用Nao的C++SDK写一个最简单的拓展库

此节我将手把手带你们写一个最简单的nao的C++拓展库

建立一个空的项目

```
cd ~/nao/nao_src
qisrc create demo
```

输入以上命令之后，会生成 `demo` 文件夹， `demo` 文件夹下，会有四个文件

- test.cpp
- qiproject.xml
- main.cpp
- CMakeLists.txt

其中 `qiproject.xml` 和 `CMakeLists.txt` 是构建工程的描述文件，对于项目的编译必不可少，前者是qibuild独有的， `CMakeLists.txt` 则是很多C++工程采用的一个跨平台的安装/编译工具(当然也有其他的，比如Makefile，linux的内核就是用的Makefile编译的，还有BUILD，百度的Apollo就是用的BUILD构建的)。

两个cpp文件自然就是代码了，下面讲讲 `qiproject.xml` 、 `CMakeLists.txt` 、还有代码的编写。

qiproject.xml

对于我们写这个比较简单，我也不太懂它存在的意义，但是删掉它又编译不通过，鸡肋啊。
实际上我们打开这个文件之后，除去注视只有四行。

```
<project version="3">
    <qibuild name="demo">
    </qibuild>
</project>
```

咱们加上一行，告诉他我们的依赖项就是naoqi的库。遂写成如下

```
<project version="3">
    <qibuild name="demo">
        <depends buildtime="true" runtime="true" names="libnaoqi"/>
    </qibuild>
</project>
```

基本上以后**再也不用改** `qiproject.xml` 文件了。

CMakeLists.txt

`CMakeLists.txt` 的正确编写是非常重要的
我们不管它默认给的 `CMakeLists.txt` ，先写一份比较常用的 `CMakeLists.txt` 的套路。

```
# 这里先给定一个最老的cmake的版本，若当前使用的cmake版本比3.0低便不能编译。CMakeLists.txt第一行基本是这个。
cmake_minimum_required(VERSION 3.0)
# 给定工程名，CMakeLists.txt基本是这个，有了这行之后PROJECT_NAME变量便会被自动赋值，之后可以调用之。
project(NaoDemo)
# 让cmake自动给你去寻找你需要的依赖库，这里我们只需要找qibuild，因为qibuild给你把你能用的库找好了，其他不是qibuild的工程里往往就不是这样
# 这一步之后，若编译起找到了qibuild这个库，那么，在你#include一个头文件时，编译器不但会去标准库里找是否有这个头文件，还回去qibuild对应的头文件中找。
find_package(qibuild)
# 为了方便，我们把工程所需要的源文件，放到变量_src中，之后${_src}便代表了现在定义的这些源文件，代码可以更简洁一些
# 我们最后的工程源文件就这俩demo.cpp main.cpp
set(_src demo.cpp main.cpp)
# 我们的目标是建立一个库所以应该调用qi_create_lib函数
# 若要生成一个可执行文件应该调用函数qi_create_bin
# qi_create_lib默认生成静态库，我们需要加一个SHARED的标志，告诉他我们要生成分享库’
qi_create_lib(NaoDemo SHARED ${_src}) # 这里也规定了生成的文件的名字，NaoDemo，它会自动加上后缀.so(shared object)和前缀lib(library)
# 到这里还不够，我们需要告诉编译器我们NaoDemo这个分享库还用了别人的库，要编译器去帮我们把这些库链接到我们的库上
# 具体用了哪些库呢主要包括naoqi的库和opencv的库
# 那么我们怎么知道NaoDemo后面那些东西是这样拼写呢？？？
# 比如你要用opencv，并且用了highgui.hpp这个头文件，那么你就去naoqi_lib这个文件夹下找，你会找到opencv文件夹
# 在opencv文件夹下找到 share/cmake/opencv2_highgui文件夹，这个文件下只有一个文件，这个文件最后一行！！！
# export_lib(OPENCV2_HIGHGUI)
# 这正是我们下面用到的这个名称，其余同理
qi_use_lib(NaoDemo ALCOMMON ALPROXIES ALVISION OPENCV2_CORE OPENCV2_HIGHGUI)
```

至此，编写完了 `CMakeLists.txt` 就这样简单几行，已经够用大多数情况了，有新的情况可以与我交流或者谷歌。

源文件！！！！

上一小部分，可以看出我们的源文件就两个cpp(实际上还有一个头文件)，现在先把自动生成的源文件删了！！

```
rm -rf main.cpp test.cpp
```

新建自己的～

```
touch main.cpp demo.cpp demo.h
```

至此，便可以开始用Qt骚气起来了。

```
qibuild configure
qibuild make
qibuild open
```

qt的操作参考之前的教程。

首先，我们需要知道，我们在做什么，我们在用C++写nao的一个组件，那我们要怎么把我们的组件跟naoqi融合在一起呢？自然要他给我们提供了接口。
naoqi整体的设计模式是一种分布式的模式，可以调用一个固定的接口去注册我们的程序，所以套路都是固定的。那我们要做哪些工作呢？

- 写一个类，这个类继承于 `ALModule` 这个基类，这个基类有一些函数是用来暴露我们的接口的。
- 在构造函数中调用functonName、BIND_METHOD这两个方法把我们想要暴露给naoqi的接口给出去。
- `ALModule` 还有一些虚函数以供重载，最基本的就是virtual void init()这个虚函数，他会在机器人初始化之时调用，具体虚函数是什么一个意思，复习C++。
- 最后在main.cpp中定义int *createModule(boost::shared*ptrAL::ALBroker) 和 int _closeModule()这两个函数，他们会被naoqi调用用来注册模块的，在前者函数中，有一个固定的注册模块的套路，直接复制即可，这也是他们设计模式的固有套路。

接下来先写头文件。

```
#ifndef _DEMO_H
#define _DEMO_H
//第一个头文件必然是之前提到的基类
#include <alcommon/almodule.h>
#include <iostream>
#include <alcommon/albroker.h>
//这个头文件很常用方便向日志打印debug数据
#include <qi/log.hpp>
//用这个做tts
#include <alproxies/altexttospeechproxy.h>
#include <alcommon/almodulecore.h>
//常用boost库
#include <boost/shared_ptr.hpp>
#include <boost/typeof/typeof.hpp>
//互斥锁
#include <boost/thread/mutex.hpp>
namespace AL{// 我懒得写 using namespace AL :)
class NaoDemo : public ALModule
{
    public:
    // 因为是继承的基类，构造函数的传参必要包括基类的构造函数的参数的
        NaoDemo(boost::shared_ptr<ALBroker> pBroker, const std::string& pNam
e);
        // 析构函数 这里重载了基类的析构函数
        ~NaoDemo();
        // 初始化函数，重载了基类的虚函数，在机器人启动时自动调用
        void init();
        // 暴露给naoqi的接口
        void say(const std::string & thing = std::string("hello"));
    private:
        /*avoid conflict*/
        //互斥锁
        boost::shared_mutex say_mutex;
};
}
#endif //DEMO_H
```

头文件很简单，就是一个继承，一些基本函数。

下面是demo.cpp

```cpp
#include "demo.h"

namespace AL{//我懒
//构造函数
NaoDemo::NaoDemo(boost::shared_ptr<AL::ALBroker> broker, const std::string &
name):
//这里要调用下父类的构造函数
    ALModule(broker, name)
{
// 开始调用基类的函数，用来表示该模块的作用
    setModuleDescription("Demo");
// 暴露接口 NaoDemo::say给naoqi
    functionName("say", getName(), "say some thing");
    addParam("thing", "thing to say");
    BIND_METHOD(NaoDemo::say);

}
//析构函数
NaoDemo::~NaoDemo()
{
// 往日志打了点数据，用choregraphe的日志查看器可看到
    qiLogInfo(getName().c_str()) << "Good Bye";
}

void NaoDemo::init()
{
// 我的习惯会让机器人刚启动时报出代码版本
    say("version 0.0.1");
}

void NaoDemo::say(const std::string &thing)
{
// 先不用互斥锁吧:)
    //boost::unique_lock<boost::shared_mutex> _unique_lock(say_mutex);
    // 不说了
    qiLogInfo(getName().c_str()) << "start say: "<<thing;
    // 调用其他的模块时的套路
    ALTextToSpeechProxy tts(getParentBroker());
    // tts
    tts.say(thing);
}
}
```

下面则是main.cpp

```
#include <signal.h>

#include <boost/shared_ptr.hpp>
#include <alcommon/albroker.h>
#include <alcommon/almodule.h>
#include <alcommon/albrokermanager.h>
#include <alcommon/altoolsmain.h>

#include "demo.h"


extern "C"
{
int _createModule(boost::shared_ptr<AL::ALBroker> pBroker)
{
  // init broker with the main broker instance
  // from the parent executable
  AL::ALBrokerManager::setInstance(pBroker->fBrokerManager.lock());
  AL::ALBrokerManager::getInstance()->addBroker(pBroker);


  // create module instances
  AL::ALModule::createModule<AL::NaoDemo>(pBroker,"NaoDemo" );

  return 0;
}

int _closeModule()
{
  return 0;
}

} // extern "C"
```

这两个函数都是套路，以后都这样用即可，这个设计模式我也不怎么懂。但是这里要注意extern "C"这个声明。
有extern "C" 和没有它的代码编译出来的二进制文件是不一样的，怎么不一样可以做个实验，写个 `foo.cpp`

```
int fun(int a, int b){
    return a+b;
}
```

用 `g++` 编译成二进制文件看看

```
g++ -c -o foo.o foo.cpp
nm foo.o
```

结果是 `0000000000000000 T __Z3fooii`

现在加上 `extern "C"` 试试看

```
int fun(int a, int b){
    return a+b;
}
```

```
g++ -c -o foo.o foo.cpp
nm foo.o
```

结果是 `0000000000000000 T _foo` 可以看到区别，我们只能给naoqi一个分享库，并没有给naoqi我们的头文件，c++的函数重载会让二进制代码里的函数名很长很怪， `extern "C"` 就是说按照C的方式编译这一段代码，即在分享库中的名字就是这个函数真正的名字。

在很多c++调用C的库的时候，都需要注意着点哦，比如做图像检测都会去尝试**YOLO**，尝试大名鼎鼎的 `darknet` ，这是用纯C语言写的深度框架，要想把这个框架跟自己的项目结合到一起就需要注意这个问题。

编译

用 `qibuild make` 或者直接用Qt编译都能生成libNaoDemo.so文件。

把代码拷贝到机器人中

先把分享库拷贝到机器人的 `~/naoqi/lib/` 目录(没有就创建)下
这里星号是机器人的ip，密码是nao

```
scp libNaoDemo.so nao@***.**.*.***:naoqi/lib
```

为了让机器人主动去加载这个分享库，我们需要修改一个 `~/naoqi/preferences/autoload.ini` 的配置文件(没有就创建)

```
# autoload.ini
#
# Use this file to list the cross-compiled modules that you wish to load.
# You must specify the full path to the module, python module or program.

[user]
#the/full/path/to/your/liblibraryname.so  # load liblibraryname.so
/home/nao/naoqi/lib/lib

[python]
#the/full/path/to/your/python_module.py   # load python_module.py

[program]
#the/full/path/to/your/program            # load program
#/home/nao/NaoUpdate/init.sh
```

里面注释写得太清楚了，不再解释了！最后重启。

用python sdk调用我们的module

下一次教程将会讲pythonsdk的安装

```
from naoqi import ALProxy
ip = "***.**.*.***"
demo = ALProxy("NaoDemo", ip, 9559)
demo.say("hello world")
```

之后便可听到一句洪亮的hello world