



Join Yuque to read better

Want to follow this author to read more or favorite this article? Please [Sign Up](#) or

[Log In](#) to Yuque

[Join now](#)

X

# 210道面试题

## Java基础(33)

### 面向对象

什么是面向对象？对比面向过程，是两种不同的处理问题的角度，面向过程更注重事情的每一个步骤及顺序，面向对象更注重事情有哪些参与者（对象）、及各自需要什么，比如洗衣机洗衣服：

- 面向过程：会将任务拆解成一系列的步骤：打开洗衣机---->放衣服---->放洗衣粉---->清洗---->烘干
- 面向对象：会拆出人和洗衣机两个对象：
  - a. 人：打开洗衣机 放衣服 放洗衣粉
  - b. 洗衣机：清洗 烘干

从以上例子能看出，面向过程比较直接高效，而面向对象更易于复用、扩展和维护。

**封装**：封装的意义，在于明确标识出允许外部使用的所有成员函数和数据项，内部细节对外部调用透明，外部调用无需修改或者关心内部实现

**继承**：继承基类的方法，并做出自己的改变和/或扩展，子类共性的方法或者属性直接使用父类的，而不需要自己再定义，只需扩展自己个性化的

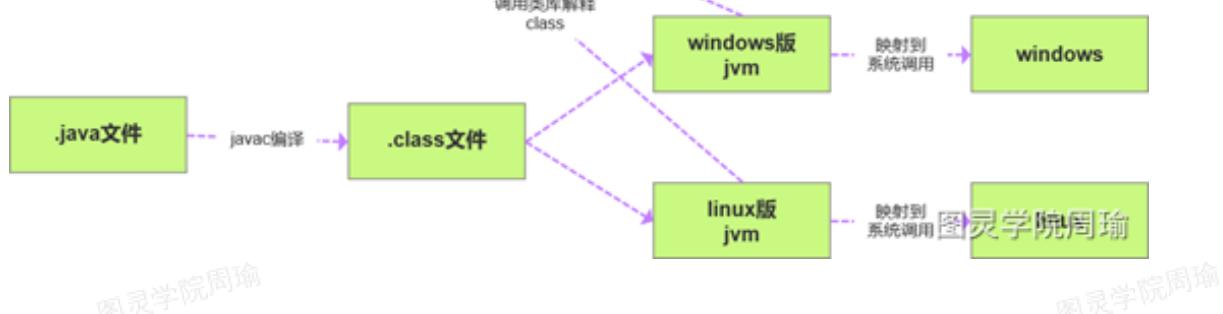
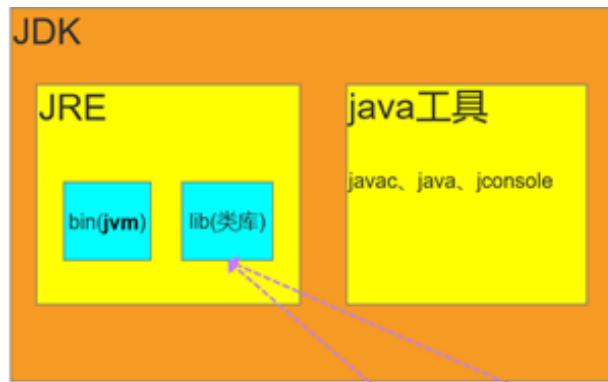
**多态**：基于对象所属类的不同，外部对同一个方法的调用，实际执行的逻辑不同

### JDK、JRE、JVM之间的区别

JDK：Java Development Kit java 开发工具

JRE：Java Runtime Environment java运行时环境

JVM：java Virtual Machine java 虚拟机



## ==和equals方法之前的区别

- **==**：对比的是栈中的值，基本数据类型是变量值，引用类型是堆中内存对象的地址
- **equals**：object中默认也是采用**==**比较，通常会重写

Object

Java | Copy Code

```
1 public boolean equals(Object obj) {
2     return (this == obj);
3 }
```

String

```

1  public boolean equals(Object anObject) {
2      if (this == anObject) {
3          return true;
4      }
5      if (anObject instanceof String) {
6          String anotherString = (String)anObject;
7          int n = value.length;
8          if (n == anotherString.value.length) {
9              char v1[] = value;
10             char v2[] = anotherString.value;
11             int i = 0;
12             while (n-- != 0) {
13                 if (v1[i] != v2[i])
14                     return false;
15                 i++;
16             }
17             return true;
18         }
19     }
20     return false;
21 }
```

上述代码可以看出，String类中被复写的equals()方法其实是比较两个字符串的内容。

```

1  public class StringDemo {
2      public static void main(String args[]) {
3          String str1 = "Hello";
4          String str2 = new String("Hello");
5          String str3 = str2; // 引用传递
6          System.out.println(str1 == str2); // false
7          System.out.println(str1 == str3); // false
8          System.out.println(str2 == str3); // true
9          System.out.println(str1.equals(str2)); // true
10         System.out.println(str1.equals(str3)); // true
11         System.out.println(str2.equals(str3)); // true
12     }
13 }
```

## hashCode()与equals()之间的关系

HashCode介绍：hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，Java中的任何类都包含有 hashCode() 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

## 以“HashSet如何检查重复”为例子来说明为什么要有hashCode

对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，看该位置是否有值，如果没有、HashSet会假设对象没有重复出现。但是如果发现有值，这时会调用equals()方法来检查两个对象是否真的相同。如果两者相同，HashSet就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。这样就大大减少了equals的次数，相应就大大提高了执行速度。

- 如果两个对象相等，则hashcode一定也是相同的
- 两个对象相等,对两个对象分别调用equals方法都返回true
- 两个对象有相同的hashcode值，它们也不一定是相等的
- 因此，equals方法被覆盖过，则hashCode方法也必须被覆盖
- hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

## final关键字的作用是什么？

修饰类：表示类不可被继承

修饰方法：表示方法不可被子类覆盖，但是可以重载

修饰变量：表示变量一旦被赋值就不可以更改它的值。

修饰成员变量：

- 如果final修饰的是类变量，只能在静态初始化块中指定初始值或者声明该类变量时指定初始值。
- 如果final修饰的是成员变量，可以在非静态初始化块、声明该变量或者构造器中执行初始值。

修饰局部变量：

系统不会为局部变量进行初始化，局部变量必须由程序员显示初始化。因此使用final修饰局部变量时，即可以在定义时指定默认值（后面的代码不能对变量再赋值），也可以不指定默认值，而在后面的代码中对final变量赋初值（仅一次）

```

1 public class FinalVar {
2     final static int a = 0; //再声明的时候就需要赋值 或者静态代码块赋值
3     /**
4      static{
5          a = 0;
6      }
7      */
8     final int b = 0; //再声明的时候就需要赋值 或者代码块中赋值 或者构造器赋值
9     /*{
10         b = 0;
11     }*/
12    public static void main(String[] args) {
13        final int localA; //局部变量只声明没有初始化，不会报错，与final无关。
14        localA = 0; //在使用之前一定要赋值
15        //localA = 1; 但是不允许第二次赋值
16    }
17 }

```

修饰基本类型数据和引用类型数据：

- 如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；
- 如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。但是引用的值是可变的。

```

1 public class FinalReferenceTest{
2     public static void main(){
3         final int[] iArr={1,2,3,4};
4         iArr[2]=-3;//合法
5         iArr=null;//非法，对iArr不能重新赋值
6
7         final Person p = new Person(25);
8         p.setAge(24);//合法
9         p=null;//非法
10    }
11 }

```

为什么局部内部类和匿名内部类只能访问局部final变量？

编译之后会生成两个class文件，Test.class Test1.class

```

1 public class Test {
2     public static void main(String[] args) {
3     }
4     //局部final变量a,b
5     public void test(final int b) {//jdk8在这里做了优化，不用写,语法糖，但实际上也是有的，也不能修改
6         final int a = 10;
7         //匿名内部类
8         new Thread(){
9             public void run() {
10                 System.out.println(a);
11                 System.out.println(b);
12             };
13         }.start();
14     }
15 }
16
17 class OutClass {
18     private int age = 12;
19
20     public void outPrint(final int x) {
21         class InClass {
22             public void InPrint() {
23                 System.out.println(x);
24                 System.out.println(age);
25             }
26         }
27         new InClass().InPrint();
28     }
29 }

```

首先需要知道的一点是：内部类和外部类是处于同一个级别的，内部类不会因为定义在方法中就会随着方法的执行完毕就被销毁。

这里就会产生问题：当外部类的方法结束时，局部变量就会被销毁了，但是内部类对象可能还存在(只有没有人再引用它时，才会死亡)。这里就出现了一个矛盾：内部类对象访问了一个不存在的变量。为了解决这个问题，就将局部变量复制了一份作为内部类的成员变量，这样当局部变量死亡后，内部类仍可以访问它，实际访问的是局部变量的"copy"。这样就好像延长了局部变量的生命周期

将局部变量复制为内部类的成员变量时，必须保证这两个变量是一样的，也就是如果我们在内部类中修改了成员变量，方法中的局部变量也得跟着改变，怎么解决问题呢？

就将局部变量设置为final，对它初始化后，我就不让你再去修改这个变量，就保证了内部类的成员变量和方法的局部变量的一致性。这实际上也是一种妥协。使得局部变量与内部类内建立的拷贝保持一致。

# String、StringBuffer、StringBuilder的区别

1. String是不可变的，如果尝试去修改，会新生成一个字符串对象，StringBuffer和StringBuilder是可变的
2. StringBuffer是线程安全的，StringBuilder是线程不安全的，所以在单线程环境下StringBuilder效率会更高

## 重载和重写的区别

1. **重载**：发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。
2. **重写**：发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为private则子类就不能重写该方法。

```
1 public int add(int a, String b)
2 public String add(int a, String b)
3 //编译报错
```

Java | Copy Code

## 接口和抽象类的区别

- 抽象类可以存在普通成员函数，而接口中只能存在public abstract 方法。
- 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的。
- 抽象类只能继承一个，接口可以实现多个。

接口的设计目的，是对类的行为进行约束（更准确的说是一种“有”约束，因为接口不能规定类不可以有什么行为），也就是提供一种机制，可以强制要求不同的类具有相同的行为。它只约束了行为的有无，但不对如何实现行为进行限制。

而抽象类的设计目的，是代码复用。当不同的类具有某些相同的行为(记为行为集合A)，且其中一部分行为的实现方式一致时（A的非真子集，记为B），可以让这些类都派生于一个抽象类。在这个抽象类中实现了B，避免让所有的子类来实现B，这就达到了代码复用的目的。而A减B的部分，留给各个子类自己实现。正是因为A-B在这里没有实现，所以抽象类不允许实例化出来（否则当调用到A-B时，无法执行）。

抽象类是对类本质的抽象，表达的是 is a 的关系，比如：BMW is a Car。抽象类包含并实现子类的通用特性，将子类存在差异化的特性进行抽象，交由子类去实现。

而接口是对行为的抽象，表达的是 like a 的关系。比如：Bird like a Aircraft（像飞行器一样可以飞），但其本质上 is a Bird。接口的核心是定义行为，即实现类可以做什么，至于实现类主体是谁、是如何实现的，接

口并不关心。

使用场景：当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

抽象类的功能要远超过接口，但是，定义抽象类的代价高。因为高级语言来说（从实际设计上来说也是）每个类只能继承一个类。在这个类中，你必须继承或编写出其所有子类的所有共性。虽然接口在功能上会弱化许多，但是它只是针对一个动作的描述。而且你可以在一个类中同时实现多个接口。在设计阶段会降低难度

## List和Set的区别

- List：有序，按对象进入的顺序保存对象，可重复，允许多个Null元素对象，可以使用Iterator取出所有元素，在逐一遍历，还可以使用get(int index)获取指定下标的元素
- Set：无序，不可重复，最多允许有一个Null元素对象，取元素时只能用Iterator接口取得所有元素，在逐一遍历各个元素

## ArrayList和LinkedList区别

1. 首先，他们的底层数据结构不同，ArrayList底层是基于数组实现的，LinkedList底层是基于链表实现的
2. 由于底层数据结构不同，他们所适用的场景也不同，ArrayList更适合随机查找，LinkedList更适合删除和添加，查询、添加、删除的时间复杂度不同
3. 另外ArrayList和LinkedList都实现了List接口，但是LinkedList还额外实现了Deque接口，所以LinkedList还可以当做队列来使用

## HashMap和HashTable有什么区别？其底层实现是什么？

区别：

1. HashMap方法没有synchronized修饰，线程非安全，HashTable线程安全；
2. HashMap允许key和value为null，而HashTable不允许

底层实现：数组+链表实现，jdk8开始链表高度到8、数组长度超过64，链表转变为红黑树，元素以内部类Node节点存在

1. 计算key的hash值，二次hash然后对数组长度取模，对应到数组下标，
2. 如果没有产生hash冲突(下标位置没有元素)，则直接创建Node存入数组，
3. 如果产生hash冲突，先进行equal比较，相同则取代该元素，不同，则判断链表高度插入链表，链表高度达到8，并且数组长度到64则转变为红黑树，长度低于6则将红黑树转回链表
4. key为null，存在下标0的位置

# 谈谈 ConcurrentHashMap 的扩容机制

## 1.7版本

1. 1.7版本的ConcurrentHashMap是基于Segment分段实现的
2. 每个Segment相对于一个小型的HashMap
3. 每个Segment内部会进行扩容，和HashMap的扩容逻辑类似
4. 先生成新的数组，然后转移元素到新数组中
5. 扩容的判断也是每个Segment内部单独判断的，判断是否超过阈值

## 1.8版本

1. 1.8版本的ConcurrentHashMap不再基于Segment实现
2. 当某个线程进行put时，如果发现ConcurrentHashMap正在进行扩容那么该线程一起进行扩容
3. 如果某个线程put时，发现没有正在进行扩容，则将key-value添加到ConcurrentHashMap中，然后判断是否超过阈值，超过了则进行扩容
4. ConcurrentHashMap是支持多个线程同时扩容的
5. 扩容之前也先生成一个新的数组
6. 在转移元素时，先将原数组分组，将每组分给不同的线程来进行元素的转移，每个线程负责一组或多组的元素转移工作

## Jdk1.7到Jdk1.8 HashMap 发生了什么变化(底层)？

1. 1.7中底层是数组+链表，1.8中底层是数组+链表+红黑树，加红黑树的目的是提高HashMap插入和查询整体效率
2. 1.7中链表插入使用的是头插法，1.8中链表插入使用的是尾插法，因为1.8中插入key和value时需要判断链表元素个数，所以需要遍历链表统计链表元素个数，所以正好就直接使用尾插法
3. 1.7中哈希算法比较复杂，存在各种右移与异或运算，1.8中进行了简化，因为复杂的哈希算法的目的就是提高散列性，来提供HashMap的整体效率，而1.8中新增了红黑树，所以可以适当的简化哈希算法，节省CPU资源

## 说一下HashMap的Put方法

先说HashMap的Put方法的大体流程：

1. 根据Key通过哈希算法与与运算得出数组下标
2. 如果数组下标位置元素为空，则将key和value封装为Entry对象（JDK1.7中是Entry对象，JDK1.8中是Node对象）并放入该位置
3. 如果数组下标位置元素不为空，则要分情况讨论
  - a. 如果是JDK1.7，则先判断是否需要扩容，如果要扩容就进行扩容，如果不扩容就生成Entry对象，并使用头插法添加到当前位置的链表中

- b. 如果是JDK1.8，则会先判断当前位置上的Node的类型，看是红黑树Node，还是链表Node
- i. 如果是红黑树Node，则将key和value封装为一个红黑树节点并添加到红黑树中去，在这个过程中会判断红黑树中是否存在当前key，如果存在则更新value
  - ii. 如果此位置上的Node对象是链表节点，则将key和value封装为一个链表Node并通过尾插法插入到链表的最后位置去，因为是尾插法，所以需要遍历链表，在遍历链表的过程中会判断是否存在当前key，如果存在则更新value，当遍历完链表后，将新链表Node插入到链表中，插入到链表后，会看当前链表的节点个数，如果大于等于8，那么则会将该链表转成红黑树
  - iii. 将key和value封装为Node插入到链表或红黑树中后，再判断是否需要进行扩容，如果需要就扩容，如果不需不需要就结束PUT方法

## 泛型中extends和super的区别

1. <? extends T>表示包括T在内的任何T的子类
2. <? super T>表示包括T在内的任何T的父类

## 深拷贝和浅拷贝

深拷贝和浅拷贝就是指对象的拷贝，一个对象中存在两种类型的属性，一种是基本数据类型，一种是实例对象的引用。

1. 浅拷贝是指，只会拷贝基本数据类型的值，以及实例对象的引用地址，并不会复制一份引用地址所指向的对象，也就是浅拷贝出来的对象，内部的类属性指向的是同一个对象
2. 深拷贝是指，既会拷贝基本数据类型的值，也会针对实例对象的引用地址所指向的对象进行复制，深拷贝出来的对象，内部的属性指向的不是同一个对象

## HashMap的扩容机制原理

### 1.7版本

1. 先生成新数组
2. 遍历老数组中的每个位置上的链表上的每个元素
3. 取每个元素的key，并基于新数组长度，计算出每个元素在新数组中的下标
4. 将元素添加到新数组中去
5. 所有元素转移完了之后，将新数组赋值给HashMap对象的table属性

### 1.8版本

1. 先生成新数组
2. 遍历老数组中的每个位置上的链表或红黑树
3. 如果是链表，则直接将链表中的每个元素重新计算下标，并添加到新数组中去
4. 如果是红黑树，则先遍历红黑树，先计算出红黑树中每个元素对应在新数组中的下标位置
  - a. 统计每个下标位置的元素个数

- b. 如果该位置下的元素个数超过了8，则生成一个新的红黑树，并将根节点的添加到新数组的对应位置
  - c. 如果该位置下的元素个数没有超过8，那么则生成一个链表，并将链表的头节点添加到新数组的对应位置
5. 所有元素转移完了之后，将新数组赋值给HashMap对象的table属性

## CopyOnWriteArrayList的底层原理是怎样的

1. 首先CopyOnWriteArrayList内部也是用过数组来实现的，在向CopyOnWriteArrayList添加元素时，会复制一个新的数组，写操作在新数组上进行，读操作在原数组上进行
2. 并且，写操作会加锁，防止出现并发写入丢失数据的问题
3. 写操作结束之后会把原数组指向新数组
4. CopyOnWriteArrayList允许在写操作时来读取数据，大大提高了读的性能，因此适合读多写少的应用场景，但是CopyOnWriteArrayList会比较占内存，同时可能读到的数据不是实时最新的数据，所以不适合实时性要求很高的场景

## 什么是字节码？采用字节码的好处是什么？

**Java中的编译器和解释器：**Java中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在Java中，这种供虚拟机理解的代码叫做字节码（即扩展名为 .class的文件），它不面向任何特定的处理器，只面向虚拟机。

每一种平台的解释器是不同的，但是实现的虚拟机是相同的。Java源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。这也就是解释了Java的编译与解释并存的特点。

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

**采用字节码的好处：**Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行。

## Java中的异常体系是怎样的

- Java中的所有异常都来自顶级父类Throwable。
- Throwable下有两个子类Exception和Error。
- Error是程序无法处理的错误，一旦出现这个错误，则程序将被迫停止运行。
- Exception不会导致程序停止，又分为两个部分RunTimeException运行时异常和CheckedException检查异常。

- RunTimeException常常发生在程序运行过程中，会导致程序当前线程执行失败。CheckedException常常发生在程序编译过程中，会导致程序编译不通过。

## Java中有哪些类加载器

JDK自带了三个类加载器：bootstrap ClassLoader、ExtClassLoader、AppClassLoader。

- BootStrapClassLoader是ExtClassLoader的父类加载器，默认负责加载%JAVA\_HOME%lib下的jar包和class文件。
- ExtClassLoader是AppClassLoader的父类加载器，负责加载%JAVA\_HOME%/lib/ext文件夹下的jar包和class类。
- AppClassLoader是自定义类加载器的父类，负责加载classpath下的类文件。

## 说说类加载器双亲委派模型

JVM中存在三个默认的类加载器：

1. BootstrapClassLoader
2. ExtClassLoader
3. AppClassLoader

AppClassLoader的父加载器是ExtClassLoader，ExtClassLoader的父加载器是BootstrapClassLoader。

JVM在加载一个类时，会调用AppClassLoader的loadClass方法来加载这个类，不过在这个方法中，会先使用ExtClassLoader的loadClass方法来加载类，同样ExtClassLoader的loadClass方法中会先使用BootstrapClassLoader来加载类，如果BootstrapClassLoader加载到了就直接成功，如果BootstrapClassLoader没有加载到，那么ExtClassLoader就会自己尝试加载该类，如果没有加载到，那么则会由AppClassLoader来加载这个类。

所以，双亲委派指得是，JVM在加载类时，会委派给Ext和Bootstrap进行加载，如果没加载到才由自己进行加载。

## GC如何判断对象可以被回收

- 引用计数法：每个对象都有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收，
- 可达性分析法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的，那么虚拟机就判断是可回收对象。

引用计数法，可能会出现A 引用了 B，B 又引用了 A，这时候就算他们都不再使用了，但因为相互引用 计数器 =1 永远无法被回收。

GC Roots的对象有：

- 虚拟机栈(栈帧中的本地变量表) 中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI(即一般说的Native方法)引用的对象

可达性算法中的不可达对象并不是立即死亡的，对象拥有一次自我拯救的机会。对象被系统宣告死亡至少要经历两次标记过程：第一次是经过可达性分析发现没有与GC Roots相连接的引用链，第二次是在由虚拟机自动建立的Finalizer队列中判断是否需要执行finalize()方法。

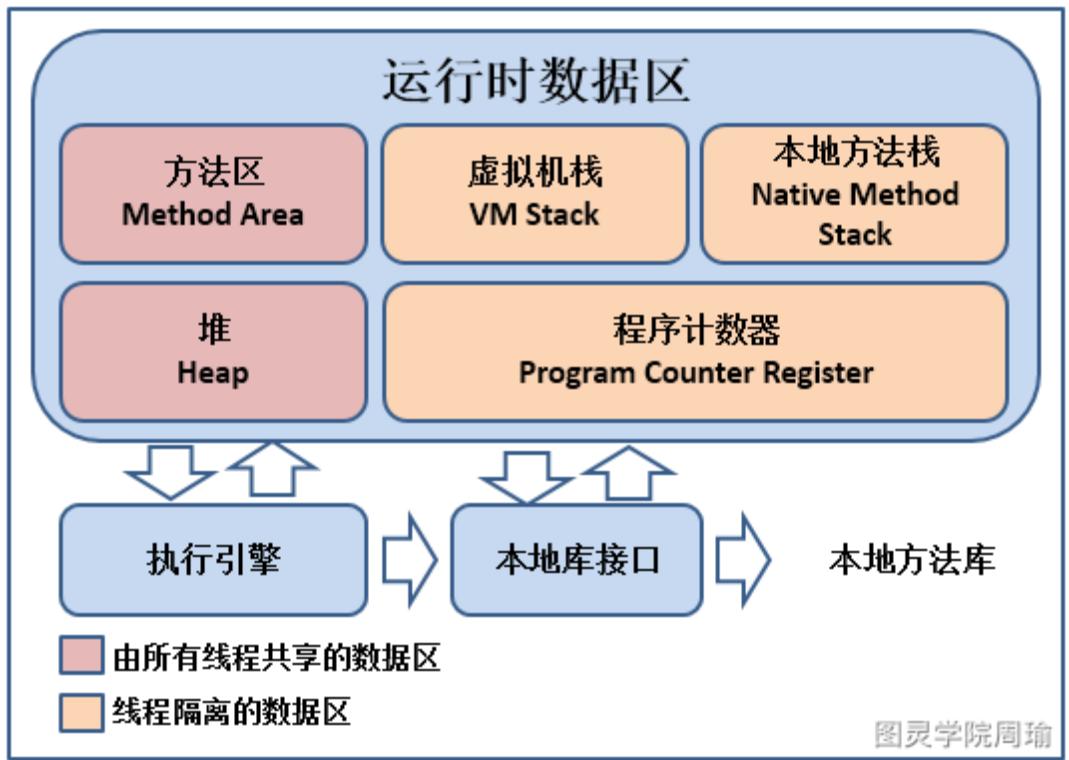
当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行 finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”

每个对象只能触发一次finalize()方法

由于finalize()方法运行代价高昂，不确定性大，无法保证各个对象的调用顺序，不推荐大家使用，建议遗忘它。

## JVM中哪些是线程共享区

堆区和方法区是所有线程共享的，栈、本地方法栈、程序计数器是每个线程独有的



## 你们项目如何排查JVM问题

对于还在正常运行的系统：

1. 可以使用jmap来查看JVM中各个区域的使用情况
2. 可以通过jstack来查看线程的运行情况，比如哪些线程阻塞、是否出现了死锁
3. 可以通过jstat命令来查看垃圾回收的情况，特别是fullgc，如果发现fullgc比较频繁，那么就得进行调优了
4. 通过各个命令的结果，或者jvisualvm等工具来进行分析
5. 首先，初步猜测频繁发送fullgc的原因，如果频繁发生fullgc但是又一直没有出现内存溢出，那么表示fullgc实际上是回收了很多对象了，所以这些对象最好能在younggc过程中就直接回收掉，避免这些对象进入到老年代，对于这种情况，就要考虑这些存活时间不长的对象是不是比较大，导致年轻代放不下，直接进入到了老年代，尝试加大年轻代的大小，如果改完之后，fullgc减少，则证明修改有效
6. 同时，还可以找到占用CPU最多的线程，定位到具体的方法，优化这个方法的执行，看是否能避免某些对象的创建，从而节省内存

对于已经发生了OOM的系统：

1. 一般生产系统中都会设置当系统发生了OOM时，生成当时的dump文件（-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/base）
2. 我们可以利用jvisualvm等工具来分析dump文件
3. 根据dump文件找到异常的实例对象，和异常的线程（占用CPU高），定位到具体的代码
4. 然后再进行详细的分析和调试

总之，调优不是一蹴而就的，需要分析、推理、实践、总结、再分析，最终定位到具体的问题

# 一个对象从加载到JVM，再到被GC清除，都经历了什么过程？

1. 用户创建一个对象，JVM首先需要到方法区去找对象的类型信息。然后再创建对象。
2. JVM要实例化一个对象，首先要在堆当中先创建一个对象。->半初始化状态
3. 对象首先会分配在堆内存中新生代的Eden。然后经过一次Minor GC，对象如果存活，就会进入S区。在后续的每次GC中，如果对象一直存活，就会在S区来回拷贝，每移动一次，年龄加1。->多大年龄才会移入老年代？年龄最大15，超过一定年龄后，对象转入老年代。
4. 当方法执行结束后，栈中的指针会先移除掉。
5. 堆中的对象，经过Full GC，就会被标记为垃圾，然后被GC线程清理掉。

## 怎么确定一个对象到底是不是垃圾？

1. 引用计数：这种方式是给堆内存当中的每个对象记录一个引用个数。引用个数为0的就认为是垃圾。这是早期JDK中使用的方式。引用计数无法解决循环引用的问题。
2. 根可达算法：这种方式是在内存中，从引用根对象向下一直找引用，找不到的对象就是垃圾。

## JVM有哪些垃圾回收算法？

1. MarkSweep 标记清除算法：这个算法分为两个阶段，标记阶段：把垃圾内存标记出来，清除阶段：直接将垃圾内存回收。这种算法是比较简单的，但是有个很严重的问题，就是会产生大量的内存碎片。
2. Copying 拷贝算法：为了解决标记清除算法的内存碎片问题，就产生了拷贝算法。拷贝算法将内存分为大小相等的两半，每次只使用其中一半。垃圾回收时，将当前这一块的存活对象全部拷贝到另一半，然后当前这一半内存就可以直接清除。这种算法没有内存碎片，但是他的问题就在于浪费空间。而且，他的效率跟存货对象的个数有关。
3. MarkCompact 标记压缩算法：为了解决拷贝算法的缺陷，就提出了标记压缩算法。这种算法在标记阶段跟标记清除算法是一样的，但是在完成标记之后，不是直接清理垃圾内存，而是将存活对象往一端移动，然后将端边界以外的所有内存直接清除。

这三种算法各有利弊，各自有各自的适合场景。

## 什么是STW？

STW: Stop-The-World，是在垃圾回收算法执行过程当中，需要将JVM内存冻结的一种状态。在STW状态下，JAVA的所有线程都是停止执行的-GC线程除外，native方法可以执行，但是，不能与JVM交互。GC各种算法优化的重点，就是减少STW，同时这也是JVM调优的重点。

# JVM有哪些垃圾回收器？

- 新生代收集器：
  - Serial
  - ParNew
  - Parallel Scavenge
- 老年代收集器：
  - CMS
  - Serial Old
  - Parallel Old
- 整堆收集器：
  - G1

## 垃圾回收分为哪些阶段

GC分为四个阶段：

- 第一：初始标记 标记出GCRoot直接引用的对象。STW
- 第二：标记Region，通过RSet标记出上一个阶段标记的Region引用到的Old区Region。
- 第三：并发标记阶段：跟CMS的步骤是差不多的。只是遍历的范围不再是整个Old区，而只需要遍历第二步标记出来的Region。
- 第四：重新标记：跟CMS中的重新标记过程是差不多的。
- 第五：垃圾清理：与CMS不同的是，G1可以采用拷贝算法，直接将整个Region中的对象拷贝到另一个Region。而这个阶段，G1只选择垃圾较多的Region来清理，并不是完全清理。

## 什么是三色标记？

三色标记：是一种逻辑上的抽象。将每个内存对象分成三种颜色：

1. 黑色：表示自己和成员变量都已经标记完毕。
2. 灰色：自己标记完了，但是成员变量还没有完全标记完。
3. 白色：自己未标记完。

## JVM参数有哪些？

JVM参数大致可以分为三类：

1. 标注指令： -开头，这些是所有的HotSpot都支持的参数。可以用java -help 打印出来。
2. 非标准指令： -X开头，这些指令通常是跟特定的HotSpot版本对应的。可以用java -X 打印出来。
3. 不稳定参数： -XX 开头，这一类参数是跟特定HotSpot版本对应的，并且变化非常大。详细的文档资料非常少。在JDK1.8版本下，有几个常用的不稳定指令：

java -XX:+PrintCommandLineFlags : 查看当前命令的不稳定指令。

java -XX:+PrintFlagsInitial : 查看所有不稳定指令的默认值。

java -XX:+PrintFlagsFinal : 查看所有不稳定指令最终生效的实际值。

## Java并发(20)

### 线程的生命周期？线程有几种状态

线程通常有五种状态，创建，就绪，运行、阻塞和死亡状态：

1. 新建状态 (New) : 新创建了一个线程对象。
2. 就绪状态 (Runnable) : 线程对象创建后，其他线程调用了该对象的start方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。
3. 运行状态 (Running) : 就绪状态的线程获取了CPU，执行程序代码。
4. 阻塞状态 (Blocked) : 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。
5. 死亡状态 (Dead) : 线程执行完了或者因异常退出了run方法，该线程结束生命周期。

阻塞的情况又分为三种：

1. 等待阻塞：运行的线程执行wait方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify或notifyAll方法才能被唤醒，wait是object类的方法
2. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
3. 其他阻塞：运行的线程执行sleep或join方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep状态超时、join等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。sleep是Thread类的方法

### sleep()、wait()、join()、yield()之间的的区别

锁池：所有需要竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池进行等待，当前的线程释放同步锁后锁池中的线程去竞争同步锁，当某个线程得到后会进入就绪队列进行等待cpu资源分配。

等待池：当我们调用wait（）方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁。只有调用了notify（）或notifyAll()后等待池的线程才会开始去竞争锁，notify（）是随机从等待池选出一个线程放到锁池，而notifyAll()是将等待池的所有线程放到锁池当中

1. sleep 是 Thread 类的静态本地方法，wait 则是 Object 类的本地方法。

2. sleep方法不会释放lock，但是wait会释放，而且会加入到等待队列中。

Plain Text | Copy Code

1 sleep就是把cpu的执行资格和执行权释放出去，不再运行此线程，当定时时间结束再取回cpu资源，参与cpu的调度，获取到cpu资源后就可以继续运行了。而如果sleep时该线程有锁，那么sleep不会释放这个锁，而是把锁带着进入了冻结状态，也就是说其他需要这个锁的线程根本不可能获取到这个锁。也就是说无法执行程序。如果在睡眠期间其他线程调用了这个线程的interrupt方法，那么这个线程也会抛出InterruptedException异常返回，这点和wait是一样的。

3. sleep方法不依赖于同步器synchronized，但是wait需要依赖synchronized关键字。

4. sleep不需要被唤醒（休眠之后推出阻塞），但是wait需要（不指定时间需要被别人中断）。

5. sleep 一般用于当前线程休眠，或者轮循暂停操作，wait 则多用于多线程之间的通信。

6. sleep 会让出 CPU 执行时间且强制上下文切换，而 wait 则不一定，wait 后可能还是有机会重新竞争到锁继续执行的。

7. yield () 执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行

8. join () 执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队列，直到线程A结束或中断线程

Java | Copy Code

```
1 public static void main(String[] args) throws InterruptedException {
2     Thread t1 = new Thread(new Runnable() {
3         @Override
4         public void run() {
5             try {
6                 Thread.sleep(3000);
7             } catch (InterruptedException e) {
8                 e.printStackTrace();
9             }
10            System.out.println("22222222");
11        }
12    });
13    t1.start();
14    t1.join();
15    // 这行代码必须要等t1全部执行完毕，才会执行
16    System.out.println("1111");
17 }
18
19 22222222
20 1111
```

不是线程安全、应该是内存安全，堆是共享内存，可以被所有线程访问，当多个线程访问一个对象时，如果不进行额外的同步控制或其他的协调操作，调用这个对象的行为都可以获得正确的结果，我们就说这个对象是线程安全的。

**堆**是进程和线程共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是用完了要还给操作系统，要不然就是内存泄漏。在Java中，堆是Java虚拟机所管理的内存中最大的一块，是所有线程共享的一块内存区域，在虚拟机启动时创建。堆所存在的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。

**栈**是每个线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立，因此，栈是线程安全的。操作系统在切换线程的时候会自动切换栈。栈空间不需要在高级语言里面显式的分配和释放。

目前主流操作系统都是多任务的，即多个进程同时运行。为了保证安全，每个进程只能访问分配给自己的内存空间，而不能访问别的进程的，这是由操作系统保障的。

在每个进程的内存空间中都会有一块特殊的公共区域，通常称为堆（内存）。进程内的所有线程都可以访问到该区域，这就是造成问题的潜在原因。

## Thread和Runnable的区别

Thread和Runnable的实质是继承关系，没有可比性。无论使用Runnable还是Thread，都会new Thread，然后执行run方法。用法上，如果有复杂的线程操作需求，那就选择继承Thread，如果只是简单的执行一个任务，那就实现Runnable。

[Java](#) | [Copy Code](#)

```
1 //会卖出多一倍的票
2 public class Test {
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5
6         new MyThread().start();
7         new MyThread().start();
8
9     }
10
11
12     static class MyThread extends Thread{
13         private int ticket = 5;
14         public void run(){
15             while(true){
16                 System.out.println("Thread ticket = " + ticket--);
17                 if(ticket < 0){
18                     break;
19                 }
20             }
21         }
22     }
23 }
```

[Java](#) | [Copy Code](#)

```
1 //正常卖出
2 public class Test2 {
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5         MyThread2 mt=new MyThread2();
6         new Thread(mt).start();
7         new Thread(mt).start();
8
9
10    }
11    static class MyThread2 implements Runnable{
12        private int ticket = 5;
13        public void run(){
14            while(true){
15                System.out.println("Runnable ticket = " + ticket--);
16                if(ticket < 0){
17                    break;
18                }
19            }
20        }
21    }
22 }
```

原因是：MyThread创建了两个实例，自然会卖出两倍，属于用法错误

# 对守护线程的理解

守护线程：为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；

守护线程类似于整个进程的一个默默无闻的小喽喽；它的生死无关重要，它却依赖整个进程而运行；哪天其他线程结束了，没有要执行的了，程序就结束了，理都没理守护线程，就把它中断了；

注意：由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

守护线程的作用是什么？

举例，GC垃圾回收线程：就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread，程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

应用场景：（1）来为其它线程提供服务支持的情况；（2）或者在任何情况下，程序结束时，这个线程必须正常且立刻关闭，就可以作为守护线程来使用；反之，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就不能是守护线程，而是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的。

`thread.setDaemon(true)`必须在`thread.start()`之前设置，否则会跑出一个`IllegalThreadStateException`异常。你不能把正在运行的常规线程设置为守护线程。

在`Daemon`线程中产生的新线程也是`Daemon`的。

守护线程不能用于去访问固有资源，比如读写操作或者计算逻辑。因为它会在任何时候甚至在一个操作的中间发生中断。

Java自带的多线程框架，比如`ExecutorService`，会将守护线程转换为用户线程，所以如果要使用后台线程就不能用Java的线程池。

## ThreadLocal的底层原理

1. `ThreadLocal`是Java中所提供的线程本地存储机制，可以利用该机制将数据缓存在某个线程内部，该线程可以在任意时刻、任意方法中获取缓存的数据
2. `ThreadLocal`底层是通过`ThreadLocalMap`来实现的，每个`Thread`对象（注意不是`ThreadLocal`对象）中都存在一个`ThreadLocalMap`，`Map`的key为`ThreadLocal`对象，`Map`的value为需要缓存的值
3. 如果在线程池中使用`ThreadLocal`会造成内存泄漏，因为当`ThreadLocal`对象使用完之后，应该要把设置的key，value，也就是`Entry`对象进行回收，但线程池中的线程不会回收，而线程对象是通过强引用指向`ThreadLocalMap`，`ThreadLocalMap`也是通过强引用指向`Entry`对象，线程不被回收，`Entry`对象也就不会被回收

收，从而出现内存泄漏，解决办法是，在使用了ThreadLocal对象之后，手动调用ThreadLocal的remove方法，手动清楚Entry对象

4. ThreadLocal经典的应用场景就是连接管理（一个线程持有一个连接，该连接对象可以在不同的方法之间进行传递，线程之间不共享同一个连接）

## 并发、并行、串行之间的区别

1. 串行在时间上不可能发生重叠，前一个任务没搞定，下一个任务就只能等着
2. 并行在时间上是重叠的，两个任务在同一时刻互不干扰的同时执行。
3. 并发允许两个任务彼此干扰。统一时间点、只有一个任务运行，交替执行

## 并发的三大特性

### 原子性

原子性是指在一个操作中cpu不可以在中途暂停然后再调度，即不被中断操作，要不全部执行完成，要不都不执行。就好比转账，从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。2个操作必须全部完成。

```
1 private long count = 0;
2
3 public void calc() {
4     count++;
5 }
```

Java | Copy Code

- 1：将 count 从主存读到工作内存中的副本中

- 2 : +1的运算
- 3 : 将结果写入工作内存
- 4 : 将工作内存的值刷回主存(什么时候刷入由操作系统决定，不确定的)

那程序中原子性指的是最小的操作单元，比如自增操作，它本身其实并不是原子性操作，分了3步的，包括读取变量的原始值、进行加1操作、写入工作内存。所以在多线程中，有可能一个线程还没自增完，可能才执行到第二部，另一个线程就已经读取了值，导致结果错误。那如果我们能保证自增操作是一个原子性的操作，那么就能保证其他线程读取到的一定是自增后的数据。

**关键字** : synchronized

## 可见性

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

若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。

```
1 //线程1
2 boolean stop = false;
3 while(!stop){
4     doSomething();
5 }
6
7 //线程2
8 stop = true;
```

Java | Copy Code

如果线程2改变了stop的值，线程1一定会停止吗？不一定。当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

**关键字** : volatile、synchronized、final

## 有序性

虚拟机在进行代码编译时，对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按照我们写的代码的顺序来执行，有可能将他们重排序。实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题。

```

1 int a = 0;
2 bool flag = false;
3
4 public void write() {
5     a = 2;           //1
6     flag = true;    //2
7 }
8
9 public void multiply() {
10    if (flag) {      //3
11        int ret = a * a; //4
12    }
13
14 }

```

write方法里的1和2做了重排序，线程1先对flag赋值为true，随后执行到线程2，ret直接计算出结果，再到线程1，这时候a才赋值为2，很明显迟了一步

**关键字：**volatile、synchronized

volatile本身就包含了禁止指令重排序的语义，而synchronized关键字是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则明确的。

synchronized关键字同时满足以上三种特性，但是volatile关键字不满足原子性。

在某些情况下，volatile的同步机制的性能确实要优于锁(使用synchronized关键字或java.util.concurrent包里面的锁)，因为volatile的总开销要比锁低。

我们判断使用volatile还是加锁的唯一依据就是volatile的语义能否满足使用的场景(原子性)

## Java死锁如何避免？

造成死锁的几个原因：

1. 一个资源每次只能被一个线程使用
2. 一个线程在阻塞等待某个资源时，不释放已占有资源
3. 一个线程已经获得的资源，在未使用完之前，不能被强行剥夺
4. 若干线程形成头尾相接的循环等待资源关系

这是造成死锁必须要达到的4个条件，如果要避免死锁，只需要不满足其中某一个条件即可。而其中前3个条件是作为锁要符合的条件，所以要避免死锁就需要打破第4个条件，不出现循环等待锁的关系。

在开发过程中：

1. 要注意加锁顺序，保证每个线程按同样的顺序进行加锁
2. 要注意加锁时限，可以针对所设置一个超时时间
3. 要注意死锁检查，这是一种预防机制，确保在第一时间发现死锁并进行解决

## 如何理解volatile关键字

保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知。

如果线程2改变了stop的值，线程1一定会停止吗？不一定。当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

```
1 //线程1
2 boolean stop = false;
3 while(!stop){
4     doSomething();
5 }
6
7 //线程2
8 stop = true;
```

Java | Copy Code

### 禁止指令重排序优化

```
1 int a = 0;
2 bool flag = false;
3
4 public void write() {
5     a = 2;           //1
6     flag = true;    //2
7 }
8
9 public void multiply() {
10    if (flag) {      //3
11        int ret = a * a; //4
12    }
13
14 }
```

Java | Copy Code

write方法里的1和2做了重排序，线程1先对flag赋值为true，随后执行到线程2，ret直接计算出结果，再到线程1，这时候a才赋值为2，很明显迟了一步。但是用volatile修饰之后就变得不一样了：

1. 使用volatile关键字会强制将修改的值立即写入主存；
2. 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

3. 由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

`inc++;` 其实是两个步骤，先加加，然后再赋值。不是原子性操作，所以`volatile`不能保证线程安全。

## 为什么用线程池？解释下线程池参数？

1、降低资源消耗；提高线程利用率，降低创建和销毁线程的消耗。

2、提高响应速度；任务来了，直接有线程可用可执行，而不是先创建线程，再执行。

3、提高线程的可管理性；线程是稀缺资源，使用线程池可以统一分配调优监控。

- `corePoolSize` 代表核心线程数，也就是正常情况下创建工作线程数，这些线程创建后并不会消除，而是一种常驻线程
- `maximumPoolSize` 代表的是最大线程数，它与核心线程数相对应，表示最大允许被创建的线程数，比如当前任务较多，将核心线程数都用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数
- `keepAliveTime`、`unit` 表示超出核心线程数之外的线程的空闲存活时间，也就是核心线程不会消除，但是超出核心线程数的部分线程如果空闲一定的时间则会被消除，我们可以通过 `setKeepAliveTime` 来设置空闲时间
- `workQueue` 用来存放待执行的任务，假设我们现在核心线程都已被使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还持续进入则会开始创建新的线程
- `ThreadFactory` 实际上是一个线程工厂，用来生产线程执行任务。我们可以选择使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂
- `Handler` 任务拒绝策略，有两种情况，第一种是当我们调用 `shutdown` 等方法关闭线程池后，这时候即使线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续想线程池提交任务就会遭到拒绝。另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，这是也就拒绝

## 线程池的底层工作原理

线程池内部是通过队列+线程实现的，当我们利用线程池执行任务时：

1. 如果此时线程池中的线程数量小于`corePoolSize`，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。
2. 如果此时线程池中的线程数量等于`corePoolSize`，但是缓冲队列`workQueue`未满，那么任务被放入缓冲队列。

3. 如果此时线程池中的线程数量大于等于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务。
4. 如果此时线程池中的线程数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过handler所指定的策略来处理此任务。
5. 当线程池中的线程数量大于corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数

## 线程池中阻塞队列的作用？为什么是先添加列队而不是先创建最大线程？

1、一般的队列只能保证作为一个有限长度的缓冲区，如果超出了缓冲长度，就无法保留当前的任务了，阻塞队列通过阻塞可以保住当前想要继续入队的任务。

阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。

阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时，线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活、不至于一直占用cpu资源

2、在创建新线程的时候，是要获取全局锁的，这个时候其它的就得阻塞，影响了整体效率。

就好比一个企业里面有10个（core）正式工的名额，最多招10个正式工，要是任务超过正式工人数（task > core）的情况下，工厂领导（线程池）不是首先扩招工人，还是这10人，但是任务可以稍微积压一下，即先放到队列去（代价低）。10个正式工慢慢干，迟早会干完的，要是任务还在继续增加，超过正式工的加班忍耐极限了（队列满了），就的招外包帮忙了（注意是临时工）要是正式工加上外包还是不能完成任务，那新来的任务就会被领导拒绝了（线程池的拒绝策略）。

## 线程池中线程复用原理

线程池将线程和任务进行解耦，线程是线程，任务是任务，摆脱了之前通过Thread创建线程时的一个线程必须对应一个任务的限制。

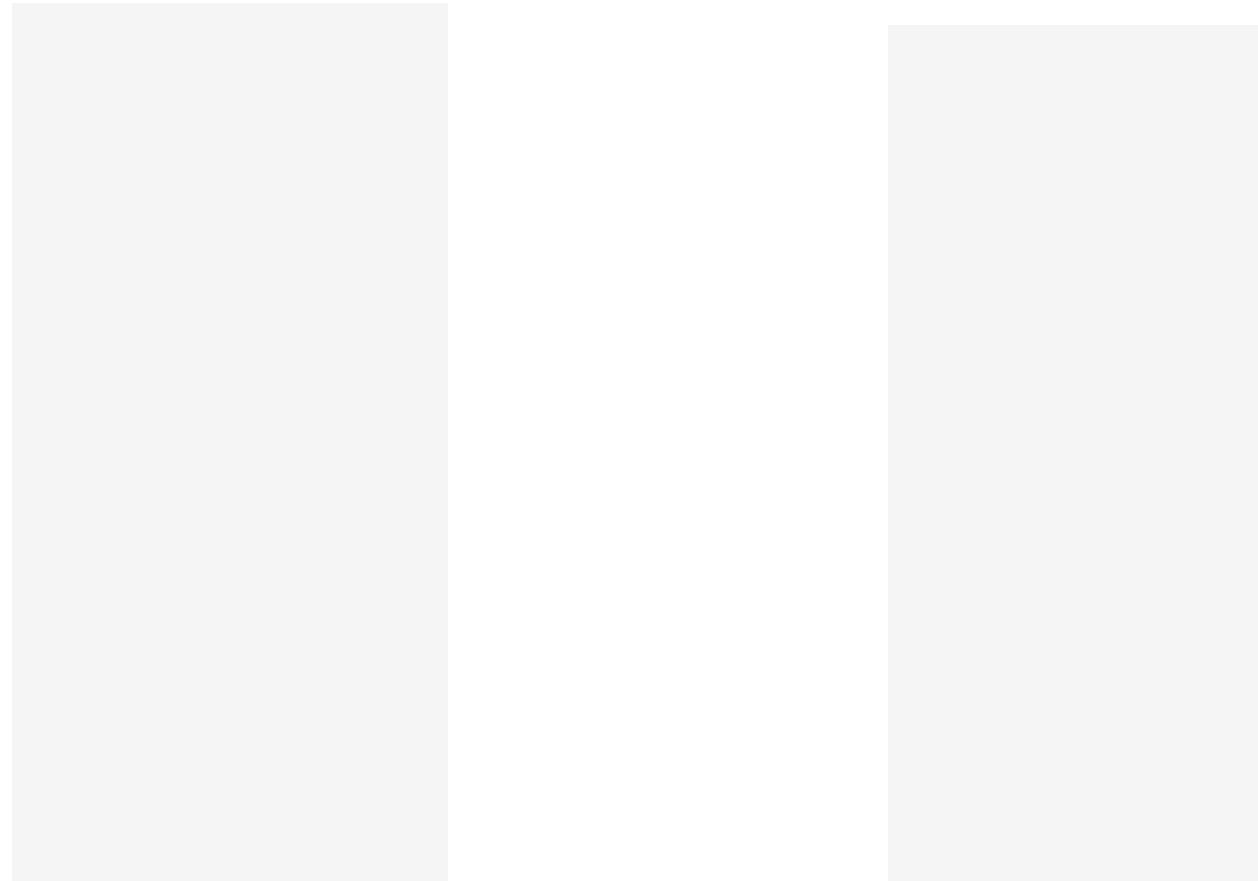
在线程池中，同一个线程可以从阻塞队列中不断获取新任务来执行，其核心原理在线程池对Thread进行了封装，并不是每次执行任务都会调用Thread.start()来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的run方法，将run方法当成一个普通的方法执行，通过这种方式只使用固定的线程就将所有任务的run方法串联起来。

# ReentrantLock中的公平锁和非公平锁的底层实现

首先不管是公平锁和非公平锁，它们的底层实现都会使用AQS来进行排队，它们的区别在于：线程在使用lock()方法加锁时，如果是公平锁，会先检查AQS队列中是否存在线程在排队，如果有线程在排队，则当前线程也进行排队，如果是非公平锁，则不会去检查是否有线程在排队，而是直接竞争锁。

不管是公平锁还是非公平锁，一旦没竞争到锁，都会进行排队，当锁释放时，都是唤醒排在最前面的线程，所以非公平锁只是体现在了线程加锁阶段，而没有体现在线程被唤醒阶段。

另外，ReentrantLock是可重入锁，不管是公平锁还是非公平锁都是可重入的。



## ReentrantLock中tryLock()和lock()方法的区别

1. tryLock()表示尝试加锁，可能加到，也可能加不到，该方法不会阻塞线程，如果加到锁则返回true，没有加到则返回false
2. lock()表示阻塞加锁，线程会阻塞直到加到锁，方法也没有返回值

## CountDownLatch和Semaphore的区别和底层原理

CountDownLatch表示计数器，可以给CountDownLatch设置一个数字，一个线程调用CountDownLatch的await()将会阻塞，其他线程可以调用CountDownLatch的countDown()方法来对CountDownLatch中的数字减一，当数字被减成0后，所有await的线程都将被唤醒。

对应的底层原理就是，调用await()方法的线程会利用AQS排队，一旦数字被减为0，则会将AQS中排队的线程依次唤醒。

Semaphore表示信号量，可以设置许可的个数，表示同时允许最多多少个线程使用该信号量，通过acquire()来获取许可，如果没有许可可用则线程阻塞，并通过AQS来排队，可以通过release()方法来释放许可，当某个线程释放了某个许可后，会从AQS中正在排队的第一个线程开始依次唤醒，直到没有空闲许可。

## Synchronized的偏向锁、轻量级锁、重量级锁

1. 偏向锁：在锁对象的头中记录一下当前获取到该锁的线程ID，该线程下次如果又来获取该锁就可以直接获取到了
2. 轻量级锁：由偏向锁升级而来，当一个线程获取到锁后，此时这把锁是偏向锁，此时如果有第二个线程来竞争锁，偏向锁就会升级为轻量级锁，之所以叫轻量级锁，是为了和重量级锁区分开来，轻量级锁底层是通过自旋来实现的，并不会阻塞线程
3. 如果自旋次数过多仍然没有获取到锁，则会升级为重量级锁，重量级锁会导致线程阻塞
4. 自旋锁：自旋锁就是线程在获取锁的过程中，不会去阻塞线程，也就无所谓唤醒线程，阻塞和唤醒这两个步骤都是需要操作系统去进行的，比较消耗时间，自旋锁是线程通过CAS获取预期的一个标记，如果没有获取到，则继续循环获取，如果获取到了则表示获取到了锁，这个过程线程一直在运行中，相对而言没有使用太多的操作系统资源，比较轻量。

## Synchronized和ReentrantLock的区别

1. synchronized是一个关键字，ReentrantLock是一个类
2. synchronized会自动的加锁与释放锁，ReentrantLock需要程序员手动加锁与释放锁
3. synchronized的底层是JVM层面的锁，ReentrantLock是API层面的锁
4. synchronized是非公平锁，ReentrantLock可以选择公平锁或非公平锁
5. synchronized锁的是对象，锁信息保存在对象头中，ReentrantLock通过代码中int类型的state标识来标识锁的状态
6. synchronized底层有一个锁升级的过程

## 谈谈你对AQS的理解，AQS如何实现可重入锁？

1. AQS是一个JAVA线程同步的框架。是JDK中很多锁工具的核心实现框架。
2. 在AQS中，维护了一个信号量state和一个线程组成的双向链表队列。其中，这个线程队列，就是用来给线程排队的，而state就像是一个红绿灯，用来控制线程排队或者放行的。在不同的场景下，有不用的意义。

3. 在可重入锁这个场景下，state就用来表示加锁的次数。0标识无锁，每加一次锁，state就加1。释放锁state就减1。

## 开发框架(28)

### Spring是什么？

轻量级的开源的J2EE框架。它是一个容器框架，用来装javabean（java对象），中间层框架（万能胶）可以起一个连接作用，比如说把Struts和hibernate粘合在一起运用，可以让我们的企业开发更快、更简洁，Spring是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架：

- 从大小与开销两方面而言Spring都是轻量级的。
- 通过控制反转(IoC)的技术达到松耦合的目的
- 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务进行内聚性的开发
- 包含并管理应用对象(Bean)的配置和生命周期，这个意义上是一个容器。
- 将简单的组件配置、组合成为复杂的应用，这个意义上是一个框架。

### 谈谈你对AOP的理解

系统是由许多不同的组件所组成的，每一个组件各负责一块特定功能。除了实现自身核心功能之外，这些组件还经常承担着额外的职责。例如日志、事务管理和安全这样的核心服务经常融入到自身具有核心业务逻辑的组件中去。这些系统服务经常被称为横切关注点，因为它们会跨越系统的多个组件。

当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。

日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。

在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP：将程序中的交叉业务逻辑（比如安全，日志，事务等），封装成一个切面，然后注入到目标对象（具体业务逻辑）中去。AOP可以对某个对象或某些对象的功能进行增强，比如对象中的方法进行增强，可以在执行某个方法之前额外的做一些事情，在某个方法执行之后额外的做一些事情

### 谈谈你对IOC的理解

容器概念、控制反转、依赖注入

IOC容器：实际上就是个map (key , value) ，里面存的是各种对象（在xml里配置的bean节点、@repository、@service、@controller、@component） ，在项目启动的时候会读取配置文件里面的bean节点，根据全限定类名使用反射创建对象放到map里、扫描到打上上述注解的类还是通过反射创建对象放到map里。

这个时候map里就有各种对象了，接下来我们在代码里需要用到里面的对象时，再通过DI注入（autowired、resource等注解，xml里bean节点内的ref属性，项目启动的时候会读取xml节点ref属性根据id注入，也会扫描这些注解，根据类型或id注入；id就是对象名）。

控制反转：

没有引入IOC容器之前，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。

引入IOC容器之后，对象A与对象B之间失去了直接联系，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方。

通过前后的对比，不难看出来：对象A获得依赖对象B的过程，由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把IOC容器比喻成“粘合剂”的由来。

依赖注入：

“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。依赖注入是实现IOC的方法，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。

## 解释下Spring支持的几种bean的作用域。

- singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。该对象的生命周期是与Spring IOC容器一致的（但在第一次被注入时才会创建）。
- prototype：为每一个bean请求提供一个实例。在每次注入时都会创建一个新的对象
- request：bean被定义为在每个HTTP请求中创建一个单例对象，也就是说在单个请求中都会复用这一个单例对象。
- session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
- application：bean被定义为在ServletContext的生命周期中复用一个单例对象。

- websocket : bean被定义为在websocket的生命周期中复用一个单例对象。
- global-session : 全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

## Spring事务的实现方式和原理以及隔离级别？

在使用Spring框架时，可以有两种使用事务的方式，一种是编程式的，一种是声明式的，`@Transactional`注解就是声明式的。

首先，事务这个概念是数据库层面的，Spring只是基于数据库中的事务进行了扩展，以及提供了一些能让程序员更加方便操作事务的方式。

比如我们可以通过在某个方法上增加`@Transactional`注解，就可以开启事务，这个方法中所有的sql都会在一个事务中执行，统一成功或失败。

在一个方法上加了`@Transactional`注解后，Spring会基于这个类生成一个代理对象，会将这个代理对象作为bean，当在使用这个代理对象的方法时，如果这个方法上存在`@Transactional`注解，那么代理逻辑会先把事务的自动提交设置为false，然后再去执行原本的业务逻辑方法，如果执行业务逻辑方法没有出现异常，那么代理逻辑中就会将事务进行提交，如果执行业务逻辑方法出现了异常，那么则会将事务进行回滚。

当然，针对哪些异常回滚事务是可以配置的，可以利用`@Transactional`注解中的`rollbackFor`属性进行配置，默认情况下会对`RuntimeException`和`Error`进行回滚。

spring事务隔离级别就是数据库的隔离级别：外加一个默认级别

- `read uncommitted` (未提交读)
- `read committed` (提交读、不可重复读)
- `repeatable read` (可重复读)
- `serializable` (可串行化)

[Plain Text](#) | [Copy Code](#)

- 1 数据库的配置隔离级别是`Read Committed`，而Spring配置的隔离级别是`Repeatable Read`，请问这时隔离级别是以哪一个为准？
- 2 以Spring配置的为准，如果spring设置的隔离级别数据库不支持，效果取决于数据库

## Spring事务传播机制

多个事务方法相互调用时，事务如何在这些方法间传播，方法A是一个事务的方法，方法A执行过程中调用了方法B，那么方法B有无事务以及方法B对事务的要求不同都会对方法A的事务具体执行造成影响，同时方法A的事务对方法B的事务执行也有影响，这种影响具体是什么就由两个方法所定义的事务传播类型所决定。

1. REQUIRED(Spring默认的事务传播类型)：如果当前没有事务，则自己新建一个事务，如果当前存在事务，则加入这个事务
2. SUPPORTS：当前存在事务，则加入当前事务，如果当前没有事务，就以非事务方法执行
3. MANDATORY：当前存在事务，则加入当前事务，如果当前事务不存在，则抛出异常。
4. REQUIRES\_NEW：创建一个新事务，如果存在当前事务，则挂起该事务。
5. NOT\_SUPPORTED：以非事务方式执行,如果当前存在事务，则挂起当前事务
6. NEVER：不使用事务，如果当前事务存在，则抛出异常
7. NESTED：如果当前事务存在，则在嵌套事务中执行，否则REQUIRED的操作一样（开启一个事务）

## Spring事务什么时候会失效？

spring事务的原理是AOP，进行了切面增强，那么失效的根本原因是这个AOP不起作用了！常见情况有如下几种

- 1、发生自调用，类里面使用this调用本类的方法（this通常省略），此时这个this对象不是代理类，而是UserService对象本身！  
解决方法很简单，让那个this变成UserService的代理类即可！
- 2、方法不是public的：@Transactional 只能用于 public 的方法上，否则事务不会失效，如果要用在非 public 方法上，可以开启 AspectJ 代理模式。
- 3、数据库不支持事务
- 4、没有被spring管理
- 5、异常被吃掉，事务不会回滚(或者抛出的异常没有被定义，默认为RuntimeException)

## 什么是bean的自动装配，有哪些方式？

开启自动装配，只需要在xml配置文件中定义“autowire”属性。

```
1 <bean id="cutomer" class="com.xxx.xxx.Customer" autowire="" />
```

Plain Text |  Copy Code

autowire属性有五种装配的方式：

- no – 缺省情况下，自动配置是通过“ref”属性手动设定。

[Plain Text](#)[Copy Code](#)

- 1 手动装配：以value或ref的方式明确指定属性值都是手动装配。
- 2 需要通过'ref'属性来连接bean。

- byName-根据bean的属性名称进行自动装配。

[Plain Text](#)[Copy Code](#)

- 1 Cutomer的属性名称是person，Spring会将bean id为person的bean通过setter方法进行自动装配。
- 2 <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="byName"/>
- 3 <bean id="person" class="com.xxx.xxx.Person"/>

- byType-根据bean的类型进行自动装配。

[Plain Text](#)[Copy Code](#)

- 1 Cutomer的属性person的类型为Person，Spring会将Person类型通过setter方法进行自动装配。
- 2 <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="byType"/>
- 3 <bean id="person" class="com.xxx.xxx.Person"/>

- constructor-类似byType，不过是应用于构造器的参数。如果一个bean与构造器参数的类型形同，则进行自动装配，否则导致异常。

[Plain Text](#)[Copy Code](#)

- 1 Cutomer构造函数的参数person的类型为Person，Spring会将Person类型通过构造方法进行自动装配。
- 2 <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="constructor"/>
- 3 <bean id="person" class="com.xxx.xxx.Person"/>

- autodetect-如果有默认的构造器，则通过constructor方式进行自动装配，否则使用byType方式进行自动装配。

[Plain Text](#)[Copy Code](#)

- 1 如果有默认的构造器，则通过constructor方式进行自动装配，否则使用byType方式进行自动装配。

@Autowired自动装配bean，可以在字段、setter方法、构造函数上使用。

## Spring中的Bean创建的生命周期有哪些步骤

Spring中一个Bean的创建大概分为以下几个步骤：

1. 推断构造方法
2. 实例化
3. 填充属性，也就是依赖注入
4. 处理Aware回调
5. 初始化前，处理@PostConstruct注解
6. 初始化，处理InitializingBean接口
7. 初始化后，进行AOP

当然其实真正的步骤更加细致，可以看下面的流程图

## Spring中Bean是线程安全的吗

Spring本身并没有针对Bean做线程安全的处理，所以：

1. 如果Bean是无状态的，那么Bean则是线程安全的
2. 如果Bean是有状态的，那么Bean则不是线程安全的

另外，Bean是不是线程安全，跟Bean的作用域没有关系，Bean的作用域只是表示Bean的生命周期范围，对于任何生命周期的Bean都是一个对象，这个对象是不是线程安全的，还是得看这个Bean对象本身。

## ApplicationContext和BeanFactory有什么区别

BeanFactory是Spring中非常核心的组件，表示Bean工厂，可以生成Bean，维护Bean，而ApplicationContext继承了BeanFactory，所以ApplicationContext拥有BeanFactory所有的特点，也是一个Bean工厂，但是ApplicationContext除开继承了BeanFactory之外，还继承了诸如EnvironmentCapable、MessageSource、ApplicationEventPublisher等接口，从而ApplicationContext还有获取系统环境变量、国际化、事件发布等功能，这是BeanFactory所不具备的

## Spring中的事务是如何实现的

1. Spring事务底层是基于数据库事务和AOP机制的
2. 首先对于使用了@Transactional注解的Bean，Spring会创建一个代理对象作为Bean
3. 当调用代理对象的方法时，会先判断该方法上是否加了@Transactional注解

4. 如果加了，那么则利用事务管理器创建一个数据库连接
5. 并且修改数据库连接的autocommit属性为false，禁止此连接的自动提交，这是实现Spring事务非常重要的一步
6. 然后执行当前方法，方法中会执行sql
7. 执行完当前方法后，如果没有出现异常就直接提交事务
8. 如果出现了异常，并且这个异常是需要回滚的就会回滚事务，否则仍然提交事务
9. Spring事务的隔离级别对应的就是数据库的隔离级别
10. Spring事务的传播机制是Spring事务自己实现的，也是Spring事务中最复杂的
11. Spring事务的传播机制是基于数据库连接来做的，一个数据库连接一个事务，如果传播机制配置为需要新开一个事务，那么实际上就是先建立一个数据库连接，在此新数据库连接上执行sql

## Spring中什么时候@Transactional会失效

因为Spring事务是基于代理来实现的，所以某个加了@Transactional的方法只有是被代理对象调用时，那么这个注解才会生效，所以如果是被代理对象来调用这个方法，那么@Transactional是不会失效的。

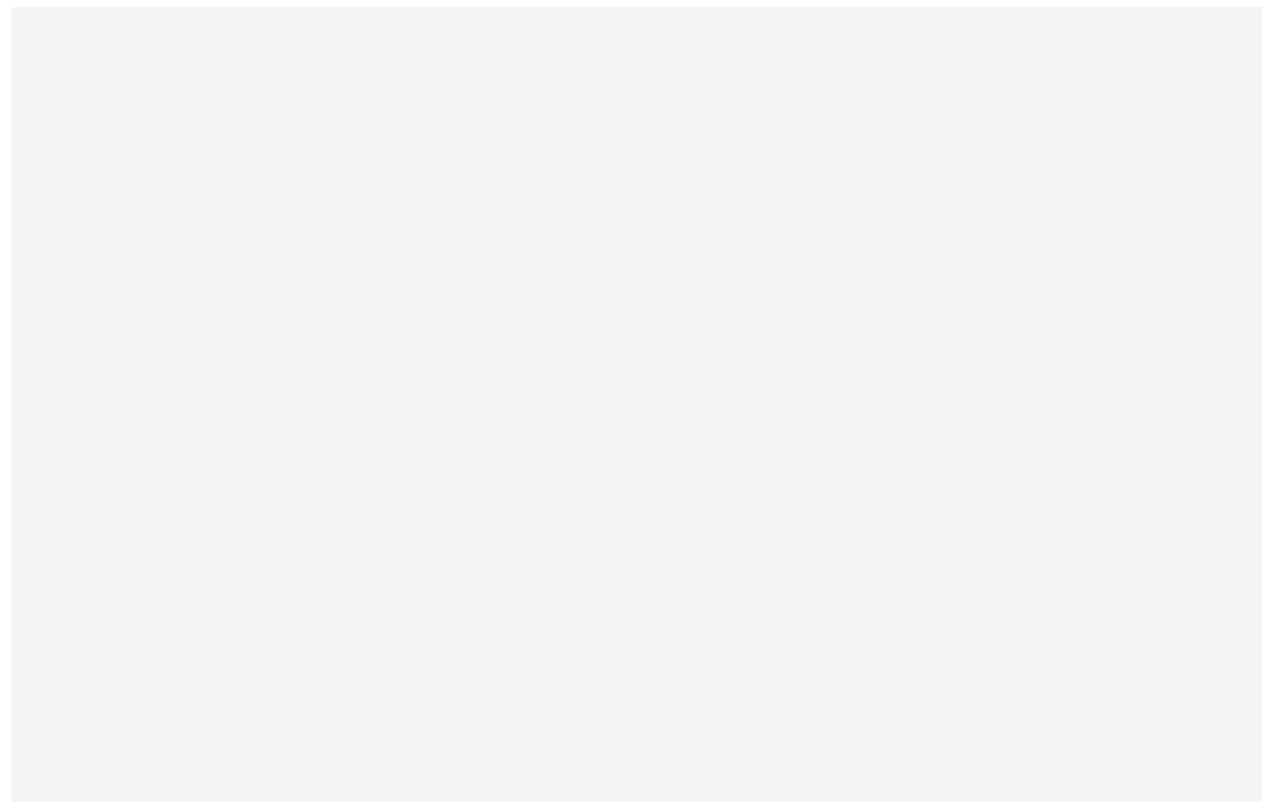
同时如果某个方法是private的，那么@Transactional也会失效，因为底层cglib是基于父子类来实现的，子类是不能重载父类的private方法的，所以无法很好的利用代理，也会导致@Transactional失效

## Spring容器启动流程是怎样的

1. 在创建Spring容器，也就是启动Spring时：
2. 首先会进行扫描，扫描得到所有的BeanDefinition对象，并存在一个Map中
3. 然后筛选出非懒加载的单例BeanDefinition进行创建Bean，对于多例Bean不需要在启动过程中去进行创建，对于多例Bean会在每次获取Bean时利用BeanDefinition去创建
4. 利用BeanDefinition创建Bean就是Bean的创建生命周期，这期间包括了合并BeanDefinition、推断构造方法、实例化、属性填充、初始化前、初始化、初始化后等步骤，其中AOP就是发生在初始化后这一步骤中
5. 单例Bean创建完了之后，Spring会发布一个容器启动事件

6. Spring启动结束
7. 在源码中会更复杂，比如源码中会提供一些模板方法，让子类来实现，比如源码中还涉及到一些 BeanFactoryPostProcessor和BeanPostProcessor的注册，Spring的扫描就是通过 BeanFactoryPostProcessor来实现的，依赖注入就是通过BeanPostProcessor来实现的
8. 在Spring启动过程中还会去处理@Import等注解

## Spring用到了哪些设计模式



## Spring Boot、Spring MVC 和 Spring 有什么区别

spring是一个IOC容器，用来管理Bean，使用依赖注入实现控制反转，可以很方便的整合各种框架，提供AOP机制弥补OOP的代码重复问题、更方便将不同类不同方法中的共同处理抽取成切面、自动注入给方法执行，比如日志、异常等

springmvc是spring对web框架的一个解决方案，提供了一个总的前端控制器Servlet，用来接收请求，然后定义了一套路由策略（url到handle的映射）及适配执行handle，将handle结果使用视图解析技术生成视图展现给前端

springboot是spring提供的一个快速开发工具包，让程序员能更方便、更快速的开发spring+springmvc应用，简化了配置（约定了默认配置），整合了一系列的解决方案（starter机制）、redis、mongodb、es，可以开箱即用

## Spring MVC 工作流程

1) 用户发送请求至前端控制器 DispatcherServlet。

2) DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3) 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

- 4) DispatcherServlet 调用 HandlerAdapter 处理器适配器。
- 5) HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)
- 6) Controller 执行完成返回 ModelAndView。
- 7) HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。8) DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。
- 9) ViewReslover 解析后返回具体 View。
- 10) DispatcherServlet 根据 View 进行渲染视图 (即将模型数据填充至视图中)。
- 11) DispatcherServlet 响应用户。

## Spring MVC的主要组件？

Handler：也就是处理器。它直接应对着MVC中的C也就是Controller层，它的具体表现形式有很多，可以是类，也可以是方法。在Controller层中@RequestMapping标注的所有方法都可以看成是一个Handler，只要可以实际处理请求就可以是Handler

### 1、HandlerMapping

initHandlerMappings(context)，处理器映射器，根据用户请求的资源uri来查找Handler的。在SpringMVC中会有很多请求，每个请求都需要一个Handler处理，具体接收到一个请求之后使用哪个Handler进行，这就是HandlerMapping需要做的事。

### 2、HandlerAdapter

initHandlerAdapters(context)，适配器。因为SpringMVC中的Handler可以是任意的形式，只要能处理请求就ok，但是Servlet需要的处理方法的结构却是固定的，都是以request和response为参数的方法。如何让固定的Servlet处理方法调用灵活的Handler来进行处理呢？这就是HandlerAdapter要做的事情。

Handler是用来干活的工具；HandlerMapping用于根据需要干的活找到相应的工具；HandlerAdapter是使用工具干活的人。

### 3、HandlerExceptionResolver

initHandlerExceptionResolvers(context)，其它组件都是用来干活的。在干活的过程中难免会出现问题，出问题后怎么办呢？这就需要有一个专门的角色对异常情况进行处理，在SpringMVC中就是HandlerExceptionResolver。具体来说，此组件的作用是根据异常设置 ModelAndView，之后再交给render方法进行渲染。

#### 4、ViewResolver

initViewResolvers(context) , ViewResolver用来将String类型的视图名和Locale解析为View类型的视图。View是用来渲染页面的，也就是将程序返回的参数填入模板里，生成html（也可能是其它类型）文件。这里就有两个关键问题：使用哪个模板？用什么技术（规则）填入参数？这其实是ViewResolver主要的工作，ViewResolver需要找到渲染所用的模板和所用的技术（也就是视图的类型）进行渲染，具体的渲染过程则交由不同的视图自己完成。

#### 5、RequestToViewNameTranslator

initRequestToViewNameTranslator(context) , ViewResolver是根据ViewName查找View，但有的Handler处理完后并没有设置View也没有设置ViewName，这时就需要从request获取ViewName了，如何从request中获取ViewName就是RequestToViewNameTranslator要做的事情了。RequestToViewNameTranslator在Spring MVC容器里只可以配置一个，所以所有request到ViewName的转换规则都要在一个Translator里面全部实现。

#### 6、LocaleResolver

initLocaleResolver(context) , 解析视图需要两个参数：一是视图名，另一个是Locale。视图名是处理器返回的，Locale是从哪里来的？这就是LocaleResolver要做的事情。LocaleResolver用于从request解析出Locale，Locale就是zh-cn之类，表示一个区域，有了这个就可以对不同区域的用户显示不同的结果。SpringMVC主要有两个地方用到了Locale：一是ViewResolver视图解析的时候；二是用到国际化资源或者主题的时候。

#### 7、ThemeResolver

initThemeResolver(context) , 用于解析主题。SpringMVC中一个主题对应一个properties文件，里面存放着跟当前主题相关的所有资源、如图片、css样式等。SpringMVC的主题也支持国际化，同一个主题不同区域也可以显示不同的风格。SpringMVC中跟主题相关的类有ThemeResolver、ThemeSource和Theme。主题是通过一系列资源来具体体现的，要得到一个主题的资源，首先要得到资源的名称，这是ThemeResolver的工作。然后通过主题名称找到对应的主题（可以理解为一个配置）文件，这是ThemeSource的工作。最后从主题中获取资源就可以了。

#### 8、MultipartResolver

initMultipartResolver(context) , 用于处理上传请求。处理方法是将普通的request包装成MultipartHttpServletRequest，后者可以直接调用getFile方法获取File，如果上传多个文件，还可以调用getFileMap得到FileName->File结构的Map。此组件中一共有三个方法，作用分别是判断是不是上传请求，将request包装成MultipartHttpServletRequest、处理完后清理上传过程中产生的临时资源。

#### 9、FlashMapManager

initFlashMapManager(context) , 用来管理FlashMap的，FlashMap主要用在redirect中传递参数。

## Spring Boot 自动配置原理？

@Import + @Configuration + Spring spi

自动配置类由各个starter提供，使用@Configuration + @Bean定义配置类，放到META-INF/spring.factories下

使用Spring spi扫描META-INF/spring.factories下的配置类

使用@Import导入自动配置类

## 如何理解 Spring Boot 中的 Starter

使用spring + springmvc使用，如果需要引入mybatis等框架，需要到xml中定义mybatis需要的bean

starter就是定义一个starter的jar包，写一个@Configuration配置类、将这些bean定义在里面，然后在starter包的META-INF/spring.factories中写入该配置类，springboot会按照约定来加载该配置类

开发人员只需要将相应的starter包依赖进应用，进行相应的属性配置（使用默认配置时，不需要配置），就可以直接进行代码开发，使用对应的功能了，比如mybatis-spring-boot-starter，spring-boot-starter-redis

## 什么是嵌入式服务器？为什么要使用嵌入式服务器？

节省了下载安装tomcat，应用也不需要再打war包，然后放到webapp目录下再运行

只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了

springboot已经内置了tomcat.jar，运行main方法时会去启动tomcat，并利用tomcat的spi机制加载springmvc

## Spring Boot中常用注解及其底层实现

1. @SpringBootApplication注解：这个注解标识了一个SpringBoot工程，它实际上是另外三个注解的组合，这三个注解是：
  - a. @SpringBootConfiguration：这个注解实际就是一个@Configuration，表示启动类也是一个配置类
  - b. @EnableAutoConfiguration：向Spring容器中导入了一个Selector，用来加载ClassPath下SpringFactories中所定义的自动配置类，将这些自动加载为配置Bean
  - c. @ComponentScan：标识扫描路径，因为默认是没有配置实际扫描路径，所以SpringBoot扫描的路径是启动类所在的当前目录
2. @Bean注解：用来定义Bean，类似于XML中的<bean>标签，Spring在启动时，会对加了@Bean注解的方法进行解析，将方法的名字做为beanName，并通过执行方法得到bean对象

3. @Controller、@Service、@ResponseBody、@Autowired都可以说

## Spring Boot是如何启动Tomcat的

1. 首先，SpringBoot在启动时会先创建一个Spring容器
2. 在创建Spring容器过程中，会利用@ConditionalOnClass技术来判断当前classpath中是否存在Tomcat依赖，如果存在则会生成一个启动Tomcat的Bean
3. Spring容器创建完之后，就会获取启动Tomcat的Bean，并创建Tomcat对象，并绑定端口等，然后启动Tomcat

## Spring Boot中配置文件的加载顺序是怎样的？

优先级从高到低，高优先级的配置覆盖低优先级的配置，所有配置会形成互补配置。

1. 命令行参数。所有的配置都可以在命令行上进行指定；
2. Java系统属性 (System.getProperties()) ；
3. 操作系统环境变量 ；
4. jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件
5. jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件 再来加载不带profile
6. jar包外部的application.properties或application.yml(不带spring.profile)配置文件
7. jar包内部的application.properties或application.yml(不带spring.profile)配置文件
8. @Configuration注解类上的@PropertySource

## Mybatis的优缺点

优点：

1. 基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。
2. 与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；
3. 很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要JDBC 支持的数据库 MyBatis 都支持）。
4. 能够与 Spring 很好的集成；
5. 提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

缺点：

1. SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL 语句的功底有一定要求。
2. SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

# MyBatis 与 Hibernate 有哪些不同？

SQL 和 ORM 的争论，永远都不会终止

开发速度的对比：

Hibernate的真正掌握要比Mybatis难些。Mybatis框架相对简单很容易上手，但也相对简陋些。

比起两者的开发速度，不仅仅要考虑到两者的特性及性能，更要根据项目需求去考虑究竟哪一个更适合项目开发，比如：一个项目中用到的复杂查询基本没有，就是简单的增删改查，这样选择hibernate效率就很快了，因为基本的sql语句已经被封装好了，根本不需要你去写sql语句，这就节省了大量的时间，但是对于一个大型项目，复杂语句较多，这样再去选择hibernate就不是一个太好的选择，选择mybatis就会加快许多，而且语句的管理也比较方便。

开发工作量的对比：

Hibernate和MyBatis都有相应的代码生成工具。可以生成简单基本的DAO层方法。针对高级查询，Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于业务流程

sql优化方面：

Hibernate的查询会将表中的所有字段查询出来，这一点会有性能消耗。Hibernate也可以自己写SQL来指定需要查询的字段，但这样就破坏了Hibernate开发的简洁性。而Mybatis的SQL是手动编写的，所以可以按需求指定查询的字段。

Hibernate HQL语句的调优需要将SQL打印出来，而Hibernate的SQL被很多人嫌弃因为太丑了。MyBatis的SQL是自己手动写的所以调整方便。但Hibernate具有自己的日志统计。Mybatis本身不带日志统计，使用Log4j进行日志记录。

对象管理的对比：

Hibernate 是完整的对象/关系映射解决方案，它提供了对象状态管理 (state management) 的功能，使开发者不再需要理会底层数据库系统的细节。也就是说，相对于常见的 JDBC/SQL 持久层方案中需要管理 SQL 语句，Hibernate采用了更自然的面向对象的视角来持久化 Java 应用中的数据。

换句话说，使用 Hibernate 的开发者应该总是关注对象的状态 (state)，不必考虑 SQL 语句的执行。这部分细节已经由 Hibernate 掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。而MyBatis在这一块没有文档说明，用户需要对对象自己进行详细的管理。

缓存机制对比：

相同点：都可以实现自己的缓存或使用其他第三方缓存方案，创建适配器来完全覆盖缓存行为。

不同点：Hibernate的二级缓存配置在SessionFactory生成的配置文件中进行详细配置，然后再在具体的表-对象映射中配置是哪种缓存。

MyBatis的二级缓存配置都是在每个具体的表-对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓存机制。并且Mybatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。

两者比较：因为Hibernate对查询对象有着良好的管理机制，用户无需关心SQL。所以在使用二级缓存时如果出现脏数据，系统会报出错误并提示。

而MyBatis在这一方面，使用二级缓存时需要特别小心。如果不能完全确定数据更新操作的波及范围，避免Cache的盲目使用。否则，脏数据的出现会给系统的正常运行带来很大的隐患。

Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

## #{}和\${}的区别是什么？

#{}是预编译处理、是占位符， \${}是字符串替换、是拼接符。

Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 来赋值；

Mybatis 在处理时，就是把{}替换成变量的值，调用 Statement 来赋值；

#{} 的变量替换是在DBMS 中、变量替换后，#{} 对应的变量自动加上单引号，{} 的变量替换是在 DBMS 外、变量替换后，{} 对应的变量不会加上单引号

使用#{}可以有效的防止 SQL 注入， 提高系统安全性。

## 简述 Mybatis 的插件运行原理，如何编写一个插件。

Mybatis 只支持针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 使用 JDK 的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke() 方法，拦截那些你指定需要拦截的方法。

编写插件：实现 Mybatis 的 Interceptor 接口并复写 intercept() 方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，在配置文件中配置编写的插件。

Java | Copy Code

```
1 @Intercepts({@Signature(type = StatementHandler.class, method = "query", args =
{Statement.class, ResultHandler.class}),
2           @Signature(type = StatementHandler.class, method = "update", args =
{Statement.class}),
3           @Signature(type = StatementHandler.class, method = "batch", args = {
4             Statement.class })})
4 @Component
5
6
7 invocation.proceed() 执行具体的业务逻辑
```

## Mysql(20)

### 索引的基本原理

索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

索引的原理：就是把无序的数据变成有序的查询

1. 把创建了索引的列的内容进行排序
2. 对排序结果生成倒排表
3. 在倒排表内容上拼上数据地址链
4. 在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

### Mysql聚簇和非聚簇索引的区别

都是B+树的数据结构

- 聚簇索引：将数据存储与索引放到了一块、并且是按照一定的顺序组织的，找到索引也就找到了数据，数据的物理存放顺序与索引顺序是一致的，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的
- 非聚簇索引：叶子节点不存储数据、存储的是数据行地址，也就是说根据索引查找到数据行的位置再取磁盘查找数据，这个就有点类似一本树的目录，比如我们要找第三章第一节，那我们先在这个目录里面找，找到对应的页码后去对应的页码看文章。

优势：1、查询通过聚簇索引可以直接获取数据，相比非聚簇索引需要第二次查询（非覆盖索引的情况下）效率要高

2、聚簇索引对于范围查询的效率很高，因为其数据是按照大小排列的

3、聚簇索引适合用在排序的场合，非聚簇索引不适合

劣势：

1、维护索引很昂贵，特别是插入新行或者主键被更新导致要分页(page split)的时候。建议在大量插入新行后，选在负载较低的时间段，通过OPTIMIZE TABLE优化表，因为必须被移动的行数据可能造成碎片。使用独享表空间可以弱化碎片

2、表因为使用UUID（随机ID）作为主键，使数据存储稀疏，这就会出现聚簇索引有可能有比全表扫描更慢，所以建议使用int的auto\_increment作为主键

3、如果主键比较大的话，那辅助索引将会变的更大，因为辅助索引的叶子存储的是主键值；过长的主键值，会导致非叶子节点占用更多的物理空间

InnoDB中一定有主键，主键一定是聚簇索引，不手动设置，则会使用unique索引，没有unique索引，则会使用数据库内部的一个行的隐藏id来当作主键索引。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

MyISAM使用的是非聚簇索引，没有聚簇索引，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。

如果涉及到大数据量的排序、全表扫描、count之类的操作的话，还是MyISAM占优势些，因为索引所占空间小，这些操作是需要在内存中完成的。

## Mysql索引的数据结构，各自优劣

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有Hash索引，B+树索引等，InnoDB存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为

单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

B+树：B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。因此，B+树索引被广泛应用于数据库、文件系统等场景。

哈希索引：哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快

如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；前提是键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应数据；

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；

哈希索引也没办法利用索引完成排序，以及like 'xxx%' 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；

哈希索引也不支持多列联合索引的最左匹配规则；

B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在哈希碰撞问题。

## 索引设计的原则？

查询更快、占用空间更小

1. 适合索引的列是出现在where子句中的列，或者连接子句中指定的列
2. 基数较小的表，索引效果较差，没有必要在此列建立索引
3. 使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间，如果搜索词超过索引前缀长度，则使用索引排除不匹配的行，然后检查其余行是否可能匹配。
4. 不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。
5. 定义有外键的数据列一定要建立索引。
6. 更新频繁字段不适合创建索引

7. 若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)
8. 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。
9. 对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。
10. 对于定义为text、image和bit的数据类型的列不要建立索引。

## InnoDB存储引擎的锁的算法

- Record lock : 单个行记录上的锁
- Gap lock : 间隙锁，锁定一个范围，不包括记录本身
- Next-key lock : record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb\_locks\_unsafe\_for\_binlog设置为1

## 关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

在业务系统中，除了使用主键进行的查询，其他的都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。

慢查询的优化首先要搞明白慢的原因是什么？是查询条件没有命中索引？是load了不需要的数据列？还是数据量太大？

所以优化也是针对这三个方向来的，

- 首先分析语句，看看是否load了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。
- 分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引。
- 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。

## 事务的基本特性和隔离级别

事务基本特性ACID分别是：

**原子性**指的是一个事务中的操作要么全部成功，要么全部失败。

**一致性**指的是数据库总是从一个一致性的状态转换到另外一个一致性的状态。比如A转账给B100块钱，假设A只有90块，支付之前我们数据库里的数据都是符合约束的，但是如果事务执行成功了，我们的数据库数据就破坏约束了，因此事务不能成功，这里我们说事务提供了一致性的保证

**隔离性**指的是一个事务的修改在最终提交前，对其他事务是不可见的。

**持久性**指的是一旦事务提交，所做的修改就会永久保存到数据库中。

隔离性有4个隔离级别，分别是：

- read uncommit** 读未提交，可能会读到其他事务未提交的数据，也叫做脏读。  
用户本来应该读取到id=1的用户age应该是10，结果读取到了其他事务还没有提交的事务，结果读取结果age=20，这就是脏读。
- read commit** 读已提交，两次读取结果不一致，叫做不可重复读。  
不可重复读解决了脏读的问题，他只会读取已经提交的事务。  
用户开启事务读取id=1用户，查询到age=10，再次读取发现结果=20，在同一个事务里同一个查询读取到不同的结果叫做不可重复读。
- repeatable read** 可重复复读，这是mysql的默认级别，就是每次读取结果都一样，但是有可能产生幻读。
- serializable** 串行，一般是不会使用的，他会给每一行读取的数据加锁，会导致大量超时和锁竞争的问题。

**脏读(Dirty Read)**：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

**不可重复读(Non-repeatable read)**:在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

## ACID靠什么保证的？

A原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql

C一致性由其他三大特性保证、程序代码要保证业务上的一致性

I隔离性由MVCC来保证

D持久性由内存+redo log来保证，mysql修改数据同时在内存和redo log记录这次操作，宕机的时候可以从redo log恢复

Plain Text |  Copy Code

- 1 InnoDB redo log 写盘，InnoDB 事务进入 prepare 状态。
- 2 如果前面 prepare 成功，binlog 写盘，再继续将事务日志持久化到 binlog，如果持久化成功，那么 InnoDB 事务则进入 commit 状态(在 redo log 里面写一个 commit 记录)

redolog的刷盘会在系统空闲时进行

## 什么是MVCC

多版本并发控制：读取数据时通过一种类似快照的方式将数据保存下来，这样读锁就和写锁不冲突了，不同的事务session会看到自己特定版本的数据，版本链

MVCC只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作。其他两个隔离级别够和MVCC不兼容，因为 READ UNCOMMITTED 总是读取最新的数据行，而不是符合当前事务版本的数据行。而 SERIALIZABLE 则会对所有读取的行都加锁。

聚簇索引记录中有两个必要的隐藏列：

**trx\_id**：用来存储每次对某条聚簇索引记录进行修改的时候的事务id。

**roll\_pointer**：每次对哪条聚簇索引记录有修改的时候，都会把老版本写入undo日志中。这个roll\_pointer就是存了一个指针，它指向这条聚簇索引记录的上一个版本的位置，通过它来获得上一个版本的记录信息。(注意插入操作的undo日志没有这个属性，因为它没有老版本)

已提交读和可重复读的区别就在于它们生成ReadView的策略不同。

开始事务时创建readview，readView维护当前活动的事务id，即未提交的事务id，排序生成一个数组

访问数据，获取数据中的事务id（获取的是事务id最大的记录），对比readview：

如果在readview的左边（比readview都小），可以访问（在左边意味着该事务已经提交）

如果在readview的右边（比readview都大）或者就在readview中，不可以访问，获取roll\_pointer，取上一版本重新对比（在右边意味着，该事务在readview生成之后出现，在readview中意味着该事务还未提交）

已提交读隔离级别的事务在每次查询的开始都会生成一个独立的ReadView，而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView。

这就是Mysql的MVCC，通过版本链，实现多版本，可并发读-写，写-读。通过ReadView生成策略的不同实现不同的隔离级别。

## 分表后非sharding\_key的查询怎么处理，分表后的排序？

1. 可以做一个mapping表，比如这时候商家要查询订单列表怎么办呢？不带user\_id查询的话你总不能扫全表吧？所以我们可以做一个映射关系表，保存商家和用户的关系，查询的时候先通过商家查询到用户列表，再通过user\_id去查询。
2. 宽表，对数据实时性要求不是很高的场景，比如查询订单列表，可以把订单表同步到离线（实时）数仓，再基于数仓去做成一张宽表，再基于其他如es提供查询服务。
3. 数据量不是很大的话，比如后台的一些查询之类的，也可以通过多线程扫表，然后再聚合结果的方式来做。或者异步的形式也是可以的。

union

排序字段是唯一索引：

- 首先第一页的查询：将各表的结果集进行合并，然后再次排序
- 第二页及以后的查询，需要传入上一页排序字段的最后一个值，及排序方式。
- 根据排序方式，及这个值进行查询。如排序字段date，上一页最后值为3，排序方式降序。查询的时候sql为 select ... from table where date < 3 order by date desc limit 0,10。这样再将几个表的结果合并排序即可。

## Mysql主从同步原理

mysql主从同步的过程：

Mysql的主从复制中主要有三个线程：`master (binlog dump thread)`、`slave (I/O thread、SQL thread)`，Master一条线程和Slave中的两条线程。

- 主节点 binlog，主从复制的基础是主库记录数据库的所有变更记录到 binlog。binlog 是数据库服务器启动的那一刻起，保存所有修改数据库结构或内容的一个文件。
- 主节点 log dump 线程，当 binlog 有变动时，log dump 线程读取其内容并发送给从节点。
- 从节点 I/O 线程接收 binlog 内容，并将其写入到 relay log 文件中。
- 从节点的SQL 线程读取 relay log 文件内容对数据更新进行重放，最终保证主从数据库的一致性。

注：主从节点使用 binglog 文件 + position 偏移量来定位主从同步的位置，从节点会保存其已接收到的偏移量，如果从节点发生宕机重启，则会自动从 position 的位置发起同步。

由于mysql默认的复制方式是异步的，主库把日志发送给从库后不关心从库是否已经处理，这样会产生一个问题就是假设主库挂了，从库处理失败了，这时候从库升为主库后，日志就丢失了。由此产生两个概念。

## 全同步复制

主库写入binlog后强制同步日志到从库，所有的从库都执行完成后才返回给客户端，但是很显然这种方式的话性能会受到严重影响。

## 半同步复制

和全同步不同的是，半同步复制的逻辑是这样，从库写入日志成功后返回ACK确认给主库，主库收到至少一个从库的确认就认为写操作完成。

## 简述MyISAM和InnoDB的区别

### MyISAM：

- 不支持事务，但是每次查询都是原子的；
- 支持表级锁，即每次操作是对整个表加锁；
- 存储表的总行数；
- 一个MYISAM表有三个文件：索引文件、表结构文件、数据文件；
- 采用非聚集索引，索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。

### InnoDB：

- 支持ACID的事务，支持事务的四种隔离级别；
- 支持行级锁及外键约束：因此可以支持写并发；
- 不存储总行数；
- 一个InnoDB引擎存储在一个文件空间（共享表空间，表大小不受操作系统控制，一个表可能分布在多个文件里），也有可能为多个（设置为独立表空，表大小受操作系统文件大小限制，一般为2G），受操作系统文件大小的限制；
- 主键索引采用聚集索引（索引的数据域存储数据文件本身），辅索引的数据域存储主键的值；因此从辅索引查找数据，需要先通过辅索引找到主键值，再访问辅索引；最好使用自增主键，防止插入数据时，为维持B+树结构，文件的大调整。

## 简述Mysql中索引类型及对数据库的性能的影响

普通索引：允许被索引的数据列包含重复的值。

唯一索引：可以保证数据记录的唯一性。

主键：是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。

联合索引：索引可以覆盖多个数据列，如像INDEX(columnA, columnB)索引。

全文索引：通过建立 倒排索引，可以极大的提升检索效率，解决判断字段是否包含的问题，是目前搜索引擎使用的一种关键技术。可以通过ALTER TABLE table\_name ADD FULLTEXT (column);创建全文索引

索引可以极大的提高数据的查询速度。

通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件

索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大，如果非聚集索引很多，一旦聚集索引改变，那么所有非聚集索引都会跟着变。

## Explain语句结果中各个字段分表表示什么

列名	描述
id	查询语句中每出现一个SELECT关键字，MySQL就会为它分配一个唯一的id值，某些子查询会被优化为join查询，那么出现的id会一样
select_type	SELECT关键字对应的那个查询的类型
table	表名
partitions	匹配的分区信息
type	针对单表的查询方式（全表扫描、索引）
possible_keys	可能用到的索引
key	实际上使用的索引
key_len	实际使用到的索引长度
ref	当使用索引列等值查询时，与索引列进行等值匹配的对象信息
rows	预估的需要读取的记录条数
filtered	某个表经过搜索条件过滤后剩余记录条数的百分比
Extra	一些额外的信息，比如排序等

## 索引覆盖是什么

索引覆盖就是一个SQL在执行时，可以利用索引来快速查找，并且此SQL所要查询的字段在当前索引对应的字段中都包含了，那么就表示此SQL走完索引后不用回表了，所需要的字段都在当前索引的叶子节点上存在，可以直接作为结果返回了

## 最左前缀原则是什么

当一个SQL想要利用索引时，就一定要提供该索引所对应的字段中最左边的字段，也就是排在最前面的字段，比如针对a,b,c三个字段建立了一个联合索引，那么在写一个sql时就一定要提供a字段的条件，这样才能用到联合索引，这是由于在建立a,b,c三个字段的联合索引时，底层的B+树是按照a,b,c三个字段从左往右去比较大小进行排序的，所以如果想要利用B+树进行快速查找也得符合这个规则

## Innodb是如何实现事务的

Innodb通过Buffer Pool，LogBuffer，Redo Log，Undo Log来实现事务，以一个update语句为例：

1. Innodb在收到一个update语句后，会先根据条件找到数据所在的页，并将该页缓存在Buffer Pool中

2. 执行update语句，修改Buffer Pool中的数据，也就是内存中的数据
3. 针对update语句生成一个RedoLog对象，并存入LogBuffer中
4. 针对update语句生成undolog日志，用于事务回滚
5. 如果事务提交，那么则把RedoLog对象进行持久化，后续还有其他机制将Buffer Pool中所修改的数据页持久化到磁盘中
6. 如果事务回滚，则利用undolog日志进行回滚

## B树和B+树的区别，为什么Mysql使用B+树

B树的特点：

1. 节点排序
2. 一个节点了可以存多个元素，多个元素也排序了

B+树的特点：

1. 拥有B树的特点
2. 叶子节点之间有指针
3. 非叶子节点上的元素在叶子节点上都冗余了，也就是叶子节点中存储了所有的元素，并且排好顺序

Mysql索引使用的是B+树，因为索引是用来加快查询的，而B+树通过对数据进行排序所以是可以提高查询速度的，然后通过一个节点中可以存储多个元素，从而可以使得B+树的高度不会太高，在Mysql中一个Innodb页就是一个B+树节点，一个Innodb页默认16kb，所以一般情况下一颗两层的B+树可以存2000万行左右的数据，然后通过利用B+树叶子节点存储了所有数据并且进行了排序，并且叶子节点之间有指针，可以很好的支持全表扫描，范围查找等SQL语句。

## Mysql锁有哪些，如何理解

按锁粒度分类：

1. 行锁：锁某行数据，锁粒度最小，并发度高
2. 表锁：锁整张表，锁粒度最大，并发度低
3. 间隙锁：锁的是一个区间

还可以分为：

1. 共享锁：也就是读锁，一个事务给某行数据加了读锁，其他事务也可以读，但是不能写
2. 排它锁：也就是写锁，一个事务给某行数据加了写锁，其他事务不能读，也不能写

还可以分为：

1. 乐观锁：并不会真正的去锁某行记录，而是通过一个版本号来实现的
2. 悲观锁：上面所的行锁、表锁等都是悲观锁

在事务的隔离级别实现中，就需要利用锁来解决幻读

## Mysql慢查询该如何优化？

1. 检查是否走了索引，如果没有则优化SQL利用索引
2. 检查所利用的索引，是否是最优索引
3. 检查所查字段是否都是必须的，是否查询了过多字段，查出了多余数据
4. 检查表中数据是否过多，是否应该进行分库分表了
5. 检查数据库实例所在机器的性能配置，是否太低，是否可以适当增加资源

## Redis(14)

### 什么是RDB和AOF

RDB : Redis DataBase，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。

#### 优点：

1. 整个Redis数据库将只包含一个文件 dump.rdb，方便持久化。
2. 容灾性好，方便备份。
3. 性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能
4. 相对于数据集大时，比 AOF 的启动效率更高。

#### 缺点：

1. 数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)
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 : Append Only File，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

#### 优点：

1. 数据安全，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。。

2. 通过 append 模式写文件，即使中途服务器宕机也不会破坏已经存在的内容，可以通过 redis-check-aof 工具解决数据一致性问题。
3. AOF 机制的 rewrite 模式。定期对AOF文件进行重写，以达到压缩的目的

缺点：

1. AOF 文件比 RDB 文件大，且恢复速度慢。
2. 数据集大的时候，比 rdb 启动效率低。
3. 运行效率没有RDB高

AOF文件比RDB更新频率高，优先使用AOF还原数据，AOF比RDB更安全也更大，RDB性能比AOF好，如果两个都配了优先加载AOF。

## Redis的过期键的删除策略

Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

- **惰性过期**：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- **定期过期**：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

## Redis线程模型、单线程快的原因

Redis基于Reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器 file event handler。这个文件事件处理器，它是单线程的，所以 Redis 才叫做单线程的模型，它采用IO多路复用机制来同时监听多个Socket，根据Socket上的事件类型来选择对应的事件处理器来处理这个事件。可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了 Redis 内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个Socket、IO多路复用程序、文件事件分派器以及事件处理器（命令请求处理器、命裔回复处理器、连接应答处理器等）。

多个 Socket 可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个 Socket，会将 Socket 放入一个队列中排队，每次从队列中取出一个 Socket 给事件分派器，事件分派器把 Socket 给对应的事件处理器。

然后一个 Socket 的事件处理完之后，IO多路复用程序才会将队列中的下一个 Socket 给事件分派器。文件事件分派器会根据每个 Socket 当前产生的事件，来选择对应的事件处理器来处理。

单线程快的原因：

- 1) 纯内存操作
- 2) 核心是基于非阻塞的IO多路复用机制
- 3) 单线程反而避免了多线程的频繁上下文切换带来的性能问题

## 简述Redis事务实现

### 1、事务开始

**MULTI**命令的执行，标识着一个事务的开始。**MULTI**命令会将客户端状态的 `flags` 属性中打开 `REDIS_MULTI` 标识来完成的。

### 2、命令入队

当一个客户端切换到事务状态之后，服务器会根据这个客户端发送来的命令来执行不同的操作。如果客户端发送的命令为 **MULTI**、**EXEC**、**WATCH**、**DISCARD** 中的一个，立即执行这个命令，否则将命令放入一个事务队列里面，然后向客户端返回 `QUEUED` 回复

- 如果客户端发送的命令为 **EXEC**、**DISCARD**、**WATCH**、**MULTI** 四个命令的其中一个，那么服务器立即执行这个命令。

- 如果客户端发送的是四个命令以外的其他命令，那么服务器并不立即执行这个命令。

首先检查此命令的格式是否正确，如果不正确，服务器会在客户端状态 (`redisClient`) 的 `flags` 属性关闭 `REDIS_MULTI` 标识，并且返回错误信息给客户端。

如果正确，将这个命令放入一个事务队列里面，然后向客户端返回 `QUEUED` 回复

事务队列是按照FIFO的方式保存入队的命令

### 3、事务执行

客户端发送 EXEC 命令，服务器执行 EXEC 命令逻辑。

- 如果客户端状态的 flags 属性不包含 REDIS\_MULTI 标识，或者包含 REDIS\_DIRTY\_CAS 或者 REDIS\_DIRTY\_EXEC 标识，那么就直接取消事务的执行。
- 否则客户端处于事务状态 (flags 有 REDIS\_MULTI 标识) ，服务器会遍历客户端的事务队列，然后执行事务队列中的所有命令，最后将返回结果全部返回给客户端；

redis 不支持事务回滚机制，但是它会检查每一个事务中的命令是否错误。

Redis 事务不支持检查那些程序员自己逻辑错误。例如对 String 类型的数据库键执行对 HashMap 类型的操作！

- WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set (CAS) 行为。可以监控一个或多个键，一旦其中有一个键被修改 (或删除) ，之后的事务就不会执行，监控一直持续到EXEC命令。
- MULTI命令用于开启一个事务，它总是返回OK。MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。
- EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nil。
- 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务，并且客户端会从事务状态中退出。
- UNWATCH命令可以取消watch对所有key的监控。

## Redis 主从复制的核心原理

通过执行slaveof命令或设置slaveof选项，让一个服务器去复制另一个服务器的数据。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

全量复制：

1. 主节点通过bgsave命令fork子进程进行RDB持久化，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的
2. 主节点通过网络将RDB文件发送给从节点，对主从节点的带宽都会带来很大的消耗
3. 从节点清空老数据、载入新RDB文件的过程是阻塞的，无法响应客户端的命令；如果从节点执行 bgrewriteaof，也会带来额外的消耗

部分复制：

1. 复制偏移量：执行复制的双方，主从节点，分别会维护一个复制偏移量offset
2. 复制积压缓冲区：主节点内部维护了一个固定长度的、先进先出(FIFO)队列 作为复制积压缓冲区，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。

3. 服务器运行ID(runid)：每个Redis节点，都有其运行ID，运行ID由节点在启动时自动生成，主节点会将自己的运行ID发送给从节点，从节点会将主节点的运行ID存起来。从节点Redis断开重连的时候，就是根据运行ID来判断同步的进度：

- 如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况)；
- 如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

## Redis有哪些数据结构？分别有哪些典型的应用场景？

Redis的数据结构有：

1. 字符串：可以用来做最简单的数据，可以缓存某个简单的字符串，也可以缓存某个json格式的字符串，Redis分布式锁的实现就利用了这种数据结构，还包括可以实现计数器、Session共享、分布式ID
2. 哈希表：可以用来存储一些key-value对，更适合用来存储对象
3. 列表：Redis的列表通过命令的组合，既可以当做栈，也可以当做队列来使用，可以用来缓存类似微信公众号、微博等消息流数据
4. 集合：和列表类似，也可以存储多个元素，但是不能重复，集合可以进行交集、并集、差集操作，从而可以实现类似，我和某人共同关注的人、朋友圈点赞等功能
5. 有序集合：集合是无序的，有序集合可以设置顺序，可以用来实现排行榜功能

## Redis分布式锁底层是如何实现的？

1. 首先利用setnx来保证：如果key不存在才能获取到锁，如果key存在，则获取不到锁
2. 然后还要利用lua脚本来保证多个redis操作的原子性
3. 同时还要考虑到锁过期，所以需要额外的一个看门狗定时任务来监听锁是否需要续约
4. 同时还要考虑到redis节点挂掉后的情况，所以需要采用红锁的方式来同时向N/2+1个节点申请锁，都申请到了才证明获取锁成功，这样就算其中某个redis节点挂掉了，锁也不能被其他客户端获取到

## Redis主从复制的核心原理

Redis的主从复制是提高Redis的可靠性的有效措施，主从复制的流程如下：

1. 集群启动时，主从库间会先建立连接，为全量复制做准备
2. 主库将所有数据同步给从库。从库收到数据后，在本地完成数据加载，这个过程依赖于内存快照RDB
3. 在主库将数据同步给从库的过程中，主库不会阻塞，仍然可以正常接收请求。否则，redis的服务就被中断了。但是，这些请求中的写操作并没有记录到刚刚生成的RDB文件中。为了保证主从库的数据一致性，主库会在内存中用专门的replication buffer，记录RDB文件生成收到的所有写操作。
4. 最后，也就是第三个阶段，主库会把第二阶段执行过程中新收到的写命令，再发送给从库。具体的操作是，当主库完成RDB文件发送后，就会把此时replication buffer中修改操作发送给从库，从库再执行这些操作。这样一来，主从库就实现同步了

5. 后续主库和从库都可以处理客户端读操作，写操作只能交给主库处理，主库接收到写操作后，还会将写操作发送给从库，实现增量同步

## Redis集群策略

Redis提供了三种集群策略：

1. 主从模式：这种模式比较简单，主库可以读写，并且会和从库进行数据同步，这种模式下，客户端直接连主库或某个从库，但是但主库或从库宕机后，客户端需要手动修改IP，另外，这种模式也比较难进行扩容，整个集群所能存储的数据受到某台机器的内存容量，所以不可能支持特大数据量
2. 哨兵模式：这种模式在主从的基础上新增了哨兵节点，但主库节点宕机后，哨兵会发现主库节点宕机，然后在从库中选择一个库作为新的主库，另外哨兵也可以做集群，从而可以保证但某一个哨兵节点宕机后，还有其他哨兵节点可以继续工作，这种模式可以比较好的保证Redis集群的高可用，但是仍然不能很好的解决Redis的容量上限问题。
3. Cluster模式：Cluster模式是用得比较多的模式，它支持多主多从，这种模式会按照key进行槽位的分配，可以使不同的key分散到不同的主节点上，利用这种模式可以使得整个集群支持更大的数据容量，同时每个主节点可以拥有自己的多个从节点，如果该主节点宕机，会从它的从节点中选举一个新的主节点。

对于这三种模式，如果Redis要存的数据量不大，可以选择哨兵模式，如果Redis要存的数据量大，并且需要持续的扩容，那么选择Cluster模式。

## 缓存穿透、缓存击穿、缓存雪崩分别是什么

缓存中存放的大多都是热点数据，目的就是防止请求可以直接从缓存中获取到数据，而不用访问Mysql。

1. 缓存雪崩：如果缓存中某一时刻大批热点数据同时过期，那么就可能导致大量请求直接访问Mysql了，解决办法就是在过期时间上增加一点随机值，另外如果搭建一个高可用的Redis集群也是防止缓存雪崩的有效手段
2. 缓存击穿：和缓存雪崩类似，缓存雪崩是大批热点数据失效，而缓存击穿是指某一个热点key突然失效，也导致了大量请求直接访问Mysql数据库，这就是缓存击穿，解决方案就是考虑这个热点key不设过期时间
3. 缓存穿透：假如某一时刻访问redis的大量key都在redis中不存在（比如黑客故意伪造一些乱七八糟的key），那么也会给数据造成压力，这就是缓存穿透，解决方案是使用布隆过滤器，它的作用就是如果它认为一个key不存在，那么这个key就肯定不存在，所以可以在缓存之前加一层布隆过滤器来拦截不存在的key

## Redis和Mysql如何保证数据一致

1. 先更新Mysql，再更新Redis，如果更新Redis失败，可能仍然不一致
2. 先删除Redis缓存数据，再更新Mysql，再次查询的时候在将数据添加到缓存中，这种方案能解决1方案的问题，但是在高并发下性能较低，而且仍然会出现数据不一致的问题，比如线程1删除了Redis缓存数据，正在更新Mysql，此时另外一个查询再查询，那么就会把Mysql中老数据又查到Redis中
3. 延时双删，步骤是：先删除Redis缓存数据，再更新Mysql，延迟几百毫秒再删除Redis缓存数据，这样就算在更新Mysql时，有其他线程读了Mysql，把老数据读到了Redis中，那么也会被删除掉，从而把数据保持一致

# Redis的持久化机制

RDB : Redis DataBase 将某一个时刻的内存快照 (Snapshot) , 以二进制的方式写入磁盘。

手动触发 :

- save命令 , 使 Redis 处于阻塞状态 , 直到 RDB 持久化完成 , 才会响应其他客户端发来的命令 , 所以在生产环境一定要慎用
- bgsave命令 , fork出一个子进程执行持久化 , 主进程只在fork过程中有短暂的阻塞 , 子进程创建之后 , 主进程就可以响应客户端请求了
- 自动触发 :
- save m n : 在 m 秒内 , 如果有 n 个键发生改变 , 则自动触发持久化 , 通过bgsave执行 , 如果设置多个、只要满足其一就会触发 , 配置文件有默认配置(可以注释掉)
- flushall : 用于清空redis所有的数据库 , flushdb清空当前redis所在库数据(默认是0号数据库) , 会 清空RDB文件 , 同时也会生成dump.rdb、内容为空
- 主从同步 : 全量同步时会自动触发bgsave命令 , 生成rdb发送给从节点

优点 :

1. 整个Redis数据库将只包含一个文件 dump.rdb , 方便持久化。
2. 容灾性好 , 方便备份。
3. 性能最大化 , fork 子进程来完成写操作 , 让主进程继续处理命令 , 所以是 IO 最大化。使用单独子进程来进行持久化 , 主进程不会进行任何 IO 操作 , 保证了 redis 的高性能
4. 相对于数据集大时 , 比 AOF的启动效率更高。

缺点 :

1. 数据安全性低。RDB 是间隔一段时间进行持久化 , 如果持久化之间 redis 发生故障 , 会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)
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。会占用cpu

AOF : Append Only File 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 , 调操作系统命令进程刷盘

1. 所有的写命令会追加到 AOF 缓冲中。
2. AOF 缓冲区根据对应的策略向硬盘进行同步操作。
3. 随着 AOF 文件越来越大 , 需要定期对 AOF 文件进行重写 , 达到压缩的目的。
4. 当 Redis 重启时 , 可以加载 AOF 文件进行数据恢复。同步策略 :

每秒同步 : 异步完成 , 效率非常高 , 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

每修改同步 : 同步持久化 , 每次发生的数据变化都会被立即记录到磁盘中 , 最多丢一条 不同步 : 由操作系统控制 , 可能丢失较多数据

优点 :

1. 数据安全
2. 通过 append 模式写文件 , 即使中途服务器宕机也不会破坏已经存在的内容 , 可以通过 redis- check-aof 工具解决数据一致性问题。

3. AOF 机制的 rewrite 模式。定期对AOF文件进行重写，以达到压缩的目的

缺点：

1. AOF 文件比 RDB 文件大，且恢复速度慢。
2. 数据集大的时候，比 rdb 启动效率低。
3. 运行效率没有RDB高

对比：

- AOF文件比RDB更新频率高，优先使用AOF还原数据。AOF比RDB更安全也更大
- RDB性能比AOF好
- 如果两个都配了优先加载AOF

## Redis单线程为什么这么快

Redis基于Reactor模式开发了网络事件处理器、文件事件处理器 fileeventhandler。它是单线程的，所以 Redis才叫做单线程的模型，它采用IO多路复用机制来同时监听多个Socket，根据Socket上的事件类型来选择对应的事件处理器来处理这个事件。可以实现高性能的网络通信模型，又可以跟内部其他单 线程的模块进行对接，保证了 Redis内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个Socket、IO多路复用程序、文件事件分派器以及事件处理器（命令请求处理器、命令回复处理器、连接应答处理器等）。

多个 Socket 可能并发的产生不同的事件，IO多路复用程序会监听多个 Socket，会将 Socket 放入一个队列中排队，每次从队列中有序、同步取出一个 Socket 给事件分派器，事件分派器把 Socket 给对应的事件处理器。

然后一个 Socket 的事件处理完之后，IO多路复用程序才会将队列中的下一个 Socket 给事件分派器。文件事件分派器会根据每个 Socket 当前产生的事件，来选择对应的事件处理器来处理。

1. Redis启动初始化时，将连接应答处理器跟AE\_READABLE事件关联。
2. 若一个客户端发起连接，会产生一个AE\_READABLE事件，然后由连接应答处理器负责和客户端建立 连接，创建客户端对应的socket，同时将这个socket的AE\_READABLE事件和命令请求处理器关联，使得客户端可以向主服务器发送命令请求。
3. 当客户端向Redis发请求时（不管读还是写请求），客户端socket都会产生一个AE\_READABLE事件，触发命令请求处理器。处理器读取客户端的命令内容，然后传给相关程序执行。
4. 当Redis服务器准备好给客户端的响应数据后，会将socket的AE\_WRITABLE事件和命令回复处理器关联，当客户端准备好读取响应数据时，会在socket产生一个AE\_WRITABLE事件，由对应命回调处理器处理，即 将准备好的响应数据写入socket，供客户端读取。
5. 命令回复处理器全部写完到 socket 后，就会删除该socket的AE\_WRITABLE事件和命回调处理器的映射。

单线程快的原因：

1. 纯内存操作
2. 核心是基于非阻塞的IO多路复用机制
3. 单线程反而避免了多线程的频繁上下文切换带来的性能问题

# 简述Redis事务实现

- 事务开始：MULTI命令的执行，标识着一个事务的开始。MULTI命令会将客户端状态的 flags属性中打开 REDIS\_MULTI标识来完成的。
- 命令入队：当一个客户端切换到事务状态之后，服务器会根据这个客户端发送来的命令来执行不同的操作。如果客户端发送的命令为MULTI、EXEC、WATCH、DISCARD中的一个，立即执行这个命令，否则将命令放入一个事务队列里面，然后向客户端返回QUEUED回复，如果客户端发送的命令为 EXEC、DISCARD、WATCH、MULTI 四个命令的其中一个，那么服务器立即执行这个命令。如果客户端发送的是四个命令以外的其他命令，那么服务器并不立即执行这个命令。首先检查此命令的格式是否正确，如果不正确，服务器会在客户端状态（redisClient）的 flags 属性关闭 REDIS\_MULTI 标识，并且返回错误信息给客户端。如果正确，将这个命令放入一个事务队列里面，然后向客户端返回 QUEUED 回复事务队列是按照FIFO的方式保存入队的命令
- 事务执行：客户端发送 EXEC 命令，服务器执行 EXEC 命令逻辑。如果客户端状态的 flags 属性不包含 REDIS\_MULTI 标识，或者包含 REDIS\_DIRTY\_CAS 或者REDIS\_DIRTY\_EXEC 标识，那么就直接取消事务的执行。否则客户端处于事务状态（flags有 REDIS\_MULTI 标识），服务器会遍历客户端的事务队列，然后执行事务队列中的所有命令，最后将返回结果全部返回给客户端；Redis不支持事务回滚机制，但是它会检查每一个事务中的命令是否错误。Redis事务不支持检查那些程序员自己逻辑错误。例如对 String 类型的数据库键执行对 HashMap 类型的操作！

## 分布式与微服务(46)

### 什么是CAP理论

CAP理论是分布式领域中非常重要的一个指导理论，C (Consistency) 表示强一致性，A (Availability) 表示可用性，P (Partition Tolerance) 表示分区容错性，CAP理论指出在目前的硬件条件下，一个分布式系统是必须要保证分区容错性的，而在这个前提下，分布式系统要么保证CP，要么保证AP，无法同时保证CAP。

分区容错性表示，一个系统虽然是分布式的，但是对外看上去应该是一个整体，不能由于分布式系统内部的某个结点挂点，或网络出现了故障，而导致系统对外出现异常。所以，对于分布式系统而言是一定要保证分区容错性的。

强一致性表示，一个分布式系统中各个结点之间能及时的同步数据，在数据同步过程中，是不能对外提供服务的，不然就会造成数据不一致，所以强一致性和可用性是不能同时满足的。

可用性表示，一个分布式系统对外要保证可用。

# 什么是BASE理论

由于不能同时满足CAP，所以出现了BASE理论：

1. BA : Basically Available , 表示基本可用，表示可以允许一定程度的不可用，比如由于系统故障，请求时间变长，或者由于系统故障导致部分非核心功能不可用，都是允许的
2. S : Soft state : 表示分布式系统可以处于一种中间状态，比如数据正在同步
3. E : Eventually consistent , 表示最终一致性，不要求分布式系统数据实时达到一致，允许在经过一段时间后再达到一致，在达到一致过程中，系统也是可用的

# 什么是RPC

RPC，表示远程过程调用，对于Java这种面试对象语言，也可以理解为远程方法调用，RPC调用和HTTP调用是有区别的，RPC表示的是一种调用远程方法的方式，可以使用HTTP协议、或直接基于TCP协议来实现RPC，在Java中，我们可以通过直接使用某个服务接口的代理对象来执行方法，而底层则通过构造HTTP请求来调用远端的方法，所以，有一种说法是RPC协议是HTTP协议之上的一种协议，也是可以理解的。

# 数据一致性模型有哪些

- 强一致性：当更新操作完成之后，任何多个后续进程的访问都会返回最新的更新过的值，这种是对用户最好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP理论，这种实现需要牺牲可用性。
- 弱一致性：系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。用户读到某一操作对系统数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。
- 最终一致性：最终一致性是弱一致性的特例，强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。到达最终一致性的时间，就是不一致窗口时间，在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。最终一致性模型根据其提供的不同保证可以划分为更多的模型，包括因果一致性和会话一致性等。

# 分布式ID是什么？有哪些解决方案？

在开发中，我们通常会需要一个唯一ID来标识数据，如果是单体架构，我们可以通过数据库的主键，或直接在内存中维护一个自增数字来作为ID都是可以的，但对于一个分布式系统，就有可能会出现ID冲突，此时有以下解决方案：

1. uuid，这种方案复杂度最低，但是会影响存储空间和性能
2. 利用单机数据库的自增主键，作为分布式ID的生成器，复杂度适中，ID长度较之uuid更短，但是受到单机数据库性能的限制，并发量大的时候，此方案也不是最优方案
3. 利用redis、zookeeper的特性来生成id，比如redis的自增命令、zookeeper的顺序节点，这种方案和单机数据库(mysql)相比，性能有所提高，可以适当选用

4. 雪花算法，一切问题如果能直接用算法解决，那就是最合适的，利用雪花算法也可以生成分布式ID，底层原理就是通过某台机器在某一毫秒内对某一个数字自增，这种方案也能保证分布式架构中的系统id唯一，但是只能保证趋势递增。业界存在tinyid、leaf等开源中间件实现了雪花算法。

## 分布式锁的使用场景是什么？有哪些实现方案？

在单体架构中，多个线程都是属于同一个进程的，所以在线程并发执行时，遇到资源竞争时，可以利用ReentrantLock、synchronized等技术来作为锁，来控制共享资源的使用。

而在分布式架构中，多个线程是可能处于不同进程中的，而这些线程并发执行遇到资源竞争时，利用ReentrantLock、synchronized等技术是没办法来控制多个进程中的线程的，所以需要分布式锁，意思就是，需要一个分布式锁生成器，分布式系统中的应用程序都可以来使用这个生成器所提供的锁，从而达到多个进程中的线程使用同一把锁。

目前主流的分布式锁的实现方案有两种：

1. zookeeper：利用的是zookeeper的临时节点、顺序节点、watch机制来实现的，zookeeper分布式锁的特点是高一致性，因为zookeeper保证的是CP，所以由它实现的分布式锁更可靠，不会出现混乱
2. redis：利用redis的setnx、lua脚本、消费订阅等机制来实现的，redis分布式锁的特点是高可用，因为redis保证的是AP，所以由它实现的分布式锁可能不可靠，不稳定（一旦redis中的数据出现了不一致），可能会出现多个客户端同时加到锁的情况

## 什么是分布式事务？有哪些实现方案？

在分布式系统中，一次业务处理可能需要多个应用来实现，比如用户发送一次下单请求，就涉及到订单系统创建订单、库存系统减库存，而对于一次下单，订单创建与减库存应该是要同时成功或同时失败的，但在分布式系统中，如果不做处理，就很有可能出现订单创建成功，但是减库存失败，那么解决这类问题，就需要用到分布式事务。常用解决方案有：

1. 本地消息表：创建订单时，将减库存消息加入在本地事务中，一起提交到数据库入本地消息表，然后调用库存系统，如果调用成功则修改本地消息状态为成功，如果调用库存系统失败，则由后台定时任务从本地消息表中取出未成功的消息，重试调用库存系统
2. 消息队列：目前RocketMQ中支持事务消息，它的工作原理是：
  - a. 生产者订单系统先发送一条half消息到Broker，half消息对消费者而言是不可见的
  - b. 再创建订单，根据创建订单成功与否，向Broker发送commit或rollback
  - c. 并且生产者订单系统还可以提供Broker回调接口，当Broker发现一段时间half消息没有收到任何操作命令，则会主动调此接口来查询订单是否创建成功
  - d. 一旦half消息commit了，消费者库存系统就会来消费，如果消费成功，则消息销毁，分布式事务成功结束
  - e. 如果消费失败，则根据重试策略进行重试，最后还失败则进入死信队列，等待进一步处理
3. Seata：阿里开源的分布式事务框架，支持AT、TCC等多种模式，底层都是基于两阶段提交理论来实现的

# 什么是ZAB协议

ZAB协议是Zookeeper用来实现一致性的原子广播协议，该协议描述了Zookeeper是如何实现一致性的，分为三个阶段：

1. 领导者选举阶段：从Zookeeper集群中选出一个节点作为Leader，所有的写请求都会由Leader节点来处理
2. 数据同步阶段：集群中所有节点中的数据要和Leader节点保持一致，如果不一致则要进行同步
3. 请求广播阶段：当Leader节点接收到写请求时，会利用两阶段提交来广播该写请求，使得写请求像事务一样在其他节点上执行，达到节点上的数据实时一致

但值得注意的是，Zookeeper只是尽量的在达到强一致性，实际上仍然只是最终一致性的。

## 简述paxos算法

Paxos算法解决的是一个分布式系统如何就某个值（决议）达成一致。一个典型的场景是，在一个分布式数据库系统中，如果各个节点的初始状态一致，每个节点执行相同的操作序列，那么他们最后能够得到一个一致的状态。为了保证每个节点执行相同的操作序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。在Paxos算法中，有三种角色：Proposer（提议者）、Acceptor（接受者）、Learner（记录员）

Proposer提议者：只要Proposer发的提案Propose被半数以上的Acceptor接受，Proposer就认为该提案例的value被选定了。

Acceptor接受者：只要Acceptor接受了某个提案，Acceptor就认为该提案例的value被选定了Learner记录员：Acceptor告诉Learner哪个value被选定，Learner就认为哪个value被选定。

Paxos算法分为两个阶段，具体如下：阶段一（prephase）：

- (a) Proposer收到client请求或者发现本地有未提交的值，选择一个提案编号N，然后向半数以上的Acceptor发送编号为N的Prepare请求。
- (b) Acceptor收到一个编号为N的Prepare请求，如果该轮paxos

本节点已经有已提交的value记录，对比记录的编号和N，大于N则拒绝响应，否则返回该记录value及编号

没有已提交记录，判断本地是否有编号N1，N1>N，则拒绝响应，否则将N1改为N（如果没有N1，则记录N），并响应prepare

阶段二（accept）：

- (a) 如果Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，那么它就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。V就是收到的响应中编号最大的value，如果响应中不包含任何value，那么V就由Proposer自己决定。
- (b) 如果Acceptor收到一个针对编号为N的提案的Accept请求，Acceptor对比本地的记录编号，如果小于等于N，则接受该值，并提交记录value。否则拒绝请求

Proposer如果收到的大多数Acceptor响应，则选定该value值，并同步给learner，使未响应的Acceptor达成一致

活锁：accept时被拒绝，加大N，重新accept，此时另外一个proposer也进行相同操作，导致accept一致失败，无法完成算法

multi-paxos：区别于paxos只是确定一个值，multi-paxos可以确定多个值，收到accept请求后，则一定时间内不再accept其他节点的请求，以此保证后续的编号不需要在经过prepare确认，直接进行accept操作。此时该节点成为了leader，直到accept被拒绝，重新发起prepare请求竞争leader资格。

## 简述raft算法

概念：

分布式一致性算法：raft会先选举出leader，leader完全负责replicatedlog的管理。leader负责接受所有客户端更新请求，然后复制到follower节点，并在“安全”的时候执行这些请求。如果leader 故障，followes会重新选举出新的leader

三种状态：一个节点任一时刻处于三者之一

leader：处理所有的客户端请求（如果客户端将请求发给了Follower，Follower将请求重定向给Leader）

follower：不会发送任何请求，只会简单地响应来自Leader或Candidate的请求

candidate：用于选举产生新的leader(候选人)

term：任期，leader产生到重新选举为一任期，每个节点都维持着当前的任期号

term是递增的，存储在log日志的entry中，代表当前entry是在哪一个term时期写入 每个任期只能有一个leader或者没有(选举失败)

每次rpc通信时传递该任期号，如果RPC收到任期号大于本地的、切换为follower，小于本地 任期号则返回错误信息

两个RPC通信：

RequestVote RPC：负责选举，包含参数lastIndex，lastTerm AppendEntries RPC：负责数据的交互。

日志序列：每一个节点上维持着一份持久化Log，通过一致性协议算法，保证每一个节点中的Log 保持一致，并且顺序存放，这样客户端就可以在每一个节点中读取到相同的数据

状态机：日志序列同步到多数节点时，leader将该日志提交到状态机，并在下一次心跳通知所有节点提交状态机（携带最后提交的lastIndex）

何时触发选举：

集群初始化时，都是follower，随机超时，变成candidate，发起选举

如果follower在election timeout内没有收到来自leader的心跳，则主动触发选举

选举过程：发出选举的节点角度

1、增加节点本地的term，切换到candidate状态2、投自己一票

其他节点投票逻辑：每个节点同一任期最多只能投一票，候选人知道的信息不能比自己少（通过副本日志和安全机制保障），先来先得

3、并行给其他节点发送RequestVote RPCs(选举请求)、包含term参数

#### 4、等待回复

4.1、收到majority(大多数)的投票，赢得选举，切换到leader状态，立刻给所有节点发心跳消息 4.2、被告知别人当选，切换到follower状态。 (原来的leader对比term，比自己的大，转换到 follower状态)

4.3、一段时间没收到majority和leader的心跳通知，则保持candidate、重新发出选举

日志序列同步：日志需要存储在磁盘持久化，崩溃可以从日志恢复

1、客户端发送命令给Leader。

2、Leader把日志条目加到自己的日志序列里。

3、Leader发送AppendEntriesRPC请求给所有的follower。携带了prevLogIndex，prevLogTerm follower收到后，进行日志序列匹配

匹配上则追加到自己的日志序列

匹配不上则拒绝请求，leader将日志index调小，重新同步直至匹配上，follower将leader的日志 序列覆盖到本地

一旦新的日志序列条目变成majority的了，将日志序列应用到状态机中

Leader在状态机里提交自己日志序列条目，然后返回结果给客户端

Leader下次发送AppendEntriesRPC时，告知follower已经提交的日志序列条目信息(lastIndex) follower收到RPC后，提交到自己的状态机里

提交状态机时，如果term为上一任期，必须与当前任期数据一起提交，否则可能出现覆盖已提交状态机 的日志

新选举出的leader一定拥有所有已提交状态机的日志条目

leader在当日志序列条目已经复制到大多数follower机器上时，才会提交日志条目。

而选出的leader的logIndex必须大于等于大多数节点，因此leader肯定有最新的日志

安全原则：

选举安全原则：对于一个给定的任期号，最多只会有一个领导人被选举出来

状态机安全原则：如果一个leader已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志

领导人完全原则：如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有领导人中

领导人只附加原则：领导人绝对不会删除或者覆盖自己的日志，只会增加

日志匹配原则：如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这个日志从头到这个索引位置之间全部完全相同

## 为什么Zookeeper可以用来作为注册中心

可以利用Zookeeper的临时节点和watch机制来实现注册中心的自动注册和发现，另外Zookeeper中的数据都是存在内存中的，并且Zookeeper底层采用了nio，多线程模型，所以Zookeeper的性能也是比较高的，所以可以用来作为注册中心，但是如果考虑到注册中心应该是注册可用性的话，那么Zookeeper则不太合适，因为Zookeeper是CP的，它注重的是一致性，所以集群数据不一致时，集群将不可用，所以用Redis、Eureka、Nacos来作为注册中心将更合适。

## Zookeeper中的领导者选举的流程是怎样的？

对于Zookeeper集群，整个集群需要从集群节点中选出一个节点作为Leader，大体流程如下：

1. 集群中各个节点首先都是观望状态（LOOKING），一开始都会投票给自己，认为自己比较适合作为leader
2. 然后相互交互投票，每个节点会收到其他节点发过来的选票，然后pk，先比较zxid，zxid大者获胜，zxid如果相等则比较myid，myid大者获胜
3. 一个节点收到其他节点发过来的选票，经过PK后，如果PK输了，则改票，此节点就会投给zxid或myid更大的节点，并将选票放入自己的投票箱中，并将新的选票发送给其他节点
4. 如果pk是平局则将接收到的选票放入自己的投票箱中
5. 如果pk赢了，则忽略所接收到的选票
6. 当然一个节点将一张选票放入到自己的投票箱之后，就会从投票箱中统计票数，看是否超过一半的节点都和自己所投的节点是一样的，如果超过半数，那么则认为当前自己所投的节点是leader
7. 集群中每个节点都会经过同样的流程，pk的规则也是一样的，一旦改票就会告诉给其他服务器，所以最终各个节点中的投票箱中的选票也将是一样的，所以各个节点最终选出来的leader也是一样的，这样集群的leader就选举出来了

## Zookeeper集群中节点之间数据是如何同步的

1. 首先集群启动时，会先进行领导者选举，确定哪个节点是Leader，哪些节点是Follower和Observer
2. 然后Leader会和其他节点进行数据同步，采用发送快照和发送Diff日志的方式
3. 集群在工作过程中，所有的写请求都会交给Leader节点来进行处理，从节点只能处理读请求
4. Leader节点收到一个写请求时，会通过两阶段机制来处理
5. Leader节点会将该写请求对应的数据发送给其他Follower节点，并等待Follower节点持久化数据成功
6. Follower节点收到数据后会进行持久化，如果持久化成功则发送一个Ack给Leader节点
7. 当Leader节点收到半数以上的Ack后，就会开始提交，先更新Leader节点本地的内存数据
8. 然后发送commit命令给Follower节点，Follower节点收到commit命令后就会更新各自本地内存数据
9. 同时Leader节点还是将当前写请求直接发送给Observer节点，Observer节点收到Leader发过来的写请求后直接执行更新本地内存数据
10. 最后Leader节点返回客户端写请求响应成功
11. 通过同步机制和两阶段提交机制来达到集群中节点数据一致

# Dubbo支持哪些负载均衡策略

1. 随机：从多个服务提供者随机选择一个来处理本次请求，调用量越大则分布越均匀，并支持按权重设置随机概率
2. 轮询：依次选择服务提供者来处理请求，并支持按权重进行轮询，底层采用的是平滑加权轮询算法
3. 最小活跃调用数：统计服务提供者当前正在处理的请求，下次请求过来则交给活跃数最小的服务器来处理
4. 一致性哈希：相同参数的请求总是发到同一个服务提供者

## Dubbo是如何完成服务导出的？

1. 首先Dubbo会将程序员所使用的@DubboService注解或@Service注解进行解析得到程序员所定义的服务参数，包括定义的服务名、服务接口、服务超时时间、服务协议等等，得到一个ServiceBean。
2. 然后调用ServiceBean的export方法进行服务导出
3. 然后将服务信息注册到注册中心，如果有多个协议，多个注册中心，那就将服务按单个协议，单个注册中心进行注册
4. 将服务信息注册到注册中心后，还会绑定一些监听器，监听动态配置中心的变更
5. 还会根据服务协议启动对应的Web服务器或网络框架，比如Tomcat、Netty等

## Dubbo是如何完成服务引入的？

1. 当程序员使用@Reference注解来引入一个服务时，Dubbo会将注解和服务的信息解析出来，得到当前所引用的服务名、服务接口是什么
2. 然后从注册中心进行查询服务信息，得到服务的提供者信息，并存在消费端的服务目录中
3. 并绑定一些监听器用来监听动态配置中心的变更
4. 然后根据查询得到的服务提供者信息生成一个服务接口的代理对象，并放入Spring容器中作为Bean

## Dubbo的架构设计是怎样的？

Dubbo中的架构设计是非常优秀的，分为了很多层次，并且每层都是可以扩展的，比如：

1. Proxy服务代理层，支持JDK动态代理、javassist等代理机制
2. Registry注册中心层，支持Zookeeper、Redis等作为注册中心
3. Protocol远程调用层，支持Dubbo、Http等调用协议
4. Transport网络传输层，支持netty、mina等网络传输框架
5. Serialize数据序列化层，支持JSON、Hessian等序列化机制

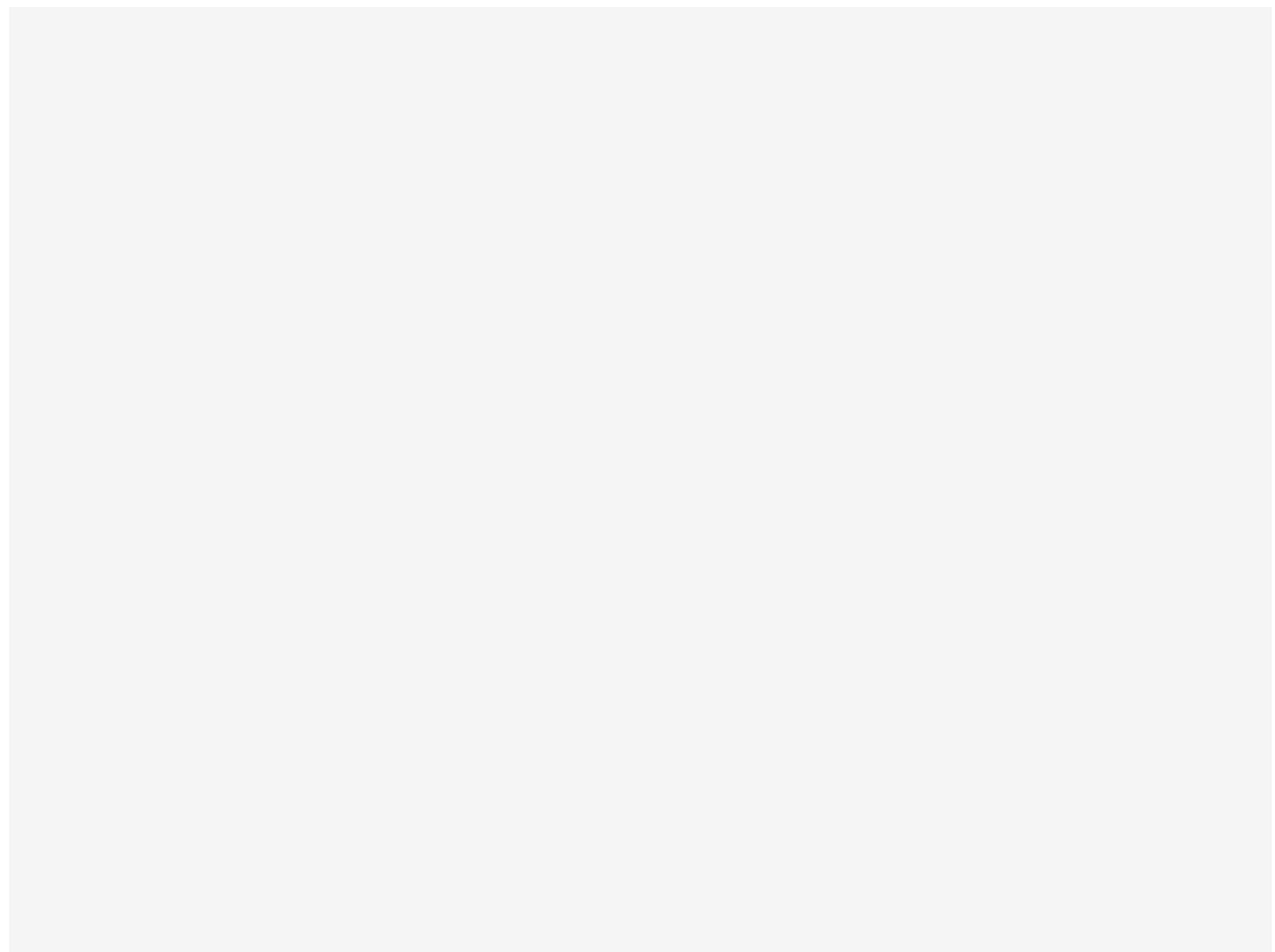
### 各层说明

- **config 配置层**：对外配置接口，以 `ServiceConfig`, `ReferenceConfig` 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类

- **proxy 服务代理层**：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 以 `ServiceProxy` 为中心，扩展接口为 `ProxyFactory`
- **registry 注册中心层**：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 `RegistryFactory`, `Registry`, `RegistryService`
- **cluster 路由层**：封装多个提供者的路由及负载均衡，并桥接注册中心，以 `Invoker` 为中心，扩展接口为 `Cluster`, `Directory`, `Router`, `LoadBalance`
- **monitor 监控层**：RPC 调用次数和调用时间监控，以 `Statistics` 为中心，扩展接口为 `MonitorFactory`, `Monitor`, `MonitorService`
- **protocol 远程调用层**：封装 RPC 调用，以 `Invocation`, `Result` 为中心，扩展接口为 `Protocol`, `Invoker`, `Exporter`
- **exchange 信息交换层**：封装请求响应模式，同步转异步，以 `Request`, `Response` 为中心，扩展接口为 `Exchanger`, `ExchangeChannel`, `ExchangeClient`, `ExchangeServer`
- **transport 网络传输层**：抽象 mina 和 netty 为统一接口，以 `Message` 为中心，扩展接口为 `Channel`, `Transporter`, `Client`, `Server`, `Codec`
- **serialize 数据序列化层**：可复用的一些工具，扩展接口为 `Serialization`, `ObjectInput`, `ObjectOutput`, `ThreadPool`

## 关系说明

- 在 RPC 中，Protocol 是核心层，也就是只要有 Protocol + Invoker + Exporter 就可以完成非透明的 RPC 调用，然后在 Invoker 的主过程中 Filter 拦截点。
- 图中的 Consumer 和 Provider 是抽象概念，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，不用 Client 和 Server 的原因是 Dubbo 在很多场景下都使用 Provider, Consumer, Registry, Monitor 划分逻辑拓普节点，保持统一概念。
- 而 Cluster 是外围概念，所以 Cluster 的目的是将多个 Invoker 伪装成一个 Invoker，这样其它人只要关注 Protocol 层 Invoker 即可，加上 Cluster 或者去掉 Cluster 对其它层都不会造成影响，因为只有一个提供者时，是不需要 Cluster 的。
- Proxy 层封装了所有接口的透明化代理，而在其它层都以 Invoker 为中心，只有到了暴露给用户使用时，才用 Proxy 将 Invoker 转成接口，或将接口实现转成 Invoker，也就是去掉 Proxy 层 RPC 是可以 Run 的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。
- 而 Remoting 实现是 Dubbo 协议的实现，如果你选择 RMI 协议，整个 Remoting 都不会用上，Remoting 内部再划为 Transport 传输层和 Exchange 信息交换层，Transport 层只负责单向消息传输，是对 Mina, Netty, Grizzly 的抽象，它也可以扩展 UDP 传输，而 Exchange 层是在传输层之上封装了 Request-Response 语义。
- Registry 和 Monitor 实际上不算一层，而是一个独立的节点，只是为了全局概览，用层的方式画在一起。



## 负载均衡算法有哪些

**1、轮询法：**将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

**2、随机法：**通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

**3、源地址哈希法：**源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

**4、加权轮询法：**不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

**5、加权随机法**：与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

**6、最小连接数法**：最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负荷合理地分流到每一台服务器。

## 分布式架构下，Session 共享有什么方案

1、采用无状态服务，抛弃session

2、存入cookie（有安全风险）

3、服务器之间进行 Session 同步，这样可以保证每个服务器上都有全部的 Session 信息，不过当服务器数量比较多的时候，同步是会有延迟甚至同步失败；

4、IP 绑定策略

使用 Nginx（或其他复杂均衡软硬件）中的 IP 绑定策略，同一个 IP 只能在指定的同一个机器访问，但是这样做失去了负载均衡的意义，当挂掉一台服务器的时候，会影响一批用户的使用，风险很大；

5、使用 Redis 存储

把 Session 放到 Redis 中存储，虽然架构上变得复杂，并且需要多访问一次 Redis，但是这种方案带来的好处也是很大的：

- 实现了 Session 共享；
- 可以水平扩展（增加 Redis 服务器）；
- 服务器重启 Session 不丢失（不过也要注意 Session 在 Redis 中的刷新/失效机制）；
- 不仅可以跨服务器 Session 共享，甚至可以跨平台（例如网页端和 APP 端）。

## 简述你对RPC、RMI的理解

RPC：在本地调用远程的函数，远程过程调用，可以跨语言实现 httpClient

RMI：远程方法调用，java中用于实现RPC的一种机制，RPC的java版本，是J2EE的网络调用机制，跨JVM调用对象的方法，面向对象的思维方式

直接或间接实现接口 `java.rmi.Remote` 成为存在于服务器端的远程对象，供客户端访问并提供一定的服务

远程对象必须实现`java.rmi.server.UnicastRemoteObject`类，这样才能保证客户端访问获得远程对象时，该远程对象将会把自身的一个拷贝以Socket的形式传输给客户端，此时客户端所获得的这个拷贝称为“存根”，而服务器端本身已存在的远程对象则称之为“骨架”。其实此时的存根是客户端的一个代理，用于与服务器端的通信，而骨架也可认为是服务器端的一个代理，用于接收客户端的请求之后调用远程方法来响应客户端的请求。

```
1 public interface IService extends Remote {
2     String service(String content) throws RemoteException;
3 }
4
5 public class ServiceImpl extends UnicastRemoteObject implements IService {
6
7     private String name;
8
9     public ServiceImpl(String name) throws RemoteException {
10         this.name = name;
11     }
12     @Override
13     public String service(String content) {
14         return "server >> " + content;
15     }
16 }
17
18 public class Server {
19     public static void main(String[] args) {
20         try {
21             IService service02 = new ServiceImpl("service02");
22             Context namingContext = new InitialContext();
23             namingContext.rebind("rmi://127.0.0.1/service02", service02);
24         } catch (Exception e) {
25             e.printStackTrace();
26         }
27         System.out.println("000000 ! ");
28     }
29 }
30
31 public class Client {
32     public static void main(String[] args) {
33         String url = "rmi://127.0.0.1/";
34         try {
35             Context namingContext = new InitialContext();
36             IService service02 = (IService) namingContext.lookup(url + "service02");
37             Class stubClass = service02.getClass();
38             System.out.println(service02 + " is " + stubClass.getName());
39             //com.sun.proxy.$Proxy0
40
41             Class[] interfaces = stubClass.getInterfaces();
42             for (Class c : interfaces) {
43                 System.out.println("implement" + c.getName() + " interface");
44             }
45             System.out.println(service02.service("hello"));
46         } catch (Exception e) {
47             e.printStackTrace();
48         }
49     }
50 }
```

# 如何实现接口的幂等性

- 唯一id。每次操作，都根据操作和内容生成唯一的id，在执行之前先判断id是否存在，如果不存在则执行后续操作，并且保存到数据库或者redis等。
- 服务端提供发送token的接口，业务调用接口前先获取token,然后调用业务接口请求时，把token携带过去,务器判断token是否存在redis中，存在表示第一次请求，可以继续执行业务，执行业务完成后，最后需要把redis中的token删除
- 建去重表。将业务中有唯一标识的字段保存到去重表，如果表中存在，则表示已经处理过了
- 版本控制。增加版本号，当版本号符合时，才能更新数据
- 状态控制。例如订单有状态已支付 未支付 支付中 支付失败，当处于未支付的时候才允许修改为支付中等

## Zookeeper的数据模型和节点类型

数据模型：树形结构

zk维护的数据主要有：客户端的会话（session）状态及数据节点（dataNode）信息。

zk在内存中构造了个DataTree的数据结构，维护着path到dataNode的映射以及dataNode间的树状层级关系。为了提高读取性能，集群中每个服务节点都是将数据全量存储在内存中。所以，zk最适于读多写少且轻量级数据的应用场景。

数据仅存储在内存是很不安全的，zk采用事务日志文件及快照文件的方案来落盘数据，保障数据在不丢失的情况下能快速恢复。

树中的每个节点被称为— Znode

Znode 兼具文件和目录两种特点。可以做路径标识，也可以存储数据，并可以具有子 Znode。具有增、删、改、查等操作。

Znode 具有原子性操作，读操作将获取与节点相关的所有数据，写操作也将 替换掉节点的所有数据。另外，每一个节点都拥有自己的 ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作

Znode 存储数据大小有限制。每个 Znode 的数据大小至多 1M，常规使用中应该远小于此值。

Znode 通过路径引用，如同 Unix 中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在 ZooKeeper 中，路径由

Unicode 字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

持久节点：一旦创建、该数据节点会一直存储在zk服务器上、即使创建该节点的客户端与服务端的会话关闭了、该节点也不会被删除

临时节点：当创建该节点的客户端会话因超时或发生异常而关闭时、该节点也相应的在zk上被删除。

有序节点：不是一种单独种类的节点、而是在持久节点和临时节点的基础上、增加了一个节点有序的性质。

## 简述zk的命名服务、配置管理、集群管理

命名服务：

通过指定的名字来获取资源或者服务地址。Zookeeper可以创建一个全局唯一的路径，这个路径就可以作为一个名字。被命名的实体可以是集群中的机器，服务的地址，或者是远程的对象等。一些分布式服务框架（RPC、RMI）中的服务地址列表，通过使用命名服务，客户端应用能够根据特定的名字来获取资源的实体、服务地址和提供者信息等

配置管理：

实际项目开发中，经常使用.properties或者xml需要配置很多信息，如数据库连接信息、fps地址端口等等。程序分布式部署时，如果把程序的这些配置信息保存在zk的znode节点下，当你要修改配置，即znode会发生变化时，可以通过改变zk中某个目录节点的内容，利用watcher通知给各个客户端，从而更改配置。

集群管理：

集群管理包括集群监控和集群控制，就是监控集群机器状态，剔除机器和加入机器。zookeeper可以方便集群机器的管理，它可以实时监控znode节点的变化，一旦发现有机器挂了，该机器就会与zk断开连接，对应的临时目录节点会被删除，其他所有机器都收到通知。新机器加入也是类似。

## 讲下Zookeeper中的watch机制

客户端，可以通过在znode上设置watch，实现实时监听znode的变化

Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端

- 父节点的创建，修改，删除都会触发Watcher事件。
- 子节点的创建，删除会触发Watcher事件。

一次性：一旦被触发就会移除，再次使用需要重新注册，因为每次变动都需要通知所有客户端，一次性可以减轻压力，3.6.0默认持久递归，可以触发多次

轻量：只通知发生了事件，不会告知事件内容，减轻服务器和带宽压力

Watcher 机制包括三个角色：客户端线程、客户端的 WatchManager 以及 ZooKeeper 服务器

1. 客户端向 ZooKeeper 服务器注册一个 Watcher 监听，
2. 把这个监听信息存储到客户端的 WatchManager 中
3. 当 ZooKeeper 中的节点发生变化时，会通知客户端，客户端会调用相应 Watcher 对象中的回调方法。watch 回调是串行同步的

## Zookeeper和Eureka的区别

zk : CP设计(强一致性)，目标是一个分布式的协调系统，用于进行资源的统一管理。

当节点crash后，需要进行leader的选举，在这个期间内，zk服务是不可用的。

eureka : AP设计（高可用），目标是一个服务注册发现系统，专门用于微服务的服务发现注册。

Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时如果发现连接失败，会自动切换至其他节点，只要有一台Eureka还在，就能保证注册服务可用（保证可用性），只不过查到的信息可能不是最新的（不保证强一致性）

同时当eureka的服务端发现85%以上的服务都没有心跳的话，它就会认为自己的网络出了问题，就不会从服务列表中删除这些失去心跳的服务，同时eureka的客户端也会缓存服务信息。eureka对于服务注册发现来说是非常好的选择。

## 如何实现分库分表

将原本存储于单个数据库上的数据拆分到多个数据库，把原来存储在单张数据表的数据拆分到多张数据表中，实现数据切分，从而提升数据库操作性能。分库分表的实现可以分为两种方式：垂直切分和水平切分。

水平：将数据分散到多张表，涉及分区键，

分库：每个库结构一样，数据不一样，没有交集。库多了可以缓解io和cpu压力

分表：每个表结构一样，数据不一样，没有交集。表数量减少可以提高sql执行效率、减轻cpu压力

垂直：将字段拆分为多张表，需要一定的重构

分库：每个库结构、数据都不一样，所有库的并集为全量数据

分表：每个表结构、数据不一样，至少有一列交集，用于关联数据，所有表的并集为全量数据

## 存储拆分后如何解决唯一主键问题

- UUID：简单、性能好，没有顺序，没有业务含义，存在泄漏mac地址的风险
- 数据库主键：实现简单，单调递增，具有一定的业务可读性，强依赖db、存在性能瓶颈，存在暴露业务信息的风险
- redis，mongodb，zk等中间件：增加了系统的复杂度和稳定性
- 雪花算法

## 雪花算法原理

第一位符号位固定为0，41位时间戳，10位workId，12位序列号，位数可以有不同实现。

优点：每个毫秒值包含的ID值很多，不够可以变动位数来增加，性能佳（依赖workId的实现）。时间戳值在高位，中间是固定的机器码，自增的序列在低位，整个ID是趋势递增的。能够根据业务场景数据库节点布置灵活调整bit位划分，灵活度高。

缺点：强依赖于机器时钟，如果时钟回拨，会导致重复的ID生成，所以一般基于此的算法发现时钟回拨，都会抛异常处理，阻止ID生成，这可能导致服务不可用。

## 如何解决不使用分区键的查询问题

- 映射：将查询条件的字段与分区键进行映射，建一张单独的表维护（使用覆盖索引）或者在缓存中维护

- 基因法：分区键的后x个bit位由查询字段进行hash后占用，分区键直接取x个bit位获取分区，查询字段进行hash获取分区，适合非分区键查询字段只有一个的情况
- 冗余：查询字段冗余存储

## Spring Cloud有哪些常用组件，作用是什么？

1. Eureka：注册中心
2. Nacos：注册中心、配置中心
3. Consul：注册中心、配置中心
4. Spring Cloud Config：配置中心
5. Feign/OpenFeign：RPC调用
6. Kong：服务网关
7. Zuul：服务网关
8. Spring Cloud Gateway：服务网关
9. Ribbon：负载均衡
10. Spring Cloud Sleuth：链路追踪
11. Zipkin：链路追踪
12. Seata：分布式事务
13. Dubbo：RPC调用
14. Sentinel：服务熔断
15. Hystrix：服务熔断

## 如何避免缓存穿透、缓存击穿、缓存雪崩？

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
- 给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效，则更新数据缓存。
- 缓存预热互斥锁

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 接口层增加校验，如用户鉴权校验，id做基础校验， $id \leq 0$ 的直接拦截；
- 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击
- 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

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

- 设置热点数据永远不过期。加互斥锁

## 分布式系统中常用的缓存方案有哪些

- 客户端缓存：页面和浏览器缓存，APP缓存，H5缓存，localStorage 和 sessionStorage
- CDN缓存：内容存储：数据的缓存，内容分发：负载均衡
- nginx缓存：静态资源
- 服务端缓存：本地缓存，外部缓存
- 数据库缓存：持久层缓存（mybatis，hibernate多级缓存），mysql查询缓存
- 操作系统缓存：PageCache、BufferCache

## 缓存过期都有哪些策略？

- 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量
- 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，但是很消耗内存、许多的过期数据都还存在内存中。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key（是随机的），并清除其中已过期的key。该策略是定时过期和惰性过期的折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。
- 分桶策略：定期过期的优化，将过期时间点相近的key放在一起，按时间扫描分桶。

## 常见的缓存淘汰算法

- FIFO (First In First Out，先进先出)，根据缓存被存储的时间，离当前最近的数据优先被淘汰；
- LRU (LeastRecentlyUsed，最近最少使用)，根据最近被使用的时间，离当前最近的数据优先被淘汰；
- LFU (LeastFrequentlyUsed，最不经常使用)，在一段时间内，缓存数据被使用次数最少的会被淘汰。

# 布隆过滤器原理，优缺点

- 位图：int[10]，每个int类型的整数是 $4 \times 8 = 32$ 个bit，则int[10]一共有320 bit，每个bit非0即1，初始化时都是0
- 添加数据时：将数据进行hash得到hash值，对应到bit位，将该bit改为1，hash函数可以定义多个，则一个数据添加会将多个（hash函数个数）bit改为1，多个hash函数的目的是减少hash碰撞的概率
- 查询数据：hash函数计算得到hash值，对应到bit中，如果有一个为0，则说明数据不在bit中，如果都为1，则该数据可能在bit中

优点：

- 占用内存小
- 增加和查询元素的时间复杂度为： $O(K)$ , ( $K$ 为哈希函数的个数，一般比较小)，与数据量大小无关哈希函数相互之间没有关系，方便硬件并行运算
- 布隆过滤器不需要存储元素本身，在某些对保密要求比较严格的场合有很大优势 数据量很大时，布隆过滤器可以表示全集
- 使用同一组散列函数的布隆过滤器可以进行交、并、差运算

缺点：

- 误判率，即存在假阳性(False Position)，不能准确判断元素是否在集合中不能获取元素本身
- 一般情况下不能从布隆过滤器中删除元素

# 分布式缓存寻址算法

- hash算法：根据key进行hash函数运算、结果对分片数取模，确定分片 适合固定分片数的场景，扩展分片或者减少分片时，所有数据都需要重新计算分片、存储
- 一致性hash：将整个hash值得区间组织成一个闭合的圆环，计算每台服务器的hash值、映射到圆环中。使用相同的hash算法计算数据的hash值，映射到圆环，顺时针寻找，找到的第一个服务器就是数据存储的服务器。新增及减少节点时只会影响节点到他逆时针最近的一个服务器之间的值 存在hash环倾斜的问题，即服务器分布不均匀，可以通过虚拟节点解决
- hash slot：将数据与服务器隔离开，数据与slot映射，slot与服务器映射，数据进行hash决定存放的slot，新增及删除节点时，将slot进行迁移即可

# 什么是Hystrix？简述实现机制

分布式容错框架

- 阻止故障的连锁反应，实现熔断

- 快速失败，实现优雅降级
- 提供实时的监控和告警

资源隔离：线程隔离，信号量隔离

- 线程隔离：Hystrix会给每一个Command分配一个单独的线程池，这样在进行单个服务调用的时候，就可以在独立的线程池里面进行，而不会对其他线程池造成影响
- 信号量隔离：客户端需向依赖服务发起请求时，首先要获取一个信号量才能真正发起调用，由于信号量的数量有限，当并发请求量超过信号量个数时，后续的请求都会直接拒绝，进入fallback流程。信号量隔离主要是通过控制并发请求数量，防止请求线程大面积阻塞，从而达到限流和防止雪崩的目的。

熔断和降级：调用服务失败后快速失败

熔断是为了防止异常不扩散，保证系统的稳定性

降级：编写好调用失败的补救逻辑，然后对服务直接停止运行，这样这些接口就无法正常调用，但又不至于直接报错，只是服务水平下降

- 通过HystrixCommand 或者HystrixObservableCommand 将所有的外部系统（或者称为依赖）包装起来，整个包装对象是单独运行在一个线程之中（这是典型的命令模式）。
- 超时请求应该超过你定义的阈值
- 为每个依赖关系维护一个小的线程池（或信号量）；如果它变满了，那么依赖关系的请求将立即被拒绝，而不是排队等待。
- 统计成功，失败（由客户端抛出的异常），超时和线程拒绝。
- 打开断路器可以在一段时间内停止对特定服务的所有请求，如果服务的错误百分比通过阈值，手动或自动的关闭断路器。
- 当请求被拒绝、连接超时或者断路器打开，直接执行fallback逻辑。
- 近乎实时监控指标和配置变化。

## Spring Cloud和Dubbo有哪些区别？

Spring Cloud是一个微服务框架，提供了微服务领域中的很多功能组件，Dubbo一开始是一个RPC调用框架，核心是解决服务调用间的问题，Spring Cloud是一个大而全的框架，Dubbo则更侧重于服务调用，所以Dubbo所提供的功能没有Spring Cloud全面，但是Dubbo的服务调用性能比Spring Cloud高，不过Spring Cloud和Dubbo并不是对立的，是可以结合起来一起使用的。

## 什么是服务雪崩？什么是服务限流？

1. 当服务A调用服务B，服务B调用C，此时大量请求突然请求服务A，假如服务A本身能抗住这些请求，但是如果服务C抗不住，导致服务C请求堆积，从而服务B请求堆积，从而服务A不可用，这就是服务雪崩，解决方式就是服务降级和服务熔断。
2. 服务限流是指在高并发请求下，为了保护系统，可以对访问服务的请求进行数量上的限制，从而防止系统不被大量请求压垮，在秒杀中，限流是非常重要的。

## 什么是服务熔断？什么是服务降级？区别是什么？

1. 服务熔断是指，当服务A调用的某个服务B不可用时，上游服务A为了保证自己不受影响，从而不再调用服务B，直接返回一个结果，减轻服务A和服务B的压力，直到服务B恢复。
2. 服务降级是指，当发现系统压力过载时，可以通过关闭某个服务，或限流某个服务来减轻系统压力，这就是服务降级。

相同点：

1. 都是为了防止系统崩溃
2. 都让用户体验到某些功能暂时不可用

不同点：熔断是下游服务故障触发的，降级是为了降低系统负载

## SOA、分布式、微服务之间有什么关系和区别？

1. 分布式架构是指将单体架构中的各个部分拆分，然后部署不同的机器或进程中去，SOA和微服务基本上都是分布式架构的
2. SOA是一种面向服务的架构，系统的所有服务都注册在总线上，当调用服务时，从总线上查找服务信息，然后调用
3. 微服务是一种更彻底的面向服务的架构，将系统中各个功能个体抽成一个个小的应用程序，基本保持一个应用对应的一个服务的架构

## 怎么拆分微服务？

拆分微服务的时候，为了尽量保证微服务的稳定，会有一些基本的准则：

1. 微服务之间尽量不要有业务交叉。
2. 微服务之前只能通过接口进行服务调用，而不能绕过接口直接访问对方的数据。
3. 高内聚，低耦合。

## 怎样设计出高内聚、低耦合的微服务？

高内聚低耦合，是一种从上而下指导微服务设计的方法。实现高内聚低耦合的工具主要有 同步的接口调用 和 异步的事件驱动 两种方式。

# 有没有了解过DDD领域驱动设计？

什么是DDD：在2004年，由Eric Evans提出了，DDD是面对软件复杂之道。Domain-Driven- Design –Tackling Complexity in the Heart of Software

大泥团：不利于微服务的拆分。大泥团结构拆分出来的微服务依然是泥团机构，当服务业务逐渐复杂，这个泥团又会膨胀成为大泥团。

DDD只是一种方法论，没有一个稳定的技术框架。DDD要求领域是跟技术无关、跟存储无关、跟通信无关。

## 什么是中台？

所谓中台，就是将各个业务线中可以复用的一些功能抽取出来，剥离个性，提取共性，形成一些可复用的组件。

大体上，中台可以分为三类 业务中台、数据中台和技术中台。大数据杀熟-数据中台

中台跟DDD结合：DDD会通过限界上下文将系统拆分成一个一个的领域，而这种限界上下文，天生就成了中台之间的逻辑屏障。

DDD在技术与资源调度方面都能够给中台建设提供不错的指导。

DDD分为战略设计和战术设计。上层的战略设计能够很好的指导中台划分，下层的战术设计能够很好的指导微服务搭建。

## 你的项目中是怎么保证微服务敏捷开发的？

- 开发运维一体化。
- 敏捷开发：目的就是为了提高团队的交付效率，快速迭代，快速试错
- 每个月固定发布新版本，以分支的形式保存到代码仓库中。快速入职。任务面板、站立会议。团队人员灵活流动，同时形成各个专家代表
- 测试环境- 生产环境 -开发测试环境SIT-集成测试环境-压测环境STR-预投产环境-生产环境PRD
- 晨会、周会、需求拆分会

## 消息队列(28)

### 如何进行产品选型？

- Kafka：
  - 优点：吞吐量非常大，性能非常好，集群高可用。
  - 缺点：会丢数据，功能比较单一。

- 使用场景：日志分析、大数据采集

- RabbitMQ：

- 优点：消息可靠性高，功能全面。

- 缺点：吞吐量比较低，消息积累会严重影响性能。erlang语言不好定制。

- 使用场景：小规模场景。

- RocketMQ：

- 优点：高吞吐、高性能、高可用，功能非常全面。

- 缺点：开源版功能不如云上商业版。官方文档和周边生态还不够成熟。客户端只支持java。

- 使用场景：几乎是全场景。

## 简述RabbitMQ的架构设计

**Broker**：rabbitmq的服务节点

**Queue**：队列，是RabbitMQ的内部对象，用于存储消息。RabbitMQ中消息只能存储在队列中。生产者投递消息到队列，消费者从队列中获取消息并消费。多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊(轮询)给多个消费者进行消费，而不是每个消费者都收到所有的消息进行消费。(注意：RabbitMQ不支持队列层面的广播消费，如果需要广播消费，可以采用一个交换器通过路由Key绑定多个队列，由多个消费者来订阅这些队列的方式。)

**Exchange**：交换器。生产者将消息发送到Exchange，由交换器将消息路由到一个或多个队列中。如果路由不到，或返回给生产者，或直接丢弃，或做其它处理。

**RoutingKey**：路由Key。生产者将消息发送给交换器的时候，一般会指定一个RoutingKey，用来指定这个消息的路由规则。这个路由Key需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。在交换器类型和绑定键固定的情况下，生产者可以在发送消息给交换器时通过指定RoutingKey来决定消息流向哪里。

**Binding**：通过绑定将交换器和队列关联起来，在绑定的时候一般会指定一个绑定键，这样RabbitMQ就可以指定如何正确的路由到队列了。

交换器和队列实际上是多对多关系。就像关系数据库中的两张表。他们通过BindingKey做关联(多对多关系表)。在投递消息时，可以通过Exchange和RoutingKey(对应BindingKey)就可以找到相对应的队列。

**信道**：信道是建立在Connection之上的虚拟连接。当应用程序与Rabbit Broker建立TCP连接的时候，客户端紧接着可以创建一个AMQP信道(Channel)，每个信道都会被指派一个唯一的D。RabbitMQ处理的每条AMQP指令都是通过信道完成的。信道就像电缆里的光纤束。一条电缆内含有许多光纤束，允许所有的连接通过多条光线束进行传输和接收。

# RabbitMQ如何确保消息发送？消息接收？

发送方确认机制：信道需要设置为 confirm 模式，则所有在信道上发布的消息都会分配一个唯一 ID。一旦消息被投递到queue（可持久化的消息需要写入磁盘），信道会发送一个确认给生产者（包含消息唯一 ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（未确认）消息给生产者。所有被发送的消息都将被 confirm（即 ack）或者被nack一次。但是没有对消息被 confirm 的快慢做任何保证，并且同一条消息不会既被 confirm 又被nack。发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者，生产者的回调方法会被触发。

ConfirmCallback接口：只确认是否正确到达 Exchange 中，成功到达则回调

ReturnCallback接口：消息失败返回时回调

接收方确认机制：消费者在声明队列时，可以指定noAck参数，当noAck=false时，RabbitMQ会等待消费者显式发回ack信号后才从内存(或者磁盘，持久化消息)中移去消息。否则，消息被消费后会被立即删除。

消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

RabbitMQ不会为未ack的消息设置超时时间，它判断此消息是否需要重新投递给消费者的唯一依据是消费该消息的消费者连接是否已经断开。这么设计的原因是RabbitMQ允许消费者消费一条消息的时间可以很长。保证数据的最终一致性；

如果消费者返回ack之前断开了链接，RabbitMQ 会重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）

## RabbitMQ事务消息

通过对信道的设置实现

1. channel.txSelect()；通知服务器开启事务模式；服务端会返回Tx.Select-Ok
2. channel.basicPublish；发送消息，可以是多条，可以是消费消息提交ack
3. channel.txCommit()提交事务；
4. channel.txRollback()回滚事务；

消费者使用事务：

1. autoAck=false，手动提交ack，以事务提交或回滚为准；

2. `autoAck=true`，不支持事务的，也就是说你即使在收到消息之后在回滚事务也是于事无补的，队列已经把消息移除了

如果其中任意一个环节出现问题，就会抛出`IOException`异常，用户可以拦截异常进行事务回滚，或决定要不要重复消息。

事务消息会降低rabbitmq的性能

## RabbitMQ死信队列、延时队列

1. 消息被消费方否定确认，使用 `channel.basicNack` 或 `channel.basicReject`，并且此时 `requeue` 属性被设置为 `false`。
2. 消息在队列的存活时间超过设置的TTL时间。
3. 消息队列的消息数量已经超过最大队列长度。

那么该消息将成为“死信”。“死信”消息会被RabbitMQ进行特殊处理，如果配置了死信队列信息，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃

为每个需要使用死信的业务队列配置一个死信交换机，这里同一个项目的死信交换机可以共用一个，然后为每个业务队列分配一个单独的路由key，死信队列只不过是绑定在死信交换机上的队列，死信交换机也不是什么特殊的交换机，只不过是用来接受死信的交换机，所以可以为任何类型【`Direct`、`Fanout`、`Topic`】

TTL：一条消息或者该队列中的所有消息的最大存活时间

如果一条消息设置了TTL属性或者进入了设置TTL属性的队列，那么这条消息如果在TTL设置的时间内没有被消费，则会成为“死信”。如果同时配置了队列的TTL和消息的TTL，那么较小的那个值将会被使用。

只需要消费者一直消费死信队列里的消息

## RabbitMQ镜像队列机制

镜像queue有master节点和slave节点。`master`和`slave`是针对一个queue而言的，而不是一个node作为所有queue的`master`，其它node作为`slave`。一个queue第一次创建的node为它的`master`节点，其它node为`slave`节点。

无论客户端的请求打到`master`还是`slave`最终数据都是从`master`节点获取。当请求打到`master`节点时，`master`节点直接将消息返回给client，同时`master`节点会通过GM（Guaranteed Multicast）协议将queue的最新状态广播到`slave`节点。GM保证了广播消息的原子性，即要么都更新要么都不更新。

当请求打到slave节点时，slave节点需要将请求先重定向到master节点，master节点将将消息返回给client，同时master节点会通过GM协议将queue的最新状态广播到slave节点。

如果有新节点加入，RabbitMQ不会同步之前的历史数据，新节点只会复制该节点加入到集群之后新增的消息。

## Kafka是什么

Kafka 是一种高吞吐量、分布式、基于发布/订阅的消息系统，最初由 LinkedIn 公司开发，使用Scala 语言编写，目前是 Apache 的开源项目。broker：Kafka 服务器，负责消息存储和转发topic：消息类别， Kafka 按照 topic 来分类消息partition：topic 的分区，一个 topic 可以包含多个 partition，topic 消息保存在各个partition 上offset：消息在日志中的位置，可以理解是消息在 partition 上的偏移量，也是代表该消息的唯一序号Producer：消息生产者Consumer：消息消费者Consumer Group：消费者分组，每个 Consumer 必须属于一个 groupZookeeper：保存着集群 broker、topic、partition 等 meta 数据；另外，还负责 broker 故障发现，partition leader 选举，负载均衡等功能

## Kafka为什么吞吐量高

Kafka的生产者采用的是异步发送消息机制，当发送一条消息时，消息并没有发送到Broker而是缓存起来，然后直接向业务返回成功，当缓存的消息达到一定数量时再批量发送给Broker。这种做法减少了网络io，从而提高了消息发送的吞吐量，但是如果消息生产者宕机，会导致消息丢失，业务出错，所以理论上kafka利用此机制提高了性能却降低了可靠性。

## Kafka的Pull和Push分别有什么优缺点

1. pull表示消费者主动拉取，可以批量拉取，也可以单条拉取，所以pull可以由消费者自己控制，根据自己的消息处理能力来进行控制，但是消费者不能及时知道是否有消息，可能会拉到的消息为空
2. push表示Broker主动给消费者推送消息，所以肯定是有消息时才会推送，但是消费者不能按自己的能力来消费消息，推过来多少消息，消费者就得消费多少消息，所以可能会造成网络堵塞，消费者压力大等问题

## 为什么要使用 kafka，为什么要使用消息队列？

**缓冲和削峰**：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。

**解耦和扩展性**：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。**冗余**：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。**健壮性**：消息队列可以堆积请求，所以消费端业务

即使短时间死掉，也不会影响主要业务的正常进行。**异步通信**：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

## Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

ISR:In-Sync Replicas 副本同步队列

AR:Assigned Replicas 所有副本ISR是由leader维护，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度，当前最新的版本0.10.x中只支持replica.lag.time.max.ms这个维度），任意一个超过阈值都会把follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。AR=ISR+OSR。

## Kafka高效文件存储设计特点：

1. Kafka 把 topic 中一个 partition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。
2. 通过索引信息可以快速定位 message 和确定 response 的最大大小。
3. 通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。
4. 通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

## Kafka与传统消息系统之间有三个关键区别

1. Kafka 持久化日志，这些日志可以被重复读取和无限期保留
2. Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性
3. Kafka 支持实时的流式处理

## Kafka创建 Topic 时如何将分区放置到不同的 Broker 中

1. 副本因子不能大于 Broker 的个数；
2. 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；
3. 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个Broker 上，依次类推；
4. 剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

# Kafka的消费者如何消费数据

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置等到下次消费时，他会接着上次位置继续消费

## Kafka消费者负载均衡策略

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员

## kafka生产数据时数据的分组策略

生产者决定数据产生到集群的那个 partition 中每一条消息都是以（key，value）格式 Key是由生产者发送数据传入所以生产者（key）决定了数据产生到集群的那个 partition

## Kafka中是怎么体现消息顺序性的？

kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1.

## Kafka如何实现延迟队列？

Kafka并没有使用JDK自带的Timer或者DelayQueue来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK的Timer和DelayQueue插入和删除操作的平均时间复杂度为 $O(n \log(n))$ ，并不能满足Kafka的高性能要求，而基于时间轮可以将插入和删除操作的时间复杂度都降为 $O(1)$ 。时间轮的应用并非Kafka独有，其应用场景还有很多，在Netty、Akka、Quartz、Zookeeper等组件中都存在时间轮的踪影。底层使用数组实现，数组中的每个元素可以存放一个TimerTaskList对象。TimerTaskList是一个环形双向链表，在其中的链表项TimerTaskEntry中封装了真正的定时任务TimerTask。Kafka中到底是怎么推进时间的呢？Kafka中的定时器借助了JDK中的DelayQueue来协助推进时间轮。具体做法是对于每个使用到的TimerTaskList都会加入到DelayQueue中。Kafka中的TimingWheel专门用来执行插入和删除TimerTaskEntry的操作，而DelayQueue专门负责时间推进的任务。再试想一下，DelayQueue中的第一个超时任务列表的expiration为200ms，第二个超时任务为840ms，这里获取DelayQueue的队头只需要 $O(1)$ 的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的200次推进中有199次属于“空推进”，而获取到第二个超时任务时有需要执行639次“空推进”，这样会无故空耗机器的性能资源，这里采用DelayQueue来辅助以少量空间换时间，从而做到了“精准推进”。Kafka中的定时器真可谓是“知人善用”，用TimingWheel做最擅长的任务添加和删除操作，而用DelayQueue做最擅长的时间推进工作，相辅相成。

# RocketMQ的事务消息是如何实现的

- a. 生产者订单系统先发送一条half消息到Broker，half消息对消费者而言是不可见的
- b. 再创建订单，根据创建订单成功与否，向Broker发送commit或rollback
- c. 并且生产者订单系统还可以提供Broker回调接口，当Broker发现一段时间half消息没有收到任何操作命令，则会主动调此接口来查询订单是否创建成功
- d. 一旦half消息commit了，消费者库存系统就会来消费，如果消费成功，则消息销毁，分布式事务成功结束
- e. 如果消费失败，则根据重试策略进行重试，最后还失败则进入死信队列，等待进一步处理

## 为什么RocketMQ不使用Zookeeper作为注册中心呢？

根据CAP理论，同时最多只能满足两个点，而zookeeper满足的是CP，也就是说zookeeper并不能保证服务的可用性，zookeeper在进行选举的时候，整个选举的时间太长，期间整个集群都处于不可用的状态，而对于一个注册中心来说肯定是不能接受的，作为服务发现来说就应该是为可用性而设计。

基于性能的考虑，NameServer本身的实现非常轻量，而且可以通过增加机器的方式水平扩展，增加集群的抗压能力，而zookeeper的写是不可扩展的，而zookeeper要解决这个问题只能通过划分领域，划分多个zookeeper集群来解决，首先操作起来太复杂，其次这样还是又违反了CAP中的A的设计，导致服务之间是不连通的。

持久化的机制带来的问题，ZooKeeper 的 ZAB 协议对每一个写请求，会在每个 ZooKeeper 节点上保持写一个事务日志，同时再加上定期的将内存数据镜像（Snapshot）到磁盘来保证数据的一致性和持久性，而对于一个简单的服务发现的场景来说，这其实没有太大的必要，这个实现方案太重了。而且本身存储的数据应该是高度定制化的。

消息发送应该弱依赖注册中心，而RocketMQ的设计理念也正是基于此，生产者在第一次发送消息的时候从NameServer获取到Broker地址后缓存到本地，如果NameServer整个集群不可用，短时间内对于生产者和消费者

并不会产生太大影响。

## RocketMQ的实现原理

RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成，它的架构原理是这样的：

Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳

Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器来发送消息

Consumer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费

## RocketMQ为什么速度快

因为使用了顺序存储、Page Cache和异步刷盘。我们在写入commitlog的时候是顺序写入的，这样比随机写入的性能就会提高很多，写入commitlog的时候并不是直接写入磁盘，而是先写入操作系统的PageCache，最后由操作系统异步将缓存中的数据刷到磁盘

## 消息队列如何保证消息可靠传输

消息可靠传输代表了两层意思，既不能多也不能少。

1. 为了保证消息不多，也就是消息不能重复，也就是生产者不能重复生产消息，或者消费者不能重复消费消息
2. 首先要确保消息不多发，这个不常出现，也比较难控制，因为如果出现了多发，很大的原因是生产者自己的原因，如果要避免出现问题，就需要在消费端做控制
3. 要避免不重复消费，最保险的机制就是消费者实现幂等性，保证就算重复消费，也不会有问题，通过幂等性，也能解决生产者重复发送消息的问题
4. 消息不能少，意思就是消息不能丢失，生产者发送的消息，消费者一定要能消费到，对于这个问题，就要考虑两个方面
5. 生产者发送消息时，要确认broker确实收到并持久化了这条消息，比如RabbitMQ的confirm机制，Kafka的ack机制都可以保证生产者能正确的将消息发送给broker
6. broker要等待消费者真正确认消费到了消息时才删除掉消息，这里通常就是消费端ack机制，消费者接收到一条消息后，如果确认没问题了，就可以给broker发送一个ack，broker接收到ack后才会删除消息

## 消息队列有哪些作用

1. 解耦：使用消息队列来作为两个系统之间的通讯方式，两个系统不需要相互依赖了
2. 异步：系统A给消息队列发送完消息之后，就可以继续做其他事情了
3. 流量削峰：如果使用消息队列的方式来调用某个系统，那么消息将在队列中排队，由消费者自己控制消费速度

## 死信队列是什么？延时队列是什么？

1. 死信队列也是一个消息队列，它是用来存放那些没有成功消费的消息的，通常可以用来作为消息重试

2. 延时队列就是用来存放需要在指定时间被处理的元素的队列，通常可以用来处理一些具有过期性操作的业务，比如十分钟内未支付则取消订单

## 如何保证消息的高效读写？

零拷贝： kafka和RocketMQ都是通过零拷贝技术来优化文件读写。

传统文件复制方式： 需要对文件在内存中进行四次拷贝。

零拷贝： 有两种方式， mmap和transfile， Java当中对零拷贝进行了封装， Mmap方式通过MappedByteBuffer对象进行操作，而transfile通过FileChannel来进行操作。 Mmap 适合比较小的文件，通常文件大小不要超过1.5G ~2G 之间。 Transfile没有文件大小限制。 RocketMQ当中使用Mmap方式来对他的文件进行读写。

在kafka当中，他的index日志文件也是通过mmap的方式来读写的。在其他日志文件当中，并没有使用零拷贝的方式。 Kafka使用transfile方式将硬盘数据加载到网卡。

## 让你设计一个MQ，你会如何设计？

两个误区：

1. 放飞自我，漫无边际。
2. 纠结技术细节。

好的方式：

1. 从整体到细节，从业务场景到技术实现。
2. 以现有产品为基础。

答题思路： MQ作用、项目大概的样子。

1. 实现一个单机的队列数据结构。 高效、可扩展。
2. 将单机队列扩展成为分布式队列。 - 分布式集群管理
3. 基于Topic定制消息路由策略。 - 发送者路由策略，消费者与队列对应关系，消费者路由策略
4. 实现高效的网络通信。 - Netty Http
5. 规划日志文件，实现文件高效读写。 - 零拷贝，顺序写。 服务重启后，快速还原运行现场。
6. 定制高级功能，死信队列、延迟队列、事务消息等等。 - 贴合实际，随意发挥。

## 网络(11)

### 什么是认证和授权？如何设计一个权限认证框架？

认证： 就是对系统访问者的身份进行确认。

授权： 就是对系统访问者的行为进行控制。 授权通常是在认证之后，对系统内的用户隐私数据进行保护。 后台接口访问权限、前台控件的访问权限。

RBAC模型： 主体 -> 角色 -> 资源 -> 访问系统的行为。

认证和授权也是对一个权限认证框架进行扩展的两个主要的方面。

## 如果没有Cookie,Session还能进行身份验证吗？

当服务器tomcat第一次接收到客户端的请求时，会开辟一块独立的session空间，建立一个session对象，同时会生成一个session id，通过响应头的方式保存到客户端浏览器的cookie当中。以后客户端的每次请求，都会在请求头部带上这个session id，这样就可以对应上服务端的一些会话的相关信息，比如用户的登录状态。

如果没有客户端的Cookie，Session是无法进行身份验证的。

当服务端从单体应用升级为分布式之后，cookie+session这种机制要怎么扩展？

1. session黏贴：在负载均衡中，通过一个机制保证同一个客户端的所有请求都会转发到同一个tomcat实例当中。  
问题：当这个tomcat实例出现问题之后，请求就会被转发到其他实例，这时候用户的session信息就丢了。
2. session复制：当一个tomcat实例上保存了session信息后，主动将session 复制到集群中的其他实例。  
问题：复制是需要时间的，在复制过程中，容易产生session信息丢失。
3. session共享：就是将服务端的session信息保存到一个第三方中，比如Redis。

## 什么是CSRF攻击？如何防止？

CSRF：Cross Site Request Forgery 跨站请求伪造，一个正常的请求会将合法用户的session id保存到浏览器的cookie。这时候，如果用户在浏览器中打来另一个tab页，那这个tab页也是可以获得浏览器的cookie。黑客就可以利用这个cookie信息进行攻击。

攻击过程：

1. 某银行网站A可以以GET请求的方式发起转账操作。[www.xxx.com/transfor.do?accountNum=100&money=1000](http://www.xxx.com/transfor.do?accountNum=100&money=1000) accountNum表示目标账户。这个请求肯定是需要登录才可以正常访问的。
2. 攻击者在某个论坛或者网站上，上传一个图片，链接地址是 [www.xxx.com/transfer.do?accountNum=888&money=10000](http://www.xxx.com/transfer.do?accountNum=888&money=10000) 其中这个accountNum就是攻击者自己的银行账户。
3. 如果有一个用户，登录了银行网站，然后又打开浏览器的另一个tab页，点击了这个图片。这时，银行就会受理到一个带了正确cookie的请求，就会完成转账。用户的钱就被盗了。

CSRF防止方式：

1. 尽量使用POST请求，限制GET请求。POST请求可以带请求体，攻击者就不容易伪造出请求。

2. 将cookie设置为HttpOnly : `response.setHeader("Set-Cookie","cookiename=cookievalue;HttpOnly")`。
3. 增加token ;
4. 在请求中放入一个攻击者无法伪造的信息，并且该信息不存在于cookie当中。这也是Spring Security框架中采用的防范方式。

## 什么是OAuth2.0协议？有哪几种认证方式？

OAuth2.0是一个开放标准，允许用户授权第三方应用程序访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方应用或分享他们数据的所有内容。

OAuth2.0协议的认证流程，简单理解，就是允许我们将之前的授权和认证过程交给一个独立的第三方进行担保。

OAuth2.0协议有四种认证方式：

1. 授权码模式
2. 简化模式
3. 密码模式
4. 客户端模式

## 什么是SSO？与OAuth2.0有什么关系？

OAuth2.0的使用场景通常称为联合登录，一处注册，多处使用

SSO Single Sign On 单点登录。 一处登录，多处同时登录

SSO的实现关键是将Session信息集中存储

## 如何设计一个开放授权平台？

开放授权平台也可以按照认证和授权两个方向来梳理。

1. 认证：就可以按照OAuth2.0协议来规划认证的过程。
2. 授权：

- a. 首先需要待接入的第三方应用在开放授权平台进行注册，注册需要提供几个必要的信息 clintID, 消息推送地址，密钥(一对公私钥，私钥由授权平台自己保存，公钥分发给第三方应用)。
- b. 然后，第三方应用引导客户发起请求时，采用公钥进行参数加密，授权开放平台使用对应的私钥解密。
- c. 接下来：授权开放平台同步响应第三方应用的只是消息是否处理成功的结果。而真正的业务数据由授权开放平台异步推动给第三方应用预留的推送地址。

## epoll和poll的区别

1. select模型，使用的是数组来存储Socket连接文件描述符，容量是固定的，需要通过轮询来判断是否发生了IO事件
2. poll模型，使用的是链表来存储Socket连接文件描述符，容量是不固定的，同样需要通过轮询来判断是否发生了IO事件
3. epoll模型，epoll和poll是完全不同的，epoll是一种事件通知模型，当发生了IO事件时，应用程序才进行IO操作，不需要像poll模型那样主动去轮询

## TCP的三次握手和四次挥手

TCP协议是7层网络协议中的传输层协议，负责数据的可靠传输。

在建立TCP连接时，需要通过三次握手来建立，过程是：

1. 客户端向服务端发送一个SYN
2. 服务端接收到SYN后，给客户端发送一个SYN\_ACK
3. 客户端接收到SYN\_ACK后，再给服务端发送一个ACK

在断开TCP连接时，需要通过四次挥手来断开，过程是：

1. 客户端向服务端发送FIN
2. 服务端接收FIN后，向客户端发送ACK，表示我接收到了断开连接的请求，客户端你可以不发数据了，不过服务端这边可能还有数据正在处理
3. 服务端处理完所有数据后，向客户端发送FIN，表示服务端现在可以断开连接
4. 客户端收到服务端的FIN，向服务端发送ACK，表示客户端也会断开连接了

## 浏览器发出一个请求到收到响应经历了哪些步骤？

1. 浏览器解析用户输入的URL，生成一个HTTP格式的请求
2. 先根据URL域名从本地hosts文件查找是否有映射IP，如果没有就将域名发送给电脑所配置的DNS进行域名解析，得到IP地址
3. 浏览器通过操作系统将请求通过四层网络协议发送出去
4. 途中可能会经过各种路由器、交换机，最终到达服务器
5. 服务器收到请求后，根据请求所指定的端口，将请求传递给绑定了该端口的应用程序，比如8080被tomcat占用了

6. tomcat接收到请求数据后，按照http协议的格式进行解析，解析得到所要访问的servlet
7. 然后servlet来处理这个请求，如果是SpringMVC中的DispatcherServlet，那么则会找到对应的Controller中的方法，并执行该方法得到结果
8. Tomcat得到响应结果后封装成HTTP响应的格式，并再次通过网络发送给浏览器所在的服务器
9. 浏览器所在的服务器拿到结果后再传递给浏览器，浏览器则负责解析并渲染

## 跨域请求是什么？有什么问题？怎么解决？

跨域是指浏览器在发起网络请求时，会检查该请求所对应的协议、域名、端口和当前网页是否一致，如果不一致则浏览器会进行限制，比如在www.baidu.com的某个网页中，如果使用ajax去访问www.jd.com是不行的，但是如果、、等标签的src属性去访问则是可以的，之所以浏览器要做这层限制，是为了用户信息安全。但是如果开发者想要绕过这层限制也是可以的：

1. response添加header，比如resp.setHeader("Access-Control-Allow-Origin", "\*");表示可以访问所有网站，不受是否同源的限制
2. jsonp的方式，该技术底层就是基于script标签来实现的，因为script标签是可以跨域的
3. 后台自己控制，先访问同域名下的接口，然后在接口中再去使用HTTPClient等工具去调用目标接口
4. 网关，和第三种方式类似，都是交给后台服务来进行跨域访问

## 零拷贝是什么

零拷贝指的是，应用程序在需要把内核中的一块区域数据转移到另外一块内核区域去时，不需要经过先复制到用户空间，再转移到目标内核区域去了，而直接实现转移。

图灵学院周瑜

图灵学院周瑜

图灵学院周瑜

## Leetcode算法 (10)

### 反转链表

反转一个单链表。

1 输入: 1->2->3->4->5  
2 输出: 5->4->3->2->1

Java | Copy Code

解法1：迭代，重复某一过程，每一次处理结果作为下一次处理的初始值，这些初始值类似于状态、每次处理都会改变状态、直至到达最终状态

从前往后遍历链表，将当前节点的next指向上一个节点，因此需要一个变量存储上一个节点prev，当前节点处理完需要寻找下一个节点，因此需要一个变量保存当前节点curr，处理完后要将当前节点赋值给prev，并将next指针赋值给curr，因此需要一个变量提前保存下一个节点的指针next

学院周瑜

1、将下一个节点指针保存到next变量 `next = curr.next`

2、将下一个节点的指针指向prev，`curr.next = prev`

3、准备处理下一个节点，将curr赋值给prev

4、将下一个节点赋值为curr，处理一个节点

解法2：递归：以相似的方法重复，类似于树结构，先从根节点找到叶子节点，从叶子节点开始遍历 大的问题(整个链表反转)拆成性质相同的小问题(两个元素反转)`curr.next.next = curr`

将所有的小问题解决，大问题即解决

只需每个元素都执行`curr.next.next = curr`，`curr.next = null`两个步骤即可为了保证链不断，必须从最后一个元素开始

```

1  public class ReverseList {
2      static class ListNode{
3          int val;
4          ListNode next;
5
6          public ListNode(int val, ListNode next) { t
7              his.val = val;
8              this.next = next;
9          }
10     }
11
12     public static ListNode iterate(ListNode head){
13         ListNode prev = null, curr, next;
14         curr = head;
15         while(curr != null){
16             next = curr.next;
17             curr.next = prev;
18             prev = curr;
19             curr = next;
20         }
21         return prev;
22     }
23
24     public static ListNode recursion(ListNode head) {
25         if (head == null || head.next == null) {
26             return head;
27         }
28         ListNode newHead = recursion(head.next);
29         head.next.next = head;
30         head.next = null;
31         return newHead;
32     }
33
34     public static void main(String[] args) {
35         ListNode node5 = new ListNode(5,null);
36         ListNode node4 = new ListNode(4,node5);
37         ListNode node3 = new ListNode(3,node4);
38         ListNode node2 = new ListNode(2,node3);
39         ListNode node1 = new ListNode(1,node2);
40         //ListNode node = iterate(node1); ListNode node_1 = recursion(node1);
41         System.out.println(node_1);
42     }

```

## 统计N以内的素数

素数：只能被1和自身整除的数，0、1除外

解法一：暴力算法

直接从2开始遍历，判断是否能被2到自身之间的数整除

Java | Copy Code

```
1 public int countPrimes(int n) {
2     int ans = 0;
3     for (int i = 2; i < n; ++i) {
4         ans += isPrime(i) ? 1 : 0;
5     }
6     return ans;
7 }
8
9 //i如果能被x整除，则x/i肯定能被x整除，因此只需判断i和根号x之中较小的即可
10 public boolean isPrime(int x) {
11     for (int i = 2; i * i <= x; ++i) {
12         if (x % i == 0) {
13             return false;
14         }
15     }
16     return true;
17 }
```

## 解法2：埃氏筛

利用合数的概念(非素数)，素数\*n必然是合数，因此可以从2开始遍历，将所有的合数做上标记

Java | Copy Code

```
1 public static int eratosthenes(int n) {
2     boolean[] isPrime = new boolean[n];
3     int ans = 0;
4     for (int i = 2; i < n; i++) {
5         if (!isPrime[i]) {
6             ans += 1;
7             for (int j = i * i; j < n; j += i) {
8                 isPrime[j] = true;
9             }
10        }
11    }
12    return ans;
13 }
```

将合数标记为true， $j = i * i$  从  $2 * i$  优化而来，系数2会随着遍历递增 ( $j += i$ ，相当于递增了系数2)，每一个合数都会有两个比本身要小的因子(0,1除外)， $2 * i$  必然会遍历到这两个因子

当2递增到大于根号n时，其实后面的已经无需再判断（或者只需判断后面一段），而2到根号n、实际上在 i 递增的过程中已经计算过了，i 实际上就相当于根号n

例如： $n = 25$  会计算以下

$$2 * 4 = 8$$

$3 * 4 = 12$

但实际上8和12已经标记过，在 $n = 17$ 时已经计算了  $3 * 4, 2 * 4$

## 寻找数组的中心索引

数组中某一个下标，左右两边的元素之后相等，该下标即为中心索引  
思路：先统计出整个数组的总和，然后从第一个元素开始叠加

总和递减当前元素，叠加递增当前元素，知道两个值相等

```
1 public static int pivotIndex(int[] nums) {  
2     int sum1 = Arrays.stream(nums).sum();  
3     int sum2 = 0;  
4     for(int i = 0; i < nums.length; i++){  
5         sum2 += nums[i];  
6         if(sum1 == sum2){  
7             return i;  
8         }  
9         sum1 = sum1 - nums[i];  
10    }  
11    return -1;  
12 }
```

Java | Copy Code

## 删除排序数组中的重复项

一个有序数组  $\text{nums}$ ，原地删除重复出现的元素，使每个元素只出现一次，返回删除后数组的新长度。  
不要使用额外的数组空间，必须在原地修改输入数组并在使用  $O(1)$  额外空间的条件下完成。

双指针算法：

数组完成排序后，我们可以放置两个指针  $i$  和  $j$ ，其中  $i$  是慢指针，而  $j$  是快指针。只要  $\text{nums}[i] = \text{nums}[j]$ ，我们就增加  $j$  以跳过重复项。

当遇到  $\text{nums}[j] \neq \text{nums}[i]$  时，跳过重复项的运行已经结束，必须把  $\text{nums}[j]$  的值复制到  $\text{nums}[i + 1]$ 。然后递增  $i$ ，接着将再次重复相同的过程，直到  $j$  到达数组的末尾为止。

```

1 public int removeDuplicates(int[] nums) {
2     if (nums.length == 0) return 0;
3     int i = 0;
4     for (int j = 1; j < nums.length; j++) {
5         if (nums[j] != nums[i]) {
6             i++;
7             nums[i] = nums[j];
8         }
9     }
10    return i + 1;
11 }

```

## x的平方根

在不使用 `sqrt(x)` 函数的情况下，得到 x 的平方根的整数部分

解法一：二分查找

x 的平方根肯定在 0 到 x 之间，使用二分查找定位该数字，该数字的平方一定是最接近 x 的，m 平方值如果大于 x，则往左边找，如果小于等于 x 则往右边找

找到 0 和 x 的最中间的数 m，

如果  $m * m > x$ ，则 m 取  $x/2$  到 x 的中间数字，直到  $m * m < x$ ，m 则为平方根的整数部分

如果  $m * m \leq x$ ，则取 0 到  $x/2$  的中间值，知道两边的界限重合，找到最大的整数，则为 x 平方根的整数部分

时间复杂度： $O(\log N)$

```

1 public static int binarySearch(int x) {
2     int l = 0, r = x, index = -1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         if ((long) mid * mid <= x) {
6             index = mid; l = mid + 1;
7         } else {
8             r = mid - 1;
9         }
10    }
11    return index;
12 }

```

解法二：牛顿迭代

假设平方根是 i，则 i 和  $x/i$  必然都是 x 的因子，而  $x/i$  必然等于 i，推导出  $i + x/i = 2*i$ ，得出  $i = (i + x/i)/2$

由此得出解法，i 可以任选一个值，只要上述公式成立，i 必然就是 x 的平方根，如果不成立， $(i + x / i) / 2$  得出的值进行递归，直至得出正确解

Java | Copy Code

```
1 public static int newton(int x) {  
2     if(x==0) return 0;  
3     return ((int)(sqrt(x,x)));  
4 }  
5  
6 public static double sqrt(double i,int x){  
7     double res = (i + x / i) / 2;  
8     if (res == i) {  
9         return i;  
10    } else {  
11        return sqrt(res,x);  
12    }  
13 }
```

## 三个数的最大乘积

一个整型数组乘积不会越界，在数组中找出由三个数字组成的大乘积，并输出这个乘积。

如果数组中全是非负数，则排序后最大的三个数相乘即为最大乘积；如果全是非正数，则最大的三个数 相乘同样也为最大乘积。

如果数组中有正数有负数，则最大乘积既可能是三个最大正数的乘积，也可能是两个最小负数（即绝对 值最大）与最大正数的乘积。

分别求出三个最大正数的乘积，以及两个最小负数与最大正数的乘积，二者之间的最大值即为所求答 案。

### 解法一：排序

Java | Copy Code

```
1 public static int sort(int[] nums) {  
2     Arrays.sort(nums);  
3     int n = nums.length;  
4     return Math.max(nums[0] * nums[1] * nums[n - 1], nums[n - 3] * nums[n - 2] * nums[n - 1]);  
5 }
```

### 解法二：线性扫描

[Java](#) | [Copy Code](#)

```
1 public static int getMaxMin(int[] nums) {  
2     // 最小的和第二小的  
3     int min1 = 0, min2 = 0;  
4     // 最大的、第二大的和第三大的  
5     int max1 = 0, max2 = 0, max3 = 0;  
6  
7     for (int x : nums) {  
8  
9         if (x < min1) {  
10            min2 = min1; min1 = x;  
11        } else if (x < min2) {  
12            min2 = x;  
13        }  
14  
15        if (x > max1) {  
16            max3 = max2;  
17            max2 = max1;  
18            max1 = x;  
19        } else if (x > max2) {  
20            max3 = max2;  
21            max2 = x;  
22        } else if (x > max3) {  
23            max3 = x;  
24        }  
25    }  
26}  
27}
```

## 两数之和

给定一个升序排列的整数数组 `numbers`，从数组中找出两个数满足相加之和等于目标数 `target`。假设每个输入只对应唯一的答案，而且不可以重复使用相同的元素。

返回两数的下标值，以数组形式返回暴力解法

[Java](#) | [Copy Code](#)

```
1 public int[] twoSum(int[] nums, int target) {  
2     int n = nums.length;  
3     for (int i = 0; i < n; ++i) {  
4         for (int j = i + 1; j < n; ++j) {  
5             if (nums[i] + nums[j] == target) {  
6                 return new int[]{i, j};  
7             }  
8         }  
9     }  
10    return new int[0];  
11}
```

时间复杂度 : O(N的平方) 空间复杂度 : O(1)

图灵学院网

哈希表：将数组的值作为key存入map，target - num作为key

Java | Copy Code

```
1 public int[] twoSum(int[] nums, int target) {  
2     Map<Integer, Integer> map = new HashMap<Integer, Integer>();  
3     for (int i = 0; i < nums.length; ++i) {  
4         if (map.containsKey(target - nums[i])) {  
5             return new int[]{map.get(target - nums[i]), i};  
6         }  
7         map.put(nums[i], i);  
8     }  
9     return new int[0];  
10 }
```

时间复杂度 : O(N) 空间复杂度 : O(N)

图灵学院网

解法一：二分查找

先固定一个值(从下标0开始)，再用二分查找查另外一个值，找不到则固定值向右移动，继续二分查找

Java | Copy Code

```
1 public int[] twoSearch(int[] numbers, int target) {  
2     for (int i = 0; i < numbers.length; ++i) {  
3         int low = i, high = numbers.length - 1;  
4         while (low <= high) {  
5             int mid = (high - low) / 2 + low;  
6             if (numbers[mid] == target - numbers[i]) {  
7                 return new int[]{i, mid};  
8             } else if (numbers[mid] > target - numbers[i]) {  
9                 high = mid - 1;  
10            } else {  
11                low = mid + 1;  
12            }  
13        }  
14    }  
15 }
```

时间复杂度 : O(N \* logN)

图灵学院网

空间复杂度 : O(1)

解法二：双指针

左指针指向数组head，右指针指向数组tail，head+tail > target 则tail 左移，否则head右移

```
1 public int[] twoPoint(int[] numbers, int target) {  
2     int low = 0, high = numbers.length - 1;  
3  
4     while (low < high) {  
5         int sum = numbers[low] + numbers[high];  
6         if (sum == target) {  
7             return new int[]{low + 1, high + 1};  
8         } else if (sum < target) {  
9             ++low;  
10        } else {  
11            --high;  
12        }  
13    }  
14  
15    return new int[]{-1, -1};  
16 }
```

时间复杂度 : O(N) 空间复杂度 : O(1)

## 斐波那契数列

求取斐波那契数列第N位的值。

斐波那契数列 : 每一位的值等于他前两位数字之和。前两位固定 0 , 1,1,2,3,5,8。。。。

解法一：暴力递归

```
1 public static int calculate(int num){  
2     if(num == 0 ){  
3         return 0;  
4     }  
5     if(num == 1){  
6         return 1;  
7     }  
8     return calculate(num-1) + calculate(num-2);  
9 }
```

## 解法二：去重递归

递归得出具体数值之后、存储到一个集合(下标与数列下标一致)，后面递归之前先到该集合查询一次，如果查到则无需递归、直接取值。查不到再进行递归计算

```
1 public static int calculate2(int num){  
2     int[] arr = new int[num+1];  
3     return recurse(arr,num);  
4 }  
5  
6 private static int recurse(int[] arr, int num) {  
7     if(num == 0 ){  
8         return 0;  
9     }  
10    if(num == 1){  
11        return 1;  
12    }  
13  
14    if(arr[num] != 0){  
15        return arr[num];  
16    }  
17  
18    arr[num] = recurse(arr,num-1) + recurse(arr,num-2);  
19    return arr[num];  
20 }  
21 }
```

### 解法三：双指针迭代

基于去重递归优化，集合没有必要保存每一个下标值，只需保存前两位即可，向后遍历，得出N的值

[Java](#) | [Copy Code](#)

```
1 public static int iterate(int num){  
2     if(num == 0 ){  
3         return 0;  
4     }  
5  
6     if(num == 1){  
7         return 1;  
8     }  
9  
10    int low = 0,high = 1;  
11    for(int i=2; i<= num; i++){  
12        int sum = low + high; low = high;  
13        high = sum;  
14    }  
15    return high;  
16 }
```

## 环形链表

给定一个链表，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达该节点，则链表中存在环。如果链表中存在环，则返回 true 。否则，返回 false 。

解法一：哈希表

[Java](#) | [Copy Code](#)

```
1 public static boolean hasCycle(ListNode head) {  
2     Set<ListNode> seen = new HashSet<ListNode>();  
3     while (head != null) {  
4         if (!seen.add(head)) {  
5             return true;  
6         }  
7         head = head.next;  
8     }  
9     return false;  
10 }
```

解法二：双指针

```

1 public static boolean hasCycle2(ListNode head) {
2     if (head == null || head.next == null) {
3         return false;
4     }
5
6     ListNode slow = head; ListNode fast = head.next;
7     while (slow != fast) {
8         if (fast == null || fast.next == null) {
9             return false;
10        }
11        slow = slow.next; fast = fast.next.next;
12    }
13    return true;
14 }
```

## 排列硬币

总共有  $n$  枚硬币，将它们摆成一个阶梯形状，第  $k$  行就必须正好有  $k$  枚硬币。给定一个数字  $n$ ，找出可形成完整阶梯行的总行数。

$n$  是一个非负整数，并且在32位有符号整型的范围内

解法一：迭代

从第一行开始排列，排完一列、计算剩余硬币数，排第二列，直至剩余硬币数小于或等于行数

```

1
2 public static int arrangeCoins(int n) {
3     for(int i=1; i<=n; i++){
4         n = n-i;
5         if (n <= i){
6             return i;
7         }
8     }
9     return 0;
10 }
```

解法二：二分查找

假设能排  $n$  行，计算  $n$  行需要多少硬币数，如果大于  $n$ ，则排  $n/2$  行，再计算硬币数和  $n$  的大小关系

```

1 public static int arrangeCoins2(int n) {
2     int low = 0, high = n;
3     while (low <= high) {
4         long mid = (high - low) / 2 + low; long cost = ((mid + 1) * mid) / 2;
5         if (cost == n) {
6             return (int)mid;
7         } else if (cost > n) {
8             high = (int)mid - 1;
9         } else {
10            low = (int)mid + 1;
11        }
12    }
13 }
14 return high;
15 }
```

### 解法三：牛顿迭代

使用牛顿迭代求平方根， $(x + n/x)/2$

假设能排  $x$  行，则  $1 + 2 + 3 + \dots + x = n$ ，即  $x(x+1)/2 = n$  推导出  $x = 2n - x$

```

1 public static double sqrtS(double x,int n){
2     double res = (x + (2*n-x) / x) / 2;
3     if (res == x) {
4         return x;
5     } else {
6         return sqrtS(res,n);
7     }
8 }
```