Java常见面试题

JavaSE

基础部分

String、StringBuffer、StringBuilder的区别

String是使用final修饰的，所以在定义后不可以修改，每次修改都是重新创建一个String对象。

StringBuffer是一个可变的字符串，并且线程安全，但是效率过低。

StringBuilder和StringBuffer相同，但是不是线程安全的，相对效率较高。

如果字符串的大小确定，尽量在创建字符串的同时指定大小。

final关键字怎么用的

final修饰一个值时，这个值不可以变。

final修饰一个引用对象时，这个引用地址不可以变。

final修饰一个方法时，这个方法不可以被继承修改。

final修饰一个类时，这个类不能被继承。并且final类中的成员变量默认也是final修饰的

抽象类和接口的区别

语法：

- 抽象类：可以有抽象方法，也可以有非抽象的，可以有构造器
- 接口：方法都是抽象的（JDK1.8后可以有具体实现），属性是常量，默认有 public static final 修饰

设计：

- 抽象类：对同一事物的抽取，比如BaseDao
- 接口：通常是一种标准的制定。在调用接口时，如果实现类发生变化，不会影响到接口的调用，解耦。

集合

List和Set的区别

- List：
    - 特点：有序（存储和读取的顺序）、可重复、可以通过索引操作元素
    - 分类：
        - 底层是数组，查询快，增删慢
            - ArrayList：线程不安全，效率高

        - Vector：线程安全，效率低
    - 底层是链表：查询慢，增删快
        - LinkedList：线程不安全，效率高
- Set：
    - 特点：无序、元素唯一
    - 分类：
        - 底层是Hash表：
            - HashSet：保证元素唯一‘，hashCode()和equals()方法
            - ThreeSet：保证元素排序，自然排序。（让对象所属类去实现comparable接口，无参构造）/（比较器接口compartor，带参构造）

HashMap和Hashtable的区别

1. HashMap继承于AbstractMap，Hashtable继承于Dictionary，两者都实现了Map接口
2. HashMap运行Key和Value为null，Hashtable不可以
3. HashMap是线程不安全的，Hashtable是线程安全的，锁住整个对象。

谈谈你对HashMap的理解

HashMap的初始值是16，加载因子是0.75，并且要求该容量必须是2的整数次幂。

在1.7中HashMap的数据结构是数组+链表，扩容采用的插头法，可能造成闭环，线程不安全。

在1.8中HashMap的数据结构是数组+链表+红黑树，在链表的长度大于8时，回自动转为红黑树（如果当前容量小于64，会先扩容）。当红黑树元素少于6个时，会再次从红黑树转为链表。在多线程put操作下，会出现数据被覆盖的情况，同样的线程不安全。

> 插入

每一个元素都是一个node，node又包含hash、key、value、next等属性，其中元素的索引计算是由数组长度-1和hash做&运算得出的。如果当前索引元素为空，就会新创建一个node节点保存数据，如果不为空，就判断元素的key是否相同，如果相同就用新的元素替换旧的元素，如果元素key不相同，就会接着判断当前节点是否为树化后的节点，如果已经树化了，就尝试用树的结构存储元素，如果不是树化的节点，就会往链表中插入数据，同时判断链表的总数是否已经大于等于8，如果大于等于8就转换为红黑树。

> 获取

通过key的hash值计算出索引位置，并且对key做equals比较，true就返回值。

谈谈ConccurentHashMap

JUC包下的线程安全的HashMap，不允许插入null键

早期的ConccurentHashMap使用的是分段锁

当前ConccurentHashMap使用的是CAS+synchronized使锁更细化。

> 插入

通过Hash值计算元素数组中的索引，如果没有节点就使用CAS进行添加（链表的头节点），添加失败就进入下一次循环。如果内部正在扩容就帮助一起扩容。如果当前所有存在节点，就使用synchronized锁住链表或红黑树的头节点，然后进行添加操作

谈谈HashSet

HashSet底层使用依然是HashMap来实现存储，值是Map的Key，value是一个固定值。

为什么采用Hash算法?

JDK1.7版本

在不使用Hash算法时，存放数据的时候需要判断唯一性，就需要进行遍历然后一个一个比较，性能太低。使用hash算法，通过计算存储对象的hashCode，再和数组长度-1做位运算，得到要存储位置的下标。但是依旧会出现一个问题，元素过多之后，可能会出现hash碰撞的问题（不同的对象计算出的hash值相同），这个时候，HashMap就会使用equals()方法进行判断，如果值相同就不存放，如果不相同，就会转换成一个链表，新进来的元素next指向上一个元素。

JDK1.8版本优化

随着元素的增加，链表会越来越长，链表的查找效率过低，所以优化为一个红黑树。

ArrayList和LinkList的区别

ArrayList底层数据结构时数组，LinkList底层数据结构是双向链表。

因为ArrayList是数组，所以在获取元素时直接返回数组下标对应的元素即可，但是数组的空间都是连续的，如果修改数据就会牵扯到位置的挪动，因此ArrayList的修改效率低。

LinkList的底层数据结构是双向链表，链表没有下标，所以查询的时候会从第一个元素开始查询，效率过低，但是链表在修改元素时只需要移动链表对应的指针就可以，效率相对高一些。

ArrayList的扩容

因为底层是数组，所以容量是固定的，默认容量是10，当传入数据时会判断当前容量是否需要扩容，创建一个新数组，容量时旧数组的1.5倍（右移 >1），将原数组的数据迁移到新数组。

手写一个线程不安全的ArrayList例子

```
public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        for (int i = 0; i < 30; i++) {
            new Thread(() -> {
                list.add(UUID.randomUUID().toString().substring(0, 8));
                System.out.println(list);
            }).start();
        }
    }
```

会出现并发修改异常（ConcurrentModificationException）

怎么解决ArrayList的线程不安全?

- 使用线程安全的`Vector`，虽然保证的线程安全，但是性能下降了。
- 使用集合工具类`Collections`下的安全同步集合
- 写时复制，JUC包中的CopyOnWriteArrayList（写时复制）类，读写分离思想

谈谈CopyOnWrite.....

CopyOnWrite容器即为写时复制的容器，往一个容器添加元素时，不直接往当前容器 Object[] 中添加，而是先将当前容器 Object[] 复制出一个新的容器，然后在新的容器中添加元素，添加完后，再将原容器的引用指向新容器，好处时进行并发读时不需要加锁，因为当前容器不会添加任何元素，所以CopyOnWrite也是一种读写分离的思想，读和写在不同的容器。

异常

请描述Java的异常体系

异常类型回答了什么被抛出，异常堆栈跟踪回答了在哪抛出，异常信息回答了为什么被抛出。

- Throwable
    - Error：JVM错误，比如内存溢出
    - Exception：异常
        - 运行时异常：不会提醒进行异常捕获，通常是逻辑异常，代码不严谨，比如数组越界、空指针异常。
            - 算术异常
            - 空指针
            - 类型转换
            - 数组越界
        - 非运行时异常：会提醒进行捕获或抛出，异常的产生通常是第三方，系统会提醒我们对此处进行处理
            - IOException
            - SQLException
            - FileNotFountException
            - NoSuchFileException
            - NoSuchMethodException

Java异常的处理原则

- 具体明确：抛出的异常应能通过异常类名和massage准确说明异常的类型和产生的原因
- 提早抛出：应尽可能早的发现并抛出异常，便于精确定位问题
- 延迟捕获：异常的捕获和处理应尽可能延迟，让掌握更多的作用域来处理异常。

Throw和Throws的区别

- Throw：作用于方法上，用于主动抛出异常
- Throws：作用方法声明上，声明该方法可能会抛出的异常

IO流

IO流的分类

按输入方向：输入流、输出流

按读取单位：字节流、字符流

IO流的四大基类：

- 字节流：
  - 输入：InputStream
    - FileInputStream：读取文件
  - 输出：OutStream
    - FileOutStream：写入文件
- 字符流：
  - 输入：Read
  - 输出：Write

IO流的选择

字节流主要用于二进制文件

字符流主要用于文本文件

通常使用高效缓冲流：

- 写数据：BufferedOutputStream
- 读数据：BufferedInputStream

BIO和NIO的区别

- BIO：就是传统的网络通信模型，即阻同步塞式IO。服务端创建一个ServerSocket，然后客户端使用一个Socket去连接ServerSocket，进行通信。服务端的Socket和客户端的Socket进行的就是阻塞式通信，客户端的Socket发送一个请求，服务端进行处理后并响应，在服务端处理的过程中，客户端同步等待客户端的响应，只有响应之后才可以接着下一步操作，在客户端比较多的情况，服务端需要大量的线程建立Socket来服务客户端，可能导致服务端的负载过高，最后奔溃宕机。针对每一个客户端请求都会长期维护一个通信的线程，即使没有发送消息。
- NIO：同步非阻塞式IO。使用缓冲区，一般都是把数据写入到buffer中，从buffer中读取数据。针对不同数据类型有不同的buffer。Channel（管道）NIO通过Channel来进行数据的读写。Selector（多路复用器），Selector会不断轮询注册的Channel，如果某个Channel发生了读写事件，Selector就会把这些Channel读取出来，进行IO操作。一个Selector就通过一个线程发送响应结果，发送完毕后线程就会被回收。NIO是非阻塞的，因为不管多少客户端都可以接入服务端，客户端并不会耗费一个线程，只会创建一个连接注册到Selector上，一个Selector不断轮询，发现事件了就通知你，然后启动一个线程处理请求即可。

反射

什么是反射

反射就是动态的获取任意一个类的对象所有属性和方法的技术，可以执行方法或对属性赋值。

用处：在框架中经常使用反射，比如自动注入，只需要定义需要使用的类并且加上@Autowire注解，就可以访问到当前类的所有方法和属性，并且赋值。

类加载的双亲委派机制

- JVM加载Java类的流程如下：
    - Java源文件 ----> 编译为class文件 ----> 类加载器(ClassLoader)加载class文件 ----> 转换为实例

ClassLoader是如何加载文件的?

根据不同的来源，Java使用不同的加载器来加载

- Java核心类，由BootstrapClassLoader（根加载器）加载，BootstrapClassLoader不继承于ClassLoader，是JVM内部实现的，所以通过Java访问不到。
- Java扩展类，由ExtClassLoader（扩展类加载器）加载
- 项目中编写的类，由AppClassLoader（系统加载器）加载

双亲委派：就是加载一个类时，会先获取到一个AppClassLoader的实例，然后向上层层请求，先由BootstrapClassLoader加载，如果BootstrapClassLoader没有再去ExtClassLoader查找，如果还是没有就会来到AppClassLoader查找，如果还没有就会报错。

多线程

创建多线程的方式有几种?

1. 继承Thread类：重写run()方法，调用start()启动线程。
2. 实现Runnable接口：重写run()方法，把实现了Runnable接口的实现类当作参数传入Thread类中，调用Thread.start()方法。
3. 实现Callable接口：重写call()方法，该方法可以有返回值，其余功能同Runnable。
4. 使用线程池的方式：使用Executors创建线程池，executorService.execute(实现了Runnable接口的实现类);，executorService.submit(实现了Callable接口的实现类);

线程的生命周期

- 新建：当一个Thread类或其子类被声明创建时，线程处于新建状态。
- 就绪：处于新建状态的线程被start后，进入线程队列等待CPU时间片，此时具备运行条件，但没分配到cpu资源
- 运行：当就绪线程被调度并获得cpu资源时，便进入运行状态
- 无限等待：调用wait()/join()方法，不会被CUP分配执行权，需要显示唤醒
- 限期等待：再调用sleep()方法时，在一定时间后会由系统自动唤醒
- 阻塞：等待获取锁。
- 死亡：线程完成它全部的工作或线程被提前强制性的终止或出现异常导致结束。

谈谈对Volatile关键字的理解（根据JMM原理）

Volatile 是 Java 虚拟机提供的轻量级的同步机制。

- 保证可见性
- 不保证原子性
- 禁止指令重排

保证可见性解释：根据JMM内存模型，程序运行时，JVM 会为每个线程创建一个独立私有的工作空间，而 Java 内存模型中规定所有变量都存放在主内存中，主内存时共享区域，所有线程都可以访问，但是线程对变量的操作必须在工作内存中进行，所以线程会把主内存的变量复制到自己的工作内存，然后对其操作，完成后再写回主内存，不能直接操作主内存的变量，各个线程中的工作内存存放主内存的变量副本拷贝，因此不同的线程之间无法访问对方的工作内存，线程间的通信主要通过主内存来完成。使用 Volatile 关键字后，如果线程对变量操作完成并写回主内存后，会使其他线程中的变量地址值无效，需要重新读取。 **原理是当 CPU 写数据时，如果发现操作的变量时共享变量时，也就是其他 CUP 也存在该变量的副本，就会通知其他 CPU 将当前变量的缓存行设置为无效状态，当其他 CPU 使用该变量时，会先检查整个变量的缓存是否有效，如果失效就会重新从内存读取整个变量，MESI协议。**

不保证原子性解释：虽然在线程操作并写回变量时会通知其他线程，但是存在多个线程的多次操作，还是会有重复提交相同数据的情况，造成数据的丢失，比如某个线程在工作内存操作完数据，并且发现可以写回到主内存，正准备写回主内存，这是该线程执行权被另一个线程抢走了，并且把数据操作后写回主内存，这时通知其他线程，但是上一个线程可能在通知之前就已经写回主内存了。**根据 JVM 编译的字节码文件来看，一个简单的 i++ 操作会被分为三个操作，先读取原始值，再修改，最后写回，每个操作都会有间隙，多线程下并不能保证原子性。**使用原子包装的整型类解决，即 AtomicInteger.getAndIncrement

禁止指令重排解释：为了提高性能，编译器会对指令做重排。指令重排需要考虑指令之间的数据依赖性。因为指令重排的存在，所以在多线程环境下，最终的结果无法预测。volatile 解决指令重排的原理：**先了解一个概念“内存屏障”，是一个CPU指令，作用有两个，一是保证特定操作的执行顺序，二是保证某些变量的内存可见性。由于编译器和处理器都能执行指令重排优化，也就是说通过插入内存屏障禁止在内存屏障前后的指令执行指令重排优化。内存屏障的另一个作用就是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取这些数据的最新版本。**

谈谈你对 CAS 的理解

CAS是原子包装类（Atomic）下的 compareAndSet() （比较并交换）方法。

unsafe是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地方法（native）来访问，unsafe相当于一个后门，基于该类可以直接操作特定内存数据。unsafe在sun,misc包，内部方法可以像 C 指针一样直接操作内存。

注意：unsafe类中所有方法都是native修饰的，表示unsafe类中的方法都是操作系统底层资源执行相应任务。

valueOffset 表示该变量在内存中的偏移地址，因为unsafe就是根据内存偏移地址获取数据的

它是一条cpu并发原语，功能是判断内存某个位置的值是否为预期值，如果是则改为更新值，不是就不做操作，这个过程就是原子性。

原语的执行必须时连续的，在执行过程中不允许被打断，也就是说CAS是一条CPU的原子指令，不会造成数据不一致的问题。

CAS的缺点

- 循环时间长，cpu开销大
- 只能保证一个共享变量的原子操作

- 引出ABA问题

谈谈什么是ABA问题

ABA问题是，当前存在两个线程，线程1和线程2，两个线程都读取到内存数据为A，线程2休眠10秒，线程1对内存的数据A修改为B，线程1把数据写回主内存后，又对其进行修改，把数据B修改为A，这时线程B苏醒，继续对内存中的数据修改，发现数据还是期望值A，然后就修改成功，但是这中间被其他线程修改过一次，这就是ABA问题。

解决ABA问题：使用时间戳原子引用类`AtomicStampedReference`，时间戳原子引用就是相当于在之前的基础加一个版本号，每一次修改都会版本号加一，这样就可以防止出现ABA问题了。`atomicStampedReference.compareAndSet(200, 201, atomicStampedReference.getStamp(), stamp += 1);`，传入期望值、修改值、当前版本号、修改后的版本号，这样一来，就算某个线程重复修改造成ABA问题，版本号也不会相同，从而避免ABA问题。

sleep和wait的区别

- 类型不同
  - sleep是定义在Thread类上面
  - wait是定义在Object类上面
- 对于资源锁处理不同
  - sleep不会释放锁
  - wait会释放锁
- 使用范围不同
  - sleep可以使用在任何代码块
  - wait必须使用在同步代码块
- 生命周期不同
  - 当线程调用sleep方法时，会进入到timeed waiting状态，等待计时结束自动放开
  - 当线程调用wait方法时，会进入到waiting状态，需要调用notify或notifyAll来唤醒线程

synchronized和lock的区别

synchronized是JVM层面的一个关键字，lock是jdk提供的一个类。

synchronized不需要手动释放锁，lock需要手动释放锁

synchronized不可以中断，lock可以手动中断，调用interrupt()可以中断

synchronized默认非公平锁，lock默认也是非公平锁，但是可以传入true指定为公平锁

synchronized没有condition条件，lock可以绑定condition，从而精确唤醒，而不是像synchronized只能随机唤醒一个或唤醒全部。

什么是死锁

死锁就是两个线程互相占用对方所需要的资源，相互等待对方把自己所需要的资源释放出来。

怎么防止死锁?

- 采用trylock(timeout)方法，设置超时时间，超时后会自动退出，防止死锁。
- 减少同步代码的嵌套操作
- 减低锁的粒度，尽量不要多个功能共用一把锁

Java中的公平锁和非公平锁是什么

公平锁指多个线程按照申请锁的顺序来获取锁

非公平锁指多个线程不按照申请顺序来获取锁，而是根据线程的优先级来获取锁，优先级高的会先抢到锁。有可能造成优先级或饥饿的现象。

并发包的ReentrantLock默认是非公平的，如果指定参数为 true 就是公平锁。非公平的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。

谈谈什么是自旋锁

是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

> 代码演示

```
public class SpinLockDemo {
    AtomicReference<Thread> atomicReference = new AtomicReference<Thread>();

    public void myLock() {
        Thread thread = Thread.currentThread();
        System.out.println(Thread.currentThread().getName() + "\t myLock执行了");
        while (!atomicReference.compareAndSet(null, thread)) {

        }
    }
    public void myUnlock() {
        Thread thread = Thread.currentThread();
        atomicReference.compareAndSet(thread, null);
        System.out.println(Thread.currentThread().getName() + "\t myUnlock执行
了");
    }
    public static void main(String[] args) {
        SpinLockDemo spinLockDemo = new SpinLockDemo();

        new Thread(() -> {
            spinLockDemo.myLock();
            try {
                Thread.sleep(5000);
            } catch (Exception e) {
                // TODO: handle exception
            }
            spinLockDemo.myUnlock();
```

```
        }, "A").start();

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            // TODO: handle exception
        }

        new Thread(() -> {
            spinLockDemo.myLock();
            spinLockDemo.myUnlock();
        }, "B").start();
    }
}
```

使用原子引用 Thread 类，线程 A 访问资源类的 myLock 方法，由于第一次访问，compareAndSet 执行成功，然后对其取反为 false 跳出循环，这时线程 A 休眠 5 秒，线程 B 执行 myLock 方法，但是在线程 B 执行compareAndSet 方法时，因为线程 A 已经对 atomicReference 中的 变量修改过，所以线程 B 一直 atomicReference 方法都是 false 然后取反为 true 一直处于循环状态等待线程 A 去 myUnlock 中再次调用 atomicReference 方法把数据修改为 null，这时线程 B 就可以跳出循环继续执行，这就是自旋锁的基本原理。

谈谈你对阻塞队列的认识

是什么：底层还是一个集合。当队列是空的时候，从队列获取元素会被阻塞，直到其他线程往空的队列插入一个元素。当队列是满的时候，向队列插入数据会被阻塞，直到其他线程从队列移除一个或清空队列。

为什么用：不需要关心什么时候阻塞线程，什么时候唤醒线程，这一切都交给阻塞队列控制。

三种常用的阻塞队列：ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue。

SynchronousQueue常用在：生产者消费者模式、线程池、消息中间件。

谈谈什么是线程池

线程的主要作用是控制运行线程的数量，处理过程中将任务放入队列，然后在创建线程后启动这些任务，如果线程数量超过了最大的线程数量，那么超出的线程排队等候，等其他线程执行完毕，再从队列中取出任务来执行。

优点：

- 降低资源消耗：通过复用已创建的线程，避免了线程的创建和消耗。
- 提高响应速度：当任务到达时，不需要创建线程就可以执行。
- 提高线程可管理性：限制了线程的无限创建，并且进行统一的分配、调优和监控。

三个常见的创建线程方法：

- newFixedThreadPool：创建固定线程数量的线程池
- newSingleThreadExecutor：创建只有一个线程的线程池
- newCachedThreadPool：创建可扩容缓冲的线程池

以上三个方法都是调用的ThreadPoolExecutor类，其中底层有7个参数

- corePoolSize：线程池中常驻核心线程数

- maximumPoolSize：线程池能够最多容纳的线程数，必须大于1
- keepAliveTime：多余的空闲线程的存活时间。当线程数超过corePoolSize时，当空闲时间达到keepAliveTime时，销毁到只剩下corePoolSize数量的线程。
- unit：keepAliveTime的单位
- workQueue：任务队列
- threadFacory：表示生成线程池中工作线程的线程工程，一般使用默认
- handler：拒绝策略，表示当队列满了且工作线程大于等于当前最大线程数，以哪种方式处理解决任务。

线程池的工作流程

1. 创建线程池，开始等待
2. 当调用execute()方法添加一个任务请求后，线程池会做出以下判断
    1. 如果当前正在运行的线程数小于corePoolSize，马上创建线程执行该任务
    2. 如果当前运行的线程数大于或等于corePoolSize，那么将这个任务放到阻塞队列
    3. 如果这个时候阻塞队列满了且正在运行的线程数量小于maximumPoolSize，那么会创建非核心线程运行该任务
    4. 如果队列也满了且正在运行的线程数量大于等于maximumPoolSize，那么线程池会启动饱和拒绝策略来执行
3. 当线程完成任务时，会从队列取出下一个任务来执行
4. 当一个线程无事可做超过一定时间（keepAliveTime）时，线程会判断
    1. 如果当前运行线程大于corePoolSize，那么当前这个线程就会被停止
    2. 所有线程完成任务后，线程池会收缩到corePoolSize的大小

实际中使用哪一种方式创建线程池

根据阿里巴巴Java开发手册，不建议使用jdk提供的三种创建线程池的方式，而是使用ThreadPoolExecutor自定义线程池的参数。因为jdk自带的三种方式创建线程池可能使队列长度或线程数量为Integer.MAX_VALUE，造成大量的请求或线程，从而导致OOM

谈谈你对ThreadLocal的理解

ThreadLocal是一个线程变量的工具类，其内部维护着一个map。

> 通过ThreadLocal的get和set方法查看

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

```
public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            //把当前ThreadLocal当作建，传进来的具体数据当作值存入map
            map.set(this, value);
        else
            createMap(t, value);
    }
```

可以看出每一个线程都有一个map对象，map对象保存本地线程对象的副本变量，所以对于不同的线程获取副本的值时，别的线程不能获取当前副本的值，形成了副本隔离，互不干扰。

使用场景有：在获取JDBC连接对象时，如果每个DAO都重新获取一次连接对象，那么在service中的事务控制就不会生效了，因为多个JDBC连接之间没有任何联系，这时使用ThreadLocal来改进，创建一个连接的工具类，使用ThreadLocal保存连接对象，这样其他线程使用连接对象时就保证了使用的是同一个对象，并且不会互相干扰。

JavaWeb

Servlet生命周期

1. Web容器加载Servlet类并实例化（默认延迟加载一次），可以指定在容器启动时加载`<load-on-startup>1</load-on-startup>`
2. 运行init()方法初始化（执行一次）
3. 用户请求servlet，服务器接收到请求后执行service
4. service运行与请求方式对应的方法（doGet或doPost）
5. 销毁实例时调用destroy方法（执行一次）

转发（forward）和重定向（redirect）的 区别

1. 转发是容器控制跳转，浏览器只用把内容读取出来，所以地址栏不变。对于客户端来说始终都是一次请求，所以保存在request的数据可以传递。
2. 重定向是服务器收到请求后，返回一个状态码（302）给浏览器，浏览器解析新的的地址进行跳转，地址栏会变。对于客户端是两次请求，所以request的数据就不可以传递了，如需数据传递就要使用session对象
3. 转发效率更高，推荐使用转发，但是转发不能访问到其他服务器上，重定向可以。

JSP四大域对象

- ServletContext：Context域，只能在同一个web应用中使用（全局）
- HttpSession：Session域，只能在同一个会话中使用
- HttpServletRequest：Request域，只能在一个请求中使用，转发可以，重定向无效。
- PageContext：page域，只能在当前JSP页面中使用。

JSP内置对象

NO	内置对象	说明
1	pageContext	页面上下文对象，可以取得任何范围内的参数。
2	request	向客户端请求数据
3	response	服务器对客户端的响应，传送数据到客户端。
4	session	会话，保存每个用户的信息，跟踪用户的操作
5	application	应用程序对象，范围是整个应用。
6	config	Servlet的配置，表示容器的配置信息
7	out	对客户端输出数据
8	page	JSP实现类的实例
9	exception	反映运行异常

Get和Post的区别

1. Get传递参数是在浏览器的地址栏传递，？连接&分割，以key-value形式。Post一般都是使用表单传递，传递到action指向的URL。
2. Get是不安全的，因为在传送过程中所有的参数都是暴露在浏览器地址栏的。Post相对安全一些，参数放在body里。
3. Get传输的数据又大小限制，Post没有大小限制，上传文件操作只能使用Post。

Spring

谈谈什么是Spring IOC

没有 IOC 的时候，比如Servlet调用Service，需要`MyService service = new MyServiceImpl()`，这时如果Service类发生变化，比如直接换了一个实现类，其中的方法实现全都不一样，这时就需要去把每一个Servlet中使用到Service的地方全都改一遍。使用IOC容器后，在Service的实现类上@Service注解，把Servlet变为Controller类，加上@Controller注解，然后在@Controller中使用到Service的上面加上@Resource注解，在服务器启动时就会创建IOC容器，然后IOC容器会创建提前定义好的bean对象，并且会把相互依赖的对象进行注入操作，通过反射创建其对应的对象。解耦，降低类之间的依赖。

IOC（控制反转）是Spring最核心的部分，IOC的前提需要先了解DI（依赖注入）。

在没有使用IOC的代码中，存在一个问题，就是多个类之间的互相依赖User user = new User();，如果某个底层的类做出改动，则依赖它的所有类都有进行改动，这种行为显然不合理，所以引入了DI的理念，就是把下层类作为参数传给上层类，实现上层类对下层类的控制。使用DI后只需要定义好接口提供给外界调用即可，DI注入包括set注入、接口注入、注解注入、构造器注入。

IOC执行过程：在spring容器启动后，读取bean的配置信息（注解、xml配置、配置类），根据Bean注册表的信息实例化Bean，再将Bean的实例化装入到容器中，提供给应用使用。

Spring Bean生命周期

1. 实例化Bean对象
2. 设置Bean的属性
3. 如果调用了各种Aware接口声明的依赖，就会在Bean初始化的阶段调用对应的方法，如：BeanNameAware，获取Bean的名字。
4. 如果实现了BeanPostProcessor接口（该接口需要另一个类中实现，并注入到bena容器），则会调用BeanPostProcessor的前置初始化方法postProcessBeforeInitialization
5. 如果实现了InitializingBean接口，则会调用init-method方法
6. 调用bean自身的init方法
7. 调用BeanPostProcessor的postProcessAfterInitialization后置方法。
8. 销毁，DisposableBean接口的销毁方法，和自身的销毁方法

Spring的作用域

1. singleton(单例)：spring默认的作用域，每个IOC容器创建唯一的一个bean实例
2. prototype(多例)：针对每个getBean请求，容器都会单独创建一个bena实例
3. request：针对每个HTTP请求容器都会单独创建一个bean实例
4. session：同一个session范围内使用同一个bean实例
5. GlobalSession：用于protlet容器，全局的HTTP Session 共享一个bean实例

注意：Spring 中的 Bena 是不安全的，避免在 Bean 对象中存放公共实例变量，比如 Service、Controller。

SpringAOP

在不修改原来代码的基础上对其进行增强，用到了单例模式的设计思想。可以把公共代码进行集中处理，减少重复代码。

SpringAOP多用于在日志系统、安全、事务等功能中。

AOP底层基于动态代理技术，JDK动态代理或cglib动态代理操作字节码的技术，运行时动态生成被调用类型的子类，并且实例化对象再返回给响应的代理对象。

AOP包括四种增强机制：

- 前置增强：在进入方法之前执行。
- 后置增强：在方法return后执行的操作
- 异常增强：在抛出异常后执行的操作
- 最终增强：最后一定会执行的操作
- 环绕增强：集成了以上四种增强。

核心概念：

- Aspect：切面，定义增强方法的代码，将一个普通的类定义为一个增强类
- Pointcut：切入点，定义连接查询条件，哪些类哪些方法需要增强
- Advice：指定什么时候做

了解cglib代理吗，和JDK动态的代理有什么区别

动态代理：创建一个代理类，在这个代理类中引用自己写的类，所有方法的的调用，都是先走的代理类对象，负责对自己类上的一些增强，再去调用那个类。

cglib代理和JDK动态代理的区别：

比如SpringAOP使用JDK动态代理，生成一个实现相同接口的代理类，构造一个实例对象出来，JDK动态代理在类有接口的时候，就会来使用。如果没有实现接口，那么SpringAOP就会使用cglib动态代理，生成自己写的类的一个子类对象，并且动态生成字节码，覆盖一些方法，在方法中加入增强代码。

Spring中的设计模式

- 在BeanFatory和Application中使用了工厂模式。
- 在创建Bean的过程中使用了单例模式和原型模式
- AOP技术使用了代理模式、装饰着模式、适配器模式。
- 事件监听使用的是观察者模式。

Spring的事务

在需要事务的方法上加事务注解，spring就会根据AOP技术在方法开始时开启事务，结束时根据结果决定提交事务或回滚事务。

隔离级别：

1. DEFAULT：使用数据库默认的隔离级别。
2. READ_UNCOMMITTED：最低 级别，允许看到其他事务未提交的数据，会产生幻读、脏读、不可重复读。
3. READ_COMMITTED：只能读取已提交的数据，可以防止脏读，会出现幻读和不可重复读。
4. REPEATABLE_READ：防止脏读、不可重复读，会产生幻读。
5. SERIALIZABLE：最高级别，事务处理为串行，阻塞的，能避免所有情况。

脏读、幻读、不可重复读的区别：

- 脏读：在一个事务进行更新数据还未提交的时候，另一个事务来读取数据，在第二个事务把数据读取出来后第一个事务回滚了，这时第二个事务读取的数据就是错误的，所谓的脏读。
- 幻读：好比注册操作，第一个用户注册判断用户名可以用，但是还没提交，这时又来一个用户使用同样的用户名判断同样可以使用，这时只有一个用户可以成功提交，而剩下一个事务明明判断过该用户名可以使用但是提交失败，这就是幻读。
- 不可重复读：一个事务多次读取数据，在读取数据的同时又来一个事务修改了数据，就会造成第一个读取数据的事务读取的数据不一致的问题，就是不可重复读。

传播机制（Propagation）：

- REQUIRED：支持当前事务，如果当前没有事务就新建一个事务。常用。
- SUPPORTS：支持当前事务，如果没有事务，就以无事务的方式运行。
- MANDATORY：支持当前事务，如果没有事务，就抛出异常。
- NESTED：支持当前事务，如果当前事务存在，则执行一个嵌套事务，如果当前没有事务，就新建一个事务。外层事务回滚连带内存事务。嵌套事务，外层代码出错则全部回滚，内层代码出错只回滚内层方法。
- REQUIRED_NEW：新建事务，如果当前存在事务，就挂起当前事务。
- NOT_SUPPORTED：以非事务方式执行，如果当前存在事务就挂起当前事务。
- NEVER：以非事务方式执行，如果当前存在事务，就抛出异常。

Mybatis

Mybatis中#和$的区别

#{}：解析为一个JDBC预编译语句的参数标记符。相当于一个参数占位符？，实际的SQL语句如 `select * from user where name = #{name}` 解析为 `select * from user where name = ?`，参数是动态传进来的。

${}：是直接把值替换进来，SQL语句如 `select * from user where name = ${name}` 解析为 `select * from user where name = jack`。

总结：预编译的效率更高，SQL可以重复利用，并且能够防止SQL注入，能用预编译就用预编译 `#{}`。但是有些特殊情况需要使用 `${}`，比如表名、字段名需要动态替换时，就需要用到 `${}`，因为预编译会在传进来的参数上加上单引号，表名和字段名不可以加单引号，而$传进来的是什么就是什么。

SQL注入：不使用预编译的情况下,会出现这种情况 `select * from user where name = jack and 1 = 1`，这个SQL永远都是true，也就是说可以把所有数据都查询出来，具有安全隐患,如果使用了预编译就会这样 `select * from user where name = 'jack and 1 = 1'`，此时会把整个参数当作一个字符串来查询,可以避免SQL注入。

Mybatis的缓存

缓存就是把查询出来的结果存储起来，下一次查询就不去查询数据库，而是直接查询缓存。为了提高查询效率，降低数据库压力。

一级缓存：在同一个SqlSessin对象，在参数和SQL完全一样的情况下，只执行一次SQL语句，默认开启，不能关闭。

二级缓存：二级缓存在SqlSessionFacory生命周期中，需要手动开启。所有的select语句都会被缓存,所有的更新语句都会刷新缓存.

使用场景：在查询操作高，更新频率低的时候，每一次更新操作都会刷新缓存，所以在使用二级缓存机制时尽量减少更新操作。

Mybatis分页插件的原理

有哪些分页方式?
一般分为物理分页和逻辑分页。

- 逻辑分页：指使用Mybatis自带的RowBounds进行分页，它会一次查询出多条数据，然后检索分页中的数据。一次查询所有。
- 物理分页：指从数据库查询指定条数的数据，平常使用pageHelper实现的就是物理分页。需要多少查询多少

逻辑分页在性能上不如物理分页，每次都查询所有的数据会导致系统性能的下降。

底层原理是使用了一个拦截器，通过start、rows、sql来设置查询的参数。

SpringMVC

谈谈你对SpringMVC的理解

SpringMVC是一个基于Java实现的轻量级的Web框架。

核心组件：

- DispatcherServlet：前端控制器，所有流程的控制中心，控制其他组件的执行。
- Handler(Controller)：后端控制器，负责处理请求的控制逻辑。
- HandlerMapping：映射器对象，用于管理URL与对应的Controller对应的映射。
- HandlerAdapter：适配器，主要用来处理方法参数、注解、视图解析器等。
- ViewResolver：视图解析器，解析对应视图关系。

执行流程：

- 请求匹配到前端控制器（DispatcherServlet）的请求路径映射
- 前端控制器接收到请求后把请求交给处理器映射器（HandlerMapping）
- HandlerMapping根据用户的请求URL查找匹配该URL的Handler，并返回一个执行链（HandlerExecuttionChain）
- DispatcherServlet再请求处理器适配器（HandlerAdapter），调用相应的Handler进行处理并返回ModelAndView给DispatcherServlet。
- DispatcherServlet对View进行渲染，将页面响应给用户

SpringCloud

谈谈你对SpringCloud的认识

SpringCloud是一个大的集合，把当前主流框架集中起来，以SpringBoot的方式开发。屏蔽了复杂的配置，提供了一套简单易上手的分布式开发工具箱。

SpringCloud解决了分布式开发中的问题，提供了许多组件来解决问题，比如服务注册与发现，负载均衡，熔断等非业务的共性问题。

说说SpringCloud有哪些组件，解决了什么问题

- 注册中心：Eureka，把每个微服务注册到注册中心上，对其他微服务暴露当前微服务
- 客户端负载均衡：Ribbon，指定以轮询或其他方式进行负载均衡
- 声明式远程方法调用：Fegin，只需要创建一个接口并用注解方式配置它，即可完成服务提供方的接口绑定
- 服务降级、熔断：Hystrix，如果某个微服务出现了故障，通过熔断监控应用是否出现故障，如果出现故障则触发降级机制，返回一个提前准备好的一个备选方案，防止长时间等待出现异常。
- 网关：Zuul，主要用来对请求的路由和过滤两个功能，路由是把一个外部的请求转发到具体的微服务上，实现外部统一访问入口的基础，过滤主要是把请求处理过程进行干预，实现请求校验、聚合等功能。
- 数据监控：Hystrix Dash Board，可以查看应用中详细的数据，比如某个微服务的访问量。

MySQL数据库

MySQL 的 MyISAM 和 InnoDB 存储引擎的区别

- MyISAM：不支持事务和外键。锁级别是表锁。索引文件和数据文件分开，这样在内存里可以缓存更多的索引，查询性能更好，适合大量查询的场景。比如，报表系统。
- InnoDB：支持事务和外键，使用行锁。

谈谈数据库设计的三大范式

- 第一范式：列不可分（原子性）
- 第二范式：要有主键
- 第三范式：不可传递依赖

如何防止SQL注入

SQL注入：通过字符串的拼接构成了一种特殊的查询语句，如`select * from user where name = 'or 1=1.....'`，这样不管后面写什么都是正确的。

解决：使用预处理（PreparedStatement）对象，而非Statement对象，并且预处理对象还可以提高SQL的执行效率。

如果使用Mybatis还可以使用#{}传值。

谈谈你对MySQL事务的理解

> 事务的特性 ACID

原子性是基础，隔离性是手段，一致性是约束条件，而持久性是目的。简称ACID。

- 原子性：事务中包含的操作，要么一起成功，要么一起失败。
- 一致性：指数据库的数据在事务的操作前后的必须满足的条件约束。比如两个账户转账前和转账后，两个账户的总额不变。
- 隔离性：指一个事务的执行不能被其他事务干扰，设置不同的隔离级别，互相干扰的程度会不同。
- 持久性：指事务一旦提交，结果便是永久性，即使发生宕机，也可以通过日志恢复数据。

> 事务的隔离级别

- 读未提交：事务A读到了事务B未提交的数据，会出现脏读。
- 读已提交（不可重复读）：一个事务在另一个事务操作数据时，读取两次数据，可能得到不一样的数据。
- 可重复读（MySQL默认隔离级别）：和不可重复读不同的地方在，事务前后两次读取的数据是相同的，就算期间被其他事务修改，还是读取相同的数据。会出现幻读。
- 串行化：最高事务级别，事务A执行期间不允许其他事务执行。降低并发。解决了幻读。

MySQL怎么实现默认的可重复读隔离级别?

MVCC（多版本并发控制）机制，InnoDB引擎会在表字段后面加两个隐藏字段，一个是保存行的创建时间，一个是删除行的创建时间。但是存储的不是时间，而是事务ID，这个事务ID自增且全局唯一。在查询数据时，一定会查询到创建事务ID<=当前事务ID的那一行，在修改数据时，会有把当前行数据复制一份，然后事务ID加一，在删除数据后，会有一个删除事务ID，查询时的事务ID必须小于删除事务ID，从而保证了可重复读。

> 当前读和快照读

- 当前读：加锁的操作都为当前读
- 快照读：不加锁的操作，普通select

谈谈索引是什么

> 索引概述

- 索引是帮助Mysql高效获取数据的**数据结构。**
- 如果没有索引，查询某个数据，需要从头开始一个一个的查询，直到找到目标数据，而索引的出现可以直接定位到需要查找的数据的位置，提高了查找数据的效率。**索引就是排好序的快速查找数据结构。**
- 除了数据本身之外，数据库还维护着一个满足特定查找算法的数据结构，这种数据结构以某种方式指向数据，这样就可以在这些数据结构基础上实现高级查找算法，这种数据结构就是索引。
- 索引本身占内存也很大，所以是以索引文件的形式存储在磁盘上。
- 平常所说的索引，一般都是B+树结构的索引，除了B树索引还有hash索引。
- 索引的优势：
  - 提高检索效率，减少数据库的IO成本。
  - 通过索引对数据排序，降低了排序成本，减少CPU的消耗。
- 索引的劣势：
  - 索引也是一张表，需要占用空间。
  - 虽然索引提高了查询速度，但是会降低增删改的速度，每次都数据的修改，数据库不仅保存数据，还要保存变化后的索引文件。

创建索引的SQL：`create index idx_user_nameEmali on user(naem,email);`

InnoDB引擎要求必须有主键，默认内置就有一个根据主键建立的索引，叫做聚簇索引。并且数据本身同时也是个索引文件，默认就是主键ID的聚簇索引。所以，InnoDB不建议使用UUID作为主键，因为占用内存过多，影响查询效率，建议使用自增主键，保证插入的顺序。

索引的数据结构

> 二叉树

二叉树：

使用二叉树优化索引，因为二叉树的时间复杂度是Ologn，并且在修改数据后有可能变为线性表，使用了二分搜索法，在数据量过大时，发生的IO操作太多，性能不理想。

> B树

B树和二叉树的复杂度相同，都是Ologn，也有可能变为线性表，效率不高

> B+树

B+树相比B树可以存储更多的数据，除了叶子节点，其他节点都不存储数据，只存储孩子节点的引用，这样就可以存储更多节点引用。数B+树的叶子节点只存储数据，并且按照大小排列，据库的索引对 B+树做了优化，支持范围统计，叶子节点可以横向跨子树统计。

总结：B+树更适合做优化索引，B+树可以根据查找的索引，对其加载一整个叶子节点的数据，相对来说对磁盘的读写更低，并且查询更稳定。

> Hash索引

理论上Hash索引比B+树效率更高，因为根据查询的关键词获取其对应的Hash值，然后再去查找数据。但是因为Hash表的限制，只能查询指定的数据，不能范围查找。不能使用部分索引键查询。并且Hash表可能变为线性的。

使用索引时需要注意什么

- 全值匹配最好
- 最佳左前缀
- 不在索引列上做任何操作，否则会导致索引失效而导致全表扫描
- 存储引擎不能使用索引范围条件最右边的列
- 尽量使用覆盖索引，减少 select *
- MySQL 在使用不等于 (!= 或 <>) 的时候无法使用索引，会导致全表扫描
- is、null、is not null 也无法使用索引
- 使用 like 以通配符开头的索引也会失效
- 字符串不加单引号索引失效
- 尽量少用 or，or 连接也会导致索引失效

如何调优SQL

> 优化

- 根据慢日志定位慢查询SQL
- 使用explain等工具分析SQL
- 修改SQL或尽量让SQL走索引

> 最佳左匹配原则

使用索引时，MySQL会一直从左到右去匹配索引，直到遇到范围查询（<、>、between、like）就停止匹配，比如建立索引（a、b、c、d），查询a=1 and b=2 and c>3 and d=4，abc索引都可以使用到，范围查询之后的d无法使用，但是如果改变建立索引的位置（a、b、d、c），就可以使用到索所有的索引。

=和in可以乱序，查询条件使用到的索引不按照建立的顺序，MySQL也会自动优化成索引可以识别的形式。

MySQL锁

> MyISAM和InnoDB锁

- MyISAM默认使用的是表级锁，不支持行级锁。适合执行全表count语句或增删改语句较少，查询较多和不需要事务的场景。
- InnoDB默认是行级锁，也支持表级锁。适合频繁增删改和需要事务的场景。

> 共享锁和排他锁

- 共享锁：可以同时读操作，不能同时写操作。
- 排他锁：其他事务无法查询或其他操作当前加锁的数据。

> 注意：行锁和索引有关系

如果查询的条件没有使用到索引，那么即使使用的是行级锁也会自动变为表级锁。

什么是悲观锁和乐观锁

- 悲观锁：对外界的修改操作持保守态度，在整个事务处理数据中将数据处于锁定状态。是利用数据库本身的锁机制来实现，会锁记录。在操作的时候就把数据锁起来，操作结束再解锁。实现方式是`select * from user where id = 1 for update`。好处是安全，缺点是性能差。
- 乐观锁：认为数据一般情况下不会造成冲突，只有在数据的提交或更新时才会检测数据是否冲突，如果发现冲突则返回冲突的信息给用户，是一种不锁记录的实现方式，采用CAS（比较并交互）模式，采用version字段或者时间戳作为判断依据。每次对数据更新都会让version+1，这样在每次提交操作时，如果version值已被更改就提交失败。如果使用业务字段作为判断依据，可能出现ABA问题。

> 乐观锁的实现方式

在表字段中加一个版本号，每次修改都会对版本号+1操作，并且校验版本号是否为预期值，如果是预期值表示执行成功，如果不是预期值将执行失败，将失败信息返回给用户。

Redis

Redis的数据类型

- String：最基本的数据类型，二进制安全
- Hash：String元素组成的字典，适合存储对象
- List：按照String插入顺序排序
- Set：String元素组成的无序列表，通过hash表实现，不允许重复
- Sorted Set：通过分数来为集合中的成员进行从小到大的排序
- HyperLog：用于计数，存储地理位置信息的Geo

MQ

谈谈开发中应用消息中间件的场景

适合场景

- 不需要实时性的

优点

- 解耦
- 异步
- 限流削峰

在不使用 MQ 的应用中，每个操作步骤都会耦合在一起，操作一、二、...是一个同步的状态，如果某个操作出现问题，可能导致整体的问题，并且需要等待上一个操作完成才能进行下一步操作，用户体验不好。

使用了MQ后，用户的请求都会先到达消息中间件，并且直接给用户一个反馈，消息再由消息中间件分发给具体的业务执行，这个过程是异步的，响应时间大大减少，并且用户的体验也有提升。

场景：

- 基础服务：订单、注册、登录 ------ > 发送邮件/发送短信

- 限流削峰：在秒杀的场景，可以使用消息中间件来给定一个最大请求值，当请求达到最大的值，就不会接收请求了。

如何保障消息的可达或不丢失

异步 confirm 模式，客户端设置 confirm 监听器，获取 MQ 服务器的异步响应，如果消息传递成功，服务器传递回 ACK=true，否则 ACK=false，publisher-confirm=true。该机制只保证消息到MQ服务器。

消息队列持久化，在消息成功传递到MQ服务器后，对消息队列持久化，否则 MQ 服务器宕机后，数据会丢失。

在消费者方进行手动确认，确保消息真正的被消费者处理了，如果没有正确处理，会重新发送。

如何保证消息的幂等性

幂等性指一次请求或多次请求某个资源，对于资源本身应该有相同的结果。

在MQ中指，消费多条相同的消息，得到与消费该消息一次相同的结果。

使用乐观锁机制保障：

在向数据库交互时，携带应该版本号，只有到版本号和数据库中的版本号相等才会操作成功，每一次操作成功，数据库的版本号+1.