设计文档

一、项目整体架构分析

目前，在同一个storm集群里运行3个不同的拓扑结构，这种方式存在如下缺点：

1. 建立不同的拓扑结构运行在在同一个storm集群里可能发生在后期不断的添加拓扑结构时，发生资源不足而导致相应的业务逻辑无法运行的情况。
2. 发生业务逻辑发生改变时，不能方便的进行业务逻辑维护和业务逻辑提交，先必须暂停原来的拓扑计算，在向集群中重新提交拓扑。
3. 项目中，部分代码耦合性比较高，不方便后期的维护修改。

针对这3个不同的日志项目分析，有如下共同点：

1. 日志都来于Kafka，通过业务配置Zookeeper信息获得kafka信息。
2. 抓取日志后，都要进行解析日志的操作。
3. 抓取日志后，都要进行日志过滤的操作。
4. 日志过滤之后，再进行业务的逻辑操作。
5. 日志处理都用到了Map类似的数据结构。

因此可以将这个过程提取出来，形成不同日志服务共同的基础。将这原运行在3个不同拓扑结构的服务集成到同一个拓扑上来。

原项目结构，如图1：

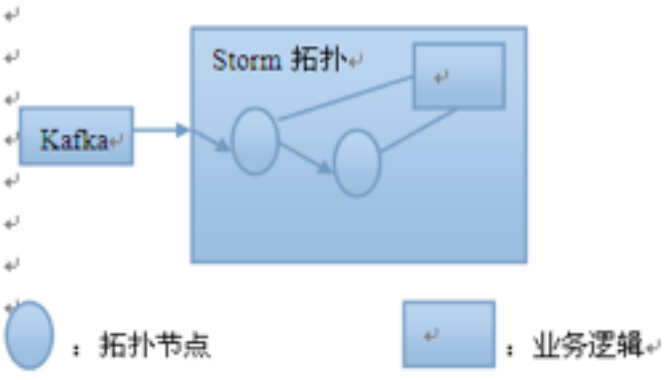

图1 原项目结构

原日志监控、日志统计、推送服务项目都运行在类似于上图的3个不同拓扑里。项目的业务逻辑写入storm拓扑逻辑中，造成模块之间的耦合性过高。

针对上述问题，我们可以进行一些改进工作，将所有的日志服务运行在同一个Topology拓扑结构里面，业务可以像插件一样部署到拓扑结构里，实现业务服务即插即用，轻松卸载，本地即可部署开发的功能。最大限度地提高代码的可复用性和业务易维护性。

基于此，我们的设计可以是这样，如图2。

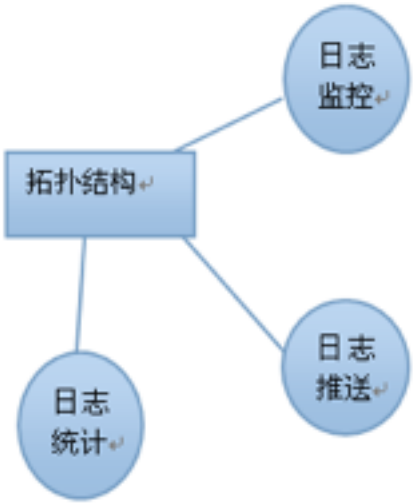

图2 改进设计

为了实现业务即插即用、本地部署卸载的功能，我们将文件系统管理功能从这拓扑结构里剥离开来，可以单独形成一个功能模块，如图3。

图3 剥离文件管理设计改进

在原来的3个项目中，拓扑集群消耗3个不同的topic的日志，为了减少复杂度，现在需要将他们合成一个topic，方便拓扑集群读取和处理。通过原日志的log_type来区分不同的业务逻辑。

输入分析：

对于项目，有两种不同类型的输入。一种为系统行为产生的日志，另一种是用户自定义输入，自定义的输入可以是一条sql语句（sql语句具有定时的功能，如只执行一次，每天等），也可以是具有含义的某些字段，如图4。

图4 系统输入

对于用户输入，有个排重的过程，排重门槛可以有多个，输出的结果或推送或写入数据库。

二、功能模块分析

总体描述： 用户开发完成业务逻辑后，将业务逻辑打包成一个jar文件，将jar文件上传到业务文件管理，同时将jar文件对应的log_type告诉业务文件管理。业务文件管理查看自己是否拥有属于这个log_type的jar。若含有这个jar文件，则认为这是一次更新操作，若不包含，则认为是一次添加业务逻辑的操作。业务文件管理保存这个jar文件，并记住这个log_type对应的jar信息。同时，通知storm拓扑结构，下载该jar文件。

1. <topic,jar>未部署

storm集群获取jar信息后，解析jar的配置，通过这份配置，通过配置，解析日志信息，用户也可以自定义日志解析器解析日志，获得日志处理结果后（解析出错如何处理，ack?），**运行jar文件定义的拦截器，执行拦截器拦截日志功能，过滤后的日志条目交给执行业务逻辑节点进行业务逻辑的执行。**

2. <topic,jar>已经部署

对于已经部署的jar，则在需要运行jar的节点进行热替换，如何停止当前运行的jar?

当用户想卸载一个业务服务时，我们通知文件管理系统，由管理系统通知storm集群，storm集群卸载对应的服务。

当用户想重启一个业务服务时，我们通知文件管理系统，由管理系统通知storm集群，storm集群重新加载对应的服务。

对于一个业务逻辑，我们主要加载依赖的环境、执行业务逻辑判断、业务逻辑输出（插入数据库）这3步骤操作。

对于业务逻辑的拦截器，输入的是我们的日志信息，返回的结果是确定是否执行下一步操作，提供一个抉择，我们必须能在拦截器里控制错误是否要消息重发?

需要的几个拓扑节点:

1. **Jar文件管理模块。**

添加这一模块是为了支持项目业务的热部署，卸载，即插即用的功能. 首先分析一下Jar文件的使用场景:

1. 用户新添加一个jar文件，它有可能可以发射数据，storm需要解析该jar文件，判断是否可以，如可以，则将该jar文件添加到可发射数据源队列。

2. Storm一条日志流入而相应的Bolt节点无法处理该日志对应的服务，则该节点请求下载该jar文件。

3. 用户删除该jar文件，那么对于可以产生数据流的jar文件，则没有必要在发射数据，其它节点的jar类则加入垃圾缓存。

4. 用户重启jar 服务，则如同1。

一个Jar文件对应的即是一个业务服务，如。我们的日志监控，日志分析推送服务。

为了解决storm解析失败或者storm重启/发生错误，记录storm解析jar文件的信息(如解析失败，成功），当storm准备下一次查询时，总是将该临时文件的信息以及storm特征值一起发送至文件服务管理模块。解析失败的次数最多不超过3次。

综合上述分析，我们需要两张表来记录信息：

表名: jar

字段	属性	字段说明
service	varchar	Jar 对应的服务名称
path	varchar	Jar 文件对应的地址
status	int	Jar 文件状态，(可用，删除，废弃）
id	int	Jar 主键
version	int	Jar 版本号
topology	varchar	Jar 对应的消费topology

status:
0, 可用，用户添加一个jar文件或者重启一个服务都可以进入此状态。
1，删除， 删除jar文件以及与该jar文件相关的记录，包括消费历史。
2，废弃， jar文件对应的service变得不可用，但相关记录还在。

表名： jar_history

字段	属性	字段说明
componentid	varchar	消费Jar文件的storm组件的特征值
count	int	Jar文件消费次数
jarid	int	Jar表组件，外键
id	int	Jar表主键
status	int	Jar文件状态
version	int	Jar文件版本号

2. Jar文件服务不能用

对于这种异常，我们应当继续保证storm集群能够运行当前的服务，并能给出相应的提示。

根据要求，jar文件服务必须要有的功能，由storm组件提供确认。

1. 提供当前storm component废弃/删除/可用的jar列表。

服务根据storm组件请求信息(可用)与jar表信息匹配，找出不一样的jar文件列表，如果发现storm组件中jar文件为可用而表中为删除(没有该记录)/废弃状态，则通知storm组件更新其中状态。

2. 提供根据service下载相应的jar文件。

3. 提供增删改查服务接口。
4. 查看jar文件消费历史。

对于storm 来说，主要使用到1和2两个功能。

功能定义：

1. method=findChangableList&type=storm&component=storm component id post
   usable=jar json[可用jar json]
   response changed jar json.
2. method=findJarByService&service=serviceName

2. Storm部分。

将原来设计中的模型构建和拦截器两个节点合二为一，成为一个拓扑节点，提供一个模型构建的拦截器交给用户实现。

在原来的storm拓扑中，如何清理掉过时的class对象，即移除的业务逻辑对象。每个节点都有自己的类加载器管理的功能。

一个jar文件对应一份配置和一个topology的名字，JVM进程如何加载。 由一个Worker进程的标识发送请求。

（使用懒加载，一条消息不能处理时，有不能处理的线程发出一个请求文件服务器 下载对应的jar文件，提高服务系统的容错性）。

一条消息到来，如发现不能处理这则消息，则将消息对应的业务名称以及topology name请求文件管理系统，如能处理这则消息，则走逻辑。

拓扑文件管理节点 负责广播jar的更新和删除事件，JVM 进程卸载相应的jar文件。

提供的jar包 需要实现的功能：

解析业务逻辑的配置：

1. 用户可以自定义数据源（sql语句）
2. 用户可以自定义模型解析器
3. 用户可以自定义拦截过滤器
4. 用户可以声明业务日志格式
5. 用户自定义业务逻辑入口【单例|多例】
6. 用户需自定义业务逻辑的名字
7. 文件系统的日志信息

配置的文件名称必须为config.xml,且位置必须在jar的根目录下。

拓扑结构设计，如图5：

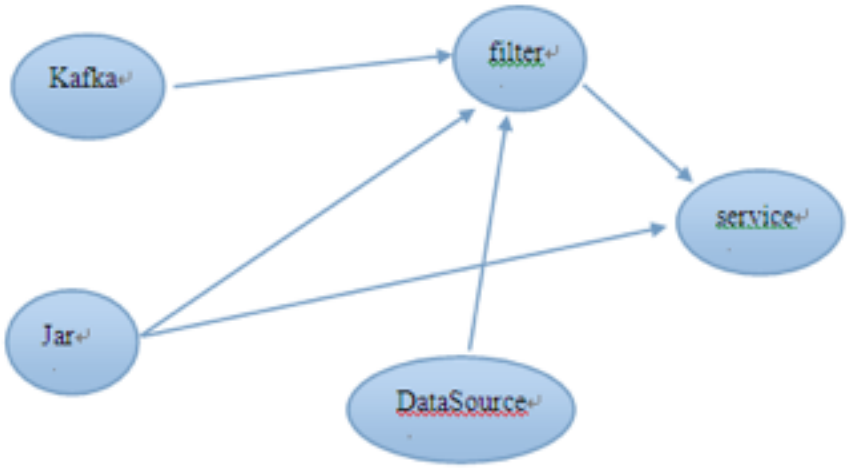

图5 拓扑结构设计

在数据源以及Jar文件管理两个storm组件，这些都需要jar文件服务的功能，在启动时，注册文件服务，当有Jar文件发生更新时，则通知这两个组件。

在当前的JVM中已存在相应服务的Jar文件，同一个进程没有必要再次访问文件服务，使用缓存可以解决这一个问题。

对于一条消息来说，比如一条日志，用户可以不用定义自己的filter和日志解析器，这条日志直接流入到service这个storm组件中。当发现这条日志对应的service。

在部署jar文件的时候，用户可以在参数中设置一些storm的运行信息，例如拓扑的acker的数量和 topology.max.spout.pending的一些信息，不同条件的差异性实现交由用户处理。

定义了数据源、日志解析器、过滤器和拦截器，服务入口这几种角色，用户实现它们定义的接口完成具体的不同业务逻辑行为，这些实现在storm的worker中是单例存在的，因此当涉及到这些实例操作成员变量的行为时，需要考虑到线程安全问题。

数据源：即生产tuple，tuple可以来源于队列，数据库，网络等，用户需要实现对应的接口，可以统计当前服务tuple消息数量，tuple处理成功/失败数量。

问题： 如何判定tuple处理成功或失败?

数据源生命周期： 初始化(只会执行一次），执行生产tuple逻辑，释放实例(只会执行一次）。

日志解析器:

即解析日志，根据相应的日志格式解析原始日志。

过滤器:

即过滤日志，将不符合业务逻辑的tuple过滤，业务可以定义多个过滤器形成过滤链条，

完成对业务的过滤。过滤器和日志解析器在实现上在同一个JVM中。

服务入口:

即业务逻辑入口，业务逻辑主要实现的地方。

日志解析器、过滤器和服务入口 均具有三个不同的生命周期:

初始化，主方法，释放(destory)

存在的一些问题:

在tuple 对应service的jar加载到JVM内存之前，这部分未处理的tuple该怎么办? 如何处理在执行定义的接口出现的异常或错误信息。

在storm中，有一个专门服务在业务入口之间传递消息的对象Input，从这个对象中，可以获取当前对应的storm组件的输入信息，也可以往这个对象中注入信息，注入的信息分为两大类，即可序列化和不能序列化的信息，获取信息时，先不能序列化的信息，其次才是可序列化的信息。

4. 异常模块

当调用发生异常时，一律上报异常，交由storm框架来处理。 异常类型分为两类：可上报的异常和可不上报的异常。

异常模块定义规范：

最底层异常抛出异常时，封装好定义好的异常，输出异常堆栈信息，调用者获取异常后，输出日志信息，向上层抛出该异常，顶层调用者获取异常后 输出日志信息，对于可上报的异常，上报异常信息。

5. 日志模块

获取storm定义对应的服务日志句柄，句柄名称为对应服务的名称，在storm目录下的logback/cluster.xml文件中定义。若有，则向实现日志接口的实例中注入日志句柄，日志的具体格式由每个不同的业务逻辑来定义。

6. 支持业务本地调试

测试的拓扑类型，如图6：

图6 支持的拓扑类型

支持五种拓扑测试。

本地测试时，用户需要提供 配置文件、测试的拓扑类型、zookeeper路径地址、topic名称、storm组件参数的信息，信息的格式以 “key=value”来区分：

参数格式：

config-file=配置文件地址 topology-type=[1|2|3|4] [zookeeper=xxxx topic=xxxx topology-name=xxx] [kafka-spout=1|jar-spout=1|datasource=1] filter-bolt=2 service-bolt=3 worker-num=1

为了简化配置，如果省略一些信息，则采用默认的设置

config-file省略，则默认读取当前路径下的config.xml文件

topology-type = 1

Zookeeper 将与storm的zookeeper路径一致
Topoic=”test“
Topology-name=”test”
Storm组件的参数设置默认都为1

也可以对storm参数进行设置进行调优功能，如：
storm.messaging.netty.buffer_size=16

业务测试步骤：
将storm部分打包成jar文件，包括必须的依赖部分，在业务项目中引入该jar，写好main方法，调用storm部分的main方法，即可进行业务调试。如：

```
public class App {
                public static void main(String[] args) throws Exception{
                    Class.forName("com.kuxun.kxtopology.LocalApp");
                    LocalApp.isLocal = true;
                    LocalApp.main(new String[]{"zookeeper=xxx",
                                    "topology-type=1","topic=appyuqiang"});
    }
}
```

三. 业务部分
略