**动态代理** 运行时动态生成字节码加载到JVM。JDK动态代理java.lang.reflect.Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)创建代理对象。调用原对象方法时，实际调用到实现Handler接口的类的invoke方法,在其中自定义处理逻辑。只能对接口进行代理。CGLIB通过继承的方式为目标类创建子类，并重写其中的方法以实现代理功能。Enhancer创建代理类，重写MethodInterceptor的intercept方法,拦截method增强代理类。

**ArrayList扩容** 创建新数组，容量是oldCapacity+(oldCapacity >> 1，用System.arraycopy将原数组中所有元素复制到新数组。将内部引用从原来的数组指向新创建的数组。原数组不再被任何引用指向，不可达会被JVM的垃圾回收器回收。

**HashMap Put** 1.数组是空的，先调用resize进行初始化（扩容之前2倍）2.根据key计算哈希值的高16位与低16位进行异或，找到该元素在数组中存储的下标3.如果key不存在则元素直接放在对应的数组下标;4.如果冲突，发现该节点是红黑树，就将这个节点添加到树上5.冲突发现是链表，判断该链表大于8并且数组容量小于 64，就进行扩容，链表插入键值对;如果链表节点大于8并且数组的容量大于64，则将这个结构转换为红黑树添加。

**HashMap 扩容** 第一次初始化调resize数组长度为16，以后扩容都是达到了阀值(数组长度\*0.75)；扩容之后，会新创建一个数组，把老数组中的数据移动到新数组：没有hash冲突的节点，则直接使用 e.hash &(newCap-1)计算新数组的索引位置；如果是红黑树，走红黑树的添加；如果是链表，则需要遍历链表，可能需要拆分链表，判断(e.hash & oldCap)是否为0，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上

**HashMap 线程不安全**1. 一个线程正在执行插入操作的同时，另一个线程可能正在进行扩容操作，导致链表形成循环，从而使程序陷入死循环或抛出异常。2. 元素重新分配到新的数组Rehash时，有其他线程同时进行读写操作，就可能发生数据丢失、覆盖。3.多线程的put操作，一个线程对 HashMap 的修改可能不会立即对其他线程可见,索引相同也会覆盖。

**ConcurrentHashMap PUT**用synchronized锁住链表头节点或红黑树根节点，采用CAS添加新节点：

1.根据 key 计算出 hash值位置。2.判断是否需要进行初始化：使用Segment[0]的容量和负载因子创建一个HashEntry数组，使用创建的HashEntry数组初始化这个节点.3.定位到 Node，拿到首节点 f，判断首节点 f:如果为 nul ，则通过cas的方式尝试添加。如果为f.hash=MOVED =-1，说明其他线程在扩容，参与一起扩容。如果都不满足，synchronized 锁住f节点，判断是链表还是红黑树，遍历插入。在 ConcurrentHashMap 中，当链表长度达到 8 且数组容量大于等于 64 时，会将链表转换为红黑树。如果数组容量小于 64，则优先进行数组扩容

**ConcurrentHashMap扩容**1.创建一个容量为原数组两倍的新数组。新数组的大小是旧数组的两倍2. 扩容过程多个线程并发参与迁移工作，每个线程通过 CAS 操作获取16个桶的范围，并负责将这些桶中的节点迁移到新数组中。3. 对于每个桶中的节点，迁移过程如下：遍历链表或红黑树：从头节点开始逐个处理桶中的节点。根据节点的哈希值重新计算它在新数组中的位置。如果哈希值的高位为 0，则节点留在原位置。如果哈希值的高位为 1，则节点被分配到新位置（即原位置 + 原数组长度）。将节点按照重新计算的位置插入到新数组中，构建新链表或红黑树。4. 当所有桶都迁移完成后，新数组替换为当前的主数组，并释放旧数组。

**进程线程** 进程是操作系统资源分配单位,线程是处理器调度单位,进程有独立代码地址空间上下文切换开销大,线程共享

**线程生命周期** 1)线程对象对创建进入New状态 2）调用线程对象的start进入就绪Runnable，随时等待CPU调度执行3）Running:就绪状态是进入到运行状态的唯一入口,真正执行运行状态4）阻塞:运行中暂时放弃对CPU的使用权，直到其进入就绪状态:1.线程在获取同步锁失败(被其它线程占用)，进入同步Blocked;2.调用Thread.sleep/join进入TimeWaiting,线程挂起不释放锁,当sleep超时/join等待线程终止，线程重新转入就绪状态.3.synchronized中执行Object.wait方法，使本线程进入waiting,释放锁,需Object.notify

**死锁必要条件**互斥条件：任意一个时刻一个线程占用。请求与保持条件：请求资源阻塞时，对已获得的资源保持不放。不剥夺条件：已获得资源在未使用完不能被其他线程剥夺。循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。**破坏**请求与保持：一次性申请所有的资源。破坏不剥夺：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。破坏循环等待条件：按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放

**JMM内存模型**.创建的实例对象存放在主内存（共享内存），每个线程把变量保存本地内存，本地内存会同步到主内存，多线程下对共享变量操作有线程安全问题，JMM的核心目标是在支持多线程并发执行的同时确保内存操作的可见性和有序性：

程序顺序规则：一个线程中的每个操作，**happens-before**于该线程中的任意后续操作。监视器锁规则：对一个监视器的解锁，happens-before于随后对同一个监视器的加锁。变量规则：对volatile域的写入，happens-before于所有其他线程对该volatile域的读取。线程启动规则：Thread.start()方法调用，happens-before于被启动线程的所有动作。线程终止规则：线程中的所有操作happen-before其他线程从该线程的join()方法成功返回。传递性：如果A->B，B->C，则A->C。

**Volatile**关键字用于多线程环境下，解决线程之间的内存可见性和指令重排序问题.可见性：当一个线程修改了volatile修饰的变量时，其他线程能够立即看到该变量的最新值，因为volatile会强制将变量的值从主内存中读取，而非线程的本工作内存。有序性：编译器或处理器为了优化性能，会对代码执行顺序进行调整（指令重排序）。在多线程环境不安全，volatile可以禁止对变量的指令重排序，保证程序的执行顺序符合预期。

**synchronized原理** 每个对象都有一个与之关联的Monitor锁，这个锁可以被一个线程持有，并阻塞其他线程进入需要相同锁的同步代码块或方法：monitorenter和monitorexit指令，指向同步代码块的开始位置和结束位置，出现异常也会执行monitorexit 指令。在执行monitorenter时，会尝试获取对象的锁，如果锁的计数器为 0 则表示锁可以被获取，获取后将锁计数器设为 1 也就是加 1。对象锁的拥有者线程才可以执行 monitorexit 指令来释放锁。在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放，其他线程可以尝试获取锁。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

**可重入性**：一个线程首次获取到某个对象的锁并进入synchronized同步块或方法后，如果该线程再次尝试获取同一个锁，Java的锁机制允许这个线程重新获取这个锁而不会发生死锁。这是通过增加锁的计数器实现的：每次成功获取锁时，计数器加1；每次释放锁时，计数器减1。只有当计数器回到0时，才真正释放锁，使得其他等待的线程有机会获取锁。

**ReentrantLock对比synchronized**增加了特性：1.通过lock.lockInterruptibly提供了能够中断等待锁的线程的机制。当前线程在等待获取锁的过程中，如果其他线程中断当前线程，当前线程就会抛出InterruptedException异常。2可实现公平锁: 通过构造器fair true可以指定是公平锁还是非公平锁。3可实现选择性绑定多个条件:借助于Condition接口与newCondition()方法。4支持超时：提供了tryLock(timeout)的方法，可以指定等待获取锁的最长等待时间，如果超过了等待时间，就会获取锁失败，不会一直等待。synchronized与wait()和notify()方法相结合可以实现等待/通知机制

**ThreadLocal**每个线程都有自己的专属本地变量，Thread类的ThreadLocal类型, ThreadLocal.ThreadLocalMap哈希表维护Entry数组,每个Entry是一个键值对,键是ThreadLocal对象本身，值是与该线程相关的值。每个线程在往ThreadLocal set时,都是往ThreadLocalMap里存，读也是在自己的map里找对应的某个ThreadLocal作为引用的key，从而实现了线程隔离。

ThrealLocal 类中可以通过Thread.currentThread()获取到当前线程对象后，直接通过getMap(Thread t)可以访问到该线程

**ThreadLocal内存泄漏**ThreadLocalMap的key是弱引用(WeakReference<ThreadLocal>)。如果 ThreadLocal 实例不再被任何强引用指向，垃圾回收器会在下次GC时回收该实例，导致对应的key变为null。value是强引用：仍然被ThreadLocalMap.Entry强引用存在，无法被 GC 回收。线程持续存活（线程池中的线程），ThreadLocalMap也会一直存在，导致key为 null的entry无法被回收，造成内存泄漏。在使用完 ThreadLocal后try-finally remove()方法。

**AQS队列同步器原理** 使用volatile state变量表示同步状态(0未锁定1锁定)，通过线程等待CLH双向队列，对等待的线程进行阻塞操作，当队列前边的线程释放锁之后，需要对后边的线程进行唤醒，前一个线程再释放,来完成获取资源线程的排队工作。多个资源抢state资源，通过cas保证原子性。实现：分为独占锁ReentrantLock，通过新线程是否加入队列决定是否公平.共享锁Semaphore、CountDownLatch：构造state值为count。当线程使用countDown时,以CAS的操作来减少 state,直至state为0该线程才会从阻塞中被唤醒。调用await()方法时，state不为0就会一直阻塞。CyclicBarrier：让一组线程到达一个屏障被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活

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

**流程：**1.线程池执行execute/submit方法向线程池添加任务，当任务小于核心线程数corePoolSize，线程池

中可以创建新的线程。2.当任务大于核心线程数corePoolSize，就向阻塞队列添加任务。3.如果阻塞队列已满，需要通过比较参数maximumPoolSize，在线程池创建新的线程，当线程数量大于maximumPoolSize，说明当前设置线程池中线程已经处理不了了，就会执行饱和策略。

**参数** corePoolSize保持最小线程数量，处于空闲状态也不会被销毁。maxPoolSize所有核心线程都在忙碌时，增加直到这个数目。keepAliveTime超过核心线程数时，多余的空闲线程在终止前等待的最长时间。workQueue等待执行任务的阻塞队列。threadFactory线程工厂，默认defaultThreadFactory()。handler（拒绝策略）：提交任务的策略：AbortPolicy：默认策略，抛出RejectedExecutionException。CallerRunsPolicy：由提交任务的线程执行该任务。DiscardPolicy：丢弃无法处理的任务。DiscardOldestPolicy：丢弃队列中最老的任务，尝试重新提交当前任务。

**常用线程池**newCachedThreadPool（SynchronousQueue）newFixedThreadPool/newSingleThreadExecutor（LinkedBlockingQueue）、newScheduleThreadPool（DelayedWorkQueue）

**CompletableFuture** 函数式编程，CompletionStage 将多个异步任务串联起来，组成一个完整的链式调用。默认使用全局共享的 ForkJoinPool.commonPool作为执行器，所有未指定执行器的异步任务都会使用该线程池，共享同一个线程池提交大量任务时，可能会导致资源竞争和线程饥饿，进而影响系统性能。为CompletableFuture提供自定义线程池：为不同任务分配独立的线程池，避免全局线程池资源争夺。根据任务特性调整线程池大小和队列类型，优化性能表现。

**双亲委派机制** 层级关系：Bootstrap Class Loader, Extension Class Loader, Application Class Loader, 自定义类加载器。JVM 不在启动时把所有.class文件都加载，而是程序在运行过程中用到了这个类才去加载，加载的过程中首先检查这个类是不是已经被加载过了，如果加载过了直接返回，否则委派给父/祖父加载器加载。TomCat/webApp/lib单独装载。

**垃圾回收算法** 标记清除法:利用可达性去遍历内存，把存活对象和垃圾对象进行标记;第二步:遍历将所有标记的对象回收掉;特点:标记和清除后会产生大量的不连续的空间分片;标记整理法:第二步将所有的存活的对象向一段移动，将端边界以外的对象都回收掉.复制算法:将内存按照容量大小分为大小相等的两块，一块用完了，将存活对象移到另一块，然后在使用过的内存空间移除;特点:不会产生空间碎片;内存使用率极低;分代收集算法:根据内存对象存活周期，将内存划成新生代和老生代，在新生代中，大量对象死去采用复制算法成本低;老年代中因为对象的存活率极高，采用标记清理或者标记整理算法进行回收;

**一次完整的GC** ]ava堆分区:Eden、To Survivor、From Survivor、老年代。对象优先在Eden分配,当eden区没有足够空间进行分配，jvm发起一次youngGC。在Eden区执行了第一次GC之后，存活的对象会被移动到Survivor分区；Eden区再次GC，这时采用复制算法，将Ede和from区一起清理，存活的对象会被复制到to区:移动一次，对象年龄加1，对象年龄大于一定阀值MaxTenuringThreshold(默认15)会直接移动到老年代。动态对象使用率判定:Survivor区相同年龄所有对象大小的总和>(Survivor 区内存大小\*使用率TargetsurvivorRatio)时，直接进入老年代。Survivor区内存不足会发生担保分配，超过指定大小的对象可以直接进入老年代：在执行每次YoungGC之前，先检查老年代最大可用连续空间是否大于新生代所有对象大小，小于就会先检査HandlePromotionFailure是否担保失败，允许担保就会判断老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小，大于将尝试YoungGC，小于或者参数不允许担保失败，进行一次 Full GC。

**G1回收过程**:1.初始标记(会STW):标记一下GC Roots关联到的对象，修改TAMS指针的值，让用户线程并发运行时，能正确地在可用的Region中分配新对象。2.并发标记:从GCRoots开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。3.最终标记(会STW):对用户线程做短暂的暂停，处理并发阶段结束后仍有引用变动的对象。4.清理阶段(会STW):更新Region的统计数据，对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个Region构成回收集，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间。这里的操作涉及存活对象的移动，必须暂停用户线程，由多条回收器线程并行完成的。

**Mysql并发事务问题** 脏读：一个事务读到其他事务未提交数据；不可重复读：一个事务先后读取两次数据不同；幻读：在一个事务中，两次执行相同的查询时，由于其他事务的插入操作，导致第二次查询结果集中出现了第一次查询时不存在的行(间隙锁锁定记录之间的间隙，防止其他事务在间隙中插入新数据)

**隔离级别**。未提交：最低隔离级别，允许脏读、不可重复读和幻读。Committed：允许不可重复读和幻读。Repeatable Read：默认隔离级别，允许幻读。Serializable。

**BinLog**:记录所有更改数据的操作，主服务器将二进制日志发送给从服务器，从服务器重放日志实现数据**同步**。

**RedoLog**:记录事务对数据页的修改操作，在事务提交前写入磁盘,用于崩溃恢复，确保已提交的事务修改不丢失-持久性

**UndoLog**:记录事务修改前的数据状态，支持事务的回滚操作,保证事务的隔离性,实现多版本并发控制 (MVCC)-一致性原则性

隔离性:事务在并发执行时相互隔离，不受其他事务修改影响。当一个事务修改一行记录时，新版本的数据则被标记为当前版，旧版本的数据会被移动到UndoLog。不同的事务可以访问到该行记录的不同版本-mvcc。

**MVCC**:维护数据的不同版本来实现无锁读取。写操作不会直接覆盖旧的数据，而是创建数据的新版本。隐藏字段、ReadView

**索引**是一种快速检索数据的数据结构（排序好），可以减少IO次数，动态维护耗费时间，降低增删改效率，需要物理存储。

B & B+ 多路平衡查找树:B 树的所有节点既存放键也存放数据，而 B+ 树只有叶子节点存放键值，其他内节点只存放键和指向叶子节点的指针。B 树的叶子节点都是独立的；B+ 树的叶子节点有指向与它相邻叶子节点的引用链。B 树的检索过程相当于对范围内的所有节点做二分查找，可能还没有到达叶子节点，检索就结束了。而 B+ 树的任何查找都是从根节点到叶子节点，范围查找只需要对链表进行遍历。B+ 树与 B 树相比，具备更少的 IO 次数、更稳定的查询效率、更适于范围查询。

**慢sql：**slow\_query\_log=ON、long\_query\_time。EXPLAIN执行计划:possible\_key、key\_len、extra、type(const、ref、index、all)针对常作为查询条件、排序、分组groupBy操作的字段建立索引。ddl设计合适类型、union all、join用小表驱动。聚簇索引:数据与索引放在一块，唯一。二级索引：数据与索引分开存储，不唯一，可以拿到主键再回表查整行数据。

覆盖索引：确保查询的字段都能通过索引直接获取，避免回表操作。覆盖索引+子查询优化深分页。

**索引失效**：违反最左前缀原则(左边开始任何连续的索引都能匹配,遇到>、between、like会停止匹配)。表达式、or、in。

**分库分表**:海量数据、磁盘/网络IO瓶颈、查询连接多cpu瓶颈。根据业务领域垂直分库、常用字段和详情字段垂直分表、多来源的数据水平分表、分库需要考虑路由规则。事务一致性问题、跨节点管理/分页/排序查询、主键避重ShardingSphere

**Redis持久化**RDB: bgsave通过fork子进程,采用copyOnWrite技术,主进程读操作访问共享内粗,主进程写操作会拷贝一份数据执行。AOF:每一个写命令都记录追加文件,比RDB慢、完整、文件大,优先级高

**Redis主从同步** 从节点执行replicaof命令建连，主节点根据replid、offset判断是否首次，首次则返回版本信息，执行bgsave生成RDB文件并发送，期间所有命令记录replBaklog并根据offset的差值发送从节点；非首次则直接执行发送。

**Redis高可用**哨兵监控主节点故障，将从节点升为主：心跳机制向集群ping，超过一半的哨兵认为下线则客观下线，选主规则可以根据主从断开时间、从节点priority配置值、offset值大小、id判断。脑裂：主节点网络抖动，哨兵从分区选主，但client往老节点写数据，网络恢复同步数据丢失，可以配置写数据最少有一个从节点、主从同步延迟时间来拒绝写入

**Redis海量数据存储、高并发写问题** 集群有多个主节点，每个主节点保存不同数据并有多个从节点，主节点之间通过心跳ping监控彼此，CRC16计算hash后对16384取模决定放哪个分片槽，从而决定key的路由，key可以设置有效部分。

**Redis为什么快** 纯内存操作，避免不必要上下文切换竞争，I/O多路复用优化网络瓶颈：Linux为提高**IO**效率，会把用户空间和内核空间都加入缓冲：写数据时把用户缓存数据拷贝到内核缓冲区然后写入设备，读数据时从设备读取数据到内核缓冲区然后拷贝到用户缓冲区。阻塞:应用程序发起IO操作后，会一直等待操作完成才执行后续。访问到内核，内核数据拷贝到用户缓冲区，过程阻塞。非阻塞:用户IO操作时，不会阻塞当前线程，立刻返回未完成状态。应用程序不断轮询是否可以读写操作。拷贝依然阻塞。IO多路复用:用户进程调用select/poll同时监听多个socket，任一可读写时得到通知来recvfrom，内核拷贝。 epoll会在通知socket就绪同时把它写入用户空间。Redis网络模型结合多路复用+事件派发的连接应答、命令回复、命令请求处理器，在命令回复、命令转换处理器支持多线程处理回复事件、增加命令转换速度，执行命令依然单线程。

**OSI七层** 物理(比特传输)、链路(数据桢)、网络(路由转发)、传输(TCP/UDP)、会话(验证登录)、表示(编码)、应用(http/ssh)

**HTTPS**=HTTP+SSL/TLS作非对称加密(通信时私钥由解密者保存，公钥由发送者知道)和安全认证。

**网页浏览过程** :浏览器输入网页URL。通过DNS协议，获取域名对应IP地址。根据IP端口号向服务器发起TCP连接请求。浏基于TCP向服务器发送HTTP请求报文。服务器收到HTTP请求，处理并返回HTTP响应报文。浏览器收到响应报文，解析响应体的HTML代码，渲染网页并根据HTML中其他URL(图片、CSS、JS)，再次发起 HTTP 请求获取这些资源内容，直到网页完全加载显示。浏览器在不需要和服务器通信时，可以主动关闭TCP连接，或者等待服务器的关闭请求。

**TCP**(传输控制)和**UDP**(用户数据报)传输层协议:udp无连接,不可靠无序,固定速率.tcp文件传输(FTP)/电子邮件(SMTP)/网页浏览(HTTP)-udp视频会议/游戏可以容忍一定的数据丢失以换取更低的延迟.

**Restful** (Representational State Transfer)资源/表示形式/状态转移,软件架构风格用标准的HTTP方法POST来操作资源

**三次握手四次挥手**一次**握手**:客户端发送带有 SYN（SEQ=x） 标志的数据包 -> 服务端，然后客户端进入 SYN\_SEND 状态，等待服务端的确认；二次握手:服务端发送带有 SYN+ACK(SEQ=y,ACK=x+1) 标志的数据包 –> 客户端,然后服务端进入 SYN\_RECV 状态；三次握手:客户端发送带有 ACK(ACK=y+1) 标志的数据包 –> 服务端，然后客户端和服务端都进入ESTABLISHED 状态，完成 TCP 三次握手（第三次握手确认客户端不仅能够发送数据，还能够接收来自服务器的数据，确保了客户端到服务器的数据传输通道是畅通的；在网络存在延迟的情况下，可能会出现过期的SYN请求到达服务器并尝试建立连接的情况。如果仅依赖两次握手，则可能会导致基于旧的SYN请求建立不必要的历史错误连接，第三次握手客户端会发送针对最新接收到的SYN-ACK的ACK响应。服务器通过这个响应认为连接已经建立成功）。

第一次**挥手**：客户端发送一个 FIN（SEQ=x） 标志的数据包->服务端，用来关闭客户端到服务端的数据传送。然后客户端进入 FIN-WAIT-1 状态。第二次挥手：服务端收到这个 FIN（SEQ=X） 标志的数据包，它发送一个 ACK （ACK=x+1）标志的数据包->客户端 。然后服务端进入 CLOSE-WAIT 状态，客户端进入 FIN-WAIT-2 状态。第三次挥手：服务端发送一个 FIN (SEQ=y)标志的数据包->客户端，请求关闭连接，然后服务端进入 LAST-ACK 状态。第四次挥手：客户端发送 ACK (ACK=y+1)标志的数据包->服务端，然后客户端进入TIME-WAIT状态，服务端在收到 ACK (ACK=y+1)标志的数据包后进入 CLOSE 状态。此时如果客户端等待2MSL后依然没有收到回复，就证明服务端已正常关闭，随后客户端也可以关闭连接了。(确保了TCP连接的完全、有序关闭，处于TIME\_WAIT状态的主动关闭方会等待2msl防止旧连接的数据包干扰新连接)

SYN Flood是DDoS（分布式拒绝服务）攻击，旨在耗尽可用服务器资源，致使服务器无法传输合法流量:伪造IP向服务器发送大量SYN包。服务器按TCP协议回应SYN-ACK，并等不到客户端最终ACK包，导致服务器为未完成的连接保持资源占用

sudo **netstat** -tlnp|grep:80以超级用户权限检查80端口是否被监听LISTEN ESTABLISHED(连接) **lsof**列出系统打开文件

**RPC**(调用远程方法像调用本地方法一样简单)client以本地调用的方式调用远程服务；client stub接收到调用后负责将方法、参数等序列化成能够进行网络传输的消息体RpcRequest；client stub找到远程服务的地址，并将消息发送到服务提供端；服务端Stub收到消息将消息反序列化为RpcRequest；服务端Stub根据RpcRequest中的类、方法、方法参数等信息调用本地的方法；服务端Stub得到方法执行结果并将序列化成能够进行网络传输的消息体RpcResponse发送至消费方；client stub接收到消息并将消息反序列化为Java对象:RpcResponse

**ZooKeeper**分布式协调服务:服务提供者将自己的信息(服务名、ip、端口)注册到zk;消费者(通过负载均衡器)向zk拉取服务列表信息;提供者会向zk发心跳,zk会向集群发健康检查,下线则剔除。提供数据存储（znode）、事件监听（允许用户在指定节点上注册一些 Watcher，并且在一些特定事件触发的时候，zk服务端会将事件通知到客户端上去）

**负载均衡策略**随机、简单轮询、加权轮询、可用性策略忽略短路服务/选择连接数空闲的、指定时间重试、ipHash保持会话

**Spring IoC和DI**程序中手动创建对象的控制权，交给IOC容器来管理，完成对象定义的依赖关系的注入，BeanFactory管理应用程序中的 Bean 及其生命周期。

**Spint AOP**将与业务无关，共同调用的逻辑（例如事务处理、日志管理、权限控制）封装起来，便于减少系统重复代码，降低模块耦合，可扩展性和可维护性。如果要代理的对象实现了某个接口，那么Spring AOP就会使用JDK动态代理去创建代理对象；没有实现接口的对象，用CGlib：通过生成一个被代理对象的子类来实现代理。

Jointpoint（连接点）：切面点抽象概念，可以是在字段、方法上，Spring中具体表现形式是PointCut（切入点），仅作用在方法上。Advice（通知）: 在连接点进行的具体操作，如何进行增强处理，分为前置、后置、异常、最终、环绕五种情况。

目标对象（target object）：被增强的对象，被AOP框架进行增强处理的对象。Weaving（织入）：将增强处理添加到目标对象中，创建一个被增强的对象，在目标对象的某些方法（jointpoint）添加不同种类的操作（通知、增强操处理）。

**Spirng bean生命周期** 创建1.Spring容器根据BeanDefinition获取实例化信息。2.调用构造器实例化Bean。3.将配置文件中指定的属性值或依赖注入到Bean中。4.处理Aware接口:BeanNameAware调用setBeanName方法,BeanFactoryAware/ApplicationContextAware调用setBeanFactory/setApplicationContext使Bean可以访问Spring容器。4.BeanPostProcessor调用postProcessBeforeInitialization。5.InitializingBean接口调用afterPropertiesSet()初始化。6.BeanPostProcessor调用postProcessAfterInitialization。Bean就绪。销毁DisposableBean接调用destroy()方法，自定义则调自定义清理。

**Spring解决Singleton作用域下循环依赖** 前提是setter注入。三级缓存让Spring 能够在 Bean 初始化完成之前就将其引用暴露给其他 Bean。一级缓存（singletonObjects）：存放完全初始化好的单例 Bean；二级缓存（earlySingletonObjects）：存放提前暴露的 Bean 实例（尚未完成初始化）;三级缓存（singletonFactories）：存放 Bean 工厂对象，用于生成提前暴露的 Bean 实例。A->B->A,初始化A需要注入B,初始化B依赖二级缓存获取A,B创建成功从二级移到一级,B注入给A,A创建成功。代理对象创建发生在initializeBean，放在一级缓存;原始对象放在二级缓存。只用二级缓存则拿不到代理对象。

三级缓存：A->B,将A对象ObjectFactory放入三级缓存, B->A，从三级缓存获取A的工厂对象，通过工厂动态生成代理对象，将代理对象放入二级缓存。

**Spring MVC** JSP:1.客户端发送请求到前端控制器DispatcherServlet。2.DispatcherServlet收到请求调用处理映射器HandlerMapping，HandlerMapping找到具体处理器生成处理器对象和拦截器返回给DispatcherServlet。3.DispatcherServlet找到处理适配器，HandlerAdapter经过适配调用具体Controller。4.Controller执行相对应的业务逻辑后，会返回一个ModelAndView对象并通过HandlerAdapter返回给DispatcherServlet。5.DispatcherServlet讲modelAndView传给视图解析器，ViewResolver解析返回具体的View。6.DispatcherServlet根据View进行视图渲染，返回给客户端。前后端分离模式:Controller注解@ResponseBody可以通过HttpMessageConvert将结果转换成JSON直接响应。

**Spring事务** 编程式事务管理：通过编码的方式直接控制事务的边界和行为。TransactionTemplate（execute callback）或TransactionManager来进行编程式事务管理commit、rollback。声明式事务管理：基于XML和注解@Transactional驱动，将事务管理与业务逻辑分离，原理是在目标方法被调用前，代理对象创建事务，执行完提交，异常回滚，业务层具有传播性。

**Spring Boot** 基于Spring快速开发框架，简化应用程序的开发过程，提供约定优于配置的理念，极大地减少了手动配置工作：1.使用注解和默认配置来替代大量的 XML或配置文件，根据项目引入的依赖，自动配置应用程序所需的组件（引入starter-data-jpa，会自动配置数据源、实体管理器和事务管理器）。2.内置了 Tomcat、Jetty Web 容器，可以直接运行应用程序，无需手动部署到外部服务器，可以在本地打独立JAR，快速启动应用进行调试3.统一Starter依赖管理，提供了一系列 Starter 模块（boot-starter-web），包含所需依赖，避免版本冲突。4. 集中式配置文件application.properties简化属性配置，如ORM 框架，开发者可以通过根据配置文件中的数据库连接信息实现自动配置数据源访问数据库;热部署快速启动;开箱即用;轻量化

**Spring Boot启动流程**1.SpringBoot通过扫描启动类所在的包和子包，自动装配配置类和BeanDefinition。2.加载配置文件:SpringBoot应用程序默认从application.properties文件中加载配置属性，也可以通过在启动类上使用@PropertySource注解指定其他的属性文件。3.SpringBoot使用SpringApplication类创建Spring容器，SpringApplication是SpringBoot的核心类，它提供了各种配置和管理Spring应用程序的方法。SpringApplication会创建一个入式的Tomcat/Jetty服务器。4.加载自动配置:SpringBoot通过@EnableAutoConfiguration注解提供了大量的自动配置，根据classpath中的jar包自动装配相应的Bean。5.当Spring容器准备就绪后Spring Boot就会启动嵌入式的Web服务器并运行。

**Spring Boot自动配置原理** 应用程序的启动类@SpringBootApplication，该注解包含了@SpringBootConfiguration、@EnableAutoConfiguration和@ComponentScan。@EnableAutoConfiguration包含@Import导入对应的配置选择器，内部就是读取该项目引用的Jar包的classPath路径下spring.factories文件中所配置类的全类名，这些配置类会根据条件注解（@Conditional\*）决定是否加载到Spring容器中。支持用户自定义自动配置注册到spring.factories。

**Spring Springmvc常见注解** @Autowired 让Spring容器帮我们自动装配bean。@Component可标注任意类为 Spring 组件。@Repository 对应持久层@Service对应服务层@Controller对应Spring MVC控制层。@Bean使用在方法上，标注该方法返回值存储spring容器。@RequestMapping映射请求路径，用于类上则表示所有方法的父路径。@RestController注解是@Controller和@ResponseBody的合集，将controller返回值转为json填入HTTP响应体中。

**Mybatis执行流程**1.读取mybatis-config.xml加载运行环境和映射文件。2.构造会话工厂，创建SqlSession对象(包含了执行SQL语句的所有方法)3.操作数据库的接口，Executor执行器(同时负责查询缓存的维护)4.执行器接口的执行方法中有一个MappedStatement类型的参数,封装了映射信息5.java对象输入参数映射到sql,sql查完输出结果映射

**Mybatis二级缓存** 一级缓存: 基于PerpetualCache的本地Map缓存，其存储作用域为 Session，当Session进行flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存。二级缓存是基于namespace和mapper的作用域起作用的,不限于某session。

**需求分析** 需求确认：产品、客户、利益相关者沟通，输出评审书面需求文档/图。功能/非功能需求

**系统设计** 整体架构:微服务(Dubbo、zk),业务架构(业务边界,不考虑技术组件数据库)/应用架构(系统层级,接入层服务层存储层)/数据架构(数据在系统流向,订单/账号/活动数据)/技术选型(每个层次组件,画像存kafak接入,指标计算存储redis/Hbase/Mysql)。UI用户界面交互,数据库设计(ER图),接口设计,流程图/用例图(actor与用例)/时序图(模块交互)/活动图(系统内部工作流)/部署图(客户端,网关,负载均衡,云应用服务器,数据库,告警监控)-planUML统一建模语言

**编码实现** 动任务分解系统功能拆解为多个小任务，scrum(短周期迭代,项目管理:会议目标,团队合作)敏捷开发配合devops(自动化持续集成改进上线运维)。按照模块化的方式开发各个功能,确保每个模块的功能独立且能测试。使用Git管理代码,UT覆盖率、转测门槛用例。CodeReview保证质量,统一日志规范提升可维护性

**测试部署** 单元、集成(模块之间交互)、系统(功能、性能、安全)、端到端测试(用户及体验)。用Docker容器化打包,构建镜像(Dockerfile),推送k8s,配置dryrun/生产环境参数,滚动升级,灰度放量:基于k8sIstio配置小流量到新pod,nginx基于用户特性ip,Prometheus、Grafana实时监控系统状态告警

**系统集成和数据一致性的挑战** 持久层选择高可用性、可扩展性，Cassandra支持分布式部署，应对大规模数据存储的需求。实施多级缓存减少对数据库压力，提高响应速度，设置合理的缓存过期时间，在数据变更时及时更新缓存，保证数据一致性。

**缓存穿透**：预热缓存、布隆过滤器、缓存空值，快速过滤无效请求，减少重复查询。限流熔断、异步处理，控制请求流量。

**削峰填谷** 请求缓冲存储在队列，生产者无需等待处理结果即可返回响应给用户，消费者按自身最大处理能力从消息队列读取消息。低峰期时，利用空闲资源快速消费积压消息。非核心业务（发通知、日志、同步第三方）延后处理。

**积压处理:**消息队列会将消息分发到多个分区（Partition）。每个消费者实例消费一个或多个分区消。工具监控积压消息kafka-consumer-groups查看消费者组的滞后量。滞后较大/pod cpu高，动态增加消费者(或线程)，通过deployment命令。