**Libjingle**

**什么是libjingle？**

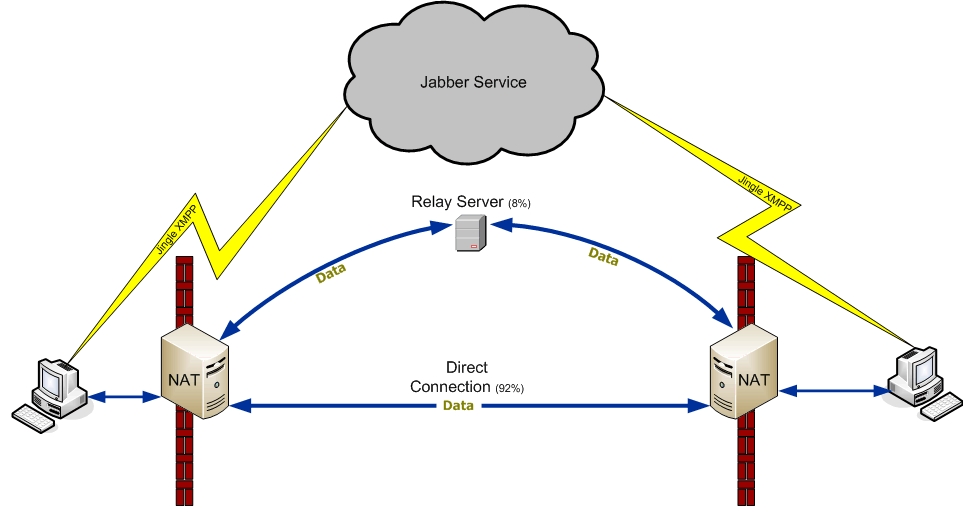
Libjingle是一个方便实现P2P传输的开源库，由google公司开发，并与2005年12月15日发布第一个版本，可以粗略的看成是Jingle协议的C++实现库，Google Talk即是基于这个库开发的。通过libjingle我们可以建立一个直通的网络连接(无视中间的NAT、防火墙、中继服务器和代理等)，无需特别关心Session建立的细节(加解密、格式等)，直接进行数据的交换。它也实现了一些辅助的功能，例如XML的解析和网络代理的处理。我们通过它可以实现如下的应用：

一个多用户的语音聊天应用

一个多用户的视频会议应用

一个多用户的现场音乐、流媒体应用

一个点到点的文件传输和共享的应用



**什么是p2p？**

对等网络，即对等计算机网络，是一种在对等者（Peer）之间分配任务和工作负载的分布式应用架构 ，是对等计算模型在应用层形成的一种组网或网络形式。P2P可以理解为对等计算或对等网络。其定义为：网络的参与者共享他们所拥有的一部分硬件资源（处理能力、存储能力、网络连接能力、打印机等），这些共享资源通过网络提供服务和内容，能被其它对等节点（Peer）直接访问而无需经过中间实体。在此网络中的参与者既是资源、服务和内容的提供者（Server），又是资源、服务和内容的获取者（Client） 。

P2P的核心是数据存储在客户本地，通过存储信息(名称、地址、分块)的查询，让终端之间直接数据传递。P2P网络让网络上的数据流量分散化，同时管理点不仅没有服务容量的压力，而且只存储数据的索引与链接。

**p2p打洞**

NAT（Network Address Translation，网络地址转换）是1994年提出的。当在专用网内部的一些主机本来已经分配到了本地IP地址（即仅在本专用网内使用的专用地址），但现在又想和因特网上的主机通信（并不需要加密）时，可使用NAT方法。

这种方法需要在专用网连接到因特网的路由器上安装NAT软件。装有NAT软件的路由器叫做NAT路由器，它至少有一个有效的外部全球IP地址。这样，所有使用本地地址的主机在和外界通信时，都要在NAT路由器上将其本地地址转换成全球IP地址，才能和因特网连接。

另外，这种通过使用少量的公有IP 地址代表较多的私有IP 地址的方式，将有助于减缓可用的IP地址空间的枯竭。

私有IP地址范围：

A: 10.0.0.0~10.255.255.255 即10.0.0.0/8

B:172.16.0.0~172.31.255.255即172.16.0.0/12

C:192.168.0.0~192.168.255.255 即192.168.0.0/16

这些地址是不会被Internet分配的，它们在Internet上也不会被路由，虽然它们不能直接和Internet网连接，但通过技术手段仍旧可以和 Internet通讯（NAT技术）。我们可以根据需要来选择适当的地址类，在内部局域网中将这些地址像公用IP地址一样地使用。在Internet上，有些不需要与 Internet通讯的设备，如打印机、可管理集线器等也可以使用这些地址，以节省IP地址资源。

NAT共分为两大类：Cone NAT和Symmetric NAT。Cone NAT指的是只要源IP端口不变，无论发往的目的IP是否相同，在NAT上都映射为同一个端口，形象的看来就像锥子一样，而Symmetric NAT对于发往不同目的IP的会话在NAT上将映射为不同的端口，也就是不同的会话。 其中Cone NAT又可细分为3类，分别是Full Cone型、Restricted Cone型和Restricted Port Cone。限制的严格程度和对局域网内主机的保护由松到紧依次为：Full Cone、Restricted Cone、Restricted Port Cone、Symmetric NAT。

这里“限制”指的是NAT对由外到内的数据包进行审查、过滤，看看数据包的源地址和他发送到的“洞”是否有关系，如果没有那么就将其丢弃。

“由松到紧”指后者不仅继承了前者在限制上的特性，而且自己还添油加醋的干了些坏事，以至于对“由外到内”的数据包比前者有着更严格的限制。例如Restricted Cone限制了外部进入内部的IP，使得只有被打洞IP发出的数据包才允许进入NAT，而Restricted Port Cone不但限制了IP，还限制了端口，使得只有被打洞的IP:PORT才能往这个洞里发送数据，其他任何来自不同于被打洞地址（IP:PORT）的数据包都不能使用这个洞将数据发送到NAT之后。

2.NAT对Session的映射

这里列出NAT对于内网地址映射到外网地址的一些普遍规律，可将其分为4种情况，以下的Session即为通信双方的链接，具体表现为在NAT上映射的外网IP:PORT。

A. 源IP不同，忽略其他因素，将映射为不同的Session

B. 源IP相同，源端口不同，将映射为不同的Session

C. 源IP相同，源端口相同，目的IP相同，目的端口不同，将映射为相同的Session

D. 源IP相同，源端口相同，目的IP不同，忽略目的端口，对于不同的NAT可分为不同的情况

a) Cone NAT：将映射为相同的Session

b) Symmetric NAT：将映射为不同的Session

以下对四种NAT类型分别予以说明：

（一）全锥形NAT（Full Cone）:IP、端口都不受限。只要客户端由内到外打通一个洞之后（NatIP:NatPort -> A:P1），其他IP的主机(B)或端口(A:P2)都可以使用这个洞发送数据到客户端。映射关系为：Client->NatIP:NatPort->Any，即任何外部主机都可通过NatIP:NatPort发送数据到Clietn上。

 （二）受限锥形NAT（Restricted Cone）：

IP受限，端口不受限。当客户端由内到外打通一个洞之后(NatIP:NatPort -> A:P1)，A机器可以使用他的其他端口（P2）主动连接客户端，但B机器则不被允许。映射关系为：Client-> NatIP:NatPort->A，即只有来自A的数据包才能通过NatIP:NatPort发送到Client上。

（三）端口受限锥型(Restricted Port Cone)：IP、端口都受限。返回的数据只接受曾经打洞成功的对象（A:P1），由A:P2、B:P1发起的数据将不被NatIP:NatPort接收。映射关系为：Client->NatIP:NatPort->A:P1，即只有来自A:P1的数据才可通过NatIP:NatPort发送到Client上。

（四）对称型NAT（Symmetric NAT）：对称型NAT具有端口受限锥型的受限特性。但更重要的是，他对每个外部主机或端口的会话都会映射为不同的端口（洞）。只有来自相同的内部地址（IP:PORT）并且发送到相同外部地址（X:x）的请求，在NAT上才映射为相同的外网端口，即相同的映射。一个外部地址（X:x）对应一个NAT上的映射，如上图红色三角，每个映射仅接收来自他绑定的外部地址的数据。注：X在这里意为任意一台外部主机，x为这台主机上的任意一个端口。映射关系为：Client->NatIP:Pa1->A:P1，当Client访问B:P1时，映射关系变为：Client->NatIP:Pb->B:P1，同理，NatIP:Pa2也就是Client访问A:P2时的映射。

3.穿透性

NAT类型简写对应：全锥形->FC，限制锥形->RC，端口限制锥形->PC，对称型->SN。还需要说明的是：下文所提到的“双方建立连接”意为双方在打洞之后而进行的直接的数据传输，而不是通常认为的“TCP是基于连接的“或是“UDP是无连接的协议”这样的概念。

（一）   其中一个具有公网地址时。无论对方是什么类型的Nat都能打通。

（二）   其中一个是FC时，都可以实现p2p通信。

（三）   其中一个是RC时，如果另外一个nat是SN型的，则不能建立通信。除此之外的都能建立通信。

（四）   当PC->PC时，能建立通信。

（五）   SN对SN，不能建立通信。

**Libjingle特点**

1. Signal-Slot机制

sigslot简单说，signal是事件源，而slot是事件处理方法，sigslot机制通过弱耦合的方式把属于两个类对象的事件源和事件处理方法联系起来的。

1. 处理同步\异步的多线程处理机制

在Ligjingle，不仅支持异步模式，而且还支持同步模式，如同windows的窗口消息处理机制，它既有PostMessage(异步)，又有SendMessage(同步)。

1. 打洞技术

LibJingle通过XMPP协议很好的支持了打洞，以及P2P传输。

通过XMPP协议，LibJingle打洞，我们可以：

1.多用户语音。

2.多用户视频。

3.多用户文件共享。

4.一对一远程桌面协助。

5.多用户在线音乐流同步。

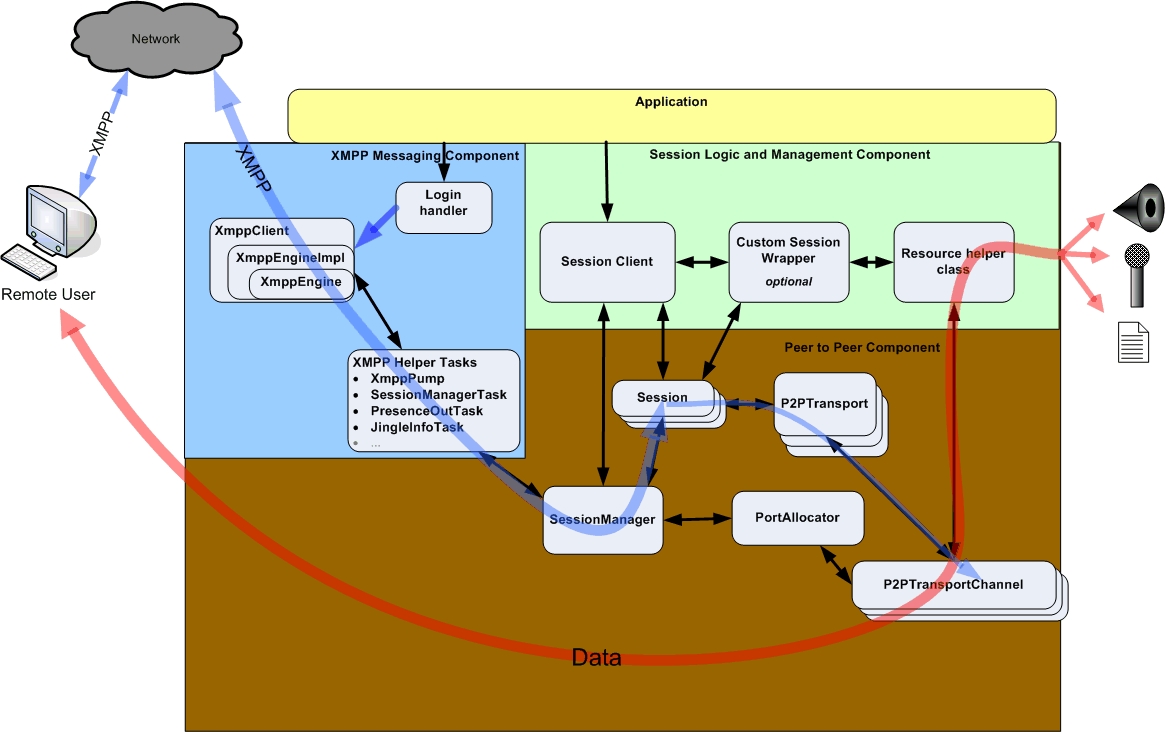
6.还有很多我们可以做到。

1. XMPP以及Gtalk的实现方式

什么事XMPP?

XMPP是一种基于XML的协议，它继承了在XML环境中灵活的发展性。因此，基于XMPP的应用具有超强的可扩展性。经过扩展以后的XMPP可以通过发送扩展的信息来处理用户的需求，以及在XMPP的顶端建立如内容发布系统和基于地址的服务等应用程 序。而且，XMPP包含了针对服务器端的软件协议，使之能与另一个进行通话，这使得开发者更容易建立客户应用程序或给一个配好系统添加功能。

**Libjingle的组件以及运作**



1. Application模块

Ligjingle的应用程序首先调用XMPP Messaging Component的XmppClient对象进行登录，然后做一些message，iq，presence等request/respond操作。

其次，每个application可能包含一个或多个session client用来做P2P操作，比如远程协助，视频会议，音频连接，文件共享等等。

1. XMPP Messaging Component模块

此模块主要由三个部分组成：XmppClient，LoginHandler和Xmpp Helper Task。此模块主要做相当于一个peer的防火墙的功能，连接服务器和客户端，负责发送所有本地的stanza请求（即XMPP协议内容），并负责接收服务器的stanza请求，并分发到各个Helper Task里。

* XmppClient主要是代理登录，发送stanza，接收stanza。之所以说是代理，是因为真正发送，接收的消息都是通过XmppEngine来实现的。
* XmppEngine能注册多个XmppStanzaHandler回调，然后所有从服务器接收的Stanza都转发到已绑定的XmppStanzaHandler进行过滤，而实际上是回调XmppTask对象。
* XmppStanzaHandler类定义如下：

//! Callback to deliver stanzas to an Xmpp application module.

//! Register via XmppEngine.SetDefaultSessionHandler or via

//! XmppEngine.AddSessionHAndler.

class **XmppStanzaHandler** {

public:

virtual ~**XmppStanzaHandler**() {}

//! Process the given stanza.

//! The handler must return true if it has handled the stanza.

//! A false return value causes the stanza to be passed on to

//! the next registered handler.

virtual bool **HandleStanza**(const XmlElement \* stanza) = 0;

};

* XmppTask是所有XmppHelperTask的基类，并继承自XmppStanzaHandler，主要有监听，过滤XmppEngine对象转发过来的Stanza消息。XmppTask有多种类型，当取类型为HL\_PEEK时，只有监听功能，无法做到过滤；而其他类型可以做到过滤。过滤是有HandlerStanza函数来完成，当返回为true时，过滤，否则XmppEngine枚举下一个绑定的XmppTask继续尝试分发、过滤。
* 所有XmppHelperTask都要继承XmppTask并要重载HandlerStanza函数和ProcessStart函数。
  + HandlerStanza是用来过滤，相当于windows消息处理的GetMessage()
  + 而ProcessStart是用来处理HandlerStanza过滤的消息。
  + 比如在源代码example/call/presencepushtask.h里：

class **PresencePushTask** : public **XmppTask** {

public:

**PresencePushTask**(XmppTaskParentInterface\* parent, CallClient\* client)

: **XmppTask**(parent, XmppEngine::HL\_TYPE),

client\_(client) {}

virtual int **ProcessStart**();

sigslot::signal1<const Status&> **SignalStatusUpdate**;

sigslot::signal1<const Jid&> **SignalMucJoined**;

sigslot::signal2<const Jid&, int> **SignalMucLeft**;

sigslot::signal2<const Jid&, const MucStatus&> **SignalMucStatusUpdate**;

protected:

virtual bool **HandleStanza**(const XmlElement \* stanza);

void **HandlePresence**(const Jid& from, const XmlElement \* stanza);

void **HandleMucPresence**(buzz::Muc\* muc,

const Jid& from, const XmlElement \* stanza);

static void **FillStatus**(const Jid& from, const XmlElement \* stanza,

Status\* status);

static void **FillMucStatus**(const Jid& from, const XmlElement \* stanza,

MucStatus\* status);

private:

CallClient\* client\_;

};

* + 这里PresencePushTask类，通过HandleStanza过滤所有presence相关的stanza并在ProcessStart里处理所有来自服务器的用户状态更新消息。
* LoginHandler部分是由XmppPump来负责的。主要调用XmppClient的connect和disconnect方法建立、断开连接，监听SignalStateChange事件来获取连接信息，类型为STATE\_OPENED的事件表示连接成功。

1. Session Logic and management commponent模块

所有p2p session逻辑相关的部分都放在了这个模块。可以session可能是处理文件传输的连接，或者可能是视频会话，或者音频会话等等。

* 我们需要继承SessionClient来处理每个Session相关具体任务，比如文件传输Session：当接收对端客户端建立一个文件传输session的时候，如果此Session是新创建的，SessionManager对象会回调所有注册的SessionClient的OnSessionCreate的接口，并以SessionManger创建的Session对象为参数穿进去;如果是已有的Session则会调用Session的OnIncomingMessage方法。
* Session对象则抽象了两个peer之间的数据传输接口。当收到OnSessionCreate回调时，SessionClient可以通过Session的方法Accept来接受创建，Reject来拒绝。
* 那怎么读写p2p数据呢？

首先需要调用session的CreateChannel方法获取TransportChannel对象指针，然后监听TransportChannel的事件SignalReadPacket来接收数据，通过SendPacket方法来发送数据。

class **TransportChannel**: public sigslot::has\_slots<> {

public:

//......

// Attempts to send the given packet. The return value is < 0 on failure.

virtual int **SendPacket**(const char \*data, size\_t len) = 0;

// Signalled each time a packet is received on this channel.

sigslot::signal3<TransportChannel\*, const char\*, size\_t> **SignalReadPacket**;

//......

};

1. Peer to peer Component模块

此模块才是libjingle核心，libjingle项目的初衷也是能够把模块设计得完美，使得所有需要通过P2P传输数据的应用层调用libjingle时，不用担心数据传输的稳定性，可靠性，高效性。

* 刚才上面提到，当服务器发送stanza时XmppEngine把Stanza发送到XmppTask过滤，在这个模块，SessionManagerTask代理SessionManager过滤session相关的stanza，并转发到SessionManager对象，如下：

class **SessionManagerTask** : public buzz::**XmppTask** {

public:

......

virtual int **ProcessStart**() {

const buzz::XmlElement \*stanza = NextStanza();

if (stanza == NULL)

return STATE\_BLOCKED;

session\_manager\_->OnIncomingMessage(stanza);

return STATE\_START;

}

protected:

virtual bool **HandleStanza**(const buzz::XmlElement \*stanza) {

if (!session\_manager\_->IsSessionMessage(stanza))

return false;

QueueStanza(stanza);

return true;

}

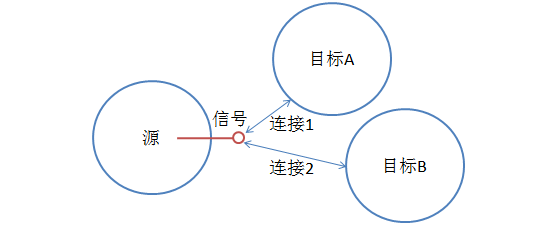
} // namespace cricket

* SessionManager类在这里起到连接上述3个模块的桥梁作用。
* 当上层调用SessionManager创建的Session对象的CreateChannel时，实际上是调用P2PTransport的CreateChannel方法。
* 上层通过P2PTransport创建P2pTransportChannel的类的。
* P2pTransportChannel继承自TransportChannel，并创建多个不同的Connection，每个Connection代表一个TCP或者UDP或者SSL连接。上层传输数据最终是调到P2pTransportChannel的相关方法，当发送，接收数据时，P2pTransportChannel选择表现最好的Connection进行传输。

**Libjingle知识点**

1. Signals机制

Signal机制的工作方式参见下图的描述。源中设置一个或多个信号，目标为了在源的信号触发时获得通知，需要连接到信号上。可以有多个目标发起连接，也可以同一个目标发起多个连接。连接创建好之后，源触发信号时，目标A和目标B就可以收到信号触发的消息了。



libjingle中，源和目标都是对象，目标对象的类必须继承于sigslot::has\_slots，信号是源对象的成员变量，必须是sigslot::signal?类型，其中?代表参数的个数（支持0到8），比如要设置带两个参数的信号，那么就是类型sigslot::signal2。要注意的是，libjingle中所有的信号变量为了方便，名称都加了前缀Signal。在目标对象初始化的时候，需要通过sigslot::signal?的connect函数将目标对象上的回调函数连接到信号上。当源对象触发信号时，目标函数的对应的回调函数会被一次调用。需要注意的是，回调函数的参数类型和个数需要和sigslot::signal?申明的一样。

Signal机制的使用很简单，下面是libjingle文档中的例子。Sender代表的是源，Sender中的SignalDanger代表的是信号，是sigslot::signal2类型的变量。Receiver代表的是目标，继承自sigslot::has\_slots。Receiver的构造函数中，通过调用SignalDanger的connect函数，连接函数OnDanger到SignalDanger上，当Sender调用Panic函数触发信号SignalDanger时，会转调到Receiver::OnDanger函数中。

// Class that sends the notification.

class **Sender** {

// The signal declaration.

// The '2' in the name indicates the number of parameters. Parameter types

// are declared in the template parameter list.

sigslot::signal2<string message, std::time\_t time> **SignalDanger**;

// When anyone calls Panic(), we will send the SignalDanger signal.

void **Panic**(){

**SignalDanger**("Help!", std::time(0));

}

// Listening class. It must inherit sigslot.

class **Receiver** : public sigslot::has\_slots<>{

// Receiver registers to get SignalDanger signals.

// When SignalDanger is sent, it is caught by OnDanger().

// Second parameter gives address of the listener function class definition.

// First parameter points to instance of this class to receive notifications.

**Receiver**(Sender sender){

**sender**->**SignalDanger**.**connect**(this, &**Receiver**.**OnDanger**);

}

// When anyone calls Panic(), Receiver::OnDanger gets the message.

// Notice that the number and type of parameters match

// those in Sender::SignalDanger, and that it doesn't return a value.

void **OnDanger**(string message, std::time\_t time){

if(message == "Help!")

{

// Call the police

...

}

}

...

}

1. Thread

libjingle考虑到使用到此库的应用程序的性能，libjingle内部支持多线程。其内组件使用1或2个全局线程：

* signaling thread 被用作创建底层（基础）组件，

例如：Session Management,Control,XMPP Messaging组件。

* worker thread (有时称作channel thread)用来集中处理p2p组件中的对象提交过来的大量资源，例如：数据流。之所以这样用另外的线程单独处理，是为了避免数据流阻塞或被XMPP/用户界面组件阻塞。使用 worker thread的类包括ChannelManage,SocketMonitor,P2PTransportChannel和 属于Port类的对象。

若起用worker thread,使之工作，在应用中必须创建一个Thread类对象，并把此对象当作SessionManager的构造函数的参数。（如果SessionManager类对象在创建时，没有传递给它Thread对象，则SessionManager类将在内部创建一个线程，当作worker thread)。

注意：尽管libjingle支持多线程，但是只有几个函数通过呼叫方线程的验证来支持线程安全，并且极少函数做了线程锁定。下面的片断示范了在函数中如何安全地呼叫线程（或线程安全地被呼叫）：

// Check that beingcalled from the channel (e.g., worker) thread.

ASSERT(talk\_base::Thread::Current()== channel\_thread\_);

channel\_thread\_->Clear(this);

libjingle中用到的所有线程，signaling thread，worker thread，其它的一些线程，都是talk\_base::Thread的对象（或子类的对象）。所有的Thread对象都被ThreadManager管理，当被请求时，ThreadManager会返回这些Thread对象。SessionManager被创建时通过调用ThreadManager::CurrentThread得到一个signal thread(当无worker thread 传递给SessionManager构造函数时，同时得到一个work thread)。所以，应用程序必须为signal thread创建一个Thread 对象（或其子类对象），并在SessionManager对象创建之前或在XmppPump工作之前，把此对象放进ThreadManager的线程池里。

Thread继承自MessageQueue，所以Thread的对象具有了Send，Post，和一些同步或异步发送消息的函数。如果要使一个对象能够接收到MessageQueue送出的消息，那么此对象必须继承和实现MessageHandler。MessageHandler定义了一个OnMessage函数，此函数在MessageQueue送出消息时被调用，用来接收MessageQueue送出的消息。

可以通过任何线程向继承自talk\_base::MessageHandler的任何对象发送消息。尽管能够做到，如果发出的消息是为了集中处理大量的数据，应用程序应该通过worker thread。调用SessionManager::worker\_thread()可以得到workerthread的句柄。调用Session::Manager::signaling\_thread()可以得到signaling thrread的句柄。

对象使用一个指定的线程有如下几种方式：

对象要求一个线程指针作输入参数，并储存这个指针。对象在创建时取得当前线程（构造函数中调用ThreadManager::CurrentThread()取得），把取得的线程存进对象内部成员变量引用它，一般应用于获取特定的线程。对象调用SessionManger::signal\_thread()或SessionManager::worker\_thread()获取线程。

由于一个对象可以被任意线程使用，对象可能需要验证当前调用是来自哪个线程的方法。应用可以调用Thread::Current（）得到当前线程的句柄，然后与对象内部保存线程的数据成员进行比较，此数据成员的值可以是从SessionManager中暴露在外面的线程，或是对象在创建时通过构造函数传进去的初始化值。

这是一个对象通过其它线程调用自身函数时而广范使用的范例：

// Note that worker\_thread\_ is not initialized until someone

// calls PseudoTcpChannel::Connect

// Also note that this method \*is\* thread-safe.

bool PseudoTcpChannel::Connect(const std::string& channel\_name) {

ASSERT(signal\_thread\_->IsCurrent());

CritScope lock(&cs\_);

if (channel\_)

return false;

ASSERT(session\_ != NULL);

worker\_thread\_ = session\_->session\_manager()->worker\_thread();

...

}

void PseudoTcpChannel::SomeFunction(){

...

// Post a message to yourself over the worker thread.

worker\_thread\_->Post(this, MSG\_PING); // <- Goes in here....

...

}

// Handle queued requests.

void PseudoTcpChannel::OnMessage(Message \*pmsg) {

if (pmsg->message\_id == MSG\_SORT)

OnSort();

else if (pmsg->message\_id == MSG\_PING) // -> And comes out here!

// Check that we're in the worker thread before proceding.

ASSERT(worker\_thread\_->IsCurrent());

OnPing();

else if (pmsg->message\_id == MSG\_ALLOCATE)

OnAllocate();

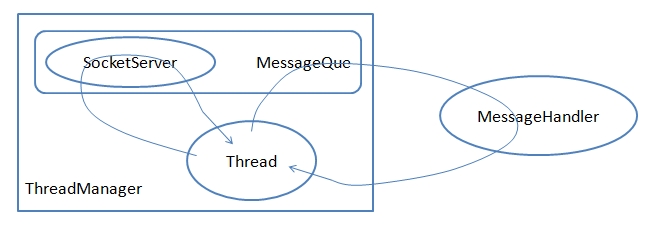
else

assert(false);

}

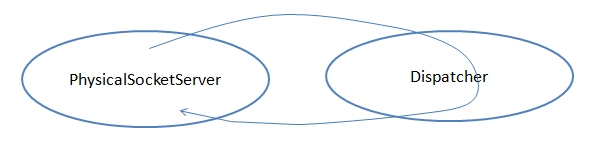
1. Thread和SocketServer

libjingle源码中，Thread和SocketServer模块的原理如下图所示。整个模型实际上是一个消息模型，Thread主要负责处理消息，MessageQueue表示的是当前的消息队列，MessageHandler由用户用来定义处理消息的动作。而ThreadManager为单实例，可以获取当前的Thread，这样用户可以往当前的Thread中投递消息。SocketServer代表的是用来侦听Socket的服务，它是一个独立的模块。



消息的处理流程主要由Thread负责。上图中有两条处理流程，分别用两根带箭头的线表示。左边箭头的处理流程为：当消息队列中没有消息时，Thread将控制权转交给SocketServer，直到有消息时会通知SocketServer返回到Thread。也就是说Thread优先的是处理消息，在空闲时，会让SocketServer侦听socket。右边的箭头是正常的消息处理流程，获取消息并处理用户定义的对应的OnMessage函数。

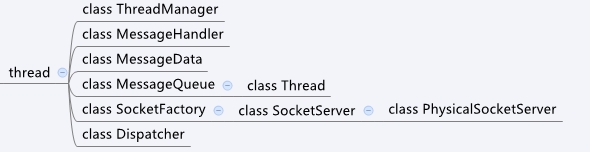
SocketServer模块只是用在libjingle内部，用户并不需要直接使用它。P2P中使用了PhysicalSocketServer作为SocketServer，它的原理如下图所示：



PhysicalSocketServer主要是侦听基于本地网卡的socket（libjingle中还有一些伪socket），然后分发socket事件到Dispatcher中。Dispatcher是PhysicalSocketServer的分发体，功能有点类似于MessageHandler。Dispatcher中定义了感兴趣的socket事件和对应的处理。

类的关系

本文提到的一些类的关系如下图所示。Thread类继承自MessageQue，可以通过Thread类来操作队列消息。PhysicalSocketServer除了实现SocketServer接口之外，还可以添加和删除Dispatcher。熟悉了这些类，基本上就了解Thread模块和SocketServer模块的工作原理。

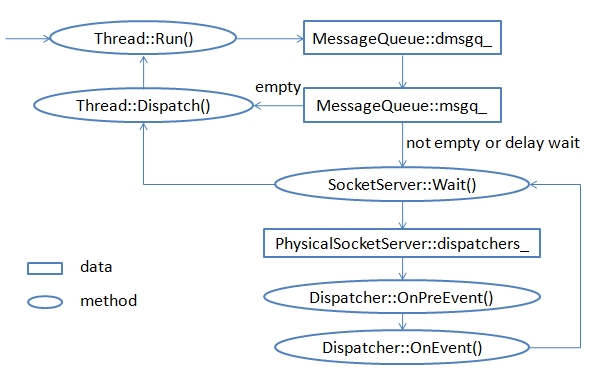


Thread的使用

获取当前线程是通过Thread的Current函数，它会转调ThreadManager对象的CurrentThread函数。由于Thread继承自MessageQue，可以直接通过Thread对象来投递消息，Post函数的第一个参数是OnMessage所处的对象，会被保存于Message对象中。处理消息只要重载MessageHandler的OnMessage函数即可。main函数则调用Thread的Run函数进入默认的消息处理循环，默认的消息处理循环在本示例中就是：循环取消息，调用MessageHandler的OnMessage函数。

处理消息

Thread的默认消息处理流程可用下图表示。默认消息处理函数的入口为Thread::Run()，另一个内嵌循环是SocketServer::Wait()。箭头指向数据成员则表示，处理相关数据。



**P2P中使用的类**

1. 登录服务器主要相关的类

class P2PHelper : public sigslot::has\_slots<>,

public Ip2pHelper,

public talk\_base::MessageHandler

class Jid

class Thread : public MessageQueue

Thread，一种是signaling thread，也就是管理XMPP消息处理的主线程。另外一种是work thread，也叫channel thread。主要用来进行P2P数据传输的。libjingle都用talk\_base::Thread来包含一个Thread；Thread中的Start()和Run()，Start是新起一个线程，马上开始运行，Run是在本身这个线程里面开始消息循环，是被阻塞的。

class XmppThread : public talk\_base::Thread,

public sigslot::has\_slots<>,

public XmppPumpNotify,

public talk\_base::MessageHandler

XmppThread就是signaling thread。

class XmppPump : public talk\_base::MessageHandler,

public talk\_base::TaskRunner

Libjingle把XmppPump作为Loginhandler。XmppPump，顾名思义就是一整个系统的动力源。由这个pump源源不断地把Message送给等待接受这些Message的地方。

class TaskRunner : public TaskParent,

public sigslot::has\_slots<>

class TaskParent

class XmppClient : public XmppTaskParentInterface,

public XmppClientInterface,

public sigslot::has\_slots<>

XmppClient是在XmppPump的控制下，可以认为它是XmppPump的一部分。它负责从接收和发送Xmpp格式的消息。

class XmppTaskParentInterface : public talk\_base::Task

class Task: public TaskParent

class XmppTask : public XmppTaskBase,

public XmppStanzaHandler,

public sigslot::has\_slots<>

是XMPP任务对象，由XmppPump管理，以Task结尾的类都是线程类。Task订阅了自己感兴趣的消息.来了消息就Pump到对应的Task里面去了。

class XmppTaskBase : public XmppTaskParentInterface

class XmppEngineImpl : public XmppEngine

class XmppEngine

是协助XmppClient工作的。

class AsyncSocket

异步socket

class SocketAddress

保存socket地址

class XmppSocket : public buzz::AsyncSocket,

public sigslot::has\_slots<>

class PhysicalSocketServer : public SocketServer

class SocketServer : public SocketFactory

class Signaler : public EventDispatcher

class EventDispatcher : public Dispatcher

事件分发（通过消息机制post和send实现异步通知）

class Dispatcher

class SocketDispatcher : public Dispatcher,

public PhysicalSocket

网络socket（通过网络接收和发送数据）

class PhysicalSocket : public AsyncSocket,

public sigslot::has\_slots<>

在physicalsocket继承自asyncsocket，是套接字服务器，提供了真正的底层操作接口