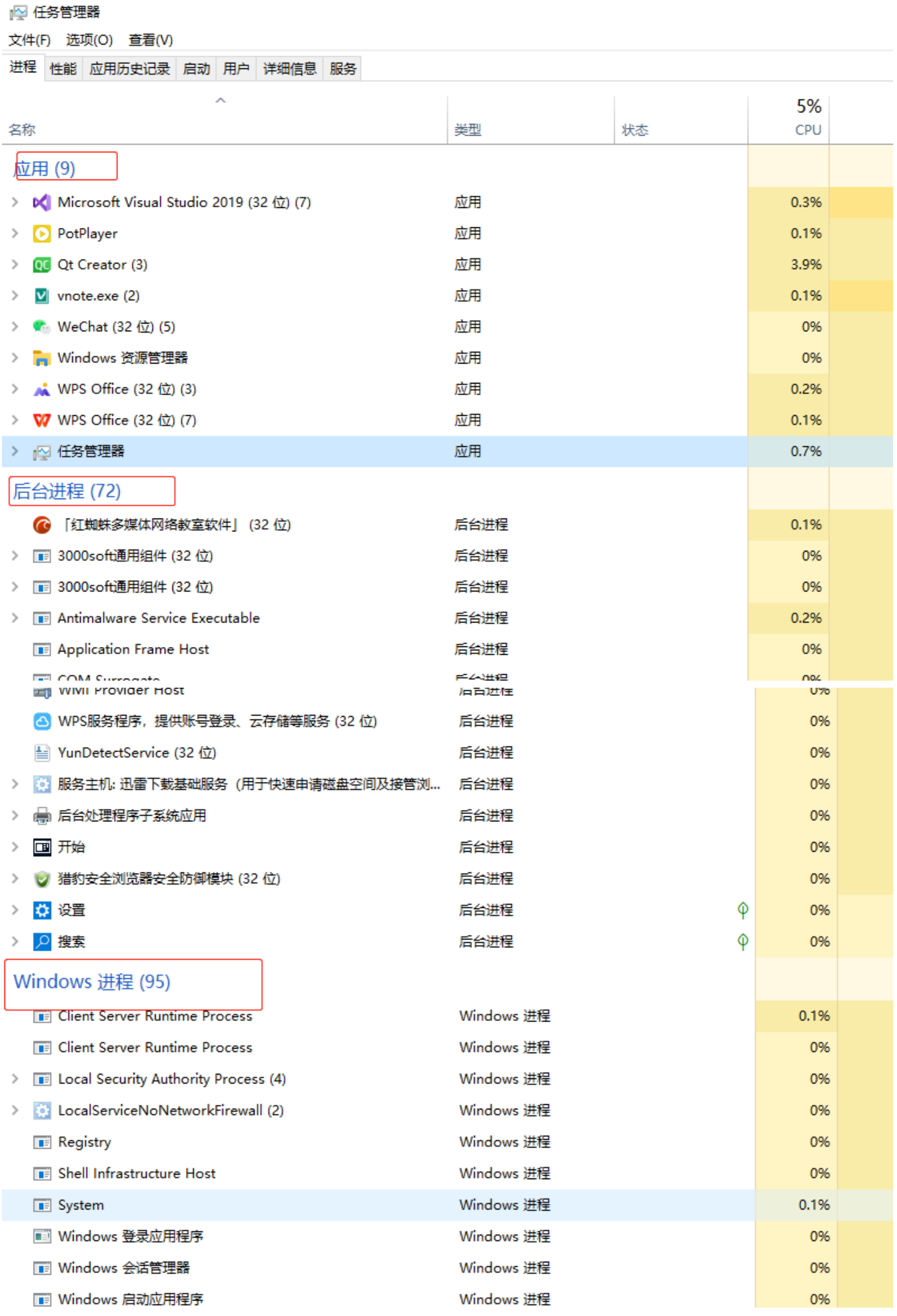
第6章-进程线程

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

1. 进程（Process）

在【任务管理器】中，能看到【进程】Tab页下，将所有的进程分为三类（*菜单【查看】->【按类型分组】*）。
应用：打开的正在运行的软件。vs、Qt、wps、微信...
后台进程：隐藏到后台，“悄悄”的运行，如： 红蜘蛛、驱动程序，杀毒软件...
window进程：操作系统启动、运行需要依赖的各种服务...

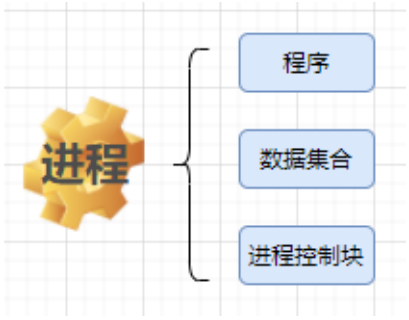

任务管理器
文件(F) 选项(O) 查看(V)
进程 性能 应用历史记录 启动 用户 详细信息 服务
名称 类型 状态 5% CPU
应用 (9)
Microsoft Visual Studio 2019 (32 位) (7) 应用 0.3%
PotPlayer 应用 0.1%
Qt Creator (3) 应用 3.9%
vnote.exe (2) 应用 0.1%
WeChat (32 位) (5) 应用 0%
Windows 资源管理器 应用 0%
WPS Office (32 位) (3) 应用 0.2%
WPS Office (32 位) (7) 应用 0.1%
任务管理器 应用 0.7%
后台进程 (72)
「红蜘蛛多媒体网络教室软件」 (32 位) 后台进程 0.1%
3000soft通用组件 (32 位) 后台进程 0%
3000soft通用组件 (32 位) 后台进程 0%
Antimalware Service Executable 后台进程 0.2%
Application Frame Host 后台进程 0%
WPS服务程序，提供账号登录、云存储等服务 (32 位) 后台进程 0%
YunDetectService (32 位) 后台进程 0%
服务主机: 迅雷下载基础服务 (用于快速申请磁盘空间及接管浏... 后台进程 0%
后台处理程序子系统应用 后台进程 0%
开始 后台进程 0%
猎豹安全浏览器安全防御模块 (32 位) 后台进程 0%
设置 后台进程 0%
搜索 后台进程 0%
Windows 进程 (95)
Client Server Runtime Process Windows 进程 0.1%
Client Server Runtime Process Windows 进程 0%
Local Security Authority Process (4) Windows 进程 0%
LocalServiceNoNetworkFirewall (2) Windows 进程 0%
Registry Windows 进程 0%
Shell Infrastructure Host Windows 进程 0%
System Windows 进程 0.1%
Windows 登录应用程序 Windows 进程 0%
Windows 会话管理器 Windows 进程 0%
Windows 启动应用程序 Windows 进程 0%

各种软件、进行是依附在操作系统上运行的。

程序是存在存储器里面的可执行代码，当我们双击快捷方式时，操作系统就会将代码从存储器中取出来开始执行，并且给没一个执行的进程分配一个PID。windows操作系统上的应用程序有的是可以开启多个的，比如说文件夹、浏览器等，有的应用程序只允许开启一个，比如说电脑版的微信。手机系统上的APP一般只允许开启一个。

进程概念：是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立的基本单元，是应用程序运行的载体。
进程是一种抽象的概念，从来没有统一的标准定义。

进程组成：程序、数据集合、进程控制块三部分组成。



进程具有4个特征：
1、动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的。
2、并发性：任何进程都可以同其他进程一起并发执行。
3、独立性：进程是系统进行资源分配和调度的一个独立单元。
4、结构性：进程由程序，数据和进程控制块三部分组成。

应用程序和进程间的关系：从任务管理器的详细信息页面可以看出，一个应用程序可以启动多个进程，PID（port ID）是进程的唯一标识符，就像人的身份证号。一个应用程序下的多个进程是树形结构，PID最小的数是根节点。在某一个进程上鼠标右键有一个结束进程树的选项，如果你选择杀死根节

点的进程树，将会杀死所有子节点的进程。

名称	状态	PID	进程名称	7% CPU
Google Chrome (5)				1.2%
Google Chrome		8872	chrome.exe	1.1%
Google Chrome		9336	chrome.exe	0%
Google Chrome		8124	chrome.exe	0%
Google Chrome		2348	chrome.exe	0%
Google Chrome		16240	chrome.exe	0.1%

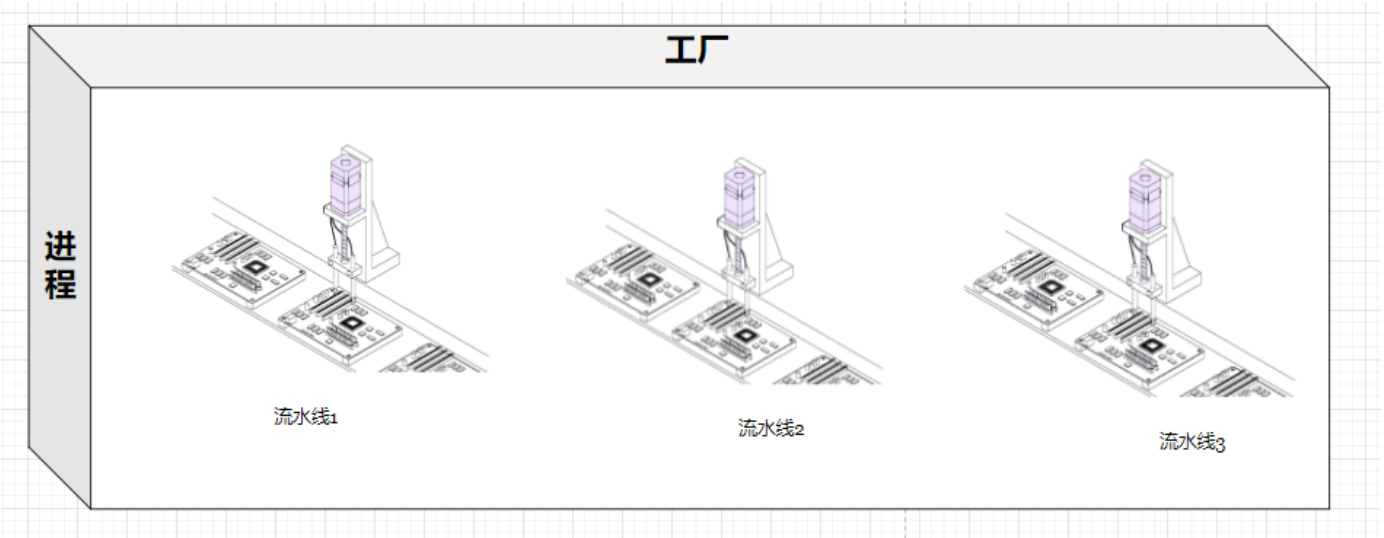
2. 线程（Thread）

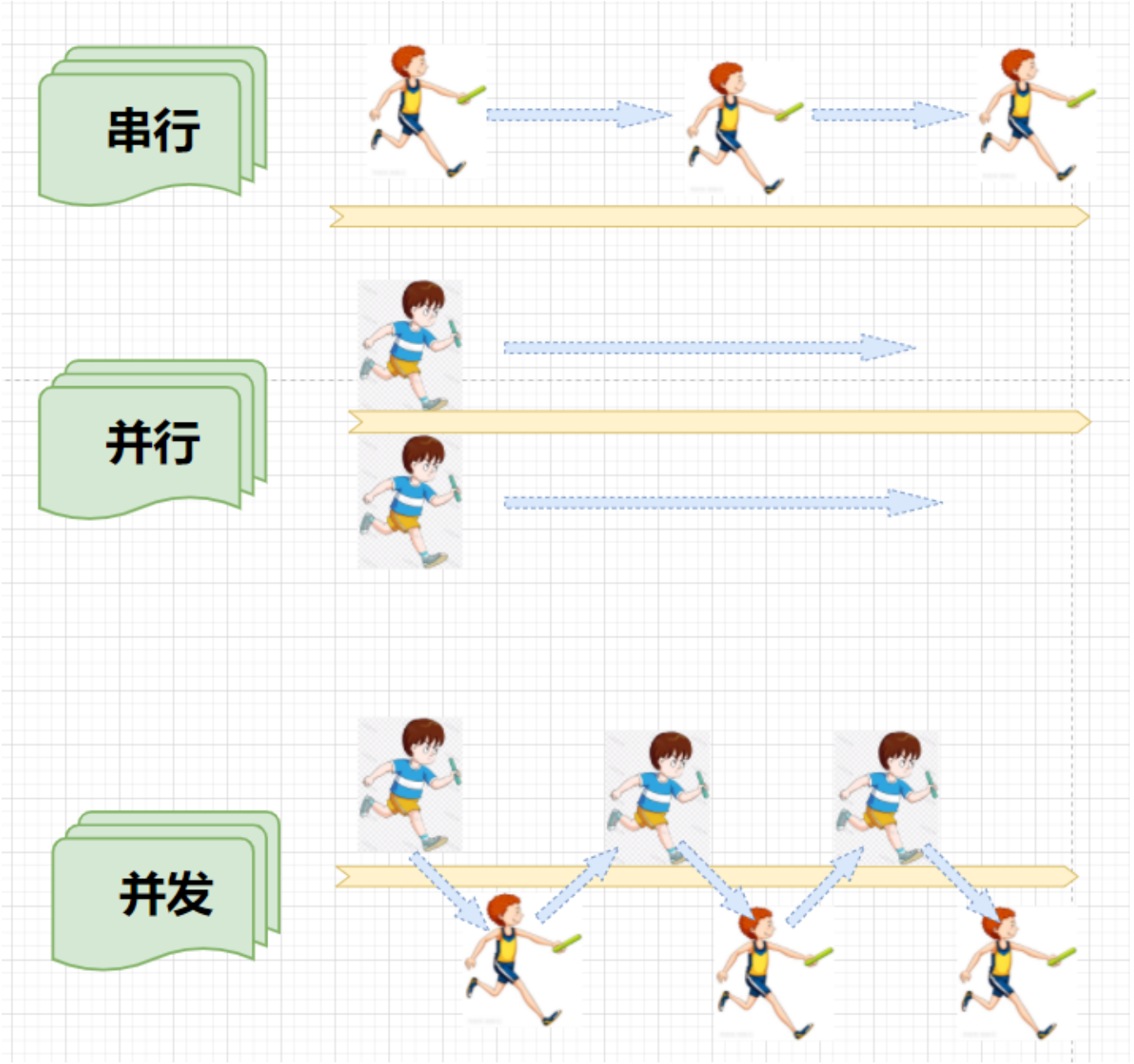
在早期的操作系统中并没有线程的概念，进程是拥有资源和独立运行的最小单位，也是程序执行的最小单位。后来，随着计算机的发展，对CPU的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了，于是就发明了线程。

线程概念：cpu能够进行调度、分配、执行、运算的最小的基本单位，是程序执行中一个单一的顺序控制流程。一个进程可以有一个或多个线程，各个线程之间共享进程的内存空间。

1. 进程类似于工厂，是系统分配资源的基本单元，线程 类似于工厂中的工人，是cpu调度和执行工作的基本单元。
2. 一个进程由一个或多个线程组成。
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段，数据集，堆等)。
4. 调度和切换：线程上下文切换比进程上下文切换要快得多。

为了更快速的完成任务，或者某些场景需要同时做多件事情，就需要使用线程，因为线程可以"同时"执行任务。
这里希望先显示进程间的切换，再显示线程

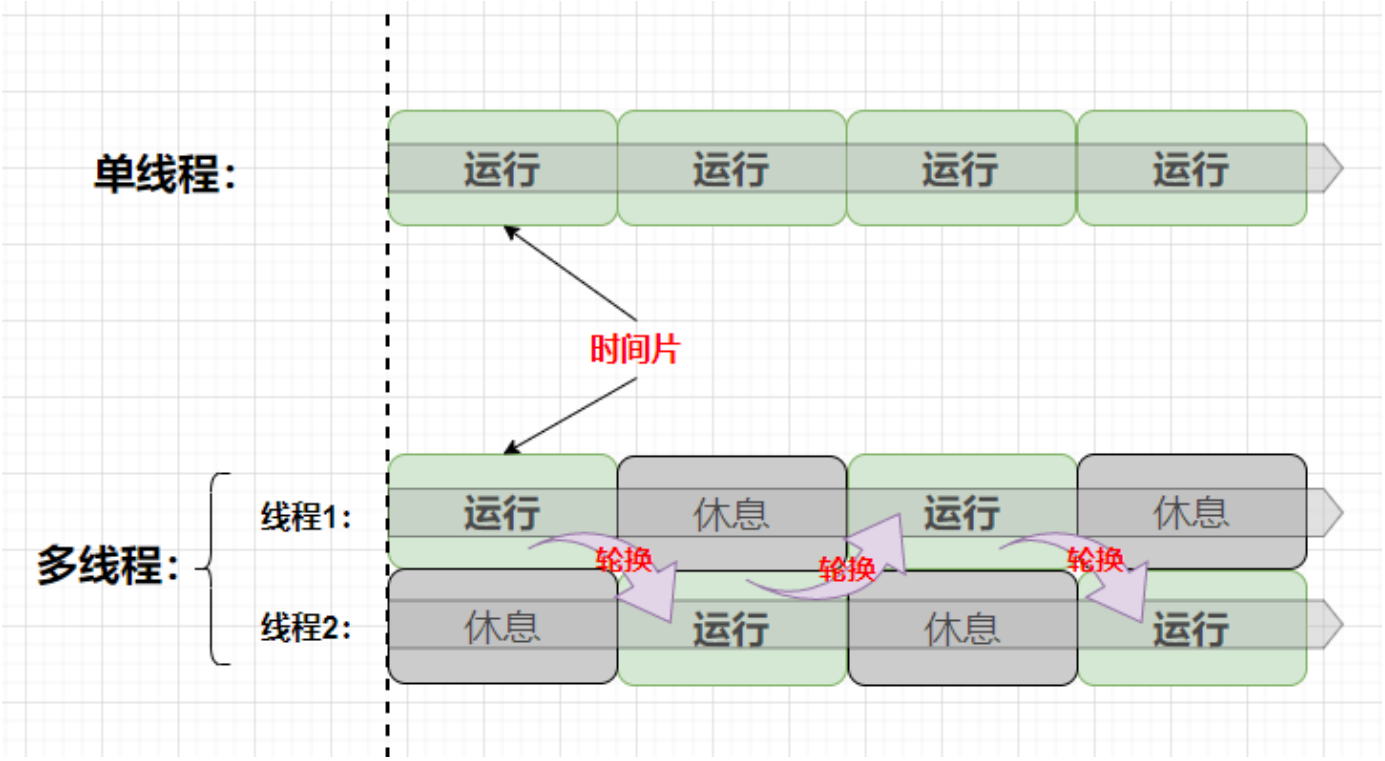




串行：按照顺序，一个执行完再执行下一个。
并行：同一个时刻，指同时执行。
并发：在同一个时间间隔内发生，指相同的时间间隔, 交替执行。

在单线程下，采用串行的方式执行。
大部分操作系统的任务调度是采用**轮换时间片**的抢占式调度方式，一个线程执行一小段时间后暂停休息并等待着被唤醒，下一个线程被唤醒并开始执行，每个线程交替轮流执行。线程执行的一小段时间叫做**时间片**。

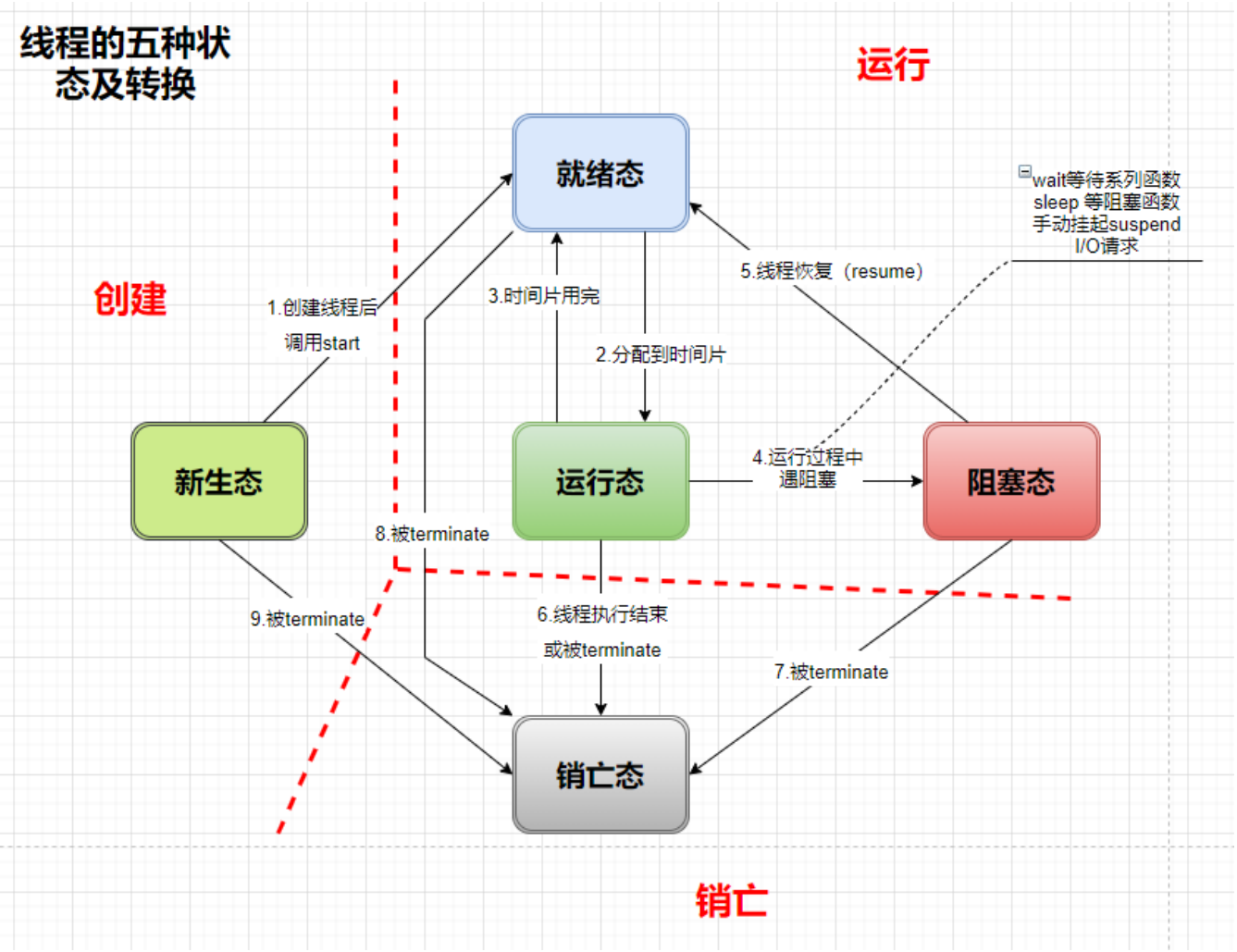
由于CPU的执行速度非常快，时间片非常短，在各个线程之间快速地切换，给人的感觉就是多个线程在“同时进行”，这就是常说的并发。



线程的状态：

- 新生态：创建出新的线程对象。
- 就绪态：创建出线程后，进入就绪态，会将线程添加到就绪队列中，等待分配到CPU时间片，就会进入运行状态。
- 运行态：运行态的线程如果时间片用完后，就会再次进入就绪状态，一般来说就绪态和运行态不需要人为参与，由操作系统进行调度，如果遇到sleep、wait、suspend、IO请求时 就会进入阻塞态。
- 阻塞态（挂起状态）：一个正在运行的线程在某些特殊情况下，如被人为挂起或执行耗时的I/O操作时，会让出CPU的使用权并暂时中止自己的执行，进入阻塞状态，处于阻塞状态的线程，就不能进入排队队列。只有当引起阻塞的原因被消除后，线程才可以转入就绪状态。当恢复线程，完成IO操作、等到资源，就会进入就绪状态。

- 销亡态：线程正常执行结束、因异常退出、被强制终止，该线程结束生命周期。



注意：

1. 线程必须通过就绪态分配到时间片才能进入运行状态，而不能直接进入运行状态。
2. 就绪状态无法进入阻塞状态。
3. 其他状态的线程可直接进入销亡态。

3. 创建线程

再window下使用API函数 `CreateThread`，Header：`#include <Windows.h>`

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,   //安全描述符，一般为NULL
    SIZE_T dwStackSize,                         //线程栈大小（字节）,, 如果传入0，使用默认大小（一般为1M）
    LPTHREAD_START_ROUTINE lpStartAddress,      //线程执行函数地址
    LPVOID lpParameter,                         //线程执行函数参数
    DWORD dwCreationFlags,                      //创建标识，0：线程立即运行，CREATE_SUSPENDED:线程挂起
    LPDWORD lpThreadId                          //线程ID,unsigned long* 类型，如果参数为NULL，id则不返回
```

```
);

CloseHandle(HANDLE);
```

注意：LPTHREAD_START_ROUTINE 是一个typedef，为函数指针类型的别名，函数原型如下：

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);  //不要省略 WINAPI
```

Qt 控制台项目下创建线程：

```
//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    if (p == nullptr) return 0;

    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        qDebug() << "赚了"<<i<<"钱" ;
        Sleep(1000);
    }
    return 0;
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    DWORD id = 0;
    int count = 10;
    HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,&count,0 /* 立即运行 */,&id);

    for(int i = 0;i<20;i++){
        Sleep(1000);
        qDebug()<<"睡觉: "<<i<<" 秒";
    }

    ::CloseHandle(handle);    //关闭句柄
    return a.exec();
}
```

> Sleep为什么不能精准延时。Sleep(100)实际上就是主动放弃当前时间片，并且告诉操作系统100ms内不占用时间片。执行Sleep函数后，线程进入到阻塞状态，100ms以后，进入就

> 绪状态，线程被添加到就绪队列中，等待时间片到来，进入运行状态。因为等待时间片这个时间不确定，所以说Sleep不能精准延时。

4. 内核对象

内核是操作系统提供底层服务的一个模块，而内核对象则是内核分配的一个内存块，它是一种数据结构，不同的内核对象具有不同的结构，负责维护对象相关的信息，少数成员如：安全描述符、使用计数等是所有内核对象都有的，其他多数都是不同的。

内核对象的所有者是操作系统而非进程，即内核对象的生命周期并不一定会随着创建该对象的进程的消亡而消亡，内核对象的存在时间可以比创建该对象的进程长，内核对象的回收是通过使用计数来实现的。使用计数是所有内核对象固有的属性，操作系统通过使用计数维护内核对象的生命周期。
操作系统内核知道当前有多少进程正在使用某个内核对象，内核对象被创建时，其使用计数为1，另一个进程访问该内核对象后，使用计数加1，当进程终止时，使用计数减1，手动关闭内核对象时，使用计数再减1，最终使用计数为0时，操作系统将销毁该内核对象。

如果结束使用内核对象，需要调用`CloseHandle()`函数，
`BOOL CloseHandle( HANDLE hObject );`
注意：并不是说调用这个函数，内核对象就被销毁了，而是使使用计数器减1。如果忘记`CloseHandle`那么再程序运行期间会发生内存泄露，直到进程结束，操作系统会回收所有的资源。

5. 线程挂起和恢复

在创建线程`CreateThread`函数 参数`DWORD dwCreationFlags`，创建标识，`0`：线程立即运行，`CREATE_SUSPENDED`：线程挂起。如果指定线程挂起（束之高阁），那么线程函数就不会被立即执行，直到我们恢复（Resume）线程。

```
1   HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,&count,CREATE_SUSPENDED/*线程挂起*/, &id);
```

输出结果，只睡觉了

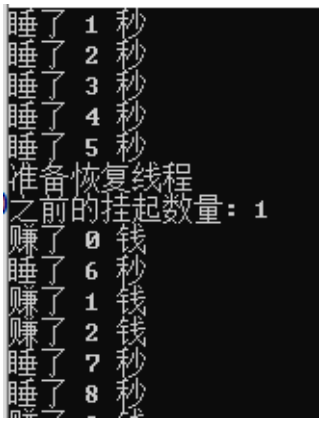
恢复线程使用`ResumeThread`函数，注意：参数中的句柄必须具有`THREAD_SUSPEND_RESUME`权限，才能恢复

```
DWORD WINAPI ResumeThread(HANDLE hThread);
```

如果恢复成功，返回的该线程之前的挂起的数量（使用内核中的挂起计数器），如果失败返回-1。

线程挂起和恢复之间 存在一个 **挂起计数器**，按照计数来计算的，所以挂起几次就恢复几次。创建线程 CREATE_SUSPENDED 挂起计数器加1。

```
...

HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,&count,CREATE_SUSPENDED,&id);

for(int i = 0;i<20;i++){
    Sleep(1000);
    qDebug() << "睡了" << i << "秒" ;
    if(i==5){
        qDebug()<<"准备恢复线程";
        DWORD preSusCount = ResumeThread(handle);
        qDebug()<<"之前的挂起数量:"<<preSusCount;
    }

}

...
```

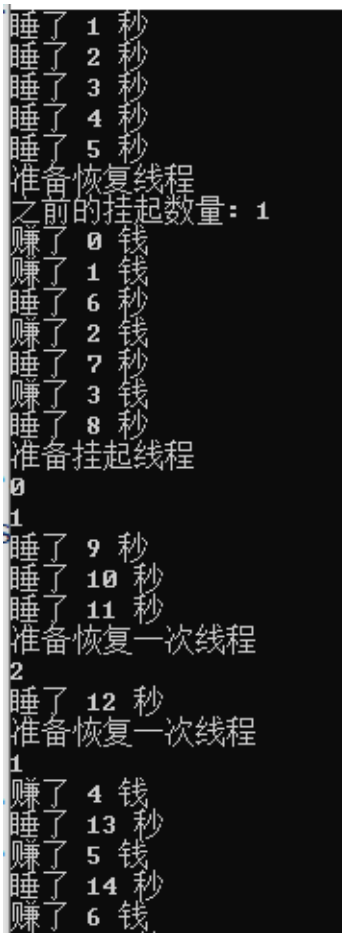


手动挂起线程的函数，句柄必须具有 THREAD_SUSPEND_RESUME 权限才能挂起成功。

```
DWORD WINAPI SuspendThread(_In_  HANDLE hThread);
```

如果函数成功，返回的是线程之前挂起的数量，否则返回-1。

```
...

HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,&count,CREATE_SUSPENDED,&id);

for(int i = 0;i<20;i++){
    ::Sleep(1000);
    qDebug() << "睡了" << i << "秒" ;
    if(i==5){
        qDebug()<<"准备恢复线程";
        DWORD preSusCount = ::ResumeThread(handle);
        qDebug()<<"之前的挂起数量:"<<preSusCount;
    }
    if(i==8){
        qDebug()<<"准备挂起线程";
        qDebug()<<::SuspendThread(handle);
        qDebug()<<::SuspendThread(handle);
    }
    if(i==11){
        qDebug()<<"准备恢复一次线程";
        qDebug()<<::ResumeThread(handle);
    }
    if(i==12){
        qDebug()<<"准备恢复一次线程";
        qDebug()<<::ResumeThread(handle);
    }
}

...
```



从输出结果看，确实是挂起几次就得恢复几次。

6. 线程退出

1. 正常退出，如顺序执行完代码，或有限次的循环，执行完毕直接自己主动退出。
   退出时，记得 `CloseHandle()`。
2. 全局变量（双变量）
   有些时候，创建的线程无法正常退出，可以设置退出标志，最好设置两个标志，可以让双方友好和谐的退出。保证所有子线程都退出后，主线程在退出。

```
//增加标志变量
bool isQuit1 = false;
bool isQuit2 = false;

//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    while(!isQuit1){
        qDebug()<<"ThreadProc";
        Sleep(1000);
    }
    isQuit2 = true;    //告诉主线程，我即将退出
    return 0;
}
```

```
14
15  int main(int argc, char *argv[])
16  {
17      DWORD id = 0;
18      int count = 10;
19      HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,(void*)&count,0,
    &id);
20
21      int i = 0;
22      while (1) {
23          ::Sleep(1000);
24          ++i;
25          qDebug() << "睡了" << i << "秒" ;
26          if(i>=10){
27              qDebug()<<"睡醒了";
28              isQuit1 = true;  //告诉子线程，你该退出了
29              break;
30          }
31      }
32
33      while(!isQuit2){  //循环等待子线程退出
34          ::Sleep(1000);
35      }
36      qDebug()<<"子线程已退出";
37      ::CloseHandle(handle);  //关闭句柄，内核对象使用计数会减1
38
39     return 0;
40  }
```

3. 终止线程。
   上例中如果子线程因为某种原因，一直无法退出，将导致主线程一直等待，无法做其他的事情。所以等待应该是有时限的。
   函数：`WaitForSingleObject`

```
1  DWORD WINAPI WaitForSingleObject (
2      HANDLE hHandle,         //等待哪个线程(创建线程时返回的句柄)
3      DWORD dwMilliseconds    //等待多少毫秒，INFINITE: 一直等待
4  );
```

当线程函数正常退出了 或 等待超时，此函数分别返回 `WAIT_OBJECT_0` 或 `WAIT_TIMEOUT`。

```
1  //增加标志变量
2  bool isQuit1 = false;
```

```
3  //bool isQuit2 = false;
4
5  //WINAPI
6  DWORD WINAPI ThreadProc(LPVOID p) {
7      while(!isQuit1){
8          qDebug()<<"ThreadProc";
9          Sleep(1000);
10     }
11     //isQuit2 = true;
12     return 0;
13 }
14
15
16 int main(int argc, char *argv[])
17 {
18     QCoreApplication a(argc, argv);
19
20     DWORD id = 0;
21     int count = 10;
22     HANDLE handle = ::CreateThread(NULL,0, &ThreadProc,(void*)&count,0,
   &id);
23
24     int i = 0;
25     while (1) {
26         ::Sleep(1000);
27         ++i;
28         qDebug() << "睡了" << i << "秒" ;
29         if(i>=3){
30             qDebug()<<"睡醒了";
31             isQuit1 = true;
32             break;
33         }
34     }
35
36 //    while(!isQuit2){
37 //        //::Sleep(1000);
38 //    }
39
40     DWORD flag =  ::WaitForSingleObject(handle,3000);  //阻塞等待
41     if(flag == WAIT_OBJECT_0){
42         qDebug()<<"WAIT_OBJECT_0   子线程已退出";  //等待子线程退出了
43     }else if(flag == WAIT_TIMEOUT ){
44         qDebug()<<"WAIT_TIMEOUT 等待超时";
45     }
```

```
        ::CloseHandle(handle);  //关闭句柄，内核对象实用计数会减1
        handle = NULL;

        return a.exec();
    }
```

如果故意让子线程不退出，等待3秒之后就会超时。

```
//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    while(!isQuit1){
        qDebug()<<"ThreadProc";
        Sleep(1000);
    }
    //isQuit2 = true;
    while(1){
        qDebug()<<"Sleep";
        Sleep(1000);
    }
    return 0;
}
```

等待超时之后，主线程中已经CloseHandle了，但子线程仍然在运行，所以等待超时后应当强制终止线程。

```
//如果成功返回非0，失败返回0，具体的失败信息可以通过 GetLastError 函数获取
BOOL WINAPI TerminateThread(
  _Inout_  HANDLE hThread,    //线程句柄，终止哪个线程
  _In_     DWORD dwExitCode   //退出码
);
```

改为如下：

```
...

DWORD flag =  ::WaitForSingleObject(handle,3000);  //阻塞等待
if(flag == WAIT_OBJECT_0){
    qDebug()<<"WAIT_OBJECT_0   子线程已退出";  //等待子线程退出了
}else if(flag == WAIT_TIMEOUT ){
    qDebug()<<"WAIT_TIMEOUT 等待超时";
    ::TerminateThread(handle,-1);   //强制杀死,退出码为-1
}
::CloseHandle(handle);  //关闭句柄，内核对象实用计数会减1

...
```

注意：TerminateThread 是一个比较危险的方法，应当用在最后万不得已最极端的情况。如果目标线程正在使用关键段，关键段不会被释放。线程在堆区申请空间，不会被释放，可能会导致内存泄漏。

7. 线程并发

```
int count = 0;  //全局变量（多个线程共享的一个资源）
//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
```

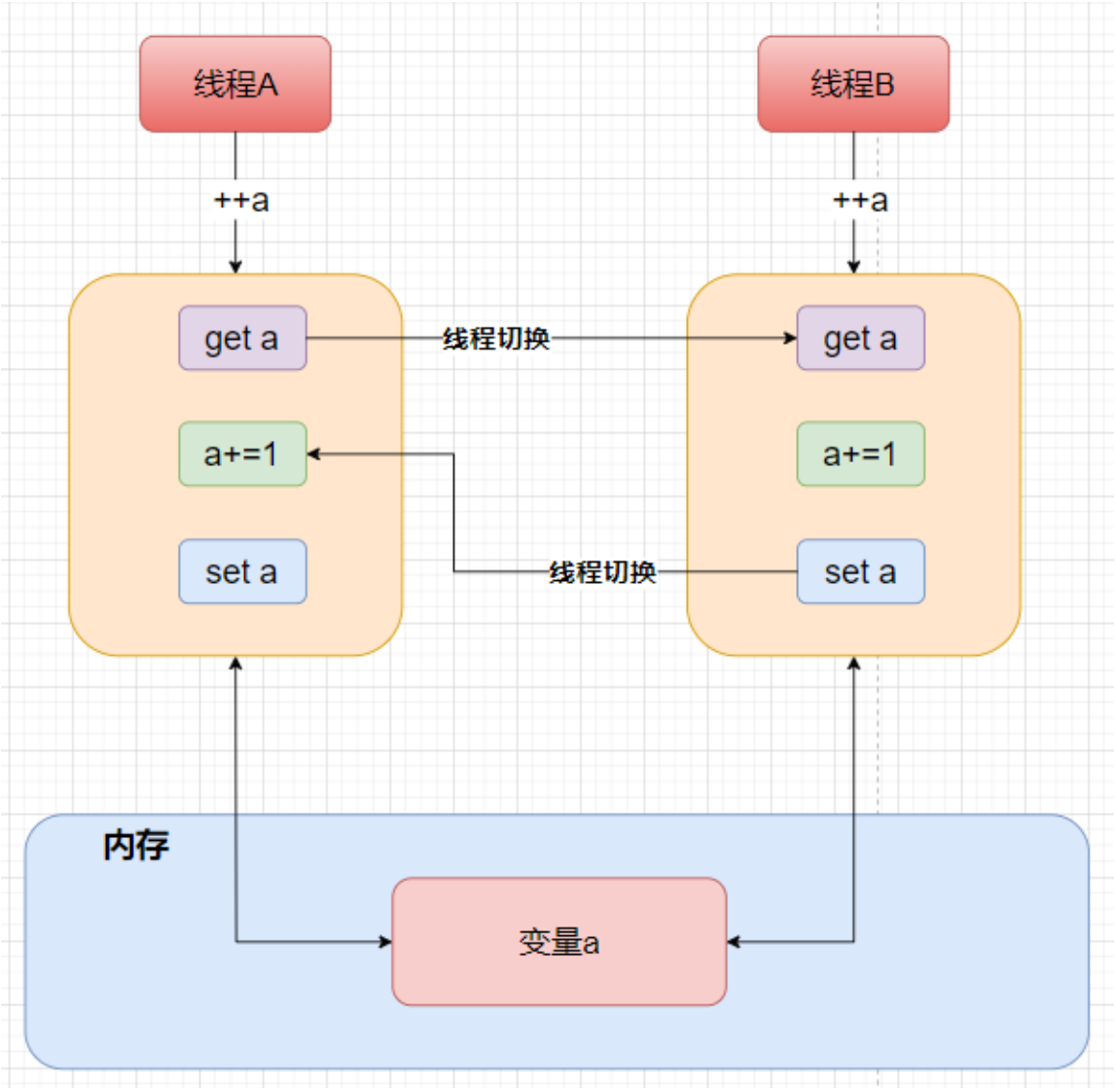
```
7           Sleep(10);  //一定要Sleep
8           count++;
9           qDebug() << count ;
10      }
11      return 0;
12  }
13
14  int main(int argc, char *argv[])
15  {
16      QCoreApplication a(argc, argv);
17
18      int count = 100;
19      HANDLE handle1 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
20      HANDLE handle2 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
21      HANDLE handle3 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
22
23      ::WaitForSingleObject(handle1,INFINITE);
24      if(handle1){
25          ::CloseHandle(handle1);
26          handle1=nullptr;
27      }
28
29      ::WaitForSingleObject(handle2,INFINITE);
30      if(handle2){
31          ::CloseHandle(handle2);
32          handle2=nullptr;
33      }
34
35      ::WaitForSingleObject(handle3,INFINITE);
36      if(handle3){
37          ::CloseHandle(handle3);
38          handle3=nullptr;
39      }
40
41      qDebug()<<"---------------"<<::count;
42
43     return a.exec();
44  }
```

最终的输出结果为

```
276
278
278
279
280
281
282
283
285
285
286
287
288
289
290
291
--------------- 291
```

并发问题：

多个线程同时操作同一个资源（内存空间、文件句柄、网络句柄），可能会导致的结果不一致的问题。发生的前提条件一定是多线程下共享资源。



8. 线程同步

线程同步，就是通过协调线程执行的顺序，避免多个线程同时操作同一个资源导致并发问题，使结果多次执行结果一致。
常见的线程同步方式：原子访问、关键段、事件、互斥量、条件变量、信号量。
上面提到的并发问题，解决方法有很多，重点学习 **锁**。

8.1 原子访问（Interlocked）

同一时刻，只允许一个线程访问一个变量。注意：他只是对一个变量保持原子 自增、自减操作，对于一个代码段来说并不适用。

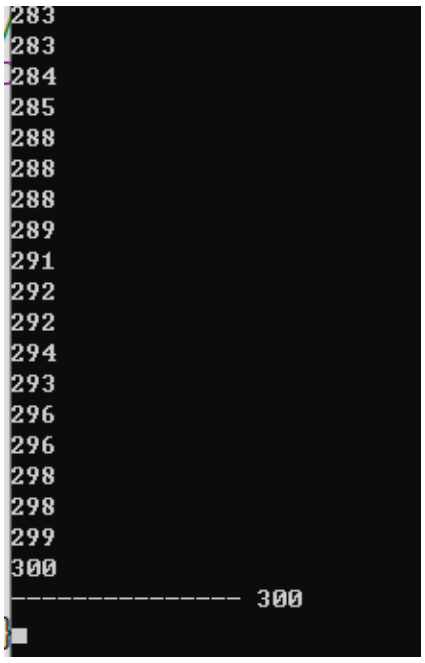
```
//header: include <windows.h>
LONG __cdecl InterlockedDecrement(_Inout_  LONG volatile *Addend);
LONG __cdecl InterlockedIncrement(_Inout_  LONG volatile *Addend);
```

参数为要原子自增、自减的变量的地址，返回值为增加或减少的结果。注意返回值一般与 `Addend` 增加的结果是一致的，但是不是绝对的。

使用原子访问，将上例修改如下：

```
int countt = 0;  //全局变量（多个线程共享的一个资源）

//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        Sleep(10);

        //countt++;
        ::InterlockedIncrement((LONG*)&countt);   //使用原子访问对变量进行自增操作

        qDebug() << countt ;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    int count = 100;
    HANDLE handle1 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
```

```
24      HANDLE handle2 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
25      HANDLE handle3 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
26
27      ::WaitForSingleObject(handle1,INFINITE);
28      if(handle1){
29          ::CloseHandle(handle1);
30          handle1=nullptr;
31      }
32
33      ::WaitForSingleObject(handle2,INFINITE);
34      if(handle2){
35          ::CloseHandle(handle2);
36          handle2=nullptr;
37      }
38
39      ::WaitForSingleObject(handle3,INFINITE);
40      if(handle3){
41          ::CloseHandle(handle3);
42          handle3=nullptr;
43      }
44
45      qDebug()<<"---------------"<<::countt;
46
47     return a.exec();
48  }
```



8.2 关键段（Critical_Section，也叫临界区）

8.2.1 关键段基本使用

结构体： CRITICAL_SECTION 和 4个函数：

```
//初始化关键段，即 初始化 CRITICAL_SECTION 的变量。
void WINAPI InitializeCriticalSection(_Out_  LPCRITICAL_SECTION lpCriti
calSection);
//进入关键段，加锁开始，其他的线程都被阻隔在外面，直到该线程离开关键段。
void WINAPI EnterCriticalSection(_Inout_     LPCRITICAL_SECTION lpCriti
calSection);
//离开关键段，其他线程可以进入了，
void WINAPI LeaveCriticalSection(_Inout_     LPCRITICAL_SECTION lpCriti
calSection);
//删除关键段，
void WINAPI DeleteCriticalSection(_Inout_    LPCRITICAL_SECTION lpCriti
calSection);
```

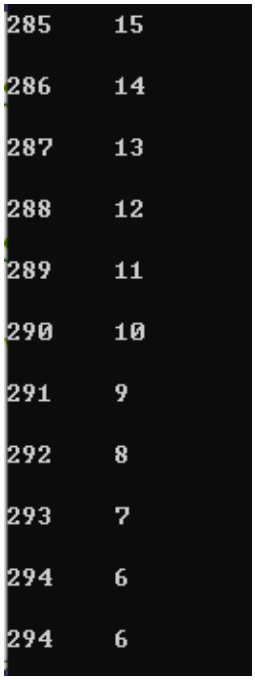
使用：

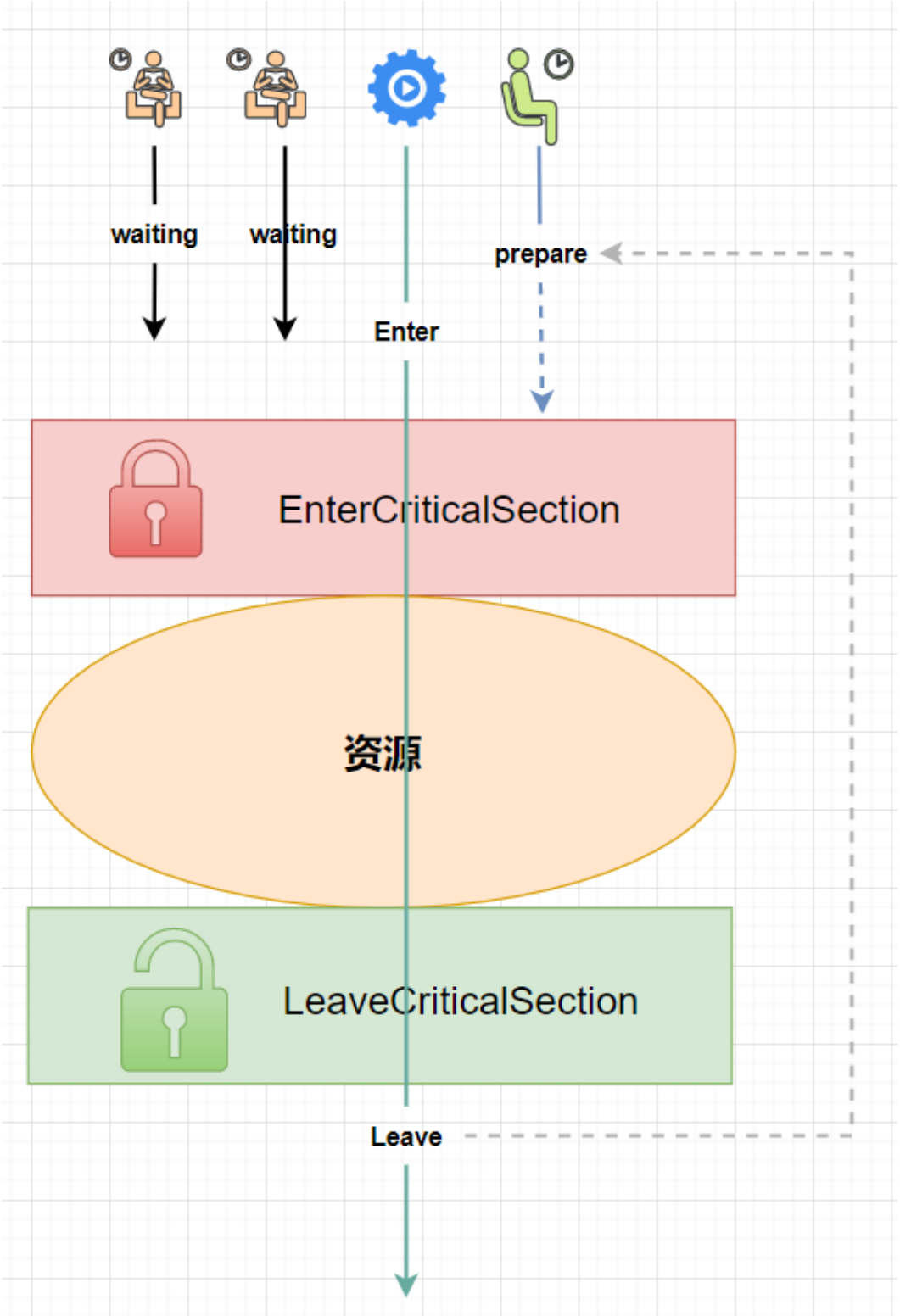
```
int countt1 = 0;  //全局变量（多个线程共享的一个资源）
int countt2 = 300;  //全局变量（多个线程共享的一个资源）

CRITICAL_SECTION cs1;

//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        Sleep(10);

        ::EnterCriticalSection(&cs1);
        countt1++;
        countt2--;
        ::LeaveCriticalSection(&cs1);

        qDebug("%d    %d \n",countt1,countt2);
    }
    return 0;
}

...

```

```
25  ::InitializeCriticalSection(&cs1);
26
27  //创建线程
28
29  ::DeleteCriticalSection(&cs1);
30
31  qDebug("%d    %d \n",countt1,countt2);
32
33  ...
```

```
289    11
290    10
291    9
292    8
293    7
294    6
295    5
296    4
297    3
298    2
299    1
300    0
300    0
```

如果不加关键段，结果如下（每次结果都不同）


285 15
286 14
287 13
288 12
289 11
290 10
291 9
292 8
293 7
294 6
294 6



8.2.2 封装关键段

```
//自定义类对关键段进行二次封装，
class CMyLock{
    CRITICAL_SECTION m_cs;
public:
    CMyLock(){
        ::InitializeCriticalSection(&m_cs);
    }
```

```
8       ~CMyLock(){
9           ::DeleteCriticalSection(&m_cs);
10      }
11  public:
12      void Lock(){
13          ::EnterCriticalSection(&m_cs);
14      }
15      void UnLock(){
16          ::LeaveCriticalSection(&m_cs);
17      }
18  };
19
20  //WINAPI
21  DWORD WINAPI ThreadProc(LPVOID p) {
22      static CMyLock lock;
23
24      if (p == nullptr) return 0;
25      int* pn = (int*)p;
26      for (int i = 0; i < *pn; i++){
27          Sleep(10);
28
29          lock.Lock();
30          countt1++;
31          countt2--;
32
33          lock.UnLock();
34          qDebug("%d    %d \n",countt1,countt2);
35      }
36      return 0;
37  }
```

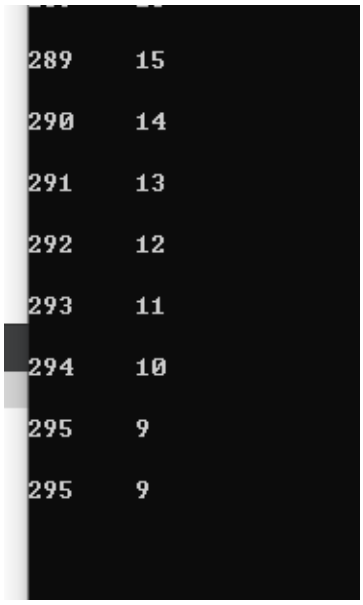
```
287    13
288    12
289    11
290    10
291    9
292    8
293    7
294    6
295    5
296    4
298    2
297    3
299    1
300    0
300    0
```

要保证线程同步，必须使用的是同一把锁。

```cpp
//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    static CMyLock lock;

    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        Sleep(10);

        lock.Lock();
        countt1++;
        countt2--;

        qDebug("%d    %d \n",countt1,countt2);
        lock.UnLock();
    }
    return 0;
}

DWORD WINAPI ThreadProc2(LPVOID p) {
    static CMyLock lock;
```

```
22
23      if (p == nullptr) return 0;
24      int* pn = (int*)p;
25      for (int i = 0; i < *pn; i++){
26          Sleep(10);
27
28          lock.Lock();
29          countt1++;
30          countt2--;
31
32          qDebug("%d    %d \n",countt1,countt2);
33          lock.UnLock();
34      }
35      return 0;
36  }
37
38  DWORD WINAPI ThreadProc3(LPVOID p) {
39      static CMyLock lock;
40
41      if (p == nullptr) return 0;
42      int* pn = (int*)p;
43      for (int i = 0; i < *pn; i++){
44          Sleep(10);
45
46          lock.Lock();
47          countt1++;
48          countt2--;
49
50          qDebug("%d    %d \n",countt1,countt2);
51          lock.UnLock();
52      }
53      return 0;
54  }
55
56  HANDLE handle1 = ::CreateThread(NULL,0, &ThreadProc,&count,0,NULL);
57  HANDLE handle2 = ::CreateThread(NULL,0, &ThreadProc2,&count,0,NULL);
58  HANDLE handle3 = ::CreateThread(NULL,0, &ThreadProc3,&count,0,NULL);
```

结果如下：

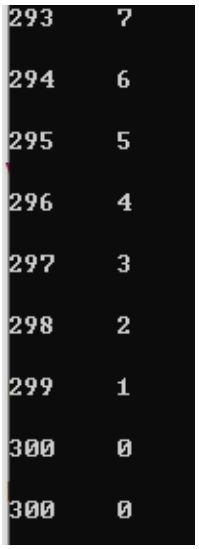


上例中每个线程函数，都用自己的一把锁，并不能保证所有的线程函数进行同步，
将 锁提升到全局。

```
static CMyLock lock;
//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        Sleep(10);

        lock.Lock();
        countt1++;
        countt2--;

        qDebug("%d    %d \n",countt1,countt2);
        lock.UnLock();
    }
    return 0;
}

DWORD WINAPI ThreadProc2(LPVOID p) {
    //static CMyLock lock;
    if (p == nullptr) return 0;
    int* pn = (int*)p;
    for (int i = 0; i < *pn; i++){
        Sleep(10);
        lock.Lock();
        countt1++;
        countt2--;
```

```
28          qDebug("%d    %d \n",countt1,countt2);
29          lock.UnLock();
30      }
31      return 0;
32  }
33
34  DWORD WINAPI ThreadProc3(LPVOID p) {
35      //static CMyLock lock;
36      if (p == nullptr) return 0;
37      int* pn = (int*)p;
38      for (int i = 0; i < *pn; i++){
39          Sleep(10);
40          lock.Lock();
41          countt1++;
42          countt2--;
43          qDebug("%d    %d \n",countt1,countt2);
44          lock.UnLock();
45      }
46      return 0;
47  }
```

结果正常：



8.2.3 优化单例

```
1   //单例-懒汉式
2   class CSingleton{
3   private:
4       static CSingleton* m_pSin;
5       CSingleton(){}
6       ~CSingleton(){};
7       CSingleton(const CSingleton &) = delete;
```

```
public:
    static CSingleton* GetSingleton(){
        if(!m_pSin){
            m_pSin = new CSingleton;
        }
        return m_pSin;
    }
};
CSingleton* CSingleton::m_pSin = nullptr;


//WINAPI
DWORD WINAPI ThreadProc(LPVOID p) {
    Sleep(200);
    CSingleton* pS = CSingleton::GetSingleton();
    qDebug()<<pS;
    return 0;
}

for(int i=0;i<50;i++){
    ::CreateThread(NULL,0, &ThreadProc,0,0,NULL);
}
```

结果不尽相同：

```
0x8e1a00
0x8e1a00
0x8e1a00
0x8e04b0
0x8e04b0
0x8e05a8
0x8e05a8
0x8e04b0
0x8e1a00
0x8e05a8
0x8e1a00
0x8e04b0
0x8e04b0
0x8e04b0
0x8e04b0
0x8e04b0
```

在单例的GetSingleton 函数中加关键段 锁住代码。

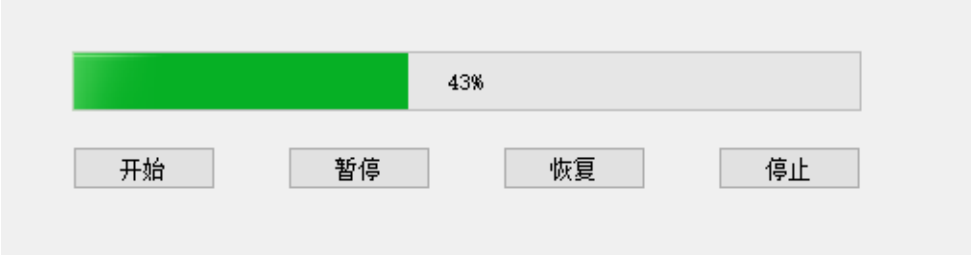
```
static CSingleton* GetSingleton(){
    if(!m_pSin){  // 先判断一下，把不需要加锁的情况排出去
        m_lock.Lock();
        if(!m_pSin){
            m_pSin = new CSingleton;
```

```
        }
        m_lock.UnLock();
    }

    return m_pSin;
}
```

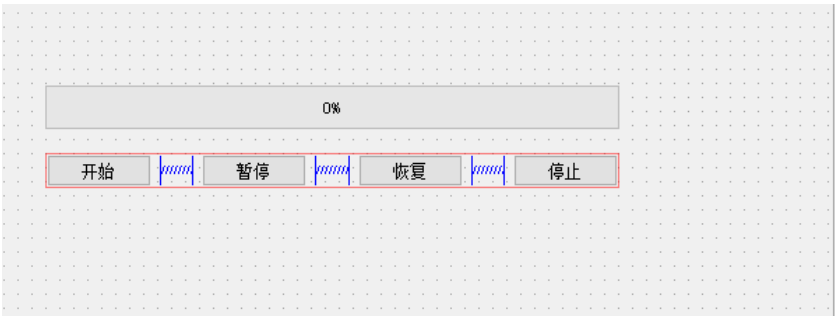
8.3 Qt下的多线程

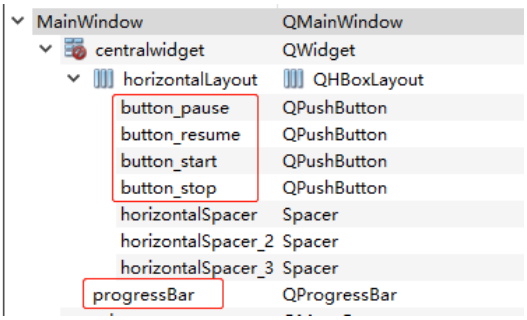
8.3.1 多线程与进度条

四个按钮分别控制进度条的状态。



设计界面，定义组件对象。






```
//MainWindow.h
class MainWindow : public QMainWindow
{
    ...

public:  //改为公共的
    Ui::MainWindow *ui;

    HANDLE m_hThread;
    bool m_isQuit ;
    bool m_isPause;
private slots:  //四个按钮对应的槽函数
    void on_button_start_clicked();
    void on_button_pause_clicked();
    void on_button_resume_clicked();

```

```
    void on_button_stop_clicked();
};
```

```
//MainWindow.cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    m_hThread = nullptr;
    m_isQuit = false;
    m_isPause = false;
}

MainWindow::~MainWindow()
{
    delete ui;

    if(m_hThread){
        ::CloseHandle(m_hThread);  //关闭句柄
        m_hThread = nullptr;
    }
}
//线程函数
DWORD WINAPI  ThreadProc(LPVOID lp){
    if(!lp) {
        qDebug()<<"线程参数为空";
        return 0;
    }

    MainWindow* pWin = (MainWindow*)lp;

    //每次创建时都要重置标记，否则while可能就直接退出了
    pWin->m_isQuit = false;
    pWin->m_isPause = false;

    int i = 0;

    qDebug()<<"ThreadProc begin";
    while(!pWin->m_isQuit){
        ::Sleep(20);

        if(pWin->m_isPause){    //如果是暂停，i 不能继续加，也不能继续设置
```

```
进度条的值
            continue;
        }

        pWin->ui->progressBar->setValue(i);  //在线程中设置 进度条的值
        i = ++i%101;
    }
    qDebug()<<"run 线程退出";
    return 0;
}

void MainWindow::on_button_start_clicked()
{
    if(!m_hThread)//创建线程
        m_hThread = ::CreateThread(nullptr,0,&ThreadProc,(void*)this,0,
nullptr);
}

void MainWindow::on_button_pause_clicked()
{
    //设置暂停标记为true
    m_isPause = true;
}

void MainWindow::on_button_resume_clicked()
{
    //暂停标记设置为false
    m_isPause = false;
}

void MainWindow::on_button_stop_clicked()
{
    //设置退出标记为true，线程退出
    m_isQuit = true;
    if(m_hThread){
        ::CloseHandle(m_hThread);  //关闭句柄
        m_hThread = nullptr;
    }
}
```

点击【开始】按钮，程序崩溃：

```
17:28:29: Starting E:\workspace\QT\Test6\build-test6-Desktop_Qt_5_12_11_MinGW_32_bit-Debug\debug\test6.exe ...
ThreadProc begin
ASSERT failure in QCoreApplication::sendEvent: "Cannot send events to objects owned by a different thread. Current thread 0x0x25d59608.
Receiver 'MainWindow' (of type 'MainWindow') was created in thread 0x0x1e077eb0", file kernel\qcoreapplication.cpp, line 578
17:28:51: 程序异常结束。
17:28:51: The process was ended forcefully.
17:28:51: E:\workspace\QT\Test6\build-test6-Desktop_Qt_5_12_11_MinGW_32_bit-Debug\debug\test6.exe crashed.
```

`ASSERT failure in QCoreApplication::sendEvent: "Cannot send events to objects owned by a different thread. Current thread 0x0x25d59608. Receiver 'MainWindow' (of type 'MainWindow') was created in thread 0x0x1e077eb0", file kernel\qcoreapplication.cpp, line 578`

报错原因：在另一个线程中设置了当前组件的值时，使用sendEvent 发送事件，跨线程报错。

> postEvent: 可以给别的线程发送事件。事件会在目的对象所属的线程中运行。这是一个异步接口。
> sendEvent: 仅用于同一个线程之间发送事件。目的对象必须与当前线程一样。这是一个同步接口。假如发送给属于另一个线程的对象，会报错：ASSERT failure in QCoreApplication::sendEvent: "Cannot send events to objects owned by a different thread. Current thread a51f48. Receiver '' (of type 'MyObject') was created in thread a3bf18", file kernel\qcoreapplication.cpp, line 539

针对于此种情况（QObject::connect），手动设置信号与槽进行连接，注意连接类型一定不能为 `DirectConnection`，可以选择 `AutoConnection` 或 `QueuedConnection`。

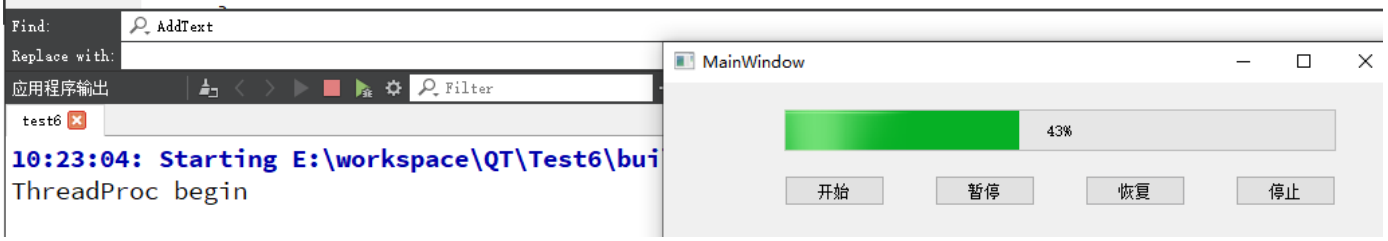
进度条报错，是否可以使用信号和槽解决呢?

```
//头文件 增加自定义信号与槽
signals:
    void sign_sendValue(int);
private slots:
    void slot_SetValue(int);

//构造中绑定连接
QObject::connect(this,SIGNAL(sign_sendValue(int)),
                 this,SLOT(slot_SetValue(int)),
                 Qt::QueuedConnection);   //一定不能使用 `DirectConnection`


void MainWindow::slot_SetValue(int v){
    this->ui->progressBar->setValue(v);
}

//线程函数
DWORD WINAPI  ThreadProc(LPVOID lp){
    if(!lp) {
```

```
20          qDebug()<<"线程参数为空";
21          return 0;
22      }
23
24      MainWindow* pWin = (MainWindow*)lp;
25
26      //每次创建时都要重置标记，否则while可能就直接退出了
27      pWin->m_isQuit = false;
28      pWin->m_isPause = false;
29
30      int i = 0;
31      qDebug()<<"ThreadProc begin";
32      while(!pWin->m_isQuit){
33          ::Sleep(20);
34          if(pWin->m_isPause){
35              continue;
36          }
37
38          i = i%101;
39          //pWin->ui->progressBar->setValue(i);
40          emit pWin->sign_sendValue(i);  //发射信号
41
42          i++;
43      }
44
45      qDebug()<<"run 线程退出";
46      return 0;
47  }
```

不再报错：

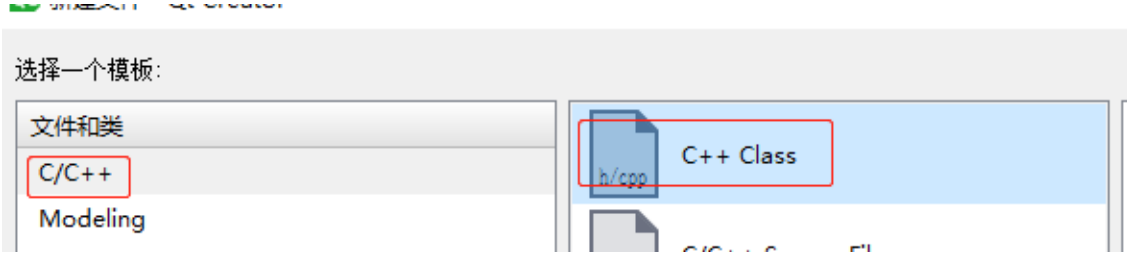
```
88      pWin->m_isQuit = false;
89      pWin->m_isPause = false;
90
91      int i = 0;
92      qDebug()<<"ThreadProc begin";
93      while(!pWin->m_isQuit){
94          ::Sleep(20);
95          if(pWin->m_isPause){
96              continue;
97          }
98
99          i = i%101;
100         //pWin->ui->progressBar->setValue(i);
101         emit pWin->sign_sendValue(i);
102
103         i++;
104
```

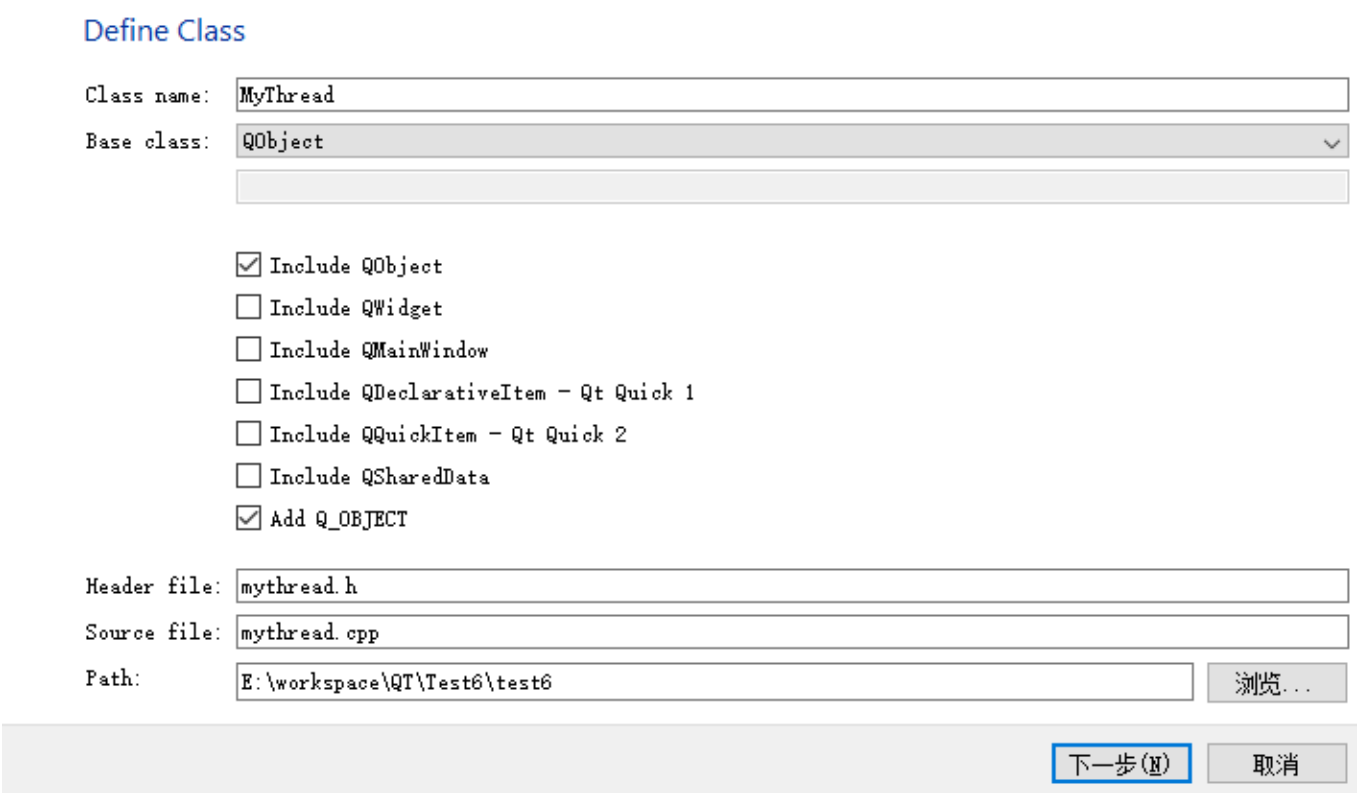


8.3.2 Qt-QThread

在Qt中封装了创建线程的类：QThread，一般情况下，手动添加一个类继承QThread，这样既能继承QThread的功能，又能扩展自己的功能。

添加新文件





Qt **线程 + 关键段**：为何要使用关键段，原来的只是靠一个暂停标志位，while 一直在空跑，也会耗费一点cpu。好一点的效果是让其等待，直到恢复。

```
#ifndef MYTHREAD2_H
#define MYTHREAD2_H

#include <QThread>
#include<windows.h>
class MyThread2 : public QThread   //改成继承QThread
{
    Q_OBJECT
public:
    explicit MyThread2(QObject *parent = nullptr);
    ~MyThread2();
signals:
    void sign_SendValue(int);
public:
    virtual void run() override;    //重写  QThread  类的 run 方法
public:
```

```
    bool m_isQuit;   //标识线程是否退出
    bool m_isPause;  //标识线程是否暂停
    CRITICAL_SECTION cs;   //关键段，锁住代码阻塞线程
};
#endif // MYTHREAD2_H
```

```
MyThread2::MyThread2(QObject *parent) : QThread(parent)
{
    m_isQuit = false;
    m_isPause = false;

    ::InitializeCriticalSection(&cs);
}

MyThread2::~MyThread2(){
    ::DeleteCriticalSection(&cs);
}

void MyThread2::run() {
    qDebug()<<"id = "<<currentThreadId();   //获取当前线程ID

    m_isQuit = false;
    m_isPause = false;
    int i = 0;

    while(!m_isQuit){

        if(m_isPause){
            qDebug()<<"EnterCriticalSection";
            ::EnterCriticalSection(&cs);
            ::LeaveCriticalSection(&cs);
            qDebug()<<"LeaveCriticalSection";
        }

        i = i%101;
        emit sign_SendValue(i);
        i++;
        ::Sleep(20);
    }

    qDebug()<<"run 线程退出";

}
```

```
//MainWindow.h
    ...

    MyThread2 m_thread2;
private slots:
    void slot_SetValue(int);
```

注意： 启动线程需要调用系统函数 `start()` 而不是直接调用 `run()`。

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ...

    //连接线程中的信号 与 mainwindow 中的设置值的槽函数
    QObject::connect(&this->m_thread,SIGNAL(sign_SendValue(int)),this,SLOT(slot_SetValue(int )),Qt::AutoConnection);
}

void MainWindow::on_button_start_clicked()
{
//    if(!m_hThread)
//        m_hThread = ::CreateThread(nullptr,0,&ThreadProc,(void*)this,0,nullptr);

    m_thread2.start();   //调用start 函数启动一个线程
}

void MainWindow::on_button_pause_clicked()
{
    //m_isPause = true;

    ::EnterCriticalSection(&m_thread2.cs);
    m_thread2.m_isPause = true;
}

void MainWindow::on_button_resume_clicked()
{
    //m_isPause = false;

    m_thread2.m_isPause = false;
    ::LeaveCriticalSection(&m_thread2.cs);
```

```
33  }
34
35  void MainWindow::on_button_stop_clicked()
36  {
37      //m_isQuit = true;
38      m_thread2.m_isQuit = true;
39  }
40
41  void MainWindow::slot_SetValue(int v){
42      this->ui->progressBar->setValue(v);
43  }
```

```
24      while(!m_isQuitProgress){
25
26          if(m_isPause){
27              qDebug()<<"EnterCriticalSection";
28              ::EnterCriticalSection(&cs);
29              ::LeaveCriticalSection(&cs);
30              qDebug()<<"LeaveCriticalSection";
```