**LaS-VPE-Platform平台开发文档v0.2**

# 开发目的

本系统旨在提供一个分布式平台，使各种视频处理以及计算机视觉算法能够在集群上分布式运行，从而高效处理视频数据或实时视频流；并且能够灵活地调整执行顺序和运行参数，以测试不同算法地组合执行效果，以及确定特定应用场景下的最佳效果参数。

本系统目前已实现行人检测跟踪与行人属性识别，未来还将加上行人Re-identification，摄像机拓扑预测等视觉应用。

# 平台与依赖项

本系统主要使用Java语言开发。由于所依赖的Spark Streaming使用Scala编写，且部分Scala类和对象在Java中不可见，因此本系统不可避免地加入了少量Scala代码。本工程采用Maven组织，由于是Java和Scala混编，所以在此强烈建议使用Intellij IDEA进行开发，其对Maven、Java和Scala的良好支持能够大幅提升开发速度，降低环境配置出现错误的可能性。

尽管Java和Scala是跨平台的，目前本系统只能运行在Linux系统上。这是由于目前已嵌入本系统的视觉算法都采用C/C++开发，并只在Linux下通过了编译和测试。未来如需向Windows移植，需准备视觉算法的Windows版本，并编写Windows下相对应的各种脚本。

视觉算法的代码并不包含在本工程的Git仓库内，而是分别存储在各自的Git仓库中，并以子模块（submodule）的形式嵌入在本工程中。在对本工程进行克隆时，可以直接加上--recursive参数，对这些仓库也一并克隆。也可以在克隆之后使用git submodule update --init --recursive进行更新。

本平台依赖Kafka、Spark Streaming、Yarn和HDFS。这些需要安装在准备运行本平台的集群上。虽然严格来说，并不要求集群中所有节点都部署这些平台，但是为了简化集群架构和减少节点间的数据传输，建议将这三个平台部署在所有节点上。本系统理论上可以很容易迁移到Mesos平台上。事实上Mesos平台还提供了GPU资源的分配机制，但此机制会限制本平台同时运行的GPU算法数量，降低平台的灵活性，因此没有采用。

# 运行机制

这里首先介绍一下Kafka、Spark Streaming和Yarn的概念。Yarn是一个分布式运算平台，其能执行的并不只限于Spark Streaming的应用，还可以是MapReduce之类的分布式应用。这些应用定义了如何将数据分布到各个节点上、各个节点如何对数据进行处理、结果如何汇集或输出的范式。这些范式通过一些工具（例如Spark中自带的spark-submit）将范式（jar包）提交到Yarn上，Yarn为该次提交的范式创建一个应用（Application），在这个应用的范畴内负责节点资源分配、运算启动和控制以及系统运行状况监控。这也意味着，相同的范式可以在Yarn上创建复数个应用实例，Yarn上也可以同时运行不同类型的范式，其互不影响。

Spark Streaming也不是只能运行在Yarn之上，其他的分布式平台例如Mesos也可以运行Spark Streaming。Spark Streaming与传统的分布式计算范式Spark和MapReduce不同，他是一个持续性的流式处理范式，也就是说，数据会源源不断地汇入，程序需要不间断地对数据进行处理和输出。我们可以将其视为一个数据处理的流水线。因此，Spark Streaming的应用与Spark和MapReduce不同，后者提交上Yarn之后运行完成后会结束，而前者则是会长期驻留在Yarn上，不断获取和处理数据，直到用户手动结束（kill）掉这个应用，或是应用崩溃，或是平台被关闭。

在本系统中，每一个视觉算法被视作一个模块，同时还包含一些辅助性的模块，例如消息处理和任务分发（Message Handling）模块，以及数据管理（Data Managing）模块。每一个模块分别作为一个Spark Streaming范式，提交到Yarn上独立长期运行，互不影响。

这些模块都定义了特定的输入，比如说对于最开始的视觉算法（例如行人跟踪），输入数据即是视频数据，而处于后续阶段的视觉算法（例如行人属性识别）的输入还应该包括前驱算法的输出。所有模块的输入都统一由Kafka获取。Kafka是一个高效的分布式实时消息分发系统。在本系统中，我们将其作为Spark Streaming的输入源，并将模块间的消息传递（例如视觉算法间的结果传递）都统一使用Kafka进行发送。

一般的Yarn + Spark Streaming应用程序都是将自己的程序打包成一个jar包，然后使用Spark Streaming自带的spark-submit工具，将jar包提交到Yarn上运行。本系统为了能够灵活处理参数，将任务提交的功能封装在了自身的jar包中。在启动本系统时，以SystemEntry这个类作为主类。这个类能够分析输入参数，决定该执行哪一个系统模块，由此生成相应的提交参数，再将自身的jar包提交上Yarn来执行。在提交到Yarn上执行后，**便以该模块的类作为主类**，启动该模块的Spark Streaming任务。这也就是为什么本工程中包含多个主类。本工程在sbin文件夹下包含了启动不同模块的linux脚本。

# 基于Spark Streaming的分布式处理

Spark Streaming并不是消息粒度的流式处理，而是批（batch）粒度的流式处理。输入的数据会按照时间窗口（Window）分批，每一批构成一个RDD。RDD（Resilient Distributed Datasets）是Spark里就有的概念，其可以理解为分布在众多节点上的一批数据。Spark中的操作都是对一个RDD中的所有数据进行的操作，例如对RDD中每条数据进行一次加法，对每条数据进行条件筛选（Filter），将RDD的所有数据汇集（Reduce）到一个本地集合中等等。Spark Streaming的底层操作其实就是基于Spark实现的。比如说，Spark Streaming定义了对每条数据各取平方，再全部相加得到结果的流式处理流程，那么其实现就是，每次汇入的数据构成了一个RDD，则对此RDD内的每条数据取平方，得到一个新的RDD，再对这个RDD进行基于加法的汇集操作。进一步，根据时间窗口的设置，输入源每隔一段时间都会产生一个新的RDD（这个RDD可以是空的，不包含任何数据），那么这个过程就被应用在每一个RDD上，如此达到流式处理的效果。

同时，由于RDD的数据是分布在各个节点上的，相应的操作也会在各个节点上独自运行。比如说数据a位于节点A上，数据b位于节点B上，在对所有数据进行平方操作时，节点A就负责将数据a平方，结果放在新RDD在节点A上的分区（Partition）中；节点B则负责将数据b平方，结果放在新RDD在节点A上的分区中。这样一来，新旧RDD在数据的分布上是完全一致的，计算过程也不需要节点间的数据交换，从而实现了高效的分布式计算。

注意在这里我们提到了RDD分区的概念。严格来说，分区与节点并不等同，一个节点上可以存在多个分区，也可以不存在任何分区。当存在多个分区时，受制于节点自身的计算资源，多个分区中的数据也只能顺序处理。而不存在分区的节点则无法参与分布式计算。因此我们在进行系统规划的时候，都尽可能地让每个节点上有且只有一个分区，并且分区内的数据条目数量尽可能一致，这样所有节点都能参与到运算中，并且处理时间相近，不会出现单个节点处理时间太长，拖慢了该RDD的处理时间（Spark Streaming中同一时间内只能处理一个RDD）。

在本系统中，我们使用Kafka作为输入源，将视觉算法分散到各个节点上，并行地对Kafka中传来的消息进行处理。比如说，对于行人检测算法模块，Kafka中传来的消息是需要处理的视频片段在HDFS中的存储地址，该模块直接解析每个RDD中的地址，从HDFS中读取视频，并执行行人检测算法，最后将检测到的每条行人轨迹再作为消息，通过Kafka发送给后续模块。

# 视觉算法的封装与嵌入

目前本系统仅支持Java封装的视觉算法。使用C/C++编写的算法能够很容易地使用JNI进行封装。使用其他语言（如Python和Matlab）编写的算法，目前尚未有简便的方法进行封装，建议首先使用C/C++重新实现，再使用JNI进行封装。

对使用深度学习框架（如Caffe，Torch等）的算法，首先将模型统一转换到Caffe2上，再使用C/C++封装成一个简单的工具函数/类，再使用JNI对此函数/类进行封装。

在使用JNI进行封装时，首先明确此算法在Java中应该提供什么接口，这些接口需要调用什么Native函数/类方法。后者定义为private native方法（无方法实现），前者定义为public的Java方法，调用后者实现相应功能，由此定义出Java类。然后命令行进入src/java目录，使用javah处理此类。比如此类全名为org.cripac.MyClass，则直接javah org.cripac.MyClass。处理完之后会生成一个C头文件，这个头文件中定义了若干C函数，这些函数与Java中定义的Native方法一一对应，相当于Java在调用这些Native方法时，会在内存中寻找这些名字的函数，由这些函数执行相应的功能并返回结果。接下来的任务就是实现这个头文件中定义的函数，将这些实现编译成为一个动态链接库。将此动态链接库以及其所依赖的动态链接库（使用ldd命令可以查看，可以不包含系统环境中已安装的）放在java.library.path里（在Hadoop环境下，应该放在所有节点的${HADOOP\_HOME}/lib/native下，见sbin/install.sh），Java类在静态初始化时，调用System.load()来加载此动态链接库。

在实现JNI的函数时，这些函数的输入有原生类型，例如jint、jlong，这些可以直接与C/C++中的类型相互转化（实际上就是宏定义出来的别名），也有JVM环境下的对象，例如jobject，jfloatArray等，这些需要调用JNI的函数，与JVM环境交互进行取值和赋值。

JNI并不支持类，所有JNI生成的都只是C函数，但是我们仍然可以对C++类进行封装。方法是，我们定义出native函数来申请C++对象，这个函数返回出一个long类型，即是申请出来的对象的指针。我们将这个long类型保存在Java类中，作为一个成员变量。在定义调用C++类的方法的Native方法时，我们在参数中加上这个long类型；在JNI实现中，我们将这个long类型强制转化成C++类的指针，通过这个指针获得之前申请的对象，再将其他参数转换并传递给这个对象的方法，完成调用。同样的，我们还需要定义释放该对象的Native方法，并在Java类的finalize方法中调用此方法。

如前文所述，视觉算法存储在独立的Git仓库内，本工程将这些仓库以Git子模块（Submodule）的形式引入。这些工程必须都能够实现自动编译，推荐使用CMake。sbin目录下包含了对所有子模块进行自动编译和安装的脚本，其中native文件夹下包含每个算法各自的自动编译脚本，以及将编译出来的动态链接库拷贝到lib/x64下；build-native-libs.sh汇总调用这些脚本；install.sh则先调用build-native-libs.sh，再将lib/x64中的动态链接库远程拷贝到集群工作节点中。在新添加算法时，也应该按照这种形式，添加相应脚本。

# 消息监听与算法执行顺序规划

在平台开发初期，我们曾采用过为每一个模块的每一种输入建立一个Topic的方案，但后来观察到这种方法会给Kafka带来较大负担。因此本平台现在采用共享Topic的方式，所有模块的每一种输入类型对应一个Topic（所有的输入类型在枚举类型DataType中定义，在后续开发中可以添加）。比如说，有很多模块都将行人轨迹（Tracklet）作为输入，那么我们在Kafka中只为其建立一个Topic。所有需要行人轨迹的模块都将监听此Topic并收取其中的内容，但是只处理指定发送给自己的内容，舍弃其他内容。

这是通过对数据进行封装实现的。对所有发送至Kafka的数据，我们都将其用TaskData类进行封装，再将整个TaskData对象序列化成二进制序列，发送到Kafka。后续模块收到这个序列后反序列化得到TaskData对象，先判断是否是发送给自己的，如果是，再从里面获得原始输入数据（predecessorRes字段）。

我们将每一个模块的每一个输入源定义为一个Port。当然，如果一个模块需要从两处提取同一个类型的数据，并需要加以区分，那么即使是同一个类型，也可以定义成两个Port。TaskData中包含一个destPorts字段，指明自身是发送到哪些Port的，那么模块就可以通过查询自身所在监听的Port是否包含在destPorts中，来判断这个TaskData是否是发送给自己的。

destPorts字段是在发送方指定的。出于灵活性考虑，每一个模块自身并不知道自己要发送给哪些模块。这是从发送给自己的TaskData中动态查询的。TaskData中还包含一个ExecutionPlan，这个相当于一个执行规划图，指定了哪个模块的输出要发送给哪些后续模块。一个模块在准备自己用于发送的TaskData的时候，首先在当前正在处理的TaskData中找到自己在ExecutionPlan中所处的位置，再从中找到自己所需要发送的Port，加入到新的TaskData的destPorts中。当然，新的TaskData也应该继承原来的TaskData中的ExecutionPlan，一并发送给后续模块。

这就需要输入给算法模块的最初始的TaskData中就包含规划好的ExecutionPlan。在工程规划中，这点应该是由用户在WebUI中先绘制任务流程图，再由Web服务器转换成ExecutionPlan，发送给初始模块。但是由于目前WebUI与系统尚未完成整合，所以我们现在使用一个模块代为生成ExecutionPlan，其名为MessageHandlingApp。这个模块从Kafka监听一些简单的命令（这些命令目前由MessageHandlingAppTest生成，可以调用sbin/run-command-generator.sh来自动发送），根据这些命令生成相对应的ExecutionPlan以及TaskData，发送给算法模块。

在上述描述中，其实存在一点不准确的地方，这是为了方便读者理解而设置的。事实上，Port并不直接属于模块。本系统在模块下面还细分出了一个概念，称之为Stream。Stream是一系列Spark Streaming操作的集合，但是限定了这一系列集合只能有一个输出源。一个模块可以包含若干个Stream，比如数据管理模块可以包含行人轨迹保存、属性保存、实时视频流切割等多个Stream。这是为了不必为每一个细小的功能都启动一个Yarn应用，从而节省资源消耗。Port确切来说是属于Stream的，因此Kafka的通信并不是模块到模块，而是Stream到Stream。这样做可以为统一模块的不同的输出指定特定的目标Port，从而增强平台的灵活性。当然，编写模块时并不一定要遵从这种设定，例如MessageHandlingApp就没用到Stream的概念。

另外，每一个Port自成一个Kafka Consumer Group，所以即使在平台中开多个模块实例，也不会出现一条数据被重复处理的情况，因为一条消息只能被同一个Group中的一个消费者消费。不过也不推荐开启多个模块，目前尚不明确这种行为会不会造成Checkpoint混乱。

# 数据管理

目前输入的视频数据都来源于HDFS。为了不增加Kafka的负担，原始视频数据并不通过Kafka进行传输。使用原始视频数据的模块（例如行人检测跟踪模块），都是收取其在HDFS上的地址，再自行从HDFS中获取视频数据。

所有模块需要保存的输出数据都经由Kafka发送到DataManagingApp，进行统一保存。

行人检测跟踪算法的行人轨迹结果保存于HDFS。其组织形式为，一个视频片段自成一个文件夹，文件夹下细分每次处理任务的结果文件夹，其命名为该次处理任务的UUID串。此文件夹下包含该次执行识别出来的所有轨迹，每条轨迹自身是一个文件夹，其中包含一个info.txt文件，记录了该条轨迹的位置信息（Bounding Box序列），以及这些Bounding Box所框出来的行人的图片。

这种保存形式会带来一个问题，就是HDFS中存储的小文件太多，导致命名服务器速度变慢。因此我们对每次任务的结果进行Hadoop Archive（HAR）打包。这是通过新建一个MapReduce应用实现的，因此在YARN的管理页面可以看到许多的MapReduce应用。这个过程非常缓慢，所以我们在Driver端的一个独立的线程中进行。Executor在将轨迹保存到HDFS之后，通过Kafka向该线程发送打包请求。该线程收到请求后，判断是否可以以及是否需要进行打包，再执行。

其他体积较小的结果，例如行人属性，则保存到图数据库。

# 其余的细节说明

## ParallelExecutor和RobustExecutor

这两个类是对任意操作的封装。ParallelExecutor能够自动或手动设定并行数，使用外部指定的数据操作（Consumer），对一个数据集合进行并行处理。RobustExecutor则是确保输入的操作不会因为某些短暂存在的问题（例如网络不稳定）而失败，当操作失败时自动重试，直到操作成功或达到重试最大次数。

## 本地终端监控与日志类

为了方便调试和检测系统状态，我们将系统中我们自行输出的日志（不包含Spark本身的）通过Kafka发送回我们启动模块的终端，同时也将SparkLauncher（本系统中等同于spark-submit的存在，向Yarn提交任务）的输出通过多线程实时打印出来。详见SystemEntry类。

在发送日志时，我们使用SynthesizedLogger类。这个类记录的日志有三个去向，一是直接打印到stdout，二是使用log4j进行日志记录，三是向Kafka终端发送到终端。若终端被关闭，则日志在Kafka中积累，不会影响其他日志记录。

## 模块的更新

为了系统平台的鲁棒性，我们启用了Spark Streaming的Check-pointing功能。当集群宕机或平台被强制结束时，正在执行的任务会在HDFS上有状态记录，这样当平台重新启动时，便能够继续执行上一次仍在执行的任务。

但是这也带来了一个问题，当更新一个模块的代码之后，重启模块并不能应用最新的代码，甚至产生错误，因为系统会首先从Checkpoint中进行恢复，导致版本不对应。因此更新代码后，如果之前的模块所在处理的数据仍有价值，应该首先让要更新的模块先把当前需要处理的数据处理完，再删除其Checkpoint文件夹（例如MessageHandlingApp的Checkpoint存放在checkpoint/message-handling），再启动新版本的模块。

## 对象序列化

建议使用系统自带的SerializationHelper进行序列化和反序列化，不要使用Apache的序列化工具，后者在之前版本中存在问题，不知道目前的状况。

## 单例

为了节省系统资源以及避免错误，我们设定某些类的对象为单例对象，例如SynthesizedLogger和检测跟踪算法。这是通过使用Singleton类进行封装实现的。Singleton是一个泛型类，指定其类型后实例化出来的就是该类型的单例管理对象。这是通过初始化是指定类名，其在之后获取单例对象时，通过类名在静态集合中查询对象实现的。Singleton类采取惰性初始模式（Lazy Initialization），直到调用其getInst()方法获取单例对象时才对对象进行初始化。同时，Singleton类是可序列化的。因此，我们可以在Driver代码中定义Singleton，在Executor代码中调用其getInst()方法，这样Driver中并不会实例化对象，而每个Executor进程中都能简单地获得一个单例对象，而无需自己再设定该如何初始化对象。

## 打包与测试

使用mvn package –Dmaven.test.skip.exec可以跳过测试的执行，但是生成测试类的代码。MessageHandlingAppTest的代码目前包含在测试代码中，如果不生成测试代码会导致其无法更新。

测试代码可以用来验证一些类的正确性。但是某些算法类耗时较长，或是当前编译机器不支持其测试（例如没有支持CUDA的GPU），那么就需要使用上述命令跳过测试执行。尽管如此，仍然鼓励开发者尽可能为自己的类添加测试代码。

# 未来开发任务

## 与WebUI的整合

尽管WebUI的结果展示功能并不需要与本平台进行交互，本平台仍然依赖WebUI进行任务规划。在当前的WebUI设计方案中已经包含了任务流程图的绘制功能，接下来的任务就是根据用户所绘制的任务流程图，自动生成ExecutionPlan和TaskData，发送给平台中的算法模块，也就是替换当前的MessageHandlingApp。这部分的实现可以参照当前的MessageHandlingApp。

## Caffe2的维护

本系统使用本项目组维护的Caffe2。此Caffe2需定期与最新的官方Caffe2版本进行合并，以确保不会因为版本过久而无法再添加对最新的算法的支持。同时在此版本上进行的修改最好也能通过Pull Request合并到官方Caffe2上。

## Python代码的嵌入

不少视觉算法使用Python进行开发，虽然可以翻译成C/C++再封装进Java，但是需要较大工作量。而Java对于Python的封装又仅限于JPython，与通常使用的CPython不兼容，也不能简单封装已写好的Python算法。

实际上Spark Streaming也带有Python的接口，所以理论上来说可以用Python单独实现算法模块。但是存在一个问题，那就是前文所述的Kafka消息封装类TaskData，目前是直接将Java对象序列化后发送给Kafka。这种封装方式在Python中就算反序列化出来也不能得到正确的结果。

一种可能的解决方法是，使用Json对对象进行序列化。这需要将ExecutionPlan重新设计，因为当前的ExecutionPlan采用的是保持对Node的引用来组织流程图，这种实现方式在Java中比较高效，但是经过Json序列化和反序列化之后并不能保持信息。重新设计的思路是，给每个Node设定一个ID，使用ID来记录Node间的连接关系以及检索Node信息。