在本系列的上篇，介绍了内存模型的基本概念，接下来看C++11中支持的几种内存模型。

几种关系术语

在接着继续解释之前，先了解一下几种关系术语。

sequenced-before

sequenced-before用于表示**单线程**之间，两个操作上的先后顺序，这个顺序是非对称、可以进行传递的关系。

它不仅仅表示两个操作之间的先后顺序，还表示了操作结果之间的可见性关系。两个操作A和操作B，如果有A sequenced-before B，除了表示操作A的顺序在B之前，还表示了操作A的结果操作B可见。

happens-before

与sequenced-before不同的是，happens-before关系表示的**不同线程**之间的操作先后顺序，同样的也是非对称、可传递的关系。

如果A happens-before B，则A的内存状态将在B操作执行之前就可见。在上一篇文章中，某些情况下一个写操作只是简单的写入内存就返回了，其他核心上的操作不一定能马上见到操作的结果，这样的关系是不满足happens-before的。

synchronizes-with

synchronizes-with关系强调的是变量被修改之后的传播关系（propagate），即如果一个线程修改某变量的之后的结果能被其它线程可见，那么就是满足synchronizes-with关系的。

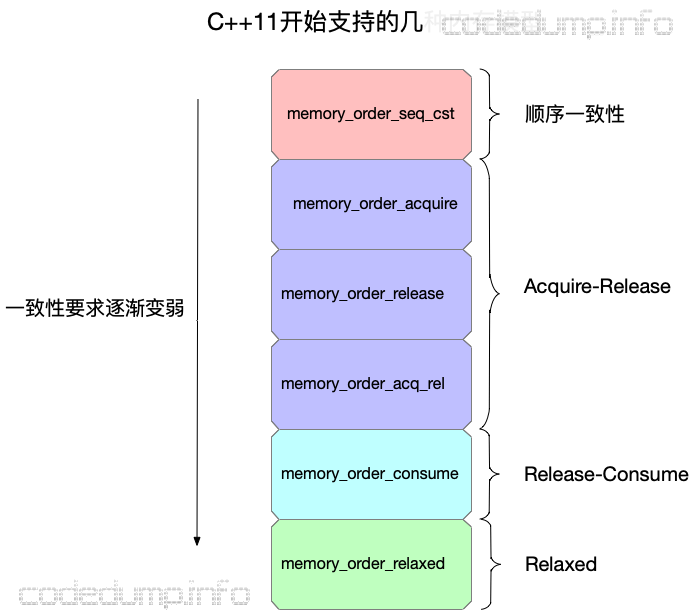
显然，满足synchronizes-with关系的操作一定满足happens-before关系了。

C++11中支持的内存模型

从C++11开始，就支持以下几种内存模型：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | enum memory\_order {  memory\_order\_relaxed,  memory\_order\_consume,  memory\_order\_acquire,  memory\_order\_release,  memory\_order\_acq\_rel,  memory\_order\_seq\_cst  }; |

与内存模型相关的枚举类型有以上六种，但是其实分为四类，如下图所示，其中对一致性的要求逐渐减弱，以下来分别讲解。

[](https://cdn.jsdelivr.net/gh/lichuang/lichuang.github.io/media/imgs/20191214-cxx11-memory-model-2/c++model.png)

memory\_order\_seq\_cst

这是默认的内存模型，即上篇文章中分析过的顺序一致性内存模型，由于在上篇中的相关概念已经做过详细的介绍，这里就不再阐述了。仅列出引用自《C++ Concurrency In Action》的示例代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <atomic>  #include <thread>  #include <assert.h>  std::atomic<bool> x,y;  std::atomic<int> z;  void write\_x()  {  x.store(true,std::memory\_order\_seq\_cst);  }  void write\_y()  {  y.store(true,std::memory\_order\_seq\_cst);  }  void read\_x\_then\_y()  {  while(!x.load(std::memory\_order\_seq\_cst));  if(y.load(std::memory\_order\_seq\_cst))  ++z;  }  void read\_y\_then\_x()  {  while(!y.load(std::memory\_order\_seq\_cst));  if(x.load(std::memory\_order\_seq\_cst))  ++z;  }  int main()  {  x=false;  y=false;  z=0;  std::thread a(write\_x);  std::thread b(write\_y);  std::thread c(read\_x\_then\_y);  std::thread d(read\_y\_then\_x);  a.join();  b.join();  c.join();  d.join();  assert(z.load()!=0);  } |

由于采用了顺序一致性模型，因此最后的断言不可能发生，即在程序结束时不可能出现z为0的情况。

memory\_order\_relaxed

这种类型对应的松散内存模型，这种内存模型的特点是：

* 针对一个变量的读写操作是原子操作；
* 不同线程之间针对该变量的访问操作先后顺序不能得到保证，即有可能乱序。

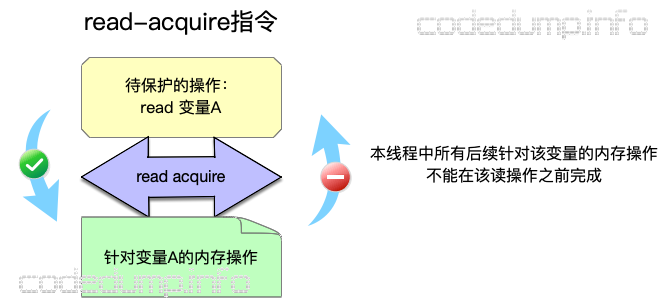
来看示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include <atomic>  #include <thread>  #include <assert.h>  std::atomic<bool> x,y;  std::atomic<int> z;  void write\_x\_then\_y()  {  x.store(true,std::memory\_order\_relaxed);  y.store(true,std::memory\_order\_relaxed);  }  void read\_y\_then\_x()  {  while(!y.load(std::memory\_order\_relaxed));  if(x.load(std::memory\_order\_relaxed))  ++z;  }  int main()  {  x=false;  y=false;  z=0;  std::thread a(write\_x\_then\_y);  std::thread b(read\_y\_then\_x);  a.join();  b.join();  assert(z.load()!=0);  } |

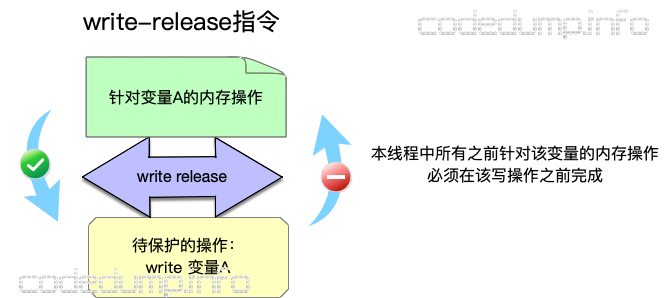
在上面的代码中，对原子变量的访问都使用memory\_order\_relaxed模型，导致了最后的断言可能失败，即在程序结束时z可能为0。

Acquire-Release

* memory\_order\_acquire：用来修饰一个读操作，表示在本线程中，所有后续的关于此变量的内存操作都必须在本条原子操作完成后执行。

[](https://cdn.jsdelivr.net/gh/lichuang/lichuang.github.io/media/imgs/20191214-cxx11-memory-model-2/read-acquire.png)

* memory\_order\_release：用来修饰一个写操作，表示在本线程中，所有之前的针对该变量的内存操作完成后才能执行本条原子操作。

[](https://cdn.jsdelivr.net/gh/lichuang/lichuang.github.io/media/imgs/20191214-cxx11-memory-model-2/write-release.png)

* memory\_order\_acq\_rel：同时包含memory\_order\_acquire和memory\_order\_release标志。

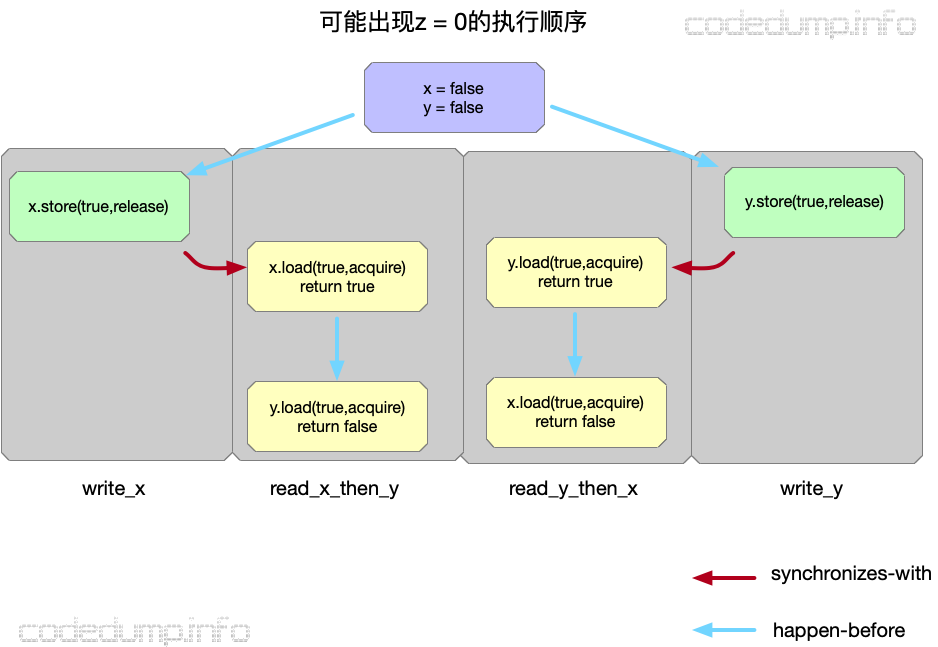
来看示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | // 5.7.cpp  #include <atomic>  #include <thread>  #include <assert.h>  std::atomic<bool> x,y;  std::atomic<int> z;  void write\_x()  {  x.store(true,std::memory\_order\_release);  }  void write\_y()  {  y.store(true,std::memory\_order\_release);  }  void read\_x\_then\_y()  {  while(!x.load(std::memory\_order\_acquire));  if(y.load(std::memory\_order\_acquire))  ++z;  }  void read\_y\_then\_x()  {  while(!y.load(std::memory\_order\_acquire));  if(x.load(std::memory\_order\_acquire))  ++z;  }  int main()  {  x=false;  y=false;  z=0;  std::thread a(write\_x);  std::thread b(write\_y);  std::thread c(read\_x\_then\_y);  std::thread d(read\_y\_then\_x);  a.join();  b.join();  c.join();  d.join();  assert(z.load()!=0);  } |

上面这个例子中，并不能保证程序最后的断言即z!=0为真，其原因在于：在不同的线程中分别针对x、y两个变量进行了同步操作并不能保证x、y变量的读取操作。

线程write\_x针对变量x使用了write-release模型，这样保证了read\_x\_then\_y函数中，在load变量y之前x为true；同理线程write\_y针对变量y使用了write-release模型，这样保证了read\_y\_then\_x函数中，在load变量x之前y为true。

然而即便是这样，仍然可能出现以下类似的情况：

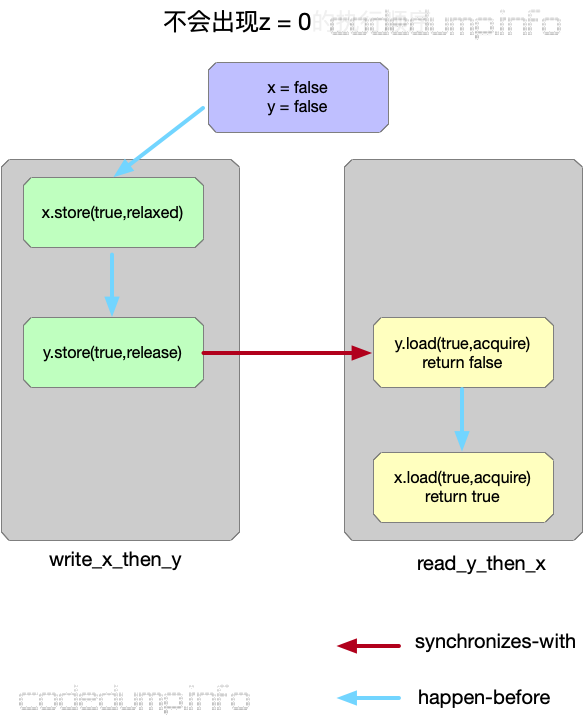
[](https://cdn.jsdelivr.net/gh/lichuang/lichuang.github.io/media/imgs/20191214-cxx11-memory-model-2/5.7.png)

如上图所示：

* 初始条件为x = y = false。
* 由于在read\_x\_and\_y线程中，对x的load操作使用了acquire模型，因此保证了是先执行write\_x函数才到这一步的；同理先执行write\_y才到read\_y\_and\_x中针对y的load操作。
* 然而即便如此，也可能出现在read\_x\_then\_y中针对y的load操作在y的store操作之前完成，因为y.store操作与此之间没有先后顺序关系；同理也不能保证x一定读到true值，因此到程序结束是就出现了z = 0的情况。

从上面的分析可以看到，即便在这里使用了release-acquire模型，仍然没有保证z=0，其原因在于：最开始针对x、y两个变量的写操作是分别在write\_x和write\_y线程中进行的，不能保证两者执行的顺序导致。因此修改如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | // 5.8.cpp  #include <atomic>  #include <thread>  #include <assert.h>  std::atomic<bool> x,y;  std::atomic<int> z;  void write\_x\_then\_y()  {  x.store(true,std::memory\_order\_relaxed);  y.store(true,std::memory\_order\_release);  }  void read\_y\_then\_x()  {  while(!y.load(std::memory\_order\_acquire));  if(x.load(std::memory\_order\_relaxed))  ++z;  }  int main()  {  x=false;  y=false;  z=0;  std::thread a(write\_x\_then\_y);  std::thread b(read\_y\_then\_x);  a.join();  b.join();  assert(z.load()!=0);  } |

[](https://cdn.jsdelivr.net/gh/lichuang/lichuang.github.io/media/imgs/20191214-cxx11-memory-model-2/5.8.png)

如上图所示：

* 初始条件为x = y = false。
* 在write\_x\_then\_y线程中，先执行对x的写操作，再执行对y的写操作，由于两者在同一个线程中，所以即便针对x的修改操作使用relaxed模型，修改x也一定在修改y之前执行。
* 在write\_x\_then\_y线程中，对y的load操作使用了acquire模型，而在线程write\_x\_then\_y中针对变量y的读操作使用release模型，因此保证了是先执行write\_x\_then\_y函数才到read\_y\_then\_x的针对变量y的load操作。
* 因此最终的执行顺序如上图所示，此时不可能出现z=0的情况。

从以上的分析可以看出，针对同一个变量的release-acquire操作，更多时候扮演了一种“线程间使用某一变量的同步”作用，由于有了这个语义的保证，做到了线程间操作的先后顺序保证（inter-thread happens-before）。

Release-Consume

从上面对Acquire-Release模型的分析可以知道，虽然可以使用这个模型做到两个线程之间某些操作的synchronizes-with关系，然后这个粒度有些过于大了。

在很多时候，线程间只想针对有依赖关系的操作进行同步，除此之外线程中的其他操作顺序如何无所谓。比如下面的代码中：

|  |  |
| --- | --- |
| 1  2 | b = \*a;  c = \*b; |

其中第二行代码的执行结果依赖于第一行代码的执行结果，此时称这两行代码之间的关系为“carry-a-dependency ”。C++中引入的memory\_order\_consume内存模型就针对这类代码间有明确的依赖关系的语句限制其先后顺序。

来看下面的示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | // 5.10  #include <string>  #include <thread>  #include <atomic>  #include <assert.h>  struct X  {  int i;  std::string s;  };  std::atomic<X\*> p;  std::atomic<int> a;  void create\_x()  {  X\* x=new X;  x->i=42;  x->s="hello";  a.store(99,std::memory\_order\_relaxed);  p.store(x,std::memory\_order\_release);  }  void use\_x()  {  X\* x;  while(!(x=p.load(std::memory\_order\_consume)))  std::this\_thread::sleep\_for(std::chrono::microseconds(1));  assert(x->i==42);  assert(x->s=="hello");  assert(a.load(std::memory\_order\_relaxed)==99);  }  int main()  {  std::thread t1(create\_x);  std::thread t2(use\_x);  t1.join();  t2.join();  } |

以上的代码中：

* create\_x线程中的store(x)操作使用memory\_order\_release，而在use\_x线程中，有针对x的使用memory\_order\_consume内存模型的load操作，两者之间由于有carry-a-dependency关系，因此能保证两者的先后执行顺序。所以，x->i == 42以及x->s==“hello”这两个断言都不会失败。
* 然而，create\_x中针对变量a的使用relax内存模型的store操作，use\_x线程中也有针对变量a的使用relax内存模型的load操作。这两者的先后执行顺序，并不受前面的memory\_order\_consume内存模型影响，所以并不能保证前后顺序，因此断言a.load(std::memory\_order\_relaxed)==99真假都有可能。

以上可以对比Acquire-Release以及Release-Consume两个内存模型，可以知道：

* Acquire-Release能保证不同线程之间的Synchronizes-With关系，这同时也约束到同一个线程中前后语句的执行顺序。
* 而Release-Consume只约束有明确的carry-a-dependency关系的语句的执行顺序，同一个线程中的其他语句的执行先后顺序并不受这个内存模型的影响。