Sigslot学习 - longrenle - 博客园

一直在搞WebRTC，发现其Web API还很不成熟，Chrome的团队也在不停地fix bug，于是下载了WebRTC的源码学习。WebRTC的源码一部分已经merge进了libjingle项目，结构比较复杂。

libjingle里面有一个基类为has_slots，搜索了一下其资料发现是一个很好用的C++库。开源库连接：http://sourceforge.jp/projects/sfnet_sigslot/

下面是转载的别人的资料，一个对sigslot简单清晰的介绍,学习分享一下！

1. 简介

sigslot是一个线程安全、类型安全，用C++实现的sig/slot机制(sig/slot机制就是对象之间发送和接收消息的机制)的开源代码库。是一个非常好用的库,只有一个头文件sigslot.h。

2. Sigslot实例

现代的C++项目通常包含大量的C++类和对象，对象之间通过成员函数调用，缺点是当类和对象规模很大时，相互之间必须记住对方提供了哪些接口，以及接口的详细信息，很不方便。

比如：我们有一个switch类和一个light类，而我们现在需要将两者关联起来，即通过switch控制light的状态，我们可能需要添加一个另外的类ToggleSwitch来将两者关联起来：

```
class Switch

{

public:

    virtual void Clicked() = 0;

};

class Light

{

public:

    void ToggleState();

    void TurnOn();

    void TurnOff();
```

```
};

class ToggleSwitch : public Switch

{

public:

    ToggleSwitch(Light& lp){m_lp = lp;}

    virtual void Clicked(){m_lp.ToggleState();}

private:

    Light& m_lp;

};

Light lp1, lp2;

ToggleSwitch tsw1(lp1), tsw2(lp2);
```

这在功能上完全可以实现，但想象一下如果大量的需要相互交互消息的类，那工作量就不是一般的大了。

使用sig/slot机制来解决上述情况，不需要关心关联类的接口细节，sigslot实现的switch和light上述功能如下：

```
class Switch

{

public:

    signal0<> Clicked;

};

class Light : public has_slots<>

{

public:

    void ToggleState();

    void TurnOn();

    void TurnOff();
```

```
};

Switch sw1, sw2;

Light lp1, lp2;
```

Sigslot机制实现该功能与第一种方法相比，switch类多了个signal0成员，light类需要从has_slots<>继承，其他没有什么变化，但省去了编写继承类用来实现两者关联的ToggleSwitch。

下面是实现功能的简单代码。

```
#include <iostream>

using namespace std;

#include "sigslot.h"

using namespace sigslot; //必须加上sigslot的命名空间

//在用vs调试时还需要将sigslot.h中很多的自定义模板结构类型前加typename

const int TRUE = 1;

const int FALSE = 0;

class Switch

{

public:

   signal0<> Clicked;

//这里的信号是不带参数的，signaln表示带几个参数

};

class Light : public has_slots<>

{

public:

Light(bool state){b_state = state;Displaystate();}

      void ToggleState(){b_state = !b_state;Displaystate();} //作为消息的响应

      void TurnOn(){b_state = TRUE;Displaystate();}
```

```
        void TurnOff(){b_state = FALSE;Displaystate();}

        void Displaystate(){cout<<"The state is "<<b_state<<endl;}

    private:

        bool b_state;

    };

    void main()

    {

        Switch sw1, sw2,all_on,all_off;

        Light lp1(TRUE), lp2(FALSE);

        sw1.Clicked.connect(&lp1,&Light::ToggleState); //绑定

        sw2.Clicked.connect(&lp2,&Light::ToggleState);

        all_on.Clicked.connect(&lp1,&Light::TurnOn);

        all_on.Clicked.connect(&lp2,&Light::TurnOn);

        all_off.Clicked.connect(&lp1,&Light::TurnOff);

        all_off.Clicked.connect(&lp2,&Light::TurnOff);

        sw1.Clicked();

        sw2.Clicked();

        all_on.Clicked();

        all_off.Clicked();

        sw1.Clicked.disconnect(&lp1);

        sw2.Clicked.disconnect(&lp2);

        all_on.Clicked.disconnect(&lp1);

        all_on.Clicked.disconnect(&lp2);

        all_off.Clicked.disconnect(&lp1);

    all_off.Clicked.disconnect(&lp2);
```

}

3. 参数类型

sig/slot可以带参数也可以不带，最多可以带8个参数。重新回顾上例，switch类的signal0<> Clicked,称之为sig，即用来发出信号；而继承has_slots<>的类light的成员函数void ToggleState() Turnon() Turnoff(),称之为slot,即信号的处理函数。 sigslot的核心就在这里，就是通过这两个建立对应关系来实现对象间的消息交互。

sig是一个成员变量，它形如

signal+n<type1,type2……>

后面的n表示signal可以接收几个参数，类型任意，最多为8个。这是由库中指定的，当然如果实际开发需要更多的参数，可以修改sigslot库。

slot是一个成员函数，它形如：

void SlotFunction(type1,type2……)

需要记住：slot的类必须继承has_slots<>;成员函数的返回值必须为void类型，这是这个库的局限性，当然如果实际开发需要返回值，也是可以修改sigslot库来实现。此外还需要注意的是slot的原形需要与sig一致。怎么说呢，就是signal只能与带有与它相同参数个数的slot函数进行绑定，而且signal的参数是直接传递给slot的。

4. Sigslot库用法

发送信号

信号(sig，即sig/slot的sig，下面提到的信号等同于此含义)：

signal1<char *, int> ReportError;

比如上面的一个ReportError这个信号，当调用ReportError("Something went wrong", ERR_SOMETHING_WRONG);时候，将自动调用ReportError的emit成员函数发出一个信号。发给谁呢？

连接信息号

通过调用sig的connect函数建立sig和slot间的对应关系。Connect函数接收两个参数，一个是消息目的对象的地址(指针)，另一个是目的对象的成员函数指针(slot)。为了让整个机制有效运行，目的类必须从has_slots<>继承，并且sig/slot参数类型必须一致。也可以将一个sig连接到多个slot上，这样每次sig发出信号的时候，每个连接的slot都能收到该信号。

断开信号连接

通过调用sig的disconnect函数断开sig和slot之间的连接，只有一个参数：目的对象的地址。一般不需要显式调用disconnect函数，在sig类和目的类(包含slot函数的类)析构函数中将自动调用disconnect断开sig和slot的连接。也可使用disconnect_all断开该sig的所有slot。

```
all_on.Clicked.connect(&lp1,&Light::TurnOn);

all_on.Clicked.connect(&lp2,&Light::TurnOn);//同上

all_on.Clicked.disconnect_all();
```

5. Sigslot库范例

在开发一个复杂工程的时候，经常会遇到这样一个问题：整个系统被分成数个模块，每个模块提供有限的功能，由上层调用组成整个系统，为了保证每个模块的独立性，我们经常会尽量限制模块与模块之间的直接联系，比如每个模块只提供有限的API或者COM接口，而内部实现则完全封闭起来。

但有的时候会出一些设计要求，必须能够使模块之间能够直接通讯，而这两个模块往往处于不同的逻辑层次，之间相差甚远，如何设计它们之间的调用模式使整个工程维持整洁变得非常困难，比如模块直接直接包含对方的头文件会引起编译变得复杂，提供api或者接口会引起版本危机等问题。

sigslot的出现为我们提供了一种解决问题的思想，它用“信号”的概念实现不同模块之间的传输问题，sigslot本身类似于一条通讯电缆，两端提供发送器和接收器，只要把两个模块用这条电缆连接起来就可以实现接口调用，而sigslot本身只是一个轻量级的作品，整个库只有一个.h文件，所以无论处于何种层次的库，都可以非常方便的包含它。

举个例子，我们设计一个发送消息的类，这个类负责在某种时刻向外界发出求救信号

```
// Class that sends the notification.
class Sender
{
public:
   // The signal declaration.
   // The ’2′ in the name indicates the number of parameters. Parameter types
   // are declared in the template parameter list.
   sigslot::signal2<std::string ,int >SignalDanger;

   // When anyone calls Panic(), we will send the SignalDanger signal.
   void Panic()
   {
      SignalDanger("Help!",0);
    }
};
```

另外一个类则负责接收求助信号

```
// Listening class. It must inherit sigslot.
```

```
class Receiver :public sigslot::has_slots<>
{
public:
    // 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(std::string message,int time)
    {
        printf("I heard something like\"%s\" at %d!\n",message.c_str(),time);
    }
};
```

现在让我们在主逻辑中把这两个类连接起来

```
Sender sender;
Receiver receiver;

// 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.
sender.SignalDanger.connect(&receiver,Receiver::OnDanger);
```

只要在任何时候调用 sender.Panic()函数，就会把求救信号发送给接收者，而且这两个发送和接收端的模块都可以独立编译，不会出现版本问题。