User Guide of the Framework of DAMPE Software

The *framework of DAMPE software* will be mentioned in this document only means the *Kernel package* of DAMPE software.

In this document,

1. 概述
	1. DAMPE software的主要组成
	2. Framework的基本概念和目的，对Framework原理的详细说明请参考《Developer Guide of the Framework of DAMPE Software》
2. 安装
	1. 安装依赖软件
	2. 下载安装DMPSW
	3. 环境设置
3. 算法 （algorithm）
	1. 什么叫算法
	2. 如何拓展算法
4. 服务 （service）
	1. 什么叫服务
	2. 默认加载的服务
	3. 如何拓展服务
5. event class
	1. 为什么需要event class
	2. 如何新增一个event module
6. Job option驱动
	1. 作业配置标准流程
	2. 作业执行时序
7. **概述：**
	1. **DAMPE software的主要组成**

严格的说，目前这套DAMPE software分为3个部分：

1. Framework
	1. 驱动模式：

DmpCore，DmpAlgorithmManager，DmpServiceManager

* 1. 功能模块拓展机制：

DmVAlg， DmpVSvc

* 1. Root IO以及数据交换机制：

DmpRootIOSvc

* 1. 其他辅助功能：

统一的Log 信息处理；随机数种子产生（\*\*）；

* 1. DAMPE项目核心参数管理：

Run Mode； 各探测器的关键几何参数；

1. Event module

我们基于root开发DAMPE的event module。Software里面的algorithm之间的数据交换都是以even module为媒介。

关于event module的介绍可参考 组会的报告《Proposal： why we need TObject to construct event module---from USTC 》

1. 其他具体的功能模块

具体针对DAMPE所写的功能模块，如simulation，Raw data conversion等，他们都是针对本实验的具体功能模块。

* 1. **Framework 的目标**

我们为DAMPE科学应用系统数据分析方面提供Framework，目标是：

* 1. 提供统一的开发平台

在该framework下开发一个特定的功能的algorithm或者service十分简单快捷。User只需要overload framework里面提供的接口。这样，用户在在开发时可以集中于精力功能模块的实现，如何驱动，管理，以及数据交流，存储的问题，Core统一负责。

如果您开发了某个很好的algorithm或service。我们可以考虑吸收进DAMPE software。这样为其他用户默认安装之后，其他人可以直接使用别人提供的功能。

对于算法，服务以及event class的具体拓展方式，请参考后面章节。

* 1. 提供统一的运行机制

Framework为所有job提供一个统一的运行机制，用户将算法，服务加载到DmpCore里面之后，Framework自行驱动它们按流程执行。运行逻辑如下：

Step 0. 配置作业（job option中）

Step 1. 初始化（DmpCore::Initialize()）

Step 2. 执行（DmpCore::Run()，不断做event loop，直到完成）

Step 3. 清理及退出工作（DmpCore::Finalize()）

* 1. 用户完全可配置的Job

在本框架下运行的Job，可在运行时按用户的要求实现完全配置。作业的配置有两个方面的含义：

* + 1. 运行时选择一个特定的算法或服务。

对具体某一个算法库的加载，可以推迟到作业配置时来选择。配置成功后，DmpCore才驱动该指定的算法库执行。这种方式是该framework灵活强大的主要体现，对以后协同开发具体功能模块有很大的帮助。

* + 1. 运行时具体算法或服务涉及的相关参数完全可配置。

每一个具体的算法（或服务）都可以在作业配置时对它自己的参数进行配置（DmpCore的某些参数也可在此配置）。对于user想写一个自己的算法，它对python的配置接口已经提供，用户只需要完善其算法关于作业配置的相关代码。

1. **安装：**

软件安装和环境设置流程：

Step 0 安装依赖的软件和库

1. Root
2. Geant4 // echo $G4LIB\_USE\_GDML, 要求等于1
3. SCons // 对python有版本要求，version > 2.4
4. python-devel // python-config，命令存在
5. Boost // 先装python-devel
6. CLHEP // clhep-config 命令存在

Step 1 下载DAMPE software

1. 从 svn上下载

svn checkout <http://dpnc.unige.ch/SVNDAMPE/DAMPE/DmpSoftware/trunk/>

1. 从git上下载

git clone <https://github.com/ChiWang/DmpSW>

以上任一方式可行。下载完成后，进入主目录，里面基本结构如下：



Step 2 安装

1. 产生event module的字典

*bash pre-install.(c)sh*

1. 安装整个软件包

*scons*

会询问你的安装路径，如果按Enter继续，将在当前目录下创建Install，并安装到里面。（即，默认的prefix 为下载目录下的Install）安装之后，进入prefix，结构如下：



Step 3 环境设置

1. 环境设置

*source /prefix/bin/thisdmpsw.(c)sh*

（prefix = 你的安装路径）

1. 检查环境变量

*echo $DMPSWSYS*

*echo $DMPSWWORK*

1. **算法：**

**3.1 什么叫algorithm**

在高能物理实验的数据处理过程中，我们很关注对于每一个event的运算。比如，在一大批事例中挑选出关心的衰变道，首先，需要拿到该事例的相关信息（tracks），然后判选径迹属性，符合条件的即是选择的事件。

对于这种需要对每一个event做loop，进行操作的代码片，我们把它叫做本framework下的算法（algorithm）。algorithm区别于本framework里面的服务（service）， service是不需要针对每一个event做什么操作。

**3.2 增加算法**

 Framework提供严格的algorithm的接口，必须从DmpVAlg这个类派生出来的algorithm才能被DmpCore驱动。同时DmpVAlg提供了如何从python脚本中操作job option的接口。所以，如果你将要创建的这个algorithm严格按照接口要求派生的，将可以直接使用它。此外，DAMPE software 有统一的log 风格（DmpLog），你创建的algorithm可以直接使用。另外，DAMPE software（其中的DmpRootIOSvc）提供了统一的IO以及algorithm之间数据交流机制，新增加的算法如果想要实现和其他算法的数据交流，推荐使用这个类（DmpRootIOSvc）。

3.2.1 继承DmpVAlg，Overload里面的纯虚函数

Framework提供严格的algorithm的接口，必须从DmpVAlg这个类派生出来的algorithm才能被DmpCore驱动。

首先，用户派生的concrete 的algorithm必须overload DmpVAlg里面的3个pure virtual function：

 其次，申明该algorithm需要的event module的指针，或你所需要的其他数据成员。

其他按你的需要增加。

必须要overload的DmpVAlg里面的3个纯虚函数是：

1. Initialize

主要完成初始化工作。该初始化在做event loop的时候对所有event都有效。所以用户自行考虑将需要的内容放到这里。

Framework要求在algorithm的Initialize里**必须要做的工作**是：

1. output event module的创建并向Data Buffer的注册；
2. 如果该algorithm需要其他algorithm的output event module作为自己的input event，必须在此向Data Buffer索取。

下面是RawDataConversion/Bgo/BgoRdcEQM/include/DmpAlgBgoRdcEQM.h中的Initialize的例子

****

打开文件；

设置Connector

创建event module并注册

（bool fIniStatus是用户的concrete algorithm直接从DmpVAlg里面继承的数据成员）

**需要强调的是**，event module只是在Initialize中创建，也就是对于整个Job，只有一个Event module的对象存在。在作event loop的时候，它的值是自然要被用户算法要更新的。

1. ProcessThisEvent

该函数就是algorithm的核心：怎么对每一个事例作运算。

* 必须清理TClonesArray里面的elements

**需要注意的是**，如果你的event module是一个TClonesArray对象所管理的event class，如上面例子中的这一行：



那么，在ProcessThisEvent()开始时，必须调用TClonesArray：：Delete()删除TClonesArray里面的elements（相当于Reset该event）。如：



如果您的算法，在Initialize时从Data Buffer里面读取了其他algorithm（或者一个input root file）的event module，在ProcessThisEvent函数内**不需要每次**再去关联该数据。以及它们对于每一事例的更新，不需要用户处理，Framework会负责实现。

* 当前事例运行状态值设置

ProcessThisEvent() 必须返回一个当前事例执行成功与否的状态值。如果整个Job中加载的若干个algorithms中的任意一个algorithm返回false，将放弃这个event的执行，数据也不会Fill到root 文件。只有所有algorithms返回true，才会Fill这个event。

* 如何让algorithm在运行时关闭整个Job

如上所述对单个事例，ProcessThisEvent必须返回true或者false。如果在一个algorithm执行过程中，发现某种情况（比如Raw data conversion 到了input数据尾部）需要停掉整个Job，必须让它有权利向DmpCore发送信号。但是并不是每个algorithm都需要这个功能。

所以，如果想在一个algorithm中就告诉DmpCore马上停掉Job，需要拿到DmpCore的singleton指针，并调用DmpCore::TerminateRun()。

例如在RawDataConversion/Bgo/BgoRdcEQM/include/DmpAlgBgoRdcEQM.h中：

 

1. Finalize

完成该algorithm的清理工作。

* 必须清理TClonesArray里面的elements

调用TClonesArray：：Delete()

* 清理event module的对象

如果是**output**的event module的对象，要在这里**需要**delete它；

如果是**input**的event module的指针，这里**不需要**delete它。

* + 1. 增加该algorithm的Options

用户新增加的algorithm派生自DmpVAlg，则将直接继承DmpVAlg的数据成员std::map<string, short> OptMap，它提供了一个作业配置选项的容器，同时Kernel里面已经绑定了void DmpVAlg::Set(const std::string &option, const std::string &value)到python中。

所以，如果您新增加的algorithm需要在job option的python脚本中设置某些参量，只需要做以下两件事：(以simulation 的algorithm DmpSimAlg.h为例)

1. 在构造函数中设置可用的option选项



1. Overload DmpVAlg::Set(cosnt std::string &option, const std::stirng &value)



所以，在Job option file(JobOpt\_DmpSim)运行时可以选择使用哪一个Physics List如下：



3.2.3 bind该算法到python中

为了实现运行时选择并加载算法的功能，要求用户用C++代码写的算法类有能力在python中创建。为了达到这个目的，必须使用boost.python库将具体算法类bind到python中。

关于boost.python的详细手册参考链接：

<http://www.boost.org/doc/libs/1_55_0/libs/python/doc/tutorial/doc/html/index.html>

或者参考DAMPE software Kernel/src/DmpKernelBinding.cc 文件

* + 1. 用一个简单的命令创建新算法

上面简单的说明了写一个算法需要写哪些核心内容。实际上，安装DAMPSW并设置好使用环境之后，能够调出这个命令“*dmpNewAlg*”，可以利用这个命令方便地创建一个新算法的模板，您需要做的就是完成模板里空缺的内容。

如RawDataConversion包中，对Bgo EQM开发了一个单独读取Bgo数据的Rdc 的算法，过程如下：



 模板提供了：

1. SConstruct-----编译该算法库的SCons脚本；
2. DmpBgoRdcEQMBinding.cc-----该算法类到python的binding部分代码；
3. DmpAlgBgoRdcEQM.h-----该算法类的头文件，里面写明了你应该实现的函数接口。

 所以，需要用户完善的就是：

1. 完成Readme.md文件-----说明算法的用途，数据对象。
2. 具体的算法实现------src中写DmpAlgBgoRdcEQM.cc文件；
3. 测试用的Job option文件或其他配置文件放入share中。

其他说明：

一般可以直接使用自动生成的SConstruct文件对该库进行编译，如果您的库有其他特别的软件依赖关系，可自行修改SConstruct文件。

如果你需要额外在这个库里面加入其他算法内，并希望它也能在python中创建，需要参照\*Binding.cc里面的内容，暴露算法的构造函数。

1. **服务：**

**4.1 什么叫service**

Service和algorithm的区别在于，service不需要针对每一个event进行什么处理。所以，DmpVSvc这个类的结构基本和DmpVAlg一样，只是没有ProcessThisEvent这个成员函数。

**4.2 DmpRootIOSvc 的使用**

4.2.1 DmpRootIOSvc的功能

目前framework默认加载的service为DmpRootIOSvc。它提供：

1. 读入或写出一个root file
2. 整个job中，对所有algorithm所创建起来的event module的管理
3. 不同algorithm之间数据交流机制
4. 按需要将某些event module定制为output的root file中的Tree/Branch

 4.2.2 对DmpRootIOSvc使用的两个方面

每一个Job中只有一个DmpRootIOSvc的singleton，它是多个algorithm能在同一个Job中串接执行的主要功能部件。对DmpRootIOSvc的使用包括两个方面：

1. 在algorithm::Initialize（）时对DmpRootIOSvc的使用
	1. 从DmpRootIOSvc中读入所需要的input event；

调用：TObject\* DmpRootIOSvc::ReadObject(const std::string &path)

* 1. 将新创建的output event module的对象注册到DmpRootIOSvc里面去
		1. 对单纯的event class

bool DmpRootIOSvc::RegisterObject(const std::string &path, const std::stirng &className, T)

* + 1. 对由TClonesArray管理的一系列event classes

下面是DmpAlgBgoRdcEQM::Initialize()中关于如何使用DmpRootIOSvc的例子：



注册到DmpRootIOSvc时指明唯一的path

1. 在Job option中，对DmpRootIOSvc的option进行配置



 输出结果：

必须用分号（；）隔开



 关于DmpRootIOSvc的原理可参考报告《DmpRootIOSvc and Data Buffer》。

 关于DmpRootIOSvc实现的原理请参考源代码。

**4.3 增加服务**

 与增加algorithm类似，用户拓展一个service需要继承抽象类DmpVSvc，并必须overload里面的两个纯虚函数。一个service也可以有在作业运行时的脚本参数配置功能，这时需要你在写自己的service时定义了option的map。

4.3.1 继承DmpVSvc，Overload里面的纯虚函数

DmpVSvc与DmpVAlg类似，但是只两个纯虚的成员函数：Initialize和Finalize函数。Service不针对各个event进行操作，所以没有ProcessThisEvent的概念。

因此，对一个Service的开发流程基本和Algorithm一样：

1. 继承DmpVSvc并overload两个纯虚函数
2. 完善函数实现
	1. 可在实现过程中设置option map
3. 绑定到python

具体例子参考DmpRootIOSvc的源代码。

4.3.2 用一个简单的命令创建新的service

用“dmpNewSvc Name”创建一个新的service的模板，模板提供基本类容：SConstruct文件，binding文件，service的头文件。用户需要完成的是实现函数。

 ***dmpNewSvc 命令没完成。***

1. **event class：**

**5.1 为什么需要event module？**

从高能物理实验数据处理的流程来看，核心部分是对事例信息的提取，分析，封装的过程。简单的说，高能物理数据分析软件是数据流驱动的软件。对各个步骤中数据类型的定义是软件的重点。

 为了软件各部分更好的交流数据，我们基于root的TObject开发自己的event class类型，由event class作为各个包（各个步骤）之间数据交流的媒介。采用基于TObject的event module的主要优势列举如下：

1. 统一的数据结构，单独的数据模块
2. 有利于DmpRootIOSvc作数据存储
3. 方便algorithm之间的数据交流（只需要一个event class或者TClonesArray的指针）
4. 方便维护和升级，只需要对event class中的数据成员进行修改

**5.2 如何新增一个event class？**

5.2.1 什么时候需要增加一个新的event class？

Algorithm重点关心对数据（event）的处理，algorithm之间通过event class做数据交流。随着DAMPE software数据分析软件的发展，会不断增加新的algorithm，一步一步将event处理成各种形式的物理数据，直到物理结果。对需要什么样的数据结构是在不断发展新的algorithm的过程中探索的，所以，当您按需求开发自己的algorithm的时候，这时需要增加一个新的event class。

5.2.2 如何增加一个新的event class？

DAMPE software官方默认的event class都放在Event这个包下面，并按子探测命名的子目录分类，同时类名有严格的命名规则：DmpEvt{sub-detector}XXX，这里sub-detector = {Psd | Stk | Bgo | Nud}，如果event class不专属于某个子探测，不需要子探测标示。

新增加的event class需要严格按照root的拓展要求来写，才能被root自动识别。

简单的说需要以下个步骤：

1. 写一个event class

和普通的class的写法一致，只是需要从TObject派生

1. event class的头文件里面，用root的宏定义明确写明基于root派生了event class，并写明该类是否有root的IO性能

这个宏为： **ClassDef（yourClassName，IO\_Level）**。其中，yourClassName为新增的event class的名字；IO\_Level标示使用root IO标示）。

我们之所以基于root的TObject开发我们的event，就是想用event class做数据存储，所以都需要IO\_Level >= 1。

在源文件中使用宏 **ClassImp（yourClassName）**，非强制性要求。

1. 写Link define 文件

它也是一个头文件，只是采用了rootcint的语法要求，用来明确指明注册什么class到root中。这个头文件的名字必须包含： **LinkDef**。

1. 产生event class的dictionary

利用rootcint，依照Link Define文件内的内容产生event class的dictionary。

1. 编译dictionary以及event class生成一个动态库
2. 使用时，只需要包含这个class的头文件并链接到这个库。

具体可参考报告《event classes and let root recognize them》

 ***最好做一个命令，自动创建event module的模板给普通开发人员。***

1. **Job option 驱动**

该框架下的作业都是由DmpCore统一驱动的，一个作业只有一个DmpCore的singleton对象。Job option采用python脚本，可以十分灵活的定制需要启动的service，需要运行的algorithm以及他们的关联顺序。前面讲过，基于本框架开发的concrete algorithm 或者service 本身也具有作业运行时参数配置的功能（如果您自己派生的具体的algorithm或service利用了这一点的话）。

**6.1 作业配置有标准流程可以参考如下：**

Step 0 导入 DMPSW核心库，以及需要的算法（或服务）库



DMPSW是核心库，libDmpAlgBgoEQM是一个算法库，前面“增加算法”的例子就是增加的这个库。

Step 1 全局设置



 DmpCore（DMPSW.Core）是作业的最高一级的管理类，所有都是有它管理和驱动。也可以不设置这些参数，那么将使用默认值。

Step 2 配置DmpRootIOSvc



 DmpRootIOSvc（DMPSW.RootIOSvc）是目前唯一默认加载的service，可以直接在此使用。

Step 3 创建并加载算法



在这里创建算法、设置参数，并显示的加载到algorithm manager里面。如果不加载，不会驱动该算法。本例中只需要一个算法，如果要多个算法串行，而且后面的算法需要读取前一个算法的数据，那么在Append到Algorithm Manager的时候，必须严格区分先后顺序。

Step 4 Run job



这里才开始真正的执行。这一部分对所有作业一致。

**6.2 DmpCore，DmpVSvc，DmpVAlg的运行时序**

Framework驱动job执行时， 先启动**所有**加载到DmpCore的service，再启动**所有**加载到DmpCore里面的algorithm。也就是说所有DmpVSvc::Initialize()是先于一切DmpVAlg::Initialize()执行的。这样在DmpVAlg::Initialize()时，就能通过service manager拿到一个具体的DmpVSvc所提供的服务。

下面是运行时序图：



说明：

1. 从Job option开始配置
2. 时间流从上到下
3. DmpCore里面的3个函数驱动一切