3 线程通信与同步

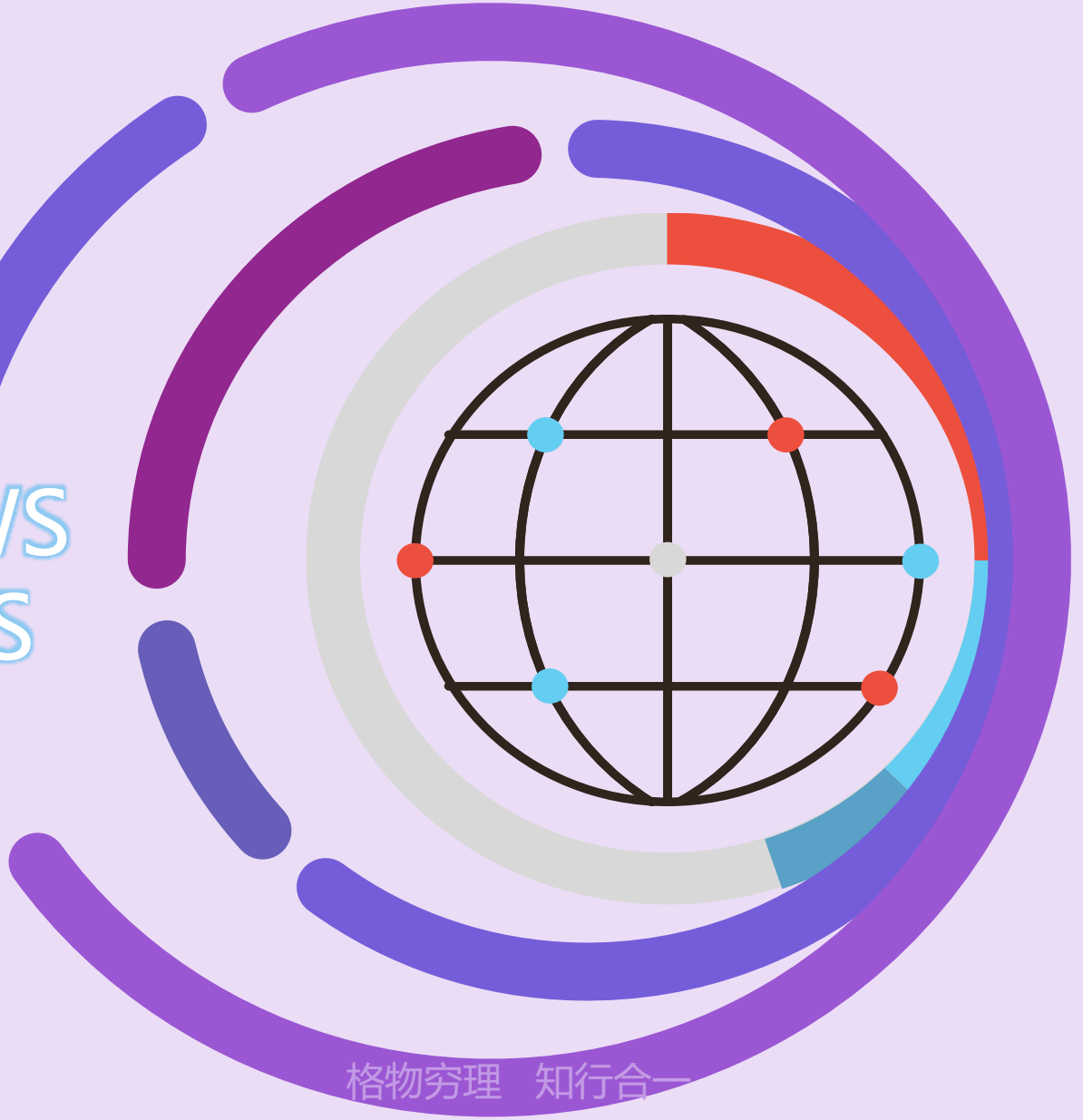
PRINCIPLE OF WINDOWS AND ITS APPLICATIONS

School of CS

Jicheng Hu

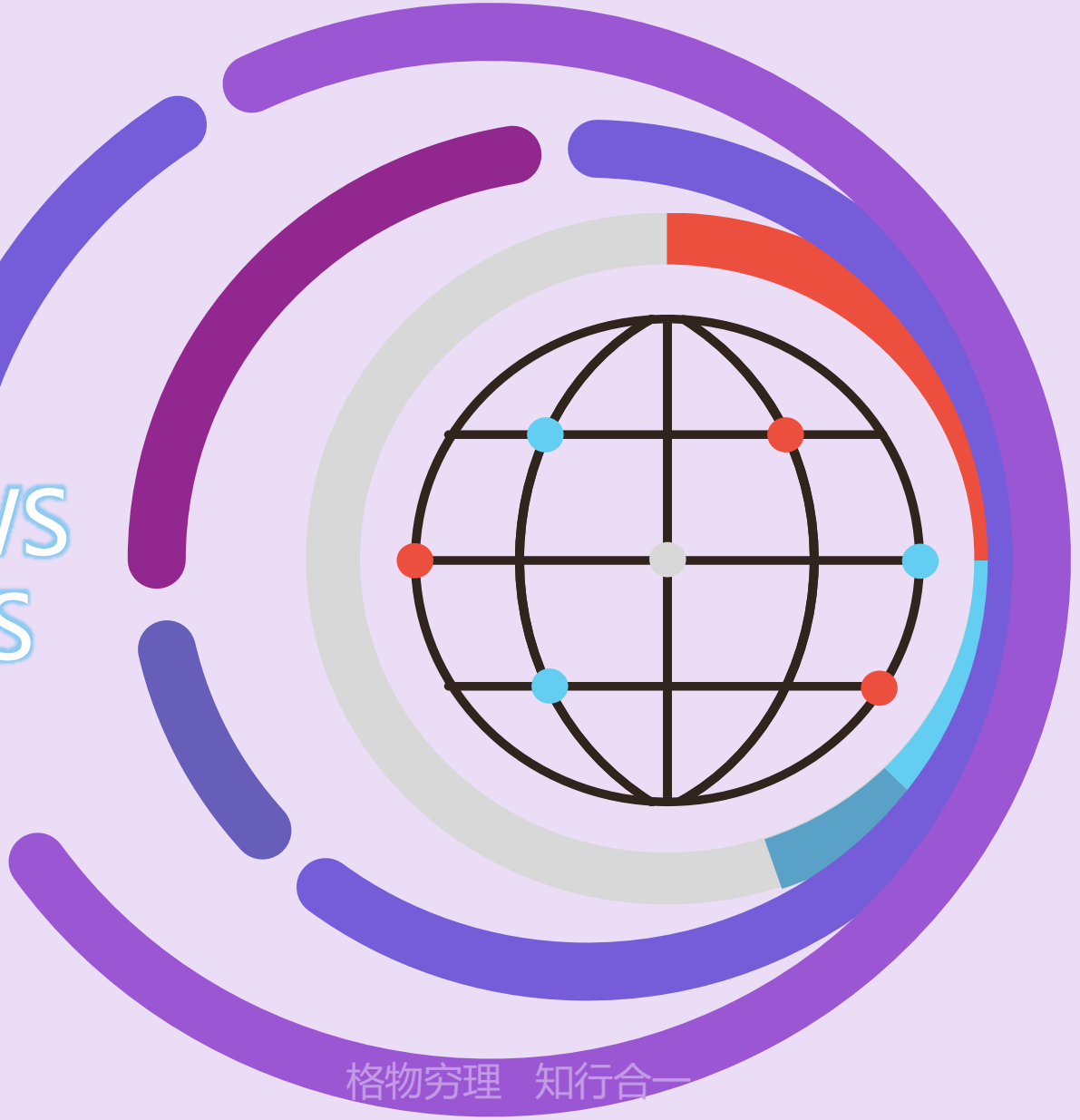
jicheng @ yahoo . com

https://gitee.com/wuhanuniversity/



3 线程通信与同步

PRINCIPLE OF WINDOWS AND ITS APPLICATIONS

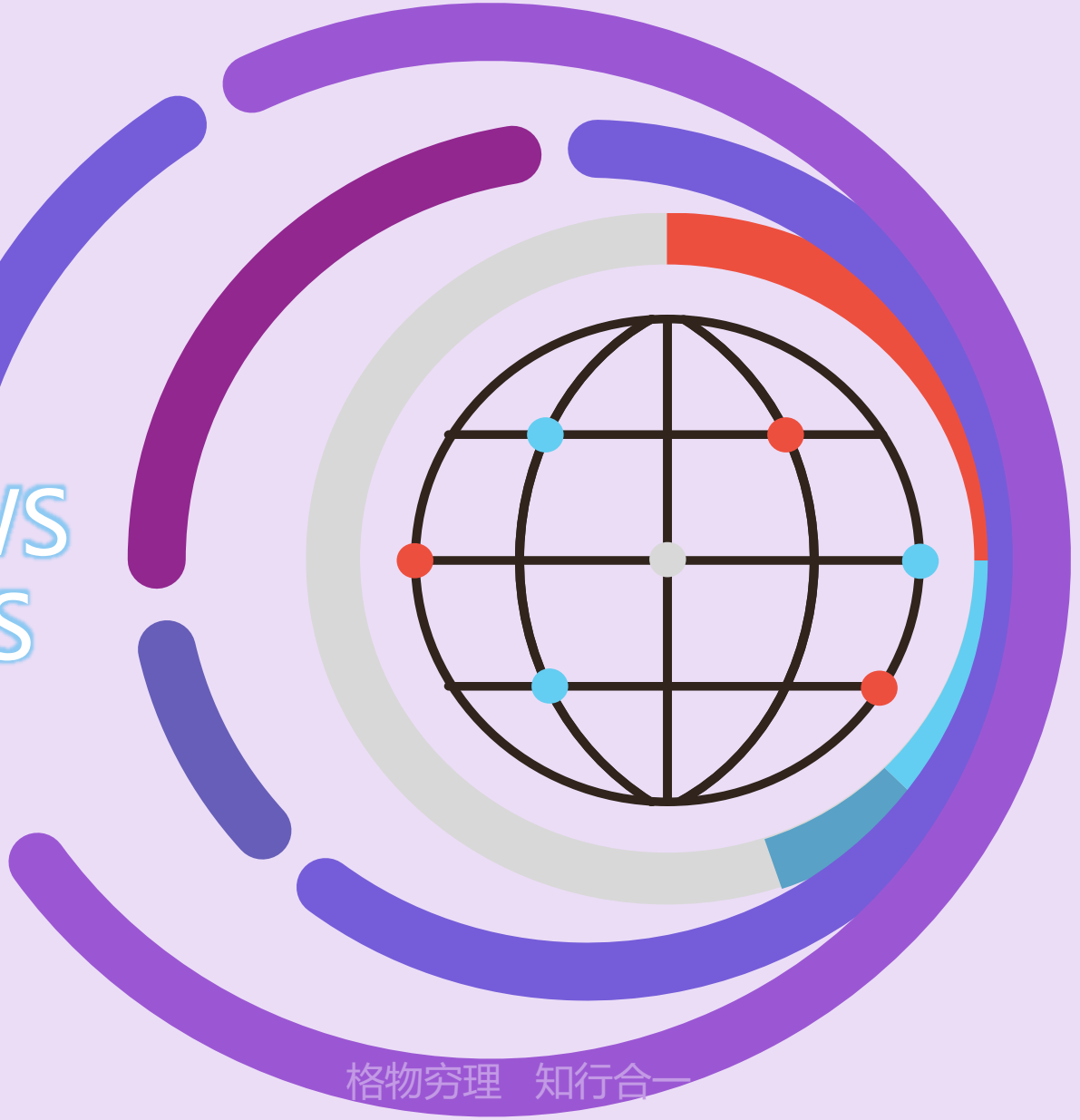


C++ 线程编程进阶参考阅读材料：

https://www.codeproject.com/Articles/1092727/Asynchronous-Multicast-Callbacks-with-Inter-Thread

格物穷理　知行合一

PRINCIPLE OF WINDOWS AND ITS APPLICATIONS



实验课时间：

周2 晚 6:00-9:30	计算机学院 B303
周3下午2:00-5:30	计算机学院 B302

格物穷理　知行合一

内容提要 -线程间通信与同步

3.1 线程及其创建过程

- 进程是计算机分配资源的单位，线程是运行调度单位
- 进程中的线程也具有线程控制块，包含内容有所属进程ID，创建和退出时间，线程启动地址等

线程创建过程

1. 在进程的地址空间中为线程创建用户态堆栈
2. 初始化线程硬件上下文
3. 创建线程对象
4. 通知内核系统为线程运行准备
5. 新创建线程handle和线程ID值返回到调用者
6. 线程进入调度准备执行

线程的生命期

工作线程的结束

- 线程正常结束
  - 自动消亡，OS清理
- 线程非正常结束，被KILL
  - OS 无法控制，造成系统损失或破坏
- 控制线程正常终止的方法
  - C# 低级事件对象ManualResetEvent
  - C++ 中 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) lpFunc, 0, 0, 0)的 lpFunc 正常退出后调用 CloseHandle (hThread) 来关闭线程对象

线程非正常结束的后果

- 内存无法回收 - 内存泄漏
- 文件缓冲没写入 - 文件被破坏
- 文件句柄未回收 - 被占用
- 共享资源的占用(网络端口，管道，DLL)

线程的创建与启动代码-c#

- 线程执行代码的编写

```
void workThread(){
}
```

- 设定函数名为线程入口

```
ThreadStart s = new ThreadStart(workThread);
```

- 线程委托对象(委托的实质是函数指针或叫函数地址)

```
Thread thread1=new Thread(s);
```

- 设定线程优先级等属性
- 线程启动　　thread1.Start();
- 线程参数传递　thread1.Start(paraObject);

线程的创建与启动代码-c#

- C#的System.Threading命名空间下的Thread类和ThreadStart类用于完成的线程创建和管理
- 使用Thread类创建线程时，只需要提供线程入口，线程入口告诉程序让这个线程做什么
  - 通过实例化一个Thread类的对象就可以创建一个线程
  - 创建新的Thread对象时，将创建新的托管线程
  - Thread类接收一个ThreadStart委托或ParameterizedThreadStart委托的构造函数，该委托包装了调用Start方法时由新线程调用的方法

```
// 创建无参数方法的托管线程
// 创建线程
Thread thread1=new Thread(new
ThreadStart(method));
// 启动线程
thread1.Start();

// 定义无参方法
static void method() {
  Console.WriteLine("这是无参的静态方法");
}
```

```
class ThreadTest
{
    public void MyThread()
    {
        Console.WriteLine("这是一个实例方法");
    }
}
ThreadTest test= new ThreadTest();
// 创建线程
Thread thread2 = new Thread(new ThreadStart
                        (test.MyThread()));
// 启动线程
thread2.Start();
```

线程的创建与启动代码-c#

- 还可以通过匿名委托或Lambda表达式来创建线程

```
// 通过匿名委托创建
Thread thread1 = new Thread(delegate() { Console.WriteLine("我是通过匿名委托创建的线程"); });
thread1.Start();

// 通过Lambda表达式创建
Thread thread2 = new Thread(() => Console.WriteLine("我是通过Lambda表达式创建的委托"));
thread2.Start();
```

线程的创建与启动代码-c#

- 还可以利用有参的委托ParameterizedThreadStart来创建线程

```
class Program
{
      static void Main(string[] args)
      {
         // 通过ParameterizedThreadStart创建线程
         Thread thread = new Thread (new ParameterizedThreadStart(Thread1));
         // 给方法传值
         thread.Start("这是一个有参数的委托");
         Console.ReadKey();
      }
     /// 创建有参的方法，方法里面的参数类型必须是Object类型
     static void Thread1(object obj)
      {
         Console.WriteLine(obj);
      }
}
```

线程的创建与启动代码-c#

- 还可以利用有参的委托ParameterizedThreadStart来创建线程

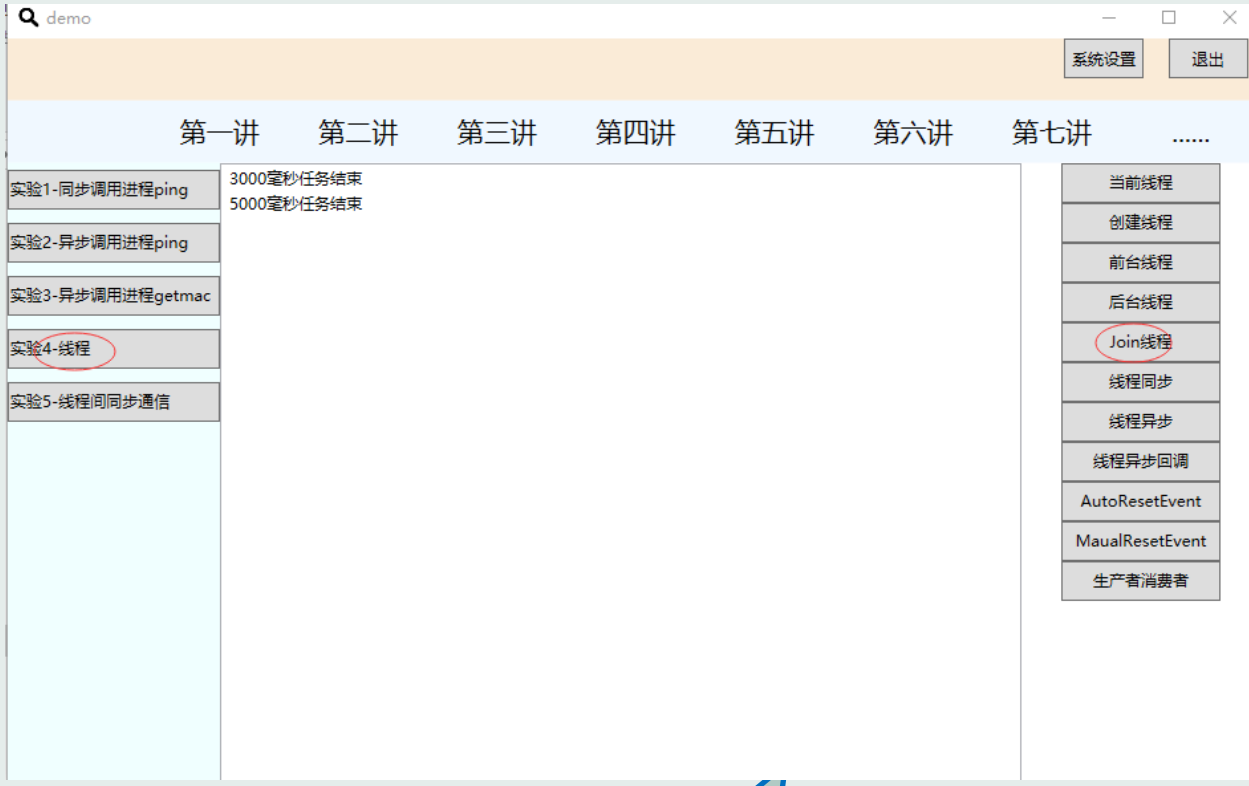
```
class Program
{
      static void Main(string[] args)
      {
         // 通过ParameterizedThreadStart创建线程
         Thread thread = new Thread (new ParameterizedThreadStart(Thread1));
         // 给方法传值
         thread.Start("这是一个有参数的委托");
         Console.ReadKey();
      }
     /// 创建有参的方法，方法里面的参数类型必须是Object类型
     static void Thread1(object obj)
      {
         Console.WriteLine(obj);
      }
}
```

线程的其它操作 - c# System.Threading.Thread的方法

方法名称	说明
Abort()	终止本线程。
GetDomain()	返回当前线程正在其中运行的当前域。
GetDomainId()	返回当前线程正在其中运行的当前域Id。
Interrupt()	中断处于 WaitSleepJoin 线程状态的线程。
Join()	已重载。 阻塞调用线程，直到某个线程终止时为止。
Resume()	继续运行已挂起的线程。
Start()	执行本线程。
Suspend()	挂起当前线程，如果当前线程已属于挂起状态则此不起作用
Sleep()	把正在运行的线程挂起一段时间。

Thread方法

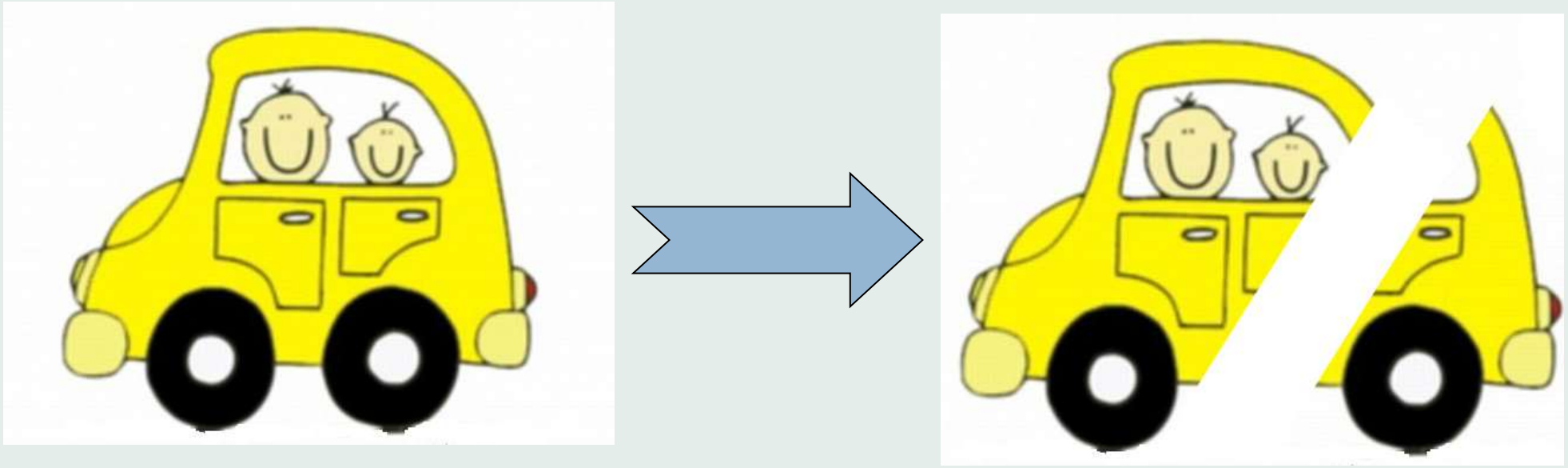
Thread.Sleep



Thread. Abort

Thread. Join

杀死正在运行的线程



丢失资源

线程的常用属性

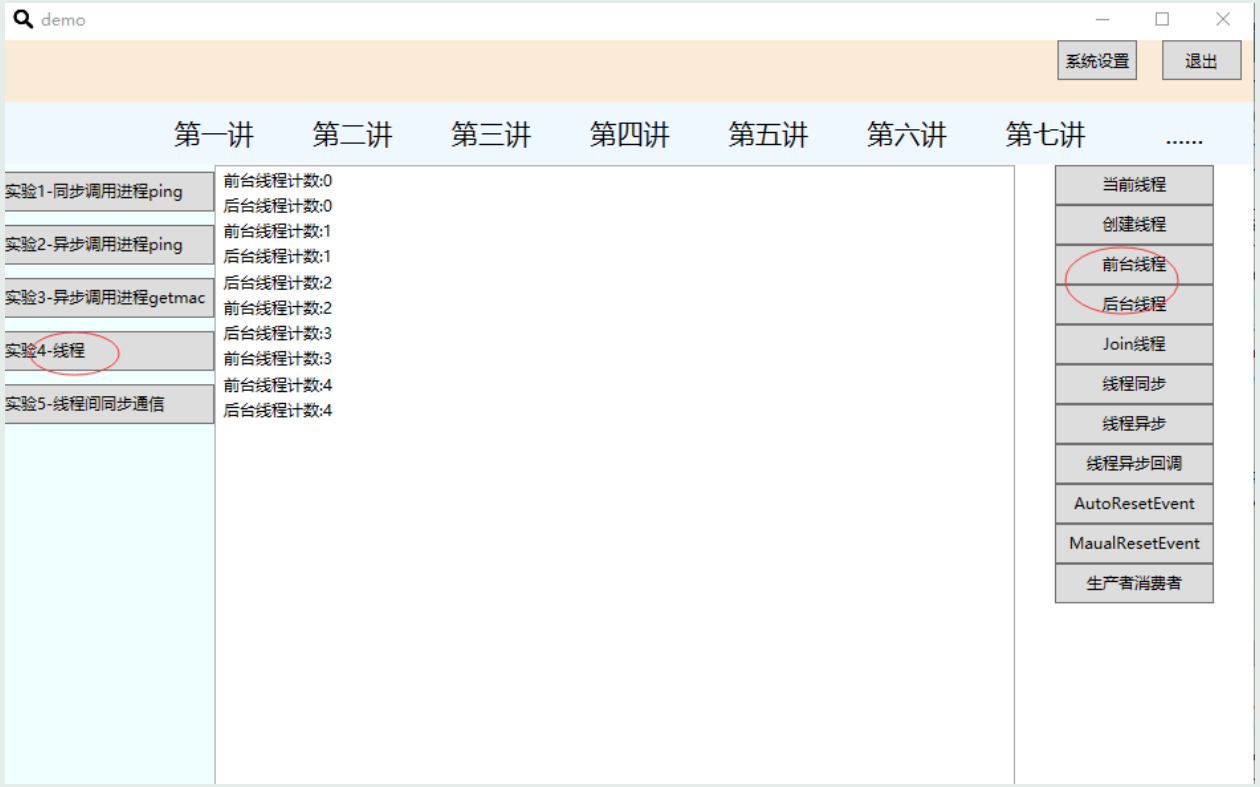
属性名称	说明
CurrentContext	获取线程正在其中执行的当前上下文。
CurrentThread	获取当前正在运行的线程。
ExecutionContext	获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。
IsAlive	获取一个值，该值指示当前线程的执行状态。
IsBackground	获取或设置一个值，该值指示某个线程是否为后台线程。
IsThreadPoolThread	获取一个值，该值指示线程是否属于托管线程池。
ManagedThreadId	获取当前托管线程的唯一标识符。
Name	获取或设置线程的名称。
Priority	获取或设置一个值，该值指示线程的调度优先级。
ThreadState	获取一个值，该值包含当前线程的状态。

前台线程与后台线程

- 前台线程：只有所有的前台线程都结束，应用程序才能结束。默认情况下创建的线程都是前台线程

- 后台线程：只要所有的前台线程结束，后台线程自动结束。
  - ✓ 通过Thread.IsBackground设置后台线程。
  - ✓ 且必须在调用Start方法之前设置线程的类型，否则一旦线程运行，将无法改变其类型

- 一般后台线程用于处理时间较短的任务，如在一个Web服务器中可以利用后台线程来处理客户端发过来的请求信息。
- 而前台线程一般用于处理需要长时间等待的任务，如在Web服务器中的监听客户端请求的程序，或是定时对某些系统资源进行扫描的程序



线程的优先级与线程调度

- windows中的线程按照优先级进行调度
- 具有最高优先权的线程一直被执行
- 相同优先级的线程 按时间片轮转执行，时间片在windows系统中通常20ms
- 当更高优先级的线程就绪时，高优先的线程会抢占执行低优先级的线程

成员名称	说明
Lowest	可以将 Thread 安排在具有任何其他优先级的线程之后。
BelowNormal	可以将 Thread 安排在具有 Normal 优先级的线程之后，在具有 Lowest 优先级的线程之前。
Normal	默认选择。可以将 Thread 安排在具有 AboveNormal 优先级的线程之后，在具有 BelowNormal 优先级的线程之前。
AboveNormal	可以将 Thread 安排在具有 Highest 优先级的线程之后，在具有 Normal 优先级的线程之前。
Highest	可以将 Thread 安排在具有任何其他优先级的线程之前。

线程状态

1. 初始化--线程处于创始中
2. 就绪--等待由CPU执行
3. 待命--只能由一个线程处于待命状态，离执行状态最近
4. 运行--在CPU的当前时间片内执行
5. 等待--线程同步需要等待
6. 接转--准备执行，但是它的内核堆栈不在内存，需要内存页面调入，调入后进入就绪状态
7. 终止--线程执行完

通过ThreadState可以检测线程是处于Unstarted、Sleeping、Running 等等状态，它比 IsAlive 属性能提供更多的特定信息

多线程

1、CPU运行速度太快，硬件处理速度跟不上，所以操作系统进行分时间片管理

- 从宏观角度来说是多线程并发的，因CPU速度太快，看起来是同时执行不同操作
- 从微观角度来讲，同一时刻通常只能有一个线程在一个核上处理

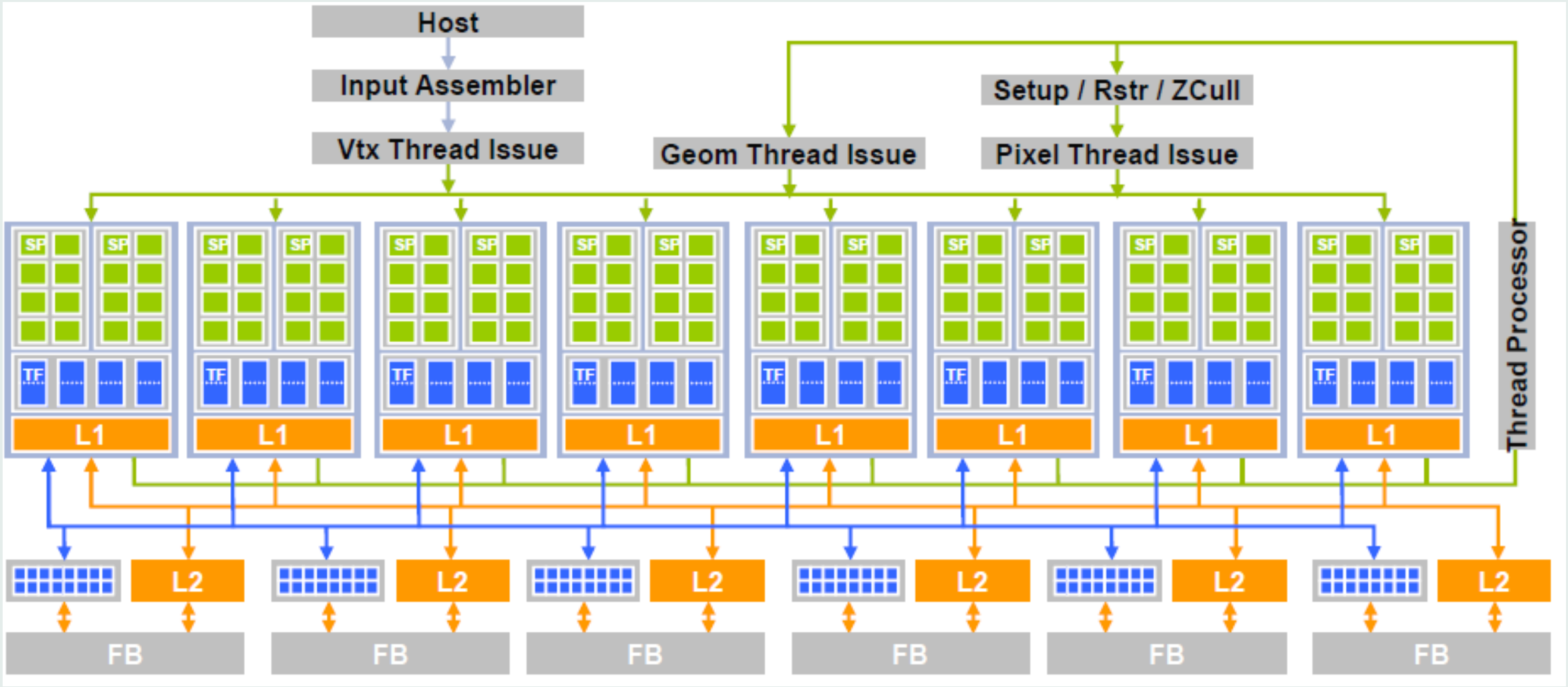
2、目前电脑都是多核的，一个核在同一时刻运行一个线程，超线程技术处理两个线程，目前最常见的CPU是16核32线程，最高EPYC 7H12

GPU? CUDA? cudnn? libcu++

多线程的优点：

- 线程机制使可以同时完成多个任务；可以使程序的响应速度更快；可以让占用大量处理时间的任务或当前没有进行处理的任务定期将处理时间让给别的任务；可以随时停止任务；可以设置每个任务的优先级以优化程序性能
- 程序具有异步执行能力以充分发挥机器计算能力，程序还可以利用其他计算机的处理能力
- 合理的线程分工使得数据计算与用户交互得到均衡

线程的并行

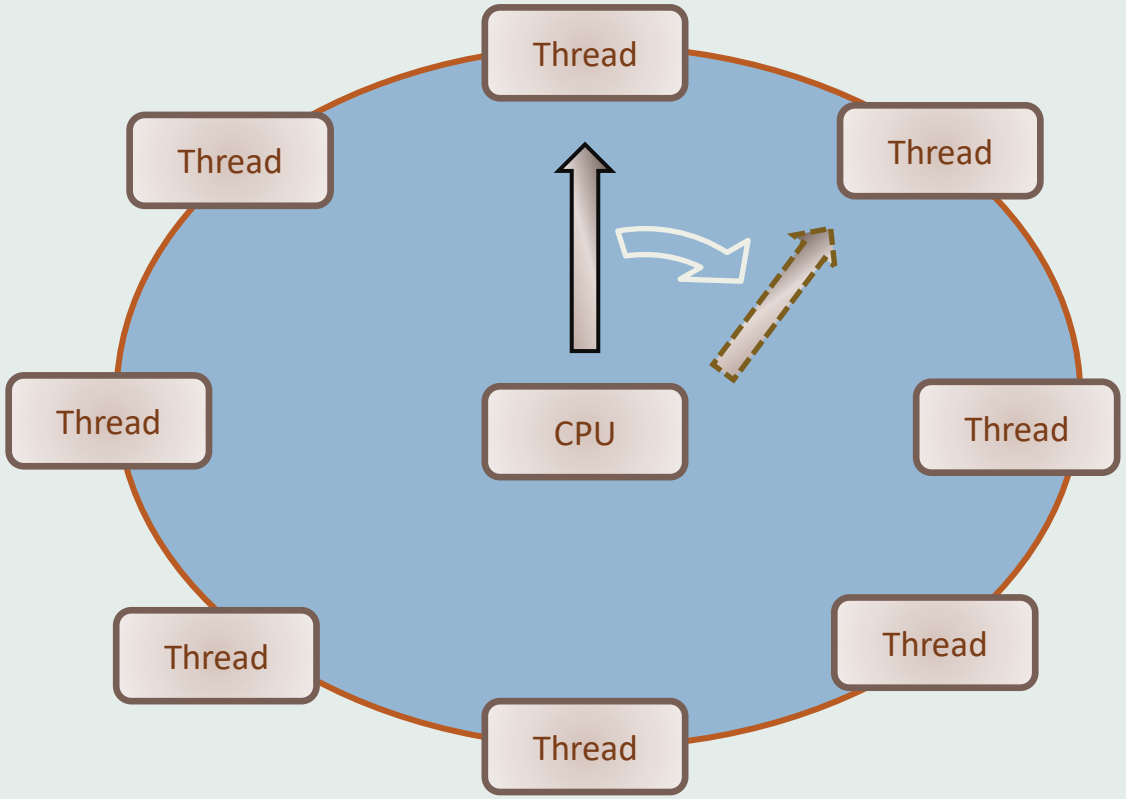

Host
Input Assembler
Vtx Thread Issue
Geom Thread Issue
Setup / Rstr / ZCull
Pixel Thread Issue
Thread Processor
SP
TF
L1
L2
FB

线程的并发

机器采用时间片轮转算法轮流执行线程，形成并发执行

线程可以很好平衡程序响应与数据处理，还能通过网络利用其它处理机资源

线程同步控制非常复杂，调试困难



线程应用场合

1. 网络通信程序
2. 与Web服务器和数据库操作
3. 执行占用大量时间的操作
4. 有不同优先级的任务
5. 用户响应效能与数据运算均衡
6. 机器学习

线程缺点

- 线程上下文信息消耗计算机资源
- 线程上下文切换过程，线程会带来资源特殊要求和潜在冲突。如果线程过多，系统管理线程的负担会加大，则其中大多数线程都不会产生明显的进度
- 线程控制代码非常复杂，并可能产生许多bug
- 线程的非正常终结会造成资源浪费影响系统的运行性能

Cache与提升多线程效率

- https://www.pcmag.com/encyclopedia/term/cache
- https://www.pcmag.com/encyclopedia/term/cache-line
- https://open-cas.github.io/cache_line.html#:~:text=A%20cache%20line%20is%20the%20smallest%20portion%20of,the%20cache%20mappings%20are%20aligned%20to%20these%20blocks.
- https://software.intel.com/content/www/us/en/develop/articles/coding-for-performance-data-alignment-and-structures.html
- https://stackoverflow.com/questions/14707803/line-size-of-l1-and-l2-caches
- https://community.intel.com/t5/Intel-oneAPI-Threading-Building/cache-size/td-p/903871
- https://www.cnblogs.com/gujiangtaoFuture/articles/11163844.html
- https://blog.csdn.net/weixin_43618070/article/details/89206134
- https://www.cnblogs.com/gujiangtaoFuture/articles/11163844.html
- 实验课详解

3.2 线程跨域访问

- 界面中的控件（textBox1等）是由主线程创建的，thread线程是另外创建的一个线程，在.NET上执行的是托管代码，C#强制要求这些代码必须是线程安全的，即不允许跨线程访问Windows窗体的控件；

防止一个线程在重画控件，另一个线程修改控件上的文字

- 在遵守.NET安全标准的前提下，实现从一个线程成功地访问另一个线程创建的空间，要使用C#的方法回调机制
- C#的方法回调机制，也是建立在委托基础上的，下面给出它的典型实现过程
  - ✓ 定义、声明回调
  - ✓ 初始化回调方法
  - ✓ 触发对象操作

```
// 定义回调
private delegate void DoSomeCallBack(Type para);
// 声明回调
DoSomeCallBack doSomeCallBack;
```

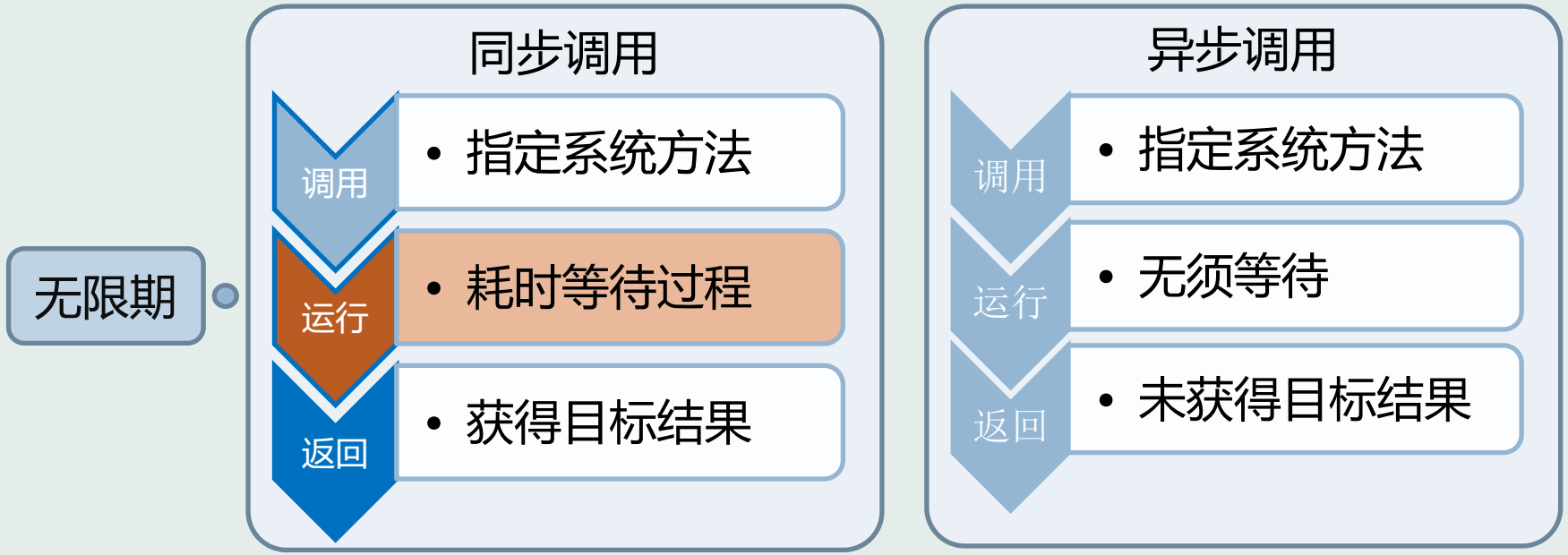
```
doSomeCallBack=new DoSomeCallBack(DoSomeMethod);
或doSomeCallBack = DoSomeMethod;
```

```
控件obj.Invoke(doSomeCallBack,arg);
或控件obj.Dispatcher.Invoke(doSomeCallBack,arg);
```

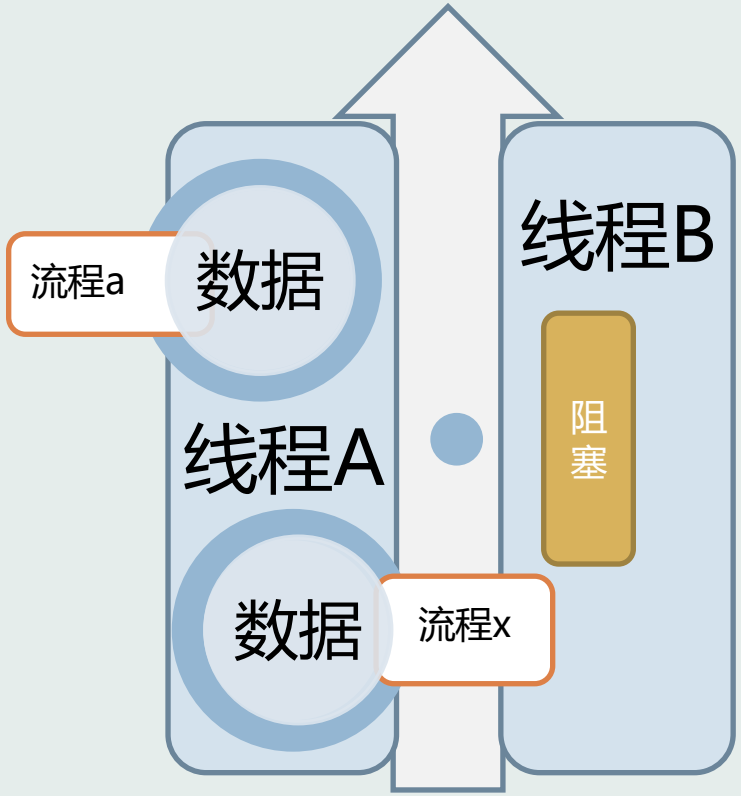
Invoke 类似C++中的SendMessage 同步
BeginInvoke 类似PostMessage 异步

Invoke方法会顺着控件树向上搜索，直到找到创建控件的那个线程

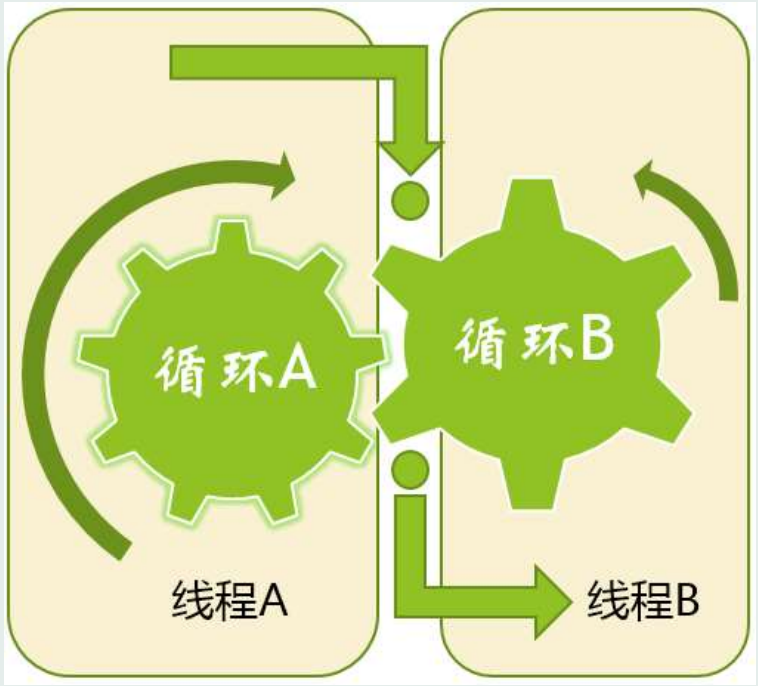
3.3 线程同步与异步调用



同步运行

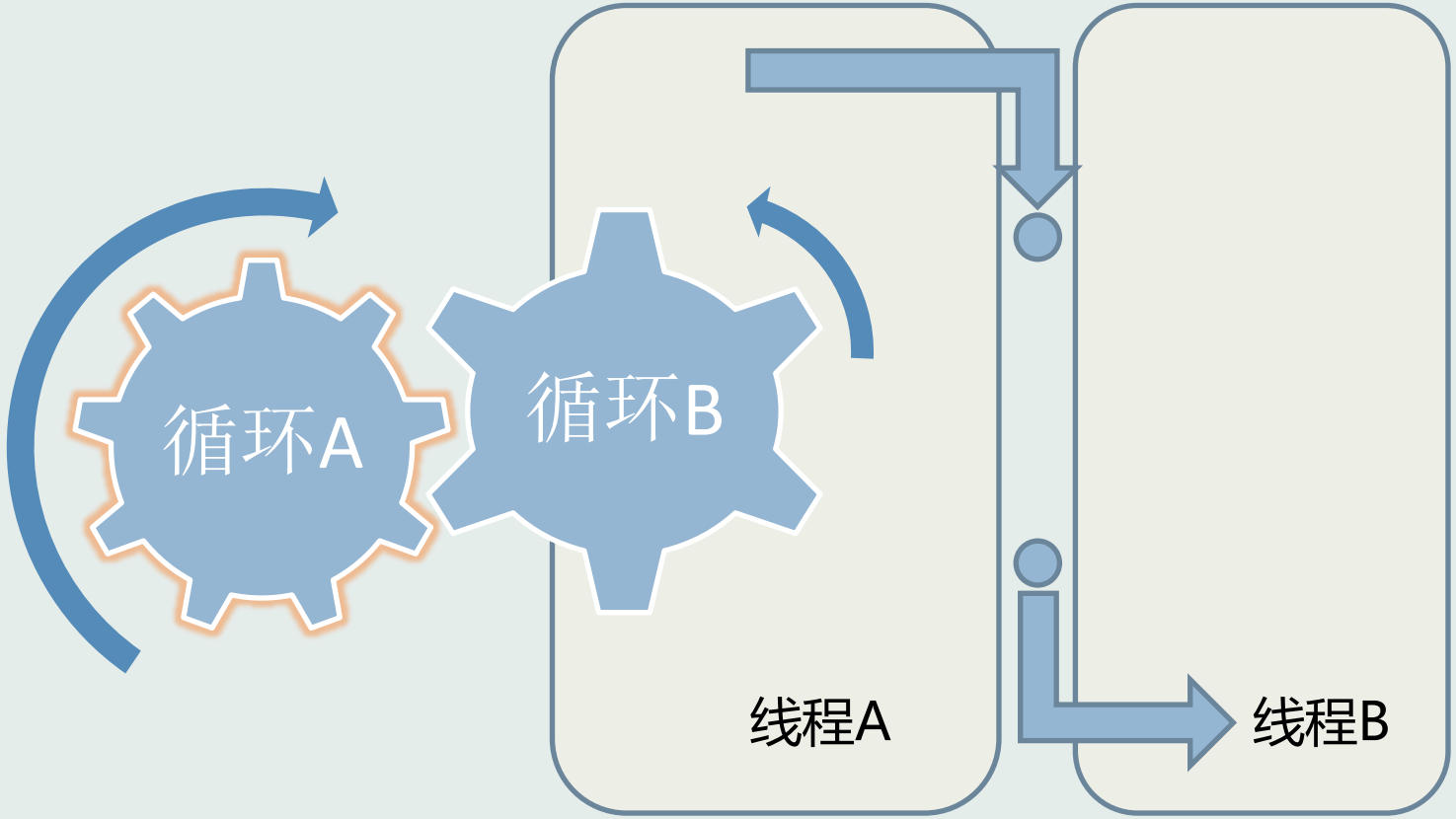


同步运行

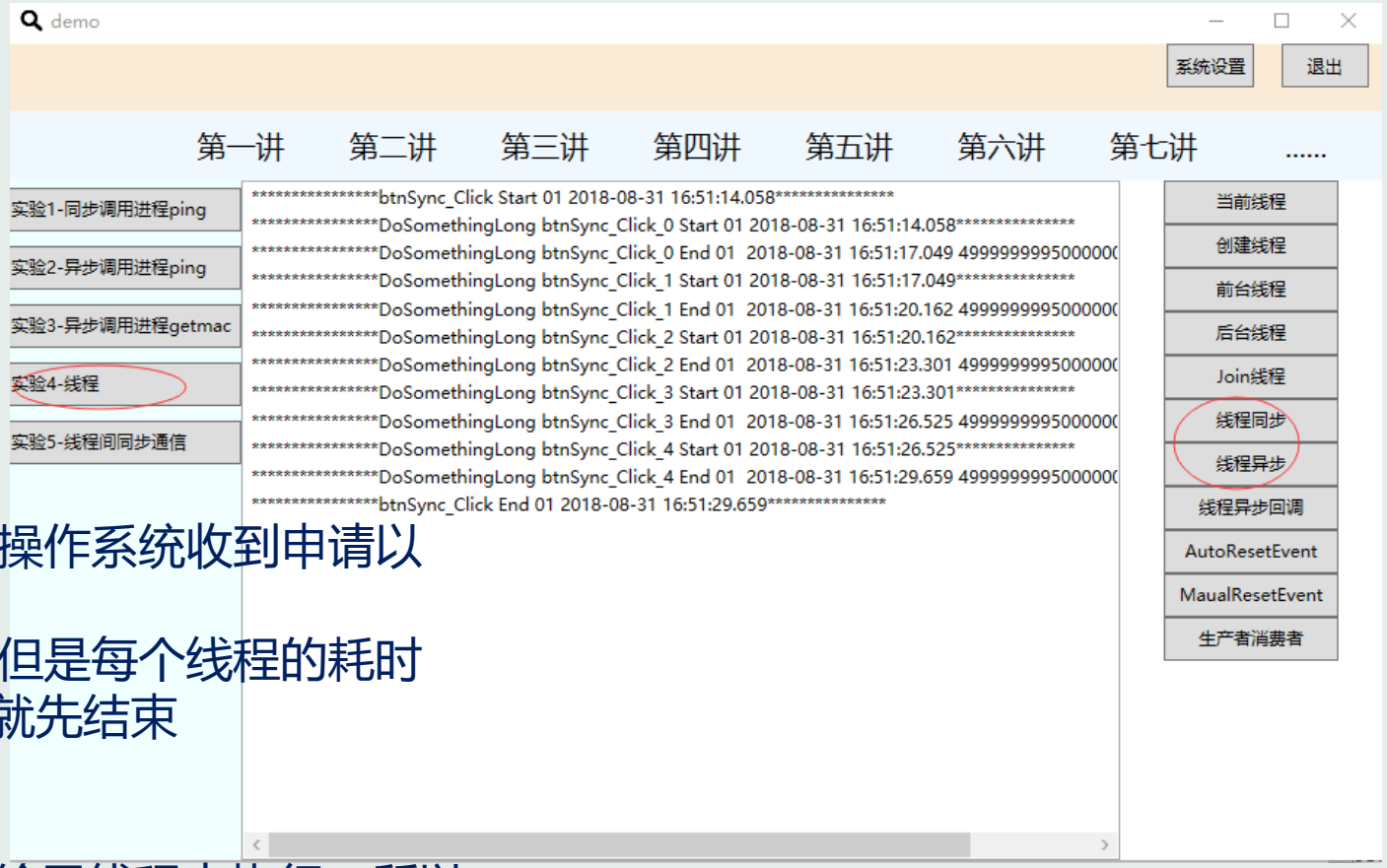


异步运行

线程的异步执行



线程的异步与同步实例



1. 同步方法执行是有序的，异步方法执行是无序的

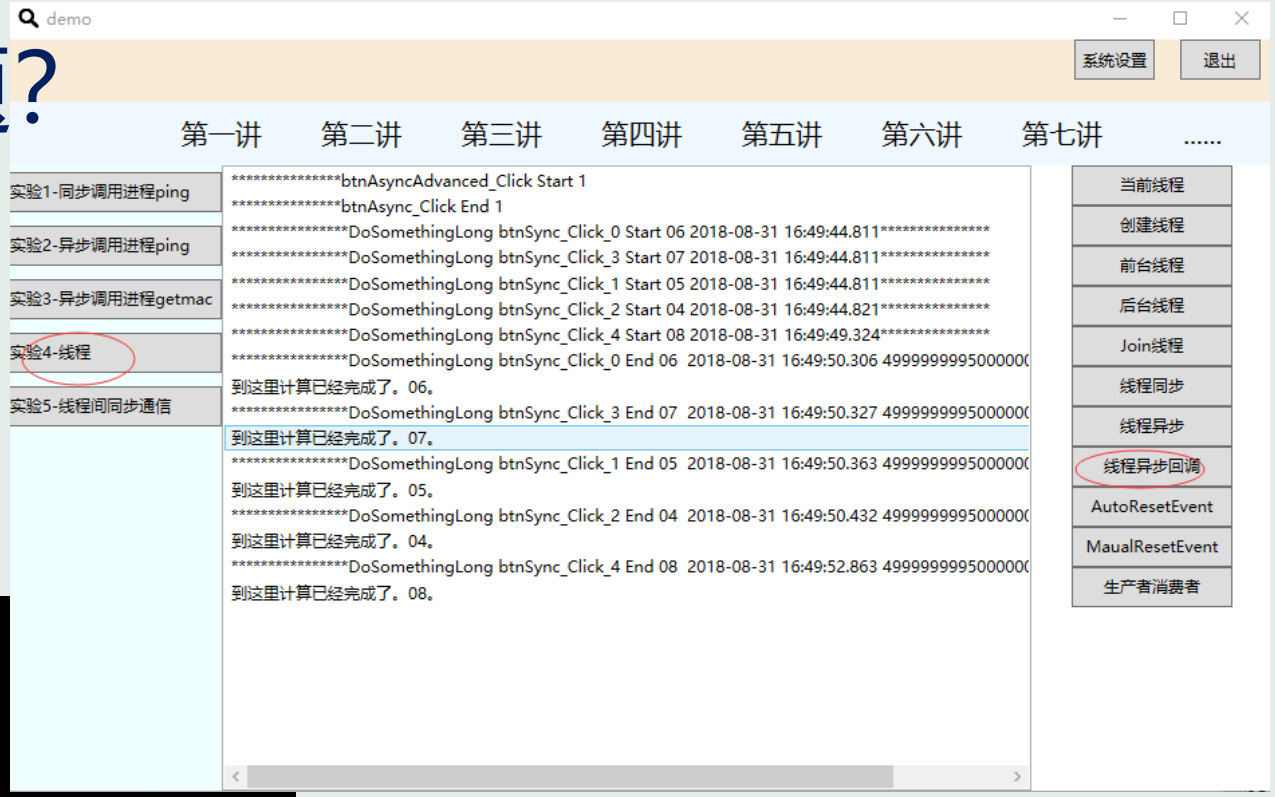
2. 异步方法无序包括启动无序和结束无序
   启动无序是因为同一时刻向操作系统申请线程，操作系统收到申请以后，返回执行的顺序是无序的，所以启动是无序的
   结束无序是因为虽然线程执行的是同样的操作，但是每个线程的耗时是不同的，所以结束的时候不一定是先启动的线程就先结束

3. 同步方法由于主线程忙于计算，所以会卡住界面
4. 异步方法由于主线程执行完了，其他计算任务交给子线程去执行，所以不会卡住界面，用户体验性好

5. 同步方法由于只有一个线程在计算，所以执行速度慢
6. 异步方法由多个线程并发运算，所以执行速度快，但并不是线性增长的（资源可能不够）。多线程也不是越多越好，只有多个独立的任务同时运行，才能加快速度

如何解决线程的异步无序问题?

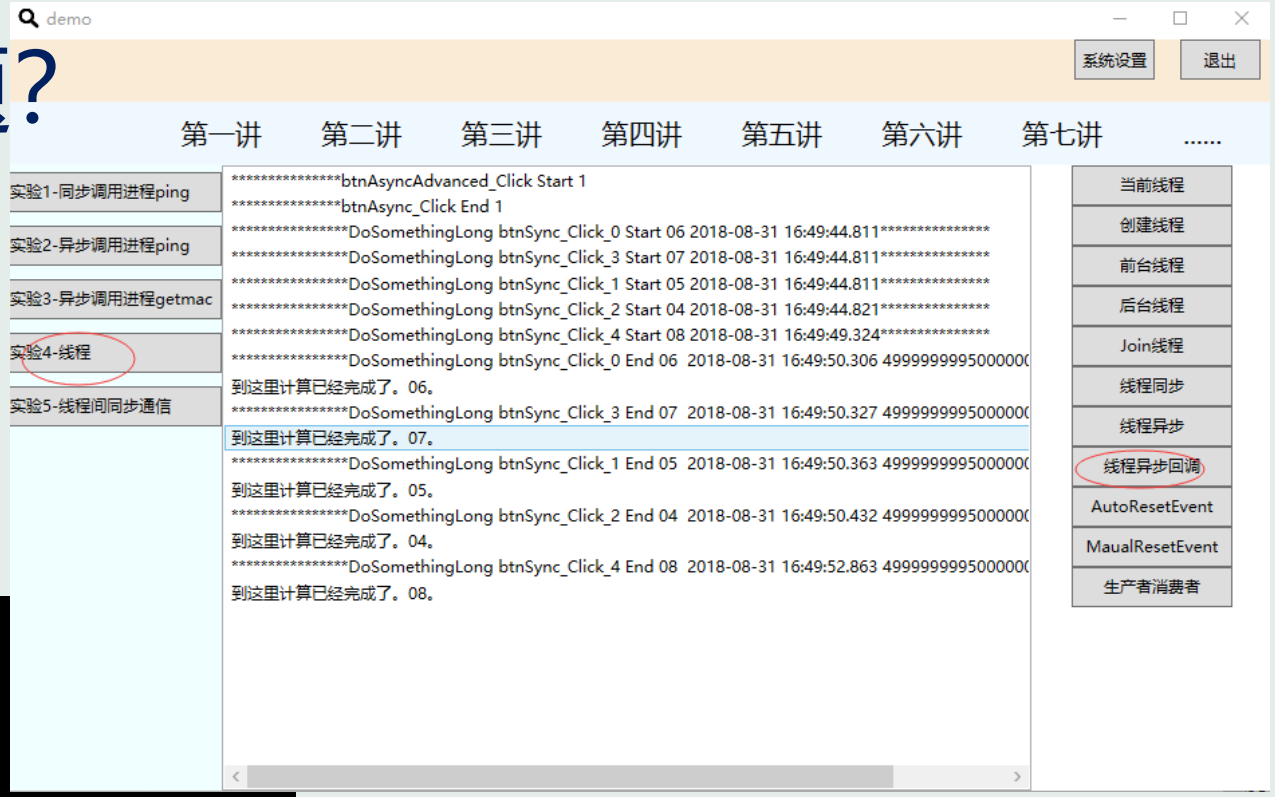
使用回调来解决异步线程的无序问题
在BeginInvoke的参数中指定回调函数



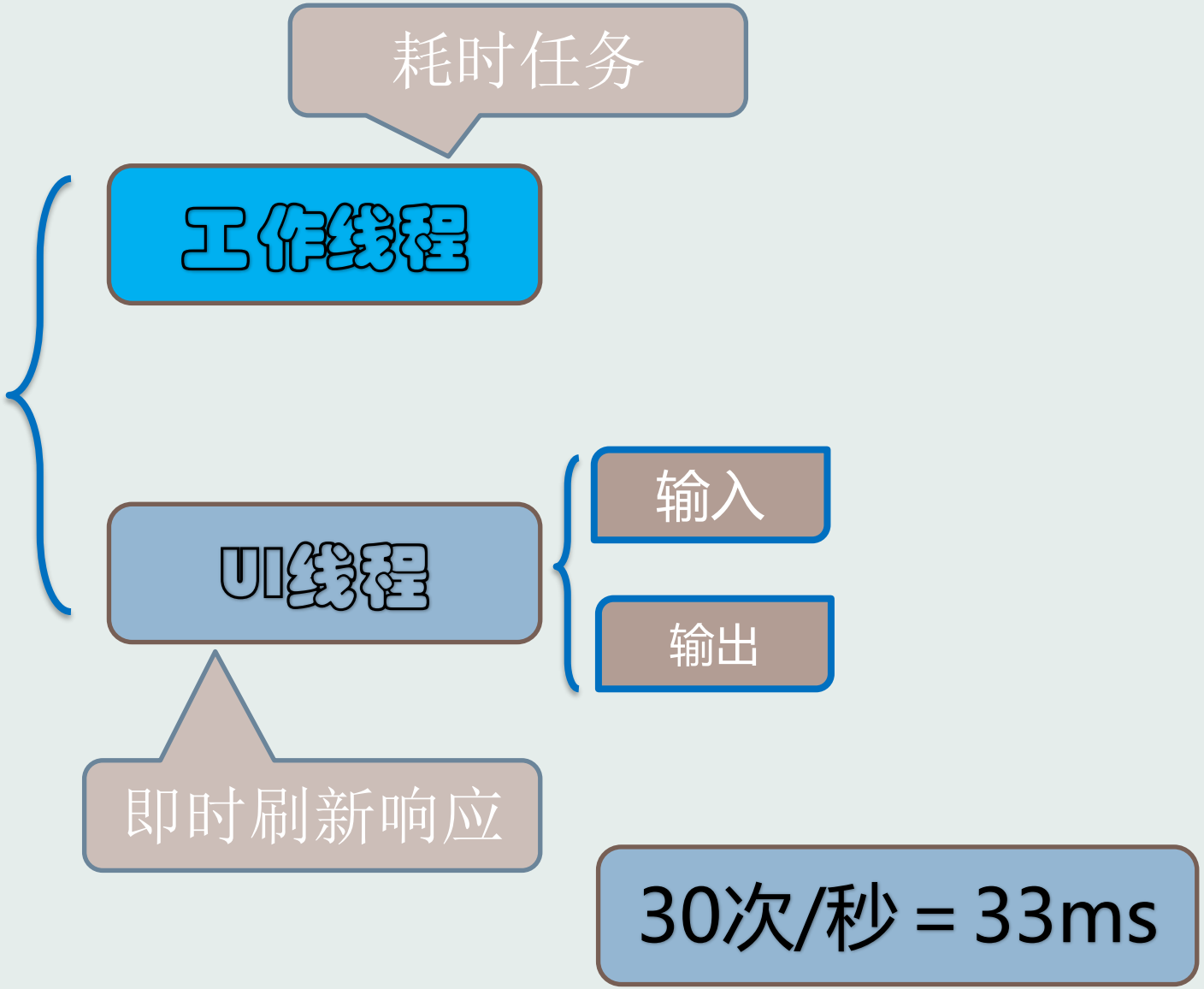

```
// 定义一个回调
AsyncCallback callback = p =>
{
    Console.WriteLine($"到这里计算已经完成了。
    {Thread.CurrentThread.ManagedThreadId.ToString("00")}。");
                update($"到这里计算已经完成了。"
        Thread.CurrentThread.ManagedThreadId.ToString("00") + "。");
};
// 异步调用回调
for (int i = 0; i < 5; i++)
{
    string name   string.Format($"btnSync_Click_{i}");
    asyncResult = action.BeginInvoke(name, callback, null);
}
```

如何解决线程的异步无序问题?

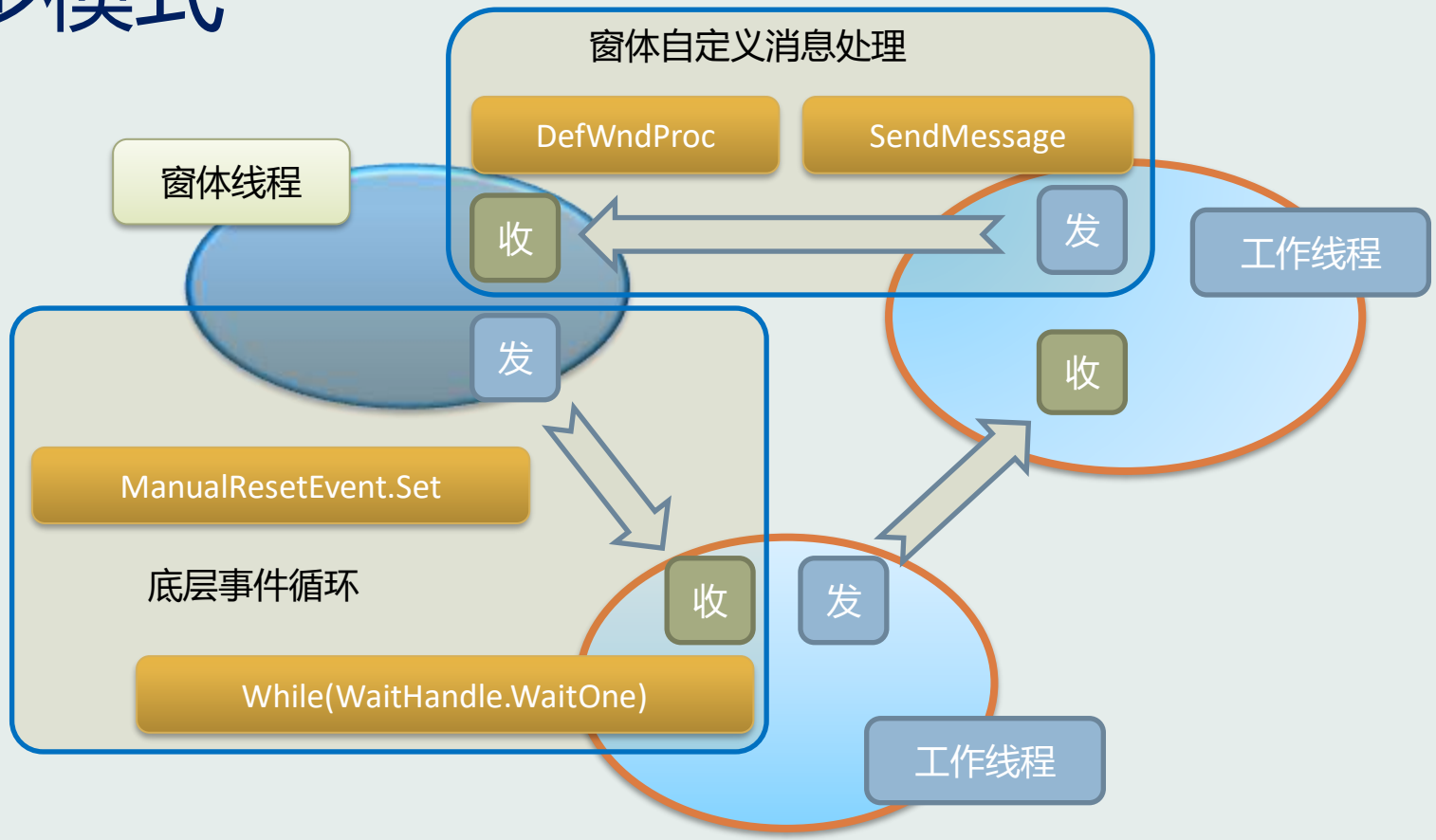
使用回调来解决异步线程的无序问题
在BeginInvoke的参数中指定回调函数




```
// 定义一个回调
AsyncCallback callback = p =>
{
    Console.WriteLine($"到这里计算已经完成了。
    {Thread.CurrentThread.ManagedThreadId.ToString("00")}。");
                update($"到这里计算已经完成了。" +
        Thread.CurrentThread.ManagedThreadId.ToString("00") + "。");
};
// 异步调用回调
for (int i = 0; i < 5; i++)
{
    string name = string.Format($"btnSync_Click_{i}");
    asyncResult = action.BeginInvoke(name, callback, null);
}
```

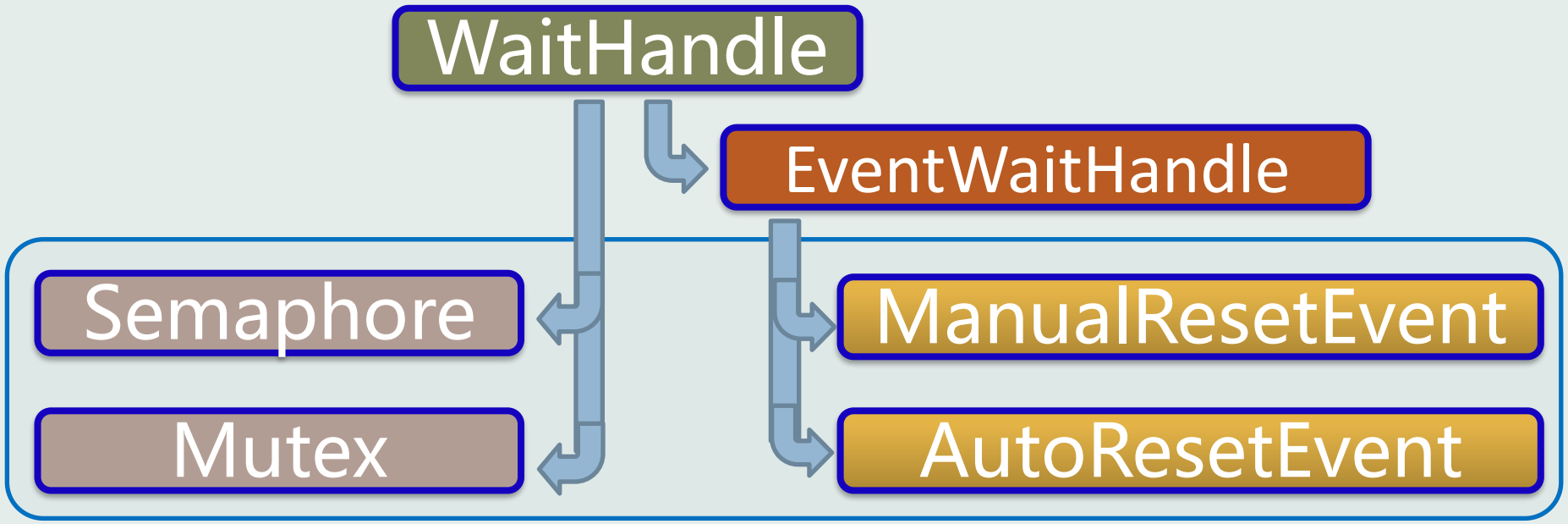

耗时任务
工作线程
UI线程
输入
输出
即时刷新响应
30次/秒 = 33ms

3.4 线程间同步模式



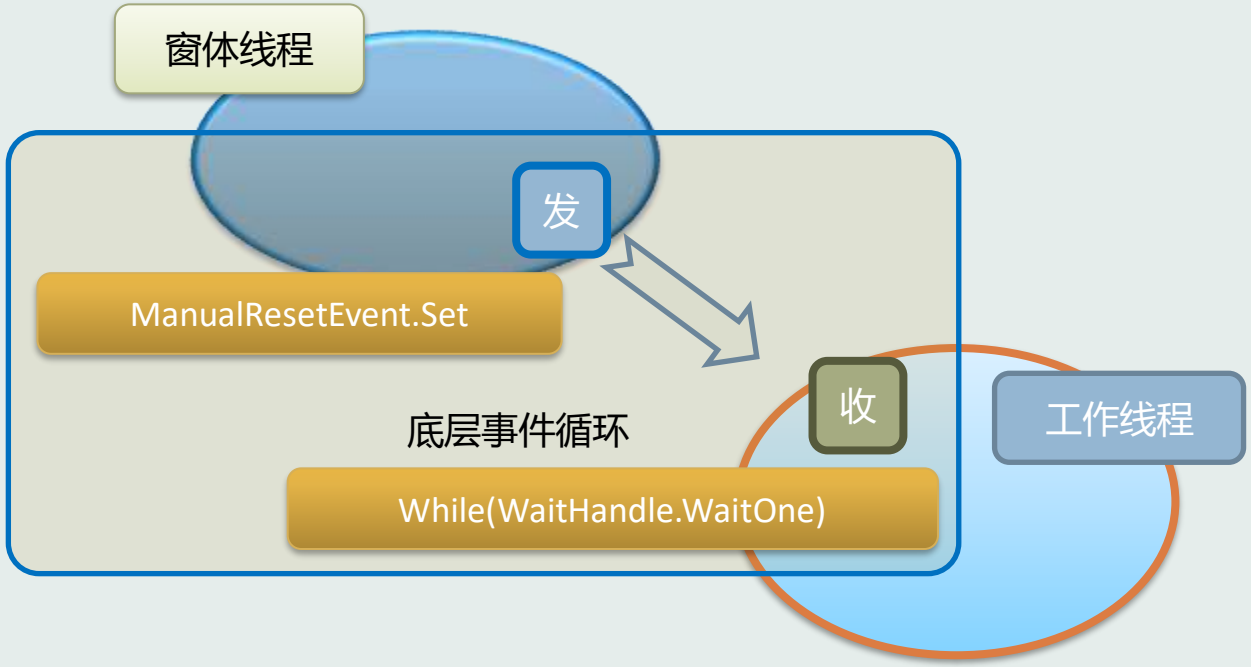
工作线程可以很容易用SendMessage来发消息
窗体线程可以发送ManualResetEvent事件给工作线程

WaitHandle类继承关系



线程如何接收消息?

工作线程没有消息队列，无法用窗体模式，线程因此显得是一个笨听众，接收方法是线程主动循环检查一些变量，但不能使用"忙检"，因为太耗CPU，要使用ManualResetEvent.WaitOne这样的方法，以最低的代价耗费cpu资源。



工作线程响应前打发时间的两种方式

- 采用IsOut + Sleep打发时间的方式

- ManualResetEvent.WaitOne打发时间的方式
  - 事件对象可实现并发执行中的前趋控制。当线程调用Wait方法时，如果等待对象状态没有激活，则调用线程暂停。对象被激活则线程继续执行

低级事件对象

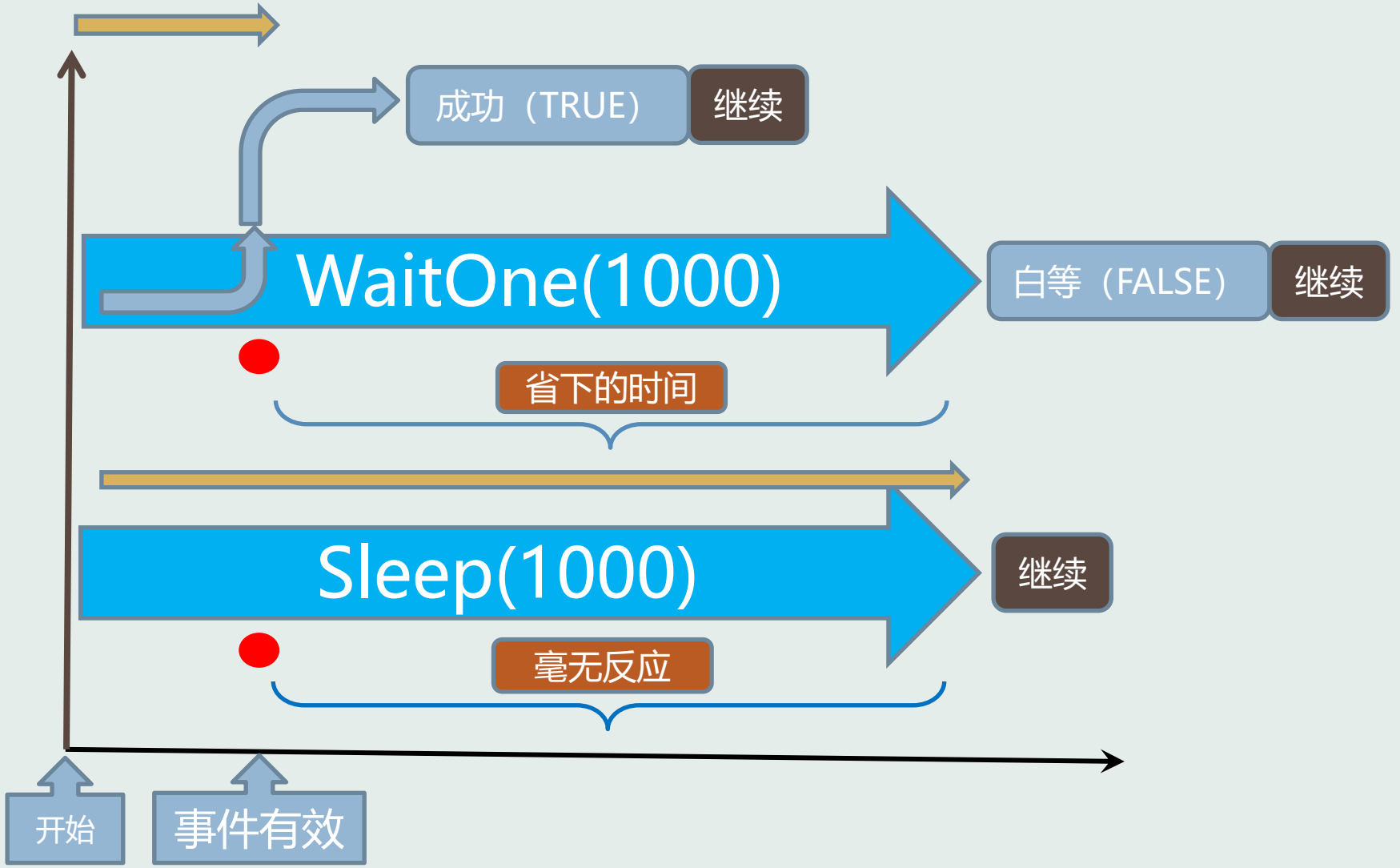
- 事件对象声明
  - public static ManualResetEvent User_Terminate_listen;
- 全局静态使得线程与窗体可以访问
- User_Terminate_listen.WaitOne();
- 代表最小的信息量(1bit)

ManualResetEvent

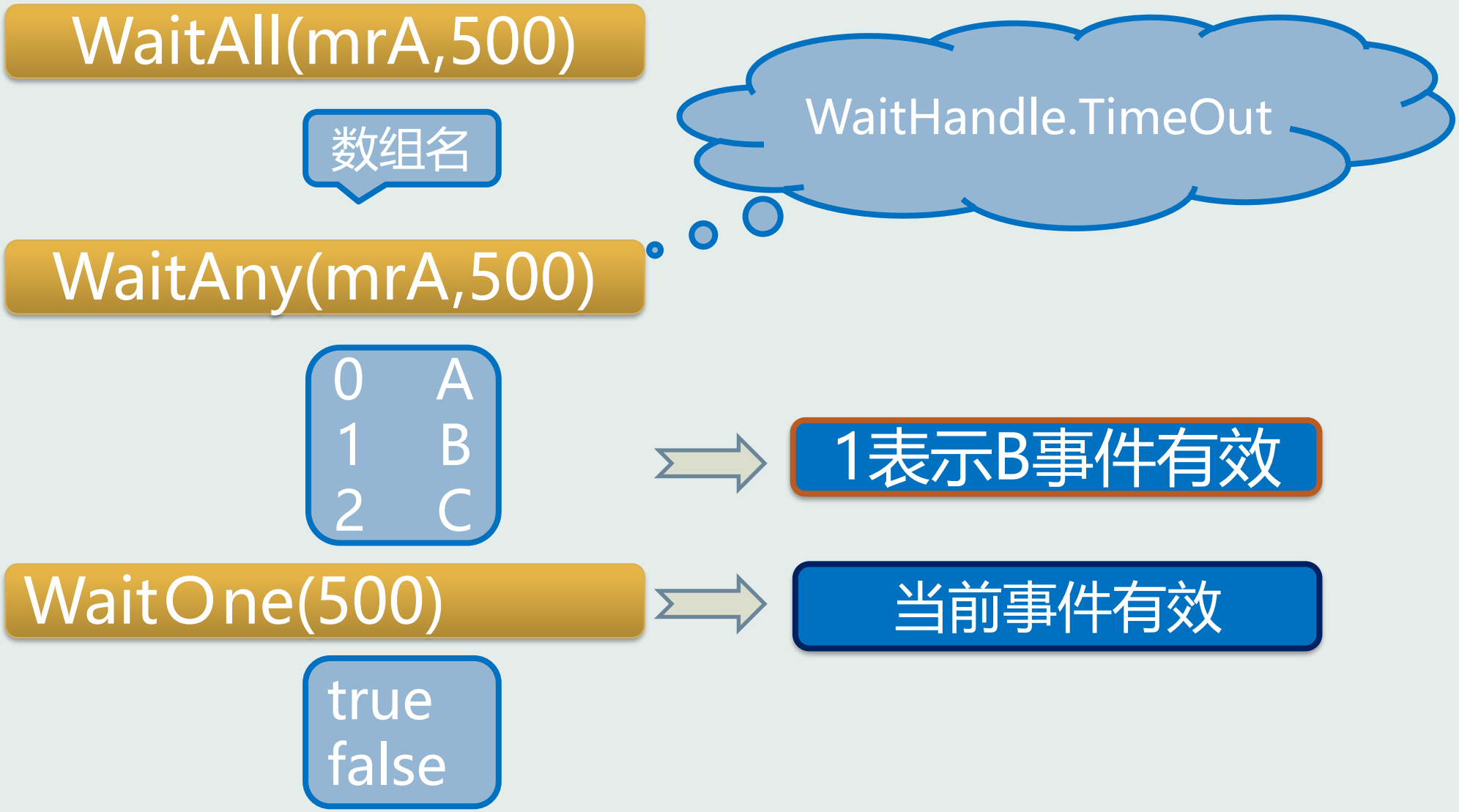
Set设置为有效

Reset设置为无效

WaitOne与Sleep比较



工作线程运行逻辑


WaitAll(mrA,500)
数组名
WaitHandle.TimeOut
WaitAny(mrA,500)
0 A
1 B
2 C
1表示B事件有效
WaitOne(500)
当前事件有效
true
false

工作线程间的通信

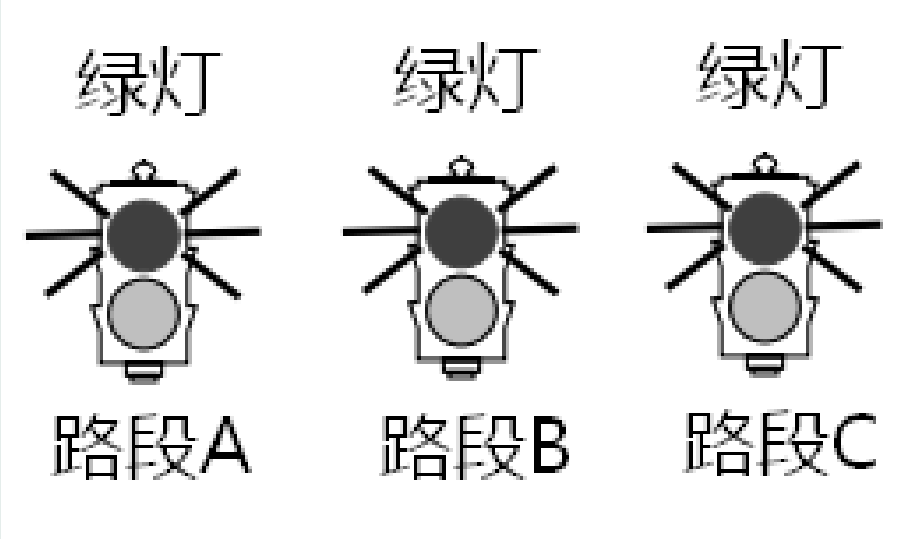
- WaitOne方法等待当前事件(信号)有效
- WaitAny方法等待事件(信号)数组中任一事件有效，对应或关系实现同步
- WaitAll方法等待事件（信号）数组中所有事件有效，对应与关系实现同步

ManualResetEvent.WaitOne要点

- 一是时间效果上阻止线程继续
- 二是在获得信号状态后返回值为true，而超时未获得返回值为false
- 三是获得信号状态将不再继续未等待完的时间。
- 它的使命是完成信号传递，至于功能是启动操作还是终止操作，程序得到的状态都是信号的true
- ManualResetEvent比AutoResetEvent要可靠，它可将信号传给多个线程，而线程会重置AutoResetEvent的状态，即中断信号的传递。

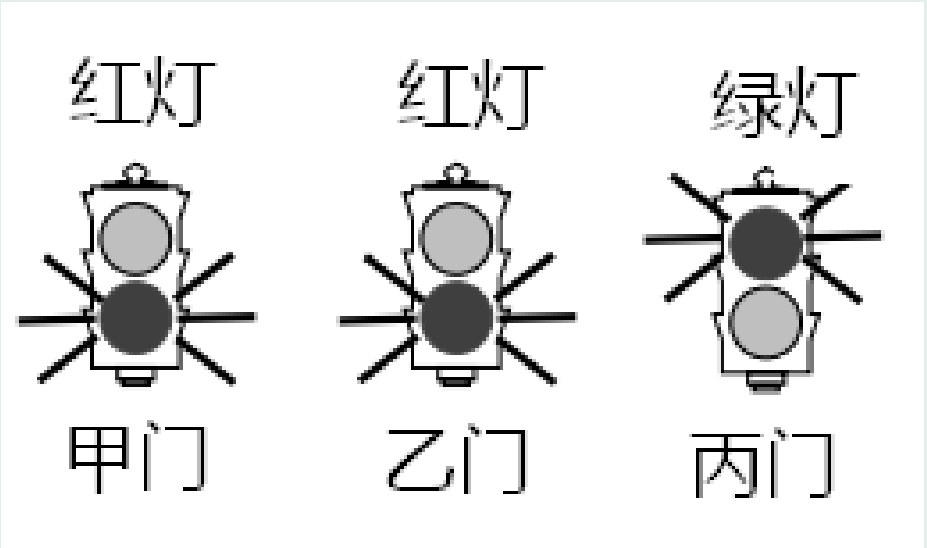
具有与关系的同步方式

- ManualResetEvent.WaitAll方法
  - 当所有事件状态同时激活时



具有或关系的同步方式

- ManualResetEvent.WaitAny方法
  - 当任一事件状态激活时



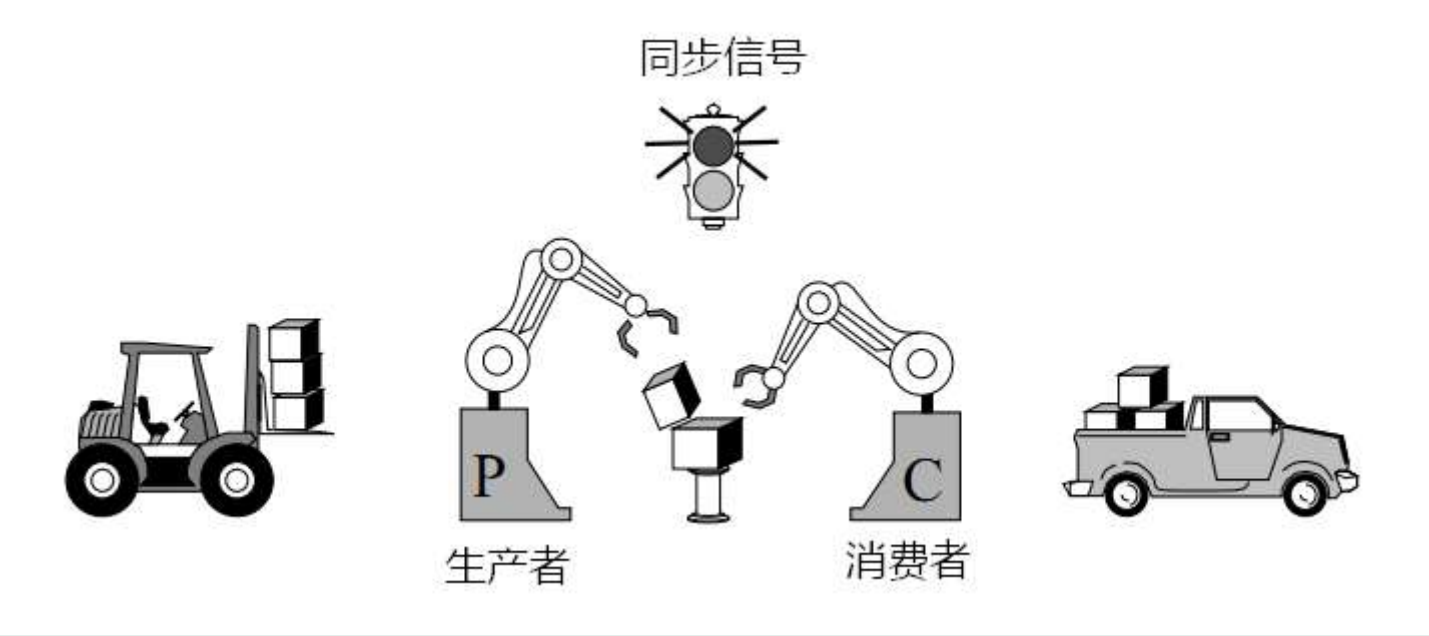
使用事件的抓屏程序

窗体线程发起抓屏事件；
工作线程抓屏，并保存

4.5线程同步与死锁

多个线程对共享资源访问会造成冲突。为了避免冲突，必须对共享资源进行同步或控制对共享资源的访问。如果在相同或不同的应用程序域中未能正确地使访问同步，则会导致出现一些问题，这些问题包括死锁和争用条件等，其中死锁是指两个线程都停止响应，并且都在等待对方完成；争用条件是指由于意外地出现对两个事件的执行时间的临界依赖性而发生反常的结果。

同步资源访问控制



需要同步的资源

1. 系统资源（如通信端口）
2. 多个进程所共享的资源（如文件句柄）
3. 由多个线程访问的单个应用程序域的资源（如全局、静态和实例字段）

同步控制类

类名	类说明
Mutex	提供对资源的独占访问，可用于同步不同进程中的线程。
Monitor	对特定对象获取锁和释放锁来公开同步访问代码区域的能力，它是局部的。
Interlocked	同步对多个线程共享的变量的访问的方法。
Semaphore	命名信号量（系统范围）或本地信号量。Windows 信号量是计数信号量，可用于控制对资源池的访问。
AutoResetEvent	本地等待处理事件，在释放了单个等待线程以后，该事件会在终止时自动重置，只能释放单个等待线程。
ManualResetEvent	表示一个本地等待处理事件，在已发事件信号后必须手动重置该事件，在对象保持已发信号状态期间，可以释放任意数目的等待线程。

互斥量Mutex介绍

- 最多只能有一个线程可以获取并拥有它，它适合不同线程对同一共享资源互斥访问的应用场合，例如对全局变量，同一个文件或者是数据库同一个对象的访问等，设置程序先获得互斥量再访问共享资源。

互斥量的使用

- 互斥量的创建
- WaitOne方法
- ReleaseMutex方法

互斥量的使用

- 线程可调用多次的 WaitOne 方法重复对其所有，使用 ReleaseMutex 方法释放对互斥量所属权，而每一个成功的 WaitOne 方法对应一次 ReleaseMutex
- WaitOne方法不仅等待互斥量的状态，还使线程拥有它
- 互斥量最好不要使用WaitAny,WaitAll方法

互斥量的使用

- 线程运行终止 mutex 被放弃，互斥量仍可被其它线程取得所属权，但会获得 AbandonedMutexException 异常

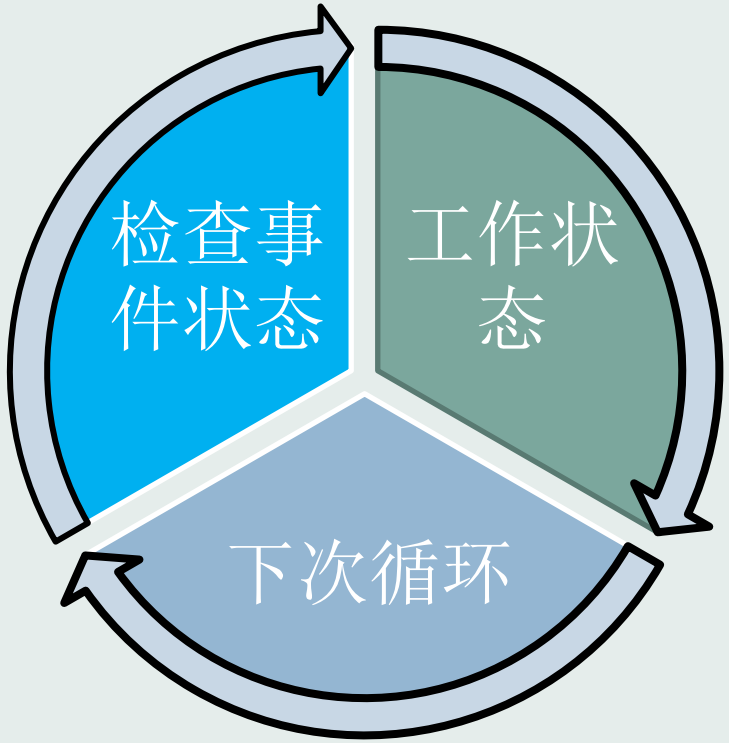
互斥量的使用

- Mutex类
  - releaseMutex方法

ManualResetEvent的使用

仅适用本机内

- 文件
- 剪切板
- 无名管道
- 事件对象



上机练习作业

- 配置CUDA开发环境 – 提交PPT获取平时成绩
- 配置cudnn开发环境 – 提交PPT获取平时成绩
- 采用信号量机制实现消费者与生产者的线程同步
  - 1个生产者，1个消费者
  - 1个生产者，多个消费者
  - 多个生产者，1个消费者
  - 多个生产者，多个消费者

THANK YOU !