1. 并发编程：
2. 什么是多线程并发和并行？

并发：处理多个同时性活动，同一个CPU上同时运行多个程序，通过时间片进行切换；

并行：多个线程跑在不同的处理器上；

1. 操作系统进程的常见状态
   1. 就绪，已经获得除了CPU外所有必备的资源，只等待CPU的状态
   2. 执行，已经获得CPU，正在执行，一个CPU中处于执行状态的进程只能有一个
   3. 阻塞，正在执行的进程由于某种原因而暂时无法执行，
2. 进程和线程的区别：
   1. 进程：并发执行的程序再执行过程中分配和管理资源的基本单位
   2. 线程：是进程的一个执行单元，比进程更小的独立运行的基本单位

1. 什么是线程安全问题？

线程安全：拥有共享数据的多个线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可正常正确的执行，不会出现数据污染等问题；

1. 什么是共享变量的内存可见性？

即 一个线程修改了共享变量的值，其他线程也能看到最新修改的值

1. 什么是Java中的原子性操作？

原子性：即一个操作一旦开始要么全部执行完，要么就不执行；

Java中原子性操作：

1. Unsafe类；2.CAS（Compareandswap）JUC包中大量使用；3.i++操作不是线程安全的，因为其是一个复合原子操作，等于三个原子操作复合；4.synchronized阻塞；
2. Automiclong的实现原理：

避免ABA问题，使用automicstampedreference，通过内部维护的一个时间戳来保证；

1. 指令重排序：

在JVM层面，虚拟机会按照自己的一些规则将程序编写顺序打乱，以尽可能充分的利用CPU

1. Synchronized关键字的内存含义：

作用：确保线程互斥的访问同步代码；保证可见性；有效解决重排序；

基于monitorenter和monitorexit指令实现；

每个对象维护一个记录着锁次数的计数器。未被锁定的计数器为0，一个线程获得锁，执行monitorenter，计数器自增1，再次获得该对象的锁，再次自增，释放锁，执行monitorexit，计数器自减1，计数器为0的时候，锁被释放；

锁的状态：

偏向锁：由于锁总是由同一线程获得，为了减少同一线程获取锁的代价，引入偏向锁，如果一个线程获得锁，进入偏向锁模式，当这个线程再次请求锁，无需做任何同步操作；当竞争激烈的时候，偏向锁失效，升级为轻量级锁；

轻量级锁：

自旋锁：轻量级锁失败，升级为自旋锁；假设不久后当前线程可以获得锁，因此JVM让当前想要获取锁的线程做几个空循环，若干循环后，获得锁，则进入临界区，不能获得锁则将线程在操作系统层面挂起；

重量级锁：自旋锁失败，升级为重量级锁；

Synchronized底层实现：

底层有多个队列组成，当多个线程一起访问某个对象监视器，对象监视器会将这些线程存储在不同的容器中：

1. ContentionList：竞争队列：所有的请求锁的线程放在这个竞争队列；
2. EntryList：ContentionList中那些有资格成为候选资源的线程被移动到EntryList；
3. WaitList：那些调用wait方法被阻塞的线程被放在这里；
4. OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程成为OnDeck；
5. Owner：当前以获得锁资源的线程；
6. ！Owner：当前释放锁的线程；

处理流程：

* 1. JVM每次从队列的尾部取一个数据用于锁竞争候选者（OnDeck），但是并发情况下ContentionList会被大量的并发线程进行CAS访问，为了降低对尾部元素的竞争，JVM会把一部分线程移动到Entry List中作为候选竞争线程；
  2. Owner线程会在unlock时，将ContentionList中的部分线程移动到Entry List中，并指定其中的某个线程为OnDeck，Owner并不直接把锁传递给OnDeck，而是把锁竞争的权力交给OnDeck，OnDeck需要重新竞争锁；叫竞争切换；
  3. OnDeck获得锁后变为Owner，而没有获得锁的线程依然留在Entry List中，如果Owner被wait阻塞，则转移到waitList，知道某个时刻通过notify或notify all唤醒，重新进入Entry List；
  4. Synchronized是非公平锁，在线程进入ContentionList中，等待的线程会先尝试自旋获取锁，如果获取不到就进入Contention List；

自旋的具体流程：

* 检测Mark　Word里是不是当前线程的ID，是则，当前线程处于偏向锁，流程结束；
* 不是，尝试获取偏向锁，成功，当前线程进入偏向锁，流程结束；
* 失败，说明有锁竞争，撤销偏向锁，进而升级为轻量级锁；
* 当前线程开始竞争锁，如果成功，当前线程获得锁　；
* 失败，表示其他线程持有锁，当前线程尝试自旋获取锁；
* 自旋成功，则依然处于轻量级锁，并获得锁，流程结束；
* 自旋失败，升级为重量级锁，进入Contention List；

锁的优化策略：

1. 减少锁的时间：
   1. 不需要同不执行的代码，尽量不放在同步块，让锁尽快释放；
   2. 锁方法不如锁方法内的需要同不的代码块；
2. 使用读写锁：
   1. Reentrant Read Write Lock可以并发读写，写操作使用写锁，只能单线程写，
3. 读写分离
4. 使用CAS（volatile＋CAS）

1. Volatile：

Volatile修饰的变量为共享变量，编译器与运行时都会注意到这个变量是共享的，不会讲变量上的操作与其他内存操作一起重排序，volatile变量不会被缓存到寄存器或者其他处理器不可见的地方，因此读取的时候总是返回最新的值；通过内存屏障防止指令重排序；

1. 什么是伪共享，为何出现，如何避免

CPU缓存系统中是以缓存行为单位存储的，目前主流的CPU cache的cache line大小都是64Bytes，在多线程下，如果需要修改一个缓存行的变量，就会无意间影响彼此的性能，这就是伪共享；

CPU三级缓存：L1，L2，L3；

解决：Java8中新增了注解：@Sun.misc.Contended 有这个注解的类后自动补齐缓存行

1. 悲观锁：总是假设最坏情况，每次拿数据都会上锁，行锁，表锁，读锁，写锁等；

乐观锁：总是假设最好情况，在更新的时候会判断一下有没有去更新这个数据，使用版本号机制和CAS算法实现；

公平锁：加锁前查看是否有队列等待的线程，有的话优先处理排在前面的线程；

非公平锁：加锁时直接尝试获取锁，获取不到就自动到队尾等待；

可重入锁：ReenTrantLock，synchronized都是可重入锁，即同一个线程每进入一次，计数器加一，计数器为0，则释放锁；

SynchronizedJVM实现，ReentrantLockJDK实现

synchronized和ReentrantLock性能相差不大

锁粒度灵活度：ReentrantLock优于synchronized

ReentrantLock独有能力：可以指定公平或非公平，synchronized只能非公平；ReentrantLock提供了Condition类，实现分组唤醒需要唤醒的线程们，synchronized随机唤醒；ReentrantLock可以中断等待锁的线程的机制；

1. 控制多线程执行顺序：
   1. 使用join方法，让正在执行的线程停止，让join的这个线程立刻执行并结束；
   2. 使用线程池的submit方法，将线程放到线程池里执行

1. Threadlocal实现原理：

Threadlocal：一个本地线程副本变量工具类，主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用适用于各个线程依赖不同的变量值来完成操作的场景；

ThreadlocalMap没有next引用，也就是解决hash冲突的方式并非链表的方式，而是采用线性探测的方式根据初始key的hashcode值确定元素在table数组中的位置，如果发现已经有其他key值得元素占用，则利用固定算法寻找一定步长的下个位置，依次判断；简单的加一或减一；

一个Threadlocal只能保存一个变量副本；ThreadlocalMap为弱引用，有内存泄漏风险；

1. 并发包中锁的实现底层（AQS：抽象队列同步器）

JUC包中ReentrantLock

和Semaphore等类的基类AQS，构建锁和同步器的框架，通过一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源的排队工作；

AQS支持的同步方式：

独占式ReentrantLock

共享式CountDownLatch

组合式ReentrantReadWriteLock

AQS维护一个共享资源state，内置FIFO来完成获取资源线程的排队工作（CLH），该队列由一个一个的Node节点组成，每个Node维护一个prev引用和next引用，AQS维护两个指针，分别指向队列的头部head，尾部tail；

线程获取资源失败，会被构造一个节点加入CLH队列中，同时当前线程会被阻塞在队列中，当持有同步状态的线程释放同步状态，会唤醒后继节点，然后此节点线程继续加入到队同步状态的竞争中；

1. CountDownLatch和join方法的区别

在当前线程调用另一个线程的join方法时候，会调用该线程的wait方法，直到这个线程执行完毕，方法执行后调用exit方法，而exit方法里调用了notifyall方法会主动唤醒当前线程；

CountDownLatch可以理解为一个计数器，在初始化CountDownLatch的时候会在类的内部初始化一个int变量，每当调用CountDownLatch方法的时候这个变量的值减一，为await方法就是去判断这个变量的值是否为0，是则表示所有的操作都完成，否则继续等待；

1. Concurrenthashmap原理（1.7+1.8）
   1. Java7
      1. 采用数组+segment+分段锁
      2. 数组中每个元素是一个链表，同时也是一个Reentrant Lock
      3. 将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段的数据的时候，其他段的数据也能被其他线程访问，实现真正的高并发；
      4. 定位一个元素，需要两次Hash操作，第一次定位到segment，第二次Hash定位到元素所在的链表的头部；
      5. 也就是ReentranlLock+segment+Hash Entry
      6. 优点缺点：
         1. 优点：写操作的时候可以只对元素所在的segment进行加锁即可，不会影响到其他的segment；
         2. 缺点：Hash的过程比普通的HashMap长；
   2. Java8:
      1. 参考JDK8中HashMap的实现，采用数组+链表+红黑树，大量采用CAS操作；
      2. CAS乐观锁，采取通过不加锁的方式处理资源，包含三个操作数：内存位置，预期原值，新值，如果内存位置的值和预期值一样，那么将内存里的值更新为新值。CAS是通过无限的循环来获取数据，如果第一次循环中，A线程获取地址的值被B线程修改了，那么A线程需要自旋，到下次循环才有可能机会执行；
      3. Java8中放弃segment，采用Node，引入红黑树，（查找性能为OlogN），
      4. 也就是synchronized+CAS+HashEntry+红黑树
2. 操作系统的死锁
   1. 死锁的产生：多个进程并发执行共享系统资源，同步机制和进程通信机制不完善，出现进程永远被阻塞
   2. 死锁的定义：每一个进程都在等待被另一个进程所占有的，不能抢占的资源
   3. 死锁产生的必要条件：
      1. 互斥：进程间互斥使用资源，一个时刻只能一个进程独占一个资源
      2. 占有和等待：一个进程请求得不到并且等待时候，不释放自己已经占有的资源
      3. 不剥夺：任一个进程不能剥夺另一个进程的资源
      4. 循环等待：每一个进程分别等待它的前一个进程所有的资源
   4. 死锁的解决办法：
      1. 死锁防止（破坏四个必要条件之一即可）
         1. 破坏第一个条件，把独占资源改成共享资源
         2. 破坏第三个条件，剥夺式调度方法
         3. 破坏第二个条件，静态分配，一个进程执行前就申请它所需的资源
         4. 破坏第四个条件，层次分配，资源被分成多个层次，按层次申请资源，按层次释放资源；
      2. 死锁避免（为申请者分配资源钱先测试系统状态，若把资源分配给申请者会产生死锁的话，则拒绝分配，否则接受申请，为它分配资源，即银行算法）
         1. 银行家算法：进程首次申请资源，测试其对资源的最大需求，判断系统现存资源能否满足，可以则分配，否则推迟分配；当进程在执行中申请资源，测试进程本次申请的资源数是否超过了该资源剩余的总量，超过则拒绝分配，可以则按当前的申请量分配，否则推迟分配；
      3. 死锁检测和恢复
3. 线程的状态
   1. Java源码中Thread类中有一个枚举类（State）定义六种 状态
   2. 初始（New）：新创建一个线程对象，没有调用start方法
   3. 运行（Runnable）：Java线程中把就绪（ready）和运行中（running）两种状态统一称为运行，线程对象创建后，其他线程调用了该对象的start方法，此时等待被线程调度选取，获得CPU使用权，此时处于就绪状态，就绪状态的线程获得CPU时间片，处于运行中；
   4. 阻塞（Blocked）：线程阻塞于锁
   5. 等待（Waiting）：进入等待线程需要等待其他线程做一些特定动作（通知或中断）
   6. 超时等待（Timed\_Waiting）：可以在指定时间后自行返回
   7. 终止（Terminated）：该线程已经执行完毕；
4. Sleep，wait，notify，notify all，join，yield
   1. Wait方法是object的，sleep是thread的静态方法，调用thread.sleep并不能改变对象的状态，因此也不会释放锁；
   2. 调用wait，notify，notify all方法，需要和锁或者synchronized搭配使用，否则报错IllegalMonitorStateException，因为任何时刻，对象的控制权只能由一个线程持有，因此调用这类方法，必须确保对其的控制权；
   3. 如果对简单对象调用wait方法，如果对他们进行赋值也会报错，因为赋值相当于改变原有的对象；
   4. Notify可以唤醒一个在该对象上等待的线程，notifyall可以唤醒所有等待的线程
   5. Wait可以挂起线程，并释放对象的资源，等待计时结束后自动恢复，wait必须要其他线程调用notify或者notify all才能唤醒；
   6. Join，线程插队，在线程1中执行Thread2.join线程1 会在线程2执行完后在执行
   7. Yield，线程让步，让出自己的分配空间给其他线程，只有于当前线程优先级相同或者更高的才能获得执行机会
5. 线程池专题：
   1. 线程池：JUC包下的Executors提供的Executor接口的实现用于创建线程池；
   2. 线程池的组成：
      1. 线程池管理器（Thread Pool）：用于创建并管理线程池，包括创建，销毁，添加新任务；
      2. 工作线程（Pool Worker）：线程池中的线程，在没有任务时，处于等待，可以循环的执行任务；
      3. 任务接口（Task）：每个任务必须实现Task接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行后的收尾工作，认为的执行状态；
      4. 任务队列（task Queue）：用于存放没有处理的任务，提供缓冲机制；
   3. 常见线程池：
      1. 单线程池：newSingleThreadExecutor:每次只有一个线程工作，串行；
      2. 固定数量线程池：newFixedThreadPool:每提交一个任务就是一个线程，知道达到线程池的最大数量，然后进入等待队列；
      3. 可缓存线程池：newCacheThreadPool：（推荐使用）当线程池大小超过的处理任务需要的线程，就会回收部分空闲的线程（60秒无执行），当有任务来，又只能的添加新线程；
      4. 定时，周期性线程池，大小无限制：newScheduleTheadPool:
   4. 使用线程池原因：
      1. 减少了创建和销毁的次数，每个工作线程可以被重复利用，可执行多个任务；
      2. 可以根据系统的承受能力，调正线程池中工作线程的数量，防止消耗过多的内存；
   5. 重要的几个类：
      1. ExecutorService：真正的线程池接口
      2. ScheduleExecutorService：解决选哟任务重复执行的问题
      3. ThreadPoolExecutor：ExecutorService的默认实现
   6. 线程池核心组成：
      1. Acc:获取调用上下文
      2. CorePoolSize：核心线程数，类似于正式员工
      3. MaximumPoolSize：最大线程数，类似于正式员工+外包
      4. WorkQueue：多余任务等待队列
      5. KeepAliveTime：非核心线程空闲时间，类似于外包人员多久没活干被解雇
      6. ThreadFactory：创建线程的工厂
      7. Handler：线程池拒绝策略（包和策略）：
         1. AbortPolicy：直接抛出RejectedExecutionException，（默认策略）
         2. CallerTRunsPolicy：只要线程池没关，就直接用调用者躲在线程运行
         3. DiscardPolicy：悄悄把任务放生，不做
         4. DiscardOldestPolicy：把队列里最久的任务扔了
   7. 线程池实现原理（源码）
      1. WorkerCountOf方法根据ctl的低29位，得到线程池当前线程数，如果线程数小于核心线程数，则执行add worker方法创建新的线程执行任务；
      2. 判断线程池是否在运行（isRunning），如果在，且任务队列是否允许插入，插入成功再次验证线程池是否运行，如果不在运行，移除插入的任务，然后抛出拒绝策略。如果在运行，且没有线程了，就启用一个线程。
      3. 如果添加非核心线程失败了，就直接拒绝；
      4. 总结：线程池本质是一个hashset，多余的任务放在阻塞队列中，只有当阻塞队列满了后，才会触发非核心线程的创建。非核心线程本质是来打杂的，空闲以后，就自己关闭，线程池提供了两个钩子（beforeExecute，afterExecute），继承线程池，在执行任务前后可以做事，
      5. 线程池原理相关技术：锁（lock，cas），阻塞队列（blockQueue），hashset（资源池）
   8. 不允许使用Executors创建线程池，存在OOM风险：
      1. 原因由于阻塞队列
      2. Blocking Queue实现方式有两种：ArrayBlockingQueue ,LinkedBlockingQUeue,；前者有界，后者不设置容量就是无界的，而FixedThreadPool和SingleThreadExecutor采用的是后者，因此可能会导致任务过多而导致内存溢出；

g) 如何计算线程池核心线程的数量：

目的：为了提高CPU的利用率；

计算密集型：N个处理器，核心线程数=N+1；

IO密集型：N个处理器，核心线程数=2N+1；

1. JVM
2. Java的内存分配：

每运行一个Java程序，产生一个Java进程，每个Java进程可能包含一个或者多个线程，每一个Java进程对应为一个JVM实例，每一个JVM实例唯一对应一个堆，每一个线程对应一个私有栈。进程所创建的所有的类实例或数组放在堆中，并由该进程的所有线程共享。对象的引用在栈中分配。局部变量new出来时，在栈和堆空间分配内存空间，当局部变量生命周期结束后，栈空间立刻被收回，堆空间区域等待GC回收；

1. 堆：存储对象，每个对象包含一个与之对应的class的信息；JVM只有一个堆，且被线程共享，不存放基本数据类型和对象引用；
2. 栈：每个线程一个栈区，栈中只保存基础数据类型和自定义对象引用；每个栈中数据私有；栈分为三个部分，基本类型变量区，执行环境上下文，操作指令区；
3. 方法区：所有线程共享，包含所有的class和static变量；包含的都是整个程序中永远唯一的元素，例如class，static变量；
4. Java的堆结构

运行时数据区，所有的类实例和数组在堆中分配内存，在JVM启动时创建，对象所占的堆内存由自动内存管理系统也就垃圾收集器回收；

堆分为 新生代，老年代，持久代；持久代存放Java类的信息；

年轻代：新生对象放在年轻代，尽可能快速的收集掉那些生命周期短的对象，分为三个区，Eden，S0,S1，，两个S区解决内存碎片化；

老年代：新生代经历N次（16次）回收还存活的对象进入老年代；

1. Java中Boolean到底占几个字节？
   1. Java虚拟机规范中定义了单个Boolean类型，编译后使用Java虚拟机规范中的int数据类型来代替，因此此时占4个字节
   2. 但是Boolean数组会被编码为Java虚拟机的byte数组，每个元素在数组中占一个字节；
2. Java中内存泄漏

内存溢出：out of memory，申请内存失败，出现OOM

内存泄漏：memory leak：申请内存后，无法释放申请的内存，最终会导致OOM

内存泄漏原因：无用对象持续占有内存或无用对象的内存得不到及时释放、

静态集合类引起内存泄漏；各种连接，没有显示的调用close进行关闭；单例模式，单例对象持有外部对象的引用，那么这个外部对象无法被JVM正常回收，导致内存泄漏；

1. 串行收集器（serial）和吞吐量收集器（parallel scavenge）的区别
   1. 串行GC，整个扫描和复制的过程均采用单线程的方式，相对于吞吐量GC来说，简单；适合于单CPU，客户端级别；
   2. 吞吐量GC：采用多线程的方式来完成垃圾收集，适合于吞吐量要求比较高的场景，比较适合中等和大规模的应用程序；
2. JMM相关（Java　Memory　Model）
   1. 规定所有的变量都存储在主内存中，每条线程有自己的工作内存，工作内存中保存了线程使用到的变量到主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问到对方工作内存的变量，线程直接变量值的传递需要在主内存中来完成；
   2. 八个原子操作：
      1. Lock：锁定：作用于主内存变量，把一个变量标识为一个线程独占状态；
      2. Unlock：解锁：作用于主内存变量，把一个处于锁定的变量释放，
      3. Read：读取：作用于主内存变量，把一个变量值从主内存传输到工作内存，以便随后的load
      4. Load：载入：作用于工作内存，把read操作从主内存得到的变量值放入工作内存的变量副本中；
      5. Use：使用：作用于工作内存，把工作内存中的一个变量值传递给执行引擎，每当JVM遇到一个需要使用变量的值的字节吗指令，就执行use；
      6. Assign：赋值：作用域工作内存，把一个从执行引擎接受到的值赋给工作内存的变量，每当虚拟机遇到一个变量复制的字节码指令就会执行assign；
      7. Store：存储：作用于工作内存，把工作内存的一个变量的值传给主内存，以便随后的write；
      8. Write：写入：作用于主内存，把store传的值传送到主内存的变量中；
   3. Read／load，store／write两个操作必须按顺序执行，但没有要求连续执行；
   4. 不允许read／load，store／write，之一单独出现；
   5. 一个变量同一时刻，只允许一个线程对其进行lock操作，lock和unlock必须成对出现；
   6. 一个变量执行lock操作，将会清空工作内存中次变量的值
3. 垃圾回收器的基本原理
   1. 垃圾回收可以有效的防止内存泄漏
4. 垃圾回收算法
   1. 引用计数法（淘汰）无法解决循环引用
   2. 根搜索法（可达性分析）
   3. 标记-清除：
      1. 采用从根集合进行扫描，对存活对象标记，标记完毕后，再扫描整个空间内未被标记的对象，进行回收，不需要移动对象
      2. 缺点：由于直接回收不存活对象，造成内存碎片
   4. 标记-整理：
      1. 标记方式类似于标记-清除，但是清除的时候不同，在回收不存活的对象占用空间后，将所有的存活对象，往左端空闲空间移动，并更新对应的指针。解决了内存碎片化
      2. 缺点：成本高
   5. 复制算法
      1. 将堆分为一个对象面，和两个空闲面，程序从对象面为对象分配空间，对象满了的时候，则将活动对象复制到空闲面，空闲面就成了对象面，原本的对象面成了空闲面
      2. Stop-and-copy算法，在切换过程中暂停程序执行
   6. 分代收集：
      1. 分为 年轻代，年老代（新生代，老年代）
      2. 所有新生对象在新生代
      3. 新生代内存分为一个Eden，两个survivor，比例8:1:1；大部分对象在Eden生成，回收时先将Eden区存活对象复制到一个survivor0中，然后清空Eden，下一次minor gc, 将Eden和S0中存活的对象复制到S1，清空S0和Eden，在下次minor gc，将Eden和S1中存活对象复制到S0中，清空Eden和S1，如此循环往复，知道有对象存活次数超过阈值15，则进入老年代；老年代满了就触发full GC（major GC）新生代，老年代一起回收。
      4. 老年代：在新生代中经历了N次（15）回收依然存活的对象，就会被放入老年代中，
      5. 内存比新生代大很多，1:2
5. GC收集器（垃圾收集器）
   1. 新生代：serial，PraNew, Parallel Scavenge
   2. 老年代：CMS，Serial Old，Parallel Old
   3. Serial：（复制算法）：新生代，单线程，标记和清理都是单线程，优点，简单高效
   4. Serial Old：（标记-整理）：老年代，单线程
   5. ParNew：（停止-复制算法）：新生代，多线程，serial的多线程版本
   6. Parallel Scavenge：（停止-复制算法）：并行收集器，吞吐量高达99%
   7. Parallel Old：（停止-复制算法）：Parallel Scavenge的老年代版本
   8. CMS：（标记-清除）：高并发，低停顿，最短GC回收停顿时间，CPU占用高
6. JVM内存调优：
   1. 使用console查看内存的真实占用情况
   2. 调优目的：减少GC的频率和Full GC的次数，因为过多的GC和Full GC会占用很多的系统资源（主要是CPU），影响系统的吞吐量
   3. 调优方式：
      1. 从最小堆分配开始，根据应用程序实际需要来逐渐增加最小堆，可以通过指令：-XX：+HeapDumpOnOutOfMemoryError来指示JVM发生OOM异常时候倾倒堆数据
      2. 使用Java性能工具：VisualIVM，Java Mission Control等，用于跟踪应用程序的性能
      3. 使用StringBuilder 来代替“+”操作符
      4. 避免使用迭代器
      5. 更好的并行控制
   4. 造成过多的Full GC的原因：
      1. 老年代空间不足：尽量避免让对象在新生代GC时被回收，让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在老年代创建对象
      2. System.gc()被显示调用，尽量不要手动触发，尽量依靠JVM自身的机制
      3. 调优主要是通过控制堆内存各个部分的比例和GC策略来实现的
         1. 新生代过小：造成新生代GC次数频繁，增大系统消耗；导致大对象直接进入老年代，占据了老年代剩余空间，引起Full GC
         2. 新生代过大：造成老年代过小，引发Full GC；新生代GC耗时大幅度增加，一般新生代占整个堆的1/3就可以
         3. Survivor过小：造成对象从Eden直接到达老年代，降低了在新生代的存活时间
         4. Survivor过大：造成Eden过小，增加了GC频率，
      4. 选择合适的GC策略和组合搭配
         1. 吞吐量优先：
         2. 暂停时间优先：
7. Exception和Error
   1. Error：由JVM生成并抛出，大多数错误与代码编写者的操作无关，例如OOM，一般JVM会选择终止线程；
   2. Exception：
      1. CheckException：必须在源码里显示的捕获，是编译器检查的一部分；IOException
      2. RuntimeException：通常是可以编码避免的逻辑错误；Null PointException，ClassCastException，
8. 如何让JVM的方法区直接爆满
   1. 循环调用动态代理，生成类以及类的实例
9. 分布式缓存和一致性hash：
   1. Hash环（一致性hash）
      1. Hash环有两次hash
         1. 把所有的机器编号hash到这个环上
         2. 把key也hash到这个环上，然后在环上进行匹配，看key和哪台机器匹配
         3. 具体来说：一个hash函数，值空间为（0到2的32 次方-1），hash值是个32位的整型数，这些数字组成一个环；然后先对机器进行hash（例如机器的IP地址），算出每个机器在这个环上的位置；再对key进行hash，算出key在环上的位置，然后从这个位置向前走遇到的第一台机器就是该key对应的机器，就把这个（key,value）存在这个机器上；
      2. 数据不均匀解决：
         1. 当机器不多，可能出现几台机器在环上贴的很近，不是均匀分布，会大致大量数据集中在一个机器上；
         2. 解决办法是，使用虚拟机器，例如一台机器在环上计算出多个位置，可以在机器的IP后加上编号，例如，ip\_1,ip\_2,ip\_3把一台物理主机生成多个虚拟机器的编号
         3. 数据先映射到虚拟机器，再从虚拟机器映射到物理机器；
      3. 服务器的机器增加，减少，如何通知客户端
         1. 手动重新配置客户端，重启客户端
         2. 引入zookeeper，服务器的节点列表注册到zk上，客户端监听zk,发现节点数发生变化，自动更新配置
         3. 引入Redis Cluster（集群分片）：
            1. Redis集群是P2P模式，无中央节点，节点之间通过Gossip的协议通信
            2. 有16384个hash槽，分配给不同的节点，存放数据，根据数据的key计算出所在的槽，根据槽找到对应的机器，hash函数为：

CRC16（key）%16384;

* + - * 1. 16384个槽的原因:为了节省存储空间，每个节点用一个Bitmap存放其对应的槽；2K = 2\*1024\*8，也就是，每个节点用2K的内存空间，总共16384个比特位，就可存储该节点对应了那些槽
        2. Redis-cluster引用了主从复制，从而提供了fail-over机制，解决了缓存雪崩问题；
  1. 常见的hash算法：
     1. MD5算法：确保信息传输完整一致。将大容量信息压缩为一种保密格式，即将任意长度字节串变换为定长的16进制数字串
     2. CRC算法：循环冗余校验
     3. MurmurHash算法：高运算性能，低碰撞概率，Redis，Lucene中使用
     4. FNV算法：快速hash大量数据保持较小的冲突率，适用于hash一些接近的字符串例如：URL，hostname，文件名，text，IP地址等

1. 类加载机制：

加载：JVM将class二进制文件加载到内存（使用类加载器classloader）

连接：验证：确保类加载的正确；准备：为类的静态变量分配内存，初始化为默认值；解析：将类中的符号引用转化为直接引用

初始化：为类的静态变量赋予正确的初始值；

1. 类加载器：

引导类加载器：bootstrapclassloader

扩展类加载器：extensionsclassloader

系统类加载器：system class loader 根据Java应用的类路径加载Java类

自定义类加载器：custom class loader

双亲委派模式：某个特定的类加载器在接到加载类的请求，首先将加载任务委托给父类加载器，父类加载器又将任务向上委托，直到最父类加载器，如果最父类加载器可以完成就成功返回，如果不行就向下传递委托任务；

好处：保证Java核心库的安全性（例如自己写了一个String类，就会因为这个机制不能被加载）

类初始化步骤：如果这个类还没有被加载和连接，就加载和连接；如果存在直接父类，且父类没有被初始化，先初始化父类；类中存在static标识的快，先执行这些；

1. Tomcat类加载机制：

**Tomcat的类加载机制是违反了双亲委托原则的，对于一些未加载的非基础类(Object,String等)，各个web应用自己的类加载器(WebAppClassLoader)会优先加载，加载不到时再交给commonClassLoader走双亲委托**

**Tomcat8可以配置<loader delegate = “true”/>不打破双亲委派**

**类加载器：BootStrap 加载JVM启动所需要的类和系统扩展目录里jar文件里的类；System 加载Tomcat启动的类； Common 加载对Tomcat内部类和所有web应用程序都可见的其他类 jar文件 catalina home/lib catalina base/lib; webapp 为部署在单个Tomcat实例中的而每个web应用程序创建一个类加载器，加载WEB-INF、classes和WEB-INF/lib 里的jar中的类；**

**查找顺序：Bootstrap---》system---》Common---》多个webAPP**

**Tomcat打破了双亲委派顺序：当某个请求想要从web应用的webAPP类加载器中加载类的时候，该类加载器会先查看自己的仓库，而不是预先进行委托处理；**

1. Java用自定义类型作为hashmap的key

需要重写hashcode方法和equals方法

1. 面向对象三特性，封装，继承，多态；五大准则

封装：将现实世界的事物抽象出计算机领域中的对象，对象同时具有属性和行为；

继承：实现复用性，可以在不重复编写以实现的功能的前提下，对功能进行复用和拓展；

多态：允许不同类的对象对同一消息做出响应。多态包括参数化多态和包含多态

单一职责（single Responsibility Principe）：一个类的功能单一；

开闭原则（Open-Close Principe）：对扩展开放，对修改关闭；

替换原则：（Liskov Substitution Principle ）:子类可以替换父类并出现在父类出现的任何地方；

依赖原则：（Dependency Inversion Principe）：具体依赖抽象，上层依赖下层；

接口分离原则：（Interface Segregation Principe）：模块间通过抽象接口隔离开；

1. 深拷贝与浅拷贝：

浅拷贝：

对于基本数据类型，浅拷贝会直接进行值传递，是两份不同数据，修改其中一个对象，不影响另一个；

对于引用数据类型，浅拷贝进行引用传递，复制的变量的引用值（内存地址），修改一个，另一个也改变；

深拷贝：

对引用数据类型的成员变量都开辟了内存空间；而浅拷贝知识传递地址指向；

深拷贝实现方法：

1. 重写clone方法
2. 通过对象序列化实现深拷贝

三．数据库

1. InnoDB使用B+树作为索引的原因

首先比较各种树的优劣势：

1. 二叉树：

有序二叉查找树，左子树小于根节点，右字数大于根节点，左右子数也是二叉查找树；

对于某些情况，二叉查找树会退化为一个有N个节点的线性链；

1. AVL树：

平衡二叉查找树，所有节点的左右子树高度差不超过1，因此执行插入还是删除操作，只要不满足平衡条件，就通过自旋来保持平衡

由于维护这种高度平衡所付出的代价比从中获得的效率收益还低，因此实际运用不多

1. 红黑树：

二叉查找树，但是每个节点增加一个存储位 表示节点的颜色red OR black。通过对任何一条从根到叶子节点的路径上各个节点的着色方式的限制，红黑确保没有一条路径会比其他路径长出两倍。是一种弱平衡二叉树

每个节点非红即黑，根节点黑，每个叶节点黑，如果一个节点时红，那么他的儿子节点是黑，对于任意节点而言，其到叶子节点的每条路径都包含相同数目的黑节点；

Treemap引用红黑树；

1. B、B+树

B树：

任意非叶子节点最多只有M个儿子，且M>2;

根节点的儿子书为[2,M]

除根节点外非叶子节点的儿子数为[M/2,M]

每个节点存放至少M/2-1 和至多M-1个关键字

所有叶子节点位于同一层

B+ 树：

非叶子节点的子树指针与关键字个数相同

非叶子节点相当于叶子节点的索引，叶子节点相当于是存储数据的数据层；

选取B+ 树的原因：

1. B+ 树的磁盘读写代价更低，
2. B+ 树的查询效率更加稳定
3. B+ 树的数据都存在叶子节点中，非叶子节点为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树的分支节点同样存储数据，我们找到具体的数据需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询，所以选用B+树作为数据库的索引
4. Float，Decimal存储金额的区别：
   1. Decimal类型可以精确的表示非常大或者非常精确的小数，大致1028以及有效位数多达28位的数字可以作为decimal类型存储而不失其精确性。
   2. Float是浮点数，不能指定小数位，decimal是精确数，可指定精度
   3. Decimal数据类型可以存储最多38个数字；
5. Datetime和Timestamp存储时间的区别：
   1. 两者存储方式不同：
      1. Timestamp，将客户端插入的时间从当前时区转化为UTC进行存储，查询时又将其转化为客户端当前时区进行返回
      2. Datetime，不做任何改变，基本是原样输入和输出
   2. 两者存储时间范围不同：
      1. Timestamp：1970-01-01 00:00:01.000000 到2038-01-19 03:14:07.999999；
      2. Datetime：1000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999
      3. 对于跨时区的业务timestamp更加适合；
6. Char，VarChar，Varbinary存储字符的区别：
   1. Char：长度不可变，例如char[10] 存入CSDN，那么所占字符依旧是10位，后面跟了6个空格，取数据的时候，char类型的要用trim（）去掉多余的空格，char类型存取速度比varchar快；char类型存储方式是对英文字符占用1个字节，一个汉字占2个字节
   2. Varchar：长度可变，以空间效率为首位，存储方式是英文字符2个字节，汉字2个字节
7. Mysql索引相关：
   1. 索引可能会提高查询速度，但一定会减慢写入速度，因为每次写入需要更新索引，所以索引应该加载常需要搜索的列上，不应该加载写多读少的列上；
   2. 聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针
   3. InnoDB存储引擎：主键索引默认是聚簇索引，二级索引会在叶子节点保存主键，当通过二级索引查找的时候，InnoDb需要通过二级索引的叶子节点获得对应的主键，然后根据主键索引查找对应的行；
   4. MyIsam存储引擎：主键聚集数据，没有聚簇索引，主键索引和二级索引工作方式一样：叶子节点存储数据行在数据文件中的物理偏移量（行指针）；
8. B+树如何优化，索引遵循的原则：
   1. 索引遵循最左性原理，例如联合索引是（a,b,c），在查询时查询条件是a, ab, abc的精确匹配，都会使用索引；
      1. 但是如果不是连续的 ，例如 ac,不会使用索引，
      2. b, c, 单独精确查询不会用到索引
      3. 使用like语句，通配符不在开头可以使用索引；
      4. 范围列查询只能使用第一个列索引，
      5. 使用函数或者表达式对列查询，不会使用索引
   2. 为什么mysql索引使用B+ 树
      1. B+ 树查询时间：跟树的高度有关，是O（Logn）
      2. Hash 索引查询时间：O（1）
      3. 使用B+树原因是：如果只查找一个值，hash是一个很好的选择，但是数据库经常会查询多条数据，由于B+ 树索引有序，并且又有链表相连，查询效率就比hash快很多了，而且数据的索引一般是在磁盘上，数据量大的情况下，可能无法一次装入内存，B+ 树的设计可以允许数据分批次加载，同时，树的高度较低，提高了查询效率；
9. 数据库设计三范式：
10. 1NF，无重复的列（原子性）

数据库表中的每一列都是不可分割的基本数据项。而不能是集合，数组，记录等非原子性的数据项。即实体中的某个属性有多个值的时候，必须拆分为不同的属性；

第一范式是基本要求，不满足则不是关系数据库；

1. 2NF，属性完全依赖于主键

也就是非主属性完全依赖于主关键字；

例如，选课关系表中（学号，姓名，年龄，课程名，成绩，学分），关键字为组合关键字（学号，课程名）；

决定关系如下：

学号，课程 ------------》 姓名，年龄，成绩，学分

不满足2NF;

存在数据冗余，例如同一门课N个学生选，；学分就重复了N-1次，同一个学生选了M们课，姓名和年龄就重复了M-1次

更新异常，若套装了某门课学分，数据表中所有行的学分都要更新，否则就会出现同一门课学分不同；

插入异常，假设要是开设一门新的课程，暂时没人选，那么没有学号关键字，课程名和学分也就无法记录入数据库；

删除异常，假设一批学生已经完成了课程的选修，那么选修记录应该被删除，课程名和学分也就被删除了；

因该分为3个表：学生表：学号，姓名，年龄；课程表：课程名，学分，选课关系表：学号，课程名，成绩；

1. 3NF，属性不依赖于其他非主属性（在2NF基础消除传递依赖）

例如，部门信息表，每个部门都有编号，名称，简介等信息，那么在员工表中列出部门编号后就不能再将部门名，简介等信息加入到员工信息表中了；否则就会有大量冗余数据；简而言之，就是属性不依赖于其他非主属性；

1. 数据库中有哪些聚集函数
   1. Sum
   2. Count
   3. Ave
   4. Max
   5. Min
2. 为什么Myisam比INNODB查询快？
   1. 主要因为INNODB在做select，要维护东西比MYISAM多；
   2. INNODB要缓存数据块，而MYISAM只缓存索引块；
   3. INNODB寻址要先映射到快，再到行，MYISAM记录的直接是文件的OFFSET，定位比INNODB快；
   4. MYISAM表锁，牺牲了写性能，提升了读性能；
3. MySQL数据量大如何解决？
   1. 优化SQL和索引
   2. 加缓存，redis；
   3. 主从复制，读写分离
   4. MySQL自带分区，SQL语句条件加上分区条件的列，避免扫描全表
   5. 垂直拆分；
   6. 水平切分
4. Mysql可重复读隔离级别的实现原理：
   1. 定义：A事务读到一条数据，此时事务B对数据修改并提交，事务A再读该数据，还是原来的内容
   2. 使用的是MVCC控制方式：Multi－Version　Concurrency　Control，多版本并发控制，类似于乐观锁；
   3. INNODB在每行记录后面隐藏了两列，分别保存了这行的创建时间（CreateTime），删除时间（DeleteTime），存储的不是实际的时间值，而是版本号，当数据被修改了，版本号加一；在读取事务开始时，系统会给当前读事务一个版本号，事务会读取版本号＜＝当前版本号的数据；此时如果其他事务修改了数据，那么这条数据的版本号加一，从而比当前读事务版本号高，就读不到更新后的数据；
5. 计算机网络相关
6. OSI：物理，数据链路，网络，传输，会话，表示，应用
7. TCP/IP：网络接口， 网络， 传输， 应用
8. 五层协议：物理，数据链路，网络，传输，应用
9. 每层作用：
   1. 物理：为上次协议提供传输数据的物理媒介
   2. 数据链路：物理地址寻址，数据的成帧，流量控制，数据检错，重发
   3. 网络：负责对子网见的数据包进行路由选择
   4. 传输：端到端，主机到主机的层次，TCP/UDP
   5. 会话：负责建立，管理，终止进程之间的会话
   6. 表示：对上层数据或信息进行变换保证一个主机应用可以被另一个主机应用程序理解；数据加密，压缩，格式转换
   7. 应用：为操作系统或网络应用程序提供网络服务的接口
10. TCP/UDP
    1. TCP 有连接，可靠 （可靠性通过顺序编号和确认ACK来实现）
    2. UDP 无连接，不可靠
11. TCP/UDP对应的协议：
    1. TCP对应协议：
       1. FTP：文件传输协议　２１　端口
       2. Telnet：远程登录，　２３端口
       3. SMTP：简单邮件传送协议　２５端口
       4. POP３：接受邮件协议　１１０端口
       5. HTTP协议：　超文本传输协议
    2. UDP对应协议：
       1. DNS：域名解析服务，将域名地址转换为IP地址，　５３端口
       2. SNMP：简单网络管理协议　管理网络设备的，　１６１端口
       3. TFTP：简单文件传输协议　６９　端口；
12. TCP 三次握手：
    1. 主机A发送连接请求 SYN：seq=X
    2. 主机B确认连接请求 SYN：seq=Y ACK: ack=X+1
    3. 主机A确认连接请求 SYN: seq=X+1 ACK: ack = Y+1
13. 客户端发起握手，进入SYN\_SEND
14. 服务端监听到连接请求，发送给SYN+ACK，进入SYN\_RECV
15. 客户端发出连接确认，SYN+ACK，进入established

为什么要三次握手？

第一次握手，客户端发，服务端知道客户端可以发信息

第二次握手，服务端收和发，客户端知道服务端可以收和发信息

第三次握手，客户端收和发，服务端知道客户端可以收和发

如果第三次握手失败，只有客户端处于成功状态，那么服务器会有定时其发送第二不的SYN+ACK数据包，如果客户再次发送了ACK成功则建立连接；

如果一直不成功，服务器或有超时设置，超时之后会给客户端发送RTS保温，进入CLOSED状态，防止SYN洪泛攻击；

1. 四次握手关闭连接：
   1. 主机A 发送FIN：seq = X
   2. 主机B 发送ACK：ack = X+1
   3. 主机B 发送FIN：seq=Y, ACK: ack=X+1
   4. 主机A 发送ACK： ack = Y+1
2. 主机A关闭连接，发送FIN，
3. 主机B响应关闭，发送ACK，先发ACK是为了防止主机A重发FIN
4. 主机B关闭连接，发送FIN，ACK
5. 主机A响应关闭：发送ACK，主机A等待2MSL后如果没有收到回复，则表示主机B正常关闭，那么主机A关闭连接；
6. 为什么连接三次，关闭四次

TCP是全双工模式，关闭连接，主机B收到主机A的FIN，仅仅表示主机A不再发送数据，但是还能接受数据；

主机B也未必全部数据都发完了；

1. IP报文的结构

版本号 4位， IPV4 为4，IPV6为6

首部长度：占4位白哦是首部的长度；

区分服务：占8位

总长度：占16位

标识：占16位

标志：占3位

片偏移：占13位

寿命TTL：占8位

协议：占8位

首部校验和：占16位

源IP地址和目的IP地址：各占32位

IP选项：可选项，长度可变

记录路由选项：

源路由选项

时间戳选项

填充

1. TCP的滑动窗口协议：
   1. 作用：提供TCP可靠性，对发送数据进行确认；流量控制：窗口大小随链路变化
   2. 机制：TCP协议一次传输多少个数据。在传输时，TCP会对所有数据进行编号，发送方在发送过程中始终保持一个窗口，只有落在发送窗口的数据帧才允许被发送；接收方也始终保持一个接受窗口，只有落在窗口内的数据才会被接收
2. HTTP协议的方法：
   1. PUT
   2. GET
   3. POST
   4. DELETE
   5. HEAD
3. HTTP请求报文
   1. 请求方法　URI　协议版本　GET／test．ｊｓｐ　ＨＴＴＰ／１．１
   2. 请求头
   3. 请求正文
4. HTTP响应报文
   1. 协议版本　响应码　描述　ＨＴＴＰ／１．１　２００　ＯＫ
   2. 响应头
   3. 响应正文
5. HTTP和HTTPS
   1. HTTP缺点：
      1. 通信明文不加密
      2. 不验证通信方身份
      3. 无法验证报文完整性
   2. HTTPS：
      1. 在HTTP上加上SSL（TSL）加密处理＋认证＋完整性保护
      2. SSL过程：
         1. 客户端请求ｈｔｔｐｓ连接，服务端返回公钥
         2. 客户端产生随机（对称）密钥
         3. 使用公钥对对称密钥加密
         4. 发送加密后的对称密钥
         5. 通过对称密钥加密的密文通信
      3. 具体过程：
         1. 客户端使用HTTPS的URL访问web服务器，要求建立SSL连接
         2. Web服务器产生一对公钥和私钥，公钥放在证书里发给客户端
         3. 客户端浏览器根据双方同意的SSL连接，建立会话密钥，用公钥加密密钥，传送给服务端
         4. 服务端用自己的私钥解密出会话密钥
         5. 服务端利用会话密钥加密与客户端之间的通信
6. DNS解析过程：
   1. 浏览器检查缓存中又没有这个域名对应的IP地址，有则结束解析
   2. 缓存无，则先检查自己本地的ｈｏｓｔｓ文件是否有这个网址映射欢喜，有，则调用IP地址映射，解析
   3. Hosts里无的话，查找本地DNS解析器缓存，是否有这个网址映射，有，则返回，完成解析
   4. Hosts与本地DNS解析器都没有，首先会找TCP/IP参数中设置的首选DNS服务器，查询
   5. 如果域名不由本地DNS服务器解析，当该服务器缓存了此网址的映射关系，则调用这个IP地址映射，完成解析
   6. 如果本地DNS服务器与缓存解析都失效，则根据本地DNS服务器设置进行查询，如果未用转发模式，本地DNS就把请求发给13台根DNS，根NS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找域名域服务器，重复上面的动作，进行查询，直至找到域名对应的主机。
7. TCP如何保证数据的可靠性：
   1. 通过序列号和确认应答信号提高可靠性
   2. 超时重发
   3. TCP连接管理
   4. TCP以段为单位进行数据包的发送的
   5. 利用滑动窗口控制提高速度
   6. 滑动窗口控制与重发控制
   7. 滑动窗口实现流量控制
   8. 拥塞控制
8. 浏览器输入网址并回车的过程
   1. DNS域名解析，找到对应的IP地址
      1. 浏览器缓存
      2. 系统缓存
      3. 路由器缓存
      4. ISP DNS缓存
      5. 递归搜索
   2. 浏览器与服务器建立TCP连接
   3. 浏览器发出文件命令
   4. 服务器发出响应
   5. 释放TCP连接
   6. 浏览器绘制网页
9. OSI每一层的协议：（SSL）协议位于TCP/IP协议与各种应用层协议之间（会话层）
   1. 应用层：HTTP，FTP，SMTP， TFTP，NFS，WAIS
   2. 表示层：Telnet， Rlogin，SNMP
   3. 会话层：SMTP， DNS
   4. 传输层：TCP， UDP
   5. 网络层：IP，ICMP，ARP，PARP
   6. 数据链路层：FDDI，PPP，
   7. 物理层：IEEE 802.1A IEEE802.2到IEEE802.11
10. Content/type类型：
    1. Application/json：消息主体是序列化后的json字符串
    2. Text/HTML：
    3. Multiparty/form-data：常见的post数据提交方式
    4. Application/x-www-form-urlencoded：浏览器原生表单；
11. Get/post区别：
    1. Get请求再URL中传输的参数有长度限制，post没有；
    2. Get比post更加不安全，参数直接暴露在URL
    3. Post参数通过requestbody传递
    4. Get请求参数会保留在浏览器记录，post不会；
    5. Get请求只能进行URL编码，post支持多种编码
    6. Get请求会被浏览器主动cache，post不会；
    7. 对于Get方式，浏览器会把HTTP header和data一起发送，服务器响应200
    8. 对于post方式，浏览器先发送header，服务器响应100（continue），浏览器再发送data，服务器响应200 Ok；
12. Cookie和session的区别和应用场景
    1. Session保存再服务器，客户端不知道其中的信息；cookie保存再客户端，服务器能够知道其中的信息
    2. Session中保存的是对象；cookie中保存的是字符串
    3. Session不能区分路径，同一个用户再访问一个网站期间，所有的session在任何一个地方都可以访问到；cookie中如果设置了路径参数，那么同一个网站中不同路径下的cookie互相是访问不到的
    4. 应用场景：
       1. Session用来保存上下文信息的机制，针对每一个用户的，变量的值保存在服务器，通过sessionID来区分不同的客户；
       2. Session可用来做登录验证信息；
       3. Cookie作用是与服务器进行交互，作为HTTP规范的一部分，可用于判断用户是否登录过网站，用来记录购物车；
13. 如何提高cookie的安全性
    1. 对保存到cookie的敏感信息加密
    2. 给cookie设置有效期
    3. 设置Http－Only　为true；
14. Redis相关
15. Redis单线程，基于内存的操作，因此内存是瓶颈，而与CPU无关，；
16. 多路复用IO，非阻塞式的IO，，在空闲的时候，会把当前线程阻塞掉，当有一个或多个IO事件的时候，就从阻塞状态中唤醒，于是程序会轮询一遍所有的流，并且只依次顺序的处理就绪的流，避免了大量的无用操作；
17. 支持多种数据结构，包括String，List，Hash，Set，Sorted　Set，BitMap，Hyper log log，也就是，字符串，集合，哈希表，有序集合，链表，位图，；
18. 线程安全，所有操作原子，速度快（）由于非阻塞IO，大部分操作时间复杂度位O（１）
19. 采用Key－Value的基本数据结构，任何二进制序列都可以作为Key使用，

使用注意：

不要使用过长的Key，消耗内存，导致查找效率低

Key也不可过短，引发可读性，可维护性问题

使用同一规范来设计Key

Redis允许的最大Key长度我５１２MB；，对Value的限制也是５１２MB；

1. 数据持久化，将数据定期自动持久化到硬盘的能力，包括RDB和AOF两种方案
   1. RDB方式：
      1. Redis定期保存数据快照到一个rbd文件中，并在启动的时候自动加载rbd文件，恢复之前保存的数据，例如save 60 100 表示60秒内发生了100 次数据修改，则进行一次RBD快照保存
      2. 可以配置多条RBD命令，让Redis执行多级快照保存策略；
      3. Redis默认开启RBD快照；默认策略如下：
         1. Save 900 1；
         2. Save 300 10；
         3. Save 60 10000；

也可以通过BG SAVE 命令手动触发RBD快照保存；

RBD的优点：

对性能影响小，保存快照会fork出子进程进行，几乎不影响Redis处理客户端请求效率

使用RBD恢复数据比AOF快；

RBD的缺点：

快照是定期生成，所以Redis Crash的时候或多或少会丢失一部分数据

* 1. AOF方式
     1. 采用AOF，Redis会把每一个写请求都记录在一个日志里
     2. Redis重启的时候，会把AOF日志的所有写操作顺序执行一遍
     3. AOF默认关闭的
     4. 优点：
        1. 安全，
        2. 发生断电也不会损坏
        3. AOF文件易读，可修改
     5. 缺点：
        1. AOF文件比RBD文件大
        2. 性能消耗比RBD高
        3. 数据恢复速度比RBD慢

1. 内存管理于数据淘汰机制：
   1. 5中数据淘汰机制：（当内存不足以容纳新写入数据的时候）
      1. Volatile-lru : 使用LRU算法进行数据淘汰，淘汰上次使用时间最早的，且使用次数最少的key, 只淘汰设定了有效期的key
      2. Allkeys-lru : 使用LRU算法进行数据淘汰，所有的Key都淘汰
      3. Volatile-random ： 随机淘汰数据，只淘汰设定了有效期的key
      4. Allkeys-random : 随机淘汰数据，所有的key都淘汰
      5. Volatile-ttl : 淘汰剩余有效期最短的key
      6. no-envivtion：内存驱逐策略，即内存不足以容纳新写入数据，新写入操作报错
   2. 一般推荐使用volatile-lru,并辨识Redis中保存数据的重要性，对于重要的，应不设置有效期；
2. Redis性能调优：
   1. 避免使用长耗时命令，绝对禁止使用keys命令，
   2. 使用长连接或连接池，避免频繁创建销毁连接
   3. 根据数据的安全级别指定合理的持久化策略
   4. 引入读写分离机制
3. 主从复制
   1. 一个master实例负责处理所有的写请求，master将写操作同步到所有的slave
   2. 可以实现读写分离和高可用
   3. 借助redis sentinel可以实现高可用，当master crash后，redis sentinel能够自动的将一个slave晋升位master，继续提供服务
4. 集群分片：
   1. 原因
      1. Redis中存储的数据量大，一台物理主机内存无法容纳
      2. Redis的写请求并发量大，一个redis实例无法承载
   2. 集群能力
      1. 能够自动的将数据分散在多个节点上
      2. 当访问的key不在当前的分片，能够自动的将请求转发到正确的分片；
      3. 当集群中部分节点失效时仍然能提供服务
      4. 第三点是由于主从复制实现的，每个分片都采用了主从复制，区别是省去了redis sentinel，由redis cluster负责进行一个分片的节点监控和自动failover
5. Redis分布式锁：
   1. 先用setnx来争抢锁，抢到以后，再用expire给锁加一个过期时间，防止锁忘记了释放
   2. 如果在setnx之后执行expire之前进程crash或者重启维护了，那么锁就永远得不到释放了，但是set指令可以同时吧setnx和expire合成一条指令来用；
6. Redis集群最大节点个数

16384. 哈希槽数量

1. 缓存穿透：
   1. 查一个不存在的key，由于缓存是不命中时，从数据库查询，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直接去数据库查询，造成缓存穿透
   2. 解决办法：
      1. 对所有有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。最常见采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一定不存在的数据会被这个bitmap拦截
      2. 粗暴的办法：如果查询返回数据为空，仍然将空结果缓存，但过期时间设置短
2. 缓存雪崩：
   1. 如果缓存集中在某一个时间内失效，发生大量的缓存穿透，所有查询都在数据库上，则造成缓存雪崩
   2. 解决办法：
      1. 缓存失效后，通过加锁或者队列来控制数据库写缓存的线程数量。比如某个key只允许一个线程查询数据和写缓存，其他线程等待
      2. 通过缓存reload机制，预先更新缓存，再将即将发生大并发访问签手动触发加载缓存
      3. 不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀，可以在原有的缓存失效时间上加一个随机值（1-5分钟），
      4. 做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效了，可以访问A2，A1失效时间设置短，A2失效时间设置长
3. 缓存击穿
   1. 针对热点key，在失效的时候，发生大并发访问
4. 缓存热点key
   1. 当前key是一个热点key，并发量非常大的时候
   2. 重建缓存不能在短时间完成，
   3. 以上两个问题出现后，会有大量的线程来重建缓存，造成后端负载过大，甚至或让应用奔溃
   4. 解决办法：
      1. 互斥锁（mutex key）：只允许一个线程重建缓存
      2. 热点数据设置为永久
5. 分布式事务的几种实现方式：
   1. ２PC
      1. 两阶段提交：引入一个作为协调者的组件来统一的掌控全部的节点并指示这些节点是否把操作结果进行真正的提交；
      2. 分为两个阶段：投票阶段，提交阶段
      3. 投票阶段：协调者会向事务的参与者询问是否可以执行操作的请求，并等待其他参与者的响应，参与者会执行相对应的事务操作并记录重做和回滚日志，所有执行成功的参与者会向协调者发送Agreement或者Abort表示执行操作的结果；
      4. 当所有的参与者都返回了确定的结果（Agreement或Abort），２PC就进入提交阶段，协调者会根据投票阶段的返回情况向所有的参与者发送提交或者回滚的指令；
      5. 当所有的参与者都决定提交事务，协调者会发送COMMIT，参与者完成操作并释放资源后向协调者返回完成消息，协调者接受到消息，结束整个事务；有参与者Abort，协调者会向事务的参与者发送回滚消息，参与者根据之前执行操作的回滚日志对操作进行回滚，并向协调者发送完成消息；
      6. 无论事务被提交还是回滚，所有资源都会被释放，事务也一定会结束；
      7. ２PC是一个阻塞协议，在执行过程中，如果事务的协调者永久宕机，事务的一部分参与者将永远无法完成事务；
   2. ３PC
      1. 为了解决２PC的一些问题，３PC引入了超时机制和准备阶段，如果协调者或者参与者在规定的时间没有接受来自其他节点的响应，就会根据当前的状态选择提交还是终止整个事务；
      2. 当参与者向协调者发送ACK后，如果长时间没有收到协调者的响应，在默认情况下，参与者会自动将超时的事务进行提交，不会向２PC中被阻塞；
   3. XA事务
      1. MySQL的INNODB引擎支持分布式事务；用的是２PC；事务管理器为协调者，资源管理器为参与者；
      2. XA能保证较强的一致性，但是在执行过程中会对相应的资源加锁，阻塞其他事务对改资源的访问；
   4. Saga
      1. 实现最终一致性，在一定的时间窗内，多个系统中的数据不一致可以接受，，过了时间窗以后，所有系统返回一致的结果；
      2. 将一系列的分布式操作转化为一系列的本地事务，每一个本地事务都会更新数据库并且向集群中的其他服务器发送一条新的消息来触发下一个本地事务；一旦本地的事务违反了业务逻辑而失败，那么就会立即触发一系列的回滚操作来撤回之前本地事务；
   5. LLT
      1. 长事务，对一些数据库资源有相对较长的一段时间，严重影响其他正常数据库事务的执行，SagaS解决此问题；
6. 高并发下分布式事务：
   1. 采用基于消息中间间的两阶段提交；
   2. 将一个分布式事务拆成一个消息事务（A系统的本地操作＋发消息）＋B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发送出来了，此时B会受到消息去执行本操作，如果本地操作失败，消息会重投，知道B操作成功；
7. Rocket MQ　分布式事务
   1. 分为三个阶段：发送Prepared消息，执行本地事务，发送确认消息
   2. 流程：（转账案例）
      1. A银行产生一条转账消息到MQ，此时MQ接受到的消息对B银行不可见
      2. A银行本地事务提交后，再向MQ发送一条确认事务提交的消息
      3. 此时MQ接受到消息对B银行可见
      4. B银行来消费消息，完成B银行的转账操作
      5. 在A银行系统设一个转账消息确认表，在B银行系统设一个转账日志记录表；
      6. 如果A银行在第二阶段发送确认消息失败，导致B银行不能消费到消息，此时将B银行系统的转账日志记录表一段时间内的数据发送给A银行系统跟转账消息确认表做一个对账业务，找出发送失败的数据，然后再次向MQ发送确认消息；实现事务回查；
8. 分布式相关

酸（ACID）碱（BASE）平衡：分别利用ACID和BASE来解决分布式服务化系统的一致性问题；

ACID: atomicity原子性，consistency一致性，isolation隔离性，durability持久性

BASE：BA：basically available基本可用，S：soft state：软状态，一段时间内可以不同不，E：Eventually Consistent最终一致；

分布式事务解决：2PC，3PC，TCC（try，confirm，cancel）

1. CAP

一致性Consistency

可用性Availability

分区容忍性Partition tolerance

CAP原理：三要素最多只能同时满足两点，不可能同时兼顾三点，在分布式系统中，分区容忍性是基本要求，而大多是web系统不需要强一致性，因此可以牺牲一致性，换取高可用性；

即数据达到最终一致性即可，通过数据的异步复制来达到系统的高可用性和最终一致性。

2. 终一致性，以及实现方式：

对于一致性，即分为从客户端和服务端两个不同的角度。客户端：一致性主要为多并发访问时跟新过的数据如果获取的问题。服务端：则是更新如何复制分布到整个系统，以保证数据最终一致。

对于关系型数据库，要求更新过的数据能被后续的访问都能看到，即为强一致性；

如果能容忍后续的部分或者全部访问不到，即为弱一致性；

如果经过一段时间后，能访问更新的数据，即为最终一致性；

3．分布式事务：

指：事务的参与者，支持事务的服务器，资源服务器以及事务管理器分别位于不同的分布式系统的不同节点上；

产生的原因：

a service多个节点

b service产生多个节点

c Resource产生多个节点

4．Zookeeper的用途，选举原理：

a) zookeeper是一个分布式应用程序协调服务。

b) 设计目的：最终一致性；可靠性，实用性

c) 原理：原子广播：保证各个server之间的同步。实现的协议为Zab协议：有两个模式：恢复模式（选主），广播模式（同步）。当服务启动或者leader崩溃，进入恢复模式，leader被选出来，且同步完成后，进入广播模式；

d) 保证事务的顺序一致性：采用递增的事务id(zxid)来标识事务，所有的提议在被提出后都加上了事务id, 事务id是一个64位数，高32位用epoch标识leader关系是否改变，每一次leader被选出来，epoch都会有一个改变，标识当前属于哪个leader的统治期，低32位用于递增计数；

e) 选举原理：

1. 服务器ID， myid; 数据ID，Zxid; 逻辑时钟 epoch；server状态：looking，following，observing，leading

2. 首先，每个server读取自身的zxid

3. 发送投票信息：每个server第一轮投自己；投票信息：myid, Zxid, epoch,;epoch会随着选举轮数的增加而递增；

4. 接收投票信息：

每个Server发出一个投票。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下

　　　　· 优先检查ZXID。ZXID比较大的服务器优先作为Leader。

　　　　· 如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器。

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0，再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了Leader。

(5) 改变服务器状态。一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING。

5 数据的垂直拆分和水平拆分

a 水平拆分：将一张表的数据分为多张表来存放，也就是表的行拆分；插入的时候建立一张临时表来提供自增的ID。取到ID后通过运算得出数据应该存放的表，新表去掉自增的属性；

b 垂直拆分：指表的列的拆分。将不常用的字段单独放在一张表中，将text，blob等打字单拆分出来放在附表中，查询的时候需要join关联

6 MQ

类别有：

a ActiveMQ：Java编写的，高可用，单机吞吐量万级，时效性ms级

b RabbitMQ: erlang编写的，高可用，单机吞吐量万级，时效性us级

c RocketMQ: Java编写的，分布式架构，单机吞吐量十万级，时效性ms级

d Kafka： Scala编写的， 分布式架构，单机吞吐量十万级，时效性ms级以内

7 ActiveMQ的通信方式：

Pub/Sub : 发布订阅方式用于多个接受客户端的方式，作为发布订阅的方式，可能存在多个接收客户端，并且接受客户端与发送客户端存在时间上的依赖。一个接收端只能接受他创建以后发送客户端发送的信息。作为订阅者，在接受消息的时，有destination的receive方法，和实现message listener接口的onMessage方法；

P2P ：点对点通信，只有一个发送者，一个接受者；

8 解决消息重复消费问题：

1） 消息不丢失

2） 消息不重复执行；

一般可以在业务端加一张表，用来存放消息是否执行成功，每次业务事务commit后，告知服务端，已经处理过该消息，这样即使消息重发了，也不会导致重复处理；大致如下：业务端的表记录已经处理过消息的id，每次一个消息进来之前先判断该消息是否执行过，如果执行过就放弃，没有就执行，执行完成就存入表中；

9 ActiveMQ发送数据的方式；

同步发送：

异步发送：JMS就是典型的异步消息处理机制，JMS消息类型有：点对点，发布订阅；

10 ActiveMQ调优：

使用非持久化消息：

需要确保消息发送成功时使用事务来将消息分批组合；

五 Spring相关

* + - 1. 代理模式：
         1. JDK动态代理：面向接口生成代理，原理是类加载器根据接口，在虚拟机内部创建接口实现类；所以一定要有借口，并且真实的业务需要实现该接口；
         2. CGLIB动态代理：采用底层的字节码技术，可以为一个类创造一个子类，解决无接口类的代理类问题；但是不能对final修饰的类进行代理；
         3. CGLIB比JDK动态代理性能更高，但是花费时间更多。所对于单例对象，无需频繁重复的创建对象，用CGLB合适，反之JDK合适；
         4. SpringAOP采用的代理模式：源码中会判断代理的对象类是否有实现接口，有实现接口就直接走JDK分支，否则走CGLIB分支；
      2. SpringAOP事务相关：
         1. 声明式事务，使用一个事务拦截器，在方法的前后/周围进行事务性增强（advice），来驱动事务完成
         2. 事务的传播特性：

Required：如果当前没有事务，就开启一个事务

Supports：支持当前事务，如果当前有事务，就参与进来，如果没有事务，就以非事务方式运行

Required new：总是用一个新的独立事务

Never：不支持当前事务，如果当前有事物，就抛出异常

* + - * 1. 事务的隔离级别：（mysql默认可重复读）

Default 无法解决问题

读未提交：会出现脏读，不可重复读，幻读

读已提交：解决脏读

可重复度：解决脏读，不可重复读

序列化：解决全部问题

* + - 1. Spring IOC相关
         1. IOC是一种设计思想，即控制反转，将设计好的对象的交给容器控制
         2. IOC和DI：组件之间依赖的关系由容器在运行期决定，由容器动态的将某个依赖关系注入到组件中。
         3. BeanFactory：可以理解为一个Hashmap，key是BeanName，value是Bean实例，只提供put,和get功能
         4. ApplicationContext：定义refresh方法，用于刷新整个容器，子接口，classpathxmlapplicaitoncontext
         5. BeanDefinition: Bean对象在Spring中实现以BeanDefinition来描述；
         6. IOC的初始化：包括BeanDefinition的Resource定位，载入，注册；
         7. Spring IOC对Bean定义的资源的载入从refresh开始，在创建容器之前检查是否有容器存在，有就销毁和关闭，作用类似于对容器的重启；
         8. 将解析的BeanDefinition注册到容器中，
      2. SpringAop源码分析
         1. Spring进行数据织入的起点：spring加载Bean的时候进行的；
         2. 过程：

UserDAO对象

Spring环境加载

获取bean

JDK　Proxy　／　CGLIB　Proxy

执行通知

* + - * 1. 源码流程

代理创建

创建代理工厂，需要三个重要信息（拦截器数组，目标对象接口数组，目标对象）；

创建代理工厂，默认在拦截器数组尾部增加一个默认拦截器，用于最终的调用目标方法；

调用Get Proxy方法，会根据接口数是否大于０做判断，返回一个代理对象（JDK　Or　CGLIB）；

注意：创建代理对象时，同时会创建一个外层拦截器，就是Spring内核的拦截器，用于控制整个AOP过程；

代理的调用：

当对代理对象进行调用时，就会触发外层拦截器；

外层拦截器根据代理配置信息，创建内层拦截器链。创建过程根据表达式判断当前拦截器是否匹配这个拦截器。

当整个链执行到最后，就会触发创建代理时那个尾部的拦截器，从而调用目标方法；最后返回；

* + - 1. SpringMVC过程：
         1. 设置属性
         2. 根据request请求的URL得到对应的handler执行链，其实就是拦截器和Controller代理对象
         3. 得到handler的适配器
         4. 循环执行handler的ｐｒｅ拦截器
         5. 执行真正的handler，并返回ModelAndView
         6. 循环执行handler的post拦截器
         7. 根据ModelAndView信息的到View实例
         8. 渲染View返回；
      2. Spring IOC如何管理Bean之间的依赖关系，如何避免循环依赖：
         1. Spring框架的三级缓存解决了循环依赖问题
         2. 理论依据是基于Java的引用传递，当我们获取到对象的引用的时候，对象的字段或属性可以延后设置；
         3. 分三步：

CreateBeanInstance：实例化，调用对象的构造方法实例化对象

PopulatedBean：填充属性，多bean的依赖属性进行填充

initializeBean：调用spring xml中的init方法；

* + - * 1. 循环依赖发生在上述的第一，第二部，构造器循环依赖和field循环依赖
        2. 解决循环依赖从初始化着手，spring使用了三级缓存

singletonFactories:单例对对象工厂的cache

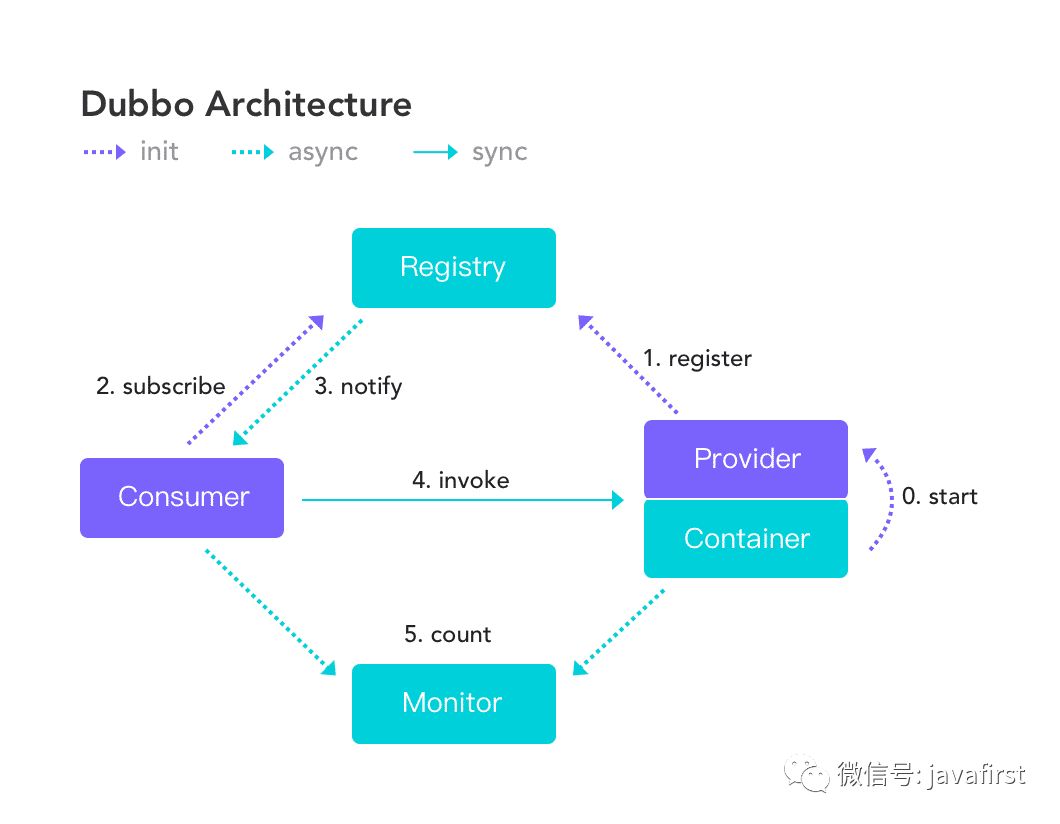
earlySingletonObjects: 提前曝光的单例对象的cache

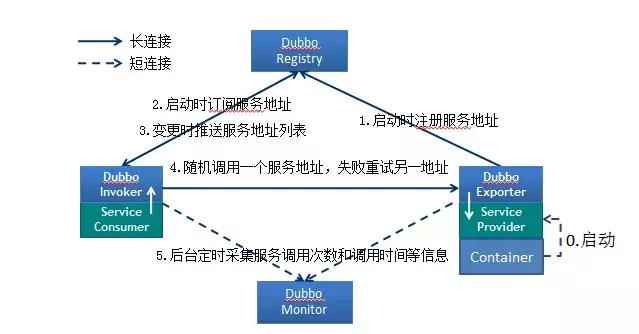
singletonObjects：单例对象的cache

* + - * 1. spring创建Bean，先从一级缓存中获取，获取不到且对象正在创建中，就再从二级缓存中获取，如果还是获取不到且允许SingletonFactories通过getObject获取，就从三级缓存中获取，如果获取到了，就从singletonfactories中移除，放进二级缓存earlySingletonObjects中
      1. mybatis和hibernate的区别：
         1. mybatis不完全是一个ORM框架，因为mybatis需要程序员自己编写sql语句，mybatis可以通过xml或者注解方式灵活的配置要运行的sql语句，并将Java对象和sql语句映射生成最终执行的sql,最后将sql执行的结果集再映射成Java对象；
         2. hibernate对象/关系映射能力强，数据库无关性好；
         3. mybatis是直接对数据库表操作
         4. hibernate是对与表有映射关系的实体类的操作，底层会自动生成相应的sql;

六． Dubbo相关：

1. Dubbo核心内容：
   1. 远程通讯：提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及请求-响应模式的信息交换方式；
   2. 集群容错：提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持；
   3. 自动发现：基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器；
2. Dubbo服务集群容错机制：
   1. Failover：默认模式，调用失败，自动切换，尝试调用其他节点可用服务，用于读操作；
   2. Fail fast：快速失败，调用只执行一次，失败立即报错，适用于写操作；
   3. Failsafe：失败安全，调用失败，直接忽略失败的调用，记录下是啊比的调用到日志文件；
   4. Fail back：失败恢复，后台记录失败请求，定时重发，通常使用与消息通知操作；
   5. Forking：并行调用多个服务器，一个成功即返回，适用于实时性要求高的读操作，但是浪费服务资源；
   6. Broadcast：广播调用所有提供者，逐个调用，任意一台报错则报错；，适用于通知所有提供者更新缓存或日志 等本地资源信息；
3. Dubbo的服务负载均衡：
   1. Random Load Balance：随机策略，配置值为random，可以设置权重，有利于充分利用服务器的资源，
   2. Round Robin ：轮询策略，配置值为roundRobin；
   3. Least Active：配置值为least active，根据 请求调用次数技术，处理请求慢的节点会受到更少的请求；
   4. Consistenthash：一致性hash策略，相同调用参数的请求发送到同一个服务提供节点上，如果某个节点发生故障无法提供服务，则会基于一致性hash算法映射到虚拟节点上；
4. Dubbo和SpringCloud的区别：
   1. 通信方式：Dubbo使用RPC通信，Spring cloud使用Http RESTFul；
   2. 组成部分不同：Dubbo注册中心：zookeeper，监控：Dubbo-monitor；
5. Dubbo包含节点角色：
   1. Provider: 暴露服务的服务提供方
   2. Consumer：调用远程服务的服务消费者；
   3. Register：服务注册与发现的注册中心；
   4. Monitor：统计服务的调用次数和调用时间的监控中心；
   5. Container：服务运行容器；
6. Dubbo服务注册与发现流程图：





1. 当一个服务接口又多种实现解决办法：
   1. 用group属性分组，服务提供方和消费方都指定同一个group；
   2. 使用版本号过渡，版本号不同的服务间互相不引用，
2. Dubbo默认同不等待阻塞，支持异步调用；
3. 服务提供者实现失效踢出基于zookeeper的临时节点原理；
4. Dubbo服务暴露的过程：
   1. 再spring实例化bean结束后，再刷新容器最后一步发布Context RefreshEvent时间的时候，通知实现了ApplicationListener的ServiceBean类进行回调onApplicationEvent事件方法，Dubbo会在这个方法中调用ServiceBean父类Service Config的export方法，此方法实现了服务的发布；
5. Dubbo服务提供者暴露一个服务的详细过程：
   1. Provider启动时候，先把提供的服务暴露在本地；
   2. 然后再把服务暴露给远程
   3. 启动netty服务，建立长连接；
   4. 连接到注册中心zk上
   5. 然后监控zk上的消费服务；
6. Dubbo的主要配置项：

Provider配置：

* 1. <dubbo:applicaiont name=””/>提供方应用信息
  2. <dubbo:regidtry address=””/>使用multicast广播注册中心暴露服务地址；
  3. <dubbo:protocol name=”dubbo” port=”20880”/>用dubbo协议在20880端口暴露服务
  4. <dubbo:service interface=”” ref=””/>声明要暴露的服务接口

Consumer配置：

* 1. <dubbo:application name=””/>消费方应用名
  2. <dubbo:registry address=””/>注册中心地址
  3. <dubbo:reference id=”” interface=””/>生成远程服务代理，可以和本地bean一样使用指定服务；

1. Dubbo的服务降级：
   1. 目的：为了保证核心服务可用；
   2. 分类：自动降级，人工降级； 读服务降级，写服务降级；
   3. 故障降级，例如调用服务挂了，直接降级；
   4. 限流降级，秒杀业务下，流量大，突发访问量特别大导致系统无法支撑，采用限流来限制访问量，达到阈值后，后续请求被降级，比如进入排队页面，挥着跳转到错误页面；
2. Dubbo降级方式：Mock
   1. 在client建立一个以mock结尾的类，实现需要的服务的接口
   2. 在xml配置文件中，添加mock配置指向mock类
3. Dubbo使用过程中遇到的问题：
   1. Dubbo设计的目的事满足高并发小数据量的RPC调用，再大数据量下性能表现不好，建议使用RMI或者HTTP；

七． 微服务相关：

1. 微服务的由来：

a. 随着技术的发展，网站规模扩大，需求激增，原先的集中式架构存在很多问题，例如，代码耦合，开发维护困难；

无法针对不同模块进行针对性优化；

无法水平扩展；

单点容错率低，并发能力差；

b. 当访问量增大，单一应用无法满足需求，将业务功能进行拆分：

优点：

系统拆分实现了流量分担，解决了并发问题；

可以针对不同模块进行优化；

方便水平扩展，负载均衡，容错率高；

缺点：

系统间相互独立，会有很多重复开发工作，影响开发效率；

c. 分布式服务：当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心。

优点:

将基础服务进行了抽取，系统之间相互调用，提高了代码复用和开发效率；

缺点：

系统之间耦合度变高，调用关系错综复杂，难以维护；

d. 服务治理（SOA）dubbo

当服务变多，此时需要增加一个调度中心，基于访问压力实时发u能力集群容量，提高集群利用率。

服务治理解决的问题：

服务注册中心：实现服务自动注册和发现，无需人为记录服务地址；

服务自动订阅：服务列表自动推送，服务调用透明化，无需关心依赖关系；

动态监控：服务状态监控，认为控制服务状态；

缺点：

服务间会有依赖关系，一旦某个环节出错影响较大；

服务关系错综复杂，运维，测试，部署困难；

e. 微服务spring cloud

微服务的特点：

单一职责：微服务每个服务都对应唯一的业务；

微：微服务的服务拆分粒度很小；

面向服务：每个服务都向外暴露服务接口API，并不关心服务的技术实现，只要提供Rest的接口即可；

自治：服务之间互相独立，互不干扰；

1. 远程调用方式：
   1. RPC：remote procedure call：远程过程调用，类似有RMI。自定义数据格式，基于TCP通信，速度快，效率高。例如dubbo；
      1. 具体了解RPC：
         1. 允许一台计算机册的程序调用另一台计算机的子程序；
         2. A程序提供服务，B程序通过网络请求参数传递给A，A本地执行结果返回给B程序；
         3. 采用的网络通讯协议
            1. RPC，采用TCP作为底层传输协议
         4. 数据传输格式：
            1. 定义请求和响应的格式，约定统一的序列化方式；
         5. RPC强调的是过程调用，调用的过程对用户透明，用户不应关心调用细节，像调用本地服务一样，因此对调用的过程进行封装；
      2. RPC调用过程：
         1. 从消费者看，当应用开始调用RPC的方式时，就会去容器中去取Bean对象，因此首先应注册Bean对象到容器中，通过Java的动态代理，将代理过程封装到代理对象中，代理对象实现接口，创建实例到容器中。相应的在调用远程对象的对象方法时，就会调用动态代理中的方法；
         2. 代理对象获得请求方法，接口和参数时，就会用序列化层将信息封装成一个请求报文，让通信层向服务端传送报文的内容，然后就到生成者；
         3. 相应的服务必须有监听器，来监听来自其他服务的请求，一般都会用容器做消息的监听，就会调用对应的bean对象的方法，来处理相应的请求。通过请求中的类，方法，参数，反射调用对应的bean，拿到结果后，通过序列化层，封装好结果报文，通信层将报文反馈给调用方，调用方解析到返回值，动态代理类返回结果，调用结束；
      3. RPC各层的作用：
         1. 消费者：
            1. 代理层：消费者将对应的接口，通过RPC框架的代理来生成一个对象放到spring容器里，代理层将代理接口生成该接口的对象，该对象处理调用时传过来的对象，方法，参数，通过序列化封装好，调用网络层；
            2. 序列化层：将请求的参数序列化成报文；将返回的报文序列化成对象；
            3. 网络层：将报文与服务端通信；接受返回结果；
         2. 生成者：
            1. 代理层：一个应用提供服务，必须由一个网络监听的模块，处理网络上的监听；服务需要注册，只有注册的服务才可以被调用；注册的服务需要被反射调用到，来进行相应的处理；
            2. 序列化层：相应的做请求的反序列化和结果的序列化；
            3. 网络层：接受客户端报文；将序列化的结果返回给客户端；
         3. 服务提供方启动后主动向注册中心注册机器IP，port，提供的服务列表；
         4. 服务消费者启动时向注册中心获取服务提供方地址列表，可实现软负载均衡和Failover；
         5. http1.1默认使用长连接，RPC比http请求快的原因：HTTP使用HTTP协议，RPC使用TCP协议，比HTTP少了应用层，表示层，会话层，RPC使用长连接，比短连接更节省资源，效率更高；
         6. 长连接和短链接对比：
            1. 短连接：client向server发连接请求，server接受，双方建立连接，短连接一般只会在client/server间传递一次读写操作。优点是：管理起来简单，存在的连接都是有用的连接；
            2. 长连接：client/server完成一次读写后，之间的连接不会主动关闭，后续的读写操作会继续使用这个连接；
            3. 长连接优点：节约TCP握手时间，可以保证高实时性，数据流向可以采用服务器端的主动推模式
            4. 长连接缺点：并发量不宜过高，持续占用服务端口
      4. RPC基于Netty这一NIO通信框架：
         1. NIO：（New IO）
            1. 主要用块实现；
            2. IO和NIO的区别：

IO面向流，NIO面向缓冲区

IO每次从流中读一个或多个字节，直至读取所有字节，没有被缓存在任何地方，不能前后移动流中的数据。

NIO，数据读取到一个稍后处理的缓冲区，需要时可在缓冲区前后移动。增加了处理过程中的灵活性。

Java IO的各种流是阻塞的。当一个线程调用read或write，该线程被阻塞，知道一些数据被读取或数据完全写入。该线程在此期间不能做其他的事；

Java NIO非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用，就不会获取，而不是保持线程阻塞，所以直至数据变得可以读取之前，该线程可以继续做其他的事。非阻塞写类似，一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程可以同时去做别的事情。线程通常将非阻塞IO的空闲时间用于在其他通道上执行IO操作，因此一个单独线程可以管理多个输入输出通道；

NIO的选择器允许一个单独线程来监视多个输入通道；

* + - 1. Netty:
         1. 基于NIO开发的网络通信框架，对比BIO，并发性能提高；
         2. 特点：

高并发

传输快：传输依赖于零拷贝特性，减少不必要的内存拷贝；

封装好：封装了NIO操作的细节，提供了易于使用的调用接口；

* + - * 1. Netty使用的线程池：

NioEventLoopGroup线程池，池中的线程:NioEventLoop;

* 1. Http：网络传输协议，基于TCP，规定了传输的格式。缺点：消息封装臃肿；Rest风格，即通过Http实现；
     1. 具体了解Http：
        1. 超文本传输协议，定义了网络传输的请求格式，响应格式，资源定位，操作方式等；
        2. 与RPC的区别：
           1. RPC没有规定数据传输方式，格式可以任意指定，不同的RPC协议，数据格式不一定相同
           2. Http定义了资源定位的路径，RPC不需要
           3. RPC需要对调用过程在API层面封装，Http协议没有要求；
        3. 优点：RPC协议更加透明，对用户方便，Http方式更加灵活，没有规定语言，API；
        4. 缺点：RPC方式需要在API层面进行封装，限制了开发的语言环境；
  2. 如何选择：
     1. RPC速度更快，
     2. RPC实现难度较为复杂，Http较为简单
     3. Http更加灵活；
  3. 使用场景：
     1. 对效率要求高，并且开发过程使用统一的技术栈，使用RPC
     2. 对灵活度要求高，使用Http
  4. 在微服务中强调的是独立，自治，灵活，因此使用Http的Rest风格；

1. Spring cloud：
   1. 整合了很多组件：配置管理（config），服务发现，智能路由，负载均衡（Ribbon），熔断器（Hystix），控制总线，集群状态等；
   2. 主要组件Netflix：
      1. Eureka：注册中心
      2. Zuul：服务网关
      3. Ribbon：负载均衡
      4. Feign：服务调用
      5. Hystix：熔断器；