必读

本pdf进行了加密（为了避免盗版），禁止复制和打印，如果有读者朋友想要进行打印或者复制，请在公众号 **小瑶学Java** 中的公众号菜单栏添加小瑶的微信进行获取 **解密密码**

请主动备注自己的来意，为什么要进行打印或者复制，说明用途，小瑶初步审核下，就会直接把密码告诉大家的。

推荐

面试时的**高频面试算法题**（如果面试准备时间不够，那么集中把这些算法题做完即可，命中率高达85%+）

需要的在下方公众号 **小夕学算法** 后台回复 **666**
一个专注用动画漫画结合的方法讲讲算法的原创公众号。

Java基础知识(*)

- https://blog.csdn.net/qq_16633405/article/details/79211002

Spring Boot 启动 流程(*)

- https://juejin.im/post/5b679fbc5188251aad213110#heading-0

Spring 一些面试题(*)

- https://www.ctolib.com/topics-35589.html

匿名内部类编译class(*)

- https://blog.csdn.net/lazyer_dog/article/details/50669473

为什么集合类没有实现Cloneable和Serializable接口？

- https://www.nowcoder.com/questionTerminal/2a4902f67d5b49b6b4c05f9d7e422caf

自动装箱原理

- https://www.jianshu.com/p/0ce2279c5691

final关键字

- http://www.importnew.com/7553.html

基于Redis的分布式锁

- https://segmentfault.com/a/1190000012919740

数据库分布式锁

- http://www.hollischuang.com/archives/1716

防备DDOS攻击(*)

- https://www.zhihu.com/question/19581905

什么时候Mysql调用行锁？(*)

- https://blog.csdn.net/songwei128/article/details/43418343

CMS,G1(*)

- https://crowhawk.github.io/2017/08/15/jvm_3/

内部类，外部类互访(*)

- https://blog.csdn.net/jueblog/article/details/13434551
- https://blog.csdn.net/Van_L_/article/details/54667365

设计模式(*)
熟背单例模式和工厂模式，会写适配器和建造者也行

- https://www.jianshu.com/p/8a293e4a888e
- https://segmentfault.com/a/1190000004255439

深拷贝，浅拷贝(*)

- https://segmentfault.com/a/1190000010648514

泛型擦除

- https://blog.csdn.net/briblue/article/details/76736356

Java 8 Stream, 函数编程

- https://www.jianshu.com/p/0c07597d8311
- https://www.jianshu.com/p/9bd647bcf1e3

中断线程

- https://www.jianshu.com/p/264d4e1b76af

Lock，tryLock，lockInterruptibly区别

- https://blog.csdn.net/u013851082/article/details/70140223

JUC

- http://www.cnblogs.com/chenpi/p/5358579.html#_label2
- http://www.cnblogs.com/chenpi/p/5614290.html

NIO

- https://blog.csdn.net/u013063153/article/details/76473578
- https://www.jianshu.com/p/052035037297
- https://segmentfault.com/a/1190000006824196

Start和run区别(*)

- https://blog.csdn.net/qq_36544760/article/details/79380963

jvm内存屏障

- https://www.jianshu.com/p/2ab5e3d7e510

Java构造器能被重载，但是不能被重写(*)

- https://blog.csdn.net/weixin_36513603/article/details/54968094

HttpSession

- https://blog.csdn.net/zy2317878/article/details/80275463

Thread类的方法

- https://blog.csdn.net/gxx_csdn/article/details/79210192

String是值类型，还是引用类型(*)

- https://blog.csdn.net/a220315410/article/details/27743607

Redis 实现消息队列

- 消息/订阅+List
- https://segmentfault.com/a/1190000012244418

minor gc full gc 区别(*)

- https://blog.csdn.net/u010796790/article/details/52213708

Java如何查看死锁

- https://blog.csdn.net/u014039577/article/details/52351626
- https://juejin.im/post/5aaf6ee76fb9a028d3753534

c3p0，dbcp与druid

- https://blog.csdn.net/qq_34359363/article/details/72763491

Spring Bean 生命周期(*)

- https://www.jianshu.com/p/3944792a5fff

Spring的BeanFactory和ApplicationContext的区别?

- ApplicationContext是实现类，继承ListableBeanFactory（继承BeanFactory），功能更多
- ApplicationContext默认立即加载，BeanFactory懒加载
- https://my.oschina.net/yao00jun/blog/215642
- https://blog.csdn.net/qq_36748278/article/details/78264764

Java 如何有效地避免OOM：善于利用软引用和弱引用

- https://www.cnblogs.com/dolphin0520/p/3784171.html

分布式数据库主键生成策略(*)

- https://www.jianshu.com/p/a0a3aa888a49
- https://tech.meituan.com/MT_Leaf.html

String底层(*)

- https://blog.csdn.net/yadicoco49/article/details/77627302

count(1)、count(*)与count(列名)的执行区别

- https://blog.csdn.net/iFuMI/article/details/77920767

主键，唯一索引区别

- 1）主键一定会创建一个唯一索引，但是有唯一索引的列不一定是主键；
- 2）主键不允许为空值，唯一索引列允许空值；
- 3）一个表只能有一个主键，但是可以有多个唯一索引；
- 4）主键可以被其他表引用为外键，唯一索引列不可以；
- 5）主键是一种约束，而唯一索引是一种索引，是表的冗余数据结构，两者有本质的差别

死锁

产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。
- 占有且等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不可强行占有:进程已获得的资源，在未使用完之前，不能强行剥夺。

- 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁：
https://segmentfault.com/a/1190000000378725

- 确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生.
- 另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。
- 死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

乐观锁，悲观锁(*)

- https://blog.csdn.net/lovejj1994/article/details/79116272

公平锁、非公平锁

- 公平锁（Fair）：加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得
  非公平锁（Nonfair）：加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待
  非公平锁性能比公平锁高5~10倍，因为公平锁需要在多核的情况下维护一个队列
  Java中的ReentrantLock 默认的lock()方法采用的是非公平锁。

OOM分析

- https://blog.csdn.net/zheng12tian/article/details/40617369

JVM调优参数
知道-Xms，-Xmx，-XX:NewRatio=n，会算就行，笔试题考过

- https://www.jianshu.com/p/a2a6a0995fee

堆设置

- -Xms:初始堆大小
  -Xmx:最大堆大小
  -XX:NewSize=n:设置年轻代大小
  -XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4
  -XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5
  -XX:MaxPermSize=n:设置持久代大小

收集器设置

- -XX:+UseSerialGC:设置串行收集器
  -XX:+UseParallelGC:设置并行收集器
  -XX:+UseParalledlOldGC:设置并行年老代收集器
  -XX:+UseConcMarkSweepGC:设置并发收集器

垃圾回收统计信息

- -XX:+PrintGC
  -XX:+PrintGCDetails
  -XX:+PrintGCTimeStamps
  -Xloggc:filename

并行收集器设置

- -XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。
  -XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间
  -XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

并发收集器设置

- -XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。
  -XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

调优总结
年轻代大小选择

- 响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。
  吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

年老代大小选择

- 响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：
  并发垃圾收集信息 持久代并发收集次数 传统GC信息 花在年轻代和年老代回收上的时间比例 减少年轻代和年老代花费的时间，一般会提高应用的效率
  吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

较小堆引起的碎片问题

- 因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：
  -XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。
  -XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

synchronized实现原理(*)

- https://blog.csdn.net/javazejian/article/details/72828483
- 内存对象头， Mark Word保存锁信息
- JVM层：Monitor对象，字节码中的monitorenter 和 monitorexit 指令
- 无锁，偏向锁，轻量级锁（自选），重量级锁
- 可重入
- notify/notifyAll和wait方法，在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因.

synchronized, lock区别(*)

- https://blog.csdn.net/u012403290/article/details/64910926

Spring容器中Bean的作用域(*)
当通过Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。Spring支持如下5种作用域：

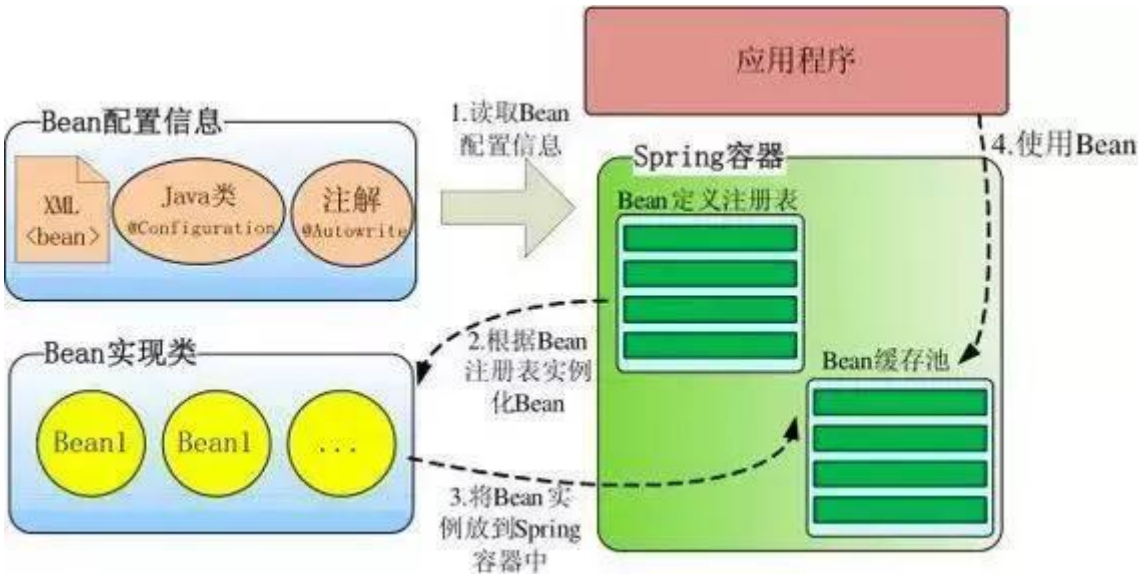
- singleton：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例
- prototype：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例
- request：对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的Bean实例。只有在Web应用中使用Spring时，该作用域才有效
- session：对于每次HTTP
  Session，使用session定义的Bean产生一个新实例。同样只有在Web应用中使用Spring时，该作用域才有效
- globalsession：每个全局的HTTP
  Session，使用session定义的Bean都将产生一个新实例。典型情况下，仅在使用portlet context的时候有效。同样只有在Web应用中使用Spring时，该作用域才有效

其中比较常用的是singleton和prototype两种作用域。对于singleton作用域的Bean，每次请求该Bean都将获得相同的实例。容器负责跟踪Bean实例的状态，负责维护Bean实例的生命周期行为；如果一个Bean被设置成prototype作用域，程序每次请求该id的Bean，Spring都会新建一个Bean实例，然后返回给程序。在这种情况下，Spring容器仅仅使用new 关键字创建Bean实例，一旦创建成功，容器不在跟踪实例，也不会维护Bean实例的状态。

如果不指定Bean的作用域，Spring默认使用singleton作用域。Java在创建Java实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，prototype作用域Bean的创建、销毁代价比较大。而singleton作用域的Bean实例一旦创建成功，可以重复使用。因此，除非必要，否则尽量避免将Bean被设置成prototype作用域。

Spring IOC实现原理， 相关知识(*)

Spring 启动时读取应用程序提供的Bean配置信息，并在Spring容器中生成一份相应的Bean配置注册表，然后根据这张注册表实例化Bean，装配好Bean之间的依赖关系，为上层应用提供准备就绪的运行环境。



Bean缓存池：HashMap实现

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。 Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、 Bean 实例代理、事件发布、资源装载等高级服务。

BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；

ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

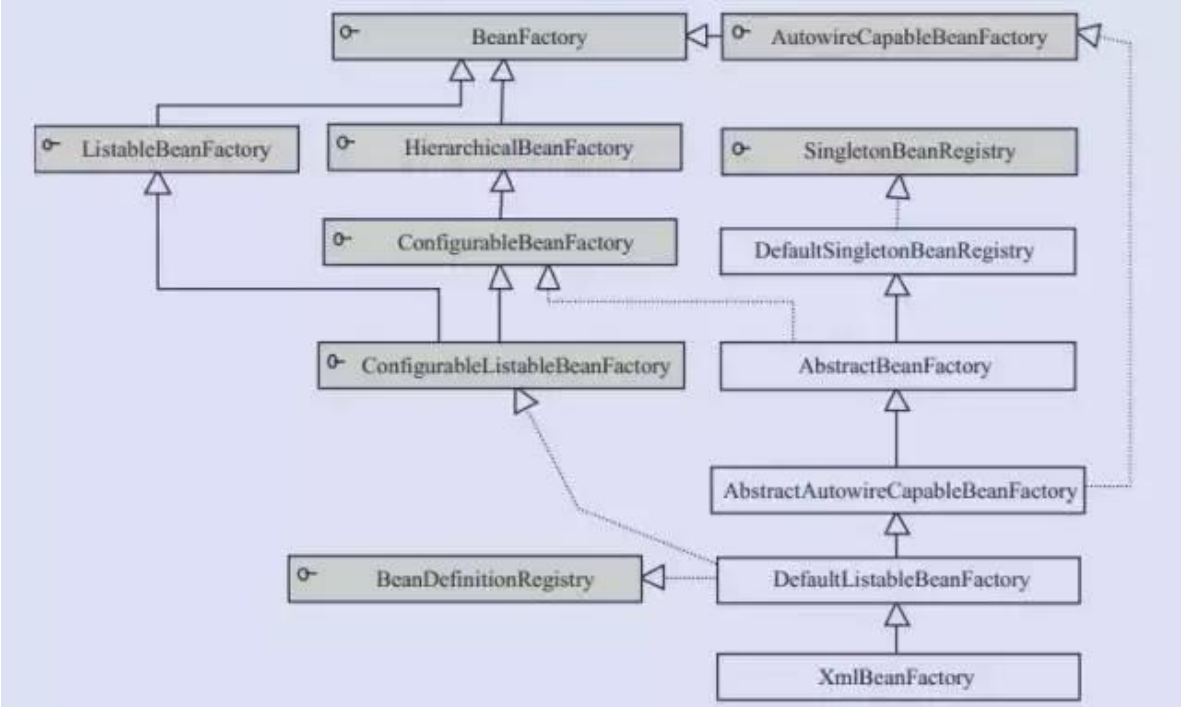


图 3-7　BeanFactory 类继承体系

BeanDefinitionRegistry： Spring 配置文件中每一个节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。

BeanFactory 接口位于类结构树的顶端 ，它最主要的方法就是 getBean(String beanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展：

ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；

HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器； 通过 HierarchicalBeanFactory 接口， Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

ConfigurableBeanFactory：是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；

SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例 Bean 的方法；

@Bean, @Component 区别

- Componet 一般放在类上面，Bean放在方法上面，自己可控制是否生成bean.
  bean 一般会放在classpath scanning路径下面，会自动生成bean.
  有Componet /bean生成的bean都提供给autowire使用.
- 在@Component中(@Component标注的类，包括@Service,@Repository, @Controller)使用@Bean注解和在@Configuration中使用是不同的。在@Component类中使用方法或字段时不会使用CGLIB增强(及不使用代理类：调用任何方法，使用任何变量，拿到的是原始对象，后面会有例子解释)。而在@Configuration类中使用方法或字段时则使用CGLIB创造协作对象（及使用代理：

拿到的是代理对象）;当调用@Bean注解的方法时它不是普通的Java语义，而是从容器中拿到的由Spring生命周期管理、被Spring代理甚至依赖于其他Bean的对象引用。在@Component中调用@Bean注解的方法和字段则是普通的Java语义，不经过CGLIB处理。

如何停止线程？

- 主线程提供volatile boolean flag, 线程内while判断flag
- 线程内while(!this.isInterrupted), 主线程里调用interrupt
- if(this.isInterrupted) throw new InterruptedException() 或return，主线程里调用interrupt
- 将一个线程设置为守护线程后，当进程中没有非守护线程后，守护线程自动结束

多线程实现方式？(*)

- extends Thread
- implements Runnable
- implements Callable, 重写call, 返回future (主线程可以用线程池submit)

线程池(*)
线程池处理过程：

- 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。
- 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。
- 如果无法将任务加入BlockingQueue(队列已满)，则创建新的线程来处理任务(注意，执行这一步骤需要获取全局锁)。
- 如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

四种线程池:

- CachedThreadPool
- FixedThreadPool
- ScheduledThreadPool
- SingleThreadExecutor

创建线程池的参数：

- corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。
- runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。
  ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按FIFO（先进先出）原则对元素进行排序。
  LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
  SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于Linked-BlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
  PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
- maximumPoolSize（线程池最大数量）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是，如果使用了无界的任务队列这个参数就没什么效果。
- ThreadFactory：用于设置创建线程的工厂
- RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新

任务时抛出异常。在JDK
1.5中Java线程池框架提供了以下4种策略。 AbortPolicy：直接抛出异常。 CallerRunsPolicy：只用调用者所在线程来运行任务。
DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。 DiscardPolicy：不处理，丢弃掉。

ArrayList,LinkedList

- ArrayList初始化可以指定大小，知道大小的建议指定
  arraylist添加元素的时候，需要判断存放元素的数组是否需要扩容（扩容大小是原来大小的1/2+1）
- LinkedList添加、删除元素通过移动指针 LinkedList遍历比arraylist慢，建议用迭代器
- ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
  对于随机访问get和set，ArrayList优于LinkedList，因为ArrayList可以随机定位，而LinkedList要移动指针一步一步的移动到节点处。（参考数组与链表来思考）
- 对于新增和删除操作add和remove，LinedList比较占优势，只需要对指针进行修改即可，而ArrayList要移动数据来填补被删除的对象的空间。

还准备了**一套上面资料对应的面试题（有答案哦）**和 面试时的高频面试算法题（如果面试准备时间不够，那么集中把这些算法题做完即可，命中率高达85%+）

需要的在下方公众号 **小夕学算法** 后台回复 **666**
一个专注用动画漫画结合的方法讲讲算法的原创公众号。

HashMap原理(*)

- HashMap**最多只允许一条Entry的键为Null**(多条会覆盖)，但允许多条Entry的值为Null
- HashSet 本身就是在 HashMap 的基础上实现的.
- 若负载因子越大，那么对空间的利用更充分，但查找效率的也就越低；若负载因子越小，那么哈希表的数据将越稀疏，对空间造成的浪费也就越严重。系统默认负载因子0.75
- 调用put方法存值时，HashMap首先会调用Key的hashCode方法，然后基于此获取Key哈希码，通过哈希码快速找到某个桶，这个位置可以被称之为bucketIndex.如果两个对象的hashCode不同，那么equals一定为false；否则，如果其hashCode相同，equals也不一定为 true。所以，理论上，hashCode可能存在碰撞的情况，当碰撞发生时，这时会取出bucketIndex桶内已存储的元素，并通过hashCode() 和 equals()来逐个比较以判断Key是否已存在。如果已存在，则使用新Value值替换旧Value值，并返回旧Value值；如果不存在，则存放新的键值对<Key, Value>到桶中。因此，在 HashMap中，equals() 方法只有在哈希码碰撞时才会被用到。
- 首先，判断key是否为null，若为null，则直接调用putForNullKey方法；若不为空，则先计算key的hash值，然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，则查找是否存在相同的key，若存在则覆盖原来key的value，否则将该元素保存在链头（最先保存的元素放在链尾）。此外，若table在该处没有元素，则直接保存。
- HashMap 永远都是在链表的表头添加新元素。

hash()和indexFor()

```
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {
    return h & (length-1);  // 作用等价于取模运算，但这种方式效率更高
}
```

- hash() 方法用于对Key的hashCode进行重新计算，而 indexFor()方法用于生成这个Entry对象的插入位置。当计算出来的hash值与hashMap的(length-1)做了&运算后，会得到位于区间[0，length-

1]的一个值。特别地，这个值分布的越均匀，HashMap 的空间利用率也就越高，存取效率也就越好，保证元素均匀分布到table的每个桶中以便充分利用空间。

- hash():使用hash()方法对一个对象的hashCode进行重新计算是为了**防止质量低下的hashCode()函数实现。**由于hashMap的支撑数组长度总是2 的幂次，通过右移可以使低位的数据尽量的不同，从而使hash值的分布尽量均匀。
- indexFor():**保证元素均匀分布**到table的每个桶中; 当length为2的n次方时，h&(length -1)就相当于对length取模，而且速度比直接取模要快得多，这是HashMap在速度上的一个优化.

扩容resize()和重哈希transfer()

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;

    // 若 oldCapacity 已达到最大值，直接将 threshold 设为 Integer.MAX_VALUE
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;              // 直接返回
    }

    // 否则，创建一个更大的数组
    Entry[] newTable = new Entry[newCapacity];

    //将每条Entry重新哈希到新的数组中
    transfer(newTable);

    table = newTable;
    threshold = (int) (newCapacity * loadFactor);  // 重新设定 threshold
}

void transfer(Entry[] newTable) {

    // 将原数组 table 赋给数组 src
    Entry[] src = table;
    int newCapacity = newTable.length;

    // 将数组 src 中的每条链重新添加到 newTable 中
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;   // src 回收

            // 将每条链的每个元素依次添加到 newTable 中相应的桶中
            do {
                Entry<K,V> next = e.next;

                // e.hash指的是 hash(key.hashCode())的返回值;
                // 计算在newTable中的位置，注意原来在同一条子链上的元素可能被分配到不同的子
链
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

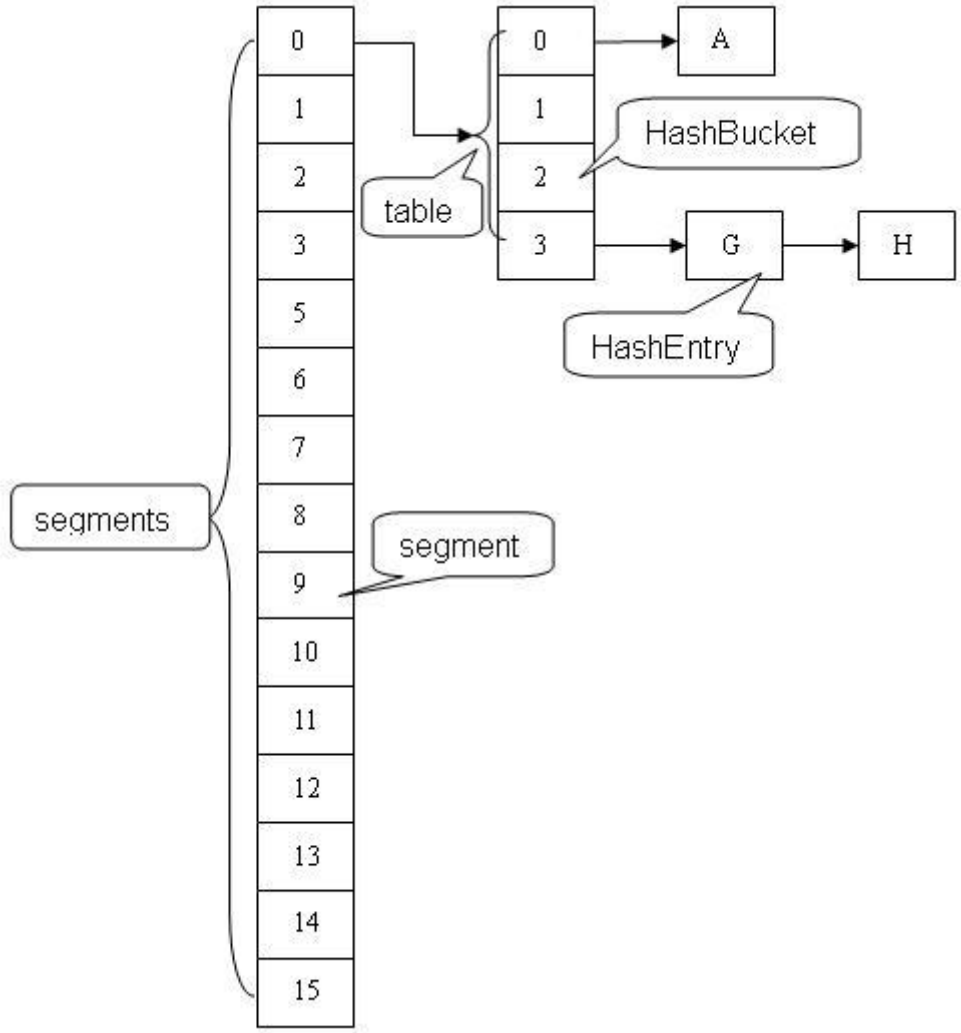
- 为了保证HashMap的效率，系统必须要在某个临界点进行扩容处理，该临界点就是HashMap中元素的数量在数值上等于threshold（table数组长度*加载因子)
- 重哈希的主要是一个重新计算原HashMap中的元素在新table数组中的位置并进行复制处理的过程

HashMap 的底层数组长度为何总是2的n次方

- 当底层数组的length为2的n次方时， h&(length - 1) 就相当于对length取模，而且速度比直接取模**快**得多，这是HashMap在速度上的一个优化
- 不同的hash值发生碰撞的概率比较小，这样就会使得**数据在table数组中分布较均匀**，空间利用率较高，查询速度也较快

ConcurrenytHashMap原理(*)

- https://blog.csdn.net/justloveyou_/article/details/72783008



- 通过锁分段技术保证并发环境下的写操作；
  通过 HashEntry的不变性、Volatile变量的内存可见性和加锁重读机制保证高效、安全的读操作；
  通过不加锁和加锁两种方案控制跨段操作的的安全性。

HashMap线程不安全

- https://blog.csdn.net/justloveyou_/article/details/72783008
- 在HashMap进行扩容重哈希时导致Entry链形成环。一旦Entry链中有环，势必会导致在同一个桶中进行插入、查询、删除等操作时陷入**死循环。**

Segment数组

```
static final class Segment<K,V> extends ReentrantLock
implements Serializable {
    transient volatile int count;    // Segment中元素的数量，可见的
    transient int modCount;  //对count的大小造成影响的操作的次数（比如put或者remove操
作）
    transient int threshold;      // 阈值，段中元素的数量超过这个值就会对Segment进行扩
容
    transient volatile HashEntry<K,V>[] table;  // 链表数组
    final float loadFactor;  // 段的负载因子，其值等同于ConcurrentHashMap的负载因子
    ...
}
```

- Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色
- 在Segment类中，count 变量是一个计数器，它表示每个 Segment 对象管理的 table 数组包含的 HashEntry 对象的个数，也就是 Segment 中包含的 HashEntry 对象的总数。特别需要注意的是，之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是对 ConcurrentHashMap 并发性的考虑：**因为这样当需要更新计数器时，不用锁定整个ConcurrentHashMap。**事实上，每次对段进行结构上的改变，如在段中进行增加/删除节点(修改节点的值不算结构上的改变)，都要更新count的值，此外，在JDK的实现中**每次读取操作开始都要先读取count的值。**特别需要注意的是，**count是volatile**的，这使得对count的任何更新对其它线程都是立即可见的。**modCount用于统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变**.**table**是一个典型的链表数组，而且也是**volatile**的，这使得对table的任何更新对其它线程也都是立即可见的。

HashEntry

```
static final class HashEntry<K,V> {
   final K key;                        // 声明 key 为 final 的
   final int hash;                    // 声明 hash 值为 final 的
   volatile V value;                 // 声明 value 被volatile所修饰
   final HashEntry<K,V> next;       // 声明 next 为 final 的

    HashEntry(K key, int hash, HashEntry<K,V> next, V value) {
        this.key = key;
        this.hash = hash;
        this.next = next;
        this.value = value;
    }

    @SuppressWarnings("unchecked")
    static final <K,V> HashEntry<K,V>[] newArray(int i) {
    return new HashEntry[i];
    }
}
```

- 在HashEntry类中，key，hash和next域都被声明为**final**的，value域被volatile所修饰，因此HashEntry对象几乎是不可变的，这是ConcurrentHashmap**读操作并不需要加锁**的一个重要原因
- 由于value域被**volatile**修饰，所以其可以确保被读线程读到最新的值，这是ConcurrentHashmap**读操作并不需要加锁**的另一个重要原因

put(), get()

- 不允许key值为null，也不允许value值为null
- **HashTable 和由同步包装器包装的HashMap每次只能有一个线程执行读或写操作**，ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下，ConcurrentHashMap 可

以支持 **16 个线程**执行**并发写操作**（如果并发级别设置为 16），及**任意数量线程的读操作。**

重哈希rehash()

- ConcurrentHashMap的重哈希实际上是对ConcurrentHashMap的某个段的重哈希，因此ConcurrentHashMap的每个段所包含的桶位自然也就不尽相同

存在key-value为null的特殊情况

```
V get(Object key, int hash) {
        if (count != 0) {            // read-volatile，首先读 count 变量
            HashEntry<K,V> e = getFirst(hash);   // 获取桶中链表头结点
            while (e != null) {
                if (e.hash == hash && key.equals(e.key)) {    // 查找链中是否存在指定Key的键值对
                    V v = e.value;
                    if (v != null)  // 如果读到value域不为 null，直接返回
                        return v;
                    // 如果读到value域为null，说明发生了重排序，加锁后重新读取
                    return readValueUnderLock(e); // recheck
                }
                e = e.next;
            }
        }
        return null;  // 如果不存在，直接返回null
    }
```

- 初始化HashEntry时发生的指令重排序导致的，也就是在HashEntry初始化完成之前便返回了它的引用
- 加锁重读

读操作不需要加锁

- 用HashEntery对象的不变性来降低读操作对加锁的需求；
- 用Volatile变量协调读写线程间的内存可见性；
- 若读时发生指令重排序现象，则加锁重读；

结构性操作的并发安全

```
 remove(Object key, int hash, Object value) {
        lock();     // 加锁
        try {
            int c = count - 1;
            HashEntry<K,V>[] tab = table;
            int index = hash & (tab.length - 1);        // 定位桶
            HashEntry<K,V> first = tab[index];
            HashEntry<K,V> e = first;
            while (e != null && (e.hash != hash || !key.equals(e.key)))  // 查找待删除的键值对
                e = e.next;

            V oldValue = null;
            if (e != null) {    // 找到
                V v = e.value;
                if (value == null || value.equals(v)) {
                    oldValue = v;
                    // All entries following removed node can stay
                    // in list, but all preceding ones need to be
```

```
                    // cloned.
                    ++modCount;
                    // 所有处于待删除节点之后的节点原样保留在链表中
                    HashEntry<K,V> newFirst = e.next;
                    // 所有处于待删除节点之前的节点被克隆到新链表中
                    for (HashEntry<K,V> p = first; p != e; p = p.next)
                        newFirst = new HashEntry<K,V>(p.key, p.hash,newFirst,
p.value);

                    tab[index] = newFirst;   // 将删除指定节点并重组后的链重新放到桶中
                    count = c;      // write-volatile，更新Volatile变量count
                }
            }
            return oldValue;
        } finally {
            unlock();          // finally子句解锁
        }
    }
```

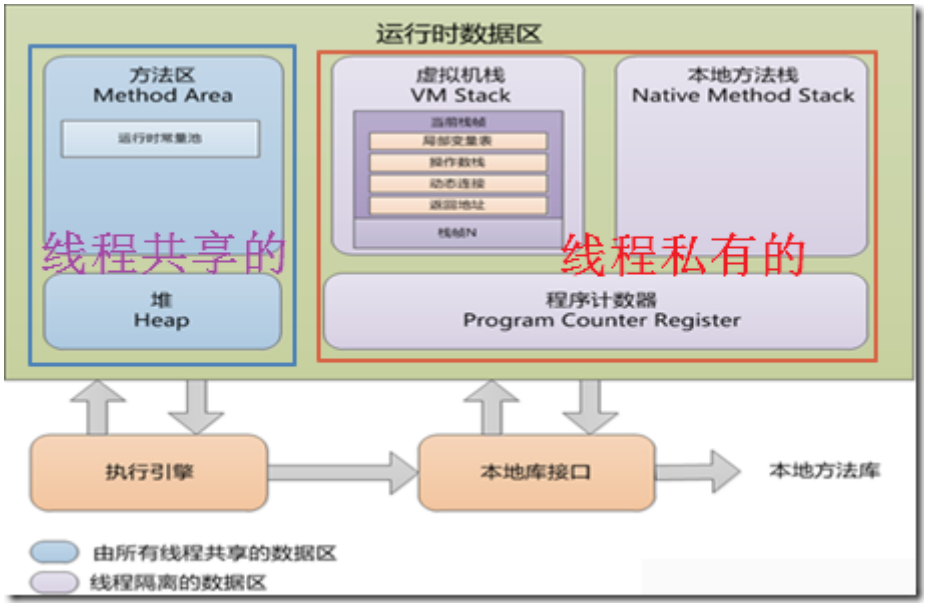
- clear操作只是把ConcurrentHashMap中所有的桶置空，每个桶之前引用的链表依然存在，只是桶不再引用这些链表而已，而链表本身的结构并没有发生任何修改。
- put操作如果需要插入一个新节点到链表中时会在链表头部插入这个新节点，此时链表中的原有节点的链接并没有被修改
- 在执行remove操作时，原始链表并没有被修改
- 只要之前对链表做结构性修改操作的写线程M在退出写方法前写volatile变量count（segment中的，segment中元素的个数），读线程N就能读取到这个volatile变量count的最新值

跨segment操作

- size(): JDK只需要在统计size前后比较modCount（Segment中的）是否发生变化就可以得知容器的大小是否发生变化
- size方法主要思路是先在没有锁的情况下对所有段大小求和，这种求和策略最多执行RETRIES_BEFORE_LOCK次(默认是两次)：在没有达到RETRIES_BEFORE_LOCK之前，求和操作会不断尝试执行（这是因为遍历过程中可能有其它线程正在对已经遍历过的段进行结构性更新）；在超过RETRIES_BEFORE_LOCK之后，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。

JVM内存模型(*)

必考，熟背



- **线程私有的数据区 包括 程序计数器、 虚拟机栈 和 本地方法栈**
- **线程共享的数据区 具体包括 Java堆 和 方法区**

线程计数器

- 在多线程情况下，当线程数超过CPU数量或CPU内核数量时，线程之间就要根据 时间片轮询抢夺CPU时间资源。也就是说，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程都需要一个独立的程序计数器去记录其正在执行的字节码指令地址。

虚拟机栈

- 每个方法从调用直至完成的过程，对应一个栈帧在虚拟机栈中入栈到出栈的过程

本地方法栈

- 本地方法栈与Java虚拟机栈非常相似，也是线程私有的，区别是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈为虚拟机执行 Native 方法服务。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常

Java堆

- Java 堆的唯一目的就是**存放对象实例**，几乎所有的对象实例（和数组）都在这里分配内存
- Java堆可以处于**物理上不连续的内存空间中，只要逻辑上是连续**的即可。而且，Java堆在实现时，既可以是固定大小的，也可以是可拓展的，并且主流虚拟机都是按可扩展来实现的（通过-Xmx(最大堆容量) 和 -Xms(最小堆容量)控制）。如果在堆中没有内存完成实例分配，并且堆也无法再拓展时，将会抛出 OutOfMemoryError 异常。
- TLAB (线程私有分配缓冲区) : 虚拟机为新生对象分配内存时，需要考虑修改指针 (该指针用于划分内存使用空间和空闲空间) 时的线程安全问题，因为存在可能出现正在给对象A分配内存，指针还未修改，对象B又同时使用原来的指针分配内存的情况。TLAB 的存在就是为了解决这个问题：每个线程在Java堆中预先分配一小块内存 TLAB，哪个线程需要分配内存就在自己的TLAB上进行分配，若TLAB用完并分配新的TLAB时，再加同步锁定，这样就大大提升了对象内存分配的效率。

方法区

- 方法区与Java堆一样，也是线程共享的并且不需要连续的内存，其用于**存储已被虚拟机加载的 类信息、常量、静态变量、即时编译器编译后的代码**等数据
- 运行时常量池：是方法区的一部分，用于存放编译期生成的各种 **字面量 和 符号引用**. 字面量比较接近Java语言层次的常量概念，如文本字符串、被声明为final的常量值. 符号引用:包括以下三类常量：类和接口的全限定名、字段的名称和描述符 和 方法的名称和描述符.

方法区的回收

- 主要是针对 常量池的回收 （ 判断引用 ） 和 对类型的卸载
- 回收类: 1) 该类所有的**实例都已经被回收**，也就是Java堆中不存在该类的任何实例加载 2) 该类的**ClassLoader已经被回收** 3) 该类对应的 **java.lang.Class 对象没有在任何地方被引用**，无法在任何地方通过反射访问该类的方法。

垃圾回收机制(*)
必考，熟背

引用计数法

- 循环引用

可达性分析算法

- 通过一系列的名为 “**GC Roots**” 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain）。当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的
- **虚拟机栈(栈帧中的局部变量表)中引用的对象**
- **方法区中类静态属性引用的对象**
- **方法区中常量引用的对象**
- **本地方法栈中Native方法引用的对象**

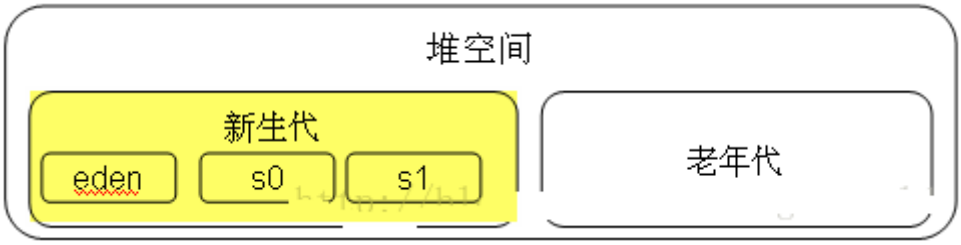
标记清除算法

- 标记-清除算法分为标记和清除两个阶段。该算法首先从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象并进行回收

- 效率问题：标记和清除两个过程的效率都不高;
- 空间问题：标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，因此标记清除之后会产生大量不连续的**内存碎片**，空间碎片太多可能会**导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作**

复制算法

- 复制算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。**这种算法适用于对象存活率低的场景，比如新生代。**这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。
- 实践中会将新生代内存分为**一块较大的Eden空间和两块较小的Survivor空间** (如下图所示)，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是 8:1，也就是每次新生代中可用内存空间为整个新生代容量的90% (80%+10%)，只有10% 的内存会被“浪费”。



- 现在商用的虚拟机都采用这种算法来回收新生代

为什么分代收集

- 不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率.

新生代进入老生代的情况

- 对象优先在Eden分配，当**Eden区没有足够空间进行分配时，虚拟机将发起一次MinorGC。**现在的商业虚拟机一般都采用复制算法来回收新生代，将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。 当进行垃圾回收时，将Eden和Survivor中还存活的对象一次性地复制到另外一块Survivor空间上，最后处理掉Eden和刚才的Survivor空间。（HotSpot虚拟机默认Eden和Survivor的大小比例是8:1）当**Survivor空间不够用时，需要依赖老年代**进行分配担保。
- **大对象直接进入老年代。**所谓的大对象是指，需要大量连续内存空间的java对象，最典型的大对象就是那种很长的字符串以及数组。
- **长期存活的对象(-XX:MaxTenuringThreshold)**将进入老年代。当对象在新生代中经历过一定次数（默认为15）的Minor GC后，就会被晋升到老年代中。
- 动态对象年龄判定。为了更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中**相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代**，无须等到MaxTenuringThreshold中要求的年龄。

内存分配担保机制

- 我们知道如果对象在复制到Survivor区时若Survivor空间不足，则会出发担保机制，将对象转入老年代；但老年代的能力也不是无限的，因此需要在minor GC时做一个是否需要Major GC 的判断：
- 如果老年代的剩余空间 < 之前转入老年代的对象的平均大小，则触发Major GC

- 如果老年代的剩余空间 > 之前转入老年代的对象的平均大小，并且允许担保失败，则直接Minor GC，不需要做Full GC
- 如果老年代的剩余空间 > 之前转入老年代的对象的平均大小，并且不允许担保失败，则触发Major GC

  出发点还是尽量为对象分配内存。但是一般会配置允许担保失败，避免频繁的去做Full GC。

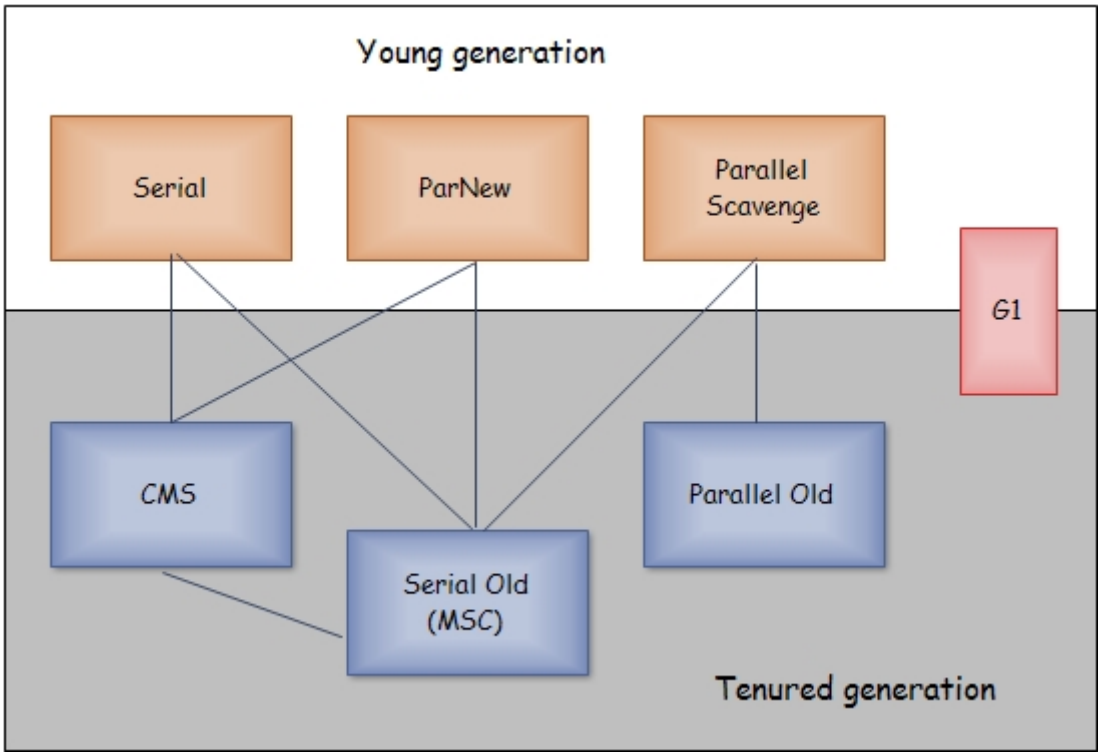
标记整理算法

- 标记整理算法的标记过程类似标记清除算法，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，类似于磁盘整理的过程，该垃圾回收算法**适用于对象存活率高的场景（老年代）**
- **无内存碎片**

新生代、老年代、永久代

- 新生代的目标就是尽可能快速的收集掉那些生命周期短的对象，一般情况下，所有新生成的对象首先都是放在新生代的. 如果老年代也满了，就会触发一次FullGC，也就是新生代、老年代都进行回收。注意，新生代发生的GC也叫做MinorGC，MinorGC发生频率比较高，不一定等 Eden区满了才触发。
- 老年代存放的都是一些生命周期较长的对象，就像上面所叙述的那样，在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中
- 永久代主要用于存放静态文件，如Java类、方法等

垃圾收集器



- Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
- Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；
- ParNew收集器 (复制算法):新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
- Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量=用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
- Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；

- CMS(Concurrent Mark Sweep)收集器（标记-清除算法）：老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
- G1(Garbage First)收集器 (标记-整理算法)：Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

CMS,G1

- https://blog.csdn.net/huanbia/article/details/75581423

内存泄露问题

- **静态集合类:** 如 HashMap、Vector 等集合类的静态使用最容易出现内存泄露，因为这些静态变量的生命周期和应用程序一致，所有的对象Object也不能被释放
- 各种资源连接包括**数据库连接、网络连接、IO连接**等没有显式调用close关闭
- **监听器**的使用，在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。

MYSQL索引(*)

建立索引

- 表的主键、外键必须有索引；
- 数据量超过300的表应该有索引；
- 经常与其他表进行连接的表，在连接字段上应该建立索引；
- 经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；
- 索引应该建在选择性高的字段上；
- 索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；

- 频繁进行数据操作的表，不要建立太多的索引；

索引失效

- **字符串不加单引号**
- 将要使用的索引列**不是复合索引列表中的第一部分**，则不会使用索引
- 应尽量**避免在 where 子句中对字段进行 null 值判断**，否则将导致引擎放弃使用索引而进行全表扫描，如：
  select id from t where num is null
- 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：
  select id from t where num=0
- 应尽量**避免在 where 子句中使用!=或<>操作符**，否则将引擎放弃使用索引而进行全表扫描。优化器将无法通过索引来确定将要命中的行数,因此需要搜索该表的所有行。
- 应尽量**避免在 where 子句中使用 or 来连接条件 (用or分割开的条件，如果or前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到)**，否则将导致引擎放弃使用索引而进行全表扫描，如：
  select id from t where num=10 or num=20
- 可以这样查询：
  select id from t where num=10
  union all
  select id from t where num=20
- **in 和 not in 也要慎用**，因为IN会使系统无法使用索引,而只能直接搜索表中的数据。如：
  select id from t where num in(1,2,3)
- 对于**连续的数值，能用 between 就不要用 in** 了：
  select id from t where num between 1 and 3
- 尽量**避免在索引过的字符数据中，使用非打头字母%搜索**。这也使得引擎无法利用索引。
  见如下例子：
  SELECT * FROM T1 WHERE NAME LIKE '%L%'
  SELECT * FROM T1 WHERE SUBSTING(NAME,2,1)='L'
  SELECT * FROM T1 WHERE NAME LIKE 'L%'
- 即使NAME字段建有索引，前两个查询依然无法利用索引完成加快操作，引擎不得不对全表所有数据逐条操作来完成任务。而第三个查询能够使用索引来加快操作
- 应尽量**避免在 where 子句中对字段进行表达式操作**，这将导致引擎放弃使用索引而进行全表扫描
- 应尽量**避免在where子句中对字段进行函数操作**，这将导致引擎放弃使用索引而进行全表扫描
- 不要在 **where 子句中的"="左边进行函数、算术运算或其他表达式运算**，否则系统将可能无法正确使用索引

共享锁，排他锁

- **InnoDB普通 select 语句默认不加锁**(快照读，MYISAM会加锁)，而CUD操作默认加排他锁
- MySQL InnoDB存储引擎，实现的是基于多版本的并发控制协议——**MVCC** (Multi-Version Concurrency Control) (注：与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。MVCC最大的好处，相信也是耳熟能详：**读不加锁**，**读写不冲突**。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。
- 多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 这样在读操作不用阻塞写操作，写操作不用阻塞读操作的同时，避免了脏读和不可重复读.MVCC 在语境中倾向于 "对多行数据打快照造平行宇宙"，然而 CAS 一般只是保护单行数据而已
- 在MVCC并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

- SELECT ... LOCK IN SHARE MODE ：**共享锁(**S锁, share locks)。其他事务可以读取数据，但不能对该数据进行修改，直到所有的共享锁被释放。
- SELECT ... FOR UPDATE：**排他锁**(X锁, exclusive locks)。如果事务对数据加上排他锁之后，则其他事务不能对该数据加任何的锁。获取排他锁的事务既能读取数据，也能修改数据。
- InnoDB**默认隔离级别 可重复读**(Repeated Read)
- 查询字段未加索引（主键索引、普通索引等）时，使用表锁
- InnoDB行级锁基于索引实现
- **索引数据重复率太高会导致全表扫描**：当表中索引字段数据重复率太高，则MySQL可能会忽略索引，进行全表扫描，此时使用表锁。可使用 force index 强制使用索引。

隔离级别

- **Read Uncommitted**（读取未提交内容）: 在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。
- **Read Committed**（读取提交内容）: 这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。
- **Repeatable Read**（可重读）: 这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。
- **Serializable**（可串行化): 这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争.

Spring IOC 怎么注入类，怎么实例化对象
实例化

- Spring IoC容器则需要**根据Bean定义里的配置元数据使用反射机制来创建Bean**
- 使用**构造器**实例化Bean 有参/无参;使用**静态工厂**实例化Bean;使用**实例工厂**实例化Bean.
- 使用@Autowire注解注入的时机则是容器刚启动的时候就开始注入；注入之前要先初始化bean；ApplicationContext 的初始化和BeanFactory 有一个重大的区别：BeanFactory在初始化容器时，并未实例化Bean，直到第一次访问某个Bean 时才实例目标Bean；而ApplicationContext 则在初始化应用上下文时就实例化所有单实例的Bean。

注入

- 接口、setter、构造器

AOP(*)
动态代理

```
@Aspect
public class Audience
{
    @Before("execution(** concert.Performance.perform(..))")         // 表演之前
    public void silenceCellPhones()
    {
        System.out.println("Silencing cell phones");
    }
    @Before("execution(** concert.Performance.perform(..))")         // 表演之前
    public void takeSeats()
    {
```

```
        System.out.println("Taking seats");
    }
    @AfterReturning("execution(** concert.Performance.perform(..))")        //
表演之后
    public void applause()
    {
        System.out.println("CLAP CLAP CLAP!!!");
    }
    @AfterThrowing("execution(** concert.Performance.perform(..))")        // 表
演失败之后
    public void demandRefound()
    {
        System.out.println("Demanding a refund");
    }
}
```

- **JDK动态代理，接口，用Proxy.newProxyInstance生成代理对象，InvocationHandler**
- **CGLIB，类，用enhancer生成代理对象，MethodInteceptor**
- **如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP；**
  如果目标对象实现了接口，可以强制使用CGLIB实现AOP；
  如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换;
- 切点+注解
- AspectJ是一个比较牛逼的AOP框架，他可以对类的成员变量，方法进行拦截。由于 AspectJ 是 Java 语言语法和语义的扩展，所以它提供了自己的一套处理方面的关键字。除了包含字段和方法之外，AspectJ 的方面声明还包含切入点和通知成员。
  Spring AOP依赖的是 Spring 框架方便的、最小化的运行时配置，所以不需要独立的启动器。但是，使用这个技术，只能通知从 Spring 框架检索出的对象。Spring的AOP技术只能是对方法进行拦截。
  在spring AOP中我们同样也可以使用类似AspectJ的注解来实现AOP功能，但是这里要注意一下，使AspectJ的注解时，AOP的实现方式还是Spring AOP。Spring缺省使用J2SE动态代理来作为AOP的代理，这样任何接口都可以被代理，Spring也可以使用CGLIB代理，对于需要代理类而不是代理接口的时候CGLIB是很有必要的。如果一个业务对象没有实现接口，默认就会使用CGLIB代理。
  Spring AOP和AscpectJ之间的关系：Spring**使用**了和aspectj**一样的注解**，并使用Aspectj来做切入点解析和匹配。但是**spring AOP运行时仍旧是纯的spring AOP**,并不依赖于Aspectj的编译器或者织入器

volatile和内存模型(*)

happens-before

- 什么是happens-before
  令A和B表示两组操作，如果A happens-before B，那么由A操作引起的内存变化，在B开始执行之前，都应该是可见的。
  A happens-before B，不代表A在B之前执行.
- 如何确保happen-before
  锁（互斥锁、读写锁等）、内存屏障

内存屏障

- 内存屏障是一个指令，这个指令可以保证屏障前后的指令遵守一定的顺序，并且保证一定的可见性
- 为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入*内存屏障*来禁止特定类型的处理器重排序。

Java内存模型

- 屏蔽各个硬件平台和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果
- Java内存模型 规定**所有的变量都是存在主存**当中（类似于前面说的物理内存），每个**线程都有自己的工作内存**（类似于前面的高速缓存）。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。

原子性

- 只有简单的**读取、赋值**（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是**原子操作**
- Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过 synchronized 和 Lock 来实现

可见性

- **当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值**. 通过 **synchronized 和 Lock 也能够保证可见性**，synchronized 和 Lock 能保证同一时刻只有一个线程获取锁然后执行同步代码，并且 在释放锁之前会将对变量的修改刷新到主存当中，因此可以保证可见性

有序性

- 指令重排序
- 不能由于 synchronized 和 Lock 可以让线程串行执行同步代码，就说它们可以保证指令不会发生重排序

volatile

- 保证了不同线程对共享变量进行操作时的**可见性**，即一个线程修改了某个变量的值，这个新值对其他线程来说是 立即可见
- **禁止进行指令重排序** (双重检查锁单例模式)
- **synchronized 也可以保证可见性**，因为每次运行synchronized块 或者 synchronized方法都会导致线程工作内存与主存的同步，使得其他线程可以取得共享变量的最新值。也就是说，synchronized 语义范围不但包括 volatile 具有的可见性，也包括原子性，**但不能禁止指令重排序**，这是二者一个功能上的差异

i被volatile修饰，如果多线程来运行i++，那么是否可以达到理想的效果?

- 不能，volatile不能保证操作的原子性

Sleep()和wait()的区别，使用wait()方法后，怎么唤醒线程(*)

笔试题经常考

- **sleep方法只让出了CPU，而并不会释放同步资源锁**
- **wait()方法则是指当前线程让自己暂时退让出同步资源锁**，以便其他正在等待该资源的线程得到该资源进而运行
- sleep()方法可以在任何地方使用；wait()方法则只能在同步方法或同步块中使用
- sleep()是线程线程类（Thread）的方法，调用会暂停此线程指定的时间，但监控依然保持，不会释放对象锁，到时间自动恢复；wait()是Object的方法，调用会放弃对象锁，进入等待队列，待**调用notify()/notifyAll()唤醒指定的线程或者所有线程**，才会进入锁池，不再次获得对象锁才会进入运行状态
- notify让之前调用wait的线程有权利重新参与线程的调度

Mybatis缓存(*)

- **一级缓存的作用域是同一个SqlSession**，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis**默认开启一级缓存。**
- **二级缓存是mapper级别的缓存**，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession去操作数据库得到数据会存在二级缓存区域，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存
- https://segmentfault.com/a/1190000013678579

Redis的数据结构(*)

- String, Hash, List, Set, ZSet

Hash底层结构

- redis的哈希对象的底层存储可以使用ziplist（压缩列表）和hashtable

Redis缓存怎么运行的？

- 使用ANSI C编写的开源、支持网络、基于内存、可选持久性的键值对存储数据库
- 主从复制
- 哨兵模式

持久化

- 快照文件
- AOF语句追加

过期策略

- https://blog.csdn.net/xiangnan129/article/details/54928672

反向代理是什么？

- 反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。客户端只会得知反向代理的IP地址，而不知道在代理服务器后面的服务器簇的存在.

负载均衡是什么？

- 负载平衡（Load balancing）是一种计算机技术，用来在多个计算机（计算机集群）、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到最优化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。 使用带有负载平衡的多个服务器组件，取代单一的组件，可以通过冗余提高可靠性。负载平衡服务通常是由专用软件和硬件来完成。 主要作用是将大量作业合理地分摊到多个操作单元上进行执行,用于解决互联网架构中的高并发和高可用的问题。

单例模式(*)

必考，静态内部类，双重检查锁至少会写一个

- 私有的构造方法；
  指向自己实例的私有静态引用；
  以自己实例为返回值的静态的公有方法。

双重检查锁

```
public class Singleton2 {

private volatile static Singleton2 singleton2;

private Singleton2() {
}

public static Singleton2 getSingleton2() {

    if (singleton2 == null) {
        synchronized (Singleton2.class) {
            if (singleton2 == null) {
                singleton2 = new Singleton2();
            }
        }
    }
    return singleton2;
}
}
```

- 第一个if (instance == null)，只有instance为null的时候，才进入synchronized.
  第二个if (instance == null)，是为了防止可能出现多个实例的情况。
- volatile: 主要在于singleton = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。

1. 给 singleton 分配内存
2. 调用 Singleton 的构造函数来初始化成员变量，形成实例
3. 将singleton对象指向分配的内存空间（执行完这步 singleton才是非 null 了）但是在 JVM 的即时编译器中存在指令重排序的优化。
   也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了
   ，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

静态内部类

```
public class Singleton1 {

private Singleton1() {
}

public static final Singleton1 getSingleton1() {
    return Singleton1Holder.singleton1;
}

private static class Singleton1Holder {
    private static final Singleton1 singleton1 = new Singleton1();
}
}
```

ThreadLocal内存泄露？

```
static class ThreadLocalMap {
    /**
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
```

```
     * ThreadLocal object).  Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table.  Such entries are referred to
     * as "stale entries" in the code that follows.
     */
    static class Entry extends WeakReference<ThreadLocal> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal k, Object v) {
            super(k);
            value = v;
        }
    }
    ...
}
```

- ThreadLocalMap里面对Key的引用是弱引用。那么，就存在这样的情况：当释放掉对threadlocal对象的强引用后，map里面的value没有被回收，但却永远不会被访问到了，因此ThreadLocal存在着内存泄露问题
- Java为了最小化减少内存泄露的可能性和影响，在ThreadLocal进行get、set操作时会清除线程Map里所有key为null的value。所以最怕的情况就是，ThreadLocal对象设null了，开始发生“内存泄露”，然后使用线程池，线程结束后被放回线程池中而不销毁，那么如果这个线程一直不被使用或者分配使用了又不再调用get/set方法，那么这个期间就会发生真正的内存泄露。因此，最好的做法是：在不使用该ThreadLocal对象时，及时调用该对象的remove方法去移除ThreadLocal.ThreadLocalMap中的对应Entry.

线程死锁检测工具？

- Jconsole, Jstack, visualVM

线程池调优？

- 设置最大线程数，防止线程资源耗尽；
- 使用有界队列，从而增加系统的稳定性和预警能力(饱和策略)；
- 根据任务的性质设置线程池大小：CPU密集型任务(CPU个数个线程)，IO密集型任务(CPU个数两倍的线程)，混合型任务(拆分)。

几种锁？

- 无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率

偏向锁

- 偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护
- 偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁

自旋锁

- 线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作. 所谓“自旋”，就是让线程去执行一个无意义的循环，循环结束后再去重新竞争锁，如果竞争不到继续循环，循环过程中线程会一直处于running状态，但是基于JVM的线程调度，会出让时间片，所以其他线程依旧有申请锁和释放锁的机会。

- 自旋锁省去了阻塞锁的时间空间（队列的维护等）开销，但是长时间自旋就变成了“忙式等待”，忙式等待显然还不如阻塞锁。所以自旋的次数一般控制在一个范围内，例如10,100等，在超出这个范围后，自旋锁会升级为阻塞锁。

轻量级锁

- 线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋获取锁，当自旋获取锁仍然失败时，表示存在其他线程竞争锁(两条或两条以上的线程竞争同一个锁)，则轻量级锁会膨胀成重量级锁。

重量级锁

- 重量锁在JVM中又叫对象监视器（Monitor），它很像C中的Mutex，除了具备Mutex(0|1)互斥的功能，它还负责实现了Semaphore(信号量)的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列（wait队列），前者负责做互斥，后一个用于做线程同步。

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程使用自旋会消耗CPU	追求响应时间,锁占用时间很短
重量级锁	线程竞争不使用自旋，不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量,锁占用时间较长

innoDB和MyISAM的区别? (*)

- https://www.jianshu.com/p/a957b18ba40d
- InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；
- InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
- InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
- InnoDB不保存表的具体行数，执行select count(*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；
- Innodb不支持全文索引，而MyISAM支持全文索引，查询效率上MyISAM要高；

索引失效 (*)

- https://blog.csdn.net/qq_32331073/article/details/79041232

MySQL 索引实现原理+几种索引 (*)

普通索引

- B+ree
- MyISAM的B+Tree的叶子节点上的data，并不是数据本身，而是数据存放的地址。主索引和辅助索引没啥区别，只是主索引中的key一定得是唯一的。这里的索引都是非聚簇索引.

InnoDB

- InnoDB 的数据文件本身就是索引文件，B+Tree的叶子节点上的data就是数据本身，key为主键，这是聚簇索引。
- 因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以 唯一 标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。
- 聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引（普通索引）搜索需要 检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录.

几种索引

- 主键索引；
- 唯一索引；
- 普通索引；
- 联合索引；
- 全文索引。

辅助索引

- https://www.cnblogs.com/xiangyangzhu/p/index.html

为什么用B+树

- https://blog.csdn.net/xlgen157387/article/details/79450295
- 在MySQL中的数据一般是放在磁盘中的，读取数据的时候肯定会有访问磁盘的操作，磁盘中有两个机械运动的部分，分别是盘片旋转和磁臂移动。盘片旋转就是我们市面上所提到的多少转每分钟，而磁盘移动则是在盘片旋转到指定位置以后，移动磁臂后开始进行数据的读写。那么这就存在一个定位到磁盘中的块的过程，而定位是磁盘的存取中花费时间比较大的一块，毕竟机械运动花费的时候要远远大于电子运动的时间。当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程，但是我们可以通过B树进行优化，提高磁盘读取时定位的效率。
- 为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息，保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的I/O操作也少一些，而且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定的。
- 总的来说，B/B+树是为了磁盘或其它存储设备而设计的一种平衡多路查找树(相对于二叉，B树每个内节点有多个分支)，与红黑树相比，在相同的的节点的情况下，一颗B/B+树的高度远远小于红黑树的高度(在下面B/B+树的性能分析中会提到)。B/B+树上操作的时间通常由存取磁盘的时间和CPU计算时间这两部分构成，而CPU的速度非常快，所以B树的操作效率取决于访问磁盘的次数，关键字总数相同的情况下B树的高度越小，磁盘I/O所花的时间越少。

B+树的插入删除

- https://www.cnblogs.com/nullzx/p/8729425.html

为什么说B+树比B树更适合数据库索引

- B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。
- B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
- 由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

JVM内存配置参数

- -Xmx Java Heap最大值，默认值为物理内存的1/4，最佳设值应该视物理内存大小及计算机内其他内存开销而定；
- -Xms Java Heap初始值，Server端JVM最好将-Xms和-Xmx设为相同值，开发测试机JVM可以保留默认值；
- -Xmn Java Heap Young区大小，不熟悉最好保留默认值；
- -Xss 每个线程的Stack大小，不熟悉最好保留默认值；

extends 抽象类和 interface 区别 (*)

- 接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象的。
- 接口中的方法定义默认为 public abstract 类型，接口中的成员变量类型默认为 public static final

- 抽象类可以有构造方法，接口中不能有构造方法。
- 抽象类中可以有普通成员变量，接口中没有普通成员变量。
- 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。
- 抽象类中的抽象方法的访问类型可以是public，protected，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。
- 抽象类中可以包含静态(static)方法，接口中不能包含静态(static)方法。
- 抽象类和接口中都可以包含静态成员变量(static)，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。
- 一个类只能继承一个抽象类，但是可以实现多个接口。
- 一个接口可以继承多个接口。
- 抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a"关系关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的，仅仅是实现了接口定义的契约而已，是"like-a"的关系。

Servlet生命周期

- 调用 init() 方法初始化
- 调用 service() 方法来处理客户端的请求
- 调用 destroy() 方法释放资源，标记自身为可回收
- 被垃圾回收器回收

Cookie, Session区别

- cookie数据存放在客户的浏览器上，session数据放在服务器上
- cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗,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。当访问增多，会比较占用你服务器的性能,考虑到减轻服务器性能方面，应当使用COOKIE
- 单个cookie在客户端的限制是3K，就是说一个站点在客户端存放的COOKIE不能3K。

非对称加密，对称加密
对称加密（Symmetric Cryptography），又称私钥加密

- 对称加密是最快速、最简单的一种加密方式，加密（encryption）与解密（decryption）用的是同样的密钥（secret
  key）,这种方法在密码学中叫做对称加密算法。对称加密有很多种算法，由于它效率很高，所以被广泛使用在很多加密协议的核心当中。对称加密通常使用的是相对较小的密钥，一般小于256 bit。因为密钥越大，加密越强，但加密与解密的过程越慢。如果你只用1 bit来做这个密钥，那黑客们可以先试着用0来解密，不行的话就再用1解；但如果你的密钥有1 MB大，黑客们可能永远也无法破解，但加密和解密的过程要花费很长的时间。密钥的大小既要照顾到安全性，也要照顾到效率，是一个trade-off。

非对称加密（Asymmetric Cryptography），又称公钥加密

- 1976年，美国学者Dime和Henman为解决信息公开传送和密钥管理问题，提出一种新的密钥交换协议，允许在不安全的媒体上的通讯双方交换信息，安全地达成一致的密钥，这就是"公开密钥系统"。相对于"对称加密算法"这种方法也叫做"非对称加密算法"。非对称加密为数据的加密与解密提供了一个非常安全的方法，它使用了一对密钥，公钥（public key）和私钥（private key）。私钥只能由一方安全保管，不能外泄，而公钥则可以发给任何请求它的人。非对称加密使用这对密钥中的一个进行加密，而解密则需要另一个密钥。比如，你向银行请求公钥，银行将公钥发给你，你使用公钥对消息加密，那么只有私钥的持有人--银行才能对你的消息解密。与对称加密不同的是，银行不需要将私钥通过网络发送出去，因此安全性大大提高。

目前通信安全的方法

- **将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。**

Http, Https 区别 (*)

- Http协议运行在TCP之上，明文传输，客户端与服务器端都无法验证对方的身份；Https是身披SSL(Secure Socket Layer)外壳的Http，运行于SSL上，SSL运行于TCP之上，是添加了加密和认证机制的HTTP。

二者之间存在如下不同：

- 端口不同：Http与Http使用不同的连接方式，用的端口也不一样，前者是80，后者是443；
- 资源消耗：和HTTP通信相比，Https通信会由于加减密处理消耗更多的CPU和内存资源；
- 开销：Https通信需要证书，而证书一般需要向认证机构购买；
- Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。SSL协议是通过非对称密钥机制保证双方身份认证，并完成建立连接，在实际数据通信时通过对称密钥机制保障数据安全性.

长连接，短连接

- http://www.cnblogs.com/0201zcr/p/4694945.html
- connection:keep-alive

短连接

- 模拟一下TCP短连接的情况:
  client 向 server 发起连接请求
  server 接到请求，双方建立连接
  client 向 server 发送消息
  server 回应 client
  一次读写完成，此时双方任何一个都可以发起 close 操作
  在第 5 点钟，一般都是 client 先发起 close 操作。因为一般的 server 不会回复完 client 后就立即关闭连接。
  当然也不排除有特殊的情况。
  从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作
- 管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段

长连接

- 再模拟一下长连接的情况:
  client 向 server 发起连接
  server 接到请求，双方建立连接
  client 向 server 发送消息
  server 回应 client
  一次读写完成，连接不关闭
  后续读写操作...
  需要TCP保活功能。

应用场景

- 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况。 每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，
  再操作的话那么处理速度会降低很多，所以每个操作完后都不断开， 再次处理时直接发送数据包就OK了，不用建立TCP连接。
  例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误， 而且频繁的socket创建也是对资源的浪费。
- 而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，

如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

三次握手，四次挥手(*)
常问问题, 熟背

为什么是三次握手不是两次握手

- 在只有两次“握手”的情形下，假设Client想跟Server建立连接，但是却因为中途连接请求的数据报丢失了，故Client端不得不重新发送一遍；这个时候Server端仅收到一个连接请求，因此可以正常的建立连接。但是，有时候Client端重新发送请求不是因为数据报丢失了，而是有可能数据传输过程因为网络并发量很大在某结点被阻塞了，这种情形下Server端将先后收到2次请求，并持续等待两个Client请求向他发送数据...问题就在这里，Cient端实际上只有一次请求，而Server端却有2个响应，极端的情况可能由于Client端多次重新发送请求数据而导致Server端最后建立了N多个响应在等待，因而造成极大的资源浪费

为什么是四次挥手

- 双向通信
- 假如现在你是客户端你想断开跟Server的所有连接该怎么做？第一步，你自己先停止向Server端发送数据，并等待Server的回复。但事情还没有完，虽然你自身不往Server发送数据了，但是因为你们之前已经建立好平等的连接了，所以此时他也有主动权向你发送数据；故Server端还得终止主动向你发送数据，并等待你的确认

GET POST区别

- 在客户端， Get 方式在通过 URL 提交数据，数据 在URL中可以看到；POST方式，数据放置在 HTML HEADER内提交。
- GET方式提交的数据最多只能有1024字节，而POST则没有此限制。
- 安全性问题。使用 Get 的时候，参数会显示在地址栏上，而 Post 不会。所以，如果这些数据是中文数据而且是非敏感数据，那么使用 get ；如果用户输入的数据不是中文字符而且包含敏感数据，那么还是使用 post 为好。
- 安全的和幂等的。所谓安全的意味着该操作用于获取信息而非修改信息。幂等的意味着对同一 URL 的多个请求应该返回同样的结果。完整的定义并不像看起来那样严格。换句话说， GET 请求一般不应产生副作用。从根本上讲，其目标是当用户打开一个链接时，她可以确信从自身的角度来看没有改变资源。比如，新闻站点的头版不断更新。虽然第二次请求会返回不同的一批新闻，该操作仍然被认为是安全的和幂等的，因为它总是返回当前的新闻。反之亦然。POST 请求就不那么轻松了。 POST 表示可能改变服务器上的资源的请求。仍然以新闻站点为例，读者对文章的注解应该通过 POST 请求实现，因为在注解提交之后站点已经不同了（比方说文章下面出现一条注解）。

TCP UDP区别 (*)

- TCP是面向连接的，UDP是无连接的；
- TCP是可靠的，UDP是不可靠的；
- TCP只支持点对点通信，UDP支持一对一、一对多、多对一、多对多的通信模式；
- TCP是面向字节流的，UDP是面向报文的；
- TCP有拥塞控制机制;UDP没有拥塞控制，适合媒体通信；
- TCP首部开销(20个字节)比UDP的首部开销(8个字节)要大；

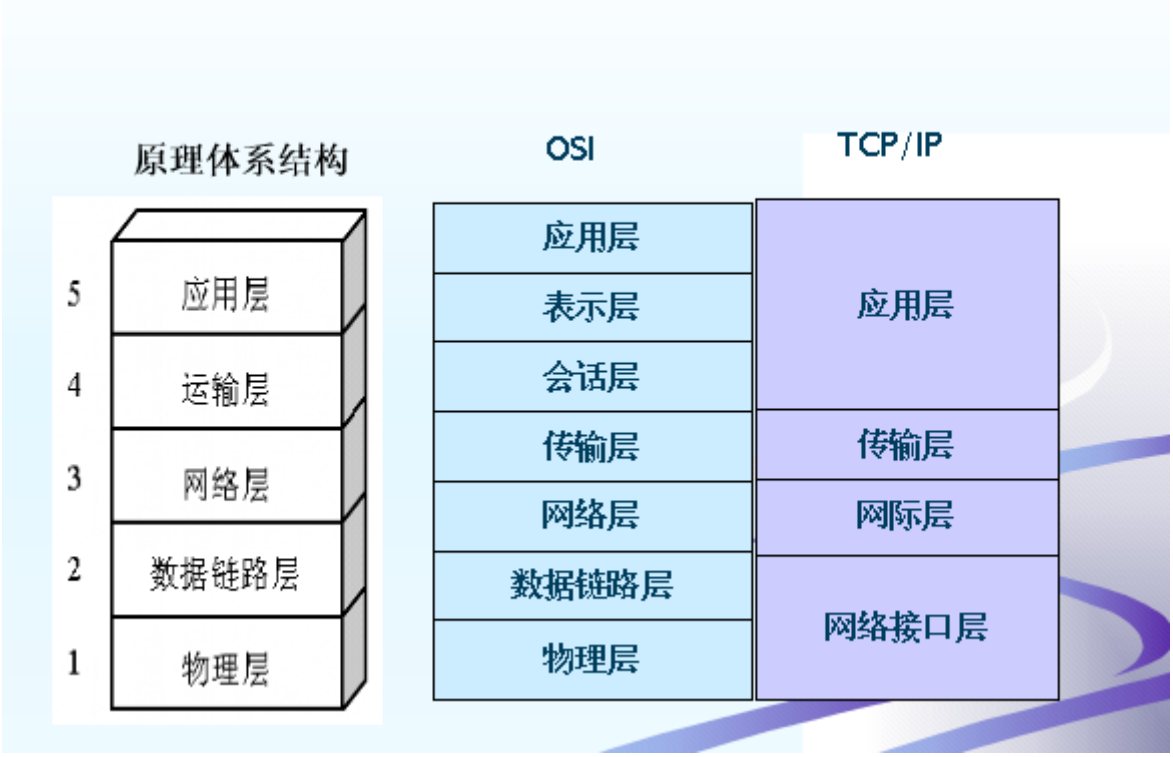
从输入网址到获得页面的过程

- (1). 浏览器查询DNS，获取域名对应的IP地址:具体过程包括浏览器搜索自身的DNS缓存、搜索操作系统的DNS缓存、读取本地的Host文件和向本地DNS服务器进行查询等。对于向本地DNS服务器进行查询，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析(此解析具有权威性)；如果要查询的域名不由本地DNS服务器区域解析，但该服务器已缓存了此

网址映射关系，则调用这个IP地址映射，完成域名解析（此解析不具有权威性）。如果本地域名服务器并未缓存该网址映射关系，那么将根据其设置发起递归查询或者迭代查询；

- (2). 浏览器获得域名对应的IP地址以后，浏览器向服务器请求建立链接，发起三次握手；
- (3). TCP/IP链接建立起来后，浏览器向服务器发送HTTP请求；
- (4). 服务器接收到这个请求，并根据路径参数映射到特定的请求处理器进行处理，并将处理结果及相应的视图返回给浏览器；
- (5). 浏览器解析并渲染视图，若遇到对js文件、css文件及图片等静态资源的引用，则重复上述步骤并向服务器请求这些资源；
- (6). 浏览器根据其请求到的资源、数据渲染页面，最终向用户呈现一个完整的页面。

OSI网络体系结构与TCP/IP协议模型 (*)

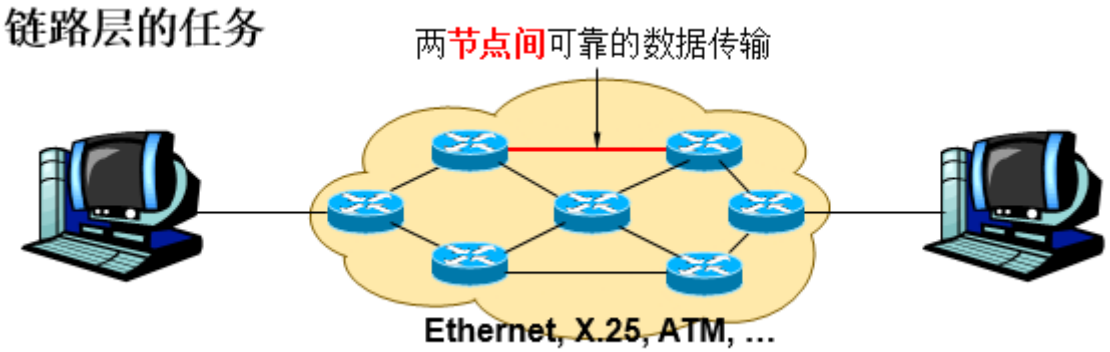


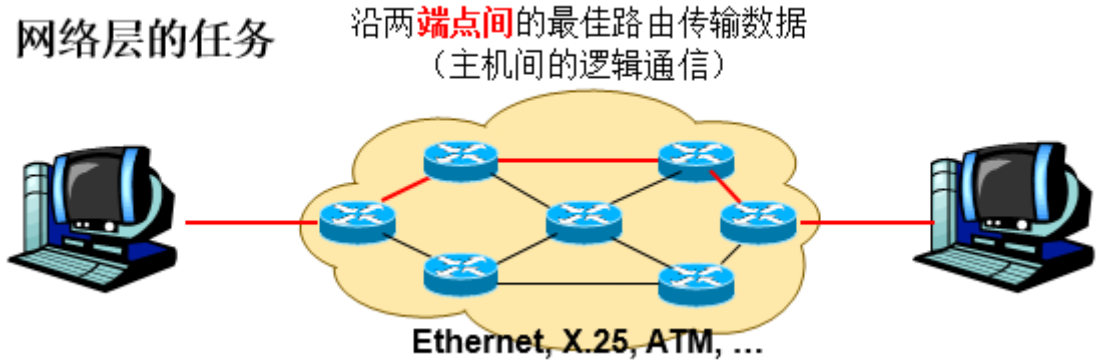
物理层

- 实现了相邻计算机节点之间比特流的透明传送，并尽可能地屏蔽掉具体传输介质和物理设备的差异，使其上层(数据链路层)不必关心网络的具体传输介质

数据链路层

- 接收来自物理层的位流形式的数据，并封装成帧，传送到上一层；同样，也将来自上层的数据帧，拆装为位流形式的数据转发到物理层。这一层在物理层提供的比特流的基础上，通过差错控制、流量控制方法，使有差错的物理线路变为无差错的数据链路，即提供可靠的通过物理介质传输数据的方法。

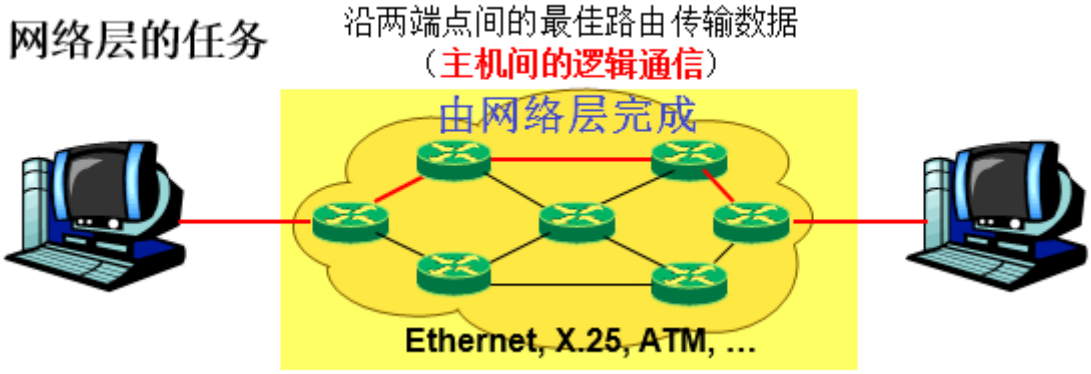


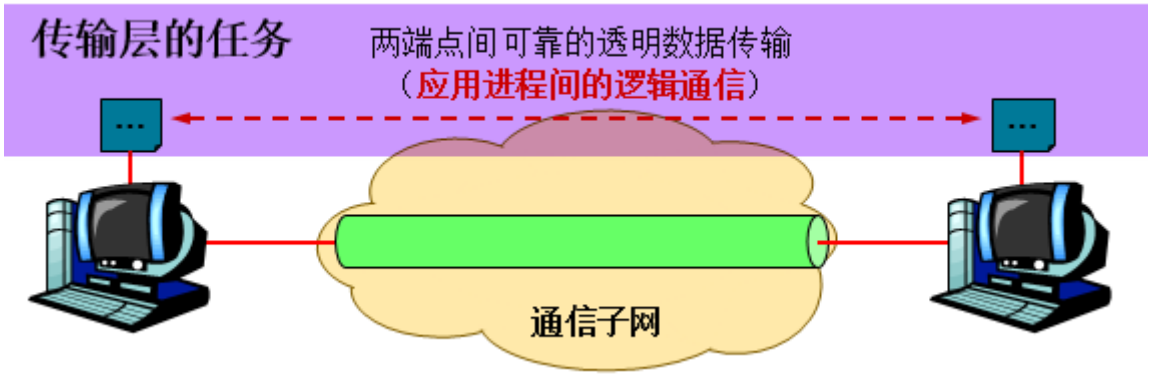


网络层

- 将网络地址翻译成对应的物理地址，并通过路由选择算法为分组通过通信子网选择最适当的路径.

传输层





- 在源端与目的端之间提供可靠的透明数据传输，使上层服务用户不必关系通信子网的实现细节。在协议栈中，传输层位于网络层之上，传输层协议为不同主机上运行的进程提供逻辑通信，而网络层协议为不同主机提供逻辑通信，如下图所示.

会话层

- 会话层是OSI模型的第五层，是用户应用程序和网络之间的接口，负责在网络中的两节点之间建立、维持和终止通信.

表示层

- 数据的编码，压缩和解压缩，数据的加密和解密.

应用层

- 用户的应用进程提供网络通信服务.

TCP和UDP分别对应的常见应用层协议
TCP

- FTP,
- Telnet,
- SMTP,
- POP3,
- HTTP

UDP

- DNS
- SNMP(简单网络管理协议，使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势)
- TFTP(Trival File Transfer Protocal)：简单文件传输协议，该协议在熟知端口69上使用UDP服务。

网络层的ARP协议工作原理

- **网络层的ARP协议完成了IP地址与物理地址的映射。**首先，每台主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址的对应关系。当源主机需要将一个数据包要发送到目的主机时，会首先检查自己ARP列表中是否存在该IP地址对应的MAC地址：如果有，就直接将数据包发送到这个MAC地址；如果没有，就向本地网段发起一个ARP请求的广播包，查询此目的主机对应的MAC地址。此ARP请求数据包里包括源主机的IP地址、硬件地址、以及目的主机的IP地址。网络中所有的主机收到这个ARP请求后，会检查数据包中的目的IP是否和自己的IP地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的MAC地址和IP地址添加到自己的ARP列表中，如果ARP表中已经存在该IP的信息，则将其覆盖，然后给源主机发送一个ARP响应数据包，告诉对方自己是它需要查找的MAC地址；源主机收到这个ARP响应数据包后，将得到的目的主机的IP地址和MAC地址添加到自己的ARP列表中，并利用此信息开始数据的传输。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

HTTP常见状态码

- 1×× : 请求处理中，请求已被接受，正在处理
- 2×× : 请求成功，请求被成功处理
- 200 OK
- 3×× : 重定向，要完成请求必须进行进一步处理
- 301 : 永久性转移
- 302 ：暂时性转移
- 304 ： 已缓存
- 4×× : 客户端错误，请求不合法
- 400：Bad Request,请求有语法问题
- 403：拒绝请求

- 404：客户端所访问的页面不存在
- 5××：服务器端错误，服务器不能处理合法请求
- 500 ：服务器内部错误
- 503 ： 服务不可用，稍等

HTTP协议是无状态的 和 Connection: keep-alive的区别

- HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。
- 从HTTP/1.1起，默认都开启了Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。
- Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间

属于网络112.10.200.0/21的地址是? (*)
笔试常考

- 前21位为网络地址，后12位为主机地址。112 对应前8位，10对应第二个8位，因此200对应第3个8位又200的二进制表示为1100
  1000 前面已经有了16位，因此11001 是属于网络地址的。000是属于主机地址
  那么，最大的地址为【112（十进制）】【10（十进制）】【11001 111】【11111111】转换为十进制为112.10.207.255 故网络的地址范围为:
  112.10.200.0~112.10.207.255

某一速率为100M的交换机有20个端口，其一个端口上连着一台笔记本电脑，此电脑从迅雷上下载一部1G的电影需要的时间可能是多久？

- 1GB/(100Mb/s ÷8,字节→比特)=81.92 s, 比81s大
- 交换机在同一时刻可进行多个端口对之间的数据传输。每一端口都可视为独立的网段，连接在其上的网络设备独自享有全部的带宽，无须同其他设备竞争使用。

与10.110.12.29 mask 255.255.255.224属于同一网段的主机IP地址是?(*)
笔试常考

- 通俗理解：

  子网掩码为255.255.255.224=255.255.255.**111**00000，也就是讲第四个字节，有三位用来表示子网号（共有000，001，010....111等8个子网号）

  请看题目给的是：10.110.12.29=10.110.12.29=10.110.12.**000**11101，从这里看，题目给出的IP所处子网号在000段，所以跟这个IP在一个网段的IP的前27位是定的：10.110.12.000

  最后5位可以是00001~11110（全0和全1是不行的），也就是讲IP范围为 10.110.12.*000* **00000**~10.110.12.*000* **11110**

数据在计算机网络中的称法？

- 应用层：报文
- 运输层：报文段/用户数据报
- 网际层：IP数据报/分组/包
- 数据链路层：帧
- 物理层：比特流

try catch finally (*)
笔试常见套路题

- https://blog.csdn.net/aaoxue/article/details/8535754

OOM, Out of Memory 错误

常见OOM情况

- Java堆内存溢出，此种情况最常见，一般由于内存泄露或者堆的大小设置不当引起。对于内存泄露，需要通过内存监控软件查找程序中的泄露代码，而堆大小可以通过虚拟机参数-Xms,-Xmx等修改。
- Java永久代溢出，即方法区溢出了，一般出现于大量Class或者jsp页面，或者采用cglib等反射机制的情况，因为上述情况会产生大量的Class信息存储于方法区。此种情况可以通过更改方法区的大小来解决，使用类似-XX:PermSize=64m -XX:MaxPermSize=256m的形式修改。另外，过多的常量尤其是字符串也会导致方法区溢出。
- java.lang.StackOverflowError, 不会抛OOM error，但也是比较常见的Java内存溢出。JAVA虚拟机栈溢出，一般是由于程序中存在死循环或者深度递归调用造成的，栈大小设置太小也会出现此种溢出。可以通过虚拟机参数-Xss来设置栈的大小。
- 静态集合类

常用Linux命令

cd

- cd /root/Docements # 切换到目录/root/Docements
- cd ./path # 切换到当前目录下的path目录中，“.”表示当前目录
- cd ../path # 切换到上层目录中的path目录中，“..”表示上一层目录

ls

- -l ：列出长数据串，包含文件的属性与权限数据等
- -a ：列出全部的文件，连同隐藏文件（开头为.的文件）一起列出来（常用）
- -d ：仅列出目录本身，而不是列出目录的文件数据
- -h ：将文件容量以较易读的方式（GB，kB等）列出来
- -R ：连同子目录的内容一起列出（递归列出），等于该目录下的所有文件都会显示出来

grep
命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来，该命令通常与管道命令一起使用，用于对一些命令的输出进行筛选加工等等

- grep [-acinv] [--color=auto] '查找字符串' filename
- -a ：将binary文件以text文件的方式查找数据
- -c ：计算找到‘查找字符串’的次数
- -i ：忽略大小写的区别，即把大小写视为相同
- -v ：反向选择，即显示出没有‘查找字符串’内容的那一行

find 寻找
find [PATH] [option] [action]
与时间有关的参数：

- -mtime n : n为数字，意思为在n天之前的“一天内”被更改过的文件；
- -mtime +n : 列出在n天之前（不含n天本身）被更改过的文件名；
- -mtime -n : 列出在n天之内（含n天本身）被更改过的文件名；
- -newer file : 列出比file还要新的文件名

cp 复制

- -a ：将文件的特性一起复制
- -p ：连同文件的属性一起复制，而非使用默认方式，与-a相似，常用于备份
- -i ：若目标文件已经存在时，在覆盖时会先询问操作的进行
- -r ：递归持续复制，用于目录的复制行为
- -u ：目标文件与源文件有差异时才会复制

mv 移动

- -f ：force强制的意思，如果目标文件已经存在，不会询问而直接覆盖
- -i ：若目标文件已经存在，就会询问是否覆盖
- -u ：若目标文件已经存在，且比目标文件新，才会更新

rm 删除

- -f ：就是force的意思，忽略不存在的文件，不会出现警告消息
- -i ：互动模式，在删除前会询问用户是否操作
- -r ：递归删除，最常用于目录删除，它是一个非常危险的参数

ps 查看进程

- -A ：所有的进程均显示出来
- -a ：不与terminal有关的所有进程
- -u ：有效用户的相关进程
- -x ：一般与a参数一起使用，可列出较完整的信息
- -l ：较长，较详细地将PID的信息列出
- ps aux # 查看系统所有的进程数据
- ps ax # 查看不与terminal有关的所有进程
- ps -lA # 查看系统所有的进程数据
- ps axjf # 查看连同一部分进程树状态

kill
该命令用于向某个工作（%jobnumber）或者是某个PID（数字）传送一个信号，它通常与ps和jobs命令一起使用，它的基本语法如下：kill -signal PID

- 1：SIGHUP，启动被终止的进程
- 2：SIGINT，相当于输入ctrl+c，中断一个程序的进行
- 9：SIGKILL，强制中断一个进程的进行
- 15：SIGTERM，以正常的结束进程方式来终止进程
- 17：SIGSTOP，相当于输入ctrl+z，暂停一个进程的进行

killall

- -i ：交互式的意思，若需要删除时，会询问用户
- -e ：表示后面接的command name要一致，但command name不能超过15个字符
- -l ：命令名称忽略大小写

例如：

- killall -SIGHUP syslogd # 重新启动syslogd

file
用于判断接在file命令后的文件的基本数据，因为在Linux下文件的类型并不是以后缀为分的

- file filename

例如：
file ./test

tar

- -c ：新建打包文件
- -t ：查看打包文件的内容含有哪些文件名
- -x ：解打包或解压缩的功能，可以搭配-C（大写）指定解压的目录，注意-c,-t,-x不能同时出现在同一条命令中
- -j ：通过bzip2的支持进行压缩/解压缩
- -z ：通过gzip的支持进行压缩/解压缩
- -v ：在压缩/解压缩过程中，将正在处理的文件名显示出来

- -f filename ：filename为要处理的文件
- -C dir ：指定压缩/解压缩的目录dir
  常用tar命令
- 压缩：tar -jcv -f filename.tar.bz2 要被处理的文件或目录名称
- 查询：tar -jtv -f filename.tar.bz2
- 解压：tar -jxv -f filename.tar.bz2 -C 欲解压缩的目录

cat
用于查看文本文件的内容，后接要查看的文件名，通常可用管道与more和less一起使用，从而可以一页页地查看数据

- cat text | less # 查看text文件中的内容 注：这条命令也可以使用less text来代替

chgrp 改变文件所属所属用户组

- chgrp [-R] dirname/filename
- -R ：进行递归的持续对所有文件和子目录更改

例如：

- chgrp users -R ./dir # 递归地把dir目录下中的所有文件和子目录下所有文件的用户组修改为users

chown 改变文件所有者

chmod 改变文件权限

- chmod [-R] xyz 文件或目录
- -R：进行递归的持续更改，即连同子目录下的所有文件都会更改

vim

JVM类加载机制 (*)

面试有概率会问

加载

- 加载是类加载过程中的一个阶段，这个阶段会在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的入口。注意这里不一定非得要从一个Class文件获取，这里既可以从ZIP包中读取（比如从jar包和war包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将JSP文件转换成对应的Class类）。

验证

- 这一阶段的主要目的是为了确保Class文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备

- 准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

  public static int v = 8080;
  实际上变量v在准备阶段过后的初始值为0而不是8080，将v赋值为8080的putstatic指令是程序被编译后，存放于类构造器方法之中，这里我们后面会解释。
  但是注意如果声明为： public static final int v = 8080;
  在编译阶段会为v生成ConstantValue属性，在准备阶段虚拟机会根据ConstantValue属性将v赋值为8080。

解析

- 解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是class文件中的： CONSTANT_Class_info
  CONSTANT_Field_info CONSTANT_Method_info 等类型的常量。

符号引用和直接引用的概念：

- 符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。
  直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

- 初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由JVM主导。到了初始阶段，才开始真正执行类中定义的Java程序代码。
  初始化阶段是执行类构造器方法的过程。方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证方法执行之前，父类的方法已经执行完毕。p.s:
  如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成()方法。

注意以下几种情况不会执行类初始化：

- 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
- 定义对象数组，不会触发该类的初始化。
- 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
- 通过类名获取Class对象，不会触发类的初始化。
- 通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
- 通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

给你一个未知长度的链表,怎么找到中间的那个节点?
快慢指针，快指针走两步，慢指针走一步，快指针到尾了，慢指针走到一半。

类加载器 (*)

面试必考

虚拟机设计团队把加载动作放到JVM外部实现，以便让应用程序决定如何获取所需的类，JVM提供了3种类加载器：

- **启动类加载器**(Bootstrap ClassLoader)：负责加载 JAVA_HOME\lib
  目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。
- **扩展类加载器**(Extension ClassLoader)：负责加载 JAVA_HOME\lib\ext
  目录中的，或通过java.ext.dirs系统变量指定路径中的类库。
- **应用程序类加载器**(Application ClassLoader)：负责加载用户路径（classpath）上的类库。
- JVM通过双亲委派模型进行类的加载，当然我们也可以通过继承java.lang.ClassLoader实现自定义的类加载器。
- 当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。
- 采用**双亲委派**的一个好处是比如加载位于rt.jar包中的类java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就**保证了使用不同的类加载器最终得到的都是同样一个Object对象。**

访问权限

- 对于外部类来说，只有两种修饰，public和默认（default），因为外部类放在包中，只有两种可能，包可见和包不可见。
- 对于内部类来说，可以有所有的修饰，因为内部类放在外部类中，与成员变量的地位一致，所以有四种可能。

Spring bean 范围

- singleton, prototype, request, session, global session

进程间的通信方式，线程间的通信方式
进程

- 管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
- 有名管道 (namedpipe) ： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
- 信号量(semophore)：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
- 消息队列(messagequeue)：
  消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 信号 (sinal)： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
- 共享内存(shared memory)：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
- 套接字(socket)： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

线程

- 锁机制：包括互斥锁、条件变量、读写锁
  - 互斥锁提供了以排他方式防止数据结构被并发修改的方法。
  - 读写锁允许多个线程同时读共享数据，而对写操作是互斥的。
  - 条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。`while+if+volatile变量`
- 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量
- 信号机制(Signal)：类似进程间的信号处理

Java线程

- synchoronized
- wait/notify
- lock condition
- semaphore
- countdownlatch
- cyclicbarrier

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

MyBatis中#{}和${}区别(*)

- #{}是经过预编译的,是安全的,而${}是未经过预编译的,仅仅是取变量的值,是非安全的,存在sql注入。

- 只能 $ {}的情况,order by、like 语句只能用 $ {}了,用#{}会多个' '导致sql语句失效.此外动态拼接sql也要用${}
- #{} 这种取值是编译好SQL语句再取值, ${} 这种是取值以后再去编译SQL语句

重要：接受从用户输出的内容并提供给语句中不变的字符串，这样做是不安全的。这会导致潜在的sql注入攻击，因此你不应该允许用户输入这些字段，或者通常自行转义并检查。

数据库数据不一致的原因

- 数据冗余

如果数据库中存在冗余数据，比如两张表中都存储了用户的地址，在用户的地址发生改变时，如果只更新了一张表中的数据，那么这两张表中就有了不一致的数据。

- 并发控制不当

比如某个订票系统中，两个用户在同一时间订同一张票，如果并发控制不当，可能会导致一张票被两个用户预订的情况。当然这也与元数据的设计有关。

- 故障和错误

如果软硬件发生故障造成数据丢失等情况，也可能引起数据不一致的情况。因此我们需要提供数据库维护和数据恢复的一些措施。

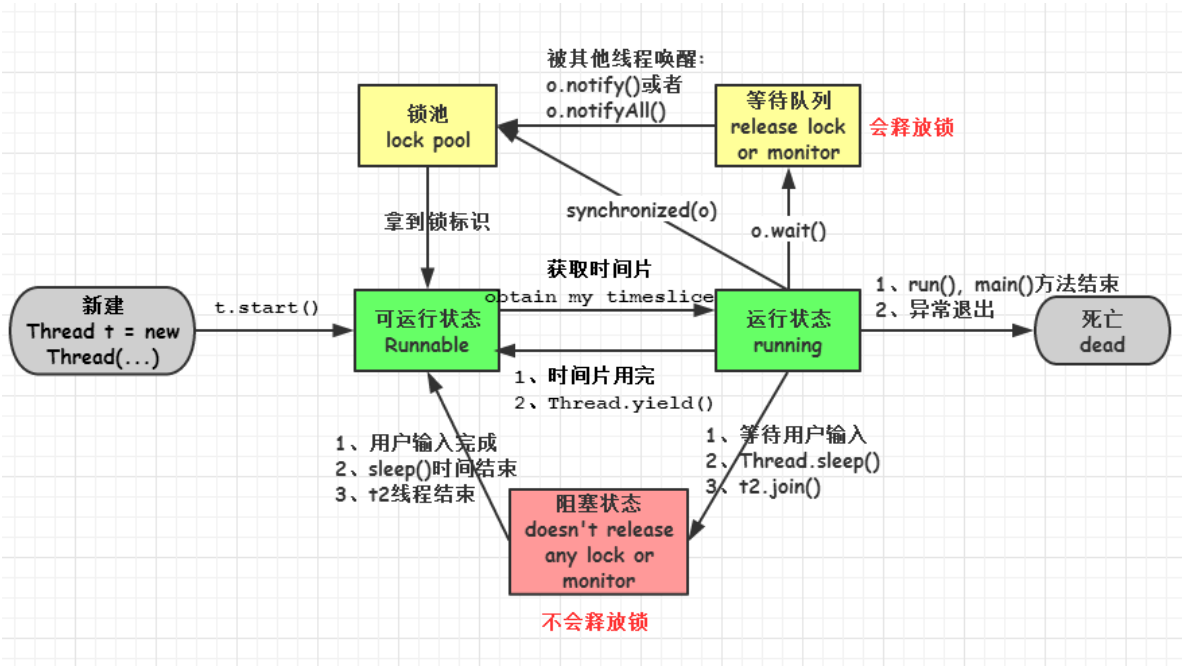
线程的状态(*)

笔试常考

- 新建(new)：新创建了一个线程对象。
- 可运行(runnable)：线程对象创建后，其他线程(比如 main 线程 ）调用了该对象 的 start ()方法。该状态的线程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 。
- 运行(running)：可运行状态(runnable)的线程获得了 cpu 时间片（ timeslice ） ，执行程序代码。

- 阻塞(block)：阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有 机会再次获得 cpu timeslice
  转到运行(running)状态。阻塞的情况分三种：

  (一). 等待阻塞：运行(running)的线程执行 o . wait ()方法， JVM 会把该线程放 入等待队列(waitting queue)中。
  (二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池(lock pool)中。
  (三). 其他阻塞: 运行(running)的线程执行Thread . sleep (long ms)或 t . join ()方法，或者发出了 I / O 请求时， JVM会把该线程置为阻塞. 当 sleep ()状态超时、 join ()等待线程终止或者超时、或者 I / O
  处理完毕时，线程重新转入可运行(runnable)状态。
- 死亡(dead)：线程 run ()、 main () 方法执行结束，或者因异常退出了 run ()方法，则该线程结束生命周期。死亡的线程不可再次复生。



如何确保N个线程可以访问N个资源同时又不导致死锁？

多线程产生死锁需要四个条件，分别是互斥性，保持和请求，不可剥夺性还有要形成闭环，这四个条件缺一不可，只要破坏了其中一个条件就可以破坏死锁，其中最简单的方法就是**线程都是以同样的顺序加锁和释放锁**，也就是破坏了第四个条件。

背包问题(*)

试题描述：小明喜欢在火车旅行的时候用手机听音乐，他有N首歌在手机里，在整个火车途中，他可以听P首歌，所以他想产生一个播放表产生P首歌曲，这个播放表的原则是：

（1）每首歌都要至少被播放一次

（2）在两首一样的歌中间，至少有N首其他的歌

小明想有多少种不同的播放表可以产生，那么给你N、M、P，你来算一下，输出结果取1000000007的余数。

- 定义 value(i,j) 为在背包装载体积为 i,物品为第 1 到 j 个的情况下,背包的最大价值
  value(v,sum(m[i])) 即为所求
  可以把空间复杂度从 O(nv) 降到 O(v)
  状态转移方程:
  value(i,j) = |-> max(value(i-w[j],j-1)+s[j],value(i-1,j-1) (i-w[j] >= 0)
  |-> value(i-1,j-1) (i-w[j] < 0)

进程，线程区别

- 1）含义
  进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,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,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.
- 2）区别
  （1）一个程序至少有一个进程，一个进程至少有一个线程.（2）线程的划分尺度小于进程，使得多线程程序的并发性高。
  （3）进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
  （4）线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程

执行控制。
（5）从逻辑角度来看，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。

int cast到byte(*)
笔试题

```
public class leftshift_operator {
public static void main(String args[]){
```

?

```
    byte x = 64;

    int i;

    byte y;

    i = x<<2;

    y = (byte)(x<<2);

    System.out.print(i+ " " +y);

}
}
```

- 输出 256 0
- 0是因为 x 01000000 被 cast 到 int 32位 00000000000000000000000001000000
  左移2位 00000000000000000000000100000000, 最低8位 00000000

CMS缺点

- CMS收集器对CPU资源非常敏感
  在并发阶段，虽然不会导致用户线程停顿，但是会因为占用了一部分线程使应用程序变慢，总吞吐量会降低，为了解决这种情况，虚拟机提供了一种"增量式并发收集器"
  的CMS收集器变种， 就是在并发标记和并发清除的时候让GC线程和用户线程交替运行，尽量减少GC
  线程独占资源的时间，这样整个垃圾收集的过程会变长，但是对用户程序的影响会减少。（效果不明显，不推荐）
- CMS处理器无法处理浮动垃圾
  CMS在并发清理阶段线程还在运行， 伴随着程序的运行自然也会产生新的垃圾，这一部分垃圾产生在标记过程之后，CMS无法再当次过程中处理，所以只有等到下次gc时候在清理掉，这一部分垃圾就称作"浮动垃圾"
- CMS是基于"标记--清除"算法实现的，所以在收集结束的时候会有大量的空间碎片产生。空间碎片太多的时候，将会给大对象的分配带来很大的麻烦，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象的，只能提前触发full gc。
- 需要更大的堆空间

++操作符和乘除
下列代码执行后的变量num3的值?

```
public static void main(String[] args) {
int num1 = 6, num2 = 7, num3 = 12;

if (++num1 == num2)
    num3 = ++num3 * 3;
System.out.println(num3);
}
```

- 39, 先++num3 ,再乘3

某网络的IP地址空间为10.0.17.0/24，采用等长子网划分，子网掩码为255.255.255.240，则该网络的最大子网个数、每个子网内的最大分配地址个数?(*)
笔试常考

- 240, 11110000
- 2^4=16 最大子网个数
- 2^4-2=14 子网最大分配地址个数

Class, newInstance, 默认构造器

```
public class whatTheHell {
public whatTheHell() {
}

private String a;
private Integer b;

public whatTheHell(String a, Integer b) {
    this.a = a;
    this.b = b;
}

public String getA() {

    return a;
}

public void setA(String a) {
    this.a = a;
}

public Integer getB() {
    return b;
}

public void setB(Integer b) {
    this.b = b;
}

public static void main(String[] args) {
    Class<whatTheHell> whatTheHellClass = whatTheHell.class;
    try {
        whatTheHell whatTheHell = whatTheHellClass.newInstance();
        System.out.println(whatTheHell == null ? true : false);
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
```

```
            e.printStackTrace();
        }

    }
    }
```

- 不提供默认构造器会报错。

若用一个大小为6的数组来实现循环队列，且当前rear和front的值分别0和3。当从队列中删除一个元素，再加入两 个元素后，rear和front的值分别为?

- 2,4
- 删除一个元素后，队首指针要加1，front=（front+1）%6，结果为4，每加入一个元素队尾指针加一，即real=（real+1）%6，加入两个元素后变为2

棵完全二叉树有600个节点，那么它的叶子节点有

- (600+1)/2=300

this, super

- this()函数主要应用于同一类中从某个构造函数调用另一个重载版的构造函数。this()只能用在构造函数中，并且也只能在第一行。所以在同一个构造函数中this()和super()不能同时出现。
- super()函数在子类构造函数中调用父类的构造函数时使用，而且必须要在构造函数的第一行

java中，当实例化子类对象时，如果有以下几个会被加载，那么加载的顺序是什么?(*)
笔试常考

- (1)父类静态代码块
- (2)子类静态代码块
- (3)父类非静态代码块
- (4)父类构造函数
- (5)子类非静态代码块
- (6)子类构造函数

设在一棵度数为3的树中，度数为3的结点数有2个，度数为2的结点数有1个，度数为1的结点数有2个，那么度数为0的结点数有（ ）个?

- 顶点数 = 所有节点度数 +1
- 2+1+2+x =（3x2 + 2x1 + 1x2 + 0 x X）+ 1
- x = 6

forward和redirect区别

- 直接转发方式（Forward），客户端和浏览器只发出一次请求，Servlet、HTML、JSP或其它信息资源，由第二个信息资源响应该请求，在请求对象request中，保存的对象对于每个信息资源是共享的。
- 间接转发方式（Redirect）实际是两次HTTP请求，服务器端在响应第一次请求的时候，让浏览器再向另外一个URL发出请求，从而达到转发的目的。
- redirect 默认302暂时跳转

操作系统，临时抱佛脚

- https://blog.csdn.net/csdn_chai/article/details/78002202
- https://zhuanlan.zhihu.com/p/23755202

前缀（波兰），后缀（逆波兰），中缀表达式

- https://blog.csdn.net/antineutrino/article/details/6763722

BigInteger, BigDecimal

- https://blog.csdn.net/zhongkelee/article/details/52289163

在一个C类地址段内，需要将网络划分为 7个子网，每个子网有15个主机，则可以使用哪个子网掩码？(*)

- 255.255.255.224
- IP地址的结构是：网络号+主机号, 划分子网是从主机号中抽取几位进行子网划分，c类地址前24位为网络号
- 224, 11100000
- 2^3=8>7
- 2^5=32>15

Linux 网络相关命令

- ping是使用的ICMP协议，是IP层协议，但是端口是应用层的，所以它只能判断能够访问ip，不能判断端口
- ifconfig是查看本机的网络设置，IP，子网掩码等
- telnet是应用层的，可以判端口访问情况
- netstat显示网络信息，如网络连接，路由表，接口状态

递归时间复杂度
master 公式
T(N) = a*T(N/b) + O(N^d)

估计递归问题复杂度的通式，只要复杂度符合以下公式，都可以套用此公式计算时间复杂度

例子：递归方式查找数组最大值 T(N) = 2*T(N/2) + O(1)

T(N)：样本量为 N 的情况下，时间复杂度
N：父问题的样本量
a：子问题发生的次数（父问题被拆分成了几个子问题，不需要考虑递归调用，只考虑单层的父子关系）
b：被拆成子问题，子问题的样本量（子问题所需要处理的样本量），比如 N 被拆分成两半，所以子问题样本量为 N/2
O(N^d)：剩余操作的时间复杂度，除去调用子过程之外，剩下问题所需要的代价（常规操作则为 O(1)）

- log(b,a) > d -> 复杂度为O(N^log(b,a))
- log(b,a) = d -> 复杂度为O(N^d * logN)
- log(b,a) < d -> 复杂度为O(N^d)

Java变量命名(*)

- $, _, 字母开头
- 不能是数字开头
- 不能使关键字private, final...

查看linux操作系统磁盘空间命令

- df

设有一个含有13个元素的Hash表(0~12),Hash函数是:H(key)=key % 13,其中%是求余数运算。用线性探查法解决冲突,则对于序列(2、 8 、31、 20、 19、 18、 53、 27),18应放在第几号格中?

- 求出18之前的序列余数为2、8、5、7、6，H(18)=5与之前的冲突，直接向后移，6、7、8都有元素，因此放在9号上

如果一个二叉树中任意节点的左右子树“高度”相差不超过 1，我们称这个二叉树为“高度平衡二叉树”。根据如上定义，一个高度为 8 的高度平衡二叉树至少有几个节点？

- 54
- 类似斐波那契数列的递推公式。Sn = Sn-1 + Sn-2 + 1,初值，S1 =1 ，S2 =2
- 1 2 4 7 12 20 33 54

某公司申请到一个C类IP地址，但要连接6个的子公司，最大的一个子公司有 26台计算机，每个子公司在一个网段中，则子网掩码应设为？(*)

- 11111111.11111111.11111111.?
- 2^3=8, 至少3个1
- 2^5=32, 32-网关-广播=30>26
- 11100000
- 224
- 255.255.255.224

> 随意补充或整理，发布时请注明原作者redfisky, 谢谢