CTK完整教程(OSGI for C++ 实现 C++ Qt 模块化)

后端 (/categories/back/) ctk (/tags/ctk/) qt (/tags/qt/) 2021/03/10

CTK框架实际应用比较可靠，但网上资料很少。本教程围绕 CTK Plugin Framework，探索 C++ 中的模块化技术，并能够基于 CTK 快速搭建 C++ 组件化框架，避免后来的人走弯路。本教程的源码下载地址：项目源代码 (https://github.com/myhhub/CTK-project)。

CTK介绍

CTK (http://www.commontk.org/) 为支持生物医学图像计算的公共开发包，其全称为 Common Toolkit。CTK插件框架的设计有很大的灵感来自OSGi并且使得应用程序由许多不同的组件组合成一个可扩展模型。这个模型允许通过那些组件间共享对象的服务通信。

当前，CTK 工作的主要范围包括：

- DICOM (http://www.commontk.org/index.php/Documentation/Dicom_Overview)：提供了从 PACS 和本地数据库中查询和检索的高级类。包含 Qt 部件，可以轻松地设置服务器连接，并发送查询和查看结果。
- DICOM Application Hosting (http://www.commontk.org/index.php/Documentation/DicomApplicationHosting)：目标是创建 DICOM Part 19 Application Hosting specifications 的 C++ 参考实现。它提供了用于创建主机和托管应用程序的基础设。
- Widgets (http://www.commontk.org/index.php/Documentation/Widgets)：用于生物医学成像应用的 Qt Widgets 集合。
- Plugin Framework (http://www.commontk.org/index.php/Documentation/Plugin_Framework)：用于 C++ 的动态组件系统，以 OSGi 规范为模型。它支持一个开发模型，在这个模型中，应用程序（动态地）由许多不同（可重用的）组件组成，遵循面向服务的方法。
- Command Line Interfaces (http://www.commontk.org/index.php/Documentation/Command_Line_Interface)：一种允许将算法编写为自包含可执行程序的技术，可以在多个终端用户应用程序环境中使用，而无需修改。

CTK Plugin Framework

CTK Plugin Framework 架构策略

- Qt Creator 的可扩展性。
  Qt Creator 通过一种简单、优雅的方式来实现可扩展性，它使用一个通用的 QObject 池来实现某些可用的接口。同时，通过使用嵌入式文本文件（.pluginspec 文件）来向插件添加元数据（例如：Name、Version 等）。

其实严格来说，CTK Plugin Framework 同时借鉴了 OSGi 和 Qt Creator 的思想。

- Qt 提供了 Qt Plugin System 和 Qt Service Framework。
  Qt Plugin System 提供了两套用于创建插件的 API，高级 API 用于扩展 Qt 本身（例如：自定义数据库驱动、图像格式、文本编解码、自定义样式等），低级 API 用于扩展 Qt 应用程序。

对于 Qt Service Framework 来说，它能使服务的开发和访问方式变得更加容易。Qt 服务提供者可以与特定于平台的服务进行交互，而无需向客户端公开平台的细节。每个服务都通过 QObject 指针公开，这意味着客户端可以通过 Qt MetaObject 系统与服务对象进行交互。

- CTK Plugin Framework 的架构策略是什么?
  CTK Plugin Framework 是基于 Qt Plugin System 和 Qt Service Framework 实现的，并且它还添加了以下特性来增强这两个系统：
  - 插件元数据（由 MANIFEST.MF 文件提供）；
  - 一个定义良好的插件生命周期和上下文；
  - 综合服务发现和注册；
  - ……
    注意： 在 Qt Plugin System 中，插件的元数据由 JSON 文件提供。

CTK Plugin Framework 的核心架构主要包含两个组件：Plugin System 本身和 Service Registry。然而，这两个组件是相互关联的，它们在 API 级别上的组合使得系统更加全面、灵活。

- Plugin System
  CTK Core 依赖于 QtCore 模块，因此 CTK Plugin Framework 基于 Qt Plugin System。Qt API 允许在运行时加载和卸载插件，这个功能在 CTK Plugin Framework 中得到了加强，以支持透明化延迟加载和解决依赖关系。

插件的元数据被编译进插件内部，可以通过 API 进行提取。此外，插件系统还使用 SQLite 缓存了元数据，以避免应用程序加载时间问题。另外，Plugin System 支持通过中央注册中心使用服务。

- Service Registry
  Qt Service Framework 是 Qt Mobility 项目发布的一个 Qt 解决方案，这种服务框架允许“声明式服务”（Getting Started with OSGi: Introducing Declarative Services ）和按需加载服务实现。为了启用动态（非持久性）服务，Qt Mobility 服务框架可以与 Service Registry 一起使用，类似于 OSGi Core Specifications 中描述的一样。

CTK Plugin Framework 优点

由于 CTK Plugin Framework 基于 OSGi，因此它继承了一种非常成熟且完全设计的组件系统，这在 Java 中用于构建高度复杂的应用程序，它将这些好处带给了本地（基于 Qt 的）C++ 应用程序。以下内容摘自 Benefits of Using OSGi (https://www.osgi.org/developer/benefits-of-using-osgi/)，并适应于 CTK Plugin Framework：

- 降低复杂性

  使用 CTK Plugin Framework 开发意味着开发插件，它们隐藏了内部实现，并通过定义良好的服务来和其它插件通信。隐藏内部机制意味着以后可以自由地更改实现，这不仅有助于 Bug 数量的减少，还使得插件的开发变得更加简单，因为只需要实现已经定义好的一定数量的功能接口即可。

- 复用

  标准化的组件模型，使得在应用程序中使用第三方组件变得非常容易。

- 现实情况

  CTK Plugin Framework 是一个动态框架，它可以动态地更新插件和服务。在现实世界中，有很多场景都和动态服务模型相匹配。因此，应用程序可以在其所属的领域中重用 Service Registry 的强大基元（注册、获取、用富有表现力的过滤语言列表、等待服务的出现和消失）。这不仅节省了编写代码，还提供了全局可见性、调试工具以及比为专用解决方案实现的更多的功能。在这样的动态环境下编写代码听起来似乎是个噩梦，但幸运的是，有支持类和框架可以消除大部分（如果不是全部的话）痛苦。

- 开发简单

  CTK Plugin Framework 不仅仅是组件的标准，它还指定了如何安装和管理组件。这个 API 可以被插件用来提供一个管理代理，这个管理代理可以非常简单，如命令 shell、图形桌面应用程序、Amazon EC2 的云计算接口、或 IBM Tivoli 管理系统。标准化的管理 API 使得在现有和未来的系统中集成 CTK Plugin Framework 变得非常容易。

- 动态更新

  OSGi 组件模型是一个动态模型，插件可以在不关闭整个系统的情况下被安装、启动、停止、更新和卸载。

- 自适应

OSGi 组件模型是从头设计的，以允许组件的混合和匹配。这就要求必须指定组件的依赖关系，并且需要组件在其可选依赖性并不总是可用的环境中生存。Service Registry 是一个动态注册表，其中插件可以注册、获取和监听服务。这种动态服务模型允许插件找出系统中可用的功能，并调整它们所能提供的功能。这使得代码更加灵活，并且能够更好地适应变化。

- 透明性

  插件和服务是 CTK 插件环境中的一等公民。管理 API 提供了对插件的内部状态的访问，以及插件之间的连接方式。可以停止部分应用程序来调试某个问题，或者可以引入诊断插件。

- 版本控制

  在 CTK Plugin Framework 中，所有的插件都经过严格的版本控制，只有能够协作的插件才会被连接在一起。

- 简单

  CTK 插件相关的 API 非常简单，核心 API 不到 25 个类。这个核心 API 足以编写插件、安装、启动、停止、更新和卸载它们，并且还包含了所有的监听类。

- 懒加载

  懒加载是软件中一个很好的点，OSGi 技术有很多的机制来保证只有当类真正需要的时候才开始加载它们。例如，插件可以用饿汉式启动，但是也可以被配置为仅当其它插件使用它们时才启动。服务可以被注册，但只有在使用时才创建。这些懒加载场景，可以节省大量的运行时成本。

- 非独占性

  CTK Plugin Framework 不会接管整个应用程序，你可以选择性地将所提供的功能暴露给应用程序的某些部分，或者甚至可以在同一个进程中运行该框架的多个实例。

- 非侵入

  在一个 CTK 插件环境中，不同插件均有自己的环境。它们可以使用任何设施，框架对此并无限制。CTK 服务没有特殊的接口需求，每个 QObject 都可以作为一个服务，每个类（也包括非 QObject）都可以作为一个接口。

CTK编译

使用cmake编译出与系统版本相应的动态库。参见CTK编译教程(64位环境 Windows + Qt + MinGW或MSVC + CMake) (../02/100643.html)。

使用 CTKWidgets

新项目-Application(Qt)-Qt Console Application，项目名称为UseCTKWidgets，pro文件的代码：

```
QT += core gui widgets

TARGET = UseCTKWidgets
TEMPLATE = app

# CTK 安装路径
CTK_INSTALL_PATH = $$PWD/../../CTKInstall

# CTK 相关库所在路径（例如：CTKCore.lib、CTKWidgets.lib）
CTK_LIB_PATH = $$CTK_INSTALL_PATH/lib/ctk-0.1

# CTK 相关头文件所在路径（例如：ctkPluginFramework.h）
CTK_INCLUDE_PATH = $$CTK_INSTALL_PATH/include/ctk-0.1

# 相关库文件（CTKCore.lib、CTKWidgets.lib）
LIBS += -L$$CTK_LIB_PATH -lCTKCore -lCTKWidgets

INCLUDEPATH += $$CTK_INCLUDE_PATH
```

主函数main加载部件，代码如下：*main.cpp*

```cpp
#include <QApplication>
#include <QFormLayout>
#include <QVBoxLayout>

#include <ctkCheckablePushButton.h>
#include <ctkCollapsibleButton.h>
#include <ctkColorPickerButton.h>
#include <ctkRangeWidget.h>

int main(int argc, char* argv[])
{
  QApplication app(argc, argv);

  // 可折叠按钮
  ctkCollapsibleButton* buttons = new ctkCollapsibleButton("Buttons");

  // 可勾选按钮
  ctkCheckablePushButton* checkablePushButton = new ctkCheckablePushButton();
  checkablePushButton->setText("Checkable");

  // 颜色拾取器
  ctkColorPickerButton* colorPickerButton = new ctkColorPickerButton();
  colorPickerButton->setColor(QColor("#9e1414"));

  ctkCollapsibleButton* sliders = new ctkCollapsibleButton("Sliders");

  QFormLayout* buttonsLayout = new QFormLayout;
  buttonsLayout->setFieldGrowthPolicy(QFormLayout::AllNonFixedFieldsGrow);
  buttonsLayout->addRow("ctkCheckablePushButton", checkablePushButton);
  buttonsLayout->addRow("ctkColorPickerButton", colorPickerButton);
  buttons->setLayout(buttonsLayout);

  QVBoxLayout* topLevelLayout = new QVBoxLayout();
  topLevelLayout->addWidget(buttons);
  topLevelLayout->addWidget(sliders);

  QFormLayout* slidersLayout = new QFormLayout;
  ctkRangeWidget* rangeWidget = new ctkRangeWidget();
  slidersLayout->addRow("ctkRangeWidget", rangeWidget);
  sliders->setLayout(slidersLayout);

  QWidget topLevel;
  topLevel.setLayout(topLevelLayout);
  topLevel.show();

  return app.exec();
}
```

项目代码：UseCTKWidgets (https://github.com/myhhub/CTK-project/tree/main/UseCTKWidgets)

初步使用 CTK Plugin Framework

项目结构

由于每一个插件都要建一个子项目，本项目刚开始创建时在QtCreator中选择*新建-其他项目-子目录项目*，新建项目名称为SampleCTK (https://github.com/myhhub/CTK-project/tree/main/SampleCTK)，然后建立主程序入口项目，这里建立一个*控制台项目*，取名叫App。

更改项目输出路径：*app.pro*

```
DESTDIR = $$OUT_PWD/../bin
```

主函数中加载插件，启动框架：*main.cpp*

```
#include <QCoreApplication>
#include "ctkPluginFrameworkFactory.h"
#include "ctkPluginFramework.h"
#include "ctkPluginException.h"
#include <QDebug>
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    app.setApplicationName("SampleCTK");//给框架创建名称，Linux下没有会报错
    ctkPluginFrameworkFactory frameWorkFactory;
    QSharedPointer<ctkPluginFramework> framework = frameWorkFactory.getFramework();
    try {
        // 初始化并启动插件框架
        framework->init();
        framework->start();
        qDebug() << "CTK Plugin Framework start ...";
    } catch (const ctkPluginException &e) {
        qDebug() << "Failed to initialize the plugin framework: " << e.what();
        qDebug() << e.message() << e.getType();
    }
    return app.exec();
}
```

如果想把CTK初始化、插件安装启动、获取等操作封装成一个类，那么要注意：需要把CTK相关的变量定义成类属性，不能是局部变量，否则会出现各种问题如获取不了服务、服务引用为空等。

没有报错的话及表示插件加载成功！

其中QSharedPointer framework这个对象比较有意思，既可以作为对象也可以作为对象指针，但要作为插件框架使用必须要用指针方法调用，所以代码里使用“->”。

项目加载CTK框架插件

项目SampleCTK新建文本文件*CTK*，然后更改扩展名为pri。文件加载CTK安装目录及源代码目录，编译出的动态库就可以当普通动态库使用加载了，*CTK.pri*里面加载代码为：

```
# CTK 安装路径
CTK_INSTALL_PATH = $$PWD/../../CTKInstall

# CTK 插件相关库所在路径（例如：CTKCore.lib、CTKPluginFramework.lib）
CTK_LIB_PATH = $$CTK_INSTALL_PATH/lib/ctk-0.1

# CTK 插件相关头文件所在路径（例如：ctkPluginFramework.h）
CTK_INCLUDE_PATH = $$CTK_INSTALL_PATH/include/ctk-0.1

# CTK 插件相关头文件所在路径（主要因为用到了 service 相关东西）
CTK_INCLUDE_FRAMEWORK_PATH = $$PWD/../../CTK-master/Libs/PluginFramework

LIBS += -L$$CTK_LIB_PATH -lCTKCore -lCTKPluginFramework

INCLUDEPATH += $$CTK_INCLUDE_PATH \
               $$CTK_INCLUDE_FRAMEWORK_PATH
```

将CTK.pri文件的内容引入 pro 文件：*app.pro*

```
include($$PWD/../CTK.pri)
```

CTK插件的接口处理

CTK框架由一个一个可分离的插件组成，框架对插件识别有一定要求，目前网上很多一整块扔出来对新人不太友好，博主这里讲解是尽量拆。单个插件最基本的格式要求分成Activator，qrc文件，以及MANIFEST.MF，以say Hello模块HelloCTK为例。

Activator注册器

每个插件都有自己的注册器Activator。

右键项目选择*新建子项目-其他项目-Empty qmake Project*，项目名称为HelloCTK，pro文件中添加代码：

```
QT += core
QT -= gui

TEMPLATE = lib
CONFIG += plugin
TARGET = HelloCTK
DESTDIR = $$OUT_PWD/../bin/plugins

include($$PWD/../CTK.pri)
```

生成的插件名(TARGET)不要有下划线，因为CTK会默认将插件名中的下划线替换成点号，最后后就导致找不到插件。

项目中添加C++类HelloActivator，代码如下：

hello_activator.h

```
#ifndef HELLO_ACTIVATOR_H
#define HELLO_ACTIVATOR_H

#include <QObject>
#include "ctkPluginActivator.h"

class HelloActivator : public QObject, public ctkPluginActivator
{
    Q_OBJECT
    Q_INTERFACES(ctkPluginActivator)
    Q_PLUGIN_METADATA(IID "HELLO_CTK")
    //向Qt的插件框架声明，希望将xxx插件放入到框架中。

public:
    void start(ctkPluginContext* context);
    void stop(ctkPluginContext* context);

};

#endif // HELLO_ACTIVATOR_H
```

hello_activator.cpp

```
#include "hello_activator.h"
#include <QDebug>

void HelloActivator::start(ctkPluginContext* context)
{
    qDebug() << "HelloCTK start";
}

void HelloActivator::stop(ctkPluginContext* context)
{
    qDebug() << "HelloCTK stop";
    Q_UNUSED(context)
    //Q_UNUSED,如果一个函数的有些参数没有用到、某些变量只声明不使用，但是又不想编译器、
}
```

activator是标准的Qt插件类，它实现ctkPluginActivator的start、stop函数并对外提供接口。我这里是Qt5的版本，所以使用Q_PLUGIN_METADATA申明插件，Qt4需要用自己的方法实现插件。

qrc文件

创建插件的资源文件，格式如下：

```
<RCC>
 <qresource prefix="/HelloCTK/META-INF">
   <file>MANIFEST.MF</file>
    </qresource>
</RCC>
```

插件加载后会寻找*同名前缀/META-INF*，所以前缀格式固定，将MANIFEST.MF文件添加进来

MENIFEST.MF文件内容如下：

可直接在MF文件里添加自己特有的元数据

```
Plugin-SymbolicName:HelloCTK
Plugin-Version:1.0.0
Plugin-Number:100 #元数据
```

注意：Plugin-SymbolicName要满足这里的前缀是：TARGET/META-INF格式。TARGET的名字最好和工程名一致，不然可能出现device not open错误。

文件包含ctk插件的基本信息，只要ctk框架正常识别到文件中Plugin-SymbolicName等信息，则判定它是一个ctk插件，能够正常调用activator中的start、stop函数。这个文件需要拷到插件生成路径下，pro文件中添加代码：

```
file.path = $$DESTDIR
file.files = MANIFEST.MF
INSTALLS += file
```

CTK插件启用

根据以上步骤，一个CTK插件接口定义基本完成，我们在App项目下调用观察插件是否能够正常加载。main函数中框架启动成功后添加以下代码：

```
 QString dir = QCoreApplication::applicationDirPath();
      dir += "/plugins/HelloCTK.dll";
      qDebug() << dir;
      QUrl url = QUrl::fromLocalFile(dir);
      QSharedPointer<ctkPlugin> plugin;
      try
      {
          plugin = framework->getPluginContext()->installPlugin(url);
      }catch(ctkPluginException e){
          qDebug() << e.message() << e.getType();
      }
      try{
          plugin->start(ctkPlugin::START_TRANSIENT);
      }catch(ctkPluginException e){
          qDebug() << e.message() << e.getType();
      }
```

控制台打印输出：

```
"C:/d/mmm/qt/ctk/CTK-examples/build-SampleCTK-Desktop_Qt_5_15_1_MSVC2019_64bit-Releas
 HelloCTK start
```

成功调用HelloCTK中start内打印输出，则表明ctk插件接口正常定义并能成功加载。其中*start*(ctkPlugin::START_TRANSIENT)表示立即启用插件，不设置参数的话加载后也不会立即打印输出。

基本使用 CTK Plugin Framework

CTK插件间通信

CTK框架插件化开发实现功能的隔离，插件通信需要参照固定标准，这里介绍两种插件间通信的方法。

通信方法一. 注册接口调用

注册接口调用

函数接口

接口就是纯虚函数类，也就是最终的服务的前身。

上面我们已经编译出需要的动态库，首先确定我们需要插件向外部暴露的功能有什么，比如这里我们需要说”Hello,CTK!”的操作，定义头文件如下：*hello_service.h*

```
#ifndef HELLO_SERVICE_H
#define HELLO_SERVICE_H

#include <QtPlugin>

class HelloService
{
public:
    virtual ~HelloService() {}
    virtual void sayHello() = 0;
};

#define HelloService_iid "org.commontk.service.demos.HelloService"
Q_DECLARE_INTERFACE(HelloService, HelloService_iid)
//此宏将当前这个接口类声明为接口，后面的一长串就是这个接口的唯一标识。
#endif // HELLO_SERVICE_H
```

Q_DECLARE_INTERFACE将接口类向Qt系统申明，然后添加它的实现对象：

接口的实现

插件就是实现这个接口类的实现类，所以理论上有多少个实现类就有多少个插件。

hello_impl.h

```
#ifndef HELLO_IMPL_H
#define HELLO_IMPL_H

#include "hello_service.h"
#include <QObject>

class ctkPluginContext;

class HelloImpl : public QObject, public HelloService
{
    Q_OBJECT
    Q_INTERFACES(HelloService)
    /*
    此宏与Q_DECLARE_INTERFACE宏配合使用。
    Q_DECLARE_INTERFACE：声明一个接口类
    Q_INTERFACES：当一个类继承这个接口类，表明需要实现这个接口类
    */

public:
    HelloImpl(ctkPluginContext* context);
    void sayHello() Q_DECL_OVERRIDE;
};

#endif // HELLO_IMPL_H
```

hello_impl.cpp

```
#include "hello_impl.h"
#include <ctkPluginContext.h>
#include <QtDebug>

HelloImpl::HelloImpl(ctkPluginContext* context)
{
    context->registerService<HelloService>(this);
}

void HelloImpl::sayHello()
{
    qDebug() << "Hello,CTK!";
}
```

这仍是Qt的插件定义格式，但是不会作为插件导出，外部功能接口可以自定义。

服务注册(Activator注册服务)

激活类里有一个独占智能指针，指向接口类【使用多态，指针都指向父类】，然后在start里new一个实现类，注册这个实现类为服务，功能是实现接口类的接口，然后将智能指针指向这个实现类。可以理解为以后向框架索取这个服务的时候，实际获取的就是这个new出来的实现类。如果不用智能指针，就需要在stop里手动delete这个实现类。

每个插件都有自己的注册器Activator，功能节接口完成后，在插件启动时注册到ctk框架的服务中，代码如下：*hello_activator.cpp*

```
#include "hello_activator.h"
#include "hello_impl.h"

void HelloActivator::start(ctkPluginContext* context)
{
    s.reset(new HelloImpl(context));
    //调用注册服务context->registerService<HelloService>(this);
}
```

接口调用

CTK插件启用后，就可以调用接口。

主函数框架及插件加载完成后，即可调用插件接口，代码如下：*main.cpp*

```
 #include "../HelloCTK/hello_service.h"

    // 获取服务引用
    ctkServiceReference reference = context->getServiceReference<HelloService>();
    if (reference) {
        // 获取指定 ctkServiceReference 引用的服务对象
        HelloService* service = qobject_cast<HelloService *>(context->getService(ref
        if (service != Q_NULLPTR) {
            // 调用服务
            service->sayHello();
        }
    }
```

在获取服务的时候，有两个重载方式【可直接使用的】

```
1、HelloService* service = context->getService<HelloService>(reference);
2、HelloService* service = qobject_cast<HelloService*>(context->getService(reference)
```

服务就是根据接口的实例，每生成一个服务就会调用一次注册器的start。把接口当做类，服务是根据类new出的对象，插件就是动态库dll。

项目代码：SampleCTK (https://github.com/myhhub/CTK-project/tree/main/SampleCTK)

优化解耦(实现类和激活类分离)

编写插件主要有3个步骤：接口类、实现类、激活类。不在实现类的构造函数里注册服务，降低耦合性，接口类就只做接口声明，实现类就只实现接口，激活类就负责将服务整合到ctk框架中。

接口类没有什么变化，实现类少了注册的代码，构造函数也无参数，注册的过程放在了激活类里。

1. 实现类

.h

```
1   HelloImpl( );
```

.cpp

```
1   HelloImpl::HelloImpl( )
2   {
3        qDebug()<<"this is imp";
4   }
```

2. 激活类

.cpp

```
1   void HelloActivator::start(ctkPluginContext* context)
2   {
3       HelloImpl* helloImpl = new HelloImpl(context);
4       context->registerService<HelloService>(helloImpl);
5       s.reset(helloImpl);
6   }
```

接口、插件、服务的关系

1、1对1

1个接口类由1个类实现，输出1个服务和1个插件。

上面项目为典型1对1关系。

2、多对1

1个类实现了多个接口类，输出多个服务和1个插件，无论想使用哪个服务最终都通过这同一个插件来实现。

实现类，实现多个接口。

```
#include "greet_impl.h"
#include <QtDebug>

GreetImpl::GreetImpl()
{

}

void GreetImpl::sayHello()
{
    qDebug() << "Hello,CTK!";
}

void GreetImpl::sayBye()
{
    qDebug() << "Bye,CTK!";
}
```

获取不同服务

```
// 获取服务引用
ctkServiceReference ref = context->getServiceReference<HelloService>();
if (ref) {
    HelloService* service = qobject_cast<HelloService *>(context->getService(ref));
    if (service != Q_NULLPTR)
        service->sayHello();
}

ref = context->getServiceReference<ByeService>();
if (ref) {
    ByeService* service = qobject_cast<ByeService *>(context->getService(ref));
    if (service != Q_NULLPTR)
        service->sayBye();
}
```

具体实现参见项目：PluginAndService/MultipleInterfaces (https://github.com/myhhub/CTK-project/tree/main/PluginAndService/MultipleInterfaces)

3、1对多

1接口由多个个类实现，也就是某一个问题提供了多种解决思路，输出1个服务和多个插件，通过ctkPluginConstants::SERVICE_RANKING和ctkPluginConstants::SERVICE_ID来调用不同的插件。这里虽然有两个插件，但都是被编译到同一个dll中的。服务的获取策略如下：容器会返回排行最低的服务，返回注册时SERVICE_RANKING属性值最小的服务。如果有多个服务的排行值相等，那么容器将返回PID值最小的那个服务。

某插件每次调用另一个插件的时候，只会生成一个实例，然后把实例存到内存当中，不会因为多次调用而生成多个服务实例。

在使用1接口2插件的时候，虽然是两个插件，也会有两个激活类【从原理上来讲1个激活类就行了，但是在start里注册两次】，其中的IID只能有一个。从Qt插件基础上来说，一个dll只能有一个IID。

多个实现类，实现1个接口。

welcome_ctk_impl.cpp

```
#include "welcome_ctk_impl.h"
#include <QtDebug>

WelcomeCTKImpl::WelcomeCTKImpl()
{

}

void WelcomeCTKImpl::welcome()
{
    qDebug() << "Welcome CTK!";
}
```

welcome_qt_impl.cpp

```
#include "welcome_qt_impl.h"
#include <QtDebug>

WelcomeQtImpl::WelcomeQtImpl()
{

}

void WelcomeQtImpl::welcome()
{
    qDebug() << "Welcome Qt!";
}
```

对应的多个激活类

welcome_ctk_activator.cpp

```
#include "welcome_ctk_impl.h"
#include "welcome_ctk_activator.h"
#include <QtDebug>

void WelcomeCTKActivator::start(ctkPluginContext* context)
{
    ctkDictionary properties;
    properties.insert(ctkPluginConstants::SERVICE_RANKING, 2);
    properties.insert("name", "CTK");

    m_pImpl = new WelcomeCTKImpl();
    context->registerService<WelcomeService>(m_pImpl, properties);
}

void WelcomeCTKActivator::stop(ctkPluginContext* context)
{
    Q_UNUSED(context)

    delete m_pImpl;
}
```

welcome_qt_activator.cpp

```
#include "welcome_qt_impl.h"
#include "welcome_qt_activator.h"
#include <QtDebug>

void WelcomeQtActivator::start(ctkPluginContext* context)
{
    ctkDictionary properties;
    properties.insert(ctkPluginConstants::SERVICE_RANKING, 1);
    properties.insert("name", "Qt");

    m_pImpl = new WelcomeQtImpl();
    context->registerService<WelcomeService>(m_pImpl, properties);
}

void WelcomeQtActivator::stop(ctkPluginContext* context)
{
    Q_UNUSED(context)

    delete m_pImpl;
}
```

获取服务

```
// 1. 获取所有服务
QList<ctkServiceReference> refs = context->getServiceReferences<WelcomeService>();
foreach (ctkServiceReference ref, refs) {
    if (ref) {
        qDebug() << "Name:" << ref.getProperty("name").toString()
                 <<  "Service ranking:" << ref.getProperty(ctkPluginConstants::SERVI
                  << "Service id:" << ref.getProperty(ctkPluginConstants::SERVICE_ID
        WelcomeService* service = qobject_cast<WelcomeService *>(context->getService
        if (service != Q_NULLPTR)
            service->welcome();
    }
}

// 2. 使用过滤表达式，获取感兴趣的服务
refs = context->getServiceReferences<WelcomeService>("(&(name=CTK))");
foreach (ctkServiceReference ref, refs) {
    if (ref) {
        WelcomeService* service = qobject_cast<WelcomeService *>(context->getService
        if (service != Q_NULLPTR)
            service->welcome();
    }
}

// 3. 获取某一个服务（由 Service Ranking 和 Service ID 决定）
ctkServiceReference ref = context->getServiceReference<WelcomeService>();
if (ref) {
    WelcomeService* service = qobject_cast<WelcomeService *>(context->getService(ref
    if (service != Q_NULLPTR)
        service->welcome();
}
```

具体实现参见项目：PluginAndService/OneInterface (https://github.com/myhhub/CTK-project/tree/main/PluginAndService/OneInterface)

通信方法二. 事件监听

CTK框架中的事件监听，即观察者模式流程上是这样：接收者注册监听事件->发送者发送事件->接收者接收到事件并响应；相比调用插件接口，监听事件插件间依赖关系更弱，不用指定事件的接收方和发送方是谁。

要使用CTK框架的事件服务，准备工作应该从cmake开始，编译出支持事件监听的动态库，名称为liborg_commontk_eventadmin.dll。现在要完成的内容是，从上面生成的主窗体中，以事件监听的方式调用一个子窗体。

1、通信主要用到了ctkEventAdmin结构体，主要定义了如下接口：

postEvent：类通信形式异步发送事件

sendEvent：类通信形式同步发送事件

publishSignal：信号与槽通信形式发送事件

unpublishSignal：取消发送事件

subscribeSlot：信号与槽通信形式订阅时间，返回订阅的ID

unsubscribeSlot：取消订阅事件

updateProperties：更新某个订阅ID的主题

2、通信的数据是：ctkDictionary

其实就是个hash表：typedef QHash<QString,QVariant> ctkDictionary

事件监听

具体项目：EventAdmin/SendEvent

加载EventAdmin动态库

添加动态库可以使用ctkPluginFrameworkLauncher，代码如下：*main.cpp*

```
1       // 获取插件所在位置
2       // 在插件的搜索路径列表中添加一条路径
3       ctkPluginFrameworkLauncher::addSearchPath("../../../../CTKInstall/lib/ctk-0.1/plu
4       // 设置并启动 CTK 插件框架
5       ctkPluginFrameworkLauncher::start("org.commontk.eventadmin");
6     ……
7   // 停止插件
8   ctkPluginFrameworkLauncher::stop();
```

事件注册监听(接收插件)

首先编写我们需要的接收者模块，并注册监听事件，这里我们新建一个模块BlogEventHandler，模块的接口处理参见上面“CTK插件的接口处理”。插件部分代码如下：

blog_event_handler.h

```
#ifndef BLOG_EVENT_HANDLER_H
#define BLOG_EVENT_HANDLER_H

#include <QObject>
#include <service/event/ctkEventHandler.h>

// 事件处理程序（或订阅者）
class BlogEventHandler : public QObject, public ctkEventHandler
{
    Q_OBJECT
    Q_INTERFACES(ctkEventHandler)

public:
    // 处理事件
    void handleEvent(const ctkEvent& event) Q_DECL_OVERRIDE
    {
        QString title = event.getProperty("title").toString();
        QString content = event.getProperty("content").toString();
        QString author = event.getProperty("author").toString();

        qDebug() << "EventHandler received the message, topic:" << event.getTopic()
                 << "properties:" << "title:" << title << "content:" << content << "
    }
};

#endif // BLOG_EVENT_HANDLER_H
```

与上面自定义接口不同，这里我们实例化ctkEventHandler对象，并实现*handleEvent*接口。构造函数中注册的服务对象是ctkEventHandler，在注册时指定触发的事件，当事件触发时调用该对象的*handleEvent*实现指定操作。

事件发送(发送插件)

监听对象完成后调用比较简单，代码如下：*blog_manager.cpp*

```cpp
#include "blog_manager.h"
#include <service/event/ctkEventAdmin.h>
#include <QtDebug>

BlogManager::BlogManager(ctkPluginContext* context)
    : m_pContext(context)
{

}

// 发布事件
void BlogManager::publishBlog(const Blog& blog)
{
    ctkServiceReference ref = m_pContext->getServiceReference<ctkEventAdmin>();
    if (ref) {
        ctkEventAdmin* eventAdmin = m_pContext->getService<ctkEventAdmin>(ref);

        ctkDictionary props;
        props["title"] = blog.title;
        props["content"] = blog.content;
        props["author"] = blog.author;
        ctkEvent event("org/commontk/bloggenerator/published", props);

        qDebug() << "Publisher sends a message, properties:" << props;

        eventAdmin->sendEvent(event);
    }
}
```

项目代码：EventAdmin/SendEvent (https://github.com/myhhub/CTK-project/tree/main/EventAdmin/SendEvent)

事件发送方式(类通信、信号槽通信)

1、类通信

原理就是直接将信息使用CTK的eventAdmin接口send/post出去。

上面项目为典型类通信。

2、信号槽通信

原理是将Qt自己的信号与CTK的发送事件绑定、槽与事件订阅绑定。

接收槽

```cpp

void BlogEventHandlerUsingSlotsActivator::start(ctkPluginContext* context)
{
    m_pEventHandler = new BlogEventHandlerUsingSlots();

    ctkDictionary props;
```

```
7       props[ctkEventConstants::EVENT_TOPIC] = "org/commontk/bloggenerator/published";
8       ctkServiceReference ref = context->getServiceReference<ctkEventAdmin>();
9       if (ref) {
10          ctkEventAdmin* eventAdmin = context->getService<ctkEventAdmin>(ref);
11          eventAdmin->subscribeSlot(m_pEventHandler, SLOT(onBlogPublished(ctkEvent)), p
12      }
13  }
```

发送信号

```
1
2   BlogManagerUsingSignals::BlogManagerUsingSignals(ctkPluginContext *context)
3   {
4       ctkServiceReference ref = context->getServiceReference<ctkEventAdmin>();
5       if (ref) {
6           ctkEventAdmin* eventAdmin = context->getService<ctkEventAdmin>(ref);
7           // 使用 Qt::DirectConnection 等同于 ctkEventAdmin::sendEvent()
8           eventAdmin->publishSignal(this, SIGNAL(blogPublished(ctkDictionary)), "org/c
9       }
10  }
```

具体项目：项目代码：EventAdmin/SignalSlot (https://github.com/myhhub/CTK-project/tree/main/EventAdmin/SignalSlot)

二者的区别

1、通过event事件通信，是直接调用CTK的接口，把数据发送到CTK框架；通过信号槽方式，会先在Qt的信号槽机制中转一次，再发送到CTK框架。故效率上来讲，event方式性能高于信号槽方式。

2、两种方式发送数据到CTK框架，这个数据包含：主题+属性。主题就是topic，属性就是ctkDictionary。 一定要注意signal方式的信号定义，参数不能是自定义的，一定要是ctkDictionary，不然会报信号槽参数异常错误。

3、两种方式可以混用，如发送event事件，再通过槽去接收；发送signal事件，再通过event是接收。

4、同步：sendEvent、Qt::DirectConnection；异步：postEvent、Qt::QueuedConnection

这里的同步是指：发送事件之后，订阅了这个主题的数据便会处理数据【handleEvent、slot】，处理的过程是在发送者的线程完成的。可以理解为在发送了某个事件之后，会立即执行所有订阅此事件的回调函数。

异步：发送事件之后，发送者便会返回不管，订阅了此事件的所有插件会根据自己的消息循环，轮到了处理事件后才会去处理。不过如果长时间没处理，CTK也有自己的超时机制。如果事件处理程序花费的时间比配置的超时时间长，那么就会被列入黑名单。一旦处理程序被列入黑名单，它就不会再被发送任何事件。

插件依赖

插件加载时一般根据首字母大小自动加载，所以在插件启用时，某个插件还没有被调用，所以发送事件没有接收方，这样就要考虑到插件依赖关系，在MANIFEST.MF中添加依赖：

```
1   Plugin-SymbolicName:Plugin-xxx-1
2   Plugin-Version:1.0.0
3   Require-Plugin:Plugin-xxx-2; plugin-version="[1.0,2.0)"; resolution:="mandatory"
```

Plugin-xxx-2：为需要依赖的插件名【就是另一个插件在MANIFEST.MF里的Plugin-SymbolicName】；

[1.0,2.0)：为Plugin-xxx-2的版本，这里是左闭右开区间，默认是1.0,；

resolution：有两个选择，optional、mandatory。前者是弱依赖，就算依赖的插件没有，当前插件也能正常使用，后者是强依赖，如果没有依赖的插件，就当前插件就不能被start。

这样就向框架申明了，该插件加载时需要先加载Plugin-xxx-2插件，所有用户插件都应该有这样一份申明。

具体实现参见项目：RequirePlugin (https://github.com/myhhub/CTK-project/tree/main/RequirePlugin)

插件元数据

获取MANIFEST.MF中的数据

```
1   QHash<QString, QString> headers = plugin->getHeaders();
2   ctkVersion version = ctkVersion::parseVersion(headers.value(ctkPluginConstants::PLUGI
3   QString name = headers.value(ctkPluginConstants::PLUGIN_NAME);
```

具体实现参见项目：GetMetaData (https://github.com/myhhub/CTK-project/tree/main/GetMetaData)

高级使用 CTK Plugin Framework

CTK 服务工厂

注册服务的时候能够用服务工厂来注册，访问服务getService中的plugin参数是执行ctkPluginContext::getService(const ctkServiceReference&)的插件,从而这里工厂根据执行的不同插件名称返回了不同的服务实现。

服务工厂的作用：

1. 在服务中可以知道是哪个其他插件在使用它；
2. 懒汉式使用服务，需要的时候才new；
3. 厂其他插件使用有服务工厂和使用无服务工的服务，没有任何区别，代码都一样；
4. 可根据需要创建多种实现的服务，就是：多种服务对应一个插件。

接口类

```
#ifndef HELLO_SERVICE_H
#define HELLO_SERVICE_H

#include <QtPlugin>

class HelloService
{
public:
    virtual ~HelloService() {}
    virtual void sayHello() = 0;
};

#define HelloService_iid "org.commontk.service.demos.HelloService"
Q_DECLARE_INTERFACE(HelloService, HelloService_iid)

#endif // HELLO_SERVICE_H
```

多实现类

```
#ifndef HELLO_IMPL_H
#define HELLO_IMPL_H

#include "hello_service.h"
#include <QObject>
#include <QtDebug>

// HelloWorld
class HelloWorldImpl : public QObject, public HelloService
{
    Q_OBJECT
    Q_INTERFACES(HelloService)

public:
    void sayHello() Q_DECL_OVERRIDE {
        qDebug() << "Hello,World!";
    }
};

// HelloCTK
class HelloCTKImpl : public QObject, public HelloService
{
    Q_OBJECT
    Q_INTERFACES(HelloService)

public:
    void sayHello() Q_DECL_OVERRIDE {
        qDebug() << "Hello,CTK!";
    }
};

#endif // HELLO_IMPL_H
```

服务工厂类

```
#ifndef SERVICE_FACTORY_H
#define SERVICE_FACTORY_H

#include <ctkServiceFactory.h>
#include <ctkPluginConstants.h>
#include <ctkVersion.h>
#include "hello_impl.h"

class ServiceFactory : public QObject, public ctkServiceFactory
{
    Q_OBJECT
    Q_INTERFACES(ctkServiceFactory)

public:
    ServiceFactory() : m_counter(0) {}

    // 创建服务对象
```

```
18      QObject* getService(QSharedPointer<ctkPlugin> plugin, ctkServiceRegistration reg
19          Q_UNUSED(registration)
20
21          qDebug() << "Create object of HelloService for: " << plugin->getSymbolicName
22          m_counter++;
23          qDebug() << "Number of plugins using service: " << m_counter;
24
25          QHash<QString, QString> headers = plugin->getHeaders();
26          ctkVersion version = ctkVersion::parseVersion(headers.value(ctkPluginConstant
27          QString name = headers.value(ctkPluginConstants::PLUGIN_NAME);
28
29          QObject* hello = getHello(version);
30          return hello;
31      }
32
33      // 释放服务对象
34      void ungetService(QSharedPointer<ctkPlugin> plugin, ctkServiceRegistration regis
35          Q_UNUSED(plugin)
36          Q_UNUSED(registration)
37          Q_UNUSED(service)
38
39          qDebug() << "Release object of HelloService for: "  << plugin->getSymbolicNa
40          m_counter--;
41          qDebug() << "Number of plugins using service: "  << m_counter;
42      }
43
44  private:
45      // 根据不同的版本，获取不同的服务
46      QObject* getHello(ctkVersion version) {
47          if (version.toString().contains("alpha")) {
48              return new HelloWorldImpl();
49          } else {
50              return new HelloCTKImpl();
51          }
52      }
53
54  private:
55      int m_counter;  // 计数器
56  };
57
58  #endif // SERVICE_FACTORY_H
```

可以根据插件，获取不同的服务。若主框架【main.cpp】的symbolicName是system.plugin

激活类

```
#ifndef HELLO_ACTIVATOR_H
#define HELLO_ACTIVATOR_H

#include <ctkPluginActivator.h>
#include <ctkPluginContext.h>
#include "hello_service.h"
#include "service_factory.h"

class HelloActivator : public QObject, public ctkPluginActivator
{
    Q_OBJECT
    Q_INTERFACES(ctkPluginActivator)
    Q_PLUGIN_METADATA(IID "HELLO")

public:
    // 注册服务工厂
    void start(ctkPluginContext* context) {
        ServiceFactory *factory = new ServiceFactory();
        context->registerService<HelloService>(factory);
    }

    void stop(ctkPluginContext* context) {
        Q_UNUSED(context)
    }
};

#endif // HELLO_ACTIVATOR_H
```

插件中访问服务

```
// 访问服务
ctkServiceReference reference = context->getServiceReference<HelloService>();
if (reference) {
    HelloService* service = qobject_cast<HelloService *>(context->getService(referenc
    if (service != Q_NULLPTR) {
        service->sayHello();
    }
}
```

具体实现参见项目：ServiceFactory (https://github.com/myhhub/CTK-project/tree/main/ServiceFactory)

CTK 事件监听

CTK一共有三种事件可以监听：框架事件、插件事件、服务事件。但是这些事件只有再变化时才能监听到，如果已经变化过后，进入一个稳定的状态，这时才去监听，那么是无法监听到的。

框架事件

针对整个框架的，相当于只有一个，因为框架只有一个，但是这里有个问题，就是监听这个事件是在框架初始化之后的，所以根本没法监听到框架事件的初始化，只能监听到结束的事件。类型有

```
1   FRAMEWORK_STARTED
2   PLUGIN_ERROR
3   PLUGIN_WARNING
4   PLUGIN_INFO
5   FRAMEWORK_STOPPED
6   FRAMEWORK_STOPPED_UPDATE
7   FRAMEWORK_WAIT_TIMEDOUT
```

服务事件

在创建、回收插件时的事情，主要体现在服务的注册和注销。类型有

```
1   REGISTERED
2   MODIFIED
3   MODIFIED_ENDMATCH
4   UNREGISTERING
```

插件事件

在安装、启动插件的过程中呈现的，主要就是插件的一个状态的变化。类型有

```
1   INSTALLED
2   RESOLVED
3   LAZY_ACTIVATION
4   STARTING
5   STARTED
6   STOPPING
7   STOPPED
8   UPDATED
9   UNRESOLVED
10  UNINSTALLED
```

监听例子

监听类，**event_listener.h**

```
#ifndef EVENT_LISTENER_H
#define EVENT_LISTENER_H

#include <QObject>
#include <ctkPluginFrameworkEvent.h>
#include <ctkPluginEvent.h>
#include <ctkServiceEvent.h>

class EventListener : public QObject
{
    Q_OBJECT

public:
    explicit EventListener(QObject *parent = Q_NULLPTR);
    ~EventListener();

public slots:
    // 监听框架事件
    void onFrameworkEvent(const ctkPluginFrameworkEvent& event);
    // 监听插件事件
    void onPluginEvent(const ctkPluginEvent& event);
    // 监听服务事件
    void onServiceEvent(const ctkServiceEvent& event);
};

#endif // EVENT_LISTENER_H
```

event_listener.cpp

```
#include "event_listener.h"

EventListener::EventListener(QObject *parent)
    : QObject(parent)
{
}

EventListener::~EventListener()
{
}

// 监听框架事件
void EventListener::onFrameworkEvent(const ctkPluginFrameworkEvent& event)
{
    if (!event.isNull()) {
```

```
            QSharedPointer<ctkPlugin> plugin = event.getPlugin();
            qDebug() << "FrameworkEvent: [" << plugin->getSymbolicName() << "]" << event
        } else {
            qDebug() << "The framework event is null";
        }
    }

    // 监听插件事件
    void EventListener::onPluginEvent(const ctkPluginEvent& event)
    {
        if (!event.isNull()) {
            QSharedPointer<ctkPlugin> plugin = event.getPlugin();
            qDebug() << "PluginEvent: [" << plugin->getSymbolicName() << "]" << event.ge
        } else {
            qDebug() << "The plugin event is null";
        }
    }

    // 监听服务事件
    void EventListener::onServiceEvent(const ctkServiceEvent &event)
    {
        if (!event.isNull()) {
            ctkServiceReference ref = event.getServiceReference();
            QSharedPointer<ctkPlugin> plugin = ref.getPlugin();
            qDebug() << "ServiceEvent: [" << event.getType() << "]" << plugin->getSymbol
        } else {
            qDebug() << "The service event is null";
        }
    }
```

启用监听,**main.cpp**:

```
    // 事件监听
    EventListener listener;
    context->connectFrameworkListener(&listener, SLOT(onFrameworkEvent(ctkPluginFrameworkI
    context->connectPluginListener(&listener, SLOT(onPluginEvent(ctkPluginEvent)));
    // 过滤 ctkEventAdmin 服务
    // QString filter = QString("(%1=%2)").arg(ctkPluginConstants::OBJECTCLASS).arg("org.
    context->connectServiceListener(&listener, "onServiceEvent"); //, filter);
```

具体实现参见项目：EventListener (https://github.com/myhhub/CTK-project/tree/main/EventListener)

CTK 服务追踪

服务追踪：如果想在B插件里使用A服务，可以专门写一个类继承ctkServiceTracker，在这个类里完成对A服务的底层操作，然后在B插件里通过这个类提供的接口来使用回收A服务。

理论上ctkServiceTracker和A服务应该是一起的，这里有点像服务工厂。优点就是获取服务的代码简单，不用各种判断空指针。

服务A

服务A实现类，log_impl.cpp

```
#include "log_impl.h"
#include <QtDebug>

LogImpl::LogImpl()
{

}

void LogImpl::debug(QString msg)
{
    qDebug() << "This is a debug message: " << msg;
}
```

服务A激活类，log_activator.cpp

```
#include "log_impl.h"
#include "log_activator.h"
#include <ctkPluginContext.h>
#include <QtDebug>

void LogActivator::start(ctkPluginContext* context)
{
    m_pPlugin = new LogImpl();
    context->registerService<LogService>(m_pPlugin);
}

void LogActivator::stop(ctkPluginContext* context)
{
    Q_UNUSED(context)

    delete m_pPlugin;
    m_pPlugin = Q_NULLPTR;
}
```

服务A的服务追踪类

服务A的服务追踪类

追踪类，建立时机：

1、可以在封装A服务的时候就建立，作为一种工具向外提供，但是不应该被编译进插件中，它并不是插件的功能而是访问插件的工具；

2、也可以在B插件中建立，完全和A服务独立开，作为访问A服务的一种手段；

3、单独建立一个空工程，为项目中的所有服务建立对应的追踪类，然后放在同一个文件夹中，其他想要的自己使用就行。

注意：B插件如果想要使用A服务，需要service_tracker.h、service_tracker.cpp、A服务的接口类。

本例采用第二种。

service_tracker.h

```
#ifndef SERVICE_TRACKER_H
#define SERVICE_TRACKER_H

#include <ctkPluginContext.h>
#include <ctkServiceTracker.h>
#include "../Log/log_service.h"

class ServiceTracker : public ctkServiceTracker<LogService *>
{
public:
    ServiceTracker(ctkPluginContext* context) : ctkServiceTracker<LogService *>(cont
    ~ServiceTracker() {}

protected:
    // 在 Service 注册时访问
    LogService* addingService(const ctkServiceReference& reference) Q_DECL_OVERRIDE
        qDebug() << "Adding service:" << reference.getPlugin()->getSymbolicName();
        // return ctkServiceTracker::addingService(reference);

        LogService* service = (LogService*)(ctkServiceTracker::addingService(referen
        if (service != Q_NULLPTR) {
            service->debug("Ok");
        }

        return service;
    }

    void modifiedService(const ctkServiceReference& reference, LogService* service)
        qDebug() << "Modified service:" << reference.getPlugin()->getSymbolicName();
        ctkServiceTracker::modifiedService(reference, service);
    }

    void removedService(const ctkServiceReference& reference, LogService* service) Q
        qDebug() << "Removed service:" << reference.getPlugin()->getSymbolicName();
        ctkServiceTracker::removedService(reference, service);
    }
};

#endif // SERVICE_TRACKER_H
```

插件B实现类, login_impl.cpp

```
#include "login_impl.h"
#include "service_tracker.h"

LoginImpl::LoginImpl(ServiceTracker *tracker)
    : m_pTracker(tracker)
{

}

bool LoginImpl::login(const QString& username, const QString& password)
{
    LogService* service = (LogService*)(m_pTracker->getService());

    if (QString::compare(username, "root") == 0 && QString::compare(password, "12345
        if (service != Q_NULLPTR)
            service->debug("Login successfully");
        return true;
    } else {
        if (service != Q_NULLPTR)
            service->debug("Login failed");
        return false;
    }
}
```

插件B激活类，login_activator.cpp

```cpp
#include "login_impl.h"
#include "login_activator.h"
#include "service_tracker.h"
#include <ctkPluginContext.h>

void LoginActivator::start(ctkPluginContext* context)
{
    // 开启服务跟踪器
    m_pTracker = new ServiceTracker(context);
    m_pTracker->open();

    m_pPlugin = new LoginImpl(m_pTracker);
    m_registration = context->registerService<LoginService>(m_pPlugin);
}

void LoginActivator::stop(ctkPluginContext* context)
{
    Q_UNUSED(context)

    // 注销服务
    m_registration.unregister();

    // 关闭服务跟踪器
    m_pTracker->close();

    delete m_pPlugin;
    m_pPlugin = Q_NULLPTR;
}
```

使用插件B

```cpp
// 获取插件所在位置
QString path = QCoreApplication::applicationDirPath() + "/plugins";

// 遍历路径下的所有插件
QDirIterator itPlugin(path, QStringList() << "*.dll" << "*.so", QDir::Files);
while (itPlugin.hasNext()) {
    QString strPlugin = itPlugin.next();
    try {
        // 安装插件
        QSharedPointer<ctkPlugin> plugin = context->installPlugin(QUrl::fromLocalFile
        // 启动插件
        plugin->start(ctkPlugin::START_TRANSIENT);
        qDebug() << "Plugin start ...";
    } catch (const ctkPluginException &e) {
        qDebug() << "Failed to install plugin" << e.what();
        return -1;
    }
}
```

```
19
20  // 获取服务引用
21  ctkServiceReference reference = context->getServiceReference<LoginService>();
22  if (reference) {
23      // 获取指定 ctkServiceReference 引用的服务对象
24      LoginService* service = qobject_cast<LoginService *>(context->getService(referen
25      if (service != Q_NULLPTR) {
26          // 调用服务
27          service->login("root", "123456");
28      }
29  }
```

具体参见项目：ServiceTracker (https://github.com/myhhub/CTK-project/tree/main/ServiceTracker)

‹ 上一篇 (/archives/2021/03/100646.html)

下一篇 › (/archives/2021/03/100644.html)