# CPU、GPU、GPGPU、显卡概念：

CPU：Central processing unit

GPU：Graphics processing unit

GPGPU：General-purpose computing on graphics processing units

显卡（Video card，Graphics card）：由GPU、显存、相关电路等组成，GPU是显卡的核心

* 集成显卡：显卡集成在主板上，不能随意更换
* 独立显卡：作为一个独立的板卡存在，可以随时更换升级

独立显卡有自己的显存；

集成显卡大多使用主机系统内存，也有单独安装了显存，但容量小

# 不同显卡比较

Intel卡：性能比AMD、NVIDIA差多，适合工作要求或者游戏要求不高的人用

AMD显卡：注重平面渲染，专业是工作；生态，驱动，接口等很差；AMD显卡就是冲着图形去的，通用计算稍微支持

NVIDIA显卡：注重3D渲染，专业是游戏；生态，驱动，接口等完善；NVIDIA最近的几代核心都是兼顾图形和通用计算（GPGPU，基于GPU的通用计算技术）的，比较典型的就看分支处理的能力，线程间同步能力，还有双精度计算能力

# 并行计算：

OpenCL stands for Open Computing Language

OpenGL stands for Open Graphic Library

CUDA：Compute Unified Device Architecture

OpenGL：主要是负责2D、3D图形，包括各种光照、纹理、烟雾、阴影等效果。程序员在开发游戏的时候，只要考虑什么时候、在哪个位置，加上什么样的烟雾，就可以了；而不必考虑究竟要怎样去编写代码，才能实现烟雾效果的问题。

CUDA是NVIDIA推出的通用并行计算架构，仅仅能够在NVIDIA的GPU硬件上运行。支持C语言编程，CUDA3.0开始支持C++和FORTRAN。CUDA比起OPENGL、OPENCL更易开发，因为其统一的开发套件(CUDA Toolkit, NVIDIA GPU Computing SDK以及NSight等)、非常丰富的库(傅里叶变换库cuFFT、基本的矩阵和向量运算cuBLAS、并行排序&搜索CUDPP、cuSPARSE、cuRAND、NPP、Thrust)以及NVCC(NVIDIA的CUDA编译器)更成熟的编译器特性。

OpenCL：OpenCL是一个开放标准，目标是面向任何一种Massively Parallel Processor（AMD GPU、NVADIA GPU、DSP、FPGA等等），期望能够对不同种类的硬件给出一个相同的编程模型，跨平台性和通用性好。不如CUDA易开发，只有AMD对OpenCL的驱动相对成熟

# cuda、gpu、N卡驱动等概念区分：

* CUDA全称是Compute Unified Device Architecture，是一个基于GPU的并行计算平台和编程模型
* 显卡：由GPU、显存、相关电路等组成，其中GPU是显卡的核心
* 显卡驱动：通常指N卡驱动。显卡是外设，因此有对应的驱动，类似usb驱动、网卡驱动。
* gpu架构：Tesla、Fermi、Kepler、Maxwell、Pascal。gpu架构是硬件设计方式，如有多少个core、是否有L1/L2缓存
* 芯片型号：GT200、GK210、GM104、GF104等。芯片是对gpu架构思想的实现，型号名称中第二个字母即表明了所使用的gpu架构
* 显卡系列：GeForce、Quadro、Tesla。NVIDIA希望区分显卡为三个系列，GeFore用于家庭娱乐，Quadro用于工作站，而Tesla系列用于服务器。
* GeForce的显卡型号：G/GS、GT、GTS、GTX。GeForce的显卡型号是不同的硬件定制，越往后性能越好，时钟频率越高显存越大，即G/GS<GT<GTS<GTX。

cuda Toolkit，由Nvidia 官方提供的cuda工具集，包含Nvidia 驱动程序、cuda相关的开发工具包。命名如cuda\_9.0.176\_384.81\_linux.run。

安装CUDA Toolkit时，会提示是否安装其中的N卡驱动

Install NVIDIA Accelerated Graphics Driver for Linux-x86\_64 384.81?

(y)es/(n)o/(q)uit: n *# 如果在这之前已经安装好更高版本的显卡驱动就不需要再重复安装，如果需要重复安装就选择 yes*

# cuda toolkit包含以下组件：

1. Compiler: CUDA-C和CUDA-C++编译器。
2. Tools: 提供一些像profiler、debuggers等工具，这些工具可以从bin/目录中获取
3. Libraries: 以下库在lib/目录中，接口在include/目录中

* cudart: CUDA Runtime
* cudadevrt: CUDA device runtime
* cupti: CUDA profiling tools interface
* nvml: NVIDIA management library
* nvrtc: CUDA runtime compilation
* cublas: BLAS (Basic Linear Algebra Subprograms，基础线性代数程序集)
* cublas\_device: BLAS kernel interface
* ...

1. CUDA Samples: 演示如何使用各种CUDA和library API的代码示例。
2. CUDA Driver：每个版本的CUDA工具包都对应一个最低版本的CUDA Driver。如果安装的CUDA Driver版本比官方推荐的还低，那么很可能会无法正常运行。CUDA Driver是向后兼容的，这意味着根据CUDA的特定版本编译的应用程序将继续在后续发布的Driver上也能继续工作。

# nvcc和nvidia-smi显示的CUDA版本不同？

在服务器上nvcc --version显示的结果如下：

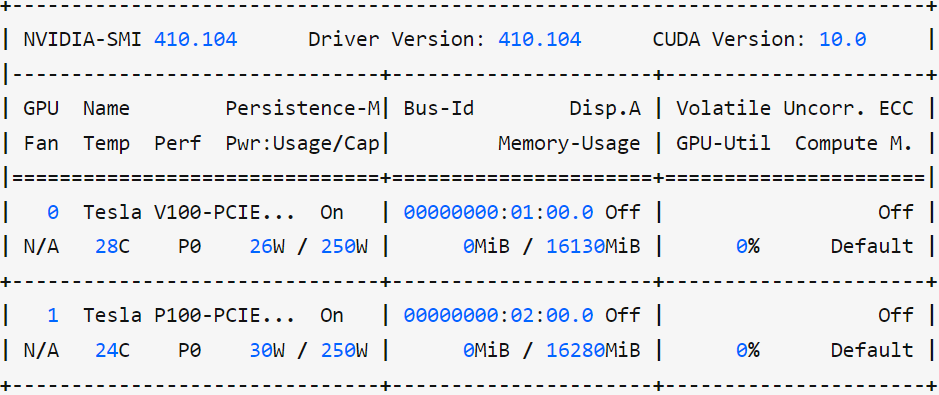
nvcc: NVIDIA (R) Cuda compiler driver

Copyright (c) 2005**-**2018 NVIDIA Corporation

Built on Tue\_Jun\_12\_23:07:04\_CDT\_2018

Cuda compilation tools, release 9.2, V9**.**2.148

而nvidia-smi显示结果如下：



可以看到，nvcc显示的CUDA 版本是9.2。而nvidia-smi显示的CUDA版本是10.0。但绝大多数情况代码也能成功跑起来。

CUDA有两种API：runtime(运行时) API和driver(驱动) API。nvidia-smi 查看的是driver API，nvcc -version 查看的是runtime API。

* 用于支持driver API的必要文件，如libcuda.so，是由GPU driver installer安装的。nvidia-smi就属于这一类API。

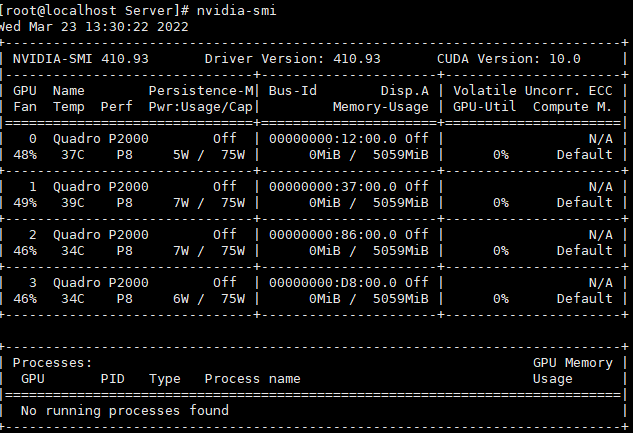
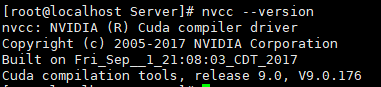
即nvidia-smi显示N卡驱动版本是410.104，cuda driver api的版本cuda 10.0

* CUDA Toolkit installer安装时会生成用于支持runtime API的必要文件，如nvcc、libcudart.so。（CUDA Toolkit Installer有时会集成GPU driver Installer）。

综上，如果driver API和runtime API的CUDA版本不一致可能是因为你使用的是单独的GPU driver installer，而不是CUDA Toolkit installer里的GPU driver installer

nvidia-smi全称是NVIDIA System Management Interface ，它是一个NVIDIA Management Library(NVML)构建的命令行实用工具

示例：

nvidia-smi中可以看到N卡驱动版本是410.93，以及cuda driver api的版本cuda 10.0

安装cuda时， 安装的是cuda 9.0，默认位于/usr/local/cuda-9.0目录，会生成nvcc、libcudart.so等开发工具。nvcc -version即显示的是cuda runtime api版本。上图中是cuda 9.0。当前系统中有：

/usr/local/cuda-9.0/lib64/libcudart.so.9.0.176

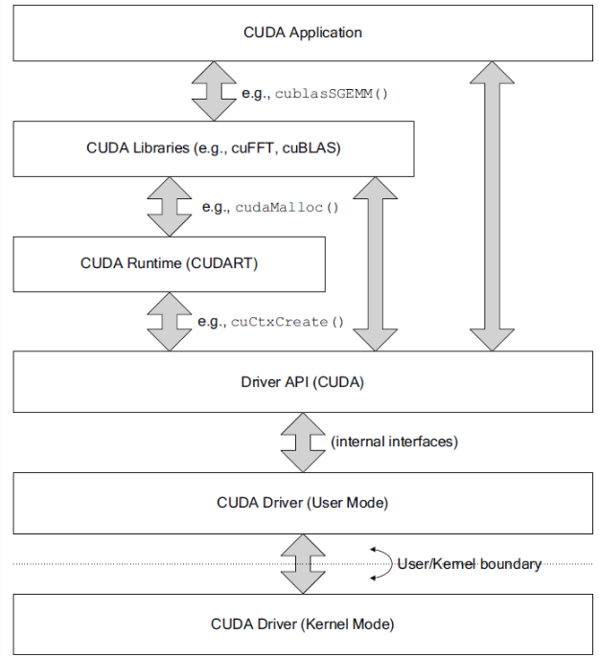
/usr/local/cuda-9.0/lib64/libcudart.so.9.0

# cuda runtime和driver API区别

runtime和driver API在很多情况下用起来的效果是等价的，但是不能混用。runtime是更高级的封装，开发人员用起来更方便，而driver API更接近底层，速度可能会更快。

cu开头的是驱动api

cuda开头的是运行时api



* LIBRARY\_PATH是程序编译期间查找动态链接库时指定的查找路径，对应的是开发阶段
* LD\_LIBRARY\_PATH是程序加载运行期间查找动态链接库时指定除了系统默认路径之外的其他路径

# CUDA编程：

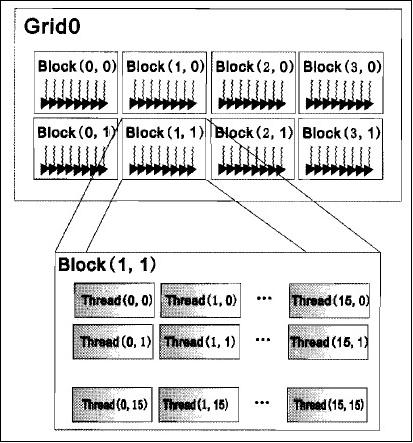
OpenCV的GPU模块实现算法：

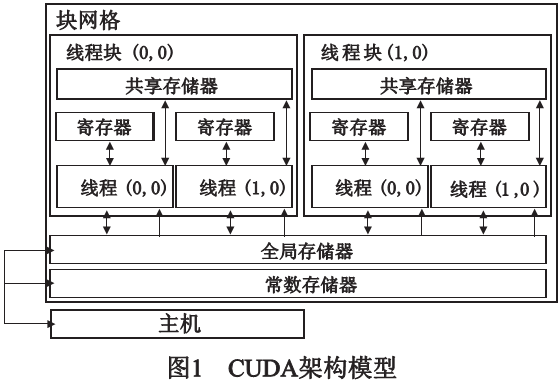
GPU自己的内存为显存。当从硬盘驱动器读数据并产生一个Mat对象的时候，数据是放在系统内存当中（由CPU掌管）。当计算时，需要先让CPU将用于计算的信息转移到GPU掌管的显存上。这是通过一个上传过程完成，需要比访问内存多很多的时间。而且最终的计算结果将要回传到系统内存才能和其他代码交互，由于传输的代价高昂，所以注定移植太小的函数到GPU上并不会提高效率。

1. Mat I1;         // 内存对象，可以用imread来创建
2. gpu::GpuMat gI; // GPU 矩阵 - 现在为空
3. gI1.upload(I1); //将内存数据上传到显存中
4. I1 = gI1;       //回传, gI1.download(I1) 也可以

一旦你的数据被传到GPU显存中，你就只能调用GPU函数来操作

可以被GPU调用并执行的函数称为KERNAL（内核函数）





**\_\_global\_\_**: 用来修饰内核函数的，内核函数是什么呢，内核函数是跑在GPU上的函数**\_\_device\_\_**: 也是用来修饰内核函数的，那和\_\_global\_\_有什么区别吗？对的。\_\_global\_\_修饰的内核函数只能被主机函数调用；\_\_device\_\_修饰的内核函数只能被内核函数调用，应该很好理解

**\_\_host\_\_**: 主机函数，供主机函数调用，跑在CPU上，可缺省哦，一般情况下，都是缺省的。 CPU也叫主机，GPU也叫设备

在主机函数中如果我们分配的是这样的一个东西：

dim3 dimGrid(32, 32);

dim3 dimBlock(16, 16);

kernelfun<<< dimGrid, dimBlock >>>();

其中，dim3是一个内置的结构体。我们调用kernelfun这个内核函数，将dimGrid和dimBlock传到<<<,>>>里去，这句话相当于发号施令，命令那些线程去干活。这里使用了32\*32 \* 16\*16个线程来干活。dimGrid表示将网格被划分为32×32个block（“网格维度是32×32，是二维网格”），dimBlock表示每个block中有16×16个thread（“线程块维度是16×16，是二维线程块”）。

那我们的内核函数kernelfun()如何知道自己执行的是哪个线程？

一个Grid可以包含多个Blocks，Blocks的组织方式可以是一维的，二维或者三维的。block包含多个Threads，这些Threads的组织方式也可以是一维，二维或者三维的。无论是多个block填充grid，还是多个thread填充block，都是先x方向，再y方向，再z方向

1、 grid划分成1维，block划分为1维

int threadId = blockIdx.x \*blockDim.x + threadIdx.x;

2、 grid划分成1维，block划分为2维

int threadId = blockIdx.x \* blockDim.x \* blockDim.y+ threadIdx.y \* blockDim.x + threadIdx.x;

3、 grid划分成1维，block划分为3维

int threadId = blockIdx.x \* blockDim.x \* blockDim.y \* blockDim.z

+ threadIdx.z \* blockDim.y \* blockDim.x

+ threadIdx.y \* blockDim.x + threadIdx.x;

4、 grid划分成2维，block划分为1维

int blockId = blockIdx.y \* gridDim.x + blockIdx.x;

int threadId = blockId \* blockDim.x + threadIdx.x;

5、 grid划分成2维，block划分为2维

int blockId = blockIdx.y \* gridDim.x + blockIdx.x;

int threadId = blockId \* (blockDim.x \* blockDim.y) + (threadIdx.y \* blockDim.x) + threadIdx.x;

6、 grid划分成2维，block划分为3维

int blockId = blockIdx.y \* gridDim.x + blockIdx.x;

int threadId = blockId \* (blockDim.x \* blockDim.y \* blockDim.z)

+ (threadIdx.z \* (blockDim.x \* blockDim.y))

+ (threadIdx.y \* blockDim.x) + threadIdx.x;

gridDim.x —— 线程网格X维度上线程块的数量

gridDim.y —— 线程网格Y维度上线程块的数量

blockDim.x  —— 一个线程块X维度上的线程数量

blockDim.y  —— 一个线程块Y维度上的线程数量

threadIdx.x —— 线程块X维度上的线程索引

threadIdx.y —— 线程块Y维度上的线程索引

cuda定义了一个结构体cudaDeviceProp，里面存入了一系列的结构体变量作为GPU的参数，出了maxThreadsPerBlock，还有很多信息

习惯的可以在内核函数的传递参数名称前加个dev\_作为标示，在内核函数名前价格kernel作为标识

kerneladd<<<10, 1000>>>(dev\_arr);//我们调遣了10个组，每个组用了1000人

#define ROWS 32

#define COLS 16

#define CHECK(res) if(res!=cudaSuccess){exit(-1);}

int \*\*da = NULL;

int \*dc = NULL;

cudaError\_t res;

res = cudaMalloc((void\*\*)(&da), ROWS \* sizeof(int\*)); CHECK(res) //有问题吧？？

res = cudaMalloc((void\*\*)(&dc), ROWS \* COLS \* sizeof(int)); CHECK(res) //有问题吧？？

cudaThreadExit();

 cudaStatus = cudaSetDevice(0);   // cudaStatus为状态指示

cudaThreadSynchronize();

程序自动通过调用cuda API函数获得设备数目和属性：

1. cudaError\_t cudaStatus;
2. **int** num = 0;
3. cudaDeviceProp prop;
4. cudaStatus = cudaGetDeviceCount(&num);
5. **for**(**int** i = 0;i<num;i++)
6. {
7. cudaGetDeviceProperties(&prop,i);
8. }

Geforce（中文一般称为精视™）

在GPU上每个线程都会运行一次该核函数

1. \_\_global\_\_ **void** addKernel(**int** \*c, **const** **int** \*a, **const** **int** \*b)
2. {
3. **int** i = threadIdx.x;
4. c[i] = a[i] + b[i];
5. }

addKernel<<<1, size>>>(dev\_c, dev\_a, dev\_b);

**goto** Error;

1. Error:
2. cudaFree(dev\_c);    //释放GPU设备端内存
3. cudaFree(dev\_a);
4. cudaFree(dev\_b);
5. **return** cudaStatus;

 线程束warp、SM等概念的含义

线程并行为细粒度的并行，而块并行为粗粒度的并行，同时也知道了CUDA的线程组织情况，即Grid-Block-Thread结构。一组线程并行处理可以组织为一个block，而一组block并行处理可以组织为一个Grid，很自然地想到，Grid只是一个网格，我们是否可以利用多个网格来完成并行处理呢？答案就是利用流。

流可以实现在一个设备上运行多个核函数。前面的块并行也好，线程并行也好，运行的核函数都是相同的（代码一样，传递参数也一样）。而流并行，可以执行不同的核函数，也可以实现对同一个核函数传递不同的参数，实现任务级别的并行

# 前向、后向兼容：

新旧是相对的，一个旧软件不能跑在新硬件上，既可以说是这个软件不向前兼容硬件，也可以说是这个硬件不向后兼容软件。再如软件和协议的兼容，新软件不兼容旧协议，既可说是软件不向后兼容协议，也可以说协议不向前兼容软件

前 forward 未来拓展

后 backward 兼容以前

# std::chrono代码段计时：

(1)旧式方法：

#include<ctime>

clock\_t tp1 = clock();

std::this\_thread::sleep\_for(std::chrono::microseconds(5000)); //500微秒

clock\_t tp2 = clock();

cout << (tp2 - tp1) / (double)CLOCKS\_PER\_SEC \* 1000<< "毫秒" << endl;

(2)C++ 11方法：

#include<chrono> //C++ 11

using namespace std::chrono;

auto s1= steady\_clock::now();

do\_some\_work();

auto e1 = steady\_clock::now();

auto durlen1 = duration\_cast<nanoseconds>(e1 - s1).count(); //纳秒,auto为long long类型

auto durlen1 = duration\_cast<microseconds>(e1 - s1).count(); //微秒,long long

auto durlen2 = duration\_cast<milliseconds>(e1 - s1).count(); //毫秒,long long

auto durlen3 = duration\_cast<seconds>(e1 - s1).count(); //秒,long long

auto durlen4 = duration\_cast<minutes>(e1 - s1).count(); //分钟,int

auto durlen5 = duration\_cast<hours>(e1 - s1).count(); //小时,int

std::chrono::steady\_clock::now()返回类型为std::chrono::time\_point< std::chrono::steady\_clock >，即std::chrono::time\_point< std::chrono::steady\_clock, std::chrono::steady\_clock::duration>，std::chrono::steady\_clock::duration即std::chrono:: nanoseconds

C++标准库中：

template <class Rep, class Period = std::ratio<1>> class duration

typedef std::ratio<1, 1000000000> std::nano;

typedef std::ratio<1, 1000000> std::micro; //1秒有1000000个该单位，即微

typedef std::ratio<1, 1000> std::milli;

模板std::chrono::duration<>形如std::chrono::duration<int,std::ratio<1,60>>

typedef duration<long long, nano> nanoseconds;

typedef duration<long long, micro> microseconds;

typedef duration<long long, milli> milliseconds;

typedef duration<long long> seconds;

typedef duration<int, ratio<60> > minutes; //std::ratio<>的第二个模板参数默认值为1

typedef duration<int, ratio<3600> > hours;

得到time\_t，打印日期字符串

auto now = std::chrono::system\_clock::now();

获取当前时刻的time\_t：

* + time\_t t = std::chrono::system\_clock::to\_time\_t(now);
  + std::time\_t t = std::chrono::duration\_cast<std::chrono::seconds>(now.time\_since\_epoch()).count();
  + time\_t t = time(nullptr);

//打印程序：

tm nowtm;

localtime\_s(&nowtm, &t);

cout << std::put\_time(&nowtm, "data:%x\ntime:%X\n"); //C++11函数

//data:12/21/18

//time:11 : 47 : 37

或者

char buf[128];

ctime\_s(buf, 128, &t); //c函数

cout << buf << endl;

//Fri Dec 21 11:47:37 2018

# std::vector<>::emplace\_back()

emplace函数的参数根据元素类型而变化，参数必须与元素类型的构造函数相匹配。emplace函数在容器中直接构造元素。传递给emplace函数的参数必须与元素类型的构造函数相匹配

emplace相关函数可以减少内存拷贝和移动。当插入rvalue，它节约了一次move构造，当插入lvalue，它节约了一次copy构造。

struct President

{

std::string name;

std::string country;

int year;

President(std::string && p\_name, std::string && p\_country, int p\_year)

: name(std::move(p\_name)), country(std::move(p\_country)), year(p\_year) {}

}

std::vector<President> elections;

elections.emplace\_back("Nelson Mandela", "South Africa", 1994); //可以

elections.emplace\_back(President("Nelson Mandela", "South Africa", 1994)); //可以

# C++多线程库：

OpenMP是目前比较流行的C++并行编程方式，它通过在代码中插入专用的pragma编译指令来指示编译器把串行代码编译成并行程序。

一旦采用了OpenMP，线程数量就将由编译器来决定(而不是您)

问题是它必须要有编译器的支持，尽管目前不少编译器都提供了OpenMP的支持，但它毕竟不是C++的一部分

现在，我们又有了一个新选择：Intel Thread Building Blocks（TBB，线程构建模块）。TBB是一个开源的C++模板库，只要是标准的C++编译器都可以使用它。

# thread::join()和thread::detach()：

C++ 11标准支持多线程并行#include<thread>

每个应用程序至少有一个进程，而每个进程至少有一个线程，每个线程都有入口函数（主线程以main函数作为入口函数）。除了主线程外，在一个进程中还可以创建多个线程。每个线程都需要一个入口函数，入口函数返回退出，该线程也会退出。

处于阻塞态的线程不占用cpu时间片

运行态---wait/阻塞io-→阻塞态

运行态-------调度--------→就绪态

就绪态-------调度--------→运行态

阻塞态---信号/io返回-→就绪态

std::thread t(do\_background\_work);

t.detach();

t.join()主调线程阻塞，等待被调线程终止，然后主调线程回收被调线程资源，并继续运行；

detach分离线程，**打破了被调线程与 std::thread 对象的联系。**detach后，被调线程在后台运行，如果主调线程先退出，则由运行时库负责被调线程的资源清理工作。

if(t.joinable())

{

t.join();

}

只要进行过一次t.join()或t.detach()，则t.joinable()值为0。只有当t.joinable()为1的时候，才可以t.join()或t.detach()。因此进行过一次t.join()或t.detach()后不能再进行t.join()或t.detach()了，否则导致错误

**一定要在与线程相关联的thread对象销毁前，确定以何种方式等待线程执行结束**

# 向线程thread()传递参数：

std::thread 可以用函数、函数指针、lambda表达式、bind对象、函数对象构造，也可以使用默认构造。 thread t;

(1)void edit\_document(std::string const filename)

std::thread t(edit\_document,name);

(2)

struct func

{

int& i;

func(int& i\_) : i(i\_) {}

void operator() ()

{

for (unsigned j = 0; j<100; ++j)

{

cout << "good" << i << endl;

}

}

};

func myfunc(num);

thread t(myfunc); //myfunc()

(3)传递给thread入口函数的参数默认是一份拷贝

**class** Task

{

**public**:

**int** age;

**public**:

    Task():age(99) {}

**void** operator()(**int** i)

    {

        age = i;

        cout << i << endl;

    }

};

**int** main()

{

    Task task;

**thread** t(task, 3); //理解为task(3)，调用()运算符重载函数

    t.join();

    cout <<"task.age:" <<task.age << endl;  //结果是99，不是3

}

构造函数在堆中申请内存，存放对象task的副本（副本可拷贝构造而来，也可移动构造而来），副本再调用其()运算符重载函数，因此main()线程中task对象自身的age没有发生改变，依然是99

如果是std::thread t(std::ref(task), 3);则“实现了按引用传递的效果”， main()线程中task对象自身的age发生改变

(4) 需要向线程传递一个引用，由线程执行一些计算最后这个计算结果将在主线程中使用，需要：

* 线程入口函数的形参为引用类型
* 传递给thread构造函数的实参为std::ref(原始实参)

(5) 传递指针给线程函数，thread拷贝的是指针对象到堆中，故线程对指针所指对象的修改将会影响到主线程中的对象

(6) thread类有移动构造和移动赋值函数，delete禁用了拷贝构造和拷贝赋值函数

每个thread实例都负责管理一个执行线程。执行线程的所有权可以在多个 std::thread 实例中互相转移，这是依赖于 std::thread 实例的可移动且不可复制性

一个thread对象只能关联一个线程，想绑定另一个线程则先取消对当前线程的所有权。不能通过赋一个新值给 std::thread 对象的方式来"丢弃"一个线程

如（1）t2=std::move(t1)，再给t1移动赋值；（2）t1.detach()，再给t1移动赋值; （3）t1.join()，再给t1移动赋值

void some\_function();

void some\_other\_function();

std::thread t1(some\_function);

std::thread t2=std::move(t1);

t1=std::thread(some\_other\_function); // std::thread(some\_other\_function)是右值，因此不用move

std::thread t3;

t3=std::move(t2);

(7)传递的参数为类的成员函数

class A

{

public:

void show(int x){cout << "x=" << x << endl;}

};

void main()

{

A a;

thread t(&A::show, &a, 10); //第一个&不可省略，需要一个函数指针。全局函数名可以隐式转换为函数指针，类的成员函数不能隐式转换为函数指针。第二个&若省略则调用temp.show(10)，temp由a拷贝构造而来；若不省略，则调用(&a)->show(10)，等价于std::ref(a)，调用a.show()

t.join();

}

vector<thread> vec\_thread(num\_threads);

for (auto& i:vec\_thread) {i.join();} //不能没有&，否则应调用拷贝构造函数，但thread没有拷贝构造函数

# thread构造函数分析：

参数传递到thread构造函数为完美转发，然后在thread构造函数里，再传递给make\_unique，make\_unique函数模板也为完美转发，make\_unique调用new运算符。因此最后还是拷贝了（或移动了）一份参数到堆中，再将堆中数据传递给线程入口函数。具体地，传入thread构造函数的实参如果是用户类型左值，则拷贝构造一份放于堆中，如果是右值，则移动构造一份放于堆中

std::thread有一可变模板参数的构造函数，

template<class \_Fn, class... \_Args, class = typename enable\_if<

!is\_same<typename decay<\_Fn>::type, thread>::value>::type>

explicit thread(\_Fn&& \_Fx, \_Args&&... \_Ax)

{ // construct with \_Fx(\_Ax...)

\_Launch(&\_Thr,

\_STD make\_unique<tuple<decay\_t<\_Fn>, decay\_t<\_Args>...> >(

\_STD forward<\_Fn>(\_Fx), \_STD forward<\_Args>(\_Ax)...));

}

make\_unique函数模板定义：

template<class \_Ty,

class... \_Types> inline

typename enable\_if<!is\_array<\_Ty>::value,

unique\_ptr<\_Ty> >::type make\_unique(\_Types&&... \_Args)

{ // make a unique\_ptr

return (unique\_ptr<\_Ty>(new \_Ty(\_STD forward<\_Types>(\_Args)...)));

}

调用\_Launch函数启动新的线程，

template<class \_Target> inline

void \_Launch(\_Thrd\_t \*\_Thr, \_Target&& \_Tg)

{ // launch a new thread

\_LaunchPad<\_Target> \_Launcher(\_STD forward<\_Target>(\_Tg));

\_Launcher.\_Launch(\_Thr);

}

代码：

**void** f(**int** i, std::string **const**& s);

**void** not\_oops(**int** some\_param)

{

**char** buffer[1024];

    sprintf(buffer, "%i", some\_param);

    std::**thread** t(f, 3, std::string(buffer));

    t.detach();

}

错误分析：thread构造函数中有string对象，f()函数里的引用s绑定该对象，由于detach可能父线程先结束，这时t销毁，thread构造函数中string对象销毁，引用悬空？？

正确分析：thread构造函数并没有在栈中创建string对象，thread模板类完美转发，然后在堆中申请内存放string对象的副本。这样即使主线程结束退出，栈销毁，s绑定的string对象依然在堆中，不会引用悬空

# 互斥量mutex、unique\_lock、lock\_guard

#include<mutex>

* 互斥量std::mutex加锁，访问该互斥量的其它所有线程阻塞
* 互斥量解锁，因这个互斥量阻塞的所有线程进入就绪态，第一个获得cpu的线程会加锁该互斥量，变为运行态
* 对同一个互斥量mutex，同一时刻只能有一个线程能对其上锁

锁上互斥量之后必须解锁，否则其他线程永远无法上锁，就可能陷入无休止的等待

mutex::lock()

……

mutex::unlock()

数据保护的原理：mutex::lock()如不能上锁则阻塞；直到成功上锁后返回，这样lock()与unlock()之间的代码才能执行。

std::mutex 的成员函数

* 构造函数，std::mutex不允许拷贝构造、拷贝赋值，也不允许 move构造、移动赋值，最初产生的 mutex 对象是处于 unlocked 状态的。
* std::mutex ::lock()： (1). 如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock之前，该线程一直拥有该锁。(2). 如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。
* std::mutex::try\_lock()，尝试锁住互斥量。 (1). 如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量。(2). 如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉。

unique\_lock比lock\_guard更加灵活

unique\_lock() noexcept;   //可以构造一个空的unique\_lock对象，此时并不拥有任何mutex

**explicit** unique\_lock (mutex\_type& m);//拥有mutex，并调用mutex::lock()对其上锁

unique\_lock (mutex\_type& m, try\_to\_lock\_t tag);//tag=try\_lock表示调用mutex.try\_lock()尝试加锁

unique\_lock (mutex\_type& m, defer\_lock\_t tag) noexcept;//tag=defer\_lock表示不对mutex加锁，只管理mutex，此时mutex应该是没有加锁的。这样，就可以用 std::unique\_lock 对象(不是互斥量)的lock()函数加锁，或传递 std::unique\_lock 对象到 std::lock()来加锁

unique\_lock (mutex\_type& m, adopt\_lock\_t tag);//tag=adopt\_lock表示mutex在此之前已经被上锁，此时unique\_locl管理mutex

unique\_lock (**const** unique\_lock&) = **delete**;//禁止拷贝构造

unique\_lock (unique\_lock&& x);// 移动构造函数。获得x管理的mutex，此后x不再和mutex相关，x此后相当于一个默认构造的unique\_lock对象

* lock\_guard不可移动不可拷贝。构造时必须传递一个mutex实例。lock\_guard(mut)，调用mut.lock()，获得锁的管理权；lock\_guard(mut,std::adopt\_lock)，仅获得管理权，之后可以调用mut.lock()上锁。析构函数中调用锁的unlock()函数。
* std::thread可移动不可拷贝（无拷贝构造和拷贝赋值函数），移动时thread管理的线程转移到新的thread对象，原thread对象不再和线程相关，相当于一个默认构造的std::thread对象
* unique\_lock可移动不可拷贝，构造时可以传递锁及std::defer\_lock和std::adopt\_lock参数，可以无参构造。可以调用unique\_lock::lock()对互斥量上锁，可以调用unique\_lock::unlock()主动释放锁。unique\_lock类中有一个标志记录是否管理一个未上锁的互斥量锁，如果是，则析构函数中调用unlock()；如果否，则析构函数就不调用unlock()。

切勿将锁数据的指针或引用传递到互斥锁作用域之外，包括：

* 成员函数返回受保护数据的指针或引用
* 以指针或引用的形式将受保护收据传递给用户提供的函数。如在用户提供的函数中可能将锁保护的数据的引用又传递给一个全局引用变量，这样在其它时候读写全局引用，则突破了锁限制

资源管理类thread、unique\_lock、unique\_ptr的共同点：

* 线程所有权的转移，由一个thread对象到另一个thread对象
* 互斥量mutex的所有权转移，由一个unique\_lock对象到另一个unique\_lock对象
* 指针的所有权转移，由一个unique\_ptr对象到另一个unique\_ptr对象

因此，thread、unique\_lock、unique\_ptr支持移动构造、移动赋值、无参构造函数(管理的资源被转移走后相当于用无参构造函数构造的对象，不再管理任何资源)

**pthread\_mutex\_unlock**

一个线程std::mutex:: lock持有锁，另一个线程std::mutex::unlock这个锁，是未定义行为;

manpage中如果当前线程不持有锁，pthread\_mutex\_unlock(pthread\_mutex\_t\*)返回EPERM，但在gcc4.8.5+centos7.7环境实际测试发现，当锁被另一个线程持有时，pthread\_mutex\_unlock会成功解锁，不会返回错误

# 条件变量condition\_variable：

#include<condition\_variable>

condition\_variable cond;

线程1

{

……

cond.notify\_one()或cond.notify\_all()

}

线程2

{

std::unique\_lock<std::mutex> lk(mut);

cond.wait(lk,[]{return !queue.empty();});

……

}

condition\_variable::notify\_one()唤醒一个等待唤醒的线程

std::condition\_variable 提供了两种 wait() 函数。

|  |
| --- |
| (1)void wait (unique\_lock<mutex>& lck); |
| (2)template <class Predicate>  void wait (unique\_lock<mutex>& lck, Predicate pred); |

void wait(Lck, Pred)完全等价于

while (!Pred())

{

wait(\_Lck);

}

(1)当前线程调用 wait() 后将被阻塞(此时当前线程应该获得了锁（mutex）)，该函数会自动调用 lck.unlock() 释放锁，使得其他被阻塞在锁竞争上的线程得以继续执行。另外，一旦当前线程获得通知(notified，通常是另外某个线程调用 notify\_\* 唤醒了当前线程)，wait() 函数也是自动调用 lck.lock()

(2)wait(unique\_lock<mutex>& lck, Predicate pred)的条件满足，程序继续执行；不满足，解锁，程序阻塞，等待唤醒。其余线程的condition\_variable::notify\_\*()语句执行，被唤醒，加锁，检查等待条件是否达成，不达成，解锁，继续阻塞，等待唤醒；达成，保持加锁状态不变，解除阻塞，程序继续向后执行

Predicate是返回类型为bool的可调用对象，如lambda表达式、函数对象、函数、函数指针等

do I need a mutex lock when calling cv.notify\_one() in each thread?

No

is std::condition\_variable thread-safe?

Yes

# 条件变量wait与锁：

以下程序存在典型问题，可能阻塞在join处

#include <thread>

#include <condition\_variable>

#include <mutex>

std::condition\_variable condvar;

std::mutex mut;

int k = 0;

void first()

{

k = 1;

condvar.notify\_all();

}

void second()

{

std::unique\_lock<std::mutex> lock(mut);

condvar.wait(lock, []() { return k == 1; });

}

int main()

{

std::thread td0(first);

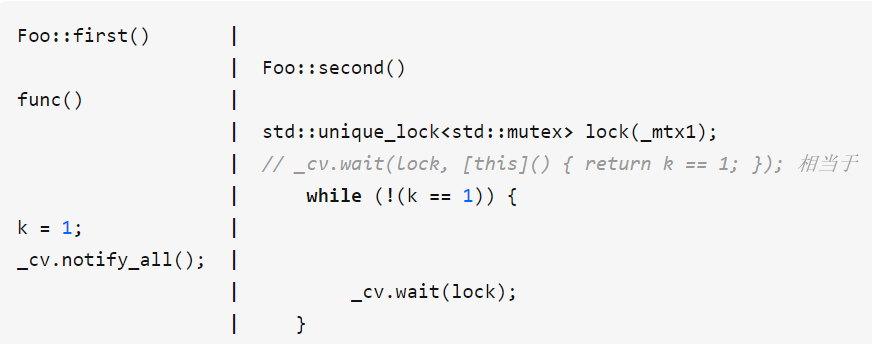
std::thread td1(second);

td0.join();

td1.join();

}

一种可能的顺序是这样的：



因此如下修改first()函数即可：

void first()

{

{

std::unique\_lock<std::mutex> lock(mut);

k = 1;

}

condvar.notify\_all();

}

条件变量wait时需要一个锁的原因就是为了保证，条件不满足和进入等待状态是一个原子操作，在这期间不允许有人修改条件使满足。但是只有wait加锁还不能完全保证，在其它线程修改条件时，也必须加锁。这样才保证正确性

# condition\_variable机制：

假设共5个线程，分别为线程0、1、2、3、4。线程1和线程2调用cond.wait(unique\_lock,condition)时因condition不满足而阻塞，释放锁。线程3、4不等待条件变量，但也竞争锁，阻塞。线程0，对同一互斥量加锁, 执行，改变条件condition使为之真, 然后调用函数 cond.notify\_one ()。系统选择等待被唤醒的两个线程中的某一个，不妨设为线程1，让其由等待唤醒状态变为等待锁状态。在线程0对mutex解锁后，系统保证让线程1优先获得mutex并加锁，而不是线程3、4，更不是线程2（因为线程2仍处于等待唤醒的状态）。线程1继续执行，执行完毕释放锁。然后线程3、4竞争锁。

如果线程0中调用的是cond.notify\_all ()，则系统将所有等待被唤醒的线程唤醒，让它们由等待唤醒状态变为等待锁状态。在线程0对mutex解锁后，线程1和线程2竞争锁，不妨设线程2竞争到锁，则线程2由wait()语句处继续执行，执行完毕后，释放锁。线程1获得锁，由wait()语句处继续执行，执行完毕后，释放锁。然后线程3、4竞争锁。

notify\_all使所有wait唤醒的的线程，变成等待该对象上的锁，一旦该对象上的锁被释放，则他们就会去竞争。

notify\_one则只是选择一个等待notify的线程进行通知，让其退出wait唤醒的状态，变成等待锁，但不惊动其他同样等待被notify的线程们。当被选择的那一个线程运行完毕以后释放锁，在此过程中，如果该对象没有再次使用notify\_\*语句，则即便该对象已经空闲，其他wait唤醒的线程由于没有得到该对象的通知，继续处在wait唤醒状态（直到这个对象发出一个notify\_one或notify\_all，它们等待的是被notify\_one或notify\_all，而不是锁。）

等待唤醒🡪被notify\_\*语句唤醒🡪获取锁（notify\_one()）或竞争到锁（notify\_all()）🡪继续执行

# std::cv\_status 枚举类型介绍

std::cv\_status condition\_variable::wait\_for(mutex&, std::chrono::duration<>&)

std::cv\_status condition\_variable::wait\_until(mutex&, std::chrono::time\_point<>&)

|  |  |
| --- | --- |
| cv\_status::no\_timeout | wait\_for 或者 wait\_until 没有超时被唤醒 |
| cv\_status::timeout | wait\_for 或者 wait\_until 超时。 |

# 死锁：

完成一个任务，需要资源r1、r2，两个资源都有锁保护

A先对r1上锁，然后尝试对r2上锁，之后才会解锁r2、r1

B先对r2上锁，然后尝试对r1上锁，之后才会解锁r1、r2

解决方法：

在一个任务需要获取多把锁，以固定的顺序上锁。但有时候我们无法在代码中定义这样的顺序。比如，假设有两个对象，它们是同一个类的两个实例。现在，我们希望在某个线程里，交换二者的内容

C++标准库提供了 std::lock() 函数，用于同时锁住多个互斥量，并且没有死锁的风险，lock(lhs.mtx\_, rhs.mtx\_)

避免死锁：

std::lock(lhs.m,rhs.m); // 1

std::lock\_guard<std::mutex> lock\_a(lhs.m,std::adopt\_lock);// 2

std::lock\_guard<std::mutex> lock\_b(rhs.m,std::adopt\_lock);// 3

lock()的参数是两个互斥锁，要么将两个锁都锁住，要么不一个都不锁

当 std::lock()成功的获取一个互斥量上的锁，并尝试从另一个互斥量上再获取锁时如果失败，就会有异常抛出，第一个锁也会随着异常的产生而自动释放所以std::lock. 异常会传播  
到 std::lock 之外。

第二行、第三行代码实现lock\_guard 对象在析构时，自动解锁，其中参数adopt\_lock表示mutex已经在之前被上锁，这里lock\_guard将拥有mutex的所有权

或者：

std::unique\_lock< std::mutex> unilock1(lhs.m, std::defer\_lock);

std::unique\_lock<std::mutex> unilock2(rhs.m.std::defer\_lock);

std::lock(unilock1,unilock2);

锁是产生死锁的一般原因，但无锁也可能产生死锁。两个std::thread 对象互相调用对方的join()函数。

**void** task(**thread**& t)

{

**for** (**int** i = 0; i < 10; i++)

    {

**for** (**int** j = 0; j < 1000000; j++) {  }

        cout << i << " ";

    }

    t.join();

}

**int** main()

{

**thread** t1;

**thread** t2;

    t1 = **thread**(task, std::ref(t2));

    t2 =  **thread**(task, std::ref(t1));

    std::cin.get();

}

# 原子类型与原子操作

(1) 除非所有线程都只是读数据，则不用锁保护。如果同时有线程读、有线程写，则一定需要锁保护。

C++中，多个线程对同一变量的同时读写是未定义行为，除非使用stomic<>

遍历打印单链表元素。“手递手”锁的模式。链表的每个节点都会有一个互斥量保护，线程必须保证在获取当前节点的互斥锁前提下，获得下一个节点的锁。一旦下一个节点上的锁被获取，那么第一个节点的锁就可以释放了，因为没有持有它的必要性了。

std::mutex mut[1024];

mut[0].lock();

for (int i = 0; true; i++)

{

cout << p->data << " ";

mut[i+1].lock()

p = p->next;

mut[0].unlock();

}

如果不上锁，获取指针p后，可能另一线程删除了p指向的节点，这样p->data、p->next出错

(2)

在单处理器系统（UniProcessor）中，能够在单条指令中完成的操作都可以认为是“原子操作”，因为中断只能发生于指令之间。但是，在对称多处理器（Symmetric Multi-Processor）结构中就不同了，由于系统中有多个处理器在独立地运行，即使能在单条指令中完成的操作也有可能受到干扰。我们以decl （递减指令）为例，这是一个典型的"读－改－写"过程，涉及两次内存访问。设想在不同CPU运行的两个进程都在递减某个计数值，可能发生的情况是：

⒈ CPU A(CPU A上所运行的进程，以下同）从内存单元把当前计数值⑵装载进它的寄存器中；

⒉ CPU B从内存单元把当前计数值⑵装载进它的寄存器中。

⒊ CPU A在它的寄存器中将计数值递减为1；

⒋ CPU B在它的寄存器中将计数值递减为1；

⒌ CPU A把修改后的计数值⑴写回内存单元。

⒍ CPU B把修改后的计数值⑴写回内存单元。

我们看到，内存里的计数值应该是0，然而它却是1。

# 线程安全单例与双重检查锁

x86汇编中，对任何内存地址中的1byte的读永远是原子的.也就是说对一个char的读取永远是原子的，对内存地址对齐2byte的int16类型的读取是原子的，对4byte对齐的int32类型读取是原子的，从奔腾开始，对8byte对齐地址的int64读取是原子的.所以如果你用的是汇编，保证这些就行了.

但C/C++中又是另一番情景:C/C++中，编译器保证基础类型的内存对齐，例如保证double类型的对齐是8(或者4，忘了)，即使是malloc出来的也可以保证对齐.但是由于各种不可避免的指针转换，例如 char a[4]，float\* p=(float\*)a的存在，使得对齐的保证基本名存实亡.而且，当一个比较长的类型，例如double被编译器放入寄存器的时候，C++标准根本不保证只用一条指令就将它放入一个寄存器中.例如我可以先把前半部分放入eax，等一会儿再把后半部分放入edx等等.不过，如果你能够确保对齐，那么大多数情况下虽然UB，但你的代码还是有可能正常工作的.

再然后，其实上面说的根本不用考虑，因为在C/C++标准中，除了使用atomic<>和相关函数外，多个线程对对同一个变量同时进行读写都是UB（未定义行为）

线程安全单例类实现，声名狼藉的双锁检查：

**class** SingleTon

{

**public**:

**static** SingleTon\* getInstance()   {

**if** (p==nullptr)   {

            std::unique\_lock<std::mutex> unilk(mut);

**if** (p==nullptr)  p = **new** SingleTon(); //双检锁，在进入临界区后再检测一次是否对象已经建立

        }

**return** p;

    }

**private**:

**static** SingleTon\* p;

**static** std::mutex mut;

};

SingleTon\* SingleTon::p = nullptr;

std::mutex SingleTon::mut;

上边代码存在3个问题：

* 存在对同一变量p的同时读写，一个线程执行临界区内p = **new** SingleTon()写，另一个线程**if** (p==nullptr)  读p值
* 指令重排序，语句p = **new** SingleTon()有两种指令顺序：(a)堆中分配内存、构造类对象、p指向内存；(b) 堆中分配内存、p指向内存、在分配的内存上构造类对象。对第二种指令顺序，p指向分配的内存后，已然非nullptr，另一线程if (p==nullptr) 判断通过，直接返回p，但此时p并未指向类对象，因为在分配的内存上还未构造类对象，因此出错。
* 没有delete p，造成内存泄漏

解决方案：

（1）增大锁粒度，解决同时读写同一变量问题、指令重排序问题

static SingleTon\* getInstance()

{

{

std::unique\_lock<std::mutex> unilk(mut);

if (p == nullptr) p = new SingleTon();

}

return p;

}

（2）使用call\_once代替锁，消耗的资源更少，特别是当初始化完成后。

使用辅助嵌套类，帮助内存回收

**class** SingleTon

{

**public**:

**static** SingleTon\* getInstance()

    {

        std::call\_once(flag, []() {**if** (p == nullptr)  p = **new** SingleTon(); });//可以直接使用静态成员变量，不能捕获[this]，this不能用于非静态成员变量

**return** p;

    }

**private**:

**static** SingleTon\* p;

**static** std::once\_flag flag;

**class** Assist

    {

**public**:

        ~Assist() { **if**(p!=nullptr) **delete** p; }  //嵌套类直接访问p

    };

**static** Assist gabage;

};

SingleTon\* SingleTon::p = nullptr;

std::once\_flag SingleTon::flag;

SingleTon::Assist SingleTon::gabage;

(3)最优单例模式设计：线程安全、自动垃圾回收

static成员函数，里有static局部变量，返回引用，delete拷贝构造和拷贝赋值函数

**class** Singleton

{

**public**:

**static** Singleton& getInstan()

    {

**static** Singleton inst;

**return** inst;

    }

    Singleton(**const** Singleton&) = **delete**; //同时也delete了移动构造

    Singleton& operator=(**const** Singleton&) = **delete**; //同时也delete了移动赋值

**private**:

    Singleton() = **default**; //必须显式定义，因为自定义了拷贝构造（虽然是delete），因此编译器不再生成构造函数

};

**int** main()

{

    Singleton& single=Singleton::getInstan();

    //Singleton single = Singleton::getInstan(); //报错，不能调用拷贝构造；如果单例类定义时没有delete，则此时调用编译器生成的拷贝构造，会有两个实例

}

C++11中，局部static变量的初始化是线程安全的。

在很多以前的编译器，对局部static变量的定义及初始化存在条件竞争。因为在多线程中，每个线程都认为他们是第一个初始化这个变量的线程；或一个线程对变量进行初始化，而另外一个线程要使用这个变量时，初始化过程还没完成。在C++11标准中，这些问题都被解决了：初始化及定义完全在一个线程中发生，并且没有其他线程可在初始化完成前对其进行处理

class my\_class {};

my\_class& get\_my\_class\_instance()

{

static my\_class instance; // 线程安全的初始化过程

return instance;

}

# 线程id和线程数的确定：

std::thread::hardware\_concurrency()返回值是CPU核芯的数量。返回值为0表示系统信息无法获取时

内置 （built-in) 类型和用户自定义(user-defined)类型

std::thread::id，获取线程id的两种方法：std:this\_thread::get\_id()或者调用 std::thread 对象的成员函数 get\_id()

std::thread::id可以作为容器的键值，包括std::map等需要比较键值大小的容器，和std::hash<std::thread::id>容器比较键值是否相等

std::cout<<std::this\_thread::get\_id()

165个任务，8个线程，每个线程负责几个任务？165/8=20，前7个线程各负责21个，最后一个线程负责18个任务，这样总体耗时最短。错误方案为前7个线程各负责20个，最后一个线程负责25个任务

const unsigned long min\_per\_thread = 25;

const unsigned long max\_threads = (length+min\_per\_thread-1) / min\_per\_thread;

unsigned long hardware\_threads = std::thread::hardware\_concurrency();

hardware\_threads = hardware\_threads == 0 ? 2 : hardware\_threads;

const unsigned long num\_threads = max\_threads < hardware\_threads ? max\_threads : hardware\_threads;

const unsigned long block\_size = (length+ num\_threads-1) / num\_threads;//前几个线程执行各block\_size个任务，最后一个线程小于等于该值

# 线程安全的栈：

线程安全的类，数据与保护数据的互斥锁作为private成员，设计对数据进行操作的函数时要考虑到多个线程同时调用对象的多个成员函数或同一成员函数时的情况，因此在函数内部合理上锁解锁

多线程栈的设计：

(1)pop()不应按值返回弹出的元素：

当pop()函数按值返回“弹出值”时(也就是从栈中将这个值移除)，会有一个潜在的问题：

当拷贝数据到调用函数空间的时候，调用函数抛出一个异常会怎么样？（当这个系统处

在重度负荷，或有严重的资源限制的情况下，内存分配就会失败。）这样，要弹出的数

据将会丢失：它的确从栈上移出了，但是拷贝失败了！ 因此传递引用，接收弹出值。或者，返回std::share\_ptr<T>

为避免固有接口上的条件竞争，需要将get\_top()和pop\_back()合并成一个函数调用

**void** pop\_back(T& topval)

{

    std::unique\_lock<mutex> uk(mut);

**if** (empty()) **throw** "empty exception";// 在调用pop前，检查栈是否为空

    topval = data[top--];

}

std::shared\_ptr<T> pop()

{

    std::lock\_guard<std::mutex> lock(m);

**if** (empty()) **throw** "empty exception";// 在调用pop前，检查栈是否为空

**const** std::shared\_ptr<T> res(std::make\_shared<T>(data[top--])); // 在修改堆栈前，分配出返回值

**return** res;

}

(2)size()、empty()。get\_top()等接口失去意义，不需要向用户提供这些接口。

虽然empty()和size()可能在被调用并返回时是正确的，但其的结果是不可靠的；当它们返回后，其他线程就可以自由地访问栈，并且可能push()多个新元素到栈中，也可能pop()一些已在栈中的元素。这样的话，之前从empty()和size()得到的结果就有问题了。

(3) 向用户提供的接口只剩下void push\_back(const T& val)和void pop\_back(T& topval)或std::shared\_ptr<T> pop\_back()

(4)可以有拷贝构造

**template**<**class** T>

**class** threadsafe\_stack

{

**private**:

**static** **const** unsigned MAXSIZE = 1024;

**mutable** mutex mut;

    T data[MAXSIZE];

**public**:

**int** top;

    threadsafe\_stack<T>& operator=(**const** threadsafe\_stack<T>& rhs) = **delete**;

    threadsafe\_stack(**const** threadsafe\_stack<T>& rhs)

    {

        std::unique\_lock<std::mutex> unilk(rhs.mut);

        top = rhs.top;

        memcpy(data, rhs.data, **sizeof**(data));

    }

}

# thread\_loacl存储类型的变量

thread\_local变量是C++ 11新引入的一种存储类型，与automatic、static、dynamic并列

automatic：exists during a block/function

static：exists for the program duration

dynamic： exists on the heap between allocation and deallocation

thread\_local修饰的变量具有线程周期（线程局部存储，Thread Local Storage，TLS）：

* 在线程创建时生成(不同编译器实现略有差异，但在线程内第一次使用前必然已构造完毕)。
* 在线程结束的时候销毁。
* 每一个线程都拥有一个独立的变量实例。

thread\_local修饰的变量包括：全局变量（全局、文件作用域、命名空间作用域）、类的static成员变量、局部变量

thread\_local变量在每个线程中都被独立初始化。同一个线程，thread\_local变量仅被初始化一次，之后使用上次的值。

对于Windows系统，编译器会把thread\_local变量放到PE文件的".tls"段中。当系统启动一个新的线程时，它会从进程的堆中分配一块足够大小的空间，然后把".tls"段中的内容复制到这块空间中，于是每个线程都有自己独立的一个".tls"副本。即thread\_local定义的同一个变量，在不同线程中的地址都是不一样的。对于一个TLS变量来说，它有可能是一个C++的全局对象，那么每个线程在启动时不仅仅是复制".tls"的内容那么简单，还需要把这些TLS对象初始化，必须逐个地调用它们的全局构造函数，而且当线程退出时，还要逐个地将它们析构，正如普通的全局对象在进程启动和退出时都要构造、析构一样

#include<thread>

thread\_local **int** g\_n = 1;

**void** f()

{

    g\_n++;

    printf("id=%d, n=%d\n", std::this\_thread::get\_id(), g\_n);

}

**void** foo()

{

    thread\_local **int** i = 0;

    printf("id=%d, n=%d\n", std::this\_thread::get\_id(), i);

    i++;

}

**void** f2()

{

    foo();

    foo();

}

**int** main()

{

    g\_n++;

    f();

    cout << "--------" << endl;

    std::**thread** t1(f);

    std::**thread** t2(f);

    t1.join();

    t2.join();

    cout << "--------" << endl;

    f2();

    cout << "--------" << endl;

    std::**thread** t4(f2);

    std::**thread** t5(f2);

    t4.join();

    t5.join();

}

/\*

id=6580, n=3

--------

id=18556, n=2

id=15224, n=2

--------

id=6580, n=0

id=6580, n=1

--------

id=18240, n=0

id=18240, n=1

id=13640, n=0

id=13640, n=1

\*/

# std::once\_flag和call\_once()

在多线程编程中，有一个常见的情景是某个任务只需要执行一次。

call\_once回调的函数只会被成功执行一次

std::once\_flag flag;

void do\_once(int i)

{

std::call\_once(flag, [i]() { std::cout << "Call\_once i=" <<i<< std::endl; });

}

int main()

{

std::thread t1(do\_once,1);

std::thread t2(do\_once,2);

std::thread t3(do\_once,3);

t1.join();

t2.join();

t3.join();

}

/\*运行结果：Called once i=1\*/

如果call\_once回调的函数是类的成员函数，则传入参数的方式形如：

std::call\_once(flag,&X::open\_connection,this)

如果在call\_once()回调的函数在执行中抛出了异常，那么会有另一个在once\_flag上等待的线程执行。

std::once\_flag flag;

inline void may\_throw\_function(bool do\_throw)

{

if (do\_throw)

{

std::cout << "throw" << endl;

throw "exception";

}

std::cout << "once" << endl;

}

inline void do\_once(bool do\_throw)

{

try { std::call\_once(flag, may\_throw\_function, do\_throw); }

catch (...) {}//表示捕获所有类型的异常

}

int main()

{

std::thread t1(do\_once, true);

std::thread t2(do\_once, true);

std::thread t3(do\_once, false);

t1.join();

t2.join();

t3.join();

}

/\*输出的结果可能是0到3行throw，和一行once\*/

# sleep函数：

以下代码均可延时（约）0.5秒，且期间不占用cpu资源。

你想睡5秒，操作系统能保证的是，5秒内不叫醒你；但操作系统一般没法保证的是，刚好5秒的时刻非得叫醒你。也就是说，你可能会睡更长的时间。要是有高优先级的任务一直在忙，则5s后仍一直没法获得cpu时间片

//C++（Windows）：

#include <windows.h>

Sleep(500);

//C++（Linux）：

#include <unistd.h>

usleep(500000);

//C++（C++11）：

#include <thread>

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

# 异步接口async()：

#include<future>

异步接口async()：

**typedef** std::chrono::steady\_clock                STEADY\_CLOCK;

**typedef** std::chrono::steady\_clock::time\_point    TIME\_POINT;

//async()方法

**template**<**int** add\_ele>

**int** calculate()//计算函数，延迟add\_ele ms

{

    std::this\_thread::sleep\_for(std::chrono::milliseconds(add\_ele));

**return** add\_ele;

}

**void** test\_future\_time()

{

**int** sum = 0;

    TIME\_POINT t11 = STEADY\_CLOCK::now();

    //异步调用，get()函数会阻塞，直到结果返回

    std::future<**int**> f1 = std::async(calculate<1>);

    std::future<**int**> f2 = std::async(calculate<2>);

    std::future<**int**> f3 = std::async(calculate<3>);

    sum = f1.get() + f2.get() + f3.get();

    TIME\_POINT t22 = STEADY\_CLOCK::now();

    cout << "test\_future\_time result is :" << sum << "  ";

    std::cout << "time span : " << std::chrono::duration<**double**, std::milli>(t22 - t11).count() << "ms" << endl;

}

std::async()

(1)方便获取线程入口函数的返回值，如std::future<> res=std::async(); res.get();

(2)有不同的线程创建策略，std::async()的第一个参数

* std::laugh::async：立即创建，在调用std::async()时立即创建一个新的线程以异步调用函数
* std::launch::deferred：延时创建，在调用std::async()时不创建线程，直到调用了future对象的get()或wait()方法时，才创建线程
* std::launch::deferred | std::launch::async：自动选择某一策略，这取决于系统和库的实现，通常是优化系统中当前并发的可用性

“std::async并不async”，如下示例：

std::async([]{ f(); }); /\*a temp std::future<void> is constructed\*/

/\* blocked by the destructor of std::future<void> \*/

std::async([]{ g(); });

我们可能期望函数g和函数f同时执行。但是如上所列代码使用std::async的方式，发起执行函数g的任务调度，一定发生在执行函数f的任务返回之后。原因是：

第一行std::async创建了一个类型为std::future<void>的临时变量Temp；

临时变量Temp在开始执行第二行之前发生析构；

std::future<void>的析构函数，会同步地等操作的返回，并阻塞当前线程。

参数传递：

向模板函数std::sync()传递参数与向thread()构造函数传递参数的方法相同。可以传全局函数、类的成员函数、lambda表达式、函数对象等

当第一个参数是一个指向成员函数的指针，第二个参数提供有这个函数成员类的

具体对象(不是直接的，就是通过指针，还可以包装在 std::ref 中)，剩余的参数可作为成员

函数的参数传入

#include<iostream>

#include<thread>

#include<future>  //async、future

**using** std::cout;

**using** std::endl;

**int** main()

{

**class** A

    {

**public**:

        A(**int** x\_) :x(x\_) {}

**void** show(**int**) { x++; cout <<"x=" << x<< endl;}

**int** get\_x() { **return** x; }

**private**:

**int** x;

    };

    A a(99);

    std::**thread** t(&A::show, a, 10);  //100， 调用temp.show(10)，temp由a拷贝构造而来

    t.join();

    cout << a.get\_x() << endl;  //99

    A a2(99);

    std::**thread** t2(&A::show, &a2, 10);  //100, 调用&a->show(10)

    t2.join();

    cout << a2.get\_x() << endl;  //100

    A a3(99);

    std::**thread** t3(&A::show, std::ref(a3), 10);  //100，调用a3.show(10)

    t3.join();

    cout << a3.get\_x() << endl; //100

    cout << "----------------------------------" << endl;

    A b(99);

    std::async(&A::show, b, 10).get(); //100

    cout << b.get\_x() << endl;  //99

    A b2(99);

    std::async(&A::show, &b2, 10).get(); //100

    cout << b2.get\_x() << endl; //100

    A b3(99);

    std::async(&A::show, std::ref(b3), 10).get(); //100

    cout << b3.get\_x() << endl; //100

}

# future：

future可移动不可拷贝，有无参构造函数

C++ 标准库将一次性事件称为期望(future)

future管理的资源是“共享状态”、“一次性事件”、“期望”、“同步状态”

std::future::valid()检查当前的future对象是否有效，即是否与某个共享状态相关联。一个有效的 std::future 对象只能通过 std::async(), std::future::get\_future()或者std::packaged\_task::get\_future()来初始化。另外由 std::future 默认构造函数创建的 std::future 对象是无效(invalid)的，当然通过std::future的move赋值后该std::future对象也可以变为valid

future 可以处理所有在线程间数据转移的必要同步

future的状态（future\_status），future\_status有三种状态：

deferred：异步操作还没开始，“要计算结果的函数仍未启动”

ready：异步操作已经完成

timeout：异步操作超时

获取future结果有三种方式：get、wait、wait\_for

std::future::get()：当与future 对象相关联的共享状态的标志变为 ready 后，调用该函数将返回保存在共享状态中的值，如果共享状态的标志不为 ready，则调用该函数会阻塞当前的调用者，而此后一旦共享状态的标志变为 ready，get 返回共享状态的值或者异常（如果抛出了异常）

std::future::wait()：等待与当前std::future 对象相关联的共享状态的标志变为 ready。如果共享状态的标志不是 ready（此时 Provider 没有在共享状态上设置值（或者异常）），调用该函数会被阻塞当前线程，直到共享状态的标志变为 ready。一旦共享状态的标志变为 ready，wait() 函数返回，当前线程被解除阻塞，但是 wait() 并不读取共享状态的值或者异常

std::future::wait\_for()：可以设置一个时间段，如wait\_for(std::chrono::seconds(2))，如果共享状态的标志在该时间段结束之前没有被 Provider 设置为 ready，则调用 wait\_for 的线程被阻塞，在等待了 rel\_time 的时间长度后 wait\_until() 返回，返回值如下：

|  |  |
| --- | --- |
| 返回值 | 描述 |
| future\_status::ready | 共享状态的标志已经变为 ready，即 Provider 在共享状态上设置了值或者异常。 |
| future\_status::timeout | 超时，即在规定的时间内共享状态的标志没有变为 ready |
| future\_status::deferred | 要计算结果的函数仍未启动 |

std::future<**const** **int**&> res = std::async(max<**int**>,4,8);

std::future\_status status;

**do**

{

    status = res.wait\_for(std::chrono::seconds(1));

**if** (status == std::future\_status::deferred)

        cout << "deferred" << endl;

**else** **if** (status == std::future\_status::timeout)

        cout << "timeout" << endl;

**else** **if** (status == std::future\_status::ready)

        cout << "ready" << endl;

} **while** (status != std::future\_status::ready);

# std::promise

std::promise可移动（有移动构造和移动赋值函数），不可拷贝。因此thread的线程入口函数的形参应该为std::promise&，向thread构造函数传实参时应该std::ref(std::promise对象)

std::promise为获取线程函数中的某个值提供便利。当“承诺”的值在线程中已经设置完毕(使用set\_value()成员函数)，则对应“期望”的状态变为“就绪”， 即使此时线程没有结束，已经可以在主线程中调用promise::get\_future().get()获取值

std::promise::get\_future()函数返回一个与 promise 共享状态相关联的 future 。调用该函数之后，返回的future对象和promise两个对象共享相同的共享状态。返回的 future 对象可以访问由 promise 对象设置在共享状态上的值或者某个异常对象。只能从 promise 共享状态获取一个 future 对象。在调用该函数之后，promise 对象通常会在某个时间点准备好(设置一个值或者一个异常对象)。

应用场景：一个线程在后台一直运行，不会退出，负责循环处理多个端口的TCP连接的数据发送与接收。每发送或接收一个TCP报文段该后台线程就需要通知相应的应用进程“成功/失败”等状态消息。使用std::promise，每处理一个报文段，则promise::set\_value(状态消息)。应用进程可以使用promise::get\_future().wait\_for()实现限定时间的等待，超时则认为失败

std::promise<string> pr1;

std::promise<string> pr2;

std::promise<string> pr3;

std::**thread** t([](std::promise<string>& p1, std::promise<string>& p2, std::promise<string>& p3)

{

    cout << "100" << endl;

    p1.set\_value("start");

    std::this\_thread::sleep\_for(std::chrono::seconds(5));

    cout << "200" << endl;

    p2.set\_value("mid");

    std::this\_thread::sleep\_for(std::chrono::seconds(5));

    cout << "300" << endl;

    p3.set\_value\_at\_thread\_exit("over");

},std::ref(pr1), std::ref(pr2), std::ref(pr3));

cout << "hello" << endl;

std::future<string> f = pr1.get\_future();

cout << "world" << endl;

string r = f.get(); //在此阻塞，直到future的状态变为就绪

cout << r << endl;

cout << pr2.get\_future().get() << endl;

cout << pr3.get\_future().get() << endl;

/\*

运行结果：

100hello world start 200 mid 300 over

\*/

# std::packaged\_task

可移动构造、移动赋值，不可拷贝构造、拷贝赋值，有无参构造函数（初始化一个空的共享状态，不包装任务）

std::packaged\_task 包装一个可调用对象，允许异步获取该可调用对象产生的结果。

std::packaged\_task对象内部包含两个最基本元素：(1)被包装的任务(stored task)，即一个可调用的对象；（2）共享状态(shared state)，用于保存任务的返回值，可以通过 std::future 对象来达到异步访问共享状态的效果。

通过 std::packged\_task::get\_future() 来获取与共享状态相关联的 std::future 对象。在调用该函数之后，两个对象共享相同的共享状态，具体解释如下：

* std::packaged\_task 对象是异步 Provider，它在某一时刻通过调用被包装的任务来设置共享状态的值。
* std::future 对象是一个异步返回对象，通过它可以获得共享状态的值，当然在必要的时候需要等待共享状态标志变为 ready.

std::packaged\_task<**int**()> task([]()

{

    std::this\_thread::sleep\_for(std::chrono::seconds(3));

**return** 7;

});

std::future<**int**> fu = task.get\_future();  //获取与packaged\_task相关联的future对象，该future对象与packaged\_task对象共享一个共享状态

std::**thread** t1(std::ref(task));  //packaged\_task不可拷贝，因此这里必须是ref()

**int** val = fu.get();//等待共享状态标志变为 ready ，即通过future实现对共享状态的异步访问

cout << val << endl;

t1.detach();  //thread对象必须有join()或detach语句

# std::future线程间传输异常

async()返回的future、与std::packaged\_task关联的future、与promise关联的future

函数作为 std::async 的一部分时，当在调用时抛出一个异常，那么这个异常就会存储到future中，之后“期望”的状态被置为“就绪”，之后调用get()会抛出这个存储的异常。

当你将函数打包入 std::packaged\_task 任务包中后，在这个任务被调用时，同样的事情也会发生；当打包函数抛出一个异常，这个异常将被存储在与std::packaged\_task关联future对象中，准备在调用get()再次抛出。

std::promise 也能提供同样的功能。当你希望存入的是一个异常而非一个数值时，你就需要调用set\_exception()成员函数，而非set\_value()。异常将被存储在与std::promise对象关联future中

（1）捕获不到异常 ： try语句std::thread(square\_root, -1)

std::**thread** t1;

**try**

{

    t1 = std::**thread**(square\_root, -1);

    cout << "world" << endl;

}

**catch** (...)

{

    cout << "exception" << endl;

}

t1.join();  //此句抛出异常

（2）能正确捕获处理异常：async().get()

**try**

{

    std::future<**double**> fu = std::async(square\_root, -1);

    cout << "hi" << endl;

    cout << fu.get() << endl; //此句抛出异常  ，被try捕获

    cout << "hello" << endl;

}

**catch** (...)

{

    cout << "exception thrown" << endl;

 }

（3）正确捕获异常，给thread()构造函数传入std::ref(packaged\_task)，然后packaged\_task::get\_future()::get()

std::packaged\_task<**int**(**double**)> task([](**double** x)

{

**if** (x < 0) **throw**"exception";

**return** **static\_cast**<**int**>(x);

});

std::future<**int**> fu = task.get\_future();

std::**thread** t1(std::ref(task), -3.4);

**try** { cout << fu.get() << endl; /此句抛出异常，被try捕获

 }

**catch** (...) { cout << "exception" << endl; }

t1.join();

/\*运行结果，exception \*/

（4）能正确捕获处理异常：

try{}

catch(…){promise::set\_exception(std::current\_exception());}

在主线程中try语句promise::get\_future()::get()

auto task = [](**double** x, std::promise<**int**>& res)

{

**if** (x < 0)

    {

**try** { **throw** "<0"; }

**catch** (...) { res.set\_exception(std::current\_exception()); }  //std::promise::set\_excption()不会影响当前线程继续运行，要想停止，则return

    }

**else**  res.set\_value(**static\_cast**<**int**>(x));

};

std::promise<**int**> res;

std::**thread** t1(task, -3.4, std::ref(res));

**try**

{

    cout << res.get\_future().get() << endl;

    cout << "over" << endl;

}

**catch** (...) { cout << "exception" << endl; }

t1.join();

/\*运行结果：exception\*/

# promise与packaged\_task用于单线程中的异常，延后异常抛出时间

（1）可以将一个异常存储在promise::get\_future()中

auto task = [](**double** x, std::promise<**int**>& res)

{

**if** (x < 0)

    {

**try** { **throw** "<0"; }

**catch** (...) { res.set\_exception(std::current\_exception()); }

    }

**else** res.set\_value(**static\_cast**<**int**>(x));

};

std::promise<**int**> prom;

task(-3.4, std::ref(prom));

**try**

{

    cout << prom.get\_future().get() << endl;

    cout << "over" << endl;

}

**catch** (...) { cout << "exception" << endl; }

/\*运行结果：exception\*/

（2）packaged\_task<>对象，调用时抛出异常，异常会存储在packaged\_task::get\_future中

std::packaged\_task<**int**(**double**)> task([](**double** x)

{

**if** (x < 0) **throw**"exception";

**return** **static\_cast**<**int**>(x);

});

std::future<**int**> fut = task.get\_future();

task(-3.4);

**try** {

    cout << fut.get() << endl; // 此句抛出异常，被try捕获

}

**catch** (...) { cout << "exception" << endl; }

/\*运行结果，exception \*/

# promise的析构函数与异常

~promise();放弃共享状态。

- 如果共享状态是ready的，释放（release）它。

- 如果共享状态是没有ready的，存储一个异常（std::future\_error类型，带错误码std::future\_errc::broken\_promise），使共享状态为ready，然后再release它。

与promise关联的future，在调用promise::set\_value()或promise::set\_exception()前，promise对象析构，则向共享状态存储一个std::future\_error类型的异常，设置共享状态的标志为ready，释放状态。

std::promise<**int**>\* prom = **new** std::promise<**int**>;

std::future<**int**> fut = prom->get\_future();

std::**thread** t1([](std::promise<**int**>& pr)

{

**delete** ≺

},std::ref(\*prom));

**try**

{

    fut.get();  //抛出异常

    cout << "hello" << endl;

}

**catch** (std::future\_error& e) //待捕获的异常类型std::future\_error

{

    cout << "exception" << endl;

    cout << e.code() << endl; //future:1

    cout << e.what() << endl; //broken promise

}

t1.join();

# std::shared\_future

std::future有局限性，在很多线程在等待的时候，只有一个线程能获取等待结果。当多个线程需要等待相同的事件的结果，就需要使用std::shared\_future

std::shared\_future 与 std::future 类似，但是 std::shared\_future 可以拷贝、多个 std::shared\_future 可以共享某个共享状态的最终结果(即共享状态的某个值或者异常)。shared\_future 可以通过某个 std::future 对象隐式转换（参见 std::shared\_future 的构造函数），或者通过 std::future::share() 显示转换，无论哪种转换，被转换的那个 std::future 对象都会变为 not-valid

# std::atomic\_flag：

有以下语句，因此没有拷贝构造、移动构造、拷贝赋值、移动赋值，同std::mutex

atomic\_flag() \_NOEXCEPT = default;

atomic\_flag(const atomic\_flag&) = delete;

atomic\_flag& operator=(const atomic\_flag&) = delete;

atomic\_flag仅有两种状态clear、set

bool test\_and\_set()函数检查std::atomic\_flag标志，如果std::atomic\_flag之前没有被设置过，则设置std::atomic\_flag的标志，并返回false，如果之前已被设置，则返回true。

test-and-set操作是原子的（因此test-and-set是原子read-modify-write（RMW）操作）

std::atomic\_flag::clear()清除std::atomic\_flag标志，使得下一次调用test\_and\_set()时返回false

自旋锁是一种用于保护多线程共享资源的锁，与一般的互斥锁(mutex)不同之处在于当自旋锁尝试获取锁的所有权时会以忙等待(busywaiting)的形式不断的循环检查锁是否可用

# std::atomic

std::atomic不等价于原子操作。因为std::atomic<T>除了保证读\写操作是原子操作外，还有另一个重要的作用是引入内存序限制。如赋值函数T operator =(T) noexcept即是采用了std::memory\_order\_seq\_cst的原子写操作，转换函数operator T()即是采用了std::memory\_order\_seq\_cst的原子读操作。

std::atomic类模板除了基本类型以外，还针对整形和指针类型做了特化

* std::atomic无拷贝无移动，有无参构造atomic() = default和constexpr atomic(T);

std::atomic<bool> ready(false);

ready = true;//主线程， 调用T std::atomic<T>::operator= (T val) noexcept

while (!ready) //调用stomic<T>::operator T() const成员函数

{

std::this\_thread::yield(); //yield作用是当前线程“放弃”执行，交出cpu时间片，让操作系统调度另一线程继续执行

} // 等待主线程中设置 ready 为 true

* T类型赋值函数，T operator =(T) noexcept，内存序(Memory Order) 默认为顺序一致性(std::memory\_order\_seq\_cst)，如果需要指定其他的内存序，需使用

void std::atomic::store (T val, memory\_order sync = memory\_order\_seq\_cst) noexcept，参数sync设置内存序(Memory Order)

* T类型转换函数，std::atomic<T>::operator T()，内存序(Memory Order) 默认为顺序一致性(std::memory\_order\_seq\_cst)，如果需要指定其他的内存序，需使用T load (memory\_order sync = memory\_order\_seq\_cst) const noexcept

std::atomic<int> bar = 5; //调用构造函数constexpr atomic(int)

**bool std::atomic<>::is\_lock\_free() const**成员函数，让用户可以查询某std::atomic的原子操作是直接用的原子指令(x.is\_lock\_free()返回true)，还是编译器和库内部用了一个锁(x.is\_lock\_free()返回false)。如果std::atomic<T>是lock-free的，则在多个线程访问该对象时不会导致线程阻塞。

**T std::atomic<>:: exchange(T val, memory\_order sync = memory\_order\_seq\_cst) noexcept**成员函数，读取并修改被封装的值，exchange会用val指定的值替换掉之前该原子对象封装的值，并返回之前该原子对象封装的值，整个过程是原子的(因此exchange操作也称为read-modify-write操作)。sync参数指定内存序(Memory Order)

**std::atomic<>::compare\_exchange\_strong：**

bool compare\_exchange\_strong (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) volatile noexcept;

bool compare\_exchange\_strong (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) noexcept;

比较std::atomic封装的值与参数expected所指定的值是否相等，如果：

* 相等，则用val替换原子对象的旧值，返回true
* 不相等，则用原子对象的旧值替换expected，原子对象旧值不变。因此调用该函数时，如果被该原子对象封装的值与参数expected所指定的值不相等，则调用后expected中的内容就是原子对象的旧值，返回false

上述整个操作是原子的

compare\_exchange\_strong直接比较原子对象所封装的值与参数expected的物理内容，所以某些情况下，对象的比较操作在使用operator==()判断时相等，但compare\_exchange\_strong判断时却可能失败，因为对象底层的物理内容中可能存在位对齐或其他逻辑表示相同但是物理表示不同的值(比如true和2或3，它们在逻辑上都表示"真"，但在物理上两者的表示并不相同)

**std::atomic<>::compare\_exchange\_weak：**

bool compare\_exchange\_weak (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) volatile noexcept;

bool compare\_exchange\_weak (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) noexcept;

std::atomic<>::compare\_exchange\_weak与std::atomic<>::compare\_exchange\_strong的唯一区别是可能会伪失败：即原子对象所封装的值与参数expected的物理内容相同，但compare\_exchange\_weak却仍然返回false。伪失败时参数expected的值不会改变。不过在某些需要循环操作的算法下伪失败是可以接受的。并且在一些平台下compare\_exchange\_weak的性能更好。

**struct** Node

{

**int** value;   Node\* next;

};

std::atomic<Node\*> list\_head(nullptr);

**void** append(**int** val)

{

    Node\* newNode = **new** Node{val,list\_head};

    //next line is the same as: list\_head = newNode, but in a thread-safe way:

**while** (!list\_head.compare\_exchange\_weak(newNode->next, newNode)) {}  //即使误判true为false，也不影响

}

**int** main()

{

    std::vector<std::**thread**> threads;

**for** (**int** i = 0; i < 5; ++i)   threads.emplace\_back(append,i);

**for** (auto& i : threads) i.join();

**for** (Node\* it = list\_head; it != nullptr; it = it->next)   cout << it->value << " ";   cout << endl;

    Node\* it = list\_head;

**while** (it != nullptr)

    {

        Node\* temp = it;

        it = it->next;

**delete** temp;

    }

    //可能的执行结果4 3 2 1 0

}

**std::atomic<>::compare\_exchange\_strong：**

bool compare\_exchange\_strong (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) volatile noexcept;

bool compare\_exchange\_strong (T& expected, T val,

memory\_order sync = memory\_order\_seq\_cst) noexcept;

std::atomic<>::compare\_exchange\_strong与std::atomic<>::compare\_exchange\_weak的唯一区别是不会伪失败，即原子对象所封装的值与参数expected的物理内容相同，比较操作一定会为true

# std::atomic<UDT>

不是任何自定义类型都可以使用std::atomic<>的，需要满足一定的标准才行。

为了使用std::atomic<UDT>(UDT是用户定义类型)，这个类型必须满足：

* 必须有无参构造函数，Udt(){}或者Udt()=default
* 拷贝构造和拷贝赋值，要么default，要么编译器自动生成，即必须是trivial；如果不满足自动生成条件，则出错
* 移动构造和移动赋值，要么default，要么编译器自动生成，即必须是trivial；如果不满足自动生成条件，则出错
* 析构函数要么default，要么编译器自动生成，即析构函数必须是plain；
* 不包含虚函数和虚基类
* 直接基类的拷贝构造、拷贝赋值、移动构造、移动赋值是trivial
* 非静态类成员变量的拷贝构造、拷贝赋值、移动构造、移动赋值是trivial

**class** Udt

{

**public**:

    Udt() {}

    Udt(**int** age\_):age(age\_) {}

    Udt(**const** Udt& rhs) = **default**;

    Udt& operator=(**const** Udt& rhs) = **default**;//= default;

    Udt(Udt&& rhs)  = **default**;

    Udt& operator=(Udt&& rhs)  = **default**;

**int** get\_age()**const** { **return** age; }

    ~Udt() = **default**;

**private**:

**int** age;

};

**int** main()

{

    std::atomic<Udt> x= Udt(3);

    cout << x.load().get\_age() << endl; //如果Udt定义时没有Udt(){}，则该句报错：“没有合适的默认构造可用”

    x.exchange(Udt(4));

    cout << x.load().get\_age() << endl; //4

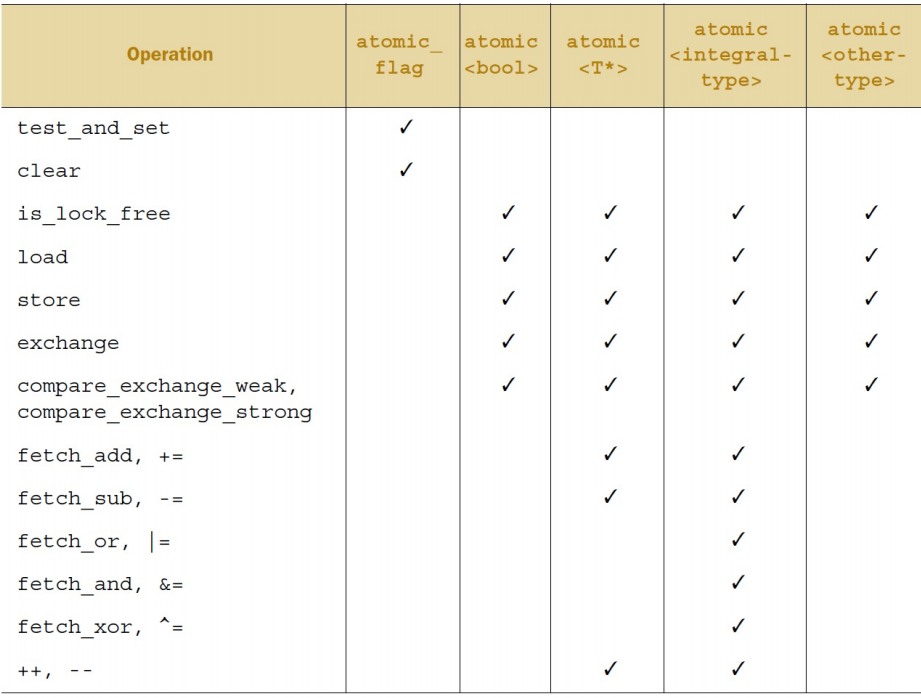
    Udt expect(4);

    x.compare\_exchange\_strong(expect,Udt(5));

    cout << x.load().get\_age() << endl; //5

}

# atomic<>不同特化类型能使用的操作



# std::atomic:: is\_lock\_free()和std::atomic\_flag：

std::atomic<int> age;

std::atomic<bool> bol;

int main()

{

cout << age.load() << endl; //0

cout << bol.load() << endl; //0

cout << std::atomic<char>().is\_lock\_free() << endl; //1

cout << std::atomic<bool>().is\_lock\_free() << endl; //1

cout << std::atomic<short>().is\_lock\_free() << endl; //1

cout << std::atomic<int>().is\_lock\_free() << endl; //1

cout << std::atomic<float>().is\_lock\_free() << endl; //1

cout << std::atomic<double>().is\_lock\_free() << endl; //1

}

std::atomic\_flag的test\_and\_set()具有memory\_order\_acquire语义，clear()具有release语义

# C++内存模型：

内存模型分为静态内存模型、动态内存模型：

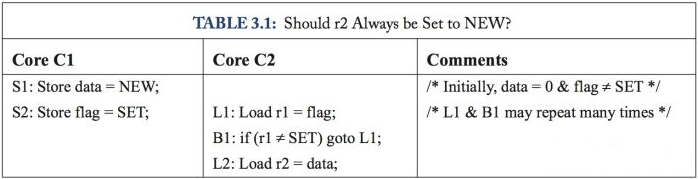
* 静态内存模型主要涉及类的对象在内存中的布局。
* 动态内存模型可理解为存储一致性模型，涉及了内存、Cache、CPU 各个层次的交互。

# 指令重排：

指令重排序(Memory Reordering)：

* 编译器对指令重排序。不同的优化级别下编译器对指令进行了不同程度的重排
* CPU对指令重排序。现代的CPU支持乱序执行，指令被CPU执行的顺序和汇编指令的顺序不一致。

示例：



假设在目标平台上对data、flag的读写都是原子操作。core C1与 core C2是两个单独的core，各自执行自己的程序，r2会一直被置为NEW吗？答案当然不是。对于core来说，他并不知道自己当前执行的是一个多线程程序。为了提高性能，CPU优化，那么对于C1而言，CPU可以执行S2->S1，也可以执行S1->S2, 对于这两种执行方式在C1看来是没有问题的，因为单线程而言这两种执行方式最后达到的效果是一样的。那么可能C1的执行顺序是S2->L1->L2->S1，得到的结果r2 = 0

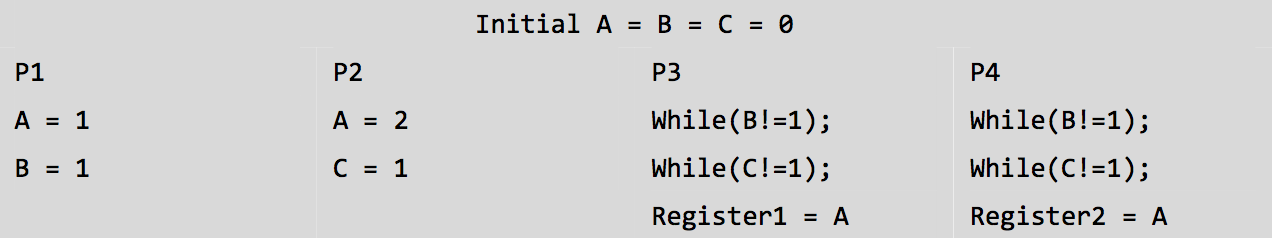
因此，多线程环境下需要对必须对编译器和CPU作出一定的约束，才能合理优化程序而不出错，这个约束即可由内存序提供。

如何修改使上述程序能确保r2被置为NEW？答案是除了data、flag的读写都是原子操作外，flag=SET使用memory\_order\_release内存序，flag读使用memory\_order\_acquire内存序。这样S1 sequenced-before S2 、S2 synchronizes-with L1+B1、L1+B1 sequenced-before L2，则可以推导出S1 happens-before L2，因此r2最后一定是NEW

# 缓存一致性cache coherence

同一CPU内的多个核心共享L3缓存，但每个核心都有自己的L1、L2缓存

示例：



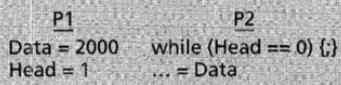
不满足缓存一致性时：程序结束后，P3和P4的输出结果（寄存器1、2）可能不同。这种情况可能发生在基于网络（而不是总线）的系统中，由于消息可经不同路径传递，系统不提供它们传递次序的保证。

满足缓存一致性时：由于缓存一致性的性质之一即对同一地址的写操作的串行化，因此Register1和Register2一定相同。

缓存一致性的定义，即满足如下三个条件则cache [coherent](https://www.zhihu.com/search?q=coherent&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A83422523%7D)：

1. 单处理器中的程序次序(program order)。对于同一地址X，若处理器P先写后读，并且在P写后没有别的处理器写X，那么P的读总是能返回P写入的值
2. 对同一地址的写操作的串行化(write serialization)。在所有处理器看来，通过不同处理器对X的写操作都是以相同次序进行的。举例，如值A、B被写入同一个地址，不存在“一个处理器先看到值A再看到值B而另一个处理器先看到值B再看到值A”的情况。如果是一个处理器依次写入A、B，那么所有处理器看到的都是先A再B。如果两个处理器分别写入A、B，那么所有处理器看到的写入顺序要么都是先A再B，要么都是先B再A。
3. 写操作最终对读操作可见。处理器P1完成对X的写后，P2读X，如果两者之间相隔时间足够长，并且没有其他处理器写X，那么P2可以获得P1写入的值。缓存一致性虽然保证写操作最终对读操作可见，但具体何时可见未描述。如考虑P1先写某个地址然后P2读该地址，如果写读操作间隔很短，P2可能无法获得P1写入的值

示例：



在一个缓存一致性系统中执行上面的程序，那么P2读到的Data一定是2000吗？

不一定。如以下过程未违背缓存一致性，但P2读到的是旧值

1. P1先完成了Data在内存上的写操作；
2. P1没有等待Data的写结果传播到P2的缓存中,继续进行Head的写操作；
3. P2读取到了内存中Head的新值；
4. P2继续执行，读到了缓存中Data的旧值

# cpu、编译器与内存序：

不同的内存序模型在不同的CPU架构下会有不同的代价。这允许专家通过采用更合理的内存序获得更大的性能升；同时允许在对性能要求不是那么严格的程序中采用默认的内存序，使得程序更容易理解。

对于x86 cpu而言，在cpu层级本身就支持release/acquire语义，因此在x86 cpu下release与acquire/consume都只影响编译器的优化，而memory\_order\_seq\_cst还会影响cpu的指令重排。x86 cpu做不到memory\_order\_relaxed内存模型。

与特定硬件优化，如重叠写(overlapping writes)、缓存(cache)等技术相结合时，要保证memory\_order\_seq\_cst是有代价的。相关论文中给出了为保证memory\_order\_seq\_cst，一个写操作的耗时从50 cycles变为170 cycles的例子。同时，memory\_order\_seq\_cst的要求也会限制编译器优化。

# 内存序与std::atomic

C++11中有6个内存序（Memory Order），分为三种：

* 顺序一致性内存序(memory\_order\_seq\_cst)
* 获取-释放内存序(memory\_order\_acquire、memory\_order\_release、memory\_order\_acq\_rel、memory\_order\_consume)
* 自由内存序(memory\_order\_relaxed)

memory order限制编译器和CPU对单线程中的指令的重排，以及对cache的控制方法。

memory order本身与多线程无关，但是单线程中合理的指令重排，在多线程环境下却可能导致程序逻辑错误。因此多线程编程时必须考虑内存序，对单线程中的指令重排加以限制，从而实现多线程间的同步，保证程序逻辑正确

std::atomic不等价于原子操作。因为std::atomic<T>除了保证读\写操作是原子操作外，还有另一个重要的作用是引入内存序限制。如赋值函数T operator =(T) noexcept即是采用了std::memory\_order\_seq\_cst的原子写操作，转换函数operator T()即是采用了std::memory\_order\_seq\_cst的原子读操作。

# memory\_order\_relaxed：

memory\_order\_relaxed在不影响单线程逻辑正确性的前提下，最大限度地进行指令重排

# memory\_order\_seq\_cst：

顺序一致性内存序（memory\_order\_seq\_cst）在C++11的6个内存序中最为简单、直观，但不利于程序的优化。

memory\_order\_seq\_cst阻止这个调用之前的读写操作被重排到后面，也阻止这个调用之后的读写操作被重排到前面

示例：

atomic<**bool**> f1=**false**;

atomic<**bool**> f2=**false**;

​

// thread1

f1.store(**true**, memory\_order\_release);

**if** (!f2.load(memory\_order\_acquire)) {

  critical\_section();

}

​

// thread2

f2.store(**true**, memory\_order\_release);

**if** (!f1.load(memory\_order\_acquire)) {

  critical\_section();

}

上述的两个线程可以同时进入临界区吗？答案是可以的。因为f1、f2是两个memory location，所以可以reorder。同时根据release和acquire对reorder的约束，f1，f2的两个store操作可以下移到下面，这是符合release和acquire语义的，这时候两个线程就可以同时进入临界区了。这个例子中f1、f2的读写如果使用memory\_order\_seq\_cst内存序，则可防止同时进入临界区

示例：

假设存在两个共享变量atomic<int> a, b，封装的初始值均为0，R1、R2为int类型，两个线程运行不同的指令，如下表格所示。a、b的读写采用顺序一致性内存序。

|  |  |
| --- | --- |
| 线程 1 | 线程 2 |
| a = 1; | b = 2; |
| R1 = b; | R2 = a; |

则指令执行顺序共6种，R1和R2 最终结果只有3种情况

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 情况1 | 情况2 | 情况3 | 情况4 | 情况5 | 情况6 |
| a = 1; | b = 2; | a = 1; | a = 1; | b = 2; | b = 2; |
| R1 = b; | R2 = a; | b = 2; | b = 2; | a = 1; | a = 1; |
| b = 2; | a = 1; | R1 = b; | R2 = a; | R1 = b; | R2 = b; |
| R2 = a; | R1 = b; | R2 = a; | R1 = b; | R2 = a; | R1 = b; |
| R1 == 0, R2 == 1 | R1 == 2, R2 == 0 | R1 == 2, R2 == 1 | R1 == 2, R2 == 1 | R1 == 2, R2 == 1 | R1 == 2, R2 == 1 |

示例：



上述程序中，原子变量data\_ready采用memory\_order\_seq\_cst内存序，再根据内存模型关系中的happens-before和synchronized-with关保证了执行顺序是3->4->1->2

# acquire release内存序：

store(memory\_order\_release)阻止这个调用之前的读写操作被重排到后面

load(memory\_order\_acquire)阻止这个调用之后的读写操作被重排到前面

consume的限制比acquire少的地方在于允许和consum变量无关的操作重排序到consume前

通常用法是acquice死循环反复读，多次读之后来确认release已经写入了，通过这样做来达到线程同步

示例：

std::atomic<singleton\*> singleton::p = nullptr;

singleton\* singleton::instance() {

if (p.load(std::memory\_order\_acquire)==nullptr) {

lock\_guard<mutex> guard(lock\_);

if (p.load(std::memory\_order\_acquire) == nullptr)

p.store(new singleton(), std::memory\_order\_release);

}

return p;

}

# 内存序与线程间同步：

happens-before关系：A指令 happens-before B指令，那么B指令执行时将看到A指令对内存的影响

sequenced-before：单线程中语句的顺序

carries-a-dependency-to：“携带依赖”，应用于单个线程，如果A操作结果要使用操作B的操作数， 则A携带依赖于B。

携带依赖是一种更严格的sequenced-before

//单线程中

int x = 3; //A

int y = 5; //B

// A sequenced-before B，但并不 B carries-a-dependency-to A

//单线程中

int x = 3; //A

int y = x+3; //B

// A sequenced-before B，且 B carries-a-dependency-to A

sychronizes-with：一个线程中用释放语义存储store(memory\_order\_release)，另一个线程中用获取语义加载load(memory\_order\_acquire)

dependency-ordered-before：“前序依赖”，一个线程中用释放语义存储，另一个线程中用memory\_order\_consume语义加载

推导规则：

* A sequenced-before B，B sychronizes-with C，C sequnced-before D，则A先行于D
* A sequenced-before B，B dependency-ordered-before C，D carries-a-dependency-to C，则A先行于D
* A sequnced-before B，B dependency-ordered-before C，C sequnced-before D（普通的sequnced-before，非携带依赖），不能推出A先行于D

示例：

std::mutex::lock()具有memory\_order\_acquire语义，unlock()具有memory\_order\_release语义。

示例：

std::atomic<std::string\*> ptr;

**int** data;

**void** producer()

{

    std::string\* p = **new** std::string("Hello");          // A

    data = 42;                          // B

    ptr.store(p, std::memory\_order\_release);            // C

}

**void** consumer()

{

    std::string\* p2;

**while** (!(p2 = ptr.load(std::memory\_order\_acquire)))  ;        // D

    assert(\*p2 == "Hello");                 // E

    assert(data == 42);                     // F

}

**int** main()

{

    std::**thread** t1(producer);

    std::**thread** t2(consumer);

    t1.join(); t2.join();

}

可以直观地得出如下关系：A sequenced-before B 、B sequenced-before C、C synchronizes-with D、D sequenced-before E、E sequenced-before F。因此，推导出A happens-before E、B happens-before F，因此，这里的E、F两处的assert永远不会fail

示例：

**void** producer()

{

    std::string\* p = **new** std::string("Hello");          // A

    data = 42;                          // B

    ptr.store(p, std::memory\_order\_release);            // C

}

**void** consumer()

{

    std::string\* p2;

**while** (!(p2 = ptr.load(std::memory\_order\_consume)))  ;      // D

    assert(\*p2 == "Hello");                 // E

    assert(data == 42);                     // F

}

可以直观地得出如下关系：A sequenced-before B、 B sequenced-before C、C dependency-ordered before D、D carries-a-dependency-to E 、E sequenced-before F。所以可以推导出A happens-before E，但B 并非happens-before F

# 内存屏障

写内存屏障std::atomic\_thread\_fence(std::memory\_order\_release)保证所有在屏障之前的写入操作都会在屏障之后的写入操作之前完成;

而读内存屏障std::atomic\_thread\_fence(std::memory\_order\_acquire)确保所有屏障之前的读取操作都会在屏障之后的读取操作前执行。内存屏障使得特定的操作无法穿越

#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); // 1

    std::atomic\_thread\_fence(std::memory\_order\_release); // 2

    y.store(**true**, std::memory\_order\_relaxed); // 3

}

**void** read\_y\_then\_x()

{

**while** (!y.load(std::memory\_order\_relaxed)); // 4

    std::atomic\_thread\_fence(std::memory\_order\_acquire); // 5

**if** (x.load(std::memory\_order\_relaxed)) // 6

        ++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); // 断言一定成功

}

由于写内存屏障，语句1、2、3不能被重排序 ，由于读内存屏障，语句4、5、6不能被重排序，同时**while** (!y.load(std::memory\_order\_relaxed)) 语句保证y的操作④读取的是在③处存储的值。

# volatile关键字

C和C++中的valatile关键字的作用于jave中不完全相同

问题:const和volatile是否可以同时修饰一个变量？有什么含义？

答案:可以。如果一个变量不会被本程序改变，通常可能给它加上const，但如果该变量可能被其他程序改变而本程序又在检测这个变量的值，就需要给它加上volatile，于是变量就同时有volatile和const了，这个时候i具有const和volatile的双重属性。i变量不可以在编译过程中被程序代码修改，同时编译器不得对i进行优化编译

 const和volatile是可以同时修饰一个变量的。const只是表示变量只读，防止当前程序“意外”修改。如果是const定义的常量，如const int num=10；且后文没有对num取地址等操作，则编译器会做优化，用字面值10代替num出现的位置。即使后文有&num，编译器也会有优化，如读num时cpu不会每次去内存取值。这个时候如果外部事件如中断服务程序改了这个变量的内存值，那么由于编译器优化就不会出有反应。加上volatile就告诉编译器每次都去内存取值

volatile：[ˈvɒlətaɪl]

软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化，以免出错。

比如要往某一地址送两指令：

int \*ip =...; //设备地址

\*ip = 1; //第一个指令

\*ip = 2; //第二个指令

以上程序compiler可能做优化而成：

int \*ip = ...;

\*ip = 2;

结果第一个指令丢失。如果用volatile, compiler就不允许做任何的优化，从而保证程序的原意：

volatile int \*ip = ...;

\*ip = 1;

\*ip = 2;

即使你要compiler做优化，它也不会把两次付值语句间化为一。它只能做其它的优化。

在 C/C++ 中，对 volatile 对象的访问，有**编译器**优化上的副作用：

* 保证内存可见性，读时从内存读，不使用之前寄存器缓存的结果；写时写到内存。
* 对volatile对象的两个访问不能被编译器重排序。如对x、y 2个volatile的变量的写，对x、y 2个volatile的变量的写x后读y

x是volatile、y不是volatile，则对x、y的读写可以被编译器乱序，但前提是不影响单线程执行的正确语意

**volatile禁止了编译器的重排序，但CPU仍可能重排序**

value=10; //①

flag = true; //②

value和flag都不是volatile时，②语句可能被编译器重排序到①语句前;

value和flag中仅一个是volatile，②语句仍可能被编译器重排序到①语句前;

value和flag中都是volatile，则编译器不会重排序，但CPU仍可能乱序（先执行②语句再①）

当前函数可能需要2次访问一个int全局变量 但是这个全局变量的2次读之间：

1. 没有调用任何非inline函数；
2. 没有任何内嵌汇编语句；
3. 可能被另外一个线程改写这个变量的值。

那么很有可能编译器在第二次访问这个变量的时候，直接使用了第一次访问的值。这就可能造成3)另一线程对该变量的修改体现不出来。

平常多线程编程的时候，加一下锁来访问这个变量，怎么就没有这个问题？因为加锁调用了非inline函数。绝大多数系统api的加锁操作都是函数而不是纯宏（宏里不调函数）。C++的编译链接模型，决定了调用一个函数之后，编译器就会认为：

1. 所有非局部变量都需要重新从内存中取值，
2. 被取过地址或者引用的局部变量往往也需要重新被求值。

或者把这个全局变量定义成volatile int，都可以让编译器读时从内存读，不使用寄存器缓存的值，则可读到另一线程修改后的值

# Windows的消息机制：

Windwos中记录消息的结构体为MSG

**struct** MSG

{

**HWND**    hwnd;        //接受该消息的窗口句柄

**UINT**    message;      //消息常量标识符，消息号

**WPARAM**  wParam;  //32位消息的特定附加信息，确切含义依赖于消息值

**LPARAM**  lParam;       //32位消息的特定附加信息，确切含义依赖于消息值

**DWORD**   time;        //消息创建时的时间

    POINT   pt;         //消息创建时的鼠标/光标在屏幕坐标系中的位置

};

其中，MSG::wParam和MSG::lParam携带消息的附加信息。例如，当收到一个键盘按下消息的时候，message成员的值就是WM\_KEYDOWN，但用户按下的到底是哪一个按键由wParam成员告知

//  窗口过程函数

//  msg为MSG::message，消息号

// wParam为MSG::wParam

// IParam为MSG::IParam

**LRESULT** CALLBACK WndProc(**HWND** hWnd, **UINT** message, **WPARAM** wParam, **LPARAM** lParam)  {  //如果按下ESC键，则窗口退出

**if**(message== WM\_KEYDOWN && wParam == VK\_ESCAPE) { PostQuitMessage(0); }

}

Windwos的消息队列：

* 系统消息队列：这个是一个操作系统唯一的队列，设备驱动将输入转化成消息（MSG结构体）送入系统消息队列
* 线程消息队列：每一个GUI线程都会维护4种线程消息队列（线程消息队列只有在线程第一次调用GUI相关函数时才会创建，默认不创建）

GUI即Graphical User Interface，图形用户界面、图形用户接口

线程第一次调用GUI相关函数时，系统为该线程分配一个THREADINFO结构用于管理线程的4种消息队列，此外该结构体还含有nExitCode、唤醒标志（wake flag）等成员

线程的4种消息队列：

* 发送消息队列（send-message queue）
* 登记消息队列（posted-message queue）
* 虚拟输入队列（virtualized-input queue）
* 应答消息队列（reply-message queue）



Windows系统的消息分为队列消息和非队列消息：

* 队列消息：消息会被送入发送消息队列、登记消息队列等线程消息队列中，消息循环从队列中取出该消息并派发到各窗口处理。如鼠标、键盘消息。

// GetMessage获取指定窗口的消息

// 获取到的消息要么为主调线程消息队列中的消息，要么为该函数根据THREADINFO结构体中的唤醒标志生成的消息

MSG msg;

**while** (GetMessage(&msg, NULL, 0, 0)) {

    TranslateMessage(&msg); //翻译消息，即将虚拟键消息转换为字符消息

    DispatchMessage(&msg);  //派发消息，即发送消息至窗口过程WndProc

}

键盘某个键被按下，对应的事件处理流程为：

1. 键盘驱动程序将事件转换成相应的消息（即MSG结构体），送入系统消息队列；
2. 系统在适当时机，从系统消息队列中取出一个消息，送往创建MSG::hwnd窗口的线程的虚拟输入队列；
3. GetMessage从线程的虚拟输入消息队列取出该WM\_KEYDOWN或WM\_SYSKEYDOWN消息；
4. TranslateMessage检查虚拟键（virtualkey）信息是否能转换成等价的字符。如果虚键信息能够转换，则TranslateMessage调用PostMessage将一个WM\_CHAR或WM\_SYSCHAR消息放置在登记消息队列中。
5. 之后调用GetMessage时，从登记消息队列中取出该WM\_CHAR或WM\_SYSCHAR消息并将其返回。
6. DispatchMessage将消息派发给MSG::hwnd对应的窗口过程处理

* 非队列消息：非队列消息不被送入系统消息队列和线程消息队列，而是直接发送到窗口过程被处理。非队列消息可以由系统系统产生，如当用户激活一个窗口，系统发送WM\_ACTIVATE、WM\_SETFOCUS,、WM\_SETCURSOR消息。也可以由应用程序调用系统函数产生，如应用调用SetWindowPos函数产生WM\_WINDOWPOSCHANGED消息、应用调用UpdateWindow()函数产生的WM\_PAINT消息。

**int** WINAPI WinMain()

{

    WNDCLASSEX wcex = {}; //创建窗口类

    RegisterClassEx(&wcex);//注册窗口类

**HWND** hWnd = CreateWindow(wcex.lpszClassName, ……);//创建窗口

    ShowWindow(hWnd, ……);

    // UpdateWindow函数用于更新窗口指定的区域

    // 如果窗口更新的区域不为空，该函数绕过系统消息队列和线程消息队列，直接调用窗口过程函数处理WM\_PAINT消息；如果更新区域为空，则不发送消息。

    UpdateWindow(hWnd);

    MSG msg;

**while** (GetMessage(&msg, NULL, 0, 0)) {

        TranslateMessage(&msg); //翻译消息

        DispatchMessage(&msg);  //派发消息

    }

}

**发送消息**

发送消息的函数：PostMessage、PostThreadMessage、SendMessage

**LRESULT** SendMessage(**HWND** hWnd,**UINT** Msg,**WPARAM** wParam,**LPARAM** lParam);

**BOOL** PostMessage(**HWND** hWnd,**UINT** Msg,**WPARAM** wParam,**LPARAM** lParam)

**BOOL** PostThreadMessage(**DWORD** idThread,**UINT** Msg,**WPARAM** wParam,**LPARAM** lParam);

* postMessage：投递消息至创建指定窗口的线程的登记消息队列，之后立即返回。 PostMessage到别的线程的时候最好使用PostThreadMessage代替。
* PostThreadMessag：把消息放到指定线程的登记消息队列中后立即返回。
* SendMessage：

1. 如果调用SendMessage的线程向该线程所建立的一个窗口发送一个消息，则是一个非队列消息，SendMessage()函数中直接调用指定窗口的窗口过程，当窗口过程完成对消息的处理后，SendMessage函数返回，返回值即窗口过程的返回值
2. 如果SendMessage是向其它线程（即使是同进程的另一个线程）创建的窗口发消息，则消息被追加到接受线程的发送消息队列。同时接收线程设定QS\_SENDMESSAGE标志。接收线程可能当前正在执行其它代码没有等待消息（即没有调用GetMessage、PeekMessage或WaitMessage），因此发送的消息不会被立即处理。接收进程继续执行，碰到GetMessage函数，首先检查QS\_SENDMESSAGE唤醒标志是否被设定，如果是，从发送消息队列中取出队首消息，并直接调用窗口过程处理之（发送消息队列中可能有多个消息，例如，几个线程可以同时向一个窗口分别发送消息）。如果在发送消息队列中再没有消息了，则QS\_SENDMESSAGE唤醒标志被关闭。

当接收线程处理该消息的时候，调用SendMessage的线程被设置成空闲状态（idle）。在接收线程的窗口过程完成消息处理后，窗口过程的返回值被置于发送线程的应答消息队列中。发送线程由idle状态被唤醒，取出包含在应答消息队列中的返回值，作为调用SendMessage的返回值。然后，发送线程继续正常执行。

例如，共享内存中存储窗口句柄，则一个进程可以SendMessage给另一个进程的窗口。

**接收消息：**

**BOOL** PeekMessage(LPMSG lpMsg, **HWND** hWnd, **UINT** wMsgFilterMin, **UINT** wMsgFilterMax, **UINT** wRemoveMsg);

**BOOL** WaitMessage();

**BOOL** GetMessage(LPMSG lpMsg, **HWND** hWnd, **UINT** wMsgFilterMin, **UINT** wMsgFilterMax);

* PeekMessage只是从消息队列中查询消息，如果有消息，则立刻返回true；没有则返回false。如果有消息，PeekMessage中wRemoveMsg参数中设置的是PM\_REMOVE，则在取出消息并将消息从队列中删除，若设置是PM\_NOREMOVE消息就不会从消息队列中取出。
* waitMessage 功能：当一个线程的消息队列中没有消息存在时，waitMessage函数会使该线程中断并处于等待状态，同时把控制权交给其它线程，直到被中断那个线程的消息队列中有了新的消息为止。
* GetMessage获取指定窗口的wMsgFilterMin和wMsgFilterMax参数给出的消息范围内的消息。

GetMessage获取不到消息，则线程被挂起，直到获取到消息并返回。

如果hWnd为NULL，则GetMessage获取属于调用该函数应用程序的任一窗口的消息，如果 wMsgFilterMin和wMsgFilterMax都是０，则GetMessage就返回所有可得到的消息

GetMessage获取到的消息要么为主调线程消息队列中的消息，要么为该函数根据THREADINFO结构体中的唤醒标志生成的消息。GetMessage伪算法如下：

**BOOL** GetMessage(MSG \*lpMsg, **HWND** hWnd, **UINT** wMsgFilterMin, **UINT** wMsgFilterMax)

{

    //查看QS\_SENDMESSAGE标志，如果有的话循环处理，直到没有消息位置

**DWORD** dwRetVal = 0;

    ThreadInfo threadInfo;

FLAG\_SENDPROCLOOP:

    GetThreadInfo(GetCurrentThreadId(), &threadInfo);

**while** (threadInfo.QS\_SENDMESSAGE == QS\_SIGNALSET) {

        //从发送消息队列中获取消息

        dwReturnVal = GetMsgFromQueue(QUEUE\_SEND, lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax);

        //判断是否取到消息,有则调用窗口函数，无则复位QS\_SENDMESSAGE标志

**if**(dwReturnVal == GETMESSAGE\_HASMESSAGE) {

            CallWindowProc(hWnd, &threadInfo, lpMsg);   //调用指定窗口的窗口函数

        }

**else** {

            QS\_SENDMESSAGE = QS\_SIGNALRESET;

**break**;

        }

    }

    //在继续处理之前再次检查发送消息队列

**if** (threadInfo.QS\_SENDMESSAGE == QS\_SIGNALSET) **goto** FLAG\_SENDPROCLOOP;

**if** (threadInfo.QS\_POSTMESSAGE == QS\_SIGNALSET) {

        //从登记消息队列中获取消息

        dwReturnVal = GetMsgFromQueue(QUEUE\_POST, lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax);

        //判断是否还有登记消息，没有了则复位QS\_POSTMESSAGE标志

**if** (dwReturnVal == GETMESSAGE\_LASTMESSAGE)

            threadInfo.QS\_POSTMESSAGE = QS\_SIGNALRESET;

**return** TRUE;

    }

    //如果退出标志被置位

**if** (threadInfo.QS\_QUIT == QS\_SIGNALSET) {

        threadInfo.QS\_QUIT = QS\_SIGNALRESET;

        FillMessage(lpMsg, MESSAGE\_QUIT);

**return** FALSE;

    }

    //检查输入消息队列

**if** (threadInfo.QS\_INPUT == QS\_SIGNALSET) {

**DWORD** dwRetVal = GetMessageFromQueue(QUEUE\_INPUT, lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax);

        //检查是否有键盘，鼠标消息

**if** (Test(dwRetVal, QS\_KEY) == QS\_LASTMOUSEKEYMESSAGE)

            threadInfo.QS\_KEY = QS\_SIGNALRESET;

**if** (Test(dwRetVal, QS\_MOUSEBUTTON) == QS\_LASTMOUSEMESSAGE)

            threadInfo.QS\_MOUSEBUTTON = QS\_SIGNALRESET;

**return** TRUE;

    }

    //测试QS\_PAINT

**if** (threadInfo.QS\_PAINT == QS\_SIGNALSET) {

        //填充MSG，如果没有窗口过程确认窗口，则复位QS\_PAINT标志

        //...

        //返回TRUE

        threadInfo.QS\_PAINT = QS\_SIGNALRESET;

**return** TRUE;

    }

**if** (threadInfo.QS\_TIMER == QS\_SIGNALSET) {

        //填充MSG，如果没有定时器报时，则复位QS\_TIMER标志

        //...

        //返回TRUE

**return** TRUE;

    }

    //等待有消息到达

    dwRetVal = MsgWaitForMultipleObjectsEx(...);

**if** (...)

**goto** FLAG\_SENDPROCLOOP;

    //等待失败

**return** FALSE;

}

当一个线程调用GetMessage或WaitMessage，但没有对这个线程或这个线程所建立窗口的消息时，系统可以挂起这个线程，这样就不再给它分配CPU时间。当有一个消息登记或发送到这个线程，系统要设置一个唤醒标志，指出现在要给这个线程分配CPU时间，以便处理消息

|  |  |
| --- | --- |
| 标志 | 含义 |
| QS\_SENDMESSAGE | 另一个线程SendMessage的消息 |
| QS\_POSTMESSAGE | PostMessage、PostThreadMessag发送的消息 |
| QS\_QUIT | 已调用PostQuitMessage |
| QS\_INPUT | 鼠标键盘消息 |
| QS\_PAINT | WM\_PAINT |
| QS\_TIMER | WM\_TIMER |

当一个线程调用PostQuitMessage时，QS\_QUIT标志就被设置。系统并不实际向线程的消息队列追加一个WM\_QUIT消息

如果线程建立的一个窗口有无效的区域， QS\_PAINT标志被设置。

GetMessage(&msg)如果检测到QS\_PAINT被设置，则msg被填充为一个WM\_PAINT消息

**WM\_CLOSE与WM\_DESTROY与PostQuitMessage**

* WM\_CLOSE：关闭应用程序窗口
* WM\_DESTROY：关闭应用程序
* PostQuitMessage：关闭消息循环，即**while** (GetMessage(&msg, NULL, 0, 0)){}

只有关闭了消息循环，应用程序的进程才真正退出(在任务管理器里消失)

GetMessage()函数如果出现错误，则返回-1；对于WM\_QUIT消息，该函数返回零，其余消息均返回非零值

win32应用程序的完整退出过程：点击窗口右上角的关闭按钮，发送WM\_CLOSE消息。WM\_CLOSE默认地被DefWindowProc处理。然后，当窗口销毁时，发送WM\_DESTROY消息。一般窗口过程中

    case WM\_DESTROY:

        PostQuitMessage(0);

        break;

PostQuitMessage()函数，设置QS\_QUIT，GetMessage处理完发送消息队列、登记消息队列检测到QS\_QUIT，返回FALSE，退出循环，WinMain函数退出，进程退出。

# WaitForSingleObject函数

Windows提供了许多内核对象来实现线程的同步，如互斥量CreateMutex、信号量CreateSemaphore等。对于线程同步而言，这些内核对象有两个非常重要的状态：“已通知”状态，“未通知”状态（或翻译为：受信状态，未受信状态）。

WaitForSingleObject函数等待一个内核对象（hObject）变为已通知状态：

**DWORD** WaitForSingleObject(  **HANDLE** hObject  **DWORD** dwMilliseconds);

在内核对象产生了信号，或者超过规定的等待时间函数才会返回。在等待函数未返回时，线程处于等待状态，此时线程只消耗很少的CPU时间。

第二个参数指明了需要等待的时间（毫秒），可以传递INFINITE指明要无限期等待下去，如果第二个参数为0，那么函数就测试同步对象的状态并立即返回。

如果内核对象在等待时间内变为已通知状态，则函数返回WAIT\_OBJECT\_0

如果等待超时，该函数返回WAIT\_TIMEOUT。

如果该函数失败，返回WAIT\_FAILED。

可以通过下面的代码来判断：

**DWORD** dw = WaitForSingleObject(hProcess, 5000); //等待一个进程结束

**switch** (dw)

{

**case** WAIT\_OBJECT\_0:  // hProcess所代表的进程在5秒内结束

**break**;

**case** WAIT\_TIMEOUT: // 等待时间超过5秒

**break**;

**case** WAIT\_FAILED: // 函数调用失败，比如传递了一个无效的句柄

**break**;

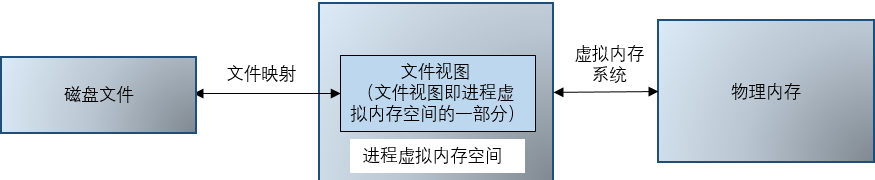
}

# 共享内存：

共享内存是进程间通信手段之一

同一进程下的多个线程共享进程虚拟内存空间，本就共享了内存

文件映射，将磁盘文件与映射到虚拟内存



操作系统创建一个文件映射对象来负责管理一个文件映射

//Win32 API，创建文件映射对象

**HANDLE** CreateFileMapping(

**HANDLE** hFile,                       //物理文件句柄

  LPSECURITY\_ATTRIBUTES lpAttributes, //安全设置，一般设为nullptr

**DWORD** flProtect,                    //保护设置

**DWORD** dwMaximumSizeHigh,            //申请的虚拟内存大小的高位

**DWORD** dwMaximumSizeLow,             //申请的虚拟内存大小的低位

**LPCTSTR** lpName                      //文件映射对象的名称，唯一标识一个文件映射对象

);

参数分析：

* hFile，物理文件句柄，指向要关联的磁盘文件

1. 如果需要和物理文件关联，要确保物理文件创建的时候的访问模式和由flProtect参数指定的"保护标识"匹配，比如：物理文件只读， 内存映射需要读写就会发生错误。
2. 如果要创建一个物理文件无关的文件映射，hFile取NVALID\_HANDLE\_VALUE，同时在参数dwMaximumSizeHigh和dwMaximumSizeLow中指定要申请的虚拟内存空间的大小，这样CreateFileMapping就可以创建一个和物理文件大小无关的内存空间给你, 甚至超过实际文件大小。如果第一个参数表示的物理文件有效, 而大小参数为0, 则返回的是一个和物理文件大小一样的虚拟内存空间地址范围

* flProtect，对申请的虚拟内存的访问权限

|  |  |
| --- | --- |
| 取值 | 描述 |
| PAGE\_READONLY | 只有读的权限，试图写或者执行的操作都是非法的 |
| PAGE\_READWRITE | 同时有读和写的权限 |
| PAGE\_WRITEONLY | 只有写的权限 |

* dwMaximumSizeHigh和dwMaximumSizeLow

DWORD是unsignd long，如果要申请的虚拟内存大小超过unsigned long表示范围，如uint64\_t s，则s的高32位存入dwMaximumSizeHigh，低32位存入dwMaximumSizeLow

MapViewOfFile用于将存储于磁盘的文件放进一个进程的虚拟地址空间，进程的虚拟地址空间中"存放"该文件的区域就叫做File View。系统同时产生一个存放在物理内存中的File Mapping Object用于维护这种映射关系。这时当多个进程需要读写那个文件数据时，它们的File View其实对应的都是同一个File Mapping Object

**LPVOID** MapViewOfFile(

**HANDLE** hFileMappingObject,   // handle to file-mapping object

**DWORD** dwDesiredAccess,       // access mode

**DWORD** dwFileOffsetHigh,      // high-order DWORD of offset

**DWORD** dwFileOffsetLow,       // low-order DWORD of offset

**SIZE\_T** dwNumberOfBytesToMap  // number of bytes to map

);

参数分析：

* hFileMappingObject，一个打开的文件映射对象的句柄，这个句柄可以由CreateFileMapping和OpenFileMapping函数返回
* dwDesiredAccess对文件视图的访问权限，还要与CreateFileMapping()函数所设置的保护属性相匹配

|  |  |
| --- | --- |
| 取值 | 描述 |
| FILE\_MAP\_READ | 读取.在调用CreateFileMapping时可以传入PAGE\_READONLY或PAGE\_READWRITE保护属性 |
| FILE\_MAP\_WRITE | 读取和写入.在调用CreateFileMapping时必须传入PAGE\_READWRITE保护属性. |
| FILE\_MAP\_EXECUTE | 将文件视图中的数据作为代码来执行 |
| FILE\_MAP\_COPY | 读取和写入.写入操作会导致系统为该页面创建一份副本.在调用CreateFileMapping时必须传入PAGE\_WRITECOPY保护属性 |
| FILE\_MAP\_ALL\_ACCESS | 等价于FILE\_MAP\_WRITE|FILE\_MAP\_READ. 文件映射对象被创建时必须指定PAGE\_READWRITE 选项 |

* dwFileOffsetHigh指定开始映射文件偏移量的高位。
* dwFileOffsetLow指定开始映射文件偏移量的低位。
* dwNumberOfBytesToMap指定需要映射的文件的字节数量，如果为0，则映射整个的文件。

共享内存实现进程间通信的程序示例：

//进程A

**HANDLE** hMutex = CreateMutex(nullptr, FALSE, TEXT("\_\_TEST\_MUTEX\_\_"));

**if** (!hMutex)

{

    cout << "Failed to create mutex!" << "     Error: " << GetLastError() << endl;

**return** 1;

}

//创建文件映射对象，文件映射对象记录文件映射关系

//传入的最后一个参数作为创建的文件映射对象的名称

//文件映射对象的名称唯一标识一个文件映射对象

hMapping = CreateFileMapping(INVALID\_HANDLE\_VALUE,nullptr,PAGE\_READWRITE,0,

**sizeof**(COMMUNICATIONOBJECT), TEXT("\_\_FILE\_MAPPING\_\_"));  // COMMUNICATIONOBJECT是一个结构体，含bExitLoop hWndClient等成员

**if** (!hMapping)

{

    cout << "Failed to create file mapping object!" << "     Error: " << GetLastError() << endl;

**return** 1;

}

//将磁盘文件映射到虚拟内存，返回虚拟内存地址（即文件视图）

PCOMMUNICATIONOBJECT pCommObject = (PCOMMUNICATIONOBJECT)MapViewOfFile(hMapping,   FILE\_MAP\_WRITE,0, 0, 0);//最后一个参数指定需要映射的文件的字节数量，如果为0，则映射整个的文件

**if** (pCommObject)

{

    pCommObject->bExitLoop = FALSE;

    pCommObject->hWndClient=hWnd;

    pCommObject->lSleepTimeout=250;

    UnmapViewOfFile(pCommObject);//不再需要把文件映射到进程的虚拟地址空间

}

CreateProcess(TEXT("进程B.exe"),……)

进程B：

//获取已命名互斥量的句柄

**HANDLE** hMutex = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, TEXT("\_\_TEST\_MUTEX\_\_"));

//打开已命名的文件映射对象，返回句柄

**HANDLE**  hMapping = OpenFileMapping(FILE\_MAP\_READ, FALSE, TEXT("\_\_FILE\_MAPPING\_\_"));

//等待内核对象（第一个参数为其句柄）变为已通知状态，否则永远（INFINATE）阻塞

WaitForSingleObject(hMutex, INFINITE);  //类似于std::mutex::lock()

//获取文件映射对象对应的内存地址，文件视图即文件对应的虚拟内存地址

pCommObject = (PCOMMUNICATIONOBJECT)MapViewOfFile(hMapping, FILE\_MAP\_READ,

    0, 0, **sizeof**(COMMUNICATIONOBJECT));

cout<<!pCommObject->bExitLoop<<endl;  //访问

UnmapViewOfFile(pCommObject);//不再需要把文件映射到进程的虚拟地址空间

ReleaseMutex(hMutex); //释放互斥量所有权，让其它进程可以获得互斥量 ，类似于std::mutex::unlock()

这样，进程A、B实现了通信，二者共同访问（读写）共享内存

# VS的Win32项目与Win32控制台应用程序

VS中开发图形界面程序一般新建Win32项目，入口函数为

//主窗体函数(入口过程)

//每个基于 Win32 的应用程序的函数必须具有 WinMain 函数

//正如每个 C 应用程序和 C++ 控制台应用程序在起始点必须具有 main 函数

//hInstance：应用程序当前实例的句柄

//hPrevInstance：应用程序上一个实例的句柄，取值为NULL

//nCmdShow：控制如何显示窗口，取值SW\_SHOWDEFAULT

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

//如果WinMain函数下文用不到参数hPrevInstance、pCmdLine，则可用UNREFERENCED\_PARAMETER宏告诉编译器下文不使用某个参数，方便编译器优化

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

}

加载到进程地址空间的每一个.exe或者.dll文件都被赋予了一个独一无二的实例句柄。可执行文件的实例被当作WinMain函数的第一个参数hInstanceExe传入

* 控制台程序的执行默认就好像在一个DOS窗口中执行一样，没有自己的界面。
* 在Win32控制台应用程序也可以显示图形界面，同时也可以不显示控制台

这时HINSTANCE hInstance= GetModuleHandle(nullptr);

ShowWindow(主窗口句柄, SW\_SHOWDEFAULT);

如果在控制台应用程序中不想显示控制台：

HWND hConsole = GetConsoleWindow();

ShowWindow(hConsole, SW\_HIDE);

但会有黑框一闪而过

HMODULE GetModuleHangdle ( LPCTSTR lpModuleName)

如执行成功，返回模块句柄。失败则返回0

参数为模块名，即dll或exe文件名，只能是映射到当前进程的模块

参数为NULL，函数将返回创建当前进程的exe文件的句柄

在TextC++Syntax.exe的main()函数中有以下语句，运行结果为：

cout << (**void**\*)GetModuleHandle(nullptr) << endl; //01270000

cout << (**void**\*)GetModuleHandle(TEXT("TestC++Syntax.exe"))<< endl; //01270000

cout << (**void**\*)GetModuleHandle(TEXT("User32.dll")) << endl; //73C30000

# CreateProcess创建进程

一个线程调用CreateProcess时，系统将创建一个进程内核对象，然后为之创建虚拟地址空间，并将可执行文件、所有必要的DLL加载到该地址空间。之后，系统为新进程的主线程创建一个线程内核对象。这个主线程一开始就会执行C\C++运行时的启动例程，它是由链接器设为应用程序入口的，最终会调用应用程序WinMain，wWinMain,main或wmain函数。如果系统成功创建了新进程和主线程，CreateProcess返回TRUE。注意：CreateProcess在进程完全初始化好之前就返回true，如果一个dll找不到或不能正确初始化，子进程就会终止，而父进程不会注意到任何初始化问题。

BOOL CreateProcess(

LPCTSTR pszApplicationName,// name of executable module

LPTSTR pszCommandLine,// command line string

LPSECURITY\_ATTRIBUTES psaProcess,// SD

LPSECURITY\_ATTRIBUTES psaThread,// SD

BOOL bInheritHandles,// handle inheritance option

DWORD fdwCreationFlags,// creation flags

LPVOID pvEnvironment,// new environment block

LPCTSTR pszCurDir,// current directory name

LPSTARTUPINFO psiStartupInfo,// startup information

LPPROCESS\_INFORMATION lpProcessInformation // process information

);

STARTUPINFO startupInfoRed = { 0 };

PROCESS\_INFORMATION processInfoRed = { 0 };

**BOOL** bSuccess = CreateProcess(TEXT("..\\x64\\Debug\\IPCWorker.exe"), TEXT("red"), nullptr,

nullptr, FALSE, 0, nullptr, nullptr, &startupInfoRed, &processInfoRed);  // IPCWorker.exe的main函数需要传入一个字符串参数，即这里的TEXT("red")

**if** (!bSuccess)

{

    cout << "Failed to create process red!" << "     Error: " << GetLastError() << endl;

**return** 1;

}

# CHAR、WCHAR、LPSTR 、LPWSTR等

typedef char CHAR;

typedef wchar\_t WCHAR;

#ifdef UNICODE

typedef wchar\_t TCHAR;

#else

typedef char TCHAR;

#endif

* CHAR 即8 位的char
* WCHAR 即16位的wchar\_t（在C++中一般为16位，但有个别的库设定为32位）
* TCHAR 要么是8位的CHAR（未定义UNICODE）要么是16位的WCHAR（定义了UNICODE）

在VS中，项目属性—配置属性--常规—字符集：

如果选择使用Unicode字符集，则定义了UNICODE宏

如果选择使用多字节字符集（MBCS），则未定义UNICODE宏

LPSTR 即char\*

LPCSTR 即const char\*

LPWSTR 即wchar\_t\*

LPCWSTR即const wchat\_t\*

LPTSTR即TCHAR\*

LPCTSTR即const TCHAR\*

如果使用MBCS字符集一般这么写：

定义一个MBCS字符数组：char arr[LEN];或者CHAR arr[LEN];

定义一个MBCS字符指针：char \*p;或者LPSTR p;

定义一个MBCS常量字符串指针：const char \* cp;或者LPCSTR cp;

定义一个MBCS常量字符串：cp=”Hello World!\n”;

如果使用Unicode字符集一般这么写：

定义一个Unicode字符数组：wchar\_t arr[LEN];或者WCHAR arr[LEN];

定义一个Unicode字符指针：wchar\_t \*p;或者LPWSTR p;

定义一个Unicode常量字符串指针：const wchar\_t \* cp;或者LPCWSTR cp;

定义一个Unicode常量字符串：cp=L”Hello World!\n”;

一般字符集和串操作离不开。

对MBCS字符串连接、复制、比较、求长运算为：strcat、strcpy、strcmp、strlen。

对Unicode字符串连接、复制、比较、求长运算为：wcscat、wcscpy、wcscmp、wcslen。

为屏蔽字符集造成的代码差异，Windows提供了头文件tchar.h，它使用一个UNICODE宏来标识当前工程使用的字符集是MBCS还是Unicode。

对于相应的字符集定义和串操作如下：

定义一个字符数组：TCHAR arr[LEN];

定义一个字符指针：LPTSTR p;

定义一个常量字符串指针：LPCTSTR cp;

定义一个常量字符串：cp=\_T(”Hello World!\n”);或cp=TEXT(”Hello World!\n”)

连接、复制、比较、求长运算为：\_tcscat、\_tcscpy、\_tcscmp、\_tcslen。

TCHAR不是一个新的类型，它是根据UNICODE宏来自动映射为char和wchar\_t，相应的LPTSTR、LPCTSTR、\_T()宏（即TEXT宏）亦是如此

int \_tmain(int argc, TCHAR\* argv[])

{

int iIndex = (int)\_tcstol(argv[0],nullptr,10);

}

if (!hMapping)

{

TCHAR des[MAX\_LOADSTRING];

#ifdef UNICODE

wsprintf(des, L"Error: %u", GetLastError()); //GetLastError返回DWORD类型，即unsigned long

#else

sprintf(des, "Error: %u", GetLastError());

#endif

MessageBox(NULL, TEXT("Failed to creat file mapping object"), des, MB\_OK);

return 1;

}

char\*与wchar\_t\*的互相转化：

WCHAR src[20]=L"wide 字符";  
CHAR des[40];  
WCHAR des2[20];  
setlocale(LC\_ALL,"");  
wcstombs(des,src,20);  
cout<<des<<endl;  
mbstowcs(des2,des,40);  
wcout<<des2<<endl;

# InvalidateRect与WM\_PAINT消息

**void** FillEllipse(**HWND** hWnd, **HDC** hDC, **int** iLeft, **int** iTop, **int** iRight, **int** iBottom, **int** iPass)

{

**HBRUSH** hBrush = nullptr;

**if** (iPass) { hBrush = CreateSolidBrush(RGB(255, 0, 0)); }

**else** { hBrush = CreateSolidBrush(RGB(0, 255, 255)); }

    //SelectObject把一个对象(位图、画笔、画刷等)选入指定的设备描述表。新的对象代替同一类型的老对象。

**HBRUSH** holdBrush = (**HBRUSH**)SelectObject(hDC, hBrush);

    Ellipse(hDC, iLeft, iTop, iRight, iBottom);

    SelectObject(hDC, holdBrush);

    DeleteObject(hBrush);

}

**LRESULT** CALLBACK WndProc(**HWND** hWnd, **UINT** message, **WPARAM** wParam, **LPARAM** lParam)

{

**HDC** hdc;

**switch** (message)

    {

**case** WM\_INVALIDATE:  //程序修改了iPass变量，即要改变椭圆的颜色

        //将矩形区域（第二个参数）添加到已有更新区域 ；如果RECT\*为nullptr，则添加整个客户区域到更新区域

        //第三个参数决定调用BeginPaint()函数时是否擦除当前更新区域现有图形

//InvalidateRect自身并不发送WM\_PAINT消息

        InvalidateRect(hWnd, nullptr, FALSE);

**break**;

**case** WM\_PAINT:

        PAINTSTRUCT ps;

        //BeginPaint函数的作用是告诉Windows系统，要开始向显示卡输出内容了，把这次显示的操作

        //请求放到系统显示队列里。操作系统会让显示操作线性化，保证每个窗口的显示是独立进行的，

        //而不是A窗口显示一部份，或者B窗口显示一部份，而是A窗口显示完成后再让B窗口显示。

        //因此，BeginPaint返回时，就获取到系统的显示资源句柄，之后就可以调GDI函数来操作了。

        //显示完成后，调用函数EndPaint释放显示资源，否则其它程序无法获取到显示资源

//BeginPaint清空更新区域Update Region

//当窗口的更新区域被标志为需要擦除背景时，BeginPaint会发送WM\_ERASEBKGND消息来重画背景

        hdc = BeginPaint(hWnd, &ps);

        FillEllipse(hWnd, hdc, 210, 10, 310, 110, iPass);

        EndPaint(hWnd, &ps);

**break**;

**case** WM\_DESTROY:

        PostQuitMessage(0);

**break**;

**default**:

**return** DefWindowProc(hWnd, message, wParam, lParam);

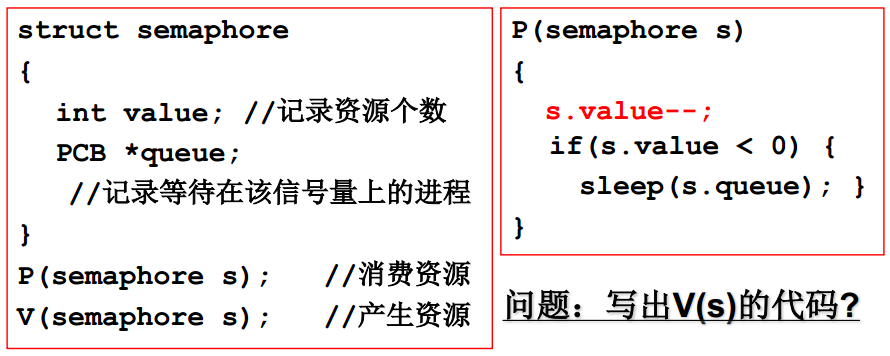
    }

**return** 0;

}

# 信号量

信号量是进程间通信手段之一



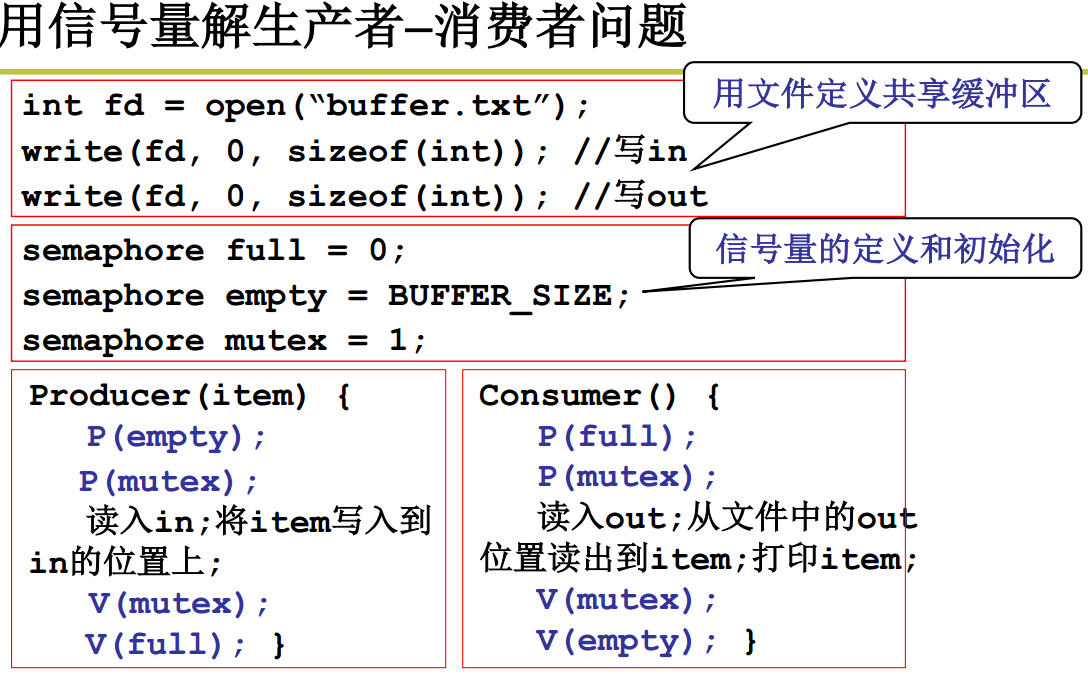
V(semaphore s)

{

s.value++;

if(s.value<=0) wakeup(s.queue)

}



资源的初值要设置

使用资源前调P

产生资源后调V

//进程A

**HANDLE** hSemaphore = CreateSemaphore(nullptr, COUNT/2,  COUNT/ 2,szSemaphoreName);

……//调用进程B、C

CloseHandle(hSemaphore);

//进程B、C

**HANDLE** hSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS,FALSE,szSemaphoreName);

WaitForSingleObject(hSemaphore,INFINITE);

……//访问共享内存

ReleaseSemaphore(hSemaphore,1,NULL);

CloseHandle(hSemaphore);

* HANDLE WINAPI CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpEventAttributes, //事件对象的安全属性，一般置为NULL

LONG lInitialCount, //**一开始可以使用的资源数目，即当前资源计数**

LONG lMaximumCount, //**资源数量总数**

LPCTSTR lpName //创建有名的信号量对象，用于进程间的共享

);如果该信号量对象已经存在，那么CreateSemaphore会返回该内核对象的句柄，并通过系统返回错误ERROR\_ALREADY\_EXISTS，通过GetLastError()获得

* 对一个信号量，如果当前资源计数大于0，信号量处于通知状态；

如果当前资源计数等于0，那么信号量处于未通知状态

调用等待函数WaitForSingleObject，传递信号量句柄，如果信号量处于未通知状态，函数不返回，线程阻塞。如果信号量处于已通知状态，计数器递减1，调用线程保持可调度状态。信标的出色之处在于它们能够以原子操作方式来执行测试和设置操作

* BOOL WINAPI ReleaseSemaphore(HANDLE hSemaphore,LONG lReleaseCount,LPLONG lpPreviousCount) 该函数**只是将lReleaseCount中的值添加给信号量的当前资源数量，通常为1，根据线程使用的资源数目**。可以通过参数\*plPreviousCount中返回当前资源数量的原始值，但实际上几乎没有应用程序关心这个值，因此可以传递NULL，将它忽略

# CreateThread与线程栈内存结构

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes, //指向SECURITY\_ATTRIBUTES的指针，用于定义新线程的安全属性，一般设置成NULL

DWORD dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress, //线程入口函数指针，LPTHREAD\_START\_ROUTINE为DWORD(\*p)(LPVOID\*)类型

LPVOID lpParameter, //传递给线程的参数

DWORD dwCreationFlags, //表示创建线程的运行状态，CREATE\_SUSPEND表示挂起当前创建的线程，0表示立即执行当前创建的进程

LPDWORD lpThreadID //返回新创建的线程的ID

);

如果函数调用成功，则返回新线程的句柄，调用WaitForSingleObject函数等待所创建线程的运行结束

DWORD为unsigned long类型

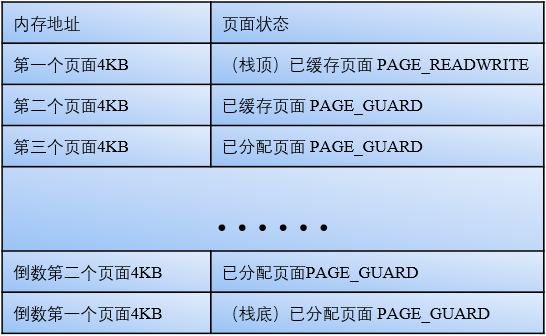
LPVOID为void\*类型

ResumeThread 恢复线程的运行; SuspendThread 挂起线程.  
这两个函数的参数都是线程句柄, 返回值是执行前的挂起计数.  
什么是挂起计数?  
SuspendThread 会给这个数 +1; ResumeThread 会给这个数 -1; 但这个数最小是 0.  
当这个数 = 0 时, 线程会运行; > 0 时会挂起.  
如果被 SuspendThread 多次, 同样需要 ResumeThread 多次才能恢复线程的运行

Windows线程默认堆栈的大小为1M。虚拟页大小与物理页相同，为4KB。

初始默认情况下，操作系统会为线程栈预订1MB内存，并调拨2个页面的空间到物理内存。

* 预定，reserve，即带MEM\_RESERVE参数的VirtualAlloc函数，即分配虚拟内存，虚拟页由未分配变为已分配状态。起始地址必须是系统分配粒度（64KB）的整数倍，大小必须是系统页面大小（4KB）的整数倍
* 调拨，commit，即带MEM\_COMMIT参数的VirtualAlloc函数，即将已分配的虚拟页缓存到物理内存，或者说调拨预订的虚拟内存到物理内存。起始地址和大小都必须是页面大小的整数倍(4KB)



如图，前两个已分配的虚拟页被缓存到物理内存，第一个页面被设置成了PAGE\_READWRITE，第二个页面被设置成PAGE\_GUARD。随着调用越来越多的函数，线程需要越来越多的栈，当访问到第二个页面的时候系统会修改该页面的PAGE\_GUARD为PAGE\_READWRITE，并缓存下一个页面。当线程访问到倒数第三个页面的的时候，系统会缓存倒数第二个页面至物理内存，同时抛出EXCEPTION\_STACK\_OVERFLOW，如果用结构化异常处理掉了，并且线程继续使用栈空间，那么倒数第二个页面会被用尽，此时不得不访问栈底页面，然而栈底页面并没有被缓存到物理内存，这时发生的访问违规将终止整个进程！即栈溢出错误！

线程的预订堆栈默认大小和初始提交的堆栈默认大小在可执行文件头中指定。借助CreateThread函数的dwCreationFlags和dwStackSize参数，可以自定义线程的预订内存和初始调拨的内存

* 要更改最初提交的堆栈大小，可使用[CreateThread](https://msdn.microsoft.com/en-us/library/ms682453(v=VS.85).aspx)，[CreateRemoteThread](https://msdn.microsoft.com/en-us/library/ms682437(v=VS.85).aspx)或[CreateFiber](https://docs.microsoft.com/en-us/windows/desktop/api/WinBase/nf-winbase-createfiber)函数的dwStackSize参数。此值向上舍入到最近的页面大小。通常，保留大小是可执行文件头中指定的默认保留大小。但是，如果dwStackSize指定的最初提交的大小大于或等于默认保留大小，则保留大小为此新提交大小四舍五入到最接近的1 MB的倍数。
* 要更改保留的堆栈大小，请将[CreateThread](https://msdn.microsoft.com/en-us/library/ms682453(v=VS.85).aspx)或[CreateRemoteThread](https://msdn.microsoft.com/en-us/library/ms682437(v=VS.85).aspx)的dwCreationFlags参数设置为STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION并使用dwStackSize参数。在这种情况下，最初提交的大小是可执行标头中指定的默认大小。

**双重检查锁：**

双重检查锁，DCLP（Double-Checked Locking Pattern）

//多线程下，不适用

singleton \*singleton::p = nullptr;

singleton\* singleton::instance() {

if (p == nullptr) p = new singleton();

return p;

}

//逻辑正确，但性能差，每次调用都要加锁

singleton \*singleton::p = nullptr;

singleton\* singleton::instance() {

lock\_guard<mutex> guard(lock\_);

if (p == nullptr) p = new singleton();

return p;

}

//逻辑错误，p = new singleton()执行时，由于cpu乱序重排，可能先给p赋值，然后才初始化p指向的内存

//另一个线程调用instance，判定p非空， 将其返回给instance的调用者，于是调用者可能使用一个还没有被构造出的对象

singleton \*singleton::p = nullptr;

singleton\* singleton::instance() {

if (p == nullptr) {

lock\_guard<mutex> guard(lock\_);

if (p == nullptr)

p = new singleton();

}

return p;

}

//最佳实现1

singleton\* singleton::p = nullptr;

std::once\_flag Singleton::m\_flag;

singleton\* singleton::instance() {

std::call\_once(m\_flag, [&](){p = new singleton();});

return p;

}

//使用相对开销低的atomic保护开销大得多的mutex的使用,同时同时构建了良好的内存屏障

mutex mutex\_x;

atomic<bool> init\_x; // 初始为 false

int x;

if (!init\_x) {

lock\_guard<mutex> lck(mutex\_x);

if (!init\_x) x = 42;

init\_x = true ;

} // 在此隐式释放 mutex\_x（RAII）

//最佳实现2，赋值函数和类型转换函数实际相当于采用了std::memory\_order\_seq\_cst的store和release

std::atomic<singleton\*> singleton::p = nullptr;

singleton\* singleton::instance() {

if (p == nullptr) {

lock\_guard<mutex> guard(lock\_);

if (p == nullptr) p = new singleton();

}

return p;

}

//将最佳实现2里的std::memory\_order\_seq\_cst改为acquire、release内存序也可以

std::atomic<singleton\*> singleton::p = nullptr;

singleton\* singleton::instance() {

if (p.load(std::memory\_order\_acquire)==nullptr) {

lock\_guard<mutex> guard(lock\_);

if (p.load(std::memory\_order\_acquire) == nullptr)

p.store(new singleton(), std::memory\_order\_release);

}

return p;

}

//最佳实现3

singleton\* singleton::p = nullptr;

std::atomic<bool> assigned=false;

singleton\* singleton::instance() {

if(!assigned)

{

lock\_guard<mutex> guard(lock\_);

{

if(!assigned ) p = new singleton();

assigned=true;

}

}

return p;

}

**一次性通知-响应模型：**

（1）使用条件变量wait(lock, pred)

（2）一次通知+单个线程响应，响应中含多次执行：

class A{

public:

void worker(){

for (int i = 0; i < 10; i++) {

if (m\_first) {

m\_bstarted.get\_future().wait();

m\_first = false;

}

std::cout << "work..." << std::endl;

}

}

void doTask(){

std::thread thr([this](){

worker();

});

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

std::cout << "ready!" << std::endl;

m\_bstarted.set\_value();

thr.join();

}

private:

std::promise<void> m\_bstarted;

bool m\_first = true;

};

（3）一次通知+多个线程响应：

std::future的share成员函数将共享状态所有权转移到share()产生的std::shared\_future中

每个反应线程都需要自己的std::shared\_future副本，该副本引用共享状态，因此通过share获得的shared\_future要被在反应线程中运行的lambda按值捕获

class B{

public:

void worker(const std::shared\_future<void>& sf) {

sf.wait();

std::cout << "work..." << std::endl;

}

void doTask() {

std::shared\_future<void> sf = m\_bstarted.get\_future().share();

std::vector<std::thread> thrpool;

for (int i = 0; i < 10; i++) {

thrpool.emplace\_back([this, sf]() { worker(sf); });

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

std::cout << "ready!" << std::endl;

m\_bstarted.set\_value();

for (auto&& thr : thrpool) thr.join();

}

private:

std::promise<void> m\_bstarted;

};

**原子增fetch\_add：**

std::atomic<T>的成员函数：

T fetch\_add( T arg, std::memory\_order order = std::memory\_order\_seq\_cst );

T operator++()  { return fetch\_add(1)+1; }

T operator++(int)  { return fetch\_add(1); }

T operator+=(T arg) { return fetch\_add(arg) + arg;}

fetch\_add是read-modify-write原子操作，返回的是修改之前的旧值

仿照std::call\_once，实现一个简陋版的call\_twice：

template <class Func, class... Args>

auto  call\_twice(Func&& func, Args&&... args)

{

    static std::atomic<int> ccnt(0);

    int v = ccnt.fetch\_add(1) + 1;

    if (v <= 2)

    {

        return std::forward<Func>(func)(std::forward<Args>(args)...);

    }

}