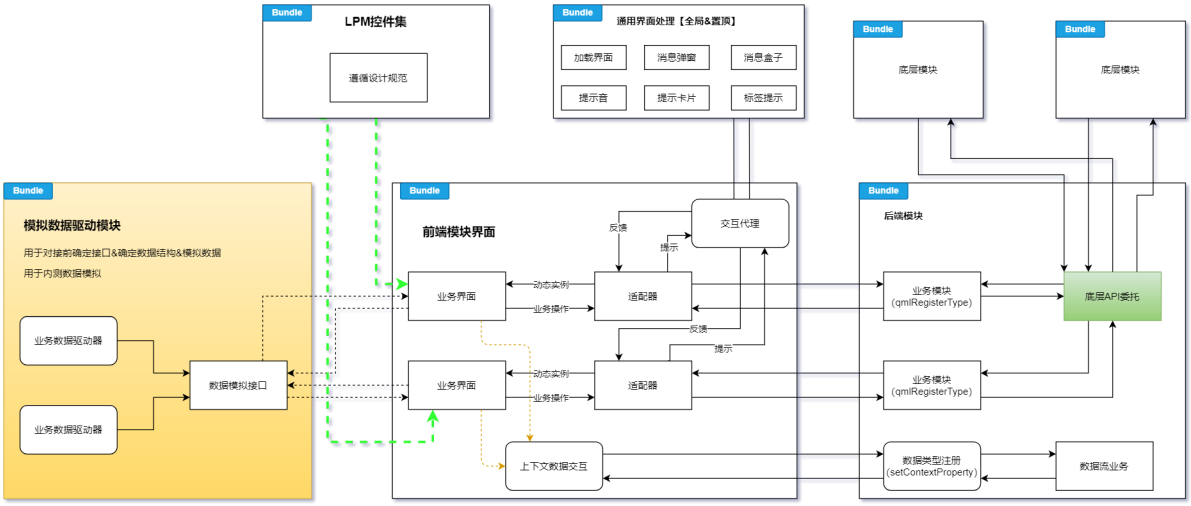
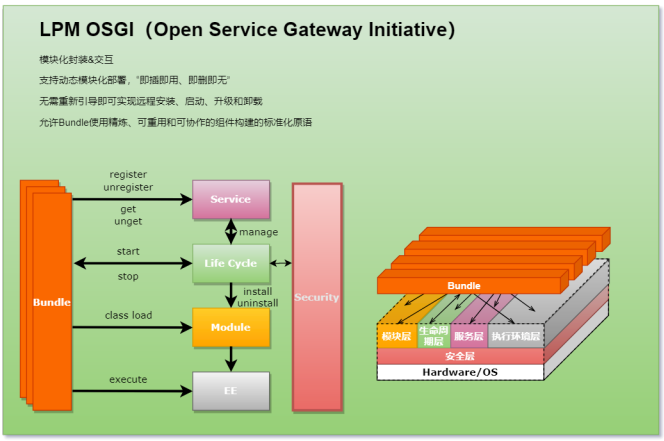
LPM Client前后端对接方式

1、分离设计





安全层

严密控制：对运行在OSGi上的应用进行详细控制

可管理：安全层不定义API进行应用控制，由生命层控制

可选：安全层为可选层

模块层

定义：对C++类/资源进行打包，为用户提供API/资源，OSGi框架下唯一实体

可解析：一旦被解析，其函数将暴露给其他安装在OSGi框架上的Bundle

清单描述：赋予Bundle详尽描述，用于Bundle的协作及签名验证

生命周期层

完整功能：必须实现一个完全覆盖安装、启动、停止、更新、卸载和监控的API

状态反射：该API必须将框架的运行状态真实反映

安全：在权限细粒度较高前提下为Bundle提供安全的运行环境，当然，安全配置必须为可选项

可控：允许远程管理OSGi

可启动：提供标准的框架启动方式

服务层

协作：提供Bundle的发布、查找和绑定彼此服务的机制

动态：服务机制能够直接处理外部应用和底层结构的变化

安全：限制对服务的访问

状态反射：可完全访问服务内部状态

版本控制：提供处理Bundles及其服务随版本更改带来变更的机制

持久化标识：提供Bundles跨框架重启服务跟踪方法

QML与C++交互规范

2、数据交互

常见数据转换

Qt Type	QML Basic Type
bool	bool
unsigned int, int	int
double	double
float, qreal	real
QString	string
QUrl	url
QColor	color
QFont	font
QDateTime	date
QPoint,QPointF	point
QSize,QSizeF	size
QRect,QRectF	rect
QMatrix4x4	matrix4x4
QQuaternion	quaternion
QVector2D, QVector3D, QVector4D	vector2d, vector3d, vector4d
Enums declared with Q_ENUM or Q_ENUMS()	enumeration

数据传输规范：

- C++传输数据到QML端使用json格式
- 列表或者表格数据使用Model/TableModel

3、属性暴露

使用Q_PROPERTY

```
Q_PROPERTY(type name
             (READ getFunction [WRITE setFunction] |
              MEMBER memberName [(READ getFunction | WRITE setFunction)])
             [RESET resetFunction]
             [NOTIFY notifySignal]
             [REVISION int]
             [DESIGNABLE bool]
             [SCRIPTABLE bool]
             [STORED bool]
             [USER bool]
             [CONSTANT]
             [FINAL])
             (READ getFunction [WRITE setFunction] |
              MEMBER memberName [(READ getFunction | WRITE setFunction)])
             [RESET resetFunction]
             [NOTIFY notifySignal]
```

```
            [REVISION int]
            [DESIGNABLE bool]
            [SCRIPTABLE bool]
            [STORED bool]
            [USER bool]
            [CONSTANT]
            [FINAL])
```

4、方法调用

使用Q_INVOKABLE

```
  class Window : public [QWidget](../qtwidgets/qwidget.html)
  {
      Q_OBJECT

  public:
      Window();
      void normalMethod();
      Q_INVOKABLE void invokableMethod();
  };
```

5、信号发射

C++中信号携带的参数名称直接在QML中使用，因此信号参数名称不可缩写。

6、C++对象注册方式

- 对于和后台数据同步的对象使用setContextProperty
- 对于在QML中实例化的类型使用qmlRegisterType

接口遵循moc规范

7、元对象编译

> 由于C++与QML混合编程是基于moc系统的，故需严格遵循该规范

以下例子简单举例

- Q_OBJECT：供moc编译工具识别
- Q_PROPERTY：定义属性，若在QML中使用了Binding，需声明NOTIFY
- Q_ENUMS：定义后，可直接在QML中使用
- Q_CLASSINFO：拓展类的信息
- Q_MOC_RUN：显示跳过moc处理

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Priority priority READ priority WRITE setPriority NOTIFY
notifySignal)
    Q_ENUMS(Priority)
    Q_CLASSINFO("Author", "Oscar Peterson")

public:
    enum Priority { High, Low, VeryHigh, VeryLow };
```

```
#ifndef Q_MOC_RUN
    ...
#endif

    MyClass(QObject *parent = 0);
    ~MyClass();

    void setPriority(Priority priority) { m_priority = priority; }
    Priority priority() const { return m_priority; }

private:
    Priority m_priority;
};
```

对于定义Q_OBJECT的实现类文件中，建议使用以下makefile规则：

```
foo.o: foo.moc

foo.moc: foo.cpp
        moc $(DEFINES) $(INCPATH) -i $< -o $@
```

这保证make在编译foo.cpp之前运行moc，之后你可以用：

```
#include "foo.moc"
```

加入到foo.cpp最后一行，之后，文件中的类定义都是完全已知的

使用moc该注意的：

如果在程序的最终构建阶段出现链接错误，如YourClass：：className（）未定义，或者说YourClass缺少vtable，则说明发生了错误。大多数情况下，您忘记编译或#include包含MOC生成的C++代码，或者（在前一种情况下）包含链接命令中的对象文件。如果使用qmake，请尝试重新运行它来更新makefile。这应该能解决编译错误。

使用moc的限制：

- **类模板没有包含Q_OBJECT，以下例子不可以使用moc**

```
class SomeTemplate<int> : public QFrame
{
    Q_OBJECT
    ...

signals:
    void mySignal(int);
};
```

- **多重继承要求QObject放到第一位**

如果你使用多重继承，moc假定第一个继承类为QObject的子类，同样的，确保第一个子类继承的第一位为QObject

不支持虚拟继承QObject

```
// correct
class SomeClass : public QObject, public OtherClass
{
    ...
};
```

- **函数指针不可以当作信号/槽的参数**

大部分情况下，在打算使用函数指针为信号/槽参数的场景时，Qt认为继承是更好的处理方式；以下为非法例子：

```
class SomeClass : public QObject
{
    Q_OBJECT

public slots:
    void apply(void (*apply)(List *, void *), char *); // WRONG
};
```

当然，你可以这样做，来绕过这个限制

```
typedef void (*ApplyFunction)(List *, void *);

class SomeClass : public QObject
{
    Q_OBJECT

public slots:
    void apply(ApplyFunction, char *);
};
```

有些时候，使用继承和虚函数代替函数指针，Qt认为是更好的方式

- **Enums和Typedefs必须完全符合信号&槽参数要求**

检查信号/槽参数时，QObject::connect()会先在字面上比对参数数据类型，因此，Alignment和Qt::Alignment会被认为是两种截然不同的类型，为解除该限制，确保在定义信号和槽以及建立连接时，参数的数据类型完全一致；比如：

```
class MyClass : public QObject
{
    Q_OBJECT

    enum Error {
        ConnectionRefused,
        RemoteHostClosed,
        UnknownError
    };

signals:
    void stateChanged(MyClass::Error error);
};
```

- **嵌套类不可以存在信号/槽**

以下结构不符合该规则

```
class A
{
public:
    class B
    {
        Q_OBJECT

    public slots:   // WRONG
        void b();
    };
};
```

- **信号/槽返回值的类型不能为引用**

信号和槽允许返回类型，但信号/槽返回引用将会被视为返回空值

- **只有信号和槽才能在`signals`和`slots`区域出现**

moc将会对在类中将非信号/槽的结构放入**signals/slots**区域的行为发出警告