Hadoop(2.x)云计算生态系统

wyyhzc

Published
with GitBook

Table of Contents

1. Hadoop2.x云计算体系介绍
2. Hadoop2特点
3. Hadoop2概述
    i. 分布式文件系统HDFS
    ii. YARN（资源管理系统）
    iii. MapReduce(分布式结算框架)
    iv. Hadoop1与Hadoop2生态系统对比
4. Hadoop2安装部署
    i. Hadoop2:QJN的HA模式的分布式部署
    ii. Hadoop2：Federation模式
    iii. 注意事项
5. Kafka分布式消息队列
    i. 架构设计
    ii. 基础知识
    iii. 实施细则
    iv. 分发
    v. 部署模式
    vi. 核心设计
        i. broker核心代码设计
        ii. controller的代码设计
        iii. zk中的存储结构
    vii. 配置
6. Samza分布式流计算
    i. 背景
    ii. 概念
    iii. 架构
    iv. 流处理系统对比
        i. MUPD8
        ii. Storm
        iii. Spark Streaming
    v. API
    vi. 容器 Container
        i. SamzaContainer
        ii. Streams
        iii. Serialization
        iv. Checkpointing
        v. State Management
        vi. Metrics
        vii. Windowing
        viii. Event Loop
        ix. JMX
    vii. Jobs
        i. JobRunner
        ii. Configuration
        iii. Packaging
        iv. YARN Jobs
        v. Logging
        vi. Reprocessing
    viii. YARN
        i. Application Master
        ii. Isolation
    ix. 案例实践

7. Storm分布式流计算
8. Spark分布式内存计算
9. Glossary

Hadoop2.x生态系统

作者：r6

weibo：@r66r

技术QQ群：192234224

相关视频：http://edu.51cto.com/lecturer/user_id-6207546.html

简介

Hadoop是一个由Apache基金会所开发的分布式系统基础架构。现在Hadoop2已经发展成熟。Hadoop使用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。

Hadoop实现了一个分布式文件系统（Hadoop Distributed File System），简称HDFS。HDFS有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。

Hadoop2的框架最核心的设计就是：HDFS和YARN。HDFS为海量的数据提供了存储，则YARN为集群提供资源管理的功能。

发展现状

自成为大数据分析工具以来，Hadoop 就是一个非常棒的数据存储与计算的分布式系统，但是需要开发 Java 应用来访问数据的 MapReduce 学习和使用起来却比较困难。

当然，还有别的办法可以从 Hadoop 中获取信息。Hbase数据是 Hadoop 的一部分，它可以让用户按照数据库范式来处理数据。Hive数据仓库则可以让你用类 SQL 的 HiveSQL 查询语言来创建查询并转化为 MapReduce 任务。但是总体的资源利用率不高。

Hadoop 的开发社区也意识到这个问题，随着 Hadoop 即将迭代到新的版本全新的Hadoop2，上述限制即将在很大程度上被优化了。

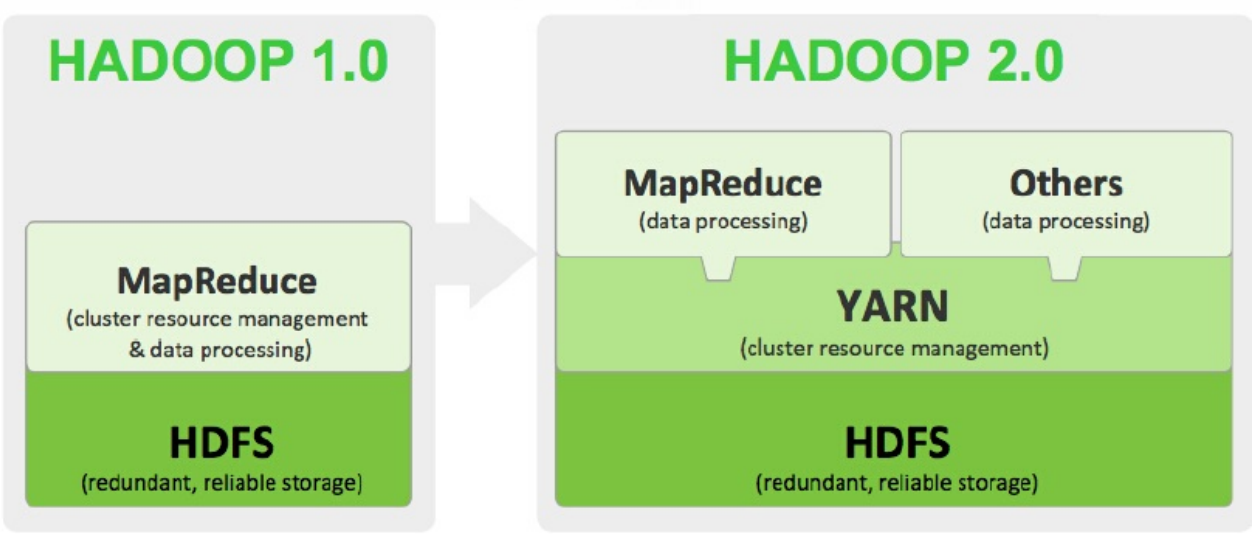
在 Hadoop 2.0 发布经理 Arun Murthy 看来，其最重要的变化是 MapReduce 框架升级为Apache YARN，这将扩展 Hadoop 中可以应用的软件种类和应用程度。Arun Murthy 本人就是 YARN 项目主管，他指出，Hadoop 1.0 和 2.0 的区别在于，前者所有的事情都是面向批处理的，而后者则允许多个应用同时在内部访问数据。

现在越来越多的分布式应用基于Hadoop2，例如：分布式内存计算的流处理系统Spark、Storm、Samza等越来越多的基于Yarn。我们相信，Hadoop2一定会在未来在大数据应用领域发挥更大的作用。

特点

1. 源代码开源（免费）
2. 社区活跃、参与者众多
3. 涉及分布式存储和计算的方方面面
4. 已得到企业界验证

Hadoop1与Hadoop2的对比



- Hadoop1中NameNode存在单点故障，Hadoop2中引入了2中模式的HA解决方案，很好的解决了NameNode与Jobtracke的单点故障问题，为生产系统提供高可用性。
- Hadoop2引入NameNode的Federation功能，解决NameNode的性能瓶颈问题，提高NameNode的可扩展性。

Hadoop 1.0中的资源管理存在以下几个缺点：

（1）静态资源配置。采用了静态资源设置策略，即每个节点实现配置好可用的slot总数，这些slot数目一旦启动后无法再动态修改。

（2）资源无法共享。Hadoop 1.0将slot分为Map slot和Reduce slot两种，且不允许共享。对于一个作业，刚开始运行时，Map slot资源紧缺而Reduce slot空闲，当Map Task全部运行完成后，Reduce slot紧缺而Map slot空闲。很明显，这种区分slot类别的资源管理方案在一定程度上降低了slot的利用率。

（3） 资源划分粒度过大。这种基于无类别slot的资源划分方法的划分粒度仍过于粗糙，往往会造成节点资源利用率过高或者过低，比如，管理员事先规划好一个slot代表2GB内存和1个CPU，如果一个应用程序的任务只需要1GB内存，则会产生“资源碎片”，从而降低集群资源的利用率，同样，如果一个应用程序的任务需要3GB内存，则会隐式地抢占其他任务的资源，从而产生资源抢占现象，可能导致集群利用率过高。

（4） 没引入有效的资源隔离机制。Hadoop 1.0仅采用了基于jvm的资源隔离机制，这种方式仍过于粗糙，很多资源，比如CPU，无法进行隔离，这会造成同一个节点上的任务之间干扰严重。

Hadoop 2.x中的资源管理方案

Hadoop 2.0指的是版本为Apache Hadoop 0.23.x、2.x或者CDH4系列的Hadoop，内核主要由HDFS、MapReduce和YARN三个系统组成，其中，YARN是一个资源管理系统，负责集群资源管理和调度，MapReduce则是运行在YARN上离线处理框

架，它与Hadoop 1.0中的MapReduce在编程模型（新旧API）和数据处理引擎（MapTask和ReduceTask）两个方面是相同的。

让我们回归到资源分配的本质，即根据任务资源需求为其分配系统中的各类资源。在实际系统中，资源本身是多维度的，包括CPU、内存、网络I/O和磁盘I/O等，因此，如果想精确控制资源分配，不能再有slot的概念，最直接的方法是让任务直接向调度器申请自己需要的资源（比如某个任务可申请1.5GB 内存和1个CPU），而调度器则按照任务实际需求为其精细地分配对应的资源量，不再简单的将一个Slot分配给它，Hadoop 2.0正式采用了这种基于真实资源量的资源分配方案。

Hadoop 2.0（YARN）允许每个节点（NodeManager）配置可用的CPU和内存资源总量，而中央调度器则会根据这些资源总量分配给应用程序。节点（NodeManager）配置参数如下：

（1）yarn.nodemanager.resource.memory-mb

可分配的物理内存总量，默认是8*1024，即8GB。

（2）yarn.nodemanager.vmem-pmem-ratio

任务使用单位物理内存量对应最多可使用的虚拟内存量，默认值是2.1，表示每使用1MB的物理内存，最多可以使用2.1MB的虚拟内存总量。

（3）yarn.nodemanager.resource.cpu-vcore

可分配的虚拟CPU个数，默认是8

Hadoop2概述

分布式文件系统HDFS

Yarn资源管理系统

分布式存储系统HDFS

- 分布式存储系统
- 提供了高可靠性、高扩展性和高吞吐率的数据存储服务

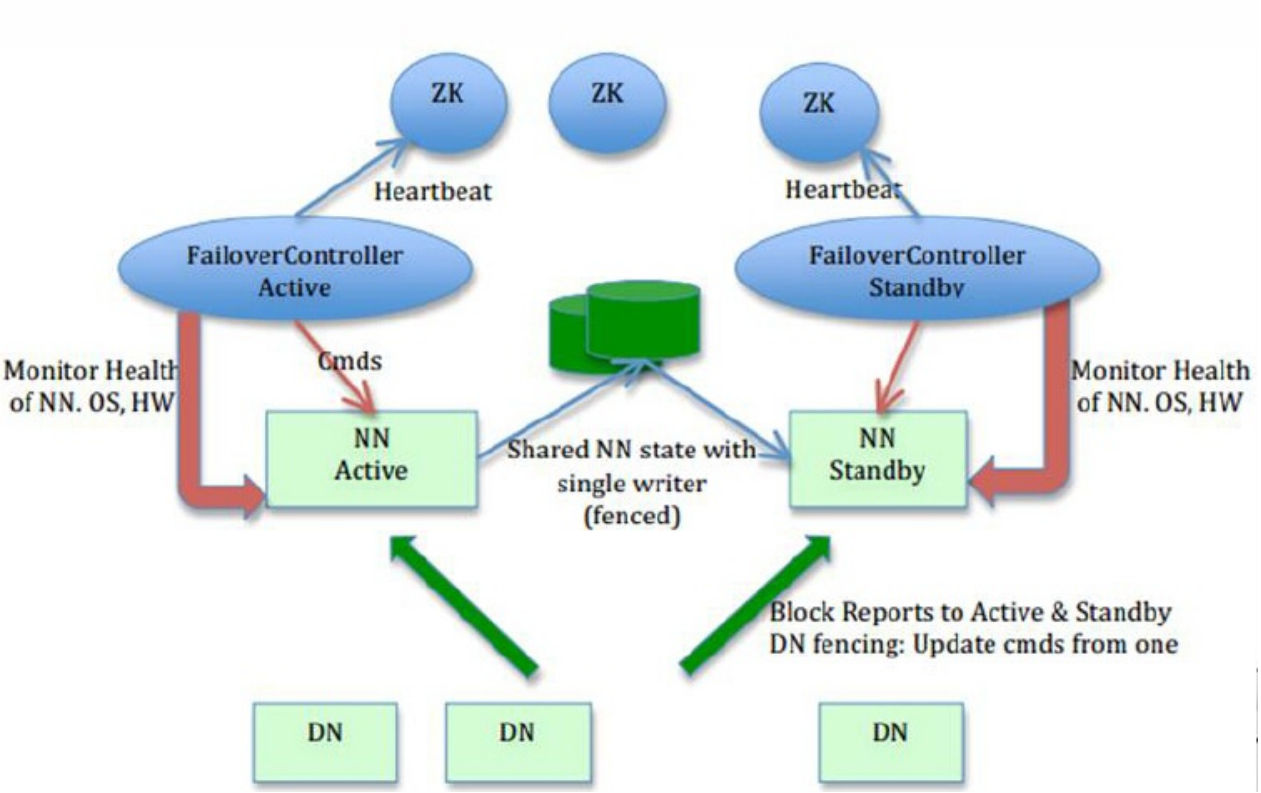
基本原理

> 将文件切分成等大的数据块，存储到多台机器上
>
> 将数据切分、容错、负载均衡等功能透明化
>
> 可将HDFS看成一个容量巨大、具有高容错性的磁盘

应用场景

> 海量数据的可靠性存储
>
> 数据归档

文件系统架构



> NameNode节点通过NFS与Zookeeper实现高可用

YARN（资源管理系统）

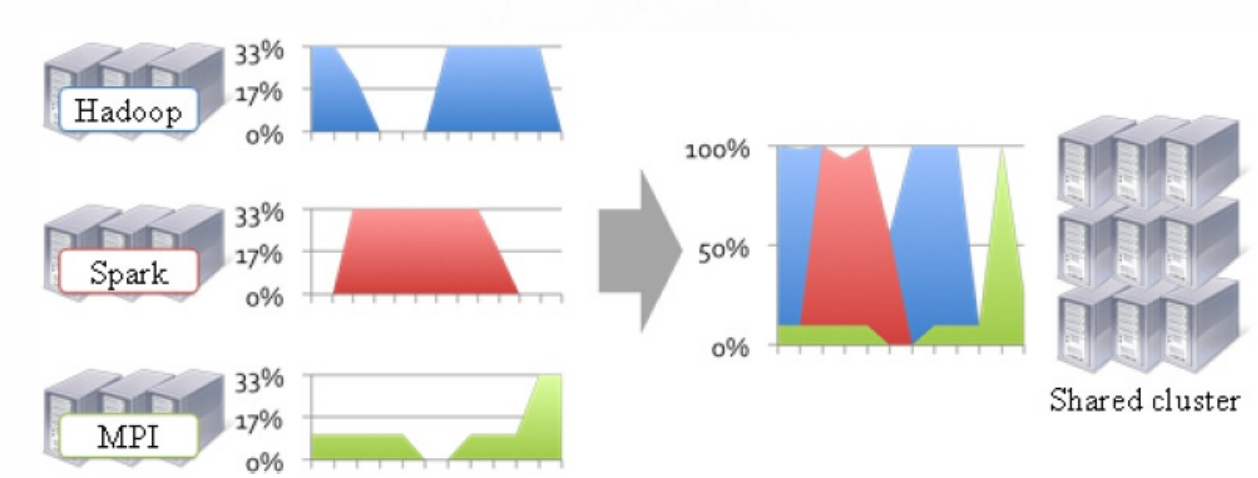
YARN是什么

> Hadoop 2.0新增系统
>
> 负责集群的资源管理和调度
>
> 使得多种计算框架可以运行在一个集群中

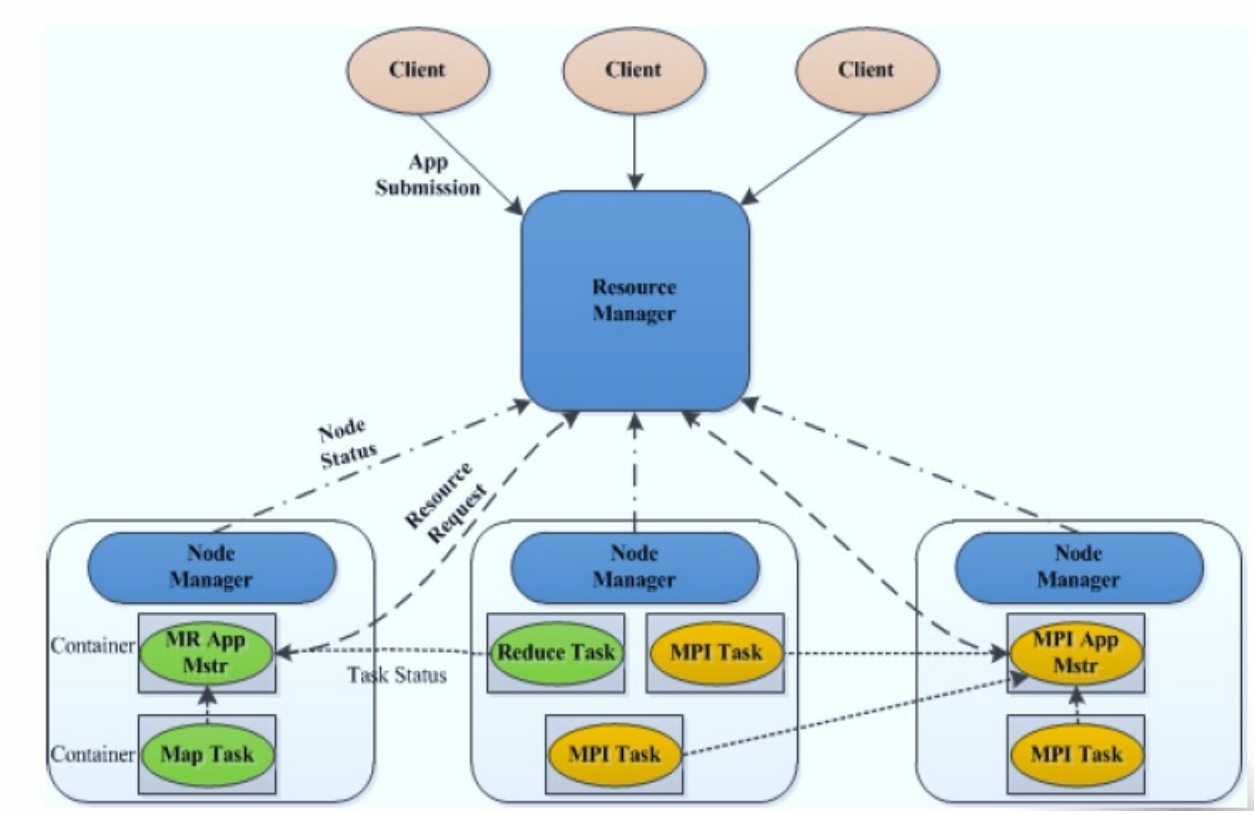
YARN的特点

> 良好的扩展性、高可用性
>
> 对多种类型的应用程序进行统一管理和调度
>
> 自带了多种多用户调度器，适合共享集群环境

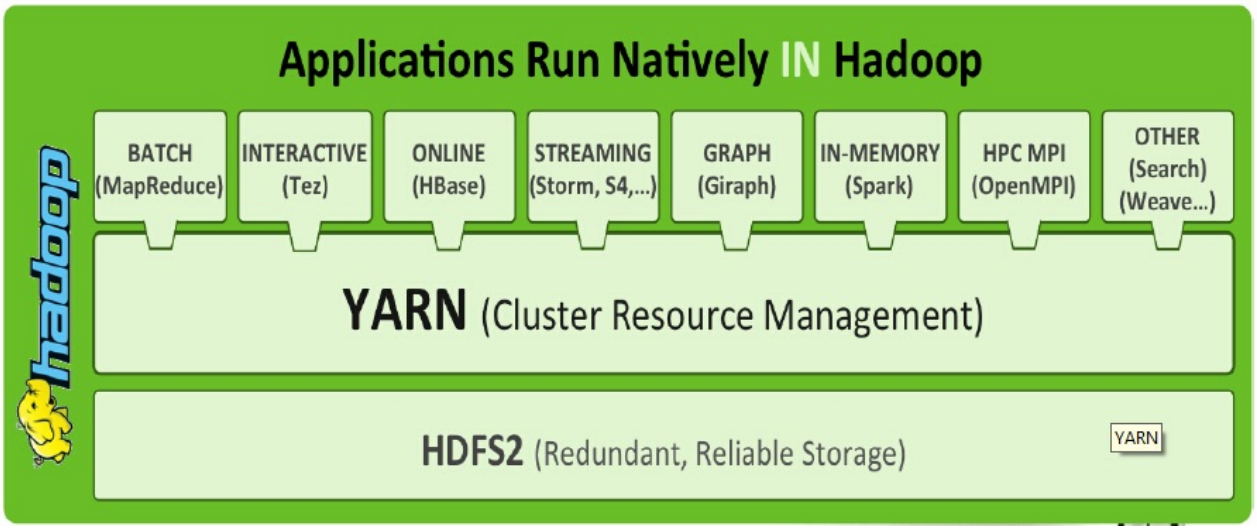
在资源的利用率上面与**hadoop1**的对比



Yarn的系统架构


Client
Client
Client
App Submission
Resource Manager
Node Status
Resource Request
Node Manager
Container
MR App Mstr
Container
Map Task
Task Status
Node Manager
Reduce Task
MPI Task
MPI Task
Node Manager
MPI App Mstr
MPI Task

Yarn的生态环境


Applications Run Natively IN Hadoop
BATCH (MapReduce)
INTERACTIVE (Tez)
ONLINE (HBase)
STREAMING (Storm, S4,...)
GRAPH (Giraph)
IN-MEMORY (Spark)
HPC MPI (OpenMPI)
OTHER (Search) (Weave...)
YARN (Cluster Resource Management)
HDFS2 (Redundant, Reliable Storage)
YARN

MapReduce(分布式结算框架)

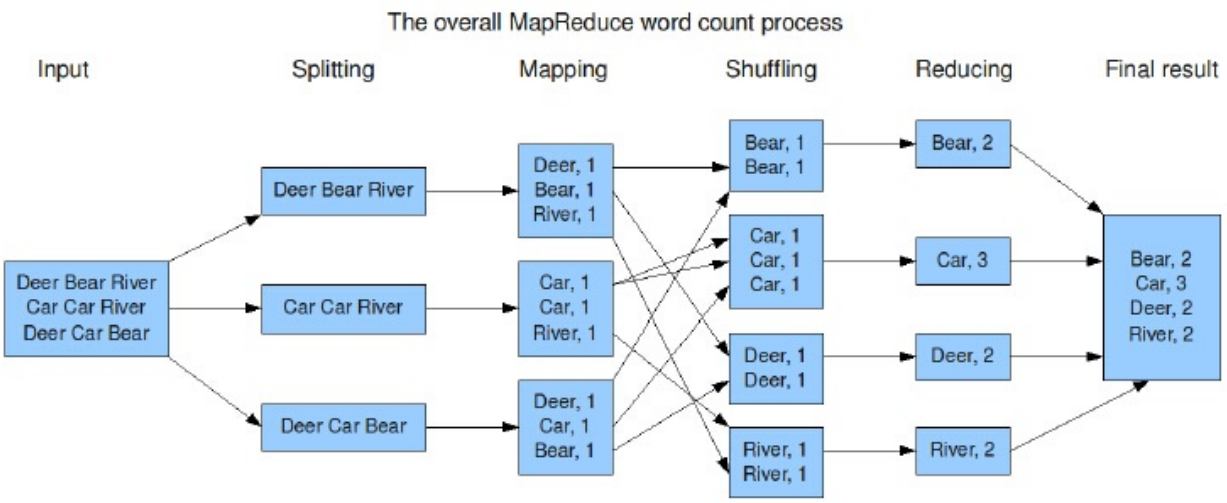
源自于Google的MapReduce论文

> 发表于2004年12月
>
> Hadoop MapReduce是Google MapReduce克隆版

MapReduce特点

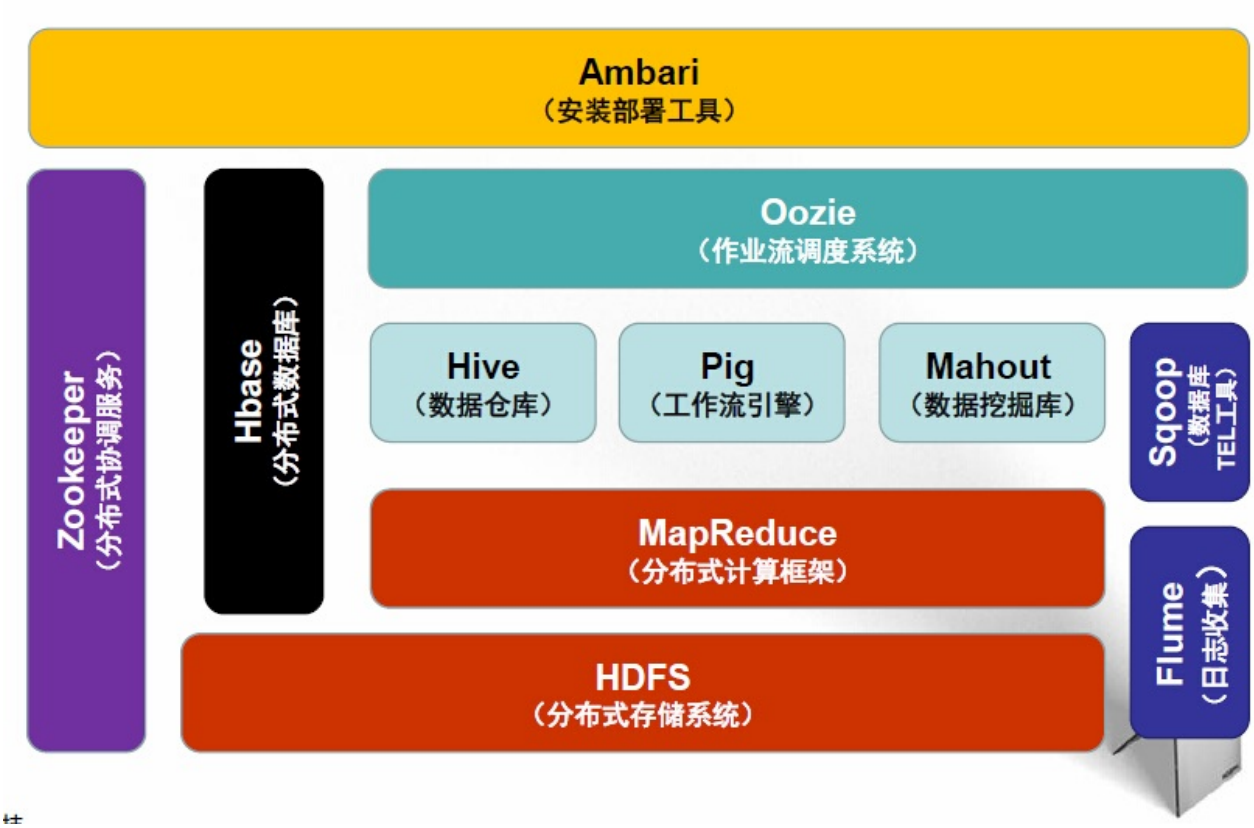
> 良好的扩展性
>
> 高容错性
>
> 适合PB级以上海量数据的离线处理

举例MapReduce流程

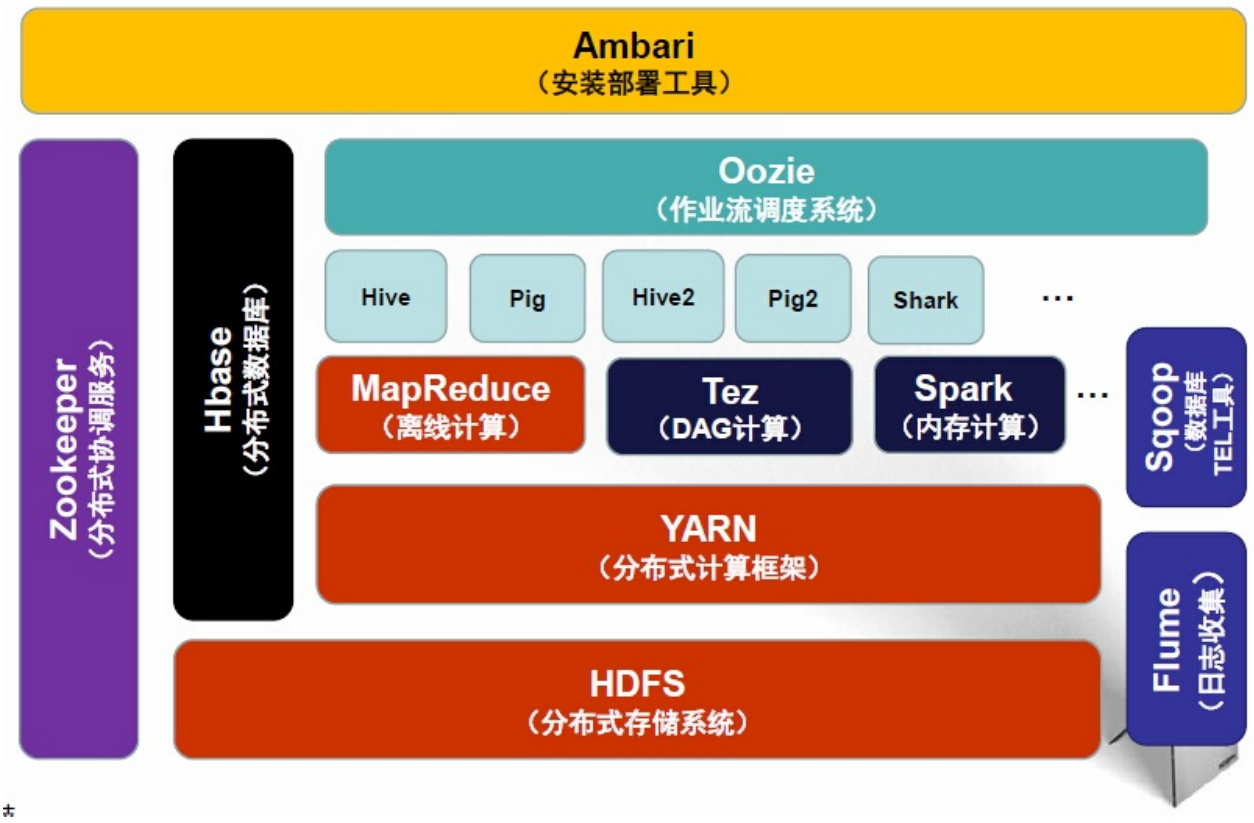


Hadoop1与Hadoop2生态系统对比

Hadoop1.0时代



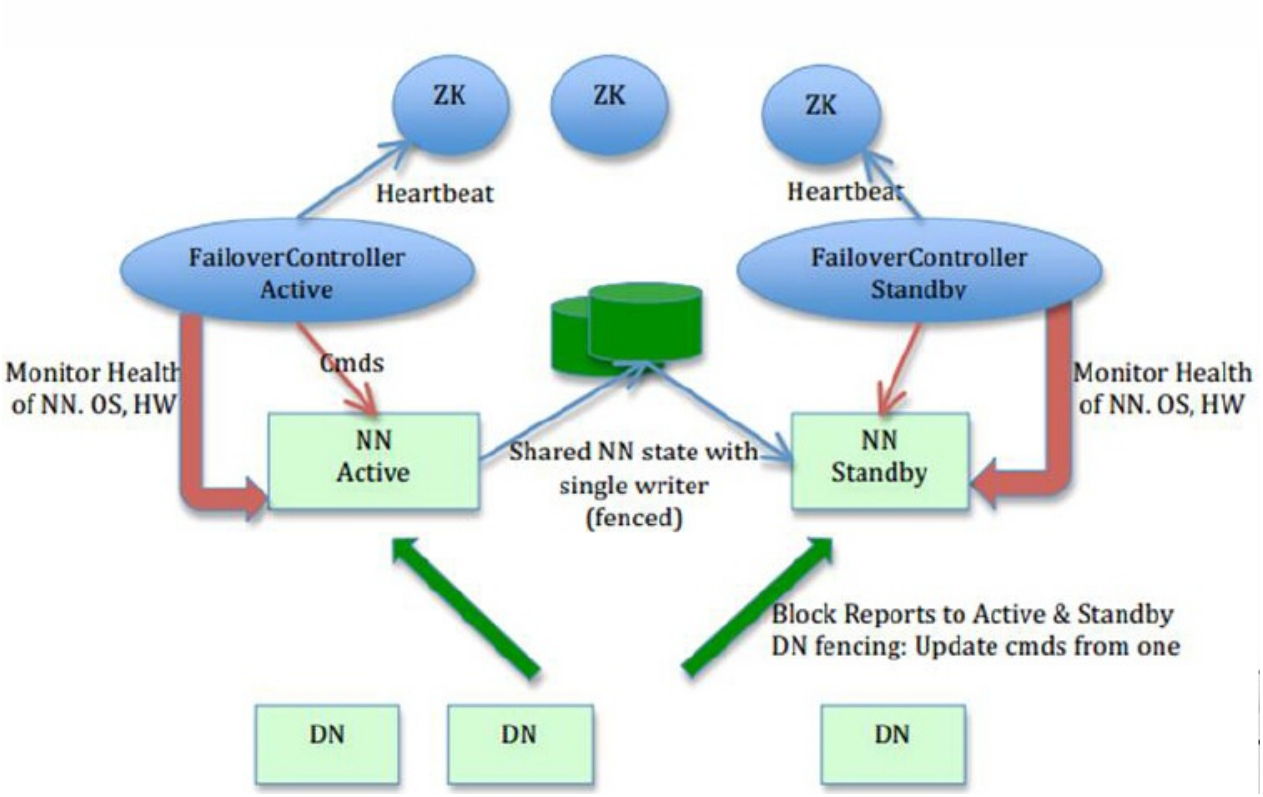
Hadoop2.0时代



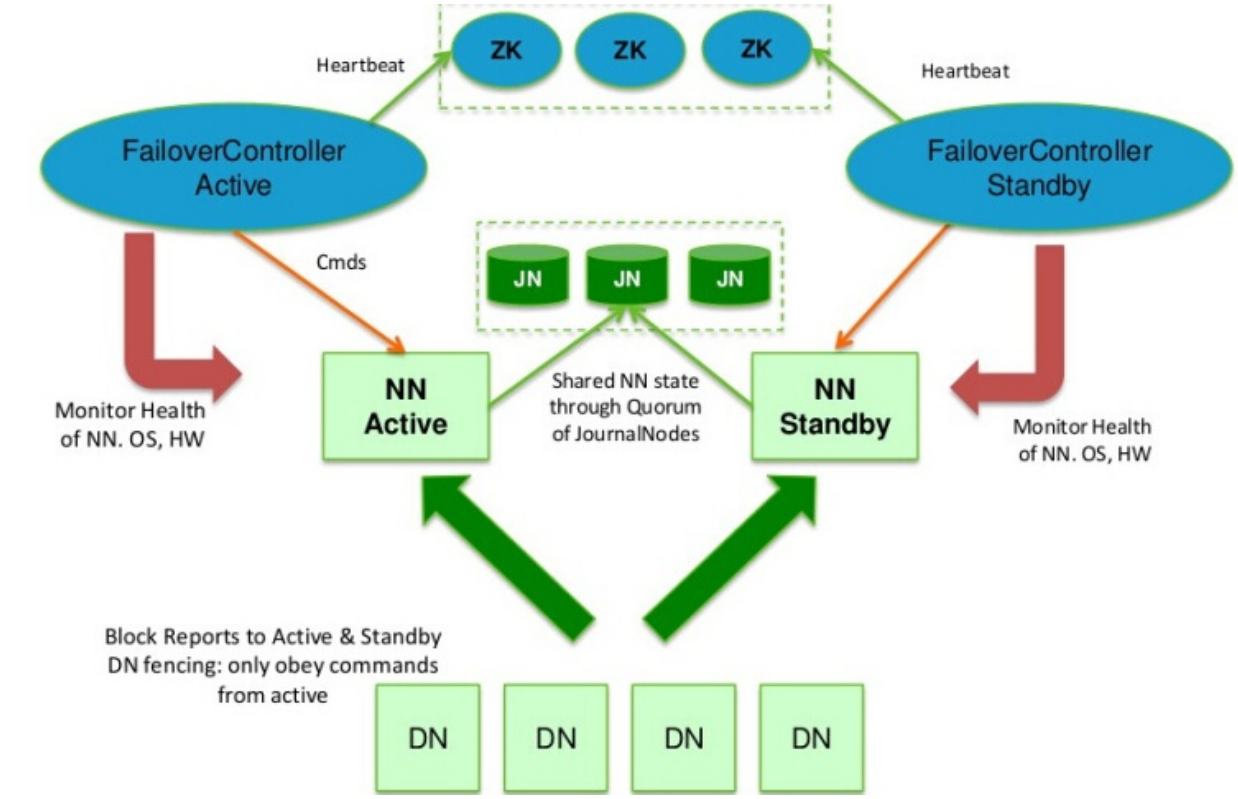
Hadoop2.x的安装与部署

Hadoop2-NameNode的HA模式的2种实现方式

NFS模式



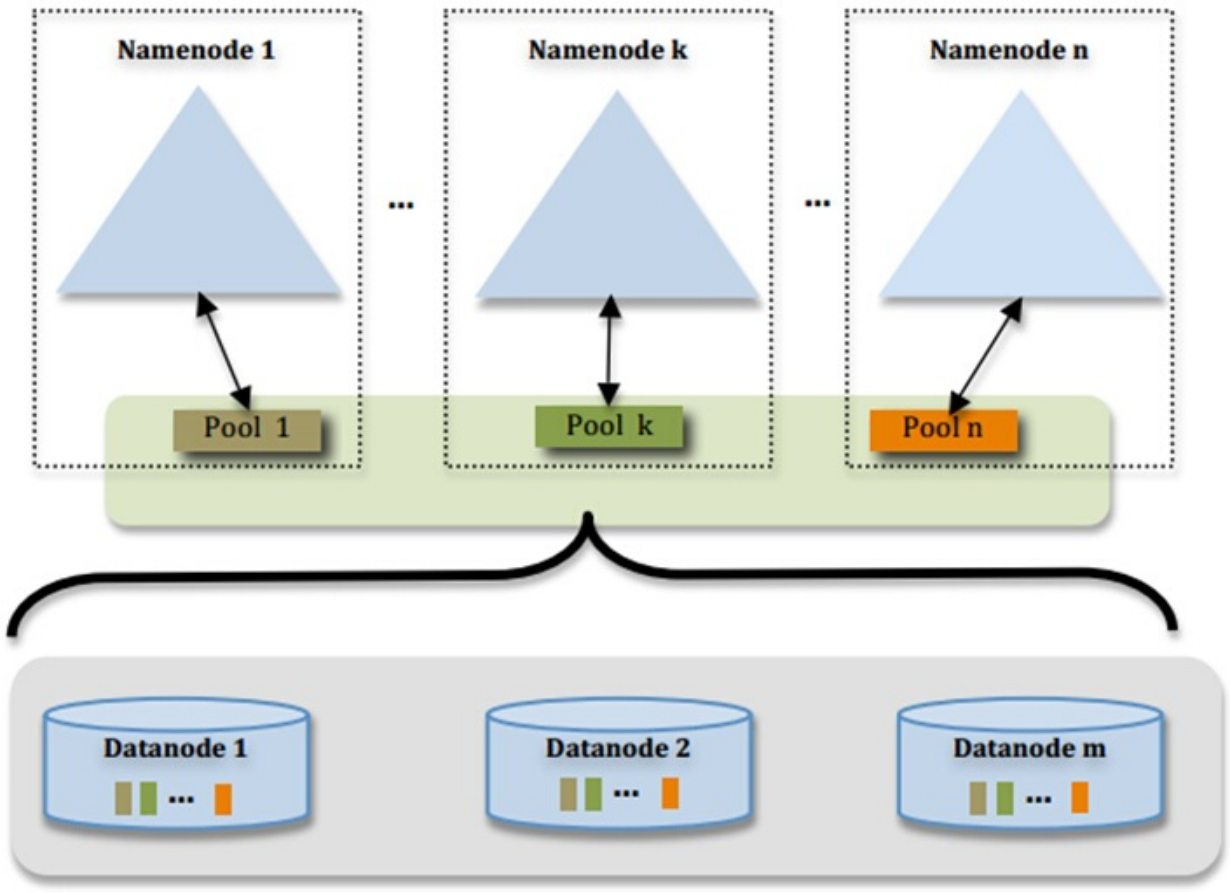
Quorum Journal Node 模式



我们可以看出**HA**的大致架构，其设计上的考虑包括：

- 利用共享存储来在两个NN间同步edits信息。 以前的HDFS是share nothing but NN，现在NN又share storage，这样其实是转移了单点故障的位置，但中高端的存储设备内部都有各种RAID以及冗余硬件包括电源以及网卡等，比服务器的可靠性还是略有提高。通过NN内部每次元数据变动后的flush操作，加上NFS的close-to-open，数据的一致性得到了保证。社区现在也试图把元数据存储放到BookKeeper上，以去除对共享存储的依赖，Cloudera也提供了Quorum Journal Manager的实现和代码，这篇中文的blog有详尽分析：基于QJM/Qurom Journal Manager/Paxos的HDFS HA原理及代码分析
- DataNode(以下简称DN)同时向两个NN汇报块信息。 这是让Standby NN保持集群最新状态的必需步骤，不赘述。
- 用于监视和控制NN进程的FailoverController进程 显然，我们不能在NN进程内进行心跳等信息同步，最简单的原因，一次FullGC就可以让NN挂起十几分钟，所以，必须要有一个独立的短小精悍的watchdog来专门负责监控。这也是一个松耦合的设计，便于扩展或更改，目前版本里是用ZooKeeper(以下简称ZK)来做同步锁，但用户可以方便的把这个ZooKeeper FailoverController(以下简称ZKFC)替换为其他的HA方案或leader选举方案。
- 隔离(Fencing))，防止脑裂)，就是保证在任何时候只有一个主NN，包括三个方面： 共享存储fencing，确保只有一个NN可以写入edits。 客户端fencing，确保只有一个NN可以响应客户端的请求。 DataNode fencing，确保只有一个NN可以向DN下发命令，譬如删除块，复制块，等等。

Hadoop2-NameNode的Federation模式实现方式



这个图过于简明，许多设计上的考虑并不那么直观，我们稍微总结一下

- 多个NN共用一个集群里DN上的存储资源，每个NN都可以单独对外提供服务
- 每个NN都会定义一个存储池，有单独的id，每个DN都为所有存储池提供存储
- DN会按照存储池id向其对应的NN汇报块信息，同时，DN会向所有NN汇报本地存储可用资源情况
- 如果需要在客户端方便的访问若干个NN上的资源，可以使用客户端挂载表，把不同的目录映射到不同的NN，但NN上必须存在相应的目录

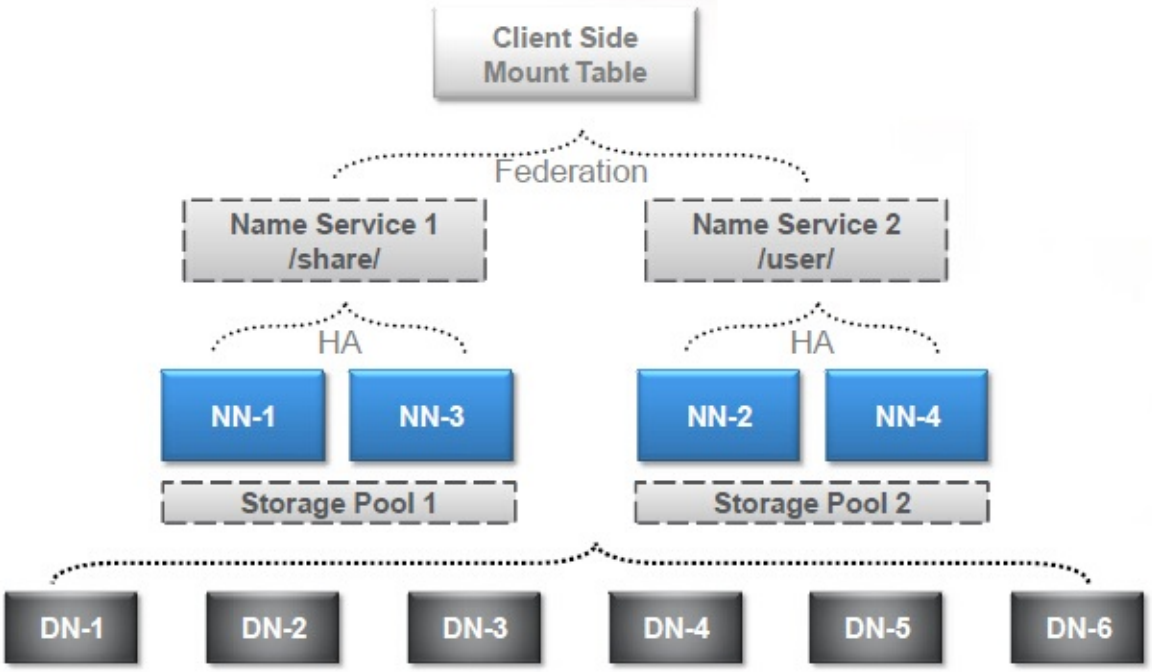
这样设计的好处大致有：

- 改动最小，向前兼容
  1. 现有的NN无需任何配置改动.
  2. 如果现有的客户端只连某台NN的话，代码和配置也无需改动。

- 分离命名空间管理和块存储管理
  1. 提供良好扩展性的同时允许其他文件系统或应用直接使用块存储池
  2. 统一的块存储管理保证了资源利用率
  3. 可以只通过防火墙配置达到一定的文件访问隔离，而无需使用复杂的Kerberos认证

- 客户端挂载表
  1. 通过路径自动对应NN
  2. 使Federation的配置改动对应用透明

复杂完整的Hadoop2的部署模式



Hadoop2经典分布式部署模式

基于QJN的HA模式的分布式部署，不含Federation模块的实践是一个经典的Hadoop2的高可用的分布式部署模式。

1.准备测试环境

准备**4**台**PC**服务器做**Hadoop2**部署

ip	hostname	namenode	fc	datanode	rm	nodemanage	QJN
10.71.84.237	hadoop201	Y	Y	Y	Y	Y	Y
10.71.84.223	hadoop202	Y	Y	Y	Y	Y	Y
10.71.84.222	hadoop203	N	N	Y	N	Y	Y
10.71.84.238	hadoop204	N	N	Y	N	Y	N

准备**3**台**PC**服务器做**zookeeper**部署

ip	hostname
10.71.83.14	hadoop10
10.71.84.16	hadoop12
10.71.84.17	hadoop13

2.安装步骤

一、初始化每台**linux**系统设置

用**root**账户登录

1. 安装centos6.4 x64的操作系统，安装过程略
2. 安装成功后初始化每台机器的系统配置

```
#增加hadoop组
groupadd -g 4000 hadoop
#增加hadoop用户
useradd -g hadoop -c "hadoopuser" -p 111111 -u 3001 -m hadoop -d /home/hadoop
#初始化hadoop用户的密码
passwd hadoop
#创建hadoop集群的数据与计算的应用目录
mkdir /app/hadoop
chown hadoop:hadoop /app/hadoop
#利用root用户安装emacs工具
yum install -y emacs
#修改机器名称,根据不同的机器修改为不同的机器名
hostname hadoop10
```

emacs -nw /etc/sysconfig/network

```
NETWORKING=yes
HOSTNAME=hadoop201
```

emacs -nw /etc/hosts

```
10.71.84.237 hadoop201
10.71.84.223 hadoop202
10.71.84.222 hadoop203
10.71.84.238 hadoop204
10.71.83.14 hadoop10
10.71.83.16 hadoop12
10.71.83.17 hadoop13
```

emacs -nw /etc/security/limits.d/90-nproc.conf 增加下面的内容

```
*          soft    nproc     1024
hadoop     soft    nproc     25535
hadoop     hard    nproc     65535
```

emacs -nw /etc/sysctl.conf 增加下面的内容

```
fs.file-max = 655350
```

设置同步的时间服务器

```
1. >yum install -y ntp

2.emacs -nw /etc/ntp.conf

>注释掉
>#server 0.centos.pool.ntp.org iburst

>#server 1.centos.pool.ntp.org iburst

>#server 2.centos.pool.ntp.org iburst

>#server 3.centos.pool.ntp.org iburst

>增加国家授权时间服务器(北京邮电大学提供)

>server s1a.time.edu.cn

3.chkconfig --level 345 ntpd on

4.ntpdate s1a.time.edu.cn

5.service ntpd start
```

用**hadoop**账户登录,修改环境变量

```
$ emacs -nw /home/hadoop/.bash_profile
```

```
    export JAVA_HOME=/app/hadoop/java/jdk1.6.0_38
    export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
    export HADOOP_HOME=/app/hadoop/hadoop/hadoop-2.5.2
    export PATH=/usr/sbin:$PATH
    export PATH=$HADOOP_HOME/bin:$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH:$HOMR/bin:$PATH
```

用**hadoop**用户上传相关程序文件到 **/app/hadoop**

java的版本为**1.6.0_38 x64**

hadoop2的版本为**2.5.2 x64** (apache提供的版本为32位版本，请下载此64位版本)

配置**ssh**免登陆（**NN**的自动切换）

> 涉及机器： hadoop201 hadoop202
>
> 配置方法：略
>
> 最终效果：hadoop201与hadoop202之间可以相互免登陆

二、每台服务器的**Hadoop2**相关核心配置文件

`core-site.xml` (/app/hadoop/hadoop/hadoop-2.5.2/etc/hadoop/core-site.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://cluster1</value>
</property>

<property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/hadoop/tmp</value>
</property>
<property>
  <name>ha.zookeeper.quorum</name>
  <value>hadoop10:2181,hadoop12:2181,hadoop13:2181</value>
</property>
```

`hdfs-site.xml` (/app/hadoop/hadoop/hadoop-2.5.2/etc/hadoop/hdfs-site.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>

    <property>
        <name>dfs.replication</name>
        <value>3</value>
    </property>

    <property>
        <name>dfs.nameservices</name>
        <value>cluster1</value>
    </property>
```

```
    <property>
        <name>dfs.ha.namenodes.cluster1</name>
        <value>hadoop201,hadoop202</value>
    </property>

    <property>
        <name>dfs.namenode.rpc-address.cluster1.hadoop201</name>
        <value>hadoop201:9000</value>
    </property>

    <property>
        <name>dfs.namenode.http-address.cluster1.hadoop201</name>
        <value>hadoop201:50070</value>
    </property>

    <property>
        <name>dfs.namenode.rpc-address.cluster1.hadoop202</name>
        <value>hadoop202:9000</value>
    </property>

    <property>
        <name>dfs.namenode.http-address.cluster1.hadoop202</name>
        <value>hadoop202:50070</value>
    </property>


    <property>
        <name>dfs.namenode.shared.edits.dir</name>
        <value>qjournal://hadoop201:8485;hadoop202:8485;hadoop203:8485/cluster1</value>
    </property>


    <property>
        <name>dfs.ha.automatic-failover.enabled.cluster1</name>
        <value>true</value>
    </property>

    <property>
        <name>dfs.client.failover.proxy.provider.cluster1</name>
        <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
    </property>

    <property>
        <name>dfs.journalnode.edits.dir</name>
        <value>/app/hadoop/hadoop/tmp/journal</value>
    </property>

    <property>
        <name>dfs.ha.fencing.methods</name>
        <value>sshfence</value>
    </property>

    <property>
        <name>dfs.ha.fencing.ssh.private-key-files</name>
        <value>/home/hadoop/.ssh/id_rsa</value>
    </property>

</configuration>
```

`mapred-site.xml` (/app/hadoop/hadoop/hadoop-2.5.2/etc/hadoop/mapred-site.xml)

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->
```

```
<configuration>
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
</configuration>
```

`yarn-site.xml` (/app/hadoop/hadoop/hadoop-2.5.2/etc/hadoop/yarn-site.xml)

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>

<property>
        <name>yarn.resourcemanager.ha.enabled</name>
        <value>true</value>
    </property>

    <property>
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>hadoop201</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>hadoop202</value>
    </property>

    <property>
        <name>yarn.resourcemanager.recovery.enabled</name>
        <value>true</value>
    </property>

    <property>
        <name>yarn.resourcemanager.store.class</name>
        <value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</value>
    </property>

    <property>
        <name>yarn.resourcemanager.zk-address</name>
        <value>hadoop10:2181,hadoop12:2181,hadoop13:2181</value>
        <description>For multiple zk services, separate them with comma</description>
    </property>

    <property>
          <name>yarn.resourcemanager.cluster-id</name>
          <value>yarn-ha</value>
    </property>

</configuration>
```

三、集群启动并初始化

1.启动每台机器上面的JournalNode节点

> 涉及的机器为：hadoop201 hadoop202 hadoop203
>
> 步骤：hadoop用户登陆每台机器，使用 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start journalnode &

2. 启动NameNode节点，此时NN的状态都是standby状态（由于还没有启动FC）

> 涉及的机器为：hadoop201 hadoop202
>
> 步骤：
>
> 1.通过hadoop账户登陆hadoop201。执行 /app/hadoop/hadoop/hadoop-2.5.2/bin/hdfs zkfc –formatZK
>
> 2.继续在hadoop201这台机器上面，执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start namenode &
>
> 3.通过hadoop账户登陆hadoop202。执行 /app/hadoop/hadoop/hadoop-2.5.2/bin/hdfs namenode -bootstrapStandby
>
> 4.继续在hadoop202上面，执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start namenode &
>
> 最后通过jps命令查看进程状态，检查logs目录下面相关NN的log是否有异常。确保NN进程已经正常启动

3. 启动FC(ZooKeeperFailoverController)进程

> 涉及的机器为：hadoop201 hadoop202
>
> 步骤：
>
> 1.通过hadoop账户登陆hadoop201。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start zkfc &
>
> 2.通过hadoop账户登陆hadoop202。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start zkfc &
>
> 最后通过jps命令查看进程状态，检查logs目录下面相关FC的log是否有异常。确保FC进程已经正常启动。此时，2个NN节点将由一个是active状态，另外一个是standby状态。

4. 启动datanode节点

> 涉及的机器为：hadoop201 hadoop202 hadoop203 hadoop204
>
> 步骤：
>
> 1.通过hadoop账户登陆每一台机器。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/sbin/hadoop-daemon.sh start datanode &
>
> 最后通过jps命令查看进程状态，检查logs目录下面相关DN的log是否有异常。确保DN进程已经正常启动。

5. 启动Yarn的Rm的HA模式

> 涉及的机器为：hadoop201 hadoop202
>
> 步骤：
>
> 1.通过hadoop账户登陆hadoop201。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/bin/yarn resourcemanager &
>
> 2.通过hadoop账户登陆hadoop202。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/bin/yarn resourcemanager &
>
> 最后通过jps命令查看进程状态，检查logs目录下面相关rm的log是否有异常。确保Yarn的rm进程已经正常启动。

6.启动Yarn的NodeManage

> 涉及的机器为：hadoop201 hadoop202 hadoop203 hadoop204
>
> 步骤：
>
> 1.通过hadoop账户登陆每一台机器。执行 nohup /app/hadoop/hadoop/hadoop-2.5.2/bin/yarn nodemanager &
>
> 最后通过jps命令查看进程状态，检查logs目录下面相关nm的log是否有异常。确保Yarn的nm进程已经正常启动。

7.NameNode的HA模式验证

我们设计了两个维度的测试矩阵：系统失效方式，客户端连接模型

系统失效有两种：

> 终止NameNode进程：
>
> ```
> ZKFC主动释放锁
>
> 模拟机器OOM、死锁、硬件性能骤降等故障
> ```
>
> NN机器掉电：
>
> ```
> ZK锁超时
>
> 模拟网络和交换机故障、以及掉电本身
> ```

客户端连接也是两种：

> 已连接的客户端(持续拷贝96M的文件，1M每块)
>
> 通过增加块的数目，我们希望客户端会不断的向NN去申请新的块；一般是在第一个文件快结束或第二个文件刚开始拷贝的时候使系统失效。
>
> 新发起连接的客户端(持续拷贝96M的文件，100M每块) 因为只有一个块，所以在实际拷贝过程中失效并不会立刻导致客户端或DN报错，但下一次新发起连接的客户端会一开始就没有NN可连；一般是在第一个文件快结束拷贝时使系统失效。 针对每一种组合，我们反复测试10－30次，每次拷贝5个文件进入HDFS，因为时间不一定掐的很准，所以有时候也会是在第三或第四个文件的时候才使系统失效，不管如何，我们会在结束后从HDFS里取出所有文件，并挨个检查文件MD5，以确保数据的完整性。

测试结果如下：

```
ZKFC主动释放锁
    5-8秒切换(需同步edits)
    客户端偶尔会有重试(~10%)
    但从未失败
ZK锁超时
    15-20s切换(超时设置为10s)
    客户端重试几率变大(~75%)
    且偶有失败(~15%)，但仅见于已连接客户端
可确保数据完整性
    MD5校验从未出错 +失败时客户端有Exception
```

我们的结论是：`Hadoop 2.0里的HDFS HA基本可满足高可用性`

总结：

Hadoop2提供了NN与Rm的HA模式，为生产系统提供了高可用的保证，而且Yarn对多种分布式计算提供更为高效的资源管理，官方也是建议大家升级到hadoop2。笔者认为hadoop2的最大优势就是yarn的资源管理，先进的管理模式使后续很多的分布式应用程序,例如：spark、storm与samza等都是基于yarn。

访问链接：

NN(hadoop201)节点访问链接 http://10.71.84.237:50070/dfshealth.htm

NN(hadoop202)节点访问链接 http://10.71.84.237:50070/dfshealth.htm

Yarn(hadoop201)RM访问链接 http://10.71.84.237:8088/cluster

Hadoop2：Federation模式

等待添加

Kafka分布式消息队列

概述

Kafka是Linkedin于2010年12月份开源的消息系统，它主要用于处理活跃的流式数据。活跃的流式数据在web网站应用中非常常见，这些数据包括网站的pv、用户访问了什么内容，搜索了什么内容等。 这些数据通常以日志的形式记录下来，然后每隔一段时间进行一次统计处理。

传统的日志分析系统提供了一种离线处理日志信息的可扩展方案，但若要进行实时处理，通常会有较大延迟。而现有的消（队列）系统能够很好的处理实时或者近似实时的应用，但未处理的数据通常不会写到磁盘上，这对于Hadoop之类（一小时或者一天只处理一部分数据）的离线应用而言，可能存在问题。Kafka正是为了解决以上问题而设计的，它能够很好地离线和在线应用。

案例

Kafka还可以作为一个传统的消息代理broker集成事件到Hadoop.

这里是其一些普通用途:

- 网站活动跟踪Website activity tracking:Web应用发送事件如页浏览量或搜索到Kafka,这样这些事件能够被hadoop实时处理和分析。
- 操作衡量Operational metrics: 实现操作监控的警报和报表，Kafka可以偶尔发送它们的消息总数到一个特定的主题，这样有一个服务就可以比较这些总数，如果数据丢失可以报警。
- 记录聚合Log aggregation: Kafka能够用于跨组织从多个服务收集日志，然后以一种标准格式提供给多个消费者，包括Hadoop 和Apache Solr.
- 流处理Stream processing: 类似Spark Streaming能从一个主题读取数据处理然后写入数据到一个新的主题，这样被其他应用再次利用，Kafka的强壮的durability（不丢失 持久性）对已流处理是非常有用的。

主要相关资料：

kafka官网：http://kafka.apache.org/

kafka系列blog：http://www.r66r.net/?s=kafka

kafka系列blog: apache kafka技术分享系列(目录索引)

架构设计

我们为什么要搭建该系统

Kafka是一个消息系统，原本开发自LinkedIn，用作LinkedIn的活动流（activity stream）和运营数据处理管道（pipeline）的基础。现在它已为多家不同类型的公司 作为多种类型的数据管道（data pipeline）和消息系统使用。

活动流数据是所有站点在对其网站使用情况做报表时要用到的数据中最常规的部分。活动数据包括页面访问量（page view）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。运营数据指的是服务器的性能数据（CPU、IO使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

近年来，活动和运营数据处理已经成为了网站软件产品特性中一个至关重要的组成部分，这就需要一套稍微更加复杂的基础设施对其提供支持。

活动流和运营数据的若干用例

- "动态汇总（News feed）"功能。将你朋友的各种活动信息广播给你 相关性以及排序。通过使用计数评级（count rating）、投票（votes）或者点击率（ click-through）判定一组给定的条目中那一项是最相关的.
- 安全：网站需要屏蔽行为不端的网络爬虫（crawler），对API的使用进行速率限制，探测出扩散垃圾信息的企图，并支撑其它的行为探测和预防体系，以切断网站的某些不正常活动。
- 运营监控：大多数网站都需要某种形式的实时且随机应变的方式，对网站运行效率进行监控并在有问题出现的情况下能触发警告。
- 报表和批处理: 将数据装载到数据仓库或者Hadoop系统中进行离线分析，然后针对业务行为做出相应的报表，这种做法很普遍。

活动流数据的特点

这种由不可变（immutable）的活动数据组成的高吞吐量数据流代表了对计算能力的一种真正的挑战，因其数据量很容易就可能会比网站中位于第二位的数据源的数据量大10到100倍。

传统的日志文件统计分析对报表和批处理这种离线处理的情况来说，是一种很不错且很有伸缩性的方法；但是这种方法对于实时处理来说其时延太大，而且还具有较高的运营复杂度。另一方面，现有的消息队列系统（messaging and queuing system）却很适合于在实时或近实时（near-real-time）的情况下使用，但它们对很长的未被处理的消息队列的处理很不给力，往往并不将数据持久化作为首要的事情考虑。这样就会造成一种情况，就是当把大量数据传送给Hadoop这样的离线系统后， 这些离线系统每个小时或每天仅能处理掉部分源数据。Kafka的目的就是要成为一个队列平台，仅仅使用它就能够既支持离线又支持在线使用这两种情况。

Kafka支持非常通用的消息语义（messaging semantics）。尽管我们这篇文章主要是想把它用于活动处理，但并没有任何限制性条件使得它仅仅适用于此目的。

部署

下面的示意图所示是在LinkedIn中部署后各系统形成的拓扑结构。

要注意的是，一个单个的Kafka集群系统用于处理来自各种不同来源的所有活动数据。它同时为在线和离线的数据使用者提供了一个单个的数据管道，在线活动和异步处理之间形成了一个缓冲区层。我们还使用kafka，把所有数据复制（replicate）到另外一个不同的数据中心去做离线处理。

我们并不想让一个单个的Kafka集群系统跨越多个数据中心，而是想让Kafka支持多数据中心的数据流拓扑结构。这是通过在集群之间进行镜像或“同步”实现的。这个功能非常简单，镜像集群只是作为源集群的数据使用者的角色运行。这意味着，一个单个的集群就能够将来自多个数据中心的数据集中到一个位置。下面所示是可用于支持批量装载（batch loads）的多数据中心拓扑结构的一个例子：

请注意，在图中上面部分的两个集群之间不存在通信连接，两者可能大小不同，具有不同数量的节点。下面部分中的这个单

个的集群可以镜像任意数量的源集群。要了解镜像功能使用方面的更多细节，请访问这里.

主要的设计元素

Kafka之所以和其它绝大多数信息系统不同，是因为下面这几个为数不多的比较重要的设计决策：

1. Kafka在设计之时为就将持久化消息作为通常的使用情况进行了考虑。
2. 主要的设计约束是吞吐量而不是功能。
3. 有关哪些数据已经被使用了的状态信息保存为数据使用者（consumer）的一部分，而不是保存在服务器之上。
4. Kafka是一种显式的分布式系统。它假设，数据生产者（producer）、代理（brokers）和数据使用者（consumer）分散于多台机器之上。

以上这些设计决策将在下文中进行逐条详述。

基础知识

首先来看一些基本的术语和概念。

消息指的是通信的基本单位。由消息生产者（producer）发布关于某话题（topic）的消息，这句话的意思是，消息以一种物理方式被发送给了作为代理（broker）的服务器（可能是另外一台机器）。若干的消息使用者（consumer）订阅（subscribe）某个话题，然后生产者所发布的每条消息都会被发送给所有的使用者。

Kafka是一个显式的分布式系统 —— 生产者、使用者和代理都可以运行在作为一个逻辑单位的、进行相互协作的集群中不同的机器上。对于代理和生产者，这么做非常自然，但使用者却需要一些特殊的支持。每个使用者进程都属于一个使用者小组（consumer group）。准确地讲，每条消息都只会发送给每个使用者小组中的一个进程。因此，使用者小组使得许多进程或多台机器在逻辑上作为一个单个的使用者出现。使用者小组这个概念非常强大，可以用来支持JMS中队列（queue）或者话题（topic）这两种语义。为了支持队列语义，我们可以将所有的使用者组成一个单个的使用者小组，在这种情况下，每条消息都会发送给一个单个的使用者。为了支持话题语义，可以将每个使用者分到它自己的使用者小组中，随后所有的使用者将接收到每一条消息。在我们的使用当中，一种更常见的情况是，我们按照逻辑划分出多个使用者小组，每个小组都是有作为一个逻辑整体的多台使用者计算机组成的集群。在大数据的情况下，Kafka有个额外的优点，对于一个话题而言，无论有多少使用者订阅了它，一条条消息都只会存储一次。

消息持久化（**Message Persistence**）及其缓存

> 不要害怕文件系统！

在对消息进行存储和缓存时，Kafka严重地依赖于文件系统。 大家普遍认为“磁盘很慢”，因而人们都对持久化结（persistent structure）构能够提供说得过去的性能抱有怀疑态度。实际上，同人们的期望值相比，磁盘可以说是既很慢又很快，这取决于磁盘的使用方式。设计的很好的磁盘结构往往可以和网络一样快。

磁盘性能方面最关键的一个事实是，在过去的十几年中，硬盘的吞吐量正在变得和磁盘寻道时间严重不一致了。结果，在一个由6个7200rpm的SATA硬盘组成的RAID-5磁盘阵列上，线性写入（linear write）的速度大约是300MB/秒，但随即写入却只有50k/秒，其中的差别接近10000倍。线性读取和写入是所有使用模式中最具可预计性的一种方式，因而操作系统采用预读（read-ahead）和后写（write-behind）技术对磁盘读写进行探测并优化后效果也不错。预读就是提前将一个比较大的磁盘块中内容读入内存，后写是将一些较小的逻辑写入操作合并起来组成比较大的物理写入操作。关于这个问题更深入的讨论请参考这篇文章ACM Queue article；实际上他们发现，在某些情况下，顺序磁盘访问能够比随即内存访问还要快！

为了抵消这种性能上的波动，现代操作系变得越来越积极地将主内存用作磁盘缓存。所有现代的操作系统都会乐于将所有空闲内存转做磁盘缓存，即时在需要回收这些内存的情况下会付出一些性能方面的代价。所有的磁盘读写操作都需要经过这个统一的缓存。想要舍弃这个特性都不太容易，除非使用直接I/O。因此，对于一个进程而言，即使它在进程内的缓存中保存了一份数据，这份数据也可能在OS的页面缓存（pagecache）中有重复的一份，结构就成了一份数据保存了两次。

更进一步讲，我们是在JVM的基础之上开发的系统，只要是了解过一些Java中内存使用方法的人都知道这两点：

- Java对象的内存开销（overhead）非常大，往往是对象中存储的数据所占内存的两倍（或更糟）。
- Java中的内存垃圾回收会随着堆内数据不断增长而变得越来越不明确，回收所花费的代价也会越来越大。

由于这些因素，使用文件系统并依赖于页面缓存要优于自己在内存中维护一个缓存或者什么别的结构 —— 通过对所有空闲内存自动拥有访问权，我们至少将可用的缓存大小翻了一倍，然后通过保存压缩后的字节结构而非单个对象，缓存可用大小接着可能又翻了一倍。这么做下来，在GC性能不受损失的情况下，我们可在一台拥有32G内存的机器上获得高达28到30G的缓存。而且，这种缓存即使在服务重启之后会仍然保持有效，而不象进程内缓存，进程重启后还需要在内存中进行缓存重建（10G的缓存重建时间可能需要10分钟），否则就需要以一个全空的缓存开始运行（这么做它的初始性能会非常糟糕）。这还大大简化了代码，因为对缓存和文件系统之间的一致性进行维护的所有逻辑现在都是在OS中实现的，这事OS做起来要比我们在进程中做那种一次性的缓存更加高效，准确性也更高。如果你使用磁盘的方式更倾向于线性读取操作，那么随着每次磁盘读取操作，预读就能非常高效使用随后准能用得着的数据填充缓存。

这就让人联想到一个非常简单的设计方案：不是要在内存中保存尽可能多的数据并在需要时将这些数据刷新（flush）到文件系统，而是我们要做完全相反的事情。所有数据都要立即写入文件系统中持久化的日志中但不进行刷新数据的任何调用。实际中这么做意味着，数据被传输到OS内核的页面缓存中了，OS随后会将这些数据刷新到磁盘的。此外我们添加了一条基于

配置的刷新策略，允许用户对把数据刷新到物理磁盘的频率进行控制（每当接收到N条消息或者每过M秒），从而可以为系统硬件崩溃时“处于危险之中”的数据在量上加个上限。

这种以页面缓存为中心的设计风格在一篇讲解Varnish的设计思想的文章中有详细的描述（文风略带有助于身心健康的傲气）。

常量时长足矣

消息系统元数据的持久化数据结构往往采用BTree。 BTree是目前最通用的数据结构，在消息系统中它可以用来广泛支持多种不同的事务性或非事务性语义。 它的确也带来了一个非常高的处理开销，Btree运算的时间复杂度为O(log N)。一般O(log N)被认为基本上等于常量时长，但对于磁盘操作来讲，情况就不同了。磁盘寻道时间一次要花10ms的时间，而且每个磁盘同时只能进行一个寻道操作，因而其并行程度很有限。因此，即使少量的磁盘寻道操作也会造成非常大的时间开销。因为存储系统混合了高速缓存操作和真正的物理磁盘操作，所以树型结构（tree structure）可观察到的性能往往是超线性的（superlinear）。更进一步讲，BTrees需要一种非常复杂的页面级或行级锁定机制才能避免在每次操作时锁定一整颗树。实现这种机制就要为行级锁定付出非常高昂的代价，否则就必须对所有的读取操作进行串行化（serialize）。因为对磁盘寻道操作的高度依赖，就不太可能高效地从驱动器密度（drive density）的提高中获得改善，因而就不得不使用容量较小(< 100GB)转速较高的SAS驱动去，以维持一种比较合理的数据与寻道容量之比。

直觉上讲，持久化队列可以按照通常的日志解决方案的样子构建，只是简单的文件读取和简单地向文件中添加内容。虽然这种结果必然无法支持BTree实现中的丰富语义，但有个优势之处在于其所有的操作的复杂度都是O(1)，读取操作并不需要阻止写入操作，而且反之亦然。这样做显然有性能优势，因为性能完全同数据大小之间脱离了关系 —— 一个服务器现在就能利用大量的廉价、低转速、容量超过1TB的SATA驱动器。虽然这些驱动器寻道操作的性能很低，但这些驱动器在大量数据读写的情况下性能还凑和，而只需1/3的价格就能获得3倍的容量。 能够存取到几乎无限大的磁盘空间而无须付出性能代价意味着，我们可以提供一些消息系统中并不常见的功能。例如，在Kafka中，消息在使用完后并没有立即删除，而是会将这些消息保存相当长的一段时间（比方说一周）。

效率最大化

我们的假设是，系统里消息的量非常之大，实际消息量是网站页面浏览总数的数倍之多（因为每个页面浏览就是我们要处理的其中一个活动）。而且我们假设发布的每条消息都会被至少读取一次（往往是多次），因而我们要为消息使用而不是消息的产生进行系统优化，

导致低效率的原因常见的有两个：过多的网络请求和大量的字节拷贝操作。

为了提高效率，API是围绕这“消息集”（message set）抽象机制进行设计的，消息集将消息进行自然分组。这么做能让网络请求把消息合成一个小组，分摊网络往返（roundtrip）所带来的开销，而不是每次仅仅发送一个单个消息。

MessageSet实现（implementation）本身是对字节数组或文件进行一次包装后形成的一薄层API。因而，里面并不存在消息处理所需的单独的序列化（serialization）或逆序列化（deserialization）的步骤。消息中的字段（field）是按需进行逆序列化的（或者说，在不需要时就不进行逆序列化）。

由代理维护的消息日志本身不过是那些已写入磁盘的消息集的目录。按此进行抽象处理后，就可以让代理和消息使用者共用一个单个字节的格式（从某种程度上说，消息生产者也可以用它，消息生产者的消息要求其校验和（checksum）并在验证后才会添加到日志中）

使用共通的格式后就能对最重要的操作进行优化了：持久化后日志块（chuck）的网络传输。为了将数据从页面缓存直接传送给socket，现代的Unix操作系统提供了一个高度优化的代码路径（code path）。在Linux中这是通过sendfile这个系统调用实现的。通过Java中的API，FileChannel.transferTo，由它来简洁的调用上述的系统调用。

为了理解sendfile所带来的效果，重要的是要理解将数据从文件传输到socket的数据路径：

- 操作系统将数据从磁盘中读取到内核空间里的页面缓存
- 应用程序将数据从内核空间读入到用户空间的缓冲区
- 应用程序将读到的数据写回内核空间并放入socke的缓冲区
- 操作系统将数据从socket的缓冲区拷贝到NIC（网络借口卡，即网卡）的缓冲区，自此数据才能通过网络发送出去

这样效率显然很低，因为里面涉及4次拷贝，2次系统调用。使用sendfile就可以避免这些重复的拷贝操作，让OS直接将数据从页面缓存发送到网络中，其中只需最后一步中的将数据拷贝到NIC的缓冲区。

我们预期的一种常见的用例是一个话题拥有多个消息使用者。采用前文所述的零拷贝优化方案，数据只需拷贝到页面缓存中一次，然后每次发送给使用者时都对它进行重复使用即可，而无须先保存到内存中，然后在阅读该消息时每次都需要将其拷贝到内核空间中。如此一来，消息使用的速度就能接近网络连接的极限。

要得到Java中对send'file和零拷贝的支持方面的更多背景知识，请参考IBM developerworks上的这篇文章。

端到端的批量压缩

多数情况下系统的瓶颈是网络而不是CPU。 这一点对于需要将消息在个数据中心间进行传输的数据管道来说，尤其如此。当然，无需来自Kafka的支持，用户总是可以自行将消息压缩后进行传输，但这么做的压缩率会非常低，因为不同的消息里都有很多重复性的内容（比如JSON里的字段名、web日志中的用户代理或者常用的字符串）。高效压缩需要将多条消息一起进行压缩而不是分别压缩每条消息。理想情况下，以端到端的方式这么做是行得通的 —— 也即，数据在消息生产者发送之前先压缩一下，然后在服务器上一直保存压缩状态，只有到最终的消息使用者那里才需要将其解压缩。

通过运行递归消息集，Kafka对这种压缩方式提供了支持。 一批消息可以打包到一起进行压缩，然后以这种形式发送给服务器。这批消息都会被发送给同一个消息使用者，并会在到达使用者那里之前一直保持为被压缩的形式。

Kafka支持GZIP和Snappy压缩协议。关于压缩的更多更详细的信息，请参见这里。

客户状态

追踪（客户）消费了什么是一个消息系统必须提供的一个关键功能之一。它并不直观，但是记录这个状态是该系统的关键性能之一。状态追踪要求（不断）更新一个有持久性的实体的和一些潜在会发生的随机访问。因此它更可能受到存储系统的查询时间的制约而不是带宽（正如上面所描述的）。

大部分消息系统保留着关于代理者使用(消费)的消息的元数据。也就是说，当消息被交到客户手上时，代理者自己记录了整个过程。这是一个相当直观的选择,而且确实对于一个单机服务器来说，它(数据)能去(放在)哪里是不清晰的。又由于许多消息系统存储使用的数据结构规模小，所以这也是个实用的选择--因为代理者知道什么被消费了使得它可以立刻删除它(数据)，保持数据大小不过大。

也许不显然的是，让代理和使用者这两者对消息的使用情况做到一致表述绝不是一件轻而易举的事情。如果代理每次都是在将消息发送到网络中后就将该消息记录为已使用的话，一旦使用者没能真正处理到该消息（比方说，因为它宕机或这请求超时了抑或别的什么原因），就会出现消息丢失的情况。为了解决此问题，许多消息系新加了一个确认功能，当消息发出后仅把它标示为已发送而不是已使用，然后代理需要等到来自使用者的特定的确认信息后才将消息记录为已使用。这种策略的确解决了丢失消息的问题，但由此产生了新问题。首先，如果使用者已经处理了该消息但却未能发送出确认信息，那么就会让这一条消息被处理两次。第二个问题是关于性能的，这种策略中的代理必须为每条单个的消息维护多个状态（首先为了防止重复发送就要将消息锁定，然后，然后还要将消息标示为已使用后才能删除该消息）。另外还有一些棘手的问题需要处理，比如，对于那些以发出却未得到确认的消息该如何处理？

消息传递语义（**Message delivery semantics**）

系统可以提供的几种可能的消息传递保障如下所示:

- 最多一次—这种用于处理前段文字所述的第一种情况。消息在发出后立即标示为已使用，因此消息不会被发出去两次，但这在许多故障中都会导致消息丢失。
- 至少一次—这种用于处理前文所述的第二种情况，系统保证每条消息至少会发送一次，但在有故障的情况下可能会导致重复发送。
- 仅仅一次—这种是人们实际想要的，每条消息只会而且仅会发送一次。

这个问题已得到广泛的研究，属于“事务提交”问题的一个变种。提供仅仅一次语义的算法已经有了，两阶段或者三阶段提交法以及Paxos算法的一些变种就是其中的一些例子，但它们都有与生俱来的的缺陷。这些算法往往需要多个网络往返（round trip），可能也无法很好的保证其活性（liveness）（它们可能会导致无限期停机）。FLP结果给出了这些算法的一些基本的局限。 Kafka对元数据做了两件很不寻常的事情。一件是，代理将数据流划分为一组互相独立的分区。这些分区的语义由生产者定义，由生产者来指定每条消息属于哪个分区。一个分区内的消息以到达代理的时间为准进行排序，将来按此顺序将消息发送给使用者。这么一来，就用不着为每一天消息保存一条元数据（比如说，将消息标示为已使用）了，我们只需为使用者、话题和分区的每种组合记录一个“最高水位标记”（high water mark）即可。因此，标示使用者状态所需的元数据总量实际上特别小。在Kafka中，我们将该最高水位标记称为“偏移量”（offset），这么叫的原因将在实现细节部分讲解。

使用者的状态

在Kafka中，由使用者负责维护反映哪些消息已被使用的状态信息（偏移量）。典型情况下，Kafka使用者的library会把状态数据保存到Zookeeper之中。然而，让使用者将状态信息保存到保存它们的消息处理结果的那个数据存储（datastore）中也许会更佳。例如，使用者也许就是要把一些统计值存储到集中式事物OLTP数据库中，在这种情况下，使用者可以在进行那个数据库数据更改的同一个事务中将消息使用状态信息存储起来。这样就消除了分布式的部分，从而解决了分布式中的一致性问题！这在非事务性系统中也有类似的技巧可用。搜索系统可用将使用者状态信息同它的索引段（index segment）存储到一起。尽管这么做可能无法保证数据的持久性（durability），但却可用让索引同使用者状态信息保存同步：如果由于宕机造成有一些没有刷新到磁盘的索引段信息丢了，我们总是可用从上次建立检查点（checkpoint）的偏移量处继续对索引进行处理。与此类似，Hadoop的加载作业（load job）从Kafka中并行加载，也有相同的技巧可用。每个Mapper在map任务结束前，将它使用的最后一个消息的偏移量存入HDFS。

这个决策还带来一个额外的好处。使用者可用故意回退（rewind）到以前的偏移量处，再次使用一遍以前使用过的数据。虽然这么做违背了队列的一般协约（contract），但对很多使用者来讲却是个很基本的功能。举个例子，如果使用者的代码里有个Bug，而且是在它处理完一些消息之后才被发现的，那么当把Bug改正后，使用者还有机会重新处理一遍那些消息。

Push和Pull

相关问题还有一个，就是到底是应该让使用者从代理那里吧数据Pull（拉）回来还是应该让代理把数据Push（推）给使用者。和大部分消息系统一样，Kafka在这方面遵循了一种更加传统的设计思路：由生产者将数据Push给代理，然后由使用者将数据代理那里Pull回来。近来有些系统，比如scribe和flume，更着重于日志统计功能，遵循了一种非常不同的基于Push的设计思路，其中每个节点都可以作为代理，数据一直都是向下游Push的。上述两种方法都各有优缺点。然而，因为基于Push的系统中代理控制着数据的传输速率，因此它难以应付大量不同种类的使用者。我们的设计目标是，让使用者能以它最大的速率使用数据。不幸的是，在Push系统中当数据的使用速率低于产生的速率时，使用者往往会处于超载状态（这实际上就是一种拒绝服务攻击）。基于Pull的系统在使用者的处理速度稍稍落后的情况下会表现更佳，而且还可以让使用者在有能力的时候往往前赶赶。让使用者采用某种退避协议（backoff protocol）向代理表明自己处于超载状态，可以解决部分问题，但是，将传输速率调整到正好可以完全利用（但从不能过度利用）使用者的处理能力可比初看上去难多了。以前我们尝试过多次，想按这种方式构建系统，得到的经验教训使得我们选择了更加常规的Pull模型。

分发

Kafka通常情况下是运行在集群中的服务器上。没有中央的“主”节点。代理彼此之间是对等的，不需要任何手动配置即可可随时添加和删除。同样，生产者和消费者可以在任何时候开启。 每个代理都可以在Zookeeper(分布式协调系统)中注册的一些元数据（例如，可用的主题）。生产者和消费者可以使用Zookeeper发现主题和相互协调。关于生产者和消费者的细节将在下面描述。

生产者

生产者自动负载均衡

对于生产者，Kafka支持客户端负载均衡，也可以使用一个专用的负载均衡器对TCP连接进行负载均衡调整。专用的第四层负载均衡器在Kafka代理之上对TCP连接进行负载均衡。在这种配置的情况，一个给定的生产者所发送的消息都会发送给一个单个的代理。使用第四层负载均衡器的好处是，每个生产者仅需一个单个的TCP连接而无须同Zookeeper建立任何连接。不好的地方在于所有均衡工作都是在TCP连接的层次完成的，因而均衡效果可能并不佳（如果有些生产者产生的消息远多于其它生产者，按每个代理对TCP连接进行平均分配可能会导致每个代理接收到的消息总数并不平均）。

采用客户端基于zookeeper的负载均衡可以解决部分问题。如果这么做就能让生产者动态地发现新的代理，并按请求数量进行负载均衡。类似的，它还能让生产者按照某些键值（key）对数据进行分区（partition）而不是随机乱分，因而可以保存同使用者的关联关系（例如，按照用户id对数据使用进行分区）。这种分法叫做“语义分区”（semantic partitioning），下文再讨论其细节。

下面讲解基于zookeeper的负载均衡的工作原理。在发生下列事件时要对zookeeper的监视器（watcher）进行注册：

- 加入了新的代理
- 有一个代理下线了
- 注册了新的话题
- 代理注册了已有话题。

生产者在其内部为每一个代理维护了一个弹性的连接（同代理建立的连接）池。通过使用zookeeper监视器的回调函数（callback），该连接池在建立/保持同所有在线代理的连接时都要进行更新。当生产者要求进入某特定话题时，由分区者（partitioner）选择一个代理分区（参加语义分区小结）。从连接池中找出可用的生产者连接，并通过它将数据发送到刚才所选的代理分区。

异步发送

对于可伸缩的消息系统而言，异步非阻塞式操作是不可或缺的。在Kafka中，生产者有个选项（producer.type=async）可用指定使用异步分发出产请求（produce request）。这样就允许用一个内存队列（in-memory queue）把生产请求放入缓冲区，然后再以某个时间间隔或者事先配置好的批量大小将数据批量发送出去。因为一般来说数据会从一组以不同的数据速度生产数据的异构的机器中发布出，所以对于代理而言，这种异步缓冲的方式有助于产生均匀一致的流量，因而会有更佳的网络利用率和更高的吞吐量。

语义分区

下面看看一个想要为每个成员统计一个个人空间访客总数的程序该怎么做。应该把一个成员的所有个人空间访问事件发送给某特定分区，因此就可以把对一个成员的所有更新都放在同一个使用者线程中的同一个事件流中。生产者具有从语义上将消息映射到有效的Kafka节点和分区之上的能力。这样就可以用一个语义分区函数将消息流按照消息中的某个键值进行分区，并将不同分区发送给各自相应的代理。通过实现kafak.producer.Partitioner接口，可以对分区函数进行定制。在缺省情况下使用的是随即分区函数。上例中，那个键值应该是member_id，分区函数可以是hash(member_id)%num_partitions。

对**Hadoop**以及其它批量数据装载的支持

具有伸缩性的持久化方案使得Kafka可支持批量数据装载，能够周期性将快照数据载入进行批量处理的离线系统。我们利用这个功能将数据载入我们的数据仓库（data warehouse）和Hadoop集群。

批量处理始于数据载入阶段，然后进入非循环图（acyclic graph）处理过程以及输出阶段（支持情况在这里）。支持这种处理模型的一个重要特性是，要有重新装载从某个时间点开始的数据的能力（以防处理中有任何错误发生）。

对于Hadoop，我们通过在单个的map任务之上分割装载任务对数据的装载进行了并行化处理，分割时，所有节点/话题/分区的每种组合都要分出一个来。Hadoop提供了任务管理，失败的任务可以重头再来，不存在数据被重复的危险。

实施细则

下面给出了一些在上一节所描述的低层相关的实现系统的某些部分的细节的简要说明。

API 设计

生产者 APIs

生产者 API 是给两个底层生产者的再封装

-kafka.producer.SyncProducerandkafka.producer.async.AsyncProducer.

```
class Producer {

  /* Sends the data, partitioned by key to the topic using either the */
  /* synchronous or the asynchronous producer */
  public void send(kafka.javaapi.producer.ProducerData producerData);

  /* Sends a list of data, partitioned by key to the topic using either */
  /* the synchronous or the asynchronous producer */
  public void send(java.util.List< kafka.javaapi.producer.ProducerData> producerData);

  /* Closes the producer and cleans up */
  public void close();

}
```

该API的目的是将生产者的所有功能通过一个单个的API公开给其使用者（client）。新建的生产者可以：

- 对多个生产者请求进行排队/缓冲并异步发送批量数据 —— kafka.producer.Producer提供了在将多个生产请求序列化并发送给适当的Kafka代理分区之前，对这些生产请求进行批量处理的能力（producer.type=async）。批量的大小可以通过一些配置参数进行控制。当事件进入队列时会先放入队列进行缓冲，直到时间到了queue.time或者批量大小到达batch.size为止，后台线程（kafka.producer.async.ProducerSendThread）会将这批数据从队列中取出，交给kafka.producer.EventHandler进行序列化并发送给适当的kafka代理分区。通过event.handler这个配置参数，可以在系统中插入一个自定义的事件处理器。在该生产者队列管道中的各个不同阶段，为了插入自定义的日志/跟踪代码或者自定义的监视逻辑，如能注入回调函数会非常有用。通过实现kafka.producer.asyn.CallbackHandler接口并将配置参数callback.handler设置为实现类就能够实现注入。

- 使用用户指定的Encoder处理数据的序列化（serialization）。Encoder的缺省值是一个什么活都不干的kafka.serializer.DefaultEncoder。

- 提供基于zookeeper的代理自动发现功能 —— 通过使用zk.connect配置参数指定zookeeper的连接url，就能够使用基于zookeeper的代理发现和负载均衡功能。在有些应用场合，可能不太适合于依赖zookeeper。在这种情况下，生产者可以从broker.list这个配置参数中获得一个代理的静态列表，每个生产请求会被随即的分配给各代理分区。如果相应的代理宕机，那么生产请求就会失败。

- 通过使用一个可选性的、由用户指定的Partitioner，提供由软件实现的负载均衡功能 —— 数据发送路径选择决策受kafka.producer.Partitioner的影响。分区API根据相关的键值以及系统中具有的代理分区的数量返回一个分区id。将该id用作索引，在broker_id和partition组成的经过排序的列表中为相应的生产者请求找出一个代理分区。缺省的分区策略是hash(key)%numPartitions。如果key为null，那就进行随机选择。使用partitioner.class这个配置参数也可以插入自定义的分区策略。

使用者API

我们有两个层次的使用者API。底层比较简单的API维护了一个同单个代理建立的连接，完全同发送给服务器的网络请求相吻合。该API完全是无状态的，每个请求都带有一个偏移量作为参数，从而允许用户以自己选择的任意方式维护该元数据。

高层API对使用者隐藏了代理的具体细节，让使用者可运行于集群中的机器之上而无需关心底层的拓扑结构。它还维护着数

据使用的状态。高层API还提供了订阅同一个过滤表达式（例如，白名单或黑名单的正则表达式）相匹配的多个话题的能力。

底层**API**

```
class SimpleConsumer {

  /* Send fetch request to a broker and get back a set of messages. */
  public ByteBufferMessageSet fetch(FetchRequest request);

  /* Send a list of fetch requests to a broker and get back a response set. */
  public MultiFetchResponse multifetch(List<FetchRequest> fetches);

  /**
   * Get a list of valid offsets (up to maxSize) before the given time.
   * The result is a list of offsets, in descending order.
   * @param time: time in millisecs,
   *              if set to OffsetRequest$.MODULE$.LATIEST_TIME(), get from the latest offset available.
   *              if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the earliest offset available.
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);
}
```

底层API不但用于实现高层API，而且还直接用于我们的离线使用者（比如Hadoop这个使用者），这些使用者还对状态的维护有比较特定的需求。

高层**API**

```
/* create a connection to the cluster */
ConsumerConnector connector = Consumer.create(consumerConfig);

interface ConsumerConnector {

  /**
   * This method is used to get a list of KafkaStreams, which are iterators over
   * MessageAndMetadata objects from which you can obtain messages and their
   * associated metadata (currently only topic).
   *  Input: a map of <topic, #streams>
   *  Output: a map of <topic, list of message streams>
   */
  public Map<String,List<KafkaStream>> createMessageStreams(Map<String,Int> topicCountMap);

  /**
   * You can also obtain a list of KafkaStreams, that iterate over messages
   * from topics that match a TopicFilter. (A TopicFilter encapsulates a
   * whitelist or a blacklist which is a standard Java regex.)
   */
  public List<KafkaStream> createMessageStreamsByFilter(
      TopicFilter topicFilter, int numStreams);

  /* Commit the offsets of all messages consumed so far. */
  public commitOffsets()

  /* Shut down the connector */
  public shutdown()
}
```

该API的中心是一个由KafkaStream这个类实现的迭代器（iterator）。每个KafkaStream都代表着一个从一个或多个分区到一个或多个服务器的消息流。每个流都是使用单个线程进行处理的，所以，该API的使用者在该API的创建调用中可以提供所需的任意个数的流。这样，一个流可能会代表多个服务器分区的合并（同处理线程的数目相同），但每个分区只会把数据发送给一个流中。

createMessageStreams方法为使用者注册到相应的话题之上，这将导致需要对使用者/代理的分配情况进行重新平衡。为了将重新平衡操作减少到最小。该API鼓励在一次调用中就创建多个话题流。createMessageStreamsByFilter方法为发现同其过滤条件想匹配的话题（额外地）注册了多个监视器（watchers）。应该注意，createMessageStreamsByFilter方法所返回的每个流都可能会对多个话题进行迭代（比如，在满足过滤条件的话题有多个的情况下）。

网络层

网络层就是一个特别直截了当的NIO服务器，在此就不进行过于细致的讨论了。sendfile是通过给MessageSet接口添加了一个writeTo方法实现的。这样就可以让基于文件的消息更加高效地利用transferTo实现，而不是使用线程内缓冲区读写方式。线程模型用的是一个单个的接收器（acceptor）线程和每个可以处理固定数量网络连接的N个处理器线程。这种设计方案在别处已经经过了非常彻底的检验，发现其实现起来简单、运行起来很快。其中使用的协议一直都非常简单，将来还可以用其它语言实现其客户端。

消息

消息由一个固定大小的消息头和一个变长不透明字节数字的有效载荷构成（opaque byte array payload）。消息头包含格式的版本信息和一个用于探测出坏数据和不完整数据的CRC32校验。让有效载荷保持不透明是个非常正确的决策：在用于序列化的代码库方面现在正在取得非常大的进展，任何特定的选择都不可能适用于所有的使用情况。都不用说，在Kafka的某特定应用中很有可能在它的使用中需要采用某种特殊的序列化类型。MessageSet接口就是一个使用特殊的方法对NIOChannel进行大宗数据读写（bulk reading and writing to an NIOChannel）的消息迭代器。

消息的格式

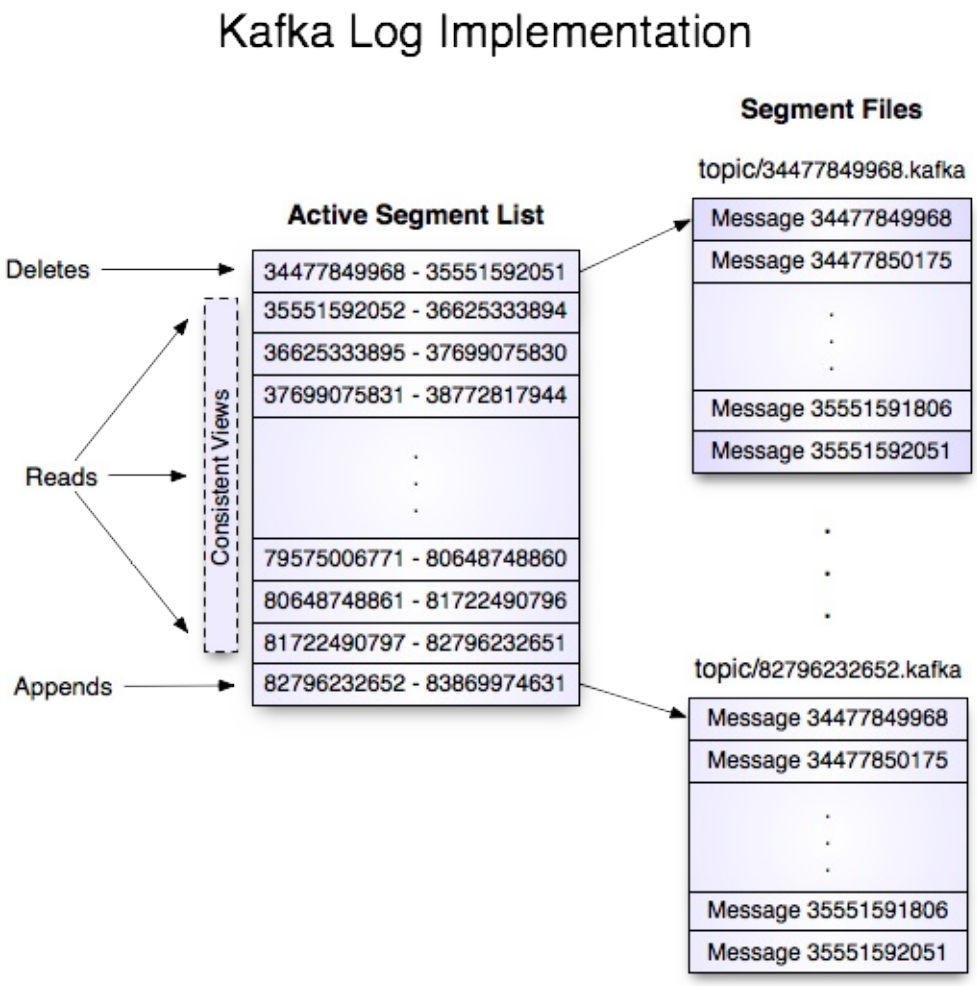
```
/**
     * A message. The format of an N byte message is the following:
     *
     * If magic byte is 0
     *
     * 1. 1 byte "magic" identifier to allow format changes
     *
     * 2. 4 byte CRC32 of the payload
     *
     * 3. N - 5 byte payload
     *
     * If magic byte is 1
     *
     * 1. 1 byte "magic" identifier to allow format changes
     *
     * 2. 1 byte "attributes" identifier to allow annotations on the message independent of the version (e.g. compressi
     *
     * 3. 4 byte CRC32 of the payload
     *
     * 4. N - 6 byte payload
     *
     */
```

日志

具有两个分区的、名称为"my_topic"的话题的日志由两个目录组成（即：my_topic_0和my_topic_1），目录中存储的是内容为该话题的消息的数据文件。日志的文件格式是一系列的“日志项”；每条日志项包含一个表示消息长度的4字节整数N，其后接着保存的是N字节的消息。每条消息用一个64位的整数偏移量进行唯一性标示，该偏移量表示了该消息在那个分区中的那个话题下发送的所有消息组成的消息流中所处的字节位置。每条消息在磁盘上的格式如下文所示。每个日志文件的以它所包含的第一条消息的偏移量来命名。因此，第一个创建出来的文件的名字将为00000000000.kafka，随后每个后加的文件的名字将是前一个文件的文件名大约再加S个字节所得的整数，其中，S是配置文件中指定的最大日志文件的大小。

消息的确切的二进制格式都有版本，它保持为一个标准的接口，让消息集可以根据需要在生产者、代理、和使用者直接进行自由传输而无须重新拷贝或转换。其格式如下所示： On-disk format of a messagemessage length : 4 bytes (value: 1+4+n) "magic" value : 1 bytecrc : 4 bytespayload : n bytes

将消息的偏移量作为消息的可不常见。我们原先的想法是使用由生产者产生的GUID作为消息id，然后在每个代理上作一个从GUID到偏移量的映射。但是，既然使用者必须为每个服务器维护一个ID，那么GUID所具有的全局唯一性就失去了价值。更有甚者，维护将从一个随机数到偏移量的映射关系带来的复杂性，使得我们必须使用一种重量级的索引结构，而且这种结构还必须与磁盘保持同步，这样我们还就必须使用一种完全持久化的、需随机访问的数据结构。如此一来，为了简化查询结构，我们就决定使用一个简单的依分区的原子计数器（atomic counter），这个计数器可以同分区id以及节点id结合起来唯一的指定一条消息；这种方法使得查询结构简化不少，尽管每次在处理使用者请求时仍有可能会涉及多次磁盘寻道操作。然而，一旦我们决定使用计数器，跳向直接使用偏移量作为id就非常自然了，毕竟两者都是分区内具有唯一性的、单调增加的整数。既然偏移量是在使用者API中并不会体现出来，所以这个决策最终还是属于一个实现细节，进而我们就选择了这种更加高效的方式。



写操作

日志可以顺序添加，添加的内容总是保存到最后一个文件。当大小超过配置中指定的大小（比如说1G）后，该文件就会换成另外一个新文件。有关日志的配置参数有两个，一个是M，用于指出写入多少条消息之后就要强制OS将文件刷新到磁盘；另一个是S，用来指定过多少秒就要强制进行一次刷新。这样就可以保证一旦发生系统崩溃，最多会有M条消息丢失，或者最长会有S秒的数据丢失，

读操作

可以通过给出消息的64位逻辑偏移量和S字节的数据块最大的字节数对日志文件进行读取。读取操作返回的是这S个字节中包含的消息的迭代器。S应该要比最长的单条消息的字节数大，但在出现特别长的消息情况下，可以重复进行多次读取，每次的缓冲区大小都加倍，直到能成功读取出这样长的一条消息。也可以指定一个最大的消息和缓冲区大小并让服务器拒绝接收比这个大小大一些的消息，这样也能给客户端一个能够读取一条完整消息所需缓冲区的大小的上限。很有可能会出现读取缓冲区以一个不完整的消息结尾的情况，这个情况用大小界定（size delimiting）很容易就能探知。

从某偏移量开始进行日志读取的实际过程需要先找出存储所需数据的日志段文件，从全局偏移量计算出文件内偏移量，然后再从该文件偏移量处开始读取。搜索过程通过对每个文件保存在内存中的范围值进行一种变化后的二分查找完成。

日志提供了获取最新写入的消息的功能，从而允许从“当下”开始消息订阅。这个功能在使用者在SLA规定的天数内没能正常使用数据的情况下也很有用。当使用者企图从一个并不存在的偏移量开始使用数据时就会出现这种情况，此时使用者会得到一个OutOfRangeException异常，它可以根据具体的使用情况对自己进行重启或者仅仅失败而退出。

以下是发送给数据使用者（consumer）的结果的格式。 MessageSetSend (fetch result)total length : 4 byteserror code : 2 bytesmessage 1 : x bytes...message n : x bytesMultiMessageSetSend (multiFetch result)total length : 4 byteserror code : 2 bytesmessageSetSend 1...messageSetSend n

删除

一次只能删除一个日志段的数据。 日志管理器允许通过可加载的删除策略设定删除的文件。 当前策略删除修改事件超过N 天以上的文件，也可以选择保留最后 N GB 的数据。 为了避免删除时的读取锁定冲突，我们可以使用副本写入模式，以便在进行删除的同时对日志段的一个不变的静态快照进行二进制搜索。

数据正确性保证

日志功能里有一个配置参数M，可对在强制进行磁盘刷新之前可写入的消息的最大条目数进行控制。在系统启动时会运行一个日志恢复过程，对最新的日志段内所有消息进行迭代，以对每条消息项的有效性进行验证。一条消息项是合法的，仅当其大小加偏移量小于文件的大小并且该消息中有效载荷的CRC32值同该消息中存储的CRC值相等。在探测出有数据损坏的情况下，就要将文件按照最后一个有效的偏移量进行截断。

要注意，这里有两种必需处理的数据损坏情况：由于系统崩溃造成的未被正常写入的数据块（block）因而需要截断的情况以及由于文件中被加入了毫无意义的数据块而造成的数据损坏情况。造成数据损坏的原因是，一般来说OS并不能保证文件索引节点（inode）和实际数据块这两者的写入顺序，因此，除了可能会丢失未刷新的已写入数据之外，在索引节点已经用新的文件大小更新了但在将数据块写入磁盘块之前发生了系统崩溃的情况下，文件就可能会获得一些毫无意义的数据。CRC值就是用于这种极端情况，避免由此造成整个日志文件的损坏（尽管未得到保存的消息当然是真的找不回来了）。

分发

Zookeeper目录

接下来讨论zookeeper用于在使用者和代理直接进行协调的结构和算法。

记法

当一个路径中的元素是用[xyz]这种形式表示的时，其意思是, xyz的值并不固定而且实际上xyz的每种可能的值都有一个zookpeer z节点（znode）。例如，/topics/[topic]表示了一个名为/topics的目录，其中包含的子目录同话题对应，一个话题一个目录并且目录名即为话题的名称。也可以给出数字范围，例如[0...5]，表示的是子目录0、1、2、3、4。箭头->用于给出z节点的内容。例如/hello -> world表示的是一个名称为/hello的z节点，包含的值为"world"。

代理节点的注册

/brokers/ids/[0...N] --> host:port (ephemeral node)

上面是所有出现的代理节点的列表，列表中每一项都提供了一个具有唯一性的逻辑代理id，用于让使用者能够识别代理的身份（这个必须在配置中给出）。在启动时，代理节点就要用/brokers/ids下列出的逻辑代理id创建一个z节点，并在自己注册到系统中。使用逻辑代理id的目的是，可以让我们在不影响数据使用者的情况下就能把一个代理搬到另一台不同的物理机器上。试图用已在使用中的代理id（比如说，两个服务器配置成了同一个代理id）进行注册会导致发生错误。

因为代理是以非长久性z节点的方式注册的，所以这个注册过程是动态的，当代理关闭或宕机后注册信息就会消失（至此要数据使用者，该代理不再有效）。

代理话题的注册

/brokers/topics/[topic]/[0...N] --> nPartions (ephemeral node)

每个代理会都要注册在某话题之下，注册后它会维护并保存该话题的分区总数。

使用者和使用者小组

为了对数据的使用进行负载均衡并记录使用者使用的每个代理上的每个分区上的偏移量，所有话题的使用者都要在Zookeeper中进行注册。

多个使用者可以组成一个小组共同使用一个单个的话题。同一小组内的每个使用者共享同一个给定的group_id。比如说，如果某个使用者负责用三台机器进行某某处理过程，你就可以为这组使用者分配一个叫做“某某”的id。这个小组id是在使用者的配置文件中指定的，并且这就是你告诉使用者它到底属于哪个组的方法。

小组内的使用者要尽量公正地划分出分区，每个分区仅为小组内的一个使用者所使用。

使用者ID的注册

除了小组内的所有使用者都要共享一个group_id之外，每个使用者为了要同其它使用者区别开来，还要有一个非永久性的、具有唯一性的consumer_id(采用hostname:uuid的形式）。 consumer_id要在以下的目录中进行注册。
/consumers/[group_id]/ids/[consumer_id] --> {"topic1": #streams, ..., "topicN": #streams} (ephemeral node)

小组内的每个使用者都要在它所属的小组中进行注册并采用consumer_id创建一个z节点。z节点的值包含了一个的map。consumer_id只是用来识别小组内活跃的每个使用者。使用者建立的z节点是个临时性的节点，因此如果这个使用者进程终止了，注册信息也将随之消失。

数据使用者偏移追踪

数据使用者跟踪他们在每个分区中耗用的最大偏移量。这个值被存储在一个Zookeeper(分布式协调系统)目录中。

/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id] --> offset_counter_value ((persistent node)

分区拥有者注册表

每个代理分区都被分配给了指定使用者小组中的单个数据使用者。数据使用者必须在耗用给定分区前确立对其的所有权。要确立其所有权，数据使用者需要将其 id 写入到特定代理分区中的一个临时节点(ephemeral node)中。
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id] --> consumer_node_id (ephemeral node)

代理节点的注册

代理节点之间基本上都是相互独立的，因此它们只需要发布它们拥有的信息。当有新的代理加入进来时，它会将自己注册到代理节点注册目录中，写下它的主机名和端口。代理还要将已有话题的列表和它们的逻辑分区注册到代理话题注册表中。在代理上生成新话题时，需要动态的对话题进行注册。

使用者注册算法

当使用者启动时，它要做以下这些事情：

1. 将自己注册到它属小组下的使用者id注册表。
2. 注册一个监视使用者id列的表变化情况（有新的使用者加入或者任何现有使用者的离开）的变化监视器。(每个变化都会触发一次对发生变化的使用者所属的小组内的所有使用者进行负载均衡。)
3. 主次一个监视代理id注册表的变化情况（有新的代理加入或者任何现有的代理的离开）的变化监视器。（每个变化都会触发一次对所有小组内的所有使用者负载均衡。）
4. 如果使用者使用某话题过滤器创建了一个消息流，它还要注册一个监视代理话题变化情况（添加了新话题）的变化监视器。（每个变化都会触发一次对所有可用话题的评估，以找出话题过滤器过滤出哪些话题。新过滤出来的话题将触发一次对该使用者所在的小组内所有的使用者负载均衡。）
5. 迫使自己在小组内进行重新负载均衡。

使用者重新负载均衡的算法

使用者重新复杂均衡的算法可用让小组内的所有使用者对哪个使用者使用哪些分区达成一致意见。使用者重新负载均衡的动作每次添加或移除代理以及同一小组内的使用者时被触发。对于一个给定的话题和一个给定的使用者小组，代理分区是在小组内的所有使用者中进行平均划分的。一个分区总是由一个单个的使用者使用。这种设计方案简化了实施过程。假设我们运行多个使用者以并发的方式同时使用同一个分区，那么在该分区上就会形成争用（contention）的情况，这样一来就需要某种形式的锁定机制。如果使用者的个数比分区多，就会出现有写使用者根本得不到数据的情况。在重新进行负载均衡的过程中，我们按照尽量减少每个使用者需要连接的代理的个数的方式，尝尝试着将分区分配给使用者。

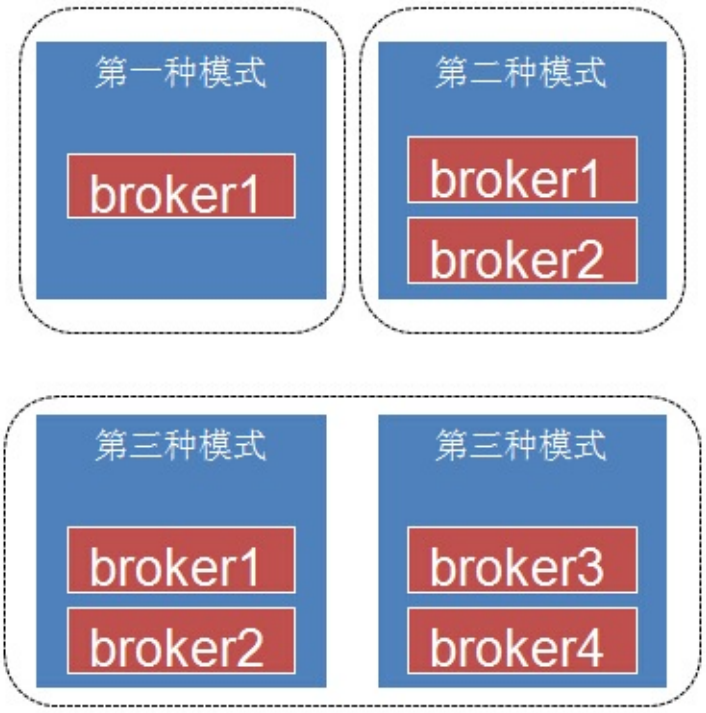
每个使用者在重新进行负载均衡时需要做下列的事情：

1. 针对Ci所订阅的每个话题T
2. 将PT设为生产话题T的所有分区
3. 将CG设为小组内同Ci 一样使用话题T的所有使用者
4. 对PT进行排序（让同一个代理上的各分区挨在一起）
5. 对CG进行排序
6. 将i设为Ci在CG中的索引值并让N = size(PT)/size(CG)
7. 将从i*N*到*(i+1)*N - 1的分区分配给使用者Ci
8. 将Ci当前所拥有的分区从分区拥有者注册表中删除
9. 将新分配的分区加入到分区拥有者注册表中 （我们可能需要多次尝试才能让原先的分区拥有者释放其拥有权）

在触发了一个使用者要重新进行负载均衡时，同一小组内的其它使用者也会几乎在同时被触发重新进行负载均衡。

部署模式

```
/*************
*kafka 0.8.1.1的安装部署
**************/
```



kafka的部署模式为3种模式

1. 单broker模式
2. 单机多broker模式 (伪集群)
3. 多机多broker模式 (真正的集群模式)

第一种模式安装

1. 在hadoopdn2机器上面上传kafka文件，并解压到 /opt/hadoop/kafka下面

2. 修改 /opt/hadoop/kafka/kafka_2.9.2-0.8.1.1/config/server.properties 配置文件

   > broker.id=0 默认不用修改 log.dirs=/opt/hadoop/kafka/kafka-logs log.flush.interval.messages=10000 默认不用修改 log.flush.interval.ms=1000 默认不用修改 zookeeper.connect=hadoopdn2:2181

3. 启动kafka的broker

> bin/kafka-server-start.sh config/server.properties

正常启动如下：

```
[2014-11-18 10:36:32,196] INFO Client environment:java.io.tmpdir=/tmp (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:java.compiler=<NA> (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:os.name=Linux (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:os.arch=amd64 (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:os.version=2.6.32-220.el6.x86_64 (org.apache.zookeeper.ZooKeeper)
```

```
[2014-11-18 10:36:32,196] INFO Client environment:user.name=hadoop (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:user.home=/home/hadoop (org.apache.zookeeper.ZooKeeper)
[2014-11-18 10:36:32,196] INFO Client environment:user.dir=/opt/hadoop/kafka/kafka_2.9.2-0.8.1.1 (org.apache.zookeeper.
[2014-11-18 10:36:32,197] INFO Initiating client connection, connectString=localhost:2181 sessionTimeout=6000 watcher=c
[2014-11-18 10:36:32,231] INFO Opening socket connection to server localhost/0:0:0:0:0:0:0:1:2181 (org.apache.zookeeper
[2014-11-18 10:36:32,238] INFO Socket connection established to localhost/0:0:0:0:0:0:0:1:2181, initiating session (org
[2014-11-18 10:36:32,262] INFO Session establishment complete on server localhost/0:0:0:0:0:0:0:1:2181, sessionid = 0x3
[2014-11-18 10:36:32,266] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
[2014-11-18 10:36:32,415] INFO Starting log cleanup with a period of 60000 ms. (kafka.log.LogManager)
[2014-11-18 10:36:32,422] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogMana
SLF4J: Failed to load class “org.slf4j.impl.StaticLoggerBinder”.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[2014-11-18 10:36:32,502] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)
[2014-11-18 10:36:32,503] INFO [Socket Server on Broker 0], Started (kafka.network.SocketServer)
[2014-11-18 10:36:32,634] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader$)
[2014-11-18 10:36:32,716] INFO 0 successfully elected as leader (kafka.server.ZookeeperLeaderElector)
[2014-11-18 10:36:32,887] INFO Registered broker 0 at path /brokers/ids/0 with address JobTracker:9092. (kafka.utils.Zk
[2014-11-18 10:36:32,941] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
[2014-11-18 10:36:33,034] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
```

1. 创建topics

   > bin/kafka-topics.sh –create –zookeeper hadoopdn2:2181 –replication-factor 1 –partitions 1 –topic test

2. 查看队列列表

   > bin/kafka-topics.sh –list –zookeeper hadoopdn2:2181

3. 查看队列明细

   > bin/kafka-topics.sh –describe –zookeeper hadoopdn2:2181 –topic test

```
Topic[队列]:test    PartitionCount[分区数量]:1    ReplicationFactor:1    Configs:
Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
```

> 第一行是所有Partition的总结。后面的行是每个partition一行。

1. 查看帮助文档
   > bin/kafka-topics.sh –help 查看与topics相关的指令

第二种模式部署：

1. 为第二个broker创建server的配置文件

   > cp server.properties server1.properties

2. 修改server1.properties

```
broker.id=1
port=9093
log.dirs=/opt/hadoop/kafka/kafka-logs-server1
zookeeper.connect=hadoopdn2:2181
```

3. 启动kafka的broker

   > nohup bin/kafka-server-start.sh config/server1.properties &

4. 通过zookeeper的客户端可以查看当前的broker

```
[zk: hadoopdn2:2181(CONNECTED) 7] ls /
```

```
[zookeeper, admin, consumers, config, controller, brokers, controller_epoch]
[zk: hadoopdn2:2181(CONNECTED) 8] ls /brokers
[topics, ids]
[zk: hadoopdn2:2181(CONNECTED) 9] ls /brokers/ids
[1, 0]
```

- 查看队列情况

> bin/kafka-topics.sh –describe test –zookeeper hadoopdn2:2181

```
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
```

- 修改**test**队列的参数

  bin/kafka-topics.sh –zookeeper hadoopdn2:2181 –partitions 3 –topic test –alter

  bin/kafka-topics.sh –describe –zookeeper hadoopdn2:2181 –topic test_kafka

```
Topic:test    PartitionCount:3    ReplicationFactor:1    Configs:
Topic: test    Partition: 0    Leader: 0    Replicas: 0[在broker0上面]    Isr: 0
Topic: test    Partition: 1    Leader: 1    Replicas: 1[在broker1上面]    Isr: 1
Topic: test    Partition: 2    Leader: 0    Replicas: 0[在broker0上面]    Isr: 0
```

> partiton： partion id，由于此处只有一个partition，因此partition id 为0 leader：当前负责读写的lead broker id relicas：当前partition的所有replication broker list isr：relicas的子集，只包含出于活动状态的broker

Replicas复制备份机制：

> kafka将每个partition数据复制到多个server上,任何一个partition有一个leader和多个follower(可以没有);备份的个数可以通过broker配置文件来设定.leader处理所有的read-write请求,follower需要和leader保持同步.Follower和consumer一样,消费消息并保存在本地日志中;leader负责跟踪所有的follower状态,如果follower”落后”太多或者失效,leader将会把它从replicas同步列表中删除.当所有的follower都将一条消息保存成功,此消息才被认为是”committed”,那么此时consumer才能消费它.即使只有一个replicas实例存活,仍然可以保证消息的正常发送和接收,只要zookeeper集群存活即可.(不同于其他分布式存储,比如hbase需要”多数派”存活才行)
>
> 当leader失效时,需在followers中选取出新的leader,可能此时follower落后于leader,因此需要选择一个”up-to-date”的follower.选择follower时需要兼顾一个问题,就是新leader server上所已经承载的partition leader的个数,如果一个server上有过多的partition leader,意味着此server将承受着更多的IO压力.在选举新leader,需要考虑到”负载均衡”.

kafka分配机制：

kafka使用zookeeper来存储一些meta信息,并使用了zookeeper watch机制来发现meta信息的变更并作出相应的动作(比如consumer失效,触发负载均衡等)

1. Broker node registry: 当一个kafka broker启动后,首先会向zookeeper注册自己的节点信息(临时znode),同时当broker和zookeeper断开连接时,此znode也会被删除.
   > 格式: /broker/ids/[0...N] –>host:port;其中[0..N]表示broker id,每个broker的配置文件中都需要指定一个数字类型的id(全局不可重复),znode的值为此broker的host:port信息.

2. Broker Topic Registry: 当一个broker启动时,会向zookeeper注册自己持有的topic和partitions信息,仍然是一个临时znode.

   > 格式: /broker/topics/[topic]/[0...N] 其中[0..N]表示partition索引号.

3. Consumer and Consumer group: 每个consumer客户端被创建时,会向zookeeper注册自己的信息;此作用主要是为了”负载均衡”.

   一个group中的多个consumer可以交错的消费一个topic的所有partitions;简而言之,保证此topic的所有partitions都能被此

group所消费,且消费时为了性能考虑,让partition相对均衡的分散到每个consumer上.

4. Consumer id Registry: 每个consumer都有一个唯一的ID(host:uuid,可以通过配置文件指定,也可以由系统生成),此id用来标记消费者信息.

   > 格式: /consumers/[group_id]/ids/[consumer_id]

   仍然是一个临时的znode,此节点的值为{“topic_name”:#streams…},即表示此consumer目前所消费的topic + partitions列表.

5. Consumer offset Tracking: 用来跟踪每个consumer目前所消费的partition中最大的offset.

   > 格式: /consumers/[group_id]/offsets/[topic]/[broker_id-partition_id]–>offset_value

   此znode为持久节点,可以看出offset跟group_id有关,以表明当group中一个消费者失效,其他consumer可以继续消费.

6. Partition Owner registry: 用来标记partition被哪个consumer消费.临时znode

   > 格式: /consumers/[group_id]/owners/[topic]/[broker_id-partition_id] –>consumer_node_id
   >
   > 当consumer启动时,所触发的操作:

   i. 首先进行”Consumer id Registry”;
   ii. 然后在”Consumer id Registry”节点下注册一个watch用来监听当前group中其他consumer的”leave”和”join”;只要此znode path下节点列表变更,都会触发此group下consumer的负载均衡.(比如一个consumer失效,那么其他consumer接管 partitions).
   iii. 在”Broker id registry”节点下,注册一个watch用来监听broker的存活情况;如果broker列表变更,将会触发所有的groups下的consumer重新balance.

```
1.Producer端使用zookeeper用来”发现”broker列表,以及和Topic下每个partition leader建立socket连接并发送消息.
2.Broker端使用zookeeper用来注册broker信息,已经监测partition leader存活性.
3.Consumer端使用zookeeper用来注册consumer信息,其中包括consumer消费的partition列表等,同时也用来发现broker列表,并和partit
```

第三种部署方式：

1. 在hadoopdn3机器上面上传kafka文件， 并解压到 /opt/hadoop/kafka下面

2. 修改 /opt/hadoop/kafka/kafka_2.9.2-0.8.1.1/config 下面的server.properties 配置文件 broker.id=2 必须修改保证每个broker的ID唯一 修改 log.dirs=/opt/hadoop/kafka/kafka-logs log.flush.interval.messages=10000 默认不用修改 log.flush.interval.ms=1000 默认不用修改 zookeeper.connect=hadoopdn2:2181

3. 通过zookeeper的客户端查看

> [zk: hadoopdn2:2181(CONNECTED) 10] ls /brokers/ids
>
> [2, 1, 0]

broker的id为2的已经注册到zookeeper上面了

4.删除kafka的队列[注意需要重启kafka集群,测试发现bug]

> bin/kafka-run-class.sh kafka.admin.DeleteTopicCommand –topic test_kafka –zookeeper hadoopdn2:2181

5.创建多副本的队列

> bin/kafka-topics.sh –create –zookeeper hadoopdn2:2181 –replication-factor 1 –partitions 1 –topic test_kafka

6.查看多版本队列的明细

> $ bin/kafka-topics.sh –describe –zookeeper hadoopdn2:2181 –topic test_kafka
>
> $ bin/kafka-topics.sh –describe –zookeeper hadoopdn2:2181 –topic test_kafka

```
Topic:test_kafka    PartitionCount:3    ReplicationFactor:3    Configs:
Topic: test_kafka    Partition: 0    Leader: 1    Replicas: 1,2,0    Isr: 1,2,0
Topic: test_kafka    Partition: 1    Leader: 2    Replicas: 2,0,1    Isr: 2,0,1
Topic: test_kafka    Partition: 2    Leader: 0    Replicas: 0,1,2    Isr: 0,1,2
```

> //————————————————————————————————

kafka的集群测试

1)发送消息

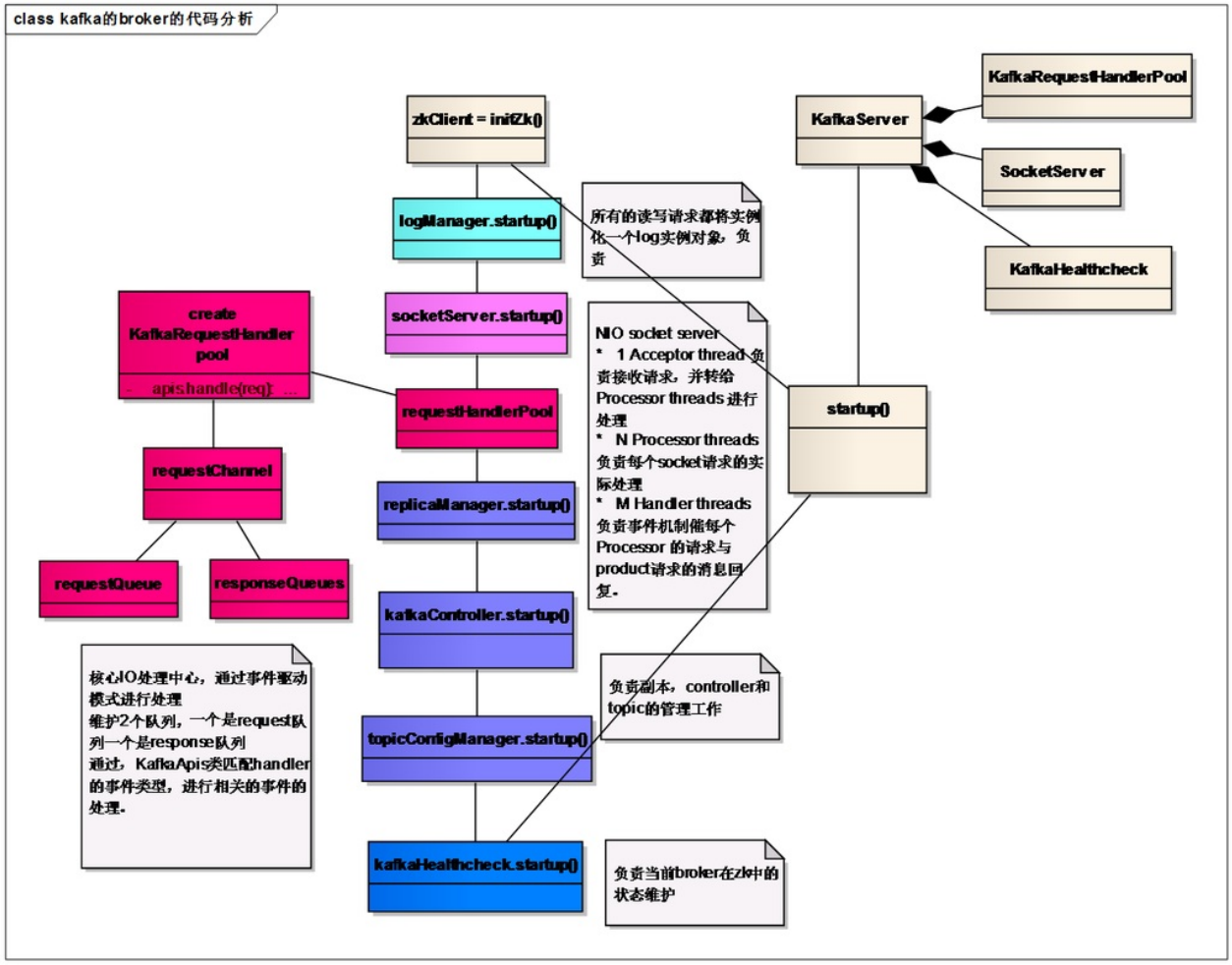
$ bin/kafka-console-producer.sh –broker-list JobTracker:9092 –topic test

2)消费消息

$ bin/kafka-console-consumer.sh –zookeeper hadoopdn2:2181 –topic test –from-beginning

核心设计

broker核心代码设计



主要的核心类如下：

> KafkaServer:
>
> 根据相关的上下文，创建KafkaServer对象，负责启动broker对象

启动过程

> zkClient 初始化=>logManager start()=> socketServer start()=>requestHandlerPool 初始化=>replicaManager start() => kafkaController start() =>topicConfigManager start() => kafkaHealthcheck start()
>
> Broker的核心IO处理在requestHandlerPool 这里面

zkClient初始化

> 利用第三方的zkclient初始与zk之间的链接，zkclient负责与zk的connection，session的维护。

logManager 初始化

> 所有的读写请求都将实例化各自的log实例对象。同时，后台的log线程将负责维护partitions与log segment。

socketServer初始化

> NIO socket server

- 1 Acceptor thread 负责接收请求，并转给Processor threads 进行处理
- N Processor threads 负责每个socket请求的实际处理
- M Handler threads 负责事件机制催每个Processor 的请求与product请求的消息回复。

replicaManager初始化

> 负责该broker的partition副本的管理工作

kafkaController初始化

> 负责该broker的Controller状态的管理工作

topicConfigManager初始化

> 负责该broker的topic的状态管理工作

kafkaHealthcheck初始化

> 负责该broker的状态在zk的维护工作

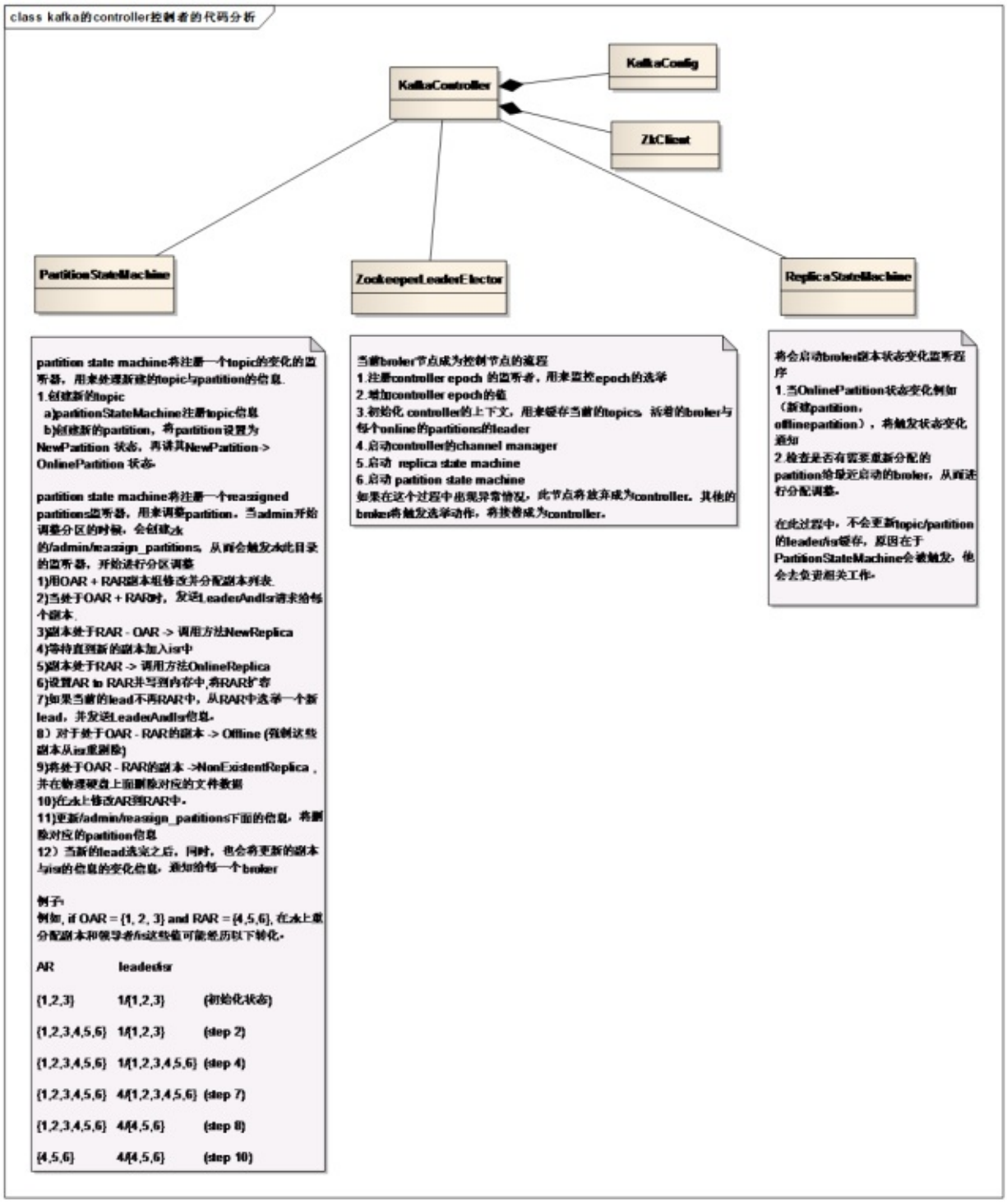
requestHandlerPool初始化

> 核心IO处理中心，通过事件驱动模式进行处理,维护2个队列，

- 一个是request队列
- 一个是response队列

> 通过，KafkaApis类匹配handler的事件类型，进行相关的事件的处理。
>
> 请求的事件类型共有
>
> - ProduceKey
> - FetchKey
> - OffsetsKey
> - MetadataKey
> - LeaderAndIsrKey
> - StopReplicaKey
> - UpdateMetadataKey
> - ControlledShutdownKey
> - OffsetCommitKey
> - OffsetFetchKey

controller的代码设计



主要的核心类如下：

controller :

> 根据相关的上下文，创建KafkaController对象，引入多个监听器监听broker，topic，partition以及副本的状态变化。

ZookeeperLeaderElector：

> 主要负责选举当前broker为lead的过程，同时，如果出现异常情况转移lead选举权。

ReplicaStateMachine：

> 主要负责broker的副本状态变化跟踪与重新分配的工作

PartitionStateMachine：

> 主要负责topic与partition的状态变化跟踪与重新分配的工作

ZookeeperLeaderElector详解

当前**broker**节点成为控制节点的流程

1. 注册controller epoch 的监听者，用来监控epoch的选举
2. 增加controller epoch的值
3. 初始化 controller的上下文，用来缓存当前的topics，活着的broker与每个online的partitions的leader
4. 启动controller的channel manager
5. 启动 replica state machine
6. 启动 partition state machine

> 如果在这个过程中出现异常情况，此节点将放弃成为controller。其他的broker将触发选举动作，将接替成为controller。

ReplicaStateMachine详解

将会启动**broker**副本状态变化监听程序

1. 当OnlinePartition状态变化例如（新建partition，offlinepartition），将触发状态变化通知
2. 检查是否有需要重新分配的partition给最近启动的broker，从而进行分配调整。

> 在此过程中，不会更新topic/partition的leader/isr缓存，原因在于PartitionStateMachine会被触发，他会去负责相关工作。

partition state machine详解

1. 创建新的topic
    i. partitionStateMachine注册topic信息
    ii. 创建新的partition，将partition设置为NewPartition 状态，再讲其NewPartition->OnlinePartition 状态
2. 分区调整

    > partition state machine将注册一个reassigned partitions监听器，用来调整partition。当admin开始调整分区的时候，会创建zk的/admin/reassign_partitions，从而会触发zk此目录的监听器，开始进行分区调整

    i. 用OAR + RAR副本组修改并分配副本列表.
    ii. 当处于OAR + RAR时，发送LeaderAndIsr请求给每个副本.
    iii. 副本处于RAR – OAR -> 调用方法NewReplica
    iv. 等待直到新的副本加入isr中
    v. 副本处于RAR -> 调用方法OnlineReplica
    vi. 设置AR to RAR并写到内存中,将RAR扩容

vii. 如果当前的lead不再RAR中，从RAR中选举一个新lead，并发送LeaderAndIsr信息。

viii. 对于处于OAR – RAR的副本 -> Offline (强制这些副本从isr重剔除)

ix. 将处于OAR – RAR的副本 ->NonExistentReplica ,并在物理硬盘上面删除对应的文件数据

x. 在zk上修改AR到RAR中。

xi. 更新/admin/reassign_partitions下面的信息，将删除对应的partition信息

xii. 当新的lead选完之后，同时，也会将更新的副本与isr的信息的变化信息，通知给每一个broker

> 例子：
>
> 例如, if OAR = {1, 2, 3} and RAR = {4,5,6}, 在zk上重分配副本和领导者/is这些值可能经历以下转化。

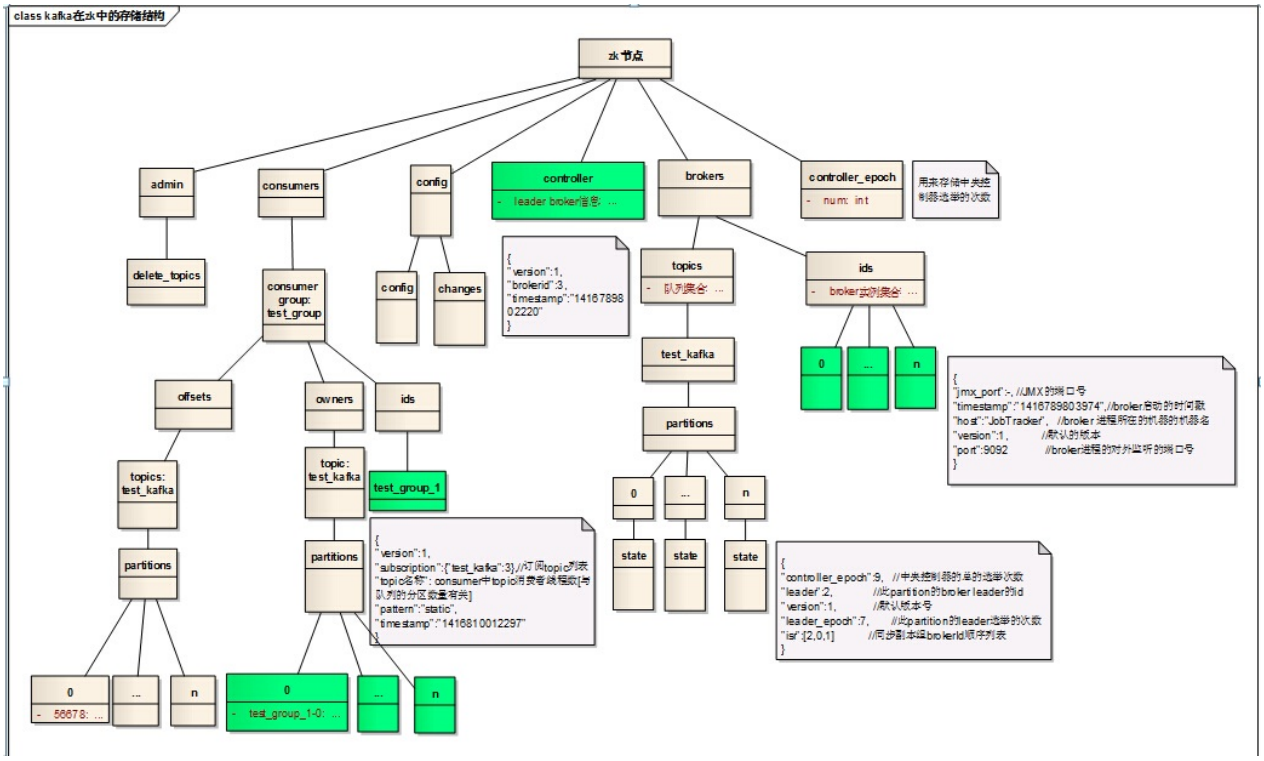
```
AR                  leader/isr

{1,2,3}             1/{1,2,3}           (初始化状态)

{1,2,3,4,5,6}       1/{1,2,3}           (step 2)

{1,2,3,4,5,6}       1/{1,2,3,4,5,6}     (step 4)

{1,2,3,4,5,6}       4/{1,2,3,4,5,6}     (step 7)

{1,2,3,4,5,6}       4/{4,5,6}           (step 8)

{4,5,6}             4/{4,5,6}           (step 10)
```

zk中的存储结构

涉及到的相关项目为

> kafka 0.8.1.1
>
> zookeeper 3.3.6

环境下面的存储的结构



图片中描述了kafka在zk中的存储结构，以及存储的相关数据，绿色代表的是zk的临时节点，当对应的进程退出后，此临时的znode将自动删除。由于consumer的offset节点保存对应的partition的消息队列的消息消费情况，当消费者退出后，继任的消费者需要在之前结束的地方继续下去，因此，此节点不是临时节点。

kafka创建的队列情况为：

```
Topic:test_kafka    PartitionCount:3    ReplicationFactor:3    Configs:
Topic: test_kafka    Partition: 0    Leader: 2    Replicas: 1,2,0    Isr: 2,0,1
Topic: test_kafka    Partition: 1    Leader: 2    Replicas: 2,0,1    Isr: 2,0,1
Topic: test_kafka    Partition: 2    Leader: 2    Replicas: 0,1,2    Isr: 2,0,1
```

> Partition 为3个，Replicas 为3个。

下面详细介绍每类主要节点：

Controller epoch:

> /controller_epoch -> int (epoch)

此值为一个数字,kafka集群中第一个broker第一次启动时为1，以后只要集群中center controller中央控制器所在broker变更或挂掉，就会重新选举新的center controller，每次center controller变更controller_epoch值就会 + 1;原因是Paxos算法更新分布式系统保证数据一致性的时候，需要确认当前事务id必须是最大的。

Broker注册信息

> /brokers/ids/[0...N]

每个broker的配置文件中都需要指定一个数字类型的id(全局不可重复),此节点为临时znode(EPHEMERAL)

```
Schema:
{
“jmx_port”:-,               //JMX的端口号
“timestamp”:”1416789803974",//broker启动的时间戳
“host”:”JobTracker”,        //broker 进程所在的机器的机器名
“version”:1,                //默认的版本
“port”:9092                 //broker进程的对外监听的端口号
}
```

> /brokers/topics/topic1/partition/[0...n]

保存broker上面建立的topic队列的相关信息，以及对应的分区的数量，以及每个分区的元数据。

```
Schema:
{
“controller_epoch”:9,   //中央控制器的总的选举次数
“leader”:2,             //此partition的broker leader的id
“version”:1,            //默认版本号
“leader_epoch”:7,       //此partition的leader选举的次数
“isr”:[2,0,1]           //同步副本组brokerId顺序列表
}
```

Controller注册信息**:**

存储center controller中央控制器所在kafka broker的信息

```
Schema:
 {
“version”:1,
“brokerid”:3,
“timestamp”:”1416789802220"
}
```

Consumer注册信息**:**

每个consumer都有一个唯一的ID(consumerId可以通过consumer的客户端配置文件指定,也可以由系统自动生成，建议开发者自己制定ID),此id用来标记消费者信息.

> /consumers/[groupId]/ids/[consumerIdString]

```
Schema:
 {
“version”:1,
“subscription”:{“test_kafka”:3},    //订阅topic列表
“topic名称”: consumer中topic消费者线程数[与队列的分区数量有关]
“pattern”:”static”,
“timestamp”:”1416810012297"
}
```

Consumer owner:

> /consumers/[groupId]/owners/[topic]/[partitionId]/consumer_thread

用来保存每个topic的的partition的是由那个消费者线程进行消费的信息。 Consumer offset:

> /consumers/[groupId]/offsets/[topic]/[partitionId] /offset number

此节点是持久化节点，保存当前需要处理的消息的偏移量，用来继任消费者继续此节点开始处理消息。

配置

server.properties详解

Samza分布式流计算

Samza已经在2015年2月初经过孵化后，成为apache的顶级项目。

官方网站：http://samza.apache.org

背景

> 参考链接：http://samza.incubator.apache.org/learn/documentation/0.8/introduction/background.html

先了解一下Samza的Background是必不可少的（至少官网上是放在第一个的），我们需要从哪些技术背景去了解呢？

什么是消息（**Messaging**）？

消息系统是一种实现近实时异步计算的流行方案。消息产生时可以被放入一个消息队列（ActiveMQ，RabbitMQ）、发布-订阅系统（Kestrel，Kafka）或者日志聚合系统（Flume、Scribe）。下游消费者从上述系统读取消息并且处理它们或者基于消息的内容产生进一步的动作。 假设你有一个网站，并且每次有人要加载一个页面，你发送一个“用户看了页面”的事件给一个消息系统。你可能会有一些做下面事情的消费者：

```
* 为了未来做数据分析，存储消息到hadoop；
* 对页面访问量进行计数并且更新到Dashboard
* 如果页面访问失败触发一个报警；
* 发送一封邮件通知另一个用户；
* 带着这个用户的相关信息加入页面展示事件，并且返回信息给消息系统；
```

总结一下，很显然，一个消息系统能解耦所有这些来自实际网页服务的工作。

那什么是流式计算（处理）？

大家知道消息系统是一个相当低层次的基础设施——它存储消息等待消费者消费他们。当你开始写产生或者消费消息的代码时，你很快会发现在处理层会有很多恶心的问题需要你亲自处理。而Samza的目标就是帮助我们干掉这些恶心的家伙！

咱们那上面提到的（计算pv并更新到dashboard）例子来说吧，当你的正在跑的消费者机器突然挂掉了，并且你当前的计算的数值丢失了会发生什么？怎么恢复？当机器服务被重启时处理该从哪里开始？如果底层的消息系统重复发送了一条信息或者丢失了一条消息怎么办？或者你想根据url来分组统计pv？又或者一台机器处理的负载太大，你想分流到多台机器上进行统计在聚合？

流式计算为上述问题提供了一个很好的解决方案，它是基于消息系统更高层次的抽象。

Samza

Samza是一个流式计算框架，它有以下特性：

- 简单的API：和绝大多数低层次消息系统API不同，相比MapReduce，Samza提供了一个非常简单的“基于回调（callback-based）”的消息处理API；
- 管理状态：samza管理快照和流处理器的状态恢复。当处理器重启，samza恢复其状态一致的快照。samza的建立是为了处理大量的状态；
- 容错性：当集群中有一台机器宕机了，基于Yarn管理的Samza会立即将你的任务导向另一台机器；
- 持久性：Samza通过kafka保证消息按顺序写入对应分区，并且不会丢失消息；
- 扩展性：Samza在每一层都做了分区和分布。kafka提供了顺序的、分区、可复制的、容错的流。Yarn则为Samza的运行提供了一个分布式环境；
- 可插拔：虽然Samza在Kafka和YARN的外部工作，但是Samza提供了可以让你在其它消息系统和执行环境里运行的可插拔的API；
- 处理器隔离：运行在YARN上的Samza同样支持Hadoop安全模型以及通过linux CGroups进行资源隔离

供选方案：

目前流行的开源流式计算方案都很年轻，并且没有一个单一系统能提供一个全面的解决方案。在这个领域面临的新难题包括如下几个：1.一个流式计算的状态应该怎样管理；2.流是否应该被缓冲到远程机器的磁盘上；3.当重复的信息被接受或者信息丢失该做什么；4.如何建立底层消息传递系统；

Samza的主要区别在于以下几个方面：

- Samza支持局部状态的容错。状态自己作为一个流被构造。如果因为机器宕机本地状态丢失，那么状态流会回放重新存储它。
- 流是有序、分区的、可回放的并且是容错的；
- YARN用来处理隔离、安全和容错；
- 任务之间是解耦的：如果有一个任务慢了并且造成了消息的积压，系统其它部分不会受到影响；

好的，背景就介绍到这里，下一篇咱们一起了解一些概念，方便后续深入学习吧，大家继续加油。

概念

参考链接：http://samza.incubator.apache.org/learn/documentation/0.8/introduction/concepts.html

希望上一篇背景篇让大家对流式计算有了宏观的认识，本篇根据官网是介绍概念，先让我们看看有哪些东西呢？

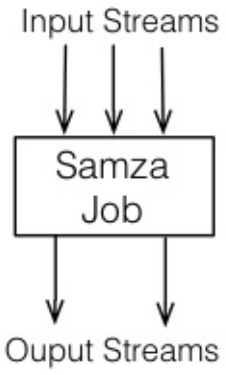
```
概念一：Streams
```

Samza是处理流的。流则是由一系列不可变的一种相似类型的消息组成。举个例子，一个流可能是在一个网站上的所有点击，或者更新到一个特定数据库表的更新操作，或者是被一个服务或者事件数据生成所有日志信息。消息能够被加到另一个流之后或者从一个流中读取。一个流能有多个消费者，并且从一个流中读取不会删除消息（使得消息能够被广播给所有消费者）。另外消息可以有一个关联的key用来做分区，这个将在后面说明。

Samza支持实现流抽取的可插拔系统：在kafka里，流是一个topic（话题），在数据库里我们可以通过消费从一个表里更新操作读取一个流；而在hadoop里我们可能跟踪在hdfs上的一个目录下的文件。

```
概念二：Jobs
```

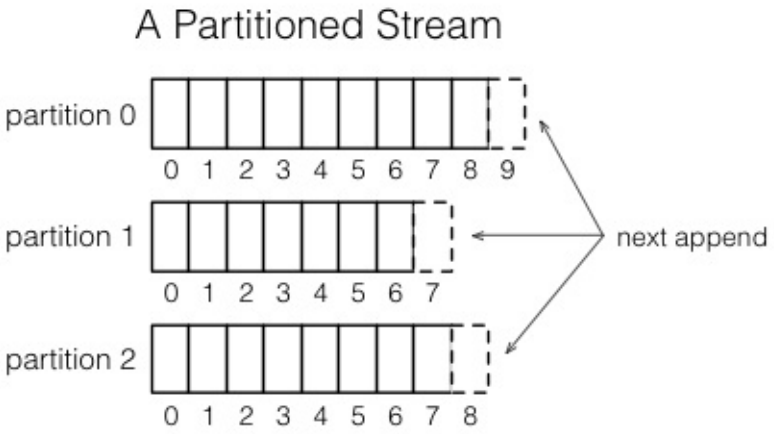
Samza的jobs 是对一组输入流设置附加值转化成输出流的程序（见下图）。为了扩展流处理器的吞吐量，我们将任务拆分更小的并行单元：分区Partitions和任务tasks




```
概念三：Partitions
```

每个流都被分割成一个或多个分区，并且在流里的每一个分区都总是一个有序的消息序列。每个消息在这个序列里有一个被叫做offset（中文称它为偏移量），它在每一个分区里都是唯一的。这个偏移量可以是一个连续的整数、字节偏移量或者字符串，这取决于底层的系统实现了。

当有一个消息加入到流中，它只会追加到流的分区中的一个。这个消息通过写入者带着一个被选择的key分配到它对应的分区中。举个例子，如果用户id被用作key，那么所有和用户id相关的消息都应该追加到这个分区中。



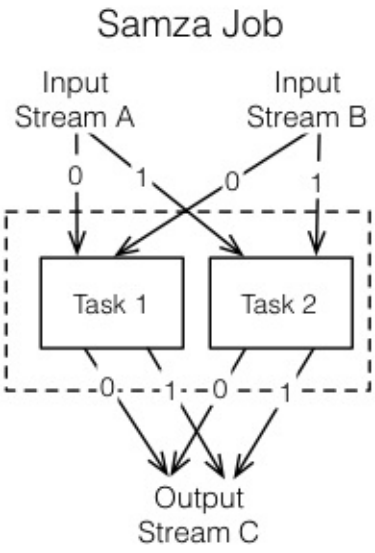
概念四：Tasks

一个job通过把他分割成多个任务Task进行扩展。任务Task作为job的并行单元，就好比上述提到的流中的分区。每个任务Task为每个job输入流消费来自一个分区的数据。

按照消息的偏移，一个任务按序处理来自它的输入分区的消息。分区之间没有定义顺序，这就允许每一个任务独立执行。YARN调度器负责分发任务给一台机器，所以作为一个整体的工作job可以分配到多个机器并行执行。

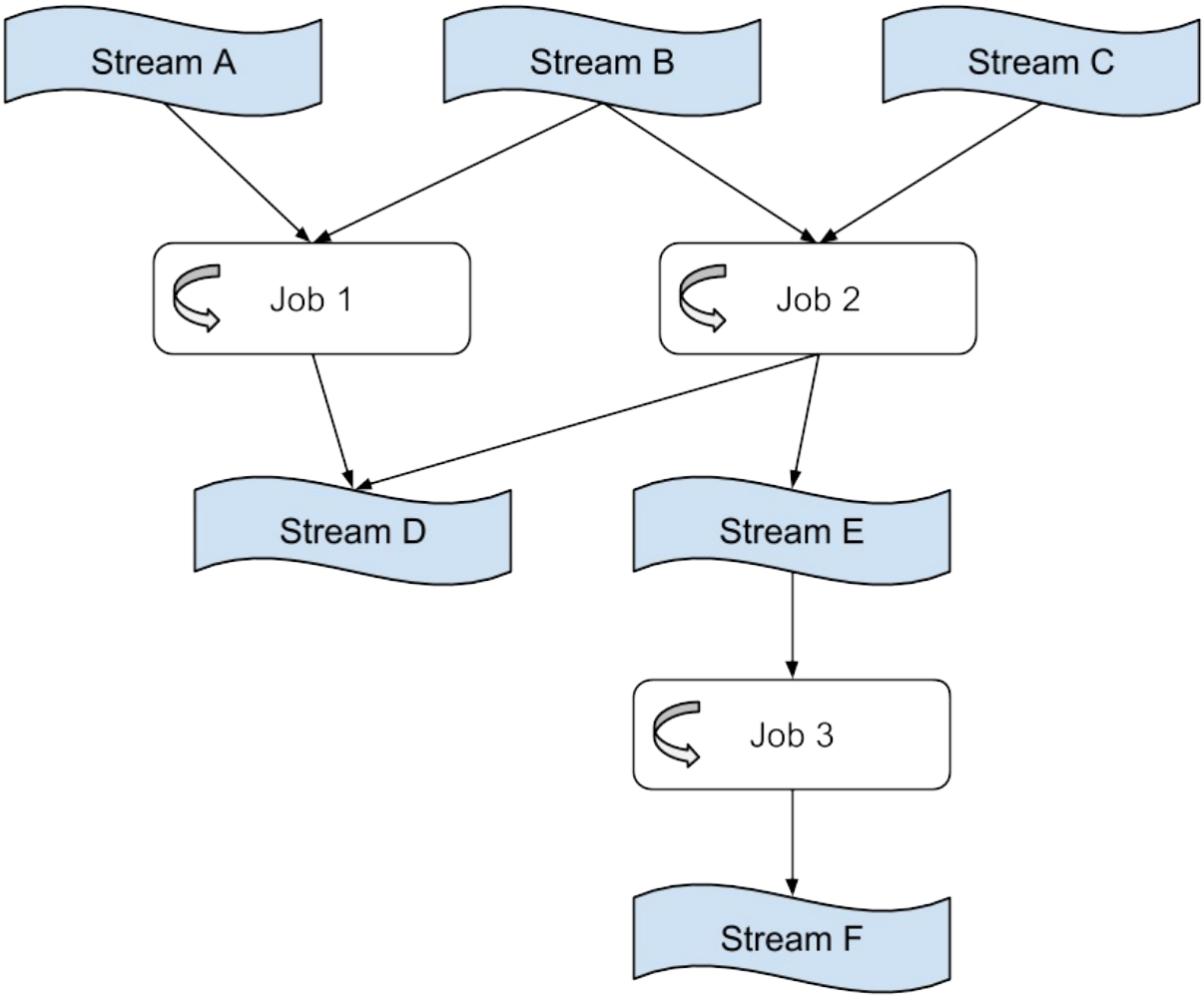
在一个job中任务Task的数量是由输入分区决定的（也就是说任务数目不能超过分区数目，否则就会存在没有输入的任务）。可是，你能改变分配给job的计算资源（比如内存、cpu核数等）去满足job的需要，可以参考下面关于container的介绍。

另外一个值得注意的是分配给task的分区的任务绝不会改变：如果有一个任务在一台失效的机器上，这个task会被在其它地方重启，仍然会消费同一个流的分区。



概念五：Dataflow Graphs

我们能组合多个jobs去创建一个数据流图，其中节点表示包含数据的流，而边则是进行数据传输。这个组合纯粹是通过jobs作为输入和输出的流来完成。这些jobs也是解耦的：他们不需要基于相同的代码库，并且添加、删除或者重启一个下游任务不会影响上游的任务。




```
概念六： Containers
```

分区Partitions和任务tasks都是并行的逻辑单元——他们不会与特定的计算资源（cpu、内存、硬盘等）的分配相符合。Containers则是物理的并行单元，并且一个容器本质上是一个Unix进程。每个容器跑着一个或多个tasks。tasks的数量是从输入的分区数自动确定和固定下来的，但是容器的数量（cpu、内存资源）是在运行时用户设定的并且能在任何时刻改变。

架构

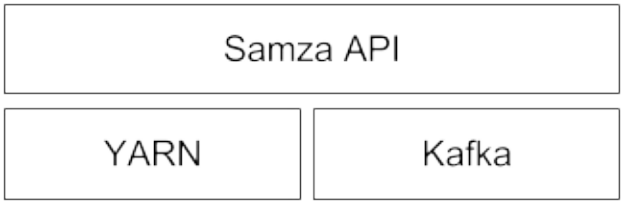
参考链接：http://samza.incubator.apache.org/learn/documentation/0.8/introduction/architecture.html

本篇紧接着概念篇，从宏观角度上看一下Samza实时计算服务的架构是什么样的？

Samza是由以下三层构成：

1. 数据流层（A streaming layer）
2. 执行层（An execution layer）
3. 处理层（A progressing layer）

那Samza是依靠哪些技术完成以上三层的组合呢？如下图所示：



1. 数据流：分布式消息中间件Kafka
2. 执行：Hadoop资源调度管理系统YARN
3. 处理： Samza API

使用过大数据Hadoop的开发者可以类比下面的架构模式（存储由HDFS负责，执行层由YARN负责，而处理层则由MapReduce负责），如下图所示：

在你进一步了解这三层的每一部分前，应该要注意到对于Samza的支持并不局限于使用Kafka和YARN，具体需要根据你的业务场景来确定使用什么技术框架、工具作支持。特别是Samza的执行层和数据流层都是可插拔的，并且允许开发者自己去实现更好的替代品。

咱们深入一点，先介绍数据流层的解决方案——kafka（熟悉kafka的开发人员可以跳过）。

这个有着浓厚文艺气息名号的Kafka是一个分布式发布/订阅消息队列系统，它支持至少一次通信保障（即系统保证没有信息丢失，但是在某些故障情况下，消费者可能收到超过一条同样的信息）和高度可用的分区特性（即使一台机器宕机了，分区依然是可用的）。

对于Kafka来讲，每一条数据流被称为一个话题（topic）。每一个话题都在多台被称作broker的机器上进行分区和复制。当一个生产者发送一条消息给一个话题，它会提供一个key，这个key被用来决定这条消息应该被发送到哪一个分区。生产者发送信息而Kafka的broker则接收和存储它们。kafka的消费者能通过在一个话题的所有分区上订阅消息来读取消息。

值得补充的是，kafka有一些有趣的特点：

- 带着同一个key的所有消息都被划分到同一个分区，这就意味着如果你想要读到一个特定用户的所有消息，你只要从包含这个用户id的分区读取即可，而不是整个topic（假设把用户id用作key）
- 一个话题的分区是按顺序到达的一序列消息，所以你可以通过单调递增偏移量offset来引用任何消息（就好比放一个索引到一个数组里）；这也意味着broker不需要跟踪被一个特定的消费者读取的消息，为什么呢？因为消费者保存了消息的偏移量offset能够跟踪到它。然后我们知道的是带着一个比当前偏移量小的消息是已经被处理过的，而每一个带着更大偏

移量的消息还没有被处理过

再来看看新一代Hadoop推出的资源管理系统YARN

YARN（Yet Another Resource Negotiator）是新一代hadoop集群调度器。它可以让你在一个集群中配置一个容器，并且执行任意命令。当一个应用和YARN相互交互时，它看起来像这样的：

1. Application：hi YARN Boy！我想用512MB内存在两台机器上跑命令X；
2. YARN：cool， 你的代码在哪里？
3. Application：代码在这里呢
4. YARN：我正在网格node1 和node2上执行你的job。

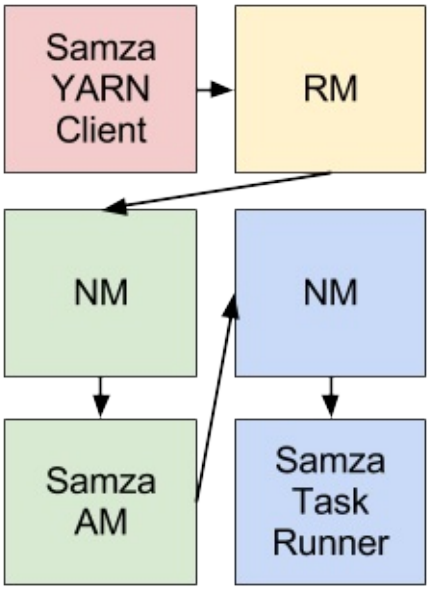
Samza使用YARN去管理部署、容错、日志、资源隔离、安全以及本地化。这里提供一个简要介绍（见 http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/）。

YARN的架构

当然为了节省大家的时间，宏观上来看一下YARN的架构吧。首先YARN也是有三个层次构成：一个资源管理器（ResourceManager），，一个节点管理器（NodeManager）和一个应用管理器（ApplicationMaster）。在一个YARN网格里，每一台机器上都跑着一台NodeManager，它负责在所在的机器上启动进程。而一个ResourceManager则与所有的NodeMananger交互告诉它们跑什么应用，反过来NodeManager也会告诉ResourceManager它们希望什么时间在集群里跑这些东东。对于第三层ApplicationMaster则是让特定应用的代码跑在YARN集群上。它负责管理应用的负载、容器（通常是UNIX进程），并且当其中一个容器失败时发出通知。

Samza and YARN

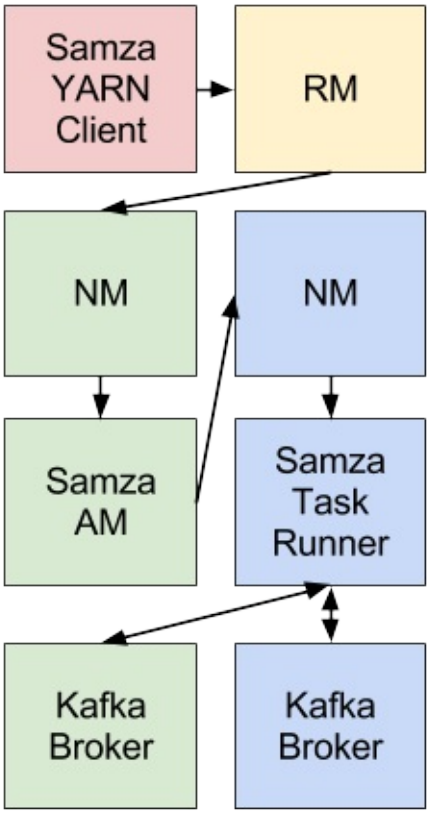
Samza提供了一个YARN ApplicationMaster和一个开箱即用的YARN任务运行器。这样说大家可能觉得不直观，Samza和YARN的集成可以用下图来概述（不同的颜色表示不同的机器）：



解释一下上面这个图吧，Samza的客户端当它想开始一个新job时会告诉YARN的RM（ResourceManager，以下简称RM）。这个RM会告诉YARN的一个NodeManager（简称NM）为Samza的ApplicationMaster（AM）在集群里分配空间。一旦NM分配了空间，就会启动这个Samza的AM。AM开始后，它会要求RM为了跑SamzaContainers需要更多的YARN的containers。而且RM会和NMs一起为containers分配空间。一旦空间被分配，NMs就会开启Samza containers。

Samza

简要介绍了一下YARN，接下来就是我们的焦点Samza。Samza通过使用YARN和Kafka提供一个阶段性的流处理和分区的框架。把它们三者放在一起的话大概就是这个样子了：



Samza的客户端使用YARN来运行一个Samza任务（job）：YARN启动并且监控一个或者多个SamzaContainers，并且你的处理逻辑代码（使用the StreamTask API）在这些容器里运行。这些Samza 流任务的输入和输出都来自Kafka的Brokers（通常他们都是作为YARN NMs位于同台机器）

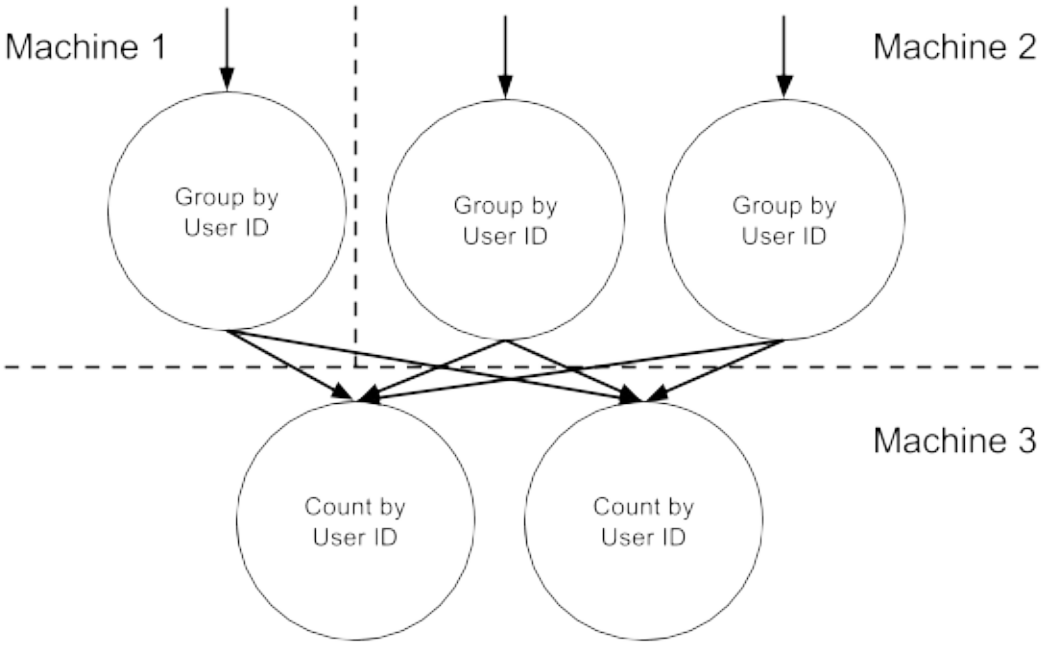
举个例子

比如我们想要统计页面访问数量。如果用SQL，你可能会写成下面这样的：

```
SELECT user_id, COUNT(*) FROM PageViewEvent GROUP BY user_id
```

尽管Samza目前不支持SQL，但是思路是相同的；计算这个需求需要两个任务：一个任务按照userid聚合消息，另一个任务则是计数。

进一步来说，第一个任务是分组工作通过发送带有相同userid的消息发送到一个中间话题的相同分区里，你可以通过使用第一个job发射的消息里的userid作为key来做到，并且这个key被映射到这个中间话题的分区（通常会取key对分区数目取余）。第二个任务处理中间话题产生的消息。在第二个任务里每个任务都会处理中间话题的一个分区。在对应分区中任务会针对每一个userid弄一个计数器，并且每次任务接受带着一个特定userid的消息时对应的计数器自增1。弄张图大家看看：



怎么样这个图是不是到处都看得到，和hadoop的Mapreduce的运行很相似对吧，每一个记录带着一个特定的key放到mapper里，被框架按照相同key进行分组，然后再reduce里进行计算统计。但是hadoop和Samza还是非常不同的，因为Hadoop的计算时基于一个固定的输入，而Samza则是和没有限定的数据流打交道。另外我自己再补充一条，流式计算框架和MapReduce另一个很大不同点在于mr的任务是会停止的，而Samza是持续不断处理。

Kafka接收到第一个job发送的消息并且把它们缓冲到硬盘，并且分布在多台机器上。这有助于系统的容错性提升：如果一台机器挂了，没有消息会被丢失，因为它们被存在其他机器里。并且如果第二个job因为某些原因消费消息的速度慢下来或者停止，第一个任务也没有影响：磁盘缓冲可以积累消息直到第二个任务快起来。

通过对topic的分区，将数据流处理拆解到任务中以及在多台机器上并行执行任务，使得Samza具有很高的消息吞吐量。通过结合YARN和Kafka，Samza实现了高容错：如果一个进程或者机器失败，它会自动在另一台机器上重启它并且继续从消息终端的地方开始处理，这些都是自动化的。

流处理系统对比

这里有一些使得Samza和其它流处理项目不同的高层设计决策。

The Stream Model 流模型

流是Samza job的输入和输出。Samza有非常强的流模型——不仅是一个简单的消息交换系统。Samza中的stream是一个分区的、每个分区有序的、可重放的、多订阅者的，无损的消息序列。(A stream in Samza is partitioned,ordered-per-partition, replayable, multi-subscriber, lossless sequence of messages)。stream不仅是系统的输入输出，而且将处理步骤进行缓存，使得它们互相隔离。

这个更强的模型需要持久化、容错、以及流的实现提供缓存功能。一个Samza job可以停止消费几分钟，甚至几小时(可能因为一个不好的配置或者长时间运行的运算)而不会对上游的job有任何影响。这使得Samza适于大规模布署，例如处理一个大型公司的所有数据流：job之间的独立使得它们可以被使用不同代码库、有不同SLA的不同小组来写代码实现、拥有、运行。

这样的设计受到我们使用Hadoop构建类似的离线处理流水线之经验的启发。在Hadoop中，processing stages是MapReduce jobs， processing stage的输出是HDFS里一个目录里的文件。下一个processing stage的输入就是前一个processsing stage的输出。我们发现stage之间的独立使得我们可以有数百个松耦合的job，这些job由不同的小组维护，组成了一个离线处理生态系统。我们的目标是把这样丰富的生态系统复制到近时实的环境里。

这个强“流模型”的第二个好处就是所有的stage都是多订阅者multi-subscriber的。在实际上，这意味着如果一个人添加了一些生成输数据流的处理流程，那么其它人可以看到这些输出，消费它们，构建于它们之上，而不会和另一个人的job有代码有任何耦合。一个讨人喜的副作用是这使用调试变得方便，因为你可以手动的查看任何一个stage的输出。

最后，这个强“流模型”strong stream model极大简化了Samza框架特性的实现。每个job只需要自己的输入输出，在出故障的时候，每个job都可以独自被恢复和重启。不必要对整个数据流图做中央控制。

为这个strong stream model而做的妥协是消息被写在磁盘上。我们乐于做出这个妥协因为MapReduce和HDFS展示了持久化存储也可以提供很高的读写吞吐量，并且几乎无限的磁盘空间。这个观察是kafka的基础，Kafka提供了数百MB/秒的有备份的吞吐，以及每个节点数TB的磁盘空间。所以按我们的使用方式，磁盘吞吐不是瓶颈。

MapReduce经常被指责进行不必要的磁盘写。但是，这个指责对于流处理不怎么适用：像MapReduce一样的批处理经常被用于处理在短时间内处理很大的历史数据集(比如，在10分钟内查询一分钟的数据)，而流处理大都需要跟进稳定的数据流(在10分钟内处理10分钟的数据)。这意味着流处理的原始吞吐量需求大都比批处理低几个数量级。 State 状态

只有非常简单的流处理问题才是无状态的(例如：一次处理一条消息，和其它消息都没有关系)。很多流处理程序需要它的任务能维护一些状态，例如：

- 如果需要知道对于一个user ID 已经有多少条消息？就需要为每个user 维持一个计数器。 如果想要知道每天有多少不同的用户访问了你的网站，你需要保持一个use ID的集合，这个集合里的user ID今天至少出现了一次。
- 如果你需要对两个流做join(比如，如果你需要知道广告的点进比click-through rate，你需要把展示广告的消息流和点击广告的消息流做join)你需要存储一个流中的消息直到你从另一个流中收到了相关的消息。
- 如果你需要把从数据库获得的一些信息添加进消息(比如，把页面访问消息用访问页面的用户的信息进行扩展)，这个job就需要访问数据库的当前状态。

一些状态，例如计数器，可以被保存在任务的内存中的计数器里，但是如果这个任务重启了，这些状态可能就丢失了。或者，你可以把这些状态存在远端的数据库里，但是如果你每处理一条消息就进行一次数据库查询，那么性能就可能差到不可接受。Kafka可以轻松地在每个节点上处理100k~500k消息/秒(和消息大小有关)，但是查询一个远端的键值存储的吞吐量可能只有1~5k请求/秒——慢了两个量级。

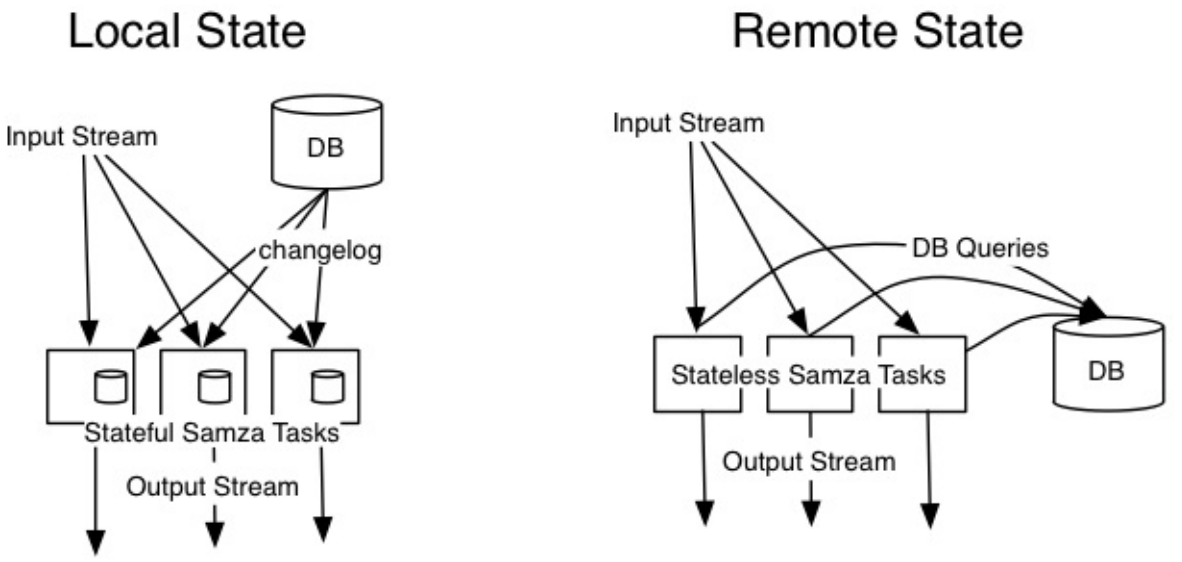
在Samza中，我们付出了专门的努力使得它具有高性能以及可靠的状态。关键在于对每个节点保持状态在本地(这样就不需要通过网络进行查询)，并且通过复制状态变化到其它流来使得它能应对机器故障(译注：这里的复制应该是说把当前任务的输出发到其它的Stream里，这就利用了其它Stream进行持久化)。

这种做法和数据库的“变化捕获”联系在一起就变得很有趣。拿之前的例子来说：你有一个包括了user ID的页面点击流，你想把这些消息用user的信息进行扩充。乍一看来，你唯一的选择就是对每一个user ID去查用数据库。但是有了Samza，就可以

做得更好。

“变化捕获”意味着每次数据库里的数据发生变化，你获得一个消息来告诉你变化是什么。如果你有一个这样的消息流，通过从数据库创建起回放整个流，你可以获得数据库的所有内容。这个changelog流同样可以做为Samza job的输入流。

现在你可以写一个Samza job，它使用页面点击事件和changelog做为输入。你确保它们使用同样的key(例如 user ID)做分区。每次一个changelog事件进来，你就把更新后的用户信息写在task的本地存储上。每次一个页面点击事件进来，你从本地存储中获取最新的用户信息。这样的话，你就可以把所有的状态保持在task的本地，而不用去查询远端的数据库。



事实上，现在你有了主数据库的一个副本，这个副本被分区，这些分区和Samza task在一台机器上。数据库写仍然需要通过主数据库，但是当你在处理消息时需要读数据库，那你就只需要查询本地的状态。

这种做法不仅比查询远端的数据库要快得多，而且它也更容易操作。如果你使用Samza处理一个很大容量的流，并且对每个消息都做一次远程查询，那么你很容易就使得数据库过载了，这样就会影响其它使用那个数据库的服务。但是，当一个task使用本地状态，它就和其它东西是独立的，就不会影响其它服务。

分区的本地状态不总是好的，也不总是必须的——Samza不会阻止你查询外部的数据库。如果你不能从你的数据库获取其变化信息，或者你需要其它远程服务的逻辑，那么当然从你的Samza job来请求一个远端的服务也是很方便的。但是如果你需要使用在本地的状态，那么 Samza本身提供了这功能。 Execution Framework 执行框架

我们所做的最后一个决定是不去为Samza构建一个自己的分布式执行系统。取而代之后是，执行环境是可插拔的，目前是由YARN来提供。这样有两点好处：

> 第一个好处是很实际的：有另一群聪明人在搞execution framework。YARN正在以很快的步骤开发，并且已经支持了很多跟资源限定和安全有关的功能。它允许你控制集群的哪一部分由哪些用户和用户组来使用，并且可以通过cgroups控制每个节点的资源使用(CPU,内存等)。YARN可以大规模运行以支持Hadoop，并且有希望成为无所不在的一层。因为Samza整个通过YARN运行，因此在YARN集群之外没有另外的守护进程或者master。也就是说，如果你已经有了Kafka和YARN，你就不用安装其它东西来运行Samza job。
>
> 第二点，Samza和YARN的组合完全是组件化的。和YARN的组合在另外一个包里，并且Samza的主框架在编译时并不依赖它。它意味着YARN可以被另一个虚拟化框架取代——我们对增加Samza和AWS的组合很感兴趣。很多公司使用AWS，AWS本身就是一个虚拟化框架，它和YARN对于Samza是一样的：它允许你来创建和销毁虚拟"container“机器以及保证这些container使用确定的资源fixed resources。因为流处理任务是长时间运行的，在AWS之内运行一个YARN集群，然后对单独的job进行调度，这种做法有点傻。反之，一个更合适的方法是为你的job直接分派一些EC2实例。

我们认为在像Mesos和YARN这样的开源的虚拟化框架，以及像Amazon提供的商业化的云服务里有很多创新，所有装Samza和它们进行组合是合理的。

MUPD8

开发者通常想看到是否有其他类似的系统可以做一个对比。我们会竭尽所能，拿其他系统的特征与samza的相关特性做一个对比。但我们不是这些系统的专家。我们的理解有可能完全失之偏颇。如果的确有的话，请让我们知道，我们会改正的。

持久性

MUPD8 没有实现持久性与消息传输的完整性（可能会有消息丢失）。在MUPD8内，流处理器接收消息的任务最多一次。samza使用kafka的消息系统，保证消息的持久化与消息传输的完整性。

顺序性

相对于持久性而言，开发者观念上更重视消息的处理是否能够按照他们写入时候的顺序进行处理。

我们并没有完整查看MUPD8对于顺序性的保障机制，看起来像是所有的消息进入队列后，按照入队的先后顺序处理消息，这点跟kafka的机制有些类似的。

消息的缓存

当我们设计一个流处理系统的时候，面临的一个关键问题就是，当我们的下游处理系统变慢的时候，我们的流处理系统将会承受越来越大的压力。

mupd8在两任务之间传递消息的时候，会缓冲消息在本地的内存队列中。当队列满时，开发者可以选择丢掉消息或者记录到本地磁盘上面。所有这些选择都不是最理想的。丢失消息将导致不正确的结果。记录到本地磁盘上面看似是最合理的选择，但当故障发生时，这些消息会由于没有副本机制，将导致无法实现故障故障转移，从而丢失消息。

samza是数据源采用kafka的作为远端缓冲，解决上面所有这些问题。kafka从架构上面解决了消息的副本与本地消息高效存储。[笔者add：samza的流处理系统是基于消息队列的分布式流处理系统]

数据状态管理

就如前面的介绍说的那样，流处理程序需要考虑在处理过程中保存数据处理的状态信息。不同的软件架构有不同的保存状态的方式和出现异常后的不同处理模式。 mupd8将状态维护到内存里面，并周期性的同步到Cassandra数据库中。

samza维护task的状态是在本地。这样能够维护的数据可以比内存更大。状态数据被持久化到了输出数据流中，这样当task失败的时候可以恢复。我们通过日志的变化来捕获状态的变化，从而能够将状态恢复到失败之前的时间点的一致性状态中，我们相信这样的设计将能够较好的支持服务的容错性。

部署与执行

MUPD8是有自己的执行框架，对以此框架的特点与特性我们并不清楚。

Samza 利用 YARN 来部署我们的代码,并在一个分布式的环境中执行相关代码。

容错机制

当我们的服务器down机之后，我们的流处理系统该做一些什么呢？ MUPD8使用类似Yarn的容管理错机制。当流处理器不能给下游流处理器发送消息的时候，它将通知MUPD8的控制器并且其他所有流处理器将会获得到通知，根据key的散列值与一台新的服务器建立通信，此时此message与对应的状态信息将会丢失。

Samza使用YARN管理容错。当node或者samza的task节点失败的时候，yorn会发现并通知Samza ApplicationMaster. 因此，具体如何来做就有samza来决定。通常来说，会在其他机器上面重启samza的task任务，因为消息数据是存储在远端的kafka服务器上面而不是本地的内存中，一次不用考虑数据的流失问题(除非是samza使用的是kafka的异步提交模式，为了提高发送的性能，但不得不承担数据丢失的风险)。

工作流

有时候为了完成一件事情，可能涉及的job不止一个.我们可能需要考虑将数据流进行重分区从而分成多个job进行并行执行，最后将执行的结果汇总到统一的一个job中进行汇总处理。

MUPD8 可以在流处理系统沙面设置来定义如何同时执行多个任务，以及如何从一个数据流到另一个数据流。

Samza是按照单个作业执行的粒度级别进行设计。通过再job中命名输入和输出流的名字进行数据交互。这隐含定义了所有正在运行的作业之间的数据流之间的关系。我们选择这个模式，可以使数据流图在不同团队的工程师在以及不同的代码库的基础上进行合作，无需任何将任何资源与技术统一到一个大的框架下面。

内存模式

MUPD8执行它的所有map/reduce的处理器统一在一个单一JVM内，使用线程机制。内存级别的数据共享效率比较高。

Samza使用为每个数据流处理器容器单独的JVM。与在单个JVM上运行多个数据流处理线程相比，将使用更多的内存，这个是缺点。然而，其优点是任务可以使更可靠的进行隔离，不同的任务之间不会受到影响。

资源隔离

MUPD8提供流处理器没有提供资源隔离的功能。流处理器可以使用节点上所有资源。

Samza使用Yarn进行资源隔离处理。每个进程的内存我们可以执行严格的限制。此外，Samza也支持CPU限制。Yarn未来将的进一步发展，也应该成为可能支持磁盘和网络进行相关限制。

Storm

开发者通常想看到是否有其他类似的系统可以做一个对比。我们会竭尽所能，拿其他系统的特征与samza的相关特性做一个对比。但我们不是这些系统的专家。我们的理解有可能完全失之偏颇。如果的确有的话，请让我们知道，我们会改正的。

storm与samza非常相似。这两种系统提供许多相同的高级特性：分区流模型，分布式执行环境，为流处理的API，容错，与kafka的集成等。

Storm 和 Samza 用了不同的名称来描述类似的功能: 在storm中的spouts 与samza的stream consumers是相同的概念, bolts(storm) 与 tasks(samza)是相同的概念, tuples(storm) 与 messages(samza) 是相同的概念. Storm 还增加了 building blocks ,这项功能在samza是没有的。

Spark

API

上一篇和大家一起宏观上学习了Samza平台的架构，重点讲了一下数据缓冲层和资源管理层，剩下的一块很重要的SamzaAPI层本节作为重点为大家展开介绍。 当你使用Samza来实现一个数据流处理逻辑时，你必须实现一个叫StreamTask的接口，如下所示：

```
public class MyTaskClass implements StreamTask {

  public void process(IncomingMessageEnvelope envelope,
                      MessageCollector collector,
                      TaskCoordinator coordinator) {
    // process message
  }
}
```

当你运行你的job时，Samza将为你的class创建一些实例(可能在多台机器上)。这些任务实例会处理输入流里的消息。在你的job的配置中你能告诉Samza你想消费哪条数据流。举一个较为完整的例子(大家也可以参看http://samza.incubator.apache.org/learn/documentation/0.8/jobs/configuration-table.html)：

```
# This is the class above, which Samza will instantiate when the job is run
task.class=com.example.samza.MyTaskClass

# Define a system called "kafka" (you can give it any name, and you can define
# multiple systems if you want to process messages from different sources)
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory

# The job consumes a topic called "PageViewEvent" from the "kafka" system
task.inputs=kafka.PageViewEvent

# Define a serializer/deserializer called "json" which parses JSON messages
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory

# Use the "json" serializer for messages in the "PageViewEvent" topic
systems.kafka.streams.PageViewEvent.samza.msg.serde=json
```

对于Samza从任务的输入流利接收的每一条消息，处理逻辑都会被调用。它主要包含三个重要的信息：消息、关键词key以及消息来自的数据流：

```
/** Every message that is delivered to a StreamTask is wrapped
 * in an IncomingMessageEnvelope, which contains metadata about
 * the origin of the message. */
public class IncomingMessageEnvelope {
  /** A deserialized message. */
  Object getMessage() { ... }

  /** A deserialized key. */
  Object getKey() { ... }

  /** The stream and partition that this message came from. */
  SystemStreamPartition getSystemStreamPartition() { ... }
}
```

注意键和值都要被声明为对象，并且需要转化为正确的类型。如果你不配置一个serializer/deserializer，它们就会成为典型的java字节数组。一个deserializer能够转化这些字节到其他任意类型，举个例子来说j一个son deserializer能够将字节数组转化为Map、List以及字符串对象。

SystemStreamPartition()这个方法会返回一个SystemStreamPartition对象，它会告诉你消息是从哪里来的。它由以下三部分组成：

1. The system：系统的名字来源于消息，就在你job的配置里定义。你可以有多个用于输入和输出的不同名字的系统；
2. The stream name： 在原系统里数据流（话题、队列）的名字。同样也是在job的配置里定义；

3. The partition： 一条数据流通常会被划分到多个分区，并且每一个分区会被Samza安排一个StreamTask实例；

API看起来像是这样的：

```
/** A triple of system name, stream name and partition. */
public class SystemStreamPartition extends SystemStream {

  /** The name of the system which provides this stream. It is
      defined in the Samza job's configuration. */
  public String getSystem() { ... }

  /** The name of the stream/topic/queue within the system. */
  public String getStream() { ... }

  /** The partition within the stream. */
  public Partition getPartition() { ... }
}
```

在上面这个job的配置例子里可以看到，这个系统名字叫“Kafka”，数据流的名字叫“PageViewEvent”。（kafka这个名字不是特定的——你能给你的系统取任何你想要的名字）。如果你有一些输入流向导入你的StreamTask，你能够使用SystemStreamPartition去决定你接受到哪一类消息。

如何发送消息呢？如果你看一下StreamTask里的process()方法，你将看到你有一个MessageCollector接口。

```
/** When a task wishes to send a message, it uses this interface. */
public interface MessageCollector {
  void send(OutgoingMessageEnvelope envelope);
}
```

为了发送一个消息，你会创建一个OutgoingMessageEnvelop对象并且把它传递给消息收集器。它至少会确定你想要发送的消息、系统以及数据流名字再发送出去。你也可以确定分区的key和另一些参数。具体可以参考javadoc（http://samza.incubator.apache.org/learn/documentation/0.7.0/api/javadocs/org/apache/samza/system/OutgoingMessageEnvelope.html）。

注意事项：

请只在process()方法里使用MessageCollector对象。如果你保持住一个MessageCollector实例并且之后再次使用它，你的消息可能会错误地发送出去。举一个例子，这儿有一个简单的任务，它把每个输入的消息拆成单词，并且发送每一个单词作为一个消息：

```
public class SplitStringIntoWords implements StreamTask {

  // Send outgoing messages to a stream called "words"
  // in the "kafka" system.
  private final SystemStream OUTPUT_STREAM =
    new SystemStream("kafka", "words");

  public void process(IncomingMessageEnvelope envelope,
                      MessageCollector collector,
                      TaskCoordinator coordinator) {
    String message = (String) envelope.getMessage();

    for (String word : message.split(" ")) {
      // Use the word as the key, and 1 as the value.
      // A second task can add the 1's to get the word count.
      collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM, word, 1));
    }
  }
}
```

samza的API的概要介绍就到这里吧，很多细节的API可以参看javadoc文档

容器 **Container**

SamzaContainer

Streams

Serialization

Checkpointing

State Management

Metrics

Windowing

Event Loop

JMX

Jobs

JobRunner

Configuration

YARN Jobs

Logging

YARN

Application Master

Storm分布式流计算

等待添加

Spark分布式内存计算

Spark是UC Berkeley AMP lab所开源的类Hadoop MapReduce的通用的并行，Spark，拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是Job中间输出结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的map reduce的算法。

Glossary