\*\*\*\*\*进程间通信

\*\*\*\*进程锁：（信号量，线程锁共享）

\*信号量：sem\_open

\*线程锁共享： pthread\_mutexattr\_setpshared(&mattr, PTHREAD\_PROCESS\_SHARED);

+线程锁：PTHREAD\_PROCESS\_PRIVATE（mutex 的默认属性即为线程锁，进程间私有）

+进程锁：PTHREAD\_PROCESS\_SHARED

\*文件锁:借助 fcntl 函数来实现文件锁

\*\*\*\*线程锁(互斥锁, 自旋锁, 条件变量, 信号量, 悲观锁，乐观锁, 读写锁, 意向锁 )

\*互斥锁

\*自旋锁

\*悲观锁，乐观锁： CAS, ABA问题

\*读写锁： 用互斥锁来替换，不建议使用

\*信号量： 多线程下不建议使用

\*意向锁： mysql使用

\*条件变量：

\*原子操作：

//一部分线程

Pthread\_mutex\_lock(&mutex);

While（queue.isEmpty()）

Pthread\_cond\_wait(&cond, &mutex); //释放mutex 加上wait锁，加上mutex

满足条件后的具体流程

Pthread\_mutex\_unlock(&mutex);

//另一部分线程

Pthread\_mutex\_lock(&mutex);

修改条件

If(条件满足)

Pthread\_cond\_signal(&cond, &mutex); //唤醒一个

Pthread\_mutex\_unlock(&mutex);

//虚假唤醒 ？？

****\*无论是悲观锁还是乐观锁，都是人们定义出来的概念，可以认为是一种思想****

\*乐观锁：乐观锁在操作数据时非常乐观，认为别人不会同时修改数据。因此乐观锁不会上锁，只是在执行更新的时候判断一下在此期间别人是否修改了数据：如果别人修改了数据则放弃操作，否则执行操作。

\*悲观锁：悲观锁在操作数据时比较悲观，认为别人会同时修改数据。因此操作数据时直接把数据锁住，直到操作完成后才会释放锁；上锁期间其他人不能修改数据。

乐观锁的概念中其实已经阐述了他的具体实现细节：主要就是两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是Compare and Swap(CAS)。

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

#### **乐观锁实现**

乐观锁的概念中其实已经阐述了他的具体实现细节：主要就是两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是Compare and Swap(CAS)。

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

注意：乐观锁本身是不加锁的，只是在更新时判断一下数据是否被其他线程更新了

#### **CAS有哪些缺点？**

1. ABA问题

假设有两个线程——线程1和线程2，两个线程按照顺序进行以下操作：

* 线程1读取内存中数据为A；
* 线程2将该数据修改为B；
* 线程2将该数据修改为A；
* 线程1对数据进行CAS操作

在第(4)步中，由于内存中数据仍然为A，因此CAS操作成功，但实际上该数据已经被线程2修改过了。这就是ABA问题。

ABA看起来似乎没有什么危害。但是在某些场景下，ABA却会带来隐患，例如栈顶问题：一个栈的栈顶经过两次(或多次)变化又恢复了原值，但是栈可能已发生了变化。

对于ABA问题，比较有效的方案是引入****版本号****，内存中的值每发生一次变化，版本号都+1；在进行CAS操作时，不仅比较内存中的值，也会比较版本号，只有当二者都没有变化时，CAS才能执行成功。  
Java中的AtomicStampedReference类便是使用版本号来解决ABA问题的。

1. 高竞争下的开销问题

在并发冲突概率大的高竞争环境下，如果CAS一直失败，会一直重试，CPU开销较大。  
针对这个问题的一个思路是****引入退出机制****，如重试次数超过一定阈值后失败退出。当然，更重要的是****避免在高竞争环境下使用乐观锁。****

1. 功能限制

CAS的功能是比较受限的，例如CAS只能保证单个变量（或者说单个内存值）操作的原子性，这意味着：

* 原子性不一定能保证线程安全，例如在Java中需要与volatile配合来保证线程安全
* 当涉及到多个变量(内存值)时，CAS也无能为力。

## **如何选择**

在乐观锁与悲观锁的选择上面，主要看下两者的区别以及适用场景就可以了。

* 当竞争不激烈 (出现并发冲突的概率小)时，乐观锁更有优势，因为悲观锁会锁住代码块或数据，其他线程无法同时访问，影响并发，而且加锁和释放锁都需要消耗额外的资源。
* 当竞争激烈(出现并发冲突的概率大)时，悲观锁更有优势，因为乐观锁在执行更新时频繁失败，需要不断重试，浪费CPU资源。

随着互联网三高架构（高并发、高性能、高可用）的提出，悲观锁已经越来越少的被应用到生产环境中了，尤其是并发量比较大的业务场景。

互斥锁，自旋锁区别：

//自旋锁

While（获取锁（lock） == 获取失败） {

}

//互斥锁

While（获取锁（lock） == 获取失败） {

该线程休眠，等待唤醒

}

//怎么选择哪种锁

1. 锁粒度
2. 时间长短 测试
3. 是否要充分利用cpu，以减少线程的切换

自旋锁（加上衰减银子） + 互斥锁 nginx

\*\*进程锁

一个机器当中多个进程去竞争临界资源的时候 临界资源 共同资源

\*\*分布式锁 redis举例

\*多台机器之间共享一个变量值

\*特征：

+排他性，互斥性

+可重入性

+高可用 合理的时间内给出合理的答复 数据中心 5秒内回复





所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做,等前一件做完了才能做下一件事。

异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。

有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。 例如，我们在socket中调用recv函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。

C++ atomic 内存模型

C++ 栅栏机制 无锁队列