**JAVA:**

1：集合框架：

**Collection** 接口的接口 对象的集合（单列集合）   
├——-**List** 接口：元素按进入先后有序保存，可重复   
│—————-├ **LinkedList**  底层链表，没有同步，线程不安全 ，查询慢，增删快

│—————-├ **ArrayList**  底层数组， 随机访问，没有同步，线程不安全 ，查询快，增删慢，  
│—————-└ **Vector** 接口实现类 ，矢量队列，底层数组， 同步， 线程安全   
│ ———————-└ **Stack** 是Vector类的实现类   
└——-**Set** 接口： 仅接收一次，不可重复，并做内部排序   
├—————-└**HashSet** 使用hash表（数组）存储元素，其实就是hashmap的key部分   
│————————└ **LinkedHashSet** 链表维护元素的插入次序   
└ —————-**TreeSet** 底层实现为二叉树，元素排好序

**Map** 接口 键值对的集合 （双列集合）   
├———**Hashtable** 接口实现类， 同步， 线程安全 ，使用数组加链表的方式存储。锁住整个Hashtable导致效率低，key与value都不可为null。初始大小11，扩容2倍+1  
├———**HashMap** 接口实现类 ，没有同步， 线程不安全- ，初始大小16，扩容因子0.75，扩容翻倍，使用数组加链表的方式存储，链表达到8转为红黑树，key与value可为null，null键存放在底层数组的0号位置  
│—————–├ **LinkedHashMap** 双向链表和哈希表实现 ，key和value可以为null  
│—————–└ **ConcurrentHashMap** 线程安全的hashmap，分段加锁提高效率，key与value都不可为null  
├ ——–**TreeMap** 红黑树对所有的key进行排序   
└———**IdentifyHashMap**

2：锁

乐观锁/悲观锁：

乐观锁：假设最好的情况，读取数据时默认数据未被修改，故不加锁，更新数据时会判断在此期间是否数据被修改，多使用版本号等机制判断数据是否被修改（不用结果对比主要解决ABA问题），适用于多读场景，例CAS（Compare and Swap）自旋操作；

悲观锁：假设最差的情况，每次读取数据都默认会被修改，故每次读取，更新都会加锁，阻塞其他人同时获取或者更新数据。例synchronized；

独享锁/共享锁（具体表现：互斥锁/读写锁）：

独享锁：一次只能被一个线程持有，例：读写锁的写锁， ReentrantLock，synchronized

共享锁：可以被多个线程持有,例：读写锁的读锁

可重入锁ReentrantLock：

又名递归锁，同一个线程在外层方法获取锁的时候，内层方法也会自动获取锁，可以一定程度避免死锁。Synchronized也是可重入锁。通过AQS数据结构来实现线程调度。

原理：每一个锁关联一个线程持有者和计数器，当计数器为 0 时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某一线程请求成功后，JVM会记下锁的持有线程，并且将计数器置为 1；此时其它线程请求该锁，则必须等待；而该持有锁的线程如果再次请求这个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数器会递减，如果计数器为 0，则释放该锁

公平锁/非公平锁：

公平锁：多个线程按照申请锁的顺序来获取锁。

非公平锁：不是按照申请顺序来获取锁。吞吐量大于公平锁

分段锁：

ConcurrentHashMap使用的锁机制。细化锁的粒度，提高效率

锁的四种状态：

无锁/偏向锁/轻量级锁/重量级锁：

后面三种锁的状态，针对于Synchronized，提高其效率。

偏向锁：一段同步代码一直被一个线程所访问，该线程会自动获取锁，降低获取锁的代价。

轻量级锁：当锁为偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的方式尝试获取锁，不会阻塞，提高性能。

重量级锁：当锁为轻量级锁的时候，另一个线程自旋到一定次数还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁，重量级锁会让其他申请的线程进入阻塞，性能降低，

自旋锁：

指尝试获取锁的线程不会立即阻塞，采用循环的方式尝试获取锁，好处减少线程上下文切换的消耗，缺点浪费CPU

AQS：

AbstractQueuedSynchronized 抽象队列式的同步器，

AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch

底层是CAS+volatile实现的，state是volatile修饰，更新state除了有setState方法外还有CAS。

除了加锁和原子操作外，final也可以起到线程安全的作用

3：isAssignableFrom()方法与instanceof关键字的区别

isAssignableFrom()方法是从类继承的角度去判断，instanceof关键字是从实例继承的角度去判断。

isAssignableFrom()方法是判断是否为某个类的父类，instanceof关键字是判断是否某个类的子类。

父类.class.isAssignableFrom(子类.class)

子类实例 instanceof 父类类型

4：设计模式：

创建型模式(共六种)：简单工厂模式（Calendar时间管理类），工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式(共七种)：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式（线程池）

行为型模式(共十一种)：策略模式、模板方法模式、观察者模式（wait/notify，AIO）、迭代子模式、责任链模式（netty中的ChannelPipeline）、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

5：JVM：

内存划分（运行时数据区）：

虚拟机栈，堆，方法区，程序计数器（寄存器），本地方法栈五个部分。

堆和方法区是线程共享，生命周期与虚拟机相同;

其他的则为线程私有，随线程而生，随线程而灭。

虚拟机栈：每执行一个方法就会往栈中压入一个元素，递归过深会导致栈空间不足。

本地方法栈：存放局部变量等。和虚拟机栈类似，只不过它是用来表示执行本地方法的

线程计数器：记录程序状态

堆：存放new出来的对象

方法区：存放类信息，常量，静态变量等

垃圾判定的方法：

引用计数法(无法检测循环引用)；

可达性分析（从GC roots出发不能到达的则是可回收对象），可作为GC roots的包括常量，全局变量，静态变量以及方法内的局部变量。

垃圾回收算法：

1标记-清理：第一步利用可达性遍历内存，标记存活对象和垃圾对象，第二步将垃圾对象清空，简单方便但容易产生内存碎片。

2标记-整理：第一步利用可达性遍历内存，标记存活对象和垃圾对象，第二步把所有存活对象堆到同一个地方，就不会产生内存碎片，适合存活对象多，垃圾少的情况，但效率低。

3复制-回收：将内存划分大小相等的两块，每次只使用其中一块，当这一块用完了，就将还活着的对象复制到另一块上，然后再把使用过的内存空间一次性清理掉。简单，速度快，不会产生碎片，但内存利用率低。

Java堆的垃圾回收：分代回收算法

堆分为新生代，老年代，永久代。新生代与老年代比例1:2

新生代：存活对象少，垃圾多，使用复制-回收机制。Minor GC

分为三个部分：Eden(伊甸园区)，S1(幸存者1区)，S2。比例8：1：1

老年代：存活对象多，垃圾少，使用标记-整理机制。Full GC，对象在新生代复制15次任未被回收，则进入老年代。

永久代：永久存在的对象。

一般大对象，数组等大于某一个设定的值直接进入老年代

jdk9到jdk13：使用G1垃圾回收机制，G1为大内存多内核处理器设计，为应用提供停顿时间和吞吐量的最佳平衡。

Java对象分配流程：

尝试栈上分配🡪成功🡪栈上分配

🡪失败🡪尝试TLAB分配🡪成功🡪TLAB分配

🡪失败🡪是否直接进入老年代🡪可以🡪老年代分配

🡪不可以🡪伊甸区分配

栈上分配：确定一个对象的作用域不会逃逸出方法之外，可以将这个对象分配在栈上，这样大量的对象就会随着方法的结束而自动销毁了，无须通过垃圾收集器回收，可以减小垃圾收集器的负载。

TLAB的全称是Thread Local Allocation Buffer，即线程本地分配缓存区，这是一个线程专用的内存分配区域。 JVM使用TLAB来避免多线程冲突，在给对象分配内存时，每个线程使用自己的TLAB，这样可以避免线程同步，提高了对象分配的效率。

Fork/join：将一个大的任务拆分成多个子任务进行并行处理，最后将子任务结果合并成最后的计算结果。

6：解决hash 冲突的主要方法

开放定址法:（线性探测再散列，二次探测再散列，伪随机探测再散列, 一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入 ）。  
再哈希法:  
链地址法:(Java hashmap就是这么做的)  
建立一个公共溢出区:(将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表)

7:1T数据使用32G内存排序（外部排序，归并排序）：

先分割为40块，每块25G(留一部分系统空间)

顺序将这40块数据读入内存，使用快速排序算法排序，排好序的数据存放回磁盘

从40个数据块中分别读取25G/40 =0.625G入内存

将临时数据存储在2G大小的输出缓冲区,

执行40路归并排序，将最终排好序的数据写入输出缓冲区，

缓冲区写满2GB时，写入硬盘上最终文件，并清空输出缓冲区

当40个0.625G的输入缓存区任何一个处理完毕，继续读取该数据块下一个0.625G数据

直到所有数据处理完成

8：类加载的过程（对象创建的过程）：

加载，验证，准备，解析，初始化，使用，卸载。

类加载的双亲委派模型过程

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

9：线程池的七个参数：

corePoolSize ：核心线程池大小，CPU密集型：核数+1，IO密集型：核数\*2+1

maximumPoolSize ：最大线程池数

keepAliveTime ：空闲线程存活时间

unit ：存活时间的时间单位

workQueue ：工作队列

threadFactory ：线程工厂

handler ：拒绝策略

四种工作队列：

ArrayBlockingQueue：基于数组的有界阻塞队列，按FIFO（先进先出）排序，当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

LinkedBlockingQuene：基于链表的无界阻塞队列（其实最大容量为Interger.MAX），按照FIFO排序。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。

SynchronousQuene：一个不缓存任务的阻塞队列，生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。

PriorityBlockingQueue：具有优先级的无界阻塞队列，优先级通过参数Comparator实现

四种拒绝策略：

CallerRunsPolicy：在调用者线程中直接执行被拒绝任务的run方法，除非线程池已经shutdown，则直接抛弃任务

AbortPolicy：该策略下，直接丢弃任务，并抛出RejectedExecutionException异常

DiscardPolicy：该策略下，直接丢弃任务，什么都不做。

DiscardOldestPolicy：该策略下，抛弃最早进入队列的那个任务，然后尝试把这次拒绝的任务放入队列。

10：java定义的8种内存操作：

lock(锁定)：作用于主内存，它把一个变量标记为一条线程独占状态；

unlock(解锁)：作用于主内存，它将一个处于锁定状态的变量释放出来，释放后的变量才能够被其他线程锁定；

read(读取)：作用于主内存，它把变量值从主内存传送到线程的工作内存中，以便随后的load动作使用；

load(载入)：作用于工作内存，它把read操作的值放入工作内存中的变量副本中；

use(使用)：作用于工作内存，它把工作内存中的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量的指令时候，将会执行这个动作；

assign(赋值)：作用于工作内存，它把从执行引擎获取的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的指令时候，执行该操作；

store(存储)：作用于工作内存，它把工作内存中的一个变量传送给主内存中，以备随后的write操作使用；

write(写入)：作用于主内存，它把store传送值放到主内存中的变量中。

11：cookie和session的区别：

cookie机制采用的是在客户端保持状态的方案，而session机制采用的是在服务器端保持状态的方案。

12：Hashmap底层是数组+链表，链表在长度为8的时候变为红黑树，在6的时候从树退回链表。使用红黑树而不是其他AVL树是因为红黑树插入速度更快，且CurrentHashMap是加锁的，插入时间过长就会导致等待时间更长。

13：priorityQueue底层是堆，堆的底层是完全二叉树

通过数组实现，默认升序排列，初始化大小11，

扩容方式：元素数量小于64,原容量\*2+2；否则变为原来1.5倍

14: hashmap的容量为什么是2的次幂：

HashMap的初始容量是2的n次幂，扩容也是2倍的形式进行扩容，是因为容量是2的n次幂，可以使得添加的元素均匀分布在HashMap中的数组上，减少hash碰撞，避免形成链表的结构，从而提高查询效率！

15：CopyOnWriteArrayList

这是一个ArrayList的线程安全的变体，其原理大概可以通俗的理解为:初始化的时候只有一个容器，很常一段时间，这个容器数据、数量等没有发生变化的时候，大家(多个线程)，都是读取(假设这段时间里只发生读取的操作)同一个容器中的数据，所以这样大家读到的数据都是唯一、一致、安全的，但是后来有人往里面增加了一个数据，这个时候CopyOnWriteArrayList 底层实现添加的原理是先copy出一个容器(可以简称副本)，再往新的容器里添加这个新的数据，最后把新的容器的引用地址赋值给了之前那个旧的的容器地址，但是在添加这个数据的期间，其他线程如果要去读取数据，仍然是读取到旧的容器里的数据。

优点：解决并发问题

缺点：内存占用大，不能保证数据实时一致性

16：迭代器Iterator执行快速失败

快速失败（fail-fast）：用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception

安全失败（fail-safe）：采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。缺点：不能保证遍历数据的实时性

java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改

17：ReentrantLock，synchronized的异同：

两者的共同点：   
1. 都是用来协调多线程对共享对象、变量的访问   
2. 都是可重入锁，同一线程可以多次获得同一个锁   
3. 都保证了可见性和互斥性   
两者的不同点：   
1. ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁   
2. ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的  
不可用性提供了更高的灵活性   
3. ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的   
4. ReentrantLock 可以实现公平锁   
5. ReentrantLock 通过 Condition 可以绑定多个条件   
6. 底层实现不一样， synchronized 是同步阻塞，使用的是悲观并发策略，lock 是同步非阻  
塞，采用的是乐观并发策略   
7. Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言  
实现。   
8. synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；  
而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象，  
因此使用 Lock 时需要在 finally 块中释放锁。   
9. Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，  
等待的线程会一直等待下去，不能够响应中断。   
10. 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。   
11. Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等

18：接口和抽象类：

Java不支持多继承，但可以实现多个接口。

抽象类是用来捕捉子类的通用特性的 。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板

接口是抽象方法的集合。接口只是一种形式，接口自身不能做任何事情。



19：消息队列MQ：

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题。实现高性能，高可用，可伸缩和最终一致性架构。是大型分布式系统不可缺少的中间件。

目前在生产环境，使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ等

20：IO的几种方式：BIO（同步阻塞），NIO（同步非阻塞），AIO（异步非阻塞）

BIO：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

NIO: 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO的特点：是面向流的，NIO是面向缓冲区的；BIO的各种流是阻塞的。而NIO是非阻塞的；BIO的Stream是单向的，而NIO的channel是双向的。

NIO的特点：事件驱动模型、单线程处理多任务、非阻塞I/O，I/O读写不再阻塞，而是返回0、基于block的传输比基于流的传输更高效、更高级的IO函数zero-copy、IO多路复用大大提高了Java网络应用的可伸缩性和实用性。基于Reactor线程模型。

21：Executors创建的四种线程池：

newCachedThreadPool：创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。内部用SynchronousQueue的阻塞队列。

newFixedThreadPool：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

newScheduledThreadPool： 创建一个定长线程池，支持定时及周期性任务执行。

newSingleThreadExecutor：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

22：阻塞队列BlockingQueue：

ArrayBlockingQueue：基于数组的阻塞队列实现

LinkedBlockingQueue：基于链表的阻塞队列

DelayQueue：只有当其指定的延迟时间到了，才能够从队列中获取到该元素

PriorityBlockingQueue：基于优先级的阻塞队列

SynchronousQueue：无缓冲的等待队列

23：当变量的值在（-128,127）之间（也就是可以用一个字节所能表示的int值）时才会被放入常量池，否则会自动装箱生成普通Integer对象

**SPRING**

1：IOC底层实现原理：xml配置文件+dom4j解决xml+工厂和单例设计模式+反射

2：AOP底层实现原理：动态代理+CGLIB动态字节码生成（可以为没有实现任何接口的类进行扩展）

3：@Autowired

可用于注入参数，也可用于构造方法，还可自动注入map（key是类名，且首字母小写），list等集合中。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />开启注解装配；

4：事务传播行为

1.PROPAGATION\_REQUIRED(默认实现)：当前没有事务则新建事务，有则加入当前事务

2.PROPAGATION\_SUPPORTS：支持当前事务，如果当前没有事务则以非事务方式执行

3.PROPAGATION\_MANDATORY：使用当前事务，如果没有则抛出异常

4.PROPAGATION\_\_REQUIRES\_NEW：新建事务，如果当前有事务则把当前事务挂起

5.PROPAGATION\_NOT\_SUPPORIED：以非事务的方式执行，如果当前有事务则把当前事务挂起

6.PROPAGATION\_NEVER：以非事务的方式执行，如果当前有事务则抛出异常

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，则执行1

5：spring的优点：

低侵入式设计，代码的污染极低。

依赖注入机制将对象之间的依赖关系交由框架处理，降低组件的耦合性。

AOP技术支持将一些通用任务，如安全，事务，日志，权限等进行集中式管理，更好的复用。

对主流的应用框架提供了集成支持。

6：BeanFactory和ApplicationContext：

是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

7：Bean的生命周期：

（1）实例化Bean：

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息以及通过BeanWrapper提供的设置属性的接口完成依赖注入。

（3）处理Aware接口：

Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean

（4）BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

（5）InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（7）DisposableBean：

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（8）destroy-method：

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

8：Bean的作用域：

singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

prototype：为每一个bean请求提供一个实例。

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

session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

9：Bean的线程安全：

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

10：spring Bean的自动装配：

装配：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给各个对象。Xml配置里有5种自动装配：

（1）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。

（2）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。

（3）byType：通过参数的数据类型进行自动装配。

（4）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。

（5）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。

11：@Autowired和@Resource之间的区别

@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），如果查询的结果不止一个，那么@Autowired会根据名称来查找；

@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

12：spring中使用的设计模式：

1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

13：spring事务：

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

spring支持编程式事务管理和声明式事务管理两种方式：

编程式事务管理使用TransactionTemplate。

声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

14：AOP的5中通知类型

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。 环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

15：可以向spring中注入一个null或者空字符串。

**Mybatis**

1：#{}和${}的区别是什么：

#{}是预编译处理，是sql的参数占位符，${}是Properties（配置）文件中的变量占位符，属于静态文本替换；

Mybatis在处理#{}时，会将sql中的#{}替换为占位符?，调用PreparedStatement的set方法来赋值；Mybatis在处理${}时，就是把${}直接替换成变量的值，纯粹的string替换。

使用#{}可以有效的防止SQL注入，提高系统安全性

SQL注入，就是通过把SQL命令插入到Web[表单](https://baike.baidu.com/item/%E8%A1%A8%E5%8D%95/5380322)提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

2：Dao接口的工作原理：

Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MapperStatement；

Mapper 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Mapper接口生成代理对象proxy，代理对象会拦截接口方法，转而执行MapperStatement所代表的sql，然后将sql执行结果返回。

因为是使用 全限名+方法名 的保存和寻找策略；所以Mapper接口里的方法不能重载。

3：分页插件pageHelper原理

PageHelper首先将前端传递的参数保存到page这个对象中，接着将page的副本存放入ThreadLoacl中，这样可以保证分页的时候，参数互不影响，接着利用了mybatis提供的拦截器，取得ThreadLocal的值，重新拼装分页SQL，完成分页。属于物理分页

4：Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式：

第一种是使用<resultMap>标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

5：Mybatis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；

6：Mybatis是否支持延迟加载？如果支持，它的实现原理是什么

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

7：MyBatis实现一对一和一对多有几种方式?具体怎么操作的？

都有联合查询和嵌套查询两种方法。

联合查询是几个表联合查询,只查询一次, 通过在resultMap里面配置association节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的 外键id，去再另外一个表里面查询数据,也是通过association配置，但另外一个表的查询通过select属性配置。

8：Mybatis的缓存机制：

一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/> 属性；

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

9：Mybatis接口绑定：

将接口里面的方法和SQL语句绑定，直接调用接口方法就行，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。

两种实现方式：一种是通过注解绑定，就是在接口的方法上面加上 @Select、@Update等注解，里面包含Sql语句来绑定；另外一种就是通过xml里面写SQL来绑定, 在这种情况下,要指定xml映射文件里面的namespace必须为接口的全路径名。当Sql语句比较简单时候,用注解绑定, 当SQL语句比较复杂时候,用xml绑定,一般用xml绑定的比较多。

10：Mybatis的mapper接口调用的要求：

①  Mapper接口方法名和mapper.xml中定义的每个sql的id相同；

②  Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同；

③  Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同；

④  Mapper.xml文件中的namespace即是mapper接口的类路径。

11：Mybatis批处理：

使用BatchExecutor完成批处理；

12：Mybatis的Executor执行器：

SimpleExecutor：默认，每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

CachingExecutor: 启用于二级缓存时的执行器，先从缓存中提取数据，数据缓存中没有数据时才从数据库里面提取数据。

Executor的这些特点，都严格限制在SqlSession生命周期范围内。可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数。

13：Mybatis解析xml映射文件:

Mybatis解析xml映射文件是按照顺序解析的，但是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。

**MYSQL**

1：索引不适用的条件：

索引列上有函数

使用了不等号

不满足最左原则

更新非常频繁的字段不适合创建索引

唯一性太差的字段不适合单独创建索引

不会出现在where子句中的字段不该创建索引

表记录太少不适合创建索引

当修改性能远大于检索性能时不应该创建索引

2：优化口诀：sql优化：尽量全值匹配、最佳左前缀法则、不在索引列上做任何操作、范围条件放最后、覆盖索引尽量用、不等于要甚用、Null/Not 有影响、Like查询要当心、字符类型加引号、OR改UNION效率高

3：索引分类

单列索引：

普通索引：没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一 点。

唯一索引：索引列中的值必须是唯一的，但是允许为空值。

主键索引：是一种特殊的唯一索引，不允许有空值。（主键约束，就是一个主键索引）。

主键索引和唯一索引的区别：

主键是一种约束，唯一索引是一种索引，本质不同

主键创建后就会包含一个唯一索引，唯一索引并不一定就是主键索引

主键索引不允许空值，唯一索引允许空值

一个表只能创建一个主键索引，但可以创建多个唯一索引

主键可以被其他表引用为外键，唯一索引不行

组合索引：在多个字段组合上创建的索引，使用时遵循最左原则（123,12,1才符合最左原则）

全文索引：就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行，有在**MyISAM**引擎上才能使用，只能在**CHAR,VARCHAR,TEXT**类型字段上使用全文索引。

聚集索引（聚簇索引）：未加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐， 跟我认知中的「表」很接近。如果给表上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，也就是上面说的「平衡树」结构，换句话说，就是整个表就变成了一个索引，就是所谓的「聚集索引」；这就是为什么一个表只能有一个主键， 一个表只能有一个「聚集索引」，因为主键的作用就是把「表」的数据格式转换成「索引（平衡树）」的格式放置。

非聚集索引， 也就是我们平时经常提起和使用的常规索引。

非聚集索引和聚集索引的区别在于， 通过聚集索引可以查到需要查找的数据， 而通过非聚集索引可以查到记录对应的主键值 ， 再使用主键的值通过聚集索引查找到需要的数据。

不管以任何方式查询表， 最终都会利用主键通过聚集索引来定位到数据， 聚集索引（主键）是通往真实数据所在的唯一路径。

有一种例外可以不使用聚集索引就能查询出所需要的数据， 这种非主流的方法 称之为「覆盖索引」查询， 也就是平时所说的复合索引或者多字段索引查询。当为字段建立索引以后， 字段中的内容会被同步到索引之中， 如果为一个索引指定两个字段， 那么这个两个字段的内容都会被同步至索引之中。当查找的字段在这复合索引里面，所以不再需要通过主键id值来查找数据。

4：一般不推荐使用like，like”%aaa%”不会使用索引，而like”aaa%”会使用索引;

5：BTree索引：树结构索引故内部其实是有序的，适合用于范围查询，例如<,>,order by等

Hash索引：检索效率非常高但仅满足= ,in,<=>查询,不能使用范围查询，不能利用组合索引查询，很适合做缓存。

6：事务的四个特征：原子性，一致性，隔离性，持续性

7：四种隔离级别

读未提交Read Uncommitted：所有事务都可以看到其他未提交事务的执行结果，本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

读提交Read Committed：一个事务只能看见已经提交事务所做的改变。因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

可重读epeatable Read：mysql默认的隔离级别，确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题

可串行化Serializable：最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

8：并行读取容易发生的问题：

脏读：读到了其他事务未提交的数据，未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。读到了并一定最终存在的数据，这就是脏读。

不可重复读：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：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。



9：数据库引擎：

MYISAM：拥有较高的插入查询速度，不支持事务。

InnoDB：支持事务，行级锁。保证并行更新数据的安全性。

Memory：将表中数据存储在内存里，适合只是临时存放，数据量不大，并且不需要较高安全性的数据。适合作为临时表，存放查询的中间结果

Archive：支持高并发的插入，本身非事务安全，很适合存储归档数据，如日志

MyISAM和InnoDB的区别：

MyISAM是非事务安全的，InnoDB是事务安全的

MyISAM粒度是表级的，InnoDB支持行级锁

MyISAM支持全文类型搜索，InnoDB不支持全文搜索

MyISAM相对简单，效率更优，InnoDB数据更安全

MyISAM适合大量select操作，InnoDB适合大量insert和update操作。

MyISAM内部维持了一个计数器，故select count（\*）可以直接调取计数器的值速度最快

10：InnoDB底层数据结构

InnoDB索引数据结构为B+Tree树。

B+数只有叶子节点存放数据，其他几点用来索引，而B-树是每个索引节点都会有Data域。不用红黑树是为了防止红黑树深度过大造成磁盘IO读写过于频繁。

11：原子性和持久性怎么保证：

原子性：undo log：记录了回滚操作的日志，如果要撤销，按照undo log的回滚日志执行一遍就可以了（保证了原子性）

持久性：redo log：不是每次都写入磁盘，而是定期通过redo log (redo log一部分在内存中，一部分在磁盘上)把数据刷入磁盘这样即便断电后，redo log因为是写入硬盘，所以其数据不会丢失，重启mysql还是可以恢复。

12：怎么解决幻读：

 多版本并发控制（MVCC）（快照读/一致性读）：将历史数据存一份快照，所以其他事务增加与删除数据，对于当前事务来说是不可见的。

间隙锁（加在索引之间的锁）：将当前数据行与上一条数据和下一条数据之间的间隙锁定，保证此范围内读取的数据是一致的。

13：数据库事务的四个特征：

原子性:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
一致性:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。  
隔离性:隔离状态执行事务，使它们好像是[系统](http://www.2cto.com/os/)在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。  
持久性:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

14：drop,delete与truncate的区别：

1：一般而言，drop > truncate > delete；

 2：表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

3：drop直接删掉表 truncate删除表中数据，再插入时自增长id又从1开始 delete删除表中数据，可以加where字句

4： DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作；truncate TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的，执行速度快；

15：数据库读取方式：

磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

16：sql语句中的连接种类：

内连接：inner join …on …;两张表的交集；

左连接（全称：左外连接）：left join…on或者left outer join …on;

左表的记录全部展示出来，右表显示符合条件的，没有匹配行的均为null；

右连接（全称：右外连接）：right join…on或者right outer join …on;

右表的记录全部展示出来，左表显示符合条件的，没有匹配行的均为null；

全连接（交叉连接）：cross join：没有 WHERE 子句的交叉联接将产生联接所涉及的表的笛卡尔积；select \* from table1 cross join table2等价select \* from table1,table2；

17：varchar是一种可变长度的类型，char是固定长度的类型。

18：列出所有正在执行的线程： show processlist

19：utf8字符集升级至utf8\_mb4可支持emoji表情。

20：数据库的主从复制：

一个服务器作为主服务器，一个或多个服务器作为从服务器，主服务器将更新写到二进制日志，当一个从服务器连接到主服务器时，通知主服务器读取日志，接收从那时起发生的所有更新。解决：数据分布，负载平衡，备份，高可用性和容错性。

21：存储过程：

存储过程是存储在数据库目录中的一段声明性SQL语句，优点有：

通常存储过程有助于提高应用程序的性能

存储过程有助于减少应用程序和数据库服务器之间的流量

存储的程序对任何应用程序都是可重用的和透明的

存储的程序是安全的

比单纯的SQL语句要快

CREATE PROCEDURE GetAll()

BEGIN

Select \* from table;

END;

CALL GetAll (); 调用存储过程

22：视图：

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。

23：游标：

是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

24：in和exists：

in()适合B表比A表数据小的情况

exists()适合B表比A表数据大的情况

当A表数据与B表数据一样大时,in与exists效率差不多,可任选一个使用.

select \* from A where id in(select id from B)

Mysql的In查询会外表使用全表扫描，内表使用索引查询

**Docker**

1：Docker相关的本地资源存放在/var/lib/docker/目录下，其中container目录存放容器信息，graph目录存放镜像信息，aufs目录下存放具体的镜像底层文件。

2：查看正在运行的容器：docker ps

查看所有容器：docker ps –a

3：停止正在运行的容器：docker kill -9 \*，强行终止；docker stop \*的话，首先给容器发送一个TERM信号，让容器做一些退出前必须的保护性、安全性操作，然后让容器自动停止运行，如果在一段时间内，容器还是没有停止，再进行kill -9，强行终止。

4：删除停止的容器：docker rm \*

5：删除镜像：docker rmi \*

6：查看镜像：docker images

7：查看容器的日志信息：docker logs \*；

8：ocker的配置文件位置：Ubuntu系统下Docker的配置文件是/etc/default/docker，CentOS系统配置文件存放在/etc/sysconfig/docker

9：docker可以运行在非linux(window和macos)上

10：进入容器的方法：docker attach命令和docker exec命令

11：容器与主机之间数据拷贝：docker cp \* \*

**HTTP/TCP**

1：TCP三次握手和四次挥手

三次握手：

第一次：客户端发送初始序号x和syn=1请求标志

第二次：服务器发送请求标志syn，发送确认标志ACK，发送自己的序号seq=y，发送客户端的确认序号ack=x+1

第三次：客户端发送ACK确认号，发送自己的序号seq=x+1，

发送对方的确认号ack=y+1

四次挥手：

第一次挥手：客户端发出释放FIN=1，自己序列号seq=u，进入FIN-WAIT-1状态

第二次挥手：服务器收到客户端的后，发出ACK=1确认标志和客户端的确认号ack=u+1，自己的序列号seq=v，进入CLOSE-WAIT状态

第三次挥手：客户端收到服务器确认结果后，进入FIN-WAIT-2状态。此时服务器发送释放FIN=1信号，确认标志ACK=1，确认序号ack=u+1，自己序号seq=w，服务器进入LAST-ACK（最后确认态）

第四次挥手：客户端收到回复后，发送确认ACK=1，ack=w+1，自己的seq=u+1，客户端进入TIME-WAIT（时间等待）。客户端经过2个最长报文段寿命后，客户端CLOSE；服务器收到确认后，立刻进入CLOSE状态

2：HTTP和TCP的区别

HTTP协议是建立在TCP协议之上的一种应用

TCP是传输层协议，就是单纯建立连接 ,主要解决数据如何在网络中传输，不涉及任何我们需要请求的实际数据，简单的传输

http是用来收发数据，即实际应用上来的, 主要是解决如何包装数据。

HTTP协议中的数据是利用TCP协议传输的，所以支持HTTP也就一定支持TCP.

3：netty对http协议进行解析，提供实现，Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈。

4：TCP的拥堵控制

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络性能就要变坏，这种情况就叫做网络拥塞。若出现拥塞而不进行控制，整个网络的吞吐量将随输入负荷的增大而下降。

四种拥堵控制算法：

1.慢开始  
2.拥塞控制  
3.快重传  
4.快恢复

5：TCP/IP四层模型：

链路层（数据链路层/网络接口层）：包括操作系统中的设备驱动程序、计算机中对应的网络接口卡

网络层（互联网层）：处理分组在网络中的活动，比如分组的选路。

运输层：主要为两台主机上的应用提供端到端的通信。

应用层：负责处理特定的应用程序细节。

6：TCP和UDP的区别：

TCP是传输控制协议，提供的是面向连接的，可靠地字节流服务

UDP是用户数据报协议，是无连接的。因为无连接，而且没有超时重发机制，所以UDP传输速度很快。

TCP保证数据按序到达，提供流量控制和拥塞控制，在网络拥堵的时候会减慢发送字节数，而UDP不管网络是否拥堵。

TCP是连接的，所以服务是一对一服务，而UDP可以1对1，也可以1对多（多播），也可以多对多。

7：HTTP1.0和HTTP1.1：

HTTP1.0每请求一个文档就要建立TCP连接，有几次握手的时间花销，如果一个主页上有很多链接的对象需要依次进行连接，每次连接下载都要消耗这些开销。

HTTP1.1采用持续连接。所谓持续连接就是服务器在发送响应后仍然在一段时间内保持这条连接。使得后序的请求和响应报文都在这条连接上进行。

8：TCP粘包/拆包的原因及解决方法

TCP是以流的方式来处理数据，一个完整的包可能会被TCP拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；

解决方法

消息定长：FixedLengthFrameDecoder类

包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder或自定义分隔符类 ：DelimiterBasedFrameDecoder

将消息分为消息头和消息体：LengthFieldBasedFrameDecoder类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

**Netty**

1：NIO仅仅是一个网络传输框架，而Netty是一个网络应用框架，包括网络以及应用的分层结构

2：在 NIO 中，只能从 Channel 中读取数据到 Buffer 中或将数据从 Buffer 中写入到 Channel。不像传统 IO 的顺序操作，NIO 中可以随意地读取任意位置的数据。

3：Buffer: 与Channel进行交互，数据是从Channel读入buffer缓冲区，从buffer缓冲区写入Channel中的

4：Netty的特点：

一个高性能、异步事件驱动的NIO框架，它提供了对TCP、UDP和文件传输的支持。

使用更高效的socket底层，对epoll空轮询引起的cpu占用飙升在内部进行了处理，避免了直接使用NIO的陷阱，简化了NIO的处理方式。

采用多种decoder/encoder 支持，对TCP粘包/分包进行自动化处理

可使用接受/处理线程池，提高连接效率，对重连、心跳检测的简单支持

可配置IO线程数、TCP参数， TCP接收和发送缓冲区使用直接内存代替堆内存，通过内存池的方式循环利用ByteBuf

通过引用计数器及时申请释放不再引用的对象，降低了GC频率

使用单线程串行化的方式，高效的Reactor线程模型

5：Reactor模型：

在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生，事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。如在Reactor中实现读：注册读就绪事件和相应的事件处理器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

6：Netty的线程模式：

Netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read和write事件，由对应的Handler处理。

7：传统JAVA中NIO，epoll 的空轮训bug

若Selector的轮询结果为空，也没有wakeup或新消息处理，则发生空轮询，CPU使用率100%，

Netty的解决办法：

对Selector的select操作周期进行统计，每完成一次空的select操作进行一次计数，超过512次，则重建Selector；

若在某个周期内连续发生N次空轮询，则触发了epoll死循环bug。

重建Selector，判断是否是其他线程发起的重建请求，若不是则将原SocketChannel从旧的Selector上去除注册，重新注册到新的Selector上，并将原来的Selector关闭

8：Netty中handler的执行顺序；

Netty中的所有handler都实现自ChannelHandler接口。按照输出输出来分，分为ChannelInboundHandler、ChannelOutboundHandler两大类。ChannelInboundHandler对从客户端发往服务器的报文进行处理，一般用来执行解码、读取客户端数据、进行业务处理等；ChannelOutboundHandler对从服务器发往客户端的报文进行处理，一般用来进行编码、发送报文到客户端。

Netty中，可以注册多个handler。ChannelInboundHandler按照注册的先后顺序执行；ChannelOutboundHandler按照注册的先后顺序逆序执行，

9：Netty发送和接收消息主要使用bytebuffer，可以实现零拷贝，合并数据不需要拷贝数据，只需要组合bytebuffer的引用，从而避免数据合并时数据的拷贝。

**LINUX**

1：select,poll,epoll之间的区别

都是IO多路复用的机制，本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。

Select：O(n)，它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

Poll：O(n)，poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

Epoll：O(1)，epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）

2：