Java研发工程师

求职面试

弱鸡总结

目录

[一、 网络部分 5](#_Toc524034511)

[1. 三次握手与四次挥手 5](#_Toc524034512)

[2. UDP 5](#_Toc524034513)

[3. TCP 5](#_Toc524034514)

[4. HTTP协议 6](#_Toc524034515)

[5. Socket编程 7](#_Toc524034516)

[6. Java NIO 7](#_Toc524034517)

[7. Netty框架 7](#_Toc524034518)

[二、 Java基础 7](#_Toc524034519)

[1. 设计模式 7](#_Toc524034520)

[2. Java多态的体现 7](#_Toc524034521)

[3. Java拆箱装箱 7](#_Toc524034522)

[4. Java泛型 8](#_Toc524034523)

[5. Java序列化 8](#_Toc524034524)

[6. 异常 8](#_Toc524034525)

[7. Collections.sort()和Arrays.sort()排序算法 9](#_Toc524034526)

[8. HashMap 与HashTable的区别 9](#_Toc524034527)

[9. Hashmap扩容 9](#_Toc524034528)

[10. ConcurrentHashMap 9](#_Toc524034529)

[三、 Java Web 12](#_Toc524034530)

[1. Session与Cookie的区别 12](#_Toc524034531)

[2. 请求转发与重定向的区别 12](#_Toc524034532)

[3. Web容器加载 12](#_Toc524034533)

[4. 进程间的通信方式 12](#_Toc524034534)

[四、 并发编程 13](#_Toc524034535)

[1. java.util.concurrent包 13](#_Toc524034536)

[2. Volatile关键字 13](#_Toc524034537)

[3. CAS 14](#_Toc524034538)

[4. AQS 14](#_Toc524034539)

[5. 锁机制 14](#_Toc524034540)

[6. 产生死锁的条件 15](#_Toc524034541)

[7. 死锁和饥饿 15](#_Toc524034542)

[8. Lock与Synchronized区别 16](#_Toc524034543)

[9. 阻塞队列 16](#_Toc524034544)

[10. 同步工具类 16](#_Toc524034545)

[11. 线程封闭 17](#_Toc524034546)

[12. 线程上下文切换 17](#_Toc524034547)

[13. wait()和sleep()的区别 17](#_Toc524034548)

[14. 线程池 18](#_Toc524034549)

[15. Executor和ExecutorService的区别 18](#_Toc524034550)

[16. ThreadPoolExector 18](#_Toc524034551)

[17. ThreadLocal 19](#_Toc524034552)

[五、 JVM 20](#_Toc524034553)

[1. 内存模型 20](#_Toc524034554)

[2. Java的对象管理 21](#_Toc524034555)

[3. JVM垃圾回收机制 22](#_Toc524034556)

[4. G1和CMS收集器 23](#_Toc524034557)

[5. Class类文件结构 24](#_Toc524034558)

[6. 类加载机制 25](#_Toc524034559)

[7. 类加载器 26](#_Toc524034560)

[8. JVM优化 27](#_Toc524034561)

[六、 SSM框架 29](#_Toc524034562)

[1. Spring 29](#_Toc524034563)

[2. spring与springboot关系 29](#_Toc524034564)

[3. spring structs 29](#_Toc524034565)

[4. SpringMVC 29](#_Toc524034566)

[5. MyBatis 29](#_Toc524034567)

[6. hibernate与mybatis区别 30](#_Toc524034568)

[七、 MySQL数据库部分 31](#_Toc524034569)

[1. 事务 31](#_Toc524034570)

[2. 事务隔离级别 31](#_Toc524034571)

[3. MyISAM与InnoDb 31](#_Toc524034572)

[4. B+树 31](#_Toc524034573)

[5. 聚集索引与非聚集索引 31](#_Toc524034574)

[6. 创建索引的原则 32](#_Toc524034575)

[7. 索引能够使用的场景 32](#_Toc524034576)

[8. 存在索引但不能使用的情况 32](#_Toc524034577)

[9. SQL优化 32](#_Toc524034578)

[10. 常用SQL优化 33](#_Toc524034579)

[11. 如何设计一个数据库表，多对多关系映射如何设计 34](#_Toc524034580)

[八、 Linux部分 34](#_Toc524034581)

[九、 其他 34](#_Toc524034582)

[1. Redis 34](#_Toc524034583)

[2. FIFO、LRU、LFU 34](#_Toc524034584)

[3. 高并发 35](#_Toc524034585)

[4. 分布式锁 35](#_Toc524034586)

[5. Xml、JSON 35](#_Toc524034587)

[6. MongoDB 35](#_Toc524034588)

[7. Memcache 35](#_Toc524034589)

[十、 附录 36](#_Toc524034590)

1. 网络部分
2. 三次握手与四次挥手

**(1) 三次握手**

为什么A还要发送一次确认呢？可以二次握手吗？

答：主要为了防止已失效的连接请求报文段突然又传送到了B，因而产生错误。如A发出连接请求，但因连接请求报文丢失而未收到确认，于是A再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，A发出了两个连接请求报文段，其中第一个丢失，第二个到达了B，但是第一个丢失的报文段只是在某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达B，此时B误认为A又发出一次新的连接请求，于是就向A发出确认报文段，同意建立连接，不采用三次握手，只要B发出确认，就建立新的连接了，此时A不理睬B的确认且不发送数据，则B一致等待A发送数据，浪费资源。

**(2) 四次挥手**

为什么A在TIME-WAIT状态必须等待2MSL(最长报文段寿命)的时间？

答：1）保证A发送的最后一个ACK报文段能够到达。这个ACK报文段有可能丢失，使得处于LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认，B超时重传FIN+ACK报文段，而A能在2MSL时间内收到这个重传的FIN+ACK报文段，接着A重传一次确认，重新启动2MSL计时器，最后A和B都进入到CLOSED状态，若A在TIME-WAIT状态不等待一段时间，而是发送完ACK报文段后立即释放连接，则无法收到B重传的FIN+ACK报文段，所以不会再发送一次确认报文段，则B无法正常进入到CLOSED状态。

2）防止“已失效的连接请求报文段”出现在本连接中。A在发送完最后一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失，使下一个新的连接中不会出现这种旧的连接请求报文段。

1. UDP

(1) 是无连接的，即发送数据之前不需要建立连接，因此减少了开销和发送数据之前的时延；

(2) UDP使用尽最大努力交付，不保证可靠交付；

(3) UDP是面向报文的。发送方的UDP对应用层交下来的报文，在添加首部后就向下交付IP层。即应用层交给UDP多长的报文，UDP就照样发送，一次发送一个报文；

(4) UDP没有拥塞控制。允许在网络拥塞时丢失一些数据，但不允许数据有太大的时延；

(5) UDP支持一对一、一对多、多对一和多对多的交互通信；

(6) UDP的首部开销小，只有8字节，TCP首部20字节

1. TCP

**2.1 主要特点**

(1) 面向连接。传输前建立连接，传输完毕释放连接；

(2) 点对点通信。

(3) 可靠交付。数据无差错、不丢失、不重复、按序到达；

(4) 全双工通信。通信双方在任何时候都能发送数据，两端都设有发送缓存和接收缓存；

(5) 面向字节流。TCP并不关心应用进程一次把多长的报文发送到TCP缓存中，而是根据对方给出的滑动窗口值和当前网络拥塞的程度来决定一个报文段应包含多少个字节；

**2.2 可靠传输的工作原理**

(1) 停止等待协议。（自动重传请求ARQ）每发送完一个分组就停止发送，等待对方的确认，收到确认后再发送下一个分组；

* 超时重传。

传输过程中数据丢失、接收端检测到数据出现差错，就不会向发送端回复确认或者确认丢失，都会引起超时重传；

* 确认丢失和确认迟到

当接收端收到重传的分组，发现分组是重复的后就会丢弃这个重复的分组，并向发送端回复确认；

(2) 连续ARQ协议

* 发送方维持一个发送窗口，位于窗口内的分组都可以连续发送出去，不需要等待前一个的确认；当收到对第n个分组的确认后就将发送窗口向前移动n个分组的位置；当发送窗口已满，但未收到确认时，在一段时间（超时重传设置时间）后重传这些数据；
* 接收方采用累积确认的方式，在收到几个分组后，对按序到达的最后一个分组发送确认；（确认号N表示到N-1为止的所有数据都已正确收到）。

**3.3 滑动窗口**

（1）发送方通过接收端发来的确认报文中的确认号和窗口字段设置发送窗口的后沿及大小；发送缓存中存放的是发送应用程序传送给发送方TCP准备发送的数据或是TCP已发送但尚未收到确认的数据；

（2）接收方缓存中存放的是按序到达、但尚未被接收应用程序读取的数据或是未按序到达的数据；如果接收缓存被填满，会使接收窗口减小到零，反之，如果接收应用程序及时从缓存中读取数据，接收窗口可以增大，但不能超过缓存大小；

**3.4 流量控制与拥塞控制**

（1）流量控制指点对点通信量的控制，所要做的就是抑制发送端发送数据的速率，以便接收端来得及接收；

（2）拥塞控制是防止过多的数据注入到网络中，使网络中的路由器或链路不致过载；

3.5 几种拥塞控制方法

（1）慢开始和拥塞避免

由小到大逐渐增大拥塞窗口，慢开始将初始拥塞窗口设置为一个最大报文段的数值，然后在每收到一个新的报文段的确认后就将窗口加倍；拥塞避免是每经过一个往返时间RTT就将拥塞窗口加1；

当拥塞窗口小于慢开始门限时使用慢开始算法，大于后使用拥塞避免算法；无论在慢开始阶段还是拥塞避免阶段，只要没有按时收到确认（认为出现拥塞），就把慢开始门限设置为出现拥塞时发送发窗口值的一半，然后将拥塞窗口重新设置为1，执行慢开始算法；

（2）快重传和快恢复

快重传算法规定，接收方及时发送对失序报文段的重复确认，不必等待该报文段的重传计时器到期，当发送发一连收到三个重复确认就立即重传对方尚未收到的报文段。

快恢复是指，当发送方连续收到三个重复确认时，把慢开始门限减半，然后将拥塞窗口值设为慢开始门限减半后的值，然后开始执行拥塞避免算法；

1. HTTP协议

1.1 GET与POST

(1) GET用于发送请求来获得服务器上的资源，而POST方法是用于向服务器提交资源让服务器处理，比如提交表单、上传文件等，可能导致建立新的资源或者对原有资源的修改；

(2) GET请求的数据会附在URL之后，而POST方法提交的数据则放置在HTTP报文实体的主体里，所以POST方法的安全性比GET方法要高；

(3) GET方法传输的数据量一般限制在2KB；POST方法对于数据大小是无限制的，真正影响到数据大小的是服务器处理程序的能力。

1.2 状态码

* 1xx表示通知信息，如请求收到了或正在进行处理；
* 2xx表示成功，如接收或知道了；
* 3xx表示重定向，如要完成请求还必须采取进一步的行动；
* 4xx表示客户的差错，如请求中有错误的语法或不能完成；
* 5xx表示服务器差错，如服务器失效无法完成请求；

1. Socket编程
2. Java NIO
3. Netty框架
4. Java基础
5. 设计模式
6. Java多态的体现

多态有四种体现形式，接口和接口的继承、类与类的继承、重载、重写。其中重载和重写是核心：

（1）重载：发生在同一个类中，该类中可以存在多个同名方法，但是方法的参数类型和个数不能一样。调用时根据传入的参数来选择哪种方法，这就是多态的体现。

（2）重写：重写发生在子类继承父类的关系中，父类中的方法被子类继承，方法名，返回值类型，参数完全一样，但是方法体不一样。当父类引用指向子类对象，调用重写的方法时，其实本质是父类的方法，但是因为子类重写了方法，变成了另一种状态，这就是多态。

1. Java拆箱装箱

System.out.println(127==127); //true

System.out.println(128==128); //true

System.out.println(new Integer(127) == new Integer(127)); //false

System.out.println(Integer.parseInt("128")==Integer.parseInt("128")); //true

System.out.println(Integer.valueOf("127")==Integer.valueOf("127")); //true

System.out.println(Integer.valueOf("128")==Integer.valueOf("128")); //false

System.out.println(Integer.parseInt("128")==Integer.valueOf("128")); //true

* Integer.parseInt每次构造一个int常量；
* Integer.valueOf返回一个Integer对象，默认在-128~127之间时返回缓存中的已有对象（如果存在的话）；

对于整型的比较，首先判断是值比较还是对象比较，值比较肯定返回true，有一个是值就是值比较。对象比较，则看对象是怎么构造出来的，如果是采用new Integer方式，则每次产生新对象，两个new出来的Integer比较肯定返回false，如果是Integer.valueOf方式的话，注意值的区间是否在-128～127之间，如果在，则构造的相同值的对象是同一个对象，==比较后返回true,否则返回false。

注意：Double和Float类型装箱都会新建对象，会返回false；

1. Java泛型

**4.1 什么是泛型，为什么要使用泛型？**

泛型，即“参数化类型”。就是将类型由原来的具体的类型参数化，然后在使用/调用时传入具体的类型（类型实参）。使用泛型的好处：提供编译时期的类型安全，更强的类型检查是最重要的特征之一，因为它省去了检查运行时类型错误的异常处理。另一个好处就是消除了强制转换的存在，这意味着你可以写更少的代码，因为编译器清楚的知道在集合中存储的类型。

**4.2 类型擦除**

java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。

**4.3 一些泛型区别**

（1）List<?>和List<Object>之间的区别：可以把List<String>赋值给List<?>，却不能把List<String>赋值给 List<Object>；

（2）List<Object>和原始类型List之间的区别：在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检 查，通过使用Object作为类型，可以告知编译器该方法可以接受任何类型的对象，比如String或Integer；

1. Java序列化

将对象转换为一串字节流的过程。需要序列化的类必须实现Serializable或者Externalizable接口。

（1）主要用途：

* **对象持久化**：将对象保存到文件中，延长对象的存在时间，下次使用时再恢复该对象。除了将对象保存在文件中，还有很多种方式实现持久化，如Hibernate框架。
* **对象复制**：通过序列化，将对象保存在内存中，可以再通过此数据得到多个对象的副本。
* **对象传输**：通过序列化，将对象转化字节流后，可以通过网络发送给另外的Java程序。

（2）指定版本号的作用

当你序列化了一个类实例后，希望更改一个字段或添加一个字段，不设置serialVersionUID，所做的任何更改都将导致无法反序化旧有实例，并在反序列化时抛出一个异常。如果你添加了serialVersionUID，在反序列旧有实例时，新添加或更改的字段值将设为初始化值（对象为null，基本类型为相应的初始默认值），字段被删除将不设置。

1. 异常

（1）非检查异常：rror 和 RuntimeException 以及他们的子类。这样的异常发生的原因多半是代码写的有问题。如除0错误ArithmeticException，错误的强制类型转换错误ClassCastException，数组索引越界ArrayIndexOutOfBoundsException，使用了空对象NullPointerException等等。

（2）检查异常：除了Error 和 RuntimeException的其它异常。该异常在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException , IOException,ClassNotFoundException 等。

1. Collections.sort()和Arrays.sort()排序算法

* Collections.sort()方法采用归并排序；
* Arrays.sort()方法，如果数组长度大于等于286且连续性好的话，就用归并排序，如果大于等于286且连续性不好的话就用快速排序。如果长度小于286且大于等于47的话就用快速排序，如果长度小于47的话就用插入排序。

1. HashMap 与HashTable的区别

* Hashtable既不支持Null key也不支持Null value。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。
* Hashtable是线程安全的，它的每个方法中都加入了Synchronize方法。HashMap不是线程安全的，在多线程并发的环境下，可能会产生死锁等问题，但HashMap效率高；
* Hashtable默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。
* Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的。 HashMap为了提高计算效率，将哈希表的大小固定为了2的幂，这样在取模预算时，不需要做除法，只需要做位运算。位运算比除法的效率要高很多。

1. Hashmap扩容

**9.1 扩容条件**

Hashmap的扩容需要同时满足两个条件：当前数据存储的数量（即size()）大小必须大于等于阈值；当前加入的数据是否发生了hash冲突。

（1）就是hashmap在存值的时候（默认大小为16，负载因子0.75，阈值12），可能达到最后存满16个值的时候，再存入第17个值才会发生扩容现象，因为前16个值，每个值在底层数组中分别占据一个位置，并没有发生hash碰撞。

（2）当然也有可能存储更多值（最多可以存26个值）都还没有扩容。原理：前11个值全部hash碰撞，存到数组的同一个位置（这时元素个数小于阈值12，不会扩容），后面所有存入的15个值全部分散到数组剩下的15个位置，前面11+15=26，所以在存入第27个值的时候才同时满足上面两个条件，这时候才会发生扩容现象。

**9.2 多线程下不安全**

扩容时旧桶数组中的某个桶的外挂单链表是通过头插法插入新桶数组中的。多线程情况下，由于缺乏同步机制，当多个线程同时resize的时候，某个线程t所持有的引用next（next指向原桶数组中某个桶外挂单链表的下一个需要转移的Entry），可能已经被转移到了新桶数组中，那么最后该线程t实际上在对新的桶数组进行transfer操作。如果有更多的线程出现这种情况，那很可能出现大量线程都在对新桶数组进行transfer，那么就会出现多个线程对同一链表无限进行链表反转的操作，极易造成死循环，数据丢失等等，因此HashMap不是线程安全的，考虑在多线程环境下使用并发工具包下的ConcurrentHashMap。

1. ConcurrentHashMap

不支持键值为空！

【JDK 1.7】

ConcurrentHashMap采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size(), containsValue()）需要使用特殊的实现。

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性。

【JDK 1.8】

JDK1.8摒弃了分段锁的概念，而是启用了一种全新的方式实现,利用CAS算法。它沿用了1.8的HashMap的思想，底层依然由“数组”+链表+红黑树的方式思想，但是为了做到并发，又增加了很多辅助的类，例如TreeBin，Traverser等对象内部类。

**重要的类**

（1）Node

Node是最核心的内部类，它包装了key-value键值对，所有插入ConcurrentHashMap的数据都包装在这里面。它与HashMap中的定义很相似，但是但是有一些差别它对value和next属性设置了volatile同步锁(与JDK7的Segment相同)，它不允许调用setValue方法直接改变Node的value域，它增加了find方法辅助map.get()方法。

（2）TreeNode

树节点类，另外一个核心的数据结构。当链表长度过长的时候，会转换为TreeNode。但是与HashMap不相同的是，它并不是直接转换为红黑树，而是把这些结点包装成TreeNode放在TreeBin对象中，由TreeBin完成对红黑树的包装。而且TreeNode在ConcurrentHashMap集成自Node类，而并非HashMap中的集成自LinkedHashMap.Entry<K,V>类，也就是说TreeNode带有next指针，这样做的目的是方便基于TreeBin的访问。

（3） TreeBin

这个类包装了很多TreeNode节点。它代替了TreeNode的根节点，在实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，这是与HashMap的区别。另外这个类还带有了读写锁。

（4） ForwardingNode

一个用于连接两个table的节点类。它包含一个nextTable指针，用于指向下一张表。而且这个节点的key value next指针全部为null，它的hash值为-1. 这里面定义的find的方法是从nextTable里进行查询节点，而不是以自身为头节点进行查找。

**扩容方法**

第一部分是构建一个nextTable,它的容量是原来的两倍，这个操作是单线程完成的。

第二个部分就是将原来table中的元素复制到nextTable中，如果遍历到的节点是forward节点，就向后继续遍历，再加上给节点上锁的机制，就完成了多线程的控制。多线程遍历节点，处理了一个节点，就把对应点的值set为forward，另一个线程看到forward，就向后遍历。这样交叉就完成了复制工作。

**Put方法**

如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就帮助进行扩容；

如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。如果这个位置是空的，那么直接放入，而且不需要加锁操作。

1. Java Web
2. Session与Cookie的区别

（1）Session 是存放在服务器端的，当浏览器第一次发送请求时，服务器自动生成了一个Session和一个Session ID用来唯一标识这个Session，并将其通过响应发送到浏览器。当浏览器第二次发送请求，会将前一次服务器响应中的Session ID放在请求中一并发送到服务器上，服务器从请求中提取出Session ID，并和保存的所有Session ID进行对比，找到这个用户对应的Session。

Session ID的保存方法

* 使用Cookie来保存，如果我们不设置这个过期时间，当浏览器关闭的时候，Cookie就消失了，这个Session ID就丢失了。
* 使用URL附加信息的方式，如aaa.jsp?JSESSIONID=\*一样的。这种方式和第一种方式里面不设置Cookie过期时间是一样的。
* 第三种方式是在页面表单里面增加隐藏域，这种方式实际上和第二种方式一样，只不过前者通过GET方式发送数据，后者使用POST方式发送数据。

（2）如果浏览器使用的是 cookie，那么所有的数据都保存在浏览器端。 Cookies不会占服务器资源，是存在客服端内存或者一个cookie的文本文件中；而“Session”则会占用服务器资源。

1. 请求转发与重定向的区别

1、转发是在服务器端完成的，重定向是在客户端发生的；

2、转发的速度快，重定向速度慢；

3、转发是同一次请求，重定向是两次请求；

4、转发地址栏没有变化，重定向地址栏有变化；

5、转发必须是在同一台服务器下完成，重定向可以在不同的服务器下完成。

1. Web容器加载

启动一个WEB项目，WEB容器会先去读取它的配置文件web.xml,读取<context-param>、<listener>两个节点。接着，容器创建一个ServletContext(servlet上下文)，这个web项目的所有部分都讲共享这个上下文。容器将<context-param>转换为键值对，并交给servletContext。容器创建<listener>中的类实例，创建监听器。

web.xml 的加载顺序是：context-param -> listener -> filter -> servlet ，而同个类型之间的实际程序调用的时候的顺序是根据对应的 xml中mapping 的顺序进行调用的。

1. 进程间的通信方式

（1）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。

（2）有名管道(named pipe)：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。

（3）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。

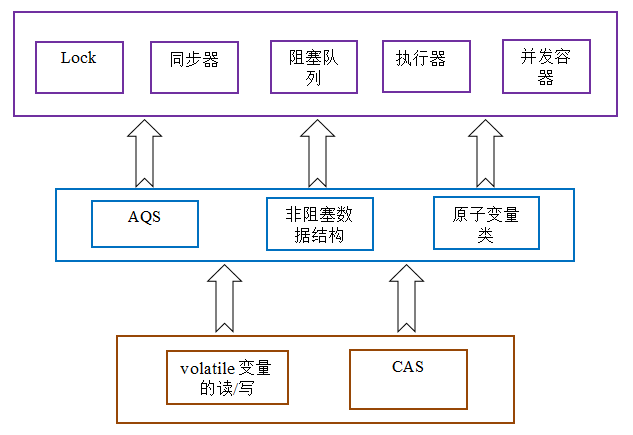
（4）消息队列( message queue )：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

（5）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。

（6）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，如信号量，配合使用，来实现进程间的同步和通信。

（7）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，它可用于不同机器间的进程通信。

1. 并发编程
2. java.util.concurrent包



1. Volatile关键字

Volatile保证数据的可见性，即当一个共享变量被volatile修饰时，它会保证修改的值立即被更新到主存，即线程每次获取volatile变量的值都是最新的。但其并不是多线程安全的。

**2.1 如何保证内存可见性**

（1）对于普通变量：读操作会优先读取工作内存的数据，如果工作内存中不存在，则从主内存中拷贝一份数据到工作内存中;写操作只会修改工作内存的副本数据，这种情况下，其它线程就无法读取变量的最新值。

（2）对于volatile变量，读操作时JMM会把工作内存中对应的值设为无效，要求线程从主内存中读取数据;写操作时JMM会把工作内存中对应的数据刷新到主内存中，这种情况下，其它线程就可以读取变量的最新值。

使用时必须满足下列条件，否则不保证数据准确性：

* 对变量的写操作不依赖当前值，如多线程下执行a++，则无法保证结果的准确性;
* 该变量没有包含在具有其它变量的不变式中；示例详见附录2-4-1

**2.2 volatile与synchronized区别**

* 粒度不同，前者针对变量 ，后者锁对象和类
* syn阻塞，volatile线程不阻塞
* syn保证三大特性，volatile不保证原子性
* syn编译器优化，volatile不优化

1. CAS

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。

（1）ABA问题。如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。

（2）自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

（3）只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者把多个共享变量合并成一个共享变量来操作。

CAS也是适用一些场合的，比如资源竞争小时，是非常适用的，不用进行内核态和用户态之间的线程上下文切换，同时自旋概率也会大大减少，提升性能，但资源竞争激烈时（比如大量线程对同一资源进行写和读操作）并不适用，自旋概率会大大增加，从而浪费CPU资源，降低性能；

1. AQS

AQS是JDK1.5提供的一个基于FIFO等待队列实现的一个用于实现同步器的基础框架。其核心思想是基于volatile int state这样的一个属性同时配合Unsafe工具对其原子性的操作来实现对当前锁的状态进行修改。当state的值为0的时候，标识改Lock不被任何线程所占有。只有一个线程能够修改成功，则持有锁，其他线程准备进入等待队列；

入队列过程中采用自旋+CAS组合来实现非阻塞的原子操作，以保证多并发情况下多个线程逐个进入队列；

AQS的等待队列基于一个双向链表实现的，HEAD节点不关联线程，后面节点关联进入的线程，他们将会按照先后顺序被串联在这个队列上。进入队列的线程，会先查看前驱节点waitStatus是否为SIGNAL，如果是则直接挂起，如果不是则将waitStatus设为SIGNAL，然后再尝试去获取锁，获取失败就进行挂起；SIGNAL的含义是：你释放锁的时候请叫醒我，谢谢！

当一个线程执行结束并释放完锁后，会唤醒头节点的下一个节点的关联线程，到此它的任务也就结束了；被唤醒的线程如果能够成功获取到释放的锁，此节点将变为HEAD节点，获取失败则继续挂起（因为非公平锁允许插队）；

1. 锁机制

（1）**可重入锁**：也叫做递归锁，是一种递归无阻塞的同步机制，指的是同一线程外层函数获得锁之后，内层递归函数不必重新再申请锁。如ReentrantLock 和synchronized 都是可重入锁。

（2）**可中断锁**：可以中断线程等待锁的过程，从而让线程先去处理其他任务。

（3）**公平与非公平锁**：公平即为按先后顺序获取锁，非公平允许插队行为。

（4）**偏向锁**：它是一种针对加锁操作的优化手段，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(涉及到一些CAS操作,耗时)的代价而引入偏向锁。其核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，当这个线程再次请求锁时，无需再做任何同步操作，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。适合无竞争的情况。

（5）**轻量级锁**：轻量级锁由偏向锁升级而来，他所带来性能的提升依据是，在同步执行的周期内不存在竞争。轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。（偏向锁是，不管是不是同一时间的竞争，只要发现有别的线程持有该锁，并且这个线程还没死，我就不愿意，我就要升级）；

（6）**自旋锁**：轻量级锁失败后，虚拟机为了避免线程直接挂起，让当前想要获取锁的线程做几个空循环，如果在此期间还不能获得锁，再将线程挂起，膨胀为重量级锁。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换时间开销比较大；

（7）**重量级锁**：当系统检查到锁是重量级锁之后，会把等待想要获得锁的线程进行阻塞，被阻塞的线程不会消耗cup。但是阻塞或者唤醒一个线程时，都需要操作系统来帮忙，这就需要从用户态转换到内核态，这个转换很耗时，有可能比用户执行代码的时间还要长。

（8）**锁消除**：消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间。

（11）**锁粗化**：如果一系列的连续操作都是对同一个对象反复加锁解锁，甚至加锁操作是出现在循环体中，那即使没有线程竞争，频繁进行互斥同步操作也会导致不必要的性能损耗。如连续利用StringBuffer（线程安全的）的append()方法进行操作时，虚拟机探测到有这样一串零碎的操作都是对同一个对象加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部，比如第一个append()方法到最后一个append()方法，这样只需要加锁一次就可以了。

1. 产生死锁的条件

（1）互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

（2）请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但又对自己获得的资源保持不放

（3）不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

（4）环路等待条件：是指进程发生死锁后，必然存在一个进程--资源之间的环形链；

1. 死锁和饥饿

产生饥饿（也称活锁）的主要原因是：在一个动态系统中，对于每类系统资源，操作系统需要确定一个分配策略，当多个进程同时申请某类资源时，由分配策略确定资源分配给进程的次序。有时资源分配策略可能是不公平的，即不能保证等待时间上界的存在 。在这种情况下，即使系统没有发生死锁，某些进程也可能会长时间等待．当等待时间给进程推进和响应带来明显影响时，称发生了进程饥饿，当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时称该进程被饿死。

* 从进程状态考虑，死锁进程都处于等待状态，忙等待(处于运行或就绪状态)的进程并非处于等待状态，但却可能被饿死；
* 死锁进程等待永远不会被释放的资源，饿死进程等待会被释放但却不会分配给自己的资源，表现为等待时限没有上界(排队等待或忙式等待)；
* 死锁一定发生了循环等待，而饿死则不然。这也表明通过资源分配图可以检测死锁存在与否，但却不能检测是否有进程饿死；
* 死锁一定涉及多个进程，而饥饿或被饿死的进程可能只有一个。
* 在饥饿的情形下，系统中有至少一个进程能正常运行，只是饥饿进程得不到执行机会。而死锁则可能会最终使整个系统陷入死锁并崩溃。

1. Lock与Synchronized区别

* Lock是一个接口，而synchronized是Java中的关键字；
* synchronized在发生异常时，会自动释放线程占有的锁；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
* Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
* 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
* Lock可以提高多个线程进行读操作的效率。
* lock可以设置为公平锁（默认非公平锁）；

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。

1. 阻塞队列

阻塞队列（BlockingQueue）：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景。

* ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列。
* LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列按照先进先出的原则对元素进行排序。
* PriorityBlockingQueue是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列（如果他们实现了Comparable方法），也可以通过比较器comparator来指定元素的排序规则。元素按照升序排列。
* DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。
* SynchronousQueue是一个不存储元素的阻塞队列。它可以直接交付工作，从而降低了将数据从生产者移动到消费者的延迟。因为SynchronousQueue没有存储功能，因此put和take会移植阻塞，直到有另一个线程已经准备好参与到交付过程中。仅当有足够多的消费者，并且总是有一个消费者准备好获取交付的工作时，才适合使用同步队列。

1. 同步工具类

**10.1 闭锁**

闭锁是一种同步工具类，可以延迟线程的进度直到其到达终止状态。可以使一个或多个线程等待一组事件的发生。

**CountDownLatch**是一个灵活的闭锁实现，对象内部存有一个整数作为计数器。调用countDown()方法就将计数器减1，当计数到达0时，则所有等待者会停止等待。计数器的操作是原子性的。计数器一旦到0，无法被重置。

**10.2** **FutureTask**

FutureTask也可以用作闭锁，表示一种抽象的可生成结果的计算，是通过Callable来实现的，可以有3种状态：等待运行、正在运行、运行完成。Future.get的行为取决于任务的状态，如果任务完成，那么get会立即返回结果，否则get将阻塞直到任务进入完成状态。

**10.3 信号量**

Semaphore用于控制同时访问某个特定资源的操作数量，或者同时执行某个指定的操作数量。还可以用来实现某种资源池（如数据库连接池），或者对容器施加边界。

Semaphore中维护了一个许可集。许可的数量可通过构造函数指定，在执行操作前首先获取许可（如果还有剩余许可），并在使用后释放许可。如果线程没有获取到许可就等待， Semaphore允许同一时间多个线程同时访问临界区。

**10.4 栅栏**

栅栏类似于闭锁，他能阻塞一组线程直到某个事件发生。栅栏与闭锁的区别在于，所有线程必须同时到达栅栏位置，才能继续执行。闭锁用于等待时间，而栅栏用于等待其他线程。

CyclicBarrier可以使一定数量的参与方反复地在栅栏位置汇集，它在并行迭代算法中非常有用，这种算法通常将一个问题拆分成一系列相互独立的子问题。

另一种形式的栅栏，Exchanger用于实现线程间的数据交换，它是一种两方栅栏，各方在栅栏位置上交换数据。当两方执行不对称的操作时，Exchanger会非常有用。例如当一个线程向缓冲区写入数据，另一个线程从缓冲区读取数据。这些线程可以使用Exchanger来汇合，并将满的缓冲区与空的缓冲区交换。当两个线程通过Exchanger交换对象时，这种交换就把这两个对象安全的发布给另一方。

1. 线程封闭

避免并发最简单的方法就是线程封闭，就是把对象封装到一个线程里，只有这一个线程能看到此对象。那么这个对象就算不是线程安全的也不会出现任何安全问题

（1）ad-hoc线程封闭

完全靠实现者控制的线程封闭，这玩意不好，直接忽略。

（2）栈封闭

多个线程访问一个方法，此方法中的局部变量都会被拷贝一分儿到线程栈中。所以局部变量是不被多个线程所共享的，也就不会出现并发问题。所以能用局部变量就别用全局的变量，全局变量容易引起并发问题。

（3）ThreadLocal封闭

ThreadLocal内部维护了一个Map，Map的key是每个线程的名称，而Map的值就是我们要封闭的对象。每个线程中的对象都对应着Map中一个值，也就是ThreadLocal利用Map实现了对象的线程封闭。

1. 线程上下文切换

线程是由CPU进行调度的，CPU的一个时间片内只执行一个线程上下文内的线程，当CPU由执行线程A切换到执行线程B的过程中会发生一些列的操作，这些操作主要有”保存线程A的执行现场“然后”载入线程B的执行现场”，这个过程称之为“上下文切换（context switch）”,这个上下文切换过程并不廉价，如果没有必要，应该尽量减少上下文切换的发生。

1. wait()和sleep()的区别

* sleep来自Thread类，和wait来自Object类
* 调用sleep()方法的过程中，线程不会释放对象锁。而 调用 wait 方法线程会释放对象锁
* sleep睡眠后不出让系统资源，wait让出系统资源其他线程可以占用CPU

sleep(milliseconds)需要指定一个睡眠时间，时间一到会自动唤醒

1. 线程池

线程池是指管理一组同构工作线程的资源池。通过重用现有的线程而不是创建新线程，可以在处理多个请求时分摊在线程创建和销毁过程中产生的巨大开销。另外，当请求到达时，工作线程通常已经存在，因此不会由于等待创建线程而推迟任务的执行，从而提高了响应性。

**Executors类**提供工厂方法用来创建不同类型的线程池，如:

* newSingleThreadExecutor创建一个只有一个线程的线程池，如果这个线程异常结束，会创建另一个线程来代替。
* newFixedThreadPool来创建固定线程数的线程池，每当提交一个任务时就创建一个线程，直到达到线程池的最大数量，这时线程池的规模将不再变化。
* newCachedThreadPool创建一个可缓存的线程池，如果线程池的规模超过了处理需求时，那么将回收空闲线程，而当需求增加时，则可以添加新的线程，线程池的规模不存在任何限制。

1. Executor和ExecutorService的区别

|  |  |
| --- | --- |
| Executor | ExecutorService |
| Executor 是 Java 线程池的核心接口，用来并发执行提交的任务 | ExecutorService 是 Executor 接口的扩展，提供了异步执行和关闭线程池的方法 |
| 提供execute()方法用来提交任务 | 提供submit()方法用来提交任务 |
| execute()方法无返回值 | submit()方法返回Future对象，可用来获取任务执行结果 |
| 不能取消任务 | 可以通过Future.cancel()取消pending中的任务 |
| 没有提供和关闭线程池有关的方法 | 提供了关闭线程池的方法 |

1. ThreadPoolExector

（1）构造方法部分参数：

* corePoolSize核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。
* maximumPoolSize线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为无界的LinkedBlockingDeque时，这个值无效。
* keepAliveTime非核心线程的闲置超时时间，超过这个时间就会被回收。
* workQueue线程池中的任务队列.，常用的有三种队列，SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue。
* threadFactory线程工厂，提供创建新线程的功能。
* RejectedExecutionHandler当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution方法。

（2）线程池规则：

**任务队列无界时**：

* 如果线程数量<=核心线程数，来一个任务创建一个线程。
* 如果线程数量>核心线程数，但<=最大线程数，并且任务队列是LinkedBlockingDeque的时候，来的任务放在队列中排队。
* 如果线程数量>核心线程数，但<=最大线程数，并且任务队列是SynchronousQueue的时候，来一个任务创建一个线程。这些线程超时会被清除。
* 如果线程数量>核心线程数，并且>最大线程数，当任务队列是LinkedBlockingQeque，来多少任务都会放进队列中排队。此时线程池的最大线程数设置是无效的。
* 如果线程数量>核心线程数，并且>最大线程数，当任务队列是SynchronousQueue的时候，会因为线程池拒绝添加任务而抛出异常。

**当任务队列有界时**

当LinkedBlockingDeque塞满时，新增的任务会直接创建新线程来执行，当创建的线程数量超过最大线程数量时会拒绝任务或抛异常。

1. ThreadLocal

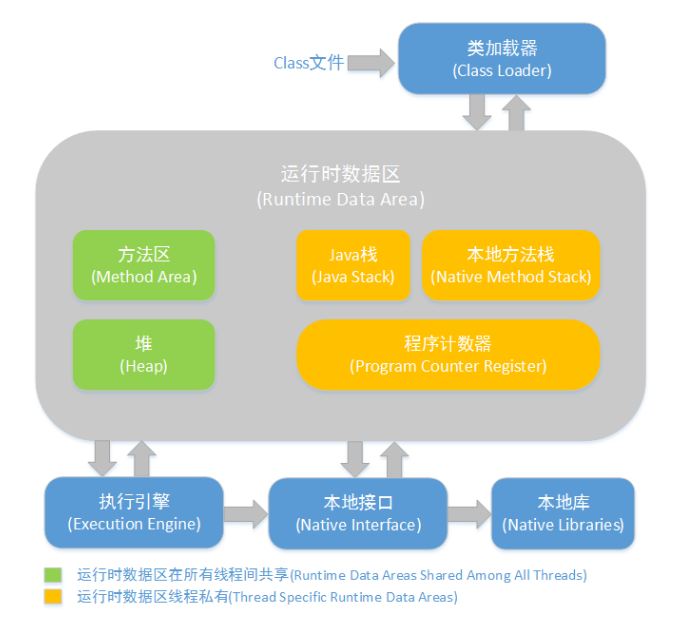
ThreadLocal 是线程的局部变量，主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰。

Thread 在内部是通过ThreadLocalMap来维护ThreadLocal变量表， 在Thread类中有一个threadLocals 变量，是ThreadLocalMap类型的，它就是为每一个线程来存储自身的ThreadLocal变量的，ThreadLocalMap是ThreadLocal类的一个内部类，这个Map里面的最小的存储单位是一个Entry，它使用ThreadLocal作为key，变量作为 value，这是因为在每一个线程里面，可能存在着多个ThreadLocal变量。

最常见的ThreadLocal使用场景为，用来解决数据库连接、Session管理等。

1. JVM
2. 内存模型

**1.1 Java内存模型**



**（1）Java堆**

是JVM所管理的内存中最大的一块，其在JVM启动时创建，用于存放实例对象，基本上每次new对象的时候都会在这里分配内存。

**（2）方法区**

用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

**（3）Java栈**

每个方法被执行时会在栈中创建一个栈帧，用于存放局部变量表（包括基本数据类型和对象引用等）、操作数栈、动态链接、方法出口等信息。栈的生命周期和当前线程相同，栈帧寿命与方法相同；

**（4）本地方法栈**

本地方法栈与虚拟机栈的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

**（5）程序计数器**

当前线程所执行的（Java方法）字节码的行号指示器，记录的是正在执行的指令的地址；native方法时该计数器值为空；

**1.2 几种常量池**

（1）字符串常量池

StringPool里放的是字符串常量和字符串对象的引用。JDK6及之前版本，字符串常量池是放在方法区中，JDK7.0后，字符串常量池被移到了堆中。因为方法区的内存空间太小，容易造成内存溢出。在HotSpot VM里实现的stringpool功能的是一个StringTable类，它是一个Hash表，默认值大小长度是1009；这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享。字符串常量由一个一个字符组成，放在了StringTable上。

（2）class常量池

Java类被编译后，就会形成一份class文件，每个class文件都有一个class常量池，用于存放编译器生成的各种字面量和符号引用。字面量包括：1.文本字符串 2.八种基本类型的值 3.被声明为final的常量等。符号引用包括：1.类和方法的全限定名 2.字段的名称和描述符 3.方法的名称和描述符。

（3）运行时常量池

class文件的常量池在类加载后会进入方法区的运行时常量池中存放，其具有动态性，它的字面量是可以动态添加的（String#intern()）。运行时常量池也是每个类都有一个。在解析阶段，把符号引用替换为直接引用，解析的过程会去查询字符串常量池（StringTable）以保证运行时常量池所引用的字符串与字符串常量池中是一致的。

**1.3 内存溢出分析**

（1）堆内存溢出

* 查看是否存在内存泄漏；
* 检查程序中是否存在某些对象生命周期过长，持有时间过长的情况，尝试减少程序运行期的内存消耗；
* 检查虚拟机堆参数（-Xmx与-Xms），与物理内存对比看是否可以调大参数。

（2）栈内存溢出

* 线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常；应该检查程序中是否存在大量递归，或者大量的本地变量增加了方法帧中本地变量表的长度。还可以通过-Xss增大栈内存容量；
* 虚拟机在扩展栈时无法申请到足够的内存空间，会抛出OutOfMemoryError异常；可能是在多线程情况下栈空间不够分配，所以可通过减少最大堆和减少栈容量来换取更多的线程。

（3）导致OutOfMemoryError异常的常见原因：

* 内存中加载的数据量过于庞大，如一次从数据库取出过多数据；
* 集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；
* 代码中存在死循环或循环产生过多重复的对象实体；
* 启动参数内存值设定的过小；

1. Java的对象管理

**2.1 对象的创建流程**

（1）当虚拟机遇到一个new关键字时，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并检查这个类是否已被加载、解析和初始化过。如果没有，则进行类加载过程。

（2）类加载检查通过后，虚拟机为新生对象分配内存（所需内存大小在类加载完后便可确定下来），内存分配方式有指针碰撞和空闲列表两种，选择哪种分配方式和GC的回收算法有关。为保证多线程情况下内存指针分配时的安全问题，虚拟机可以采用CAS加失败重试的方式保证更新操作的原子性，也可以采用为每个线程在堆中预先分配一小块内存（本地线程缓冲区），只有当缓冲区不够用并分配新的空间时再采用同步锁定，保证线程安全。

（3）内存分配完成后，虚拟机将分配到的内存空间初始化为零值。这使得对象实例在Java代码中不复制初始值就可以使用。

（4）接下来。虚拟机对对象进行必要的设置，如该对象是哪个类的实例、如何找到该类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。

（5）然后执行Java程序的init()方法对对象进行初始化，至此一个对象才完全产生出来。

**2.2 对象分配原则**

* 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
* 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
* 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC对象的年龄加1，直到达到阀值对象进入老年区。
* 动态判断对象的年龄。如果Survivor区中相同年龄的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可以直接进入老年代。
* 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，查看设置值是否允许担保失败，如果true则只进行Monitor GC,如果false则进行Full GC。

**2.3 可达性分析**

Java中通过可达性分析来判定对象是否存活，其基本思想是通过一系列称为GC Roots的对象作为起始点，从这些节点开始向下搜索，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，则此对象不可用，将被回收

可作为GC Roots的对象包括以下几种：

* 虚拟机栈（栈帧中的本地变量表）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；

1. JVM垃圾回收机制

Hotspot VM提供的垃圾回收器是一个分代垃圾回收器（Generational GC）将内存划分为不同的阶段，也就是说，不同的生命周期的对象放置在不同的地址池中。这种分代方式可以减少垃圾回收的停顿时间以及大范围对象的回收成本。

**3.1 Hotspot VM将其堆空间分为三个分代空间：**

（1）年轻代（Young Generation）

* Java应用在分配Java对象时，这些对象会被分配到年轻代堆空间中去
* 这个空间大多是小对象并且会被频繁回收
* 由于年轻代堆空间的垃圾回收会很频繁，因此其垃圾回收算法会更加重视回收效率

（2）年老代（Old Generationn）

* 年轻代堆空间的长期存活对象会转移到（也许是永久性转移）年老代堆空间
* 这个堆空间通常比年轻代的堆空间大，并且其空间增长速度较缓
* 由于大部分JVM堆空间都分配给了年老代，因此其垃圾回收算法需要更节省空间，此算法需要能够处理低垃圾密度的堆空间

（3）持久代（Permanent Generation）

存放VM和Java类的元数据（metadata），以及interned字符串和类的静态变量

**3.2 次收集（Minor GC）和全收集（Full GC）**

当年轻代堆空间满了的时候，会触发次收集将还存活的对象移到年老代堆空间。当年老代堆空间满了的时候，会触发一个覆盖全范围的对象堆的全收集。可以使用System.gc()方法来显式的启动全收集。

**新生代收集器**：Serial（复制）、PraNew（停止-复制）、ParallelScavenge（停止-复制）

**年老代收集器**：SerialOld（标记-整理）、ParallelOld（停止-复制）、CMS（标记-清理）

**G1收集器**：整体上是标记整理，局部上是复制算法。

1. G1和CMS收集器

**4.1 CMS（Concurrent Mark Sweep）**

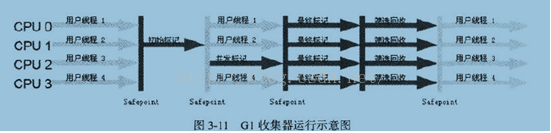


CMS收集器是一种以获取最短回收停顿时间为目标的收集器，基于标记-清除算法实现。他的运作包括初始标记、并发标记、重新标记、并发清除。其中初始标记和重新标记仍需要stop the world。初始标记仅仅是标记GC Roots能直接关联到的对象，速度很快。并发标记是进行GC Roots可达性分析，找出存活的对象。重新标记是为了修正并发标记期间因用户程序继续运行而导致的标记变动的那一部分对象的标记记录。这一阶段会比初始标记阶段稍长，但远比并发标记短。

**CMS的缺点：**

* CMS对CPU资源非常敏感。因为采用并发设计，所以会占用一部分线程资源导致应用程序变慢，总吞吐量降低。
* CMS无法处理浮动垃圾。浮动垃圾是指CMS在并发清除阶段用户线程还在运行，并产生新的垃圾。这些垃圾出现在标记过程之后，只能下次GC时清除。还有就是，由于垃圾回收阶段用户线程还在运行，所以CMS并不能等到年老代完全满了再收集，需要预留一些空间供并发收集时线程运行使用（92%）。当预留空间不够使用时，虚拟机将启用后备方案，即启用Serial Old收集器重新进行年老代的收集，停顿时间将变长。
* 由于采用标记-清除算法，产生大量内存碎片。可能会提前触发Full GC，所以虚拟机可以设置一个参数，在执行多少次标记-清楚后，执行一次标记-整理算法。

**4.2 G1收集器**



G1是一款面向服务端应用的垃圾收集器。G1将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，他们都是一部分Region的集合（不需要连续）。

运作过程和CMS有很多相似之处：初始标记、并发标记、最终标记、筛选回收。前面几个过程类似CMS，最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。

**G1特点**：

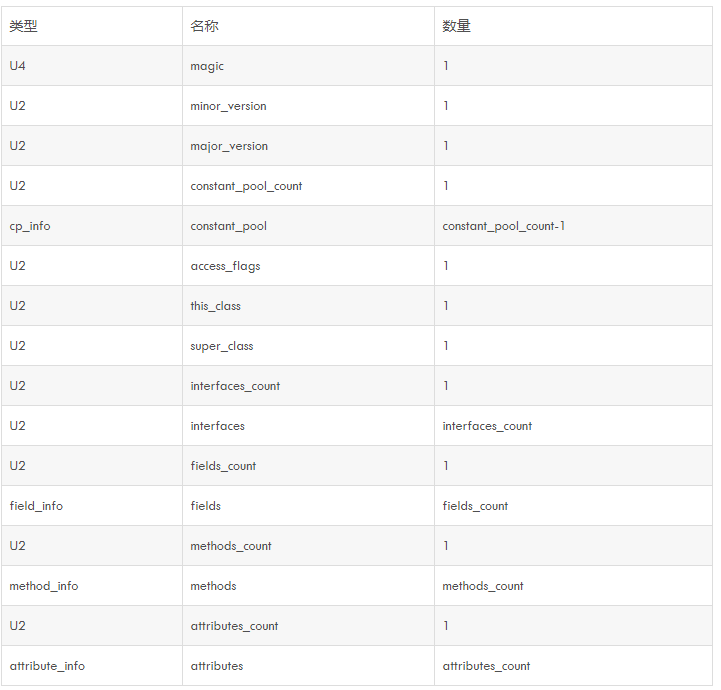
* 并行与并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
* 分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。
* 空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
* 可预测的停顿：因为它可以有计划的避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个region里的垃圾堆积的价值大小（所获空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先收集价值最大的region。

1. Class类文件结构

class文件是一组以8位字节为基础的二进制流，各个数据项目严格按照顺序紧凑地排列在class文件中，中间没有任何分隔符，这点和png、jpg等图片文件格式类似。当遇到需要占用8位字节以上空间的数据项时，则会按照大端序的字节顺序分隔为若干个8位字节进行存储。

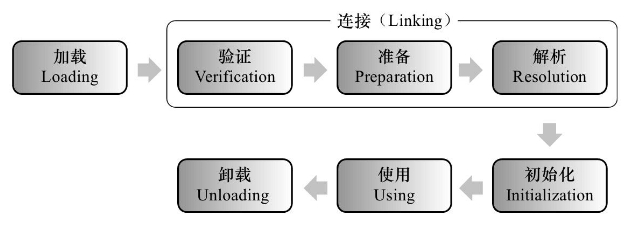
注：大端序是指按最高位字节在地址最低位，最低位字节在地址最高位的顺序存储数据。小端序反之。一般PowerPC等处理器、TCP/IP等采用大端序，x86采用小端序。

Java虚拟机规范规定class文件格式结构只有两种数据类型：无符号数和表。其中无符号数属于基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节。无符号数可以用来描述数字、索引引用、数量值或按照utf-8编码构成的字符串值。而表是由多个无符号数或其他表构成的复合数据结构，所有的表都以“\_info”结尾。表用于描述有层次关系的复合结构的数据。



1. 类加载机制

**6.1 类加载的时机**



JVM类加载机制分为五个部分：加载，验证，准备，解析，初始化。其中除解析外，其他阶段都会按顺序开始。

**对类进行初始化的5种情况**：

（1）使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

（2）使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

（3）当初始化一个类的时候，发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

（4）当虚拟机启动时，用户需要制定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

（5）当使用JDK1.7的动态语言支持时，如果一个java.lang.incoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_incokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

**6.2 类的加载过程**

**（1）加载**

1. 通过一个类的全限定名来获取定义此类的二进制字节流。字节流可从一个class文件中获取，可从ZIP包中读取（比如从jar包和war包中读取），可从网络中获取（Applet），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）。
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 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访问入口。

**（2）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。文件格式验证、元数据验证、字节码验证、符号引用验证。

**（3）准备**

准备阶段是正式为类变量（被static修饰的变量）分配内存并设置类变量的初始值阶段（零值），即在方法区中分配这些变量所使用的内存空间。

**（4）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。

* 符号引用以一组符号来描述所引用的目标，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。

**（5）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。

1. 类加载器

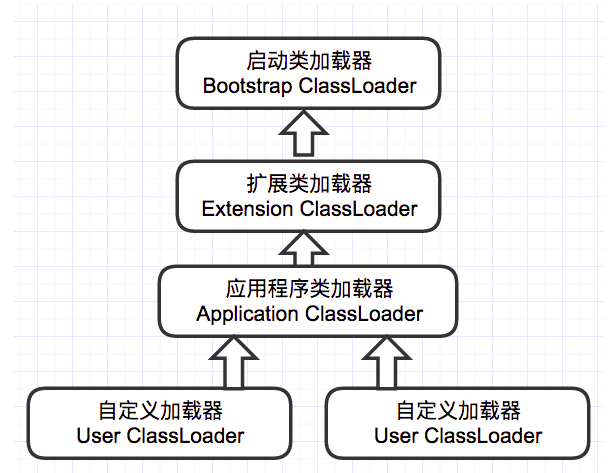
虚拟机设计团队把加载动作放到JVM外部实现，以便让应用程序决定如何获取所需的类，实现这个动作的代码模块称为类加载器。

JVM提供了3种类加载器：

* 启动类加载器：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：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：负责加载用户路径（classpath）上的类库。

**双亲委派模型**

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。

1. JVM优化

**8.1 Full GC的优化**  
对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

（1）旧生代空间不足

调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

（2）Pemanet Generation空间不足

增大Perm Gen空间，避免太多静态对象

（3）统计得到的GC后晋升到年老代的平均大小大于年老代剩余空间

控制好新生代和旧生代的比例

（4）System.gc()被显示调用

垃圾回收不要手动触发，尽量依靠JVM自身的机制

**（2）分代比例优化**

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，当各部分比例不良设置会导致以下后果

1). 新生代设置过小

一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2). 新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

一般说来新生代占整个堆1/3比较合适

3). Survivor设置过小

导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4). Survivor设置过大

导致eden过小，增加了GC频率

另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

**8.3 GC组合策略**

由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，JVM提供两种较为简单的GC策略的设置方式

1). 吞吐量优先

JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n来设置

2). 暂停时间优先

JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n来设置

**8.4 JVM常见配置**

-xmx最大堆内存

-xms初始时堆内存

-xmn堆内新生代大小

-xss线程堆栈大小

1. SSM框架
2. Spring

Spring组件用过那些

Spring注解如何实现的

BeanFactory是什么

Bean在加载的时候的流程

FactoryBean是什么？

BeanFactory有什么实现类

Spring如何加载配置文件

Spring 有哪些技术 AOP具体实现，写一个AOP切面，切面切点怎么定义以及相关概念。JDK代理实现，手写代理模式类图。

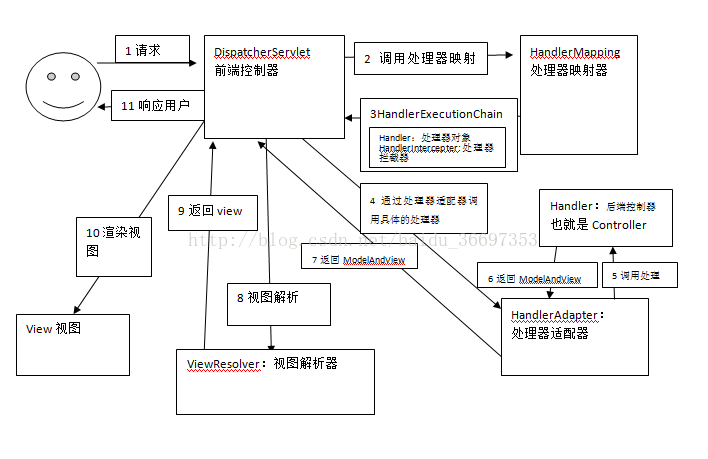
CGLIB与JDK代理区别，应用场景

Java事务怎么写（这里是指Java的sql包的原生事务）

@transactional注解实现原理

@transient注解原理

1. spring与springboot关系
2. spring structs
3. SpringMVC

****

1. MyBatis
2. hibernate与mybatis区别
3. MySQL数据库部分
4. 事务

所谓的事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。事务的提出主要是为了解决并发情况下保持数据一致性的问题。

* Atomic（原子性）：事务中包含的操作被看做一个逻辑单元，这个逻辑单元中的操作要么全部成功，要么全部失败，数据回滚到执行前的状态；
* Consistency（一致性）：事务的执行使数据从一个状态转换为另一状态，但是对于整个数据的完整性保持稳定；
* Isolation（隔离性）：在该事务执行过程中，无论发生的任何数据的改变都应该只存在于该事务之中，对外界不存在任何影响。只有在事务正确提交后，才会显示该事务对数据的改变，其他事务才能获取到改变后的数据；
* Durability（持久性）：事务完成之后，它对于 系统的影响是永久的，该修改即使出现系统故障也将一直保留，真实的修改了数据库

利用事务机制，对数据库的操作要么全部执行，要么全部不执行，保证数据库的一致性。需要使用事务的SQL语句通常是更新和删除操作等。

1. 事务隔离级别

（1）read uncommitted(未提交读)：在读取时是不会加锁的，但在更新数据时，对其加行级共享锁（其它事务不能更改，但可以读取，导致脏读），事务结束时释放。这种隔离级别未处理任何以上4个问题。

（2）read conmmitted(提交读)：解决了脏读问题，解决方式是给写数据加行级排他锁，给读的数据加行级共享锁，一旦读完该行，立即释放该行级共享锁。这种模式下虽然处理了脏读，但是并没有处理丢失更新和不可重复读的问题。

（3）reapetable read(可重复读)：解决了不可重复读的问题，同时也解决了更新丢失的问题。给写的数据加行级排他锁，给读的数据加行级共享锁，直到事务结束才释放。

（4）serializable (序列化)：事务读数据则加表级共享锁，事务写数据则加表级排他锁，幻读问题也得到了解决。

1. MyISAM与InnoDb

（1）MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。

（2）在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。InnoDB的辅助索引data域存储相应记录主键的值而不是地址。

1. B+树
2. 聚集索引与非聚集索引

（1）聚集索引：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。索引的叶子节点就是对应的数据节点，可以直接获取到对应的全部列的数据。

（2）非聚集索引：叶节点仍然是索引节点，只是有一个指针指向对应的数据块，此如果使用非聚集索引查询，而查询列中包含了其他该索引没有覆盖的列，那么他还要进行第二次的查询，查询节点上对应的数据行的数据。

1. 创建索引的原则

（1）对于查询频率高的字段创建索引；

（2）对排序、分组、联合查询频率高的字段创建索引；

（3）索引的数目不宜太多；每创建一个索引都会占用相应的物理控件，而且过多的索引会导致insert、update、delete语句的执行效率降低；

（4）若在实际中，需要将多个列设置索引时，可以采用多列索引；

（5）选择唯一性索引；

（6）尽量使用数据量少的索引；如果索引的值很长，那么查询的速度会受到影响。

（7）尽量使用前缀来索引；如果索引字段的值很长，最好使用值的前缀来索引。

（8）删除不再使用或者很少使用的索引；

1. 索引能够使用的场景

（1）匹配全值，对索引中所有列都指定具体值；

（2）匹配值的范围查询，对索引的值能够进行范围查询；

（3）匹配最左前缀，仅仅使用索引中的最左边列进行查找；

（4）仅仅对索引进行查询，当查询的列都在索引的字段中时，查询的效率更高；

（5）匹配列前缀，仅仅使用索引中的第一列，并且只包含索引第一列的开头一部分进行查找；

（6）能够实现索引匹配部分精确，二其他部分进行范围匹配；

1. 存在索引但不能使用的情况

（1）以%开头的LIKE查询不能够利用B树索引；

Select \* from actor where last\_name like ‘%NI%’;

解决办法：首先通过扫描二级索引last\_name获得满足条件like %NI%的主键列表，再根据主键回表去检索记录；

Select \* from (select actor\_id form actor where last\_name like ‘%NI%’) a, actor b where a.actor\_id=b.actor\_id;

（2）数据类型出现隐式转换的时候也不会使用索引，特别是当列类型是字符串，那么一定要在where条件中把字符常量值用引号引起来；

（3）复合索引的情况下，如果查询条件不包含索引列最左边部分，即不满足最左匹配就不会使用索引；

（4）如果MySQL估计使用索引比全表扫描更慢，则不会使用索引；

（5）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；

1. SQL优化

* Show status 了解各种SQL的执行频率
* Explain 分析低效的SQL执行计划
* 通过show profile分析SQL，查看线程的每个状态和消耗的时间
* 通过trace分析优化器如何选择执行计划（生成一个跟踪文件）

**优化方法：**

（1）定期分析表和检查表

Analyze table \*\*\*; 本语句用于分析和存储表的关键字分布，分析的结果将可以使得系统得到准确的统计信息，使得SQL能够生成正确的执行计划。

Check table \*\*\*; 检查表的作用是检查一个或多个表是否有错误；

（2）定期优化表

Optimize table \*\*\*; 可以将表中的空间碎片进行合并，并且可以消除由于删除或者更新造成的空间浪费；InnoDB可通过alter table \*\*\* engine=innodb; 不修改引擎的方式来回收不用空间；

1. 常用SQL优化

**11.1 大批量插入数据**

（1）优化load命令

MyISAM可先使用disable keys关闭非唯一索引的更新，导入完再打开；

InnoDB可采用一下方式：

* 将导入的数据按照主键的顺序排列，可有效提高导入效率；
* 在导入数据之前执行set unique\_checks=0; 关闭唯一性校验；
* 如果应用使用自动提交的方式，建议在使用前执行set autocommit=0关闭自动提交，导入结束再打开；

（2）优化insert语句

* 从同一客户插入多行，尽量使用多个值表的insert语句，大大缩减客户端与数据库之间的连接关闭等消耗；insert into \*\*\* values(1,2),(3,4)…
* 从不同客户插入多行，可以使用insert delayed语句得到更高的速度。Delayed可以让insert语句马上执行，数据都将被放在内存队列中，其实还并没被写入到磁盘；
* 将索引为念和数据文件在不同的磁盘上存放（利用建表中的选项）
* 进行批量插入，可通过增加bulk\_inert\_buffer\_size变量值的方法来提高速度（MyISAM）
* 从一个文本文件装载表时，可使用load data infile，这比insert快20倍

**11.2 优化order by语句**

MySQL中有两种排序方式，第一种通过有序索引顺序扫描直接返回有序数据；第二种通过对返回数据进行排序（filesort排序）；所以优化目标为尽量减少额外的排序，通过索引直接返回有序数据。

不过，MySQL对filesort有两种排序算法：两次扫描算法（内存开销小）、一次扫描算法（效率高）

**11.3 优化group by语句**

默认情况下，MySQL会对所有group by col,col2…的字段进行排序，所以如果想避免排序结果的消耗，可以指定order by null禁止排序；

**11.4 优化嵌套查询**

用join代替子查询，如使用left join替代not in，当子查询中对查询字段建有索引时性能更好；

**11.5 优化or条件**

对于含有or的查询子句，如果要利用索引，则or之间的每个条件列都必须用到索引；如果没有索引，则应该考虑增加独立的索引；复合索引的列上做or操作不能用到索引；

11.6 优化分页查询

（1）在索引上完成排序分页的操作，最后根据主键关联回原表查询所需要的其他列内同。

Select film\_id,description form film order by title limit 50,5; //全表扫描

Select a.film\_id, a.description from film a inner join(select film\_id from film order by title limit 50,5) b on a.film\_id=b.film\_id; //不进行全表扫描，效率更高

（2）将limit查询转化成某个位置的查询。通过增加一个字段，用来记录上一页最后一行的编号，可根据最后一行的记录向后追溯。这样把limit m,n转换为limit n的查询，只适合在排序字段不会出现重复值的特定环境，能够减轻分页翻页的压力；但如果排序字段出现大量重复值，分页结果可能会丢失部分记录。

1. 如何设计一个数据库表，多对多关系映射如何设计
2. Linux部分

Linux查看端口的指令，查看日志的指令

linux命令中查看端口的方式

linux中查看日志的方式

linux查找文件关键字

1. 其他
2. Redis

rabbitmq 交换机 持久化 确认机制 消息丢失处理

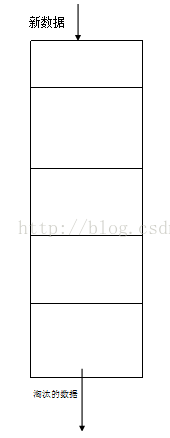
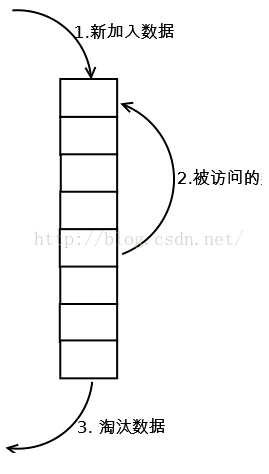
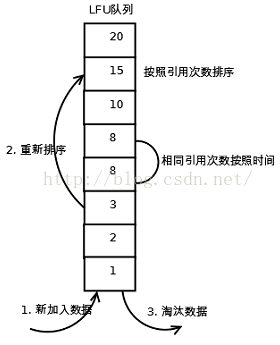
rabbitmq怎么实现集群？【自己面试题整理了这个问题，但是还是没说出来，，涉及broker,cluster等等的】

redis分布式session实现

cookie session的区别

redis集群实现，事务实现，高可用配置，redis的数据结构有哪些

1. FIFO、LRU、LFU

**2.1 FIFO先进先出**

（1）新访问的数据插入FIFO队列尾部，数据在FIFO队列中顺序移动；

（2）淘汰FIFO队列头部的数据；

**2.2 LRU最近最少使用**

（1）新数据插入到链表头部；

（2）每当缓存命中（即缓存数据被访问），则将数据移到链表头部；

（3）当链表满的时候，将链表尾部的数据丢弃。

**2.3 LFU**

（1）新加入数据插入到队列尾部（因为引用计数为1）；

（2）队列中的数据被访问后，引用计数增加，队列重新排序；

（3）当需要淘汰数据时，将已经排序的列表最后的数据块删除。

1. 高并发
2. 分布式锁

基于数据库实现分布式锁

基于缓存（redis，memcached，tair）实现分布式锁

基于Zookeeper实现分布式锁

1. Xml、JSON
2. MongoDB
3. Memcache

1. 附录
2. 2-4-1

public class NumberRange {

　　private volatile int lower = 0;

　　private volatile int upper = 10;

　　public int getLower() { return lower; }

　　public int getUpper() { return upper; }

　　public void setLower(int value) {

　　if (value > upper)

　　throw new IllegalArgumentException(...);

　　lower = value;

　　}

　　public void setUpper(int value) {

　　if (value < lower)

　　throw new IllegalArgumentException(...);

　　upper = value;

　　}

}

上述代码中，上下界初始化分别为0和10，假设线程A和B在某一时刻同时执行了setLower(8)和setUpper(5)，且都通过了不变式的检查，设置了一个无效范围(8, 5)，所以在这种场景下，需要通过sychronize保证方法setLower和setUpper在每一时刻只有一个线程能够执行。