## 二、高并发

### 1、造成线程安全的主要因素有哪些？

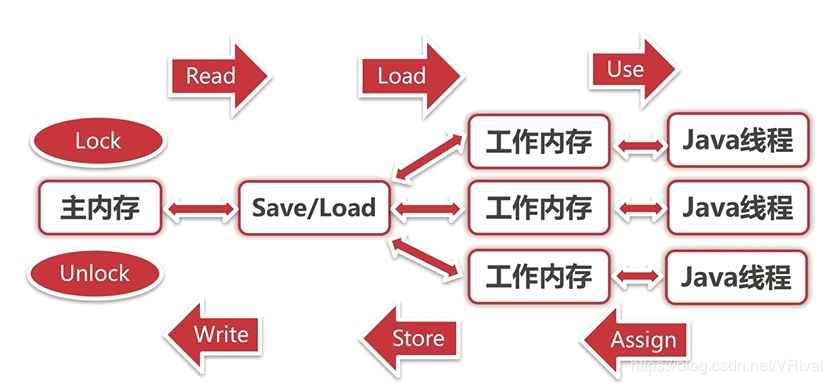
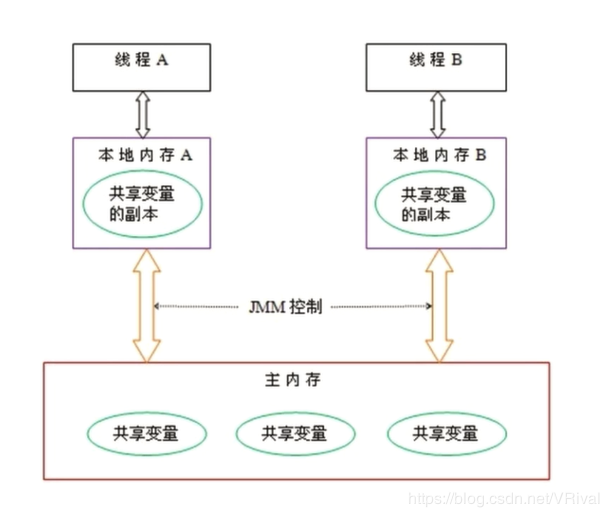
**原子性**：提供了互斥访问，同一时刻只能有一个线程对它进行操作【原子性是面向CPU、指令级别的】

**可见性**：一个线程对主内存的修改可以及时的被其他线程观察到。

**有序性**：一个线程观察其他线程中的指令执行顺序，由于指令重排序的存在，该观察结果一般杂乱无序。

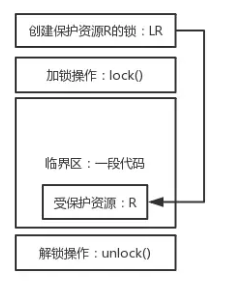
补充：

1. Java内存模型



### 2、互斥锁是什么？java锁如何分类？

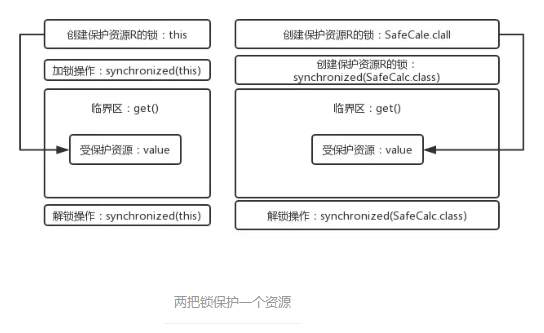
互斥操作：

Java语言提供的锁技术：synchronized

       锁是一种通用的技术方案，Java语言提供的synchronized关键字，就是锁的一种实现。synchronized关键字可以用来修饰方法，也可以用来修饰代码块，但是他没有显示的加锁lock()和解锁unlock()操作，这两个操作是被Java默默加上的，Java编译器会在synchronized修饰的方法或者代码块前后自动加上加锁lock()和解锁unlock()，这样做的好处就是加锁lock()和解锁unlock()一定是成对出现的，毕竟忘记解锁unlock()可是一个致命的bug。

       Java中有一条隐式规则：当synchronized修饰静态方法的时候，锁定的是当前类的Class对象；当synchronized修饰非静态方式的时候，锁定的是当前实例对象this；当synchronized修饰代码块的时候，锁定的是其后括号的对象。



共享资源的使用是互斥的，即一个线程获得资源的使用权后就会将该资源加锁，使用完后会将其解锁，如果在使用过程中有其他线程想要获取该资源的锁，那么它就会被阻塞陷入睡眠状态，直到该资源被解锁才会被唤醒，如果被阻塞的资源不止一个，那么它们都会被唤醒，但是获得资源使用权的是第一个被唤醒的线程，其它线程又陷入沉睡.

互斥锁在各个语言中定义都不太相同

在大多数语言中, 互斥锁使用线程调度来实现的, 假如现在锁被锁住了, 那么后面的线程就会进入”休眠”状态, 直到解锁之后, 又会唤醒线程继续执行. 这也叫空等待(sleep-waiting).

**但严格来说, 只要是同一时刻只能被拿到一次的锁都叫互斥锁，自旋锁也是**

**java锁如何分类:**

**偏向锁、轻量级锁、重量级锁**

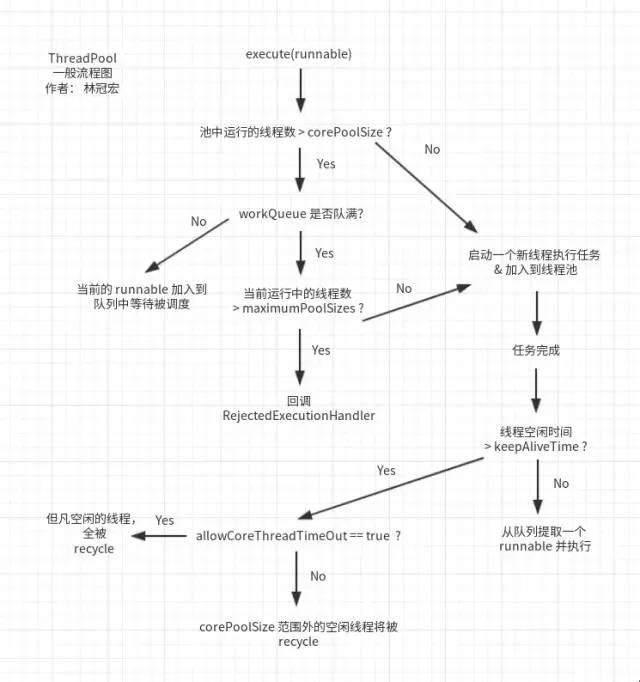


### 3、关于锁名词的分类

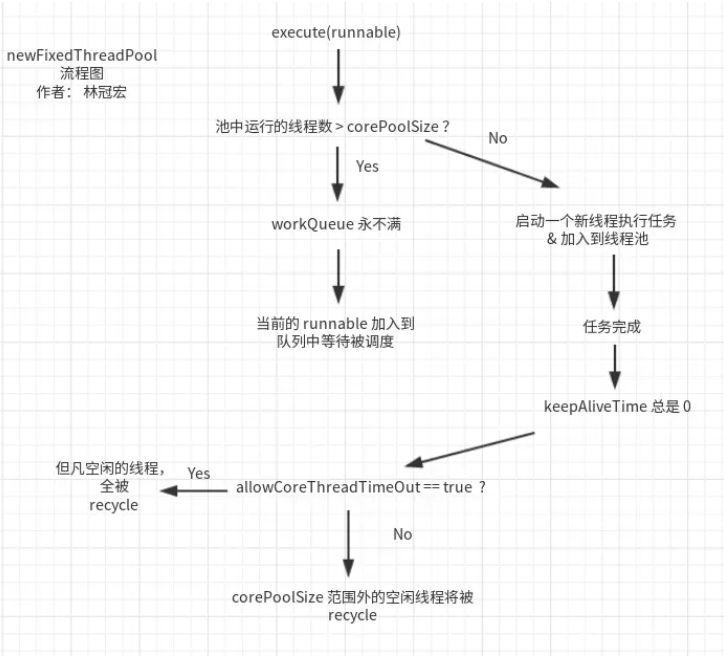
重入锁、独占锁、共享锁、公平锁、非公平锁、乐观锁、悲观锁

### 4、通过线程池创建线程的流程是什么？

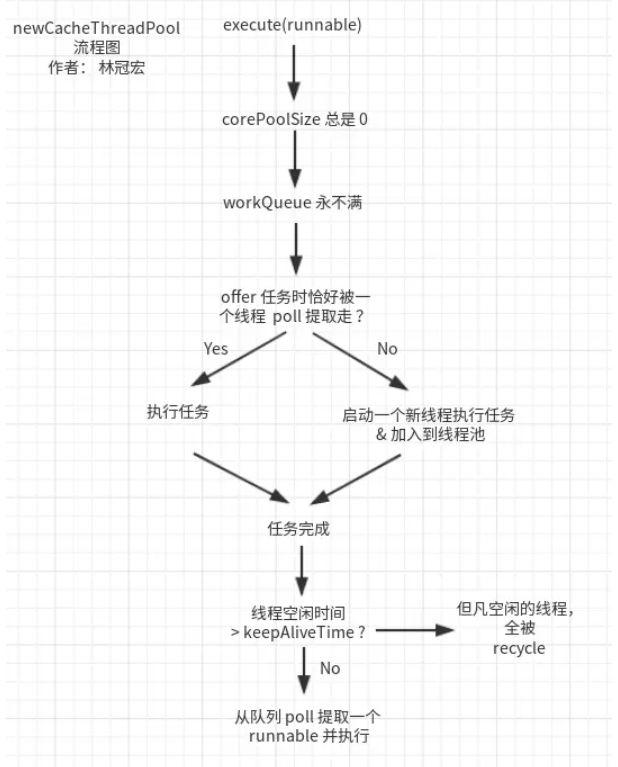
一般流程图



newFixedThreadPool 流程图



newCacheThreadPool 流程图



newSingleThreadPool 流程图

public static ExecutorService newSingleThreadExecutor() {

return

new FinalizableDelegatedExecutorService

(

new ThreadPoolExecutor

(

1,

1,

0L,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>(),

threadFactory

)

);

}

### 5、synchronize和volitile的区别？

区别：

1、volatile是变量修饰符，而synchronized则作用于一段代码或方法。

2、volatile只是在线程内存和“主”内存间同步某个变量的值；而synchronized通过锁定和解锁某个监视器同步所有变量的值, 显然synchronized要比volatile消耗更多资源。

3、volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

4、volatile保证数据的可见性，但不能保证原子性；而synchronized可以保证原子性，也可以间接保证可见性，因为它会将私有内存中和公共内存中的数据做同步。

5、volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

线程安全包含原子性和可见性两个方面，Java的同步机制都是围绕这两个方面来确保线程安全的。

**关键字volatile主要使用的场合是在多个线程中可以感知实例变量被修改，并且可以获得最新的值使用，也就是多线程读取共享变量时可以获得最新值使用。**

**关键字volatile提示线程每次从共享内存中读取变量，而不是私有内存中读取，这样就保证了同步数据的可见性。但是要注意的是：如果修改实例变量中的数据**

**注意：volatile本身并不处理数据的原子性，而是强制对数据的读写及时的影响到主内存中。**

**Synchronized注意：能够保证线程间的数据可视性（同一时刻，只有一个线程执行某一个方法或一个代码块，包含两个特性：互斥性+可见性）**

### 10、设计一个高并发的网站通用的技术有哪些？

[高可用高并发常用到的9种技术](https://www.cnblogs.com/doit8791/p/9380827.html)

**1、分层**

分层是企业应用系统中最常见的一种架构模式，将系统在横向维度上切分成几个部分，每个部分负责一部分相对简单并比较单一的职责，然后通过上层对下层的依赖和调度组成一个完整的系统。

在网站的分层架构中，**常见的为3层，即应用层、服务层、数据层**。应用层具体负责业务和视图的展示；服务层为应用层提供服务支持；数据库提供数据存储访问服务，如数据库、缓存、文件、搜索引擎等。

分层架构是逻辑上的，在物理部署上，三层架构可以部署在同一个物理机器上，但是随着网站业务的发展，必然需要对已经分层的模块分离部署，即三层结构分别部署在不同的服务器上，是网站拥有更多的计算资源以应对越来越多的用户访问。

所以虽然分层架构模式最初的目的是规划软件清晰的逻辑结构以便于开发维护，但在网站的发展过程中，分层结构对网站支持高并发向分布式方向的发展至关重要。

**2、冗余**

网站需要7×24小时连续运行，那么就得有相应的冗余机制，以防某台机器宕掉时无法访问，而冗余则可以通过部署至少两台服务器构成一个集群实现服务高可用。数据库除了定期备份还需要实现冷热备份。甚至可以在全球范围内部署灾备数据中心。

**3、分隔**

如果说分层是将软件在横向方面进行切分，那么分隔就是在纵向方面对软件进行切分。

网站越大，功能越复杂，服务和数据处理的种类也越多，将这些不同的功能和服务分隔开来，包装成高内聚低耦合的模块单元，不仅有助于软件的开发维护也便于不同模块的分布式部署，提高网站的并发处理能力和功能扩展能力。

大型网站分隔的粒度可能会很小。比如在应用层，将不同业务进行分隔，例如将购物、论坛、搜索、广告分隔成不同的应用，有对立的团队负责，部署在不同的服务器上。

**4、异步**

使用异步，业务之间的消息传递不是同步调用，而是将一个业务操作分成多个阶段，每个阶段之间通过共享数据的方法异步执行进行协作。

**具体实现则在单一服务器内部可用通过多线程共享内存对了的方式处理**；

**在分布式系统中可用通过分布式消息队列来实现异步。**

异步架构的典型就是生产者消费者方式，两者不存在直接调用。

**5、分布式**

对于大型网站，分层和分隔的一个主要目的是为了切分后的模块便于**分布式部署**，**即将不同模块部署在不同的服务器上，通过远程调用协同工作。分布式意味着可以使用更多的计算机完同样的工作，计算机越多，CPU、内存、存储资源就越多，能过处理的并发访问和数据量就越大，进而能够为更多的用户提供服务。**

在网站应用中，常用的分布式方案有一下几种.

**分布式应用和服务**：将分层和分隔后的应用和服务模块分布式部署，可以改善网站性能和并发性、加快开发和发布速度、减少数据库连接资源消耗。

**分布式静态资源**：网站的静态资源如JS、CSS、Logo图片等资源对立分布式部署，并采用独立的域名，即人们常说的动静分离。静态资源分布式部署可以减轻应用服务器的负载压力；通过使用独立域名加快浏览器并发加载的速度。

**分布式数据和存储：**大型网站需要处理以P为单位的海量数据，单台计算机无法提供如此大的存储空间，这些数据库需要分布式存储。

**分布式计算：**目前网站普遍使用Hadoop和MapReduce分布式计算框架进行此类批处理计算，其特点是移动计算而不是移动数据，将计算程序分发到数据所在的位置以加速计算和分布式计算。

**6、安全**

网站在安全架构方面有许多模式：通过密码和手机校验码进行身份认证；登录、交易需要对网络通信进行加密；为了防止机器人程序滥用资源，需要使用验证码进行识别；对常见的XSS攻击、SQL注入需要编码转换；垃圾信息需要过滤等。

**7、自动化**

具体有自动化发布过程，自动化代码管理、自动化测试、自动化安全检测、自动化部署、自动化监控、自动化报警、自动化失效转移、自动化失效恢复等。

**8、集群**

**对于用户访问集中的模块需要将独立部署的服务器集群化，即多台服务器部署相同的应用构成一个集群，通过负载均衡设备共同对外提供服务。**

服务器集群能够为相同的服务提供更多的并发支持，因此当有更多的用户访问时，只需要向集群中加入新的机器即可；另外可以**实现当其中的某台服务器发生故障时**，可以通过**负载均衡的失效转移机制**将请求转移至集群中其他的服务器上，因此可以提高系统的可用性。

**9、缓存**

**缓存目的就是减轻服务器的计算，使数据直接返回给用户。**在现在的软件设计中，缓存已经无处不在。具体实现有**CDN、反向代理、本地缓存、分布式缓存**等。

使用缓存有**两个条件**：**访问数据热点不均衡**，**即某些频繁访问的数据需要放在缓存中；数据在某个时间段内有效，不过很快过期，否在会因为数据过期而脏读，影响数据的正确性。**

### 11、CAS是什么？

CAS机制中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

从思想上来说，synchronized属于悲观锁，悲观的认为程序中的并发情况严重，所以严防死守，CAS属于乐观锁，乐观地认为程序中的并发情况不那么严重，所以让线程不断去重试更新。

在java中除了上面提到的Atomic系列类，以及Lock系列类夺得底层实现，甚至在JAVA1.6以上版本，synchronized转变为重量级锁之前，也会采用CAS机制。

**CAS的缺点：**

**1） CPU开销过大**

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很到的压力。

**2） 不能保证代码块的原子性**

CAS机制所保证的知识一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用synchronized了。

**3） ABA问题**

这是CAS机制最大的问题所在。（后面有介绍）

1. JAVA中CAS的底层实现

2. CAS的ABA问题和解决办法。

什么是unsafe呢？**Java语言不像C，C++那样可以直接访问底层操作系统，但是JVM为我们提供了一个后门，这个后门就是unsafe。unsafe为我们提供了硬件级别的原子操作。**

至于valueOffset对象，是通过unsafe.objectFiledOffset方法得到，所代表的是AtomicInteger对象value成员变量在内存中的偏移量。我们可以简单的把valueOffset理解为value变量的内存地址。

我们上面说过，CAS机制中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

而unsafe的compareAndSwapInt方法的参数包括了这三个基本元素：valueOffset参数代表了V，expect参数代表了A，update参数代表了B。

正是unsafe的compareAndSwapInt方法保证了Compare和Swap操作之间的原子性操作。

在Java中，AtomicStampedReference类就实现了用版本号作比较额CAS机制。

**1. java语言CAS底层如何实现？**

**利用unsafe提供的原子性操作方法。**

**2.什么事ABA问题？怎么解决？**

当一个值从A变成B，又更新回A，普通CAS机制会误判通过检测。

**利用版本号比较可以有效解决ABA问题。**