁（悲观锁），使用 wait 和 notify、notifyAll 来切换线程状态非常消耗系统资源；线程的挂起和唤醒间隔很短暂，这样很浪费资源，影响性能。所以 JVM 对 sychronized 关键字进行了优化，把锁分为 无锁、偏向锁、轻量级锁、重量级锁 状态。

无锁：没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功，其他修改失败的线程会不断重试直到修改成功。

偏向锁：对象的代码一直被同一线程执行，不存在多个线程竞争，该线程在后续的执行中自动获取锁，降低获取锁带来的性能开销。偏向锁，指的就是偏向第一个加锁线程，该线程是不会主动释放偏向锁的，只有当其他线程尝试竞争偏向锁才会被释放。

偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；

如果线程处于活动状态，升级为轻量级锁的状态。

轻量级锁：轻量级锁是指当锁是偏向锁的时候，被第二个线程 B 所访问，此时偏向锁就会升级为轻量级锁，线程 B 会通过自旋的形式尝试获取锁，线程不会阻塞，从而提高性能。

当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。

重量级锁：指当有一个线程获取锁之后，其余所有等待获取该锁的线程都会处于阻塞状态。

重量级锁通过对象内部的监视器（monitor）实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。



## **49.什么是死锁？**

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

　　例如，在某个计算机系统中只有一台打印机和一台输入 设备，进程P1正占用输入设备，同时又提出使用打印机的请求，但此时打印机正被进程P2 所占用，而P2在未释放打印机之前，又提出请求使用正被P1占用着的输入设备。这样两个进程相互无休止地等待下去，均无法继续执行，此时两个进程陷入死锁状态。

**1.系统资源的竞争**

　　当系统中供多个进程共享的资源如打印机、公用队列的等，其数目不足以满足诸进程的需要时，会引起诸进程对资源的竞争而产生死锁。

**2.进程运行推进顺序不当引起死锁**

　　**● 进程推进顺序合法**

　　当进程P1和P2并发执行时，如果按照下述顺序推进：P1：Request（R1）； P1：Request（R2）； P1: Relese（R1）；P1: Relese（R2）； P2：Request（R2）； P2：Request（R1）； P2: Relese（R2）；P2: Relese（R1）；这两个进程便可顺利完成，这种不会引起进程死锁的推进顺序是合法的。

　　**● 进程推进顺序非法**

　　若P1保持了资源R1,P2保持了资源R2，系统处于不安全状态，因为这两个进程再向前推进，便可能发生死锁。例如，当P1运行到P1：Request（R2）时，将因R2已被P2占用而阻塞；当P2运行到P2：Request（R1）时，也将因R1已被P1占用而阻塞，于是发生进程死锁。

**产生死锁的四个必要条件：**

　　**● 互斥条件**：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

　　**● 请求与保持条件**：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

　　**● 不可剥夺条件**：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。

　　**● 循环等待条件**：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

　　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。

## **50.怎么防止死锁？**

**1.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。**

**2.尽量使用 Java. util. concurrent 并发类代替自己手写锁。**

**3.尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。**

**4.尽量减少同步的代码块.**

**死锁避免的基本思想：**系统对进程发出每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，如果分配后系统可能发生死锁,则不予分配，否则予以分配。这是一种保证系统不进入死锁状态的动态策略。

　　理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。只要打破四个必要条件之一就能有效预防死锁的发生：

　　**● 打破互斥条件**：改造独占性资源为虚拟资源，大部分资源已无法改造。

　　**● 打破不可抢占条件**：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。

　　**● 打破占有且申请条件**：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。

　　**● 打破循环等待条件**：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

**死锁避免和死锁预防的区别：**

　　死锁预防是设法至少破坏产生死锁的四个必要条件之一，严格的防止死锁的出现；而死锁避免则不那么严格的限制产生死锁的必要条件的存在，因为即使死锁的必要条件存在，也不一定发生死锁。死锁避免是在系统运行过程中注意避免死锁的最终发生

## **51.ThreadLocal 是什么？有哪些使用场景？**

ThreadLocal 是线程本地存储，在每个线程中都创建了一个 ThreadLocalMap 对象，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。

为了说明ThreadLocal的应用场景。我们来看一个框架的样例。Spring的事务管理器通过AOP切入业务代码，在进入业务代码前，会依据相应的事务管理器提取出相应的事务对象，假如事务管理器是DataSourceTransactionManager，就会从DataSource中获取一个连接对象，通过一定的包装后将其保存在ThreadLocal中。而且Spring也将DataSource进行了包装，重写了当中的getConnection()方法，或者说该方法的返回将由Spring来控制，这样Spring就能让线程内多次获取到的Connection对象是同一个。

为什么要放在ThreadLocal里面呢？由于Spring在AOP后并不能向应用程序传递參数。应用程序的每一个业务代码是事先定义好的，Spring并不会要求在业务代码的入口參数中必须编写Connection的入口參数。此时Spring选择了ThreadLocal，通过它保证连接对象始终在线程内部，不论什么时候都能拿到，此时Spring很清楚什么时候回收这个连接，也就是很清楚什么时候从ThreadLocal中删除这个元素

1.Hiberante的Session 工具类HibernateUtil

2.通过不同的线程对象设置Bean属性，保证各个线程Bean对象的独立性。

ThreadLocal使用的一般步骤:

1、在多线程的类（如ThreadDemo类）中。创建一个ThreadLocal对象threadXxx，用来保存线程间须要隔离处理的对象xxx。
2、在ThreadDemo类中。创建一个获取要隔离訪问的数据的方法getXxx()，在方法中推断，若ThreadLocal对象为null时候，应该new()一个隔离訪问类型的对象，并强制转换为要应用的类型。
3、在ThreadDemo类的run()方法中。通过getXxx()方法获取要操作的数据。这样能够保证每一个线程相应一个数据对象，在不论什么时刻都操作的是这个对象。

与Synchonized的对照:

ThreadLocal和Synchonized都用于解决多线程并发訪问。可是ThreadLocal与synchronized有本质的差别。synchronized是利用锁的机制，使变量或代码块在某一时该仅仅能被一个线程訪问。而ThreadLocal为每个线程都提供了变量的副本，使得每个线程在某一时间訪问到的并非同一个对象，这样就隔离了多个线程对数据的数据共享。而Synchronized却正好相反，它用于在多个线程间通信时可以获得数据共享。
Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。

一句话理解ThreadLocal:向ThreadLocal里面存东西就是向它里面的Map存东西的，然后ThreadLocal把这个Map挂到当前的线程底下，这样Map就仅仅属于这个线程了。

## **52.说一下 synchronized 底层实现原理？**

1、jvm基于进入和退出Monitor对象来实现方法同步和代码块同步。

方法级的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有monitor（虚拟机规范中用的是管程一词）， 然后再执行方法，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor。

代码块的同步是利用monitorenter和monitorexit这两个字节码指令。它们分别位于同步代码块的开始和结束位置。当jvm执行到monitorenter指令时，当前线程试图获取monitor对象的所有权，如果未加锁或者已经被当前线程所持有，就把锁的计数器+1；当执行monitorexit指令时，锁计数器-1；当锁计数器为0时，该锁就被释放了。如果获取monitor对象失败，该线程则会进入阻塞状态，直到其他线程释放锁。

## **53.synchronized 和 volatile 的区别是什么？**

1. volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的
3. volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性
4. volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
5. volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

**volatile和synchronized特点**

首先需要理解线程安全的两个方面：**执行控制**和**内存可见**。

**执行控制**的目的是控制代码执行（顺序）及是否可以并发执行。

**内存可见**控制的是线程执行结果在内存中对其它线程的可见性。根据[Java内存模型](https://www.cnblogs.com/kaleidoscope/p/9505829.html)的实现，线程在具体执行时，会先拷贝主存数据到线程本地（CPU缓存），操作完成后再把结果从线程本地刷到主存。

synchronized关键字解决的是执行控制的问题，它会阻止其它线程获取当前对象的监控锁，这样就使得当前对象中被synchronized关键字保护的代码块无法被其它线程访问，也就无法并发执行。更重要的是，synchronized还会创建一个**内存屏障**，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中，从而保证了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作，都**happens-before**于随后获得这个锁的线程的操作。

volatile关键字解决的是内存可见性的问题，会使得所有对volatile变量的读写都会直接刷到主存，即保证了变量的可见性。这样就能满足一些对变量可见性有要求而对读取顺序没有要求的需求。

使用volatile关键字仅能实现对原始变量(如boolen、 short 、int 、long等)操作的原子性，但需要特别注意， volatile不能保证复合操作的原子性，即使只是i++，实际上也是由多个原子操作组成：read i; inc; write i，假如多个线程同时执行i++，volatile只能保证他们操作的i是同一块内存，但依然可能出现写入脏数据的情况。

在Java 5提供了原子数据类型atomic wrapper classes，对它们的increase之类的操作都是原子操作，不需要使用sychronized关键字。

对于volatile关键字，当且仅当满足以下所有条件时可使用：

1. 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。

2. 该变量没有包含在具有其他变量的不变式中。

## **54.synchronized 和 Lock 有什么区别？**

总结来说，Lock和synchronized有以下几点不同：

　　1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　5）Lock可以提高多个线程进行读操作的效率。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

　　而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁

## **55.synchronized 和 ReentrantLock 区别是什么？**

synchronized 竞争锁时会一直等待；ReentrantLock 可以尝试获取锁，并得到获取结果

synchronized 获取锁无法设置超时；ReentrantLock 可以设置获取锁的超时时间

synchronized 无法实现公平锁；ReentrantLock 可以满足公平锁，即先等待先获取到锁

synchronized 控制等待和唤醒需要结合加锁对象的 wait() 和 notify()、notifyAll()；ReentrantLock 控制等待和唤醒需要结合 Condition 的 await() 和 signal()、signalAll() 方法

synchronized 是 JVM 层面实现的；ReentrantLock 是 JDK 代码层面实现

synchronized 在加锁代码块执行完或者出现异常，自动释放锁；ReentrantLock 不会自动释放锁，需要在 finally{} 代码块显示释放

　 1、synchronized是关键字，ReentrantLock是类；

　　2、ReentrantLock可以获取锁的时间进行设置，避免死锁；

　　3、ReentrantLock可以获取各种锁的信息；

　　4、ReentrantLock可以灵活实现多路通知；

　　5、机制：sync操作Mark Word，lock调用Unsafe类的park（）方法。

## **56.说一下 atomic 的原理？**

无锁即无障碍的运行, 所有线程都可以到达临界区, 接近于无等待.

无锁采用CAS(compare and swap)算法来处理线程冲突, 其原理如下

CAS原理

CAS包含3个参数CAS(V,E,N).V表示要更新的变量, E表示预期值, N表示新值.

仅当V值等于E值时, 才会将V的值设为N, 如果V值和E值不同, 则说明已经有其他线程做了更新, 则当前线程什么

都不做. 最后, CAS返回当前V的真实值. CAS操作是抱着乐观的态度进行的, 它总是认为自己可以成功完成操作.

当多个线程同时使用CAS操作一个变量时, 只有一个会胜出, 并成功更新, 其余均会失败.失败的线程不会被挂起,

仅被告知失败, 并且允许再次尝试, 当然也允许失败的线程放弃操作.基于这样的原理, CAS操作即时没有锁,

也可以发现其他线程对当前线程的干扰, 并进行恰当的处理.

CPU指令

另外, 虽然上述步骤繁多, 实际上CAS整一个操作过程是一个原子操作, 它是由一条CPU指令完成的,

从指令层保证操作可靠, 不会被多线程干扰.

无锁与volatile

无锁可以通过cas来保证原子性与线程安全, 他与volatile什么区别呢?

当给变量加了volatile关键字, 表示该变量对所有线程可见, 但不保证原子性.

以volatile i, i++为例, 分为以下四步

加载i

对i进行+1

回写i的值

用内存屏障通知其他线程i的值

其中前三步是线程不安全的, 可能其他线程会对i进行读写.

这样怎么实现线程安全呢？请大家自己先考虑一下这个问题，其实我们在语言层面是没有做任何同步的操作的，大家也可以看到源码没有任何锁加在上面，可它为什么是线程安全的呢？这就是Atomic包下这些类的奥秘：语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。所以说，CAS并不是无阻塞，只是阻塞并非在语言、线程方面，而是在硬件层面，所以无疑这样的操作会更快更高效！

虽然基于CAS的线程安全机制很好很高效，但要说的是，并非所有线程安全都可以用这样的方法来实现，这只适合一些粒度比较小，型如计数器这样的需求用起来才有效，否则也不会有锁的存在了。

# 四、反射

## **57.什么是反射？**

JAVA反射机制是在运行状态中，对于任意一个实体类，都能够知道这个类的所有属性和方法；

对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

JAVA反射（放射）机制：“程序运行时，允许改变程序结构或变量类型，这种语言称为[动态语言](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E8%AF%AD%E8%A8%80/797407)”。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。但是JAVA有着一个非常突出的动态相关机制：Reflection，用在Java身上指的是我们可以于运行时加载、探知、使用编译期间完全未知的classes。换句话说，Java程序可以加载一个运行时才得知名称的class，获悉其完整构造（但不包括methods定义），并生成其对象实体、或对其fields设值、或唤起其methods。

## **58.什么是 java 序列化？什么情况下需要序列化？**

序列化：将 Java 对象转换成字节流的过程。

反序列化：将字节流转换成 Java 对象的过程。

当 Java 对象需要在网络上传输 或者 持久化存储到文件中时，就需要对 Java 对象进行序列化处理。

序列化的实现：类实现 Serializable 接口，这个接口没有需要实现的方法。实现 Serializable 接口是为了告诉 jvm 这个类的对象可以被序列化。

注意事项：

某个类可以被序列化，则其子类也可以被序列化

声明为 static 和 transient 的成员变量，不能被序列化。static 成员变量是描述类级别的属性，transient 表示临时数据

反序列化读取序列化对象的顺序要保持一致

## **59.动态代理是什么？有哪些应用？**

动态代理：当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

动态代理的应用：Spring的AOP，加事务，加权限，加日志。

## **60.怎么实现动态代理？**

动态代理实现：首先必须定义一个接口，还要有一个InvocationHandler(将实现接口的类的对象传递给它)处理类。再有一个工具类Proxy(习惯性将其称为代理类，因为调用他的newInstance()可以产生代理对象,其实他只是一个产生代理对象的工具类）。利用到InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回

# 五、对象拷贝

## **61.为什么要使用克隆？**

想对一个对象进行处理，又想保留原有的数据进行接下来的操作

## **62.如何实现对象克隆？**

**1). 实现Cloneable接口并重写Object类中的clone()方法；**

**2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆。**

## **63.深拷贝和浅拷贝区别是什么？**

浅拷贝：复制基本类型的属性；引用类型的属性复制，复制栈中的变量 和 变量指向堆内存中的对象的指针，不复制堆内存中的对象。

深拷贝：复制基本类型的属性；引用类型的属性复制，复制栈中的变量 和 变量指向堆内存中的对象的指针和堆内存中的对象。

# **六、Java Web**

## **64.jsp 和 servlet 有什么区别？**

Servlet

一种服务器端的Java应用程序

由 Web 容器加载和管理

用于生成动态 Web 内容

负责处理客户端请求

Jsp

是 Servlet 的扩展，本质上还是 Servlet

每个 Jsp 页面就是一个 Servlet 实例

Jsp 页面会被 Web 容器编译成 Servlet，Servlet 再负责响应用户请求

区别

Servlet 适合动态输出 Web 数据和业务逻辑处理，对于 html 页面内容的修改非常不方便；Jsp 是在 Html 代码中嵌入 Java 代码，适合页面的显示

内置对象不同，获取内置对象的方式不同

## **65.jsp 有哪些内置对象？作用分别是什么？**

1.HttpServletRequet类的Request对象：代表请求对象，主要用于接受客户端通过HTTP协议连接传输服务器端的数据。

2.HttpSevletResponse类的Response对象：代表响应对象，主要用于向客户端发送数据。

3.JspWriter类的out对象：主要用于向客户端输出数据，out的基类是jspWriter

4.HttpSession类的session对象：主要用来分别保存每个月的信息与请求关联的会话；会话状态的维持是web应用开发者必须面对的问题。

5.ServletContext类的application对象：主要用于保存用户信息，代码片段的运行环境；它是一个共享的内置对象，即一个容器中的多个用户共享一个application

，故其保存的信息被所有用户所共享。

6.PageContext类的PageContext对象：管理网页属性，为jsp页面包装页面的上下文，管理对属于jsp的特殊可见部分中已经命名对象的访问，它的创建和初始化都是由容器来完成的。

7.ServletConfig类的Config对象：代码片段配置对象，标识Servlet的配置。

8.Object类的Page对象，处理jsp页面，是object类的一个实例，指的是jsp实现类的实例

9.Exception对象：处理jsp文件执行时发生的错误和异常，只有在错误页面里才使用，前提是在页面指令里要有isErrorPage=true。

## **66.说一下 jsp 的 4 种作用域？**

**application 作用域**

     如果把变量放到application里，就说明它的作用域是application，它的有效范围是整个应用。 整个应用是指从应用启动，到应用结束。我们没有说“从服务器启动，到服务器关闭”，是因为一个服务器可能部署多个应用，当然你关闭了服务器，就会把上面所有的应用都关闭了。 application作用域里的变量，它们的存活时间是最长的，如果不进行手工删除，它们就一直可以使用。

application作用域上的信息传递是通过ServletContext实现的，它提供的主要方法如下所示：

Object getAttribute（String name）  //从application中获取信息；

void setAttribute（String name, Object value）  //向application作用域中设置信息。

**session作用域**

       session作用域比较容易理解，同一浏览器对服务器进行多次访问，在这多次访问之间传递信息，就是session作用域的体现。如果把变量放到session里，就说明它的作用域是session，它的有效范围是当前会话。所谓当前会话，就是指从用户打开浏览器开始，到用户关闭浏览器这中间的过程。这个过程可能包含多个请求响应。也就是说，只要用户不关浏览器，服务器就有办法知道这些请求是一个人发起的，整个过程被称为一个会话（session），而放到会话中的变量，就可以在当前会话的所有请求里使用。

 session是通过HttpSession接口实现的，它提供的主要方法如下所示：

Object HttpSession.getAttribute（String name） //从session中获取信息。

void HttpSession.setAttribute（String name, Object value）//向session中保存信息。

HttpSession HttpServletRequest.getSessio() //获取当前请求所在的session的对象。

 session的开始时刻比较容易判断，它从浏览器发出第一个HTTP请求即可认为会话开始。但结束时刻就不好判断了，因为浏览器关闭时并不会通知服务器，所以只能通过如下这种方法判断：如果一定的时间内客户端没有反应，则认为会话结束。Tomcat的默认值为120分钟，但这个值也可以通过HttpSession的setMaxInactiveInterval()方法来设置：

void setMaxInactiveInterval(int interval)

如果想主动让会话结束，例如用户单击“注销”按钮的时候，可以使用 HttpSession 的 invalidate()方法，用于强制结束当前session：void invalidate()

**request作用域**

     一个HTTP请求的处理可能需要多个Servlet合作，而这几个Servlet之间可以通过某种方式传递信息，但这个信息在请求结束后就无效了。request里的变量可以跨越forward前后的两页。但是只要刷新页面，它们就重新计算了。如果把变量放到request里，就说明它的作用域是request，它的有效范围是当前请求周期。 所谓请求周期，就是指从http请求发起，到服务器处理结束，返回响应的整个过程。在这个过程中可能使用forward的方式跳转了多个jsp页面，在这些页面里你都可以使用这个变量。

Servlet之间的信息共享是通过HttpServletRequest接口的两个方法来实现的：

void setAttribute（String name, Object value） //将对象value以name为名称保存到request作用域中。

Object getAttribute（String name） //从request作用域中取得指定名字的信息。

     JSP中的doGet()、doPost()方法的第一个参数就是HttpServletRequest对象，使用这个对象的 setAttribute()方法即可传递信息。那么在设置好信息之后，要通过何种方式将信息传给其他的Servlet呢？这就要用到RequestDispatcher接口的forward()方法，通过它将请求转发给其他Servlet。

 RequestDispatcher ServletContext.getRequestDispatcher(String path) //取得Dispatcher以便转发，path为转发的目的Servlet。

 void RequestDispatcher.forward(ServletRequest request, ServletResponse response)//将request和response转发

     因此，只需要在当前Servlet中先通过setAttribute()方法设置相应的属性，然后使用forward()方法进行跳转，最后在跳转到的Servlet中通过使用getAttribute()方法即可实现信息传递。

  **需要注意两点：**

1、转发不是重定向，转发是在Web应用内部进行的。

2、转发对浏览器是透明的，也就是说，无论在服务器上如何转发，浏览器地址栏中显示的仍然是最初那个Servlet的地址。

**page作用域**

page对象的作用范围仅限于用户请求的当前页面，对于page对象的引用将在响应返回给客户端之后被释放，或者在请求被转发到其他地方后被释放。page里的变量只要页面跳转了，它们就不见了。如果把变量放到pageContext里，就说明它的作用域是page，它的有效范围只在当前jsp页面里。从把变量放到pageContext开始，到jsp页面结束，你都可以使用这个变量。

以上介绍的作用范围越来越小，request和page的生命周期都是短暂的，它们之间的区别：一个request可以包含多个page页（include，forward及filter）。

**67.session 和 cookie 有什么区别？**

1）Cookie以文本文件格式存储在浏览器中，而session存储在服务端它存储了限制数据量。它只允许4kb它没有在cookie中保存多个变量。

（2）cookie的存储限制了数据量，只允许4KB，而session是无限量的

（3）我们可以轻松访问cookie值但是我们无法轻松访问会话值，因此它更安全

（4）设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。

## **68.说一下 session 的工作原理？**

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

**可以。**

**通过url重写，把 sessionid 作为参数追加的原 url 中，后续的浏览器与服务器交互中携带 sessionid 参数。**

**服务器的返回数据中包含 sessionid，浏览器发送请求时，携带 sessionid 参数。**

**通过 Http 协议其他 header 字段，服务器每次返回时设置该 header 字段信息，浏览器中 js 读取该 header 字段，请求服务器时，js设置携带该 header 字段**

## **70.spring mvc 和 struts 的区别是什么？**

springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过虑器。

springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

Struts采用值栈存储请求和响应的数据，通过OGNL存取数据， springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

springmvc与struts2有什么区别?

1. 底层实现机制

struts2:filter

springmvc:servlet

2. 运行效率

struts2:底层是Servlet,参数基于属性封装,如果配置单例,会出现线程安全问题,所以配置多例

springmvc:底层是Servlet,单例

3. 参数封装

struts2:基于属性封装

springmvc:基于方法进行封装

## **71.如何避免 sql 注入？**

**不要使用动态SQL**

避免将用户提供的输入直接放入SQL语句中；最好使用准备好的语句和参数化查询，这样更安全。

**不要将敏感数据保留在纯文本中**

加密存储在数据库中的私有/机密数据；这样可以提供了另一级保护，以防攻击者成功地排出敏感数据。

**限制数据库权限和特权**

将数据库用户的功能设置为最低要求；这将限制攻击者在设法获取访问权限时可以执行的操作。

**避免直接向用户显示数据库错误**

攻击者可以使用这些错误消息来获取有关数据库的信息。

**对访问数据库的Web应用程序使用Web应用程序防火墙（WAF）**

这为面向Web的应用程序提供了保护，它可以帮助识别SQL注入尝试；根据设置，它还可以帮助防止SQL注入尝试到达应用程序（以及数据库）。

**定期测试与数据库交互的Web应用程序**

这样做可以帮助捕获可能允许SQL注入的新错误或回归。

**将数据库更新为最新的可用修补程序**

这可以防止攻击者利用旧版本中存在的已知弱点/错误。

**总结：**SQL注入是一种流行的攻击攻击方法，但是通过采取适当的预防措施，例如确保数据加密，保护和测试Web应用程序，以及您是最新的补丁程序，您可以采取有意义的步骤来保持您的数据安全。

## **72.什么是 XSS 攻击，如何避免？**

XSS 攻击，即跨站脚本攻击（Cross Site Scripting），它是 web 程序中常见的漏洞。

攻击者往 web 页面里插入恶意的 HTML 代码（Javascript、css、html 标签等），当用户浏览该页面时，嵌入其中的 HTML 代码会被执行，从而达到恶意攻击用户的目的。如盗取用户 cookie 执行一系列操作，破坏页面结构、重定向到其他网站等。

**预防思路**

1. web 页面中可由用户输入的地方，如果对输入的数据转义、过滤处理
2. 后台输出页面的时候，也需要对输出内容进行转义、过滤处理（因为攻击者可能通过其他方式把恶意脚本写入数据库）
3. 前端对 html 标签属性、css 属性赋值的地方进行校验

**73.什么是 CSRF 攻击，如何避免？**

CSRF：Cross Site Request Forgery（跨站点请求伪造）。

CSRF 攻击者在用户已经登录目标网站之后，诱使用户访问一个攻击页面，利用目标网站对用户的信任，以用户身份在攻击页面对目标网站发起伪造用户操作的请求，达到攻击目的。

避免方法：

1. CSRF 漏洞进行检测的工具，如 CSRFTester、CSRF Request Builder...
2. 验证 HTTP Referer 字段
3. 添加并验证 token
4. 添加自定义 http 请求头
5. 敏感操作添加验证码
6. 使用 post 请求

# 七、异常

## **74.throw 和 throws 的区别？**

**throw：**

表示方法内抛出某种异常对象

如果异常对象是非 RuntimeException 则需要在方法申明时加上该异常的抛出 即需要加上 throws 语句 或者 在方法体内 try catch 处理该异常，否则编译报错

执行到 throw 语句则后面的语句块不再执行

**throws：**

方法的定义上使用 throws 表示这个方法可能抛出某种异常

需要由方法的调用者进行异常处理

## **75.final、finally、finalize 有什么区别？**

**fina**l

final用于声明属性、方法和类，分别表示属性不变，方法不可覆盖和类不可以被继承。

final属性：被final修饰的变量不可以变。一个是引用不可变，第二个是对象不可变。

被final修饰的引用地址不能被改变，但是可以对这个引用的内容进行更改，可以appen一些内容进去。所以被final修饰的必须初始化。

final的方法：方法不能当方法声明为final时，该方法不能被任何子类重写这个方法，但子类依然可以使用这个方法。

final参数：表示这个参数在这个方法的内部不允许被修改。

final类：当一个类声明为final时，此类不能被继承，所以的方法不能被重写。一个类不能既是abstract又是final。

**finally**

作为异常处理的一部分，他只能用于try/catch语句中，并且附带一个语句块，表示这个语句一定要被执行。

**finalize**

finalize是Object类的一个方法，在垃圾回收器执行时会调用被回收的对象的finalize（）方法，可以覆盖此方法来实现对其他资源回收。

## **76.try-catch-finally 中哪个部分可以省略？**

catch 和 finally 语句块可以省略其中一个。

## **77.try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？**

会执行，finally会在return之前执行；

## **78.常见的异常类有哪些？**

ArithmeticException——由于除数为0引起的异常；

ArrayStoreException——由于数组存储空间不够引起的异常；

ClassCastException—一当把一个对象归为某个类，但实际上此对象并不是由这个类 创建的，也不是其子类创建的，则会引起异常；

IllegalMonitorStateException——监控器状态出错引起的异常；

NegativeArraySizeException—一数组长度是负数，则产生异常；

NullPointerException—一程序试图访问一个空的数组中的元素或访问空的对象中的 方法或变量时产生异常；

SecurityException——由于访问了不应访问的指针，使安全性出问题而引起异常；

IndexOutOfBoundsExcention——由于数组下标越界或字符串访问越界引起异常；

IOException——由于文件未找到、未打开或者I/O操作不能进行而引起异常；

ClassNotFoundException——未找到指定名字的类或接口引起异常；

CloneNotSupportedException——一程序中的一个对象引用Object类的clone方法，但 此对象并没有连接Cloneable接口，从而引起异常；

InterruptedException—一当一个线程处于等待状态时，另一个线程中断此线程，从 而引起异常，有关线程的内容，将在下一章讲述；

NoSuchMethodException一所调用的方法未找到，引起异常；

IllegalAccessExcePtion—一试图访问一个非public方法；

StringIndexOutOfBoundsException——访问字符串序号越界，引起异常；

ArrayIdexOutOfBoundsException—一访问数组元素下标越界，引起异常；

NumberFormatException——字符的UTF代码数据格式有错引起异常；

IllegalThreadException—一线程调用某个方法而所处状态不适当，引起异常；

FileNotFoundException——未找到指定文件引起异常；

EOFException——未完成输入操作即遇文件结束引起异常。

# 八、网络

## **79.http 响应码 301 和 302 代表的是什么？有什么区别？**

**301 Moved Permanently**

被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则这个响应也是可缓存的。

**302 Found**

请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。

当网站迁移或url地址进行调整时，服务端需要重定向返回，保证原请求自动跳转新的地址。

http 协议的 301 和 302 状态码都代表重定向。浏览器请求某url收到这两个状态码时，都会显示和跳转到 Response Headers 中的Location。即在浏览器地址输入 url A，却自动跳转到url B。

**区别：**

301 表示被请求 url 永久转移到新的 url；302 表示被请求 url 临时转移到新的 url。

301 搜索引擎会索引新 url 和新 url 页面的内容；302 搜索引擎可能会索引旧 url 和 新 url 的页面内容。

302 的返回码可能被别人利用，劫持你的网址。因为搜索引擎索引他的网址，他返回 302 跳转到你的页面。

## **80.forward 和 redirect 的区别？**

注意：转发与重定向的区别？

1、重定向两次请求，转发一次请求

2、重定向地址栏的地址变化，转发地址不变

3、重新定向可以访问外部网站 转发只能访问内部资源

5、转发的性能要优于重定向

注意：客户端地址与服务器端地址的写法？

客户端地址：

是客户端去访问服务器的地址，服务器外部的地址，特点：写上web应用名称

服务器端地址：

服务器内部资源的跳转的地址，特点：不需要写web应用的名称

## **81.简述 tcp 和 udp的区别？**

**（1）TCP面向连接；UDP面向无连接**

**（2）TCP保证数据正确性；UDP可能丢包**

**（3）TCP传输速度慢；UDP速度快**

**（4）每一条TCP连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信**

**（5）TCP对系统资源要求较多，UDP对系统资源要求较少。**

## **82.tcp 为什么要三次握手，两次不行吗？为什么？**

为什么需要三次握手：

三次握手能确认双发收发功能都正常，缺一不可。

第一次握手：Client什么都不能确认；Server确认了对方发送正常。

第二次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己接收正常，对方发送正常。

第三次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己发送、接收正常，对方发送接收正常。

## **83.说一下 tcp 粘包是怎么产生的？**

发送方发送的多个数据包，到接收方缓冲区首尾相连，粘成一包，被接收。

2、原因

TCP 协议默认使用 Nagle 算法可能会把多个数据包一次发送到接收方。

应用程读取缓存中的数据包的速度小于接收数据包的速度，缓存中的多个数据包会被应用程序当成一个包一次读取。

3、处理方法

发送方使用 TCP\_NODELAY 选项来关闭 Nagle 算法

数据包增加开始符和结束，应用程序读取、区分数据包。

在数据包的头部定义整个数据包的长度，应用程序先读取数据包的长度，然后读取整个长度的包字节数据，保证读取的是单个包且完整。

## **84.OSI 的七层模型都有哪些？**



## **85.get 和 post 请求有哪些区别？**

在浏览器进行回退操作时,get请求是无害的,而post请求则会重新请求一次

get请求参数是连接在url后面的,而post请求参数是存放在requestbody内的

get请求因为浏览器对url长度有限制(不同浏览器长度限制不一样)对传参数量有限制,而post请求因为参数存放在requestbody内所以参数数量没有限制(事实上get请求也能在requestbody内携带参数,只不过不符合规定,有的浏览器能够获取到数据,而有的不能)

因为get请求参数暴露在url上,所以安全方面post比get更加安全

get请求浏览器会主动cache,post并不会,除非主动设置

get请求参数会保存在浏览器历史记录内,post请求并不会

get请求只能进行url编码,而post请求可以支持多种编码方式

get请求产生1个tcp数据包,post请求产生2个tcp数据包

浏览器在发送get请求时会将header和data一起发送给服务器,服务器返回200状态码,而在发送post请求时,会先将header发送给服务器,服务器返回100,之后再将data发送给服务器,服务器返回200 OK

## **86.如何实现跨域？**

浏览器对于javascript的同源策略的限制,例如a.cn下面的js不能调用b.cn中的js,对象或数据(因为a.cn和b.cn是不同域),所以跨域就出现了.

上面提到的,同域的概念又是什么呢??? 简单的解释就是相同域名,端口相同,协议相同

**同源策略:**

请求的url地址,必须与浏览器上的url地址处于同域上,也就是域名,端口,协议相同.

比如:我在本地上的域名是study.cn,请求另外一个域名一段数据

jsonp:

jsonp 全称是JSON with Padding,是为了解决跨域请求资源而产生的解决方案,是一种依靠开发人员创造出的一种非官方跨域数据交互协议。

一个是描述信息的格式，一个是信息传递双方约定的方法。

jsonp的产生:

1.AJAX直接请求普通文件存在跨域无权限访问的问题,不管是静态页面也好.

2.不过我们在调用js文件的时候又不受跨域影响,比如引入jquery框架的,或者是调用相片的时候

3.凡是拥有scr这个属性的标签都可以跨域例如<script><img><iframe>

4.如果想通过纯web端跨域访问数据只有一种可能,那就是把远程服务器上的数据装进js格式的文件里.

5.而json又是一个轻量级的数据格式,还被js原生支持

6.为了便于客户端使用数据，逐渐形成了一种非正式传输协议，人们把它称作JSONP，该协议的一个要点就是允许用户传递一个callback 参数给服务端，

## **87.说一下 JSONP 实现原理？**

假设a网页调用b网站的服务

a网站会准备一个方法，例如callme(args)

a网站在页面插入一个script标签，src指向b网站的地址，并带上callme作为参数

b网站处理后，把结果和回调方法的名字组成一个字符串返回，例如callme(‘ok’)

由于是script标签，所以字符串会被当成js解析执行，相当于调用到了callme方法

主要利用了script可以跨站点访问的特性，且只能用GET请求，需要服务端做点配合，并且需要信任服务器(安全考虑)。jquery的jsonp ajax只是封装了这个过程，让你看上去和普通ajax没什么区别，其实却一点关系都没有。

# 九、设计模式

## **88.说一下你熟悉的设计模式？**

**单例模式、工厂模式。**

## **89.简单工厂和抽象工厂有什么区别？**

**简单工厂模式**

**是由一个工厂对象创建产品实例，简单工厂模式的工厂类一般是使用静态方法，通过不同的参数的创建不同的对象实例**

**可以生产结构中的任意产品，不能增加新的产品**

**抽象工厂模式**

**提供一个创建一系列相关或相互依赖对象的接口，而无需制定他们具体的类，生产多个系列产品**

**生产不同产品族的全部产品，不能新增产品，可以新增产品族**

# **十、Spring/Spring MVC**

## **90.为什么要使用 spring？**

1.方便解耦，便于开发（Spring就是一个大工厂，可以将所有对象的创建和依赖关系维护都交给spring管理）

2.spring支持aop编程（spring提供面向切面编程，可以很方便的实现对程序进行权限拦截和运行监控等功能）

3.声明式事务的支持（通过配置就完成对事务的支持，不需要手动编程）

4.方便程序的测试，spring 对junit4支持，可以通过注解方便的测试spring 程序

5.方便集成各种优秀的框架（）

6.降低javaEE API的使用难度（Spring 对javaEE开发中非常难用的一些API 例如JDBC,javaMail,远程调用等，都提供了封装，是这些API应用难度大大降低）

把模块的配置参数统一管理，模块不需要自行读取配置。

Spring提供依赖注入，把依赖的模块自动推送进来，不需要模块自己拉取。

此外，Spring提供了对很多其他第三方框架的集成功能，减少了样板代码（boilerplate）。

## Spring bean的生命周期？

1.实例化 Instantiation

2.属性赋值 Populate

3.初始化 Initialization

4.销毁 Destruction

## **91.解释一下什么是 aop？**

在业务系统中，总有一些不得不处理的事情，我们

将这些重复性的代码抽取出来，放在专门的类中，

在通过spring的AOP的核心对代码段进行增强处理。

在不改变原代码的基础上进行功能增强。

有五种增强方式：前置增强，后置增强，环绕增强，引介增强。异常增强。

## **92.解释一下什么是 ioc？**

**通过引入IOC容器，利用依赖关系注入的方式，实现对象之间的解耦**。

**使用IOC框架应该注意什么**

使用IOC框架产品能够给我们的开发过程带来很大的好处，但是也要充分认识引入IOC框架的缺点，做到心中有数，杜绝滥用框架。

第一、软件系统中由于引入了第三方IOC容器，生成对象的步骤变得有些复杂，本来是两者之间的事情，又凭空多出一道手续，所以，我们在刚开始使用IOC框架的时候，会感觉系统变得不太直观。所以，引入了一个全新的框架，就会增加团队成员学习和认识的培训成本，并且在以后的运行维护中，还得让新加入者具备同样的知识体系。

第二、由于IOC容器生成对象是通过反射方式，在运行效率上有一定的损耗。如果你要追求运行效率的话，就必须对此进行权衡。

第三、具体到IOC框架产品(比如：Spring)来讲，需要进行大量的配制工作，比较繁琐，对于一些小的项目而言，客观上也可能加大一些工作成本。

第四、IOC框架产品本身的成熟度需要进行评估，如果引入一个不成熟的IOC框架产品，那么会影响到整个项目，所以这也是一个隐性的风险。

我们大体可以得出这样的结论：一些工作量不大的项目或者产品，不太适合使用IOC框架产品。另外，如果团队成员的知识能力欠缺，对于IOC框架产品缺乏深入的理解，也不要贸然引入。最后，特别强调运行效率的项目或者产品，也不太适合引入IOC框架产品，象WEB2.0网站就是这种情况。

## **93.spring 有哪些主要模块？**

**1. Spring Core**

**框架的最基础部分，提供 IoC 容器，对 bean 进行管理。**

**2.Spring Context**

**基于 bean，提供上下文信息，扩展出JNDI、EJB、电子邮件、国际化、校验和调度等功能。**

**3．Spring DAO**

**提供了JDBC的抽象层，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码，还提供了声明性事务管理方法。**

**4.Spring ORM**

**提供了常用的“对象/关系”映射APIs的集成层。 其中包括JPA、JDO、Hibernate、MyBatis 等。**

**5.Spring AOP**

**提供了符合AOP Alliance规范的面向方面的编程实现。**

**6.Spring Web**

**提供了基础的 Web 开发的上下文信息，可与其他 web 进行集成。**

**7.Spring Web MVC**

**提供了 Web 应用的 Model-View-Controller 全功能实现。**

## **94.spring 常用的注入方式有哪些？**

构造方法注入，setter注入，基于注解的注入。

## **95.spring 中的 bean 是线程安全的吗？**

 Spring容器中的Bean是否线程安全，容器本身并没有提供Bean的线程安全策略，因此可以说Spring容器中的Bean本身不具备线程安全的特性，但是具体还是要结合具体scope的Bean去研究。

Spring 的 bean 作用域（scope）类型

1、singleton:单例，默认作用域。

2、prototype:原型，每次创建一个新对象。

3、request:请求，每次Http请求创建一个新对象，适用于WebApplicationContext环境下。

4、session:会话，同一个会话共享一个实例，不同会话使用不用的实例。

5、global-session:全局会话，所有会话共享一个实例。

线程安全这个问题，要从单例与原型Bean分别进行说明。

原型Bean

对于原型Bean,每次创建一个新对象，也就是线程之间并不存在Bean共享，自然是不会有线程安全的问题。

单例Bean

对于单例Bean,所有线程都共享一个单例实例Bean,因此是存在资源的竞争。

如果单例Bean,是一个无状态Bean，也就是线程中的操作不会对Bean的成员执行查询以外的操作，那么这个单例Bean是线程安全的。比如Spring mvc 的 Controller、Service、Dao等，这些Bean大多是无状态的，只关注于方法本身。

对于有状态的bean，Spring官方提供的bean，一般提供了通过ThreadLocal去解决线程安全的方法，比如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等。

## **96.spring 支持几种 bean 的作用域？**

**spring 支持几种 bean 的作用域？**

**singleton：单例模式，在整个Spring IoC容器中，使用 singleton 定义的 bean 只有一个实例**

**prototype：原型模式，每次通过容器的getbean方法获取 prototype 定义的 bean 时，都产生一个新的 bean 实例**

**只有在 Web 应用中使用Spring时，request、session、global-session 作用域才有效**

**request：对于每次 HTTP 请求，使用 request 定义的 bean 都将产生一个新实例，即每次 HTTP 请求将会产生不同的 bean 实例。**

**session：同一个 Session 共享一个 bean 实例。**

**global-session：同 session 作用域不同的是，所有的Session共享一个Bean实例。**

## **97.spring 自动装配 bean 有哪些方式？**

.在Spring中，装配Bean有三种方式：

（1）在XML中进行显式配置

（2）在java中进行显式配置

（3）隐式的bean发现和自动装配（一般推荐使用自动装配bean的方式）

## **98.spring 事务实现方式有哪些？**

1）编程式事务管理对基于 POJO 的应用来说是唯一选择。我们需要在代码中调用 beginTransaction()、commit()、rollback()等事务管理相关的方法，这就是编程式事务管理。

　　（2）基于 TransactionProxyFactoryBean的声明式事务管理

　　（3）基于 @Transactional 的声明式事务管理

　　（4）基于Aspectj AOP配置事务

## **99.说一下 spring 的事务隔离？**

DEFAULT 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。

未提交读（read uncommited）:脏读，不可重复读，虚读都有可能发生

已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生

可重复读 （repeatable read） :避免脏读和不可重复读.但是虚读有可能发生.

串行化的 （serializable）:避免以上所有读问题.

## **100.说一下 spring mvc 运行流程？**

1. 客户端请求提交到DispatcherServlet（前段控制器）

2. 由DispatcherServlet控制器查询一个或多个HandlerMapping（处理器映射器），找到处理请求的Controller

3. DispatcherServlet将请求提交到Controller

4. Controller调用业务逻辑处理后，返回ModelAndView

5. DispatcherServlet查询一个或多个ViewResolver视图解析器，找到ModelAndView指定的视图

6. 视图负责将结果显示到客户端

## **101.spring mvc 有哪些组件？**

**前端控制器（DispatcherServlet）**

**处理器映射器（HandlerMapping）**

**处理器适配器（HandlerAdapter）**

**拦截器（HandlerInterceptor）**

**语言环境处理器（LocaleResolver）**

**主题解析器（ThemeResolver）**

**视图解析器（ViewResolver）**

**文件上传处理器（MultipartResolver）**

**异常处理器（HandlerExceptionResolver）**

**数据转换（DataBinder）**

**消息转换器（HttpMessageConverter）**

**请求转视图翻译器（RequestToViewNameTranslator）**

**页面跳转参数管理器（FlashMapManager）**

**处理程序执行链（HandlerExecutionChain）**

## **102.@RequestMapping 的作用是什么？**

**@RequestMapping是一个用来处理请求地址映射的注解，可用于类或者方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。**

**@RequestMapping注解有六个属性，下面进行详细的说明。**

**1.value, method**

**value:指定请求的实际地址，指定的地址可以是URI Template模式。**

**method:指定请求的method类型，GET、POST、PUT、DELETE等。**

**2.consumes, produces**

**consumes:指定处理请求的提交内容类型（Content-Type），例如application/json,text/html。**

**produces:指定返回的内容类型，仅当request请求头中的（Accept）类型中包含该指定类型才返回。**

**3.params, headers**

**params:指定request中必须包含某些参数值才让该方法处理。**

**headers:指定request中必须包含某些指定的header值，才能让该方法处理请求。**

## **103.@Autowired 的作用是什么？**

@Autowired 是一个注释，它可以对类成员变量、方法及构造函数进行标注，让 spring 完成 bean 自动装配的工作。

@Autowired 默认是按照类去匹配，配合 @Qualifier 指定按照名称去装配 bean。

### @**Autowired 与@Resource的区别？**

**① @Autowired 属于spring支持的注解，@Resource是jdk1.6支持的注解。**

**② @Autowired按byType自动注入，而@Resource默认按 byName自动注入，**

**@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。**

**③  @Autowired与@Resource都可以用来装配bean. 都可以写在字段上,或写在setter方法上。**

**④  @Autowired默认按类型装配（这个注解是属业spring的），默认情况下必须要求依赖对象必须存在，如果要允许null值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用。**

**⑤@Resource（这个注解属于J2EE的），默认按照名称进行装配，名称可以通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名进行安装名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。**

### spring的生命周期？

1. BeanFactory加载完bean definition和class，实例化除了bean对象。

2. 检查有没有实现BeanNameAware，有则调用setBeanName（得到bean id）

3. 检查有没有实现BeanClassLoaderAware，有则调用setBeanClassLoader。

4. 检查有没有实现EnvironmentAware，有则调用setEnvironment。

5. 检查有没有实现EmbeddedValueResolverAware，有则调用setEmbeddedValueResolver。

6. 检查有没有实现ResourceLoaderAware，有则调用setResourceLoader。

7. 检查有没有实现ApplicationEventPublisherAware，有则调用setApplicationEventPublisher。

8. 检查有没有实现MessageSourceAware，有则调用setMessageSource。

9. 检查有没有实现ApplicationContextAware，有则调用setApplicationContext。

10. 检查有没有实现ServletContextAware，有则调用setServletContext。

11. 调用BeanPostProcessors中的所有postProcessBeforeInitialization，对bean进行一些更进一步的配置。

12. 调用InitializingBean接口中的afterPropertiesSet执行bean自身提供的初始化代码。

13. 调用通过其他方式指定的init-method方法，执行bean自身的初始化。

14. 调用BeanPostProcessors中的所有postProcessAfterInitialization方法。

# **十一、Spring Boot/Spring Cloud**

## **104.什么是 spring boot？**

springboot就是Spring开源框架下的子项目，是Spring的一站式解决方案，主要是简化了spring的使用难度，降低了对配置文件的要求，使得开发人员能够更容易得上手。

2.为什么使用Spring Boot？

SpringBoot有很多得优点

简化了Spring配置文件，

没有代码和XML文件的生成

内置TomCat

能够独立运行

简化监控

## **105.为什么要用 spring boot？**

快速整合第三方框架（Maven依赖关系，Maven继承）；

打包方式完全采用注解化；

 Spring3.0之后采用注解方式启动SpringMVC，SpringBoot Web组件集成SpringMVC框架

简化XML的配置；

内置嵌入HTTP服务器（Tomcat（默认）、Jetty），降低了对环境的要求；

最终以Java应用程序进行执行（SpringBoot项目中没有Web.xml），运行中应用状态的监控；

## **106.spring boot 核心配置文件是什么？**

Spring Boot 有两种类型的配置文件，application 和 bootstrap 文件

Spring Boot会自动加载classpath目前下的这两个文件，文件格式为 properties 或 yml 格式

\*.properties 文件是 key=value 的形式

\*.yml 是 key: value 的形式

\*.yml 加载的属性是有顺序的，但不支持 @PropertySource 注解来导入配置，一般推荐用yml文件，看下来更加形象

bootstrap 配置文件是系统级别的，用来加载外部配置，如配置中心的配置信息，也可以用来定义系统不会变化的属性.bootstatp 文件的加载先于application文件

application 配置文件是应用级别的，是当前应用的配置文件

## **107.spring boot 配置文件有哪几种类型？它们有什么区别？**

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景

## **108.spring boot 有哪些方式可以实现热部署？**

Spring Loaded

Spring-boot-devtools

## **109.jpa 和 hibernate 有什么区别？**

JPA Java Persistence API，是Java EE 5的标准ORM接口，也是ejb3规范的一部分。

Hibernate，当今很流行的ORM框架，是JPA的一个实现，但是其功能是JPA的超集。

JPA和Hibernate之间的关系，可以简单的理解为JPA是标准接口，Hibernate是实现。那么Hibernate是如何实现与JPA的这种关系的呢。Hibernate主要是通过三个组件来实现的，及hibernate-annotation、hibernate-entitymanager和hibernate-core。

hibernate-annotation是Hibernate支持annotation方式配置的基础，它包括了标准的JPA annotation以及Hibernate自身特殊功能的annotation。

hibernate-core是Hibernate的核心实现，提供了Hibernate所有的核心功能。

hibernate-entitymanager实现了标准的JPA，可以把它看成hibernate-core和JPA之间的适配器，它并不直接提供ORM的功能，而是对hibernate-core进行封装，使得Hibernate符合JPA的规范。

Jpa是一种规范，而Hibernate是它的一种实现。除了Hibernate，还有EclipseLink(曾经的toplink)，OpenJPA等可供选择，所以使用Jpa的一个好处是，可以更换实现而不必改动太多代码。

## **110.什么是 spring cloud？**

而是在Spring Boot 基础上构建的,用于快速构建分布式系统的通用模式的工具集.

使用Spring Cloud开发的应用程序非常适合在Docker或者Pass上部署,所以又叫云原生应用(Cloud Native Application). 云原生可以简单的理解为面向云环境的软件架构;

## **spring cloud 断路器（Hystrix）的作用是什么？**

****该框架的目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix 具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能。****

## **spring cloud 的核心组件有哪些？**

Eureka注册中心

Config配置中心(也可以用阿波罗框架代替config)

Zuul网关

Ribbon负载均衡

Hystrix熔断器

Fegin远程调用

Actuator健康监测

# **十二、Hibernate**

## 113. 为什么要使用 hibernate？

1. hibernate 是对 jdbc 的封装，大大简化了数据访问层的繁琐的重复性代码。
2. hibernate 是一个优秀的 ORM 实现，很多程度上简化了 DAO 层的编码功能。
3. 可以很方便的进行数据库的移植工作。
4. 提供了缓存机制，是程序执行更改的高效。

## 114. 什么是 ORM 框架？

　　ORM（Object Relation Mapping）对象关系映射，是把数据库中的关系数据映射成为程序中的对象。

　　使用 ORM 的优点：提高了开发效率降低了开发成本、开发更简单更对象化、可移植更强。

## 115. hibernate 中如何在控制台查看打印的 SQL 语句？

　　在 Config 里面把 hibernate. show\_SQL 设置为 true 就可以。但不建议开启，开启之后会降低程序的运行效率。

**116. hibernate 有几种查询方式？**

　　三种：hql、原生 SQL、条件查询 Criteria。

**117. hibernate 实体类可以被定义为 final 吗？**

　　实体类可以定义为 final 类，但这样的话就不能使用 hibernate 代理模式下的延迟关联提供性能了，所以不建议定义实体类为 final。

**118. 在 hibernate 中使用 Integer 和 int 做映射有什么区别？**

　　Integer 类型为对象，它的值允许为 null，而 int 属于基础数据类型，值不能为 null。

**119. hibernate 是如何工作的？**

1. 读取并解析配置文件。
2. 读取并解析映射文件，创建 SessionFactory。
3. 打开 Session。
4. 创建事务。
5. 进行持久化操作。
6. 提交事务。
7. 关闭 Session。
8. 关闭 SessionFactory。

**120. get()和 load()的区别？**

1. 数据查询时，没有 OID 指定的对象，get() 返回 null；load() 返回一个代理对象。
2. load()支持延迟加载；get() 不支持延迟加载。

**121. 说一下 hibernate 的缓存机制？**

hibernate 常用的缓存有一级缓存和二级缓存：

一级缓存：也叫 Session 缓存，只在 Session 作用范围内有效，不需要用户干涉，由 hibernate 自身维护，可以通过：evict(object)清除 object 的缓存；clear()清除一级缓存中的所有缓存；flush()刷出缓存；

二级缓存：应用级别的缓存，在所有 Session 中都有效，支持配置第三方的缓存，如：EhCache。

**122. hibernate 对象有哪些状态？**

1. 临时/瞬时状态：直接 new 出来的对象，该对象还没被持久化（没保存在数据库中），不受 Session 管理。
2. 持久化状态：当调用 Session 的 save/saveOrupdate/get/load/list 等方法的时候，对象就是持久化状态。
3. 游离状态：Session 关闭之后对象就是游离状态。

**123. 在 hibernate 中 getCurrentSession 和 openSession 的区别是什么？**

1. getCurrentSession 会绑定当前线程，而 openSession 则不会。
2. getCurrentSession 事务是 Spring 控制的，并且不需要手动关闭，而 openSession 需要我们自己手动开启和提交事务。

**124. hibernate 实体类必须要有无参构造函数吗？为什么？**

　　hibernate 中每个实体类必须提供一个无参构造函数，因为 hibernate 框架要使用 reflection api，通过调用 ClassnewInstance() 来创建实体类的实例，如果没有无参的构造函数就会抛出异常。

# **十三、Mybatis**

## **125.mybatis 中 #{}和 ${}的区别是什么？**

#{}是预编译处理，${}是字符串替换。

（1）mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值。

（2）mybatis在处理${}时，就是把${}替换成变量的值。

（3）使用#{}可以有效的防止SQL注入，提高系统安全性。原因在于：预编译机制。

        预编译是提前对SQL语句进行预编译，而其后注入的参数将不会再进行SQL编译。我们知道，SQL注入是发生在编译的过程中，因为恶意注入了某些特殊字符，最后被编译成了恶意的执行操作。而预编译机制则可以很好的防止SQL注入。预编译完成之后，SQL的结构已经固定，即便用户输入非法参数，也不会对SQL的结构产生影响，从而避免了潜在的安全风险。

## MyBatis中resultType和resultMap的区别

如果只是返回一个值，比如说String或者int，那直接用resultType就行了，如果sql查询结果返回的列名和实体类中的字段名一致，可以使用resultType，MyBatis会自动把查询结果赋值给和字段名一致的字段。

resultMap要配置一下，表和类的一一对应关系，所以说就算你的字段名和你的实体类的属性名不一样也没关系，都会给你映射出来，但是，resultType就比较鸡肋了，必须字段名一样

## **126.mybatis 有几种分页方式？**

一.数组分页

原理：进行数据库查询操作时，获取到数据库中所有满足条件的记录，保存在应用的临时数组中，再通过List的subList方法，获取到满足条件的所有记录。

缺点：数据库查询并返回所有的数据，而我们需要的只是极少数符合要求的数据。当数据量少时，还可以接受。当数据库数据量过大时，每次查询对数据库和程序的性能都会产生极大的影响。

二.借助Sql语句进行分页

实现：通过sql语句实现分页也是非常简单的，只是需要改变我们查询的语句就能实现了，即在sql语句后面添加limit分页语句。

**三 拦截器分页**

MappedStatement是保存了xxMapper.xml中一个sql语句节点的所有信息的包装类，可以通过它获取到节点中的所有信息。在示例中我们拿到了id值，也就是方法的名称，通过名称区拦截所有需要分页的请求。

四.RowBounds实现分页

原理：通过RowBounds实现分页和通过数组方式分页原理差不多，都是一次获取所有符合条件的数据，然后在内存中对大数据进行操作，实现分页效果。只是数组分页需要我们自己去实现分页逻辑，这里更加简化而已。

存在问题：一次性从数据库获取的数据可能会很多，对内存的消耗很大，可能导师性能变差，甚至引发内存溢出。

分页方式：逻辑分页和物理分页。

**逻辑分页：** 使用 MyBatis 自带的 RowBounds 进行分页，它是一次性查询很多数据，然后在数据中再进行检索。

**物理分页：** 自己手写 SQL 分页或使用分页插件 PageHelper，去数据库查询指定条数的分页数据的形式。

## **127.RowBounds 是一次性查询全部结果吗？为什么？**

RowBounds 表面是在“所有”数据中检索数据，其实并非是一次性查询出所有数据，因为 MyBatis 是对 jdbc 的封装，在 jdbc 驱动中有一个 Fetch Size 的配置，它规定了每次最多从数据库查询多少条数据，假如你要查询更多数据，它会在你执行 next()的时候，去查询更多的数据。就好比你去自动取款机取 10000 元，但取款机每次最多能取 2500 元，所以你要取 4 次才能把钱取完。只是对于 jdbc 来说，当你调用 next()的时候会自动帮你完成查询工作。这样做的好处可以有效的防止内存溢出。

## **128.mybatis 逻辑分页和物理分页的区别是什么？**

1. 逻辑分页是一次性查询很多数据，然后再在结果中检索分页的数据。这样做弊端是需要消耗大量的内存、有内存溢出的风险、对数据库压力较大。
2. 物理分页是从数据库查询指定条数的数据，弥补了一次性全部查出的所有数据的种种缺点，比如需要大量的内存，对数据库查询压力较大等问题。

## **129.mybatis 是否支持延迟加载？延迟加载的原理是什么？**

MyBatis 支持延迟加载，设置 lazyLoadingEnabled=true 即可。

延迟加载的原理的是调用的时候触发加载，而不是在初始化的时候就加载信息。比如调用 a. getB(). getName()，这个时候发现 a. getB() 的值为 null，此时会单独触发事先保存好的关联 B 对象的 SQL，先查询出来 B，然后再调用 a. setB(b)，而这时候再调用 a. getB(). getName() 就有值了，这就是延迟加载的基本原理。

## **130.说一下 mybatis 的一级缓存和二级缓存？**

一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，它的声明周期是和 SQLSession 一致的，有多个 SQLSession 或者分布式的环境中数据库操作，可能会出现脏数据。当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认一级缓存是开启的。

二级缓存：也是基于 PerpetualCache 的 HashMap 本地缓存，不同在于其存储作用域为 Mapper 级别的，如果多个SQLSession之间需要共享缓存，则需要使用到二级缓存，并且二级缓存可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态)。

开启二级缓存数据查询流程：二级缓存 -> 一级缓存 -> 数据库。

缓存更新机制：当某一个作用域(一级缓存 Session/二级缓存 Mapper)进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

## **131.mybatis 和 hibernate 的区别有哪些？**

灵活性：MyBatis 更加灵活，自己可以写 SQL 语句，使用起来比较方便。

可移植性：MyBatis 有很多自己写的 SQL，因为每个数据库的 SQL 可以不相同，所以可移植性比较差。

学习和使用门槛：MyBatis 入门比较简单，使用门槛也更低。

二级缓存：hibernate 拥有更好的二级缓存，它的二级缓存可以自行更换为第三方的二级缓存。

## **132.mybatis 有哪些执行器（Executor）？**

MyBatis 有三种基本的Executor执行器：

1. SimpleExecutor：每执行一次 update 或 select 就开启一个 Statement 对象，用完立刻关闭 Statement 对象；
2. ReuseExecutor：执行 update 或 select，以 SQL 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后不关闭 Statement 对象，而是放置于 Map 内供下一次使用。简言之，就是重复使用 Statement 对象；
3. BatchExecutor：执行 update（没有 select，jdbc 批处理不支持 select），将所有 SQL 都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement 对象，每个 Statement 对象都是 addBatch()完毕后，等待逐一执行 executeBatch()批处理，与 jdbc 批处理相同。

## **133.mybatis 分页插件的实现原理是什么？**

分页插件的基本原理是使用 MyBatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 SQL，然后重写 SQL，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

## **134.mybatis 如何编写一个自定义插件？**

**自定义插件实现原理**

MyBatis 自定义插件针对 MyBatis 四大对象（Executor、StatementHandler、ParameterHandler、ResultSetHandler）进行拦截：

* 1. Executor：拦截内部执行器，它负责调用 StatementHandler 操作数据库，并把结果集通过 ResultSetHandler 进行自动映射，另外它还处理了二级缓存的操作；
  2. StatementHandler：拦截 SQL 语法构建的处理，它是 MyBatis 直接和数据库执行 SQL 脚本的对象，另外它也实现了 MyBatis 的一级缓存；
  3. ParameterHandler：拦截参数的处理；
  4. ResultSetHandler：拦截结果集的处理。

**自定义插件实现关键**

MyBatis 插件要实现 Interceptor 接口，接口包含的方法，如下：

public interface Interceptor {
Object intercept(Invocation invocation) throws Throwable;
Object plugin(Object target);
void setProperties(Properties properties);
}

1. setProperties 方法是在 MyBatis 进行配置插件的时候可以配置自定义相关属性，即：接口实现对象的参数配置；
2. plugin 方法是插件用于封装目标对象的，通过该方法我们可以返回目标对象本身，也可以返回一个它的代理，可以决定是否要进行拦截进而决定要返回一个什么样的目标对象，官方提供了示例：return Plugin. wrap(target, this)；
3. intercept 方法就是要进行拦截的时候要执行的方法。
4. **自定义插件实现示例**

官方插件实现：

@Intercepts({@Signature(type = Executor. class, method = "query",
args = {MappedStatement. class, Object. class, RowBounds. class, ResultHandler. class})})public class TestInterceptor implements Interceptor {
public Object intercept(Invocation invocation) throws Throwable {
Object target = invocation. getTarget(); //被代理对象
Method method = invocation. getMethod(); //代理方法
Object[] args = invocation. getArgs(); //方法参数
// do something . . . . . . 方法拦截前执行代码块
Object result = invocation. proceed();
// do something . . . . . . . 方法拦截后执行代码块
return result;
}
public Object plugin(Object target) {
return Plugin. wrap(target, this);
}
}

# **十四、RabbitMQ**

## **135.rabbitmq 的使用场景有哪些？**

场景说明：用户注册后，需要发注册邮件和注册短信,传统的做法有两种1.串行的方式;2.并行的方式

场景：双11是购物狂节,用户下单后,订单系统需要通知库存系统,传统的做法就是订单系统调用库存系统的接口.

场景:秒杀活动，一般会因为流量过大，导致应用挂掉,为了解决这个问题，一般在应用前端加入消息队列。

作用:

1.可以控制活动人数，超过此一定阀值的订单直接丢弃(我为什么秒杀一次都没有成功过呢^^)

2.可以缓解短时间的高流量压垮应用(应用程序按自己的最大处理能力获取订单)

## **136.rabbitmq 有哪些重要的角色？**

Broker:它提供一种传输服务,它的角色就是维护一条从生产者到消费者的路线，保证数据能按照指定的方式进行传输,

Exchange：消息交换机,它指定消息按什么规则,路由到哪个队列。

Queue:消息的载体,每个消息都会被投到一个或多个队列。

Binding:绑定，它的作用就是把exchange和queue按照路由规则绑定起来.

Routing Key:路由关键字,exchange根据这个关键字进行消息投递。

vhost:虚拟主机,一个broker里可以有多个vhost，用作不同用户的权限分离。

Producer:消息生产者,就是投递消息的程序.

Consumer:消息消费者,就是接受消息的程序.

Channel:消息通道,在客户端的每个连接里,可建立多个channel.

## **137.rabbitmq 有哪些重要的组件？**

1、Exchange:

　　如果不指定Exchange的话，RabbitMQ默认使用，(AMQP default)注意一下，需要将routing key等于queue name相同

2、name、type：

　　fanout（效率最好，不需要routing key，routing key如何设置都可以）、direct、topic（#一个或多个，\*一个）、headers

3、Auto Delete：

　　当最后一个Binding到Exchange的Queue删除之后，自动删除该Exchange

4、Binding：

　　Exchange和Queue之间的连接关系，Exchange之间也可以Binding

5、Queue：

　　实际物理上存储消息的

6、Durability：

　　是否持久化，Durable：是，即使服务器重启，这个队列也不会消失，Transient：否

7、Exclusive：

　　这个queue只能由一个exchange监听restricted to this connection，使用场景：顺序消费

8、Message：

　　由properties（有消息优先级、延迟等特性）和Body（Payload消息内容）组成，还有content\_type、content\_encoding、priority

correlation\_id、reply\_to、expiration、message\_id等属性

## **138.rabbitmq 中 vhost 的作用是什么？**

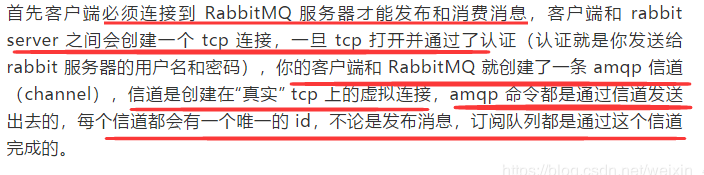
vhost本质上是一个mini版的RabbitMQ服务器，拥有自己的队列、绑定、交换器和权限控制；

vhost通过在各个实例间提供逻辑上分离，允许你为不同应用程序安全保密地运行数据；

vhost是AMQP概念的基础，必须在连接时进行指定，RabbitMQ包含了默认vhost：“/”；

当在RabbitMQ中创建一个用户时，用户通常会被指派给至少一个vhost，并且只能访问被指派vhost内的队列、交换器和绑定，vhost之间是绝对隔离的。

## **139.rabbitmq 的消息是怎么发送的？**



## **140.rabbitmq 怎么保证消息的稳定性？**

1. 提供了事务的功能。
2. 通过将 channel 设置为 confirm（确认）模式。

## **141.rabbitmq 怎么避免消息丢失？**

**1.消息持久化**

RabbitMQ 的消息默认存放在内存上面，如果不特别声明设置，消息不会持久化保存到硬盘上面的，如果节点重启或者意外crash掉，消息就会丢失。

所以就要对消息进行持久化处理。如何持久化，下面具体说明下：

要想做到消息持久化，必须满足以下三个条件，缺一不可。

1） Exchange 设置持久化

2）Queue 设置持久化

3）Message持久化发送：发送消息设置发送模式deliveryMode=2，代表持久化消息

**2.ACK确认机制**

多个消费者同时收取消息，比如消息接收到一半的时候，一个消费者死掉了(逻辑复杂时间太长，超时了或者消费被停机或者网络断开链接)，如何保证消息不丢？

这个使用就要使用Message acknowledgment 机制，就是消费端消费完成要通知服务端，服务端才把消息从内存删除。

这样就解决了，及时一个消费者出了问题，没有同步消息给服务端，还有其他的消费端去消费，保证了消息不丢的case。

**3.设置集群镜像模式**

1）单节点模式：最简单的情况，非集群模式，节点挂了，消息就不能用了。业务可能瘫痪，只能等待。

2）普通模式：默认的集群模式，某个节点挂了，该节点上的消息不能用，有影响的业务瘫痪，只能等待节点恢复重启可用（必须持久化消息情况下）。

3）镜像模式：把需要的队列做成镜像队列，存在于多个节点，属于RabbitMQ的HA方案

**下面介绍下三种HA策略模式：**

1）同步至所有的

2）同步最多N个机器

3）只同步至符合指定名称的nodes

*命令处理HA策略模版：rabbitmqctl set\_policy [-p Vhost] Name Pattern Definition [Priority]*

1）为每个以“rock.wechat”开头的队列设置所有节点的镜像，并且设置为自动同步模式

rabbitmqctl set\_policy ha-all "^rock.wechat" '{"ha-mode":"all","ha-sync-mode":"automatic"}'

rabbitmqctl set\_policy -p rock ha-all "^rock.wechat" '{"ha-mode":"all","ha-sync-mode":"automatic"}'

2）为每个以“rock.wechat.”开头的队列设置两个节点的镜像，并且设置为自动同步模式

rabbitmqctl set\_policy -p rock ha-exacly "^rock.wechat" \

'{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}'

3）为每个以“node.”开头的队列分配指定的节点做镜像

rabbitmqctl set\_policy ha-nodes "^nodes\." \

'{"ha-mode":"nodes","ha-params":["rabbit@nodeA", "rabbit@nodeB"]}'

但是：HA 镜像队列有一个很大的缺点就是：   **系统的吞吐量会有所下降**

**4.消息补偿机制**

为什么还要消息补偿机制呢？难道消息还会丢失，没错，系统是在一个复杂的环境，不要想的太简单了，虽然以上的三种方案，基本可以保证消息的高可用不丢失的问题，

但是作为有追求的程序员来讲，要绝对保证我的系统的稳定性，有一种危机意识。

比如：持久化的消息，保存到硬盘过程中，当前队列节点挂了，存储节点硬盘又坏了，消息丢了，怎么办？

产线网络环境太复杂，所以不知数太多，消息补偿机制需要建立在消息要写入DB日志，发送日志，接受日志，两者的状态必须记录。

然后根据DB日志记录check 消息发送消费是否成功，不成功，进行消息补偿措施，重新发送消息处理。

## **142.要保证消息持久化成功的条件有哪些？**

queue，exchange和Message都持久化；

　　引入RabbitMQ的mirrored-queue即镜像队列，这个相当于配置了副本，当master在此特殊时间内crash掉，可以自动切换到slave，这样有效的保障了HA,；

　　要在producer引入事务机制或者Confirm机制来确保消息已经正确的发送至broker端

## **143.rabbitmq 持久化有什么缺点？**

持久化的缺地就是降低了服务器的吞吐量，

因为使用的是磁盘而非内存存储，

从而降低了吞吐量。可尽量使用 ssd 硬盘来缓解吞吐量的问题。

## **144.rabbitmq 有几种广播类型？**

三种广播模式：

1 fanout: 所有bind到此exchange的queue都可以接收消息

（纯广播，绑定到RabbitMQ的接受者都能收到消息）；

2 direct: 通过routingKey和exchange决定的那个唯一的queue可以接收消息；

3 topic: 所有符合routingKey(此时可以是一个表达式)的routingKey所bind的queue可以接收消息；

## **145.rabbitmq 怎么实现延迟消息队列？**

1. 通过消息过期后进入死信交换器，
2. 再由交换器转发到延迟消费队列，实现延迟功能；
4. 使用 RabbitMQ-delayed-message-exchange 插件实现延迟功能

## **146.rabbitmq 集群有什么用？**

集群主要有以下两个用途：

高可用：某个服务器出现问题，整个 RabbitMQ 还可以继续使用；

高容量：集群可以承载更多的消息量。

## **147.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

## **148.rabbitmq 集群搭建需要注意哪些问题？**

各节点之间使用“--link”连接，此属性不能忽略。

各节点使用的 erlang cookie 值必须相同，此值相当于“秘钥”的功能，用于各节点的认证。

整个集群中必须包含一个磁盘节点。

## **149.rabbitmq 每个节点是其他节点的完整拷贝吗？为什么？**

不是，原因有以下两个：

存储空间的考虑：如果每个节点都拥有所有队列的完全拷贝，

这样新增节点不但没有新增存储空间，反而增加了更多的冗余数据；

性能的考虑：如果每条消息都需要完整拷贝到每一个集群节点，

那新增节点并没有提升处理消息的能力，最多是保持和单节点相同的性能甚至是更糟。

## **150.rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况？**

如果唯一磁盘的磁盘节点崩溃了，不能进行以下操作：

不能创建队列

不能创建交换器

不能创建绑定

不能添加用户

不能更改权限

不能添加和删除集群节点

唯一磁盘节点崩溃了，集群是可以保持运行的，但你不能更改任何东西

## **151.rabbitmq 对集群节点停止顺序有要求吗？**

RabbitMQ 对集群的停止的顺序是有要求的，

应该先关闭内存节点，最后再关闭磁盘节点。如果顺序恰好相反的话，可能会造成消息的丢失。

# **十五、Kafka**

## **152.kafka 可以脱离 zookeeper 单独使用吗？为什么？**

kafka 不能脱离 zookeeper 单独使用，

因为 kafka 使用 zookeeper 管理和协调 kafka 的节点服务器。

## **153.kafka 有几种数据保留的策略？**

按照过期时间保留

按照存储的消息大小保留。

## **154.kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？**

这个时候 kafka 会执行数据清除工作，时间和大小不论那个满足条件，都会清空数据。

## **155.什么情况会导致 kafka 运行变慢？**

cpu 性能瓶颈

磁盘读写瓶颈

网络瓶颈

## **156.使用 kafka 集群需要注意什么？**

集群的数量不是越多越好，最好不要超过 7 个，

因为节点越多，消息复制需要的时间就越长，整个群组的吞吐量就越低。

集群数量最好是单数，因为超过一半故障集群就不能用了，设置为单数容错率更高。

**Kafka如何保证数据不丢失**

1.生产者数据的不丢失

kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到，其中状态有0,1，-1。

如果是同步模式：ack机制能够保证数据的不丢失，如果ack设置为0，风险很大，一般不建议设置为0。即使设置为1，也会随着leader宕机丢失数据。

producer.type=sync

request.required.acks=1

如果是异步模式：也会考虑ack的状态，除此之外，异步模式下的有个buffer，通过buffer来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果buffer满了数据还没有发送出去，有个选项是配置是否立即清空buffer。可以设置为-1，永久阻塞，也就数据不再生产。

异步模式下，即使设置为-1。也可能因为程序员的不科学操作，操作数据丢失，比如kill -9，但这是特别的例外情况。

producer.type=async

request.required.acks=1

queue.buffering.max.ms=5000

queue.buffering.max.messages=10000

queue.enqueue.timeout.ms = -1

batch.num.messages=200

结论：producer有丢数据的可能，但是可以通过配置保证消息的不丢失。

**消费者数据的不丢失**

通过offset commit 来保证数据的不丢失，kafka自己记录了每次消费的offset数值，下次继续消费的时候，会接着上次的offset进行消费。

而offset的信息在kafka0.8版本之前保存在zookeeper中，在0.8版本之后保存到topic中，即使消费者在运行过程中挂掉了，再次启动的时候会找到offset的值，找到之前消费消息的位置，接着消费，由于offset的信息写入的时候并不是每条消息消费完成后都写入的，所以这种情况有可能会造成重复消费，但是不会丢失消息。

唯一例外的情况是，我们在程序中给原本做不同功能的两个consumer组设置KafkaSpoutConfig.bulider.setGroupid的时候设置成了一样的groupid，这种情况会导致这两个组共享同一份数据，就会产生组A消费partition1，partition2中的消息，组B消费partition3的消息，这样每个组消费的消息都会丢失，都是不完整的。  为了保证每个组都独享一份消息数据，groupid一定不要重复才行

**kafka集群中的broker的数据不丢失**

每个broker中的partition我们一般都会设置有replication（副本）的个数，生产者写入的时候首先根据分发策略（有partition按partition，有key按key，都没有轮询）写入到leader中，follower（副本）再跟leader同步数据，这样有了备份，也可以保证消息数据的不丢失。

# **十六、Zookeeper**

## **157.zookeeper 是什么？**

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

## **158.zookeeper 都有哪些功能？**

1. 集群管理：监控节点存活状态、运行请求等；

2. 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；

3. 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。

4. 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。

谈下你对 ZAB 协议的了解？

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 Zookeeper 集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有服务器进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步。当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

## Zookeeper 怎么保证主从节点的状态同步？

Zookeeper 的核心是原子广播机制，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，它们分别是恢复模式和广播模式。

1. 恢复模式

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。

2. 广播模式

一旦 leader 已经和多数的 follower 进行了状态同步后，它就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 ZooKeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

## **159.zookeeper 有几种部署模式？**

1. 单机部署：一台集群上运行；

2. 集群部署：多台集群运行；

3. 伪集群部署：一台集群启动多个 Zookeeper 实例运行。

## **160.zookeeper 怎么保证主从节点的状态同步？**

Zookeeper的核心是原子广播机制，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，它们分别是恢复模式和广播模式。

(1) 恢复模式

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。

(2) 广播模式

一旦Leader已经和多数的Follower进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个Server加入ZooKeeper服务中，它会在恢复模式下启动，发现Leader，并和Leader进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper服务一直维持在Broadcast状态，直到Leader崩溃了或者Leader失去了大部分的Followers支持。

## **161.集群中为什么要有主节点？**

## **162.集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？**

**可以**

首先需要明确zookeeper选举的规则：leader选举，要求 **可用节点数量 > 总节点数量/2** 。注意 是 > , 不是 ≥。

## **163.说一下 zookeeper 的通知机制？**

client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

集群中为什么要有主节点？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

说一下两阶段提交和三阶段提交的过程？分别有什么问题？

两阶段提交协议 2PC

1. 第一阶段（投票阶段）：

（1）协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应；

（2）参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。 （3）各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

2. 第二阶段（提交执行阶段）： 当协调者节点从所有参与者节点获得的相应消息都为”同意”时：

（1）协调者节点向所有参与者节点发出”正式提交(commit)”的请求；

（2）参与者节点正式完成操作，并释放在整个事务期间内占用的资源；

（3）参与者节点向协调者节点发送”完成”消息；

（4）协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

两阶段提交存在的问题：

1. 执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态；

2. 参与者发生故障：协调者需要给每个参与者额外指定超时机制，超时后整个事务失败；

3. 协调者发生故障：参与者会一直阻塞下去。需要额外的备机进行容错；

4. 二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

三阶段提交协议 3PC

与两阶段提交不同的是，三阶段提交有两个改动点：

1. 引入超时机制。同时在协调者和参与者中都引入超时机制；

2. 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 CanCommit、PreCommit、DoCommit 三个阶段。

1. CanCommit 阶段 3PC 的 CanCommit 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

（1）事务询问：协调者向参与者发送 CanCommit 请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

（2）响应反馈：参与者接到 CanCommit 请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回 Yes 响应，并进入预备状态。否则反馈 No。

2. PreCommit 阶段 协调者根据参与者的反应情况来决定是否可以继续事务的 PreCommit 操作。根据响应情况，有以下两种可能：

假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行。

（1）发送预提交请求：协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段。

（2）事务预提交：参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中。 （3）响应反馈：如果参与者成功的执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。 （1）发送中断请求：协调者向所有参与者发送 abort 请求。 （2）中断事务：参与者收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

3. doCommit 阶段 该阶段进行真正的事务提交，也可以分为以下两种情况。

3.1 执行提交

（1）发送提交请求：协调接收到参与者发送的 ACK 响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。

（2）事务提交：参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

（3）响应反馈：事务提交完之后，向协调者发送 ACK 响应。

（4）完成事务：协调者接收到所有参与者的 ACK 响应之后，完成事务。

3.2 中断事务

协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

（1）发送中断请求：协调者向所有参与者发送 abort 请求。

（2）事务回滚：参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

（3）反馈结果：参与者完成事务回滚之后，向协调者发送 ACK 消息。

（4）中断事务：协调者接收到参与者反馈的 ACK 消息之后，执行事务的中断。

三阶段提交的问题：

网络分区可能会带来问题。需要四阶段解决：四阶段直接调用远程服务的数据状态，确定当前数据一致性的情况。

Zookeeper 宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

Zookeeper 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 Zookeeper 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。所以：

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票 > 1.5)

2 个节点的 cluster 就不能挂掉任何1个节点了(leader 可以得到 1 票 <= 1)

11. 说下四种类型的数据节点 Znode？

1. PERSISTENT：持久节点，除非手动删除，否则节点一直存在于 Zookeeper 上。

2. EPHEMERAL：临时节点，临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 Zookeeper连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

3. PERSISTENT\_SEQUENTIAL：持久顺序节点，基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

4. EPHEMERAL\_SEQUENTIAL：临时顺序节点，基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

Zookeeper 和 Dubbo 的关系？

Dubbo 的将注册中心进行抽象，是得它可以外接不同的存储媒介给注册中心提供服务，有 ZooKeeper，Memcached，Redis 等。

引入了 ZooKeeper 作为存储媒介，也就把 ZooKeeper 的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时 候就需要分流，负载均衡就是为了分流而存在的，一个 ZooKeeper 群配合相应的 Web 应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不 够，节点之间的数据和资源需要同步，ZooKeeper 集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动 的时候，向 ZooKeeper 上的指定节点 /dubbo/${serviceName}/providers 目录下写入自己的 URL 地址，这个操作就完成了服务的发布。 其他特性还有 Mast 选举，分布式锁等。

# **MySql**

## 163左连接、右连接、全连接的区别？

左（外）连接（LEFT JOIN），以左表为基准，查询出左表所有的数据和右表中连接字段相等的记录，如果右表中没有对应数据，则在左表记录后显示为空（NULL）.如果把两个表分别看成一个集合的话，则显示的结果为JOIN左边的集合。  
  
　　同理，右（外）连接(RIGHT JOIN )是以右表为基准，查询出右表所有的数据和左表中连接字段相等的记录，如果左表没有对应数据则在右表对应数据行显示为空（NULL）.如果把两个表分别看成一个集合的话，则显示的结果为JOIN右边的集合。

内连接(INNER JOIN )是查询出两个表对应的数据，如果把两个表分别看成一个集合的话，内连接的结果即为两个表的交集。

　　全连接（FULL JOIN ）将两个表的数据全部查出来，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，如果把两个表分别看成一个集合的话，全外连接的结果即为两个表的并集。

## **164.数据库的三范式是什么？**

第一范式：当关系模式R的所有属性都不能在分解为更基本的数据单位时，称R是满足第一范式的，简记为1NF。满足第一范式是关系模式规范化的最低要

求，否则，将有很多基本操作在这样的关系模式中实现不了。

第二范式：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式，简记为2NF。

第三范式：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，简记为3NF.

三大范式只是一般设计数据库的基本理念，可以建立冗余较小、结构合理的数据库。如果有特殊情况，当然要特殊对待，数据库设计最重要的是看需求跟性能，需求>性能>表结构。所以不能一味的去追求范式建立数据库

## **165.一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？**

一般情况下，我们创建的表的类型是InnoDB，如果新增一条记录（不重启mysql的情况下），这条记录的id是8；但是如果重启（文中提到的）MySQL的话，这条记录的ID是6。因为InnoDB表只把自增主键的最大ID记录到内存中，所以重启数据库或者对表OPTIMIZE操作，都会使最大ID丢失。

但是，如果我们使用表的类型是MylSAM，那么这条记录的ID就是8。因为MylSAM表会把自增主键的最大ID记录到数据文件里面，重启MYSQL后，自增主键的最大ID也不会丢失。

注：如果在这7条记录里面删除的是中间的几个记录（比如删除的是3,4两条记录），重启MySQL数据库后，insert一条记录后，ID都是8。因为内存或者数据库文件存储都是自增主键最大ID

## **166.如何获取当前数据库版本？**

select version()

mysql -V

## **167.说一下 ACID 是什么？**

ACID 一般是指数据库事务的ACID

一个事务一般是指多个操作的集合，比如插入数据库分两段插入，第二次插入错误，第一次插入操作也需要回退

ACID的翻译

1.Atomicity 原子性

2.Consistency 一致性

3.Isolation 隔离性

4.Durability 耐久性

原子性，指的是整个事务是一个独立的单元，要么操作成功，要么操作不成功

一致性，事务必须要保持和系统处于一致的状态（如果不一致会导致系统其它的方出现bug）

隔离性，事务是并发控制机制，他们的交错也需要一致性，隔离隐藏，一般通过悲观或者乐观锁实现

耐久性，一个成功的事务将永久性地改变系统的状态，所以在它结束之前，所有导致状态的变化都

              记录在一个持久的事务日志中

## **168.char 和 varchar 的区别是什么？**

1、char(n)类型

    char类型时定长的类型，即当定义的是char(10)，输入的是"abc"这三个字符时，它们占的空间一样是10个字节，包括7个空字节。当输入的字符长度超过指定的数时，char会截取超出的字符。而且，当存储char值时，MySQL是自动删除输入字符串末尾的空格。

    char是适合存储很短的、一般固定长度的字符串。例如，char非常适合存储密码的MD5值，因为这是一个定长的值。对于非常短的列，char比varchar在存储空间上也更有效率。

2、varchar(n)类型

     varchar(n)类型用于存储可变长的，长度为n个字节的可变长度且非Unicode的字符数据。n必须是介于1和8000之间的数值，存储大小为输入数据的字节的实际长度+1/2. 比如varchar(10), 然后输入abc三个字符，那么实际存储大小为3个字节。除此之外，varchar还需要使用1或2个额外字节记录字符串的长度，如果列的最大长度小于等于255字节（是定义的最长长度，不是实际长度），则使用1个字节表示长度，否则使用2个字节来表示。

## **169.float 和 double 的区别是什么？**

两者的主要区别如下：

1.在内存中占有的字节数不同

单精度浮点数在机内存占4个字节

双精度浮点数在机内存占8个字节

2.有效数字位数不同

单精度浮点数有效数字8位

双精度浮点数有效数字16位

3.数值取值范围

单精度浮点数的表示范围：-3.40E+38~3.40E+38

双精度浮点数的表示范围：-1.79E+308~-1.79E+308

4.在程序中处理速度不同

一般来说，CPU处理单精度浮点数的速度比处理双精度浮点数快，如果不声明，默认小数为double类型，所以如果要用float的话，必须进行强转

## **170.mysql 的内连接、左连接、右连接有什么区别？**

具体答案详情可查看：

<https://blog.csdn.net/plg17/article/details/78758593>

## **171.mysql 索引是怎么实现的？**

索引**是一个排序的列表**，在这个列表中存储着索引的值和包含这个值的数据所在行的物理地址，在数据十分庞大的时候，索引可以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。

1、索引需要占用**磁盘空间**，因此在创建索引时要考虑到磁盘空间是否足够

2、创建索引时需要**对表加锁**，因此实际操作中需要在业务空闲期间进行

LIKE以%开头会导致索引失效；使用覆盖索引解决之

索引的优缺点

优势：可以快速检索，减少I/O次数，加快检索速度；根据索引分组和排序，可以加快分组和排序；

劣势：索引本身也是表，因此会占用存储空间，一般来说，索引表占用的空间的数据表的1.5倍；索引表的维护和创建需要时间成本，这个成本随着数据量增大而增大；构建索引会降低数据表的修改操作（删除，添加，修改）的效率，因为在修改数据表的同时还需要修改索引表；

**常见的索引类型有：主键索引、唯一索引、普通索引、全文索引、组合索引、**覆盖索引

1、主键索引：即主索引，根据主键pk\_clolum（length）建立索引，不允许重复，不允许空值；

ALTER TABLE 'table\_name' ADD PRIMARY KEY pk\_index('col')；

2、唯一索引：用来建立索引的列的值必须是唯一的，允许空值

ALTER TABLE 'table\_name' ADD UNIQUE index\_name('col')；

3、普通索引：用表中的普通列构建的索引，没有任何限制

ALTER TABLE 'table\_name' ADD INDEX index\_name('col')；

4、全文索引：用大文本对象的列构建的索引（下一部分会讲解）



ALTER TABLE 'table\_name' ADD FULLTEXT INDEX ft\_index('col')；

5、组合索引：用多个列组合构建的索引，这多个列中的值不允许有空值

ALTER TABLE 'table\_name' ADD INDEX index\_name('col1','col2','col3')；

\*遵循“最左前缀”原则，把最常用作为检索或排序的列放在最左，依次递减，组合索引相当于建立了col1,col1col2,col1col2col3三个索引，而col2或者col3是不能使用索引的。

\*在使用组合索引的时候可能因为列名长度过长而导致索引的key太大，导致效率降低，在允许的情况下，可以只取col1和col2的前几个字符作为索引

ALTER TABLE 'table\_name' ADD INDEX index\_name(col1(4),col2（3))；

表示使用col1的前4个字符和col2的前3个字符作为索引

索引类型

Mysql目前主要有以下几种索引类型：FULLTEXT（全文索引），HASH，BTREE

什么时候要使用索引？

主键自动建立唯一索引；

经常作为查询条件在WHERE或者ORDER BY 语句中出现的列要建立索引；

作为排序的列要建立索引；

查询中与其他表关联的字段，外键关系建立索引

高并发条件下倾向组合索引；

用于聚合函数的列可以建立索引，例如使用了max(column\_1)或者count(column\_1)时的column\_1就需要建立索引

什么时候不要使用索引？

经常增删改的列不要建立索引；

有大量重复的列不建立索引；

表记录太少不要建立索引。只有当数据库里已经有了足够多的测试数据时，它的性能测试结果才有实际参考价值。如果在测试数据库里只有几百条数据记录，它们往往在执行完第一条查询命令之后就被全部加载到内存里，这将使后续的查询命令都执行得非常快--不管有没有使用索引。只有当数据库里的记录超过了1000条、数据总量也超过了MySQL服务器上的内存总量时，数据库的性能测试结果才有意义。

索引失效的情况：

在组合索引中不能有列的值为NULL，如果有，那么这一列对组合索引就是无效的。

在一个SELECT语句中，索引只能使用一次，如果在WHERE中使用了，那么在ORDER BY中就不要用了。

LIKE操作中，'%aaa%'不会使用索引，也就是索引会失效，但是‘aaa%’可以使用索引。

在索引的列上使用表达式或者函数会使索引失效，例如：select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。其它通配符同样，也就是说，在查询条件中使用正则表达式时，只有在搜索模板的第一个字符不是通配符的情况下才能使用索引。

在查询条件中使用不等于，包括<符号、>符号和！=会导致索引失效。特别的是如果对主键索引使用！=则不会使索引失效，如果对主键索引或者整数类型的索引使用<符号或者>符号不会使索引失效。（经erwkjrfhjwkdb同学提醒，不等于，包括&lt;符号、>符号和！，如果占总记录的比例很小的话，也不会失效）

在查询条件中使用IS NULL或者IS NOT NULL会导致索引失效。

字符串不加单引号会导致索引失效。更准确的说是类型不一致会导致失效，比如字段email是字符串类型的，使用WHERE email=99999 则会导致失败，应该改为WHERE email='99999'。

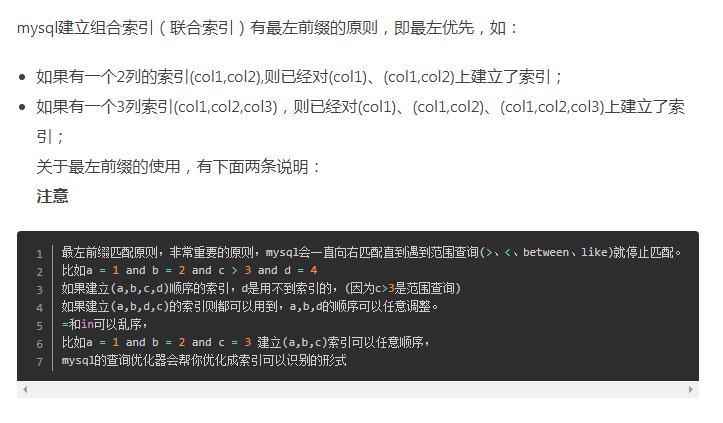
在查询条件中使用OR连接多个条件会导致索引失效，除非OR链接的每个条件都加上索引，这时应该改为两次查询，然后用UNION ALL连接起来。

如果排序的字段使用了索引，那么select的字段也要是索引字段，否则索引失效。特别的是如果排序的是主键索引则select \* 也不会导致索引失效。

尽量不要包括多列排序，如果一定要，最好为这队列构建组合索引；

索引的优化

1、最左前缀



索引的最左前缀和和B+Tree中的“最左前缀原理”有关，举例来说就是如果设置了组合索引<col1,col2,col3>那么以下3中情况可以使用索引：col1，<col1,col2>，<col1,col2,col3>，其它的列，比如<col2,col3>，<col1,col3>，col2，col3等等都是不能使用索引的。

根据最左前缀原则，我们一般把排序分组频率最高的列放在最左边，以此类推。

2、带索引的模糊查询优化

在上面已经提到，使用LIKE进行模糊查询的时候，'%aaa%'不会使用索引，也就是索引会失效。如果是这种情况，只能使用全文索引来进行优化（上文有讲到）。

3、为检索的条件构建全文索引，然后使用

SELECT \* FROM tablename MATCH(index\_colum) ANGAINST(‘word’);

4、使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

为什么mysql的索引使用B+树而不是B树呢？？

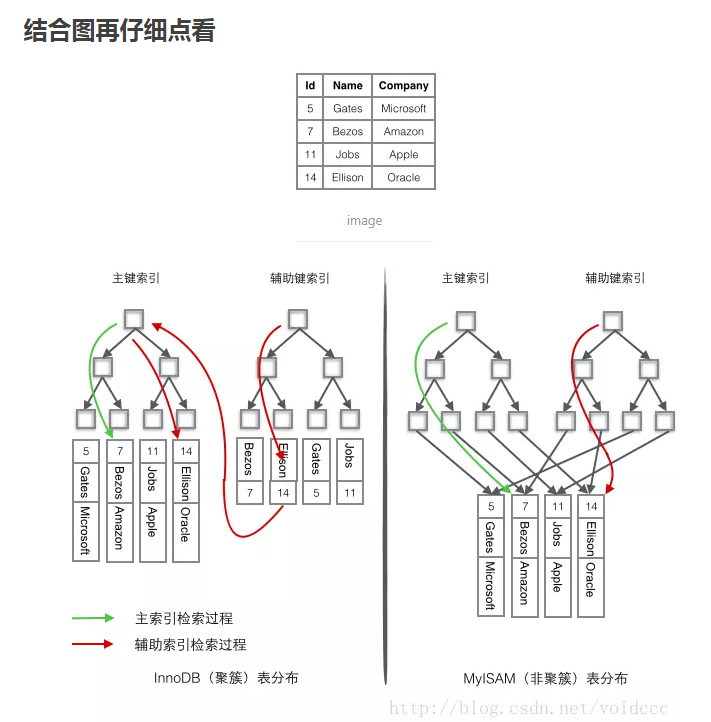
（1）B+树更适合外部存储(一般指磁盘存储),由于内节点(非叶子节点)不存储data，所以一个节点可以存储更多的内节点，每个节点能索引的范围更大更精确。也就是说使用B+树单次磁盘IO的信息量相比较B树更大，IO效率更高。

（2）mysql是关系型数据库，经常会按照区间来访问某个索引列，B+树的叶子节点间按顺序建立了链指针，加强了区间访问性，所以B+树对索引列上的区间范围查询很友好。而B树每个节点的key和data在一起，无法进行区间查找。

**聚簇索引与非聚簇索引（也叫二级索引）**

1. 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据
2. 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key\_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

澄清一个概念：innodb中，在聚簇索引之上创建的索引称之为辅助索引，辅助索引访问数据总是需要二次查找，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引，辅助索引叶子节点存储的不再是行的物理位置，而是主键值



1. InnoDB使用的是聚簇索引，将**主键组织到一棵B+树**中，而**行数据就储存在叶子节点**上，若使用"where id = 14"这样的条件查找主键，则**按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据**。
2. 若**对Name列进行条件搜索，则需要两个步骤**：**第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键**。第二步**使用主键在主索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据**。（**重点在于通过其他键需要建立辅助索引**）

MyISM使用的是非聚簇索引，**非聚簇索引的两棵B+树看上去没什么不同**，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于**索引树是独立的，通过辅助键检索无需访问主键的索引树**。

## **172.怎么验证 mysql 的索引是否满足需求？**

在select语句前加上explain就可以了：

EXPLAIN SELECT surname,first\_name form a,b WHERE a.id=b.id

查看索引是否生效

## **173.说一下数据库的事务隔离？**

• ① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。
• ② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。
• ③ Read committed (读已提交)：可避免脏读的发生。
• ④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

**2、出现的问题：**

当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：脏读、不可重复读、幻读。

**（1）脏读**

**1.脏读定义：**

1）说法1：指在一个事务处理过程里读取了另一个未提交的事务中的数据，读取数据不一致。

  2）说法2：指事务A对数据进行增删改操作，但未提交，另一事务B可以读取到未提交的数据。如果事务A这时候回滚了，则第二个事务B读取的即为脏数据。

**2.举例：**

当一个事务正在多次修改某个数据，而在这个事务中多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。

例如：用户A向用户B转账100元，对应SQL命令如下

　　update account set money=money+100 where name=’B’; (此时A通知B)

　　update account set money=money - 100 where name=’A’;

当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

**（2）不可重复读**

**1.不可重复读定义：**

1）说法1：是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

  2）说法2：一个事务A中发生了两次读操作，第一次读操作和第二次读操作之间，另一个事务B对数据进行了修改，这时两个事务读取的数据不一致。

**2.举例：**

例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

**3.不可重复读和脏读的区别：**

脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

  在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

**（3）虚读(幻读)**

**1.幻读定义：**

1）说法1：是事务非独立执行时发生的一种现象。

  2）说法2：第一个事务A对一定范围的数据进行批量修改，第二个事务B在这个范围增加一条数据，这时候第一个事务就会丢失对新增数据的修改。

**2.举例：**

例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

**3.幻读和不可重复读区别：**

都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

## **174.说一下 mysql 常用的引擎？**

MySQL常用的四种引擎的介绍

（1）：MyISAM存储引擎：不支持事务、也不支持外键，优势是访问速度快，对事务完整性没有 要求或者以select，insert为主的应用基本上可以用这个引擎来创建表

支持3种不同的存储格式，分别是：静态表；动态表；压缩表

静态表：表中的字段都是非变长字段，这样每个记录都是固定长度的，优点存储非常迅速，容易缓存，出现故障容易恢复；缺点是占用的空间通常比动态表多（因为存储时会按照列的宽度定义补足空格）ps：在取数据的时候，默认会把字段后面的空格去掉，如果不注意会把数据本身带的空格也会忽略。

动态表：记录不是固定长度的，这样存储的优点是占用的空间相对较少；缺点：频繁的更新、删除数据容易产生碎片，需要定期执行OPTIMIZE TABLE或者myisamchk-r命令来改善性能

压缩表：因为每个记录是被单独压缩的，所以只有非常小的访问开支

（2）InnoDB存储引擎\*

该存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。但是对比MyISAM引擎，写的处理效率会差一些，并且会占用更多的磁盘空间以保留数据和索引。

InnoDB存储引擎的特点：支持自动增长列，支持外键约束

(3)：MEMORY存储引擎

Memory存储引擎使用存在于内存中的内容来创建表。每个memory表只实际对应一个磁盘文件，格式是.frm。memory类型的表访问非常的快，因为它的数据是放在内存中的，并且默认使用HASH索引，但是一旦服务关闭，表中的数据就会丢失掉。

MEMORY存储引擎的表可以选择使用BTREE索引或者HASH索引，两种不同类型的索引有其不同的使用范围

Hash索引优点：

Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引。

Hash索引缺点： 那么不精确查找呢，也很明显，因为hash算法是基于等值计算的，所以对于“like”等范围查找hash索引无效，不支持；

Memory类型的存储引擎主要用于哪些内容变化不频繁的代码表，或者作为统计操作的中间结果表，便于高效地对中间结果进行分析并得到最终的统计结果，。对存储引擎为memory的表进行更新操作要谨慎，因为数据并没有实际写入到磁盘中，所以一定要对下次重新启动服务后如何获得这些修改后的数据有所考虑。

（4）MERGE存储引擎

Merge存储引擎是一组MyISAM表的组合，这些MyISAM表必须结构完全相同，merge表本身并没有数据，对merge类型的表可以进行查询，更新，删除操作，这些操作实际上是对内部的MyISAM表进行的。

## **175.说一下 mysql 的行锁和表锁？**

**多个事务操作同一行数据时，后来的事务处于阻塞等待状态。这样可以避免了脏读等数据一致性的问题。后来的事务可以操作其他行数据，解决了表锁高并发性能低的问题**。

**InnoDB的行锁是针对索引加的锁，不是针对记录加的锁。并且该索引不能失效，否则都会从行锁升级为表锁**。

**行锁**

行锁的劣势：开销大；加锁慢；会出现死锁

行锁的优势：锁的粒度小，发生锁冲突的概率低；处理并发的能力强

加锁的方式：自动加锁。对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁；对于普通SELECT语句，InnoDB不会加任何锁；当然我们也可以显示的加锁：

共享锁：select \* from tableName where … + lock in share more

排他锁：select \* from tableName where … + for update

InnoDB和MyISAM的最大不同点有两个：一，InnoDB支持事务(transaction)；二，默认采用行级锁。加锁可以保证事务的一致性，可谓是有人(锁)的地方，就有江湖(事务)；我们先简单了解一下事务知识。

**间隙锁**

当我们用范围条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做”间隙(GAP)”。InnoDB也会对这个”间隙”加锁，这种锁机制就是所谓的间隙锁(Next-Key锁)。

**危害：若执行的条件是范围过大，则InnoDB会将整个范围内所有的索引键值全部锁定，很容易对性能造成影响**。

**排他锁**

排他锁，也称写锁，独占锁，当前写操作没有完成前，它会阻断其他写锁和读锁。



**表锁**

表锁的优势：开销小；加锁快；无死锁

表锁的劣势：锁粒度大，发生锁冲突的概率高，并发处理能力低

加锁的方式：自动加锁。查询操作（SELECT），会自动给涉及的所有表加读锁，更新操作（UPDATE、DELETE、INSERT），会自动给涉及的表加写锁。也可以显示加锁：

共享读锁：lock table tableName read;

独占写锁：lock table tableName write;

批量解锁：unlock tables;

**共享读锁**

对MyISAM表的读操作（加读锁），不会阻塞其他进程对同一表的读操作，但会阻塞对同一表的写操作。只有当读锁释放后，才能执行其他进程的写操作。在锁释放前不能取其他表。

**独占写锁**

对MyISAM表的写操作（加写锁），会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其他进程的读写操作。在锁释放前不能写其他表。

总结：**表锁，读锁会阻塞写，不会阻塞读。而写锁则会把读写都阻塞**。

**什么场景下用表锁**

InnoDB默认采用行锁，在未使用索引字段查询时升级为表锁。MySQL这样设计并不是给你挖坑。它有自己的设计目的。

即便你在条件中使用了索引字段，MySQL会根据自身的执行计划，考虑是否使用索引(所以explain命令中会有possible\_key 和 key)。如果MySQL认为全表扫描效率更高，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。

第一种情况：**全表更新**。事务需要更新大部分或全部数据，且表又比较大。若使用行锁，会导致事务执行效率低，从而可能造成其他事务长时间锁等待和更多的锁冲突。

第二种情况：**多表查询**。事务涉及多个表，比较复杂的关联查询，很可能引起死锁，造成大量事务回滚。这种情况若能一次性锁定事务涉及的表，从而可以避免死锁、减少数据库因事务回滚带来的开销。

**页锁**

开销和加锁时间介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发处理能力一般。只需了解一下。

**总结**

1 InnoDB 支持表锁和行锁，使用索引作为检索条件修改数据时采用行锁，否则采用表锁。

2 InnoDB 自动给修改操作加锁，给查询操作不自动加锁

3 行锁可能因为未使用索引而升级为表锁，所以除了检查索引是否创建的同时，也需要通过explain执行计划查询索引是否被实际使用。

4 行锁相对于表锁来说，优势在于高并发场景下表现更突出，毕竟锁的粒度小。

5 当表的大部分数据需要被修改，或者是多表复杂关联查询时，建议使用表锁优于行锁。

6 为了保证数据的一致完整性，任何一个数据库都存在锁定机制。锁定机制的优劣直接影响到一个数据库的并发处理能力和性能

## **176.说一下乐观锁和悲观锁？**

悲观锁

总是假设最坏的情况，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塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁

总是假设最好的情况，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，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子：

假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ；而当前帐户余额字段（ balance ）为 $100 。当需要对账户信息表进行更新的时候，需要首先读取version字段。

操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50（ $100-$50 ）。

在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。

操作员 A 完成了修改工作，提交更新之前会先看数据库的版本和自己读取到的版本是否一致，一致的话，就会将数据版本号加1（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。

操作员 B 完成了操作，提交更新之前会先看数据库的版本和自己读取到的版本是否一致，但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，而自己读取到的版本号为1 ，不满足 “ 当前最后更新的version与操作员第一次读取的版本号相等 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

2. CAS算法

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2 循环时间长开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

## **177.mysql 问题排查都有哪些手段？**

## **178.如何做 mysql 的性能优化？**

**建索引的几大原则**

1.最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

3.尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录

4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

5.尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

**查询优化神器 – explain命令**

关于explain命令相信大家并不陌生，具体用法和字段含义可以参考官网[explain-output](http://dev.mysql.com/doc/refman/5.5/en/explain-output.html)，这里需要强调rows是核心指标，绝大部分rows小的语句执行一定很快（有例外，下面会讲到）。所以优化语句基本上都是在优化rows。

**慢查询优化基本步骤**

0.先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE

1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高

2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）

3.order by limit 形式的sql语句让排序的表优先查

4.了解业务方使用场景

5.加索引时参照建索引的几大原则

6.观察结果，不符合预期继续从0分析

## **十八、Redis**

## **179.redis 是什么？都有哪些使用场景？**

首先要说redis,应该先说一下nosql,

NoSQL(NoSQL = Not Only SQL )，意即“不仅仅是SQL”，

泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题，包括超大规模数据的存储。

（例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

Redis:REmote DIctionary Server(远程字典服务器)

是完全开源免费的，用C语言编写的，遵守BSD协议，

是一个高性能的(key/value)分布式内存数据库，基于内存运行

并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,

也被人们称为数据结构服务器。

优点：

Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储

Redis支持数据的备份，即master-slave模式的数据备份

应用场景：

内存存储和持久化：redis支持异步将内存中的数据写到硬盘上，同时不影响继续服务

取最新N个数据的操作，如：可以将最新的10条评论的ID放在Redis的List集合里面

模拟类似于HttpSession这种需要设定过期时间的功能

发布、订阅消息系统

定时器、计数器

## **180.redis 有哪些功能？**

五种用途

1.全页面缓存

整页缓存。如果你正在使用服务器端呈现的内容，则不需要为每个单独的请求重新渲染每个页面。使用如Redis这样的缓存，你可以缓存经常请求的内容，从而大大减少请求最多的页面的延迟，并且大多数框架针对Redis缓存页面都有hooks。

2.顺序排列

由于Redis在内存中Set数据结构可以非常快速和高效进行递增和递减，性能远远高于SQL查询。比较这与Redis的排序集相结合意味着你可以以毫秒为单位抓取列表中评分最高的项目，而且实现起来非常容易。

3.会话Session存储

我所见的Redis最常见的用途是会话存储。与其他会话存储（如Memcache）不同，Redis可以保留数据，以便在缓存停止的情况下，在重新启动时，所有数据仍然存在。即便不是需要严格持续的任务，此功能仍可以为你的用户省去大量的麻烦。没有人会乐于见到他们的会话被无缘无故随机删掉。

4.队列

使用Redis可以做的一个不太常见，但非常有用的事情是排队。无论是电子邮件队列还是其他应用程序使用的数据，你都可以在Redis中创建一个高效的队列。任何熟悉堆栈以及会push和pop项目的开发人员都可以轻松自然地使用此功能。

5.pub/sub

Redis在真实世界的最终用法即我将在这篇文章中提出的pub / sub。这是Redis内置的最强大的功能之一；得到的可能是无限的。你可以创建一个实时聊天系统，在社交网络上触发好友请求的通知等等。这个功能是Redis提供的最被低估的功能之一，但功能非常强大，而且使用简单。

## **181.redis 和 memecache 有什么区别？**

**[Redis和Memcache区别，优缺点对比](https://www.cnblogs.com/JavaBlackHole/p/7726195.html)**

1、 Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等。

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。

3、虚拟内存–Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘

4、过期策略–memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10

5、分布式–设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从

6、存储数据安全–memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）

7、灾难恢复–memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复

8、Redis支持数据的备份，即master-slave模式的数据备份。

**redis和memecache的不同在于[2]：**

1、存储方式：

memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小

redis有部份存在硬盘上，这样能保证数据的持久性，支持数据的持久化（笔者注：有快照和AOF日志两种持久化方式，在实际应用的时候，要特别注意配置文件快照参数，要不就很有可能服务器频繁满载做dump）。

2、数据支持类型：

redis在数据支持上要比memecache多的多。

3、使用底层模型不同：

新版本的redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

4、运行环境不同：

redis目前官方只支持LINUX 上去行，从而省去了对于其它系统的支持，这样的话可以更好的把精力用于本系统 环境上的优化，虽然后来微软有一个小组为其写了补丁。但是没有放到主干上

个人总结一下，有持久化需求或者对数据结构和处理有高级要求的应用，选择redis，其他简单的key/value存储，选择memcache。



## **182.redis 为什么是单线程的？**

redis是存储在内存上的，读写的话不会受到硬盘 I/O 速度的限制

2.数据结构简单，对数据操作也简单

3.多路IO复用模型，非阻塞IO**解释：多路：多网络连接   复用：复用同一个线程**

4. 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

5.使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

## **183.什么是缓存穿透？什么是雪崩？怎么解决？**

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决办法：

     1.布隆过滤

  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。

  补充:  Bloom filter

  适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集

  基本原理及要点：对于原理来说很简单，位数组+k个独立hash函数。将hash函数对应的值的位数组置1，查找时如果发现所有hash函数对应位都是1说明存在，很明显这个过程并不保证查找的结果是100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是counting Bloom filter，用一个counter数组代替位数组，就可以支持删除了。添加时增加计数器，删除时减少计数器。

     2. 缓存空对象. 将 null 变成一个值.

  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。

 缓存空对象会有两个问题：

1. 第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间 ( 如果是攻击，问题更严重 )，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
2. 第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

缓存雪崩：

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

1. 解决方法：在缓存的时候给过期时间加上一个**随机值**，这样就会大幅度的**减少缓存在同一时间过期**。

对于“Redis挂掉了，请求全部走数据库”这种情况，我们可以有以下的思路：

1. 事发前：实现Redis的**高可用**(主从架构+Sentinel 或者Redis Cluster)，尽量避免Redis挂掉这种情况发生。
2. 事发中：万一Redis真的挂了，我们可以设置**本地缓存(ehcache)+限流(hystrix)**，尽量避免我们的数据库被干掉(起码能保证我们的服务还是能正常工作的)
3. 事发后：redis持久化，重启后自动从磁盘上加载数据，**快速恢复缓存数据**。

## **184.redis 支持的数据类型有哪些？**

redis支持的五种数据类型：

1、string（字符串）

2、hash（哈希）

　　Redis hash 是一个键值(key=>value)对集合。

　　Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

3、list（列表）

　　Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

4、set（集合）

　　Redis的Set是string类型的无序集合。

5、zset(sorted set：有序集合)

　　Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

　　不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

　　zset的成员是唯一的,但分数(score)却可以重复。

## **185.redis 支持的 java 客户端都有哪些？**

Redisson,Jedis，lettuce等等。

## **186.jedis 和 redisson 有哪些区别？**

Redis的Java客户端很多，官方推荐的有三种：Jedis、Redisson和lettuce。

在这里对Jedis和Redisson进行对比介绍

Jedis：

轻量，简洁，便于集成和改造

支持连接池

支持pipelining、事务、LUA Scripting、Redis Sentinel、Redis Cluster

不支持读写分离，需要自己实现

文档差（真的很差，几乎没有……）

Redisson：

基于Netty实现，采用非阻塞IO，性能高

支持异步请求

支持连接池

支持pipelining、LUA Scripting、Redis Sentinel、Redis Cluster

不支持事务，官方建议以LUA Scripting代替事务

支持在Redis Cluster架构下使用pipelining

支持读写分离，支持读负载均衡，在主从复制和Redis Cluster架构下都可以使用

内建Tomcat Session Manager，为Tomcat 6/7/8提供了会话共享功能

可以与Spring Session集成，实现基于Redis的会话共享

文档较丰富，有中文文档

对于Jedis和Redisson的选择，同样应遵循前述的原理，尽管Jedis比起Redisson有各种各样的不足，但也应该在需要使用Redisson的高级特性时再选用Redisson，避免造成不必要的程序复杂度提升。

## **187.怎么保证缓存和数据库数据的一致性？**

最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。

1. 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
2. 更新的时候，**先更新数据库，然后再删除缓存**。

最初级的缓存不一致问题及解决方案

问题：先修改数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。

解决思路：先删除缓存，再修改数据库。如果数据库修改失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中。

**如果一个内存队列中可能积压的更新操作特别多**，那么你就要**加机器**，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。

## **188.redis 持久化有几种方式？**

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

**二者优缺点**

**RDB存在哪些优势呢？**

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

**AOF的优势有哪些呢？**

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

**常用配置**

**RDB持久化配置**

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10            #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

**AOF持久化配置**

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always     #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no          #从不同步。高效但是数据不会被持久化。

## **189.redis 怎么实现分布式锁？**

**可靠性**

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了

redis提供了Redisson来实现分布式锁

## **190.redis 分布式锁有什么缺陷？**

**Redis分布式锁的缺点**

其实上面那种方案最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题， **导致各种脏数据的产生**。

所以这个就是redis cluster，或者是redis master-slave架构的 **主从异步复制**导致的redis分布式锁的最大缺陷：在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。

## **191.redis 如何做内存优化？**

一.redisObject对象

二.缩减键值对象

三.共享对象池

四.字符串优化

五.编码优化

六.控制key的数量

<https://www.cnblogs.com/jandison/p/6902396.html>

## **192.redis 淘汰策略有哪些？**

redis内存数据数据集大小升到一定大的时候，就会实行数据淘汰策略（回收策略）。

1，volatile-lru：从已设置过期时间的哈希表(server.db[i].expires)中随机挑选多个key,然后在选到的key中用lru算法淘汰最近最少使用的数据

2，allkey-lru：从所有key的哈希表（server.db[i].dict）中随机挑选多个key,然后再选到的key中利用lru算法淘汰最近最少使用的数据

3，volatile-ttl：从已设置过期时间的哈希表（server.db[i].expires)中随机挑选多个key,然后在选到的key中选择过期时间最小的数据淘汰掉。

4，volatile-random：从已设置过期时间的哈希表（server.db[i].expires）中随机挑选key淘汰掉。

5，allkey-random：从所有的key的哈希表（server.db[i].dict）中随机挑数据淘汰

6，no-eviction（驱逐）：内存达到上限，不淘汰数据。

redis确认驱逐某个键值对后，会删除这个数据，并将这个数据变更消息发布到本地（AOF持久化）和从机（主从连接）。

**LRU数据淘汰机制**是这样的：在数据集中随机挑选几个键值对，去除其中最近最少使用的键值对淘汰。所以Redis并不是保证取得所有数据集中最少最少使用的键值对，而只是在随机挑选的几个键值对中。

**TTL数据淘汰机制**：从国企时间redisDB.expires表中随机挑选几个键值对，取出其中最快过期的键值对淘汰。所以Redis并不保证取得所有过期时间表中最快过期的键值对，而是随机挑选的几个键值对中。

无论是什么机制，都是从所有的键值对中挑选合适的淘汰。

## **193.redis 常见的性能问题有哪些？该如何解决？**

1.master写内存快照，seve命令调度rdbsave函数，会阻塞主线程的工程，当快照比较大的时候对性能的影响是非常大的，会间断性暂停服务 。所以master最好不要写内存快照。

2.master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响master重启时的恢复速度。master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化，如果数据比较关键，某个slave开启AOF备份数据，策略每秒为同步一次。

3.master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂的服务暂停现象。

4.redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，slave和master最好在同一个局域网内。

# **十九、JVM**

## **194.说一下 jvm 的主要组成部分？及其作用？**

JVM内存区域分为五个部分，分别是堆，方法区，虚拟机栈，本地方法栈，程序计数器。

1.堆。 堆是Java对象的存储区域，任何用new字段分配的Java对象实例和数组，都被分配在堆上，Java堆可使用-Xms -Xmx进行内存控制，值得一提的是从JDK1.7版本之后，运行时常量池从方法区移到了堆上。

2.方法区。它用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据，方法区在JDK1.7版本及以前被称为永久代，从JDK1.8永久代被移除。

3.虚拟机栈。虚拟机栈中执行每个方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

4.本地方法栈。与虚拟机栈发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

5.程序计数器。指示Java虚拟机下一条需要执行的字节码指令。

以上五个区域是Java虚拟机内存划分情况，其中方法区和堆被JVM中多个线程共享，比如类的静态常量就被存放在方法区，供类对象之间共享，虚拟机栈，本地方法栈，pc寄存器是每个线程独立拥有的，不会与其他线程共享。

所以Java在通过new创建一个类对象实例的时候，一方面会在虚拟机栈中创建一个该对象的引用，另一方面会在堆上创建类对象的实例，然后将对象引用指向该对象的实例。对象引用存放在每一个方法对应的栈帧中。

## **195.说一下 jvm 运行时数据区？**

1. 程序计数器

    解释：【指向当前线程所执行的字节码的行号】，其实就是一小块内存，记录着当前程序运行到哪了字节码解释器的工作就是通过改变这个计数器的值来选取下一条需要执行的字节码指令（字节码指令是什么？）。分支，循环，跳转，异常处理，线程回复等都需要依赖这个计数器来完成。

    由于Java的多线程是通过线程轮流切换完成的，一个线程没有执行完时就需要一个东西记录它执行到哪了，下次抢占到了CPU资源时再从这开始，这个东西就是程序计数器，正是因为这样，所以它也是“线程私有”的内存。

    如果一个线程执行一个主要方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址;如果正在执行的是一个本地方法，这个计数器的值则为空，此内存区域是唯一一个在Java的虚拟机规范中没有规定任何OutOfMemoryError异常情况的区域。

2. Java虚拟机栈

    与程序计数器一样，Java的虚拟机栈也是线程私有的，虚拟机栈描述的是Java的方法执行的内存模型，方法每个执行在同时的创建³³都会一个栈桢用于存储局部变量表，操作数栈，动态链接，方法出口等信息，下图为栈桢结构

    通常所说的JVM里的堆和栈里这个栈就是Java的虚拟机栈，或者说是Java的虚拟机栈中的局部变量表部分。

    局部变量表存放了编译期可知的各种基本数据类型，对象引用（仅限局部变量的，不包含成员变量的）。其中每个局部变量空间（Slot）有32位，所以long和double类型的数据会占用两个局部变量空间，其他类型包括对象引用占用一个。对象引用调用的是存在堆中的对象，这个引用可以是对象的起始地址或者是指向对象的句柄。局部变量表所需的内存在编译期就已经确定了也就是进入这个方法时就已经确定了，运行期间不会更改。

    操作数栈则存储方法内一些进行了运算操作后的结果。

    动态链接，在方法内调用接口，通过字面量链接到具体的实现类，实现Java的动态特性。

    方法出口（返回地址），return或者发生Exception等。

    如果方法methodOne方法调用了methodTwo，那么methodOne就会先入栈创建一个栈桢，接着methodTwo再入栈成为栈顶（假设没有其他的方法执行），methodTwo执行完先出栈，接着methodOne执行完出栈。

    在使用递归的情况下，如果线程请求的栈的深度超过虚拟机所允许栈的深度就会抛出StackOverflowError；但是大部分虚拟机栈的深度都可以动态扩展，HotSpot中使用XSS可以设置栈的深度，如果扩展时无法请求到足够的内存就会抛出OutOfMemoryError。

3. 本地方法栈

    本地方法栈和虚拟机栈相似，区别就是虚拟机为虚拟机栈执行Java服务（字节码服务），而本地方法栈为虚拟机使用到的Native方法服务。本地方法栈中使用的语言，使用方式，数据结构没有强制要求。

4. Java堆

    堆是JVM里最大的一块内存区域，被所有线程共享，在虚拟机启动时创建，此区域的目的就是存放对象实例和数组，几乎所有的对象实例都在这分配（随着JIT的发展已经不是那么绝对了）的.java堆是垃圾收集管理的主要区域，由于现在收集器基本都采用分代收集方法，所以Java的堆中还可以分为新生代，老年代，永久代.1.8之后取消了永久代;其中新生代又划分为Eden空间，From Survivor空间，To Survivor空间。无论怎么划分都是为了更好的回收，分配，利用内存。下图为1.8后的内存模型

    根据的Java虚拟机规范，Java的堆可以处于物理不连续的空间中，只要逻辑连续即可。在实现时，既可以实现成固定大小的也可以是可扩展的（通过-Xmx和-Xms控制），如果堆中没有足够的内存完成实例分配，并且堆也无法得到扩展时，将会抛出的OutOfMemoryError异常。

5. 方法区

    方法区也是一个线程共享的区域，存储已被虚拟机加载的类信息，常量（final），静态变量（static），JIT（即时编译器）编译后的代码等数据。

Java虚拟机规范把方法区描述为堆的一个逻辑部分，其实堆和方法区可以看成数据部分；虚拟机栈和程序计数器可以看成指令部分；方法区存储一些不会变更的数据，之前热点上使用GC分代收集管理方法区，所以方法区也被称为永久代（本质上两者不等价），但是现在已经使用Native Memory来代替永久代了。

    虚拟机对方法区规范非常宽松，除了和Java的堆一样不需要连续的内存和可以选择固定大小意外，还可以选择不实现垃圾回收。垃圾回收行为在这个区域比较少见但还是有必要的，主要是针对常量池回收和类型的卸载。

5.1运行时常量池

    运行时常量池是方法区的一部分，用于存放编译期生成的各种字面量和符号引用，运行时常量池相对于类常量池另外一个特性就是具备动态性，运行期间可能将新的常量放入池中。

## **196.说一下堆栈的区别？**

  栈内存:栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），for循环内部定义的也是局部变量，是先加载函数才能进行局部变量的定义，所以方法先进栈，然后再定义变量，变量有自己的作用域，一旦离开作用域，变量就会被释放。栈内存的更新速度很快，因为局部变量的生命周期都很短。

  堆内存:存储的是数组和对象（其实数组就是对象），凡是new建立的都是在堆中，堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性），如果一个数据消失，这个实体也没有消失，还可以用，所以堆是不会随时释放的，但是栈不一样，栈里存放的都是单个变量，变量被释放了，那就没有了。堆里的实体虽然不会被释放，但是会被当成垃圾，Java有垃圾回收机制不定时的收取。

堆与栈的区别很明显：

            1.栈内存存储的是局部变量而堆内存存储的是实体；

            2.栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短；

            3.栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。

## **197.队列和栈是什么？有什么区别？**

队列（Queue）：是限定只能在表的一端进行插入和另一端删除操作的线性表

栈（Stack）：是限定只能在表的一端进行插入和删除操作的线性表

1：队列和栈是两种不同的数据结构。它们有以下区别：

（1）操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。

（2）可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。

（3）操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

## **198.什么是双亲委派模型？**

**当需要加载一个类的时候，子类加载器并不会马上去加载，而是依次去请求父类加载器加载，一直往上请求到最高类加载器：启动类加载器。当启动类加载器加载不了的时候，依次往下让子类加载器进行加载。当达到最底下的时候，如果还是加载不到该类，就会出现ClassNotFound的情况。**

**好处：保证了程序的安全性。例子：比如我们重新写了一个String类，加载的时候并不会去加载到我们自己写的String类，因为当请求上到最高层的时候，启动类加载器发现自己能够加载String类，因此就不会加载到我们自己写的String类了。**

双亲委派模型工作工程：

1.当Application ClassLoader 收到一个类加载请求时，他首先不会自己去尝试加载这个类，而是将这个请求委派给父类加载器Extension ClassLoader去完成。

2.当Extension ClassLoader收到一个类加载请求时，他首先也不会自己去尝试加载这个类，而是将请求委派给父类加载器Bootstrap ClassLoader去完成。

3.如果Bootstrap ClassLoader加载失败(在<JAVA\_HOME>\lib中未找到所需类)，就会让Extension ClassLoader尝试加载。

4.如果Extension ClassLoader也加载失败，就会使用Application ClassLoader加载。

5.如果Application ClassLoader也加载失败，就会使用自定义加载器去尝试加载。

6.如果均加载失败，就会抛出ClassNotFoundException异常。

## **199.说一下类加载的执行过程？**

五步：

**1.加载：将java源代码编译后的.class字节码文件以二进制流的方式加载进内存**

**连接**

**2.验证：验证加载进来的二进制流是否符合虚拟机的规范，不会危害的虚拟机自身的安全**

**3.准备：给类变量（静态变量）赋予初始值，基本数据/引用类型数据**

**4.解析：将字符串引用转换为直接引用**

**5.初始化：变量赋予初始值、执行静态语句块、执行构造函数等等。**

## **200.怎么判断对象是否可以被回收？**

**简单的说就是给对象添加一个计数器，每当有一个地方引用它时，计数器就加1；当引用失效，计数器就减1；任何时刻计数器为0的对象，就是不可能再使用的。**

**优点：效率高，实现简单**

**缺点：无法解决对象之间循环引用的问题**

**（2）可达性算法**

**算法的基本思想是通过一系列的成为“GC Roots”的对象作为起点，从这些起点开始向下搜索，搜索的路径就成为引用链（Reference Chain），当一个对象到GC Roots没有任何的引用链相连的话，也就是该对象不可达，则证明该对象是不可用的**

在可达性的算法中，至少要被标记两次，才会真正的宣告一个对象死亡：如果对象那在进行可达性分析发现没有GC Roots相连的引用链，那么将会被第一次标记并且进行一次筛选，筛选的条件是此对象那个是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

        如果这个对象被判定位有必要执行finalize()方法，那么这个对象将会被放置在一个叫做F-Queue的队列中，并在稍后有一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，并不会承诺会等待它允许结束，这样的原因是，如果一个对象的finalize()方法执行缓慢，或者发生死循环，可能会到导致F-Queue队列中的其他对象永远处于等待，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，如果对象要在finalize()方法中拯救自己--只需要重新与引用链上的任何一个对象建立关联即可（这种自救的机会只能有一次），譬如把自己的this关键字赋值给某各类的变量或者类的成员变量即可，那在第二次标记时，就会被移除回收集合。

        如果对象在这个时候还没有逃脱，基本上就要被回收了。

## **201.java 中都有哪些引用类型？**

所以在 JDK.1.2 之后，Java 对引用的概念进行了扩充，将引用分为了：强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4 种，这 4 种引用的强度依次减弱。

**1.强引用**

Java中默认声明的就是强引用，比如：

Object obj = new Object(); //只要obj还指向Object对象，Object对象就不会被回收
obj = null; //手动置null

只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

**2.软引用**

软引用是用来描述一些非必需但仍有用的对象。**在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常**。这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。

在 JDK1.2 之后，用java.lang.ref.SoftReference类来表示软引用。

**3.弱引用**

弱引用的引用强度比软引用要更弱一些，**无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收**。在 JDK1.2 之后，用 java.lang.ref.WeakReference 来表示弱引用。

**4.虚引用**

虚引用是最弱的一种引用关系，如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收，在 JDK1.2 之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个null，也就是说将永远无法通过虚引用来获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

## **202.说一下 jvm 有哪些垃圾回收算法？**

**1.标记-清除算法**

分为两个阶段：

1. 扫描标记所有需要回收的对象
2. 清楚所有被标记的对象

优缺点：

1. **优点：** 实现简单效率高。
2. **缺点：** 会带来大量不连续的内存空间碎片。

2.**复制算法**

将内存分成两个大小完成相同的两块，当执行清理时，将存活对象复制到保留内存的快中，在清空当前的这一块。

优缺点：

**优点：** 比标记-清除算法效率高。

**缺点：** 可用内存空间变为了原来的一半。

3.**标记-压缩算法**

标记-压缩，又称为标记-整理算法，标记过程和标记-清除算法一样，在清除前将存活对象全部移到一端，然后清除掉边界外的内存。

优缺点：

1. **优点：** 解决了标记-清除、复制算法两种方法的主要问题。
2. **缺点：** 效率不高。

4.**分代收集算法**

分代收集算法其实是根据各个年代的特点，分别选用上述三种不同的垃圾收集算法。

**新生代：** 经常有大量对象死亡，可以选择复制算法。

**老年代：** 对象存活几率较高，可以选择标记-清除或标记-压缩算法。

## **203.说一下 jvm 有哪些垃圾回收器？**

**1.Serial**

串行收集器，采用复制算法， HotSpot运行在Client模式下的默认新生代收集器。

特点：

1. 单线程， 一但进行垃圾收集必须暂停其他所有线程（Stop The World）
2. 单线程垃圾收集简单高效。

**2.ParNew**

并行收集器, 采用复制算法, Serial的多线程版本, Server模式下的首选新生代收集器

特点:

1. 多线程, 多核CPU下对系统资源的利用率较高, 其余属性与Serial相同.
2. 除了Serial外唯一能和CMS收集器配合工作的收集器.

3.**Parallel Scavenge**

并行多线程新生代收集器, 采用复制算法.

特点:

1. 关注垃圾收集器的吞吐量,
2. 实现吞吐量可控, 高效地利用CPU资源.
3. 无法与CMS收集器配合工作

4.**Serial Old**

Serial的老年代版本, 单线程收集器, 采用标记-整理算法, 在Client模式下使用.

特性:

1. 在JDK1.5以及之前的版本中与Parallel Scavenge配合使用
2. 可以作为CMS的后备预案, 在并发收集发生Concurrent Mode Failure时使用.

**5.Parallel Old**

Parallel Scavenge的年老代版本, 多线程上映标记-整理算法, jdk1.6中开始提供.

特性:

1. Parallel Scavenge+Serial Old地位尴尬. Parallel Scavenge + Parallel Old可以再之中吞吐量和CPU资源敏感的场合搭配使用.

**6.CMS（Concurrent Mark Sweep）**

目前主流的互联网企业选用的垃圾收集器. 使用标记-清除算法的年老代并发收集器

特性:

1. 低停顿
2. 无法处理浮动垃圾
3. 标记-清理算法造成碎片

收集过程:

1. 初始标记

需要暂停其他线程, 标记一下GC Roots能够直接关联到的对象, 速度很快.

1. 并发标记

并发执行, 进行GC Roots Tracing可达性分析, 对非直接关联对象进行标记, 耗时长

1. 重新标记

暂停其他线程, 并发修正并发标记产生变动的对象的记录.

1. 并发清理

并发运行, 清理标记的对象.

整个过程耗时最长的并发标记和并发清理都是并发执行的, 因此可以认为CMS是与用户线程并发执行的.

特点:

1. CMS启动回收线程数=(CPU数目 + 3) /4 对CPU资源敏感.
2. 无法处理浮动垃圾, 即在并发清理阶段产生的垃圾无法清除, 需要等到下一次GC. 需要留给用户线程内存空间, 预留的内存不够产生Concurrent Mode Failure 导致Full GC, JVM启动后备预案，临时启用Serial Old进行老年代垃圾收集.

**7.G1收集器**

面向服务端应用的收集器,主要目标用于配备多颗 CPU 的服务器治理大内存.

整个Java堆划分成为多个大小相等的区域Region, 新生代和年老代是一部分Region的集合.

根据每个Region里面垃圾堆积的价值大小, 维护一个优先列表, 每次根据列表回收价值最大的Region, 保证了回收的效率.

建立Remembered Set 避免不同Region和区域间的对象引用. 每个Region都有一个Remembered Set, 当对引用进行操作时会维护被引用对象所属的Region的Remembered Set.

特点:

1. 并行与并发: 充分利用多CPU多核环境, 缩短Stop the World的时间.
2. 分代收集: 能够采用不同的方式去处理新创建的对象和已经存活了一段时间的多谢和熬过了多次GC的多谢.
3. 空间整合: 整体来看是给予标记-整理算法, 局部Region来看是基于复制算法. 因此运行期间不会产生碎片.
4. 可以预测的停顿. 低停顿是G1和CMS共同的关注点, G1还能建立可预测的停顿时间模型, 能够制定一个长度的时间片, 是垃圾收集不超过该时间.

收集过程:

1. 初始标记

需要暂停其他线程, 标记一下GC Roots能够直接关联到的对象, 速度很快.

1. 并发标记

并发执行, 进行GC Roots Tracing可达性分析, 对非直接关联对象进行标记, 耗时长

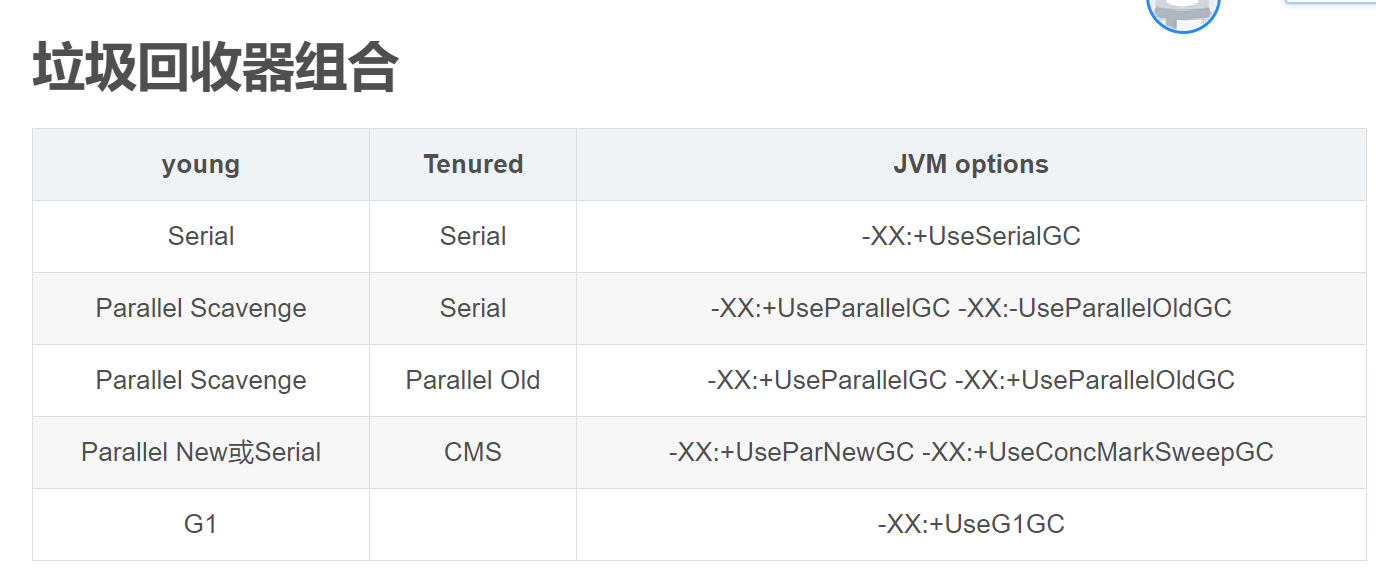
1. 最终标记

需要暂停其他线程, 修正并发标记阶段因用户线程继续运行而导致标记发生变化的那部分对象的标记记录。

1. 筛选回收

对每个Region的回收价值和成本进行排序, 根据用户期望的GC停顿时间进行回收.

详细垃圾回收器如下图：



## **204.详细介绍一下 CMS 垃圾回收器？**

CMS垃圾回收特点

cms只会回收老年代和永久带（1.8开始为元数据区，需要设置CMSClassUnloadingEnabled），不会收集年轻带；

cms是一种预处理垃圾回收器，它不能等到old内存用尽时回收，需要在内存用尽前，完成回收操作，否则会导致并发回收失败；所以cms垃圾回收器开始执行回收操作，有一个触发阈值，默认是老年代或永久带达到92%；

CMS 处理过程有七个步骤：

1. 初始标记(CMS-initial-mark) ,会导致swt；

2. 并发标记(CMS-concurrent-mark)，与用户线程同时运行；

3. 预清理（CMS-concurrent-preclean），与用户线程同时运行；

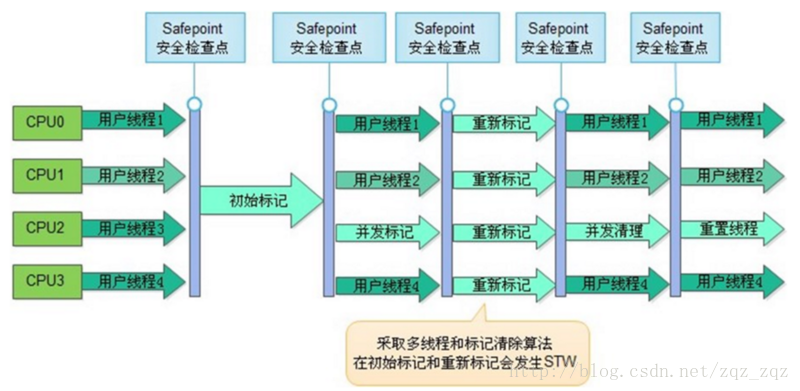
4. 可被终止的预清理（CMS-concurrent-abortable-preclean） 与用户线程同时运行；

5. 重新标记(CMS-remark) ，会导致swt；

6. 并发清除(CMS-concurrent-sweep)，与用户线程同时运行；

7. 并发重置状态等待下次CMS的触发(CMS-concurrent-reset)，与用户线程同时运行；

cms运行流程图如下所示：



## 205.新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别？

**新生代回收器**：Serial、ParNew、Parallel Scavenge

**老年代回收器**：Serial Old、Parallel Old、CMS

**整堆回收器**：G1

## **206.简述分代垃圾回收器是怎么工作的？**

**第一步：标记**

这个过程的第一步叫做标记。这一阶段，垃圾回收器识别哪些内存正在使用，哪些内存未被使用。

**第二步：清除**

这一步会删掉标记出的未引用对象。

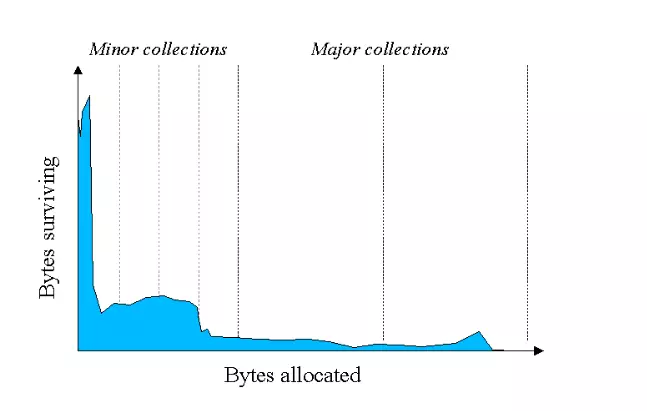
**压缩**

为了提升性能，删除了未引用对象后，还可以将剩下的已引用对象放在一起（压缩），这样就能更简单快捷地分配新对象了。

**为什么使用分代垃圾回收?**

如前所述，标记并整理所有JVM里的对象，效率是很低的。随着越来越多的对象被分配，对象列表不断增长，导致垃圾回收时间越来越长。然而，根据应用程序的经验分析，大多数的对象都是短命的。

下面是这类数据的一个例子。Y轴表示分配的字节数，X轴表示随着时间分配的字节数。



正如你所看到的，随着时间的推移，仍然存活的对象越来越少。事实上，大多数对象的生命周期都很短，如图左侧较高的值所示。

## **207.说一下 jvm 调优的工具？**

1、jps:JVM Process Status Tool，显示指定系统内所有的HotSpot虚拟机进程。

2、 jstat（JVM statistics Monitoring）是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

3、jmap（JVM Memory Map）命令用于生成heap dump文件，如果不使用这个命令，还阔以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。 jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。

4、jinfo：实时地查看和调整虚拟机各项参数

5、jstack：堆栈跟踪工具，jstack用于生成虚拟机当前时刻的线程快照。

6、jhat：分析内存转储快照，不推荐使用，而且慢

7、JConsole：JMX的可视化管理工具

8、VisualVM：多合一故障管理工具

这个工具也很牛bility。它同jconsole都是一个基于图形化界面的、可以查看本地及远程的JAVA GUI监控工具，Jvisualvm同jconsole的使用方式一样，直接在命令行打入jvisualvm即可启动，jvisualvm界面更美观一些，数据更实时

## **208.常用的 jvm 调优的参数都有哪些？**

jvm配置

XX比X的稳定性更差，并且版本更新不会进行通知和说明。

1、-Xms

    s为strating，表示堆内存起始大小

2、-Xmx

    x为max，表示最大的堆内存

（一般来说-Xms和-Xmx的设置为相同大小，因为当heap自动扩容时，会发生内存抖动，影响程序的稳定性）

3、-Xmn

    n为new，表示新生代大小

（-Xss：规定了每个线程虚拟机栈（堆栈）的大小）

4、-XX:SurvivorRator=8

    表示堆内存中新生代、老年代和永久代的比为8:1:1

5、-XX:PretenureSizeThreshold=3145728

    表示当创建（new）的对象大于3M的时候直接进入老年代

6、-XX:MaxTenuringThreshold=15

    表示当对象的存活的年龄（minor gc一次加1）大于多少时，进入老年代

7、-XX:-DisableExplicirGC

    表示是否（+表示是，-表示否）打开GC日志

## 209. 什么情况下会发生内存溢出？

栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常，方法递归调用产生这种结果。

如果Java虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么Java虚拟机将抛出一个OutOfMemory 异常。(线程启动过多)

参数 -Xss 去调整JVM栈的大小

## **210. JVM中一次完整的GC流程是怎样的？对象如何晋升到老年代？**

**Java堆 = 老年代 + 新生代**

**新生代 = Eden + S0 + S1**

**当 Eden 区的空间满了， Java虚拟机会触发一次** **Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor区。**

**大对象（需要大量连续内存空间的Java对象，如那种很长的字符串）直接进入老年态；**

**如果对象在Eden出生，并经过第一次Minor GC后仍然存活，并且被Survivor容纳的话，年龄设为1，每熬过一次Minor GC，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。即长期存活的对象进入老年态。**

**老年代满了而无法容纳更多的对象，Minor GC 之后通常就会进行Full GC，Full GC 清理整个内存堆 – 包括年轻代和年老代。**

**Major GC 发生在老年代的GC，清理老年区，经常会伴随至少一次Minor GC，比Minor GC慢10倍以上。**