工业大数据实时分析与可视化

部署文档

撰写人	撰写时间	版本
李捷承	2016.11.30	1.0

目录

0 简介

0.1 项目目标

工业 大数据 实时 分析 与 可视化

0.2 系统简介

系统版本:

```
OS:         CentOS 7 1511 版
Python:     2.7.5 (CentOS 7 自带)
Java:       1.8.0_65 (CentOS 7 自带)
Zookeeper:  3.4.9
Storm:      1.0.2
Kafka:      0.9.0.1
Redis       3.2.5
Node:       6.9.1
```

IP 及 端口 分配:

```
Zookeeper: 192.168.1.1 192.168.1.2 192.168.1.3 开放端口: 2181 2888 3888
Storm: Nimbus: 192.168.1.4 192.168.1.5
       Supervisor: 192.168.1.6 192.168.1.7 192.168.1.8 192.168.1.9
       开放端口: 3772 3773 3774 6627 6699 8000 8080 6700 6701 6702 6703
Kafka: Broker: 192.168.1.10 192.168.1.11 192.168.1.12 开放端口: 9092 9999
Kafka Manager: 开放端口: 9000
Redis: 192.168.1.13 开放端口: 6379
Node: 192.168.1.14 开放端口: 取决于 node 服务器监听的端口(在 NodeJs 代码中写的)
```

PS: 建议直接在 firewall 中配置这些机器之间的互访不做端口过滤. 使用 rich rule: 对指定的 IP 不做拦截. 例如要设置来自 192.168.1.1 的访问不做端口过滤, 命令如下

```
sudo firewall-cmd --permanent --add-rich-rule="rule family='ipv4' source
address='192.168.1.1' accept"
```

1 Zookeeper

1.1 简介

Apache Zookeeper 是 Hadoop 的一个子项目, 是一个致力于开发和管理开源服务器, 并且能实现高可靠性的分布式协调框架. 它包含一个简单的原语集, 分布式应用程序可以基于它实现同步服务, 配置维护和命名服务等.

Zookeeper 保证 2n + 1 台机器的集群最大允许 n 台机器挂掉而事务不中断.

1.2 搭建

1.2.1 单机模式

此模式主要用于**开发人员本地环境下测试代码**

1.2.1.1 解压 Zookeeper 并进入其根目录

```
tar xzf zookeeper-3.4.9.tar.gz -C /usr/local/
cd /usr/local/zookeeper-3.4.9
```

1.2.1.2 创建配置文件 conf/zoo.cfg

```
cp conf/zoo_sample.cfg conf/zoo.cfg
```

1.2.1.3 修改内容如下:

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/lib/zookeeper/data
dataLogDir=/var/lib/zookeeper/logs
clientPort=2181
```

- tickTime: 是 zookeeper 的最小时间单元的长度(以毫秒为单位), 它被用来设置心跳检测和会话最小超时时间(tickTime 的两倍)
- initLimit: 初始化连接时能容忍的最长 tickTime 个数
- syncLimit: follower 用于同步的最长 tickTime 个数
- dataDir: 服务器存储 **数据快照** 的目录
- dataLogDir: 服务器存储 **事务日志** 的目录
- clientPort: 用于 client 连接的 server 的端口

其中需要注意的是`dataDir`和`dataLogDir`, 分别是 zookeeper 运行时的数据目录和日志目录, 要保证 **这两个目录已创建** 且 **运行 zookeeper 的用户拥有这两个目录的所有权**

1.2.1.4 测试

- 启动/关闭 Zookeeper:

```
bin/zkServer.sh start
bin/zkServer.sh stop
```

- 查看 Zookeeper 状态:

```
bin/zkServer.sh status
```

显示`mode: standalone`, 单机模式.

- 使用 java 客户端连接 ZooKeeper

```
./bin/zkCli.sh -server 127.0.0.1:2181
```

然后就可以使用各种命令了, 跟文件操作命令很类似, 输入 help 可以看到所有命令.

1.2.2 集群模式

此模式是 **生产环境中实际使用的模式**

因为 zookeeper 保证 2n + 1 台机器最大允许 n 台机器挂掉, 所以配置集群模式最好是奇数台机器: 3, 5, 7...

最少 3 台构成集群

1.2.2.1 hosts 映射(可选)

```
echo "192.168.1.1 zoo1" >> /etc/hosts
echo "192.168.1.2 zoo2" >> /etc/hosts
echo "192.168.1.3 zoo3" >> /etc/hosts
```

1.2.2.2 修改 zookeeper-3.4.9/conf/zoo.cfg 文件

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/lib/zookeeper/data
dataLogDir=/var/lib/zookeeper/log
clientPort=2181
server.1=192.168.1.1:2888:3888
server.2=192.168.1.2:2888:3888
server.3=192.168.1.3:2888:3888
```

与单机模式的不同就是最后三条: `server.X=host:portA:portB`

```
server.1=192.168.1.1:2888:3888
server.2=192.168.1.2:2888:3888
server.3=192.168.1.3:2888:3888
```

或

```
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

X 为标识为 X 的机器, host 为其 hostname 或 IP, portA 用于这台机器与集群中的 Leader 机器通信, portB 用于 server 选举 leader.

PS: 要配单机伪分布式的话, 可以修改这里为

```
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

然后每个 zookeeper 实例的 dataDir 和 dataLogDir 配置为不同的即可

1.2.2.3 myid 文件

在标示为 X 的机器上, 将 X 写入 ${dataDir}/myid 文件, 如: 在 192.168.1.2 机器上的 /var/lib/zookeeper/data 目录下建立文件 myid, 写入 2

```
echo "2" > /var/lib/zookeeper/data/myid
```

1.2.2.4 开放端口

CentOS 7 使用 firewalld 代替了原来的 iptables, 基本使用如下

```
systemctl start firewalld                                    # 启动防火墙
firewall-cmd --state                                         # 检查防火墙状
态
firewall-cmd --zone=public --add-port=2888/tcp --permanent   # 永久开启
2888 端口
firewall-cmd --reload                                        # 重新加载防火
墙规则
firewall-cmd --list-all                                      # 列出所有防火
墙规则
```

把 Zookeeper 用到的端口开放出来

```
firewall-cmd --zone=public --add-port=2181/tcp --permanent   # 永久开启
2181 端口
firewall-cmd --zone=public --add-port=2888/tcp --permanent   # 永久开启
2888 端口
firewall-cmd --zone=public --add-port=3888/tcp --permanent   # 永久开启
3888 端口
firewall-cmd --reload                                        # 重新加载防火
墙规则
```

1.2.2.5 测试

- 在 **集群中所有机器上** 启动 zookeeper(尽量同时):

```
bin/zkServer.sh start
```

- 查看状态, 应该有一台机器显示`mode: leader`, 其余为`mode: follower`

```
bin/zkServer.sh status
```

- 使用 java 客户端连接 ZooKeeper

```
./bin/zkCli.sh -server 192.168.1.1:2181
```

然后就可以使用各种命令了, 跟文件操作命令很类似, 输入help可以看到所有命令.

- 关闭 zookeeper:

```
./bin/zkServer.sh stop
```

1.2.3 Zookeeper 常见问题

查看状态时, 应该有一台机器显示`mode: leader`, 其余为`mode: follower`

```
bin/zkServer.sh status
```

当显示`Error contacting service. It is probably not running.`时, 可以查看日志

```
cat zookeeper.out
```

查看 zookeeper.out 日志可以看到是那些机器连不上, 可能是 **网络, ip, 端口, 配置文件, myid 文件** 的问题.
正常应该是: 先是一些 java 异常, 这是因为 ZooKeeper 集群启动的时候, 每个结点都试图去连接集群中的其它结点, 先启动的肯定连不上后面还没启动的, 所以上面日志前面部分的异常是可以忽略的, 当集群所有的机器的 zookeeper 都启动起来, 就没有异常了, 并选举出来了 leader.

PS: 因为 zkServer.sh 脚本中是用 nohup 命令启动 zookeeper 的, 所以 zookeeper.out 文件是在调用 zkServer.sh 时的路径下, 如:用 `bin/zkServer.sh start` 启动则 zookeeper.out 文件在 `zookeeper-3.4.9/` 下; 用 `zkServer.sh start` 启动则 zookeeper.out 文件在 `zookeeper-3.4.9/bin/` 下.

2 Storm

2.1 简介

Apache Storm: 分布式实时计算系统

与 Hadoop 的批处理相类似, Storm 可以对大量的数据流进行可靠的实时处理, 这一过程也称为"流式处理", 是分布式大数据处理的一个重要方向. Storm 支持多种类型的应用, 包括: 实时分析, 在线机器学习, 连续计算, 分布式RPC(DRPC), ETL等. Strom 的一个重要特点就是"快速"的数据处理, 有 benchmark 显示 Storm 能够达到单个节点每秒百万级 tuple 处理(Tuple 是 Storm 的最小数据单元)的速度. 快速的数据处理, 优秀的可扩展性与容错性, 便捷的可操作性与维护性, 活跃的社区技术支持, 这就是 Storm.

Storm的适用场景:

1. 流数据处理. Storm 可以用来处理源源不断流进来的消息, 处理之后将结果写入到某个存储中去.
2. 分布式 rpc. 由于 Storm 的处理组件是分布式的, 而且处理延迟极低, 所以可以作为一个通用的分布式 rpc 框架来使用.

2.2 搭建

2.2.1 单机模式

此模式主要用于 **开发人员本地环境下测试代码**

2.2.1.1 搭建 Zookeeper (单机 or 集群)

见 1.2.1 单机模式 or 见 1.2.2 集群模式

2.2.1.2 安装 Storm 依赖库(Java, Python)

在集群中的所有机器上安装 Storm 必要的依赖组件: Java7(or 8), Python2.7.

使用 CentOS 7 自带的 Python 2.7.5 及 openjdk 1.8.0_65 即可

2.2.1.3 解压 Storm 并启动 Storm 各个后台进程

不需额外配置, 即是单机模式

- **Nimbus**: 运行

```
nohup bin/storm nimbus > logs/nimbus-boot.log 2>&1 &
```

启动 Nimbus 后台程序, 并放到后台执行, 标准输出和错误输出定向到 `./logs/nimbus-boot.log`, 有问题时可以去看这个文件

- **Supervisor**: 运行

```
nohup bin/storm supervisor > logs/supervisor-boot.log 2>&1 &
```

启动 Supervisor 后台程序, 并放到后台执行, 标准输出和错误输出定向到 `./logs/supervisor-boot.log`, 有问题时可以去看这个文件

- **Storm UI**: 运行

```
nohup bin/storm ui > logs/ui-boot.log 2>&1 &
```

启动 Storm UI 后台程序, 并放到后台执行, 标准输出和错误输出定向到 `./logs/ui-boot.log`, 有问题时可以去看这个文件.

Storm UI 可以在浏览器中方便地监控集群与拓扑运行状况, 启动后可以通过 *http://{nimbus host}:8080* 观察集群的 Worker 资源使用情况, Topologies 的运行状态等信息.

PS: Storm 后台进程被启动后, 将在 Storm 安装部署目录下的 logs/ 子目录下生成各个进程的日志文件, 这是 Storm 的默认设置, 日志文件的路径与相关配置信息可以在 {STORM_HOME}/logback/cluster.xml 文件中修改.

2.2.2 集群模式

此模式是 **生产环境中实际使用的模式**

2.2.2.1 hosts 映射(可选)

最好配置主机名, 配置文件 conf/storm.yaml 中若是填写 IP, 在 Storm UI 中显示不正常

```
echo "192.168.1.4 nim1" >> /etc/hosts
echo "192.168.1.5 nim2" >> /etc/hosts

echo "192.168.1.6 sup1" >> /etc/hosts
echo "192.168.1.7 sup2" >> /etc/hosts
echo "192.168.1.8 sup3" >> /etc/hosts
echo "192.168.1.9 sup4" >> /etc/hosts
```

2.2.2.2 搭建 Zookeeper 集群

见 1.2.2 集群模式

关于 ZooKeeper 部署的两点说明:

- ZooKeeper 必须在监控模式下运行. 因为 ZooKeeper 是个快速失败系统, 如果遇到了故障, ZooKeeper 服务会主动关闭. 更多详细信息请参考: http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc_supervision. 我们选择第一个方案: daemontools: http://cr.yp.to/daemontools.html
- 需要设置一个 cron 服务来定时压缩 ZooKeeper 的数据与事务日志. 因为 ZooKeeper 的后台进程不会处理这个问题, 如果不配置 cron, ZooKeeper 的日志会很快填满磁盘空间. 更多详细信息请参考: http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc_maintenance
    - 代替方案: http://nileader.blog.51cto.com/1381108/932156 第四种

    > 在 zoo.cfg 中配置的:
    > autopurge.purgeInterval 这个参数指定了清理频率, 单位是小时, 需要填写一个 1 或更大的整数, 默认是 0, 表示不开启自己清理功能.
    > autopurge.snapRetainCount 这个参数和上面的参数搭配使用, 这个参数指定了需要保留的快照数目. 默认是保留 3 个.

2.2.2.3 安装 Storm 依赖库(Java, Python)

在集群中的所有机器上安装 Storm 必要的依赖组件: Java7(or 8), Python2.7.

使用 CentOS 7 自带的 Python 2.7.5 及 openjdk 1.8.0_65 即可

2.2.2.4 解压 Storm 并进入其根目录

```
tar xzf apache-storm-1.0.2.tar.gz -C /usr/local/
cd /usr/local/apache-storm-1.0.2
```

2.2.2.5 修改 conf/storm.yaml 配置文件

storm.yaml 会覆盖 defaults.yaml 中各个配置项的默认值, 以下几个是在安装集群时必须配置的选项

```
############ These MUST be filled in for a storm configuration
# yaml 文件的配置使用"-"来表示数据的层次结构，配置项的:后必须有空格，否则该配置项无法识别
# Storm 关联的 ZooKeeper 集群的地址列表
storm.zookeeper.servers:
    - "192.168.1.1"
    - "192.168.1.2"
    - "192.168.1.3"

# 如果使用的 ZooKeeper 集群的端口不是默认端口，还需要配置 storm.zookeeper.port
# storm.zookeeper.port: 2181

# Storm 工作目录，需要提前创建该目录并给以足够的访问权限
storm.local.dir: "/var/lib/storm-workdir"

# 用作 nimbus 的机器的 host list，若 nimbus 是单机，可以使用 nimbus.seeds: ["nim1"]，这里用的双机
# 若是填写 IP，在 Storm UI 中显示不正常
nimbus.seeds: ["nim1", "nim2"]

# Supervisor工作节点上 worker 的端口，每个 worker 占用一个单独的端口用于接收消息，有几个端口就最多会有几个 worker 运行，这里配置了 4 个
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
```

2.2.2.6 开放端口

把 Storm 用到的端口开放出来

```
firewall-cmd --zone=public --add-port=3772/tcp --permanent     # drpc.port
firewall-cmd --zone=public --add-port=3773/tcp --permanent     #
drpc.invocations.port
firewall-cmd --zone=public --add-port=3774/tcp --permanent     #
drpc.http.port
firewall-cmd --zone=public --add-port=6627/tcp --permanent     #
nimbus.thrift.port
firewall-cmd --zone=public --add-port=6699/tcp --permanent     #
pacemaker.port
firewall-cmd --zone=public --add-port=8000/tcp --permanent     #
logviewer.port
firewall-cmd --zone=public --add-port=8080/tcp --permanent     #
storm.ui.port
firewall-cmd --zone=public --add-port=6700/tcp --permanent     #
supervisor.slots.ports -- 取决于上面的配置
firewall-cmd --zone=public --add-port=6701/tcp --permanent     #
firewall-cmd --zone=public --add-port=6702/tcp --permanent     #
firewall-cmd --zone=public --add-port=6703/tcp --permanent     #
firewall-cmd --reload                                          # 重新加载防火
墙规则
```

2.2.2.7 启动 Storm 各个后台进程

和 Zookeeper 一样, Storm 也是快速失败(fail-fast)的系统, 这样 Storm 才能在任意时刻被停止, 并且当进程重启后被正确地恢复执行. 这也是为什么 Storm 不在进程内保存状态的原因, 即使 Nimbus 或 Supervisors 被重启, 运行中的 Topologies 不会受到影响. 以下是启动 Storm 各个后台进程的方式:

```
# 在 Storm 主控(Master)节点上运行 Nimbus 后台程序, 并放到后台执行, 标准输出和错误输出
定向到 `./logs/nimbus-boot.log`, 有问题时可以去看这个文件
nohup bin/storm nimbus > logs/nimbus-boot.log 2>&1 &

# 在 Storm 各个工作节点(Worker)上运行 Supervisor 后台程序, 并放到后台执行, 标准输出
和错误输出定向到 `./logs/supervisor-boot.log`, 有问题时可以去看这个文件
nohup bin/storm supervisor > logs/supervisor-boot.log 2>&1 &

# 在 Storm 主控(Master)节点上运行 Storm UI 后台程序, 并放到后台执行, 标准输出和错误输
出定向到 `./logs/ui-boot.log`, 有问题时可以去看这个文件.
nohup bin/storm ui > logs/ui-boot.log 2>&1 &

# 在需要查看 work.log 节点上运行 Logviewer 后台程序, 并放到后台执行, 标准输出和错误输
出定向到 `./logs/logviewer-boot.log`, 有问题时可以去看这个文件.
nohup bin/storm logviewer > logs/logviewer-boot.log 2>&1 &
```

Storm UI 可以在浏览器中方便地监控集群与拓扑运行状况, 启动后可以通过 *http://{nimbus host}:8080* 观察集群的 Worker 资源使用情况, Topologies 的运行状态等信息. Logviewer 是 Storm UI 中用来查看 Nimbus/Supervisor 的 log 的工具.

PS: Storm UI 必须在 Nimbus 机器(Nimbus 集群的话, 其中一台即可)上, 否则 UI 无法正常工作.

Storm 后台进程被启动后, 将在 Storm 安装部署目录下的 logs/ 子目录下生成各个进程的日志文件, 这是 Storm 的默认设置, 日志文件的路径与相关配置信息可以在 ${STORM_HOME}/logback/cluster.xml 文件中修改.

至此, Storm 集群已经部署, 配置完毕, 可以 向集群 提交拓扑 运行了.

2.2.2.8 查看 Storm 状态

利用 Storm UI, 通过 http://{nimbus host}:8080 查看集群的各种状态.

2.2.2.9 监控 Supervisor 的运行情况(可选)

Storm 提供了一种机制, 使 Supervisor 定期运行管理人员提供的脚本, 以确定节点是否正常.

管理人员可以让 Supervisor 执行位于 storm.health.check.dir 中的脚本来确定节点是否处于健康状态, 如果脚本检测到节点处于不正常状态, 则在标准输出中打印一行以 ERROR 开头的字符串.

Supervisor 将定期运行storm.health.check.dir 中的脚本并检查输出, 如果脚本的输出包含字符串 ERROR, Supervisor 将关闭所有工作线程并退出.

如果 Supervisor 正在运行, 可以调用"/bin/storm node-health-check"来确定节点是否正常.

在 conf/storm.yaml 配置 storm.health.check.dir:

```
storm.health.check.dir: " healthchecks"
```

配置执行 healthcheck 脚本的周期:

```
storm.health.check.timeout.ms: 5000
```

PS: 脚本必须具有执行权限.

2.2.2.10 配置外部库与环境变量(可选)

如果你需要使用某些外部库或者定制插件的功能, 你可以将相关 jar 包放入 extlib 与 extlib-daemon 目录下. 注意, extlib-daemon 目录仅用于存储后台进程(Nimbus, Supervisor, DRPC, UI, Logviewer)所需的 jar 包, 例如 HDFS 以及定制的调度库.

另外可以使用 STORM_EXT_CLASSPATH 和 STORM_EXT_CLASSPATH_DAEMON 两个环境变量来配置普通外部库与"仅用于后台进程"外部库的 classpath.

2.3 向 Storm 集群提交任务

- 启动 Storm Topology:

```
storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3
```

其中, allmycode.jar 是包含 Topology 实现代码的 jar 包, org.me.MyTopology 的 main 方法是 Topology 的入口, arg1, arg2 和 arg3 为 org.me.MyTopology 执行时需要传入的参数.

- 停止 Storm Topology:

```
storm kill {toponame}
```

其中, {toponame} 为 Topology 提交到 Storm 集群时指定的 Topology 任务名称.

2.4 Storm 常见配置问题

- 运行 storm 命令报错

出现语法错误:

```
File "/home/storm/apache-storm-0.9.3/bin/storm", line 61
   normclasspath = cygpath if sys.platform == 'cygwin' else identity
                           ^
SyntaxError: invalid syntax
```

这是由于系统中安装的低版本 Python 部分语法不支持, 需要重新安装高版本 Python(如2.7.x).

PS: 部分系统Python默认安装位置不是 `/usr/bin/python`, 必须在 Python 安装完成之后将安装版本 Python关联到该位置. 参考操作方法: `cd /usr/bin` `mv python python.bk``ln -s /usr/local/Python-2.7.8/python python`

- Storm 在 ssh 断开后自动关闭

这是由于 Storm 是由默认的 Shell 机制打开运行, 在 ssh 或 telnet 断开后终端会将挂断信号发送到控制进程, 进而会关闭该 Shell 进程组中的所有进程. 因此需要在 Storm 后台启动时使用 `nohup` 命令和 `&` 标记可以使进程忽略挂断信号, 避免程序的异常退出:

```
nohup bin/storm nimbus > logs/nimbus-boot.log 2>&1 &
nohup bin/storm supervisor > logs/supervisor-boot.log 2>&1 &
nohup bin/storm ui > logs/ui-boot.log 2>&1 &
nohup bin/storm logviewer > logs/logviewer-boot.log 2>&1 &
```

- Storm UI 网页无法打开

检查 Storm 主机(nimbus 与 ui 所在运行服务器)的防火墙设置, 是否存在监控端口屏蔽(ui 的默认端口是 8080)

PS: 测试环境下可以不考虑安全问题直接关闭防火墙

- Strom UI 网页中没有 topology 信息

只有集群(Cluster)模式的 topology 才会在监控页面显示, 需要将提交到集群的 topology 的运行模式由本地模式(local mode)改为集群模式

- Storm UI 网页中无法打开各个端口的 worker.log

在需要查看 log 的机器上启动 logviewer 进程:

```
nohup bin/storm logviewer > logs/logviewer-boot.log 2>&1 &
```

- expected '', but found BlockMappingStart 错误

Storm 启动失败, 在 nohup.out 中有如下错误信息

```
Exception in thread "main" expected '<document start>', but found
BlockMappingStart
```

一般在这类信息后会有相关错误位置说明信息, 如

```
in 'reader', line 23, column 2:
     nimbus.host: "hd124"
     ^
```

或者

```
in 'reader', line 7, column 1:
     storm.zookeeper.port: 2181
     ^
```

这类错误主要是storm.yaml文件的配置格式错误造成的, 一般是配置项的空格遗漏问题. 如上面两例分别表示nimbus.host与storm.zookeeper.port两个配置项开头缺少空格, 或者":"后缺少空格. 正确添加空格后重新启动Storm即可.

- Storm worker 数量与配置数量不一致

在 topology 中设置 worker 数量:

`conf.setNumWorkers(6);`

但是, 集群中实际的 worker 数量却不到6.

这是由于每个 supervisor 中有 worker 数量的上限, 这个上限值除了要满足系统允许的最大 slot 上限值 `8` 之外, 还需要小于 Storm 配置文件中的端口数量:

```
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
```

例如这里 supervisor 只配置了 4 个端口, 那么在这个 supervisor 上最多只能运行 4 个 worker 进程. 因此, 如果需要更多的 worker 就需要配置更多的端口.

- 日志无法记录到程序中配置的路径

Storm 默认将日志统一记录到 `$STORM_HOME/logs` 目录中, 不支持在程序中自定义的路径. 但是, 集群的日志记录目录是可以修改的, 0.9 以上版本的 Storm 可以
在 `$STORM_HOME/logback/cluster.xml` 配置文件中修改, 其他早期版本可以
在`log4j/*.properties` 配置文件中修改.

3 Kafka

3.1 简介

Apache Kafka是一个分布式消息发布订阅系统. Kafka 系统快速, 可扩展并且可持久化. 它的分区特性, 可复制和可容错都是其不错的特性.

3.2 搭建

3.2.1 hosts 映射(可选, 建议)

```
echo "192.168.1.10 kfk1" >> /etc/hosts
echo "192.168.1.11 kfk2" >> /etc/hosts
echo "192.168.1.12 kfk3" >> /etc/hosts
```

3.2.2 搭建 Zookeeper (单机 or 集群)

Broker, Producer, Consumer 的运行都需要 ZooKeeper

见 1.2.1 单机模式 or 见 1.2.2 集群模式

3.2.3 Broker 的配置

```
tar xzf kafka_2.11-0.9.0.1.tgz -C /usr/local/
cd /usr/local/kafka_2.11-0.9.0.1
```

config 文件夹下是各个组件的配置文件, server.properties 是 Broker 的配置文件, 需要修改的有

```
########################## Server Basics ##########################
broker.id=0                    # 本 Broker 的 id, 只要非负数且各 Broker 的 id
不同即可, 一般依次加 1

#################### Socket Server Settings ####################
listeners=PLAINTEXT://:9092    # Broker 监听的端口, Producer, Consumer 会连接这
个端口
port=9092                      # 同上
host.name=kfk1                 # 本 Broker 的 hostname

########################### Topic Basics ###########################
delete.topic.enable=true       # 配置为可以使用 delete topic 命令

########################### Log Basics ###########################
log.dirs=/var/lib/kafka-logs   # log 目录, 此目录要存在且有足够权限

###################### Log Retention Policy ######################
log.roll.hours=2               # 开始一个新的 log 文件片段的最大时间
log.retention.hours=24         # 控制一个 log 文件保留多长个小时
log.retention.bytes=1073741824 # 所有 log 文件的最大大小
log.segment.bytes=104857600    # 单一的 log 文件最大大小
log.cleanup.policy=delete      # log 清除策略
log.retention.check.interval.ms=60000

############################ Zookeeper ############################
zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181 # Zookeeper 的连接信息
```

注意: broker.id 和 host.name 在每台机器上是不一样的, 要按实际填写

即在 kfk2, kfk3 上

```
broker.id=1
host.name=kfk2
```

```
broker.id=2
host.name=kfk3
```

PS: 在 kafka 安装目录下的 `./site-docs` 目录下有 `kafka_config.html`, `producer_config.html`, `consumer_config.html` 三个文件, 分别讲解 broker, producer, consumer 配置参数含义. 附录 A 是其翻译.

3.2.4 开放端口

把 Kafka 用到的端口开放出来

```
firewall-cmd --zone=public --add-port=9000/tcp --permanent      # 永久开启
9000 端口(kafka manager)
firewall-cmd --zone=public --add-port=9092/tcp --permanent      # 永久开启
9092 端口(brokers)
firewall-cmd --zone=public --add-port=9999/tcp --permanent      # 永久开启
9999 端口(JMX)
firewall-cmd --reload                                           # 重新加载防火
墙规则
```

3.2.5 Broker 运行与终止

Broker 运行与终止命令如下.

```
# 运行: 如下有两种方式
# 以守护进程的方式启动(推荐)
bin/kafka-server-start.sh -daemon config/server.properties
# 将 Broker 放到后台执行, 且不受终端关闭的影响, 标准输出和错误输出定向到
`./logs/kafka-server-boot.log`, 有问题时可以去看这个文件
nohup bin/kafka-server-start.sh config/server.properties > logs/kafka-
server-boot.log 2>&1 &

# 终止
bin/kafka-server-stop.sh config/server.properties
```

3.2.6 测试

我们使用 Kafka 自带的基于 终端 的 Producer 和 Consumer 脚本做测试.

先只启动一台机器上的 Broker. 在 kfk1 上运行

```
nohup bin/kafka-server-start.sh config/server.properties > logs/kafka-
server-boot.log 2>&1 &
```

1. 创建 Topic

创建一个 名为"TestCase"的 单分区 单副本 的 Topic.

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic TestCase --
replication-factor 1 --partitions 1
```

查看有哪些 Topic:

```
$ bin/kafka-topics.sh --list --zookeeper localhost:2181
TestCase
```

运行`describe topics`命令, 可以知道 Topic 的具体分配:

```
$ bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
```

解释一下输出的内容. 第一行给出了所有 partition 的一个摘要, 每行给出一个 partition 的信息. 因为我们这个 topic 只有一个 partition 所以只有一行信息.

- "leader" 负责所有 partition 的读和写请求的响应. "leader" 是随机选定的.
- "replicas" 是备份节点列表, 包含所有复制了此 partition log 的节点, 不管这个节点是否为 leader 也不管这个节点当前是否存活, 只是显示.
- "isr" 是当前处于同步状态的备份节点列表. 即 "replicas" 列表中处于存活状态并且与 leader 一致的节点.

可以发现这个 Topic 没有副本而且它在 [我们创建它时集群仅有的一个节点] Broker 0 上.

另外, 除去手工创建 Topic 以外, 你也可以将你的 Brokers 配置成当消息发布到一个不存在的 Topic 时自动创建此 Topic.

2. 启动 生产者

Kafka 附带一个 **终端生产者** 可以从文件或者标准输入中读取输入然后发送这个消息到 Kafka 集群. 默认情况下每行信息被当做一个消息发送.

运行生产者脚本然后在终端中输入一些消息, 即可发送到 Broker.

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic TestCase
This is a message
This is another message
```

PS: 通过键入 **Ctrl-C** 来终止终端生产者.

3. 启动 消费者

Kafka 也附带了一个 **终端生产者** 可以导出这些消息到标准输出.

```
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic TestCase --
-from-beginning
This is a message
This is another message
```

`--from-beginning` 参数使得可以接收到 topic 的所有消息, 包括 consumer 启动前的. 去掉后则为仅接收 consumer 启动后 kafka 收到的消息.

如果你在不同的终端运行生产者和消费者这两个命令, 那么现在你就应该能再生产者的终端中键入消息同时在消费者的终端中看到.

所有的命令行工具都有很多可选的参数; 不添加参数直接执行这些命令将会显示它们的使用方法, 更多内容可以参考他们的手册.

PS: 通过键入 **Ctrl-C** 来终止终端消费者.

4. 配置一个多节点集群

我们已经成功的以单 Broker 的模式运行起来了, 但这并没有意思. 对于 Kafka 来说, 一个单独的 Broker 就是一个大小为 1 的集群, 所以集群模式就是多启动几个 Broker 实例.

我们将我们的集群扩展到3个节点. 在另外两台机器 kfk2, kfk3 上运行

```
nohup bin/kafka-server-start.sh config/server.properties > logs/kafka-
server-boot.log 2>&1 &
```

现在我们可以创建一个新的 Topic 并制定副本数量为 3:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic my-
replicated-topic --replication-factor 3 --partitions 1
```

运行`describe topics`命令, 可以知道每个 Broker 具体的工作:

```
$ bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-
replicated-topic
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3
Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 1    Replicas:
1,2,0    Isr: 1,2,0
```

注意本例中 Broker 1 是这个有一个 partition 的 topic 的 leader.

现在我们发布几个消息到我们的新 topic 上:

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-
replicated-topic
my test message 1
my test message 2
```

现在让我们消费这几个消息:

```
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic my-
replicated-topic --from-beginning
my test message 1
my test message 2
```

现在让我们测试一下集群容错. Broker 1 正在作为 leader, 所以我们杀掉它:

```
$ ps | grep server.properties
...
7564 ttys002    0:15.91
/System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...
...

$ kill -9 7564
```

或在 kfk1 机器上运行

```
bin/kafka-server-stop.sh config/server.properties
```

此时, 集群领导已经切换到一个从服务器上, Broker 1 节点也不在出现在同步副本列表中了:

```
$ bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-
replicated-topic
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3
Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 2    Replicas:
1,2,0    Isr: 2,0
```

而且现在消息的消费仍然能正常进行, 即使原来负责写的节点已经失效了.

```
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning -
-topic my-replicated-topic
...
my test message 1
my test message 2
```

5. 使用 Kafka Connect 进行数据导入导出

从终端写入数据, 数据也写回终端是默认的. 但是你可能希望从一些其它的数据源或者导出 Kafka 的数据到其它的系统. 相比其它系统需要自己编写集成代码, 你可以直接使用Kafka的 Connect 直接导入或者导出数据. Kafka Connect 是 Kafka 自带的用于数据导入和导出的工具. 它是一个扩展的可运行连接器(runs*connectors*)工具, 可实现自定义的逻辑来实现与外部系统的集成交互. 在这个快速入门中我们将介绍如何通过一个简单的从文本导入数据, 导出数据到文本的连接器来调用 Kafka Connect. 首先我们从创建一些测试的基础数据开始:

```
echo -e "foo\nbar" > test.txt
```

接下来我们采用*standalone*模式启动两个 connectors, 也就是让它们都运行在独立的, 本地的, 不同的进程中. 我们提供三个参数化的配置文件, 第一个提供共有的配置用于 Kafka Connect 处理, 包含共有的配置比如连接哪个 Kafka broker 和数据的序列化格式. 剩下的配置文件制定每个 connector 创建的特定信息. 这些文件包括唯一的 connector 的名字, connector 要实例化的类和其它的一些 connector 必备的配置.

```
bin/connect-standalone.sh config/connect-standalone.properties
config/connect-file-source.properties config/connect-file-sink.properties
```

上述简单的配置文件已经被包含在 Kafka 的发行包中, 它们将使用默认的之前我们启动的本地集群配置创建两个 connector: 第一个作为源 connector 从一个文件中读取每行数据然后将他们发送 Kafka 的 topic, 第二个是一个输出(sink)connector 从 Kafka 的 topic 读取消息, 然后将它们输出成输出文件的一行行的数据. 在启动的过程你讲看到一些日志消息, 包括一些提示 connector 正在被实例化的信息. 一旦 Kafka Connect 进程启动以后, 源 connector 应该开始从 `test.txt` 中读取数据行, 并将他们发送到 topic `connect-test` 上, 然后输出 connector 将会开始从 topic 读取消息然后把它们写入到 `test.sink.txt` 中.

我们可以查看输出文件来验证通过整个管线投递的数据:

```
$ cat test.sink.txt
foo
bar
```

注意这些数据已经被保存到了 Kafka 的 `connect-test` topic 中, 所以我们还可以运行一个终端消费者来看到这些数据(或者使用自定义的消费者代码来处理数据):

```
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic connect-
test --from-beginning
{"schema":{"type":"string","optional":false},"payload":"foo"}
{"schema":{"type":"string","optional":false},"payload":"bar"}
...
```

connector 在持续的处理着数据, 所以我们可以向文件中添加数据然后观察到它在这个管线中的传递:

```
echo "Another line" >> test.txt
```

你应该可以观察到新的数据行出现在终端消费者中和输出文件中.

6. 使用 Kafka Streams 来处理数据

Kafka Streams 是一个用来对 Kafka brokers 中保存的数据进行实时处理和分析的客户端库. 这个入门示例将演示如何启动一个采用此类库实现的流处理程序. 下面是 `WordCountDemo` 示例代码的 GIST(为了方便阅读已经转化成了 Java 8 的 lambda 表达式).

```
KTable wordCounts = textLines
    // 按照空格将每个文本行拆分成单词
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // 确保每个单词作为记录的 key 值以便于下一步的聚合
    .map((key, value) -> new KeyValue<>(value, value))
    // 计算每个单词的出现频率并将他们保存到 "Counts" 的表中
    .countByKey("Counts")
```

上述代码实现了计算每个单词出现频率直方图的单词计数算法. 但是它与之前常见的操作有限数据的示例相比有明显的不同, 它被设计成一个操作 **无边界限制的流数据** 的程序. 与有界算法相似它是一个有状态算法, 它可以跟踪并更新单词的计数. 但是它必须支持处理无边界限制的数据输入的假设, 它将在处理数据的过程持续的输出自身的状态和结果, 因为它不能明确的知道合适已经完成了所有输入数据的处理.

接下来我们准备一些发送到 Kafka topic 的输入数据, 随后它们将被 Kafka Streams 程序处理.

```
echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" >
file-input.txt
```

接下来我们使用终端生产者发送这些输入数据到名为 **streams-file-input** 的输入 topic (在实际应用中, 流数据会是不断流入处理程序启动和运行用的Kafka):

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor
1 --partitions 1 --topic streams-file-input
```

```
cat file-input.txt | bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic streams-file-input
```

现在我们可以启动WordCount示例程序来处理这些数据了:

```
bin/kafka-run-class.sh
org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

在 STDOUT 终端不会有任何日志输出, 因为所有的结果被不断的写回了另外一个名为 **streams-wordcount-output** 的 topic 上. 这个实例将会运行一会儿, 之后与典型的流处理程序不同它将会自动退出.

现在我们可以通过读取这个单词计数示例程序的输出 topic 来验证结果:

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic streams-
wordcount-output --from-beginning \
            --formatter kafka.tools.DefaultMessageFormatter \
            --property print.key=true \
            --property print.value=true \
            --property
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
            --property
value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

以下输出数据将会被打印到终端上:

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
join    1
kafka   3
summit  1
```

可以看到, 第一列是 Kafka 的消息的健, 第二列是这个消息的值, 他们都是 `java.lang.String` 格式的. 注意这个输出结果实际上是一个持续更新的流, 每一行(例如, 上述原始输出的每一行)是一个单词更新之后的计数. 对于 key 相同的多行记录, 每行都是前面一行的更新.

现在你可以向 **streams-file-input** topic写入更多的消息并观察 **streams-wordcount-output** topic 表述更新单词计数的新的消息.

4 Redis

Redis 是一个开源的可基于内存亦可持久化的日志型、Key-Value 数据库, 它通常被称为数据结构服务器, 因为值(value)可以是字符串(String), 哈希(Map), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型, 各式各样的问题都可以很自然地映射到这些数据结构上. 用户可以很方便地将 Redis 扩展成一个能够包含数百 GB 数据、每秒钟处理上百万次请求的系统

4.1 Redis 安装

下载地址: http://redis.io/download

本教程使用的最新文档版本为 3.2.5, 下载并安装:

```
$ wget http://download.redis.io/releases/redis-3.2.5.tar.gz
$ tar xzf redis-3.2.5.tar.gz
$ cd redis-3.2.5
$ make
```

make 完, redis-3.2.5/src 目录下会出现编译后的 redis 服务程序 redis-server, 还有客户端程序 redis-cli

下面启动 redis 服务.

```
$ ./src/redis-server
```

注意这种方式启动 redis 使用的是默认配置. 也可以通过启动参数告诉 redis 使用指定配置文件使用下面命令启动.

```
$ ./src/redis-server ./redis.conf
```

redis.conf 是一个默认的配置文件. 我们可以根据需要使用自己的配置文件.

启动 redis 服务进程后, 就可以使用客户端程序 redis-cli 和 redis server 交互了. 比如:

```
$ ./src/redis-cli    # or ./src/redis-cli -h 127.0.0.1 -p 6379
redis 127.0.0.1:6379> ping
PONG
```

4.2 Redis 配置

Redis 的配置文件位于 Redis 安装目录下, 文件名为 redis.conf.

你可以通过 **CONFIG** 命令查看或设置配置项.

4.2.1 查看方法

CONFIG GET 命令基本语法:

```
redis 127.0.0.1:6379> CONFIG GET CONFIG_SETTING_NAME
```

例如

```
redis 127.0.0.1:6379> CONFIG GET loglevel
1) "loglevel"
2) "notice"
```

使用 `*` 号获取所有配置项:

```
redis 127.0.0.1:6379> CONFIG GET *

  1) "dbfilename"
  2) "dump.rdb"
  3) "requirepass"
  4) ""
  5) "masterauth"
  6) ""
  ...
```

4.2.2 修改方法

你可以通过修改 redis.conf 文件或使用 **CONFIG set** 命令来修改配置.

CONFIG SET 命令基本语法:

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VALUE
```

例如

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
OK
redis 127.0.0.1:6379> CONFIG GET loglevel
1) "loglevel"
2) "notice"
```

4.2.3 常用配置

```
daemonize yes
bind 0.0.0.0
port 6379
requirepass yourpassword
```

4.2.4 附: 参数说明

redis.conf 配置项说明如下:

1. Redis 默认不是以守护进程的方式运行, 可以通过该配置项修改, 使用 yes 启用守护进程
   daemonize no
2. 当 Redis 以守护进程方式运行时, Redis 默认会把 pid 写入 /var/run/redis.pid 文件, 可以通过 pidfile 指定
   pidfile /var/run/redis.pid
3. 绑定用于接受 redis 访问请求的本机 IP 地址, 可绑定多个, 用空格隔开. 即如果你的 redis 服务器有两张网卡, 一张是 ip-1, 另一张是 ip-2, 如果你 bind ip-1, 那么只有对 ip-1 的请求会被受理.
   bind 127.0.0.1 则只能本机访问
   bind 0.0.0.0 则只有本子网内的机器可以访问
   注释掉 `# bind 127.0.0.1` 则接受所有访问
4. 指定 Redis 监听端口, 默认端口为 6379, 作者在自己的一篇博文中解释了为什么选用 6379 作为默认端口, 因为 6379 在手机按键上 MERZ 对应的号码, 而 MERZ 取自意大利歌女 Alessia Merz 的名字
   port 6379
5. 设置 Redis 连接密码, 如果配置了连接密码, 客户端在连接Redis时需要通过 `AUTH <password>` 命令提供密码, 默认关闭
   requirepass yourpassword
6. 当 客户端闲置多长时间后关闭连接, 如果指定为0, 表示关闭该功能
   timeout 300
7. 指定日志记录级别, Redis 总共支持四个级别: debug, verbose, notice, warning, 默认为 verbose
   loglevel verbose
8. 日志记录方式, 默认为标准输出, 如果配置 Redis 为守护进程方式运行, 而这里又配置为日志记录方式为标准输出, 则日志将会发送给 /dev/null
   logfile stdout
9. 设置数据库的数量, 默认数据库为 0, 可以使用 SELECT 命令在连接上指定数据库 id
   databases 16
10. 指定在多长时间内, 有多少次更新操作, 就将数据同步到数据文件, 可以多个条件配合
    save
    Redis默认配置文件中提供了三个条件:
    save 900 1

save 300 10
save 60 10000
分别表示 900 秒(15 分钟)内有 1 个更改, 300 秒(5 分钟)内有 10 个更改以及 60 秒内有 10000 个更改.

11. 指定存储至本地数据库时是否压缩数据, 默认为 yes, Redis 采用 LZF 压缩, 如果为了节省 CPU 时间, 可以关闭该选项, 但会导致数据库文件变的巨大
    rdbcompression yes
12. 指定本地数据库文件名, 默认值为 dump.rdb
    dbfilename dump.rdb
13. 指定本地数据库存放目录
    dir ./
14. 设置当本机为 slave 服务时, 设置 master 服务的 IP 地址及端口, 在 Redis 启动时, 它会自动从 master 进行数据同步
    slaveof
15. 当 master 服务设置了密码保护时, slave 服务连接 master 的密码
    masterauth
16. 设置同一时间最大客户端连接数, 默认无限制, Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数, 如果设置 maxclients 0, 表示不作限制. 当客户端连接数到达限制时, Redis 会关闭新的连接并向客户端返回 "max number of clients reached" 错误信息
    maxclients 128
17. 指定 Redis 最大内存限制, Redis 在启动时会把数据加载到内存中, 达到最大内存后, Redis 会先尝试清除已到期或即将到期的 Key, 当此方法处理后, 仍然到达最大内存设置, 将无法再进行写入操作, 但仍然可以进行读取操作. Redis 新的 vm 机制, 会把 Key 存放内存, Value 会存放在 swap 区
    maxmemory
18. 指定是否在每次更新操作后进行日志记录, Redis 在默认情况下是异步的把数据写入磁盘, 如果不开启, 可能会在断电时导致一段时间内的数据丢失. 因为 redis 本身同步数据文件是按上面 save 条件来同步的, 所以有的数据会在一段时间内只存在于内存中. 默认为 no
    appendonly no
19. 指定更新日志文件名, 默认为 appendonly.aof
    appendfilename appendonly.aof
20. 指定更新日志条件, 共有 3 个可选值:
    no: 表示等操作系统进行数据缓存同步到磁盘(快)
    always: 表示每次更新操作后手动调用 fsync() 将数据写到磁盘(慢, 安全)
    everysec: 表示每秒同步一次(折衷, 默认值)
    appendfsync everysec
21. 指定是否启用虚拟内存机制, 默认值为 no, 简单的介绍一下, VM 机制将数据分页存放, 由 Redis 将访问量较少的页即冷数据 swap 到磁盘上, 访问多的页面由磁盘自动换出到内存中(在后面的文章我会仔细分析 Redis 的 VM 机制)
    vm-enabled no
22. 虚拟内存文件路径, 默认值为 /tmp/redis.swap, 不可多个 Redis 实例共享
    vm-swap-file /tmp/redis.swap
23. 将所有大于 vm-max-memory 的数据存入虚拟内存, 无论 vm-max-memory 设置多小, 所有索引数据都是内存存储的(Redis 的索引数据就是 keys), 也就是说, 当 vm-max-memory 设置为 0 的时候, 其实是所有 value 都存在于磁盘. 默认值为 0
    vm-max-memory 0
24. Redis swap 文件分成了很多的 page, 一个对象可以保存在多个 page 上面, 但一个 page 上不能被多个对象共享, vm-page-size 是要根据存储的数据大小来设定的, 作者建议如果存储很多小对

象, page 大小最好设置为 32 或者 64 bytes; 如果存储很大大对象, 则可以使用更大的 page, 如果不确定, 就使用默认值
vm-page-size 32

25. 设置 swap 文件中的 page 数量, 由于页表(一种表示页面空闲或使用的 bitmap)是在放在内存中的, 在磁盘上每 8 个 pages 将消耗 1 byte 的内存.
vm-pages 134217728
26. 设置访问 swap 文件的线程数,最好不要超过机器的核数,如果设置为 0,那么所有对 swap 文件的操作都是串行的, 可能会造成比较长时间的延迟. 默认值为 4
vm-max-threads 4
27. 设置在向客户端应答时, 是否把较小的包合并为一个包发送, 默认为开启
glueoutputbuf yes
28. 指定在超过一定的数量或者最大的元素超过某一临界值时, 采用一种特殊的哈希算法
hash-max-zipmap-entries 64
hash-max-zipmap-value 512
29. 指定是否激活重置哈希, 默认为开启(后面在介绍 Redis 的哈希算法时具体介绍)
activerehashing yes
30. 指定包含其它的配置文件, 可以在同一主机上多个 Redis 实例之间使用同一份配置文件, 而同时各个实例又拥有自己的特定配置文件
include /path/to/local.conf

4.3 集群(即分布式)

redis 从 3.0 以后的版本开始支持集群功能, 也就是正真意义上实现了分布式. Redis 集群是一个分布式(distributed), 容错(fault-tolerant)的 Redis 实现, 集群的其中一个主要设计目标是达到线性可扩展性(linear scalability). 目前在数据的一致性和容错性上尚不稳定, 所以暂时没有配置集群.

PS: 在单机上运行 Redis 最大的需求是内存.

5 Node

简单的说 Node.js 就是运行在服务端的 JavaScript, Node.js 是一个基于 Chrome JavaScript V8引擎建立的一个平台.

5.1 Node 安装

下载地址: https://nodejs.org/en/download/

- 下载 Node LTS(长期支持版) linux 平台预编译版本 node-v6.9.1-linux-x64.tar.xz
- 解压: `tar xJf node-v6.9.1-linux-x64.tar.xz`
- 配置环境变量到 /etc/profile (or ~/.bashrc)

```
export NODE_HOME=/usr/local/node-v6.9.1-linux-x64
export PATH=$NODE_HOME/bin:$PATH
```

- `source /etc/profile  # or source ~/.bashrc` 使环境变量生效
- 验证:

```
$ node -v
v6.9.1
```

Node 部署就完成了.

6 最后

- 因为 storm kafka zookeeper 都是 Java 的, 所以可以调整 Java 堆大小以优化上面这些程序的运行. Java 堆太小会导致程序难以运行; Java 堆太大(超出物理内存)会导致程序被交换到磁盘, 性能急剧降低. 例如: 4 G 内存的专用服务器可以分配 3 G 的 Java 堆, 最好的建议是运行负载测试, 然后确保远低于会导致系统交换的堆大小.

附录 A kafka 部分配置参数说明

boker 部分参数说明 (配置文件位于 config/server.properties)

名称	描述
zookeeper.connect	zookeeper host string
advertised.host.name	过时的: 当advertised.listeners或listeners没设置时候才使用.请改用advertised.listeners.Hostname发布到Zookeeper供客户端使用.在IaaS环境中, 这可能需要不同于broker绑定的接口.如果没有设置, 将会使用host.name(如果配置了).否则将从java.net.InetAddress.getCanonicalHostName()获取.
advertised.listeners	发布到Zookeeper供客户端使用监听.如果不同于上面的监听.在IaaS环境中, 可能需要不同于broker绑定的接口.如果没设置, 则使用listeners.
advertised.port	过时的: 当advertised.listeners或listeners没有设置才使用.请改用advertised.listeners.端口发布到Zookeeper供客户端使用, 在IaaS环境中, 可能需要不同于broker绑定的端口.如果没有设置, 将发布broker绑定的同一个端口.
auto.create.topics.enable	启用自动创建topic
auto.leader.rebalance.enable	启用自动平衡leader.如果需要, 后台线程会定期检查并触发leader平衡.
background.threads	用于各种后台处理任务的线程数
broker.id	服务器的broker id.如果未设置, 将生成一个独一无二的broker id.要避免zookeeper生成的broker id和用户配置的broker id冲突, 从

	reserved.broker.max.id + 1开始生成.
compression.type	为给定topic指定最终的压缩类型.支持标准的压缩编码器('gzip', 'snappy', 'lz4').也接受'未压缩', 就是没有压缩.保留由producer设置的原始的压缩编码.
delete.topic.enable	启用删除topic.如果此配置已关闭, 通过管理工具删除topic将没有任何效果
host.name	不赞成: 当listeners没有设置才会使用.请改用listeners.broker的hostname.如果设置, 它将只绑定到此地址.如果没有设置, 它将绑定到所有接口
leader.imbalance.check.interval.seconds	由控制器触发分区再平衡检查的频率
leader.imbalance.per.broker.percentage	允许每个broker的leader比例不平衡.如果每个broker的值高于此值, 控制器将触发leader平衡, 该值以百分比的形式指定.
listeners	监听列表 - 监听逗号分隔的URL列表和协议.指定hostname为0.0.0.0绑定到所有接口, 将hostname留空则绑定到默认接口.合法的listener列表是: PLAINTEXT://myhost:9092,TRACE://:9091 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093
log.dir	保存日志数据的目录 (补充log.dirs属性)
log.dirs	保存日志数据的目录.如果未设置, 则使用log.dir中的值
log.flush.interval.messages	消息刷新到磁盘之前, 累计在日志分区的消息数
log.flush.interval.ms	topic中的消息在刷新到磁盘之前保存在内存中的最大时间(以毫秒为单位), 如果未设置, 则使用log.flush.scheduler.interval.ms中的值
log.flush.offset.checkpoint.interval.ms	我们更新的持续记录的最后一次刷新的频率.作为日志的恢复点.
log.flush.scheduler.interval.ms	日志刷新的频率(以毫秒为单位)检查是否有任何日志需要刷新到磁盘
log.retention.bytes	删除日志之前的最大大小
log.retention.hours	删除日志文件保留的小时数(以小时为单位).第三级是log.retention.ms属性
	删除日志文件之前保留的分钟数(以分钟为单位).

log.retention.minutes	次于log.retention.ms属性.如果没设置, 则使用log.retention.hours的值.
log.retention.ms	删除日志文件之前保留的毫秒数(以毫秒为单位), 如果未设置, 则使用log.retention.minutes的值.
log.roll.hours	新建一个日志段的最大时间(以小时为单位), 次于log.roll.ms属性.
log.roll.jitter.hours	从logRollTimeMillis(以小时为单位)减去最大抖动, 次于log.roll.jitter.ms属性.
log.roll.ms	新建一个日志段之前的最大事时间(以毫秒为单位).如果未设置, 则使用log.roll.hours的值.
og.segment.bytes	单个日志文件的最大大小
og.segment.delete.delay.ms	从文件系统中删除文件之前的等待的时间
message.max.bytes	服务器可以接收的消息的最大大小
min.insync.replicas	当producer设置acks为"all"(或"-1")时.min.insync.replicas指定必须应答成功写入的replicas最小数.如果不能满足最小值, 那么producer抛出一个异常(NotEnoughReplicas或NotEnoughReplicasAfterAppend).当一起使用时, min.insync.replicas和acks提供最大的耐用性保证.一个典型的场景是创建一个复制因子3的topic, 设置min.insync.replicas为2, 并且ack是“all”.如果多数副本没有接到写入时, 将会抛出一个异常.
num.io.threads	服务器用于执行网络请求的io线程数
num.network.threads	服务器用于处理网络请求的线程数.
num.recovery.threads.per.data.dir	每个数据目录的线程数, 当重启和冲洗停机时用于日志恢复.
num.replica.fetchers	从源broker复制消息的提取线程数.递增该值可提高follower broker的I/O的并发.
offset.metadata.max.bytes	与offset提交关联的元数据条目的最大大小
offsets.commit.required.acks	提交可能已接收之前需要acks, 通常, 不应覆盖默认的(-1)
offsets.commit.timeout.ms	Offset提交延迟, 直到所有副本都收到提交或超时. 这类似于生产者请求超时.
offsets.load.buffer.size	当加载offset到缓存时, 从offset段读取的批量大小.

offsets.retention.check.interval.ms	检查旧offset的频率.
offsets.retention.minutes	offset topic的日志保留时间(分钟)
offsets.topic.compression.codec	压缩编解码器的offset topic - 压缩可以用于实现“原子”提交
offsets.topic.num.partitions	offset commit topic的分区数(部署之后不应更改)
offsets.topic.replication.factor	offset topic复制(设置越高以确保可用性).为了确保offset topic有效的复制因子, 第一次请求offset topic时, 活的broker的数量必须最少最少是配置的复制因子数. 如果不是, offset topic将创建失败或获取最小的复制因子(活着的broker, 复制因子的配置)
offsets.topic.segment.bytes	offset topic段字节应该相对较小一点, 以便于加快日志压缩和缓存加载
port	不赞成: 当listener没有设置才使用.请改用listeners.该port监听和接收连接.
queued.max.requests	在阻塞网络线程之前允许的排队请求数
quota.consumer.default	过时的: 当默认动态的quotas没有配置或在Zookeeper时.如果每秒获取的字节比此值高, 所有消费者将通过clientId/consumer区分限流.
quota.producer.default	过时的: 当默认动态的quotas没有配置, 或在zookeeper时.如果生产者每秒比此值高, 所有生产者将通过clientId区分限流.
replica.fetch.min.bytes Minimum	每个获取响应的字节数.如果没有满足字节数, 等待replicaMaxWaitTimeMs.
replica.fetch.wait.max.ms	跟随者副本发出每个获取请求的最大等待时间, 此值应始终小于replica.lag.time.max.ms, 以防止低吞吐的topic的ISR频繁的收缩.
replica.high.watermark.checkpoint.interval.ms	达到高“水位”保存到磁盘的频率.
replica.lag.time.max.ms	如果一个追随者没有发送任何获取请求或至少在这个时间的这个leader的没有消费完.该leader将从isr中移除这个追随者.
replica.socket.receive.buffer.bytes	用于网络请求的socket接收缓存区
replica.socket.timeout.ms	网络请求的socket超时, 该值最少是replica.fetch.wait.max.ms
	该配置控制客户端等待请求的响应的最大时间, .

request.timeout.ms	如果超过时间还没收到消费.客户端将重新发送请求, 如果重试次数耗尽, 则请求失败.
socket.receive.buffer.bytes	socket服务的SO_RCVBUF缓冲区.如果是-1, 则默认使用OS的.
socket.request.max.bytes	socket请求的最大字节数
socket.send.buffer.bytes	socket服务的SO_SNDBUF缓冲区.如果是-1, 则默认使用OS的.
unclean.leader.election.enable	是否启用不在ISR中的副本参与选举leader的最后的手段.这样做有可能丢失数据.
zookeeper.connection.timeout.ms	连接zookeeper的最大等待时间, 如果未设置, 则使用zookeeper.session.timeout.ms.
zookeeper.session.timeout.ms	Zookeeper会话的超时时间
zookeeper.set.acl	设置客户端使用安全的ACL
broker.id.generation.enable	启用自动生成broker id.启用该配置时应检查reserved.broker.max.id.
broker.rack	broker机架, 用于机架感知副本分配的失败容错. 例如: RACK1, us-east-1d
connections.max.idle.ms Idle	连接超时: 闲置时间超过该设置, 则服务器socket处理线程关闭这个连接.
controlled.shutdown.enable	启用服务器的关闭控制.
controlled.shutdown.max.retries	控制因多种原因导致的shutdown失败, 当这样失败发生, 尝试重试的次数
controlled.shutdown.retry.backoff.ms	在每次重试之前, 系统需要时间从导致先前故障的状态(控制器故障转移, 复制延迟等)恢复. 此配置是重试之前等待的时间数.
controller.socket.timeout.ms	控制器到broker通道的sockt超时时间
default.replication.factor	自动topic的默认的副本数
fetch.purgatory.purge.interval.requests	拉取请求清洗间隔(请求数)
group.max.session.timeout.ms	已注册的消费者允许的最大会话超时时间, 设置的时候越长使消费者有更多时间去处理心跳之间的消息.但察觉故障的时间也拉长了.
group.min.session.timeout.ms	已经注册的消费者允许最小的会话超时时间, 更短的时间去快速的察觉到故障, 代价是频繁的心跳, 这可能会占用大量的broker资源.

inter.broker.protocol.version	指定broker内部通讯使用的版本.通常是所有的broker更新到新的版本之后出现.有效的值为: 0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2.1, 0.9.0.0, 0.9.0.1.查看ApiVersion找到的全部列表.
log.cleaner.backoff.ms	当没有日志要清理时, 休眠的时间
log.cleaner.dedupe.buffer.size	所有消除线程的删除重复数据使用的内存总量.
log.cleaner.delete.retention.ms	删除的数据保留多长时间?
log.cleaner.enable	在服务器上启用日志清洗处理? 如果使用的任何topic的cleanup.policy=compact包含内部的offset topic, 应启动.如果禁用, 那些topic将不会被压缩并且会不断的增大.
log.cleaner.io.buffer.load.factor	日志清除重复数据删除缓冲去的负载因子.重复数据删除缓冲区已满的百分比.较高的值将允许同时清除更多的日志, 但将会导致更多的hash冲突.
log.cleaner.io.buffer.size	所有日志清洁器线程I/O缓存的总内存
log.cleaner.io.max.bytes.per.second	日志清理器限制, 以便其读写i/o平均小与此值.
log.cleaner.min.cleanable.ratio	脏日志与日志的总量的最小比率, 以符合清理条件

producer参数说明(配置文件位于config/producer.properties或者在程序内定义)

```
    # 指定kafka节点列表, 用于获取metadata, 不必全部指定
    metadata.broker.list=192.168.2.105:9092,192.168.2.106:9092

    # 指定分区处理类.默认kafka.producer.DefaultPartitioner, 表通过key哈希到对应分区
    #partitioner.class=com.meituan.mafka.client.producer.CustomizePartitioner

    # 是否压缩, 默认0表示不压缩, 1表示用gzip压缩, 2表示用snappy压缩.压缩后消息中会有头来指明消息压缩类型, 故在消费者端消息解压是透明的无需指定.
    compression.codec=none

    # 指定序列化处理类(mafka client API调用说明-->3.序列化约定wiki), 默认为kafka.serializer.DefaultEncoder,即byte[]
    serializer.class=com.meituan.mafka.client.codec.MafkaMessageEncoder
    # serializer.class=kafka.serializer.DefaultEncoder
    # serializer.class=kafka.serializer.StringEncoder

    # 如果要压缩消息, 这里指定哪些topic要压缩消息, 默认empty, 表示不压缩.
    #compressed.topics=

    ########## request ack ###############
    # producer接收消息ack的时机.默认为0.
    # 0: producer不会等待broker发送ack
```

```
    # 1: 当leader接收到消息之后发送ack
    # 2: 当所有的follower都同步消息成功后发送ack.
    request.required.acks=0

    # 在向producer发送ack之前,broker允许等待的最大时间
    # 如果超时,broker将会向producer发送一个error ACK.意味着上一次消息因为某种
    # 原因未能成功(比如follower未能同步成功)
    request.timeout.ms=10000
    ########## end ####################

    # 同步还是异步发送消息, 默认“sync”表同步, "async"表异步.异步可以提高发送吞吐量,
    # 也意味着消息将会在本地buffer中,并适时批量发送, 但是也可能导致丢失未发送过去的消息
    producer.type=sync

    ############## 异步发送 (以下四个异步参数可选) ###################
    # 在async模式下,当message被缓存的时间超过此值后,将会批量发送给broker,默认为5000ms
    # 此值和batch.num.messages协同工作.
    queue.buffering.max.ms = 5000

    # 在async模式下,producer端允许buffer的最大消息量
    # 无论如何,producer都无法尽快的将消息发送给broker,从而导致消息在producer端大量沉积
    # 此时,如果消息的条数达到阀值,将会导致producer端阻塞或者消息被抛弃, 默认为10000
    queue.buffering.max.messages=20000

    # 如果是异步, 指定每次批量发送数据量, 默认为200
    batch.num.messages=500

    # 当消息在producer端沉积的条数达到"queue.buffering.max.meesages"后
    # 阻塞一定时间后,队列仍然没有enqueue(producer仍然没有发送出任何消息)
    # 此时producer可以继续阻塞或者将消息抛弃,此timeout值用于控制"阻塞"的时间
    # -1: 无阻塞超时限制,消息不会被抛弃
    # 0:立即清空队列,消息被抛弃
    queue.enqueue.timeout.ms=-1
    ############### end ###############

    # 当producer接收到error ACK,或者没有接收到ACK时,允许消息重发的次数
    # 因为broker并没有完整的机制来避免消息重复,所以当网络异常时(比如ACK丢失)
    # 有可能导致broker接收到重复的消息,默认值为3.
    message.send.max.retries=3

    # producer刷新topic metada的时间间隔,producer需要知道partition leader的位置,以
及当前topic的情况
    # 因此producer需要一个机制来获取最新的metadata,当producer遇到特定错误时,将会立即刷
新
    # (比如topic失效,partition丢失,leader失效等),此外也可以通过此参数来配置额外的刷新
机制, 默认值600000
    topic.metadata.refresh.interval.ms=60000
```

consumer参数说明(配置文件位于config/consumer.properties或者在程序内定义)

```
    # zookeeper连接服务器地址, 此处为线下测试环境配置(kafka消息服务-->kafka broker集群线上部署环境wiki)
    # 配置例子: "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002"

zookeeper.connect=192.168.2.225:2181,192.168.2.225:2182,192.168.2.225:2183/config/mobile/mq/mafka

    # zookeeper的session过期时间, 默认5000ms, 用于检测消费者是否挂掉, 当消费者挂掉, 其他消费者要等该指定时间才能检查到并且触发重新负载均衡
    zookeeper.session.timeout.ms=5000
    zookeeper.connection.timeout.ms=10000

    # 指定多久消费者更新offset到zookeeper中.注意offset更新时基于time而不是每次获得的消息.一旦在更新zookeeper发生异常并重启, 将可能拿到已拿到过的消息
    zookeeper.sync.time.ms=2000

    #指定消费组
    group.id=xxx

    # 当consumer消费一定量的消息之后,将会自动向zookeeper提交offset信息
    # 注意offset信息并不是每消费一次消息就向zk提交一次,而是现在本地保存(内存),并定期提交,默认为true
    auto.commit.enable=true

    # 自动更新时间.默认60 * 1000
    auto.commit.interval.ms=1000

    # 当前consumer的标识,可以设定,也可以有系统生成,主要用来跟踪消息消费情况,便于观察
    conusmer.id=xxx

    # 消费者客户端编号, 用于区分不同客户端, 默认客户端程序自动产生
    client.id=xxxx

    # 最大取多少块缓存到消费者(默认10)
    queued.max.message.chunks=50

    # 当有新的consumer加入到group时,将会reblance,此后将会有partitions的消费端迁移到新
    # 的consumer上,如果一个consumer获得了某个partition的消费权限,那么它将会向zk注册
    # "Partition Owner registry"节点信息,但是有可能此时旧的consumer尚没有释放此节点,
    # 此值用于控制,注册节点的重试次数.
    rebalance.max.retries=5

    # 获取消息的最大尺寸,broker不会像consumer输出大于此值的消息chunk
    # 每次feth将得到多条消息,此值为总大小,提升此值,将会消耗更多的consumer端内存
    fetch.min.bytes=6553600

    # 当消息的尺寸不足时,server阻塞的时间,如果超时,消息将立即发送给consumer
    fetch.wait.max.ms=5000
    socket.receive.buffer.bytes=655360
```

```
    # 如果zookeeper没有offset值或offset值超出范围.那么就给个初始的offset.有smallest,
largest,
    # anything可选, 分别表示给当前最小的offset, 当前最大的offset, 抛异常.默认largest
    auto.offset.reset=smallest

    # 指定序列化处理类(mafka client API调用说明-->3.序列化约定wiki), 默认为
kafka.serializer.DefaultDecoder,即byte[]
    derializer.class=com.meituan.mafka.client.codec.MafkaMessageDecoder
```

附录 B 非官方组件 Kafka Manager (建议配置)

1 构建 Kafka Manager

Kafka Manager 官方并没有提供 Kafka Manager 的二进制发布包, 必须自己构建. (在本文的附件中有已构建好的 1.3.1.8 版 Kafka Manager, 可跳过这一步)

1. 下载源码, 这里选择 1.3.1.8 版, 解压.
2. 进入下载的 kafka-manager 源码目录, 执行 `./sbt clean dist`. 因为有许多scala 依赖和 maven 依赖需要下载, 所以这一步需要联网, 且要等待一段时间(国内一般网速差不多要一个小时).
3. 当上一步的命令执行完毕, 且终端输出 `success`, 则构建完成, 目标在 kafka-manager 源码目录下的 `./target/universal/` 目录下, 名为 kafka-manager-1.3.1.8.zip

2 配置 Kafka Manager

只需在一台机器上配置 Kafka Manager 即可, 只要可以连接上 zookeeper 集群和 kafka 集群就可以监控整个 kafka 集群.

将已构建好的 kafka-manager-1.3.1.8.zip 解压到 /usr/local/

修改 /usr/local/kafka-manager-1.3.1.8/conf/application.conf

```
kafka-manager.zkhosts="192.168.1.1:2181,192.168.1.2:2181,192.168.1.3:2181"
kafka-manager.zkhosts=${?ZK_HOSTS}
```

只修改第一条为 Zookeeper 的连接字符串, 第二条不要删

第二种方案: 配置环境变量 export ZK_HOSTS=192.168.1.1:2181,192.168.1.2:2181,192.168.1.3:2181, 则不需要修改 application.conf 文件

3 启动 Kafka Manager

```
nohup bin/kafka-manager -Dconfig.file=conf/application.conf > kafka-manager-
boot.log 2>&1 &
```

默认将通过 9000 访问, 可在启动时指定

```
nohup bin/kafka-manager -Dconfig.file=conf/application.conf -Dhttp.port=9001
> kafka-manager-boot.log 2>&1 &
```

或在配置文件 application.conf 中指定 http.port=9001

4 使用 Kafka Manager

使用浏览器访问 http://{配置了 Kafka Manager 的机器的 ip}:9000

添加集群, 填写 ZK_HOSTS.

5 注意

1. 更改 kafka 的启动命令

Kafka Manager 底层使用 JMX 来做性能测量, 要在启动 kafka 时指定 JMX 使用的端口 JMX_PORT 和 hostname. 新的启动命令如下

```
JMX_PORT=9999 bin/kafka-server-start.sh -daemon config/server.properties --
override -Djava.rmi.server.hostname=`hostname`
```

2. 远程连接 JMX 问题

JMX 除了要监听用户指定的端口 , 还需要两个随机端口号(真的是随机的, 所以无法指定开放). 若开启了防火墙, 远程连接时, 访问 JMX 失败.

解决:

如果可以关闭 firewall 就选择关闭, 就可以远程连接 JMX.

如果不可以关闭 firewall, 就在 firewall 中配置 rich rule: 对指定的 IP 不做拦截. 命令如下, 令 Kafka 集群所有机器对 Kafka Manager 所在机器 IP 不做拦截.

```
sudo firewall-cmd --permanent --add-rich-rule="rule family='ipv4' source
address='192.168.1.1' accept"
```