王道面试宝典

第一章 JAVA基础

1 JDK、JRE、JVM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。
> - JVM：Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。
>
> 具体来说JDK其实包含了JRE，同时还包含了编译java源码的编译器javac，还包含了很多java调试和分析的工具。简单来说，如果你需要运行java程序，那么你只需要安装JRE就可以了，如果你需要编写并调试java程序，那么需要安装JDK。而Java程序都是在经过编译之后交给JVM去运行的。

2 == 和equals的区别是什么?

> - ==
>
> 在Java中，==对于基本类型和引用类型的效果是不一样的
>
> - 基本类型：比较值是否相同
> - 引用类型：比较的是对象的内存地址是否相同
>
> 因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。
>
> - equals
>
> `equals()` 方法存在两种使用情况：
>
> - **类没有覆盖 `equals()` 方法**：通过`equals()`比较该类的两个对象时，等价于通过“==”比较这两个对象，使用的默认是 `Object`类`equals()`方法。
> - **类覆盖了 `equals()` 方法**：一般我们都覆盖 `equals()`方法来比较两个对象中的属性是否相等；若它们的属性相等，则返回 true(即，认为这两个对象相等)。

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()不一定是true。
>
> 代码示例：
>
> ```
> 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));
> ```
>
> 执行结果：
>
> ```
> str1：1179395 | str2：1179395
> false
> ```
>
> 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 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。

5 String str="i" 与 String str = new String("i") 一样吗?

> 不一样，因为内存分配的方式不一样。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。

6 普通类和抽象类有什么区别?

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

7 抽象类和接口有什么区别?

> - 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。

8 简述一下类加载的过程？Java的类加载器有哪些？双亲委派模型是如何实现的？

类加载是指类从被加载到虚拟机内存中开始，到被卸载出内存位置，他的整个生命周期包括

加载 —> 验证 —> 准备 —> 解析 —> 初始化 —> 使用 —> 卸载 7个部分

1. 加载

加载只是类加载的一个阶段，大家不要误以为是整个类加载的过程

1.1 JDK在执行程序运行命令时会去JRE目录中找到jvm.dll， 并初始化JVM，这时会产生一个Bootstrap Loader（启动类加载器）

1.2 Bootstrap Loader 自动加载 Extended Loader（标准扩展类加载器）

1.3 Bootstrap Loader 自动加载 AppClass Loader（系统类加载器）

1.4 最后由 AppClass Loader 加载 我们指定（想要运行）的 java 类

双亲委派模型

实现双亲委派模型的代码都在`java.lang.ClassLoader.loadClass()`中，过程如下：

1. 先检查此类是否被加载过，如果没有加载则调用父加载器的`loadClass()`方法
2. 若父加载器为空，则默认使用启动类加载器作为父加载器
3. 若父类加载失败，会抛出一个异常，然后再调用自己的`finalClass()`方法来进行加载

结合起来我们可以这么理解：

1. 首先要启动启动类加载器，这个时候会调用启动类加载器的父加载器，但是由于启动类加载器是所有类的父加载器，所以其父加载器为空（类似于Object类是所有类的父类），然后它就会调用自己的`findClass()`方法来启动加载
2. 标准扩展类加载器启动时就会借助其父类-启动类加载器 作为父加载器来启动
3. 系统类加载器启动时就会借助其父类-标准扩展类加载器 作为父加载器来启动
4. 最后我们编写的普通类就会借助其父类-系统类加载器 作为父加载器来启动

总而言之，我们去加载我们写的普通的类的时候，会委派给他的父类加载器（AppClass Loader）来加载，一层一层向上委派，最后都会让启动类加载器来加载

2. 验证

验证主要分为以下几个步骤：文件格式验证—> 元数据验证 —> 字节码验证—> 符号引用验证

文件格式验证：主要是检查字节码的字节流是否符合class文件格式的规范，验证该文件能否被当前的jvm所处理

元数据验证：对字节码描述的信息进行语义分析，保证其描述的内容符合java语言的语法规范，能被java虚拟机识别

字节码验证：该部分最为复杂，对方法体内的内容进行验证，保证代码在运行时不会做出什么危害虚拟机安全的事件

符号引用验证：来验证一些符号引用的真实性与可行性

3. 准备

准备阶段会为类里面的静态成员变量分配内存并设置类的初始值

> 4. 解析
>
> 解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程
>
> 5. 初始化
>
> 类的初始化是类加载过程的最后一个步骤，这个才是执行类中定义的Java程序代码

9 intern方法相关

> intern是String类的一个成员方法，以Java8版本的intern方法实现为标准

它是一个native方法，该方法首先会从字符串常量池中检测该对象是否已存在：

- 如果存在就返回字符串常量池中，该对象的引用
- 如果不存在就将存在于堆上的字符串对象的引用存入常量池（注意不会在常量池中创建新对象）

例如：

```
String s1 = "Hello";
String s2 = new String("Hello");
String s3 = new String("World");
String s4 = s2 + s3;
System.out.println(s1 == s2.intern());
System.out.println(s2 == s2.intern());
System.out.println(s3 == s3.intern());
System.out.println(s4 == s4.intern());
```

程序运行的结果应该是

> true
> false
> false
> true

经典面试题（来自《深入理解java虚拟机 第二版》）：

```
String str1 = new StringBuilder("计算机").append("软件").toString();
System.out.println(str1.intern() == str1);

String str2 = new StringBuilder("ja").append("va").toString();
System.out.println(str2.intern() == str2);
```

> 第一个结果是true这很容易理解，因为堆上的“计算机软件”经过intern将引用存入常量池，所以str1.intern()和str1实际上是同一个对象
>
> 第二个结果却是false，这其实也不难理解。根源在于str2.intern()之前常量池中就已经存在“java”对象了。启动任何一个类的main方法时，都会类加载一些启动类，sun.misc包下的Version类中就有“java”字符串常量加入常量池！

10. StringBuffer和StringBuilder的区别

它们的共同点在于都是可变字符串，都继承了AbstractStringBuilder。StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象，不会产生效率问题和空间浪费问题

> StringBuffer是线程安全的，StringBuilder是线程不安全的
>
> - StringBuilder的效率会比StringBuffer效率更高，单线程的程序推荐使用StringBuilder
> - 在多线程的程序中，应该优先考虑使用StringBuffer，安全性要更重要

> - 它们的效率都比String高很多

第二章 JAVA高级

集合

1 HashMap底层数据结构分析

HashMap 1.7

JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列。**

HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 `(n - 1) & hash` 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

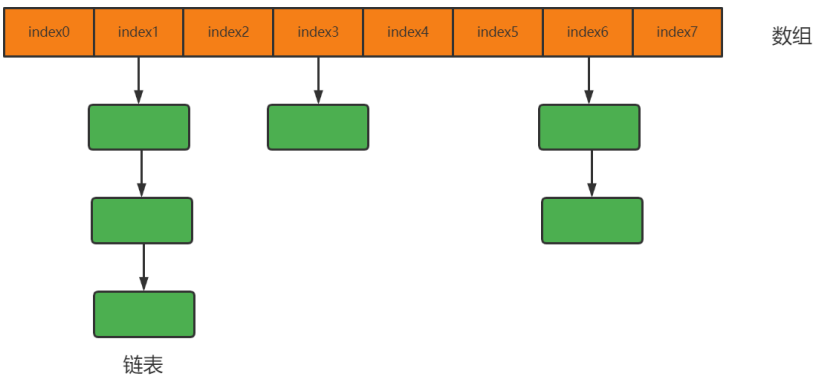
所谓扰动函数指的就是 HashMap 的 hash 方法

```
// JDK 1.8 之后
static final int hash(Object key) {
    int h;
    // key.hashCode()：返回散列值也就是hashcode
    // ^ ：按位异或
    // >>>:无符号右移，忽略符号位，空位都以0补齐
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

// JDK 1.8之前
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

所谓 **“拉链法”** 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

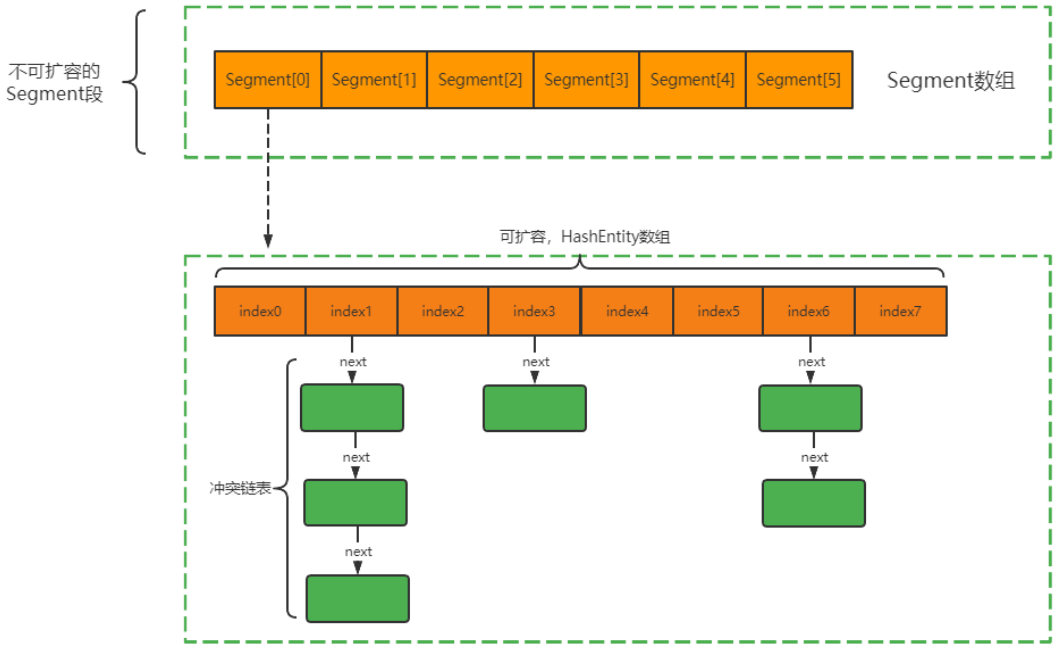


HashMap 1.8

相比于之前的版本，JDK1.8 以后在解决哈希冲突时有了较大的变化。在发生Hash冲突时，当链表长度大于阈值（默认为8）时，会首先调用 `treeifyBin()`方法。这个方法会根据 HashMap 数组来决定是否转换为红黑树。只有当数组长度大于或者等于 64 的情况下，才会执行转换红黑树操作，以减少搜索时间。否则，就是只是执行 `resize()` 方法对数组扩容。

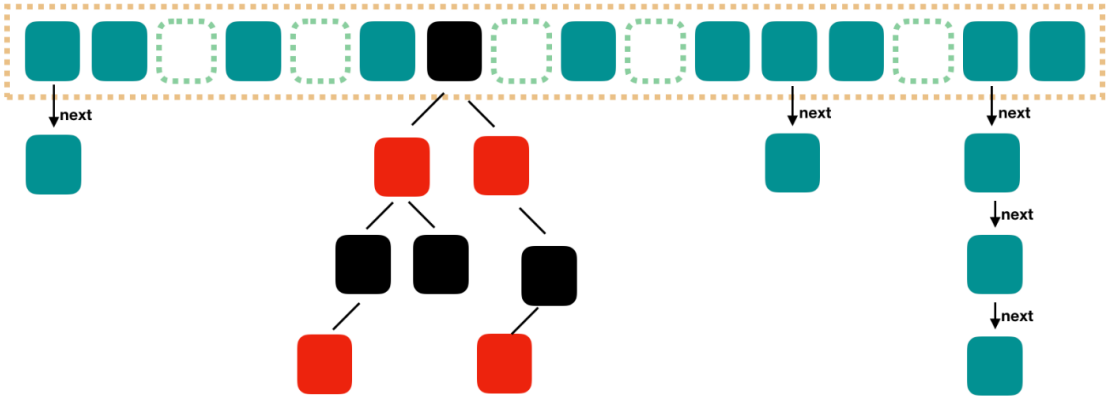
2 ConcurrentHashMap底层数据结构分析

ConcurrentHashMap 1.7



JDK 1.7中的ConcurrentHashMap的存储结构如上图，ConcurrentHashMap由很多个Segment组合，是一个Segment数组，而每一个Segment是一个类似于HashMap的结构，所以每一个 HashMap 的内部可以进行扩容。但是 Segment 的个数一旦**初始化就不能改变**，默认 Segment 的个数是 16 个，你也可以认为 ConcurrentHashMap 默认支持最多 16 个线程并发。

ConcurrentHashMap 1.8



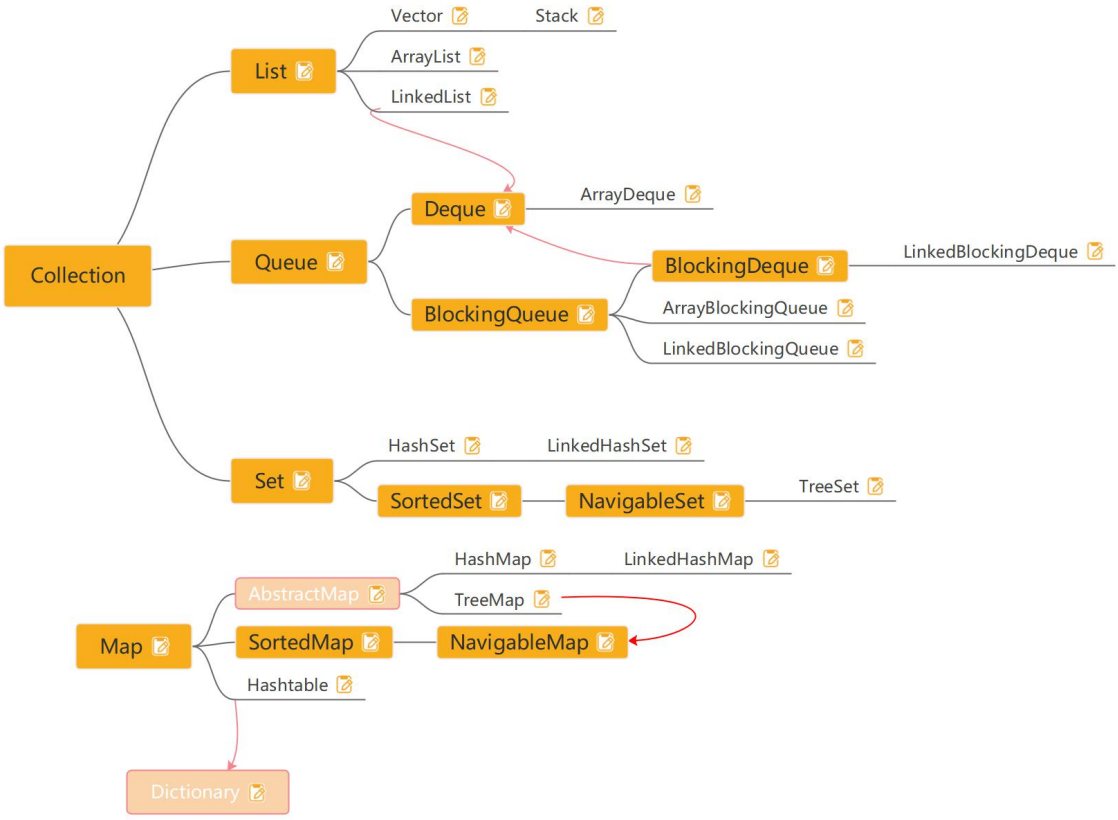
可以发现 Java8 的 ConcurrentHashMap 相对于 Java7 来说变化比较大，不再是之前的 **Segment 数组 + HashEntry 数组 + 链表**，而是 **Node 数组 + 链表 / 红黑树**。当冲突链表达到一定长度时，链表会转换成红黑树。

总结：

Java7 中 ConcurrentHashMap 使用的分段锁，也就是每一个 Segment 上同时只有一个线程可以操作，每一个 Segment 都是一个类似 HashMap 数组的结构，它可以扩容，它的冲突会转化为链表。但是 Segment 的个数一但初始化就不能改变。

Java8 中的 ConcurrentHashMap 使用的 Synchronized 锁加 CAS 的机制。结构也由 Java7 中的 **Segment 数组 + HashEntry 数组 + 链表** 进化成了 **Node 数组 + 链表 / 红黑树**，Node 是类似于一个 HashEntry 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。

3 各个集合类特点




```
1.2 各个集合类的特点
1.2.1   Collection
1，Collection是Collection集合体系的顶级接口
2，一些 Collection 的子实现是有序的，而另一些Collection的子实现则是无序的
3，一些 Collection 的子实现是允许存储重复元素的，而另一些Collection的子实现则是不允许存储重复元素的
4，一些 Collection 的子实现是允许存储null，而另一些Collection的子实现则是不允许存储null
1.2.2   List
1，List的是Collection的子接口
2，List所描述的数据结构是一个线性表子
3，List的子类存储元素都是有序的
4，List的子类存储元素允许重复元素存在
5，List的子类存储元素允许null值

1.2.2.1 ArrayList
1，ArrayList是List接口的一个具体子实现
2，ArrayList底层是一个数组
3，ArrayList底层数组的默认初始长度10，数组的扩容机制1.5倍
4，ArrayList存储的元素是有序的
5，ArrayList存储的元素是允许重复元素存在
6，ArrayList存储的元素是允许null值
7，ArrayList是线程不安全的
```

1.2.2.2 LinkedList

1，LinkedList是List接口的一个子实现，同时它还是Deque接口的自实现
2，LinkedList底层结构是一个双向链表
3，LinkedList存储的元素是有序的
4，LinkedList存储的元素是允许null值
5，LinkedList存储的元素是允许重复元素存在
6，LinkedList是线程不安全的
7，LinkedList不仅仅可以作为一个线性表，还可以作为一个普通队列/双端队列/栈使用

1.2.2.3 Vector和Stack

Vector
1，Vector是List的一个具体实现
2，Vector底层是一个数组
3，Vector底层数组的默认初始长度10
4，Vector扩容机制：
(取决于我们在构造方法有没有给这个Vector一个大于0的增量，
如果给了一个大于0的增量，扩容扩为 旧长度+增量，
如果没有给一个大于0 的增量，扩为原来的二倍)
5，Vector存储的元素是有序的
6，Vector存储的元素是允许重复元素存在
7，Vector存储的元素是允许null值
8，Vector是线程安全的

Stack
1，Stack 是Vector的一个子类
2，Stack底层是个数组(是复用父类Vector的底层数组)
3，Stack表示的数据结构一个栈
4，Stack底层数组默认长度10，默认扩容机制是扩为原来的2倍
5，Stack可以存储null和重复元素
6，Stack是线程安全的

注意1：如果我们要在具体的逻辑中使用一个栈的话，优先使用Deque而非Stack
注意2：如果我们使用Stack的时候，尽量不要直接使用从Vector继承来的方法(不是语法不允许)，我们只是希望Stack作为一个栈存在(而栈操作常用的api是：push pop，peek)

1.2.3 Queue

1，Queue接口是Collection接口一个子接口
2，Queue代表/描述的数据结构是队列
3，Queue下的子类存储数据是有序的
4，Queue下的子类能存储重复元素
5，Queue下的子类不能存储null值 (除了LinkedList以外)

1.2.3.1 Deque

1，Deque接口是Queue接口的一个子接口
2，Deque这个借口不仅仅可以作为普通队列，它还定义了双端队列，栈
3，Deque下的子类存储数据是有序的
4，Deque下的子类允许存储重复数据
5，Deque下的子类不能存储null值 (LinkedList除外)

1.2.3.1.1 ArrayDeque

1，ArrayDeque是Deque接口的一个具体自实现
2，ArrayDeque可以作为：普通队列/双端队列/栈 存在
3，ArrayDeque底层数组(并且是一个循环数组)
4，ArrayDeque底层数组默认的初始容量16， 扩容机制2倍
5，ArrayDeque存储数据能保证有序
6，ArrayDeque允许重复元素

7，ArrayDeque不允许存储null值
8，ArrayDeque是线程不安全的
9,我们可以在构造方法里指定ArrayDeque的底层数组长度:
但是给定的数组长度并不真的是我们给定的值
而是一个大于我们给定值的最小的2的幂值

1.2.3.2 BlockingQueue

BlockingQueue是个阻塞队列:
阻塞队列:
是一个队列
添加的时候如果没有位置，添加操作等待
删除的时候， 队列中没有数据可以删除，删除操作等待

1.2.4 Set

1，Set接口是Collection的子接口
2，Set描述的是集合这种数据结构
3，Set它的一些自实现是有序的，它的有些子实现是无序的
4，Set有些子实现允许存储null值，它有些子实现不允许存储null值
5，Set的子实现都是不允许重复元素

1.2.4.1 HashSet

1，HashSet是Set接口一个具体子实现
2，HashSet的底层持有一个HashMap对象
3，HashSet中存储的元素，实际上都存储到底层持有的HashMap中了，并且作为key存在
4，HashSet中存储的元素是无序的
5，HashSet中不允许存储重复元素(重复元素的定义和HashMap的key的定义一样)
6，HashSet中允许存储null值

1.2.4.2 LinkedHashSet

1，LinkedHashSet是HashSet的一个子类
2，LinkedHashSet底层持有一个LinkedHashMap对象
3，LinkedHashSet存储的元素是有序的(通过底层的双向链表保证有序)
4，LinkedHashSet 不允许存储重复的元素
5，LinkedHashSet 允许存储null值

其它特点遵照于HashSet，以及LinkedHashMap的key值

1.2.4.3 TreeSet

1，TreeSet是Set接口一个子类
2，TreeSet底层持有一个TreeMap对象
3，TreeSet描述的数据结构是一个红黑树(因为TreeMap的数据结构就是红黑树)
3，TreeSet存储的元素大小有序
4，TreeSet不允许存储重复元素(这个地方的重复： 元素大小比较重复)
5，TreeSet不允许存储null值(因为null没有办法比较大小)
6，TreeSet是线程不安全的

1.2.5 Map

1，Map接口是Map集合体系的顶级接口
2，Map所存储的数据是Key-value类型的数据(键值对： 具有自我描述性)
3，Map的子实现有些是有序的，有些子实现是无序的
4，Map的子实现不允许存储重复元素
5，Map有些子实现可以存储null键，有些子实现不允许存储null键

1.2.5.1 HashMap

1， HashMap是Map接口的子实现
2， HashMap表示的数据结构是一个Hash表
3， HashMap的底层结构： 数组+链表+红黑树(jdk1.8之前是存储的数组+链表)

```
4，  HashMap底层数组的默认初始容量 16，  数组的扩容机制 2倍
5，  HashMap不允许存储重复元素key
        Key值重复的定义:满足如下两点
                    Key的hash值一样
                    Key直接相等 或者 相equals
6，  HashMap允许存储null键和null值
7，  HashMap 加载因子默认是0.75
8，  HashMap中当某个数组位置的链表长度超过8，达到9的时候会由链表转化为红黑树
9，  HashMap如果存在红黑树，在扩容的时候和删除元素的时候，都有可能导致红黑树转化为链表
10，HashMap创建的时候,定它一个指定数组长度，它会把底层数组构建成一个长度大于等于给定值的最小的2的幂值
11，HashMap是线程不安全的


1.2.5.2 LinkedHashMap
1，LinkedHashMap是HashMap一个子类
2，LinkedHashMap基本上完全复用了HashMap的底层结构,参数,方法
3，LinkedHashMap的特点基本遵从于HashMap
4，LinkedHashMap底层在HashMap的基础上(数组+链表+红黑树)额外维护了一个双向链表 (这个双向链表用来记录存储顺序)
5，LinkedHashMap存储的key-vlaue数据是有序的

1.2.5.3 TreeMap
1，TreeMap是Map一个子实现
2，TreeMap描述数据结构是红黑树
3，TreeMap底层是链表
4，TreeMap存储的key-vlaue数据是按照key的大小是有序的
5，TreeMap中不允许重复的key
6，TreeMap中不允许null 的key
7，TreeMap是线程不安全的


注意:
如果我们希望在TreeMap中存储数据，key-value，我们可以有两个选择
第一个选择，让key本身可以比较(继承Comparable接口实现compareTo)
        第二个选择，不想让key本身实现Comparable接口实现compareTo方法，
那么就可以在创建TreeMap的时候手动提供一个比较器


1.2.5.4 Hashtable
1，Hashtable 是Map的子实现 (Map是1.2出现， Hashtable是1.0出现)
2，Hashtable底层是数组 + 链表  (没有红黑树结构)
3，数组默认初始容量11， 扩容机制:扩为原来的2倍+1
4，不允许存储null键和null值
5，计算hash值的方式: key.hashCode();
6，线程安全
```

4, 常见集合类面试问答

```
// 1, ArrayList和LinkedList区别
     都是List子实现，描述的数据结构都是线性表，都有序，允许存储重复元素，允许存储null，都线程不安全
     但是ArrayList底层是个数组(默认长度10，扩容机制1.5倍)，LinkedList底层是个双向链表
     并且LinkedList不仅仅是List的子实现还是Deque接口的子实现，也就是说LinkedList除了作为线性表以外还可以作为队列/双端队列/栈
```

// 2，ArrayList和Vector区别
都是List子实现，描述的数据结构都是线性表，都有序，允许存储重复元素，允许存储null
但是ArrayList底层数组(默认长度10，扩容机制1.5倍)，Vector底层数组(默认长度10，扩容机制优先扩增量，如果增量小于等于0那么就扩为原来2倍)
并且ArrayList是jdk1.2时候出现线程不安全，Vector是jdk1.0时候出现线程安全

// 3，List和Set以及Map间的区别
List和Set是Collection的子接口，存储单个数据元素
Map属于Map集合体系存储key-value数据
List集合存储元素有序，元素可以重复
Set存储元素存储元素不可重复
Map的key不可重复

// 4，HashMap和Hashtable的区别
底层结构不同HashMap是:数组+链表+红黑树；Hashtable是:数组+链表
HashMap和Hashtable的底层结构在jdk1.8之前是一样的
HashMap数组默认长度16扩容机制2倍，Hashtable数组默认长度11扩容机制2倍+1
HashMap是jdk1.2时候出现线程不安全，Hashtable是jdk1.0时候出现线程安全
HashMap中key的hash值是hashCode异或上hashCode向right移16位
Hashtable中key的hash值直接用的hashCode
HashMap中的key和value都允许null，Hashtable中的key和value都不允许null

// 5，什么是Iterator
Iterator是Java提供的一个无需关注数据底层，并用于顺序访问集合数据的一个设计模式
通过Iterator我们可以对集合类数据进行原地遍历(不需要再额外复制一份数据)
Collection下的子实现都实现了Iterator遍历数据的方式

// 6，Iterator 和 ListIterator 有什么区别
Iterator这个类型提供了三个方法(hashNext，next，remove)用于对Java集合类数据从前向后遍历
ListIterator是Iterator的子类型，提供给List接口下的实现使用，用于在Iterator向后遍历的基础上向前遍历的功能(hashPrevious，previous)

// 7，iterator和foreach循环的关系
我们使用foreach循环遍历集合类数据，要求被遍历的集合类应该实现了Iterator方法(即Iterable子类型)，因为foreach循环遍历的本质就是依赖于集合类本身的iterator方法进行的(经过代码编译之后，会把foreach循环编译成iterator遍历)
注意：数组使用foreach循环是编译成普通的fori循环

// 8，HashMap的底层结构和存储过程
1，要存储一份key-value数据
2，把key拿出，计算hash值：　通过key的hashCode 异或上hashCode向right移动16位
　　异或的原因：　希望hash值的高位也参与到取模运算，充分散列
3，hash和数组长度取模，的到一个下标(就是key-value数据要存储的位置)
4，判断这个下标位置有没有内容，没有直接存储，存储一个Node类型(key，value，hash，next)
5，如果这个位置已经存储了别的key-value数据 --> 先判断重复
　　重复的依据：key的hash值是否相等 以及，key是否直接相等或者相equals
6，重复，新value覆盖旧value
7，不重复，　接着判断这个位置存储的是树还是单链表
　　7.1 如果是个单链表，按照next方向比较，重复，新value覆盖旧value
　　不重复，最终添加到这个链表尾部
　　如果添加之后导致这个链表过长(超过8达到9)转化为红黑树(如果数组长度小于64，优先扩容)
　　7.2 如果是个树，按照hash值比较大小，确定比较方向
　　最终是否重复：key的hash值是否相等 以及，key是否直接相等或者相equals
　　重复：替换

不重复：添加到红黑树上

8，添加完成，有可能超出阈值(数组长度*加载因子) --> 扩容

9，原本在x位置元素，扩容之后，要么还在x位置，要么在旧长度+x位置 （取决于高位）

10，如果是个树在扩容之后，可能被拆开(拆成两部分:低位和高位)，如果被拆开的是红黑树，那么被拆开的任何一部分数据量小于等于6的时候，由红黑树转化为链表

// 9，HashMap中为什么引入红黑树

HashMap在jdk1.8的时候引入红黑树，在引入红黑树之前，HashMap的结构是数组+链表结构

引入红黑树的原因是担心存储的key-value数据经过散列之后，存储到某一个数组位置的元素过多,导致链表边长效率降低.

// 10，Collection和Collections有什么区别

Collection是Collection集合体系的顶级结构，是个集合接口用来定义了一个存储数据的容器概念

Collections则是集合类的一个工具类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作

// 11，Java中的Arrays和Collections有什么区别

Arrays是针对数组的工具类,可以对数组进行排序/查找/复制填充等功能。 大大提高了开发人员的工作效率。

Collections则是集合类的一个工具类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作

// 12，Queue的add方法和offer方法的区别

Queue的add方法和offer方法都是对队列添加的方法

当Queue不可以添加数据的时候继续使用add方法会抛出异常

当Queue不可以添加数据的时候继续使用offer方法不会抛出异常，会返回一个false

// 13，什么是阻塞队列，集合类中的阻塞队列接口

一个大小有限的队列

当队列添加满的时候，添加线程等待/阻塞

当队列为空的时候，删除线程等待/阻塞

// 14，HashMap在jdk1.7和jdk1.8有什么区别

HashMap在jdk1.8的时候引入了红黑树，用来解决存储的key-value数据经过散列之后，存储到某一个数组位置的元素过多,导致链表边长效率降低的问题.

HashMap在jdk1.7时候添加数据在某个数组位置的链表上是采用头插法，而JDK1.8及之后使用的都是尾插法.(所以在1.7的时候添加数据环形链表死循环)

扩容后数据存储位置的计算方式也不一样;

// 15，怎么确保一个集合类不能被修改

通过Collections.unmodifiable方法，把集合类包装成不可修改的视图结果使用

eg:

```
Collections.unmodifiableCollection();
Collections.unmodifiableList();
Collections.unmodifiableSet();
Collections.unmodifiableMap();
```

// 16，怎么使一个Java集合类变成一个线程安全的集合类?

通过Collections.synchronized方法，获得一个线程安全的集合类

eg:

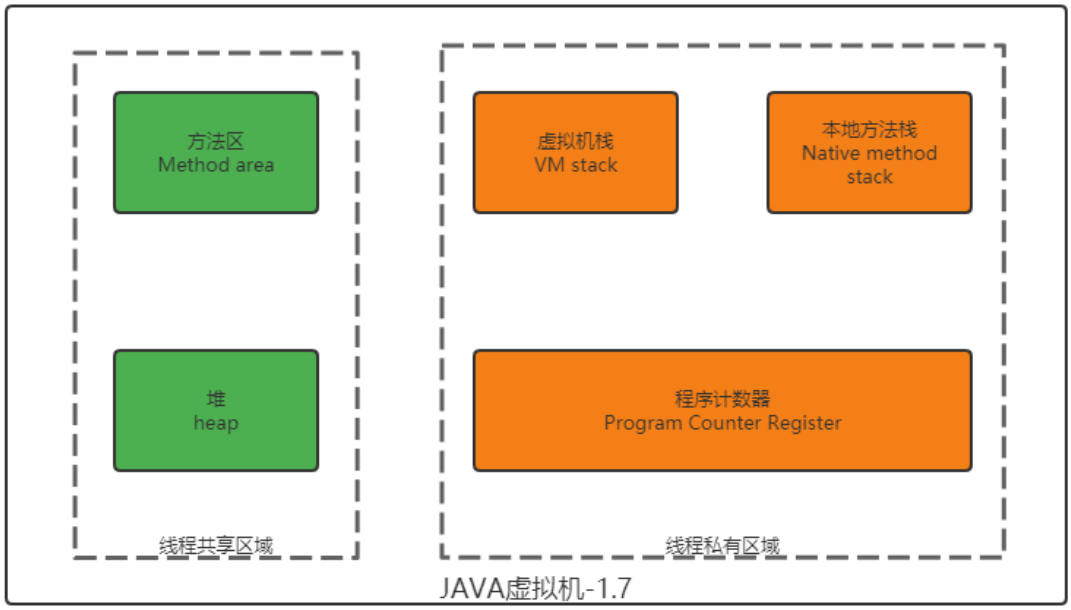
```
Collections.synchronizedCollection();
Collections.synchronizedList()
Collections.synchronizedSet();
Collections.synchronizedMap();
```

JVM

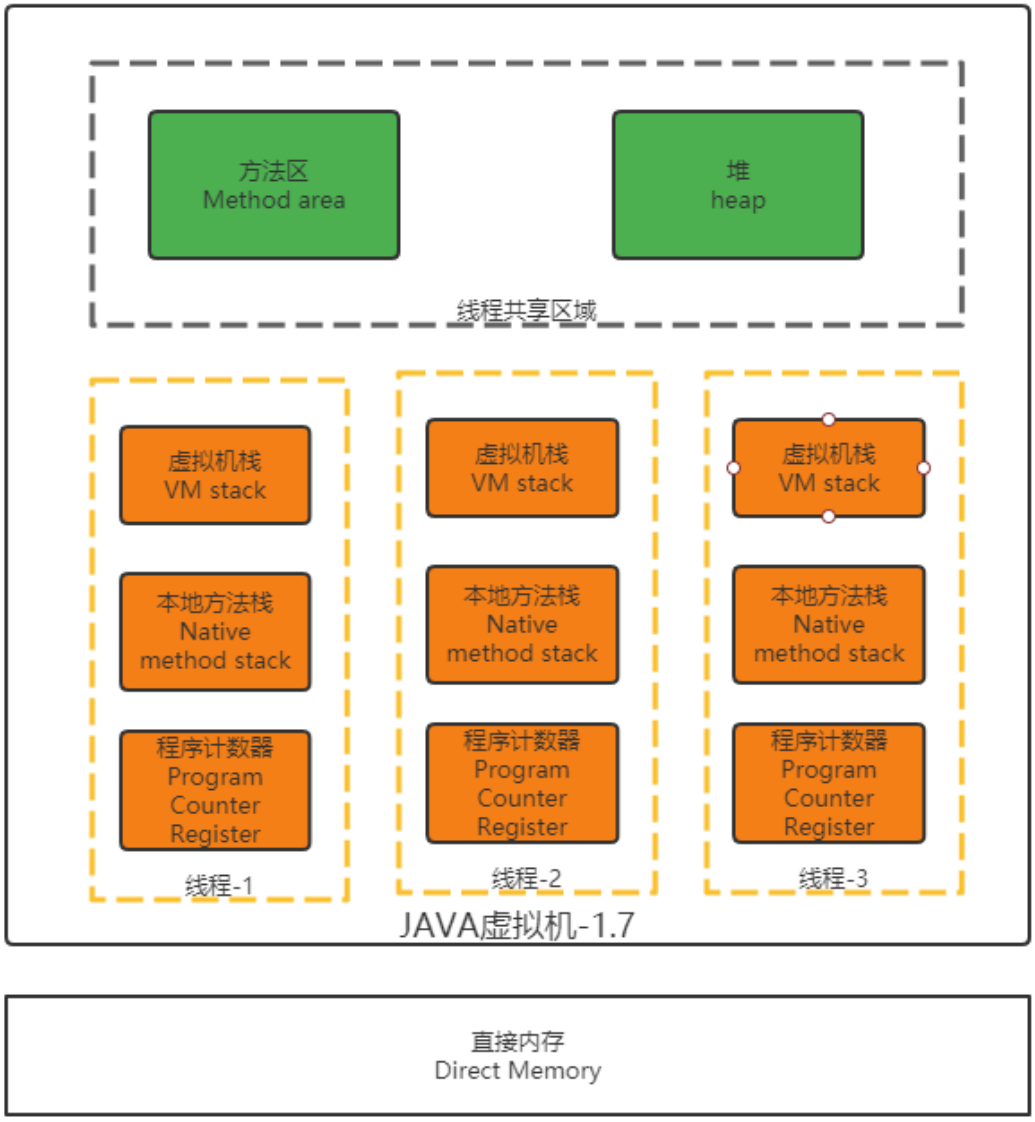
3 介绍一下Java内存区域（运行时数据区）

JVM运行时内存分布图在JDK1.8的时候发生了比较大的改动，故在此分开来看

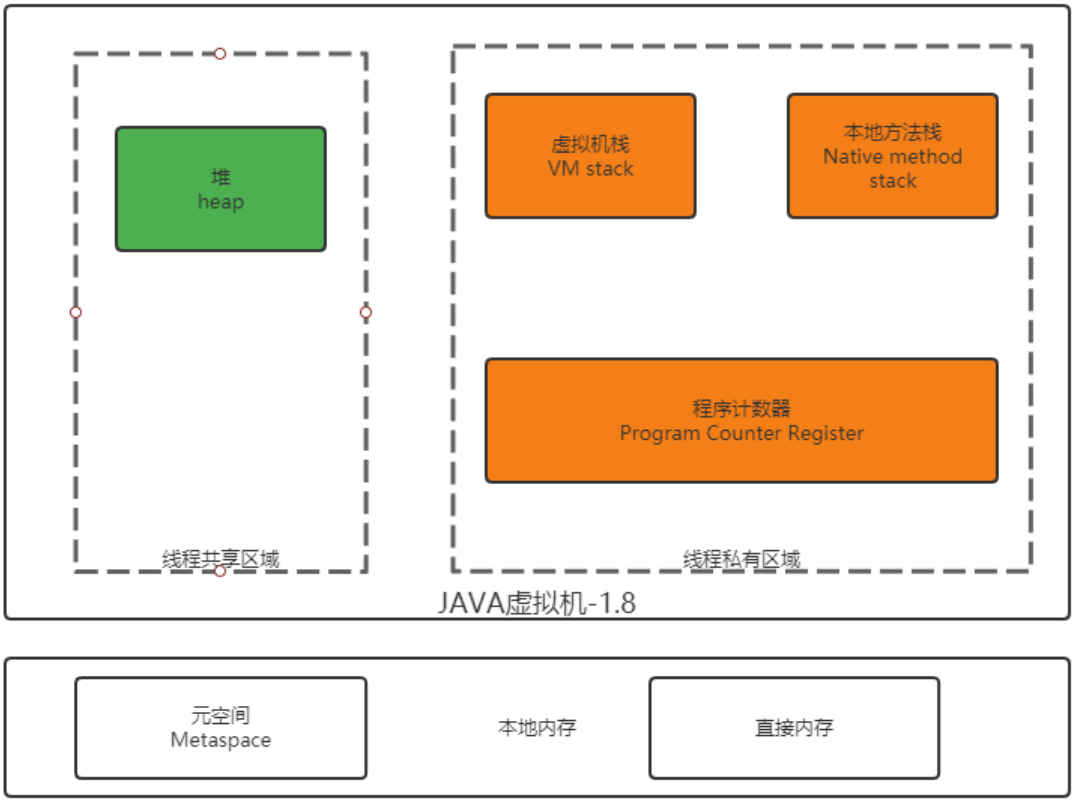
- JDK 1.7



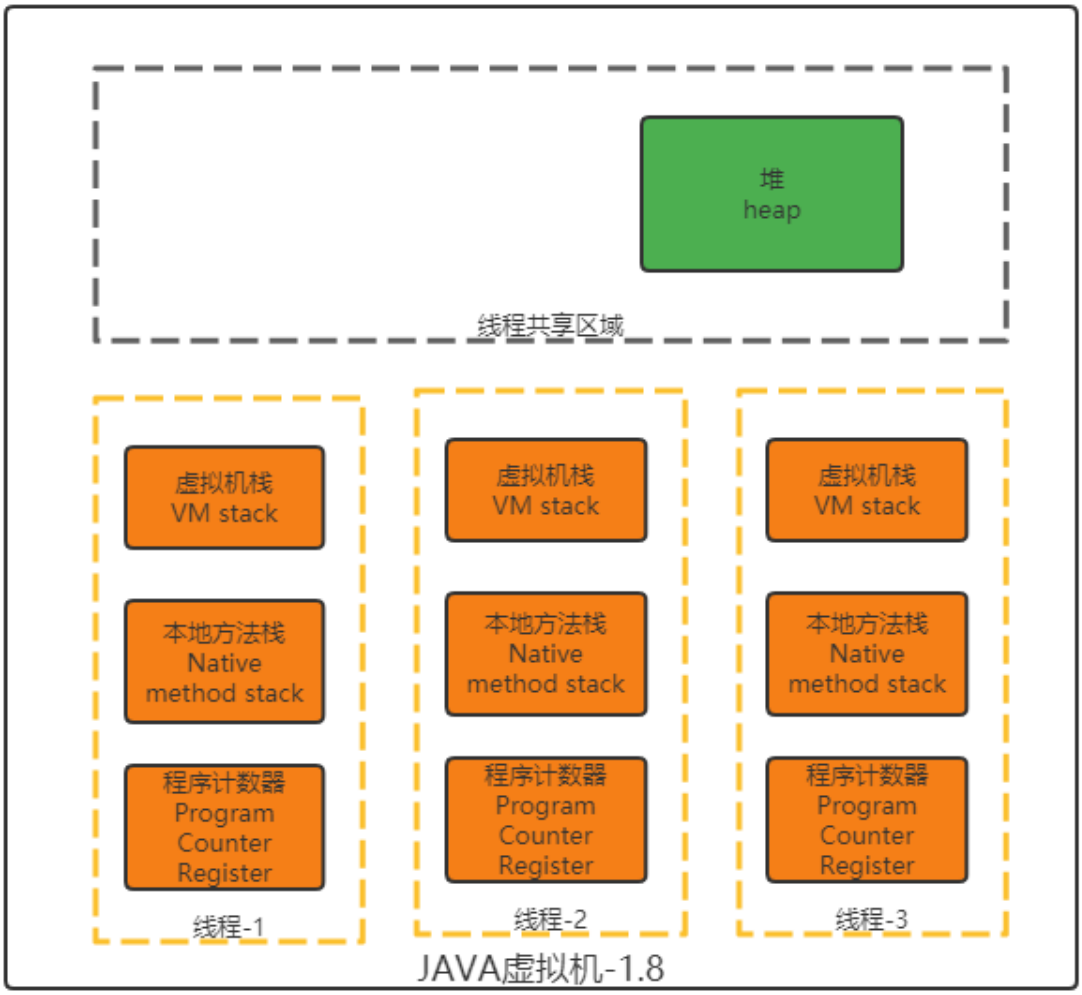
即具体来说如下图所示


方法区
Method area
堆
heap
线程共享区域
虚拟机栈
VM stack
本地方法栈
Native
method stack
程序计数器
Program
Counter
Register
线程-1
虚拟机栈
VM stack
本地方法栈
Native
method stack
程序计数器
Program
Counter
Register
线程-2
虚拟机栈
VM stack
本地方法栈
Native
method stack
程序计数器
Program
Counter
Register
线程-3
JAVA虚拟机-1.7
直接内存
Direct Memory

- JDK 1.8


堆
heap
线程共享区域
虚拟机栈
VM stack
本地方法栈
Native method
stack
程序计数器
Program Counter Register
线程私有区域
JAVA虚拟机-1.8
元空间
Metaspace
本地内存
直接内存

具体来说如下图所示



名词解释：

- **程序计数器**
  - 程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在Java虚拟机的概念模型里，字节码解释器工作时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，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
  - 另外，**为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。**
  - 如果线程正在执行的是一个Java方法，那么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；如果正在执行的是本地方法，那么这个计数器值则应为空
- **虚拟机栈**
  - 和程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期与线程相同，描述的是Java方法执行的内存模型，每次方法调用的数据都是通过栈传递的。
  - Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息等，每一个方法被调用直至执行完毕的过程，就对应着 一个栈帧在虚拟机栈中从入栈到出栈的过程。
  - **Java 虚拟机栈会出现两种错误：** `StackOverFlowError` **和** `OutOfMemoryError`**。**

        - `StackOverFlowError`： 若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出`StackOverFlowError`错误。
        - `OutOfMemoryError`： Java 虚拟机栈的内存大小可以动态扩展， 如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出`OutOfMemoryError`异常。

        > 说明：HotSpot虚拟机的栈容量是不可以动态扩展的，以前的Classic虚拟机倒是可以。

    - Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。
- 本地方法栈

  和虚拟机栈所发挥的作用非常相似，区别是： **虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。**
- 堆

  Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。**

  Java 世界中“几乎”所有的对象都在堆中分配，但是，随着 JIT 编译器的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。从 JDK 1.7 开始已经默认开启逃逸分析，如果某些方法中的对象引用没有被返回或者未被外面使用（也就是未逃逸出去），那么对象可以直接在栈上分配内存。

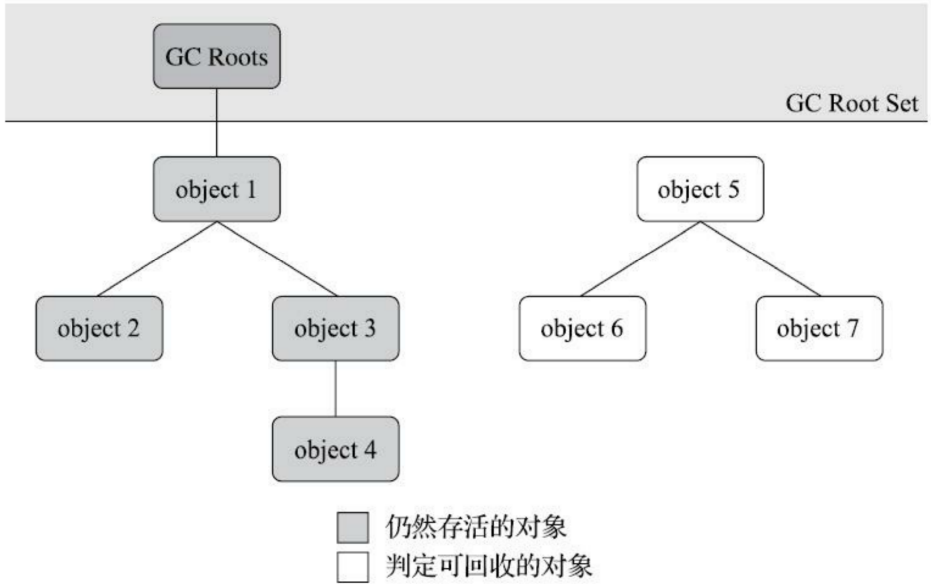
  Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆（Garbage Collected Heap）**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**
- 方法区

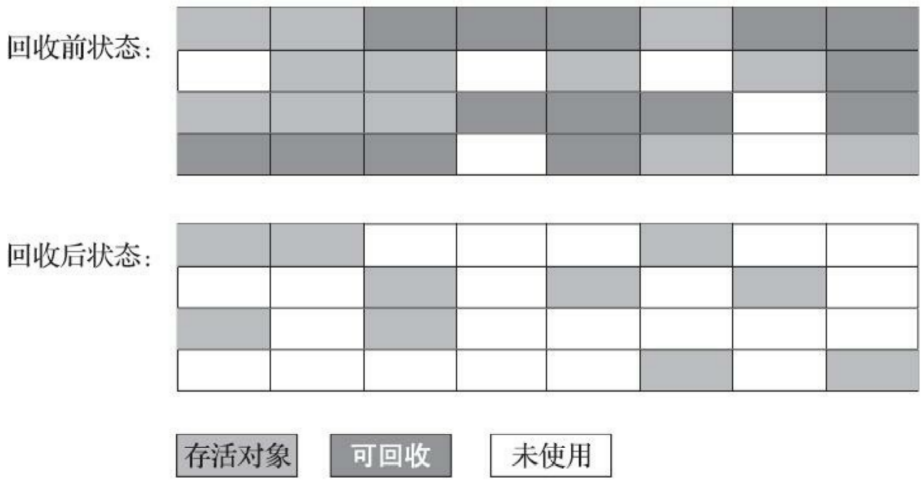
  方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 **Java 虚拟机规范把方法区描述为堆的一个逻辑部分**，但是它却有一个别名叫做 **Non-Heap（非堆）**，目的应该是与 Java 堆区分开来。

4 说一下运行时常量池

- **运行时常量池（Runtime Constant Pool）是方法区的一部分。**
- Class文件中除了有类的版本、字 段、方法、接口等描述信息外，还有一项信息是**常量池表（Constant Pool Table）**，用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。
- Java虚拟机对于Class文件每一部分（自然也包括常量池）的格式都有严格规定，如每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可、加载和执行，但对于运行时常量池，《Java虚拟机规范》并没有做任何细节的要求，不同提供商实现的虚拟机可以按照自己的需要来实现这个内存区域，不过一般来说，除了保存Class文件中描述的符号引用外，还会把由符号引用翻译出来 的直接引用也存储在运行时常量池中。
- 运行时常量池相对于Class文件常量池的另外一个重要特征是具备**动态性**，Java语言并不要求常量一定只有编译期才能产生，也就是说，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的 intern()方法。 既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存 时会抛出**OutOfMemoryError异常。**

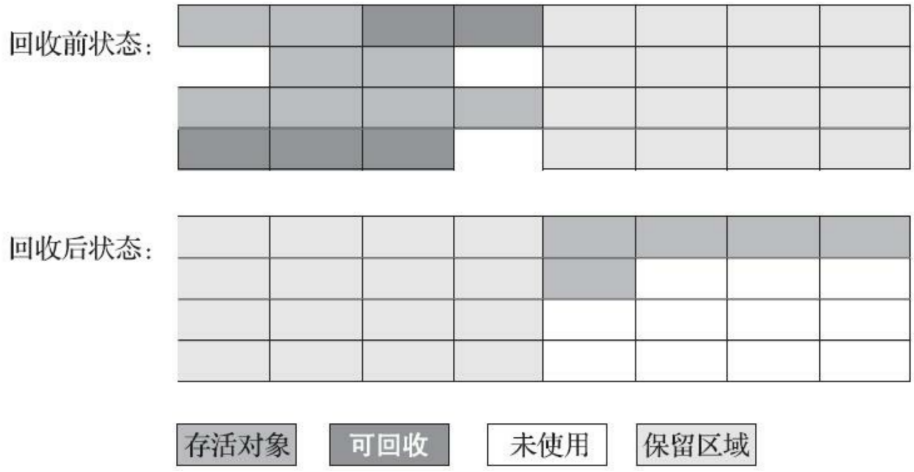
5 JVM垃圾回收

- **如何确定垃圾(判断对象的死活)**
  - **引用计数算法**
    - **思想**: 在对象中添加一个引用计数器，每当有一个地方 引用它时，计数器值就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可 能再被使用的。
    - **弊端**: 无法解决循环引用的问题
  - **可达性分析算法(根搜索算法)**
    - **思想**: 通过 一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过 程所走过的路径称为“引用链”（Reference Chain），如果某个对象到GC Roots间没有任何引用链相连， 或者用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的。
    - 

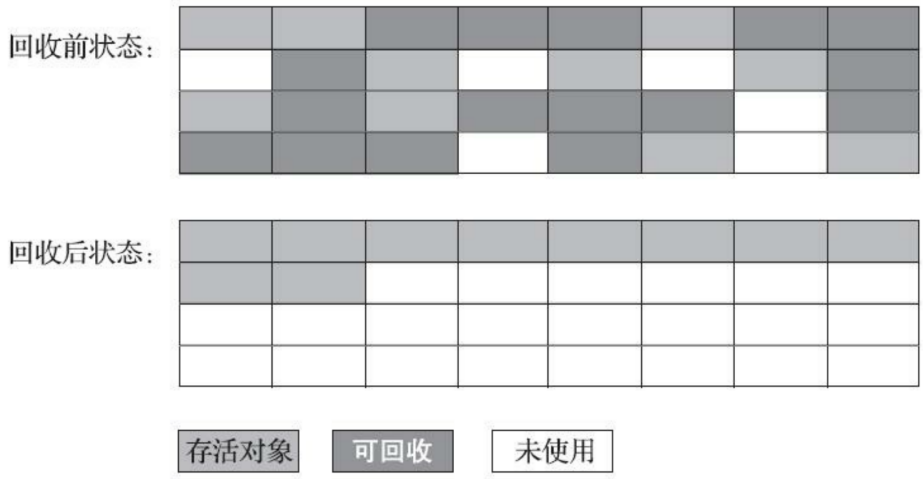
    - 对象object 5、object 6、object 7虽然互有关联，但是它们到GC Roots是不可达的，因此它们将会被判定为可回收的对象。
    - **可作为GC Roots的对象包括以下几种**
      1. 在方法区中类静态属性引用的对象，譬如Java类的引用类型静态变量
      2. ·在方法区中常量引用的对象，譬如字符串常量池（String Table）里的引用
      3. ·在本地方法栈中JNI（即通常所说的Native方法）引用的对象
- **垃圾回收算法:**
  - **标记清除算法（Mark-Sweep）**
    - 思想: 首先标记出所有需要回 收的对象，在标记完成后，统一回收掉所有被标记的对象，也可以反过来，标记存活的对象，统一回 收所有未被标记的对象。
    - 

    - 弊端: 会产生内存碎片

- **标记复制算法（Semispace Copying）**
  - **思想**: 它将可用 内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着 的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。如果内存中多数对象都是存 活的，这种算法将会产生大量的内存间复制的开销，但对于多数对象都是可回收的情况，算法需要复 制的就是占少数的存活对象，而且每次都是针对整个半区进行内存回收，分配内存时也就不用考虑有 空间碎片的复杂情况，只要移动堆顶指针，按顺序分配即可



  - 弊端: 这种复制回收算法的代价是将可用内存缩小为了原来的一半，空间浪费未免太多了一 点。
- **标记整理算法（Mark-Compact）**
  - 思想: 其中的标记过程仍然与"标记-清除"算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存



  - 弊端: 复杂耗时
- 分代收集理论
  - 基于2个假说:
    - 弱分代假说（Weak Generational Hypothesis）: 绝大多数对象都是朝生夕灭的.
    - 强分代假说（Strong Generational Hypothesis）： 熬过越多次垃圾收集过程的对象就越难以消亡。
  - 新生代与老年代: 每次垃圾收集时都发现有大批对象死去，而每次回收后存活的少量对象，将会逐步晋升到老年代中存放(每熬过一次新生代垃圾回收,年龄+1,默认15岁就会被移动到老年代).对于新生代的对象,一般采用标记复制算法,因为存活的少,复制的就少.对于老年代对象,一般采用标记整理算法,这些对象都具有"老而不死"的特性,需要整理的对象就会很少.

并发

6 什么是线程和进程？有什么区别？

> - **进程**: 计算机程序在某个数据集合上的运行活动,进程是操作系统进行资源调度与分配的基本单位.简单理解就是正在内存中运行的计算机程序或软件
> - **线程**:进程中有多个子任务,每个子任务就是一个线程.从执行路径的角度看,一条执行路径(一个执行控制单元)就是一个线程.线程是CPU进行资源调度与分配的基本单位.
> - **进程与线程的关系**:线程依赖于进程而存在,一个进程最少有1个线程.线程之间相互独立, 共享进程资源.

7 并发和并行有什么区别？

> - **并发：** 同一时间段，多个任务都在执行 (单位时间内不一定同时执行)；
> - **并行：** 单位时间内，多个任务同时执行。

8 为什么要使用多线程呢？

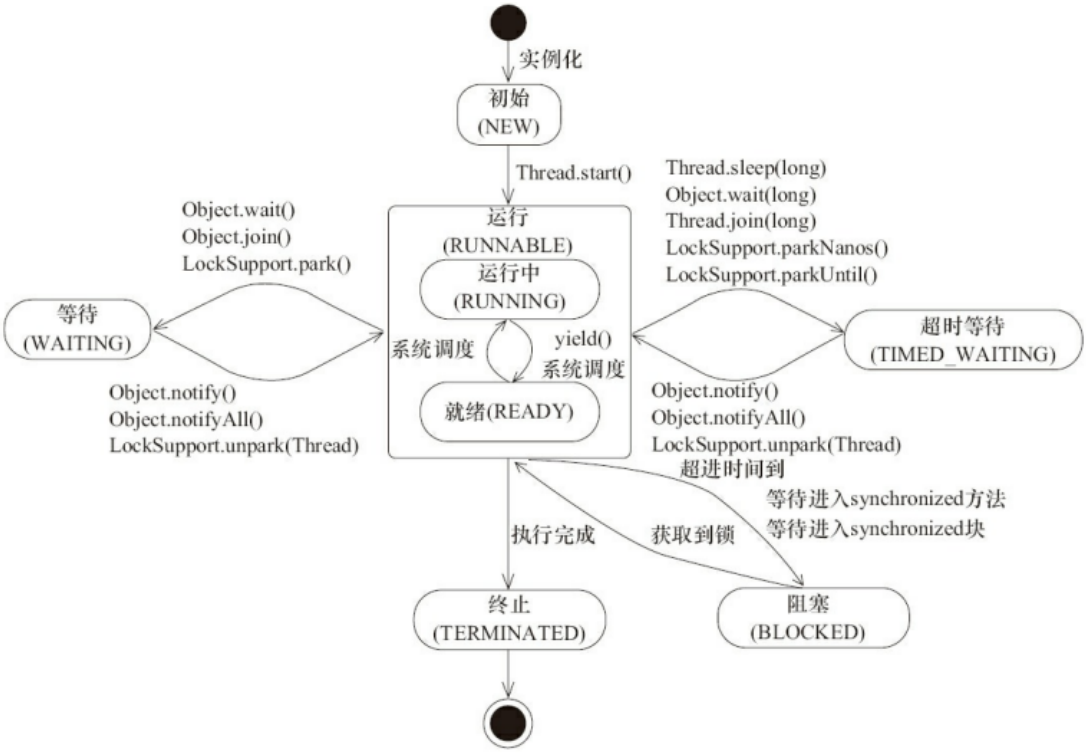
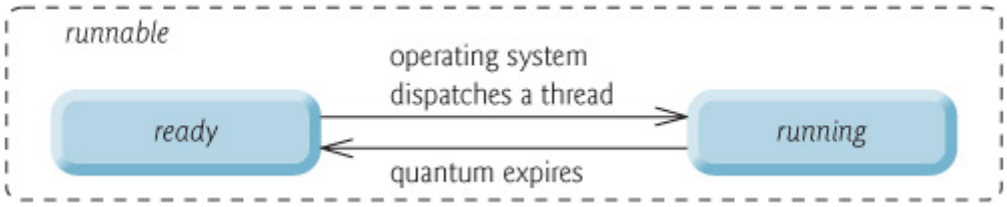
> 先从总体上来说：
>
> - **从计算机底层来说：** 线程可以比作是轻量级的进程，是程序执行的最小单位,线程间的切换和调度的成本远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
> - **从当代互联网发展趋势来说：** 现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。
>
> 再深入到计算机底层来探讨：
>
> - **单核时代：** 在单核时代多线程主要是为了提高 CPU 和 IO 设备的综合利用率。举个例子：当只有一个线程的时候会导致 CPU 计算时，IO 设备空闲；进行 IO 操作时，CPU 空闲。我们可以简单地说这两者的利用率目前都是 50%左右。但是当有两个线程的时候就不一样了，当一个线程执行 CPU 计算时，另外一个线程可以进行 IO 操作，这样两个的利用率就可以在理想情况下达到 100%了。
> - **多核时代:** 多核时代多线程主要是为了提高 CPU 利用率。举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU 只会一个 CPU 核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

9 使用多线程可能带来什么问题？

> 并发编程的目的就是为了能提高程序的执行效率提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：内存泄漏、死锁、线程不安全等等。

10 说一下线程的生命周期和状态

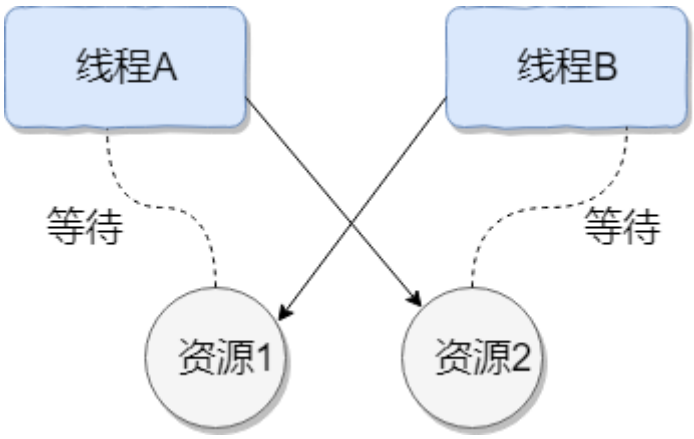
> Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。
>
状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

> 线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：
>
> 
>
>
> 由上图可以看出：线程创建之后它将处于 **NEW（新建）** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY（可运行）** 状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 **RUNNING（运行）** 状态。
>
> 操作系统隐藏 Java 虚拟机（JVM）中的 READY 和 RUNNING 状态，它只能看到 RUNNABLE 状态，所以 Java 系统一般将这两个状态统称为 **RUNNABLE（运行中）** 状态 。
>
> 
>
>
> 当线程执行 `wait()`方法之后，线程进入 **WAITING（等待）** 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep（long millis）`方法或 `wait（long millis）`方法可以将 Java 线程置于 TIMED WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED（阻塞）** 状态。线程在执行 Runnable 的`run()`方法之后将会进入到 **TERMINATED（终止）** 状态。

11 什么是线程死锁？如何避免死锁？

线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过一个例子来说明线程死锁,代码模拟了上图的死锁的情况 (代码来源于《并发编程之美》):

```
public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource2");
                synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get
resource2");
                }
            }
        }, "线程 1").start();

        new Thread(() -> {
            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get resource2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource1");
                synchronized (resource1) {
                    System.out.println(Thread.currentThread() + "get
resource1");
                }
            }
        }, "线程 2").start();
    }
}
```

Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 `synchronized (resource1)` 获得 `resource1` 的监视器锁，然后通过 `Thread.sleep(1000);`让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

学过操作系统的朋友都知道产生死锁必须具备以下四个条件：

1. 互斥条件：该资源任意一个时刻只由一个线程占用。
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件:线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

如何预防死锁？ 破坏死锁的产生的必要条件即可：

1. **破坏请求与保持条件** ：一次性申请所有的资源。
2. **破坏不剥夺条件** ：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
3. **破坏循环等待条件** ：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

如何避免死锁?

避免死锁就是在资源分配时，借助于算法（比如银行家算法）对资源分配进行计算评估，使其进入安全状态。

安全状态指的是系统能够按照某种进行推进顺序（P1、P2、P3.....Pn）来为每个进程分配所需资源，直到满足每个进程对资源的最大需求，使每个进程都可顺利完成。称<P1、P2、P3.....Pn>序列为安全序列。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
        new Thread(() -> {
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource2");
                synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get
resource2");
                }
            }
        }, "线程 2").start();
```

Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 1,5,main]waiting get resource2
Thread[线程 1,5,main]get resource2
Thread[线程 2,5,main]get resource1
Thread[线程 2,5,main]waiting get resource2
Thread[线程 2,5,main]get resource2

Process finished with exit code 0
```

我们分析一下上面的代码为什么避免了死锁的发生?

线程 1 首先获得到 resource1 的监视器锁,这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

12 说一下sleep()方法和wait()方法的区别和共同点?

- 两者最主要的区别在于：`sleep()` **方法没有释放锁，而** `wait()` **方法释放了锁** 。
- 两者都可以暂停线程的执行。
- `wait()` 通常被用于线程间交互/通信，`sleep()` 通常被用于暂停执行。
- `wait()` 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 `notify()` 或者 `notifyAll()` 方法。`sleep()` 方法执行完成后，线程会自动苏醒。或者可以使用 `wait(long timeout)` 超时后线程会自动苏醒。

13 为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来!

new 一个 Thread，线程进入了新建状态。调用 `start()` 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 `start()` 会执行线程的相应准备工作，然后自动执行 `run()` 方法的内容，这是真正的多线程工作。 但是，直接执行 `run()` 方法，会把 `run()` 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结： 调用 `start()` **方法方可启动线程并使线程进入就绪状态，直接执行** `run()` **方法的话不会以多线程的方式执行。**

14 线程池有哪几种配置方式？缺点是什么?

JUC包提供了以下几种创建线程池的方式

- newCachedThreadPool
  - 底层：返回ThreadPoolExecutor实例，corePoolSize为0；maximumPoolSize为Integer.MAX_VALUE；keepAliveTime为60L；unit为TimeUnit.SECONDS；workQueue为SynchronousQueue(同步队列)
  - 通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。
  - 适用：执行很多短期异步的小程序或者负载较轻的服务器
- newFixedThreadPool
  - 底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，corePoolSize为nThread，maximumPoolSize为nThread；keepAliveTime为0L(不限时)；unit为：TimeUnit.MILLISECONDS；WorkQueue为：new LinkedBlockingQueue() 无界阻塞队列
  - 通俗：创建可容纳固定数量线程的池子，每隔线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞

队列)
    - 适用：执行长期的任务，性能好很多
- newSingleThreadExecutor
    - 底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；unit为：TimeUnit.MILLISECONDS；workQueue为：new LinkedBlockingQueue() 无界阻塞队列
    - 通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)
    - 适用：一个任务一个任务执行的场景
- NewScheduledThreadPool
    - 底层：创建ScheduledThreadPoolExecutor实例，corePoolSize为传递来的参数，maximumPoolSize为Integer.MAX_VALUE；keepAliveTime为0；unit为：TimeUnit.NANOSECONDS；workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列
    - 通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构
    - 适用：周期性执行任务的场景

线程池任务执行流程：

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

第三章 数据库

MySQL

1 说一下数据库的引擎和锁?

Mysql的引擎有很多种，最常见是MyISAM和Innodb。锁分为三种，行级锁、表级锁、页级锁。

- Innode支持页级锁和行级锁，默认为行级锁。行级锁开销大，加锁慢，并发程度高。
- MyISAMA采用表级锁，表级锁开销小，加锁快，并发程度低。

页级锁的开销和加锁时间，介于行级锁和表级锁之间。表级锁速度快，但是冲突多，行级锁冲突少，但是速度慢。采取折中的页级锁。

2 Mysql如何分库分表

首先要清楚为什么要进行分库分表，当mysql内数据量到达500w条数据之后，由于查询的数据过多，即便进行了很多的优化，还是不可避免的出现操作性能的下降，这时就要考虑对其进行切分，减缓数据库的压力，减少查询时间。

数据库进行切分时，按照切分的方式不同，可以分为垂直切分和水平切分。

- **垂直切分**又有垂直分库和垂直分表两种。
    - 垂直分库，就是业务的特点，将不同业务之间关联度比较低的表存储在不同的数据库内。比如用户系统、订单系统；垂直分表，就是根据第一范式的要求，原本需要存放在同一个表里

面的诸多字段拆分到不同的表中，比如将一些不经常使用的字段，或者是较大的字段与那些需要经常使用的字段拆分到不同的表中。
    - 当一个应用在垂直方向很难再进一步切分时，或者垂直切分后，但是表内数据的行数仍然非常多时，这个时候查询数据的时候仍然会有性能问题时，就要考虑进行水平切分。
- 水平切分分为**库内分表**，比如将用户表进一步切分为用户表1、用户表2等等。每个表内的数据量变少，但是这个时候仍然会有一个问题，多个表同时存放在一个库内时，仍然会使用同一台机器的cpu、io等，所以还可以进一步切分到不同机器上**分库分表**。

在分库分表操作完成之后，需要对数据库进行操作时，通常有两种方式，一种是通过代码设置某些sql应该访问哪个库或者哪个表，另外一种方式是交给中间件来代为处理，例如MyCat或者是Sharing-JDBC

3.说一说mysql的索引，主键索引和普通索引的区别?

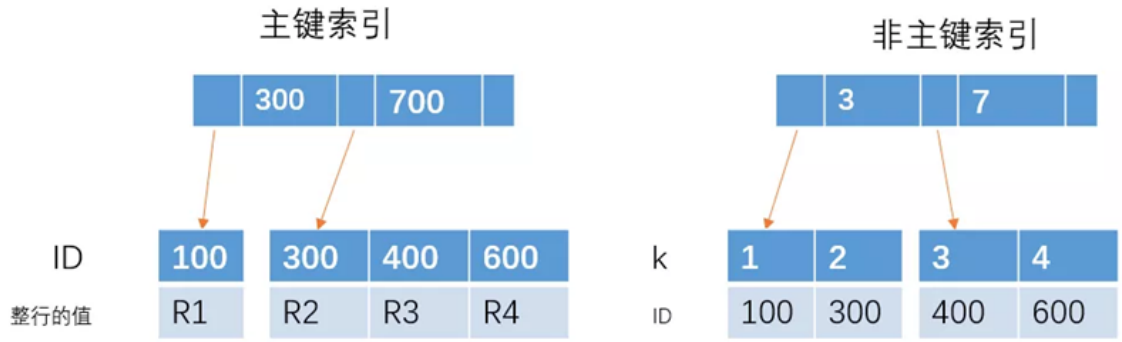
索引可以理解为了帮助数据库高效获取数据，它是一种排序好的数据结构。所以本质上来说，索引就是一种数据结构。那数据结构的话，我们平时熟悉的有比如二叉树、红黑树、Hash表、B-树等。

那mysql的索引它采用的是哪种数据结构呢？主要采用的B+树。

为什么不采用其他数据结构的原因，首先是二叉树和红黑树，当数据量变得非常多的时候，树的高度也变得非常的高，而这些树是存在于磁盘中的，也就是说需要访问对应高度次的磁盘，才可以取到最终的数据。每访问一次磁盘，就需要花费一定的时间，所以这两种数据结构用在数据库中并不是十分的恰当。Hash表，是对相应的数据进行一次hash运算，得到hash值，但是hash值的话并不是有序的，所以如果想查询大于某一个数值的所有数据时，仍然免不了会全表扫描。

针对二叉树和红黑树存在的问题，我们设想能否让树的高度不高，但同时可以存储很多的元素呢？那就是BTree，但是这里面需要注意的是也不是正常的B树，而是B+树，因为B树无论叶子节点还是非叶子节点，都会保存数据，但是如果在非叶子节点保存数据，那么在非叶子节点中能够保存的索引指针就会变得少，指针变少，如果还想要存海量数据，那只能够增加树的高度。会导致磁盘io变多，查询性能变低。所以数据库采用的索引是B树的变种，B+树，在非叶子节点并不会存放数据。

主键索引和非主键索引的区别在于：非主键索引的叶子节点存放的是主键的值，而主键索引的叶子节点存放的是整行数据，所以非主键索引也叫做二级索引。



4 说一下悲观锁和乐观锁

悲观锁认为数据库并发的概率很高，数据很有可能会被并发修改，因此需要借助于数据库的锁机制，将数据进行加锁，然后修改。悲观锁的执行流程如下：

1. 首先修改数据前，对当前数据加上锁
2. 加锁如果失败，表明数据正在被修改，要等待其他操作完毕再操作
3. 加锁成功，开启事务，对这个数据进行修改，事务提交解锁

乐观锁认为数据一般情况下不会发生冲突，需要更新数据的时候，会去比较一下版本号，判断期间有没有别人更新了该数据。可以在表内维护一个版本号字段来实现乐观锁。

乐观锁和悲观锁之间各有优缺点，比如在读数据比较多的情况下，发生冲突的概率很低，所以这个时候可以使用乐观锁；但是如果需要频繁的进行更改数据，这个时候，用悲观锁反而更加的有效，避免了很多不必要的问题的产生。

5 说一下数据库的传播行为？隔离级别？

什么叫传播行为呢？比如在当前事务内调用了其他方法，那么这个方法的行为和当前事务之间存在怎么样的关系呢？，数据库一共有7中传播行为：

- Propagation_required:如果当前没有事务就创建一个事务，如果存在事务，则加入该事务。
- Propatation_supports:支持当前事务，如果当前存在事务，则加入该事务，如果不存在该事务，则以非事务执行
- Propagation_requires_new:无论当前存不存在事务，创建一个新事务
- Propagation_mandatory:如果当前存在事务，则加入该事务；如果不存在事务，则抛出异常
- Propagation_not_supported:以非事务方式运行，如果当前存在事务，则把当前事务挂起
- Propagation_never：以非事务方式运行，如果当前存在事务，则抛出异常
- Propagation_nested：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

数据库的隔离级别由低到高分为四种：

1. read_uncommitted
2. read_committed
3. repeatable_read
4. serializable

6 说一下MyISAM和InnoDB的区别？

区别：

1. InnoDB支持事务，MyISAM不支持事务
2. InnoDB支持外键，MyISAM不支持外键
3. Innodb的B+树主键索引的叶子节点就是数据文件，非主键索引的叶子节点存放的是主键的值；MyISAM的B+树索引的叶子节点都是数据文件的地址
4. Innodb不保存表的具体行数，统计全部数量需要全表扫描；MyISAM保存了整个表的行数
5. MyISAM表格可以被压缩后进行数据查询
6. Innodb支持表级锁、行级锁，默认行级锁；MyISAM支持表级锁。但是Innodb的行锁是实现在索引上的，也就是说，如果没有命中索引，也将使用表锁。
7. Innodb表必须有主键，如果没有设置，则会自己新建一个用来当做主键；MyISAM没有
8. Innodb存储文件为frm、ibd，而MyISAM是frm、MYD、MYI

7 Datetime和timestamp的精确度区别？

Datetime占用8个字节，而timestamp占用4个字节。

Timestamp所能存储的时间范围是1970-01-01 00：00：01.000000到2038-01-19 03：14：07.999999。

而datetime所能存储的时间范围是1000-01-01 00：00：01.000000到9999-12-31 23：59：59.999999。

8 列举几种防范sql注入的方式

最简单有效的就是采用PreparedStatement预编译sql语句来避免这些问题，不会把用户输入的各种参数当做sql语句的关键字进行解析。

或者可以使用正则表达式匹配用户输入的数据，或者用字符串过滤等

9 拷贝表的sql命令

```
Create table new_table_name select * from old_table_name;
```

10 Sql触发器了解吗？存储过程呢?

Sql的触发器与事件相关联的。它并不是通过程序调用，也不是由手动开启，而是由事件来触发，比如说当执行完某个sql语句时，会触发其他sql语句的执行。

因为触发器的sql里面包含;，而sql语句遇到分号就结束了，所以首先设置结束符为其他任意符号，后面再修改回来即可。语法含义：

1. 监视位置：on哪张表
2. 监视事件：insert/update/delete表示对这张表的什么操作
3. 触发时间：在事件的何时触发，之前还是之后,before/after
4. 触发事件：begin后面的代码其中new表示刚刚插入被监视表的数据

存储过程可以理解为已经编译好的一个可执行过程，包含一个或者多个sql语句。你可以定义为是使用SQL语言定义方法，然后可以去调用方法。

```
mysql> delimiter !!    #将语句的结束符号从分号;临时改为两个!!(可以是自定义)
mysql> create trigger a_insert after
    -> insert on a1
    -> for each row
    -> begin
    -> insert into a2 set id = new.id,num = new.num;
    -> end!!
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;    #将语句的结束符号恢复为分号
```

11 如果sql 查询很慢，如果排查并优化?

Sql如果查询速度很慢，首先应该启用慢查询sql分析，记录下哪些sql的执行速度较低。接着就是对这些sql语句进行分析，找出其中非瓶颈所在。可以使用explain来模拟优化器执行sql的整个过程。知道数据库是如何处理sql语句的。另外，当了解到问题所在的原因，比如是由于数据量过多，查询的时候进行全表扫描导致等，这个时候可以给相应的查询字段加上索引。

执行计划：

```
-- 实际SQL，查找id为6的账户
select * from account where id = 6;

-- 查看SQL是否使用索引，前面加上explain即可
explain select * from account where id = 6;
```

执行结果：

信息 结果 1 剖析 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	account	(Null)	const	PRIMARY	PRIMA	4	const	1	100.00	(Null)

- id：选择标识符
- select_type：表示查询的类型。
- table：输出结果集的表
- partitions：匹配的分区
- type：表示表的连接类型
- possible_keys：表示查询时，可能使用的索引
- key：表示实际使用的索引
- key_len：索引字段的长度
- ref：列与索引的比较
- rows：扫描出的行数(估算的行数)
- filtered：按表条件过滤的行百分比
- Extra：执行情况的描述和说明

重点关注 type、possible_keys、key以及rows这几个字段

12 写缓冲（Change buffer）了解吗?

我们知道，MySQL的数据是存储在页上的。那么当一个页的数据不再内存中的时候，发生了写操作，这个时候该如何处理呢?

正常的思路是把这个页的数据读到内存中，然后对其进行写操作，然后在把这些数据刷回磁盘。但是由于数据库预读的特点，这一页的数据都会被读取出来。但是对于写多读少的业务，每一次操作都需要去读磁盘，这显然会影响到性能。

innodb对这里的写入操作进行了优化：写操作并不会立即写入到磁盘，而是只记录缓冲变更，等未来数据从磁盘真正被读出来的时候，才会将缓冲变更合并到缓冲池里，这种做法可以有效减少写操作时候的磁盘读取，提高数据库性能，这里对于数据变更的缓存区，就是写缓冲。

写缓冲等于是将写操作没有命中缓存的情况转化为了命中缓存的情况，由于redo log依然是按策略落盘的，可以理解为数据是安全的，而写缓存的数据也不是长期保留在缓存里的，数据库有策略定期将写缓存中的数据写入磁盘的操作，这里的写缓冲写入策略可以由数据库参数控制（有的采取定时写入，有的采取满则写入，也有的采取闲时写入）

注意：当数据库全都是唯一索引的时候，写缓冲是没有办法工作的，因为不将所有数据都读出来，没有办法判断这次写操作是否合法，是否符合唯一索引的条件

13 自增表 7条数据,删除2条,重启数据库,再插入一条,id是几?

id=6。

mysql数据库的的auto_increment值是保存在内存中的，innoDB引擎的表的auto_increment在数据库服务停止时并不会做持久化操作，Mysql会在下次数据库重启的时候，直接获取表中最大的id，然后把自增值设置为 max(id) + 1。

Redis

3.1 了解Redis的持久化策略吗?

Redis的持久化方式有俩种，持久化策略有4种：

- RDB（数据快照模式），定期存储，保存的是数据本身，存储文件是紧凑的
- AOF（追加模式），每次修改数据时，同步到硬盘(写操作日志)，保存的是数据的变更记录
- 如果只希望数据保存在内存中的话，俩种策略都可以关闭
- 也可以同时开启俩种策略，当Redis重启时，AOF文件会用于重建原始数据

3.2 Redis的数据类型有哪几种？知道setnx和setex是干嘛的吗？

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

- setnx: 设置 key对应的值为 string类型的 value。 如果key 已经存在，返回 0，nx 是not exist 的意思，可用于分布式锁。例如：setnx key value
- setex: 设置key 对应的值为 string 类型的 value，并指定此键值对应的有效期。

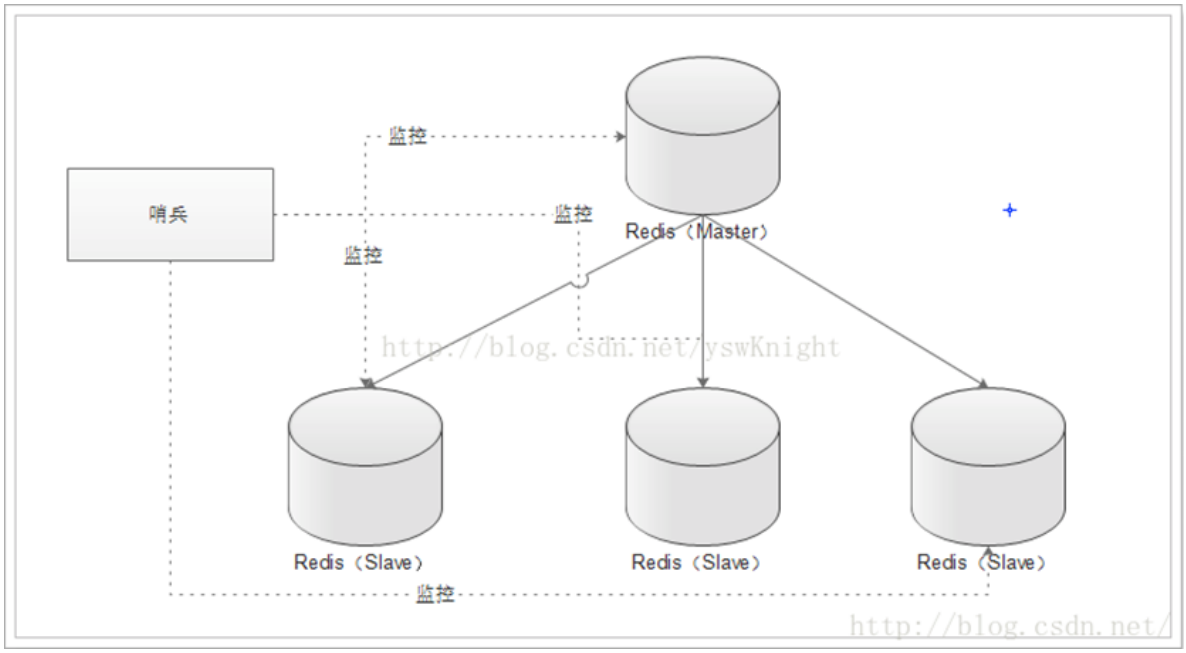
  例如：setex key seconds value

3.3 说一下redis的集群模式、哨兵模式

Redis的集群模式：

- 主从复制模式：

  Slave从节点服务启动并连接到Master之后，它将主动发送一个SYNC命令。Master服务主节点收到同步命令后将启动后台存盘进程，同时收集所有接收到的用于修改数据集的命令，在后台进程执行完毕后，Master将传送整个数据库文件到Slave，以完成一次完全同步。而Slave从节点服务在接收到数据库文件数据之后将其存盘并加载到内存中。此后，Master主节点继续将所有已经收集到的修改命令，和新的修改命令依次传送给Slaves，Slave将在本次执行这些数据修改命令，从而达到最终的数据同步。
- 哨兵模式：



哨兵(sentinel) 是一个分布式系统,你可以在一个架构中运行多个哨兵(sentinel) 进程,这些进程使用流言协议(gossipprotocols)来接收关于Master是否下线的信息,并使用投票协议(agreement protocols)来决定是否执行自动故障迁移,以及选择哪个Slave作为新的Master。

每个哨兵(sentinel) 会向其它哨兵(sentinel)、master、slave定时发送消息,以确认对方是否”活”着,如果发现对方在指定时间(可配置)内未回应,则暂时认为对方已挂(所谓的”主观认为宕机” Subjective Down,简称sdown).

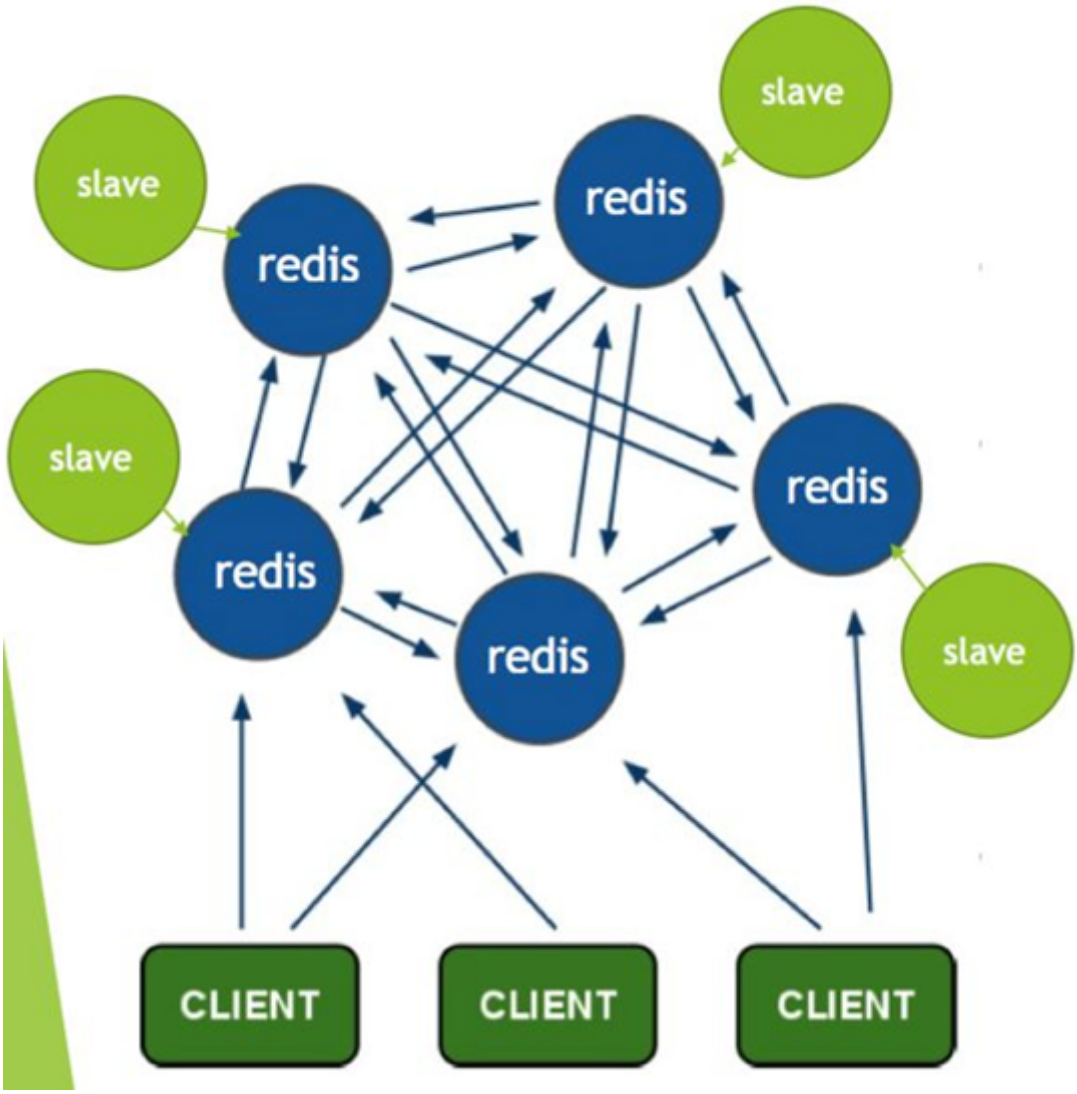
若“哨兵群”中的多数sentinel,都报告某一master没响应,系统才认为该master"彻底死亡"(即:客观上的真正down机,Objective Down,简称odown),通过一定的vote算法,从剩下的slave节点中,选一台提升为master,然后自动修改相关配置。

虽然哨兵(sentinel) 释出为一个单独的可执行文件 redis-sentinel ,但实际上它只是一个运行在特殊模式下的 Redis 服务器，你可以在启动一个普通 Redis 服务器时通过给定 --sentinel 选项来启动哨兵(sentinel)。

- Cluster集群模式：

自动将数据进行分片，每个节点上放一部分数据，如何放的呢?

Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态，每个节点都和其他所有节点连接，其结构图如如下：



结构特点：

1， 所有的redis节点彼此互联（PING-PONG机制），内部使用二进制协议优化传输速度和带宽。

2， 节点的fail是通过集群中超过半数的节点检测失效时才失效。

3， 客户端与redis节点直连，中间不需要proxy层，客户端不需要连接所有集群的所有节点，连接集群中任意一个可用节点即可。

4， Redis-cluster把所有的物理节点映射到16384个哈希槽（hash slot）上，不一定是平均分配，cluster负责维护node<->slot<->value。

5， 当有数据需要存储的时候，redis通过CRC16算法计算出key的值，然后把这个值对16384取余，得到的结果在哪个节点负责的范围内就会把值存入哪个节点。

6， 如上图，每个master都是一个节点，而slave是该master的备份，当redis的一个master节点挂了，slave会顶替掉master，如何一个节点的master和slave都挂了，那么则认为该redis集群挂了，因为有一部分数据丢失了，也有一部分数据永远也存取不了了。

3.4 说下Redis的缓存雪崩，缓存穿透？如何保证redis缓存里面的都是热点数据?

- 缓存穿透

  指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。那么如果查询的这个数据一直在缓存中不存在，那么就会一直查询数据库，如果这个查询量很大的话，就有可能击穿我们的数据库，这个就叫缓存穿透。如何避免？当查询数据为空的时候，也可以在缓存中打上一个标记，这样下次就会去查缓存并且得知该数据为空了。

- 缓存雪崩

  指在某一段时间，缓存集中失效。假如大量缓存在某一个时刻同时失效，这个时候就会导致大量的请求去请求数据库，可能会引起数据库瘫痪。举例：马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

如何避免？设置热点缓存的时候过期时间分散设置。

如何保证redis里面都是热点数据?

当redis使用的内存超过了设置的最大内存时，会触发redis的key淘汰机制，在redis 3.0中有6种淘汰策略：

- noeviction: 不删除策略。当达到最大内存限制时, 如果需要使用更多内存,则直接返回错误信息。（redis默认淘汰策略）
- allkeys-lru: 在所有key中优先删除最近最少使用(less recently used ,LRU) 的 key。
- allkeys-random: 在所有key中随机删除一部分 key。
- volatile-lru: 在设置了超时时间（expire ）的key中优先删除最近最少使用(less recently used ,LRU) 的 key。
- volatile-random: 在设置了超时时间（expire）的key中随机删除一部分 key。
- volatile-ttl: 在设置了超时时间（expire ）的key中优先删除剩余时间(time to live,TTL) 短的key。

我们可以把redis的淘汰策略设置为LRU策略（或者是别的策略，视具体业务场景而定）

第四章 JAVAEE

1.Tomcat的类加载器是如何设计的？为什么要打破双亲委派模型?

回答这个问题之前，我们首先要将类加载器的特点描述清楚。类加载器的特点是采用双亲委派模型，整个过程如下：

如果一个类加载器收到了加载类的请求，它首先不会自己尝试去加载该类，而是把这个请求委派给父类加载器去完成。只有当父类加载器无法完成该请求时，子类加载器才会尝试去加载。

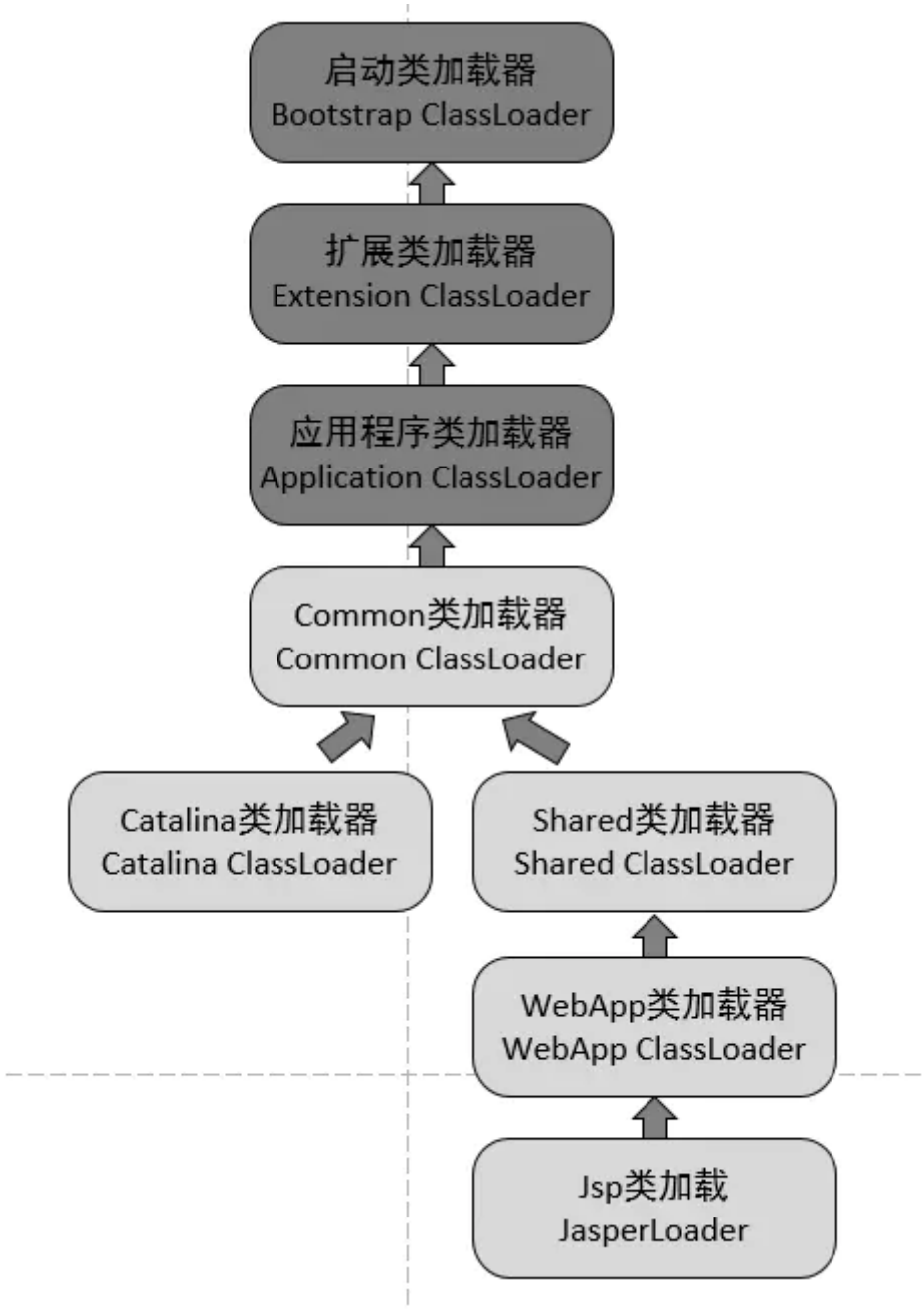
那么tomcat需要去解决什么样的问题呢?

1.一个tomcat服务器中可能需要部署多个应用，不同的应用程序可能需要使用到同一个第三方类库的不同版本。应当　　保障在每一个应用中类库都是独立的。

2.tomcat本身也有需要依赖的类库，不能和应用依赖的类库混淆。

那么tomcat使用默认类加载器可以吗?

如果使用默认类加载器根据双亲委派机制，是不可能实现加载两个相同类库的不同版本的，默认加载机制是只要全类名相同，那么在内存中有且只有一份。**综上所述，使用默认类加载器的双亲委派机制在tomcat中是不合适的，所以需要打破双亲委派机制。**



CommonClassLoader:tomcat最基本的类加载器，加载/common/*路径下的class，可以被tomcat服务器以及各个web应用访问到

CatalinaClassLoader:tomcat服务器私有的类加载器，负责加载/server/*路径下的class，对各个web应用不可见

SharedClassLoader:各个web应用所共享的类加载器，负责加载/shared/*路径下的class，对于tomcat服务器来说是不可见的

WebAppClassLoader：各个web应用私有的类加载器，加载每个应用WEB-INF/classes以及WEB-INF/lib路径下的class，只对当前web应用是可见的

2.EE规范中的Context域、Session域、Request域之间的区别

这三个域的行为特性都是针对的是单体应用而言的，如果是集群或者分布式应用，那么将有可能不符合下面的描述。

Context域，是指在每个应用中有且只有唯一的一个ServletContext对象，该对象可以作为一个运行时共享数据的场所。当前应用下的任何Servlet都可以在当前域中进行数据的共享，这样就可以实现多个servlet共享数据的问题。Context域不区分用户，无论任何用户访问都是共享同一份数据，如果希望保存用户相关数据，比如登录状态等，则不可以使用Context域

Session域，一般情况下是在客户端第一次访问服务器时，执行到reqeust.getSession()时，服务器会给当前客户端开辟一块内存空间，也就是生成一个HttpSession对象，同时将session对象的id通过set-Cookie响应头发送给客户端；客户端下次访问时会将该id再次通过Cookie携带回来，通过这种方式就可以实现客户端每次访问服务器时，无论访问的是哪个servlet，那么都是得到的同一个session对象，可以实现数据的共享。

Request域，三个域中最小的一个。每发送一次请求，就会生成一个request对象，只有得到相同request对象引用的组件才可以进行数据共享。也就是只有转发的两个组件之间可以进行数据共享。

3.简述Cookie和Session的区别

Cookie和Session同属于会话技术。会话技术的出现主要是为了弥补HTTP协议无状态性这一不足。因为对于服务器来说，是无法通过HTTP请求报文区分出不同用户，但是又有这样的需求，需要服务器给客户端保存对应的信息，也就出现了会话技术。

Cookie是客户端技术，数据的产生在服务器，数据的保存是在客户端。过程是：服务器生成数据之后，通过set-Cookie：key=value响应给客户端，客户端下次访问服务器时，再通过Cookie：key=value请求头携带给服务器，服务器通过取出cookie的值，就可以知道客户端的身份。

Session是服务器技术，数据的产生以及保存都是在服务器。但是Session底层依赖于Cookie。过程是：当客户端访问服务器时，如果没有携带Cookie:JSESSIONID=xxx，那么执行到request.getSession()时便会生成一个新的session对象，并且在响应的时候通过set-Cookie：JSESSIONID=xxx返回给客户端；客户端下次访问的时候通过取出里面的JSESSIONID的值就可以拿到对应的session对象。

第五章 常用框架

1.谈一下IOC

IOC是Inverse of Control的简称，中文叫控制反转。指的是将实例的生成权由应用程序反转给Spring容器，由Spring容器来管理实例，从而降低应用程序各个实例之间的耦合度。

容器中也可以维护组件之间的依赖关系，在容器中维护组件之间的依赖关系，降低了耦合度

2.谈一下DI

DI是Dependency Injection的简称，中文叫依赖注入。依赖注入是控制反转的延续，当容器创建组件的时候，同时给这个组件注入他需要的依赖，这个依赖可以是容器中的其他组件，也可以是一些值。

依赖注入更多的话大家可以认为是维护组件之间的依赖关系，避免你在使用的时候再来进行维护。

3.谈一下AOP

AOP是Aspect oriented Programming的简称，中文叫面向切面编程。

AOP通过提供另一种程序结构思维方式来补充面向对象的编程 （OOP）。OOP的模块化单元是类，而AOP的模块化单元是切面。切面能够让你对于模块的关注，跨域多种类型和对象。

那么我们怎么去理解AOP，AOP的核心事情就是在容器这个大前提下，对容器中的组件做增强，而增强的方式是动态代理（如果有接口的实现是JDK动态代理，如果没有接口的实现则是CGlib动态代理）

AOP提供的最重要的功能是声明式服务，最具有代表性的是声明式事务管理。

4.BeanFactory和FactoryBean之间的联系和区别

BeanFactory是Spring中的容器接口，承担的是容器的功能，生成并管理组件，BeanFactory生成的全部组件。

FactoryBean是Spring中用来注册特定组件的方式，该接口中提供了一个getObject方法，而这个getObject方法返回值就是你注册到容器中的组件

5.Spring的生命周期

是从容器初始化的时候，组件开始执行生命周期；到容器关闭的时候组件执行对应的生命周期方法。

注意BeanPostProcessor是所有的组件执行生命周期的方法，利用它的特性可以做到狸猫换太子，批量对组件进行增强，或者做一些通用性的业务处理。

注意scope对生命周期的影响

- 单例singleton是从容器初始化的时候开始生命周期的，并且会执行到destroy方法
- 原型prototype是获得容器中组件的时候开始生命周期的，每次获得组件都会执行生命周期方法，并且不会执行到destroy方法

 生命周期

6.事务的传播行为

多个数据库操作之间如何共享事务，发生于不同的服务之间产生调用关系时，比如service1中的methodA调用了service2中的methodB这时候产生事务的传播行为。主要是发生异常时谁提交谁回滚。

> REQUIRED 默认的传播行为 -- 同生共死
>
> 要么一起提交要么一起回滚

> REQUIRES_NEW -- 自私型
>
> 别人不能影响他，但是他可以影响别人。通常内部方法更重要。

> NESTED -- 无私型
>
> 别人能影响他，但是他不会影响别人。
>
> 通常外围方法比内部更重要，比如注册功能调用发放优惠券功能

7.SpringMVC的核心流程

- 首先请求全部都由DispatcherServlet来进行处理（底层就是一个HttpServlet），执行处理最终会执行到其**doDispatch方法**（核心）
- 通过HandlerMapping建立映射关系，主要是建立请求URL和Handler方法之间的映射关系，返回一个HandlerExecutionChain，其中已经包含Handler和HandlerInterceptor的list
- 通过HandlerAdapter执行Handler方法（通过反射执行的）
- 执行Handler方法的返回值为ModelAndView，另外呢我们在Handler方法中可以响应JSON，响应JSON是我们后续主要的处理方式

核心流程图（这个流程是理解SpringMVC的关键）

 SpringMVC核心流程

8.SpringBoot约定大于配置的原理

约定大于配置首先要清楚一点：**并不是**约定项和配置项同时存在的使用，约定项的优先级更高；而是在没有自定义配置项时，提供默认的配置项（约定）

约定大于配置在SpringBoot中有大量的体现：比如引入一些依赖的时候没有写版本号，提供默认的版本号；比如注册默认的组件。

约定大于配置让程序开发人员能够以极少的配置快速启动一个Spring应用，我们在实际开发过程中，通常是整合了SpringMVC和MyBatis，而这两个框架的整合，我们是以极少的配置来完成的

组件的约定大于配置主要是因为SpringBoot引入的依赖中包含了大量的自动配置类，而自动配置类的生效是有条件的，而这些条件主要是@ConditionalOnXXX和@ConditionalOnMissingXXX，根据特定的条件决定其他组件是否生效，其中最有代表性的是@ConditionalOnMissingBean，如果容器中没有这个组件，那么就生效，而和它通常一起出现的是@Bean注解，生效就会导致JavaConfig向容器中注册了默认的组件。

自动配置类可以参考starter依赖下的autoconfigure依赖中的/META-INF/spring.factories，在这个文件中包含了key为EnableAutoConfiguration对应的字符串List，而这个List就是SpringBoot应用加载的自动配置类的全限定类名的List。

第六章 微服务

1. 谈一下Dubbo的异步调用

- 用法：

  首先开启异步调用配置，假如是在SSM项目中，应该添加消费`<dubbo:method>通过 async="true"`标识。假如是在SpringBoot项目中，应该添加注解`@Service(interfaceClass = UserApi.class,async = true)`

- 调用用户服务的接口的UserAPI接口的时候，会异步调用，我们直接调用并不会立马给返回结果，需要去主动获取，代码如下：

```
//此时userDO为空，因为是异步调用，不会立马去返回
UserDO userDO = UserAPI.register(UserVO);

Future<UserDO> future = RpcContext.getContext().getFuture();
//这个时候才会真正的取到userDO对象
UserDO userDO = future.get();
```

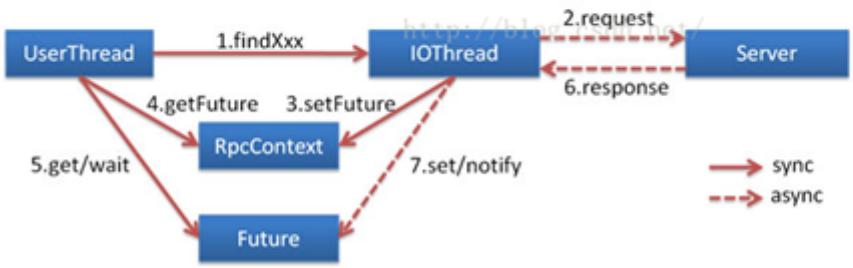
流程图：

异步调用

(+) (#)

基于NIO的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。

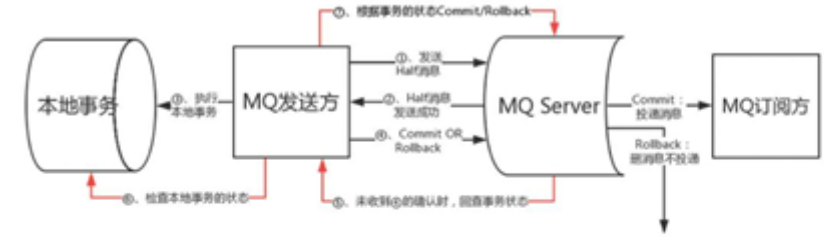
2.0.6及其以上版本支持



2. 消息队列都有哪些？你们项目用的RocketMQ对比其他消息队列有什么优势？

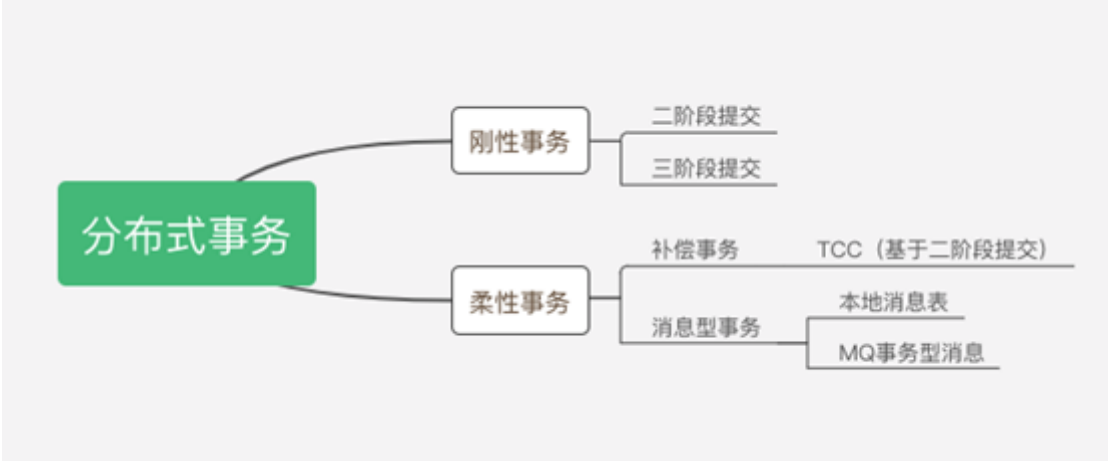
特性	ActiveMQ	RabbitMQ	Rocket MQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低;管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

3. RocketMQ如何实现分布式事务？



1，事务发起方发送prepare消息到MQ
2，消息发送成功后执行本地事务
3，根据本地事务执行结果返回commit或者是rollback
4，如果消息时rollback，MQ将删除该prepare消息不进行下发，如果是commit消息，mq将会把这个消息发送到consumer端
5，执行本地事务的过程中，执行端挂掉，或者超时，MQ将会不停的询问其同组的其他producer来获取状态
6，Consumer端的消费成功机制有MQ保证

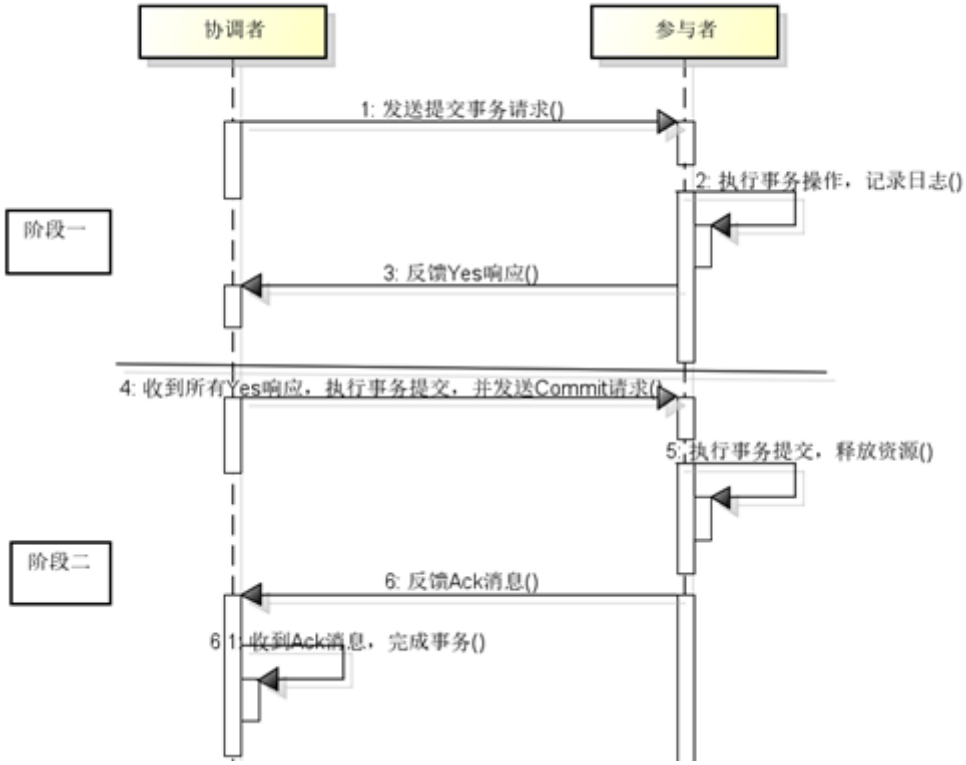
4. 分布式事务有哪几种?



5. 介绍一下二阶段提交事务

二阶段提交把事务分为两个阶段，并引入了事务管理器

- 阶段一：
  - 提交事务请求：协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应。
  - 执行事务：各个参与者节点执行事务操作。并将Undo和Redo信息记入事务日志中。
  - 各参与者向协调者反馈事务询问的响应：如果参与者成功执行了事务操作，那么就反馈给协调者yes响应，表示事务可以执行，如果参与者没有成功执行事务，那么就反馈给协调者no响应，表示事务不可以执行。
- 阶段二：
  - 执行事务提交：如果事务管理器从所有事务参与者处获得的响应都为Yes，那么就会执行事务提交，协调者向所有参与者节点发出commit请求。
  - 参与者提交事务：参与者接收到commit请求后，会正式执行事务提交的操作，并在完成提交之后释放整个事务执行期间占有的事务资源，并向协调者发送ack消息
  - 完成事务：协调者接收到所有参与者反馈的Ack消息后，完成事务。
  - 如果收到No响应，或者是等待操作，没有收到所有响应，进入事务中断，然后事务协调者发送Rollback请求，请求资源回滚，释放资源



6. 介绍一下分布式锁

单机环境下，我们用Synchronized关键字来修饰方法或者是修饰代码块，以此来保证方法内或者是代码块内的代码在同一时间只能被一个线程执行。但是如果到了分布式环境中，Synchronized就不好使了，因为他是JVM提供的关键字，只能在一台JVM中保证同一时间只有一个线程执行，那么假如现在有多台JVM了（项目有多个节点了），那么就需要分布式锁来保证同一时间，在多个节点内，只有一个线程在执行该代码。分布式锁有以下三种实现方式

- 基于数据库：

  数据库自带的通过唯一性索引或者主键索引，通过它的唯一性，当某个节点来使用某个方法时就在数据库中添加固定的值，这样别的节点来使用这个方法就会因为不能添加这个唯一字段而失败。这样就实现了锁。一个节点执行完毕后就删除这一行数据，别的节点就又能访问这个方法了，这就是释放锁。但是一方面，数据库本身性能就低，而且这种形式非常容易死锁，虽然可以各种优化，但还是不推荐使用。数据库的乐观锁也是一个道理，一般不用

- Redis分布式锁

  SETNX关键字，设置参数如果存在返回0，如果不存在返回value和1

  expire关键字，为key设置过期时间，解决死锁。

  delete关键字，删除key，释放锁

  实现思想：

  （1）获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。

  （2）获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

  （3）释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

- zookeeper实现分布式锁

  ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的步骤如下：

  （1）创建一个目录mylock；

（2）线程A想获取锁就在mylock目录下创建临时顺序节点；

（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；

（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；

（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

速度没有redis快，但是能够解决死锁问题，可用性高于redis

7. Zookeeper的作用？Zookeeper在某一个瞬间挂了Dubbo服务还能调用吗？

Zookeeper作为注册中心为分布式环境下各个微服务之间的调用提供协调工作，当Zookeeper在某个瞬间挂了Dubbo还是一样可以调用，因为启动dubbo时，消费者会从zk拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用

8. Zookeeper集群最少需要配置几台机器？为什么？

最少：3台

Zookeeper集群的写操作，由leader节点负责，它会把通知所有节进行写入操作，只有收到半数以上节点的成功反馈，才算成功。如果是部署2个节点的话，那就必须都成功。

Zookeeper的选举策略也是需要半数以上的节点同意才能当选leader，如果是偶数节点可能导致票数相同的情况

Zookeeper关于leader的选举机制，主要提供了三种方式：

- LeaderElection
- AuthFastLeaderElection
- FastLeaderElection

默认的算法是FastLeaderElection

在zookeeper的选举过程中，为了保证选举过程最后能选出leader，就一定不能出现两台机器得票相同的僵局，所以一般的，要求zk集群的server数量一定要是奇数，也就是2n+1台，并且，如果集群出现问题，其中存活的机器必须大于n+1台，否则leader无法获得多数server的支持，系统就自动挂掉。所以一般是3个或者3个以上节点。

9. 说一下消息队列消息的类型有哪些？

- 普通消息：普通消息也叫做无序消息，简单来说就是没有顺序的消息。因为不需要保证消息的顺序，所以消息可以大规模的并发的发生和消费，吞吐量很高，适合大部分场景
- 有序消息：有序消息就是按照一定的先后顺序的消息类型。

  有序消息还可以进一步分为：全局有序消息（1个MessageQueue）、局部有序消息（多个MessageQueue）
- 延时消息：简单来说就是当producer将消息发送到broker之后，会延时一定时间后才投递给consumer进行消费。
- 事务消息：本地事务和事务消息的投递保持一致性

10. 说一下，如何实现顺序消息

消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。

顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

在MQ的模型中，顺序需要由3个阶段去保障：

- 消息被发送时保持顺序
- 消息被存储时保持和发送的顺序一致
- 消息被消费时保持和存储的顺序一致

发送时保持顺序意味着对于有顺序要求的消息：

- 用户应该在同一个线程中采用同步的方式发送。
- 存储保持和发送的顺序一致则要求在同一线程中被发送出来的消息A和B，存储时在空间上A一定在B之前。

```
/*
*  要保证消息的有序性，即需要把有先后顺序的消息，发送到同一个MessageQueue中去，如何实现呢？
*  a. 在发送消息的时候，就需要给send方法，传递一个MessageQueueSelector，选择消息发送的具体是Topic中的那个MessageQueue
*  b. MessageQueueSelector中有一个方法select方法，该方法中有一个参数List<MessageQueue> mqs，该参数代表某个Topic对应的所有的MessageQueue
*  c. 在select方法中，实现消息的路由，即决定消息发送到哪个MessageQueue中，实例中就是一种方式，利用orderId对MessageQueue数量取余，这样一来
      同一个id的消息，就会被发送到同一个MessageQueue中去了
*/
SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
               @Override
               public MessageQueue select(List<MessageQueue> mqs, Message msg,
Object arg) {
                   Long id = (Long) arg;  //根据订单id选择发送queue
                   long index = id % mqs.size();
                   return mqs.get((int) index);
               }
 }, orderList.get(i).getOrderId());//订单id
```

- 而消费保持和存储一致则要求消息A、B到达Consumer之后必须按照先A后B的顺序被处理。

```
/*
*  这里我们只需要保证，同一个MessageQueue中的消息，是前一个消息消费完了，才消费后一个消息，此时我们只需要设置MessageListenerOrderly
*  这种类型的监听器，在监听器内实现消费逻辑即可
*/
consumer.registerMessageListener(new MessageListenerOrderly() {
           @Override
           public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeOrderlyContext context) {
               // 消费逻辑
               return ConsumeOrderlyStatus.SUCCESS;
           }
       });
```

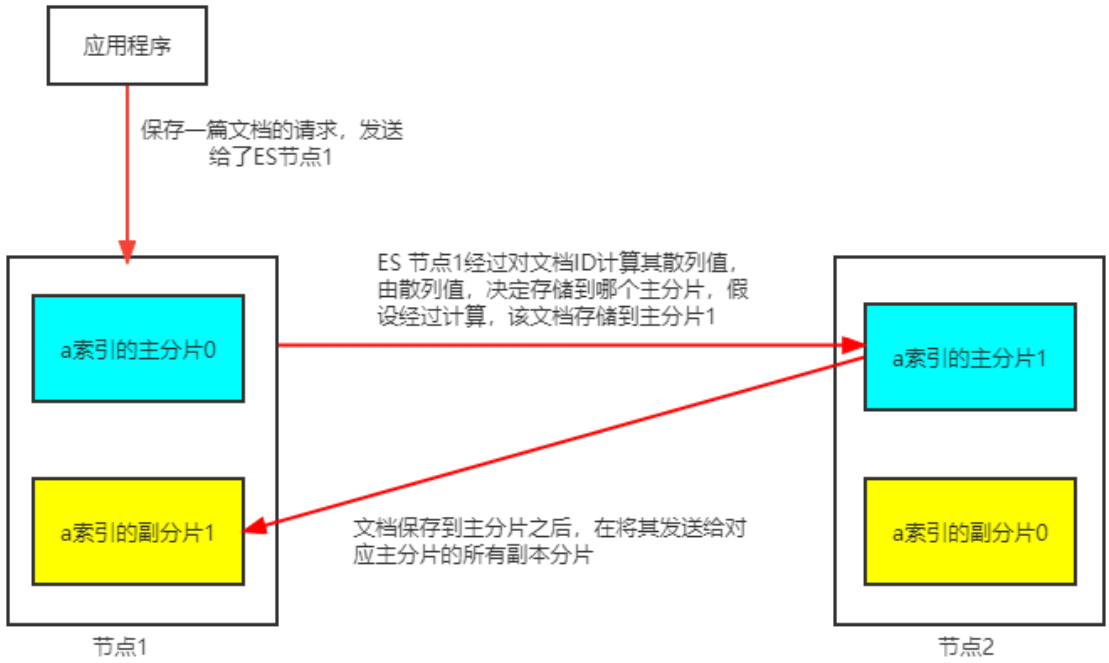
11 在Elastic Search集群中，数据如何存储，和检索?

- 无论是单机还是集群模式，ES中的索引数据都是以分片的形式存在的，一个索引的一个分片中只存储该索引中的一部分数据，即一个索引中的文档数据，被存储在其所属的若干个分片中，每个分片只存储索引部分数据
- 分片又被分成了主分片和副本分片，一个索引中的文档到底被被分成几部分来存储，主要看索引到底包含多少个主分片，有多少个主分片，相当于索引数据数据就被分成几部分，分别存储在不同的主分片中，而副本分片主要是作为主分片的数据副本而存在，每个主分片都可以有对应的副本分片。
- 在创建索引的时候就可以定义，索引的主分片数量，以及每个主分片对应的副本分片的数量。
- 所以，在ES集群中，一个索引的多个分片数据，就保存在不同的ES服务器实例或者说不同的ES Node上，一个单机版的ES服务器，也可以看做是只包含一个ES Node的ES集群，所以一个ES Node可以包含索引的多个分片数据。
- 但是切记一点，ES 不允许将一个主分片和它对应的副本分片存储在同一个ES Node中。这主要是为了防止，一个ES Node 宕机导致，某分片数据全部丢失(主分片和该主分片对应的副本分片的数据全部丢失)。这是因为如果每个主分片至少有一个副本分片，那么即使该主分片所在的ES Node宕机也没关系，ES会自动将其副本分片变为主分片，保证数据的正常访问。
- 一个ES的索引由**一个或多个**主分片以及**零个或多个**副本分片构成，即ES不强制要求主分片一定有对应的副本分片

基于以上的基础知识，我们就可以来看下在ES集群中，我们就可以来研究集群中数据的存储和检索了。

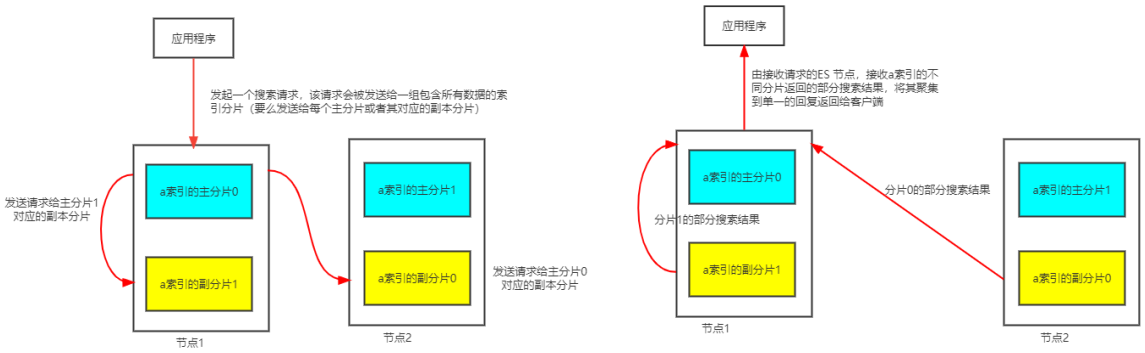
我们先来看看文档数据的存储：

- 当存储或一篇文档的时候，ES首先根据文档ID的散列值，选择一个主分片，并将该文档发送到该主分片保存。
- 然后，该文档被发送到主分片对应的所有的副本分片进行保存，这使得副本分片和主分片之间保持数据同步
- 数据同步使得副本分片可以服务与搜索请求，并在原有主分片无法访问时自动升级为主分片



再来看看在集群中搜索文档数据的过程，当在索引中搜索的时候，Elastic Search会在索引的所有分片中进行查找，这些分片可以是主分片，也可以是副本分片，原因是主分片和副本分片通常包含一样的文档。ES在索引的主分片和副本分片中进行搜索请求的负载均衡，使得副本分片对于搜索的性能和容错都有帮助。下面看看具体的搜索过程:

- 搜索请求首先被一个ES Node接收，并将请求转发到一组包含所有数据的索引分片
- 在选择转发请求的分片时，使用轮训算法选择可用分片(这里轮训的是某主分片和其对应的副本分片，并且对于索引中的每一个主分片与其对应的副本分片都会有这样的轮训选择过程)，并将搜索请求转发到所有选中的分片，
- 然后，ES将从这些接收到请求的分片收集搜索的结果，将其聚集到单一的回复，然后将回复返回给客户端



在保存一篇文档的时候，我们讲解了如何决定一篇文档所在的分片的，这一过程我们称为文档路由。当ES散列文档ID时就会发生文档的路由，来决定文档应该索引到哪个分片中

同时，还需要注意一点，索引的主分片数量在创建索引的时候指定，一旦索引创建成功，其主分片数量就无法在被修改，只能修改主分片对应的副本分片的数量

12 如何解决ES中的深度分页问题

我们在平常使用ElasticSearch构建查询条件的时候一般用的都是from+size的方式进行分页查询，但是如果我们的页数太深/页面大小太大(from*size)>10000就会引发一个错误，我们将会得到一个错误


```
{
  ▼ "error": {
    ▼ "root_cause": [
        {
            "type": "query_phase_execution_exception",
            "reason": "Result window is too large, from + size must be less than or equal to: [10000] but was [10009]. See the scroll api for a more efficient way to reque
            large data sets. This limit can be set by changing the [index.max_result_window] index level setting."
        }
    ],
    "type": "search_phase_execution_exception",
    "reason": "all shards failed",
    "phase": "query",
    "grouped": true,
    ▼ "failed_shards": [
        {
            "shard": 0,
            "index": "wechat_bot_chat_msg",
            "node": "ulw84A5wSyWBwIPy-P3QfQ",
          ▼ "reason": {
                "type": "query_phase_execution_exception",
                "reason": "Result window is too large, from + size must be less than or equal to: [10000] but was [10009]. See the scroll api for a more efficient way to
                request large data sets. This limit can be set by changing the [index.max_result_window] index level setting."
            }
        }
    ]
  },
  "status": 500
}
```


为什么会发生这样的错误呢?

- 这就和ES获取分页数据的方式有关系了，如果我们采用from + size的方式获取某一页的数据，那么ES会从**每一个分片**(分片对应的英文Shard)，获取from + size条数据，然后汇总，排序，取汇总结果中从from开始的size条数据返回给我们(其他查询到的结果数据就丢弃掉了)。
- 这意味着假设在一个有 5 个主分片的索引中搜索。当我们请求结果的第一页（结果从 1 到 10 ），每一个分片产生前 10 的结果，ES需要对 50 个结果排序得到全部结果的前 10 个。
- 现在假设我们请求第 1000 页—结果从 10001 到 10010 。所有都以相同的方式工作除了每个分片不得不产生前10010个结果以外。然后ES还需要对全部 50050 个结果排序，最后丢弃掉这些结果中的 50040 个结果。
- 对结果排序的成本随分页的深度成指数上升。当获取的页数大到一定程度，那么获取一页数据，就可以直接让ES的内存爆炸

如何解决这一问题呢？——> 三种方式

- Scroll方式(滚动查询)

为了满足深度分页的场景，es 提供了 scroll(滚动查询) 的方式进行分页读取。原理上是对某次查询生成一个游标 scroll_id ， 后续的查询只需要根据这个游标去取数据，直到结果集中返回的 hits 字段为空，就表示遍历结束。scroll_id 的生成可以理解为建立了一个临时的历史快照，在此之后的增删改查等操作不会影响到这个快照的结果。可以把 scroll 理解为关系型数据库里的 cursor，因此，scroll 并不适合用来做实时搜索，而更适用于后台批处理任务，比如群发。

可以把 scroll 分为初始化和遍历两步，初始化时将所有符合搜索条件的搜索结果缓存起来，可以想象成快照，在遍历时，从这个快照里取数据，也就是说，在初始化后对索引插入、删除、更新数据都不会影响遍历结果,并且维护这个快照需要占用很多资源。

使用scroll，并不适合用来做实时搜索，因为是基于开启滚动搜索时创建的数据快照来进行搜索的, 这种方式而更适用于后台批处理任务

```
# 先查询，在返回的响应中会包含scrollid
GET test_dev/_search?scroll=5m
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "age": 28
          }
        }
      ]
    }
  },
  "size": 10,
  "from": 0,
  "sort": [
    {
      "timestamp": {
        "order": "desc"
      },
      "_id": {
        "order": "desc"
      }
    }
  ]
}
```

```
{
  "_scroll_id":
"DnF1ZXJ5VGhlbkZldGNoBQAAAAAABn5FjRXNkZmY3ZmVGJPVXJ1NWs2MUh5RGcAAAAAAAAZ_BYOVzZ
GZmN2ZlRiT1VydTVrNjFIeURnAAAAAAAAGfgWNFc2RmZjdmZUYk9VcnU1azYxSHlEZwAAAAAABn6FjR
XNkZmY3ZmVGJPVXJ1NWs2MUh5RGcAAAAAAAAZ-xYOVzZGZmN2ZlRiT1VydTVrNjFIeURn",
  "took": 0,
  "timed_out": false,
   ...

}
```

之后对于相同条件的分页数据查询，只需要携带scrollid即可

```
POST /_search/scroll
{
  "scroll":"3m",
  "scroll_id":
"DnF1ZXJ5VGhlbkZldGNoBQAAAAAAABOmFjRXNkZmY3ZmVGJPVXJ1NWs2MUh5RGcAAAAAAAAdKBYOVzZ
GZmN2ZlRiT1VydTVrNjFIeURnAAAAAAAAHScWNFc2RmZjdmZUYk9VcnU1azYxSHlEZwAAAAAAAB0qFjR
XNkZmY3ZmVGJPVXJ1NWs2MUh5RGcAAAAAAAAdKRYOVzZGZmN2ZlRiT1VydTVrNjFIeURn"
}
```

- scroll-scan 高效滚动

scroll 能比较好的地返回排序的结果。但是，按照默认设定排序结果仍然需要代价。一般来说，你仅仅想要找到结果，不关心顺序。你可以通过组合 scroll 和 scan 来关闭任何打分或者排序，以最高效的方式返回结果。你需要做的就是将 search_type=scan 加入到查询的字符串中：

```
POST /my_index/_search?scroll=1m&search_type=scan
{
   "query": {
        "match" : {
            "cityName" : "杭州"
        }
    }
}
```

扫描式的滚动请求和标准的滚动请求有四处不同：

a. 不算分，关闭排序。结果会按照在索引中出现的顺序返回;
b. 不支持聚合;
c. 初始 search 请求的响应不会在 hits 数组中包含任何结果。第一批结果就会按照第一个 scroll 请求返回。(请求分页数据　时和普通的滚动 查询方式一样，每次携带scrolllid请求下一页数据)
d. 可以通过参数 size 控制每个分片上而非每个请求的结果数目，所以 size 为 10 的情况下，如果命中了 5 个分片，那么 每个scroll 请求最多会返回 50 个结果。

- search_after 深分页

scroll 的方式，官方的建议不用于实时的请求（一般用于数据导出），因为每一个scroll_id 不仅会占用大量的资源，而且会生成历史快照，对于数据的变更不会反映到快照上。

search_after 分页的方式是根据上一页的最后一条数据来确定下一页的位置，同时在分页请求的过程中，如果有索引数据的增删改查，这些变更也会实时的反映到游标上。但是需要注意，因为每一页的数据依赖于上一页最后一条数据，所以无法跳页请求。

为了找到每一页最后一条数据，每个文档必须有一个全局唯一值，官方推荐使用 _uid 作为全局唯一值，其实使用业务层的 id 也可以。

```
# 首次查询
POST /my_index/_search
{
    "size":2,
    "query": {
        "match" : {
            "cityName" : "杭州"
        }
    },
```

```
    "sort": [
        {"updateTime": "desc"}
    ]
}

# 查询返回的结果
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 534,
    "max_score": null,
    "hits": [
      {
        "_index": "my_index",
        "_type": "my_type",
        "_id": "2019061010810316",
        "_score": null,
        "_source": {
        }
```

```
# 二次查询
POST /my_index/my_type/_search
{
    "size":2,
    "query": {
        "match" : {
            "cityName" : "杭州"
        }
    },
    # 这里携带上一次查询到的最后一条数据的id
    "search_after": [1560137241000],
    "sort": [
        {"updateTime": "desc"}
    ]
}
```

最后总结下，scroll，scroll-scan，search_after都可以解决深度分页问题，但是scroll不够实时，scroll-scan高效不支持对查询结果的排序和聚合，search_after是官方推荐的方式，但是用起来麻烦，因为每次得记录上一次查询的最后一条记录的唯一标识，同时三种方式**都不支持跳页**，但是没关系，技术层面解决不了的问题，可以在业务层面来解决：

在分页显示的时候，可以在业务层面做到一下几点即可：

- 增加默认的筛选条件，尽量减少数据量的展示，比如：最近一个月；
- 限制总分页数，比如：淘宝、京东仅显示100页查询结果，百度仅显示76页；
- 修改跳页的展现方式，改为滚动显示，或小范围跳页，比如：谷歌、百度的小范围跳页(但是这里请注意,因为限定了返回的总页数在一个比较小的范围，所以，即使跳页也不涉及深分页)。

13. ES的脑裂问题

es集群由多个**数据节点**和一个**主节点**（可以有多个备选主节点）组成。其中数据节点负责数据存储和具体操作，如执行搜索、聚合等任务，计算压力较大。主节点负责创建、删除索引、分配分片、追踪集群中的节点状态等管理工作，计算压力较轻，在es集群中扮演着"大脑"的角色。

正常情况下，当主节点无法工作时，会从备选主节点中选举一个出来变成新主节点，原主节点回归后变成备选主节点

但有时因为网络抖动等原因，主节点没能及时响应，集群误以为主节点挂了，选举了一个新主节点，此时一个es集群中有了两个主节点，其他节点不知道该听谁的调度，结果将是灾难性的！这种类似一个人得了精神分裂症，就被称之为“脑裂”现象。

之所以产生脑裂问题的原因是主节点因为各种原因，在收到请求后未能及时响应，导致主节点未能及时响应的原因，一般主要有以下几点:

- 网络抖动

  内网一般不会出现es集群的脑裂问题，可以监控内网流量状态。外网的网络出现问题的可能性大些

- 节点负载

  如果主节点同时承担数据节点的工作，可能会因为工作负载大而导致对应的 ES 实例停止响。

- 内存回收

  由于数据节点上es进程占用的内存较大，较大规模的内存回收操作也能造成es进程失去响应。

如何解决脑裂问题呢?

- 不要把主节点同时设为数据节点（node.master和node.data不要同时设为true），让主节点只做管理工作
- 将节点响应超时（discovery.zen.ping_timeout）稍稍设置长一些（默认是3秒），避免误判。
- 设置需要超过半数的备选节点同意，才能发生主节点重选，类似需要参议院半数以上通过，才能弹劾现任总统。（discovery.zen.minimum_master_nodes = 半数以上备选主节点数）。

14 如何实现数据库与ES的数据同步

- 首先，在我们的项目中，我们采用的方式，是在搜索服务中专门实现了一个接口，我们手动调用该接口，实现数据库与ES的数据同步，我们的这种方式就相当于，有一个后台管理系统，由运营在后台系统将商品上架，后台系统调用搜索服务接口，讲商品数据同步写入ES，这种方式是同步方式实现，我们也可以说，后台系统将商品数据封装在消息中，发送给RocketMQ, 搜索服务通过消费消息，将商品数据写入ES
- 除此之外，还可以有其他方式，比如可以专门有一个服务，负责数据的同步，该服务或者以定时任务的方式，或者基于专门的数据同步中间件比如canal来完成数据库与ES的数据同步

15 在nacos中服务提供者是如何向nacos注册中心（Registry）续约的?

这里其实就是在问，nacos注册中心如何实时感知服务状态(服务是否存活)。其本质还是通过心跳机制。每一次服务向注册中心发送心跳数据，就相当于一次注册续约。

nacos服务客户端（要注册到nacos的服务）启动时会每隔一段时间（默认5秒）向nacos发生心跳包，nacos注册中心15秒内没有检测到心跳包会默认认为nacos处于一种不健康的状态，30秒还没收到则认为这个服务已不可用。

16 在nacos集群中所采用的一致性协议是什么?

Nacos是支持集群的，在Nacos集群中，针对不同类型的数据，会采用不同的数据一致性协议，来保证Nacos集群中数据的一致性。在Nacos中，存储的数据分成两种，临时实例和持久化实例。针对临时实例，采用自定义的Distro协议(弱一致性协议)来保证集群节点中临时实例数据的一致性，针对持久化实例，采用Raft协议实现持久化实例数据的强一致性。

Distro协议如何保证数据的一致性呢?

- 首先，在Distro协议中，每一个Nacos实例的地位是平等的，都可以处理数据的写请求(比如服务的注册)
- 因为各个服务可能会注册到不同的Nacos实例(Nacos服务器)中去，所以，要想保持各个Nacos实例数据的一致性，就需要让各个Nacos实例相互之间进行数据同步
- 数据的同步采用定时同步的方式，通过异步复制的方式进行
- 所以可以认为每个Nacos实例中，都存储了一份完整的数据(比如服务注册表)

分布式一致性协议Raft如何保证数据的强一致性呢?

Raft 协议有效的借鉴了美国总统大选的策略，采用精英（Raft 称呼这个精英为 Leader）领导全局的方案，整个集群中只有 Leader 可以处理 client 发送过来的请求，其他非 Leader 节点即使接收到请求也必须将其转发到 Leader 节点进行处理。Raft 集群中的成员分三种角色：

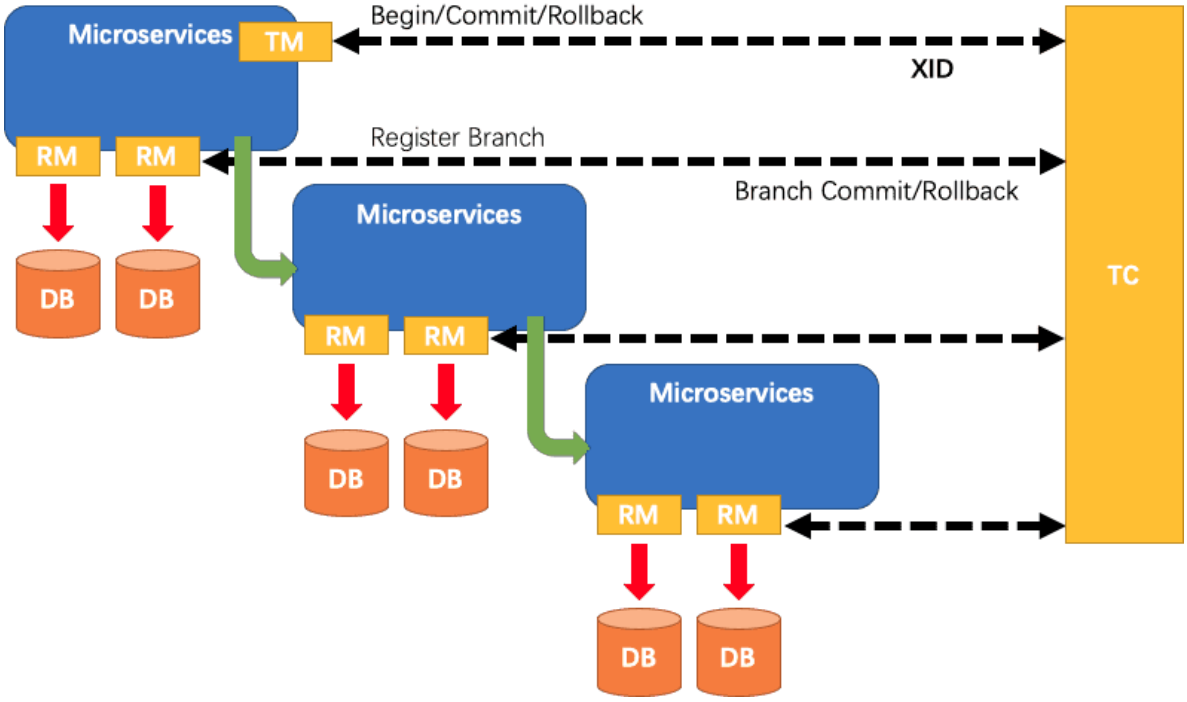
- Leader
- Follower
- Candidate

Raft 协议强依赖 Leader 节点来确保集群数据一致性。即 client 发送过来的数据均先到达 Leader 节点，Leader 接收到数据后，先将数据标记为 uncommitted 状态，随后 Leader 开始向所有 Follower 复制数据并等待响应，在获得集群中大于 N/2 个 Follower 的已成功接收数据完毕的响应后，Leader 将数据的状态标记为 committed，随后向 client 发送数据已接收确认，在向 client 发送出已数据接收后，再向所有 Follower 节点发送通知表明该数据状态为committed。

17 如何修改服务的数据库地址一次，就可以让修改在所有服务实例中生效

- 首先，如果要做到这一点，那么数据库服务器地址配置，必须得存放到配置中心，比如Nacos配置中心，然后让所有的服务实例，从配置中心读取配置
- 其次，如果还要让配置动态生效，那么对于Nacos配置中心而言，我们可以给服务设置针对指定配置集的动态刷新的监听器，只要监听器所监听的数据集中的配置内容发生了改变，在监听器中可以拿到最新配置集内容，然后重新根据新的数据库服务器地址，创建新的数据源对象，放入Spring容器即可，之后用这个新的数据源访问数据库访问的就是新的数据库服务器了

18 Seata AT 模式中是如何实现事物的回滚的?



Seata本身的事物提交过程还是分成两个阶段的

- 第一阶段首先由TM(Transaction Manager)发起全局事物，并调用各个服务(业务逻辑中涉及的服务)，每个服务执行自己的本地事物，在各个本地提交事物之前会先向，会通过RM向TC(Transaction Coordinator)注册本地事物
- 在各个本地事物提交之前，先查询出待修改数据在事物提交之前的值，我们称之为前置镜像，在执行本地事物的提交，事物提交成功之后，在查询出事物提交之后的值,我们称之为后置镜像，然后用前后置镜像生成一条回滚日志，将回滚日志插入到undo_log表中(分布式事务涉及的每个数据库中都需要创建这张表，这是Seata需要的)，然后将本地事物的执行结果，通过RM报告给TC，于是TC就可以知道该分布式事务中涉及的每个本地事物的状态
- 在第一阶段中，即各个服务执行本地事物的过程中，如果有服务执行失败了，执行失败的服务本身的本地事物会回滚，同时，执行失败的服务会根据调用链，层层向上抛出异常，最终异常抛出给包含TM的服务(即分布式事务的发起者)，让TM感知到服务调用出现了问题，此时TM会向TC发起rollback请求
- 接收到rollback请求之后，TC向本次分布式事务中包含的所有本地事物(之前在各个服务执行本地事物之前，由RM向TC注册)，发送rollback请求
- 各个服务的RM接收到该请求之后，执行各个服务本地事物的回滚，获取undo_log中的一条回滚日志，根据回滚日志中记录的前置镜像（在此之前还需要先验证下后置镜像的值，和当前本地事物中涉及的数据值是否相同，相同在继续回滚，不同抛出异常），去恢复数据，实际就是实现了一个反向补偿，从而实现回滚的效果

19 RocketMQ如何保证客户端获取消息的及时性?

要回答这个问题，我们首先要清楚，实际上从理论上来说，RocketMQ客户端从Broker中获取消息，有两种不同的方式：

- 推模式：当Server收到消息发送者发送过来的消息后，Server端主动把消息推送给client，这个方式实时性比较好，但是增加了Server的工作负担，对Server的性能造成影响；另外Client如果不能够及时处理Server推送的消息，也是很大的问题。
- 拉模式：Client循环的从Server拉取消息，由client控制着主动权。弊端：拉取消息的时间间隔不好设定，间隔太短循环空拉取造成资源浪费，间隔时间太长，就会增加消息消费的延迟，影响业务使用。

RocketMQ客户端采用哪种方式呢？RocketMQ的消息消费方式，采用了“长轮询”方式，兼具了Push和Pull的有点，所以说到底，是RocketMQ的长轮询机制，保证了RocketMQ消息接收的及时性(本质上还是采用拉模式)。具体过程如下：

- Client发送消息请求，Server端接受请求，如果有消息，则获取消息并返回。
- 如果发现Server队列里没有新消息，Server端不立即返回，而是持有这个请求(即将这个请求的处理阻塞)一段时间（通过设置超时时间来实现），在这段时间内轮询Server队列内是否有新的消息，如果有新消息，就利用现有的连接返回消息给消费者；如果这段时间内没有新消息进入队列，则返回空。

这样做的好处是，消费消息的主动权既保留在Client端，也不会出现Server积压大量消息后，短时间内推送给Client大量消息的情况(因为消息是客户端主动获取的)，使client因为性能问题出现消费不及时的情况(因为一有消息立马不在阻塞，broker返回消息给客户端)，而且这样做还能避免大量的空轮训(即无效额轮训)。

这样做的弊端是，长轮询的弊端：在持有消费者请求的这段时间，占用了系统资源，因此长轮询适合客户端连接数可控的业务场景中。