一、Java 基础知识
1、Object 类相关方法
2、基本数据类型
3、序列化
4、String、StringBuffer、StringBuilder
5、重载与重写
6、final
7、反射
8、JDK 动态代理
9、Java IO
二、Java 集合框架
1、List（线性结构）
2、Map（K，V 对）
3、Set（唯一值）
三、Java 多线程
1、synchronized
2、Lock
3、volatile
4、线程的五种状态
1). New
2). Runnable
3). Blocked
4). Waiting（无限期等待）
5). Timed Waiting（有期限等待）
6). Terminated
5、 wait() 与 sleep()
6、 yield()
7、 join()
8、线程池
1)、分类
2)、线程池的几个重要参数
3)、线程池线程工作过程
4)、线程池拒绝策略（默认抛出异常）
5)、如何根据 CPU 核心数设计线程池线程数量
9、线程使用方式
10、Runnable 和 Callable 比较
11、hapens-before
12、ThreadLocal
四、Java 虚拟机
1、Java 内存结构
2、Java 类加载机制
3、垃圾回收算法
4、典型垃圾回收器
五、MySQL （Inno DB）
1、聚簇索引与非聚簇索引
2、为何使用 B 树做索引而不是红黑树?
3、最左前缀原则
4、什么情况下可以用到 B 树索引
5、事务隔离级别
6、MVCC （多版本并发控制）
六、Spring 相关
1、Bean 的作用域
2、Bean 生命周期
3、Spring AOP
4、Spring 事务传播行为

5、Spring IoC
6、Spring MVC 工作流程
七、计算机网络
1、TCP/IP 五层模型
2、浏览器输入地址后做了什么?
3、三次握手与四次挥手
4、TIME_WAIT 与 CLOSE_WAIT
5、TCP 滑动窗口
6、TCP 粘包和拆包
八、MQ 消息队列
1、场景作用
2、如何保证消息不被重复消费呢?
3、怎么保证从消息队列里拿到的数据按顺序执行?
4、如何解决消息队列的延时以及过期失效问题?消息队列满了以后该怎么处理?有几百万消息持续积压几小时,说说怎么解决?
4、如何保证消息的可靠性传输(如何处理消息丢失的问题)?
九、Redis
1、数据类型
2、Redis 如何实现 key 的过期删除?
3、Redis 的持久化机制
4、如何解决 Redis 缓存雪崩和缓存穿透?
5、如何使用 Redis 实现消息队列?
十、Nginx
1、正向代理和反向代理
2、负载均衡
3、动静分离
4、Nginx 四个组成部分

一、Java 基础知识

1、Object 类相关方法

- getClass 获取当前运行时对象的 Class 对象。
- hashCode 返回对象的 hash 码。
- clone 拷贝当前对象， 必须实现 Cloneable 接口。**浅拷贝**对基本类型进行值拷贝，对引用类型拷贝引用；**深拷贝**对基本类型进行值拷贝，对引用类型对象不但拷贝对象的引用还拷贝对象的相关属性和方法。两者不同在于深拷贝创建了一个新的对象。
- equals 通过内存地址比较两个对象是否相等，String 类重写了这个方法使用值来比较是否相等。
- toString 返回类名@哈希码的 16 进制。
- notify 唤醒当前对象监视器的任一个线程。
- notifyAll 唤醒当前对象监视器上的所有线程。
- wait 1、暂停线程的执行；2、三个不同参数方法（等待多少毫秒；额外等待多少毫秒；一直等待）3、与 `Thread.sleep(long time)` 相比，sleep 使当前线程休眠一段时间，并没有释放该对象的锁，wait 释放了锁。
- finalize 对象被垃圾回收器回收时执行的方法。

2、基本数据类型

- 整型：byte(8)、short(16)、int(32)、long(64)
- 浮点型：float(32)、double(64)
- 布尔型：boolean(8)
- 字符型：char(16)

3、序列化

Java 对象实现序列化要实现 Serializable 接口。

- 反序列化并不会调用构造方法。反序列的对象是由 JVM 自己生成的对象，不通过构造方法生成。
- 序列化对象的引用类型成员变量，也必须是可序列化的，否则，会报错。
- 如果想让某个变量不被序列化，使用 transient 修饰。
- 单例类序列化，需要重写 readResolve() 方法。

4、String、StringBuffer、StringBuilder

- String 由 char[] 数组构成，使用了 final 修饰，是不可变对象，可以理解为常量，线程安全；对 String 进行改变时每次都会新生成一个 String 对象，然后把指针指向新的引用对象。
- StringBuffer 线程安全；StringBuiler 线程不安全。
- 操作少量字符数据用 String；单线程操作大量数据用 StringBuilder；多线程操作大量数据用 StringBuffer。

5、重载与重写

- 重载 发生在同一个类中，方法名相同，参数的类型、个数、顺序不同，方法的返回值和修饰符可以不同。
- 重写 发生在父子类中，方法名和参数相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 或者 final 则子类就不能重写该方法。

6、final

- 修饰基本类型变量，一经出初始化后就不能够对其进行修改。
- 修饰引用类型变量，不能够指向另一个引用。
- 修饰类或方法，不能被继承或重写。

7、反射

- 在运行时动态的获取类的完整信息
- 增加程序的灵活性
- JDK 动态代理使用了反射

8、JDK 动态代理

- 使用步骤
    - 创建接口及实现类
    - 实现代理处理器：实现 InvokationHandler ，实现 invoke（Proxy proxy，Method method，Object[] args） 方法
    - 通过 Proxy.newProxyInstance(ClassLoaderloader, Class[] interfaces, InvocationHandler h) 获得代理类
    - 通过代理类调用方法。

9、Java IO

- 普通 IO ，面向流，同步阻塞线程。
- NIO，面向缓冲区，同步非阻塞。

二、Java 集合框架

1、List（线性结构）

- ArrayList Object[] 数组实现，默认大小为 10 ，支持随机访问，连续内存空间，插入末尾时间复杂度 o(1)，插入第 i 个位置时间复杂度 o(n - i)。扩容，大小变为 **1.5** 倍，Arrays.copyOf（底层 System.ArrayCopy），复制到新数组，指针指向新数组。
- Vector 类似 ArrayList，线程安全，扩容默认增长为原来的 **2** 倍，还可以指定增长空间长度。
- LinkedList 基于链表实现，1.7 为双向链表，1.6 为双向循环链表，取消循环更能分清头尾。

2、Map（K，V 对）

- HashMap
    - 底层数据结构，JDK 1.8 是**数组 + 链表 + 红黑树**，JDK 1.7 无红黑树。链表长度大于 8 时，转化为红黑树，优化查询效率。
    - 初始容量为 **16**，通过 tableSizeFor 保证容量为 2 的幂次方。寻址方式，高位异或，**(n-1)&h** 取模，优化速度。
    - 扩容机制，当元素数量大于容量 x 负载因子 0.75 时，容量扩大为原来的 2 倍，新建一个数组，然后转移到新数组。
    - 基于 Map 实现。
    - 线程不安全。
- HashMap (1.7) 多线程循环链表问题
    - 在多线程环境下，进行扩容时，1.7 下的 HashMap 会形成循环链表。
    - 怎么形成循环链表： 假设有一 HashMap 容量为 2 ， 在数组下标 1 位置以 A -> B 链表形式存储。有一线程对该 map 做 put 操作，由于触发扩容条件，需要进行扩容。这时另一个线程也 put 操作，同样需要扩容，并完成了扩容操作，由于复制到新数组是头部插入，所以 1 位置变为 B -> A 。这时第一个线程继续做扩容操作，首先复制 A ，然后复制 B ，再判断 B.next 是否为空时，由于第二个线程做了扩容操作，导致 B.next = A，所以在将 A 放到 B 前，A.next 又等于 B ，导致循环链表出现。
- HashTable
    - 线程安全，方法基本全用 Synchronized 修饰。
    - 初始容量为 11 ，扩容为 2n + 1 。
    - 继承 **Dictionary** 类。
- ConcurrentHashMap
    - 线程安全的 HashMap。
    - 1.7 采用分段锁的形式加锁；1.8 使用 Synchronized 和 CAS 实现同步，若数组的 Node 为空，则通过 CAS 的方式设置值，不为空则加在链表的第一个节点。获取第一个元素是否为空使用 Unsafe 类提供的 getObjectVolatile 保证可见性。
    - 对于读操作，数组由 volatile 修饰，同时数组的元素为 Node，Node 的 K 使用 final 修饰，V 使用 volatile 修饰，下一个节点也用 volatile 修饰，保证多线程的可见性。
- LinkedHashMap LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。
- TreeMap 有序的 Map，红黑树结构，可以自定义比较器来进行排序。
- Collections.synchronizedMap 如何实现 Map 线程安全？ 基于 Synchronized ，实际上就是锁住了当前传入的 Map 对象。

3、Set（唯一值）

- HashSet 基于 HashMap 实现，使用了 HashMap 的 K 作为元素存储，V 为 new Object()，在 add() 方法中如果两个元素的 Hash 值相同，则通过 equals 方法比较是否相等。
- LinkedHashSet LinkedHashSet 继承于 HashSet，并且其内部是通过 LinkedHashMap 来实现的。
- TreeSet 红黑树实现有序唯一。

三、Java 多线程

1、synchronized

- 修饰代码块 底层实现，通过 monitorenter & monitorexit 标志代码块为同步代码块。
- 修饰方法 底层实现，通过 ACC_SYNCHRONIZED 标志方法是同步方法。
- 修饰类 class 对象时，实际锁在类的实例上面。
- 单例模式

```
public class Singleton {

    private static volatile Singleton instance = null;

    private Singleton(){}

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

- 偏向锁，自旋锁，轻量级锁，重量级锁
    - 通过 synchronized 加锁，第一个线程获取的锁为偏向锁，这时有其他线程参与锁竞争，升级为轻量级锁，其他线程通过循环的方式尝试获得锁，称自旋锁。若果自旋的次数达到一定的阈值，则升级为重量级锁。
    - 需要注意的是，在第二个线程获取锁时，会先判断第一个线程是否仍然存活，如果不存活，不会升级为轻量级锁。

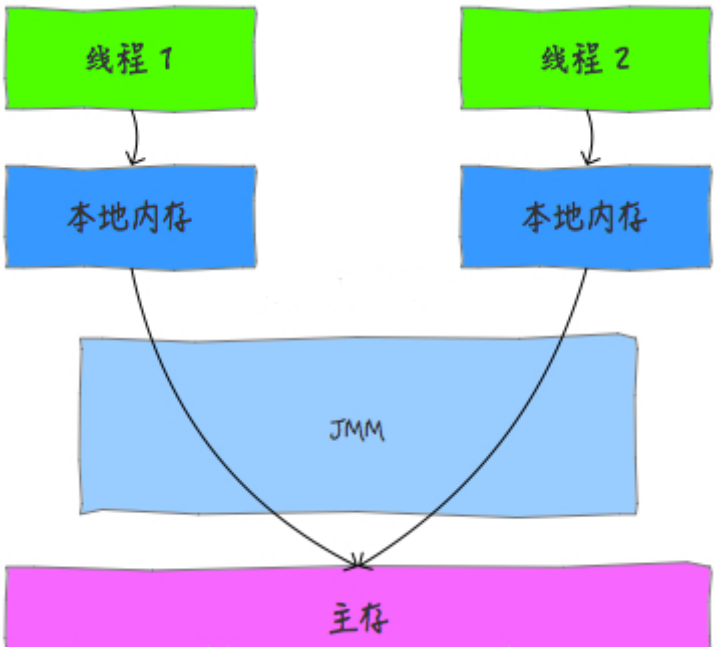
2、Lock

- ReentrantLock
    - 基于 AQS （AbstractQueuedSynchronizer）实现，主要有 state (资源) + FIFO (线程等待队列) 组成。
    - 公平锁与非公平锁：区别在于在获取锁时，公平锁会判断当前队列是否有正在等待的线程，如果有则进行排队。
    - 使用 lock() 和 unLock() 方法来加锁解锁。
- ReentrantReadWriteLock
    - 同样基于 AQS 实现，内部采用内部类的形式实现了读锁（共享锁）和写锁 （排它锁）。

- 非公平锁吞吐量高 在获取锁的阶段来分析，当某一线程要获取锁时，非公平锁可以直接尝试获取锁，而不是判断当前队列中是否有线程在等待。一定情况下可以避免线程频繁的上下文切换，这样，活跃的线程有可能获得锁，而在队列中的锁还要进行唤醒才能继续尝试获取锁，而且线程的执行顺序一般来说不影响程序的运行。

3、volatile

- Java 内存模型



- 在多线程环境下，保证变量的可见性。使用了 volatile 修饰变量后，**在变量修改后会立即同步到主存中，每次用这个变量前会从主存刷新。**
- 禁止 JVM 指令重排序。
- 单例模式双重校验锁变量为什么使用 volatile 修饰？ 禁止 JVM 指令重排序，new Object()分为三个步骤：申请内存空间，将内存空间引用赋值给变量，变量初始化。如果不禁止重排序，有可能得到一个未经初始化的变量。

4、线程的五种状态

1). New

一个新的线程被创建，还没开始运行。

2). Runnable

一个线程准备就绪，随时可以运行的时候就进入了 Runnable 状态。

Runnable 状态可以是实际正在运行的线程，也可以是随时可以运行的线程。

多线程环境下，每个线程都会被分配一个固定长度的 CPU 计算时间，每个线程运行一会儿就会停止让其他线程运行，这样才能让每个线程公平的运行。这些等待 CPU 和正在运行的线程就处于 Runnable 状态。

3). Blocked

例如一个线程在等待 I/O 资源，或者它要访问的被保护代码已经被其他线程锁住了，那么它就在阻塞 Blocked 状态，这个线程所需的资源到位后就转入 Runnable 状态。

4). Waiting（无限期等待）

如果一个线程在等待其他线程的唤醒，那么它就处于 Waiting 状态。以下方法会让线程进入等待状态：

- Object.wait()
- Thread.join()
- LockSupport.park()

5). Timed Waiting（有期限等待）

无需等待被其他线程显示唤醒，在一定时间后有系统自动唤醒。

以下方法会让线程进入有限等待状态：

- Thread.sleep(sleeptime)
- Object.wait(timeout)
- Thread.join(timeout)
- LockSupport.parkNanos(timeout)
- LockSupport.parkUntil(timeout)

6). Terminated

一个线程正常执行完毕，或者意外失败，那么就结束了。

5、 wait() 与 sleep()

- 调用后线程进入 waiting 状态。
- wait() 释放锁，sleep() 没有释放锁。
- 调用 wait() 后需要调用 notify() 或 notifyAll() 方法唤醒线程。
- wait() 方法声明在 Object 中，sleep() 方法声明在 Thread 中。

6、 yield()

- 调用后线程进入 runnable 状态。
- 让出 CPU 时间片，之后有可能其他线程获得执行权，也有可能这个线程继续执行。

7、 join()

- 在线程 B 中调用了线程 A 的 Join()方法，直到线程 A 执行完毕后，才会继续执行线程 B。
- 可以保证线程的顺序执行。
- join() 方法必须在 线程启动后调用才有意义。
- 使用 wait() 方法实现。

8、线程池

1)、分类

- FixThreadPool 固定数量的线程池，适用于对线程管理，高负载的系统
- SingleThreadPool 只有一个线程的线程池，适用于保证任务顺序执行
- CacheThreadPool 创建一个不限制线程数量的线程池，适用于执行短期异步任务的小程序，低负载系统
- ScheduledThreadPool 定时任务使用的线程池，适用于定时任务

2)、线程池的几个重要参数

- int corePoolSize, 核心线程数
- int maximumPoolSize, 最大线程数
- long keepAliveTime, TimeUnit unit, 超过 corePoolSize 的线程的存活时长，超过这个时间，多余的线程会被回收。

- BlockingQueue workQueue, 任务的排队队列
- ThreadFactory threadFactory, 新线程的产生方式
- RejectedExecutionHandler handler) 拒绝策略

3)、线程池线程工作过程

corePoolSize -> 任务队列 -> maximumPoolSize -> 拒绝策略

> 核心线程在线程池中一直存活，当有任务需要执行时，直接使用核心线程执行任务。当任务数量大于核心线程数时，加入等待队列。当任务队列数量达到队列最大长度时，继续创建线程，最多达到最大线程数。当设置回收时间时，核心线程以外的空闲线程会被回收。如果达到了最大线程数还不能够满足任务执行需求，则根据拒绝策略做拒绝处理。

4)、线程池拒绝策略（默认抛出异常）

|:---|:---| | AbortPolicy | 抛出 RejectedExecutionException | | DiscardPolicy | 什么也不做，直接忽略 | | DiscardOldestPolicy | 丢弃执行队列中最老的任务，尝试为当前提交的任务腾出位置 | | CallerRunsPolicy | 直接由提交任务者执行这个任务 |

5)、如何根据 CPU 核心数设计线程池线程数量

- IO 密集型 2nCPU
- 计算密集型 nCPU+1
  - 其中 n 为 CPU 核心数量，可通过 `Runtime.getRuntime().availableProcessors()` 获得核心数：。
  - 为什么加 1：即使当计算密集型的线程偶尔由于缺失故障或者其他原因而暂停时，这个额外的线程也能确保 CPU 的时钟周期不会被浪费。

9、线程使用方式

- 继承 Tread 类
- 实现 Runnable 接口
- 实现 Callable 接口：带有返回值

10、Runnable 和 Callable 比较

1. 方法签名不同， `void Runnable.run()` , `V Callable.call() throws Exception`
2. 是否允许有返回值， `Callable` 允许有返回值
3. 是否允许抛出异常， `Callable` 允许抛出异常。
4. 提交任务方式， `Callable` 使用 `Future<T> submit(Callable<T> task)` 返回 Future 对象，调用其 get() 方法可以获得返回值， `Runnable` 使用 `void execute(Runnable command)` 。

11、hapens-before

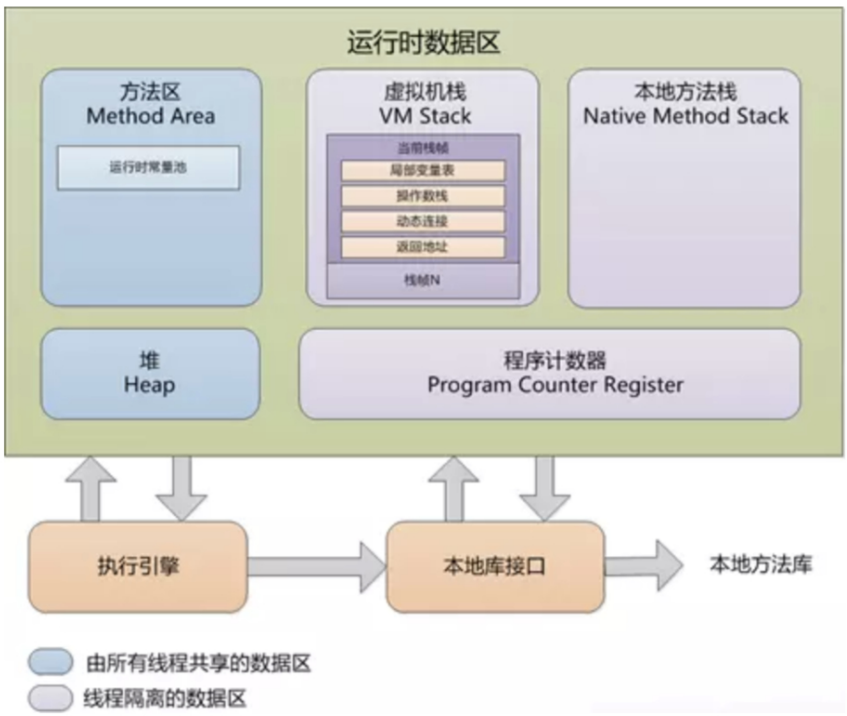
如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

12、ThreadLocal

- 场景 主要用途是为了保持线程自身对象和避免参数传递，主要适用场景是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。
- 原理 为每个线程创建变量副本，不同线程之间不可见，保证线程安全。使用 ThreadLocalMap 存储变量副本，以 ThreadLocal 为 K，这样一个线程可以拥有多个 ThreadLocal 对象。
- 实际 使用多数据源时，需要根据数据源的名字切换数据源，假设一个线程设置了一个数据源，这个时候就有可能有另一个线程去修改数据源，可以使用 ThreadLocal 维护这个数据源名字，使每个线程持有数据源名字的副本，避免线程安全问题。

四、Java 虚拟机

1、Java 内存结构



- 堆 由线程共享，存放 new 出来的对象，是垃圾回收器的主要工作区域。
- 栈 线程私有，分为 Java 虚拟机栈和本地方法栈，存放局部变量表、操作栈、动态链接、方法出口等信息，方法的执行对应着入栈到出栈的过程。
- 方法区 线程共享，存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等信息，JDK 1.8 中方法区被元空间取代，使用直接内存。

2、Java 类加载机制

- 加载 加载字节码文件。
- 链接
  - 验证 验证字节码文件的正确性。
  - 准备 为静态变量分配内存。
  - 解析 将符号引用（如类的全限定名）解析为直接引用（类在实际内存中的地址）。
- 初始化 为静态变量赋初值。

> 双亲委派模式

> 当一个类需要加载时，判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 `loadClass()` 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 `BootstrapClassLoader` 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为 null 时，会使用启动类加载器 `BootstrapClassLoader` 作为父类加载器。

3、垃圾回收算法

- Mark-Sweep（标记-清除）算法 标记需要回收的对象，然后清除，会造成许多内存碎片。
- Copying（复制）算法 将内存分为两块，只使用一块，进行垃圾回收时，先将存活的对象复制到另一块区域，然后清空之前的区域。
- Mark-Compact（标记-整理）算法（压缩法） 与标记清除算法类似，但是在标记之后，将存活对象向一端移动，然后清除边界外的垃圾对象。
- Generational Collection（分代收集）算法 分为年轻代和老年代，年轻代时比较活跃的对象，使用复制算法做垃圾回收。老年代每次回收只回收少量对象，使用标记整理法。

4、典型垃圾回收器

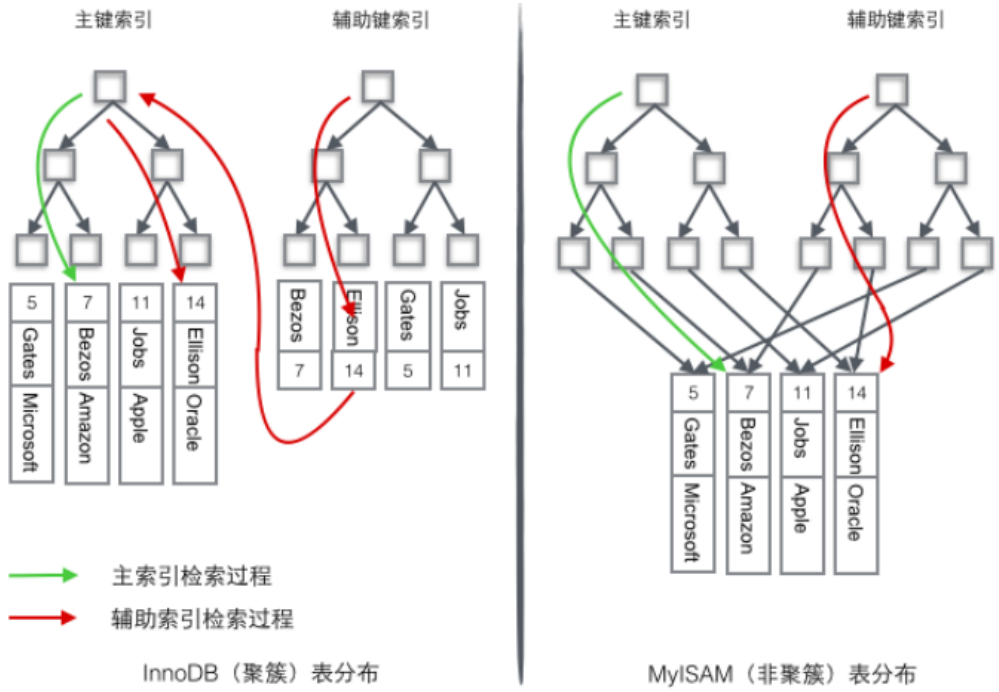
- CMS
    - 简介 以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是 Mark-Sweep 算法。
    - 场景 如果你的应用需要更快的响应，不希望有长时间的停顿，同时你的 CPU 资源也比较丰富，就适合适用 CMS 收集器。
    - 垃圾回收步骤
    1. 初始标记 (Stop the World 事件 CPU 停顿， 很短) 初始标记仅标记一下 GC Roots 能直接关联到的对象，速度很快；
    2. 并发标记 (收集垃圾跟用户线程一起执行) 并发标记过程就是进行 GC Roots 查找的过程；
    3. 重新标记 (Stop the World 事件 CPU 停顿，比初始标记稍微长，远比并发标记短) 修正由于并发标记时应用运行产生变化的标记。
    4. 并发清理，标记清除算法；
    - 缺点
        - 并发标记时和应用程序同时进行，占用一部分线程，所以吞吐量有所下降。
        - 并发清除时和应用程序同时进行，这段时间产生的垃圾就要等下一次 GC 再清除。
        - 采用的标记清除算法，产生内存碎片，如果要新建大对象，会提前触发 Full GC 。
- G1
    - 简介 是一款面向服务端应用的收集器，它能充分利用多 CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型，即可以设置 STW 的时间。
    - 垃圾回收步骤 1、初始标记(stop the world 事件 CPU 停顿只处理垃圾)； 2、并发标记(与用户线程并发执行)； 3、最终标记(stop the world 事件 ,CPU 停顿处理垃圾)； 4、筛选回收(stop the world 事件 根据用户期望的 GC 停顿时间回收)
    - 特点
        - 并发与并行 充分利用多核 CPU ，使用多核来缩短 STW 时间，部分需要停顿应用线程的操作，仍然可以通过并发保证应用程序的执行。
        - 分代回收 新生代，幸存带，老年代
        - 空间整合 总体看是采用标记整理算法回收，每个 Region 大小相等，通过复制来回收。
        - 可预测的停顿时间 使用 -XX:MaxGCPauseMillis=200 设置最长目标暂停值。

> 在 Java 语言中，可作为 GC Roots 的对象包括 4 种情况：

> a) 虚拟机栈中引用的对象（栈帧中的本地变量表）； b) 方法区中类静态属性引用的对象； c) 方法区中常量引用的对象； d) 本地方法栈中 Native 方法引用的对象。

五、MySQL （Inno DB）

1、聚簇索引与非聚簇索引



- 都使用 B+ 树作为数据结构
- 聚簇索引中数据存在主键索引的叶子结点中，得到 key 即得到 data ； 非聚簇索引的数据存在单独的空间。
- 聚簇索引中辅助索引的叶子结点存的是主键；非聚簇索引中叶子结点存的是数据的地址；
- 聚簇索引的优势是找到主键就找到数据，只需一次磁盘 IO ； 当 B+ 树的结点发生变化时，地址也会发生变化，这时非聚簇索引需要更新所有的地址，增加开销。

2、为何使用 B 树做索引而不是红黑树?

索引很大，通常作为文件存储在磁盘上面，每次检索索引都需要把索引文件加载进内存，所以磁盘 IO 的次数是衡量索引数据结构好坏的重要指标。应用程序在从磁盘读取数据时，不只是读取需要的数据，还会连同其他数据以页的形式做预读来减少磁盘 IO 的次数。数据库的设计者将每个节点的大小设置为一页的大小，同时每次新建节点时都重新申请一个页，这样检索一个节点只需要一次 IO，根据索引定位到数据只需要 h- 1 （h 为 B 树高度，根节点常驻内存） 次 IO，而 d (度，可以理解为宽度)与 h 称反比，即 d 越大，高度就越小，所以树越扁，磁盘 IO 次数越少，即渐进复杂度为 logdN ， 这也是为什么不选择红黑树做索引的原因。前面可以得出结论，d 越大，索引的性能越好。节点由 key 和 data 组成，页的大小一定，key 和 data 越小，d 越大。B + 树去掉了节点内的 data 域，所以有更大的 d , 性能更好。

3、最左前缀原则

在 MySQL 中，可以指定多个列为索引，即联合索引。比如 index(name，age) ，最左前缀原则是指查询时精确匹配到从最左边开始的一列或几列（name；name&age） ，就可以命中索引。如果所有列都用到了，顺序不同，查询引擎会自动优化为匹配联合索引的顺序，这样是能够命中索引的。

4、什么情况下可以用到 B 树索引

(1) 定义有主键的列一定要建立索引。因为主键可以加速定位到表中的某行

(2) 定义有外键的列一定要建立索引。外键列通常用于表与表之间的连接，在其上创建索引可以加快表间的连接

(3) 对于经常查询的数据列最好建立索引。

① 对于需要在指定范围内快速或频繁查询的数据列，因为索引已经排序，其指定的范围是连续的，查询可以利用索引的排序，加快查询的时间

② 经常用在 `where` 子句中的数据列，将索引建立在 `where` 子句的集合过程中，对于需要加速或频繁检索的数据列，可以让这些经常参与查询的数据列按照索引的排序进行查询，加快查询的时间。

5、事务隔离级别

- **Read uncommitted** 读未提交，可能出现脏读，不可重复读，幻读。
- **Read committed** 读提交，可能出现不可重复读，幻读。
- **Repeatable read** 可重复读，可能出现脏读。
- **Serializable** 可串行化，同一数据读写都加锁，避免脏读，性能不忍直视。

> Inno DB 默认隔离级别为可重复读级别，分为快照度和当前读，并且通过行锁和间隙锁解决了幻读问题。

6、MVCC （多版本并发控制）

- 实现细节
  - 每行数据都存在一个版本，每次数据更新时都更新该版本。
  - 修改时 Copy 出当前版本随意修改，各个事务之间互不干扰。
  - 保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃 copy（rollback）。
- Inno DB 实现

> 在 InnoDB 中为每行增加两个隐藏的字段，分别是该行数据**创建时的版本号**和**删除时的版本号**，这里的版本号是系统版本号（可以简单理解为事务的 ID），每开始一个新的事务，系统版本号就自动递增，作为事务的 ID 。通常这两个版本号分别叫做创建时间和删除时间。

详细参考：[《 脏读、幻读和不可重复读》]

六、Spring 相关

1、Bean 的作用域

|:---|:---| | 类别 | 说明 | |singleton| 默认在 Spring 容器中仅存在一个实例 | |prototype| 每次调用 getBean() 都重新生成一个实例 | |request| 为每个 HTTP 请求生成一个实例 | |session| 同一个 HTTP session 使用一个实例，不同 session 使用不同实例 |

2、Bean 生命周期

简单来说四步：

- 1. 实例化 Instantiation
- 1. 属性赋值 Populate
- 1. 初始化 Initialization
- 1. 销毁 Destruction

在这四步的基础上面，Spring 提供了一些拓展点：

- **Bean 自身的方法**: 这个包括了 Bean 本身调用的方法和通过配置文件中 %3Cbean %3E 的 init-method 和 destroy-method 指定的方法
- **Bean 级生命周期接口方法**: 这个包括了 BeanNameAware、BeanFactoryAware、InitializingBean 和 DiposableBean 这些接口的方法
- **容器级生命周期接口方法**：这个包括了 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。
- **工厂后处理器接口方法**: 这个包括了 AspectJWeavingEnabler, ConfigurationClassPostProcessor, CustomAutowireConfigurer 等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下文装配配置文件之后立即调用。

3、Spring AOP

实现方式两种：

- JDK 动态代理：带有接口的对象，在运行期实现
- CGlib 静态代理：在编译期实现。

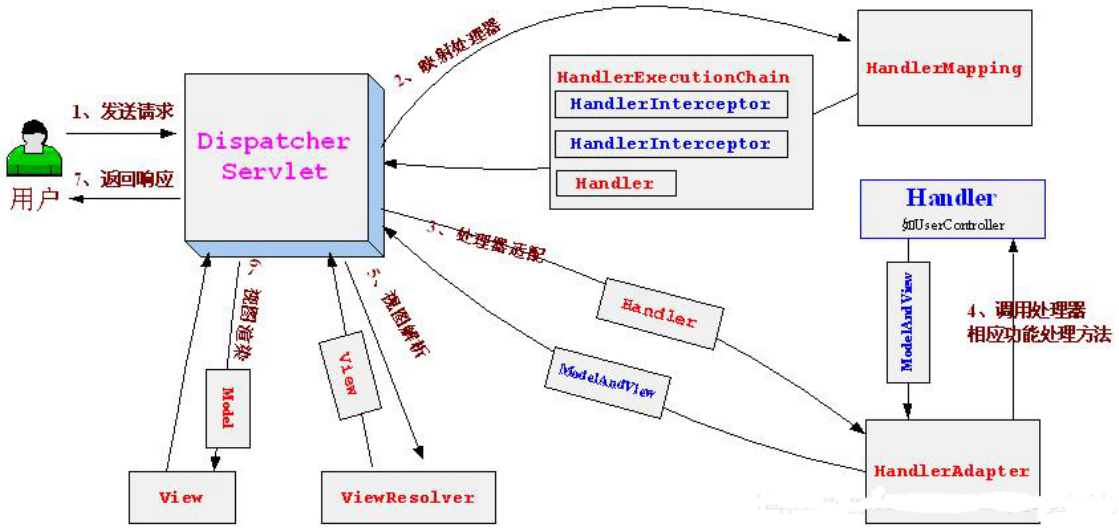
4、Spring 事务传播行为

默认 **PROPAGATION_REQUIRED** ，如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

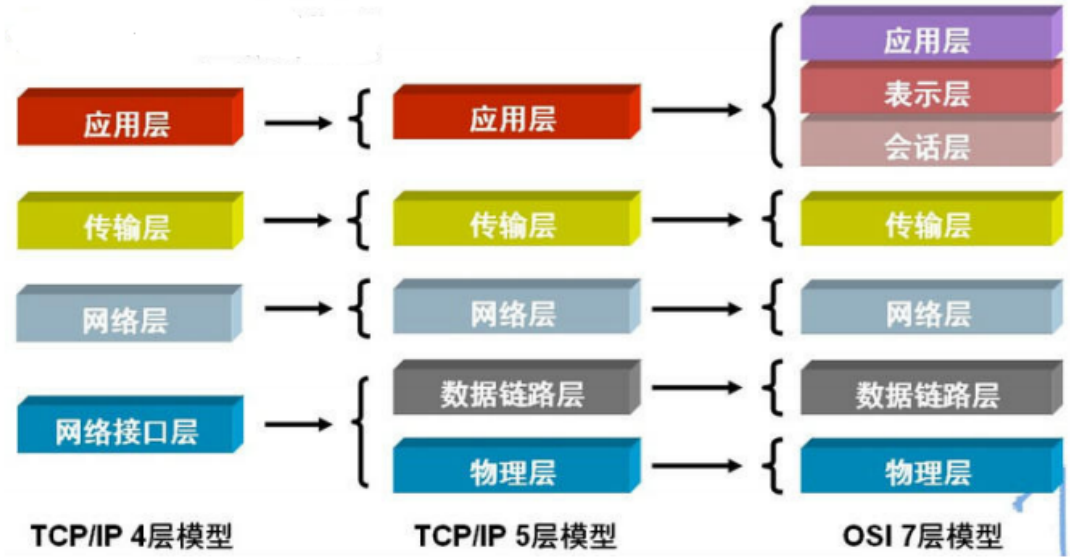
5、Spring IoC

6、Spring MVC 工作流程



七、计算机网络

1、TCP/IP 五层模型

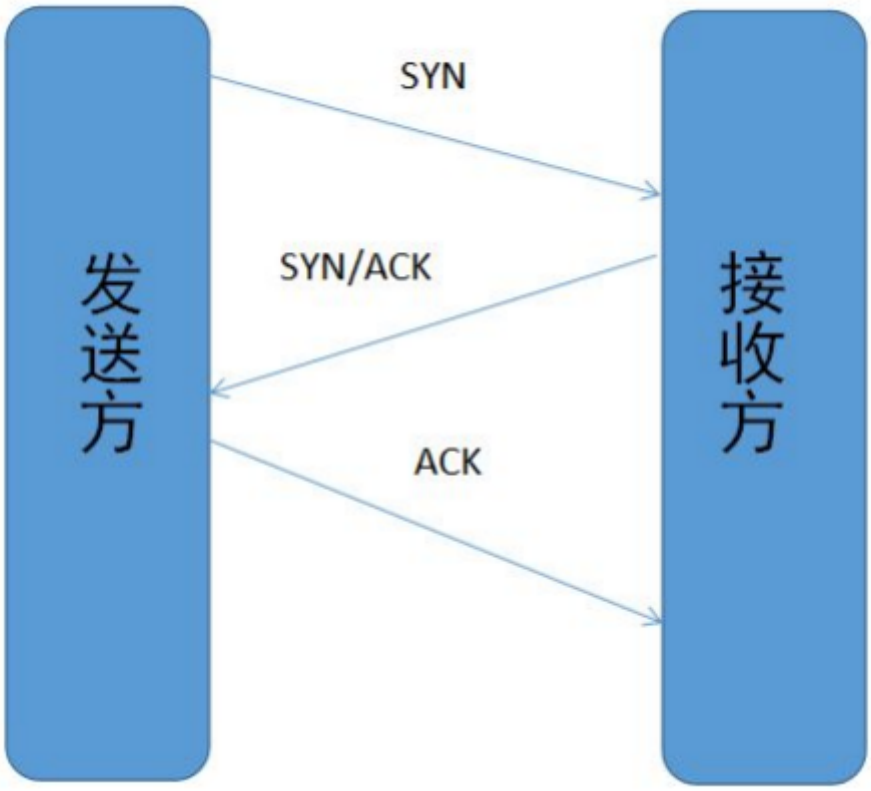


2、浏览器输入地址后做了什么?

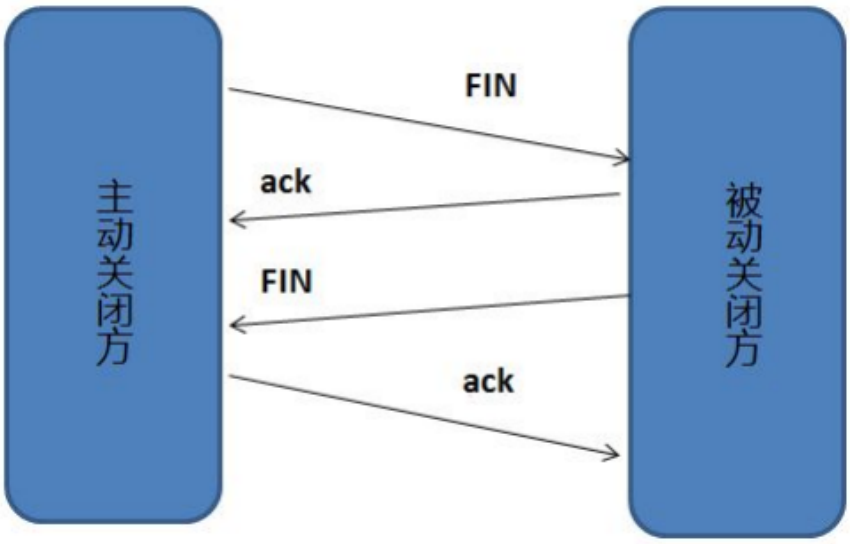
<table>
<tr><th>过程</th><th>使用的协议</th></tr>
<tr><td>1. 浏览器查找域名的IP地址
(DNS查找过程：浏览器缓存、路由器缓存、DNS 缓存)</td><td rowspan="2">DNS：获取域名对应IP</td></tr>
<tr><td>2. 浏览器向web服务器发送一个HTTP请求
(cookies会随着请求发送给服务器)</td></tr>
<tr><td>3. 服务器处理请求
(请求 处理请求 & 它的参数、cookies、生成一个HTML响应)</td><td rowspan="3">• TCP：与服务器建立TCP连接
• IP：建立TCP协议时，需要发送数据，发送数据在网络层使用IP协议
• OPSF：IP数据包在路由器之间，路由选择使用OPSF协议
• ARP：路由器在与服务器通信时，需要将ip地址转换为MAC地址，需要使用ARP协议
• HTTP：在TCP建立完成后，使用HTTP协议访问网页</td></tr>
<tr><td>4. 服务器发回一个HTML响应</td></tr>
<tr><td>5. 浏览器开始显示HTML</td></tr>
</table>

3、三次握手与四次挥手

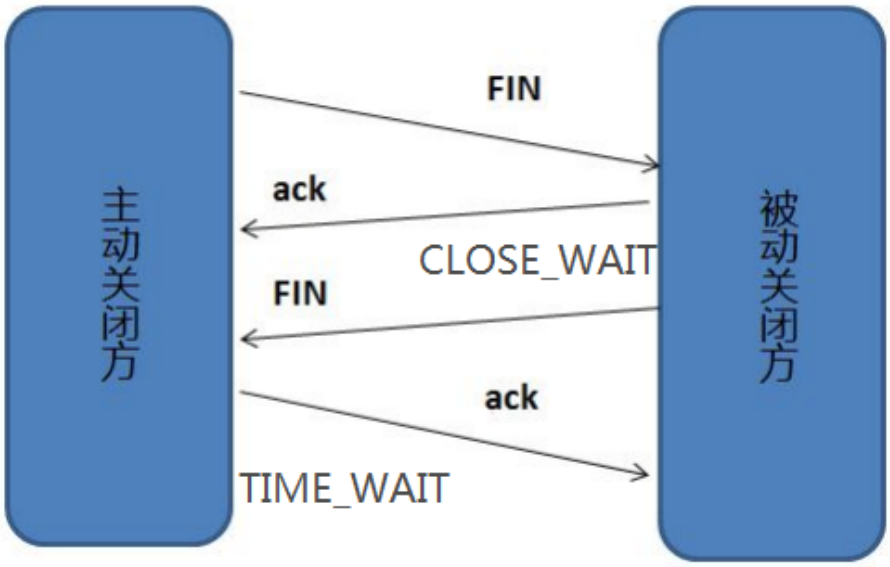
- 三次握手



- 四次挥手



4、TIME_WAIT 与 CLOSE_WAIT

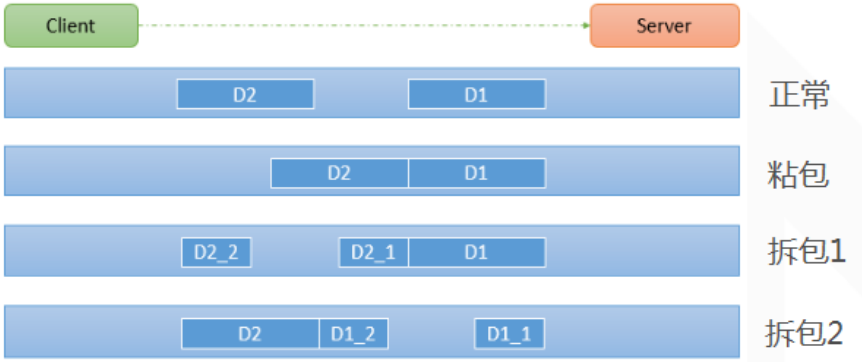


5、TCP 滑动窗口

TCP 流量控制，主要使用**滑动窗口协议**，滑动窗口是接受数据端使用的窗口大小，用来告诉发送端接收端的缓存大小，以此可以控制发送端发送数据的大小，从而达到流量控制的目的。这个窗口大小就是我们一次传输几个数据。对所有数据帧按顺序赋予编号，发送方在发送过程中始终保持着一个发送窗口，只有落在发送窗口内的帧才允许被发送；同时接收方也维持着一个接收窗口，只有落在接收窗口内的帧才允许接收。

6、TCP 粘包和拆包

- 现象



- 产生原因 1、要发送的数据大于 TCP 发送缓冲区剩余空间大小，将会发生拆包。 2、待发送数据大于 MSS（最大报文长度），TCP 在传输前将进行拆包。 3、要发送的数据小于 TCP 发送缓冲区的大小，TCP ���多次写入缓冲区的数据一次发送出去，将会发生粘包。 4、接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。
- 解决方式 1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。 2、发送端将每个数据包封装为固定长度（不够的可以通过补 0 填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。 3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

八、MQ 消息队列

1、场景作用

削峰填谷，异步解耦。

2、如何保证消息不被重复消费呢?

这个问题可以换个思路，保证消息重复消费，其实是保证程序的幂等性。无论消息如何重复，程序运行的结果是一致的。比如消费消息后做数据库插入操作，为了防止消息重复消费，可以在插入前先查询一下有没有对应的数据。

3、怎么保证从消息队列里拿到的数据按顺序执行?

消费端在接收到消息后放入内存队列，然后对队列中的消息进行有序消费。

4、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决?

消息过期失效问题，如果消息一段时间不消费，导致过期失效了，消息就丢失了，只能重新查出丢失的消息，重新发送。 再来说消息积压的问题：（思路是快速消费掉积压的消息）

- 首先排查消费端问题，恢复消费端正常消费速度。
- 然后着手处理队列中的积压消息。
    - 停掉现有的 consumer。
    - 新建一个 topic，设置之前 10 倍的 partation，之前 10 倍的队列。
    - 写一个分发程序，将积压的消息均匀的轮询写入这些队列。
    - 然后临时用 10 倍的机器部署 consumer，每一批 consumer 消费 1 个临时的队列。
    - 消费完毕后，恢复原有架构。

消息队列满了：只能边接收边丢弃，然后重新补回丢失的消息，再做消费。

4、如何保证消息的可靠性传输（如何处理消息丢失的问题）？

kafka 为例：

- 消费者丢了数据： 每次消息消费后，由自动提交 offset 改为手动提交 offset 。
- kafka 丢了消息： 比较常见的一个场景，就是 kafka 某个 broker 宕机，然后重新选举 partition 的 leader 时。要是此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后大家选举某个 follower 成为 leader 之后，不就少了一些数据。
    - 给 topic 设置**replication.factor**参数：这个值必须大于 1，要求每个 partition 必须有至少两个副本。
    - 在 kafka 服务端设置**min.insync.replicas**参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower。
    - 在 producer 端设置**acks=all**：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。
    - 在 producer 端设置**retries=MAX**（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里。
- 生产者丢了消息： 如果按照上述的思路设置了 ack=all，一定不会丢，要求是，你的 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

九、Redis

1、数据类型

- String

> 常用命令： set,get,decr,incr,mget 等。

- Hash

> 常用命令： hget,hset,hgetall 等

- List

> 常用命令： lpush,rpush,lpop,rpop,lrange 等

可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询。

- Set

> 常用命令： sadd,spop,smembers,sunion 等

- Sort Set

> 常用命令： zadd,zrange,zrem,zcard 等

2、Redis 如何实现 key 的过期删除?

> 定期删除和惰性删除的形式。

- 定期删除 Redis 每隔一段时间从设置过期时间的 key 集合中，随机抽取一些 key ，检查是否过期，如果已经过期做删除处理。
- 惰性删除 Redis 在 key 被访问的时候检查 key 是否过期，如果过期则删除。

3、Redis 的持久化机制

> 数据快照（RDB）+ 修改数据语句文件（AOF）

4、如何解决 Redis 缓存雪崩和缓存穿透?

- 缓存雪崩 缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。
    - 解决方式
        - 事前：保证 Redis 集群的稳定性，发现机器宕机尽快补上，设置合适的内存淘汰策略。
        - 事中：本地缓存 + 限流降级，避免大量请求落在数据库上。
        - 事后：利用 Redis 持久化机制尽快恢复缓存。
- 缓存穿透 一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。
    - 解决方式 将不存在的数据列举到一个足够大的 map 上，这样遭到攻击时，直接拦截 map 中的请求，请求到数据库上面。或是把不存在的也做缓存，值为 null ，设置过期时间。

5、如何使用 Redis 实现消息队列?

Redis 实现消息队列依赖于 Redis 集群的稳定性，通常不建议使用。

- Redis 自带发布订阅功能，基于 publish 和 subscribe 命令。
- 使用 List 存储消息，lpush，rpop 分别发送接收消息。

十、Nginx

> Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。
> Nginx 主要提供反向代 理、负载均衡、动静分离(静态资源服务)等服务。

1、正向代理和反向代理

- 正向代理 代理客户端访问服务器。典型：VPN
- 反向代理 代替服务器接收客户端请求，然后转发给服务器，服务器接收请求并将处理的结果通过代理服务器转发给客户端。

2、负载均衡

> 将请求分摊到多台机器上去，高并发，增加吞吐量。

- 负载均衡算法
    - 权重轮询
    - fair
    - ip_hash
    - url_hash

3、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，我们就可以根据静态资源的特点将其做缓存操作，这就是网站静态化处理的核心思路。

4、Nginx 四个组成部分

- Nginx 二进制可执行文件：由各模块源码编译出一个文件
- Nginx.conf 配置文件：控制 Nginx 行为
- acess.log 访问日志： 记录每一条 HTTP 请求信息
- error.log 错误日志：定位问题