1.mutex

lock\_guard<mutex>

unique\_lock<mutex>;(在有condtion\_varible的时候使用);

2.Condition\_variable

void wait( [std::unique\_lock](http://zh.cppreference.com/w/cpp/thread/unique_lock)<[std::mutex](http://zh.cppreference.com/w/cpp/thread/mutex)>& lock, Predicate pred );阻塞等待直到满足后面pred谓词条件。

可忽略虚假唤醒，等价于 while(!pred){

Wait(lock)

}

此重载可用于在等待特定条件成为 true 时忽略虚假唤醒。注意进入此方法前，必须得到 lock ， wait(lock) 退出后也会重获得它，即lock 能用做 pred() 访问的保障。

http://zh.cppreference.com/w/cpp/thread/condition\_variable/wait

互斥锁非常适用与那些包装小的操作，简单简洁，逻辑清楚。

2.原子操作：

原子对象能够保证数据不会被竞争破坏，并且原子变量能够保证访问得到的原子对象的值总是最最新的。

1. 这涉及到cpu与cache and register

每一个cpu都有自己的cache,某线程某时刻把变量的值读到cache存好，在次使用的时候直接访问cache,然而多线程会设计到该值被其他线程改变，对于原子变量，cpu总是去内存获得。

不过仅仅是原子操作，还不能保证多线程的安全。

因为编译器的优化以及cpu的优化，会造成代码重排，如果操作够大，指令复杂，甚至指令都会优化（这对单线程来说是没有任何问题的，但多线程就可能发生问题），导致与我们预计的结果不一样、

重排：

Int a = 4;

Int b = 3;

Int c = a+4; 针对优化来说，极有可能b的初始化到a前面了，但对于有依赖关系的c,a肯定在c之前。

For example:

thread A executes the following code:

|  |  |
| --- | --- |
|  | K:= 0;    L:= 0;  // [..] - a lot of code skipped    K:= 5;    L:= 1; |

thread B executes the following code:

|  |  |
| --- | --- |
| 1 | if L = 1 then N:= N + K; |

This is predcite; 因此我门更需要定于memory\_order，告诉cpu那些变量需要线程同步

当多个线程访问原子对象时，所有原子操作都会对一个原子对象定义良好的行为:在任何其他原子操作能够访问该对象之前，每个原子操作都是完全在对象上执行的。这保证了这些对象上没有数据竞争，而这正是定义原子性的特性。 \\* 但是，每个线程可能在内存位置上执行操作，而不是原子对象本身:这些操作可能会对其他线程产生可见的副作用。这种类型的参数允许指定操作的内存顺序，以确定这些(可能非原子)可见的副作用是如何在线程间同步的，使用原子操作作为同步点:

对其他线程可见，是因为cpu去内存去取资源，因此可见。

2.1Memory\_order的6种类型。

：1.给cpu多大的代码空间，cpu的性能就多高。

1. memory\_order\_release 就是告诉cpu前面的操作不能重排到这

之后，并且这之前的变量都去内存中找（也就是我们说的对其他线程可见）

Memory\_order\_acquire 就是告诉cpu这之后的在cache的东西都是无效的，跟我去内存中找。

1. load配acuire

store配 release

只要不需要那种while循环来等待的，就直接用relax.

std::memory\_order\_relaxed 单个线程按照代码顺序来，但在多线程基本就是随意。

模型  
std::memory\_order\_release & std::memory\_order\_acquire  
组合他2起达到线程同步的效果。

sd::memory\_order\_seq\_cst

严格的顺序一致，前面后面的都去内存找，性能不高。

1. Relaxed ordering: 在单个线程内，所有原子操作是顺序进行的。按照什么顺序？基本上就是代码顺序（sequenced-before）。这就是唯一的限制了！两个来自不同线程的原子操作是什么顺序？两个字：任意。

2. Release -- acquire: 来自不同线程的两个原子操作顺序不一定？那怎么能限制一下它们的顺序？这就需要两个线程进行一下同步（synchronize-with）。同步什么呢？同步对一个变量的读写操作。线程 A 原子性地把值写入 x (release), 然后线程 B 原子性地读取 x 的值（acquire）. 这样线程 B 保证读取到 x 的最新值。注意 release -- acquire 有个牛逼的副作用：线程 A 中所有发生在 release x 之前的写操作，对在线程 B acquire x 之后的任何读操作都可见！本来 A, B 间读写操作顺序不定。这么一同步，在 x 这个点前后， A, B 线程之间有了个顺序关系，称作 inter-thread happens-before.

3. Sequential consistency: 理解了前面的几个，顺序一致性就最好理解了。Release -- acquire 就同步一个 x，顺序一致就是对所有的变量的所有原子操作都同步。这么一来，我擦，所有的原子操作就跟由一个线程顺序执行似的。（都在内存里找）

<http://zh.cppreference.com/w/cpp/atomic/memory_order>

aquire语义：load 之后的读写操作无法被重排至 load 之前。即 load-load, load-store 不能被重排。

release语义：store 之前的读写操作无法被重排至 store 之后。即 load-store, store-store 不能被重排。

Load 像读 把atomic内的值读出来, store像写 ，把T写进atomic;

Compare\_exchange\_weak;

https://en.cppreference.com/w/cpp/atomic/atomic/compare\_exchange

原子地比较原子对象和非原子参数的值，若相等则进行 atomic\_exchange ，若不相等则进行 atomic\_load