Java 面试宝典 9.0

目录

一 Java 基础

1.Java 基础 知识

1.1 面向对象的特征（了解）

面向对象的特征：封装、继承、多态、抽象。

封装：就是把对象的属性和行为（数据）结合为一个独立的整体，并尽可能隐藏对象的内部实现细节，就是把不想告诉或者不该告诉别人的东西隐藏起来，把可以告诉别人的公开，别人只能用我提供的功能实现需求，而不知道是如何实现的。增加安全性。

继承：子类继承父类的数据属性和行为，并能根据自己的需求扩展出新的行为，提高了代码的复用性。

多态：指允许不同的对象对同一消息做出相应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式（发送消息就是函数调用）。封装和继承几乎都是为多态而准备的，在执行期间判断引用对象的实际类型，根据其实际的类型调用其相应的方法。

抽象表示对问题领域进行分析、设计中得出的抽象的概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。在 Java 中抽象用 abstract 关键字来修饰，用 abstract 修饰类时，此类就不能被实例化，从这里可以看出，抽象类（接口）就是为了继承而存在的。

1.2 Java 的基本数据类型有哪些（了解）

	数据类型	字节数	位数
整型	byte	1	8
	short	2	16
	int	4	32
	long	8	64
浮点型	float	4	32
	double	8	64
布尔型	boolean	1	8
字符型	char	2	16

1.3JDK　JRE　JVM 的区别　(必会)

JDK（Java Development Kit）是整个 Java 的核心，是 java 开发工具包，包括了 Java 运行环境 JRE、Java 工具和 Java 基础类库。

JRE（Java Runtime Environment）是运行 JAVA 程序所必须的环境的集合，包含 java 虚拟机和 java 程序的一些核心类库。

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，是整个 java 实现跨平台的最核心的部分，能够运行以 Java 语言写作的软件程序。

1.4 重载和重写的区别（必会）

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。

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。

1.5 Java 中==和 equals 的区别（必会）

== 的作用：

基本类型：比较的就是值是否相同

引用类型：比较的就是地址值是否相同

equals 的作用:

引用类型：默认情况下，比较的是地址值。

特：String、Integer、Date 这些类库中 equals 被重写，比较的是内容而不是地址！

面试题：请解释字符串比较之中 “ == ” 和 equals() 的区别?

答： ==：比较的是两个字符串内存地址（堆内存）的数值是否相等，属于数值比较；

equals()：比较的是两个字符串的内容，属于内容比较。

1.6 String、StringBuffer、StringBuilder 三者之间的区别（必会）

String 字符串常量

StringBuffer 字符串变量（线程安全）

StringBuilder 字符串变量（非线程安全）

String 中的 String 类中使用 final 关键字修饰字符数组来保存字符串，private final char value[] ，**String** 对象是不可变的，也就可以理解为常量，线程安全。

AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。

StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

小结：

（1）如果要操作少量的数据用 String；

（2）多线程操作字符串缓冲区下操作大量数据用 StringBuffer；

（3）单线程操作字符串缓冲区下操作大量数据用 StringBuilder。

1.7 接口和抽象类的区别是什么？（必会）

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

构造函数：抽象类可以有构造函数；接口不能有。

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

实现数量：类可以实现很多个接口；但是只能继承一个抽象类。

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

1.8 string 常用的方法有哪些？（了解）

indexOf()：返回指定字符的索引。
charAt()：返回指定索引处的字符。
replace()：字符串替换。
trim()：去除字符串两端空白。
split()：分割字符串，返回一个分割后的字符串数组。
getBytes()：返回字符串的 byte 类型数组。
length()：返回字符串长度。
toLowerCase()：将字符串转成小写字母。
toUpperCase()：将字符串转成大写字符。
substring()：截取字符串。
equals()：字符串比较。

1.9 什么是单例模式？有几种？（必会）

单例模式：某个类的实例在 多线程环境下只会被创建一次出来。
单例模式有饿汉式单例模式、懒汉式单例模式和双检锁单例模式三种。
饿汉式：线程安全，一开始就初始化。

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

懒汉式：非线程安全，延迟初始化。

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

双检锁：线程安全，延迟初始化。

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
}
```

1.10 反射（了解）

在 Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的

属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息

以及动态调用对象方法的功能成为 Java 语言的反射机制。

获取 Class 对象的 3 种方法 ：

调用某个对象的 getClass()方法

Person p=new Person();

Class clazz=p.getClass();

调用某个类的 class 属性来获取该类对应的 Class 对象

Class clazz=Person.class;

使用 Class 类中的 forName()静态方法(最安全/性能最好)

Class clazz=Class.forName("类的全路径"); (最常用)

1.11 jdk1.8 的新特性（高薪常问）

- **1 Lambda 表达式**

  Lambda 允许把函数作为一个方法的参数。

  ```
  new Thread(()-> System.out.println("abc")).start();
  ```

- **2 方法引用**

  方法引用允许直接引用已有 Java 类或对象的方法或构造方法。

```
1.   ArrayList<String> list = new ArrayList<>();
2.         list.add("a");
3.         list.add("b");
4.         list.add("c");
5.         list.forEach(System.out::println);
```

上例中我们将 System.out::println 方法作为静态方法来引用。

- **3 函数式接口**

有且仅有一个抽象方法的接口叫做函数式接口，函数式接口可以被隐式转换为 Lambda 表达式。通常函数式接口

上会添加@FunctionalInterface 注解。

- **4 接口允许定义默认方法和静态方法**

从 JDK8 开始，允许接口中存在一个或多个默认非抽象方法和静态方法。

- **5 Stream API**

新添加的 Stream API（java.util.stream）把真正的函数式编程风格引入到 Java 中。这种风格将要处理的元素集

合看作一种流，流在管道中传输，并且可以在管道的节点上进行处理，比如筛选，排序，聚合等。

```
1.   List<String> list = Arrays.asList("abc", "", "abc", "bc", "efg", "abcd", "def", "jkl");
2.
3.        list.stream()//获取集合的流对象
4.              .filter(string -> !string.isEmpty())//对数据进行过滤操作，过滤掉空字符串
5.              .distinct()//去重
6.              .forEach(a -> System.out.println(a));
```

- **6 日期/时间类改进**

之前的 JDK 自带的日期处理类非常不方便，我们处理的时候经常是使用的第三方工具包，比如 commons-lang

包等。不过 JDK8 出现之后这个改观了很多，比如日期时间的创建、比较、调整、格式化、时间间隔等。

这些类都在 java.time 包下，LocalDate/LocalTime/LocalDateTime。

- **7 Optional 类**

Optional 类是一个可以为 null 的容器对象。如果值存在则 isPresent()方法会返回 true，调用 get()方法会返回该对象。

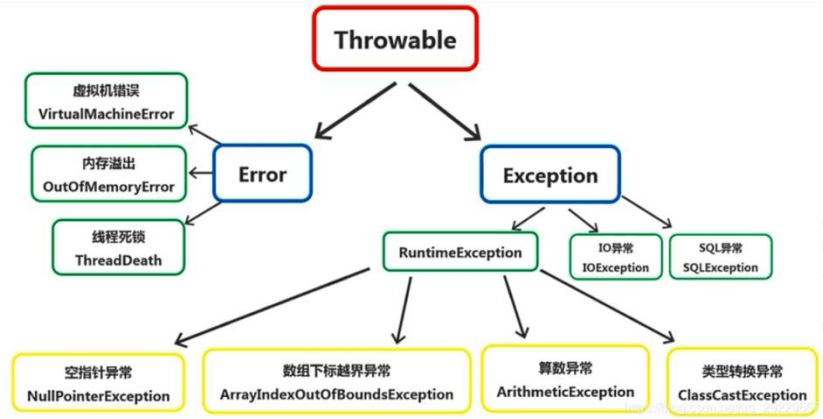
```
String string = "abc";
```

```
Optional<String> optional = Optional.of(string);
boolean present = optional.isPresent();
String value = optional.get();
System.out.println(present+"/"+value);
```

- **8 Java8 Base64 实现**

  Java 8 内置了 Base64 编码的编码器和解码器。

1.12 Java 的异常（必会）



Throwable 是所有 Java 程序中错误处理的父类，有两种资类：Error 和 Exception。

Error：表示由 JVM 所侦测到的无法预期的错误，由于这是属于 JVM 层次的严重错误，导致 JVM 无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。

Exception：表示可恢复的例外，这是可捕捉到的。

1.运行时异常：都是 RuntimeException 类及其子类异常，如 NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。运行时异常的特点是 Java 编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也会编译通过。

2.非运行时异常（编译异常）：是 RuntimeException 以外的异常，类型上都属于 Exception 类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如 IOException、SQLException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。

常见的 RunTime 异常几种如下：

NullPointerException - 空指针引用异常
ClassCastException - 类型强制转换异常。
IllegalArgumentException - 传递非法参数异常。
ArithmeticException - 算术运算异常
ArrayStoreException - 向数组中存放与声明类型不兼容对象异常
IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常
NumberFormatException - 数字格式异常
SecurityException - 安全异常
UnsupportedOperationException - 不支持的操作异常

1.13 BIO、NIO、AIO 有什么区别？（高薪常问）

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

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

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

1.14 Threadloal 的原理（高薪常问）

ThreadLocal：为共享变量在每个线程中创建一个副本，每个线程都可以访问自己内部的副本变量。通过 threadlocal 保证线程的安全性。

其实在 ThreadLocal 类中有一个静态内部类 ThreadLocalMap(其类似于 Map)，用键值对的形式存储每一个线程的变量副本，ThreadLocalMap 中元素的 key 为当前 ThreadLocal 对象，而 value 对应线程的变量副本。

ThreadLocal 本身并不存储值，它只是作为一个 key 保存到 ThreadLocalMap 中，但是这里要注意的是它作为一个 key 用的是弱引用，因为没有强引用链，弱引用在 GC 的时候可能会被回收。这样就会在 ThreadLocalMap 中存在一些 key 为 null 的键值对（Entry）。因为 key 变成 null 了，我们是没法访问这些 Entry 的，但是这些 Entry 本身是不会被清除的。如果没有手动删除对应 key 就会导致这块内存即不会回收也无法访问，也就是内存泄漏。

使用完 ThreadLocal 之后，记得调用 remove 方法。 在不使用线程池的前提下，即使不调用 remove 方法，线程的"变量副本"也会被 gc 回收，即不会造成内存泄漏的情况。

1.16 同步锁、死锁、乐观锁、悲观锁 （高薪常问）

同步锁：

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

死锁：

何为死锁，就是多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

乐观锁：

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_conditio 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

悲观锁：

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java 中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。

1.17 说一下 synchronized 底层实现原理？（高薪常问）

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的 class 对象
- 同步方法块，锁是括号里面的对象

1.18 synchronized 和 volatile 的区别是什么？（高薪常问）

volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。

volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。

volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

1.19synchronized 和 Lock 有什么区别？ （高薪常问）

首先 synchronized 是 java 内置关键字，在 jvm 层面，Lock 是个 java 类；

synchronized 无法判断是否获取锁的状态，Lock 可以判断是否获取到锁；

synchronized 会自动释放锁(a 线程执行完同步代码会释放锁 ； b 线程执行过程中发生异常会释放锁)，Lock 需在 finally 中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；

用 synchronized 关键字的两个线程 1 和线程 2，如果当前线程 1 获得锁，线程 2 线程等待。如果线程 1 阻塞，线程 2 则会一直等待下去，而 Lock 锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；

synchronized 的锁可重入、不可中断、非公平，而 Lock 锁可重入、可判断、可公平（两者皆可）；

Lock 锁适合大量同步的代码的同步问题，synchronized 锁适合代码少量的同步问题。

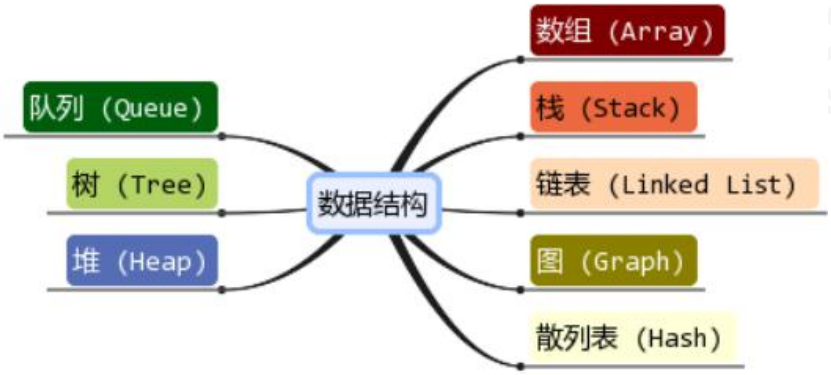
1.20 手写冒泡排序？(必会)

```
public class Sort {
 public static void sort() {
     Scanner input = new Scanner(System.in);
     int sort[] = new int[10];
     int temp;
     System.out.println("请输入 10 个排序的数据：");
     for (int i = 0; i < sort.length; i++) {
         sort[i] = input.nextInt();
     }
     for (int i = 0; i < sort.length - 1; i++) {
         for (int j = 0; j < sort.length - i - 1; j++)          {
             if (sort[j] < sort[j + 1]) {
                 temp = sort[j];
                 sort[j] = sort[j + 1];
                 sort[j + 1] = temp;
             }
         }
     }
     System.out.println("排列后的顺序为：");
     for(int i=0;i<sort.length;i++){
         System.out.print(sort[i]+"======");
     }
 }
 public static void main(String[] args) {
     sort();
 }
}
```

2.集合(必会)

2.1 常见的数据结构（了解）

常用的数据结构有：数组，栈，队列，链表，树，散列，堆，图等



数组是最常用的数据结构，数组的特点是长度固定，数组的大小固定后就无法扩容了，数组只能存储一种类型的数据，添加，删除的操作慢，因为要移动其他的元素。

栈是一种基于先进后出（FILO）的数据结构，是一种只能在一端进行插入和删除操作的特殊线性表。它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。

队列是一种基于先进先出（FIFO）的数据结构，是一种只能在一端进行插入，在另一端进行删除操作的特殊线性表，它按照先进先出的原则存储数据，先进入的数据，在读取数据时先被读取出来。

链表是一种物理存储单元上非连续、非顺序的存储结构，其物理结构不能只表示数据元素的逻辑顺序，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列的结节（链表中的每一个元素称为结点）组成，结点可以在运行时动态生成。根据指针的指向，链表能形成不同的结构，例如单链表，双向链表，循环链表等。

树是我们计算机中非常重要的一种数据结构，同时使用树这种数据结构，可以描述现实生活中的很多事物，例如家谱、单位的组织架构等等。有二叉树、平衡树、红黑树、B树、B+树。

散列表，也叫哈希表，是根据关键码和值（key 和 value）直接进行访问的数据结构，通过 key 和 value 来映射到集合中的一个位置，这样就可以很快找到集合中的对应元素。

堆是计算机学科中一类特殊的数据结构的统称，堆通常可以被看作是一棵完全二叉树的数组对象。

图的定义：图是由一组顶点和一组能够将两个顶点相连的边组成的

2.2 集合和数组的区别（了解）

区别：数组长度固定　集合长度可变

数组中存储的是同一种数据类型的元素，可以存储基本数据类型，也可以存储引用数据类型；

集合存储的都是对象，而且对象的数据类型可以不一致。在开发当中一般当对象较多的时候，使用集合来存储对象。

2.3List 和 Map、Set 的区别（必会）

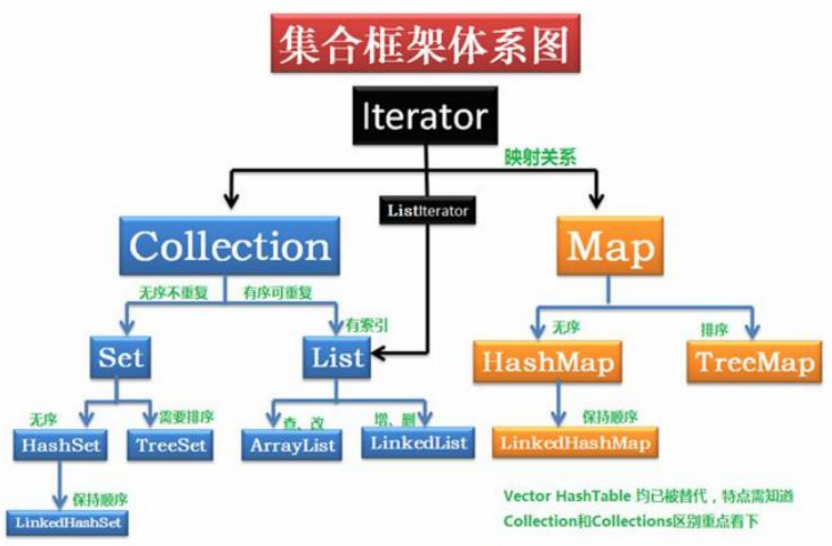
List 和 Set 是存储单列数据的集合，Map 是存储键值对这样的双列数据的集合；

List 中存储的数据是有顺序的，并且值允许重复；

Map 中存储的数据是无序的，它的键是不允许重复的，但是值是允许重复的；

Set 中存储的数据是无顺序的，并且不允许重复，但元素在集合中的位置是由元素的 hashcode 决定，即位置是固定的（Set 集合是根据 hashcode 来进行数据存储的，所以位置是固定的，但是这个位置不是用户可以控制的，所以对于用户来说 set 中的元素还是无序的）。

2.4 List 和 Map、Set 的实现类（必会）



- **Connection 接口:**

List 有序，可重复

ArrayList

优点: 底层数据结构是数组，查询快，增删慢。

缺点: 线程不安全，效率高

Vector

优点: 底层数据结构是数组，查询快，增删慢。

缺点: 线程安全，效率低，已给舍弃了

LinkedList

优点: 底层数据结构是链表，查询慢，增删快。

缺点: 线程不安全，效率高

Set 无序,唯一

HashSet

底层数据结构是哈希表。(无序,唯一)

如何来保证元素唯一性?

依赖两个方法：hashCode()和 equals()

LinkedHashSet

底层数据结构是链表和哈希表。(FIFO 插入有序,唯一)

1.由链表保证元素有序

2.由哈希表保证元素唯一

TreeSet

底层数据结构是红黑树。(唯一，有序)

1. 如何保证元素排序的呢?

自然排序

比较器排序

2.如何保证元素唯一性的呢?

根据比较的返回值是否是 0 来决定

- **Map 接口有四个实现类：**

HashMap

基于 hash 表的 Map 接口实现，非线程安全，高效，支持 null 值和 null 键，线程不安全。

HashTable

线程安全，低效，不支持 null 值和 null 键；

LinkedHashMap

线程不安全，是 HashMap 的一个子类，保存了记录的插入顺序；

TreeMap

能够把它保存的记录根据键排序，默认是键值的升序排序，线程不安全。

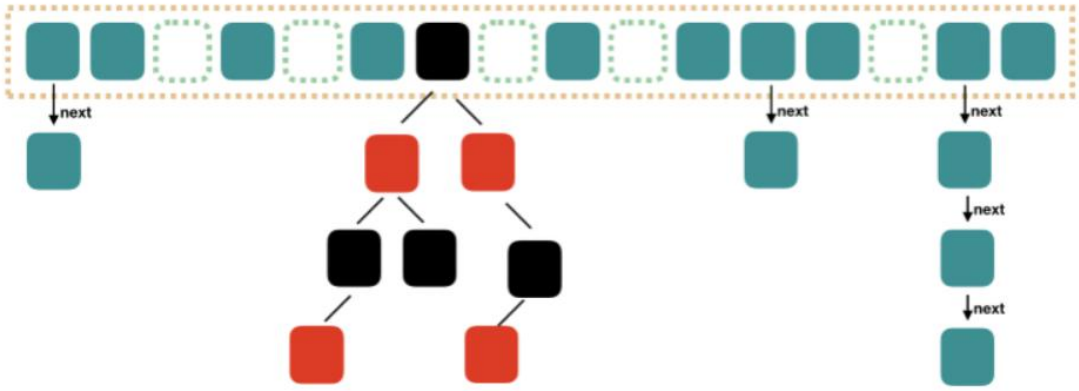
2.5Hashmap 的底层原理（高薪常问）

HashMap 在 JDK1.8 之前的实现方式 **数组+链表,**

但是在 JDK1.8 后对 HashMap 进行了底层优化,改为了由 **数组+链表或者数值+红黑树** 实现,主要的目的是提高查找效率

JDK版本	实现方式	节点数>=8	节点数<=6
1.8以前	数组+单向链表	数组+单向链表	数组+单向链表
1.8以后	数组+单向链表+红黑树	数组+红黑树	数组+单向链表

Java8 HashMap 结构



1. **Jdk8 数组+链表或者数组+红黑树**实现，当链表中的元素超过了 8 个以后， 会将链表转换为红黑树，当红黑树节点 小于 等于 6 时又会退化为链表。

2. 当 new HashMap():底层没有创建数组，首次调用 put()方法示时，底层创建长度为 16 的数组，jdk8 底层的数组是：Node[],而非 Entry[]，用数组容量大小乘以加载因子得到一个值，一旦数组中存储的元素个数超过该值就会调用 rehash 方法将数组容量增加到原来的两倍，专业术语叫做扩容，在做扩容的时候会生成一个新的数组，原来的所有数据需要重新计算哈希码值重新分配到新的数组，所以扩容的操作非常消耗性能.

默认的负载因子大小为 0.75，数组大小为 16。也就是说，默认情况下，那么当 HashMap 中元素个数超过 16*0.75=12 的时候，就把数组的大小扩展为 2*16=32，即扩大一倍。

3. 在我们 Java 中任何对象都有 hashcode，hash 算法就是通过 hashcode 与自己进行向右位移 16 的异或运算。这样做是为了计算出来的 hash 值足够随机，足够分散，还有产生的数组下标足够随机，

map.put(k,v)实现原理

（1）首先将 k,v 封装到 Node 对象当中（节点）。

（2）先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。

（3）下标位置上如果没有任何元素，就把 Node 添加到这个位置上。如果说下标对应的位置上有链表。此时，就会拿着 k 和链表上每个节点的 k 进行 equal。如果所有的 equals 方法返回都是 false，那么这个新的节点将被添加到链表的末尾。如其中有一个 equals 返回了 true，那么这个节点的 value 将会被覆盖。

map.get(k)实现原理

(1)、先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。

(2)、在通过数组下标快速定位到某个位置上。重点理解如果这个位置上什么都没有，则返

回 null。如果这个位置上有单向链表，那么它就会拿着参数 K 和单向链表上的每一个节点的 K 进行 equals，如果所有 equals 方法都返回 false，则 get 方法返回 null。如果其中一个节点的 K 和参数 K 进行 equals 返回 true，那么此时该节点的 value 就是我们要找的 value 了，get 方法最终返回这个要找的 value。

4. **Hash 冲突**

不同的对象算出来的数组下标是相同的这样就会产生 hash 冲突，当单线链表达到一定长度后效率会非常低。

5. 在链表长度大于 8 的时候，将链表就会变成红黑树，提高查询的效率。

2.6 Hashmap 和 hashtable ConcurrentHashMap 区别（高薪常问）

区别对比一(HashMap 和 HashTable 区别)：

1、HashMap 是非线程安全的，HashTable 是线程安全的。

2、HashMap 的键和值都允许有 null 值存在，而 HashTable 则不行。

3、因为线程安全的问题，HashMap 效率比 HashTable 的要高。

4、Hashtable 是同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而 Hashtable 适合于多线程环境。一般现在不建议用 HashTable，①是 HashTable 是遗留类，内部实现很多没优化和冗余。②即使在多线程环境下，现在也有同步的 ConcurrentHashMap 替代，没有必要因为是多线程而用 HashTable。

区别对比二(HashTable 和 ConcurrentHashMap 区别)：

HashTable 使用的是 Synchronized 关键字修饰，ConcurrentHashMap 是 JDK1.7 使用了锁分段技术来保证线程安全的。JDK1.8ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

3.多线程(必会)

3.1 什么是线程?线程和进程的区别?（了解）

线程：是进程的一个实体，是 cpu 调度和分派的基本单位，是比进程更小的可以独立运行的基本单位。

进程：具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位。

特点：线程的划分尺度小于进程，这使多线程程序拥有高并发性，进程在运行时各自内存单元相互独立，线程之间 内存共享，这使多线程编程可以拥有更好的性能和用户体验。

3.2 创建线程有几种方式（必会）

1.继承 Thread 类并重写 run 方法创建线程，实现简单但不可以继承其他类

2.实现 Runnable 接口并重写 run 方法。避免了单继承局限性，编程更加灵活，实现解耦。

3..实现 Callable 接口并重写 call 方法，创建线程。可以获取线程执行结果的返回值，并且可以抛出异常。

4.使用线程池创建（使用 java.util.concurrent.Executor 接口）

```
//第一种
Thread thread=new Thread();
thread.start();

//第二种
Thread thread1=new Thread(new Runnable() {
    @Override
    public void run() {

    }
});
thread1.start();
//第三种
FutureTask<String> task=new FutureTask<String>(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return null;
    }
});
Thread thread2=new Thread(task);
thread2.start();

```

```
//第四种
ExecutorService es= Executors.newFixedThreadPool(1);
es.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() );
    }
});
  es.shutdown();
```

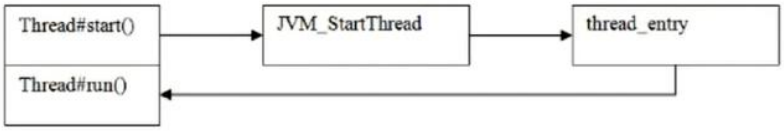
3.3Runnable 和 Callable 的区别？（必会）

主要区别

Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，支持泛型

Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息

3.4 如何启动一个新线程、调用 start 和 run 方法的区别？（必会）



线程对象调用 run 方法不开启线程。仅是对象调用方法。

线程对象调用 start 开启线程，并让 jvm 调用 run 方法在开启的线程中执行

调用 start 方法可以启动线程，并且使得线程进入就绪状态，而 run 方法只是 thread 的一个普通方法，还是在主线程中执行。

3.5 线程有哪几种状态以及各种状态之间的转换？(必会)

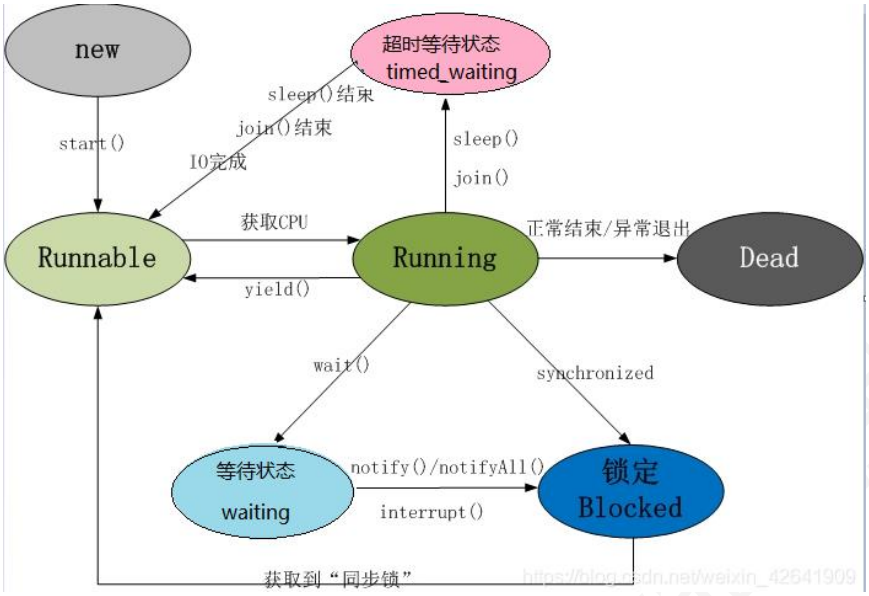
1. 第一是 new->**新建状态。**在生成线程对象，**并没有调用该对象的 start 方法**，这是线程处于创建状态。

2. 第二是 Runnable->**就绪状态。**当**调用了线程对象的 start 方法**之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。

3. 第三是 Running->**运行状态。**线程调度程序将处于就绪状态的线程**设置为当前线程**，此时线程就进入了运行状态，**开始运行 run 函数当中的代码。**

4. 第四是**阻塞状态。**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。阻塞的情况分三种：
(1)等待 – 通过调用线程的 **wait()** 方法，让线程等待某工作的完成。
(2)超时等待 – 通过调用线程的 **sleep()** 或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。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。
(3)同步阻塞 – 线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。

5. 第五是 dead->**死亡状态**: 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.



3.6 线程相关的基本方法？（必会）

线程相关的基本方法有 wait，notify，notifyAll，sleep，join，yield 等

1.线程等待（wait）

调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意的是调用 wait()方法后，会释放对象的锁。因此，wait 方法一般用在同步方法或同步代码块中。

2.线程睡眠（sleep）

sleep 导致当前线程休眠，与 wait 方法不同的是 sleep 不会释放当前占有的锁,sleep(long)会导致线程进入 TIMED-WATING 状态，而 wait()方法会导致当前线程进入 WATING 状态.

3.线程让步（yield）

yield 会使当前线程让出 CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。一般情况下，优先级高的线程有更大的可能性成功竞争得到 CPU 时间片，但这又不是绝对的，有的操作系统对 线程优先级并不敏感。

4.线程中断（interrupt）

中断一个线程，其本意是给这个线程一个通知信号，会影响这个线程内部的一个中断标识位。这个线程本身并不会因此而改变状态(如阻塞，终止等)

5.Join 等待其他线程终止

join() 方法，等待其他线程终止，在当前线程中调用一个线程的 join() 方法，则当前线程转为阻塞状态，回到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 的宠幸.

6.线程唤醒（notify）

Object 类中的 notify() 方法，唤醒在此对象监视器上等待的单个线程，如果所有线程都在此对象上等待，则会选择唤醒其中一个线程，选择是任意的，并在对实现做出决定时发生，线程通过调用其中一个 wait() 方法，在对象的监视器上等待，直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程，被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。类似的方法还有 notifyAll() ，唤醒再次监视器上等待的所有线程。

3.7 wait()和 sleep()的区别？（必会）

1. 来自不同的类

wait():来自 Object 类；

sleep():来自 Thread 类；

2.关于锁的释放：

wait():在等待的过程中会释放锁；

sleep():在等待的过程中不会释放锁

3.使用的范围：

wait():必须在同步代码块中使用；

sleep():可以在任何地方使用；

4.是否需要捕获异常

wait():不需要捕获异常；

sleep():需要捕获异常；

4. 线程池

4.1 为什么需要线程池（了解）

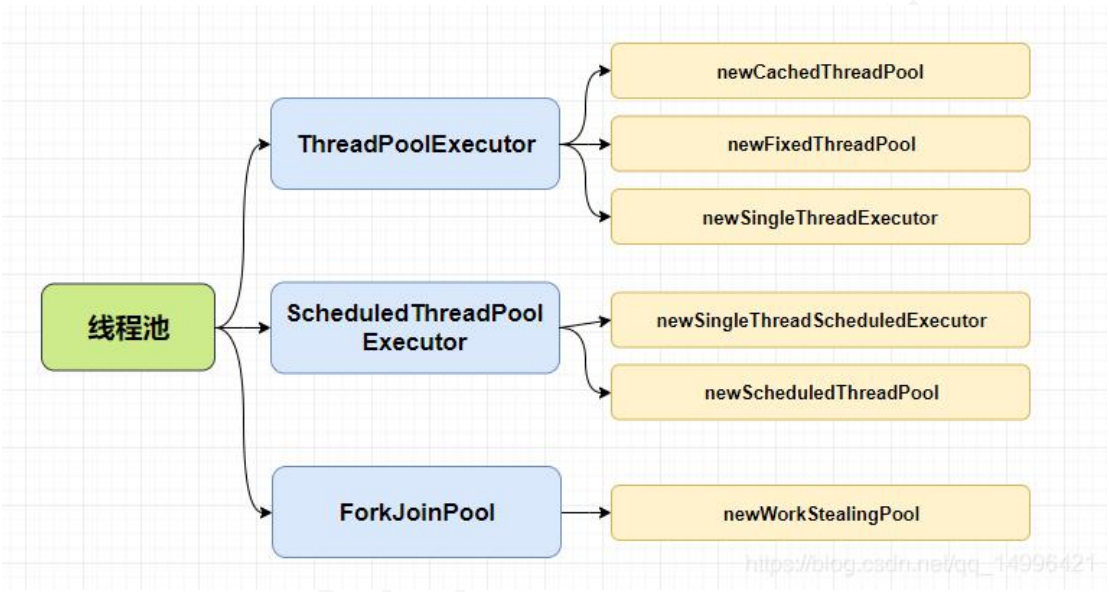
在实际使用中，线程是很占用系统资源的，如果对线程管理不完善的话很容易导致系统问题。因此，在大多数并发框架中都会使用线程池来管理线程，使用线程池管理线程主要有如下好处：

1、使用线程池可以重复利用已有的线程继续执行任务，避免线程在创建销毁时造成的消耗

2、由于没有线程创建和销毁时的消耗，可以提高系统响应速度

3、通过线程可以对线程进行合理的管理，根据系统的承受能力调整可运行线程数量的大小等

4.2 线程池的分类（高薪常问）



1. **newCachedThreadPool**：创建一个可进行缓存重复利用的线程池
2. **newFixedThreadPool**：创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程，线程池中的线程处于一定的量，可以很好的控制线程的并发量
3. **newSingleThreadExecutor**：创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。线程池中最多执行一个线程，之后提交的线程将会排在队列中以此执行
4. **newSingleThreadScheduledExecutor**：创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期执行
5. **newScheduledThreadPool**：创建一个线程池，它可安排在给定延迟后运行命令或者定期的执行
6. **newWorkStealingPool**：创建一个带并行级别的线程池，并行级别决定了同一时刻最多有多少个线程在执行，如不传并行级别参数，将默认为当前系统的 CPU 个数

4.3 核心参数（高薪常问）

corePoolSize：核心线程池的大小

maximumPoolSize：线程池能创建线程的最大个数

keepAliveTime：空闲线程存活时间

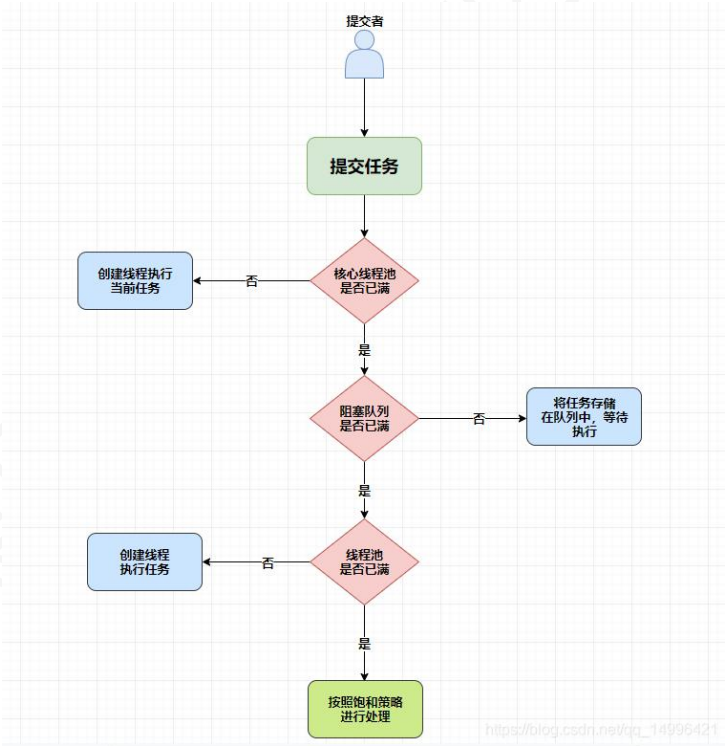
unit：时间单位，为 keepAliveTime 指定时间单位

workQueue：阻塞队列，用于保存任务的阻塞队列

threadFactory：创建线程的工程类

handler：饱和策略（拒绝策略）

4.4 线程池的原理（高薪常问）



线程池的工作过程如下：

当一个任务提交至线程池之后，

1. 线程池首先判断核心线程池里的线程是否已经满了。如果不是，则创建一个新的工作线程来执行任务。否则进入 2.

2. 判断工作队列是否已经满了，倘若还没有满，将线程放入工作队列。否则进入 3.
3. 判断线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行。如果线程池满了，则交给饱和策略来处理任务。

4.5 拒绝策略（了解）

ThreadPoolExecutor.AbortPolicy（系统默认）： 丢弃任务并抛出 RejectedExecutionException 异常，让你感知到任务被拒绝了，我们可以根据业务逻辑选择重试或者放弃提交等策略

ThreadPoolExecutor.DiscardPolicy： 也是丢弃任务，但是不抛出异常，相对而言存在一定的风险，因为我们提交的时候根本不知道这个任务会被丢弃，可能造成数据丢失。
ThreadPoolExecutor.DiscardOldestPolicy： 丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程），通常是存活时间最长的任务，它也存在一定的数据丢失风险
ThreadPoolExecutor.CallerRunsPolicy：既不抛弃任务也不抛出异常，而是将某些任务回退到调用者，让调用者去执行它。

4.6 线程池的关闭（了解）

关闭线程池，可以通过 shutdown 和 shutdownNow 两个方法

原理：遍历线程池中的所有线程，然后依次中断

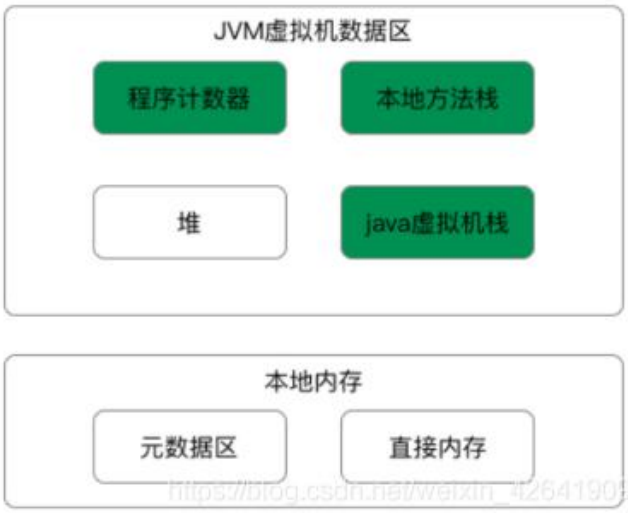
1、shutdownNow 首先将线程池的状态设置为 STOP,然后尝试停止所有的正在执行和未执行任务的线程，并返回等待执行任务的列表；

2、shutdown 只是将线程池的状态设置为 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程

6. Jvm

虚拟机，一种能够运行 java 字节码的虚拟机。

6.1 JDK1.8 JVM 运行时内存（高薪）



程序计数器:
线程私有的(每个线程都有一个自己的程序计数器)，是一个指针．代码运行，执行命令．而每个命令都是有行号的,会使用程序计数器来记录命令执行到多少行了.记录代码执行的位置

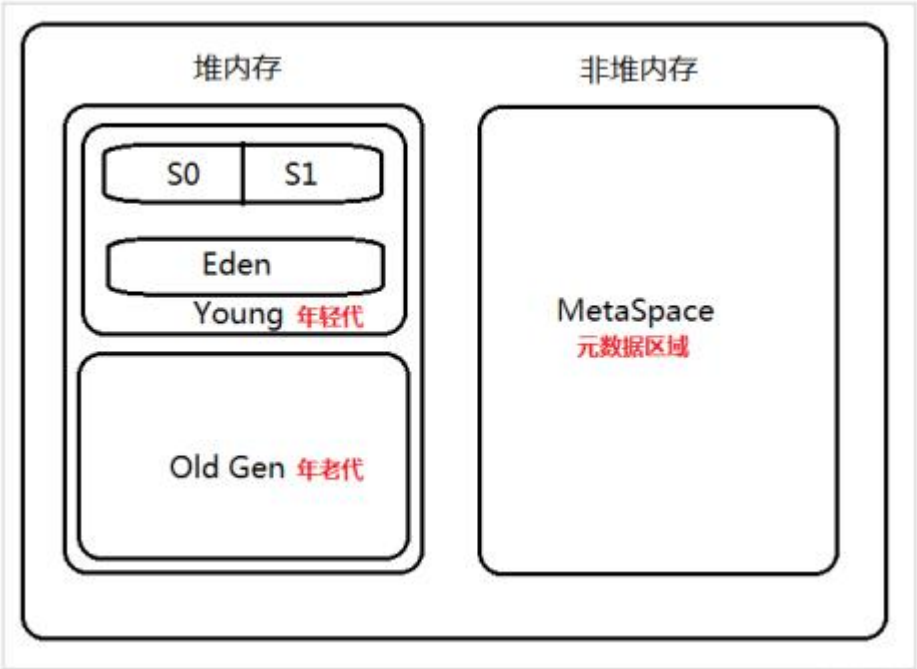
Java 虚拟机栈:
线程私有的(每个线程都有一个自己的 Java 虚拟机栈)．一个方法运行，就会给这个方法创建一个栈帧，栈帧入栈执行代码，执行完毕之后出栈(弹栈)存引用变量，基本数据类型

本地方法栈:
线程私有的(每个线程都有一个自己的本地方法栈)，和 Java 虚拟机栈类似, Java 虚拟机栈加载的是普通方法,本地方法加载的是 **native** 修饰的方法.
native:在 java 中有用 native 修饰的,表示这个方法不是 java 原生的.

堆:
线程共享的(所有的线程共享一份)．存放对象的,new 的对象都存储在这个区域.还有就是常量池.
元空间: 存储.class 信息，类的信息,方法的定义,静态变量等.而常量池放到堆里存储

JDK1.8 和 JDK1.7 的 jvm 内存最大的区别是，在 1.8 中方法区是由元空间(元数据区)来实现的，常量池.
1.8 不存在方法区,将方法区的实现给去掉了.而是在本地内存中,新加入元数据区(元空间).

6.2 JDK1.8 堆内存结构（高薪常问）



Young 年轻区（代）：Eden+S0+S1, S0 和 S1 大小相等，新创建的对象都在年轻代

Tenured 年老区: 经过年轻代多次垃圾回收存活下来的对象存在年老代中.

Jdk1.7 和 Jdk1.8 的区别在于, 1.8 将永久代中的对象放到了元数据区，不存永久代这一区域了.

6.3 Gc 垃圾回收（高薪常问）

JVM 的垃圾回收动作可以大致分为两大步，首先是「如何发现垃圾」，然后是「如何回收垃圾」。说明一点，线程私有的不存在垃圾回收，只有线程共享的才会存在垃圾回收，所以堆中存在垃圾回收.

6.3.1 如何发现垃圾

Java 语言规范并没有明确的说明 JVM 使用哪种垃圾回收算法，但是常见的用于「发现垃圾」的算法有两种，引用计数算法和根搜索算法。

1．引用计数算法

该算法很古老（了解即可）。核心思想是，堆中的对象每被引用一次，则计数器加 1，每减少一个引用就减 1，当对象的引用计数器为 0 时可以被当作垃圾收集。

优点：快。

缺点：无法检测出循环引用。如两个对象互相引用时，他们的引用计数永远不可能为 0。

2．根搜索算法(也叫可达性分析)

根搜索算法是把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即可以当作垃圾。

Java 中可作为 GC Root 的对象有

1.虚拟机栈中引用的对象

2.本地方法栈引用的对象

2.方法区中静态属性引用的对象

3.方法区中常量引用的对象

6.3.2 如何回收垃圾

Java 中用于「回收垃圾」的常见算法有 4 种：

1．标记-清除算法（mark and sweep)

分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成之后统一回收掉所有被标记的对象。

缺点：首先，效率问题，标记和清除效率都不高。其次，标记清除之后会产生大量的不连续的内存碎片。

2．标记-整理算法

是在标记-清除算法基础上做了改进，标记阶段是相同的，但标记完成之后不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，在移动过程中清理掉可回收的对象，这个过程叫做整理。

优点：内存被整理后不会产生大量不连续内存碎片。

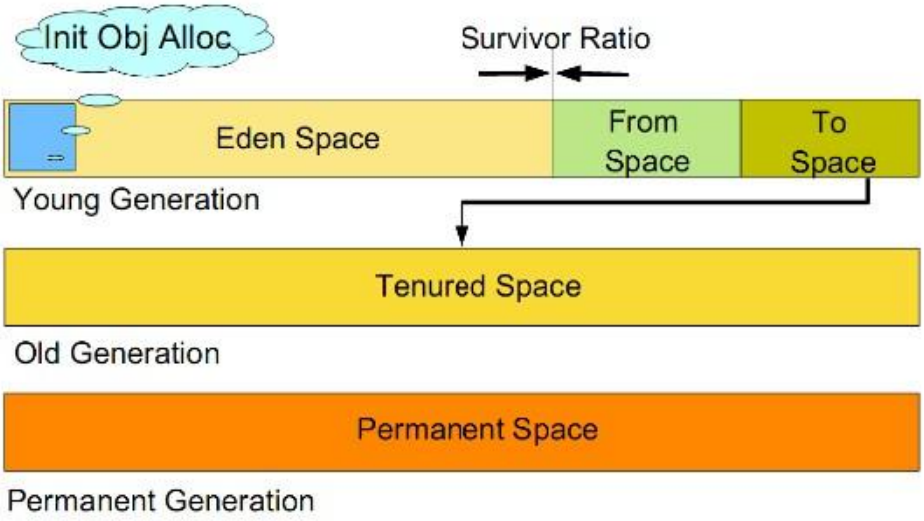
3．复制算法（copying)

将可用内存按容量分成大小相等的两块，每次只使用其中一块，当这块内存使用完了，就将还存活的对象复制到另一块内存上去，然后把使用过的内存空间一次清理掉。

缺点：可使用的内存只有原来一半。

4．分代收集算法（generation)

当前主流 JVM 都采用分代收集(Generational Collection)算法，这种算法会根据对象存活周期的不同将内存划分为年轻代、年老代、永久代，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。



(1) 年轻代（Young Generation）

1.所有新生成的对象首先都是放在年轻代的。

2.新生代内存按照 8:1:1 的比例分为一个 eden 区和两个 Survivor(survivor0,survivor1)区。大部分对象在 Eden 区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 和这个 survivor0 区，此时 survivor0 区是空的，然后将 survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。

3.当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC，也就是新生代、老年代都进行回收。

4.新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)

(2) 年老代（Old Generation）

1.在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2.内存比新生代也大很多(大概是 2 倍)，当老年代内存满时触发 Major GC 即 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率比较高。

(3) 持久代（Permanent Generation）

用于存放静态文件，如 Java 类、方法等。持久代对垃圾回收没有显著影响，从 JDK8 以后已经废弃，将存放静态文件，如 Java 类、方法等这些存储到了元数据区.

6.4 JVM 调优参数（了解）

这里只给出一些常见的性能调优的参数及其代表的含义。(大家记住 5.6 个就行，并不需要都记住.)

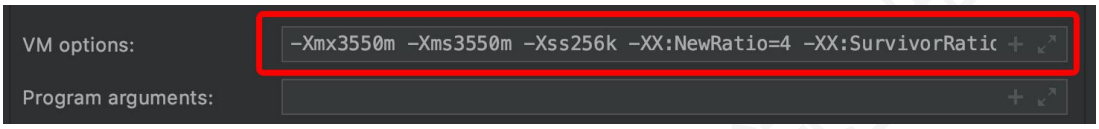
```
VM options:    -Xmx3550m -Xms3550m -Xmn2g -Xss256k
```

-Xmx3550m：设置 JVM 最大可用内存为 3550M。

-Xms3550m：设置 JVM 初始内存为 3550m。注意：此值一般设置成和-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g：设置年轻代大小为 2G。整个 JVM 内存大小=年轻代大小 + 年老代大小 + 持久代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss256k：设置每个线程的栈大小。JDK5.0 以后每个线程栈大小为 1M，以前每个线程栈大小为 256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。



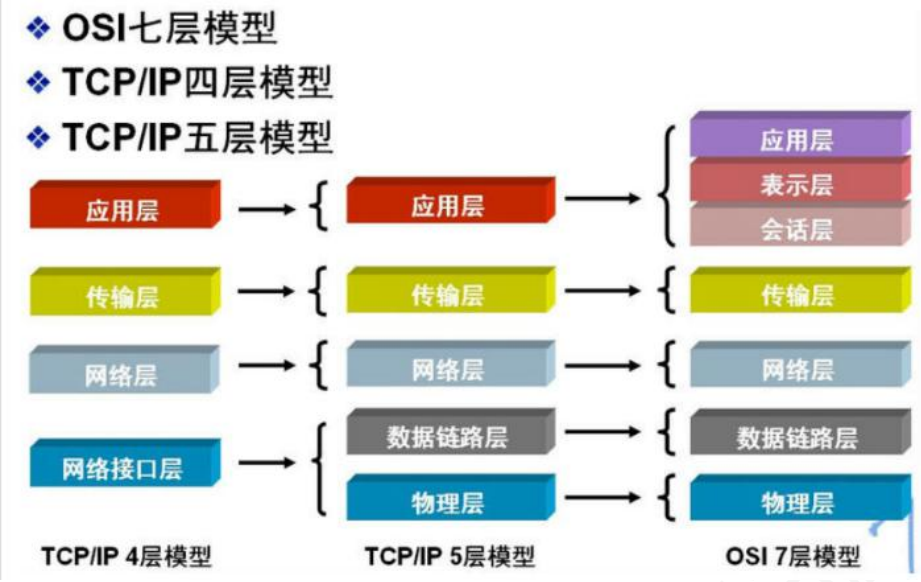
-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4。（该值默认为 2）

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4。

一、Web

1.网络通讯部分

1.1 TCP 与 UDP 区别? （了解）



TCP(Transmission Control Protocol 传输控制协议)是一种面向连接(连接导向)的、可靠的、 基于 IP 的传输层协议。

UDP 是 User Datagram Protocol 的简称，中文名是用户数据报协议，是 OSI 参考模型中的传输层协议，它是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

TCP 和 UDP 都是来自于传输层的协议。传输层位于应用层和网络层之间，负责位于不同主机中进程之间的通信。

TCP 与 UDP 区别

类别	TCP	UDP
是否连接	面向连接	面向非连接
传输可靠性	可靠	不可靠
应用场合	少量数据	传输大量数据
速度	慢	快

1.TCP 基于连接 UDP 无连接
2.TCP 要求系统资源较多，UDP 较少
3.TCP 保证数据正确性，UDP 可能丢包
4.TCP 保证数据顺序，UDP 不保证

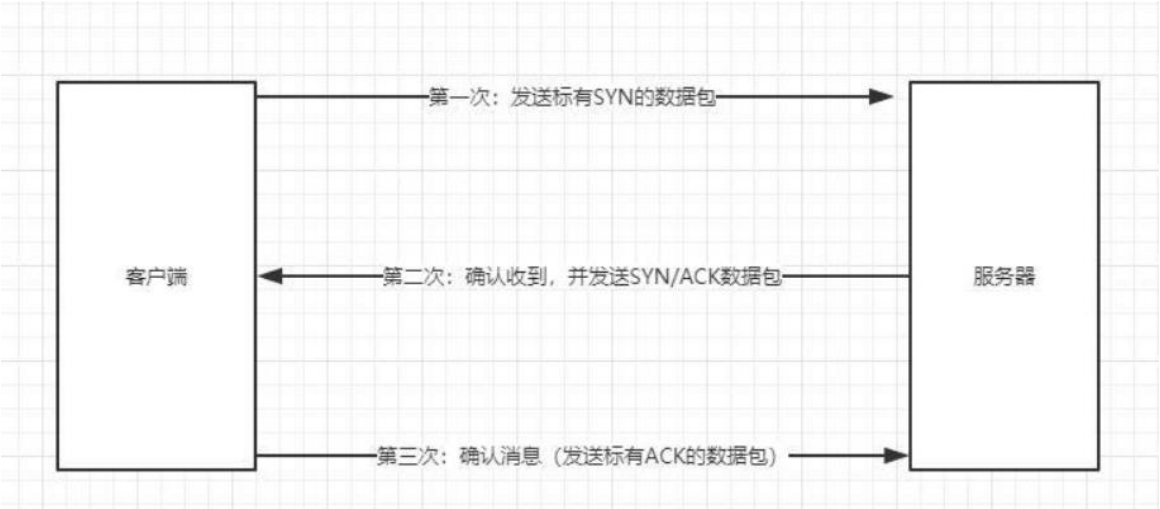
1.2 什么是 HTTP 协议?

客户端和服务器端之间数据传输的格式规范，格式简称为“超文本传输协议”。

是一个基于请求与响应模式的、无状态的、应用层的协议，基于 TCP 的连接方式

1.3 TCP 的三次握手

为了准确无误地把数据送达目标处，TCP 协议采用了三次握手策略。



为什么要三次握手?

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是 双方确认自己与对方的发送与接收是正常的。

SYN：同步序列编号（Synchronize Sequence Numbers）。是 TCP/IP 建立连接时使用的握手信号。

第一次握手：客户端给服务器发送一个 SYN。客户端发送网络包，服务端收到了。服务器得出结论：客户端的发送能力，服务端的接收能力正常。
第二次握手：服务端收到 SYN 报文之后，会应答一个 SYN+ACK 报文。服务端发包，客户端收到了。客户端得出结论：服务端的接收和发送能力，客户端的接收和发送能力正常。但是此时服务端不能确认客户端的接收能力是否正常。
第三次握手;客户端收到 SYN+ACK 报文之后，回应一个 ACK 报文。客户端发包，服务端收到了。服务器得出结论：客户端的接收和发送能力，自己的接收发送能力都正常。
通过三次握手，双方都确认对方的接收以及发送能力正常。

1.4 HTTP 中重定向和请求转发的区别?

实现
转发：用 request 的 getRequestDispatcher()方法得到 ReuqestDispatcher 对象，调用 forward（）方法

request.getRequestDispatcher("other.jsp").forward(request, response);

重定向：调用 response 的 sendRedirect()方法

response.sendRedirect("other.jsp");

1> 重定向 2 次请求，请求转发 1 次请求

2> 重定向地址栏会变，请求转发地址栏不变

3> 重定向是浏览器跳转，请求转发是服务器跳转

4> 重定向可以跳转到任意网址，请求转发只能跳转当前项目

5>请求转发不会丢失请求数据，重定向会丢失

1.5 Get 和 Post 的区别?

1. Get 是不安全的，因为在传输过程，数据被放在请求的 URL 中；Post 的所有操作对用户来说都是不可见的。

2. Get 传送的数据量较小，一般传输数据大小不超过 2k-4k (根据浏览器不同，限制不一样，但相差不大这主要是因为受 URL 长度限制；Post 传送的数据量较大，一般被默认为不受限制。

3. Get 限制 Form 表单的数据集的值必须为 ASCII 字符；而 Post 支持整个 ISO10646 字符集。

4. Get 执行效率却比 Post 方法好。Get 是 form 提交的默认方法。

2.cookie 和 session 的区别？（必会）

1.存储位置不同

cookie 存放在客户端电脑，是一个磁盘文件。Ie 浏览器是可以从文件夹中找到。

session 是存放在服务器内存中的一个对象。 chrome 浏览器进行安全处理，只能通过浏览器找到。Session 是服务器端会话管理技术，并且 session 就是 cookie 实现的。

2.存储容量不同

单个 cookie 保存的数据<=4KB，一个站点最多保存 20 个 Cookie。

对于 session 来说并没有上限，但出于对服务器端的性能考虑，session 内不要存放过多的东 西，并且设置 session 删除机制。

3.存储方式不同

cookie 中只能保管 ASCII 字符串，并需要通过编码方式存储为 Unicode 字符或者二进制数据。

session 中能够存储任何类型的数据，包括且不限于 string，integer，list，map 等。

4.隐私策略不同

cookie 对客户端是可见的，别有用心的人可以分析存放在本地的 cookie 并进行 cookie 欺骗，所以它是不安全的。

session 存储在服务器上，不存在敏感信息泄漏的风险。

5. 有效期上不同

开发可以通过设置 cookie 的属性，达到使 cookie 长期有效的效果。

session 依赖于名为 JSESSIONID 的 cookie，而 cookie JSESSIONID 的过期时间默认为-1，只需关闭窗口该 session 就会失效，因而 session 不能达到长期有效的效果。

6.服务器压力不同

cookie 保管在客户端，不占用服务器资源。对于并发用户十分多的网站，cookie 是很好的选择。

session 是保管在服务器端的，每个用户都会产生一个 session。假如并发访问的用户十分多，会产生十分多的 session，耗费大量的内存。

3.Jsp 和 Servlet（了解）

1.Jsp 和 Servlet 的区别?

相同点

jsp 经编译后就变成了 servlet，jsp 本质就是 servlet，jvm 只能识别 java 的类，不能识别 jsp 代码，web 容器将 jsp 的代码编译成 jvm 能够识别的 java 类。其实就是当你通过 http 请求一个 JSP 页面是，首先 Tomcat 会调用 servlet 的 service（）方法将 JSP 编译成为 Servlet，然后执行 Servlet。

不同点

JSP 侧重视图，Sevlet 主要用于控制逻辑。

Servlet 中没有内置对象 。

JSP 中的内置对象都是必须通过 HttpServletRequest 对象，HttpServletResponse 对象以及 HttpServlet 对象得到。

2.Servlet 的生命周期

```
// 1. servlet 对象创建时，调用此方法
public void init(ServletConfig servletConfig);

// 2. 用户访问 servlet 时，调用此方法
public void service(ServletRequest servletRequest, ServletResponse servletResponse);

// 3. servlet 对象销毁时，调用此方法
public void destroy();
```

3.JSP 九大内置对象

out 对象：用于向客户端、浏览器输出数据。
request 对象：封装了来自客户端、浏览器的各种信息。
response 对象：封装了服务器的响应信息。
exception 对象：封装了 jsp 程序执行过程中发生的异常和错误信息。
config 对象：封装了应用程序的配置信息。
page 对象：指向了当前 jsp 程序本身。
session 对象：用来保存会话信息。也就是说，可以实现在同一用户的不同请求之间共享数
application 对象：代表了当前应用程序的上下文。可以在不同的用户之间共享信息。
pageContext 对象：提供了对 jsp 页面所有对象以及命名空间的访问。

4.Ajax 和 axios 的介绍（必会）

Ajax 即"Asynchronous JavaScript And XML"（异步 JavaScript 和 XML），是指一种创建交互式、快速动态网页应用的网页开发技术，无需重新加载整个网页的情况下，能够更新部分网页的技术。
Ajax 应用程序的优势在于：
1. 通过异步模式，提升了用户体验
2. 优化了浏览器和服务器之间的传输，减少不必要的数据往返，减少了带宽占用
3. Ajax 引擎在客户端运行，承担了一部分本来由服务器承担的工作，从而减少了大用户量下的服务器负载。

axios 是什么？怎样使用它？怎么解决跨域的问题？
axios 的是一种异步请求，用法和 ajax 类似，安装 npm install axios --save 即可使用，请求中包括 get,post,put, patch ,delete 等五种请求方式.

```
<script>
    // 引入 axios
    import Axios from 'axios'

    export default {
        methods: {
            testAxios() {
                const url = 'https://www.baidu.com/'

                Axios.get(url).then(response => {
                    if (response.data) {
```

```
                        console.log(response.data)
                    }
                }).catch(err => {
                    alert('请求失败')
                })
            }
        }
    }
</script>
```

axios 是一种异步请求方式，有 cdn 引入和 npm 方法引入并使用
解决跨域常用的有两种方式
1.CORS 解决跨域问题，这需要通过后端来解决，通过设置 header 头来通配。使服务器允许跨域请求接口数据，而前端正常使用 axios 请求方式。
2.通过接口代理的方式，在 vue 项目中创建一个 vue.config.js，导入一个 devserve，并配置里面的选项即可。

二、数据库

1.连接查询（必会）

1.左连接
（左外连接）以左表为基准进行查询,左表数据会全部显示出来,右表 如果和左表匹配 的数据则显示相应字段的数据,如果不匹配,则显示为 NULL;
2.右连接
（右外连接)以右表为基准进行查询,右表数据会全部显示出来,右表 如果和左表匹配的数据则显示相应字段的数据,如果不匹配,则显示为 NULL;

2.聚合函数（必会）

1.聚合函数

SQL 中提供的聚合函数可以用来统计、求和、求最值等等。

2.分类

COUNT：统计行数量

SUM：获取单个列的合计值
AVG：获取某个列的平均值
MAX：获取列的最大值
MIN：获取列的最小值

3.SQL 关键字(必会)

1.分页

MySQL 的分页关键词 limit

SELECT * FROM student3 LIMIT 2,6; 查询学生表中数据，从第三条开始显示，显示 6 条

2.分组

MySQL 的分组关键字：group by

SELECT sex, count(*) FROM student3 GROUP BY sex;

3. 去重

去重关键字：distinct

select DISTINCT NAME FROM student3;

4. SQL Select 语句完整的执行顺序: (必会)

查询中用到的关键词主要包含如下展示，并且他们的顺序依次为 **form...on...left join...where...group by...avg()/sum()...having..select...**

order by...asc/desc...limit...

from: 需要从哪个数据表检索数据

where: 过滤表中数据的条件

group by: 如何将上面过滤出的数据分组算结果

order by : 按照什么样的顺序来查看返回的数据

5. 数据库三范式(必会)

第一范式：1NF 原子性，列或者字段不能再分，要求属性具有原子性，不可再分解；

第二范式：2NF 唯一性，一张表只说一件事，是对记录的惟一性约束，要求记录有惟一标识，

第三范式：3NF 直接性，数据不能存在传递关系，即每个属性都跟主键有直接关系，而不是间接关系。

6. 存储引擎 （高薪常问）

1.MyISAM 存储引擎

主要特点：
MySQL5.5 版本之前的默认存储引擎
支持表级锁（表级锁是 MySQL 中锁定粒度最大的一种锁，表示对当前操作的整张表加锁）；
不支持事务，外键。
适用场景：对事务的完整性没有要求，或以 select、insert 为主的应用基本都可以选用 MYISAM。在 Web、数据仓库中应用广泛。
特点：
1、不支持事务、外键
2、每个 myisam 在磁盘上存储为 3 个文件，文件名和表名相同，扩展名分别是
.frm -------存储表定义
.MYD --------MYData，存储数据
.MYI --------MYIndex，存储索引

2.InnoDB 存储引擎

主要特点：
MySQL5.5 版本之后的默认存储引擎；
支持事务；
支持行级锁（行级锁是 Mysql 中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁）；
支持聚集索引方式存储数据。

7.数据库事务（必会）

1.事务特性

原子性：即不可分割性，事务要么全部被执行，要么就全部不被执行。
一致性：事务的执行使得数据库从一种正确状态转换成另一种正确状态
隔离性：在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何其他事务，
持久性：事务正确提交后，其结果将永久保存在数据库中，即使在事务提交后有了其他故障，事务的处理结果也会得到保存。

2.隔离级别

（1）读未提交（read Uncommited）：
在该隔离级别，所有的事务都可以读取到别的事务中未提交的数据，会产生脏读问题，在项目中基本不怎么用， 安全性太差；
(2) 读已提交（read commited）：
这是大多数数据库默认的隔离级别，但是不是 MySQL 的默认隔离级别；这个隔离级别满足了简单的隔离要求：一个事务只能看见已经提交事务所做的改变，所以会避免脏读问题；由于一个事务可以看到别的事务已经提交的数据，于是随之而来产生了不可重复读和虚读等问题（下面详细介绍这种问题，结合问题来理解隔离级别的含义）；
(3 ）可重复读（Repeatable read）：
这是 MySQL 的默认隔离级别，它确保了一个事务中多个实例在并发读取数据的时候会读取到一样的数据；不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB 和 Falcon 存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。
(4) 可串行化（serializable）：
事物的最高级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争，一般为了提升程序的吞吐量不会采用这个；

8.索引

1. 索引的概念和优点（了解）

概念：

索引存储在内存中，为服务器存储引擎为了快速找到记录的一种数据结构。索引的主要作用是加快数据查找速度，提高数据库的性能。

优点：

(1) 创建唯一性索引，保证数据库表中每一行数据的唯一性
(2) 大大加快数据的检索速度，这也是创建索引的最主要的原因
(3) 加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
(4) 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

2. 索引的分类（必会）

(1) 普通索引：最基本的索引，它没有任何限制。

(2) 唯一索引：与普通索引类似，不同的就是索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

(3) 主键索引：它是一种特殊的唯一索引，用于唯一标识数据表中的某一条记录，不允许有空值，一般用 primary key 来约束。

(4) 联合索引（又叫复合索引）：多个字段上建立的索引，能够加速复合查询条件的检索。

(5) 全文索引：老版本 MySQL 自带的全文索引只能用于数据库引擎为 MyISAM 的数据表，新版本 MySQL 5.6 的 InnoDB 支持全文索引。默认 MySQL 不支持中文全文检索，可以通过扩展 MySQL，添加中文全文检索或为中文内容表提供一个对应的英文索引表的方式来支持中文。

3. 索引的底层实现原理（高薪常问）

1. 索引结构

索引是在 Mysql 的存储引擎(InnoDB,MyISAM)层中实现的，而不是在服务层实现的. 所以每种存储引擎的索引都不一定完全相同，也不是所有的存储引擎都支持所有的索引类型的, Mysql 目前提供了以下 4 种索引:

B+Tree 索引: 最常见的索引类型，大部分索引都支持 B+树索引.

Hash 索引: 只有 Memory 引擎支持，使用场景简单.

R-Tree 索引(空间索引): 空间索引是 MyISAM 引擎的一个特殊索引类型，主要地理空间数据，使用也很少.

S-Full-text(全文索引): 全文索引也是 MyISAM 的一个特殊索引类型，主要用于全文索引, InnoDB 从 Mysql5.6 版本开始支持全文索引.

MyISAM、InnoDB、Memory三种存储引擎对各种索引类型的支持

索引	InnoDB引擎	MyISAM引擎	Memory引擎
B+TREE索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-tree 索引	不支持	支持	不支持
Full-text	5.6版本之后支持	支持	不支持

2. BTree 结构

B+Tree 是在 BTree 基础上进行演变的，所以我们先来看看 BTree, BTree 又叫多路平衡搜索树，一颗 m 叉 BTree 特性如下:

(1) 树中每个节点最多包含 m 个孩子.

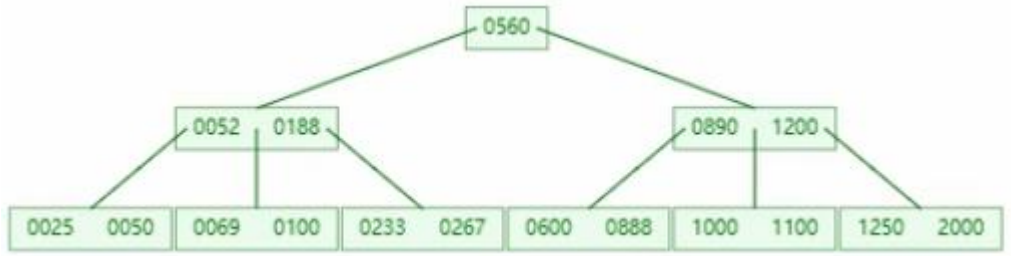
(2) 除根节点与叶子节点外，每个节点至少有[ceil(m/2)] 个孩子(ceil 函数指向上取整).

(3) 若根节点不是叶子节点，则至少有两个孩子.

(4) 每个非叶子节点由 n 个 Key 和 n+1 个指针组成，其中 [ceil(m/2) -1] <= n <= m-1.

以 5 叉 BTree 为例, key 的数量: 公式推导 [ceil(m/2) -1] <= n <= m-1.

所以 2 <= n <= 4，中间节点分裂父节点,两边节点分裂.



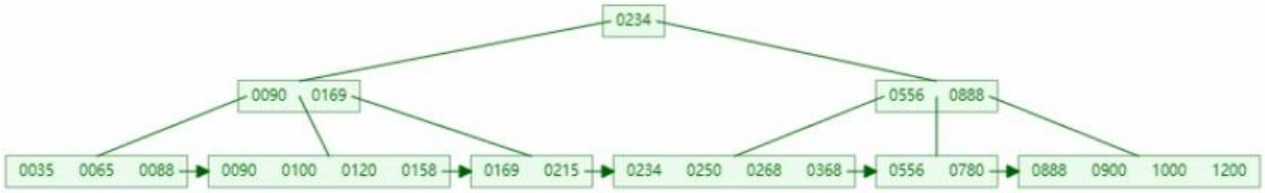
3.B+Tree 结构

B+Tree 为 BTree 的变种, B+Tree 与 BTree 的区别:

1.B+Tree 的叶子节点保存所有的 key 信息，依 key 大小顺序排列.

2.B+Tree 叶子节点元素维护了一个单项链表.

所有的非叶子节点都可以看作是 key 的索引部分.

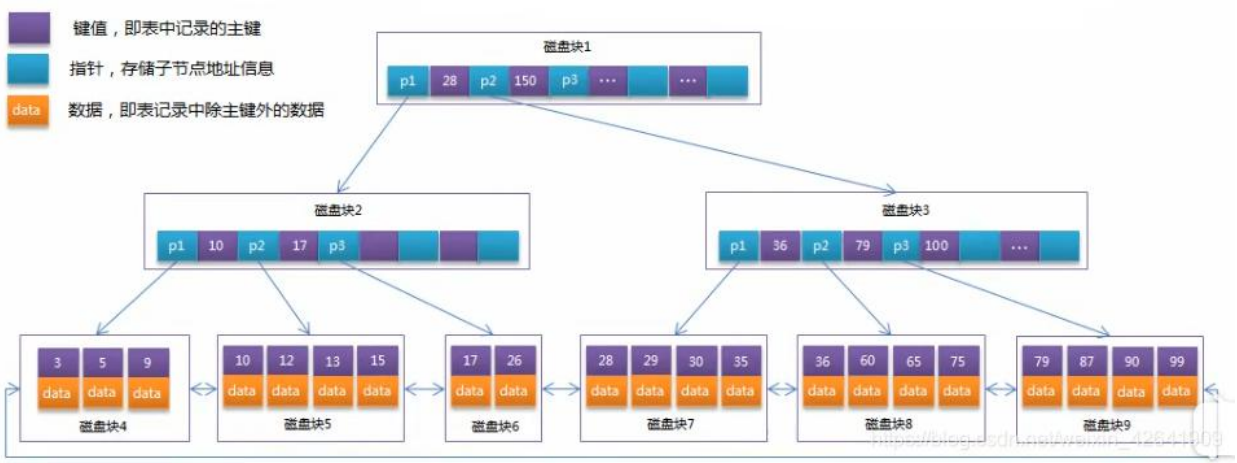


由于 B+Tree 只有叶子节点保存 key 信息，查询任何 key 都要从 root 走的叶子. 所以 B+Tree 查询效率更稳定.

Mysql 中的 B+Tree

MySql 索引数据结构对经典的 B+Tree 进行了优化，在原 B+Tree 的基础上，增加了一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的 B+Tree，提高区间访问的性能.

MySql 中的 B+Tree 索引结构示意图:



4. 如何避免索引失效（高薪常问）

(1) 范围查询，右边的列不能使用索引，否则右边的索引也会失效.

索引生效案例

```
select * from tb_seller where name = "小米科技" and status = "1" and address = "北京市";
```

```
select * from tb_seller where name = "小米科技" and status >= "1" and address = "北京市";
```

索引失效案例

```
select * from tb_seller where name = "小米科技" and status > "1" and address = "北京市";
```

address 索引失效，因为 status 是大于号，范围查询.

(2) 不要在索引上使用运算，否则索引也会失效.

比如在索引上使用切割函数，就会使索引失效.

```
select * from tb_seller where substring(name, 3, 2) = "科技";
```

(3) 字符串不加引号，造成索引失效.

如果索引列是字符串类型的整数，条件查询的时候不加引号会造成索引失效. Mysql 内置的优化会有隐式转换.

索引失效案例

```
select * from tb_seller where name = "小米科技" and status = 1
```

(4) 尽量使用覆盖索引，避免 select *，这样能提高查询效率.

如果索引列完全包含查询列，那么查询的时候把要查的列写出来，不使用 select *

```
select sellerid, name, status from tb_seller where name = "小米科技" and staus = "1"
and address = "西安市";
```

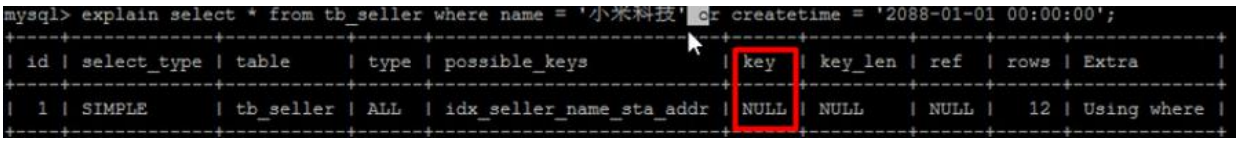
(5) or 关键字连接

用 or 分割开的条件，如果 or 前面的列有索引, or 后面的列没有索引，那么查询的时候前后索引都会失效

如果一定要用 or 查询，可以考虑下 or 连接的条件列都加索引，这样就不会失效了.

- **索引失效案例:**

```
select * from tb_seller where name = "小米科技" or createTiem = "2018-01-01
00:00:00";
```



9.数据库锁(高薪常问)

1.行锁和表锁

1.主要是针对锁粒度划分的，一般分为：行锁、表锁、库锁
行锁：访问数据库的时候，锁定整个行数据，防止并发错误。
表锁：访问数据库的时候，锁定整个表数据，防止并发错误。

2.行锁 和 表锁 的区别：
表锁： 开销小，加锁快，不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低
行锁： 开销大，加锁慢，会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高

2.悲观锁和乐观锁

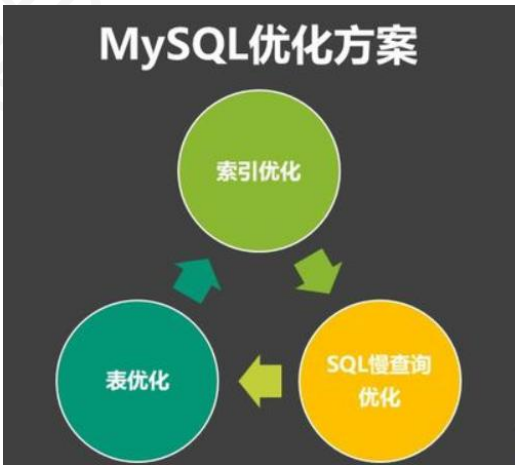
（1）悲观锁：顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 block 直到它拿到锁。

传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

（2）乐观锁： 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 write_condition 机制的其实都是提供的乐观锁。

10.MySql 优化(高薪常问)



1) 定位执行效率慢的 sql 语句.(了解)

- 命令:show status like 'Com____',通过这条命令，我们可以知道当前数据库是以查询为主还是更新为主. 如果是查询为主，就重点查询; 如果增删改多就重点优化写入操作.
- explain + sql语句查询sql执行过程，通过执行计划，我们能得到哪些信息：

  A：哪些步骤花费的成本比较高

  B：哪些步骤产生的数据量多，数据量的多少用线条的粗细表示，很直观

  C：这条sql语句是否走索引
- show profile 分析 SQL,可以查看所有 sql 语句的执行效率(所用时间). 前提是这个命令需要被打开，严格的说也就是打开这个命令后执行的所有 sql 语句，它都能记录下执行时间，并展示出来. 可以通过这个命令分析哪些 sql 语句执行效率低. 耗时长，就更有针对性的优化这条 sql.
- 慢查询日志(常用的工具)

慢查询日志记录了所有执行时间超过参数 long_query_time 的 sql 语句的日志，long_query_time 默认为 10 秒(可以通过配置文件设置)，日志保存在 /var/lib/mysql/目录下，有个 slow_query.log 文件,

2) 优化索引(高薪)

2.1 索引设计原则

索引的设计需要遵循一些已有的原则，这样便于提升索引的使用效率，更高效的使用索引.

- 对查询频次较高，且数据量比较大的表，建立索引.
- 索引字段的选择，最佳候选列应当从 where 子句的条件中提取，如果 where 子句中的组合比较多，那么应当挑选最常用，过滤效果最好的列的组合.
- 使用唯一索引，区分度越高，使用索引的效率越高.
- 索引并非越多越好，如果该表赠,删,改操作较多，慎重选择建立索引， 过多索引会降低表维护效率.
- 使用短索引，提高索引访问时的 I/O 效率，因此也相应提升了 Mysql 查询效率.
- 如果 where 后有多个条件经常被用到，建议建立符合 索引，复合索引需要遵循最左前缀法则，N 个列组合而成的复合索引，相当于创建了 N 个索引.

  复合索引命名规则 index_表名_列名 1_列名 2_列明 3

  比如:create index idx_seller_name_sta_addr on tb_seller(name, status, address)

2.2 避免索引失效

- 如果在查询的时候，使用了复合索引，要遵循最左前缀法则，也就是查询从索引的最左列开始，并且不能跳过索引中的列.
- 尽量不要在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进

行全表扫描

- 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

- 不做列运算where age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数.计算表达式等，都会是索引失效.
- 查询 like，如果是 '%aaa' 也会造成索引失效.
- 应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描

3) Sql 语句调优(高薪)

- 根据业务场景建立复合索引只查询业务需要的字段，如果这些字段被索引覆盖，将极大的提高查询效率.
- 多表连接的字段上需要建立索引，这样可以极大提高表连接的效率.
- where 条件字段上需要建立索引，但 Where 条件上不要使用运算函数，以免索引失效.
- 排序字段上，因为排序效率低，添加索引能提高查询效率.
- 优化 insert 语句：批量列插入数据要比单个列插入数据效率高.
- 优化 order by 语句：在使用 order by 语句时，不要使用 select *, select 后面要查有索引的列，如果一条 sql 语句中对多个列进行排序，在业务允许情况下，尽量同时用升序或同时用降序.
- 优化 group by 语句：在我们对某一个字段进行分组的时候, Mysql 默认就进行了排序，但是排序并不是我们业务所需的，额外的排序会降低效率. 所以在用的时候可以禁止排序，使用 order by null 禁用.

```
select age, count(*) from emp group by age order by null
```

- 尽量避免子查询，可以将子查询优化为 join 多表连接查询.

4) 合理的数据库设计(了解)

根据数据库三范式来进行表结构的设计。设计表结构时，就需要考虑如何设计才能更有效的查询，遵循**数据库三范式：**

i. **第一范式**：数据表中每个字段都必须是不可拆分的最小单元，也就是确保每一列的原子性；

ii. **第二范式**：满足一范式后，表中每一列必须有唯一性，都必须依赖于主键；

iii. **第三范式**：满足二范式后，表中的每一列只与主键直接相关而不是间接相关(外键也是直接相关)，字段没有冗余。

注意：没有最好的设计，只有最合适的设计，所以不要过分注重理论。三范式可以作为一个基本依据，不要生搬硬套。有时候可以根据场景合理地反规范化：

A：保留冗余字段。当两个或多个表在查询中经常需要连接时，可以在其中一个表上增加若干冗余的字段，以 避免表之间的连接过于频繁，一般在冗余列的数据不经常变动的情况下使用。

B：增加派生列。派生列是由表中的其它多个列的计算所得，增加派生列可以减少统计运算，在数据汇总时可以大大缩短运算时间，前提是这个列经常被用到，这也就是反第三范式。

C：分割表。

数据表拆分：主要就是垂直拆分和水平拆分。

水平切分:将记录散列到不同的表中，各表的结构完全相同，每次从分表中查询，提高效率。

垂直切分:将表中大字段单独拆分到另外一张表，形成一对一的关系。

D: 字段设计

1. 表的字段尽可能用 NOT NULL
2. 字段长度固定的表查询会更快
3. 把数据库的大表按时间或一些标志分成小表

四. 框架

1. Mybatis 框架

1.1 谈一谈你对 Mybatis 框架的理解(了解)

MyBatis 是一款优秀的持久层框架，一个半 ORM（对象关系映射）框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

1.2 在 mybatis 中,${} 和 #{} 的区别是什么?(必会)

#{} 是占位符，预编译处理，${}是字符串替换。

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

Mybatis 在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止 SQL 注入，提高系统安全性。

1.3 MyBatis 编程步骤是什么样的？(了解)

1、 创建 SqlSessionFactory
2、 通过 SqlSessionFactory 创建 SqlSession
3、 通过 sqlsession 执行数据库操作
4、 调用 session.commit()提交事务
5、 调用 session.close()关闭会话

1.4 在 mybatis 中,resultType 和 ResultMap 的区别是什么?(必会)

如果数据库结果集中的列名和要封装实体的属性名完全一致的话用 resultType 属性

如果数据库结果集中的列名和要封装实体的属性名有不一致的情况用 resultMap 属性，通过 resultMap 手动建立对象关系映射，resultMap 要配置一下表和类的一一对应关系，所以说就算你的字段名和你的实体类的属性名不一样也没关系，都会给你映射出来

1.5 在 Mybatis 中你知道的动态 SQL 的标签有哪些?作用分别是什么?(必会)

1. <if>if 是为了判断传入的值是否符合某种规则,比如是否不为空.

2. <where> where 标签可以用来做动态拼接查询条件,当和 if 标签配合的时候,不用显示的声明类型 where 1 = 1 这种无用的条件

3. <foreach> foreach 标签可以把传入的集合对象进行遍历,然后把每一项的内容作为参数传到 sql 语句中.

4. <include> include 可以把大量的重复代码整理起来,当使用的时候直接 include 即可,减少重复代码的编写;

5. <set>适用于更新中,当匹配某个条件后,才会对该字段进行跟新操作

1.6 谈一下你对 mybatis 缓存机制的理解?(了解)

Mybatis 有两级缓存，一级缓存是 SqlSession 级别的，默认开启，无法关闭；二级缓存是 Mapper 级别的，二级缓存默认是没有开启的，但是手动开启

1. 一级缓存:基础 PerpetualCache 的 HashMap 本地缓存,其存储作用域为 Session,当 Session flush 或 close 之后，Session 中的所有 Cache 就将清空

2. 二级缓存其存储作用域为 Mapper(Namespace），使用二级缓存属性类需要实现 Serializable 序列化接口

3. 对于缓存数据更新机制,当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C(增加)/U(更新)/D(删除)操作后,默认该作用域下所有 select 中的缓存将被 clear.

需要在 setting 全局参数中配置开启二级缓存，如下 conf.xml 配置:

```
<settings>
        <setting name="cacheEnabled" value="true"/>默认是false：关闭二级缓存
<settings>
```

当我们的配置文件配置了 cacheEnabled=true 时，就会开启二级缓存，二级缓存是 mapper 级别的，如果你配置了二级缓存，那么查询数据的顺序应该为：二级缓存→一级缓存→数据库。

2. Spring 框架

2.1 Spring 的两大核心是什么?谈一谈你对 IOC 的理解？谈一谈你对 DI 的理解?谈一谈你对 AOP 的理解?(必会)

1. Spring 的两大核心是：IOC（控制反转）和 AOP（面向切面编程） DI（依赖注入）

2. IOC 的意思是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到 Spring 容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。最直观的表达就是，IOC 让对象的创建不用去 new 了，可以由 spring 根据我们提供的配置文件自动生产，我们需要对象的时候，直接从 Spring 容器中获取即可.

Spring 的配置文件中配置了类的字节码位置及信息，容器生成的时候加载配置文件识别字节码信息，通过反射创建类的对象.

Spring 的 IOC 有三种注入方式 ： 构造器注入, setter 方法注入，根据注解注入。

3. DI 的意思是依赖注入，和控制反转是同一个概念的不同角度的描述，即应用程序在运行时依赖 Io c 容器来动态注入对象需要的外部资源。

4. AOP，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect）. SpringAOP 使用的动态代理，所谓的动态代理就是说 AOP 框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个 AOP 对象，这个 AOP 对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

5. Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理：

(1)JDK 动态代理只提供接口代理，不支持类代理，核心 InvocationHandler 接口和 Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起，Proxy 利用 InvocationHandler 动态创建一个符合某一接口的的实例，生成目标类的代理对象。

(2) 如果代理类没有实现 InvocationHandler 接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

2.2 Spring 的生命周期?(高薪常问)

1. 实例化一个 Bean，也就是我们通常说的 new
2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入
3. 如果这个 Bean 实现 dao 了 BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法，此处传递的是 Spring 配置文件中 Bean 的 ID
4. 如果这个 Bean 实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory()，传递的是 Spring 工厂本身（可以用这个方法获取到其他 Bean）
5. 如果这个 Bean 实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文，该方式同样可以实现步骤 4，但比 4 更好，以为 ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法
6. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用 After 方法，也可用于内存或缓存技术
7. 如果这个 Bean 在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法
8. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个 Bean 了，那这个 Bean 是一个 single 的，所以一般情况下我们调用同一个 ID 的 Bean 会是在内容地址相同的实例

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 接口，会调用其实现的 destroy 方法
10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法

2.3 Spring 支持 bean 的作用域有几种吗？每种作用域是什么样的?(必会)

Spring 支持如下 5 种作用域：

（1）singleton：默认作用域，单例 bean，每个容器中只有一个 bean 的实例。

（2）prototype：每次请求都会为 bean 创建实例。

（3）request：为每一个 request 请求创建一个实例，在请求完成以后，bean 会失效并被垃圾回收器回收。

（4）session：与 request 范围类似，同一个 session 会话共享一个实例，不同会话使用不同的实例。

（5）global-session：全局作用域，所有会话共享一个实例。如果想要声明让所有会话共享的存储变量的话，那么这全局变量需要存储在 global-session 中。

2.4 BeanFactory 和 ApplicationContext 有什么区别(了解)

BeanFactory：

Spring 最顶层的接口,实现了 Spring 容器的最基础的一些功能，调用起来比较麻烦，一般面向 Spring 自身使用

BeanFactory 在启动的时候不会去实例化 Bean，从容器中拿 Bean 的时候才会去实例化

ApplicationContext：

是 BeanFactory 的子接口,扩展了其功能，一般面向程序员身使用

ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了

2.5 Spring 框架中都用到了哪些设计模式?(必会)

1. 工厂模式：BeanFactory 就是简单工厂模式的体现，用来创建对象的实例
2. 单例模式：Bean 默认为单例模式
3. 代理模式：Spring 的 AOP 功能用到了 JDK 的动态代理和 CGLIB 字节码生成技术
4. 模板方法：用来解决代码重复的问题。比如．RestTemplate, JmsTemplate, JpaTemplate
5. 观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如 Spring 中 listener 的实现

--ApplicationListener

2.6 Spring 事务的实现方式和实现原理(必会)

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring 是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。

spring 事务实现主要有两种方法

1、编程式，beginTransaction()、commit()、rollback()等事务管理相关的方法

2、声明式，利用注解 Transactional 或者 aop 配置

2.6 你知道的 Spring 的通知类型有哪些,分别在什么时候执行?(了解)

Spring 的通知类型有四种，分别为：

前置通知[]before]：在切点运行之前执行

后置通知[after-returning]：在切点正常结束之后执行

异常通知[after-throwing]：在切点发生异常的时候执行

最终通知[after]：在切点的最终执行

Spring 还有一种特殊的通知,叫做环绕通知

环绕通知运行程序员以编码的方式自己定义通知的位置，用于解决其他通知时序问题

2.7 Spring 的对象默认是单例的还是多例的？单例 bean 存不存在线程安全问题呢?(必会)

1. 在 spring 中的对象默认是单例的，但是也可以配置为多例。

2. 单例 bean 对象对应的类存在可变的成员变量并且其中存在改变这个变量的线程时，多线程操作该 bean 对象时会出现线程安全问题。

原因是：多线程操作如果改变成员变量，其他线程无法访问该 bean 对象，造成数据混乱。

解决办法：在 bean 对象中避免定义可变成员变量；

在 bean 对象中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中。

2.8 @Resource 和@Autowired 依赖注入的区别是什么? @Qualifier 使用场景是什么?(了解)

@**Resource**

只能放在属性上，表示先按照属性名匹配 IOC 容器中对象 id 给属性注入值若没有成功，会继续根据当前属性的类型匹配 IOC 容器中同类型对象来注入值

若指定了 name 属性@Resource(name = "对象 id"),则只能按照对象 id 注入值。

@**Autowird**

放在属性上：表示先按照类型给属性注入值如果 IOC 容器中存在多个与属性同类型的对象，则会按照属性名注入值

也可以配合@Qualifier("IOC 容器中对象 id")注解直接按照名称注入值。

放在方法上：表示自动执行当前方法，如果方法有参数，会自动从 IOC 容器中寻找同类型的对象给参数传值

也可以在参数上添加@Qualifier("IOC 容器中对象 id")注解按照名称寻找对象给参数传值。

@**Qualifier** 使用场景：

@Qualifier("IOC 容器中对象 id")可以配合@Autowird 一起使用，表示根据指定的 id 在 Spring 容器中匹配对象

2.8 Spring 的常用注解(必会)

1. @Component(任何层) @Controller @Service @Repository（dao）：用于实例化对象

2. @Scope：设置 Spring 对象的作用域

3. @PostConstruct @PreDestroy：用于设置 Spring 创建对象在对象创建之后和销毁之前要执行的方法

4. @Value: 简单属性的依赖注入

5. @Autowired: 对象属性的依赖注入

6. @Qualifier: 要和@Autowired 联合使用，代表在按照类型匹配的基础上，再按照名称匹配。

7. @Resource 按照属性名称依赖注入

8. @ComponentScan: 组件扫描

9. @Bean: 表在方法上,用于将方法的返回值对象放入容器

10. @PropertySource: 用于引入其它的 properties 配置文件

11. @Import: 在一个配置类中导入其它配置类的内容

12. @Configuration: 被此注解标注的类,会被 Spring 认为是配置类。Spring 在启动的时候会自动扫描并加载所有配置类，然后将配置 类中 bean 放入容器

13. @Transactional 此注解可以标在类上，也可以表在方法上，表示当前类中的方法具有事务管理功能。

2.9 Spring 的事务传播行为(高薪常问)

spring 事务的传播行为说的是，当多个事务同时存在的时候，spring 如何处理这些事务的行为。

备注(方便记忆): propagation 传播

require 必须的/suppor 支持/mandatory 强制托管/requires-new 需要新建/not -supported 不支持/never 从不/nested 嵌套的

① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按 REQUIRED 属性执行。

2.10 Spring 中的隔离级别（高薪常问)

ISOLATION 隔离的意思

① ISOLATION_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION_READ_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION_READ_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。解决脏读问题

④ ISOLATION_REPEATABLE_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。行锁

⑤ ISOLATION_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。表锁

3.SpringMVC 框架

3.1 谈一下你对 SpringMVC 框架的理解(了解)

SpringMVC 是一个基于 Java 的实现了 MVC 设计模式的请求驱动类型的轻量级 Web 框架，通过把 Model，View，Controller 分离，将 web 层进行职责解耦，把复杂的 web 应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

在我看来,SpringMVC 就是将我们原来开发在 servlet 中的代码拆分了,一部分由 SpringMVC 完成,一部分由我们自己完成

3.2 SpringMVC 主要组件(必会)

前端控制器 DispatcherServlet：接收请求、响应结果，相当于转发器，有了 DispatcherServlet 就减少了其它组件之间的耦合度。

处理器映射器 HandlerMapping：根据请求的 URL 来查找 Handler

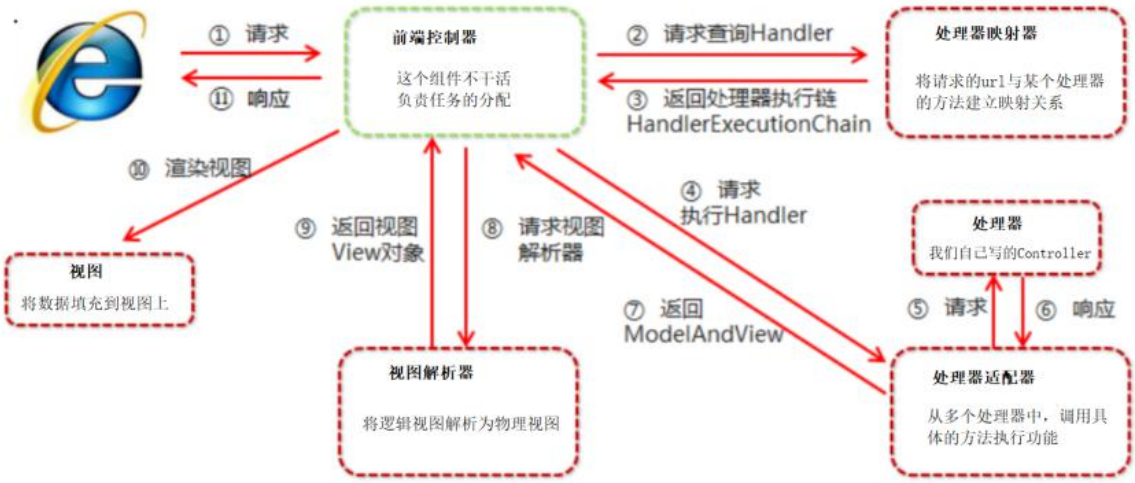
处理器适配器 HandlerAdapter：负责执行 Handler

处理器 Handler：处理业务逻辑的 Java 类

视图解析器 ViewResolver：进行视图的解析，根据视图逻辑名将 ModelAndView 解析成真正的视图（view）

视图 View：View 是一个接口， 它的实现类支持不同的视图类型，如 jsp，freemarker，pdf 等等

3.3 谈一下 SpringMVC 的执行流程以及各个组件的作用(必会)



1. 用户发送请求到前端控制器（DispatcherServlet）

2. 前端控制器（DispatcherServlet）收到请求调用处理器映射器（HandlerMapping），去查找处理器（Handler）

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

4. 前端控制器（DispatcherServlet）调用处理器映射器（HandlerMapping）

5. 处理器适配器（HandlerAdapter）去调用自定义的处理器类(Controller，也叫后端控制器)。

6.自定义的处理器类(Controller，也叫后端控制器)将得到的参数进行处理并返回结果给处理器映射器（HandlerMapping）

7. 处理器适配器（HandlerAdapter）将得到的结果返回给前端控制器（DispatcherServlet）

8. DispatcherServlet(前端控制器)将 ModelAndView 传给视图解析器(ViewReslover)

9. 视图解析器(ViewReslover)将得到的参数从逻辑视图转换为物理视图并返回给前端控制器（DispatcherServlet）

10. 前端控制器（DispatcherServlet）调用物理视图进行渲染并返回

11. 前端控制器（DispatcherServlet）将渲染后的结果返回

3.4 说一下 SpringMVC 支持的转发和重定向的写法(必会)

1）转发：

forward 方式:在返回值前面加"forward:",比如"" "forward:user.do?name=method4"

2) 重定向:

redirect 方式：在返回值前面加 redirect:，比如"redirect:http://www.baidu.com"

3.5 SpringMVC 的常用注解(必会)

1.@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

2.@RequestBody：注解实现接收 http 请求的 json 数据，将 json 转换为 java 对象。

3.@ResponseBody：注解实现将 conreoller 方法返回对象转化为 json 对象响应给客户。

4.@PathVariable 用户从 url 路径上获取指定参数，标注在参数前 @PathVariabel("要获取的参数名")。

5.@RequestParam: 标注在方法参数之前，用于对传入的参数做一些限制，支持三个属性:

- value：默认属性，用于指定前端传入的参数名称
- required：用于指定此参数是否必传
- defaultValue：当参数为非必传参数且前端没有传入参数时，指定一个默认值

6. @ControllerAdvice 标注在一个类上，表示该类是一个全局异常处理的类。

7. @ExceptionHandler(Exception.class) 标注在异常处理类中的方法上，表示该方法可以处理的异常类型。

3.6 谈一下 SpringMVC 统一异常处理的思想和实现方式(必会)

使用 SpringMVC 之后,代码的调用者是 SpringMVC 框架,也就是说最终的异常会抛到框架中,然后由框架指定异常处理类进行统一处理

方式一: 创建一个自定义异常处理器(实现 HandlerExceptionResolver 接口),并实现里面的异常处理方法,然后将这个类交给 Spring 容器管理

方式二: 在类上加注解(@ControllerAdvice)表明这是一个全局异常处理类

在方法上加注解(@ExceptionHandler),在 ExceptionHandler 中有一个 value 属性,可以指定可以处理的异常类型

3.7 在 SpringMVC 中，如果想通过转发将数据传递到前台,有几种写法?(必会)

方式一：直接使用 request 域进行数据的传递

```
request.setAttirbuate("name", value);
```

方式二：使用 Model 进行传值，底层会将数据放入 request 域进行数据的传递

```
model.addAttribuate("name", value);
```

方式三：使用 ModelMap 进行传值，底层会将数据放入 request 域进行数据的传递

```
modelmap.put("name",value);
```

方式四：借用 ModelAndView 在其中设置数据和视图

```
mv.addObject("name",value);
mv.setView("success");
return mv;
```

3.8 在 SpringMVC 中拦截器的使用步骤是什么样的?(必会)

1 定义拦截器类:

SpringMVC 为我们提供了拦截器规范的接口，创建一个类实现 HandlerInterceptor,重写接口中的抽象方法;

preHandle 方法：在调用处理器之前调用该方法，如果该方法返回 true 则请求继续向下进行，否则请求不会继续向下进行,处理器也不会调用

postHandle 方法：在调用完处理器后调用该方法

afterCompletion 方法：在前端控制器渲染页面完成之后调用此方法

2 注册拦截器:

在 SpringMVC 核心配置文件中注册自定义的拦截器

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="拦截路径规则"/>
        <mvc:exclude-mapping path="不拦截路径规则"/>
        <bean class="自定义拦截器的类全限定名"/>
    </mvc:interceptor>
</mvc:interceptors>
```

3.9 在 SpringMVC 中文件上传的使用步骤是什么样的？前台三要素是什么?(必会)

文件上传步骤:

1.加入文件上传需要的 commons-fileupload 包

2.配置文件上传解析器,springmvc 的配置文件的文件上传解析器的 id 属性必须为 multipartResolver

3.后端对应的接收文件的方法参数类型必须为 MultipartFile,参数名称必须与前端的 name 属性保持一致

文件上传前端三要素:

1.form 表单的提交方式必须为 post

2.enctype 属性需要修改为:multipart/form-data

3.必须有一个 type 属性为 file 的 input 标签,其中需要有一个 name 属性;如果需要上传多个文件需要添加 multiple 属性

3.10 SpringMVC 中如何解决 GET|POST 请求中文乱码问题？(了解)

（1）解决 post 请求乱码问题：在 web.xml 中配置一个 CharacterEncodingFilter 过滤器，设置成 utf-8；

```
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

(2) get 请求中文参数出现乱码解决方法有两个：

①修改 tomcat 配置文件添加编码与工程编码一致，如下：

```
    <ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" p
rotocol="HTTP/1.1" redirectPort="8443"/>
```

②另外一种方法对参数进行重新编码：

```
String userName= new String(request.getParamter("userName").getBytes("ISO885
9-1"),"utf-8")
```

ISO8859-1 是 tomcat 默认编码，需要将 tomcat 编码后的内容按 utf-8 编码。

4. Dubbo

4.1 什么是 dubbo(必会)

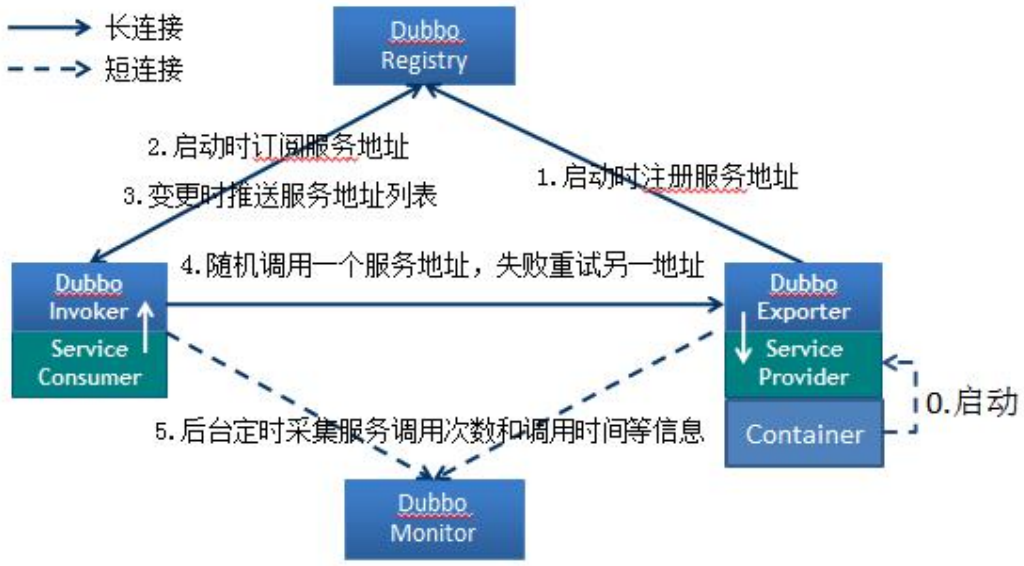
工作在 soa 面向服务分布式框架中的服务管理中间件。Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，以及 SOA 服务治理方案。

它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo 采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。关于注册中心、协议支持、服务监控等内容。

Dubbo 使用的是缺省协议， 采用长连接和 nio 异步通信， 适合小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。

反之， dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。

4.2 Dubbo 的实现原理(必会)



4.3 节点角色说明(必会)

Provider：暴露服务的服务提供方。

Consumer：调用远程服务的服务消费方。

Registry：服务注册与发现的注册中心。

Monitor：统计服务的调用次调和调用时间的监控中心。

Container：服务运行容器。

4.4 调用关系说明(必会)

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

4.5 在实际开发的场景中应该如何选择 RPC 框架(了解)

SpringCloud ： Spring 全家桶，用起来很舒服，只有你想不到，没有它做不到。可惜因为发布的比较晚，国内还没出现比较成功的案例，大部分都是试水，不过毕竟有 Spring 作背景，还是比较看好。

Dubbox：相对于 Dubbo 支持了 REST，估计是很多公司选择 Dubbox 的一个重要原因之一，但如果使用 Dubbo 的 RPC 调用方式，服务间仍然会存在 API 强依赖，各有利弊，懂的取舍吧。

Thrift： 如果你比较高冷，完全可以基于 Thrift 自己搞一套抽象的自定义框架吧。

Hessian：如果是初创公司或系统数量还没有超过 5 个，推荐选择这个，毕竟在开发速度.运维成本.上手难度等都是比较轻量.简单的，即使在以后迁移至 SOA，也是无缝迁移。

rpcx/gRPC：在服务没有出现严重性能的问题下，或技术栈没有变更的情况下，可能一直不会引入，即使引入也只是小部分模块优化使用。

5. Zookeeper

1. Zookeeper 是什么(了解)

Zookeeper 是一个**分布式协调服务**的开源框架，主要用来解决分布式集群中应用系统的一致性问题，例如怎样避免同时操作同一数据造成脏读的问题.

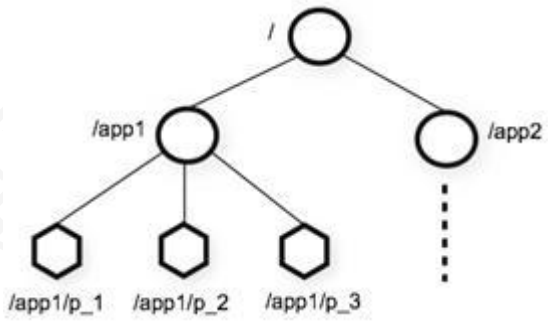
ZooKeeper 本质上是**一个分布式的小文件存储系统**. 提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理. 从而用来维护和监控你存储的数据的状态变化. 通过监控这些数据状态的变化，从而可以达到基于数据的集群管理.

在大数据生态系统里，很多组件的命名都是某种动物，比如 hadoop 就是大象, hive 就是蜜蜂，而 Zookeeper 就是动物管理员.

2. Zookeeper 的数据模型(必会)

- ZK 本质上是一个分布式的小文件存储系统.
- ZK 表现为一个分层的文件系统目录树结构，既能存储数据，而且还能像目录一样有子节点. 每个节点可以存最多 1M 左右的数据.
- 每个节点称做一个 Znode, 每个 Znode 都可以通过其路径唯一标识.
- 而且客户端还能给节点添加 watch，也就是监听器，可以监听节点的变化，这个功能常在实际开发中作为监听服务器集群机器上下线操作.

2.1 节点结构



图中的每个节点称为一个 Znode。 每个 Znode 由 3 部分组成:
① stat：此为状态信息，描述该 Znode 的版本，权限等信息
② data：与该 Znode 关联的数据
③ children：该 Znode 下的子节点

2.2 节点类型

Znode 有 2 大类 4 小类，两大类分别为永久节点和临时节点.

- 永久节点(Persistent)：客户端和服务器端断开连接后，创建的节点不会消失，只有在客户端执行删除操作的时候，他们才能被删除.
- 临时节点(Ephemeral)：客户端和服务器端断开连接后，创建的节点会被删除.

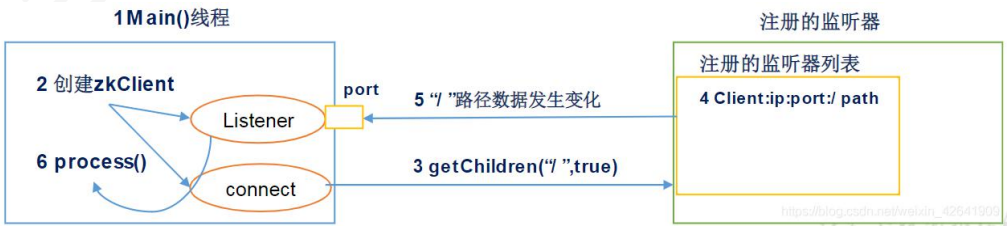
Znode 还有一个序列化的特性，这个序列号对于此节点的父节点来说是唯一的，这样便会记录每个子节点创建的先后顺序. 它的格式为“%10d”(10 位数字，没有数值的数位用 0 补充，例如“0000000001”)，因此节点可以分为 4 小类:

- ◆ 永久节点(Persistent)
- ◆ 永久_序列化节点(Persistent_Sequential)
- ◆ 临时节点(Ephemeral)
- ◆ 临时_序列化节点(Ephemeral_Sequential)

3. Zookeeper 的 watch 监听机制(高薪常问)

- 在 ZooKeeper 中还支持一种 watch(监听)机制，它允许对 ZooKeeper 注册监听，当监听的对象发生指定的事件的时候, ZooKeeper 就会返回一个通知.
- Watcher 分为以下三个过程：客户端向 ZK 服务端注册 Watcher、服务端事件发生触发 Watcher、客户端回调 Watcher 得到触发事件情况.
  触发事件种类很多，如：节点创建，节点删除，节点改变，子节点改变等。
- Watcher 是一次性的. 一旦被触发将会失效. 如果需要反复进行监听就需要反复进行注册.

3.1 监听器原理



- 首先要有一个 main()线程
- 在 main 线程中创建 Zookeeper 客户端， 这时就会创建两个线程， 一个复制网络连接通信(connect)，一个负责监听(listener).
- 通过 connect 线程将注册的监听事件发送给 zk，常见的监听有

监听节点数据的变化 `get path [watch]`

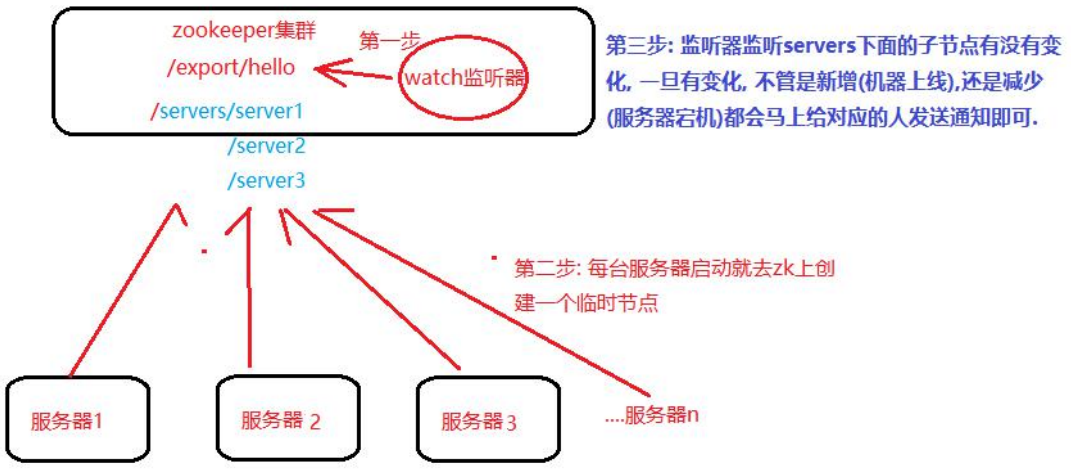
监听节点状态的变化 `stat path [watch]`

监听子节点增减的变化 `ls path [watch]`

- 将注册的监听事件添加到 zk 的注册的监听器列表中
- 监听到有数据或路径变化，就会将这个消息发送给 listener 线程.
- listener 线程内部调用了 process()方法.此方法是程序员自定义的方法，里面可以写明监听到事件后做如何的通知操作.

3.2 监听器实际应用

监听器+ZK 临时节点能够很好的监听服务器的上线和下线.



- 第一步: 先想 zk 集群注册一个监听器，监听某一个节点路径
- 第二步: 主要服务器启动，就去 zk 上指定路径下创建一个临时节点.
- 第三步: 监听器监听 servers 下面的子节点有没有变化，一旦有变化，不管新增(机器上线)还是减少(机器下线)都会马上给对应的人发送通知.

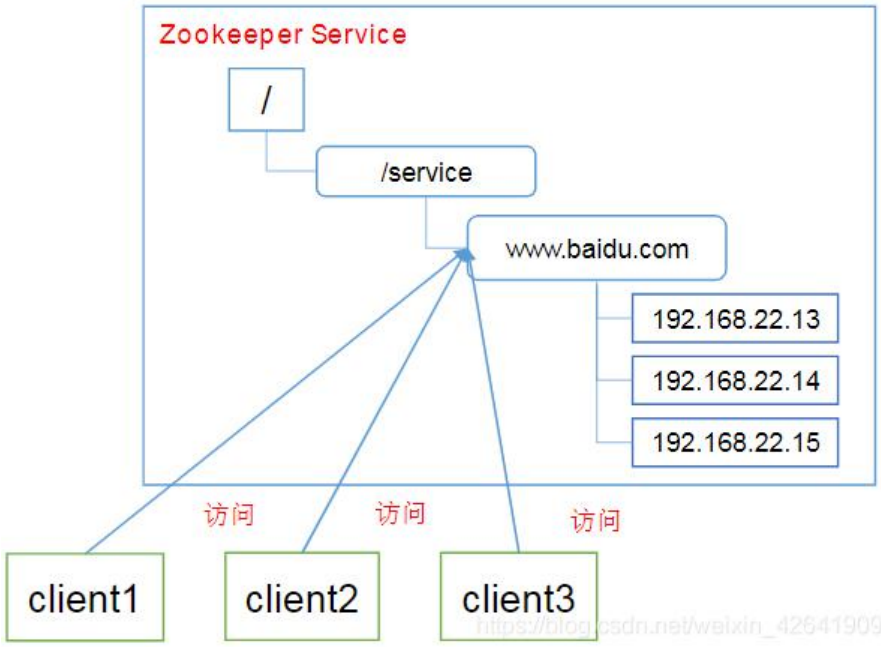
4. Zookeeper 的应用场景(高薪常问)

ZK 提供的服务包括: 统一命名服务，统一配置管理，统一集群管理，集群选主，服务动态上下线，分布式锁等.

4.1 统一命名服务

统一命名服务使用的是 ZK 的 node 节点全局唯一的这个特点.

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。例如：IP 不容易记住，而域名容易记住。创建一个节点后，节点的路径就是全局唯一的，可以作为全局名称使用.

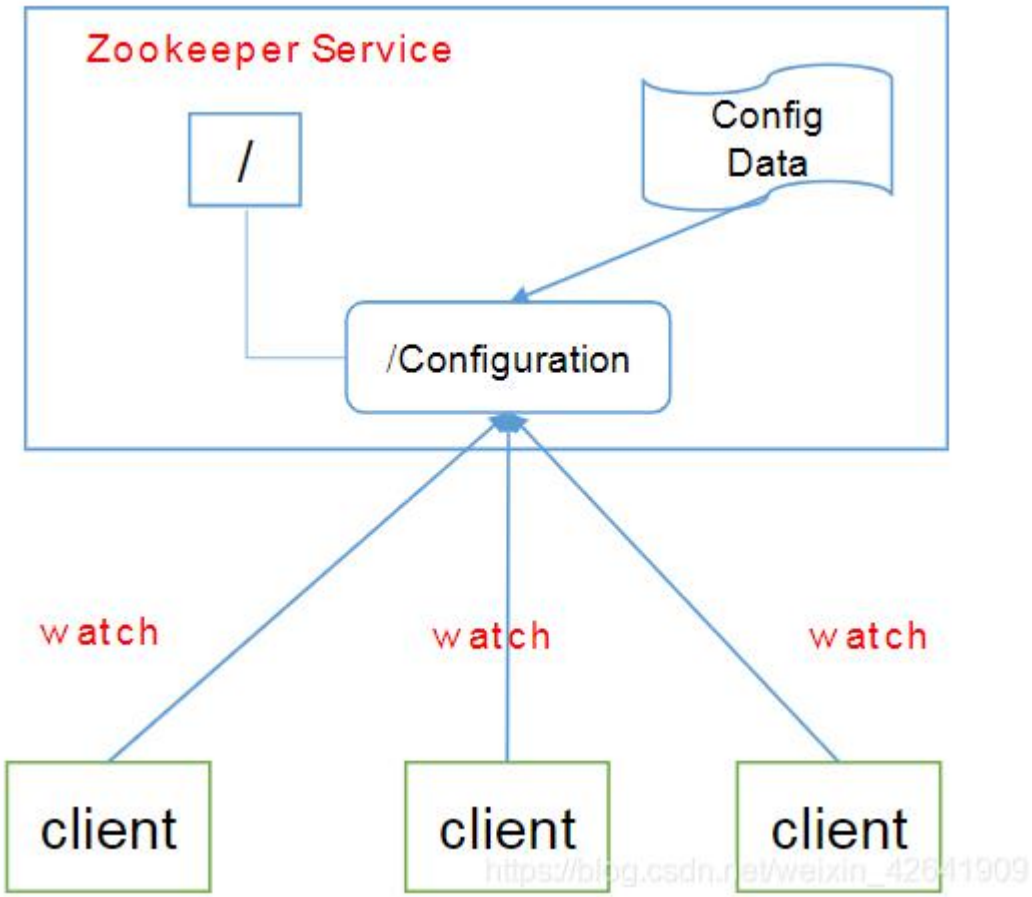


4.2 统一配置管理

统一配置管理，使用的是 Zookeeper 的 watch 机制

- 需求：分布式环境下，要求所有节点的配置信息是一致的，比如 Kafka 集群. 对配置文件修改后，希望能够快速同步到各个节点上.
- 方案：可以把所有的配置都放在一个配置中心，然后各个服务分别去监听配置中心，一旦发现里面的内容发生变化，立即获取变化的内容，然后更新本地配置即可.
- 实现：配置管理可交由 Zookeeper 实现
    - 可将配置信息写入 Zookeeper 上的一个 Znode.
    - 各个客户端服务器监听这个 Znode.

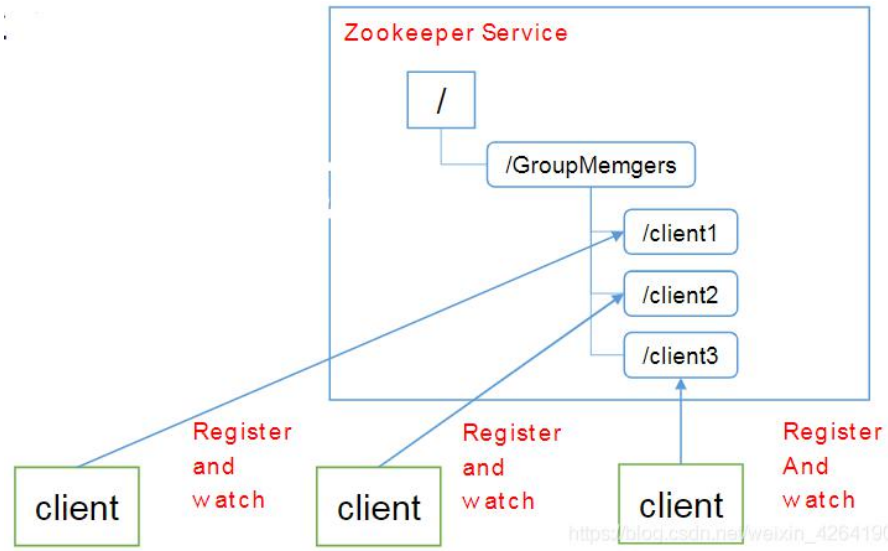
- 一旦 Znode 中的数据被修改, Zookeeper 将通知各个客户端服务器.



4.3 统一集群管理

统一集群管理使用的是 Zookeeper 的 watch 机制

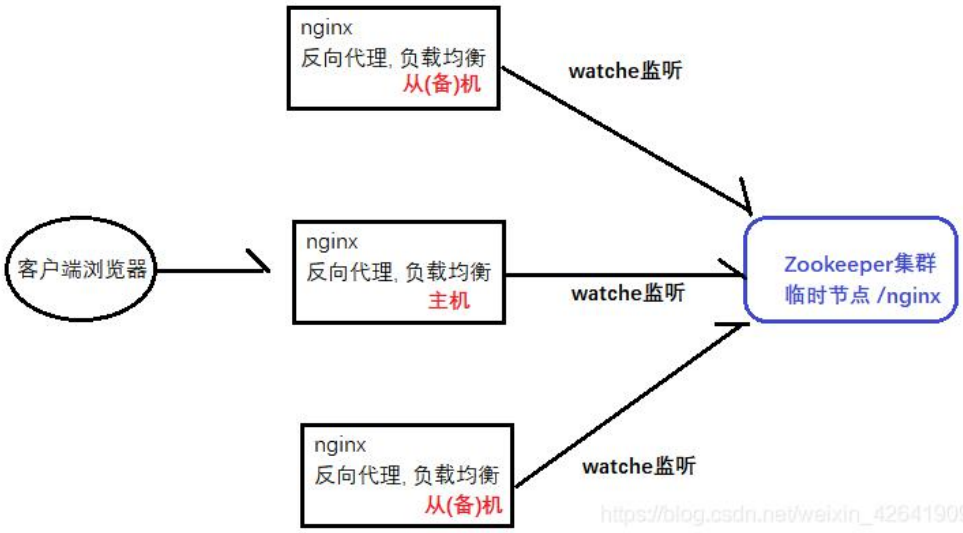
- 需求: 分布式环境中, 实时掌握每个节点的状态是必要的, 可以根据节点实时状态做出一些调整.
- 方案: Zookeeper 可以实现实时监控节点状态变化
  - 可将节点信息写入 Zookeeper 上的一个 Znode.
  - 监听这个 Znode 可获取它的实时状态变化.



4.4 集群选主

集群选主使用的是 zookeeper 的临时节点.

- 需求：在集群中，很多情况下是要区分主从节点的，一般情况下主节点负责数据写入，从节点负责数据读取，那么问题来了，怎么确定哪一个节点是主节点的，当一个主节点宕机的时候，其他从节点怎么再来选出一个主节点呢?
- 实现：使用 Zookeeper 的临时节点可以轻松实现这一需求，我们把上面描述的这个过程称为集群选主的过程，首先所有的节点都认为是从节点，都有机会称为主节点，然后开始选主，步骤比较简单
    - ◆ 所有参与选主的主机都去 Zookeeper 上创建**同一个临时节点**，那么最终一定只有一个客户端请求能够 创建成功。
    - ◆ 成功创建节点的客户端所在的机器就成为了 Master，其他没有成功创建该节点的客户端，成为从节点
    - ◆ 所有的从节点都会在主节点上注册一个子节点变更的 Watcher，用于监控当前主节点是否存活，一旦 发现当前的主节点挂了，那么其他客户端将会重新进行选主。



4.5 分布式锁

分布式锁使用的是 Zookeeper 的**临时有序节点**

- 需求: 在分布式系统中, 很容出现多台主机操作同一资源的情况, 比如两台主机同时往一个文件中追加写入文本, 如果不去做任何的控制, 很有可能出现一个写入操作被另一个写入操作覆盖掉的状况.

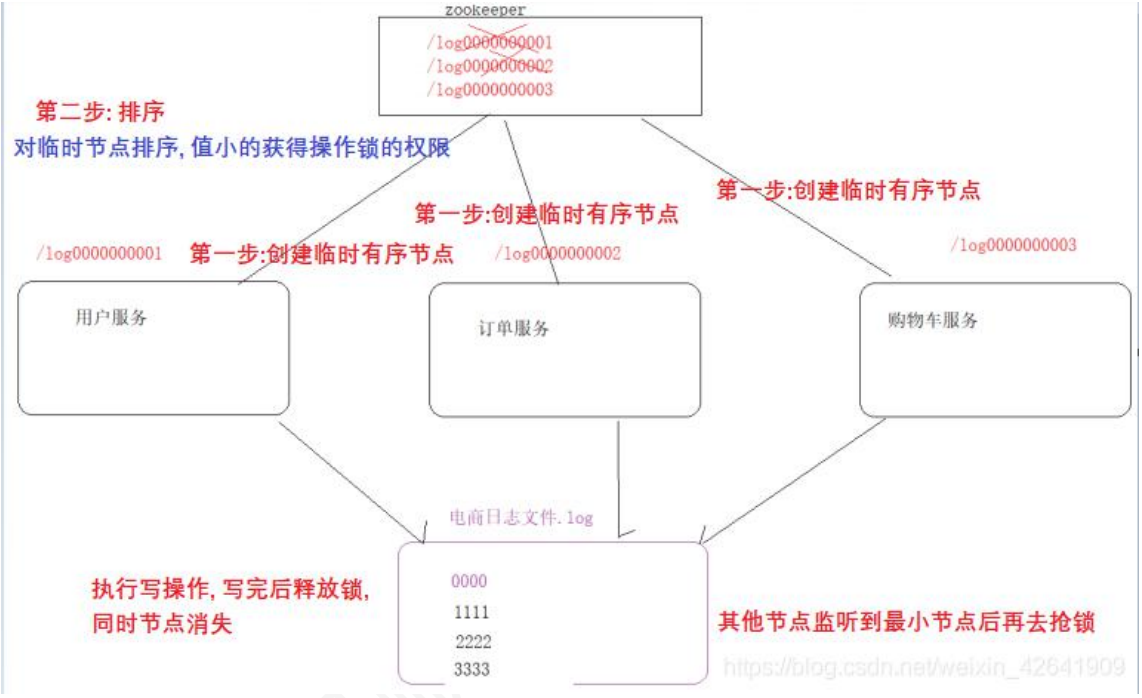
- 方案: 此时我们可以来一把锁, 哪个主机获取到了这把锁, 就执行写入, 另一台主机等待; 直到写入操作执行完毕, 另一台主机再去获得锁, 然后写入.
  这把锁就称为分布式锁, 也就是说:**分布式锁是控制分布式系统之间同步访问共享资源的一种方式**

  - 实现: 使用 Zookeeper 的临时有序节点可以轻松实现这一需求.

  1. 所有需要执行操作的主机都去 Zookeeper 上创建一个**临时有序节点**.
  2. 然后获取到 Zookeeper 上创建出来的这些节点进行一个**从小到大**的排序.
  3. 判断自己创建的节点是不是最小的, 如果是, 自己就获取到了锁; 如果不是, 则对最小的节点注册一个监听.

4. 如果自己获取到了锁，就去执行相应的操作. 当执行完毕之后，连接断开，节点消失，锁就被释放了.

5. 如果自己没有获取到锁，就等待，一直监听节点是否消失，锁被释放后，再重新执行抢夺锁的操作.



5. Zookeeper 集群[高级](高薪常问)

5.1 ZK 集群介绍

Zookeeper 在一个系统中一般会充当一个很重要的角色，所以一定要保证它的高可用，这就需要部署 Zookeeper 的集群. Zookeeper 有三种运行模式：单机模式，集群模式和伪集群模式.

- 单机模式：使用一台主机不是一个 Zookeeper 来对外提供服务，有单点故障问题，仅适合于开发、测试环境.
- 集群模式：使用多台服务器，每台上部署一个 Zookeeper 一起对外提供服务，适合于生产环境.
- 伪集群模式：在服务器不够多的情况下，也可以考虑在一台服务器上部署多个 Zookeeper 来对外提供服务.

5.2 数据一致性处理

ZK 是一个分布式协调开源框架，用于分布式系统中保证数据一致性问题，那么 ZK 是如何保证数据一致性的呢？

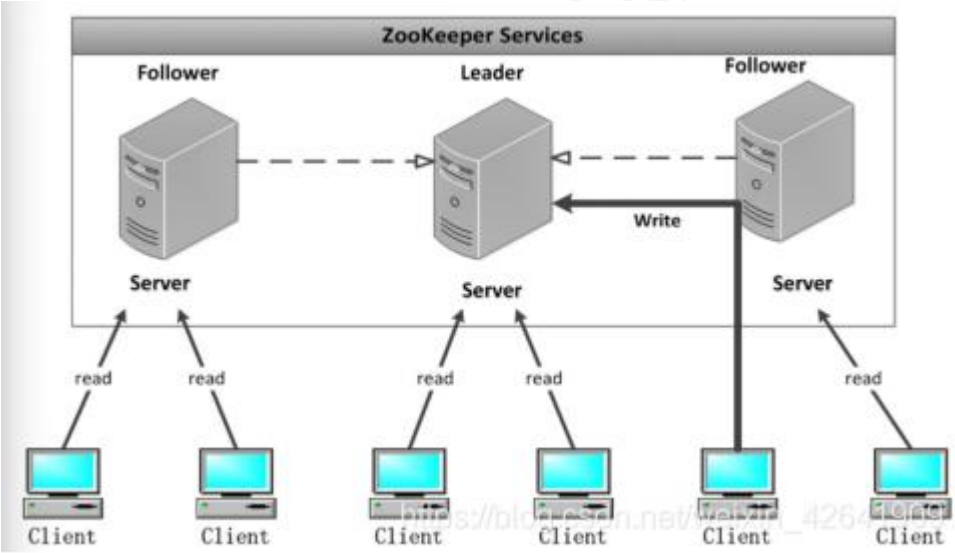
5.2.1 集群角色

Leader：负责投票的发起和决议，更新系统状态，是**事务请求(写请求)** 的唯一处理者，一个 ZooKeeper 同一时刻只会有一个 Leader.
对于 create 创建/setData 修改/delete 删除等有写操作的请求，则需要统一转发给 leader 处理, leader 需要决定编号和执行操作，这个过程称为一个事务.

Follower：接收客户端请求，参与选主投票. 处理客户端非事务(读操作)请求，转发事务请求(写请求)给 Leader；

Observer：针对访问量比较大的 zookeeper 集群，为了增加并发的读请求. 还可新增观察者角色.

作用：可以接受客户端请求，把请求转发给 leader, **不参与投票，只同步 leader 的状态**.



5.2.2 Zookeeper 的特性

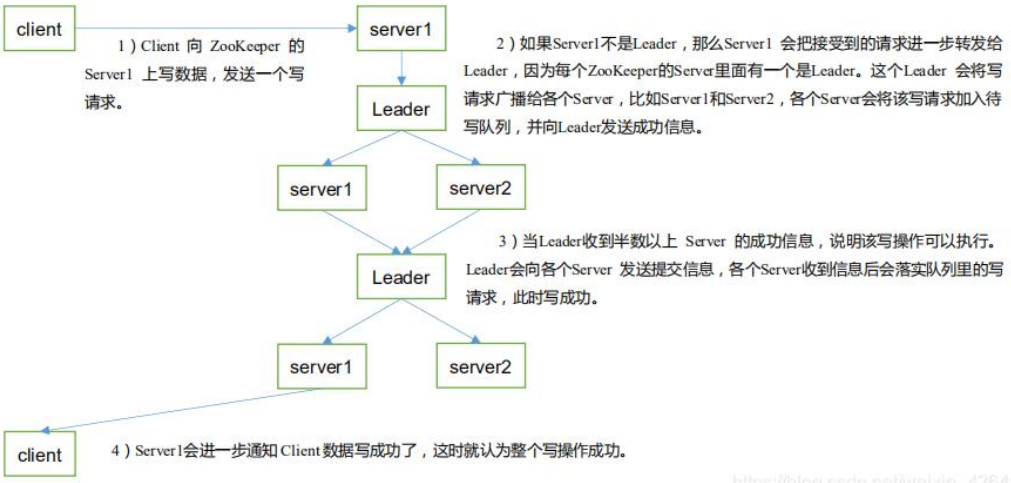
1）Zookeeper：一个领导者(Leader)，多个跟随者(Follower)组成的集群.
2）集群中只要有半数以上节点存活, Zookeeper 集群就能正常服务.
3）**全局数据一致**：每个 Server 保存一份相同的数据副本, Client 无论连接到哪个 Server，数据都是一致的.
4）更新请求**顺序性**：从同一个客户端发起的事务请求,最终会严格按照顺序被应用到 zookeeper 中.

5）**数据更新原子性**：一次数据更新要么成功，要么失败。
6）**实时性**，在一定时间范围内，Client 能读到最新数据。

5.2.3 ZAB 协议

- Zookeeper 采用 ZAB(Zookeeper Atomic Broadcast)协议来保证 分布式数据一致性 。
- ZAB 并不是一种通用的分布式一致性算法,而是一种专为 Zookeeper 设计的崩溃可恢复的原子消息广播算法。
- ZAB 协议包括两种基本模式: 崩溃恢复 模式和 消息广播 模式:
- 消息广播模式主要用来进行事务请求的处理
- 崩溃恢复模式主要用来在集群启动过程,或者 Leader 服务器崩溃退出后进行新的 Leader 服务器的选举以及数据同步.

5.2.4 ZK 集群写数据流程



- Client 向 Zookeeper 的 Server1 上写数据，发送一个写请求.
- 如果 Server1 不是 Leader，那么 Server1 会把接受的请求进一步**转发**给 Leader，因为每个 Zookeeper 的 Server 里面有一个是 Leader. 这个 Leader 会将写请求**广播**给各个 Server，比如 Server1 和 Server2，各个 Server 会将该写请求加入待写队列，并向 Leader 发送成功信息(**ack 反馈机制**).

- 当 Leader 收到**半数以上** Server 的成功信息，说明该写操作可以执行. Leader 会向各个 Server 发送**事务提交**信息，各个 Server 收到信息后会落实队列里面的写请求，此时写成功.
- Server1 会进一步通知 Client 数据写成功了，这是就认为整个写操纵成功.

5.3 ZK 集群选举机制

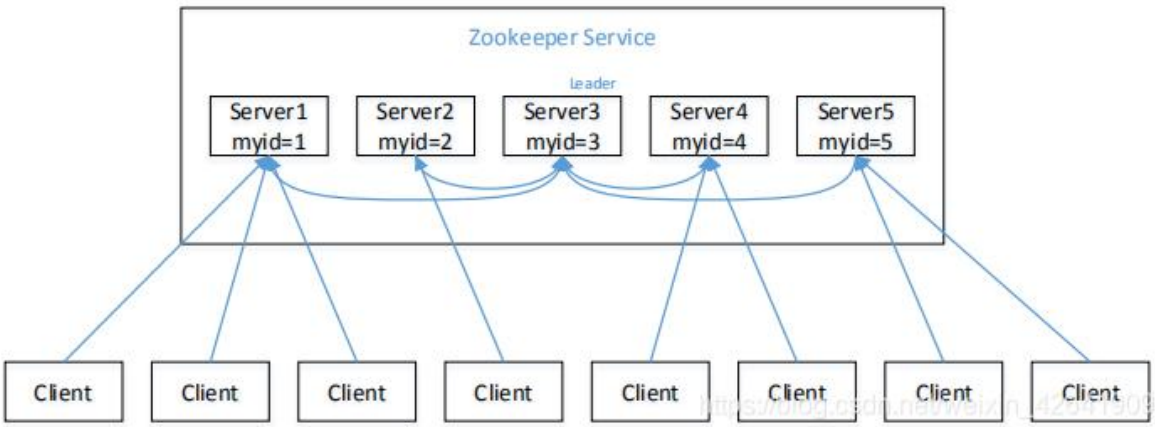
- Zookeeper 服务器有四个状态：
  looking：寻找 leader 状态，当服务器处于该状态时，它会认为当前集群中没有 leader，因此需要进入 leader 选举状态.
  leading：领导者状态，表明当前服务器角色是 leader.
  following：跟随者状态，表明当前服务器角色是 follower.
  observing：观察者状态，表明当前服务器角色是 observer。

  - **半数机制：集群中半数以上机器存活，集群可用，所以 Zookeeper 适合安装奇数台服务器**. 集群启动时，如果当前机器票数超过了总票数一半则为 Leader, Leader 产生后，投过票的机器就不能再投票了.

  - Zookeeper 虽然在配置文件中没有指定主从节点. 但是, Zookeeper 工作时，是有一个节点 Leader，其他则为 Follower, Leader 是通过内部的选举机制临时产生的.
  配置文件中会指定每台 ZK 的 myid，而且不能重复，通常用 1,2,3…区分每台 ZK 的 myid.

5.3.1 集群启动器的选举机制

在集群初始化阶段，当有一台服务器 server1 启动时，其单独无法进行和完成 leader 选举，当第二台服务器 server2 启动时，此时两台机器可以相互通信，每台机器都试图找到 leader，于是进入 leader 选举过程.



1. 服务器 1 启动，服务器 1 状态保持为 looking.

2. 服务器 2 启动，发起一次选举. 服务器 1 投票给比自己 ID 号大的服务器 2. 服务器 2 投票给自己.

投票结果: 服务器 1 票数 0 票，服务器 2 票数 2 票，没有半数以上结果，选举无法完成，服务器 1, 2 状态保持 looking.

3. 服务器 3 启动，发起一次选举. 此时服务器 1 和 2 都会更改选票为服务器 3，服务器 3 投票给自己.

投票结果: 服务器 1 为 0 票，服务器 2 为 0 票，服务器 3 为 3 票. 此时服务器 3 的票数已经超过半数，服务器 3 当选 Leader. 服务器 1，2 更改状态为 follower，服务器 3 更改状态为 leader；

4. 服务器 4 启动，发起一次选举. 此时服务器 1，2，3 已经不是 looking 状态，不会更改选票信息，服务器 4 投票给自己.

投票结果：服务器 3 为 3 票，服务器 4 为 1 票。此时服务器 4 服从多数，更改选票信息为服务器 3，并更改状态为 following；

5. 服务器 5 启动，同 4 一样当小弟.

5.3.2 服务器运行时期的 Leader 选举

在 zk 运行期间, leader 与非 leader 服务器各司其职，即便当有非 leader 服务器宕机或者新加入，此时也不会影响 leader. 但是一旦 leader 服务器宕机了，那么整个集群将会暂停对外服务，进入新一轮 leader 选 举，其过程和启动时期的 Leader 选举过程基本一致.

假设正在运行的有 server1,server2,server3 三台服务器,当前 leader 是 server2,若某一时刻 leader 挂了，此时便开始 leader 选举. 选举过程如下：

- 变更转态, server1 和 server3 变更为 looking 状态.
- 开始投票，每台服务器投票给比自己 myid 大的机器，没有比自己大的就投给自己.
- 这样 server3 有 2 票, server1 有 1 票, server3 的票数超过了集群一半，当选 leader, server1 变更状态 follower.

6. 为什么 zookeeper 集群的数目，一般为奇数个？(高薪常问)

1.容错

由于在增删改操作中需要半数以上服务器通过，来分析以下情况。

2 台服务器，至少 2 台正常运行才行（2 的半数为 1，半数以上最少为 2），正常运行 1 台服务器都不允许挂掉

3 台服务器，至少 2 台正常运行才行（3 的半数为 1.5，半数以上最少为 2），正常运行可以允许 1 台服务器挂掉

4 台服务器，至少 3 台正常运行才行（4 的半数为 2，半数以上最少为 3），正常运行可以允许 1 台服务器挂掉

5 台服务器，至少 3 台正常运行才行（5 的半数为 2.5，半数以上最少为 3），正常运行可以允许 2 台服务器挂掉

6 台服务器，至少 3 台正常运行才行（6 的半数为 3，半数以上最少为 4），正常运行可以允许 2 台服务器挂掉

通过以上可以发现，3 台服务器和 4 台服务器都最多允许 1 台服务器挂掉，5 台服务器和 6 台服务器都最多允许 2 台服务器挂掉

但是明显 4 台服务器成本高于 3 台服务器成本，6 台服务器成本高于 5 服务器成本。这是由于半数以上投票通过决定的。

2.防脑裂

一个 zookeeper 集群中，可以有多个 follower.observer 服务器，但是必需只能有一个 leader 服务器。

如果 leader 服务器挂掉了，剩下的服务器集群会通过半数以上投票选出一个新的 leader 服务器。

集群互不通讯情况：

一个集群 3 台服务器，全部运行正常，但是其中 1 台裂开了，和另外 2 台无法通讯。3 台机器里面 2 台正常运行过半票可以选出一个 leader。

一个集群 4 台服务器，全部运行正常，但是其中 2 台裂开了，和另外 2 台无法通讯。4 台机器里面 2 台正常工作没有过半票以上达到 3，无法选出 leader 正常运行。

一个集群 5 台服务器，全部运行正常，但是其中 2 台裂开了，和另外 3 台无法通讯。5 台机器里面 3 台正常运行过半票可以选出一个 leader。

一个集群 6 台服务器，全部运行正常，但是其中 3 台裂开了，和另外 3 台无法通讯。6 台机器里面 3 台正常工作没有过半票以上达到 4，无法选出 leader 正常运行。

通过以上分析可以看出，为什么 zookeeper 集群数量总是单出现，主要原因还是在于第 2 点，防脑裂，对于第 1 点，无非是正本控制，但是不影响集群正常运行。但是出现第 2 种裂的情况，zookeeper 集群就无法正常运行了。

6.SpringBoot

6.1 SpringBoot 是什么(了解)

是 Spring 的子项目,主要简化 Spring 开发难度,去掉了繁重配置,提供各种启动器,可以让程序员很快上手,节省开发时间.

6.2 SpringBoot 的优点(必会)

SpringBoot 对上述 Spring 的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期。

版本锁定：解决是 maven 依赖版本容易冲突的问题，集合了常用的并且测试过的所有版本

使用了 Starter（启动器）管理依赖并能对版本进行集中控制,如下的父工程带有版本号,就是对版本进行了集中控制.

```
 <!--引入父工程-->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.RELEASE</version>
   </parent>
```

起步依赖 ：解决了完成某一个功能要整合的 jar 包过多的问题，集合了常用的 jar 包
自动配置：解决了整合框架或者技术的配置文件过多，集合了所有的约定的默认配置
内置 Tomcat：通过内置的 tomcat，无需再用其他外置的 Tomcat 就直接可以运行 javaEE 程序
总之：人们把 Spring Boot 称为搭建程序的脚手架。其最主要作用就是帮我们快速的构建庞大的 spring 项目，并且尽可能的减少一切 xml 配置，做到开箱即用，迅速上手，让我们关注与业务而非配置。

6.3 运行 SpringBoot 项目的方式(必会)

- 可以打包
- 可以使用 Maven 插件直接运行.
- 直接运行 main 方法.

6.4 SpringBoot 的启动器 starter(必会)

(1)什么是 starter?

starter 启动器,可以通过启动器集成其他的技术,比如说: web, mybatis, redis 等等.可以提供对应技术的开发和运行环境.
比如: pom 中引入 spring-boot-starter-web, 就可以进行 web 开发.

(2)starter 执行原理?

- SpringBoot 在启动时候会去扫描 jar 包中的一个名为 **spring.factories**.
- 根据文件中的配置,去加载自动配置类. 配置文件格式是 key=value, value 中配置了很多需要 Spring 加载的类.
- Spring 会去加载这些自动配置类, Spring 读取后,就会创建这些类的对象,放到 Spring 容器中.后期就会从 Spring 容器中获取这些类对象.

(3)SpringBoot 中常用的启动器

- spring-boot-starter-web, 提供 web 技术支持
- spring-boot-starter-test
- spring-boot-starter-jdbc
- spring-boot-starter-jpa
- spring-boot-starter-redis…等等

6.5 SpringBoot 运行原理剖析(必会)

(一) SpringApplication 类作用及 run()方法作用

- SpringApplication 这个类整合了其他框架的启动类, 只要运行这一个类,所有的整合就都完成了.
- 调用 run 函数, 将当前启动类的字节码传入(主要目的是传入@SpringBootApplication 这个注解), 以及 main 函数的 args 参数.
- 通过获取当前启动类的核心信息, 创建 IOC 容器.

(二) 当前启动类@SpringBootApplication 详细剖析

run 函数传入的当前启动类字节码, 最重要的是传入了@**SpringBootApplication**,点开该注解源码, 会发现有多个注解组成,接下来会详细解释每个注解的含义.
点开这个注解源码, 发现有 4 类注解.

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
```

(1) 第一类: JDK 原生注解 4 个

@Target(ElementType.TYPE) //当前注解的使用范围

@Retention(RetentionPolicy.RUNTIME) //生命周期

@Documented //声明在生成 doc 文档时是否带着注解

@Inherited //声明是否子类会显示父类的注解

(2)第二类: @SpringBootConfiguration

点开该注解源码，会发现本质是@Configuration,定义该类是个配置类功能等同于 xml 配置文件.

```
import ...

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}
```

提到@Configuration 就要提到他的搭档@Bean，使用这两个注解就可以创建一个简单的 **Spring 配置类**，可以用来替代相应的 **xml 配置文件**.可以理解为创建了 IOC 容器了.

(3)第三类: @ComponentScan，包扫描功能.

这个注解对应 Spring 的 XML 配置中的@ComponentScan，其实就是自动扫描并加载符合条件的组件(比如@Component 和@Repository 等)或者 bean 定义，最终将这些 bean 定义加载到 IoC 容器中.

也可以通过 basePackages 等属性来细粒度的定制@ComponentScan 自动扫描的范围，如果不指定，则默认扫描@ComponentScan 所在类的 package 及子包进行扫描。

注：所以 SpringBoot 的启动类最好是放在 root package 下，因为默认不指定 basePackages，这样能扫描 root package 及子包下的所有类.

(4)第四类: @EnableAutoConfiguration

点开源码会发现,本质是@import，自动导入功能

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```


1. @EnableAutoConfiguration 也是借助@Import 的帮助，将所有符合自动配置条件的 bean 定义加载到 IoC 容器.

@EnableAutoConfiguration 会根据类路径中的 jar 依赖为项目进行自动配置，如：添加了 spring-boot-starter-web 依赖，会自动添加 Tomcat 和 SpringMVC 的依赖，SpringBoot 会对 Tomcat 和 SpringMVC 进行自动配置.

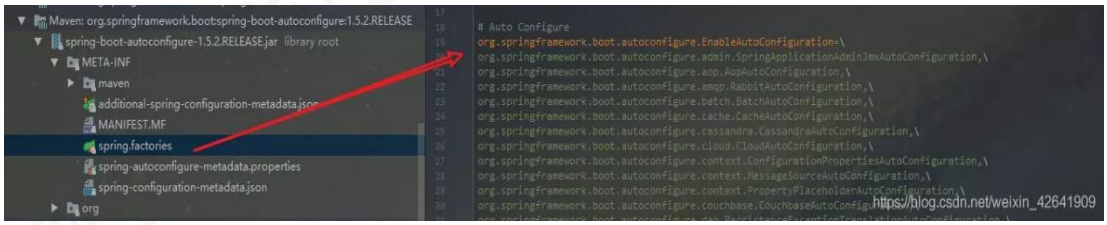
2. 那么 SpringBoot 是如何完成自动配置的呢?

A. SpringBoot 自动配置的注解是 @EnableAutoConfiguration.

B. 我们用的时候是在启动类上加@SpringBootApplication,这个注解是复合注解,内部包含 @EnableAutoConfiguration

C. @EnableAutoConfiguration 内部有一个@Import，这个注解才是完成自动配置的关键.

D. @Import 导入一个类(AutoConfigurationImportSelector),这个类内部提供了一个方法(selectImports). 这个方法会扫描导入的所有 jar 包下的 spring.factories 文件. 解析文件中自动配置类 key=value，将列表中的类创建,并放到 Spring 容器中.



8.5.3 总结

总之一个@**SpringBootApplication** 注解就搞定了所有事，它封装了核心的 @SpringBootConfiguration+@EnableAutoConfiguration+@ComponentScan 这三个类,大大节省了程序员配置时间,这就是 SpringBoot 的核心设计思想.

6.6 SpringBoot 热部署(了解)

导入 spring-boot-devtools 这个 jar 包：就可以完成热部署了.

6.7 SpringBoot 中的配置文件(必会)

(1)有哪些配置文件?

application.yml 或 application.properties

bootstrap.yml 或 bootstrap.properties

(2)**上面两种配置文件有什么区别?**

1. bootstrap 由父 ApplicationContext 加载，比 application 配置文件优先被加载.
2. bootstarp 里的属性不能被覆盖.
3. application: springboot 项目中的自动化配置.
4. bootstrap:
   使用 spring cloud config 配置中心时，需要加载连接配置中心的配置属性的，就可以使用 bootstrap 来完成.
   加载不能被覆盖的属性.
   加载一些加密/解密的数据.

(3)读取配置文件的方式?

- 读取默认配置文件

  需要注入 Environment 类，使用 environment.getProperty(peorperties 中的 key)，这样就能获得 key 对应的 value 值

  @value(${key.value}) 直接读取

- 读取自定义配置文件
  - 自定义配置文件后缀必须是.propeties
    - 编写和自定义配置文件对应的 java 类，类上放 3 个注解
      - @ConfigurationProperties(“前缀”)
      - @PropertySource(“指定配置文件”)
      - @Component 包扫描
  - 读取的时候就跟读取默认配置文件一样.

6.8 SpringBoot 支持哪些日志框架(了解)

Java Utils logging

Log4j2

Lockback

如果你使用了启动器,那么 springboo 默认将 Lockback 作为日志框架.

6.9 SpringBoot 常用注解(必会)

- @SpringBootApplication: 它封装了核心的@SpringBootConfiguration +@EnableAutoConfiguration +@ComponentScan 这三个类,大大节省了程序员配置时间,这就是 SpringBoot 的核心设计思想.
- @EnableScheduling 是通过@Import 将 Spring 调度框架相关的 bean 定义都加载到 IoC 容器
- @MapperScan:spring-boot 支持 mybatis 组件的一个注解,通过此注解指定 mybatis 接口类的路径，即可完成对 mybatis 接口的扫描
- @RestController 是 @Controller 和 @ResponseBody 的结合,一个类被加上 @RestController 注解,数据接口中就不再需要添加@ResponseBody,更加简洁。
- @RequestMapping,我们都需要明确请求的路径.
- @GetMappping,@PostMapping, @PutMapping, @DeleteMapping 结合 @RequestMapping 使用, 是 Rest 风格的, 指定更明确的子路径.
- @PathVariable：路径变量注解，用{}来定义 url 部分的变量名.
- @Service 这个注解用来标记业务层的组件，我们会将业务逻辑处理的类都会加上这个注解交给 spring 容器。事务的切面也会配置在这一层。当让 这个注解不是一定要用。有个泛指组件的注解，当我们不能确定具体作用的时候 可以用泛指组件的注解托付给 spring 容器
- @Component 和 spring 的注解功能一样, 注入到 IOC 容器中.
- @ControllerAdvice 和 @ExceptionHandler 配合完成统一异常拦截处理.

备注: 面试的时候记住 6.7 个即可~

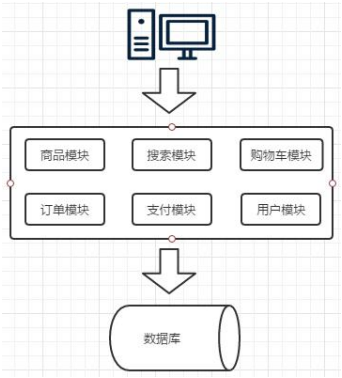
7. SpringCloud

7.1 SOA 和微服务的区别?(必会)

谈到 SOA 和微服务的区别，那咱们先谈谈架构的演变

1. 集中式架构

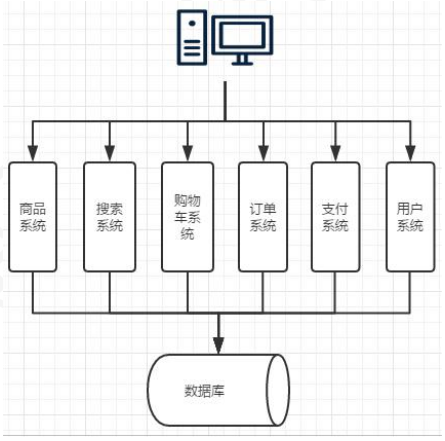
项目功能简单，一个项目只需一个应用，将所有功能部署在一起，这样的架构好处是减少了部署节点和成本.



缺点: 代码耦合，开发维护困难，无法水平扩展，单点容错率低，并发能力差

2. 垂直拆分架构

当访问量逐渐增大，单一应用无法满足需求，此时为了应对更高的并发和业务需求，我们根据业务功能对系统进行拆分：



优点：

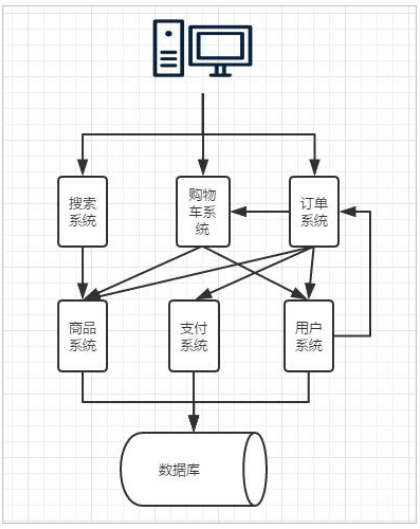
- 系统拆分实现了流量分担，解决了并发问题
- 可以针对不同模块进行优化，方便水平扩展，负载均衡，容错率提高

缺点：

◆ 系统间相互独立，会有很多重复开发工作，影响开发效率

3. 分布式服务

当垂直应用越来越多，随着项目业务功能越来越复杂，并非垂直应用这一条线进行数据调用，应用和应用之间也会互相调用，也就是完成某一个功能,需要多个应用互相调用，这就是将功能拆完来完成的分布式架构.



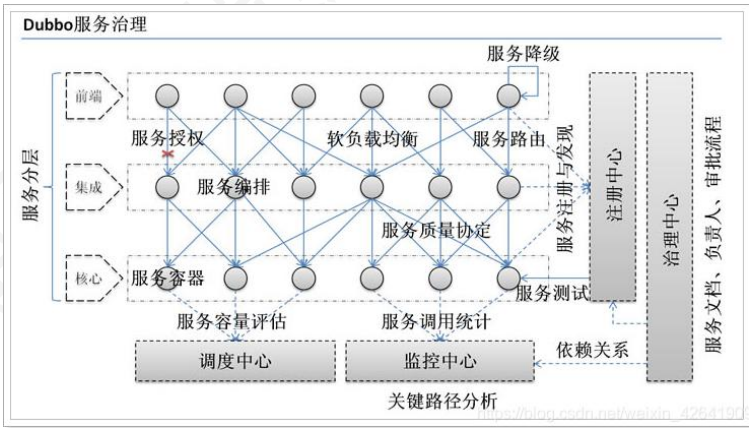
优点: 将基础服务进行了抽取，系统间相互调用，提高了代码复用和开发效率

缺点: 系统间耦合度变高，调用关系错综复杂，难以维护.

4. 服务治理架构 SOA

SOA ： 面向服务的架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键，而最初的服务治理基石是 Dubbo 服务治理



以前分布式服务的问题?

- 服务越来越多，需要管理每个服务的地址，调用关系错综复杂，难以理清依赖关系
- 服务过多，服务状态难以管理，无法根据服务情况动态管理

SOA 服务治理架构的优点:

- 服务注册中心，实现服务自动注册和发现，无需人为记录服务地址

- 服务自动订阅，服务列表自动推送，服务调用透明化，无需关心依赖关系
- 动态监控服务状态监控报告，人为控制服务状态

SOA 服务治理架构的缺点：

- 服务间依然会有依赖关系，一旦某个环节出错会影响较大(容错机制)
- 服务关系复杂，运维、测试部署困难，不符合开发-运维一站式维护的思想.

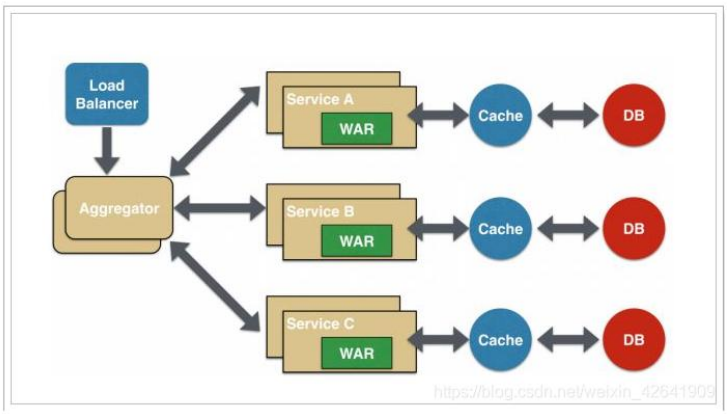
5. 微服务

前面说的 SOA，英文翻译过来是面向服务。微服务，似乎也是服务，都是对系统进行拆分。因此两者非常容易混淆，但其实缺有一些差别：

微服务的特点：

- 单一职责：微服务中每一个服务都对应唯一的业务能力，做到单一职责.
- 微：微服务的服务拆分粒度很小，例如一个用户管理就可以作为一个服务。每个服务虽小，但“五脏俱全”。
- 面向服务：面向服务是说每个服务都要对外暴露 Rest 风格服务接口 API。并不关心服务的技术实现，做到与平台和语言无关，也不限定用什么技术实现，只要提供 Rest 的接口即可。
- 自治：自治是说服务间互相独立，互不干扰
  - 团队独立：每个服务都是一个独立的开发团队，人数不能过多。
  - 技术独立：因为是面向服务，提供 Rest 接口，使用什么技术没有别人干涉
  - 前后端分离：采用前后端分离开发，提供统一 Rest 接口，后端不用再为 PC、移动段开发不同接口
  - 数据库分离：每个服务都使用自己的数据源
  - 部署独立，服务间虽然有调用，但要做到服务重启不影响其它服务。有利于持续集成和持续交付。每个服务都是独立的组件，可复用，可替换，降低耦合，易维护. 基于 docker 容器是开发.

微服务结构图：



目前微服务微服务架构主流的是 SpringBoot+Dubbo 和 SpringBoot+SpringCloud

的架构模式.

综上，无论是 SOA 还是微服务，都需要进行服务调度，目前主流的服务调度室 RPC 和 HTTP 两种协议，而 Dubbo 基于 RPC 的远程调度机构, SpringCloud 是基于 Rest 风格(基于 http 协议实现的)的 Spring 全家桶微服务服务治理框架. 说到这里也可以继续说下 Dubbo 和 SpringCloud 的区别.

7.2 SpringCloud 是什么?(了解)

SpringCloud 是一系列框架的集合，集成 SpringBoot，提供很多优秀服务：服务发现和注册，统一配置中心，负载均衡，网关，熔断器等的一个微服务治理框架.

7.3 SpringCloud 的优势?(了解)

- 因为 SpringCloud 源于 Spring，所以它的质量，稳定性，持续性都是可以保证的。
- SpringCloiud 天热支持 SpringBoot 框架，就可以提高开发效率，能够实现需求。
- SpringCloud 更新很快，后期支持很给力。
- SpringCloud 可以用来开发微服务。

7.4 SpringCloud 有哪些核心组件?(必会)

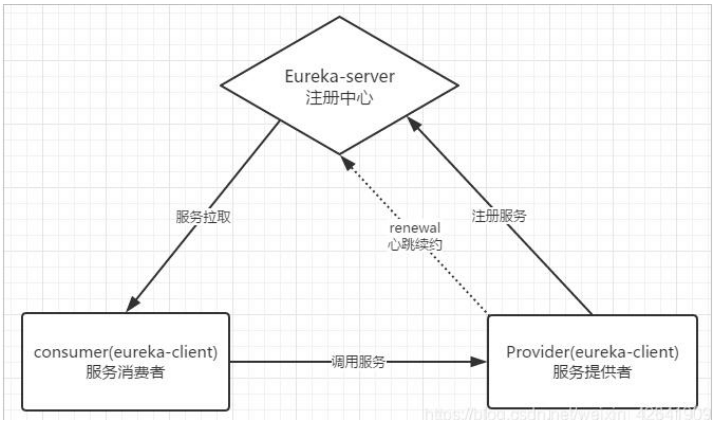
- Eureka: 注册中心，服务注册和发现
- Ribbon: 负载均衡，实现服务调用的负载均衡
- Hystrix: 熔断器
- Feign: 远程调用
- Zuul: 网关
- Spring Cloud Config: 配置中心

  (1)Eureka

  提供服务注册和发现，是注册中心. 有两个组件: Eureka 服务端和 Eureka 客户端

- Eureka 服务端: 作为服务的注册中心，用来提供服务注册，支持集群部署.
- Eureka 客户端: 是一个 java 客户端，将服务注册到服务端，同事将服务端的信息缓存到本地，客户端和服务端定时交互.

1. 原理



- Eureka-Server：就是服务注册中心（可以是一个集群），对外暴露自己的地址。
- 提供者：启动后向 Eureka 注册自己信息（地址，服务名称等），并且定期进行服务续约
- 消费者：服务调用方，会定期去 Eureka 拉取服务列表，然后使用负载均衡算法选出一个服务进行调用。
- 心跳(续约)：提供者定期通过 http 方式向 Eureka 刷新自己的状态

2. 服务下线、失效剔除和自我保护

- 服务的注册和发现都是可控制的，可以关闭也可以开启。默认都是开启
- 注册后需要心跳，心跳周期默认 30 秒一次，超过 90 秒没发心跳认为宕机
- 服务拉取默认 30 秒拉取一次.
- Eureka 每个 60 秒会剔除标记为宕机的服务
- Eureka 会有自我保护，当心跳失败比例超过阈值(默认 85%)，那么开启自我保护，不再剔除服务。
- Eureka 高可用就是多台 Eureka 互相注册在对方上.

(2)Ribbon

- Ribbon 是 Netflix 发布的云中服务开源项目．给客户端提供负载均衡，也就是说 Ribbon 是作用在消费者方的.
- 简单来说，它是一个客户端负载均衡器，它会自动通过某种算法去分配你要连接的机器.
- SpringCloud 认为 Ribbon 这种功能很好，就对它进行了封装，从而完成负载均衡.
- Ribbon 默认负责均衡策略是轮询策略.

(3)Hystrix 熔断器

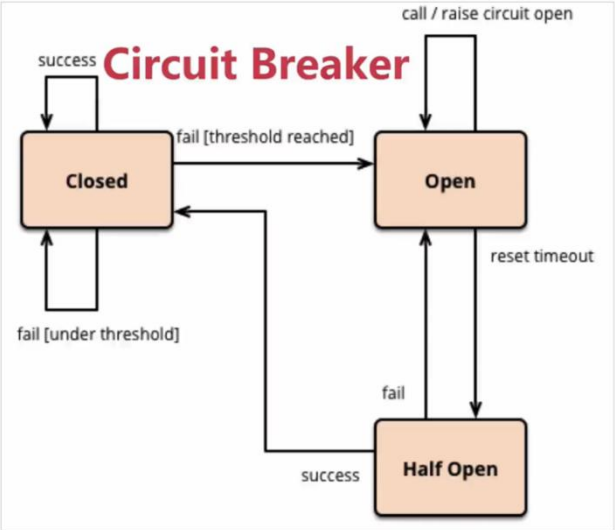
- 有时候可能是网络问题，一些其它问题，导致代码无法正常运行，这是服务就挂了，崩溃了．熔断器就是为了解决无法正常访问服务的时，提供的一种解决方案.
- 解决因为一个服务崩溃而引起的一系列问题，使问题只局限于这个服务中,不会影响其他服务.

- Hystrix 提供了两种功能，一种是服务降级，一种是服务熔断.

1. 服务降级原理

- Hystrix 为每个服务分配了小的线程池，当用户发请求过来，会通过线程池创建线程来执行任务，当创建的线程池已满或者请求超时(这里和多线程线程池不一样,不存在任务队列)，则启动服务降级功能.
- 降级指的请求故障时，不会阻塞，会返回一个友好提示(可以自定义，例如网站维护中请稍后重试)， 也就是说不会影响其他服务的运行.

2. 服务熔断原理



状态机有 3 个状态：

- Closed：关闭状态（断路器关闭），所有请求都正常访问。
- Open：打开状态（断路器打开），所有请求都会被降级。Hystix 会对请求情况计数，当一定时间内失败请求百分比达到阈值，则触发熔断，断路器会完全关闭。默认失败比例的阈值是 50%，请求次数最少不低于 20 次。
- Half Open：半开状态，open 状态不是永久的，打开后会进入休眠时间（默认是 5S）。随后断路器会自动进入半开状态。此时会释放 1 次请求通过，若这个请求是健康的，则会关闭断路器，否则继续保持打开，再次进行 5 秒休眠计时。

(4)Feign：远程调用组件

- 后台系统中，微服务和微服务之间的调用可以通过 Feign 组件来完成.
- Feign 组件集成了 Ribbon 负载均衡策略(默认开启的，使用轮询机制), Hystrix 熔断器(默认关闭的，需要通过配置文件进行设置开启)
- 被调用的微服务需要提供一个接口，加上@@FeignClient("url")注解
- 调用方需要在启动类上加上@EnableFeignClients，开启 Feign 组件功能.

(5)Gateway: 路由/网关

- 对于项目后台的微服务系统，每一个微服务都不会直接暴露给用户来调用的，但是如果用户知道了某一个服务的 ip:端口号:url:访问参数，就能直接访问你. 如果再是恶意访问，恶意攻击，就会击垮后台微服务系统.因此，需要一个看大门的大 boss，来保护我们的后台系统.

- Gateway 支持过滤器功能，对请求或响应进行拦截，完成一些通用操作。

  Gateway 提供两种过滤器方式： “pre”和“post”

  pre 过滤器，在转发之前执行，可以做参数校验、权限校验、流量监控、日志输出、协议转换等。

  post 过滤器，在后端微服务响应之后并且给前端响应之前执行，可以做响应内容、响应头的修改，日志的输出，流量监控等。

- Gateway 还提供了两种类型过滤器

  (一) GatewayFilter：局部过滤器，针对单个路由

  1. GatewayFilter 局部过滤器，是针对单个路由的过滤器。
  2. 在 Spring Cloud Gateway 组件中提供了大量内置的局部过滤器，对请求和响应做过滤操作。
  3. 遵循约定大于配置的思想，只需要在配置文件配置局部过滤器名称，并为其指定对应的值，就可以让其生效.

  (二) GlobalFilter ： 全局过滤器，针对所有路由.

  1. GlobalFilter 全局过滤器，不需要在配置文件中配置，系统初始化时加载，并作用在每个路由上。
  2. Spring Cloud Gateway 核心的功能也是通过内置的全局过滤器来完成。
  3. 自定义全局过滤器步骤：

     ① 定义类实现 GlobalFilter 和 Ordered 接口

     ② 复写方法

     ③ 完成逻辑处理

(6)Spring Cloud Config

- 在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在 Spring Cloud 中，有分布式配置中心组件 spring Cloud Config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程 Git 仓库中.

7.5 SpringBoot 和 SpringCloud 的关系(必会)

- SpringBoot 是为了解决 Spring 配置文件冗余问题，简化开发的框架.
- SpringCloud 是为了解决微服务之间的协调和配置问题，还有服务之间的通信，熔断，负载均衡远程调度任务框架.
- SpringCloud 需要依赖 SpringBoot 搭建微服务, SpringBoot 使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，SpringCloud 很大的一部分是基于 SpringBoot 来实现。
- SpringBoot 不需要依赖 SpringCloud 就可以独立开发. SpringBoot 也可以集成 Dubbo 进行开发.

7.6 SpringCloud 和 Dubbo 的区别(高薪常问)

- SpringCloud 和 Dubbo 都是主流的微服务架构.
  - SpringCloud 是 Apache 下的 Spring 体系下的微服务解决方案.
  - Dubbo 是阿里系统中分布式微服务治理框架.
- 技术方面对比
  - SpringCloud 功能远远超过 Dubbo, Dubbo 只实现了服务治理(注册和发现). 但是 SpringCloud 提供了很多功能，有 21 个子项目
  - Dubbo 可以使用 Zookeeper 作为注册中心，实现服务的注册和发现，SpringCloud 不仅可以使用 Eureka 作为注册中心，也可以使用 Zookeeper 作为注册中心.
  - Dubbo 没有实现网关功能，只能通过第三方技术去整合. 但是 SpringCloud 有 zuul 路由网关，对请求进行负载均衡和分发. 提供熔断器，而且和 git 能完美集成.
- 性能方面对比
  - 由于 Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC。
  - 而 SpringCloud 是基于 Http 协议+Rest 接口调用远程过程的，相对来说，Http 请求会有更大的报文，占的带宽也会更多。
  - 使用 Dubbo 时，需要给每个实体类实现序列化接口，将实体类转化为二进制进行 RPC 通信调用.而使用 SpringCloud 时，实体类就不需要进行序列化.

刚才有提到注册中心不一样,那么 Eureka 和 Zookeeper 有什么区别？我们继续往下说~

7.7 Eureka 和 Zookeeper 的区别(高薪常问)

在谈这个问题前我们先看下 CAP 原则: C(Consistency)-数据一致性; A(Availability)-服务可用性; P(Partition tolerance)-服务对网络分区故障的容错性，这三个特性在任何分布式系统中不能同时满足，最多同时满足两个，而且 P(分区容错性)是一定要满足的.

- Eureka 满足 AP(服务可用性和容错性), Zookeeper 满足 CP(数据一致性和容错性)
- Zookeeper 满足 CP，数据一致性，服务的容错性. 数据在各个服务间同步完成后才返回用户结果，而且如果服务出现网络波动导致监听不到服务心跳，会立即从服务列表中剔除，服务不可用.
- Eureka 满足 AP，可用性，容错性. 当因网络故障时, Eureka 的自我保护机制不会立即剔除服务，虽然用户获取到的服务不一定是可用的，但至少能够获取到服务列表. 用户访问服务列表时还可以利用重试机制，找到正确的服务. 更服务分布式服务的高可用需求.
- Eureka 集群各节点平等, Zookeeper 集群有主从之分.

    1. 如果 Zk 集群中有服务宕机,会重新进行选举机制,选择出主节点，因此可能会导致整个集群因为选主而阻塞，服务不可用.

    2. Eureka 集群中有服务宕机,因为是平等的各个服务器,所以其他服务器不受影响.

- Eureka 的服务发现者会主动拉取服务, ZK 服务发现者是监听机制

    1. Eureka 中获取服务列表后会缓存起来，每隔 30 秒重新拉取服务列表.

    2. Zk 则是监听节点信息变化，当服务节点信息变化时，客户端立即就得到通知.

五.技术点

1. Redis

1.1 Redis 是什么?

Redis 是 C 语言开发的一个开源的（遵从 BSD 协议）高性能非关系型（NoSQL）的（key-value）键值对数据库。可以用作数据库、缓存、消息中间件等。

1.2 Redis 的存储结构有哪些?

String，字符串，是 redis 的最基本的类型，一个 key 对应一个 value。是二进制安全的，最大能存储 512MB。

Hash，散列，是一个键值(key=>value)对集合。string 类型的 field 和 value 的映射表，特别适合用于存储对象。每个 hash 可以存储 232 -1 键值对（40 多亿）

List，列表，是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列边或者尾部（右边）。最多可存储 232 - 1 元素(4294967295，每个列表可存储 40 亿)

Set，集合，是 string 类型的无序集合，最大的成员数为 232 -1(4294967295，每个集合可存储 40 多亿个成员)。

Sorted set，有序集合，和 set 一样也是 string 类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。zset 的成员是唯一的,但分数(score)却可以重复。

1.3 Redis 的优点?

1 因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value 数据库。Redis 支持事务 、持久化

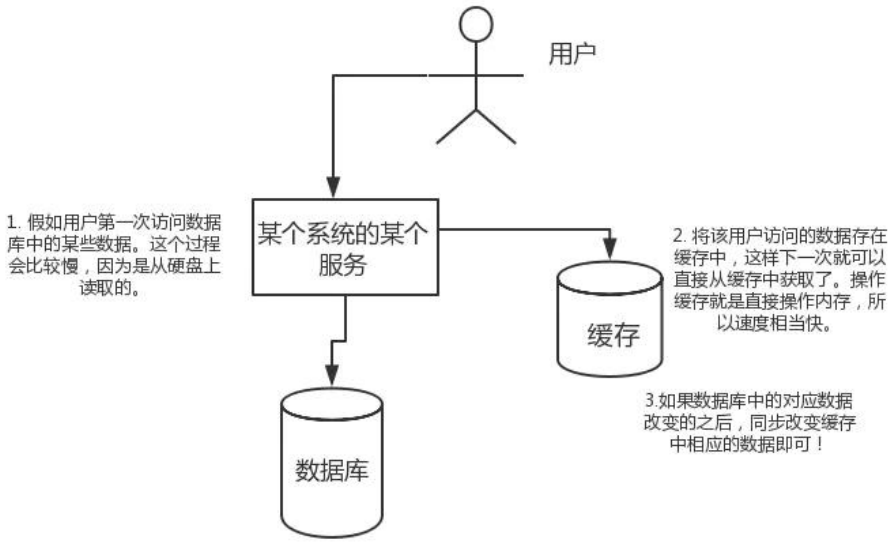
2、单线程操作，避免了频繁的上下文切换。

3、采用了非阻塞 I/O 多路复用机制。I/O 多路复用就是只有单个线程，通过跟踪每个 I/O 流的状态，来管理多个 I/O 流。

1.4 为什么要用 Redis

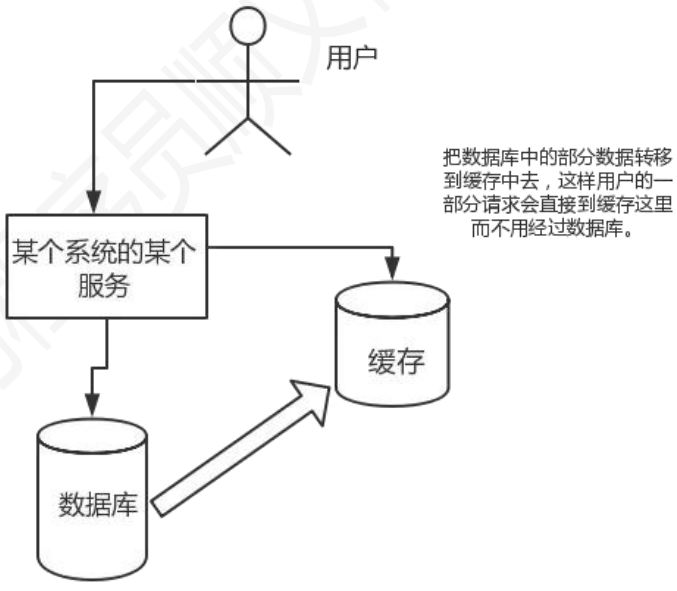
高性能:

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可!



高并发：

直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。



1.5 redis 的持久化

Redis 提供了两种持久化的方式，分别是 RDB（Redis DataBase）和 AOF（Append Only File）。

RDB，简而言之，就是在不同的时间点，将 redis 存储的数据生成快照并存储到磁盘等介质上。

AOF，则是换了一个角度来实现持久化，那就是将 redis 执行过的所有写指令记录下来，在下次 redis 重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。

RDB 和 AOF 两种方式也可以同时使用，在这种情况下，如果 redis 重启的话，则会优先采用 AOF 方式来进行数据恢复，这是因为 AOF 方式的数据恢复完整度更高。

1.6 Redis 的缺点

1.6.1 缓存和数据库双写一致性问题

一致性的问题很常见，因为加入了缓存之后，请求是先从 redis 中查询，如果 redis 中存在数据就不会走数据库了，如果不能保证缓存跟数据库的一致性就会导致请求获取到的数据不是最新的数据。

解决方案：

1、编写删除缓存的接口，在更新数据库的同时，调用删除缓存
的接口删除缓存中的数据。这么做会有耦合高以及调用接口失败的情况。

2、消息队列：ActiveMQ，消息通知。

1.6.2 缓存的并发竞争问题

并发竞争，指的是同时有多个子系统去 set 同一个 key 值。

解决方案：

1、最简单的方式就是准备一个分布式锁，大家去抢锁，抢到
锁就做 set 操作即可

1.6.3 缓存雪崩问题

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波
请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案：

1、给缓存的失效时间，加上一个随机值，避免集体失效。

2、使用互斥锁，但是该方案吞吐量明显下降了。

3、搭建 redis 集群。

1.6.4 缓存击穿问题

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

解决方案：

1、利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，
再去请求数据库。没得到锁，则休眠一段时间重试

2、采用异步更新策略，无论 key 是否取到值，都直接返回，
value 值中维护一个缓存失效时间，缓存如果过期，异步起一个线程
去读数据库，更新缓存。

1.7 Redis 集群

1.7.1 主从复制

主从复制原理

从服务器连接主服务器，发送 SYNC 命令。主服务器接收到 SYNC 命名后，开始执行 BGSAVE 命令生成 RDB 文件并使用缓冲区记录此后执行的所有写命令。主服务器 BGSAVE 执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令。从服务器收到快照文件后丢弃所有旧数据，载入收到的快照。主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令。

从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令（从服务器初始化完成）。主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令（从服务器初始化完成后的操作）。

优点

支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。为了分载 Master 的读操作压力，Slave 服务器可以为客户端提供只读操作的服务，写服务仍然必须由 Master 来完成 Slave 同样可以接受其它 Slaves 的连接和同步请求，这样可以有效的分载 Master 的同步压力 Master Server 是以非阻塞的方式为 Slaves 提供服务。所以在 Master-Slave 同步期间，客户端仍然可以提交查询或修改请求。Slave Server 同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis 则返回同步之前的数据。

缺点

Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的 IP 才能恢复。主机宕机，宕机前有部分数据未能及时同步到从机，切换 IP 后还会引入数据不一致的问题，降低了系统的可用性。Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。

1.7.2 哨兵模式

当主服务器中断服务后，可以将一个从服务器升级为主服务器，以便继续提供服务，但是这个过程需要人工手动来操作。为此，Redis2.8 中提供了哨兵工具来实现自动化的系统监控和故障恢复功能。哨兵的作用就是监控 Redis 系统的运行状况，它的功能包括以下两个。

1、监控主服务器和从服务器是否正常运行。

2、主服务器出现故障时自动将从服务器转换为主服务器。

哨兵的工作方式

每个 Sentinel （哨兵）进程以每秒钟一次的频率向整个集群中的 Master 主服务器，Slave 从务器以及其他 Sentinel（哨兵）进程发送一个 PING 命令。如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel（哨兵）进程标记为主观下线（SDOWN）。如果一个 Master 主服务器被标记为主观下线（SDOWN），则正在监视这个 Master 主服务器的所有 Sentinel（哨兵）进程要以每秒一次的频率确认 Master 主服务器的确进入了主观下线状态。当有足够数量的 Sentinel（哨兵）进程（大于等于配置文件指定的值）在指定的时间范围内确认 Master 主服务器进入了主观下线状态（SDOWN），则 Master 主服务器会被标记为客观下线（ODOWN）。

在一般情况下， 每个 Sentinel（哨兵）进程会以每 10 秒一 169 / 196 次的频率向集群中的所有 Master 主服务器，Slave 从服务器发送 INFO 命令。当 Master 主服务器被 Sentinel（哨兵）进程标记为客观下线（ODOWN）时，Sentinel（哨兵）进程向下线的 Master 主服务器的所有 Slave 从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。若没有足够数量的 Sentinel (哨兵) 进程同意 Master 主服务器下线， Master 主服务器的客观下线状态就会被移除。若 Master 主服务器重新向 Sentinel （哨兵）进程发送 PING 命令返回有效回复，Master 主服务器的主观下线状态就会被移除。

优点

哨兵模式是基于主从模式的，所有主从的优点，哨兵模式都具有。主从可以自动切换，系统更健壮，可用性更高。

缺点

Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。

1.7.3 Redis-Cluster 集群

redis 的哨兵模式基本已经可以实现高可用，读写分离，但是在这种模式下每台 redis 服务器都存储相同的数据，很浪费内存，所以在 redis3.0 上加入了 cluster 模式，实现的

redis 的分布式存储，也就是说每台 redis 节点上存储不同的内容。Redis-Cluster 采用无中心结构,它的特点如下：

所有的 redis 节点彼此互联(PING-PONG 机制),内部使用二进制协议优化传输速度和带宽。节点的 fail 是通过集群中超过半数的节点检测失效时才生效。客户端与 redis 节点直连,不需要中间代理层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。

工作方式

在 redis 的每一个节点上，都有这么两个东西，一个是插槽（slot），它的取值范围是：0-16383。还有一个就是 cluster，可以理解为是一个集群管理的插件。当我们的存取的 key 到达的时候，redis 会根据 crc16 的算法得出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作。为了保证高可用，redis-cluster 集群引入了主从模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点。当其它主节点 ping 一个主节点 A 时，如果半数以上的主节点与 A 通信超时，那么认为主节点 A 宕机了。如果主节点 A 和它的从节点 A1 都宕机了，那么该集群就无法再提供服务了。

1.8 Redis 的分布式锁

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 Redlock，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1>安全特性：互斥访问，即永远只有一个 client 能拿到锁

2>避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区

3>容错性：只要大部分 Redis 节点存活就可以正常提供服务

Redis 实现分布式锁

Redis 为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对 Redis 的连接并不存在竞争关系 Redis 中可以使用 SETNX 命令实现分布式锁。

当且仅当 key 不存在，将 key 的值设为 value。 若给定的 key 已经存在，则 SETNX 不做任何动作

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

返回值：设置成功，返回 1 。设置失败，返回 0 。

```
127.0.0.1:6379> setnx lock-key value1
(integer) 1
127.0.0.1:6379> setnx lock-key value2
(integer) 0
127.0.0.1:6379> get lock-key
"value1"
```

使用 SETNX 完成同步锁的流程及事项如下：

使用 SETNX 命令获取锁，若返回 0（key 已存在，锁已存在）则获取失败，反之获取成功

为了防止获取锁后程序出现异常，导致其他线程/进程调用 SETNX 命令总是返回 0 而进入死锁状态，需要为该 key 设置一个“合理”的过期时间

释放锁，使用 DEL 命令将锁数据删除

2. RocketMQ

2.1 消息中间件的区别?

MQ	描述
RabbitMQ	erlang 开发，对消息堆积的支持并不好，当大量消息积压的时候，会导致 RabbitMQ 的性能急剧下降。每秒钟可以处理几万到十几万条消息。
RocketMQ	java 开发，面向互联网集群化功能丰富，对在线业务的响应时延做了很多的优化，大多数情况下可以做到毫秒级的响应，每秒钟大概能处理几十万条消息。
Kafka	Scala 开发，面向日志功能丰富，性能最高。当你的业务场景中，每秒钟消息数量没有那么多的时候，Kafka 的时延反而会比较高。所以，Kafka 不太适合在线业务场景。
ActiveMQ	java 开发，简单，稳定，性能不如前面三个。小型系统用也 ok，但是不推荐。推荐用互联网主流的。

2.2 为什么要使用 MQ?

因为项目比较大，做了分布式系统，所有远程服务调用请求都是同步执行经常出问题，所以引入了 mq

作用	描述
解耦	系统耦合度降低，没有强依赖关系
异步	不需要同步执行的远程调用可以有效提高响应时间
削峰	请求达到峰值后，后端 service 还可以保持固定消费速率消费，不会被压垮

2.3 RocketMQ 由哪些角色组成，每个角色作用和特点是什么?

生产者（Producer）：负责产生消息，生产者向消息服务器发送由业务应用程序系统生成的消息。

消费者（Consumer）：负责消费消息，消费者从消息服务器拉取信息并将其输入用户应用程序。

消息服务器（Broker）：是消息存储中心，主要作用是接收来自 Producer 的消息并存储， Consumer 从这里取得消息。

名称服务器（NameServer）：用来保存 Broker 相关 Topic 等元信息并给 Producer ，提供 Consumer 查找 Broker 信息。

2.4 RocketMQ 消费模式有几种?

消费模型由 Consumer 决定，消费维度为 Topic。

集群消费

1.一条消息只会被同 Group 中的一个 Consumer 消费

2.多个 Group 同时消费一个 Topic 时，每个 Group 都会有一个 Consumer 消费到数据

广播消费

消息将对一 个 Consumer Group 下的各个 Consumer 实例都消费一遍。即即使这些 Consumer 属于同一个 Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

2.5 RocketMQ 如何做负载均衡?

通过 Topic 在多 Broker 中分布式存储实现。

(1)producer 端

发送端指定 message queue 发送消息到相应的 broker，来达到写入时的负载均衡

- 提升写入吞吐量，当多个 producer 同时向一个 broker 写入数据的时候，性能会下降
- 消息分布在多 broker 中，为负载消费做准备

默认策略是随机选择：

- producer 维护一个 index
- 每次取节点会自增
- index 向所有 broker 个数取余
- 自带容错策略

其他实现：

- SelectMessageQueueByHash
- hash 的是传入的 args
- SelectMessageQueueByRandom
- SelectMessageQueueByMachineRoom 没有实现

也可以自定义实现 **MessageQueueSelector** 接口中的 select 方法

```
MessageQueue select(final List<MessageQueue> mqs, final Message msg, final Object arg);
```

(2) consumer 端

采用的是平均分配算法来进行负载均衡。

其他负载均衡算法

平均分配策略(默认)(AllocateMessageQueueAveragely)

环形分配策略(AllocateMessageQueueAveragelyByCircle)

手动配置分配策略(AllocateMessageQueueByConfig)

机房分配策略(AllocateMessageQueueByMachineRoom)

一致性哈希分配策略(AllocateMessageQueueConsistentHash)

靠近机房策略(AllocateMachineRoomNearby)

追问：当消费负载均衡 consumer 和 queue 不对等的时候会发生什么?

Consumer 和 queue 会优先平均分配，如果 Consumer 少于 queue 的个数，则会存在部分 Consumer 消费多个 queue 的情况，如果 Consumer 等于 queue 的个数，那就是一个 Consumer 消费一个 queue，如果 Consumer 个数大于 queue 的个数，那么会有部分 Consumer 空余出来，白白的浪费了。

2.6 消息重复消费如何解决?

影响消息正常发送和消费的重要原因是网络的不确定性。

出现原因

正常情况下在 consumer 真正消费完消息后应该发送 ack，通知 broker 该消息已正常消费，从 queue 中剔除

当 ack 因为网络原因无法发送到 broker，broker 会认为词条消息没有被消费，此后会开启消息重投机制把消息再次投递到 consumer。

消费模式：在 CLUSTERING 模式下，消息在 broker 中会保证相同 group 的 consumer 消费一次，但是针对不同 group 的 consumer 会推送多次

解决方案

- 数据库表：处理消息前，使用消息主键在表中带有约束的字段中 insert
- Map：单机时可以使用 map 做限制，消费时查询当前消息 id 是不是已经存在
- Redis：使用分布式锁。

2.7 如何让 RocketMQ 保证消息的顺序消费

你们线上业务用消息中间件的时候，是否需要保证消息的顺序性?

如果不需要保证消息顺序，为什么不需要?假如我有一个场景要保证消息的顺序，你们应该如何保证?

首先多个 queue 只能保证单个 queue 里的顺序，queue 是典型的 FIFO，天然顺序。多个 queue 同时消费是无法绝对保证消息的有序性的。所以总结如下：

同一 topic，同一个 QUEUE，发消息的时候一个线程去发送消息，消费的时候 一个线程去消费一个 queue 里的消息。

追问：怎么保证消息发到同一个 queue?

Rocket MQ 给我们提供了 MessageQueueSelector 接口，可以自己重写里面的接口，实现自己的算法，举个最简单的例子：判断 i % 2 == 0，那就都放到 queue1 里，否则放到 queue2 里。

2.8 RocketMQ 如何保证消息不丢失

首先在如下三个部分都可能会出现丢失消息的情况：

Producer 端

Broker 端

Consumer 端

2.8.1 Producer 端如何保证消息不丢失

采取 send()同步发消息，发送结果是同步感知的。

发送失败后可以重试，设置重试次数。默认 3 次。

producer.setRetryTimesWhenSendFailed(10);

集群部署，比如发送失败了的原因可能是当前 Broker 宕机了，重试的时候会发送到其他 Broker 上。

2.8.2 Broker 端如何保证消息不丢失

修改刷盘策略为同步刷盘。默认情况下是异步刷盘的。

flushDiskType = SYNC_FLUSH

集群部署，主从模式，高可用。

2.8.3 Consumer 端如何保证消息不丢失

完全消费正常后在进行手动 ack 确认。

2.9 RocketMQ 的消息堆积如何处理

1. 如果可以添加消费者解决，就添加消费者的数据量.

2、如果出现了 queue，但是消费者多的情况。可以使用准备一个临时的 topic，同时创建一些 queue，在临时创建一个消费者来把这些消息转移到 topic 中，让消费者消费。

2.10 RocketMQ 如何实现分布式事务?

1、生产者向 MQ 服务器发送 half 消息。

2、half 消息发送成功后，MQ 服务器返回确认消息给生产者。

3、生产者开始执行本地事务。

4、根据本地事务执行的结果（UNKNOW、commit、rollback）向 MQ Server 发送提交或回滚消息。

5、如果错过了（可能因为网络异常、生产者突然宕机等导致的异常情况）提交/回滚消息，则 MQ 服务器将向同一组中的每个生产者发送回查消息以获取事务状态。

6、回查生产者本地事物状态。

7、生产者根据本地事务状态发送提交/回滚消息。

8、MQ 服务器将丢弃回滚的消息，但已提交（进行过二次确认的 half 消息）的消息将投递

给消费者进行消费。

Half Message：预处理消息，当 broker 收到此类消息后，会存储到 RMQ_SYS_TRANS_HALF_TOPIC 的消息消费队列中

检查事务状态：Broker 会开启一个定时任务，消费 RMQ_SYS_TRANS_HALF_TOPIC 队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，Broker 会定时去回调在重新检查。

超时：如果超过回查次数，默认回滚消息。

也就是他并未真正进入 Topic 的 queue，而是用了临时 queue 来放所谓的 half message，等提交事务后才会真正的将 half message 转移到 topic 下的 queue。

2.11 任何一台 Broker 突然宕机了怎么办?

Broker 主从架构以及多副本策略。Master 收到消息后会同步给 Slave，这样一条消息就不止一份了，Master 宕机了还有 slave 中的消息可用，保证了 MQ 的可靠性和高可用性。而且 Rocket MQ4.5.0 开始就支持了 Dlegder 模式，基于 raft 的，做到了真正意义的 HA。

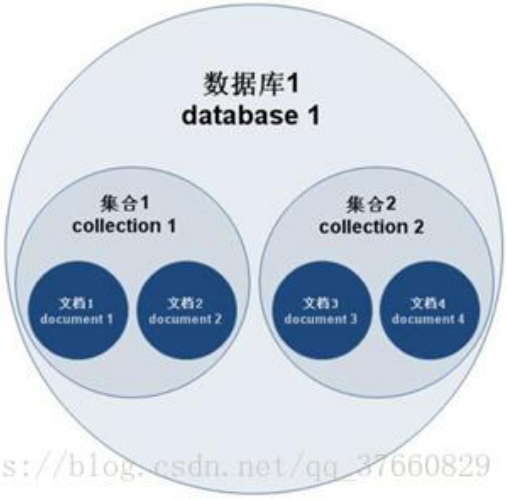
3. MongoDb

3.1 MongoDB 是什么?

mongodb 是属于文档型的非关系型数据库，是开源、高性能、高可用、可扩展的

数据逻辑层次关系：文档=>集合=>数据库

在关系型数据库中每一行的数据对应 mongodb 里是一个文档。mongodb 的文档是以 BSON（binary json）格式存储的，其格式就是 json 格式。



1>集合

集合是一组文档（即上面的 users 集合）。集合相当于关系数据库中的表，但集合中的文档长度可不同（集合中的文档中的键值对个数可不同）、集合中文档的 key 可不同。向集合中插入第一个文档时，集合会被自动创建。

2>文档

文档是一组键值对，用{ }表示，字段之间用逗号分隔。相当于关系数据库中的一行（一条记录）。示例：一个文档

```
{
    id: "1",
    name: "张三",
    age: 28,
    email: "zhangs@gmail.com"
}
```

这样写是为了方便看字段，也可以写在一起{id:"1",name:"张三",age:28,email:"zhangs@gmail.com"}，一样的。

说明：

文档中的键值对是有序的

一个文档中不能有重复的 key（对应关系数据库中的一条记录）

以"_"开头的 key 是保留的，有特殊含义。

3>字段

即一个键值对，key 必须是 String 类型，value 可以是任意类型。

3.2 MongoDB 和关系型数据库 mysql 区别

上图中，左边的是 MySQL 数据库中 users 表，右边是 MongoDB 中 users 集合。虽然表现形式不同，但是数据内容还是一样的。其中：

test：表示数据库
users：表示集合，类似 MySQL 中的表
{id:"1",name:"张三",age:28,email:"zhangs@gmail.com"}：表示一个文档，类似于 MySQL 中的记录
id、name、age 和 email：表示字段

关系数据库	MongoDB
数据库 database	数据库 database
表格 table	集合 collection
行 row	文档 document
列 column	字段 field
索引 index	索引 index
表联合 table joins	嵌入文档
主键 primary key	主键 primary key 。MongoDB 默认主键为 _id

3.3 MongoDB 有 3 个数据库

一个 MongoDB 中可以建立多个数据库，这些数据库是相互独立的，有自己的集合和权限。不同的数据库使用不同的文件存储（不存储在一个文件中）。

MongoDB 默认有 3 个数据库：

admin： 从权限的角度来看，这是"root"数据库。将一个用户添加到这个数据库，这个用户会自动继承所有数据库的权限。一些特定的服务器端命令也只能在这个数据库中运行，比如列出所有的数据库或者关闭服务器。

local: 这个数据库永远不会被复制，里面的数据都是本地的（不会复制到其他 MongoDB 服务器上），可以用来存储限于本地单台服务器的任意集合

config: 当 Mongo 用于分片设置时，config 数据库在内部使用，用于保存分片的相关信息。

3.4 Mongo 中的数据类型

1. null
2. false 和 true
3. 数值
4. UTF-8 字符串
5. 日期 new Date()
6. 正则表达式
7. 数组
8. 嵌套文档

9. 对象 ID ObjectId()
10. 二进制数据
11. 代码

3.5 MongoDB 适用业务场景

网站数据：MongoDB 非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性

缓存：由于性能很高，MongoDB 也适合作为信息基础设施的缓存层。在系统重启之后，由 MongoDB 搭建的持久化缓存层可以避免下层的数据源过载

大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储

高伸缩性的场景：MongoDB 非常适合由数十或数百台服务器组成的数据库。MongoDB 的路线图中已经包含对 MapReduce 引擎的内置支持

用于对象及 JSON 数据的存储：MongoDB 的 BSON 数据格式非常适合文档化格式的存储及查询.

4. Nginx

4.1 Nginx 是什么?

Nginx 是一个高性能的 HTTP 和反向代理服务器，及电子邮件代理服务器，同时也是一个非常高效的反向代理、负载平衡。

4.2 Nginx 的作用?

1.反向代理，将多台服务器代理成一台服务器。

2.负载均衡，将多个请求均匀的分配到多台服务器上，减轻每台服务器的压力，提高服务的吞吐量。

3.动静分离，nginx 可以用作静态文件的缓存服务器，提高访问速度

4.3 Nginx 的优势?

(1) 可以高并发连接（5 万并发，实际也能支持 2~4 万并发）。
(2) 内存消耗少。
(3) 成本低廉。
(4) 配置文件非常简单。
(5) 支持 Rewrite 重写。

(6) 内置的健康检查功能。
(7) 节省带宽。
(8) 稳定性高。
(9) 支持热部署。

4.4 什么是反向代理?

反向代理是指以代理服务器来接受 internet 上的连接请求，然后将请求，发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

反向代理总结就一句话：代理端代理的是服务端.

4.5 什么是正向代理?

一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端才能使用正向代理。

正向代理总结就一句话：代理端代理的是客户端。

4.6 什么是负载均衡?

负载均衡即是代理服务器将接收的请求均衡的分发到各服务器中，负

载均衡主要解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量，减少后台服务器大并发压力。

4.7 Nginx 是如何处理一个请求的?

首先，nginx 在启动时，会解析配置文件，得到需要监听的端口与 ip 地址，然后在 nginx 的 master 进程里面先初始化好这个监控的 socket，再进行 listen,然后再 fork 出多个子进程出来，子进程会竞争 accept 新的连接。

此时,客户端就可以向 nginx 发起连接了。当客户端与 nginx 进行三次握手，与 nginx

建立好一个连接后,此时，某一个子进程会 accept 成功，然后创建 nginx 对连接的封装，即 ngx_connection_t 结构体,接着，根据事件调用相应的事件处理模块，如 http 模块与客户端进行数据的交换。

最后，nginx 或客户端来主动关掉连接，到此，一个连接就寿终正寝了。

4.8 为什么 Nginx 性能这么高?

得益于它的事件处理机制：异步非阻塞事件处理机制：运用了 epoll 模型，提供了一个队列，排队解决。

5. FastDFS

5.1 FastDFS 是什么?

FastDFS 是一个开源的轻量级分布式文件系统，它可以对文件进行管理，功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大 容量存储和负载均衡的问题。特别适合以文件为载体的在线服务，如相册网站、 视频网站等等。

5.2 FastDFS 组成

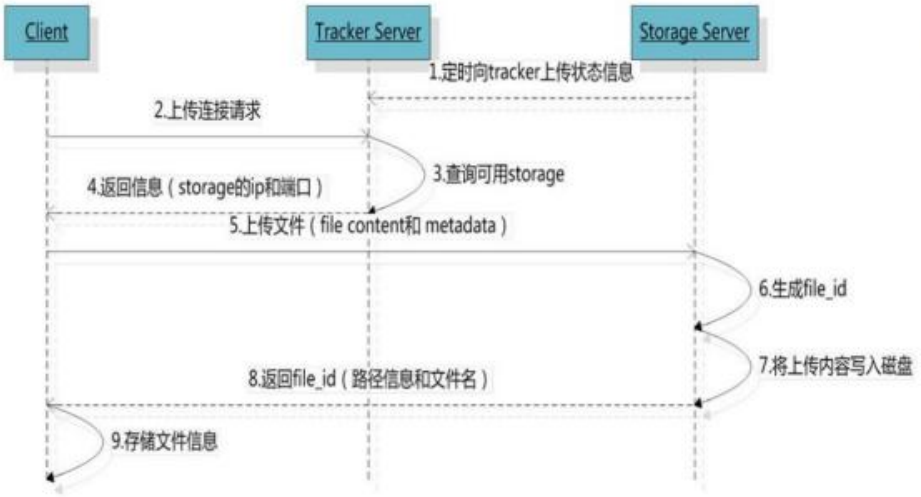
1. Storage server(存储服务器)

Storage server 一般都是以组(group)为组织单位，一个组中有多个 Storageserver，数据互为备份(意味着每个 Storageserver 的内容是一致的，他们之间没有主从之分)，组的存储空间以组内最小的 Storage server 为准，所以为了避 免浪费存储空间最好的话每个 Storage server 的配置最好相同。

2. Tracker server(调度服务器、追踪服务器)

232Tracker server 主要负责管理所有的 Storage server 和 group，每个 storage 在启动后会连接 Tracker，告知自己所属的 group 等信息，并保持周期性的心跳，tracker 根据 storage 的心跳信息，建立 group==>[storage server list]的映射表。

5.3 FastDFS 的流程



1、选择 tracker server

当集群中不止一个 tracker server 时，由于 tracker 之间是完全对等的关系，客户端在 upload 文件时可以任意选择一个 trakcer。

2、选择存储的 group

当 tracker 接收到 upload file 的请求时，会为该文件分配一个可以存储该文件的 group，支持如下选择 group 的规则：

1. Round robin，所有的 group 间轮询。
2. Specified group，指定某一个确定的 group。
3. Load balance，剩余存储空间多多 group 优先。
4. 选择 storage server。

3、选择 storage server

233 当选定 group 后，tracker 会在 group 内选择一个 storage server 给客户端，支持如下选择 storage 的规则：

1. Round robin，在 group 内的所有 storage 间轮询。
2. First server ordered by ip，按 ip 排序。
3. First server ordered by priority，按优先级排序（优先级在 storage 上配置）。

4、选择 storage path

当分配好 storage server 后，客户端将向 storage 发送写文件请求，storage 将会为文件分配一个数据存储目录，支持如下规则：

1. Round robin，多个存储目录间轮询。
2. 剩余存储空间最多的优先。

5、生成 Fileid

选定存储目录之后，storage 会为文件生一个 Fileid，由 storage server ip、文件创建时间、文件大小、文件 crc32 和一个随机数拼接而成，然后将这个二进制串进行 base64 编码，转换为可打印的字符串。

6、选择两级目录

当选定存储目录之后，storage 会为文件分配一个 fileid，每个存储目录下有两级 256*256 的子目录，storage 会按文件 fileid 进行两次 hash （猜测），路由到其中一个子目录，然后将文件以 fileid 为文件名存储到该子目录下。

7、生成文件名

当文件存储到某个子目录后，即认为该文件存储成功，接下来会为 234 该文件生成一个文件名，文件名由 group、存储目录、两级子目录、fileid、 文件后缀名（由客户端指定，主要用于区分文件类型）拼接而成。

5.4 FastDFS 如何现在组内的多个 storage server 的数据同步?

当客户端完成文件写至 group 内一个 storage server 之后即认为文件上传成功，storage server 上传完文件之后，会由后台线程将文件同步至同 group 内其他的 storage server。后台线程同步参考的依据是每个 storageserver 在 写完文件后，同时会写一份 binlog，binlog 中只包含文件名等元信息,storage 会记录向 group 内其他 storage 同步的进度，以便重启后能接上次的进度继续同步；进度以时间戳的方式进行记录，所以最好能保证集群内所有 server 的时钟保持同步。

6. JWT

JSON Web token 简称 JWT， 是用于对应用程序上的用户进行身份验证的标记。也就是说，使用 JWTS 的应用程序不再需要保存有关其用户的 cookie 或其他 session 数据。此特性便于可伸缩性，同时保证应用程序的安全。

在身份验证过程中，当用户使用其凭据成功登录时，将返回 JSON Web token，并且必须在本地保存 (通常在本地存储中)。

每当用户要访问受保护的路由或资源 (端点) 时，用户代理(user agent)必须连同请求一起发送 JWT，通常在授权标头中使用 Bearer schema。后端服务器接收到带有 JWT 的请求时，首先要做的是验证 token。

6.1 组成

一个 JWT 实际上就是一个字符串，它由三部分组成，头部、载荷与签名。

头部（Header)

头部用于描述关于该 JWT 的最基本的信息，例如其类型以及签名所用的算法等。这也可以被表示成一个 JSON 对象。

{"typ":"JWT","alg":"HS256"}

在头部指明了签名算法是 HS256 算法。 我们进行 BASE64 编码（ http://base64.xpcha.com/ ），编码后的字符串如下：eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

载荷（playload）

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分：

1. 标准中注册的声明（建议但不强制使用）

iss: jwt 签发者

sub: jwt 所面向的用户

aud: 接收 jwt 的一方

exp: jwt 的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该 jwt 都是不可用的.

iat: jwt 的签发时间

jti: jwt 的唯一身份标识，主要用来作为一次性 token

2 .公共的声明

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密。

3. 私有的声明

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为 base64 是对称解密的，意味着该部分信息可以归类为明文信息。

这个指的就是自定义的 claim。比如前面那个结构举例中的 admin 和 name 都属于自定的 claim。这些 claim 跟 JWT 标准规定的 claim 区别在于：JWT 规定的 claim，

JWT 的接收方在拿到 JWT 之后，都知道怎么对这些标准的 claim 进行验证(还不知道是否能够验证)；而 private claims 不会验证，除非明确告诉接收方要对这些 claim 进行验证以及规则才行。定义一个 payload：

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

然后将其进行 base64 加密，得到 Jwt 的第二部分。

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRta
W4iOnRydWV9
```

签证（signature）

jwt 的第三部分是一个签证信息，这个签证信息由三部分组成：

header (base64 后的)

payload (base64 后的)

secret

这个部分需要 base64 加密后的 header 和 base64 加密后的 payload 使用.连接组成的字符串，然后通过 header 中声明的加密方式进行加盐 secret 组合加密，然后就构成了 jwt 的第三部分。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

注意

secret 是保存在服务器端的，jwt 的签发生成也是在服务器端的，secret 就是用来进行 jwt 的签发和 jwt 的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个 secret，那就意味着客户端是可以自我签发 jwt 了。

6.2 使用场景

1. 一次性验证

比如用户注册后需要发一封邮件让其激活账户，通常邮件中需要有一个链接，这个链接需要具备以下的特性：能够标识用户，该链接具有时效性（通常只允许几小时之内激活），不能被篡改以激活其它可能的账户……这种场景就和 jwt 的特性非常贴近，jwt 的 payload 中固定的参数：iss 签发者和 exp 过期时间正是为其做准备的。

2. restful api 的无状态认证

使用 jwt 来做 restful api 的身份认证也是值得推崇的一种使用方案。客户端和服务端共享 secret；过期时间由服务端校验，客户端定时刷新；签名信息不可被修改.....spring security oauth jwt 提供了一套完整的 jwt 认证体系，以笔者的经验来看：使用 oauth2 或 jwt 来做 restful api 的认证都没有大问题，oauth2 功能更多，支持的场景更丰富，后者实现简单。

3.使用 jwt 做单点登录+会话管理(不推荐)

6.3 面试问题：

1. JWT token 泄露了怎么办？(常问)

使用 https 加密你的应用，返回 jwt 给客户端时设置 httpOnly=true 并且使用 cookie 而不是 LocalStorage 存储 jwt，这样可以防止 XSS 攻击和 CSRF 攻击。

2. Secret 如何设计?

jwt 唯一存储在服务端的只有一个 secret，个人认为这个 secret 应该设计成和用户相关的属性，而不是一个所有用户公用的统一值。这样可以有效的避免一些注销和修改密码时遇到的窘境。

3. 注销和修改密码?

传统的 session+cookie 方案用户点击注销，服务端清空 session 即可，因为状态保存在服务端。但 jwt 的方案就比较难办了，因为 jwt 是无状态的，服务端通过计算来校验有效性。没有存储起来，所以即使客户端删除了 jwt，但是该 jwt 还是在有效期内，只不过处于一个游离状态。分析下痛点：注销变得复杂的原因在于 jwt 的无状态。提供几个方案，视具体的业务来决定能不能接受：

仅仅清空客户端的 cookie，这样用户访问时就不会携带 jwt，服务端就认为用户需要

重新登录。这是一个典型的假注销，对于用户表现出退出的行为，实际上这个时候携带对应的 jwt 依旧可以访问系统。

清空或修改服务端的用户对应的 secret，这样在用户注销后，jwt 本身不变，但是由于 secret 不存在或改变，则无法完成校验。这也是为什么将 secret 设计成和用户相关的原因。

借助第三方存储自己管理 jwt 的状态，可以以 jwt 为 key，实现去 Redis 一类的缓存中间件中去校验存在性。方案设计并不难，但是引入 Redis 之后，就把无状态的 jwt 硬生生变成了有状态了，违背了 jwt 的初衷。实际上这个方案和 session 都差不多了。

修改密码则略微有些不同，假设号被到了，修改密码（是用户密码，不是 jwt 的 secret）之后，盗号者在原 jwt 有效期之内依旧可以继续访问系统，所以仅仅清空 cookie 自然是不够的，这时，需要强制性的修改 secret。

4. 如何解决续签问题

传统的 cookie 续签方案一般都是框架自带的，session 有效期 30 分钟，30 分钟内如果有访问，session 有效期被刷新至 30 分钟。而 jwt 本身的 payload 之中也有一个 exp 过期时间参数，来代表一个 jwt 的时效性，而 jwt 想延期这个 exp 就有点身不由己了，因为 payload 是参与签名的，一旦过期时间被修改，整个 jwt 串就变了，jwt 的特性天然不支持续签。

解决方案

1. 每次请求刷新 jwt。

jwt 修改 payload 中的 exp 后整个 jwt 串就会发生改变，那就让它变好了，每次请求都返回一个新的 jwt 给客户端。只是这种方案太暴力了，会带来的性能问题。

2.只要快要过期的时候刷新 jwt

此方案是基于上个方案的改造版，只在前一个 jwt 的最后几分钟返回给客户端一个新的 jwt。这样做，触发刷新 jwt 基本就要看运气了，如果用户恰巧在最后几分钟访问了服务器，触发了刷新，万事大吉。如果用户连续操作了 27 分钟，只有最后的 3 分钟没有操作，导致未刷新 jwt，无疑会令用户抓狂。

3. 完善 refreshToken

借鉴 oauth2 的设计，返回给客户端一个 refreshToken，允许客户端主动刷新 jwt。一般而言，jwt 的过期时间可以设置为数小时，而 refreshToken 的过期时间设置为数天。

4. 使用 Redis 记录独立的过期时间

在 Redis 中单独为每个 jwt 设置了过期时间，每次访问时刷新 jwt 的过期时间，若 jwt 不存在与 Redis 中则认为过期。

5. 如何防止令牌被盗用?

令牌:

```
eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJObzAwMDEiLCJpYXQiOjE1NjkxNTg4MDgsInN1YiI6IuS4u-mimCIsImlzcyI6Ind3dy5pdGhlaW1hLmNvbSIsImV4cCI6MTU2OTE1O
```

```
DgyMywiYWRkcmVzcyI6IuS4reWbvSIsIm1vbmV5IjoxMDAsImFnZSI6MjV9.lkaOahB
KcQ-c8sBPp1Op-siL2k6RiwcEiR17JsZDw98
```

如果令牌被盗，只要该令牌不过期，任何服务都可以使用该令牌，有可能引起不安全操作。我们可以在每次生成令牌的时候，将用户的客户端信息获取，同时获取用户的 IP 信息，然后将 IP 和客户端信息以 MD5 的方式进行加密，放到令牌中作为载荷的一部分，用户每次访问微服务的时候，要先经过微服务网关，此时我们也获取用户客户端信息，同时获取用户的 IP，然后将 IP 和客户端信息拼接到一起再进行 MD5 加密，如果 MD5 值和载荷不一致，说明用户的 IP 发生了变化或者终端发生了变化，有被盗的嫌疑，此时不让访问即可。这种解决方案比较有效。

当然，还有一些别的方法也能减少令牌被盗用的概率，例如设置令牌超时时间不要太长。

六. 探花交友

一、项目介绍

探花交友是一个陌生人在线交友平台，在该平台中可以搜索附近的人，查看好友动态，平台提供大数据分析，通过后台推荐系统帮我们快速匹配自己的“意中人”。项目主要分为交友，圈子，消息，视频，我的五大块。项目的亮点是使用 MongoDB 技术实现海量数据的存储；使用 FastDFS 实现小视频的上传和下载；使用百度 AI 技术实现非人脸无法上传。项目采用前后端分离的方式开发，采用的是 YApi 在线文档进行数据交互管理。

1.1 开发技术

前端：

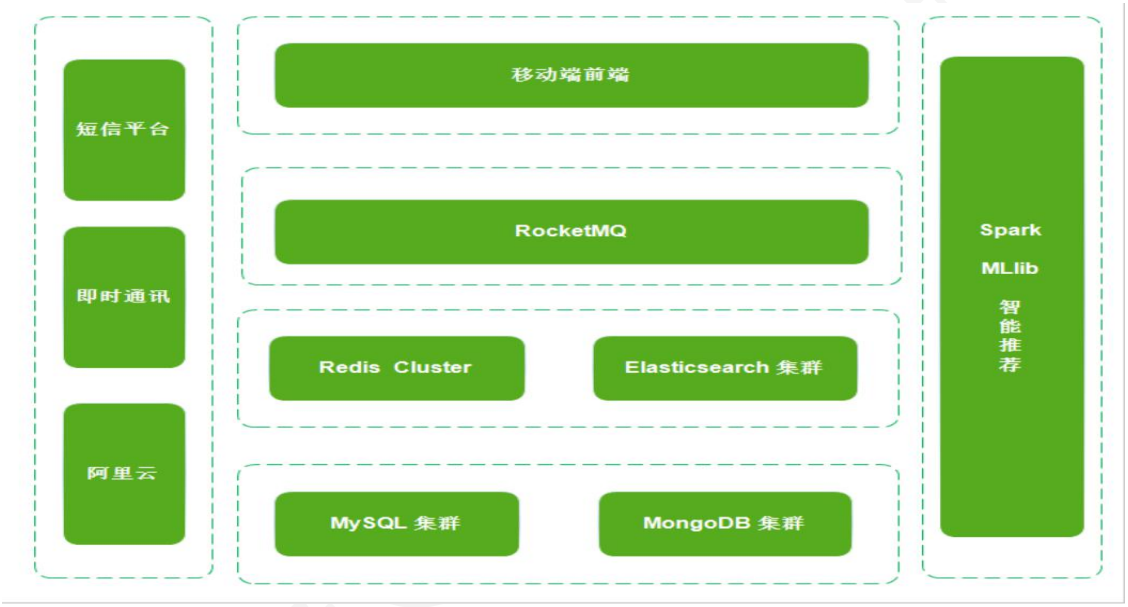
- flutter + android + 环信 SDK + redux + shared_preferences + connectivity + iconfont + webview + sqflite

后端：

- Spring Boot + SpringMVC + Mybatis + MybatisPlus + Dubbo
- MongoDB geo 实现地理位置查询

- MongoDB 实现海量数据的存储
- Redis 数据的缓存
- Spark + MLlib 实现智能推荐
- 第三方服务 环信即时通讯
- 第三方服务 阿里云 OSS 、 短信服务

1.2 技术架构



1.3 开发方式

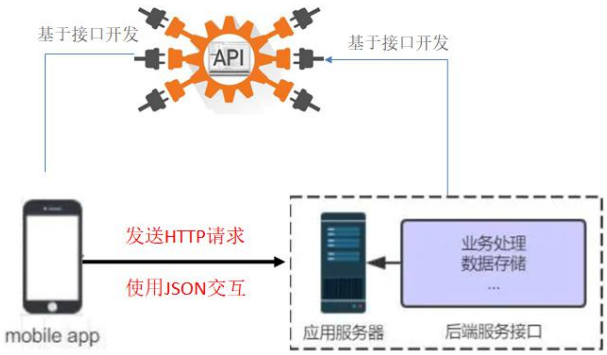
探花交友项目采用前后端分离的方式开发，就是前端由前端团队负责开发，后端负责接口的开发，这种开发方式有 2 点好处：

扬长避短，每个团队做自己擅长的事情

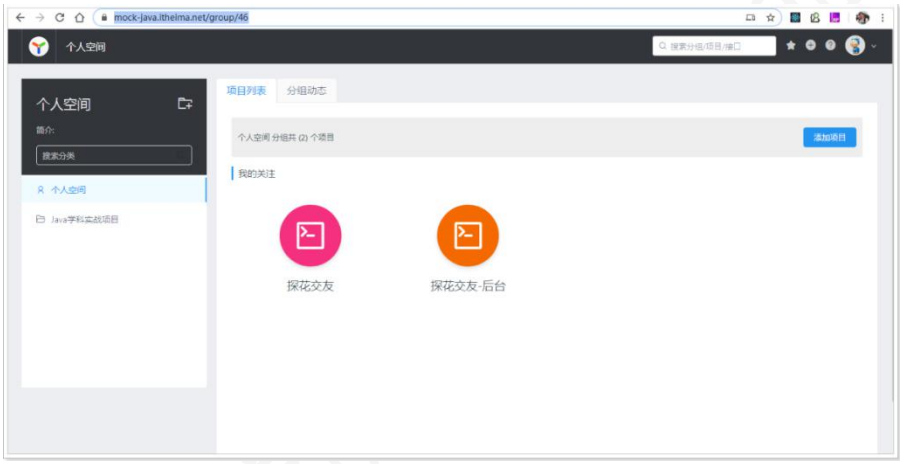
前后端可以并行开发，事先约定好接口地址以及各种参数、响应数据结构等。

前后端分离

- 前后端分离开发基于HTTP+JSON交互
- 通过接口文档（API文档）定义规范
- 后端按照文档定义请求及响应数据
- 前端按照文档发送请求解析响应



我们在项目中使用 YApi 在线接口文档来进行项目开发。YApi 是一个开源的接口定义、管理、提供 mock 数据的管理平台。具体操作如下：



接口定义：

二、功能介绍

探花交友平台，涵盖了主流常用的一些功能，如：交友、聊天、动态等。具体功能如列表所述：

2.1 功能列表

功能	说明	备注
注册、登录	用户无需单独注册，直接通过手机号登录即可	首次登录成功后需要完善个人信息
交友	主要功能有：测灵魂、桃花传音、搜附近、探花等	通过不同的趣味模式让我们找到自己更熟悉的人
圈子	类似微信朋友圈，用户可以发动态、查看好友动态等	
消息	通知类消息 + 即时通讯消息	
小视频	类似抖音，用户可以发小视频，评论等	显示小视频列表需要进行推荐算法计算后进行展现。
我的	我的动态、关注数、粉丝数、通用设置等	

2.2 注册登录

用户通过手机验证码进行登录,如果是第一次登录则需要完善个人信息,在上传图片时，需要对上传的图片做人像的校验，防止用户上传非人像的图片作为头像。流程完成后，则登

录成功。

2.2.1 技术要点：

阿里云通信 + Redis + JWT + OSS 云存储 + 百度 AI

2.2.2 涉及的表：

tb_user（用户表）
tb_info（用户详细信息表）

2.2.3 实现分析：

实现用户登录我们要处理四个接口来进行实现，分别是：

发送手机验证码(阿里云通信)

- 阿里云平台短信发送

校验用户登录(Redis)

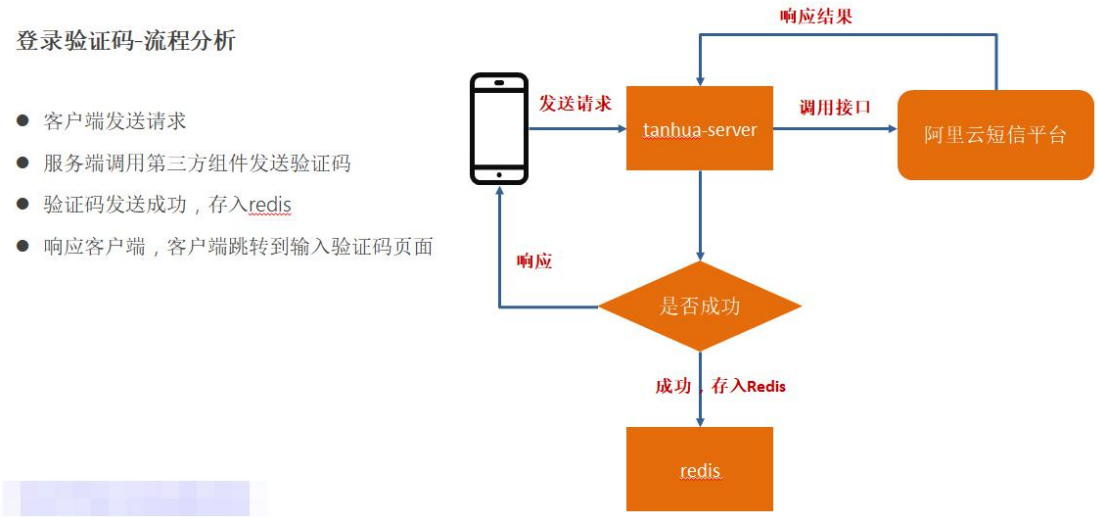
- 后台需要验证手机号与验证码是否正确
- 校验成功之后，需要按照 JWT 规范进行返回响应

首次登录完善个人信息(百度 AI + OSS)

校验 token 是否有效(Redis + JWT)

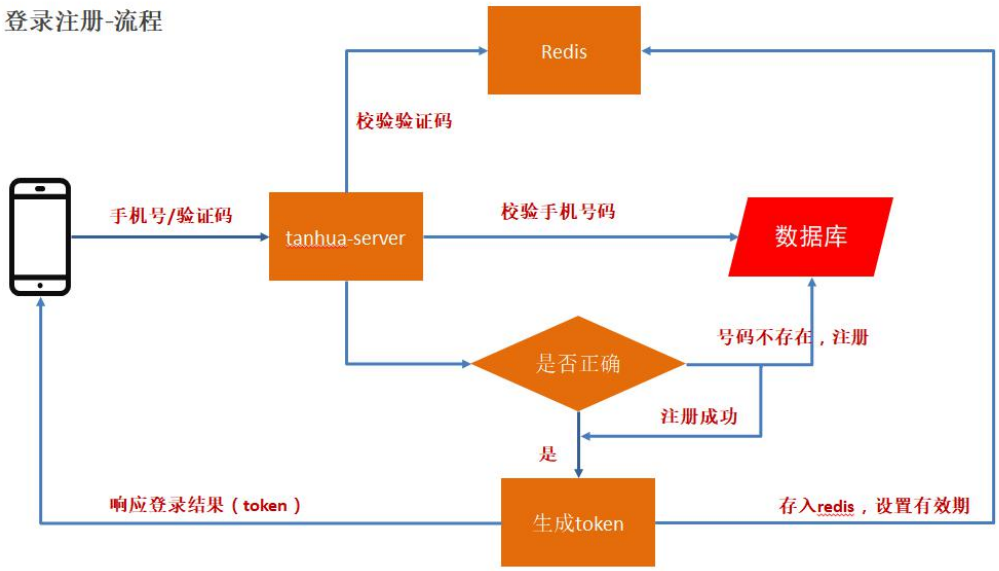
- 校验存储到 Redis 中的 token 是否有效

2.2.3.1 发送手机验证码流程：



用户在客户端输入手机号，点击发送验证码，向后台 tanhua-server 进行请求，调用后台阿里云短信平台进行短信发送。tanhua-server 服务会会携带用户给发过来的两个参数(用户的手机号和后台生成的随机 6 位数)通过调用阿里云通信第三方平台进行接口的响应。如果阿里云短信平台在接受响应成功后,会把验证码发送给客户，并把这个验证码保存到 Redis 中，并给这个验证码设置有效期为 5 分钟。

2.2.3.2 校验用户登录流程：



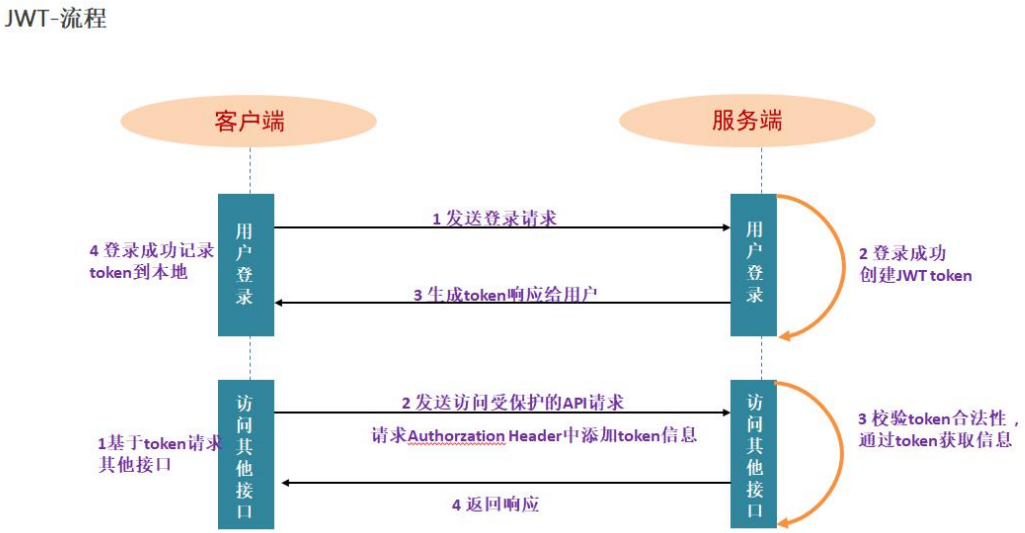
当用户在手机上接受到验证码之后，开始输入验证码校验。当用户输入完 6 位验证码之后，会自动触发后台 tanhua-server 服务进行校验功能。对用户传输过来的手机号和验证码进行验证。如果校验通过会根据客户的手机号生成一个 token，给客户返回一份，方便用户在 App 上访问其他微服务的时候进行身份的识别。同时在通过 RocketMQ 响应给 JWT 的鉴权中心通知其他微服务。在用户通过校验之后，我们会做 3 个工作：

1. 我们会根据用户的手机号给用户创建一个账户并存入 mysql 的用户表中。

2. 我们会调用 RocketMQtemplate 发送一条消息给消息中间件，用来给后台做 log 日志的统计，便于后台管理系统展示平台用户日活量，注册量等等。

3. 把创建的新用户 id 注册到环信通讯中，来实现即时通信的功能。

2.2.3.3 用户登录



考虑到我们项目采用的是前后端分离架构，我们采用 JWT 生成 token 的方式来给用户进行身份识别，当我们登录成功后我们会根据用户的 id 和手机号生成一个 token 返回给用户并保存到 Redis 中。这样用户在访问受保护的资源的时候，我们会在后台进行校验客户端发过来的 token，如果校验通过则进行放行。

在这块我们考虑到三个问题：

1.是 token 续签；2. token 的安全性； 3. 如何统一处理 token；

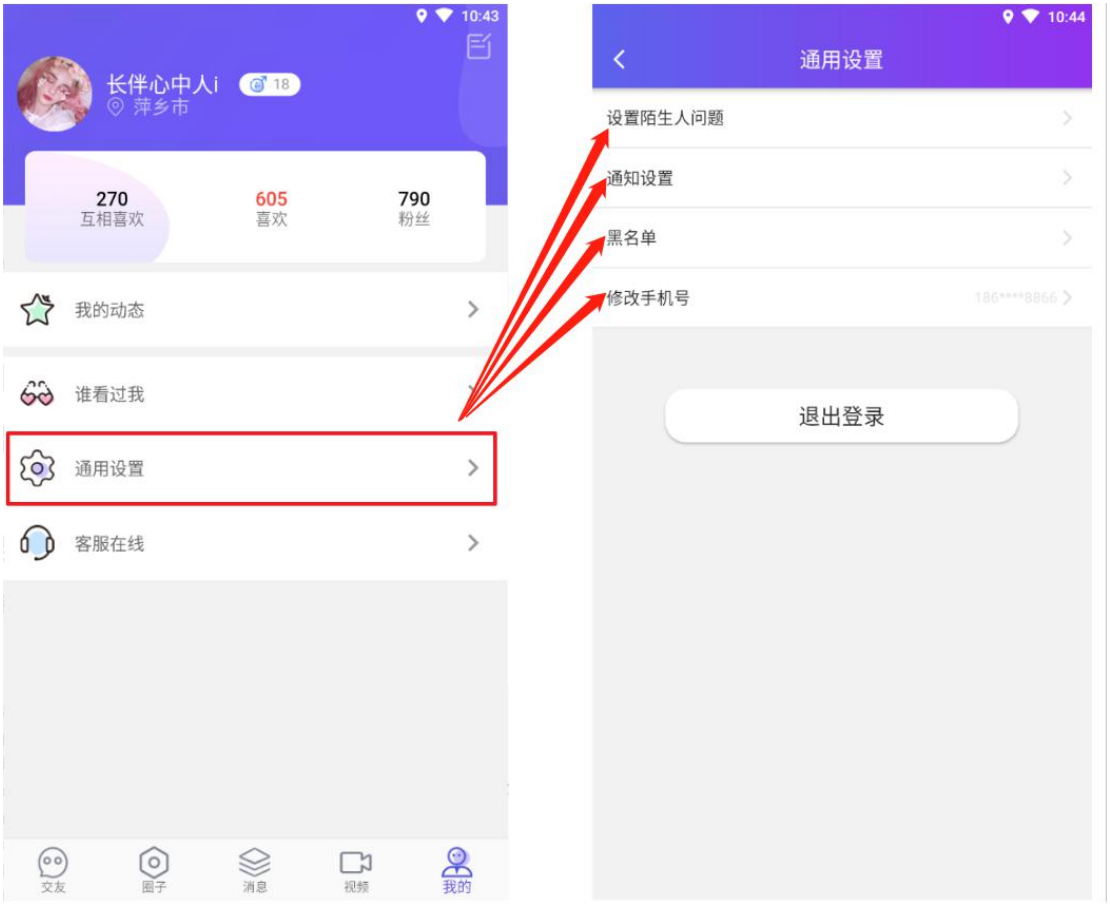
在解决 token 续签的问题上，我们这里采取的是在请求头获取 token 时判断 token 是否存在，如果不存在就创建 token 并保存到 Redis 中，如果存在我们就重新从 Redis 中获取 token 并进行续签 (Duration.ofHours(1))。在安全问题上，我们这里采用加密加盐 (SignatureAlgorithm.HS256,secret)的方式来解决。为了实现 token 统一校验，我们这里采用的是 springmvc 拦截机制+Threadlocal 局部线程的方式来解决。这里我们自定义了一个 TokenInterceptor 拦截器，实现 preHandle() 方法，这样的话就可以在用户请求进入 controller 层之前进行拦截，通过 token 获取了用户对象，并将用户对象存储到 Threadlocal 中，这样就实现了 token 统一校验的功能。具体的实现过程如下图：

2.2.3.4 首次登录完善个人信息

用户在完成登录操作之后，如果是第一次登录，用户会跳到完善个人信息页面，这块我们需要输入用户昵称，生日，所在地，设置头像。其中，头像数据需要做图片上传，这里采用阿里云的 OSS 服务作为我们的图片存储服务器。并且对头像要做人脸识别，非人脸照片

不得上传，这里我们调用的是百度 AI 的人脸识别接口进行校验。我们需要在本地创建一个 ossTemplate 模板和 aipFaceTemplate 模板，把百度 AI 和阿里 OSS 给我们提供的 SDK 进行修改。我会在 yml 中创建以上两个模板所需要的参数。

2.3 通用功能的实现



2.3.1 涉及的表：

tb_settings（通用设置）
tb_question（陌生人问题表）
tb_black_list（黑名单）

2.3.2 实现分析：

通用设置，包含探花交友 APP 基本的软件设置功能。主要功能有设置陌生人问题，通用设置，黑名单等。设置陌生人问题这块主要为了在添加你的时候需要回答问题，根据用户

回答的问题，进行确认关系。我们这块主要做问题的添加和问题查看。这块就是根据用户的 id 查询用户的问题，问题查看就是如果没有问题那么就设置问题，如果有就 update 重新更新一下。通用设置这块我们主要做的接口是给用户推送喜欢通知，推送评论通知，推送平台公告通知，这块我们根据用户的 id 去通用设置表里边保存或者更新用户通知情况。黑名单这块我们做的是黑名单列表的查询和黑名单的移除。主要查询用户表和黑名单表，如果黑名单移除，那么直接在黑名单里边根据用户的 id 删除。这块主要做了黑名单列表分页的问题，因为随着我们往下拉取，后台会根据用户的 id 动态展示黑名单列表。这块就使用到 mybatisPlus 的分页 Ipage。通过创建 Page 对象，传入当前页，和每页条数，调用 findPage(page，XXX 条件)来实现；

2.4 今日佳人

在用户登录成功后，就会进入首页，首页中有今日佳人、推荐好友、探花、搜附近等功能。今日佳人，是根据用户的行为数据统计出来匹配值最高的人。缘分值的计算是由用户的行为进行打分，如：点击、点赞、评论、学历、婚姻状态等信息组合而成的。

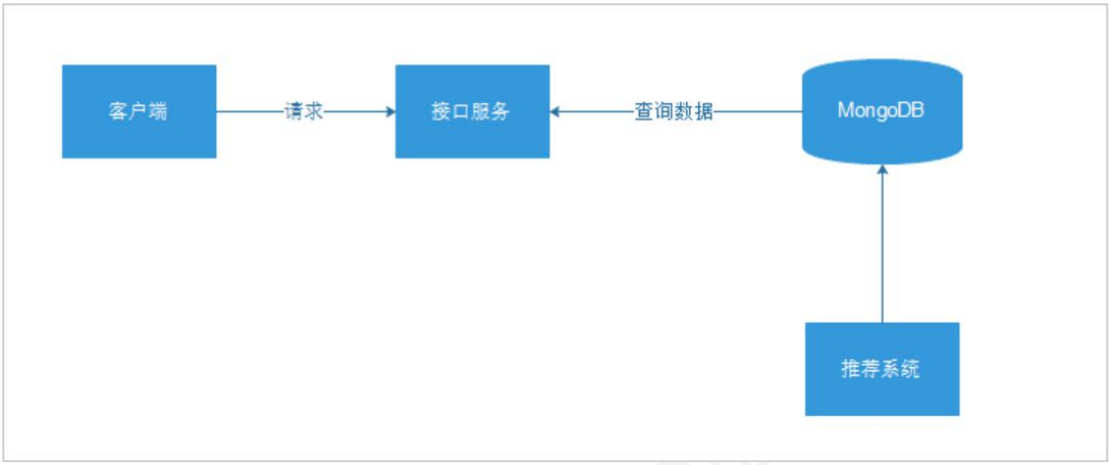
2.4.1 技术要点：

MongoDB

2.4.2 涉及的表：

recommend_user(推荐用户表)

2.4.3 实现流程：

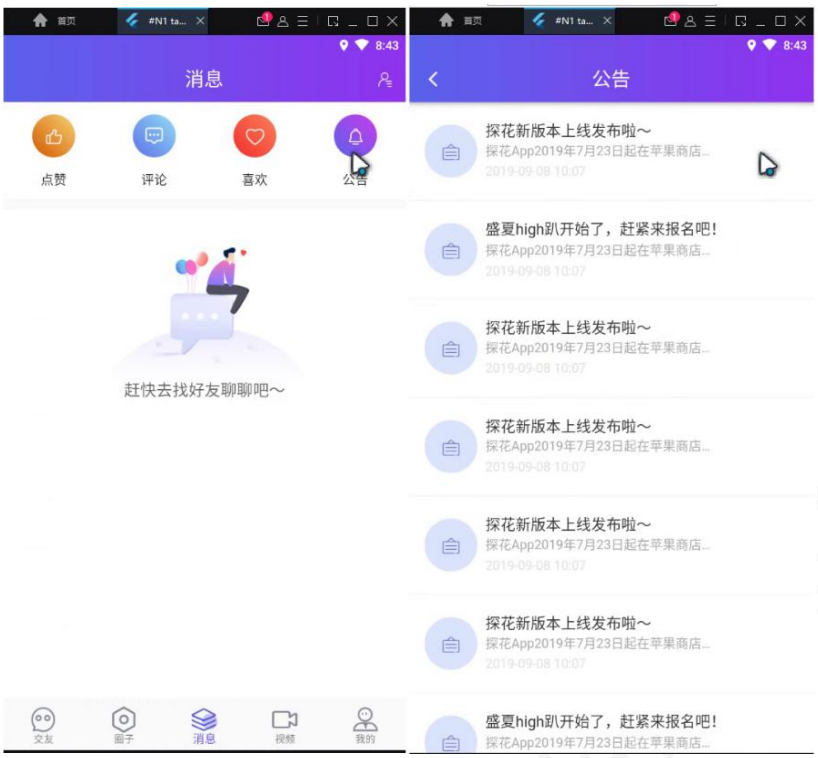


2.4.4 实现分析：

当我们进入首页之后，上边会有一个滑动区域，有今日佳人，探花，公告等模块。我在这块负责今日佳人模块实现，主要根据用户的 id 查找对应的推荐佳人表，根据缘分值进行排序，获取第一个进行展示。缘分值是我们通过用户平时浏览平台动态和浏览小视频，对浏览的内容进行点赞、评论的情况以及用户本身的学历、婚姻状态等信息进行计算的。我们会把计算的最终值存储到 MongoDB 中方便用户进行快速查询。当用户对圈子，视频的动态执行操作时，如：发布、浏览、点赞、喜欢等，平台就会通过 RocketMQ 给推荐系统发送消息。推荐系统接收消息，并且处理消息数据，处理之后将结果数据写入到 MongoDB 中，Spark 系统拉取数据，然后进行推荐计算，计算之后的结果数据写入到 Redis 中，为每个用户都进行个性化推荐，如果有用户没有数据的，查询 MongoDB 中的默认数据。我们这块动态计分规则是：浏览 +1，点赞 +5，喜欢 +8，评论 + 10，文字长度：50 以内 1 分，50~100 之间 2 分，100 以上 3 分。

2.5 公告

主要用于接收平台发布的一些消息和福利，比如：新增功能通知，活动通知，版本迭代通知。



2.5.1 涉及的表：

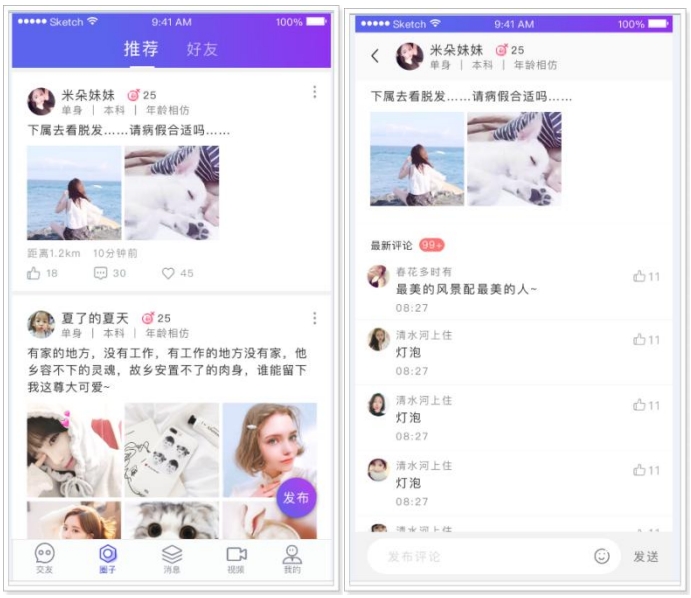
tb_announcement (公告表)

2.5.2 实现分析：

这块我们主要做两个事情，一个是公告查询分页，一个是公告对象转换 VO。分页的话我们这里使用的是 MybatisPlus 的分页功能，公告对象转化 VO 主要是为了解耦和不做代码污染，我们通过创建一个 list 的方式把公告对象遍历到我们创建的 AnnouncementVO 对象中，在这里我们要做一个时间的校验，如果创建时间不存在我们需要给他生成个时间在进行加入。

2.6 圈子

类似微信朋友圈，我们可以发布图片，文字；可以对圈子里边的内容进行点赞，评论，喜欢操作。和微信朋友圈不一样的功能是微信只能查看好友的圈子，但是我们这个平台圈子推荐的不仅仅是好友还有系统智能推荐用户点赞喜欢的好友。



2.6.1 技术要点：

MongoDB + Redis + MongoDB geo + rocketMQ + 华为云内容审核 + Spark + Mllib

2.6.2 涉及的表：

quanzi_publish（发布表）
quanzi_mine_{userId}（相册表）
quanzi_friend_{userId}（时间线表）
quanzi_comment（评论表）
tanhua_users（好友表）
tb_user（用户表）

2.6.3 实现分析：

2.6.3.1 通过页面分析，剖析如下：

1、推荐频道为根据问卷及喜好推荐相似用户动态(**RocketMQ+Spark + MLlib**)

2、显示内容为用户头像、用户昵称、用户性别、用户年龄、用户标签和用户发布动态

3、图片最多不超过 6 张

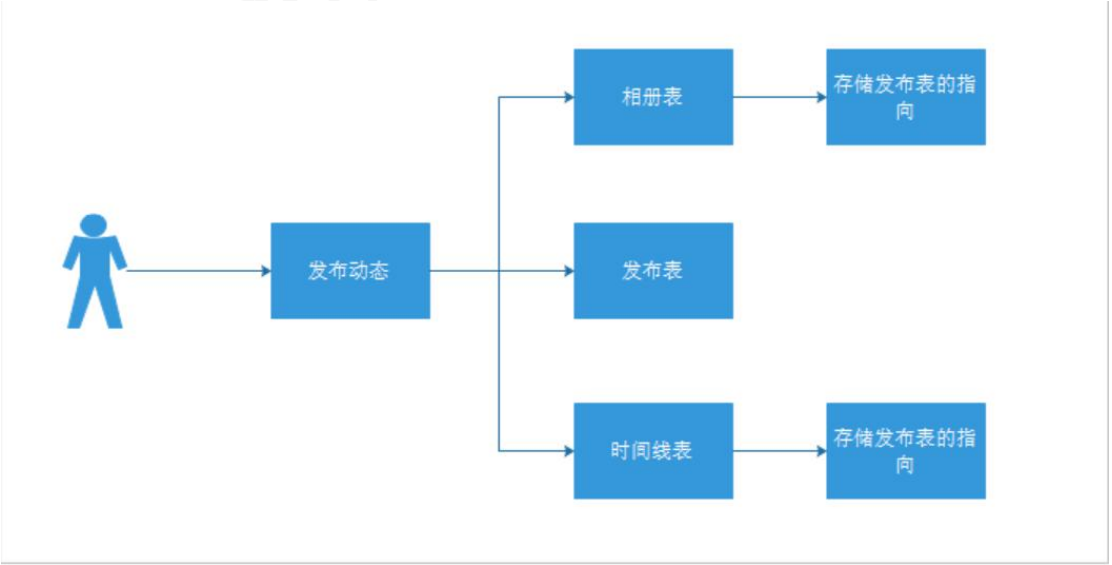
4、动态下方显示发布时间距离当时时间，例如 10 分钟前、3 小时前、2 天前，显示时间进行取整。

5、动态下方显示距离为发布动态地与本地距离(**MongoDB Geo**)

6、显示用户浏览量

7、显示点赞数、评论数 转发数

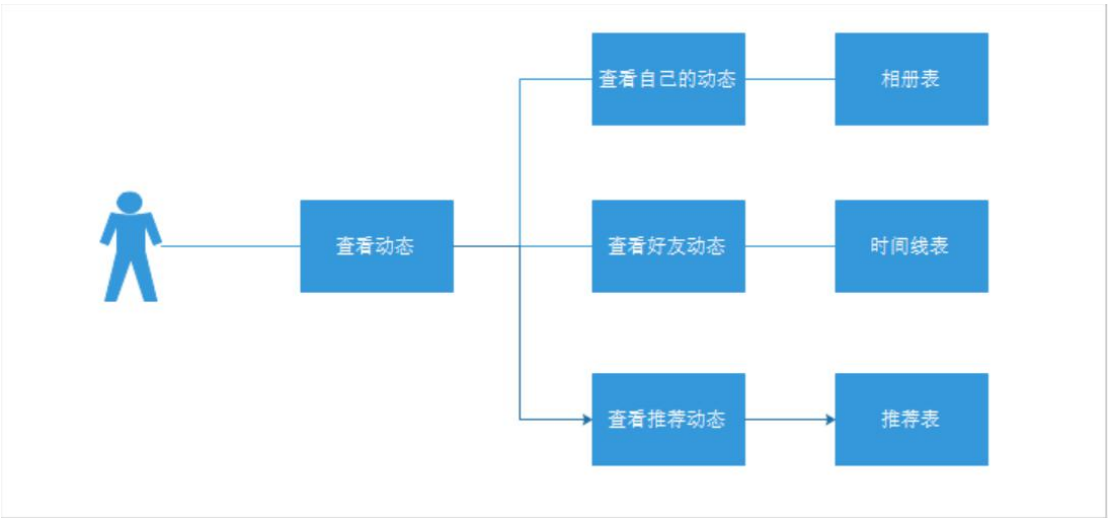
2.6.3.2 发布流程：



- 用户发布动态，首先将动态内容写入到发布表。
- 然后，将发布的指向写入到自己的相册表中。

- 最后，将发布的指向写入到好友的时间线中。

2.6.3.3 查看流程：



- 用户查看动态，如果查看自己的动态，直接查询相册表即可
- 如果查看好友动态，查询时间线表即可
- 如果查看推荐动态，查看推荐表即可

由此可见，查看动态的成本较低，可以快速的查询到动态数据。

2.6.3.4 实现话术：

我负责项目中的圈子功能的实现，类似微信的朋友圈，基本的功能为：发布动态、浏览好友动态、浏览推荐动态、点赞、评论、喜欢等功能。我们这里采用 spark + mllib 实现智能推荐，这样的话就会根据我们自身的条件，喜好进行匹配推送用户感兴趣的动态。针对用户上传的动态我们采用 rocketMQ 发送消息调用华为云内容审核进行自动处理，考虑到需要处理海量的动态信息数据，我们这里选用了 MongoDB+Redis 的形式来进行数据的存储，其中 MongoDB 负责存储，Redis 负责缓存。

首先是查询好友动态接口的实现，用户点击好友动态之后，服务器会从 ThreadLocal 中获取用户 id，根据用户 id 从该用户对应的好友动态时间线表中查询出所有好友动态 id 以及对应的好友 id，再根据动态 id 从总动态表查询动态详细信息，根据动态发布的时间倒序排序。再根据好友 id 从用户详情表中查询好友用户详细信息，再将这些信息构造成 vo 对象返回给前端进行分页展示。

接下来说一下用户发布动态接口的实现，用户在客户端写好文本，上传好图片，确定好位置点击发布按钮之后，请求会携带一系列参数，文本内容，图片文件，地理位置的经纬度等，后端代码先创建对象，设置相应的属性，保存到 MongoDB 动态表中，此时动态的状态属性默认设置为 0，表示待审核。然后通过 rocketMQ 向消息中间件发送一条消息，包含动态 id，消息消费者接收到消息之后会调用华为云的内容审核服务，对文字信息和图片信息进行审核，审核通过后将动态的状态属性改为 1，表示已通过，不然就改为 2，表示驳

回。然后前端在展示动态的时候只会查询状态为已通过的动态。动态发布成功之后也会像消息中间件发送消息，消息内容包括动态 id（objectId 类型）、用户 id、操作类型、Long 类型的 pid。消息消费者会以此生成一条该用户对该动态的一个评分记录保存到 MongoDB 的动态评分表中，作为大数据推荐的依据。

然后说一下用户对动态的一系列操作的接口的实现。我们是将用户的点赞、评论、喜欢操作记录到 MongoDB 的一个 comment 表中，用 type 字段来区分具体的操作类型，比如 1 代表点赞，2 代表评论，3 代表喜欢。用户进行操作之后服务器就会生成相应操作类型的 comment 记录保存到 MongoDB 的 comment 表中。保存成功后向消息中间件发送消息，消息内容包括动态 id（objectId 类型）、用户 id、操作类型、Long 类型的 pid。消息消费者会以此生成一条该用户对该动态的一个评分记录保存到 MongoDB 的动态评分表中，作为大数据推荐的依据。

最后是推荐频道的动态展示接口的实现，服务器使用 Spark + Mllib 技术，根据用户的动态评分表，基于用户 userCF 算法进行推荐，将推荐的结果以 pid 组成的字符串形式存入 Redis。用户进入推荐频道之后，服务器去查找 Redis 有没有推荐结果，如果没有的话，就从 MongoDB 中查询默认的推荐动态数据展示，如果有推荐结果，就获取字符串，转换成 pid 的数组，再根据这些 pid 从 MongoDB 的总动态表中查找对应动态信息，构造 vo 对象的分页结果，返回给前端展示给用户。

2.7 小视频

用户可以上传小视频，也可以查看小视频列表，并且可以对推荐的视频进行点赞操作。通过页面分析我们需要获取视频展示出来的视频封页，好友的头像，昵称，点赞数量。

2.7.1 技术要点：

FastDFS + CDN

2.7.2 涉及的表：

video(小视频表)
quanzi_comment（评论表）
quanzi_publish（发布表）
tb_user（用户表）

2.7.3 实现分析：

小视频功能类似于抖音、快手小视频的应用，用户可以上传小视频进行分享，也可以浏览查看别人分享的视频，并且可以对视频评论和点赞操作。对于存储而言，小视频的存储量以及容量都是非常巨大的，所以我们选择自己搭建分布式存储系统 FastDFS 进行存储。对于推荐算法，我们将采用多种权重的计算方式进行计算，对于加载速度，除了提升服务器带宽外可以通过 CDN 的方式进行加速。发布视频的流程是：客户通过点击发布视频，会向后台传递 3 个数据，一个是视频封面图片，一个是视频，一个是文字。我们在后台会把视频封面图片保存到阿里的 OSS 图片存储服务器上会给我们返回一个图片的存储链接，把视频上传到本地部署的 FastDFS 上边会给我们返回一个视频的存储链接，之后我们这边构造一个 Video 对象，把需要的参数(文字，图片 URL，视频 URL，用户 ID)通过调用 VideoAPI 进行保存到 MongoDB 中，之后构造返回值给用户进行显示发布成功。接下来用户会跳转到小视频列表页面。分页查询视频列表是根据后台推荐系统进行智能推荐的，这个根据用户浏览视频的时候点赞，喜欢，评论的情况进行分数积累，把对应的分数值存储到推荐表中，当用户下拉刷新的时候，会从 MongoDB 中查询出推荐的视频。

2.8 探花

左划喜欢，右划不喜欢，每天限量不超过 50 个，开通会员可增加限额。双方互相喜欢则配对成功，可以互加好友。

通过页面分析：我们需要展示用户的信息，包括：头像，昵称，年龄，是否单身，学历等。

实现：数据来源推荐系统计算后的结果。

2.8.1 技术要点：

MongoDB + OSS + spark

2.8.2 涉及的表：

recommend_quanzi（推荐表）
tanhua_users（好友表）
tb_user（用户表）

2.8.3 实现分析：

当用户的通过点击探花进入时，会根据用户的 id 去 MongoDB 中查询 10 条推荐的好友信息，推荐依据是来源于推荐系统通过 spak + Mallib 根据用户平时喜欢，点赞，评论进行生成。当用户选择喜欢的时候我们会把用户和喜欢好友的 id 写入到环信通讯中，再重新存入到 MongoDB 中，方便用户做选择。当用户选择不喜欢时候，我们要清除当前数据，不予以推荐。我们这块实现主要做了三个接口，分别是展示卡片列表接口，右滑喜欢接口，左滑不喜欢接口。展示列表我们根据获取的用户 id 去推荐表中查询默认的推荐好友。如果选择不喜欢会根据当前用户的 id 和推荐人的 id 调用接口将该用户的从推荐表中删除，如果选择喜欢，会根据当前用户的 id 和推荐人的 id 通过调用之前的一个喜欢接口生成一个实例保存到 mongoDB 数据库中，并且判断所喜欢的用户是否为自己的粉丝，如果是的话则将

两人添加到好友关系表中，并且将环信中的两人绑定为好友关系，这样在好友列表中可以查询到对方并且可以进行聊天了。

2.9 测试灵魂

测灵魂是我们项目的一大亮点。平台会提供对应的初级灵魂，中级灵魂，高级灵魂对应的答卷。通过用户答题，我们会在后台进行用户行为数据分析，给用户发送一份信息报告，同时也会推荐和用户本人灵魂属性相符的人。我们这块会为每个灵魂等级设置 10 个问题，测试题为顺序回答，回答出初级题，我们才可以点击中级灵魂测试。

2.9.1 技术要点：

MySQL + MongoDB

2.9.2 涉及的表：

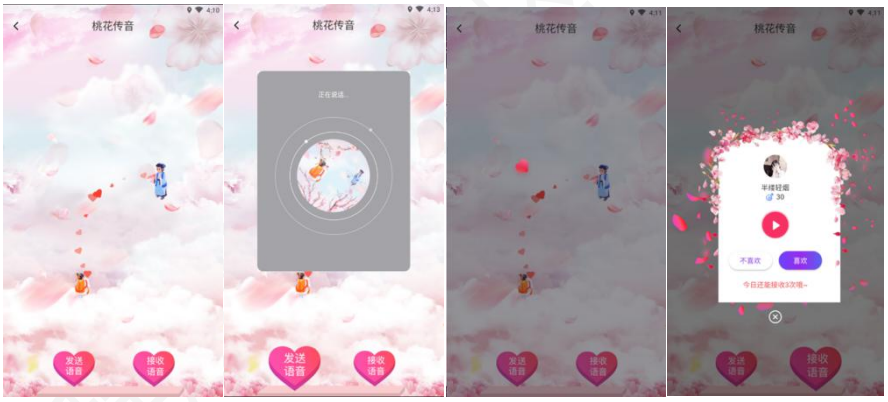
tb_questionnaire(调查问卷)
tb_test_question(问卷问题)
tb_option(问题选项)
tb_conclusion(性格信息) **Mysql 中的表**
testSoul_Questionnaire (调查问卷) **MongoDB 中的表**
testsoul_user_score(用户得分)

testsoul_report

2.9.3 实现分析：

灵魂测试是通过用户答题的方式来进行分类，寻找同一类型或者内心深处的朋友。我们这块主要采用的技术有：MongoDB +Sprak+Mllib 当用户点击灵魂测试，我们会去后台加载用户的灵魂信息。如果是新用户我们会停留在初级灵魂题，后边的中级灵魂，高级灵魂则需解锁上一级才能进行答题。如果是老用户我们会加载用户之前测试灵魂的结果进行展示。如果之前他测评通过的是高级灵魂，那么就可以查看初级，中级，高级灵魂的测评。每一级的测评灵魂题都会有 10 道题，测评结束之后会根据用户的回答生成一个测试报告，通过点击我们可以查看属于自己的唯一一份灵魂报告。生成的报告主要根据用户的属性外向，判断，抽象，理性组成。我们可以把自己的报告进行分享或者选择重做。当进行报告查看时候我们还可以查看系统给我推荐的测试结果相同的好友。通过点击好友头像，我们可以通过回答陌生人问题来进行确认好友，如果对方回复并通过就可以进行正常的交友啦。

2.10 桃花传音



2.10.1 技术要点：

MongoDB + FastDFS + Quartz

2.10.2 涉及的表：

user_sound（用户语音）
user_soundTime（语音剩余次数）
tb_user（用户表）

2.10.3 实现分析：

语音匹配类似漂流瓶类似，可以发送匿名语音，可以收听匿名语音。收听完匿名语音可以选择喜欢或不喜欢。如果双方互相喜欢那么可以配对成功互相关注，一个人每天只能接收 8 次，自己接收过语音，其他人不能再使用。如果双方都点击喜欢，就可以添加为好友。这块我们采用的是 MongoDB 做数据存储，使用 Dubbo 做分布式服务的调用，使用 FastDFS 存储语音。用户可以发布一段语音文件，上传到 fastDFS 中，并且将返回的地址值，与当前用户的一些信息封装成实例保存到 mongoDB 中，其他用户会进行接收操作时，随机在数据库中获取一条语音信息，然后进行选择打招呼，和扔掉，打招呼的话就可以进行调用环信的发送一条信息，如果对方回复的话则成为好友，同时该语音文件会在数据库中删除，如果扔掉的话则不会再数据库中消失，继续等待被获取。

七、黑马头条

一、项目介绍

1.1 项目背景

随着智能手机的普及，人们更加习惯于通过手机来看新闻。由于生活节奏的加快，很多人只能利用碎片时间来获取信息，因此，对于移动资讯客户端的需求也越来越高。黑马头条项目正是在这样背景下开发出来。黑马头条项目采用当下火热的微服务+大数据技术架构实现。本项目主要着手于获取最新最热新闻资讯，通过大数据分析用户喜好精确推送咨询新闻。

1.2 项目概述

黑马头条项目是对在线教育平台业务进行大数据统计分析的系统。碎片化、切换频繁、社交化和个性化现如今成为人们阅读行为的标签。黑马头条对海量信息进行搜集，通过系统计算分类，分析用户的兴趣进行推送从而满足用户的需求。

1.3 需求说明

1.4 功能架构图

1.5 APP 主要功能大纲

- 频道栏：用户可以通过此功能添加自己感兴趣的频道，在添加标签时，系统可依据用户喜好进行推荐
- 文章列表：需要显示文章标题、文章图片、评论数等信息，且需要监控文章是否在 APP 端展现的行为
- 搜索文章：联想用户想搜索的内容，并记录用户的历史搜索信息
- 查看文章：用户点击文章进入查看文章页面，在此页面上可进行点赞、评论、不喜欢、分享等操作；除此之外还需要收集用户查看文章的时间，是否看我等行为信息
- 注册登录：登录时，验证内容为手机号登录/注册，通过手机号验证码进行登录/注册，首次登录用户自动注册账号。
- 实名认证：用户可以进行身份证认证和实名认证，实名认证之后即可成为自媒体人，在平台上发布文章
- 个人中心：用户可以在其个人中心查看收藏、关注的人、以及系统设置等功能

1.6 自媒体端功能大纲

- 内容管理：自媒体用户管理文章页面，可以根据条件进行筛选，文章包含草稿、已发布、未通过、已撤回状态。用户可以对文章进行修改，上/下架操作、查看文章状态等操作
- 评论管理：管理文章评论页面，显示用户已发布的全部文章，可以查看文章总评论数和粉丝评论数，可以对文章进行关闭评论等操作
- 素材管理：管理自媒体文章发布的图片，便于用户发布带有多张图片的文章
- 图文数据：自媒体人发布文章的数据：阅读数、评论数、收藏量、转发量，用户可以查看对应文章的阅读数据

- 粉丝画像：内容包括：粉丝性别分布、粉丝年龄分布、粉丝终端分布、粉丝喜欢分类分布

1.7 平台管理端功能大纲

- 用户管理：系统后台用来维护用户信息，可以对用户进行增删改查操作，对于违规用户可以进行冻结操
- 用户审核：管理员审核用户信息页面，用户审核分为身份审核和实名审核，身份审核是对用户的身份信息进行审核，包括但不限于工作信息、资质信息、经历信息等；实名认证是对用户实名身份进行认证
- 内容管理：管理员查询现有文章，并对文章进行新增、删除、修改、置顶等操作
- 内容审核：管理员审核自媒体人发布的内容，包括但不限于文章文字、图片、敏感信息等
- 频道管理：管理频道分类界面，可以新增频道，查看频道，新增或修改频道关联的标签
- 网站统计：统计内容包括：日活用户、访问量、新增用户、访问量趋势、热门搜索、用户地区分布等数据
- 内容统计：统计内容包括：文章采集量、发布量、阅读量、阅读时间、评论量、转发量、图片量等数据
- 权限管理：超级管理员对后台管理员账号进行新增或删除角色操作

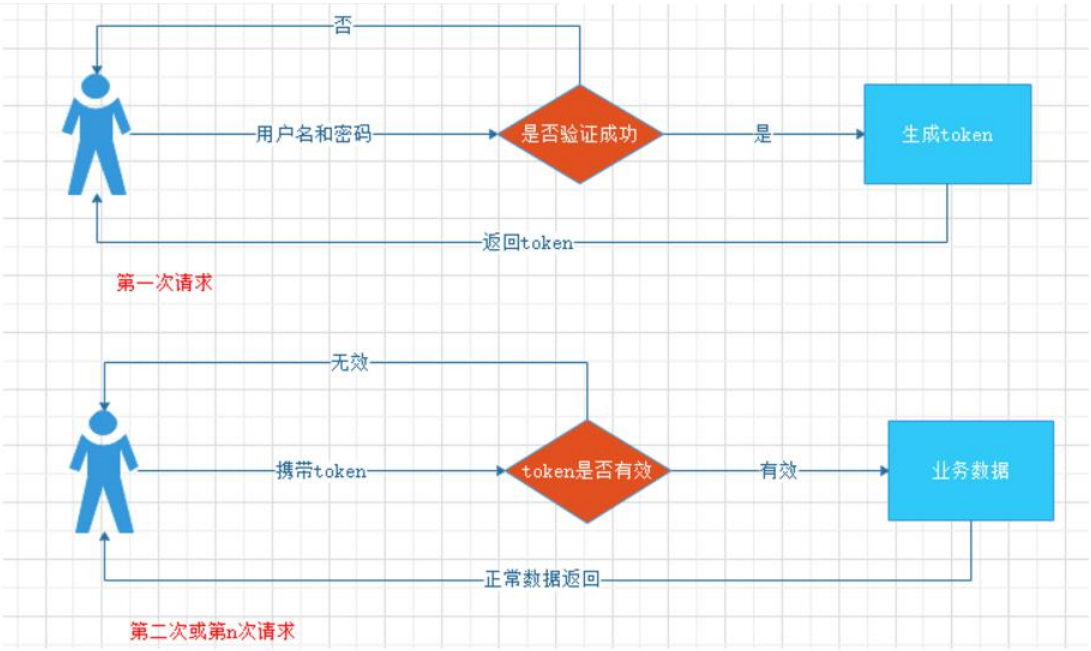
1.8 其它需求

1.9 交互需求

二．功能介绍

随着 Restful API、微服务的兴起，基于 Token 的认证现在已经越来越普遍。基于 token 的用户认证是一种**服务端无状态**的认证方式，所谓服务端无状态指的 token 本身包含登录用户所有的相关数据，而客户端在认证后的每次请求都会携带 token，因此服务器端无需存放 token 数据。

当用户认证后，服务端生成一个 token 发给客户端，客户端可以放到 cookie 或 localStorage 等存储中，每次请求时带上 token，服务端收到 token 通过验证后即可确认用户身份。



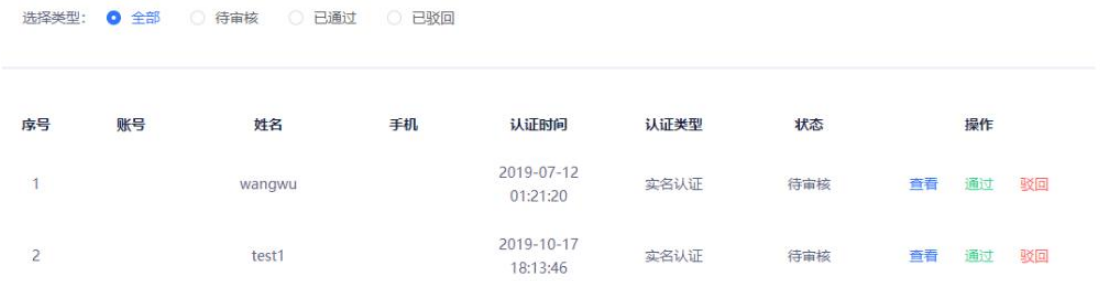
2.1 admin 端

2.1.1 登录实现

根据用户名和密码登录，验证用户名和密码不能为空。首先查询该用户是否存在。根据名称查询，判断，如果用户是空，数据不存在。然后进行密码比对，与数据库里的信息比，如果不一致，返回密码错误，如果密码正确，返回 token，并且返回部分用户信息（隐藏敏感信息）给前端。

2.1.2 App 端用户认证审核

18



流程说明

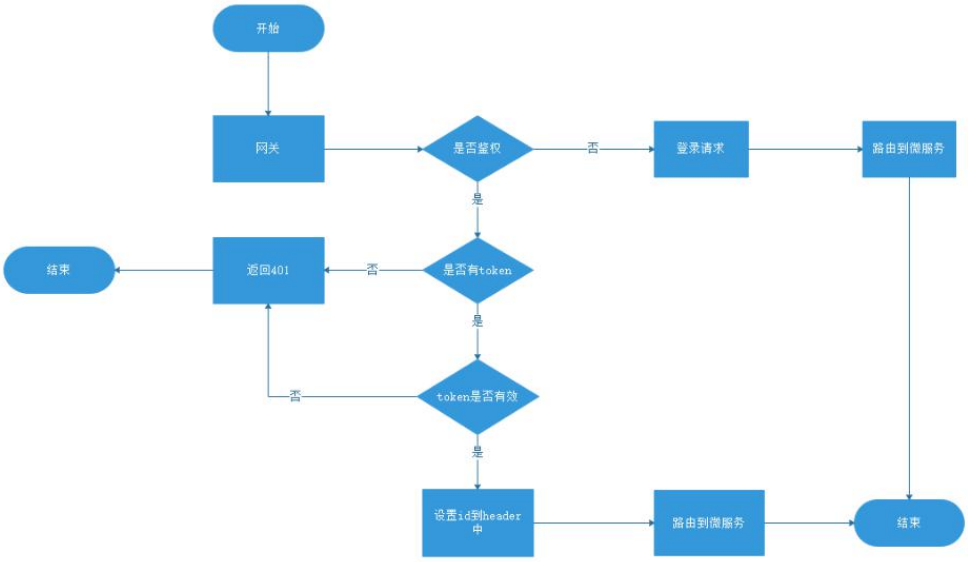
用户审核功能

根据 ID 查询用户，判断审核状态，如果是审核失败，修改用户状态，并且添加失败的原因，并保存并更新后的数据。审核通过，需要创建自媒体用户，构建自媒体用户，通过 Feign 远程调用接口，判断结果。如果创建自媒体用户成功，那么则继续创建作者，如果作者不为空，需要将 authorId 更新到自媒体账号中，修改 apUserRealname 状态，修改 apUser 中的 flag，构建作者信息，通过 Feign 远程调用接口，判断结果。

2.2 app 端

2.2.1 APP 端用户认证

全局过滤器实现 jwt 校验



通过网关进行授权验证

首先判断地址中是否包含登录的地址，如果是登陆的地址，不需要 token,直接放行，如果不是登陆地址，需要验证 token。先获取 token（获取请求头对象），判断 token 是否为空，如果为空返回 401，结束请求，不为空需要验证 token，验证通过，需要将用户 ID 放入到请求头中，供后面的微服务使用。

2.2.2 登录功能

app 端登录-需求分析

- 点击**登录**可以根据 app 端的手机号和密码进行登录
- 点击**不登录，先看看**可以在无登录状态下进入 app

app 端登录-思路分析

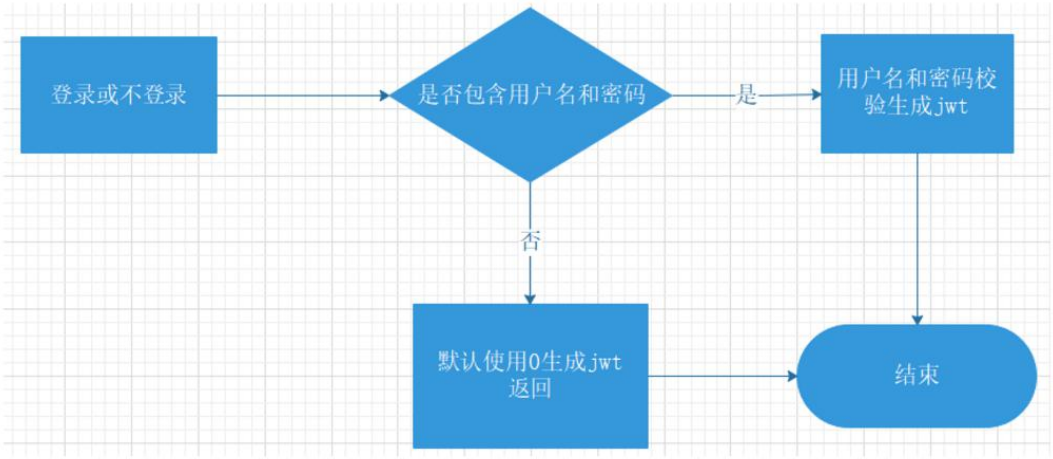
概念介绍：**用户设备**，即当前用户所使用的终端设备。

1，用户点击**登录**

- 用户输入手机号和密码到后端进行校验，校验成功生成 token 返给前端
- 其他请求需要带着 token 到 app 网关去校验 jwt,校验成功，放行

2，用户点击**不登录，先看看**

- 获取用户的设备 id 到后端根据设备 id 生成 token,设置 jwt 存储的 id 为 0
- 其他请求需要带着 token 到 app 网关去校验 jwt,校验成功，放行



端登录-功能实现

用户登陆，查询用户，判断用户的密码是否与传递进来的密码一致，判断用户的密码是否与传递进来的密码一致，密码一致生成 token，发送 token 给前端。

2.2.3 关注作者或取消关注

如上效果：

当前登录后的用户可以关注作者，也可以取消关注作者

思路分析

一个用户关注了作者，作者是由用户实名认证以后开通的作者权限，才有了作者信息，作者肯定是 app 中的一个用户。

从用户的角度出发：一个用户可以关注其他多个作者

从作者的角度出发：一个用户（同是作者）也可以拥有很多个粉丝(用户)

实现步骤：

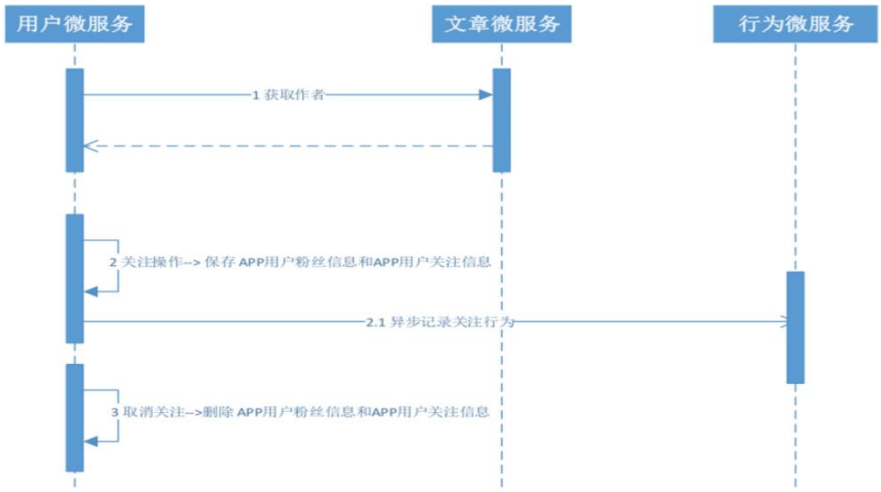
1 前端传递作者 id 获取作者信息，最终获取到作者在当前 app 端的账号 id

2 如果是关注操作，需要保存数据，用户保存关注的作者，作者保存当前的粉丝

2.1 异步记录关注行为（后面开发，为了推荐做准备）

3 如果是取消关注，删除用户关注的作者，删除作者当前的粉丝

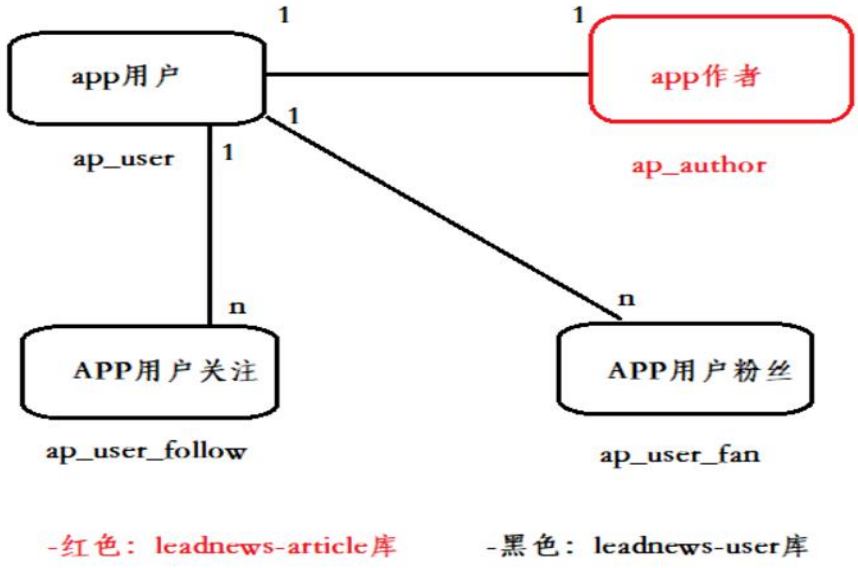
流程分析：



app 端关注信息表

记录了当前登录用户和关注人（作者）的关系，方便当前用户查看关注的作者

- app 端用户粉丝表
- 记录了作者与粉丝的关系，方便作者查看自己的粉丝，同时当前作者也是 app 中的一个用户



app 用户表与 app 作者表是一对一关系，只有在用户认证以后才会有作者出现

- app 用户表与 app 用户关注表是一对多的关系，一个用户可以关注多个作者
- app 用户表与 app 用户粉丝表是一对多的关系，一个用户可以拥有多个粉丝

ap_user_follow APP用户关注信息表

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	int	11		☑	☑	☑	☑	☐	主键
user_id	int	11		☐	☐	☑	☐	☐	用户ID
follow_id	int	11		☐	☐	☑	☐	☐	关注作者ID
follow_name	varchar	20		☐	☐	☐	☐	☐	粉丝昵称
level	tinyint	1		☐	☐	☑	☐	☐	关注度
is_notice	tinyint	1		☐	☐	☑	☐	☐	是否动态通知
created_time	datetime			☐	☐	☐	☐	☐	创建时间

-

ap_user_fan APP用户粉丝信息表

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	int	11		☑	☑	☑	☑	☐	主键
user_id	int	11		☐	☐	☑	☐	☐	用户ID
fans_id	int	11		☐	☐	☑	☐	☐	粉丝ID
fans_name	varchar	20		☐	☐	☐	☐	☐	粉丝昵称
level	tinyint	1		☐	☐	☑	☐	☐	粉丝忠实度
created_time	datetime			☐	☐	☐	☐	☐	创建时间
is_display	tinyint	1		☐	☐	☑	☐	☐	是否可见我动态
is_shield_letter	tinyint	1		☐	☐	☑	☐	☐	是否屏蔽私信
is_shield_comment	tinyint	1		☐	☐	☑	☐	☐	是否屏蔽评论

使用代码生成器生成这两张表对应的代码

- 实现分析
- 判断当前用户是否是正常登陆(不是设备 ID)，发送消息，行为处理都是可以记录的，有登录用户,设置登录用户的 ID，根据 authorId 获取 userId，获取关注的历史记录，查询该用户对应作者的粉丝信息，判断操作类型，0 关注，1 取消。

2.2.4 用户行为处理

用户操作行为记录-点赞行为

需求分析

当前登录的用户点击了”赞“,就要保存当前行为数据

涉及到的表

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	bigint	20		☑	☑	☑	☐	☐	
entry_id	int	11		☐	☐	☑	☐	☐	实体ID
article_id	bigint	20		☐	☐	☑	☐	☐	文章ID
type	tinyint	1		☐	☐	☑	☐	☐	点赞内容类型
operation	tinyint	1		☐	☐	☑	☐	☐	0 点赞
created_time	datetime			☐	☐	☐	☐	☐	登录时间

实现分析

调用微服务查询当前用户点赞记录，更新点赞量时需要判断当前用户是否已经对这篇文章有过点赞数据,没有点赞记录，则更新文章库中文章表的点赞数,行为库中添加点赞记录，如果有点赞记录，不再新增，但是可以修改（取消点赞），获取当前的点赞状态，判断点赞记录的类型与当前是否一致，不一致且操作为取消点赞，如果点赞记录为空,保存点赞记录，取消点赞更新文章表中点赞数,更新行为库记录。

2.2.5 评论系统

文章详情页下方可以查看评论信息，按照点赞数量倒序排列，展示**评论内容、评论的作者、点赞数、回复数、时间**，默认查看 20 条评论，如果想查看更多，可以点击加载更多进行分页

评论概述：

登录用户可以进行评论，可以针对当前文章发布评论。

评论内容不为空且不超过 140 字，评论内容需要做文本反垃圾检测，评论内容需要做敏感词过滤，用户登录后才可以对文章发表评论。

评论话术：

每次用户要评论时，都会进行过滤，通过滤类，确保用户登陆后才能进行评论。

然后识别评论内容，字数判断，140 字以内。文本反垃圾检测、阿里云安全审核敏感词。

都通过后，保存用户评论，正式发布评论。

点赞话术：

同样通过过滤器，保证用户登录后才可以对评论点赞。

根据当前用户 id 和评论 id 查询是否已有点赞记录，如果已有显示为已点赞。

如果没有点赞记录,新增点赞记录,更新评论的点赞数+1

已有点赞记录,判断点赞记录中的点赞状态是否与当前操作一致

不一致进行点赞数更新和点赞记录更新点赞操作,

根据点赞或取消点赞，评论数进行加减 1

评论列表话术：

根据文章 id 查询评论列表，按照点赞的数量进行排序。

如果用户已经登录，需要判断当前登录用户，对评论列表中的哪些评论已点赞。

评论列表会进行分页查询。

回复话术：

回复功能包括，回复评论、回复列表展示、回复点赞。

当用户点击了评论中的回复时就可以查看所有评论回复的内容。

当用户针对当前评论回复时，需要更新回复数量。

当用户对回复点赞时，保存当前回复的点赞信息。

2.2.6 文章搜索

文章搜索话术：

在 APP 端正上方有搜索框，可以输入搜索的关键字。

根据用户输入的关键字通过 es 分词查找文章列表。

文章列表的展示和 home 页展示一样，当用户点击某一篇文章时，可查看详情。

从 es 的查询效率远比直接查数据库要快。所以需要把文章相关的数据存储到 es 中。

搜索时查询 es、展示列表，记录当前用户的搜索历史。

查询结果按照发布时间倒序排列。

搜索记录话术：

搜索历史记录会保存之前的 5 调搜索关键词。用户可以选择手动删除。

在搜索时，会把搜索记录进行保存，存储的是用户实体信息和搜索关键词记录。

在用户再次点击到搜索框时，会查询当前用户的搜索记录列表，展示给用户。

当用户删除记录时，会将状态修改为 0，让历史记录无效，便不会展示。

关键词联想：

根据用户输入的关键字进行联想，默认展示 5 条联想词。

将用户搜索记录经过数据清洗后保存到联想词表,这是一个动态的数据，定时每天抓取搜索较高的 1 万条数据放到这里,之后根据关键字进行查询

由于每次用户输入都会访问数据库，所以使用 redis 缓存进行缓解压力。

讲清晰的词分成词组，根据用户输入的在 redis 中查询出 5 条，并展示。

数据结构为 Trie 树

2.3 自媒体文章

2.3.1 自媒体文章列表查询

业务/需求:

所属：内容列表

分页：需要进行分页

条件：文章状态、关键字、频道列表、发布日期

可以通过左侧的 “内容列表” 进入 文章查询页面。

筛选条件有：

文章状态： 全部、草稿、待审核、审核通过、审核失败

关键字：　　用户输入

频道列表：　通过查询列表并展示

发布日期：　可选择区间

默认查询当前用户的文章，状态：全部，不带有其他筛选信息，分页展示文章列表。

话术：

1.从本地线程获取到当前用户的 id，如果 id 为 null 则用户未登录，给出”请登录”提示。

确保登陆后，通过用户 id 查询出，用户的文章列表。

2.其他相关条件查询时，都会先判断是否为空，

确保用户有输入或选择后再加入对应的查询条件。

条件包含:

关键字、所属频道 ID、文章状态、发布时间的开始和结束

其中所属频道 ID 在 ad_channel 表中，因为该表 id 从 1 开始递增。

如果 id 值大于 0，会根据所属 id 进行查询，

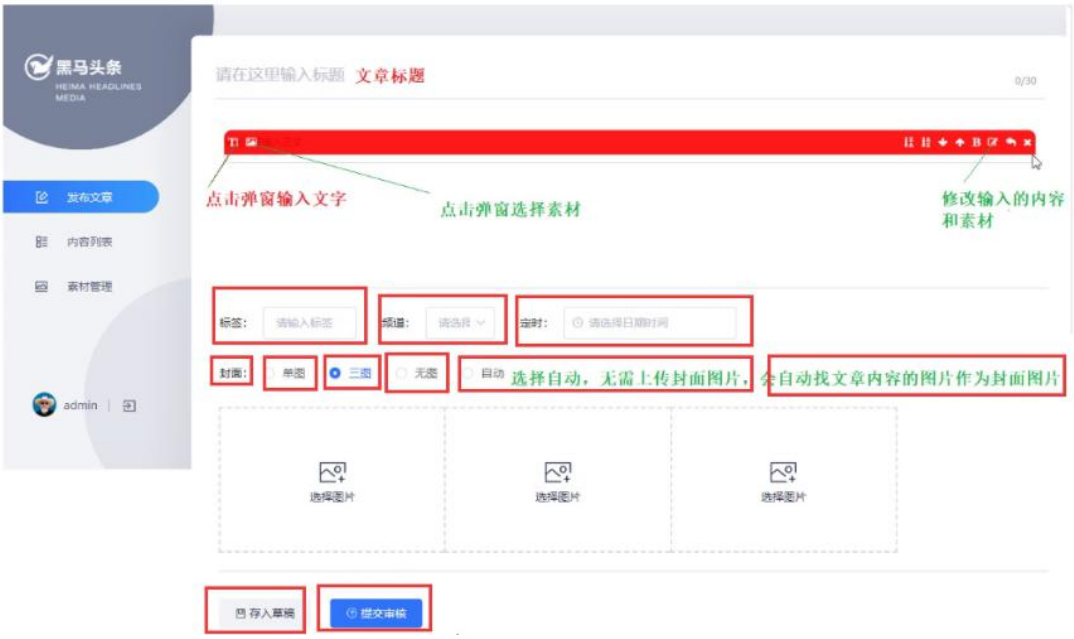
（频道 ID 从 ad_channel 表中查询出所有频道加载到页面即可）

3.对查询的结果进行排序，我们选择根据发布时间倒叙排列。根据实际业务需求可选创建时间、发布时间等进行排序

4.封装好后将其返回，完成文章查询功能。

2.3.2 自媒体文章的发布、修改、保存功能

业务/需求：



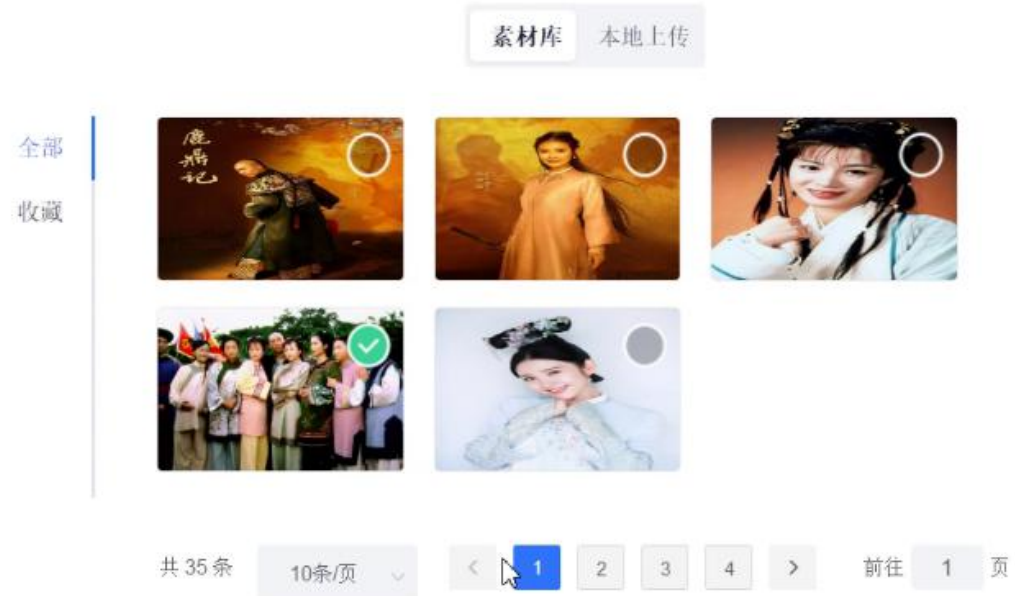
文章包含：

标题： 

内容：

文字内容：点击后可以手动输入需要添加的内容。可以多条。

图片内容：可以从素材库选择或本地上传，可以上传多个图片。



标签：用户自定义标签

频道：可以从列表中选择

定时：设置发布时间，到时间自动发布

封面图片：无图、一张图、三张图

跟前端布局有关。一般都是以上三种图片布局

话术：

该功能为保存、修改、保存草稿的共有方法，

其中需要注意的是

修改：如果有那么是修改文章，先删除所有素材关联关系，再继续操作。

点击修改按钮，携带文章 id，进入编辑页面，可以修改文章的详细信息。

修改的详细信息，都在文章发布功能中实现，如果有 ID 为修改，无 ID 为新增。

封面图片：

如果选择是自动需要从内容中截取图片做为封面图片

截取规则为：内容图片的个数小于等于 2 则为单图截图一张图，内容图片大于等于 3，则为多图，截图三张图，内容中没有图片，则为无图

设置发布和创建时间，通过当前线程获取当前用户，未登录给出提示。

通过 dto 对象获取封面图片，并提取内容中的图片，返回一个图片地址 url 的 List 集合。

判断当前发布文章的封面类型 如果是-1 为自动。

关于封面图片选择 “自动” 选项时，我们会提取内容中的图片。

通过判断 type 是 image 统计数量并获取其值。

内容中图片数量 0 为无图。小于等于 2 为 1 图 大于 2 为 3 图。

超过 3 图会取前三个作为封面。

多张封面图片使用逗号分隔存入。

另外如果是保存草稿不需要添加素材和文章的关系。否则正常添加关系即可。

2.3.3 自媒体文章的修改

业务/需求：

通过点击现有文章进入编辑页面。可以对现有文章进行编辑修改。

话术：

点击修改的时候，就是根据文章 id 查询，跳转至编辑页面进行展示。

这里只做一个页面跳转，具体修改功能的实现是由之前自媒体文章发布功能实现。

2.3.4 自媒体文章的删除功能

业务/需求：

点击现有文章的删除按钮，可以删除文章。

想要删除时，需要几个前提条件

1. 文章 id 存在
2. 文章存在
3. 状态不是 9

然后可以进行文章和素材关系的解除。之后再进行删除文章即可。

话术：

在做文章删除功能时，除了状态为 9 已发布的文章不能删除外。

其他状态状态下都可以进行删除。

我们会先把文章和素材的关联关系删除掉，然后再进行对应文章的删除。

2.3.5 自媒体文章的上下架功能

业务/需求：

对已发布的文章进行上下架操作

针对当前状态为 9(已发布)的文章进行上下架操作，并把数据同步到文章库中。使用 kafka 实现

话术：

前台页面只有文章状态为 9 的会出现上下架选项，

先 WmNewsdto 对象保证不为空，并且 id 不为空。然后获取文章

确保文章存在，并且状态为 9 已发布时，就可以进行上下架操作了。

我们只需要把 enable 修改一下即可。然后通过 Kafka 同步到 App 端

直接发送一个包含上下架信息的主题(Topic)，存的是 map 转换后的 json 包含 enable（上下架状态），和 articleId(发布文章库 ID)。

2.3.6 自媒体文章审核

1.自媒体文章审核的流程

业务/需求：

自媒体文章可以自由发布，所以文章的内容是否符合规范、安全非常重要。所以我们要细化内容审核的流程，来把控发布的文章。

文章审核分为 人工审核 和 自动审核。

人工审核：

状态为 4 的进行人工审核。我们只需要把文章分配给审核工作人员即可。

如果人工审核的文章发布时间大于当前时间，即可直接修改状态为 8 审核通过(待发布)

否则保存文章相关数据。

自动审核：

状态为 1 的需要进行自动审核。

我们会调用阿里云文本反垃圾服务，进行文章自动审核。

如果审核不成功或需要人工审核，修改文章状态。

如果当前文章有图片，调用阿里云图片审核服务，同样如果不成功，需要修改文章状态。

自动审核通过

如果文章发布时间大于当前时间，修改自媒体文章状态为 8 审核通过(待发布)

否则保存文章相关数据

2.文章审核

业务/需求：

自媒体和文章我们使用 Spring-Cloud 的 feign 进行远程接口调用。

自媒体 feign

1 根据文章 id 查询自媒体文章的数据

2 在审核的过程中,审核失败或者成功需要修改自媒体文章的状态

3 在文章进行保存的时候需要查询作者信息，需要通过自媒体用户关联查询作者信息

文章 feign

1 保存文章信息 ap_article

2 保存文章内容 ap_article_content

3 在保存文章的时候需要关联作者，需要根据名称查询作者信息

考虑到发表的文章会随时间增加，数据量巨大。所以使用分布式 id，分库进行保存。

所以我们选择使用 雪花算法生成数据库 id，Mybaits 逆向工程使用 ASSIGN_ID 生成。

雪花算法：

snowflake 是 Twitter 开源的分布式 ID 生成算法，结果是一个 long 型的 ID。其核心

思想是：使用 41bit 作为毫秒数，10bit 作为机器的 ID（5 个 bit 是数据中心，5 个 bit 的机器 ID），12bit 作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），最后还有一个符号位，永远是 0

话术：

文章审核：

我们的文章审核功能围绕着状态来实现。首先我们发布的文章初始状态肯定是 1(待审核)，所以我们会根据文章**状态**进行如下操作：

状态 1

进行自动审核，通过阿里云安全的内容文本检测、图片审核，

DFA 敏感词过滤来完成审核。

文本审核结果会有 1 审核通过 2 审核失败 3 需要人工审核。

自动审核通过后，根据当前时间是否大于发布时间进行判断

是：直接保存到文章库

否：修改状态为 8 审核通过(待发布)

先保存数据，等发布时间到了再进行发布。

状态 8 和状态 4

8 和 4 为人工审核通过或自动审核通待发布的文章，

所以只需要判断当前时间是否大于发布时间。

是：直接保存到文章库

否：先保存数据，等发布时间到了再进行发布。

2.3.7 自媒体文章自动审核流程

作为内容类产品，内容安全非常重要，所以需要进行对自媒体用户发布的文章进行审核以后才能到 app 端展示给用户。

审核的流程如下：

1.当自媒体用户提交发布文章之后，会发消息给 kafka 提交审核，平台运营端监听消息

2.根据自媒体文章 id 查询文章信息

3.如果当前文章的状态为 4（人工审核通过），则无需再进行自动审核，如果发布时间大于当前时间,将状态修改为 8,否则保存 APP 文章相关数据

4.文章状态为 8,发布时间小于等于当前时间,则直接保存 app 文章相关数据

5.文章状态为 1，则进行自动审核

5.1 调用阿里云文本反垃圾服务，进行文本审核，如果审核不成功或需要人工审核，修改自媒体文章状态

5.2 如果文章中有图片,调用阿里云图片审核服务，如果审核不通过或需要人工审核，修改自媒体文章状态

5.3 文章内容中是否有自管理的敏感词，如果有则审核不通过，修改自媒体文章状态

5.4 自动审核通过,如果文章发布时间大于当前时间，修改自媒体文章状态为 8（审核通过待发布状态）

5.5 自动审核通过,如果文章发布时间小于等于当前时间，保存 app 相关数据

6.保存 app 相关数据,修改自媒体文章状态为 9 （已发布）

ap_article 文章

ap_article_content 文章内容

ap_author 文章作者

7.创建索引（为后续 app 端的搜索功能做数据准备）

表结构

（1） wm_news 自媒体文章表 在自媒体库、

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	int	11		☑	☑	☐	☑	☐	主键
user_id	int	11		☐	☐	☑	☐	☐	自媒体用户ID
title	varchar	36		☐	☐	☐	☐	☐	标题
content	longtext			☐	☐	☐	☐	☐	图文内容
type	tinyint	1		☐	☐	☑	☐	☐	文章布局
channel_id	int	11		☐	☐	☑	☐	☐	图文频道ID
labels	varchar	20		☐	☐	☐	☐	☐	
created_time	datetime			☐	☐	☐	☐	☐	创建时间
submited_time	datetime			☐	☐	☐	☐	☐	提交时间
status	tinyint	2		☐	☐	☑	☐	☐	当前状态 0 草稿 1 提交（待审核）2 审核失败 3 人工审核 4 人工审核通过 8 审核通过（待发布） 9 已发布
publish_time	datetime			☐	☐	☐	☐	☐	定时发布时间，不定时则为空
reason	varchar	50		☐	☐	☐	☐	☐	拒绝理由
article_id	int	11		☐	☐	☑	☐	☐	发布库文章ID
images	varchar	255	''	☐	☐	☐	☐	☐	//图片用逗号分隔
enable	tinyint	1	1	☐	☐	☑	☐	☐	
				☐	☐	☐	☐	☐	

status 字段：0 草稿 1 待审核 2 审核失败 3 人工审核 4 人工审核通过 8 审核通过（待发布） 9 已发布

（2）ap_author 文章作者表 在 article 库

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	int	11		☑	☑	☑	☑	☐	主键
name	varchar	20		☐	☐	☐	☐	☐	作者名称
type	tinyint	1		☐	☐	☑	☐	☐	0 爬取数据 1 签约合作商 2 平台自媒体人
user_id	int	11		☐	☐	☑	☐	☐	社交账号ID
created_time	datetime			☐	☐	☐	☐	☐	创建时间
wm_user_id	int	11		☐	☐	☑	☐	☐	自媒体账号

（3）ap_article 文章信息表 在 article 库

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	bigint	20		☑	☑	☑	☑	☐	
title	varchar	50		☐	☐	☐	☐	☐	标题
author_id	int	11		☐	☐	☑	☐	☐	文章作者的ID
author_name	varchar	20		☐	☐	☐	☐	☐	作者昵称
channel_id	int	10		☐	☐	☑	☐	☐	文章所属频道ID
channel_name	varchar	10		☐	☐	☐	☐	☐	频道名称
layout	tinyint	1		☐	☐	☑	☐	☐	文章布局
flag	tinyint	3		☐	☐	☑	☐	☐	文章标记
images	varchar	1000		☐	☐	☐	☐	☐	文章图片
labels	varchar	500		☐	☐	☐	☐	☐	文章标签最多3个 逗号分隔
likes	int	5		☐	☐	☑	☐	☐	点赞数量
collection	int	5		☐	☐	☑	☐	☐	收藏数量
comment	int	5		☐	☐	☑	☐	☐	评论数量
views	int	5		☐	☐	☑	☐	☐	阅读数量
province_id	int	11		☐	☐	☑	☐	☐	省市
city_id	int	11		☐	☐	☑	☐	☐	市区
county_id	int	11		☐	☐	☑	☐	☐	区县
created_time	datetime			☐	☐	☐	☐	☐	创建时间
publish_time	datetime			☐	☐	☐	☐	☐	发布时间
sync_status	tinyint	1	0	☐	☐	☐	☐	☐	同步状态
origin	tinyint	1	0	☐	☐	☑	☐	☐	来源

- layout 文章布局 0 无图文章 1 单图文章 2 多图文章
- flag 文章标记 0 普通文章 1 热点文章 2 置顶文章 3 精品文章 4 大 V 文章
- images 文章图片 多张图片使用逗号分隔

（4）ap_article_content 文章内容表 在 article 库

Field Name	Datatype	Len	De	PK?	Not Null?	Un	Au	Ze	Comment
id	bigint	20		☑	☑	☑	☑	☐	主键
article_id	bigint	20		☐	☐	☑	☐	☐	文章ID
content	longtext			☐	☐	☐	☐	☐	文章内容

自媒体文章审核实现

当自媒体用户提交发布文章之后，会发消息给 kafka 提交审核，平台运营端监听消息，根据自媒体文章 id 查询文章信息，判断当前文章的状态，从自媒体文章中提取文本和图片内容，调用阿里云文本反垃圾服务，进行文本审核，如果审核不成功或需要人工审核，修改自媒体文章，调用阿里云图片审核服务，如果审核不通过或需要人工审核，修改自媒体文章状态，文章内容中是否有自管理的敏感词，如果有则审核不通过，修改自媒体文章状态，自媒体文章发布时间大于当前时间，修改自媒体文章状态为 8（审核通过待发布状态），审核通过，修改自媒体文章状态为 9 （审核通过），保存 app 相关数据，如果当前文章的状态为 4（人工审核通过），则无需再进行自动审核审核，保存 app 文章相关数据即可，文章状态为 8,发布时间<=当前时间,则修改发布状态，文章状态为 1，则进行自动审核，

2.3.8 定时任务扫描待发布文章

业务/需求:

定时任务的作用就是每分钟去扫描那些待发布的文章，如果当前文章的状态为 8，

并且发布时间小于当前时间的，立刻发布当前文章

流程：

1. 文章数据准备，远程调用接口自媒体端文章查询功能，得到状态为 8 且发布时间小于当前时间的文章。
2. 在任务调度中心创建调度任务，新建一个执行器，策略为轮询，每分钟执行一次。
3. 将扫描到符合条件的文章进行发布。

话术：

通过自媒体短的文章查询准备好数据后，我在项目中引入了 xxl-job 调度任务中心，定义一个自媒体文章审核的执行器，路由策略采用轮询，每分钟执行一次扫描，如果发现有当前文章的状态为 8，并且发布时间小于当前时间的，立刻发布当前文章。

2.3.8 人工审核文章-admin 端

业务/需求:

前提：自动审核失败，状态为 3 的。

自动审核使用的是阿里云和 DFA。某些情况自动审核会不通过，比如一些无法识别、无法分辨的信息等。会把文章从自动审核转到人工审核状态，并添加到人工审核列表。审核人员可以从列表中看到需要进行人工审核的文章，并且进行审核后选择 通过或不通过。

平台管理员可以查看待人工审核的文章信息，可以通过（状态改为 4）或驳回（状态改为 2）也可以通过点击查看按钮，查看文章详细信息，查看详情后可以根据内容判断是否需要通过审核

话术：

我们在人工审核这里主要考虑的是列表的展示、并且增加一些搜索功能，方便找到对应的文章。可以根据文章 ID 查看文章详情。对详情内容进行详细过审后，可以根据人工结果选择操作通过或不通过。

当审核通过后修改文章状态为 4，如果审核不通过 修改状态为 2，并添加审核不通过

原因。

2.3.9 自媒体登陆

通过网关进行授权验证

首先判断地址中是否包含登录的地址，如果是登陆的地址，不需要 token，直接放行，如果不是登陆地址，需要验证 token。先获取 token（获取请求头对象），判断 token 是否为空，如果为空返回 401，结束请求，不为空需要验证 token，验证通过，需要将用户 ID 放入到请求头中，供后面的微服务使用。

2.4 新热文章计算

业务/需求:

筛选出文章列表中最近 5 天热度较高的文章在每个频道的首页展示

根据用户的行为（阅读、点赞、评论、收藏）实时计算热点文章

话术：

新热文章计算分为三个部分

1.我们会定时计算热点文章，设置定时任务每日凌晨 1 点开始，查询前 5 天的文章。

计算每个文章的分值，其中不同的行为设置不同的权重（阅读：1，点赞：3，评论：5，收藏：8）

按照分值排序，给每个频道找出分值较高的 30 条数据，存入缓存中

2.实时计算热点文章

- 行为微服务，用户阅读或点赞了某一篇文章，发送消息给 kafka
- 文章微服务，接收行为消息，使用 kafkastream 流式处理进行聚合，发消息给 kafka
- 文章微服务，接收聚合之后的消息，计算文章分值（当日分值计算方式，在原有权重的基础上再*3）
- 根据当前文章的频道 id 查询缓存中的数据
- 当前文章分值与缓存中的数据比较，如果当前分值大于某一条缓存中的数据，则直接替换
- 新数据重新设置到缓存中

3. 查询热点数据

- 判断是否是首页
- 是首页，选择是推荐，channelId 值为`0`，从所有缓存中筛选出分值最高的 30 条数据返回
- 是首页，选择是具体的频道，channelId 是具体的数字，从缓存中获取对应的频道中的数据返回
- 不是，则查询数据库中的数据

三、技术点

1、Kafka

1.1 Kafka 是什么?

Kafka 是一款分布式流处理框架，用于实时构建流处理应用。它有一个核心 的功能广为人知，即作为企业级的消息引擎被广泛使用。

你一定要先明确它的流处理框架地位，这样能给面试官留 下一个很专业的印象。

1.2 Kafka 的特点?

高吞吐量、低延迟：kafka 每秒可以处理几十万条消息，它的延迟最低只有几毫秒
可扩展性：kafka 集群支持热扩展
持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失
容错性：允许集群中节点失败（若副本数量为 n,则允许 n-1 个节点失败）
高并发：支持数千个客户端同时读写

1.3 什么是消费者组?

消费者组是 Kafka 独有的概念，如果面试官问这 个，就说明他对此是有一定了解的。我先给出标准答案：
1、定义：即消费者组是 Kafka 提供的可扩展且具有容错性的消费者机制。
2、原理：在 Kafka 中，消费者组是一个由多个消费者实例 构成的组。多个实例共同订阅若干个主题，实现共同消费。同一个组下的每个实例都配置有 相同的组 ID，被分配不同的订阅分区。当某个实例挂掉的时候，其他实例会自动地承担起 它负责消费的分区。

此时，又有一个小技巧给到你:消费者组的题目，能够帮你在某种程度上掌控下面的面试方向。

如果你擅长位移值原理，就不妨再提一下**消费者组的位移提交机制**;

如果你擅长 Kafka Broker，可以提一下**消费者组与 Broker 之间的交互**;

如果你擅长与消费者组完全不相关的 Producer，那么就可以这么说:“**消费者组要消 费的数据完全来自于 Producer 端生产的消息，我对 Producer 还是比较熟悉的。**”

1.4 Kafka 中，ZooKeeper 的作用是什么?

这是一道能够帮助你脱颖而出的题目。碰到这个题目，请在心中暗笑三声。

目前，Kafka 使用 ZooKeeper 存放集群元数据、成员管理、Controller 选举，以及其他一些管理类任务。之后，等 KIP-500 提案完成后，Kafka 将完全不再依赖 于 ZooKeeper。

记住，一定要突出“目前”，以彰显你非常了解社区的演进计划。“存放元数据”是指主题 分区的所有数据都保存在 ZooKeeper 中，且以它保存的数据为权威，其他“人”都要与它 保持对齐。“成员管理”是指 Broker 节点的注册、注销以及属性变更，等 等。“Controller 选举”是指选举集群 Controller，而其他管理类任务包括但不限于主题 删除、参数配置等。

不过，抛出 KIP-500 也可能是个双刃剑。碰到非常资深的面试官，他可能会进一步追问你 KIP-500 是做的。一言以蔽之:KIP-500 思想，是使用社区自研的基于 Raft 的共识算法， 替代 ZooKeeper，实现 Controller 自选举。

1.5 如何估算 Kafka 集群的机器数量?

这道题目考查的是机器数量和所用资源之间的关联关系。所谓资源，也就是 CPU、内存、磁盘和带宽。

通常来说，CPU 和内存资源的充足是比较容易保证的，因此，你需要从磁盘空间和带宽占用两个维度去评估机器数量。

在预估磁盘的占用时，你一定不要忘记计算副本同步的开销。如果一条消息占用 1KB 的磁 盘空间，那么，在有 3 个副本的主题中，你就需要 3KB 的总空间来保存这条消息。显式地 将这些考虑因素答出来，能够彰显你考虑问题的全面性，是一个难得的加分项。

对于评估带宽来说，常见的带宽有 1Gbps 和 10Gbps，但你要切记，这两个数字仅仅是最大值。因此，你最好和面试官确认一下给定的带宽是多少。然后，明确阐述出当带宽占用接 近总带宽的 90% 时，丢包情形就会发生。这样能显示出你的网络基本功。

1.6 Kafka 为什么不支持读写分离?

这道题目考察的是你对 Leader/Follower 模型的思考。

Leader/Follower 模型并没有规定 Follower 副本不可以对外提供读服务。很多框架都是允 许这么做的，只是 Kafka 最初为了避免不一致性的问题，而采用了让 Leader 统一提供服 务的方式。

不过，在开始回答这道题时，你可以率先亮出观点:**自 Kafka 2.4 之后，Kafka 提供了有限度的读写分离，也就是说，Follower 副本能够对外提供读服务。**

说完这些之后，你可以再给出之前的版本不支持读写分离的理由。

场景不适用：读写分离适用于那种读负载很大，而写操作相对不频繁的场景，可 Kafka 不属于这样的场景。

同步机制：Kafka 采用 PULL 方式实现 Follower 的同步，因此，Follower 与 Leader

存 在不一致性窗口。如果允许读 Follower 副本，就势必要处理消息滞后(Lagging)的问题。