## Sigslot：信号槽

所有的槽必须继承has\_slots。

signal\_with\_thread\_policy提供信号，sigslot.connect连接信号与槽，将slot对象与方法封存储到signal中，不同的signal具有不同的槽参数类型，所有连接时注意参数对应。

## event

event.h/event.cc文件中只有talk\_base::Event类。

### 1.1 talk\_base::Event

该类主要实现了跨平台的Win32 Event功能（正如前言中所说，本文假定读者已经很熟悉Win32平台的各种组件，如有不明白的地方可以参考MSDN）。talk\_base::Event类的各个成员函数与Win32 Event所提供的API几乎一致，所以不再多做解释。

在Linux系统中，WebRTC使用了mutex和条件变量来实现Event的功能。首先，我将对Win 32 API和pthread API做一下类比：

Event::Event(bool manual\_reset, bool initially\_signaled)

创建event，

manual：wait后是否手动取消event状态

initially\_signaled：初始状态是否event状态

void Event::Set()

触发event状态，解除所有等待 event的等待

void Event::Reset()

设置为非event状态

bool Event::Wait(int milliseconds)

等待event状态的到来，最多等待milliseconds时长，如果当前是event状态，则直接返回。

## CriticalSection:临界区

CriticalSection()

创建临界区

CriticalSection::Enter()

进入临界区，相当于可递归版本的pthread\_mutex\_lock()，如果自己进入过，则可以继续进入，否则等待锁释放

CriticalSection::TryEnter()

相当于可递归版本的pthread\_mutex\_trylock()

CritScope, TryCritScope:

采用构造函数和析构函数对CriticalSection的Enter和TryEnter进行封装

SharedExclusiveLock

共享互斥锁，有时也被称为单写多读锁。   
简单点的应用场景就是：一个写（LockExclusive）多个读（LockShared）。

LockExclusive，UnlockExclusive：加锁结束私有的锁

LockShared，UnlockShared：加锁解锁共有的锁。

即有LockExclusive时，其他所有的锁都要等待，LockShared时其他LockShared不需要等待。

## talk\_base::MessageHandler

talk\_base::MessageHandler类的主要功能是定义了消息处理器的基本数据结构，子类在继承了该类之后要重载OnMessage函数，并实现消息响应的逻辑。

talk\_base::FunctorMessageHandler : public MessageHandler

 只是封装了一个函数

FunctorMessageHandler(const FunctorT& functor)

构造某个函数的Handler

OnMessage(Message\* msg)

执行函数

result()

返回执行的结果

talk\_base::FunctorMessageHandler类的主要功能是将一个函数投递到目标线程执行。该类主要通过functor模板实现（熟悉C++的读者应该不会对它陌生，不熟悉的可以上网查找），而且定义了一个针对返回值类型为void的函数的特化版本（模板的特化和偏特也应该是一个C++程序员掌握的一个知识点，该语法有些难度）。用户不需要创建或者继承talk\_base::FunctorMessageHandler类，仅需调用talk\_base::Thread::Invoke函数就能使用它的功能。

## messagequeue

messagequeue.h/messagequeue.cc文件是多路信号分离器的重要组成部分。它实现了消息一个完整地消息队列，该队列包括立即执行消息队列、延迟执行消息队列和具有优先级的消息队列。其中，talk\_base::MessageQueue类也是talk\_base::Thread类的基类。所以，所有的WebRTC的线程都是支持消息队列的。

MessageQueueManager

talk\_base::MessageQueueManager类是一个全局单例类。这个类看似比较复杂，但是功能其实非常简单——仅仅为了在所有的talk\_base::MessagerQueue中删除与指定的talk\_base::MessageHandler相关的消息。WebRTC的消息队列在发送消息的时候要指定消息处理器（talk\_base::MessageHandler）。如果某个消息处理器被析构，那么与之相关的所有消息都将无法处理。所以，创建了这个全局单例类来解决这个问题（见talk\_base::MessageHandler析构函数）。

4.2 MessageData

这一节的内容将包括talk\_base::MessageData类以及多个它的子类和几个工具函数。这些类和函数都很简单，所以就不介绍代码和原理，仅仅罗列一下它们的功能。

4.2.1 talk\_base::MessageData

定义了基类，并将析构函数定义为虚函数。

4.2.2 talk\_base::TypedMessageData

使用模板定义的talk\_base::MessageData的一个子类，便于扩展。

4.2.3 talk\_base::ScopedMessageData

类似于talk\_base::TypedMessageData，用于指针类型。在析构函数中，自动对该指针调用delete。

4.2.4 talk\_base::ScopedRefMessageData

类似于talk\_base::TypedMessageData，用于引用计数的指针类型。

4.2.5 talk\_base::WrapMessageData函数

模板函数，便于创建talk\_base::TypedMessageData

4.2.6 talk\_base::UseMessageData函数

模板函数，用于将talk\_base::TypedMessageData中的Data取出

4.2.7 talk\_base::DisposeData

这是一个很特殊的消息，用以将某个对象交给消息引擎销毁。可能的用途有2个：1. 有些函数不便在当前函数范围内销毁对象，见范例talk\_base::HttpServer::Connection::~Connection；2.某对象属于某一线程，因此销毁操作应该交给所有者线程（未见范例）。WebRTC用户不需要自行使用该类，调用talk\_base::MessageQueue::Dispose函数即可使用它的功能。

以上7个类或函数的实现非常简单，有C++使用经验的读者非常容易就能理解（标准库中就有相似的类）

4.3 Message

这一节将简单介绍一下3个类：talk\_base::Message、talk\_base::DelayedMessage和talk\_base::MessageList。

4.3.1 talk\_base::Message

定义了消息的基本数据结构。

  Location posted\_from：从哪里抛出的，便于调试

MessageHandler \*phandler：消息处理函数

MessageData \*pdata：数据

uint32\_t message\_id：消息id

int64\_t ts\_sensitive：敏感时间，如果大于0，则消息触发时间大于ts\_sensitive时，会发出警告日志

4.3.2 talk\_base::DelayedMessage

定义了延迟触发消息的数据结构。在talk\_base::MessageQueue中，延迟消息被存放在以talk\_base::DelayedMessage::msTrigger\_排序（talk\_base::DelayedMessage类定义了operator<）的队列中。如果2个延迟消息的触发时间相同，响应顺序按先进先出原则。

这里我将简单介绍一下各个成员变量的用途：

cmsDelay\_：延迟多久触发消息，仅作调试使用

msTrigger\_：触发消息的时间

num\_：添加消息的时间

msg\_：消息本身

在使用延迟消息时，不需要自行构建talk\_base::DelayedMessage实例。直接调用talk\_base::MessageQueue::PostDelayed或者talk\_base::MessageQueue::PostAt函数即可。

4.3.3 talk\_base::MessageList

消息列表，定义为std::list<talk\_base::Message>