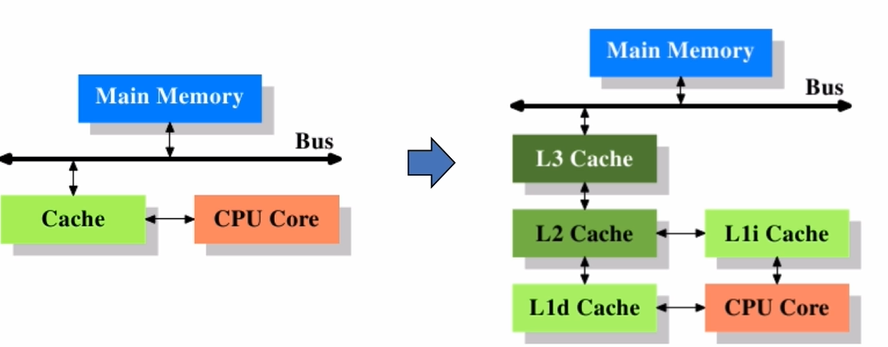
# 并发基础

## CPU多级缓存



* 为什么需要CPU cache：

CPU的频率太快了，快到主存跟不上，这样在处理器时钟周期内，CPU常常需要等待主存，浪费资源。所以cache的出现，是为了缓解CPU和内存之间速度的不匹配问题(结构：cpu->cache->memory)

* CPUcache有什么意义：

1. 时间局部性：如果某个数据被访问，name在不久的将来它很可能被再次访问
2. 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

### 缓存一致性(MESI)

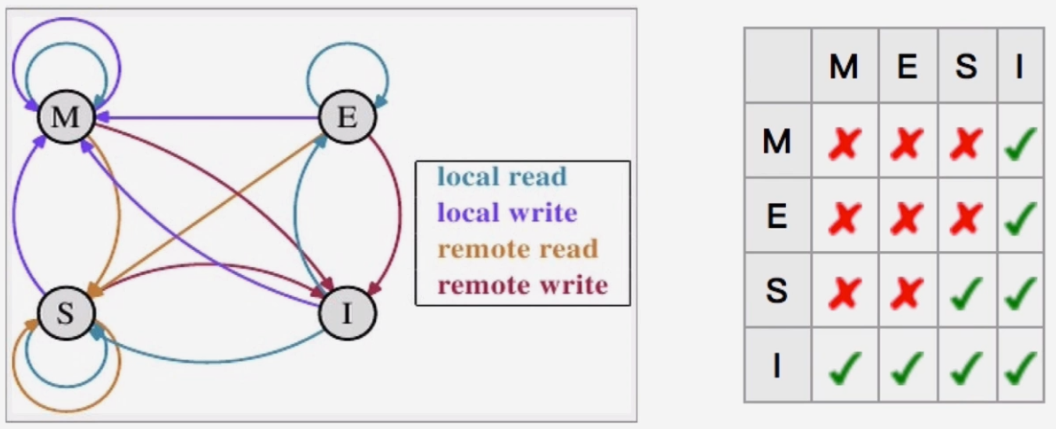
* 用户保证多个CPU cache之间缓存共享数据的一致

M: 被修改（Modified)，

E: 独享的（Exclusive)，

S: 共享的（Shared)

I: 无效的（Invalid）



### 乱序执行优化

* 处理器为提高运算速度而做出违背代码原有顺序的优化

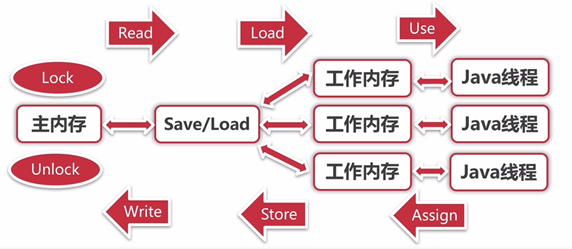
## java内存模型(JMM)

### 同步八种操作

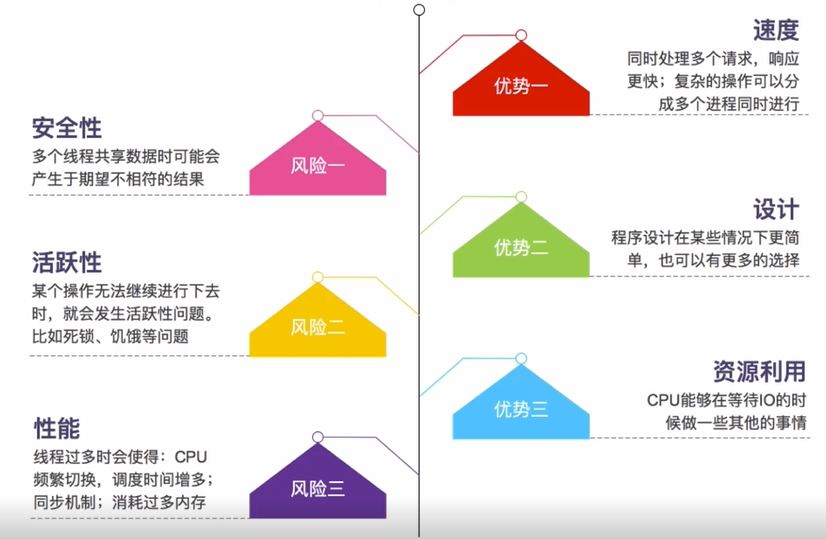
* lock(锁)：作用于主内存的变量，把一个变量标识为一条线程独占状态
* unlock(解锁)：作用于主内存的变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其它线程锁定
* read(读取)：作用于主内存的变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
* load(载入)：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中
* use(使用)：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎
* assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的赋值给工作内存的变量
* store(存储)：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，一遍随后的write的操作
* write(写入)：作用于主内存的变量，它把store操作从工作内存中的一个变量的值传送到主内存的变量中。

### 同步规则

* 如果要把一个变量从主内存中复制到一个工作内存，就需要按顺序地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。但java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行
* 不允许read和load、store和write操作之一单独出现
* 不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中
* 不允许一个线程无原因地(没有发生过任何assign操作)把数据从工作内存同步回主内存中
* 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化(load或assign)的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作
* 一个变量在同一时刻只允许一个线程对其进行lock操作，但lock操作可以被同一个线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。lock和unlock必须成对出现
* 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值
* 如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量
* 对一个变量执行unlock操作之前，必须先把此变量同步到主内存中(执行store和write操作)



## 并发的优势与风险



# 第二章 线程安全性

## 定义：

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些进程将如何交替执行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

## 具体体现：

* 原子性：提供了互斥访问，同一时刻只能有一个线程来对它进行操作
* 可见性：一个线程对主内存的修改可以及时的被其它线程观察到
* 有序性：一个线程观察其他线程中的指令执行顺序，由于指令重新排序的存在，该观察结果一般杂乱无序。

## 原子性-Atomic包

* AtomicXXX:CAS、Unsafe.compareAndSwapXXX
* AtomicLong、LongAdder,前者准确性更高，但是在超高并发情况下，由于底层是一个恶while循环比较，所以效率低一点，后者在超高并发下效率高，因为拆分成了数组，最后计算每个数组的值，但是不是很准确。
* AtomicReference、AtomicReferenceFieldUpdater,后者是改变某个类中字段的值，这个字段不能是static修饰的，必须只能是volatile修饰的。
* AtomicStampReference:CAS的ABA问题（是指在CAS操作的时候，其他线程将变量的值由A改到了B，然后又改回了A，当前线程使用期望值A与当前变量进行比较的时候，发现变量没有变，于是CAS就将A值进行了交换操作），解决方法：每次变量更新的时候，把变量的版本号加一，这时通过版本号就知道变量被修改了几次。

## 原子性-锁

* synchronized：依赖JVM(作用对象的作用范围内操作)
* Lock：依赖特殊的CPU指令，代码实现，ReentrantLock

### 原子性- synchronized

* 修饰代码块：大括号括起来的代码，作用于调用的对象
* 修饰方法：整个方法，作用于调用的对象，如果当前类是一个父类，其子类想调用父类的synchronized方法，是带不走这个关键字的，因为synchronized不属于方法声明的一部分。
* 修饰静态方法：整个静态方法，作用于所有对象
* 修饰类：括号括起来的部分，作用于所有对象

### 原子性-对比

* synchronized：不可中断锁，适合竞争不激烈，可读性好
* Lock：可中断锁，多样化同步，竞争激烈时能维持常态
* Atomic：竞争激烈时能维持常态，比lock性能好，只能同步一个值

## 可见性

导致共享变量在线程间不可见的原因：

1. 线程交叉执行
2. 重排序结合线程交叉执行
3. 共享变量更新后的值没有在工作内存与主存间及时更新

### 可见性-synchronized

JMM关于synchronized的两条规定：

1. 线程解锁前，必须把共享变量的最新值刷新到主内存
2. 线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值（注意，加锁与解锁是用一把锁）

### 可见性-volatile

通过加入内存屏障和禁止重排序优化来实现：

1. 对volatile变量写操作时，会在写操作后加入一条store屏障指令，将本地内存中的共享变量值刷新到主内存
2. 对volatile变量读操作时，会在写操作后加入一条load屏障指令，从主内存中读取共享变量

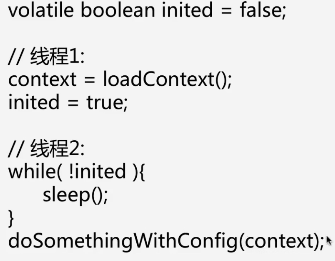




### 可见性-volatile使用

不具备原子性，使用条件：

1. 对变量的写操作不依赖于当前值
2. 是该变量没有包含在具有其他变量的不变的式子中



## 有序性

* Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。
* volatile、synchronized、Lock

### happens-before原则

* 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
* 锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作
* volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
* 传递规则：如果操作A先行发生于操作B，而操作B又发生于操作C，则可以得出操作A先行发生于操作C
* 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作
* 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码监测到中断事件的发生
* 线程终结规则：线程中所有的操作都先行发生于线程的终止监测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
* 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

# 第三章 安全发布对象

* 发布对象：是一个对象能够被当前范围之外的代码所使用
* 对象逸出：一种错误的发布。放一个对象还没有构造完成时，就使它被其它线程所见

## 安全发布对象

* 在静态初始化函数中初始化一个对象的引用，
* 将对象的引用保存到volatile类型域或者AtomicReference对象中
* 将对象的引用保存到某个正确构造对象的final类型域中
* 将对象的引用保存到一个有锁的保护域中

# 第四章 线程安全策略

## 不可变对象

* 不可变对象需要满足的条件
* 对象创建以后起状态就不能修改
* 对象所有域都是final类型
* 对象是正确创建的(在对象创建期间，this引用没有逸出)
* final关键字：类、方法、变量
* 修饰类：不能被继承
* 修饰方法：1.方法不被继承类修改；2.效率
* 修饰变量：基本数据类型变量(不能改变值)、引用变量(不能指向另外一个对象)
* Collections.unmodifiableXXX:Collection、List、Set、Map…
* Guava:ImmutableXXX: Collection、List、Set、Map…

## 线程封闭

* Ad-hoc线程封闭：程序控制实现，最糟糕，忽略
* 堆栈封闭：局部变量，无并发问题
* ThreadLocal线程封闭：特别好的封闭方法

## 线程不安全类与写法

* StringBuilder、StringBuffer
* SimpleDateFormat、JodaTime
* ArrayList,HashSet,HashMap等Collections
* 先检查再执行:if(condition(a)){handle(a);}

## 线程安全-同步容器

ArrayList -> Vector,Stack

HashMap -> HashTable(key,value不能为null)

Collections.synchronizedXXX(List、Set、Map)

## 线程安全-并发容器 J.U.C

* ArrayList -> CopyOnWriteArrayList(缺点，1.数据量很大消耗内存会引起fullgc和minorgc，2.不能用于实时性；读操作是在原数组上读的没有加锁的，写操作时加锁的，避免多个线程并发修改。)
* HashSet、TreeSet -> CopyOnWriteArraySet(底层实现CopyOnWriteArrayList)、ConcurrentSkipListSet(jdk6出现的，与treeset一样支持自然排序，对元素的单个操作都是线程安全的，但是批量操作如addAll、removeAll等，底层还是调用的add、和remove不保证在此操作的时候不被其它线程打断)
* HashMap、TreeMap -> ConcurrentHashMap(不允许空值,对读操作做了优化，高并发超级高)、ConcurrentSkipListMap(内部时使用SkipList跳表的结构来实现的)，性能上，前者的存取速度是后者的4倍；后者的key是有序的，前者做不到；后者支持更高的并发度

## J.U.C的整体架构图

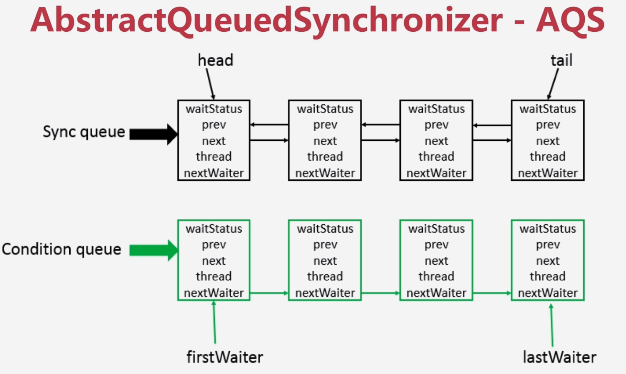


## 安全共享共享策略总结

* 线程限制：一个被线程限制的对象，有线程独占，并且只能被占有他的线程修改
* 共享制度：一个共享制度的对象，在没有额外同步的情况下，可以被多个线程并发访问，但是任何线程都不能修改它
* 线程安全对象：一个线程安全的对象或者容器，在内部通过同步机制来保证线程安全，所以其他线程无需额外的同步就可以通过公共接口随意访问它
* 被守护对象：被守护对象只能通过获取特定的锁来访问

# 第五章 J.U.C之AQS

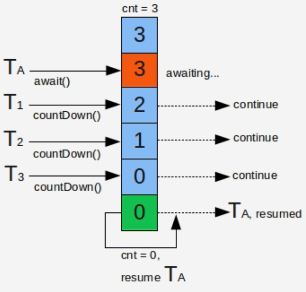
## AbstractQueuedSynchronizer – AQS数据结构



* 使用Node实现FIFO队列，可以用于构建锁或者其他同步装置的基础框架
* 利用了一个int类型表示状态
* 使用方法是继承
* 子类通过继承并通过实现它的方法管理其状态{acquire和release}的方法操纵状态
* 可以同时实现排它锁和共享锁模式（独占、共享）

## AQS同步组件

* CountDownLatch

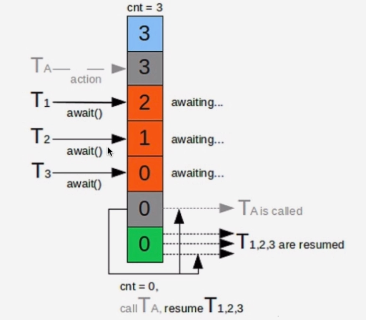


比如说并行计算，当某个处理的运算量很大时，可以将该任务拆分成多个子任务，等待所有的子任务完成之后，父任务才拿到所有子任务的结果进行计算

* Semaphore信号量，必须由获取它的线程释放。常用于限制可以访问某些资源的线程数量，例如通过 Semaphore 限流。

如果 acquire 的数量大于 release 的数量，则 通路迟早会被使用完，如果线程比较多，得不到后续运行，出现线程堆积内存，最终java进程崩掉；如果 acquire 的数量小于 release 的数量，就会出现并发执行的线程越来越多（换句话说，处理越来越快），最终也有可能出现问题。

* CyclicBarrier



* 与CountDowLatch的区别：1. CountDowLatch的计数器只能用一次，CyclicBarrier的计数器可以使用reset方法重置循环使用；2. CountDowLatch主要是实现一个或N个线程需要等待其他线程完成某个操作之后才能继续往下执行，而CyclicBarrier主要是实现了多个线程之间相互等待，直到所有的线程都满足了条件之后才能继续执行后续的操作，CyclicBarrier能处理更复杂的场景。
* ReentrantLock
* ReentrantLock(可重入锁)和synchronized区别

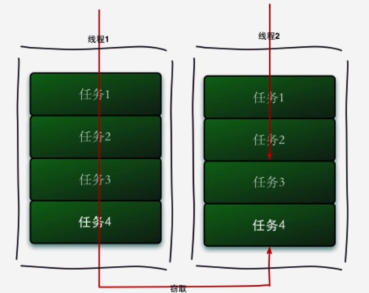
1. 可重入性
2. 锁的实现
3. 性能的区别
4. 功能区别：便利性，细粒度和灵活度

* ReentrantLock独有的功能

1. 可指定是公平锁还是非公平锁
2. 提供了一个condition类，可以分组唤醒需要唤醒的线程
3. 提供能够中断等待锁的线程的机制，lock.lockInterruptibly()

锁的选取：1.当只有少量竞争者的时候，synchronized是首选；2.竞争者不少但线程增长的趋势能够预估的，ReentrantLock是一个很好的选择。切忌乱用，synchronized是不会引发死锁的，因为JVM会在释放

* Condition是多个线程中的通信工具
* FutureTask
* Callable与Runnable接口对比，runnable是一个接口，里面就一个run方法，使用一个类去实现它，然后创建一个线程去执行它；Callable是一个泛型接口，里面有一个call函数，它的返回类型就是传进去的类型。
* Future接口 对于一个具体的callable或者runnable的一个任务，他可以进行取消，查询任务是否被取消，是否完成，获取结果。
* FutureTask类 实现了RunnableFuture接口，而RunnableFuture继承了Runnable，Future接口，因此这个类即可得到runnable被线程执行，也可得到future得到线程执行后的结果。
* Fork/Join 框架(java7提供的一个用于并行执行任务的框架，它是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架)
* 主要采用的是工作窃取算法，是指某个线程从其它队列里窃取任务来执行，流程图如下：



优势在于充分利用线程进行并行计算，并减少了线程间的竞争；缺点在某些情况下还是存在线程竞争。

* 核心是两个类:ForkJoinPool(负责来做实现，包括上面的工作窃取算法，它管理工作线程和提供关于任务的状态，以及他们的执行信息)和ForkJoinTask(主要提供在任务中执行fork和join操作的机制)
* BlockingQueue阻塞队列
* ArrayBlockingQueue先进先出的规矩
* DelayQueue元素过期优先顺序排序
* LinkedBlockingQueue内部是一个链表
* PriorityBlockingQueue使用，必须实现comparable接口
* SynchronousQueue内部仅允许一个元素，是一个同步阻塞队列

# 第六章 线程池

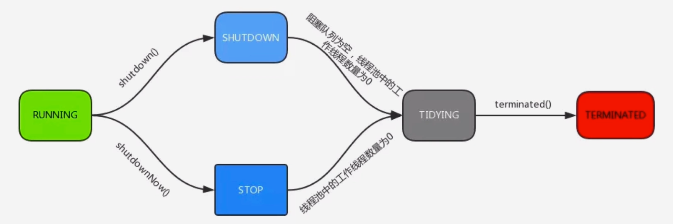
* new Thread弊端
* 每次new Thread新建对象，性能差
* 线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或者OOM
* 缺少更多功能，如更多执行、定期执行、线程中断
* 线程池的好处
* 重用存在的线程，减少对象创建、消亡的开销，性能佳
* 可有效控制最大并发线程数，提高系统资源利用率，同事可以避免过多资源竞争，避免阻塞
* 提供定时执行、定期执行、单线程、并发数控制等功能

## ThreadPoolExecutor

* corePoolSize：核心线程数量
* maximumPoolSize：最大线程数
* workQueue：阻塞队列，存储等待执行的任务，很重要，会对线程池运行过程产生重大影响（作用：它是保存等待执行的任务的一个阻塞队列，当我们提交一个新的任务到线程池中，线程池会根据当前正在运行的线程的数量来决定该任务的处理方式(1.直接切换，2.无界队列(LinkedBlockingQueue)，3.有界队列)）

三者的关系：如果运行的线程数小于corePoolSize，直接创建新线程来处理任务，即使线程池中的其他线程是空闲的；如果线程池中的线程数量大于等于corePoolSize，且小于等于maximumPoolSize，则只有当workQueue满的时候才会去创建新的线程去执行任务，如果设置的corePoolSize和maximumPoolSize是相同的话，那么线程池的大小是固定的；如果有新任务提交，且workQueue还没有满，就把请求放到workQueue里面，等待有空闲的线程，然后队列里面去取出任务进行处理；如果我们运行的线程数量大于了maximumPoolSize，这时如果workQueue已经满了，那么就要指定策略去处理任务。

* keepAliveTime：线程没有任务执行时最多保持多久时间终止
* unit：keepAliveTime的时间单位
* threadFactory：线程工厂，用来创建线程，非守护的线程，优先级
* rejectHandler：当拒绝处理人时的策略
* 直接抛出异常，默认的策略
* 用调用者所在的线程来执行任务
* 丢弃队列中最靠前的任务并执行当前任务
* 直接丢弃调当前任务
* 状态
* 示意图



* running：判断接受新提交的任务，也能处理阻塞队列中的任务
* shutdown：关闭状态，当一个线程池实例属于此状态的时候，不能再接受新提交的任务，但却可以处理阻塞队列中已经保存的任务
* stop：不能接受新任务，也不能处理队列中的任何任务，它会中断正在处理任务中的线程
* tidying：如果所有的任务都已经终止了，这时候有效线程数为0，线程池会进入该状态
* terminated：tidying状态调用terminated()方法会进入terminated状态
* 方法
* execute():提交任务，交给线程池执行
* submit():提交任务，能够返回执行结果 execute+Future
* shutdown():关闭线程池，等待任务都执行完
* shutdownNow():关闭线程池，不等待任务执行完
* getTaskCount():线程池已执行和未执行的任务总数
* getCompletedTaskCount():已完成的任务数量
* getPoolSize():线程池当前的线程数量
* getActiveCount():当前线程池中正在执行任务的线程数量

## Executor框架接口

* Executors.newCachedThreadPool创建一个可缓存的线程池
* Executors.newFixedThreadPool创建一个定长的线程池，可以控制线程的最大并发数，超出的线程会在队列中等待
* Executors.newScheduledThreadPool创建的也是一个定长的线程池，它支持定时以及周期性的任务还行
* Executors.newSingleThreadExecutor创建的是一个单线程块的线程池，它只会用唯一的一个工作线程来执行任务，保证所有任务按照指定顺序(先入先出，优先级)去执行

## 合理配置

* CPU密集型任务，就需要尽量压榨CPU，参考值可以设为NCPU+1
* IO密集型任务，参考值可以设置为2\*NCPU

# 第七章 多线程并发拓展

## 死锁

* 概念

是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

* 排他锁，等待事务 B 完成并释放其对行 2 持有的共享锁之前被阻塞。
* 必要条件
* 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
* 请求与保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
* 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
* 环路等待条件指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

## 多线程并发最佳实战

* 使用本地变量，而不是创建一个类或实例变量，
* 使用不可变类，可以降低代码中的需要的同步数量
* 最小化锁的作用域范围：S=1/(1-a+a/n)，a表示并行数量所占的比例，n是并行处理的节点个数，S表示的加速比
* 使用线程池的Executor，而不是直接new Thread知行，创建一个线程的代价是昂贵的
* 宁可使用同步也不要使用线程的wait和notify
* 使用BlockingQueue实现生产-消费模式
* 使用并发集合而不是加了锁的同步集合
* 使用Semaphore创建有界的访问
* 宁可使用同步代码块，也不实用同步的方法
* 避免使用静态变量

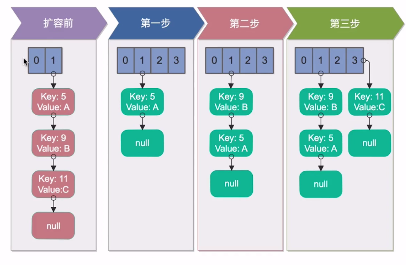
## Spring与线程安全

* Spring bean：singleton、prototype
* 无状态对象，每个单例的无状态对象都是线程安全的，该对象是没有状态的，自然也不会因为多个线程的交替调度而改变自身的状态，导致线程安全问题，无状态对象包含使用的DTO对象、VO这些只作为数据的实体对象，还有使用的service，dao和controller等等，这些对象并没有状态，他们只是用来执行某些操作的
* Spring根本就没有对bean的多线程安全问题做出任何的保障与措施，对于每个bean的线程安全问题根本原因是因为每个bean的自身设计没有在在bean中声明任何有状态的示例变量或类变量，当然如果必须在spring管理的bean里面加入有状态的实例变量或类变量，把我们的这个类变成一个有状态的对象的时候，那么我们就需要使用ThreadLocal，把变量变成线程私有的，如果bean的实例变量或类变量需要在线程之间共享，那么我们只能使用synchronized，lock、CAS等这些来实现线程同步的方法了。

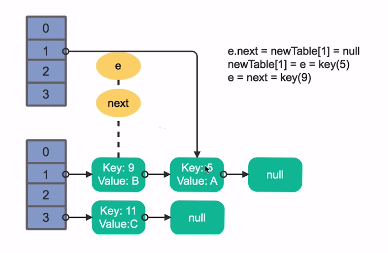
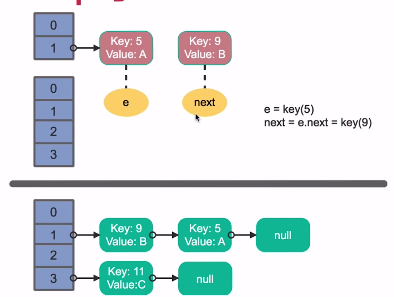
## HashMap与ConcurrentHashMap

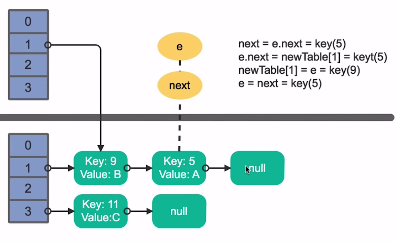
### Hashmap的扩容示意图

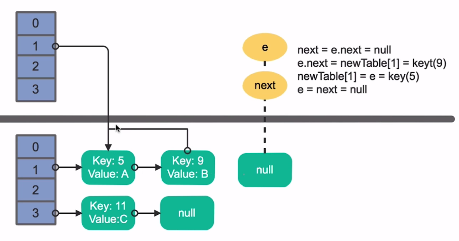
单线程下的rehash



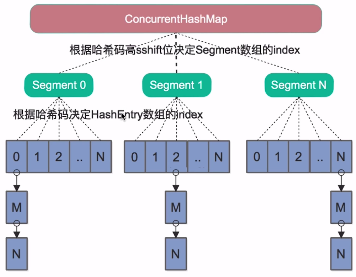
多线程下的rehash



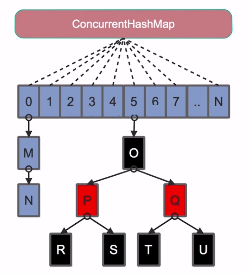




### ConcurrentHashMap示意图



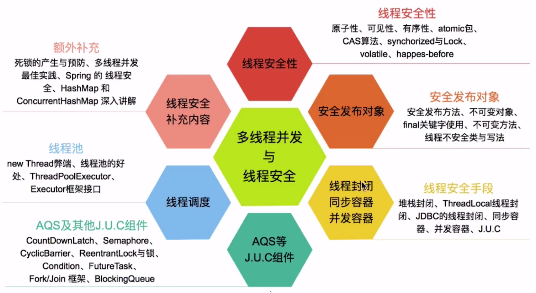
JDK8引入红黑树提高并发



### 区别

* HashMap是线程不安全的，ConcurrentHashMap是线程安全的
* HashMap允许key和value为空，ConcurrentHashMap不允许
* HashMap不允许通过Iterator遍历的同时，通过hashmap来修改，ConcurrentHashMap是允许该行为的

## 多线程并发与线程安全总结



# 第八章 高并发处理思路与手段

## 扩容

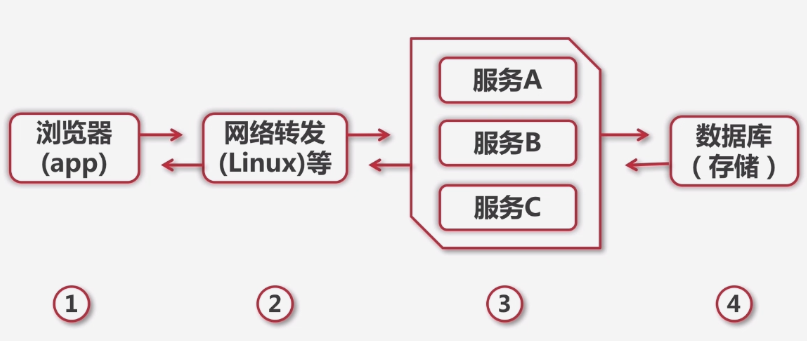
* 垂直扩容(纵向扩展)：提高系统部件能力
* 水平扩容(横向扩展)：增加更多系统成员来实现，这样想到集群

## 扩容-数据库

* 读操作扩展：memcache、redis、CDN等缓存
* 写操作扩展：Cassandra，Hbase等

# 第九章 高并发之缓存思路

## 缓存



## 缓存特征

* 命中率：命中数/(命中数+没有命中数)
* 最大元素(空间)
* 清空策略：FIFO(先进先出)，LFU(最少使用策略)，LRU(最近最少使用策略)，过期时间，随机等

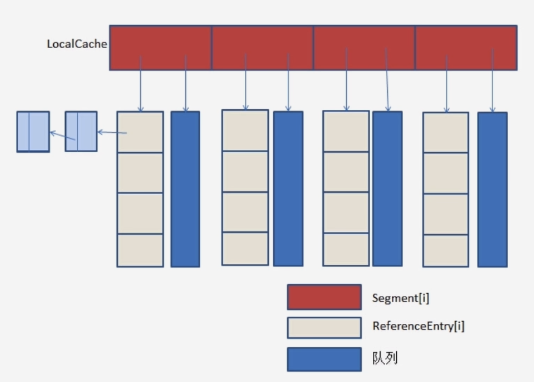
## 缓存命中率影响因素

* 业务场景和业务需求
* 缓存的设计(粒度和策略)
* 缓存容量和基础设施

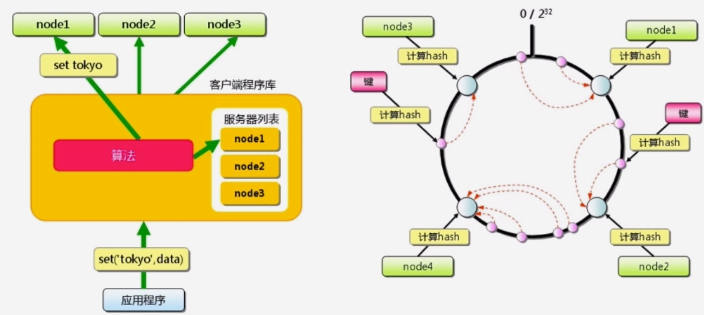
## 缓存分类和应用场景

* 本地缓存：编程实现(成员变量，局部变量，静态变量)、Guava Cache
* 分布式缓存：Memcache、Redis，自身就一个独立的应用。

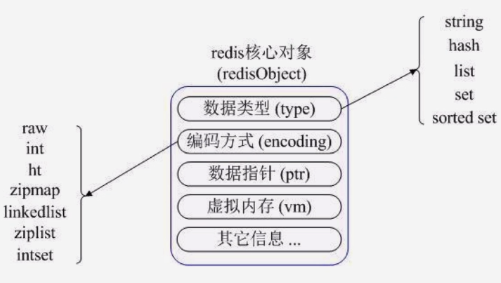
## 缓存-GUAVA Cache



## 缓存-Memcache



## 缓存-Redis



* 性能极高：读的速度10万次/秒，写8万次/秒，具有原子性，对合并后的原子性操作

## 高并发场景下缓存常见问题

* 缓存一致性

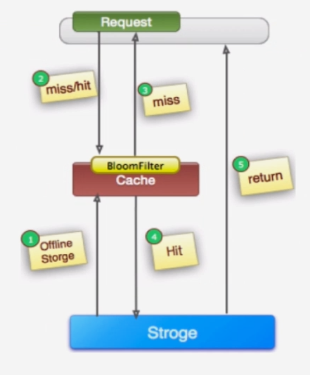
即缓存数据与数据库数据一致，缓存节点与副本一致，更新策略



* 缓存并发问题：使用互斥锁(mutex key)



* 缓存穿透问题：1.缓存空对象，集合是缓存空集合不是null；2.布隆过滤



* 缓存雪崩现象：由于缓存的原因大量的请求到达后端数据库，导致数据库崩溃整个系统崩溃，原因很多种。多级缓存来处理

