基础题目

1. Java线程的状态

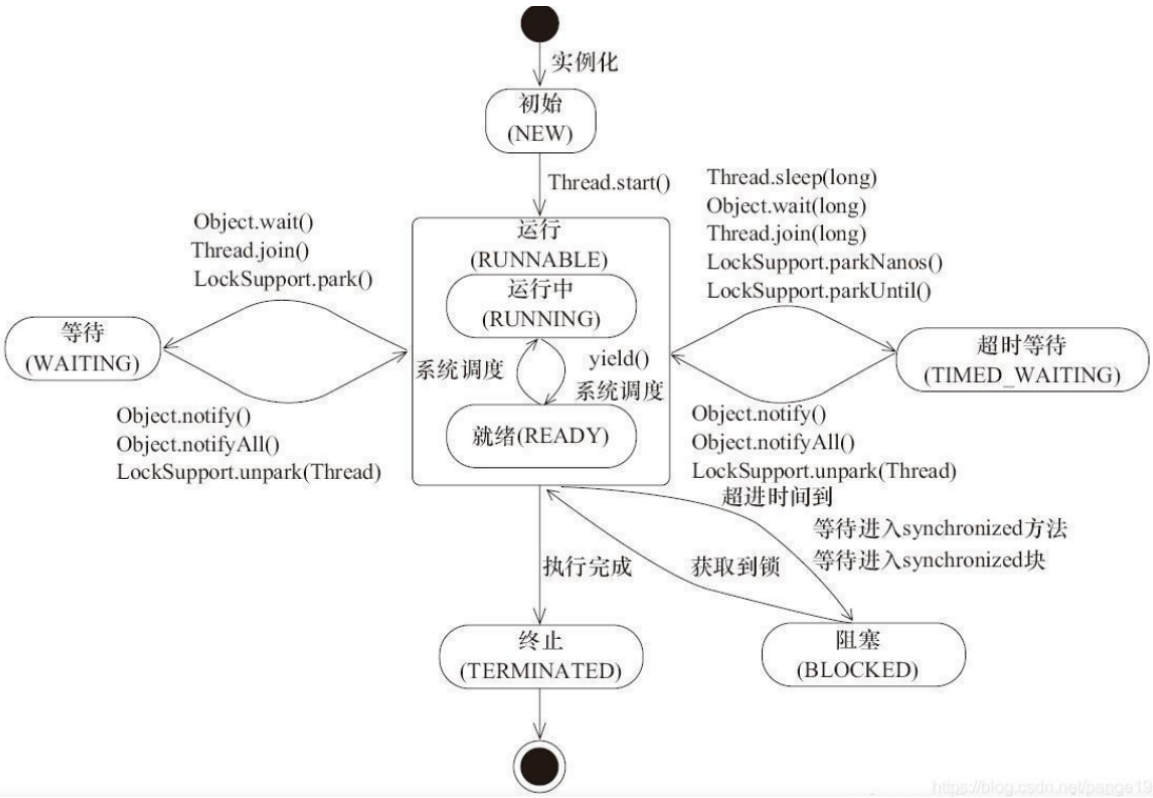
- **初始New：**新创建了一个线程对象
- **运行Runnable**（就绪ready、运行中running）：位于可运行线程池中，等待被线程调度选中，获取CPU的使用权
    - 新建线程，调用线程的start()方法，此线程进入就绪状态（**new->ready**）
    - 当前线程时间片用完了，调用当前线程的yield()方法，当前线程强制进入就绪状态（**running->ready**）
    - 当前线程sleep()方法结束，其他线程join()结束（**timed_waiting->ready**）
    - 等待用户输入完毕（**waiting->ready**）
    - 某个线程拿到对象锁（**blocking>ready**）

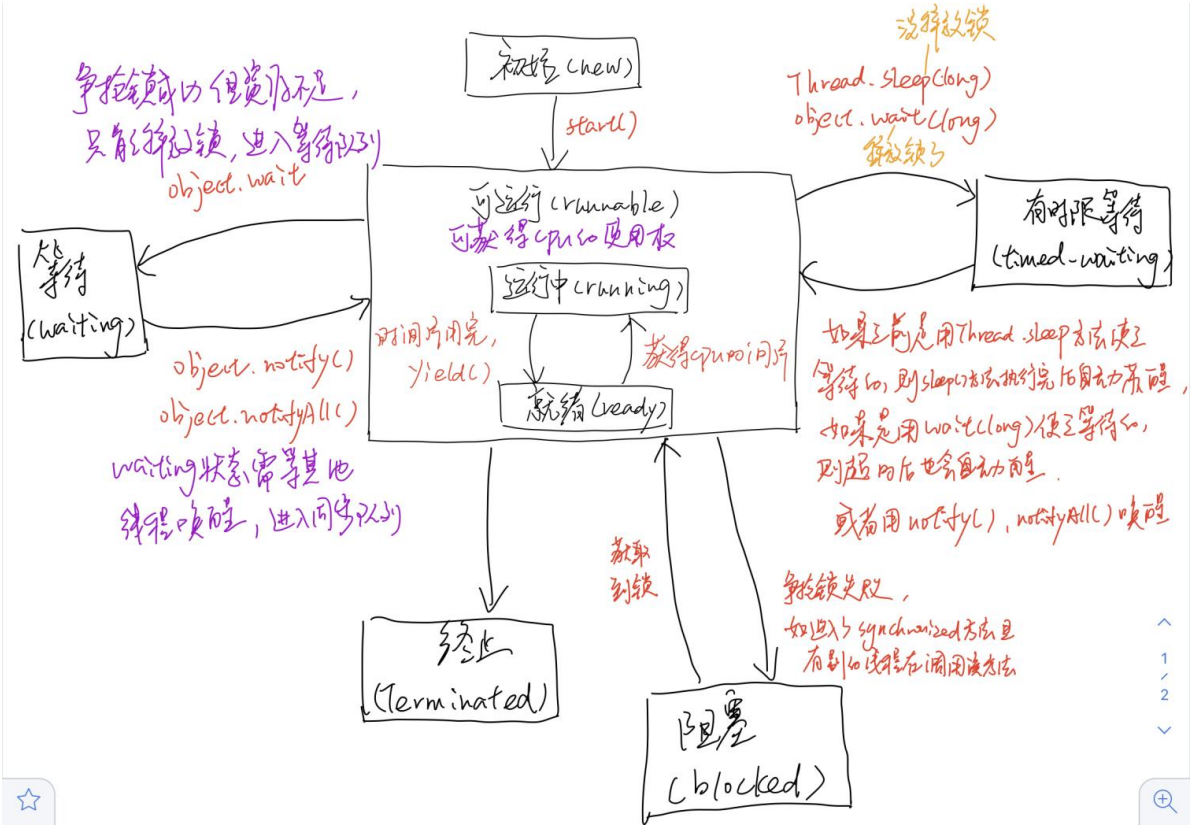
  在获得CPU时间片后变为运行中状态（running）
- **阻塞Blocking：**线程阻塞于同步锁
- **等待Watiing：**调用某个对象的wait()方法，释放锁，进入等待队列，不会被分配CPU执行时间，它们要等待被显式地唤醒，否则会处于无限期等待的状态
- **超时Timed-waiting：**该状态不同于WAITING，它可以在指定的时间后自行返回
- **终止Terminated：**表示该线程已经执行完毕，run()方法完成时，或者主线程的main()方法完成时

同步队列：同步队列里面放的都是想争夺对象锁的线程，Blocking队列

等待队列：等待唤醒被notify或notifyAll唤醒，Waiting队列

！！！！下面是yyj整理的！！！





初始：这个线程对象刚刚被new

可运行：这个线程对象位于可运行线程池中，等待被线程调度选中、获得CPU的使用权。新建线程调用Thread.start()方法就进入可运行状态

n 分为运行中和就绪两种子状态，当获得CPU时间片时为运行中，时间片用完就调用该线程的yield()方法变为就绪

阻塞：可运行状态争抢锁失败，如等待进入synchronized代码块或被lock，进入阻塞状态，当获取到锁时回到可运行状态

等待：可运行阶段争抢锁成功但资源不够，调用该线程对象的wait()方法，进入等待队列，要被其他线程调用notify()或notifyAll()的时候才被唤醒，转为可运行状态

有时限等待（超时等待)：也是可运行阶段争抢锁成功但资源不够，调用该线程对象的wait(long)方法，或者调用静态方法Thread.sleep(long)。

【wait和sleep的区别：

1. wait()释放了锁，sleep()没释放
2. sleep()方法执行完成后，线程会自动苏醒，wait(long timeout) 超时后线程会自动苏醒，普通的wait()一定要notify()或notifyAll()唤醒
3. sleep() 是 Thread 类的静态本地方法，wait() 则是 Object 类的本地方法】

所以若使线程超时等待的是sleep，那sleep方法执行完后线程自动苏醒回到可执行状态；若使线程超时等待的是wait(long)，那超时后线程自动苏醒回到可执行状态。

终止：线程已执行完毕或主线程的main()方法完成。

2. 进程和线程的区别，进程间如何通讯，线程间如何通讯

进程是系统进行资源分配和调度的一个独立单位；线程是进程的一个实体，是CPU调度和分派的基本单位，基本上不拥有系统资源。

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。**这就是进程和线程的重要区别。**

进程间通讯：信号Sinal ，套接字Socket，信号量Semaphore，共享存储SharedMemory，消息队列MessageQueue，管道pipe

- 管道：具有固定的读端和写端，只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间），通过内核缓冲区实现数据传输
- 消息队列：一个消息的链表，是一系列保存在内核中消息的列表，进程可以根据自定义条件接收特定类型的消息
- 共享内存：允许两个或多个进程共享一个给定的存储区，这一段存储区可以被两个或两个以上的进程映射至自身的地址空间中，一个进程写入共享内存的信息，可以被其他进程共享

  共享内存中的内容往往是在解除映射时才写回文件，因此，采用共享内存的通信方式效率非常高
- 信号量：常作为一种锁机制，用于实现进程间的互斥与同步，而不是用于存储进程间通信数据

线程间通讯：线程通信就是当多个线程共同操作共享的资源时，互相告知自己的状态以避免资源争夺

- 共享内存：多个线程访问内存中的同一个被volatile关键字修饰的变量时，当某一个线程修改完该变量后，需要先将这个最新修改的值写回到主内存，从而保证下一个读取该变量的线程取得的就是主内存中该数据的最新值
- 消息传递：线程间的通信需要对象Object来完成，Object调用wait()、notify()、notifyAll()方法如同开关信号，完成等待方和通知方的交互。注意，从wait()方法返回的前提是获得调用对象的锁

  join()方法是主线程要等待子线程结束，当前线程A调用线程B的join()方法后，会让当前线程A阻塞，直到线程B的逻辑执行完成，A线程才会解除阻塞，然后继续执行自己的业务逻辑
- 管道流：管道输入/输出流的形式，主要用于线程之间的数据传输，传输的媒介为管道

3. HashMap的数据结构是什么？如何实现的？和HashTable，ConcurrentHashMap的区别

- HashMap：底层数组+链表实现，主干是一个Entry数组，每一个Entry包含一个key-value键值对（所谓Map），链表则是主要为了解决哈希冲突而存在的。初始size为16，可以存储null键和null值，线程不安全。当链表冲突元素大于8时发生“树化”变为红黑树，以提高查找效率；低于6时再次“链化”。

  计算数组下标的过程为，先根据key的值计算到一个hashCode，将hashCode的高16位二进制和低16位二进制进行异或运算，得到的结果再与当前数组长度减一进行与运算

  JDK1.8和JDK1.8对链表的头/尾插入和数组效标的运算要求不一样，
- HashTable：数组+链表实现，无论key还是value都不能为null，线程安全，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化。初始size为11
- ConcurrentHashMap：采用分段的数组+链表实现，线程安全，通过把整个Map分为N个Segment，使用锁分离技术，使得多个修改操作并发进行（Hashtable通过锁住整张表保证线程安全），提高了效率

在HashMap中不能由get()方法来判断HashMap中是否存在某个key，应该用containsKey()方法来判断。而在Hashtable中，无论是key还是value都不能为null。

4. Cookie和Session的区别

Cookie和Session的介绍：

- **Cookie：**HTTP/1.1 引入 Cookie 来保存状态信息。Cookie 是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器之后向同一服务器再次发起请求时被携带上，用于告知服务端两个请求是否来自同一浏览器。

  由于之后每次请求都会需要携带 Cookie 数据，因此会带来额外的性能开销。但现在逐渐不常用，使用本地缓存

  过程：客户端发送请求后，服务器发送的响应报文包含 Set-Cookie 首部字段，客户端得到响应报文后把 Cookie 内容保存到浏览器中；

  客户端之后对同一个服务器发送请求时，会从浏览器中取出 Cookie 信息并通过 Cookie 请求首部字段发送给服务器

- **Session：**除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

  Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

  过程：客户端发送请求后，服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；

  客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作

区别：

- cookie数据保存在客户端，session数据保存在服务器端；
- cookie不是很安全，考虑到安全应当使用session；

  session会在一定时间内保存在服务器上，考虑到减轻服务器性能方面，应当使用cookie；
- cookie大小有限制，session没有；cookie是服务器存储在本地计算机上的小块文本；
- 一旦客户端禁用Cookie，Session也会失效

5. 索引有什么用？如何建索引?

索引是一种单独的、物理的对数据库表中一列或多列的值进行排序的一种存储结构；

索引是帮助MySQL高效获取数据的排好序的数据结构，本质是一种优化查询的数据结构，目的就是为了减少磁盘I/O的次数 ，加快查询速率

索引的结构有平衡二叉树AVL、B树、B+树、红黑树

6. ArrayList是如何实现的，ArrayList和LinkedList的区别？ArrayList如何实现扩容?

ArrayList底层维护的是一个Object数组，元素都存放在这个Object数组中，数据的默认大小是10，超过大小是按1.5倍进行扩容，扩容的时候需要将原来的数据复制到新的数组，而老的数组进行回收，非常费时间。

两个都是List的子类：

- ArrayList是数组结构，而LinkedList是链表结构
- LinkedList维护了两个指针，头和尾，所以Linkedlist在插入的时候比较快
- ArrayList在插入的时候没有LinkedList快，但是ArraList通过下标访问，访问速度比较快

7. equals方法实现

基本数据类型：比较应该使用（==），比较值；

复合数据类型：

- “==”：判断是否指向同一内存空间，比较内存地址
- “equals”：内容是否相同，比较内容

程序在运行的时候会创建一个字符串缓冲池，当使用String s2 = "Monday" （已有String s1 = "Monday"）这样的表达是创建字符串的时候，程序首先会在这个String缓冲池中寻找相同值的对象，如果找到了，就直接引用，而不创建，此时“==”成立，equals成立；

如果是String s2= new String("Monday")，那就是要一个新的，一个新的对象将被创建在内存中，就不“==”了，但是还是equals；

要同时实现equals()和hashCode()两个方法，首先要确定好两个对象等价的条件是什么，Java为每一种类型都都默认实现了该类型的hashCode()方法

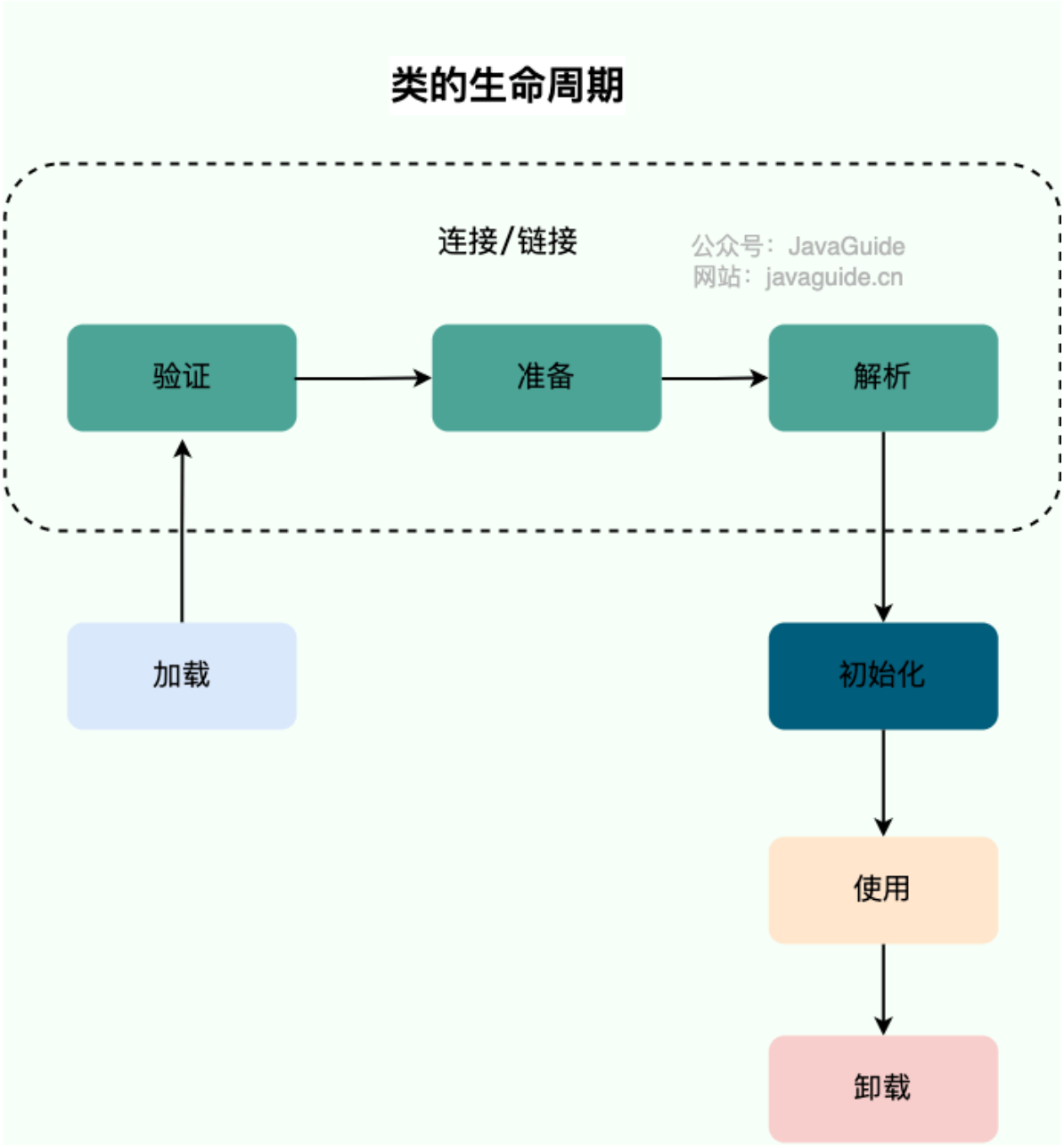
8. 面向对象

9. 线程状态，BLOCKED和WAITING有什么区别

- BLOCKED：无法获取到与同步方法/代码块相关联的锁，与之相关的是同步队列
- WAITING：线程可以通过`wait,join,LockSupport.park`方式进入`wating`状态，与之相关联的是等待队列。进入wating状态的线程等待唤醒(notify或notifyAll)才有机会获取cpu的时间片段来继续执行
- `blocked`状态是处于`wating`状态的线程重新焕发生命力的必由之路

10. JVM如何加载及卸载字节码文件

类的生命周期：加载→连接（包括验证、准备、解析）→初始化→使用→卸载



类的加载过程：

1. 加载（通过类加载器完成，具体用哪个类加载器由双亲委派模型决定，在方法区中生成对应的结构）

- 通过全类名获取定义此类的二进制字节流（可以通过jar、zip、动态代理技术运行时生成等.....）
- 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
- 在内存中生成一个代表该类的 `Class` 对象，作为方法区这些数据的访问入口

2.验证（目的是确保class文件中的字节流包含的信息符合《java虚拟机规范》的要求）

- class文件格式验证
- 字节码语义检查
- 程序语义检查
- 类的正确性检查

(第一个直接检查class文件，主要为了保证输入的字节流能正确地解析并存储于方法区，后三个阶段检查的是方法区里的存储结构)

3.准备（正式为类变量分配内存并设置类变量初始值）

需要注意的点：

- 现在只分配类的静态变量，也就是用static关键字修饰的变量，不分配实例变量
- 类的静态变量的引用放在方法区，对象在JDK8之后放到堆【JDK8之后原来在方法区的字符串常量池和静态变量对象都放到了堆中】
- "初始值"是指数据类型的默认值而不是用户赋予的初始值，比如我们定义了`public static int value=111`，那么 value 变量在准备阶段的初始值就是 0 而不是 111（初始化阶段才会赋值）。特殊情况：比如给 value 变量加上了 final 关键字`public static final int value=111`，那么准备阶段 value 的值就被赋值为 111。

4.解析（将常量池内的符号引用替换为直接引用）

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符 7 类符号引用进行。

符号引用：

5.初始化（执行初始化方法`<clinit>()`，开始真正执行类中定义的 Java 程序代码）

只有这6种情况，主动去使用类了，才会去初始化类：

1. 遇到`new`、`getstatic`、`putstatic`或`invokestatic`这 4 条直接码指令时
2. 使用`java.lang.reflect`包的方法对类进行反射调用时如`Class.forname("...")`等
3. 初始化一个类，如果其父类还未初始化，则先触发该父类的初始化
4. 当虚拟机启动时，用户需要定义一个要执行的主类 (包含`main`方法的那个类)，虚拟机会先初始化这个类
5. `MethodHandle`和`VarHandle`可以看作是轻量级的反射调用机制，而要想使用这 2 个调用， 就必须先使用`findStaticVarHandle`来初始化要调用的类
6. 当一个接口中定义了 JDK8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化

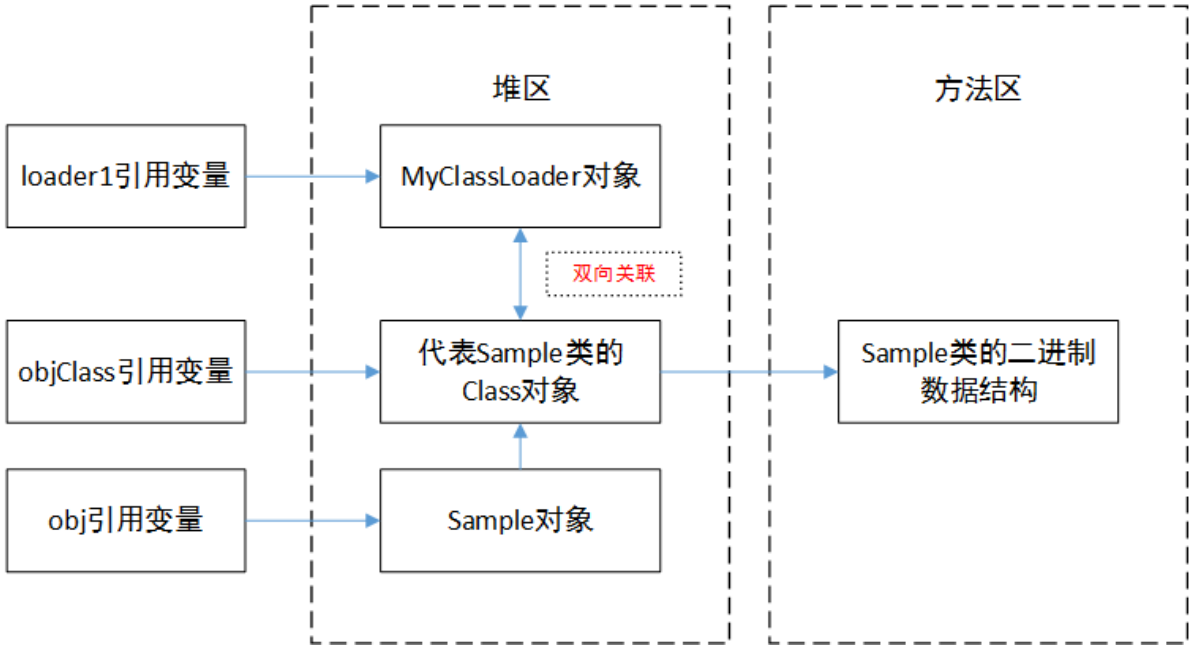
类卸载：

卸载类即该类的class对象被GC。

卸载类需要满足 3 个要求:

1. 该类的所有的实例对象都已被 GC，也就是说堆不存在该类的实例对象。
2. 该类没有在其他任何地方被引用
3. 该类的类加载器的实例已被 GC

所以，在 JVM 生命周期内，由 **jvm 自带的类加载器加载的类**是不会被卸载的。但是**由我们自定义的类加载器加载的类**是可能被卸载的。



loader1变量和obj变量间接应用代表Sample类的Class对象，而objClass变量则直接引用它。

如果程序运行过程中，将上图左侧三个引用变量都置为null，此时Sample对象结束生命周期，MyClassLoader对象结束生命周期，代表Sample类的Class对象也结束生命周期，Sample类在方法区内的二进制数据**被卸载**。

当再次有需要时，会检查Sample类的Class对象是否存在，**如果存在会直接使用，不再重新加载**；如果不存在Sample类会被重新加载，在Java虚拟机的堆区会生成一个新的代表Sample类的Class实例(可以通过哈希码查看是否是同一个实例)。

11. JVM GC，GC算法

Java GC：泛指java的垃圾回收机制，该机制是java与C/C++的主要区别之一，我们在日常写java代码的时候，一般都不需要编写内存回收或者垃圾清理的代码，也不需要像C/C++那样做类似delete/free的操作。

主要回收区域是Java堆（线程共享，只有在运行期间才可知道这个方法创建了哪些对象没需要多少内存，所以回收是动态的）。

Java的堆内存基于Generation算法（Generational Collector）划分为新生代、年老代和持久代。新生代又被进一步划分为Eden和Survivor区，最后Survivor由FromSpace（Survivor0）和ToSpace（Survivor1）组成。

年轻代

几乎所有新生成的对象首先都是放在年轻代的。新生代内存按照8:1:1的比例分为一个Eden区和两个Survivor(Survivor0,Survivor1)区。大部分对象在Eden区中生成。当新对象生成，Eden Space申请失败（因为空间不足等），则会发起一次GC(Scavenge GC)。

回收时先将Eden区存活对象复制到一个Survivor0区，然后清空Eden区，当这个Survivor0区也存放满了时，则将Eden区和Survivor0区存活对象复制到另一个Survivor1区，然后清空Eden和这个Survivor0区，此时Survivor0区是空的，然后将Survivor0区和Survivor1区交换，即保持Survivor1区为空， 如此往复。当Survivor1区不足以存放 Eden和Survivor0的存活对象时，就将存活对象直接存放到老年代。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收。新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例。

老年代

在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。一般来说，大对象会被直接分配到老年代。

对象存活判断

判断对象是否存活一般有两种方式：

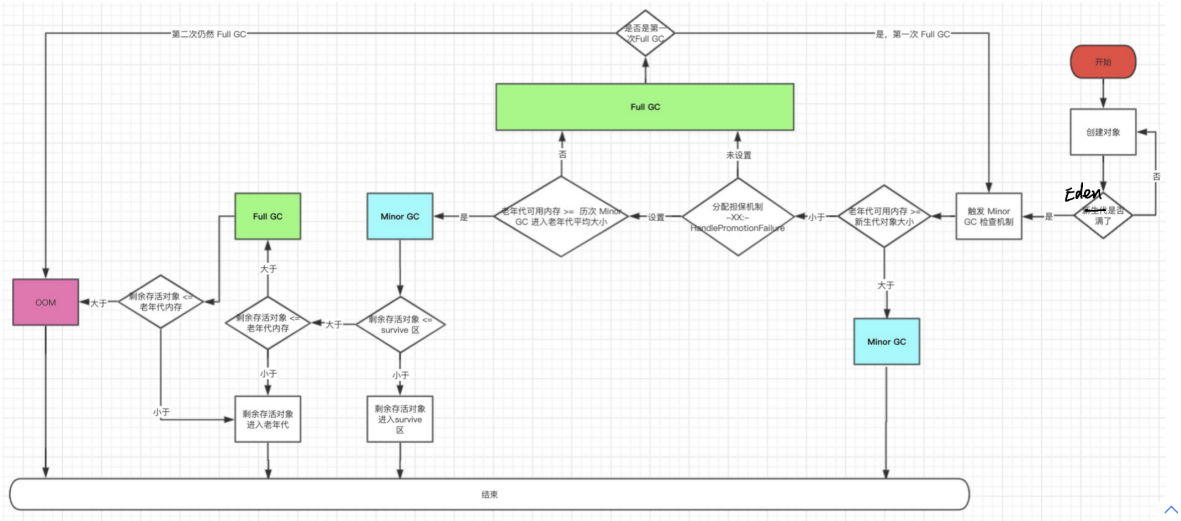
- 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
- 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象

GC算法

- **引用计数算法：**每个对象在创建的时候，就给这个对象绑定一个计数器。每当有一个引用指向该对象时，计数器加一；每当有一个指向它的引用被删除时，计数器减一。这样，当没有引用指向该对象时，该对象死亡，计数器为0，这时对这个对象进行垃圾回收操作
- **标记-清除算法：**为每个对象存储一个标记位，记录对象的状态（活着或是死亡）。分为两个阶段，一个是标记阶段，这个阶段内，为每个对象更新标记位，检查对象是否死亡；第二个阶段是清除阶段，该阶段对死亡的对象进行清除，执行 `GC` 操作

  注：没有移动对象，导致可能出现很多碎片空间无法利用的情况
- **标记-整理算法：**标记-整理法是标记-清除法的一个改进版。同样，在标记阶段，该算法也将所有对象标记为存活和死亡两种状态；不同的是，在第二个阶段，该算法并没有直接对死亡的对象进行清理，而是将所有存活的对象整理一下，放到另一处空间，然后把剩下的所有对象全部清除。这样就达到了标记-整理的目的。
- **复制算法：**该算法将内存平均分成两部分，然后每次只使用其中的一部分，当这部分内存满的时候，将内存中所有存活的对象复制到另一个内存中，然后将之前的内存清空，只使用这部分内存，循环下去
- **分代收集算法**：“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法

！！！！！！！JVM GC的完整过程！！！！！！！！！



创建对象时会优先在eden区分配空间，如果eden区空间不足，触发**young GC（minor GC）**，用来回收eden已有的对象，为新对象创建空出位置。

minor GC之前，要判断老年代可用内存是否够放新生代对象大小（做这个判断的原因是万一survior存不下，survior中的部分数据会来到老年区，这个检查是提前做准备）：

- 够的话直接minor GC就可以了（ GC 完 survivor 不够放，老年代也绝对够放 ）
- 不够的话判断是否通过 -XX:-HandlePromotionFailure 参数开启了允许分配担保失败：
  - 如果未开启，那直接full GC
  - 如果开启了，那根据分配担保规则—— 如果老年代中剩余空间大小≥历次 minor GC 进入老年代的对象的大小，那就允许进行 minor GC,判断老年代中剩余空间大小是否≥历次minor GC进入老年代的对象的大小：
    - 若否，则进行full GC
    - 若是，则进行minor GC

      【需要注意的是，这个规则只是从概率上保证直接进行minor GC能够成功，但也有可能失败，所以后面的minor GC过程中也有可能老年代放不下】

      下面讲述后面minor GC的过程：
      - 如果eden要清除的数据survior放得下，那就放入survior
      - 如果eden要清除的数据survior放不下了，那要移动一部分进入老年代：
        - 如果老年代放得下，那就移动一部分进入老年代
        - 如果老年代放不下，那就full GC
        - 如果full GC之后也放不下了，那只能OOM了

12. 什么情况会出现Full GC，什么情况会出现yong GC

堆内存划分为 Eden、Survivor 和 Tenured/Old 空间

Young GC：在新生代的Eden区域满了之后就会触发，采用复制算法来回收新生代的垃圾

Full GC：从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC(Young GC)，对老年代GC称为Major GC(Old GC)，而Full GC是对整个堆来说的

产生Full GC的情况（主要答前三个）：

- 手动调用System.gc()方法：此方法的调用是建议JVM进行Full GC，非一定，但有很大可能
- 发生Minor GC前，先进行判断，当老年代剩余空间小于新生代所有内存时，判断-XX:-HandlePromotionFailure 是否设置，如果该参数没有设置，或者该参数设置了但老年代最大可用的连续空间小于平均新生晋升大小，则full GC
- 经过前面的判断后，在正常进行minor GC时，由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，若老年代的可用内存小于该对象大小。，那也会full GC
- 老年代空间不足
- 方法区（也就是永久代/元空间）空间不足

13. JVM内存模型

简要言之，jmm是jvm的一种规范，定义了jvm的内存模型。它屏蔽了各种硬件和操作系统的访问差异，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

它不像c那样直接访问硬件内存，相对安全很多，它的主要目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。可以保证并发编程场景中的原子性、可见性和有序性。

Java内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。线程间的通信一般有两种方式进行，一是通过消息传递，二是共享内存。

JMM 属于语言级的内存模型，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。方法是在适当位置会插入内存屏障，happens-before规则、volatile关键字。

14. Java运行时数据区

管理内存有五大区域：方法区、堆、栈、本地方法栈、程序计数器

其中方法区和堆是所有线程共享的，栈，本地方法栈和程序计数器则为线程私有的

- **程序计数器**：线程私有，为了线程切换可以恢复到正确执行位置
- **栈（虚拟机栈）**：线程私有，每个方法被执行的时候都会创建一个栈帧用于存储局部变量表，每一个方法被调用的过程就对应一个栈帧在栈中从入栈到出栈的过程
- **本地方法栈**：与虚拟机栈发挥的作用十分相似，区别是，本地方法栈为虚拟机使用到的native方法服务（调用底层c代码）
- **堆**：堆是java虚拟机管理内存最大的一块内存区域，因为堆存放的对象是线程共享的，所以多线程的时候也需要同步机制。它的目的是存放对象实例，也是GC所管理的主要区域
- **方法区**：同堆一样，是所有线程共享的内存区域。用于存储已被虚拟机加载的类信息、常量、静态变量，如static修饰的变量加载类的时候就被加载到方法区中（运行时常量池也在这里面）

15. 事务的实现原理

事务：事务是数据库提供的一种手段，通过这种手段，应用程序员将一系列的数据库操作组合在一起作为一个整体以供数据库系统提供一组保证

16. 事务的四个关键属性(ACID)

属性	描述
原子性A	事务包含的一组更新操作是原子不可分的，要么全部成功，要么全部失败回滚（如i++包含从主内存读取i、在工作内存把i+1、把新的值写回到主内存这三个步骤）
一致性C	事务执行前后数据库都要处于一致性状态（如银行转账系统A转钱给B前后总钱数一样多）
隔离性I	当多个用户并发访问数据库时，数据库为每个用户开启的事务不能被其他事务干扰，多个并发事务要相互隔离
持久性D	一旦事务提交，所作的修改要永远保存到数据库中，即使系统崩溃也不能丢失

- **原子性**：事务包含的一组更新操作是原子不可分的，要么全部提交成功，要么全部失败回滚（系统在做操作前，会先在磁盘上将操作存储下来），如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响
- **一致性**：指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。一个事务所做的修改在最终提交以前，对其它事务是不可见的。

  举例来说，假设用户A和用户B两者的钱加起来一共是1000，那么不管A和B之间如何转账、转几次账，事务结束后两个用户的钱相加起来应该还得是1000，这就是事务的一致性。
- **隔离性**：隔离性是当多个用户并发访问数据库时，比如同时操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离
- **持久性**：一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失

为什么要有这四个属性呢?

- 只有满足一致性，事务的执行结果才是正确的
- 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性
- 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性
- 事务满足持久化是为了能应对系统崩溃的情况

17. 事务隔离级别

为什么要设置事务隔离级别?

在数据库操作中，在并发的情况下可能出现如下问题：

名称	描述
更新丢失	更新被覆盖
脏读	读到了其他事务未提交的数据
不可重复读	同一事务对同一行读取两次，两次读取到的结果不同，因为另一事务在这期间对这行做了修改。
幻读	（本质上也属于不可重复读，但是指一定范围的多行）同一事务两次读取某一范围的数据时读到不同的结果，因为另一事务在这期间insert了新的行

不可重复读侧重的是数据的修改，幻读侧重的是数据的新增或删除。解决不可重复读，只用锁住行，而解决幻读，需要锁住整张表。

- **更新丢失**（Lost update）：更新被覆盖
- **脏读**（Dirty Reads）：读到另外事务未提交的数据
- **不可重复读**（Non-repeatable Reads）：一个事务对同一行数据重复读取两次，但是却得到了不同的结果（另一事务也访问了该同一数据集合并做了修改）
- **幻影读**：本质上也属于不可重复读的情况，但只是指新insert的行。T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同

所以设置了四个隔离级别，由低到高依次为：

名称	描述	解决的问题	没解决的问题
读取未提交	一个事务读的时候没有限制，写的时候其他事务不能写但可以读，所以这个事务进行的更新操作不会丢失但它未提交的数据会被其他事务读到，	更新丢失	脏读、不可重复读、幻读
读取已提交（大多数数据库默认的）	一个事务读的时候没有限制，写的时候在提交之前别的事务不能读也不能写，从而避免读到未提交的数据，但读操作没有限制，所以两次读取还是可能不同	更新丢失、脏读	不可重复读、幻读
可重复读（mysql默认的）	读时禁止写（只是禁止对现在读取的行进行写，还是可以insert新的行），写时禁止任何，保证同一事务中多次读取的结果一样	更新丢失、脏读、不可重复读	幻读（因为加的锁是行级锁，不是表级锁）
可串行化	所有事务串行执行	所有	

- **未提交读（READ UNCOMMITTED）**：如果一个事务已经开始写数据，则另外一个事务则不允许同时进行写操作，但允许其他事务读此行数据

  避免了更新丢失，却可能出现脏读。因为事务B可能读取到事务A未提交的数据

- **提交读（READ COMMITTED）**：读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行

  避免了脏读，但是却可能出现不可重复读。如事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变

- **可重复读（REPEATABLE READ）**：保证在同一个事务中多次读取同一数据的结果是一样的，这样在这个事务还没有结束时，另外一个事务也访问该同一数据。读事务将禁止写事务（但允许读事务），写事务则禁止任何其他事务

  避免了不可重复读取和脏读，但是有时可能出现幻象读。如T1改完数据后，T2又插入了一个，而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读

- **可串行化（SERIALIZABLE）**：强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题

18. CopyOnWriteArrayList实现原理

支持高效率并发且是线程安全的，读操作无锁的ArrayList。它不存在扩容的概念，每次写操作都要复制一个副本，在副本的基础上修改后改变Array引用，合适读多写少的场景，不能用于实时读的场景。

CopyOnWriteArrayList的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，读操作的性能得到大幅度提升。

`CopyOnWriteArrayList` 类的所有可变操作（add，set等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，并不直接修改原有数组对象，而是对原有数据进行一次拷贝，将修改的内容写入副本中。写完之后，再将修改完的副本替换成原来的数据，这样就可以保证写操作不会影响读操作了。

19. 深克隆（深拷贝）和浅克隆（浅拷贝）

浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址

浅克隆方式：直接调用Object.clone()，注意要implements Cloneable

```java
public class Person implements Cloneable {
    private Address address;
    // 省略构造函数、Getter&Setter方法
    @Override
    public Person clone() {
        try {
            Person person = (Person) super.clone();
            return person;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

如上图，这样调用person.clone()的时候只是把person拷贝了，它的引用类型属性addresss没有拷贝

深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址

深拷贝方式：重写该类的clone()方法，在该重写方法里把每个引用类型属性也给clone()了

```java
public class Address implements Cloneable{
    private String name;
    // 省略构造函数、Getter&Setter方法
    @Override
    public Address clone() {
        try {
            return (Address) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

```
public Person clone() {
    try {
        Person person = (Person) super.clone();
        person.setAddress(person.getAddress().clone());
        return person;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

如上图，把address也implements Cloneable，然后把person的clone方法改一下，把address也clone了

总之深浅克隆都会在堆中新分配一块区域，区别在于对象属性引用的对象是否需要进行克隆（递归性的）

20. 排序算法

内部排序：数据记录在内存中进行排序。

外部排序：因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存（磁盘）。这里只讨论内部排序。

排序算法	时间复杂度			空间复杂度	排序方式	稳定性
	最好	最坏	平均			
冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	In-place	不稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	In-place	稳定
希尔排序	O(n log n)	$O(n^2)$	-	O(1)	In-place	不稳定
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)	Out-place	稳定
快速排序	O(n log n)	O(n log n)	O(n log n)	O(log n)	In-place	不稳定
堆排序	O(n log n)	O(n log n)	O(n log n)	O(1)	In-place	不稳定
计数排序	O(n+k)	O(n+k)	O(n+k)	O(k)	Out-place	稳定
桶排序	O(n+k)	$O(n^2)$	O(n+k)	O(n+k)	Out-place	稳定
基数排序	O(n×k)	O(n×k)	O(n×k)	O(n+k)	Out-place	稳定

n：数据规模

k："桶" 的个数

In-place：占用常数内存，不占用额外内存

Out-place：占用额外内存

- **冒泡排序**：重复地遍历要排序的序列，依次比较两个元素，如果它们的顺序错误就把它们交换过来。遍历序列的工作是重复地进行直到没有再需要交换为止，此时说明该序列已经排序完成。每一轮都会把此轮最大/最小的那个移到最后面。
- **选择排序**：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排

序完毕。【**不稳定**：在把后面的数据放到前面时，实际上两个数据进行了**交换**（注意不是整体后移），所以当a1=a2时且原本a1在前面时，a1可能因为和a2后面的某个数字发生交换而跑到了a2后面，所以最后的结果a1可能会在a2后面】

- **插入排序**：从第一个元素开始，认为它已经排好序，对于未排序的数据，从已排序部分从后向前进行扫描，找到合适的位置将未排序的数据插进去。【**稳定**：因为是整体后移而非交换】
- **希尔排序**：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，设置若干个gap，每相隔一个gap的元素在同一组，组内自己插入排序，一开始gap=length/2，后面每一轮gap都缩为前一轮的一半。等gap为1时，其实就是传统的全组插入排序，但这时已经排过很多轮了，微调一下就可以了。【**不稳定**：因为gap的原因相同元素可能被分到不同组】
- **归并排序**：分治法的典型应用，一个递归的过程——递归函数的输入为一个长度为n的数组，如果这个n为1则直接返回，否则把这个数组切成两半，递归调用两次，分别对两个子数组进行排序，并将这两个排序好的子数组进行合并，合并过程如下：若两个子数组分别长n1、n2，则设置一个长为n1+n2的额外空间，然后双指针法两个指针分别指向两个子数组的第一个元素，然后不断把最小的那个元素移到额外空间里，最后返回这个额外空间就是这两个子数组合并后的结果，然后返回到上一层递归函数，再进行合并.....【**稳定**：合并时按顺序】
- **快速排序**：也是分治法、递归——递归函数的参数是左边界l、右边界r，然后i=l，j=r，基准值pivot=nums[i]，while(i<j)时j、i交替走，交替更改nums[i]、nums[j]，最后ij相遇，另nums[i]=pivot，然后把[l,i-1]和[i+1,r]两半部分也进行递归。【**不稳定:**因为j--的条件是>=而不是>，i++的条件是<=而不是<】
- **堆排序**：

  1. 基本思想

  利用大顶堆(小顶堆)堆顶记录的是最大关键字(最小关键字)这一特性，使得每次从无序中选择最大记录(最小记录)变得简单。

  > ① 将待排序的序列构造成一个最大堆，此时序列的最大值为根节点
  > ② 依次将根节点与待排序序列的最后一个元素交换
  > ③ 再维护从根节点到该元素的前一个节点为最大堆，如此往复，最终得到一个递增序列

  【**空间复杂度O(1)**：因为所有元素是就地构建为大顶堆/小顶堆的，**不稳定**：因为调整堆的过程中可能会改变相同元素的相对顺序】

技术深度

1. 有没有看过JDK源码，看过的类实现原理是什么

hashmap

2. HTTP协议

超文本传输协议，是一种应用层协议，是万维网的数据通信的基础。

HTTP是一个客户端终端（用户）和服务器端（网站）请求和应答的标准（TCP），定义Web客户端如何从Web服务器请求Web页面，以及服务器如何把Web页面传送给客户端。

HTTP协议采用了请求/响应模型。通常，由HTTP客户端发起一个请求，创建一个到服务器指定端口（默认是80端口）的TCP连接。HTTP服务器则在那个端口监听客户端的请求。一旦收到请求，服务器会向客户端返回一个状态，比如"HTTP/1.1 200 OK"，以及返回的内容，如请求的文件、错误消息、或者其它信息。

http协议是基于TCP/IP协议之上的（但不是必须TCP/IP）。HTTP协议规定，请求从客户端发出，最后服务器端响应该请求并返回。换句话说，肯定是先从客户端开始建立通信的，服务器端在没有接收到请求之前不会发送响应。

使用HTTP协议，每当有新的请求发送时，就会有对应的新响应产生。协议本身并不保留之前一切的请求或响应报文的信息。这是为了更快地处理大量事务。

而Cookie可以帮助HTTP协议管理用户状态。

HTTP/1.1协议中共定义了八种方法（也叫"动作"）来以不同方式操作指定的资源：

Get、Head、Post、Put、Delete、Trace、Options、Connect

3. TCP协议

传输层协议，应用程序之间的通讯，是面向连接的、可靠的。

当应用程序希望通过 TCP 与另一个应用程序通信时，它会发送一个通信请求。这个请求必须被送到一个确切的地址。在双方三次"握手"之后，TCP 将在两个应用程序之间建立一个连接，四次"握手"才能断开连接

4. 一致性Hash算法

hash算法：将任意长度的二进制值映射为较短的固定长度的二进制值

普通的hash算法出现的问题是，在分布式的存储系统中，如果有一个机器加入或退出这个集群，则所有的数据映射都无效了，如果是持久化存储则要做数据迁移，如果是分布式缓存，则其他缓存就失效了。

一致性Hash算法：将hash空间连成一个环，将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或唯一主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置。

然后将objectA、objectB、objectC、objectD四个对象通过特定的Hash函数计算出对应的key值，然后散列到Hash环上,然后从数据所在位置沿环顺时针"行走"，第一台遇到的服务器就是其应该定位到的服务器。

这样，在进行服务器添加时，只需通过hash算法将Node X映射到环中，通过按顺时针迁移的规则，那么Object C被迁移到了Node X中，其它对象还保持这原有的存储位置。受影响的收据只有Node X到Node C之间的数据，数据的迁移达到了最小。

通过增加虚拟节点可以保证平衡性，做法是为每个物理节点关联几个虚拟节点，虚拟节点的数据均定位到对应的物理节点上。

5. 类加载器如何卸载字节码

6. IO和NIO的区别，NIO优点

区别：

IO	NIO
面向流	面向缓冲
阻塞IO	非阻塞IO
	有选择器

- **面向流/缓冲：**NIO的缓冲导向将数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动，增加了处理过程中的灵活性
- **阻塞/非阻塞：**传统的IO操作，比如read()，当没有数据可读的时候，线程一直阻塞被占用，直到数据到来。NIO中没有数据可读时，read()会立即返回0，线程不会阻塞
- **选择器：**NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来"选择"通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

  NIO中，客户端创建一个连接后，先要将连接注册到Selector，相当于客人进入餐厅后，告诉前台你要用餐，前台会告诉你你的桌号是几号，然后你就可能到那张桌子坐下了，SelectionKey就是桌号。当某一桌需要服务时，前台就记录哪一桌需要什么服务，比如1号桌要点菜，2号桌要结帐，服务员从前台取一条记录，根据记录提供服务，完了再来取下一条。这样服务的时间就被最有效的利用起来了

7. Java线程池的实现原理，keepAliveTime等参数的作用

其实java线程池的实现原理很简单，说白了就是一个线程集合workerSet和一个阻塞队列workQueue。当用户向线程池提交一个任务(也就是线程)时，线程池会先将任务放入workQueue中。workerSet中的线程会不断的从workQueue中获取线程然后执行。当workQueue中没有任务的时候，worker就会阻塞，直到队列中有任务了就取出来继续执行

参数：

- **corePoolSize:** 规定线程池有几个线程(worker)在运行。
- **maximumPoolSize:** 当workQueue满了,不能添加任务的时候，这个参数才会生效。规定线程池最多只能有多少个线程(worker)在执行
- **keepAliveTime:** 超出corePoolSize大小的那些线程的生存时间,这些线程如果长时间没有执行任务并且超过了keepAliveTime设定的时间，就会消亡
- **unit:** 生存时间对于的单位
- **workQueue:** 存放任务的队列
- **threadFactory:** 创建线程的工厂
- **handler:** 当workQueue已经满了，并且线程池线程数已经达到maximumPoolSize，将执行**拒绝策略**

8. HTTP补充

- **HTTP连接：**如果每进行一次 HTTP 通信就要新建一个 TCP 连接，那么开销会很大。长连接只需要建立一次 TCP 连接就能进行多次 HTTP 通信：
  - 从 HTTP/1.1 开始默认是长连接的，如果不希望使用长连接，则在请求头中要加入 `Connection : close`

- 在 HTTP/1.1 之前默认是短连接的，如果需要使用长连接，则使用 `Connection : Keep-Alive`

- **流水线：**流水线是在同一条长连接上连续发出请求，而不用等待响应返回，这样可以减少延迟
- **URI、URL、URN：**URI 除了包含 URL，还包含 URN

  URL可以用来定位资源，而URN只是一个我i一的命名，并不能用来定位，所以URI和URL几乎可以混用

  抽象地说，每个URL都是URI，但并非每个URI都是URL
- **HTTP连接池实现原理：**

  客户端发起请求时，如何获得一个http链接呢？其实是从连接池里拿连接

  连接池里针对每一个目标服务器都会由一些连接保存（也可能妹有），一个RouteToPool管理一个池，持有pending 、leased 、available 这些对象，CPool维护三个总的对象

  介绍：

  - **LinkedList available** – 存放可用连接

    available 表示可用的连接，他们是用LinkedList来存放的。

    当需要取连接时，是从链表的头部开始遍历，直到获取可用的连接为止；同时，当连接被使用完后，也会被放入链表的头部。这样做的目的是为了尽快的获取到可用的连接，因为在链表头部的都是刚放入链表的连接，离过期时间肯定是最远的。如果从链表尾部获取的话，那么很可能会获取到失效的连接。

    同时，删除链表的失效连接是从链表尾部开始遍历的。定期清理。
  - **HashSet leased** – 存放被租用的连接
    leased存放正在被使用的连接。如果一个连接被创建或者从available 链表中取出，就会先放入leased集合中。同时，用完连接后，就从leased集合中移除掉掉。因为就add和remove操作，所以使用HashSet的数据结构，效率很高。

    maxTotal的配置就是available链表和leased集合的总和限制。
  - **LinkedList pending** – 存放等待获取连接的线程的Future
    当从池中获取连接时，如果available链表没有现成可用的连接，且当前路由或连接池已经达到了最大数量的限制，也不能创建连接了，此时不会阻塞整个连接池，而是将当前线程用于获取连接的Future放入pending链表的末尾，之后当前线程调用await()，释放持有的锁，并等待被唤醒。

    当有连接被release()释放回连接池时，会从pending链表头获取future，并唤醒其线程继续获取连接，做到了先进先出。

  建立连接：

  - 如果available中有可用连接，则直接返回该连接
  - 否则，判断routeToPool和全局的连接数量是否分别达到maxPerRoute和MaxTotal的限制，如果都没达到，则创建一个连接，然后返回
  - 如果上面的条件都没达成，就挂起当前线程，然后构造一个Future对象放入pending队列，等待有连接释放后唤醒自己

  断开连接：

  - 若连接没有被标记重用，则分别从routeToPool和外层CPool中删除该链接，并关闭该连接
  - 否则，从routeToPool中的leased集合删除，并添加到available队首
  - 然后从外层CPool中的leased集合中删除，并添加到available队首
  - 唤醒该routeToPool的pending队列的第一个PoolEntryFulture，将其从pending队列和三处，并从外层CPool的pending队列中删除

定时清理：

- 如果每次获取连接时都要去判断连接是否过期或者关闭，会造成一定的性能损耗。另外如果连接长时间没用，长期闲置在那也是一种资源浪费。所以httpclient提供了一个机制，就是后台线程定时的清除过期和闲置过久的连接

- **三次握手、四次挥手，为什么呢**

  因为三次握手的时候，确认信息AKN和建立连接的信号是在第二次握手一起发过来的，但是挥手的时候，服务端第二次只发送确认收到的信息，然后处理完数据，第三次才发送断开的信号，所以多了一次

9. HTTPS与HTTP

HTTP 有以下安全性问题：

- 使用明文进行通信，内容可能会被窃听
- 不验证通信方的身份，通信方的身份有可能遭遇伪装
- 无法证明报文的完整性，报文有可能遭篡改

HTTPS 并不是新协议，而是让 HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信

通过使用 SSL，HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）

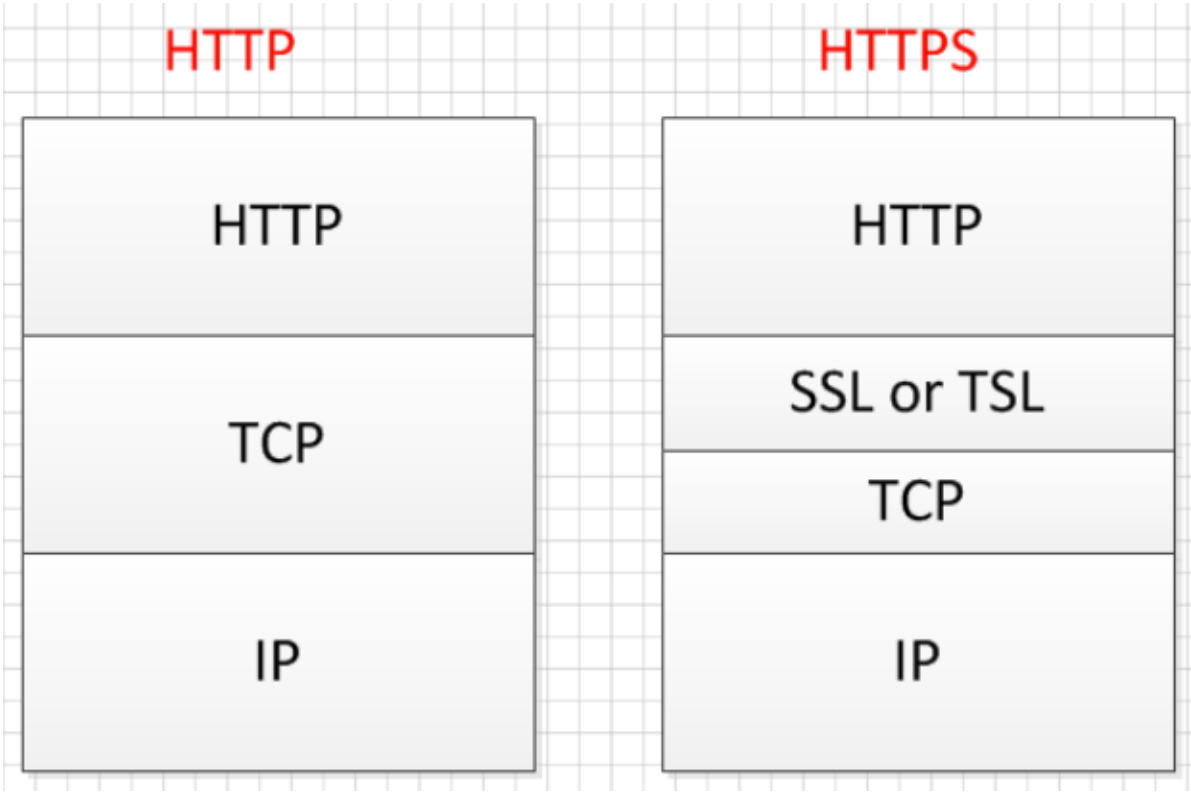
HTTPS 的缺点：

- 因为需要进行加密解密等过程，因此速度会更慢
- 需要支付证书授权的高额费用

！！！！！！！！！【HTTPS中SSL/TLS的原理】！！！！！！！！！！！！！！！！

SSL：安全套接字协议，TLS基于SSL

- 对称加密：通信双方使用同一密钥进行加密和解密。缺点：如果密钥一暴露那就失去保护功能了。
- 非对称加密：所有想和A通信的发送方发出的消息都是用A提供的同一公钥进行加密的，A收到这些消息后用自己的私钥进行解密。缺点：C和A正式通信之前要先获得A的公钥，如果这时候来一个攻击者B，给C发送一个诈包，骗它说这是A的公钥（实际上是B自己的公钥），那么C给A发的消息就会用B的公钥进行加密，B捕获这些消息就可以用自己的对其解密。
- 证书加密（数字签名技术）：由一个受信任的第三方证书颁发机构CA给各服务器颁发证书，CA知道服务器的公钥，对公钥进行hash运算后得到一个hash值，然后对这个hash值进行RSA加密运算得到一个数字签名。服务器把这个证书发给客户端，客户端知道CA的公钥，对签名进行RSA解密运算得到hash值，再对公钥进行hash运算也得到hash值，比较这两个hash是否相同，相同则验证通过，该公钥没有经过篡改。



Https总体过程——两次http请求：

第一次：

1.客户端想服务器发起HTTPS的请求，连接到服务器的443端口；

2.服务器将包含非对称加密的公钥的**证书**回传到客户端

3.服务器接受验证该证书，如果有问题，则HTTPS请求无法继续；如果没有问题，则上述证书中的公钥是合格的。客户端这个时候随机生成一个私钥，成为client key，用于对称加密数据。使用前面的公钥对client key进行非对称加密；

第二次：

4.进行二次HTTP请求，将加密之后的client key传递给服务器；

5.服务器使用私钥进行解密，得到client key,使用client key对数据进行对称加密

6.将对称加密的数据传递给客户端，客户端使用非对称解密，得到服务器发送的数据，完成第二次HTTP请求。

也就是正式通信数据用对称加密，该对称加密的密钥用非对称加密，该非对称加密的公钥用证书加密（数字签名）

10. 数据库连接池实现原理

负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个。

数据库连接池的基本思想，就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。

我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。

更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况，为系统开发、测试及性能调整提供依据。

11. 数据库的实现原理

技术框架

1.看过哪些开源框架的源码

2.为什么要用Redis，Redis有哪些优缺点？Redis如何实现扩容？

用redis的意义：

- 高性能。对于常用数据，直接存到内存的缓存里就不用每次都从磁盘里拿，和传统数据库不同，redis是内存数据库，数据存在内存中，且redis存储的是键值对数据，读写速度非常快，被广泛应用于缓存。
- 高并发。直接操作缓存能够承受的数据库请求数量是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，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。进而，我们也就提高了系统整体的并发。

【还有一种著名的分布式缓存脚Memcached，两者都是基于内存的，性能都很高，但redis支持更丰富的数据类型，有灾难恢复机制，且原生支持集群模式.....】

redis的优缺点：

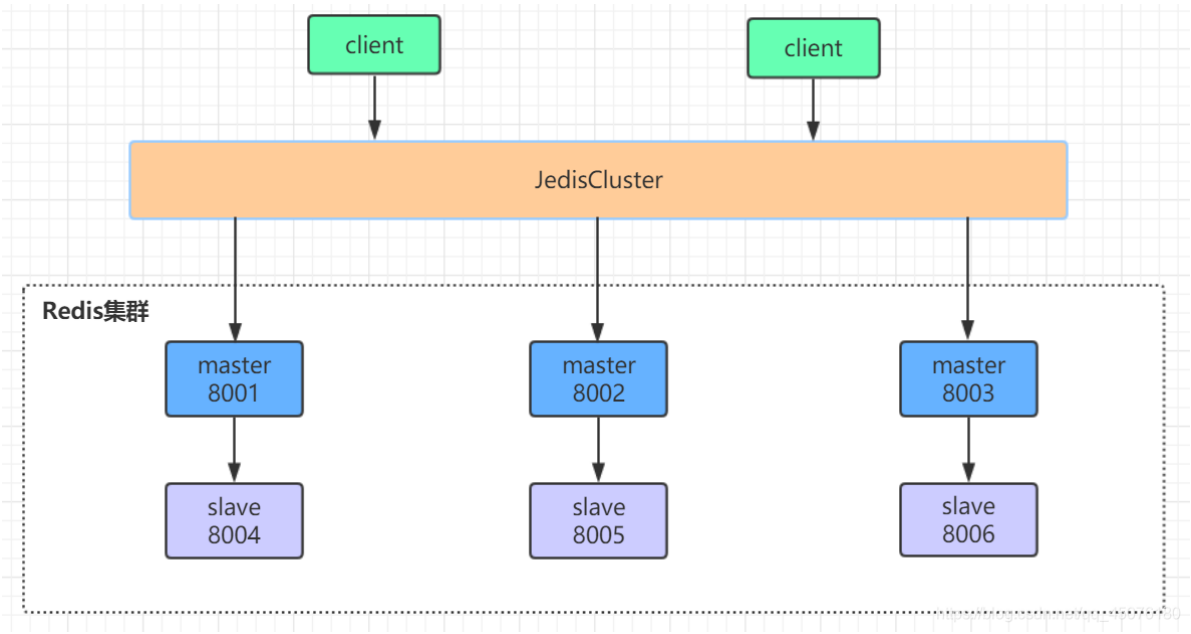
- 优点：
  1. **支持多种数据类型**：支持set（集合）、zset（有序集合）、list、hash、string这基本类型，还支持基数统计、位存储、地理位置这三种特殊数据结构。 如果在做好友系统，查看自己的好友关系，如果采用其他的key-value系统，则必须把对应的好友拼接成字符串，然后在提取好友时，再把value进行解析，而redis则相对简单，直接支持list的存储(采用双向链表或者压缩链表的存储方式)。
  2. **持久化存储**：内存数据库最担心的是万一机器宕机数据就丢失，redis使用**RDB快照文件**（redis在内存中所存储的全部数据的二进制表示，可以通过加载该文件恢复数据，rdb快照可以存在硬盘或复制多份传到远端服务器）或**AOF**（通过保存Redis服务器所执行的写命令来记录数据库状态，存在硬盘里）做数据持久化。
  3. **性能很好**： 全内存操作，所以读写性能很好。
- 缺点；
  1. 由于是内存数据库，所以单台机器存储的数据量跟机器本身的内存大小有关。虽然redis本身有key过期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

     删除数据的策略：惰性删除（只会在取出 key 的时候才对数据进行过期检查）、定期删除（每隔一段时间抽取一批 key 执行删除过期 key 操作）
  2. 通过生成rdb文件进行同步备份，会消耗一定的CPU资源
  3. 修改配置文件，进行重启，将硬盘中的数据加载进内存，时间比较久。在这个过程中，redis不能提供服务。
  4. 不支持回滚，redis事务不满足原子性，也不满足持久性。

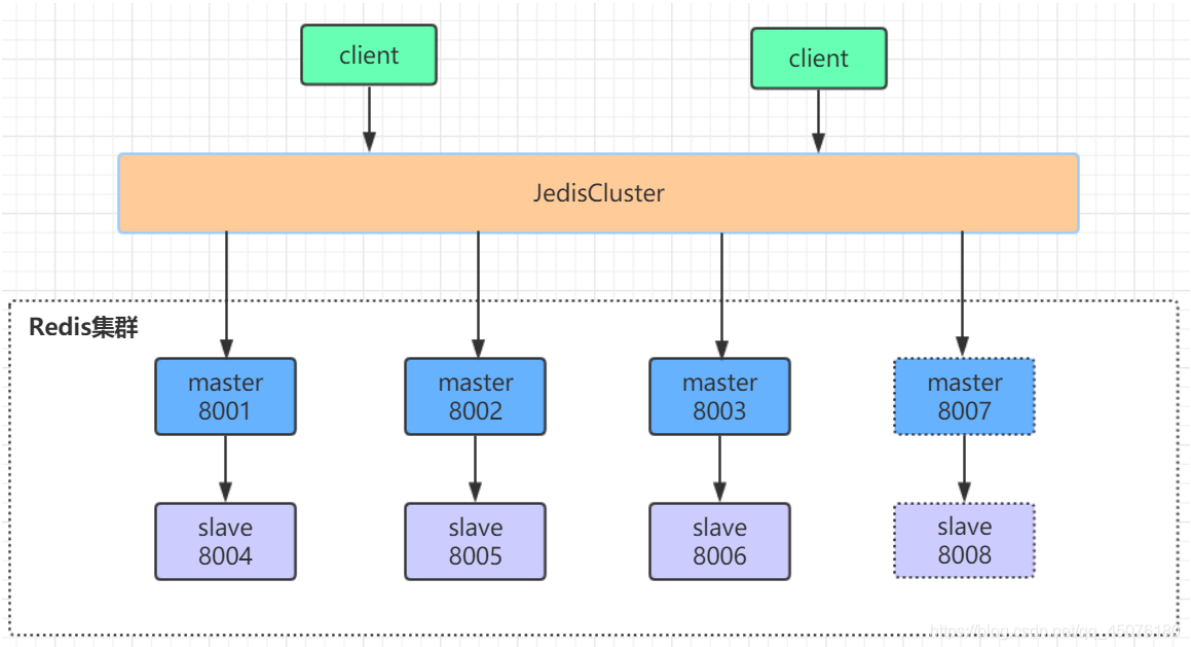
redis如何扩容：

可以通过增加/删除redis集群节点来实现redis集群的动态扩容、缩容。

举例：现在redis集群中有8001-8006这6台主机，且都是一主一从的结构：



要想添加8007、8008这一主一从两个节点：



步骤如下：

1. 启动原集群，通过cluster nodes命令看是否工作正常

```
192.168.100.100:8001> cluster  nodes
910192740c6062e683433f326fb029477e88f0a4 192.168.100.100:8005@18005 slave cace1363f80592d3dc0910ce3366adfef8daf095 0 1604842711000 2 connected
3dffc18ff90ce132e57d3d233083be5d7c10efb0 192.168.100.100:8003@18003 master - 0 1604842710000 3 connected 10923-16383
9d271fbd6fc1c6a2ab0763ca76ed9f5d710a5768 192.168.100.100:8006@18006 slave 3dffc18ff90ce132e57d3d233083be5d7c10efb0 0 1604842711605 3 connected
4662bca5de1280b0a09fd3fe769a1edc90878f7a 192.168.100.100:8004@18004 slave baf0c2f3afde2410e34351a8261a703f1394cee9 0 1604842709000 1 connected
baf0c2f3afde2410e34351a8261a703f1394cee9 192.168.100.100:8001@18001 myself,master - 0 1604842710000 1 connected 0-5460
cace1363f80592d3dc0910ce3366adfef8daf095 192.168.100.100:8002@18002 master - 0 1604842709000 2 connected 5461-10922
```

2. 在集群所在的主文件夹/usr/local/redis-cluster下创建8007和8008两个文件夹，并拷贝8001-8006任意一个节点的redis.conf配置文件到这两个文件夹下
3. 使用redis-cli的add-node命令新增一个主节点8007（master），最后看到日志里出现"[OK] New node added correctly"提示代表新节点加入成功

```
src/redis-cli --cluster add-node 192.168.100.100:8007 192.168.100.100:8001
#前面ip:8007是新增的，后面ip:8001是原来的（一般都要指定一个原来的）
```

4. 使用redis-cli的rehash命令为8007分配hash槽

```
src/redis-cli --cluster reshard 192.168.100.100:8001
#找到集群中的任意一个主节点，对其进行重新分片工作。
#执行上面的命令后，进入手动分配槽位流程：

... ...
How many slots do you want to move (from 1 to 16384)? 600
(ps:需要多少个槽移动到新的节点上，自己设置，比如600个hash槽)
What is the receiving node ID? 2728a594a0498e98e4b83a537e19f9a0a3790f38
(ps:把这600个hash槽移动到哪个节点上去，需要指定节点id)
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node 1:all
(ps:输入all为从所有主节点(8001,8002,8003)中分别抽取相应的槽数指定到新节点中，抽取的总槽数为600个)
 ... ...
Do you want to proceed with the proposed reshard plan (yes/no)? yes
(ps:输入yes确认开始执行分片任务)
... ...
```

5. 使用redis-cli的add-node命令一个主节点8008

```
src/redis-cli --cluster add-node 192.168.100.100:8008 192.168.100.100:8001
#前面ip:8008是新增的，后面ip:8001是原来的（一般都要指定一个原来的）
```

6. 进入8008的客户端，使用集群命令replicate，把当前的8008节点指定到主节点8007下

```
# 进入8008的客户端
[root@CentOS7 redis-6.0.9]# src/redis-cli  -p 8008

# 在8008客户端下指定8008节点的主从关系
# 4b339ad25b4884c2ff6de8a8ec2bc8766f8faf0b 是8007节点的id
127.0.0.1:8008> cluster replicate 4b339ad25b4884c2ff6de8a8ec2bc8766f8faf0b
```

3.Netty是如何使用线程池的，为什么这么使用

Netty：一个广泛使用的java网络编程框架

一般的线程池ThreadPoolExecutor设置了一个任务队列，存放等待运行的任务，存在性能瓶颈：如果线程数量特别多时，多个线程去竞争一个队列，就会导致性能下降。

Netty使用**事件循环线程池（EventLoopGroup）**这一接口,更适用于事件循环的场景。并且提供了多种实现该接口的类。

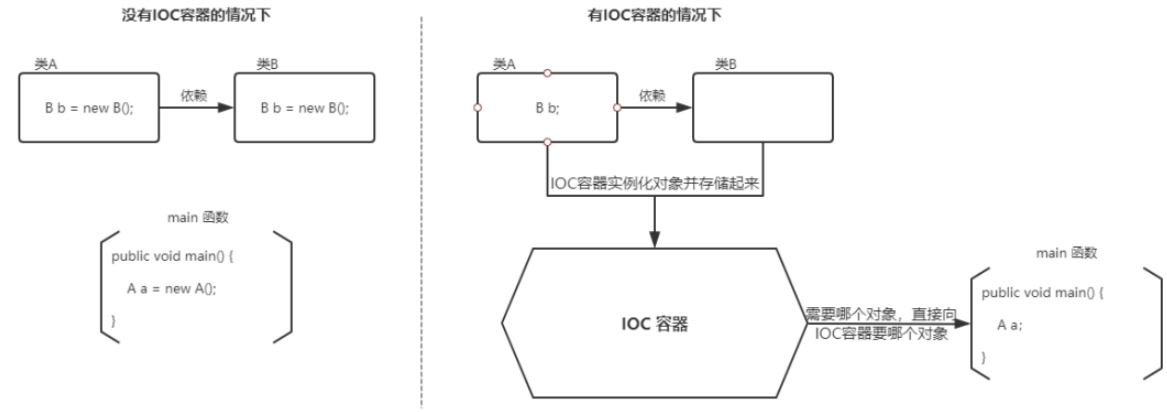
4.为什么要使用Spring，Spring的优缺点有哪些

Spring集合了很多模块，使用这些模块可以很方便地协助进行开发，如IOC、AOP，还能很方便的集成第三方组件（缓存等），测试也很方便。

两个核心功能：IOC、AOP

5.Spring的IOC容器初始化流程

Spring IOC（控制反转）：一种设计思想，也就是将原本在程序中手动创建对象的控制权交给spring框架来管理。



IOC容器： IoC 容器实际上就是个 Map（key，value），Map 中存放的是各种对象。 将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样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。 IoC 容器就像是一个工厂一样，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。

bean代指的就是被IOC容器所管理的对象

IOC容器初始化流程：

1. 资源定位
2. BeanDefiniton载入
3. 向IOC容器中注册这些BeanDefinition

6.如何配置Bean

三种方式：XML文件、注解、java配置类

XML文件

例如有以下两个类：

```
public class User {
    private String name;
    private int age;
    private Dog dog;
    //get、set、toString方法略
}
```

```
public class Dog {
    private String name;
    private String breed; //品种
    private int age;
   //get、set、toString方法略
}
```

配置文件：applicationContext.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="jackma" class="com.tyq.dto.User">
        <property name="name" value="jackma" />
        <property name="age" value="55" />
        <property name="dog" ref="jm" />      #引用类型的成员变量，也是一个bean
     </bean>

    <bean id="jm" class="com.tyq.dto.Dog">
        <property name="name" value="jack" />
        <property name="breed" value="金毛" />
        <property name="age" value="2" />
    </bean>
</beans>
```

如何使用：

```
public class test {
    public static void main(String args[]){
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");#加载配置
        User user = (User) context.getBean("jackma");
        System.out.println(user);
    }
}
```

注解

将类声明为bean的注解：

- `@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。(通过类路径扫描来自动侦测以及自动装配到 Spring 容器中（可以使用 `@ComponentScan` 注解定义要扫描的路径)
- `@Controller`：对应 Spring MVC 控制层，主要用于接受用户请求并调用 `Service` 层返回数据给前端页面。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。

作用于方法的注解：

`@Bean`：表明标有该注解的方法中会定义产生这个bean，默认方法名就是bean名，方法返回值就是这个bean对象

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

上面的代码相当于下面的xml配置文件：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

用于注入Bean的注解：

- @Autowired
- @Resource
- @Inject

Java配置类

- 使用@Configuration注解需要作为配置的类，表示该类将定义Bean的元数据，同时该通过@Configuration注解的类也是一个Bean
- 使用@Bean注解相应的方法，该方法名默认就是Bean的name，该方法返回值就是Bean的对象。
- 使用AnnotationConfigApplicationContext或子类进行加载基于java类的配置。

例子：两个类：

```
@Component("jackma")
public class User {
    private String name;
    private int age;
    private Dog dog;

  //get，set方法略
}
```

```
public class Dog {
    private String name;
    private String kind;
    private int age;
    //get，set方法略
}
```

配置类：（作用相当于上面的applicationContext.xml文件）

```
@Configuration
public class DemoConfig {
    @Bean
```

```
    public User jackma(){
        return new User();
    }
    @Bean
    public Dog dog(){
        return  new Dog();
    }
    @Bean  //两个狗
    public Dog haqi(){
        return new Dog();
    }
}
```

使用：

```
public class test {
    public static void main(String args[]){
        ApplicationContext context = new
AnnotationConfigApplicationContext(DemoConfig.class);#加载配置
        User user = (User) context.getBean("jackma");
        System.out.println(user);
    }
}
```

7.Bean的作用域和生命周期

Bean的作用域

- **singleton（单例，默认）**：IOC容器中每个类只有唯一的一个实例对象。目的：提高性能、少创建实例、减少垃圾回收、缓存快速获取

  在多线程的情况下存在竞争统一资源的问题

- **prototype（多例）**：每次获取都会创建一个新的 bean 实例。也就是说，连续 `getBean()` 两次，得到的是不同的 Bean 实例。
- **request（仅web应用可用）**：每次http请求都会产生一个新的bean（请求bean），且该bean仅在当前http request内有效。
- **session（仅web应用可用）**：每一次来自新 session 的 HTTP 请求都会产生一个新的 bean（会话bean），该 bean 仅在当前 HTTP session 内有效。
- **websocket（仅web应用可用）**：每一次 WebSocket 会话产生一个新的 bean。

Bean作用域的配置方法

- xml方式，scope属性

  ```
  <bean id="..." class="..." scope="singleton"></bean>
  ```

- 注解方式，@Scope注解

```
@Bean
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Person personPrototype() {
    return new Person();
}
```

Bean的生命周期

Bean的生命周期完全由IOC容器进行管理（注意这里指单例bean）

（1）实例化

Bean容器找到bean的定义，并利用java reflection api创建一个bean的实例

1. Bean 容器找到配置文件中 Spring Bean 的定义。
2. Bean 容器利用 Java Reflection API 创建一个 Bean 的实例。

（2） 属性赋值

若涉及属性值则用set方法设置属性值，如果实现了一些xxxAware接口就调用相应的方法

3. 如果涉及到一些属性值 利用 `set()` 方法设置一些属性值。
4. 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法，传入 Bean 的名字。
5. 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoader` 对象的实例。
6. 如果 Bean 实现了 `BeanFactoryAware` 接口，调用 `setBeanFactory()` 方法，传入 `BeanFactory` 对象的实例。
7. 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。

（3）初始化

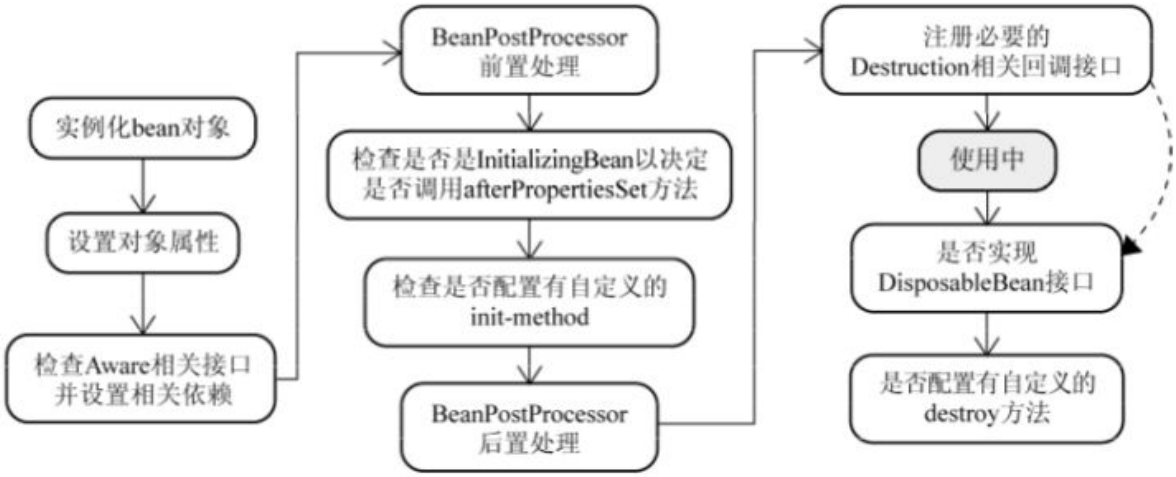
根据情况执行postProcessBeforeInitialization()方法、afterPropertiesSet()方法、init-method相关方法、postProcessAfterInitialization()方法

8. 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
9. 如果 Bean 实现了`InitializingBean`接口，执行`afterPropertiesSet()`方法。
10. 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
11. 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessAfterInitialization()` 方法

（4） 销毁

根据情况执行destroy()方法、destroy-method相关方法

12. 当要销毁 Bean 的时候，如果 Bean 实现了 `DisposableBean` 接口，执行 `destroy()` 方法。
13. 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。



8.Spring的IOC容器实现原理，为什么可以通过byName和ByType找到Bean

9.Spring AOP实现原理

AOP(Aspect-Oriented Programming: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基于动态代理:

- 如果要代理的对象，实现了某个接口，使用 **JDK Proxy**去创建代理对象
- 如果要代理的对象没有实现任何接口，使用 **Cglib**去创建代理对象

AspectJ是静态AOP代理，在编译阶段对程序源代码进行修改，spring AOP是运行阶段动态生成代理对象，**切面较多时建议选择Aspect，因为效率更高**

10.消息中间件是如何实现的，技术难点有哪些

11.Zookeeper实现原理，以及选主算法

12.为什么需要配置中心，配置中心如何实现的

系统架构

1. 如何搭建一个高可用系统
2. 哪些设计模式可以增加系统的可扩展性
3. 介绍设计模式，如模板模式，命令模式，策略模式，适配器模式、桥接模式、装饰模式，观察者模式，状态模式，访问者模式。
4. 抽象能力，怎么提高研发效率。

5. 什么是高内聚低耦合，请举例子如何实现
6. 什么情况用接口，什么情况用消息
7. 如果AB两个系统互相依赖，如何解除依赖
8. 如何写一篇设计文档，目录是什么
9. 什么场景应该拆分系统，什么场景应该合并系统
10. 系统和模块的区别，分别在什么场景下使用

分布式系统

1. 分布式事务，两阶段提交。
2. 如何实现分布式锁
3. 如何实现分布式Session
4. 如何保证消息的一致性
5. 负载均衡
6. 正向代理（客户端代理）和反向代理（服务器端代理）
7. CDN实现原理
8. 怎么提升系统的QPS和吞吐量
9. DNS的实现原理
10. 介绍下PAXOS协议
11. 介绍下Zookeeper的ZAB协议，如何选举LEADER？如何

实战能力

1. 有没有处理过线上问题？出现内存泄露，CPU利用率标高，应用无响应时如何处理的。
2. 开发中有没有遇到什么技术问题？如何解决的
3. 如果有几十亿的白名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能。
4. 新浪微博是如何实现把微博推给订阅者
5. Google是如何在一秒内把搜索结果返回给用户的。
6. 12306网站的订票系统如何实现，如何保证不会票不被超卖。
7. 如何实现一个秒杀系统，保证只有几位用户能买到某件商品。
8. 缓存失效如何解决？
9. 从数据库查询10G的数据并加载到内存中？
10. 如何设计一个流控功能？

软能力

1. 如何学习一项新技术，比如如何学习Java的，重点学习什么
2. 有关注哪些新的技术
3. 工作任务非常多非常杂时如何处理
4. 项目出现延迟如何处理
5. 和同事的设计思路不一样怎么处理
6. 如何保证开发质量
7. 职业规划是什么？短期，长期目标是什么
8. 团队的规划是什么
9. 能介绍下从工作到现在自己的成长在那里

qil何学习Java的，重点学习什么

有关注哪些新的技术

工作任务非常多非常杂时如何处理

项目出现延迟如何处理

和同事的设计思路不一样怎么处理

如何保证开发质量

职业规划是什么？短期，长期目标是什么

团队的规划是什么

能介绍下从工作到现在自己的成长在那里