信号槽介绍

Qt信号槽

信号与槽可以说是qt中最经典的一个设计。

示例：

```
class Button : public QObject{
    Q_OBJECT
public:
    Button(QObject* parent = Q_NULLPTR) :QObject(parent) {};
signals:
    void PressDown();
};

class Label : public QObject{
    Q_OBJECT
public:
    Label(QObject* parent = Q_NULLPTR) :QObject(parent) {};
public slots:
    void OnShow() {qDebug() << "press down";};
};

auto *btn = new Button;
auto *lab = new Label;
QObject::connect(btn, &Button::PressDown, lab, &Label::OnShow);
emit btn->PressDown();
```

1.通过connect信号和槽就进行了绑定，调用信号PressDown()，就会触发槽函数Lable::OnPressDown。

信号和槽要解决的问题：

最容易对比的就是设计这模式中基于接口的观察者模式

基于接口观察者模式来实现同样功能：

```
//设计事件
class IEvent {
public:
   virtual  void OnPressDown() = 0;
};

class Button{
public:
    void SetIEvent(IEvent* event) { event_ = event;}
    void PressDown() {
        if (event_) event_->OnPressDown();
    };
private:
    IEvent* event_ = nullptr;
};
```

```
//接收事件
class Label : public IEvent {
public:
    void OnPressDown() override { std::cout << "press down\n"; };
};

auto* btn = new Button;
auto* lab = new Label;
btn->SetIEvent(lab);
btn->PressDown();
btn->SetIEvent(NULL);
delete lab;
```

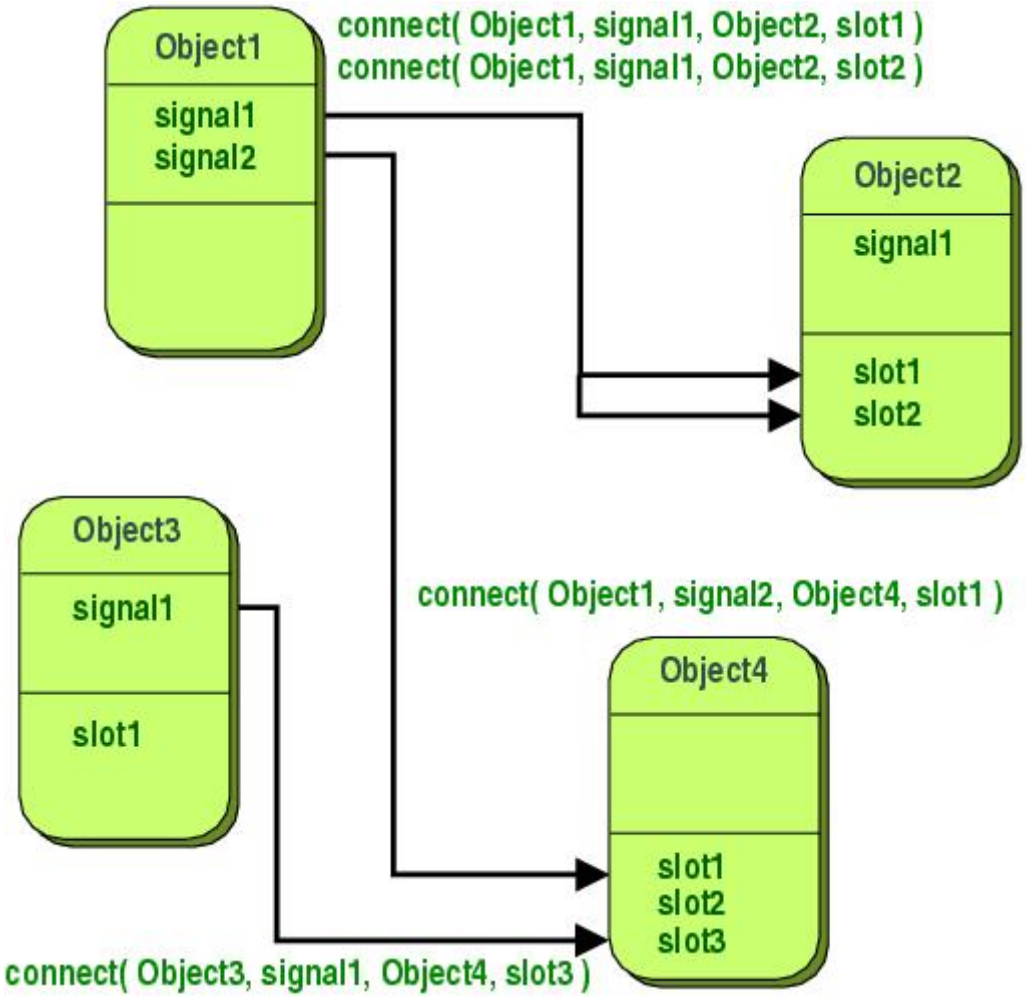
对比与思考：

- 对于设计者而言，接口可以很好的进行功能的抽象并解耦解耦，但对于使用者而言，接口的使用并不是特别的友好，会有以下的问题：

1. 使用需要继承接口，要针对接口去实现功能。
2. 接口名不能改变，接口名变化，对应模块的实现名也要更改。
3. 多接口继承时会感觉非常复杂，如果出现重名，需要再进行一层封装，再进行转发。
4. 需要对生命周期进行管理，析构时需要相互通知，解除对对方的持有。

- 接口是设计者和使用者之间的桥梁，而这个桥梁比较固定，需要相互持有彼此，因此产生了耦合，而信号槽能作为一个第三者彻底的解耦；

  信号和槽是观察者模式的升级版，相对于观察者模式确实能使模块间更加的解耦，模块拼接可以更加的自由：



qt的信号槽确实好用，但是只能针对于qt，并不能从qt剥离，有没有很快的方式实现一套自己的信号槽。

webrtc信号槽的实

实现的非常简洁易懂，实现逻辑是通过变参模板保存对象指针和对象的方法地址。

使用的方法：

```
#include "sigslot.h"
class Button {
 public:
  sigslot::signal1<int> PressDown;//信号就是一个模板类，直接在对象中申明
};

//可以链接任意参数匹配的普通函数，只要类继承sigslot::has_slots<>
class Label : public sigslot::has_slots<>{
 public:
  void OnShow(int i) { std::cout << "press down" << i; };
};

int main(int argc, char* argv[]) {
  auto* btn = new Button;
  auto* lab = new Label;
  btn->PressDown.connect(lab, &Lable::OnShow); //信号链接
  btn->PressDown(2); //信号发送
  return 0;
}
```

可以看到跟qt的信号槽一样非常简洁

- 实现原理

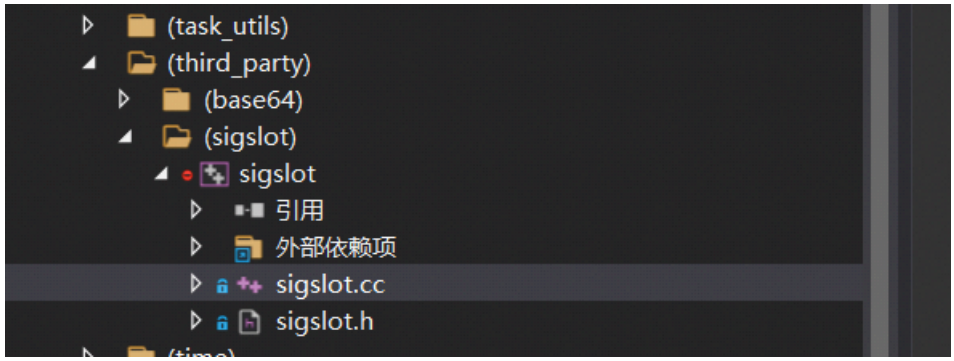
```
//多参数模板
template <typename... Args>
using signal = signal_with_thread_policy<SIGSLOT_DEFAULT_MT_POLICY, Args...>;

template <typename mt_policy = SIGSLOT_DEFAULT_MT_POLICY>
using signal0 = signal_with_thread_policy<mt_policy>;

template <typename A1, typename mt_policy = SIGSLOT_DEFAULT_MT_POLICY>
using signal1 = signal_with_thread_policy<mt_policy, A1>;
```

```
  signal_with_thread_policy() {}
//链接时通过m_connected_slots保存了pclass指针和对应的pclass方法pmemfun
  template <class desttype>
  void connect(desttype* pclass, void (desttype::*pmemfun)(Args...)) {
    lock_block<mt_policy> lock(this);
    this->m_connected_slots.push_back(_opaque_connection(pclass, pmemfun));//信号持有槽函数对象指针和槽函数的地址。
    pclass->signal_connect(static_cast<_signal_base_interface*>(this));    //槽函数对象保存信号。
                                                                           //相互持有析构的时候能够删除掉链接。
  }
```

```
//执行时直接遍历m_connected_slots执行保存方法函数指针
  void emit(Args... args) {
    lock_block<mt_policy> lock(this);
    this->m_current_iterator = this->m_connected_slots.begin();
    while (this->m_current_iterator != this->m_connected_slots.end()) {
      _opaque_connection const& conn = *this->m_current_iterator;
      ++(this->m_current_iterator);
      conn.emit<Args...>(args...);
    }
  }
```

```
//has_slots 作用就是生命周期的管理，当槽函数对象析构时，能够断开信号槽。

template <class mt_policy = SIGSLOT_DEFAULT_MT_POLICY>
class has_slots : public has_slots_interface, public mt_policy {
  ~has_slots() { this->disconnect_all(); }
 private:
  static void do_signal_connect(has_slots_interface* p,
                                _signal_base_interface* sender) {
    has_slots* const self = static_cast<has_slots*>(p);
    lock_block<mt_policy> lock(self);
    self->m_senders.insert(sender);
  }

  static void do_signal_disconnect(has_slots_interface* p,
                                   _signal_base_interface* sender) {
    has_slots* const self = static_cast<has_slots*>(p);
    lock_block<mt_policy> lock(self);
    self->m_senders.erase(sender);
  }

  static void do_disconnect_all(has_slots_interface* p) {
    has_slots* const self = static_cast<has_slots*>(p);
    lock_block<mt_policy> lock(self);
    while (!self->m_senders.empty()) {
      std::set<_signal_base_interface*> senders;
      senders.swap(self->m_senders);
      const_iterator it = senders.begin();
      const_iterator itEnd = senders.end();

      while (it != itEnd) {
        _signal_base_interface* s = *it;
        ++it;
        s->slot_disconnect(p);
      }
    }
  }

 private:
  sender_set m_senders;
};
```

- 代码位置：



webrtc信号槽实现非常简单明了，非常的好提取，就一个头文件，直接拷贝就能使用。但相对于qt的信号槽，不支持抛线程，不支持异步调用，不支持Lambda；

webrtc扩展Lambda表达式的支持

webrtc的实现可以很容易扩展到lambda表达式的支持。

示例：

```
template <class mt_policy, typename... Args>
class signal_with_thread_policy : public _signal_base<mt_policy> {
  using Func = std::function<void(Args...)>;
  std::vector<Func> vec_func_slots_;
  int connect(const Func&& fn) {
    vec_func_slots_.push_back(fn);
    return vec_func_slots_.size() - 1;
  }
  int disconnect(int index) {
    vec_func_slots_.erase(index);
    return;
  }
```

添加一个vec_func_slots_存储function，connect的时候返回一个索引，disconnnect的时候通过索引删除。

这样我们就可以这样使用了

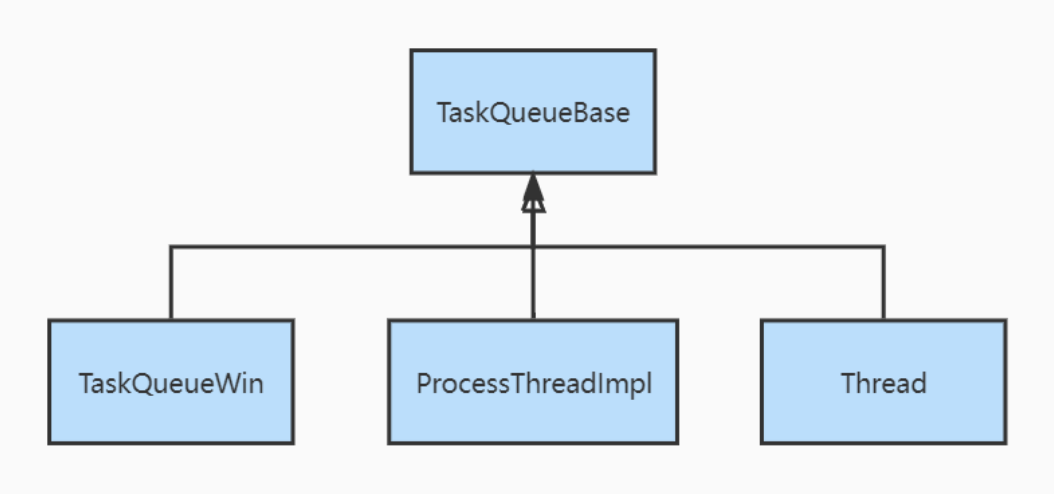
```
#include "sigslot.h"
class Button {
 public:
  sigslot::signal1<int> PressDown;
};

int main(int argc, char* argv[]) {
  auto* btn = new Button;
  auto index = btn->PressDown.connect([](int i) {
      std::cout << "press down" << i;
      });
  btn->PressDown(2);
  btn->PressDown->disconnect(index);
  return 0;
}
```

qt信号槽的扩展

*QObject::connect*的第*5*个参数可以看出*qt*对型号槽的扩展

1. Qt::AutoConnection： 默认值，使用这个值则连接类型会在信号发送时决定。如果接收者和发送者在同一个线程，则自动使用Qt::DirectConnection类型。如果接收者和发送者不在一个线程，则自动使用Qt::QueuedConnection类型。
2. Qt::DirectConnection：槽函数会在信号发送的时候直接被调用，槽函数运行于信号发送者所在线程。效果看上去就像是直接在信号发送位置调用了槽函数。这个在多线程环境下比较危险，可能会造成奔溃。
3. Qt::QueuedConnection：槽函数在控制回到接收者所在线程的事件循环时被调用，槽函数运行于信号接收者所在线程。发送信号之后，槽函数不会立刻被调用，等到接收者的当前函数执行完，进入事件循环之后，槽函数才会被调用。多线程环境下一般用这个。
4. Qt::BlockingQueuedConnection：槽函数的调用时机与Qt::QueuedConnection一致，不过发送完信号后发送者所在线程会阻塞，直到槽函数运行完。接收者和发送者绝对不能在一个线程，否则程序会死锁。在多线程间需要同步的场合可能需要这个。
5. Qt::UniqueConnection：这个flag可以通过按位或（|）与以上四个结合在一起使用。当这个flag设置时，当某个信号和槽已经连接时，再进行重复的连接就会失败。也就是避免了重复连接。

webrtc线程调用的实现



- TaskQueueBase实现

```
class RTC_LOCKABLE RTC_EXPORT TaskQueueBase {
virtual void Delete() = 0;
virtual void PostTask(std::unique_ptr<QueuedTask> task) = 0;
virtual void PostDelayedTask(std::unique_ptr<QueuedTask> task,
                             uint32_t milliseconds) = 0;
                             }
}
```

- PcocessThread实现

```
bool ProcessThreadImpl::Process() {
    //遍历module执行process
    for (ModuleCallback& m : modules_) {
       m.module->Process();
```

```
    }
    //任务队列
    while (!queue_.empty()) {
        QueuedTask* task = queue_.front();
        queue_.pop();
        mutex_.Unlock();
        if (task->Run()) {
            delete task;
        }
        mutex_.Lock();
    }
    //wake_up_基于事件模型的等待同步
    int64_t time_to_wait = next_checkpoint - rtc::TimeMillis();
    if (time_to_wait > 0)
    wake_up_.Wait(static_cast<int>(time_to_wait));
}
```

- Thread 网络

```
explicit Thread::Thread(SocketServer* ss):ss_(ss);

bool Thread::ProcessMessages(int cmsLoop) {
  while (true) {
    Message msg;
    if (!Get(&msg, cmsNext))
      return !IsQuitting();
    Dispatch(&msg);

    if (cmsLoop != kForever) {
      cmsNext = static_cast<int>(TimeUntil(msEnd));
      if (cmsNext < 0)
        return true;
    }
  }
}

bool Thread::Get(Message* pmsg, int cmsWait, bool process_io) {
    {
      // Wait and multiplex in the meantime
      if (!ss_->Wait(static_cast<int>(cmsNext), process_io))
        return false;
    }
}

void Thread::WakeUpSocketServer() {
  ss_->WakeUp();
}
```

同步调用异步调用

```
void PostTask(std::unique_ptr<QueuedTask> task);
ReturnT Invoke(const Location& posted_from, FunctionView<ReturnT()> functor);
```

总结:

1.qt的信号槽确实功能强大，将线程调用方式进行了抽象统一，但是却不能移植出来，需要依赖qt的moc机制。

2.webrtc信号槽实现简洁，容易移植，跨线程调用这些也有非常好的实现。