Sniper Python Binding及Property的设计

邹佳恒 林韬

IHEP

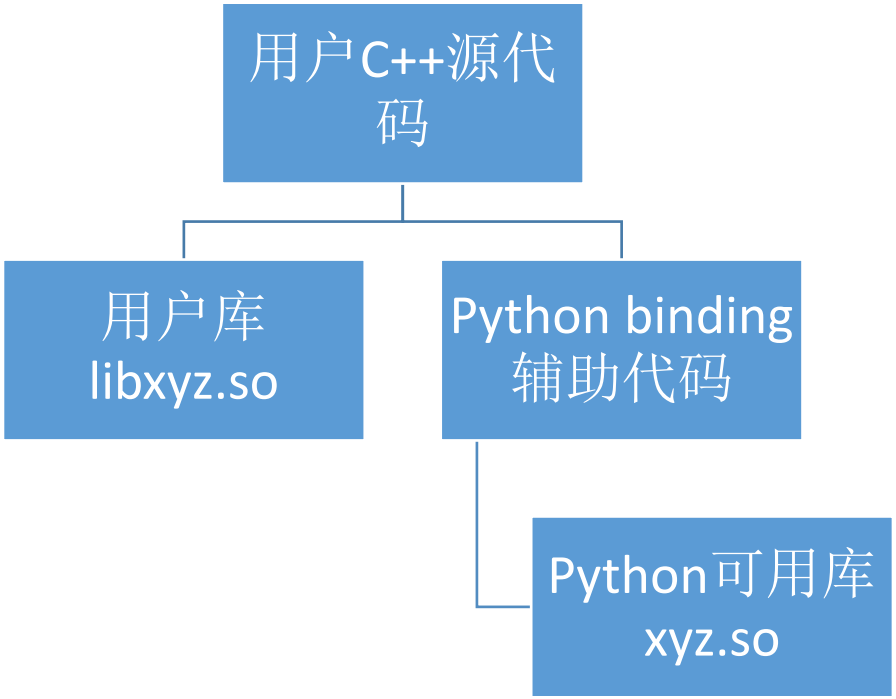
OUTLINE

- Python Binding简介
- Sniper中的Python Binding
- Binding中Property的设计

Python Binding简介

- 为何需要Python Binding？
  - Python语言简洁，标准库强大
  - 解释性语言，便于快速开发
  - 可用于作业配置
    - Sniper希望以Python一致地配置作业
- 使用何种技术进行Binding？
  - 原生的Python C API
  - Boost.Python
  - ROOT中的PyCintex
  - 其他，如Swig，ctypes
- Binding的基本做法？

Binding基本做法



- 用户编写C++代码
- 生成普通的库libxyz.so
- 以手动/自动的方式，生成binding时的辅助代码
- 生成python可用的库xyz.so
- 注：编译链接时可以将libxyz.so与xyz.so合成一个库

Sniper Python Binding

- Sniper的设计理念：保持整体架构及设计的简单
- 我们尽量避免复杂的设计
- 我们的Binding主要围绕作业配置
- 对Python Binding技术选择时的考虑
  - Binding时应该尽量简单，并且技术应该成熟
  - 尽量对框架或用户代码没有侵入性
  - 用户在编写代码时，不应该考虑有binding的存在
- 我们考察了框架Gaudi和Belle2的basf2
  - Gaudi：主要基于ROOT的PyCintex
  - Belle2：基于Boost.Python

Gaudi及basf2 binding的做法

- Gaudi基于ROOT的PyCintex进行Binding
  - 使用ROOT的REFLEX产生需要的字典文件，需要编写一些xml格式的selection文件
- Gaudi有多套作业配置系统，因此Binding时的代码没有侵入性。
  - Job Option
  - Python
- 使用Boost.Spirit进行作业脚本的解析
  - 作业配置属性是以字符串的形式进行传递、解析
  - 解析会占用一定的时间

- Basf2基于Boost.Python
  - 在框架中需要编写少量的辅助代码，以实现将接口暴露至Python之中
- Basf2以Python脚本的形式对作业进行配置
- 框架代码与Boost.Python有部分耦合
  - 在作业配置中，支持简单的数据类型、列表及字典
  - 无法使用更为复杂的类型，例如字典中嵌入字典（作业配置中，可能没有这样的需求）
- 使用运行时对类型进行检查，转换

Sniper的做法——使用Boost.Python

- 从Sniper依赖的库考虑，我们选择使用Boost.Python。
  - Boost库可以认为是C++的准标准库，使用广泛
  - ROOT则是在高能物理界较为流行。另外，随着ROOT6的出现，原先ROOT使用的Python Binding技术很有可能又会变化
- 在框架中侵入Python的代码并不好，但是却带来了便利
  - 某些简单的基本类型及复杂的数据类型无需经过额外的解析处理。
    - 例如：Gaudi中使用Boost.Spirit库进行解析，复杂度提升
- 存在一些自动产生辅助代码的工具
  - 可以使用PY++这样的工具产生辅助代码

Sniper Python的进展

- 对框架中的类进行了binding
  - SniperMgr
  - AlgMgr
  - AlgBase
  - SvcBase
  - PropertyBase
- 用户可以在Python脚本中调用由C++编写的算法
- 用户也可以在Python中直接定义算法，并且调用
- 此处，具体的例子可以参考大亚湾二期DocDB 75-v2
- 去除了Job Option的配置方式

Property的设计

- 何为Property？
  - 此处的Property，是指在C++代码中为某一变量预留了一个接口，以供在作业配置中动态的设置该变量
- 为何需要单独讨论Property？
  - Property直接与用户的使用相关
  - Property需要**跨越**C++及Python
  - 一个典型的问题，就是如何在C++中声明这样一个变量，让其可在Python中进行更改，并且保证两者之间数值的同步。
- 目前，使用侵入式的模式，将Boost.Python库嵌入Sniper的代码中。

Property设计的变化

- 原始的Property，使用管道式的设计。
  - 用户先在Python中设置property的内容
  - 再在初始化某个对象时，调用设置的命令，对该变量进行设置
  - 优点：设计简单
  - 缺点：使用需要注意，因为有先后的顺序。不能保证Python中的对象与C++对象同步

Property设计的变化（cont.）

- 结合Gaudi和basf2后设计的property
  - 使用Boost.Python中的object存储python中的对象
  - 使用C++模版特化技术，针对不同的类型进行转派。
    - 从共同基类派生，便于管理。
    - `template<typename T> class Property;`
    - `template<typename T> class Property< std::vector< T > >;`
    - `template<typename Key, typename T> class Property< std::map< Key, T > >;`
  - 因此，我们目前支持的property属性类型有限。
  - 编译期类型完全确定，不需要运行时判断类型。
  - 代码：
    - https://github.com/mirguest/MirguestIssueReport/tree/master/Prog/cpp/BOOST/python

模板特化例子：	针对int进行特化
template<typename T>	template<>
T f(T x) {return x;}	int f<int>(int x) {return 2*x;}

示例（片段）

C++

```
declareProperty(name, "x", x);
declareProperty(name, "y", y);
declareProperty(name, "vx", v_x);
declareProperty(name, "vy", v_y);
declareProperty(name, "mx", m_x);
declareProperty(name, "my", m_y);
```

Python

```
setProperty("dummy", "x", 42)
setProperty("dummy", "y", 12.34)
setProperty("dummy", "vx", range(5))
setProperty("dummy", "vy", [1.23*i for i in range(5)])
setProperty("dummy", "mx", {str(i):i for i in range(5)})
setProperty("dummy", "my", {str(i):i*1.23 for i in range(5)})
```

declareProperty的定义

Scalar	Vector	Map

template<typename T>
PropertyBase*
declareProperty(
std::string objname,
std::string key,
T& var) {
PropertyBase* pb = new
Property<T>(key, var);
…
return pb;
}
	template<typename T>
	PropertyBase*
	declareProperty(
	std::string objname,
	std::string key,
	std::vector<T>& var) {
	PropertyBase* pb = new
	**Property< std::vector<T>
	>(key, var);**
	…
	return pb;
	}
		template<typename Key,
		typename T>
		PropertyBase*
		declareProperty(
		std::string objname,
		std::string key,
		std::map< Key, T >& var) {
		PropertyBase* pb = new
		**Property< std::map< Key, T >
		>(key, var);**
		…
		return pb;
		}

小结

- Sniper使用Boost Python进行Binding
- Sniper中的基本接口已经暴露到Python之中
- 用户可以调用C++或Python中定义的算法
- 改善了Property的设计，使得使用更为方便

Q & A