**java面试题**

# Java基础

## JDK 和 JRE 有什么区别？

### JDK

* Java Development Kit 的简称，java 开发工具包，提供了 java 的开发环境和运行环境。

### JRE

* Java Runtime Environment 的简称，java 运行环境，为 java 的运行提供了所需环境。

### 具体来说 JDK 其实包含了 JRE，同时还包含了编译 java 源码的编译器 javac，还包含了很多 java 程序调试和分析的工具。简单来说：如果你需要运行 java 程序，只需安装 JRE 就可以了，如果你需要编写 java 程序，需要安装 JDK。

## == 和 equals 有什么区别？

### == 解读

* 对于基本类型和引用类型 == 的作用效果是不同的，如下所示：

• 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；

* 代码示例：

• String x = "string";String y = "string";String z = new String("string");System.out.println(x==y); // trueSystem.out.println(x==z); // falseSystem.out.println(x.equals(y)); // trueSystem.out.println(x.equals(z)); // true

* 代码解读

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

### equals 解读

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

• 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;Cat c2 = new Cat;System.out.println(c1.equals(c2)); // false

* 代码解读

• 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("老王");String s2 = new String("老王");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。

## 两个对象的 hashCode() 相同，则 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 = "通话";String str2 = "重地";System.out.println(String.format("str1：%d | str2：%d", str1.hashCode(),str2.hashCode()));System.out.println(str1.equals(str2));

* 执行的结果：

• str1：1179395 | str2：1179395false

* 代码解读

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

## 了解 final 这个关键字吗？一般是怎么用的

### final 修饰的类叫最终类，该类不能被继承。

### final 修饰的方法不能被重写。

### final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

## 静态变量和实例变量有什么区别？

## Object 这个类了解吗？能挑几个你熟悉的方法简单介绍下吗？

## java 中的 Math.round(-1.5) 等于多少？

### 等于 -1，因为在数轴上取值时，中间值（0.5）向右取整，所以正 0.5 是往上取整，负 0.5 是直接舍弃。

## 一个 char 型变量中能不能存贮一个中文汉字，为什么？

## String 属于基础的数据类型吗？

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

## String 类里面的方法了解吗？

### 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。

## 是否可以继承 String 类？

## 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。

## String str="i" 与 String str=new String("i") 有什么区别？

### 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。

## 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性吗？这里是值传递还是引用传递？

## 抽象类必须要有抽象方法吗？

### 不需要，抽象类不一定非要有抽象方法。

* 示例代码

• abstract class Cat { public static void sayHi() { System.out.println("hi~"); }}

* 上面代码，抽象类并没有抽象方法但完全可以正常运行

## 普通类和抽象类有哪些区别？

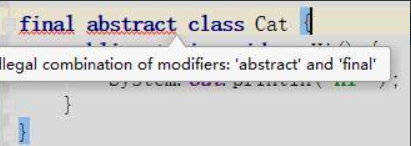
### 普通类不能包含抽象方法，抽象类可以包含抽象方法。

### 抽象类不能直接实例化，普通类可以直接实例化。

## 抽象类能使用 final 修饰吗？

### 不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼此就会产生矛盾，所以 final 不能修饰抽象类，如下图所示，编辑器也会提示错误信息：

### 



## 接口和抽象类有什么区别？

### 实现

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

### 构造函数

* 抽象类可以有构造函数；接口不能有。

### main 方法

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

### 实现数量

* 类可以实现很多个接口；但是只能继承一个抽象类

### 访问修饰符

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

## 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。

## BIO、NIO、AIO 有什么区别？

### BIO

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

### NIO

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

### AIO

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

## Files的常用方法都有哪些？

### Files.exists()

* Files.exists()：检测文件路径是否存在。

### Files.createFile()

* Files.createFile()：创建文件。

### Files.createDirectory()

* Files.createDirectory()：创建文件夹

### Files.delete()

* Files.delete()：删除一个文件或目录。

### Files.copy()

* Files.copy()：复制文件。

### Files.move()

* Files.move()：移动文件。

### Files.size()

* Files.size()：查看文件个数。

### Files.read()

* Files.read()：读取文件。

### Files.write()

* Files.write()：写入文件。

## 怎么实现动态代理？

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

## 访问修饰符 public，private，protected，以及不写（默认）时的区别？

## float f=3.4; 这样写会不会有问题？ long l = 5; 呢？

### float f=3.4; 这样写会不会有问题？

* 不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成 float f =3.4F;。

## int 和 Integer 有什么区别？自动拆箱和自动装箱有了解过吗？

### int 和 Integer 有什么区别

* Java 为每个原始类型提供了包装类型：原始类型: boolean，char，byte，short，int，long，float，double包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

### 自动拆箱和自动装箱有了解过吗

* 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。
* 什么是自动装箱拆箱

• 很简单，下面两句代码就可以看到装箱和拆箱过程

• 1 //自动装箱2 Integer total = 99;3 4 //自动拆箱5 int totalprim = total;

## & 和 && 这两个运算符的作用一样吗？

### &amp;运算符有两种用法：(1)按位与；(2)逻辑与。&amp;&amp;运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true 整个表达式的值才是 true。&amp;&amp;之所以称为短路运算是因为，如果&amp;&amp;左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&amp;&amp;而不是&amp;，例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为：username != null &amp;&amp;!username.equals(“”)，二者的顺序不能交换，更不能用&amp;运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（| |）的差别也是如此。

## 数组有没有 length() 方法？String 有没有 length() 方法？

## 构造器（constructor）是否可被重写（override）

## Java中会存在内存泄漏吗？简单描述一下。

## Java中怎么获取当前时间，那种方式效率更高，为什么？

## 面向对象的特性有哪些？

## 多态有什么好处？

## Java中覆盖和重载是什么意思？

## 抽象类和接口的区别有哪些？

## JDK中常用的包有哪些？

## JDK，JRE 和 JVM 的联系和区别

## 值传递和引用传递的区别？

## Java里的枚举实现机制是什么？

## Java中的内部类

## 位运算(&amp;、|、~、^、&lt;&lt;、&gt;&gt;、&gt;&gt;&gt;)

## 递归的优缺点

## Java的四种引用

# Java进阶

## 说一下 HashMap 的实现原理？

### HashMap概述

* HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

### HashMap的数据结构

* 在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

### 当我们往Hashmap中put元素时,首先根据key的hashcode重新计算hash值,根绝hash值得到这个元素在数组中的位置(下标),如果该数组在该位置上已经存放了其他元素,那么在这个位置上的元素将以链表的形式存放,新加入的放在链头,最先加入的放入链尾.如果数组中该位置没有元素,就直接将该元素放到数组的该位置上。

### 需要注意Jdk 1.8中对HashMap的实现做了优化,当链表中的节点数据超过八个之后,该链表会转为红黑树来提高查询效率,从原来的O(n)到O(logn)

## 说一下 HashSet 的实现原理？

### HashSet底层由HashMap实现

### HashSet的值存放于HashMap的key上

### HashMap的value统一为PRESENT

## 哪些集合类是线程安全的？

### vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

### statck：堆栈类，先进后出。

### hashtable：就比hashmap多了个线程安全。

### enumeration：枚举，相当于迭代器。

## 说一下 runnable 和 callable 有什么区别？

### 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；

### 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。

## 线程的 run() 和 start() 有什么区别？

### 每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，方法run()称为线程体。通过调用Thread类的start()方法来启动一个线程。

### start()方法来启动一个线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码； 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行状态， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

### run()方法是在本线程里的，只是线程里的一个函数,而不是多线程的。 如果直接调用run(),其实就相当于是调用了一个普通函数而已，直接待用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

## 启动一个线程是调用 run() 还是 start() 方法？

### 启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

## 线程池中 submit( )和 execute() 方法有什么区别？

### 接收的参数不一样

### submit有返回值，而execute没有

### submit方便Exception处理

## ThreadLocal 是什么？有哪些使用场景？

### 线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

## 说一下 synchronized 底层实现原理？

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

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

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

## 说一下 volatile 的实现原理

## synchronized 和 volatile 的区别是什么？

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

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

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

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

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

## synchronized 和 Lock 有什么区别？

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

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

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

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

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

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

## synchronized 和 ReentrantLock 区别是什么？

### 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：

* ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
* ReentrantLock可以获取各种锁的信息
* 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。

## 说一下 atomic 的原理？

### Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

### Atomic系列的类中的核心方法都会调用unsafe类中的几个本地方法。我们需要先知道一个东西就是Unsafe类，全名为：sun.misc.Unsafe，这个类包含了大量的对C代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过unsafe分配内存的时候，如果自己指定某些区域可能会导致一些类似C++一样的指针越界到其他进程的问题。

## 多线程锁的升级原理是什么？

### 锁的升级的目的

* 锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

### synchronized 锁升级原理

* 在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

### 偏向锁

* 顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁／解锁的一些CAS操作（比如等待队列的一些CAS操作），这种情况下，就会给线程加一个偏向锁。 如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

### 轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，轻量级锁就会升级为重量级锁；

### 重量级锁是synchronized ，是 Java 虚拟机中最为基础的锁实现。在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。

## 讲一讲你对AQS的理解？

## ReentrantLock 中 ConditionObject 的使用了解吗？

## 动态代理是什么？有哪些应用？

### 动态代理：

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

### 动态代理的应用：

* Spring的AOP加事务加权限加日志

## 怎么实现动态代理？

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

## Java的泛型了解吗？

## 平时会用 lombok 吗？

## 有去看过 jsp 的东西吗？它和 servlet 有什么区别？

### jsp 和 servlet 有什么区别？

* jsp经编译后就变成了Servlet.（JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码，Web容器将JSP的代码编译成JVM能够识别的java类）
* jsp更擅长表现于页面显示，servlet更擅长于逻辑控制。
* Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到。
* Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

## JSR303、349 里面的规范有了解过吗？你一般是怎么对参数做校验的

## 全局异常处理是怎么做的？

## 讲讲类的加载机制？

## JVM 的内存结构了解吗？

## 讲一下常见的 GC 算法？

### 什么是堆？

* 堆指用于动态（即执行程序时）存放对象的内存空间。而这个对象，在面向对象的编程中，它指“具有属性和行为的事物”，然而在GC的世界中，对象表示的是“通过应用程序利用的数据的集合”。具体到Java堆，它是所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。（此处的对象实例可以理解为前面所说的对象，因为不仅仅Java有自动的GC，python、JavaScript等语言也有，所以在广义上说对象是更好的表述，当然，Java的数组也是分配在堆上的）。

### GC算法的评判标准

* GC算法的评判标准主要是以下4点：

• 吞吐量：即单位时间内的处理能力。

• 最大暂停时间：因执行GC而暂停执行程序所需的时间。

• 堆的使用效率：鱼与熊掌不可兼得，堆使用效率和吞吐量、最大暂停时间是不可能同时满足的。即可用的堆越大，GC运行越快；相反，想要利用有限的堆，GC花费的时间就越长。

• 访问的局部性：在存储器的层级构造中，我们知道越是高速存取的存储器容量会越小（具体可以参看我写的存储器那篇文章）。由于程序的局部性原理，将经常用到的数据放在堆中较近的位置，可以提高程序的运行效率。

### 常用的GC算法

* 引用计数法

• 什么是引用计数法？

• 所谓的引用计数法就是给每个对象一个引用计数器，每当有一个地方引用它时，计数器就会加1；当引用失效时，计数器的值就会减1；任何时刻计数器的值为0的对象就是不可能再被使用的。

• 这个引用计数法时没有被Java所使用的，但是python有使用到它。而且最原始的引用计数法没有用到GC Roots。

• 优点

• 可即时回收垃圾：在该方法中，每个对象始终知道自己是否有被引用，当被引用的数值为0时，对象马上可以把自己当作空闲空间链接到空闲链表。

• 最大暂停时间短。

• 没有必要沿着指针查找

• 缺点

• 计数器值的增减处理非常繁重

• 计算器需要占用很多位

• 实现繁琐

• 循环引用无法回收

* 标记-清除法

• 什么是标记-清除算法？

• 该算法分为标记和清除两个阶段。标记就是把所有活动对象都做上标记的阶段；清除就是将没有做上标记的对象进行回收的阶段。如下图所示。

• 优点

• 实现简单

• 与保守式GC算法兼容（保守式GC在后面介绍）

• 缺点

• 碎片化：如上图所示，在回收的过程中会产生被细化的分块，到后面，即使堆中分块的总大小够用，但是却因为分块太小而不能执行分配。

• 分配速度：因为分块不是连续的，因此每次分块都要遍历空闲链表，找到足够大的分块，从而造成时间的浪费。

• 与写时复制技术不兼容：所谓写时复制就是fork的时候，内存空间只引用而不复制，只有当该进程的数据发生变化时，才会将数据复制到该进程的内存空间。这样，当两个进程中的内存数据相同的时候，就能节约大量的内存空间了。而对于标记-清除算法，它的每个对象都有一个标志位来表示它是否被标记，在每一次运行标记-清除算法的时候，被引用的对象都会进行标记操作，这个仅仅标记位的改变，也会变成对象数据的改变，从而引发写时复制的复制过程，与写时复制的初衷就背道而驰了。

* 复制算法

• 什么是复制算法？

• 复制算法就是将内存空间按容量分成两块。当这一块内存用完的时候，就将还存活着的对象复制到另外一块上面，然后把已经使用过的这一块一次清理掉。这样使得每次都是对半块内存进行内存回收。内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶的指针，按顺序分配内存即可，实现简单，运行高效。

• 优点

• 优秀的吞吐量

• 可实现高速分配：复制算法不用使用空闲链表。这是因为分块是连续的内存空间，因此，调用这个分块的大小，只需要这个分块大小不小于所申请的大小，移动指针进行分配即可。

• 不会发生碎片化。

• 与缓存兼容。

• 缺点

• 堆的使用效率低下。

• 不兼容保守式GC算法。

• 递归调用函数。

* 标记-清除算法

• 什么是标记-压缩算法？

• 标记-压缩算法与标记-清理算法类似，只是后续步骤是让所有存活的对象移动到一端，然后直接清除掉端边界以外的内存。

• 优缺点

• 该算法可以有效的利用堆，但是压缩需要花比较多的时间成本。

### 可达性

* 所谓的可达性就是通过一系列称为“GC Roots”的对象为起点，从这些节点开始向下搜索，搜索走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是GC Roots到这个对象不可达）时，则说明此对象是不可用的。如下图所示，ABC可达，DE不可达。
* 那么那些对象可以作为GC Roots呢？以Java为例，有以下几种：

• 栈（栈帧中的本地变量表）中引用的对象。

• 方法区中的静态成员。

• 方法区中的常量引用的对象（全局变量）。

• 本地方法栈中JNI（一般说的Native方法）引用的对象。

• 注：第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

### 保守式GC与准确式GC

* 保守式GC

• 所谓保守式GC就是“不能识别指针和非指针的GC”。

• 对于寄存器、调用栈、全局变量空间来说，都是不明确的根。例如调用栈中，装着函数内的局部变量和参数值。而局部变量，如C语言中的int、double这样就是非指针，但是也会有像void\*这样的指针。

• 那么保守式GC会怎么检查不明确的根呢？1、是不是被正确对齐的值？（在32位CPU的情况下，为4的倍数）2、是不是指着堆内？3、是不是指向对象的开头？当然，这些只是基本的检查项目。

• 上面的检查方法会将一些非指针识别成指针。例如一个数值和一个地址，它们两个值相等，这个时候，那个值也可以被识别成指针

• 保守式GC的优点是语言处理程序不依赖与GC。缺点为识别指针和非指针需要付出成本、错误识别指针会压迫堆、能够使用的GC算法有限。例如GC复制算法就不能使用，因为其可能会将非指针重写。

* 准确式GC

• 准确式GC能够正确识别指针和非指针的GC。正确的根的创建方法是依赖于语言处理程序的实现的。我们可以通过打标签、不把寄存器和栈等当作根的方法来实现。

• 其优点就是完全能够识别指针，能够使用复制算法等需要移动对象的算法。但是在创建准确式GC时，语言处理程序必须对GC进行一些支援，而且创建正确的根就必须付出一定的代价。

• 其实我们垃圾回收机的实现都不是仅仅用哪一种回收算法，都是将几个结合使用，特别是分代算法，后面我们会详细的介绍。

## 讲一下常见的垃圾收集器？

### 垃圾回收器是什么

* JVM垃圾收集器发展历程大致可以分为以下四个阶段： Serial（串行）收集器 -&gt; Parallel(并行)收集器 -&gt; CMS（并发）收集器 -&gt; G1（并发）收集器

### Serial类收集器

* Serial类收集器是一个单线程的收集器:

• 它只会用单个收集线程去进行垃圾回收的工作

• 它在进行垃圾收集的时候会“Stop The World”暂停其他所有的工作表线程，直到它收集结束

• Serial收集器采取复制算法在新生代进行单线程的回收工作

• Serial Old收集器采取标记-整理算法在老年代进行单线程的回收工作

### Parallel类收集器

* Parallel类收集器就是Serial收集器的多线程版本：

• 它使用多个收集线程取进行垃圾回收工作

• 它在进行垃圾收集的时候也会“Stop The World”暂停其他所有的工作表线程，直到它收集结束

• ParNew收集器采取复制算法在新生代进行多线程的回收工作

• Parallel Scavenge收集器也是一个新生代收集器，不过它被称为“吞吐量优先”收集器，它提供了2个能精确控制吞吐量的参数：

• -XX : MaxGCPauseMillis：控制最大垃圾收集停顿时间

• -XX : GCTimeRatio : 直接设置吞吐量大小，垃圾收集时间占总时间的比率

• Parallel Scavenge收集器还有一个开关参数-XX: UseAdaptiveSizePolicy，打开这个开关后就不用手动指定新生代的大小(-Xmn),Eden与Survivor区的比例(-XX:SurvivorRatio)等细节参数了，JVM会动态调整这些参数已提供最合适的停顿时间或者最大吞吐量。

• Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法在老年代进行垃圾回收

### CMS收集器

* CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。它是一个基于标记-清除算法实现的，运作过程分为4个步骤：

• 初始标记(CMS initial mark): 需要“Stop The World”，仅仅只是标记下GC Roots能直接关联到的对象，速度很快

• 并发标记(CMS concurrent mark): CMS线程与应用线程一起并发执行，从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长

• 重新标记(CMS remark)：重新标记就是为了修正并发标记期间因用户线程继续运作而导致标记产生变动的那一部分对象的标记记录，可以多线程并行

• 并发清除(CMS concurrent sweep)：CMS线程与应用线程一起并发执行，进行垃圾清除

* CMS收集器优点

• 并发收集、低停顿

* CMS的三个明显的缺点

• CMS收集器对CPU的资源非常敏感。CPU的数量较少不足4个（比如2个）时，CMS对用户程序的影响就可能变的很大。

• CMS收集器无法处理浮动垃圾(Floating Carbage)，可能出现"Concurrent Mode Failture"失败而导致产生另一次Full GC的产生。浮动垃圾就是并发清理阶段，用户线程产生的新垃圾

• CMS是基于标记-清除算法的，收集结束后会有大量的空间碎片，就可能会在老年代中无法分配足够大的连续空间而不得不触发另一次Full GC。

### G1收集器

* 介绍

• 同优秀的CMS一样，G1也是关注最小停顿时间的垃圾回收器，也同样适合大尺寸堆内存，官方也推荐用G1来代替选择CMS。

• G1收集器的最大特点就是引入了分区的思路，弱化了分代的概念G1从整体来看是基于标记-整理算法实现的，从局部(两个Region之间)来看是基于复制算法实现的

* G1相对于CMS的改进

• G1是基于标记-整理算法，不会产生空间碎片，在分配大的连续对象是不会因为无法得到连续空间而不得不提前触发一次Full GC

• 停顿时间可控，G1可以通过设置停顿时间来控制垃圾回收时间

• 并行与并发，G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间

* G1与CMS的区别

• 堆内存模型的不同

• G1之前的JVM堆内存模型，堆被分为新生代，老年代，永久代(1.8之前，1.8之后是元空间)，新生代中又分为Eden和两个Survivor区。

• G1收集器的堆内存模型，堆被分为很多个大小连续的区域（Region），Region的大小可以通过-XX: G1HeapRegionSize参数指定，大小区间为[1M，32M]。

• 每个Region被标记了E、S、O和H，这些区域在逻辑上被映射为Eden，Survivor，老年代和巨型区(Humongous Region)。巨型区域是为了存储超过50%标准region大小的巨型对象。

• 应用分代的不同

• G1可以在新生代和老年代使用，而CMS只能在老年代使用。

• 收集算法的不同

• G1是复制+标记-整理算法，CMS是标记清除算法。

* G1收集器的工作流程

• G1收集器的工作流程大致分为如下几个步骤：

• 初始标记(Initial Marking): 需要“Stop The World”，仅仅只是标记下GC Roots能直接关联到的对象，速度很快

• 并发标记(Concurrent Marking): G1线程与应用线程一起并发执行，从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长

• 最终标记(Final Marking): 最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，需要“Stop The World”，可以多线程并行

• 筛选回收(Live Data Counting and Evacuation): 对各个Region的回收价值和成本进行排序，根据用户所期待的GC停顿时间制定回收计划。具体地，在后台维护一个优先队列，每次根据允许的收集停顿时间，优先回收价值最大的Region。

* G1的GC模式

• G1提供了两种GC模式，Young GC和Mixed GC，两种都是完全Stop The World的

• YoungGC

• 在分配一般对象(非巨型对象)时，当所有的Eden Region使用达到最大阈值并且无法申请到足够内存时，会触发一次YoungGC。

• 每次YoungGC会回收所有的Eden以及Survivor区，并且将存活对象复制到Old区以及另一部分的Survivor。

• MixedGC

• 当越来越多的对象晋升到老年代old region时，为了避免堆内存被耗尽，虚拟机会触发一次mixed gc，该算法并不是一个old gc，除了回收整个young region，还会回收一部分的old region。这里需要注意：是一部分老年代，而不是全部老年代，可以选择哪些old region进行收集，从而可以对垃圾回收的耗时时间进行控制

• G1没有fullGC概念，需要fullGC时，调用serialOldGC进行全堆扫描（包括eden、survivor、o、perm）。

## 有过 JVM 的调优经验吗？你是怎么做的？

## 单线程和多线程的区别和联系？

## 如何指定多个线程的执行顺序？

## 线程和进程的区别？

## 多线程产生死锁的4个必要条件？

### 互斥条件：

* 进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

### 请求和保持条件：

* ：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放

### 不可剥夺条件：

* 是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

### 环路等待条件：

* 是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

## 对于线程死锁有解决思路吗？

## sleep() 和 wait(n)、wait() 的区别：

### sleep()：

* 方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

### wait()

* wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程。

## synchronized 关键字

### synchronized 底层实现原理

* synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。
* Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

• 普通同步方法，锁是当前实例对象

• 静态同步方法，锁是当前类的class对象

• 同步方法块，锁是括号里面的对象

## volatile 关键字

## ThreadLocal实现原理？

### ThreadLocal 是什么？有哪些使用场景？

* 线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

## Atomic 相关类？

## 线程池的实现原理？

## Lock的API？有哪些实现？

## ReentrantLock的实现原理？

## AQS了解过吗？

## ABA问题了解过吗？怎么解决？

## synchronized跟Lock接口的优缺点？

## 生产者消费者模式有了解过吗？

## HashMap的长度为什么是2的幂次方？

## HashMap处理冲突方法,各个方法优缺点

## HashMap高并发情况下会出现什么问题？

## ConcurrentHashMap的实现原理？

## ConcurrentHashMap如何扩容？

## LinkedHashMap的实现原理？

## 如何用LinkedHashMap实现LRU？

## PriorityQueue的实现原理有了解过吗？

## PriorityQueue的应用场景有哪些？

# 计算机网络篇

## 说几个你了解的端口号及对应的服务？

## OSI 与 TCP/IP 各层的结构与功能，都有哪些协议？

## TCP 和 UDP 的区别是什么？

### TCP是什么

* TCP（Transmission Control Protocol）协议全称是传输控制协议，是一种面向连接的、可靠的、基于字节流的传输层通信协议，由RFC 793定义。

### UDP是什么

* UDP（User Datagram Protocol）全称是用户数据电报协议，是一种无连接的协议，提供不可靠的用户数据报服务，1980 年发布的 RFC 768 定义了 UDP 协议。

### TCP、UDP两者的区别

* TCP是面向连接的（在客户端和服务器之间传输数据之前要先建立连接），UDP是无连接的（发送数据之前不需要先建立连接）
* TCP提供可靠的服务（通过TCP传输的数据。无差错，不丢失，不重复，且按序到达）；UDP提供面向事务的简单的不可靠的传输。
* UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性比较高的通讯或广播通信。随着网速的提高，UDP使用越来越多。
* 没一条TCP连接只能是点到点的，UDP支持一对一，一对多和多对多的交互通信。
* TCP对系统资源要去比较多，UDP对系统资源要求比较少
* UDP程序结构更加简单
* TCP是流模式，UDP是数据报模式

### 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

## TCP 为什么要三次握手？

### 为了实现可靠数据传输， TCP 协议的通信双方， 都必须维护一个序列号， 以标识发送出去的数据包中， 哪些是已经被对方收到的。 三次握手的过程即是通信双方相互告知序列号起始值， 并确认对方已经收到了序列号起始值的必经步骤。

### 如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认。

## 为什么要传回 SYN ？

## 传了 SYN，为啥还要传 ACK ？

## TCP断开连接时，为什么要四次挥手？

### 什么是TCP的四次挥手

* 在网络数据传输中，传输层协议断开连接的过程我们称为四次挥手

### 四次挥手的具体细节

* 第一次挥手：Client将FIN置为1，发送一个序列号seq给Server；进入FIN\_WAIT\_1状态；
* 第二次挥手：Server收到FIN之后，发送一个ACK=1，acknowledge number=收到的序列号+1；进入CLOSE\_WAIT状态。此时客户端已经没有要发送的数据了，但仍可以接受服务器发来的数据。
* 第三次挥手：Server将FIN置1，发送一个序列号给Client；进入LAST\_ACK状态；
* 第四次挥手：Client收到服务器的FIN后，进入TIME\_WAIT状态；接着将ACK置1，发送一个acknowledge number=序列号+1给服务器；服务器收到后，确认acknowledge number后，变为CLOSED状态，不再向客户端发送数据。客户端等待2\*MSL（报文段最长寿命）时间后，也进入CLOSED状态。完成四次挥手。

## TCP 协议如何保证可靠传输？

## 描述一下在浏览器中输入 url 地址 —&gt; 页面渲染完成的过程

## 了解 Session、Cookie 与 Token这几个概念吗？

## Get 与 POST 有什么区别？

### 什么是Get请求

* GET - 从指定的资源请求数据

### 什么是POST请求

* POST - 向指定的资源提交要被处理的数据

### Get 与 POST 有什么区别？

* GET 请求可被缓存 &lt;=&gt; POST 请求不会被缓存
* GET 请求保留在浏览器历史记录中 &lt;=&gt; POST请求不会保留在浏览器历史记录中
* GET 请求可被收藏为书签 &lt;=&gt; POST 不能收藏为书签
* GET 请求长度有限制 &lt;=&gt; POST 请求没有限制
* GET请求不能处理敏感数据

## Http 和 Https 的区别是什么？

### https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

### http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

### http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

### http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## HTTP 层常见的状态码有哪些？

## HTTP 的长连接了解吗？

## SQL 注入是什么原理？

### 原理

* sql注入的原理是将sql代码伪装到输入参数中，传递到服务器解析并执行的一种攻击手法。也就是说，在一些对server端发起的请求参数中植入一些sql代码，server端在执行sql操作时，会拼接对应参数，同时也将一些sql注入攻击的“sql”拼接起来，导致会执行一些预期之外的操作。

### 解决sql注入原理

* sql预编译

• 在知道了sql注入的原理之后，我们同样也了解到mysql有预编译的功能，指的是在服务器启动时，mysql client把sql语句的模板（变量采用占位符进行占位）发送给mysql服务器，mysql服务器对sql语句的模板进行编译，编译之后根据语句的优化分析对相应的索引进行优化，在最终绑定参数时把相应的参数传送给mysql服务器，直接进行执行，节省了sql查询时间，以及mysql服务器的资源，达到一次编译、多次执行的目的，除此之外，还可以防止SQL注入。

• 具体是怎样防止SQL注入的呢？实际上当将绑定的参数传到mysql服务器，mysql服务器对参数进行编译，即填充到相应的占位符的过程中，做了转义操作。 我们常用的jdbc就有预编译功能，不仅提升性能，而且防止sql注入。

• String sql = "select id, no from user where id=?"; PreparedStatement ps = conn.prepareStatement(sql); ps.setInt(1, id); ps.executeQuery();

* 严格的参数校验

• 参数校验就没得说了，在一些不该有特殊字符的参数中提前进行特殊字符校验即可

## XSS 攻击了解吗？

## TCP三次握手四次挥手

### TCP三次握手

* 客户端发送请求【寻址请求】

• 第一次握手 客户端向服务端发送连接请求报文；请求发送后，客户端便进入 SYN-SENT 状态；

* 服务器端收到报文请求，回应客户端【确认请求】

• 第二次握手 服务端收到连接请求报文后，如果同意连接，则会发送一个应答，发送完成后便进入 SYN-RECEIVED 状态；

* 客户端收到服务端的报文进行回应。【连接请求】

• 第三次握手 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文；客户端发完这个报文后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

### TCP四次挥手

* 数据验证请求码

• 第一次挥手 A：B，我不玩了 客户端A向服务端B发送连接释放请求；

* 传输结束标记

• 第二次挥手 B：OK，A不玩了，知道了 服务端B 收到连接释放请求后，发送 ACK 包，并进入 CLOSE\_WAIT 状态； 此时服务端B不再接收客户端A发送的数据，但服务端B 若此时还有没发完的数据会继续发送；

* 确认结束标记

• 第三次挥手 B：A，我也不玩了，拜拜 服务端 B 向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态；

* 连接断开标记

• 第四次挥手A：OK，B不玩了，拜拜客户端A 收到释放请求后，向 服务端B 发送确认应答，此时 客户端A 进入 TIME-WAIT 状态；客户端A的 TIME-WAIT状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有 B 的重发请求，就进入 CLOSED 状态。当 B 收到确认应答后，也便进入 CLOSED 状态。

## TCP挥手过程最后一次ack如果客户端没收到怎么办

## 对于socket编程，accept方法是干什么的，在三次握手中属于第几次

## TCP的四次挥手，time wait状态有什么意义

## 浏览器一次请求的过程

## Reactor模式

## http 响应码 301 和 302 代表的是什么？有什么区别？

### http 响应码 301 和 302 代表的是什么？

* 301，302 都是HTTP状态的编码，都代表着某个URL发生了转移。

### 区别

* 301 redirect: 301 代表永久性转移(Permanently Moved)。302 redirect: 302 代表暂时性转移(Temporarily Moved )。

## forward 和 redirect 的区别？

### Forward和Redirect代表了两种请求转发方式：直接转发和间接转发。

### 直接转发方式（Forward），客户端和浏览器只发出一次请求，Servlet、HTML、JSP或其它信息资源，由第二个信息资源响应该请求，在请求对象request中，保存的对象对于每个信息资源是共享的。

### 间接转发方式（Redirect）实际是两次HTTP请求，服务器端在响应第一次请求的时候，让浏览器再向另外一个URL发出请求，从而达到转发的目的

### 举个通俗的例子：

* 直接转发就相当于：“A找B借钱，B说没有，B去找C借，借到借不到都会把消息传递给A”；间接转发就相当于："A找B借钱，B说没有，让A去找C借"。

## 说一下 tcp 粘包是怎么产生的？

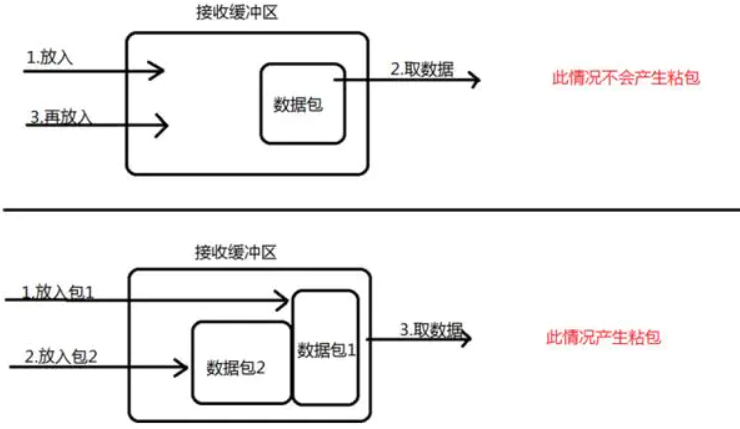
### ①. 发送方产生粘包

* 采用TCP协议传输数据的客户端与服务器经常是保持一个长连接的状态（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据；但当发送的数据包过于的小时，那么TCP协议默认的会启用Nagle算法，将这些较小的数据包进行合并发送（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是粘包的状态了。



### ②. 接收方产生粘包

* 接收方采用TCP协议接收数据时的过程是这样的：数据到底接收方，从网络模型的下方传递至传输层，传输层的TCP协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C语言用recv、read等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 &gt; 应用层拿数据速度）



## OSI 的七层模型都有哪些？

### 应用层：网络服务与最终用户的一个接口。

### 表示层：数据的表示、安全、压缩。

### 会话层：建立、管理、终止会话。

### 传输层：定义传输数据的协议端口号，以及流控和差错校验。

### 网络层：进行逻辑地址寻址，实现不同网络之间的路径选择。

### 数据链路层：建立逻辑连接、进行硬件地址寻址、差错校验等功能。

### 物理层：建立、维护、断开物理连接。

## 如何实现跨域？

### 方式一：图片ping或script标签跨域

* 图片ping常用于跟踪用户点击页面或动态广告曝光次数。script标签可以得到从其他来源数据，这也是JSONP依赖的根据。

### 方式二：JSONP跨域

* JSONP（JSON with Padding）是数据格式JSON的一种“使用模式”，可以让网页从别的网域要数据。根据 XmlHttpRequest 对象受到同源策略的影响，而利用 &lt;script&gt;元素的这个开放策略，网页可以得到从其他来源动态产生的JSON数据，而这种使用模式就是所谓的 JSONP。用JSONP抓到的数据并不是JSON，而是任意的JavaScript，用 JavaScript解释器运行而不是用JSON解析器解析。所有，通过Chrome查看所有JSONP发送的Get请求都是js类型，而非XHR。
* 缺点：

• 只能使用Get请求不能注册success、error等事件监听函数，不能很容易的确定JSONP请求是否失败JSONP是从其他域中加载代码执行，容易受到跨站请求伪造的攻击，其安全性无法确保

### 方式三：CORS

* Cross-Origin Resource Sharing（CORS）跨域资源共享是一份浏览器技术的规范，提供了 Web 服务从不同域传来沙盒脚本的方法，以避开浏览器的同源策略，确保安全的跨域数据传输。现代浏览器使用CORS在API容器如XMLHttpRequest来减少HTTP请求的风险来源。与 JSONP 不同，CORS 除了 GET 要求方法以外也支持其他的 HTTP 要求。服务器一般需要增加如下响应头的一种或几种：

• Access-Control-Allow-Origin: \*Access-Control-Allow-Methods: POST, GET, OPTIONSAccess-Control-Allow-Headers: X-PINGOTHER, Content-TypeAccess-Control-Max-Age: 86400

* 跨域请求默认不会携带Cookie信息，如果需要携带，请配置下述参数：

• "Access-Control-Allow-Credentials": true// Ajax设置"withCredentials": true

### 方式四：window.name+iframe

* window.name通过在iframe（一般动态创建i）中加载跨域HTML文件来起作用。然后，HTML文件将传递给请求者的字符串内容赋值给window.name。然后，请求者可以检索window.name值作为响应。
* iframe标签的跨域能力；window.name属性值在文档刷新后依旧存在的能力（且最大允许2M左右）。
* 每个iframe都有包裹它的window，而这个window是top window的子窗口。contentWindow属性返回&lt;iframe&gt;元素的Window对象。你可以使用这个Window对象来访问iframe的文档及其内部DOM。
* &lt;!-- 下述用端口 10000表示：domainA 10001表示：domainB--&gt;&lt;!-- localhost:10000 --&gt;&lt;script&gt; var iframe = document.createElement('iframe'); iframe.style.display = 'none'; // 隐藏 var state = 0; // 防止页面无限刷新 iframe.onload = function() { if(state === 1) { console.log(JSON.parse(iframe.contentWindow.name)); // 清除创建的iframe iframe.contentWindow.document.write(''); iframe.contentWindow.close(); document.body.removeChild(iframe); } else if(state === 0) { state = 1; // 加载完成，指向当前域，防止错误(proxy.html为空白页面) // Blocked a frame with origin "http://localhost:10000" from accessing a cross-origin frame. iframe.contentWindow.location = 'http://localhost:10000/proxy.html'; } }; iframe.src = 'http://localhost:10001'; document.body.appendChild(iframe);&lt;/script&gt;&lt;!-- localhost:10001 --&gt;&lt;!DOCTYPE html&gt;...&lt;script&gt;window.name = JSON.stringify({a: 1, b: 2});&lt;/script&gt;&lt;/html&gt;

### 方式五：window.postMessage()

* HTML5新特性，可以用来向其他所有的 window 对象发送消息。需要注意的是我们必须要保证所有的脚本执行完才发送 MessageEvent，如果在函数执行的过程中调用了它，就会让后面的函数超时无法执行。
* 下述代码实现了跨域存储localStorage
* &lt;!-- 下述用端口 10000表示：domainA 10001表示：domainB--&gt;&lt;!-- localhost:10000 --&gt;&lt;iframe src="http://localhost:10001/msg.html" name="myPostMessage" style="display:none;"&gt;&lt;/iframe&gt;&lt;script&gt; function main() { LSsetItem('test', 'Test: ' + new Date()); LSgetItem('test', function(value) { console.log('value: ' + value); }); LSremoveItem('test'); } var callbacks = {}; window.addEventListener('message', function(event) { if (event.source === frames['myPostMessage']) { console.log(event) var data = /^#localStorage#(\d+)(null)?#([\S\s]\*)/.exec(event.data); if (data) { if (callbacks[data[1]]) { callbacks[data[1]](data[2] === 'null' ? null : data[3]); } delete callbacks[data[1]]; } } }, false); var domain = '\*'; // 增加 function LSsetItem(key, value) { var obj = { setItem: key, value: value }; frames['myPostMessage'].postMessage(JSON.stringify(obj), domain); } // 获取 function LSgetItem(key, callback) { var identifier = new Date().getTime(); var obj = { identifier: identifier, getItem: key }; callbacks[identifier] = callback; frames['myPostMessage'].postMessage(JSON.stringify(obj), domain); } // 删除 function LSremoveItem(key) { var obj = { removeItem: key }; frames['myPostMessage'].postMessage(JSON.stringify(obj), domain); }&lt;/script&gt;&lt;!-- localhost:10001 --&gt;&lt;script&gt; window.addEventListener('message', function(event) { console.log('Receiver debugging', event); if (event.origin == 'http://localhost:10000') { var data = JSON.parse(event.data); if ('setItem' in data) { localStorage.setItem(data.setItem, data.value); } else if ('getItem' in data) { var gotItem = localStorage.getItem(data.getItem); event.source.postMessage( '#localStorage#' + data.identifier + (gotItem === null ? 'null#' : '#' + gotItem), event.origin ); } else if ('removeItem' in data) { localStorage.removeItem(data.removeItem); } } }, false);&lt;/script&gt;
* 注意Safari一下，会报错：

• Blocked a frame with origin “http://localhost:10001” from accessing a frame with origin “http://localhost:10000“. Protocols, domains, and ports must match.作者：Java程序员聚集地链接：https://juejin.im/post/6844903971144859661来源：掘金著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### 方式六：修改document.domain跨子域

* 前提条件：这两个域名必须属于同一个基础域名!而且所用的协议，端口都要一致，否则无法利用document.domain进行跨域，所以只能跨子域
* 在根域范围内，允许把domain属性的值设置为它的上一级域。例如，在”aaa.xxx.com”域内，可以把domain设置为 “xxx.com” 但不能设置为 “XXX.com Porn - Free PORNO videos” 或者”com”。
* 现在存在两个域名aaa.xxx.com和bbb.xxx.com。在aaa下嵌入bbb的页面，由于其document.name不一致，无法在aaa下操作bbb的js。可以在aaa和bbb下通过js将document.name = 'xxx.com';设置一致，来达到互相访问的作用

### 方式七：WebSocket

* WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信，同时允许跨域通讯，是server push技术的一种很棒的实现。相关文章，请查看：WebSocket、WebSocket-SockJS
* 需要注意：WebSocket对象不支持DOM 2级事件侦听器，必须使用DOM 0级语法分别定义各个事件。

### 方式八：代理

* 同源策略是针对浏览器端进行的限制，可以通过服务器端来解决该问题
* DomainA客户端（浏览器） ==&gt; DomainA服务器 ==&gt; DomainB服务器 ==&gt; DomainA客户端（浏览器）

## 说一下 JSONP 实现原理？

### jsonp 即 json+padding，动态创建script标签，利用script标签的src属性可以获取任何域下的js脚本，通过这个特性(也可以说漏洞)，服务器端不在返货json格式，而是返回一段调用某个函数的js代码，在src中进行了调用，这样实现了跨域。

# 容器

## java 容器都有哪些？

### Collection

* List

• Vector

• Stack

• ArrayList

• LinkedList

* Queue

• LinkedList

• PriorityQueue

* Set

• HashSet

• LinkHashSet

• TreeSet

### Map

* HashMap
* TreeMap

## Collection 和 Collections 有什么区别？

### java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

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

## 说一下 HashMap 的实现原理？

### HashMap概述

* HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

### HashMap的数据结构

* 在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

### 当我们往Hashmap中put元素时,首先根据key的hashcode重新计算hash值,根绝hash值得到这个元素在数组中的位置(下标),如果该数组在该位置上已经存放了其他元素,那么在这个位置上的元素将以链表的形式存放,新加入的放在链头,最先加入的放入链尾.如果数组中该位置没有元素,就直接将该元素放到数组的该位置上。

### 需要注意Jdk 1.8中对HashMap的实现做了优化,当链表中的节点数据超过八个之后,该链表会转为红黑树来提高查询效率,从原来的O(n)到O(logn)

## 说一下 HashSet 的实现原理？

### HashSet底层由HashMap实现

### HashSet的值存放于HashMap的key上

### HashMap的value统一为PRESENT

## List、Set、Map 之间的区别是什么？

### 



## HashMap 和 Hashtable 有什么区别？

### hashMap去掉了HashTable 的contains方法，但是加上了containsValue（）和containsKey（）方法。

### hashTable同步的，而HashMap是非同步的，效率上逼hashTable要高。

### hashMap允许空键值，而hashTable不允许。

## HashMap 还是 TreeMap 分别适用于那些场景？

### 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。

## ArrayList 和 LinkedList 的区别是什么？

### 最明显的区别是 ArrrayList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)。

## 怎么快速在数组和 List 之间转换？

### List转换成为数组

* 调用ArrayList的toArray方法

### 数组转换成为List

* 调用Arrays的asList方法。

## Array 和 ArrayList 有何区别？

### 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

## 在 Queue 中 poll()和 remove()有什么区别？

### 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。

## 哪些集合类是线程安全的？

### vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

### statck：堆栈类，先进后出。

### hashtable：就比hashmap多了个线程安全。

### enumeration：枚举，相当于迭代器。

## 迭代器 Iterator 是什么？

### 迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。

## Iterator 怎么使用？有什么特点？

### Iterator 怎么使用？

* (1) 使用方法iterator()要求容器返回一个Iterator。第一次调用Iterator的next()方法时，它返回序列的第一个元素。注意：iterator()方法是java.lang.Iterable接口,被Collection继承。
* (2) 使用next()获得序列中的下一个元素。
* (3) 使用hasNext()检查序列中是否还有元素。
* (4) 使用remove()将迭代器新返回的元素删除。

### 有什么特点？

* Java中的Iterator功能比较简单，并且只能单向移动

### Iterator是Java迭代器最简单的实现，为List设计的ListIterator具有更多的功能，它可以从两个方向遍历List，也可以从List中插入和删除元素。

## Iterator 和 ListIterator 有什么区别？

### 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。

# 多线程与并发

## 并行和并发有什么区别？

### 并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。

### 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。

### 在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群。

### 所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

## 线程和进程的区别？

### 简而言之，进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。线程是进程的一个实体，是cpu调度和分派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

## 守护线程是什么？

### 守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程。

## 创建线程有哪几种方式？

### ①. 继承Thread类创建线程类

* 定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。
* 创建Thread子类的实例，即创建了线程对象。
* 调用线程对象的start()方法来启动该线程。

### ②. 通过Runnable接口创建线程类

* 定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
* 创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
* 调用线程对象的start()方法来启动该线程。

### ③. 通过Callable和Future创建线程

* 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
* 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
* 使用FutureTask对象作为Thread对象的target创建并启动新线程。
* 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

## 哪些集合类是线程安全的？

### vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

### statck：堆栈类，先进后出。

### hashtable：就比hashmap多了个线程安全。

### enumeration：枚举，相当于迭代器。

## 说一下 runnable 和 callable 有什么区别？

### 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；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。

## 线程有哪些状态？

### 线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

* 创建状态

• 在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。

* 就绪状态

• 当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

* 运行状态

• 线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。

* 阻塞状态

• 线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep,suspend，wait等方法都可以导致线程阻塞。

* 死亡状态

• 如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪

## sleep() 和 wait() 有什么区别？

### sleep()：方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

### wait()：wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程。

## notify() 和 notifyAll()有什么区别？

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

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

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

## 线程的 run()和 start()有什么区别？

### 每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，方法run()称为线程体。通过调用Thread类的start()方法来启动一个线程。

### start()方法来启动一个线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码； 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行状态， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

### run()方法是在本线程里的，只是线程里的一个函数,而不是多线程的。 如果直接调用run(),其实就相当于是调用了一个普通函数而已，直接待用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

## 创建线程池有哪几种方式？

### ①. newFixedThreadPool(int nThreads)

* 创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

### ②. newCachedThreadPool()

* 创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

### ③. newSingleThreadExecutor()

* 这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

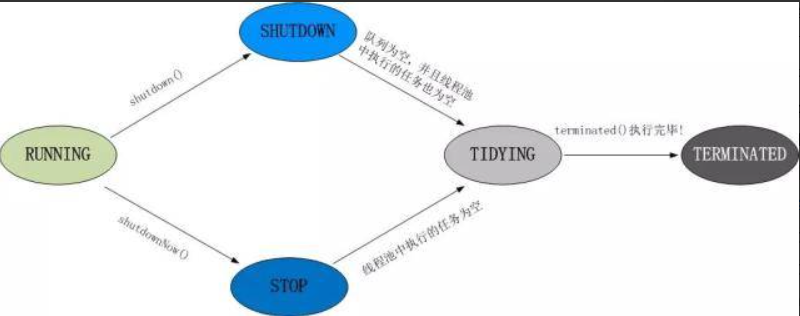
### ④. newScheduledThreadPool(int corePoolSize)

* 创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

## 线程池都有哪些状态？

### 线程池有5种状态：Running、ShutDown、Stop、Tidying、Terminated。

### 线程池各个状态切换框架图：



## 线程池中 submit()和 execute()方法有什么区别？

### 接收的参数不一样submit有返回值，而execute没有submit方便Exception处理

## 在 java 程序中怎么保证多线程的运行安全？

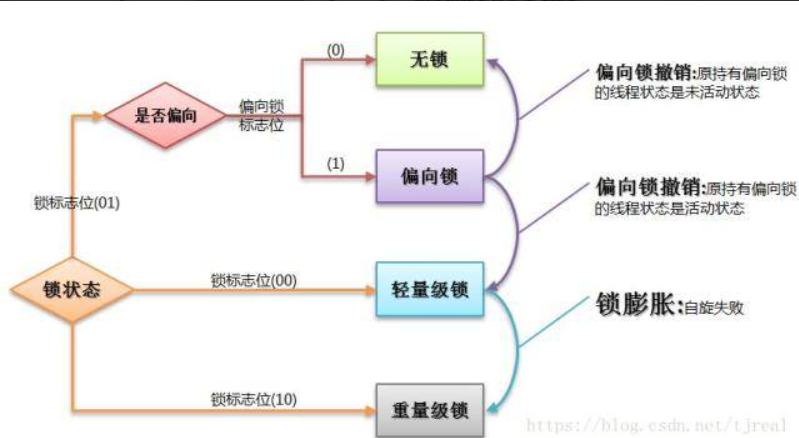
### 线程安全在三个方面体现：

* 原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，（atomic,synchronized）；
* 可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized,volatile）；
* 有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before原则）

## 多线程锁的升级原理是什么？

### 在Java中，锁共有4种状态，级别从低到高依次为：无状态锁，偏向锁，轻量级锁和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级。

### 锁升级的图示过程：



## 什么是死锁？

### 死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。是操作系统层面的一个错误，是进程死锁的简称，最早在 1965 年由 Dijkstra 在研究银行家算法时提出的，它是计算机操作系统乃至整个并发程序设计领域最难处理的问题之一。

## 多线程产生死锁的4个必要条件？

### 互斥条件：

* 进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

### 请求和保持条件：

* ：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放

### 不可剥夺条件：

* 是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

### 环路等待条件：

* 是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

## 怎么防止死锁？

### 死锁的四个必要条件：

* 互斥条件

• 互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

* 请求和保持条件

• 请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放

* 不可剥夺条件

• 不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

* 环路等待条件

• 环路等待条件：是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

### 这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之 一不满足，就不会发生死锁。

### 理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和 解除死锁。

### 所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确 定资源的合理分配算法，避免进程永久占据系统资源。

### 此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

## ThreadLocal 是什么？有哪些使用场景？

### 线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

## 说一下 synchronized 底层实现原理？

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

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

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

## 说一下 volatile 的实现原理

## 说一下 atomic 的原理？

### Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

### Atomic系列的类中的核心方法都会调用unsafe类中的几个本地方法。我们需要先知道一个东西就是Unsafe类，全名为：sun.misc.Unsafe，这个类包含了大量的对C代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过unsafe分配内存的时候，如果自己指定某些区域可能会导致一些类似C++一样的指针越界到其他进程的问题。

## synchronized 关键字

### synchronized 底层实现原理

* synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。
* Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

• 普通同步方法，锁是当前实例对象

• 静态同步方法，锁是当前类的class对象

• 同步方法块，锁是括号里面的对象

## synchronized 和 volatile 的区别是什么？

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

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

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

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

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

## synchronized 和 Lock 有什么区别？

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

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

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

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

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

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

## synchronized 和 ReentrantLock 区别是什么？

### 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：

### ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁ReentrantLock可以获取各种锁的信息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。

## 说一下 runnable 和 callable 有什么区别？

### 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；

### 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。

## 启动一个线程是调用 run() 还是 start() 方法？

### 启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

## 线程池中 submit( )和 execute() 方法有什么区别？

### 接收的参数不一样

### submit有返回值，而execute没有

### submit方便Exception处理

## 多线程锁的升级原理是什么？

### 锁的升级的目的

* 锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

### synchronized 锁升级原理

* 在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

### 偏向锁

* 顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁／解锁的一些CAS操作（比如等待队列的一些CAS操作），这种情况下，就会给线程加一个偏向锁。 如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

### 轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，轻量级锁就会升级为重量级锁；

### 重量级锁是synchronized ，是 Java 虚拟机中最为基础的锁实现。在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。

## 讲一下常见的 GC 算法？

### 什么是堆？

* 堆指用于动态（即执行程序时）存放对象的内存空间。而这个对象，在面向对象的编程中，它指“具有属性和行为的事物”，然而在GC的世界中，对象表示的是“通过应用程序利用的数据的集合”。具体到Java堆，它是所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。（此处的对象实例可以理解为前面所说的对象，因为不仅仅Java有自动的GC，python、JavaScript等语言也有，所以在广义上说对象是更好的表述，当然，Java的数组也是分配在堆上的）。

### GC算法的评判标准

* GC算法的评判标准主要是以下4点：

• 吞吐量：即单位时间内的处理能力。

• 最大暂停时间：因执行GC而暂停执行程序所需的时间。

• 堆的使用效率：鱼与熊掌不可兼得，堆使用效率和吞吐量、最大暂停时间是不可能同时满足的。即可用的堆越大，GC运行越快；相反，想要利用有限的堆，GC花费的时间就越长。

• 访问的局部性：在存储器的层级构造中，我们知道越是高速存取的存储器容量会越小（具体可以参看我写的存储器那篇文章）。由于程序的局部性原理，将经常用到的数据放在堆中较近的位置，可以提高程序的运行效率。

### 常用的GC算法

* 引用计数法

• 什么是引用计数法？

• 所谓的引用计数法就是给每个对象一个引用计数器，每当有一个地方引用它时，计数器就会加1；当引用失效时，计数器的值就会减1；任何时刻计数器的值为0的对象就是不可能再被使用的。

• 这个引用计数法时没有被Java所使用的，但是python有使用到它。而且最原始的引用计数法没有用到GC Roots。

• 优点

• 可即时回收垃圾：在该方法中，每个对象始终知道自己是否有被引用，当被引用的数值为0时，对象马上可以把自己当作空闲空间链接到空闲链表。

• 最大暂停时间短。

• 没有必要沿着指针查找

• 缺点

• 计数器值的增减处理非常繁重

• 计算器需要占用很多位

• 实现繁琐

• 循环引用无法回收

* 标记-清除法

• 什么是标记-清除算法？

• 该算法分为标记和清除两个阶段。标记就是把所有活动对象都做上标记的阶段；清除就是将没有做上标记的对象进行回收的阶段。如下图所示。

• 优点

• 实现简单

• 与保守式GC算法兼容（保守式GC在后面介绍）

• 缺点

• 碎片化：如上图所示，在回收的过程中会产生被细化的分块，到后面，即使堆中分块的总大小够用，但是却因为分块太小而不能执行分配。

• 分配速度：因为分块不是连续的，因此每次分块都要遍历空闲链表，找到足够大的分块，从而造成时间的浪费。

• 与写时复制技术不兼容：所谓写时复制就是fork的时候，内存空间只引用而不复制，只有当该进程的数据发生变化时，才会将数据复制到该进程的内存空间。这样，当两个进程中的内存数据相同的时候，就能节约大量的内存空间了。而对于标记-清除算法，它的每个对象都有一个标志位来表示它是否被标记，在每一次运行标记-清除算法的时候，被引用的对象都会进行标记操作，这个仅仅标记位的改变，也会变成对象数据的改变，从而引发写时复制的复制过程，与写时复制的初衷就背道而驰了。

* 复制算法

• 什么是复制算法？

• 复制算法就是将内存空间按容量分成两块。当这一块内存用完的时候，就将还存活着的对象复制到另外一块上面，然后把已经使用过的这一块一次清理掉。这样使得每次都是对半块内存进行内存回收。内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶的指针，按顺序分配内存即可，实现简单，运行高效。

• 优点

• 优秀的吞吐量

• 可实现高速分配：复制算法不用使用空闲链表。这是因为分块是连续的内存空间，因此，调用这个分块的大小，只需要这个分块大小不小于所申请的大小，移动指针进行分配即可。

• 不会发生碎片化。

• 与缓存兼容。

• 缺点

• 堆的使用效率低下。

• 不兼容保守式GC算法。

• 递归调用函数。

* 标记-清除算法

• 什么是标记-压缩算法？

• 标记-压缩算法与标记-清理算法类似，只是后续步骤是让所有存活的对象移动到一端，然后直接清除掉端边界以外的内存。

• 优缺点

• 该算法可以有效的利用堆，但是压缩需要花比较多的时间成本。

### 可达性

* 所谓的可达性就是通过一系列称为“GC Roots”的对象为起点，从这些节点开始向下搜索，搜索走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是GC Roots到这个对象不可达）时，则说明此对象是不可用的。如下图所示，ABC可达，DE不可达。
* 那么那些对象可以作为GC Roots呢？以Java为例，有以下几种：

• 栈（栈帧中的本地变量表）中引用的对象。

• 方法区中的静态成员。

• 方法区中的常量引用的对象（全局变量）。

• 本地方法栈中JNI（一般说的Native方法）引用的对象。

• 注：第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

### 保守式GC与准确式GC

* 保守式GC

• 所谓保守式GC就是“不能识别指针和非指针的GC”。

• 对于寄存器、调用栈、全局变量空间来说，都是不明确的根。例如调用栈中，装着函数内的局部变量和参数值。而局部变量，如C语言中的int、double这样就是非指针，但是也会有像void\*这样的指针。

• 那么保守式GC会怎么检查不明确的根呢？1、是不是被正确对齐的值？（在32位CPU的情况下，为4的倍数）2、是不是指着堆内？3、是不是指向对象的开头？当然，这些只是基本的检查项目。

• 上面的检查方法会将一些非指针识别成指针。例如一个数值和一个地址，它们两个值相等，这个时候，那个值也可以被识别成指针

• 保守式GC的优点是语言处理程序不依赖与GC。缺点为识别指针和非指针需要付出成本、错误识别指针会压迫堆、能够使用的GC算法有限。例如GC复制算法就不能使用，因为其可能会将非指针重写。

* 准确式GC

• 准确式GC能够正确识别指针和非指针的GC。正确的根的创建方法是依赖于语言处理程序的实现的。我们可以通过打标签、不把寄存器和栈等当作根的方法来实现。

• 其优点就是完全能够识别指针，能够使用复制算法等需要移动对象的算法。但是在创建准确式GC时，语言处理程序必须对GC进行一些支援，而且创建正确的根就必须付出一定的代价。

• 其实我们垃圾回收机的实现都不是仅仅用哪一种回收算法，都是将几个结合使用，特别是分代算法，后面我们会详细的介绍。

## 讲一下常见的垃圾收集器？

### 垃圾回收器是什么

* JVM垃圾收集器发展历程大致可以分为以下四个阶段： Serial（串行）收集器 -&gt; Parallel(并行)收集器 -&gt; CMS（并发）收集器 -&gt; G1（并发）收集器

### Serial类收集器

* Serial类收集器是一个单线程的收集器:

• 它只会用单个收集线程去进行垃圾回收的工作

• 它在进行垃圾收集的时候会“Stop The World”暂停其他所有的工作表线程，直到它收集结束

• Serial收集器采取复制算法在新生代进行单线程的回收工作

• Serial Old收集器采取标记-整理算法在老年代进行单线程的回收工作

### Parallel类收集器

* Parallel类收集器就是Serial收集器的多线程版本：

• 它使用多个收集线程取进行垃圾回收工作

• 它在进行垃圾收集的时候也会“Stop The World”暂停其他所有的工作表线程，直到它收集结束

• ParNew收集器采取复制算法在新生代进行多线程的回收工作

• Parallel Scavenge收集器也是一个新生代收集器，不过它被称为“吞吐量优先”收集器，它提供了2个能精确控制吞吐量的参数：

• -XX : MaxGCPauseMillis：控制最大垃圾收集停顿时间

• -XX : GCTimeRatio : 直接设置吞吐量大小，垃圾收集时间占总时间的比率

• Parallel Scavenge收集器还有一个开关参数-XX: UseAdaptiveSizePolicy，打开这个开关后就不用手动指定新生代的大小(-Xmn),Eden与Survivor区的比例(-XX:SurvivorRatio)等细节参数了，JVM会动态调整这些参数已提供最合适的停顿时间或者最大吞吐量。

• Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法在老年代进行垃圾回收

### CMS收集器

* CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。它是一个基于标记-清除算法实现的，运作过程分为4个步骤：

• 初始标记(CMS initial mark): 需要“Stop The World”，仅仅只是标记下GC Roots能直接关联到的对象，速度很快

• 并发标记(CMS concurrent mark): CMS线程与应用线程一起并发执行，从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长

• 重新标记(CMS remark)：重新标记就是为了修正并发标记期间因用户线程继续运作而导致标记产生变动的那一部分对象的标记记录，可以多线程并行

• 并发清除(CMS concurrent sweep)：CMS线程与应用线程一起并发执行，进行垃圾清除

* CMS收集器优点

• 并发收集、低停顿

* CMS的三个明显的缺点

• CMS收集器对CPU的资源非常敏感。CPU的数量较少不足4个（比如2个）时，CMS对用户程序的影响就可能变的很大。

• CMS收集器无法处理浮动垃圾(Floating Carbage)，可能出现"Concurrent Mode Failture"失败而导致产生另一次Full GC的产生。浮动垃圾就是并发清理阶段，用户线程产生的新垃圾

• CMS是基于标记-清除算法的，收集结束后会有大量的空间碎片，就可能会在老年代中无法分配足够大的连续空间而不得不触发另一次Full GC。

### G1收集器

* 介绍

• 同优秀的CMS一样，G1也是关注最小停顿时间的垃圾回收器，也同样适合大尺寸堆内存，官方也推荐用G1来代替选择CMS。

• G1收集器的最大特点就是引入了分区的思路，弱化了分代的概念G1从整体来看是基于标记-整理算法实现的，从局部(两个Region之间)来看是基于复制算法实现的

* G1相对于CMS的改进

• G1是基于标记-整理算法，不会产生空间碎片，在分配大的连续对象是不会因为无法得到连续空间而不得不提前触发一次Full GC

• 停顿时间可控，G1可以通过设置停顿时间来控制垃圾回收时间

• 并行与并发，G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间

* G1与CMS的区别

• 堆内存模型的不同

• G1之前的JVM堆内存模型，堆被分为新生代，老年代，永久代(1.8之前，1.8之后是元空间)，新生代中又分为Eden和两个Survivor区。

• G1收集器的堆内存模型，堆被分为很多个大小连续的区域（Region），Region的大小可以通过-XX: G1HeapRegionSize参数指定，大小区间为[1M，32M]。

• 每个Region被标记了E、S、O和H，这些区域在逻辑上被映射为Eden，Survivor，老年代和巨型区(Humongous Region)。巨型区域是为了存储超过50%标准region大小的巨型对象。

• 应用分代的不同

• G1可以在新生代和老年代使用，而CMS只能在老年代使用。

• 收集算法的不同

• G1是复制+标记-整理算法，CMS是标记清除算法。

* G1收集器的工作流程

• G1收集器的工作流程大致分为如下几个步骤：

• 初始标记(Initial Marking): 需要“Stop The World”，仅仅只是标记下GC Roots能直接关联到的对象，速度很快

• 并发标记(Concurrent Marking): G1线程与应用线程一起并发执行，从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长

• 最终标记(Final Marking): 最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，需要“Stop The World”，可以多线程并行

• 筛选回收(Live Data Counting and Evacuation): 对各个Region的回收价值和成本进行排序，根据用户所期待的GC停顿时间制定回收计划。具体地，在后台维护一个优先队列，每次根据允许的收集停顿时间，优先回收价值最大的Region。

* G1的GC模式

• G1提供了两种GC模式，Young GC和Mixed GC，两种都是完全Stop The World的

• YoungGC

• 在分配一般对象(非巨型对象)时，当所有的Eden Region使用达到最大阈值并且无法申请到足够内存时，会触发一次YoungGC。

• 每次YoungGC会回收所有的Eden以及Survivor区，并且将存活对象复制到Old区以及另一部分的Survivor。

• MixedGC

• 当越来越多的对象晋升到老年代old region时，为了避免堆内存被耗尽，虚拟机会触发一次mixed gc，该算法并不是一个old gc，除了回收整个young region，还会回收一部分的old region。这里需要注意：是一部分老年代，而不是全部老年代，可以选择哪些old region进行收集，从而可以对垃圾回收的耗时时间进行控制

• G1没有fullGC概念，需要fullGC时，调用serialOldGC进行全堆扫描（包括eden、survivor、o、perm）。

## sleep() 和 wait(n)、wait() 的区别：

### sleep()：

* 方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

### wait()

* wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程。

## ThreadLocal实现原理？

### ThreadLocal 是什么？有哪些使用场景？

* 线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

# 反射

## 什么是反射？

### 反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

### Java反射：

* 在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

### Java反射机制主要提供了以下功能：

* 在运行时判断任意一个对象所属的类
* 在运行时构造任意一个类的对象。
* 在运行时判断任意一个类所具有的成员变量和方法。
* 在运行时调用任意一个对象的方法。

## 什么是 java 序列化？什么情况下需要序列化？

### 什么是 java 序列化？

* 简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存object states，但是Java给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。

### 什么情况下需要序列化？

* a）当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；
* b）当你想用套接字在网络上传送对象的时候；
* c）当你想通过RMI传输对象的时候；

## 动态代理是什么？有哪些应用？

### 动态代理是什么？

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

### 动态代理的应用：

* Spring的AOP
* 加事务
* 加权限
* 加日志

## 怎么实现动态代理？

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

# Spring系列

## @RequestMapping 的作用是什么？

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

### RequestMapping注解有六个属性，下面我们把她分成三类进行说明。

* value， method：

• value：指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

• method：指定请求的method类型， GET、POST、PUT、DELETE等；

* consumes，produces

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

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

* params，headers

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

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

## @Autowired 的作用是什么？

## @Colltroller 和 @Component 的区别是什么？

## @Component 和 @Bean 的区别是什么？

### 作用对象不同：@Component作用于类，@Bean作用于方法。

### @Component通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（使用@ComponentScan注解定义要扫描的路径从中找出识别了需要装配的类自动装配到spring的Bean容器中）。@Bean注解通常是在标有该注解的方法中定义产生这个bean，@Bean告诉Spring这是某个类的实例，当我需要用它的时候还给我。

### @Bean注解比@Component注解的自定义性更强，而且很多地方只能通过@Bean注解来注册Bean，比如第三方库中的类。

## jpa 和 hibernate 有什么区别？

## spring

### Spring概述

* 什么是 Spring 框架?

• spring是开源的轻量级框架

• 简单来说，Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

* Spring的俩大核心概念

• 解释一下什么是 ioc？

• 首先来解释下容器：在每个框架中都有个容器的概念，所谓的容器就是将常用的服务封装起来，然后用户只需要遵循一定的规则就可以达到统一、灵活、安全、方便和快速的目的。

• 然后IOC容器是具有依赖注入功能的容器，负责实例化、定位、配置应用程序中的对象以及建立这些对象间的依赖。

• 解释一下什么是 aop？

• Aop（面向切面编程）能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的扩展性和可维护性。

* Spring框架的设计目标，设计理念，和核心是什么

• Spring设计目标：Spring为开发者提供一个一站式轻量级应用开发平台；Spring设计理念：在JavaEE开发中，支持POJO和JavaBean开发方式，使应用面向接口开发，充分支持OOP（面向对象）设计方法；Spring通过IOC容器实现对象耦合关系的管理，并实现依赖反转，将对象之间的依赖关系交给IOC容器，实现解耦；Spring框架的核心：IOC容器和AOP模块。通过IOC容器管理POJO对象以及他们之间的耦合关系；通过AOP以动态非侵入的方式增强服务。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。

* Spring的优缺点是什么？

• 优点

• 方便解耦，简化开发Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。AOP编程的支持Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。声明式事务的支持只需要通过配置就可以完成对事务的管理，而无需手动编程。方便程序的测试Spring对Junit4支持，可以通过注解方便的测试Spring程序。方便集成各种优秀框架Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。降低JavaEE API的使用难度Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

• 缺点

• Spring明明一个很轻量级的框架，却给人感觉大而全Spring依赖反射，反射影响性能使用门槛升高，入门Spring需要较长时间

* Spring有哪些应用场景

• 应用场景：JavaEE企业应用开发，包括SSH、SSM等Spring价值：Spring是非侵入式的框架，目标是使应用程序代码对框架依赖最小化；Spring提供一个一致的编程模型，使应用直接使用POJO开发，与运行环境隔离开来；Spring推动应用设计风格向面向对象和面向接口开发转变，提高了代码的重用性和可测试性；

* spring 有哪些主要模块？

• 核心容器

• 数据访问/集成,

• Web

• AOP（面向切面编程

• 工具

• 消息

• 测试

* 为什么要使用 spring？不用不行吗？还可以用什么？

• 1.简介

• 目的：解决企业应用开发的复杂性

• 功能：使用基本的JavaBean代替EJB，并提供了更多的企业应用功能

• 范围：任何Java应用

• 简单来说，Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

• 2.轻量

• 从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的处理开销也是微不足道的。此外，Spring是非侵入式的：典型地，Spring应用中的对象不依赖于Spring的特定类。

• 3.控制反转

• Spring通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为IoC与JNDI相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

• 4.面向切面

• Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。

• 5.容器

• Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置原型（prototype），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring不应该被混同于传统的重量级的EJB容器，它们经常是庞大与笨重的，难以使用。

• 6.框架

• Spring可以将简单的组件配置、组合成为复杂的应用。在Spring中，应用对象被声明式地组合，典型地是在一个XML文件里。Spring也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

• 所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

* 使用Spring框架有什么好处呢？

• 框架能更让我们高效的编程以及更方便的维护我们的系统。

• 轻量：Spring是轻量的，相对其他框架来说。

• 控制反转：Spring通过控制反转实现了松散耦合，对象给出他们的依赖，而不是创建或查找依赖的对象们

• 面向切面编程（AOP）：Spring支持面向切面编程，并且把业务逻辑和系统服务分开。

• 容器：Spring包含并管理应用中对象的生命周期和配置

• 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

• 事务管理：Spring提供一个持续的事务管理接口，提供声明式事务和编程式事务。

• 异常处理:Spring提供方便的API把具体技术相关的异常转化为一致的unchecked异常。

* 你第二点提到了spring的控制反转，能解释下吗

• 控制反转

• 首先来解释下控制反转。控制反转(Inversion Of Control，缩写为IOC)是一个重要的面向对象编程的法则来削减程序的耦合问题，也是spring框架的核心。应用控制反转，对象在被创建的时候，由一个调控系统内的所有对象的外界实体，将其所依赖的对象的引用，传递给它。也可以说，依赖被注入到对象中。所以，控制反转是关于一个对象如何获取他所依赖的对象的引用，这个责任的反转。另外，控制反转一般分为两种类型，依赖注入（Dependency Injection，简称DI）和依赖查找(Dependency Lookup)。依赖注入应用比较广泛。

• 还有几个常见的问题:

• 谁依赖谁-当然是应用程序依赖于IOC容器。

• 为什么需要依赖-应用程序需要IOC容器来提供对象需要的外部资源。

• 谁注入谁-很明显是IOC容器注入应用程序某个对象，应用程序依赖的对象

• 注入了什么-就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）

* Spring 框架中都用到了哪些设计模式？

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

* 详细讲解一下核心容器（spring context应用上下文) 模块

• 这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为一个容器。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真正的应用代码中分离。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

* 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：上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。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。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。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。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。

* Spring 应用程序有哪些不同组件？

• Spring 应用一般有以下组件：

• 接口 - 定义功能。Bean 类 - 它包含属性，setter 和 getter 方法，函数等。Bean 配置文件 - 包含类的信息以及如何配置它们。Spring 面向切面编程（AOP） - 提供面向切面编程的功能。用户程序 - 它使用接口。

* 使用 Spring 有哪些方式？

• 使用 Spring 有以下方式：

• 作为一个成熟的 Spring Web 应用程序。作为第三方 Web 框架，使用 Spring Frameworks 中间层。作为企业级 Java Bean，它可以包装现有的 POJO（Plain Old Java Objects）。用于远程使用。

### Spring控制反转(IOC)

* 什么是Spring IOC 容器？

• 控制反转即IOC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

* 控制反转(IOC)有什么作用

• 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的解耦，由容器去维护具体的对象托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

* IOC的优点是什么？

• IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和JNDI查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC容器支持加载服务时的饿汉式初始化和懒加载。

* Spring IOC 的实现机制

• Spring 中的 IOC 的实现原理就是工厂模式加反射机制。

• 示例：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[] a) { Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple"); if(f!=null){ f.eat(); } }}

* Spring 的 IOC支持哪些功能

• Spring 的 IOC 设计支持以下功能：其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring RuntimeBeanReference 对象。对于 IOC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。

* 那IOC与new对象有什么区别吗

• 这就是正转与反转的区别。传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转。而反转则是容器来帮助我们创建并注入依赖对象。

* 那IOC有什么优缺点吗？

• 优点

• 很明显，实现了组件之间的解耦，提高程序的灵活性和可维护性

• 缺点

• 对象生成因为是反射编程，在效率上有些损耗。但相对于IOC提高的维护性和灵活性来说，这点损耗是微不足道的，除非某对象的生成对效率要求特别高。

* 那你能说下IOC容器是怎么工作的吗？

• 首先说下两个概念。

• Bean的概念：Bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没什么区别了。

• 元数据BeanDefinition：确定如何实例化Bean、管理bean之间的依赖关系以及管理bean，这就需要配置元数据，在spring中由BeanDefinition代表。

• 工作原理

• 准备配置文件：配置文件中声明Bean定义也就是为Bean配置元数据。

• 由IOC容器进行解析元数据：IOC容器的Bean Reader读取并解析配置文件，根据定义生成BeanDefinition配置元数据对象，IOC容器根据BeanDefinition进行实例化、配置以及组装Bean。

• 实例化IOC容器：由客户端实例化容器，获取需要的Bean。

• 下面举个例子：

• @Test public void testHelloWorld() { //1、读取配置文件实例化一个IoC容器 ApplicationContext context = new ClassPathXmlApplicationContext("helloworld.xml"); //2、从容器中获取Bean，注意此处完全“面向接口编程，而不是面向实现” HelloApi helloApi = context.getBean("hello", HelloApi.class); //3、执行业务逻辑 helloApi.sayHello(); }

* 那你知道BeanFactory和ApplicationContext的区别吗？

• BeanFactory是spring中最基础的接口。它负责读取读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的生命周期。

• ApplicationContext是BeanFactory的子接口，除了提供上述BeanFactory的所有功能外，还提供了更完整的框架功能：如国际化支持，资源访问，事件传递等。常用的获取ApplicationContext的方法：2.1 FileSystemXmlApplicationContext：从文件系统或者url指定的xml配置文件创建，参数为配置文件名或者文件名数组。2.2 ClassPathXmlApplicationContext：从classpath的xml配置文件创建，可以从jar包中读取配置文件2.3 WebApplicationContextUtils：从web应用的根目录读取配置文件，需要先在web.xml中配置，可以配置监听器或者servlet来实现。

• ApplicationContext的初始化和BeanFactory有一个重大区别：BeanFactory在初始化容器时，并未实例化Bean，知道第一次访问某个Bean时才实例化Bean；而ApplicationContext则在初始化应用上下文时就实例化所有的单例Bean，因此ApplicationContext的初始化时间会比BeanFactory稍长一些。

* Spring 如何设计容器的，BeanFactory和ApplicationContext的关系详解

• Spring 作者 Rod Johnson 设计了两个接口用以表示容器。BeanFactoryApplicationContextBeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 “低级容器”。ApplicationContext 可以称之为 “高级容器”。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。当然，除了这两个大接口，还有其他的辅助接口，这里就不介绍他们了。BeanFactory和ApplicationContext的关系为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的 ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。有点复杂？ 先不要慌，我来解释一下。最上面的是 BeanFactory，下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）通常用户看到的就是 “高级容器”。 但 BeanFactory 也非常够用啦！左边灰色区域的是 “低级容器”， 只负载加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置，生命周期事件回调等。

• 小结

• 说了这么多，不知道你有没有理解Spring IOC？ 这里小结一下：IOC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：加载配置文件，解析成 BeanDefinition 放在 Map 里。调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。上面就是 Spring 低级容器（BeanFactory）的 IOC。至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IOC。他支持不同信息源头，支持 BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

* ApplicationContext通常的实现是什么？

• FileSystemXmlApplicationContext ：此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。ClassPathXmlApplicationContext：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。WebXmlApplicationContext：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

* 什么是Spring的依赖注入？

• 控制反转IOC是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：依赖注入和依赖查找依赖注入：相对于IOC而言，依赖注入(DI)更加准确地描述了IOC的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。

* 依赖注入的基本原则

• 依赖注入的基本原则是：应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由IOC容器负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给IOC容器负责。容器全权负责组件的装配，它会把符合依赖关系的对象通过属性（JavaBean中的setter）或者是构造器传递给需要的对象。

* 依赖注入有什么优势

• 依赖注入之所以更流行是因为它是一种更可取的方式：让容器全权负责依赖查询，受管组件只需要暴露JavaBean的setter方法或者带参数的构造器或者接口，使容器可以在初始化时组装对象的依赖关系。其与依赖查找方式相比，主要优势为：查找定位操作与应用代码完全无关。不依赖于容器的API，可以很容易地在任何容器以外使用应用对象。不需要特殊的接口，绝大多数对象可以做到完全不必依赖容器。

* 有哪些不同类型的依赖注入实现方式？

• 依赖注入是时下最流行的IOC实现方式，依赖注入分为接口注入（Interface Injection），Setter方法注入（Setter Injection）和构造器注入（Constructor Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从Spring4开始已被废弃。构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

* 构造器依赖注入和 Setter方法注入的区别

• 构造函数注入 setter 注入没有部分注入 有部分注入不会覆盖 setter 属性 会覆盖 setter 属性任意修改都会创建一个新实例 任意修改不会创建一个新实例适用于设置很多属性 适用于设置少量属性

• 两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

### 那你知道spring aop的原理吗？

* spring aop就是基于动态代理的，如果要代理的对象实现了某个接口，那么spring aop会使用jdk proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用jdk的动态代理，这时spring aop会使用cglib动态代理，这时候spring aop会使用cglib生成一个被代理对象的子类作为代理。

### Spring Beans

* 什么是Spring beans？

• Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中 的形式定义。

* 一个 Spring Bean 定义 包含什么？

• 一个Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个bean，它的生命周期详情及它的依赖。

* 如何给Spring 容器提供配置元数据？Spring有几种配置方式

• 这里有三种重要的方法给Spring 容器提供配置元数据。XML配置文件。基于注解的配置。基于java的配置。

* Spring配置文件包含了哪些信息

• Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

* Spring基于xml注入bean的几种方式

• Set方法注入；构造器注入：通过index设置参数的位置；通过type设置参数类型；静态工厂注入；实例工厂；

* 你怎样定义类的作用域？

• 当定义一个 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

* 解释Spring支持的几种bean的作用域

• Spring框架支持以下五种bean的作用域：

• singleton : bean在每个Spring ioc 容器中只有一个实例。prototype：一个bean的定义可以有多个实例。request：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。session：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。global-session：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

• 注意： 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

* spring 中的 bean 是线程安全的吗？

• 不是线程安全的

* Spring中的单例Bean的线程安全问题了解吗？

• 大部分时候我们并没有在系统中使用多线程。单例Bean存在线程安全问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。常见的有两种解决方法：

• 在Bean中尽量避免定义可变的成员变量（不太现实）

• 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在Threadlocal中。

* 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。

* Spring中的Bean的生命周期你了解吗？

• Bean容器找到配置文件中Spring Bean的定义。

• Bean容器利用Java反射机制创建一个Bean的实例。

• 如果涉及一些属性值，利用set()方法设置一些属性值。

• 如果Bean实现了BeanNameAware接口，调用setBeanName（）方法，传入Bean的名称。

• 如果Bean实现了BeanClassLoaderAware接口，调用setBeanClassLoader（）方法，传入ClassLoader对象的实例。

• 如果Bean实现了BeanFactoryAware接口，调用setBeanClassLoader()方法，传入ClassLoader对象的实例。

• 与上面类似，如果实现了其他\*.Aware接口，就调用相应的方法。

• 如果有和加载这个Bean的Spring容器相关的BeaPostProcessor对象，执行postProcessBeforeInitialization（）方法

• 如果Bean实现了InitializingBean接口，执行afterPropertiesSet（）方法

• 如果Bean在配置文件中的定义包含init-method属性，执行指定的方法。

• 如果有和加载这个 Bean的 Spring 容器相关的 BeanPostProcessor 对象，执行postProcessAfterInitialization() 方法

• 当要销毁Bean的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。

• 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。

* spring 支持几种 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：每次HTTP请产生一个新的Bean，该Bean仅在当前HTTP session内有效。

• global-session：全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。

* 将一个类声明为Spring的 bean 的注解有哪些?

• @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，主要用于数据库的操作。

• @Service：对应服务层，主要涉及一些复杂的逻辑

• @Controller：对应Spring MVC控制层，主要用于接收用户请求并调用Service层返回数据给前端页面。

* spring 自动装配 bean 有哪些方式？

• Spring容器负责创建应用程序中的bean同时通过ID来协调这些对象之间的关系。作为开发人员，我们需要告诉Spring要创建哪些bean并且如何将其装配到一起。

• spring中bean装配有两种方式

• 隐式的bean发现机制和自动装配

• 在java代码或者XML中进行显示配置

* 哪些是重要的bean生命周期方法？ 你能重载它们吗？

• 有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

* 什么是Spring的内部bean？什么是Spring inner beans？

• 在Spring框架中，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。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现，内部bean通常是匿名的，它们的Scope一般是prototype。

* 在 Spring中如何注入一个java集合？

• Spring提供了以下四种集合类的配置元素(配置标签)：该标签用来装配可重复的list值。该标签用来装配没有重复的set值。该标签可用来注入键和值可以为任何类型的键值对。该标签支持注入键和值都是字符串类型的键值对。

* 什么是bean装配？

• 装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

* 什么是bean的自动装配？

• 在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring 容器能够自动装配相互合作的bean，这意味着容器不需要和配置，能通过Bean工厂自动处理bean之间的协作。这意味着 Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

* 解释不同方式的自动装配，spring 自动装配 bean 有哪些方式？

• 在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。在Spring框架xml配置中共有5种自动装配：no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。byType：通过参数的数据类型进行自动装配。constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

* 使用@Autowired注解自动装配的过程是怎样的？

• 使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，&lt;context:annotation-config /&gt;。在启动spring IOC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IOC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；如果查询的结果不止一个，那么@Autowired会根据名称来查找；如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

* 自动装配有哪些局限性？

• 自动装配的局限性是：重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。基本数据类型：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

* 你可以在Spring中注入一个null 和一个空字符串吗？

• 可以。

### Spring注解

* 什么是基于Java的Spring注解配置? 给一些注解的例子

• 基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。@Configurationpublic class StudentConfig { @Bean public StudentBean myStudent() { return new StudentBean(); }}

* 怎样开启注解装配？

• 注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置 &lt;context:annotation-config/&gt;元素。

* @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。

* @Required 注解有什么作用

• 这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。示例：public class Employee { private String name; @Required public void setName(String name){ this.name=name; } public string getName(){ return name; }}

* @Autowired 注解有什么作用

• @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。public class Employee { private String name; @Autowired public void setName(String name) { this.name=name; } public string getName(){ return name; }}

* @Autowired和@Resource之间的区别

• @Autowired和@Resource可用于：构造函数、成员变量、Setter方法@Autowired和@Resource之间的区别在于@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。@Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

* @Qualifier 注解有什么作用

• 当您创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，您可以使用@Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义

* @RequestMapping 注解有什么用？

• @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。此注释可应用于两个级别：类级别：映射请求的 URL方法级别：映射 URL 以及 HTTP 请求方法

### Spring数据访问

* 解释对象/关系映射集成模块

• Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的ORM框架，如Hiberate，JDO和 MyBatis，JPA，TopLink，JDO，OJB等待 。Spring的事务管理同样支持以上所有ORM框架及JDBC。

* 在Spring框架中如何更有效地使用JDBC？

• 使用Spring JDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate

* 解释JDBC抽象和DAO模块

• 通过使用JDBC抽象和DAO模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

* spring DAO 有什么用？

• Spring DAO（数据访问对象） 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。

* spring JDBC API 中存在哪些类？

• JdbcTemplateSimpleJdbcTemplateNamedParameterJdbcTemplateSimpleJdbcInsertSimpleJdbcCall

* JdbcTemplate是什么

• JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

* 使用Spring通过什么方式访问Hibernate？使用 Spring 访问 Hibernate 的方法有哪些？

• 在Spring中有两种方式访问Hibernate：使用 Hibernate 模板和回调进行控制反转扩展 HibernateDAOSupport 并应用 AOP 拦截器节点

* 如何通过HibernateDaoSupport将Spring和Hibernate结合起来？

• 用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：配置the Hibernate SessionFactory继承HibernateDaoSupport实现一个DAO在AOP支持的事务中装配

* Spring支持的事务管理类型， spring 事务实现方式有哪些？

• Spring支持两种类型的事务管理：编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

* 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。

* 说一下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 PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

* 说一下 spring 的事务隔离？

• Spring的事务有哪几种隔离级别?

• TransactionDefinition接口中定义了五个隔离级别的常量：

• ISOLATION\_DEFAULT：使用后端数据库默认的隔离级别（一般用这个就好了），MySQL默认采用的是REPEATABLE\_READ隔离级别，Oracle默认采用的是READ\_COMMITTED隔离级别。

• ISOLATION\_READ\_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据，可能导致脏读、幻读或不可重复读。

• ISOLATION\_READ\_COMMITTED：允许读取并发事务以及提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

• ISOLATION\_REPEATABLE\_READ：对同一字段的多次读取结果都是一致的，除非数据是被事务自己修改的，可以阻止脏读和不可重复读，但幻读仍有可能发生。

• ISOLATION\_SERIALIZABLE：最高的隔离级别，所有事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读和幻读。但是这将严重影响程序性能，通常也不会用到。

* Spring框架的事务管理有哪些优点？

• 为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。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支持声明式事务管理。和Spring各种数据访问抽象层很好得集成。

* 你更倾向用那种事务管理类型？

• 大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。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。

* spring 事务实现方式有哪些？

• 实现方式共有两种

• 编码方式

• 声明式事务管理方式

• 基于AOP技术实现的bai声明式事务管理，实质du就是：在方法执行前后进行拦截，然后在目标方法开始之前创建并加入事务，执行完目标方法后根据执行情况提交或回滚事务。

• 声明式事务管理又有两种方式：基于XML配置文件的方式；另一个是在业务方法上进行@Transactional注解，将事务规则应用到业务逻辑中。

* 那你知道Spring事务有哪几种事务传播行为吗？

• 在TransactionDefinition中定义了7种事务传播行为： 支持当前事务的情况

• PROPAGATION\_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务

• PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

• PROPAGATION\_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常（mandatory：强制） 不支持当前事务的情况

• PROPAGATION\_REQUIRES\_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。

• PROPAGATION\_NOT\_SUPPORTED：以非事务的方式运行，如果当前存在事务，则把当前事务挂起。

• PROPAGATION\_NEVER：以非事务的方式运行，如果当前存在事务，则抛出异常。 其他情况

• PROPAGATION\_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于PROPAGATION\_REQUIRED。

* Spring事务不生效的几大原因

• 原因1：是否是数据库引擎设置不对造成的。比如我们常用的mysql，引擎MyISAM，是不支持事务操作的，需要改成InnoDB才能支持。

• 原因2：入口的方法必须是public，否则事务不起作用（这一点由Spring的AOP特性决定）private方法，final方法和static方法不能添加事务，加了也不生效。

• 原因3：Spring事务管理默认只对出现运行时异常（kava.lang.RuntimeException及其子类）进行回滚（至于Spring为什么这么设计：因为Spring认为Checked异常属于业务，程序员应该给出解决方案而不应该直接扔给框架）。

• 原因4：@EnableTransactionManagement // 启注解事务管理，等同于xml配置方式的 &lt;tx:annotation-driven /&gt;。没有使用该注解开启事务。

• 原因5：请确认你的类是否被代理了。（因为Spring的事务实现原理是AOP，只有通过代理对象调用方法才能被拦截，事务才能生效）。

• 原因6：请确保你的业务和事务入口在同一个线程里，否则事务也是不生效的，比如下面的代码：

• @Transactional@Overridepublic void save(User user1, User user2) { new Thread(() -&gt; { saveError(user1, user2); System.out.println(1 / 0); }).start();}

• 原因7：同一个类中一个无事务的方法调用另一个有事务的方法，事务是不会起作用的

### spring 常用的注入方式有哪些？

* 构造方法注入
* setter注入
* 基于注解的注入

### Spring 框架中用到了哪些设计模式？

* 工厂设计模式：Spring使用工厂模式通过BeanFactory、ApplicationContext创建Bean对象。
* 代理设计模式：Spring AOP功能的实现。
* 单例设计模式：Spring中的Bean默认都是单例的。
* 模板方法模式：Spring中jdbcTemplate、hibernateTemplate等以Template结尾的对数据库操作的类，就是用到了模板模式
* 包装器设计模式：我们的项目需要链接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户需求都太切换不同的数据源。
* 观察者模式：Spring事件驱动模型就是观察者模式很经典的一个应用。
* 适配器模式：Spring AOP的增强或通知使用到了适配器模式。SpringMVC中也是用到了适配器模式适配Controller

### Spring面向切面编程(AOP)

* 什么是AOP

• OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理等。

* Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式？

• AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

* JDK动态代理和CGLIB动态代理的区别

• Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

• InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

* 解释一下Spring AOP里面的几个名词

• （1）切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。 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 @AspectJ 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。 应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。（3）通知（Advice）：在AOP术语中，切面的工作被称为通知。（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。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）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

* Spring在运行时通知对象

• 通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候，Spring才会创建被代理的对象。因为Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入SpringAOP的切面。

* Spring只支持方法级别的连接点

• 因为Spring基于动态代理，所以Spring只支持方法连接点。Spring缺少对字段连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们可以利用Aspect来补充。

* 在Spring AOP 中，关注点和横切关注的区别是什么？在 spring aop 中 concern 和 cross-cutting concern 的不同之处

• 关注点（concern）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（cross-cutting concern）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。

* Spring通知有哪些类型？

• 在AOP术语中，切面的工作被称为通知，实际上是程序执行时要通过SpringAOP框架触发的代码段。Spring切面可以应用5种类型的通知：前置通知（Before）：在目标方法被调用之前调用通知功能；后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；返回通知（After-returning ）：在目标方法成功执行之后调用通知；异常通知（After-throwing）：在目标方法抛出异常后调用通知；环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

* 什么是切面 Aspect？

• aspect 由 pointcount 和 advice 组成，切面是通知和切点的结合。 它既包含了横切逻辑的定义, 也包括了连接点的定义. Spring AOP 就是负责实施切面的框架, 它将切面所定义的横切逻辑编织到切面所指定的连接点中. AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:如何通过 pointcut 和 advice 定位到特定的 joinpoint 上如何在 advice 中编写切面代码.可以简单地认为, 使用 @Aspect 注解的类就是切面.

* 解释基于XML Schema方式的切面实现

• 在这种情况下，切面由常规类以及基于XML的配置实现。

* 解释基于注解的切面实现

• 在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

## springmvc

### 概述

* 什么是Spring MVC？简单介绍下你对Spring MVC的理解？

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

* Spring MVC的优点

• （1）可以支持各种视图技术,而不仅仅局限于JSP；（2）与Spring框架集成（如IoC容器、AOP等）；（3）清晰的角色分配：前端控制器(dispatcherServlet) , 请求到处理器映射（handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器（ViewResolver）。（4） 支持各种请求资源的映射策略。

### 核心组件

* spring mvc 有哪些组件？

• DispatcherServlet：中央控制器，把请求给转发到具体的控制类

• Controller：具体处理请求的控制器

• HandlerMapping：映射处理器，负责映射中央处理器转发给controller时的映射策略

• ModelAndView：服务层返回的数据和视图层的封装类

• ViewResolver：视图解析器，解析具体的视图

• Interceptors ：拦截器，负责拦截我们定义的请求然后做处理工作

* Spring MVC的主要组件？

• （1）前端控制器 DispatcherServlet（不需要程序员开发）作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。（2）处理器映射器HandlerMapping（不需要程序员开发）作用：根据请求的URL来查找Handler（3）处理器适配器HandlerAdapter注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。（4）处理器Handler（需要程序员开发）（5）视图解析器 ViewResolver（不需要程序员开发）作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）（6）视图View（需要程序员开发jsp）View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

* 什么是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。

* 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。

* Spring MVC的控制器是不是单例模式,如果是,有什么问题,怎么解决？

• 答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

### 工作原理

* 请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程？

• （1）用户发送请求至前端控制器DispatcherServlet；（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；（6）Handler执行完成返回ModelAndView；（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；（9）ViewResolver解析后返回具体View；（10）DispatcherServlet对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。

* 说一下 spring mvc 运行流程？

• 客户端（浏览器）发送请求，直接请求到DispatcherServlet。

• DispatcherServlet根据请求细腻调用HandlerMapping，解析请求对应的Handler。

• 解析到对应的Handler（也就是Controller）后，开始由HandlerAdapter适配器处理。

• HandlerAdapter会根据Handler来调用真正的处理器来处理请求，并处理相应的业务逻辑。

• 处理器处理完业务后，会返回一个ModelAndView对象，Model是返回的数据对象，View是个逻辑上的View。

• ViewResolver会根据View查找实际的View。

• DispatcherServlet把返回的Model传给View（视图渲染）。

• 把View返回给请求者（浏览器）。

### MVC框架

* MVC是什么？MVC设计模式的好处有哪些

• mvc是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验的总结）。模型（model）-视图（view）-控制器（controller），三层架构的设计模式。用于实现前端页面的展现与后端业务数据处理的分离。mvc设计模式的好处分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。有利于系统的并行开发，提升开发效率。

### 常用注解

* 注解原理是什么

• 注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

* Spring MVC常用的注解有哪些？

• @RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。@Conntroller：控制器的注解，表示是表现层,不能用用别的注解代替

* SpingMvc中的控制器的注解一般用哪个,有没有别的注解可以替代？

• 答：一般用@Controller注解,也可以使用@RestController,@RestController注解相当于@ResponseBody ＋ @Controller,表示是表现层,除此之外，一般不用别的注解代替。

* @Controller注解的作用

• 在Spring MVC 中，控制器Controller 负责处理由DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model ，然后再把该Model 返回给对应的View 进行展示。在Spring MVC 中提供了一个非常简便的定义Controller 的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller 标记一个类是Controller ，然后使用@RequestMapping 和@RequestParam 等一些注解用以定义URL 请求和Controller 方法之间的映射，这样的Controller 就能被外界访问到。此外Controller 不会直接依赖于HttpServletRequest 和HttpServletResponse 等HttpServlet 对象，它们可以通过Controller 的方法参数灵活的获取到。@Controller 用于标记在一个类上，使用它标记的类就是一个Spring MVC Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器。单单使用@Controller 标记在一个类上还不能真正意义上的说它就是Spring MVC 的一个控制器类，因为这个时候Spring 还不认识它。那么要如何做Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给Spring 来管理。有两种方式：在Spring MVC 的配置文件中定义MyController 的bean 对象。在Spring MVC 的配置文件中告诉Spring 该到哪里去找标记为@Controller 的Controller 控制器。

* @RequestMapping注解的作用

• RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。RequestMapping注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。value， methodvalue： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；method： 指定请求的method类型， GET、POST、PUT、DELETE等；consumes，producesconsumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html;produces: 指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；params，headersparams： 指定request中必须包含某些参数值是，才让该方法处理。headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。

* @ResponseBody注解的作用

• 作用： 该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。使用时机：返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

* @PathVariable和@RequestParam的区别

• 请求路径上有个id的变量值，可以通过@PathVariable来获取 @RequestMapping(value = “/page/{id}”, method = RequestMethod.GET)@RequestParam用来获得静态的URL请求入参 spring注解时action里用到。

### 其他

* Spring MVC与Struts2区别

• 相同点都是基于mvc的表现层框架，都用于web项目的开发。不同点前端控制器不一样。Spring MVC的前端控制器是servlet：DispatcherServlet。struts2的前端控制器是filter：StrutsPreparedAndExcutorFilter。请求参数的接收方式不一样。Spring MVC是使用方法的形参接收请求的参数，基于方法的开发，线程安全，可以设计为单例或者多例的开发，推荐使用单例模式的开发（执行效率更高），默认就是单例开发模式。struts2是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，只能设计为多例的开发。Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，Spring MVC通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。与spring整合不一样。Spring MVC是spring框架的一部分，不需要整合。在企业项目中，Spring MVC使用更多一些。

* Spring MVC怎么样设定重定向和转发的？

• （1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"（2）重定向：在返回值前面加"redirect:"，譬如"redirect:www.baidu.com"

* Spring MVC怎么和AJAX相互调用的？

• 通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。具体步骤如下 ：（1）加入Jackson.jar（2）在配置文件中配置json的映射（3）在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

* 如何解决POST请求中文乱码问题，GET的又如何处理呢？

• （1）解决post请求乱码问题：在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；&lt;filter&gt; &lt;filter-name&gt;CharacterEncodingFilter&lt;/filter-name&gt; &lt;filter-class&gt;org.springframework.web.filter.CharacterEncodingFilter&lt;/filter-class&gt;&lt;init-param&gt; &lt;param-name&gt;encoding&lt;/param-name&gt; &lt;param-value&gt;utf-8&lt;/param-value&gt;&lt;/init-param&gt;&lt;/filter&gt;&lt;filter-mapping&gt; &lt;filter-name&gt;CharacterEncodingFilter&lt;/filter-name&gt; &lt;url-pattern&gt;/\*&lt;/url-pattern&gt;&lt;/filter-mapping&gt;复制代码（2）get请求中文参数出现乱码解决方法有两个：1、修改tomcat配置文件添加编码与工程编码一致，如下：&lt;ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/&gt;复制代码2、另外一种方法对参数进行重新编码：String userName = new String(request.getParamter(“userName”).getBytes(“ISO8859-1”),“utf-8”)ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

* Spring MVC的异常处理？

• 答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

* 如果在拦截请求中，我想拦截get方式提交的方法,怎么配置

• 答：可以在@RequestMapping注解里面加上method=RequestMethod.GET。

* 怎样在方法里面得到Request,或者Session？

• 答：直接在方法的形参中声明request,Spring MVC就自动把request对象传入。

* 如果想在拦截的方法里面得到从前台传入的参数,怎么得到？

• 答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

* 如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象？

• 答：直接在方法中声明这个对象,Spring MVC就自动会把属性赋值到这个对象里面。

* Spring MVC中函数的返回值是什么？

• 答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好

* Spring MVC用什么对象从后台向前台传递数据的？

• 答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

* 怎么样把ModelMap里面的数据放入Session里面？

• 答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

* Spring MVC里面拦截器是怎么写的

• 有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在Spring MVC的配置文件中配置拦截器即可：&lt;!-- 配置Spring MVC的拦截器 --&gt;&lt;mvc:interceptors&gt; &lt;!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 --&gt; &lt;bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"&gt;&lt;/bean&gt; &lt;!-- 只针对部分请求拦截 --&gt; &lt;mvc:interceptor&gt; &lt;mvc:mapping path="/modelMap.do" /&gt; &lt;bean class="com.zwp.action.MyHandlerInterceptorAdapter" /&gt; &lt;/mvc:interceptor&gt;

* 介绍一下 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。

## springboot

### 什么是 spring boot？

* Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，使开发者能快速上手

### 为什么要用 spring boot？

* 快速开发，快速整合，配置简化、内嵌服务容器

### SpringBoot与SpringCloud 区别

* SpringBoot是快速开发的Spring框架，SpringCloud是完整的微服务框架，SpringCloud依赖于SpringBoot。

### Spring Boot 有哪些优点？

* 容易上手，提升开发效率，为 Spring 开发提供一个更快、更简单的开发框架。
* 开箱即用，远离繁琐的配置。
* 提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
* SpringBoot总结就是使编码变简单、配置变简单、部署变简单、监控变简单等等

### spring boot 核心配置文件是什么？

* spring boot 核心的两个配置文件

• bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud 配置就会使用这个文件。且 boostrap 里面的属性不能被覆盖；

• application (. yml 或者 . properties)： 由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

### spring boot 配置文件有哪几种类型？它们有什么区别？

### 自己的使用

* 注解

• @Autowired

• 注入Service

• @RequestMapping

• @RequestMapping注解分别注释在类和方法上，所以在前端写链接的时候要写完全的路径（类上标签的路径+方法标签上的路劲）

• @RequestMapping("/st/mobile")

### 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，例如：java 如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。@ComponentScan：Spring组件扫描。

### 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, Lockback 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，但是不管是那种日志框架他都支持将配置文件输出到控制台或者文件中。

### SpringBoot Starter的工作原理

* 我个人理解SpringBoot就是由各种Starter组合起来的，我们自己也可以开发Starter在sprinBoot启动时由@SpringBootApplication注解会自动去maven中读取每个starter中的spring.factories文件,该文件里配置了所有需要被创建spring容器中的bean，并且进行自动配置把bean注入SpringContext中 //（SpringContext是Spring的配置文件）

### Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？

* 配置变更JDK 版本升级第三方类库升级响应式 Spring 编程支持HTTP/2 支持配置属性绑定更多改进与加强

### SpringBoot支持什么前端模板，

* thymeleaf，freemarker，jsp，官方不推荐JSP会有限制

### SpringBoot的缺点

* 我觉得是为难人，SpringBoot在目前我觉得没有什么缺点，非要找一个出来我觉得就是由于不用自己做的配置，报错时很难定位。

### 运行 Spring Boot 有哪几种方式？

* 打包用命令或者放到容器中运行用 Maven/ Gradle 插件运行直接执行 main 方法运行

### Spring Boot 需要独立的容器运行吗？

* 可以不需要，内置了 Tomcat/ Jetty 等容器。

### 开启 Spring Boot 特性有哪几种方式？

* 继承spring-boot-starter-parent项目导入spring-boot-dependencies项目依赖

### SpringBoot 实现热部署有哪几种方式？

* 热部署就是可以不用重新运行SpringBoot项目可以实现操作后台代码自动更新到以运行的项目中主要有两种方式：Spring LoadedSpring-boot-devtools

### SpringBoot事物的使用

* SpringBoot的事物很简单，首先使用注解EnableTransactionManagement开启事物之后，然后在Service方法上添加注解Transactional便可。

### Async异步调用方法

* 在SpringBoot中使用异步调用是很简单的，只需要在方法上使用@Async注解即可实现方法的异步调用。 注意：需要在启动类加入@EnableAsync使异步调用@Async注解生效。

### 如何在 Spring Boot 启动的时候运行一些特定的代码？

* 可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法

### Spring Boot 有哪几种读取配置的方式？

* Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationPropertie注解来绑定变量

### 什么是 JavaConfig？

* Spring JavaConfig 是 Spring 社区的产品，Spring 3.0引入了他，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。常用的Java config：@Configuration：在类上打上写下此注解，表示这个类是配置类@ComponentScan：在配置类上添加 @ComponentScan 注解。该注解默认会扫描该类所在的包下所有的配置类，相当于之前的 &lt;context:component-scan &gt;。@Bean：bean的注入：相当于以前的&lt; bean id="objectMapper" class="org.codehaus.jackson.map.ObjectMapper" /&gt;@EnableWebMvc：相当于xml的&lt;mvc:annotation-driven &gt;@ImportResource： 相当于xml的 &lt; import resource="applicationContext-cache.xml"&gt;

### SpringBoot的自动配置原理是什么

* 主要是Spring Boot的启动类上的核心注解SpringBootApplication注解主配置类，有了这个主配置类启动时就会为SpringBoot开启一个@EnableAutoConfiguration注解自动配置功能。有了这个EnableAutoConfiguration的话就会：从配置文件META\_INF/Spring.factories加载可能用到的自动配置类去重，并将exclude和excludeName属性携带的类排除过滤，将满足条件（@Conditional）的自动配置类返回

### 你如何理解 Spring Boot 配置加载顺序？

* 在 Spring Boot 里面，可以使用以下几种方式来加载配置。1.properties文件；2.YAML文件；3.系统环境变量；4.命令行参数；

### 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。

### YAML 配置的优势在哪里 ?

* YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢？配置有序，在一些特殊的场景下，配置有序很关键简洁明了，他还支持数组，数组中的元素可以是基本数据类型也可以是对象相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

### Spring Boot 是否可以使用 XML 配置 ?

* Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

### spring boot 核心配置文件是什么？bootstrap.properties 和 application.properties 有何区别 ?

* 单纯做 Spring Boot 开发，可能不太容易遇到 bootstrap.properties 配置文件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时侯。spring boot 核心的两个配置文件bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud 配置就会使用这个文件。且 boostrap 里面的属性不能被覆盖；application (. yml 或者 . properties)： 由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

### 什么是 Spring Profiles？

* 在项目的开发中，有些配置文件在开发、测试或者生产等不同环境中可能是不同的，例如数据库连接、redis的配置等等。那我们如何在不同环境中自动实现配置的切换呢？Spring给我们提供了profiles机制给我们提供的就是来回切换配置文件的功能Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

### SpringBoot多数据源拆分的思路

* 先在properties配置文件中配置两个数据源，创建分包mapper，使用@ConfigurationProperties读取properties中的配置，使用@MapperScan注册到对应的mapper包中

### SpringBoot多数据源事务如何管理

* 第一种方式是在service层的@TransactionManager中使用transactionManager指定DataSourceConfig中配置的事务第二种是使用jta-atomikos实现分布式事务管理

### 保护 Spring Boot 应用有哪些方法？

* 在生产中使用HTTPS使用Snyk检查你的依赖关系升级到最新版本启用CSRF保护使用内容安全策略防止XSS攻击

### 如何实现 Spring Boot 应用程序的安全性？

* 为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展WebSecurityConfigurerAdapter 并覆盖其方法。

### 比较一下 Spring Security 和 Shiro 各自的优缺点 ?

* 由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter ，包括 Spring Security 的 Starter ，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是 Spring Boot 项目，一般选择 Spring Security 。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 Spring Security 相比，主要有如下一些特点：Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单Spring Security 功能强大；Shiro 功能简单

### 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); } }

### 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。

### 如何使用 Spring Boot 实现全局异常处理？

* Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

### 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。

### SpringBoot性能如何优化

* 如果项目比较大，类比较多，不使用@SpringBootApplication，采用@Compoment指定扫包范围在项目启动时设置JVM初始内存和最大内存相同将springboot内置服务器由tomcat设置为undertow

### 如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？Spring Boot项目如何热部署？

* 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。&lt;dependency&gt; &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt; &lt;artifactId&gt;spring-boot-devtools&lt;/artifactId&gt;&lt;/dependency&gt;

### SpringBoot微服务中如何实现 session 共享 ?

* 在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

### 您使用了哪些 starter maven 依赖项？

* 使用了下面的一些依赖项spring-boot-starter-web 嵌入tomcat和web开发需要servlet与jsp支持spring-boot-starter-data-jpa 数据库支持spring-boot-starter-data-redis redis数据库支持spring-boot-starter-data-solr solr支持mybatis-spring-boot-starter 第三方的mybatis集成starter自定义的starter(如果自己开发过就可以说出来)

### Spring Boot 中的 starter 到底是什么 ?

* 首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration ，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性(spring.factories)注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

### Spring Boot 中如何实现定时任务 ?

* 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。

### 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。总结就是打包用的

### SpringBoot如何实现打包

* 进入项目目录在控制台输入mvn clean package，clean是清空已存在的项目包，package进行打包或者点击左边选项栏中的Mavne，先点击clean在点击package

### 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。

## spring cloud

### spring cloud 的核心组件有哪些？

### spring cloud 断路器的作用是什么？

### spring cloud gateway了解吗？

### 什么是微服务架构

* 什么是微服务架构

• 微服务架构就是将单体的应用程序分成多个应用程序，这多个应用程序就成为微服务，每个微服务运行在自己的进程中，并使用轻量级的机制通信。这些服务围绕业务能力来划分，并通过自动化部署机制来独立部署。这些服务可以使用不同的编程语言，不同数据库，以保证最低限度的集中式管理。

* 为什么需要学习Spring Cloud

• 首先springcloud基于spingboot的优雅简洁，可还记得我们被无数xml支配的恐惧？可还记得springmvc，mybatis错综复杂的配置，有了spingboot，这些东西都不需要了，spingboot好处不再赘诉，springcloud就基于SpringBoot把市场上优秀的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理什么叫做开箱即用？即使是当年的黄金搭档dubbo+zookeeper下载配置起来也是颇费心神的！而springcloud完成这些只需要一个jar的依赖就可以了！springcloud大多数子模块都是直击痛点，像zuul解决的跨域，fegin解决的负载均衡，hystrix的熔断机制等等等等

* 什么是 spring cloud？
* SpringCloud的优缺点

• 优点： 1.耦合度比较低。不会影响其他模块的开发。 2.减轻团队的成本，可以并行开发，不用关注其他人怎么开发，先关注自己的开发。 3.配置比较简单，基本用注解就能实现，不用使用过多的配置文件。 4.微服务跨平台的，可以用任何一种语言开发。 5.每个微服务可以有自己的独立的数据库也有用公共的数据库。 6.直接写后端的代码，不用关注前端怎么开发，直接写自己的后端代码即可，然后暴露接口，通过组件进行服务通信。缺点： 1.部署比较麻烦，给运维工程师带来一定的麻烦。 2.针对数据的管理比麻烦，因为微服务可以每个微服务使用一个数据库。 3.系统集成测试比较麻烦 4.性能的监控比较麻烦。【最好开发一个大屏监控系统】总的来说优点大过于缺点，目前看来Spring Cloud是一套非常完善的分布式框架，目前很多企业开始用微服务、Spring Cloud的优势是显而易见的。因此对于想研究微服务架构的同学来说，学习Spring Cloud是一个不错的选择。

* SpringBoot和SpringCloud的区别？

• SpringBoot专注于快速方便的开发单个个体微服务。SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务SpringBoot可以离开SpringCloud独立使用开发项目， 但是SpringCloud离不开SpringBoot ，属于依赖的关系SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

* Spring Cloud和SpringBoot版本对应关系

• Spring Cloud Version SpringBoot VersionHoxton 2.2.xGreenwich 2.1.xFinchley 2.0.xEdgware 1.5.xDalston 1.5.x

* SpringCloud由什么组成

• 这就有很多了，我讲几个开发中最重要的Spring Cloud Eureka：服务注册与发现Spring Cloud Zuul：服务网关Spring Cloud Ribbon：客户端负载均衡Spring Cloud Feign：声明性的Web服务客户端Spring Cloud Hystrix：断路器Spring Cloud Config：分布式统一配置管理等20几个框架，开源一直在更新

* 使用 Spring Boot 开发分布式微服务时，我们面临什么问题

• （1）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。（2）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。（3）冗余-分布式系统中的冗余问题。（4）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。（5）性能-问题 由于各种运营开销导致的性能问题。

* Spring Cloud 和dubbo区别?

• （1）服务调用方式：dubbo是RPC springcloud Rest Api（2）注册中心：dubbo 是zookeeper springcloud是eureka，也可以是zookeeper（3）服务网关，dubbo本身没有实现，只能通过其他第三方技术整合，springcloud有Zuul路由网关，作为路由服务器，进行消费者的请求分发,springcloud支持断路器，与git完美集成配置文件支持版本控制，事物总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素。

### Eureka

* 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。

* 什么是Eureka

• Eureka作为SpringCloud的服务注册功能服务器，他是服务注册中心，系统中的其他服务使用Eureka的客户端将其连接到Eureka Service中，并且保持心跳，这样工作人员可以通过Eureka Service来监控各个微服务是否运行正常。

* Eureka怎么实现高可用

• 集群吧，注册多台Eureka，然后把SpringCloud服务互相注册，客户端从Eureka获取信息时，按照Eureka的顺序来访问。

* 什么是Eureka的自我保护模式，

• 默认情况下，如果Eureka Service在一定时间内没有接收到某个微服务的心跳，Eureka Service会进入自我保护模式，在该模式下Eureka Service会保护服务注册表中的信息，不在删除注册表中的数据，当网络故障恢复后，Eureka Servic 节点会自动退出自我保护模式

* DiscoveryClient的作用

• 可以从注册中心中根据服务别名获取注册的服务器信息。

* 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中的节点服务挂了就要选举在选举期间注册服务瘫痪,虽然服务最终会恢复,但是选举期间不可用的，选举就是改微服务做了集群，必须有一台主其他的都是从Eureka各个节点是平等关系,服务器挂了没关系，只要有一台Eureka就可以保证服务可用，数据都是最新的。如果查询到的数据并不是最新的，就是因为Eureka的自我保护模式导致的Eureka本质上是一个工程,而ZooKeeper只是一个进程Eureka可以很好的应对因网络故障导致部分节点失去联系的情况,而不会像ZooKeeper 一样使得整个注册系统瘫痪ZooKeeper保证的是CP，Eureka保证的是APCAP：C：一致性&gt;Consistency;取舍：(强一致性、单调一致性、会话一致性、最终一致性、弱一致性)A：可用性&gt;Availability;P：分区容错性&gt;Partition tolerance;

### Zuul

* 什么是网关?

• 网关相当于一个网络服务架构的入口，所有网络请求必须通过网关转发到具体的服务。

* 网关的作用是什么

• 统一管理微服务请求，权限控制、负载均衡、路由转发、监控、安全控制黑名单和白名单等

* 什么是Spring Cloud Zuul（服务网关）

• Zuul是对SpringCloud提供的成熟对的路由方案，他会根据请求的路径不同，网关会定位到指定的微服务，并代理请求到不同的微服务接口，他对外隐蔽了微服务的真正接口地址。三个重要概念：动态路由表，路由定位，反向代理：动态路由表：Zuul支持Eureka路由，手动配置路由，这俩种都支持自动更新路由定位：根据请求路径，Zuul有自己的一套定位服务规则以及路由表达式匹配反向代理：客户端请求到路由网关，网关受理之后，在对目标发送请求，拿到响应之后在 给客户端它可以和Eureka,Ribbon,Hystrix等组件配合使用，Zuul的应用场景：对外暴露，权限校验，服务聚合，日志审计等

* 网关与过滤器有什么区别

• 网关是对所有服务的请求进行分析过滤，过滤器是对单个服务而言。

* 常用网关框架有那些？

• Nginx、Zuul、Gateway

* Zuul与Nginx有什么区别？

• Zuul是java语言实现的，主要为java服务提供网关服务，尤其在微服务架构中可以更加灵活的对网关进行操作。Nginx是使用C语言实现，性能高于Zuul，但是实现自定义操作需要熟悉lua语言，对程序员要求较高，可以使用Nginx做Zuul集群。

* 既然Nginx可以实现网关？为什么还需要使用Zuul框架

• Zuul是SpringCloud集成的网关，使用Java语言编写，可以对SpringCloud架构提供更灵活的服务。如何设计一

* 如何设计一套API接口

• 考虑到API接口的分类可以将API接口分为开发API接口和内网API接口，内网API接口用于局域网，为内部服务器提供服务。开放API接口用于对外部合作单位提供接口调用，需要遵循Oauth2.0权限认证协议。同时还需要考虑安全性、幂等性等问题。

* ZuulFilter常用有那些方法

• Run()：过滤器的具体业务逻辑shouldFilter()：判断过滤器是否有效filterOrder()：过滤器执行顺序filterType()：过滤器拦截位置

* 如何实现动态Zuul网关路由转发

• 通过path配置拦截请求，通过ServiceId到配置中心获取转发的服务列表，Zuul内部使用Ribbon实现本地负载均衡和转发。

* Zuul网关如何搭建集群

• 使用Nginx的upstream设置Zuul服务集群，通过location拦截请求并转发到upstream，默认使用轮询机制对Zuul集群发送请求。

### Ribbon

* 负载平衡的意义什么？

• 简单来说： 先将集群，集群就是把一个的事情交给多个人去做，假如要做1000个产品给一个人做要10天，我叫10个人做就是一天，这就是集群，负载均衡的话就是用来控制集群，他把做的最多的人让他慢慢做休息会，把做的最少的人让他加量让他做多点。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。

* Ribbon是什么？

• Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法Ribbon客户端组件提供一系列完善的配置项，如连接超时，重试等。简单的说，就是在配置文件中列出后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随即连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。（有点类似Nginx）

* Nginx与Ribbon的区别

• Nginx是反向代理同时可以实现负载均衡，nginx拦截客户端请求采用负载均衡策略根据upstream配置进行转发，相当于请求通过nginx服务器进行转发。Ribbon是客户端负载均衡，从注册中心读取目标服务器信息，然后客户端采用轮询策略对服务直接访问，全程在客户端操作。

* Ribbon底层实现原理

• Ribbon使用discoveryClient从注册中心读取目标服务信息，对同一接口请求进行计数，使用%取余算法获取目标服务集群索引，返回获取到的目标服务信息。

* @LoadBalanced注解的作用

• ​ 开启客户端负载均衡。

### Hystrix

* 什么是断路器

• 当一个服务调用另一个服务由于网络原因或自身原因出现问题，调用者就会等待被调用者的响应 当更多的服务请求到这些资源导致更多的请求等待，发生连锁效应（雪崩效应）断路器有三种状态打开状态：一段时间内 达到一定的次数无法调用 并且多次监测没有恢复的迹象 断路器完全打开 那么下次请求就不会请求到该服务半开状态：短时间内 有恢复迹象 断路器会将部分请求发给该服务，正常调用时 断路器关闭关闭状态：当服务一直处于正常状态 能正常调用

* 什么是 Hystrix？

• 在分布式系统，我们一定会依赖各种服务，那么这些个服务一定会出现失败的情况，就会导致雪崩，Hystrix就是这样的一个工具，防雪崩利器，它具有服务降级，服务熔断，服务隔离，监控等一些防止雪崩的技术。Hystrix有四种防雪崩方式:服务降级：接口调用失败就调用本地的方法返回一个空服务熔断：接口调用失败就会进入调用接口提前定义好的一个熔断的方法，返回错误信息服务隔离：隔离服务之间相互影响服务监控：在服务发生调用时,会将每秒请求数、成功请求数等运行指标记录下来。

* 谈谈服务雪崩效应

• 雪崩效应是在大型互联网项目中，当某个服务发生宕机时，调用这个服务的其他服务也会发生宕机，大型项目的微服务之间的调用是互通的，这样就会将服务的不可用逐步扩大到各个其他服务中，从而使整个项目的服务宕机崩溃.发生雪崩效应的原因有以下几点单个服务的代码存在bug. 2请求访问量激增导致服务发生崩溃(如大型商城的枪红包，秒杀功能). 3.服务器的硬件故障也会导致部分服务不可用.

* 在微服务中，如何保护服务?

• 一般使用使用Hystrix框架，实现服务隔离来避免出现服务的雪崩效应，从而达到保护服务的效果。当微服务中，高并发的数据库访问量导致服务线程阻塞，使单个服务宕机，服务的不可用会蔓延到其他服务，引起整体服务灾难性后果，使用服务降级能有效为不同的服务分配资源,一旦服务不可用则返回友好提示，不占用其他服务资源，从而避免单个服务崩溃引发整体服务的不可用.

* 服务雪崩效应产生的原因

• 因为Tomcat默认情况下只有一个线程池来维护客户端发送的所有的请求，这时候某一接口在某一时刻被大量访问就会占据tomcat线程池中的所有线程，其他请求处于等待状态，无法连接到服务接口。

* 谈谈服务降级、熔断、服务隔离

• 服务降级：当客户端请求服务器端的时候，防止客户端一直等待，不会处理业务逻辑代码，直接返回一个友好的提示给客户端。服务熔断是在服务降级的基础上更直接的一种保护方式，当在一个统计时间范围内的请求失败数量达到设定值（requestVolumeThreshold）或当前的请求错误率达到设定的错误率阈值（errorThresholdPercentage）时开启断路，之后的请求直接走fallback方法，在设定时间（sleepWindowInMilliseconds）后尝试恢复。服务隔离就是Hystrix为隔离的服务开启一个独立的线程池，这样在高并发的情况下不会影响其他服务。服务隔离有线程池和信号量两种实现方式，一般使用线程池方式。

* 服务降级底层是如何实现的？

• Hystrix实现服务降级的功能是通过重写HystrixCommand中的getFallback()方法，当Hystrix的run方法或construct执行发生错误时转而执行getFallback()方法。

### Feign

* 什么是Feign？

• Feign 是一个声明web服务客户端，这使得编写web服务客户端更容易他将我们需要调用的服务方法定义成抽象方法保存在本地就可以了，不需要自己构建Http请求了，直接调用接口就行了，不过要注意，调用方法要和本地抽象方法的签名完全一致。

* SpringCloud有几种调用接口方式

• FeignRestTemplate

* Ribbon和Feign调用服务的区别

• 调用方式同：Ribbon需要我们自己构建Http请求，模拟Http请求然后通过RestTemplate发给其他服务，步骤相当繁琐而Feign则是在Ribbon的基础上进行了一次改进，采用接口的形式，将我们需要调用的服务方法定义成抽象方法保存在本地就可以了，不需要自己构建Http请求了，直接调用接口就行了，不过要注意，调用方法要和本地抽象方法的签名完全一致。

### Bus

* 什么是 Spring Cloud Bus？

• Spring Cloud Bus就像一个分布式执行器，用于扩展的Spring Boot应用程序的配置文件，但也可以用作应用程序之间的通信通道。Spring Cloud Bus 不能单独完成通信，需要配合MQ支持Spring Cloud Bus一般是配合Spring Cloud Config做配置中心的Springcloud config实时刷新也必须采用SpringCloud Bus消息总线

### Config

* 什么是Spring Cloud Config?

• Spring Cloud Config为分布式系统中的外部配置提供服务器和客户端支持，可以方便的对微服务各个环境下的配置进行集中式管理。Spring Cloud Config分为Config Server和Config Client两部分。Config Server负责读取配置文件，并且暴露Http API接口，Config Client通过调用Config Server的接口来读取配置文件。

* 分布式配置中心有那些框架？

• Apollo、zookeeper、springcloud config。

* 分布式配置中心的作用？

• 动态变更项目配置信息而不必重新部署项目。

* SpringCloud Config 可以实现实时刷新吗？

• springcloud config实时刷新采用SpringCloud Bus消息总线。

### Gateway

* 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。

### SpringCloud主要项目

* SpringCloud主要项目

• Spring Cloud的子项目，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

• Config能够管理所有微服务的配置文件集中配置管理工具，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(重点，这些组件用的最多)

• 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。我觉得SpringCloud的福音是Netflix，他把人家的组件都搬来进行封装了，使开发者能快速简单安全的使用

* Spring Cloud Bus

• 用于传播集群状态变化的消息总线，使用轻量级消息代理链接分布式系统中的节点，可以用来动态刷新集群中的服务配置信息。简单来说就是修改了配置文件，发送一次请求，所有客户端便会重新读取配置文件。需要利用中间插件MQ

* Spring Cloud Consul

• Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现与配置。与其它分布式服务注册与发现的方案，Consul 的方案更“一站式”，内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value 存储、多数据中心方案，不再需要依赖其它工具（比如 ZooKeeper 等）。使用起来也较为简单。Consul 使用 Go 语言编写，因此具有天然可移植性(支持Linux、windows和Mac OS X)；安装包仅包含一个可执行文件，方便部署，与 Docker 等轻量级容器可无缝配合。

* Spring Cloud Security

• 安全工具包，他可以对对Zuul代理中的负载均衡从前端到后端服务中获取SSO令牌资源服务器之间的中继令牌使Feign客户端表现得像OAuth2RestTemplate（获取令牌等）的拦截器在Zuul代理中配置下游身份验证Spring Cloud Security提供了一组原语，用于构建安全的应用程序和服务，而且操作简便。可以在外部（或集中）进行大量配置的声明性模型有助于实现大型协作的远程组件系统，通常具有中央身份管理服务。它也非常易于在Cloud Foundry等服务平台中使用。在Spring Boot和Spring Security OAuth2的基础上，可以快速创建实现常见模式的系统，如单点登录，令牌中继和令牌交换。

* Spring Cloud Sleuth

• 在微服务中，通常根据业务模块分服务，项目中前端发起一个请求，后端可能跨几个服务调用才能完成这个请求（如下图）。如果系统越来越庞大，服务之间的调用与被调用关系就会变得很复杂，假如一个请求中需要跨几个服务调用，其中一个服务由于网络延迟等原因挂掉了，那么这时候我们需要分析具体哪一个服务出问题了就会显得很困难。Spring Cloud Sleuth服务链路跟踪功能就可以帮助我们快速的发现错误根源以及监控分析每条请求链路上的性能等等

* Spring Cloud Stream

• 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

• Spring Cloud Task的目标是为Spring Boot应用程序提供创建短运行期微服务的功能。在Spring Cloud Task中，我们可以灵活地动态运行任何任务，按需分配资源并在任务完成后检索结果。Tasks是Spring Cloud Data Flow中的一个基础项目，允许用户将几乎任何Spring Boot应用程序作为一个短期任务执行。

* Spring Cloud Zookeeper

• SpringCloud支持三种注册方式Eureka， Consul(go语言编写)，zookeeperSpring Cloud Zookeeper是基于Apache Zookeeper的服务治理组件。

* Spring Cloud Gateway

• Spring cloud gateway是spring官方基于Spring 5.0、Spring Boot2.0和Project Reactor等技术开发的网关，Spring Cloud Gateway旨在为微服务架构提供简单、有效和统一的API路由管理方式，Spring Cloud Gateway作为Spring Cloud生态系统中的网关，目标是替代Netflix Zuul，其不仅提供统一的路由方式，并且还基于Filer链的方式提供了网关基本的功能，例如：安全、监控/埋点、限流等。

* Spring Cloud OpenFeign

• Feign是一个声明性的Web服务客户端。它使编写Web服务客户端变得更容易。要使用Feign，我们可以将调用的服务方法定义成抽象方法保存在本地添加一点点注解就可以了，不需要自己构建Http请求了，直接调用接口就行了，不过要注意，调用方法要和本地抽象方法的签名完全一致。

* 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。目前Spring Cloud的最新版本是Hoxton。

# 数据库

## MySQL

### 数据库的三范式了解吗？

* 第一范式：每个列都不可以再拆分。
* 第二范式：在第一范式的基础上，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：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他非主键。

### 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时最大的 id 是多少？

### 什么是存储过程，有哪些优缺点？

* 什么是存储过程？

• 存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。

* 优点

• 存储过程是预编译过的，执行效率高。

• 存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯

• 安全性高，执行存储过程需要有一定权限的用户。

• 存储过程可以重复使用，减少数据库开发人员的工作量。

* 缺点

• 调试麻烦，但是用 PL/SQL Developer 调试很方便！弥补这个缺点。

• 移植问题，数据库端代码当然是与数据库相关的。但是如果是做工程型项目，基本不存在移植问题。

• 重新编译问题，因为后端代码是运行前编译的，如果带有引用关系的对象发生改变时，受影响的存储过程、包将需要重新编译（不过也可以设置成运行时刻自动编译）。

• 如果在一个程序系统中大量的使用存储过程，到程序交付使用的时候随着用户需求的增加会导致数据结构的变化，接着就是系统的相关问题了，最后如果用户想维护该系统可以说是很难很难、而且代价是空前的，维护起来更麻烦。

### 创建一个简单的存储过程

* 创建存储过程的简单语法

• create procedure 名称()begin.........end

* 创建一个简单的存储过程

• create procedure testa()begin select \* from users;    select \* from orders;end;

* 调用存储过程

• call testa();

### mysql 索引类型有几种

* 普通索引

• 即不应用任何限制条件的索引，该索引可以在任何数据类型中创建。字段本身的约束条件可以判断其值是否为空或唯一。

* 唯一索引

• 使用UNIQUE参数可以设置唯一索引。创建该索引时，索引的值必须唯一。主键是一种特殊唯一索引

* 全文索引

• 使用FULLTEXT参数可以设置全文索引。全文索引只能创建在CHAR、VARCHAR、TEXT类型的字段上。查询数据量较大的字符串类型字段时，使用全文索引可以提高查询速度。注意：全文索引在默认情况下是对大小写字母不敏感的，可以通过使用二进制对索引的列进行排序以执行大小写敏感的全文索引。MySQL中只有MyISAM存储引擎支持全文索引

* 单列索引

• 顾名思义，单列索引值对应一个字段的索引。可以包括上述的三种索引方式。应用该索引的条件只需要保证该索引值对应一个字段即可。

* 多列索引

• 多列索引是在表的多个字段上创建一个索引。该索引只想创建时对应的多个字段，可以通过这几个字段进行查询。要想应用该索引，用户必须使用这些字段中的第一个字段。

* 空间索引

• 使用SPATIAL参数可以设置控件索引。控件索引只能建立在控件数据类型（LINESTRING、POINT、GEOMETRY等）上，这样可以提高系统获取控件数据的效率。MySQL中只有MyISAM存储引擎支持空间索引，且该字段不能为空值。

### mysql 索引是怎么实现的？

### 使用索引一定能提高查询性能吗？

### 怎么验证 mysql 的索引是否满足需求？

### 哪些场景下，索引会失效？

### 说一下 mysql 常用的引擎？

### 说一下 mysql 的行锁和表锁？

### 说一下数据库的乐观锁和悲观锁？

### mysql 问题排查都有哪些手段？

### 如何做 mysql 的性能优化？

### 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

### 索引

* 什么是索引？

• 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。

• 索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

• 更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

* 索引有哪些优缺点？

• 索引的优点

• 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

• 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

• 索引的缺点

• 时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；

• 空间方面：索引需要占物理空间。

* 怎么创建索引的，有什么好处，有哪些分类

• 创建索引的语法

• create index depe\_unique\_ide on depe(dept\_no) tablespace idx\_

• 有什么好处

• 创建索引可以增加查询速度，唯一索引可以保证数据库列的一致性，可以确定表与表之间的连接

• 索引的分类：

• 逻辑分类：

• 单列索引，复合索引，唯一索引，非唯一索引，函数索引

• 物理分类：

• B数索引，反向键索引，位图索引

* 简述有哪些索引和作用

• 唯一索引：不允许有俩行具有相同的值

• 主键索引：为了保持数据库表与表之间的关系

• 聚集索引：表中行的物理顺序与键值的逻辑（索引）顺序相同。

• 非聚集索引：聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致

• 复合索引：在创建索引时，并不是只能对一列进行创建索引，可以与主键一样，讲多个组合为索引

• 全文索引： 全文索引为在字符串数据中进行复杂的词搜索提供有效支持

• 索引的作用：通过索引可以大大的提高数据库的检索速度，改善数据库性能

* 索引使用场景

• 当数据多且字段值有相同的值得时候用普通索引。

• 当字段多且字段值没有重复的时候用唯一索引。

• 当有多个字段名都经常被查询的话用复合索引。

• 普通索引不支持空值，唯一索引支持空值。

• 但是，若是这张表增删改多而查询较少的话，就不要创建索引了，因为如果你给一列创建了索引，那么对该列进行增删改的时候，都会先访问这一列的索引，

• 若是增，则在这一列的索引内以新填入的这个字段名的值为名创建索引的子集，

• 若是改，则会把原来的删掉，再添入一个以这个字段名的新值为名创建索引的子集，

• 若是删，则会把索引中以这个字段为名的索引的子集删掉。

• 所以，会对增删改的执行减缓速度，

• 所以，若是这张表增删改多而查询较少的话，就不要创建索引了。

• 更新太频繁地字段不适合创建索引。

• 不会出现在where条件中的字段不该建立索引。

* 主键索引与唯一索引的区别

• 主键是一种约束，唯一索引是一种索引，两者在本质上是不同的。

• 主键创建后一定包含一个唯一性索引，唯一性索引并不一定就是主键。

• 唯一性索引列允许空值，而主键列不允许为空值

• 主键列在创建时，已经默认为空值 ++ 唯一索引了。

• 一个表最多只能创建一个主键，但可以创建多个唯一索引。

• 主键更适合那些不容易更改的唯一标识，如自动递增列、身份证号等。

• 主键可以被其他表引用为外键，而唯一索引不能

* 索引有哪几种类型？

• 主键索引: 数据列不允许重复，不允许为NULL，一个表只能有一个主键。

• 唯一索引: 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

• 可以通过 ALTER TABLE table\_name ADD UNIQUE (column); 创建唯一索引

• 可以通过 ALTER TABLE table\_name ADD UNIQUE (column1,column2); 创建唯一组合索引

• 普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值。

• 可以通过ALTER TABLE table\_name ADD INDEX index\_name (column);创建普通索引

• 可以通过ALTER TABLE table\_name ADD INDEX index\_name(column1, column2, column3);创建组合索引

• 全文索引： 是目前搜索引擎使用的一种关键技术。

• 可以通过ALTER TABLE table\_name ADD FULLTEXT (column);创建全文索引

* 索引的数据结构（b树，hash）

• 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。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

• B树索引

• mysql通过存储引擎取数据，基本上90%的人用的就是InnoDB了，按照实现方式分，InnoDB的索引类型目前只有两种：BTREE（B树）索引和HASH索引。B树索引是Mysql数据库中使用最频繁的索引类型，基本所有存储引擎都支持BTree索引。通常我们说的索引不出意外指的就是（B树）索引（实际是用B+树实现的，因为在查看表索引时，mysql一律打印BTREE，所以简称为B树索引）

• 查询方式：

• 主键索引区:PI(关联保存的时数据的地址)按主键查询,

• 普通索引区:si(关联的id的地址,然后再到达上面的地址)。所以按主键查询,速度最快

• B+tree性质：

• n棵子tree的节点包含n个关键字，不用来保存数据而是保存数据的索引。所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。所有的非终端结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。B+ 树中，数据对象的插入和删除仅在叶节点上进行。B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

• 哈希索引

• 简要说下，类似于数据结构中简单实现的HASH表（散列表）一样，当我们在mysql中用哈希索引时，主要就是通过Hash算法（常见的Hash算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的Hash值，与这条数据的行指针一并存入Hash表的对应位置；如果发生Hash碰撞（两个不同关键字的Hash值相同），则在对应Hash键下以链表形式存储。当然这只是简略模拟图。

* 索引的基本原理

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

• 索引的原理很简单，就是把无序的数据变成有序的查询

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

* 索引算法有哪些？

• 索引算法有 BTree算法和Hash算法

• BTree算法

• BTree是最常用的mysql数据库索引算法，也是mysql默认的算法。因为它不仅可以被用在=,&gt;,&gt;=,&lt;,&lt;=和between这些比较操作符上，而且还可以用于like操作符，只要它的查询条件是一个不以通配符开头的常量， 例如：

• -- 只要它的查询条件是一个不以通配符开头的常量select \* from user where name like 'jack%'; -- 如果一通配符开头，或者没有使用常量，则不会使用索引，例如： select \* from user where name like '%jack';

• Hash算法

• Hash Hash索引只能用于对等比较，例如=,&lt;=&gt;（相当于=）操作符。由于是一次定位数据，不像BTree索引需要从根节点到枝节点，最后才能访问到页节点这样多次IO访问，所以检索效率远高于BTree索引。

* 索引设计的原则？

• 适合索引的列是出现在where子句中的列，或者连接子句中指定的列基数较小的类，索引效果较差，没有必要在此列建立索引使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。

* 创建索引的原则

• 索引虽好，但也不是无限制的使用，最好符合一下几个原则

• 最左前缀匹配原则，组合索引非常重要的原则，mysql会一直向右匹配直到遇到范围查询(&gt;、&lt;、between、like)就停止匹配，比如a = 1 and b = 2 and c &gt; 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。较频繁作为查询条件的字段才去创建索引更新频繁字段不适合创建索引若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。定义有外键的数据列一定要建立索引。对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。对于定义为text、image和bit的数据类型的列不要建立索引。

* 创建索引的三种方式

• 第一种方式：在执行CREATE TABLE时创建索引

• CREATE TABLE user\_index2 (id INT auto\_increment PRIMARY KEY,first\_name VARCHAR (16),last\_name VARCHAR (16),id\_card VARCHAR (18),information text,KEY name (first\_name, last\_name),FULLTEXT KEY (information),UNIQUE KEY (id\_card));

• 第二种方式：使用ALTER TABLE命令去增加索引

• ALTER TABLE table\_name ADD INDEX index\_name (column\_list);

• ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。其中table\_name是要增加索引的表名，column\_list指出对哪些列进行索引，多列时各列之间用逗号分隔。索引名index\_name可自己命名，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

• 第三种方式：使用CREATE INDEX命令创建

• CREATE INDEX index\_name ON table\_name (column\_list);

• CREATE INDEX可对表增加普通索引或UNIQUE索引。（但是，不能创建PRIMARY KEY索引）

* 如何删除索引

• 根据索引名删除普通索引、唯一索引、全文索引：alter table 表名 drop KEY 索引名

• alter table user\_index drop KEY name;alter table user\_index drop KEY id\_card;alter table user\_index drop KEY information;

• 删除主键索引：alter table 表名 drop primary key（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：

• 需要取消自增长再行删除：

• alter table user\_index-- 重新定义字段MODIFY id int,drop PRIMARY KEY

• 但通常不会删除主键，因为设计主键一定与业务逻辑无关。

* 创建索引时需要注意什么？

• 非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

* 使用索引查询一定能提高查询的性能吗？为什么

• 通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

• 索引需要空间来存储，也需要定期维护， 每当有记录在表中增减或索引列被修改时，索引本身也会被修改。 这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5 次的磁盘I/O。 因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况:基于一个范围的检索，一般查询返回结果集小于表中记录数的30%基于非唯一性索引的检索

* 百万级别或以上的数据如何删除

• 关于索引：

• 关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询MySQL官方手册得知删除数据的速度和创建的索引数量是成正比的。

• 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）然后删除其中无用数据（此过程需要不到两分钟）删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟左右。与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

* 前缀索引

• 语法：index(field(10))，使用字段值的前10个字符建立索引，默认是使用字段的全部内容建立索引。

• 前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。

• 实操的难度：在于前缀截取的长度

• 我们可以利用select count(\*)/count(distinct left(password,prefixLen));，通过从调整prefixLen的值（从1自增）查看不同前缀长度的一个平均匹配度，接近1时就可以了（表示一个密码的前prefixLen个字符几乎能确定唯一一条记录）

* 什么是最左前缀原则？什么是最左匹配原则

• 顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。

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

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

* B树和B+树的区别

• 在B树中，你可以将键和值存放在内部节点和叶子节点；但在B+树中，内部节点都是键，没有值，叶子节点同时存放键和值。

• B+树的叶子节点有一条链相连，而B树的叶子节点各自独立。

* 使用B树的好处

• B树可以在内部节点同时存储键和值，因此，把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。这种特性使得B树在特定数据重复多次查询的场景中更加高效。

* 使用B+树的好处

• 由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。 B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

* 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。

* 数据库为什么使用B+树而不是B树

• B树只适合随机检索，而B+树同时支持随机检索和顺序检索；B+树空间利用率更高，可减少I/O次数，磁盘读写代价更低。一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。B+树的内部结点并没有指向关键字具体信息的指针，只是作为索引使用，其内部结点比B树小，盘块能容纳的结点中关键字数量更多，一次性读入内存中可以查找的关键字也就越多，相对的，IO读写次数也就降低了。而IO读写次数是影响索引检索效率的最大因素；B+树的查询效率更加稳定。B树搜索有可能会在非叶子结点结束，越靠近根节点的记录查找时间越短，只要找到关键字即可确定记录的存在，其性能等价于在关键字全集内做一次二分查找。而在B+树中，顺序检索比较明显，随机检索时，任何关键字的查找都必须走一条从根节点到叶节点的路，所有关键字的查找路径长度相同，导致每一个关键字的查询效率相当。B-树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树的叶子节点使用指针顺序连接在一起，只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。增删文件（节点）时，效率更高。因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

* 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，

• 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。

* 什么是聚簇索引？何时使用聚簇索引与非聚簇索引

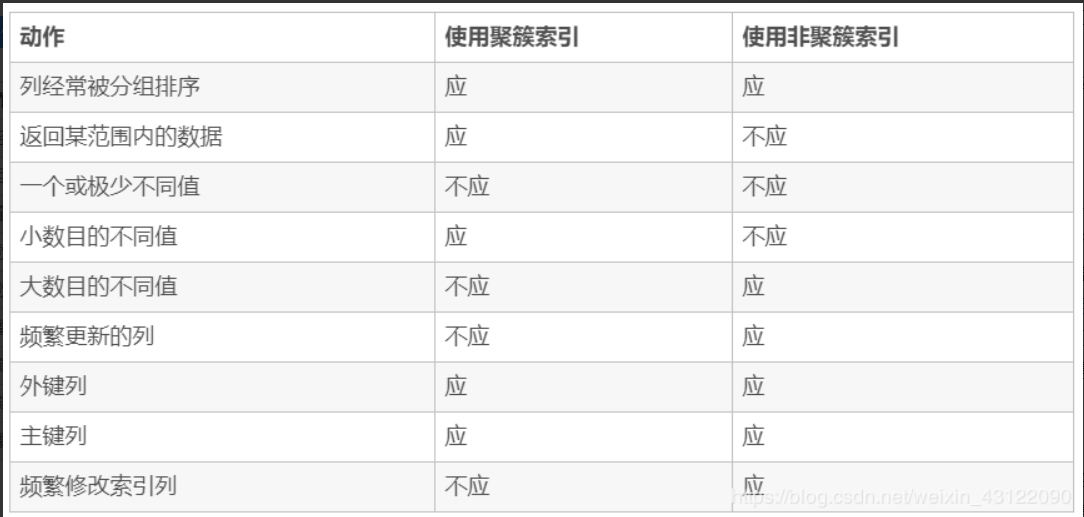
• 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据

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

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

• 何时使用聚簇索引与非聚簇索引

•



* 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 &lt; 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。

* 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。

### 事务

* 什么是数据库事务？

• 事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

• 事务最经典也经常被拿出来说例子就是转账了。

• 假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

* 说一下 ACID 是什么？

• 事物的四大特性

• 原子性：

• 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

• 一致性：

• 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；

• 隔离性：

• 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；

• 持久性：

• 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

• 关系性数据库需要遵循ACID规则

* 什么是脏读？幻读？不可重复读？

• 脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

* 什么是事务的隔离级别？MySQL的默认隔离级别是什么？

• 说一下数据库的事务隔离？

• 为了达到事务的四大特性，数据库定义了4种不同的事务隔离级别，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

•



• MySQL的默认隔离级别是什么？

• READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

• 注意：

• 这里需要注意的是：Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVVC（多版本并发控制），通过保存修改的旧版本信息来支持并发一致性读和回滚等特性。因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是READ-COMMITTED(读取提交内容):，但是你要知道的是InnoDB 存储引擎默认使用 \*\*REPEATABLE-READ（可重读）\*\*并不会有任何性能损失。InnoDB 存储引擎在 分布式事务 的情况下一般会用到\*\*SERIALIZABLE(可串行化)\*\*隔离级别。

### 锁

* 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。

* 从锁的类别上分MySQL都有哪些锁呢？

• 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。共享锁就是让多个线程同时获取一个锁。

• 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。排它锁也称作独占锁，一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

* 隔离级别与锁的关系

• 在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

* 按照锁的粒度分数据库锁有哪些？锁机制与InnoDB锁算法

• 在关系型数据库中，可以按照锁的粒度把数据库锁分为行级锁(INNODB引擎)、表级锁(MYISAM引擎)和页级锁(BDB引擎 )。

• 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

* 行级锁，表级锁和页级锁对比

• 行级锁 行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。

• 特点：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。

• 表级锁 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

• 特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

• 页级锁 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

• 特点：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

* MySQL中InnoDB引擎的行锁是怎么实现的？

• InnoDB是基于索引来完成行锁例: select \* from tab\_with\_index where id = 1 for update;for update 可以根据条件来完成行锁锁定，并且 id 是有索引键的列，如果 id 不是索引键那么InnoDB将完成表锁，并发将无从谈起

* 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

• 相关知识点：

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

* 什么是死锁？怎么解决？

• 死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

• 常见的解决死锁的方法

• 1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

• 如果业务不好处理,可以用分布式事务锁或者使用乐观锁

* 数据库的乐观锁和悲观锁是什么？怎么实现的？

• 数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观并发控制（乐观锁）和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

• 悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制

• //核心SQL,主要靠for updateselect status from t\_goods where id=1 for update;

• 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用版本号机制或CAS算法实现。

• //核心SQLupdate table set x=x+1, version=version+1 where id=#{id} and version=#{version};

• 两种锁的使用场景

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

### 视图

* 为什么要使用视图？什么是视图？

• 为了提高复杂SQL语句的复用性和表操作的安全性，MySQL数据库管理系统提供了视图特性。所谓视图，本质上是一种虚拟表，在物理上是不存在的，其内容与真实的表相似，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储存的数据值形式存在。行和列数据来自定义视图的查询所引用基本表，并且在具体引用视图时动态生成。

• 视图使开发者只关心感兴趣的某些特定数据和所负责的特定任务，只能看到视图中所定义的数据，而不是视图所引用表中的数据，从而提高了数据库中数据的安全性。

* 视图有哪些特点？

• 视图的特点如下:

• 视图的列可以来自不同的表，是表的抽象和在逻辑意义上建立的新关系。视图是由基本表(实表)产生的表(虚表)。视图的建立和删除不影响基本表。对视图内容的更新(添加，删除和修改)直接影响基本表。当视图来自多个基本表时，不允许添加和删除数据。

• 视图的操作包括创建视图，查看视图，删除视图和修改视图。

* 视图的使用场景有哪些？

• 视图根本用途：简化sql查询，提高开发效率。如果说还有另外一个用途那就是兼容老的表结构。

• 下面是视图的常见使用场景：

• 重用SQL语句；简化复杂的SQL操作。在编写查询后，可以方便的重用它而不必知道它的基本查询细节；使用表的组成部分而不是整个表；保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限；更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

* 视图的优点

• 查询简单化。视图能简化用户的操作数据安全性。视图使用户能以多种角度看待同一数据，能够对机密数据提供安全保护逻辑数据独立性。视图对重构数据库提供了一定程度的逻辑独立性

* 视图的缺点

• 性能。数据库必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，即使是视图的一个简单查询，数据库也把它变成一个复杂的结合体，需要花费一定的时间。修改限制。当用户试图修改视图的某些行时，数据库必须把它转化为对基本表的某些行的修改。事实上，当从视图中插入或者删除时，情况也是这样。对于简单视图来说，这是很方便的，但是，对于比较复杂的视图，可能是不可修改的这些视图有如下特征：1.有UNIQUE等集合操作符的视图。2.有GROUP BY子句的视图。3.有诸如AVG\SUM\MAX等聚合函数的视图。 4.使用DISTINCT关键字的视图。5.连接表的视图（其中有些例外）

* 什么是游标？

• 游标是系统为用户开设的一个数据缓冲区，存放SQL语句的执行结果，每个游标区都有一个名字。用户可以通过游标逐一获取记录并赋给主变量，交由主语言进一步处理。

### 存储过程与函数

* 什么是存储过程？有哪些优缺点？

• 存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快

* 优点

• 存储过程是预编译过的，执行效率高。存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。安全性高，执行存储过程需要有一定权限的用户。存储过程可以重复使用，减少数据库开发人员的工作量。

* 缺点

• 调试麻烦，但是用 PL/SQL Developer 调试很方便！弥补这个缺点。移植问题，数据库端代码当然是与数据库相关的。但是如果是做工程型项目，基本不存在移植问题。重新编译问题，因为后端代码是运行前编译的，如果带有引用关系的对象发生改变时，受影响的存储过程、包将需要重新编译（不过也可以设置成运行时刻自动编译）。如果在一个程序系统中大量的使用存储过程，到程序交付使用的时候随着用户需求的增加会导致数据结构的变化，接着就是系统的相关问题了，最后如果用户想维护该系统可以说是很难很难、而且代价是空前的，维护起来更麻烦。

### 触发器

* 什么是触发器？触发器的使用场景有哪些？

• 触发器是用户定义在关系表上的一类由事件驱动的特殊的存储过程。触发器是指一段代码，当触发某个事件时，自动执行这些代码。

• 使用场景

• 可以通过数据库中的相关表实现级联更改。实时监控某张表中的某个字段的更改而需要做出相应的处理。例如可以生成某些业务的编号。注意不要滥用，否则会造成数据库及应用程序的维护困难。大家需要牢记以上基础知识点，重点是理解数据类型CHAR和VARCHAR的差异，表存储引擎InnoDB和MyISAM的区别。

* MySQL中都有哪些触发器？

• 在MySQL数据库中有如下六种触发器：

• Before InsertAfter InsertBefore UpdateAfter UpdateBefore DeleteAfter Delete

### 常用SQL语句

* SQL语句主要分为哪几类

• 数据定义语言DDL（Data Ddefinition Language）CREATE，DROP，ALTER主要为以上操作 即对逻辑结构等有操作的，其中包括表结构，视图和索引。

• 数据查询语言DQL（Data Query Language）SELECT这个较为好理解 即查询操作，以select关键字。各种简单查询，连接查询等 都属于DQL。

• 数据操纵语言DML（Data Manipulation Language）INSERT，UPDATE，DELETE主要为以上操作 即对数据进行操作的，对应上面所说的查询操作 DQL与DML共同构建了多数初级程序员常用的增删改查操作。而查询是较为特殊的一种 被划分到DQL中。

• 数据控制功能DCL（Data Control Language）GRANT，REVOKE，COMMIT，ROLLBACK主要为以上操作 即对数据库安全性完整性等有操作的，可以简单的理解为权限控制等

* SQL语句的语法顺序：

• SELECTFROMJOINONWHEREGROUP BYHAVINGUNIONORDER BYLIMIT

* 超键、候选键、主键、外键分别是什么？

• 超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

• 候选键：是最小超键，即没有冗余元素的超键。

• 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

• 外键：在一个表中存在的另一个表的主键称此表的外键。

* SQL 约束有哪几种？

• NOT NULL: 用于控制字段的内容一定不能为空（NULL）。

• UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束。

• PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个。

• FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

• CHECK: 用于控制字段的值范围。

* 六种关联查询

• 交叉连接（CROSS JOIN）

• 内连接（INNER JOIN）

• 内连接分为三类

• 等值连接：ON A.id=B.id不等值连接：ON A.id &gt; B.id自连接：SELECT \* FROM A T1 INNER JOIN A T2 ON T1.id=T2.pid

• select r.\*,s.\* from r inner join s on r.c=s.c

• 外连接（LEFT JOIN/RIGHT JOIN）

• 左外连接：LEFT OUTER JOIN, 以左表为主，先查询出左表，按照ON后的关联条件匹配右表，没有匹配到的用NULL填充，可以简写成LEFT JOIN右外连接：RIGHT OUTER JOIN, 以右表为主，先查询出右表，按照ON后的关联条件匹配左表，没有匹配到的用NULL填充，可以简写成RIGHT JOIN

• select r.\*,s.\* from r left join s on r.c=s.c

• select r.\*,s.\* from r right join s on r.c=s.c

• 联合查询（UNION与UNION ALL）

• SELECT \* FROM A UNION SELECT \* FROM B UNION ...

• 就是把多个结果集集中在一起，UNION前的结果为基准，需要注意的是联合查询的列数要相等，相同的记录行会合并如果使用UNION ALL，不会合并重复的记录行效率 UNION 高于 UNION ALL

• 全连接（FULL JOIN）

• SELECT \* FROM A LEFT JOIN B ON A.id=B.id UNIONSELECT \* FROM A RIGHT JOIN B ON A.id=B.id

• MySQL不支持全连接可以使用LEFT JOIN 和UNION和RIGHT JOIN联合使用

• select r.\*,s.\* from r full join s on r.c=s.c

• 交叉连接（CROSS JOIN）

• select r.\*,s.\* from r,s

• SELECT \* FROM A,B(,C)或者SELECT \* FROM A CROSS JOIN B (CROSS JOIN C)#没有任何关联条件，结果是笛卡尔积，结果集会很大，没有意义，很少使用内连接（INNER JOIN）SELECT \* FROM A,B WHERE A.id=B.id或者SELECT \* FROM A INNER JOIN B ON A.id=B.id多表中同时符合某种条件的数据记录的集合，INNER JOIN可以缩写为JOIN

• mysql 的内连接、左连接、右连接有什么区别？

• 内连接,显示两个表中有联系的所有数据;

• 左链接,以左表为参照,显示所有数据,右表中没有则以null显示

• 右链接,以右表为参照显示数据，,左表中没有则以null显示

* 什么是子查询

• 条件：一条SQL语句的查询结果做为另一条查询语句的条件或查询结果嵌套：多条SQL语句嵌套使用，内部的SQL查询语句称为子查询。

* mysql中 in 和 exists 区别

• mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

• 如果查询的两个表大小相当，那么用in和exists差别不大。如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

* char 和 varchar 的区别是什么？

• char的特点

• char表示定长字符串，长度是固定的；

• 如果插入数据的长度小于char的固定长度时，则用空格填充；

• 因为长度固定，所以存取速度要比varchar快很多，甚至能快50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法；

• 对于char来说，最多能存放的字符个数为255，和编码无关

• varchar的特点

• varchar表示可变长字符串，长度是可变的；

• 插入的数据是多长，就按照多长来存储；

• varchar在存取方面与char相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法；

• 对于varchar来说，最多能存放的字符个数为65532

• 总之，结合性能角度（char更快）和节省磁盘空间角度（varchar更小），具体情况还需具体来设计数据库才是妥当的做法。

* 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)。在早期 MySQL 版本中， 50 代表字节数，现在代表字符数。

* int(20)中20的涵义

• 是指显示字符的长度。20表示最大显示宽度为20，但仍占4字节存储，存储范围不变；不影响内部存储，只是影响带 zerofill 定义的 int 时，前面补多少个 0，易于报表展示

* mysql中int(10)和char(10)以及varchar(10)的区别

• int(10)的10表示显示的数据的长度，不是存储数据的大小；chart(10)和varchar(10)的10表示存储数据的大小，即表示存储多少个字符。

• char(10)表示存储定长的10个字符，不足10个就用空格补齐，占用更多的存储空间

• varchar(10)表示存储10个变长的字符，存储多少个就是多少个，空格也按一个字符存储，这一点是和char(10)的空格不同的，char(10)的空格表示占位不算一个字符

* float 和 double 的区别是什么？

• FLOAT类型数据可以存储至多8位十进制数，并在内存中占4字节。

• DOUBLE类型数据可以存储至多18位十进制数，并在内存中占8字节。

* drop、delete、truncate的区别是什么？

• 三者都表示删除，但是三者有一些差别：

•



• 因此，在不再需要一张表的时候，用drop；在想删除部分数据行时候，用delete；在保留表而删除所有数据的时候用truncate。

* UNION与UNION ALL的区别？

• 如果使用UNION ALL，不会合并重复的记录行效率 UNION 高于 UNION ALL

### SQL优化

* 说出一些数据库优化方面的经验?

• 有外键约束的话会影响增删改的性能，如果应用程序可以保证数据库的完整性那就去除外键Sql语句全部大写，特别是列名大写，因为数据库的机制是这样的，sql语句发送到数据库服务器，数据库首先就会把sql编译成大写在执行，如果一开始就编译成大写就不需要了把sql编译成大写这个步骤了如果应用程序可以保证数据库的完整性，可以不需要按照三大范式来设计数据库其实可以不必要创建很多索引，索引可以加快查询速度，但是索引会消耗磁盘空间如果是jdbc的话，使用PreparedStatement不使用Statement，来创建SQl，PreparedStatement的性能比Statement的速度要快，使用PreparedStatement对象SQL语句会预编译在此对象中，PreparedStatement对象可以多次高效的执行

* 怎么优化SQL查询语句吗

• 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用索引可以提高查询SELECT子句中避免使用\*号，尽量全部大写SQL应尽量避免在 where 子句中对字段进行 is null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，使用 IS NOT NULLwhere 子句中使用 or 来连接条件，也会导致引擎放弃使用索引而进行全表扫描in 和 not in 也要慎用，否则会导致全表扫描

* 你怎么知道SQL语句性能是高还是低

• 查看SQL的执行时间使用explain关键字可以模拟优化器执行SQL查询语句，从而知道MYSQL是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈。

* SQL的执行顺序

• FROM：将数据从硬盘加载到数据缓冲区，方便对接下来的数据进行操作。WHERE：从基表或视图中选择满足条件的元组。（不能使用聚合函数）JOIN（如right left 右连接-------从右边表中读取某个元组，并且找到该元组在左边表中对应的元组或元组集）ON：join on实现多表连接查询，推荐该种方式进行多表查询，不使用子查询。GROUP BY：分组，一般和聚合函数一起使用。HAVING：在元组的基础上进行筛选，选出符合条件的元组。（一般与GROUP BY进行连用）SELECT：查询到得所有元组需要罗列的哪些列。DISTINCT：去重的功能。UNION：将多个查询结果合并（默认去掉重复的记录）。ORDER BY：进行相应的排序。LIMIT 1：显示输出一条数据记录（元组）

## MyBatis

### MyBatis简介

* MyBatis是什么？

• 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，灵活度高。

• 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。

* Mybatis优缺点

• 优点

• 与传统的数据库访问技术相比，ORM有以下优点：

• 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

• 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

• 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）

• 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

• 能够与Spring很好的集成

• 缺点

• 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

• 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

* Hibernate 和 MyBatis 的区别

• 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。

* 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。

* 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。

* 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。

### 映射器

* MyBatis中#{}和${}的区别

• #{}是占位符，预编译处理；${}是拼接符，字符串替换，没有预编译处理。Mybatis在处理#{}时，#{}传入参数是以字符串传入，会将SQL中的#{}替换为?号，调用PreparedStatement的set方法来赋值。#{} 可以有效的防止SQL注入，提高系统安全性；${} 不能防止SQL 注入#{} 的变量替换是在DBMS 中；${} 的变量替换是在 DBMS 外

* 模糊查询like语句该怎么写

• 1 ’%${question}%’ 可能引起SQL注入，不推荐2 "%"#{question}"%" 注意：因为#{…}解析成sql语句时候，会在变量外侧自动加单引号’ '，所以这里 % 需要使用双引号" "，不能使用单引号 ’ '，不然会查不到任何结果。3 CONCAT(’%’,#{question},’%’) 使用CONCAT()函数，（推荐）4 使用bind标签（不推荐）&lt;select id="listUserLikeUsername" resultType="com.jourwon.pojo.User"&gt;  &lt;bind name="pattern" value="'%' + username + '%'" /&gt;  select id,sex,age,username,password from person where username LIKE #{pattern}&lt;/select&gt;

* 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);&lt;select id="selectUser" resultMap="UserResultMap"&gt; select \* from user where user\_name = #{0} and dept\_id = #{1}&lt;/select&gt;

• #{}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);&lt;select id="selectUser" resultMap="UserResultMap"&gt; select \* from user where user\_name = #{userName} and dept\_id = #{deptId}&lt;/select&gt;

• #{}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还是比较直观的，（推荐使用）。

• 方法3：Map传参法

• public User selectUser(Map&lt;String, Object&gt; params);&lt;select id="selectUser" parameterType="java.util.Map" resultMap="UserResultMap"&gt; select \* from user where user\_name = #{userName} and dept\_id = #{deptId}&lt;/select&gt;

• #{}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);&lt;select id="selectUser" parameterType="com.jourwon.pojo.User" resultMap="UserResultMap"&gt; select \* from user where user\_name = #{userName} and dept\_id = #{deptId}&lt;/select&gt;

• #{}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）。

* Mybatis如何执行批量操作

• 使用foreach标签foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach标签的属性主要有item，index，collection，open，separator，close。item   表示集合中每一个元素进行迭代时的别名，随便起的变量名；index   指定一个名字，用于表示在迭代过程中，每次迭代到的位置，不常用；open   表示该语句以什么开始，常用“(”；separator 表示在每次进行迭代之间以什么符号作为分隔符，常用“,”；close   表示以什么结束，常用“)”。在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有一下3种情况：如果传入的是单参数且参数类型是一个List的时候，collection属性值为list如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个Map的，map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key具体用法如下：&lt;!-- 批量保存(foreach插入多条数据两种方法) int addEmpsBatch(@Param("emps") List&lt;Employee&gt; emps); --&gt;&lt;!-- MySQL下批量保存，可以foreach遍历 mysql支持values(),(),()语法 --&gt; //推荐使用&lt;insert id="addEmpsBatch"&gt; INSERT INTO emp(ename,gender,email,did) VALUES &lt;foreach collection="emps" item="emp" separator=","&gt; (#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id}) &lt;/foreach&gt;&lt;/insert&gt;复制代码&lt;!-- 这种方式需要数据库连接属性allowMutiQueries=true的支持 如jdbc.url=jdbc:mysql://localhost:3306/mybatis?allowMultiQueries=true --&gt; &lt;insert id="addEmpsBatch"&gt; &lt;foreach collection="emps" item="emp" separator=";"&gt; INSERT INTO emp(ename,gender,email,did) VALUES(#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id}) &lt;/foreach&gt;&lt;/insert&gt;复制代码使用ExecutorType.BATCHMybatis内置的ExecutorType有3种，默认为simple,该模式下它为每个语句的执行创建一个新的预处理语句，单条提交sql；而batch模式重复使用已经预处理的语句，并且批量执行所有更新语句，显然batch性能将更优； 但batch模式也有自己的问题，比如在Insert操作时，在事务没有提交之前，是没有办法获取到自增的id，这在某型情形下是不符合业务要求的具体用法如下：//批量保存方法测试@Test public void testBatch() throws IOException{ SqlSessionFactory sqlSessionFactory = getSqlSessionFactory(); //可以执行批量操作的sqlSession SqlSession openSession = sqlSessionFactory.openSession(ExecutorType.BATCH); //批量保存执行前时间 long start = System.currentTimeMillis(); try { EmployeeMapper mapper = openSession.getMapper(EmployeeMapper.class); for (int i = 0; i &lt; 1000; i++) { mapper.addEmp(new Employee(UUID.randomUUID().toString().substring(0, 5), "b", "1")); } openSession.commit(); long end = System.currentTimeMillis(); //批量保存执行后的时间 System.out.println("执行时长" + (end - start)); //批量 预编译sql一次==》设置参数==》10000次==》执行1次 677 //非批量 （预编译=设置参数=执行 ）==》10000次 1121 } finally { openSession.close(); }}复制代码mapper和mapper.xml如下public interface EmployeeMapper { //批量保存员工 Long addEmp(Employee employee);}复制代码&lt;mapper namespace="com.jourwon.mapper.EmployeeMapper" &lt;!--批量保存员工 --&gt; &lt;insert id="addEmp"&gt; insert into employee(lastName,email,gender) values(#{lastName},#{email},#{gender}) &lt;/insert&gt;&lt;/mapper&gt;

* 如何获取生成的主键

• 新增标签中添加：keyProperty=" ID " 即可

• &lt;insert id="insert" useGeneratedKeys="true" keyProperty="userId" &gt; insert into user( user\_name, user\_password, create\_time) values(#{userName}, #{userPassword} , #{createTime, jdbcType= TIMESTAMP})&lt;/insert&gt;

* 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。&lt;select id="getOrder" parameterType="int" resultType="com.jourwon.pojo.Order"&gt; select order\_id id, order\_no orderno ,order\_price price form orders where order\_id=#{id};&lt;/select&gt;第2种： 通过&lt;resultMap&gt;来映射字段名和实体类属性名的一一对应的关系。&lt;select id="getOrder" parameterType="int" resultMap="orderResultMap"&gt;select \* from orders where order\_id=#{id}&lt;/select&gt;&lt;resultMap type="com.jourwon.pojo.Order" id="orderResultMap"&gt; &lt;!–用id属性来映射主键字段–&gt; &lt;id property="id" column="order\_id"&gt; &lt;!–用result属性来映射非主键字段，property为实体类属性名，column为数据库表中的属性–&gt;&lt;result property ="orderno" column ="order\_no"/&gt;&lt;result property="price" column="order\_price" /&gt;&lt;/reslutMap&gt;

* 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。在 sqlMapConfig.xml 中配置 mapper.xml 的位置&lt;mappers&gt; &lt;mapper resource="mapper.xml 文件的地址" /&gt; &lt;mapper resource="mapper.xml 文件的地址" /&gt;&lt;/mappers&gt;定义 mapper 接口实现类集成 SqlSessionDaoSupportmapper 方法中可以 this.getSqlSession()进行数据增删改查。spring 配置&lt;bean id=" " class="mapper 接口的实现"&gt; &lt;property name="sqlSessionFactory" ref="sqlSessionFactory"&gt;&lt;/property&gt;&lt;/bean&gt;

• 第二种：使用 org.mybatis.spring.mapper.MapperFactoryBean：在 sqlMapConfig.xml 中配置 mapper.xml 的位置，如果 mapper.xml 和mappre 接口的名称相同且在同一个目录，这里可以不用配置定义 mapper 接口：&lt;mappers&gt; &lt;mapper resource="mapper.xml 文件的地址" /&gt; &lt;mapper resource="mapper.xml 文件的地址" /&gt;&lt;/mappers&gt;mapper.xml 中的 namespace 为 mapper 接口的地址mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致Spring 中定义&lt;bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean"&gt; &lt;property name="mapperInterface" value="mapper 接口地址" /&gt; &lt;property name="sqlSessionFactory" ref="sqlSessionFactory" /&gt;&lt;/bean&gt;

• 第三种：使用 mapper 扫描器：

• mapper.xml 文件编写：mapper.xml 中的 namespace 为 mapper 接口的地址；mapper 接口中的方法名和 mapper.xml 中的定义的 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。定义 mapper 接口：注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录配置 mapper 扫描器：&lt;bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"&gt; &lt;property name="basePackage" value="mapper 接口包地址 "&gt;&lt;/property&gt; &lt;property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/&gt;&lt;/bean&gt;使用扫描器后从 spring 容器中获取 mapper 的实现对象。

* 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，这样比起原来了SqlSession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；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。

* 使用MyBatis的mapper接口调用时有哪些要求？

• Mapper接口方法名和mapper.xml中定义的每个sql的id相同。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。Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同。Mapper.xml文件中的namespace即是mapper接口的类路径。

* 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接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

* 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；毕竟namespace不是必须的，只是最佳实践而已。原因就是namespace+id是作为Map&lt;String, MappedStatement&gt;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。

* 简述Mybatis的Xml映射文件和Mybatis内部数据结构之间的映射关系？

• 答：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，&lt;parameterMap&gt;标签会被解析为ParameterMap对象，其每个子元素会被解析为ParameterMapping对象。&lt;resultMap&gt;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&lt;select&gt;、&lt;insert&gt;、&lt;update&gt;、&lt;delete&gt;标签均会被解析为MappedStatement对象，标签内的sql会被解析为BoundSql对象。

* 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&lt;resultMap&gt;标签，逐一定义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，将列别名书写为对象属性名，比如T\_NAME AS NAME，对象属性名一般是name，小写，但是列名不区分大小写，Mybatis会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成T\_NAME AS NaMe，Mybatis一样可以正常工作。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。

* Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签？

• 还有很多其他的标签，&lt;resultMap&gt;、&lt;parameterMap&gt;、&lt;sql&gt;、&lt;include&gt;、&lt;selectKey&gt;，加上动态sql的9个标签，trim|where|set|foreach|if|choose|when|otherwise|bind等，其中&lt;sql&gt;为sql片段标签，通过&lt;include&gt;标签引入sql片段，&lt;selectKey&gt;为不支持自增的主键生成策略标签。

* 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标签也就可以正常解析完成了。

* Mybatis能执行一对多，一对一的联系查询吗，有哪些实现方法

• 能，不止可以一对多，一对一还可以多对多，一对多实现方式：单独发送一个SQL去查询关联对象，赋给主对象，然后返回主对象使用嵌套查询，似JOIN查询，一部分是A对象的属性值，另一部分是关联对 象 B的属性值，好处是只要发送一个属性值，就可以把主对象和关联对象查出来子查询

* Mybatis是否可以映射Enum枚举类？

• Mybatis可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和getResult()接口方法。TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占位符参数和获取列查询结果。

* Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理吗？

• 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，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。

* Mybatis是如何进行分页的？分页插件的原理是什么？

• Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页，可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。举例：select \* from student，拦截sql后重写为：select t.\* from (select \* from student) t limit 0, 10

* 简述Mybatis的插件运行原理，以及如何编写一个插件。

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

* Mybatis的一级、二级缓存

• 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。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&lt;cache/&gt;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。

## Redis

### 概述

* 什么是Redis？

• Redis 是一个使用 C 语言写成的，开源的高性能key-value非关系缓存数据库。它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。Redis的数据都基于缓存的，所以很快，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也可以实现数据写入磁盘中，保证了数据的安全不丢失，而且Redis的操作是原子性的。

* Redis有哪些优缺点？

• 优点

• 读写性能优异， Redis能读的速度是110000次/s，写的速度是81000次/s。支持数据持久化，支持AOF和RDB两种持久化方式。支持事务，Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。数据结构丰富，除了支持string类型的value外还支持hash、set、zset、list等数据结构。支持主从复制，主机会自动将数据同步到从机，可以进行读写分离

• 缺点

• 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。Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

* 使用redis有哪些好处？

• (1) 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都很低(2)支持丰富数据类型，支持string，list，set，sorted set，hash(3) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行(4) 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

* 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。

• 高性能：

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

• 高并发：

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

* 为什么要用 Redis 而不用 map/guava 做缓存?

• 缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

• 使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached服务的高可用，整个程序架构上较为复杂。

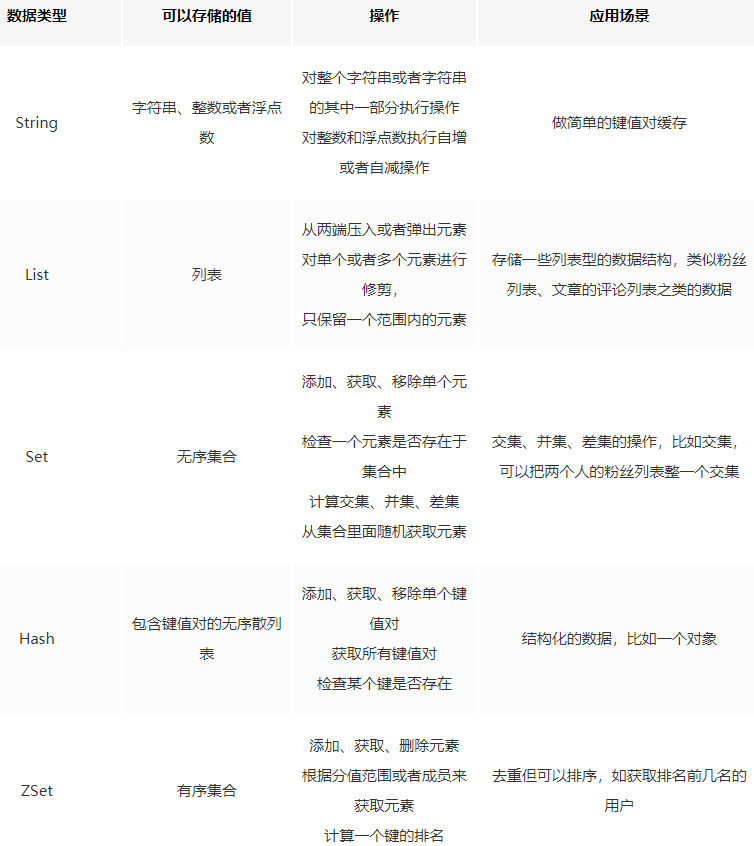
* Redis为什么这么快

• 1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是O(1)；2、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；4、使用多路 I/O 复用模型，非阻塞 IO；5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

* Redis有哪些数据类型

• Redis主要有5种数据类型，包括String，List，Set，Zset，Hash，满足大部分的使用要求

•



* Redis的应用场景

• 计数器

• 可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

• 缓存

• 将热点数据放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

• 会话缓存

• 可以使用 Redis 来统一存储多台应用服务器的会话信息。当应用服务器不再存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

• 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。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。

• 查找表

• 例如 DNS 记录就很适合使用 Redis 进行存储。查找表和缓存类似，也是利用了 Redis 快速的查找特性。但是查找表的内容不能失效，而缓存的内容可以失效，因为缓存不作为可靠的数据来源。

• 消息队列(发布/订阅功能)

• List 是一个双向链表，可以通过 lpush 和 rpop 写入和读取消息。不过最好使用 Kafka、RabbitMQ 等消息中间件。

• 分布式锁实现

• 在分布式场景下，无法使用单机环境下的锁来对多个节点上的进程进行同步。可以使用 Redis 自带的 SETNX 命令实现分布式锁，除此之外，还可以使用官方提供的 RedLock 分布式锁实现。

• 其它

• Set 可以实现交集、并集等操作，从而实现共同好友等功能。ZSet 可以实现有序性操作，从而实现排行榜等功能。

* 持久化

• 什么是Redis持久化？ 持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

* Redis 的持久化机制是什么？各自的优缺点？

• Redis 提供两种持久化机制 RDB（默认） 和 AOF 机制:

• RDB：是Redis DataBase缩写快照

• RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。

• 优点：

• 1、只有一个文件 dump.rdb，方便持久化。2、容灾性好，一个文件可以保存到安全的磁盘。3、性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能4.相对于数据集大时，比 AOF 的启动效率更高。

• 缺点：

• 1、数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)2、AOF（Append-only file)持久化方式： 是指所有的命令行记录以 redis 命令请 求协议的格式完全持久化存储)保存为 aof 文件。

• AOF：持久化：

• AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复

• 优点：

• 1、数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。2、通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。3、AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会对命令 进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）)

• 缺点：

• 1、AOF 文件比 RDB 文件大，且恢复速度慢。2、数据集大的时候，比 rdb 启动效率低。俩种持久化的优缺点是什么？AOF文件比RDB更新频率高，优先使用AOF还原数据。AOF比RDB更安全也更大RDB性能比AOF好如果两个都配了优先加载AOF

* 如何选择合适的持久化方式

• 一般来说， 如果想达到足以媲美PostgreSQL的数据安全性，你应该同时使用两种持久化功能。在这种情况下，当 Redis 重启的时候会优先载入AOF文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。有很多用户都只使用AOF持久化，但并不推荐这种方式，因为定时生成RDB快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外，使用RDB还可以避免AOF程序的bug。如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

* Redis持久化数据和缓存怎么做扩容？

• 如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

* Redis的过期键的删除策略

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

• 过期策略通常有以下三种：

• 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

• Redis中同时使用了惰性过期和定期过期两种过期策略。

* Redis key的过期时间和永久有效分别怎么设置？

• expire和persist命令。

* 我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

• 除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：1、定时去清理过期的缓存；2、当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

• 两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

* 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。

* Redis的内存淘汰策略有哪些

• Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

• 全局的键空间选择性移除

• noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。（这个是最常用的）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

• 设置过期时间的键空间选择性移除

• volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

• 总结

• Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

* Redis主要消耗什么物理资源？

• 内存。

* Redis的内存用完了会发生什么？

• 如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

* Redis如何做内存优化？

• 可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。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

### 线程模型

* Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器（file event handler）。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。
* 文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。
* 虽然文件事件处理器以单线程方式运行， 但通过使用 I/O 多路复用程序来监听多个套接字， 文件事件处理器既实现了高性能的网络通信模型， 又可以很好地与 redis 服务器中其他同样以单线程方式运行的模块进行对接， 这保持了 Redis 内部单线程设计的简单性。

### 事务

* 什么是事务？

• 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。

* Redis事务的概念

• Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

* Redis事务的三个阶段

• 事务开始 MULTI命令入队事务执行 EXEC事务执行过程中，如果服务端收到有EXEC、DISCARD、WATCH、MULTI之外的请求，将会把请求放入队列中排队

* Redis事务相关命令

• Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的Redis会将一个事务中的所有命令序列化，然后按顺序执行。redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。如果在一个事务中的命令出现错误，那么所有的命令都不会执行；如果在一个事务中出现运行错误，那么正确的命令会被执行。WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。 当操作被打断时，返回空值 nil 。通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。UNWATCH命令可以取消watch对所有key的监控。

* 事务管理（ACID）概述

• 原子性（Atomicity）原子性是指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。一致性（Consistency）事务前后数据的完整性必须保持一致。隔离性（Isolation）多个事务并发执行时，一个事务的执行不应影响其他事务的执行持久性（Durability）持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在\_AOF\_持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

* Redis事务支持隔离性吗

• Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

* Redis事务保证原子性吗，支持回滚吗

• Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

* Redis事务其他实现

• 基于Lua脚本，Redis可以保证脚本内的命令一次性、按顺序地执行，其同时也不提供事务运行错误的回滚，执行过程中如果部分命令运行错误，剩下的命令还是会继续运行完基于中间标记变量，通过另外的标记变量来标识事务是否执行完成，读取数据时先读取该标记变量判断是否事务执行完成。但这样会需要额外写代码实现，比较繁琐

### 集群方案

### 使用存储流程

* 咱们用的缓存是通过注解的方式，一共有四个，咱们框架用的是Cacheable，value就像一个咱们java包名，你后面查的所有东西都在这个下面找，找到就取出，没有找到就去数据库中查查找，然后再存入缓存，key就是你要查的东西，他就会先去咱定义的value中拿去。（必须是唯一的类似id一样），不然会出现冲突（我的1.0版本理解）

# 数据结构与算法篇

## 数据结构跟语言有关系吗？算法的实现和语言有关系吗？

## 都了解过哪些数据结构？分别对应的使用场景了解吗？

## 说几个你知道的排序算法，以及对应的时间复杂度？

## 算法的时间复杂度是怎么计算的？

## 实现一个斐波那契数列

## 给定一个字符串，找出该字符串的最长回文子串

## 翻转给定的字符串

## 替换指定的字符串

## 查找数组中重复的数字

## 查找数组中第2大的数字

## 有序数组中某个数出现的次数

## 删除链表中的指定节点

## 翻转单链表

## 将两个有序单向链表合并，使得合并后的链表仍然有序

## 给定一个链表，判断链表中是否存在环。

### 如果不考虑空间复杂度，可以使用一个map记录走过的节点，当遇到第一个在map中存在的节点时，就说明回到了出发点，即链表有环，同时也找到了环的入口。

### 快慢指针，即采用两个指针walker和runner，walker每次移动一步而runner每次移动两步。当walker和runner第一次相遇时，证明链表有环

## 前序遍历一个二叉树

## 根据前序和中序构造一颗二叉树

## 翻转二叉树

## 判断一棵二叉树是否是二叉搜索树(BST)

## 判断二叉树是否是完全二叉树

## 判断一棵二叉树是否是平衡二叉树

## 链表跟数组的区别以及优缺点

## 栈和队列有什么特点.区别是什么

## 怎么实现一个栈

## 怎么实现一个队列

## 学习过二叉树吗？二叉树的三种遍历方式

## 非递归方式遍历二叉树

## 二叉树有什么缺点

## 自平衡二叉树有了解吗

## 自平衡二叉树的翻转过程

## 自平衡二叉树有什么缺点

## 红黑树有了解吗

## 红黑树的四种需要平衡的情况

## B Tree原理

## B-Tree原理

## 堆

## 二项队列

## 哈夫曼树

## 伸展树

# 设计模式

## 什么是设计模式？

### 简而言之，设计模式（Design pattern）就是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的，该踩的坑别人已经替你踩了，然后总结出一些比较好的方案，你学着用就好！

## 设计模式的类型

### 创建型模式

* 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活，包括：

• 工厂模式（Factory Pattern）抽象工厂模式（Abstract Factory Pattern）单例模式（Singleton Pattern）建造者模式（Builder Pattern）原型模式（Prototype Pattern）

### 结构型模式

* 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式，包括：

• 适配器模式（Adapter Pattern）桥接模式（Bridge Pattern）过滤器模式（Filter、Criteria Pattern）组合模式（Composite Pattern）装饰器模式（Decorator Pattern）外观模式（Facade Pattern）享元模式（Flyweight Pattern）代理模式（Proxy Pattern）

### 行为型模式

* 这些设计模式特别关注对象之间的通信，包括：

• 责任链模式（Chain of Responsibility Pattern）命令模式（Command Pattern）解释器模式（Interpreter Pattern）迭代器模式（Iterator Pattern）中介者模式（Mediator Pattern）备忘录模式（Memento Pattern）观察者模式（Observer Pattern）状态模式（State Pattern）空对象模式（Null Object Pattern）策略模式（Strategy Pattern）模板模式（Template Pattern）访问者模式（Visitor Pattern）

### J2EE 模式

* 这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的，包括：

• MVC 模式（MVC Pattern）业务代表模式（Business Delegate Pattern）组合实体模式（Composite Entity Pattern）数据访问对象模式（Data Access Object Pattern）前端控制器模式（Front Controller Pattern）拦截过滤器模式（Intercepting Filter Pattern）服务定位器模式（Service Locator Pattern）传输对象模式（Transfer Object Pattern）

## 设计模式的六大原则

### 开闭原则（Open Close Principle）：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

### 里氏代换原则（Liskov Substitution Principle）：里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

### 依赖倒转原则（Dependence Inversion Principle）：这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

### 接口隔离原则（Interface Segregation Principle）：这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

### 迪米特法则，又称最少知道原则（Demeter Principle）：最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

### 合成复用原则（Composite Reuse Principle）：尽量使用合成/聚合的方式，而不是使用继承。

## 单例模式

### 1.什么是单例

* 保证一个类只有一个实例，并且提供一个访问该全局访问点

### 2.那些地方用到了单例模式

* 网站的计数器，一般也是采用单例模式实现，否则难以同步。应用程序的日志应用，一般都是单例模式实现，只有一个实例去操作才好，否则内容不好追加显示。多线程的线程池的设计一般也是采用单例模式，因为线程池要方便对池中的线程进行控制Windows的（任务管理器）就是很典型的单例模式，他不能打开俩个windows的（回收站）也是典型的单例应用。在整个系统运行过程中，回收站只维护一个实例。

### 3.单例优缺点

* 优点：

• 在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。提供了对唯一实例的受控访问。由于在系统内存中只存在一个对象，因此可以节约系统资源，当需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。允许可变数目的实例。避免对共享资源的多重占用。

* 缺点：

• 不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。单例类的职责过重，在一定程度上违背了“单一职责原则”。滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

### 4.单例模式使用注意事项：

* 使用时不能用反射模式创建单例，否则会实例化一个新的对象使用懒单例模式时注意线程安全问题饿单例模式和懒单例模式构造方法都是私有的，因而是不能被继承的，有些单例模式可以被继承（如登记式模式）

### 5.单例防止反射漏洞攻击

* private static boolean flag = false;private Singleton() {if (flag == false) {flag = !flag;} else {throw new RuntimeException("单例模式被侵犯！");}}public static void main(String[] args) {}

### 6.如何选择单例创建方式

* 如果不需要延迟加载单例，可以使用枚举或者饿汉式，相对来说枚举性好于饿汉式。 如果需要延迟加载，可以使用静态内部类或者懒汉式，相对来说静态内部类好于懒韩式。 最好使用饿汉式

### 7.单例创建方式

* （主要使用懒汉和懒汉式）

• 饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。懒汉式: 类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。枚举单例: 使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。双重检测锁方式 (因为JVM本质重排序的原因，可能会初始化多次，不推荐使用)

* 1.饿汉式

• 饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。

• package com.lijie;//饿汉式public class Demo1 { // 类初始化时,会立即加载该对象，线程安全,调用效率高 private static Demo1 demo1 = new Demo1(); private Demo1() { System.out.println("私有Demo1构造参数初始化"); } public static Demo1 getInstance() { return demo1; } public static void main(String[] args) { Demo1 s1 = Demo1.getInstance(); Demo1 s2 = Demo1.getInstance(); System.out.println(s1 == s2); }}

* 2.懒汉式

• 懒汉式: 类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。

• package com.lijie;//懒汉式public class Demo2 { //类初始化时，不会初始化该对象，真正需要使用的时候才会创建该对象。 private static Demo2 demo2; private Demo2() { System.out.println("私有Demo2构造参数初始化"); } public synchronized static Demo2 getInstance() { if (demo2 == null) { demo2 = new Demo2(); } return demo2; } public static void main(String[] args) { Demo2 s1 = Demo2.getInstance(); Demo2 s2 = Demo2.getInstance(); System.out.println(s1 == s2); }}

* 3.静态内部类

• 静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。

• package com.lijie;// 静态内部类方式public class Demo3 { private Demo3() { System.out.println("私有Demo3构造参数初始化"); } public static class SingletonClassInstance { private static final Demo3 DEMO\_3 = new Demo3(); } // 方法没有同步 public static Demo3 getInstance() { return SingletonClassInstance.DEMO\_3; } public static void main(String[] args) { Demo3 s1 = Demo3.getInstance(); Demo3 s2 = Demo3.getInstance(); System.out.println(s1 == s2); }}

* 4.枚举单例式

• 枚举单例: 使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。

• package com.lijie;//使用枚举实现单例模式 优点:实现简单、枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞 缺点没有延迟加载public class Demo4 { public static Demo4 getInstance() { return Demo.INSTANCE.getInstance(); } public static void main(String[] args) { Demo4 s1 = Demo4.getInstance(); Demo4 s2 = Demo4.getInstance(); System.out.println(s1 == s2); } //定义枚举private static enum Demo {INSTANCE;// 枚举元素为单例private Demo4 demo4;private Demo() {System.out.println("枚举Demo私有构造参数");demo4 = new Demo4();}public Demo4 getInstance() {return demo4;}}}

* 5.双重检测锁方式

• 双重检测锁方式 (因为JVM本质重排序的原因，可能会初始化多次，不推荐使用)

• package com.lijie;//双重检测锁方式public class Demo5 {private static Demo5 demo5;private Demo5() {System.out.println("私有Demo4构造参数初始化");}public static Demo5 getInstance() {if (demo5 == null) {synchronized (Demo5.class) {if (demo5 == null) {demo5 = new Demo5();}}}return demo5;}public static void main(String[] args) {Demo5 s1 = Demo5.getInstance();Demo5 s2 = Demo5.getInstance();System.out.println(s1 == s2);}}

## 工厂模式

### 1.什么是工厂模式

* 它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式

### 2.工厂模式好处

* 工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。利用工厂模式可以降低程序的耦合性，为后期的维护修改提供了很大的便利。将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

### 3.为什么要学习工厂设计模式

* 不知道你们面试题问到过源码没有，你知道Spring的源码吗，MyBatis的源码吗，等等等 如果你想学习很多框架的源码，或者你想自己开发自己的框架，就必须先掌握设计模式（工厂设计模式用的是非常非常广泛的）

### 4.Spring开发中的工厂设计模式

* 1.Spring IOC

• 看过Spring源码就知道，在Spring IOC容器创建bean的过程是使用了工厂设计模式Spring中无论是通过xml配置还是通过配置类还是注解进行创建bean，大部分都是通过简单工厂来进行创建的。当容器拿到了beanName和class类型后，动态的通过反射创建具体的某个对象，最后将创建的对象放到Map中

* 2.为什么Spring IOC要使用工厂设计模式创建Bean呢

• 在实际开发中，如果我们A对象调用B，B调用C，C调用D的话我们程序的耦合性就会变高。（耦合大致分为类与类之间的依赖，方法与方法之间的依赖。）在很久以前的三层架构编程时，都是控制层调用业务层，业务层调用数据访问层时，都是是直接new对象，耦合性大大提升，代码重复量很高，对象满天飞为了避免这种情况，Spring使用工厂模式编程，写一个工厂，由工厂创建Bean，以后我们如果要对象就直接管工厂要就可以，剩下的事情不归我们管了。Spring IOC容器的工厂中有个静态的Map集合，是为了让工厂符合单例设计模式，即每个对象只生产一次，生产出对象后就存入到Map集合中，保证了实例不会重复影响程序效率。

### 5.工厂模式分类

* 工厂模式分为简单工厂、工厂方法、抽象工厂模式

• 简单工厂 ：用来生产同一等级结构中的任意产品。（不支持拓展增加产品）工厂方法 ：用来生产同一等级结构中的固定产品。（支持拓展增加产品） 抽象工厂 ：用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）

* 5.1 简单工厂模式

• 什么是简单工厂模式

• 简单工厂模式相当于是一个工厂中有各种产品，创建在一个类中，客户无需知道具体产品的名称，只需要知道产品类所对应的参数即可。但是工厂的职责过重，而且当类型过多时不利于系统的扩展维护。

• 代码演示：

• 创建工厂

• package com.lijie;public interface Car {public void run();}

• 创建工厂的产品（宝马）

• package com.lijie;public class Bmw implements Car {public void run() {System.out.println("我是宝马汽车...");}}

• 创建工另外一种产品（奥迪）

• package com.lijie;public class AoDi implements Car {public void run() {System.out.println("我是奥迪汽车..");}}

• 创建核心工厂类，由他决定具体调用哪产品

• package com.lijie;public class CarFactory { public static Car createCar(String name) {if ("".equals(name)) { return null;}if(name.equals("奥迪")){return new AoDi();}if(name.equals("宝马")){return new Bmw();}return null;}}

• 演示创建工厂的具体实例

• package com.lijie;public class Client01 {public static void main(String[] args) {Car aodi =CarFactory.createCar("奥迪");Car bmw =CarFactory.createCar("宝马");aodi.run();bmw.run();}}

• 单工厂的优点/缺点

• 优点：简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。缺点：很明显工厂类集中了所有实例的创建逻辑，容易违反GRASPR的高内聚的责任分配原则

* 5.2 工厂方法模式

## 各种设计模式的介绍及实现

### 工厂模式

* 使用优点：

• 1、一个调用者想创建一个对象，只要知道其名称就可以了。2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。3、屏蔽产品的具体实现，调用者只关心产品的接口

* 使用缺点：

• 1、每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

* 使用场景：

• 1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。 2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。 3、设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

* 注意事项：

• 1、作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

* 实现DEMO：

• 新建接口

• public interface Color { void according();}

• 新建接口的实现类

• public class Blue implements Color { @Override public void according() { System.out.println("我是蓝色"); }}

• public class Red implements Color { @Override public void according() { System.out.println("我是红色"); }}

• 创建一个工厂，生成基于给定信息的实体类的对象

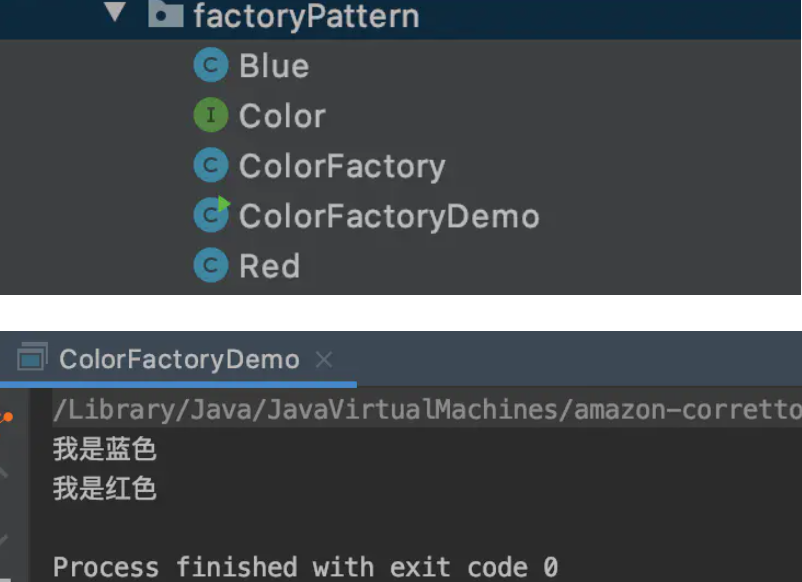
• public class ColorFactory { public Color getColor(String type){ if (StringUtils.isEmpty(type)){ return null; } if (type.equals("RED")){ return new Red(); } if (type.equals("BLUE")){ return new Blue(); } return null; }}

• 使用该工厂，通过传递类型信息来获取实体类的对象

• public class ColorFactoryDemo { public static void main(String[] args) { ColorFactory colorFactory = new ColorFactory(); // 获取蓝色类并调用方法 Color blue = colorFactory.getColor("BLUE"); blue.according(); // 获取红色类调用方法 Color red = colorFactory.getColor("RED"); red.according(); }}

• 查看结果

•



### 抽象工厂模式

* 使用优点：

• 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

* 使用缺点：

• 产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

* 使用场景：

• 1、QQ 换皮肤，一整套一起换。2、生成不同操作系统的程序。

* 注意事项：

• 1、产品族难扩展，产品等级易扩展。

* 实现DEMO：

• 新建形状接口

• public interface Shape { void draw();}

• 新建形状接口实现类

• public class Rectangle implements Shape { @Override public void draw() { System.out.println("Rectangle"); }}

• public class Square implements Shape { @Override public void draw() { System.out.println("Square"); }}

• 新建颜色接口

• public interface Color { void fill();}

• 新建颜色接口实现类

• public class Red implements Color { @Override public void fill() { System.out.println("Red"); }}

• public class Blue implements Color { @Override public void fill() { System.out.println("Blue"); }}

• 为 Color 和 Shape 对象创建抽象类来获取工厂

• public interface AbstractFactory { /\*\* \* 获取颜色 \* \* @param color \* @return \*/ public abstract Color getColor(String color); /\*\* \* 获取形状 \* \* @param shape \* @return \*/ public abstract Shape getShape(String shape);}

• 创建扩展了 AbstractFactory 的工厂类，基于给定的信息生成实体类的对象

• public class ShapeFactory extends AbstractFactory { @Override public Color getColor(String color) { return null; } @Override public Shape getShape(String shape) { if (StringUtils.isEmpty(shape)){ return null; } if (shape.equalsIgnoreCase("SQUARE")){ return new Square(); } if (shape.equalsIgnoreCase("RECTANGLE")){ return new Rectangle(); } return null; }}

• public class ColorFactory extends AbstractFactory { @Override public Color getColor(String color) { if (StringUtils.isEmpty(color)){ return null; } if (color.equalsIgnoreCase("RED")){ return new Red(); } if (color.equalsIgnoreCase("BLUE")){ return new Blue(); } return null; } @Override public Shape getShape(String shape) { return null; }}

• 创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂

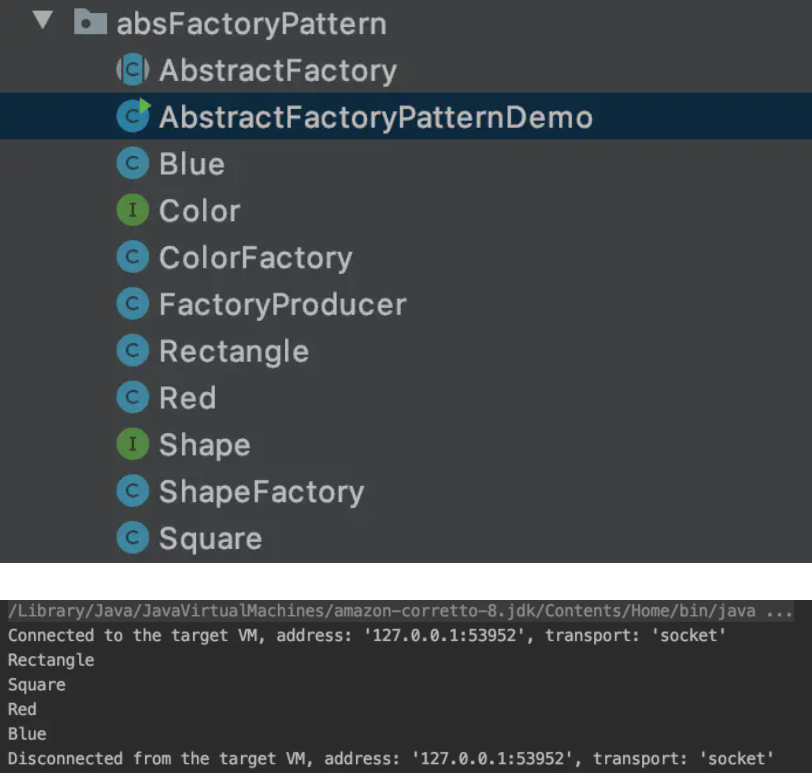
• public class FactoryProducer { public static AbstractFactory getFactory(String type){ if (StringUtils.isEmpty(type)){ return null; } if (type.equalsIgnoreCase("SHAPE")){ return new ShapeFactory(); }else if (type.equalsIgnoreCase("COLOR")){ return new ColorFactory(); } return null; }}

• 使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象

• public class AbstractFactoryPatternDemo { public static void main(String[] args){ // 获取形状工厂 AbstractFactory shape = FactoryProducer.getFactory("SHAPE"); // 获取颜色工厂 AbstractFactory color = FactoryProducer.getFactory("COLOR"); // 获取形状为 Rectangle 的对象 Shape rectangle = shape.getShape("RECTANGLE"); rectangle.draw(); // 获取形状为 Square 的对象 Shape square = shape.getShape("SQUARE"); square.draw(); //获取颜色为 Red 的对象 Color red = color.getColor("RED"); red.fill(); // 获取颜色为 Blue 的对象 Color blue = color.getColor("BLUE"); blue.fill(); }}

• 运行，查看结果

•



### 单例模式

* 使用优点：

• 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。2、避免对资源的多重占用（比如写文件操作）。

* 使用缺点：

• 1、没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

* 使用场景：

• 1、要求生产唯一序列号。2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

* 注意事项：

• 1、getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

* 实现DEMO：

• 新建 Singleton 类，提供获取对象的方法

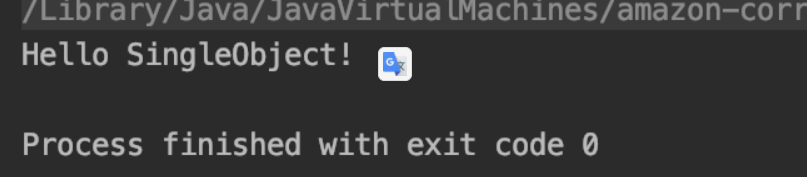
• public class SingleObject { // 创建 SingleObject 的一个对象 private static SingleObject singleObject = new SingleObject(); //让构造函数为 private，这样该类就不会被实例化 private SingleObject(){} //获取唯一可用的对象 public static SingleObject getInstance(){ return singleObject; } public void showMessage(){ System.out.println("Hello SingleObject!"); }}

• 从 singleton 类获取唯一的对象并调用方法

• public class SingleObjectDemo { public static void main(String[] args) { //不合法的构造函数 //编译时错误：构造函数 SingleObject() 是不可见的 //SingleObject object = new SingleObject(); //获取唯一可用的对象 SingleObject object = SingleObject.getInstance(); //显示消息 object.showMessage(); }}

• 查看结果

•



* 单例模式的几种实现方式

• 懒汉式，线程不安全

• public class LazyThreadNotSafety { private static LazyThreadNotSafety lazyThreadSafety; private LazyThreadNotSafety(){ } public static LazyThreadNotSafety getInstance(){ if (lazyThreadSafety == null){ lazyThreadSafety = new LazyThreadNotSafety(); } return lazyThreadSafety; }}

• 描述：

• 这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。 这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

• 懒汉式，线程安全

• public class LazyThreadSafety { private static LazyThreadSafety lazyThreadSafety; private LazyThreadSafety(){ } public static synchronized LazyThreadSafety getInstance(){ if (lazyThreadSafety == null){ lazyThreadSafety = new LazyThreadSafety(); } return lazyThreadSafety; }}

• 描述：

• 这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。

• 优点：

• 第一次调用才初始化，避免内存浪费。

• 缺点：

• 必须加锁 synchronized 才能保证单例，但加锁会影响效率。 getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

• 饿汉式

• public class HungryType { private static HungryType hungryType = new HungryType(); private HungryType(){ } public static HungryType getInstance(){ return hungryType; }}

• 描述：

• 这种方式比较常用，但容易产生垃圾对象。

• 优点：

• 没有加锁，执行效率会提高。

• 缺点：

• 类加载时就初始化，浪费内存。

• 双检锁/双重校验锁

• public class DoubleLock { private volatile static DoubleLock doubleLock; private DoubleLock(){ } public static DoubleLock getInstance(){ if (doubleLock == null){ synchronized (DoubleLock.class){ if (doubleLock == null){ doubleLock = new DoubleLock(); } } } return doubleLock; }}

• 描述：

• 这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

• 登记式/静态内部类

• public class Singleton { private static class SingletonHolder { private static final Singleton INSTANCE = new Singleton(); } private Singleton (){} public static final Singleton getInstance() { return SingletonHolder.INSTANCE; } }

• 描述：

• 这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

• 枚举

• public enum Singleton { INSTANCE; public void whateverMethod() { } }

• 描述：

• 这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。

* 经验之谈：一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

### 策略模式

* 使用优点：

• 1、算法可以自由切换。2、避免使用多重条件判断。3、扩展性良好。

* 使用缺点：

• 1、策略类会增多。2、所有策略类都需要对外暴露。

* 使用场景：

• 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

* 注意事项：

• 如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

* 实现DEMO：

• 新建策略类应该实现的接口

• public interface Strategy { public int doOperation(int num1, int num2);}

• 新建三个策略类

• @Servicepublic class StrategyAdd implements Strategy { @Override public int doOperation(int num1, int num2) { return num1 + num2; }}

• @Servicepublic class StrategyReduction implements Strategy { @Override public int doOperation(int num1, int num2) { return num1 - num2; }}

• @Servicepublic class StrategyTake implements Strategy { @Override public int doOperation(int num1, int num2) { return num1 \* num2; }}

• 创建 Context 类

• public class Context { private Strategy strategy; public Context(Strategy strategy){ this.strategy = strategy; } public int executeStrategy(int num1, int num2){ return strategy.doOperation(num1, num2); }}

• 新建策略枚举类

• public enum StrategyEnum { ADD("strategyAdd"), SUB("strategyReduction"), TAKE("strategyTake"); /\*\* 描述 \*/ private String clazz; private StrategyEnum(String clazz) { this.clazz = clazz; } public String getClazz() { return clazz; } public void setClazz(String clazz) { this.clazz = clazz; } public static Map&lt;String, String&gt; toMap() { Map&lt;String, String&gt; enumMap = new HashMap&lt;String, String&gt;(); StrategyEnum[] ary = StrategyEnum.values(); for (int i = 0, size = ary.length; i &lt; size; i++) { enumMap.put(ary[i].name(), ary[i].getClazz()); } return enumMap; }}

• 创建 SpringContextUtils 类

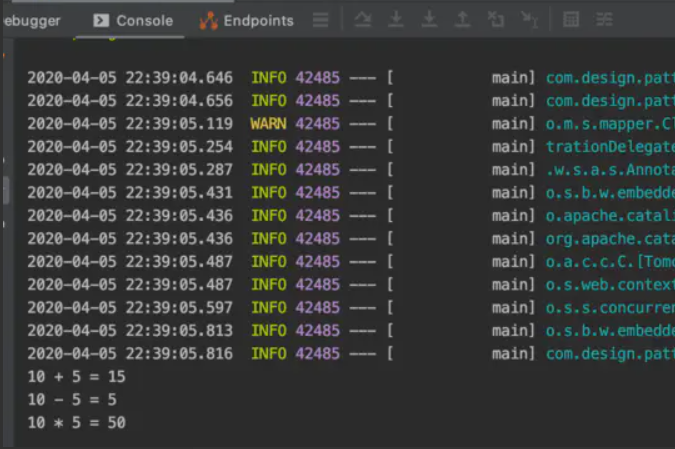
• @Componentpublic class SpringContextUtils implements ApplicationContextAware { public static ApplicationContext applicationContext; @Override public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { SpringContextUtils.applicationContext = applicationContext; } public static Object getBean(String name) { return applicationContext.getBean(name); } public static &lt;T&gt; T getBean(String name, Class&lt;T&gt; requiredType) { return applicationContext.getBean(name, requiredType); } public static boolean containsBean(String name) { return applicationContext.containsBean(name); } public static boolean isSingleton(String name) { return applicationContext.isSingleton(name); } public static Class&lt;? extends Object&gt; getType(String name) { return applicationContext.getType(name); }}

• 使用 StrategyDemo 实现 CommandLineRunner接口来实现启动之后调用run方法来查看当它改变策略 Strategy 时的行为变化。

• @Componentpublic class StrategyDemo implements CommandLineRunner { @Override public void run(String... args) throws Exception { // 获取枚举类中的相应处理类型 String clazzName = getClassName("ADD"); // 通过反射获取类 Object strategyAdd = SpringContextUtils.getBean("strategyAdd"); Strategy bean = SpringContextUtils.getBean(clazzName, Strategy.class); Context context = new Context(bean); System.out.println("10 + 5 = " + context.executeStrategy(10, 5)); // 获取枚举类中的相应处理类型 String clazzName1 = getClassName("SUB"); // 通过反射获取类 Strategy bean1 = SpringContextUtils.getBean(clazzName1, Strategy.class); context = new Context(bean1); System.out.println("10 - 5 = " + context.executeStrategy(10, 5)); // 获取枚举类中的相应处理类型 String clazzName2 = getClassName("TAKE"); // 通过反射获取类 Strategy bean2 = SpringContextUtils.getBean(clazzName2, Strategy.class); context = new Context(bean2); System.out.println("10 \* 5 = " + context.executeStrategy(10, 5)); } /\*\* \* 获取对应的策略类 \* \* @param type \* @return \*/ private static String getClassName(String type) { String clazzName = ""; Map&lt;String, String&gt; maps = StrategyEnum.toMap(); for (Map.Entry&lt;String, String&gt; entry : maps.entrySet()) { if (entry.getKey().equalsIgnoreCase(type)) { clazzName = entry.getValue(); break; } } return clazzName; }}

• 查看结果

•



• 策略模式有很重要的几点：

• 枚举类中各个策略类必须首字母要小些，否则SpringContextUtils.getBean是获取不到的。

• 策略类必须加@Service，必须注入到spring容器中，不然无法通过反射获取实例。

• SpringContextUtils类是通过反射获取相应实例的，而SpringContextUtils是实现了ApplicationContextAware接口，启动springboot项目之后，setApplicationContext方法来注入ApplicationContext的实例，所以这里不能用main方法直接启动，那样的话ApplicationContext的实例是无法注入的。

• StrategyDemo类必须加@Component注解，将此类实例化到spring容器中，要不然项目启动是无法调用run方法的！

## 简单工厂和抽象工厂有什么区别？

# Java Web

## jsp 和 servlet 有什么区别？

### jsp经编译后就变成了Servlet.（JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码，Web容器将JSP的代码编译成JVM能够识别的java类）

### jsp更擅长表现于页面显示，servlet更擅长于逻辑控制

### Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到。

### Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

## jsp 有哪些内置对象？作用分别是什么？

### JSP有9个内置对象：

* request：封装客户端的请求，其中包含来自GET或POST请求的参数；
* response：封装服务器对客户端的响应；
* pageContext：通过该对象可以获取其他对象；
* session：封装用户会话的对象；
* application：封装服务器运行环境的对象；
* out：输出服务器响应的输出流对象；
* config：Web应用的配置对象；
* page：JSP页面本身（相当于Java程序中的this）；
* exception：封装页面抛出异常的对象。

## 说一下 jsp 的 4 种作用域？

### JSP中的四种作用域包括page、request、session和application，具体来说：

* page代表与一个页面相关的对象和属性。
* request代表与Web客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个Web组件；需要在页面显示的临时数据可以置于此作用域。
* session代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的session中。
* application代表与整个Web应用程序相关的对象和属性，它实质上是跨越整个Web应用程序，包括多个页面、请求和会话的一个全局作用域。

## session 和 cookie 有什么区别？

### 由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session.典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个Session是保存在服务端的，有一个唯一标识。在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如Memcached之类的来放 Session。

### 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

### Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。所以，总结一下：Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

## 说一下 session 的工作原理？

### 其实session是一个存在服务器上的类似于一个散列表格的文件。里面存有我们需要的信息，在我们需要用的时候可以从里面取出来。类似于一个大号的map吧，里面的键存储的是用户的sessionid，用户向服务器发送请求的时候会带上这个sessionid。这时就可以从中取出对应的值了。

## 如果客户端禁止 cookie 能实现 session 还能用吗？

### Cookie与 Session，一般认为是两个独立的东西，Session采用的是在服务器端保持状态的方案，而Cookie采用的是在客户端保持状态的方案。但为什么禁用Cookie就不能得到Session呢？因为Session是用Session ID来确定当前对话所对应的服务器Session，而Session ID是通过Cookie来传递的，禁用Cookie相当于失去了Session ID，也就得不到Session了。

### 假定用户关闭Cookie的情况下使用Session，其实现途径有以下几种：

* 设置php.ini配置文件中的“session.use\_trans\_sid = 1”，或者编译时打开打开了“--enable-trans-sid”选项，让PHP自动跨页传递Session ID。
* 手动通过URL传值、隐藏表单传递Session ID。
* 用文件、数据库等形式保存Session ID，在跨页过程中手动调用。

## spring mvc 和 struts 的区别是什么？

### 拦截机制的不同

* Struts2是类级别的拦截，每次请求就会创建一个Action，和Spring整合时Struts2的ActionBean注入作用域是原型模式prototype，然后通过setter，getter吧request数据注入到属性。Struts2中，一个Action对应一个request，response上下文，在接收参数时，可以通过属性接收，这说明属性参数是让多个方法共享的。Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了，只能设计为多例。
* SpringMVC是方法级别的拦截，一个方法对应一个Request上下文，所以方法直接基本上是独立的，独享request，response数据。而每个方法同时又何一个url对应，参数的传递是直接注入到方法中的，是方法所独有的。处理结果通过ModeMap返回给框架。在Spring整合时，SpringMVC的Controller Bean默认单例模式Singleton，所以默认对所有的请求，只会创建一个Controller，有应为没有共享的属性，所以是线程安全的，如果要改变默认的作用域，需要添加@Scope注解修改。
* Struts2有自己的拦截Interceptor机制，SpringMVC这是用的是独立的Aop方式，这样导致Struts2的配置文件量还是比SpringMVC大。

### 底层框架的不同

* Struts2采用Filter（StrutsPrepareAndExecuteFilter）实现，SpringMVC（DispatcherServlet）则采用Servlet实现。Filter在容器启动之后即初始化；服务停止以后坠毁，晚于Servlet。Servlet在是在调用时初始化，先于Filter调用，服务停止后销毁。

### 性能方面

* Struts2是类级别的拦截，每次请求对应实例一个新的Action，需要加载所有的属性值注入，SpringMVC实现了零配置，由于SpringMVC基于方法的拦截，有加载一次单例模式bean注入。所以，SpringMVC开发效率和性能高于Struts2。

### 配置方面

* spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高。

## 如何避免 sql 注入？

### PreparedStatement（简单又有效的方法）使用正则表达式过滤传入的参数字符串过滤JSP中调用该函数检查是否包函非法字符JSP页面判断代码

## 什么是 XSS 攻击，如何避免？

### XSS攻击又称CSS,全称Cross Site Script （跨站脚本攻击），其原理是攻击者向有XSS漏洞的网站中输入恶意的 HTML 代码，当用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。XSS 攻击类似于 SQL 注入攻击，SQL注入攻击中以SQL语句作为用户输入，从而达到查询/修改/删除数据的目的，而在xss攻击中，通过插入恶意脚本，实现对用户游览器的控制，获取用户的一些信息。 XSS是 Web 程序中常见的漏洞，XSS 属于被动式且用于客户端的攻击方式

### XSS防范的总体思路是：对输入(和URL参数)进行过滤，对输出进行编码。

## 什么是 CSRF 攻击，如何避免？

### CSRF（Cross-site request forgery）也被称为 one-click attack或者 session riding，中文全称是叫跨站请求伪造。一般来说，攻击者通过伪造用户的浏览器的请求，向访问一个用户自己曾经认证访问过的网站发送出去，使目标网站接收并误以为是用户的真实操作而去执行命令。常用于盗取账号、转账、发送虚假消息等。攻击者利用网站对请求的验证漏洞而实现这样的攻击行为，网站能够确认请求来源于用户的浏览器，却不能验证请求是否源于用户的真实意愿下的操作行为。

### 如何避免：

* 1. 验证 HTTP Referer 字段

• HTTP头中的Referer字段记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网站，而如果黑客要对其实施 CSRF攻击，他一般只能在他自己的网站构造请求。因此，可以通过验证Referer值来防御CSRF 攻击。

* 2. 使用验证码

• 关键操作页面加上验证码，后台收到请求后通过判断验证码可以防御CSRF。但这种方法对用户不太友好。

* 3. 在请求地址中添加token并验证

• CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于cookie中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有token或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于session之中，然后在每次请求时把token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成 http://url?csrftoken=tokenvalue。而对于 POST 请求来说，要在 form 的最后加上 &lt;input type="hidden" name="csrftoken" value="tokenvalue"/&gt;，这样就把token以参数的形式加入请求了。

* 4. 在HTTP 头中自定义属性并验证

• 这种方法也是使用 token 并进行验证，和上一种方法不同的是，这里并不是把 token 以参数的形式置于 HTTP 请求之中，而是把它放到 HTTP 头中自定义的属性里。通过 XMLHttpRequest 这个类，可以一次性给所有该类请求加上 csrftoken 这个 HTTP 头属性，并把 token 值放入其中。这样解决了上种方法在请求中加入 token 的不便，同时，通过 XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去。

# 缓存相关

## 为什么要用缓存？

### 聊缓存之前我们先聊聊数据库

* 在增删改查中，数据库查询占据了数据库操作的80%以上，非常频繁的磁盘I/O读取操作，会导致数据库性能极度低下。而数据库的重要性就不言而喻了：数据库通常是企业应用系统最核心的部分数据库保存的数据量通常非常庞大数据库查询操作通常很频繁，有时还很复杂我们知道，对于多数Web应用，整个系统的瓶颈在于数据库。原因很简单，Web应用中的其他因素，例如网络带宽、负载均衡节点、应用服务器（包括CPU、内存、硬盘灯、连接数等）、缓存，都很容易通过水平的扩展（俗称加机器）来实现性能的提高。而对于MySQL，由于数据一致性的要求，无法通过简单的增加机器来分散向数据库 写数据 带来的压力。虽然可以通过前置缓存（Redis等）、读写分离、分库分表来减轻压力，但是与系统其它组件的水平扩展相比，受到了太多的限制，而切会大大增加系统的复杂性。因此数据库的连接和读写要十分珍惜。可能你会想到那就直接用缓存呗，但大量的用、不分场景的用缓存显然是不科学的。我们不能手里有了一把锤子，看什么都是钉子。但缓存也不是万能的，要慎用缓存，想要用好缓存并不容易。因此我花了点时间整理了一下关于缓存的实现以及常见的一些问题

### 为什么

* 系统访问特点：二八定律即80%的业务访问集中在20%的数据上

### 总结

* 减少往数据库跑，从而提高性能

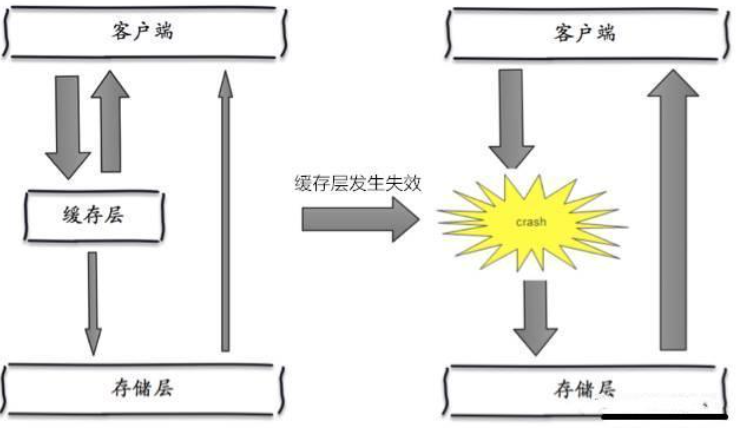
## 什么是缓存雪崩？如何解决？

### 什么是缓存雪崩？

* 概念

• 缓存雪崩是指，缓存层出现了错误，不能正常工作了。于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。

• 图解



* 产生原因

• a. 由于Cache层承载着大量请求，有效的保护了Storage层(通常认为此层抗压能力稍弱)，所以Storage的调用量实际很低，所以它很爽。b. 但是，如果Cache层由于某些原因(宕机、cache服务挂了或者不响应了)整体crash掉了，也就意味着所有的请求都会达到Storage层，所有Storage的调用量会暴增，所以它有点扛不住了，甚至也会挂掉

• 我们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。

* 解决方案

• 加锁/队列 保证缓存单线程的写

• 失效时的雪崩效应对底层系统的冲击非常可怕。

• 大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

• 加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。

• 假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

• 加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

• 避免缓存同时失效

• 将缓存失效时间分散开，比如我们可以在原有的失效时间基础上，末尾增加一个随机值。

• 缓存降级

• 当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。

• 系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

• 降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

• 在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级。

• 比如可以参考日志级别设置预案：

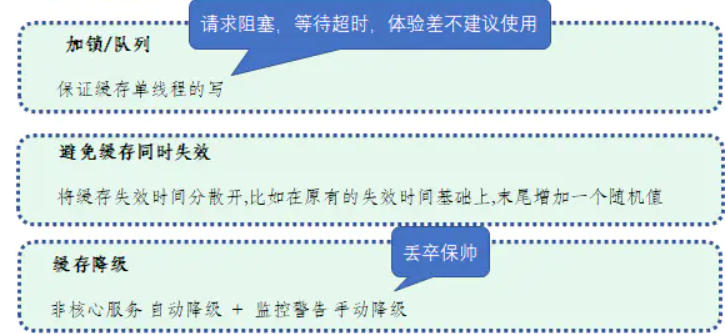
• 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

• 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

• 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

• 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

• 图示



### 如何解决？

* 解决方案

• 加锁/队列 保证缓存单线程的写

• 失效时的雪崩效应对底层系统的冲击非常可怕。

• 大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

• 加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。

• 假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

• 加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

• 避免缓存同时失效

• 将缓存失效时间分散开，比如我们可以在原有的失效时间基础上，末尾增加一个随机值。

• 缓存降级

• 当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。

• 系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

• 降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

• 在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级。

• 比如可以参考日志级别设置预案：

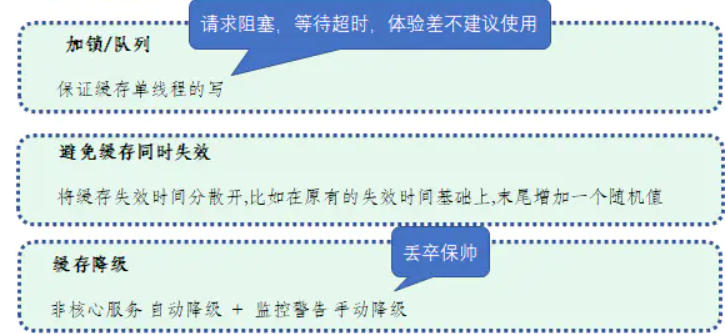
• 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

• 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

• 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

• 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

• 图示



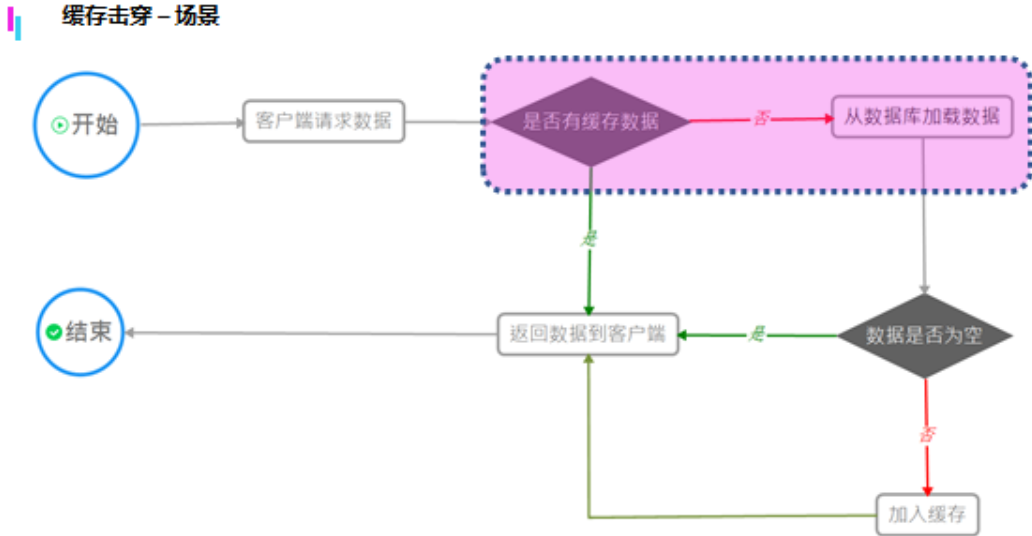
## 什么是缓存穿透？如何解决？

### 什么是缓存穿透？

* 什么是缓存击穿/缓存穿透

• 缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，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存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

• 图解



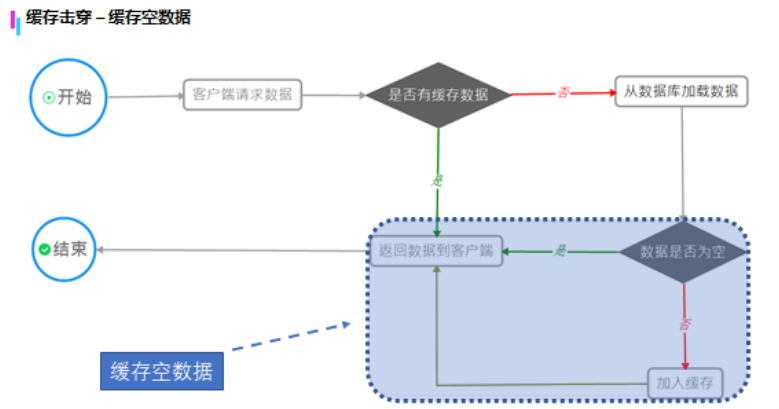
### 如何解决？

* 解决方案

• 解决方案1

• 一个简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

•

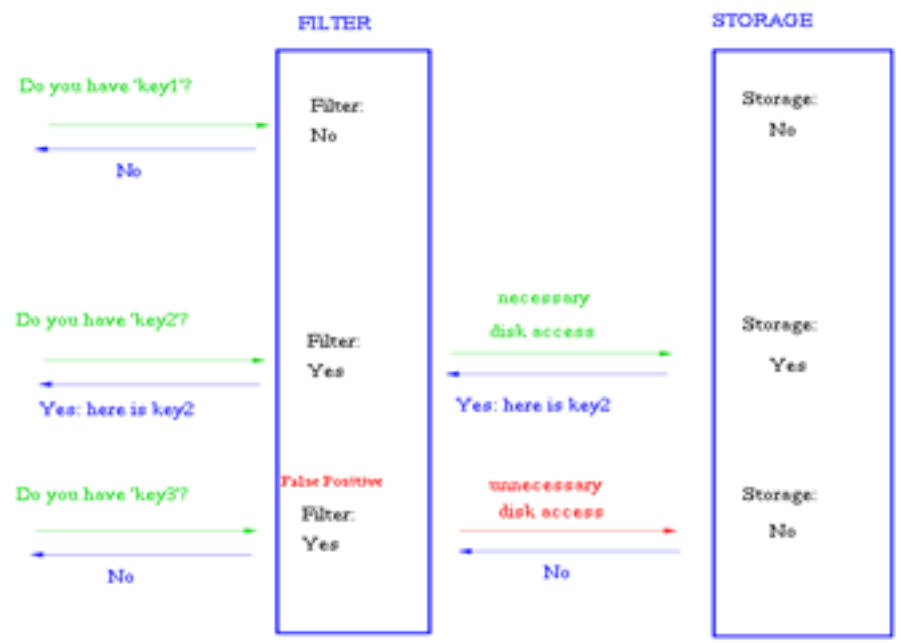


• 解决方案2

• 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。

• 例如，商城有100万用户数据，将所有用户id刷入一个Map。当请求过来以后，先判断Map中是否包含该用户id，不包含直接返回，包含的话先去缓存中查是否有这条数据，有的话返回，没有的话再去查数据库。这样不仅减轻了数据库的压力，缓存系统的压力也将大大降低。

•



## redis和memcached的区别？

### redis和Memcached相同点

* redis是一个key-value存储系统，这点和Memcached类似
* 和Memcached一样，为了保证效率，数据都是缓存在内存中。

### 和Memcached不同点

* 不同的是它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集。
* 区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。

## 都了解哪些redis中的数据结构？分别是怎么用的？

## 你现在用的redis是单线程的还是多线程的？

## 单线程的redis为什么还可以这么快？

## 缓存预热是怎么做的？举几个需要缓存预热的场景？

## 缓存与数据库数据不一致时该怎么办？

## 举几个可能会导致缓存与DB数据不一致的场景？

## redis中的数据过期策略了解吗？

## redis的持久化了解吗？

## redis用的是主从架构吗？数据是怎么复制的有了解过吗?

## redis的哨兵机制了解吗？

## 讲一讲LRU算法的原理？

## 有用过本地缓存吗？

# 异常

## throw 和 throws 的区别？

### throws是用来声明一个方法可能抛出的所有异常信息，throws是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。而throw则是指抛出的一个具体的异常类型。

## Error 和 Exception 有什么区别？

## final、finally、finalize 这几个关键字有什么区别？

### final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

### finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

### finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System的gc()方法的时候，由垃圾回收器调用finalize(),回收垃圾。

## try-catch-finally 中哪个部分可以省略？

### catch 可以省略

### 原因：

* 更为严格的说法其实是：try只适合处理运行时异常，try+catch适合处理运行时异常+普通异常。也就是说，如果你只用try去处理普通异常却不加以catch处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用catch显示声明以便进一步处理。而运行时异常在编译时没有如此规定，所以catch可以省略，你加上catch编译器也觉得无可厚非。
* 理论上，编译器看任何代码都不顺眼，都觉得可能有潜在的问题，所以你即使对所有代码加上try，代码在运行期时也只不过是在正常运行的基础上加一层皮。但是你一旦对一段代码加上try，就等于显示地承诺编译器，对这段代码可能抛出的异常进行捕获而非向上抛出处理。如果是普通异常，编译器要求必须用catch捕获以便进一步处理；如果运行时异常，捕获然后丢弃并且+finally扫尾处理，或者加上catch捕获以便进一步处理。
* 至于加上finally，则是在不管有没捕获异常，都要进行的“扫尾”处理。

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

### 会执行，在 return 前执行。

### 代码示例

* 代码示例1：

• /\* \* java面试题--如果catch里面有return语句，finally里面的代码还会执行吗？ \*/public class FinallyDemo2 { public static void main(String[] args) { System.out.println(getInt()); } public static int getInt() { int a = 10; try { System.out.println(a / 0); a = 20; } catch (ArithmeticException e) { a = 30; return a; /\* \* return a 在程序执行到这一步的时候，这里不是return a 而是 return 30；这个返回路径就形成了 \* 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40 \* 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量了，而是常量30 \*/ } finally { a = 40; }// return a; }}

• 执行结果：30

* 代码示例2：

• package com.java\_02;/\* \* java面试题--如果catch里面有return语句，finally里面的代码还会执行吗？ \*/public class FinallyDemo2 { public static void main(String[] args) { System.out.println(getInt()); } public static int getInt() { int a = 10; try { System.out.println(a / 0); a = 20; } catch (ArithmeticException e) { a = 30; return a; /\* \* return a 在程序执行到这一步的时候，这里不是return a 而是 return 30；这个返回路径就形成了 \* 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40 \* 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量了，而是常量30 \*/ } finally { a = 40; return a; //如果这样，就又重新形成了一条返回路径，由于只能通过1个return返回，所以这里直接返回40 }// return a; }}

• 执行结果：40

## 说几个你熟悉的异常类吧？

### NullPointerException：当应用程序试图访问空对象时，则抛出该异常。

### SQLException：提供关于数据库访问错误或其他错误信息的异常。

### IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

### NumberFormatException：当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。

### FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。

### IOException：当发生某种I/O异常时，抛出此异常。此类是失败或中断的I/O操作生成的异常的通用类。

### ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。

### ArrayStoreException：试图将错误类型的对象存储到一个对象数组时抛出的异常。

### IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。

### ArithmeticException：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

### NegativeArraySizeException：如果应用程序试图创建大小为负的数组，则抛出该异常。

### NoSuchMethodException：无法找到某一特定方法时，抛出该异常。

### SecurityException：由安全管理器抛出的异常，指示存在安全侵犯。

### UnsupportedOperationException：当不支持请求的操作时，抛出该异常。

### RuntimeExceptionRuntimeException：是那些可能在Java虚拟机正常运行期间抛出的异常的超类

## 异常处理原则

### 1. 能处理的异常尽早处理

### 2. 具体明确原则

### 3. 系统需要有统一异常处理机制

### 4. 一些其他注意点

* try语句块内要分清稳定代码和非稳定代码，对于稳定的不会出现异常的代码不要放到try语句块中；catch捕获的异常一定要处理，吃掉异常不处理的话将是灭顶之灾；finally中不要使用return语句,因为finally语句块最后一定会执行,这里的return语句会覆盖之前的return语句。

# java算法

## 二分查找

### 又叫折半查找，要求待查找的序列有序。每次取中间位置的值与待查关键字比较，如果中间位置的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

# JVM

## 线程

### 基本概念：

* JVM 是可运行 Java 代码的假想计算机 ，包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收，堆 和 一个存储方法域。JVM 是运行在操作系统之上的，它与硬件没有直接的交互。

### 运行过程：

## JVM内存区域

## JVM运行时内存

## 垃圾回收与算法

## JAVA四种引用类型

## GC分代收集算法VS分区收集算法

## GC垃圾收集器

## JavaIo和NIO

## JVM类加载机制

# 对象拷贝

## 为什么要使用克隆？

### 想对一个对象进行处理，又想保留原有的数据进行接下来的操作，就需要克隆了，Java语言中克隆针对的是类的实例。

## 如何实现对象克隆？

### 有两种方式：

* 1). 实现Cloneable接口并重写Object类中的clone()方法；
* 2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，

• 代码如下：

• import java.io.ByteArrayInputStream;import java.io.ByteArrayOutputStream;import java.io.ObjectInputStream;import java.io.ObjectOutputStream;import java.io.Serializable; public class MyUtil { private MyUtil() { throw new AssertionError(); } @SuppressWarnings("unchecked") public static &lt;T extends Serializable&gt; T clone(T obj) throws Exception { ByteArrayOutputStream bout = new ByteArrayOutputStream(); ObjectOutputStream oos = new ObjectOutputStream(bout); oos.writeObject(obj); ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray()); ObjectInputStream ois = new ObjectInputStream(bin); return (T) ois.readObject(); // 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义 // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放 }}

• 下面是测试代码：

• import java.io.Serializable; /\*\* \* 人类 \* @author nnngu \* \*/class Person implements Serializable { private static final long serialVersionUID = -9102017020286042305L; private String name; // 姓名 private int age; // 年龄 private Car car; // 座驾 public Person(String name, int age, Car car) { this.name = name; this.age = age; this.car = car; } public String getName() { return name; } public void setName(String name) { this.name = name; } public int getAge() { return age; } public void setAge(int age) { this.age = age; } public Car getCar() { return car; } public void setCar(Car car) { this.car = car; } @Override public String toString() { return "Person [name=" + name + ", age=" + age + ", car=" + car + "]"; } }/\*\* \* 小汽车类 \* @author nnngu \* \*/class Car implements Serializable { private static final long serialVersionUID = -5713945027627603702L; private String brand; // 品牌 private int maxSpeed; // 最高时速 public Car(String brand, int maxSpeed) { this.brand = brand; this.maxSpeed = maxSpeed; } public String getBrand() { return brand; } public void setBrand(String brand) { this.brand = brand; } public int getMaxSpeed() { return maxSpeed; } public void setMaxSpeed(int maxSpeed) { this.maxSpeed = maxSpeed; } @Override public String toString() { return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]"; } }

• class CloneTest { public static void main(String[] args) { try { Person p1 = new Person("郭靖", 33, new Car("Benz", 300)); Person p2 = MyUtil.clone(p1); // 深度克隆 p2.getCar().setBrand("BYD"); // 修改克隆的Person对象p2关联的汽车对象的品牌属性 // 原来的Person对象p1关联的汽车不会受到任何影响 // 因为在克隆Person对象时其关联的汽车对象也被克隆了 System.out.println(p1); } catch (Exception e) { e.printStackTrace(); } }}

* 注意

• 基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

## 深拷贝和浅拷贝区别是什么？

### 浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝（例：assign()）

### 深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse()和JSON.stringify()，但是此方法无法复制函数类型）

# 消息队列相关

## 应用场景

### 解耦

### 异步

### 流量消峰

### 日志记录

## 为什么要用消息队列？

## 消息队列有什么缺点？

## 项目中用到了那种消息队列，简单说一下使用的场景？

## 比较一下你熟悉的几种消息队列？

## 如何保证消息队列的高可用？

## 如何保证消息不被重复消费

## 如何处理消息丢失的问题？

## 如何保证消息的顺序性？

## 如何应对消息大量积压？

# 微服务

# 日志

# Netty 与 RPC

# Zookeeper

# Nginx

## 什么是Nginx？

### Nginx是一个 轻量级/高性能的反向代理Web服务器，他实现非常高效的反向代理、负载平衡，他可以处理2-3万并发连接数，官方监测能支持5万并发，现在中国使用nginx网站用户有很多，例如：新浪、网易、 腾讯等。

## 为什么要用Nginx？

### 跨平台、配置简单、方向代理、高并发连接：处理2-3万并发连接数，官方监测能支持5万并发，内存消耗小：开启10个nginx才占150M内存 ，nginx处理静态文件好，耗费内存少，而且Nginx内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。使用Nginx的话还能：节省宽带：支持GZIP压缩，可以添加浏览器本地缓存稳定性高：宕机的概率非常小接收用户请求是异步的

## 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

## 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区块结果 }

## 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

## 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。

## 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

## 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，需要用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。nginz 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。

## 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 │ ├── koi-utf│ ├── koi-win│ ├── mime.types # 媒体类型│ ├── mime.types.default│ ├── nginx.conf # Nginx主配置文件│ ├── nginx.conf.default│ ├── scgi\_params # scgi相关参数文件│ ├── scgi\_params.default │ ├── uwsgi\_params # uwsgi相关参数文件│ ├── uwsgi\_params.default│ └── 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

## 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} # 事件区块结束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区块结果 error\_page 500502503504 /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 } } ......

## 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

## 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。

## 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; } }

## 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

### 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; }

## 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;}}复制代码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;}}复制代码为什么就多了一个 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; } }复制代码上面配置了单个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。刚才有提到过Nginx是基于漏桶算法原理实现的，实际上限流一般都是基于漏桶算法和令牌桶算法实现的。接下来我们来看看两个算法的介绍：

## 漏桶流算法和令牌桶算法知道？

### 漏桶算法

* 漏桶算法是网络世界中流量整形或速率限制时经常使用的一种算法，它的主要目的是控制数据注入到网络的速率，平滑网络上的突发流量。漏桶算法提供了一种机制，通过它，突发流量可以被整形以便为网络提供一个稳定的流量。也就是我们刚才所讲的情况。漏桶算法提供的机制实际上就是刚才的案例：突发流量会进入到一个漏桶，漏桶会按照我们定义的速率依次处理请求，如果水流过大也就是突发流量过大就会直接溢出，则多余的请求会被拒绝。所以漏桶算法能控制数据的传输速率。

### 令牌桶算法

* 令牌桶算法是网络流量整形和速率限制中最常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送。Google开源项目Guava中的RateLimiter使用的就是令牌桶控制算法。令牌桶算法的机制如下：存在一个大小固定的令牌桶，会以恒定的速率源源不断产生令牌。如果令牌消耗速率小于生产令牌的速度，令牌就会一直产生直至装满整个令牌桶。

## 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。

## 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; }复制代码创建目录mkdir /usr/local/static/image复制代码进入目录cd /usr/local/static/image复制代码放一张照片上去#1.jpg复制代码重启 nginxsudo nginx -s reload复制代码打开浏览器 输入 server\_name/image/1.jpg 就可以访问该静态图片了

## 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 weightweight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，达到合理有效的地利用主机资源。upstream backserver { server 192.168.0.12 weight=2; server 192.168.0.13 weight=8; } 复制代码权重越高，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; } 复制代码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; }

## 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; } }

## Nginx怎么判断别IP不可访问？

### # 如果访问的ip地址为192.168.9.115,则返回403if ($remote\_addr = 192.168.9.115) { return 403; }

## Rewrite全局变量是什么？

### 变量含义$args这个变量等于请求行中的参数，同$query\_string$content length请求头中的Content-length字段。$content\_type请求头中的Content-Type字段。$document\_root当前请求在root指令中指定的值。$host请求主机头字段，否则为服务器名称。$http\_user\_agent客户端agent信息$http\_cookie客户端cookie信息$limit\_rate这个变量可以限制连接速率。$request\_method客户端请求的动作，通常为GET或POST。$remote\_addr客户端的IP地址。$remote\_port客户端的端口。$remote\_user已经经过Auth Basic Module验证的用户名。$request\_filename当前请求的文件路径，由root或alias指令与URI请求生成。$schemeHTTP方法（如http，https）。$server\_protocol请求使用的协议，通常是HTTP/1.0或HTTP/1.1。$server\_addr服务器地址，在完成一次系统调用后可以确定这个值。$server\_name服务器名称。$server\_port请求到达服务器的端口号。$request\_uri包含请求参数的原始URI，不包含主机名，如”/foo/bar.php?arg=baz”。$uri不带请求参数的当前URI，$uri不包含主机名，如”/foo/bar.html”。$document\_uri与$uri相同。

# Kafka

# RabbitMQ

# Hbase

# MongoDB

# Cassandra

# 负载均衡

# 一致性算法

# 加密算法

# 分布式缓存

# Hadoop

# Spark

# Storm

# YARN

# 机器学习

# 云计算

# 其它

## 分布式相关概念

## ORM框架

## 日志

## 单测

## 设计模式

## 规则引擎

## 其它常见的中间件如：Dubbo、Netty、Zookeeper、ES、Kafka等

## 大数据相关，如：Spark、Flink、Storm、HDFS、Hive等

## 其它编程语言基础，如：Python、Go、JS等

## 常用工具如：Git、Maven、Gradle、Linux常用命令、Docker、UML等