技术文档

# Linux

## install

### rpm安装方式

#### wget安装

下载rpm包

<http://mirrors.163.com/centos/6.2/os/x86_64/Packages/wget-1.12-1.4.el6.x86_64.rpm>

rpm安装

rpm ivh wget-1.12-1.4.el6.x86\_64.rpm

### yum安装方式

#### wget安装

yum -y install wget

## Shell

### 系统信息

arch ##显示机器的[处理器架构](https://www.baidu.com/s?wd=%E5%A4%84%E7%90%86%E5%99%A8%E6%9E%B6%E6%9E%84&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

uname -m ##显示机器的[处理器架构](https://www.baidu.com/s?wd=%E5%A4%84%E7%90%86%E5%99%A8%E6%9E%B6%E6%9E%84&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

uname -r ##显示正在使用的[内核版本](https://www.baidu.com/s?wd=%E5%86%85%E6%A0%B8%E7%89%88%E6%9C%AC&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)

cat /proc/cpuinfo ##显示CPU info的信息

cat /proc/cpuinfo | grep -E 'processor|model name'

[grep](#_grep)正则表达式获取processor和model name信息

cat /proc/interrupts ##显示中断

cat /proc/meminfo ##校验内存使用

cat /proc/swaps ##显示哪些swap被使用

cat /proc/version ##显示内核的版本

cat /proc/net/dev ##显示[网络适配器](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E9%80%82%E9%85%8D%E5%99%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9nHN9nvc1nHbkPhf3rH0s0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErHD4n1fdn1R4Pj6vnWc3n1n4)及统计

cat /proc/mounts ##显示已加载的文件系统

### 关机

halt ##立刻关机

poweroff ##立刻关机

shutdown -h now ##立刻关机（root用户使用）

shutdown -h 10 ##10分钟后自动关机

### 用户管理

#### 创建用户

useradd 用户名

-c comment 指定一段注释性描述。

-d 目录 指定用户主目录，如果此目录不存在，则同时使用-m选项，可以创建主目录。

-g 用户组 指定用户所属的用户组。

-G 用户组，用户组 指定用户所属的附加组。

-s Shell文件 指定用户的登录Shell。

-u 用户号 指定用户的用户号，如果同时有-o选项，则可以重复使用其他用户的标识号。

#### 修改用户密码

passwd 用户名

#### 设置用户密码

passwd 用户名

#### 删除用户

userdel 用户名

#### 查看所有用户

cat /etc/passwd | awk -F : '{print $1}'

### 查找

#### grep

## 权限

# Java

## JVM(Java虚拟机)

### 运行时数据区



#### 所有线程共享的数据区（堆空间线程共享）

##### 方法区(Method Area)

存放被虚拟机加载的类信息、常量、静态变量、编译后的代码数据，也称为Permanent Generation。

-XX:MaxPermSize=16m ##设置持久代最大为16m。

##### 堆(Heap)

存放对象实例

-Xmx3500m ##设置堆最大可用内存。

-Xms3500m ##设置堆最小可用内存。

-Xmn2g

##设置年轻代大小（整个堆大小=年轻代大小+年老代大小+持久代大小）。

##持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。

##此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-XX:NewRatio=4

##设置年轻代（包括Eden和两个Survivor区）与年老代的比值（不包括持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5。

-XX:SurvivorRatio=4

##设置年轻代中Eden区与Survivor区的大小比值。

设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6。

-XX:MaxTenuringThreshold=0

##设置垃圾最大年龄。

如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代 。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间 ，增加在年轻代即被回收的概率。

#### 线程隔离的数据区（栈空间线程私有）

生命周期与线程相同。

##### 虚拟机栈(Virtual Machine Stack)

Java方法执行的内存模型。

Stack Frame(栈帧)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放编译可知的各种基本数据类型、对象引用类型和returnAddress类型(指向一条字节码指令的地址)。

long和double占用2个局部变量空间(Slot)，其余占用1个。

局部变量表所需内存空间在编译期间完成分配，当进入方法时，这个方法需要在帧中分配多大局部变量空间是完全确定的，方法在运行期间不会改变局部变量表的大小。

如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

-Xss128k

##设置每个线程的堆栈大小。

##根据应用的线程所需内存大小进行调整。

##在相同物理内存下，减小这个值能生成更多的线程。

##但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

##### 本地方法栈(Native Method Stack)

与虚拟机栈类似。

虚拟机栈为虚拟机执行Java方法（字节码）服务。

本地方法栈为虚拟机使用的Native方法服务。

有的虚拟机(例如Sun HotSpot)直接把本地方法栈和虚拟机栈合二为一。

##### 程序计数器(Program Counter Register)

当前线程所执行的字节码的行号指示器，记录正在执行的虚拟机字节码指令的地址，如果正在执行Native方法，这个计数器值为空(Undefined)

唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

各个线程之间计数器互不影响，独立存储。

### 垃圾回收算法

#### 标记-清除算法(Mark-Sweep)

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经介绍过了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，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(Copying)

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它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。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是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。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块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”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

#### 标记-整理算法(Mark-Compact)

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，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。

#### 分代收集算法(Generational Collection)

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

### 垃圾回收器

#### 串行回收器

只适用于小数据量的情况。

#### 并行回收器（吞吐量优先）

并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

XX:+UseParallelGC

##选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20

##配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

-XX:+UseParallelOldGC

##配置年老代垃圾收集方式为并行收集。

-XX:MaxGCPauseMillis=100

##设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

-XX:+UseAdaptiveSizePolicy

##设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

#### 并发回收器(响应时间优先)

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

-XX:+UseConcMarkSweepGC

##CMS(Concurrent Mark-Sweep)

##设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC

##设置年轻代为并行收集。可与CMS收集同时使用。jdk5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

-XX:CMSFullGCsBeforeCompaction=5

##由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection

##打开对年老代的压缩。可能会影响性能，但是可以消除碎片。

### 辅助信息

#### -XX:+PrintGC

##输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs]

                [Full GC 121376K->10414K(130112K), 0.0650971 secs]

#### -XX:+PrintGCDetails

##输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]

                [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

#### -XX:+PrintGCTimeStamps

##可与上面两个混合使用。

##输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

#### -XX:+PrintGCApplicationConcurrentTime

##打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用

##输出形式：Application time: 0.5291524 seconds

#### -XX:+PrintGCApplicationStoppedTime

##打印垃圾回收期间程序暂停的时间。可与上面混合使用

输出形式：Total time for which application threads were stopped: 0.0468229 seconds

#### -XX:PrintHeapAtGC

##打印GC前后的详细堆栈信息

#### -Xloggc:filename

##与上面几个配合使用，把相关日志信息记录到文件以便分析。

### Eclipse Memory Analysis

#### 安装

在Eclipse help -> Eclipse Marketplace下搜索Memory

## 集合

### HashMap的实现原理

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

#### HashMap的数据结构

在Java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

#### HashMap的存取实现

##### 存储

当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标）， 如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

##### 读取

从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

#### 归纳

简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，  
也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

## 多线程(Thread)

### 进程与线程的区别

进程：每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销，一个进程包含1--n个线程。（进程是资源分配的最小单位）。

线程：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。（线程是cpu调度的最小单位）。

线程和进程一样分为五个阶段：创建、就绪、运行、阻塞、终止。

多进程是指操作系统能同时运行多个任务（程序）。

多线程是指在同一程序中有多个顺序流在执行。

### Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

总结：

实现Runnable接口比继承Thread类所具有的优势：

适合多个相同的程序代码的线程去处理同一个资源。

可以避免java中的单继承的限制。

增加程序的健壮性，代码可以被多个线程共享，代码和数据独立。

线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类。

### 线程状态转换



**新建状态（New）**：新创建了一个线程对象。

**就绪状态（Runnable）**：线程对象创建后，其他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，变得可运行，等待获取CPU的使用权。

**运行状态（Running）**：就绪状态的线程获取了CPU，执行程序代码。

**阻塞状态（Blocked）**：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、**等待阻塞**：运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)

（二）、**同步阻塞**：运行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中。

（三）、**其他阻塞**：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。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，线程重新转入就绪状态。（注意,sleep是不会释放持有的锁）

**死亡状态（Dead）**：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

### 线程调度

#### 线程优先级

Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

MAX\_PRIORITY

          ##线程可以具有的最高优先级，取值为10。

MIN\_PRIORITY

          ##线程可以具有的最低优先级，取值为1。

NORM\_PRIORITY

          ##分配给线程的默认优先级，取值为5。

#### Thread.sleep

线程睡眠，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态，**不释放锁**。

#### Object.wait

线程等待，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法。**wait释放锁，必须在对象锁代码块里调用,否则会抛出java.lang.IllegalMonitorStateException**。

#### Object.notify(Object.notifyAll)

Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。**notify和notifyAll()必须在对象锁代码块里调用,否则会抛出java.lang.IllegalMonitorStateException**。

#### Thread.yield

线程让步，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

yield应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

#### Thread.join

线程加入，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

join是Thread类的一个方法，启动线程后直接调用，即join()的作用是：“等待该线程终止”，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子线程结束了才能执行。

#### Thread.****interrupt****

不要以为它是中断某个线程！它只是向线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出异常，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

#### ****wait和sleep区别****

共同点：

他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

wait()和sleep()都可以通过interrupt()方法 打断线程的暂停状态 ，从而使线程立刻抛出InterruptedException。

如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep/join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出InterruptedException 。

不同点：

Thread类的方法：sleep(),yield()等

Object的方法：wait()和notify()等

每个对象都有一个锁来控制同步访问。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

sleep，wait必须捕获异常，而notify和notifyAll不需要捕获异常。

所以sleep()和wait()方法的最大区别是：

　　　　sleep()睡眠时，保持对象锁，仍然占有该锁；

　　　　而wait()睡眠时，释放对象锁。

　　但是wait()和sleep()都可以通过interrupt()方法打断线程的暂停状态，从而使线程立刻抛出InterruptedException（但不建议使用该方法）。

sleep()方法

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CUP的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会;

　　 sleep()是Thread类的Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个Synchronized块中调用Sleep()方法是，线程虽然休眠了，但是对象的机锁并木有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。

　　在sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级。

wait（）方法

wait()方法是Object类里的方法；当一个线程执行到wait()方法时，它就进入到一个和该对象相关的等待池中，同时失去（释放）了对象的机锁（暂时失去机锁，wait(long timeout)超时时间到后还需要返还对象锁）；其他线程可以访问；

　　wait()使用notify或者notifyAlll或者指定睡眠时间来唤醒当前等待池中的线程。

　　wiat()必须放在synchronized block中，否则会在program runtime时扔出java.lang.IllegalMonitorStateException异常。

## Spring Boot

### Spring Boot四个核心

#### 自动配置

#### 起步依赖

#### 命令行界面(Spring Boot CLI)

#### Actuator

Actuator提供运行时检视应用程序内部情况的能力，包括以下细节:

Spring应用程序上下文里配置的Bean。

Spring Boot自动配置做的决策。

应用程序取到的环境变量、系统属性、配置属性和命令行参数。

应用程序里线程的当前状态。

应用程序最近处理过的HTTP请求的追踪情况。

各种和内存用量、垃圾回收、Web请求以及数据源用量相关的指标。

Actuator通过Web端点和Shell界面向外界提供信息。

## Maven

## Gradle

# Server

## Nginx

### 简介

Nginx("engine x")是一款是由俄罗斯的程序设计师Igor Sysoev所开发高性能的 Web和 反向代理 服务器，也是一个 IMAP/POP3/SMTP 代理服务器。

在高连接并发的情况下，Nginx是Apache服务器不错的替代品。

### 安装

#### 安装编译工具及库文件

yum -y install make zlib zlib-devel gcc-c++ libtool openssl openssl-devel

#### 安装PCRE

PCRE 作用是让 Nginx 支持 Rewrite 功能。

wget <http://downloads.sourceforge.net/project/pcre/pcre/8.35/pcre-8.35.tar.gz>

tar zxvf pcre-8.35.tar.gz

cd pcre-8.35

./configure

make && make install

pcre-config –version

#### 安装Nginx

wget <http://nginx.org/download/nginx-1.6.2.tar.gz>

tar zxvf nginx-1.6.2.tar.gz

cd nginx-1.6.2

./configure --prefix=../nginx --with-http\_stub\_status\_module --with-http\_ssl\_module --with-pcre=../pcre-8.35

make && make install ##编译安装

$home/nginx/sbin/nginx -v ##查看nginx版本

### nginx.conf配置

$home/nginx/conf/nginx.conf

user nginx nginx; ##Nginx用户及组：用户 组。window下不指定。

worker\_processes 8; ##工作进程：数目。根据硬件调整，通常等于CPU数量或者2倍于CPU。

error\_log logs/error.log; ##错误日志存放路径。

error\_log logs/error.log notice; ##错误日志存放路径。

error\_log logs/error.log info; ##错误日志存放路径。

pid logs/nginx.pid; ##pid（进程标识符）存放路径。

http

{

server

{

listen 80; ##监听端口

server\_name localhost; ##域名

index index.html index.htm index.php;

root /usr/local/webserver/nginx/html; ##站点目录

#######location前缀含义:#################

## = ：精确匹配（必须全部相等）。 ##

## ~ ：大小写敏感。 ##

## ~\* ：忽略大小写。 ##

## ^~ ：只需匹配uri部分。 ##

## @ ：内部服务跳转。 ##

location [ = | ~ | ~\* | ^~ ] uri {

}

}

}

#### location

基础知识:

location 是在 server 块中配置。

可以根据不同的 URI 使用不同的配置（location 中配置），来处理不同的请求。

location 是有顺序的，会被第一个匹配的location 处理。

配置语法:

**location** **[ = | ~ | ~\* | ^~ ] uri** { ... }

**location** **@name** { ... }

前缀含义:

**=** **##精确匹配**

location = / {

#规则

}

# 则匹配到 `http://www.example.com/` 这种请求。

**~ ##大小写敏感**

location ~ /Example/ {

#规则

}

#请求示例

#http://www.example.com/Example/ [成功]

#http://www.example.com/example/ [失败]

**~\* ##大小写忽略**

location ~\* /Example/ {

#规则

}

# 则会忽略 uri 部分的大小写

#http://www.example.com/Example/ [成功]

#http://www.example.com/example/ [成功]

**^~ ##只匹配以 uri 开头**

location ^~ /img/ {

#规则

}

#以 /img/ 开头的请求，都会匹配上

#http://www.example.com/img/a.jpg [成功]

#http://www.example.com/img/b.mp4 [成功]

**@ ##nginx内部跳转**

location /img/ {

error\_page 404 @img\_err;

}

location @img\_err {

##规则

}

**##**以 /img/ 开头的请求，如果链接的状态为 404。则会匹配到 @img\_err 这条规则上。

### 启动Nginx

$home/nginx/sbin/nginx

### 重新载入配置文件

$home/nginx/sbin/nginx -s reload

### 重启Nginx

$home/nginx/sbin/nginx -s reopen

### 停止Nginx

$home/nginx/sbin/nginx -s stop

# 数据库(database)

## 数据库事务

### 四大特性(ACID)

#### 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

#### 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

#### 隔离性（Isolation）

事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

不考虑事务的隔离性，会发生的几种问题:

##### 脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

##### 不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

##### 幻读

幻读，因为事务1读取的数据状态并不能支持他的下一步的业务，见鬼了一样。

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

#### 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

### 隔离级别

在MySQL数据库中，支持下面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

#### Serializable（串行化）

可避免脏读、不可重复读、幻读的发生。

#### Repeatable read（可重复读）

可避免脏读、不可重复读的发生。

#### Read committed（读已提交）

可避免脏读的发生。

#### Read uncommitted（读未提交）

最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。

## 数据库优化

### 百万级数据库优化方案

#### 避免全表扫描

对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

#### 避免null值判断

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

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。

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

#### 避免使用!=或<>操作符

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。

#### 避免使用or连接条件

应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，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 Name = 'admin'

可以这样查询：

select id from t where num = 10

union all

select id from t where Name = 'admin'

#### 慎用in和not in条件

in 和 not in 也要慎用，否则会导致全表扫描，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

很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

#### 慎用like条件

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。

#### 避免在where子句中使用参数

如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num = @num

可以改为[强制使用索引](#_强制使用索引)：

select id from t with(index(索引名)) where num = @num

select /\*+ index(table IndexName)\*/\* from t where num = @num

#### 避免在 where 子句中对字段进行表达式操作

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。如：

select id from t where num/2 = 100

应改为:

select id from t where num = 100\*2

#### 避免在where子句中对字段进行函数操作

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。如：

select id from t where substring(name,1,3) = ’abc’ -–name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′) = 0 -–‘2005-11-30’ --生成的id

应改为:

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'

#### 避免在where 子句中的“=”左边进行函数、算术运算或其他表达式运算

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。

#### 复合索引必须使用该索引的第一个字段

在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

#### 先分页再join

对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

#### 索引并不是越多越好

索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

#### 应尽可能的避免更新[聚合](http://www.baidu.com/link?url=f3MEUUvzoVZ8wfzYsWu0b5p1u-9aoTrl4VsYr8Nfk_hZvTOTsaPCYmLgzGXSjskWfDGBBc_M11UZOSV7mkQN-K)(clustered)索引数据列

因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

#### 尽量使用数字型字段

若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

#### 尽可能的使用 varchar/nvarchar 代替 char/nchar

因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

#### 避免使用select \*

任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

#### 尽量使用表变量来代替临时表

如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

#### 避免频繁创建和删除临时表

避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

#### 新建临时表使用 select into 代替 create table

在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替create table，避免造成大量log，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

#### 临时表先 truncate table，然后 drop table

如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

#### 尽量避免使用游标

因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

#### 基于集的解决方案

使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

#### 游标并不是不可使用

与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

#### SET NOCOUNT ON和SET NOCOUNT OFF的使用

在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

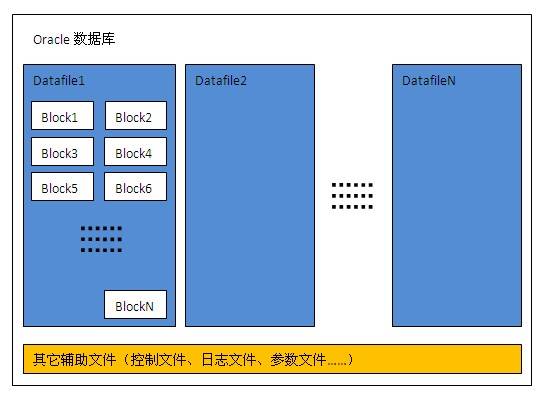
#### 尽量避免大事务操作

尽量避免大事务操作，提高系统并发能力。

## Oracle

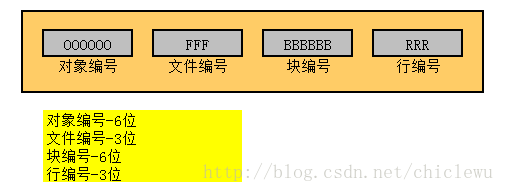
### 数据块(Block)

数据块是数据库中数据在磁盘中存储的最小单位，也是一次IO访问的最小单位，一个数据块通常可以存储多条记录，数据块大小是DBA在创建数据库或表空间时指定，可指定为2K、4K、8K、16K或32K字节。下图是一个[Oracle](http://lib.csdn.net/base/oracle)数据库典型的物理结构，一个数据库可以包括多个数据文件，一个数据文件内又包含多个数据块:



### ROWID

ROWID是每条记录在数据库中的唯一标识，通过ROWID可以直接定位记录到对应的文件号及数据块位置。ROWID内容包括对像号、文件号、数据块号、记录槽号，如下图所示：



### 索引

#### 强制使用索引

强制索引语法：SELECT /\*+index(table IndexName)\*/\* from table

如果查询语句中，表用到别名时，上边语句中的table一定要是别名，否则不走执行强制索引。

#### 索引使用条件

INDEX\_COLUMN = ?

INDEX\_COLUMN > ?

INDEX\_COLUMN >= ?

INDEX\_COLUMN < ?

INDEX\_COLUMN <= ?

INDEX\_COLUMN between ? and ?

INDEX\_COLUMN in (?,?,...,?)

INDEX\_COLUMN like ?||'%'（后导模糊查询）

T1. INDEX\_COLUMN=T2. COLUMN1（两个表通过索引字段关联）

#### 索引不使用条件

|  |  |
| --- | --- |
| 查询条件 | 不能使用索引原因 |
| INDEX\_COLUMN <> ?  INDEX\_COLUMN not in (?,?,...,?) | 不等于操作不能使用索引 |
| function(INDEX\_COLUMN) = ?  INDEX\_COLUMN + 1 = ?  INDEX\_COLUMN || 'a' = ? | 经过普通运算或函数运算后的索引字段不能使用索引 |
| INDEX\_COLUMN like '%'||?  INDEX\_COLUMN like '%'||?||'%' | 含前导模糊查询的Like语法不能使用索引 |
| INDEX\_COLUMN is null | B-TREE索引里不保存字段为NULL值记录，因此IS NULL不能使用索引 |
| NUMBER\_INDEX\_COLUMN='12345'  CHAR\_INDEX\_COLUMN=12345 | Oracle在做数值比较时需要将两边的数据转换成同一种数据类型，如果两边数据类型不同时会对字段值隐式转换，相当于加了一层函数处理，所以不能使用索引。 |
| a.INDEX\_COLUMN=a.COLUMN\_1 | 给索引查询的值应是已知数据，不能是未知字段值。 |

##经过函数运算的字段要使用索引可以使用**函数索引**，这种需求建议与DBA沟通。

有时候我们会使用多个字段的组合索引，如果查询条件中第一个字段不能使用索引，那整个查询也不能使用索引。

如：我们company表建了一个id+name的组合索引，以下SQL是不能使用索引的

select \* from company where name=?

Oracle9i后引入了一种index skip scan的索引方式来解决类似的问题，但是通过index skip scan提高性能的条件比较特殊，使用不好反而性能会更差。

#### 什么字段需要索引

主键及外键通常都要有索引。

字段出现在查询条件中，并且查询条件可以使用索引。

语句执行频率高，一天会有几千次以上。

通过字段条件可筛选的记录集很小，那数据筛选比例是多少才适合。

这个没有固定值，需要根据表数据量来评估，以下是经验公式，可用于快速评估：

小表(记录数小于10000行的表)：筛选比例<10%。

大表：(筛选返回记录数)<(表总记录数\*单条记录长度)/10000/16。

单条记录长度≈字段平均内容长度之和+字段数\*2。

以下是一些字段是否需要建B-TREE索引的经验分类：

|  |  |  |
| --- | --- | --- |
|  | 字段类型 | 常见字段名 |
| 需要建索引的字段 | 主键 | ID,PK |
| 外键 | PRODUCT\_ID,COMPANY\_ID,MEMBER\_ID,  ORDER\_ID,TRADE\_ID,PAY\_ID |
| 有对象或身份标识意义字段 | HASH\_CODE,USERNAME,IDCARD\_NO,  EMAIL,TEL\_NO,IM\_NO |
| 索引慎用字段,需要进行数据分布及使用场景详细评估 | 日期 | GMT\_CREATE,GMT\_MODIFIED |
| 年月 | YEAR,MONTH |
| 状态标志 | PRODUCT\_STATUS,ORDER\_STATUS,  IS\_DELETE,VIP\_FLAG |
| 类型 | ORDER\_TYPE,IMAGE\_TYPE,GENDER,  CURRENCY\_TYPE |
| 区域 | COUNTRY,PROVINCE,CITY |
| 操作人员 | CREATOR,AUDITOR |
| 数值 | LEVEL,AMOUNT,SCORE |
| 长字符 | ADDRESS,COMPANY\_NAME,  SUMMARY,SUBJECT |
| 不适合建索引的字段 | 描述备注 | DESCRIPTION,REMARK,MEMO,DETAIL |
| 大字段 | FILE\_CONTENT,EMAIL\_CONTENT |

#### B-TREE索引

B-TREE索引也称为平衡树索引(Balance Tree)，它是一种按字段排好序的树形目录结构，主要用于提升查询性能和唯一约束支持。B-TREE索引的内容包括**根节点**、**分支节点**、**叶子节点**。

**叶子节点内容**：索引字段内容+表记录ROWID。

**根节点，分支节点内容**：当一个数据块中不能放下所有索引字段数据时，就会形成树形的根节点或分支节点，根节点与分支节点保存了索引树的顺序及各层级间的引用关系。



##### 复合索引(组合索引)

一个索引由多个字段组成，称为组合索引。

对于复合索引，在Where限制条件中出现所有被索引的列时，优化器会选择走索引；

对于复合索引，在Where限制条件中出现联合索引中前导的列，即创建联合索引时前导的列时，优化器会选择走索引；

对于创建复合索引时，应考虑被索引字段的优先顺序，应将经常作为限制条件的字段放在首位；重复值少，即高基数(high-cardinaltiy)的列往前靠，而重复值多的字段往后靠；

##### 反向索引

##### 函数索引

#### 位图索引

#### 全文索引

### 执行计划

#### 执行计划常用列字段解释

基数（Rows） ##Oracle估计的当前操作的返回结果集行数。

字节（Bytes） ##执行该步骤后返回的字节数。

耗费（COST）、CPU耗费 ##Oracle估计的该步骤的执行成本，用于说明SQL执行的代价，理论上越小越好（该值可能与实际有出入）。

时间（Time） ##Oracle估计的当前操作所需的时间。

#### 表访问的几种方式

##### TABLE ACCESS FULL（全表扫描）

Oracle会读取表中所有的行，并检查每一行是否满足SQL语句中的 Where 限制条件；

全表扫描时可以使用多块读（即一次I/O读取多块数据块）操作，提升吞吐量；

使用建议：数据量太大的表不建议使用全表扫描，除非本身需要取出的数据较多，占到表数据总量的 5% ~ 10% 或以上。

##### ****TABLE ACCESS BY ROWID（通过ROWID的表存取）****

ROWID是由Oracle自动加在表中每行最后的一列伪列，既然是伪列，就说明表中并不会物理存储ROWID的值。

你可以像使用其它列一样使用它，只是不能对该列的值进行增、删、改操作；

一旦一行数据插入后，则其对应的ROWID在该行的生命周期内是唯一的，即使发生行迁移，该行的ROWID值也不变。

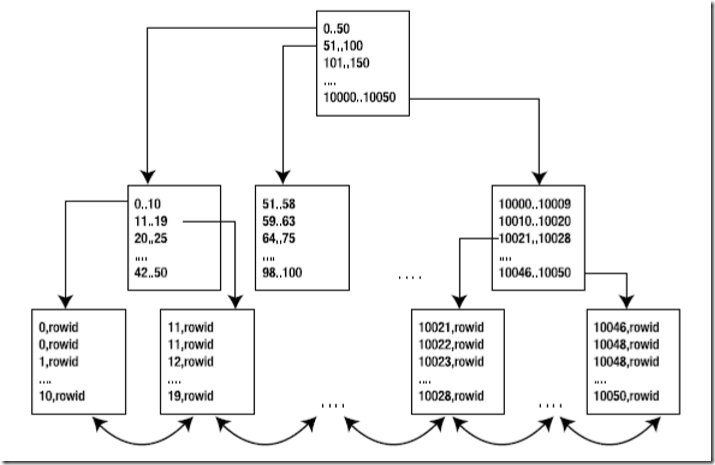
让我们再回到 TABLE ACCESS BY ROWID 来：

行的ROWID指出了该行所在的数据文件、数据块以及行在该块中的位置，所以通过ROWID可以快速定位到目标数据上，这也是Oracle中存取**单行**数据最快的方法。

##### ****TABLE ACCESS BY INDEX SCAN（索引扫描）****

在索引块中，既存储每个索引的键值，也存储具有该键值的行的ROWID。

一个数字列上建索引后该索引可能的概念结构如下图：



索引扫描分为两步：

扫描索引得到对应的ROWID。

通过ROWID定位到具体的行读取数据。

#### 索引扫描类型

##### INDEX UNIQUE SCAN（索引唯一扫描）

针对唯一性索引（UNIQUE INDEX）的扫描，每次至多只返回一条记录；

表中某字段存在 UNIQUE、PRIMARY KEY 约束时，Oracle常实现唯一性扫描；

##### INDEX RANGE SCAN（索引范围扫描）

使用一个索引存取多行数据；

发生索引范围扫描的三种情况：

在唯一索引列上使用了范围操作符（如：>   <   <>   >=   <=   between）。

在组合索引上，只使用部分列进行查询（查询时必须包含前导列，否则会走全表扫描）。

对非唯一索引列上进行的任何查询。

##### INDEX FULL SCAN（索引全扫描）

进行全索引扫描时，查询出的数据都必须从索引中可以直接得到（注意全索引扫描只有在[CBO](#_CBO（Cost-Based_Optimization）_基于代价的优)模式下才有效）。

##### INDEX FAST FULL SCAN（索引快速扫描）

扫描索引中的所有数据块，与 [INDEX FULL SCAN](#_INDEX_FULL_SCAN（索引全扫描）) 类似，但是一个显著的区别是它不对查询出的数据进行排序（即数据不是以排序顺序被返回）。

##### INDEX SKIP SCAN（索引跳跃扫描）

Oracle 9i后提供，有时候[复合索引](#_组合索引(复合索引))的前导列（索引包含的第一列）没有在查询语句中出现，oralce也会使用该复合索引，这时候就使用的INDEX SKIP SCAN;

什么时候会触发 INDEX SKIP SCAN 呢？

前提条件：表有一个复合索引，且在查询时有除了前导列（索引中第一列）外的其他列作为条件，并且优化器模式为CBO时

当Oracle发现前导列的唯一值个数很少时，会将每个唯一值都作为常规扫描的入口，在此基础上做一次查找，最后合并这些查询；

例如：

假设表emp有ename（雇员名称）、job（职位名）、sex（性别）三个字段，并且建立了如 create index idx\_emp on emp (sex, ename, job) 的复合索引；

因为性别只有 '男' 和 '女' 两个值，所以为了提高索引的利用率，Oracle可将这个复合索引拆成 ('男', ename, job)，('女', ename, job) 这两个复合索引；

当查询 select \* from emp where job = 'Programmer' 时，该查询发出后：

Oracle先进入sex为'男'的入口，这时候使用到了 ('男', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

再进入sex为'女'的入口，这时候使用到了 ('女', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

最后合并查询到的来自两个入口的结果集。

### 优化器

#### RBO（Rule-Based Optimization） 基于规则的优化器

RBO有严格的使用规则，只要按照这套规则去写SQL语句，无论数据表中的内容怎样，也不会影响到你的执行计划；

换句话说，RBO对数据“不敏感”，它要求SQL编写人员必须要了解各项细则；

RBO一直沿用至ORACLE 9i，从ORACLE 10g开始，RBO已经彻底被抛弃。

#### CBO（Cost-Based Optimization） 基于代价的优化器

CBO是一种比RBO更加合理、可靠的优化器，在ORACLE 10g中完全取代RBO；

CBO通过计算各种可能的执行计划的“代价”，即COST，从中选用COST最低的执行方案作为实际运行方案；

它依赖数据库对象的统计信息，统计信息的准确与否会影响CBO做出最优的选择，也就是对数据“敏感”。

## MySQL

# 系统

## 可维护性

可维护性用平均维修时间（MTTR）来度量，即系统发生故障后维修和重新恢复正常运行平均花费的时间。系统的可维护性越好，平均维修时间越短。

## 高可用性

计算机系统的可用性用[平均无故障时间](https://baike.baidu.com/item/%E5%B9%B3%E5%9D%87%E6%97%A0%E6%95%85%E9%9A%9C%E6%97%B6%E9%97%B4)（MTTF）来度量，即计算机系统平均能够正常运行多长时间，才发生一次故障。系统的可用性越高，平均无故障时间越长。计算机系统的可用性定义为：MTTF/(MTTF+MTTR) \* 100%。由此可见，计算机系统的可用性定义为系统保持正常运行时间的百分比。

# 分布式理论

## CAP原理

### 一致性（Consistency）

这个和数据库ACID的一致性类似，但这里关注的所有数据节点上的数据一致性和正确性，而数据库的ACID关注的是在在一个事务内，对数据的一些约束。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新值。

### 可用性（Availability）

每一个操作总是能够在一定时间内返回结果。需要注意“一定时间”和“返回结果”。“一定时间”是指，系统结果必须在给定时间内返回。“返回结果”是指系统返回操作成功或失败的结果。

### 分区容忍性（Partition tolerance）

是否可以对数据进行分区。这是考虑到性能和可伸缩性。

任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance），最多只能同时满足两项。所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

## 分布式事务

### 两阶段提交协议（2PC）

两阶段提交协议是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。

#### 协议参与者

在两阶段提交协议中，系统一般包含两类机器（或节点）：

##### 协调者（coordinator）

通常一个系统中只有一个；

##### 事务参与者（participants，cohorts或workers）

一般包含多个，在数据存储系统中可以理解为数据副本的个数。

协议中假设每个节点都会记录写前日志（write-ahead log）并持久性存储，即使节点发生故障日志也不会丢失。协议中同时假设节点不会发生永久性故障而且任意两个节点都可以互相通信。

#### 两个阶段的执行

##### 请求阶段（commit-request phase，或称表决阶段，voting phase）

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。

在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

##### 提交阶段（commit phase）

在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。

当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。

参与者在接收到协调者发来的消息后将执行相应的操作。

#### 两阶段提交的缺点

同步阻塞问题

##执行过程中，所有参与节点都是事务阻塞型的。

当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

单点故障

##由于协调者的重要性，一旦协调者发生故障。

参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

数据不一致

##在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。

而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。

#### 两阶段提交无法解决的问题

当协调者出错，同时参与者也出错时，两阶段无法保证事务执行的完整性。

考虑协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。

那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

### 消息事务+最终一致性

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：



1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

3、A执行本地事务

4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

* 步骤一出错，则整个事务失败，不会执行A的本地操作
* 步骤二出错，则整个事务失败，不会执行A的本地操作
* 步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息
* 步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

### 三阶段提交协议(3PC)

三阶段提交协议在协调者和参与者中都引入超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。

#### ****三个阶段的执行****

##### CanCommit阶段

3PC的CanCommit阶段其实和2PC的准备阶段很像。

协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

##### PreCommit阶段

Coordinator根据Cohort的反应情况来决定是否可以继续事务的PreCommit操作。

根据响应情况，有以下两种可能。

A.假如Coordinator从所有的Cohort获得的反馈都是Yes响应，那么就会进行事务的预执行：

发送预提交请求。Coordinator向Cohort发送PreCommit请求，并进入Prepared阶段。

事务预提交。Cohort接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

响应反馈。如果Cohort成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

B.假如有任何一个Cohort向Coordinator发送了No响应，或者等待超时之后，Coordinator都没有接到Cohort的响应，那么就中断事务：

发送中断请求。Coordinator向所有Cohort发送abort请求。

中断事务。Cohort收到来自Coordinator的abort请求之后（或超时之后，仍未收到Cohort的请求），执行事务的中断。

##### DoCommit阶段

该阶段进行真正的事务提交，也可以分为以下两种情况:

执行提交

**A.发送提交请求。**Coordinator接收到Cohort发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有Cohort发送doCommit请求。

**B.事务提交。**Cohort接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

**C.响应反馈。**事务提交完之后，向Coordinator发送ACK响应。

**D.完成事务。**Coordinator接收到所有Cohort的ACK响应之后，完成事务。

中断事务

Coordinator没有接收到Cohort发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

#### 三阶段提交协议和两阶段提交协议的不同

对于协调者(Coordinator)和参与者(Cohort)都设置了超时机制（在2PC中，只有协调者拥有超时机制，即如果在一定时间内没有收到cohort的消息则默认失败）。

在2PC的准备阶段和提交阶段之间，插入预提交阶段，使3PC拥有CanCommit、PreCommit、DoCommit三个阶段。

PreCommit是一个缓冲，保证了在最后提交阶段之前各参与节点的状态是一致的。

#### 三阶段提交协议的缺点

如果进入PreCommit后，Coordinator发出的是abort请求，假设只有一个Cohort收到并进行了abort操作，而其他对于系统状态未知的Cohort会根据3PC选择继续Commit，此时系统状态发生不一致性。

## 分布式锁

### 分布式锁的实现

#### 基于数据库实现分布式锁

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

这种简单的实现有以下几个问题：

这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。

这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。

这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

非阻塞的？搞一个while循环，直到insert成功再返回成功。

这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

基于MySql的InnoDB引擎，在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，可以通过commit方法解锁。

#### 基于缓存（redis，memcached，tair）实现分布式锁

#### 基于Zookeeper实现分布式锁

### 分布式锁应该是怎么样的

这里以方法锁为例，资源锁同理

可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。

这把锁要是一把可重入锁（避免死锁）。

这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）。

有高可用的获取锁和释放锁功能。

获取锁和释放锁的性能要好。

## [Session共享](https://blog.csdn.net/qq_34666857/article/details/77112985)

### 粘性session

粘性session是指Ngnix每次都将同一用户的所有请求转发至同一台服务器上，即将用户与服务器绑定。

### 服务器session复制

即每次session发生变化时，创建或者修改，就广播给所有集群中的服务器，使所有的服务器上的session相同。

### session共享

缓存session，使用redis， memcached。

### session持久化

将session存储至数据库中，像操作数据一样操做session。

# 消息队列框架

# 版本管理(版本控制)

## 常用术语

### 仓库（Repository）

受版本控制的所有文件修订历史的共享数据库。

### 工作空间（Workspace)

本地硬盘或Unix 用户帐户上编辑的文件副本。

### 工作树/区（Working tree）

工作区中包含了仓库的工作文件。您可以修改的内容和提交更改作为新的提交到仓库。

### 暂存区（Staging area）

暂存区是工作区用来提交更改（commit）前可以暂存工作区的变化。

### 索引（Index）

索引是暂存区的另一种术语。

### 签入（Checkin）

将新版本复制回仓库。

### 签出（Checkout）

从仓库中将文件的最新修订版本复制到工作空间。

### 提交（Commit）

对各自文件的工作副本做了更改，并将这些更改提交到仓库。

### 冲突（Conflict）

多人对同一文件的工作副本进行更改，并将这些更改提交到仓库。

### 合并（Merge）

将某分支上的更改联接到此主干或同为主干的另一个分支。

### 分支（Branch）

从主线上分离开的副本，默认分支叫master。

### 锁（Lock）

获得修改文件的专有权限。

### 头（HEAD）

头是一个象征性的参考，最常用以指向当前选择的分支。

### 修订（Revision）

表示代码的一个版本状态。Git通过用SHA1 hash算法表示的ID来标识不同的版本。

### 标记（Tags）

标记指的是某个分支某个特定时间点的状态。通过标记，可以很方便的切换到标记时的状态。

## Git

### 创建本地ssh

ssh-keygen -t rsa -C "13322808776@189.cn

### 验证是否配置成功

ssh -T git@github.com

### 本地git仓库关联GitHub仓库

git remote add origin git@github.com:bsc2012/documents.git

### 初始化[git](http://lib.csdn.net/base/28)仓库

git init

### 添加文件

git add ./\*

### 删除文件

git rm filename

### 提交缓存

git commit -m '提交'

### 提交到远程GitHub仓库

git push -u origin master

### ****与GitHub远程仓库同步****

git pull origin master

## github

<https://github.com/bsc2012>

账号:bsc2012 密码:bsc\*\*\*\*\*\*\*213