

前言

本手册收集了Java相关技术栈的面试题，旨在帮助大家找到心仪的工作。内容大部分来自互联网，由公众号：**Java专栏**进行整理汇总。

另外打铁还需自身硬，希望通过这个手册能够帮助大家查漏补缺。

如果你是初学者希望这个手册可以给你提供一个相对完善的学习路径，如果你是工作多年的程序员，希望这个手册可以让你重温核心知识点。

声明：本手册独家整理，仅供个人使用。未经允许严禁进行商业等用途。

整理不易，目前全手册共计24W字。如果你觉得对你有用，可以进行打赏。（**金额不重要，有那个心意就很满足了！**）



Java面试手册

微信扫描二维码

加入微信交流群，

获取最新Java面试手册



Java相关

基础&进阶篇

1. 什么是Java

Java是一门面向对象的高级编程语言，不仅吸收了C++语言的各种优点，比如继承了C++语言面向对象的技术核心。还摒弃了C++里难以理解的多继承、指针等概念，同时也增加了垃圾回收机制，释放掉不被使用的内存空间，解决了管理内存空间的烦恼。

因此Java语言具有功能强大和简单易用两个特征。Java语言作为静态面向对象编程语言的代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程。

2. Java的特点有哪些

Java语言是一种分布式的面向对象语言，具有面向对象、平台无关性、简单性、解释执行、多线程、安全性等很多特点，下面针对这些特点进行逐一介绍。

1. 面向对象

Java 是一种面向对象的语言，它对对象中的类、对象、继承、封装、多态、接口、包等均有很好的支持。为了简单起见，Java 只支持类之间的单继承，但是可以使用接口来实现多继承。使用 Java 语言开发程序，需要采用面向对象的思想设计程序和编写代码。

2. 平台无关性

平台无关性的具体表现在于，Java 是“一次编写，到处运行（Write Once, Run Anywhere）”的语言，因此采用 Java 语言编写的程序具有很好的可移植性，而保证这一点的正是 Java 的虚拟机机制。在引入虚拟机之后，Java 语言在不同的平台上运行不需要重新编译。

Java 语言使用 Java 虚拟机机制屏蔽了具体平台的相关信息，使得 Java 语言编译的程序只需生成虚拟机上的目标代码，就可以在多种平台上不加修改地运行。

3. 简单性

Java 语言的语法与 C 语言和 C++ 语言很相近，使得很多程序员学起来很容易。对 Java 来说，它舍弃了很多 C++ 中难以理解的特性，如操作符的重载和多继承等，而且 Java 语言不使用指针，加入了垃圾回收机制，解决了程序员需要管理内存的问题，使编程变得更加简单。

4. 解释执行

Java 程序在 Java 平台运行时会被编译成字节码文件，然后可以在有 Java 环境的操作系统上运行。在运行文件时，Java 的解释器对这些字节码进行解释执行，执行过程中需要加入的类在连接阶段被载入到运行环境中。

5. 多线程

Java 语言是多线程的，这也是 Java 语言的一大特性，它必须由 Thread 类和它的子类来创建。Java 支持多个线程同时执行，并提供多线程之间的同步机制。任何一个线程都有自己的 run() 方法，要执行的方法就写在 run() 方法体内。

6. 分布式

Java 语言支持 Internet 应用的开发，在 Java 的基本应用编程接口中就有一个网络应用编程接口，它提供了网络应用编程的类库，包括 URL、URLConnection、Socket 等。Java 的 RIM 机制也是开发分布式应用的重要手段。

7. 健壮性

Java 的强类型机制、异常处理、垃圾回收机制等都是 Java 健壮性的重要保证。对指针的丢弃是 Java 的一大进步。另外，Java 的异常机制也是健壮性的一大体现。

8. 高性能

Java 的高性能主要是相对其他高级脚本语言来说的，随着 JIT（Just in Time）的发展，Java 的运行速度也越来越高。

9. 安全性

Java 通常被用在网络环境中，为此，Java 提供了一个安全机制以防止恶意代码的攻击。除了 Java 语言具有许多的安全特性以外，Java 还对通过网络下载的类增加一个安全防范机制，分配不同的名字空间以防替代本地的同名类，并包含安全管理机制。

Java 语言的众多特性使其在众多的编程语言中占有较大的市场份额，Java 语言对对象的支持和强大的 API 使得编程工作变得更加容易和快捷，大大降低了程序的开发成本。Java 的“一次编写，到处执行”正是它吸引众多商家和编程人员的一大优势。

3. JDK和JRE和JVM的区别

1. JDK

JDK (Java SE Development Kit) , Java标准的开发包，提供了编译、运行Java程序所需要的各种工具和资源，包括了Java编译器、Java运行时环境、以及常用的Java类库等。

2. JRE

JRE (Java Runtime Environment) , Java运行时环境，用于解释执行Java的字节码文件。普通用户只需要安装JRE来运行Java程序即可，而作为一名程序员必须安装JDK，来编译、调试程序。

3. JVM

JVM (Java Virtual Machine) , Java虚拟机，是JRE的一部分。它是整个Java实现跨平台的核心，负责解释执行字节码文件，是可运行Java字节码文件的虚拟计算机。所有平台上的JVM向编译器提供相同的接口，而编译器只需要面向虚拟机，生成虚拟机能识别的代码，然后由虚拟机来解释执行。

当使用Java编译器编译Java程序时，生成的是与平台无关的字节码，这些字节码只面向JVM。也就是说**JVM是运行Java字节码的虚拟机**。

不同平台的JVM是不同的，但是他们都提供了相同的接口。JVM是Java程序跨平台的关键部分，只要为不同平台实现了相同的虚拟机，编译后的Java字节码就可以在该平台上运行。

为什么要采用字节码：

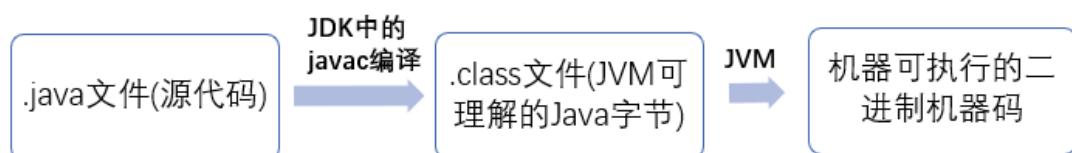
在Java中，JVM可以理解的代码就叫做字节码（即Java源代码经过虚拟机编译器编译后扩展名为.class的文件），它不面向任何特定的处理器，只面向虚拟机。Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java程序无须重新编译便可在多种不同操作系统的计算机上运行。

什么是跨平台：

所谓跨平台性，是指Java语言编写的程序，一次编译后，可以在多个系统平台上运行。

实现原理：Java程序是通过Java虚拟机在系统平台上运行的，只要该系统可以安装相应的Java虚拟机，该系统就可以运行Java程序。

Java程序从源代码到运行需要三步：



4. 总结

1. JDK 用于开发，JRE 用于运行Java程序；如果只是运行Java程序，可以只安装JRE，无须安装JDK。

微信搜索公众号：Java专栏，获取最新面试手册

2. JDK包含JRE，JDK 和 JRE 中都包含 JVM。
3. JVM 是 Java 编程语言的核心并且具有平台独立性。

4. Oracle JDK 和 OpenJDK 的对比

- Oracle JDK版本将每三年发布一次，而OpenJDK版本每三个月发布一次；
- OpenJDK 是一个参考模型并且是完全开源的，而Oracle JDK是OpenJDK的一个实现，并不是完全开源的；
- Oracle JDK 比 OpenJDK 更稳定。OpenJDK和Oracle JDK的代码几乎相同，但Oracle JDK有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到Oracle JDK就可以解决问题；
- 在响应性和JVM性能方面，Oracle JDK与OpenJDK相比提供了更好的性能；
- Oracle JDK不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
- Oracle JDK根据二进制代码许可协议获得许可，而OpenJDK根据GPL v2许可获得许可。

5. Java有哪些数据类型

Java中有 8 种基本数据类型，分别为：

- **6 种数字类型（四个整数形，两个浮点型）**：byte、short、int、long、float、double
- **1 种字符类型**：char
- **1 种布尔型**：boolean。

byte:

- byte 数据类型是8位、有符号的，以二进制补码表示的整数；
- 最小值是 **-128 (-2^7)**；
- 最大值是 **127 (2^7-1)**；
- 默认值是 **0**；
- byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一；
- 例子：byte a = 100, byte b = -50。

short:

- short 数据类型是 16 位、有符号的以二进制补码表示的整数
- 最小值是 **-32768 (-2^15)**；
- 最大值是 **32767 (2^15 - 1)**；
- Short 数据类型也可以像 byte 那样节省空间。一个short变量是int型变量所占空间的二分之一；
- 默认值是 **0**；
- 例子：short s = 1000, short r = -20000。

int:

- int 数据类型是32位、有符号的以二进制补码表示的整数；
- 最小值是 **-2,147,483,648 (-2^31)**；
- 最大值是 **2,147,483,647 (2^31 - 1)**；
- 一般地整型变量默认为 int 类型；
- 默认值是 **0**；

- 例子: int a = 100000, int b = -200000。

long:

- 注意: Java 里使用 long 类型的数据一定要在数值后面加上 L, 否则将作为整型解析
- long 数据类型是 64 位、有符号的以二进制补码表示的整数;
- 最小值是 **-9,223,372,036,854,775,808 (-2^63)** ;
- 最大值是 **9,223,372,036,854,775,807 (2^63 -1)** ;
- 这种类型主要使用在需要比较大整数的系统上;
- 默认值是 **0L**;
- 例子: long a = 100000L, Long b = -200000L。
"L"理论上不分大小写, 但是若写成"l"容易与数字"1"混淆, 不容易分辨。所以最好大写。

float:

- float 数据类型是单精度、32位、符合IEEE 754标准的浮点数;
- float 在储存大型浮点数组的时候可节省内存空间;
- 默认值是 **0.0f**;
- 浮点数不能用来表示精确的值, 如货币;
- 例子: float f1 = 234.5f。

double:

- double 数据类型是双精度、64 位、符合IEEE 754标准的浮点数;
- 浮点数的默认类型为double类型;
- double类型同样不能表示精确的值, 如货币;
- 默认值是 **0.0d**;
- 例子: double d1 = 123.4。

char:

- char类型是一个单一的 16 位 Unicode 字符;
- 最小值是 **\u0000** (即为 0) ;
- 最大值是 **\uffff** (即为 65535) ;
- char 数据类型可以储存任何字符;
- 例子: char letter = 'A'; (单引号)

boolean:

- boolean数据类型表示一位的信息;
- 只有两个取值: true 和 false;
- 这种类型只作为一种标志来记录 true/false 情况;
- 默认值是 **false**;
- 例子: boolean one = true。

这八种基本类型都有对应的包装类分别为: **Byte、Short、Integer、Long、Float、Double、Character、Boolean**

类型名称	字节、位数	最小值	最大值	默认值	例子
byte字节	1字节, 8位	-128 (-2^7)	127 (2^7-1)	0	byte a = 100, byte b = -50
short短整型	2字节, 16位	-32768 (-2^15)	32767 (2^15 - 1)	0	short s = 1000, short r = -20000
int整形	4字节, 32位	-2,147,483,648 (-2^31)	2,147,483,647 (2^31 - 1)	0	int a = 100000, int b = -200000
long长整型	8字节, 64位	-9,223,372,036,854,775,808 (-2^63)	9,223,372,036,854,775,807 (2^63 - 1)	0L	long a = 100000L, Long b = -200000L
double双精度	8字节, 64位		double类型同样不能表示精确的值, 如货币	0.0d	double d1 = 123.4
float单精度	4字节, 32位	在储存大型浮点数组的时候可节省内存空间	不同统计精准的货币值	0.0f	float f1 = 234.5f
char字符	2字节, 16位	\u0000 (即为0)	\uffff (即为65,535)	可以储存任何字符	char letter = 'A';
boolean布尔	返回true和false两个值	这种类型只作为一种标志来记录 true/false 情况;	只有两个取值: true 和 false;	false	boolean one = true

6. Java中引用数据类型有哪些，它们与基本数据类型有什么区别？

引用数据类型分3种：类，接口，数组；

简单来说，只要不是基本数据类型，都是引用数据类型。那他们有什么不同呢？

1、从概念方面来说

1,基本数据类型:变量名指向具体的数值

2,引用数据类型:变量名不是指向具体的数值,而是指向存数据的内存地址,也及时hash值

2、从内存的构建方面来说(内存中有堆内存和栈内存两者)

1,基本数据类型:被创建时,在栈内存中会被划分出一定的内存,并将数值存储在该内存中.

2,引用数据类型:被创建时,首先会在栈内存中分配一块空间,然后在堆内存中也会分配一块具体的空间用来存储数据的具体信息,即hash值,然后由栈中引用指向堆中的对象地址.

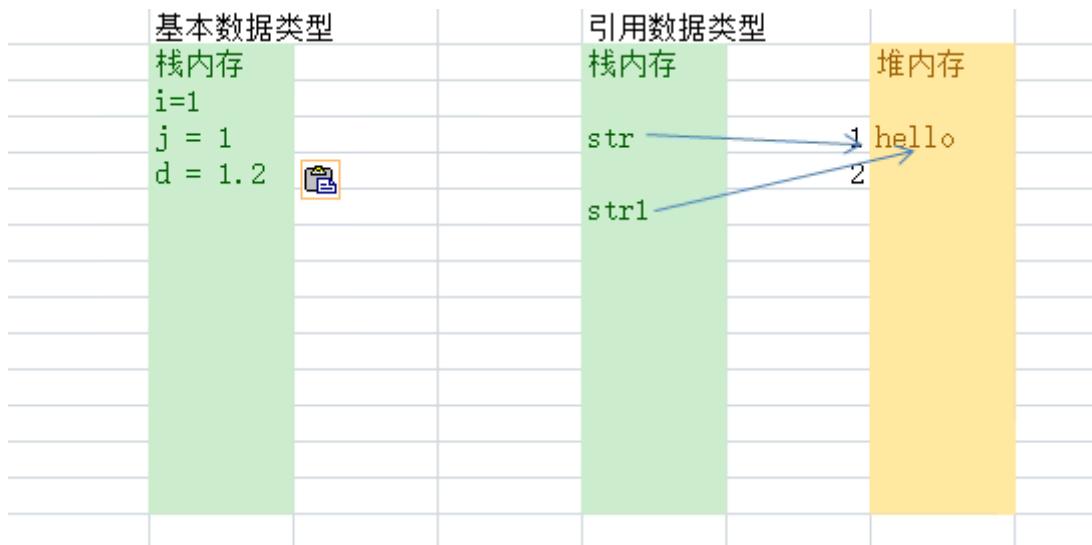
举个例子

```

//基本数据类型作为方法参数被调用
public class Main{
    public static void main(String[] args){
        //基本数据类型
        int i = 1;
        int j = 1;
        double d = 1.2;

        //引用数据类型
        String str = "Hello";
        String str1= "Hello";
    }
}

```



由上图可知，基本数据类型中会存在两个相同的1，而引用型类型就不会存在相同的数据。

假如"hello"的引用地址是xxxxx1，声明str变量并其赋值"hello"实际上就是让str变量引用了"hello"的内存地址，这个内存地址就存储在堆内存中，是不会改变的，当再次声明变量str1也是赋值为"hello"时，此时就会在堆内存中查询是否有"hello"这个地址，如果堆内存中已经存在这个地址了，就不会再次创建了，而是让str1变量也指向xxxxx1这个地址，如果没有的话，就会重新创建一个地址给str1变量。

7. 从使用方面来说

1. 基本数据类型：判断数据是否相等，用==和!=判断。

2. 引用数据类型：判断数据是否相等，用equals()方法，==和!=是比较数值的。而equals()方法是比较内存地址的。

补充：数据类型选择的原则

- 如果要表示整数就使用int，表示小数就使用double；
- 如果要描述日期时间数字或者表示文件（或内存）大小用long；
- 如果要实现内容传递或者编码转换使用byte；
- 如果要实现逻辑的控制，可以使用boolean；
- 如果要使用中文，使用char避免中文乱码；
- 如果按照保存范围： byte < int < long < double；

8. Java中的自动装箱与拆箱

什么是自动装箱拆箱?

从下面的代码中就可以看到装箱和拆箱的过程

```
//自动装箱  
Integer total = 99;  
  
//自定拆箱  
int totalprim = total;
```

装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类型转换为基本数据类型。

在Java SE5之前，自动装箱要这样写：Integer i = new `` Integer(10``);

对于Java的自动装箱和拆箱，我们看看源码编译后的class文件，其实装箱调用包装类的valueOf方法，拆箱调用的是Integer.intValue方法，下面就是变编译后的代码：

```
public class box.BoxTest {  
    public box.BoxTest();  
    Code:  
        0: aload_0  
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V  
        4: aload_0  
        5: iconst_1  
        6: invokestatic #2                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
        9: putfield      #3                  // Field i:Ljava/lang/Integer;  
       12: aload_0  
       13: iconst_2  
       14: invokestatic #2                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
       17: putfield      #4                  // Field j:Ljava/lang/Integer;  
       20: aload_0  
       21: aload_0  
       22: getfield      #3                  // Field i:Ljava/lang/Integer;  
       25: invokevirtual #5                  // Method java/lang/Integer.intValue:()I  
       28: putfield      #6                  // Field m:I  
       31: aload_0  
       32: aload_0  
       33: getfield      #4                  // Field j:Ljava/lang/Integer;  
       36: invokevirtual #5                  // Method java/lang/Integer.intValue:()I  
       39: putfield      #7                  // Field n:I  
       42: return  
}
```

常见面试一：

这段代码输出什么？

```
public class Main {
    public static void main(String[] args) {

        Integer i1 = 100;
        Integer i2 = 100;
        Integer i3 = 200;
        Integer i4 = 200;

        System.out.println(i1==i2);
        System.out.println(i3==i4);
    }
}
```

答案是：

```
true  
false
```

为什么会出现这样的结果？输出结果表明i1和i2指向的是同一个对象，而i3和i4指向的是不同的对象。此时只需一看源码便知究竟，下面这段代码是Integer的valueOf方法的具体实现：

```
public static Integer valueOf(int i) {
    if(i >= -128 && i <= IntegerCache.high)
        return IntegerCache.cache[i + 128];
    else
        return new Integer(i);
}
```

```
private static class IntegerCache {
    static final int high;
    static final Integer cache[];

    static {
        final int low = -128;

        // high value may be configured by property
        int h = 127;
        if (integerCacheHighPropValue != null) {
            // Use Long.decode here to avoid invoking methods that
            // require Integer's autoboxing cache to be initialized
            int i = Long.decode(integerCacheHighPropValue).intValue();
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - -low);
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);
    }
}
```

```
private IntegerCache() {}  
}
```

从这2段代码可以看出，在通过valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。

上面的代码中i1和i2的数值为100，因此会直接从cache中取已经存在的对象，所以i1和i2指向的是同一个对象，而i3和i4则是分别指向不同的对象。

常见面试二：

```
public class Main {  
    public static void main(String[] args) {  
  
        Double i1 = 100.0;  
        Double i2 = 100.0;  
        Double i3 = 200.0;  
        Double i4 = 200.0;  
  
        System.out.println(i1==i2);  
        System.out.println(i3==i4);  
    }  
}
```

输出结果为：

```
false  
false
```

原因很简单，在某个范围内的整型数值的个数是有限的，而浮点数却不是。

9. 为什么要有包装类型？

让基本数据类型也具有对象的特征

基本类型	包装器类型
boolean	Boolean
char	Character
int	Integer
byte	Byte
short	Short
long	Long
float	Float
double	Double

为了让基本类型也具有对象的特征，就出现了包装类型（如我们在使用集合类型Collection时就一定要使用包装类型而非基本类型）因为容器都是装object的，这是就需要这些基本类型的包装器类了。

自动装箱：`new Integer(6);`，底层调用：`Integer.valueOf(6)`

自动拆箱：`int i = new Integer(6);`，底层调用：`i.intValue()`方法实现。

```
Integer i = 6;
Integer j = 6;
System.out.println(i==j);
```

答案在下面这段代码中找：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

二者的区别：

1. 声明方式不同：基本类型不使用new关键字，而包装类型需要使用new关键字来在堆中分配存储空间；
2. 存储方式及位置不同：基本类型是直接将变量值存储在栈中，而包装类型是将对象放在堆中，然后通过引用来使用；
3. 初始值不同：基本类型的初始值如int为0，boolean为false，而包装类型的初始值为null；
4. 使用方式不同：基本类型直接赋值直接使用就好，而包装类型在集合如Collection、Map时会使用到。

10. `a=a+b`与`a+=b`有什么区别吗？

`+=`操作符会进行隐式自动类型转换，此处`a+=b`隐式的将加操作的结果类型强制转换为持有结果的类型，而`a=a+b`则不会自动进行类型转换。如：

```
byte a = 127;
byte b = 127;
b = a + b; // 报编译错误:cannot convert from int to byte
b += a;
```

以下代码是否有错，有的话怎么改？

```
short s1= 1;
s1 = s1 + 1;
```

有错误。short类型在进行运算时会自动提升为int类型，也就是说`s1+1`的运算结果是int类型，而`s1`是short类型，此时编译器会报错。

正确写法：

```
short s1= 1;
s1 += 1;
```

`+=` 操作符会对右边的表达式结果强转匹配左边的数据类型, 所以没错.

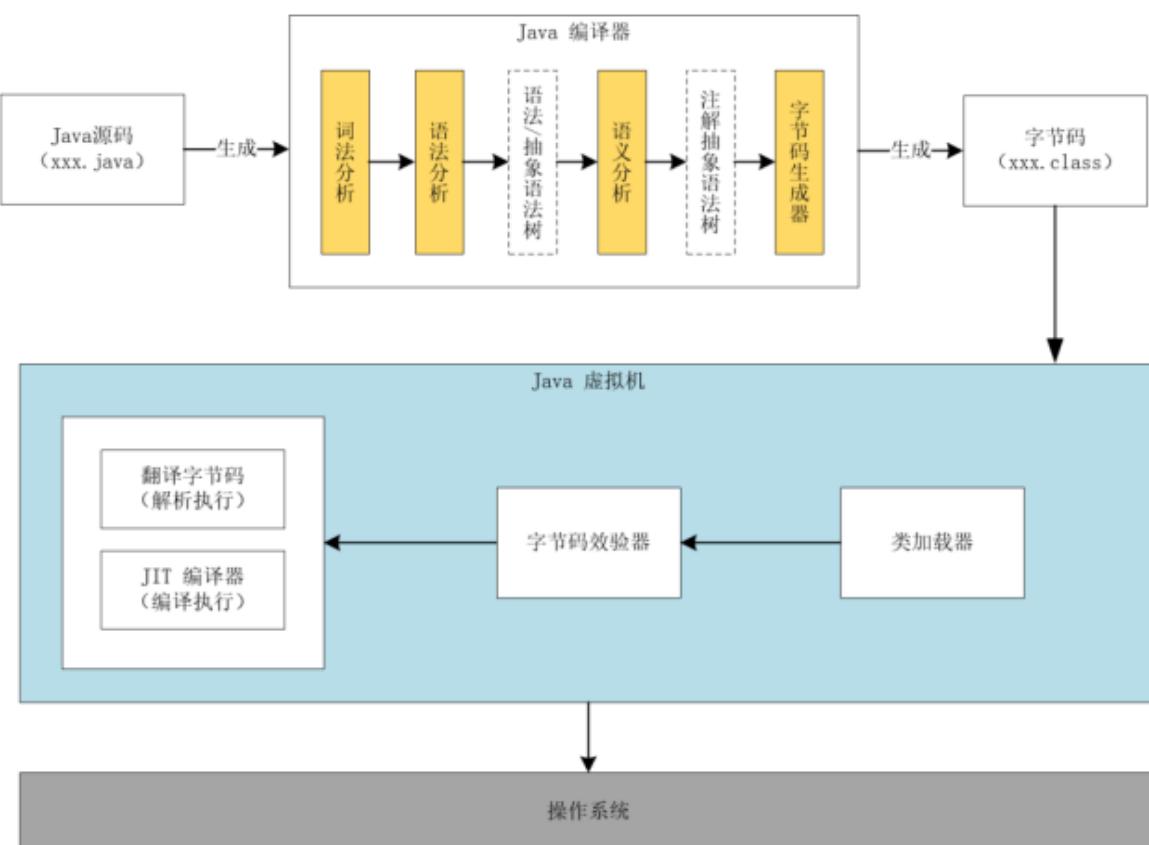
11. 能将 int 强制转换为 byte 类型的变量吗? 如果该值大于 byte 类型的范围, 将会出现什么现象?

我们可以做强制转换, 但是 Java 中 int 是 32 位的, 而 byte 是 8 位的, 所以, 如果强制转化, int 类型的高 24 位将会被丢弃, 因为 byte 类型的范围是从 -128 到 127

12. Java 程序是如何执行的

我们日常的工作中都使用开发工具 (IntelliJ IDEA 或 Eclipse 等) 可以很方便的调试程序, 或者是通过打包工具把项目打包成 jar 包或者 war 包, 放入 Tomcat 等 Web 容器中就可以正常运行了, 但你有没有想过 Java 程序内部是如何执行的? 其实不论是在开发工具中运行还是在 Tomcat 中运行, Java 程序的执行流程基本都是相同的, 它的执行流程如下:

- 先把 Java 代码编译成字节码, 也就是把 .java 类型的文件编译成 .class 类型的文件。这个过程的大致执行流程: Java 源代码 -> 词法分析器 -> 语法分析器 -> 语义分析器 -> 字符码生成器 -> 最终生成字节码, 其中任何一个节点执行失败就会造成编译失败;
- 把 class 文件放置到 Java 虚拟机, 这个虚拟机通常指的是 Oracle 官方自带的 Hotspot JVM;
- Java 虚拟机使用类加载器 (Class Loader) 装载 class 文件;
- 类加载完成之后, 会进行字节码效验, 字节码效验通过之后 JVM 解释器会把字节码翻译成机器码交由操作系统执行。但不是所有代码都是解释执行的, JVM 对此做了优化, 比如, 以 Hotspot 虚拟机来说, 它本身提供了 JIT (Just In Time) 也就是我们通常所说的动态编译器, 它能够在运行时将热点代码编译为机器码, 这个时候字节码就变成了编译执行。Java 程序执行流程图如下:



13. final 在 Java 中有什么作用?

final作为Java中的关键字可以用于三个地方。用于修饰类、类属性和类方法。

特征：凡是引用final关键字的地方皆不可修改！

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

14. final有哪些用法?

final也是很多面试喜欢问的地方,但我觉得这个问题很无聊,通常能回答下以下5点就不错了:

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可以被改变.如果修饰引用,那么表示引用不可变,引用指向的内容可变.
- 被final修饰的方法,JVM会尝试将其内联,以提高运行效率
- 被final修饰的常量,在编译阶段会存入常量池中.

除此之外,编译器对final域要遵守的两个重排序规则更好:

在构造函数内对一个final域的写入,与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序 初次读一个包含final域的对象的引用,与随后初次读这个final域,这两个操作之间不能重排序.

15. static都有哪些用法?

所有的人都知道static关键字这两个基本的用法:静态变量和静态方法.也就是被static所修饰的变量/方法都属于类的静态资源,类实例所共享.

除了静态变量和静态方法之外,static也用于静态块,多用于初始化操作:

```
public class PreCache{  
    static{  
        //执行相关操作  
    }  
}
```

此外static也多用于修饰内部类,此时称之为静态内部类.

最后一种用法就是静态导包,即 import static. import static是在JDK 1.5之后引入的新特性,可以用来指定导入某个类中的静态资源,并且不需要使用类名,可以直接使用资源名,比如:

```
import static java.lang.Math.*;  
  
public class Test{  
  
    public static void main(String[] args){  
        //System.out.println(Math.sin(20));传统做法  
        System.out.println(sin(20));  
    }  
}
```

16. static和final区别

关键词	修饰物	影响
final	变量	分配到常量池中，程序不可改变其值
final	方法	子类中将不能被重写
final	类	不能被继承
static	变量	分配在内存堆上，引用都会指向这一个地址而不会重新分配内存
static	方法块	虚拟机优先加载
static	类	可以直接通过类来调用而不需要new

17. 为什么有些java类要实现Serializable接口

为了网络进行传输或者持久化

什么是序列化

将对象的状态信息转换为可以存储或传输的形式的过程

除了实现Serializable接口还有什么序列化方式

- Json序列化
- FastJson序列化
- ProtoBuff序列化

18. 什么是java序列化，如何实现java序列化？或者请解释Serializable接口的作用。

我们有时候将一个java对象变成字节流的形式传出去或者从一个字节流中恢复成一个java对象，例如，要将java对象存储到硬盘或者传送给网络上的其他计算机，这个过程我们可以自己写代码去把一个java对象变成某个格式的字节流再传输。

但是，jre本身就提供了这种支持，我们可以调用 `OutputStream` 的 `writeObject` 方法来做，如果要让java帮我们做，要被传输的对象必须实现 `Serializable` 接口，这样，javac编译时就会进行特殊处理，编译的类才可以被 `writeObject` 方法操作，这就是所谓的序列化。需要被序列化的类必须实现 `Serializable` 接口，该接口是一个mini接口，其中没有需要实现方法，implements Serializable只是为了标注该对象是可被序列化的。

例如，在web开发中，如果对象被保存在了Session中，tomcat在重启时要把Session对象序列化到硬盘，这个对象就必须实现Serializable接口。如果对象要经过分布式系统进行网络传输，被传输的对象就必须实现Serializable接口。

19. 什么是内部类？内部类的作用

内部类的定义

将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类。

内部类的作用：

1、成员内部类 成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。

2、局部内部类 局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

3、匿名内部类 匿名内部类就是没有名字的内部类

4、静态内部类 指被声明为static的内部类，他可以不依赖内部类而实例化，而通常的内部类需要实例化外部类，从而实例化。静态内部类不可以有与外部类有相同的类名。不能访问外部类的普通成员变量，但是可以访问静态成员变量和静态方法（包括私有类型）一个静态内部类去掉static就是成员内部类，他可以自由的引用外部类的属性和方法，无论是静态还是非静态。但是不可以有静态属性和方法

20. Exception与Error包结构

Java可抛出(Throwable)的结构分为三种类型：**被检查的异常(CheckedException)**，**运行时异常(RuntimeException)**，**错误(Error)**。

1、运行时异常

定义： RuntimeException及其子类都被称为运行时异常。

特点： 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时，倘若既“没有通过throws声明抛出它”，也“没有用try-catch语句捕获它”，还是会编译通过。例如，除数为零时产生的ArithmaticException异常，数组越界时产生的IndexOutOfBoundsException异常，fail-fast机制产生的ConcurrentModificationException异常（java.util包下面的所有的集合类都是快速失败的，“快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException异常，从而产生fail-fast机制，这个错叫并发修改异常。Fail-safe，java.util.concurrent包下面的所有的类都是安全失败的，在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。）等，都属于运行时异常。

常见的五种运行时异常：

- `ClassCastException` (类转换异常)
- `IndexOutOfBoundsException` (数组越界)
- `NullPointerException` (空指针异常)
- `ArrayStoreException` (数据存储异常，操作数组是类型不一致)
- `BufferOverflowException`

2、被检查异常

定义：Exception类本身，以及Exception的子类中除了"运行时异常"之外的其它子类都属于被检查异常。

特点：Java编译器会检查它。此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException就属于被检查异常。

当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。被检查异常通常都是可以恢复的。如：

`IOException`

`FileNotFoundException`

`SQLException`

被检查的异常适用于那些不是因程序引起的错误情况，比如：读取文件时文件不存在引发的`FileNotFoundException`。然而，不被检查的异常通常都是由于糟糕的编程引起的，比如：在对象引用时没有确保对象非空而引起的`NullPointerException`。

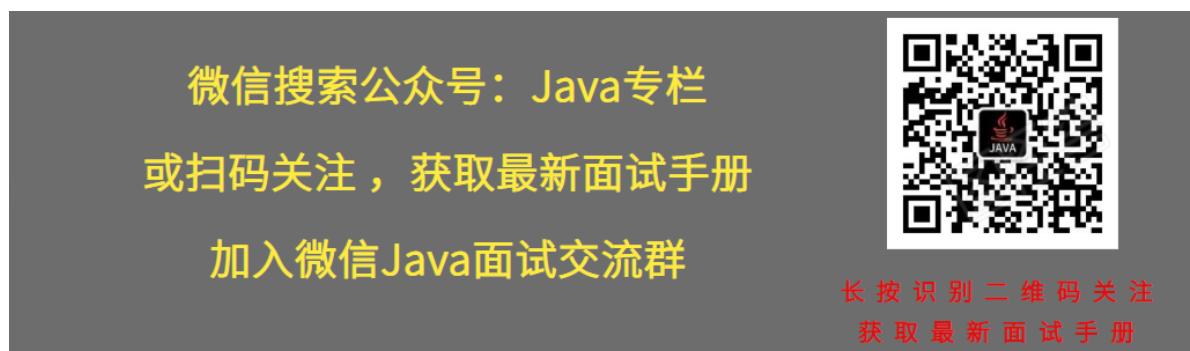
3、错误

定义：Error类及其子类。

特点：和运行时异常一样，编译器也不会对错误进行检查。

当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError就属于错误。出现这种错误会导致程序终止运行。OutOfMemoryError、ThreadDeath。

Java虚拟机规范规定JVM的内存分为了好几块，比如堆，栈，程序计数器，方法区等



21. try {}里有一个return语句，那么紧跟在这个try后的finally{}里的code会不会被执行，什么时候被执行，在return前还是后？

我们知道finally{}中的语句是一定会执行的，那么这个可能正常脱口而出就是return之前，return之后可能就出了这个方法了，鬼知道跑哪里去了，但更准确的应该是在return中间执行，请看下面程序代码的运行结果：

```
public classTest {  
    public static void main(String[] args) {  
        System.out.println(newTest().test());  
    }  
    static int test()  
    {  
        intx = 1;  
        try  
        {
```

```
    return x;
}
finally
{
    ++x;
}
}
```

执行结果如下：

```
1
```

运行结果是1，为什么呢？主函数调用子函数并得到结果的过程，好比主函数准备一个空罐子，当子函数要返回结果时，先把结果放在罐子里，然后再将程序逻辑返回到主函数。所谓返回，就是子函数说，我不运行了，你主函数继续运行吧，这没什么结果可言，结果是在说这话之前放进罐子里的。

22. 运行时异常与一般异常有何异同？

异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误。java编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。

23. error和exception有什么区别？

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。exception表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

24. 简单说说Java中的异常处理机制的简单原理和应用。

异常是指java程序运行时（非编译）所发生的非正常情况或错误，与现实生活中的事件很相似，现实生活中的事件可以包含事件发生的时间、地点、人物、情节等信息，可以用一个对象来表示，Java使用面向对象的方式来处理异常，它把程序中发生的每个异常也都分别封装到一个对象来表示的，该对象中包含有异常的信息。

Java对异常进行了分类，不同类型的异常分别用不同的Java类表示，所有异常的根类为 `java.lang.Throwable`。

`Throwable`下面又派生了两个子类：

- `Error`和`Exception`，`Error`表示应用程序本身无法克服和恢复的一种严重问题，程序只有奔溃了，例如，说内存溢出和线程死锁等系统问题。
- `Exception`表示程序还能够克服和恢复的问题，其中又分为系统异常和普通异常：

系统异常是软件本身缺陷所导致的问题，也就是软件开发人员考虑不周所导致的问题，软件使用者无法克服和恢复这种问题，但在这种问题下还可以让软件系统继续运行或者让软件挂掉，例如，数组脚本越界 (`ArrayIndexOutOfBoundsException`)，空指针异常 (`NullPointerException`)、类转换异常 (`ClassCastException`)；

普通异常是运行环境的变化或异常所导致的问题，是用户能够克服的问题，例如，网络断线，硬盘空间不够，发生这样的异常后，程序不应该死掉。

java为系统异常和普通异常提供了不同的解决方案，编译器强制普通异常必须try..catch处理或用throws声明继续抛给上层调用方法处理，所以普通异常也称为checked异常，而系统异常可以处理也可以不处理，所以，编译器不强制用try..catch处理或用throws声明，所以系统异常也称为unchecked异常。

25. == 和 equals 的区别是什么？

"=="

对于基本类型和引用类型 == 的作用效果是不同的，如下所示：

- 基本类型：比较的是值是否相同；
- 引用类型：比较的是引用是否相同；

```
String x = "string";
String y = "string";
String z = new String("string");
System.out.println(x==y); // true
System.out.println(x==z); // false
System.out.println(x.equals(y)); // true
System.out.println(x.equals(z)); // true
```

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

equals

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

首先来看默认情况下 equals 比较一个有相同值的对象，代码如下：

```
class Cat {
    public Cat(String name) {
        this.name = name;
    }

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

Cat c1 = new Cat("叶痕秋");
Cat c2 = new Cat("叶痕秋");
System.out.println(c1.equals(c2)); // false
```

微信搜索公众号：Java专栏，获取最新面试手册

输出结果出乎我们的意料，竟然是 false？这是怎么回事，看了 equals 源码就知道了，源码如下：

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

原来 equals 本质上就是 ==。

那问题来了，两个相同值的 String 对象，为什么返回的是 true？代码如下：

```
String s1 = new String("叶子");  
String s2 = new String("叶子");  
System.out.println(s1.equals(s2)); // true
```

同样的，当我们进入 String 的 equals 方法，找到了答案，代码如下：

```
public boolean equals(Object anObject) {  
    if (this == anObject) {  
        return true;  
    }  
    if (anObject instanceof String) {  
        String anotherString = (String)anObject;  
        int n = value.length;  
        if (n == anotherString.value.length) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = 0;  
            while (n-- != 0) {  
                if (v1[i] != v2[i])  
                    return false;  
                i++;  
            }  
            return true;  
        }  
    }  
    return false;  
}
```

原来是 String 重写了 Object 的 equals 方法，把引用比较改成了值比较。

总结

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

26. Hashcode的作用

java的集合有两类，一类是List，还有一类是Set。前者有序可重复，后者无序不重复。当我们在set中插入的时候怎么判断是否已经存在该元素呢，可以通过equals方法。但是如果元素太多，用这样的方法就会比较慢。

于是有人发明了哈希算法来提高集合中查找元素的效率。这种方式将集合分成若干个存储区域，每个对象可以计算出一个哈希码，可以将哈希码分组，每组分别对应某个存储区域，根据一个对象的哈希码就可以确定该对象应该存储的那个区域。

hashCode方法可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

27. 两个对象的 hashCode() 相同，那么 equals() 也一定为 true 吗？

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

代码示例：

```
String str1 = "keep";
String str2 = "brother";
System.out.println(String.format("str1: %d | str2: %d", str1.
hashCode(), str2.hashCode()));
System.out.println(str1.equals(str2));
```

执行结果：

```
str1: 1179395 | str2: 1179395
false
```

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

28. 泛型常用特点

泛型是Java SE 1.5之后的特性，《Java 核心技术》中对泛型的定义是：

“泛型”意味着编写的代码可以被不同类型的对象所重用。

“泛型”，顾名思义，“泛指的类型”。我们提供了泛指的概念，但具体执行的时候却可以有具体的规则来约束，比如我们用的非常多的ArrayList就是个泛型类，ArrayList作为集合可以存放各种元素，如Integer，String，自定义的各种类型等，但在我们使用的时候通过具体的规则来约束，如我们可以约束集合中只存放Integer类型的元素，如

```
List<Integer> initData = new ArrayList<>()
```

使用泛型的好处？

以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型的集合，如整型集合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，字符串型数据，而这并不是最重要的，因为我们只要把底层存储设置了Object即可，添加的数据全部都可向上转型为Object。更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。

29. 面向对象的特征

面向对象的编程语言有封装、继承、抽象、多态等4个主要的特征。

1. 封装：把描述一个对象的属性和行为的代码封装在一个模块中，也就是一个类中，属性用变量定义，行为用方法进行定义，方法可以直接访问同一个对象中的属性。
2. 抽象：把现实生活中的对象抽象为类。分为过程抽象和数据抽象
 - 数据抽象 -->鸟有翅膀,羽毛等(类的属性)
 - 过程抽象 -->鸟会飞,会叫(类的方法)
1. 继承：子类继承父类的特征和行为。子类可以有父类的方法，属性（非private）。子类也可以对父类进行扩展，也可以重写父类的方法。缺点就是提高代码之间的耦合性。
2. 多态：多态是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定（比如：向上转型，只有运行才能确定其对象属性）。方法覆盖和重载体现了多态性。

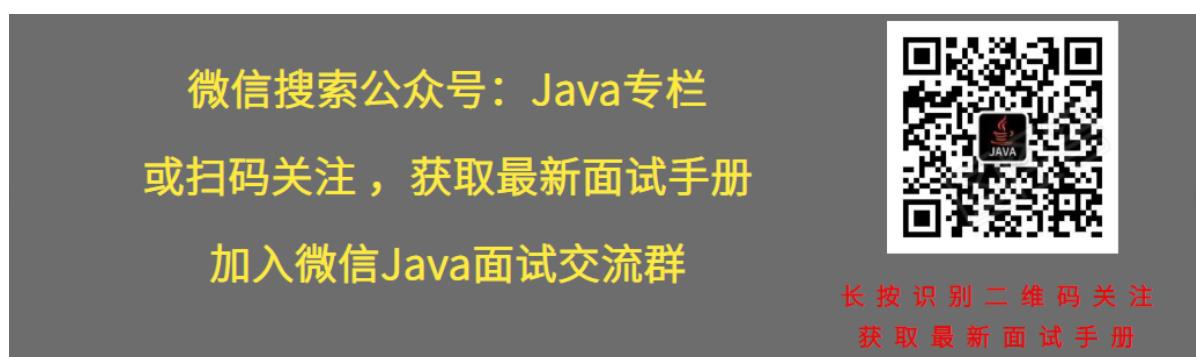
30. Java多态的理解

1. 多态是继封装、继承之后，面向对象的第三大特性。
2. 多态现实意义理解：
 - 现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三既是学生也是人，即出现两种形态。
 - Java作为面向对象的语言，同样可以描述一个事物的多种形态。如Student类继承了Person类，一个Student的对象便既是Student，又是Person。
1. 多态体现为父类引用变量可以指向子类对象。
2. 前提条件：必须有子父类关系。

注意：在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

1. 多态的定义与使用格式

定义格式：父类类型 变量名=new 子类类型();



31. 重载和重写的区别

重写(Override)

从字面上看，重写就是重新写一遍的意思。其实就是在子类中把父类本身有的方法重新写一遍。子类继承了父类原有的方法，但有时子类并不想原封不动的继承父类中的某个方法，所以在方法名，参数列表，返回类型(除过子类中方法的返回值是父类中方法返回值的子类时)都相同的情况下，对方法体进行修改或重写，这就是重写。但要注意子类函数的访问修饰权限不能少于父类的。

```
public class Father {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Son s = new Son();  
        s.sayHello();  
    }  
  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
}  
  
class Son extends Father{  
  
    @Override  
    public void sayHello() {  
        // TODO Auto-generated method stub  
        System.out.println("Hello by ");  
    }  
}
```

重写 总结：

- 1.发生在父类与子类之间
- 2.方法名，参数列表，返回类型（除过子类中方法的返回类型是父类中返回类型的子类）必须相同
- 3.访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）
- 4.重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常

重载 (Overload)

在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同甚至是参数顺序不同）则视为重载。同时，重载对返回类型没有要求，可以相同也可以不同，但不能通过返回类型是否相同来判断重载。

```
public class Father {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Father s = new Father();  
        s.sayHello();  
        s.sayHello("wintershii");  
    }  
}
```

```

    }

    public void sayHello() {
        System.out.println("Hello");
    }

    public void sayHello(String name) {
        System.out.println("Hello" + " " + name);
    }
}

```

重载 总结：

- 1.重载Overload是一个类中多态性的一种表现
- 2.重载要求同名方法的参数列表不同(参数类型, 参数个数甚至是参数顺序)
- 3.重载的时候, 返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准

33. Java创建对象有几种方式?

java中提供了以下四种创建对象的方式:

- new创建新对象
- 通过反射机制
- 采用clone机制
- 通过序列化机制

34. ConcurrentModificationException异常出现的原因

```

public class Test {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(2);
        Iterator<Integer> iterator = list.iterator();
        while(iterator.hasNext()){
            Integer integer = iterator.next();
            if(integer==2)
                list.remove(integer);
        }
    }
}

```

执行上段代码是有问题的, 会抛出 `ConcurrentModificationException` 异常。

原因: 调用 `list.remove()` 方法导致 `modCount` 和 `expectedModCount` 的值不一致。

```

final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}

```

解决办法：在迭代器中如果要删除元素的话，需要调用 `Iterator` 类的 `remove` 方法。

```
public class Test {  
    public static void main(String[] args) {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(2);  
        Iterator<Integer> iterator = list.iterator();  
        while(iterator.hasNext()) {  
            Integer integer = iterator.next();  
            if(integer==2)  
                iterator.remove(); //注意这个地方  
        }  
    }  
}
```

35. HashMap和HashTable、 ConcurrentHashMap区别？

相同点：

1. HashMap和Hashtable都实现了Map接口
2. 都可以存储key-value数据

不同点：

1. HashMap可以把null作为key或value, HashTable不可以
2. HashMap线程不安全，效率高。HashTable线程安全，效率低。
3. HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。

什么是fail-fast?

就是最快的时间能把错误抛出而不是让程序执行。

36. 如何保证线程安全又效率高？

Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

ConcurrentHashMap将整个Map分为N个segment(类似HashTable)，可以提供相同的线程安全，但是效率提升N倍，默认N为16。

37. 我们能否让HashMap同步？

HashMap可以通过下面的语句进行同步：

```
Map m = Collections.synchronizedMap(hashMap);
```

38. Java 中 IO 流分为几种?

按功能来分：输入流 (input) 、输出流 (output) 。

按类型来分：字节流和字符流。

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

39. BIO、NIO、AIO 有什么区别?

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

40. Files的常用方法都有哪些?

- Files. exists(): 检测文件路径是否存在。
- Files. createFile(): 创建文件。
- Files. createDirectory(): 创建文件夹。
- Files. delete(): 删除一个文件或目录。
- Files. copy(): 复制文件。
- Files. move(): 移动文件。
- Files. size(): 查看文件个数。
- Files. read(): 读取文件。
- Files. write(): 写入文件。

41. Java反射的作用于原理

1、定义：

反射机制是在运行时，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意个对象，都能够调用它的任意一个方法。在java中，只要给定类的名字，就可以通过反射机制来获得类的所有信息。

这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

2、哪里会用到反射机制？

jdbc就是典型的反射

```
Class.forName('com.mysql.jdbc.Driver.class');//加载MySQL的驱动类
```

这就是反射。如hibernate，struts等框架使用反射实现的。

42. 反射的实现方式

第一步：获取Class对象，有4种方法： 1) Class.forName("类的路径"); 2) 类名.class 3) 对象名.getClass() 4) 基本类型的包装类，可以调用包装类的Type属性来获得该包装类的Class对象

43. 实现Java反射的类：

1) Class: 表示正在运行的Java应用程序中的类和接口 注意：所有获取对象的信息都需要Class类来实现。 2) Field: 提供有关类和接口的属性信息，以及对它的动态访问权限。 3) Constructor: 提供关于类的单个构造方法的信息以及它的访问权限 4) Method: 提供类或接口中某个方法的信息

44. 反射机制的优缺点：

优点：

- 1、能够运行时动态获取类的实例，提高灵活性；
- 2、与动态编译结合

缺点：

- 1、使用反射性能较低，需要解析字节码，将内存中的对象进行解析。

解决方案：

- 1、通过setAccessible(true)关闭JDK的安全检查来提升反射速度；
 - 2、多次创建一个类的实例时，有缓存会快很多
 - 3、ReflectASM工具类，通过字节码生成的方式加快反射速度
- 2、相对不安全，破坏了封装性（因为通过反射可以获得私有方法和属性）

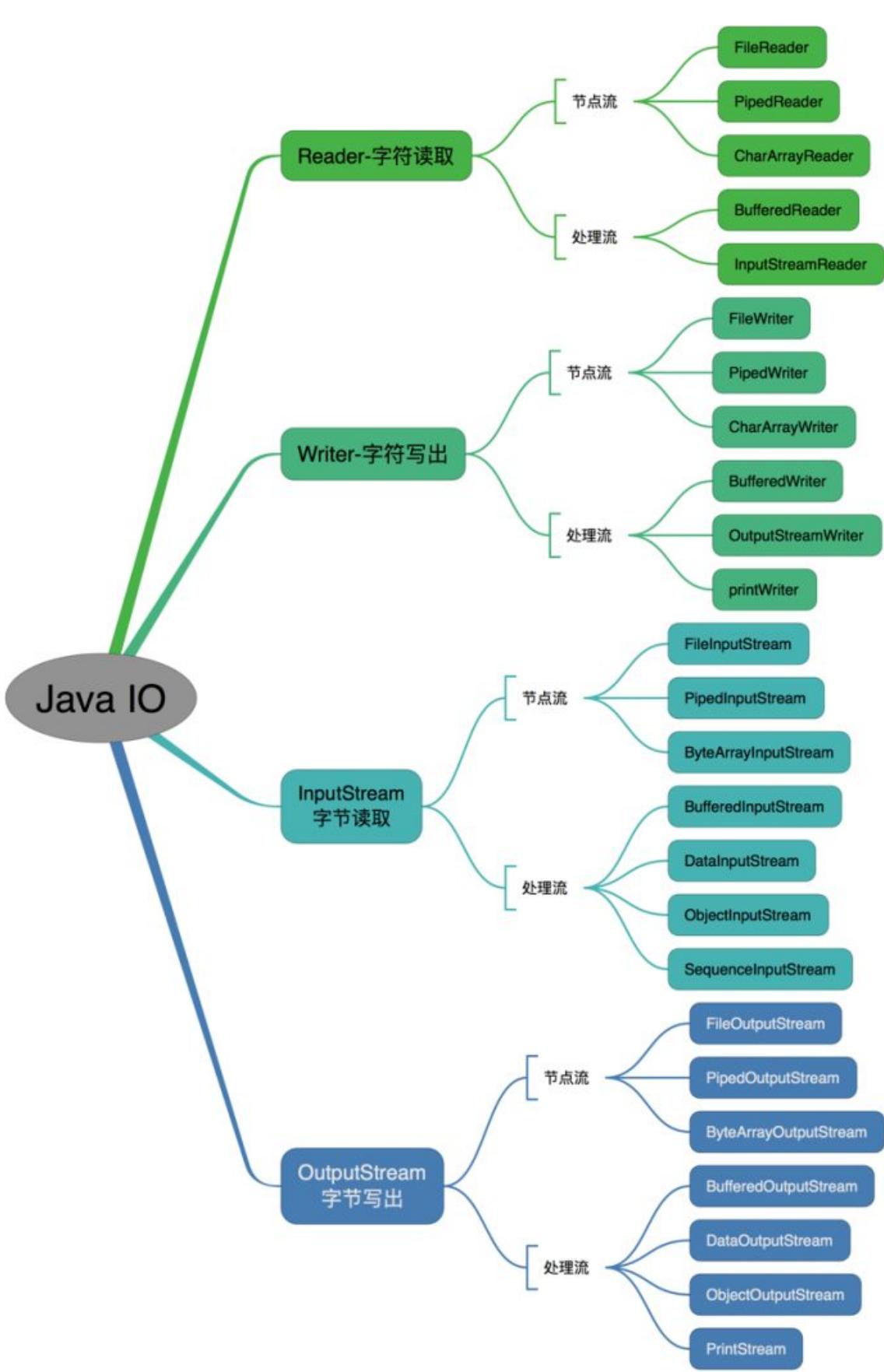
45. Java 中 IO 流分为几种？

- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- 按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java Io 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：



字符串&集合面试题汇总

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

1. Java 中操作字符串都有哪些类？它们之间有什么区别？

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

`String` 和 `StringBuffer`、`StringBuilder` 的区别在于 `String` 声明的是不可变的对象，每次操作都会生成新的 `String` 对象，然后将指针指向新的 `String` 对象。

而 `StringBuffer`、`StringBuilder` 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 `String`。

`StringBuffer` 和 `StringBuilder` 最大的区别在于，`StringBuffer` 是线程安全的，而 `StringBuilder` 是非线程安全的，但 `StringBuilder` 的性能却高于 `StringBuffer`，

所以在单线程环境下推荐使用 `StringBuilder`，多线程环境下推荐使用 `StringBuffer`。

2. String、StringBuffer和StringBuilder区别(类似上一题)

1、数据可变和不可变

- `String` 底层使用一个不可变的字符数组 `private final char value[]`；所以它内容不可变。
- `StringBuffer` 和 `StringBuilder` 都继承了 `AbstractStringBuilder` 底层使用的是可变字符数组：`char[] value;`

2、线程安全

- `StringBuilder` 是线程不安全的，效率较高；而 `StringBuffer` 是线程安全的，效率较低。

通过他们的 `append()` 方法来看，`StringBuffer` 是有同步锁，而 `StringBuilder` 没有：

```
@Override  
public synchronized StringBuffer append(Object obj) {  
    toStringCache = null;  
    super.append(String.valueOf(obj));  
    return this;  
}  
  
@Override  
public StringBuilder append(String str) {  
    super.append(str);  
    return this;  
}
```

3、相同点

`StringBuilder` 与 `StringBuffer` 有公共父类 `AbstractStringBuilder`。

最后，操作可变字符串速度：`StringBuilder > StringBuffer > String`，这个答案就显得不足为奇了。

4. `String str="i"` 与 `String str=new String("i")` 一样吗？

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

代码示例：

```
String x = "叶痕秋";
String y = "叶痕秋";
String z = new String("叶痕秋");
System.out.println(x == y); // true
System.out.println(x == z); // false
```

`String x = "叶痕秋"` 的方式，Java 虚拟机会将其分配到常量池中，而常量池中没有重复的元素，比如当执行“叶痕秋”时，java虚拟机会先在常量池中检索是否已经有“叶痕秋”，如果有那么就将“叶痕秋”的地址赋给变量，如果没有就创建一个，然后在赋给变量；

而 `String z = new String("叶痕秋")` 则会被分到堆内存中，即使内容一样还是会创建新的对象。

5. `String` 类的常用方法都有那些？

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

6. `String s = new String("xyz");` 创建了几个`StringObject`? 是否可以继承`String`类?

两个或一个都有可能，“xyz”对应一个对象，这个对象放在字符串常量缓冲区，常量“xyz”不管出现多少遍，都是缓冲区中的那一个。NewString每写一遍，就创建一个新的对象，它使用常量“xyz”对象的内容来创建出一个新String对象。如果以前就用过“xyz”，那么这里就不会创建“xyz”了，直接从缓冲区拿，这时创建了一个StringObject；但如果以前没有用过“xyz”，那么此时就会创建一个对象并放入缓冲区，这种情况它创建两个对象。至于String类是否继承，答案是否定的，因为String默认final修饰，是不可继承的。

7. 下面这条语句一共创建了多少个对象：String

```
s="a"+"b"+"c"+"d";
```

对于如下代码：

```
String s1 = "a";  
  
String s2 = s1 + "b";  
  
String s3 = "a" + "b";  
  
System.out.println(s2 == "ab");  
  
System.out.println(s3 == "ab");
```

第一条语句打印的结果为false，第二条语句打印的结果为true，这说明javac编译可以对字符串常量直接相加的表达式进行优化，不必要等到运行期再去进行加法运算处理，而是在编译时去掉其中的加号，直接将其编译成一个这些常量相连的结果。

题目中的第一行代码被编译器在编译时优化后，相当于直接定义了一个“abcd”的字符串，所以，上面的代码应该只创建了一个String对象。写如下两行代码，

```
String s ="a" + "b" +"c" + "d";  
  
System.out.println(s== "abcd");
```

最终打印的结果应该为true。

8. 简述Java中的集合

1. Collection下：List系(有序、元素允许重复)和Set系(无序、元素不重复)

set根据equals和hashcode判断，一个对象要存储在Set中，必须重写equals和hashCode方法

2. Map下：HashMap线程不同步；TreeMap线程同步

3. Collection系列和Map系列：Map是对Collection的补充，两个没什么关系

9. List、Map、Set三个接口，存取元素时，各有什么特点？

首先，List与Set具有相似性，它们都是单列元素的集合，所以，它们有一个共同的父接口，叫Collection。

1、Set里面不允许有重复的元素

即不能有两个相等（注意，不是仅仅是相同）的对象，即假设Set集合中有了一个A对象，现在我要向Set集合再存入一个B对象，但B对象与A对象equals相等，则B对象存储不进去，所以，Set集合的add方法有一个boolean的返回值，当集合中没有某个元素，此时add方法可成功加入该元素时，则返回true，当集合含有与某个元素equals相等的元素时，此时add方法无法加入该元素，返回结果为false。Set取元素时，不能细说要取第几个，只能以Iterator接口取得所有的元素，再逐一遍历各个元素。

2、List表示有先后顺序的集合

注意，不是那种按年龄、按大小、按价格之类的排序。当我们多次调用add(Obj)方法时，每次加入的对象就像火车站买票有排队顺序一样，按先来后到的顺序排序。有时候，也可以插队，即调用add(int index, Obj e)方法，就可以指定当前对象在集合中的存放位置。一个对象可以被反复存储进List中，每调用一次add方法，这个对象就被插入进集合中一次，其实，并不是把这个对象本身存储进了集合中，而是在集合中用一个索引变量指向这个对象，当这个对象被add多次时，即相当于集合中有多个索引指向了这个对象。List除了可以用Iterator接口取得所有的元素，再逐一遍历各个元素之外，还可以调用get(index i)来明确说明取第几个。

3、Map与List和Set不同

它是双列的集合，其中有put方法，定义如下：put(obj key,obj value)，每次存储时，要存储一对key/value，不能存储重复的key，这个重复的规则也是按equals比较相等。取则可以根据key获得相应的value，即get(Object key)返回值为key所对应的value。另外，也可以获得所有的key的结合，还可以获得所有的value的结合，还可以获得key和value组合成的Map.Entry对象的集合。

总结

List以特定次序来持有元素，可有重复元素。Set无法拥有重复元素，内部排序。Map保存key-value值，value可多值。



10. Set里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用==还是equals()？它们有何区别？

Set里的元素是不能重复的，元素重复与否是使用equals()方法进行判断的。

==和equal区别也是考烂了的题，这里再重复说一下：

==操作符专门用来比较两个变量的值是否相等，也就是用于比较变量所对应的内存中所存储的数值是否相同，要比较两个基本类型的数据或两个引用变量是否相等，只能用==操作符。

equals方法是用于比较两个独立对象的内容是否相同，就好比去比较两个人的长相是否相同，它比较的两个对象是独立的。

比如：两条new语句创建了两个对象，然后用a/b这两个变量分别指向了其中一个对象，这是两个不同的对象，它们的首地址是不同的，即a和b中存储的数值是不相同的，所以，表达式a==b将返回false，而这两个对象中的内容是相同的，所以，表达式a.equals(b)将返回true。

11. ArrayList和LinkedList区别？

1. ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2. 对于随机访问get和set，ArrayList优于LinkedList，因为LinkedList要移动指针。
3. 对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

12. ArrayList和Vector的区别

两个类都实现了List接口（List接口继承了collection接口），**他们都是有序集合**，即存储在这两个集合中的元素的位置都是有顺序的，相当于一种动态的数组，我们以后可以按位置索引号取出某个元素，并且其中的数据是允许重复的，这是与HashSet之类的集合的最大不同处，HashSet之类的集合不可以按索引号去检索其中的元素，也不允许有重复的元素。

ArrayList与Vector的区别主要包括两个方面：

同步性：

vector是线程安全的，也就是说它的方法之间是线程同步的，而ArrayList是线程不安全的，它的方法之间是线程不同的。如果只有一个线程会访问到集合，那最好是使用ArrayList，因为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用vector，因为不需要我们自己再去考虑和编写线程安全的代码。

数据增长：

ArrayList与vector都有一个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加ArrayList与vector的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要取得一定的平衡。

vector默认增长为原来两倍，而ArrayList的增长策略在文档中没有明确规定（从源代码看到的是增长为原来的1.5倍）。ArrayList与vector都可以设置初始的空间大小，vector还可以设置增长的空间大小，而ArrayList没有提供设置增长空间的方法。

总结：即Vector增长原来的一倍，ArrayList增加原来的0.5倍。

13. ArrayList,Vector,LinkedList的存储性能和特性

ArrayList和vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，

vector由于使用了synchronized方法（线程安全），通常性能上较ArrayList差。而LinkedList使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，索引就变慢了，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

LinkedList也是线程不安全的，LinkedList提供了一些方法，使得LinkedList可以被当作堆栈和队列来使用。

14. HashMap和Hashtable的区别

HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口，

主要区别在于HashMap允许空(null)键值(key)，由于非线程安全，在只有一个线程访问的情况下，效率要高于Hashtable。

HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

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

Hashtable继承自Dictionary类，而HashMap是Java1.2引进的Map interface的一个实现。

最大的不同是，`Hashtable`的方法是 `synchronize` 的，而 `HashMap` 不是，在多个线程访问 `Hashtable` 时，不需要自己为它的方法实现同步，而 `HashMap` 就必须为之提供同步。

就 `HashMap` 与 `HashTable` 主要从三方面来说。

- 历史原因: `Hashtable` 是基于陈旧的 `Dictionary` 类的，`HashMap` 是 Java 1.2 引进的 Map 接口的一个实现
- 同步性: `Hashtable` 是线程安全的，也就是说是同步的，而 `HashMap` 是线程不安全的，不是同步的
- 值: 只有 `HashMap` 可以让你将空值作为一个表的条目的 key 或 value

15. Java 中的同步集合与并发集合有什么区别？

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java 1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java 5 介绍了并发集合 `ConcurrentHashMap`，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

不管是同步集合还是并发集合他们都支持线程安全，他们之间主要的区别体现在性能和可扩展性，还有他们如何实现的线程安全上。

同步 `HashMap`, `Hashtable`, `HashSet`, `Vector`, `ArrayList` 相比他们并发的实现

(`ConcurrentHashMap`, `CopyOnWriteArrayList`, `CopyOnwriteHashSet`) 会慢得多。造成如此慢的主要原因是锁，同步集合会把整个 Map 或 List 锁起来，而并发集合不会。并发集合实现线程安全是通过使用先进的和成熟的技术像锁剥离。

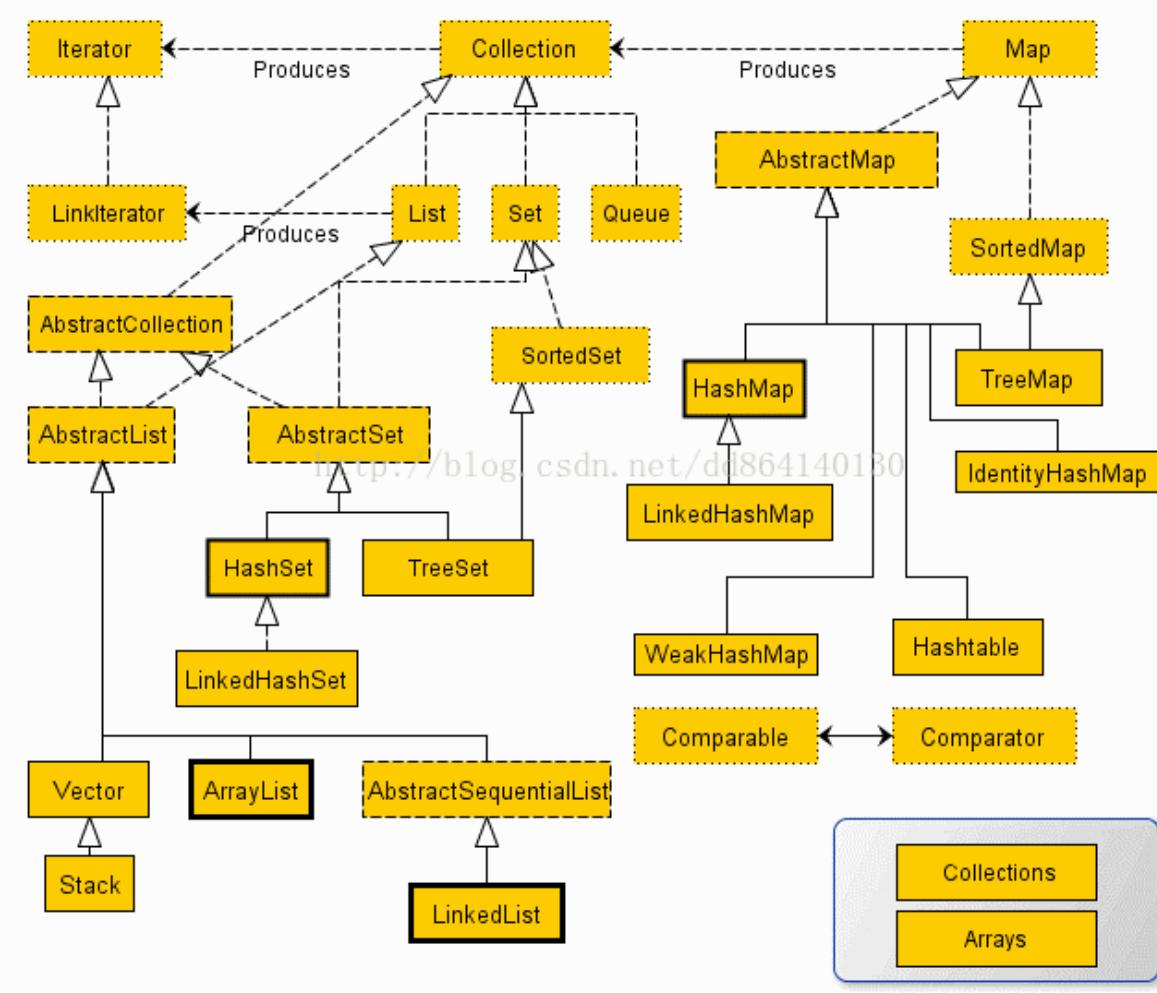
比如 `ConcurrentHashMap` 会把整个 Map 划分成几个片段，只对相关的几个片段上锁，同时允许多线程访问其他未上锁的片段。

同样的，`CopyOnwriteArrayList` 允许多个线程以非同步的方式读，当有线程写的时候它会将整个 List 复制一个副本给它。

如果在读多写少这种对并发集合有利的条件下使用并发集合，这会比使用同步集合更具有可伸缩性。

16. Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的，尤其是对我们经常使用的 List, Map 的原理更该如此。这里我们看这张图即可：



17. poll()方法和remove()方法区别?

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

18. LinkedHashMap和PriorityQueue的区别

PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

19. WeakHashMap与HashMap的区别是什么?

WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key 对象没有任何引用时，key/value 将会被回收。

20. ArrayList和LinkedList的区别?

最明显的区别是 ArrayList底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。

21. ArrayList和Array有什么区别?

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

ArrayList 是Java集合框架类的一员,可以称它为一个动态数组. array 是静态的,所以一个数据一旦创建就无法更改他的大小

22. ArrayList和HashMap默认大小?

在 java 7 中，ArrayList 的默认大小是 10 个元素，HashMap 的默认大小是16个元素（必须是2的幂）。这就是 Java 7 中 ArrayList 和 HashMap 类的代码片段

```
private static final int DEFAULT_CAPACITY = 10;  
  
//from HashMap.java JDK 7  
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

23. Comparator和Comparable的区别?

相同点

都是用于比较两个对象“顺序”的接口

都可以使用Collections.sort()方法来对对象集合进行排序

不同点

Comparable位于java.lang包下，而Comparator则位于java.util包下

Comparable 是在集合内部定义的方法实现的排序， Comparator 是在集合外部实现的排序

总结

使用Comparable接口来实现对象之间的比较时，可以使这个类型（设为A）实现Comparable接口，并可以使用Collections.sort()方法来对A类型的List进行排序，之后可以通过a1.compareTo(a2)来比较两个对象；

当使用Comparator接口来实现对象之间的比较时，只需要创建一个实现Comparator接口的比较器（设为AComparator），并将其传给Collections.sort()方法即可对A类型的List进行排序，之后也可以通过调用比较器AComparator.compare(a1, a2)来比较两个对象。

可以说一个是自己完成比较，一个是外部程序实现比较的差别而已。

用 Comparator 是策略模式 (strategy design pattern)，就是不改变对象自身，而用一个策略对象 (strategy object) 来改变它的行为。

比如：你想对整数采用绝对值大小来排序，Integer是不符合要求的，你不需要去修改Integer类（实际上你也不能这么做）去改变它的排序行为，这时候只要（也只有）使用一个实现了Comparator接口的对象来实现控制它的排序就行了。

两种方式，各有各的特点：使用Comparable方式比较时，我们将比较的规则写入了比较的类型中，其特点是高内聚。但如果哪天这个规则需要修改，那么我们必须修改这个类型的源代码。如果使用Comparator方式比较，那么我们不需要修改比较的类，其特点是易维护，但需要自定义一个比较器，后续比较规则的修改，仅仅是改这个比较器中的代码即可。

24. 如何实现集合排序？

你可以使用有序集合，如TreeSet或TreeMap，你也可以使用有顺序的集合，如list，然后通过Collections.sort()来排序。

25. 如何打印数组内容

你可以使用Arrays.toString()和Arrays.deepToString()方法来打印数组。由于数组没有实现toString()方法，所以如果将数组传递给System.out.println()方法，将无法打印出数组的内容，但是Arrays.toString()可以打印每个元素。

26. LinkedList的是单向链表还是双向？

双向循环列表，具体实现自行查阅源码。

27. TreeMap是实现原理

TreeMap是一个通过红黑树实现有序的key-value集合。

TreeMap继承AbstractMap，也即实现了Map，它是一个Map集合

TreeMap实现了NavigableMap接口，它支持一系列的导航方法，

TreeMap实现了Cloneable接口，它可以被克隆

TreeMap本质是Red-Black Tree，它包含几个重要的成员变量：root、size、comparator。其中root是红黑树的根节点。它是Entry类型，Entry是红黑树的节点，它包含了红黑树的6个基本组成：key、value、left、right、parent和color。Entry节点根据根据Key排序，包含的内容是value。Entry中key比较大小是根据比较器comparator来进行判断的。size是红黑树的节点个数。

28. 遍历ArrayList时如何正确移除一个元素

错误写法示例一：

```
public static void remove(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        String s = list.get(i);  
        if (s.equals("bb")) {  
            list.remove(s);  
        }  
    }  
}
```

错误写法示例二：

```
public static void remove(ArrayList<String> list) {  
    for (String s : list) {  
        if (s.equals("bb")) {  
            list.remove(s);  
        }  
    }  
}
```

要分析产生上述错误现象的原因唯有翻一翻jdk的ArrayList源码，先看下ArrayList中的remove方法（注意ArrayList中的remove有两个同名方法，只是入参不同，这里看的是入参为Object的remove方法）是怎么实现的：

```
public boolean remove(Object o) {  
    if (o == null) {  
        for (int index = 0; index < size; index++)  
            if (elementData[index] == null)  
                fastRemove(index);  
        return true;  
    }  
    } else {  
        for (int index = 0; index < size; index++)  
            if (o.equals(elementData[index])) {  
                fastRemove(index);  
                return true;  
            }  
    }  
    return false;  
}
```

按一般执行路径会走到else路径下最终调用faseRemove方法：

```
private void fastRemove(int index) {  
    modCount++;  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData, index,  
                         numMoved);  
    elementData[--size] = null; // Let gc do its work  
}
```

可以看到会执行System.arraycopy方法，导致删除元素时涉及到数组元素的移动。针对错误写法一，在遍历第二个元素字符串bb时因为符合删除条件，所以将该元素从数组中删除，并且将后一个元素移动（也是字符串bb）至当前位置，导致下一次循环遍历时后一个字符串bb并没有遍历到，所以无法删除。针对这种情况可以倒序删除的方式来避免：

```
public static void remove(ArrayList<String> list) {  
    for (int i = list.size() - 1; i >= 0; i--) {  
        String s = list.get(i);  
        if (s.equals("bb")) {  
            list.remove(s);  
        }  
    }  
}
```

因为数组倒序遍历时即使发生元素删除也不影响后序元素遍历。

而错误二产生的原因却是foreach写法是对实际的Iterable、hasNext、next方法的简写，问题同样处在上文的fastRemove方法中，可以看到第一行把modCount变量的值加一，但在ArrayList返回的迭代器（该代码在其父类AbstractList中）：

```
public Iterator<E> iterator() {  
    return new Itr();  
}
```

这里返回的是AbstractList类内部的迭代器实现private class Itr implements Iterator，看这个类的next方法：

```
public E next() {  
    checkForComodification();  
    try {  
        E next = get(cursor);  
        lastRet = cursor++;  
        return next;  
    } catch (IndexOutOfBoundsException e) {  
        checkForComodification();  
        throw new NoSuchElementException();  
    }  
}
```

第一行checkForComodification方法：

```
final void checkForComodification() {  
    if (modCount != expectedModCount)  
        throw new ConcurrentModificationException();  
}
```

这里会做迭代器内部修改次数检查，因为上面的remove(Object)方法把修改了modCount的值，所以才会报出并发修改异常。要避免这种情况的出现则在使用迭代器迭代时（显示或foreach的隐式）不要使用ArrayList的remove，改为用Iterator的remove即可。

```
public static void remove(ArrayList<String> list) {  
    Iterator<String> it = list.iterator();  
    while (it.hasNext()) {  
        String s = it.next();  
        if (s.equals("bb")) {  
            it.remove();  
        }  
    }  
}
```

29. HashMap的实现原理

HashMap是基于哈希表实现的map，哈希表（也叫关联数组）一种通用的数据结构，是Java开发者常用的类，常用来存储和获取数据，功能强大使用起来也很方便，是居家旅行...不对，是Java开发需要掌握的基本技能，也是面试必考的知识点，所以，了解HashMap是很有必要的。

原理

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

内部存储结构

HashMap类实现了Map< K, V>接口，主要包含以下几个方法：

- V put(K key, V value)
- V get(Object key)
- V remove(Object key)
- Boolean containsKey(Object key)

HashMap使用了一个内部类Node< K, V>来存储数据

我阅读的是Java 8的源码，在Java 8之前存储数据的内部类是Entry< K, V>，代码大体都是一样的

Node代码：

```
static class Node<K,V> implements Map.Entry<K,V> {  
    final int hash;  
    final K key;  
    V value;  
    Node<K,V> next;  
    ...  
}
```

可以看见Node类中除了键值对 (key-value) 以外，还有额外的两个数据：

- hash : 这个是通过计算得到的散列值
- next: 指向另一个Node，这样HashMap可以像链表一样存储数据

因此可以知道，HashMap的结构大致如下：

我们可以将每个横向看成一个个的桶，每个桶中存放着具有相同Hash值的Node,通过一个list来存放每个桶。

内部变量

微信搜索公众号：Java专栏，获取最新面试手册

```

// 默认容量大小
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
// 最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;
// 装载因子
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 转换为二叉树的阀值
static final int TREEIFY_THRESHOLD = 8;
// 转换为二叉树的最低阀值
static final int UNTREEIFY_THRESHOLD = 6;
// 二叉树最小容量
static final int MIN_TREEIFY_CAPACITY = 64;
// 哈希表
transient Node<K,V>[] table;
// 键值对的数量
transient int size;
// 记录HashMap结构改变次数，与HashMap的快速失败相关
transient int modCount;
// 扩容的阀值
int threshold;
// 装载因子
final float loadFactor;

```

常用方法

put操作

put函数大致的思路为：

1. 对key的hashCode()做hash，然后再计算index；
2. 如果没碰撞直接放到bucket里；
3. 如果碰撞了，以链表的形式存在buckets后；
4. 如果碰撞导致链表过长(大于等于TREEIFY_THRESHOLD)，就把链表转换成红黑树；
5. 如果节点已经存在就替换old value(保证key的唯一性)
6. 如果bucket满了(超过load factor*current capacity)，就要resize。

```

public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length; // resize()是调整table数组大小的，如果table数组
        // 为空或长度为0，重新调整大小
    if ((p = tab[i = (n - 1) & hash]) == null) // i = (n - 1) & hash | 这里计算出来的
        // i值就是存放数组的位置，如果当前位置为空，则直接放入其中
        tab[i] = newNode(hash, key, value, null);
    else { // hash冲突
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k)))) // 如果hash相
            // 同，并且key值也相同，则找到存放位置
            e = p;
    }
}

```

```

    else if (p instanceof TreeNode) // 如果当前p是二叉树，则放入二叉树中
        e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
    else { // 存放到链表中
        for (int binCount = 0; ; ++binCount) {
            if ((e = p.next) == null) { // 遍历链表并将值放到链表最后
                p.next = newNode(hash, key, value, null);
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    treeifyBin(tab, hash); // 如果链表中的值大于
TREEIFY_THRESHOLD - 1，则将链表转换成二叉树
                break;
            }
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                break;
            p = e;
        }
    }
    if (e != null) { // 表示对于当前key早已经存在
        v oldValue = e.value;
        if (!onlyIfAbsent || oldValue == null) // 如果onlyIfAbsent为false或则
oldValue为空，替换原来的值
            e.value = value;
        afterNodeAccess(e);
        return oldValue; // 返回原来的值
    }
}
++modCount; // HashMap结构修改次数，主要用于判断迭代器中fail-fast
if (++size > threshold) // 如果++size后的值比阀值大，则重新调整大小
    resize();
afterNodeInsertion(evict);
return null;
}

```

代码也比较容易看懂，值得注意的就是

```

else if (p instanceof TreeNode) // 如果当前p是二叉树，则放入二叉树中
    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

```

与

```

if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
    treeifyBin(tab, hash); // 如果链表中的值大于TREEIFY_THRESHOLD - 1，则将链表转换成二
叉树

```

这是Java 8相对于以前版本一个比较大的改变。

在Java 8以前，每次产生hash冲突，就将记录追加到链表后面，然后通过遍历链表来查找。如果某个链表中记录过大，每次遍历的数据就越多，效率也就很低，复杂度为O(n)；

在Java 8中，加入了一个常量TREEIFY_THRESHOLD=8，如果某个链表中的记录大于这个常量的话，HashMap会动态的使用一个专门的treemap实现来替换掉它。这样复杂度是O(logn)，比链表的O(n)会好很多。

对于前面产生冲突的那些KEY对应的记录只是简单的追加到一个链表后面，这些记录只能通过遍历来进行查找。但是超过这个阈值后HashMap开始将列表升级成一个二叉树，使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap希望key值最好是实现了Comparable接口的，这样它可以按照顺序来进行插入。

get操作

在理解了put之后，get就很简单了。大致思路如下：

1. bucket里的第一个节点，直接命中；
2. 如果有冲突，则通过key.equals(k)去查找对应的entry
3. 若为树，则在树中通过key.equals(k)查找，O(logn)；
4. 若为链表，则在链表中通过key.equals(k)查找，O(n)。

```
public V get(Object key) {  
    Node<K,V> e;  
    return (e = getNode(hash(key), key)) == null ? null : e.value;  
}  
  
final Node<K,V> getNode(int hash, Object key) {  
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;  
    if ((tab = table) != null && (n = tab.length) > 0 &&  
        (first = tab[(n - 1) & hash]) != null) {  
        if (first.hash == hash && // always check first node  
            ((k = first.key) == key || (key != null && key.equals(k)))) // 如果  
hash相同并且key值一样则返回当前node  
            return first;  
        if ((e = first.next) != null) {  
            if (first instanceof TreeNode) // 如果当前node为二叉树，则在二叉树中查找  
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);  
            do { // 遍历链表  
                if (e.hash == hash &&  
                    ((k = e.key) == key || (key != null && key.equals(k))))  
                    return e;  
            } while ((e = e.next) != null);  
        }  
    }  
    return null;  
}
```

30. HashMap自动扩容

如果在初始化HashMap中没有指定初始容量，那么默认容量为16，但是如果后来HashMap中存放的数量超过了16，那么便会有大量的hash冲突；在HashMap中有自动扩容机制，如果当前存放的数量大于某个界限，HashMap便会调用resize()方法，扩大HashMap的容量。

当hashmap中的元素个数超过数组大小 $loadFactor$ 时，就会进行数组扩容， $loadFactor$ 的默认值为0.75，也就是说，默认情况下，数组大小为16，那么当hashmap中元素个数超过 $16 \times 0.75 = 12$ 的时候，就把数组的大小扩展为 $2 \times 16 = 32$ ，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能。

HashMap的capacity必须满足是2的N次方,如果在构造函数内指定的容量n不满足,HashMap会通过下面的算法将其转换为大于n的最小的2的N次方数。

```
// 减1→移位→按位或运算→加1返回
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```



31. HashMap线程安全吗?

HashMap是非线程安全的, 如果在多线程环境下, 可以使用HashTable, HashTable中所有CRUD操作都是线程同步的, 同样的, 线程同步的代价就是效率变低了。

再Java 5以后, 有了一个线程安全的HashMap——ConcurrentHashMap, ConcurrentHashMap相对于HashTable来说, ConcurrentHashMap将hash表分为16个桶(默认值), 诸如get, put, remove等常用操作只锁当前需要用到的桶。试想, 原来只能一个线程进入, 现在却能同时16个写线程进入(写线程才需要锁定, 而读线程几乎不受限制, 并发性的提升是显而易见)。

快速失败(fast-fail)

“快速失败”也就是fail-fast, 它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时, 有可能会产生fail-fast机制。记住是有可能, 而不是一定。例如: 假设存在两个线程(线程1、线程2), 线程1通过Iterator在遍历集合A中的元素, 在某个时候线程2修改了集合A的结构(是结构上面的修改, 而不是简单的修改集合元素的内容), 那么这个时候程序就会抛出ConcurrentModificationException异常, 从而产生fail-fast机制。

在HashMap的forEach方法中有以下代码:

```
@Override
public void forEach(BiConsumer<? super K, ? super V> action) {
    Node<K,V>[] tab;
    if (action == null)
        throw new NullPointerException();
    if (size > 0 && (tab = table) != null) {
        int mc = modCount;
        for (int i = 0; i < tab.length; ++i) {
            for (Node<K,V> e = tab[i]; e != null; e = e.next)
                action.accept(e.key, e.value);
```

```
        }
        if (modCount != mc)
            throw new ConcurrentModificationException();
    }
}
```

在上面我们说到，modCount是记录每次HashMap结构修改。forEach方法会在进入for循环之前，将modCount赋值给mc，如果在for循环之后，HashMap的结构变化了，那么导致的结果就是modCount != mc，则抛出ConcurrentModificationException()异常。

32. HashMap总结

- 1、什么时候会使用HashMap？他有什么特点？是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。
- 2、你知道HashMap的工作原理吗？通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Factor则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，HashMap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。
- 3、你知道get和put的原理吗？equals()和hashCode()的都有什么作用？通过对key的hashCode()进行hashing，并计算下标(n-1 & hash)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点
- 4、你知道hash的实现吗？为什么要这样实现？在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。
- 5、如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？如果超过了负载因子(默认0.75)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

前段时间因为找工作的缘故背了一些关于HashMap的面试题，死记硬背，也不是很懂，最近看了源码，很多知识才变的清晰，而且看源码挺有趣的。再接再厉。

33. Java集合框架是什么？说出一些集合框架的优点？

每种编程语言中都有集合。集合框架的部分优点如下：

- 1、使用核心集合类降低开发成本，而非实现我们自己的集合类。
- 2、随着使用经过严格测试的集合框架类，代码质量会得到提高。
- 3、通过使用JDK附带的集合类，可以降低代码维护成本。
- 4、复用性和可操作性。

34. 集合框架中的泛型有什么优点？

Java1.5引入了泛型，所有的集合接口和实现都大量地使用它。

泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和instanceOf操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。

35. Java集合框架的基础接口有哪些？

Collection为集合层级的根接口。一个集合代表一组对象，这些对象即为它的元素。Java平台不提供这个接口任何直接的实现。

Set是一个不能包含重复元素的集合。这个接口对数学集合抽象进行建模，被用来代表集合，就如一副牌。

List是一个有序集合，可以包含重复元素。你可以通过它的索引来访问任何元素。List更像长度动态变换的数组。

Map是一个将key映射到value的对象。一个Map不能包含重复的key：每个key最多只能映射一个value。

一些其它的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

36. 为何Collection不从Cloneable和Serializable接口继承？

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

37. 为何Map接口不继承Collection接口？

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

38. Iterator是什么？

Iterator接口提供遍历任何Collection的接口。我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。

39. Iterator和ListIterator的区别是什么？

下面列出了他们的区别：Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

40. Enumeration和Iterator接口的区别?

Enumeration速度是Iterator的2倍，同时占用更少的内存。但是，Iterator远远比Enumeration安全，因为其他线程不能够修改正在被iterator遍历的集合里面的对象。同时，Iterator允许调用者删除底层集合里面的元素，这对Enumeration来说是不可能的。

41. 为何没有像Iterator.add()这样的方法，向集合中添加元素？

语义不明，已知的是，Iterator的协议不能确保迭代的次序。然而要注意，ListIterator没有提供一个add操作，它要确保迭代的顺序。

42. 为何迭代器没有一个方法可以直接获取下一个元素，而不需要移动游标？

它可以在当前Iterator的顶层实现，但是它用得很少，如果将它加到接口中，每个继承都要去实现它，这没有意义。

43. Iterator和ListIterator之间有什么区别？

- 1、我们可以使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。
- 2、Iterator只可以向前遍历，而ListIterator可以双向遍历。
- 3、ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

44. 遍历一个List有哪些不同的方式？

```
List<String> strList = new ArrayList<>();
//使用for-each循环
for(String obj : strList){
    System.out.println(obj);
}
//using iterator
Iterator<String> it = strList.iterator();
while(it.hasNext()){
    String obj = it.next();
    System.out.println(obj);
}
```

使用迭代器更加线程安全，因为它可以确保，在当前遍历的集合元素被更改的时候，它会抛出ConcurrentModificationException。

45. 通过迭代器fail-fast属性，你明白了什么？

每次我们尝试获取下一个元素的时候，Iterator fail-fast属性检查当前集合结构里的任何改动。如果发现任何改动，它抛出ConcurrentModificationException。Collection中所有Iterator的实现都是按fail-fast来设计的（ConcurrentHashMap和CopyOnWriteArrayList这类并发集合类除外）。

46. fail-fast与fail-safe有什么区别？

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有集合类都是快速失败的，而java.util.concurrent包下面的所有类都是安全失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。

47. 在迭代一个集合的时候，如何避免ConcurrentModificationException？

在遍历一个集合的时候，我们可以使用并发集合类来避免ConcurrentModificationException，比如使用CopyOnWriteArrayList，而不是ArrayList。

48. 为何Iterator接口没有具体的实现？

Iterator接口定义了遍历集合的方法，但它的实现则是集合实现类的责任。每个能够返回用于遍历的Iterator的集合类都有它自己的Iterator实现内部类。

这就允许集合类去选择迭代器是fail-fast还是fail-safe的。比如，ArrayList迭代器是fail-fast的，而CopyOnWriteArrayList迭代器是fail-safe的。

49. UnsupportedOperationException是什么？

UnsupportedOperationException是用于表明操作不支持的异常。在JDK类中已被大量运用，在集合框架java.util.Collections.UnmodifiableCollection将会在所有add和remove操作中抛出这个异常。

50. 在Java中，HashMap是如何工作的？

HashMap在Map.Entry静态内部类实现中存储key-value对。HashMap使用哈希算法，在put和get方法中，它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候，HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。Entry存储在LinkedList中，所以如果存在entry，它使用equals()方法来检查传递的key是否已经存在，如果存在，它会覆盖value，如果不存在，它会创建一个新的entry然后保存。当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。下面的图片解释了详细内容。

其它关于HashMap比较重要的问题是容量、负荷系数和阀值调整。HashMap默认的初始容量是32，负荷系数是0.75。阀值是为负荷系数乘以容量，无论何时我们尝试添加一个entry，如果map的大小比阀值大的时候，HashMap会对map的内容进行重新哈希，且使用更大的容量。容量总是2的幂，所以如果你知道你需要存储大量的key-value对，比如缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

51. hashCode()和equals()方法有何重要性？

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：

- (1) 如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。
- (2) 如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

52. 我们能否使用任何类作为Map的key？

我们可以使用任何类作为Map的key，然而在使用它们之前，需要考虑以下几点：

- (1) 如果类重写了equals()方法，它也应该重写hashCode()方法。
- (2) 类的所有实例需要遵循与equals()和hashCode()相关的规则。请参考之前提到的这些规则。
- (3) 如果一个类没有使用equals()，你不应该在hashCode()中使用它。
- (4) 用户自定义key类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比如，我有一个类MyKey，在HashMap中使用它。

```
//传递给MyKey的name参数被用于equals()和hashCode()中 MyKey key = new MyKey('Pankaj');
//assume hashCode=1234 myHashMap.put(key, 'Value'); // 以下的代码会改变key的hashCode()和
equals()值 key.setName('Amit'); //assume new hashCode=7890 //下面会返回null，因为HashMap会
尝试查找存储同样索引的key，而key已被改变了，匹配失败，返回null myHashMap.get(new
MyKey('Pankaj')); 那就是为何String和Integer被作为HashMap的key大量使用。
```

53. Map接口提供了哪些不同的集合视图？

Map接口提供三个集合视图：

1、Set keyset()：返回map中包含的所有key的一个Set视图。集合是受map支持的，map的变化会在集合中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

2、Collection values(): 返回一个map中包含的所有value的一个Collection视图。这个collection受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个collection时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

3、Set<Map.Entry<K,V>> entrySet(): 返回一个map中包含的所有映射的一个集合视图。这个集合受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作，以及对迭代器返回的entry进行setValue外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

54. HashMap和HashTable有何不同？

- (1) HashMap允许key和value为null，而HashTable不允许。
- (2) HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。
- (3) 在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。
- (4) HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。
- (5) HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用ConcurrentHashMap。

55. 如何决定选用HashMap还是TreeMap？

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

56. ArrayList和Vector有何异同点？

ArrayList和Vector很多时候都很类似。

- 1、两者都是基于索引的，内部由一个数组支持。
- 2、两者维护插入的顺序，我们可以根据插入顺序来获取元素。
- 3、ArrayList和Vector的迭代器实现都是fail-fast的。
- 4、ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

- 1、Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。
- 2、ArrayList比Vector快，它因为有同步，不会过载。
- 3、ArrayList更加通用，因为我们可以通过Collections工具类轻易地获取同步列表和只读列表。

57. Array和ArrayList有何区别？什么时候更适合用Array？

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是固定的。

Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。

- 1、如果列表的大小已经指定，大部分情况下是存储和遍历它们。
- 2、对于遍历基本数据类型，尽管Collections使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢。
- 3、如果你要使用多维数组，使用`[][]`比`List<List<>>`更容易。

58. ArrayList和LinkedList有何区别？

ArrayList和LinkedList两者都实现了List接口，但是它们之间有些不同。

- 1、ArrayList是由Array所支持的基于一个索引的数据结构，所以它提供对元素的随机访问，复杂度为 $O(1)$ ，但LinkedList存储一系列的节点数据，每个节点都与前一个和下一个节点相连接。所以，尽管有使用索引获取元素的方法，内部实现是从起始点开始遍历，遍历到索引的节点然后返回元素，时间复杂度为 $O(n)$ ，比ArrayList要慢。
- 2、与ArrayList相比，在LinkedList中插入、添加和删除一个元素会更快，因为在一个元素被插入到中间的时候，不会涉及改变数组的大小，或更新索引。
- 3、LinkedList比ArrayList消耗更多的内存，因为LinkedList中的每个节点存储了前后节点的引用。

59. 哪些集合类提供对元素的随机访问？

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机访问。

60. EnumSet是什么？

java.util.EnumSet是使用枚举类型的集合实现。当集合创建时，枚举集合中的所有元素必须来自单个指定的枚举类型，可以是显示的或隐示的。EnumSet是不同步的，不允许值为null的元素。它也提供了一些有用的方法，比如copyOf(Collection c)、of(E first,E...rest)和complementOf(EnumSet s)。

61. 哪些集合类是线程安全的？

Vector、HashTable、Properties和Stack是同步类，所以它们是线程安全的，可以在多线程环境下使用。Java1.5并发API包括一些集合类，允许迭代时修改，因为它们都工作在集合的克隆上，所以它们在多线程环境中是安全的。

62. 并发集合类是什么？

Java1.5并发包（java.util.concurrent）包含线程安全集合类，允许在迭代时修改集合。迭代器被设计为fail-fast的，会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、ConcurrentHashMap、CopyOnWriteArraySet。

63. BlockingQueue是什么？

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。

BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。

Java提供了集中BlockingQueue的实现，比如ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue等。

64. 队列和栈是什么，列出它们的区别？

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque接口允许从两端检索元素。

栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口。

65. Collections类是什么？

Java.util.Collections是一个工具类仅包含静态方法，它们操作或返回集合。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如折半搜索、排序、混编和逆序等。

66. Comparable和Comparator接口是什么？

如果我们想使用Array或Collection的排序方法时，需要在自定义类里实现Java提供Comparable接口。

Comparable接口有compareTo(T obj)方法，它被排序方法所使用。我们应该重写这个方法，如果“this”对象比传递的对象参数更小、相等或更大时，它返回一个负整数、0或正整数。

但是，在大多数实际情况下，我们想根据不同参数进行排序。

比如，作为一个CEO，我想对雇员基于薪资进行排序，一个HR想基于年龄对他们进行排序。这就是我们需要使用Comparator接口的情景，因为Comparable.compareTo(Object o)方法实现只能基于一个字段进行排序，我们不能根据对象排序的需要选择字段。

Comparator接口的compare(Object o1, Object o2)方法的实现需要传递两个对象参数，若第一个参数比第二个小，返回负整数；若第一个等于第二个，返回0；若第一个比第二个大，返回正整数。

67. Comparable和Comparator接口有何区别？

Comparable和Comparator接口被用来对对象集合或者数组进行排序。

Comparable接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。

Comparator接口被用来提供不同的排序算法，我们可以选择需要使用的Comparator来对给定的对象集合进行排序。

68. 我们如何对一组对象进行排序？

如果我们需要对一个对象数组进行排序，我们可以使用Arrays.sort()方法。如果我们需要排序一个对象列表，我们可以使用Collection.sort()方法。两个类都有用于自然排序（使用Comparable）或基于标准的排序（使用Comparator）的重载方法sort()。Collections内部使用数组排序方法，所有它们两者都有相同的性能，只是Collections需要花时间将列表转换为数组。

69. 当一个集合被作为参数传递给一个函数时，如何才可以确保函数不能修改它？

在作为参数传递之前，我们可以使用Collections.unmodifiableCollection(Collection c)方法创建一个只读集合，这将确保改变集合的任何操作都会抛出UnsupportedOperationException。

70. 我们如何从给定集合那里创建一个synchronized的集合？

我们可以使用Collections.synchronizedCollection(Collection c)根据指定集合来获取一个synchronized（线程安全的）集合。

71. 集合框架里实现的通用算法有哪些？

Java集合框架提供常用的算法实现，比如排序和搜索。Collections类包含这些方法实现。大部分算法是操作List的，但一部分对所有类型的集合都是可用的。部分算法有排序、搜索、混编、最大最小值。

72. 大写的O是什么？举几个例子？

大写的O描述的是，就数据结构中的一系列元素而言，一个算法的性能。Collection类就是实际的数据结构，我们通常基于时间、内存和性能，使用大写的O来选择集合实现。

比如：例子1：ArrayList的get(index i)是一个常量时间操作，它不依赖list中元素的数量。所以它的性能是O(1)。

例子2：一个对于数组或列表的线性搜索的性能是O(n)，因为我们需要遍历所有的元素来查找需要的元素。

73. 与Java集合框架相关的有哪些最好的实践？

- 1、根据需要选择正确的集合类型。比如，如果指定了大小，我们会选用Array而非ArrayList。如果我们想根据插入顺序遍历一个Map，我们需要使用TreeMap。如果我们不想重复，我们应该使用Set。
- 2、一些集合类允许指定初始容量，所以如果我们能够估计到存储元素的数量，我们可以使用它，就避免了重新哈希或大小调整。
- 3、基于接口编程，而非基于实现编程，它允许我们后来轻易地改变实现。
- 4、总是使用类型安全的泛型，避免在运行时出现ClassCastException。
- 5、使用JDK提供的不可变类作为Map的key，可以避免自己实现hashCode()和equals()。
- 6、尽可能使用Collections工具类，或者获取只读、同步或空的集合，而非编写自己的实现。它将会提供代码重用性，它有着更好的稳定性和可维护性。

74. TreeMap和TreeSet在排序时如何比较元素？Collections工具类中的sort()方法如何比较元素？

TreeSet要求存放的对象所属的类必须实现Comparable接口，该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。Collections工具类的sort方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现Comparable接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java中对函数式编程的支持）。

Java并发编程

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

1. 多线程有什么用？

一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用？在我看来，这个回答更扯淡。所谓“知其然知其所以然”，“会用”只是“知其然”，“为什么用”才是“知其所以然”，只有达到“知其然知其所以然”的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：

1、发挥多核CPU的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。单核CPU上所谓的“多线程”那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程“同时”运行罢了。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

微信搜索公众账号：Java专栏，获取最新面试手册

2、防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

3、便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

2. 多线程和单线程的区别和联系？

- 1、在单核 CPU 中，将 CPU 分为很小的时间片，在每一时刻只能有一个线程在执行，是一种微观上轮流占用 CPU 的机制。
- 2、多线程会存在线程上下文切换，会导致程序执行速度变慢，即采用一个拥有两个线程的进程执行所需要的时间比一个线程的进程执行两次所需要的时间要多一些。

结论：即采用多线程不会提高程序的执行速度，反而会降低速度，但是对于用户来说，可以减少用户的响应时间。

3. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

线程

与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序

是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程

是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

4. 线程的创建方式

方法一：继承Thread类，作为线程对象存在（继承Thread对象）

```
public class CreatThreadDemo1 extends Thread{
    /**
     * 构造方法： 继承父类方法的Thread(String name); 方法
     * @param name
     */
    public CreatThreadDemo1(String name){
        super(name);
    }

    @Override
    public void run() {
        while (!interrupted()){
            System.out.println(getName()+"线程执行了...");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        CreatThreadDemo1 d1 = new CreatThreadDemo1("first");
        CreatThreadDemo1 d2 = new CreatThreadDemo1("second");

        d1.start();
        d2.start();

        d1.interrupt(); //中断第一个线程
    }
}
```

常规方法，不多做介绍了，interrupt方法，是来判断该线程是否被中断。（终止线程不允许用stop方法，该方法不会释放占用的资源。所以我们在设计程序的时候，要按照中断线程的思维去设计，就像上面的代码一样）。

让线程等待的方法

- Thread.sleep(200); //线程休息2ms
- Object.wait(); //让线程进入等待，直到调用Object的notify或者notifyAll时，线程停止休眠

方法二：实现Runnable接口，作为线程任务存在

```
public class CreatThreadDemo2 implements Runnable {
    @Override
    public void run() {
        while (true){
            System.out.println("线程执行了...");
        }
    }
}
```

```

public static void main(String[] args) {
    //将线程任务传给线程对象
    Thread thread = new Thread(new CreateThreadDemo2());
    //启动线程
    thread.start();
}
}

```

Runnable 只是用来修饰线程所执行的任务，它不是一个线程对象。想要启动Runnable对象，必须将它放到一个线程对象里。

方法三：匿名内部类创建线程对象

```

public class CreateThreadDemo3 extends Thread{
    public static void main(String[] args) {
        //创建无参线程对象
        new Thread(){
            @Override
            public void run() {
                System.out.println("线程执行了...");
            }
        }.start();
        //创建带线程任务的线程对象
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("线程执行了...");
            }
        }).start();
        //创建带线程任务并且重写run方法的线程对象
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable run 线程执行了...");
            }
        }) {
            @Override
            public void run() {
                System.out.println("Override run 线程执行了...");
            }
        }.start();
    }
}

```

创建带线程任务并且重写run方法的线程对象中，为什么只运行了Thread的run方法。我们看看Thread类的源码，

```

public
class Thread implements Runnable {
    ...
}

```

我们可以看到Thread实现了Runnable接口，而Runnable接口里有一个run方法。

所以，我们最终调用的重写的方法应该是Thread类的run方法。而不是Runnable接口的run方法。

方法四：创建带返回值的线程

```
public class CreatThreadDemo4 implements Callable {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        CreatThreadDemo4 demo4 = new CreatThreadDemo4();

        FutureTask<Integer> task = new FutureTask<Integer>(demo4); //FutureTask最终实现的是Runnable接口

        Thread thread = new Thread(task);

        thread.start();

        System.out.println("我可以在这里做点别的业务逻辑...因为FutureTask是提前完成任务");
        //拿出线程执行的返回值
        Integer result = task.get();
        System.out.println("线程中运算的结果为:"+result);
    }

    //重写Callable接口的call方法
    @Override
    public Object call() throws Exception {
        int result = 1;
        System.out.println("业务逻辑计算中...");
        Thread.sleep(3000);
        return result;
    }
}
```

Callable接口介绍：

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

返回指定泛型的call方法。然后调用FutureTask对象的get方法得到call方法的返回值。

方法五：定时器Timer

```
public class CreateThreadDemo5 {  
  
    public static void main(String[] args) {  
        Timer timer = new Timer();  
  
        timer.schedule(new TimerTask() {  
            @Override  
            public void run() {  
                System.out.println("定时器线程执行了...");  
            }  
        }, 0, 1000); //延迟0, 周期1s  
    }  
}
```

方法六：线程池创建线程

```
public class CreateThreadDemo6 {  
    public static void main(String[] args) {  
        //创建一个具有10个线程的线程池  
        ExecutorService threadPool = Executors.newFixedThreadPool(10);  
        long threadpoolUseTime = System.currentTimeMillis();  
        for (int i = 0; i < 10; i++) {  
            threadPool.execute(new Runnable() {  
                @Override  
                public void run() {  
                    System.out.println(Thread.currentThread().getName() + "线程执行  
了...");  
                }  
            });  
        }  
        long threadpoolUseTime1 = System.currentTimeMillis();  
        System.out.println("多线程用时" + (threadpoolUseTime1 - threadpoolUseTime));  
        //销毁线程池  
        threadPool.shutdown();  
        threadpoolUseTime = System.currentTimeMillis();  
    }  
}
```

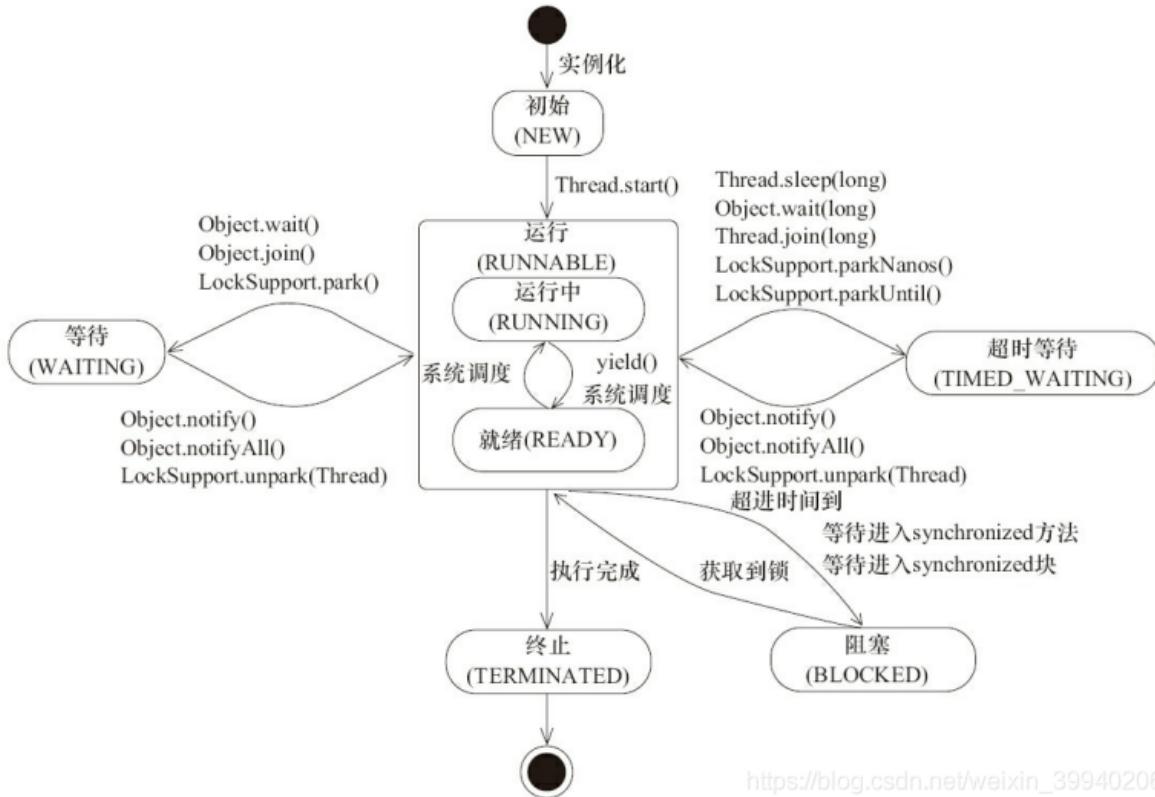
方法七：利用java8新特性 stream 实现并发

5. 线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面6种不同状态的其中一个状态（图源《Java 并发编程艺术》）

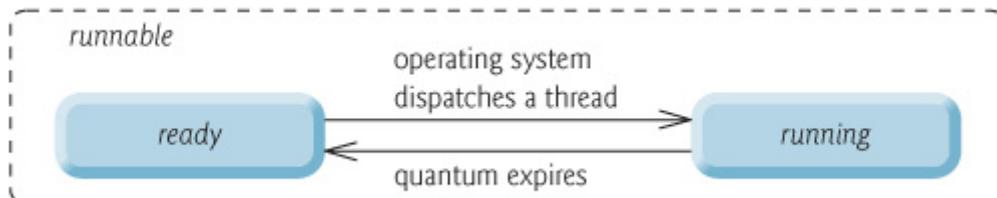
状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕 https://blog.csdn.net/weixin_39940206

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4节）：



操作系统隐藏 Java 虚拟机 (JVM) 中的 RUNNABLE 和 RUNNING 状态，它只能看到 RUNNABLE 状态
(图源：HowToDoInJava: Java Thread Life Cycle and Thread States)，所以 Java 系统一般将这两个状态统称为 RUNNABLE (运行中) 状态。

操作系统隐藏 Java 虚拟机 (JVM) 中的 RUNNABLE 和 RUNNING 状态，它只能看到 RUNNABLE 状态
(图源：HowToDoInJava:)，所以 Java 系统一般将这两个状态统称为 RUNNABLE (运行中) 状态。



当线程执行 `wait()` 方法之后，线程进入 WAITING (等待) 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 TIME_WAITING(超时等待) 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep(long millis)` 方法或 `wait(long millis)` 方法可以将 Java 线程置于 TIMED_WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 BLOCKED (阻塞) 状态。线程在执行 Runnable 的 `run()` 方法之后将会进入到 TERMINATED (终止) 状态。

6. 如何停止一个正在运行的线程

- 1、使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
- 2、使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
- 3、使用interrupt方法中断线程。

```
class MyThread extends Thread {  
    volatile boolean stop = false;  
  
    public void run() {  
        while (!stop) {  
            System.out.println(getName() + " is running");  
            try {  
                sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println("wake up from block...");  
                stop = true; // 在异常处理代码中修改共享变量的状态  
            }  
        }  
        System.out.println(getName() + " is exiting...");  
    }  
}  
  
class InterruptThreadDemo3 {  
    public static void main(String[] args) throws InterruptedException {  
        MyThread m1 = new MyThread();  
        System.out.println("Starting thread...");  
        m1.start();  
        Thread.sleep(3000);  
        System.out.println("Interrupt thread...: " + m1.getName());  
        m1.stop = true; // 设置共享变量为true  
        m1.interrupt(); // 阻塞时退出阻塞状态  
        Thread.sleep(3000); // 主线程休眠3秒以便观察线程m1的中断情况  
        System.out.println("Stopping application...");  
    }  
}
```

7. start()方法和run()方法的区别

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。

如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

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

看看Thread的start方法说明哈~

```
/**  
 * Causes this thread to begin execution; the Java virtual machine  
 * calls the <code>run</code> method of this thread.  
 * <p>  
 * The result is that two threads are running concurrently: the  
 * current thread (which returns from the call to the  
 * <code>start</code> method) and the other thread (which executes its  
 * <code>run</code> method).  
 * <p>  
 * It is never legal to start a thread more than once.  
 * In particular, a thread may not be restarted once it has completed  
 * execution.  
 *  
 * @exception IllegalThreadStateException if the thread was already  
 * started.  
 * @see #run()  
 * @see #stop()  
 */  
public synchronized void start() {  
    ....  
}
```

JVM执行start方法，会另起一条线程执行thread的run方法，这才起到多线程的效果~ 「为什么我们不能直接调用run()方法？」如果直接调用Thread的run()方法，其方法还是运行在主线程中，没有起到多线程效果。

9. Runnable接口和Callable接口的区别

有点深的问题了，也看出一个Java程序员学习知识的广度。

- 1、Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；
- 2、Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而 Callable+Future/FutureTask却可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，真的是非常有用。

10. 什么是线程安全？

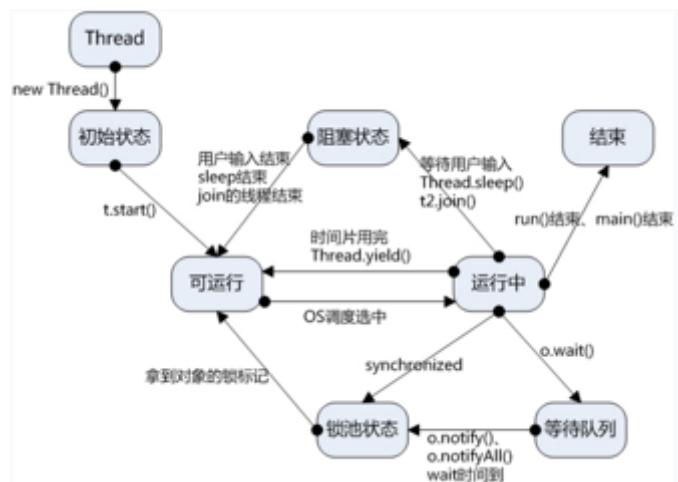
线程安全就是说多线程访问同代码，不会产生不确定的结果。

在多线程环境中，当各线程不共享数据的时候，即都是私有（private）成员，那么一定是线程安全的。但这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized，就是一段代码同时只能有一个线程来操作不然中间过程可能会产生不可预知的结果。

如果你的代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的一样的，就是线程安全的。

11. 线程的状态转换？



1、新建状态 (New)：新创建了一个线程对象。

2、就绪状态 (Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3、运行状态 (Running)：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态 (Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

(一)、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)

(二)、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

(三)、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。(注意，sleep是不会释放持有的锁)

5、死亡状态 (Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

12. 在多线程中，什么是上下文切换(context-switching)?

单核CPU也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程同时执行的，时间片一般是几十毫秒 (ms)。

操作系统中，CPU时间分片切换到另一个就绪的线程，则需要保存当前线程的运行的位置，同时需要加载需要恢复线程的环境信息。

13. Java中堆和栈有什么不同？

栈：在函数中定义的基本类型的变量和对象的引用变量都是在函数的栈内存中分配。

堆：堆内存用于存放由new创建的对象和数组。

从通俗化的角度来说，堆是用来存放对象的，栈是用来存放执行程序的

14. 如何确保线程安全？

- 对非安全的代码进行加锁控制
- 使用线程安全的类
- 多线程并发情况下，线程共享的变量改为方法级的局部变量

15. 什么是竞态条件？你怎样发现和解决竞争？

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

在临界区中使用适当的同步就可以避免竞态条件。

界区实现方法有两种，一种是用synchronized，一种是用Lock显式锁实现。

16. 用户线程和守护线程有什么区别？

守护线程都是为JVM中所有非守护线程的运行提供便利服务：只要当前JVM实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。

因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

17. 如何创建守护线程？以及在什么场合来使用它？

任何线程都可以设置为守护线程和用户线程，通过方法Thread.setDaemon(bool on)；true则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在Thread.start()之前调用，否则运行时会抛出异常。

守护线程相当于后台管理者 比如：进行内存回收,垃圾清理等工作

18. 线程安全的级别

不可变

不可变的对象一定是线程安全的，并且永远也不需要额外的同步。

Java类库中大多数基本数值类如Integer、String和BigInteger都是不可变的。

无条件的线程安全

由类的规格说明所规定的约束在对象被多个线程访问时仍然有效，不管运行时环境如何排列，线程都不需要任何额外的同步。

如 Random、ConcurrentHashMap、Concurrent集合、atomic

有条件的线程安全

有条件的线程安全类对于单独的操作可以是线程安全的，但是某些操作序列可能需要外部同步。

有条件的线程安全的最常见的例子是遍历由 Hashtable 或者 Vector 或者返回的迭代器

非线程安全(线程兼容)

线程兼容类不是线程安全的，但是可以通过正确使用同步而在并发环境中安全地使用。

如ArrayList HashMap

线程对立

线程对立是那些不管是否采用了同步措施，都不能在多线程环境中并发使用的代码。

如System.setOut()、System.runFinalizersOnExit()

19. 你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OSdependent)。

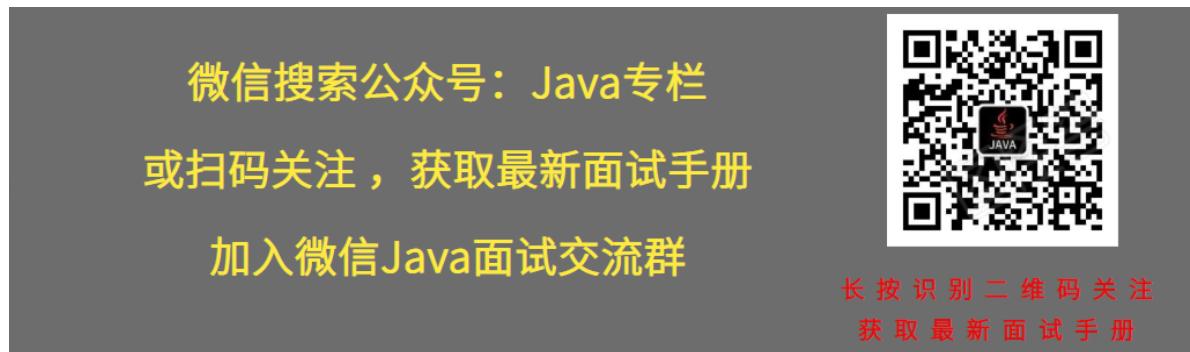
可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

20. 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦创建一个线程并启动它，它的执行便依赖于线程调度器的实现。

时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。

线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择。



21. volatile关键字的作用

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

- 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
- 禁止进行指令重排序。
- 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，并不能保证原子性； 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。

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见 `java.util.concurrent.atomic` 包下的类，比如 `AtomicInteger`。

22. volatile 变量和 atomic 变量有什么不同？

首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。

Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 volatile修饰count变量那么 count++ 操作就不是原子性的。

而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

23. volatile 是什么?可以保证有序性吗?

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

- 1) 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的，volatile关键字会强制将修改的值立即写入主存。
- 2) 禁止进行指令重排序。

volatile 不是原子性操作

什么叫保证部分有序性？

当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

```
x = 2;          //语句1
y = 0;          //语句2
flag = true;    //语句3
x = 4;          //语句4
y = -1;         //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

使用 Volatile 一般用于 状态标记量 和 单例模式的双检锁

24. 什么是Java内存模型

Java内存模型定义了一种多线程访问Java内存的规范。Java内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下Java内存模型的几部分内容：

- 1、Java内存模型将内存分为了主内存和工作内存。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去
- 2、定义了几个原子操作，用于操作主内存和工作内存中的变量
- 3、定义了volatile变量的使用规则
- 4、happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的

25. sleep方法和wait方法有什么区别

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu给其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。在调用sleep()方法的过程中，线程不会释放对象锁。

当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备，获取对象锁进入运行状态。

26. 线程的sleep()方法和yield()方法有什么区别？

- ① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
- ② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
- ③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
- ④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

27. Thread.sleep(0)的作用是什么

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

28. 线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：

- 1、Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的
- 2、Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的

29. 在线程中你怎么处理不可控制异常？

在Java中有两种异常。

非运行时异常（Checked Exception）：这种异常必须在方法声明的throws语句指定，或者在方法体内捕获。例如：IOException和ClassNotFoundException。

运行时异常（Unchecked Exception）：这种异常不必在方法声明中指定，也不需要在方法体中捕获。例如，NumberFormatException。

因为run()方法不支持throws语句，所以当线程对象的run()方法抛出非运行异常时，我们必须捕获并且处理它们。当运行时异常从run()方法中抛出时，默认行为是在控制台输出堆栈记录并且退出程序。

好在，java提供给我们一种在线程对象里捕获和处理运行时异常的一种机制。实现用来处理运行时异常的类，这个类实现UncaughtExceptionHandler接口并且实现这个接口的uncaughtException()方法。示例：

```
package concurrency;

import java.lang.Thread.UncaughtExceptionHandler;

public class Main2 {
    public static void main(String[] args) {
```

```

        Task task = new Task();
        Thread thread = new Thread(task);
        thread.setUncaughtExceptionHandler(new ExceptionHandler());
        thread.start();
    }
}

class Task implements Runnable{
    @Override
    public void run() {
        int numero = Integer.parseInt("TTT");
    }
}

class ExceptionHandler implements UncaughtExceptionHandler{
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.printf("An exception has been captured\n");
        System.out.printf("Thread: %s\n", t.getId());
        System.out.printf("Exception: %s: %s\n",
e.getClass().getName(),e.getMessage());
        System.out.printf("Stack Trace: \n");
        e.printStackTrace(System.out);
        System.out.printf("Thread status: %s\n",t.getState());
    }
}

```

当一个线程抛出了异常并且没有被捕获时（这种情况只可能是运行时异常），JVM检查这个线程是否被预置了未捕获异常处理器。如果找到，JVM将调用线程对象的这个方法，并将线程对象和异常作为传入参数。

Thread类还有另一个方法可以处理未捕获到的异常，即静态方法setDefaultUncaughtExceptionHandler()。这个方法在应用程序中为所有的线程对象创建了一个异常处理器。

当线程抛出一个未捕获到的异常时，JVM将为异常寻找以下三种可能的处理器。

- 首先，它查找线程对象的未捕获异常处理器。
- 如果找不到，JVM继续查找线程对象所在的线程组（ThreadGroup）的未捕获异常处理器。
- 如果还是找不到，如同本节所讲的，JVM将继续查找默认的未捕获异常处理器。
- 如果没有一个处理器存在，JVM则将堆栈异常记录打印到控制台，并退出程序。

30. 同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越小越好。

借着这一条，我额外提一点，虽说同步的范围越小越好，但是在Java虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

31. 有三个线程T1,T2,T3,如何保证顺序执行?

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

实际上先启动三个线程中哪一个都行，因为在每个线程的run方法中用join方法限定了三个线程的执行顺序。

```
public class JoinTest2 {  
  
    // 1.现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行  
  
    public static void main(String[] args) {  
  
        final Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                System.out.println("t1");  
            }  
        });  
        final Thread t2 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    // 引用t1线程，等待t1线程执行完  
                    t1.join();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("t2");  
            }  
        });  
        Thread t3 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    // 引用t2线程，等待t2线程执行完  
                    t2.join();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("t3");  
            }  
        });  
        t3.start(); //这里三个线程的启动顺序可以任意，大家可以试下！  
        t2.start();  
        t1.start();  
    }  
}
```

32. 什么是CAS

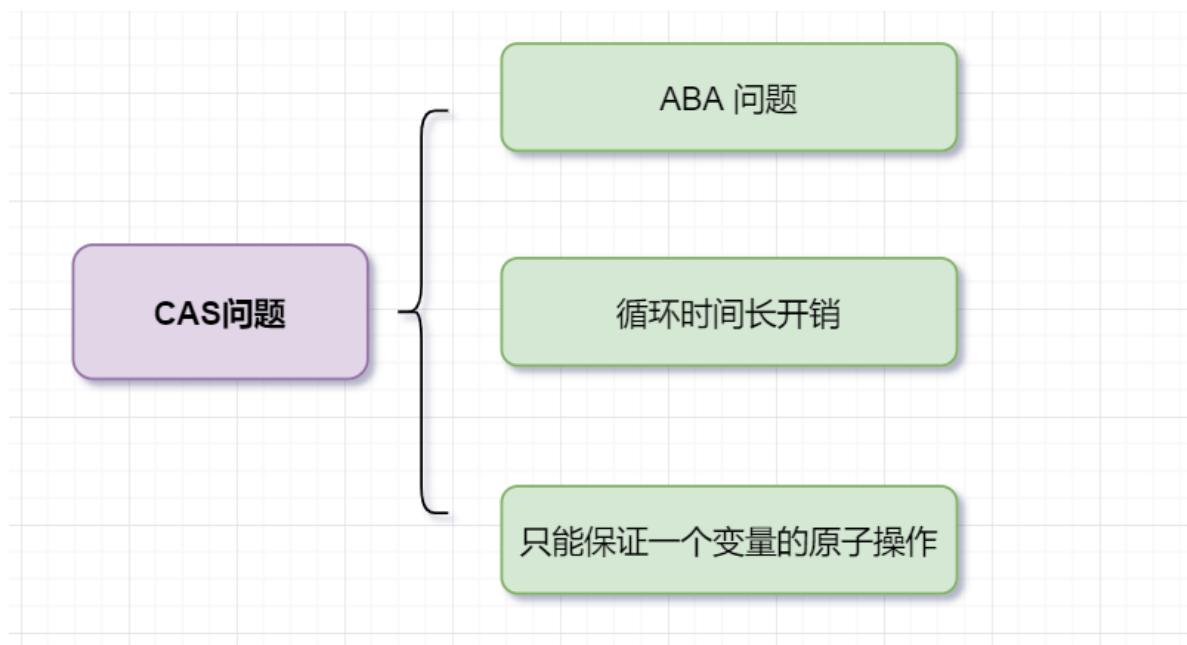
CAS，全称为Compare and Swap，即比较-替换。

假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

33. CAS? CAS有什么缺陷，如何解决？

CAS涉及3个操作数，内存地址值V，预期原值A，新值B；如果内存位置的值V与预期原A值相匹配，就更新为新值B，否则不更新

CAS有什么缺陷？



ABA 问题

并发环境下，假设初始条件是A，去修改数据时，发现是A就会执行修改。但是看到的虽然是A，中间可能发生了A变B，B又变回A的情况。此时A已经非彼A，数据即使成功修改，也可能有问题。

可以通过AtomicStampedReference「解决ABA问题」，它，一个带有标记的原子引用类，通过控制变量值的版本来保证CAS的正确性。

循环时间长开销

自旋CAS，如果一直循环执行，一直不成功，会给CPU带来非常大的执行开销。

很多时候，CAS思想体现，是有个自旋次数的，就是为了避开这个耗时问题~

只能保证一个变量的原子操作。

CAS保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS目前无法直接保证操作的原子性的。

可以通过这两个方式解决这个问题：

- 使用互斥锁来保证原子性；

- 将多个变量封装成对象，通过AtomicReference来保证原子性。

34. 什么是AQS

简单说一下AQS，AQS全称为AbstractQueuedSynchronizer，翻译过来应该是抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。

AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

35. 线程池作用

(如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略) 合理利用线程池能够带来三个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

36. ThreadLocal是什么

从名字我们就可以看到ThreadLocal叫做线程变量，意思是ThreadLocal中填充的变量属于**当前**线程，该变量对其他线程而言是隔离的。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

从字面意思来看非常容易理解，但是从实际使用的角度来看，就没那么容易了，作为一个面试常问的点，使用场景那也是相当的丰富：

- 1、在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。
- 2、线程间数据隔离
- 3、进行事务操作，用于存储线程事务信息。
- 4、数据库连接，Session会话管理。

37. ThreadLocal有什么用

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

38. ThreadLocal原理，使用注意点，应用场景有哪些？

回答四个主要点：

- ThreadLocal是什么？
- ThreadLocal原理
- ThreadLocal使用注意点
- ThreadLocal的应用场景

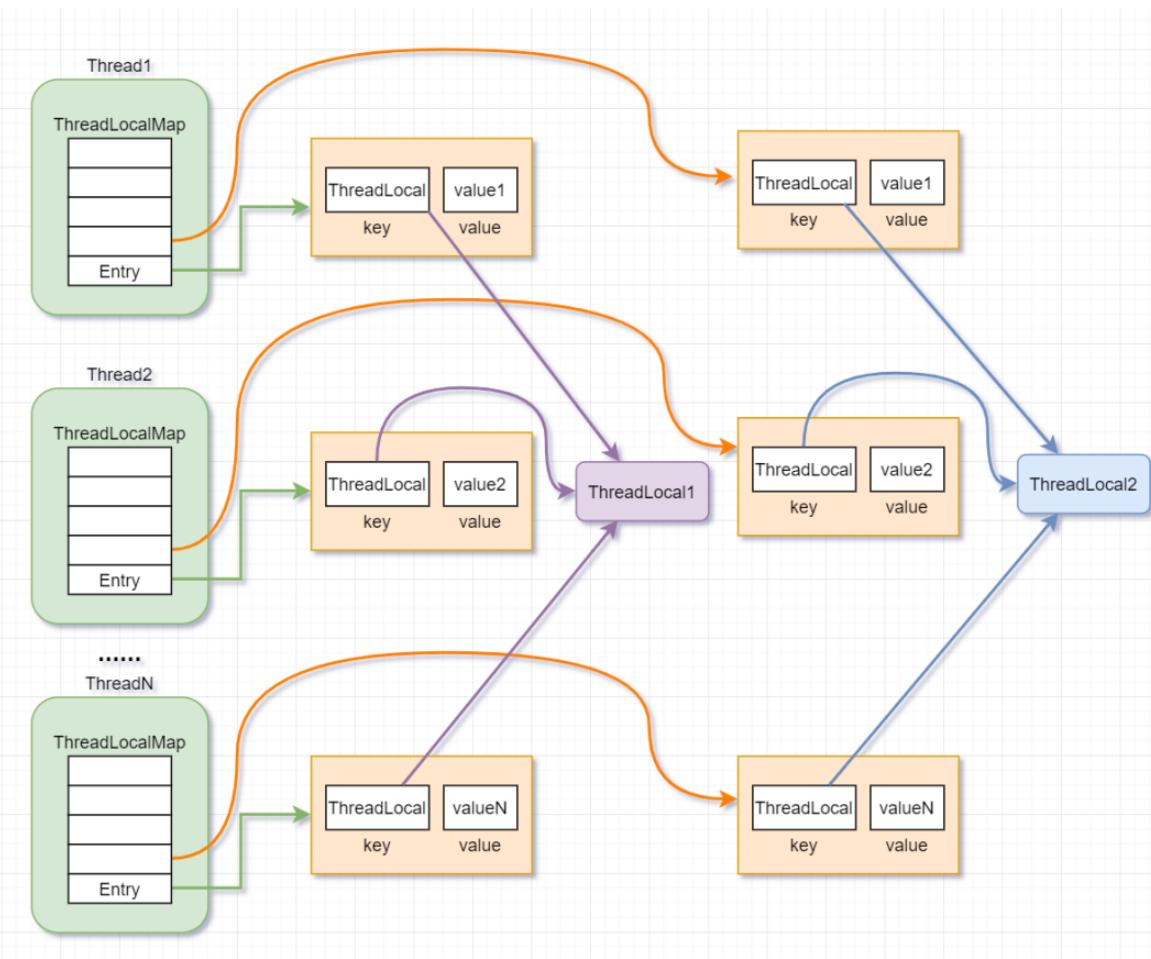
ThreadLocal是什么？

ThreadLocal，即线程本地变量。如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作自己本地内存里面的变量，从而起到线程隔离的作用，避免了线程安全问题。

```
// 创建一个ThreadLocal变量  
static ThreadLocal<String> localVariable = new ThreadLocal<>();
```

ThreadLocal原理

ThreadLocal内存结构图：



由结构图是可以看出：

- Thread对象中持有一个ThreadLocal.ThreadLocalMap的成员变量。
- ThreadLocalMap内部维护了Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型值。

对照着几段关键源码来看，更容易理解一点哈~

```

public class Thread implements Runnable {
    //ThreadLocal.ThreadLocalMap是Thread的属性
    ThreadLocal.ThreadLocalMap threadLocals = null;
}

```

ThreadLocal中的关键方法set()和get()

```

public void set(T value) {
    Thread t = Thread.currentThread(); //获取当前线程t
    ThreadLocalMap map = getMap(t); //根据当前线程获取到ThreadLocalMap
    if (map != null)
        map.set(this, value); //K, V设置到ThreadLocalMap中
    else
        createMap(t, value); //创建一个新的ThreadLocalMap
}

public T get() {
    Thread t = Thread.currentThread(); //获取当前线程t
    ThreadLocalMap map = getMap(t); //根据当前线程获取到ThreadLocalMap
    if (map != null) {
        //由this (即ThreadLoca对象) 得到对应的value, 即ThreadLocal的泛型值
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

```

ThreadLocalMap的Entry数组

```

static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
}

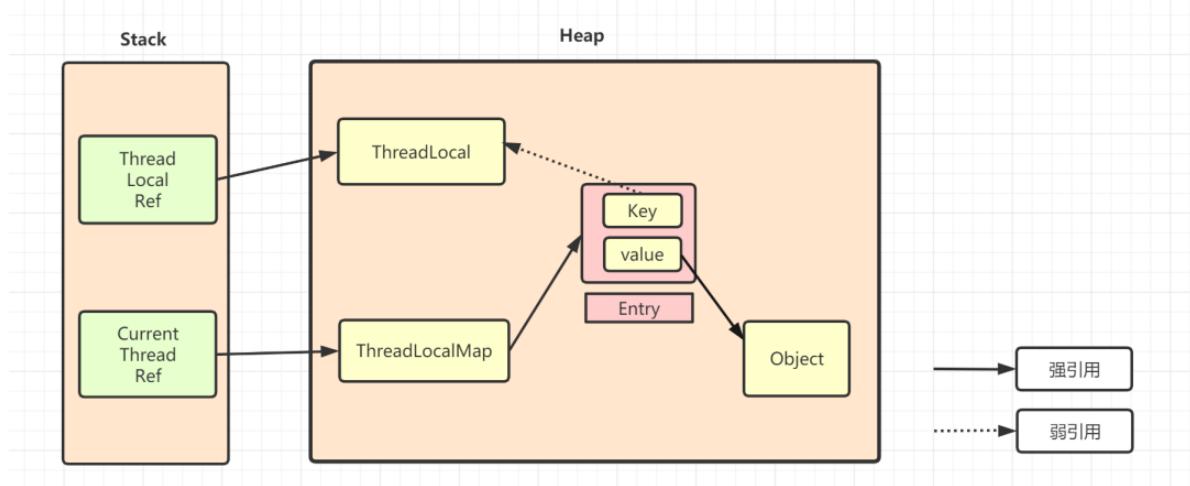
```

所以怎么回答「**ThreadLocal的实现原理**」？如下，最好是能结合以上结构图一起说明哈~

- Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，即每个线程都有一个属于自己的ThreadLocalMap。
- ThreadLocalMap内部维护着Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型值。
- 每个线程在往ThreadLocal里设置值的时候，都是往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离。

ThreadLocal 内存泄露问题

先看看一下的TreadLocal的引用示意图哈,



ThreadLocalMap中使用的 key 为 ThreadLocal 的弱引用，如下

```
static class ThreadLocalMap {  
    static class Entry extends WeakReference<ThreadLocal<?>> {  
        /** The value associated with this ThreadLocal. */  
        Object value;  
  
        Entry(ThreadLocal<?> k, Object v) {  
            super(k);  
            value = v;  
        }  
    }  
}
```

弱引用：只要垃圾回收机制一运行，不管JVM的内存空间是否充足，都会回收该对象占用的内存。

弱引用比较容易被回收。因此，如果ThreadLocal (ThreadLocalMap的Key) 被垃圾回收器回收了，但是因为ThreadLocalMap生命周期和Thread是一样的，它这时候如果不被回收，就会出现这种情况：ThreadLocalMap的key没了，value还在，这就会「造成了内存泄漏问题」。

如何「解决内存泄漏问题」？使用完ThreadLocal后，及时调用remove()方法释放内存空间。

ThreadLocal的应用场景

- 数据库连接池
- 会话管理中使用

39. notify()和notifyAll()有什么区别？

notify()和notifyAll()都是Object对象用于通知处在等待该对象的线程的方法。

- void notify(): 唤醒一个正在等待该对象的线程。
- void notifyAll(): 唤醒所有正在等待该对象的线程。

notify可能会导致死锁，而notifyAll则不会

任何时候只有一个线程可以获得锁，也就是说只有一个线程可以运行synchronized 中的代码

使用notifyall,可以唤醒 所有处于wait状态的线程，使其重新进入锁的竞争队列中，而notify只能唤醒一个。

wait() 应配合while循环使用，不应使用if，务必在wait()调用前后都检查条件，如果不满足，必须调用notify()唤醒另外的线程来处理，自己继续wait()直至条件满足再往下执行。

notify() 是对notifyAll()的一个优化，但它有很精确的应用场景，并且要求正确使用。不然可能导致死锁。正确的场景应该是 WaitSet中等待的是相同的条件，唤醒任一个都能正确处理接下来的事项，如果唤醒的线程无法正确处理，务必确保继续notify()下一个线程，并且自身需要重新回到WaitSet中.

40. 为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的， wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

41. wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于： wait()方法立即释放对象监视器， notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

42. wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于： wait()方法立即释放对象监视器， notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

43. 线程中断是否能直接调用stop,为什么？

Java提供的终止方法只有一个stop,但是不建议使用此方法,因为它有以下三个问题:

1. stop方法是过时的 从Java编码规则来说,已经过时的方式不建议采用.
2. stop方法会导致代码逻辑不完整 stop方法是一种"恶意"的中断,一旦执行stop方法,即终止当前正在运行的线程,不管线程逻辑是否完整,这是非常危险的.

44. 什么是阻塞（Blocking）和非阻塞（Non-Blocking）？

阻塞和非阻塞通常用来形容多线程间的相互影响。比如一个线程占用了临界区资源，那么其他所有需要这个资源的线程就必须在这个临界区中进行等待。等待会导致线程挂起，这种情况就是阻塞。此时，如果占用资源的线程一直不愿意释放资源，那么其他所有阻塞在这个临界区上的线程都不能工作。

非阻塞的意思与之相反，它强调没有一个线程可以妨碍其他线程执行。所有的线程都会尝试不断前向执行。

45. 什么是自旋

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。

既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

46. 自旋锁的优缺点？

自旋锁不会引起调用者休眠，如果自旋锁已经被别的线程保持，调用者就一直循环在那里看是否该自旋锁的保持者释放了锁。由于自旋锁不会引起调用者休眠，所以自旋锁的效率远高于互斥锁。

虽然自旋锁效率比互斥锁高，但它会存在下面两个问题：1、自旋锁一直占用CPU，在未获得锁的情况下，一直运行，如果不能在很短的时间内获得锁，会导致CPU效率降低。2、试图递归地获得自旋锁会引起死锁。递归程序决不能在持有自旋锁时调用它自己，也决不能在递归调用时试图获得相同的自旋锁。

由此可见，我们要慎重的使用自旋锁，自旋锁适合于锁使用者保持锁时间比较短并且锁竞争不激烈的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

47. 什么是线程池？为什么要使用它？

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。

从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）

线程池提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java并发编程的艺术》提到的来说一下使用线程池的好处：

- **降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。** 当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

48. 常用的线程池模式以及不同线程池的使用场景？

以下是Java自带的几种线程池：

1、newFixedThreadPool 创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

2、newCachedThreadPool 创建一个可缓存的线程池。这种类型的线程池特点是：

- 1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Intgerger. MAX_VALUE),这样可灵活的往线程池中添加线程。
- 2).如果长时间没有往线程池中提交任务, 即如果工作线程空闲了指定的时间(默认为1分钟), 则该工作线程将自动终止。终止后, 如果你又提交了新的任务, 则线程池重新创建一个工作线程。

3、newSingleThreadExecutor创建一个单线程化的Executor, 即只创建唯一的工作者线程来执行任务, 如果这个线程异常结束, 会有另一个取代它, 保证顺序执行(我觉得这点是它的特色)。

单工作线程最大的特点是可保证顺序地执行各个任务, 并且在任意给定的时间不会有多个线程是活动的。

4、newScheduleThreadPool 创建一个定长的线程池, 而且支持定时的以及周期性的任务执行, 类似于Timer。

49. 在Java中Executor、ExecutorService、Executors的区别?

Executor 和 ExecutorService 这两个接口主要的区别是:

- ExecutorService 接口继承了 Executor 接口, 是 Executor 的子接口
- Executor 和 ExecutorService 第二个区别是: Executor 接口定义了 execute()方法用来接收一个 Runnable接口的对象, 而 ExecutorService 接口中的 submit()方法可以接受Runnable和Callable接口的对象。
- Executor 和 ExecutorService 接口第三个区别是 Executor 中的 execute() 方法不返回任何结果, 而 ExecutorService 中的 submit()方法可以通过一个 Future 对象返回运算结果。
- Executor 和 ExecutorService 接口第四个区别是除了允许客户端提交一个任务, ExecutorService 还提供用来控制线程池的方法。比如: 调用 shutDown() 方法终止线程池。

Executors 类提供工厂方法用来创建不同类型的线程池。

比如: newSingleThreadExecutor() 创建一个只有一个线程的线程池, newFixedThreadPool(int numThreads)来创建固定线程数的线程池, newCachedThreadPool()可以根据需要创建新的线程, 但如果已有线程是空闲的会重用已有线程。

50. 请说出与线程同步以及线程调度相关的方法。

- wait(): 使一个线程处于等待 (阻塞) 状态, 并且释放所持有的对象的锁;
- sleep(): 使一个正在运行的线程处于睡眠状态, 是一个静态方法, 调用此方法要处理 InterruptedException异常;
- notify(): 唤醒一个处于等待状态的线程, 当然在调用此方法的时候, 并不能确切的唤醒某一个等待状态的线程, 而是由JVM确定唤醒哪个线程, 而且与优先级无关;
- notifyAll(): 唤醒所有处于等待状态的线程, 该方法并不是将对象的锁给所有线程, 而是让它们竞争, 只有获得锁的线程才能进入就绪状态;

微信搜索公众号: Java专栏

或扫码关注, 获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

微信搜索公众号: Java专栏, 获取最新面试手册

51. 举例说明同步和异步。

如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

52. 不使用stop停止线程？

当run() 或者 call() 方法执行完的时候线程会自动结束，如果要手动结束一个线程，你可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程。

使用自定义的标志位决定线程的执行情况

```
public class SafeStopThread implements Runnable{
    private volatile boolean stop=false;//此变量必须加上volatile
    int a=0;
    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(!stop){
            synchronized ("") {
                a++;
                try {
                    Thread.sleep(100);
                } catch (Exception e) {
                    // TODO: handle exception
                }
                a--;
                String tn=Thread.currentThread().getName();
                System.out.println(tn+":a="+a);
            }
        }
        //线程终止
    }
    public void terminate(){
        stop=true;
    }
    public static void main(String[] args) {
        SafeStopThread t=new SafeStopThread();
        Thread t1=new Thread(t);
        t1.start();
        for(int i=0;i<5;i++){
            new Thread(t).start();
        }
        t.terminate();
    }
}
```

53. 如何控制某个方法允许并发访问线程的大小?

Semaphore两个重要的方法就是semaphore.acquire() 请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）semaphore.release()释放一个信号量，此时信号量个数+1

```
public class SemaphoreTest {  
    private Semaphore mSemaphore = new Semaphore(5);  
    public void run(){  
        for(int i=0; i< 100; i++){  
            new Thread(new Runnable() {  
                @Override  
                public void run() {  
                    test();  
                }  
            }).start();  
        }  
  
        private void test(){  
            try {  
                mSemaphore.acquire();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(Thread.currentThread().getName() + " 进来了");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(Thread.currentThread().getName() + " 出去了");  
            mSemaphore.release();  
        }  
    }  
}
```

54. 如何创建线程池

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险**

Executors 返回线程池对象的弊端如下：

- **FixedThreadPool 和 SingleThreadExecutor**： 允许请求的队列长度为 Integer.MAX_VALUE,可能堆积大量的请求，从而导致OOM。
- **CachedThreadPool 和 ScheduledThreadPool**： 允许创建的线程数量为 Integer.MAX_VALUE，可能会创建大量线程，从而导致OOM。

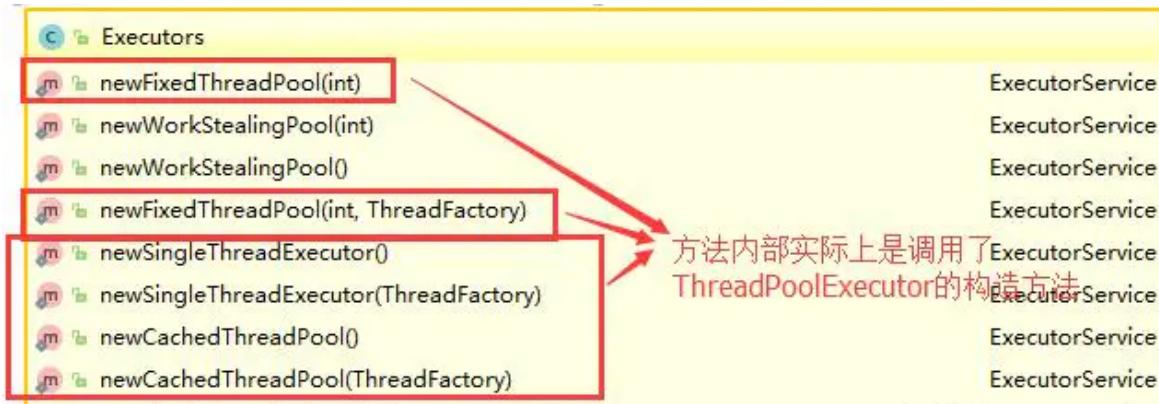
方式一：通过构造方法实现

```
ThreadPoolExecutor  
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>)  
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, ThreadFactory)  
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, RejectedExecutionHandler)  
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, ThreadFactory, RejectedExecutionHandler)
```

方式二：通过Executor框架的工具类Executors来实现 我们可以创建三种类型的ThreadPoolExecutor：

- **FixedThreadPool**：该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
- **SingleThreadExecutor**：方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
- **CachedThreadPool**：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

对应Executors工具类中的方法如图所示：



55. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

- 1、高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
- 2、并发不高、任务执行时间长的业务要区分开看：
 - 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
 - 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换
- 3、并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

56. 什么是线程安全

又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释地最好的：如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

1、不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

2、绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

3、相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有個线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。

4、线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

57. Java中interrupted 和isInterrupted方法的区别？

interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。

Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。

非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。

无论如何，一个线程的中断状态都有可能被其它线程调用中断来改变。

58. Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中，而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

59. 说一说自己对于 synchronized 关键字的理解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。另外，在Java早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，Java的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。庆幸的是在Java 6之后Java官方对从JVM层面对synchronized较大优化，所以现在的synchronized锁效率也优化得很

微信搜索公众号：Java专栏，获取最新面试手册

不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

60. 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗 synchronized关键字最主要的三种使用方式：

修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法: 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。**

修饰代码块: 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。**总结：**synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓存功能！

61. Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

- 1、获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java
- 2、打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取到的是具体某个线程当前运行的堆栈，

62. 一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。

另外重要的一点是：如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放

63. 如何在两个线程之间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

64. 如何在两个线程间共享数据？

同一个Runnable，使用全局变量。

第一种：将共享数据封装到一个对象中，把这个共享数据所在的对象传递给不同的Runnable

第二种：将这些Runnable对象作为某一个类的内部类，共享的数据作为外部类的成员变量，对共享数据的操作分配给外部类的方法来完成，以此实现对操作共享数据的互斥和通信，作为内部类的Runnable来操作外部类的方法，实现对数据的操作

```
class ShareData {  
    private int x = 0;  
  
    public synchronized void addx(){  
        x++;  
        System.out.println("x++ : "+x);  
    }  
    public synchronized void subx(){  
        x--;  
        System.out.println("x-- : "+x);  
    }  
}  
  
public class ThreadsVisitData {  
  
    public static ShareData share = new ShareData();  
  
    public static void main(String[] args) {  
        //final ShareData share = new ShareData();  
        new Thread(new Runnable() {  
            public void run() {  
                for(int i = 0;i<100;i++){  
                    share.addx();  
                }  
            }  
        }).start();  
        new Thread(new Runnable() {  
            public void run() {  
                for(int i = 0;i<100;i++){  
                    share.subx();  
                }  
            }  
        }).start();  
    }  
}
```

65. Java中活锁和死锁有什么区别？

活锁：一个线程通常会有会响应其他线程的活动。如果其他线程也会响应另一个线程的活动，那么就有可能发生活锁。同死锁一样，发生活锁的线程无法继续执行。然而线程并没有阻塞——他们在忙于响应对方无法恢复工作。这就相当于两个在走廊相遇的人：甲向他自己的左边靠想让乙过去，而乙向他的右边靠想让甲过去。可见他们阻塞了对方。甲向他的右边靠，而乙向他的左边靠，他们还是阻塞了对方。

死锁：两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候，死锁会让你的程序挂起无法完成任务。

66. Java中的死锁

在Java中使用多线程，就会**有可能导致死锁**问题。死锁会让程序一直**卡住**，不再程序往下执行。我们只能通过**中止并重启**的方式来让程序重新执行。这是我们非常不愿意看到的一种现象，我们要**尽可能避免**死锁的情况发生！

死锁的四个必要条件

- 1、**互斥条件**：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用完释放。
- 2、**请求和保持条件**：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
- 3、**不剥夺条件**：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
- 4、**环路等待条件**：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{A, B, C, …, Z} 中的A正在等待一个B占用的资源；B正在等待C占用的资源，……，Z正在等待已被A占用的资源。

死锁实例

```
/**  
 * 死锁类示例  
 */  
public class DeadLock implements Runnable {  
    public int flag = 1;  
    //静态对象是类的所有对象共享的  
    private static Object o1 = new Object(), o2 = new Object();  
  
    @Override  
    public void run() {  
        System.out.println("flag:{}"+flag);  
        if (flag == 1) { //先锁o1，再对o2加锁，环路等待条件  
            synchronized (o1) {  
                try {  
                    Thread.sleep(500);  
                } catch (Exception e) {  
                    e.printStackTrace();  
                }  
  
                synchronized (o2) {  
                    System.out.println("1");  
                }  
            }  
        }  
        if (flag == 0) { //先锁o2，在锁o1  
            synchronized (o2) {  
                try {  
                    Thread.sleep(500);  
                } catch (Exception e) {  
                    e.printStackTrace();  
                }  
                synchronized (o1) {  
                    System.out.println("0");  
                }  
            }  
        }  
    }  
}
```

```

public static void main(String[] args) {
    DeadLock td1 = new DeadLock();
    DeadLock td2 = new DeadLock();
    td1.flag = 1;
    td2.flag = 0;
    //td1,td2都处于可执行状态，但JVM线程调度先执行哪个线程是不确定的。
    //td2的run()可能在td1的run()之前运行
    new Thread(td1).start();
    new Thread(td2).start();
}
}

```

- 1、当DeadLock 类的对象flag=1时 (td1)，先锁定o1,睡眠500毫秒
- 2、而td1在睡眠的时候另一个flag==0的对象 (td2) 线程启动，先锁定o2,睡眠500毫秒
- 3、td1睡眠结束后需要锁定o2才能继续执行，而此时o2已被td2锁定；
- 4、td2睡眠结束后需要锁定o1才能继续执行，而此时o1已被td1锁定；
- 5、td1、td2相互等待，都需要得到对方锁定的资源才能继续执行，从而死锁。

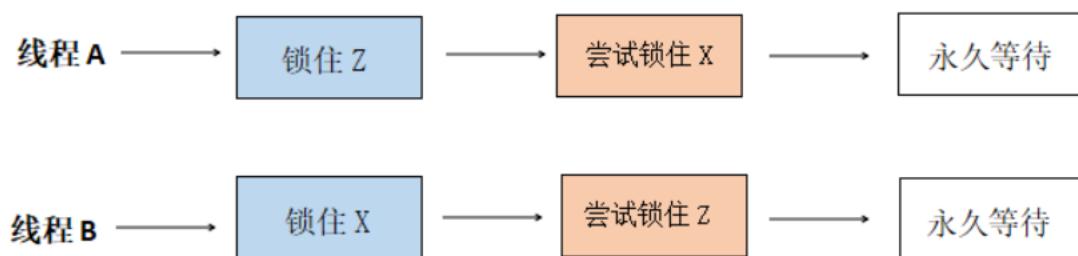
67. 如何避免死锁和检测

预防死锁

- 破坏互斥条件：使资源同时访问而非互斥使用，就没有进程会阻塞在资源上，从而不发生死锁
- 破坏请求和保持条件：采用静态分配的方式，静态分配的方式是指进程必须在执行之前就申请需要的全部资源，且直至所要的资源全部得到满足后才开始执行，只要有一个资源得不到分配，也不给这个进程分配其他的资源。
- 破坏不剥夺条件：即当某进程获得了部分资源，但得不到其它资源，则释放已占有的资源，但是只适用于内存和处理器资源。
- 破坏循环等待条件：给系统的所有资源编号，规定进程请求所需资源的顺序必须按照资源的编号依次进行。

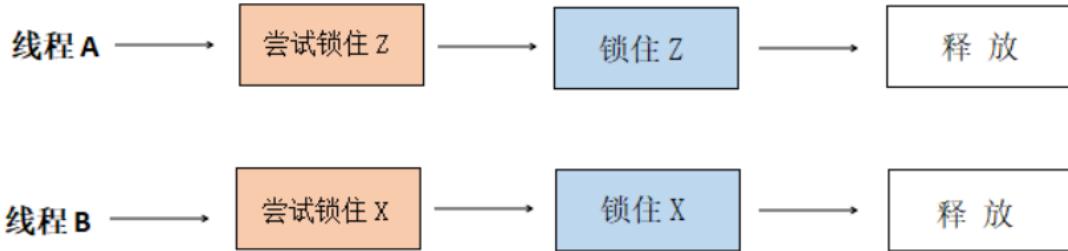
设置加锁顺序

如果两个线程 (A和B),当A线程已经锁住了Z,而又去尝试锁住X,而X已经被线程B锁住,线程A和线程B分别持有对应的锁,而又去争夺其他一个锁(尝试锁住另一个线程已经锁住的锁)的时候,就会发生死锁,如下图:



https://blog.csdn.net/qq_14996421

两个线程试图以不同的顺序来获得相同的锁，如果按照相同的顺序来请求锁，那么就不会出现循环的加锁依赖性，因此也就不会产生死锁，每个需要锁Z和锁X的线程都以相同的顺序来获取Z和X，那么就不会发生死锁了，如下图所示：



https://blog.csdn.net/qq_14996421

- 这样死锁就永远不会发生。针对两个特定的锁，可以尝试按照锁对象的hashCode值大小的顺序，分别获得两个锁，这样锁总是会以特定的顺序获得锁，我们通过设置锁的顺序，来防止死锁的发生，在这里我们使用System.identityHashCode方法来定义锁的顺序，这个方法将返回由Object.hashCode返回的值，这样就可以消除死锁发生的可能性。

```

public class DeadLockExample3 {

    // 加时赛锁，在极少数情况下，如果两个hash值相等，使用这个锁进行加锁
    private static final Object tieLock = new Object();

    public void transferMoney(final Account fromAcct,
                             final Account toAcct,
                             final DollarAmount amount)
        throws InsufficientFundsException {
        class Helper {
            public void transfer() throws InsufficientFundsException {
                if (fromAcct.getBalance().compareTo(amount) < 0)
                    throw new InsufficientFundsException();
                else {
                    fromAcct.debit(amount);
                    toAcct.credit(amount);
                }
            }
        }
    }

    // 得到两个锁的hash值
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    // 根据hash值判断锁顺序，决定锁的顺序
    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    }

    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    }
}

} else {// 如果两个对象的hash相等，通过tieLock来决定加锁的顺序，否则又会重新引入死锁--加时赛锁
    synchronized (tieLock) {
        synchronized (fromAcct) {
}

```

```
        synchronized (toAcct) {
            new Helper().transfer();
        }
    }
}
```

- 在极少数情况下，两个对象可能拥有两个相同的散列值，此时必须通过某种任意的方法来决定锁的顺序，否则可能又会重新引入死锁。
 - 为了避免这种情况，可以使用“加时(Tie-Breaking)”锁，这获得这两个Account锁之前，从而消除了死锁发生的可能性

68. 什么是可重入锁（ReentrantLock）？

Java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。

ReentrantLock 类实现了 Lock，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

Reentrant 锁意味着什么呢？

简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了synchronized的语义；如果线程进入由线程已经拥有的监控器保护的synchronized块，就允许线程继续进行，当线程退出第二个（或者后续）synchronized块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个synchronized块时，才释放锁。

69. 讲一下 synchronized 关键字的底层原理

`synchronized` 关键字底层原理属于 JVM 层面。

① synchronized 同步语句块的情况

```
public class SynchronizedDemo {  
    public void method() {  
        synchronized (this) {  
            System.out.println("synchronized 代码块");  
        }  
    }  
}
```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 `javac SynchronizedDemo.java` 命令生成编译后的 .class 文件，然后执行 `javap -c -s -v -l SynchronizedDemo.class`。

```

public void method();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=3, args_size=1
  0: aload_0
  1: dup
  2: astore_1
  3: monitorenter           // Field java/lang/System.out:Ljava/io/PrintStream;
  4: getstatic   #2          // String Method 1 start
  7: ldc         #3          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  9: invokevirtual #4
 12: aload_1
 13: monitorexit            // Field java/lang/System.out:Ljava/io/PrintStream;
 14: goto       22
 17: astore_2
 18: aload_1
 19: monitorexit
 20: aload_2
 21: athrow
 22: return
Exception table:
  from   to target type
    4    14   17  any
   17    20   17  any
LineNumberTable:
  line  5: 0
  line  6: 4
  line  7: 12
  line  8: 22
StackMapTable: number_of_entries = 2
  frame_type = 255 /* full_frame */
  offset_delta = 17
  locals = [ class test/SynchronizedDemo, class java/lang/Object ]
  stack = [ class java/lang/Throwable ]
  frame_type = 250 /* chop */
  offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"

```

从上面我们可以看出：

synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置， monitorexit 指令则指明同步代码块的结束位置。当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monior对象存在于每个Java对象的对象头中， synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权.当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

② synchronized 修饰方法的情况

```

public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("synchronized 方法");
    }
}

```

```

public test.SynchronizedDemo2();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
  0: aload_0
     1: invokespecial #1           // Method java/lang/Object.<init>():V
     4: return
LineNumberTable:
  line 3: 0

public synchronized void method();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
  stack=2, locals=1, args_size=1
  0: getstatic    #2           // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc         #3           // String synchronized 鑷规码
  5: invokevirtual #4           // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 5: 0
  line 6: 8
}
SourceFile: "SynchronizedDemo2.java"

```

synchronized 修饰的方法并没有 monitoreenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

70. synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

- 1、ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
- 2、ReentrantLock可以获取各种锁的信息
- 3、ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

71. ConcurrentHashMap的并发度是什么

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗？

72. ReadWriteLock是什么

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

73. FutureTask是什么

FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

74. 如果你提交任务时，线程池队列已满，这时会发生什么

这里区分一下：

如果使用的是无界队列LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务

如果使用的是有界队列比如ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，会根据maximumPoolSize的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue继续满，那么则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy

75. 生产者消费者模型的作用是什么

这个问题很理论，但是很重要：

- 1、通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用
- 2、解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要受到相互的制约

76. 什么是乐观锁和悲观锁

1、乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

2、悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

77. CyclicBarrier和CountDownLatch的区别

两个看上去有点像的类，都在 `java.util.concurrent` 下，都可以用来表示代码运行到某个点上，二者的区别在于：

- 1、`CyclicBarrier` 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；`CountDownLatch` 则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行
- 2、`CyclicBarrier` 只能唤起一个任务，`CountDownLatch` 可以唤起多个任务

3、`CyclicBarrier`可重用，`CountDownLatch`不可重用，计数值为0该`CountDownLatch`就不可再用了

区别	CountDownLatch	CyclicBarrier
计数方式	递减计数	加法计数
可重复利用性	不可重复利用	可重复利用
初始值	初始值为N, N>0	N为0
计数方式	调用countDown, N-1	调用await, N+1
阻塞条件	N>0,调用await一直阻塞	N小于指定值
何时释放等待线程	计数为0时	计数达到指定值N

78. Hashtable的size()方法中明明只有一条语句"return count",为什么还要做同步?

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有多条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是`size()`方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：

1、同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行`Hashtable`的`put`方法添加数据，线程B则可以正常调用`size()`方法读取`Hashtable`中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对`size++`，线程B就已经读取`size`了，那么对于线程B来说读取到的`size`一定是不准确的。而给`size()`方法加了同步之后，意味着线程B调用`size()`方法只有在线程A调用`put`方法完毕之后才可以调用，这样就保证了线程安全性

2、CPU执行代码，执行的不是Java代码，这点很关键，一定得记住。Java代码最终是被翻译成机器码执行的，机器码才是真正可以和硬件电路交互的代码。即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句"return count"假设被翻译成了三句汇编语句执行，一句汇编语句和其机器码做对应，完全可能执行完第一句，线程就切换了。

79. Linux环境下如何查找哪个线程使用CPU最长

这是一个比较偏实践的问题，这种问题我觉得挺有意义的。可以这么做：

- 1、获取项目的pid, jps或者ps -ef | grep java, 这个前面有讲过
- 2、top -H -p pid, 顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号，我笔记本上没有部署Linux环境下的Java工程，因此没有办法截图演示，网友朋友们如果公司是使用Linux环境部署项目的话，可以尝试一下。

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

JVM面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



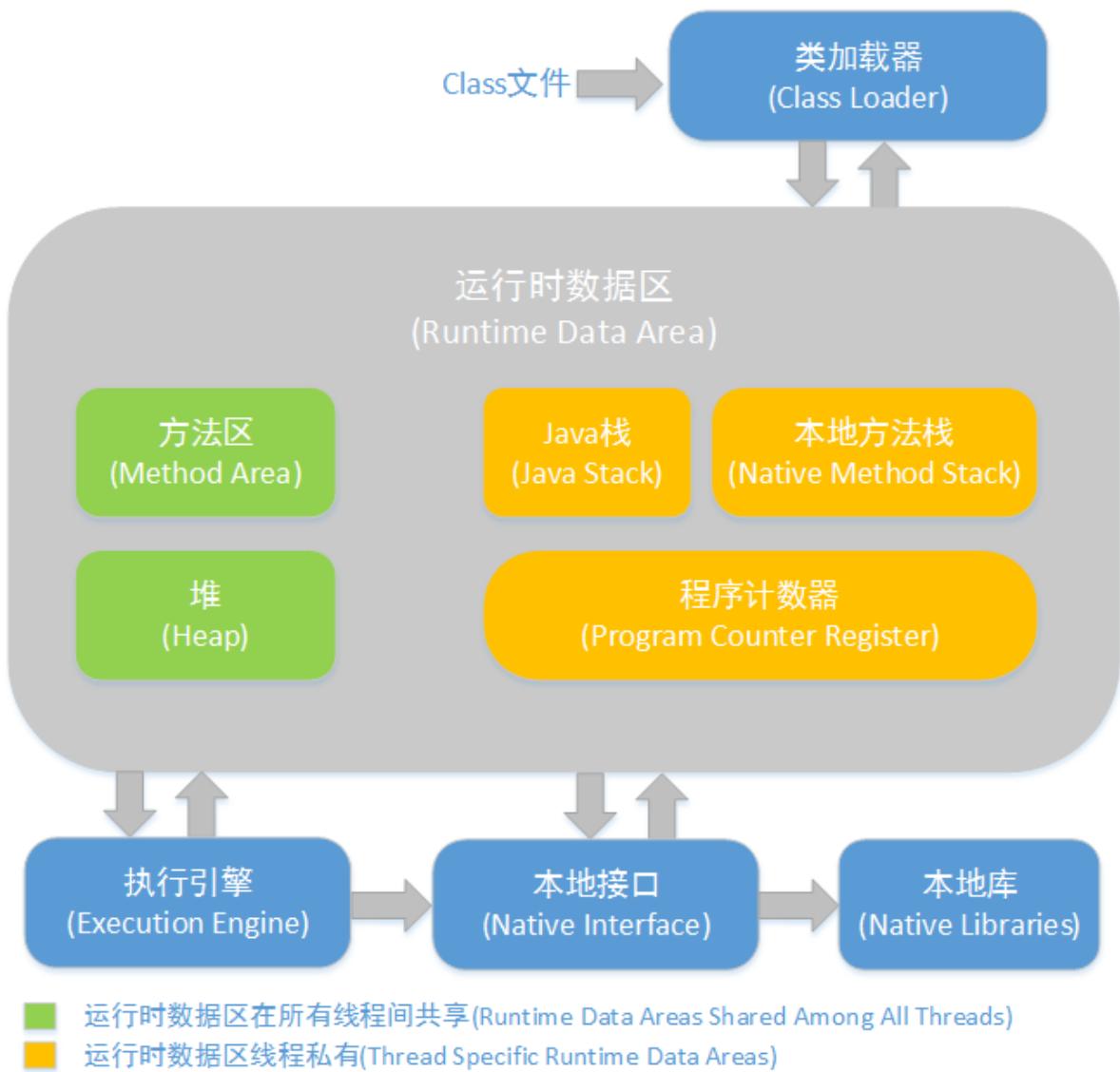
长按识别二维码关注
获取最新面试手册

1. 什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”？

Java虚拟机是一个可以执行Java字节码的虚拟机进程。

Java源文件被编译成能被Java虚拟机执行的字节码文件。Java被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

2. Java内存结构？



方法区和堆是所有线程共享的内存区域；而java栈、本地方法栈和程序员计数器是运行时线程私有的内存区域。

- 1、Java堆 (Heap) ,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
- 2、方法区 (Method Area) ,方法区 (Method Area) 与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
- 3、程序计数器 (Program Counter Register) ,程序计数器 (Program Counter Register) 是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。
- 4、JVM栈 (JVM Stacks) ,与程序计数器一样，Java虚拟机栈 (Java Virtual Machine Stacks) 也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧 (Stack Frame) 用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
- 5、本地方法栈 (Native Method Stacks) ,本地方法栈 (Native Method Stacks) 与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

3. Java内存分配

- 寄存器：我们无法控制。
- 静态域：static 定义的静态成员。
- 常量池：编译时被确定并保存在 .class 文件中的 (final) 常量值和一些文本修饰的符号引用（类和接口的全限定名，字段的名称和描述符，方法和名称和描述符）。
- 非 RAM 存储：硬盘等永久存储空间。
- 堆内存：new 创建的对象和数组，由 Java 虚拟机自动垃圾回收器管理，存取速度慢。
- 栈内存：基本类型的变量和对象的引用变量（堆内存空间的访问地址），速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

4. Java 堆的结构是什么样子的？什么是堆中的永久代（Perm Gen space）？

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些 对象回收掉之前，他们会一直占据堆内存空间。

5. Java 中堆和栈有什么区别？

JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

栈

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配，当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量分配的内存空间，该内存空间可以立即被另作它用。

堆

堆内存用来存放由 new 创建的对象和数组，在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或者对象之后，还可以在栈中定义一个特殊的变量，让栈中的这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或者对象，引用变量就相当于是为数组或者对象起的一个名称。

6. 解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用JVM中的栈空间；

而通过new关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为Eden、Survivor（又可分为From Survivor和To Survivor）、Tenured；

方法区和堆都是各个线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量、JIT编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的100、“hello”和常量都是放在常量池中，常量池是方法区的一部分。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过JVM的启动参数来进行调整，栈空间用光了会引发StackOverflowError，而堆和常量池空间不足则会引发OutOfMemoryError。

```
String str = new String("hello");
```

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而“hello”这个字面量是放在方法区的。

补充1：

较新版本的Java（从Java 6的某个更新开始）中，由于JIT编译器的发展和“逃逸分析”技术的逐渐成熟，栈上分配、标量替换等优化技术使得对象一定分配在堆上这件事情已经变得不那么绝对了。

补充2：

运行时常量池相当于Class文件常量池具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。看看下面代码的执行结果是什么并且比较一下Java 7以前和以后的运行结果是否一致。

```
String s1 = new StringBuilder("go").append("od").toString();
System.out.println(s1.intern() == s1);

String s2 = new StringBuilder("ja").append("va").toString();
System.out.println(s2.intern() == s2);
```

7. JVM内存分哪几个区，每个区的作用是什么？

Java虚拟机主要分为以下一个区：

方法区：

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
3. 该区域是被线程共享的。
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

虚拟机栈：

1. 虚拟机栈也就是我们平常所称的栈内存，它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
2. 虚拟机栈是线程私有的，它的生命周期与线程相同。
3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对

象相关联的位置。局部变量所需的内存空间在编译器间确定

4. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
5. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

堆

Java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

程序计数器

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

8. 怎么获取 Java 程序使用的内存？堆使用的百分比？

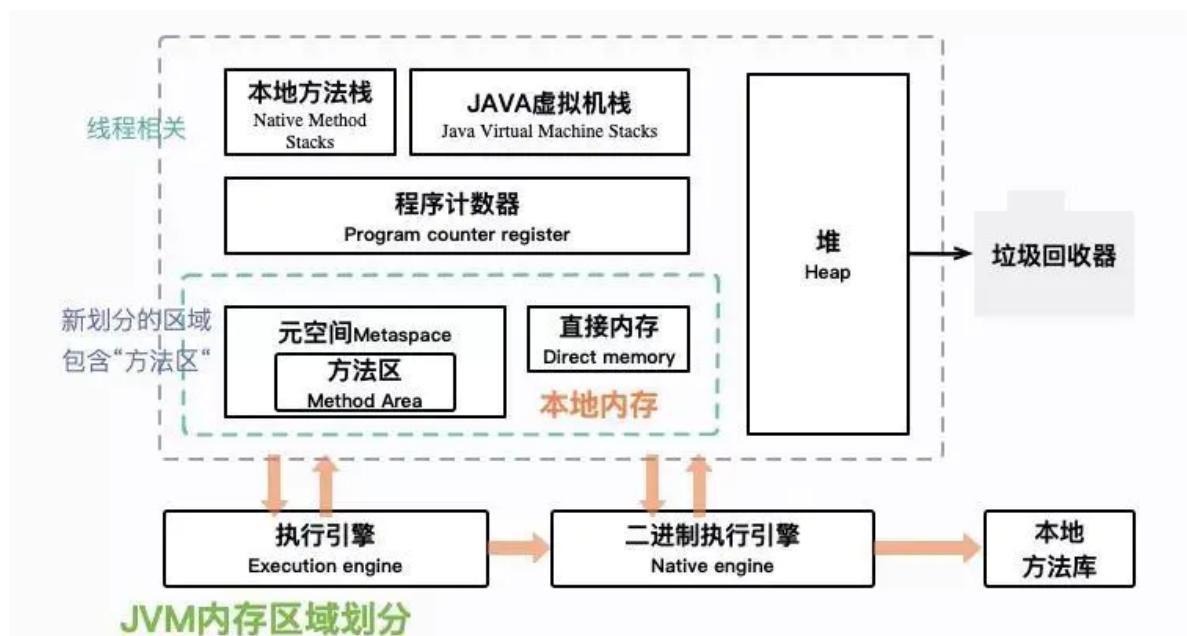
可以通过 `java.lang.Runtime` 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。

通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。

- `Runtime.freeMemory()` 方法返回剩余空间的字节数
- `Runtime.totalMemory()`方法总内存的字节数
- `Runtime.maxMemory()` 返回最大内存的字节数

9. JVM有哪些内存区域？(JVM的内存布局是什么？)

JVM包含堆、元空间、Java虚拟机栈、本地方法栈、程序计数器等内存区域。其中，堆是占用内存最大的一块。我们平常的-Xmx、-Xms等参数，就是针对于堆进行设计的。



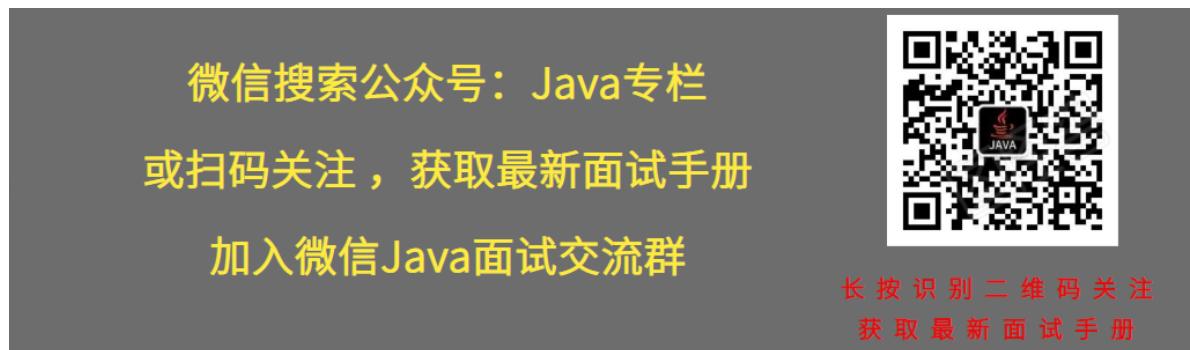
堆：JVM堆中的数据，是共享的，是占用内存最大的一块区域

微信搜索公众号：Java专栏，获取最新面试手册

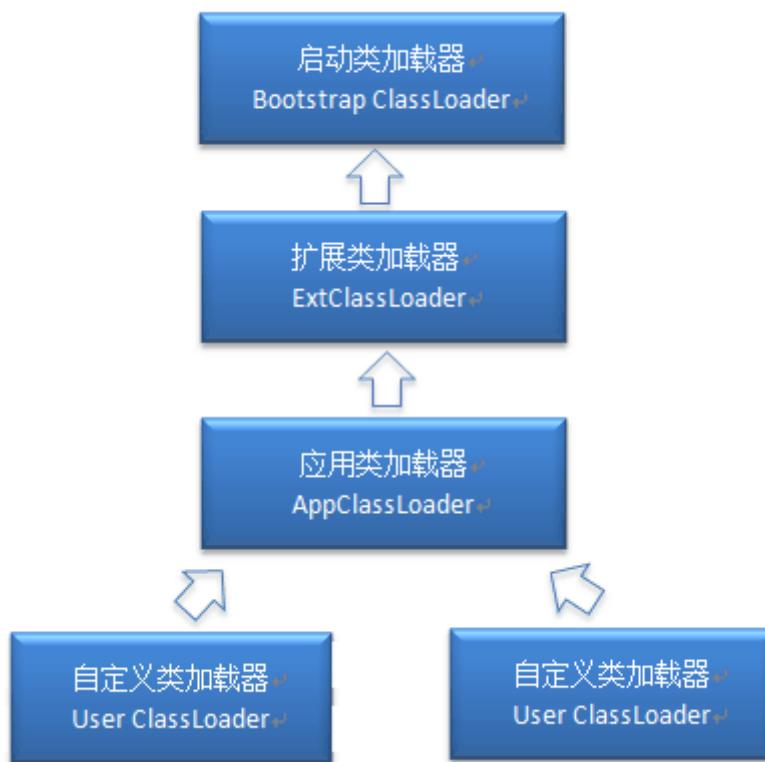
虚拟机栈: Java虚拟机栈，是基于线程的，用来服务字节码指令的运行

程序计数器: 当前线程所执行的字节码的行号指示器

元空间: 方法区就在这里，非堆本地内存：其他的内存占用空间



10. 类加载器



1、启动类加载器：Bootstrap ClassLoader

负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库

2、扩展类加载器：Extension ClassLoader

该加载器由sun.misc.Launcher\$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类），开发者可以直接使用扩展类加载器。

3、应用程序类加载器：Application ClassLoader

该类加载器由sun.misc.Launcher\$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器

11. JVM加载class文件的原理机制?

JVM中类的装载是由类加载器 (ClassLoader) 和它的子类来实现的, Java中的类加载器是一个重要的Java运行时系统组件, 它负责在运行时查找和装入类文件中的类。

由于Java的跨平台性, 经过编译的Java源程序并不是一个可执行程序, 而是一个或多个类文件。当Java程序需要使用某个类时, JVM会确保这个类已经被加载、连接 (验证、准备和解析) 和初始化。类的加载是指把类的.class文件中的数据读入到内存中, 通常是创建一个字节数组读入.class文件, 然后产生与所加载类对应的Class对象。加载完成后, Class对象还不完整, 所以此时的类还不可用。当类被加载后就进入连接阶段, 这一阶段包括验证、准备 (为静态变量分配内存并设置默认的初始值) 和解析 (将符号引用替换为直接引用) 三个步骤。最后JVM对类进行初始化, 包括:

- 1、如果类存在直接的父类并且这个类还没有被初始化, 那么就先初始化父类;
- 2、如果类中存在初始化语句, 就依次执行这些初始化语句。

类的加载是由类加载器完成的, 类加载器包括: 根加载器 (BootStrap) 、扩展加载器 (Extension) 、系统加载器 (System) 和用户自定义类加载器 (java.lang.ClassLoader的子类) 。

从Java 2 (JDK 1.2) 开始, 类加载过程采取了父亲委托机制 (PDM) 。PDM更好的保证了Java平台的安全性, 在该机制中, JVM自带的Bootstrap是根加载器, 其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载, 父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明:

- 1、**Bootstrap**: 一般用本地代码实现, 负责加载JVM基础核心类库 (rt.jar) ;
- 2、**Extension**: 从java.ext.dirs系统属性所指定的目录中加载类库, 它的父加载器是Bootstrap;
- 3、**System**: 又叫应用类加载器, 其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类, 是用户自定义加载器的默认父加载器。

12. Java类加载过程

Java 类加载需要经历一下 7 个过程:

1、加载

加载是类加载的第一个过程, 在这个阶段, 将完成一下三件事情:

- 通过一个类的全限定名获取该类的二进制流。
- 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
- 在内存中生成该类的 Class 对象, 作为该类的数据访问入口。

2、验证

验证的目的是为了确保 Class 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

文件格式验证: 验证字节流是否符合 Class 文件的规范, 如 主次版本号是否在当前虚拟机范围内, 常量池中的常量是否有不被支持的类型.

元数据验证:对字节码描述的信息进行语义分析, 如这个类是否有父类, 是否集成了不被继承的类等。

字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。

符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

3、准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

```
public static int value=123; //在准备阶段 value 初始值为 0 。在初始化阶段才会变为 123 。
```

4、解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

初始化 初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由

5、初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java 程序代码。

6、使用

7、卸载

13. JVM中对象的创建过程

1、拿到内存创建指令

当虚拟机遇到内存创建的指令的时候（new 类名），来到了方法区，找根据new的参数在常量池中定位一个类的符号引用。

2、检查符号引用

检查该符号引用有没有被加载、解析和初始化过，如果没有则执行类加载过程，否则直接准备为新的对象分配内存

3、分配内存

虚拟机为对象分配内存（堆）分配内存分为指针碰撞和空闲列表两种方式；分配内存还要保证并发安全，有两种方式。

1) 指针碰撞

所有的存储空间分为两部分，一部分是空闲，一部分是占用，需要分配空间的时候，只需要计算指针移动的长度即可。

2) 空闲列表

虚拟机维护了一个空闲列表，需要分配空间的时候去查该空闲列表进行分配并对空闲列表做更新。可以看出，内存分配方式是由java堆是否规整决定的，java堆的规整是由垃圾回收机制来决定的。微信搜索公众号：Java专栏，获取最新面试手册

3) 安全性问题的思考

假如分配内存策略是指针碰撞，如果在高并发情况下，多个对象需要分配内存，如果不做处理，肯定会出现线程安全问题，导致一些对象分配不到空间等。

下面是解决方案：

- 线程同步策略

也就是每个线程都进行同步，防止出现线程安全。

- 本地线程分配缓冲

也称TLAB (Thread Local Allocation Buffer)，在堆中为每一个线程分配一小块独立的内存，这样以来就不存并发问题了，Java 层面与之对应的是 ThreadLocal 类的实现

4、初始化

1. 分配完内存后要对对象的头 (Object Header) 进行初始化，这新信息包括：该对象对应类的元数据、该对象的GC代、对象的哈希码。
2. 抽象数据类型默认初始化为null，基本数据类型为0，布尔为false....

5、调用对象的初始化方法

也就是执行构造方法。

14. Java对象结构

Java对象由三个部分组成：对象头、实例数据、对齐填充。

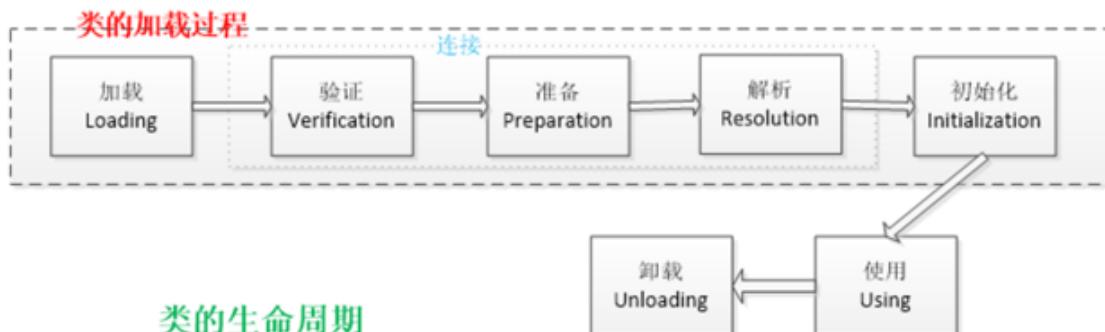
1、对象头由两部分组成，第一部分存储对象自身的运行时数据：哈希码、GC分代年龄、锁标识状态、线程持有的锁、偏向线程ID（一般占32/64 bit）。第二部分是指针类型，指向对象的类元数据类型（即对象代表哪个类）。如果是数组对象，则对象头中还有一部分用来记录数组长度。

2、实例数据用来存储对象真正的有效信息（包括父类继承下来的和自己定义的）

3、对齐填充：JVM要求对象起始地址必须是8字节的整数倍（8字节对齐）

15. 类的生命周期

类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程，如下图：



- 1、加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
微信搜索公众号：Java专栏，获取最新面试手册

2、连接, 连接又包含三块内容: 验证、准备、初始化。 1) 验证, 文件格式、元数据、字节码、符号引用验证; 2) 准备, 为类的静态变量分配内存, 并将其初始化为默认值; 3) 解析, 把类中的符号引用转换为直接引用

3、初始化, 为类的静态变量赋予正确的初始值

4、使用, new出对象程序中使用

5、卸载, 执行垃圾回收

16. 如何判断对象可以被回收?

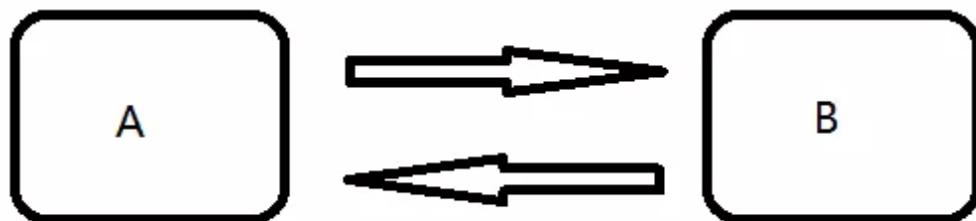
在堆里面存放着Java世界中几乎所有的对象实例, 垃圾收集器在对堆进行回收前, 第一件事就是判断哪些对象已死(可回收).

1、引用计数法

在JDK1.2之前, 使用的是引用计数器算法。

在对象中添加一个引用计数器, 当有地方引用这个对象的时候, 引用计数器的值就+1, 当引用失效的时候, 计数器的值就-1, 当引用计数器被减为零的时候, 标志着这个对象已经没有引用了, 可以回收了!

引用计数法:

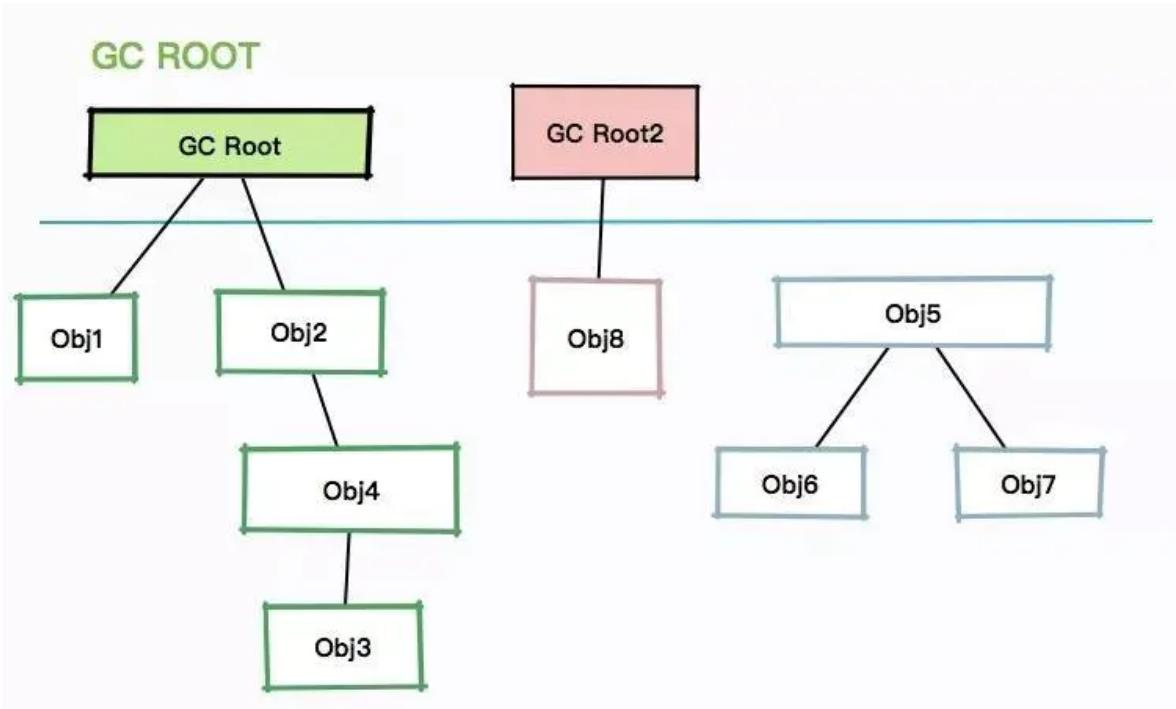


问题:

如果在A类中调用B类的方法, B类中调用A类的方法, 这样当其他所有的引用都消失了之后, A和B还有一个相互的引用, 也就是说两个对象的引用计数器各为1, 而实际上这两个对象都已经没有额外的引用, 已经是垃圾了。但是该算法并不会计算出该类型的垃圾。

2、可达性分析法

在主流商用语言(如Java、C#)的主流实现中, 都是通过可达性分析算法来判定对象是否存活的: 通过一系列的称为 GC Roots 的对象作为起点, 然后向下搜索; 搜索所走过的路径称为引用链/Reference Chain, 当一个对象到 GC Roots 没有任何引用链相连时, 即该对象不可达, 也就说明此对象是不可用的, 如下图: 虽然E和F相互关联, 但它们到GC Roots是不可达的, 因此也会被判定为可回收的对象。



注: 即使在可达性分析算法中不可达的对象, VM也并不是马上对其回收, 因为要真正宣告一个对象死亡, 至少要经历两次标记过程: 第一次是在可达性分析后发现没有与GC Roots相连接的引用链, 第二次是GC对在F-Queue执行队列中的对象进行的小规模标记(对象需要覆盖`finalize()`方法且没被调用过).

17. 如果对象的引用被置为 null, 垃圾收集器是否会立即释放对象占用的内存?

不会, 在下一个垃圾回收周期中, 这个对象将是可被回收的。

18. Java的四种引用, 强弱软虚

1、强引用

强引用是平常中使用最多的引用, 强引用在程序内存不足 (OOM) 的时候也不会被回收, 使用方式:

```
String str = new String("str");
```

2、软引用

软引用在程序内存不足时, 会被回收, 使用方式:

```
// 注意: wrf这个引用也是强引用, 它是指向SoftReference这个对象的,
// 这里的软引用指的是指向new String("str")的引用, 也就是SoftReference类中T
SoftReference<String> wrf = new SoftReference<String>(new String("str"));
```

可用场景: 创建缓存的时候, 创建的对象放进缓存中, 当内存不足时, JVM就会回收早先创建的对象。

3、弱引用

弱引用就是只要JVM垃圾回收器发现了它, 就会将之回收, 使用方式:

微信搜索公众号: Java专栏, 获取最新面试手册

```
WeakReference<String> wrf = new WeakReference<String>(str);
```

可用场景：Java源码中的`java.util.WeakHashMap`中的`key`就是使用弱引用，我的理解就是，一旦我不需要某个引用，JVM会自动帮我处理它，这样我就不需要做其它操作。

4、虚引用

虚引用的回收机制跟弱引用差不多，但是它被回收之前，会被放入`ReferenceQueue`中。注意哦，其它引用是被JVM回收后才被传入`ReferenceQueue`中的。由于这个机制，所以虚引用大多被用于引用销毁前的处理工作。还有就是，虚引用创建的时候，必须带有`ReferenceQueue`，使用例子：

```
PhantomReference<String> prf = new PhantomReference<String>(new String("str"),  
new ReferenceQueue<>());
```

可用场景：对象销毁前的一些操作，比如说资源释放等。`**Object.finalize()`虽然也可以做这类动作，但是这种方式即不安全又低效

上诉所说的几类引用，都是指对象本身的引用，而不是指`Reference`的四个子类的引用（`SoftReference`等）。

19. 什么情况下会发生栈溢出？

栈的大小可以通过`-Xss`参数进行设置，当递归层次太深的时候，就会发生栈溢出。比如循环调用，递归等。

20. GC是什么？为什么要有GC

GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。

Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：

```
System.gc() 或Runtime.getRuntime().gc()
```

但JVM可以屏蔽掉显示的垃圾回收调用。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。

垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

补充：

微信搜索公众号：Java专栏，获取最新面试手册

垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。与垃圾回收相关的JVM参数：

-Xms / -Xmx：堆的初始大小 / 堆的最大大小

-Xmn：堆中年轻代的大小

-XX:-DisableExplicitGC：让System.gc()不产生任何作用

-XX:+PrintGCDetails：打印GC的细节

-XX:+PrintGCDateStamps：打印GC操作的时间戳

-XX:NewSize / XX:MaxNewSize：设置新生代大小/新生代最大大小

-XX:NewRatio：可以设置老生代和新生代的比例

-XX:PrintTenuringDistribution：设置每次新生代GC后输出幸存者乐园中对象年龄的分布

-XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

-XX:TargetSurvivorRatio：设置幸存区的目标使用率

21. 简述 Java 垃圾回收机制。

在 Java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收

22. JVM 的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收（Full GC）。

注：Java 8 中已经移除了永久代，新加了一个叫做元数据区的native 内存区。

23. 什么是分布式垃圾回收（DGC）？它是如何工作的？

DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

24. JVM垃圾处理方法

1、标记-清除算法（老年代）

该算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象（可达性分析），在标记完成后统一清理掉所有被标记的对象。



该算法会有两个问题：

1. 效率问题，标记和清除效率不高。
2. 空间问题：标记清除后会产生大量不连续的内存碎片，空间碎片太多可能会导致在运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集。

所以它一般用于“垃圾不太多的区域，比如老年代”。

2、复制算法（新生代）

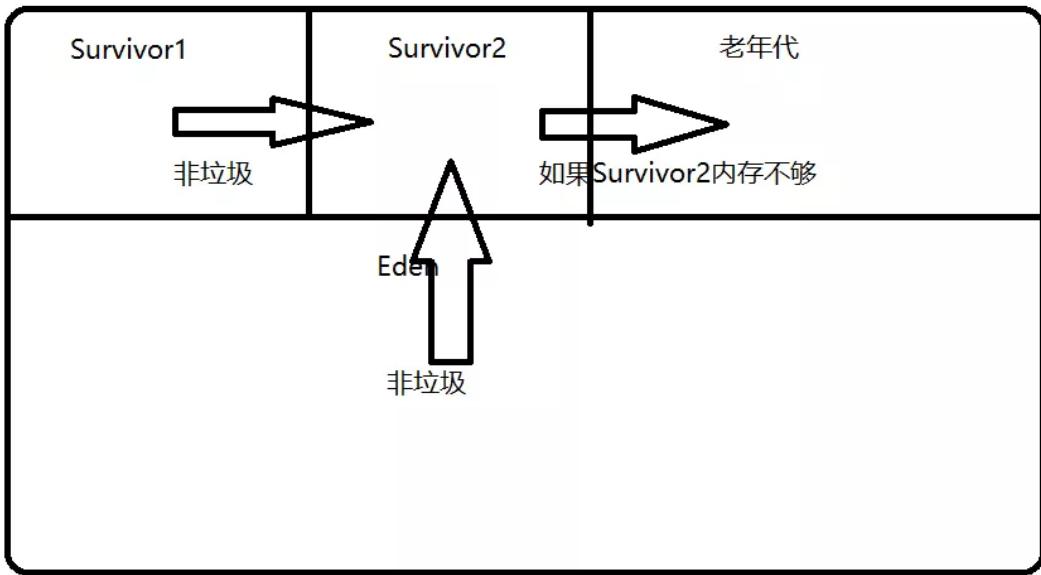
该算法的核心是将可用内存按容量划分为大小相等的两块，每次只用其中一块，当这一块的内存用完，就将还存活的对象（非垃圾）复制到另外一块上面，然后把已使用过的内存空间一次清理掉。

优点：不用考虑碎片问题，方法简单高效。

缺点：内存浪费严重。

现代商用VM的新生代均采用复制算法，但由于新生代中的98%的对象都是生存周期极短的，因此并不需完全按照1 : 1的比例划分新生代空间，而是将新生代划分为一块较大的Eden区和两块较小的Survivor区（HotSpot默认Eden和Survivor的大小比例为8 : 1），每次只用Eden和其中一块Survivor。

当发生MinorGC时，将Eden和Survivor中还存活着的对象一次性地拷贝到另外一块Survivor上，最后清理掉Eden和刚才用过的Survivor的空间。当Survivor空间不够用（不足以保存尚存活的对象）时，需要依赖老年代进行空间分配担保机制，这部分内存直接进入老年代。



复制算法的空间分配担保：

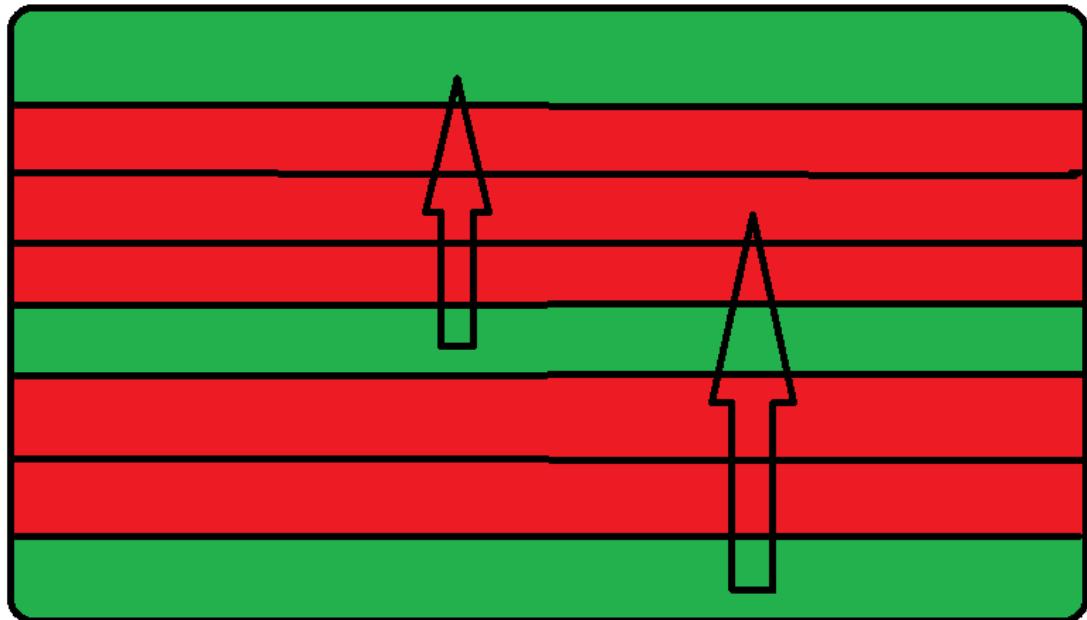
在执行Minor GC前, VM会首先检查老年代是否有足够的空间存放新生代尚存活对象, 由于新生代使用复制收集算法, 为了提升内存利用率, 只使用了其中一个Survivor作为轮换备份, 因此当出现大量对象在Minor GC后仍然存活的情况时, 就需要老年代进行分配担保, 让Survivor无法容纳的对象直接进入老年代, 但前提是老年代需要有足够的空间容纳这些存活对象.

但存活对象的大小在实际完成GC前是无法明确知道的, 因此Minor GC前, VM会先首先检查老年代连续空间是否大于新生代对象总大小或历次晋升的平均大小, 如果条件成立, 则进行Minor GC, 否则进行Full GC(让老年代腾出更多空间).

然而取历次晋升的对象的平均大小也是有一定风险的, 如果某次Minor GC存活后的对象突增, 远远高于平均值的话, 依然可能导致担保失败(Handle Promotion Failure, 老年代也无法存放这些对象了), 此时就只好在失败后重新发起一次Full GC(让老年代腾出更多空间).

3、标记-整理算法（老年代）

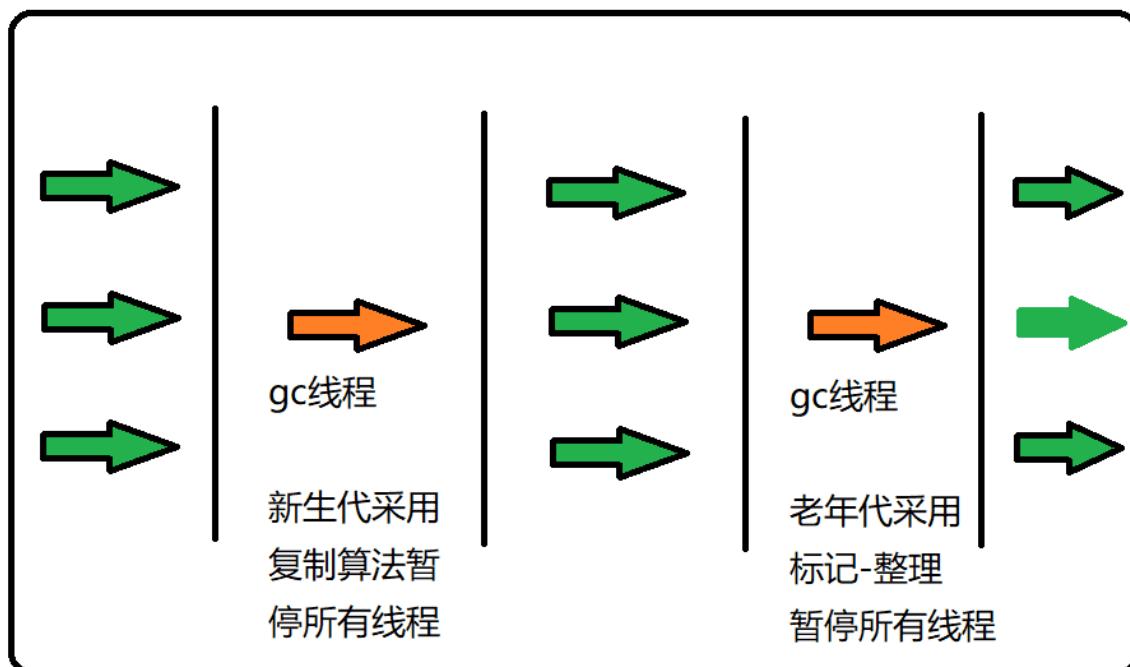
标记清除算法会产生内存碎片问题, 而复制算法需要有额外的内存担保空间, 于是针对老年代的特点, 又有了标记整理算法. 标记整理算法的标记过程与标记清除算法相同, 但后续步骤不再对可回收对象直接清理, 而是让所有存活的对象都向一端移动, 然后清理掉端边界以外的内存.



25. 你能说出来几个垃圾收集器

1、Serial

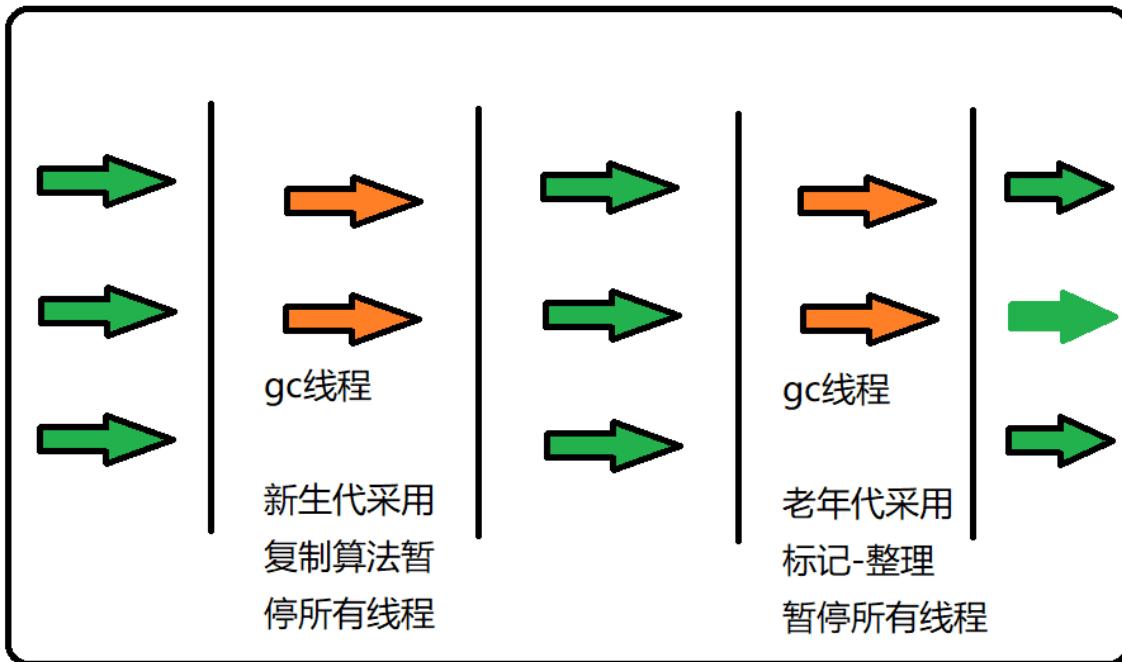
Serial收集器是Hotspot运行在Client模式下的默认新生代收集器, 它在进行垃圾收集时, 会暂停所有的进程, 用一个线程去完成GC工作



特点：简单高效，适合jvm管理内存不大的情况（十兆到百兆）。

2、Parnew

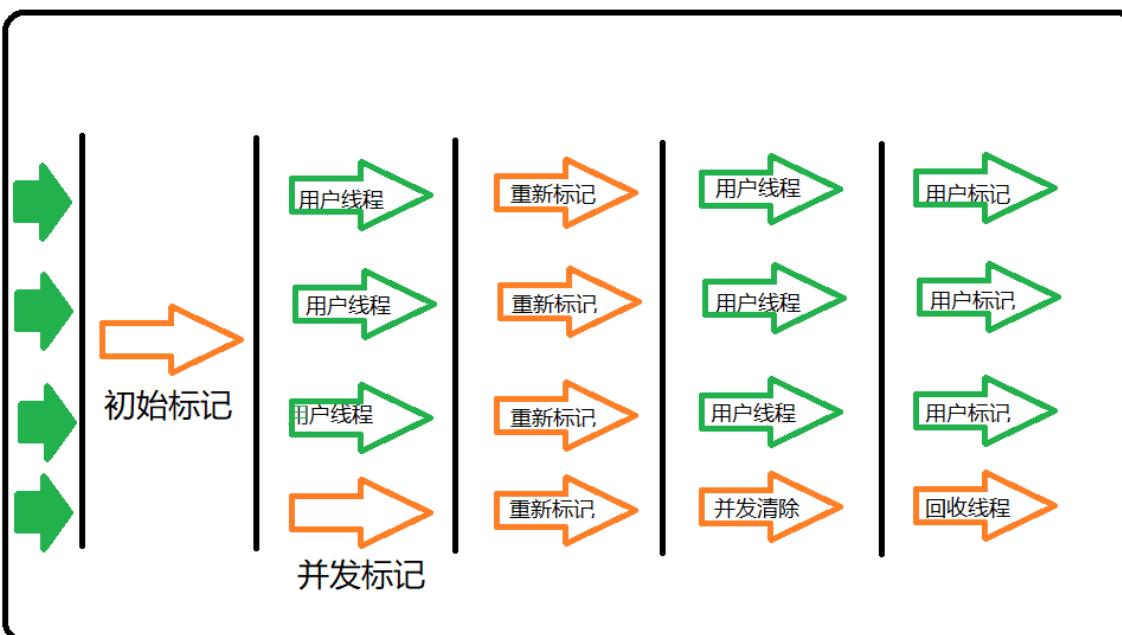
ParNew收集器其实是Serial的多线程版本，回收策略完全一样，但是他们又有着不同。



我们说了Parnew是多线程gc收集，所以它配合多核心的cpu效果更好，如果是一个cpu，他俩效果就差不多。（可用XX:ParallelGCThreads参数控制GC线程数）

3、Cms

CMS(Concurrent Mark Sweep)收集器是一款具有划时代意义的收集器，一款**真正意义上的并发收集器**，虽然现在已经有了理论意义上表现更好的G1收集器，但现在主流互联网企业线上选用的仍是CMS(如Taobao)，又称多并发低暂停的收集器。



由他的英文组成可以看出，它是基于标记-清除算法实现的。整个过程分4个步骤：

1. 初始标记(CMS initial mark):仅只标记一下GC Roots能直接关联到的对象，速度很快
2. 并发标记(CMS concurrent mark: GC Roots Tracing过程)
3. 重新标记(CMS remark):修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录
4. 并发清除(CMS concurrent sweep: 已死对象将会就地释放)

可以看到，初始标记、重新标记需要STW(stop the world 即：挂起用户线程)操作。因为最耗时的操作是并发标记和并发清除。所以总体上我们认为CMS的GC与用户线程是并发运行的。

优点：并发收集、低停顿

缺点：

1. CMS默认启动的回收线程数=(CPU数目+3)*4

当CPU数>4时, GC线程最多占用不超过25%的CPU资源, 但是当CPU数<=4时, GC线程可能就会过多的占用用户CPU资源, 从而导致应用程序变慢, 总吞吐量降低.

2. 无法清除浮动垃圾 (GC运行到并发清除阶段时用户线程产生的垃圾), 因为用户线程是需要内存的, 如果浮动垃圾施放不及时, 很可能就造成内存溢出, 所以CMS不能像别的垃圾收集器那样等老年代几乎满了才触发, CMS提供了参数 `-xx:CMSInitiatingOccupancyFraction` 来设置GC触发百分比(1.6后默认92%), 当然我们还得设置启用该策略 `-xx:+UseCMSInitiatingOccupancyOnly`
3. 因为CMS采用标记-清除算法, 所以可能会带来很多的碎片, 如果碎片太多没有清理, jvm会因为无法分配大对象内存而触发GC, 因此CMS提供了 `-xx:+UseCMSCompactAtFullCollection` 参数, 它会在GC执行完后接着进行碎片整理, 但是又会有个问题, 碎片整理不能并发, 所以必须单线程去处理, 所以如果每次GC完都整理用户线程stop的时间累积会很长, 所以
`XX:CMSFullGCsBeforeCompaction` 参数设置隔几次GC进行一次碎片整理 (默认为0) 。

4. G1

同优秀的CMS垃圾回收器一样, G1也是关注最小时延的垃圾回收器, 也同样适合大尺寸堆内存的垃圾收集, 官方也推荐使用G1来代替选择CMS。G1最大的特点是**引入分区**的思路, **弱化分代**的概念, 合理利用垃圾收集各个周期的资源, 解决了其他收集器甚至CMS的众多缺陷。



因为每个区都有E、S、O代，所以在G1中，不需要对整个Eden等代进行回收，而是寻找可回收对象比较多的区，然后进行回收（虽然也需要STW操作，但是花费的时间是很少的），保证高效率。

新生代收集

G1的新生代收集跟ParNew类似，如果存活时间超过某个阈值，就会被转移到S/O区。

年轻代内存由一组不连续的heap区组成，这种方法使得可以动态调整各代区域的大小

老年代收集

分为以下几个阶段：

1. 初始标记 (Initial Mark: Stop the World Event)

在G1中，该操作附着一次年轻代GC，以标记Survivor中有可能引用到老年代对象的Regions.

2. 扫描根区域 (Root Region Scanning: 与应用程序并发执行)

扫描Survivor中能够引用到老年代的references. 但必须在Minor GC触发前执行完

3. 并发标记 (Concurrent Marking : 与应用程序并发执行)

在整个堆中查找存活对象，但该阶段可能会被Minor GC中断

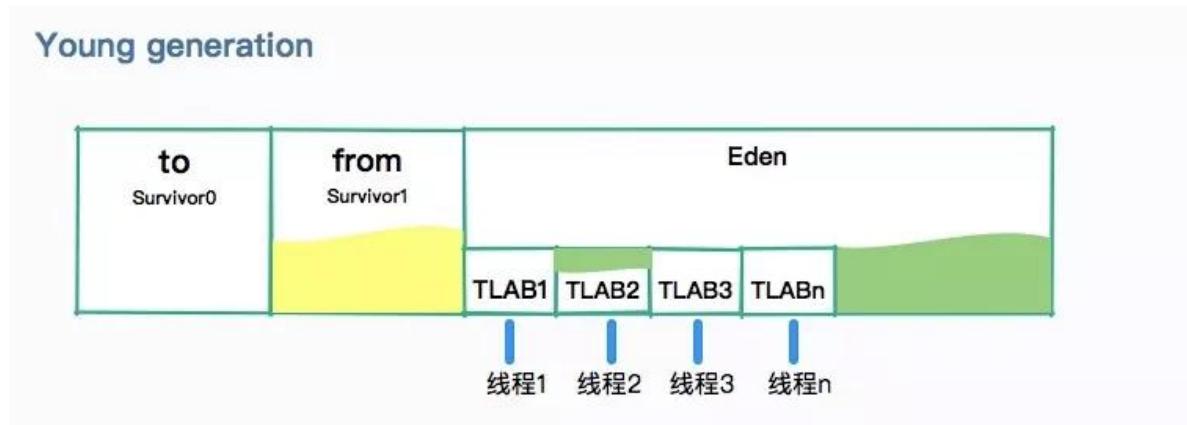
4. 重新标记 (Remark : Stop the World Event)

完成堆内存中存活对象的标记。使用snapshot-at-the-beginning(SATB, 起始快照)算法，比CMS所用算法要快得多(空Region直接被移除并回收，并计算所有区域的活跃度).

5. 清理 (Cleanup : Stop the World Event and Concurrent)

在含有存活对象和完全空闲的区域上进行统计(STW)、擦除Remembered Sets(使用Remembered Set来避免扫描全堆，每个区都有对应一个Set用来记录引用信息、读写操作记录)(STW)、重置空regions并将他们返还给空闲列表(free list)(Concurrent)

26. 简单描述一下（分代）垃圾回收的过程



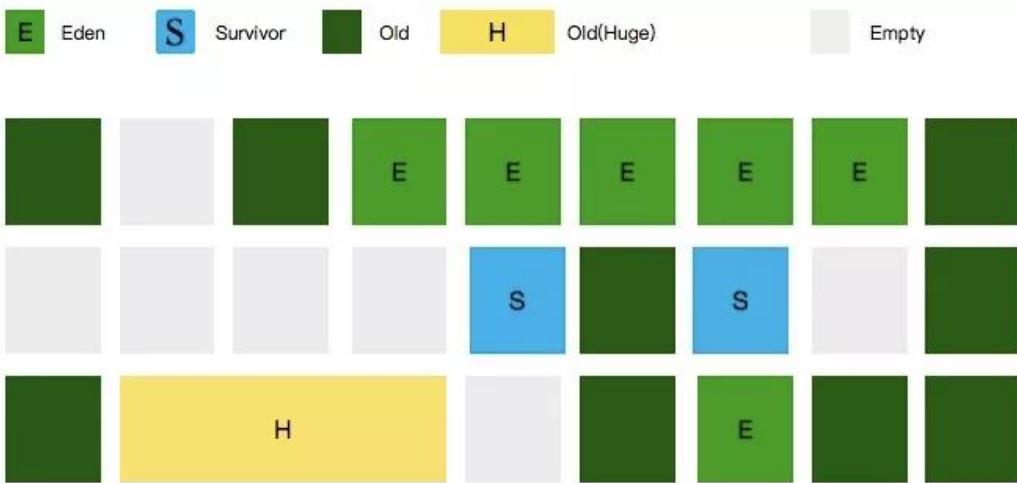
分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

当年轻代中的Eden区分配满的时候，就会触发年轻代的GC（Minor GC）。具体过程如下：

- 1、在Eden区执行了第一次GC之后，存活的对象会被移动到其中一个Survivor分区（以下简称from）
- 2、Eden区再次GC，这时会采用复制算法，将Eden和from区一起清理。存活的对象会被复制到to区。接下来，只需要清空from区就可以了

27. 你都用过G1垃圾回收器的哪几个重要参数？



最重要的是MaxGCPauseMillis，可以通过它设定G1的目标停顿时间，它会尽量的去达成这个目标。

G1HeapRegionSize可以设置小堆区的大小，一般是2的次幂。

InitiatingHeapOccupancyPercent，启动并发GC时的堆内存占用百分比。G1用它来触发并发GC周期，基于整个堆的使用率，而不只是某一代内存的使用比例，默认是45%。

再多？不是专家，就没必要要求别人也是。

28. 有什么堆外内存的排查思路？

进程占用的内存，可以使用top命令，看RES段占用的值。如果这个值大大超出我们设定的最大堆内存，则证明堆外内存占用了很大的区域。

使用gdb可以将物理内存dump下来，通常能看到里面的内容。更加复杂的分析可以使用perf工具，或者谷歌开源的gperftools。那些申请内存最多的native函数，很容易就可以找到。

29. 串行 (serial) 收集器和吞吐量 (throughput) 收集器的区别

吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。而串行收集器对大多数的小应用（在现代处理器上需要大概100M左右的内存）就足够了。

30. GC日志的real、user、sys是什么意思？

1、real 实际花费的时间，指的是从开始到结束所花费的时间。比如进程在等待I/O完成，这个阻塞时间也会被计算在内。

2、user 指的是进程在用户态（User Mode）所花费的时间，只统计本进程所使用的时间，是指多核。

3、sys 指的是进程在核心态（Kernel Mode）花费的CPU时间量，指的是内核中的系统调用所花费的时间，只统计本进程所使用的时间。

这个是用来看日志用的，如果你不看日志，那不了解也无妨。不过，这三个参数的意义，在你能看到的地方，基本上都是一致的，比如操作系统。

31. GC日志分析

摘录GC日志一部分（前部分为年轻代gc回收；后部分为full gc回收）：

```
2016-07-05T10:43:18.093+0800: 25.395: [GC [PSYoungGen: 274931K->10738K(274944K)]  
371093K->147186K(450048K), 0.0668480 secs] [Times: user=0.17 sys=0.08, real=0.07  
secs]  
2016-07-05T10:43:18.160+0800: 25.462: [Full GC [PSYoungGen: 10738K->0K(274944K)]  
[ParOldGen: 136447K->140379K(302592K)] 147186K->140379K(577536K) [PSPermGen:  
85411K->85376K(171008K)], 0.6763541 secs] [Times: user=1.75 sys=0.02, real=0.68  
secs]
```

通过上面日志分析得出，PSYoungGen、ParOldGen、PSPermGen属于Parallel收集器。其中PSYoungGen表示gc回收前后年轻代的内存变化；ParOldGen表示gc回收前后老年代的内存变化；PSPermGen表示gc回收前后永久区的内存变化。young gc主要是针对年轻代进行内存回收比较频繁，耗时短；full gc会对整个堆内存进行回城，耗时长，因此一般尽量减少full gc的次数。

32. MinorGC, MajorGC、FullGC都什么时候发生？

MinorGC在年轻代空间不足的时候发生，MajorGC指的是老年代的GC，出现MajorGC一般经常伴有MinorGC。

FullGC有三种情况。

- 1、当老年代无法再分配内存的时候
- 2、元空间不足的时候
- 3、显示调用System.gc的时候。另外，像CMS一类的垃圾回收器，在MinorGC出现promotion failure的时候也会发生FullGC

33. 新生代、老年代、持久代都存储哪些东西

1、新生代：

1. 方法中new一个对象，就会先进入新生代。

2、老年代：

1. 新生代中经历了N次垃圾回收仍然存活的对象就会被放到老年代中。
2. 大对象一般直接放入老年代。
3. 当Survivor空间不足。需要老年代担保一些空间，也会将对象放入老年代。

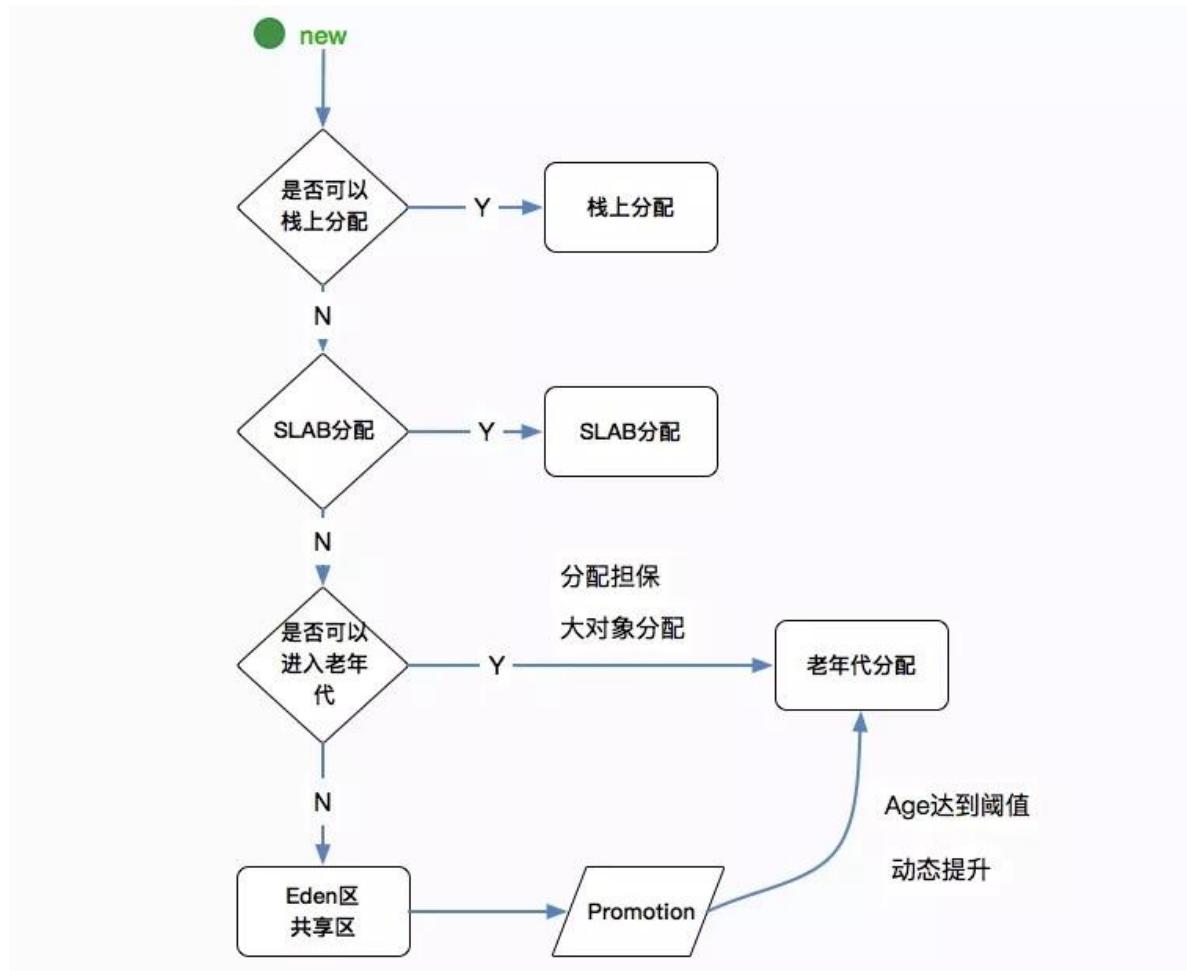
3、永久代：

指的就是方法区。

34. 对象是怎么从年轻代进入老年代的?

这是老掉牙的题目了。在下面四种情况下，对象会从年轻代进入老年代。

- 1、如果对象够老，会通过提升（Promotion）进入老年代，这一般是根据对象的年龄进行判断的。
- 2、动态对象年龄判定。有的垃圾回收算法，比如G1，并不要求age必须达到15才能晋升到老年代，它会使用一些动态的计算方法。
- 3、分配担保。当 Survivor 空间不够的时候，就需要依赖其他内存（指老年代）进行分配担保。这个时候，对象也会直接在老年代上分配。
- 4、超出某个大小的对象将直接在老年代分配。不过这个值默认为0，意思是全部首选Eden区进行分配。



35. 可达性算法中，哪些对象可作为GC Roots对象。

- 1、虚拟机栈中引用的对象
- 2、方法区静态成员引用的对象
- 3、方法区常量引用对象
- 4、本地方法栈JNI引用的对象

36. Java中会存在内存泄漏吗， 请简单描述。

先解释什么是内存泄漏：

所谓内存泄露就是指一个不再被程序使用的对象或变量一直被占据在内存中。java中有垃圾回收机制，它可以保证当对象不再被引用的时候，对象将自动被垃圾回收器从内存中清除掉。

由于Java使用有向图的方式进行垃圾回收管理，可以消除引用循环的问题，例如有两个对象，相互引用，只要它们和根进程不可达，那么GC也是可以回收它们的。

java中的内存泄露的情况：

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景，通俗地说，就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的，这就是java中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中(例如放在一个全局map对象中)，然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。

37. 什么时候进行MinGC和FullGC

MinGC:

1. 当Eden区满时,触发Minor GC.

FullGC:

1. 调用System.gc时，系统建议执行Full GC，但是不必然执行
2. 老年代空间不足
3. 方法区空间不足
4. 通过Minor GC后进入老年代的平均大小大于老年代的剩余空间
5. 堆中分配很大的对象，而老年代没有足够的空间

38. System.gc() 和 Runtime.gc() 会做什么事情？

这两个方法用来提示JVM要进行垃圾回收。但是，立即开始还是延迟进行垃圾回收是取决于JVM的。

39. 垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

40. 垃圾收集算法

GC最基础的算法有三种： 标记 -清除算法、 复制算法、 标记-压缩算法， 我们常用的垃圾回收器一般都采用分代收集算法。

1、标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

2、复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

3、标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

4、分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

41. 调优命令有哪些？

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

1、jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

2、jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

3、jmap，JVM Memory Map命令用于生成heap dump文件

4、jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看

5、jstack，用于生成java虚拟机当前时刻的线程快照。

6、jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

42. 你知道哪些JVM性能调优

- 设定堆内存大小

-Xmx：堆内存最大限制。

- 设定新生代大小。新生代不宜太小，否则会有大量对象涌入老年代

-XX:NewSize：新生代大小

-XX:NewRatio 新生代和老生代占比

-XX:SurvivorRatio：伊甸园空间和幸存者空间的占比

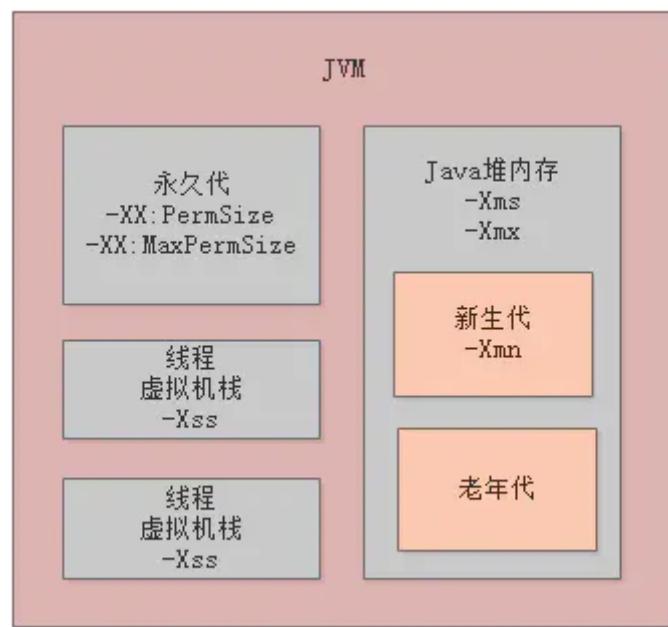
- 设定垃圾回收器 年轻代用 -XX:+UseParNewGC 年老代用-XX:+UseConcMarkSweepGC

43. 常用的调优工具有哪些

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

- 1、jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控
- 2、jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。
- 3、MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
- 4、GChisto，一款专业分析gc日志的工具

44. 跟JVM内存相关的几个核心参数图解



45. OOM你遇到过哪些情况，SOF你遇到过哪些情况

OOM：

1、**OutOfMemoryError**异常

除了程序计数器外，虚拟机内存的其他几个运行时区域都有可能发生OutOfMemoryError(OOM)异常的可能。

Java Heap 溢出：

一般的异常信息：java.lang.OutOfMemoryError:Java heap space
微信搜索公众号：Java专栏，获取最新面试手册

java堆用于存储对象实例，我们只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。

出现这种异常，一般手段是先通过内存映像分析工具(如Eclipse Memory Analyzer)对dump出来的堆转存快照进行分析，重点是确认内存中的对象是否是必要的，先分清是因为内存泄漏(Memory Leak)还是内存溢出(Memory Overflow)。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收。

如果不存在泄漏，那就应该检查虚拟机的参数(-Xmx与-Xms)的设置是否适当。

2、虚拟机栈和本地方法栈溢出

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常

这里需要注意当栈的大小越大可分配的线程数就越少。

3、运行时常量池溢出

异常信息：java.lang.OutOfMemoryError:PermGenspace

如果要向运行时常量池中添加内容，最简单的做法就是使用String.intern()这个Native方法。该方法的作用是：如果池中已经包含一个等于此String的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize和-XX:MaxPermSize限制方法区的大小，从而间接限制其中常量池的容量。

4、方法区溢出

方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。也有可能是方法区中保存的class对象没有被及时回收掉或者class信息占用的内存超过了我们配置。

异常信息：java.lang.OutOfMemoryError:PermGenspace

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收，判定条件是很苛刻的。在经常动态生成大量Class的应用中，要特别注意这点。

SOF (堆栈溢出StackOverflow) :

StackOverflowError 的定义：当应用程序递归太深而发生堆栈溢出时，抛出该错误。

因为栈一般默认为1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过1m而导致溢出。

栈溢出的原因：递归调用，大量循环或死循环，全局变量是否过多，数组、List、map数据过大。

46. finalize() 方法什么时候被调用？析构函数 (finalization) 的目的是什么？

垃圾回收器 (garbage collector) 决定回收某对象时，就会运行该对象的 finalize() 方法但是在 Java 中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说 finalize() 可能永远不被执行，显然指望它做收尾工作是靠不住的。

那么 finalize() 究竟是做什么的呢？

它最主要的用途是回收特殊渠道申请的内存。Java 程序有垃圾回收器，所以一般情况下内存问题不用程序员操心。但有一种 JNI (Java Native Interface) 调用non-Java 程序 (C 或 C++)， finalize() 的工作就是回收这部分的内存。

47. 你都有哪些手段用来排查内存溢出？

(这个话题很大，可以从实践环节中随便摘一个进行总结，下面举例一个最普通的)

你可以来一个中规中矩的回答：

内存溢出包含很多种情况，我在平常工作中遇到最多的就是堆溢出。有一次线上遇到故障，重新启动后，使用jstat命令，发现Old区在一直增长。我使用jmap命令，导出了一份线上堆栈，然后使用MAT进行分析。通过对GC Roots的分析，我发现了一个非常大的HashMap对象，这个原本是有位同学做缓存用的，但是一个无界缓存，造成了堆内存占用一直上升。后来，将这个缓存改成 guava的Cache，并设置了弱引用，故障就消失了。

这个回答不是十分出彩，但着实是常见问题，让人挑不出毛病。

48. 生产上如何配置垃圾收集器的？



首先是内存大小问题，基本上每一个内存区域我都会设置一个上限，来避免溢出问题，比如元空间。通常，堆空间我会设置成操作系统的2/3（这是想给其他进程和操作系统预留一些时间），超过8GB的堆优先选用G1。

接下来，我会对JVM进行初步优化。比如根据老年代的对象提升速度，来调整年轻代和老年代之间的比例。

再接下來，就是专项优化，主要判断的依据就是系统容量、访问延迟、吞吐量等。我们的服务是高并发的，所以对STW的时间非常敏感。

我会通过记录详细的GC日志，来找到这个瓶颈点，借用gceasy（重点）这样的日志分析工具，很容易定位到问题。之所以选择采用工具，是因为gc日志看起来实在是太麻烦了，gceasy号称是AI学习分析问题，可视化做的较好。

49. 假如生产环境CPU占用过高，请谈谈你的分析思路和定位。

top

top -Hp \$pid

printf %x \$tid

jstack \$pid > \$pid.log

less \$pid.log

这个可真是太太太常见了，不过已经烂大街了。如果你还是一个有经验的开发者，不知道的话，需要反省一下了。

首先，使用top -H命令获取占用CPU最高的线程，并将它转化为16进制。

然后，使用jstack命令获取应用的栈信息，搜索这个16进制。这样能够方便的找到引起CPU占用过高的具体原因。

如果有条件的话，直接使用arthas就行操作就好了，不用再做这些费事费力的操作。

50. 对于JDK自带的监控和性能分析工具用过哪些？

jps: 用来显示Java进程；

jstat: 用来查看GC；

jmap: 用来dump堆；

jstack: 用来dump栈；

jhsdb: 用来查看执行中的内存信息；

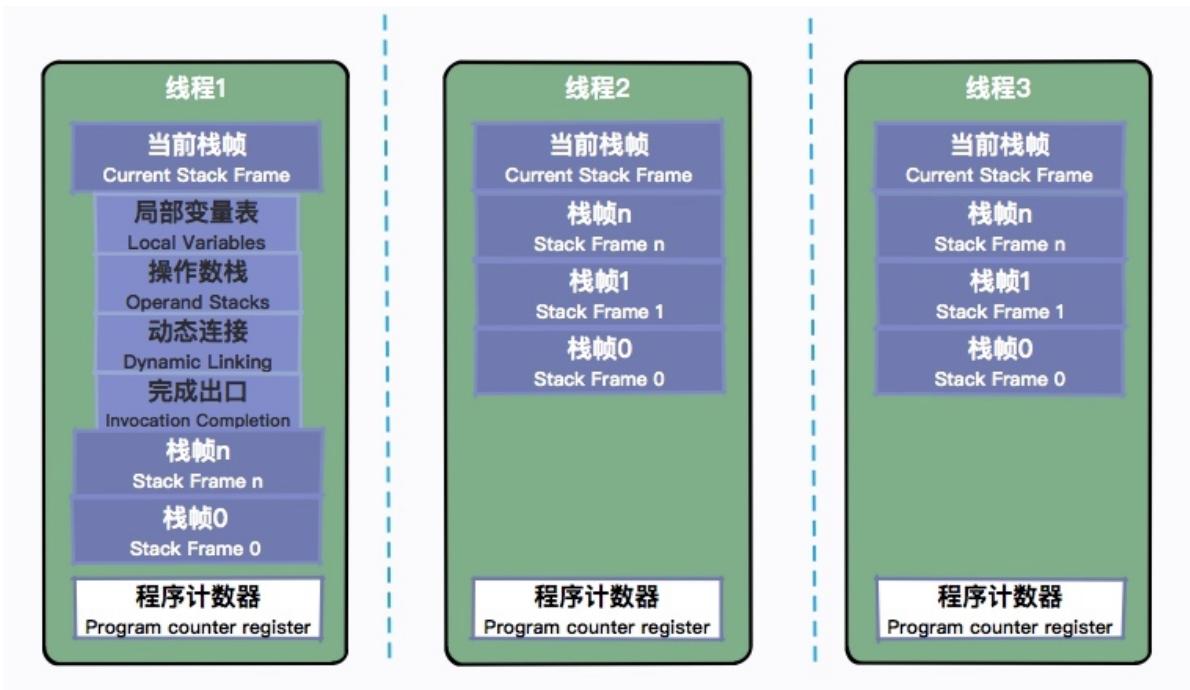
都是非常常用的工具，要熟练掌握。因为线上环境通常都有很多限制，用不了图形化工具。当出现这些情况，上面的命令就是救命的。

51. 栈帧都有哪些数据？

JVM的运行是基于栈的，和C语言的栈类似，它的大多数数据都是在堆里面的，只有少部分运行时的数据存在于栈上。

在JVM中，每个线程栈里面的元素，就叫栈帧。

栈帧包含：局部变量表、操作数栈、动态连接、返回地址等。

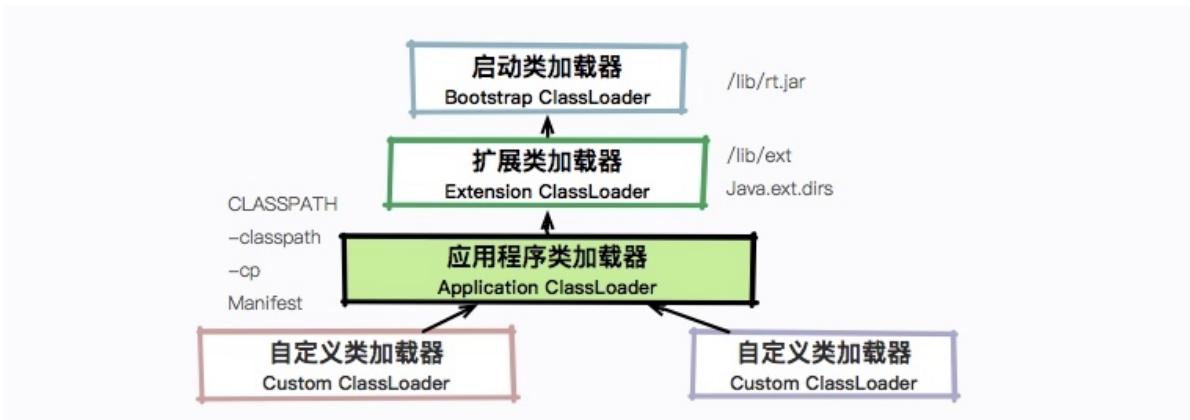


52. JIT是什么？

为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化。完成这个任务的编译器，就称为即时编译器（Just In Time Compiler），简称JIT编译器。

53. Java的双亲委托机制是什么？

它的意思是，除了顶层的启动类加载器以外，其余的类加载器，在加载之前，都会委派给它的父加载器进行加载。这样层层向上传递，直到祖先们都无法胜任，它才会真正的加载。



Java默认是这种行为。当然Java中也有很多打破双亲行为的骚操作，比如SPI（JDBC驱动加载），OSGI等。

54. 有哪些打破了双亲委托机制的案例？

- 1、Tomcat可以加载自己目录下的class文件，并不会传递给父类的加载器。
- 2、Java的SPI，发起者是BootstrapClassLoader，BootstrapClassLoader已经是最上层的了。它直接获取了AppClassLoader进行驱动加载，和双亲委派是相反的。。

55. invokedynamic指令是什么？

属于比较高级的题目。没看过虚拟机的一般是不知道的。所以如果你不太熟悉，不要气馁，加油！（小拳拳锤你胸口）。

invokedynamic是Java7之后新加入的字节码指令，使用它可以实现一些动态类型语言的功能。我们使用的Lambda表达式，在字节码上就是invokedynamic指令实现的。它的功能有点类似反射，但它是使用方法句柄实现的，执行效率更高。

56. safepoint是什么？

STW并不会只发生在内存回收的时候。现在程序员这么卷，碰到几次safepoint的问题几率也是比较大的。

当发生GC时，用户线程必须全部停下来，才可以进行垃圾回收，这个状态我们可以认为JVM是安全的(safe)，整个堆的状态是稳定的。



如果在GC前，有线程迟迟进入不了safepoint，那么整个JVM都在等待这个阻塞的线程，造成了整体GC的时间变长。

数据结构与算法

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. 什么是数据结构?

简单地说，数据结构是以某种特定的布局方式存储数据的容器。这种“布局方式”决定了数据结构对于某些操作是高效的，而对于其他操作则是低效的。首先我们需要理解各种数据结构，才能在处理实际问题时选取最合适的数据结构。

2. 为什么我们需要数据结构?

数据是计算机科学当中最关键的实体，而数据结构则可以将数据以某种组织形式存储，因此，数据结构的价值不言而喻。

无论你以何种方式解决何种问题，你都需要处理数据——无论是涉及员工薪水、股票价格、购物清单，还是只是简单的电话簿问题。

数据需要根据不同的场景，按照特定的格式进行存储。有很多数据结构能够满足以不同格式存储数据的需求。

3. 常见的数据结构

首先列出一些最常见的数据结构，我们将逐一说明：

- 数组
- 栈
- 队列
- 链表
- 树
- 图
- 字典树（这是一种高效的树形结构，但值得单独说明）
- 散列表（哈希表）

1. 数组

数组是最简单、也是使用最广泛的数据结构。栈、队列等其他数据结构均由数组演变而来。下图是一个包含元素（1, 2, 3和4）的简单数组，数组长度为4。



每个数据元素都关联一个正数值，我们称之为索引，它表明数组中每个元素所在的位置。大部分语言将初始索引定义为零。关注Java技术栈微信公众号，回复“面试”获取更多博主精心整理的面试题。

以下是数组的两种类型：

- 一维数组（如上所示）
- 多维数组（数组的数组）

数组的基本操作

- Insert——在指定索引位置插入一个元素
- Get——返回指定索引位置的元素
- Delete——删除指定索引位置的元素
- Size——得到数组所有元素的数量

面试中关于数组的常见问题

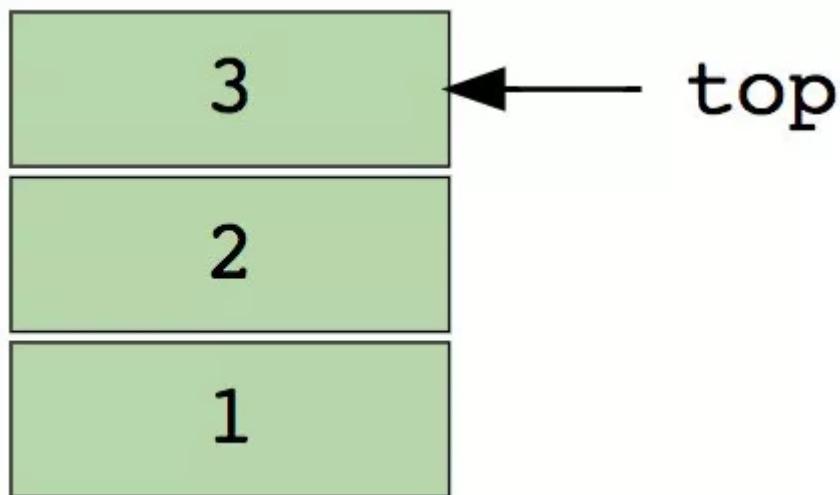
- 寻找数组中第二小的元素
- 找到数组中第一个不重复出现的整数
- 合并两个有序数组
- 重新排列数组中的正值和负值

2. 栈

著名的撤销操作几乎遍布任意一个应用。但你有没有思考过它是如何工作的呢？这个问题的解决思路是按照将最后的状态排列在先的顺序，在内存中存储历史工作状态（当然，它会受限于一定的数量）。这没办法用数组实现。但有了栈，这就变得非常方便了。

可以把栈想象成一列垂直堆放的书。为了拿到中间的书，你需要移除放置在这上面的所有书。这就是 LIFO（后进先出）的工作原理。

下图是包含三个数据元素（1, 2和3）的栈，其中顶部的3将被最先移除：



栈的基本操作

- Push——在顶部插入一个元素
- Pop——返回并移除栈顶元素
- isEmpty——如果栈为空，则返回true
- Top——返回顶部元素，但并不移除它

面试中关于栈的常见问题

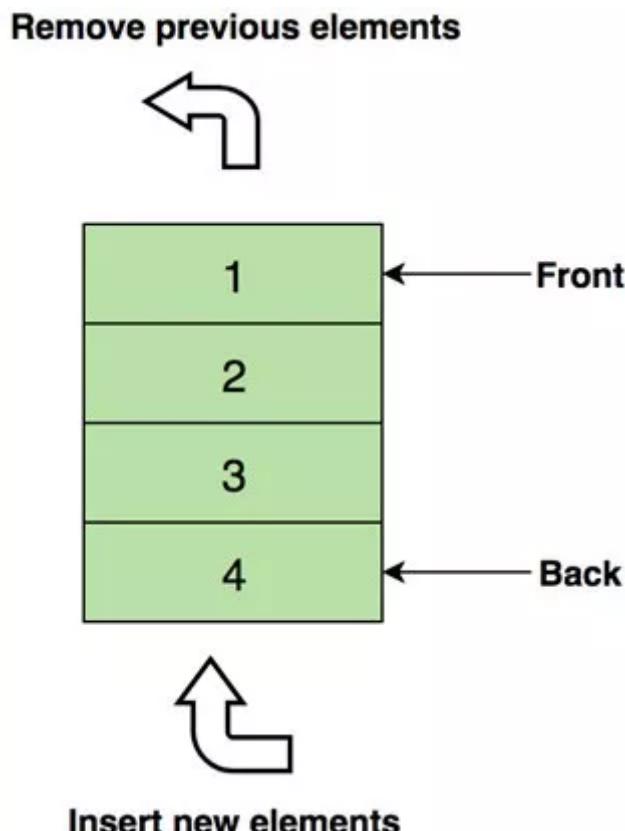
- 使用栈计算后缀表达式
- 对栈的元素进行排序
- 判断表达式是否括号平衡

3. 队列

与栈相似，队列是另一种顺序存储元素的线性数据结构。栈与队列的最大差别在于栈是LIFO（后进先出），而队列是FIFO，即先进先出。

一个完美的队列现实例子：售票亭排队队伍。如果有新人加入，他需要到队尾去排队，而非队首——排在前面的人会先拿到票，然后离开队伍。

下图是包含四个元素（1, 2, 3和4）的队列，其中在顶部的1将被最先移除：



移除先入队的元素、插入新元素

队列的基本操作

- Enqueue()——在队列尾部插入元素
- Dequeue()——移除队列头部的元素
- isEmpty()——如果队列为空，则返回true
- Top()——返回队列的第一个元素

面试中关于队列的常见问题

- 使用队列表示栈
- 对队列的前k个元素倒序
- 使用队列生成从1到n的二进制数

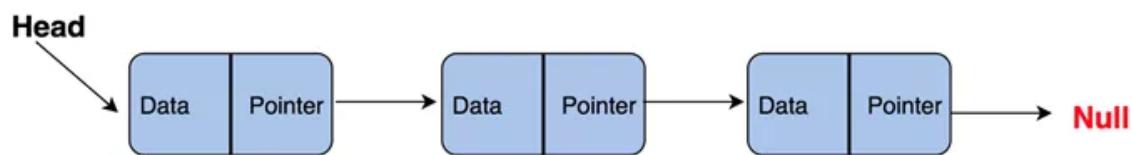
4. 链表

链表是另一个重要的线性数据结构，乍一看可能有点像数组，但在内存分配、内部结构以及数据插入和删除的基本操作方面均有所不同。关注Java技术栈微信公众号，回复“面试”获取更多博主精心整理的面试题。

链表就像一个节点链，其中每个节点包含着数据和指向后续节点的指针。链表还包含一个头指针，它指向链表的第一个元素，但当列表为空时，它指向null或无具体内容。

链表一般用于实现文件系统、哈希表和邻接表。

这是链表内部结构的展示：



链表包括以下类型：

- 单链表（单向）
- 双向链表（双向）

链表的基本操作：

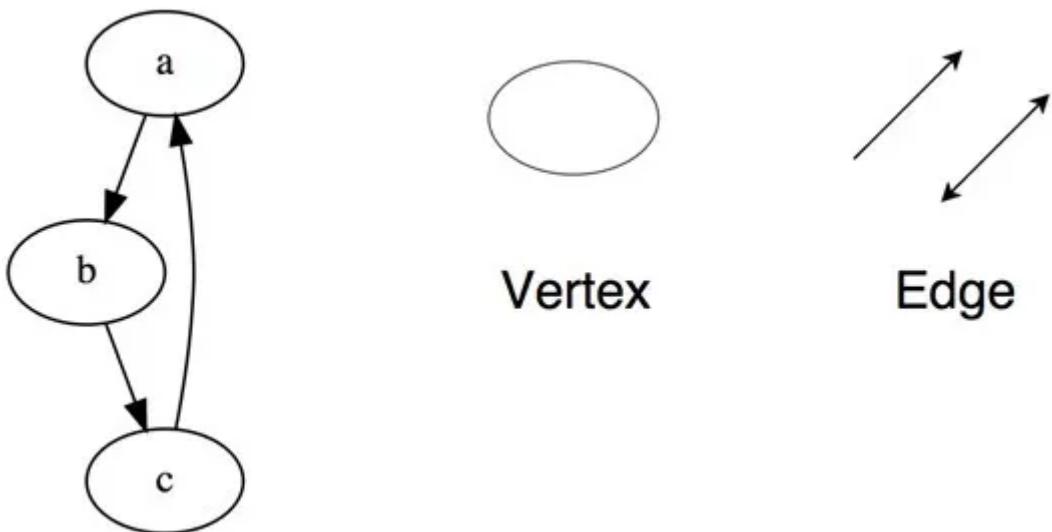
- InsertAtEnd - 在链表的末尾插入指定元素
- InsertAtHead - 在链接列表的开头/头部插入指定元素
- Delete - 从链接列表中删除指定元素
- DeleteAtHead - 删除链接列表的第一个元素
- Search - 从链表中返回指定元素
- isEmpty - 如果链表为空，则返回true

面试中关于链表的常见问题

- 反转链表
- 检测链表中的循环
- 返回链表倒数第N个节点
- 删除链表中的重复项

5. 图

图是一组以网络形式相互连接的节点。节点也称为顶点。一对节点 (x, y) 称为边 (edge)，表示顶点 x 连接到顶点 y 。边可以包含权重/成本，显示从顶点 x 到 y 所需的成本。



图的类型

- 无向图
- 有向图

在程序语言中，图可以用两种形式表示：

- 邻接矩阵
- 邻接表

常见图遍历算法

- 广度优先搜索
- 深度优先搜索

面试中关于图的常见问题

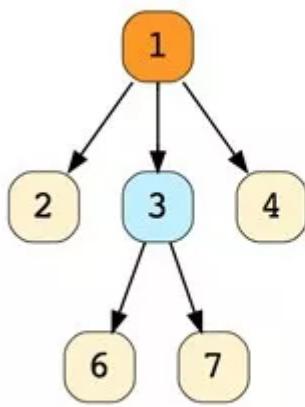
- 实现广度和深度优先搜索
- 检查图是否为树
- 计算图的边数
- 找到两个顶点之间的最短路径

6. 树

树形结构是一种层级式的数据结构，由顶点（节点）和连接它们的边组成。树类似于图，但区分树和图的重要特征是树中不存在环路。

树形结构被广泛应用于人工智能和复杂算法，它可以提供解决问题的有效存储机制。

这是一个简单树的示意图，以及树数据结构中使用的基本术语：



Root	
Parent	
Child	
Leaf	
Sibling	

Root - 根节点

Parent - 父节点

Child - 子节点

Leaf - 叶子节点

Sibling - 兄弟节点

以下是树形结构的主要类型：

- N元树
- 平衡树
- 二叉树
- 二叉搜索树
- AVL树
- 红黑树
- 2-3树

其中，二叉树和二叉搜索树是最常用的树。

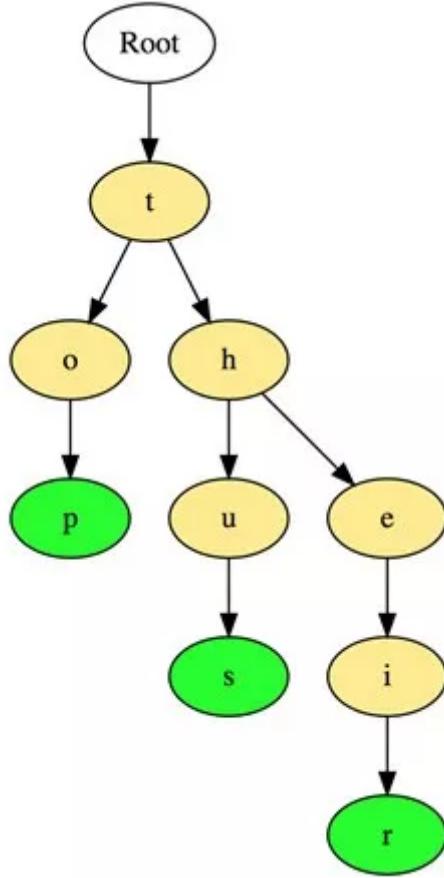
面试中关于树结构的常见问题：

- 求二叉树的高度
- 在二叉搜索树中查找第k个最大值
- 查找与根节点距离k的节点
- 在二叉树中查找给定节点的祖先节点

7. 字典树 (Trie)

字典树，也称为“前缀树”，是一种特殊的树状数据结构，对于解决字符串相关问题非常有效。它能够提供快速检索，主要用于搜索字典中的单词，在搜索引擎中自动提供建议，甚至被用于IP的路由。

以下是在字典树中存储三个单词“top”，“so”和“their”的例子：



这些单词以顶部到底部的方式存储，其中绿色节点“p”，“s”和“r”分别表示“top”，“thus”和“theirs”的底部。

面试中关于字典树的常见问题

- 计算字典树中的总单词数
- 打印存储在字典树中的所有单词
- 使用字典树对数组的元素进行排序
- 使用字典树从字典中形成单词
- 构建T9字典（字典树+ DFS）

8. 哈希表

哈希法（Hashing）是一个用于唯一标识对象并将每个对象存储在一些预先计算的唯一索引（称为“键（key）”）中的过程。因此，对象以键值对的形式存储，这些键值对的集合被称为“字典”。可以使用键搜索每个对象。基于哈希法有很多不同的数据结构，但最常用的数据结构是哈希表。

哈希表通常使用数组实现。

散列数据结构的性能取决于以下三个因素：

- 哈希函数
- 哈希表的大小
- 碰撞处理方法

下图为如何在数组中映射哈希键值对的说明。该数组的索引是通过哈希函数计算的。

3	<key>
	<data>
16	<key>
17	<key>
	<data>

面试中关于哈希结构的常见问题：

- 在数组中查找对称键值对
- 追踪遍历的完整路径
- 查找数组是否是另一个数组的子集
- 检查给定的数组是否不相交

4. 冒泡排序

定义一个布尔变量 `hasChange`，用来标记每轮是否进行了交换。在每轮遍历开始时，将 `hasChange` 设置为 `false`。

若当轮没有发生交换，说明此时数组已经按照升序排列，`hasChange` 依然是为 `false`。此时外层循环直接退出，排序结束。

代码示例

```
import java.util.Arrays;

public class BubbleSort {

    private static void bubbleSort(int[] nums) {
        boolean hasChange = true;
        for (int i = 0, n = nums.length; i < n - 1 && hasChange; ++i) {
            hasChange = false;
            for (int j = 0; j < n - i - 1; ++j) {
                if (nums[j] > nums[j + 1]) {
                    swap(nums, j, j + 1);
                    hasChange = true;
                }
            }
        }
    }
}
```

```

private static void swap(int[] nums, int i, int j) {
    int t = nums[i];
    nums[i] = nums[j];
    nums[j] = t;
}

public static void main(String[] args) {
    int[] nums = {1, 2, 7, 9, 5, 8};
    bubbleSort(nums);
    System.out.println(Arrays.toString(nums));
}
}

```

算法分析

空间复杂度 $O(1)$ 、时间复杂度 $O(n^2)$ 。

分情况讨论：

1. 给定的数组按照顺序已经排好：只需要进行 $n-1$ 次比较，两两交换次数为 0，时间复杂度为 $O(n)$ ，这是最好的情况。
2. 给定的数组按照逆序排列：需要进行 $n*(n-1)/2$ 次比较，时间复杂度为 $O(n^2)$ ，这是最坏的情况。
3. 给定的数组杂乱无章。在这种情况下，平均时间复杂度 $O(n^2)$ 。

因此，时间复杂度是 $O(n^2)$ ，这是一种稳定的排序算法。

稳定是指，两个相等的数，在排序过后，相对位置保持不变。

5. 插入排序

先来看一个问题。一个有序的数组，我们往里面添加一个新的数据后，如何继续保持数据有序呢？很简单，我们只要遍历数组，找到数据应该插入的位置将其插入即可。

这是一个动态排序的过程，即动态地往有序集合中添加数据，我们可以通过这种方法保持集合中的数据一直有序。而对于一组静态数据，我们也可以借鉴上面讲的插入方法，来进行排序，于是就有了插入排序算法。

那么插入排序具体是如何借助上面的思想来实现排序的呢？

首先，我们将数组中的数据分为两个区间，**已排序区间和未排序区间**。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。

与冒泡排序对比：

- 在冒泡排序中，经过每一轮的排序处理后，数组后端的数是排好序的。
- 在插入排序中，经过每一轮的排序处理后，数组前端的数是排好序的。

代码示例

```

import java.util.Arrays;

public class InsertionSort {

    private static void insertionSort(int[] nums) {
        for (int i = 1, j, n = nums.length; i < n; ++i) {

```

```

        int num = nums[i];
        for (j = i - 1; j >= 0 && nums[j] > num; --j) {
            nums[j + 1] = nums[j];
        }
        nums[j + 1] = num;
    }

public static void main(String[] args) {
    int[] nums = {1, 2, 7, 9, 5, 8};
    insertionSort(nums);
    System.out.println(Arrays.toString(nums));
}
}

```

算法分析

空间复杂度 $O(1)$, 时间复杂度 $O(n^2)$ 。

分情况讨论:

1. 给定的数组按照顺序排好序: 只需要进行 $n-1$ 次比较, 两两交换次数为 0, 时间复杂度为 $O(n)$, 这是最好的情况。
2. 给定的数组按照逆序排列: 需要进行 $n*(n-1)/2$ 次比较, 时间复杂度为 $O(n^2)$, 这是最坏的情况。
3. 给定的数组杂乱无章: 在这种情况下, 平均时间复杂度是 $O(n^2)$ 。

因此, 时间复杂度是 $O(n^2)$, 这也是一种稳定的排序算法。

6. 选择排序

选择排序算法的实现思路有点类似插入排序, 也分已排序区间和未排序区间。但是选择排序每次会从未排序区间中找到最小的元素, 将其放到已排序区间的末尾。

代码示例

```

import java.util.Arrays;

public class SelectionSort {

    private static void selectionSort(int[] nums) {
        for (int i = 0, n = nums.length; i < n - 1; ++i) {
            int minIndex = i;
            for (int j = i; j < n; ++j) {
                if (nums[j] < nums[minIndex]) {
                    minIndex = j;
                }
            }
            swap(nums, minIndex, i);
        }
    }

    private static void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
}

```

```
public static void main(String[] args) {
    int[] nums = {1, 2, 7, 9, 5, 8};
    selectionSort(nums);
    System.out.println(Arrays.toString(nums));
}
```

算法分析

空间复杂度 $O(1)$, 时间复杂度 $O(n^2)$ 。

那选择排序是稳定的排序算法吗?

答案是否定的, **选择排序是一种不稳定的排序算法**。选择排序每次都要找剩余未排序元素中的最小值, 并和前面的元素交换位置, 这样破坏了稳定性。

比如 5, 8, 5, 2, 9 这样一组数据, 使用选择排序算法来排序的话, 第一次找到最小元素 2, 与第一个 5 交换位置, 那第一个 5 和中间的 5 顺序就变了, 所以就不稳定了。正是因此, 相对于冒泡排序和插入排序, 选择排序就稍微逊色了。

7. 归并排序

归并排序的核心思想是分治, 把一个复杂问题拆分成若干个子问题来求解。

归并排序的算法思想是: 把数组从中间划分为两个子数组, 一直递归地把子数组划分成更小的数组, 直到子数组里面只有一个元素的时候开始排序。排序的方法就是按照大小顺序合并两个元素。接着依次按照递归的顺序返回, 不断合并排好序的数组, 直到把整个数组排好序。

代码示例

```
import java.util.Arrays;

public class Mergesort {

    private static void merge(int[] nums, int low, int mid, int high, int[] temp) {
        int i = low, j = mid + 1, k = low;
        while (k <= high) {
            if (i > mid) {
                temp[k++] = nums[j++];
            } else if (j > high) {
                temp[k++] = nums[i++];
            } else if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }
        System.arraycopy(temp, low, nums, low, high - low + 1);
    }

    private static void mergesort(int[] nums, int low, int high, int[] temp) {
        if (low >= high) {
            return;
        }
```

```

        int mid = (low + high) >>> 1;
        mergeSort(nums, low, mid, temp);
        mergeSort(nums, mid + 1, high, temp);
        merge(nums, low, mid, high, temp);
    }

    private static void mergeSort(int[] nums) {
        int n = nums.length;
        int[] temp = new int[n];
        mergeSort(nums, 0, n - 1, temp);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 7, 4, 5, 3};
        mergeSort(nums);
        System.out.println(Arrays.toString(nums));
    }
}

```

算法分析

空间复杂度 $O(n)$, 时间复杂度 $O(n \log n)$.

对于规模为 n 的问题, 一共要进行 $\log(n)$ 次的切分, 每一层的合并复杂度都是 $O(n)$, 所以整体时间复杂度为 $O(n \log n)$.

由于合并 n 个元素需要分配一个大小为 n 的额外数组, 所以空间复杂度为 $O(n)$.

这是一种稳定的排序算法。

8. 快速排序

快速排序也采用了分治的思想: 把原始的数组筛选成较小和较大的两个子数组, 然后递归地排序两个子数组。

代码示例

```

import java.util.Arrays;

public class Quicksort {

    private static void quickSort(int[] nums) {
        quickSort(nums, 0, nums.length - 1);
    }

    private static void quickSort(int[] nums, int low, int high) {
        if (low >= high) {
            return;
        }
        int[] p = partition(nums, low, high);
        quickSort(nums, low, p[0] - 1);
        quickSort(nums, p[0] + 1, high);
    }

    private static int[] partition(int[] nums, int low, int high) {
        int less = low - 1, more = high;
        while (low < more) {
            if (nums[more] < nums[low]) {
                swap(nums, low, more);
            }
            more--;
        }
        swap(nums, less + 1, more);
        return new int[]{less + 1, more};
    }
}

```

```

        if (nums[low] < nums[high]) {
            swap(nums, ++less, low++);
        } else if (nums[low] > nums[high]) {
            swap(nums, --more, low);
        } else {
            ++low;
        }
    }
    swap(nums, more, high);
    return new int[] {less + 1, more};
}

private static void swap(int[] nums, int i, int j) {
    int t = nums[i];
    nums[i] = nums[j];
    nums[j] = t;
}

public static void main(String[] args) {
    int[] nums = {1, 2, 7, 4, 5, 3};
    quickSort(nums);
    System.out.println(Arrays.toString(nums));
}
}

```

算法分析

空间复杂度 $O(\log n)$, 时间复杂度 $O(n \log n)$.

对于规模为 n 的问题, 一共要进行 $\log(n)$ 次的切分, 和基准值进行 $n-1$ 次比较, $n-1$ 次比较的时间复杂度是 $O(n)$, 所以快速排序的时间复杂度为 $O(n \log n)$.

但是, 如果每次在选择基准值的时候, 都不幸地选择了子数组里的最大或最小值。即每次把数组分成了两个更小长度的数组, 其中一个长度为 1, 另一个的长度是子数组的长度减 1。这样的算法复杂度变成 $O(n^2)$ 。

和归并排序不同, 快速排序在每次递归的过程中, 只需要开辟 $O(1)$ 的存储空间来完成操作来实现对数组的修改; 而递归次数为 $\log n$, 所以它的整体空间复杂度完全取决于压堆栈的次数。

如何优化快速排序?

前面讲到, 最坏情况下快速排序的时间复杂度是 $O(n^2)$, 实际上, 这种 $O(n^2)$ 时间复杂度出现的主要原因还是因为我们基准值选得不够合理。最理想的基准点是: **被基准点分开的两个子数组中, 数据的数量差不多。**

如果很粗暴地直接选择第一个或者最后一个数据作为基准值, 不考虑数据的特点, 肯定会出现之前讲的那样, 在某些情况下, 排序的最坏情况时间复杂度是 $O(n^2)$.

有两个比较常用的分区算法。

1. 三数取中法

我们从区间的首、尾、中间, 分别取出一个数, 然后对比大小, 取这 3 个数的中间值作为分区点。这样每间隔某个固定的长度, 取数据出来比较, 将中间值作为分区点的分区算法, 肯定要比单纯取某一个数据更好。但是, 如果要排序的数组比较大, 那“三数取中”可能就不够了, 可能要“五数取中”或者“十数取中”。

2. 随机法

随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。这种方法并不能保证每次分区点都选的比较好，但是从概率的角度来看，也不大可能会出现每次分区点都选的很差的情况，所以平均情况下，这样选的分区点是比较好的。时间复杂度退化为最糟糕的 $O(n^2)$ 的情况，出现的可能性不大。

9. 二分查找

二分查找是一种非常高效的查找算法，高效到什么程度呢？我们来分析一下它的时间复杂度。

假设数据大小是 n ，每次查找后数据都会缩小为原来的一半，也就是会除以 2。最坏情况下，直到查找区间被缩小为空，才停止。

被查找区间的大小变化为：

```
n, n/2, n/4, n/8, ..., n/(2^k)
```

可以看出来，这是一个等比数列。其中 $n/(2^k)=1$ 时， k 的值就是总共缩小的次数。而每一次缩小操作只涉及两个数据的大小比较，所以，经过了 k 次区间缩小操作，时间复杂度就是 $O(k)$ 。通过 $n/(2^k)=1$ ，我们可以求得 $k=\log_2 n$ ，所以时间复杂度就是 $O(\log n)$ 。

代码示例

注意容易出错的 3 个地方。

1. 循环退出条件是 `low <= high`，而不是 `low < high`；
2. `mid` 的取值，可以是 `mid = (low + high) / 2`，但是如果 `low` 和 `high` 比较大的话，`low + high` 可能会溢出，所以这里写为 `mid = (low + high) >>> 1`；
3. `low` 和 `high` 的更新分别为 `low = mid + 1`、`high = mid - 1`。

非递归实现：

```
public class BinarySearch {  
  
    private static int search(int[] nums, int low, int high, int val) {  
        while (low <= high) {  
            int mid = (low + high) >>> 1;  
            if (nums[mid] == val) {  
                return mid;  
            } else if (nums[mid] < val) {  
                low = mid + 1;  
            } else {  
                high = mid - 1;  
            }  
        }  
        return -1;  
    }  
  
    /**  
     * 二分查找(非递归)  
     *  
     * @param nums 有序数组  
     * @param val 要查找的值  
     * @return 要查找的值在数组中的索引位置  
     */  
    private static int search(int[] nums, int val) {  
        // 其他实现逻辑  
    }  
}
```

```

        return search(nums, 0, nums.length - 1, val);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 5, 7, 8, 9};

        // 非递归查找
        int r1 = search(nums, 7);
        System.out.println(r1);
    }
}

```

递归实现：

```

public class BinarySearch {

    private static int searchRecursive(int[] nums, int low, int high, int val) {
        while (low <= high) {
            int mid = (low + high) >>> 1;
            if (nums[mid] == val) {
                return mid;
            } else if (nums[mid] < val) {
                return searchRecursive(nums, mid + 1, high, val);
            } else {
                return searchRecursive(nums, low, mid - 1, val);
            }
        }
        return -1;
    }

    /**
     * 二分查找(递归)
     *
     * @param nums 有序数组
     * @param val 要查找的值
     * @return 要查找的值在数组中的索引位置
     */
    private static int searchRecursive(int[] nums, int val) {
        return searchRecursive(nums, 0, nums.length - 1, val);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 5, 7, 8, 9};

        // 递归查找
        int r2 = searchRecursive(nums, 7);
        System.out.println(r2);
    }
}

```

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

10. 二分查找 II

前面讲的二分查找算法，是最为简单的一种，在不存在重复元素的有序数组中，查找值等于给定值的元素。

接下来，我们来看看二分查找算法四种常见的变形问题，分别是：

1. 查找第一个值等于给定值的元素
2. 查找最后一个值等于给定值的元素
3. 查找第一个大于等于给定值的元素
4. 查找最后一个小于等于给定值的元素

1、查找第一个值等于给定值的元素

```
public static int search(int[] nums, int val) {  
    int n = nums.length;  
    int low = 0, high = n - 1;  
    while (low <= high) {  
        int mid = (low + high) >>> 1;  
        if (nums[mid] < val) {  
            low = mid + 1;  
        } else if (nums[mid] > val) {  
            high = mid - 1;  
        } else {  
            // 如果nums[mid]是第一个元素，或者nums[mid-1]不等于val  
            // 说明nums[mid]就是第一个值为给定值的元素  
            if (mid == 0 || nums[mid - 1] != val) {  
                return mid;  
            }  
            high = mid - 1;  
        }  
    }  
    return -1;  
}
```

2、查找最后一个值等于给定值的元素

```
public static int search(int[] nums, int val) {  
    int n = nums.length;  
    int low = 0, high = n - 1;  
    while (low <= high) {  
        int mid = (low + high) >>> 1;  
        if (nums[mid] < val) {
```

```

        low = mid + 1;
    } else if (nums[mid] > val) {
        high = mid - 1;
    } else {
        // 如果nums[mid]是最后一个元素，或者nums[mid+1]不等于val
        // 说明nums[mid]就是最后一个值为给定值的元素
        if (mid == n - 1 || nums[mid + 1] != val) {
            return mid;
        }
        low = mid + 1;
    }
}
return -1;
}

```

3、查找第一个大于等于给定值的元素

```

public static int search(int[] nums, int val) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] < val) {
            low = mid + 1;
        } else {
            // 如果nums[mid]是第一个元素，或者nums[mid-1]小于val
            // 说明nums[mid]就是第一个大于等于给定值的元素
            if (mid == 0 || nums[mid - 1] < val) {
                return mid;
            }
            high = mid - 1;
        }
    }
    return -1;
}

```

4、查找最后一个小于等于给定值的元素

```

public static int search(int[] nums, int val) {
    int n = nums.length;
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        if (nums[mid] > val) {
            high = mid - 1;
        } else {
            // 如果nums[mid]是最后一个元素，或者nums[mid+1]大于val
            // 说明nums[mid]就是最后一个小于等于给定值的元素
            if (mid == n - 1 || nums[mid + 1] > val) {
                return mid;
            }
            low = mid + 1;
        }
    }
}

```

```
    return -1;
}
```

11. 删除排序数组中的重复项

题目描述

给定一个排序数组，你需要在**原地**删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在**原地修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 `nums = [1,1,2]`,

函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 `nums = [0,0,1,1,1,2,2,3,3,4]`,

函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int cnt = 0, n = nums.length;
        for (int i = 1; i < n; ++i) {
            if (nums[i] == nums[i - 1]) ++cnt;
            else nums[i - cnt] = nums[i];
        }
        return n - cnt;
    }
}
```

12. 删除排序数组中的重复项 II

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现两次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定 `nums = [1,1,1,2,2,3]`,

函数应返回新长度 `length = 5`，并且原数组的前五个元素被修改为 `1, 1, 2, 2, 3`。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 `nums = [0,0,1,1,1,1,2,3,3]`,

函数应返回新长度 `length = 7`，并且原数组的前五个元素被修改为 `0, 0, 1, 1, 2, 3, 3`。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

解法

从数组下标 1 开始遍历数组。

用计数器 `cnt` 记录当前数字重复出现的次数，`cnt` 的最小计数为 0；用 `cur` 记录新数组下个待覆盖的元素位置。

遍历时，若当前元素 `nums[i]` 与上个元素 `nums[i-1]` 相同，则计数器 +1，否则计数器重置为 0。如果计数器小于 2，说明当前元素 `nums[i]` 可以添加到新数组中，即：`nums[cur] = nums[i]`，同时 `cur++`。

遍历结果，返回 `cur` 值即可。

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int cnt = 0, cur = 1;
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] == nums[i - 1]) ++cnt;
            else cnt = 0;
            if (cnt < 2) nums[cur++] = nums[i];
        }
        return cur;
    }
}
```

13. 移除元素

题目描述

给你一个数组 $nums$ 和一个值 val ，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 $O(1)$ 额外空间并 原地 修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

给定 $nums = [3, 2, 2, 3]$, $val = 3$,

函数应该返回新的长度 2 ，并且 $nums$ 中的前两个元素均为 2 。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 $nums = [0, 1, 2, 2, 3, 0, 4, 2]$, $val = 2$,

函数应该返回新的长度 5 ，并且 $nums$ 中的前五个元素为 $0, 1, 3, 0, 4$ 。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```

// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}

```

解法

```

class Solution {
    public int removeElement(int[] nums, int val) {
        int cnt = 0, n = nums.length;
        for (int i = 0; i < n; ++i) {
            if (nums[i] == val) {
                ++cnt;
            } else {
                nums[i - cnt] = nums[i];
            }
        }
        return n - cnt;
    }
}

```

14. 移动零

题目描述

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

`[0,1,0,3,12]`

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

解法

```

class Solution {
    public void moveZeroes(int[] nums) {
        int n;
        if (nums == null || (n = nums.length) < 1) {
            return;
        }
        int zeroCount = 0;
        for (int i = 0; i < n; ++i) {
            if (nums[i] == 0) {

```

```

        ++zeroCount;
    } else {
        nums[i - zeroCount] = nums[i];
    }
}
while (zeroCount > 0) {
    nums[n - zeroCount--] = 0;
}
}
}
}

```

15. 数组中重复的数字

题目描述

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0 ~ n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

```

输入:
[2, 3, 1, 0, 2, 5, 3]
输出: 2 或 3

```

限制：

```
2 <= n <= 100000
```

解法

0 ~ n-1 范围内的数，分别还原到对应的位置上，如：数字 2 交换到下标为 2 的位置。

若交换过程中发现重复，则直接返回。

```

class Solution {
    public int findRepeatNumber(int[] nums) {
        for (int i = 0, n = nums.length; i < n; ++i) {
            while (nums[i] != i) {
                if (nums[i] == nums[nums[i]]) return nums[i];
                swap(nums, i, nums[i]);
            }
        }
        return -1;
    }

    private void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
}

```

16. 旋转数组

题目描述

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例 1：

```
[1,2,3,4,5,6,7]
```

示例 2：

```
[-1,-100,3,99]
```

说明：

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 要求使用空间复杂度为 $O(1)$ 的 **原地** 算法。

解法

若 $k=3$ ， $\text{nums}=[1,2,3,4,5,6,7]$ 。

先将 nums 整体翻转： $[1,2,3,4,5,6,7] \rightarrow [7,6,5,4,3,2,1]$

再翻转 $0 \sim k-1$ 范围内的元素： $[7,6,5,4,3,2,1] \rightarrow [5,6,7,4,3,2,1]$

最后翻转 $k \sim n-1$ 范围内的元素，即可得到最终结果： $[5,6,7,4,3,2,1] \rightarrow [5,6,7,1,2,3,4]$

```
class Solution {
    public void rotate(int[] nums, int k) {
        if (nums == null) {
            return;
        }
        int n = nums.length;
        k %= n;
        if (n < 2 || k == 0) {
            return;
        }

        rotate(nums, 0, n - 1);
        rotate(nums, 0, k - 1);
        rotate(nums, k, n - 1);
    }

    private void rotate(int[] nums, int i, int j) {
        while (i < j) {
            int t = nums[i];
            nums[i] = nums[j];
            nums[j] = t;
            ++i;
            --j;
        }
    }
}
```

17. 螺旋矩阵

题目描述

给定一个包含 $m \times n$ 个元素的矩阵 (m 行, n 列) , 请按照顺时针螺旋顺序, 返回矩阵中的所有元素。

示例 1:

```
输入:  
[  
 [ 1, 2, 3 ],  
 [ 4, 5, 6 ],  
 [ 7, 8, 9 ]  
]  
输出: [1,2,3,6,9,8,7,4,5]
```

示例 2:

```
输入:  
[  
 [1, 2, 3, 4],  
 [5, 6, 7, 8],  
 [9,10,11,12]  
]  
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

提示:

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].length$
- $1 \leq m, n \leq 10$
- $-100 \leq \text{matrix}[i][j] \leq 100$

解法

从外往里一圈一圈遍历并存储矩阵元素即可。

```
class Solution {  
    private List<Integer> res;  
  
    public List<Integer> spiralOrder(int[][] matrix) {  
        int m = matrix.length, n = matrix[0].length;  
        res = new ArrayList<>();  
        int i1 = 0, i2 = m - 1;  
        int j1 = 0, j2 = n - 1;  
        while (i1 <= i2 && j1 <= j2) {  
            add(matrix, i1++, j1++, i2--, j2--);  
        }  
        return res;  
    }  
  
    private void add(int[][] matrix, int i1, int j1, int i2, int j2) {  
        if (i1 == i2) {  
            for (int j = j1; j <= j2; ++j) {  
                res.add(matrix[i1][j]);  
            }  
        }  
    }  
}
```

```
        }
        if (j1 == j2) {
            for (int i = i1; i <= i2; ++i) {
                res.add(matrix[i][j1]);
            }
            return;
        }
        for (int j = j1; j < j2; ++j) {
            res.add(matrix[i1][j]);
        }
        for (int i = i1; i < i2; ++i) {
            res.add(matrix[i][j2]);
        }
        for (int j = j2; j > j1; --j) {
            res.add(matrix[i2][j]);
        }
        for (int i = i2; i > i1; --i) {
            res.add(matrix[i][j1]);
        }
    }
}
```

18. 两数之和

题目描述

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

解法

用哈希表（字典）存放数组值以及对应的下标。

遍历数组，当发现 `target - nums[i]` 在哈希表中，说明找到了目标值。

```

class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0, n = nums.length; i < n; ++i) {
            int num = target - nums[i];
            if (map.containsKey(num)) {
                return new int[]{map.get(num), i};
            }
            map.put(nums[i], i);
        }
        return null;
    }
}

```

19. 三数之和

给你一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？
请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

给定数组 `nums = [-1, 0, 1, 2, -1, -4]`,

满足要求的三元组集合为：

```
[
    [-1, 0, 1],
    [-1, -1, 2]
]
```

解法

“排序 + 双指针”实现。

```

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n;
        if (nums == null || (n = nums.length) < 3) {
            return Collections.emptyList();
        }
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < n - 2; ++i) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            int p = i + 1, q = n - 1;
            while (p < q) {
                if (p > i + 1 && nums[p] == nums[p - 1]) {
                    ++p;
                }

```

```

        continue;
    }
    if (q < n - 1 && nums[q] == nums[q + 1]) {
        --q;
        continue;
    }
    if (nums[p] + nums[q] + nums[i] < 0) {
        ++p;
    } else if (nums[p] + nums[q] + nums[i] > 0) {
        --q;
    } else {
        res.add(Arrays.asList(nums[p], nums[q], nums[i]));
        ++p;
        --q;
    }
}
}
return res;
}
}

```

20. 四数之和

题目描述

给定一个包含 n 个整数的数组 `nums` 和一个目标值 `target`，判断 `nums` 中是否存在四个元素 a, b, c 和 d ，使得 $a + b + c + d$ 的值与 `target` 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

示例：

给定数组 `nums = [1, 0, -1, 0, -2, 2]`，和 `target = 0`。

满足要求的四元组集合为：

```
[
  [-1, 0, 0, 1],
  [-2, -1, 1, 2],
  [-2, 0, 0, 2]
]
```

解法

“排序 + 双指针”实现。

```

class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        int n;
        if (nums == null || (n = (nums.length)) < 4) {
            return Collections.emptyList();
        }
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < n - 3; ++i) {

```

```

        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }
        for (int j = i + 1; j < n - 2; ++j) {
            if (j > i + 1 && nums[j] == nums[j - 1]) {
                continue;
            }
            int p = j + 1, q = n - 1;
            while (p < q) {
                if (p > j + 1 && nums[p] == nums[p - 1]) {
                    ++p;
                    continue;
                }
                if (q < n - 1 && nums[q] == nums[q + 1]) {
                    --q;
                    continue;
                }
                int t = nums[i] + nums[j] + nums[p] + nums[q];
                if (t == target) {
                    res.add(Arrays.asList(nums[i], nums[j], nums[p],
                        nums[q]));
                    ++p;
                    --q;
                } else if (t < target) {
                    ++p;
                } else {
                    --q;
                }
            }
        }
    }
    return res;
}
}

```

21. 较小的三数之和

题目描述

给定一个长度为 n 的整数数组和一个目标值 $target$, 寻找能够使条件 $nums[i] + nums[j] + nums[k] < target$ 成立的三元组 i, j, k 个数 ($0 \leq i < j < k < n$)。

示例:

`[-2,0,1,3]`

进阶: 是否能在 $O(n^2)$ 的时间复杂度内解决?

解法

双指针解决。

```

class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        Arrays.sort(nums);
    }
}

```

```

        int n = nums.length;
        int count = 0;
        for (int i = 0; i < n - 2; ++i) {
            count += threeSumSmaller(nums, i + 1, n - 1, target - nums[i]);
        }
        return count;
    }

    private int threeSumSmaller(int[] nums, int start, int end, int target) {
        int count = 0;
        while (start < end) {
            if (nums[start] + nums[end] < target) {
                count += (end - start);
                ++start;
            } else {
                --end;
            }
        }
        return count;
    }
}

```

22. 最接近的三数之和

题目描述

给定一个包括 n 个整数的数组 `nums` 和一个目标值 `target`。找出 `nums` 中的三个整数，使得它们的和与 `target` 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

例如，给定数组 `nums = [-1, 2, 1, -4]`，和 `target = 1`.

与 `target` 最接近的三个数的和为 2 . ($-1 + 2 + 1 = 2$).

解法

双指针解决。

```

class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int res = 0;
        int n = nums.length;
        int diff = Integer.MAX_VALUE;
        for (int i = 0; i < n - 2; ++i) {
            int t = twoSumClosest(nums, i + 1, n - 1, target - nums[i]);
            if (Math.abs(nums[i] + t - target) < diff) {
                res = nums[i] + t;
                diff = Math.abs(nums[i] + t - target);
            }
        }
        return res;
    }

    private int twoSumClosest(int[] nums, int start, int end, int target) {
        int res = 0;

```

```
int diff = Integer.MAX_VALUE;
while (start < end) {
    int val = nums[start] + nums[end];
    if (val == target) {
        return val;
    }
    if (Math.abs(val - target) < diff) {
        res = val;
        diff = Math.abs(val - target);
    }
    if (val < target) {
        ++start;
    } else {
        --end;
    }
}
return res;
}
```

23. 合并两个有序数组

题目描述

给你两个有序整数数组 $nums1$ 和 $nums2$ ，请你将 $nums2$ 合并到 $nums1$ 中，使 $num1$ 成为一个有序数组。

说明:

- 初始化 $nums1$ 和 $nums2$ 的元素数量分别为 m 和 n 。
- 你可以假设 $nums1$ 有足够的空间（空间大小大于或等于 $m + n$ ）来保存 $nums2$ 中的元素。

示例:

```
输入:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],           n = 3

输出: [1,2,2,3,5,6]
```

解法

双指针解决。

```

class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1, j = n - 1;
        int k = m + n - 1;
        while (j >= 0) {
            if (i >= 0 && nums1[i] >= nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
    }
}

```

24. 寻找旋转排序数组中的最小值

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

```

输入: [3,4,5,1,2]
输出: 1

```

示例 2:

```

输入: [4,5,6,7,0,1,2]
输出: 0

```

解法

二分查找。

若 `nums[m] > nums[r]`，说明最小值在 `m` 的右边，否则说明最小值在 `m` 的左边（包括 `m`）。

```

class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int m = (l + r) >>> 1;
            if (nums[m] > nums[r]) {
                l = m + 1;
            } else {
                r = m;
            }
        }
    }
}

```

```
    return nums[1];
}
}
```

25. 寻找旋转排序数组中的最小值 II

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1：

```
输入: [1,3,5]
输出: 1
```

示例 2：

```
输入: [2,2,2,0,1]
输出: 0
```

说明：

- 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

```
class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int m = (l + r) >>> 1;
            if (nums[m] > nums[r]) {
                l = m + 1;
            } else if (nums[m] < nums[r]) {
                r = m;
            } else {
                --r;
            }
        }
        return nums[l];
    }
}
```

26. 除自身以外数组的乘积

题目描述

给你一个长度为 n 的整数数组 `nums`，其中 $n > 1$ ，返回输出数组 `output`，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

示例:

```
[1,2,3,4]
```

提示：题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明：请不要使用除法，且在 $O(n)$ 时间复杂度内完成此题。

进阶：

你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

解法

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] output = new int[n];
        for (int i = 0, left = 1; i < n; ++i) {
            output[i] = left;
            left *= nums[i];
        }
        for (int i = n - 1, right = 1; i >= 0; --i) {
            output[i] *= right;
            right *= nums[i];
        }
        return output;
    }
}
```

27. 无重复字符的最长子串

题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:

```
"abc"，所以其
```

示例 2:

```
"b"
```

示例 3:

```
"wke"
```

解法

- 定义一个哈希表存放字符及其出现的位置；
- 定义 i, j 分别表示不重复子串的开始位置和结束位置；
- j 向后遍历，若遇到与 [i, j] 区间内字符相同的元素，更新 i 的值，此时 [i, j] 区间内不存在重复字符，计算 res 的最大值。

```
class Solution {  
    public int lengthOfLongestSubstring(String s) {  
        int res = 0;  
        Map<Character, Integer> chars = new HashMap<>();  
        for (int i = 0, j = 0; j < s.length(); ++j) {  
            char c = s.charAt(j);  
            if (chars.containsKey(c)) {  
                // chars.get(c)+1 可能比 i 还小，通过 max 函数来锁住左边界  
                // e.g. 在"tmmzuxt"这个字符串中，遍历到最后一步时，最后一个字符't'和第一个  
                // 字符't'是相等的。如果没有 max 函数，i 就会回到第一个't'的索引0处的下一个位置  
                i = Math.max(i, chars.get(c) + 1);  
            }  
            chars.put(c, j);  
            res = Math.max(res, j - i + 1);  
        }  
        return res;  
    }  
}
```

28. 反转字符串中的元音字母

题目描述

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

示例 1:

```
输入: "hello"  
输出: "holle"
```

示例 2:

```
输入: "leetcode"  
输出: "leotcede"
```

说明:

元音字母不包含字母"y"。

解法

将字符串转为字符数组（或列表），定义双指针 p、q，分别指向数组（列表）头部和尾部，当 p、q 指向的字符均为元音字母时，进行交换。

依次遍历，当 $p \geq q$ 时，遍历结束。将字符数组（列表）转为字符串返回即可。

```
class Solution {
    public String reverseVowels(String s) {
        if (s == null) {
            return s;
        }
        char[] chars = s.toCharArray();
        int p = 0, q = chars.length - 1;
        while (p < q) {
            if (!isVowel(chars[p])) {
                ++p;
                continue;
            }
            if (!isVowel(chars[q])) {
                --q;
                continue;
            }
            swap(chars, p++, q--);
        }
        return String.valueOf(chars);
    }

    private void swap(char[] chars, int i, int j) {
        char t = chars[i];
        chars[i] = chars[j];
        chars[j] = t;
    }

    private boolean isVowel(char c) {
        switch(c) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':
                return true;
            default:
                return false;
        }
    }
}
```

29. 字符串转换整数

题目描述

请你来实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

微信搜索公众号：Java专栏，获取最新面试手册

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 $[-2^{31}, 2^{31} - 1]$ 。如果数值超过这个范围，请返回 `INT_MAX` ($2^{31} - 1$) 或 `INT_MIN` (-2^{31})。

示例 1：

```
输入: "42"
输出: 42
```

示例 2：

```
输入: " -42"
输出: -42
解释: 第一个非空白字符为 '-'，它是一个负号。
      我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。
```

示例 3：

```
输入: "4193 with words"
输出: 4193
解释: 转换截止于数字 '3'，因为它的下一个字符不为数字。
```

示例 4：

```
输入: "words and 987"
输出: 0
解释: 第一个非空字符是 'w'，但它不是数字或正、负号。
      因此无法执行有效的转换。
```

示例 5：

```
输入: "-91283472332"
输出: -2147483648
解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
      因此返回 INT_MIN (-2^{31}) 。
```

解法

遍历字符串，注意做溢出处理。

```
class Solution {
    public int myAtoi(String s) {
```

```

    if (s == null) return 0;
    int n = s.length();
    if (n == 0) return 0;
    int i = 0;
    while (s.charAt(i) == ' ') {
        // 仅包含空格
        if (++i == n) return 0;
    }
    int sign = 1;
    if (s.charAt(i) == '-') sign = -1;
    if (s.charAt(i) == '-' || s.charAt(i) == '+') ++i;
    int res = 0, flag = Integer.MAX_VALUE / 10;
    for (; i < n; ++i) {
        // 非数字, 跳出循环体
        if (s.charAt(i) < '0' || s.charAt(i) > '9') break;
        // 溢出判断
        if (res > flag || (res == flag && s.charAt(i) > '7')) return sign >
0 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
        res = res * 10 + (s.charAt(i) - '0');
    }
    return sign * res;
}
}

```

30. 贼金信

题目描述

给定一个赎金信 (ransom) 字符串和一个杂志(magazine)字符串，判断第一个字符串ransom能不能由第二个字符串magazines里面的字符构成。如果可以构成，返回 true；否则返回 false。

(题目说明：为了不暴露赎金信字迹，要从杂志上搜索各个需要的字母，组成单词来表达意思。)

注意：

你可以假设两个字符串均只含有小写字母。

```

canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true

```

解法

用一个数组或字典 chars 存放 magazine 中每个字母出现的次数。遍历 ransomNote 中每个字母，判断 chars 是否包含即可。

```

class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        int[] chars = new int[26];
        for (int i = 0, n = magazine.length(); i < n; ++i) {
            int idx = magazine.charAt(i) - 'a';
            ++chars[idx];
        }
        for (int i = 0, n = ransomNote.length(); i < n; ++i) {
            int idx = ransomNote.charAt(i) - 'a';
            if (chars[idx] == 0) return false;
        }
    }
}

```

```
    --chars[idx];
}
return true;
}
}
```

31. 两数相加

题目描述

给出两个**非空**的链表用来表示两个非负的整数。其中，它们各自的位数是按照**逆序**的方式存储的，并且它们的每个节点只能存储**一位**数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例：

```
输入: (2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 0 -> 8
原因: 342 + 465 = 807
```

```
/**
 * Definition for singly-linked list.
 */
public class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry = 0;
        ListNode dummy = new ListNode(-1);
        ListNode cur = dummy;
        while (l1 != null || l2 != null || carry != 0) {
            int t = (l1 == null ? 0 : l1.val) + (l2 == null ? 0 : l2.val) +
carry;
            carry = t / 10;
            cur.next = new ListNode(t % 10);
            cur = cur.next;
            l1 = l1 == null ? null : l1.next;
            l2 = l2 == null ? null : l2.next;
        }
        return dummy.next;
    }
}
```

32. 两数相加 II

题目描述

给定两个**非空链表**来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储单个数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶：

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例：

```
输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 8 -> 0 -> 7
```

解法

利用栈将数字逆序。

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        Deque<Integer> s1 = new ArrayDeque<>();
        Deque<Integer> s2 = new ArrayDeque<>();
        for (; l1 != null; l1 = l1.next) {
            s1.push(l1.val);
        }
        for (; l2 != null; l2 = l2.next) {
            s2.push(l2.val);
        }
        int carry = 0;
        ListNode dummy = new ListNode(-1);
        while (!s1.isEmpty() || !s2.isEmpty() || carry != 0) {
            carry += (s1.isEmpty() ? 0 : s1.pop()) + (s2.isEmpty() ? 0 :
s2.pop());
            // 创建结点，利用头插法将结点插入链表
            ListNode node = new ListNode(carry % 10);
            node.next = dummy.next;
            dummy.next = node;
            carry /= 10;
        }
        return dummy.next;
    }
}
```

33. 从尾到头打印链表

题目描述

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入: head = [1,3,2]
输出: [2,3,1]
```

限制：

- $0 \leq$ 链表长度 ≤ 10000

解法

栈实现。或者其它方式，见题解。

- 栈实现：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] reversePrint(ListNode head) {
        Stack<Integer> s = new Stack<>();
        while (head != null) {
            s.push(head.val);
            head = head.next;
        }
        int[] res = new int[s.size()];
        int i = 0;
        while (!s.isEmpty()) {
            res[i++] = s.pop();
        }
        return res;
    }
}
```

- 先计算链表长度 n ，然后创建一个长度为 n 的结果数组。最后遍历链表，依次将节点值存放在数组上（从后往前）。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
```

```

*     ListNode next;
*     ListNode(int x) { val = x; }
* }
*/
class Solution {
    public int[] reversePrint(ListNode head) {
        if (head == null) return new int[]{};
        // 计算链表长度n
        int n = 0;
        ListNode cur = head;
        while (cur != null) {
            ++n;
            cur = cur.next;
        }
        int[] res = new int[n];
        cur = head;
        while (cur != null) {
            res[--n] = cur.val;
            cur = cur.next;
        }
        return res;
    }
}

```

34. 删除链表的节点

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

示例 1:

```

输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点, 那么在调用了你的函数之后, 该链表应变为 4 -> 1 -> 9.

```

示例 2:

```

输入: head = [4,5,1,9], val = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点, 那么在调用了你的函数之后, 该链表应变为 4 -> 5 -> 9.

```

说明:

- 题目保证链表中节点的值互不相同
- 若使用 C 或 C++ 语言，你不需要 `free` 或 `delete` 被删除的节点

解法

定义一个虚拟头节点 `dummy` 指向 `head`，`pre` 指针初始指向 `dummy`。

循环遍历链表，`pre` 往后移动。当指针 `pre.next` 指向的节点的值等于 `val` 时退出循环，将 `pre.next` 指向 `pre.next.next`，然后返回 `dummy.next`。

```
/**
```

```

* Definition for singly-linked list.
* public class ListNode {
*     int val;
*     ListNode next;
*     ListNode(int x) { val = x; }
* }
*/
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        while (pre.next != null && pre.next.val != val) {
            pre = pre.next;
        }
        pre.next = pre.next == null ? null : pre.next.next;
        return dummy.next;
    }
}

```

35. 删除排序链表中的重复元素

题目描述

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1：

```

输入: 1->1->2
输出: 1->2

```

示例 2：

```

输入: 1->1->2->3->3
输出: 1->2->3

```

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
*     int val;
*     ListNode next;
*     ListNode() {}
*     ListNode(int val) { this.val = val; }
*     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
* }
*/
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode cur = head;
        while (cur != null && cur.next != null) {
            if (cur.val == cur.next.val) {
                cur.next = cur.next.next;
            }
        }
    }
}

```

```
        } else {
            cur = cur.next;
        }
    }
    return head;
}
}
```

36. 删除排序链表中的重复元素 II

题目描述

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现的数字。

示例 1:

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

示例 2:

```
输入: 1->1->1->2->3
输出: 2->3
```

解法

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(-1, head);
        ListNode cur = dummy;
        while (cur.next != null && cur.next.next != null) {
            if (cur.next.val == cur.next.next.val) {
                int val = cur.next.val;
                while (cur.next != null && cur.next.val == val) {
                    cur.next = cur.next.next;
                }
            } else {
                cur = cur.next;
            }
        }
        return dummy.next;
    }
}
```

37. 移除链表元素

题目描述

删除链表中等于给定值 ***val*** 的所有节点。

示例:

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

解法

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(-1, head);
        ListNode pre = dummy;
        while (pre != null && pre.next != null) {
            if (pre.next.val != val) pre = pre.next;
            else pre.next = pre.next.next;
        }
        return dummy.next;
    }
}
```

38. 两两交换链表中的节点

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例:

```
1->2->3->4
```

解法

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(0, head);
        ListNode pre = dummy, cur = head;
        while (cur != null && cur.next != null) {
            ListNode t = cur.next;
            cur.next = t.next;
            t.next = cur;
            pre.next = t;
            pre = cur;
            cur = pre.next;
        }
        return dummy.next;
    }
}

```

39. 排序链表

题目描述

在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1：

```

输入: 4->2->1->3
输出: 1->2->3->4

```

示例 2：

```

输入: -1->5->3->4->0
输出: -1->0->3->4->5

```

解法

先用快慢指针找到链表中点，然后分成左右两个链表，递归排序左右链表。最后合并两个排序的链表即可。

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 * }
 */

```

```

*     ListNode() {}
*     ListNode(int val) { this.val = val; }
*     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
* }
*/
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode t = slow.next;
        slow.next = null;
        ListNode l1 = sortList(head);
        ListNode l2 = sortList(t);
        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        cur.next = l1 == null ? l2 : l1;
        return dummy.next;
    }
}

```

40. 反转链表

题目描述

反转一个单链表。

示例:

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

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

解法

定义指针 `p`、`q` 分别指向头节点和下一个节点，`pre` 指向头节点的前一个节点。

遍历链表，改变指针 `p` 指向的节点的指向，将其指向 `pre` 指针指向的节点，即 `p.next = pre`。然后 `pre` 指针指向 `p`，`p`、`q` 指针往前走。

当遍历结束后，返回 `pre` 指针即可。

迭代版本

```
/*
 * Definition for singly-linked list.
 */
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null, p = head;
        while (p != null) {
            ListNode q = p.next;
            p.next = pre;
            pre = p;
            p = q;
        }
        return pre;
    }
}
```

递归版本

```
/*
 * Definition for singly-linked list.
 */
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode res = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return res;
    }
}
```

41. 二叉树的前序遍历

题目描述

给定一个二叉树，返回它的 前序遍历。

示例：

输入： [1,null,2,3]

```
1
 \
 2
 /
3
```

输出： [1,2,3]

进阶：递归算法很简单，你可以通过迭代算法完成吗？

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

1. 定义一个栈，先将根节点压入栈
2. 若栈不为空，每次从栈中弹出一个节点
3. 处理该节点
4. 先把节点右孩子压入栈，接着把节点左孩子压入栈（如果有孩子节点）
5. 重复 2-4
6. 返回结果

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> preorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        preorder(root);
    }

    private void preorder(TreeNode root) {
        if (root == null) return;
        res.add(root.val);
        preorder(root.left);
        preorder(root.right);
    }
}
```

```

        return res;
    }

    private void preorder(TreeNode root) {
        if (root != null) {
            res.add(root.val);
            preorder(root.left);
            preorder(root.right);
        }
    }
}

```

非递归

```


/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        if (root == null) {
            return Collections.emptyList();
        }
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> s = new ArrayDeque<>();
        s.push(root);
        while (!s.isEmpty()) {
            TreeNode node = s.pop();
            res.add(node.val);
            if (node.right != null) {
                s.push(node.right);
            }
            if (node.left != null) {
                s.push(node.left);
            }
        }
        return res;
    }
}


```

42. 二叉树的后序遍历

题目描述

给定一个二叉树，返回它的 *后序遍历*。

示例:

输入: [1,null,2,3]

```
1
 \
 2
 /
3
```

输出: [3,2,1]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

先序遍历的顺序是：头、左、右，如果我们改变左右孩子的顺序，就能将顺序变成：头、右、左。

我们先不打印头节点，而是存放到另一个收集栈 s2 中，最后遍历结束，输出收集栈元素，即是后序遍历：左、右、头。

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> postorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        postorder(root);
        return res;
    }
}
```

```

private void postorder(TreeNode root) {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        res.add(root.val);
    }
}

```

非递归

```


/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        if (root == null) {
            return Collections.emptyList();
        }
        Deque<TreeNode> s1 = new ArrayDeque<>();
        List<Integer> s2 = new ArrayList<>();
        s1.push(root);
        while (!s1.isEmpty()) {
            TreeNode node = s1.pop();
            s2.add(node.val);
            if (node.left != null) {
                s1.push(node.left);
            }
            if (node.right != null) {
                s1.push(node.right);
            }
        }
        Collections.reverse(s2);
        return s2;
    }
}


```

43. 二叉树的中序遍历

题目描述

给定一个二叉树，返回它的中序遍历。

示例:

输入: [1,null,2,3]

```
1
 \
 2
 /
3
```

输出: [1,3,2]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

解法

递归遍历或利用栈实现非递归遍历。

非递归的思路如下：

1. 定义一个栈
2. 将树的左节点依次入栈
3. 左节点为空时，弹出栈顶元素并处理
4. 重复 2-3 的操作

递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    private List<Integer> res;

    public List<Integer> inorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        inorder(root);
        return res;
    }
}
```

```
private void inorder(TreeNode root) {  
    if (root != null) {  
        inorder(root.left);  
        res.add(root.val);  
        inorder(root.right);  
    }  
}
```

非递归

```
/**  
 * Definition for a binary tree node.  
 * public class TreeNode {  
 *     int val;  
 *     TreeNode left;  
 *     TreeNode right;  
 *     TreeNode() {}  
 *     TreeNode(int val) { this.val = val; }  
 *     TreeNode(int val, TreeNode left, TreeNode right) {  
 *         this.val = val;  
 *         this.left = left;  
 *         this.right = right;  
 *     }  
 * }  
 */  
class Solution {  
    public List<Integer> inorderTraversal(TreeNode root) {  
        if (root == null) {  
            return Collections.emptyList();  
        }  
        List<Integer> res = new ArrayList<>();  
        Deque<TreeNode> s = new ArrayDeque<>();  
        while (root != null || !s.isEmpty()) {  
            if (root != null) {  
                s.push(root);  
                root = root.left;  
            } else {  
                root = s.pop();  
                res.add(root.val);  
                root = root.right;  
            }  
        }  
        return res;  
    }  
}
```

44. 最小栈

题目描述

设计一个支持 push, pop, top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) -- 将元素 x 推入栈中。
- pop() -- 删除栈顶的元素。
- top() -- 获取栈顶元素。
- getMin() -- 检索栈中的最小元素。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

解法

```
class MinStack {

    private Deque<Integer> s;
    private Deque<Integer> helper;

    /** initialize your data structure here. */
    public MinStack() {
        s = new ArrayDeque<>();
        helper = new ArrayDeque<>();
    }

    public void push(int x) {
        s.push(x);
        int element = helper.isEmpty() || x < helper.peek() ? x : helper.peek();
        helper.push(element);
    }

    public void pop() {
        s.pop();
        helper.pop();
    }

    public int top() {
        return s.peek();
    }

    public int getMin() {
        return helper.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

```
* MinStack obj = new MinStack();
* obj.push(x);
* obj.pop();
* int param_3 = obj.top();
* int param_4 = obj.getMin();
*/
```

45. 队列的最大值

题目描述

请定义一个队列并实现函数 `max_value` 得到队列里的最大值，要求函数 `max_value`、`push_back` 和 `pop_front` 的均摊时间复杂度都是 $O(1)$ 。

若队列为空，`pop_front` 和 `max_value` 需要返回 -1

示例 1：

```
输入：
["MaxQueue", "push_back", "push_back", "max_value", "pop_front", "max_value"]
[], [1], [2], [], [], []
输出：[null, null, null, 2, 1, 2]
```

示例 2：

```
输入：
["MaxQueue", "pop_front", "max_value"]
[], [], []
输出：[null, -1, -1]
```

限制：

- `1 <= push_back, pop_front, max_value` 的总操作数 ≤ 10000
- `1 <= value <= 10^5`

解法

利用一个辅助队列按单调顺序存储当前队列的最大值。

```
class MaxQueue {
    private Deque<Integer> p;
    private Deque<Integer> q;

    public MaxQueue() {
        p = new ArrayDeque<>();
        q = new ArrayDeque<>();
    }

    public int max_value() {
        return q.isEmpty() ? -1 : q.peekFirst();
    }

    public void push_back(int value) {
        while (!q.isEmpty() && q.peekLast() < value) {
            q.pollLast();
        }
        q.offerLast(value);
    }
}
```

```

        }
        p.offerLast(value);
        q.offerLast(value);
    }

    public int pop_front() {
        if (p.isEmpty()) return -1;
        int res = p.pollFirst();
        if (q.peek() == res) q.pollFirst();
        return res;
    }

}

/**
 * Your MaxQueue object will be instantiated and called as such:
 * MaxQueue obj = new MaxQueue();
 * int param_1 = obj.max_value();
 * obj.push_back(value);
 * int param_3 = obj.pop_front();
 */

```

46. 冒泡排序

冒泡排序是最简单的排序之一了，其大体思想就是通过与相邻元素的比较和交换来把小的数交换到最前面。这个过程类似于水泡向上升一样，因此而得名。举个栗子，对5,3,8,6,4这个无序序列进行冒泡排序。首先从后向前冒泡，4和6比较，把4交换到前面，序列变成5,3,8,4,6。同理4和8交换，变成5,3,4,8,6,3和4无需交换。5和3交换，变成3,5,4,8,6,3.这样一次冒泡就完了，把最小的数3排到最前面了。对剩下的序列依次冒泡就会得到一个有序序列。冒泡排序的时间复杂度为O(n^2)。

实现代码：

```

public class BubbleSort {

    public static void bubbleSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        for(int i=0; i<arr.length-1; i++) {
            for(int j=arr.length-1; j>i; j--) {
                if(arr[j] < arr[j-1]) {
                    swap(arr, j-1, j);
                }
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

```

47. 选择排序

选择排序的思想其实和冒泡排序有点类似，都是在一次排序后把最小的元素放到最前面。但是过程不同，冒泡排序是通过相邻的比较和交换。而选择排序是通过对整体的选择。举个栗子，对5,3,8,6,4这个无序序列进行简单选择排序，首先要选择5以外的最小数来和5交换，也就是选择3和5交换，一次排序后就变成了3,5,8,6,4.对剩下的序列一次进行选择和交换，最终就会得到一个有序序列。其实选择排序可以看成冒泡排序的优化，因为其目的相同，只是选择排序只有在确定了最小数的前提下才进行交换，大大减少了交换的次数。选择排序的时间复杂度为 $O(n^2)$

实现代码：

```
public class SelectSort {  
  
    public static void selectSort(int[] arr) {  
        if(arr == null || arr.length == 0)  
            return ;  
        int minIndex = 0;  
        for(int i=0; i<arr.length-1; i++) { //只需要比较n-1次  
            minIndex = i;  
            for(int j=i+1; j<arr.length; j++) { //从i+1开始比较，因为minIndex默认为i了，i就没必要比了。  
                if(arr[j] < arr[minIndex]) {  
                    minIndex = j;  
                }  
            }  
  
            if(minIndex != i) { //如果minIndex不为i，说明找到了更小的值，交换之。  
                swap(arr, i, minIndex);  
            }  
        }  
  
        }  
  
        public static void swap(int[] arr, int i, int j) {  
            int temp = arr[i];  
            arr[i] = arr[j];  
            arr[j] = temp;  
        }  
    }
```

48. 插入排序

插入排序不是通过交换位置而是通过比较找到合适的位置插入元素来达到排序的目的的。相信大家都有过打扑克牌的经历，特别是牌数较大的。在分牌时可能要整理自己的牌，牌多的时候怎么整理呢？就是拿到一张牌，找到一个合适的位置插入。这个原理其实和插入排序是一样的。举个栗子，对5,3,8,6,4这个无序序列进行简单插入排序，首先假设第一个数的位置时正确的，想一下在拿到第一张牌的时候，没必要整理。然后3要插到5前面，把5后移一位，变成3,5,8,6,4.想一下整理牌的时候应该也是这样吧。然后8不用动，6插在8前面，8后移一位，4插在5前面，从5开始都向后移一位。注意在插入一个数的时候要保证这个数前面的数已经有序。简单插入排序的时间复杂度也是 $O(n^2)$ 。

实现代码：

```
public class InsertSort {
```

```

public static void insertSort(int[] arr) {
    if(arr == null || arr.length == 0)
        return ;

    for(int i=1; i<arr.length; i++) { //假设第一个数位置时正确的；要往后移，必须要假设第一个。

        int j = i;
        int target = arr[i]; //待插入的

        //后移
        while(j > 0 && target < arr[j-1]) {
            arr[j] = arr[j-1];
            j--;
        }

        //插入
        arr[j] = target;
    }
}

```

49. 快速排序

快速排序一听名字就觉得很高端，在实际应用当中快速排序确实也是表现最好的排序算法。快速排序虽然高端，但其实其思想是来自冒泡排序，冒泡排序是通过相邻元素的比较和交换把最小的冒泡到最顶端，而快速排序是比较和交换小数和大数，这样一来不仅把小数冒泡到上面同时也把大数沉到下面。

举个栗子：对5,3,8,6,4这个无序序列进行快速排序，思路是右指针找比基准数小的，左指针找比基准数大的，交换之。

5,3,8,6,4 用5作为比较的基准，最终会把5小的移动到5的左边，比5大的移动到5的右边。

5,3,8,6,4 首先设置i,j两个指针分别指向两端，j指针先扫描（思考一下为什么？）4比5小停止。然后扫描，8比5大停止。交换i,j位置。

5,3,4,6,8 然后j指针再扫描，这时j扫描4时两指针相遇。停止。然后交换4和基准数。

4,3,5,6,8 一次划分后达到了左边比5小，右边比5大的目的。之后对左右子序列递归排序，最终得到有序序列。

上面留下来了一个问题为什么一定要j指针先动呢？首先这也不是绝对的，这取决于基准数的位置，因为在最后两个指针相遇的时候，要交换基准数到相遇的位置。一般选取第一个数作为基准数，那么就是在左边，所以最后相遇的数要和基准数交换，那么相遇的数一定要比基准数小。所以j指针先移动才能先找到比基准数小的数。

快速排序是不稳定的，其时间平均时间复杂度是 $O(n \lg n)$ 。

实现代码：

```

public class Quicksort {
    //一次划分
    public static int partition(int[] arr, int left, int right) {

```

```

        int pivotKey = arr[left];
        int pivotPointer = left;

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right--;
            while(left < right && arr[left] <= pivotKey)
                left++;
            swap(arr, left, right); //把大的交换到右边，把小的交换到左边。
        }
        swap(arr, pivotPointer, left); //最后把pivot交换到中间
        return left;
    }

    public static void quickSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int pivotPos = partition(arr, left, right);
        quickSort(arr, left, pivotPos-1);
        quickSort(arr, pivotPos+1, right);
    }

    public static void sort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        quickSort(arr, 0, arr.length-1);
    }

    public static void swap(int[] arr, int left, int right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
    }
}

```

其实上面的代码还可以再优化，上面代码中基准数已经在pivotKey中保存了，所以不需要每次交换都设置一个temp变量，在交换左右指针的时候只需要先后覆盖就可以了。这样既能减少空间的使用还能降低赋值运算的次数。优化代码如下：

```

public class QuickSort {

    /**
     * 划分
     * @param arr
     * @param left
     * @param right
     * @return
     */
    public static int partition(int[] arr, int left, int right) {
        int pivotKey = arr[left];

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right--;
            arr[left] = arr[right]; //把小的移动到左边
            while(left < right && arr[left] <= pivotKey)

```

```

        left++;
        arr[right] = arr[left]; //把大的移动到右边
    }
    arr[left] = pivotKey; //最后把pivot赋值到中间
    return left;
}

/**
 * 递归划分子序列
 * @param arr
 * @param left
 * @param right
 */
public static void quickSort(int[] arr, int left, int right) {
    if(left >= right)
        return ;
    int pivotPos = partition(arr, left, right);
    quickSort(arr, left, pivotPos-1);
    quickSort(arr, pivotPos+1, right);
}

public static void sort(int[] arr) {
    if(arr == null || arr.length == 0)
        return ;
    quickSort(arr, 0, arr.length-1);
}

}

```

总结快速排序的思想：冒泡+二分+递归分治，慢慢体会。。。

50. 堆排序

堆排序是借助堆来实现的选择排序，思想同简单的选择排序，以下以大顶堆为例。注意：如果想升序排序就使用大顶堆，反之使用小顶堆。原因是堆顶元素需要交换到序列尾部。

首先，实现堆排序需要解决两个问题：

如何由一个无序序列建成一个堆？

如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

第一个问题，可以直接使用线性数组来表示一个堆，由初始的无序序列建成一个堆就需要自底向上从第一个非叶元素开始挨个调整成一个堆。

第二个问题，怎么调整成堆？首先是将堆顶元素和最后一个元素交换。然后比较当前堆顶元素的左右孩子节点，因为除了当前的堆顶元素，左右孩子堆均满足条件，这时需要选择当前堆顶元素与左右孩子节点的较大者（大顶堆）交换，直至叶子节点。我们称这个自堆顶自叶子的调整成为筛选。

从一个无序序列建堆的过程就是一个反复筛选的过程。若将此序列看成是一个完全二叉树，则最后一个非终端节点是 $n/2$ 取底个元素，由此筛选即可。举个栗子：

49,38,65,97,76,13,27,49序列的堆排序建初始堆和调整的过程如下

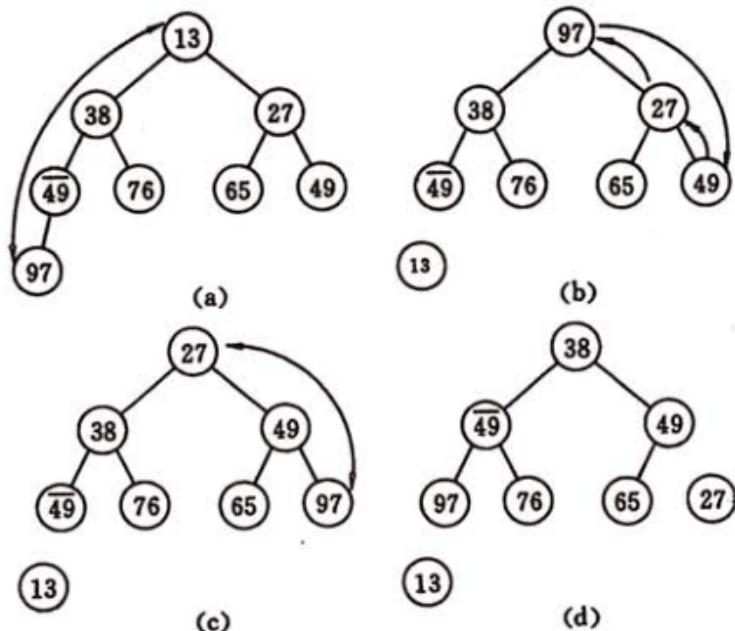


图 10.11 输出堆顶元素并调整建新堆的过程

- (a) 堆; (b) 13 和 97 交换后的情形; (c) 调整后的新堆;
(d) 27 和 97 交换后再进行调整建成的新堆

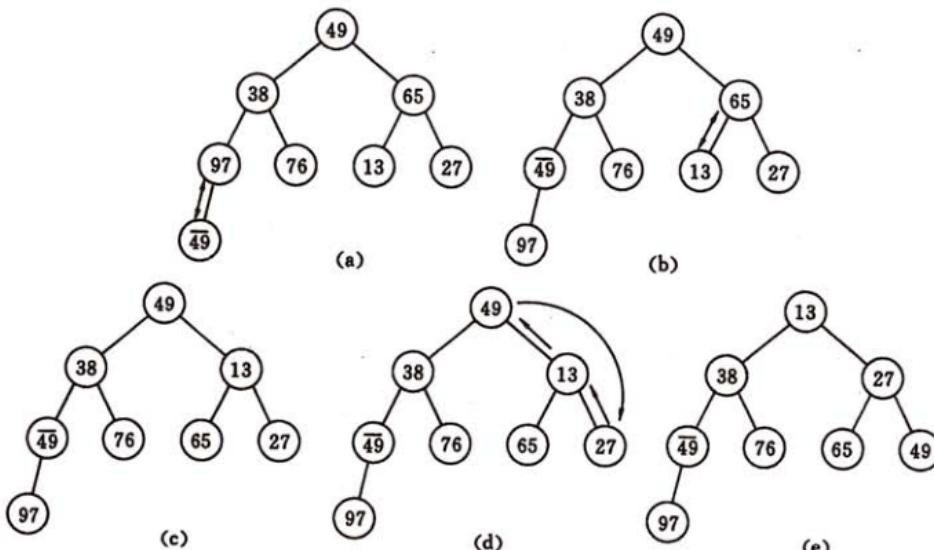


图 10.12 建初始堆过程示例

- (a) 无序序列; (b) 97 被筛选之后的状态; (c) 65 被筛选之后的状态;
(d) 38 被筛选之后的状态; (e) 49 被筛选之后建成的堆

实现代码:

```
public class HeapSort {  
    /**  
     * 堆筛选，除了start之外，start~end均满足大顶堆的定义。  
     * 调整之后start~end称为一个大顶堆。  
     * @param arr 待调整数组  
     * @param start 起始指针  
     * @param end 结束指针  
     */
```

```

/*
public static void heapAdjust(int[] arr, int start, int end) {
    int temp = arr[start];

    for(int i=2*start+1; i<=end; i*=2) {
        //左右孩子的节点分别为2*i+1,2*i+2

        //选择出左右孩子较小的下标
        if(i < end && arr[i] < arr[i+1]) {
            i++;
        }
        if(temp >= arr[i]) {
            break; //已经为大顶堆, =保持稳定性。
        }
        arr[start] = arr[i]; //将子节点上移
        start = i; //下一轮筛选
    }

    arr[start] = temp; //插入正确的位置
}

public static void heapSort(int[] arr) {
    if(arr == null || arr.length == 0)
        return;

    //建立大顶堆
    for(int i=arr.length/2; i>=0; i--) {
        heapAdjust(arr, i, arr.length-1);
    }

    for(int i=arr.length-1; i>=0; i--) {
        swap(arr, 0, i);
        heapAdjust(arr, 0, i-1);
    }
}

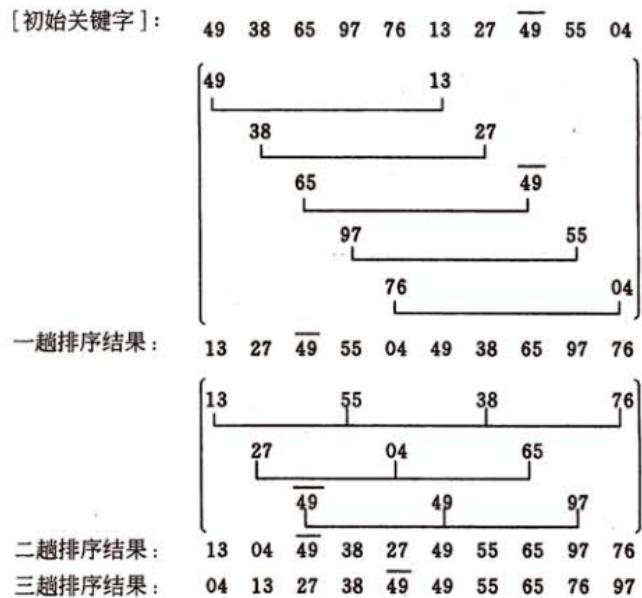
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
}

```

51. 希尔排序

希尔排序是插入排序的一种高效率的实现，也叫缩小增量排序。简单的插入排序中，如果待排序列是正序时，时间复杂度是 $O(n)$ ，如果序列是基本有序的，使用直接插入排序效率就非常高。希尔排序就利用了这个特点。基本思想是：先将整个待排记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录基本有序时再对全体记录进行一次直接插入排序。

举个栗子：



从上述排序过程可见，希尔排序的特点是，子序列的构成不是简单的逐段分割，而是将某个相隔某个增量的记录组成一个子序列。如上面的例子，第一趟排序时的增量为5，第二趟排序的增量为3。由于前两趟的插入排序中记录的关键字是和同一子序列中的前一个记录的关键字进行比较，因此关键字较小的记录就不是一步一步地向前挪动，而是跳跃式地往前移，从而使得进行最后一趟排序时，整个序列已经做到基本有序，只要作记录的少量比较和移动即可。因此希尔排序的效率要比直接插入排序高。

希尔排序的分析是复杂的，时间复杂度是所取增量的函数，这涉及一些数学上的难题。但是在大量实验的基础上推出当n在某个范围内时，时间复杂度可以达到 $O(n^{1.3})$ 。

实现代码：

```
public class ShellSort {

    /**
     * 希尔排序的一趟插入
     * @param arr 待排数组
     * @param d 增量
     */
    public static void shellInsert(int[] arr, int d) {
        for(int i=d; i<arr.length; i++) {
            int j = i - d;
            int temp = arr[i]; //记录要插入的数据
            while (j>=0 && arr[j]>temp) { //从后向前，找到比其小的数的位置
                arr[j+d] = arr[j]; //向后挪动
                j -= d;
            }

            if (j != i - d) //存在比其小的数
                arr[j+d] = temp;
        }
    }

    public static void shellSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        int d = arr.length / 2;
        while(d >= 1) {
            shellInsert(arr, d);
            d /= 2;
        }
    }
}
```

```

        d /= 2;
    }
}

}

```

52. 归并排序

归并排序是另一种不同的排序方法，因为归并排序使用了递归分治的思想，所以理解起来比较容易。其基本思想是，先递归划分子问题，然后合并结果。把待排序列看成由两个有序的子序列，然后合并两个子序列，然后把子序列看成由两个有序序列。。。倒着来看，其实就是先两两合并，然后四四合并。。。最终形成有序序列。空间复杂度为O(n)，时间复杂度为O(nlogn)。

举个栗子：

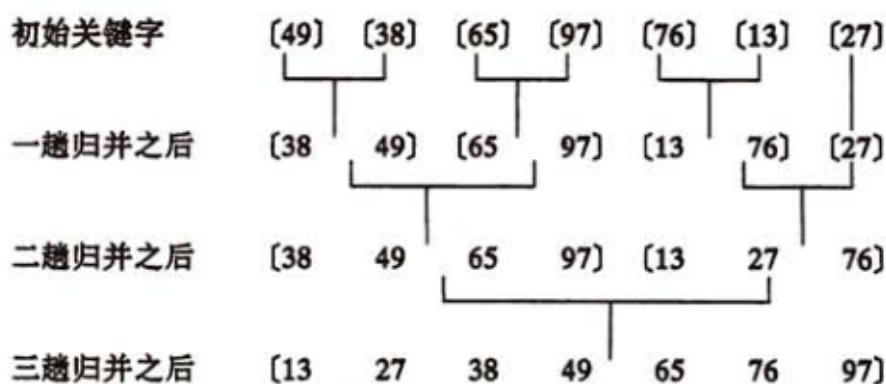


图 10.13 2-路归并排序示例

实现代码：

```

public class MergeSort {

    public static void mergeSort(int[] arr) {
        mSort(arr, 0, arr.length-1);
    }

    /**
     * 递归分治
     * @param arr 待排数组
     * @param left 左指针
     * @param right 右指针
     */
    public static void mSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int mid = (left + right) / 2;

        mSort(arr, left, mid); //递归排序左边
        mSort(arr, mid+1, right); //递归排序右边
        merge(arr, left, mid, right); //合并
    }

    /**

```

```

* 合并两个有序数组
* @param arr 待合并数组
* @param left 左指针
* @param mid 中间指针
* @param right 右指针
*/
public static void merge(int[] arr, int left, int mid, int right) {
    // [left, mid] [mid+1, right]
    int[] temp = new int[right - left + 1]; //中间数组

    int i = left;
    int j = mid + 1;
    int k = 0;
    while(i <= mid && j <= right) {
        if(arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while(i <= mid) {
        temp[k++] = arr[i++];
    }

    while(j <= right) {
        temp[k++] = arr[j++];
    }

    for(int p=0; p<temp.length; p++) {
        arr[left + p] = temp[p];
    }
}

}

```

53. 计数排序

如果在面试中有面试官要求你写一个 $O(n)$ 时间复杂度的排序算法，你千万不要立刻说：这不可能！虽然前面基于比较的排序的下限是 $O(n \log n)$ 。但是确实也有线性时间复杂度的排序，只不过有前提条件，就是待排序的数要满足一定的范围的整数，而且计数排序需要比较多的辅助空间。其基本思想是，用待排序的数作为计数数组的下标，统计每个数字的个数。然后依次输出即可得到有序序列。

实现代码：

```

public class CountSort {

    public static void countSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        int max = max(arr);

        int[] count = new int[max+1];

```

```

        Arrays.fill(count, 0);

        for(int i=0; i<arr.length; i++) {
            count[arr[i]]++;
        }

        int k = 0;
        for(int i=0; i<=max; i++) {
            for(int j=0; j<count[i]; j++) {
                arr[k++] = i;
            }
        }
    }

    public static int max(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int ele : arr) {
            if(ele > max)
                max = ele;
        }

        return max;
    }
}

```

54. 桶排序

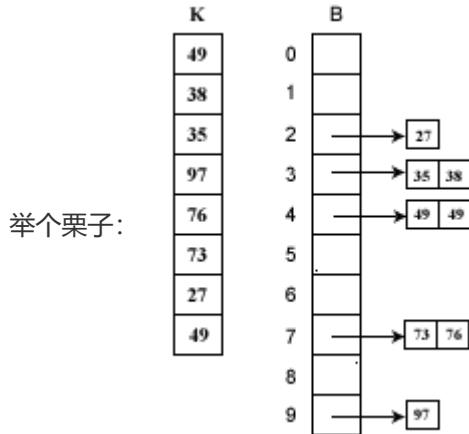
桶排序算是计数排序的一种改进和推广，但是网上有许多资料把计数排序和桶排序混为一谈。其实桶排序要比计数排序复杂许多。

对桶排序的分析和解释借鉴这位兄弟的文章（有改动）：<http://hxraid.iteye.com/blog/647759>

桶排序的基本思想：

假设有一组长度为N的待排关键字序列K[1....n]。首先将这个序列划分成M个的子区间(桶)。然后基于某种映射函数，将待排序列的关键字k映射到第i个桶中(即桶数组B的下标 i)，那么该关键字k就作为B[i]中的元素(每个桶B[i]都是一组大小为N/M的序列)。接着对每个桶B[i]中的所有元素进行比较排序(可以使用快排)。然后依次枚举输出B[0]....B[M]中的全部内容即是一个有序序列。bindex=f(key)其中，bindex 为桶数组B的下标(即第bindex个桶)，k为待排序列的关键字。桶排序之所以能够高效，其关键在于这个映射函数，它必须做到：如果关键字k1<k2，那么f(k1)<=f(k2)。

也就是说B(i)中的最小数据都要大于B(i-1)中最大数据。很显然，映射函数的确定与数据本身的特点有很大的关系。



假如待排序列 $K=\{49, 38, 35, 97, 76, 73, 27, 49\}$ 。这些数据全部在1—100之间。因此我们定制10个桶，然后确定映射函数 $f(k)=k/10$ 。则第一个关键字49将定位到第4个桶中($49/10=4$)。依次将所有关键字全部堆入桶中，并在每个非空的桶中进行快速排序后得到如图所示。只要顺序输出每个 $B[i]$ 中的数据就可以得到有序序列了。

桶排序分析：

桶排序利用函数的映射关系，减少了几乎所有的比较工作。实际上，桶排序的 $f(k)$ 值的计算，其作用就相当于快排中划分，希尔排序中的子序列，归并排序中的子问题，已经把大量数据分割成了基本有序的数据块(桶)。然后只需要对桶中的少量数据做先进的比较排序即可。

对 N 个关键字进行桶排序的时间复杂度分为两个部分：

(1) 循环计算每个关键字的桶映射函数，这个时间复杂度是 $O(N)$ 。

(2) 利用先进的比较排序算法对每个桶内的所有数据进行排序，其时间复杂度为 $\sum O(N_i \log N_i)$ 。其中 N_i 为第 i 个桶的数据量。

很显然，第(2)部分是桶排序性能好坏的决定因素。尽量减少桶内数据的数量是提高效率的唯一办法(因为基于比较排序的最好平均时间复杂度只能达到 $O(N \log N)$ 了)。因此，我们需要尽量做到下面两点：

(1) 映射函数 $f(k)$ 能够将 N 个数据平均的分配到 M 个桶中，这样每个桶就有 $[N/M]$ 个数据量。

(2) 尽量的增大桶的数量。极限情况下每个桶只能得到一个数据，这样就完全避开了桶内数据的“比较”排序操作。当然，做到这一点很不容易，数据量巨大的情况下， $f(k)$ 函数会使得桶集合的数量巨大，空间浪费严重。这就是一个时间代价和空间代价的权衡问题了。

对于 N 个待排数据， M 个桶，平均每个桶 $[N/M]$ 个数据的桶排序平均时间复杂度为：

$O(N) + O(M[N/M] \log(N/M)) = O(N + N(\log N - \log M)) = O(N + N \log N - N \log M)$ 当 $N=M$ 时，即极限情况下每个桶只有一个数据时。桶排序的最好效率能够达到 $O(N)$ 。

总结：桶排序的平均时间复杂度为线性的 $O(N+C)$ ，其中 $C=N*(\log N - \log M)$ 。如果相对于同样的 N ，桶数量 M 越大，其效率越高，最好的时间复杂度达到 $O(N)$ 。当然桶排序的空间复杂度为 $O(N+M)$ ，如果输入数据非常庞大，而桶的数量也非常多，则空间代价无疑是昂贵的。此外，桶排序是稳定的。

实现代码：

```

public class BucketSort {
    public static void bucketSort(int[] arr) {
        if(arr == null && arr.length == 0)
            return;

        int bucketNums = 10; //这里默认为10，规定待排数[0,100)
        List<List<Integer>> buckets = new ArrayList<List<Integer>>(); //桶的索引
    }
}

```

```

        for(int i=0; i<10; i++) {
            buckets.add(new LinkedList<Integer>()); //用链表比较合适
        }

        //划分桶
        for(int i=0; i<arr.length; i++) {
            buckets.get(f(arr[i])).add(arr[i]);
        }

        //对每个桶进行排序
        for(int i=0; i<buckets.size(); i++) {
            if(!buckets.get(i).isEmpty()) {
                Collections.sort(buckets.get(i)); //对每个桶进行快排
            }
        }

        //还原排好序的数组
        int k = 0;
        for(List<Integer> bucket : buckets) {
            for(int ele : bucket) {
                arr[k++] = ele;
            }
        }

    }

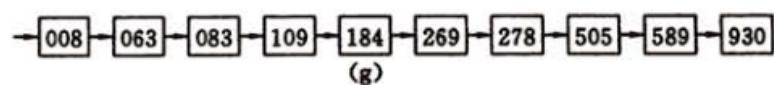
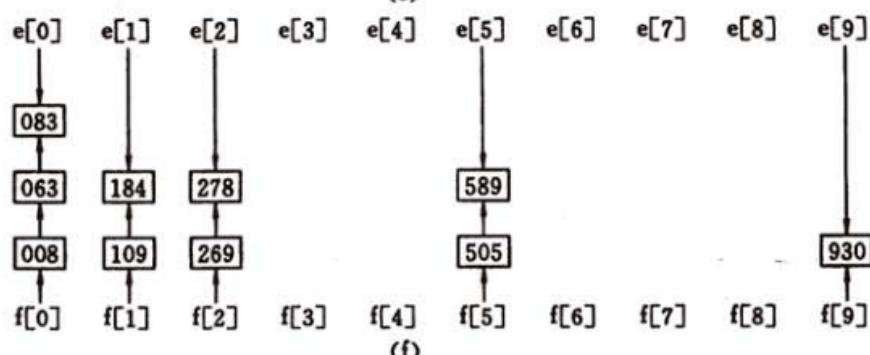
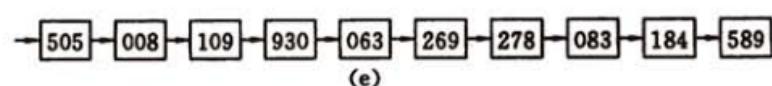
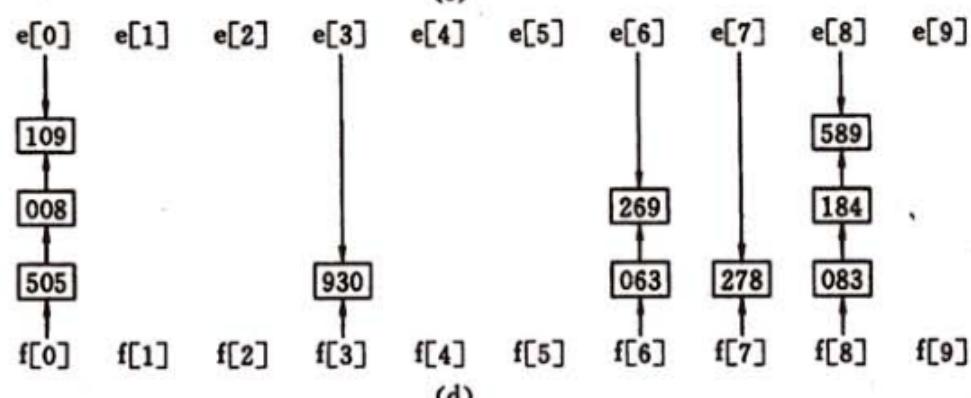
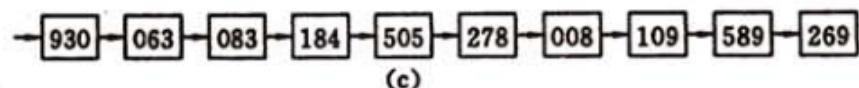
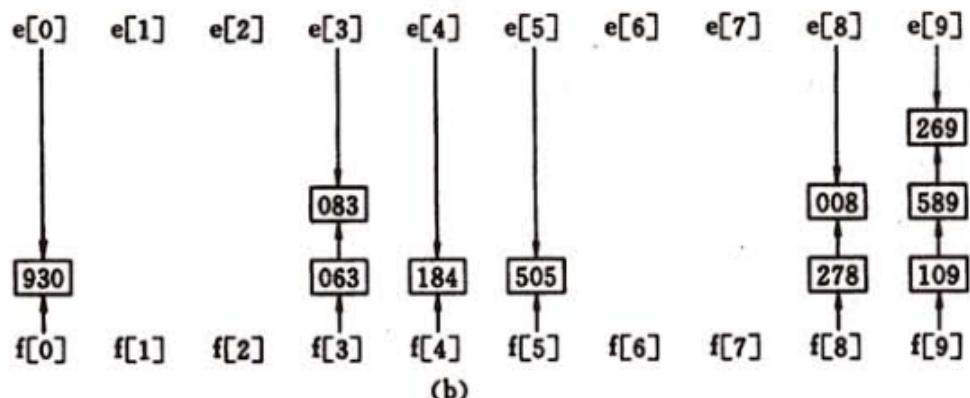
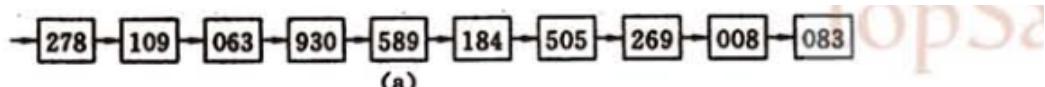
    /**
     * 映射函数
     * @param x
     * @return
     */
    public static int f(int x) {
        return x / 10;
    }
}

```

55. 基数排序

基数排序又是一种和前面排序方式不同的排序方式，基数排序不需要进行记录关键字之间的比较。基数排序是一种借助多关键字排序思想对单逻辑关键字进行排序的方法。所谓的多关键字排序就是有多个优先级不同的关键字。比如说成绩的排序，如果两个人总分相同，则语文高的排在前面，语文成绩也相同则数学高的排在前面。。。如果对数字进行排序，那么个位、十位、百位就是不同优先级的关键字，如果要进行升序排序，那么个位、十位、百位优先级一次增加。基数排序是通过多次的收分配和收集来实现的，关键字优先级低的先进行分配和收集。

举个栗子：



实现代码：

```
public class RadixSort {  
  
    public static void radixSort(int[] arr) {  
        if(arr == null && arr.length == 0)  
            return ;  
  
        int maxBit = getMaxBit(arr);
```

```

        for(int i=1; i<=maxBit; i++) {

            List<List<Integer>> buf = distribute(arr, i); //分配
            collecte(arr, buf); //收集
        }

    }

    /**
     * 分配
     * @param arr 待分配数组
     * @param iBit 要分配第几位
     * @return
     */
    public static List<List<Integer>> distribute(int[] arr, int iBit) {
        List<List<Integer>> buf = new ArrayList<List<Integer>>();
        for(int j=0; j<10; j++) {
            buf.add(new LinkedList<Integer>());
        }
        for(int i=0; i<arr.length; i++) {
            buf.get(getNBit(arr[i], iBit)).add(arr[i]);
        }
        return buf;
    }

    /**
     * 收集
     * @param arr 把分配的数据收集到arr中
     * @param buf
     */
    public static void collecte(int[] arr, List<List<Integer>> buf) {
        int k = 0;
        for(List<Integer> bucket : buf) {
            for(int ele : bucket) {
                arr[k++] = ele;
            }
        }
    }

}

/**
 * 获取最大位数
 * @param x
 * @return
 */
public static int getMaxBit(int[] arr) {
    int max = Integer.MIN_VALUE;
    for(int ele : arr) {
        int len = (ele+"").length();
        if(len > max)
            max = len;
    }
    return max;
}

```

```

* 获取x的第n位, 如果没有则为0.
* @param x
* @param n
* @return
*/
public static int getNBit(int x, int n) {

    String sx = x + "";
    if(sx.length() < n)
        return 0;
    else
        return sx.charAt(sx.length()-n) - '0';
}

}

```

###

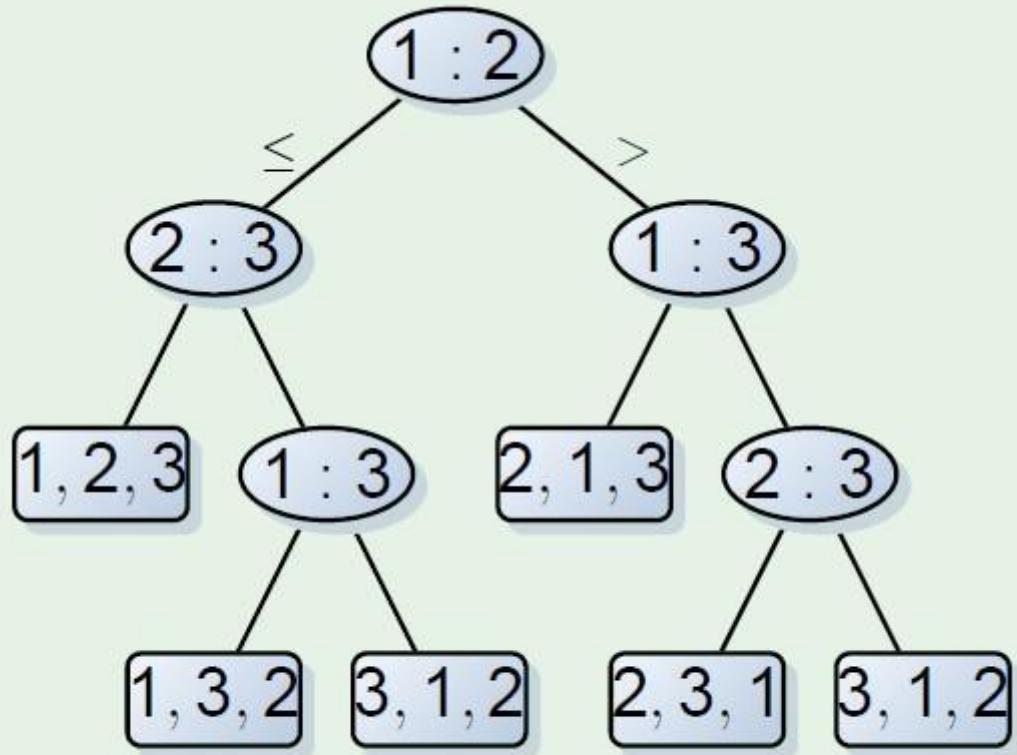
56. 排序算法各自的使用场景和适用场合。

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$

- 从平均时间来看, 快速排序是效率最高的, 但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后者相比较的结果是, 在n较大时归并排序使用时间较少, 但使用辅助空间较多。
- 上面说的简单排序包括除希尔排序之外的所有冒泡排序、插入排序、简单选择排序。其中直接插入排序最简单, 但序列基本有序或者n较小时, 直接插入排序是好的方法, 因此常将它和其他的排序方法, 如快速排序、归并排序等结合在一起使用。
- 基数排序的时间复杂度也可以写成 $O(d*n)$ 。因此它最适合于n值很大而关键字较小的的序列。若关键字也很大, 而序列中大多数记录的最高关键字均不同, 则亦可先按最高关键字不同, 将序列分成若干小的子序列, 而后进行直接插入排序。
- 从方法的稳定性来比较, 基数排序是稳定的内排方法, 所有时间复杂度为 $O(n^2)$ 的简单排序也是稳定的。但是快速排序、堆排序、希尔排序等时间性能较好的排序方法都是不稳定的。稳定性需要根据具体需求选择。
- 上面的算法实现大多数是使用线性存储结构, 像插入排序这种算法用链表实现更好, 省去了移动元素的时间。具体的存储结构在具体的实现版本中也是不同的。

附：基于比较排序算法时间下限为 $O(n \log n)$ 的证明：

基于比较排序下限的证明是通过决策树证明的, 决策树的高度 $\Omega(n \lg n)$, 这样就得出了比较排序的下限。



$$\text{Sort} \langle a_1, a_2, a_3 \rangle = \langle 6, 8, 5 \rangle$$

首先要引入决策树。首先决策树是一颗二叉树，每个节点表示元素之间一组可能的排序，它予以京进行的比较相一致，比较的结果是树的边。先来说明一些二叉树的性质，令T是深度为d的二叉树，则T最多有 2^d 片树叶。具有L片树叶的二叉树的深度至少是 $\log L$ 。所以，对n个元素排序的决策树必然有 $n!$ 片树叶（因为n个数有 $n!$ 种不同的大小关系），所以决策树的深度至少是 $\log(n!)$ ，即至少需要 $\log(n!)$ 次比较。而 $\log(n!) = \log n + \log(n-1) + \log(n-2) + \dots + \log 2 + \log 1 \geq \log n + \log(n-1) + \log(n-2) + \dots + \log(n/2) \geq (n/2)\log(n/2) \geq (n/2)\log n - n/2 = O(n\log n)$ 所以只用到比较的排序算法最低时间复杂度是 $O(n\log n)$ 。

网络协议面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. 什么是网络编程

- 网络编程的本质是多台计算机之间的数据交换。数据传递本身没有多大的难度，不就是把一个设备中的数据发送给其他设备，然后接受另外一个设备反馈的数据。现在的网络编程基本上都是基于请求/响应方式的，也就是一个设备发送请求数据给另外一个，然后接收另一个设备的反馈。在网络编程中，发起连接程序，也就是发送第一次请求的程序，被称作客户端(Client)，等待其他程序连接的程序被称作服务器(Server)。客户端程序可以在需要的时候启动，而服务器为了能够时刻相应连接，则需要一直启动。
- 例如以打电话为例，首先拨号的人类似于客户端，接听电话的人必须保持电话畅通类似于服务器。连接一旦建立以后，就客户端和服务器端就可以进行数据传递了，而且两者的身份是等价的。在一些程序中，程序既有客户端功能也有服务器端功能，最常见的软件就是QQ、微信这类软件了。

2. 网络编程中两个主要的问题

- 1、一个是如何准确的定位网络上一台或多台主机，
- 2、另一个就是找到主机后如何可靠高效的进行数据传输。

- 在TCP/IP协议中IP层主要负责网络主机的定位，数据传输的路由，由IP地址可以唯一地确定Internet上的一台主机。
- 而TCP层则提供面向应用的可靠（TCP）的或非可靠（UDP）的数据传输机制，这是网络编程的主要对象，一般不需要关心IP层是如何处理数据的。
- 目前较为流行的网络编程模型是客户机/服务器（C/S）结构。即通信双方一方作为服务器等待客户提出请求并予以响应。客户则在需要服务时向服务器提出申请。服务器一般作为守护进程始终运行，监听网络端口，一旦有客户请求，就会启动一个服务进程来响应该客户，同时自己继续监听服务端口，使后来的客户也能及时得到服务。

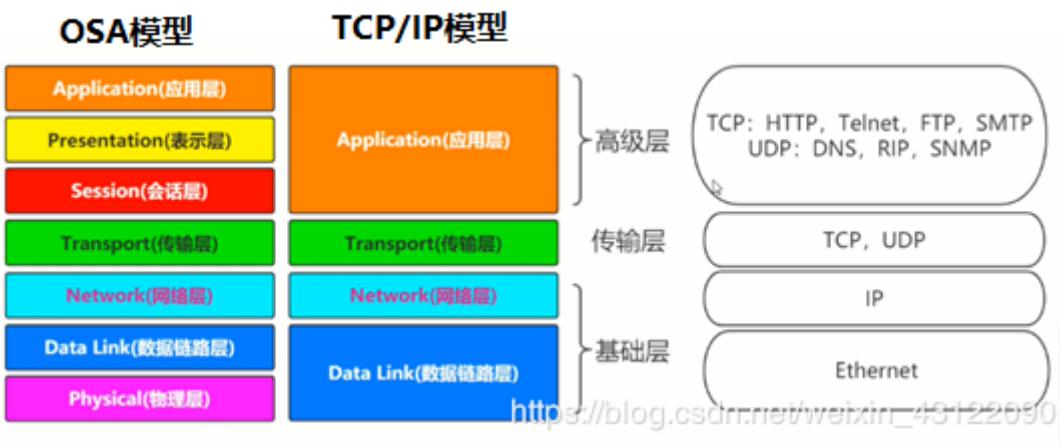
3. 网络协议是什么

在计算机网络要做到井井有条的交换数据，就必须遵守一些事先约定好的规则，比如交换数据的格式、是否需要发送一个应答信息。这些规则被称为网络协议。

4. 为什么要对网络协议分层

- 1、简化问题难度和复杂度。由于各层之间独立，我们可以分割大问题为小问题。
- 2、灵活性好。当其中一层的技术变化时，只要层间接口关系保持不变，其他层不受影响。
- 3、易于实现和维护。
- 4、促进标准化工作。分开后，每层功能可以相对简单地被描述

5. 计算机网络体系结构



OSI参考模型

- OSI (Open System Interconnect)，即开放式系统互联。一般都叫OSI参考模型，是ISO (国际标准化组织) 组织在1985年研究的网络互连模型。ISO为了更好的使网络应用更为普及，推出了OSI参考模型，这样所有的公司都按照统一的标准来指定自己的网络，就可以互通互联了。
- OSI定义了网络互连的七层框架 (物理层、数据链路层、网络层、传输层、会话层、表示层、应用层)。

OSI七层模型	功能	对应的网络协议	TCP/IP四层概念模型
应用层	文件传输，文件管理，电子邮件的信息处理——apdu	HTTP、TFTP、FTP、NFS、WAIS、SMTP	
表示层	确保一个系统的应用层发送的消息可以被另一个系统的应用层读取，编码转换，数据解析，管理数据的解密和加密，最小单位——ppdu	Telnet、Rlogin、SNMP、Gopher	
会话层	负责在网络中的两节点建立，维持和终止通信，在一层协议中，可以解决节点连接的协调和管理问题。包括通信连接的建立，保持会话过程通信连接的畅通，两节点之间的对话，决定通信是否被终端一斤通信终端是决定从何处重新发送，最小单位——spdu	SMTP、DNS	应用层
传输层	定义一些传输数据的协议和端口。传输协议同时进行流量控制，或是根据接收方接收数据的快慢程度，规定适当的发送速率，解决传输效率及能力的问题——tpdu	TCP、UDP	传输层
网络层	控制子网的运行，如逻辑编址，分组传输，路由选择最小单位——分组（包）报文	IP、ICMP、ARP、RARP、AKP、UUCP	网络层
数据链路层	主要是对物理层传输的比特流包装，检测保证数据传输的可靠性，将物理层接收的数据进行MAC（媒体访问控制）地址的封装和解封装，也可以简单的理解为物理寻址。交换机就处在这一层，最小的传输单位——帧	FDDI、Ethernet、Arpanet、PDN、SLIP、PPP、STP、HDLC、SDLC、帧中继	数据链路层
物理层	定义物理设备的标准，主要对物理连接方式，电气特性，机械特性等制定统一标准，传输比特流，因此最小的传输单位——位（比特流）	IEEE 802.1A、IEEE 802.2到IEEE 802.	

https://blog.csdn.net/weixin_43122090

TCP/IP参考模型

TCP/IP四层协议（数据链路层、网络层、传输层、应用层）

- 1、应用层** 应用层最靠近用户的一层，是为计算机用户提供应用接口，也为用户直接提供各种网络服务。我们常见应用层的网络服务协议有：HTTP、HTTPS、FTP、TELNET等。
- 2、传输层** 建立了主机端到端的链接，传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。该层向高层屏蔽了下层数据通信的细节，使高层用户看到的只是在两个传输实体间的一条主机到主机的、可由用户控制和设定的、可靠的数据通路。我们通常说的，TCP UDP就是在这一层。端口号既是这里的“端”。
- 3、网络层** 本层通过IP寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点，正确无误地按照地址传送给目的端的运输层。就是通常说的IP层。这一层就是我们经常说的IP协议层。IP协议是Internet的基础。

4、数据链路层 通过一些规程或协议来控制这些数据的传输，以保证被传输数据的正确性。实现这些规程或协议的 硬件 和软件加到物理线路，这样就构成了数据链路，

6. 什么是TCP/IP和UDP

1、TCP/IP即传输控制/网络协议，是面向连接的协议，发送数据前要先建立连接(发送方和接收方的成对的两个之间必须建立连接)，TCP提供可靠的服务，也就是说，通过TCP连接传输的数据不会丢失，没有重复，并且按顺序到达

2、UDP它是属于TCP/IP协议族中的一种。是无连接的协议，发送数据前不需要建立连接，是没有可靠性的协议。因为不需要建立连接所以可以在在网络上以任何可能的路径传输，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。

7. TCP与UDP区别：

1、TCP是面向连接的协议，发送数据前要先建立连接，TCP提供可靠的服务，也就是说，通过TCP连接传输的数据不会丢失，没有重复，并且按顺序到达；

2、UDP是无连接的协议，发送数据前不需要建立连接，是没有可靠性；

3、TCP通信类似于要打个电话，接通了，确认身份后，才开始进行通行；

4、UDP通信类似于学校广播，靠着广播播报直接进行通信。

5、TCP只支持点对点通信，UDP支持一对一、一对多、多对一、多对多；

6、TCP是面向字节流的，UDP是面向报文的；面向字节流是指发送数据时以字节为单位，一个数据包可以拆分成若干组进行发送，而UDP一个报文只能一次发完。

7、TCP首部开销（20字节）比UDP首部开销（8字节）要大

8、UDP 的主机不需要维持复杂的连接状态表

8. TCP和UDP的应用场景：

对某些实时性要求比较高的情况使用UDP，比如游戏，媒体通信，实时直播，即使出现传输错误也可以容忍；其它大部分情况下，HTTP都是用TCP，因为要求传输的内容可靠，不出现丢失的情况

9. 形容一下TCP和UDP

TCP通信可看作打电话：

李三(拨了个号码)：喂，是王五吗？王五：哎，您谁啊？李三：我是李三，我想给你说点事儿，你现在方便吗？王五：哦，我现在方便，你说吧。甲：那我说了啊？乙：你说吧。（连接建立了，接下来就是说正事了...）

UDP通信可看为学校里的广播：

播音室：喂喂喂！全体操场集合

10. 运行在TCP或UDP的应用层协议分析。

运行在TCP协议上的协议：

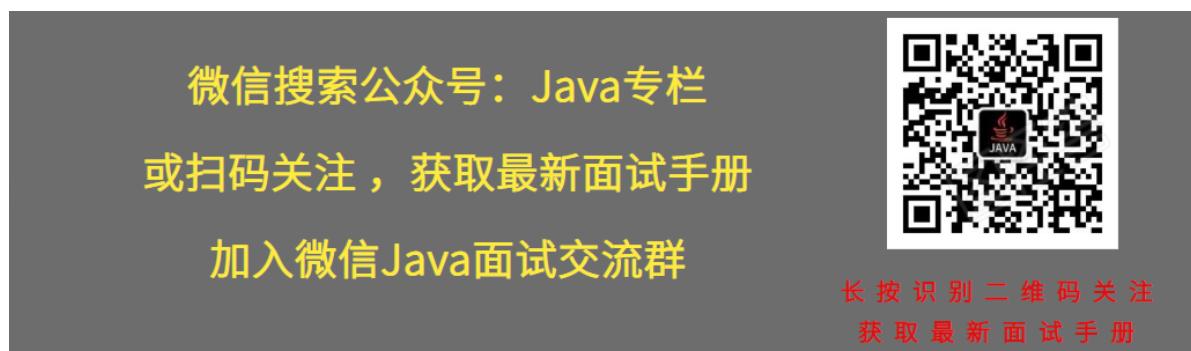
- HTTP (Hypertext Transfer Protocol, 超文本传输协议) , 主要用于普通浏览。
- HTTPS (HTTP over SSL, 安全超文本传输协议) , HTTP协议的安全版本。
- FTP (File Transfer Protocol, 文件传输协议) , 用于文件传输。
- POP3 (Post Office Protocol, version 3, 邮局协议) , 收邮件用。
- SMTP (Simple Mail Transfer Protocol, 简单邮件传输协议) , 用来发送电子邮件。
- TELNET (Teletype over the Network, 网络电传) , 通过一个终端 (terminal) 登陆到网络。
- SSH (Secure Shell, 用于替代安全性差的TELNET) , 用于加密安全登陆用。

运行在UDP协议上的协议：

- BOOTP (Boot Protocol, 启动协议) , 应用于无盘设备。
- NTP (Network Time Protocol, 网络时间协议) , 用于网络同步。
- DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议) , 动态配置IP地址。

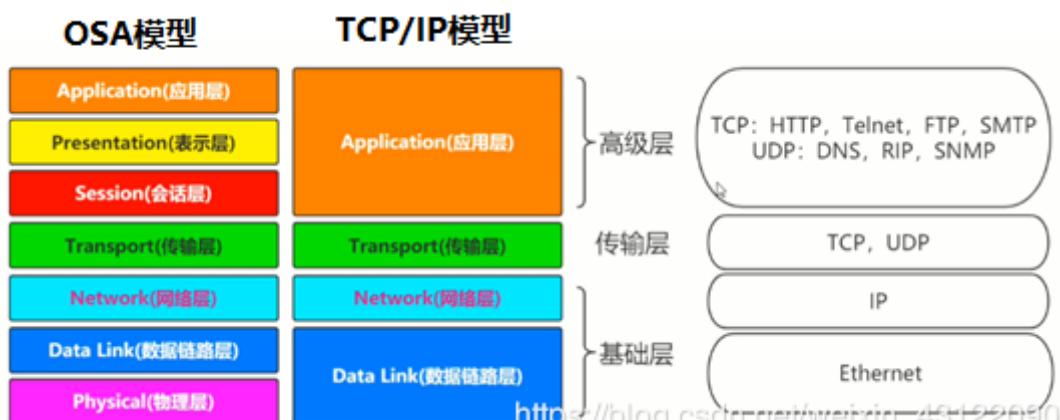
运行在TCP和UDP协议上：

- DNS (Domain Name Service, 域名服务) , 用于完成地址查找, 邮件转发等工作。
- ECHO (Echo Protocol, 回绕协议) , 用于查错及测量应答时间 (运行在TCP和UDP协议上) 。
- SNMP (Simple Network Management Protocol, 简单网络管理协议) , 用于网络信息的收集和网络管理。
- DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议) , 动态配置IP地址。
- ARP (Address Resolution Protocol, 地址解析协议) , 用于动态解析以太网硬件的地址。



11. 什么是Http协议？

- Http协议是对客户端和服务器端之间数据之间实现可靠性的传输文字、图片、音频、视频等超文本数据的规范，格式简称为“超文本传输协议”
- Http协议属于应用层，及用户访问的第一层就是http



12. Http和Https的区别?

Http协议运行在TCP之上，明文传输，客户端与服务器端都无法验证对方的身份；Https是身披SSL(Secure Socket Layer)外壳的Http，运行于SSL上，SSL运行于TCP之上，是添加了加密和认证机制的HTTP。二者之间存在如下不同：

- 端口不同：Http与Https使用不同的连接方式，用的端口也不一样，前者是80，后者是443；
- 资源消耗：和HTTP通信相比，Https通信会由于加减密处理消耗更多的CPU和内存资源；
- 开销：Https通信需要证书，而证书一般需要向认证机构购买；

Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。

13. 什么是http的请求体？

- 1、HTTP请求体是我们请求数据时先发送给服务器的数据，毕竟我向服务器那数据，先要表明我要什么吧
- 2、HTTP请求体由：请求行、请求头、请求数据组成的，
- 3、注意：GIT请求是没有请求体的

POST请求

```
POST http://120.77.0.58:8888/api/admin/login HTTP/1.1
Host: 120.77.0.58:8888
Connection: keep-alive
Content-Length: 46
Accept: application/json, text/plain, */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36
Content-Type: application/json;charset=UTF-8
Referer: http://120.77.0.58:8888/admin/index.html
Origin: http://120.77.0.58:8888
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
Cookie: rdt_uuid=75e59825-2a77-4a39-8626-dc3de0cdec49; _ga=GA1.1.498866331.1583946077; sidebar_collapsed=false; scroll-cookie=0!; rememberme=1
{"username": "468671109", "password": "520lifie"}
```

GET请求是没有请求体的

```
GET http://120.77.0.58:8888/archives/java%9D%A2%8%AF%95%E9%A2%98-%E4%BD%A0%E7%9F%A5%E9%81%93.jsp%5%90%97 HTTP/1.1
Host: 120.77.0.58:8888
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://120.77.0.58:8888/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
Cookie: rdt_uuid=75e59825-2a77-4a39-8626-dc3de0cdec49; _ga=GA1.1.498866331.1583946077; sidebar_collapsed=false; scroll-cookie=0!; rememberme=1
```

14. HTTP的响应报文有哪些？

1、http的响应报是服务器返回给我们的数据，必须先有请求体再有响应报文

2、响应报文包含三部分 状态行、响应首部字段、响应内容实体实现

```
HTTP/1.1 200 OK
Access-Control-Allow-Headers: Content-Type, ADMIN-Authorization, API-Authorization
Date: Tue, 31 Mar 2020 08:48:34 GMT
Connection: keep-alive
Access-Control-Allow-Credentials: true
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
Content-Language: zh-CN
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Max-Age: 3600

3fec
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="google-site-verification" content="" />
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="TOC先想想一些问题1我们开发人员编写的Java代码是怎么让电脑认识的首先先了解电脑">
    <meta name="keyword" content="" />
    <meta name="generator" content="Halo 1.3.0"/>
```

https://blog.csdn.net/weixin_43122090

15. HTTPS工作原理

- 1、首先HTTP请求服务端生成证书，客户端对证书的有效期、合法性、域名是否与请求的域名一致、证书的公钥（RSA加密）等进行校验；
- 2、客户端如果校验通过后，就根据证书的公钥的有效，生成随机数，随机数使用公钥进行加密（RSA加密）；
- 3、消息体产生的后，对它的摘要进行MD5（或者SHA1）算法加密，此时就得到了RSA签名；
- 4、发送给服务端，此时只有服务端（RSA私钥）能解密。
- 5、解密得到的随机数，再用AES加密，作为密钥（此时的密钥只有客户端和服务端知道）。

16. 三次握手与四次挥手

(1). 三次握手（我要和你建立链接，你真的要和我建立链接么，我真的要和你建立链接，成功）

- 第一次握手：Client将标志位SYN置为1，随机产生一个值seq=j，并将该数据包发送给Server，Client进入SYN_SENT状态，等待Server确认。
- 第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=j+1，随机产生一个值seq=k，并将该数据包发送给Client以确认连接请求，Server进入SYN_RECV状态。
- 第三次握手：Client收到确认后，检查ack是否为j+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=k+1，并将该数据包发送给Server，Server检查ack是否为k+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

(2). 四次挥手（我要和你断开链接；好的，断吧。我也要和你断开链接；好的，断吧）：

- 第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。
- 第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。此时TCP链接处于半关闭状态，即客户端已经没有要发送的数据了，但服务端若发送数据，则客户端仍要接收。
- 第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST_ACK状态。
- 第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

17. 为什么 TCP 链接需要三次握手，两次不可以么？

“三次握手”的目的是为了防止已失效的连接请求报文突然又传到了服务端，因而产生错误。

- 正常的情况：A发出连接请求，但因连接请求报文丢失而未收到确认，于是A再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接。A共发送了两个连接请求报文段，其中第一个丢失，第二个到达了B。没有“已失效的连接请求报文段”。
- 现假定出现了一种异常情况：即A发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达B。本来这是一个早已失效的报文段。但B收到此失效的连接请求报文段后，就误认为是A再次发出一个新的连接请求。于是就向A发出确认报文段，同意建立连接。

假设不采用“三次握手”，那么只要B发出确认，新的连接就建立了。由于现在A并没有发出建立连接的请求，因此不会理睬B的确认，也不会向B发送数据。但B却以为新的运输连接已经建立，并一直等待A发来数据。这样，B的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。

18. 用现实理解三次握手的具体细节

三次握手的目的是建立可靠的通信信道，主要的目的就是双方确认自己与对方的发送与接收机能正常。

- 1、第一次握手：客户什么都不能确认；服务器确认了对方发送正常
- 2、第二次握手：客户确认了：自己发送、接收正常，对方发送、接收正常；服务器确认了：自己接收正常，对方发送正常
- 3、第三次握手：客户确认了：自己发送、接收正常，对方发送、接收正常；服务器确认了：自己发送、接收正常，对方发送接收正常 所以三次握手就能确认双发收发功能都正常，缺一不可。

19. 建立连接可以两次握手吗？为什么？

不可以。

因为可能会出现已失效的连接请求报文段又传到了服务器端。> client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。

而且，两次握手无法保证Client正确接收第二次握手的报文（Server无法确认Client是否收到），也无法保证Client和Server之间成功互换初始序列号。

20. 为什么要四次挥手？

TCP 协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP 是全双工模式，这就意味着，当 A 向 B 发出 FIN 报文段时，只是表示 A 已经没有数据要发送了，而此时 A 还是能够接受到来自 B 发出的数据；B 向 A 发出 ACK 报文段也只是告诉 A，它自己知道 A 没有数据要发了，但 B 还是能够向 A 发送数据。

所以想要愉快的结束这次对话就需要四次挥手。

21. TCP 协议如何来保证传输的可靠性

TCP 提供一种面向连接的、可靠的字节流服务。其中，面向连接意味着两个使用 TCP 的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个 TCP 连接。在一个 TCP 连接中，仅有两方进行彼此通信；而字节流服务意味着两个应用程序通过 TCP 链接交换 8 bit 字节构成的字节流，TCP 不在字节流中插入记录标识符。

对于可靠性，TCP 通过以下方式进行保证：

- **数据包校验**：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
- **对失序数据包重排序**：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
- **丢弃重复数据**：对于重复数据，能够丢弃重复数据；
- **应答机制**：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；
- **超时重发**：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；
- **流量控制**：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议是可变大小的滑动窗口协议。

22. 客户端不断进行请求链接会怎样？DDos(Distributed Denial of Service)攻击？

服务器端会为每个请求创建一个链接，并向其发送确认报文，然后等待客户端进行确认

(1). DDos 攻击：

- 客户端向服务端发送请求链接数据包
- 服务端向客户端发送确认数据包
- 客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认

(2). DDos 预防：(没有彻底根治的办法，除非不使用TCP)

- 限制同时打开 SYN 半链接的数目
- 缩短 SYN 半链接的 Time out 时间
- 关闭不必要的服务

23. GET 与 POST 的区别?

GET与POST是我们常用的两种HTTP Method，二者之间的区别主要包括如下五个方面：

- 1、从功能上讲，GET一般用来从服务器上获取资源，POST一般用来更新服务器上的资源；
- 2、从REST服务角度上说，GET是幂等的，即读取同一个资源，总是得到相同的数据，而POST不是幂等的，因为每次请求对资源的改变并不是相同的；进一步地，GET不会改变服务器上的资源，而POST会对服务器资源进行改变；
- 3、从请求参数形式上看，GET请求的数据会附在URL之后，即将请求数据放置在HTTP报文的 请求头中，以?分割URL和传输数据，参数之间以&相连。特别地，如果数据是英文字母/数字，原样发送；否则，会将其编码为 application/x-www-form-urlencoded MIME 字符串(如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如：%E4%BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII)；而POST请求会把提交的数据则放置在是HTTP请求报文的 请求体中。
- 4、就安全性而言，POST的安全性要比GET的安全性高，因为GET请求提交的数据将明文出现在URL上，而且POST请求参数则被包装到请求体中，相对更安全。
- 5、从请求的大小看，GET请求的长度受限于浏览器或服务器对URL长度的限制，允许发送的数据量比较小，而POST请求则是没有大小限制的。

24. 为什么在GET请求中会对URL进行编码？

我们知道，在GET请求中会对URL中非西文字符进行编码，这样做的目的就是为了**避免歧义**。看下面的例子，

针对“name1=value1&name2=value2”的例子，我们来谈一下数据从客户端到服务端的解析过程。首先，上述字符串在计算机中用ASCII码表示为：

```
6E616D6531 3D 76616C756531 26 6E616D6532 3D 76616C756532 6E616D6531: name1  
3D: = 76616C756531: value1 26: & 6E616D6532: name2 3D: = 76616C756532:  
value2复制代码
```

服务端在接收到该数据后就可以遍历该字节流，一个字节一个字节的吃，当吃到3D这字节后，服务端就知道前面吃得字节表示一个key，再往后吃，如果遇到26，说明从刚才吃的3D到26字节之间的是上一个key的value，以此类推就可以解析出客户端传过来的参数。

现在考虑这样一个问题，如果我们的参数值中就包含=或&这种特殊字符的时候该怎么办？比如，“name1=value1”，其中value1的值是“va&lu=e1”字符串，那么实际在传输过程中就会变成这样“name1=va&lu=e1”。这样，我们的本意是只有一个键值对，但是服务端却会解析成两个键值对，这样就产生了歧义。

那么，如何解决上述问题带来的歧义呢？解决的办法就是对参数进行URL编码：例如，我们对上述会产生歧义的字符进行URL编码后结果：“name1=va%26lu%3D”，这样服务端会把紧跟在“%”后的字节当成普通的字节，就是不会把它当成各个参数或键值对的分隔符。

25. TCP与UDP的区别

TCP (Transmission Control Protocol)和UDP(User Datagram Protocol)协议属于传输层协议，它们之间的区别包括：

- TCP是面向连接的， UDP是无连接的；
- TCP是可靠的， UDP是不可靠的；
- TCP只支持点对点通信， UDP支持一对一、一对多、多对一、多对多的通信模式；
- TCP是面向字节流的， UDP是面向报文的；
- TCP有拥塞控制机制;UDP没有拥塞控制，适合媒体通信；
- TCP首部开销(20个字节)比UDP的首部开销(8个字节)要大；

26. TCP和UDP分别对应的常见应用层协议

1. TCP 对应的应用层协议：

- **FTP**: 定义了文件传输协议，使用21端口。常说某某计算机开了FTP服务便是启动了文件传输服务。下载文件，上传主页，都要用到FTP服务。
- **Telnet**: 它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于DOS模式下的通信服务。如以前的BBS是纯字符界面的，支持BBS的服务器将23端口打开，对外提供服务。
- **SMTP**: 定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置-中常看到有这么SMTP端口设置这个栏，服务器开放的是25号端口。
- **POP3**: 它是和SMTP对应，POP3用于接收邮件。通常情况下，POP3协议所用的是110端口。也就是说，只要你有相应的使用POP3协议的程序（例如Foxmail或Outlook），就可以不以Web方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是163邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。
- **HTTP**: 从Web服务器传输超文本到本地浏览器的传送协议。

2. UDP 对应的应用层协议：

- **DNS**: 用于域名解析服务，将域名地址转换为IP地址。DNS用的是53号端口。
- **SNMP**: 简单网络管理协议，使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。
- **TFTP(Trivial File Transfer Protocol)**: 简单文件传输协议，该协议在熟知端口69上使用UDP服务

27. TCP 的拥塞避免机制

拥塞：对资源的需求超过了可用的资源。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降。

拥塞控制：防止过多的数据注入到网络中，使得网络中的路由器或链路不致过载。

拥塞控制的方法：.

1、慢启动 + 拥塞避免：

慢启动：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小；

拥塞避免: 拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍，这样拥塞窗口按线性规律缓慢增长。

2、快重传 + 快恢复：

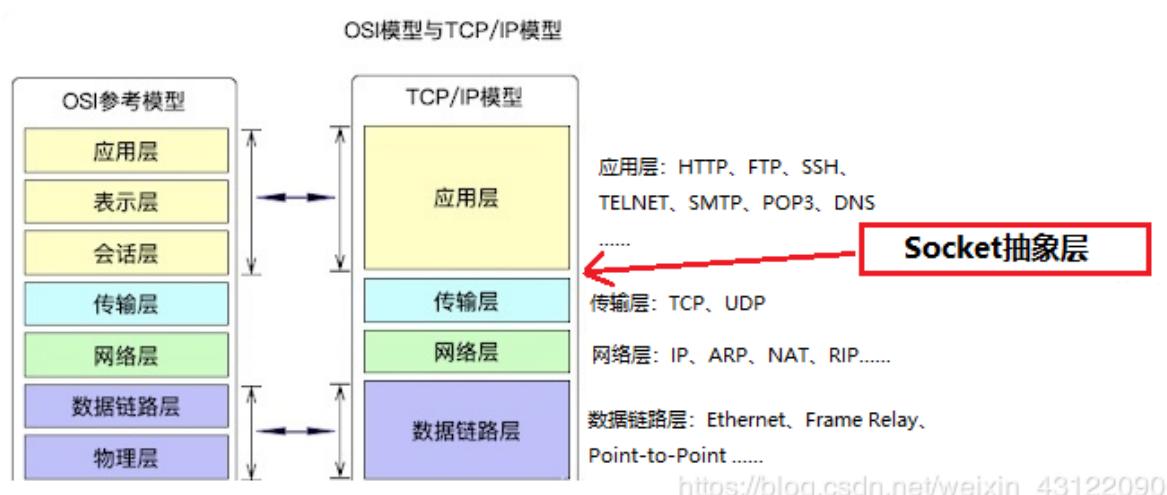
快重传: 快重传要求接收方在收到一个 **失序的报文段** 后就立即发出 **重复确认** (为的是使发送方及早知道有报文段没有到达对方) 而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

快恢复: 快重传配合使用的还有快恢复算法，当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半，但是接下去并不执行慢开始算法：因为如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。

28. 什么是Socket

- 1、网络上的两个程序通过一个双向的通讯连接实现数据的交换，这个双向链路的一端称为一个Socket。Socket通常用来实现客户方和服务方的连接。Socket是TCP/IP协议的一个十分流行的编程界面，一个Socket由一个IP地址和一个端口号唯一确定。
- 2、但是，Socket所支持的协议种类也不光TCP/IP、UDP，因此两者之间是没有必然联系的。在Java环境下，Socket编程主要是指基于TCP/IP协议的网络编程。
- 3、socket连接就是所谓的长连接，客户端和服务器需要互相连接，理论上客户端和服务器端一旦建立起连接将不会主动断掉的，但是有时候网络波动还是有可能的
- 4、Socket偏向于底层。一般很少直接使用Socket来编程，框架底层使用Socket比较多，

29. socket属于网络的那个层面



Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个外观模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

30. Socket通讯的过程

基于TCP:

服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

基于UDP:

UDP 协议是用户数据报协议的简称，也用于网络数据的传输。虽然 UDP 协议是一种不太可靠的协议，但有时在需要较快地接收数据并且可以忍受较小错误的情况下，UDP 就会表现出更大的优势。我客户端只需要发送，服务端能不能接收的到我不管

31. Socket和http的区别和应用场景

- 1、Socket连接就是所谓的长连接，理论上客户端和服务器端一旦建立起连接将不会主动断掉；
- 2、Socket适用场景：网络游戏，银行持续交互，直播，在线视屏等。
- 3、http连接就是所谓的短连接，即客户端向服务器端发送一次请求，服务器端响应后连接即会断开等待下次连接
- 4、http适用场景：公司OA服务，互联网服务，电商，办公，网站等等等等

32. 一次完整的HTTP请求所经历几个步骤？

HTTP通信机制是在一次完整的HTTP通信过程中，Web浏览器与Web服务器之间将完成下列7个步骤：

1、建立TCP连接

怎么建立连接的，看上面的三次握手

2、Web浏览器向Web服务器发送请求行

一旦建立了TCP连接，**Web浏览器就会向Web服务器发送请求命令**。例如：GET /sample/hello.jsp
HTTP/1.1。

3、Web浏览器发送请求头

浏览器发送其请求命令之后，还要以头信息的形式向Web服务器发送一些别的信息，**之后浏览器发送了一空白行来通知服务器**，它已经结束了该头信息的发送。

4、Web服务器应答

客户机向服务器发出请求后，服务器会客户机回送应答，**HTTP/1.1 200 OK，应答的第一部分是协议的版本号和应答状态码。**

5、Web服务器发送应答头

正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。

6、Web服务器向浏览器发送数据

Web服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，**它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据。**

33. 浏览器中输入：“www . xxx . com”之后都发生了什么？请详细阐述。

解析：经典的网络协议问题。

- 1、由域名→IP地址 寻找IP地址的过程依次经过了浏览器缓存、系统缓存、hosts文件、路由器缓存、递归搜索根域名服务器。
- 2、建立TCP/IP连接（三次握手具体过程）
- 3、由浏览器发送一个HTTP请求
- 4、经过路由器的转发，通过服务器的防火墙，该HTTP请求到达了服务器
- 5、服务器处理该HTTP请求，返回一个HTML文件
- 6、浏览器解析该HTML文件，并且显示在浏览器端
- 7、这里需要注意：
 - HTTP协议是一种基于TCP/IP的应用层协议，进行HTTP数据请求必须先建立TCP/IP连接
 - 可以这样理解：HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力。
 - 两个计算机之间的交流无非是两个端口之间的数据通信，具体的数据会以什么样的形式展现是以不同的应用层协议来定义的。

34. 什么是 HTTP 协议无状态协议？怎么解决Http协议无状态协议？

HTTP 是一个无状态的协议，也就是没有记忆力，这意味着每一次的请求都是独立的，缺少状态意味着如果后续处理需要前面的信息，则它必须要重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就很快。

HTTP 的这种特性有优点也有缺点：

- **优点：**解放了服务器，每一次的请求“点到为止”，不会造成不必要的连接占用
- **缺点：**每次请求会传输大量重复的内容信息，并且，在请求之间无法实现数据的共享

解决方案：

1. 使用参数传递机制：

将参数拼接在请求的 URL 后面，实现数据的传递（GET方式），例如：/param/list?

`username=wmyskxz`

问题：可以解决数据共享的问题，但是这种方式一不安全，二数据允许传输量只有1kb

2. 使用 Cookie 技术

3. 使用 Session 技术

35. Session、Cookie 与 Application

Cookie和Session都是客户端与服务器之间保持状态的解决方案，具体来说，cookie机制采用的是在客户端保持状态的方案，而session机制采用的是在服务器端保持状态的方案。

1、Cookie 及其相关 API：

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie，而客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器，服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

2、Session 及其相关 API：

同样地，会话状态也可以保存在服务器端。客户端请求服务器，如果服务器记录该用户状态，就获取Session来保存状态，这时，如果服务器已经为此客户端创建过session，服务器就按照sessionid把这个session检索出来使用；如果客户端请求不包含sessionid，则为此客户端创建一个session并且生成一个与此session相关联的sessionid，并将这个sessionid在本次响应中返回给客户端保存。保存这个sessionid的方式可以采用 **cookie机制**，这样在交互过程中浏览器可以自动的按照规则把这个标识发给服务器；若浏览器禁用Cookie的话，可以通过 **URL重写机制** 将sessionid传回服务器。

3、Session 与 Cookie 的对比：

- **实现机制**: Session的实现常常依赖于Cookie机制，通过Cookie机制回传SessionID；
- **大小限制**: Cookie有大小限制并且浏览器对每个站点也有cookie的个数限制，Session没有大小限制，理论上只与服务器的内存大小有关；
- **安全性**: Cookie存在安全隐患，通过拦截或本地文件找得到cookie后可以进行攻击，而Session由于保存在服务器端，相对更加安全；
- **服务器资源消耗**: Session是保存在服务器端上会存在一段时间才会消失，如果session过多会增加服务器的压力。

4、Application:

Application (ServletContext)：与一个Web应用程序相对应，为应用程序提供了一个全局的状态，所有客户都可以使用该状态。

36. 滑动窗口机制

由发送方和接收方在三次握手阶段，互相将自己的最大可接收的数据量告诉对方。

也就是自己的数据接收缓冲池的大小。这样对方可以根据已发送的数据量来计算是否可以接着发送。在处理过程中，当接收缓冲池的大小发生变化时，要给对方发送更新窗口大小的通知。这就实现了流量的控制。

37. 常用的HTTP方法有哪些？

- **GET**: 用于请求访问已经被URI（统一资源标识符）识别的资源，可以通过URL传参给服务器。
- **POST**: 用于传输信息给服务器，主要功能与GET方法类似，但一般推荐使用POST方式。
- **PUT**: 传输文件，报文主体中包含文件内容，保存到对应URI位置。
- **HEAD**: 获得报文首部，与GET方法类似，只是不返回报文主体，一般用于验证URI是否有效。
- **DELETE**: 删除文件，与PUT方法相反，删除对应URI位置的文件。
- **OPTIONS**: 查询相应URI支持的HTTP方法。

38. 常见HTTP状态码

- 1、1xx (临时响应)
- 2、2xx (成功)
- 3、3xx (重定向) : 表示要完成请求需要进一步操作
- 4、4xx (错误) : 表示请求可能出错，妨碍了服务器的处理
- 5、5xx (服务器错误) : 表示服务器在尝试处理请求时发生内部错误

常见状态码：

- 200 (成功)
- 304 (未修改) : 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容
- 401 (未授权) : 请求要求身份验证
- 403 (禁止) : 服务器拒绝请求
- 404 (未找到) : 服务器找不到请求的网页

39. SQL 注入

SQL注入就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

1、SQL注入攻击的总体思路：

1. 寻找到SQL注入的位置
2. 判断服务器类型和后台数据库类型
3. 针对不通的服务器和数据库特点进行SQL注入攻击

2、SQL注入攻击实例：

比如，在一个登录界面，要求输入用户名和密码，可以这样输入实现免帐号登录：

```
用户名: 'or 1 = 1 --密 码: 复制代码
```

用户一旦点击登录，如若没有做特殊处理，那么这个非法用户就很得意的登陆进去了。这是为什么呢？下面我们分析一下：从理论上说，后台认证程序中会有如下的SQL语句：

```
String sql = "select * from user_table where username='“+userNmae+” ’ and password='“+password+” ’";
```

因此，当输入了上面的用户名和密码，上面的SQL语句变成：

```
SELECT * FROM user_table WHERE username=' or 1 = 1 - and password=''
```

分析上述SQL语句我们知道，`username=' or 1=1` 这个语句一定会成功；然后后面加两个`-`，这意味着注释，它将后面的语句注释，让他们不起作用。这样，上述语句永远都能正确执行，用户轻易骗过系统，获取合法身份。

3、应对方法：

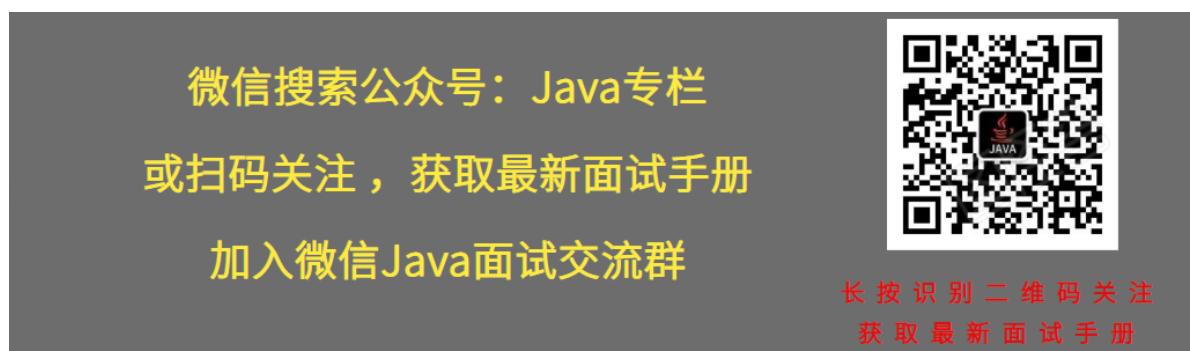
1.参数绑定：

使用预编译手段，绑定参数是最好的防SQL注入的方法。目前许多的ORM框架及JDBC等都实现了SQL预编译和参数绑定功能，攻击者的恶意SQL会被当做SQL的参数而不是SQL命令被执行。在mybatis的mapper文件中，对于传递的参数我们一般是使用#和

不能识别此Latex公式：来获取参数值。当使用#时，变量是占位符，就是一般我们使用javajdbc的PrepareStatement时的占位符，所有可以防止sql注入；当使用复制代码

时，变量就是直接追加在sql中，一般会有sql注入问题。

2.使用正则表达式过滤传入的参数



40. XSS 攻击

XSS是一种经常出现在web应用中的计算机安全漏洞，与SQL注入一起成为web中最主流的攻击方式。XSS是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到web页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。

1、XSS攻击的危害：

- 盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号
- 控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力
- 盗窃企业重要的具有商业价值的资料
- 非法转账
- 强制发送电子邮件
- 网站挂马
- 控制受害者机器向其它网站发起攻击

2、原因解析：

- **主要原因：**过于信任客户端提交的数据！
- **解决办法：**不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。
- **进一步分析细节：**客户端提交的数据本来就是应用所需要的，但是恶意攻击者利用网站对客户端提交数据的信任，在数据中插入一些符号以及javascript代码，那么这些数据将会成为应用代码中的。
微信搜索公众号：Java专栏，获取最新面试手册

一部分了，那么攻击者就可以肆无忌惮地展开攻击啦，因此我们绝不可以信任任何客户端提交的数据！！！

3、XSS 攻击分类：

- 1. 反射性 XSS 攻击（非持久性 XSS 攻击）：

漏洞产生的原因是攻击者注入的数据反映在响应中。一个典型的非持久性XSS攻击包含一个带XSS攻击向量的链接(即每次攻击需要用户的点击)，例如，正常发送消息：

```
http://www.test.com/message.php?send=Hello,world! 复制代码
```

接收者将会接收信息并显示Hello,World；但是，非正常发送消息：

```
http://www.test.com/message.php?send=<script>alert('foolish!')</script>! 复制代码
```

接收者接收消息显示的时候将会弹出警告窗口！

- 2. 持久性XSS攻击（留言板场景）：

XSS攻击向量(一般指XSS攻击代码)存储在网站数据库，当一个页面被用户打开的时候执行。也就是说，每当用户使用浏览器打开指定页面时，脚本便执行。与非持久性XSS攻击相比，持久性XSS攻击危害性更大。从名字就可以了解到，持久性XSS攻击就是将攻击代码存入数据库中，然后客户端打开时就执行这些攻击代码。

例如，留言板表单中的表单域：

```
<input type="text" name="content" value="这里是用户填写的数据"> 复制代码
```

正常操作流程是：用户提交相应留言信息——将数据存储到数据库——其他用户访问留言板，应用去数据并显示；而非正常操作流程是攻击者在value填写：

```
<script>alert('foolish!'); </script> <!--或者html其他标签（破坏样式）、一段攻击型代码--> 复制代码
```

并将数据提交、存储到数据库中；当其他用户取出数据显示的时候，将会执行这些攻击性代码。

4、修复漏洞方针：

漏洞产生的根本原因是 **太相信用户提交的数据，对用户所提交的数据过滤不足所导致的**，因此解决方案也应该从这个方面入手，具体方案包括：

- 将重要的cookie标记为http only，这样的话Javascript 中的document.cookie语句就不能获取到cookie了（如果在cookie中设置了HttpOnly属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击）；
- 表单数据规定值的类型，例如：年龄应为只能为int、name只能为字母数字组合。。。。
- 对数据进行Html Encode 处理
- 过滤或移除特殊的Html标签，例如：`<script>`, `<iframe>`, `< for <, > for>`, `" for`
- 过滤JavaScript 事件的标签，例如 “onclick=”, “onfocus” 等等。

需要注意的是，在有些应用中是允许html标签出现的，甚至是javascript代码出现。因此，我们在过滤数据的时候需要仔细分析哪些数据是有特殊要求（例如输出需要html代码、javascript代码拼接、或者此表单直接允许使用等等），然后区别处理！

41. OSI 网络体系结构与 TCP/IP 协议模型

OSI 是一个理论上的网络通信模型，而 TCP/IP 则是实际上的网络通信标准。但是，它们的初衷是一样的，都是为了使得两台计算机能够像两个知心朋友那样能够互相准确理解对方的意思并做出优雅的回应。现在，我们对 OSI 七层模型的各层进行简要的介绍：

1、物理层

参考模型的最低层，也是OSI模型的第一层，实现了相邻计算机节点之间比特流的透明传送，并尽可能地屏蔽掉具体传输介质和物理设备的差异，使其上层(数据链路层)不必关心网络的具体传输介质。

2、数据链路层 (data link layer)

接收来自物理层的位流形式的数据，并封装成帧，传送到上一层；同样，也将来自上层的数据帧，拆装为位流形式的数据转发到物理层。这一层在物理层提供的比特流的基础上，通过差错控制、流量控制方法，使有差错的物理线路变为无差错的数据链路，即提供可靠的通过物理介质传输数据的方法。

3、网络层

将网络地址翻译成对应的物理地址，并通过路由选择算法为分组通过通信子网选择最适当的路径。

4、传输层 (transport layer)

在源端与目的端之间提供可靠的透明数据传输，使上层服务用户不必关系通信子网的实现细节。在协议栈中，传输层位于网络层之上，传输层协议为不同主机上运行的进程提供逻辑通信，而网络层协议为不同主机提供逻辑通信，如下图所示。

实际上，网络层可以看作是传输层的一部分，其为传输层提供服务。但对于终端系统而言，网络层对它们而言是透明的，它们知道传输层的存在，也就是说，在逻辑上它们认为是传输层为它们提供了端对端的通信，这也是分层思想的妙处。

5、会话层 (Session Layer)

会话层是OSI模型的第五层，是用户应用程序和网络之间的接口，负责在网络中的两节点之间建立、维持和终止通信。

6、表示层 (Presentation Layer)：数据的编码，压缩和解压缩，数据的加密和解密

表示层是OSI模型的第六层，它对来自应用层的命令和数据进行解释，以确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。

7、应用层 (Application layer)：为用户的应用进程提供网络通信服务

42. 网络层的 ARP 协议工作原理？

地址解析协议(ARP)是通过解析网路层地址来找寻数据链路层地址的一个在网络协议包中极其重要的网络传输协议。

网络层的ARP协议完成了IP地址与物理地址的映射。首先，每台主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址的对应关系。当源主机需要将一个数据包要发送到目的主机时，会首先检查自己ARP列表中是否存在该IP地址对应的MAC地址：如果有，就直接将数据包发送到这个MAC地址；如果没有，就向本地网段发起一个ARP请求的广播包，查询此目的主机对应的MAC地址。此ARP请求数据包里包括源主机的IP地址、硬件地址、以及目的主机的IP地址。网络中所有的主机收到这个ARP请求后，会检查数据包中的目的IP是否和自己的IP地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的MAC地址和IP地址添加到自己的ARP列表中，如果ARP表中已经存在该IP的信息，则将其覆盖，然后给源主机发送一个ARP响应数据包，告诉对方自己是它需要查找的MAC地址；源主机收到这个ARP响应数据包后，将得到的目的主机的IP地址和MAC地址添加到自己的ARP列表中，并利用此信息开始数据的传输。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

43. IP地址的分类

整个的因特网就是一个单一的、抽象的网络。IP地址就是给因特网上的每一个主机（或路由器）的每一个接口分配一个在全世界范围是唯一的32位标识符，它是一个逻辑地址，用以屏蔽掉物理地址的差异。IP地址编址方案将IP地址空间划分为A、B、C、D、E五类，其中A、B、C是基本类，D、E类作为多播和保留使用，为特殊地址。

每个IP地址包括两个标识码（ID），即网络ID和主机ID。同一个物理网络上的所有主机都使用同一个网络ID，网络上的一个主机（包括网络上工作站、服务器和路由器等）有一个主机ID与其对应。A~E类地址的特点如下：

- A类地址：以0开头，第一个字节范围：0~127；
- B类地址：以10开头，第一个字节范围：128~191；
- C类地址：以110开头，第一个字节范围：192~223；
- D类地址：以1110开头，第一个字节范围为224~239；
- E类地址：以1111开头，保留地址

1、A类地址：1字节的网络地址 + 3字节主机地址，网络地址的最高位必须是“0”

一个A类IP地址是指，在IP地址的四段号码中，第一段号码为网络号码，剩下的三段号码为本地计算机的号码。如果用二进制表示IP地址的话，A类IP地址就由1字节的网络地址和3字节主机地址组成，网络地址的最高位必须是“0”。A类IP地址中网络的标识长度为8位，主机标识的长度为24位，A类网络地址数量较少，有126个网络，每个网络可以容纳主机数达1600多万台。

A类IP地址的地址范围1.0.0.0到127.255.255.255（二进制表示为：00000001 00000000 00000000 00000000 - 01111110 11111111 11111111 11111111），最后一个地址是广播地址。A类IP地址的子网掩码为255.0.0.0，每个网络支持的最大主机数为256的3次方-2=16777214台。

2、B类地址：2字节的网络地址 + 2字节主机地址，网络地址的最高位必须是“10”

一个B类IP地址是指，在IP地址的四段号码中，前两段号码为网络号码。如果用二进制表示IP地址的话，B类IP地址就由2字节的网络地址和2字节主机地址组成，网络地址的最高位必须是“10”。B类IP地址中网络的标识长度为16位，主机标识的长度为16位，B类网络地址适用于中等规模的网络，有16384个网络，每个网络所能容纳的计算机数为6万多台。

B类IP地址的地址范围128.0.0.0-191.255.255.255（二进制表示为：10000000 00000000 00000000 00000000 - 10111111 11111111 11111111 11111111），最后一个地址是广播地址。B类IP地址的子网掩码为255.255.0.0，每个网络支持的最大主机数为256的2次方-2=65534台。

3、C类地址：3字节的网络地址 + 1字节主机地址，网络地址的最高位必须是“110”

一个C类IP地址是指，在IP地址的四段号码中，前三段号码为网络号码，剩下的一段号码为本地计算机的号码。如果用二进制表示IP地址的话，C类IP地址就由3字节的网络地址和1字节主机地址组成，网络地址的最高位必须是“110”。C类IP地址中网络的标识长度为24位，主机标识的长度为8位，C类网络地址数量较多，有209万余个网络。适用于小规模的局域网络，每个网络最多只能包含254台计算机。

C类IP地址的地址范围192.0.0.0-223.255.255.255（二进制表示为：11000000 00000000 00000000 00000000 - 11011111 11111111 11111111 11111111），C类IP地址的子网掩码为255.255.255.0，每个网络支持的最大主机数为256-2=254台。

4、D类地址：多播地址，用于1对多通信，最高位必须是“1110”

D类IP地址在历史上被叫做多播地址(multicast address)，即组播地址。在以太网中，多播地址命名了一组应该在这个网络中应用接收到一个分组的站点。多播地址的最高位必须是“1110”，范围从224.0.0.0到239.255.255.255。

5、E类地址：为保留地址，最高位必须是“1111”

44. IP地址与物理地址

物理地址是数据链路层和物理层使用的地址，IP地址是网络层和以上各层使用的地址，是一种逻辑地址，其中ARP协议用于IP地址与物理地址的对应。

45. 影响网络传输的因素有哪些？

将一份数据从一个地方正确地传输到另一个地方所需要的时间我们称之为响应时间。影响这个响应时间的因素有很多。

1、网络带宽：

所谓带宽就是一条物理链路在 1s 内能够传输的最大比特数，注意这里是比特（bit）而不是字节数，也就是 b/s。网络带宽肯定是影响数据传输的一个关键环节，因为在当前的网络环境中，平均网络带宽只有 1.7 MB/s 左右。

2、传输距离：

也就是数据在光纤中要走的距离，虽然光的传播速度很快，但也是有时间的，由于数据在光纤中的移动并不是走直线的，会有一个折射率，所以大概是光的 2/3，这个时间也就是我们通常所说的传输延时。传输延时是一个无法避免的问题，例如，你要给在杭州和青岛的两个机房的一个数据库进行同步数据操作，那么必定会存在约 30ms 的一个延时。

3、TCP 拥塞控制：

我们知道 TCP 传输是一个“停-等-停-等”的协议，传输方和接受方的步调要一致，要达到步调一致就要通过拥塞控制来调节。TCP 在传输时会设定一个“窗口”，这个窗口的大小是由带宽和 RTT (Round-Trip Time, 数据在两端的来回时间，也就是响应时间) 决定的。计算的公式是带宽 (b/s) \times RTT (s)。通过这个值就可以得出理论上最优的 TCP 缓冲区的大小。Linux 2.4 已经可以自动地调整发送端的缓冲区的大小，而到 Linux 2.6.7 时接收端也可以自动调整了。

46. 什么是对称加密与非对称加密

对称密钥加密是指加密和解密使用同一个密钥的方式，这种方式存在的最大问题就是密钥发送问题，即如何安全地将密钥发给对方；

而非对称加密是指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息后，使用自己的私钥进行解密。由于非对称加密的方式不需要发送用来解密的私钥，所以可以保证安全性；但是和对称加密比起来，非常的慢

47. 什么是Cookie

cookie是由Web服务器保存在用户浏览器上的文件（key-value格式），可以包含用户相关的信息。客户端向服务器发起请求，就提取浏览器中的用户信息由http发送给服务器

48. 什么是Session

session 是浏览器和服务器会话过程中，服务器会分配的一块储存空间给session。

服务器默认为客户浏览器的cookie中设置 sessionid，这个sessionid就和cookie对应，浏览器在向服务器请求过程中传输的cookie 包含 sessionid，服务器根据传输cookie 中的 sessionid 获取出会话中存储的信息，然后确定会话的身份信息。

49. Cookie和Session对于HTTP有什么用？

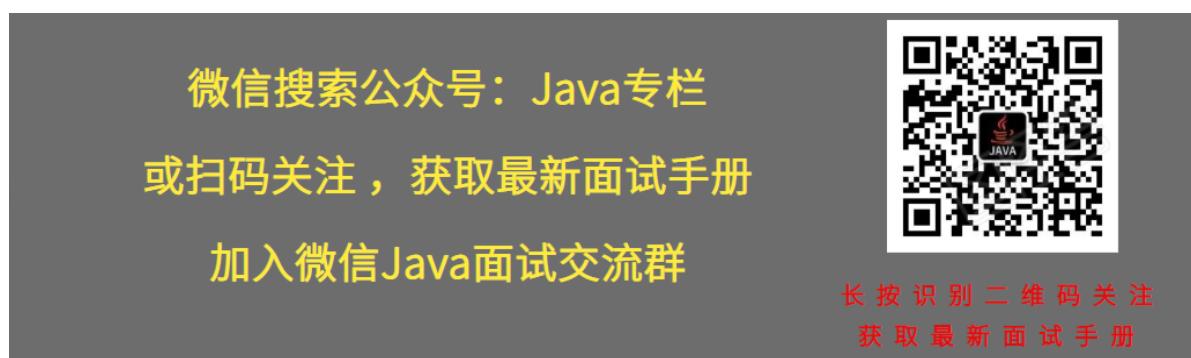
HTTP协议本身是无法判断用户身份。所以需要cookie或者session

50. Cookie与Session区别

- 1、Cookie数据存放在客户端上，安全性较差，Session数据放在服务器上，安全性相对更高
- 2、单个cookie保存的数据不能超过4K，session无此限制
- 3、session一定时间内保存在服务器上，当访问增多，占用服务器性能，考虑到服务器性能方面，应当使用cookie。

数据库

MySQL面试题



1. MySQL中的varchar和char有什么区别.

char是一个定长字段,假如申请了 char(10) 的空间,那么无论实际存储多少内容.该字段都占用10个字符,而varchar是变长的,也就是说申请的只是最大长度,占用的空间为实际字符长度+1,最后一个字符存储使用了多少的空间.

在检索效率上来讲,char > varchar,因此在使用中,如果确定某个字段的值的长度,可以使用char,否则应该尽量使用varchar.例如存储用户MD5加密后的密码,则应该使用char.

2. varchar(10)和int(10)代表什么含义?

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

3. MySQL中varchar与char的区别以及varchar(50)中的50代表的涵义

- 1、varchar与char的区别char是一种固定长度的类型, varchar则是一种可变长度的类型
- 2、varchar(50)中50的涵义最多存放50个字符, varchar(50)和(200)存储hello所占空间一样, 但后者在排序时会消耗更多内存, 因为order by col采用fixed_length计算col长度(memory引擎也一样)
- 3、int (20) 中20的涵义是指显示字符的长度但要加参数的, 最大为255, 比如它是记录行数的id,插入10笔资料, 它就显示000000000001 ~~~00000000010, 当字符的位数超过11, 它也只显示11位, 如果你没有加那个让它未满11位就前面加0的参数, 它不会在前面加020表示最大显示宽度为20, 但仍占4字节存储, 存储范围不变;
- 4、mysql为什么这么设计对大多数应用没有意义, 只是规定一些工具用来显示字符的个数; int(1)和int(20)存储和计算均一样;

4. innodb的事务与日志的实现方式

- 1、有多少种日志; 错误日志: 记录出错信息, 也记录一些警告信息或者正确的信息。查询日志: 记录所有对数据库请求的信息, 不论这些请求是否得到了正确的执行。慢查询日志: 设置一个阈值, 将运行时间超过该值的所有SQL语句都记录到慢查询的日志文件中。二进制日志: 记录对数据库执行更改的所有操作。中继日志: 事务日志:
- 2、事物的4种隔离级别隔离级别读未提交(RU)读已提交(RC)可重复读(RR)串行
- 3、事物是如何通过日志来实现的, 说得越深入越好。事物日志是通过redo和innodb的存储引擎日志缓冲 (Innodb log buffer) 来实现的, 当开始一个事务的时候, 会记录该事务的lsn(log sequence number)号; 当事务执行时, 会往InnoDB存储引擎的日志的日志缓存里面插入事务日志; 当事务提交时, 必须将存储引擎的日志缓冲写入磁盘 (通过innodb_flush_log_at_trx_commit来控制), 也就是写数据前, 需要先写日志。这种方式称为“预写日志方式”

5. MySQL的binlog有几种录入格式?分别有什么区别?**

有三种格式,statement,row和mixed.

- statement模式下,记录单元为语句.即每一个sql造成的影响会记录.由于sql的执行是有上下文的,因此在保存的时候需要保存相关的信息,同时还有一些使用了函数之类的语句无法被记录复制.
- row级别下,记录单元为每一行的改动,基本是可以全部记下来但是由于很多操作,会导致大量行的改动(比如alter table),因此这种模式的文件保存的信息太多,日志量太大.
- mixed. 一种折中的方案,普通操作使用statement记录,当无法使用statement的时候使用row.

此外,新版的MySQL中对row级别也做了一些优化,当表结构发生变化的时候,会记录语句而不是逐行记录.

6. 超大分页怎么处理?**

超大的分页一般从两个方向上来解决.

- 数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于 `select * from table where age > 20 limit 1000000,10` 这种查询其实也是有可以优化的余地的. 这条语句需要 load 1000000 数据然后基本上全部丢弃,只取 10 条当然比较慢. 当时我们可以修改为 `select * from table where id in (select id from table where age > 20 limit 1000000,10)`. 这样虽然也 load 了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快. 同时如果 ID 连续的好,我们还可以 `select * from table where id > 1000000 limit 10`, 效率也是不错的,优化的可能性有许多种,但是核心思想都一样,就是减少 load 的数据.
- 从需求的角度减少这种请求....主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止 ID 泄露且连续被人恶意攻击.

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至 redis 等 k-V 数据库中,直接返回即可.

在阿里巴巴《Java 开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

7. 【推荐】利用延迟关联或者子查询优化超多分页场景。

说明 : MySQL 并不是跳过 offset 行,而是取 offset+N 行,然后返回放弃前 offset 行,返回 N 行,那当 offset 特别大的时候,效率就非常的低下,要么控制返回的总页数,要么对超过特定阈值的页数进行 SQL 改写。

正例 : 先快速定位需要获取的 id 段,然后再关联 :

```
SELECT a.* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20 ) b where a.id=b.id
```

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

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

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

所以优化也是针对这三个方向来的,

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

8. 上面提到横向分表和纵向分表,可以分别举一个适合他们的例子吗?

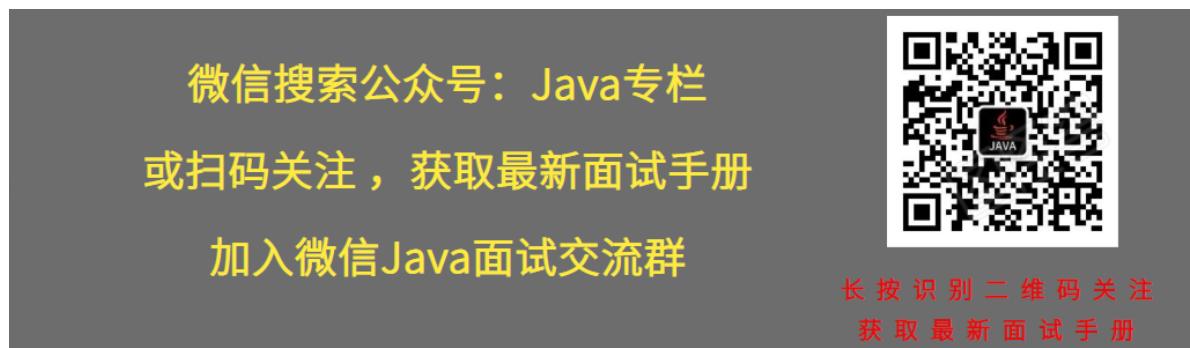
横向分表是按行分表.假设我们有一张用户表,主键是自增ID且同时是用户的ID.数据量较大,有1亿多条,那么此时放在一张表里的查询效果就不太理想.我们可以根据主键ID进行分表,无论是按尾号分,或者按ID的区间分都是可以的.假设按照尾号0-99分为100个表,那么每张表中的数据就仅有100w.这时的查询效率无疑是能够满足要求的.

纵向分表是按列分表.假设我们现在有一张文章表.包含字段 id-摘要-内容 .而系统中的展示形式是刷新出一个列表,列表中仅包含标题和摘要,当用户点击某篇文章进入详情时才需要正文内容.此时,如果数据量大,将内容这个很大且不经常使用的列放在一起会拖慢原表的查询速度.我们可以将上面的表分为两张. `id-摘要`, `id-内容`.当用户点击详情,那主键再来取一次内容即可.而增加的存储量只是很小的主键字段.代价很小.当然,分表其实和业务的关联度很高,在分表之前一定要做好调研以及benchmark.不要按照自己的猜想盲目操作.

9. 什么是存储过程？有哪些优缺点？

存储过程是一些预编译的SQL语句。1、更加直白的理解：存储过程可以说是一个记录集，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。2、存储过程是一个预编译的代码块，执行效率比较高，一个存储过程替代大量T_SQL语句，可以降低网络通信量，提高通信速率，可以一定程度上确保数据安全

但是在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java开发手册》中禁止使用存储过程,我个人的理解是,在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁,在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好.



10. 说一说三个范式

第一范式: 每个列都不可以再拆分. 第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分. 第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

在设计数据库结构的时候,要尽量遵守三范式,如果不遵守,必须有足够的理由.比如性能.事实上我们经常会为了性能而妥协数据库的设计.

11. MySQL的复制原理以及流程

基本原理流程，3个线程以及之间的关联；

- 1、**主**: binlog线程——记录下所有改变了数据库数据的语句，放进master上的binlog中；
- 2、**从**: io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进自己的relay log 中；
- 3、**从**: sql执行线程——执行relay log中的语句；

12. MySQL由哪些部分组成, 分别用来做什么

1、Server

- 连接器: 管理连接, 权限验证.
- 分析器: 词法分析, 语法分析.
- 优化器: 执行计划生成, 索引的选择.
- 执行器: 操作存储引擎, 返回执行结果.

2、存储引擎: 存储数据, 提供读写接口.

13. 如果一个表有一列定义为TIMESTAMP, 将发生什么?

每当行被更改时, 时间戳字段将获取当前时间戳。列设置为 AUTO INCREMENT 时, 如果在表中达到最大值, 会发生什么情况? 它会停止递增, 任何进一步的插入都将产生错误, 因为密钥已被使用。

怎样才能找出最后一次插入时分配了哪个自动增量? LAST_INSERT_ID 将返回由 Auto_increment 分配的最后一个值, 并且不需要指定表名称。

14. MySQL 里记录货币用什么字段类型好

NUMERIC 和 DECIMAL 类型被 MySQL 实现为同样的类型, 这在 SQL92 标准允许。他们被用于保存值, 该值的准确精度是极其重要的值, 例如与金钱有关的数据。当声明一个类是这些类型之一时, 精度和规模的能被(并且通常是)指定。

例如:

```
salary DECIMAL(9,2)
```

在这个例子中, 9(precision)代表将被用于存储值的总的小数位数, 而 2(scale)代表将被用于存储小数点后的位数。因此, 在这种情况下, 能被存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

15. MySQL 数据库作发布系统的存储, 一天五万条以上的增量, 预计运维三年, 怎么优化?

- 1、设计良好的数据库结构, 允许部分数据冗余, 尽量避免 join 查询, 提高效率。
- 2、选择合适的表字段数据类型和存储引擎, 适当的添加索引。
- 3、MySQL 库主从读写分离。
- 4、找规律分表, 减少单表中的数据量提高查询速度。5、添加缓存机制, 比如 memcached, apc 等。
- 5、不经常改动的页面, 生成静态页面。
- 6、书写高效率的 SQL。比如 SELECT * FROM TABLE 改为 SELECT field_1, field_2, field_3 FROM TABLE.

16. 优化数据库的方法

- 1、选取最适用的字段属性，尽可能减少定义字段宽度，尽量把字段设置 NOTNULL，例如‘省份’、‘性别’最好适用 ENUM
- 2、使用连接(JOIN)来代替子查询
- 3、适用联合(UNION)来代替手动创建的临时表
- 4、事务处理
- 5、锁定表、优化事务处理
- 6、适用外键，优化锁定表
- 7、建立索引
- 8、优化查询语句

17. 简单描述 MySQL 中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）

索引是一种特殊的文件(InnoDB 数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。

普通索引(由关键字 KEY 或 INDEX 定义的索引)的唯一任务是加快对数据的访问速度。

普通索引允许被索引的数据列包含重复的值。如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。也就是说，唯一索引可以保证数据记录的唯一性。

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

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

索引可以极大的提高数据的查询速度，但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件。

18. SQL 注入漏洞产生的原因？如何防止？

SQL 注入产生的原因：程序开发过程中不注意规范书写 sql 语句和对特殊字符进行过滤，导致客户端可以通过全局变量 POST 和 GET 提交一些 sql 语句正常执行。防止 SQL 注入的方式：

开启配置文件中的 magic_quotes_gpc 和 magic_quotes_runtime 设置

执行 sql 语句时使用 addslashes 进行 sql 语句转换Sql 语句书写尽量不要省略双引号和单引号。

过滤掉 sql 语句中的一些关键词： update、insert、delete、select、*。

提高数据库表和字段的命名技巧，对一些重要的字段根据程序的特点命名，取不易被猜到的。

19. 存储时期

Datetime:

以 YYYY-MM-DD HH:MM:SS 格式存储时期时间， 精确到秒， 占用 8 个字节得存储空间， datetime 类型与时区无关

Timestamp:

以时间戳格式存储， 占用 4 个字节， 范围小 1970-1-1 到 2038-1-19， 显示依赖于所指定得时区， 默认在第一个列行的数据修改时可以自动得修改timestamp 列得值

Date (生日) :

占用得字节数比使用字符串.datetime.int 储存要少， 使用 date 只需要 3 个字节， 存储日期月份， 还可以利用日期时间函数进行日期间得计算Time:存储时间部分得数据

注意:不要使用字符串类型来存储日期时间数据（通常比字符串占用得储存空间小，在进行查找过滤可以利用日期得函数）使用 int 存储日期时间不如使用 timestamp 类型

20. 解释 MySQL 外连接、内连接与自连接的区别

先说什么是交叉连接:

交叉连接又叫笛卡尔积， 它是指不使用任何条件， 直接将一个表的所有记录和另一个表中的所有记录一一匹配。

内连接 则是只有条件的交叉连接， 根据某个条件筛选出符合条件的记录， 不符合条件的记录不会出现在结果集中， 即内连接只连接匹配的行。

外连接 其结果集中不仅包含符合连接条件的行， 而且还会包括左表、右表或两个表中的所有数据行，

这三种情况依次称之为左外连接， 右外连接， 和全外连接。左外连接， 也称左连接， 左表为主表， 左表中的所有记录都会出现在结果集中， 对于那些在右表中并没有匹配的记录， 仍然要显示， 右边对应的那些字段值以NULL 来填充。右外连接， 也称右连接， 右表为主表， 右表中的所有记录都会出现在结果集中。左连接和右连接可以互换， MySQL 目前还不支持全外连接。

21. 存储引擎常用命令

查看MySQL提供的所有存储引擎

```
mysql> show engines;
```

```
Type 'help,' or '\h' for help. Type '\c' to clear the current input statement.
mysql> show engines;
+----+-----+-----+-----+-----+-----+
| Engine | Support | Comment | Transactions | XA | Savepoints |
+----+-----+-----+-----+-----+-----+
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES | YES | YES |
| MRG_MYISAM | YES | Collection of identical MyISAM tables | NO | NO | NO |
| MEMORY | YES | Hash based, stored in memory, useful for temporary tables | NO | NO | NO |
| BLACKHOLE | YES | /dev/null storage engine (anything you write to it disappears) | NO | NO | NO |
| MyISAM | YES | MyISAM storage engine | NO | NO | NO |
| CSV | YES | CSV storage engine | NO | NO | NO |
| ARCHIVE | YES | Archive storage engine | NO | NO | NO |
| PERFORMANCE_SCHEMA | YES | Performance Schema | NO | NO | NO |
| FEDERATED | NO | Federated MySQL storage engine | NULL | NULL | NULL |
+----+-----+-----+-----+-----+-----+
```

从上图我们可以看出 MySQL 当前默认的存储引擎是InnoDB，并且在5.7版本所有的存储引擎中只有 InnoDB 是事务性存储引擎，也就是说只有 InnoDB 支持事务。

查看MySQL当前默认的存储引擎

我们也可以通过下面的命令查看默认的存储引擎。

```
mysql> show variables like '%storage_engine%';
```

查看表的存储引擎

```
show table status like "table_name" ;
```

```
mysql> show table status like "tb_base";
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Name | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time |
| Update_time | Check_time | Collation | Checksum | Create_options | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| tb_base | InnoDB | 10 | Dynamic | 0 | 0 | 16384 | 0 | 0 | 0 | 0 | 2 | 2019-05-05 15:58:24 |
| NULL | NULL | utf8_general_ci | NULL | | | | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

22. MySQL支持哪些存储引擎?

MySQL支持多种存储引擎,比如InnoDB,MyISAM,Memory,Archive等等.在大多数的情况下,直接选择使用 InnoDB引擎都是最合适的,InnoDB也是MySQL的默认存储引擎.

23. InnoDB和MyISAM有什么区别?

- InnoDB支持事物，而MyISAM不支持事物
- InnoDB支持行级锁，而MyISAM支持表级锁
- InnoDB支持MVCC, 而MyISAM不支持
- InnoDB支持外键，而MyISAM不支持
- InnoDB不支持全文索引，而MyISAM支持。

	MyISAM	Innodb
文件格式	数据和索引是分别存储的, 数据 .MYD , 索引 .MYI	数据和索引是集中存储的, .ibd
文件能否移动	能, 一张表就对应 .frm 、 .MYD 、 .MYI 3个文件	否, 因为关联的还有 data 下的其它文件
记录存储顺序	按记录插入顺序保存	按主键大小有序插入
空间碎片 (删除记录并 flush table 表名 之后, 表文件大小不变)	产生。定时整理: 使用命令 optimize table 表名 实现	不产生
事务	不支持	支持
外键	不支持	支持
锁支持 (锁是避免资源争用的一个机制, MySQL锁对用户几乎是透明的)	表级锁定	行级锁定、表级锁定, 锁定力度小并发能力高

24. myisamchk 是用来做什么的?

它用来压缩 MyISAM 表, 这减少了磁盘或内存使用。

MyISAM Static 和 MyISAM Dynamic 有什么区别?

在 MyISAM Static 上的所有字段有固定宽度。动态 MyISAM 表将具有像 TEXT, BLOB 等字段, 以适应不同长度的数据类型。

MyISAM Static 在受损情况下更容易恢复。

25. 为什么要尽量设定一个主键?**

主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的ID列作为主键.设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全.

26. 主键使用自增ID还是UUID?

推荐使用自增ID,不要使用UUID.

因为在InnoDB存储引擎中,主键索引是作为聚簇索引存在的,也就是说,主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序),如果主键索引是自增ID,那么只需要不断向后排列即可,如果是UUID,由于到来的ID与原来的大小不确定,会造成非常多的数据插入,数据移动,然后导致产生很多的内存碎片,进而造成插入性能的下降.

总之,在数据量大一些的情况下,用自增主键性能会好一些.

图片来源于《高性能MySQL》: 其中默认后缀为使用自增ID,_uuid为使用UUID为主键的测试,测试了插入100w行和300w行的性能.

表 5-1：向InnoDB表插入数据的测试结果

表名	行数	时间 (秒)	索引大小 (MB)
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

关于主键是聚簇索引,如果没有主键,InnoDB会选择一个唯一键来作为聚簇索引,如果没有唯一键,会生成一个隐式的主键.

If you define a PRIMARY KEY on your table, InnoDB uses it as the clustered index.

If you do not define a PRIMARY KEY for your table, MySQL picks the first UNIQUE index that has only NOT NULL columns as the primary key and InnoDB uses it as the clustered index.

27. 字段为什么要要求定义not null?

MySQL官网这样介绍:

NULL columns require additional space in the rowto record whether their values are NULL.
For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte.

null值会占用更多的字节,且会在程序中造成很多与预期不符的情况.

28. 如果要存储用户的密码散列,应该使用什么字段进行存储?

密码散列,盐,用户身份证号等固定长度的字符串应该使用char而不是varchar来存储,这样可以节省空间且提高检索效率.

29. 什么是索引?

索引是一种数据结构,可以帮助我们快速的进行数据的查找.

30. 索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关,在MySQL中使用较多的索引有Hash索引,B+树索引等,而我们经常使用的InnoDB存储引擎的默认索引实现为:B+树索引.

31. 唯一索引比普通索引快吗,为什么

唯一索引不一定比普通索引快,还可能慢.

1、查询时,在未使用 `limit 1` 的情况下,在匹配到一条数据后,唯一索引即返回,普通索引会继续匹配下一条数据,发现不匹配后返回.如此看来唯一索引少了一次匹配,但实际上这个消耗微乎其微.

2、更新时,这个情况就比较复杂了.普通索引将记录放到 change buffer 中语句就执行完毕了.而对唯一索引而言,它必须要校验唯一性,因此,必须将数据页读入内存确定没有冲突,然后才能继续操作.对于**写多读少**的情况,普通索引利用 change buffer 有效减少了对磁盘的访问次数,因此普通索引性能要高于唯一索引.

32. 索引的优缺点

优点

- 提高数据检索的效率,降低数据库的 IO 成本。
- 通过索引列对数据进行排序,降低数据排序的成本,降低了 CPU 的消耗。

缺点

- 虽然索引大大提高了查询速度,同时却会降低更新表的速度,如对表进行 INSERT、UPDATE 和 DELETE。因为更新表时,MySQL 不仅要保存数据,还要保存一下索引文件每次更新添加了索引列的字段,都会调整因为更新所带来的键值变化后的索引信息。
- 实际上索引也是一张表,该表保存了主键与索引字段,并指向实体表的记录,所以索引列也是要占用空间的。

33. 做过哪些MySQL索引相关优化

- 尽量使用主键查询:聚簇索引上存储了全部数据,相比普通索引查询,减少了回表的消耗。
- MySQL5.6之后引入了索引下推优化,通过适当的使用联合索引,减少回表判断的消耗。
- 若频繁查询某一列数据,可以考虑利用覆盖索引避免回表。
- 联合索引将高频字段放在最左边。

34. 怎么看到为表格定义的所有索引?

索引是通过以下方式为表格定义的:

```
SHOW INDEX FROM ;
```

35. 索引分类

单值索引:即一个索引只包含单个列,一个表可以有多个单列索引

- 建表时,加上 `key(列名)` 指定
- 单独创建, `create index 索引名 on 表名(列名)`
- 单独创建, `alter table 表名 add index 索引名(列名)`

唯一索引:索引列的值必须唯一,但允许有 null 且 null 可以出现多次

- 建表时,加上 `unique(列名)` 指定
- 单独创建, `create unique index idx_表名_列名 on 表名(列名)`
- 单独创建, `alter table 表名 add unique 索引名(列名)`

主键索引:设定为主键后数据库会自动建立索引,innodb 为聚簇索引,值必须唯一且不能为 null

- 建表时,加上 `primary key(列名)` 指定

复合索引：即一个索引包含多个列

- 建表时，加上 `key(列名列表)` 指定
- 单独创建，`create index 索引名 on 表名(列名列表)`
- 单独创建，`alter table 表名 add index 索引名(列名列表)`

36. 什么情况下设置了索引但无法使用

- 1、以“%”开头的 LIKE 语句，模糊匹配
- 2、OR 语句前后没有同时使用索引
- 3、数据类型出现隐式转化（如 varchar 不加单引号的话可能会自动转换为 int 型）

37. B-Tree 和 B+Tree

区别

1. B-Tree 的关键字和记录是放在一起的，叶子节点可以看作外部节点，不包含任何信息；B+Tree 的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。
2. 在 B-Tree 中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而 B+Tree 中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。从这个角度看 B-Tree 的性能好像要比 B+Tree 好，而在实际应用中却是 B+Tree 的性能要好些。因为 B+Tree 的非叶子节点不存放实际的数据，这样每个节点可容纳的元素个数比 B-Tree 多，树高比 B-Tree 小，这样带来的好处是减少磁盘访问次数。尽管 B+Tree 找到一个记录所需的比较次数要比 B-Tree 多，但是每次磁盘访问的时间相当于成百上千次内存比较的时间，因此实际中 B+Tree 的性能可能还会好些，而且 B+Tree 的叶子节点使用指针连接在一起，方便顺序遍历（例如查看一个目录下的所有文件，一个表中的所有记录等），这也是很多数据库和文件系统使用 B+Tree 的缘故。

为什么 B+Tree 比 B-Tree 更适合实际应用中操作系统的文件索引和数据库索引？

1. B+Tree 的磁盘读写代价更低

B+Tree 的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B-Tree 更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

1. B+Tree 的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

38. Hash索引和B+树所有有什么区别或者说优劣呢?**

首先要知道Hash索引和B+树索引的底层实现原理:

hash索引底层就是hash表,进行查找时,调用一次hash函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

- hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而B+树的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

- hash索引不支持使用索引进行排序,原理同上.
- hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测.**AAAA**和**AAAAB**的索引没有相关性.
- hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.
- hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

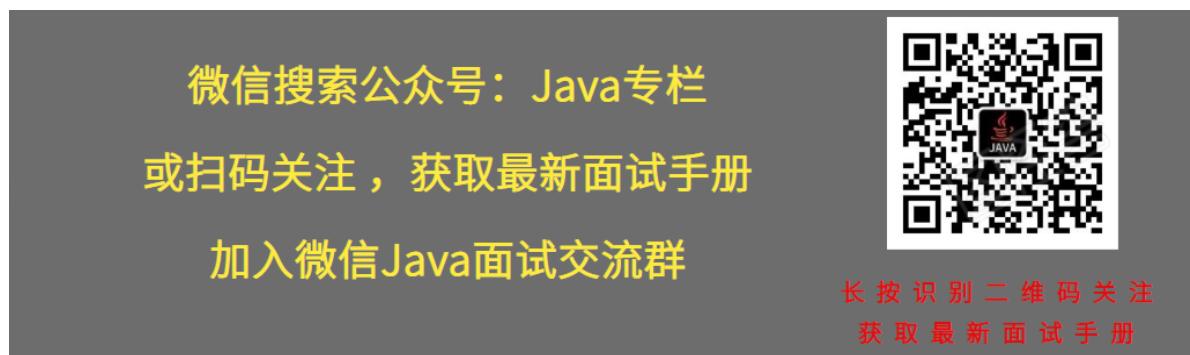
因此,在大多数情况下,直接选择B+树索引可以获得稳定且较好的查询速度.而不需要使用hash索引.

39. 为什么用 B+ 树做索引而不用哈希表做索引?

1、哈希表是把索引字段映射成对应的哈希码然后再存放在对应的位置,这样的话,如果我们要进行模糊查找的话,显然哈希表这种结构是不支持的,只能遍历这个表.而B+树则可以通过最左前缀原则快速找到对应的数据。

2、如果我们要进行范围查找,例如查找ID为100 ~ 400的人,哈希表同样不支持,只能遍历全表。

3、索引字段通过哈希映射成哈希码,如果很多字段都刚好映射到相同值的哈希码的话,那么形成的索引结构将会是一条很长的链表,这样的话,查找的时间就会大大增加。



40. 上面提到了B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据,什么是聚簇索引?

在B+树的索引中,叶子节点可能存储了当前的key值,也可能存储了当前的key值以及整行的数据,这就是聚簇索引和非聚簇索引. 在InnoDB中,只有主键索引是聚簇索引,如果没有主键,则挑选一个唯一键建立聚簇索引. 如果没有唯一键,则隐式的生成一个键来建立聚簇索引.

当查询使用聚簇索引时,在对应的叶子节点,可以获取到整行数据,因此不用再次进行回表查询.

41. 非聚簇索引一定会回表查询吗?**

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行 `select age from employee where age < 20` 的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

42. 在建立索引的时候,都有哪些需要考虑的因素呢?**

建立索引的时候一般要考虑到字段的使用频率,经常作为条件进行查询的字段比较适合. 如果需要建立联合索引的话,还需要考虑联合索引中的顺序. 此外也要考虑其他方面,比如防止过多的所有对表造成太大的压力. 这些都和实际的表结构以及查询方式有关.

43. 联合索引是什么?为什么需要注意联合索引中的顺序?**

MySQL可以使用多个字段同时建立一个索引,叫做联合索引. 在联合索引中,如果想要命中索引,需要按照建立索引时的字段顺序挨个使用,否则无法命中索引.

具体原因为:

MySQL使用索引时需要索引有序,假设现在建立了"name,age,school"的联合索引,那么索引的排序为: 先按照name排序,如果name相同,则按照age排序,如果age的值也相等,则按照school进行排序.

当进行查询时,此时索引仅仅按照name严格有序,因此必须首先使用name字段进行等值查询,之后对于匹配到的列而言,其按照age字段严格有序,此时可以使用age字段用做索引查找...以此类推. 因此在建立联合索引的时候应该注意索引列的顺序,一般情况下,将查询需求频繁或者字段选择性高的列放在前面. 此外可以根据特例的查询或者表结构进行单独的调整.

44. 创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,之后会拿到对语句的分析,也就是执行计划,其中包含了许多信息. 可以通过其中和索引有关的信息来分析是否命中了索引,例如possible_key,key,key_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

45. 那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?

- 使用不等于查询,
- 列参与了数学运算或者函数
- 在字符串like时左边是通配符.类似于'%aaa'.
- 当mysql分析全表扫描比使用索引快的时候不使用索引.
- 当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引.

以上情况,MySQL无法使用索引.

46. 什么是事务?**

事务是逻辑上的一组操作，要么都执行，要么都不执行。

理解什么是事务最经典的就是转账的栗子,相信大家也都了解,这里就不再说一遍了.

事务是一系列的操作,他们要符合ACID特性.最常见的理解就是:事务中的操作要么全部成功,要么全部失败.但是只是这样还不够的.

47. ACID是什么?可以详细说一下吗?

A=Atomicity

原子性: 就是上面说的,要么全部成功,要么全部失败.不可能只执行一部分操作.

C=Consistency

一致性: 系统(数据库)总是从一个一致性的状态转移到另一个一致性的状态,不会存在中间状态.

I=Isolation

隔离性: 通常来说:一个事务在完全提交之前,对其他事务是不可见的.注意前面的通常来说加了红色,意味着有例外情况.

D=Durability

持久性: 一旦事务提交,那么就永远是这样子了,哪怕系统崩溃也不会影响到这个事务的结果.

48. 同时有多个事务在进行会怎么样呢?**

事务 (transaction) 是作为一个单元的一组有序的数据库操作。如果组中的所有操作都成功，则认为事务成功，即使只有一个操作失败，事务也不成功。如果所有操作完成，事务则提交，其修改将作用于所有其他数据库进程。如果一个操作失败，则事务将回滚，该事务所有操作的影响都将取消。

事务特性:

- 1、**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。
- 2、**一致性或可串性。** 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
- 3、**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，

4、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。或者这样理解：事务就是被绑定在一起作为一个逻辑工作单元的 SQL 语句分组，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上有节点。为了确保要么执行，要么不执行，就可以使用事务。要将有组语句作为事务考虑，就需要通过 ACID 测试，即原子性，一致性，隔离性和持久性。

49. MySQL 中的事务回滚机制概述

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位。

事务回滚是指将该事务已经完成的对数据库的更新操作撤销。要同时修改数据库中两个不同表时，如果它们不是一个事务的话，当第一个表修改完，可能第二个表修改过程中出现了异常而没能修改，此时就只有第二个表依旧是未修改之前的状态，而第一个表已经被修改完毕。而当你把它们设定为一个事务的时候，当第一个表修改完，第二表修改出现异常而没能修改，第一个表和第二个表都要回到未修改的状态，这就是所谓的事务回滚

50. 并发事务带来哪些问题？

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对同一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- **脏读 (Dirty read)** : 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改 (Lost to modify)** : 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- **不可重复读 (Unrepeatable read)** : 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- **幻读 (Phantom read)** : 幻读与不可重复读类似。它发生在一个事务 (T1) 读取了几行数据，接着另一个并发事务 (T2) 插入了一些数据时。在随后的查询中，第一个事务 (T1) 就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复读和幻读区别：

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

51. 怎么解决这些问题呢？MySQL的事务隔离级别了解吗？

MySQL的四种隔离级别如下：

- 未提交读(READ UNCOMMITTED)

这个隔离级别下，其他事务可以看到本事务没有提交的部分修改。因此会造成脏读的问题(读取到了其他事务未提交的部分，而之后该事务进行了回滚)。

这个级别的性能没有足够大的优势，但是又有很多的问题，因此很少使用。
微信搜索公众号：Java专栏，获取最新面试手册

- 已提交读(READ COMMITTED)

其他事务只能读取到本事务已经提交的部分.这个隔离级别有 不可重复读的问题,在同一个事务内的两次读取,拿到的结果竟然不一样,因为另外一个事务对数据进行了修改.

- REPEATABLE READ(可重复读)

可重复读隔离级别解决了上面不可重复读的问题(看名字也知道),但是仍然有一个新问题,就是 幻读,当你读取id> 10 的数据行时,对涉及到的所有行加上了读锁,此时例外一个事务新插入了一条id=11的数据,因为是新插入的,所以不会触发上面的锁的排斥,那么进行本事务进行下一次的查询时会发现有一条id=11的数据,而上次的查询操作并没有获取到,再进行插入就会有主键冲突的问题.

- SERIALIZABLE(可串行化)

这是最高的隔离级别,可以解决上面提到的所有问题,因为他强制将所有的操作串行执行,这会导致并发性能极速下降,因此也不是很常用.

52. Innodb使用的是哪种隔离级别呢?

InnoDB默认使用的是可重复读隔离级别.

53. MySQL 中有哪几种锁?

- 1、表级锁： 开销小， 加锁快； 不会出现死锁； 锁定粒度大， 发生锁冲突的概率最高， 并发度最低。
- 2、行级锁： 开销大， 加锁慢； 会出现死锁； 锁定粒度最小， 发生锁冲突的概率最低， 并发度也最高。
- 3、页面锁： 开销和加锁时间界于表锁和行锁之间； 会出现死锁； 锁定粒度界于表锁和行锁之间， 并发度一般。

54. 对MySQL的锁了解吗?

当数据库有并发事务的时候,可能会产生数据的不一致,这时候需要一些机制来保证访问的次序,锁机制就是这样的一个机制.

就像酒店的房间,如果大家随意进出,就会出现多人抢夺同一个房间的情况,而在房间上装上锁,申请到钥匙的人才可以入住并且将房间锁起来,其他人只有等他使用完毕才可以再次使用.

55. 锁机制与InnoDB锁算法

MyISAM和InnoDB存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比：

- 表级锁： MySQL中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB引擎都支持表级锁。

- **行级锁**: MySQL中锁定**粒度最小**的一种锁, 只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小, 并发度高, 但加锁的开销也最大, 加锁慢, 会出现死锁。

•

InnoDB存储引擎的锁的算法有三种:

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

56. MySQL都有哪些锁呢?像上面那样子进行锁定岂不是有点阻碍并发效率了?

从锁的类别上来讲,有共享锁和排他锁.

共享锁: 又叫做读锁. 当用户要进行数据的读取时,对数据加上共享锁.共享锁可以同时加上多个.

排他锁: 又叫做写锁. 当用户要进行数据的写入时,对数据加上排他锁.排他锁只可以加一个,他和其他的排他锁,共享锁都相斥.

用上面的例子来说就是用户的行为有两种,一种是来看房,多个用户一起看房是可以接受的.一种是真正的入住一晚,在这期间,无论是想入住的还是想看房的都不可以.

锁的粒度取决于具体的存储引擎,InnoDB实现了行级锁,页级锁,表级锁.

他们的加锁开销从大大小,并发能力也是从大到小.

57. 锁的优化策略

- 1、读写分离
- 2、分段加锁
- 3、减少锁持有的时间

多个线程尽量以相同的顺序去获取资源

不能将锁的粒度过于细化, 不然可能会出现线程的加锁和释放次数过多, 反而效率不如一次加一把大锁。

58. Explain 性能分析

是什么

查看执行计划: 使用 EXPLAIN 关键字可以模拟优化器执行 SQL 查询语句, 从而知道 MySQL 是如何处理 SQL 语句的。分析查询语句或是表结构的性能瓶颈。

能干嘛

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- 哪些索引被实际使用
- 表之间的引用
- 每张表有多少行被优化器查询

怎么玩

Explain + SQL 语句。

Explain 执行后返回的信息：

id select_type table type possible_keys key key_len ref rows Extra

© 技术社区

各字段解释

1. id: select 查询的序列号，包含一组数字，表示查询中执行 select 子句或操作表的顺序。
 - id 相同，执行顺序由上至下
 - id 不同，如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行
 - id 有相同也有不同：id 如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id 值越大，优先级越高，越先执行

id 号每个号码，表示一趟独立的查询。一个 sql 的查询趟数越少越好。
2. select_type: 代表查询的类型，主要是用于区别普通查询、联合查询、子查询等的复杂查询，取值范围如下：
 - simple: 简单的 select 查询，查询中不包含子查询或者 UNION
 - primary: 查询中若包含任何复杂的子部分，最外层查询则被标记为 primary
 - derived: 在 FROM 列表中包含的子查询被标记为 DERIVED (衍生)，MySQL 会递归执行这些子查询，把结果放在临时表里。
 - subquery: 在 SELECT 或 WHERE 列表中包含了子查询
 - dependent subquery: 在 SELECT 或 WHERE 列表中包含了子查询，子查询基于外层
 - uncacheable subquery: 无法使用缓存的子查询
 - union: 若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION；若 UNION 包含在 FROM 子句的子查询中，外层 SELECT 将被标记为：DERIVED
 - union result: 从 UNION 表获取结果的 SELECT
3. table: 这个数据是基于哪张表的。
4. type: 是查询的访问类型。是较为重要的一个指标，结果值从最好到最坏依次是：system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL，一般来说，得保证查询至少达到 range 级别，最好能达到 ref。

只需要记住：system > const > eq_ref > ref > range > index > ALL 就行了，其他的不常见。

- system: 表只有一行记录（等于系统表），这是 const 类型的特例，平时不会出现，这个也可以忽略不计。
- const: 表示通过索引一次就找到了，const 用于比较 primary key 或者 unique 索引。因为只匹配一行数据，所以很快。如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量。
- eq_ref: 唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描。
- ref: 非唯一性索引扫描，返回匹配某个单独值的所有行。本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合条件的行，所以他应该属于查找和扫描的混合体。
- range: 只检索给定范围的行，使用一个索引来选择行。key 列显示使用了哪个索引一般就是在 where 语句中出现了 between、<、>、in 等的查询这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫描全部索引。
- index: 出现 index 是 sql 使用了索引但是没用索引进行过滤，一般是使用了覆盖索引或者是利用索引进行了排序分组。

- all: 将遍历全表以找到匹配的行。

其他 type 如下：

- index_merge: 在查询过程中需要多个索引组合使用，通常出现在有 or 关键字的 sql 中。
- ref_or_null: 对于某个字段既需要过滤条件，也需要 null 值的情况下。查询优化器会选择用 ref_or_null 连接查询。
- index_subquery: 利用索引来关联子查询，不再全表扫描。
- unique_subquery: 该连接类型类似于 index_subquery。子查询中的唯一索引。

5. possible_keys: 显示可能应用在这张表中的索引，一个或多个。查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用。

6. key: 实际使用的索引。如果为 NULL，则没有使用索引。

7. key_len: 表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。key_len 显示的值为索引字段的最大可能长度，并非实际使用长度。如何计算 key_len?

- 先看索引上字段的类型 + 长度，比如：int=4; varchar(20)=20; char(20)=20
- 如果是 varchar 或者 char 这种字符串字段，视字符集要乘不同的值，比如 utf-8 要乘 3，GBK 要乘 2
- varchar 这种动态字符串要加 2 个字节
- 允许空的字段要加 1 个字节

8. ref: 显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值。

9. rows: 显示 MySQL 认为它执行查询时必须检查的行数。越少越好！

10. Extra: 其他的额外重要的信息。

- Using filesort: 说明 mysql 会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。MySQL 中无法利用索引完成的排序操作称为“文件排序”。**排序字段若通过索引去访问将大大提高排序速度。**
- Using temporary: 使用临时表保存中间结果，MySQL 在对查询结果排序时使用临时表。常见于排序 order by 和分组查询 group by。
- Using index: 表示相应的 select 操作中使用了覆盖索引 (Covering Index)，避免访问了表的数据行，效率不错！如果同时出现 using where，表明索引被用来执行索引键值的查找；如果没有同时出现 using where，表明索引只是用来读取数据而非利用索引执行查找。
- Using where: 表明使用了 where 过滤。
- Using join buffer: 使用了连接缓存。
- impossible where: where 子句的值总是 false，不能用来获取任何数据。
- select tables optimized away: 在没有 group by 子句的情况下，基于索引优化 MIN/MAX 操作或者对于 MyISAM 存储引擎优化 COUNT(*) 操作，不必等到执行阶段再进行计算，查询执行计划生成的阶段即完成优化。
- distinct: 优化 distinct 操作，在找到第一匹配的元祖后即停止找同样值的动作。

59. 如何优化SQL

1、SQL语句中IN包含的值不应过多

MySQL对于IN做了相应的优化，即将IN中的常量全部存储在一个数组里面，而且这个数组是排好序的。但是如果数值较多，产生的消耗也是比较大的。再例如：`select id from table_name where num in(1,2,3)` 对于连续的数值，能用 between 就不要用 in 了；再或者使用连接来替换。

2、SELECT语句务必指明字段名称

SELECT *增加很多不必要的消耗（cpu、io、内存、网络带宽）；增加了使用覆盖索引的可能性；当表结构发生改变时，前断也需要更新。所以要求直接在select后面接上字段名。

3、当只需要一条数据的时候，使用limit 1

这是为了使EXPLAIN中type列达到const类型

4、如果排序字段没有用到索引，就尽量少排序

5、如果限制条件中其他字段没有索引，尽量少用or

or两边的字段中，如果有一个不是索引字段，而其他条件也不是索引字段，会造成该查询不走索引的情况。很多时候使用 union all 或者是union(必要的时候)的方式来代替“or”会得到更好的效果

6、尽量用union all代替union

union和union all的差异主要是前者需要将结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的CPU运算，加大资源消耗及延迟。**当然，union all的前提条件是两个结果集没有重复数据。**

7、不使用ORDER BY RAND()

```
select id from `table_name` order by rand() limit 1000;
```

上面的sql语句，可优化为

```
select id from `table_name` t1 join (select rand() * (select max(id) from `table_name`) as nid) t2 on t1.id > t2.nid limit 1000;
```

8、区分in和exists， not in和not exists

```
select * from 表A where id in (select id from 表B)
```

上面sql语句相当于

```
select * from 表A where exists(select * from 表B where 表B.id=表A.id)
```

区分in和exists主要是造成了驱动顺序的改变（这是性能变化的关键），如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询。所以**IN适合于外表大而内表小的情况； EXISTS适合于外表小而内表大的情况。**

关于not in和not exists，推荐使用not exists，不仅仅是效率问题，not in可能存在逻辑问题。**如何高效的写出一个替代not exists的sql语句？**

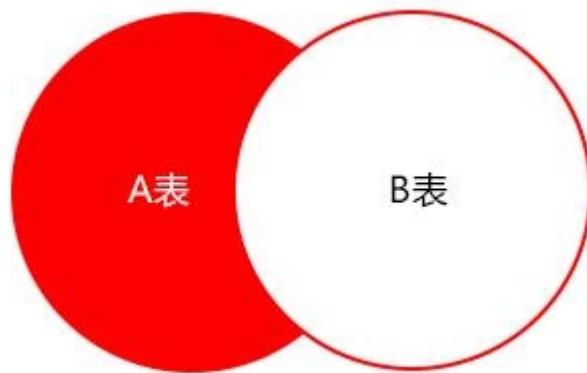
原sql语句

```
select colname ... from A表 where a.id not in (select b.id from B表)
```

高效的sql语句

```
select colname ... from A表 Left join B表 on where a.id = b.id where b.id is null
```

取出的结果集如下图表示，A表不在B表中的数据



9、使用合理的分页方式以提高分页的效率

```
select id,name from table_name limit 866613, 20
```

使用上述sql语句做分页的时候，可能有人会发现，随着表数据量的增加，直接使用limit分页查询会越来越慢。

优化的方法如下：可以取前一页的最大行数的id，然后根据这个最大的id来限制下一页的起点。比如此例中，上一页最大的id是866612。sql可以采用如下的写法：

```
select id,name from table_name where id> 866612 limit 20
```

10、分段查询

在一些用户选择页面中，可能一些用户选择的时间范围过大，造成查询缓慢。主要的原因是扫描行数过多。这个时候可以通过程序，分段进行查询，循环遍历，将结果合并处理进行展示。

如下图这个sql语句，扫描的行数成百万级以上的时候就可以使用分段查询

```
>explain select dep_stand_position,fdst from t_sch_deptime force index(idx_sch_deptime) where scheduled_deptime between n 1487116800 and 1487548800;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | dynamic_201702 | range | idx_sch_deptime | idx_sch_deptime | 4 | NULL | 3610145 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

11、避免在 where 子句中对字段进行 null 值判断

对于null的判断会导致引擎放弃使用索引而进行全表扫描。

12、不建议使用%前缀模糊查询

例如LIKE "%name"或者LIKE "%name%"，这种查询会导致索引失效而进行全表扫描。但是可以使用LIKE "name%"。

那如何查询%name%？

微信搜索公众号：Java专栏，获取最新面试手册

如下图所示，虽然给secret字段添加了索引，但在explain结果果并没有使用

```
mysql> explain select create_time,type from `tbl_flight_item` where secret like '%9573aa%';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tbl_flight_item | NULL | ALL | NULL | NULL | NULL | NULL | 61439228 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set. 1 warning (0.00 sec)

mysql> show create table tbl_flight_item;
+-----+-----+
| Table | Create Table |
+-----+-----+
| `tbl_flight_item` | CREATE TABLE `tbl_flight_item` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `create_time` int(11) DEFAULT NULL COMMENT '记录时间',
  `type` smallint(6) DEFAULT NULL COMMENT '类型',
  `secret` char(32) DEFAULT NULL COMMENT 'md5',
  `level` char(2) DEFAULT NULL COMMENT '等级',
  PRIMARY KEY (`id`),
  KEY `idx_create_time` (`create_time`),
  KEY `idx_type` (`type`),
  KEY `idx_secret` (`secret`),
  KEY `idx_level` (`level`)
) ENGINE=InnoDB AUTO_INCREMENT=65872985 DEFAULT CHARSET=latin1 |
+-----+-----+
```

那么如何解决这个问题呢，答案：**使用全文索引**

在我们查询中经常会用到select id,fnum,fdst from table_name where user_name like '%zhangsan%'。这样的语句，普通索引是无法满足查询需求的。庆幸的是在MySQL中，有全文索引来帮助我们。

创建全文索引的sql语法是：

```
ALTER TABLE `table_name` ADD FULLTEXT INDEX `idx_user_name`(`user_name`);
```

使用全文索引的sql语句是：

```
select id,fnum,fdst from table_name where match(user_name) against('zhangsan' in boolean mode);
```

注意：在需要创建全文索引之前，请联系DBA确定能否创建。同时需要注意的是查询语句的写法与普通索引的区别

13、避免在where子句中对字段进行表达式操作

比如

```
select user_id,user_project from table_name where age*2=36;
```

中对字段就行了算术运算，这会造成引擎放弃使用索引，建议改成

```
select user_id,user_project from table_name where age=36/2;
```

14、避免隐式类型转换

where 子句中出现 column 字段的类型和传入的参数类型不一致的时候发生的类型转换，建议先确定 where中的参数类型

```

mysql> explain select id,create_time from flight_anum where type=43;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | flight_anum | NULL | ALL | idx_type | NULL | NULL | NULL | 2 | 50.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set. 3 warnings (0.00 sec)

mysql> show create table flight_anum;
+-----+-----+
| Table | Create Table |
+-----+-----+
| flight_anum | CREATE TABLE `flight_anum` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`create_time` int(11) DEFAULT NULL COMMENT '记录时间',
`type` varchar(6) DEFAULT NULL COMMENT '类型',
PRIMARY KEY (`id`),
KEY `idx_type` (`type`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=latin1 |
+-----+-----+
1 row in set (0.00 sec)

mysql> explain select id,create_time from flight_anum where type='43';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | flight_anum | NULL | ref | idx_type | idx_type | 9 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

15、对于联合索引来说，要遵守最左前缀法则

举例来说索引含有字段id,name,school，可以直接用id字段，也可以id,name这样的顺序，但是name;school都无法使用这个索引。所以在创建联合索引的时候一定要注意索引字段顺序，常用的查询字段放在最前面

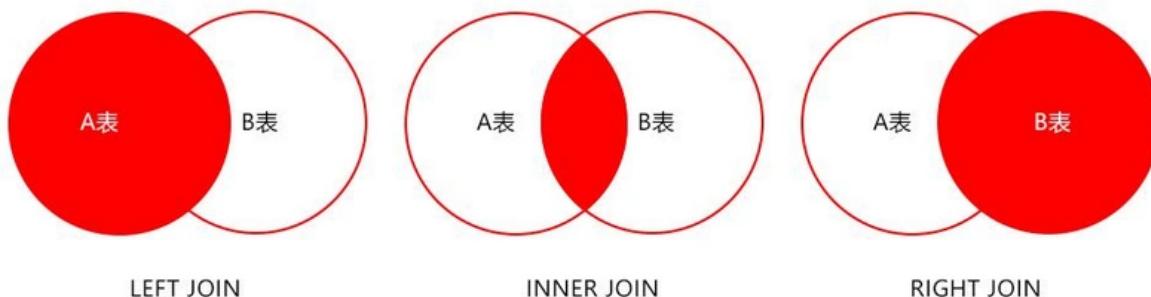
16、必要时可以使用force index来强制查询走某个索引

有的时候MySQL优化器采取它认为合适的索引来检索sql语句，但是可能它所采用的索引并不是我们想要的。这时就可以采用force index来强制优化器使用我们制定的索引。

17、注意范围查询语句

对于联合索引来说，如果存在范围查询，比如between,>,<等条件时，会造成后面的索引字段失效。

18、关于JOIN优化



- LEFT JOIN A表为驱动表
- INNER JOIN MySQL会自动找出那个数据少的表作用驱动表
- RIGHT JOIN B表为驱动表

注意：MySQL中没有full join，可以用以下方式来解决

```
select * from A left join B on B.name = A.name  
where B.name is null  
union all  
select * from B;
```

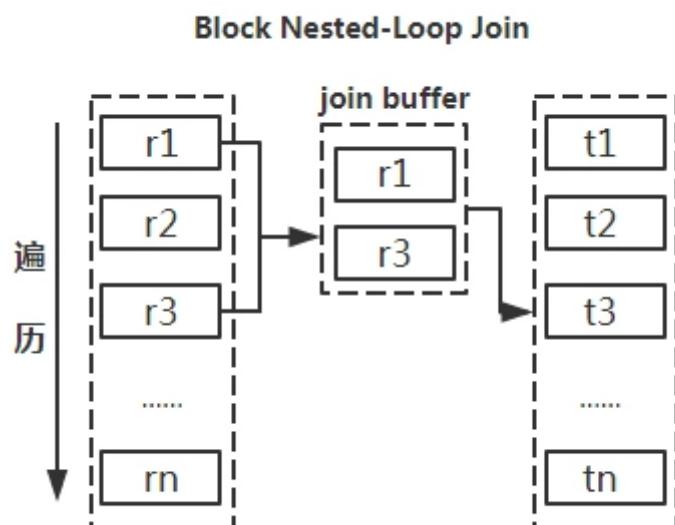
尽量使用inner join，避免left join

参与联合查询的表至少为2张表，一般都存在大小之分。如果连接方式是inner join，在没有其他过滤条件的情况下MySQL会自动选择小表作为驱动表，但是left join在驱动表的选择上遵循的是左边驱动右边的原则，即left join左边的表名为驱动表。

合理利用索引

被驱动表的索引字段作为on的限制字段。

利用小表去驱动大表



从原理图能够直观的看出如果能够减少驱动表的话，减少嵌套循环中的循环次数，以减少 IO总量及CPU运算的次数。

巧用STRAIGHT_JOIN

inner join是由mysql选择驱动表，但是有些特殊情况需要选择另个表作为驱动表，比如有group by、order by等「Using filesort」、「Using temporary」时。STRAIGHT_JOIN来强制连接顺序，在STRAIGHT_JOIN左边的表名就是驱动表，右边则是被驱动表。在使用STRAIGHT_JOIN有个前提条件是该查询是内连接，也就是inner join。其他链接不推荐使用STRAIGHT_JOIN，否则可能造成查询结果不准确。

```

SELECT `rl`.`fid`,
`rl`.`update_time` AS `add_time`,
`f`.`fnum`,
`f`.`dst_parking`,
`f`.`aircraft_num`,
`f`.`touch_down_runway`,
`f`.`forg`,
`f`.`fdst`,
`f`.`flight_status_code`,
`fd`.`fdst` AS `fd_fdst`,
`ds`.`touch_down_runway_source`,
`ds`.`touch_down_runway_update_time`,
`u`.`truelname`,
`a`.`aircraft_model`
FROM `_____` AS `rl`
STRAIGHT_JOIN `_____` AS `f` ON `rl`.`fid` = `f`.`fid`
LEFT JOIN `_____` AS `a` ON `f`.`aircraft_num` = `a`.`aircraft_num`
LEFT JOIN `_____` AS `fl` ON `f`.`fid` = `fl`.`arr_fid`
LEFT JOIN `_____` AS `fd` ON `fl`.`dep_fid` = `fd`.`fid`
LEFT JOIN `_____` AS `ds` ON `f`.`fid` = `ds`.`fid`
LEFT JOIN `_____` AS `u` ON `ds`.`touch_down_runway_source` = `u`.`uid`
WHERE `rl`.`op_type` = 0
AND `f`.`fdst` = 'PVG'
AND `f`.`fid` != ''
GROUP BY `rl`.`fid`
ORDER BY `rl`.`update_time` DESC
LIMIT 20;

```

这种方式有时可能减少3倍的时间。

52条 SQL性能优化策略

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1、对查询进行优化，应尽量避免全表扫描，首先应考虑在where及order by涉及的列上建立索引。

2、应尽量避免在where子句中对字段进行null值判断，创建表时NULL是默认值，但大多数时候应该使用NOT NULL，或者使用一个特殊的值，如0, -1作为默认值。

3、应尽量避免在where子句中使用!=或<>操作符，MySQL只有对以下操作符才使用索引：<, <=, =, >, >=, BETWEEN, IN, 以及某些时候的LIKE。

4、应尽量避免在where子句中使用or来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，可以使用UNION合并查询：select id from t where num=10 union all select id from t where num=20。

5、in和not in也要慎用，否则会导致全表扫描，对于连续的数值，能用between就不要用in了：Select id from t where num between 1 and 3。

6、下面的查询也将导致全表扫描：select id from t where name like'%abc%'或者select id from t where name like'%abc'若要提高效率，可以考虑全文检索。而select id from t where name like'abc%'才用到索引。

7、如果在where子句中使用参数，也会导致全表扫描。

8、应尽量避免在where子句中对字段进行表达式操作，应尽量避免在where子句中对字段进行函数操作。

9、很多时候用exists代替in是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where num=a.num)
```

10、索引固然可以提高相应的select的效率，但同时也降低了insert及update的效率，因为insert或update时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

11、尽可能的避免更新clustered索引数据列，因为clustered索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新clustered索引数据列，那么需要考虑是否应将该索引建为clustered索引。

12、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

13、尽可能的使用varchar/nvarchar代替char/nchar，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

14、最好不要使用""返回所有：select from t，用具体的字段列表代替"*"，不要返回用不到的任何字段。

15、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

16、使用表的别名(Alias): 当在SQL语句中连接多个表时,请使用表的别名并把别名前缀于每个Column上。这样一来,就可以减少解析的时间并减少那些由Column歧义引起的语法错误。

17、使用“临时表”暂存中间结果 :

简化SQL语句的重要方法就是采用临时表暂存中间结果,但是临时表的好处远远不止这些,将临时结果暂存在临时表,后面的查询就在tempdb中了,这可以避免程序中多次扫描主表,也大大减少了程序执行中“共享锁”阻塞“更新锁”,减少了阻塞,提高了并发性能。

18、一些SQL查询语句应加上nolock, 读、写是会相互阻塞的,为了提高并发性能,对于一些查询,可以加上nolock,这样读的时候可以允许写,但缺点是可能读到未提交的脏数据。

使用nolock有3条原则:

- 查询的结果用于“插、删、改”的不能加nolock;
- 查询的表属于频繁发生页分裂的,慎用nolock;
- 使用临时表一样可以保存“数据前影”,起到类似Oracle的undo表空间的功能,能采用临时表提高并发性能的,不要用nolock。

19、常见的简化规则如下:

不要有超过5个以上的表连接 (JOIN),考虑使用临时表或表变量存放中间结果。少用子查询,视图嵌套不要过深,一般视图嵌套不要超过2个为宜。

20、将需要查询的结果预先计算好放在表中,查询的时候再Select。这在SQL7.0以前是最重要的手段,例如医院的住院费计算。

21、用OR的字句可以分解成多个查询,并且通过UNION 连接多个查询。他们的速度只同是否使用索引有关,如果查询需要用到联合索引,用UNION all执行的效率更高。多个OR的字句没有用到索引,改写成UNION的形式再试图与索引匹配。一个关键的问题是否用到索引。

22、在IN后面值的列表中,将出现最频繁的值放在最前面,出现得最少的放在最后面,减少判断的次数。

23、尽量将数据的处理工作放在服务器上,减少网络的开销,如使用存储过程。

存储过程是编译好、优化过、并且被组织到一个执行规划里、且存储在数据库中的SQL语句,是控制流语言的集合,速度当然快。反复执行的动态SQL,可以使用临时存储过程,该过程(临时表)被放在Tempdb中。

24、当服务器的内存够多时,配制线程数量 = 最大连接数+5,这样能发挥最大的效率;否则使用配制线程数量<最大连接数启用SQL SERVER的线程池来解决,如果还是数量 = 最大连接数+5,严重的损害服务器的性能。

25、查询的关联同写的顺序 :

```
select a.personMemberID, * from chineseresume a, personmember b where  
personMemberID = b.referenceid and a.personMemberID = 'JCNPRH39681' (A = B ,B =  
'号码')
```

```
select a.personMemberID, * from chineseresume a, personmember b where  
a.personMemberID = b.referenceid and a.personMemberID = 'JCNPRH39681' and  
b.referenceid = 'JCNPRH39681' (A = B ,B = '号码', A = '号码')
```

```
select a.personMemberID, * from chineseresume a, personmember b where  
b.referenceid = 'JCNPRH39681' and a.personMemberID = 'JCNPRH39681' (B = '号码',  
A = '号码')
```

26、尽量使用exists代替select count(1)来判断是否存在记录，count函数只有在统计表中所有行数时使用，而且count(1)比count(*)更有效率。

27、尽量使用“>=”，不要使用“>”。

28、索引的使用规范：

- 索引的创建要与应用结合考虑，建议大的OLTP表不要超过6个索引；
- 尽可能的使用索引字段作为查询条件，尤其是聚簇索引，必要时可以通过index index_name来强制指定索引；
- 避免对大表查询时进行table scan，必要时考虑新建索引；
- 在使用索引字段作为条件时，如果该索引是联合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用；
- 要注意索引的维护，周期性重建索引，重新编译存储过程。

29、下列SQL条件语句中的列都建有恰当的索引，但执行速度却非常慢：

```
SELECT * FROM record WHERE substrIng(card_no,1,4)='5378' (13秒)
```

```
SELECT * FROM record WHERE amount/30< 1000 (11秒)
```

```
SELECT * FROM record WHERE convert(char(10),date,112)='19991201' (10秒)
```

分析：

WHERE子句中对列的任何操作结果都是在SQL运行时逐列计算得到的，因此它不得不进行表搜索，而没有使用该列上面的索引。

如果这些结果在查询编译时就能得到，那么就可以被SQL优化器优化，使用索引，避免表搜索，因此将SQL重写成下面这样：

```
SELECT * FROM record WHERE card_no like '5378%' (< 1秒)

SELECT * FROM record WHERE amount< 1000*30 (< 1秒)

SELECT * FROM record WHERE date= '1999/12/01' (< 1秒)
```

30、当有一批处理的插入或更新时，用批量插入或批量更新，绝不会一条条记录的去更新。

31、在所有的存储过程中，能够用SQL语句的，我绝不会用循环去实现。

例如：列出上个月的每一天，我会用connect by去递归查询一下，绝不会去用循环从上个月第一天到最后一天。

32、选择最有效率的表名顺序（只在基于规则的优化器中有效）：

Oracle的解析器按照从右到左的顺序处理FROM子句中的表名，FROM子句中写在最后的表（基础表driving table）将被最先处理，在FROM子句中包含多个表的情况下，你必须选择记录条数最少的表作为基础表。

如果有3个以上的表连接查询，那就需要选择交叉表（intersection table）作为基础表，交叉表是指那个被其他表所引用的表。

33、提高GROUP BY语句的效率，可以通过将不需要的记录在GROUP BY之前过滤掉。下面两个查询返回相同结果，但第二个明显就快了许多。

低效：

```
SELECT JOB , AVG(SAL)

FROM EMP

GROUP BY JOB

HAVING JOB = 'PRESIDENT'

OR JOB = 'MANAGER'
```

高效：

```
SELECT JOB , AVG(SAL)  
FROM EMP  
WHERE JOB = 'PRESIDENT'  
OR JOB = 'MANAGER'  
GROUP BY JOB
```

34、SQL语句用大写，因为Oracle总是先解析SQL语句，把小写的字母转换成大写的再执行。

35、别名的使用，别名是大型数据库的应用技巧，就是表名、列名在查询中以一个字母为别名，查询速度要比建连接表快1.5倍。

36、避免死锁，在你的存储过程和触发器中访问同一个表时总是以相同的顺序；事务应尽可能地缩短，在一个事务中应尽可能减少涉及到的数据量；永远不要在事务中等待用户输入。

37、避免使用临时表，除非却有需要，否则应尽量避免使用临时表，相反，可以使用表变量代替；大多数时候(99%)，表变量驻扎在内存中，因此速度比临时表更快，临时表驻扎在TempDb数据库中，因此临时表上的操作需要跨数据库通信，速度自然慢。

38、最好不要使用触发器：

- 触发一个触发器，执行一个触发器事件本身就是一个耗费资源的过程；
- 如果能够使用约束实现的，尽量不要使用触发器；
- 不要为不同的触发事件(Insert, Update和Delete)使用相同的触发器；
- 不要在触发器中使用事务型代码。

39、索引创建规则：

- 表的主键、外键必须有索引；
- 数据量超过300的表应该有索引；
- 经常与其他表进行连接的表，在连接字段上应该建立索引；
- 经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；
- 索引应该建在选择性高的字段上；
- 索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；
- 复合索引的建立需要进行仔细分析，尽量考虑用单字段索引代替；
- 正确选择复合索引中的主列字段，一般是选择性较好的字段；
- 复合索引的几个字段是否经常同时以AND方式出现在Where子句中？单字段查询是否极少甚至没有？如果是，则可以建立复合索引；否则考虑单字段索引；
- 如果复合索引中包含的字段经常单独出现在Where子句中，则分解为多个单字段索引；
- 如果复合索引所包含的字段超过3个，那么仔细考虑其必要性，考虑减少复合的字段；
- 如果既有单字段索引，又有这几个字段上的复合索引，一般可以删除复合索引；
- 频繁进行数据操作的表，不要建立太多的索引；

- 删除无用的索引，避免对执行计划造成负面影响；
- 表上建立的每个索引都会增加存储开销，索引对于插入、删除、更新操作也会增加处理上的开销。另外，过多的复合索引，在有单字段索引的情况下，一般都是没有存在价值的；相反，还会降低数据增加删除时的性能，特别是对频繁更新的表来说，负面影响更大。
- 尽量不要对数据库中某个含有大量重复的值的字段建立索引。

40、MySQL查询优化总结：

使用慢查询日志去发现慢查询，使用执行计划去判断查询是否正常运行，总是去测试你的查询看看是否他们运行在最佳状态下。

久而久之性能总会变化，避免在整个表上使用count(*)，它可能锁住整张表，使查询保持一致以便后续相似的查询可以使用查询缓存，在适当的情形下使用GROUP BY而不是DISTINCT，在WHERE、GROUP BY和ORDER BY子句中使用有索引的列，保持索引简单，不在多个索引中包含同一个列。

有时候MySQL会使用错误的索引，对于这种情况使用USE INDEX，检查使用SQL_MODE=STRICT的问题，对于记录数小于5的索引字段，在UNION的时候使用LIMIT而不是用OR。

为了避免在更新前SELECT，使用INSERT ON DUPLICATE KEY或者INSERT IGNORE，不要用UPDATE去实现，不要使用MAX，使用索引字段和ORDER BY子句，LIMIT M, N实际上可以减缓查询在某些情况下，有节制地使用，在WHERE子句中使用UNION代替子查询，在重新启动的MySQL，记得来温暖你的数据库，以确保数据在内存和查询速度快，考虑持久连接，而不是多个连接，以减少开销。

基准查询，包括使用服务器上的负载，有时一个简单的查询可以影响其他查询，当负载增加在服务器上，使用SHOW PROCESSLIST查看慢的和有问题的查询，在开发环境中产生的镜像数据中测试的所有可疑的查询。

41、MySQL备份过程：

- 从二级复制服务器上进行备份；
- 在进行备份期间停止复制，以避免在数据依赖和外键约束上出现不一致；
- 彻底停止MySQL，从数据库文件进行备份；
- 如果使用MySQL dump进行备份，请同时备份二进制日志文件 - 确保复制没有中断；
- 不要信任LVM快照，这很可能产生数据不一致，将来会给你带来麻烦；
- 为了更容易进行单表恢复，以表为单位导出数据——如果数据是与其他表隔离的。
- 当使用mysqldump时请使用-opt；
- 在备份之前检查和优化表；
- 为了更快的进行导入，在导入时临时禁用外键约束。；
- 为了更快的进行导入，在导入时临时禁用唯一性检测；
- 在每一次备份后计算数据库，表以及索引的尺寸，以便更够监控数据尺寸的增长；
- 通过自动调度脚本监控复制实例的错误和延迟；
- 定期执行备份。

42、查询缓冲并不自动处理空格，因此，在写SQL语句时，应尽量减少空格的使用，尤其是在SQL首和尾的空格（因为查询缓冲并不自动截取首尾空格）。

43、member用mid做标准进行分表方便查询么？一般的业务需求中基本上都是以username为查询依据，正常应当是username做hash取模来分表。

而分表的话MySQL的partition功能就是干这个的，对代码是透明的；在代码层面去实现貌似是不合理的。

44、我们应该为数据库里的每张表都设置一个ID做为其主键，而且最好的是一个INT型的（推荐使用UNSIGNED），并设置上自动增加的AUTO_INCREMENT标志。

45、在所有的存储过程和触发器的开始处设置SET NOCOUNT ON，在结束时设置SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送DONE_IN_PROC消息。

46、MySQL查询可以启用高速查询缓存。这是提高数据库性能的有效MySQL优化方法之一。当同一个查询被执行多次时，从缓存中提取数据和直接从数据库中返回数据快很多。

47、EXPLAIN SELECT查询用来跟踪查看效果：

使用EXPLAIN关键字可以让你知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的。

48、当只要一行数据时使用LIMIT 1：

当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。

在这种情况下，加上LIMIT 1可以增加性能。这样一来，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

49、选择表合适存储引擎：

- myisam：应用时以读和插入操作为主，只有少量的更新和删除，并且对事务的完整性，并发性要求不是很高的。
- InnoDB：事务处理，以及并发条件下要求数据的一致性。除了插入和查询外，包括很多的更新和删除。（InnoDB有效地降低删除和更新导致的锁定）。

对于支持事务的InnoDB类型的表来说，影响速度的主要原因是AUTOCOMMIT默认设置是打开的，而且程序没有显式调用BEGIN 开始事务，导致每插入一条都自动提交，严重影响了速度。可以在执行SQL前调用begin，多条SQL形成一个事物（即使autocommit打开也可以），将大大提高性能。

50、优化表的数据类型，选择合适的数据类型：

原则：更小通常更好，简单就好，所有字段都得有默认值，尽量避免null。

例如：数据库表设计时候更小的占磁盘空间尽可能使用更小的整数类型。（mediumint就比int更合适）

比如时间字段：datetime和timestamp，datetime占用8个字节，而timestamp占用4个字节，只用了一半，而timestamp表示的范围是1970—2037适合做更新时间

MySQL可以很好的支持大数据量的存取，但是一般说来，数据库中的表越小，在它上面执行的查询也就越快。

因此，在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。

例如：在定义邮政编码这个字段时，如果将其设置为CHAR(255)，显然给数据库增加了不必要的空间。甚至使用VARCHAR这种类型也是多余的，因为CHAR(6)就可以很好的完成任务了。

同样的，如果可以的话，我们应该使用MEDIUMINT而不是BIGIN来定义整型字段，应该尽量把字段设置为NOT NULL，这样在将来执行查询的时候，数据库不用去比较NULL值。

对于某些文本字段，例如“省份”或者“性别”，我们可以将它们定义为ENUM类型。因为在MySQL中，ENUM类型被当作数值型数据来处理，而数值型数据被处理起来的速度要比文本类型快得多。这样，我们又可以提高数据库的性能。

51、字符串数据类型：char, varchar, text选择区别。

52、任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。

一千行SQL命令

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

基本操作

```
/* Windows服务 */
-- 启动MySQL
net start mysql
-- 创建Windows服务
sc create mysql binPath= mysqld_bin_path(注意：等号与值之间有空格)
/* 连接与断开服务器 */
mysql -h 地址 -P 端口 -u 用户名 -p 密码
SHOW PROCESSLIST -- 显示哪些线程正在运行
SHOW VARIABLES -- 显示系统变量信息
```

数据库操作

```
/* 数据库操作 */ -----
-- 查看当前数据库
    SELECT DATABASE();
-- 显示当前时间、用户名、数据库版本
    SELECT now(), user(), version();
-- 创建库
    CREATE DATABASE[ IF NOT EXISTS] 数据库名 数据库选项
    数据库选项:
        CHARACTER SET charset_name
        COLLATE collation_name
-- 查看已有库
    SHOW DATABASES[ LIKE 'PATTERN']
-- 查看当前库信息
    SHOW CREATE DATABASE 数据库名
-- 修改库的选项信息
    ALTER DATABASE 库名 选项信息
-- 删除库
    DROP DATABASE[ IF EXISTS] 数据库名
    同时删除该数据库相关的目录及其目录内容
```

表的操作

```
-- 创建表
    CREATE [TEMPORARY] TABLE[ IF NOT EXISTS] [库名.]表名 ( 表的结构定义 )[ 表选项]
    每个字段必须有数据类型
    最后一个字段后不能有逗号
    TEMPORARY 临时表，会话结束时表自动消失
    对于字段的定义:
        字段名 数据类型 [NOT NULL | NULL] [DEFAULT default_value]
        [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY] [COMMENT 'string']
-- 表选项
    -- 字符集
        CHARSET = charset_name
        如果表没有设定，则使用数据库字符集
    -- 存储引擎
        ENGINE = engine_name
        表在管理数据时采用的不同的数据结构，结构不同会导致处理方式、提供的特性操作等不同
        常见的引擎: InnoDB MyISAM Memory/Heap BDB Merge Example CSV MaxDB Archive
        不同的引擎在保存表的结构和数据时采用不同的方式
        MyISAM表文件含义: .frm表定义, .MYD表数据, .MYI表索引
        InnoDB表文件含义: .frm表定义, 表空间数据和日志文件
        SHOW ENGINES -- 显示存储引擎的状态信息
        SHOW ENGINE 引擎名 {LOGS|STATUS} -- 显示存储引擎的日志或状态信息
    -- 自增起始数
        AUTO_INCREMENT = 行数
    -- 数据文件目录
        DATA DIRECTORY = '目录'
    -- 索引文件目录
        INDEX DIRECTORY = '目录'
    -- 表注释
        COMMENT = 'string'
```

```

-- 分区选项
PARTITION BY ... (详细见手册)

-- 查看所有表
SHOW TABLES[ LIKE 'pattern']
SHOW TABLES FROM 库名

-- 查看表结构
SHOW CREATE TABLE 表名 (信息更详细)
DESC 表名 / DESCRIBE 表名 / EXPLAIN 表名 / SHOW COLUMNS FROM 表名 [LIKE
'PATTERN']
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']

-- 修改表
-- 修改表本身的选项
ALTER TABLE 表名 表的选项
eg: ALTER TABLE 表名 ENGINE=MYISAM;

-- 对表进行重命名
RENAME TABLE 原表名 TO 新表名
RENAME TABLE 原表名 TO 库名.表名 (可将表移动到另一个数据库)
-- RENAME可以交换两个表名

-- 修改表的字段机构 (13.1.2. ALTER TABLE语法)
ALTER TABLE 表名 操作名
-- 操作名
ADD[ COLUMN] 字段定义      -- 增加字段
AFTER 字段名      -- 表示增加在该字段名后面
FIRST      -- 表示增加在第一个
ADD PRIMARY KEY(字段名)    -- 创建主键
ADD UNIQUE [索引名] (字段名) -- 创建唯一索引
ADD INDEX [索引名] (字段名) -- 创建普通索引
DROP[ COLUMN] 字段名      -- 删除字段
MODIFY[ COLUMN] 字段名 字段属性      -- 支持对字段属性进行修改, 不能修改字段名
(所有原有属性也需写上)
CHANGE[ COLUMN] 原字段名 新字段名 字段属性      -- 支持对字段名修改
DROP PRIMARY KEY      -- 删除主键(删除主键前需删除其AUTO_INCREMENT属性)
DROP INDEX 索引名      -- 删除索引
DROP FOREIGN KEY 外键      -- 删除外键

-- 删除表
DROP TABLE[ IF EXISTS] 表名 ...
-- 清空表数据
TRUNCATE [TABLE] 表名

-- 复制表结构
CREATE TABLE 表名 LIKE 要复制的表名

-- 复制表结构和数据
CREATE TABLE 表名 [AS] SELECT * FROM 要复制的表名

-- 检查表是否有错误
CHECK TABLE tb1_name [, tb1_name] ... [option] ...

-- 优化表
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tb1_name [, tb1_name] ...
-- 修复表
REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE tb1_name [, tb1_name] ... [QUICK]
[EXTENDED] [USE_FRM]

-- 分析表
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tb1_name [, tb1_name] ...

```

数据操作

```
/* 数据操作 */ -----
-- 增
INSERT [INTO] 表名 [(字段列表)] VALUES (值列表) [, (值列表), ...]
    -- 如果要插入的值列表包含所有字段并且顺序一致，则可以省略字段列表。
    -- 可同时插入多条数据记录！
REPLACE 与 INSERT 完全一样，可互换。
INSERT [INTO] 表名 SET 字段名=值[, 字段名=值, ...]

-- 查
SELECT 字段列表 FROM 表名[ 其他子句]
    -- 可来自多个表的多个字段
    -- 其他子句可以不使用
    -- 字段列表可以用*代替，表示所有字段

-- 删
DELETE FROM 表名[ 删除条件子句]
    没有条件子句，则会删除全部

-- 改
UPDATE 表名 SET 字段名=新值[, 字段名=新值] [更新条件]
```

字符集编码

```
/* 字符集编码 */ -----
-- MySQL、数据库、表、字段均可设置编码
-- 数据编码与客户端编码不需一致
SHOW VARIABLES LIKE 'character_set_%'      -- 查看所有字符集编码项
    character_set_client          客户端向服务器发送数据时使用的编码
    character_set_results         服务器端将结果返回给客户端所使用的编码
    character_set_connection      连接层编码

SET 变量名 = 变量值
    SET character_set_client = gbk;
    SET character_set_results = gbk;
    SET character_set_connection = gbk;
SET NAMES GBK;   -- 相当于完成以上三个设置

-- 校对集
    校对集用以排序
    SHOW CHARACTER SET [LIKE 'pattern']/SHOW CHARSET [LIKE 'pattern']    查看所有字符集
        SHOW COLLATION [LIKE 'pattern']        查看所有校对集
        CHARSET 字符集编码        设置字符集编码
        COLLATE 校对集编码        设置校对集编码
```

数据类型（列类型）

```
/* 数据类型（列类型） */ -----
1. 数值类型
-- a. 整型 -----
    类型      字节      范围（有符号位）
    tinyint    1字节    -128 ~ 127      无符号位: 0 ~ 255
    smallint   2字节    -32768 ~ 32767
```

mediumint 3字节 -8388608 ~ 8388607

int 4字节

bigint 8字节

int(M) M表示总位数

- 默认存在符号位, **unsigned** 属性修改

- 显示宽度, 如果某个数不够定义字段时设置的位数, 则前面以0补填, **zerofill** 属性修改

例: **int(5)** 插入一个数 '123', 补填后为 '00123'

- 在满足要求的情况下, 越小越好。

- 1表示**bool**值真, 0表示**bool**值假。MySQL没有布尔类型, 通过整型0和1表示。常用**tinyint(1)**表示布尔型。

-- b. 浮点型 -----

类型 字节 范围

float(单精度) 4字节

double(双精度) 8字节

浮点型既支持符号位 **unsigned** 属性, 也支持显示宽度 **zerofill** 属性。

不同于整型, 前后均会补填0.

定义浮点型时, 需指定总位数和小数位数。

float(M, D) **double(M, D)**

M表示总位数, D表示小数位数。

M和D的大小会决定浮点数的范围。不同于整型的固定范围。

M既表示总位数(不包括小数点和正负号), 也表示显示宽度(所有显示符号均包括)。

支持科学计数法表示。

浮点数表示近似值。

-- c. 定点数 -----

decimal -- 可变长度

decimal(M, D) M也表示总位数, D表示小数位数。

保存一个精确的数值, 不会发生数据的改变, 不同于浮点数的四舍五入。

将浮点数转换为字符串来保存, 每9位数字保存为4个字节。

2. 字符串类型

-- a. char, varchar -----

char 定长字符串, 速度快, 但浪费空间

varchar 变长字符串, 速度慢, 但节省空间

M表示能存储的最大长度, 此长度是字符数, 非字节数。

不同的编码, 所占用的空间不同。

char, 最多255个字符, 与编码无关。

varchar, 最多65535字符, 与编码有关。

一条有效记录最大不能超过65535个字节。

utf8 最大为21844个字符, **gbk** 最大为32766个字符, **latin1** 最大为65532个字符

varchar 是变长的, 需要利用存储空间保存 **varchar** 的长度, 如果数据小于255个字节, 则采用一个字节来保存长度, 反之需要两个字节来保存。

varchar 的最大有效长度由最大行大小和使用的字符集确定。

最大有效长度是65532字节, 因为在**varchar**存字符串时, 第一个字节是空的, 不存在任何数据, 然后还需两个字节来存放字符串的长度, 所以有效长度是 $65535-1-2=65532$ 字节。

例: 若一个表定义为 **CREATE TABLE tb(c1 int, c2 char(30), c3 varchar(N))**

charset=utf8; 问N的最大值是多少? 答: $(65535-1-2-4-30*3)/3$

-- b. blob, text -----

blob 二进制字符串(字节字符串)

tinyblob, **blob**, **mediumblob**, **longblob**

text 非二进制字符串(字符字符串)

tinytext, **text**, **mediumtext**, **longtext**

text 在定义时, 不需要定义长度, 也不会计算总长度。

text 类型在定义时, 不可给**default**值

-- c. binary, varbinary -----

类似于**char**和**varchar**, 用于保存二进制字符串, 也就是保存字节字符串而非字符字符串。

char, **varchar**, **text** 对应 **binary**, **varbinary**, **blob**.

3. 日期时间类型

一般用整型保存时间戳, 因为PHP可以很方便的将时间戳进行格式化。

datetime 8字节 日期及时间 1000-01-01 00:00:00 到 9999-12-31 23:59:59

微信搜索公众号: Java专栏, 获取最新面试手册

date	3字节	日期	1000-01-01 到 9999-12-31
timestamp	4字节	时间戳	19700101000000 到 2038-01-19 03:14:07
time	3字节	时间	-838:59:59 到 838:59:59
year	1字节	年份	1901 - 2155
datetime	YYYY-MM-DD hh:mm:ss		
timestamp	YY-MM-DD hh:mm:ss YYYYMMDDhhmmss YYMMDDhhmmss YYYYYMMDDhhmmss YYMMDDhhmmss		
date	YYYY-MM-DD YY-MM-DD YYYYMMDD YYMMDD YYYYYMMDD YYMMDD		
time	hh:mm:ss hhmmss hhmmss		
year	YYYY YY YYYYY YY		

4. 枚举和集合

-- 枚举(enum) -----

enum(val1, val2, val3...)

在已知的值中进行单选。最大数量为65535.

枚举值在保存时，以2个字节的整型(smallint)保存。每个枚举值，按保存的位置顺序，从1开始逐一递增。

表现为字符串类型，存储却是整型。

NULL值的索引是NULL。

空字符串错误值的索引值是0。

-- 集合(set) -----

set(val1, val2, val3...)

create table tab (gender set('男', '女', '无'));

insert into tab values ('男', '女');

最多可以有64个不同的成员。以bigint存储，共8个字节。采取位运算的形式。

当创建表时，SET成员值的尾部空格将自动被删除。

列属性 (列约束)

/* 列属性 (列约束) */ -----

1. PRIMARY 主键

- 能唯一标识记录的字段，可以作为主键。
- 一个表只能有一个主键。
- 主键具有唯一性。
- 声明字段时，用 primary key 标识。

也可以在字段列表之后声明

例: create table tab (id int, stu varchar(10), primary key (id));

- 主键字段的值不能为null。
- 主键可以由多个字段共同组成。此时需要在字段列表后声明的方法。

例: create table tab (id int, stu varchar(10), age int, primary key (stu, age));

2. UNIQUE 唯一索引 (唯一约束)

使得某字段的值也不能重复。

3. NULL 约束

`null`不是数据类型，是列的一个属性。

表示当前列是否可以为`null`，表示什么都没有。

`null`，允许为空。默认。

`not null`，不允许为空。

```
insert into tab values (null, 'val');
```

-- 此时表示将第一个字段的值设为`null`，取决于该字段是否允许为`null`

4. DEFAULT 默认值属性

当前字段的默认值。

```
insert into tab values (default, 'val'); -- 此时表示强制使用默认值。
```

```
create table tab ( add_time timestamp default current_timestamp );
```

-- 表示将当前时间的时间戳设为默认值。

```
current_date, current_time
```

5. AUTO_INCREMENT 自动增长约束

自动增长必须为索引（主键或`unique`）

只能存在一个字段为自动增长。

默认为1开始自动增长。可以通过表属性 `auto_increment = x`进行设置，或 `alter table tbl auto_increment = x;`

6. COMMENT 注释

例: `create table tab (id int) comment '注释内容';`

7. FOREIGN KEY 外键约束

用于限制主表与从表数据完整性。

```
alter table t1 add constraint `t1_t2_fk` foreign key (t1_id) references t2(id);
```

-- 将表t1的t1_id外键关联到表t2的id字段。

-- 每个外键都有一个名字，可以通过 `constraint` 指定

存在外键的表，称之为从表（子表），外键指向的表，称之为父表（父表）。

作用：保持数据一致性，完整性，主要目的是控制存储在外键表（从表）中的数据。

MySQL中，可以对InnoDB引擎使用外键约束：

语法：

`foreign key` (外键字段) `references` 主表名 (关联字段) [主表记录删除时的动作] [主表记录更新时的动作]

此时需要检测一个从表的外键需要约束为主表的已存在的值。外键在没有关联的情况下，可以设置为`null`。前提是该外键列，没有`not null`。

可以不指定主表记录更改或更新时的动作，那么此时主表的操作被拒绝。

如果指定了 `on update` 或 `on delete`: 在删除或更新时，有如下几个操作可以选择：

1. `cascade`, 级联操作。主表数据被更新（主键值更新），从表也被更新（外键值更新）。主表记录被删除，从表相关记录也被删除。

2. `set null`, 设置为`null`。主表数据被更新（主键值更新），从表的外键被设置为`null`。主表记录被删除，从表相关记录外键被设置成`null`。但注意，要求该外键列，没有`not null`属性约束。

3. `restrict`, 拒绝父表删除和更新。

注意，外键只被InnoDB存储引擎所支持。其他引擎是不支持的。

建表规范

```
/* 建表规范 */ -----
```

-- Normal Format, NF

- 每个表保存一个实体信息
- 每个具有一个ID字段作为主键
- ID主键 + 原子表

-- 1NF, 第一范式

字段不能再分，就满足第一范式。

-- 2NF, 第二范式

满足第一范式的前提下，不能出现部分依赖。

消除复合主键就可以避免部分依赖。增加单列关键字。

-- 3NF, 第三范式

满足第二范式的前提下，不能出现传递依赖。

某个字段依赖于主键，而有其他字段依赖于该字段。这就是传递依赖。

将一个实体信息的数据放在一个表内实现。

SELECT

```
/* SELECT */ -----
SELECT [ALL|DISTINCT] select_expr FROM -> WHERE -> GROUP BY [合计函数] -> HAVING -
> ORDER BY -> LIMIT
a. select_expr
-- 可以用 * 表示所有字段。
select * from tb;
-- 可以使用表达式（计算公式、函数调用、字段也是个表达式）
select stu, 29+25, now() from tb;
-- 可以为每个列使用别名。适用于简化列标识，避免多个列标识符重复。
- 使用 as 关键字，也可省略 as。
select stu+10 as add10 from tb;
b. FROM 子句
用于标识查询来源。
-- 可以为表起别名。使用as关键字。
SELECT * FROM tb1 AS tt, tb2 AS bb;
-- from子句后，可以同时出现多个表。
-- 多个表会横向叠加到一起，而数据会形成一个笛卡尔积。
SELECT * FROM tb1, tb2;
-- 向优化符提示如何选择索引
USE INDEX、IGNORE INDEX、FORCE INDEX
SELECT * FROM table1 USE INDEX (key1,key2) WHERE key1=1 AND key2=2 AND
key3=3;
SELECT * FROM table1 IGNORE INDEX (key3) WHERE key1=1 AND key2=2 AND
key3=3;
c. WHERE 子句
-- 从from获得的数据源中进行筛选。
-- 整型1表示真，0表示假。
-- 表达式由运算符和运算数组成。
-- 运算数：变量（字段）、值、函数返回值
-- 运算符：
=, <=>, <>, !=, <=, <, >=, >, !, &&, ||
in (not) null, (not) like, (not) in, (not) between and, is (not),
and, or, not, xor
is/is not 加上ture/false/unknown, 检验某个值的真假
<=>与<>功能相同，<=>可用于null比较
d. GROUP BY 子句, 分组子句
GROUP BY 字段/别名 [排序方式]
分组后会进行排序。升序: ASC, 降序: DESC
以下[合计函数]需配合 GROUP BY 使用:
count 返回不同的非NULL值数目 count(*)、count(字段)
sum 求和
max 求最大值
min 求最小值
avg 求平均值
group_concat 返回带有来自一个组的连接的非NULL值的字符串结果。组内字符串连接。
```

e. HAVING 子句, 条件子句

与 **where** 功能、用法相同, 执行时机不同。

where 在开始时执行检测数据, 对原数据进行过滤。

having 对筛选出的结果再次进行过滤。

having 字段必须是查询出来的, **where** 字段必须是数据表存在的。

where 不可以使用字段的别名, **having** 可以。因为执行**WHERE**代码时, 可能尚未确定列值。

where 不可以使用合计函数。一般需用合计函数才会用 **having**

SQL标准要求**HAVING**必须引用**GROUP BY**子句中的列或用于合计函数中的列。

f. ORDER BY 子句, 排序子句

order by 排序字段/别名 排序方式 [,排序字段/别名 排序方式] ...

升序: **ASC**, 降序: **DESC**

支持多个字段的排序。

g. LIMIT 子句, 限制结果数量子句

仅对处理好的结果进行数量限制。将处理好的结果看作是一个集合, 按照记录出现的顺序, 索引从0开始。

limit 起始位置, 获取条数

省略第一个参数, 表示从索引0开始。**limit** 获取条数

h. DISTINCT, ALL 选项

distinct 去除重复记录

默认为 **all**, 全部记录

UNION

/* UNION */ -----

将多个**select**查询的结果组合成一个结果集合。

SELECT ... UNION [ALL|DISTINCT] SELECT ...

默认 **DISTINCT** 方式, 即所有返回的行都是唯一的

建议, 对每个**SELECT**查询加上小括号包裹。

ORDER BY 排序时, 需加上 **LIMIT** 进行结合。

需要各**select**查询的字段数量一样。

每个**select**查询的字段列表(数量、类型)应一致, 因为结果中的字段名以第一条**select**语句为准。

子查询

/* 子查询 */ -----

- 子查询需用括号包裹。

-- from型

from后要求是一个表, 必须给子查询结果取个别名。

- 简化每个查询内的条件。

- **from**型需将结果生成一个临时表格, 可用以原表的锁定的释放。

- 子查询返回一个表, 表型子查询。

select * from (select * from tb where id>0) as subfrom where id>1;

-- where型

- 子查询返回一个值, 标量子查询。

- 不需要给子查询取别名。

- **where**子查询内的表, 不能直接用以更新。

select * from tb where money = (select max(money) from tb);

-- 列子查询

如果子查询结果返回的是一列。

使用 **in** 或 **not in** 完成查询

`exists` 和 `not exists` 条件

如果子查询返回数据，则返回1或0。常用于判断条件。

```
select column1 from t1 where exists (select * from t2);
```

-- 行子查询

查询条件是一个行。

```
select * from t1 where (id, gender) in (select id, gender from t2);
```

行构造符：(col1, col2, ...) 或 ROW(col1, col2, ...)

行构造符通常用于与对能返回两个或两个以上列的子查询进行比较。

-- 特殊运算符

`!= all()` 相当于 `not in`

`= some()` 相当于 `in`。`any` 是 `some` 的别名

`!= some()` 不等同于 `not in`，不等于其中某一个。

`all, some` 可以配合其他运算符一起使用。

连接查询 (join)

```
/* 连接查询(join) */ -----
```

将多个表的字段进行连接，可以指定连接条件。

-- 内连接(inner join)

- 默认就是内连接，可省略inner。

- 只有数据存在时才能发送连接。即连接结果不能出现空行。

`on` 表示连接条件。其条件表达式与`where`类似。也可以省略条件（表示条件永远为真）

也可用`where`表示连接条件。

还有 `using`，但需字段名相同。 `using(字段名)`

-- 交叉连接 cross join

即，没有条件的内连接。

```
select * from tb1 cross join tb2;
```

-- 外连接(outer join)

- 如果数据不存在，也会出现在连接结果中。

-- 左外连接 left join

如果数据不存在，左表记录会出现，而右表为null填充

-- 右外连接 right join

如果数据不存在，右表记录会出现，而左表为null填充

-- 自然连接(natural join)

自动判断连接条件完成连接。

相当于省略了`using`，会自动查找相同字段名。

`natural join`

`natural left join`

`natural right join`

```
select info.id, info.name, info.stu_num, extra_info.hobby, extra_info.sex from
info, extra_info where info.stu_num = extra_info.stu_id;
```

TRUNCATE

```
/* TRUNCATE */ -----
TRUNCATE [TABLE] tbl_name
清空数据
删除重建表
区别:
1, truncate 是删除表再创建, delete 是逐条删除
2, truncate 重置auto_increment的值。而delete不会
3, truncate 不知道删除了几条, 而delete知道。
4, 当被用于带分区的表时, truncate 会保留分区
```

备份与还原

```
/* 备份与还原 */ -----
备份, 将数据的结构与表内数据保存起来。
利用 mysqldump 指令完成。
-- 导出
mysqldump [options] db_name [tables]
mysqldump [options] --database DB1 [DB2 DB3...]
mysqldump [options] --all--database
1. 导出一张表
mysqldump -u用户名 -p密码 库名 表名 > 文件名(D:/a.sql)
2. 导出多张表
mysqldump -u用户名 -p密码 库名 表1 表2 表3 > 文件名(D:/a.sql)
3. 导出所有表
mysqldump -u用户名 -p密码 库名 > 文件名(D:/a.sql)
4. 导出一个库
mysqldump -u用户名 -p密码 --lock-all-tables --database 库名 > 文件名(D:/a.sql)
可以-w携带WHERE条件
-- 导入
1. 在登录mysql的情况下:
source 备份文件
2. 在不登录的情况下
mysql -u用户名 -p密码 库名 < 备份文件
```

视图

什么是视图:

视图是一个虚拟表, 其内容由查询定义。同真实的表一样, 视图包含一系列带有名称的列和行数据。但是, 视图并不在数据库中以存储的数据值集形式存在。行和列数据来自由定义视图的查询所引用的表, 并且在引用视图时动态生成。

视图具有表结构文件, 但不存在数据文件。

对其中所引用的基础表来说, 视图的作用类似于筛选。定义视图的筛选可以来自当前或其它数据库的一个或多个表, 或者其它视图。通过视图进行查询没有任何限制, 通过它们进行数据修改时的限制也很少。

视图是存储在数据库中的查询的sql语句, 它主要出于两种原因: 安全原因, 视图可以隐藏一些数据, 如: 社会保险基金表, 可以用视图只显示姓名, 地址, 而不显示社会保险号和工资数等, 另一原因是可使复杂的查询易于理解和使用。

-- 创建视图

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW view_name
[(column_list)] AS select_statement
```

- 视图名必须唯一，同时不能与表重名。
 - 视图可以使用**select**语句查询到的列名，也可以自己指定相应的列名。
 - 可以指定视图执行的算法，通过**ALGORITHM**指定。
 - **column_list**如果存在，则数目必须等于**SELECT**语句检索的列数
- 查看结构
`SHOW CREATE VIEW view_name`
- 删除视图
 - 删除视图后，数据依然存在。
 - 可同时删除多个视图。`DROP VIEW [IF EXISTS] view_name ...`
- 修改视图结构
 - 一般不修改视图，因为不是所有的更新视图都会映射到表上。`ALTER VIEW view_name [(column_list)] AS select_statement`
- 视图作用
 1. 简化业务逻辑
 2. 对客户端隐藏真实的表结构
- 视图算法(**ALGORITHM**)
- MERGE** 合并
将视图的查询语句，与外部查询需要先合并再执行！
- TEMPORARY** 临时表
将视图执行完毕后，形成临时表，再做外层查询！
- UNDEFINED** 未定义(默认)，指的是MySQL自主去选择相应的算法。

事务

- 事务是指逻辑上的一组操作，组成这组操作的各个单元，要不全成功要不全失败。
- 支持连续SQL的集体成功或集体撤销。
 - 事务是数据库在数据完整性方面的一个功能。
 - 需要利用 **InnoDB** 或 **BDB** 存储引擎，对自动提交的特性支持完成。
 - **InnoDB**被称为事务安全型引擎。
- 事务开启
`START TRANSACTION;` 或者 `BEGIN;`
开启事务后，所有被执行的SQL语句均被认作当前事务内的SQL语句。
- 事务提交
`COMMIT;`
- 事务回滚
`ROLLBACK;`
如果部分操作发生问题，映射到事务开启前。
- 事务的特性
 1. 原子性 (**Atomicity**)
事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
 2. 一致性 (**Consistency**)
事务前后数据的完整性必须保持一致。
 - 事务开始和结束时，外部数据一致
 - 在整个事务过程中，操作是连续的
 3. 隔离性 (**Isolation**)
多个用户并发访问数据库时，一个用户的事务不能被其它用户的事物所干扰，多个并发事务之间的数据要相互隔离。
 4. 持久性 (**Durability**)
一个事务一旦被提交，它对数据库中的数据改变就是永久性的。
- 事务的实现
 1. 要求是事务支持的表类型

2. 执行一组相关操作前开启事务
3. 整组操作完成后，都成功，则提交；如果存在失败，选择回滚，则会回到事务开始的备份点。

-- 事务的原理

利用InnoDB的自动提交(autocommit)特性完成。

普通的MySQL执行语句后，当前的数据提交操作均可被其他客户端可见。

而事务是暂时关闭“自动提交”机制，需要commit提交持久化数据操作。

-- 注意

1. 数据定义语言(DDL)语句不能被回滚，比如创建或取消数据库的语句，和创建、取消或更改表或存储的子程序的语句。
2. 事务不能被嵌套

-- 保存点

SAVEPOINT 保存点名称 -- 设置一个事务保存点

ROLLBACK TO SAVEPOINT 保存点名称 -- 回滚到保存点

RELEASE SAVEPOINT 保存点名称 -- 删除保存点

-- InnoDB自动提交特性设置

SET autocommit = 0|1; 0表示关闭自动提交，1表示开启自动提交。

- 如果关闭了，那普通操作的结果对其他客户端也不可见，需要commit提交后才能持久化数据操作。

- 也可以关闭自动提交来开启事务。但与START TRANSACTION不同的是，

SET autocommit是永久改变服务器的设置，直到下次再次修改该设置。(针对当前连接)

而START TRANSACTION记录开启前的状态，而一旦事务提交或回滚后就需要再次开启事务。(针对当前事务)

锁表

/* 锁表 */

表锁定只用于防止其它客户端进行不正当地读取和写入

MyISAM 支持表锁，InnoDB 支持行锁

-- 锁定

LOCK TABLES tbl_name [AS alias]

-- 解锁

UNLOCK TABLES

触发器

/* 触发器 */ -----

触发程序是与表有关的命名数据库对象，当该表出现特定事件时，将激活该对象

监听：记录的增加、修改、删除。

-- 创建触发器

CREATE TRIGGER trigger_name trigger_time trigger_event ON tbl_name FOR EACH ROW
trigger_stmt

参数：

trigger_time是触发程序的动作时间。它可以是 before 或 after，以指明触发程序是在激活它的语句之前或之后触发。

trigger_event指明了激活触发程序的语句的类型

INSERT：将新行插入表时激活触发程序

UPDATE：更改某一行时激活触发程序

DELETE：从表中删除某一行时激活触发程序

tbl_name：监听的表，必须是永久性的表，不能将触发程序与TEMPORARY表或视图关联起来。

trigger_stmt：当触发程序激活时执行的语句。执行多个语句，可使用BEGIN...END复合语句结构

-- 删除

```

DROP TRIGGER [schema_name.]trigger_name
可以使用old和new代替旧的和新的数据
    更新操作，更新前是old，更新后是new.
    删除操作，只有old.
    增加操作，只有new.

-- 注意
    1. 对于具有相同触发程序动作时间和事件的给定表，不能有两个触发程序。

-- 字符连接函数
concat(str1,str2,...])
concat_ws(separator,str1,str2,...)

-- 分支语句
if 条件 then
    执行语句
elseif 条件 then
    执行语句
else
    执行语句
end if;

-- 修改最外层语句结束符
delimiter 自定义结束符号
    SQL语句
自定义结束符号
delimiter ;      -- 修改回原来的分号

-- 语句块包裹
begin
    语句块
end

-- 特殊的执行
1. 只要添加记录，就会触发程序。
2. Insert into on duplicate key update 语法会触发：
    如果没有重复记录，会触发 before insert, after insert;
    如果有重复记录并更新，会触发 before insert, before update, after update;
    如果有重复记录但是没有发生更新，则触发 before insert, before update
3. Replace 语法 如果有记录，则执行 before insert, before delete, after delete, after
insert

```

SQL编程

```

/* SQL编程 */
--// 局部变量
-- 变量声明
declare var_name[,...] type [default value]
    这个语句被用来声明局部变量。要给变量提供一个默认值，请包含一个default子句。值可以被指定为一个表达式，不需要为一个常数。如果没有default子句，初始值为null。
-- 赋值
    使用 set 和 select into 语句为变量赋值。
    - 注意：在函数内是可以使用全局变量（用户自定义的变量）

--// 全局变量
-- 定义、赋值
set 语句可以定义并为变量赋值。
set @var = value;
    也可以使用select into语句为变量初始化并赋值。这样要求select语句只能返回一行，但是可以是多个字段，就意味着同时为多个变量进行赋值，变量的数量需要与查询的列数一致。

```

还可以把赋值语句看作一个表达式，通过**select**执行完成。此时为了避免=被当作关系运算符看待，使用:=代替。(**set**语句可以使用= 和 :=)。

```
select @var:=20;
select @v1:=id, @v2=name from t1 limit 1;
select * from tbl_name where @var:=30;
select into 可以将表中查询获得的数据赋给变量。
    -| select max(height) into @max_height from tb;
-- 自定义变量名

为了避免select语句中，用户自定义的变量与系统标识符（通常是字段名）冲突，用户自定义变量在变量名前使用@作为开始符号。
```

```
@var=10;
    - 变量被定义后，在整个会话周期都有效（登录到退出）
--// 控制结构 -----
-- if语句
if search_condition then
    statement_list
[elseif search_condition then
    statement_list]
...
[else
    statement_list]
end if;
-- case语句
CASE value WHEN [compare-value] THEN result
[WHEN [compare-value] THEN result ...]
[ELSE result]
END
-- while循环
[begin_label:] while search_condition do
    statement_list
end while [end_label];
    - 如果需要在循环内提前终止 while循环，则需要使用标签；标签需要成对出现。
    -- 退出循环
        退出整个循环 leave
        退出当前循环 iterate
        通过退出的标签决定退出哪个循环
--// 内置函数 -----
-- 数值函数
abs(x)          -- 绝对值 abs(-10.9) = 10
format(x, d)    -- 格式化千分位数值 format(1234567.456, 2) = 1,234,567.46
ceil(x)         -- 向上取整 ceil(10.1) = 11
floor(x)        -- 向下取整 floor (10.1) = 10
round(x)         -- 四舍五入去整
mod(m, n)       -- m%n m mod n 求余 10%3=1
pi()            -- 获得圆周率
pow(m, n)       -- m^n
sqrt(x)         -- 算术平方根
rand()          -- 随机数
truncate(x, d) -- 截取d位小数
-- 时间日期函数
now(), current_timestamp();      -- 当前日期时间
current_date();                  -- 当前日期
current_time();                  -- 当前时间
date('yyyy-mm-dd hh:ii:ss');    -- 获取日期部分
time('yyyy-mm-dd hh:ii:ss');    -- 获取时间部分
date_format('yyyy-mm-dd hh:ii:ss', '%d %y %a %d %m %b %j'); -- 格式化时间
unix_timestamp();                -- 获得unix时间戳
from_unixtime();                 -- 从时间戳获得时间
```

```

-- 字符串函数
length(string)          -- string长度, 字节
char_length(string)      -- string的字符个数
substring(str, position [,length])    -- 从str的position开始, 取length个字符
replace(str ,search_str ,replace_str)  -- 在str中用replace_str替换search_str
instr(string ,substring)   -- 返回substring首次在string中出现的位置
concat(string [,...])     -- 连接字串
charset(str)             -- 返回字串字符集
lcase(string)            -- 转换成小写
left(string, length)     -- 从string2中的左边起取length个字符
load_file(file_name)     -- 从文件读取内容
locate(substring, string [,start_position]) -- 同instr, 但可指定开始位置
lpad(string, length, pad) -- 重复用pad加在string开头, 直到字串长度为length
ltrim(string)            -- 去除前端空格
repeat(string, count)    -- 重复count次
rpad(string, length, pad) -- 在str后用pad补充, 直到长度为length
rtrim(string)            -- 去除后端空格
strcmp(string1 ,string2)  -- 逐字符比较两字串大小

-- 流程函数
case when [condition] then result [when [condition] then result ...] [else
result] end 多分支
if(expr1,expr2,expr3) 双分支。
-- 聚合函数
count()
sum();
max();
min();
avg();
group_concat()
-- 其他常用函数
md5();
default();
--// 存储函数, 自定义函数 -----
-- 新建
CREATE FUNCTION function_name (参数列表) RETURNS 返回值类型
    函数体
    - 函数名, 应该合法的标识符, 并且不应该与已有的关键字冲突。
    - 一个函数应该属于某个数据库, 可以使用db_name.funciton_name的形式执行当前函数所属数据
库, 否则为当前数据库。
    - 参数部分, 由"参数名"和"参数类型"组成。多个参数用逗号隔开。
    - 函数体由多条可用的mysql语句, 流程控制, 变量声明等语句构成。
    - 多条语句应该使用 begin...end 语句块包含。
    - 一定要有 return 返回值语句。
-- 删除
DROP FUNCTION [IF EXISTS] function_name;
-- 查看
SHOW FUNCTION STATUS LIKE 'partten'
SHOW CREATE FUNCTION function_name;
-- 修改
ALTER FUNCTION function_name 函数选项
--// 存储过程, 自定义功能 -----
-- 定义
存储存储过程 是一段代码(过程), 存储在数据库中的sql组成。
一个存储过程通常用于完成一段业务逻辑, 例如报名, 交班费, 订单入库等。
而一个函数通常专注与某个功能, 视为其他程序服务的, 需要在其他语句中调用函数才可以, 而存储过程不能
被其他调用, 是自己执行 通过call执行。
-- 创建
CREATE PROCEDURE sp_name (参数列表)

```

过程体

参数列表：不同于函数的参数列表，需要指明参数类型

IN，表示输入型

OUT，表示输出型

INOUT，表示混合型

注意，没有返回值。

存储过程

```
/* 存储过程 */ -----
```

存储过程是一段可执行性代码的集合。相比函数，更偏向于业务逻辑。

调用：**CALL** 过程名

-- 注意

- 没有返回值。
- 只能单独调用，不可夹杂在其他语句中

-- 参数

IN|OUT|INOUT 参数名 数据类型

IN 输入：在调用过程中，将数据输入到过程体内部的参数

OUT 输出：在调用过程中，将过程体处理完的结果返回到客户端

INOUT 输入输出：既可输入，也可输出

-- 语法

CREATE PROCEDURE 过程名 (参数列表)

BEGIN

 过程体

END

用户和权限管理

```
/* 用户和权限管理 */ -----
```

-- root密码重置

1. 停止MySQL服务

2. [Linux] /usr/local/mysql/bin/safe_mysqld --skip-grant-tables &
[Windows] mysqld --skip-grant-tables

3. use mysql;

4. UPDATE `user` SET PASSWORD=PASSWORD("密码") WHERE `user` = "root";

5. FLUSH PRIVILEGES;

用户信息表：mysql.user

-- 刷新权限

FLUSH PRIVILEGES;

-- 增加用户

CREATE USER 用户名 **IDENTIFIED BY** [PASSWORD] 密码(字符串)

- 必须拥有mysql数据库的全局CREATE USER权限，或拥有INSERT权限。

- 只能创建用户，不能赋予权限。

- 用户名，注意引号：如 '**user_name**'@'192.168.1.1'

- 密码也需引号，纯数字密码也要加引号

- 要在纯文本中指定密码，需忽略PASSWORD关键词。要把密码指定为由PASSWORD()函数返回的混编值，需包含关键字PASSWORD

-- 重命名用户

RENAME USER old_user **TO** new_user

-- 设置密码

```

SET PASSWORD = PASSWORD('密码') -- 为当前用户设置密码
SET PASSWORD FOR 用户名 = PASSWORD('密码') -- 为指定用户设置密码
-- 删除用户
DROP USER 用户名
-- 分配权限/添加用户
GRANT 权限列表 ON 表名 TO 用户名 [IDENTIFIED BY [PASSWORD] 'password']
  - all privileges 表示所有权限
  - *.* 表示所有库的所有表
  - 库名.表名 表示某库下面的某表
  GRANT ALL PRIVILEGES ON `pms`.* TO 'pms'@'%' IDENTIFIED BY 'pms0817';
-- 查看权限
SHOW GRANTS FOR 用户名
  -- 查看当前用户权限
  SHOW GRANTS; 或 SHOW GRANTS FOR CURRENT_USER; 或 SHOW GRANTS FOR CURRENT_USER();
-- 撤消权限
REVOKE 权限列表 ON 表名 FROM 用户名
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 用户名 -- 撤销所有权限
-- 权限层级
-- 要使用GRANT或REVOKE，您必须拥有GRANT OPTION权限，并且您必须用于您正在授予或撤销的权限。
全局层级：全局权限适用于一个给定服务器中的所有数据库，mysql.user
  GRANT ALL ON *.*和 REVOKE ALL ON *.*只授予和撤销全局权限。
数据库层级：数据库权限适用于一个给定数据库中的所有目标，mysql.db, mysql.host
  GRANT ALL ON db_name.*和REVOKE ALL ON db_name.*只授予和撤销数据库权限。
表层级：表权限适用于一个给定表中的所有列，mysql.tables_priv
  GRANT ALL ON db_name.tbl_name和REVOKE ALL ON db_name.tbl_name只授予和撤销表权限。
列层级：列权限适用于一个给定表中的单一列，mysql.columns_priv
  当使用REVOKE时，您必须指定与被授权列相同的列。
-- 权限列表
ALL [PRIVILEGES] -- 设置除GRANT OPTION之外的所有简单权限
ALTER -- 允许使用ALTER TABLE
ALTER ROUTINE -- 更改或取消已存储的子程序
CREATE -- 允许使用CREATE TABLE
CREATE ROUTINE -- 创建已存储的子程序
CREATE TEMPORARY TABLES -- 允许使用CREATE TEMPORARY TABLE
CREATE USER -- 允许使用CREATE USER, DROP USER, RENAME USER和REVOKE ALL PRIVILEGES。
CREATE VIEW -- 允许使用CREATE VIEW
DELETE -- 允许使用DELETE
DROP -- 允许使用DROP TABLE
EXECUTE -- 允许用户运行已存储的子程序
FILE -- 允许使用SELECT...INTO OUTFILE和LOAD DATA INFILE
INDEX -- 允许使用CREATE INDEX和DROP INDEX
INSERT -- 允许使用INSERT
LOCK TABLES -- 允许对您拥有SELECT权限的表使用LOCK TABLES
PROCESS -- 允许使用SHOW FULL PROCESSLIST
REFERENCES -- 未被实施
RELOAD -- 允许使用FLUSH
REPLICATION CLIENT -- 允许用户询问从属服务器或主服务器的地址
REPLICATION SLAVE -- 用于复制型从属服务器（从主服务器中读取二进制日志事件）
SELECT -- 允许使用SELECT
SHOW DATABASES -- 显示所有数据库
SHOW VIEW -- 允许使用SHOW CREATE VIEW
SHUTDOWN -- 允许使用mysqladmin shutdown
SUPER -- 允许使用CHANGE MASTER, KILL, PURGE MASTER LOGS和SET GLOBAL语句,
        mysqladmin debug命令；允许您连接（一次），即使已达到max_connections。
UPDATE -- 允许使用UPDATE

```

```
USAGE    -- “无权限”的同义词
GRANT OPTION      -- 允许授予权限
```

表维护

```
/* 表维护 */
-- 分析和存储表的关键字分布
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE 表名 ...
-- 检查一个或多个表是否有错误
CHECK TABLE tbl_name [,tbl_name] ... [option] ...
option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
-- 整理数据文件的碎片
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [,tbl_name] ...
```

杂项

```
/* 杂项 */
1. 可用反引号(`)为标识符(库名、表名、字段名、索引、别名)包裹，以避免与关键字重名！中文也可以作为标识符！
2. 每个库目录存在一个保存当前数据库的选项文件db.opt。
3. 注释：
   单行注释 # 注释内容
   多行注释 /* 注释内容 */
   单行注释 -- 注释内容      (标准SQL注释风格，要求双破折号后加一空格符(空格、TAB、换行等))
4. 模式通配符：
   _ 任意单个字符
   % 任意多个字符，甚至包括零字符
   单引号需要进行转义 \''
5. CMD命令行内的语句结束符可以为 ";"，"\G"，"\g"，仅影响显示结果。其他地方还是用分号结束。
delimiter 可修改当前对话的语句结束符。
6. SQL对大小写不敏感
7. 清除已有语句：\c
```

Redis面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

微信搜索公众号：Java专栏，获取最新面试手册

1. 什么是 Redis? 简述它的优缺点?

Redis 的全称是: Remote Dictionary.Server, 本质上是一个 Key-Value 类型的内存数据库, 很像 memcached, 整个数据库统统加载在内存当中进行操作, 定期通过异步操作把数据库数据 flush 到硬盘上进行保存。

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 DB。

Redis 的出色之处不仅仅是性能, Redis 最大的魅力是支持保存多种数据结构, 此外单个 value 的最大限制是 1GB, 不像 memcached 只能保存 1MB 的数据, 因此 Redis 可以用来实现很多有用的功能。

比方说用他的 List 来做 FIFO 双向链表, 实现一个轻量级的高性能消息队列服务, 用他的 Set 可以做高性能的 tag 系统等等。

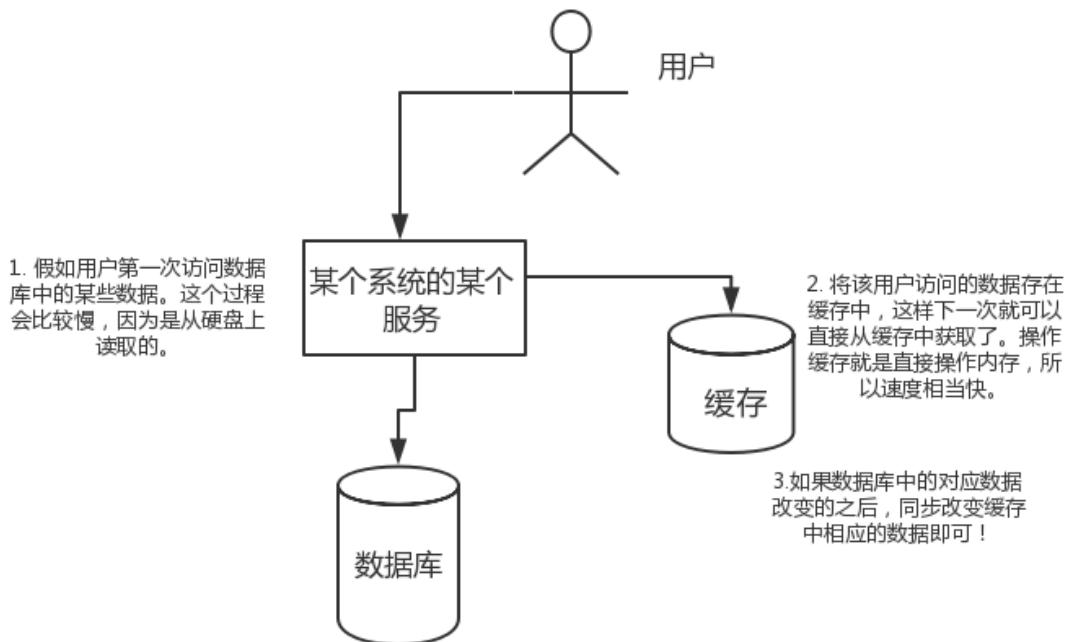
另外 Redis 也可以对存入的 Key-Value 设置 expire 时间, 因此也可以被当作一个功能加强版的 memcached 来用。Redis 的主要缺点是数据库容量受到物理内存的限制, 不能用作海量数据的高性能读写, 因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

2. 为什么要用 redis/为什么要用缓存

主要从“高性能”和“高并发”这两点来看待这个问题。

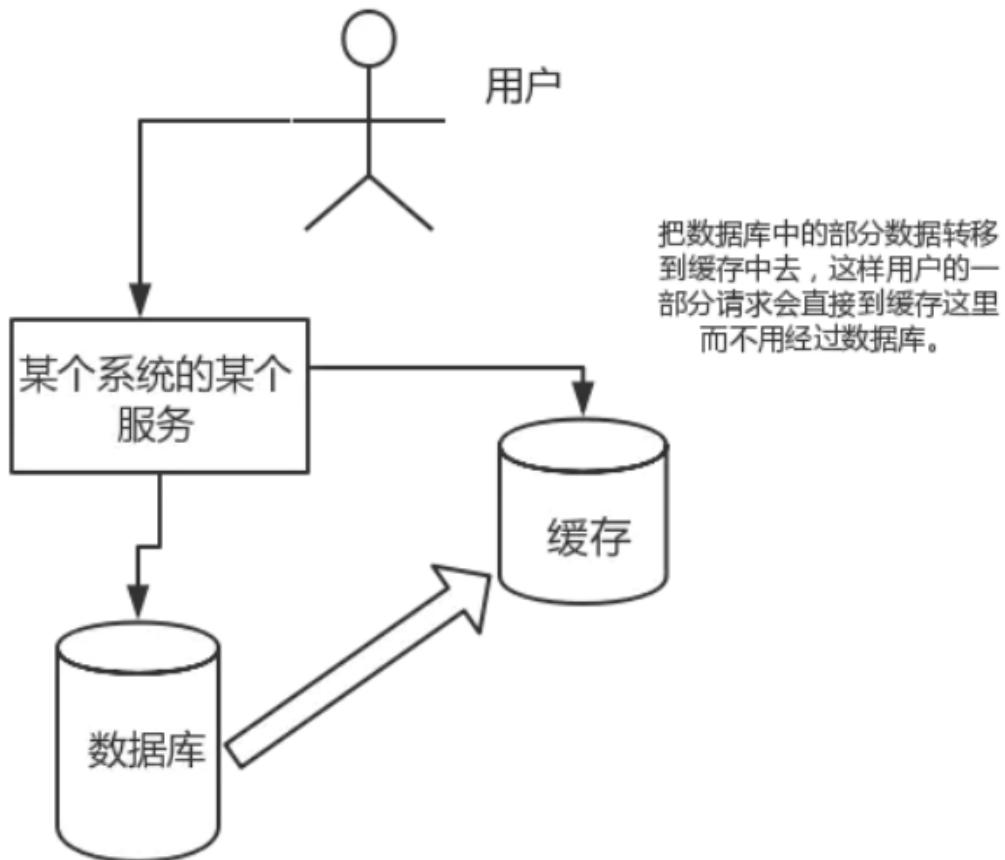
1、高性能:

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!



2、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。



2. Redis 与 memcached 相比有哪些优势？

- 1、redis支持更丰富的数据类型（支持更复杂的应用场景）：Redis不仅仅支持简单的k/v类型的数据，同时还提供list, set, zset, hash等数据结构的存储。memcache支持简单数据类型，String。
- 2、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而Memecache把数据全部存在内存之中。
- 3、集群模式：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是redis 目前是原生支持 cluster 模式的。
- 4、Memcached是多线程，非阻塞IO复用的网络模型；Redis使用单线程的多路 IO 复用模型。

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

3. Redis 支持哪几种数据类型？

String、List、Set、Sorted Set、hashes

4. Redis 主要消耗什么物理资源？

内存。

5. Redis 有哪几种数据淘汰策略？

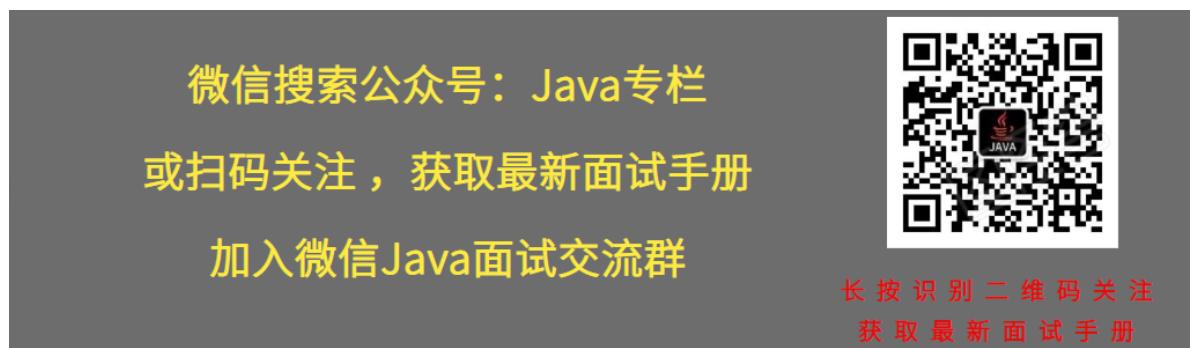
- 1.noeviction: 返回错误当内存限制达到，并且客户端尝试执行会让更多内存被使用的命令。
- 2.allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。
- 3.volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键，使得新添加的数据有空间存放。
- 4.allkeys-random: 回收随机的键使得新添加的数据有空间存放。
- 5.volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
- 6.volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键，使得新添加的数据有空间存放。

6. Redis 官方为什么不提供 Windows 版本？

因为目前 Linux 版本已经相当稳定，而且用户量很大，无需开发 windows 版本，反而会带来兼容性等问题。

7. 一个字符串类型的值能存储最大容量是多少？

512M



8. 为什么 Redis 需要把所有数据放到内存中？

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。

所以 redis 具有快速和数据持久化的特征，如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。

在内存越来越便宜的今天，redis 将会越来越受欢迎，如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

9. 如何保证缓存与数据库双写时的数据一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的
问题，那么你如何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔
有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以
保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个
请求。

10. Redis中数据库默认是多少个db即作用？

Redis默认支持16个数据库，可以通过配置databases来修改这一数字。客户端与Redis建立连接后会自动选择0号数据库，不过可以随时使用select命令更换数据库。

Redis支持多个数据库，并且每个数据库是隔离的不能共享，并且基于单机才有，如果是集群就没有数据
库的概念。

11. Redis 集群方案应该怎么做？都有哪些方案？

- 1、codis
- 2、目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在节点数量改变情况下，旧节
点数据可恢复到新 hash 节点。

redis cluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

3、在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对 key 进行 hash 计算，然后去对应的 redis 实例操作数据。这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控等等。

12. Redis 集群方案什么情况下会导致整个集群不可用？

有 A, B, C 三个节点的集群，在没有复制模型的情况下，如果节点 B 失败了，那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

13. MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？

redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

14. Redis 有哪些适合的场景？

1、会话缓存 (Session Cache)

最常用的一种使用 Redis 的情景是会话缓存 (sessioncache)，用 Redis 缓存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。

2、全页缓存 (FPC)

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。

此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

3、队列

Redis 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。

如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

4、排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合 (Set) 和有序集合 (SortedSet) 也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。

所以，我们要从排序集合中获取到排名最靠前的 10 个用户-我们称之为“user_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user_scores 0 10 WITHSCORES Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

5、发布/订阅

最后（但肯定不是最重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！

15. 说说 Redis 哈希槽的概念？

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

16. Redis 集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型，每个节点都会有 N-1 个复制品。

17. Redis 集群会有写操作丢失吗？为什么？

Redis 并不能保证数据的强一致性，这意味着在实际中集群在特定的条件下可能会丢失写操作。

18. Redis 集群之间是如何复制的？

异步复制

19. Redis 集群最大节点个数是多少？

16384 个

20. Redis 集群如何选择数据库？

Redis 集群目前无法做数据库选择，默认在 0 数据库。

21. Redis 中的管道有什么用？

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应，这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多 POP3 协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。

22. 怎么理解 Redis 事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行，事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

23. Redis 事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH、UNWATCH

- 1、DISCARD用来取消一个事务；
- 2、EXEC用来执行一个事务；
- 3、MULTI用来组装一个事务；
- 4、WATCH用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行
- 5、WATCH取消 WATCH 命令对所有 key 的监视。

24. Redis key 的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令

expire 指令可以设置 key 的超时时间，单位秒。即在多少秒后过期。返回1代表设置成功；返回 0 代表设置不成功，此时是因为key不存在导致的。

```
127.0.0.1:6379> expire "key-aaa" 10
(integer) 1
127.0.0.1:6379> expire "key-not-exists" 10
(integer) 0
```

3 秒后使用 ttl 命令查询剩余的超时时间：

```
127.0.0.1:6379> ttl "key-aaa"
(integer) 7
```

使用 persist 清除过期时间

```
127.0.0.1:6379> set "key-aaa" "value-bbb" EX 15
OK
127.0.0.1:6379> ttl "key-aaa"
(integer) 11
127.0.0.1:6379> persist "key-aaa"
(integer) 1
127.0.0.1:6379> ttl "key-aaa"
(integer) -1
127.0.0.1:6379> get "key-aaa"
"value-bbb"
```

persist 返回值：

1：成功清理过期时间。

0：key 不存在，或者没有设置过期时间

25. Redis 如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。

26. Redis 回收进程如何工作的？

一个客户端运行了新的命令，添加了新的数据。Redis 检查内存使用情况，如果大于 maxmemory 的限制，则根据设定好的策略进行回收。一个新的命令被执行，等等。

所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地收回回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

27. 加锁机制

如果该客户端面对的是一个 redis cluster 集群，他首先会根据 hash 节点选择一台机器。这里注意，仅仅只是选择一台机器！这点很关键！紧接着，就会发送一段 lua 脚本到 redis 上。

为啥要用 lua 脚本呢？因为一大坨复杂的业务逻辑，可以通过封装在 lua 脚本中发送给 redis，保证这段复杂业务逻辑执行的原子性。

那么，这段 lua 脚本是什么意思呢？这里 KEYS[1]代表的是你加锁的那个 key，比如说：RLock lock = redisson.getLock("myLock");这里你自己设置了加锁的那个锁 key 就是“myLock”。

ARGV[1]代表的就是锁 key 的默认生存时间，默认 30 秒。ARGV[2]代表的是加锁的客户端的 ID，类似于下面这样：8743c9c0-0795-4907-87fd-6c719a6b4586:1给大家解释一下，第一段 if 判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁 key 不存在的话，你就进行加锁。如何加锁呢？很简单，用下面的命令：hset myLock8743c9c0-0795-4907-87fd-6c719a6b4586:1 1，通过这个命令设置一个 hash 数据结构，这行命令执行后，会出现一个类似下面的数据结构：

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁 key 完成了加锁。接着会执行“pexpire myLock 30000”命令，设置 myLock 这个锁 key 的生存时间是 30 秒。好了，到此为止，ok，加锁完成了。

28. 锁互斥机制

那么在这个时候，如果客户端 2 来尝试加锁，执行了同样的一段 lua 脚本，会咋样呢？很简单，第一个 if 判断会执行“exists myLock”，发现 myLock 这个锁 key 已经存在了。接着第二个 if 判断，判断一下，myLock 锁 key 的 hash 数据结构中，是否包含客户端 2 的 ID，但是明显不是的，因为那里包含的是客户端 1 的 ID。

所以，客户端 2 会获取到 ttl myLock 返回的一个数字，这个数字代表了 myLock 这个锁 key 的剩余生存时间。比如还剩 15000 毫秒的生存时间。此时客户端 2 会进入一个 while 循环，不停的尝试加锁。

29. watch dog 自动延期机制

客户端 1 加锁的锁 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢？

简单！只要客户端 1 一旦加锁成功，就会启动一个 watch dog 看门狗，他是一个后台线程，会每隔 10 秒检查一下，如果客户端 1 还持有锁 key，那么就会不断的延长锁 key 的生存时间。

30. 使用过 Redis 做异步队列么，你是怎么用的？有什么缺点？**

般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候，要适当 sleep 一会再重试。

缺点：

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 rabbitmq 等。

能不能生产一次消费多次呢？

使用 pub/sub 主题订阅者模式，可以实现 1:N 的消息队列。

31. 什么是缓存穿透？如何避免？什么是缓存雪崩？何如避免？

缓存穿透

一般的缓存系统，都是按照 key 去缓存查询，如果不存在对应的 value，就应该去后端系统查找（比如 DB）。一些恶意的请求会故意查询不存在的 key，请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

如何避免？

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该 key 对应的数据 insert 了之后清理缓存。

2：对一定不存在的 key 进行过滤。可以把所有的可能存在的 key 放到一个大的 Bitmap 中，查询时通过该 bitmap 过滤。

缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

如何避免？

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。

2：做二级缓存，A1 为原始缓存，A2 为拷贝缓存，A1 失效时，可以访问 A2，A1 缓存失效时间设置为短期，A2 设置为长期

3：不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀

32. 缓存穿透、缓存击穿、缓存雪崩解决方案？

缓存穿透：

指查询一个一定不存在的数据，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到 DB 去查询，可能导致 DB 挂掉。

解决方案：

1. 查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短；

2. 布隆过滤器：将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对 DB 的查询。

缓存击穿：

对于设置了过期时间的 key，缓存在某个时间点过期的时候，恰好这时间点对这个 Key 有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把 DB 压垮。

解决方案：

1. 使用互斥锁：当缓存失效时，不立即去 load db，先使用如 Redis 的 setnx 去设置一个互斥锁，当操作成功返回时再进行 load db 的操作并回设缓存，否则重试 get 缓存的方法。

2. 永远不过期：物理不过期，但逻辑过期（后台异步线程去刷新）。

缓存雪崩：

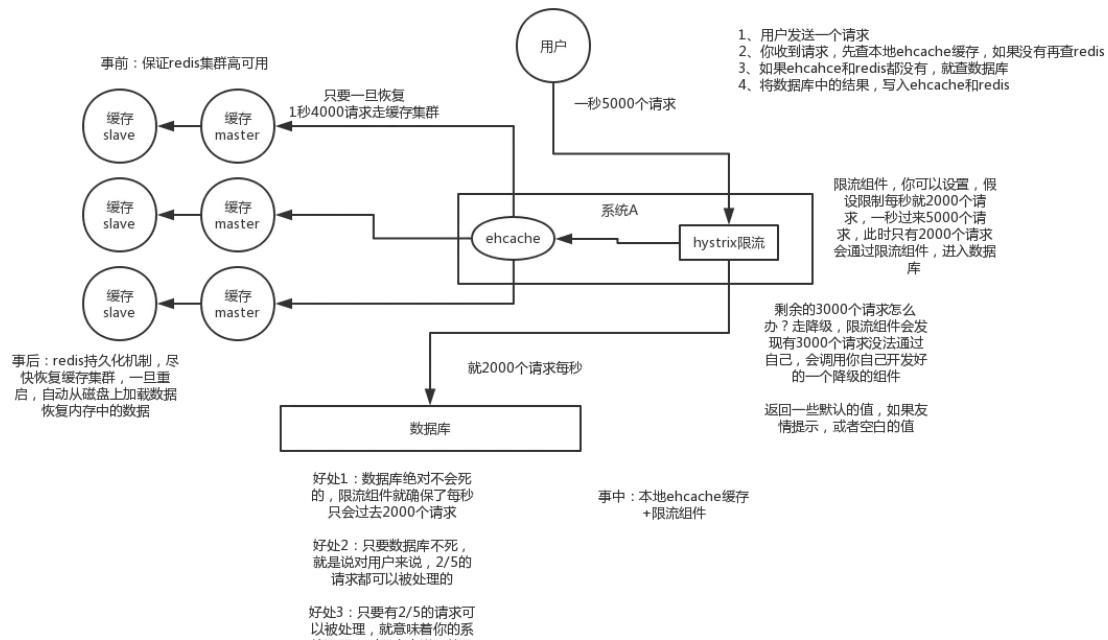
设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。与缓存击穿的区别：雪崩是很多 key，击穿是某一个key 缓存。

解决方案：

将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

33. 缓存雪崩和缓存穿透问题解决方案

1、缓存雪崩



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

• 解决办法

- 事前：尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。
- 事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL崩掉
- 事后：利用 redis 持久化机制保存的数据尽快恢复缓存
- 缓存雪崩

2、缓存穿透

- **简介：**一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

• 解决办法

最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

34. redis 和 memcached 什么区别？为什么高并发下有时单线程的 redis 比多线程的memcached 效率要高？**

区别：

- 1.mc 可缓存图片和视频。rd 支持除 k/v 更多的数据结构；
- 2.rd 可以使用虚拟内存，rd 可持久化和 aof 灾难恢复，rd 通过主从支持数据备份；
- 3.rd 可以做消息队列。

原因：

mc 多线程模型引入了缓存一致性和锁，加锁带来了性能损耗。

redis 主从复制如何实现的？redis 的集群模式如何实现？redis 的 key 是如何寻址的？

主从复制实现：主节点将自己内存中的数据做一份快照，将快照发给从节点，从节点将数据恢复到内存中。之后再每次增加新数据的时候，主节点以类似于 mysql 的二进制日志方式将语句发送给从节点，从节点拿到主节点发送过来的语句进行重放。

分片方式：

- 客户端分片
- 基于代理的分片
- Twemproxy
- codis-路由查询分片
- Redis-cluster (本身提供了自动将数据分散到 Redis Cluster 不同节点的能力，整个数据集合的某个数据子集存储在哪个节点对于用户来说是透明的)

redis-cluster 分片原理：Cluster 中有一个 16384 长度的槽(虚拟槽)，编号分别为 0-16383。

每个 Master 节点都会负责一部分的槽，当有某个 key 被映射到某个 Master 负责的槽，那么这个 Master 负责为这个 key 提供服务，至于哪个 Master 节点负责哪个槽，可以由用户指定，也可以在初始化的时候自动生成，只有 Master 才拥有槽的所有权。Master 节点维护着一个 16384/8 字节的位序列，Master 节点用 bit 来标识对于某个槽自己是否拥有。比如对于编号为 1 的槽，Master 只要判断序列的第二位（索引从 0 开始）是不是为 1 即可。

这种结构很容易添加或者删除节点。比如如果我想新添加个节点 D，我需要从节点 A、B、C 中得部分槽到 D 上。

35. 使用 redis 如何设计分布式锁？说一下实现思路？使用 zk 可以吗？如何实现？这两种有什么区别？

redis:

- 1.线程 A setnx(上锁的对象,超时时的时间戳 t1)，如果返回 true，获得锁。
- 2.线程 B 用 get 获取 t1,与当前时间戳比较,判断是否超时,没超时 false,若超时执行第 3 步;
- 3.计算新的超时时间 t2,使用 getset 命令返回 t3(该值可能其他线程已经修改过),如果

t1==t3，获得锁，如果 t1!=t3 说明锁被其他线程获取了。

4. 获取锁后，处理完业务逻辑，再去判断锁是否超时，如果没超时删除锁，如果已超时，不用处理（防止删除其他线程的锁）。

zk:

1. 客户端对某个方法加锁时，在 zk 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点 node1；

2. 客户端获取该路径下所有已经创建的子节点，如果发现自己创建的 node1 的序号是最小的，就认为这个客户端获得了锁。

3. 如果发现 node1 不是最小的，则监听比自己创建节点序号小的最大的节点，进入等待。

4. 获取锁后，处理完逻辑，删除自己创建的 node1 即可。

区别：zk 性能差一些，开销大，实现简单。

36. 知道 redis 的持久化吗？底层如何实现的？有什么优点缺点？

RDB(Redis DataBase:在不同的时间点将 redis 的数据生成的快照同步到磁盘等介质上):

内存到硬盘的快照，定期更新。缺点：耗时，耗性能(fork+io 操作)，易丢失数据。

AOF(Append Only File):

将 redis 所执行过的所有指令都记录下来，在下次 redis 重启时，只需要执行指令就可以了)：

写日志。缺点：体积大，恢复速度慢。bgsave 做镜像全量持久化，aof 做增量持久化。因为 bgsave 会消耗比较长的时间，不够实时，在停机的时候会导致大量的数据丢失，需要 aof 来配合，在 redis 实例重启时，优先使用 aof 来恢复内存的状态，如果没有 aof 日志，就会使用 rdb 文件来恢复。Redis 会定期做 aof 重写，压缩 aof 文件日志大小。Redis4.0 之后有了混合持久化的功能，将 bgsave 的全量和 aof 的增量做了融合处理，这样既保证了恢复的效率又兼顾了数据的安全性。bgsave 的原理，fork 和 cow，fork 是指 redis 通过创建子进程来进行 bgsave 操作，cow 指的是 copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

37. redis 持久化机制(怎么保证 redis 挂掉之后再重启数据可以进行恢复)

1、快照 (snapshotting) 持久化 (RDB)

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save 900 1          #在900秒(15分钟)之后, 如果至少有1个key发生变化, Redis就会自动触发  
BGSAVE命令创建快照。  
save 300 10         #在300秒(5分钟)之后, 如果至少有10个key发生变化, Redis就会自动触发  
BGSAVE命令创建快照。  
save 60 10000       #在60秒(1分钟)之后, 如果至少有10000个key发生变化, Redis就会自动触发  
BGSAVE命令创建快照。  
复制代码
```

2、AOF (append-only file) 持久化

与快照持久化相比, AOF持久化的实时性更好, 因此已成为主流的持久化方案。默认情况下Redis没有开启AOF (append only file) 方式的持久化, 可以通过appendonly参数开启:

```
appendonly yes  
复制代码
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令, Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同, 都是通过dir参数设置的, 默认的文件名是appendonly.aof。

在Redis的配置文件中存在三种不同的 AOF 持久化方式, 它们分别是:

```
appendfsync always    #每次有数据修改发生时都会写入AOF文件, 这样会严重降低Redis的速度  
appendfsync everysec   #每秒钟同步一次, 显示地将多个写命令同步到硬盘  
appendfsync no         #让操作系统决定何时进行同步  
复制代码
```

为了兼顾数据和写入性能, 用户可以考虑 appendfsync everysec选项, 让Redis每秒同步一次AOF文件, Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃, 用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候, Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

3、Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化 (默认关闭, 可以通过配置项 aof-use-rdb-preamble 开启)。

如果把混合持久化打开, AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。当然缺点也是有的, AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式, 可读性较差。

4、补充内容: AOF 重写

AOF重写可以产生一个新的AOF文件, 这个新的AOF文件和原有的AOF文件所保存的数据库状态一样, 但体积更小。

AOF重写是一个有歧义的名字, 该功能是通过读取数据库中的键值对来实现的, 程序无须对现有AOF文件进行任何读入、分析或者写入操作。

在执行 BGREWRITEAOF 命令时, Redis 服务器会维护一个 AOF 重写缓冲区, 该缓冲区会在子进程创建新AOF文件期间, 记录服务器执行的所有写命令。当子进程完成创建新AOF文件的工作之后, 服务器会将重写缓冲区中的所有内容追加到新AOF文件的末尾, 使得新旧两个AOF文件所保存的数据库状态一致。最后, 服务器用新的AOF文件替换旧的AOF文件, 以此来完成AOF文件重写操作

微信搜索公众号: Java专栏, 获取最新面试手册

38. redis 过期策略都有哪些？LRU 算法知道吗？

过期策略：

定时过期(一 key 一定时器)，惰性过期：只有使用 key 时才判断 key 是否已过期，过期则清除。定期过期：前两者折中。

LRU:new LinkedHashMap<K, V>(capacity, DEFAULT_LOAD_FACTORY, true); //第三个参数置为 true，代表 linkedlist 按访问顺序排序，可作为 LRU 缓存；设为 false 代表按插入顺序排序，可作为 FIFO 缓存

LRU 算法实现：

- 1.通过双向链表来实现，新数据插入到链表头部；
- 2.每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
- 3.当链表满的时候，将链表尾部的数据丢弃。

LinkedHashMap：HashMap 和双向链表合二为一即是 LinkedHashMap。HashMap 是无序的， LinkedHashMap 通过维护一个额外的双向链表保证了迭代顺序。该迭代顺序可以是插入顺序（默认），也可以是访问顺序。

39 在选择缓存时，什么时候选择 redis，什么时候选择 memcached？

选择 redis 的情况：

- 1、复杂数据结构，value 的数据是哈希，列表，集合，有序集合等这种情况下，会选择redis，因为 memcache 无法满足这些数据结构，最典型的使用场景是，用户订单列表，用户消息，帖子评论等。
- 2、需要进行数据的持久化功能，但是注意，不要把 redis 当成数据库使用，如果 redis 挂了，内存能够快速恢复热数据，不会将压力瞬间压在数据库上，没有 cache 预热的过程。对于只读和数据一致性要求不高的场景可以采用持久化存储
- 3、高可用，redis 支持集群，可以实现主动复制，读写分离，而对于 memcache 如果想要实现高可用，需要进行二次开发。
- 4、存储的内容比较大，memcache 存储的 value 最大为 1M。

选择 memcache 的场景：

- 1、纯 KV，数据量非常大的业务，使用 memcache 更合适，原因是：
 - a) memcache 的内存分配采用的是预分配内存池的管理方式，能够省去内存分配的时间，redis 是临时申请空间，可能导致碎片化。
 - b) 虚拟内存使用，memcache 将所有的数据存储在物理内存里，redis 有自己的 vm 机制，理论上能够存储比物理内存更多的数据，当数据超量时，引发 swap，把冷数据刷新到磁盘上，从这点上，数据量大时，memcache 更快

- c) 网络模型，memcache 使用非阻塞的 IO 复用模型，redis 也是使用非阻塞的 IO 复用模型，但是 redis 还提供了一些非 KV 存储之外的排序，聚合功能，复杂的 CPU 计算，会阻塞整个 IO 调度，从这点上由于 redis 提供的功能较多，memcache 更快些
- d) 线程模型，memcache 使用多线程，主线程监听，worker 子线程接受请求，执行读写，这个过程可能存在锁冲突。redis 使用的单线程，虽然无锁冲突，但是难以利用多核的特性提升吞吐量。

40. 缓存与数据库不一致怎么办

假设采用的主存分离，读写分离的数据库，

如果一个线程 A 先删除缓存数据，然后将数据写入到主库当中，这个时候，主库和从库同步没有完成，线程 B 从缓存当中读取数据失败，从从库当中读取到旧数据，然后更新至缓存，这个时候，缓存当中的就是旧的数据。

发生上述不一致的原因在于，主从库数据不一致问题，加入了缓存之后，主从不一致的时间被拉长了

处理思路：在从库有数据更新之后，将缓存当中的数据也同时进行更新，即当从库发生了数据更新之后，向缓存发出删除，淘汰这段时间写入的旧数据。

主从数据库不一致如何解决场景描述，对于主从库，读写分离，如果主从库更新同步有时差，就会导致主从库数据的不一致

- 1、忽略这个数据不一致，在数据一致性要求不高的业务下，未必需要时时一致性
- 2、强制读主库，使用一个高可用的主库，数据库读写都在主库，添加一个缓存，提升数据读取的性能。
- 3、选择性读主库，添加一个缓存，用来记录必须读主库的数据，将哪个库，哪个表，哪个主键，作为缓存的 key，设置缓存失效的时间为主从库同步的时间，如果缓存当中有这个数据，直接读取主库，如果缓存当中没有这个主键，就到对应的从库中读取。

41. Redis 常见的性能问题和解决方案

- 1、master 最好不要做持久化工作，如 RDB 内存快照和 AOF 日志文件
- 2、如果数据比较重要，某个 slave 开启 AOF 备份，策略设置成每秒同步一次
- 3、为了主从复制的速度和连接的稳定性，master 和 Slave 最好在一个局域网内
- 4、尽量避免在压力大得主库上增加从库
- 5、主从复制不要采用网状结构，尽量是线性结构，Master<--Slave1<---Slave2

42. Redis 的数据淘汰策略有哪些

volatile-lru 从已经设置过期时间的数据集中挑选最近最少使用的数据淘汰

volatile-ttl 从已经设置过期时间的数据库集当中挑选将要过期的数据

volatile-random 从已经设置过期时间的数据集任意选择淘汰数据

allkeys-lru 从数据集中挑选最近最少使用的数据淘汰

allkeys-random 从数据集中任意选择淘汰的数据

no-eviction 禁止驱逐数据

43. redis 常见数据结构以及使用场景分析

1、String

常用命令: set,get,decr,incr,mget 等。

String数据结构是简单的key-value类型， value其实不仅可以是String， 也可以是数字。

常规key-value缓存应用； <br**> 常规计数：微博数，粉丝数等。

2、Hash

常用命令: hget,hset,hgetall 等。

hash 是一个 string 类型的 field 和 value 的映射表， hash 特别适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。比如我们可以 hash 数据结构来存储用户信息，商品信息等等。

3、List

常用命令: lpush,rpush,lpop,rpop,range等

list 就是链表， Redis list 的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，消息列表等功能都可以用Redis的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 range 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

4、Set

常用命令: sadd,spop,smembers,sunion 等 set 对外提供的功能与list类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时， set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：

5、Sorted Set

常用命令: zadd,zrange,zrem,zcard等

和set相比， sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

举例：在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

44. Redis 当中有哪些数据结构

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

我们实际项目中比较常用的是string，hash如果你是Redis中高级用户，还需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub。

如果说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用 keys 指令可以扫出指定模式的 key 列表。

对方接着追问：如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题？

这个时候你要回答 redis 关键的一个特性：redis 的单线程的。keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 scan 指令，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

45. 使用 Redis 做过异步队列吗，是如何实现的

使用 list 类型保存数据信息，rpush 生产消息，lpop 消费消息，当 lpop 没有消息时，可以 sleep 一段时间，然后再检查有没有信息，如果不 sleep 的话，可以使用 blpop，在没有信息的时候，会一直阻塞，直到信息的到来。redis 可以通过 pub/sub 主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

46. Redis 如何实现延时队列

使用 sortedset，使用时间戳做 score，消息内容作为 key，调用 zadd 来生产消息，消费者使用 zrangebyscore 获取 n 秒之前的数据做轮询处理。

47. 请用Redis和任意语言实现一段恶意登录保护的代码，限制1小时内每用户Id最多只能登录5次。具体登录函数或功能用空函数即可，不用详细写出。

用列表实现：列表中每个元素代表登陆时间，只要最后的第5次登陆时间和现在时间差不超过1小时就禁止登陆。用Python写的代码如下：

```
#!/usr/bin/env python3
import redis
import sys
import time

r = redis.StrictRedis(host='127.0.0.1', port=6379, db=0)
```

```

try:
    id = sys.argv[1]
except:
    print('input argument error')
    sys.exit(0)
if r.llen(id) >= 5 and time.time() - float(r.lindex(id, 4)) <= 3600:
    print("you are forbidden logining")
else:
    print('you are allowed to login')
    r.lpush(id, time.time())
# Login_func()

```

48. 为什么redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。

如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

MongoDB面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注

获取最新面试手册

1. mongodb是什么？

MongoDB 是由 C++语言编写的，是一个基于分布式文件存储的开源数据库系统。在高负载的情况下，添加更多的节点，可以保证服务器性能。MongoDB 旨在给 WEB 应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储给一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组。

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

← field: value
← field: value
← field: value
← field: value

2. mongodb有哪些特点?

- 1、MongoDB 是一个面向文档存储的数据库，操作起来比较简单和容易。
- 2、你可以在 MongoDB 记录中设置任何属性的索引 (如： FirstName="Sameer",Address="8 Gandhi Road") 来实现更快的排序。
- 3、你可以通过本地或者网络创建数据镜像，这使得 MongoDB 有更强的扩展性。
- 4、如果负载的增加 (需要更多的存储空间和更强的处理能力) ，它可以分布在计算机网络中的其他节点上这就是所谓的分片。
- 5、Mongo 支持丰富的查询表达式。查询指令使用 JSON 形式的标记，可轻易查询文档中内嵌的对象及数组。
- 6、MongoDb 使用 update() 命令可以实现替换完成的文档 (数据) 或者一些指定的数据字段。
- 7、Mongodb 中的 Map/reduce 主要是用来对数据进行批量处理和聚合操作。
- 8、Map 和 Reduce。Map 函数调用 emit(key,value) 遍历集合中所有的记录，将 key 与 value 传给 Reduce 函数进行处理。
- 9、Map 函数和 Reduce 函数是使用 Javascript 编写的，并可以通过 db.runCommand 或 mapreduce 命令来执行 MapReduce 操作。
- 10、GridFS 是 MongoDB 中的一个内置功能，可以用于存放大量小文件。
- 11、MongoDB 允许在服务端执行脚本，可以用 Javascript 编写某个函数，直接在服务端执行，也可以把函数的定义存储在服务端，下次直接调用即可。

3. mongodb的结构介绍

数据库中存储的对象设计 bson，一种类似 json 的二进制文件，由键值对组成

4. MongoDB的优势有哪些

- 面向文档的存储：以 JSON 格式的文档保存数据。
- 任何属性都可以建立索引。
- 复制以及高可扩展性。
- 自动分片。
- 丰富的查询功能。
- 快速的即时更新。
- 来自 MongoDB 的专业支持。

5. 什么是集合

集合就是一组 MongoDB 文档。它相当于关系型数据库 (RDBMS) 中的表这种概念。集合位于单独的一个数据库中。一个集合内的多个文档可以有多个不同的字段。一般来说，集合中的文档都有着相同或相关的目的。

6. 什么是文档

文档由一组key value组成。文档是动态模式,这意味着同一集合里的文档不需要有相同的字段和结构。在关系型数据库中table中的每一条记录相当于MongoDB中的一个文档。

7. 什么是NoSQL数据库?NoSQL与RDBMS直接有什么区别?为什么要使用和不使用NoSQL数据库?说一说NoSQL数据库的几个优点?

NoSQL是非关系型数据库, NoSQL = Not Only SQL。

关系型数据库采用的结构化的数据, NoSQL采用的是键值对的方式存储数据。

在处理非结构化/半结构化的大数据时; 在水平方向上进行扩展时; 随时应对动态增加的数据项时可以优先考虑使用NoSQL数据库。

再考虑数据库的成熟度; 支持; 分析和商业智能; 管理及专业性等问题时, 应优先考虑关系型数据库。

8. NoSQL数据库有哪些类型?

NoSQL数据库的类型

例如: MongoDB, Cassandra, CouchDB, Hypertable, Redis, Riak, HBASE, Memcache

9. MySQL与MongoDB之间最基本的差别是什么?

MySQL和MongoDB两者都是免费开源的数据库。MySQL和MongoDB有许多基本差别包括数据的表示(data representation), 查询, 关系, 事务, schema的设计和定义, 标准化(normalization), 速度和性能。

通过比较MySQL和MongoDB, 实际上我们是在比较关系型和非关系型数据库, 即数据存储结构不同。

10. 你怎么比较MongoDB、CouchDB及CouchBase?

MongoDB和CouchDB都是面向文档的数据库。MongoDB和CouchDB都是开源NoSQL数据库的最典型代表。除了都以文档形式存储外它

们没有其他的共同点。

MongoDB和CouchDB在数据模型实现、接口、对象存储以及复制方法等方面有很多不同。

11. MongoDB成为最好NoSQL数据库的原因是什么?

以下特点使得MongoDB成为最好的NoSQL数据库:

- 面向文件的
- 高性能
- 高可用性
- 易扩展性

- 丰富的查询语言

12. journal回放在条目(entry)不完整时(比如恰巧有一个中途故障了)会遇到问题吗?

每个journal (group)的写操作都是一致的，除非它是完整的否则在恢复过程中它不会回放。

13. 分析器在MongoDB中的作用是什么?

MongoDB中包括了一个可以显示数据库中每个操作性能特点的数据库分析器。通过这个分析器你可以找到比预期慢的查询(或写操作);

利用这一信息，比如，可以确定是否需要添加索引。

14. 如果用户移除对象的属性，该属性是否从存储层中删除?

是的，用户移除属性然后对象会重新保存(re-save())。

15. 能否使用日志特征进行安全备份?

是的。

16. 允许空值null吗?

对于对象成员而言，是的。然而用户不能够添加空值(null)到数据库从集(collection)因为空值不是对象。然而用户能够添加空对象{}。

17. 更新操作立刻fsync到磁盘?

不会，磁盘写操作默认是延迟执行的。写操作可能在两三秒(默认在60秒内)后到达磁盘。

例如，如果一秒内数据库收到一千个对一个对象递增的操作，仅刷新磁盘一次。(注意，尽管fsync选项在命令行和经过getLastError_old是有效的)

18. 如何执行事务/加锁?

MongoDB没有使用传统的锁或者复杂的带回滚的事务，因为它设计的宗旨是轻量，快速以及可预计的高性能。可以把它类比成MySQL MyISAM的自动提交模式。通过精简对事务的支持，性能得到了提升，特别是在一个可能会穿过多个服务器的系统里。

19. 为什么我的数据文件如此庞大?

MongoDB会积极的预分配预留空间来防止文件系统碎片。

20. 启用备份故障恢复需要多久?

从备份数据库声明主数据库宕机到选出一个备份数据库作为新的主数据库将花费10到30秒时间。

这期间在主数据库上的操作将会失败--包括写入和强一致性读取(strong consistent read)操作。

然而，你还能在第二数据库上执行最终一致性查询(eventually consistent query)(在slaveOk模式下)，即使在这段时间里。

21. 什么是master或primary?

它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员。

在一个备份集群中，当失效备援(failover)事件发生时，一个另外的成员会变成primary。

22. 什么是secondary或slave?

Secondary从当前的primary上复制相应的操作。它是通过跟踪复制oplog(local.oplog.rs)做到的。

23. 必须调用getLastError来确保写操作生效了么?

不用。不管你有没有调用getLastError(又叫"Safe Mode")服务器做的操作都一样。调用getLastError只是为了确认写操作成功提交了。当然，你经常想得到确认，但是写操作的安全性和是否生效不是由这个决定的。

24. 我应该启动一个集群分片(sharded)还是一个非集群分片的MongoDB环境?

为开发便捷起见，我们建议以非集群分片(unsharded)方式开始一个 MongoDB 环境，除非一台服务器不足以存放你的初始数据集。从非集群分片升级到集群分片(sharding)是无缝的，所以在你的数据集还不是很大的时候没必要考虑集群分片(sharding)。

25. 分片(sharding)和复制(replication)是怎样工作的?

每一个分片(shard)是一个分区数据的逻辑集合。分片可能由单一服务器或者集群组成，我们推荐为每一个分片(shard)使用集群。

26. 数据在什么时候才会扩展到多个分片(shard)里?

MongoDB 分片是基于区域(range)的。所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中。只有当存在多余一个块的时候，才会有多个分片获取数据的选项。现在，每个默认块的大小是 64Mb，所以你需要至少 64 Mb 空间才可以实施一个迁移。

27. 当我试图更新一个正在被迁移的块(chunk)上的文档时会发生什么?

更新操作会立即发生在旧的分片(shard)上，然后更改才会在所有权转移(ownership transfers)前复制到新的分片上。

28. 如果在一个分片(shard)停止或者很慢的时候，我发起一个查询会怎样?

如果一个分片(shard)停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片(shard)响应很慢，MongoDB则会等待它的响应。

29. 我可以把moveChunk目录里的旧文件删除吗?

没问题，这些文件是在分片(shard)进行均衡操作(balancing)的时候产生的临时文件。一旦这些操作已经完成，相关的临时文件也应该被删除掉。但目前清理工作是需要手动的，所以请小心地考虑再释放这些文件的空间。

30. 怎么查看 Mongo 正在使用的链接?

```
db._adminCommand("connPoolStats");
```

31. 如果块移动操作(moveChunk)失败了，我需要手动清除部分转移的文档吗?

不需要，移动操作是一致(consistent)并且是确定性的(deterministic);一次失败后，移动操作会不断重试；当完成后，数据只会出现在新的分片里(shard)。

32. 如果我在使用复制技术(replication)，可以一部分使用日志(journaling)而其他部分则不使用吗?

可以。

33. MongoDB在A:{B,C}上建立索引，查询A:{B,C}和A:{C,B}都会使用索引吗？

不会，只会在A:{B,C}上使用索引。

34. 如果一个分片（Shard）停止或很慢的时候，发起一个查询会怎样？

如果一个分片停止了，除非查询设置了“Partial”选项，否则查询会返回一个错误。如果一个分片响应很慢，MongoDB会等待它的响应。

35. MongoDB支持存储过程吗？如果支持的话，怎么用？

MongoDB支持存储过程，它是javascript写的，保存在db.system.js表中。

36. 如何理解MongoDB中的GridFS机制，MongoDB为何使用GridFS来存储文件？

GridFS是一种将大型文件存储在MongoDB中的文件规范。使用GridFS可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了BSON对象有限制的问题。

37. MongoDB支持存储过程吗？如果支持的话，怎么用？

MongoDB支持存储过程，它是javascript写的，保存在db.system.js表中。

38. 如何理解MongoDB中的GridFS机制，MongoDB为何使用GridFS来存储文件？

GridFS是一种将大型文件存储在MongoDB中的文件规范。使用GridFS可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了BSON对象有限制的问题。

39. 为什么MongoDB的数据文件很大？

MongoDB采用的预分配空间的方式来防止文件碎片。

40. MongoDB在A:{B,C}上建立索引，查询A:{B,C}和A:{C,B}都会使用索引吗？

不会，只会在A:{B,C}上使用索引。

41. 如果用户移除对象的属性，该属性是否从存储层中删除？

是的，用户移除属性然后对象会重新保存（re-save()）。

42. 能否使用日志特征进行安全备份？

是的

43. 更新操作立刻fsync到磁盘？

一般磁盘的写操作都是延迟执行的

44. 如何执行事务/加锁？

因为mongodb设计就是轻量高性能，所以没有传统的锁和复杂的事务的回滚

45. 什么是master或primary？

当前备份集群负责所有的写入操作的主要节点，在集群中，当主节点（master）失效，另一个成员会变为master

46. getLastErrors的作用

调用getLastErrors可以确认当前的写操作是否成功的提交

47. 分片（sharding）和复制（replication）是怎样工作的？

分片可能是单一的服务器或者集群组成，推荐使用集群

48. 数据在什么时候才会扩展到多个分片（shard）里？

mongodb分片是基于区域的，所以一个集合的所有对象都放置在同一个块中，只有当存在多余一个块的时候，才会有多个分片获取数据的选项

49. 什么是“mongod”

mongod是处理MongoDB系统的主要进程。它处理数据请求，管理数据存储，和执行后台管理操作。当我们运行mongod命令意味着正在启动MongoDB进程，并且在后台运行。

50. "mongod"参数有什么

- 传递数据库存储路径， 默认是"/data/db"
- 端口号 默认是 "27017"

51. 什么是"mongo"

它是一个命令行工具用于连接一个特定的mongod实例。当我们没有带参数运行mongo命令它将使用默认的端口号和localhost连接

52. MongoDB哪个命令可以切换数据库

MongoDB 用 use +数据库名称的方式来创建数据库。use 会创建一个新的数据库，如果该数据库存在，则返回这个数据库。

53. 什么是非关系型数据库

非关系型数据库是对不同于传统关系型数据库的统称。非关系型数据库的显著特点是不使用SQL作为查询语言，数据存储不需要特定的表格模式。由于简单的设计和非常好的性能所以被用于大数据和Web Apps等

54. 非关系型数据库有哪些类型

- -Key-Value 存储: Amazon S3
- 图表: Neo4j
- 文档存储: MongoDB
- 基于列存储: Cassandra

55. 为什么用MongoDB?

- 1、架构简单
- 2、没有复杂的连接
- 3、深度查询能力,MongoDB支持动态查询。
- 4、容易调试
- 5、容易扩展
- 6、不需要转化/映射应用对象到数据库对象
- 7、使用内部内存作为存储工作区,以便更快的存取数据。

56. 在哪些场景使用MongoDB

- 大数据
- 内容管理系统
- 移动端Apps
- 数据管理

57. MongoDB中的命名空间是什么意思？

MongoDB内部有预分配空间的机制，每个预分配的文件都用0进行填充。

数据文件每新分配一次，它的大小都是上一个数据文件大小的2倍，每个数据文件最大2G。欢迎关注公种浩：程序员追风，回复003领取一套200页的2020最新的Java面试题手册。

MongoDB每个集合和每个索引都对应一个命名空间，这些命名空间的元数据集中在16M的*.ns文件中，平均每个命名占用约628字节，也即整个数据库的命名空间的上限约为24000。

如果每个集合有一个索引（比如默认的_id索引），那么最多可以创建12000个集合。如果索引数更多，则可创建的集合数就更少了。同时，如果集合数太多，一些操作也会变慢。

要建立更多的集合的话，MongoDB也是支持的，只需要在启动时加上“--nssize”参数，这样对应数据库的命名空间文件就可以变得更大以便保存更多的命名。这个命名空间文件（.ns文件）最大可以为2G。

每个命名空间对应的盘区不一定是连续的。与数据文件增长相同，每个命名空间对应的盘区大小都是随分配次数不断增长的。目的是为了平衡命名空间浪费的空间与保持一个命名空间数据的连续性。

需要注意的一个命名空间

freelist，这个命名空间用于记录不再使用的盘区（被删除的**Collection**或索引）。每当命名空间需要分配新盘区时，会先查看freelist是否有大小合适的盘区可以使用，如果有就回收空闲的磁盘空间。

58. 哪些语言支持MongoDB？

C、C++、C#、Java、Node.js、Perl、Php等

59. 在MongoDB中如何创建一个新的数据库

MongoDB用use + 数据库名称的方式来创建数据库。use会创建一个新的数据库，如果该数据库存在，则返回这个数据库。

60. 在MongoDB中如何查看数据库列表

使用命令"show dbs"

61. MongoDB中的分片是什么意思

分片是将数据水平切分到不同的物理节点。当应用数据越来越大的时候，数据量也会越来越大。当数据量增长时，单台机器有可能无法存储数据或可接受的读取写入吞吐量。利用分片技术可以添加更多的机器来应对数据量增加以及读写操作的要求。

62. 如何查看使用MongoDB的连接Sharding - MongoDB Manual

21. 如何查看使用MongoDB的连接

使用命令"db.adminCommand("connPoolStats")"

```
>db.adminCommand("connPoolStats")
```

63. 什么是复制

复制是将数据同步到多个服务器的过程，通过多个数据副本存储到多个服务器上增加数据可用性。复制可以保障数据的安全性，灾难恢复，无需停机维护（如备份，重建索引，压缩），分布式读取数据。

64. 在MongoDB中如何在集合中插入一个文档

要想将数据插入 MongoDB 集合中，需要使用 insert() 或 save() 方法。

```
db.collectionName.insert({"key":"value"})
db.collectionName.save({"key":"value"})
```

65. 在MongoDB中如何除去一个数据库Collection Methods

24. 在MongoDB中如何除去一个数据库

MongoDB 的 dropDatabase() 命令用于删除已有数据库。

```
db.dropDatabase()
```

66. 在MongoDB中如何查看一个已经创建的集合

可以使用show collections 查看当前数据库中的所有集合清单

```
show collections
```

67. 在MongoDB中如何删除一个集合

MongoDB 利用 db.collection.drop() 来删除数据库中的集合。

```
db.collectionName.drop()
```

68. 为什么要在MongoDB中使用分析器

数据库分析工具(Database Profiler)会针对正在运行的mongod实例收集数据库命令执行的相关信息。包括增删改查的命令以及配置和管理命令。分析器(profiler)会写入所有收集的数据到 system.profile集合，一个capped集合在管理员数据库。分析器默认是关闭的你能通过per数据库或per实例开启。

69. MongoDB支持主键外键关系吗

默认MongoDB不支持主键和外键关系。用Mongodb本身的API需要硬编码才能实现外键关联，不够直观且难度较大。

70. MongoDB支持哪些数据类型

String、Integer、Double、Boolean、Object、Object ID、Arrays、Min/Max Keys、Datetime、Code、Regular Expression等

71. 为什么要在MongoDB中用"Code"数据类型

"Code"类型用于在文档中存储 JavaScript 代码。

72. 为什么要在MongoDB中用"Regular Expression"数据类型

"Regular Expression"类型用于在文档中存储正则表达式

73. 为什么在MongoDB中使用"Object ID"数据类型

"ObjectID"数据类型用于存储文档id

74. "ObjectId"由哪些部分组成

一共有四部分组成:时间戳、客户端ID、客户进程ID、三个字节的增量计数器

*id*是一个12字节长的十六进制数，它保证了每一个文档的唯一性。在插入文档时，需要提供`_id`。如果不提供，那么MongoDB就会为每一文档提供一个唯一的*id*。*id*的头4个字节代表的是当前的时间戳，接着的后3个字节表示的是机器*id*号，接着的2个字节表示MongoDB服务器进程*id*，最后的3个字节代表递增值。

75. 在MongoDb中什么是索引

索引用于高效的执行查询。没有索引MongoDB将扫描查询整个集合中的所有文档这种扫描效率很低，需要处理大量数据。索引是一种特殊的数据结构，将一小块数据集保存为容易遍历的形式。索引能够存储某种特殊字段或字段集的值，并按照索引指定的方式将字段值进行排序。

76. 如何添加索引

使用 db.collection.createIndex() 在集合中创建一个索引

```
db.collectionName.createIndex({columnName:1})
```

77. 用什么方法可以格式化输出结果

使用pretty() 方法可以格式化显示结果

```
>db.collectionName.find().pretty()
```

78. 如何使用"AND"或"OR"条件循环查询集合中的文档

在 find() 方法中，如果传入多个键，并用逗号(,)分隔它们，那么 MongoDB 会把它看成是AND条件。

```
>db.mycol.find({key1:value1, key2:value2}).pretty()
```

若基于OR条件来查询文档，可以使用关键字\$or。

```
>db.mycol.find(
{
  $or: [
    {key1: value1}, {key2:value2}
  ]
}).pretty()
```

79. 在MongoDB中如何更新数据

update() 与 save() 方法都能用于更新集合中的文档。update() 方法更新已有文档中的值，而 save() 方法则是用传入该方法的文档来替换已有文档。

80. 如何删除文档

MongoDB 利用 remove() 方法 清除集合中的文档。它有 2 个可选参数：

- deletion criteria: (可选) 删除文档的标准。
- justOne: (可选) 如果设为 true 或 1，则只删除一个文档。

```
>db.collectionName.remove({key:value})
```

81. 在MongoDB中如何排序

MongoDB 中的文档排序是通过 sort() 方法来实现的。sort() 方法可以通过一些参数来指定要进行排序的字段，并使用 1 和 -1 来指定排序方式，其中 1 表示升序，而 -1 表示降序。

```
>db.connectionName.find({key:value}).sort({columnName:1})
```

82. 什么是聚合

聚合操作能够处理数据记录并返回计算结果。聚合操作能将多个文档中的值组合起来，对成组数据执行各种操作，返回单一的结果。它相当于 SQL 中的 count(*) 组合 group by。对于 MongoDB 中的聚合操作，应该使用 aggregate() 方法。

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

83. 在MongoDB中什么是副本集

在MongoDB中副本集由一组MongoDB实例组成，包括一个主节点多个次节点，MongoDB客户端的所有数据都写入主节点(Primary),副节点从主节点同步写入数据，以保持所有复制集内存储相同的数据，提高数据可用性。

框架相关面试题

Spring面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. 什么是spring?

Spring 是个java企业级应用的开源开发框架。Spring主要用来开发Java应用，但是有些扩展是针对构建J2EE平台的web应用。Spring 框架目标是简化Java企业级应用开发，并通过POJO为基础的编程模型促进良好的编程习惯。

2. 说说你对Spring的理解

1、Spring是一个开源框架，主要是为简化企业级应用开发而生。可以实现EJB可以实现的功能，Spring是一个IOC和AOP容器框架。
 ◆ **控制反转（IOC）**：Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。
 ◆ **依赖注入（DI）**：Spring使用Java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。
 ◆ **面向切面编程（AOP）**：在面向对象编程(OOP)思想中，我们将事物纵向抽象成一个个的对象。而在面向切面编程中，我们将一个个对象某些类似方面横向抽象成一个切面，对这个切面进行一些如权限验证，事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。

2、在Spring中，所有管理的都是JavaBean对象，而BeanFactory和ApplicationContext就是Spring框架的那个IOC容器，现在一般使用ApplicationContext，其不但包括了BeanFactory的作用，同时还进行了更多的扩展。

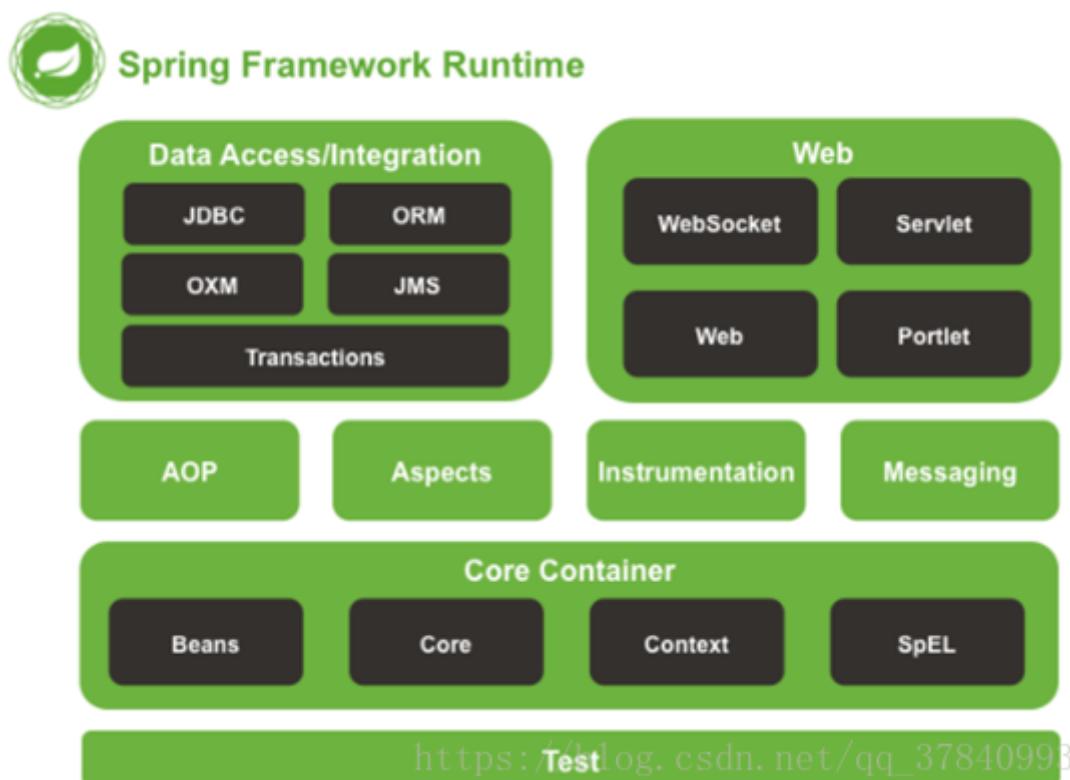
3. 使用Spring框架的好处是什么？

- 1、**轻量**：Spring 是轻量的，基本的版本大约2MB。
- 2、**控制反转**：Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- 3、**面向切面的编程(AOP)**：Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- 4、**容器**：Spring 包含并管理应用中对象的生命周期和配置。
- 5、**MVC框架**：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
- 6、**事务管理**：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- 7、**异常处理**：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

序号	好处	说明
1	轻量	Spring 是轻量的，基本的版本大约2MB。
2	控制反转	Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
3	面向切面编程(AOP)	Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
4	容器	Spring 包含并管理应用中对象的生命周期和配置。
5	MVC框架	Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
6	事务管理	Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务 (JTA) 。
7	异常处理	Spring 提供方便的API把具体技术相关的异常(比如由JDBC, Hibernate or JDO抛出的)转化为一致的unchecked 异常。
8	最重要的	用的人多！！！

4. Spring由哪些模块组成？

截止到目前Spring 框架已集成了 20 多个模块。这些模块主要被分如下图所示的核心容器、数据访问 / 集成、Web、AOP（面向切面编程）、工具、消息和测试模块。



以下是Spring 框架的基本模块：

- Core module

- Bean module
- Context module
- Expression Language module
- JDBC module
- ORM module
- OXM module
- Java Messaging Service(JMS) module
- Transaction module
- Web module
- Web-Servlet module
- Web-Struts module
- Web-Portlet module

核心模块	说明
Spring Core	核心容器: 核心容器提供Spring框架的基本功能。Spring以bean的方式组织和管理Java应用中的各个组件及其关系。Spring使用BeanFactory来产生和管理Bean，它是工厂模式的实现。BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开
Spring Context	应用上下文: 是一个配置文件，向Spring框架提供上下文信息。Spring上下文包括企业服务，如JNDI、EJB、电子邮件、国际化、校验和调度功能
Spring AOP	面向切面编程: 是面向对象编程的有效补充和完善，Spring的AOP是基于动态代理实现的，实现的方式有两种分别是Schema和AspectJ这两种方式
Spring Dao	JDBC和Dao模块: JDBC、DAO的抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理，和不同数据库供应商所抛出的错误信息。异常层次结构简化了错误处理，并且极大的降低了需要编写的代码数量，比如打开和关闭链接
Spring ORM	对象实体映射: Spring框架插入了若干个ORM框架，从而提供了ORM对象的关系工具，其中包括了Hibernate、JDO和IBatis SQL Map等，所有这些都遵从Spring的通用事物和DAO异常层次结构。
Spring Web	Web模块: Web上下文模块建立在应用程序上下文模块之上，为基于web的应用程序提供了上下文。所以Spring框架支持与Struts集成，web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作
Spring Web MVC	MVC模块: MVC框架是一个全功能的构建Web应用程序的MVC实现。通过策略接口，MVC框架变成为高度可配置的。MVC容纳了大量视图技术，其中包括JSP、POI等，模型由JavaBean构成，存放于m当中，而视图是一个接口，负责实现模型，控制器表示逻辑代码，由c的事情。Spring框架的功能可以用在任何J2EE服务器当中，大多数功能也适用于不受管理的环境。Spring的核心要点就是支持不绑定到特定J2EE服务的可重用业务和数据的访问的对象，毫无疑问这样的对象可以在不同的J2EE环境，独立应用程序和测试环境之间重用。

5. Spring框架使用了哪些设计模式

- 1、单例模式
- 2、原型模式
- 3、工厂模式
- 4、适配器模式
- 5、包装模式
- 6、代理模式
- 7、观察者模式
- 8、策略模式
- 9、模板模式

6. Spring支持的ORM

Spring支持以下ORM：

Hibernate

iBatis

JPA (Java Persistence API)

TopLink

JDO (Java Data Objects)

OJB

7. Spring Framework 有哪些不同的功能？

轻量级 - Spring 在代码量和透明度方面都很轻便。

IOC - 控制反转

AOP - 面向切面编程可以将应用业务逻辑和系统服务分离，以实现高内聚。

容器 - Spring 负责创建和管理对象（Bean）的生命周期和配置。

MVC - 对 web 应用提供了高度可配置性，其他框架的集成也十分方便。

事务管理 - 提供了用于事务管理的通用抽象层。Spring 的事务支持也可用于容器较少的环境。

JDBC 异常 - Spring 的 JDBC 抽象层提供了一个异常层次结构，简化了错误处理策略。

8. 什么是Spring的MVC框架？

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

9. springmvc常用到的注解，作用是什么，原理。

@Controller注解

是在Spring的org.springframework.stereotype包下，org.springframework.stereotype.Controller注解类型用于指示Spring类的实例是一个控制器

使用@Controller注解的类不需要继承特定的父类或者实现特定的接口，相对之前的版本实现Controller接口变的更加简单。

而Controller接口的实现类只能处理一个单一的请求动作，而@Controller注解注解的控制器可以同时支持处理多个请求动作，使程序开发变的更加灵活。 @Controller用户标记一个类，使用它标记的类就是一个Spring MVC Controller对象，即：一个控制器类。 Spring使用扫描机制查找应用程序中所有基于注解的控制器类，分发处理器会扫描使用了该注解的方法，并检测该方法是否使用了@RequestMapping注解，而使用@RequestMapping注解的方法才是真正处理请求的处理器。为了保证Spring能找到控制器，我们需要完成两件事：

@RequestParam注解

下面来说org.springframework.web.bind.annotation包下的第三个注解，即：@RequestParam注解，该注解类型用于将指定的请求参数赋值给方法中的形参。那么@RequestParam注解有什么属性呢？它有4种属性，下面将逐一介绍这四种属性：

- 1、name属性该属性的类型是String类型，它可以指定请求头绑定的名称；
- 2、value属性该属性的类型是String类型，它可以设置是name属性的别名；
- 3、required属性该属性的类型是boolean类型，它可以设置指定参数是否必须绑定；
- 4、defaultValue属性该属性的类型是String类型，它可以设置如果没有传递参数可以使用默认值。

@PathVariable注解

下面来说org.springframework.web.bind.annotation包下的第四个注解，即：@PathVariable注解，该注解类型可以非常方便的获得请求url中的动态参数。@PathVariable注解只支持一个属性value，类型String，表示绑定的名称，如果省略则默认绑定同名参数。

10. 在 Spring 中，有几种配置 Bean 的方式？

- 基于XML的配置
- 基于注解的配置
- 基于Java的配置

11. 请解释一下 Spring Bean 的生命周期？

- 首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；
Spring上下文中的Bean生命周期也类似，如下：

1、实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

微信搜索公众号：Java专栏，获取最新面试手册

2、设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息以及通过BeanWrapper提供的设置属性的接口完成依赖注入。

3、处理Aware接口：

接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：

- 如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值；
- 如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。
- 如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

4、BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

5、InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了init-method属性，则会自动调用其配置的初始化方法。

6、如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

7、DisposableBean：

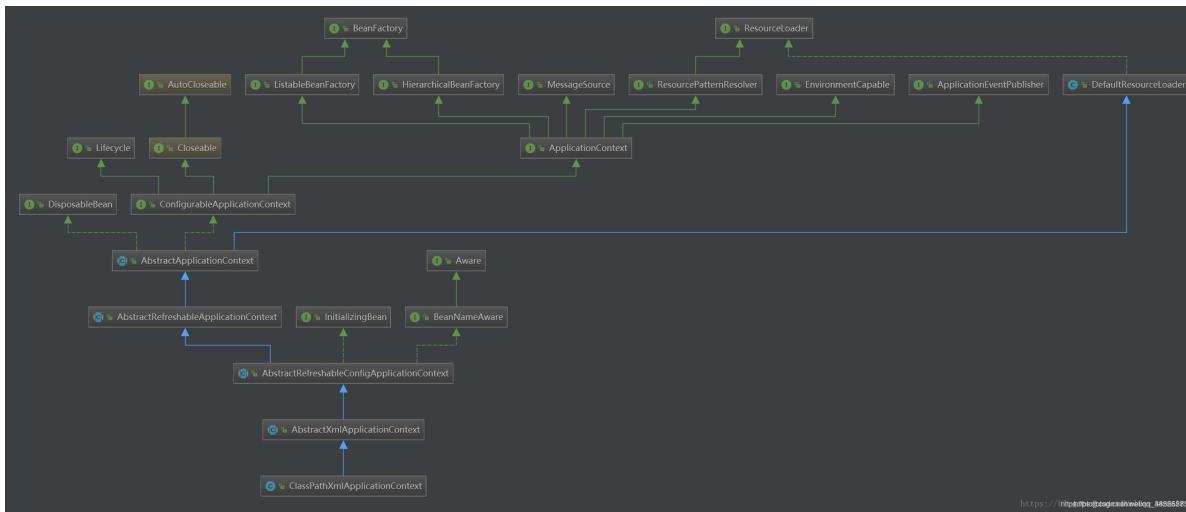
当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

8、destroy-method：

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。

12. BeanFactory和ApplicationContext有什么区别？

在类图关系上BeanFactory是ApplicationContext接口的父接口



Type hierarchy of 'org.springframework.beans.factory.BeanFactory':



<

>

Press 'Ctrl+T' to see the supertype hierarchy

BeanFactory

是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。

ApplicationContext

ApplicationContext接口是由BeanFactory接口派生而来，因而具有BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，ApplicationContext包还提供了以下的功能

1. MessageSource, 提供国际化的消息访问
2. 资源访问，如URL和文件
3. 事件传播
4. 载入多个（有继承关系）上下文，使得每一个上下文都专注于一个特定的层次，比如应用的web层，其实通过上面的类图大家应该也能看的出来BeanFactory和ApplicationContext的区别。

13. Spring框架中的单例bean是线程安全的吗?

肯定不是线程安全的，当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求多对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对该单例状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

14. Spring如何处理线程并发问题?

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而 ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

15. 什么是Spring的内部bean?

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在 Spring 的 基于XML的 配置元数据中，可以在 或\ 元素内使用 元素，内部bean通常是匿名的，它们的 Scope一般是prototype。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="CustomerBean" class="com.dpb.common.Customer">
        <property name="person" ref="PersonBean"/>
    </bean>

    <bean id="PersonBean" class="com.dpb.common.Person">
        <property name="name" value="波波烤鸭"/>
        <property name="address" value="深圳"/>
        <property name="age" value="17"/>
    </bean>
</beans>
```

改为内部bean的方式

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="CustomerBean" class="com.dpb.common.Customer">
        <property name="person">
            <bean class="com.dpb.common.Person">
                <property name="name" value="波波烤鸭"/>
                <property name="address" value="湖南"/>
                <property name="age" value="17"/>
            </bean>
        </property>
    </bean>
</beans>

```

内部 bean 也支持构造器注入

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="CustomerBean" class="com.dpb.common.Customer">
        <constructor-arg>
            <bean class="com.dpb.common.Person">
                <property name="name" value="波波烤鸭"/>
                <property name="address" value="湖南"/>
                <property name="age" value="17"/>
            </bean>
        </constructor-arg>
    </bean>
</beans>

```

id 或 name 值在bean类是没有必要以一个内部 bean 呈现

16. Spring Bean 有哪些作用域，它们之间有什么区别？

singleton：这种 bean 范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个 bean 的实例，单例的模式由 bean factory 自身来维护。

prototype：原形范围与单例范围相反，为每一个 bean 请求提供一个实例。

request：在请求 bean 范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean 会失效并被垃圾回收器回收。

Session：与请求范围类似，确保每个 session 中有一个 bean 的实例，在 session 过期后，bean 会随之失效。

global-session：global-session 和 Portlet 应用相关。当你的应用部署在 Portlet 容器中工作时，它包含很多 portlet。如果你想要声明让所有的 portlet 共用全局的存储变量的话，那么这全局变量需要存储在 global-session 中。

17. Springmvc controller方法中为什么不能定义局部变量？

因为controller是默认单例模式，高并发下全局变量会出现线程安全问题

现这种问题如何解决呢？

- 1、既然是全局变量惹的祸，那就将全局变量都编程局部变量，通过方法参数来传递。
- 2、jdk提供了java.lang.ThreadLocal,它为多线程并发提供了新思路。
- 3、使用@Scope("session"), 会话级别

```
@Controller
//把这个bean 的范围设置成session, 表示这bean是会话级别的,
@Scope("session")
public class XxxController{
    private List<String> list ;

    //当bean加载完之后, 就会执行init方法, 并且将List实例化;
    @PostConstruct
    public void init(){
        list = new ArrayList<String>();
    }
}
```

- 4、将控制器的作用域从单例改为原型，即在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller

18. 在 Spring中如何注入一个java集合？

Spring提供以下几种集合的配置元素：

元素	说明
list	类型用于注入一列值，允许有相同的值。
set	类型用于注入一组值，不允许有相同的值。
map	类型用于注入一组键值对，键和值都可以为任意类型。
props	类型用于注入一组键值对，键和值都只能为String类型。

```
<!-- 配置 student对象 -->
<bean class="com.dpb.javabean.Student">
    <property name="id" value="10"/>
    <property name="name" value="波波烤鸭"/>
    <!-- 对象注入 -->
    <property name="cat" ref="catId"></property>
    <!-- List集合注入 -->
    <property name="games">
        <list>
            <value>LOL</value>
            <value>DNF</value>
            <value>CS</value>
        </list>
    </property>

```

```
</list>
</property>
<property name="score">
    <map>
        <entry key="数学" value="99"/>
        <entry key="英语" value="78"/>
        <entry key="化学" value="84"/>
    </map>
</property>
<property name="props">
    <props>
        <prop key="userName">admin</prop>
        <prop key="password">123</prop>
    </props>
</property>
```

19. 什么是Spring MVC框架的控制器？

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

20. Spring DAO 有什么用？

Spring DAO 使得 JDBC, Hibernate 或 JDO 这样的数据访问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

21. Spring JDBC API 中存在哪些类？

- JdbcTemplate
- SimpleJdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert
- SimpleJdbcCall

22. Spring框架中有哪些不同类型的事件？

Spring 提供了以下5种标准的事件：

- 1、上下文更新事件（ContextRefreshedEvent）**：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。
- 2、上下文开始事件（ContextStartedEvent）**：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。
- 3、上下文停止事件（ContextStoppedEvent）**：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。

4、上下文关闭事件 (ContextClosedEvent) : 当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。

5、请求处理事件 (RequestHandledEvent) : 在Web应用中，当一个http请求 (request) 结束触发该事件。

如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

23. 请解释一下，Spring 框架有哪些自动装配模式，它们之间有何区别？

no : 这是 Spring 框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在 bean 定义中用 `标签` 明确的设置依赖关系。

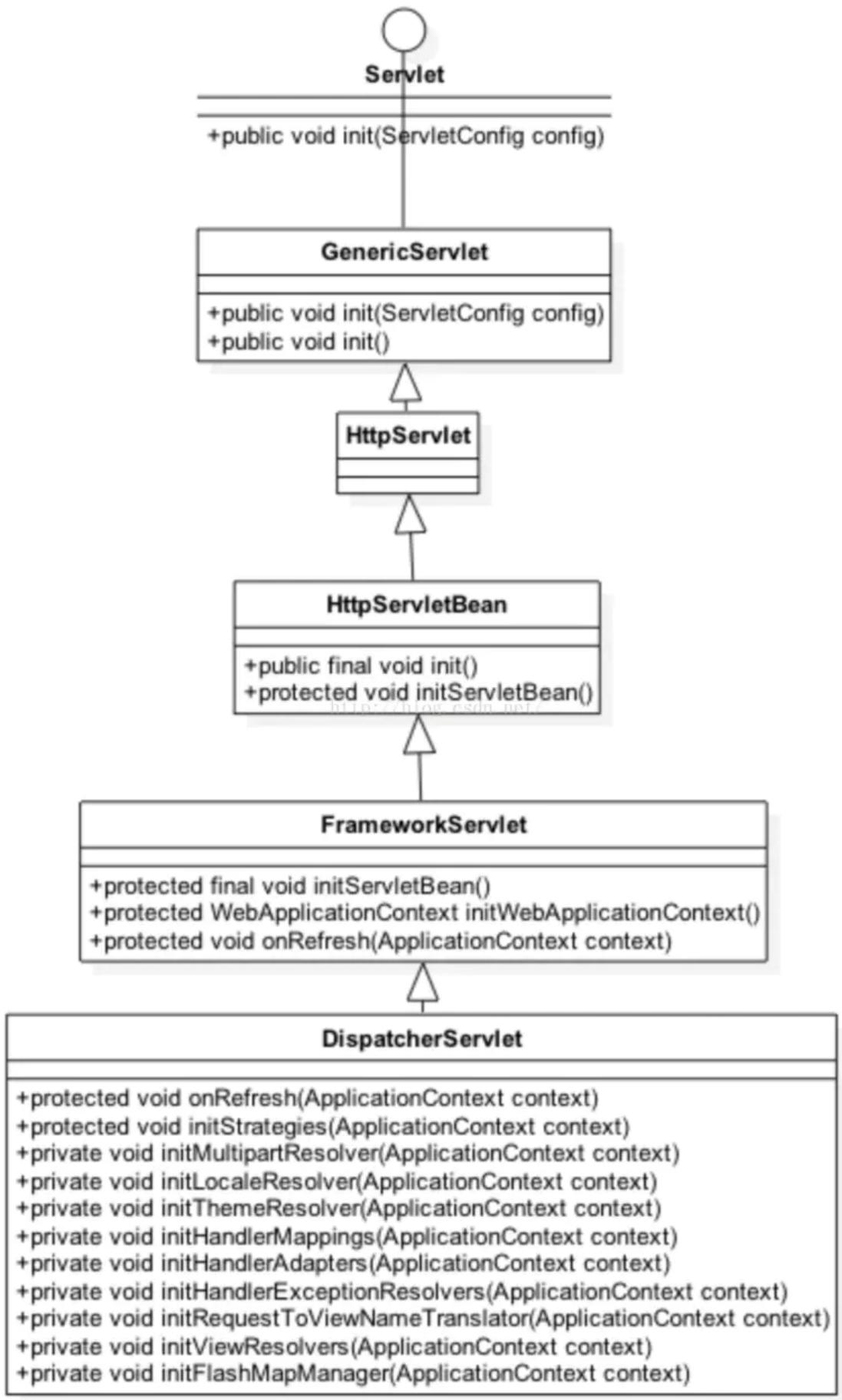
byName : 该选项可以根据 **bean 名称** 设置依赖关系。当向一个 bean 中自动装配一个属性时，容器将根据 bean 的名称自动在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性，如果没有找到的话就报错。

byType : 该选项可以根据 **bean 类型** 设置依赖关系。当向一个 bean 中自动装配一个属性时，容器将根据 bean 的类型自动在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性，如果没有找到的话就报错。

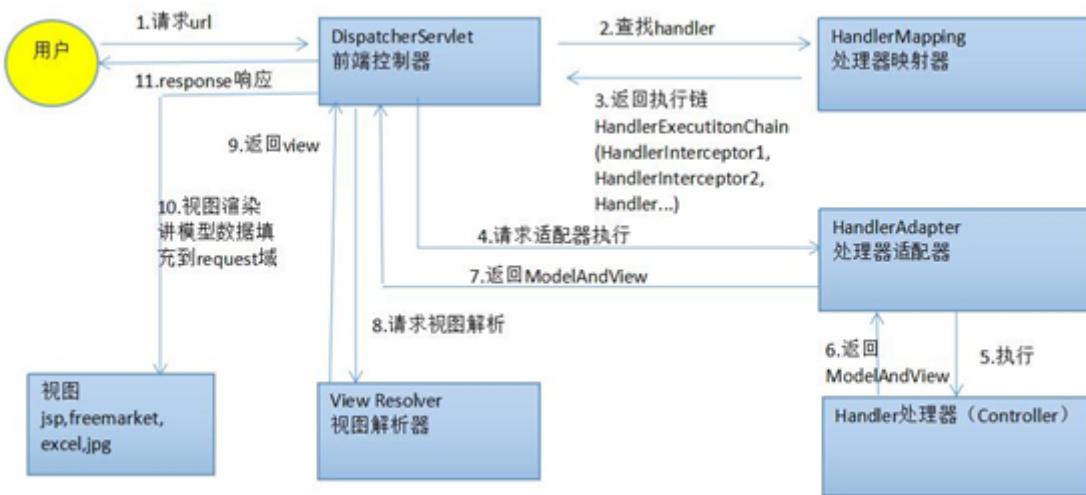
constructor : 构造器的自动装配和 byType 模式类似，但是仅仅适用于与有构造器相同参数的 bean，如果在容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常。

autodetect : 该模式自动探测使用构造器自动装配或者 byType 自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在 bean 内部没有找到相应的构造器或者是无参构造器，容器就会自动选择 byType 的自动装配方式。

24. Springmvc 中DispatcherServlet初始化过程。



25. SpringMVC执行流程和原理 SpringMVC流程：



- 1、用户发出请求到前端控制器DispatcherServlet。
- 2、DispatcherServlet收到请求调用HandlerMapping（处理器映射器）。
- 3、HandlerMapping找到具体的处理器(可查找xml配置或注解配置)，生成处理器对象及处理器拦截器(如果有)，再一起返回给DispatcherServlet。
- 4、DispatcherServlet调用HandlerAdapter（处理器适配器）。
- 5、HandlerAdapter经过适配调用具体的处理器（Handler/Controller）。
- 6、Controller执行完成返回ModelAndView对象。
- 7、HandlerAdapter将Controller执行结果 ModelAndView 返回给DispatcherServlet。
- 8、DispatcherServlet将 ModelAndView 传给ViewReslover（视图解析器）。
- 9、ViewReslover解析后返回具体View（视图）。
- 10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
- 11、DispatcherServlet响应用户。

26. WebApplicationContext

WebApplicationContext 继承了 ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的 ApplicationContext，因为它能处理主题，并找到被关联的servlet。

27. 你用过哪些重要的 Spring 注解？

- @Controller - 用于 Spring MVC 项目中的控制器类。
- @Service - 用于服务类。
- @RequestMapping - 用于在控制器处理程序方法中配置 URI 映射。
- @ResponseBody - 用于发送 Object 作为响应，通常用于发送 XML 或 JSON 数据作为响应。
- @PathVariable - 用于将动态值从 URI 映射到处理程序方法参数。
- @Autowired - 用于在 spring bean 中自动装配依赖项。
- @Qualifier - 使用 @Autowired 注解，以避免在存在多个 bean 类型实例时出现混淆。
- @Scope - 用于配置 spring bean 的范围。

@Configuration, @ComponentScan 和 @Bean - 用于基于 java 的配置。
@Aspect, @Before, @After, @Around, @Pointcut - 用于切面编程 (AOP)

28. DispatcherServlet

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

29. @Component, @Controller, @Repository, @Service 有何区别？

@Component: 这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller: 这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service: 此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository: 这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

30. @Autowired 注解解释

通过类型来实现自动注入bean。和@Qualifier注解配合使用可以实现根据name注入bean。

31. @Qualifier 注解解释

和@Autowire一块使用，在同一类型的bean有多个的情况下可以实现根据name注入的需求

32. @Resource 注解解释

默认是根据name注入bean的，可以通过设置类型来实现通过类型来注入

33. @Controller 注解

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

34. @RequestMapping 注解

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。
此注解可应用于两个级别：

类级别：映射请求的 URL

方法级别：映射 URL 以及 HTTP 请求方法

35. @Required 注解有什么用？

@Required 应用于 bean 属性 setter 方法。此注解仅指示必须在配置时使用 bean 定义中的显式属性值或使用自动装配填充受影响的 bean 属性。如果尚未填充受影响的 bean 属性，则容器将抛出 BeanInitializationException。

36. @Autowire和@Resource区别

对比项	@Autowire	@Resource
注解来源	Spring注解	JDK注解(JSR-250标准注解，属于J2EE)
装配方式	优先按类型	优先按名称
属性	required	name、type
作用范围	字段、setter方法、构造器	字段、setter方法

37. SpringMVC中的拦截器和Servlet中的filter有什么区别？

首先最核心的一点他们的拦截侧重点是不同的，SpringMVC中的拦截器是依赖DK的反射实现的，SpringMVC的拦截器主要是进行拦截请求，通过对Handler进行处理的时候进行拦截，先声明的拦截器中的preHandle方法会先执行，然而它的postHandle方法（他是介于处理完业务之后和返回结果之前）和afterCompletion方法却会后执行。并且Spring的拦截器是按照配置的先后顺序进行拦截的。

而Servlet的filter是基于函数回调实现的过滤器，Filter主要是针对URL地址做一个编码的事情、过滤掉没用的参数、安全校验（比较泛的，比如登录不登录之类）

38. 讲讲Spring加载流程。

初始化环境—>加载配置文件—>实例化Bean—>调用Bean显示信息

首先从大的几个核心步骤来去说明，因为Spring中的具体加载过程和用到的类实在是太多了。

1、首先是先从AbstractBeanFactory中去调用doGetBean (name, requiredType, final Object[] args, boolean typeCheckOnly 【这个是判断进行创建bean还是仅仅用来做类型检查】) 方法，然后第一步要做的就是先去对传入的参数name进行做转换，因为有可能传进来的name="&XXX"之类，需要去除&符号

2、然后接着是去调用getSingleton () 方法，其实在上一个面试题中已经提到了这个方法，这个方法就是利用“三级缓存”来去避免循环依赖问题的出现的。【这里补充一下，只有在是单例的情况下才会去解决循环依赖问题】

3、对从缓存中拿到的bean其实是最原始的bean，还未长大，所以这里还需要调用getObjectForBeanInstance (Object beanInstance, String name, String beanName, RootBeanDefinition mbd) 方法去进行实例化。

4、然后会解决单例情况下尝试去解决循环依赖，如果isPrototypeCurrentlyInCreation (beanName) 返回为true的话，会继续下一步，否则throw new BeanCurrentlyInCreationException(beanName);

- 5、因为第三步中缓存中如果没有数据的话，就直接去parentBeanFactory中去获取bean，然后判断containsBeanDefinition (beanName) 中去检查已加载的XML文件中是否包含有这样的bean存在，不存在的话递归去getBean () 获取，如果没有继续下一步
- 6、这一步是吧存储在XML配置文件中的GernericBeanDifinition转换为RootBeanDifinition对象。这里主要进行一个转换，如果父类的bean不为空的话，会一并合并父类的属性
- 7、这一步核心就是需要跟这个Bean有关的所有依赖的bean都要被加载进来，通过刚刚的那个RootBeanDifinition对象去拿到所有的beanName,然后通过registerDependentBean (dependsOnBean, beanName) 注册bean的依赖
- 8、然后这一步就是会根据我们在定义bean的作用域的时候定义的作用域是什么，然后进行判断在进行不同的策略进行创建（比如isSingleton、isPrototype）
- 9、这个是最最后一步的类型装换，会去检查根据需要的类型是否符合bean的实际类型去做一个类型转换。Spring中提供了许多的类型转换器

39. Spring的IOC理解：

- 1、IOC就是控制反转，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。DI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IoC容器来动态注入对象需要的外部资源。
- 2、最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生成，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。
- 3、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。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

40. 列举 IoC 的一些好处

- IoC的一些好处是：
- 它将最小化应用程序中的代码量。
 - 它将使您的应用程序易于测试，因为它不需要单元测试用例中的任何单例或 JNDI 查找机制。
 - 它以最小的影响和最少的侵入机制促进松耦合。
 - 它支持即时的实例化和延迟加载服务。

41. Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

```

interface Fruit {
    public abstract void eat();
}

class Apple implements Fruit {
    public void eat(){
        System.out.println("Apple");
    }
}

class Orange implements Fruit {
    public void eat(){
        System.out.println("Orange");
    }
}

class Factory {
    public static Fruit getInstance(String className) {
        Fruit f=null;
        try {
            f=(Fruit)Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return f;
    }
}

class Client {
    public static void main(String[] args) {
        Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple");
        if (f!=null){
            f.eat();
        }
    }
}

```

42. Spring通知有哪些类型？

- 1、**前置通知 (Before advice)**：在某连接点 (join point) 之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
- 2、**返回后通知 (After returning advice)**：在某连接点 (join point) 正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
- 3、**抛出异常后通知 (After throwing advice)**：在方法抛出异常退出时执行的通知。
- 4、**后通知 (After (finally) advice)**：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
- 5、**环绕通知 (Around Advice)**：包围一个连接点 (join point) 的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和Boss4，都只提供环绕通知。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice

before advice

target method 执行

around after advice

after advice

afterReturning

②有异常情况下的执行顺序:

around before advice

before advice

target method 执行

around after advice

after advice

afterThrowing:异常发生

java.lang.RuntimeException: 异常发生

43. 解释一下Spring AOP里面的几个名词:

1、切面 (Aspect) : 被抽取的公共模块，可能会横切多个对象。在Spring AOP中，切面可以使用通用类（基于模式的风格）或者在普通类中以 @Aspect 注解来实现。

2、连接点 (Join point) : 指方法，在Spring AOP中，一个连接点总是代表一个方法的执行。

3、通知 (Advice) : 在切面的某个特定的连接点 (Join point) 上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

4、切入点 (Pointcut) : 切入点是指我们要对哪些join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add、search。

5、引入 (Introduction) : (也被称为内部类型声明 (inter-type declaration))。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

6、目标对象 (Target Object) : 被一个或者多个切面 (aspect) 所通知 (advise) 的对象。也有人把它叫做 被通知 (adviced) 对象。既然Spring AOP是通过运行时代理实现的，这个对象永远是一个被代理 (proxied) 对象。

7、织入 (Weaving) : 指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

切入点 (pointcut) 和连接点 (join point) 匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。切入点使得定位通知 (advice) 可独立于OO层次。例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

44. Spring AOP的实现原理。

AOP (Aspect-Oriented Programming, 面向方面编程)：是OOP的补充和完善。OOP引入了封装、继承、多态性等建立一种对象层次结构（从上到下的关系）。当需要为分散的对象引入公共行为的时候（从左到右的关系），OOP就显得无能为力。例如：日志功能。日志代码往往水平的散步所有对象层次中，与对象的核心功能毫无关系。这种代码被称为横切（cross-cutting）代码还有像安全性、异常处理、透明的持续性等都称为横切代码。在OOP设计中，它们导致了大量代码的重复，不利于模块的重用。

AOP与OOP相反，利用“横切”技术将影响多个类的公共行为封装到一个可重用模块，称为Aspect。简单点，就是将那些与业务无关，却被业务模块所共同调用的逻辑封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”Spring提供了两种方式生成代理对象：JDKProxy和Cglib具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

45. 在Spring AOP 中，关注点和横切关注的区别是什么？

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

46. Spring中AOP的底层是怎么实现的？

Spring中AOP底层的实现其实是基于JDK的动态代理和cglib动态创建类进行动态代理来实现的：

1、第一种基于JDK的动态代理的原理是：

需要用到的几个关键成员 InvocationHandler（你想要通过动态代理生成的对象都必须实现这个接口）真实的需要代理的对象（帮你代理的对象）Proxy对象（是JDK中java.lang.reflect包下的）

下面是具体如何动态利用这三个组件生成代理对象

- 首先你的真是要代理的对象必须要实现InvocationHandler这个接口，并且覆盖这个接口的 invoke(Object proxyObject, Method method, Object[] args)方法，这个Invoker中方法的参数的 proxyObject就是你要代理的真实目标对象，方法调用会被转发到该类的invoke()方法，method是真实对象中调用方法的Method类，Object[] args是真实对象中调用方法的参数
- 然后通过Proxy类去调用newProxyInstance(classLoader, interfaces, handler)方法，classLoader是指真实代理对象的类加载器,interfaces是指真实代理对象需要实现的接口，还可以同时指定多个接口，handler方法调用的实际处理器（其实就是帮你代理的那个对象），代理对象的方法调用都会转发到这里，然后直接就能生成你想要的对象类了。

47. Spring事务的实现方式和实现原理：

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。

1、Spring事务的种类：

spring支持编程式事务管理和声明式事务管理两种方式：

微信搜索公众号：Java专栏，获取最新面试手册

①编程式事务管理使用TransactionTemplate。

②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

2、spring的事务传播行为：

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。

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 PROPAGATIONQUIRES_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。

3、Spring中的隔离级别：

① ISOLATION_DEFAULT：这是个 PlatformTransactionManager 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。

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。

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。

讲讲Spring事务的传播属性。

1、PROPAGATION_REQUIRED,那么由于执行ServiceA.methodA的时候, ServiceA.methodA已经起了事务, 这时调用ServiceB.methodB, ServiceB.methodB看到自己已经运行在ServiceA.methodA的事务内部, 就不再起新的事务。而假如ServiceA.methodA运行的时候发现自己没有在事务中, 他就会为自己分配一个事务。这样, 在ServiceA.methodA或者在ServiceB.methodB内的任何地方出现异常, 事务都会被回滚。即使ServiceB.methodB的事务已经被提交, 但是ServiceA.methodA在接下来fail要回滚, ServiceB.methodB也要回滚。

2、PROPAGATION_SUPPORTS —— 支持当前事务, 如果当前没有事务, 就以非事务方式执行。

3、PROPAGATION_MANDATORY ——支持当前事务, 如果当前没有事务, 就抛出异常。

4、PROPAGATION_REQUIRES_NEW ——支持当前事务, 如果当前没有事务, 就将当前事务挂起。如ServiceA.methodA的事务级别为PROPAGATION_REQUIRED, ServiceB.methodB的事务级别为PROPAGATION_REQUIRES_NEW, 那么当执行到ServiceB.methodB的时候, ServiceA.methodA所在的事务就会挂起, ServiceB.methodB会起一个新的事务, 等待ServiceB.methodB的事务完成以后, A才继续执行。他与PROPAGATION_REQUIRED的事务区别在于事务的回滚程度了。因为ServiceB.methodB是新起一个事务, 那么就是存在两个不同的事务。如果ServiceB.methodB已经提交, 那么ServiceA.methodA失败回滚, ServiceB.methodB是不会回滚的。如果ServiceB.methodB失败回滚, 如果他抛出的异常被ServiceA.methodA捕获, ServiceA.methodA事务仍然可能提交。

5、PROPAGATION_NOT_SUPPORTED —— 以非事务方式执行当前操作, 如果当前存在事务, 就把事务挂起来。 /

6、PROPAGATION_NEVER —— 以非事务方式执行, 如果当前存在事务, 则抛异常。

7、PROPAGATION_NESTED—— 如果当前存在事务, 则在嵌套事务内执行, 关键是savepoint。如果当前没有事务, 则进行与PROPAGATION_REQUIRED类似的操作。与PROPAGATION_REQUIRES_NEW的区别是NESTED的事务和他的父事务是相依的, 它的提交是要等父事务一块提交。也就是说, 如果父事务最后回滚, 它也要回滚。

48. Spring如何管理事务的。

Spring事务管理主要包括3个接口, Spring事务主要由以下三个共同完成的:

1、**PlatformTransactionManager**: 事务管理器, 主要用于平台相关事务的管理。主要包括三个方法: ①、commit: 事务提交。②、rollback: 事务回滚。③、getTransaction: 获取事务状态。

2、**TransacitonDefinition**: 事务定义信息, 用来定义事务相关属性, 给事务管理器PlatformTransactionManager使用这个接口有下面四个主要方法: ①、getIsolationLevel: 获取隔离级别。②、getPropagationBehavior: 获取传播行为。③、getTimeout获取超时时间。④、isReadOnly: 是否只读 (保存、更新、删除时属性变为false--可读写, 查询时为true--只读) 事务管理器能够根据这个返回值进行优化, 这些事务的配置信息, 都可以通过配置文件进行配置。

3、**TransationStatus**: 事务具体运行状态, 事务管理过程中, 每个时间点事务的状态信息。例如: ①、hasSavepoint(): 返回这个事务内部是否包含一个保存点。②、isCompleted(): 返回该事务是否已完成, 也就是说, 是否已经提交或回滚。③、isNewTransaction(): 判断当前事务是否是一个新事务。

50. Spring框架的事务管理有哪些优点?

它为不同的事务API如JTA, JDBC, Hibernate, JPA和DO, 提供一个不变的编程模式。它为编程式事务管理提供了一套简单的API而不是一些复杂的事务API如它为编程式事务管理提供了一套简单的API而不是一些复杂的事务API如它支持声明式事务管理。它支持声明式事务管理。它和Spring各种数据访问抽象层很好得集成。它和Spring各种数据访问抽象层很好得集成。

51. 怎样用注解的方式配置Spring?

1. 配置文件中开启扫描

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 开启注解 配置扫描路径 -->
    <context:component-scan base-package="com.dpb.javabean"/>
    <!-- 如果有多个路径 ,号隔开
        <context:component-scan base-
package="com.dpb.javabean,com.dpb.factory"/>
    -->
</beans>
```

2. 加载容器

如果是web项目的话，在web.xml文件添加如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

如果是java项目可以通过BeanFactory或ApplicationContext来直接加载Spring容器

```
BeanFactory bf =
    new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
// 或者
ApplicationContext =
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

52. Spring怎么配置事务（具体说出一些关键的xml 元素）。

配置事务的方法有两种：

1、基于XML的事务配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- from the file 'context.xml' -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <!-- 数据元信息 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <!-- 管理事务的类，指定我们用谁来管理我们的事务-->
    <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 首先我们要把服务对象声明成一个bean 例如HelloService -->
    <bean id="helloService" class="com.yintong.service.HelloService"/>

    <!-- 然后是声明一个事物建议tx:advice,spring为我们提供了事物的封装，这个就是封装在了
<tx:advice/>中 -->
    <!-- <tx:advice/>有一个transaction-manager属性，我们可以用它来指定我们的事物由谁来管
理。
    默认：事务传播设置是 REQUIRED，隔离级别是DEFAULT -->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!-- 配置这个事物建议的属性 -->
        <tx:attributes>
            <!-- 指定所有get开头的方法执行在只读事务上下文中 -->
            <tx:method name="get*" read-only="true"/>
            <!-- 其余方法执行在默认的读写上下文中 -->
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- 我们定义一个切面，它匹配FooService接口定义的所有操作 -->
    <aop:config>
        <!-- <aop:pointcut/>元素定义AspectJ的切面表示法，这里是表示
com.yintong.service.helloService包下的任意方法。 -->
```

```

<aop:pointcut id="helloServiceOperation" expression="execution(*
com.yintong.service.helloService.*(..))"/>
<!-- 然后我们用一个通知器: <aop:advisor/>把这个切面和tx:advice绑定在一起, 表示当这个
切面: fooServiceOperation执行时tx:advice定义的通知逻辑将被执行 -->
<aop:advisor advice-ref="txAdvice" pointcut-ref="helloServiceOperation"/>
</aop:config>

</beans>

```

2、基于注解方式的事务配置。

@Transactional：直接在Java源代码中声明事务的做法让事务声明和将受其影响的代码距离更近了，而且一般来说不会有不恰当的耦合的风险，因为，使用事务性的代码几乎总是被部署在事务环境中。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="helloService" class="com.yintong.service.HelloService"/>
    <bean id="txManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!-- 配置注解事务 -->
    <tx:annotation-driven transaction-manager="txManager"/>
</beans>

```

主要在类中定义事务注解@Transactional，如下：

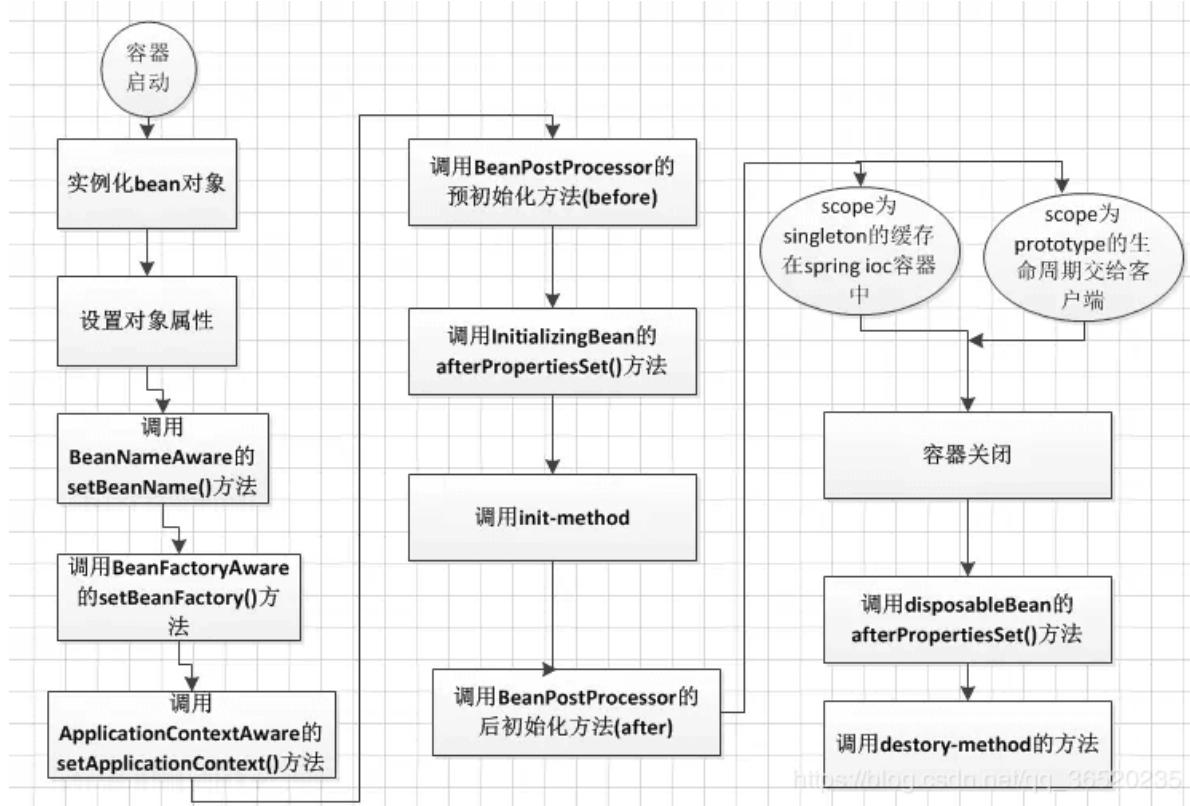
```

//{@Transactional} 注解可以声明在类上, 也可以声明在方法上。在大多数情况下, 方法上的事务会首先执
行
@Transactional(readOnly = true)
public class HelloService{
    public Foo getFoo(String fooName) {
    }
    //{@Transactional} 注解的事务设置将优先于类级别注解的事务设置    propagation:可选的传播性
    设置
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)

    public void updateFoo(He1 he1) {
    }
}

```

53. Springbean的生命周期



通过这张图能大致看懂spring的生命周期，详解：

首先会先进行实例化bean对象 然后是进行对bean的一个属性进行设置 接着是对BeanNameAware（其实就是为了让Spring容器来获取bean的名称）、BeanFactoryAware（让bean的BeanFactory调用容器的服务）、ApplicationContextAware（让bean当前的applicationContext可以来取调用Spring容器的服务）

然后是实现BeanPostProcessor这个接口中的两个方法，主要是对调用接口的前置初始化 postProcessBeforeInitialization 这里主要是对xml中自己定义的初始化方法 init-method = “xxxx” 进行调用 然后是继续对BeanPostProcessor这个接口中的后置初始化方法进行一个调用 postProcessAfterInitialization () 其实到这一步，基本上这个bean的初始化基本已经完成，就处于就绪状态 然后就是当Spring容器中如果使用完毕的话，就会调用destory () 方法 最后会去执行我们自己定义的销毁方法来进行销毁，然后结束生命周期。

54. spring容器的bean什么时候被实例化？

1、如果你使用BeanFactory作为Spring Bean的工厂类，则所有的bean都是在第一次使用该Bean的时候实例化

2、如果你使用ApplicationContext作为Spring Bean的工厂类，则又分为以下几种情况：如果bean的 scope是singleton的，并且lazy-init为false（默认是false，所以可以不用设置），则 ApplicationContext启动的时候就实例化该Bean，并且将实例化的Bean放在一个map结构的缓存中，下次再使用该 Bean的时候，直接从这个缓存中取。如果bean的scope是singleton的，并且lazy-init为 true，则该Bean的实例化是在第一次使用该Bean的时候进行实例化。如果bean的scope是prototype的，则该Bean的实例化是在第一次使用该Bean的时候进行实例化。

55. 说说 BeanFactory 和 ApplicationContext 的区别？什么是延迟实例化，它的优缺点是什么？

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中 ApplicationContext是BeanFactory的子接口。

1、BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。 ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。
- 统一的资源文件访问方式。
- 提供在监听器中注册bean的事件。
- 同时加载多个配置文件。
- 载入多个（有继承关系）上下文，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

2.

- BeanFactory采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFactory加载后，直至第一次使用调用getBean方法才会抛出异常。
- ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean，确保当你需要的时候，你就不用等待，因为它们已经创建好了。
- 相对于基本的BeanFactory， ApplicationContext唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。
- BeanFactory通常以编程的方式被创建， ApplicationContext还能以声明的方式创建，如使用ContextLoader。
- BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是： BeanFactory需要手动注册，而 ApplicationContext则是自动注册。

56. BeanFactory - BeanFactory 实现举例

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从正真的应用代码中分离。

最常用的BeanFactory 实现是XmlBeanFactory 类。

57. XMLBeanFactory

最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

58. Spring中Bean的作用域有哪些?

singleton作用域

在默认情况下,spring的ApplicationContext容器在启动时,自动实例化所有singleton的Bean并缓存于容器中.虽然启动时会花费一些时间,但带来两个好处:首先对Bean提前的实例化操作会及早发现一些潜在的配置问题.其次Bean以缓存的方式保存,当运行时使用到该Bean时就无须再实例化了,加快了运行效率.如果用户不希望在容器启动时提前实例化singleton的Bean,可以通过lazy-init属性进行控制.

prototype

在默认情况下,spring容器在启动时不实例化prototype的Bean.此外,spring容器将prototype的Bean交给调用者后,就不再管理它的生命周期.

request作用域

每次HTTP请求都会创建一个新的Bean,HTTP请求处理完毕后,销毁这个Bean.该作用域仅适用于webApplicationContext环境.

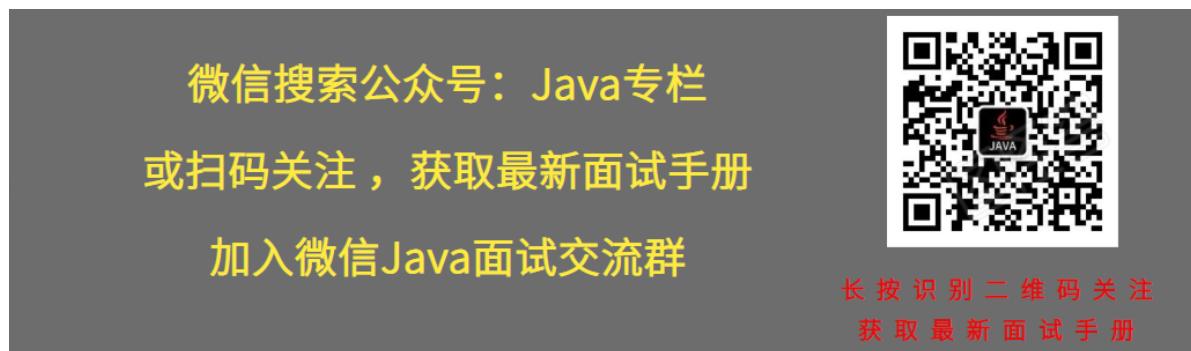
session作用域

同一个HTTP session共享一个Bean,不同HTTP session使用不同的Bean,当HTTP Session结束后,实例才被销毁.该作用域仅适用于webApplicationContext环境

globalSession作用域

同一个全局session共享一个Bean,一般用于portlet应用环境,该作用域仅适用于webApplicationContext环境.

MyBatis面试题



1. 什么是Mybatis?

- 1、Mybatis是一个半ORM (对象关系映射) 框架, 它内部封装了JDBC, 开发时只需要关注SQL语句本身, 不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql, 可以严格控制sql执行性能, 灵活性高。
- 2、MyBatis 可以使用 XML 或注解来配置和映射原生信息, 将 POJO 映射成数据库中的记录, 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。
- 3、通过xml文件或注解的方式将要执行的各种 statement 配置起来, 并通过java对象和 statement 中 sql 的动态参数进行映射生成最终执行的sql语句, 最后由mybatis框架执行sql并将结果映射为java对象并返回。 (从执行sql到返回result的过程) 。

2. Mybatis的优点：

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

3. MyBatis框架的缺点：

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

4、MyBatis框架适用场合：

- 1、MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。
- 2、对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

5、MyBatis与Hibernate有哪些不同？

相同点

- 都是对jdbc的封装，都是持久层的框架，都用于dao层的开发。

不同点

- 映射关系
 - MyBatis 是一个半自动映射的框架，配置Java对象与sql语句执行结果的对应关系，多表关联关系配置简单
 - Hibernate 是一个全表映射的框架，配置Java对象与数据库表的对应关系，多表关联关系配置复杂

SQL优化和移植性

- Hibernate 对SQL语句封装，提供了日志、缓存、级联（级联比 MyBatis 强大）等特性，此外还提供 HQL (Hibernate Query Language) 操作数据库，数据库无关性支持好，但会多消耗性能。如果项目需要支持多种数据库，代码开发量少，但SQL语句优化困难。
- MyBatis 需要手动编写 SQL，支持动态 SQL、处理列表、动态生成表名、支持存储过程。开发工作量相对大些。直接使用SQL语句操作数据库，不支持数据库无关性，但sql语句优化容易。

6. Mybatis 比 IBatis 比较大的几个改进是什么？

- 1、有接口绑定,包括注解绑定 sql 和 xml 绑定 Sql
- 2、动态 sql 由原来的节点配置变成 OGNL 表达式 3) 在一对一,一对多的时候引进了association,在一对多的时候引入了 collection 节点,不过都是在 resultMap 里面配置

7. ORM是什么

ORM (Object Relational Mapping)，对象关系映射，是一种为了解决关系型数据库数据与简单Java对象 (POJO) 的映射关系的技术。简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系型数据库中。

8. 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

- Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。
- 而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

9. 传统JDBC开发存在什么问题？

- 频繁创建数据库连接对象、释放，容易造成系统资源浪费，影响系统性能。可以使用连接池解决这个问题。但是使用jdbc需要自己实现连接池。
- sql语句定义、参数设置、结果集处理存在硬编码。实际项目中sql语句变化的可能性较大，一旦发生变化，需要修改java代码，系统需要重新编译，重新发布。不好维护。
- 使用PreparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。
- 结果集处理存在重复代码，处理麻烦。如果可以映射成Java对象会比较方便。

10. JDBC编程有哪些不足之处，MyBatis是如何解决的？

1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

- 解决：在mybatis-config.xml中配置数据链接池，使用连接池管理数据库连接。

2、Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。 -

- 解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

3、向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

- 解决： Mybatis自动将java对象映射至sql语句。

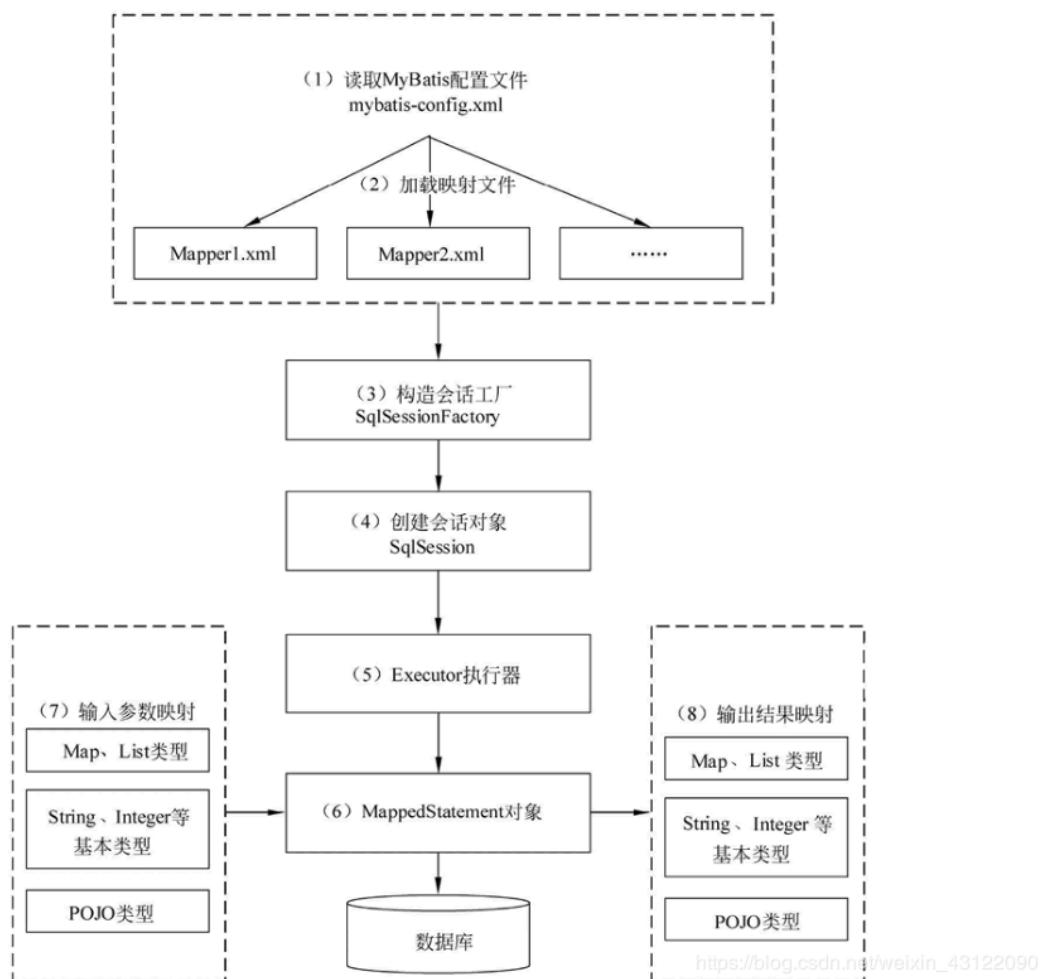
4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

- 解决： Mybatis自动将sql执行结果映射至java对象。

11. MyBatis编程步骤是什么样的？

- 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

12. 请说说MyBatis的工作原理



1、读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

2、加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在 MyBatis 配置文件 mybatis-config.xml 中加载。mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。

3、构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

4、创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。

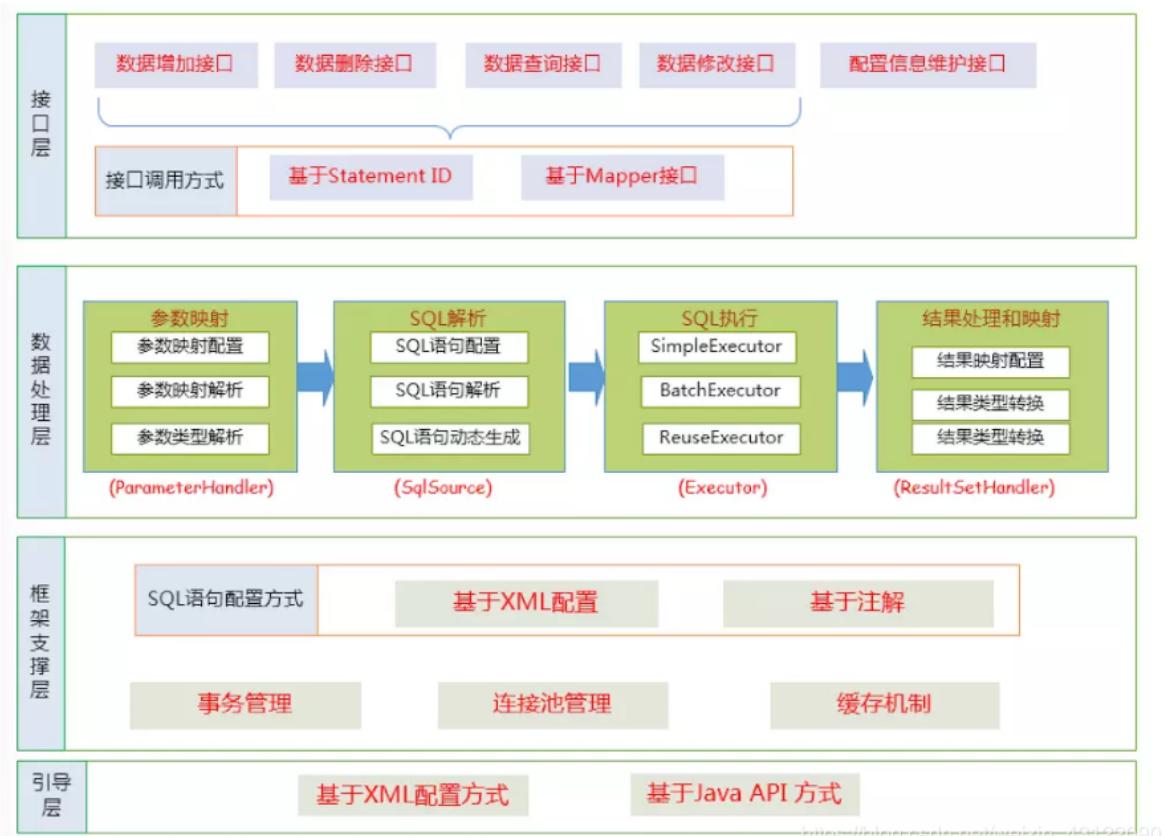
5、Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

6、MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信息。

7、输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过程。

8、输出结果映射：输出结果类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

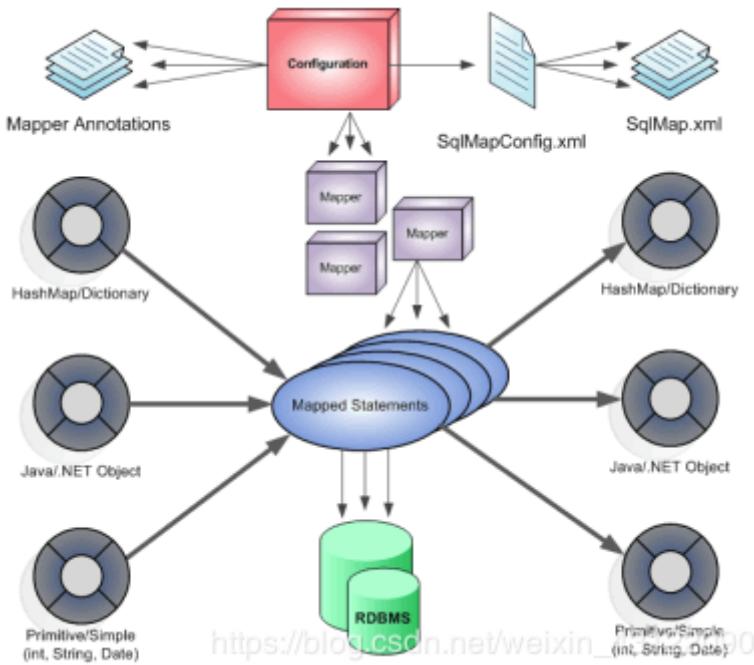
13. MyBatis的功能架构是怎样的



我们把Mybatis的功能架构分为三层：

- API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接到调用请求就会调用数据处理层来完成具体的数据处理。
- 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。
- 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

14. MyBatis的框架架构设计是怎么样的



这张图从上往下看。MyBatis的初始化，会从mybatis-config.xml配置文件，解析构造成Configuration这个类，就是图中的红框。

1、加载配置：配置来源于两个地方，一处是配置文件，一处是Java代码的注解，将SQL的配置信息加载成为一个个MappedStatement对象（包括了传入参数映射配置、执行的SQL语句、结果映射配置），存储在内存中。

2、SQL解析：当API接口层接收到调用请求时，会接收到传入SQL的ID和传入对象（可以是Map、JavaBean或者基本数据类型），Mybatis会根据SQL的ID找到对应的MappedStatement，然后根据传入参数对象对MappedStatement进行解析，解析后可以得到最终要执行的SQL语句和参数。

3、SQL执行：将最终得到的SQL和参数拿到数据库进行执行，得到操作数据库的结果。

4、结果映射：将操作数据库的结果按照映射的配置进行转换，可以转换成HashMap、JavaBean或者基本数据类型，并将最终结果返回。

15. 什么是DBMS

数据库管理系统(database management system)是一种操纵和管理数据库的大型软件，用于建立、使用和维护数据库，简称dbms。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过dbms访问数据库中的数据，数据库管理员也通过dbms进行数据库的维护工作。它可使多个应用程序和用户用不同的方法在同时或不同时刻去建立，修改和询问数据库。DBMS提供数据定义语言DDL (Data Definition Language) 与数据操作语言DML (Data Manipulation Language)，供用户定义数据库的模式结构与权限约束，实现对数据的追加权、删除等操作。

16. 为什么需要预编译

定义：

- SQL 预编译指的是数据库驱动在发送 SQL 语句和参数给 DBMS 之前对 SQL 语句进行编译，这样 DBMS 执行 SQL 时，就不需要重新编译。

为什么需要预编译

- JDBC 中使用对象 PreparedStatement 来抽象预编译语句，使用预编译。预编译阶段可以优化 SQL 的执行。预编译之后的 SQL 多数情况下可以直接执行，DBMS 不需要再次编译，越复杂的 SQL，编译的复杂度将越大，预编译阶段可以合并多次操作为一个操作。同时预编译语句对象可以重复利用。把一个 SQL 预编译后产生的 PreparedStatement 对象缓存下来，下次对于同一个 SQL，可以直接使用这个缓存的 PreparedStatement 对象。Mybatis默认情况下，将对所有的 SQL 进行预编译。
- 还有一个重要的原因，复制SQL注入

17. #{}和\${}的区别是什么？

- `#{}是预编译处理`
- `${} 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。`

18. 当实体类中的属性名和表中的字段名不一样，怎么办？

第1种：通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
<select id="selectorder" parameterType="int" resultType="me.gacl.domain.order">
    select order_id id, order_no orderno ,order_price price from orders where
    order_id=#{id};</select>
```

第2种：通过 `<resultMap>` 来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder" parameterType="int" resultMap="orderresultmap">select *
from orders where order_id=#{id}</select>
<resultMap type="me.gacl.domain.order" id="orderresultmap">      <!-用id属性来映射主
键字段->      <id property="id" column="order_id">
      <!-用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性->
      <result property = "orderno" column ="order_no"/>      <result property="price"
column="order_price" /></resultMap>
```

19. 模糊查询like语句该怎么写？

第1种：在Java代码中添加sql通配符。

```
string wildcardname = "%smi%";
List<name> names = mapper.selectlike(wildcardname);

<select id="selectlike">select * from foo where bar like #{value}</select>
```

第2种：在sql语句中拼接通配符，会引起sql注入

微信搜索公众号：Java专栏，获取最新面试手册

```
string wildcardname = "smi";
List<Name> names = mapper.selectlike(wildcardname);

<select id="selectlike">select * from foo where bar like "%"#{value}%"</select>
```

20. Mybatis都有哪些Executor执行器？它们之间的区别是什么？

- Mybatis有三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。
- **SimpleExecutor**: 每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。
- **ReuseExecutor**: 执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。
- **BatchExecutor**: 执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

21. Mybatis中如何指定使用哪一种Executor执行器？

- 在Mybatis配置文件中，在设置（settings）可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数，如 SqlSession openSession(ExecutorType execType)。
- 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）；BATCH 执行器将重用语句并执行批量更新。

22. 通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao 接口，就是人们常说的 Mapper 接口，接口的全限名，就是映射文件中的 namespace 的值，接口的方法名，就是映射文件中 mappedStatement 的 id 值，接口方法内的参数，就是传递给 sql 的参数。Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MappedStatement，举例： com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到 namespace 为 com.mybatis3.mappers.StudentDao 下面 id = findStudentById 的 MappedStatement。在 MyBatis 中，每一个 <select>、<insert>、<update>、<delete> 标签，都会被解析为一个 MappedStatement 对象。

Dao 接口里的方法可以重载，但是Mybatis的XML里面的ID不允许重复。

Mybatis版本3.3.0，亲测如下：

```
/**  
 * Mapper接口里面方法重载  
 */  
public interface StuMapper {  
  
    List<Student> getAllStu();  
  
    List<Student> getAllStu(@Param("id") Integer id);  
}
```

然后在 `StuMapper.xml` 中利用Mybatis的动态sql就可以实现。

```
<select id="getAllStu" resultType="com.pojo.Student">  
    select * from student  
    <where>  
        <if test="id != null">  
            id = #{id}  
        </if>  
    </where>  
</select>
```

能正常运行，并能得到相应地结果，这样就实现了在Dao接口中写重载方法。

Mybatis 的 Dao 接口可以有多个重载方法，但是多个接口对应的映射必须只有一个，否则启动会报错。

23. Mybatis是如何进行分页的？分页插件的原理是什么？

1、Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用 Mybatis 的分页插件。

2、分页插件的原理：实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。

举例：

```
select * from student, 拦截 sql 后重写为: select t.* from (select * from student) t  
limit 0, 10
```

24. Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用 `<resultMap>` 标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

25. 如何获取自动生成的(主)键值?

insert 方法总是返回一个int值，这个值代表的是插入的行数。

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。

示例：

```
<insert id="insertname" usegeneratedkeys="true" keyproperty="id">
    insert into names (name) values (#{name})
</insert>
```

```
name name = new name();
name.setName("fred");

int rows = mapper.insertname(name);
// 完成后，id已经被设置到对象中
System.out.println("rows inserted = " + rows);
System.out.println("generated key value = " + name.getId());
```

26. 在mapper中如何传递多个参数?

方法1：顺序传参法

```
public User selectUser(String name, int deptId);

<select id="selectUser" resultMap="UserResultMap">
    select * from user
    where user_name = #{0} and dept_id = #{1}
</select>
```

- #{}里面的数字代表传入参数的顺序。
- 这种方法不建议使用，sql层表达不直观，且一旦顺序调整容易出错。

方法2：@Param注解传参法

```
public User selectUser(@Param("userName") String name, int @Param("deptId") deptId);

<select id="selectUser" resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- #{}里面的名称对应的是注解@Param括号里面修饰的名称。
- 这种方法在参数不多的情况下还是比较直观的，（推荐使用）。

方法3：Map传参法

```
public User selectUser(Map<String, Object> params);

<select id="selectUser" parameterType="java.util.Map" resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- #{}里面的名称对应的是Map里面的key名称。
- 这种方法适合传递多个参数，且参数易变能灵活传递的情况。（推荐使用）。

方法4：Java Bean传参法

```
public User selectUser(User user);

<select id="selectUser" parameterType="com.jourwon.pojo.User"
resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- #{}里面的名称对应的是User类里面的成员属性。
- 这种方法直观，需要建一个实体类，扩展不容易，需要加属性，但代码可读性强，业务逻辑处理方便，推荐使用。（推荐使用）。

27. Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？

- Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true | false。
- 它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。
- 当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。
微信搜索公众号：Java专栏，获取最新面试手册

28. Mybatis动态sql有什么用？执行原理？有哪些动态sql？

Mybatis动态sql可以在Xml映射文件内，以标签的形式编写动态sql，执行原理是根据表达式的值完成逻辑判断并动态拼接sql的功能。

Mybatis提供了9种动态sql标签：`trim|where|set|foreach|if|choose|when|otherwise|bind`。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能。

29. Xml映射文件中，除了常见的select|insert|update|delete标签之外，还有哪些标签？

还有很多其他的标签，、、、、，加上动态sql的9个标签，

`trim|where|set|foreach|if|choose|when|otherwise|bind`等，其中为sql片段标签，通过标签引入sql片段，为不支持自增的主键生成策略标签。

30. Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；

原因就是namespace+id是作为Map `<string,MapperStatement>` 的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

31. 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

32. 一对一、一对多的关联查询？

```
<mapper namespace="com.lcb.mapping.userMapper">
    <!--association 一对关联查询 -->
    <select id="getClass" parameterType="int" resultMap="classesResultMap">
        select * from class c,teacher t where c.teacher_id=t.t_id and c.c_id=#{id}
    </select>

    <resultMap type="com.lcb.user.Classes" id="classesResultMap">
        <!-- 实体类的字段名和数据表的字段名映射 -->
        <id property="id" column="c_id"/>
```

```

<result property="name" column="c_name"/>
<association property="teacher" javaType="com.lcb.user.Teacher">
    <id property="id" column="t_id"/>
    <result property="name" column="t_name"/>
</association>
</resultMap>

<!--collection 一对多关联查询 -->
<select id="getClass2" parameterType="int" resultMap="classesResultMap2">
    select * from class c,teacher t,student s where c.teacher_id=t.t_id and
c.c_id=s.class_id and c.c_id=#{id}
</select>

<resultMap type="com.lcb.user.Classes" id="classesResultMap2">
    <id property="id" column="c_id"/>
    <result property="name" column="c_name"/>
    <association property="teacher" javaType="com.lcb.user.Teacher">
        <id property="id" column="t_id"/>
        <result property="name" column="t_name"/>
    </association>

    <collection property="student" ofType="com.lcb.user.Student">
        <id property="id" column="s_id"/>
        <result property="name" column="s_name"/>
    </collection>
</resultMap>
</mapper>

```

33. MyBatis 里面的动态 Sql 是怎么设定的?用什么语法?

MyBatis 里面的动态 Sql 一般是通过 if 节点来实现,通过 OGNL 语法来实现,但是如果要写的完整,必须配合 where,trim 节点,where 节点是判断包含节点有内容就插入 where,否则不插入,trim 节点是用来判断如果动态语句是以 and 或 or 开始,那么会自动把这个 and 或者 or 取掉。

34. Mybatis 能执行一对一、一对多的关联查询吗? 都有哪些实现方式,以及它们之间的区别?

能, Mybatis 不仅可以执行一对一、一对多的关联查询,还可以执行多对一, 多对多的关联查询, 多对一查询, 其实就是一对一查询, 只需要把 selectOne()修改为 selectList()即可;

多对多查询, 其实就是一对多查询, 只需要把 selectOne()修改为 selectList()即可。关联对象查询, 有两种实现方式, 一种是单独发送一个 sql 去查询关联对象, 赋给主对象, 然后返回主对象。另一种是使用嵌套查询, 嵌套查询的含义为使用 join 查询, 一部分列是 A 对象的属性值, 另外一部分列是关联对象 B 的属性值, 好处是只发一个 sql 查询, 就可以把主对象和其关联对象查出来。

35. MyBatis实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在resultMap里面配置association节点配置一对一的类就可以完成;

嵌套查询是先查一个表,根据这个表里面的结果的外键id,去再另外一个表里面查询数据,也是通过association配置,但另外一个表的查询通过select属性配置。

36. MyBatis实现一对多有几种方式,怎么操作的?

有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在resultMap里面的collection节点配置一对多的类就可以完成; 嵌套查询是先查一个表,根据这个表里面的结果的外键id,去再另外一个表里面查询数据,也是通过配置collection,但另外一个表的查询通过select节点配置。

37. Mybatis是否支持延迟加载? 如果支持, 它的实现原理是什么?

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载, association指的就是一对一, collection指的就是一对多查询。在Mybatis配置文件中,可以配置是否启用延迟加载 lazyLoadingEnabled=true | false。

它的原理是, 使用CGLIB创建目标对象的代理对象, 当调用目标方法时, 进入拦截器方法, 比如调用a.getB().getName(), 拦截器invoke()方法发现a.getB()是null值, 那么就会单独发送事先保存好的查询关联B对象的sql, 把B查询上来, 然后调用a.setB(b), 于是a的对象b属性就有值了, 接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了, 不光是Mybatis, 几乎所有的包括Hibernate, 支持延迟加载的原理都是一样的。

38. Mybatis的一级、二级缓存:

1) 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存, 其存储作用域为 Session, 当 Session flush 或 close 之后, 该 Session 中的所有 Cache 就将清空, 默认打开一级缓存。

2) 二级缓存与一级缓存其机制相同, 默认也是采用 PerpetualCache, HashMap 存储, 不同在于其存储作用域为 Mapper(Namespace), 并且可自定义存储源, 如 Ehcache。默认不打开二级缓存, 要开启二级缓存, 使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态), 可在它的映射文件中配置 <cache/> ;

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/U/D操作后, 默认该作用域下所有 select 中的缓存将被 clear。

39. 什么是MyBatis的接口绑定? 有哪些实现方式?

接口绑定,就是在MyBatis中任意定义接口,然后把接口里面的方法和SQL语句绑定,我们直接调用接口方法就可以,这样比起原来Session提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上 @Select、@Update等注解,里面包含SQL语句来绑定;另外一种就是通过xml里面写SQL来绑定,在这种情况下,要指定xml映射文件里面的namespace必须为接口的全路径名。当SQL语句比较简单时候,用注解绑定,当SQL语句比较复杂时候,用xml绑定,一般用xml绑定的比较多。

40. 接口绑定有几种实现方式,分别是怎么实现的?

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上@Select@Update 等注解里面包含 Sql 语句来绑定,另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名.

41. 什么情况下用注解绑定,什么情况下用 xml 绑定?

当 Sql 语句比较简单时候,用注解绑定; 当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多

42. 使用MyBatis的mapper接口调用时有哪些要求?

1、Mapper接口方法名和mapper.xml中定义的每个sql的id相同; 2、Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同; 3、Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的结果Type的类型相同; 4、Mapper.xml文件中的namespace即是 mapper接口的类路径。

43. Mapper编写有哪几种方式?

第一种：接口实现类继承SqlSessionDaoSupport：使用此种方法需要编写mapper接口， mapper接口实现类、 mapper.xml文件。

(1) 在sqlMapConfig.xml中配置mapper.xml的位置
<mappers>
 <mapper resource="mapper.xml文件的地址" />
 <mapper resource="mapper.xml文件的地址" />
</mappers>

(2) 定义mapper接口
(3) 实现类集成SqlSessionDaoSupport
mapper方法中可以this.getSqlSession()进行数据增删改查。
(4) spring 配置
<bean id=" " class="mapper接口的实现">
 <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>

第二种：使用org.mybatis.spring.mapper.MapperFactoryBean：

(1) 在sqlMapConfig.xml中配置mapper.xml的位置, 如果mapper.xml和mappe接口的名称相同且在同一个目录, 这里可以不用配置
<mappers>
 <mapper resource="mapper.xml文件的地址" />
 <mapper resource="mapper.xml文件的地址" />
</mappers>

(2) 定义mapper接口:

```
① mapper.xml 中的 namespace 为 mapper 接口的地址  
② mapper 接口中方法名和 mapper.xml 中定义的 statement 的 id 保持一致  
③ Spring 中定义  
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">  
    <property name="mapperInterface" value="mapper 接口地址" />  
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />  
</bean>
```

第三种：使用 mapper 扫描器：

(1) mapper.xml 文件编写：

mapper.xml 中的 namespace 为 mapper 接口的地址；
mapper 接口中方法名和 mapper.xml 中定义的 statement 的 id 保持一致；
如果将 mapper.xml 和 mapper 接口的名称保持一致则不用在 sqlMapConfig.xml 中进行配置。

(2) 定义 mapper 接口：

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录

(3) 配置 mapper 扫描器：

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
    <property name="basePackage" value="mapper 接口包地址" />  
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />  
</bean>
```

(4) 使用扫描器后从 spring 容器中获取 mapper 的实现对象。

44. Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能**否**定义在 A 标签的后面，还是说必须定义在 A 标签的前面？

虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不存在，此时，Mybatis 会将 A 标签标记为未解析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也可以正常解析完成了。

45. 简述 Mybatis 的插件运行原理，以及如何编写一个插件。

1、Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke() 方法，当然，只会拦截那些你指定需要拦截的方法。

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

46. Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？

1、Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。

2、Mybatis 提供了 9 种动态 sql 标签：

trim|where|set|foreach|if|choose|when|otherwise|bind。

3、其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

47. 简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系？

Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中，parameterMap 标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象；resultMap 标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个 select、insert、update、delete 标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。

SpringBoot面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. 什么是springboot ?

用来简化spring应用的初始搭建以及开发过程 使用特定的方式来进行配置 (properties或yml文件)

创建独立的spring引用程序 main方法运行

嵌入的Tomcat 无需部署war文件

简化maven配置

自动配置spring添加对应功能starter自动化配置

spring boot来简化spring应用开发，约定大于配置，去繁从简，just run就能创建一个独立的，产品级别的应用

2. Springboot 有哪些优点？

- 减少开发，测试时间和努力。
- 使用JavaConfig有助于避免使用XML。
- 避免大量的Maven导入和各种版本冲突。
- 提供意见发展方法。
- 通过提供默认值快速开始开发。
- 没有单独的Web服务器需要。这意味着你不再需要启动Tomcat, Glassfish或其他任何东西。
- 需要更少的配置 因为没有web.xml文件。只需添加用@ Configuration注释的类，然后添加用@Bean注释的方法，Spring将自动加载对象并像以前一样对其进行管理。您甚至可以将@Autowired添加到bean方法中，以使Spring自动装入需要的依赖关系中。
- 基于环境的配置 使用这些属性，您可以将您正在使用的环境传递到应用程序： - Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后，Spring将在(application{environment} .properties) 中加载后续的应用程序属性文件。

3. Spring Boot 的目录结构是怎样的？

```
cn
+- javastack
    +- MyApplication.java
    |
    +- customer
        |   +- Customer.java
        |   +- CustomerController.java
        |   +- CustomerService.java
        |   +- CustomerRepository.java
        |
    +- order
        +- Order.java
        +- OrderController.java
        +- OrderService.java
        +- OrderRepository.java
```

这个目录结构是主流及推荐的做法，而在入口类上加上 `@SpringBootApplication` 注解来开启 Spring Boot 的各项能力，如自动配置、组件扫描等。

```
package cn.javastack.MyApplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

如果你不想这么做，你也可以充分利用 `@EnableAutoConfiguration` 和 `@ComponentScan` 注解自定义你的行为，不过这不是推荐的做法。

4. 怎么理解 Spring Boot 中“约定优于配置”

Spring Boot Starter、Spring Boot Jpa 都是“约定优于配置”的一种体现。都是通过“约定优于配置”的设计思路来设计的，Spring Boot Starter 在启动的过程中会根据约定的信息对资源进行初始化；Spring Boot Jpa 通过约定的方式来自动生成 Sql，避免大量无效代码编写。

5. 如何在自定义端口上运行 Spring Boot 应用程序？

在 `application.properties` 中指定端口 `serverport=8090`。

6. Spring Boot 初始化环境变量流程？

- 1、调用 `prepareEnvironment` 方法去设置环境变量
- 2、接下来有三个方法 `getOrCreateEnvironment`, `configureEnvironment`,
`environmentPrepared`
- 3、`getOrCreateEnvironment` 去初始化系统环境变量
- 4、`configureEnvironment` 去初始化命令行参数
- 5、`environmentPrepared` 当广播到来的时候调用 `onApplicationEnvironmentPreparedEvent` 方法去使用 `postProcessEnvironment` 方法 `load yml` 和 `properties` 变量

7. Spring Boot 扫描流程？

- 1、调用 `run` 方法中的 `refreshContext` 方法
- 2、用 `AbstractApplicationContext` 中的 `refresh` 方法
- 3、委托给 `invokeBeanFactoryPostProcessors` 去处理调用链
- 4、其中一个方法 `postProcessBeanDefinitionRegistry` 会去调用 `processConfigBeanDefinitions` 解析 `beandefinitions`
- 5、在 `processConfigBeanDefinitions` 中有一个 `parse` 方法，其中有 `componentScanParser.parse` 的方法，这个方法会扫描当前路径下所有 `Component` 组件

8. Spring Boot 配置加载顺序详解

使用 Spring Boot 会涉及到各种各样的配置，如开发、测试、线上就至少 3 套配置信息了。Spring Boot 可以轻松的帮助我们使用相同的代码就能使开发、测试、线上环境使用不同的配置。

在 Spring Boot 里面，可以使用以下几种方式来加载配置。本章内容基于 Spring Boot 2.0 进行详解。

1、properties文件；

2、YAML文件；

3、系统环境变量；

4、命令行参数；

等等.....

配置属性加载的顺序如下：

- 1、开发者工具 `Devtools` 全局配置参数；
- 2、单元测试上的 `@TestPropertySource` 注解指定的参数；
- 3、单元测试上的 `@SpringBootTest` 注解指定的参数；
- 4、命令行指定的参数，如 `java -jar springboot.jar --name="Java技术栈"`；
- 5、命令行中的 `SPRING_APPLICATION_JSON` 指定参数，如 `java -Dspring.application.json='{"name":"Java技术栈"}' -jar springboot.jar`；
- 6、`ServletConfig` 初始化参数；
- 7、`ServletContext` 初始化参数；
- 8、JNDI参数（如 `java:comp/env/spring.application.json`）；
- 9、Java系统参数（来源：`System.getProperties()`）；
- 10、操作系统环境变量参数；
- 11、`RandomValuePropertySource` 随机数，仅匹配：`random.*`；
- 12、JAR包外面的配置文件参数（`application-{profile}.properties` (YAML)）
- 13、JAR包里面的配置文件参数（`application-{profile}.properties` (YAML)）
- 14、JAR包外面的配置文件参数（`application.properties` (YAML)）
- 15、JAR包里面的配置文件参数（`application.properties` (YAML)）
- 16、`@Configuration` 配置文件上 `@PropertySource` 注解加载的参数；
- 17、默认参数（通过 `SpringApplication.setDefaultProperties` 指定）；

数字小的优先级越高，即数字小的会覆盖数字大的参数值

9. Spring Boot 如何定义多套不同环境配置？

提供多套配置文件，如：

```
application.properties  
application-dev.properties  
application-test.properties  
application-prod.properties
```

然后在application.properties文件中指定当前的环境spring.profiles.active=test,这时候读取的就是application-test.properties文件。

10. Spring Boot 有哪几种读取配置的方式?

Spring Boot 可以通过

- @PropertySource
- @Value
- @Environment,
- @ConfigurationProperties

来绑定变量

11. SpringBoot 实现热部署有哪几种方式?

主要有两种方式:

- Spring Loaded
- Spring-boot-devtools

12. Spring Boot 支持哪些日志框架? 推荐和默认的日志框架是哪个

Spring Boot 支持 Java Util Logging, Log4j2, Logback 作为日志框架, 如果你使用 Starters 启动器, Spring Boot 将使用 Logback 作为默认日志框架

13. 如何重新加载Spring Boot上的更改, 而无需重新启动服务器?

这可以使用DEV工具来实现。通过这种依赖关系, 您可以节省任何更改, 嵌入式tomcat将重新启动。

Spring Boot有一个开发工具 (DevTools) 模块, 它有助于提高开发人员的生产力。Java开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。

开发人员可以重新加载Spring Boot上的更改, 而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot在发布它的第一个版本时没有这个功能。

这是开发人员最需要的功能。DevTools模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供H2数据库控制台以更好地测试应用程序。

org.springframework.boot

spring-boot-devtools

true

14. 你如何理解 Spring Boot 中的 Starters?

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

15. spring-boot-starter-parent 有什么用？

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent，spring-boot-starter-parent 主要有如下作用：

定义了 Java 编译版本为 1.8。

使用 UTF-8 格式编码。

继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。

执行打包操作的配置。

自动化的资源过滤。

自动化的插件配置。

针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

16. 什么是 Spring Boot Stater ?

启动器是一套方便的依赖没描述符，它可以放在自己的程序中。你可以一站式的获取你所需要的 Spring 和相关技术，而不需要依赖描述符的通过示例代码搜索和复制黏贴的负载。

例如，如果你想使用 Sping 和 JPA 访问数据库，只需要你的项目包含 spring-boot-starter-data-jpa 依赖项，你就可以完美进行。

17. SpringBoot常用的starter有哪些?

- 1、`spring-boot-starter-web` (嵌入tomcat和web开发需要servlet与jsp支持)
- 2、`spring-boot-starter-data-jpa` (数据库支持)
- 3、`spring-boot-starter-data-redis` (redis数据库支持)
- 4、`spring-boot-starter-data-solr` (solr搜索应用框架支持)
- 5、`mybatis-spring-boot-starter` (第三方的mybatis集成starter)

18. Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

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

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

bootstrap 配置文件有以下几个应用场景。

- 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
- 一些固定的不能被覆盖的属性；
- 一些加密/解密的场景；

19. Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

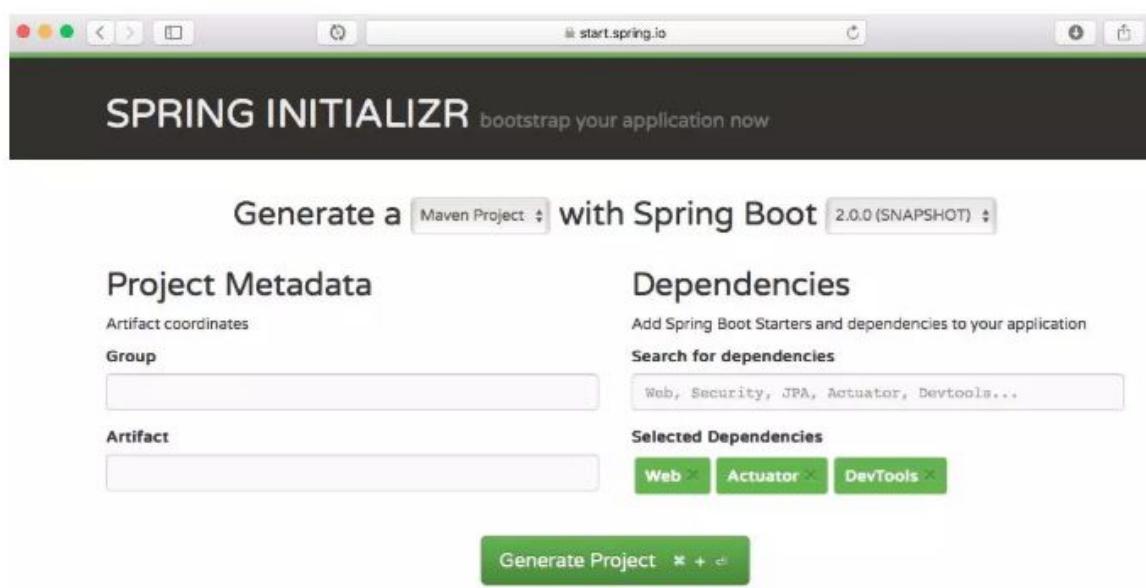
@SpringBootConfiguration: 组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration: 打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能：@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan: Spring 组件扫描。

20. 创建一个 Spring Boot Project 的最简单的方法是什么？

Spring Initializr 是启动 Spring Boot Projects 的一个很好的工具。



https://blog.csdn.net/Dome_

就像上图中所展示的一样，我们需要做一下几步：

- 1、登录 Spring Initializr，按照以下方式进行选择：
- 2、选择 com.in28minutes.springboot 为组
- 3、选择 student-services 为组件
- 4、选择下面的依赖项

Web

Actuator

DevTools

5、点击生 GenerateProject

6、将项目导入 Eclipse。文件 - 导入 - 现有的 Maven 项目

21. Spring Initializr 是创建 Spring Boot Projects 的唯一方法吗？

不是的。

Spring Initializr 让创建 Spring Boot 项目变的很容易，但是，你也可以通过设置一个 maven 项目并添加正确的依赖项来开始一个项目。

在我们的 Spring 课程中，我们使用两种方法来创建项目。

第一种方法是 start.spring.io 。

另外一种方法是在项目的标题为“Basic Web Application”处进行手动设置。

手动设置一个 maven 项目

这里有几个重要的步骤：

1、在 Eclipse 中，使用文件 - 新建 Maven 项目来创建一个新项目

2、添加依赖项。

3、添加 maven 插件。

4、添加 Spring Boot 应用程序类。

到这里，准备工作已经做好！

22. 如何集成 Spring Boot 和 ActiveMQ？

对于集成 Spring Boot 和 ActiveMQ，我们使用依赖关系。它只需要很少的配置，并且不需要样板代码。

23. 什么是 Swagger？你用 Spring Boot 实现了它吗？

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与之进行交互。因此，Swagger 消除了调用服务时的猜测。

24. 运行 Spring Boot 有哪几种方式?

- 1、打包用命令或者放到容器中运行
- 2、用 Maven/ Gradle 插件运行
- 3、直接执行 main 方法运行

25. Spring Boot 打成的 jar 和普通的 jar 有什么区别 ?

Spring Boot 项目最终打包成的 jar 是可执行 jar , 这种 jar 可以直接通过 java -jar xxx.jar 命令来运行 , 这种 jar 不可以作为普通的 jar 被其他项目依赖 , 即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖 , 主要还是他和普通 jar 的结构不同。普通的 jar 包 , 解压后直接就是包名 , 包里就是我们的代码 , 而 Spring Boot 打包成的可执行 jar 解压后 , 在 \BOOT-INF\classes 目录下才是我们的代码 , 因此无法被直接引用。如果非要引用 , 可以在 pom.xml 文件中增加配置 , 将 Spring Boot 项目打包成两个 jar , 一个可执行 , 一个可引用。

26. 如何使用 Spring Boot 实现分页和排序?

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的传递给存储库方法。

27. Spring Boot 中如何实现定时任务 ?

定时任务也是一个常见的需求 , Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式 , 一个就是使用 Spring 中的 @Scheduled 注解 , 另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz , 则按照 Quartz 的方式 , 定义 Job 和 Trigger 即可。

28. Spring Boot 还提供了其它的哪些 Starter Project Options?

Spring Boot 也提供了其它的启动器项目包括 , 包括用于开发特定类型应用程序的典型依赖项。

- spring-boot-starter-web-services - SOAP Web Services
- spring-boot-starter-web - Web 和 RESTful 应用程序
- spring-boot-starter-test - 单元测试和集成测试
- spring-boot-starter-jdbc - 传统的 JDBC
- spring-boot-starter-hateoas - 为服务添加 HATEOAS 功能
- spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权
- spring-boot-starter-data-jpa - 带有 Hibernate 的 Spring Data JPA
- spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务

29. 为什么我们需要 spring-boot-maven-plugin?

spring-boot-maven-plugin 提供了一些像 jar 一样打包或者运行应用程序的命令。

- 1、spring-boot:run 运行你的 SpringBooty 应用程序。
- 2、spring-boot: repackage 重新打包你的 jar 包或者是 war 包使其可执行
- 3、spring-boot: start 和 spring-boot: stop 管理 Spring Boot 应用程序的生命周期（也可以说是为了集成测试）。
- 4、spring-boot:build-info 生成执行器可以使用的构造信息。

30. 什么是YAML?

YAML是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML文件就更加结构化，而且更少混淆。可以看出YAML具有分层配置数据。

31. 什么是JavaConfig?

Spring JavaConfig是Spring社区的产品，它提供了配置Spring IoC容器的纯Java方法。因此它有助于避免使用XML配置。使用JavaConfig的优点在于：

面向对象的配置。由于配置被定义为JavaConfig中的类，因此用户可以充分利用Java中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean方法等。

减少或消除XML配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在XML和Java之间来回切换。

JavaConfig为开发人员提供了一种纯Java方法来配置与XML配置概念相似的Spring容器。

从技术角度来讲，只使用JavaConfig配置类来配置容器是可行的，但实际上很多人认为将JavaConfig与XML混合匹配是理想的。

类型安全和重构友好。JavaConfig提供了一种类型安全的方法来配置Spring容器。由于Java 5.0对泛型的支持，现在可以按类型而不是按名称检索bean，不需要任何强制转换或基于字符串的查找。Spring JavaConfig是Spring社区的产品，它提供了配置Spring IoC容器的纯Java方法。因此它有助于避免使用XML配置。使用JavaConfig的优点在于：

面向对象的配置。由于配置被定义为JavaConfig中的类，因此用户可以充分利用Java中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean方法等。

减少或消除XML配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在XML和Java之间来回切换。

JavaConfig为开发人员提供了一种纯Java方法来配置与XML配置概念相似的Spring容器。

从技术角度来讲，只使用JavaConfig配置类来配置容器是可行的，但实际上很多人认为将JavaConfig与XML混合匹配是理想的。

类型安全和重构友好。JavaConfig提供了一种类型安全的方法来配置Spring容器。由于Java 5.0对泛型的支持，现在可以按类型而不是按名称检索bean，不需要任何强制转换或基于字符串的查找。

32. Spring Boot、Spring MVC 和 Spring 有什么区别？

1、Spring

Spring最重要的特征是依赖注入。所有 SpringModules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

2、Spring MVC

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServlet, MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变得非常简单。

3、SpringBoot

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/" />
```

https://blog.csdn.net/Dome_...

Spring Boot 通过一个自动配置和启动的项来解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

33. Springboot自动配置的原理

只要使用了 @EnableAutoConfiguration 注解就能实现自动配置。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

@Import(AutoConfigurationImportSelector.class) 这个是自动配置的关键，它完成了自动配置的主要逻辑。下面是代码的主要片段

```
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    // 获取配置的元数据
```

```
AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader
    .loadMetadata(this.beanClassLoader);
// 这个方法包含了加载的主要逻辑，它能找到所有自动注入的类
AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(
    autoConfigurationMetadata, annotationMetadata);
return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
```

下面来看 `getAutoConfigurationEntry` 的主要逻辑

```
protected AutoConfigurationEntry getAutoConfigurationEntry(
    AutoConfigurationMetadata autoConfigurationMetadata,
    AnnotationMetadata annotationMetadata) {
    // ....
    // 获取候选配置类
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
        attributes);
    // ... 过滤、去重、排除一些配置类
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

`SpringFactoriesLoader` 是spring提供的一个扩展机制，它能加载模块下的 `META-INF/spring.factories` 文件，这个Properties格式的文件中的key是接口、注解、或抽象类的全名，value是以逗号","分隔的实现类。`SpringFactoriesLoader`能将相应的实现类注入Spring容器中。

查看一下`spring-boot-autoconfigure`模块下的 `META-INF/spring.factories` 文件

```
...
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfig
uration,\ 
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ 
...
```

就可以看出来它加载哪些自动配置类；通过这个机制我们可以自己写自动配置类并且在模块下的 `META-INF/spring.factories` 文件中写入

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.yuan.interview.springboot.XXXAutoConfiguration
```

这样直接引入我们的模块就会加载 `org.yuan.interview.springboot.XXXAutoConfiguration` 配置类了。

34. 如何禁用一个特定自动配置类？

1、使用 `@EnableAutoConfiguration` 的 `exclude` 属性。

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

2、如果类不在类路径上，可以使用 `@EnableAutoConfiguration` 的 `excludeName` 属性

```
@EnableAutoConfiguration(excludeName={Foo.class})
```

3、可以使用配置 `spring.autoconfigure.exclude` 属性来控制要排除的自动配置类列表。在 `application.properties` 配置文件中配置属性，并且可以使用逗号分隔添加多个类。

35. Spring Boot中的监视器是什么？

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。

有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

36. 什么是 Spring Batch？

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要；包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

37. Spring Boot 中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过（CORS，Cross-origin resource sharing）来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS，现在可以通过实现 `WebMvcConfigurer` 接口然后重写 `addCorsMappings` 方法解决跨域问题。

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("*")
```

```
        .allowCredentials(true)
        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
        .maxAge(3600);
    }

}
```

项目中前后端分离部署，所以需要解决跨域的问题。

我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。

当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。

我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedOrigin("*");
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.setAllowCredentials(true);
        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new
UrlBasedCorsConfigurationSource();
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
corsConfiguration);
        return new CorsFilter(urlBasedCorsConfigurationSource);
    }

}
```

38. 什么是 CSRF 攻击？

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

39. 我们如何监视所有 Spring Boot 微服务？

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我们将使用位于的开源项目。它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

40. 什么是嵌入式服务器？我们为什么要使用嵌入式服务器呢？

思考一下在你的虚拟机上部署应用程序需要些什么。

- 1、安装 Java
- 2、安装 Web 或者是应用程序的服务器（Tomcat/WebSphere/WebLogic 等等）
- 3、部署应用程序 war 包

如果我们想简化这些步骤，应该如何做呢？

让我们来思考如何使服务器成为应用程序的一部分？

你只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了，

这个想法是嵌入式服务器的起源。

当我们创建一个可以部署的应用程序的时候，我们将会把服务器（例如，tomcat）嵌入到可部署的服务器中。

例如，对于一个 Spring Boot 应用程序来说，你可以生成一个包含 Embedded Tomcat 的应用程序 jar。你就可以像运行正常 Java 应用程序一样来运行 web 应用程序了。

嵌入式服务器就是我们的可执行单元包含服务器的二进制文件（例如，tomcat.jar）。

41. 当 Spring Boot 应用程序作为 Java 应用程序运行时，后台会发 生什么？

如果你使用 Eclipse IDE，Eclipse maven 插件确保依赖项或者类文件的改变一经添加，就会被编译并在目标文件中准备好！在这之后，就和其它的 Java 应用程序一样了。

当你启动 java 应用程序的时候，spring boot 自动配置文件就会魔法般的启用了。

当 Spring Boot 应用程序检测到你正在开发一个 web 应用程序的时候，它就会启动 tomcat。

42. RequestMapping 和 GetMapping 的不同之处在哪里？

RequestMapping 具有类属性的，可以进行 GET,POST,PUT 或者其它的注释中具有的请求方法。
GetMapping 是 GET 请求方法中的一个特例。它只是 RequestMapping 的一个延伸，目的是为了提高清晰度。

43. 为什么我们不建议在实际的应用程序中使用 Spring Data Rest？

我们认为 Spring Data Rest 很适合快速原型制造！在大型应用程序中使用需要谨慎。

通过 Spring Data REST 你可以把你的数据实体作为 RESTful 服务直接发布。

当你设计 RESTful 服务的时候，最佳实践表明，你的接口应该考虑到两件重要的事情：

你的模型范围。

你的客户。

通过 With Spring Data REST，你不需要再考虑这两个方面，只需要作为 TEST 服务发布实体。

这就是为什么我们建议使用 Spring Data Rest 在快速原型构造上面，或者作为项目的初始解决方法。对于完整演变项目来说，这并不是一个好的注意。

Spring & SpringBoot常用注解

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. @SpringBootApplication

这里先单独拎出 `@SpringBootApplication` 注解说一下，虽然我们一般不会主动去使用它。

这个注解是 Spring Boot 项目的基石，创建 SpringBoot 项目之后会默认在主类加上。

```
@SpringBootApplication
public class SpringSecurityJwtGuideApplication {
    public static void main(java.lang.String[] args) {
        SpringApplication.run(SpringSecurityJwtGuideApplication.class, args);
    }
}Copy to clipboardErrorCopied
```

我们可以把 `@SpringBootApplication` 看作是 `@Configuration`、`@EnableAutoConfiguration`、`@ComponentScan` 注解的集合。

```
package org.springframework.boot.autoconfigure;
@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) })
public @interface SpringBootApplication {
    .....
}

package org.springframework.boot;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}Copy to clipboardErrorCopied
```

根据 SpringBoot 官网，这三个注解的作用分别是：

- `@EnableAutoConfiguration`：启用 SpringBoot 的自动配置机制
- `@ComponentScan`：扫描被 `@Component` (`@Service`, `@Controller`) 注解的 bean，注解默认会扫描该类所在的包下所有的类。
- `@Configuration`：允许在 Spring 上下文中注册额外的 bean 或导入其他配置类

2. Spring Bean 相关

2.1. @Autowired

自动导入对象到类中，被注入进的类同样要被 Spring 容器管理比如：Service 类注入到 Controller 类中。

```
@Service  
public class UserService {  
    ....  
}  
  
@RestController  
@RequestMapping("/users")  
public class UserController {  
    @Autowired  
    private UserService userService;  
    ....  
}
```

2.2. @Component, @Repository, @Service, @Controller

我们一般使用 `@Autowired` 注解让 Spring 容器帮我们自动装配 bean。要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类，可以采用以下注解实现：

- `@Component`：通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪个层，可以使用 `@Component` 注解标注。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Controller`：对应 Spring MVC 控制层，主要用于接受用户请求并调用 Service 层返回数据给前端页面。

2.3. @RestController

`@RestController` 注解是 `@Controller` 和 `@ResponseBody` 的合集，表示这是个控制器 bean，并且是将函数的返回值直接填入 HTTP 响应体中，是 REST 风格的控制器。

现在都是前后端分离，说实话我已经很久没有用过 `@Controller`。如果你的项目太老了的话，就当我没说。

单独使用 `@Controller` 不加 `@ResponseBody` 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的 Spring MVC 的应用，对应于前后端不分离的情况。`@Controller + @ResponseBody` 返回 JSON 或 XML 形式数据

2.4. @Scope

声明 Spring Bean 的作用域，使用方法：

```
@Bean  
@Scope("singleton")  
public Person personSingleton() {  
    return new Person();  
}Copy to clipboardErrorCopied
```

四种常见的 Spring Bean 的作用域：

- singleton：唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype：每次请求都会创建一个新的 bean 实例。
- request：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。
- session：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

2.5. @Configuration

一般用来声明配置类，可以使用 @Component 注解替代，不过使用 @Configuration 注解声明配置类更加语义化。

```
@Configuration  
public class AppConfig {  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}Copy to clipboardErrorCopied
```

3. 处理常见的 HTTP 请求类型

5 种常见的请求类型：

- **GET**：请求从服务器获取特定资源。举个例子：`GET /users`（获取所有学生）
- **POST**：在服务器上创建一个新的资源。举个例子：`POST /users`（创建学生）
- **PUT**：更新服务器上的资源（客户端提供更新后的整个资源）。举个例子：`PUT /users/12`（更新编号为 12 的学生）
- **DELETE**：从服务器删除特定的资源。举个例子：`DELETE /users/12`（删除编号为 12 的学生）
- **PATCH**：更新服务器上的资源（客户端提供更改的属性，可以看做是部分更新），使用的比较少，这里就不举例子了。

3.1. GET 请求

```
@GetMapping("users")` 等价于  
`@RequestMapping(value="/users",method=RequestMethod.GET)  
@GetMapping("/users")  
public ResponseEntity<List<User>> getAllUsers() {  
    return userRepository.findAll();  
}Copy to clipboardErrorCopied
```

3.2. POST 请求

```
@PostMapping("users")` 等价于  
`@RequestMapping(value="/users",method=RequestMethod.POST)
```

关于 @RequestBody 注解的使用，在下面的“前端传值”这块会讲到。

```
@PostMapping("/users")  
public ResponseEntity<User> createUser(@Valid @RequestBody UserCreateRequest  
userCreateRequest) {  
    return userRespository.save(user);  
}Copy to clipboardErrorCopied
```

3.3. PUT 请求

```
@PutMapping("/users/{userId}")` 等价于  
`@RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)  
@PutMapping("/users/{userId}")  
public ResponseEntity<User> updateUser(@PathVariable(value = "userId") Long  
userId,  
    @Valid @RequestBody UserUpdateRequest userUpdateRequest) {  
    ....  
}Copy to clipboardErrorCopied
```

3.4. DELETE 请求

```
@DeleteMapping("/users/{userId}")` 等价于  
`@RequestMapping(value="/users/{userId}",method=RequestMethod.DELETE)  
@DeleteMapping("/users/{userId}")  
public ResponseEntity deleteUser(@PathVariable(value = "userId") Long userId){  
    ....  
}Copy to clipboardErrorCopied
```

3.5. PATCH 请求

一般实际项目中，我们都是 PUT 不够用了之后才用 PATCH 请求去更新数据。

```
@PatchMapping("/profile")  
public ResponseEntity updateStudent(@RequestBody StudentUpdateRequest  
studentUpdateRequest) {  
    studentRepository.updateDetail(studentUpdateRequest);  
    return ResponseEntity.ok().build();  
}Copy to clipboardErrorCopied
```

4. 前后端传值

掌握前后端传值的正确姿势，是你开始 CRUD 的第一步！

4.1. @PathVariable 和 @RequestParam

@PathVariable 用于获取路径参数， @RequestParam 用于获取查询参数。

举个简单的例子：

```
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getClassRelatedTeachers(
    @PathVariable("klassId") Long klassId,
    @RequestParam(value = "type", required = false) String type) {
    ...
}Copy to clipboardErrorCopied
```

如果我们请求的 url 是： /klasses/{123456}/teachers?type=web

那么我们服务获取到的数据就是： klassId=123456, type=web。

4.2. @RequestBody

用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且 Content-Type 为 application/json 格式的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用 HttpMessageConverter 或者自定义的 HttpMessageConverter 将请求的 body 中的 json 字符串转换为 java 对象。

我用一个简单的例子来给演示一下基本使用！

我们有一个注册的接口：

```
@PostMapping("/sign-up")
public ResponseEntity signUp(@RequestBody @Valid UserRegisterRequest
userRegisterRequest) {
    userService.save(userRegisterRequest);
    return ResponseEntity.ok().build();
}Copy to clipboardErrorCopied
```

UserRegisterRequest 对象：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserRegisterRequest {
    @NotBlank
    private String userName;
    @NotBlank
    private String password;
    @NotBlank
    private String fullName;
}Copy to clipboardErrorCopied
```

我们发送 post 请求到这个接口，并且 body 携带 JSON 数据：

```
{"userName": "coder", "fullName": "shuangkou", "password": "123456"}Copy to
clipboardErrorCopied
```

这样我们的后端就可以直接把 json 格式的数据映射到我们的 UserRegisterRequest 类上。



POST ▼ http://localhost:9333/api/users/sign-up Send

JSON ▼ Auth ▼ Query Header 2 Docs

```
1 {"userNmae": "coder", "fullNmae": "shuangkou", "password": "123456"}
```

需要注意的是：一个请求方法只可以有一个 `@RequestBody`，但是可以有多个 `@RequestParam` 和 `@PathVariable`。如果你的方法必须要用两个 `@RequestBody` 来接受数据的话，大概率是你的数据库设计或者系统设计出问题了！

5. 读取配置信息

很多时候我们需要将一些常用的配置信息比如阿里云 oss、发送短信、微信认证的相关配置信息等等放到配置文件中。

下面我们来看一下 Spring 为我们提供了哪些方式帮助我们从配置文件中读取这些配置信息。

我们的数据源 `application.yml` 内容如下：

```
wuhan2020: 2020年初武汉爆发了新型冠状病毒，疫情严重，但是，我相信一切都会过去！武汉加油！中国加油！

my-profile:
  name: Guide哥
  email: koushuangbwcx@163.com

library:
  location: 湖北武汉加油中国加油
  books:
    - name: 天才基本法
      description: 二十二岁的林朝夕在父亲确诊阿尔茨海默病这天，得知自己暗恋多年的校园男神裴之即将出国深造的消息——对方考取的学校，恰是父亲当年为她放弃的那所。
    - name: 时间的秩序
      description: 为什么我们记得过去，而非未来？时间“流逝”意味着什么？是我们存在于时间之内，还是时间存在于我们之中？卡洛·罗韦利用诗意的文字，邀请我们思考这一亘古难题——时间的本质。
    - name: 了不起的我
      description: 如何养成一个新习惯？如何让心智变得更成熟？如何拥有高质量的关系？如何走出人生的艰难时刻？Copy to clipboardErrorCopied
```

5.1. @value(常用)

使用 `@value("${property}")` 读取比较简单的配置信息：

```
@value("${wuhan2020}")
String wuhan2020;Copy to clipboardErrorCopied
```

5.2. @ConfigurationProperties(常用)

通过 `@ConfigurationProperties` 读取配置信息并与 bean 绑定。

```
@Component
@ConfigurationProperties(prefix = "library")
class LibraryProperties {
    @NotEmpty
    private String location;
    private List<Book> books;

    @Setter
    @Getter
    @ToString
    static class Book {
        String name;
        String description;
    }
省略getter/setter
.....
}Copy to clipboardErrorCopied
```

你可以像使用普通的 Spring bean 一样，将其注入到类中使用。

5.3. PropertySource (不常用)

`@PropertySource` 读取指定 properties 文件

```
@Component
@PropertySource("classpath:website.properties")

class Website {
    @Value("${url}")
    private String url;

省略getter/setter
.....
}Copy to clipboardErrorCopied
```

6. 参数校验

数据的校验的重要性就不用说了，即使在前端对数据进行校验的情况下，我们还是要对传入后端的数据再进行一遍校验，避免用户绕过浏览器直接通过一些 HTTP 工具直接向后端请求一些违法数据。

JSR(Java Specification Requests) 是一套 JavaBean 参数校验的标准，它定义了很多常用的校验注解，我们可以直接将这些注解加在我们 JavaBean 的属性上面，这样就可以在需要校验的时候进行校验了，非常方便！

校验的时候我们实际用的是 **Hibernate Validator** 框架。Hibernate Validator 是 Hibernate 团队最初的数据校验框架，Hibernate Validator 4.x 是 Bean Validation 1.0 (JSR 303) 的参考实现，Hibernate Validator 5.x 是 Bean Validation 1.1 (JSR 349) 的参考实现，目前最新版的 Hibernate Validator 6.x 是 Bean Validation 2.0 (JSR 380) 的参考实现。

SpringBoot 项目的 spring-boot-starter-web 依赖中已经有 hibernate-validator 包，不需要引用相关依赖。如下图所示（通过 idea 插件—Maven Helper 生成）：

```
▼ spring-boot-starter-web : 2.1.8.RELEASE [compile]
  ▼ hibernate-validator : 6.0.17.Final [compile]
    classmate : 1.4.0 [compile]
    jboss-logging : 3.3.3.Final [compile]
    validation-api : 2.0.1.Final [compile]
    spring-boot-starter : 2.1.8.RELEASE [compile]
  ▼ spring-boot-starter-json : 2.1.8.RELEASE [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-datatype-jdk8 : 2.9.9 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-datatype-jsr310 : 2.9.9 [compile]
    jackson-annotations : 2.9.0 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
  ▼ jackson-module-parameter-names : 2.9.9 [compile]
    jackson-core : 2.9.9 [compile]
    jackson-databind : 2.9.9.3 [compile]
    spring-boot-starter : 2.1.8.RELEASE [compile]
    spring-web : 5.1.9.RELEASE [compile]
  ▼ spring-boot-starter-tomcat : 2.1.8.RELEASE [compile]
    javax.annotation-api : 1.3.2 [compile]
    tomcat-embed-core : 9.0.24 [compile]
```

非 SpringBoot 项目需要自行引入相关依赖包

需要注意的是：所有的注解，推荐使用 JSR 注解，即 `javax.validation.constraints`，而不是 `org.hibernate.validator.constraints`

6.1. 一些常用的字段验证的注解

- `@NotEmpty` 被注释的字符串的不能为 null 也不能为空
- `@NotBlank` 被注释的字符串非 null，并且必须包含一个非空白字符
- `@Null` 被注释的元素必须为 null
- `@NotNull` 被注释的元素必须不为 null
- `@AssertTrue` 被注释的元素必须为 true
- `@AssertFalse` 被注释的元素必须为 false
- `@Pattern(regex=, flag=)` 被注释的元素必须符合指定的正则表达式
- `@Email` 被注释的元素必须是 Email 格式。
- `@Min(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@Max(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@DecimalMin(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@Size(max=, min=)` 被注释的元素的大小必须在指定的范围内
- `@Digits(integer, fraction)` 被注释的元素必须是一个数字，其值必须在可接受的范围内
- `@Past` 被注释的元素必须是一个过去的日期
- `@Future` 被注释的元素必须是一个将来的日期
-

6.2. 验证请求体(RequestBody)

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Person {  
  
    @NotNull(message = "classId 不能为空")  
    private String classId;  
  
    @Size(max = 33)  
    @NotNull(message = "name 不能为空")  
    private String name;  
  
    @Pattern(regexp = "((^Man$|^Woman$|^UGM$))", message = "sex 值不在可选范围")  
    @NotNull(message = "sex 不能为空")  
    private String sex;  
  
    @Email(message = "email 格式不正确")  
    @NotNull(message = "email 不能为空")  
    private String email;  
  
}Copy to clipboardErrorCopied
```

我们在需要验证的参数上加上了 `@valid` 注解，如果验证失败，它将抛出 `MethodArgumentNotValidException`。

```
@RestController  
@RequestMapping("/api")  
public class PersonController {  
  
    @PostMapping("/person")  
    public ResponseEntity<Person> getPerson(@RequestBody @Valid Person person) {  
        return ResponseEntity.ok().body(person);  
    }  
}Copy to clipboardErrorCopied
```

6.3. 验证请求参数(Path Variables 和 Request Parameters)

一定一定不要忘记在类上加上 `Validated` 注解了，这个参数可以告诉 Spring 去校验方法参数。

```
@RestController  
@RequestMapping("/api")  
@Validated  
public class PersonController {  
  
    @GetMapping("/person/{id}")  
    public ResponseEntity<Integer> getPersonByID(@Valid @PathVariable("id")  
        @Max(value = 5,message = "超过 id 的范围了") Integer id) {  
        return ResponseEntity.ok().body(id);  
    }  
}Copy to clipboardErrorCopied
```

7. 全局处理 Controller 层异常

介绍一下我们 Spring 项目必备的全局处理 Controller 层异常。

相关注解：

1. `@ControllerAdvice` :注解定义全局异常处理类
2. `@ExceptionHandler` :注解声明异常处理方法

如何使用呢？拿我们在第 5 节参数校验这块来举例子。如果方法参数不对的话就会抛出 `MethodArgumentNotValidException`，我们来处理这个异常。

```
@ControllerAdvice
@ResponseBody
public class GlobalExceptionHandler {

    /**
     * 请求参数异常处理
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
    handleMethodArgumentNotValidException(MethodArgumentNotValidException ex,
    HttpServletRequest request) {
        .....
    }
}Copy to clipboardErrorCopied
```

8. JPA 相关

8.1. 创建表

`@Entity` 声明一个类对应一个数据库实体。

`@Table` 设置表名

```
@Entity
@Table(name = "role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    省略getter/setter.....
}Copy to clipboardErrorCopied
```

8.2. 创建主键

`@Id`：声明一个字段为主键。

使用 `@Id` 声明之后，我们还需要定义主键的生成策略。我们可以使用 `@GeneratedValue` 指定主键生成策略。

1.通过 `@GeneratedValue` 直接使用 JPA 内置提供的四种主键生成策略来指定主键生成策略。

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id;Copy to clipboardErrorCopied
```

JPA 使用枚举定义了 4 中常见的主键生成策略，如下：

枚举替代常量的一种用法

```
public enum GenerationType {  
  
    /**  
     * 使用一个特定的数据库表格来保存主键  
     * 持久化引擎通过关系数据库的一张特定的表格来生成主键,  
     */  
    TABLE,  
  
    /**  
     * 在某些数据库中,不支持主键自增长,比如oracle、PostgreSQL其提供了一种叫做"序列  
     *(sequence)"的机制生成主键  
     */  
    SEQUENCE,  
  
    /**  
     * 主键自增长  
     */  
    IDENTITY,  
  
    /**  
     * 把主键生成策略交给持久化引擎(persistence engine),  
     *持久化引擎会根据数据库在以上三种主键生成 策略中选择其中一种  
     */  
    AUTO  
}  
Copy to clipboardErrorCopied  
@GeneratedValue`注解默认使用的策略是`GenerationType.AUTO  
public @interface GeneratedValue {  
  
    GenerationType strategy() default AUTO;  
    String generator() default "";  
}Copy to clipboardErrorCopied
```

一般使用 MySQL 数据库的话，使用 `GenerationType.IDENTITY` 策略比较普遍一点（分布式系统的话需要另外考虑使用分布式 ID）。

2.通过 `@GenericGenerator` 声明一个主键策略，然后 `@GeneratedValue` 使用这个策略

```
@Id  
@GeneratedValue(generator = "IdentityIdGenerator")  
@GenericGenerator(name = "IdentityIdGenerator", strategy = "identity")  
private Long id;Copy to clipboardErrorCopied
```

等价于：

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;Copy to clipboardErrorCopied

```

jpa 提供的主键生成策略有如下几种：

```

public class DefaultIdentifierGeneratorFactory
    implements MutableIdentifierGeneratorFactory, Serializable,
ServiceRegistryAwareService {

    @SuppressWarnings("deprecation")
    public DefaultIdentifierGeneratorFactory() {
        register( "uuid2", UUIDGenerator.class );
        register( "guid", GUIDGenerator.class );           // can be done with
        UUIDGenerator + strategy
        register( "uuid", UUIDHexGenerator.class );        // "deprecated"
for new use
        register( "uuid.hex", UUIDHexGenerator.class );    // uuid.hex is
deprecated
        register( "assigned", Assigned.class );
        register( "identity", IdentityGenerator.class );
        register( "select", SelectGenerator.class );
        register( "sequence", SequenceStyleGenerator.class );
        register( "seqhilo", SequenceHiLoGenerator.class );
        register( "increment", IncrementGenerator.class );
        register( "foreign", ForeignGenerator.class );
        register( "sequence-identity", SequenceIdentityGenerator.class );
        register( "enhanced-sequence", SequenceStyleGenerator.class );
        register( "enhanced-table", TableGenerator.class );
    }

    public void register(String strategy, Class generatorClass) {
        LOG.debugf( "Registering IdentifierGenerator strategy [%s] -> [%s]",
strategy, generatorClass.getName() );
        final Class previous = generatorStrategyToClassNameMap.put( strategy,
generatorClass );
        if ( previous != null ) {
            LOG.debugf( "      - overriding [%s]", previous.getName() );
        }
    }

}Copy to clipboardErrorCopied

```

8.3. 设置字段类型

`@Column` 声明字段。

示例：

设置属性 `userName` 对应的数据库字段名为 `user_name`, 长度为 32, 非空

```

@Column(name = "user_name", nullable = false, length=32)
private String userName;Copy to clipboardErrorCopied

```

设置字段类型并且加默认值，这个还是挺常用的。

```
column(columnDefinition = "tinyint(1) default 1")
private Boolean enabled;Copy to clipboardErrorCopied
```

8.4. 指定不持久化特定字段

如果我们想让 `secrect` 这个字段不被持久化，可以使用 `@Transient` 关键字声明。

```
Entity(name="USER")
public class User {

    .....
    @Transient
    private String secrect; // not persistent because of @Transient

}Copy to clipboardErrorCopied
```

除了 `@Transient` 关键字声明，还可以采用下面几种方法：

```
static String secrect; // not persistent because of static
final String secrect = "Satish"; // not persistent because of final
transient String secrect; // not persistent because of transientCopy to
clipboardErrorCopied
```

一般使用注解的方式比较多。

8.5. 声明大字段

`@Lob` : 声明某个字段为大字段。

```
@Lob
private String content;Copy to clipboardErrorCopied
```

更详细的声明：

```
@Lob
//指定 Lob 类型数据的获取策略， FetchType.EAGER 表示非延迟 加载，而 FetchType.LAZY 表示
延迟加载 ：
@Basic(fetch = FetchType.EAGER)
//columnDefinition 属性指定数据表对应的 Lob 字段类型
@Column(name = "content", columnDefinition = "LONGTEXT NOT NULL")
private String content;Copy to clipboardErrorCopied
```

8.6. 创建枚举类型的字段

可以使用枚举类型的字段，不过枚举字段要用 `@Enumerated` 注解修饰。

```
public enum Gender {
    MALE("男性"),
    FEMALE("女性");

    private String value;
    Gender(String str){
        value=str;
    }
}
```

```
}Copy to clipboardErrorCopied
@Entity
@Table(name = "role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    @Enumerated(EnumType.STRING)
    private Gender gender;
    省略getter/setter.....
}Copy to clipboardErrorCopied
```

数据库里面对应存储的是 MAIL/FEMAIL。

8.7. 增加审计功能

只要继承了 `AbstractAuditBase` 的类都会默认加上下面四个字段。

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@MappedSuperclass
@EntityListeners(value = AuditingEntityListener.class)
public abstract class AbstractAuditBase {

    @CreatedDate
    @Column(updatable = false)
    @JsonIgnore
    private Instant createdAt;

    @LastModifiedDate
    @JsonIgnore
    private Instant updatedAt;

    @CreatedBy
    @Column(updatable = false)
    @JsonIgnore
    private String createdBy;

    @LastModifiedBy
    @JsonIgnore
    private String updatedBy;
}
Copy to clipboardErrorCopied
```

我们对应的审计功能对应地配置类可能是下面这样的（Spring Security 项目）：

```

@Configuration
@EnableJpaAuditing
public class AuditSecurityConfiguration {
    @Bean
    AuditorAware<String> auditorAware() {
        return () -> Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getName);
    }
}Copy to clipboardErrorCopied

```

简单介绍一下上面设计到的一些注解：

1. `@CreatedDate` : 表示该字段为创建时间时间字段，在这个实体被 insert 的时候，会设置值
2. `@CreatedBy` : 表示该字段为创建人，在这个实体被 insert 的时候，会设置值
`@LastModifiedDate`、`@LastModifiedBy` 同理。

`@EnableJpaAuditing`：开启 JPA 审计功能。

8.8. 删除/修改数据

`@Modifying` 注解提示 JPA 该操作是修改操作,注意还要配合 `@Transactional` 注解使用。

```

@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    @Modifying
    @Transactional(rollbackFor = Exception.class)
    void deleteByUserName(String userName);
}Copy to clipboardErrorCopied

```

8.9. 关联关系

- `@OneToOne` 声明一对关系
- `@OneToMany` 声明一对多关系
- `@ManyToOne` 声明多对一关系
- `MangToMany` 声明多对多关系

9. 事务 `@Transactional`

在要开启事务的方法上使用 `@Transactional` 注解即可!

```

@Transactional(rollbackFor = Exception.class)
public void save() {
    .....
}
Copy to clipboardErrorCopied

```

我们知道 `Exception` 分为运行时异常 `RuntimeException` 和非运行时异常。在 `@Transactional` 注解中如果不配置 `rollbackFor` 属性,那么事物只会在遇到 `RuntimeException` 的时候才会回滚,加上 `rollbackFor=Exception.class`,可以让事物在遇到非运行时异常时也回滚。

`@Transactional` 注解一般用在可以作用在类或者方法上。

- **作用于类**: 当把 `@Transactional` 注解放在类上时, 表示所有该类的 public 方法都配置相同的事务属性信息。
- **作用于方法**: 当类配置了 `@Transactional`, 方法也配置了 `@Transactional`, 方法的事务会覆盖类的事务配置信息。

10. json 数据处理

10.1. 过滤 json 数据

`@JsonIgnoreProperties` 作用在类上用于过滤掉特定字段不返回或者不解析。

```
//生成json时将userRoles属性过滤
@JsonIgnoreProperties({"userRoles"})
public class User {

    private String userName;
    private String fullName;
    private String password;
    @JsonIgnore
    private List<UserRole> userRoles = new ArrayList<>();
}
```

`@JsonIgnore` 一般用于类的属性上, 作用和上面的 `@JsonIgnoreProperties` 一样。

```
public class User {

    private String userName;
    private String fullName;
    private String password;
    //生成json时将userRoles属性过滤
    @JsonIgnore
    private List<UserRole> userRoles = new ArrayList<>();
}
```

10.2. 格式化 json 数据

`@JsonFormat` 一般用来格式化 json 数据。:

比如:

```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'", timezone="GMT")
private Date date;
```

10.3. 扁平化对象

```
@Getter
@Setter
@Override
public class Account {
    @JsonUnwrapped
```

```
private Location location;
@JsonUnwrapped
private PersonInfo personInfo;

@Getter
@Setter
@ToString
public static class Location {
    private String provinceName;
    private String countyName;
}

@Getter
@Setter
@ToString
public static class PersonInfo {
    private String userName;
    private String fullName;
}

Copy to clipboardErrorCopied
```

未扁平化之前：

```
{
    "location": {
        "provinceName": "湖北",
        "countyName": "武汉"
    },
    "personInfo": {
        "userName": "coder1234",
        "fullName": "shaungkou"
    }
}Copy to clipboardErrorCopied
```

使用 @JsonUnwrapped 扁平对象之后：

```
@Getter
@Setter
@ToString
public class Account {
    @JsonUnwrapped
    private Location location;
    @JsonUnwrapped
    private PersonInfo personInfo;
    .....
}Copy to clipboardErrorCopied
{
    "provinceName": "湖北",
    "countyName": "武汉",
    "userName": "coder1234",
    "fullName": "shaungkou"
}Copy to clipboardErrorCopied
```

11. 测试相关

`@ActiveProfiles` 一般作用于测试类上，用于声明生效的 Spring 配置文件。

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
@ActiveProfiles("test")
@Slf4j
public abstract class TestBase {
    ...
}
```

`@Test` 声明一个方法为测试方法

`@Transactional` 被声明的测试方法的数据会回滚，避免污染测试数据。

`@withMockUser` Spring Security 提供的，用来模拟一个真实用户，并且可以赋予权限。

```
@Test
@Transactional
@withMockUser(username = "user-id-18163138155", authorities =
"ROLE_TEACHER")
void should_import_student_success() throws Exception {
    ...
}
```

微服务

SpringCloud面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. 什么是微服务？

单个轻量级服务一般为一个**单独微服务**，微服务讲究的是专注某个功能的实现，比如登录系统只专注于用户登录方面功能的实现，讲究的是**职责单一，开箱即用，可以独立运行**。微服务架构系统是一个**分布式的系统**，按照业务进行划分服务单元模块，解决单个系统的不足，满足越来越**复杂的业务需求**。

马丁福勒（Martin Fowler）：就目前而言，对于微服务业界并没有一个统一的、标准的定义。但通常而言，微服务架构是一种架构模式或者说是架构风格，它提倡将单一应用程序划分成一组小的服务。每个服务运行在其独立的自己的进程中服务之间相互配合、相互协调，为用户提供最终价值。服务之间采用轻量级通信。每个服务都围绕具体业务进行构建，并能够独立部署到生产环境等。另外应尽量避免统一的、集中的服务管理机制。

通俗的来讲：

微信搜索公众号：Java专栏，获取最新面试手册

微服务就是一个独立的职责单一的服务应用程序。在 intelliJ idea 工具里面就是用maven开发的一个个独立的module，具体就是使用springboot 开发的一个小的模块，处理单一专业的业务逻辑，一个模块只做一个事情。

微服务强调的是服务大小，关注的是某一个点，具体解决某一个问题/落地对应的一个服务应用，可以看做是idea 里面一个 module。

比如你去医院：你的牙齿不舒服，那么你就去牙科。你的头疼，那么你就去脑科。一个个的科室，就是一个微服务，一个功能就是一个服务。

2. 什么是微服务架构？

在前面你理解什么是微服务，那么对于微服务架构基本上就已经理解了。

微服务架构 就是 对微服务进行管理整合应用的。微服务架构 依赖于 微服务，是在微服务基础之上的。

例如：上面已经列举了什么是微服务。在医院里，每一个科室都是一个独立的微服务，那么 这个医院 就是一个大型的微服务架构，就类似 院长 可以 对下面的 科室进行管理。微服务架构主要就是这种功能。

3. 微服务的优缺点是什么？说下你在项目中碰到的坑。

优点：松耦合，聚焦单一业务功能，无关开发语言，团队规模降低。在开发中，不需要了解多有业务，只专注于当前功能，便利集中，功能小而精。微服务一个功能受损，对其他功能影响并不是太大，可以快速定位问题。微服务只专注于当前业务逻辑代码，不会和 html、css 或其他界面进行混合。可以灵活搭配技术，独立性比较舒服。

缺点：随着服务数量增加，管理复杂，部署复杂，服务器需要增多，服务通信和调用压力增大，运维工程师压力增大，人力资源增多，系统依赖增强，数据一致性，性能监控。

4. 你所知道微服务的技术栈有哪些？列举一二。

微服务条目	落地技术
服务开发	SpringBoot、Spring、SpringMVC
服务配置与管理	Netflix公司的Archaius、阿里的Diamond等
服务注册与发现	Eureka、Consul、Zookeeper等
服务调用	Rest（服务通信）、RPC（Dubbo）、GRpc
服务熔断器	Hystrix、Envoy等
负载均衡	Nginx、Ribbon等
服务接口调用（客户端简化工具）	Fegin等
消息队列	Kafka、RabbitMQ、ActiveMQ等
服务配置中心管理	SpringCloudConfig、Chef等
服务路由（API网关）	Zuul等
服务监控	Zabbix, Nagios, Metrics, Spectator等
全链路追踪	Zipkin, Brave, Dapper等
服务部署	Docker, OpenStack, Kubernetes等
数据流操作开发包	SpringCloud Stream（封装与Redis, Rabbit, kafka等发送接收消息）
事件消息总线	Spring Cloud Bus

5. 微服务之间如何独立通讯的？

同步通信：dubbo通过 RPC 远程过程调用、springcloud通过 REST 接口json调用 等。

异步：消息队列，如：RabbitMq、ActiveM、Kafka 等。

6. Spring Cloud的版本关系

Spring Cloud是一个由许多子项目组成的综合项目，各子项目有不同的发布节奏。为了管理Spring Cloud与各子项目的版本依赖关系，发布了一个清单，其中包括了某个Spring Cloud版本对应的子项目版本。

为了避免Spring Cloud版本号与子项目版本号混淆，Spring Cloud版本采用了名称而非版本号的命名，这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，例如Angel是第一个版本，Brixton是第二个版本。

当Spring Cloud的发布内容积累到临界点或者一个重大BUG被解决后，会发布一个"service releases"版本，简称SRX版本，比如Greenwich.SR2就是Spring Cloud发布的Greenwich版本的第2个SRX版本。

Spring Cloud和SpringBoot版本对应关系

Spring Cloud Version	SpringBoot Version
Hoxton	2.2.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

Spring Cloud和各子项目版本对应关系

Component	Edgware.SR6	Greenwich.SR2
spring-cloud-bus	1.3.4.RELEASE	2.1.2.RELEASE
spring-cloud-commons	1.3.6.RELEASE	2.1.2.RELEASE
spring-cloud-config	1.4.7.RELEASE	2.1.3.RELEASE
spring-cloud-netflix	1.4.7.RELEASE	2.1.2.RELEASE
spring-cloud-security	1.2.4.RELEASE	2.1.3.RELEASE
spring-cloud-consul	1.3.6.RELEASE	2.1.2.RELEASE
spring-cloud-sleuth	1.3.6.RELEASE	2.1.1.RELEASE
spring-cloud-stream	Ditmars.SR5	Fishtown.SR3
spring-cloud-zookeeper	1.2.3.RELEASE	2.1.2.RELEASE
spring-boot	1.5.21.RELEASE	2.1.5.RELEASE
spring-cloud-task	1.2.4.RELEASE	2.1.2.RELEASE
spring-cloud-gateway	1.0.3.RELEASE	2.1.2.RELEASE
spring-cloud-openfeign	暂无	2.1.2.RELEASE

注意：Hoxton版本是基于SpringBoot 2.2.x版本构建的，不适用于1.5.x版本。随着2019年8月SpringBoot 1.5.x版本停止维护，Edgware版本也将停止维护。

7. Spring Cloud的子项目（主要项目）

大致可分成两类，

一类是对现有成熟框架"Spring Boot化"的封装和抽象，也是数量最多的项目；

第二类是开发了一部分分布式系统的基础设施的实现，如Spring Cloud Stream扮演的就是kafka, ActiveMQ这样的角色。

Spring Cloud Config

集中配置管理工具，分布式系统中统一的外部配置管理，默认使用Git来存储配置，可以支持客户端配置的刷新及加密、解密操作。

Spring Cloud Netflix

Netflix OSS 开源组件集成，包括Eureka、Hystrix、Ribbon、Feign、Zuul等核心组件。

- Eureka：服务治理组件，包括服务端的注册中心和客户端的服务发现机制；
- Ribbon：负载均衡的服务调用组件，具有多种负载均衡调用策略；

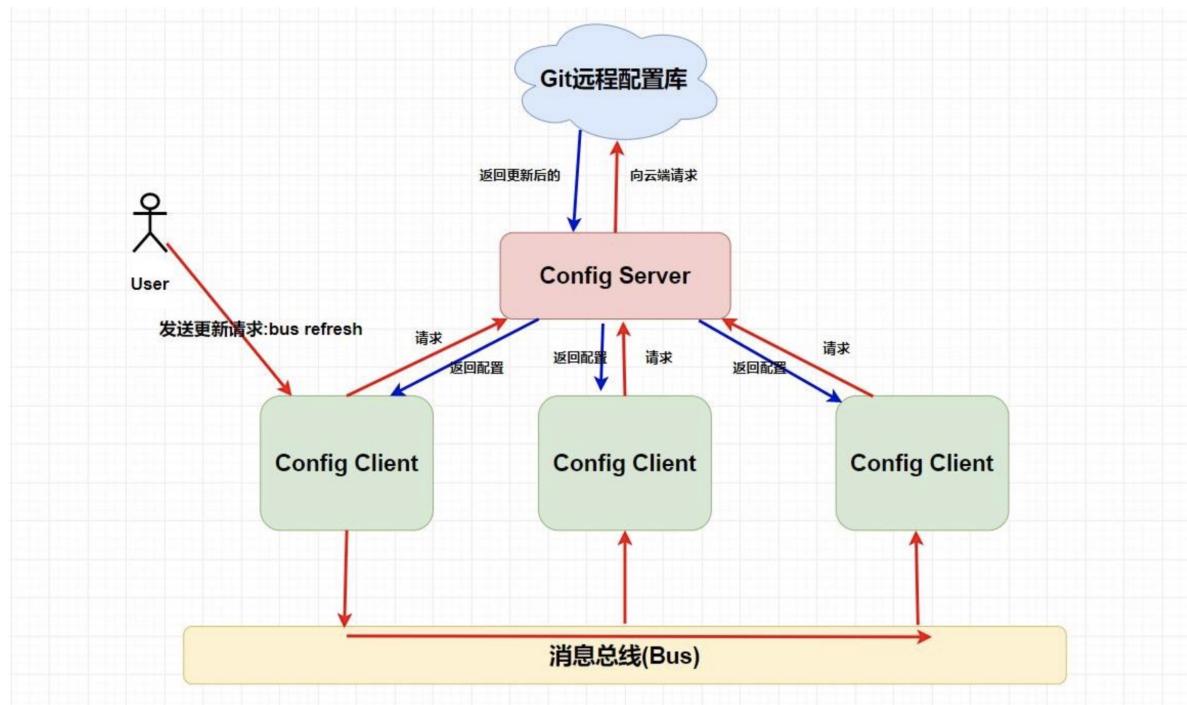
- Hystrix：服务容错组件，实现了断路器模式，为依赖服务的出错和延迟提供了容错能力；
- Feign：基于Ribbon和Hystrix的声明式服务调用组件；
- Zuul：API网关组件，对请求提供路由及过滤功能。

Spring Cloud Bus

用于将服务和服务实例与分布式消息系统链接在一起的事件总线。在集群中传播状态更改很有用（例如配置更改事件）。

你可以简单理解为 Spring Cloud Bus 的作用就是 **管理和广播分布式系统中的消息**，也就是消息引擎系统中的广播模式。当然作为 **消息总线** 的 `spring cloud bus` 可以做很多事而不仅仅是客户端的配置刷新功能。

而拥有了 `spring cloud bus` 之后，我们只需要创建一个简单的请求，并且加上 `@RefreshScope` 注解就能进行配置的动态修改了，下面我画了张图供你理解。



Spring Cloud Consul

基于Hashicorp Consul的服务治理组件。

Spring Cloud Security

安全工具包，对Zuul代理中的负载均衡OAuth2客户端及登录认证进行支持。

Spring Cloud Sleuth

Spring Cloud应用程序的分布式请求链路跟踪，支持使用Zipkin、HTrace和基于日志（例如ELK）的跟踪。

轻量级事件驱动微服务框架，可以使用简单的声明式模型来发送及接收消息，主要实现为Apache Kafka及RabbitMQ。

Spring Cloud Task

用于快速构建短暂、有限数据处理任务的微服务框架，用于向应用中添加功能性和非功能性的特性。

Spring Cloud Zookeeper

基于Apache Zookeeper的服务治理组件。

Spring Cloud Gateway

API网关组件，对请求提供路由及过滤功能。

Spring Cloud OpenFeign

基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC注解的接口实现用于服务调用，在Spring Cloud 2.0中已经取代Feign成为了一等公民。

8. spring cloud 的核心组件有哪些？

- Eureka：服务注册与发现。
- Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求。
- Ribbon：实现负载均衡，从一个服务的多台机器中选择一台。
- Hystrix：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题。
- Zuul：网关管理，由 Zuul 网关转发请求给对应的服务。

9. springcloud如何实现服务的注册？

- 1、服务发布时，指定对应的服务名，将服务注册到注册中心(eureka zookeeper)
- 2、注册中心加@EnableEurekaServer，服务用@EnableDiscoveryClient，然后用ribbon或feign进行服务直接的调用发现。

10. 什么是Eureka

eureka是Netflix开发的服务发现组件，本身是一个基于REST的服务。Spring Cloud将它集成在其子项目spring-cloud-netflix中，以实现Spring Cloud的服务发现功能。eureka现在已经从1.0升级到2.0，可惜的是eureka2.0不在开源，但也不影响我们的使用。由于基于REST服务，自然而然的就能想到，这个服务一定会有心跳检测、健康检查和客户端缓存等机制。

Eureka包括两个端：

- **Eureka Server：注册中心服务端**，用于维护和管理注册服务列表。
- **Eureka Client：注册中心客户端**，向注册中心注册服务的应用都可以叫做Eureka Client（包括Eureka Server本身）。

Eureka Server

- 依赖介绍
 - spring-cloud-starter-netflix-eureka-server eureka服务端的标识，标志着此服务是做为注册中心
- application.properties配置

```
spring.application.name=eureka-server //服务名称
server.port=8000 //服务端口

eureka.client.register-with-eureka=false //自身不做为服务注册到注册中心

eureka.client.fetch-registry=false //从注册表拉取信息

eureka.client.serviceUrl.defaultZone=http://localhost:8000/eureka/ 服务注册地址
```

- 运行服务spring-cloud-eureka
- 运行成功后访问localhost:8000，会显示eureka提供的服务页面

Eureka Client

- 依赖介绍
 - spring-cloud-starter-netflix-eureka-client eureka客户端所需依赖。
 - spring-boot-starter-web web服务所需，内置tomcat服务器。
- application.properties配置

```
spring.application.name=eureka-client-a
server.port=8001

eureka.client.serviceUrl.defaultZone=http://localhost:8000/eureka/
```

- 运行服务spring-cloud-clientA
- 运行成功后访问localhost:8000，会显示eureka提供的服务页面

11. Eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别？

zookeeper 是CP原则，**强一致性和分区容错性**。

eureka 是AP原则 **可用性和分区容错性**。

zookeeper当主节点故障时，zk会在剩余节点重新选择主节点，耗时过长，虽然最终能够恢复，但是**选取主节点期间会导致服务不可用**，这是不能容忍的。

eureka各个节点是平等的，**一个节点挂掉，其他节点仍会正常保证服务**。

12. 你所知道的微服务技术栈？

当Eureka Server 节点在短时间内丢失了过多实例的连接时（比如网络故障或频繁启动关闭客户端）节点会进入自我保护模式，保护注册信息，不再删除注册数据，故障恢复时，自动退出自我保护模式。

13. 使用Spring Cloud有什么优势？

使用Spring Boot开发分布式微服务时，我们面临以下问题

- 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- 冗余-分布式系统中的冗余问题。
- 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- 性能-问题 由于各种运营开销导致的性能问题。
- 部署复杂性-Devops技能的要求。

14. SpringBoot 和 SpringCloud 之间关系？

SpringBoot: 专注于快速方便的开发单个个体微服务（关注微观）；

SpringCloud: 关注全局的微服务协调治理框架，将SpringBoot开发的一个个单体微服务组合并管理起来（关注宏观）；

SpringBoot可以离开SpringCloud独立使用，但是SpringCloud不可以离开SpringBoot，属于依赖关系。

15. SpringCloud 和 Dubbo 有哪些区别？

首先，他们都是**分布式管理框架**。

dubbo 是**二进制传输**，占用带宽会少一点。SpringCloud是**http 传输**，带宽会多一点，同时使用http协议一般会使用**JSON报文**，消耗会更大。

dubbo 开发难度较大，所依赖的 jar 包有很多问题**大型工程无法解决**。SpringCloud 对第三方的继承可以**一键式生成，天然集成**。

SpringCloud 接口协议约定比较松散，**需要强有力的行政措施来限制接口无序升级**。

最大的区别: **Spring Cloud抛弃了Dubbo 的RPC通信，采用的是基于HTTP的REST方式**。

严格来说，这两种方式各有优劣。虽然在一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适。

16. 什么是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。在spring cloud config组件中，分两个角色，一是config server，二是config client。

使用：

- 1、添加pom依赖
- 2、配置文件添加相关配置
- 3、启动类添加注解@EnableConfigServer

17. 什么是Ribbon?

ribbon是一个负载均衡客户端，可以很好的控制http和tcp的一些行为。feign默认集成了ribbon。

18. 什么是feign？它的优点是什么？

- 1、feign采用的是基于接口的注解
- 2、feign整合了ribbon，具有负载均衡的能力
- 3、整合了Hystrix，具有熔断的能力

使用：

- 1、添加pom依赖。
- 2、启动类添加@EnableFeignClients
- 3、定义一个接口@FeignClient(name="xxx")指定调用哪个服务

19. Ribbon和Feign的区别？

- 1、Ribbon都是调用其他服务的，但方式不同。
- 2、启动类注解不同，Ribbon是@RibbonClient feign的是@EnableFeignClients
- 3、服务指定的位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。
- 4、调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发给其他服务，步骤相当繁琐。Feign需要将调用的方法定义成抽象方法即可。

20. 什么是Spring Cloud Gateway?

Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul网关。网关作为流量的，在微服务系统中有着非常作用，网关常见的功能有路由转发、权限校验、限流控制等作用。

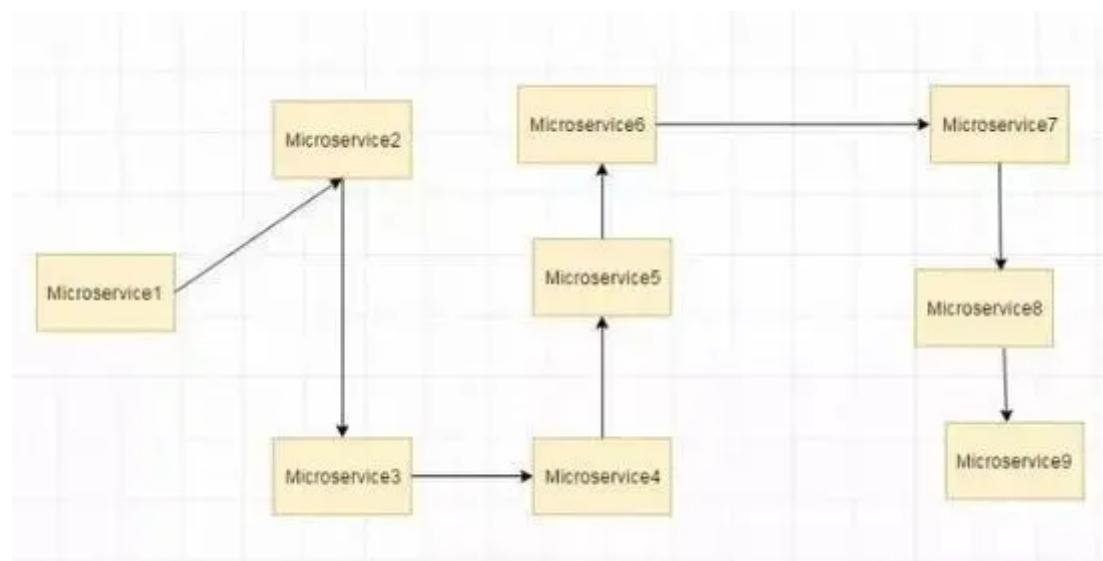
使用了一个RouteLocatorBuilder的bean去创建路由，除了创建路由RouteLocatorBuilder可以让你添加各种predicates和filters，predicates断言的意思，顾名思义就是根据具体的请求的规则，由具体的route去处理，filters是各种过滤器，用来对请求做各种判断和修改。

21. 什么是 Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务



假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方。我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-producer 公开的服务。

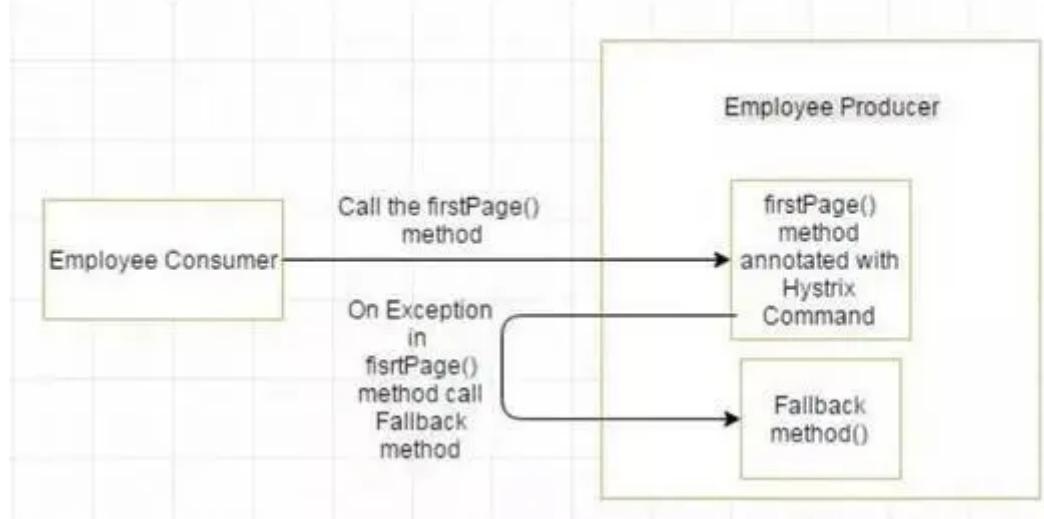
简化图如下所示



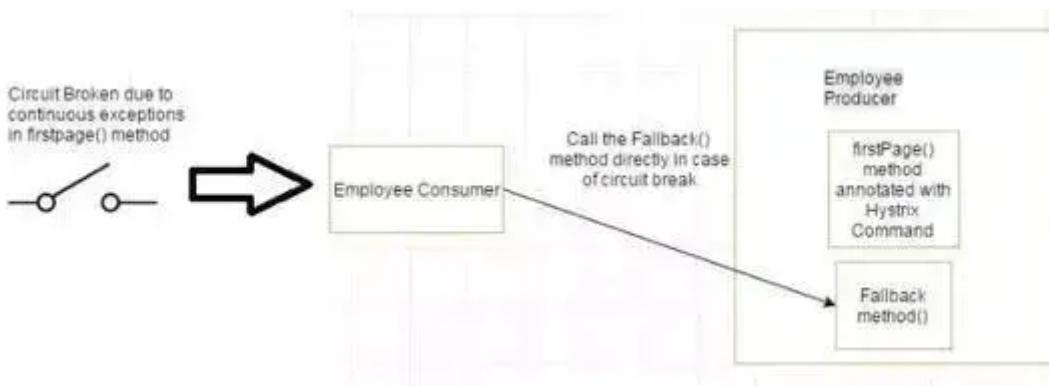
现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值

22. 什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firstPage 方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。



23. 什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit, JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。

Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。

在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

- 1、使用功能区进行负载平衡。
- 2、获取服务实例，然后获取基本 URL。
- 3、利用 REST 模板来使用服务。前面的代码如下

```

@Controller
public class ConsumerControllerClient {
    @Autowired
    private LoadBalancerClient loadBalancer;
    public void getEmployee() throws RestClientException, IOException {
        ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
        System.out.println(serviceInstance.getUri());
        String baseUrl=serviceInstance.getUri().toString();
    }
}

```

```

baseUrl=baseUrl+"/employee";
RestTemplate restTemplate = new RestTemplate();
 ResponseEntity<String> response=null;
try{
    response=restTemplate.exchange(baseUrl,
        HttpMethod.GET, getHeaders(),String.class);
}
catch (Exception ex)
{
    System.out.println(ex);
}
System.out.println(response.getBody());

```

之前的代码，有像 `NullPointerException` 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

24. 什么是Spring Cloud Gateway?

Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul网关。网关作为流量的，在微服务系统中有着非常作用，网关常见的功能有路由转发、权限校验、限流控制等作用。

使用了一个`RouteLocatorBuilder`的bean去创建路由，除了创建路由`RouteLocatorBuilder`可以让你添加各种predicates和filters，predicates断言的意思，顾名思义就是根据具体的请求的规则，由具体的route去处理，filters是各种过滤器，用来对请求做各种判断和修改。

25. 服务注册和发现是什么意思？ Spring Cloud如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka服务注册和发现可以在这种情况下提供帮助。由于所有服务都在Eureka服务器上注册并通过调用Eureka服务器完成查找，因此无需处理服务地点的任何更改和处理。

26. 负载平衡的意义什么？

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

27. 什么是熔断？什么是服务降级？

服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，暂时停止对该服务的调用。

服务降级是从整个系统的负荷情况出发和考虑的，对某些负荷会比较高的情况，为了预防某些功能（业务场景）出现负荷过载或者响应慢的情况，在其内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的fallback（退路）错误处理信息。这样，虽然提供的是一个有损的服务，但却保证了整个系统的稳定性和可用性。

28. 什么是Netflix Feign？它的优点是什么？

Feign是受到Retrofit，JAXRS-2.0和WebSocket启发的java客户端联编程序。Feign的第一个目标是将约束分母的复杂性统一到http apis，而不考虑其稳定性。在employee-consumer的例子中，我们使用了employee-producer使用REST模板公开的REST服务。

但是我们必须编写大量代码才能执行以下步骤

- 使用功能区进行负载平衡。
- 获取服务实例，然后获取基本URL。
- 利用REST模板来使用服务。前面的代码如下

```
@Controller
public class ConsumerControllerClient {

    @Autowired
    private LoadBalancerClient loadBalancer;

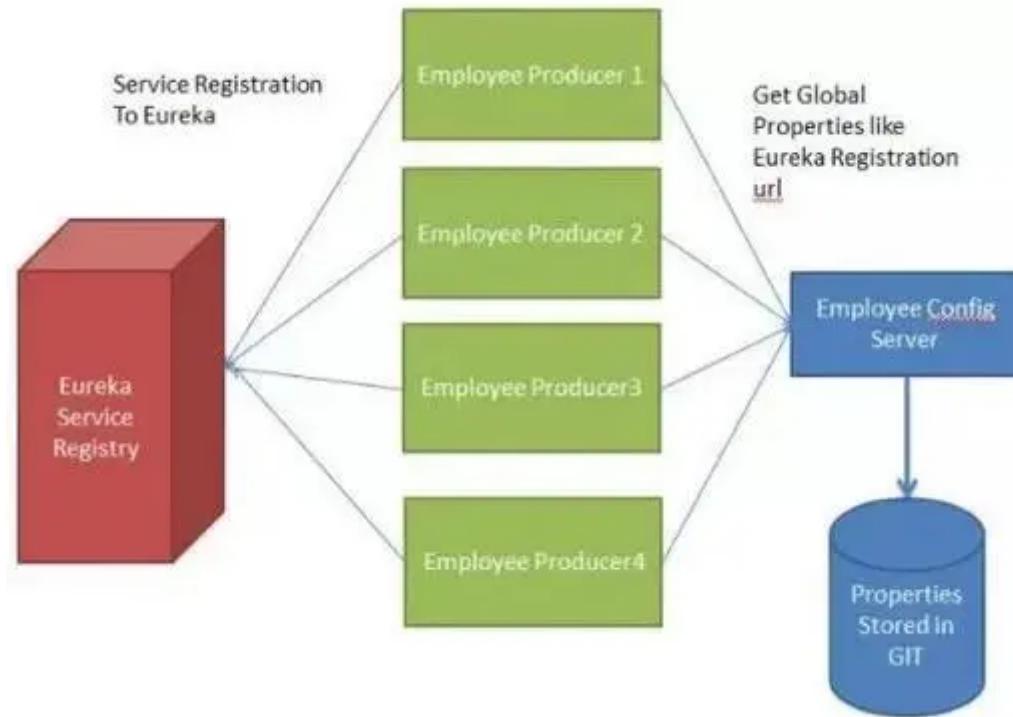
    public void getEmployee() throws RestClientException, IOException {
        ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
        System.out.println(serviceInstance.getUri());
        String baseUrl=serviceInstance.getUri().toString();
        baseUrl=baseUrl+"/employee";
        RestTemplate restTemplate = new RestTemplate();
        ResponseEntity<String> response=null;
        try{
            response=restTemplate.exchange(baseUrl,
                HttpMethod.GET, getHeaders(),String.class);
        }catch (Exception ex)
        {
            System.out.println(ex);
        }
        System.out.println(response.getBody());
    }
}
```

之前的代码，有像NullPointerException这样的例外的机会，并不是最优的。我们将看到如何使用Netflix Feign使呼叫变得更加轻松和清洁。如果Netflix Ribbon依赖关系也在类路径中，那么Feign默认也会负责负载平衡。

29. 什么是 Spring Cloud Bus? 我们需要它吗?

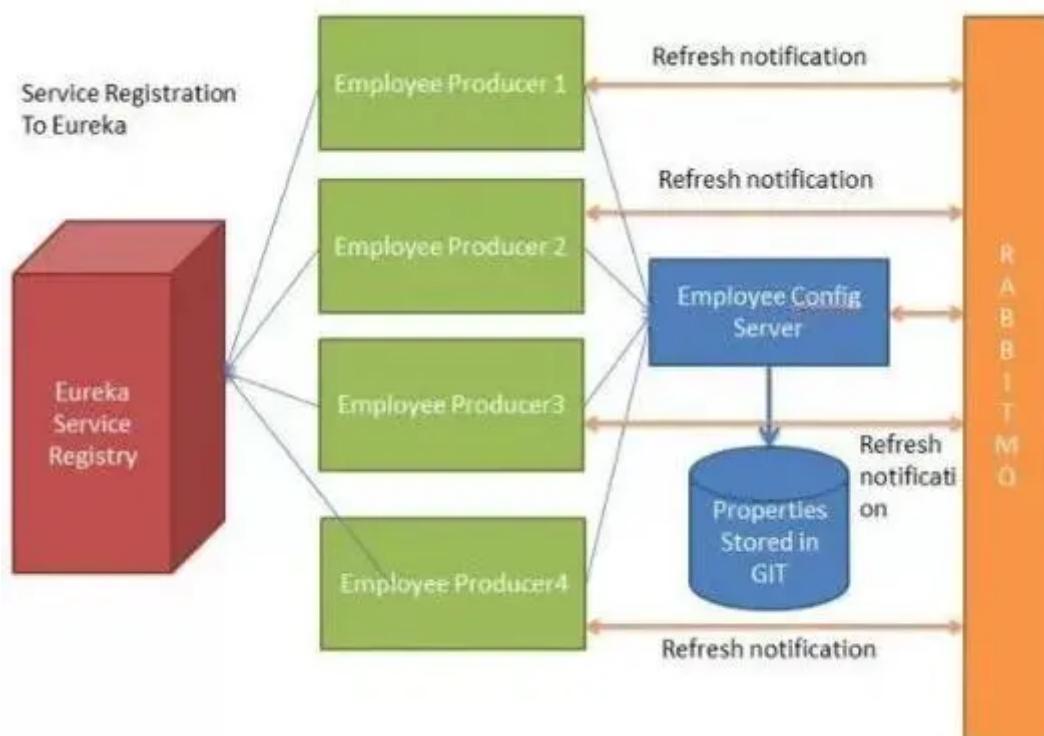
考虑以下情况：我们有多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 `http://localhost:8080/refresh`。同样对于 Employee Producer2 `http://localhost:8081/refresh` 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

30. 微服务是如何对外提供统一接口的(zuul具体使用)

因为每一个微服务都是独立运行的，都有自己独立的IP和端口，而当他们需要统一对外提供服务这时候就需要

SpringCloud网关 zuul网关也是netflix公司旗下的项目

使用它也很简单

在pom依赖中引入 Spring-cloud-starter-netflix-zuul

在SpringBoot启动类中 开启 @EnableZuulProxy

然后在配置文件中定义 路由规则：

routes:

路由名称:

path: /映射路径/**

serviceId: Eureka中的服务名称

zuul也提供了过滤器功能，如果要做一些token检查 或者 过滤时可以使用

用法 就是写一个类 继承 ZuulFilter类

会要求我们实现几个方法

filterType: 过滤器什么时候执行 pre 前置 post 过程中 after 之后

shouldFilter: 过滤器是否执行 可以写判断方法 返回boolean值 true执行， false不执行此过滤器

filterOrder: 过滤器的执行顺序 排序号

run: 具体过滤器的方法

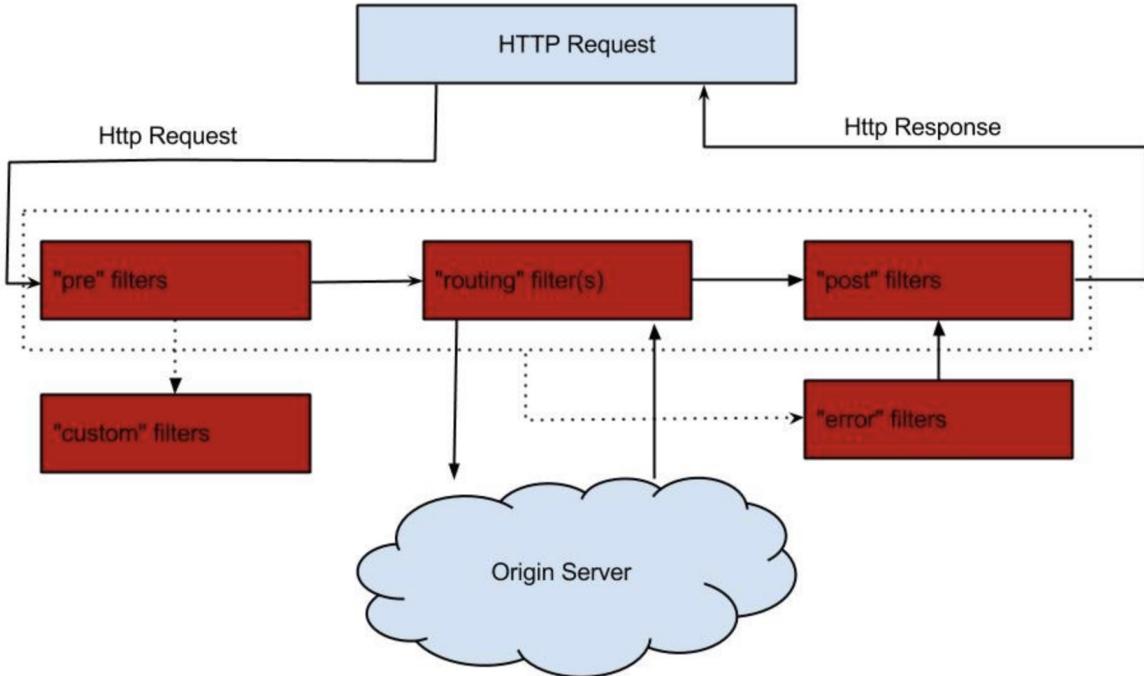
31. Zuul 的过滤功能

如果说，路由功能是 zuul 的基操的话，那么过滤器就是 zuul 的利器了。毕竟所有请求都经过网关 (Zuul)，那么我们可以进行各种过滤，这样我们就能实现 限流，灰度发布，权限控制 等等。

简单实现一个请求时间日志打印

要实现自己定义的 Filter 我们只需要继承 ZuulFilter 然后将这个过滤器类以 @Component 注解加入 Spring 容器中就行了。

在给你们看代码之前我先给你们解释一下关于过滤器的一些注意点。



过滤器类型：`Pre`、`Routing`、`Post`。前置 `Pre` 就是在请求之前进行过滤，`Routing` 路由过滤器就是我们上面所讲的路由策略，而 `Post` 后置过滤器就是在 `Response` 之前进行过滤的过滤器。你可以观察上图结合着理解，并且下面我会给出相应的注释。

```
// 加入Spring容器
@Component
public class PreRequestFilter extends ZuulFilter {
    // 返回过滤器类型 这里是前置过滤器
    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }
    // 指定过滤顺序 越小越先执行，这里第一个执行
    // 当然不是只真正第一个 在Zuul内置中有其他过滤器会先执行
    // 那是写死的 比如 SERVLET_DETECTION_FILTER_ORDER = -3
    @Override
    public int filterOrder() {
        return 0;
    }
    // 什么时候该进行过滤
    // 这里我们可以进行一些判断，这样我们就可以过滤掉一些不符合规定的请求等等
    @Override
    public boolean shouldFilter() {
        return true;
    }
    // 如果过滤器允许通过则怎么进行处理
    @Override
    public Object run() throws ZuulException {
        // 这里我设置了全局的RequestContext并记录了请求开始时间
        RequestContext ctx = RequestContext.getCurrentContext();
        ctx.set("startTime", System.currentTimeMillis());
        return null;
    }
}
```

```

// Lombok的日志
@Slf4j
// 加入 Spring 容器
@Component
public class AccessLogFilter extends ZuulFilter {
    // 指定该过滤器的过滤类型
    // 此时是后置过滤器
    @Override
    public String filterType() {
        return FilterConstants.POST_TYPE;
    }
    // SEND_RESPONSE_FILTER_ORDER 是最后一个过滤器
    // 我们此过滤器在它之前执行
    @Override
    public int filterOrder() {
        return FilterConstants.SEND_RESPONSE_FILTER_ORDER - 1;
    }
    @Override
    public boolean shouldFilter() {
        return true;
    }
    // 过滤时执行的策略
    @Override
    public Object run() throws ZuulException {
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletRequest request = context.getRequest();
        // 从RequestContext获取原先的开始时间 并通过它计算整个时间间隔
        Long startTime = (Long) context.get("startTime");
        // 这里我可以获取HttpServletRequest来获取URI并且打印出来
        String uri = request.getRequestURI();
        long duration = System.currentTimeMillis() - startTime;
        log.info("uri: " + uri + ", duration: " + duration / 100 + "ms");
        return null;
    }
}

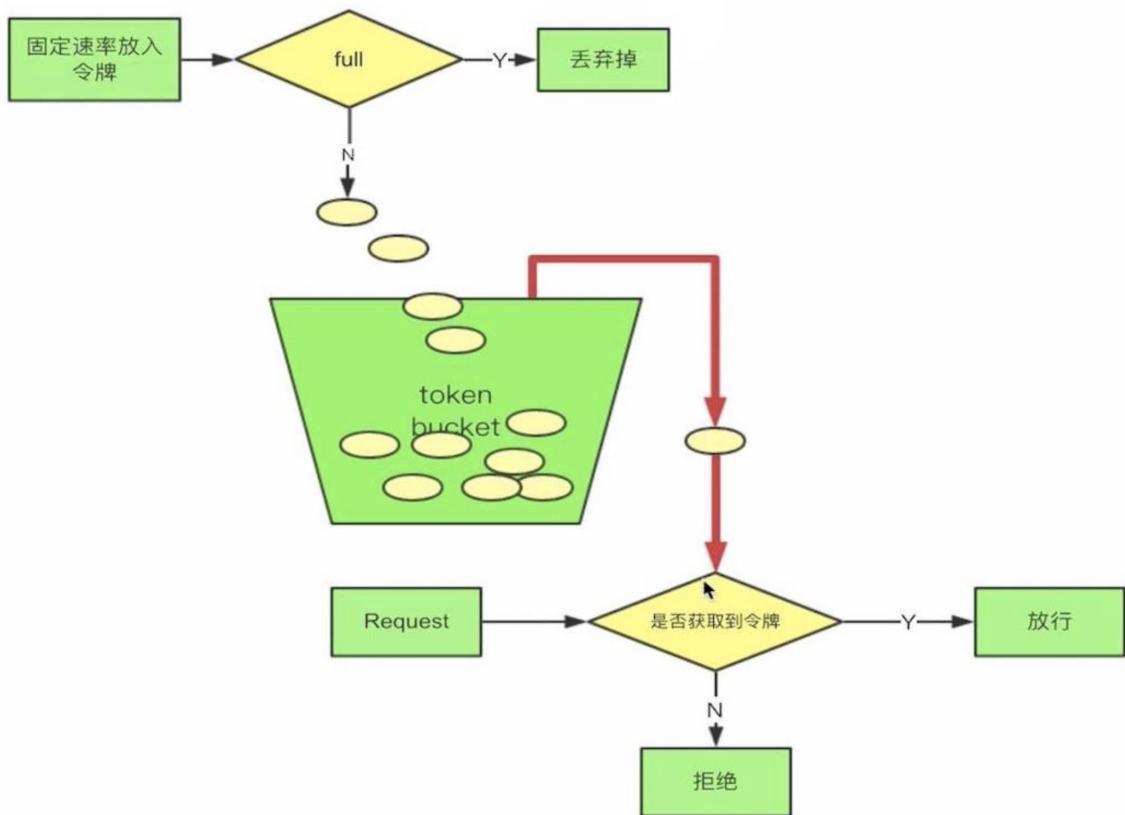
```

上面就简单实现了请求时间日志打印功能，你有没有感受到 zuul 过滤功能的强大了呢？

没有？好的、那我们再来。

令牌桶限流

当然不仅仅是令牌桶限流方式，zuul 只要是限流的活它都能干，这里我只是简单举个例子



我先来解释一下什么是 令牌桶限流 吧。

首先我们会有个桶，如果里面没有满那么就会以一定 **固定的速率** 会往里面放令牌，一个请求过来首先要从桶中获取令牌，如果没有获取到，那么这个请求就拒绝，如果获取到那么就放行。很简单吧，啊哈哈。

下面我们就通过 zuul 的前置过滤器来实现一下令牌桶限流。

```

package com.lqq.zuul.filter;

import com.google.common.util.concurrent.RateLimiter;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;
import org.springframework.stereotype.Component;

@Component
@Slf4j
public class RouteFilter extends ZuulFilter {
    // 定义一个令牌桶，每秒产生2个令牌，即每秒最多处理2个请求
    private static final RateLimiter RATE_LIMITER = RateLimiter.create(2);

    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }

    @Override
    public int filterOrder() {
        return -5;
    }

    @Override

```

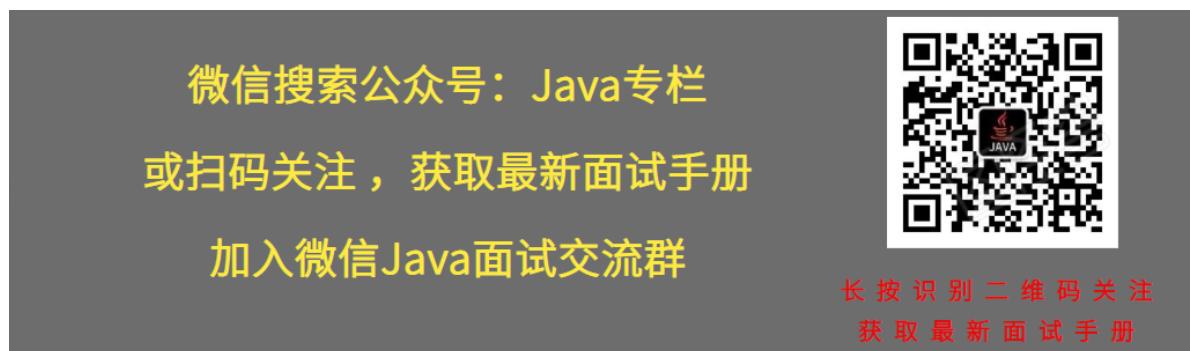
```
public Object run() throws ZuulException {
    log.info("放行");
    return null;
}

@Override
public boolean shouldFilter() {
    RequestContext context = RequestContext.getCurrentContext();
    if(!RATE_LIMITER.tryAcquire()) {
        log.warn("访问量超载");
        // 指定当前请求未通过过滤
        context.setSendZuulResponse(false);
        // 向客户端返回响应码429, 请求数量过多
        context.setResponseStatus(429);
        return false;
    }
    return true;
}
}
```

这样我们就能将请求数量控制在一秒两个，有没有觉得很酷？

中间件相关

Dubbo面试题



1. Dubbo是什么？

Dubbo是阿里巴巴开源的基于 Java 的高性能 RPC 分布式服务框架，现已成为 Apache 基金会孵化项目。

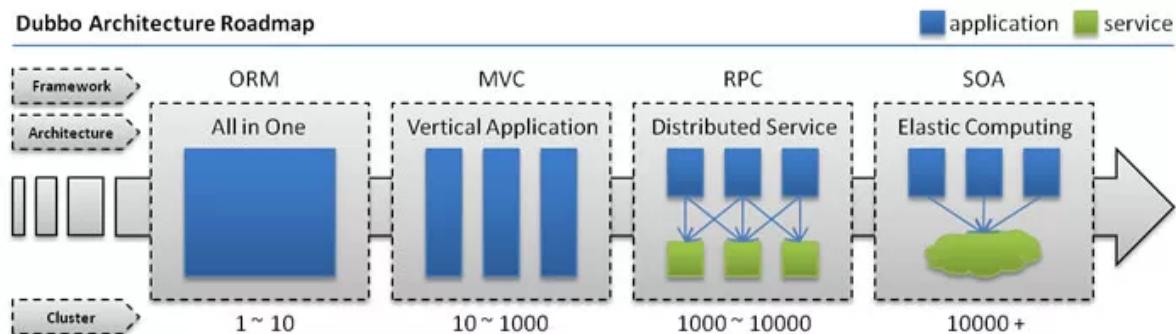
面试官问你如果这个都不清楚，那下面的就没必要问了。

2. 为什么要用Dubbo?

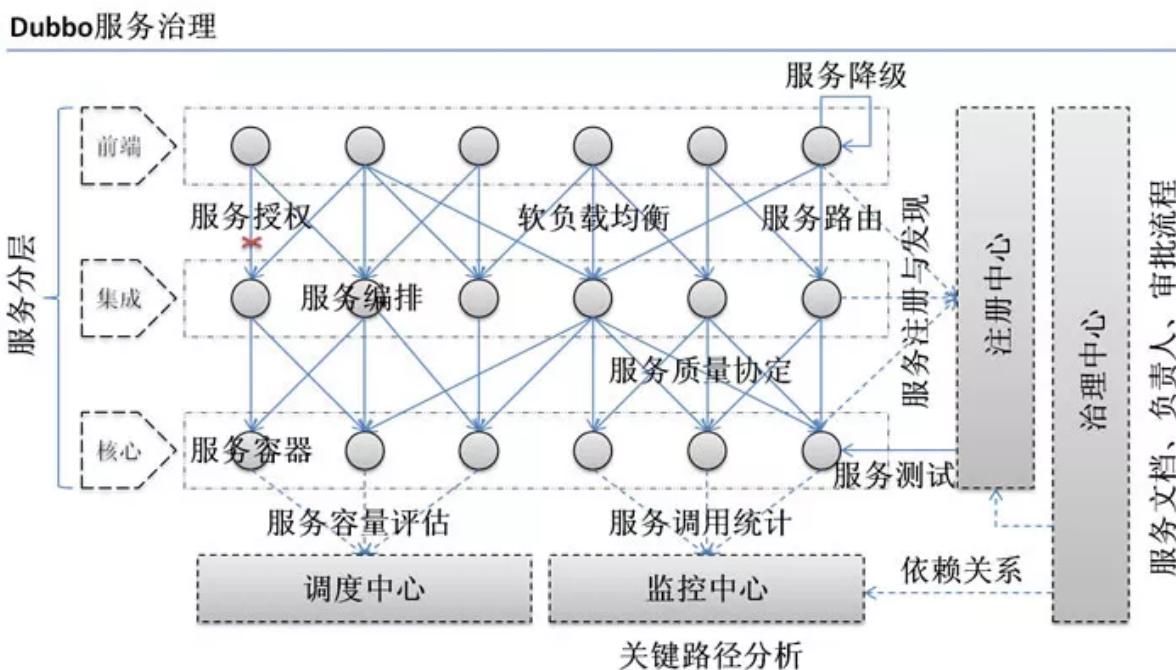
因为是阿里开源项目，国内很多互联网公司都在用，已经经过很多线上考验。内部使用了 Netty、Zookeeper，保证了高性能高可用性。

使用 Dubbo 可以将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，可用于提高业务复用灵活扩展，使前端应用能更快速的响应多变的市场需求。

下面这张图可以很清楚的诠释，最重要的一点是，分布式架构可以承受更大规模的并发流量。



下面是 Dubbo 的服务治理图。



3. Dubbo 和 Spring Cloud 有什么区别?

两个没关联，如果硬要说区别，有以下几点。

1、通信方式不同

Dubbo 使用的是 RPC 通信，而 Spring Cloud 使用的是 HTTP RESTful 方式。

2、组成部分不同

组件	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务监控	Dubbo-monitor	Spring Boot Admin
断路器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Gateway
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task
...

4. Dubbo都支持什么协议，推荐用哪种？

- 1、dubbo:// (推荐)
- 2、rmi://
- 3、hessian://
- 4、http://
- 5、webservice://
- 6、thrift://
- 7、memcached://
- 8、redis://
- 9、rest://

5. Dubbo需要 Web 容器吗？

不需要，如果硬要用 Web 容器，只会增加复杂性，也浪费资源。

6. Dubbo内置了哪几种服务容器?

- Spring Container
- Jetty Container
- Log4j Container

Dubbo 的服务容器只是一个简单的 Main 方法，并加载一个简单的 Spring 容器，用于暴露服务。

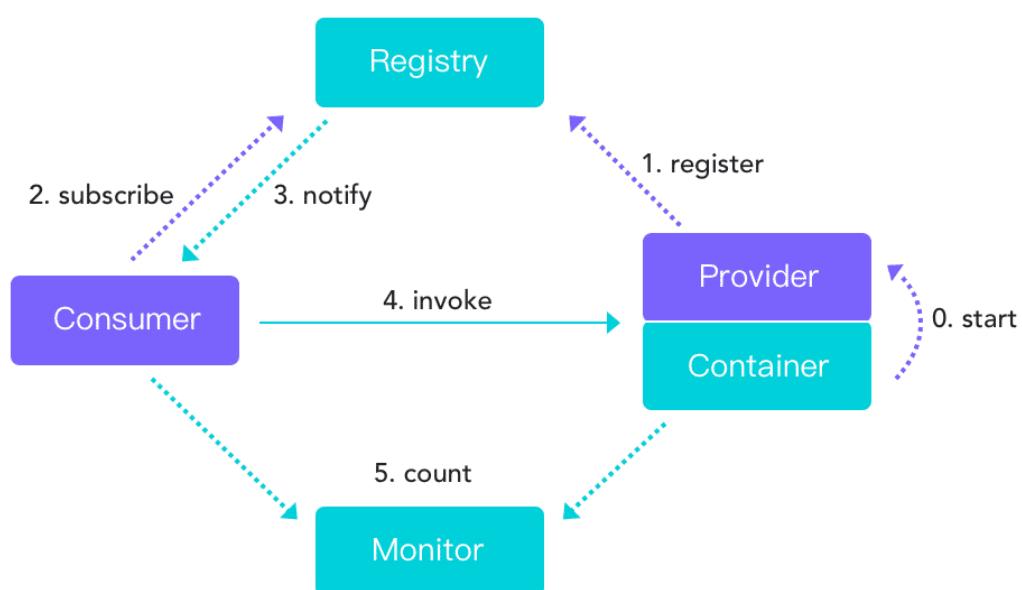
7. Dubbo里面有哪几种节点角色?

节点	角色说明
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	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

8. 画一画服务注册与发现的流程图

Dubbo Architecture

..... init async → sync



该图来自 Dubbo 官网，供你参考，如果说你熟悉 Dubbo，面试官经常会让你画这个图，记好了。
微信搜索公众号：Java专栏，获取最新面试手册

9. Dubbo默认使用什么注册中心，还有别的选择吗？

推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

10. Dubbo有哪几种配置方式？

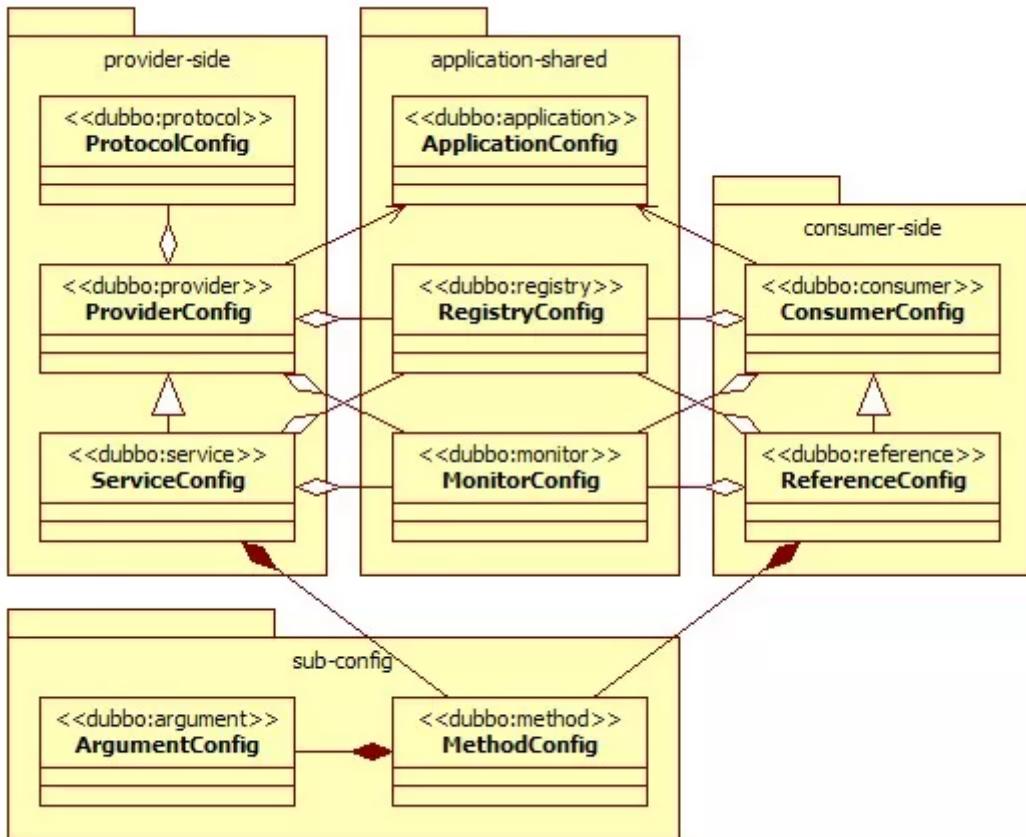
- 1、Spring 配置方式
- 2、Java API 配置方式

11. Dubbo 核心的配置有哪些？

我曾经面试就遇到过面试官让你写这些配置，我也是蒙逼。。

配置	配置说明
dubbo:service	服务配置
dubbo:reference	引用配置
dubbo:protocol	协议配置
dubbo:application	应用配置
dubbo:module	模块配置
dubbo:registry	注册中心配置
dubbo:monitor	监控中心配置
dubbo:provider	提供方配置
dubbo:consumer	消费方配置
dubbo:method	方法配置
dubbo:argument	参数配置

配置之间的关系见下图。



12. 在 Provider 上可以配置的 Consumer 端的属性有哪些？

- 1、timeout: 方法调用超时
- 2、retries: 失败重试次数，默认重试 2 次
- 3、loadbalance: 负载均衡算法，默认随机
- 4、actives 消费者端，最大并发调用限制

13. Dubbo启动时如果依赖的服务不可用会怎样？

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，默
认 check="true"，可以通过 check="false" 关闭检查。

14. Dubbo推荐使用什么序列化框架，你知道的还有哪些？

默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。hessian是一个采用二进制格式
传输的服务框架，相对传统soap web service，更轻量，更快速。

Hessian原理与协议简析：

http的协议约定了数据传输的方式，hessian也无法改变太多：

- 1、hessian中client与server的交互，基于http-post方式。
- 2、hessian将辅助信息，封装在http header中，比如“授权token”等，我们可以基于http-header来封装
关于“安全校验”“meta数据”等。hessian提供了简单的“校验”机制。

3、对于hessian的交互核心数据，比如“调用的方法”和参数列表信息，将通过post请求的body体直接发送，格式为字节流。

4、对于hessian的server端响应数据，将在response中通过字节流的方式直接输出。

hessian的协议本身并不复杂，在此不再赘言；所谓协议(protocol)就是约束数据的格式，client按照协议将请求信息序列化成字节序列发给server端，server端根据协议，将数据反序列化成“对象”，然后执行指定的方法，并将方法的返回值再次按照协议序列化成字节流，响应给client，client按照协议将字节流反序列话成“对象”。

15. Dubbo默认使用的是什么通信框架，还有别的选择吗？

Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有Mina、Grizzly。

16. Dubbo有哪几种集群容错方案，默认是哪种？

集群容错方案	说明
Failover Cluster	失败自动切换，自动重试其它服务器（默认）
Failstash Cluster	快速失败，立即报错，只发起一次调用
Failsafe Cluster	失败安全，出现异常时，直接忽略
Failback Cluster	失败自动恢复，记录失败请求，定时重发
Forking Cluster	并行调用多个服务器，只要一个成功即返回
Broadcast Cluster	广播逐个调用所有提供者，任意一个报错则报错

17. Dubbo有哪几种负载均衡策略，默认是哪种？

负载均衡策略	说明
Random LoadBalance	随机，按权重设置随机概率（默认）
RoundRobin LoadBalance	轮询，按公约后的权重设置轮询比率
LeastActive LoadBalance	最少活跃调用数，相同活跃数的随机
ConsistentHash LoadBalance	一致性 Hash，相同参数的请求总是发到同一提供者

18. 注册了多个同样的服务，如果测试指定的某一个服务呢？

可以配置环境点对点直连，绕过注册中心，将以服务接口为单位，忽略注册中心的提供者列表。

19. Dubbo 支持服务多协议吗？

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

20. Dubbo 支持哪些协议，每种协议的应用场景，优缺点？

dubbo: 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

rmi: 采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞；

webservice: 基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；**http**: 基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；**hessian**: 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache: 基于 memcached 实现的 RPC 协议 **redis**: 基于 redis 实现的 RPC 协议

21. Dubbo 集群的负载均衡有哪些策略

Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀；

RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题；

LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收更少的请求；**ConstantHash LoadBalance**: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

22. 当一个服务接口有多种实现时怎么做？

当一个接口有多种实现时，可以用 group 属性来分组，服务提供方和消费方都指定同一个 group 即可。

23. 服务调用超时问题怎么解决

dubbo在调用服务不成功时，默认是会重试两次的。这样在服务端的处理时间超过了设定的超时时间时，就会有重复请求，比如在发邮件时，可能就会发出多份重复邮件，执行注册请求时，就会插入多条重复的注册数据，那么怎么解决超时问题呢？如下对于核心的服务中心，去除dubbo超时重试机制，并重新评估设置超时时间。业务处理代码必须放在服务端，客户端只做参数验证和服务调用，不涉及业务流程处理 全局配置实例

```
<dubbo:provider delay="-1" timeout="6000" retries="0"/>
```

当然Dubbo的重试机制其实是非常好的QOS保证，它的路由机制，是会帮你把超时的请求路由到其他机器上，而不是本机尝试，所以 dubbo的重试机制也能一定程度的保证服务的质量。但是请一定要综合线上的访问情况，给出综合的评估。

24. 服务上线怎么兼容旧版本？

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

25. Dubbo可以对结果进行缓存吗？

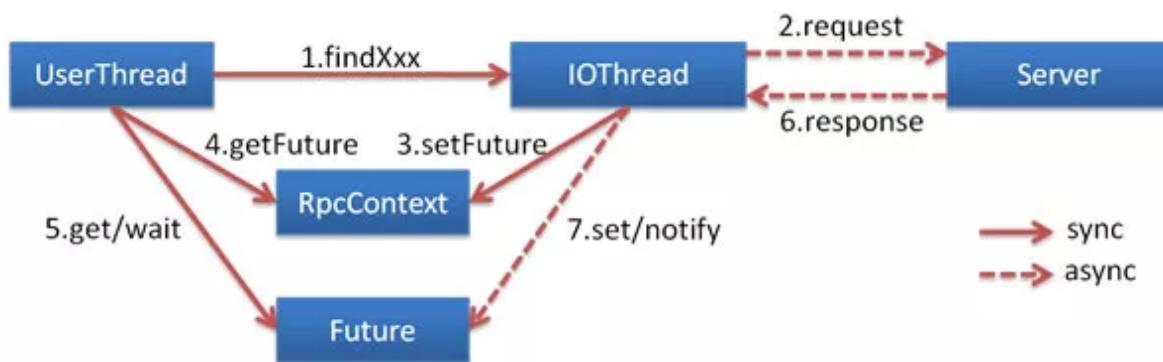
可以，Dubbo 提供了声明式缓存，用于加速热门数据的访问速度，以减少用户加缓存的工作量。

26. Dubbo服务之间的调用是阻塞的吗？

默认是同步等待结果阻塞的，支持异步调用。

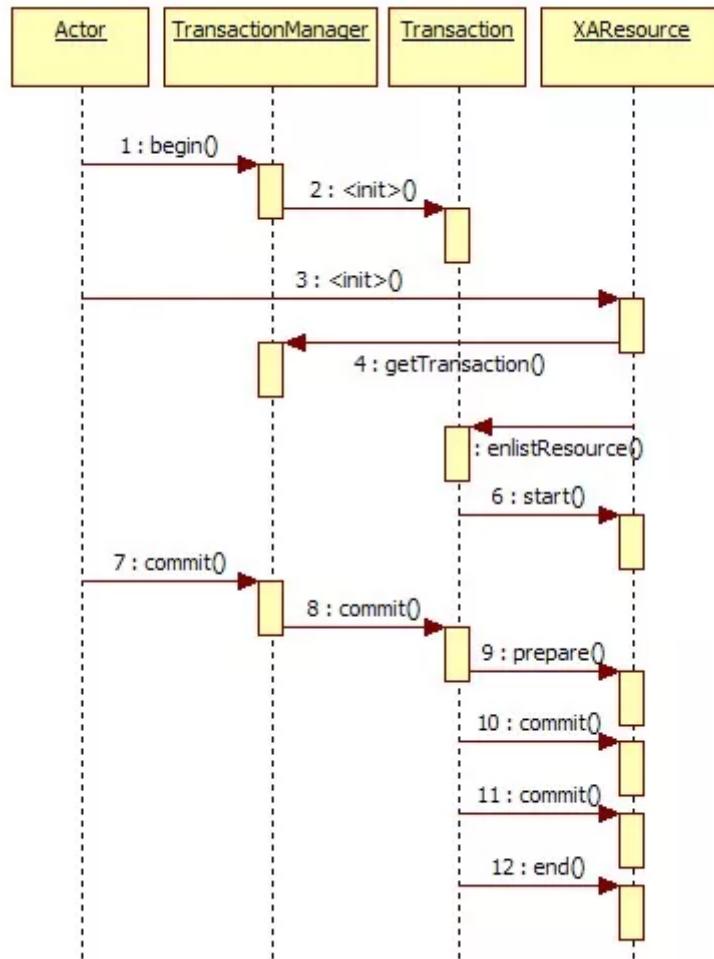
Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

异步调用流程图如下。



27. Dubbo支持分布式事务吗？

目前暂时不支持，后续可能采用基于 JTA/XA 规范实现，如以图所示。



28. Dubbo telnet 命令能做什么？

dubbo 通过 telnet 命令来进行服务治理

```
telnet localhost 8090
```

29. Dubbo如何优雅停机？

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

30. 服务提供者能实现失效踢出是什么原理？

服务失效踢出基于 Zookeeper 的临时节点原理。

31. 如何解决服务调用链过长的问题？

Dubbo 可以使用 Pinpoint 和 Apache Skywalking(Incubator) 实现分布式服务追踪，当然还有其他很多方案。

32. 服务读写推荐的容错策略是怎样的？

读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。

写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错。

33. Dubbo必须依赖的包有哪些？

Dubbo 必须依赖 JDK，其他为可选。

34. Dubbo的管理控制台能做什么？

管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。

35. 说说 Dubbo 服务暴露的过程。

Dubbo 会在 Spring 实例化完 bean 之后，在刷新容器最后一步发布 ContextRefreshEvent 事件的时候，通知实现了 ApplicationListener 的 ServiceBean 类进行回调 onApplicationEvent 事件方法，Dubbo 会在这个方法中调用 ServiceBean 父类 ServiceConfig 的 export 方法，而该方法真正实现了服务的（异步或者非异步）发布。

36. Dubbo 停止维护了吗？

2014 年开始停止维护过几年，17 年开始重新维护，并进入了 Apache 项目。

37、Dubbo 和 Dubbox 有什么区别？

Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加了服务可 Restful 调用，更新了开源组件等。

38. 在使用过程中都遇到了些什么问题？如何解决的？

- 1、同时配置了 XML 和 properties 文件，则 properties 中的配置无效只有 XML 没有配置时，properties 才生效。
- 2、dubbo 缺省会在启动时检查依赖是否可用，不可用就抛出异常，阻止 spring 初始化完成，check 属性默认为 true。测试时有些服务不关心或者出现了循环依赖，将 check 设置为 false

3、为了方便开发测试，线下有一个所有服务可用的注册中心，这时，如果有一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行。

解决：让服务提供者开发方，只订阅服务，而不注册正在开发的服务，通过直连测试正在开发的服务。
设置 dubbo:registry 标签的 register 属性为 false。

4、spring 2.x 初始化死锁问题。在 spring 解析到 dubbo:service 时，就已经向外暴露了服务，而 spring 还在接着初始化其他 bean，如果这时有请求进来，并且服务的实现类里有调用 applicationContext.getBean() 的用法。getBean 线程和 spring 初始化线程的锁的顺序不一样，导致了线程死锁，不能提供服务，启动不了。

解决：不要在服务的实现类中使用 applicationContext.getBean(); 如果不想依赖配置顺序，可以将 dubbo:provider 的 deploy 属性设置为 -1，使 dubbo 在容器初始化完成后再暴露服务。

5、服务注册不上

检查 dubbo 的 jar 包有没有在 classpath 中，以及有没有重复的 jar 包

检查暴露服务的 spring 配置有没有加载

在服务提供者机器上测试与注册中心的网络是否通

6、出现 RpcException: No provider available for remote service 异常

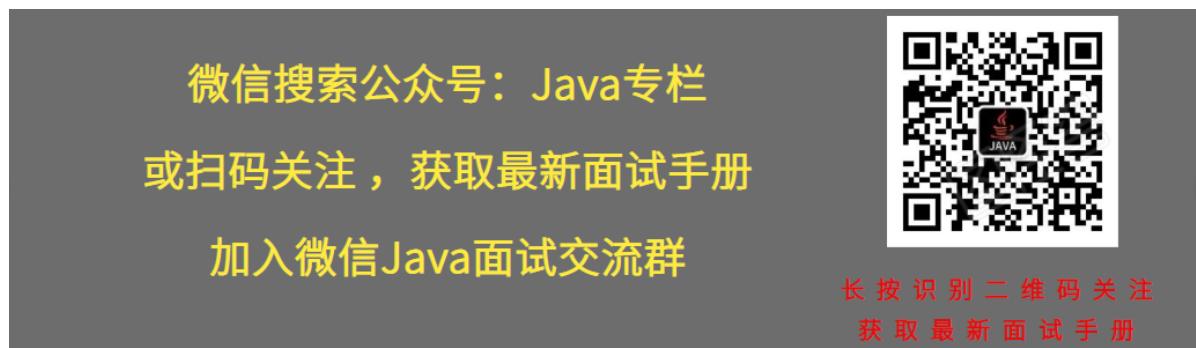
表示没有可用的服务提供者，

- 检查连接的注册中心是否正确
- 到注册中心查看相应的服务提供者是否存在
- 检查服务提供者是否正常运行

7、出现“消息发送失败”异常

通常是接口方法的传入传出参数未实现 Serializable 接口。

Nginx面试题



1. 请解释一下什么是 Nginx？

Nginx是一个web服务器和反向代理服务器，用于 HTTP、HTTPS、SMTP、POP3 和 IMAP 协议。

2. 什么是正向代理和反向代理?

- 正向代理就是一个人发送一个请求直接就到达了目标的服务器
- 反方代理就是请求统一被Nginx接收，nginx反向代理服务器接收到之后，按照一定的规则分发给了后端的业务处理服务器进行处理了

3. 使用“反向代理服务器的优点是什么?

反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和web服务器之间的中间层。这对于安全方面来说是很好的，特别是当您使用web托管服务时。

4. 请列举 Nginx 的一些特性。

Nginx 服务器的特性包括：

反向代理/L7负载均衡器

嵌入式Perl解释器

动态二进制升级

可用于重新编写URL，具有非常好的PCRE支持

5. Nginx的优缺点?

优点：

- 占内存小，可实现高并发连接，处理响应快
- 可实现http服务器、虚拟主机、方向代理、负载均衡
- Nginx配置简单
- 可以不暴露正式的服务器IP地址

缺点：

- 动态处理差：nginx处理静态文件好,耗费内存少，但是处理动态页面则很鸡肋，现在一般前端用nginx作为反向代理抗住压力，

6. 为什么Nginx性能这么高?

因为他的事件处理机制：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

7. Nginx应用场景?

- http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。
- 虚拟主机。可以实现在一台服务器虚拟出多个网站，例如个人网站使用的虚拟机。
- 反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要多台服务器集群可以使用nginx做反向代理。并且多台服务器可以平均分担负载，不会应为某台服务器负载高宕机而某台服务器闲置的情况。
- nginx 中也可以配置安全管理、比如可以使用Nginx搭建API接口网关,对每个接口服务进行拦截。

8. 请解释Nginx服务器上的Master和Worker进程分别是什么？

Master进程：读取及评估配置和维持

Worker进程：处理请求

9. 请解释你如何通过不同于80的端口开启Nginx？

为了通过一个不同的端口开启Nginx，你必须进入`/etc/Nginx/sites-enabled/`，如果这是默认文件，那么你必须打开名为“default”的文件。编辑文件，并放置在你想要的端口：

```
Like server { listen 81; }
```

10. Nginx负载均衡的算法怎么实现的？策略有哪些？

为了避免服务器崩溃，大家会通过负载均衡的方式来分担服务器压力。将对台服务器组成一个集群，当用户访问时，先访问到一个转发服务器，再由转发服务器将访问分发到压力更小的服务器。

Nginx负载均衡实现的策略有以下五种：

1、轮询(默认)

每个请求按时间顺序逐一分配到不同的后端服务器，如果后端某个服务器宕机，能自动剔除故障系统。

```
upstream backserver {
    server 192.168.0.12;
    server 192.168.0.13;
}
```

2、权重 weight

weight的值越大分配

到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下。其次是为在主从的情况下设置不同的权值，达到合理有效地利用主机资源。

```
upstream backserver {
    server 192.168.0.12 weight=2;
    server 192.168.0.13 weight=8;
}
```

权重越高，在被访问的概率越大，如上例，分别是20%，80%。

3、ip_hash(IP绑定)

每个请求按访问IP的哈希结果分配，使来自同一个IP的访客固定访问一台后端服务器，并且可以有效解决动态网页存在的session共享问题

```
upstream backserver {  
    ip_hash;  
    server 192.168.0.12:88;  
    server 192.168.0.13:80;  
}
```

4、fair(第三方插件)

必须安装upstream_fair模块。

对比 weight、ip_hash更加智能的负载均衡算法，fair算法可以根据页面大小和加载时间长短智能地进行负载均衡，响应时间短的优先分配。

```
upstream backserver {  
    server server1;  
    server server2;  
    fair;  
}
```

哪个服务器的响应速度快，就将请求分配到那个服务器上。

5、url_hash(第三方插件)

必须安装Nginx的hash软件包

按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，可以进一步提高后端缓存服务器的效率。

```
upstream backserver {  
    server squid1:3128;  
    server squid2:3128;  
    hash $request_uri;  
    hash_method crc32;  
}
```

11. Nginx配置高可用性怎么配置？

当上游服务器(真实访问服务器)，一旦出现故障或者是没有及时相应的话，应该直接轮训到下一台服务器，保证服务器的高可用

Nginx配置代码：

```
server {  
    listen      80;  
    server_name www.lijie.com;  
    location / {  
        ##### 指定上游服务器负载均衡服务器  
        proxy_pass http://backserver;  
        #####nginx与上游服务器(真实访问的服务器)超时时间 后端服务器连接的超时时间_发起握手  
       等候响应超时时间  
        proxy_connect_timeout 1s;
```

```
    #####nginx发送给上游服务器(真实访问的服务器)超时时间
    proxy_send_timeout 1s;
    ##### nginx接受上游服务器(真实访问的服务器)超时时间
    proxy_read_timeout 1s;
    index  index.html index.htm;
}
}
```

12. Nginx怎么判断IP不可访问?

```
# 如果访问的ip地址为192.168.9.115，则返回403
if ($remote_addr = 192.168.9.115) {
    return 403;
}
```

13. 怎么限制浏览器访问?

```
## 不允许谷歌浏览器访问 如果是谷歌浏览器返回500
if ($http_user_agent ~ Chrome) {
    return 500;
}
```

14. Nginx目录结构有哪些?

```
[root@localhost ~]# tree /usr/local/nginx
/usr/local/nginx
├── client_body_temp
├── conf                                # Nginx所有配置文件的目录
│   ├── fastcgi.conf                      # fastcgi相关参数的配置文件
│   ├── fastcgi.conf.default              # fastcgi.conf的原始备份文件
│   ├── fastcgi_params                    # fastcgi的参数文件
│   ├── fastcgi_params.default            # fastcgi_params的默认值
│   ├── koi-utf
│   ├── koi-win
│   ├── mime.types                       # 媒体类型
│   ├── mime.types.default                # mime.types的默认值
│   ├── nginx.conf                        # Nginx主配置文件
│   ├── nginx.conf.default                # nginx.conf的默认值
│   ├── scgi_params                      # scgi相关参数文件
│   ├── scgi_params.default              # scgi_params的默认值
│   ├── uwsgi_params                     # uwsgi相关参数文件
│   ├── uwsgi_params.default              # uwsgi_params的默认值
│   └── win-utf
└── fastcgi_temp                         # fastcgi临时数据目录
    └── html                               # Nginx默认站点目录
        ├── 50x.html                         # 错误页面优雅替代显示文件，例如当出现502错误时会调用此页面
        └── index.html                       # 默认的首页文件
```

```

├── logs                                # Nginx日志目录
│   ├── access.log                         # 访问日志文件
│   ├── error.log                           # 错误日志文件
│   └── nginx.pid                          # pid文件, Nginx进程启动后, 会把所有进程的ID号写到此文件
├── proxy_temp                            # 临时目录
└── sbin                                 # Nginx命令目录
    └── nginx                             # Nginx的启动命令
├── scgi_temp                            # 临时目录
└── uwsgi_temp                           # 临时目录

```

15. Nginx配置文件nginx.conf有哪些属性模块?

```

worker_processes 1;                                # worker进程的数量
events {                                         # 事件区块开始
    worker_connections 1024;                      # 每个worker进程支持的最大连接数
}
http {                                           # HTTP区块开始
    include      mime.types;                      # Nginx支持的媒体类型库文件
    default_type application/octet-stream;        # 默认的媒体类型
    sendfile     on;                            # 开启高效传输模式
    keepalive_timeout 65;                        # 连接超时
    server {                                     # 第一个Server区块开始, 表示一个独立的虚拟主机站点
        listen       80;                          # 提供服务的端口, 默认80
        server_name localhost;                   # 提供服务的域名主机名
        location / {                           # 第一个location区块开始
            root   html;                        # 站点的根目录, 相当于Nginx的安装目录
            index  index.html index.htm;        # 默认的首页文件, 多个用空格分开
        }                                       # 第一个location区块结束
        error_page 500 502 503 504 /50x.html;    # 出现对应的http状态码时, 使用50x.html回应客户
        location = /50x.html {                  # location区块开始, 访问50x.html
            root   html;                        # 指定对应的站点目录为html
        }
    }
    .....

```

16. 为什么Nginx性能这么高?

因为他的事件处理机制：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

17. Nginx静态资源?

静态资源访问，就是存放在nginx的html页面，我们可以自己编写

18. 如何用Nginx解决前端跨域问题?

使用Nginx转发请求。把跨域的接口写成调本域的接口，然后将这些接口转发到真正的请求地址。

19. Nginx怎么处理请求的?

nginx接收一个请求后，首先由listen和server_name指令匹配server模块，再匹配server模块里的location，location就是实际地址

```
server {  
    # 第一个Server区块开始，表示一个独立的虚拟主机  
    站点  
    listen      80;          # 提供服务的端口，默认80  
    server_name localhost;  # 提供服务的域名主机名  
    location / {  
        # 第一个location区块开始  
        root   html;          # 站点的根目录，相当于Nginx的安装目录  
        index  index.html index.htm;  # 默认的首页文件，多个用空格分开  
    }  
    # 第一个location区块结束  
}
```

20. Nginx虚拟主机怎么配置?

- 1、基于域名的虚拟主机，通过域名来区分虚拟主机——应用：外部网站
- 2、基于端口的虚拟主机，通过端口来区分虚拟主机——应用：公司内部网站，外部网站的管理后台
- 3、基于ip的虚拟主机。

基于虚拟主机配置域名

需要建立/data/www /data/bbs目录，windows本地hosts添加虚拟机ip地址对应的域名解析；对应域名网站目录下新增index.html文件；

```
#当客户端访问www.lijie.com,监听端口号为80,直接跳转到data/www目录下文件  
server {  
    listen      80;  
    server_name www.lijie.com;  
    location / {  
        root   data/www;  
        index  index.html index.htm;  
    }  
}  
  
#当客户端访问www.lijie.com,监听端口号为80,直接跳转到data/bbs目录下文件  
server {  
    listen      80;  
    server_name bbs.lijie.com;  
    location / {  
        root   data/bbs;  
        index  index.html index.htm;  
    }  
}
```

基于端口的虚拟主机

使用端口来区分，浏览器使用域名或ip地址:端口号 访问

```
#当客户端访问www.lijie.com,监听端口号为8080,直接跳转到data/www目录下文件
server {
    listen      8080;
    server_name 8080.lijie.com;
    location / {
        root   data/www;
        index  index.html index.htm;
    }
}

#当客户端访问www.lijie.com,监听端口号为80直接跳转到真实ip服务器地址 127.0.0.1:8080
server {
    listen      80;
    server_name www.lijie.com;
    location / {
        proxy_pass http://127.0.0.1:8080;
        index  index.html index.htm;
    }
}
```

21. location的作用是什么？

location指令的作用是根据用户请求的URI来执行不同的应用，也就是根据用户请求的网站URL进行匹配，匹配成功即进行相关的操作

location的语法能说出来吗？

注意：~代表自己输入的英文字母

匹配符	匹配规则	优先级
=	精确匹配	1
^~	以某个字符串开头	2
~	区分大小写的正则匹配	3
~*	不区分大小写的正则匹配	4
!~	区分大小写不匹配的正则	5
!~*	不区分大小写不匹配的正则	6
/	通用匹配, 任何请求都会匹配到	7

22. Location正则案例

示例：

```
#优先级1, 精确匹配, 根路径
location =/ {
    return 400;
}

#优先级2, 以某个字符串开头, 以av开头的, 优先匹配这里, 区分大小写
location ^~ /av {
    root /data/av/;
}

#优先级3, 区分大小写的正则匹配, 匹配/media*****路径
location ~ /media {
    alias /data/static/;
}

#优先级4 , 不区分大小写的正则匹配, 所有的****.jpg|gif|png 都走这里
location ~* .*\.(jpg|gif|png|js|css)$ {
    root /data/av/;
}

#优先7, 通用匹配
location / {
    return 403;
}
```

23. Nginx常用变量

- \$host: 请求的主机头

```
if ($host = 'bbs.gitlib.com') {  
    rewrite ^/$ http://bbs.bliwan.com permanent;  
}
```

- \$remote_addr: 客户端IP地址
- \$remote_port: 客户端端口号
- \$remote_user: 已经经过Auth Basic Module验证的用户名
- \$http_referer: 请求引用地址
- \$http_user_agent: 客户端代理信息(UA)
- \$http_x_forwarded_for: 相当于网络访问路径
- \$body_bytes_sent: 页面传送的字节数
- \$time_local: 服务器时间
- \$request: 客户端请求
- \$request_uri: 请求的URI,带参数, 不包含主机名
- \$request_filename: 请求的文件路径
- \$request_method: 请求的方法, 如GET、POST
- \$args: 客户端请求中的参数
- \$query_string: 等同于\$args, 客户端请求的参数
- \$nginx_version: 当前nginx版本
- \$status: 服务器响应状态码
- \$server_addr: 服务器地址
- \$server_port: 请求到达的服务器端口号
- \$server_protocol: 请求的协议版本
- \$content_type: HTTP请求信息里的Content-Type字段
- \$content_length: HTTP请求信息里的Content-Length字段
- \$uri: 请求中的当前URI(不带请求参数, 参数位于\$args)
- \$document_root: 当前请求在root指令中指定的值
- \$document_uri: 与\$uri相同

24. Nginx设置重定向

return形式

```
# 301永久重定向，302临时重定向
return 301 https://example.com$request_uri;

# return 返回形式
return code;
return code URL;
return URL;
```

rewrite形式

```
rewrite ^/$ http://bbs.gitlib.com permanent;
```

rewrite flag说明:

last: 停止处理后续**rewrite**指令集，然后对当前重写的新URI在**rewrite**指令集上重新查找

break: 停止处理后续**rewrite**指令集，并不在重新查找，但是当前**location**内剩余非**rewrite**语句和**location**外的非**rewrite**语句可以执行

redirect: 如果**replacement**不是以**http://**或**https://**开始，返回302临时重定向

permant: 返回301永久重定向

25. 限流怎么做的？

Nginx限流就是限制用户请求速度，防止服务器受不了

限流有3种

- 正常限制访问频率（正常流量）
- 突发限制访问频率（突发流量）
- 限制并发连接数

Nginx的限流都是基于漏桶流算法，底下会说道什么是桶铜流

实现三种限流算法

1、正常限制访问频率（正常流量）：

限制一个用户发送的请求，我Nginx多久接收一个请求。

Nginx中使用**ngx_http_limit_req_module**模块来限制的访问频率，限制的原理实质是基于漏桶算法原理来实现的。在**nginx.conf**配置文件中可以使用**limit_req_zone**命令及**limit_req**命令限制单个IP的请求处理频率。

```
#定义限流维度，一个用户一分钟一个请求进来，多余的全部漏掉
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/m;

#绑定限流维度
server{

    location/seckill.html{
        limit_req zone=zone;
        proxy_pass http://lj_seckill;
    }

}
```

1r/s代表1秒一个请求，1r/m一分钟接收一个请求，如果Nginx这时还有别人的请求没有处理完，Nginx就会拒绝处理该用户请求。

2、突发限制访问频率（突发流量）：

限制一个用户发送的请求，我Nginx多久接收一个。

上面的配置一定程度可以限制访问频率，但是也存在着一个问题：如果突发流量超出请求被拒绝处理，无法处理活动时候的突发流量，这时候应该如何进一步处理呢？

Nginx提供burst参数结合nodelay参数可以解决流量突发的问题，可以设置能处理的超过设置的请求数外能额外处理的请求数。我们可以将之前的例子添加burst参数以及nodelay参数：

```
#定义限流维度，一个用户一分钟一个请求进来，多余的全部漏掉
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/m;

#绑定限流维度
server{

    location/seckill.html{
        limit_req zone=zone burst=5 nodelay;
        proxy_pass http://lj_seckill;
    }

}
```

为什么就多了一个 burst=5 nodelay; 呢，多了这个可以代表Nginx对于一个用户的请求会立即处理前五个，多余的就慢慢来落，没有其他用户的请求我就处理你的，有其他的请求的话我Nginx就漏掉不接受你的请求

3、限制并发连接数

Nginx中的ngx_http_limit_conn_module模块提供了限制并发连接数的功能，可以使用limit_conn_zone指令以及limit_conn执行进行配置。接下来我们可以通过一个简单的例子来看下：

```
http {
    limit_conn_zone $binary_remote_addr zone=myip:10m;
    limit_conn_zone $server_name zone=myServerName:10m;
}

server {
    location / {
        limit_conn myip 10;
        limit_conn myServerName 100;
        rewrite / http://www.lijie.net permanent;
    }
}
```

上面配置了单个IP同时并发连接数最多只能10个连接，并且设置了整个虚拟服务器同时最大并发数最多只能100个链接。当然，只有当请求的header被服务器处理后，虚拟服务器的连接数才会计数。

26. 为什么要做动静分离？

- Nginx是当下最热的Web容器，网站优化的重要点在于静态化网站，网站静态化的关键点则是是动静分离，动静分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来，动静资源做好了拆分以后，我们则根据静态资源的特点将其做缓存操作。
- 让静态的资源只走静态资源服务器，动态的走动态的服务器
- Nginx的静态处理能力很强，但是动态处理能力不足，因此，在企业中常用动静分离技术。
- 对于静态资源比如图片，js，css等文件，我们则在反向代理服务器nginx中进行缓存。这样浏览器在请求一个静态资源时，代理服务器nginx就可以直接处理，无需将请求转发给后端服务器tomcat。
- 若用户请求的动态文件，比如servlet,jsp则转发给Tomcat服务器处理，从而实现动静分离。这也是反向代理服务器的一个重要的作用。

27. Nginx怎么做的动静分离？

只需要指定路径对应的目录。location/可以使用正则表达式匹配。并指定对应的硬盘中的目录。如下：
(操作都是在Linux上)

```
location /image/ {
    root /usr/local/static/;
    autoindex on;
}
```

1. 创建目录

```
mkdir /usr/local/static/image
```

2. 进入目录

```
cd /usr/local/static/image
```

3. 放一张照片上去

1.jpg

4.重启 nginx

```
sudo nginx -s reload
```

打开浏览器 输入 server_name/image/1.jpg 就可以访问该静态图片了

28. Nginx条件判断

if判断

```
if ($http_user_agent ~ (125LA|WinHttpRequest|360Spider)) {
    return 444;
}

if ($http_referer ~* "filter=author&orderby=dateline") {
    return 444;
}

if ($host = 'bbs.gitlib.com') {
    rewrite ^/$ http://bbs1.gitlib.com permanent;
}
```

比较符说明:

- 使用=、!= 比较的一个变量和字符串，true/false
- 使用~、~*与正则表达式匹配的变量，如果这个正则表达式中包含右花括号}或者分号;则必须给整个正则表达式加引号
- 使用-f、!-f 检查一个文件是否存在
- 使用-d、!-d 检查一个目录是否存在
- 使用-e、!-e 检查一个文件、目录、符号链接是否存在
- 使用-x、!-x 检查一个文件是否可执行

set设置变量

```
if ($host ~* (.*)\.yzz\.cn) {
    set $domain $1;
}
root /www/website/www/gitlib/$domain/;

# set语法
set variable value;
```

Zookeeper面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

1. ZooKeeper 是什么？

据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 保证了如下分布式一致性特性：

- 1、顺序一致性
- 2、原子性
- 3、单一视图
- 4、可靠性
- 5、实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 znode (Zookeeper Transaction Id)。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper 最新的 znode。

2. ZooKeeper 提供了什么？

- 1、文件系统
- 2、通知机制

3. Zookeeper 都有哪些功能？

- 1、集群管理：监控节点存活状态、运行请求等；
- 2、主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 Zookeeper 可以协助完成这个过程；
- 3、分布式锁：Zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。Zookeeper 可以对分布式锁进行控制。

4、命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

4. Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。

5. 有哪些著名的开源项目用到了 ZooKeeper？

1、Kafka : ZooKeeper 主要为 Kafka 提供 Broker 和 Topic 的注册以及多个 Partition 的负载均衡等功能。

2、Hbase : ZooKeeper 为 Hbase 提供确保整个集群只有一个 Master 以及保存和提供 regionserver 状态信息（是否在线）等功能。

3、Hadoop : ZooKeeper 为 Namenode 提供高可用支持。

6. 说一下 Zookeeper 的通知机制？

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

7. Zookeeper 机制的特点

1、一次性触发：触发数据发生改变时，一个 watcher event 会被发送到 client，但是 client 只会收到一次这样的信息。

2、异步通知：watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。

Zookeeper 只能保证最终的一致性，而无法保证强一致性。

3、数据监视：Zookeeper 有数据监视和子数据监视 `getdata()` and `exists()` 设置数据监视，`getchildren()` 设置了子节点监视。

4、注册 watcher `getData`、`exists`、`getChildren`

5、触发 watcher `create`、`delete`、`setData`

6、`setData()` 触发：znode 上设置的 data watch（如果 set 成功的话）。一个成功的 `create()` 操作会触发被创建的 znode 上的数据 watch，以及其父节点上的 child watch。而一个成功的 `delete()` 操作将同时触发一个 znode 的 data watch 和 child watch（因为这样就没有子节点了），同时也会触发其父节点的 childwatch。

7、当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。

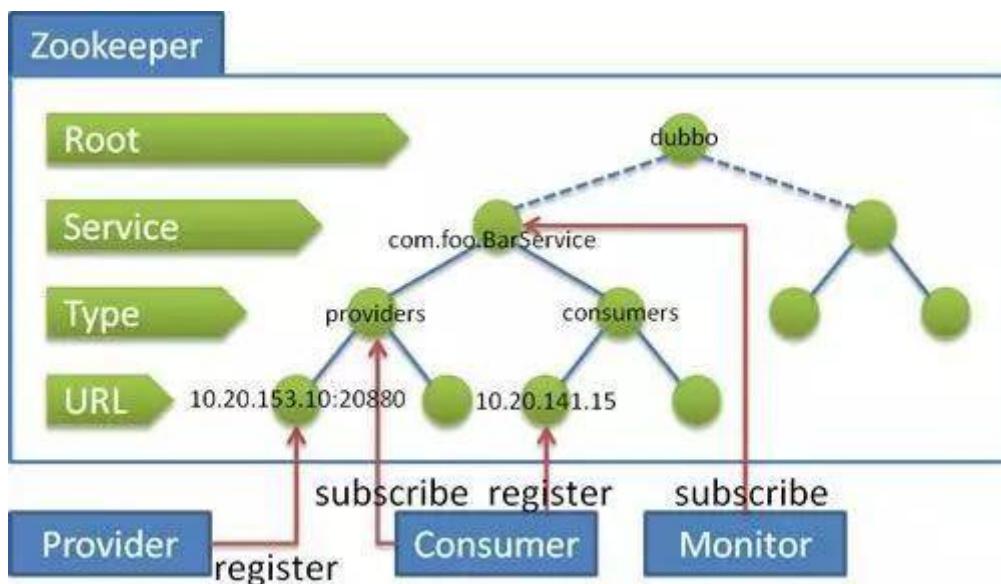
只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

8、Watch是轻量级的，其实就是本地 JVM 的 Callback，服务器端只是存了是否有设置了 Watcher 的布尔类型。

8. Zookeeper 和 Dubbo 的关系？

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

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



9. Zookeeper的java客户端都有哪些？

java客户端：zk自带的zkclient及Apache开源的Curator。

10. Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用，配合 Watcher 事件通知机制，可以非常方便的构建一系列分布式应用中都会涉及的核心功能，如：

1、数据发布/订阅

2、负载均衡

3、命名服务

4、分布式协调/通知

5、集群管理

6、Master 选举

7、分布式锁

8、分布式队列

数据发布/订阅

介绍

数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

目的

动态获取数据（配置信息）

实现数据（配置信息）的集中式管理和数据的动态更新

设计模式

Push 模式

Pull 模式

数据（配置信息）特性

(1) 数据量通常比较小

(2) 数据内容在运行时会发生动态更新

(3) 集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等

基于 Zookeeper 的实现方式

· 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点

· 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher

· 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper 会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

负载均衡

zk 的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。

对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

zk 的命名服务（文件系统）

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

zk 的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。

对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

Zookeeper 分布式锁（文件系统、通知机制）

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。

对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

- (1) 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
- (2) 队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

11. Zookeeper 怎么保证主从节点的状态同步？

Zookeeper 的核心是原子广播机制，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。

1、恢复模式

当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

2、广播模式

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

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

可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

集群规则为 $2N+1$ 台， $N > 0$ ，即最少需要 3 台。

13. ZAB 协议

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

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

14. 四种类型的数据节点 Znode

1、PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上

2、EPHEMERAL-临时节点

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

3、PERSISTENT_SEQUENTIAL-持久顺序节点

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

4、EPHEMERAL_SEQUENTIAL-临时顺序节点

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

15. 说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

16. Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制：

- 1、客户端注册 watcher
- 2、服务端处理 watcher
- 3、客户端回调 watcher

Watcher 特性总结：

(1) 一次性

无论是服务端还是客户端，一旦一个 Watcher 被 触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

(2) 客户端串行执行

客户端 Watcher 回调的过程是一个串行同步的过程。

(3) 轻量

- Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
- 客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

4、watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。

Zookeeper 只能保证最终的一致性，而无法保证强一致性。

5、注册 watcher getData、exists、getChildren

6、触发 watcher create、delete、setData

7、当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

17. zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期；纪元；世；新时代）用来标识 leader 周期，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

18. 分布式集群中为什么会有 Master？

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

19. 集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：

全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容。

20. zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

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

所以

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

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

21. zookeeper 负载均衡和 nginx 负载均衡区别

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

22. Zookeeper 有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

23. 客户端注册 Watcher 实现

- 1、调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象
- 2、标记请求 request，封装 Watcher 到 WatchRegistration
- 3、封装成 Packet 对象，发服务端发送 request
- 4、收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理
- 5、请求返回，完成注册。

24. 服务端处理 Watcher 实现

1、服务端接收 Watcher 并存储

接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。

2、Watcher 触发

以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

- 封装 WatchedEvent
将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent 对象
- 查询 Watcher
从 WatchTable 中根据节点路径查找 Watcher
- 没找到；说明没有客户端在该数据节点上注册过 Watcher
- 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）

3、调用 process 方法来触发 Watcher

这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。

25. Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的？

不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。

一般是客户端执行 `getData("/节点 A",true)`，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。

在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

26. 服务器角色

Leader

- (1) 事务请求的唯一调度和处理者，保证集群事务处理的顺序性
- (2) 集群内部各服务的调度者

Follower

- (1) 处理客户端的非事务请求，转发事务请求给 Leader 服务器
- (2) 参与事务请求 Proposal 的投票
- (3) 参与 Leader 选举投票

Observer

- (1) 3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力
- (2) 处理客户端的非事务请求，转发事务请求给 Leader 服务器
- (3) 不参与任何形式的投票

27. Zookeeper 下 Server 工作状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

- 1、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。
- 2、FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。
- 3、LEADING：领导者状态。表明当前服务器角色是 Leader。
- 4、OBSERVING：观察者状态。表明当前服务器角色是 Observer。

28. 数据同步

整个集群完成 Leader 选举之后，Learner (Follower 和 Observer 的统称) 回向Leader 服务器进行注册。当 Learner 服务器想 Leader 服务器完成注册后，进入数据同步环节。

数据同步流程：（均以消息传递的方式进行）

Learner 向 Learder 注册

数据同步

同步确认

Zookeeper 的数据同步通常分为四类：

- 1、直接差异化同步 (DIFF 同步)
- 2、先回滚再差异化同步 (TRUNC+DIFF 同步)
- 3、仅回滚同步 (TRUNC 同步)
- 4、全量同步 (SNAP 同步)

在进行数据同步前，Leader 服务器会完成数据同步初始化：

peerLastZxid:

- 从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid (该Learner 服务器最后处理的 ZXID)

minCommittedLog:

- Leader 服务器 Proposal 缓存队列 committedLog 中最小 ZXIDmaxCommittedLog:
- Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID直接差异化同步 (DIFF 同步)
- 场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog之间先回滚再差异化同步 (TRUNC+DIFF 同步)
- 场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚--回滚到 Leader服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID仅回滚同步 (TRUNC 同步)

- 场景：peerLastZxid 大于 maxCommittedLog

全量同步 (SNAP 同步)

- 场景一：peerLastZxid 小于 minCommittedLog

- 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

29. zookeeper是如何保证事务的顺序一致性的？

zookeeper采用了全局递增的事务Id来标识，所有的proposal (提议) 都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch (时期; 纪元; 世; 新时代) 用来标识leader周期，如果有新的leader产生出来，epoch会自增，低32位用来递增计数。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

30. ZAB 和 Paxos 算法的联系与区别?

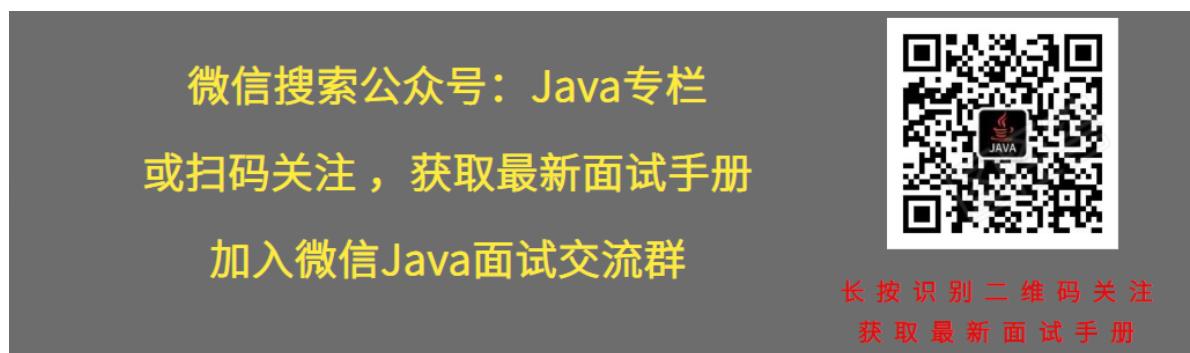
相同点:

- 1、两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
- 2、Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交
- 3、ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点:

ZAB 用来构建高可用的分布式数据主备系统 (Zookeeper)，Paxos 是用来构建分布式一致性状态机系统。

MQ面试题



1. RabbitMQ是什么？

MQ (Message Queue) 消息队列，是 "先进先出" 的一种数据结构。

MQ 一般用来解决应用解耦，异步处理，流量削峰等问题，实现高性能，高可用，可伸缩和最终一致性架构。

应用解耦：当 A 系统生产关键数据，发送数据给多个其他系统消费，此时 A 系统和其他系统产生了严重的耦合，如果将 A 系统产生的数据放到 MQ 当中，其他系统去 MQ 获取消费数据，此时各系统独立运行只与 MQ 交互，添加新系统消费 A 系统的数据也不需要去修改 A 系统的代码，达到了解耦的效果。

异步处理：互联网类企业对用户的直接操作，一般要求每个请求在 200ms 以内完成。对于一个系统调用多个系统，不使用 MQ 的情况下，它执行完返回的耗时是调用完所有系统所需时间的总和；使用 MQ 进行优化后，执行的耗时则是执行主系统的耗时加上发送数据到消息队列的耗时，大幅度提升系统性能和用户体验。

流量削峰：MySQL 每秒最高并发请求在 2000 左右，用户访问量高峰期的时候涌入的大量请求，会将 MySQL 打死，然后系统就挂掉，但过了高峰期，请求量可能远低于 2000，这种情况去增加服务器就不值得，如果使用 MQ 的情况，将用户的请求全部放到 MQ 中，让系统去消费用户的请求，不要超过系统所能承受的最大请求数量，保证系统不会再高峰期挂掉，高峰期过后系统还是按照最大请求数量处理完请求。

2. 为什么要用RocketMq?

总的来说，RocketMq具有以下几个优势：

- 吞吐量高：单机吞吐量可达十万级
- 可用性高：分布式架构
- 消息可靠性高：经过参数优化配置，消息可以做到0丢失
- 功能支持完善：MQ功能较为完善，还是分布式的，扩展性好
- 支持10亿级别的消息堆积：不会因为堆积导致性能下降
- 源码是java：方便我们查看源码了解它的每个环节的实现逻辑，并针对不同的业务场景进行扩展
- 可靠性高：天生为金融互联网领域而生，对于要求很高的场景，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况
- 稳定性高：RoketMQ在上可能更值得信赖，这些业务场景在阿里双11已经经历了多次考验

3. 使用 MQ 的缺陷有哪些？

系统可用性降低：以前只要担心系统的问题，现在还要考虑 MQ 挂掉的问题，MQ 挂掉，所关联的系统都会无法提供服务。

系统复杂性变高：要考虑消息丢失、消息重复消费等问题。

一致性问题：多个 MQ 消费系统，部分成功，部分失败，要考虑事务问题。

4. 你了解哪些常用的 MQ？

ActiveMQ：支持万级的吞吐量，较成熟完善；官方更新迭代较少，社区的活跃度不是很高，有消息丢失的情况。

RabbitMQ：延时低，微妙级延时，社区活跃度高，bug 修复及时，而且提供了很友善的后台界面；用 Erlang 语言开发，只熟悉 Java 的无法阅读源码和自行修复 bug。

RocketMQ：阿里维护的消息中间件，可以达到十万级的吞吐量，支持分布式事务。

Kafka：分布式的中间件，最大优点是其吞吐量高，一般运用于大数据系统的实时运算和日志采集的场景，功能简单，可靠性高，扩展性高；缺点是可能导致重复消费。

5. MQ 有哪些使用场景？

异步处理：用户注册后，发送注册邮件和注册短信。用户注册完成后，提交任务到 MQ，发送模块并行获取 MQ 中的任务。

系统解耦：比如用注册完成，再加一个发送微信通知。只需要新增发送微信消息模块，从 MQ 中读取任务，发送消息即可。无需改动注册模块的代码，这样注册模块与发送模块通过 MQ 解耦。

流量削峰：秒杀和抢购等场景经常使用 MQ 进行流量削峰。活动开始时流量暴增，用户的请求写入 MQ，超过 MQ 最大长度丢弃请求，业务系统接收 MQ 中的消息进行处理，达到流量削峰、保证系统可用性的目的。

日志处理: 日志采集方收集日志写入 kafka 的消息队列中，处理方订阅并消费 kafka 队列中的日志数据。

消息通讯: 点对点或者订阅发布模式，通过消息进行通讯。如微信的消息发送与接收、聊天室等。

6. RabbitMQ特点?

- 1、可靠性**: RabbitMQ使用一些机制来保证可靠性，如持久化、传输确认及发布确认等。
- 2、灵活的路由**: 在消息进入队列之前，通过交换器来路由消息。对于典型的路由功能，RabbitMQ 已经提供了一些内置的交换器来实现。针对更复杂的路由功能，可以将多个交换器绑定在一起，也可以通过插件机制来实现自己的交换器。
- 3、扩展性**: 多个RabbitMQ节点可以组成一个集群，也可以根据实际业务情况动态地扩展 集群中节点。
- 4、高可用性**: 队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列仍然可用。
- 5、多种协议**: RabbitMQ除了原生支持AMQP协议，还支持STOMP， MQTT等多种消息中间件协议。
- 6、多语言客户端**: RabbitMQ 几乎支持所有常用语言，比如 Java、Python、Ruby、PHP、C#、JavaScript 等。
- 7、管理界面**: RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息、集群中的节点等。
- 8、插件机制**: RabbitMQ 提供了许多插件，以实现从多方面进行扩展，当然也可以编写自己的插件。

7. AMQP是什么?

RabbitMQ就是 AMQP 协议的 Erlang 的实现(当然 RabbitMQ 还支持 STOMP2、MQTT3 等协议) AMQP 的模型架构 和 RabbitMQ 的模型架构是一样的，生产者将消息发送给交换器，交换器和队列绑定。

RabbitMQ 中的交换器、交换器类型、队列、绑定、路由键等都是遵循的 AMQP 协议中相应的概念。目前 RabbitMQ 最新版本默认支持的是 AMQP 0-9-1。

8. AMQP协议3层?

- 1、Module Layer**: 协议最高层，主要定义了一些客户端调用的命令，客户端可以用这些命令实现自己的业务逻辑。
- 2、Session Layer**: 中间层，主要负责客户端命令发送给服务器，再将服务端应答返回客户端，提供可靠性同步机制和错误处理。
- 3、TransportLayer**: 最底层，主要传输二进制数据流，提供帧的处理、信道服用、错误检测和数据表示等。

9. Rocketmq的工作流程是怎样的?

RocketMq的工作流程如下：

- 1、**首先启动NameServer**。NameServer启动后监听端口，等待Broker、Producer以及Consumer连接上来
- 2、**启动Broker**。启动之后，会跟所有的NameServer建立并保持一个长连接，定时发送心跳包。心跳包中包含当前Broker信息(ip、port等)、Topic信息以及Broker与Topic的映射关系
- 3、**创建Topic**。创建时需要指定该Topic要存储在哪些Broker上，也可以在发送消息时自动创建Topic
- 4、**Producer发送消息**。启动时先跟NameServer集群中的其中一台建立长连接，并从NameServer中获取当前发送的Topic所在的Broker；然后从队列列表中轮询选择一个队列，与队列所在的Broker建立长连接，进行消息的发送
- 5、**Consumer消费消息**。跟其中一台NameServer建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，进行消息的消费

10. Rocketmq如何保证高可用性?

1、**集群化部署NameServer**。Broker集群会将所有的broker基本信息、topic信息以及两者之间的映射关系，轮询存储在每个NameServer中（也就是说每个NameServer存储的信息完全一样）。因此，NameServer集群化，不会因为其中的一两台服务器挂掉，而影响整个架构的消息发送与接收；

2、**集群化部署多broker**。producer发送消息到broker的master，若当前的master挂掉，则会自动切换到其他的master

consumer默认会访问broker的master节点获取消息，那么master节点挂了之后，该怎么办呢？它就会自动切换到同一个broker组的slave节点进行消费

那么你肯定会想到会有这样一个问题：consumer要是直接消费slave节点，那master在宕机前没有来得及把消息同步到slave节点，那这个时候，不就会出现消费者取不到消息的情况了？

这样，就引出了下一个措施，来保证消息的高可用性

3、设置同步复制

前面已经提到，消息发送到broker的master节点上，master需要将消息复制到slave节点上，rocketmq提供两种复制方式：同步复制和异步复制

异步复制，就是消息发送到master节点，只要master写成功，就直接向客户端返回成功，后续再异步写入slave节点

同步复制，就是等master和slave都成功写入内存之后，才会向客户端返回成功

那么，要保证高可用性，就需要将复制方式配置成同步复制，这样即使master节点挂了，slave上也有当前master的所有备份数据，那么不仅保证消费者消费到的消息是完整的，并且当master节点恢复之后，也容易恢复消息数据

在master的配置文件中直接配置brokerRole：SYNC_MASTER即可

11. RocketMq如何负载均衡？

1、producer发送消息的负载均衡：默认会轮询向Topic的所有queue发送消息，以达到消息平均落到不同的queue上；而由于queue可以落在不同的broker上，就可以发到不同broker上（当然也可以指定发送到某个特定的queue上）

2、consumer订阅消息的负载均衡：假设有5个队列，两个消费者，则第一个消费者消费3个队列，第二个则消费2个队列，以达到平均消费的效果。而需要注意的是，当consumer的数量大于队列的数量的话，根据rocketMq的机制，多出来的队列不会去消费数据，因此建议consumer的数量小于或者等于queue的数量，避免不必要的浪费

12. RocketMq的存储机制了解吗？

RocketMq采用文件系统进行消息的存储，相对于ActiveMq采用关系型数据库进行存储的方式就更直接，性能更高了

RocketMq与Kafka在写消息与发送消息上，继续沿用了Kafka的这两个方面：顺序写和零拷贝

1、顺序写

我们知道，操作系统每次从磁盘读写数据的时候，都需要找到数据在磁盘上的地址，再进行读写。而如果是机械硬盘，寻址需要的时间往往会长

而一般来说，如果把数据存储在内存上面，少了寻址的过程，性能会好很多；但Kafka的数据存储在磁盘上面，依然性能很好，这是为什么呢？

这是因为，Kafka采用的是顺序写，直接追加数据到末尾。实际上，磁盘顺序写的性能极高，在磁盘个数一定，转数一定的情况下，基本和内存速度一致

因此，磁盘的顺序写这一机制，极大地保证了Kafka本身的性能

2、零拷贝

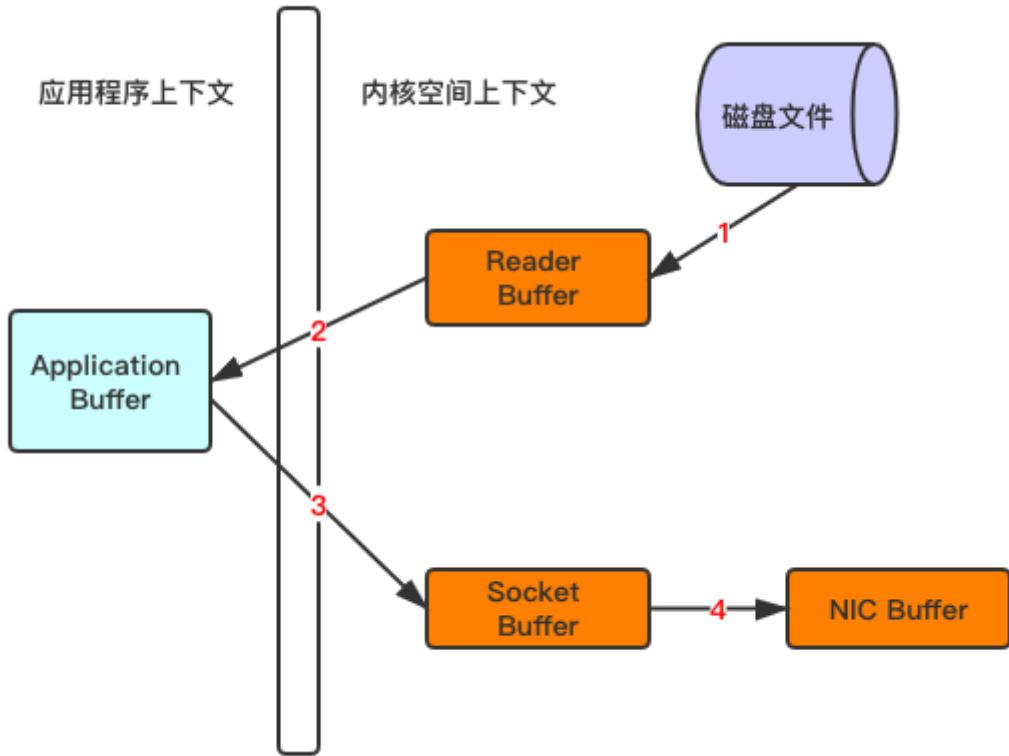
比如：读取文件，再用socket发送出去这一过程

```
buffer = File.read  
Socket.send(buffer)
```

传统方式实现：

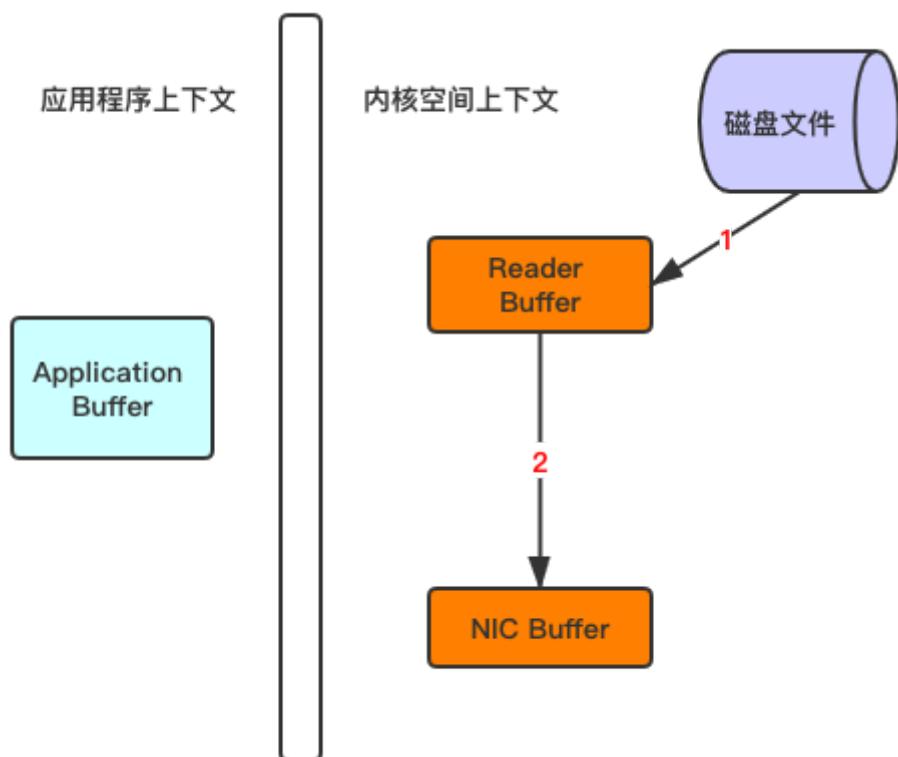
先读取、再发送，实际会经过以下四次复制

- 1、将磁盘文件，读取到操作系统内核缓冲区**Read Buffer**
- 2、将内核缓冲区的数据，复制到应用程序缓冲区**Application Buffer**
- 3、将应用程序缓冲区**Application Buffer**中的数据，复制到socket网络发送缓冲区
- 4、将**Socket buffer**的数据，复制到**网卡**，由网卡进行网络传输



传统方式，读取磁盘文件并进行网络发送，经过的四次数据copy是非常繁琐的

重新思考传统IO方式，会注意到在读取磁盘文件后，**不需要做其他处理，直接用网络发送出去**这种场景下，第二次和第三次数据的复制过程，不仅没有任何帮助，反而带来了巨大的开销。那么这里使用了**零拷贝**，也就是说，直接由内核缓冲区**Read Buffer**将数据复制到**网卡**，省去第二步和第三步的复制。



那么采用零拷贝的方式发送消息，必定会大大减少读取的开销，使得RocketMq读取消息的性能有一个质的提升

此外，还需要再提一点，零拷贝技术采用了MappedByteBuffer内存映射技术，采用这种技术有一些限制，其中有一条就是传输的文件不能超过2G，这也就是为什么RocketMq的存储消息的文件CommitLog

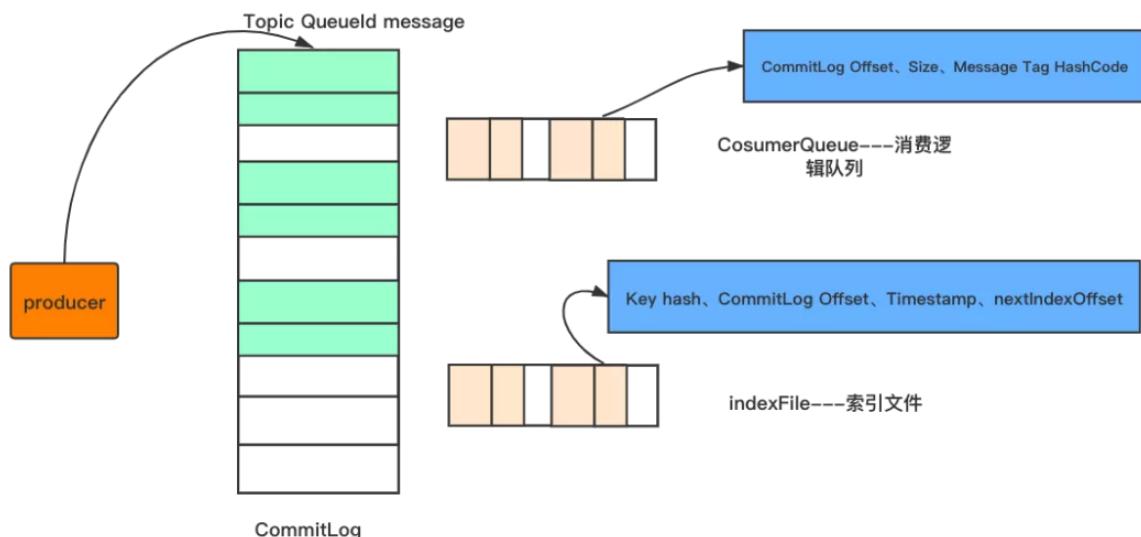
微信搜索公众号：Java专栏，获取最新面试手册

的大小规定为1G的原因

小结：RocketMq采用文件系统存储消息，并采用顺序写写入消息，使用零拷贝发送消息，极大地保证了RocketMq的性能

13. RocketMq的存储结构是怎样的？

如图所示，消息生产者发送消息到broker，都是会按照顺序存储在CommitLog文件中，每个commitLog文件的大小为1G



CommitLog-存储所有的消息元数据，包括Topic、Queueld以及message

ConsumerQueue-消费逻辑队列：存储消息在CommitLog的offset

IndexFile-索引文件：存储消息的key和时间戳等信息，使得RocketMq可以采用key和时间区间来查询消息

也就是说，rocketMq将消息均存储在CommitLog中，并分别提供了ConsumerQueue和IndexFile两个索引，来快速检索消息

14. RocketMq性能比较高的原因？

RocketMq采用文件系统存储消息，采用顺序写的方式写入消息，使用零拷贝发送消息，这三者的结合极大地保证了RocketMq的性能

15. AMQP模型的几大组件?

- 交换器(Exchange): 消息代理服务器中用于把消息路由到队列的组件。
- 队列(Queue): 用来存储消息的数据结构, 位于硬盘或内存中。
- 绑定(Binding): 一套规则, 告知交换器消息应该将消息投递给哪个队列。

16. 生产者Producer?

消息生产者, 就是投递消息的一方。

消息一般包含两个部分: 消息体(payload)和标签(Label)。

17. 消费者Consumer?

消费消息, 也就是接收消息的一方。

消费者连接到RabbitMQ服务器, 并订阅到队列上。消费消息时只消费消息体, 丢弃标签。

18. Broker服务节点?

Broker可以看做RabbitMQ的服务节点。一般请下一个Broker可以看做一个RabbitMQ服务器。

19. Queue队列?

Queue: RabbitMQ的内部对象, 用于存储消息。多个消费者可以订阅同一队列, 这时队列中的消息会被平摊(轮询)给多个消费者进行处理。

20. Exchange交换器?

Exchange: 生产者将消息发送到交换器, 有交换器将消息路由到一个或者多个队列中。当路由不到时, 或返回给生产者或直接丢弃。

21. RoutingKey路由键?

生产者将消息发送给交换器的时候, 会指定一个RoutingKey, 用来指定这个消息的路由规则, 这个RoutingKey需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。

22. Binding绑定?

通过绑定将交换器和队列关联起来, 一般会指定一个BindingKey, 这样RabbitMq就知道如何正确路由消息到队列了。

23. 交换器4种类型?

主要有以下4种。

fanout: 把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中。

direct: 把消息路由到BindingKey和RoutingKey完全匹配的队列中。

topic: 匹配规则：

RoutingKey 为一个 点号'.' 分隔的字符串。比如: java.xiaoka.show

BindingKey和RoutingKey一样也是点号"."分隔的字符串。

BindingKey可使用 * 和 # 用于做模糊匹配，*匹配一个单词，#匹配多个或者0个

headers: 不依赖路由键匹配规则路由消息。是根据发送消息内容中的headers属性进行匹配。性能差，基本用不到。

24. 如何保证消息不被重复消费?

消息被重复消费，就是消费方多次接受到了同一条消息。根本原因就是，第一次消费完之后，消费方给 MQ 确认已消费的反馈，MQ 没有成功接受。比如网络原因、MQ 重启等。

所以 MQ 是无法保证消息不被重复消费的，只能业务系统层面考虑。

不被重复消费的问题，就被转化为消息消费的幂等性的问题。幂等性就是指一次和多次请求的结果一致，多次请求不会产生副作用。

保证消息消费的幂等性可以考虑下面的方式：

- 给消息生成全局 id，消费成功过的消息可以直接丢弃
- 消息中保存业务数据的主键字段，结合业务系统需求场景进行处理，避免多次插入、是否可以根据主键多次更新而并不影响结果等

25. 如何保证消息不丢失?

生产者丢失消息：如网络传输中丢失消息、MQ 发生异常未成功接收消息等情况。**解决办法：**主流的 MQ 都有确认或事务机制，可以保证生产者将消息送达到 MQ。如 RabbitMQ 就有事务模式和 confirm 模式。

MQ 丢失消息：MQ 成功接收消息内部处理出错、宕机等情况。**解决办法：**开启 MQ 的持久化配置。

消费者丢失消息：采用消息自动确认模式，消费者取到消息未处理挂掉了。**解决办法：**改为手动确认模式，消费者成功消费消息再确认。

26. 如何保证消息的顺序性?

生产者保证消息入队的顺序。

MQ 本身是一种先进先出的数据接口，将同一类消息，发到同一个 queue 中，保证出队是有序的。

避免多消费者并发消费同一个 queue 中的消息。

27. 消息大量积压怎么解决?

消息的积压来自于两方面：要么发送快了，要么消费变慢了。

单位时间发送的消息增多，比如赶上大促或者抢购，短时间内不太可能优化消费端的代码来提升消费性能，唯一的办法是通过扩容消费端的实例数来提升总体的消费能力。严重影响 QM 甚至整个系统时，可以考虑临时启用多个消费者，并发接受消息，持久化之后再单独处理，或者直接丢弃消息，回头让生产者重新生产。

如果短时间内没有服务器资源扩容，没办法的办法是将系统降级，通过关闭某些不重要的业务，减少发送的数据量，最低限度让系统还能正常运转，服务重要业务。

监控发现，产生和消费消息的速度没什么变化，出现消息积压的情况，检查是有消费失败反复消费的情况。

监控发现，消费消息的速度变慢，检查消费实例，日志中是否有大量消费错误、消费线程是否死锁、是否卡在某些资源上。

28. 如何保证MQ的高可用?

ActiveMQ:

Master-Slave 部署方式主从热备，方式包括通过共享存储目录来实现(shared filesystem Master-Slave)、通过共享数据库来实现(shared database Master-Slave)、5.9版本后新特性使用 ZooKeeper 协调选择 master(Replicated LevelDB Store)。

Broker-Cluster 部署方式进行负载均衡。

RabbitMQ:

单机模式与普通集群模式无法满足高可用，镜像集群模式指定多个节点复制 queue 中的消息做到高可用，但消息之间的同步网络性能开销较大。

RocketMQ:

有多 master 多 slave 异步复制模式和多 master 多 slave 同步双写模式支持集群部署模式。

Producer 随机选择 NameServer 集群中的其中一个节点建立长连接，定期从 NameServer 获取 Topic 路由信息，并向提供 Topic 服务的 Broker Master 建立长连接，且定时向 Master 发送心跳，只能将消息发送到 Broker master。

Consumer 同时与提供 Topic 服务的 Master、Slave 建立长连接，从 Master、Slave 订阅消息都可以，订阅规则由 Broker 配置决定。

Kafka:

由多个 broker 组成，每个 broker 是一个节点；topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 存放一部分数据，这样每个 topic 的数据就分散存放在多个机器上的。

replica 副本机制保证每个 partition 的数据同步到其他节点，形成多 replica 副本；所有 replica 副本会选举一个 leader 与 Producer、Consumer 交互，其他 replica 就是 follower；写入消息 leader 会把数据同步到所有 follower，从 leader 读取消息。

每个 partition 的所有 replica 分布在不同的机器上。某个 broker 宕机，它上面的 partition 在其他节点有副本，如果有 partition 的 leader，会进行重新选举 leader。

29. 生产者消息运转？

- 1、Producer先连接到Broker,建立连接Connection,开启一个信道(Channel)。
- 2、Producer声明一个交换器并设置好相关属性。
- 3、Producer声明一个队列并设置好相关属性。
- 4、Producer通过路由键将交换器和队列绑定起来。
- 5、Producer发送消息到Broker,其中包含路由键、交换器等信息。
- 6、相应的交换器根据接收到的路由键查找匹配的队列。
- 7、如果找到，将消息存入对应的队列，如果没有找到，会根据生产者的配置丢弃或者退回给生产者。
- 8、关闭信道。
- 9、管理连接。

30. 消费者接收消息过程？

- 1、Producer先连接到Broker,建立连接Connection,开启一个信道(Channel)。
- 2、向Broker请求消费响应的队列中消息，可能会设置响应的回调函数。
- 3、等待Broker回应并投递相应队列中的消息，接收消息。
- 4、消费者确认收到的消息,ack。
- 5、RabbitMq从队列中删除已经确定的消息。
- 6、关闭信道。
- 7、关闭连接。

31. 交换器无法根据自身类型和路由键找到符合条件队列时，有哪些处理？

mandatory : true 返回消息给生产者。

mandatory: false 直接丢弃。

32. 死信队列?

DLX, 全称为 Dead-Letter-Exchange, 死信交换器, 死信邮箱。当消息在一个队列中变成死信 (dead message) 之后, 它能被重新被发送到另一个交换器中, 这个交换器就是 DLX, 绑定 DLX 的队列就称之为死信队列。

33. 导致的死信的几种原因?

- 消息被拒 (Basic.Reject /Basic.Nack) 且 requeue = false。
- 消息TTL过期。
- 队列满了, 无法再添加。

34. 延迟队列?

存储对应的延迟消息, 指当消息被发送以后, 并不想让消费者立刻拿到消息, 而是等待特定时间后, 消费者才能拿到这个消息进行消费。

35. 优先级队列?

优先级高的队列会先被消费。

可以通过x-max-priority参数来实现。

当消费速度大于生产速度且Broker没有堆积的情况下, 优先级显得没有意义。

36. 事务机制?

RabbitMQ 客户端中与事务机制相关的方法有三个:

- 1、channel.txSelect 用于将当前的信道设置成事务模式。
- 2、channel . txCommit 用于提交事务。
- 3、channel . txRollback 用于事务回滚, 如果在事务提交执行之前由于 RabbitMQ 异常崩溃或者其他原因抛出异常, 通过txRollback来回滚。

37. 发送确认机制?

生产者把信道设置为confirm确认模式, 设置后, 所有再改信道发布的消息都会被指定一个唯一的ID, 一旦消息被投递到所有匹配的队列之后, RabbitMQ就会发送一个确认 (Basic.Ack)给生产者 (包含消息的唯一ID), 这样生产者就知道消息到达对应的目的地了。

38. 消费者获取消息的方式?

- 推
- 拉

39. 消费者某些原因无法处理当前接受的消息如何来拒绝?

- channel .basicNack
- channel .basicReject

40. 消息传输保证层级?

At most once:最多一次。消息可能会丢失，单不会重复传输。

At least once: 最少一次。消息决不会丢失，但可能会重复传输。

Exactly once: 恰好一次，每条消息肯定仅传输一次。

41. vhost?

每一个RabbitMQ服务器都能创建虚拟的消息服务器，也叫虚拟主机(virtual host)，简称vhost。

默认为“/”。

42. 集群中的节点类型?

内存节点: ram,将变更写入内存。

磁盘节点: disc,磁盘写入操作。

RabbitMQ要求最少有一个磁盘节点。

43. 队列结构?

通常由以下两部分组成?

- rabbit_amqqueue_process :负责协议相关的消息处理，即接收生产者发布的消息、向消费者交付消息、处理消息的确认(包括生产端的 confirm 和消费端的 ack) 等。
- backing_queue:是消息存储的具体形式和引擎，并向 rabbit amqqueue process 提供相关的接口以供调用。

44. RabbitMQ中消息可能有的几种状态?

alpha: 消息内容(包括消息体、属性和 headers) 和消息索引都存储在内存中。

beta: 消息内容保存在磁盘中，消息索引保存在内存中。

gamma: 消息内容保存在磁盘中，消息索引在磁盘和内存中都有。

delta: 消息内容和索引都在磁盘中。

45. 它有哪几种部署类型？分别有什么特点？

RocketMQ有4种部署类型

1、单Master

单机模式，即只有一个Broker，如果Broker宕机了，会导致RocketMQ服务不可用，不推荐使用

2、多Master模式

组成一个集群，集群每个节点都是Master节点，配置简单，性能也是最高，某节点宕机重启不会影响RocketMQ服务

缺点：如果某个节点宕机了，会导致该节点存在未被消费的消息在节点恢复之前不能被消费

3、多Master多Slave模式，异步复制

每个Master配置一个Slave，多对Master-Slave，Master与Slave消息采用异步复制方式，主从消息一致只会有毫秒级的延迟

优点是弥补了多Master模式（无slave）下节点宕机后在恢复前不可订阅的问题。在Master宕机后，消费者还可以从Slave节点进行消费。采用异步模式复制，提升了一定的吞吐量。总结一句就是，采用多Master多Slave模式，异步复制模式进行部署，系统将会有较低的延迟和较高的吞吐量

缺点就是如果Master宕机，磁盘损坏的情况下，如果没有及时将消息复制到Slave，会导致有少量消息丢失

4、多Master多Slave模式，同步双写

与多Master多Slave模式，异步复制方式基本一致，唯一不同的是消息复制采用同步方式，只有master和slave都写成功以后，才会向客户端返回成功

优点：数据与服务都无单点，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高

缺点就是会降低消息写入的效率，并影响系统的吞吐量

实际部署中，一般会根据业务场景的所需要的性能和消息可靠性等方面来选择后两种

Kafka面试题

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

微信搜索公众号：Java专栏，获取最新面试手册

1. 什么是kafka

Kafka是分布式发布-订阅消息系统，它最初是由LinkedIn公司开发的，之后成为Apache项目的一部分，Kafka是一个分布式，可划分的，冗余备份的持久性的日志服务，它主要用于处理流式数据。

2. 为什么要使用 kafka，为什么要使用消息队列

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

解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。

冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。

健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。

异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

3. Kafka中的ISR、 AR又代表什么？ ISR的伸缩又指什么

ISR:In-Sync Replicas 副本同步队列

AR:Assigned Replicas 所有副本

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

4. Kafka中的broker 是干什么的

broker 是消息的代理，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉取指定Topic的消息，然后进行业务处理，broker在中间起到一个代理保存消息的中转站。

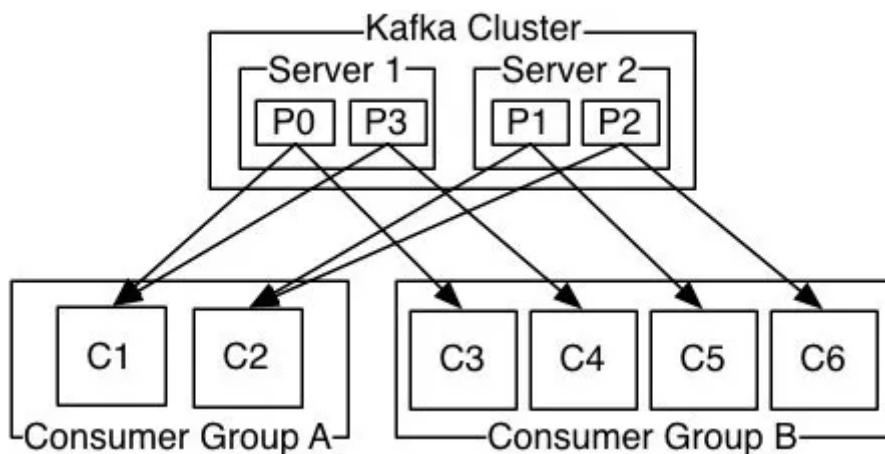
5. 什么是消费者组？

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。

官网上的介绍言简意赅，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。

但实际上，消费者组（Consumer Group）其实包含两个概念，作为队列，消费者组允许你分割数据处理到一组进程集合上（即一个消费者组中可以包含多个消费者进程，他们共同消费该topic的数据），这有助于你的消费能力的动态调整；作为发布-订阅模型（publish-subscribe），Kafka允许你将同一份消息广播到多个消费者组里，以此来丰富多种数据使用场景。

需要注意的是：在消费者组中，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。因此，消费者组在一定程度上也保证了消费者程序的高可用性。



注意：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（Offset），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端产生的消息，我对Producer还是比较熟悉的。”

总之，你总得对consumer group相关的方向有一定理解，然后才能像面试官表名你对某一块很理解。

6. Kafka中的zookeeper起到什么作用，可以不用zookeeper么**

zookeeper是一个分布式的协调组件，早期版本的kafka用zk做meta信息存储，consumer的消费状态，group的管理以及 offset的值。考虑到zk本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了zookeeper的作用。新的consumer使用了kafka内部的group coordination协议，也减少了对zookeeper的依赖，

但是broker依然依赖于ZK，zookeeper在kafka中还用来选举controller 和 检测broker是否存活等等。

7. Kafka为什么那么快

Cache Filesystem Cache PageCache缓存

顺序写 由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。

Zero-copy 零拷技术减少拷贝次数

Batching of Messages 批量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。

Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。

8. 如何设置Kafka能接收的最大消息的大小?

对于SRE来讲，该题简直是送分题啊，但是，最大消息的设置通常情况下有生产者端，消费者端，broker端和topic级别的参数，我们需要正确设置，以保证可以正常的生产和消费。

- Broker端参数： message.max.bytes, max.message.bytes (topic级别) , replica.fetch.max.bytes (否则follow会同步失败)
- Consumer端参数： fetch.message.max.bytes

9. 监控Kafka的框架都有哪些？

对于SRE来讲，依然是送分题。但基础的我们要知道，Kafka本身是提供了JMX (Java Management Extensions) 的，我们可以通过它来获取到Kafka内部的一些基本数据。

- **Kafka Manager**: 更多是Kafka的管理，对于SRE非常友好，也提供了简单的瞬时指标监控。
- **Kafka Monitor**: LinkedIn开源的免费框架，支持对集群进行系统测试，并实时监控测试结果。
- **CruiseControl**: 也是LinkedIn公司开源的监控框架，用于实时监测资源使用率，以及提供常用运维操作等。无UI界面，只提供REST API，可以进行多集群管理。
- **JMX监控**: 由于Kafka提供的监控指标都是基于JMX的，因此，市面上任何能够集成JMX的框架都可以使用，比如Zabbix和Prometheus。
- 已有大数据平台自己的监控体系：像Cloudera提供的CDH这类大数据平台，天然就提供Kafka监控方案。
- **JMXTool**: 社区提供的命令行工具，能够实时监控JMX指标。可以使用kafka-run-class.sh kafka.tools.JmxTool来查看具体的用法。

10. 如何估算Kafka集群的机器数量？

该题也算是SRE的送分题吧，对于SRE来讲，任何生产的系统第一步需要做的就是容量预估以及集群的架构规划，实际上也就是机器数量和所用资源之间的关联关系，资源通常来讲就是CPU，内存，磁盘容量，带宽。但需要注意的是，Kafka因为独有的设计，对于磁盘的要求并不是特别高，普通机械硬盘足够，而通常的瓶颈会出现在带宽上。

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。同时，需要考虑到整个业务Topic数据保存的最大时间，以上几个因素，基本可以预估出来磁盘的容量需求。

需要注意的是：对于磁盘来讲，一定要提前和业务沟通好场景，而不是等待真正有磁盘容量瓶颈了才去扩容磁盘或者找业务方沟通方案。

对于带宽来说，常见的带宽有1Gbps和10Gbps，通常我们需要知道，当带宽占用接近总带宽的90%时，丢包情形就会发生。

11. Kafka能手动删除消息吗？

Kafka不需要用户手动删除消息。它本身提供了留存策略，能够自动删除过期消息。当然，它是支持手动删除消息的。

- 对于设置了Key且参数cleanup.policy=compact的主题而言，我们可以构造一条的消息发送给Broker，依靠Log Cleaner组件提供的功能删除掉该 Key 的消息。

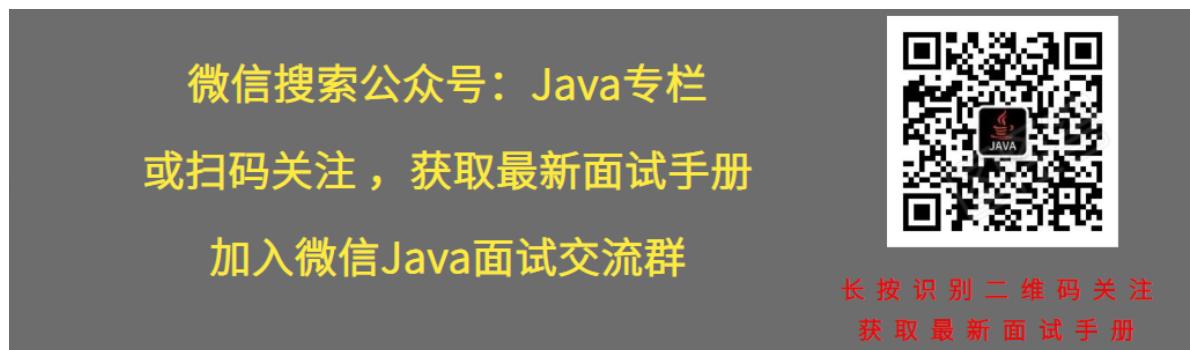
- 对于普通主题而言，我们可以使用kafka-delete-records命令，或编写程序调用Admin.deleteRecords方法来删除消息。这两种方法殊途同归，底层都是调用Admin的deleteRecords方法，通过将分区Log Start Offset值抬高的方式间接删除消息。

12. __consumer_offsets是做什么用的？

这是一个内部主题，主要用于存储消费者的偏移量，以及消费者的元数据信息（消费者实例，消费者id等等）

需要注意的是：Kafka的GroupCoordinator组件提供对该主题完整的管理功能，包括该主题的创建、写入、读取和Leader维护等。

ElasticSearch面试题



1. 简要介绍一下Elasticsearch？

Elasticsearch 是一个分布式、RESTful 风格的搜索和数据分析引擎，能够解决不断涌现出的各种用例。作为 Elastic Stack 的核心，它集中存储您的数据，帮助您发现意料之中以及意料之外的情况。

ElasticSearch 是基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。

核心特点如下：

- 分布式的实时文件存储，每个字段都被索引且可用于搜索。
- 分布式的实时分析搜索引擎，海量数据下近实时秒级响应。
- 简单的restful api，天生的兼容多语言开发。
- 易扩展，处理PB级结构化或非结构化数据。

2. 安装 Elasticsearch 需要依赖什么组件吗？

ES 早期版本需要JDK，在7.X版本后已经集成了 JDK，已无需第三方依赖。

3. 如何启动 Elasticsearch 服务器?

启动方式有很多种，一般 bin 路径下

```
./elasticsearch -d
```

就可以后台启动。

打开浏览器输入 <http://ES> IP:9200 就能知道集群是否启动成功。

如果启动报错，日志里会有详细信息，逐条核对解决就可以。

4. ElasticSearch中的集群、节点、索引、文档、类型是什么？

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

节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。

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

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

类型是索引的逻辑类别/分区，其语义完全取决于用户。

5. Elasticsearch 支持哪些类型的查询？

查询主要分为两种类型：精确匹配、全文检索匹配。

- 精确匹配，例如 term、exists、term set、range、prefix、ids、wildcard、regexp、fuzzy 等。
- 全文检索，例如match、match_phrase、multi_match、match_phrase_prefix、query_string 等

6. 精准匹配检索和全文检索匹配检索的不同？

两者的本质区别：

- 精确匹配用于：是否完全一致？

举例：邮编、身份证号的匹配往往是精准匹配。

- 全文检索用于：是否相关？

举例：类似B站搜索特定关键词如“马保国 视频”往往是模糊匹配，相关的都返回就可以。

7. Elasticsearch中的分片是什么？

在大多数环境中，每个节点都在单独的盒子或虚拟机上运行。

索引 - 在Elasticsearch中，索引是文档的集合。

分片 - 因为Elasticsearch是一个分布式搜索引擎，所以索引通常被分割成分布在多个节点上的被称为分片的元素。

8. 请解释一下 Elasticsearch 中聚合？

聚合有助于从搜索中使用的查询中收集数据，聚合为各种统计指标，便于统计信息或做其他分析。聚合可帮助回答以下问题：

- 我的网站平均加载时间是多少？
- 根据交易量，谁是我最有价值的客户？
- 什么会被视为我网络上的大文件？
- 每个产品类别中有多少个产品？

聚合的分三类：

主要查看7.10 的官方文档，早期是4个分类，别大意啊！

- 分桶 Bucket 聚合

根据字段值，范围或其他条件将文档分组为桶（也称为箱）。

- 指标 Metric 聚合

从字段值计算指标（例如总和或平均值）的指标聚合。

- 管道 Pipeline 聚合

子聚合，从其他聚合（而不是文档或字段）获取输入。

9. 解释一下 Elasticsearch Cluster？

Elasticsearch 集群是一组连接在一起的一个或多个 Elasticsearch 节点实例。

Elasticsearch 集群的功能在于在集群中的所有节点之间分配任务，进行搜索和建立索引。

10. 解释一下 Elasticsearch Node？

节点是 Elasticsearch 的实例。实际业务中，我们会说：ES集群包含3个节点、7个节点。

这里节点实际就是：一个独立的 Elasticsearch 进程，一般将一个节点部署到一台独立的服务器或者虚拟机、容器中。

不同节点根据角色不同，可以划分为：

- 主节点

帮助配置和管理在整个集群中添加和删除节点。

- 数据节点

存储数据并执行诸如CRUD（创建/读取/更新/删除）操作，对数据进行搜索和聚合的操作。

- 客户端节点（或者说：协调节点）将集群请求转发到主节点，将与数据相关的请求转发到数据节点
- 摄取节点

用于在索引之前对文档进行预处理。

11. 解释一下 Elasticsearch 集群中的 Type 的概念？

5.X 以及之前的 2.X、1.X 版本 ES 支持一个索引多个 type 的，举例 ES 6.X 中的 Join 类型在早期版本实际是多 Type 实现的。

在 6.0.0 或更高版本中创建的索引只能包含一个 Mapping 类型。

Type 将在 Elasticsearch 7.0.0 中的 API 中弃用，并在 8.0.0 中完全删除。

12. 解释一下 Elasticsearch 的分片？

当文档数量增加，硬盘容量和处理能力不足时，对客户端请求的响应将延迟。

在这种情况下，将索引数据分成小块的过程称为分片，可改善数据搜索结果的获取。

13. 定义副本、创建副本的好处是什么？

副本是分片的对应副本，用在极端负载条件下提高查询吞吐量或实现高可用性。

所谓高可用主要指：如果某主分片出了问题，对应的副本分片会提升为主分片，保证集群的高可用。

14. Elasticsearch Analyzer 中的字符过滤器如何利用？

字符过滤器将原始文本作为字符流接收，并可以通过添加、删除或更改字符来转换字符流。

字符过滤分类如下：

- HTML Strip Character Filter.

用途：删除 HTML 元素，如，并解码 HTML 实体，如 &。

- Mapping Character Filter

用途：替换指定的字符。

- Pattern Replace Character Filter

用途：基于正则表达式替换指定的字符。

15. REST API 在 Elasticsearch 方面有哪些优势？

REST API 是使用超文本传输协议的系统之间的通信，该协议以 XML 和 JSON 格式传输数据请求。

REST 协议是无状态的，并且与带有服务器和存储数据的用户界面分开，从而增强了用户界面与任何类型平台的可移植性。它还提高了可伸缩性，允许独立实现组件，因此应用程序变得更加灵活。

REST API 与平台和语言无关，只是用于数据交换的语言是 XML 或 JSON。

借助：REST API 查看集群信息或者排查问题都非常方便。

16. Elasticsearch 中常用的 cat 命令有哪些？

面试时说几个核心的就可以，包含但不限于：

含义	命令
别名	GET _cat/aliases?v
分配相关	GET _cat/allocation
计数	GET _cat/count?v
字段数据	GET _cat/fielddata?v
运行状况	GET _cat/health?
索引相关	GET _cat/indices?v
主节点相关	GET _cat/master?v
节点属性	GET _cat/nodeattrs?v
节点	GET _cat/nodes?v
待处理任务	GET _cat/pending_tasks?v
插件	GET _cat/plugins?v
恢复	GET _cat / recovery?v
存储库	GET _cat /repositories?v
段	GET _cat /segments?v
分片	GET _cat/shards?v
快照	GET _cat/snapshots?v
任务	GET _cat/tasks?v
模板	GET _cat/templates?v
线程池	GET _cat/thread_pool?v

17. 你能否列出与 Elasticsearch 有关的主要可用字段数据类型？

- 字符串数据类型，包括支持全文检索的 text 类型 和 精准匹配的 keyword 类型。
- 数值数据类型，例如字节，短整数，长整数，浮点数，双精度数，half_float, scaled_float。
- 日期类型，日期纳秒Date nanoseconds，布尔值，二进制（Base64编码的字符串）等。
- 范围（整数范围 integer_range，长范围 long_range，双精度范围 double_range，浮动范围 float_range，日期范围 date_range）。
- 包含对象的复杂数据类型，nested、Object。
- GEO 地理位置相关类型。

- 特定类型如：数组（数组中的值应具有相同的数据类型）

18. Elasticsearch了解多少，说说你们公司es的集群架构，索引数据大小，分片有多少，以及一些调优手段。

如实结合自己的实践场景回答即可。

比如：ES集群架构13个节点，索引根据通道不同共20+索引，根据日期，每日递增20+，索引：10分片，每日递增1亿+数据，
每个通道每天索引大小控制：150GB之内。

设计阶段调优

- 1、根据业务增量需求，采取基于日期模板创建索引，通过roll over API滚动索引；
- 2、使用别名进行索引管理；
- 3、每天凌晨定时对索引做force_merge操作，以释放空间；
- 4、采取冷热分离机制，热数据存储到SSD，提高检索效率；冷数据定期进行shrink操作，以缩减存储；
- 5、采取curator进行索引的生命周期管理；
- 6、仅针对需要分词的字段，合理的设置分词器；
- 7、Mapping阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。

写入调优

- 1、写入前副本数设置为0；
- 2、写入前关闭refresh_interval设置为-1，禁用刷新机制；
- 3、写入过程中：采取bulk批量写入；
- 4、写入后恢复副本数和刷新间隔；
- 5、尽量使用自动生成的id。

查询调优

- 1、禁用wildcard；
- 2、禁用批量terms（成百上千的场景）；
- 3、充分利用倒排索引机制，能keyword类型尽量keyword；
- 4、数据量大时候，可以先基于时间敲定索引再检索；
- 5、设置合理的路由机制。

其他调优

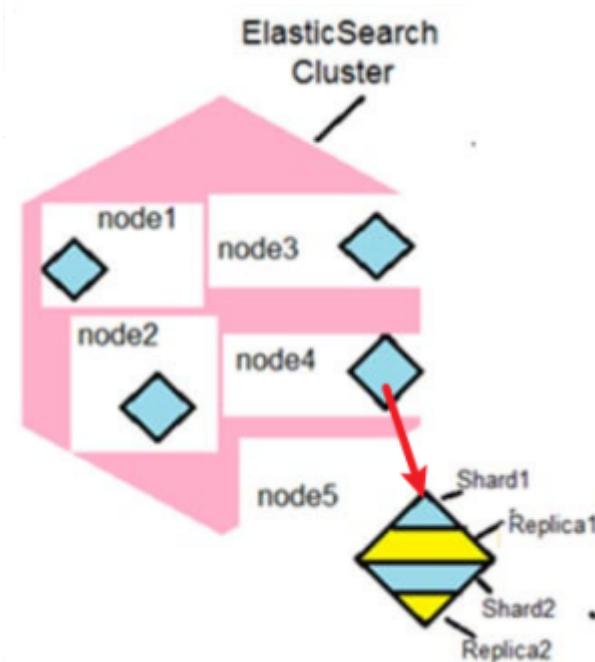
部署调优，业务调优等。

上面的提及一部分，面试者就基本对你之前的实践或者运维经验有所评估了。

19. 解释一下 Elasticsearch 集群中的 索引的概念？

Elasticsearch 集群可以包含多个索引，与关系数据库相比，它们相当于数据库表其他类别概念，如下表所示，点到为止。

Mysql	Elasticsearch
database	index
table	<u>type</u>
row	document
schema	mapping
index	everything is index
select *	GET http://
update *	PUT http://



20. Elasticsearch 索引数据多了怎么办，如何调优，部署

索引数据的规划，应在前期做好规划，正所谓“设计先行，编码在后”，这样才能有效的避免突如其来的数据激增导致集群处理能力不足引发的线上客户检索或者其他业务受到影响。

如何调优，正如问题1所说，这里细化一下：

动态索引层面

基于 模板+时间+rollover api滚动 创建索引，举例：设计阶段定义：blog索引的模板格式为：
blog_index_时间戳的形式，每天递增数据。

这样做的好处：不至于数据量激增导致单个索引数据量非常大，接近于上线2的32次幂-1，索引存储达到了TB+甚至更大。

一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。

存储层面

冷热数据分离存储，热数据（比如最近3天或者一周的数据），其余为冷数据。

对于冷数据不会再写入新数据，可以考虑定期force_merge加shrink压缩操作，节省存储空间和检索效率。

部署层面

一旦之前没有规划，这里就属于应急策略。

结合ES自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

21. 我们可以在 Elasticsearch 中执行搜索的各种可能方式有哪些？

方式一：

基于 DSL 检索（最常用） Elasticsearch提供基于JSON的完整查询DSL来定义查询。

```
GET /shirts/_search
{
  "query": {
    "bool": {
      "filter": [
        { "term": { "color": "red" } },
        { "term": { "brand": "gucci" } }
      ]
    }
  }
}
```

方式二：

基于 URL 检索

```
GET /my_index/_search?q=user:seina
```

方式三：

类SQL 检索

```
POST /_sql?format=txt
{
  "query": "SELECT * FROM uint-2020-08-17 ORDER BY itemid DESC LIMIT 5"
}
```

功能还不完备，不推荐使用。

22. 在 Elasticsearch 中删除索引的语法是什么？

可以使用以下语法删除现有索引：

```
DELETE <index_name>
```

支持通配符删除：

```
DELETE my_*
```

23. 在 Elasticsearch 中列出集群的所有索引的语法是什么？

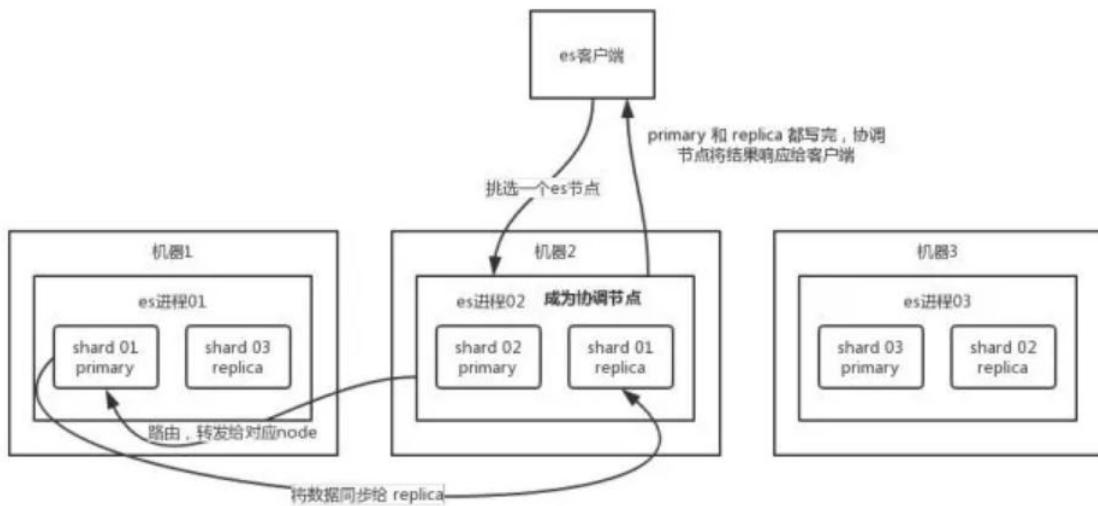
```
GET _cat/indices
```

24. 在索引中更新 Mapping 的语法？

```
PUT test_001/_mapping
{
  "properties": {
    "title": {
      "type": "keyword"
    }
  }
}
```

25. ES 写数据过程

- 客户端选择一个 node 发送请求过去，这个 node 就是 `coordinating node` (协调节点)。
- `coordinating node` 对 `document` 进行路由，将请求转发给对应的 node (有 `primary shard`)。[路由的算法是？]
- 实际的 node 上的 `primary shard` 处理请求，然后将数据同步到 `replica node`。
- `coordinating node` 如果发现 `primary node` 和所有 `replica node` 都搞定之后，就返回响应结果给客户端。



<https://blog.csdn.net/abcd1101>

26. ES 读数据过程

可以通过 `doc id` 来查询，会根据 `doc id` 进行 hash，判断出来当时把 `doc id` 分配到了哪个 `shard` 上面去，从那个 `shard` 去查询。

- 客户端发送请求到任意一个 node，成为 `coordinate node`。
- `coordinate node` 对 `doc id` 进行哈希路由，将请求转发到对应的 node，此时会使用 `round-robin` 随机轮询算法，在 `primary shard` 以及其所有 `replica` 中随机选择一个，让读请求负载均衡。
- 接收请求的 node 返回 `document` 给 `coordinate node`。
- `coordinate node` 返回 `document` 给客户端。

写请求是写入 primary shard，然后同步给所有的 replica shard；读请求可以从 primary shard 或 replica shard 读取，采用的是随机轮询算法。

27. 底层 lucene

简单来说，lucene 就是一个 jar 包，里面包含了封装好的各种建立倒排索引的算法代码。我们用 Java 开发的时候，引入 lucene jar，然后基于 lucene 的 api 去开发就可以了。

通过 lucene，我们可以将已有的数据建立索引，lucene 会在本地磁盘上面，给我们组织索引的数据结构。

28. ES中的倒排索引是什么？

传统的检索方式是通过文章，逐个遍历找到对应关键词的位置。

倒排索引，是通过分词策略，形成了词和文章的映射关系表，也称倒排表，这种词典 + 映射表即为倒排索引。

其中词典中存储词元，倒排表中存储该词元在哪些文中出现的位置。

有了倒排索引，就能实现 $O(1)$ 时间复杂度的效率检索文章了，极大的提高了检索效率。

加分项：

倒排索引的底层实现是基于：FST (Finite State Transducer) 数据结构。

Lucene 从 4+ 版本后开始大量使用的数据结构是 FST。FST 有两个优点：

- 1、空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间；
- 2、查询速度快。 $O(\text{len}(\text{str}))$ 的查询时间复杂度。

29. 请解释在 Elasticsearch 集群中添加或创建索引的过程？

要添加新索引，应使用创建索引 API 选项。创建索引所需的参数是索引的配置Settings，索引中的字段Mapping 以及索引别名 Alias。

也可以通过模板 Template 创建索引。

30. 详细描述一下Elasticsearch索引文档的过程。

协调节点默认使用文档ID参与计算（也支持通过routing），以便为路由提供合适的分片。

`shard = hash(document_id) % (num_of_primary_shards)`

当分片所在的节点接收到来自协调节点的请求后，会将请求写入到Memory Buffer，然后定时（默认是每隔1秒）写入到Filesystem Cache，这个从Memory Buffer到Filesystem Cache的过程叫做refresh；

当然在某些情况下，存在Memory Buffer和Filesystem Cache的数据可能会丢失，ES是通过translog的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到translog中，当Filesystem cache中的数据写入到磁盘中时，才会清除掉，这个过程叫做flush；

在flush过程中，内存中的缓冲将被清除，内容被写入一个新段，段的fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的translog将被删除并开始一个新的translog。

flush触发的时机是定时触发（默认30分钟）或者translog变得太大（默认为512M）时；

31. 详细描述一下Elasticsearch更新和删除文档的过程

删除和更新也都是写操作，但是Elasticsearch中的文档是不可变的，因此不能被删除或者改动以展示其变更；

磁盘上的每个段都有一个相应的.del文件。当删除请求发送后，文档并没有真的被删除，而是在.del文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del文件中被标记为删除的文档将不会被写入新段。

在新的文档被创建时，Elasticsearch会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

32. 详细描述一下Elasticsearch搜索的过程

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；

在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询Filesystem Cache的，但是有部分数据还在Memory Buffer，所以搜索是近实时的。

每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并丰富文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。

补充：Query Then Fetch的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch增加了一个预查询的处理，询问Term和 Document frequency，这个评分更准确，但是性能会变差。

33. Elasticsearch对于大数据量（上亿量级）的聚合如何实现？

Elasticsearch 提供的首个近似聚合是cardinality 度量。它提供一个字段的基数，即该字段的distinct 或者unique值的数目。它是基于HLL算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 = 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。

34. 你可以列出 Elasticsearch 各种类型的分析器吗？

Elasticsearch Analyzer 的类型为内置分析器和自定义分析器。

- Standard Analyzer

标准分析器是默认分词器，如果未指定，则使用该分词器。

它基于Unicode文本分割算法，适用于大多数语言。

- Whitespace Analyzer

基于空格字符切词。

- Stop Analyzer

在simple Analyzer的基础上，移除停用词。

- Keyword Analyzer

不切词，将输入的整个串一起返回。

自定义分词器的模板

自定义分词器的在Mapping的Setting部分设置：

```
PUT my_custom_index
{
  "settings": {
    "analysis": {
      "char_filter": {},
      "tokenizer": {},
      "filter": {},
      "analyzer": {}
    }
  }
}
```

脑海中还是上面的三部分组成的图示。其中：

“char_filter”:{},——对应字符过滤部分；

“tokenizer”:{},——对应文本切分为分词部分；

“filter”:{},——对应分词后再过滤部分；

“analyzer”:{}——对应分词器组成部分，其中会包含：1. 2. 3。

35. ES是如何实现master选举的？

前置条件：

1、只有是候选主节点 (master: true) 的节点才能成为主节点。

2、最小主节点数 (min_master_nodes) 的目的是防止脑裂。

Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping (节点之间通过这个RPC来发现彼此) 和 Unicast (单播模块包含一个主机列表以控制哪些节点需要 ping 通) 这两部分；

获取主节点的核心入口为 findMaster，选择主节点成功返回对应 Master，否则返回 null。

选举流程大致描述如下：

第一步：确认候选主节点数达标，elasticsearch.yml 设置的值

discovery.zen.minimum_master_nodes;

第二步：对所有候选主节点根据nodeId字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第0位）节点，暂且认为它是master节点。

第三步：如果对某个节点的投票数达到一定的值（候选主节点数n/2+1）并且该节点自己也选举自己，那这个节点就是master。否则重新选举一直到满足上述条件。

- 补充：

- 这里的 id 为 string 类型。

- master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data 节点可以关闭 http 功能。

36. Master 节点和 候选 Master 节点有什么区别？

主节点负责集群相关的操作，例如创建或删除索引，跟踪哪些节点是集群的一部分，以及决定将哪些分片分配给哪些节点。

拥有稳定的主节点是衡量集群健康的重要标志。

而候选主节点是被选具备候选资格，可以被选为主节点的那些节点。

37. Elasticsearch中的属性 enabled, index 和 store 的功能是什么？

- **enabled: false**, 启用的设置仅可应用于顶级映射定义和 Object 对象字段, 导致 Elasticsearch 完全跳过对字段内容的解析。

仍然可以从 _source 字段中检索 JSON, 但是无法搜索或以其他任何方式存储 JSON。

如果对非全局或者 Object 类型, 设置 enable : false 会报错如下:

```
"type": "mapper_parsing_exception",
"reason": "Mapping definition for [user_id] has unsupported parameters:
[enabled : false]"
```

- **index: false**, 索引选项控制是否对字段值建立索引。它接受 true 或 false, 默认为 true。未索引的字段不可查询。

如果非要检索, 报错如下:

```
"type": "search_phase_execution_exception",
"reason": "Cannot search on field [user_id] since it is not indexed."
```

- **store:**

某些特殊场景下, 如果你只想检索单个字段或几个字段的值, 而不是整个 _source 的值, 则可以使用源过滤来实现;

这个时候, store 就派上用场了。

1、 enabled	<ul style="list-style-type: none">• true (缺省) false• 仅存储, 不做搜索和聚合分析
2、 index	<ul style="list-style-type: none">• true (缺省) false• 是否构建倒排索引
3、 index_option	<ul style="list-style-type: none">• docs freqs positions offsets• 存储倒排索引的哪些信息
4、 norms	<ul style="list-style-type: none">• true (默认) false• 是否存储归一化相关参数, 如果字段仅用于过滤和聚合分析, 可关闭
5、 doc_values	<ul style="list-style-type: none">• true (缺省) false• 是否启动 doc_values, 用户聚合和排序分析
6、 fielddata	<ul style="list-style-type: none">• true (缺省) false• 是否为 text 类型启动 fielddata, 实现排序和聚合分析
7、 store	<ul style="list-style-type: none">• false (默认) true• 是否存储改字段值
8、 coerce	<ul style="list-style-type: none">• true (缺省) false• 是否开启自动数据类型转换功能, 比如: 字符串转数字, 浮点转整形
9、 multi-fields	<ul style="list-style-type: none">• 灵活的使用多字段解决多样的业务需求
10、 dynamic	<ul style="list-style-type: none">• true (缺省) false strict• 控制 mapping 的自动更新
11、 data_detection	<ul style="list-style-type: none">• true (缺省) false• 是否自动识别日期类型

<https://blog.csdn.net/wujunshishuo987>

38. Elasticsearch中的节点（比如共20个），其中的10个选了一个master，另外10个选了另一个master，怎么办？

当集群 master 候选数量不小于3个时，可以通过设置最少投票通过数量
(`discovery.zen.minimum_master_nodes`) 超过所有候选节点一半以上来解决脑裂问题；
当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data 节点，避免脑裂问题。

39. 如何解决ES集群的脑裂问题

所谓集群脑裂，是指 Elasticsearch 集群中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master 的情况。

当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量
(`discovery.zen.minimum_master_nodes`) 超过所有候选节点一半以上来解决脑裂问题；
当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data 节点，避免脑裂问题。

40. 详细描述一下ES索引文档的过程？

这里的索引文档应该理解为文档写入 ES，创建索引的过程。

第一步：

客户端向集群某节点写入数据，发送请求。（如果没有指定路由/协调节点，请求的节点扮演协调节点的角色。）

第二步：

协调节点接受到请求后，默认使用文档 ID 参与计算（也支持通过 routing），得到该文档属于哪个分片。随后请求会被转到另外的节点。

```
bash# 路由算法：根据文档id或路由计算目标的分片id
shard = hash(document_id) % (num_of_primary_shards)
```

第三步：

当分片所在的节点接收到来自协调节点的请求后，会将请求写入到 Memory Buffer，然后定时（默认是每隔 1 秒）写入到 Filesystem Cache，这个从 Memory Buffer 到 Filesystem Cache 的过程就叫做 refresh；

第四步：

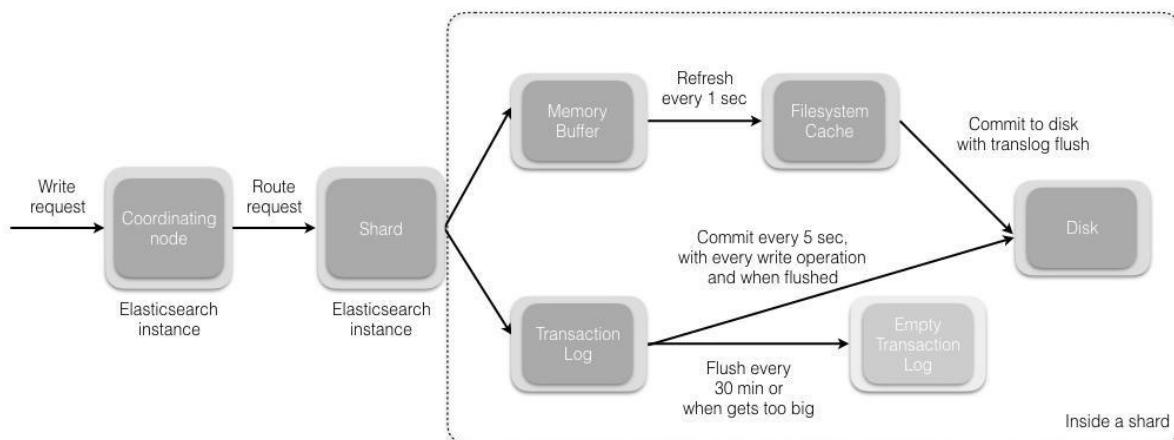
当然在某些情况下，存在 Memory Buffer 和 Filesystem Cache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystem cache 中的数据写入到磁盘中时，才会清除掉，这个过程叫做 flush；

第五步：

在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync 将创建一个新的提交点，并将内容刷新到磁盘，旧的 translog 将被删除并开始一个新的 translog。

第六步：

`flush` 触发的时机是定时触发（默认 30 分钟）或者 `translog` 变得太大（默认为 512 M）时。



• 补充：关于 Lucene 的 Segment

- **Lucene 索引是由多个段组成，段本身是一个功能齐全的倒排索引。**
- **段是不可变的，允许 Lucene 将新的文档增量地添加到索引中，而不用从头重建索引。**
- 对于每一个搜索请求而言，索引中的所有段都会被搜索，并且每个段会消耗 CPU 的时钟周、文件句柄和内存。这意味着段的数量越多，搜索性能会越低。
- 为了解决这个问题，Elasticsearch 会合并小段到一个较大的段，提交新的合并段到磁盘，并删除那些旧的小段。（段合并）

41. 详细描述一下ES更新和删除文档的过程？

删除和更新也都是写操作，但是 Elasticsearch 中的文档是不可变的，因此不能被删除或者改动以展示其变更。

磁盘上的每个段都有一个相应的 `.del` 文件。当删除请求发送后，文档并没有真的被删除，而是在 `.del` 文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在 `.del` 文件中被标记为删除的文档将不会被写入新段。

在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新时，旧版本的文档在 `.del` 文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

42. 详细描述一下ES搜索的过程？

搜索被执行成一个两阶段过程，即 `Query Then Fetch`；

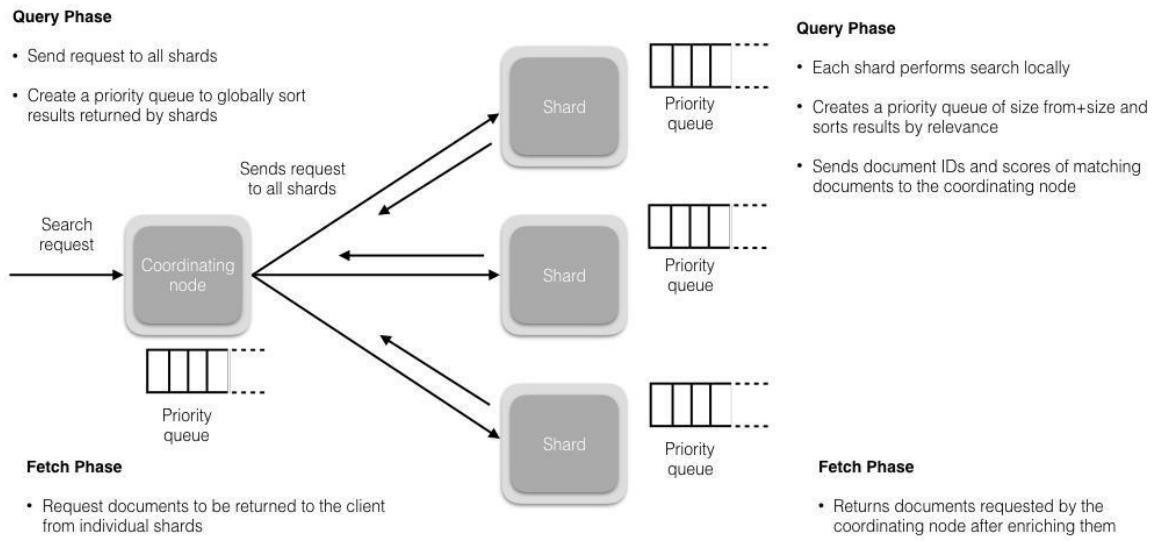
Query阶段：

查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 `from + size` 的优先队列。PS：在搜索的时候是会查询 `Filesystem Cache` 的，但是有部分数据还在 `Memory Buffer`，所以搜索是近实时的。

每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

Fetch阶段：

协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 `GET` 请求。每个分片加载并丰富文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。



43. 在并发情况下，ES如何保证读写一致？

可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；另外对于写操作，一致性级别支持quorum/one/all，默认为quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。

对于读操作，可以设置replication为sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置replication为async时，也可以通过设置搜索请求参数为primary来查询主分片，确保文档是最新版本。

44. ES对于大数据量（上亿量级）的聚合如何实现？

Elasticsearch 提供的首个近似聚合是cardinality 度量。它提供一个字段的基数，即该字段的distinct 或者unique值的数目。它是基于HLL算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 = 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。

45. 对于GC方面，在使用ES时要注意什么？

- 1) 倒排词典的索引需要常驻内存，无法GC，需要监控data node上segment memory增长趋势。
- 2) 各类缓存，field cache, filter cache, indexing cache, bulk queue等等，要设置合理的大小，并且要应该根据最坏的情况来看heap是否够用，也就是各类缓存全部占满的时候，还有heap空间可以分配给其他任务吗？避免采用clear cache等“自欺欺人”的方式来释放内存。
- 3) 避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用scan & scroll api来实现。
- 4) cluster stats驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过tribe node连接。
- 5) 想知道heap够不够，必须结合实际应用场景，并对集群的heap使用情况做持续的监控。

46. 说说你们公司ES的集群架构，索引数据大小，分片有多少，以及一些调优手段？

根据实际情况回答即可，如果是我回答：

我司有多个ES集群，下面列举其中一个。该集群有20个节点，根据数据类型和日期分库，每个索引根据数据量分片，比如日均1亿+数据的，控制单索引大小在200GB以内。

下面重点列举一些调优策略，仅是我做过的，不一定全面，如有其它建议或者补充欢迎留言。

部署层面：

- 1) 最好是64GB内存的物理机器，但实际上32GB和16GB机器用的比较多，但绝对不能少于8G，除非数据量特别少，这点需要和客户方面沟通并合理说服对方。
- 2) 多个内核提供的额外并发远胜过稍微快一点点的时钟频率。
- 3) 尽量使用SSD，因为查询和索引性能将会得到显著提升。
- 4) 避免集群跨越大的地理距离，一般一个集群的所有节点位于一个数据中心中。
- 5) 设置堆内存：节点内存/2，不要超过32GB。一般来说设置`export ES_HEAP_SIZE=32g`环境变量，比直接写`-Xmx32g -Xms32g`更好一点。
- 6) 关闭缓存swap。内存交换到磁盘对服务器性能来说是致命的。如果内存交换到磁盘上，一个100微秒的操作可能变成10毫秒。再想想那么多10微秒的操作时延累加起来。不难看出swapping对于性能是多么可怕。
- 7) 增加文件描述符，设置一个很大的值，如65535。Lucene使用了大量的文件，同时，Elasticsearch在节点和HTTP客户端之间进行通信也使用了大量的套接字。所有这一切都需要足够的文件描述符。
- 8) 不要随意修改垃圾回收器（CMS）和各个线程池的大小。
- 9) 通过设置`gateway.recover_after_nodes`、`gateway.expected_nodes`、`gateway.recover_after_time`可以在集群重启的时候避免过多的分片交换，这可能会让数据恢复从数个小时缩短为几秒钟。

索引层面：

- 1) 使用批量请求并调整其大小：每次批量数据 5-15 MB 大是个不错的起始点。
- 2) 段合并：Elasticsearch默认值是20MB/s，对机械磁盘应该是个不错的设置。如果你用的是SSD，可以考虑提高到100-200MB/s。如果你在做批量导入，完全不在意搜索，你可以彻底关掉合并限流。另外还可以增加`index.translog.flush_threshold_size`设置，从默认的512MB到更大一些的值，比如1GB，这可以在一次清空触发的时候在事务日志里积累出更大的段。
- 3) 如果你的搜索结果不需要近实时的准确度，考虑把每个索引的`index.refresh_interval`改到30s。
- 4) 如果你在做大批量导入，考虑通过设置`index.number_of_replicas: 0`关闭副本。
- 5) 需要大量拉取数据的场景，可以采用`scan & scroll api`来实现，而不是`from/size`一个大范围。

存储层面：

- 1) 基于数据+时间滚动创建索引，每天递增数据。控制单个索引的量，一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。
- 2) 冷热数据分离存储，热数据（比如最近3天或者一周的数据），其余为冷数据。对于冷数据不会再写入新数据，可以考虑定期`force_merge`加`shrink`压缩操作，节省存储空间和检索效率

47. 在并发情况下，Elasticsearch如何保证读写一致？

可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；

另外对于写操作，一致性级别支持`quorum/one/all`，默认为`quorum`，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因

导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。

对于读操作，可以设置`replication`为`sync`（默认），这使得操作在主分片和副本分片都完成后才会返回；如果设置`replication`为`async`时，也可以通过设置搜索请求参数`_preference`为`primary`来查询主分片，确保文档是最新版本。

Linux

Linux常用命令

微信搜索公众号：Java专栏

或扫码关注，获取最新面试手册

加入微信Java面试交流群



长按识别二维码关注
获取最新面试手册

启动网络命令

ip addr 查看网卡信息

```
service network start 启动网卡
service network stop 关闭网卡
service network restart 重启网络
```

pwd命令

pwd命令，查看当前目录的路径

linux下所有的绝对路径都是从根目录"/"开始

root:是linux下root用户的根目录

home:是linux下其他用户的默认根目录（例如：在linux上创建了一个bow用户，那么就会在/home下面生成一个bow目录作为bow用户的根目录）

etc:是linux下系统配置文件目录

tmp:临时文件目录，所有用户都可以用

ls命令

ls [参数] 目录路径

ls 表示查看目录下的文件

```
ls #表示查看当前目录下的文件
ls -l #表示查看当前目录下的详细信息
ls -a #表示查看当前目录下的所有文件(包含隐藏文件)
ls -la #表示查看当前目录下的所有文件(包含隐藏文件)的详细信息
ls -lh #是以适当的单位来显示文件的大小 ls -lh表示查看当前目录下的文件的详细信息，并以合适单位显示文件大小
```

ls -l / #表示查看根目录"/"下文件的详细信息

```
ls /etc #表示查看目录/etc下的文件  
ls --help #查看命令的帮助文档  
--help参数: 所有linux上的命令都有, 但写法上有如下几种:  
(1)--help  
(2)--h  
(3)-help  
(4)-h  
ll命令: 它和ls -l命令功能相同, 但是不是所有的linux上都默认安装
```

cd命令

cd 目录路径 #进入一个目录, 目录路径可以是绝对路径(以/开始的路径都是绝对路径), 也可以是相对路径
相对路径: 以非/开始的路径,
注意: "."表示当前目录
"..."表示当前目录的上一级目录, 它可以多个一起使用
"~"表示当前用户的根目录 例如: root用户时, ~表示/root目录 bow用户时, ~表示/home/bow目录

```
cd / #表示进入系统根目录  
cd usr/ #表示进入当前目录下的usr目录  
cd local/ #表示进入当前目录下的local目录  
cd ./bin #表示进入当前目录下的bin目录  
cd .. #表示进入当前目录的上一级目录  
cd ../../ #表示进入当前目录的上级目录的上一级目录  
cd /usr/local/bin #进入/usr/local/bin目录  
cd ../etc #表示进入和当前目录同级的etc目录 #..表示当前目录的上一级目录 ../etc表示当前目录上级目录下的etc目录 (和当前目录同级)  
cd ~ #表示进入当前用户的根目录 (cd ~ 和直接执行cd后不加目录的效果相同)  
#例如: root用户进入/root目录, bow用户进入/home/bow目录  
  
cd ~/data #表示进入当前用户根目录下的data目录 例如: root用户则进入了/root/data目录
```

mkdir命令

mkdir 目录路径 #创建一个目录, 目录路径可以是绝对路径也可以是相对路径

```
mkdir data #在当前目录下创建一个data目录  
mkdir ./dir #在当前目录下创建一个dir目录  
mkdir /root/tmp #在/root目录下创建一个tmp目录
```

mkdir创建目录时, 只有在目录的上级目录存在时, 才会创建

mkdir -p 目录#创建目录时, 如果没有父目录, 会创建父目录, 递归地创建目录
mkdir -p a/b/c #在当前目录下创建3级目录

rm命令

```
rm [参数] 目录路径 #删除目录命令, rm默认只能删除空目录
```

```
rm ./dir #删除当前目录下的dir目录
```

```
rm -p 目录路径 #表示删除目录和它的父目录 (目录要是一个空目录)
```

```
rm -p a/b/c #删除当前目录下的a/b/c目录
```

touch命令

touch 命令 #创建文件命令

touch 文件路径

```
touch 1.txt #在当前目录下创建一个1.txt文件
```

```
touch /root/2.txt #在/root目录下创建一个2.txt文件
```

rm命令

rm [参数] 路径 #删除命令

rm 1.txt #删除当前目录下的1.txt文件, 删除时会提示, 是否删除如果输入y表示删除, 输入n表示不删除

rm -f /root/2.txt #-f表示强制删除, 不会提示, 强制删除/root目录下的2.txt

```
rm -r a/ #递归的删除当前目录下a目录下的所有内容
```

```
[root@bow ~]# rm -r a/
```

```
rm: 是否进入目录"a/"? y
```

```
rm: 是否进入目录"a/b"? y
```

```
rm: 是否进入目录"a/b/c"? y
```

```
rm: 是否删除普通空文件 "a/b/c/3.txt"? y
```

```
rm: 是否删除目录 "a/b/c"? y
```

```
rm: 是否删除普通空文件 "a/b/2.txt"? y
```

```
rm: 是否删除目录 "a/b"? y
```

```
rm: 是否删除普通空文件 "a/1.txt"? y
```

```
rm: 是否删除目录 "a/"? y
```

```
rm -rf a/ #强制删除当前目录下a目录及a目录下的所有内容
```

```
rm -rf * #删除当前目录下的所有内容
```

```
rm -rf a/* #删除当前目录下a目录下的所有内容
```

```
rm -rf *.txt #删除当前目录下的所有txt文件
```

```
rm -rf *s* #删除当前目录下所有名字中包含s的文件或文件夹
```

echo命令

echo #输出命令，可以输入变量，字符串的值

```
echo Hello World #打印Hello World  
echo $PATH #打印环境变量PATH的值, 其中$是取变量值的符号, 用法: $变量名 或者 ${变量名}  
  
echo -n #打印内容但不换行  
echo -n Hello world
```

>和>>命令

和>>:输出符号，将内容输出到文件中，>表示覆盖(会删除原文件内容) >>表示追加

```
echo Hello World > 1.txt #将Hello World输出到当前目录下的1.txt文件  
#如果当前目录下没有1.txt文件会创建一个新文件,  
#如果当前目录下有1.txt, 则会删除原文件内容, 写入Hello World  
echo 1234 >> 1.txt #将1234追加到当前目录下的1.txt中, 如果文件不存在会创建新文件
```

通过>和>>都可以创建文件

文件查看命令

cat 文件路径 #查看文件的所有内容

```
cat 1.txt #查看当前目录下1.txt的内容  
cat /root/1.txt #查看/root目录下的1.txt文件内容
```

more 文件路径 #分页查看文件内容

more linux常用命令.txt #分页查看当前目录下linux常用命令.txt文件的内容
#按空格或回车, 会继续加载文件内容, 按q退出查看
#当加载到文件末尾时, 会自动退出查看

less 文件路径 #分页查看文件内容

less linux常用命令.txt #分页查看文件内容, 按空格继续加载文件, 按q退出查看, 不会自动退出查看

head [参数] 文件路径 #从文件开始查看文件

```
head linux常用命令.txt #查看文件的前10行内容  
  
head -n 文件路径 # n是一个正整数, 表示查看文件的前n行数据  
head -20 linux常用命令.txt #查看文件的前20行内容
```

tail [参数] 文件路径 #从文件的末尾查看文件内容

tail linux常用命令.txt #查看文件的后10行内容

```
tail -n 文件路径 # n是一个正整数, 表示查看文件的后n行数据  
tail -15 linux常用命令.txt #查看文件后15行内容
```

```
tail -f 文件路径 #动态的查看文件的最后几行内容(查看文件时, 等待文件更新, 如果文件更新了, 会显示出新的内容)
```

tail -f 1.txt #查看文件1.txt的最新内容, tail -f 一般用来查看日志文件 按CTRL+C或才CTRL+Z退出查看

```
CTRL+C: 表示暂停进程  
CTRL+Z: 表示停止进程
```

文件编辑命令

vi/vim命令：这两个命令在使用上几乎完全一样（个人喜欢vim命令）

安装vim命令：（安装是需要网络的）

```
yum -y install vim
```

yum命令是centos和red hat系统上使用官方资源包去安装软件的命令

```
yum -y install 软件名  
yum -y remove 软件名
```

查看虚拟机能不能上外网：

```
ping www.baidu.com  
CTRL+C或者CTRL+Z退出
```

vim命令总体分为两类

vim 文件路径 --进入非编辑模式

非编辑模式命令：

```
yy: 复制光标当前行  
p: 粘贴  
dd: 删除光标当前行  
$: 光标跳到当前行的行尾  
^: 光标跳到当前行的行首
```

```
:s/原字符串/新字符串/:替换光标当前行内容  
:%s/原字符串/新字符串/g:全文替换 #g表示global i表示ignore忽略大小写
```

```
/要查找的内容:从光标当前行向后查找内容  
/d #在文件中查找d字母  
?要查找的内容: 从光标当前位置向前查找内容  
?d #查找文件中的d字母  
CTRL+F: 向下翻1页  
CTRL+B: 向上翻1页
```

```
:set nu: 显示文件的行号
```

```
:set nonu: 去掉行号显示  
u:撤消  
  
**:set ff :显示文件的格式 #unix表示在unix上的文件 dos表示文件是windows上的文件**  
:w : 表示保存文件  
:q :表示退出vim命令  
:wq:保存并退出  
:w! :强制保存  
:q!:强制退出但不保存  
:wq!:强制保存并退出  
i:表示进入编辑模式，并且光标在当前行  
o: 表示进入编辑模式，并且光标出现的当前行的下一行(新行)
```

编辑模式命令：

编辑模式下可以通过方向键控制光标的位置，并且可以输入文件到光标当前位置

```
ESC:退出编辑模式
```

cp命令

cp 拷贝命令

cp [参数] 原文件路径 目标文件路径

```
cp 1.txt a/ #将1.txt文件拷贝到a目录下  
cp 1.txt 2.txt #将1.txt拷贝到2.txt  
cp -r a data # -r参数表示将目录和目录下的文件一起拷贝，将a目录拷贝到data目录
```

scp命令

scp 远程拷贝命令，它可以将本地文件拷贝到远程服务器，也可以将远程服务器的文件拷贝到本地，也可以将一台服务器文件拷贝到另一台

```
scp -r 本地文件路径 用户名@ip[:port]:远程路径 #将本地文件拷贝到远程服务器  
scp -r 2.txt root@192.168.5.105:/root/data/ #将本地的2.txt拷贝到192.168.5.105的/root/data目录下  
  
scp -r 用户名@ip[:port]:远程文件路径 本地路径 #将远程文件拷贝到本地  
scp -r root@192.168.5.105:/root/3.bak /root/data #将远程的/root/3.bak文件拷贝到本地的/root/data目录
```

scp -r 用户名@ip[:port]:远程文件路径 用户名@ip[:port]:远程文件路径 #将文件从一台服务器拷贝到另一台服务器

```
scp -r root@192.168.5.105:/root/tmp root@192.168.5.105:/root/data/ #  
将/root/tmp拷贝到远程的/root/data目录下
```

mv命令

mv 移动命令,它可以移动文件,也可以给文件改名

mv 原文件路径 目标文件路径 #将文件从一个地方拷贝到另一个地方

```
mv 1.txt 12.txt #将文件1.txt改名为12.txt  
mv tmp tmp #将tmp目录改名为tm  
mv 12.txt tm #将文件12.txt移动到tm目录下
```

man命令

man 命令,查看命令的命令,查看命令帮助文档(显示的信息最详细)

```
man mv #查看mv命令的文件  
  
man命令和命令的 --help参数结果相似(man命令只适用于linux本身的命令)
```

free命令

free命令,它是用来查看系统内存的命令

```
free #查看系统内存使用情况  
free -h #查看内存使用情况,并且以合适的单位显示大小
```

df命令

df命令,它是查看系统硬盘的命令

```
df #查看系统硬盘使用情况  
df -h #查看硬盘使用,并以合适单位显示大小
```

wc命令

wc 命令,word count的缩写,它是查看文件的单词个数

wc [参数] 文件

```
wc -l linux常用命令.txt #-l表示line行数 计算文件的行数  
wc -w linux常用命令.txt #-w表示word单词个数 计算文件的单词个数
```

ps命令

ps命令,它是查看系统进程的命令

ps -aux

ps -ef

jps 查看java进程

kill命令

kill 进程id #结束进程

```
root 21752 1.6 0.5 158800 5532 ? Ss 08:34 0:00 sshd: root@pts/0
```

kill 21752 #结束ssh登陆的进程

kill -9 进程id #强制结束进程

用户和权限命令

创建用户组:

groupadd 用户组名称 #创建一个用户组

```
groupadd bows #创建一个叫bows的用户组
```

删除用户组:

```
groupdel 用户组名称 #删除一个用户组(删除时必须是用户组下没有用户时)
```

```
groupdel bows #删除用户组
```

创建用户:

useradd 用户名 [-g 用户组名 -G 用户组名] #创建一个用户,-g指定用户的主用户组,-G指定用户的其他用户组

```
useradd bow -g bows #创建bow用户,并指定它的主用户组是bows
```

```
id 用户名 #查看用户的id
```

```
id bow #查看用户bow的id
```

删除用户:

```
userdel 用户名 #删除用户
```

```
userdel bow #删除用户bow
```

切换用户:

```
su 用户名 #切换用户,但不加载用户的环境变量
```

```
su - 用户名 #切换用户,并加载用户的环境变量(建议使用这种方式切换用户)
```

```
su bow #切换到bow用户(root用户切换到其他用户是不需要输入密码的,其他用户切换到root用户是要输入root用户密码的,其他用户之间的切换也是需要密码)
```

```
exit #退出当前用户的登陆
```

修改用户密码:

```
passwd 用户名 #修改用户密码  
passwd bow #修改bow用户的密码
```

权限:

文件类型	用户权限	用户组权限	其他用户权限	
-	rw-	r--	r--	. 1 root
root 5890 3月 23 14:11 linux常用命令.txt				
d	rwx	r-x	r-x	. 4 root
root 81 3月 24 08:06 data				

d表示文件夹 u表示用户权限 g表示用户组权限 o表示其他用户权限

r:表示读权限 数字表示为4

w:表示写权限 数字表示为2

x:表示执行权限 数字表示为1

-:表示没有权限

chmod 赋权限命令

```
chmod 权限 文件路径  
-rw-r--r--. 1 root root 31 3月 24 07:46 2.txt  
chmod u+x 2.txt #给用户加上执行权限  
-rwxr--r--. 1 root root 31 3月 24 07:46 2.txt  
chmod g+w 2.txt #给用户组加写权限  
-rwxrw-r--. 1 root root 31 3月 24 07:46 2.txt  
chmod o+x 2.txt #给其他用户加执行权限  
-rwxrw-r-x. 1 root root 31 3月 24 07:46 2.txt  
chmod g-w 2.txt #去掉用户的写权限  
-rwxr--r-x. 1 root root 31 3月 24 07:46 2.txt
```

用3个数字来设置文件或目录的权限,第1个数字表示用户权限,第2数字表示用户组权限,第3个数字表示其他用户权限

```
chmod 755 2.txt #设置用户的权限为rwx,用户组的权限r-x,其他用户的权限r-x  
-rwxr-xr-x. 1 root root 31 3月 24 07:46 2.txt  
chmod 766 2.txt #设置用户权限为rwx,用户组权限rw-,其他用户的权限rw-  
-rwxrw-rw-. 1 root root 31 3月 24 07:46 2.txt
```

设置目录权限时,要使用-R参数,保证目录下的所有文件和目录的权限相同

```
drwxr-xr-x. 4 root root 81 3月 24 08:06 data  
chmod -R 777 data #将data目录以及它下面的所有文件的权限设置为rwxrwxrwx  
drwxrwxrwx. 4 root root 81 3月 24 08:06 data
```

chown 命令,它是更改文件所属用户

```
chown -R 用户[:用户组] 目录或文件
-rwxrw-rw-. 1 root root 31 3月 24 07:46 2.txt
chown bow 2.txt #将2.txt的所属用户改为bow
-rwxrw-rw-. 1 bow root 31 3月 24 07:46 2.txt
chown bow:bows 2.txt #将2.txt所属的用户改为bow, 用户组改为bows
-rwxrw-rw-. 1 bow bows 31 3月 24 07:46 2.txt
drwxr--r--. 4 root root 81 3月 24 08:06 data
chown -R bow:bows data #将data目录及它子目录文件的所属用户改为bow, 用户组改为bows
drwxr--r--. 4 bow bows 81 3月 24 08:06 data
```

查找命令

find命令,可以根据文件的时间,名称等查找文件

```
find *.txt #查找txt文件
```

grep 命令,查找内容

```
grep cat linux常用命令.txt #在linux常用命令.txt文件中查询包含cat的行,查找文件内容
```

| 通道符号,连接两个命令的,将前一个命令的查询结果传给后一个命令

```
ps -ef | grep sshd #查看系统中sshd的进程
ps -ef | grep java #查看所有java进程
grep -v # -v参数表示查询不包含查找条件的行
grep -v cat linux常用命令.txt #查找linux常用命令.txt中不包含cat的行

ps -ef | grep sshd | grep -v grep #查询sshd的进程,不包括grep的行
```

-了解性查询命令

```
who命令 #查询系统中的用户(登陆的用户)
whoami命令 #查看系统当前用户名
whereis命令 #查看系统安装的某个软件的路径
whereis python #查看python的安装路径
which 命令 #查找软件的可执行文件路径
which python #查看python可执行文件路径
```

压缩命令

安装zip和unzip命令:

```
yum -y install zip unzip
```

zip压缩命令

zip 压缩文件名 要压缩的文件路径

```
zip 2.zip 2.txt #将2.txt压缩到2.zip中  
zip data.zip data #只会压缩文件夹,不会压缩文件夹下的内容  
zip da.zip da/* #压缩文件夹和文件夹内的文件(压缩文件夹和它的下一级文件)  
zip -r data.zip date #-r表示递归地将文件夹及它的子目录文件全部压缩
```

unzip解压命令

unzip 压缩文件路径

```
unzip 2.zip #将2.zip压缩包解压到当前目录下  
unzip -l 压缩文件名 #不解压文件,查看压缩包内的文件  
unzip -l da.zip #查看da.zip压缩文件中包含的文件  
unzip da.zip -d 目标目录 #将压缩文件解压到指定目录  
unzip da.zip -d tm/ #将压缩文件da.zip解压到tm目录下
```

tar命令,用来压缩和解压缩.tar和.tar.gz包

压缩.tar包:

```
tar cvf 压缩文件名 要压缩的文件或目录  
tar cvf 2.tar 2.txt #将2.txt压缩为2.tar包  
tar cvf data.tar data #将data目录夸张到data.tar包中
```

解压.tar包:

tar xvf 压缩文件名 [-C 指定解压目录]

```
tar xvf 2.tar #将2.tar解压到当前目录  
tar xvf 2.tar -C a/ #将2.tar解压到a目录  
tar xvf data.tar #解压data.tar到当前目录
```

压缩.tar.gz包:

```
tar zcvf 压缩文件名 要压缩的文件  
tar zcvf tm.tar.gz tm #将当前目录下的tm目录压缩为tm.tar.gz
```

解压.tar.gz包:

```
tar zxvf 压缩文件名  
tar zxvf tm.tar.gz #将tm.tar.gz解压到当前目录  
gzip命令,将文件压缩为.gz包(可以用来压缩.tar文件)  
gzip 要压缩的文件  
gzip 2.txt #将2.txt压缩为2.txt.gz  
gzip data.tar #将data.tar压缩为data.tar.gz
```

source命令

source 文件路径 #让配置文件修改结果立即生效,(还可以在shell脚本中引用其他的shell脚本)

```
/etc/profile #linux上的系统环境变量配置文件  
source /etc/profile #将系统环境变量生效
```

export命令

export 导入全局变量(环境变量)

```
export 变量名=变量值  
export 变量名
```

变量的赋值：

变量名=变量值

<<EOF

<<EOF ... EOF:将<<EOF和EOF之间的多行内容传给前面的命令,
其中EOF可以是任意字符串,但约定都使用EOF

```
[root@bow ~]# cat <<EOF  
> HELLO  
> WORD  
> JOB  
> SMITH  
> EOF  
HELLO  
WORD  
JOB  
SMITH
```

<<EOF是shell脚本中使用sqlplus的基础

```
[root@bow ~]# cat <<A  
> 11234  
> 1234  
> 1234  
> 1253  
> 1253  
> A  
11234  
1234  
1234  
1253  
1253
```

注意:EOF必须顶行写

```
[root@bow ~]# cat <<EOF
> ASDF
> EOF
> ASDFASDF
> EOF
ASDF
EOF
ASDFASDF
```

cut命令

cut 截取命令

```
-f 参数,指定列
-d 参数指定列和列之间的分隔符,默认的分隔符是\t(行向制表符)
cut -f 1 1.txt #取1.txt文件中的第1列内容(列分隔符默认为\t)
cut -f 2 1.txt #取1.txt文件中的第2列内容
cut -f 1 -d ',' 3.txt #取3.txt文件中的第1列(列分隔符为,)
cut -f 2 -d ',' 3.txt #取3.txt第2列
```

wc -l linux常用命令.txt | cut -f 1 -d '' #取文件linux常用命令.txt的行数(分隔符是空格)

```
[root@bow ~]# cut -f 1 -d ',' <<EOF
> A,B,C
> D,E,F
> EOF
A
D
```

printf命令

```
%ns    输出字符串, n是数字, 指代输出几个字符
%ni    输出整数。n是数字, 指代输出几个数字
%m.nf  位数和小数位数。例如: %8.2f 代表输出8位数, 其中2位是小数, 6位是整数
```

printf 格式字符串 内容

```
[root@bow ~]# printf '%s,%s,%s\n' abc def ghj k1j k1o qer #一行单词第三个打印成一行, 单词和单词之间用逗号隔开
abc,def,ghj
k1j,k1o,qer
[root@bow ~]# printf '%s %s\n' $(cat 4.txt) #将文件4.txt中的一行内容中的单词划分为两个一组打印 cat 合作查看文件内容 $(cat 4.txt)表示取cat命令的执行结果
empno ename
job sal
comm depno
5.txt内容
A B C D E
F G H
[root@bow ~]# printf '%s,%s\n' $(cat 5.txt)
```

```
A,B  
C,D  
E,F  
G,H  
[root@bow ~]# printf '%5.2f\n' 12.1 #%5.2f表示输出一个小数,数的长度是5,其中有两个小数  
12.10  
[root@bow ~]# printf '%5.2f\n' 121234.116134 #如果输出的值最大长度超出5,那么整数部分  
不变量,小数部分会按照四舍五入的方法保存两位  
121234.12  
[root@bow ~]# printf '%i\n' 1234.5678 #%i只取数字的整数部分  
-bash: printf: 1234.5678: 无效数字  
1234
```

awk命令

awk 命令字符串 要处理的内容

```
[root@bow ~]# awk '{printf $1 "\n"}' 1.txt #printf 打印 $n 表示取第几列 $1表示取第1  
列  
Hello  
smith  
tomcat
```

awk '{print \$2}' 1.txt #取1.txt的第2列,print和printf功能相同是打印,比printf多一个换行功能

```
[root@bow ~]# awk '{printf $1 ","}' 1.txt  
Hello,smith,tomcat,[root@bow ~]#  
[root@bow ~]# awk '{printf $1}' 1.txt  
Hellosmithtomcat  
[root@bow ~]# awk '{printf $1 "\v"}' 1.txt  
Hello  
    smith  
        tomcat  
[root@bow ~]# awk '{printf $1 ","}' 1.txt  
Hello,smith,tomcat,
```

sed命令

sed 参数 命令 要处理的内容

-n 一般sed命令会把所有数据都输出到屏幕。如果加入此选择，则只会把经过sed命令处理的行输出到屏幕。

-e 允许对输入数据应用多条sed命令编辑

-i 用sed的修改结果直接修改读取的数据的文件，而不是修改屏幕输出

```
[root@bow ~]# sed '2p' 1.txt #查询第2行  
Hello world  
smith 18  
smith 18  
tomcat etl  
[root@bow ~]# sed -n '2p' 1.txt  
smith 18  
[root@bow ~]# sed -i 's/18/20/g' 1.txt 使用sed命令修改1.txt内容,将1.txt中18替换为20
```

微信搜索公众号：Java专栏，获取最新面试手册

```
[root@bow ~]# cat 1.txt
Hello world
smith 20
tomcat etl
a\ 追加，在当前行后添加一行或多行。添加多行时除最后一行外，每行末尾需要用"\\"代表数据未完
结。
d 删除，删除指定的
p 打印，输出指定的行
[root@bow ~]# sed -i '2a !' 1.txt #在第2行后面追加一行 !
[root@bow ~]# cat 1.txt
Hello world
smith 20
!
tomcat etl
[root@bow ~]# sed -i '3d' 1.txt #删除文件的第3行内容
[root@bow ~]# cat 1.txt
Hello world
smith 20
tomcat etl
[root@bow ~]# vim 6.txt
[root@bow ~]# cat 6.txt
abcd/home/bow
if ad
-e /home/bow
abcd/home/bow
if ad
-e /home/bow
abcd/home/bow
if ad
-e /home/bow
#将6.txt文件中的/home/bow修改为/user/bw
#注意：替换时，的符号是根据/来判断 s/原字符串/目标字符串/g 如果原字符串或新的字符串中有/时，需
要使用\来转义
# 错误写法:s//home/bow//user/bw/g 正确写法 s/>\home\b bow\user\b bw\g
[root@bow ~]# sed -i 's/>\home\b bow\user\b bw\g' 6.txt
[root@bow ~]# cat 6.txt
abcd/user/bw
if ad
-e /user/bw
abcd/user/bw
if ad
-e /user/bw
abcd/user/bw
if ad
-e /user/bw
```

注意:linux中字符串的下标是从0开始的

service命令

service服务命令

```
service 服务名 [命令]
命令:enable|disable|start|stop|restart|status
start:启动服务
```

```
stop:关闭服务  
restart:重启服务  
status:查看服务状态  
service network start #遍历网络  
service network stop #关闭网络  
service network restart #重启网络  
service network status #查看网络状态  
service iptables start #centos6及6以下版本,启动防火墙的命令  
service iptables stop #centos6及6以下版本,关闭防火墙(注意,关闭防火墙,只是临时关闭,下次  
重启之后防火墙依然会启动)  
service iptables restart #重启防火墙  
service mysqld start #启动mysql数据库  
service mysqld restart #启动mysql数据库  
service mysqld stop #关闭mysql数据库
```

chkconfig命令

chkconfig命令检查, 设置系统的各种服务

```
chkconfig 服务名 on|off #on表示打开服务 off表示关闭服务 通过chkconfig设置的服务是永久生  
效  
centos6及以下版本永久关闭或打开防火墙  
chkconfig iptables on #打开防火墙  
chkconfig iptables off #永久地关闭防火墙
```

防火墙:

centos7以上:

```
systemctl start firewalld #启动防火墙  
systemctl stop firewalld #关闭防火墙(临时关闭)  
systemctl status firewalld #查看防火墙状态  
systemctl disable firewalld #永久关闭防火墙  
systemctl enable firewalld #打开防火墙(不是启动防火墙)  
通过firewall-cmd来配置防火墙
```

centos6及以下:

防火墙配置文件:/etc/iptables,这个文件可以详细的配置防火墙,如果没有/etc/iptables文件可以使用
iptables save可以生成该文件

iptables 命令配置防火墙

```
service iptables start #centos6及6以下版本,启动防火墙的命令  
service iptables stop #centos6及6以下版本,关闭防火墙(注意,关闭防火墙,只是临时关  
闭,下次重启之后防火墙依然会启动)  
service iptables restart #重启防火墙
```

环境变量配置文件

/etc/profile是linux系统上配置系统环境变量的一个文件(针对所有用户的配置)
用户根目录下的.bash_profile:是用户环境变量的配置(针对当前用户有效)

```
su - 用户名 #切换用户时,会加载用户根目录下的.bash_profile环境变量配置文件  
su 用户名 #不会加载.bash_profile
```

网络配置文件

网卡配置文件目录:/etc/sysconfig/network-scripts

网卡配置文件名都是以ifcfg-开头,其中ifcfg-lo是本地网卡,是不需要配置的

```
vim /etc/sysconfig/network-scripts/ifcfg-enp0s3  
#网卡类型  
TYPE="Ethernet"  
#协议 dhcp表示:ip地址是自动分配的,static表示静态ip(手动配置ip地址),none表示没有协议(也是  
需要手动配置ip地址)  
BOOTPROTO="dhcp"  
DEFROUTE="yes"  
#网卡名称  
NAME="enp0s3"  
UUID="deed3fd2-bd67-459b-8a49-ef0dd6e575a2"  
DEVICE="enp0s3"  
#配置网卡是否随机启动,yes:表示随机启动,no:表示需要手动启动  
ONBOOT="yes"  
#配置静态ip,BOOTPROTO必须是static或none  
#ip地址配置  
IPADDR=192.168.1.106  
#配置子网掩码  
NETMASK=255.255.255.0  
#配置网关  
GATEWAY=192.168.1.1  
#配置dns:域名解析服务器可以配置多个  
DNS1=192.168.1.1  
DNS2=192.168.5.1
```

修改完网卡文件之后,重启网络即可

sudo命令

sudo命令,它在非root用户下,去调用一些root用户的命令,或者修改一些文件

sudo命令是需要配置的,sudo的配置文件是/etc/sudoers

```
#给bow用户配置sudo权限  
[root@bow ~]# vim /etc/sudoers  
##  
## Allow root to run any commands anywhere  
root ALL=(ALL) ALL  
#给bow用户设置sudo命令权限  
bow ALL=(ALL) ALL
```

sudo命令的使用:

sudo 命令

```
[root@bow ~]# su - bow  
上一次登录: 四 3月 26 07:30:53 CST 2020pts/0 上  
[bow@bow ~]$ sudo vim /etc/profile
```

ping命令

ping命令查看网络连通性的命令和windows上的功能一样

```
ping ip (0.0.0.100)
```

ifconfig命令

ifconfig命令属于net-tools软件包,使用前需要安装net-tools

net-tools的安装:

```
yum -y install net-tools  
ifconfig查看ip地址
```

netstat命令

netstat命令也属于net-tools软件包

```
netstat -tulp | grep 1521 #查看oracle监听器程序是否正常启动
```

rpm命令

rpm是linux上的安装命令,用来安装.rpm格式的安装包

```
rpm -ivh .rpm文件的路径 #表示安装软件包  
  
rpm -qa #查看已安装的软件  
rpm -qa | grep mysql #查看已经安装的mysql软件包  
  
rpm -e --nodeps 安装包名 #卸载软件包 -e表示卸载 --nodeps表示不理会的依赖关系  
OK, 本文就这样。
```