# 数据索引理解、分类比较

## 索引理解

什么是索引： <http://blog.jobbole.com/24006/>

<https://www.cnblogs.com/xiangyangzhu/p/index.html>

**在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引**

能够提高检索的性能，但是会影响插入速度

索引的原理：对要查询的字段建立索引其实就是把该字段按照一定的方式排序；建立的索引只对该字段有用，如果查询的字段改变，那么这个索引也就无效了，比如图书馆的书是按照书名的第一个字母排序的，那么你想要找作者叫张三的就不能用改索引了；还有就是如果索引太多会降低查询的速度

为什么要创建索引呢？这是因为，创建索引可以大大提高系统的性能。   
第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。   
第二，可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。   
第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。   
第四，在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。   
第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？这种想法固然有其合理性，然而也有其片面性。虽然，索引有许多优点， 但是，为表中的每一个列都增加索引，是非常不明智的。这是因为，增加索引也有许多不利的一个方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据 量的增加而增加。   
第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。   
第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。因此，在创建索引的时候，应该仔细考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列 上创建索引，例如：

在经常需要搜索的列上，可以加快搜索的速度；   
在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；   
在经常用在连接的列上，这 些列主要是一些外键，可以加快连接的速度；   
在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；   
在经常需要排序的列上创 建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；   
在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因 为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。   
第二，对于那 些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比 例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。   
第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。   
第四，当修改性能远远大于检索性能时，不应该创建索 引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因 此，当修改性能远远大于检索性能时，不应该创建索引。

<https://www.cnblogs.com/mxmbk/articles/5226344.html>

## 分类比较

### Hash索引

1）Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。   
     由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。

（2）Hash 索引无法被用来避免数据的排序操作。   
     由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；

（3）Hash 索引不能利用部分索引键查询。   
     对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

（4）Hash 索引在任何时候都不能避免表扫描。   
     前面已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

（5）Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。   
     对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

### B-/+Tree索引的性能分析

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

### B+树

1、B+**树的层级更少**：相较于B树B+每个**非叶子**节点存储的关键字数更多，树的层级更少所以查询数据更快；

2、B+**树查询速度更稳定**：B+所有关键字数据地址都存在**叶子**节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定;

3、B+**树天然具备排序功能：**B+树所有的**叶子**节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。

4、B+**树全节点遍历更快：**B+树遍历整棵树只需要遍历所有的**叶子**节点即可，，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

**B树**相对于**B+树**的优点是，如果经常访问的数据离根节点很近，而**B树**的**非叶子**节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比**B+树**快。

### 红黑树

红黑树能够以O(log2(N))的时间复杂度进行搜索、插入、删除操作。此外,任何不平衡都会在3次旋转之内解决。这一点是AVL所不具备的。

红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下的情况在数据较小，可以完全放到内存中时，红黑树的时间复杂度比B树低。

如linux中进程的调度用的是红黑树。

反之，数据量较大，外存中占主要部分时，B树因其读磁盘次数少，而具有更快的速度。

## 参考链接

平衡二叉树、B树、B+树、B\*树： <https://zhuanlan.zhihu.com/p/27700617>

数据库索引的作用和优点缺点<https://blog.csdn.net/ko_tin/article/details/61958420>

# 线程

## 线程中断用处

「通知线程应该中断了」，具体到底中断还是继续运行，应该由被通知的线程自己处理。

## 线程中断处理

具体来说，当对一个线程，调用 interrupt() 时，

* 如果线程处于被阻塞状态（例如处于sleep, wait, join 等状态），那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常。仅此而已。
* 如果线程处于正常活动状态，那么会将该线程的中断标志设置为 true，仅此而已。被设置中断标志的线程将继续正常运行，不受影响。

interrupt() 并不能真正的中断线程，需要被调用的线程自己进行配合才行。也就是说，一个线程如果有被中断的需求，那么就可以这样做。

① 在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程。

* 在调用阻塞方法时正确处理InterruptedException异常。（例如，catch异常后就结束线程。）

## 废弃stop方法原因

一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。所以，Thread.stop, Thread.suspend, Thread.resume 都已经被废弃了。

以前通过thread.stop()可以停止一个线程，注意stop()方法是可以由一个线程去停止另外一个线程，这种方法太过暴力而且是不安全的，怎么说呢，线程A调用线程B的stop方法去停止线程B，调用这个方法的时候线程A其实并不知道线程B执行的具体情况，这种突然间地停止会导致线程B的一些清理工作无法完成，还有一个情况是执行stop方法后线程B会马上释放锁，这有可能会引发数据不同步问题。基于以上这些问题，stop()方法被抛弃了。

## 进程与线程

<https://blog.csdn.net/mxsgoden/article/details/8821936>

## Sleep与wait

<https://segmentfault.com/a/1190000016537670>

sleep 一般用于当前线程休眠，或者轮循暂停操作，wait 则多用于多线程之间的通信。

# 并发编程

## Volatile与synchronized

JVM内存模型：主内存和线程独立的工作内存

Java内存模型规定，对于多个线程共享的变量，存储在主内存当中，每个线程都有自己独立的工作内存（比如CPU的寄存器），线程只能访问自己的工作内存，不可以访问其他线程的工作内存。

工作内存中保存了主内存共享变量的副本，线程要操作这些共享变量，只能通过操作工作内存中的副本来实现，操作完毕之后再同步回到主内存当中

### volatile修饰的变量具有可见性

volatile是变量修饰符，其修饰的变量具有可见性。

可见性也就是说一旦某个线程修改了该被volatile修饰的变量，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，可以立即获取修改之后的值。

在Java中为了加快程序的运行效率，对一些变量的操作通常是在该线程的寄存器或是CPU缓存上进行的，之后才会同步到主存中，而加了volatile修饰符的变量则是直接读写主存。

### volatile禁止指令重排

指令重排是指处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证各个语句的执行顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

程序执行到volatile修饰变量的读操作或者写操作时，在其前面的操作肯定已经完成，且结果已经对后面的操作可见，在其后面的操作肯定还没有进行。

### Volatile原理

volatile的特殊规则就是：

read、load、use动作必须连续出现。

assign、store、write动作必须连续出现。

所以，使用volatile变量能够保证:

每次读取前必须先从主内存刷新最新的值。

每次写入后必须立即同步回主内存当中。

也就是说，volatile关键字修饰的

### synchronized

synchronized可作用于一段代码或方法，既可以保证可见性，又能够保证原子性。

可见性体现在：通过synchronized或者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中

原子性表现在：要么不执行，要么执行到底。

### 总结

<https://blog.csdn.net/SEU_Calvin/article/details/52370068>

（1）从而我们可以看出volatile虽然具有可见性但是并不能保证原子性。

（2）性能方面，synchronized关键字是防止多个线程同时执行一段代码，就会影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized。

但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。

## 乐观锁与悲观锁

https://blog.csdn.net/qq\_34337272/article/details/81072874

### 两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

### 乐观锁常见的两种实现方式

* 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功

* CAS算法

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试

## Synchronized的底层优化

https://www.cnblogs.com/paddix/p/5405678.html

### 偏向锁

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）

### 轻量级锁

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

### 适应性自旋（Adaptive Spinning）

从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

### 锁粗化（Lock Coarsening）

锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。

### 锁消除（Lock Elimination）

锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。

## 线程间的协作(wait/notify/sleep/yield/join)

<http://www.cnblogs.com/paddix/p/5381958.html>

## 死锁

### 死锁产生的原因

* 系统资源的竞争

系统资源的竞争导致系统资源不足，以及资源分配不当，导致死锁。

* 进程运行推进顺序不合适

进程在运行过程中，请求和释放资源的顺序不当，会导致死锁。

### 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，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。
* 请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
* 不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。
* 循环等待条件: 若干进程间形成首尾相接循环等待资源的关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

### 对待死锁的策略

(1) 死锁预防：破坏导致死锁必要条件中的任意一个就可以预防死锁。例如，要求用户申请资源时一次性申请所需要的全部资源，这就破坏了保持和等待条件；将资源分层，得到上一层资源后，才能够申请下一层资源，它破坏了环路等待条件。预防通常会降低系统的效率。

(2) 死锁避免：避免是指进程在每次申请资源时判断这些操作是否安全，例如，使用银行家算法。死锁避免算法的执行会增加系统的开销。

(3) 死锁检测：死锁预防和避免都是事前措施，而死锁的检测则是判断系统是否处于死锁状态，如果是，则执行死锁解除策略。

(4) 死锁解除：这是与死锁检测结合使用的，它使用的方式就是剥夺。即将某进程所拥有的资源强行收回，分配给其他的进程。

### 银行家算法

按照银行家算法的思想，当进程请求资源时，系统将按如下原则分配系统资源：

(1) 当一个进程对资源的最大需求量不超过系统中的资源数时可以接纳该进程。

(2) 进程可以分期请求资源，当请求的总数不能超过最大需求量。

(3) 当系统现有的资源不能满足进程尚需资源数时，对进程的请求可以推迟分配，但总能使进程在有限的时间里得到资源。

(4) 当系统现有的资源能满足进程尚需资源数时，必须测试系统现存的资源能否满足该进程尚需的最大资源数，若能满足则按当前的申请量分配资源，否则也要推迟分配。

## synchronized和lock

synchronized 是Java的关键字，是Java的内置特性，在JVM层面实现了对临界资源的同步互斥访问，通过对对象的头文件来操作，从而达到加锁和释放锁的目的。

1）不能响应中断；(线程的中断操作对于正在等待获取的锁对象的synchronized方法或者代码块并不起作用，也就是对于synchronized来说，如果一个线程在等待锁，那么结果只有两种，要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效)

2）同一时刻不管是读还是写都只能有一个线程对共享资源操作，其他线程只能等待

3）锁的释放由虚拟机来完成，不用人工干预，不过此即使缺点也是优点，优点是不用担心会造成死锁，缺点是由可能获取到锁的线程阻塞之后其他线程会一直等待，性能不高。

lock接口的提出就是为了完善synchronized的不完美的，首先lock是基于jdk层面实现的接口，和虚拟机层面不是一个概念；其次对于lock对象中的多个方法的调用，可以灵活控制对共享资源变量的操作，不管是读操作还是写操作

## 如何排查死锁

https://www.cnblogs.com/aflyun/p/9194104.html

### jps + jstack

jps : JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

Jstack : jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。

### jconsole

Jconsole（Java Monitoring and Management Console）是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控，是一个基于JMX（java management extensions）的GUI性能监测工具。jconsole使用jvm的扩展机制获取并展示虚拟机中运行的应用程序的性能和资源消耗等信息。

优势点在于：1. 可以远程可视化监控服务器的java进程；2.可以观测堆的不同区块的占用空间量（优于jVv默认功能）。

### Java Visual VM

可以heapdump。

# Java容器实现原理

## ConcurrentHashMap

<https://www.jianshu.com/p/e694f1e868ec>

## HashMap

<https://blog.csdn.net/tuke_tuke/article/details/51588156>

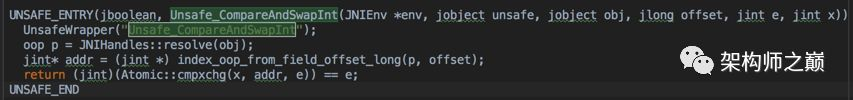
1.8改变了插入位置顺序，降低了形成循环链表的概率(是否完全消除暂时我还没法确定)。

<https://blog.csdn.net/carson_ho/article/details/79373134>

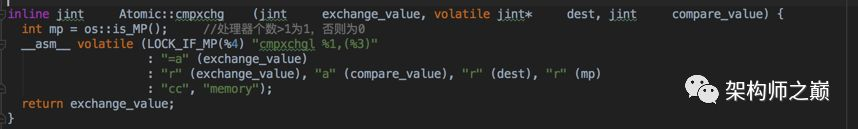
## JUC包中原子类使用及其原理

CAS+volatile关键字实现原子操作。

JAVA中的CAS操作都是通过sun包下Unsafe类实现，而Unsafe类中的方法都是native方法，由JVM本地实现。



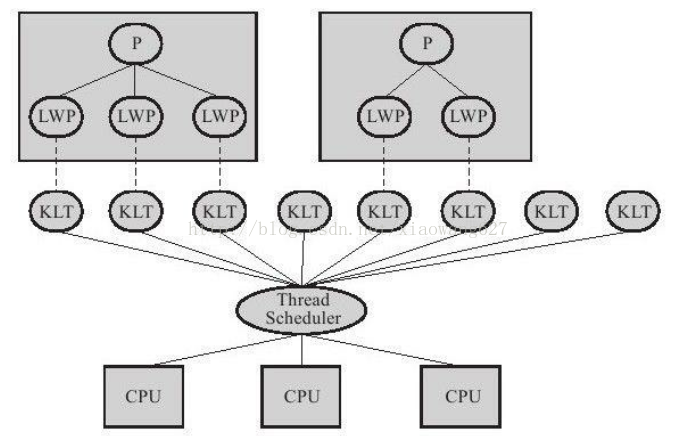
Unsafe中对CAS的实现是C++写的，从上图可以看出最后调用的是Atomic:comxchg这个方法，这个方法的实现放在hotspot下的os\_cpu包中，说明这个方法的实现和操作系统、CPU都有关系，我们以linux的X86处理器的实现为例来进行分析。



Linux的X86下主要是通过cmpxchgl这个指令在CPU级完成CAS操作的，但在多处理器情况下必须使用lock指令加锁来完成。当然不同的操作系统和处理器的实现会有所不同

# JVM

## 线程实现方式



这种实现方式线程直接由操作系统内核支持。程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程，轻量级进程与内核线程之间是1:1的关系，称为一对一的线程模型 。

## 双亲委派模型

<https://www.imooc.com/article/34493>

findClass（）用于写类加载逻辑、loadClass（）方法的逻辑里如果父类加载器加载失败则会调用自己的findClass（）方法完成加载，保证了双亲委派规则。

1、如果不想打破双亲委派模型，那么只需要重写findClass方法即可

2、如果想打破双亲委派模型，那么就重写整个loadClass方法

## 永久代与元空间

<https://aoyouzi.iteye.com/blog/2243929>

## OopMap、SafePoint(安全点)以及安全区域

### OopMap

 在正式的GC之前，要进行可达性分析来标记出将来可能要宣告死亡的对象。如果每次GC的时候都要遍历所有的引用，这样的工作量是非常大的。因为在可达性分析的时候要保证期间不发生引用关系的变化，所有执行线程要停顿等待，称为“Stop The World”，程序中的线程需要停止来配合可达性分析。

    所以，每次直接遍历整个引用链肯定是不现实的。 为了应对这种尴尬的问题，最早有保守式GC和后来的准确式GC。这里准确式GC就会提到一个OopMap，用来保存类型的映射表。

### SafePoint(安全点)以及安全区域

Safepoint保证了程序执行时，在不长的时间里就会遇到可进入GC的安全点，但如果线程没有分配cpu时间，必须线程处于sleep或blocked状态，就无法响应JVM的中断请求，走到安全点去挂起。Safe Region解决了这一问题。

    安全区域是指在一段代码片段中，引用关系不会发生变化，在该区域的任何地方发生GC都是安全的。当代码执行到安全区域时，首先标识自己已经进入了安全区域，那样如果在这段时间里JVM发起GC，就不用管标示自己在安全区域的那些线程了，在线程离开安全区域时，会检查系统是否正在执行GC，如果是，就等到GC完成后再离开安全区域。

## JVM空间分配担保(失败担保机制)

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间。如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlerPromotionFailure设置是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小。如果大于，将尝试着进行一次Monitor GC，尽管这次GC是有风险的。如果小于，或者HandlerPromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC了。

## 动态绑定的原理

<https://blog.csdn.net/huangrunqing/article/details/51996424>

动态绑定具体的调用过程为：

* 首先会找到被调用方法所属类的全限定名
* 在此类的方法表中寻找被调用方法，如果找到，会将方法表中此方法的索引项记录到常量池中（这个过程叫常量池解析），如果没有，编译失败。
* 根据具体实例化的对象找到方法区中此对象的方法表，再找到方法表中的被调用方法，最后通过直接地址找到字节码所在的内存空间。

方法表是实现动态调用的核心。上面讲过方法表存放在方法区中的类型信息中。为了优化对象调用方法的速度，方法区的类型信息会增加一个指针，该指针指向一个记录该类方法的方法表，方法表中的每一个项都是对应方法的指针。

# 分布式

## 实现分布式锁的几种方式

<https://www.cnblogs.com/austinspark-jessylu/p/8043726.html>

### 什么是分布式锁？

* 当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。
* 与单机模式下的锁不仅需要保证进程可见，还需要考虑进程与锁之间的网络问题。（我觉得分布式情况下之所以问题变得复杂，主要就是需要考虑到网络的延时和不可靠。。。一个大坑）
* 分布式锁还是可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存如 Redis、Memcache。至于利用数据库、文件等做锁与单机的实现是一样的，只要保证标记能互斥就行。

### 使用分布式锁的两个场景：

* 效率：使用分布式锁可以避免不同节点重复相同的工作，这些工作会浪费资源。比如用户付了钱之后有可能不同节点会发出多封短信。
* 正确性：加分布式锁同样可以避免破坏正确性的发生，如果两个节点在同一条数据上面操作，比如多个节点机器对同一个订单操作不同的流程有可能会导致该笔订单最后状态出现错误，造成损失。

# 数据库

## 事务隔离级别

具体描述各种异常

<https://www.jianshu.com/p/4e3edbedb9a8>

## **数据库事务**(Database Transaction)

**数据库事务**(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。 **事务**处理可以确保除非**事务**性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。

## Mysql慢查询优化

<https://blog.csdn.net/qq_35571554/article/details/82800463>

## 缓存更新策略

<https://www.jianshu.com/p/8950c52ce53b>

先更新数据库再删除缓存

## 缓存穿透、缓存击穿、缓存雪崩区别和解决方案

<https://blog.csdn.net/kongtiao5/article/details/82771694>

<https://blog.csdn.net/zeb_perfect/article/details/54135506>

### 缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

* 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
* 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

### 缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。

* 设置热点数据永远不过期。
* 加互斥锁

### 缓存雪崩

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

* 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
* 如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。
* 设置热点数据永远不过期。

# 计算机网络

## TCP与UDP区别

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付

Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4.每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

5、TCP对系统资源要求较多，UDP对系统资源要求较少。

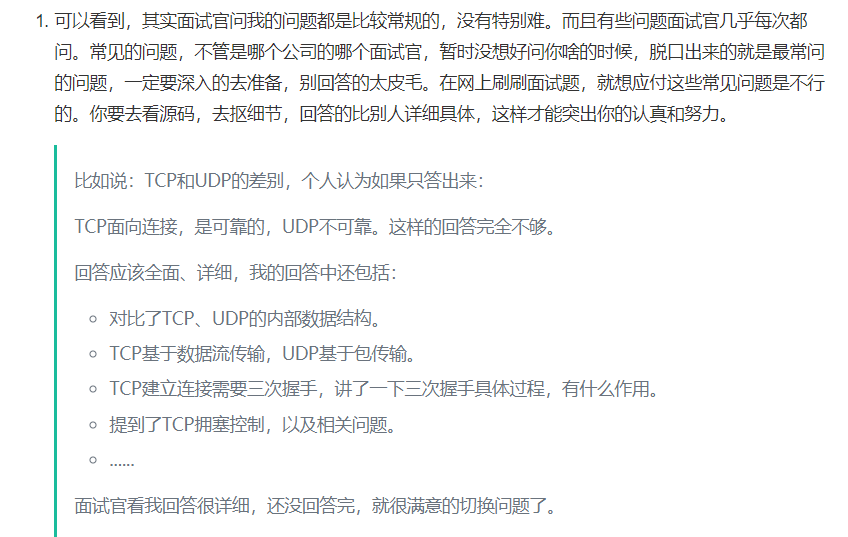
### 为什么UDP有时比TCP更有优势?

UDP以其简单、传输快的优势，在越来越多场景下取代了TCP,如实时游戏。

（1）网速的提升给UDP的稳定性提供可靠网络保障，丢包率很低，如果使用应用层重传，能够确保传输的可靠性。

（2）TCP为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了繁琐的握手过程，由于TCP内置的系统协议栈中，极难对其进行改进。

采用TCP，一旦发生丢包，TCP会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大，基于UDP对实时性要求较为严格的情况下，采用自定义重传机制，能够把丢包产生的延迟降到最低，尽量减少网络问题对游戏性造成影响。



## TCP粘包，拆包及解决方法

### 背景

UDP是基于报文发送的，从UDP的帧结构可以看出，在UDP首部采用了16bit来指示UDP数据报文的长度，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题。而TCP是基于字节流的，虽然应用层和TCP传输层之间的数据交互是大小不等的数据块，但是TCP把这些数据块仅仅看成一连串无结构的字节流，没有边界；另外从TCP的帧结构也可以看出，在TCP的首部没有表示数据长度的字段，基于上面两点，在使用TCP传输数据时，才有粘包或者拆包现象发生的可能。

### 原因

发生TCP粘包或拆包有很多原因，现列出常见的几点

1、要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。

2、待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

3、要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。

4、接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

等等。

### 解决方法

1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

2、发送端将每个数据包封装为固定长度（不够的可以通过补0填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

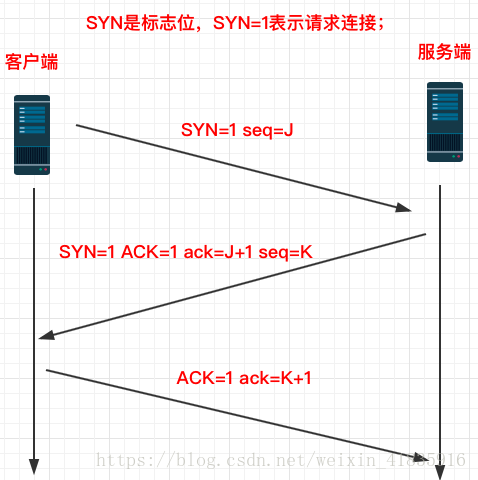
## 网络分层及各层协议



## 三次握手

三次握手的目的是建立可靠的通信信道。确认自己与对方的发送与接收机能正常。

TCP三次握手过程:



为什么需要三次握手：

三次握手能确认双发收发功能都正常，缺一不可。

第一次握手：Client什么都不能确认；Server确认了对方发送正常。

第二次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己接收正常，对方发送正常。

第三次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己发送、接收正常，对方发送接收正常。

三次握手的另一个目标是确认确认双方都支持TCP，告知对方用TCP传输。

第一次握手：Server猜测Client可能要建立TCP请求，但不确定，因为也可能是Client乱发了一个数据包给自己。

第二次握手：通过ack=J+1，Client知道Server是支持TCP的，且理解了自己要建立TCP连接的意图。

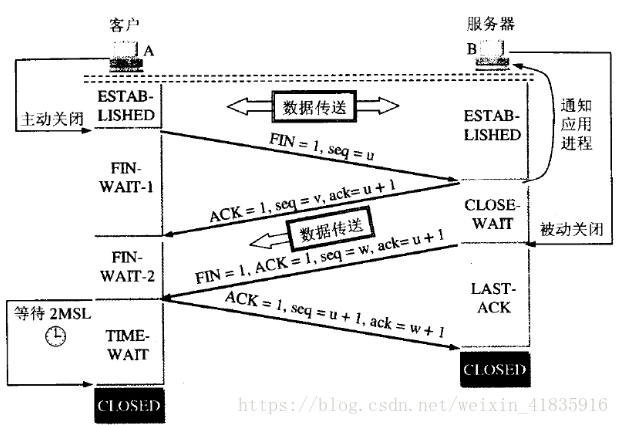
第三次握手：通过ack=K+1，Server知道Client是支持TCP的，且确实是要建立TCP连接。

## 四次挥手

1、第一次挥手：主机1（可以使客户端，也可以是服务器端），设置Sequence Number和Acknowledgment Number，向主机2发送一个FIN报文段；此时，主机1进入FIN\_WAIT\_1状态；这表示主机1没有数据要发送给主机2了。

2、第二次挥手：主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1；主机1进入FIN\_WAIT\_2状态；主机2告诉主机1，我“同意”你的关闭请求。

3、第三次挥手：主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST\_ACK状态。

4、第四次挥手：主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME\_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。 

为什么要time wait，为什么是2msl( Maximum Segment Lifetime)

为了保证A发送的最后一个ACK报文能够到达B。这个ACK报文段有可能丢失，因而使处在LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认。B会超时重传这个FIN+ACK报文段，而A就能在2MSL时间内收到这个重传的FIN+ACK报文段。如果A在TIME-WAIT状态不等待一段时间，而是在发送完ACK报文段后就立即释放连接，就无法收到B重传的FIN+ACK报文段，因而也不会再发送一次确认报文段。这样，B就无法按照正常的步骤进入CLOSED状态。

MSL指的是任何IP数据报能够在因特网上存活的最长时间。假设现在一个MSL的时候，接收端需要发送一个应答，这时候，我们也必须等待这个应答的消失，这个应答的消失也是需要一个MSL，所以我们需要等待2MSL。

## TCP如何保证可靠性

<https://uule.iteye.com/blog/2429131>

* **校验和**

发送的数据包的二进制相加然后取反，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。

* **确认应答+序列号**

TCP给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

* **超时重传**

当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

* **流量控制**

TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP使用的流量控制协议是可变大小的滑动窗口协议。

接收方有即时窗口（滑动窗口），随ACK报文发送

* **拥塞控制**

当网络拥塞时，减少数据的发送。

发送方有拥塞窗口，发送数据前比对接收方发过来的即使窗口，取小

慢启动、拥塞避免、拥塞发送、快速恢复。

## http完整流程

<https://www.cnblogs.com/xuzekun/p/7527736.html>

1.对www.baidu.com这个网址进行DNS域名解析，得到对应的IP地址

2.根据这个IP，找到对应的服务器，发起TCP的三次握手

3.建立TCP连接后发起HTTP请求

4.服务器响应HTTP请求，浏览器得到html代码

5.浏览器解析html代码，并请求html代码中的资源（如js、css图片等）（先得到html代码，才能去找这些资源）

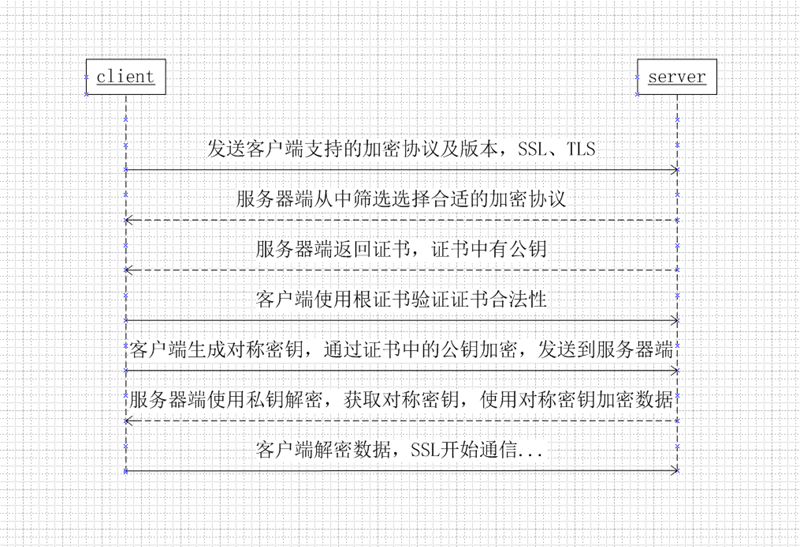
6.浏览器对页面进行渲染呈现给用户

注：1.DNS域名解析采用的是递归查询的方式，过程是，先去找DNS缓存->缓存找不到就去找根域名服务器->根域名又会去找下一级，这样递归查找之后，找到了，给我们的web浏览器

2.为什么HTTP协议要基于TCP来实现？ TCP是一个端到端的可靠的面相连接的协议，HTTP基于传输层TCP协议不用担心数据传输的各种问题（当发生错误时，会重传）

3.最后一步浏览器是如何对页面进行渲染的？ a）解析html文件构成 DOM树，b）解析CSS文件构成渲染树， c）边解析，边渲染 ， d）JS 单线程运行，JS有可能修改DOM结构，意味着JS执行完成前，后续所有资源的下载是没有必要的，所以JS是单线程，会阻塞后续资源下载。

## https建立连接过程



## Tcp心跳、应用层心跳

<https://www.zhihu.com/question/35013918/answer/63664974>

## Tcp拥塞控制、流量控制

<https://zhuanlan.zhihu.com/p/37379780>

拥塞控制：拥塞控制是作用于网络的，它是防止过多的数据注入到网络中，避免出现网络负载过大的情况；常用的方法就是：（ 1 ）慢开始、拥塞避免（ 2 ）快重传、快恢复。

流量控制：流量控制是作用于接收者的，它是控制发送者的发送速度从而使接收者来得及接收，防止分组丢失的。

# 操作系统

## 进程间通信

<https://www.jianshu.com/p/c1015f5ffa74>

* 匿名管道(pipe)
* 有名管道(FIFO)
* 信号(Signal)
* 消息(Message)队列
* 共享内存(share memory)
* 信号量(semaphore)
* 套接字(socket)

## Epoll,poll,select

<https://www.cnblogs.com/jeakeven/p/5435916.html>

## 操作系统中进程调度策略

FCFS(先来先服务，队列实现，非抢占的)：先请求CPU的进程先分配到CPU

SJF(最短作业优先调度算法)：平均等待时间最短，但难以知道下一个CPU区间长度

优先级调度算法(可以是抢占的，也可以是非抢占的)：优先级越高越先分配到CPU，相同优先级先到先服务，存在的主要问题是：低优先级进程无穷等待CPU，会导致无穷阻塞或饥饿；解决方案：老化

时间片轮转调度算法(可抢占的)：队列中没有进程被分配超过一个时间片的CPU时间，除非它是唯一可运行的进程。如果进程的CPU区间超过了一个时间片，那么该进程就被抢占并放回就绪队列。

多级队列调度算法：将就绪队列分成多个独立的队列，每个队列都有自己的调度算法，队列之间采用固定优先级抢占调度。其中，一个进程根据自身属性被永久地分配到一个队列中。

多级反馈队列调度算法：与多级队列调度算法相比，其允许进程在队列之间移动：若进程使用过多CPU时间，那么它会被转移到更低的优先级队列；在较低优先级队列等待时间过长的进程会被转移到更高优先级队列，以防止饥饿发生。

# 算法

## LRU算法

https://blog.csdn.net/elricboa/article/details/78847305

LRU全称是Least Recently Used，即最近最久未使用的意思。

LRU算法的设计原则是：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

### 实现LRU

1.用一个数组来存储数据，给每一个数据项标记一个访问时间戳，每次插入新数据项的时候，先把数组中存在的数据项的时间戳自增，并将新数据项的时间戳置为0并插入到数组中。每次访问数组中的数据项的时候，将被访问的数据项的时间戳置为0。当数组空间已满时，将时间戳最大的数据项淘汰。

2.利用一个链表来实现，每次新插入数据的时候将新数据插到链表的头部；每次缓存命中（即数据被访问），则将数据移到链表头部；那么当链表满的时候，就将链表尾部的数据丢弃。

3.利用链表和hashmap。当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。

对于第一种方法，需要不停地维护数据项的访问时间戳，另外，在插入数据、删除数据以及访问数据时，时间复杂度都是O(n)。对于第二种方法，链表在定位数据的时候时间复杂度为O(n)。所以在一般使用第三种方式来是实现LRU算法。

## 解决哈希冲突的方法

通过哈希函数去计算哈希值，难免会有冲突的时候，解决冲突的方法有如下几种：

* 开放定址法：依靠数组中的空位解决碰撞冲突
  + 线性探测法：直接检测散列表的下一个位置（即索引值加1），如果仍冲突，继续
  + 二次探测法：即H + 1^2, H + 2^2, H + 3^2…
  + 伪随机探测
* 再哈希法：使用多个哈希函数，第一个冲突时，使用第二个哈希函数，知道不冲突为止；
* 链地址法：将所有哈希地址相同的关键字，都链接到同一个链表中

## 用rand1To M实现等概率随机产生1~N的随机函数rand1To N

<https://blog.csdn.net/qq_24690761/article/details/50839592>

1-m 等概率 转化成 等1-n概率 问题

/实质就是看 把 m 看成m进制 n<m 则 m进制能装下n

/ m>n 扩充成m^2 也就是增加一位 看能不能放下n 以此类推