## 并发

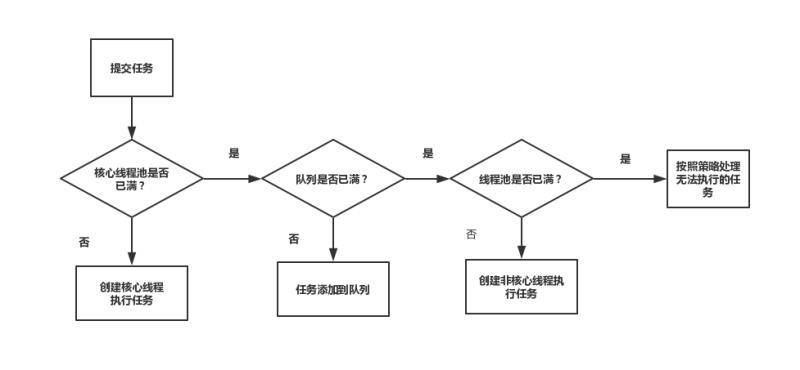
1. 进程与线程

进程是资源分配的基本单位（进程），线程是cpu调度的基本单位（cpu的一个核心每次只能运行一个线程），进程之间基本上是独立的，而线程共享堆和方法区（每个线程都有自己的程序计数器，虚拟机栈，本地方法栈），所以线程之间切换负担比进程小很多，一个进程至少拥有一个线程

1. Syn与ReentrantLock区别：1.lock获取锁时可以中断，可以设置获取锁的时间，可以用condition实现多路通知，lock获取锁后要手动释放锁 2. 两者都是可重入锁，syn为非公平锁，lock可以设置公平与非公平（非公平锁与公平锁区别：非公平锁首先会尝试获取锁，没有获取到才加入阻塞队列）3.syn是基于jvm层面实现的，lock是java代码层面基于AQS与CAS自旋机制实现的
2. Syn的底层原理：进入同步块会**执行monitorenter指令**来尝试获取此对象的**monitor对象（存在于每个java对象的对象头中）的持有权**，当monitor没有被任何对象持有时，为0，每获取一次，**monitor+1**。退出同步块时，会执行**monitorexit**（有正常退出和异常退出，都会释放锁）指令，monitor-1，直到monitor变为0。当修饰方法时，会增加一个标识符注明这是一个同步方法，每次执行同步方法尝试获取monitor。 Syn在实际中的使用：单例模式（双重校验锁）
3. <https://juejin.im/post/5c936018f265da60ec281bcb#heading-8> Monitor是基于mutex计算机原语实现的重量级锁（开销大），jvm对syn的优化：1.自旋锁：线程并不会马上进入阻塞状态，而是会自旋指定时间（因为如果每次获取不到锁都进入阻塞状态，执行上下文切换开销很大），在此时间内没有获取锁则阻塞 2.适应性自旋锁：自旋时间不固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。如果上次自旋获取成功，那么这次自旋次数会增多，上次获取失败，那么这次次数会减少甚至忽略 3.锁消除：代码上要获取锁，但实际检测到并没有线程安全问题，则会消除锁 4.锁粗化：在使用锁的时候，需要让同步块的作用范围尽可能小，这样做的目的是为了使需要同步的操作数量尽可能小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁 5.轻量级锁，偏向锁
4. 轻量级锁（**建立，拷贝，指针，自旋，膨胀**）：不再使用monitor，而是基于对象头。1.如果锁状态为无锁（01），虚拟机首先将在当前线程的栈帧中**建立一个名为锁记录**（Lock Record）的空间。2.将对象头的**mark word拷贝到线程的锁记录**中去。 3. 拷贝成功后，虚拟机将使用**CAS操作**尝试将对象的Mark Word更新为**指向Lock Record的指针**。如果这个更新成功了，则执行步骤4，否则执行步骤5 4.更新成功，这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位将转变为"00"，即表示此对象处于轻量级锁的状态 5.更新失败，虚拟机**检查对象的mark word**是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，可以直接进入同步块继续执行，否则说明这个锁对象已经被其其它线程抢占了。进行**自旋执行步骤3**，如果自旋结束仍然没有获得锁，轻量级锁就需要**膨胀为重量级锁**，锁标志位状态值变为"10"，Mark Word中储存就是指向monitor对象的指针，当前线程以及后面等待锁的线程也要进入阻塞状态
5. 释放锁：1.使用CAS操作将对象当前的Mark Word和线程中复制的Displaced Mark Word（即Lock Record中拷贝的Mark word）**替换回来** 2.如果成功，那么恢复无锁状态 3.如果失败，说明有其他线程已经尝试了获取该锁(此时锁已膨胀)，那就要在释放锁的同时，**唤醒**被挂起的线程。
6. 偏向锁（**检查（锁，id），CAS，轻量级锁**）：默认开启，无竞争情况下，连CAS自旋操作也省略了。1.检查mark word是否是可偏向锁状态 2.是则检查mark word存储的线程id是否是本线程，是则获取锁成功 3，不是，则通过CAS操作修改为本线程id 4.修改成功则获取锁成功，失败则拥有该锁的线程到达安全点之后，挂起这个线程，升级为轻量级锁。
7. 释放锁：1.偏向锁的释放采用了一种**只有竞争才会释放锁**的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。 2. 等待全局安全点(在这个是时间点上没有字节码正在执行)挂起 3. **检查持有偏向锁的线程是否活着**，如果不处于活动状态，则将对象头设置为无锁状态，否则设置为被锁定状态。如果为锁定状态，则锁升级为轻量级锁状态(00)
8. 锁状态：无锁，偏向锁，轻量级锁，重量级锁中mark word的变化 <https://blog.csdn.net/liudun_cool/article/details/86286872>



1. CAS操作：从地址V读取A，经过计算后得到B，再次从地址V获取值，判断是否为A，为A，则把B存入地址V（简单说就是：检验current与预期值是否相同，不相同则重新取值自增） ABA问题：加入版本或者时间戳 CAS与自旋一起使用非常消耗资源。
2. ThreadLocal作用：通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。**而使用ThreadLocal创建的变量只能被当前线程访问，其他线程则无法访问和修改，每个线程都有一个自己独立的一个变量**（注意一个ThreadLocal对象就只有一个value（如果要存储多个value，就要new多个threadLocal对象）） 原理：**每个线程都维护一个变量名为ThreadLocals的ThreadLocalMap对象，此map对象的key值为ThreadLocal对象**，所以每次取值或者插入都要先用当前线程获取map对象，再用当前的ThreadLocal作为key值取值或者插入。 ThreadLocal导致的内存泄漏问题：如果应用使用了线程池，那么之前的线程实例处理完之后出于**线程复用**的目的依然存活，所以，ThreadLocal设定的值被后面的线程持有，导致内存泄露。 解决：假如使用强引用，当ThreadLocal不再使用需要回收时，发现某个线程中ThreadLocalMap存在该ThreadLocal的强引用，无法回收，造成内存泄漏。entry中的key即ThreadLocal为**弱引用**，而用了弱引用后，GC就会及时回收生命周期已经结束的ThreadLocal，一定程度防止了内存泄漏。但是，回收Threanlocal对象后，value是却强引用，这会导致存在key为null，value有效的entry，所以，为了防止内存泄漏，**在ThreadLocal使用前后都调用remove清理**，同时对异常情况也要在finally中清理。 Remov方法作用：把map中当前ThreadLocal位置上的key和value都指向null，同时检测整个Entry[]表中对key为null的Entry一并擦除，重新调整索引。 使用：ThreadLocal一般加static修饰，同时使用前使用后都要remove。
3. ReentrantLock原理：公平锁：先判断锁是否被占有，如果是判断当前锁是否被自己占有，如果是，则state++（重入次数加1），再**判断CLH队列中是否有线程在等待**，如果有，则把**自己放入CLH队列挂起**（循环往前找，直到前驱节点不为CANCLED，如果前驱节点是0则把前驱节点设为SIGNAL，如果是SIGNAL，则自己可以安全挂起），如果没有则尝试获取锁。非公平锁：调用lock方法后直接用CAS操作**尝试获取锁**，如果失败，进入tryAcquire方法判断锁是否被占有，如果是判断当前锁是否被自己占有，如果是，则state++。如果锁没被占有，又**直接获取一次锁**，还是失败，则**加入阻塞队列**。当锁被释放之后，排在CLH队列队首的下一个线程会被唤醒。 释放锁：首先尝试**释放锁**（每释放一次state--），释放成功，那么查看**头结点的状态是否为SIGNAL(-1)**，如果是则**唤醒头结点的下个节点关联的线程**(也就是说，头节点是获取到了锁的线程关联的节点)，如果释放失败那么返回false表示解锁失败。 超时锁获取：非公平锁：直接获取锁，失败则入队，自旋**判断前节点是否为head节点**，如果是，尝试获取锁，获取失败则判断是否超时，超时则直接返回false，没有则**挂起当前线程直到超时时间过期**。这里为什么还需要自旋呢？因为当前线程节点的前驱状态可能不是SIGNAL（明明在addwrite中确保了所有前驱节点都是signal，为什么还要判断signal呢？因为他的前驱节点可能在队列中等待时被中断或超时了），那么在当前这一轮循环中线程不会被挂起，然后更新超时时间，开始新一轮的尝试（**线程挂起总是找到前驱节点为signal才能安全挂起**） Reentrantlock实现了Lock接口，内部**有两个类（公平与非公平**，根据用户选择new那个类），都继承自AQS抽象类，都重写了tryAcquire和tryRelease方法。
4. AQS是什么（抽象类**AbstractQueuedSynchronizer**）：AQS是抽象的队列同步器，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等。它维护了一个**volatile int state**（代表共享资源）和一个**FIFO线程等待队列**（多线程争用资源被阻塞时会进入此队列）。AQS定义两种资源共享方式：Exclusive（**独占**，只有一个线程能执行，如ReentrantLock）和Share（**共享**，多个线程可同时执行，如Semaphore，CountDownLatch）。不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时**只需要实现共享资源state的获取与释放方式即可**，至于具体**线程等待队列的维护**（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。（只需要实现tryAcquire，tryRelease，tryAcquireShared，tryReleaseShared，isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它，这几个方法即可）
5. AQS内部原理：1. 内部Node类，Node结点是对每一个**等待获取资源的线程的封装**，其包含了，next指针，需要同步的线程本身及其等待状态，变量waitStatus则表示当前Node结点的等待状态，共有5种取值**CANCELLED**（1，已被取消调度）、**SIGNAL**(-1，表示后继结点在等待当前结点唤醒，后继结点入队时，会将前继结点的状态更新为SIGNAL)、CONDITION（-2，当其他线程调用了Condition的signal()方法后，CONDITION状态的结点将从**等待队列转移到同步队列**中，等待获取同步锁）、PROPAGATE（-3，共享模式下，前继结点不仅会唤醒其后继结点，同时也可能会**唤醒后继的后继结点**。）、**0（**新节点入队时的默认状态）。 2. 方法：**acquire()**,此方法是独占模式下线程获取共享资源的顶层入口。如果**获取到资源，线程直接返回**，否则**进入等待队列**，**直到获取到资源为止**，且整个过程**忽略中断**的影响。 *addWaiter(),根据共享独占构造Node节点，将当前线程的节点加入到等待队列的队尾，并返回当前线程所在的结点 enq(),（如果队列为null，新建队列）将node加入队尾 acquireQueued（）,进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源 shouldParkAfterFailedAcquire(),此方法主要用于检查状态，看看自己是否真的可以去休息了（前驱节点为siganl即可休息，不是则判断是否为canceled，不是则把前驱节点设置为siganl，是则一直往前找，直到前驱节点不为canceled，把他插入此节点后面）* 3**.release**方法此方法是独占模式下线程释放共享资源的顶层入口。它会释放资源，**如果彻底释放了（即state=0）**,它会**唤醒**等待队列里的其他线程来获取资源 4. **acquireShared**：此方法是共享模式下线程获取共享资源的顶层入口。它会**获取指定量的资源**，获取成功则直接返回，**获取失败则进入等待队列**，直到获取到资源为止，整个过程忽略中断 5. **releaseShared**：此方法是共享模式下线程释放共享资源的顶层入口。它会**释放指定量的资源**，如果成功释放且**允许唤醒**等待线程，它会**唤醒**等待队列里的其他线程来获取资源
6. 三个中断方法：isInterrupted()：判断此线程是否被打上中断标记（即中断状态码），是返回true，否则false。interrupted()：静态方法，返回执行此语句的线程的中断状态码，并且当中断状态码为true时会被重置为false interrupt()：给这个线程打上中断的标记，但实际上并没有被中断，打上中断标记后调用Thread.isinterrepted()会返回true。
7. Volatile：保证了变量的可见性，禁止指令重排序。Volatile实现原理：修改后，立刻写入主存，并使得其他线程的缓存无效，由于无效，其他线程会重新去主存读取最新值。（缓存一致协议）禁止重排序原理：底层在读/写操作前后插入了内存屏障，避免了重排序。 Java内存模型：所有的变量**都是存在主存**当中，每个线程都有**自己的工作内存**。线程对变量的**所有操作**都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程**不能访问**其他线程的工作内存。
8. Final域重排序规则：1.在构造函数内对一个final域的写入，与随后把这个被构造对象（即拥有这个final值的对象）的引用赋值给一个引用变量，这两个操作之间不能重排序。（先写入final变量，后把此对象赋值给某个引用） 2.初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。 原理：final域写之后和读之前都会插入一个内存屏障。 例子：https://blog.csdn.net/riemann\_/article/details/96390511
9. 辅助类（都位于JUC包下面）：CountDownLatch：一个或者多个线程，**等待其他多个线程完成某件事情之后才能执行**。原理：初始时传值为N，则state也为N，每个线程countDown一次，则CAS操作state--，直到state为0，会unpark等待线程 CyclicBarrier：多个线程互相等待，直到**到达同一个同步点，再继续一起执行**，与 CyclicBarrier 不同的是，CountdownLatch 不能重复使用。 Semaphore：保证每一时刻，**最多有固定数目的线程在执行**。Acquire方法阻塞获取许可证，release方法释放许可证。原理：Semaphore与CountDownLatch一样，也是共享锁的一种实现。它默认构造AQS的state为permits，当所有的permits都被获取（为0），多余的线程就会被挂起，只有有线程释放许可证（permits++），其他线程才能运行（可以指定是否公平）。
10. 线程池（核心，队列，最大线程数）：corePoolSize核心线程数（核心线程会**一直存活，即使空闲**），maximumPoolSize最大线程数，keepAliveTime非核心线程空闲存活时间，unit存活时间单位，workQueue阻塞队列，threadFactory用于设置创建线程的工厂（在线程开始之前执行一些操作，如设置名字，设置异常处理handler等），handler线程池的饱和策略（1.默认抛出异常，2.直接丢弃，3.丢弃队列里最老的任务，将此任务继续交给线程池，4.交给线程池调用所在的线程处理）。



1. 异常： 不用线程池或者线程池使用execute()方法执行任务时，有异常会直接抛出。而用submit方法（submit方法返回Future对象，execute返回void）执行任务时，发生异常并不会抛出，这时有四种方法捕获异常：1. 手动在可能发生异常处try catch 2. 把执行结果给future，用try catch捕获future的get方法 3. 为线程设置异常处理handler 4. 继承ThreadPoolExecutor，重写after方法
2. 同步集合与并发集合：同步集合（HashTable,Vector，Stack，Collections.synchronizedMap(), Collections.synchronizedList()），并发集合（ConcurrentHashMap，ConcurrentSkipListMap等Concurrent开头的容器，CopyOnWriteArrayList，还有所有阻塞队列），同步集合与并发集合区别是，同步集合效率更低（就是简单地再方法前加了一个syn）。
3. CopyOnWriteArrayList：读取是完全不用加锁的，并且更厉害的是：写入也不会阻塞读取操作。只有**写入和写入**之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。原理：当 List 需要被**修改的时候**，我并不修改原有内容，而是对原有数据进行一次**复制**，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。
4. 阻塞队列：1.ArrayBlockingQueue：**数组实现的有界阻塞队列**，FIFO，一旦创建，容量不可改变，可以设置公平性（牺牲吞吐量），take方法取（为0阻塞），put方法放入（满了阻塞） 2. LinkedBlockingQueue：**链表实现有界阻塞队列**，FIFO，与ArrayBlockingQueue相比起来具有**更高的吞吐量**，容量可以选择进行设置，不设置的话，将是一个**无界**的阻塞队列，最大长度为Integer.MAX\_VALUE 3. PriorityBlockingQueue：支持**优先级**的**无界**阻塞队列（优先级高的元素先被拿出），默认情况下元素采用**自然顺序**进行排序，也可以通过自定义类实现**compareTo()**方法来指定元素排序规则，或者初始化时通过构造器参数**Comparator**来指定排序规则 4. SynchronousQueue：可以指定公平性，实际上**没有存储任何数据元素**，因为只有线程在删除数据时，其他线程才能插入数据，同样的，如果当前有线程在插入数据时，线程才能删除数据。 5. LinkedTransferQueue：由链表数据结构构成的无界阻塞队列，transfer方法，如果有消费者正在拿，则直接把数据交给消费者，如果没有，则插入队尾，直到有消费者消费刚才插入的元素才退出。 6. LinkedBlockingDeque：有界双端队列，上面的queue都是先进先出，这个可以先进后出 7. DelayQueue：存放实现Delayed接口的数据的**无界**阻塞队列，只有当数据对象的**延时时间达到时**才能插入到队列进行存储（所以设置时间早的总是在延迟一定时间后先执行），如果当前所有的数据都还没有达到创建时所指定的延时期，则队列没有队头，并且线程通过poll等方法获取数据元素则返回null（传入的元素都要实现Delayed接口）
5. 为什么要使用线程池？1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。2.可以根据系统的承受能力，调整线程池中工作线线程的数目防止因为消耗过多的内存，而把服务器累趴下 3.提高响应速度， 如果任务到达了，相对于从线程池拿线程，重新去创建一条线程执行，速度肯定慢很多。
6. 常用线程池：1.newCachedThreadPool：线程池无界（MAXVALUE），可以复用。内部实现：core=0，max=MAXVALUE，time=60s，queue= SynchronousQueue（所以队列大小为0，所有线程的执行都是**非核心线程**，空闲存活时间为60s，所以任务一传入就会运行） 2. newFixedThreadPool：定长，线程最大并发数就是设置的大小，超出的线程会在队列中等待。内部实现：core=n，max=n，time=0，queue= LinkedBlockingQueue（永远**不会用到非核心线程**（所以存活时间无意义），因为queue的size为MAXVALUE）。 3. newScheduledThreadPool：创建一个定长线程池，支持**定时及周期性**任务执行。内部实现：core=n，max=MAXVALUE，time=0，queue= DelayedWorkQueue（因为queue无界，所以**永远不会用到非核**心线程） 4. newSingleThreadExecutor：单线程化的线程池。内部原理：core=1，max=1，time=0，queue= LinkedBlockingQueue（因为无界，所以**非核心线程不会存在**，而核心线程是固定的1，所以永远只有1个线程在执行任务，与上面的newFixed区别是把n变为了1）
7. 线程间交换信息：共享对象或者阻塞队列。线程间通信：轮询volatile变量，使用wait/notify，使用await/signal
8. 线程常用方法：1**.join(1000)**把运行此语句的线程（main）加入调用此方法的线程后面运行（main加入阻塞队列），如果1000毫秒（如果没有设置时间，则调用线程执行完毕，main线程才能执行，即使调用线程进入阻塞状态或者线程让步，main线程也不能先执行））内thread线程没有执行完，那么main线程进入就绪状态，通俗说就是调用的先执行。 2**.yield**从执行状态变为就绪状态（就绪后并不是说优先级高的就运行，只是大概率优先级高的运行，所以此方法最主要的作用是当前的线程暂时让步给别的线程）重新竞争cpu资源，通俗地说就是立刻暂停 3.**sleep**线程阻塞指定时间，此时间结束后进入就绪队列 4.**wait**线程进入阻塞队列（wait与notify都要写到syn语句块里面，要不然会报错。Wait一般与while一起使用，而不是if，因为用wihle被唤醒后，还会执行一次条件判断，如果条件不满足还会阻塞，而if被唤醒后会立刻执行，所以wait一般放在while中，真正的业务代码放在while循环下面，while循环的条件是不能执行业务代码时为真，如size==0时要wait，不为0才能取） 5.**notify**线程从阻塞队列变为就绪队列（如果有两个线程正在wait，而调用一个notify则会随机选取一个），注意，虽然wait线程进入了就绪队列，但要等到notify线程退出syn块才能获得锁并继续运行。
9. 线程状态转换：new（新建，还没有执行start），Runnable（就绪 yield，start后没有获得cpu资源），Running（运行），Blocked（阻塞 sleep，wait，**join**），Dead（死亡） 就绪状态：线程start后，并不是立刻运行，因为cpu资源就这么多，所以暂时处于就绪状态，等待调度。 阻塞状态：让出cpu资源，让其他线程运行
10. Callable和future：Callable接口（call方法，有返回值与异常）与Runable（run方法）不同的是，前者线程执行完毕后可以**返回数据或者抛出异常**，且Callable要**与ExecutorService，Future一起使用**。线程池使用submit执行了实现了callable接口的类后，会返回Future类型的对象，此Future对象调用isDone方法判断是否执行完毕。调用**get方法获取返回结果**，如果线程没有结束，get方法会阻塞直到执行完毕。FutureTask的使用：把实现Callable的类传入FutureTask对象，再把FutureTask对象传入submit方法，可以直接用此FutureTask对象get返回值。
11. 阻塞队列的实现原理：使用的是reentrantlock的多路通知机制（condition），类中拥有两个Condition类：notEmpty与notFull（notEmpty与notFull调用阻塞或通知方法都要在lock块中），当要**取元素**而count为0时（为空），**notEmpty**调用await方法（取走一个元素后，**notFull**调用signal方法通知放入元素的线程），当要**放入**元素而count为length时（为满），**notFull**调用await方法（放入一个元素后，**notEmpty**调用signal方法通知取元素线程），所以消费者只会通知生产者，生产者只会通知消费者。
12. 死锁产生的条件：1.资源互斥访问 2.一个线程因为请求资源而阻塞时，对已获得的资源保持不放。 3. 进程已获得的资源，在末使用完之前，不能强行剥夺。 4. 若干进程之间形成一种头尾相接的循环等待资源关系。 阻止循环等待。
13. 预防死锁方法：1.资源共享 2. 进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦投入运行后，这些资源就一直归它所有 3. 当一个已保持了某些不可剥夺资源的进程，请求新的资源而得不到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请 4. （有用）为了破坏循环等待条件，可釆用顺序资源分配法。首先给系统中的资源编号，规定每个进程，必须按编号递增的顺序请求资源，同类资源一次申请完。（银行家算法） 5.（有用）超时放弃算法，即lock里面的tryLock方法
14. 检测死锁：用JStack命令（java虚拟机自带的一种堆栈跟踪工具，Jstack工具可以用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因）和Jconsole工具（jdk/bin目录下的工具，对运行在Java应用程序的资源消耗和性能进行监控，并画出大量的图表）
15. 线程安全：在多个线程向共享资源做了写操作时才有可能发生线程不安全，只要资源没有发生变化,多个线程读取相同的资源就是安全的。多线程对一些资源的竞争的时候就会产生竞态条件。导致竞态条件发生的代码区称作临界区。保证线程安全：用syn/lock。
16. 用syn()锁住的是**对象**，如果syn()里的是不同的对象，虽然是同一段代码，但也起不到作用。所以当用syn锁住**string**，希望用同一个string的线程能够串行执行时，string应该用intern()生成（并且string是不可变类型，一旦改变就是不同对象了）。当用不同的对象想锁住代码块时，应该用**类.Class或者修饰静态方法**。用syn(this)并不能解决线程安全问题，因为锁住的是正在运行的线程，而不是外部的共享资源（而匿名内部类（Runable）里面的成员变量是线程私有的，并不需要保证线程安全）。Syn**修饰方法**锁住的就是线程对象本身。
17. Java原子类原理：cas操作保证原子性。lazeSet方法：设置值，以一种延迟的方式；设置值后，不保证该值的修改能被其他线程立刻看到，也就是说，其他线程有可能读到的还是旧的值；set方法：修改后，别的线程可以立刻看到更新的值。好处：用了lazeset后，能提高系统性能，因为在多个cpu缓存之间同步一个内存值的代价是很昂贵的。
18. Java线程池怎么保证核心线程一直运行：调用execute提交一个任务后，会根据传进来的Runnable对象创建一个Worker对象，然后调用此Worker对象的run方法，run方法委托给runWorker执行，runWorker中有while循环，不断调用getTask方法获取新任务，**getTask方法通过阻塞队列的take方法获取任务（take方法就是一直获取，获取不到就阻塞，直到最后获取到）**
19. 单线程池的应用场景：不适合并发但是却可能引起IO阻塞及影响UI线程响应的操作，如数据库操作、文件操作等。
20. Juc包：原子类，reentrantlock等锁，并发集合，线程池，辅助类。

## 容器

Arraylist

1. ArrayList与LinkedList的区别：1.底层实现，2.插入，删除，查找操作，3.线程安全

4.内存空间占用

1. ArrayList：有序，允许null和重复，不是线程安全的（可以使用Collections包装）
2. Arraylist大小：没有指定大小时为10（先声明为size为0的数组，再在第一个add时初始化大小为10），小于0抛出异常，大于或等于0就设置为这个，输入容器时先把容器转换为数组，再判断大小，系统规定arraylist数组容量最大值为：MAX\_ARRAY\_SIZE = 2147483639
3. Arraylist扩容机制：1.判断是否是length为0的默认初始化的数组（即没有指定大小的数组），如果是初始化大小为10，2.判断容量是否够用，如果不够每次都扩容为原来（length）的1.5倍，3.判断1.5倍是否满足需求（size+1），不满足直接扩容为需求值 1.5倍是：oldCapacity + (oldCapacity >> 1) 3->3+1=4
4. Arraylist如何扩容的：生成一个新的length的数组，把原始数据复制到新数组
5. Remove操作：范围检查，remove操作后面的每一个元素集体向前移动一位覆盖remove位置上的元素，把size-1位置上的元素变为null add操作：判断是否需要扩容，将元素加到最后一位 get操作：判断index是否符合要求，直接取index位置上的元素 set操作： 与get相同 contains操作：循环判断是否存在
6. Transient：因为实际的length与size不相等，所以序列化时就不用序列号哪些length-size后面的元素，所以先序列化非transient的东西，序列化elementData时，只序列化size大小的元素（加快了序列化速度，减少空间浪费）（被transient关键字修饰的变量不再能被序列化。）
7. 实现RandomAccess接口并没有方法，只是标识这个类有随机访问的功能（数组）
8. List，set，map之间的区别：list元素有序，可以重复。Set元素无序，不能重复。Map元素无序，key值不能重复。
9. Collection与Collections的区别：Collection接口是java集合的基本接口，他提供了集合对象进行操作的通用接口方法（如获取迭代器，add方法等），list/set都是它的子类（Map接口是map的顶层接口）。Collections是一个包装类，包含了很多静态方法，不能被实例化，就像一个工具类，提供排序方法，构造同步容器的方法等。
10. ArrayList与Vector的区别：后者是线程安全的，使用了Syn包裹，而ArrayList非线程安全，ArrayList性能更好（因为没有锁），Vector扩容每次增加1倍，而ArrayList每次增加0.5.
11. Arraylist没有缩容，数组的长度永远不会减少。Remove方法和clear方法只是会把相关位置的值置为null，以便垃圾回收

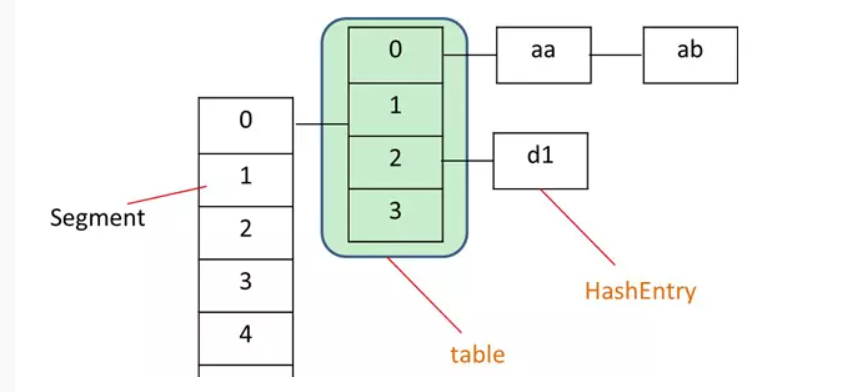
HashMap

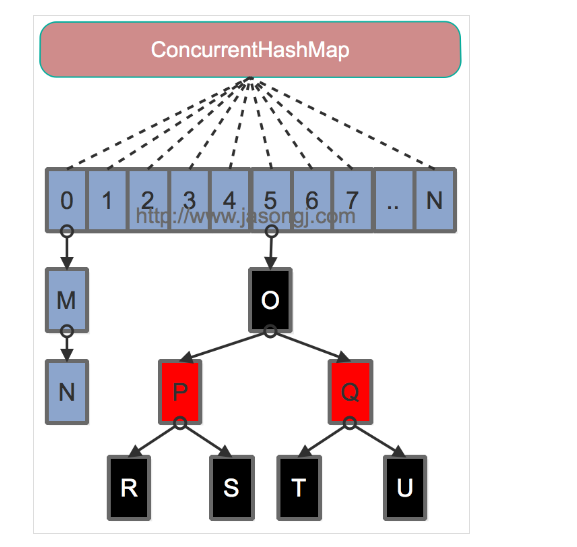
https://www.zhihu.com/collection/518095252

1. 初始化：Node[] table默认长度为16，负载因子默认为0.75，threshold=table.length\*factor，threshold为所能容纳键值对的个数，超过threshold要resize扩容 初始化传入值时，实际的length为大于或等于传入值的2的幂次方的数，如new HashMap(10)，则实际length为16，如果传入16，则length为16
2. Hashmap：无序，允许一条key为null，多条value为null，key值不能重复，访问速度块，非线程安全（Collections或者Concurrenthashmap）,**key值是不可变类型**（即hashcode()得出的值不能改变，因为如果改变，取值就就找不到这个值了）
3. HashSet：HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。（HashSet的add操作就是map.put(key,PRESENT)，PRESENT是一个static Object类） TreeSet：有序的set集合，内部有一个TreeMap，同样是调用TreeMap的方法来实现相关方法，如add（e）方法就是map.put(e, PRESENT)。 （**所以set都是基于map实现的**）
4. Hashtable线程安全，并发性太低(因为都是只用syn包裹) linkedhashmap：保存了记录插入的顺序 **TreeMap**：可以根据键值排序，key必须实现comparable接口或者向treemap中传入comparator（treemap内部是完全由红黑树实现的，所以他的内部是排好序的，它的查找时间复杂度不是1，而是logn，插入时，判断key相同会覆盖，以此达到去重的效果）
5. Put操作：1判断table是否为null，是则**扩容**，key的hash值传播高位到低位运算（异或，右移）和取模运算（&运算）确定数组**位置** 2.判断此位置是否有元素，没有则直接插入，有则比较key值是否相等，相等则**更新value值** 3.不相等则判断此位置是否为红黑树，是则插入红黑树中，不是则插入链表中，插入链表后判断链表长度是否大于或等于8，是则**转化为红黑树** 4.判断size是否为threshold，是则**扩容**
6. 扩容机制（resize）：会伴随着一次重新hash分配，并且会遍历hash表中所有的元素，是非常耗时的。在编写程序中，要尽量避免resize。步骤：1.如果table为null，则**初始化table**，返回即可，2.判断length是否大于等于最大值，是则不再扩充直接返回，没有则扩充为**原来的2倍**（length和threshold都乘2） 3. 遍历oldtable，如果某个**节点没有next**，则直接取模运算得出新的index并赋值给newtable，4.节点为**红黑树**，则用红黑树的算法，5.如果节点为**链表**：判断e.hash&oldCap是否为0（e.hash&(oldCap-1)是取模运算，别搞混了），是0说明位置不变，不是0说明位置变为原位置+oldlength（为什么？实际上length\*2就是length的二进制的1左移了一位，即length-1的二进制多出了一个1，这个1可能起作用（原位置+length），也可能不起作用（原位置），比如1010&111和1010&1111是不同的，而0010&111与0010&1111结果相同，所以我们只需判断第四位是否为1即可，即1010&1000为1000，说明1010位置会变，而0010&1000为0，说明0010位置不变），运用这种算法比重新计算index更高效（而重新计算index也是这两种情况，要么原位置，要么原位置+oldlength）
7. Get方法：1.传播高位到低位运算与取模运算计算数组位置 2.如果table为null或者此位置元素为null返回null 3.判断数组中此位置key值是否等于传入的key值，是则返回value 4.不是则查找红黑树或者链表，找到则返回，没有返回null
8. 为什么长度是2的幂次方？如果length为2的幂次方，那么，hash%length==hash&(length-1)，采用&运算比%运算更快 如10=1010 15=1111 ，10&15=10即10%16=10 2的幂次方的32位二进制只有1个1，其他都是0，如2：10，4：100，8：1000，16：10000，32：100000，所以二进制减1后都是1，如8-1=7=111，而1与任何数&操作都是那个数
9. 插入链表由头插法变为尾插法，避免了并发场景下链表成环的问题：两个线程put操作（原链表：1->2->3），一个resize了形成了一个新的table（而因为头插法，新的链表成了(1->3->2），一个没有导致一个线程中2.next指向3，而3.next指向2，成了一个环，在这个线程resize的时候就会死循环了。但是hashmap没有解决数据丢失的问题：即在插入时，两个线程都判断某个位置为null可以插入，第一个线程插入后，第二个线程也插入了，那么第一个线程插入的数据就丢失了
10. 负载因子：如果内存空间很多要提高效率，可以降低factor的值，如果内存紧张，而效率要求不高可以提高factor的值（因为，length相同，factor越大，存储的数据越多，碰撞的次数越多），为什么要选择0.75？是时间和空间的平衡，根据泊松分布，在负载因子默认为0.75的时候，单个hash槽内元素个数为8的概率小于百万分之一，所以将7作为一个分水岭，等于7的时候不转换，put操作时大于等于8的时候才进行转换，remove操作时小于等于6的时候红黑树就化为链表。
11. 为什么要扩容：减少hash冲突的概率
12. 为什么要转化为红黑树：红黑树(logn)比链表(n)查找更快
13. 为什么hash不用hashcode，而要多加一些运算呢？首先，我们知道index为（length-1）&hash，而length-1显然为一个前面都是0，后面都是1的32位值，这样的话，如果只用hashcode与length-1与运算，那么hashcode的前几位完全没有参与运算，如果hashcode是一个后几位都是0的值，那么很容易发生碰撞。而先给hashcode高位运算，可以将原先不参与运算的高位也给包含进来了（当只有高位有1时，高位运算把高位推向了低位，当只有低位有1时，高位运算对他没有改变）
14. Hashmap1.7与1.8的区别：1.8增加了红黑树，而1.7只有链表。插入链表时，由头插法变为了尾插法。
15. 遍历map：entrySet()获得所有键值对，keySet()获得所有key，values()获得所有value。

ConcurrentHashMap

1. Hashtable效率低的原因：他对数据操作的时候都会用syn对方法上锁，不论是set还是get，所以效率比较低下，Collections.synchronizedMap同样
2. Hashtable的不允许key或者value为null，直接会抛出空指针异常，为什么呢？如果你使用null值，就会使得其无法判断对应的key是不存在还是为空，因为你无法再调用一次contains(key）来对key是否存在进行判断（原来key=‘1‘是不存在的。这时，两个线程，一个线程调用get(‘1’)发现返回是null，于是调用contains(’1‘)判断是不存在还是value为null，而另一个线程此时插入了key=’1’的键值对，那么显然第一个线程返回的信息是错误的），ConcurrentHashMap同理。
3. 安全失败机制：concurrent包下面的容器都是安全失败，可以在多线程下并发遍历，并发修改。原理：在遍历（注意只有在使用迭代器遍历时才会，使用get不会，因为volatile可见性，直接使用get不会有安全问题）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，所以遍历时集合的改变检测不到
4. 快速失败：在用迭代器遍历容器时，如果容器内容发生了改变，会抛出异常，原理：hashmap中有一个modCount变量，集合每次改变，modCount++，每次迭代器迭代时，都会检测modCound的值，与期望值不相同，则抛出异常
5. Hashtable与hashmap的不同：1.初始容量：11，16 2.扩容机制：当前容量翻倍+1，容量翻倍 3.迭代器不同，不是快速失败，是快速失败。
6. Concurrenthashmap1.7（一个ConcurrentHashMap里包含一个Segment数组，每个Segment里包含一个HashEntry数组，我们称之为table，每个HashEntry是一个链表结构的元素。 Segment[] segments, Segment类继承reentrantlock，Segment类属性：HashEntry[] table，hashentry类属性包含key，value，next）： put方法：1.根据key的hash值定位到某个segment 2.自旋**获取当前segment的锁**  3. 计算在HashEntry数组中的位置，判断此此位置是否为null，不为null则遍历则位置上的链表，更新或者插入链表 4. 为空则需要新建一个 HashEntry 并加入到 hashentry数组中 中，同时会先判断是否需要扩容 5.释放锁 所以他是对每个segment加锁，则理论上数组有多长，此concurrenthashmap的并发度就有多高（默认16） get方法：不加锁，非常高效





1. 1.8（Node[] table, node继承hashMap的Entry）：内部结构与hashmap相同，初始数组大小也为16，链表大于8时转化为红黑树 ，扩容也是2倍 1.7采用分段锁，1.8之后用cas思想代替分段锁，核心就是尽量降低同步锁的粒度（主要区别：红黑树，分段锁（syn+CAS，lock），尾插法）
2. Put方法（初始化，定位，自旋，扩容，写入，转化，大小，扩容）：1.判断table是否初始化 2.定位到key在table中的位置，判断此位置是否为null，为null则用CAS自旋写入数据 3.如果不为null判断是否正在被别的线程扩容（即table定位到的节点的hash值是否为-1），是则帮助扩容 4.不为null也没在扩容则用syn加锁（为node加锁，相当于segment）写入数据（链表(hash>=0)或红黑树（hash<0，treebin的值为-2，treebin代表的是红黑树的根节点的hash值）），退出syn 5.判断链表node数是否达到8，达到则转化为红黑树 6.容器大小加一，判断是否需要扩容，如果需要就扩容，如果正在扩容，就帮助扩容 同样，get方法也没有加锁
3. Put方法的第一步就是给key值再hash，即执行spread（）方法，算出key的hash值，以后使用的都是这个hash值，而不是hashcode()计算出的值（hashmap中的hash值也是再hash的值）
4. 初始化table：table为null时，自旋判断**sizeCtl是否小于0**（小于0说明此table正在**被其他线程初始化**或者扩容），是则**让出cpu**（即线程让步），不是则用cas操作给sizeCtl赋值为-1，并再次判断table是否为null，是则**初始化table**，最后重新给sizeCtl赋值（赋正值，sizeCtl表示：-1，表示正在初始化，-n，表示正在由n-1个线程执行扩容操作，正值为此容器的容量即length\*0.75或者length-length>>2）
5. Size()方法：baseCount加上counterCells数组的所有非null值得出
6. helpTransfer()帮助扩容：1.如果table不是null，put方法传进来的node节点是转移类型，node的新table不是null，则尝试帮助扩容 2.帮助扩容：根据length得到标识符rs，循环检测：如果oldtable没变，newtable没变（相比1得到的没变），且sizeCtl小于0 3.再判断，如果扩容没有结束，没有达到最大线程数，则帮助扩容（执行transfer方法），否则退出循环退出方法

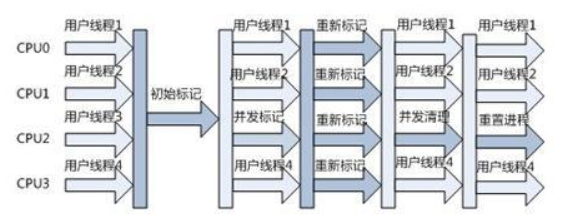
<https://segmentfault.com/a/1190000014982330> map各个方法分析

1. addCount()数组容量加1且判断是否扩容：1. 判断计数盒子属性是否是空，如果是空，就尝试修改 **baseCount 变量，对该变量进行加 X**（默认为1），不为null直接第3步 2. 如果计数盒子不是空，或者修改 baseCount 变量失败了，则放弃对 baseCount 进行操作。 3.如果计数盒子不是null，且长度不为0，且计数盒子上**随机取的一个元素不为null**，那么，就执行cas操作把这个元素的**value值加上x**，上面任意一条不满足（即countercells为没有被初始化，或者没有插入一条数据，或者随机选取的位置上没有插入过数据，或者执行cas操作失败），就执行fulladdcound方法 4.fulladdcound：简而言之就是初始化countercells，或者初始化某个位置上的cell，或者更新此cell的value值，或者扩容countercells 5.判断是否需要扩容，如果正在扩容，则帮助扩容，如果需要扩容，就开始扩容 （**简而言之**：通过修改 baseCount，或者通过使用 CounterCell。当 CounterCell 数组被初始化了，就**优先使用他**，不再使用 baseCount。CounterCell是一个类，里面有一个volatile long类型的value）

## Jvm

类加载过程

1. 类加载过程：加载-连接（验证，准备，解析）-初始化
2. 加载：3步（1.获取二进制字节流 2.将字节流中静态存储结构转化为方法区的的运行时数据结构 3.生成一个class对象，作为方法去访问这些数据的入口），重点：1.字节码来源：编译生成的.class文件，或者jar包中的.class文件 2.类加载器：启动类加载器，扩展类加载器，应用类加载器，自定义类加载器，为什么要自定义加载器呢:1，可以对编译后的代码进行加密，然后通过自定义的类加载器进行解密 2.可以从非标准来源加载代码，如网络，就需要自己指定加载来源 （Class.forName()执行类加载整个过程，Classloader.loadClass()只执行加载这一步）
3. 验证：主要是为了保证加载进来的字节流符合虚拟机规范，不会造成安全错误：文件格式校验，元数据校验，字节码校验，符号引用校验
4. 准备：主要是为类变量（注意，不是实例变量是static）分配内存，并且赋予初值（不是我们赋予的值，而是0或null，常量就是我们赋予的）
5. 解析：将常量池内的符号引用（如代码中的方法hello()这个字符串）替换为直接引用(hello方法存储的位置或者偏移量)的过程
6. 初始化：对static变量或语句进行初始化（即给static变量赋值，运行static初始化块）真正执行代码的过程，如果父类未初始化，则先初始化父类 顺序：按代码顺序执行：父类变量或初始化块->子类变量或初始化块，时机：new，调用静态变量或方法，反射，程序入口类（包含main）
7. 什么是STOP THE WORLD（STW）？等待所有用户线程进入安全点（safe point）后并阻塞，做一些全局性操作的行为。引发原因：GC，RevokeBias（撤销偏向锁）等。GC：即在执行垃圾收集算法时,Java应用程序的其他所有除了垃圾收集收集器线程之外的线程都被挂起，这些工作都是由虚拟机在后台自动发起和自动完成的，是在用户不可见的情况下把用户正常工作的线程全部停下来，这对于实时性要求很高的程序来说是难以接受的，例如采用复制算法时，为了保证在复制存活的对象的时候，对象的一致性
8. Java对象组成：对象头（缓存对象自身的运行时数据即**mark word：**hash码，年龄分代，锁状态等 类型指针：判断是那个类的实例 数组长度：只有数组对象才有），实例数据，对齐填充（HotSpot虚拟机的对齐方式为8字节对齐，即一个对象必须为8字节的整数倍，如果不是，则通过这个对齐填充来占位填充） 参考多线程第四条的连接，对象头详解参考多线程第九条连接
9. 类加载器：1.BootstrapClassLoader(启动类加载器) ：最顶层的加载类，由C++实现，负责加载jdk的lib目录下的类 2. ExtensionClassLoader(扩展类加载器) ：主要负责加载jre的lib目录下的ext目录下的类 3. AppClassLoader(应用程序类加载器) :面向我们用户的加载器，负责加载当前应用classpath下的所有jar包和类。
10. 双亲委派模型：即在类加载的时候，系统会首先判断当前类是**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（父类也会委派给他的父类），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，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（它的搜索范围中没找到所需的类），才由自己来处理。
11. 双亲委派模型的好处：假如黑客定义了一个java.lang.String类，该String类具有系统的String类一样的功能，只是在某个函数稍作修改。比如equals函数，这个函数经常使用，如果在这这个函数中，黑客加入一些“病毒代码”。并且通过自定义类加载器加入到JVM中。此时，如果没有双亲委派模型，那么JVM就可能误以为黑客自定义的java.lang.String类是系统的String类，导致“病毒代码”被执行。而有了双亲委派模型，黑客自定义的java.lang.String类永远都不会被加载进内存。因为首先是最顶端的类加载器加载系统的java.lang.String类，最终自定义的类加载器无法加载java.lang.String类。**某个类是由此类和他的加载器一起决定的**，就算用了自定义加载器加载String类，但因为加载器不是启动类加载器，所以和java.lang.String也是不同的。
12. 不用双亲委派模型：自定义一个类加载器，然后重写loadClass()方法
13. 垃圾收集器：1.serial收集器：单线程收集垃圾，其他所有线程停止（STOP THE WORLD），直到收集完毕（新生代采用复制算法，老年代标记整理）（简单高效，适用于client端） 2.parnew：多线程收集垃圾，所有用户线程停止（复制，标记整理）（适用于server端） 3. Parallel Scavenge：多线程收集垃圾，所有用户线程停止（复制，标记整理）（与parnew区别：关注吞吐量，提供了很多参数供用户找到最合适的停顿时间或最大吞吐量） 3.CMS：GC线程与用户线程同时执行（标记清除算法，以最短停顿时间为目标，有两次stw）
14. CMS收集器：1.初始标记：**暂停**所有的其他线程，并记录下**直接与 GC root 相连**的对象，速度很快 2. 并发标记：同时开启 GC 和用户线程，用一个闭包结构去**记录可达对象**（根据第一步得到的对象向下追溯标记）。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会**不断的更新引用域**，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会**跟踪**记录这些发生**引用更新**的地方。 3.重新标记：**暂停用户线程**，重新标记阶段就是为了**修正**并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短 4.并发清除：开启用户线程，同时 GC 线程开始对**未标记的区域做清扫**。 优点：并发收集，停顿时间短 缺点：对cpu资源敏感，无法处理浮动垃圾，使用标记清除算法，会产生大量空间碎片

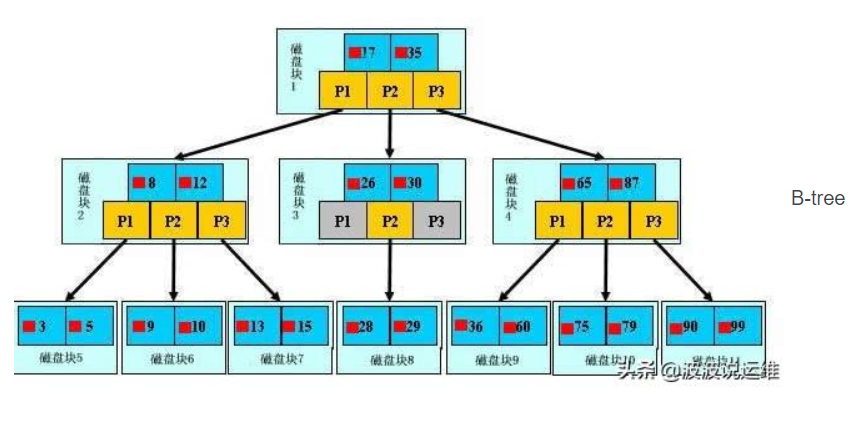


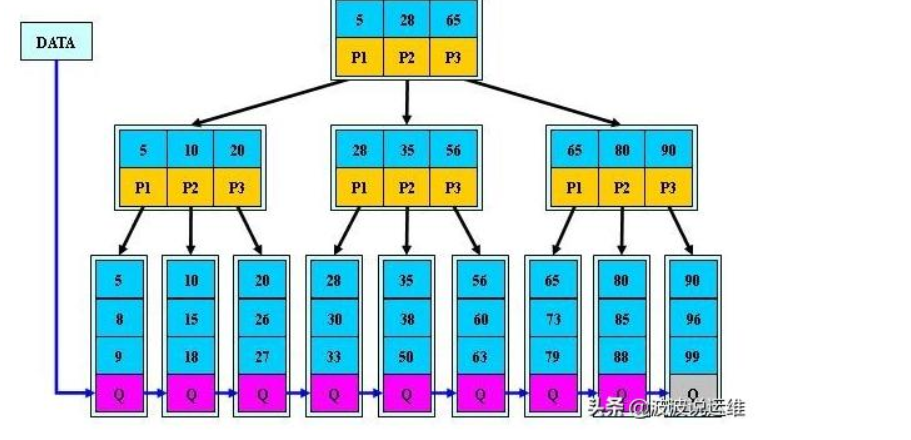
1. G1（复制算法）：面向大内存，多核系统的收集器，具有**高吞吐量和可预测的停顿时间**。G1收集器将内存划分为一系列大小相等的**region区域**（大小为1-30M，启动时确定），每个region区域逻辑上属于一个分代区域，并且在物理上不连续，当一个属于老年代的区域收集完成后，**变成新可用Region**并可能成为下一个新生代。当申请的对象大于region的一半时，会被放入一个巨型区域中。1.初始标记 2.并发标记 3.最终标记 4.筛选回收 G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region
2. 垃圾收集算法：1.标记清除：**标记需要回收对象**，标记完成后全部回收所有标记对象。问题：效率问题，产生不连续空间碎片 2.复制算法：把内存分为两块，每次只是用其中一块，如果这一块用完，就将还存活的对象复制到另一块，然后把这一块完全清理掉，重复这一过程（因为对象地址改变了，所以需要更新对象的引用） 3.标记整理算法：标记需要回收对象，标记完成后让所有存活对象向一端移动，然后清理掉边界之外的内存。 4.分代收集算法：根据对象存活周期的不同将内存分为几块，一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。（比如在新生代中，每次收集都会有**大量对象死去**，所以可以选择复制算法，只需要付出**少量对象的复制成本**就可以完成每次垃圾收集。而**老年代的对象存活几率是比较高**的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。）
3. Java内存区域：**线程私有（生命周期都与线程相同）**：1.程序计数器：字节码解释器通过改变程序计数器来依次**读取指令**，从而实现代码的**流程控制**，如顺序执行，循环，选择等。在多线程的情况下程序计数器用于**记录当前线程执行的位置**，从而当线程切换回来时，能够直到线程上次运行到哪里了。 2.java虚拟机栈：每一个**方法就是一个栈帧**，函数调用，栈帧入栈，return或者异常，出栈。栈帧：**局部变量表**（局部定义的8中**基本数据类型**，方法参数，对象引用）、操作数栈、动态链接、方法出口信息。两种异常，一种虚拟机栈的内存大小不能扩展超出内存（StackOverFlowError），一种可以扩展还是超出（OutOfMemoryError）。 3.本地方法栈：与虚拟机栈几乎相同，唯一的区别是虚拟机栈为字节码服务，本地方法栈为native方法服务，在hotspot中两种栈合二为一。 **线程共享**：1.堆：存放**对象实例**，几乎所有的对象实例和数组都分配在这里。**堆分为新生代（Eden，From survivor，To survivor）和老年代**，1.8后永久代被移除，取而代之的是元空间，元空间使用直接内存。堆在虚拟机启动时创建。最容易出现outofmemoryerror错误。 2.方法区：用于存储已被虚拟机加载的类信息，**常量，静态变量**，即时编译后的**代码**等信息（注意没有实例变量，实例变量随对象存储在堆中）。**运行时常量池**：存放编译期生成的各种字面量（文本字符串，8种基本类型**的值**（如int i=8,则8就是字面量），final常量）和符号引用（类，方法，成员变量的名词和描述符）。 字符串常量池：jdk7后的字符串常量池放到堆中了，实现字符串常量池的是一个StringTable，他是一个Hash表，被所有类所共享，默认值为1009，StringTable中放的是字符串常量或者堆内字符串对象的引用
4. 对象的创建：1.**类加载检查**，检查这个类有没有被加载。 2.分配内存，对象所需的**内存大小**在类加载完成后便可确定，**分配方式**有指针碰撞（jvm采用标记整理算法，用过与没用过的内存中间有一个分界指针，只需要把指针向前移动即可分配）与空闲列表（jvm采用标记清除算法，虚拟机会维护一个列表，该表会记录那些内存块是可用的，分配时找一份足够大的分配给对象）两种，用**CAS加自旋**或者**TLAB**（为每一个**线程**在TLAB区域分配一块内存，创建对象时，首先在TLAB区域分配内存，TLAB用完时再使用CAS加自旋分配）来保证线程安全。 3.**初始化零值**，把所有字段（即成员变量）初始化为0/null值 4.**设置对象头**（参考jvm第8条），如设置类型指针，GC年龄分带，hash码等。 5.执行**初始化方法**，执行构造器。
5. Java对象访问：1.句柄：Java 堆中将会划分出一块内存来作为**句柄池**，reference 中存储的就是对象的句柄地址，而句柄中包含了**对象实例数据与类型数据**各自的具体**地址**信息 2.直接指针：reference 中存储的直接就是对象的地址。 优劣：使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在**对象被移动时**只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是**速度快**，它节省了一次指针定位的时间开销。
6. 对象分配回收过程：大部分情况下，对象首先在Eden区分配(Eden->(经过一次垃圾回收还存活，年龄变为1)->survivor1或2->(年龄增长到一定程度，parallel默认为15，可以设置)->老年代)。当Eden区没有足够的空间时，虚拟机将发起一次Minor GC，速度比较块，比较频繁。Major GC/Full GC发生在老年代的GC，速度比较慢，通常伴随Minor GC。大对象如数组等直接存储到老年代。每回收一次垃圾，对象的年龄增加1。
7. 判断对象已经死亡：1.引用计数法：每给对象增加一个引用，计数器就+1，引用失效，计数器-1，计数器为0时表示可以回收。问题：交叉引用。 2.可达性分析算法：通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象不与任何GC Roots相连，则证明此对象是不可用的。
8. 四种引用：强引用，永远不会回收 软引用：内存空间不足才回收 弱引用：GC线程一旦发现只具有弱引用的对象，就会回收他 虚引用：一个对象持有虚引用，就像没有引用一样，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。
9. 不可达对象并非非死不可，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收
10. 判断一个类是无用的类：无任何实例，加载该类的加载器已经被回收，该类的class对象没有被任何地方引用，无法在任何地方通过反射访问该类。
11. GC调优常用命令：1.指定堆内存大小：-Xms2G -Xmx5G表示堆内存最小为2G，最大为5G 2.指定新生代内存大小：-XX:NewSize=1G -XX:MaxNewSize=2G表示新生代内存最小为1G，最大为2G（因为Full GC成本远高于Minor GC因此尽可能将对象分配在新生代，可以通过查看日志最大限度降低对象直接进入老年代的情况） 3.永久代/元空间大小：-XX:PermSize=N //方法区 (永久代) 初始大小 -XX:MaxPermSize=N //方法区 (永久代) 最大大小 1.8使用的是元空间： -XX:MetaspaceSize=N -XX:MaxMetaspaceSize=N 4.指定垃圾回收器： -XX:+UseSerialGC -XX:+UseParallelGC -XX:+USeParNewGC -XX:+UseG1GC -XX:+UseConcMarkSweepGC
12. Java7之前，方法区位于**永久代**，永久代和堆相互隔离，java7中，**static变量**（静态变量如果是基本类型，则存储方法区，如果是对象，则引用存储方法区，实例存储堆）从永久代永久移植到了堆中，java8中，取消了永久代，方法区存放于**元空间**，元空间仍然与堆不相连，但与堆共享物理内存，逻辑上可以认为在堆中。
13. Java中class对象：获取：Class.forName()，类名.class，instance.getClass()。类的Class对象是存放在堆中。类的元数据（元数据并不是类的Class对象！Class对象是加载的最终产品，类的方法代码，变量名，方法名，访问权限，返回值等等都是在方法区的）才是存在方法区的。
14. Java中对象的三种状态：可触及状态（即有引用），可复活状态（没有引用，准备释放内存，调用finalize方法），不可触及状态（执行完finalize方法后，此对象还是没有被引用，则垃圾回收器会真正的回收此对象）
15. 触发回收的时机：1.年轻代Eden满了就minor GC。如果是CMS收集器，那么老年代满了的时候，会触发Marjor GC，非CMS老年代满了直接Full GC。2.CMS发现下述两种情形，会触发Full GC：a.在垃圾收集时发现年轻代的Survivor区放不下，而且老年代也放不下，直接Full GC。b.大对象想直接进入老年代，但是老年代空间不足，直接Full GC。3.显示调用System.gc()函数，可以触发Full GC

## Mysql

1. MyISAM与InnoDB的区别（锁，事务，索引，外键，mvcc）：1.myisam只支持表级锁，而Innodb支持表级锁和行级锁，默认为行级锁 2.只有innodb支持事务 3.只有innodb支持外键 4.只有innodb支持mvcc 5.innodb支持聚簇索引，myisam只支持非聚簇索引
2. Mysql事务(ACID:原子，一致，隔离，持久)：脏读，不可重复读，幻读，丢失修改。事务隔离级别：读写未提交，读写已提交，可重复读，可串行化
3. Mysql上的锁（行级锁3，表级锁，读写锁）：1.行级锁：record lock（给一条记录加锁，不能改变这条记录上的值），gap lock（记录本身不加锁，锁定记录边上的间隙，where A=10，不能插入/删除A=10的行），next-key lock（前两种锁的结合，锁定一个范围，可以解决幻读问题，where A between 10 and 30 确保10到30范围的既不能插入/删除，也不能修改）（说明：其中A为非唯一索引，RR隔离级别）2.表级锁：对当前操作的整张表加锁，实现简单 ，资源消耗也比较少，加锁快，不会出现死锁 。 3.读写锁：加了读锁还可以加读锁，不能加写锁，加了写锁不能加任何锁。Select虽然为读锁，但只有在**可串行化的隔离级别才加读**锁，**其他隔离级别因为mvcc，快照读**，不加锁。Update/insert/delete**加写锁**
4. 不同隔离级别的加锁情况（主键，唯一，普通，无）：RC：主键索引（锁定某一行），唯一索引（锁定某一行数据与二级索引），非唯一索引（给满足条件的行与二级索引加锁**record lock**），无索引（**先全表加锁，在释放不满足条件的锁**） RR：唯一索引，主键索引于上同，非唯一索引（给满足条件的二级索引与行加**next-key**锁），无索引（**全表加**next-key锁），innodb默认为RR级别
5. Mvcc：是个行级锁的变种，它在普通读情况下避免了加锁操作（没有读锁也没有写锁），因此速度更快，缺点：每行记录都需要额外的存储空间，需要做更多的行维护和检查。快照读（即mvcc）：不加锁。**当前读**：可以指定读锁与写锁（**手动加锁**）如：select…lock in share mode;（读锁） select…for update（写锁） 以上都是针对select语句，而update/insert/delete都是当前读，都要加写锁。
6. Mvcc的实现原理：undolog的工作过程：开始事务，记录数据行数据快照到undo log，更新数据，将undo log写到磁盘，将数据写到磁盘，提交事务。Innodb通过undo log保存了已更改行的旧版本的信息的快照，InnoDB的内部实现中为每一行数据增加了三个隐藏列用于实现MVCC。DB\_TRX\_ID：插入或更新行的最后一个事务的事务标识符。DB\_ROLL\_PTR：写入回滚段的撤消日志记录（若行已更新，则撤消日志记录包含在更新行之前重建行内容所需的信息） DB\_ROW\_ID：行标识 MVCC只在READ COMMITED 和 REPEATABLE READ 两个隔离级别下工作。 Select：
7. 索引：**唯一索引**（unique，新插入数据时，检查字段值是否**重复**，重复则拒绝插入），**普通索引**（index，普通索引的唯一任务是加快**访问速度**，应该给那些经常出现的查询条件创建索引），**主键索引**（primary key，必须为表创建一个主键索引，如果没有指定，则系统默认帮我们创建）
8. 为什么使用索引：1.通过唯一索引，可以确保表中数据的**唯一性** 2.加快检索**速度** 3.将随机IO变为**顺序IO** 4.加速表与表之间的连接，利于实现数据库**完整性** 为什么不每一个列都创建一个索引呢？1.对表中的索引修改时，索引也要维护，降低了数据的**维护**速度 2.索引需要占据**物理空间** 3.创建索引和维护索引需要耗费时间
9. Mysql基本存储结构：mysql的记录都存储在页中（页默认大小为16K），各个页可以组成一个**双向链表**，每个数据页中的记录（数据行）又可以组成一个**单向链表**，每个数据页都会为存储在它里边儿的记录生成一个**页目录**，在通过主键查找某条记录的时候可以在页目录中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录，以**非主键**作为搜索条件：只能从最小记录开始依次**遍历单链表**中的每条记录。页中的每条记录，页与页之间的记录，都是**有序**的。 查找顺序：先从根节点开始查找，定位到下一层的某个节点，然后逐层查找，直到数据存放的页，最后根据页目录查找出数据。
10. B树与B+树（结构，优点2）：B树的所有节点既**存放键(key) 也存放数据(data**);而B+树只有叶子节点存放 key 和 data，其他内节点只存放key。Mysql中，myisam与innodb使用的都是B+树。 为什么使用b+树？1.b+树的磁盘读写代价更低（相同数据时，b+树会更矮胖，相对b树加载的页就更少，IO次数更少）更稳定（因为只有叶节点存储数据，所以每次都要加载到叶节点），2.方便扫库，b树需要中序遍历，而b+树只需要扫描叶子节点。

磁盘块2中的数据都小于磁盘块1，磁盘块3中的数据介于17，35之间，磁盘块5中的数据大于磁盘块1中的数据。





1. 非聚簇索引与聚簇索引：非聚簇索引：即索引结构和数据分开存放的索引，**Myisam**使用的索引，innodb中使用的**二级索引**（唯一索引，普通索引都是二级索引，也叫辅助索引）都是非聚簇索引。聚簇索引：即索引结构和数据一起存放的索引，Innodb**主键**使用的索引为聚簇索引。二级索引可能回表，也可能不回表，如覆盖索引。Myisam中，B+树叶节点的data域存放的是**数据记录的地址**，在innodb中，二级索引存放的是索引和**主键的值**，需要查询的数据索引中没有时，只能回表。
2. 覆盖索引：覆盖索引即**需要查询的字段正好是索引的字段**，那么直接根据该索引，就可以查到数据了， 而无需回表查询。如index（name,city），如果只是根据name查询city，那么就没有回表。
3. 哈希索引：对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快，但是范围查询却无能为力。
4. 最左前缀：联合索引如index(name,id)是**先对name排序**，再对name相同的id排序，所以id可以说是无序的。所以如果查询name或者name与cid用的是ref查找类型，即用上了索引的二分查找，而**只查找cid**用的是index类型，即对整个索引进行**扫描**（查找的是联合索引的一部分才会运用此类型，也会触发索引）。mysql查询**优化器**：我们的索引是（name，cid），而查询用的是where name=xx and cid=xx，那么可以正常用索引进行查找，如果用的是where  cid=xx and name=xx 呢？他就会帮我们优化成前一个查询。**最左前缀原则指的是**，如果查询的时候查询条件**精确匹配索引的左边一列或连续几列**，则此列就可以被用到，同时索引只能匹配相等，遇到**范围查询**如<,>，like等就不能进一步匹配了
5. 一条sql语句的执行过程：mysql主要分为两层：server层（主要包括连接器，查询缓存，分析器，优化器，执行器，日志模块binlog）与存储引擎层（负责数据的存储与读取，支持myisam与innodb等，其中innodb有日志模块redolog）。**连接器**：身份认证和权限相关(登录 MySQL 的时候) **查询缓存**：执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用） **分析器**：没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。 **优化器**：按照 MySQL 认为最优的方案去执行 **执行器**：执行语句，然后从存储引擎返回数据。
6. 一条写sql语句的执行过程（基本上按照查询流程走）：执行更新操作时需要用到日志模块：binlog（归档日志），所有的存储引擎都可以使用。Innodb还自带了一个日志模块redo log（重做日志）。顺序：调用innodb引擎的api接口，写入数据，innodb把数据保存在内存中，同时记录redo log，此时redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态，更新完成。 为什么redo log要引入prepare预提交状态？为了保持数据一致性，如果先写redo log直接提交，然后写bin log写完redo log后机器就挂了，那么binlog没有记录，后续备份时会造成数据丢失。先写binlog在写redo log时同理。
7. 大表优化（记录过多时）：1.限定数据范围：禁止不带任何数据范围条件的查找（即不能查找全部） 2.读写分离：主库负责写，从库负责读。 3.垂直拆分：根据列进行拆分，如把一张有10列的表拆分成两张各5列的表（可以减少I/O次数，简化表结构） 4.水平分区（分库）：表结构不变，表中的数据分散到不同的服务器上。两种分库常见方案：客户端代理（分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现），中间件代理（在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中）。
8. Sql执行慢的原因：一直执行慢：1.没用到索引（where后面直接范围比较，字段本身就没有索引，where后面的等式左边用了运算（如where c-1=1000就不会用索引，而where c=1000+1就会用索引），where后面字段用了函数操作） 2.系统选错索引：当范围查询时，可能走索引也可能全表扫描，系统判断那个代价小就用那个，基数越大则走索引更有优势（基数是索引的区分度，索引上不同的值越多，基数越大，系统通过采样计算基数所以可能出错），我们可以强制走索引（select \* from t force index(a) where a<1000）. 2.偶尔执行慢：redolog写满了，内存不够用，拿不到锁。
9. Sql执行顺序：form，join，on，where，group by，sum/avg，having，select，distinct，order by，limit （group by是分组，与count与avg，sum等函数一起使用，用having限定分组，如限定只选取记录条数大于2的 having count(\*)>2）(首先from拿到表，然后join与on把多表合成一个表，where从表中限定符合条件的行，对符合条件的行分组，对分组做操作如count计算，用having限定符合条件的分组，select选出初步结果，用distinct对结果去重，用order by对结果排序，用limit对结果限定条数)
10. Myisam为什么查询比较快？myisam适合于读多，并发少的场景。1.innodb寻址要映射到块，在到行，myisam记录的直接是文件的offset，定位比innodb快 2.innodb还要维护mvcc一致。
11. Explain：使用explain查询和分析sql的执行记录，可以进行sql的性能优化。用法：explain select \* from table; 字段介绍：select\_type查询类型，1.simple简单的select，不适用union或者子查询。 Possible\_keys：查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用 key：查询中实际使用的索引，若没有使用索引，显示为NULL

## Redis

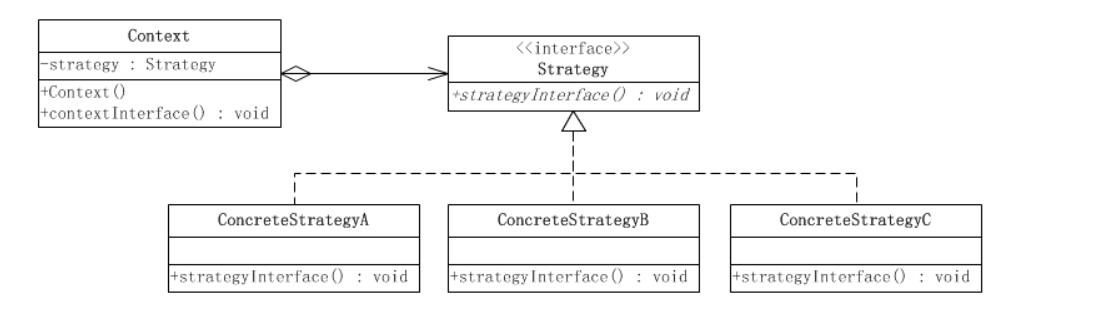
1. 链接：https://mp.weixin.qq.com/s?\_\_biz=MzI4Njg5MDA5NA==&mid=2247484609&idx=1&sn=4c053236699fde3c2db1241ab497487b&chksm=ebd745c0dca0ccd682e91938fc30fa947df1385b06d6ae9bb52514967b0736c66684db2f1ac9###rd
2. redis：Redis是基于内存，常用作于缓存的一种技术，并且Redis存储的方式是以key-value的形式。
3. 为什么要用缓存：1.高性能，缓存查询速度比数据库查询速度快。2.高并发：缓存分担了部分请求，这样用户的一部分请求会直接到缓存这里而不用经过数据库，支持更快的并发。
4. Redis的键值对：key一定是字符串，value任意（String，list，hash，set，sorted set）。对象系统：key与value并不是直接使用string，list等几种数据类型，而是基于这些数据类型创建了一个对象系统（redisObject），对象包括：实际的对象类型（5种），对象编码格式（7种如ziplist等），指针等。
5. Redis底层数据结构：
6. Redis中的数据库：redis默认有16个数据库，数据库之间是相互隔离的。底层结构：**1个server包括一个大小为16的db数组**，1个client指向当前正在使用的db，每个db包含**两个哈希表**，一个哈希表包含此数据库所有key-value，一个包含所有key的过期时间
7. （重点）过期策略（key到了过期时间会立马删除掉吗？）：定时删除(对内存友好，对CPU不友好)：到时间点上就把所有过期的键删除了。惰性删除(对CPU极度友好，对内存极度不友好)：每次从键空间取键的时候，判断一下该键是否过期了，如果过期了就删除。定期删除(折中)：每隔一段时间去删除过期键，限制删除的执行时长和频率。 Redis采用的是惰性删除+定期删除两种策略，所以说，在Redis里边如果过期键到了过期的时间了，未必被立马删除的。
8. （重点）如何**判断一个键是否过期**呢？首先检查数据库中的过期时间哈希表，如果没在里面说明不会过期，如果存在就获取过期时间，检查当前的时间是否大于过期时间，是则过期。 **定期删除原理**：从数据库中的过期时间哈希表中随机选取20个键，删除其中过期的键，如果过期的键比例超过25%则重复上几步，扫描时间上线默认25毫秒。
9. （重点）内存淘汰策略（如果定期删除漏掉了很多过期key，也没及时去查(没走惰性删除)，大量过期key堆积在内存里，导致redis内存块耗尽了（可以设置内存最大使用量），咋整？）：allkeys-lru：从所有数据集中挑选**最近最少**使用的数据淘汰。volatile-lru 设置了过期时间的key参与近似的lru淘汰策略（即随机选取一定的值，并从中进行淘汰，而不是所有数据集）。
10. Lru实现原理（https://zhuanlan.zhihu.com/p/34133067）：可以用双向链表和hashmap实现。Key存储在hashmap中，使得get(key)和save时间复杂度都是1，hashmap的value指向双向链表某个节点，双向链表中存储了key对应的value。1.Save时，首先判断hashmap中是否存在key，如果节点存在，更新节点的值，并把这个节点移动队头。如果不存在，需要构造新的节点，并且尝试把节点塞到队头，如果LRU空间不足，则通过 tail 淘汰掉队尾的节点，同时在 HashMap 中移除 Key。2.get时，通过hashmap定位此节点在链表中的位置，然后把此节点插入表头，并返回数据。
11. Redis会随机选取固定数目的key，然后比较他们的lru访问时间（在server中提供了一个全局lrulock供每个object更新自己的时钟，sever每100ms更新一次lrulock，每个object初始化/更新的时候会从lrulock中得到现在的时钟），然后淘汰最近最少使用的key，固定值可以改变，值越大，消耗越高，volatile就越接近allkeys，默认为15.
12. Redis持久化（因为redis是基于内存的，所以重启后数据会全部丢失，所以需要我们把数据写入到磁盘中便于恢复）：1. RDB(基于快照)，将某一时刻的所有数据保存到一个RDB文件中。 2. AOF(append-only-file)，当Redis服务器执行写命令的时候，将执行的写命令保存到AOF文件中。
13. RDB（默认持久化方式）:可以手动（SAVE,BGSAVE）/定期（BGSAVE）执行持久化，RDB文件是一个经过压缩的二进制文件，redis启动时，如果发现有RDB文件，就会自动载入RDB文件，服务器在载入RDB文件期间，会处于阻塞状态。生成RDB文件的命令：SAVE:阻塞redis服务器进程。BGSAVE:由子进程生成RDB文件，服务器进程继续接受请求。 默认：（save 900 1）在900秒(15分钟)之后，至少有1个key发生变化等，就会执行BGSAVE命令 （save 300 10）（save 60 10000）（三个条件触发一条，就会BGSAVE）
14. AOF：redis执行了一个写命令后（如set），以协议格式将被执行的命令追加到aof\_buf缓冲区的末尾，随后调用flushAppendOnlyFile()函数考虑是否将缓冲区的内容写入到AOF文件（操作系统的内存缓冲区），最后确定是否将AOF文件同步（即写入磁盘） faof函数：由三种写入文件选项：1.有数据修改就写入 2.默认，每秒同步一次 3.从不同步
15. AOF文件载入：创建一个伪客户端，不断从aof文件中读取写命令并执行。 AOF重写：**可以让aof文件体积变小**（比如多条set语句可以合成一条），BGREWRITEAOF命令，aof重写不用读取原来的aof文件，而是通过**读取数据库**生成命令并用生成的新的aof文件**替换旧的**。解决重写导致aof与数据库不一致：创建的**子进程**执行重写操作，还没有重写完时，客户端发起的写操作会到aof缓冲区中，子进程完成重写后，把缓冲区中的数据写入aof文件。
16. 过期策略：RDB：创建RDB文件时，程序会对数据库中的所有键值进行过期检查，过期的键值不写入文件，载入文件时过期的不会被载入。 AOF：如果过期的键还没被删除，AOF也会写入，当过期的键被删除后（惰性删除或定期删除），会**追加一条del**命令显示该键被删除，重写aof文件时，过期的键会被忽略。 复制模式：主服务器来控制从服务器统一删除过期键(保证主从服务器数据的一致性)，主服务器删除后，根据命令传播，从服务器也会删除，主服务器没删除，从服务器遇到客户端访问过期键任然会把过期键发送给客户端。
17. 优缺点：RDB：优：载入时恢复数据快、文件体积小。缺：会一定程度上丢失数据(因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。) AOF：优：丢失数据少(默认配置只丢失一秒的数据)。缺：恢复数据相对较慢，文件体积大 如果redis同时开启两者，服务器优先使用aof还原数据，因为aof丢失数据少。服务器可以同时使用两者。
18. Redis事件：文件事件：客户端与服务器网络通信，时间事件：一些定时操作
19. 文件事件：I/O多路复用程序会同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生，将所有产生事件的Socket放到一个队列，然后每次处理一个，依次有序向文件事件分派器传送套接字，当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O 多路复用程序才会继续向文件事件分派器传送下一个套接字。
20. 时间事件：Redis服务器将时间事件放在一个链表中，当时间事件执行器运行时，会遍历整个链表。其中，时间事件包括周期性事件（serverCron函数，包括持久化，清理过期键等）与定时事件
21. 1. 文件事件和时间事件之间是合作关系，服务器会轮流处理这两种事件，并且处理事件的过程中不会发生抢占。2. 时间事件的实际处理事件通常会比设定的到达时间晚一些
22. Redis为什么这么快？1. 纯内存操作 2. 核心是基于非阻塞的IO多路复用机制 3. 单线程避免了多线程的频繁上下文切换问题
23. Redis单线程不是浪费cpu吗？因为单线程已经够用了，CPU不是redis的瓶颈。Redis的瓶颈最有可能是机器内存或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。
24. 一个服务器可以与多个客户端建立网络连接（服务器使用链表连接多个客户端状态）。Redis服务器使用单线程单进程的方式处理命令请求。客户端与服务器交流过程：客户端将命令请求发送给服务器，服务器读取命令请求，分析出命令参数，命令执行器根据参数查找命令的实现函数，执行实现函数并得出命令回复，服务器将命令回复返回给客户端。
25. 为什么使用服务器主从架构：1.单台服务器内存有限 2.单台服务器支持并发量也有限 3.万一redis服务器挂了，所有请求全走关系数据库了，那就更炸了。（高可用，高并发）
26. 服务器主从架构是：一台主服务器，多台从服务器。主服务器负责接收写请求。从服务器负责接收读请求。从服务器的数据由主服务器复制过去，主从服务器的数据是一致的。
27. 复制功能：同步（初次同步，断线同步），命令传播（主服务器的写命令也给从服务器执行）
28. 复制的前置工作：主服务向从服务器发送SAVLEOF命令，从服务器收到命令后，记录下主服务器的ip和端口，并与主服务器建立socket，并为该socket绑定文件处理器，建立socket后，从服务器给主服务器发送ping命令，并进行身份认证
29. 完整重同步（初次同步）：从服务器向主服务器发送PSYNC命令，收到PSYNC命令的主服务器执行BGSAVE命令，在后台生成一个**RDB文件**。并用一个缓冲区来记录从现在开始执行的所有写命令。当主服务器的BGSAVE命令执行完后，将生成的RDB文件发送给从服务器，从服务器接收和载入RBD文件。将自己的数据库状态更新至与主服务器执行BGSAVE命令时的状态。主服务器将所有**缓冲区的写命令发送给从服务器**，从服务器执行这些写命令，达到数据最终一致性。
30. 部分重同步（断线同步）：从服务器接受到主服务器发送的SAVLEOF命令，判断是否是第一次执行复制，是则执行完整重同步，不是则向主服务器发送服务器的**运行id和偏移量**，主服务器判断运行id是否相同，**复制积压缓冲区是否存在丢失的偏移量的数据**，如果是，则部分重同步（即从复制积压缓冲区中将偏移量相差的部分发送给从服务器）。 偏移量：主服务器每次**传播N个字节**，就将自己的复制偏移量加上N，从服务器每次收到主服务器的N个字节，就将自己的复制偏移量加上N 复制积压缓冲区：当主服务器进行命令传播时，不仅仅会将**写命令**发送给所有的从服务器，还会将写命令入队到复制积压缓冲区里面(这个大小可以调的)
31. 哨兵机制：如果主服务器挂了，我们可以将从服务器升级为主服务器，等到旧的主服务器(挂掉的那个)重连上来，会将它(挂掉的主服务器)变成从服务器。
32. 启动和初始化sentinel：sentinel本质上是一个特殊的redis服务器，初始化时，sentinel不读取RDB,AOF文件，启动时，会将普通的Redis服务器的代码替换成Sentinel专用代码，接着，初始化Sentinel的状态，并根据给定的配置文件初始化Sentinel监视的主服务器列表，最后，Sentinel会创建两个连向主服务器的网络连接：命令连接，订阅连接
33. 获取和更新信息：Sentinel通过主服务器发送INFO命令来获得主服务器属下所有从服务器的地址信息，并为这些从服务器创建相应的实例结构，和命令连接，订阅连接。Sentinel通过命令连接以2秒一次的频率向所有主从服务器hello频道发送命令，并通过订阅连接接收hello频道的信息。
34. 判断主服务器是否下线：sentinel会**每秒一次**向所有主从服务器和其他sentinel发送**ping命令，**如果一个主服务器在down-after-milliseconds毫秒内连续向Sentinel发送**无效回复**，那么当前Sentinel就会主观认为该主服务器已经下线了（主观下线），同时它会向同样监视该主服务器的**Sentinel询问**，看它们是否也认为该主服务器是否下线，如果**足够多**的Sentinel认为该主服务器是下线的，那么就判定该主服务为**客观下线**，并对主服务器执行故障转移操作
35. 当判断一个主服务器客观下线后，会以**先到先得**的方式选出一个领头的sentinel，此sentinel会执行故障转移操作：1. 在已下线主服务器属下的从服务器中，挑选一个转换为主服务器 2. 让已下线主服务器属下的所有从服务器改为复制新的主服务器 3. 已下线的主服务器重新连接时，让他成为新的主服务器的从服务器
36. 数据丢失：1.有部分数据没有复制到从服务器，主服务器就宕机了 2. 当master脱离正常网络，与slave断开连接，但master并没有宕机，此时sentinel 认为master宕机了，然后开始选举新的master，这个时候，集群中就会有两个master，就是所谓的脑裂。此时虽然某个slave转换成了master，但可能client还没有来得及切换到新的master，还在继续向旧的master写数据，就会丢失数据了。（因为旧master再次连接的时候，会被作为一个slave挂到新的master上去，自己的数据会被清空，重新从新的master复制数据。）min-slaves-to-write 1 min-slaves-max-lag 10 这两条命令减少数据丢失
37. 缓存雪崩（请求全部走数据库）：1. 对缓存数据设置**相同的过期时间**，导致某段时间内缓存失效，请求全部走数据库。解决：在缓存的时候给过期时间加上一个**随机值**，这样就会大幅度的减少缓存在同一时间过期 2. **Redis挂掉了**，请求全部走数据库。 解决：事发前：实现Redis的**高可用**(主从架构+Sentinel 或者Redis Cluster)，尽量避免Redis挂掉这种情况发生 事发时：万一Redis真的挂了，我们可以设置**本地缓存(ehcache)+限流(hystrix)**，尽量避免我们的数据库被干掉(起码能保证我们的服务还是能正常工作的) 事发后：**redis持久化**，重启后自动加载持久化数据，快速恢复缓存。
38. 缓存穿透：缓存穿透是指查询一个数据库**一定不存在的数据**。由于缓存不命中，并且出于容错考虑，如果从数据库查不到数据则不写入缓存，这将导致这个不存在的数据**每次请求都要到数据库去查询**，失去了缓存的意义。解决：1. 我们可以使用**布隆过滤器**(BloomFilter)（布隆过滤器判断数据库中没有这个数据，则直接返回null，而不是去数据库查询）或者压缩filter提前拦截，不合法就不让这个请求到数据库层！ 2. 当我们从数据库找不到的时候，我们也将这个**空对象设置到缓存**里边去。下次再请求的时候，就可以从缓存里边获取了，这种情况我们一般会将空对象设置一个较短的过期时间
39. 布隆过滤器：一种专门用来解决**去重**（用set去重内存消耗太大）问题的高级数据结构（并不是完全精确），当布隆过滤器说某个值**存在**时，这个值可能不存在；当它说**不存在**时，那么一定不存在。（redis4.0的server内置了布隆过滤器） **缺点**：不准确，删除困难（不能简单的置为0，会妨碍其他值1）。
40. 原理：布隆过滤器本质上由长度为 m 的**位（bit）数组**（仅包含 0 或 1 位值的列表）组成，最初所有的值均设置为0，当我们向布隆过滤器添加数据时，会使用 多个 hash 函数对 key 进行运算，然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1。向布隆过滤器查询时，与添加操作相同得到多个不同位置，查看对应的位置 是否都为 1，只要有一个位为0，那么说明布隆过滤器中这个key不存在。如果都为1，并不是一定存在，可能是由于其他key存在导致的。
41. 使用：1. 不要让实际元素数量远大于初始化数量 2. 当实际元素数量超过初始化数量时，应该对布隆过滤器**进行重建**，重新分配一个 size 更大的过滤器，再将所有的历史元素批量添加。
42. 缓存击穿：**某个缓存过期后**，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。 解决：对关键代码（从数据库读取数据）**加入双重校验锁**，每一刻都只有一个线程能访问数据库，且获取锁之前和之后都会判断缓存中是否已经存在数据了，存在则直接从缓存中读取。 如果应用程序是集群架构，则采用**分布式锁**。
43. 缓存不一致：操作缓存一般是直接删除即使之失效，而不更新（因为如果多写少读时，每次写入都更新缓存没有必要，不如直接删除，等到写的时候才读取）
44. 最初级：Cache Aside pattern：1.**读的时候**，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应 2. **更新的时**候，先更新数据库，然后再删除缓存。 问题：更新时，如果只更新了数据库，而删除缓存失败，那么会导致后面读写的都是**错误的数据** 解决方法：先删除缓存，再更新数据库。那么，如何能保证原子操作呢？缓存失效，线程A查询数据库，得到一个旧值，线程B将新值写入数据库，线程B删除缓存，线程A写入旧值进缓存，最后导致缓存里的还是旧值，而数据库是新值 （A写入缓存前，B更新了数据库与缓存）
45. 先删除缓存再更新数据库可能出现的问题：A线程为更新线程，A线程先删除了缓存数据（1），还没等到更新数据库（1），B线程（读取线程）就读取了缓存，发现被删除，然后读取数据库，再写入缓存（1），这时，A线程成功写入数据库（2），最后数据库中的数据为2，而缓存中的数据为1，不一致。解决方法：异步串行化：更新和读取数据时，把数据的hashcode路由到jvm中的某个队列（因为根据hashcode路由，所以可知一个数据一个队列，一个队列对应一个工作线程），然后逐个执行队列中的请求，这样就保证了更新全部完成之后才能读取，读取彻底完成之后，才能更新。可能导致的问题：读请求长时间阻塞（当前面有很多条写数据时），读请求并发量过高。（可以根据具体情况添加服务器，多台服务器处理写请求）
46. Redis集群（集群中的每一个 Redis 节点都**互相两两相连**，客户端任意直连到集群中的任意一台，就可以对其他 Redis 节点进行**读写的操作**，因为如果槽不属于自己管，那么就会使用一个特殊的 MOVED 命令来进行一个跳转，告诉客户端去连接这个节点以获取数据）：集群使用16384个哈希槽，数据库中的每个键都属于这 16384 个哈希槽的其中一个，集群使用公式 CRC16(key) % 16384 来计算键 key 属于哪个槽， 其中 CRC16(key) 语句用于计算键 key 的 CRC16 校验和，**集群中的每个节点负责处理一部分哈希槽**，集群中的每个节点两两互相连接。 集群的作用：**数据分区**（提高了可以存储的容量和集群响应速度），**高可用**（集群支持自动故障转移，当某一节点发生故障，集群仍然可以提供服务）。
47. Redis集群最少台数：6台，3台master节点，每个master节点对应1台slave节点，如果1台master对应2台slave，则需要9台服务器。
48. 数据分区方案：1.哈希值%节点数（即服务器数），问题：当新增或删减节点时，节点数量发生变化，系统中所有的数据都需要 重新计算映射关系，引发大规模数据迁移。 2.一致性哈希分区：将整个哈希值空间组织成一个虚拟的圆环，范围(即圆的周长)是 [0 , 2^32-1]，对于每一个数据，根据 key 计算 hash 值，确数据在环上的位置，然后从此位置沿顺时针行走，找到的第一台服务器就是其应该映射到的服务器，将 增减节点的影响限制在相邻节点，当节点数量少时，对单个节点的影响可能很大，如去掉某个节点，可能此节点的数据都会转移到下一个节点 3. 带有虚拟节点的一致性哈希分区：redis使用的方案，在上个方案的基础上增加了虚拟节点的概念（槽），**每个实际节点包含一定数量的槽**，槽是数据管理和迁移的基本单位。槽 解耦了数据和实际节点 之间的关系，增加或删除节点对系统的影响很小，如删除某个节点，只需要把它的槽均匀地分配给其他节点便可。
49. 节点通信机制：每个节点，都提供两个TCP端口：普通端口（程序员指定的端口，用于client连接的端口），集群端口（普通端口+10000，**集群端口只用于节点之间的通信**，如搭建集群、增减节点、故障转移等操作时节点间的通信；不要使用客户端连接集群接口。为了保证集群可以正常工作，在配置防火墙时，要同时开启普通端口和集群端口）
50. Gossip协议：节点通信可以分为几种类型：单对单，广播，Gossip等。1.广播：是指向集群内所有节点发送消息，优点 是集群的收敛速度快(集群收敛是指集群内所有节点获得的集群信息是一致的)，缺点 是每条消息都要发送给所有节点，CPU、带宽等消耗较大。2.Gossip协议：每个节点都 **“随机” 的与部分节点通信** （并不是真正的随机，而是根据特定的规则选择通信的节点），经过一番杂乱无章的通信，每个节点的状态很快会达到一致。Gossip 协议的优点有：负载 (比广播) 低、去中心化、容错性高 (因为通信有冗余) 等；缺点 主要是集群的**收敛速度慢**
51. 消息类型：集群中的节点采用 **固定频率（每秒10次）** 的 定时任务 进行通信相关的工作。节点间发送的消息主要分为 5 种：meet 消息、ping 消息、pong 消息、fail 消息、publish 消息。
52. 数据结构：节点为了存储集群状态而提供的数据结构中，最关键的是 clusterNode 和 clusterState 结构：前者记录了**一个节点**的状态（如创建时间，节点id，端口，槽等），后者记录了**集群**作为一个整体的状态（**槽分布信息**，自身节点等）。
53. HyperLogLog（场景：统计一个页面每天有多少用户访问n）使用：如果我们用set存储的话，需要花费大量的内存，而用hll只需要花费不多于12KB的内存（16384\*6/8/1024），用pfadd加入字符串，用pfcount获取n值
54. 引入：伯努利实验，每次出现反面就停止，记录这次从开始到出现反面抛的次数k，总共抛n次（出现了n次反面），取n个k中的最大值kmax，则n约等于2^kmax。优化：进行m轮的n次实验（总共最多进行了m\*n\*kmax次抛硬币），每次都找出一个kmax，取m个kmax的调和平均数，则n约等于2^kmax。（利用伯努利实验的改进版，我们可以根据k求出n，n即使我们要求的总共访问用户量）对应：我们可以根据存入的字符串的hash值从右边起首次出现的1（即硬币中的反面）右边的连续0的个数记为k，一共存入了n个字符串（即n个用户），得到n个k，取其中的kmax，那么2^kmax就约等于n。优化：分桶，每个桶中存储的都是kmax，我们求出16384个kmax的调和平均数即可用求出n
55. Hll原理：首先，redis里面有2^14=16384个桶，**每个桶可以存储6bit**（即每个桶最大值为2^6=64，字符串被存入时，会被hash成64位的值（相同字符串hash值相同），**前14位用来分桶**，每个桶中存储在同一个桶中的所有字符串（一个桶可能存放很多不同的字符串，因为数据是百万级别）的kmax，然后根据16384个kmax求出**调和平均数k**，即可用2^k求出n <https://juejin.im/post/5c7900bf518825407c7eafd0>
56. Hll的存储结构：密集型存储结构：16384个桶，每个桶6bit，**即12KB**。稀疏存储结构（便于缩小内存空间）：当都是0时存储一个非0的计数值即可(01xxxxxx yyyyyyyy)，如初始化都是0时，用01111111 11111111（即2^14）表示，只需要2字节 <https://github.com/Snailclimb/JavaGuide/blob/master/docs/database/Redis/redis-collection/Reids(4)%E2%80%94%E2%80%94%E7%A5%9E%E5%A5%87%E7%9A%84HyperLoglog%E8%A7%A3%E5%86%B3%E7%BB%9F%E8%AE%A1%E9%97%AE%E9%A2%98.md>
57. Redis分布式锁（分布式环境（即多台应用服务器）中的java代码如何加锁？适用于非集群redis环境）：1.在redis中设置**一个key**，只有获取到了key的应用服务器才能执行加锁代码块 2.（锁超时问题）setnx虽然是原子操作，但不支持设置操作时间，可以**用set原子**设置key和它的过期时间 3.（del导致误删除问题）可以把本线程的**id当作key的value**，只有value相同，才能执行删除操作 4.（删除锁非原子操作i++问题）用**lua脚本**使之变为原子操作 5.（过期时间大于业务时间）设置一个**守护线程**，当业务线程还没完成但是却快过期时，守护线程会给key增加过期时间，当业务线程执行完毕时，会关闭守护线程 ：<https://juejin.im/post/5b16148a518825136137c8db> <https://juejin.im/post/5b737b9b518825613d3894f4> **redison**已经帮我们实现了reids的分布式锁，不用我们自己实现（用zookeeper实现分布式锁更好）
58. 集群redis分布式锁（redlock）：算法：1. 得到当前的时间，微秒单位 2. Client使用相同的key和随机数,按照顺序在每个Master实例中尝试获得锁。在获得锁的过程中，为每一个锁操作设置一个快速失败时间(如果想要获得一个10秒的锁， 那么每一个锁操作的失败时间设为5-50ms，确保client不会在已经宕机的server上浪费时间)。 3. 客户端计算出与master获得锁操作过程中消耗的时间，当且仅当Client获得锁消耗的时间小于锁的**存活时间**，并且在一半以上的master节点中获得锁。才认为client成功的获得了锁。 4. 如果已经获得了锁，Client执行任务的时间窗口是锁的**存活时间减去获得锁消耗的时间**。 5. 如果Client获得锁的数量不足一半以上，或获得锁的时间超时，那么认为**获得锁失败**。客户端需要立刻尝试在**所有的master节点中释放锁**， 即使在第二步中没有成功获得该Master节点中的锁，仍要进行释放操作。 失败重试：如果一个Client无法获得锁，它将在一个**随机延时后开始重试**。使用随机延时的目的是为了与其他申请同一个锁的Client错开申请时间，减少脑裂(split brain)发生的可能性 崩溃和恢复：如果我们的节点没有持久化机制，client 从 5 个 master 中的 3 个处获得了锁，然后其中一个重启了，这时注意整个环境中又出现了 3 个 master 可供另一个 client 申请同一把锁！ 违反了互斥性。如果我们开启了 AOF 持久化那么情况会稍微好转一些，因为 Redis 的过期机制是语义层面实现的，所以在 server 挂了的时候时间依旧在流逝，重启之后锁状态不会受到污染。但是考虑断电之后呢，AOF部分命令没来得及刷回磁盘直接丢失了，除非我们配置刷回策略为 fsnyc = always，但这会损伤性能。解决这个问题的方法是，当一个节点重启之后，**我们规定在 max TTL 期间它是不可用的**，这样它就不会干扰原本已经申请到的锁，等到它 crash 前的那部分锁都过期了，环境不存在历史锁了，那么再把这个节点加进来正常工作
59. 发布订阅：发布者向频道发布信息，订阅了此频道的所有消费者都会自动接收到此条信息。模式订阅：h?llo，可以订阅hello，hallo等频道。Hell\*，可以订阅hellooo，helllll等频道
60. 发布订阅原理（pubsub）：服务器维护了一个频道字典，字典的key为频道，value为订阅了此频道的所有client组成的链表。订阅就是向链表中加入客户端，发布就是遍历此频道下的链表并发送消息。服务器维护了一个模式链表，新增一个模式就把此模式加入链表，而每个模式又有一个客户端链表，发布时，服务器先发送信息给此频道下面的所有客户端，然后遍历模式链表，只要**模式与此频道匹配**，就把信息发送给此模式下的所有客户端。 问题：1.如果在某个客户端宕机时发布信息，那么此客户端就**接受不到此条信息**了，且发布信息的服务器也不知道 2.消息**不能持久化**，redis重启后所有消息都会被丢失
61. 持久化的发布订阅系统（Stream，redis5.0）：每个stream有一个key（首次添加消息时自动创建），对应一个消息队列，消息队列中存储多个消息，每个消息包括消息id和消息内容，消息id为‘整数-整数’形式，前一个整数为毫秒数，后整数为此消息为此毫秒数内生成第几个消息，消息内容为普通键值对。消费者组可以独立消费信息，也可以多个消费者同时加入一个消费者组进行组内消费，在消费者组中，多个消费者只有一个消费者能消费到某条信息，确保消息的唯一性。消费者组有一个last\_delivered\_id，用来表示消费者组消费在Stream上消费位置的游标信息。每个消费者有一个pending\_ids，用来表示已经被客户端获取，但是还没有ack的消息，用来确保消费者宕机后还能继续消费宕机其间的消息（消费者消费完后会发送ack给服务端）。
62. Redis使用场景：1.显示最新项目列表（如微博显示最新评论，每次有评论就加入到redis种的list） 2.排行榜（用排序set） 3.利用hll统计（pfadd增加，scard统计命令）多少用户访问了某个资源 4.利用布隆过滤器去重（如抖音的推荐视频要去重） 5.计数（INCRBY原子递增操作） 6.分布式锁 7.频率限制，限制用户5秒内只能访问某接口2次（使用lua脚本和incr计数操作）
63. Redis存储数据：最多能存放2.5亿个key，一个string类型的value最多512M，list/set/hashmap/sortedset元素个数最多为2^32-1。

## Java基础语法

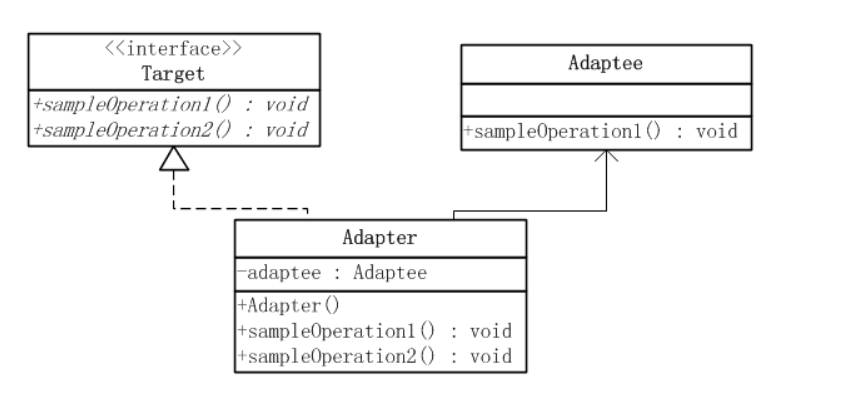
1. overriding与overloading（返回值，参数），枚举（用法，定义：类中，接口中，自定义），hashcode（equels：String，Integer，对象与hashcode，线程），数组（初始化，二维数组，使用：Arrays），常量（类/方法，变量：成员/局部/String/static final（编译），对象），正则表达式（Matcher：类不能为私有，构造方法可以。Pattern与Matcher。Replace。//w），String（相加（变量，常量），常量池（不可变对象）与堆，intern，build与buffer，方法：转换，截取，定位），基本数据类型（大小（浮点，char），字符串格式化：printf/format，dowhile），变量（初始化：局部/成员/数组/对象，类型转换（基本，String），大数，移位/&/^/|）,包装类（String：final/+，是否有new/范围，常量池（4），+与拆箱），函数（传参（基本类型/对象），可变参数），时间（SimpleDateFormat，Date与Calender），static（父子加载，类加载，静态常量不加载类），反射（创建对象/数组，获取变量/方法，调用方法），继承（子类初始化父类必定初始化，子类调用父类static，子类不加载），异常（继承关系，抛出/声明异常，执行语句，finally/return顺序，自定义异常），接口内部类（接口中的static：变量/方法，静态/非静态内部类：资源访问，创建，匿名/局部），范型（声明，extends，Class<T>，）
2. 范型的理解和使用场景：在没有用范型时，想要实现一个通用的，可以处理不同类型的方法，需要使用Objec类强转：1.每次使用时**都需要强制转换**成想要的类型2.在编译时编译器并不知道类型转换是否正常，运行时才知道，**不安全**。而使用范型的好处：1.类型**安全**，写编译时就能发现错误 2.**消除强制类型转换**
3. 反射的机制与作用：反射机制允许程序在**执行时动态获取**某个类的各种信息，也可以实现动态**创建**类的对象、**变更**属性的内容或执行特定的方法等功能，从而增强了程序的灵活性与可移植性。反射机制的作用：1.当需要使用的类还没写出来时，可以用反射而不是import，避免了编译期间报错。 2.**解耦**，可以把类名字放置在配置文件中，要修改时只需要修改配置文件即可，而不用修改代码。3.**idea**中的方法属性提示，就是运用反射获取方法属性的例子。
4. 反射缺点：1.性能问题，要慢于直接调用 2.降低代码可读性，维护比较难 3.内部属性暴漏，反射能访问私有属性和方法，可能带来一些问题。
5. 多态：（存在继承关系，子类重写父类方法，父类引用指向子类实例）一个引用变量到底会指向哪个类的**实例对象**，该引用变量发出的**方法调用**到底是哪个类中实现的方法，必须在由程序运行期间才能决定。从而实现**不修改程序代码就可以改变程序运行时所绑定的具体代码**，让程序可**以选择多个运行状态**，这就是多态性。
6. 引用拷贝：引用之间直接赋值，导致两个引用指向的对象地址相同，即同一个对象 对象拷贝：调用clone方法，两个引用指向的对象不同（浅拷贝：对象中的变量拷贝，对象中的对其他对象的引用共享 深拷贝：对象中的变量拷贝，对象中的引用所指向的对象也深拷贝，消耗最大）。
7. 什么是面向对象：**对象包括属性与行为**，面向对象就是把构成问题的事物分解成一个个对象，建立对象**不是为了实现一个步骤**，而是为了**描述**某个事物在解决问题中的行为。**程序是对象的集合**，它们通过**发送消息**来告知彼此所要做的。也就是说：以对象为中心，以消息(发送消息即为函数调用)为驱动。
8. 面对对象的四个特性：封装（核心思想就是“隐藏细节”、“数据安全”，将对象不需要让外界访问的成员变量和方法私有化，只提供符合开发者意愿的公有方法来访问这些数据和逻辑，保证了数据的**安全和程序的稳定**），继承（从一般到特殊，是一种扩展关系），多态，抽象（抽象就是将一类实体的共同特性抽象出来，封装在一个新的概念(类) 中）
9. Java序列化和反序列化：序列化就是将对象写入到IO流中。序列化机制允许将实现序列化的Java对象转换为**字节序列**，这些字节序列可以保存在磁盘上，或通过网络传输，以达到以后恢复成原来的对象。序列化机制使得对象可以脱离程序的运行而独立存在。使用场景：网络上传输的对象都必须序列化（Controller传递的对象不是序列化，而是转化为json）。（Serializable只是一个标记接口，没有方法） （ArrayList的数组虽然是Transient，但是因为他实现了writeObject/readObject方法，此方法是自定义序列化，可以选择序列化内容，或者加密序列化）
10. Java中util包和sql包中都有Date类，其中，sql包中的Date继承自util包中的Date。
11. Java中数组构造：Java中的所有数组都是通过Array类实例化。Array类没有public的构造方法，数组是通过Array的newInstance()方法进行实例化。静态初始化：声明时就用{}赋值了，动态初始化：只声明大小，那个位置有什么元素后来赋值。数组的引用存储在栈中，数组对象存储在堆中（数组中存储基本类型也是存储在堆中，因为数组本身就是对象）。
12. Finalize方法：他在对象被垃圾回收前被调用。不建议用它来清除回收对象，建议用于某些非内存资源的释放，如socket，文件等。可以通过调用System.gc()来手动回收垃圾，但jvm并不保证一定会GC，执行GC也不保证一定执行finalize方法。执行finalize的过程：当对象变成不可达时，判断此对象是否覆盖了finalize方法，没有覆盖则直接回收，覆盖了，并且从未执行finalize方法，则放入队列，等待低优先级线程执行，执行完成后若还不可达，则回收。
13. Java中异常：所有异常都继承自Throwable，有两种异常：Error（系统错误，我们无法解决，如栈溢出）Exception，其中Exception又分为非运行时异常：IOExeception/ClassNotFoundException和运行时异常：RunTimeException。非运行时异常需要我们手动catch。运行时异常则不用，如1/0系统会帮我们抛出。

## 设计模式

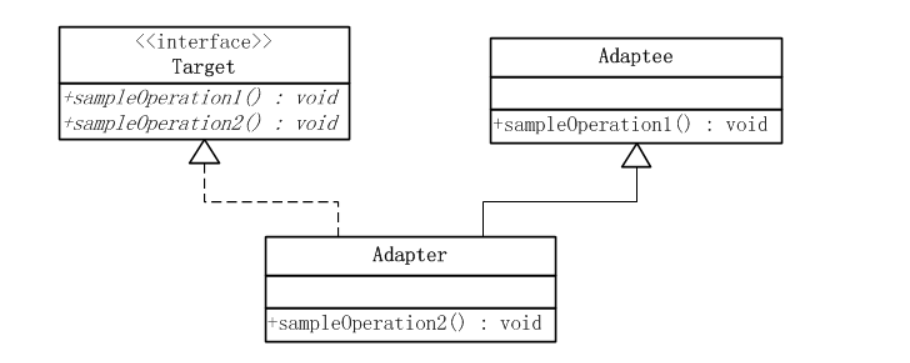
1. 单例模式：共有6种（饿汉2，懒汉3（懒加载，用的时候才加载），枚举）。懒汉/饿汉包含三部分：private static的声明，private的构造器，public static的getinstance()。其中懒汉式：双重校验锁为volatile与Singleton.class，内部类中private static的声明被放类private static的SingletonHolder的内部类中。 枚举只有两行：无任何修饰的instance和构造器。
2. 策略模式，下图中，Context新增一个replaceStrategy(Strategy s)的方法



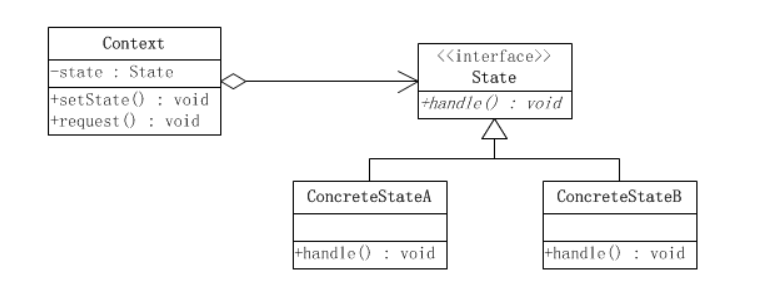
1. 适配器模式，对象适配器（target类被客户端调用，提供两个方法，1是适配前的方法，2是被adapter适配后的方法。adapter类拥有adaptee类（关联关系））https://www.cnblogs.com/java-my-life/archive/2012/04/13/2442795.html



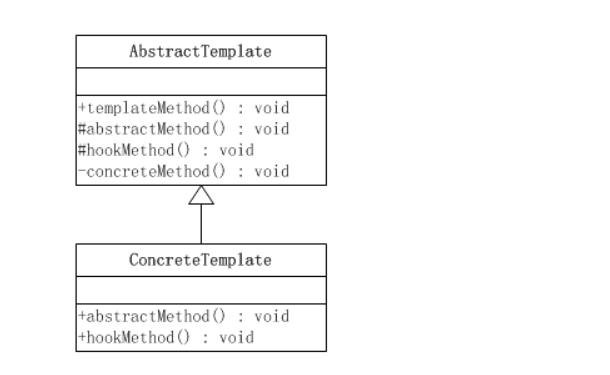
类适配器（与对象适配器不同的是，adapter类继承自adaptee类）：



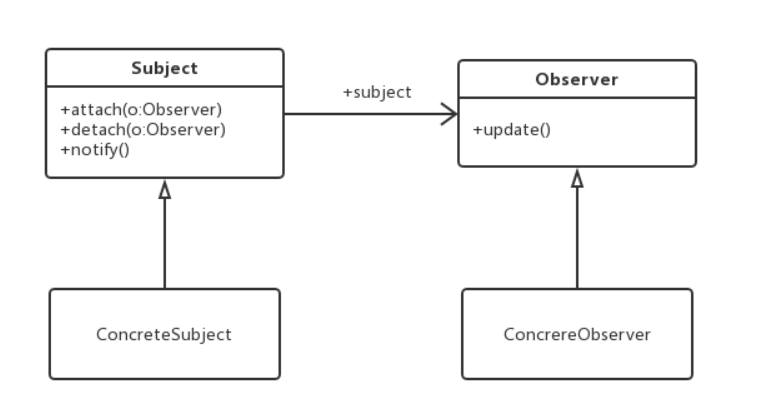
1. 状态模式（注意这里实现接口应该用**虚线**，State接口里面有每个状态都需要实现的方法，状态转移尽量写在具体的状态中而不要都写在Context中，所以Context对象应该传递给每个状态）



1. 模板方法模式（根据是否定义hook方法，会输出不同结果，hook方法就是简单的返回true或false，其他方法根据true或false实现不同功能）

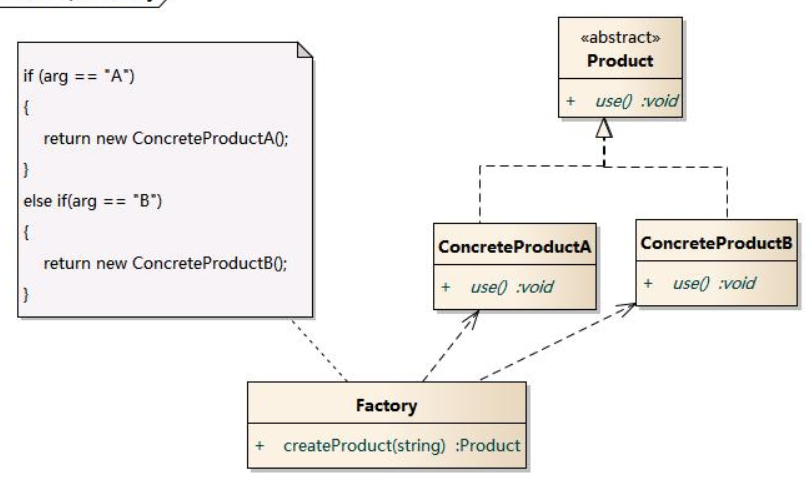


1. 观察者模式，Subject子类拥有(ArrayList<Observer>)在此Subject注册的所有Observer（关联关系），Subject有三个方法，增加观察者，去除观察者，通知所有观察者。Observe（接口）有一个方法update，每次Subject通知观察者时都会调用此Subject的update方法。

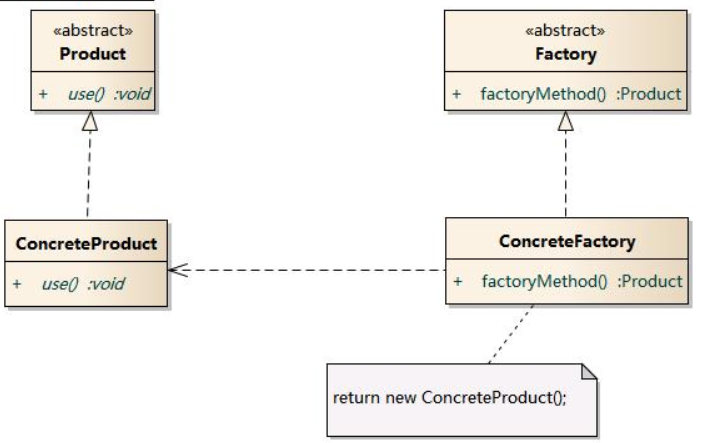


1. 工厂模式

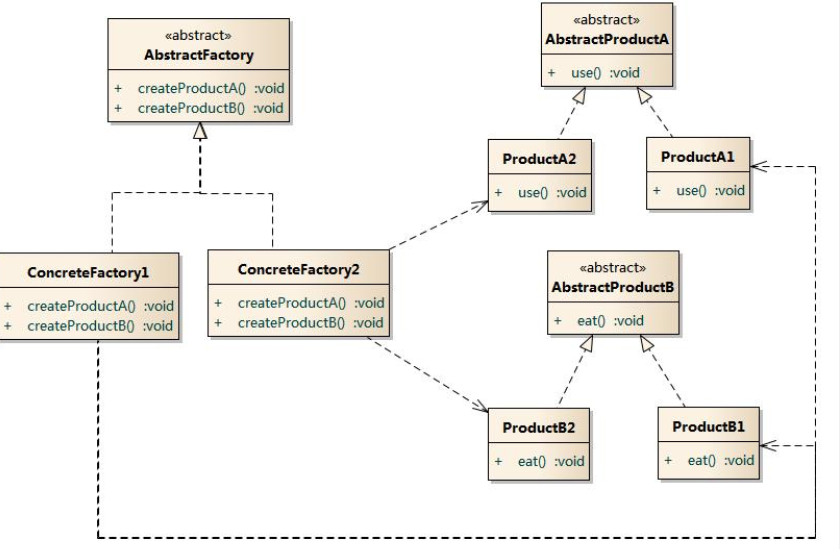
简单工厂，只有一个工厂类，其中用if/else直接返回Product的子类



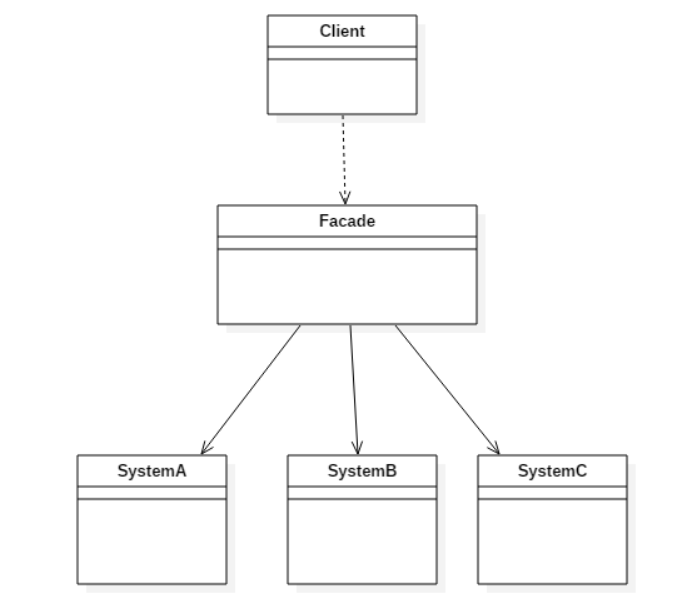
工厂方法模式，每个Product子类都有一个与之对应的工厂（也是一对多），最常用（1.体现了开闭原则：即新增一个类不用修改任何代码，直接继承 2.把new一个类的复杂过程封装在了工厂里，用户不必知道）



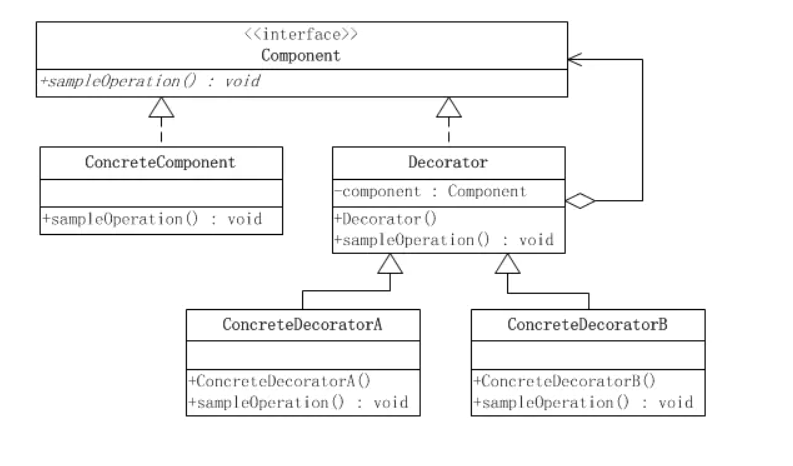
抽象工厂模式：需要一组对象共同完成某种功能时，并且可能存在多组对象完成不同功能的情况。比如Factory2生成ProductA2与ProductB1，而Factory1生成ProductA1与ProductB2。抽象工厂和工厂方法不同地方在于，抽象工厂不只生产一种产品。



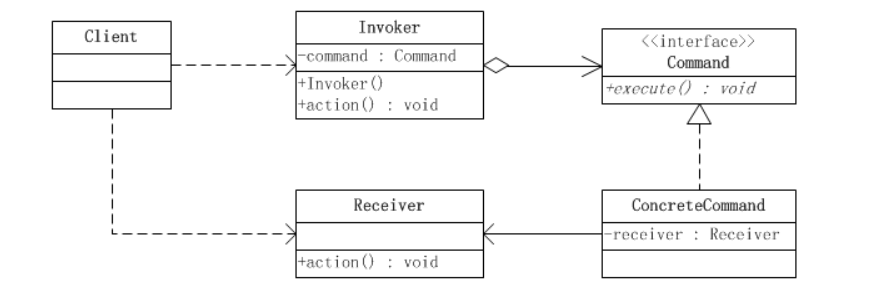
1. 外观模式：提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层的接口，让子系统更容易使用。其实就是为了方便客户的使用，把一群操作，封装成一个方法。如电脑只有启动，关闭方法，但是启动方法却调用了硬盘，CPU，内存的启动初始化方法。用户只需要知道电脑的方法，无需知道CPU等的方法。 Façade与系统A/B/C是关联关系。



1. 装饰者模式：当我们设计好了一个类，我们需要给这个类添加一些辅助的功能，并且不希望改变这个类的代码，这时候就是装饰者模式大展雄威的时候了。体现了开闭原则。 Decorator类拥有Component接口，decorator类的sample方法调用了传入的ConcreteComponent的sample方法，而ConcreteDecoratorA通过调用super.sample方法来间接调用ConcreteComponent的sample方法，从而在此方法周围加上一些额外功能。

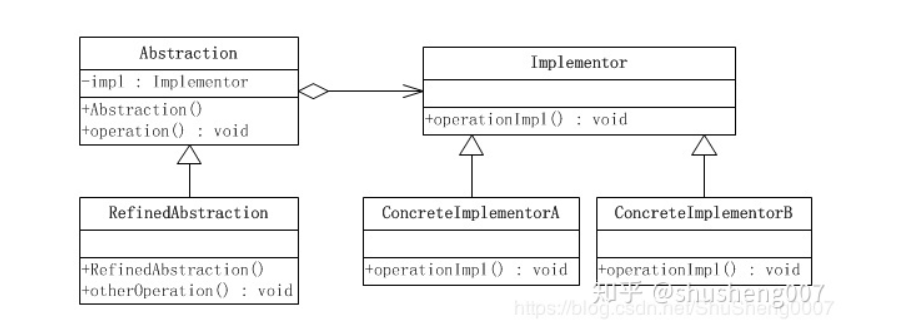


1. 命令模式：命令模式把一个请求或者操作封装到一个对象中。每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方接收到请求，并执行操作。命令模式使得请求的一方和接收的一方独立开来。 Invoker拥有Command，Command拥有Receiver，调用Invoker的action方法，就是调用了Receiver的action方法。创建时，要把接收者对象传入命令对象，把命令对象传入调用者对象，最后要执行什么方法直接执行调用者就可以。



1. 桥接模式

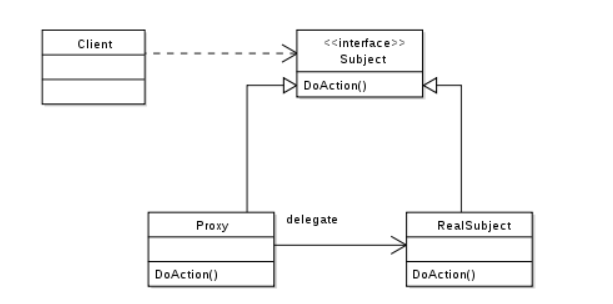
类图（Abstraction是抽象类，不能为接口，因为他拥有接口），客户端通过new具体抽象类，把具体接口传进此具体抽象类，在调用具体抽象类的方法来实现不同的具体抽象类与不同的接口的搭配。



例子两个维度（笔与颜色），笔拥有颜色，把颜色（setColor方法）传入Pen，可以实现不同的笔有不同的颜色。



1. 代理模式，用Proxy对象调用真实对象的方法，并且还可以在方法前后增加一些功能。静态代理，Proxy类拥有RealSubject类。动态代理中，Proxy类变为重写Proxy.newProxyInstance静态方法，不用继承接口，但还是拥有RealSubject类。



Cglib代理，静态/动态代理都要求目标对象实现一个接口或者多个接口，但是有时候目标对象只是一个单独的对象,**并没有实现任何的接口**,这个时候就可以使用构建目标对象子类的方式实现代理（所以目标对象不能是final类）,这种方法就叫做:Cglib代理（继承MethodInterceptor类，重写intercept方法，把需要代理的类传入进去），spring-core中包含了Cglib的jar包，在spring aop中，如果目标对象有接口，用动态代理，没有则用Cglib

代理模式与装饰模式区别：装饰：能动态的新增或组合对象的行为。 代理：为其他对象提供一种代理以控制对这个对象的访问。

## Spring与mybatis

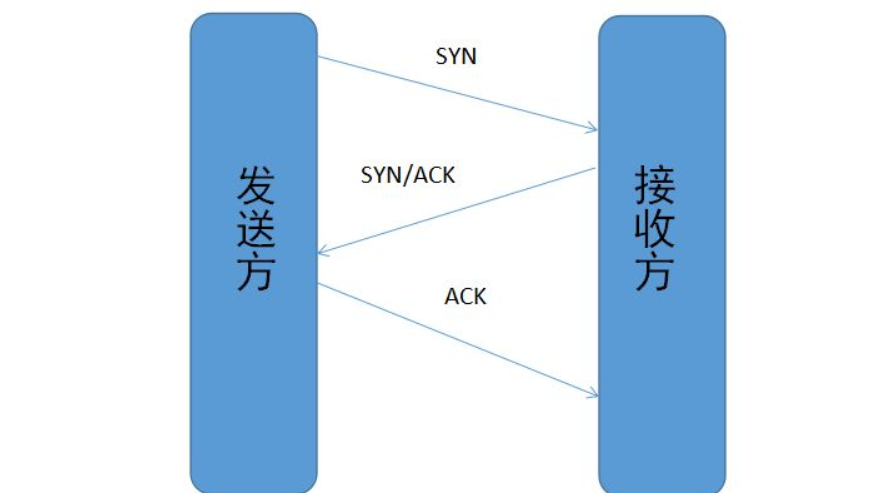
1. Ioc（Inversion of Control）与aop（Aspect Oriented Programming）：（是，内部，用处）1.ioc控制反转，是一种**设计思想**，就是将原本在程序中手动创建对象的控制权交由spring框架来管理。**Ioc容器**是spring用来实现 IoC 的载体， IoC 容器实际上就是个**Map**（key，value）,Map 中存放的是各种对象。Ioc可以很大程度上简化应用的开发，把应用从复杂的**依赖关系中解放出来**。当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。 2.aop：面向切面编程，能够将那些**与业务无关**，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）**封装起来，便于减少系统的重复代码**，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。Spring aop是基于动态代理的。3. Spring aop与AspectJ区别：Spring AOP 属于**运行**时增强，而 AspectJ 是**编译**时增强。 Spring AOP **基于代理(Proxying)**，而 AspectJ 基于**字节码操作**，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。**AspectJ 相比于 Spring AOP 功能更加强大**，但是 Spring AOP 相对来说更简单，当切面太多的话，最好选择 AspectJ ，它比Spring AOP 快很多。
2. @Controller与@RestController：1.单独使用@Controller的话一般使用在要返回一个**视图**的情况，这种情况属于比较传统的Spring MVC 的应用，对应于前后端不分离的情况。2.@RestController返回**json或xml数据**，@RestController**只返回对象**，对象数据直接以 **JSON 或 XML 形式**写入 HTTP 响应(Response)中，这种情况属于 RESTful Web服务，这也是目前日常开发所接触的最常用的情况（前后端分离），同时时用**@Controller+@ResponseBody**也可以实现@RestController
3. Spring中bean的作用域：**singleton**：唯一 bean 实例，Spring 中的 bean 默认都是单例的。 **prototype** : 每次请求（从context中获取）都会创建一个新的 bean 实例。 **request** : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。 **session** :同一个HTTP session共享一个bean，该bean仅在当前 HTTP session 内有效。
4. 将一个类声明为spring的bean有哪些注解：@Component：通用的注解，可标注任意类为 Spring 组件。如果一个Bean不知道属于哪个层，可以使用@Component 注解标注 @Repository : 对应持久层即 Dao 层，主要用于数据库相关操作 @Service : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。 @Controller : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。
5. Spring mvc工作流程：1.客户端（浏·览器）发送请求，直接请求到 DispatcherServlet 2. DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler 3. 解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理（HandlerAdapter作用就是调用具体的方法对用户发来的请求来进行处理） 4. HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑 5. 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View 6. ViewResolver 会根据逻辑 View 查找实际的 View 7. DispaterServlet 把返回的 Model 传给 View（视图渲染） 8.把 View 返回给请求者（浏览器）
6. 什么是spring boot starter：是一系列**依赖关系的集合**，因为它的存在，项目的依赖之间的关系对我们来说变的更加简单了。比如没有spring boot之前，开发web应用需要使用像Spring MVC，Tomcat和Jackson这样的库，这些依赖我们需要手动一个一个添加，但是，有了 Spring Boot Starters 我们只需要一个只需添加一个spring-boot-starter-web一个依赖就可以了。（jar包就是starter）
7. Spring boot嵌入式容器：tomcat，jetty，undertow。如何使用jetty而不是默认的tomcat：在maven中从starter-web中排除tomcat，然后添加jetty的依赖就可以了。
8. SpringBootApplication注解：@SpringBootApplication可以看作是@Configuration（允许在Context中注册额外的bean或导入其他配置类），@EnableAutoConfiguration（启用 SpringBoot 的自动配置机制）、@ComponentScan（扫描被@Component (@Service,@Controller)注解的bean，注解默认会扫描该类所在的包下所有的类） 注解的集合。
9. Cookie与session的使用：1.由于HTTP协议是无状态的协议，**所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session**。典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书（购物车中的书都放在此用户的session中），**当用户第一次打开浏览器访问服务器的时候会创建session，默认有效期30分钟** 2. 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，**需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了**。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。（**cookie也可以用来保存用户的一些信息，如账号密码，网站设置等**），cookie默认有效时间为浏览器窗口关闭。
10. Cookie与Session的区别：1.cookie**只能存储阿斯克码**，假如需求存取Unicode字符或者二进制数据，需要**先进行编码**。Cookie中也不能直接存取Java对象。若要存储略微复杂的信息，运用Cookie是比较艰难的。 Session中能够存取**任何类型的数据**，包括而不限于String、Integer、List、Map等。Session中也能够直接保管Java Bean乃至任何Java对象等，运用起来十分便当 2.cookie存储在**客户端**，对客户端是可见的。 而Session存储在**服务器**上，对客户端是透明的，不存在**敏感信息泄露的风险**。所以cookie上的信息尽量都进行加密。 3.cookie不会占用**服务器资源**，session会占用。
11. Spring boot的自动配置是如何实现的：
12. Spring中自定义注解：（<https://juejin.im/post/5cf376e16fb9a07eee5eb6eb>） 字段注解：通过@interface声明一个注解，@Target定义注解的使用位置（ElementType.FIELD，表示此注解只能被声明在一个类的方法前），@Constraint通过使用validatedBy来指定与注解关联的验证器，@Retention用来说明该注解类的生命周期（如RetentionPolicy.RUNTIME: 注解保留在程序运行期间，此时可以通过反射获得定义在某个类上的所有注解）。 类注解：也是@interface
13. Spring中单例bean的线程安全：在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中。
14. Spring中循环依赖：导致程序一直循环调用，直至内存溢出。1.构造器的循环依赖：直接抛出异常 2.单例模式下的setter循环依赖：通过三级缓存处理 3.非单例循环依赖：无法处理。 构造器的循环依赖：bean创建过程中会把一个标识符记录在缓存中，创建完毕后会把标识符清除，如果bean在创建过程中发现自己已经在缓存中，则抛出异常。 单例模式下的setter循环依赖：
15. Spring注入方式：构造器注入，字段注入，方法参数注入（都是针对@Autowired注解）
16. Springboot配置文件两种：.yml与.property
17. Mybatis中$/#的区别：**$是静态文本直接替换，#是参数占位符**，mybatis会将#{}替换成问号，在 sql 执行前会使用 PreparedStatement 的参数设置方法，按序给 sql 的?号占位符设置参数值。 Mybatis传递参数用@Param注解
18. Mybatis的动态sql：以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。Foreach/if/choose/when等
19. Springboot中mybatis集成分页插件（pagehelper）：首先在pom中引入，然后修改配置文件，再执行PageHelper.startPage(pageNo,pageSize);就可以了。

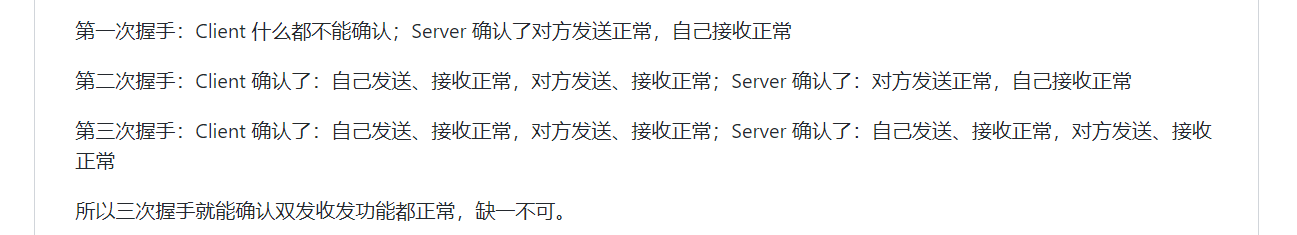
## 操作系统

1. 进程通信方式：1.匿名管道（一个内核缓冲区，单工，只能用于父子或兄弟进程） 2.有名管道（只要由路径就可以访问，存放在内存中） 3.信号（一个进程发送给另一个进程的消息） 4.消息队列（放在内核中的消息链表） 5.共享内存 6.信号量（PV操作，获取到信号量才能执行） 7.套接字
2. 进程状态：就绪，运行，阻塞。
3. 进程调度的策略：1.先来先服务（FCFS） 2.最短作业优先调度算法（SJF）：作业所需的运行时间越短，先运行，但是难以知道作业的运行时间 3.优先级调度算法（抢占/非抢占）：优先级高的先运行，但是会导致低优先级的会无限阻塞（解决方法：老化） 4.时间片轮转调度算法：每一个进程运行固定的时间片 5.多级队列调度算法：将就绪队列分成多个独立的队列，每个队列都有自己的调度算法，队列之间采用固定优先级抢占调度。其中，一个进程根据自身属性被永久地分配到一个队列中。 6. 多级反馈队列调度算法：与多级队列调度算法相比，其允许进程在队列之间移动：若进程使用过多CPU时间，那么它会被转移到更低的优先级队列；在较低优先级队列等待时间过长的进程会被转移到更高优先级队列，以防止饥饿发生
4. 虚拟地址：虚拟内存是计算机系统内存管理的一种技术。它使得应用程序**认为它拥有连续可用的内存**（一个连续完整的地址空间），而实际上，它通常是被**分隔成多个物理内存碎片**，还有部分暂时**存储在外部磁盘存储器上**，在需要时进行数据交换。与没有使用虚拟内存技术的系统相比，使用这种技术的系统使得大型程序的编写变得更容易，对真正的物理内存（例如RAM）的使用也更有效率。如果这个空间是虚拟的，我们就叫做虚拟地址空间；如果这个空间是真实存在的，我们就叫做物理地址空间。**虚拟地址空间是可以任意的大的，因为是虚拟的**。而物理地址空间是真实存在的，所以是有限的。
5. （https://www.zhihu.com/question/50796850）没有分段与分页时出现的问题：1.地址空间不隔离（程序A可能因为地址打错了而访问程序B的地址） 2.程序运行时地址不确定（第一次使用时地址可能为0x00000010，第二次使用时地址变了） 3.内存使用效率低（进程A需要100M内存，而内存只剩余99M，导致进程A不能运行） 分段（解决问题1，2）：把虚拟地址空间映射到了连续的物理地址空间，并且你写的程序操作的是虚拟地址 分页（解决问题1，2，3）：分页这个技术仍然是一种虚拟地址空间到物理地址空间映射的机制。但是，粒度更加的小了。单位不是整个程序，而是某个“页”，一段虚拟地址空间组成的某一页映射到一段物理地址空间组成的某一页。
6. 内存分配：

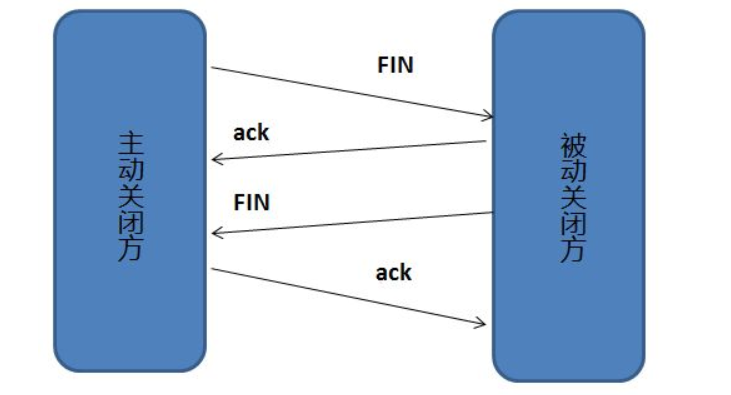
## 网络

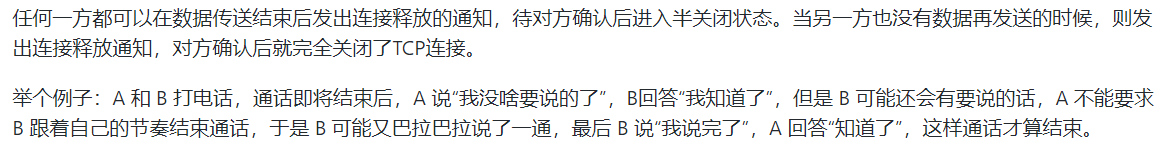
1. TCP三次握手四次挥手的原因：





最初两端TCP进程都处于关闭状态，然后接收方进程创建传输控制块TCP，服务器进程进入**收听**（LISTEN）状态。发送方也是首先创建传输控制块TCP，然后发送SYN后进入**同步已发送状态**（SYN-SENT）。接受方收到SYN并且同意建立连接后发送SYN与ACK并且进入**同步收到**（SYN-RCVD）状态。发送方接受到确认后，还要发送一次ACK，并进入**已建立连接状态**（ESTABLISHED）。接受方收到ACK后也进入已**建立连接状态**。（为什么最后还要发送一次ACK呢/为什么要三次握手？为了防止当发送方发送多次请求建立连接的报文时，某个报文延迟到连接已经断开后才到达，这时接收方发送一个SYN,ACK报文给发送方，而发送方并不会返回任何信息给接收方（因为它的传输任务已经在上一次连接中完成了），如果没有最后一次的ACK，连接建立后，接收方会一直等待发送方发送报文，白白浪费资源，而加入了最后的ACK确认机制，当过一段时间没有接收到发送方最后的ACK时，接收方就会关闭，而不会一直等待。）A（SYN-SENT,ESTABLISHED）B（LISTEN，SYN-RCVD,ESTABLISHED）





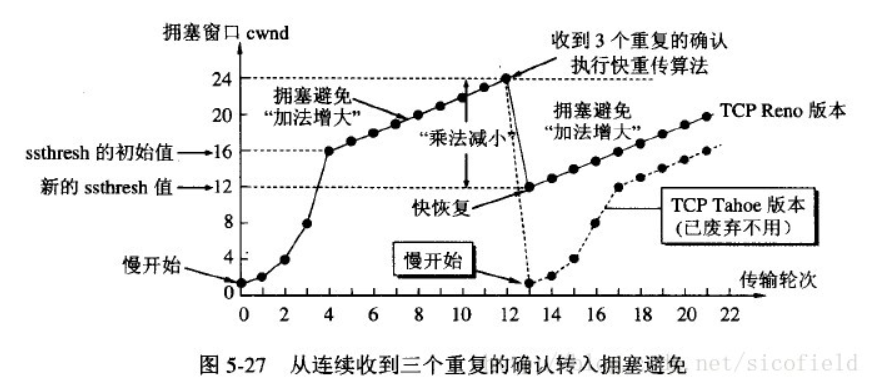
A发送FIN后进入终止等待1状态（FIN-WAIT-1），B接受FIN并发送ACK后进入等待关闭状态（CLOSE-WAIT），这时A已经没有数据要发送了，但若B发送数据，A仍要接受。A收到ACK后进入终止等待2状态（FIN-WAIT-2）。若B已经没有要发送的数据了，则向A发送FIN，进入最后确认状态（LAST-ACK）。A发送ACK后进入时间等待（TIME-WAIT）状态，现在TCP连接还没有关闭，必须等待一定的时间（默认4分钟）A才进入CLOSED状态。B接受到ACK后立刻关闭。A（FIN-WAIT-1,FIN-WAIT-2,TIME-WAIT）B（CLOSE-WAIT,LAST-ACK）。为什么最后A要等待四分钟呢？（1.防止ACK报文丢失，当B长时间没有接受ACK报文时，会重新发送FIN+ACK报文给A，A则重新发送ACK并重新等待4分钟。）

1. 网络协议有哪些层？物理层（引脚标准等），数据链路层（PPP），网络层（IP，ARP），运输层（TCP,UDP），应用层（DNS,HTTP,STMP等）。物理层：在物理层上所传送的数据单位是比特。 物理层(physical layer)的作用是实现相邻计算机节点之间比特流的透明传送，**尽可能屏蔽掉具体传输介质和物理设备的差异** 数据链路层：数据链路层将网络层交下来的 IP 数据报**组装成帧**，在两个相邻节点间的链路上传送帧。每一帧包括数据和必要的控制信息（如同步信息，地址信息，差错控制等） 网络层：进行通信的两个计算机之间可能会经过很多个数据链路，也可能还要经过很多通信子网。网络层的任务就是**选择合适的网间路由和交换结点**， 确保数据及时传送。 运输层：就是跨主机两个进程之间通信，比如用tcp之类的，端口端口之间搭建通道，就是传输层解决的。**让两个进程可以实现通信。屏蔽了网络，物理等底层**。 应用层：应用层协议定义的是应用进程（进程：主机中正在运行的程序）间的通信和交互的规则
2. TCP与UDP的区别：1.UDP无连接 2.UDP不保证可靠交付 3.UDP面向报文，TCP面向字节流 4.UDP没有拥塞控制 5.UDP支持1对1，1对多，多对1，多对多，而TCP只能1对1 6.UDP首部开销小（只有8字节）。
3. **http状态码和每个状态码含义**：（https://juejin.im/post/590082e6a22b9d0065be1a5c#comment）1xx：代表请求已被接受，需要继续处理。这类响应是**临时响应**，只包含状态行和某些可选的响应头信息，并以空行结束。（102：处理将被继续执行） 2xx：**成功**，代表请求已成功被服务器接收、理解、并接受（200：成功，201：已创建，202：已接收但未处理） 3xx：**重定向**，代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的Location域中指明。（301表示请求的网页已经被永久移动到新位置 **304表示请求的网页未更改，可以直接用客户端缓存**） 4xx：**客户端错误**（如400表示服务器不理解请求语法，404表示服务器找不到请求的网页） 5xx：**服务器错误**（503表示服务暂时无法使用）
4. 输入url回车后发送了什么：<https://zhuanlan.zhihu.com/p/80551769> 1.url解析（判断你输入的是一个合法的 URL 还是一个待搜索的关键词） 2.dns查询（查询网址的ip） 3.tcp连接（客户端发起tcp连接） 4.处理请求（服务器处理请求） 5.接受响应 6.渲染页面
5. URI和URL的区别：URI（统一资源标识符）**包括URL**（统一资源定位符）和URN，URI用来**唯一的标识**一个资源，URL是一种具体的URI，可以用来**标识**一个资源，而且还指明了如何**定位**这个资源。URI一般包括：访问资源的命名机制，存放资源的主机名，资源自身的名称，由路径表示，着重强调于资源。 URL一般包括：协议，存有该资源的主机IP地址(有时也包括端口号)，主机资源的具体地址。如目录和文件名等。
6. Get与post的区别：1.get相当于select，post相当于insert，update，delete，get请求的数据放在url中可以被看到。



Get请求不走缓存：1.在get请求后面拼接随机数或者时间戳 2.服务器端设置header(“Cache-Control: no-cache, must-revalidate”)

1. 网络7层模型：物理层，数据链路层，网络层，运输层，会话层，表示层，应用层。 多出了会话层（session，它定义了如何开始、控制和结束一个会话，包括对多个双向消息的控制和管理，以便在只完成连续消息的一部分时可以通知应用，从而使表示层看到的数据是连续的）与表示层（这一层的主要功能是定义数据格式和加密，例如FTP（文件传输协议）允许使用二进制或者ASCII传输）。
2. 路由器工作过程：收到数据包后，解封以太网帧头部，提取目的MAC地址，查看目的MAC地址是不是自己本身的MAC地址，如果不是自己的MAC地址则丢弃，如果是自己的MAC地址，上传到上层解析IP层，并判断ip地址是否是本机，如果是本机则上传到上一层，如果不是则查询路由表，匹配出接口，然后根据下一跳ip地址到ARP表中查询下一跳的MAC地址，如果没查到就发送ARP请求查找MAC地址，最后把MAC地址封装，发送。
3. 路由表：1.目标网络地址和网络掩码。发送来的IP数据报中的IP地址与网络掩码想与等于网络地址，则从此接口发送出去。（一般包括默认路由，即没在表中的网络地址都走默认路由） 2.网关，即到达目标网络的下一跳服务器地址。 3.接口，到达目标网络，本地计算机用于发送数据包的接口。
4. Arp协议：首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址之间的对应关系。当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：源主机 IP地址，源主机MAC地址，目的主机的IP 地址。当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。
5. 拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不过载。 慢开始：如果立刻把大量数据字节注入到网络，那么可能会引起网络阻塞，较好的方法是从小到达逐渐增大发送窗口（拥塞窗口以字节为单位），cwnd初始为1，每经过一次传播轮次，cwnd加倍。一旦出现超时重传，TCP就把ssthresh的值设置为当前cwnd的一半，同时当前cwnd重置为1。 拥塞避免：让拥塞窗口cwnd缓慢增大，每经过一个往返时间就把cwnd+1。 快重传与快恢复：如果接受方收到一个乱序（1，3就是乱序）的报文段就立刻会送一个ACK给发送方，如果发送方一连收到3个重复的确认，就应该立刻发送接受方未收到的报文。发送方认为现在的网络可能没有出现拥塞， 那么将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。



1. Ping命令协议：icmp，网络层

## Linux

1. 常用命令

## 算法

1. 常考手撕算法：bfs等树的操作