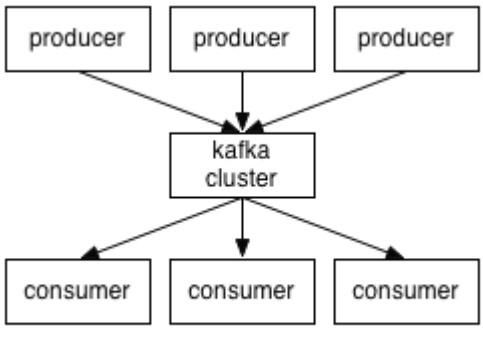
Kafka 集群文档

一、入门

1、简介

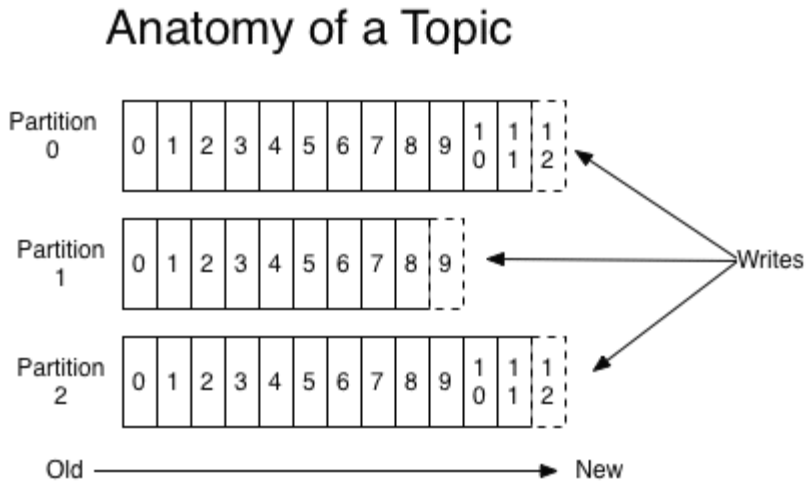
Kafka 是 linkedin 用于日志处理的分布式消息队列，同时支持离线和在线日志处理。kafka 对消息保存时根据 Topic 进行归类，发送消息者成为 Producer,消息接受者成为 Consumer,此外 kafka 集群有多个 kafka 实例组成，每个实例(server)称为 broker。无论是 kafka 集群，还是 producer 和 consumer 都依赖于 zookeeper 来保证系统可用性，为集群保存一些 meta 信息。



(摘自官网)

2、Topics/logs

一个 Topic 可以认为是一类消息，每个 topic 将被分成多个 partition(区),每个 partition 在存储层面是 append log 文件。任何发布到此 partition 的消息都会被直接追加到 log 文件的尾部，每条消息在文件中的位置称为 offset（偏移量），offset 为一个 long 型数字，它是唯一标记一条消息。kafka 并没有提供其他额外的索引机制来存储 offset，因为在 kafka 中几乎不允许对消息进行“随机读写”。



(摘自官网)

在 kafka 中，即使消息被消费,消息仍然不会被立即删除。日志文件将会根据 broker 中的配置要求,保留一定的时间之后删除;比如 log 文件保留 2 天,那么两天后,文件会被清除,无论其中的消息是否被消费。kafka 通过这种简单的手段,来释放磁盘空间,以及减少消息消费之后对文件内容改动的磁盘 IO 开支。

对于 consumer 而言,它需要保存消费消息的 offset,对于 offset

的保存和使用,由 consumer 来控制;当 consumer 正常消费消息时,offset 将会"线性"的向前驱动,即消息将依次顺序被消费。事实上 consumer 可以使用任意顺序消费消息,它只需要将 offset 重置为任意值。(offset 将会保存在 zookeeper 中,参见下文)

kafka集群几乎不需要维护任何consumer和producer状态信息,这些信息由 zookeeper 保存;因此 producer 和 consumer 的客户端实现非常轻量级,它们可以随意离开,而不会对集群造成额外的影响。

partitions 的设计目的有多个。最根本原因是 kafka 基于文件存储。通过分区,可以将日志内容分散到多个 server 上,来避免文件尺寸达到单机磁盘的上限,每个 partiton 都会被当前 server(kafka 实例)保存;可以将一个 topic 切分多任意多个 partitions 来保存消息。此外越多的 partitions 意味着可以容纳更多的 consumer,有效提升并发消费的能力。(具体原理参见下文)。

3、Distribution

一个 Topic 的多个 partitions,被分布在 kafka 集群中的多个 server 上;每个 server(kafka 实例)负责 partitions 中消息的读写操作;此外kafka还可以配置partitions需要备份的个数(replicas),每个 partition 将会被备份到多台机器上,以提高可用性。

基于 replicated 方案,那么就意味着需要对多个备份进行调度;每个 partition 都有一个 server 为"leader";leader 负责所有的读写操作,如果 leader 失效,那么将会有其他 follower 来接管(成为新

的 leader);follower 只是单调的和 leader 跟进,同步消息即可。由此可见作为 leader 的 server 承载了全部的请求压力,因此从集群的整体考虑,有多少个 partitions 就意味着有多少个"leader",kafka 会将"leader"均衡的分散在每个实例上,来确保整体的性能稳定。

- ✓ 发送到 partitions 中的消息将会按照它接收的顺序追加到日志中。
- ✓ 对于消费者而言,它们消费消息的顺序和日志中消息顺序一致。
- ✓ 如果 Topic 的"replication factor"为 N,那么允许 N-1 个 kafka 实例失效。

4、Producers

Producer 将消息发布到指定的 Topic 中,同时 Producer 也能决定将此消息归属于哪个 partition;比如基于"round-robin"方式或者通过其他的一些算法等。

5、Consumers

本质上 kafka 只支持 Topic。每个 consumer 属于一个 consumer group;反过来说,每个 group 中可以有多个 consumer。发送到 Topic 的消息,只会被订阅此 Topic 的每个 group 中的一个 consumer 消费。

如果所有的 consumer 都具有相同的 group,这种情况和 queue 模式很像;消息将会在 consumers 之间负载均衡。

如果所有的 consumer 都具有不同的 group,那这就是"发布-订阅

"，消息将会广播给所有的消费者。

在 kafka 中，一个 partition 中的消息只会被 group 中的一个 consumer 消费；每个 group 中 consumer 消息消费互相独立；我们可以认为一个 group 是一个"订阅"者，一个 Topic 中的每个 partions，只会被一个"订阅者"中的一个 consumer 消费，不过一个 consumer 可以消费多个 partitions 中的消息。kafka 只能保证一个 partition 中的消息被某个 consumer 消费时，消息是顺序的。事实上，从 Topic 角度来说，消息仍不是有序的。

kafka 的设计原理决定，对于一个 topic，同一个 group 中不能有多于 partitions 个数的 consumer 同时消费，否则将意味着某些 consumer 将无法得到消息。

二、使用场景

1、Message

对于一些常规的消息系统，kafka 是个不错的选择；partitons/replication 和容错，可以使 kafka 具有良好的扩展性和性能优势。不过 kafka 并没有提供 JMS 中的"事务性"、"消息确认机制"、"消息分组"等企业级特性，kafka 只能作为"常规"的消息系统，在一定程度上，尚未确保消息的发送与接收绝对可靠（比如，消息重发，消息发送丢失等）。

2、Websit activity tracking

kafka 可以作为"网站活性跟踪"的最佳工具;可以将网页/用户操作等信息发送到 kafka 中。并实时监控,或者离线统计分析等。

3、Log Aggregation

kafka 的特性决定它非常适合作为"日志收集中心";application 可以将操作日志"批量"、"异步"的发送到 kafka 集群中,而不是保存在本地或者 DB 中;kafka 可以批量提交消息/压缩消息等,这对 producer 端而言,几乎感觉不到性能的开支。此时 consumer 端可以使 hadoop 等其他系统化的存储和分析系统。

三、设计原理

kafka 的设计初衷是希望作为一个统一的信息收集平台,能够实时的收集反馈信息,并需要能够支撑较大的数据量,且具备良好的容错能力。

1、持久性

kafka 使用文件存储消息,这就直接决定 kafka 在性能上严重依赖文件系统的本身特性。且无论任何 OS 下,对文件系统本身的优化几乎没有可能。文件缓存/直接内存映射等是常用的手段。因为 kafka 是对日志文件进行 append 操作,因此磁盘检索的开支是较小的;同时为了减少磁盘写入的次数,broker 会将消息暂时 buffer 起来,当消息的

个数(或尺寸)达到一定阀值时,再 flush 到磁盘,这样减少了磁盘 IO 调用的次数。

2、性能

需要考虑的影响性能点很多,除磁盘 IO 之外,我们还需要考虑网络 IO,这直接关系到 kafka 的吞吐量问题。kafka 并没有提供太多高超的技巧;对于 producer 端,可以将消息 buffer 起来,当消息的条数达到一定阀值时,批量发送给 broker;对于 consumer 端也是一样,批量 fetch 多条消息。不过消息量的大小可以通过配置文件来指定。对于 kafka broker 端,似乎有个 sendfile 系统调用可以潜在的提升网络 IO 的性能:将文件的数据映射到系统内存中,socket 直接读取相应的内存区域即可,而无需进程再次 copy 和交换。 其实对于 producer/consumer/broker 三者而言,CPU 的开支应该都不大,因此启用消息压缩机制是一个良好的策略;压缩需要消耗少量的 CPU 资源,不过对于 kafka 而言,网络 IO 更应该需要考虑。可以将任何在网络上传输的消息都经过压缩。kafka 支持 gzip/snappy 等多种压缩方式。

3、生产者

负载均衡: producer 将会和 Topic 下所有 partition leader 保持 socket 连接;消息由 producer 直接通过 socket 发送到 broker,中间不会经过任何"路由层"。事实上,消息被路由到哪个 partition 上,由 producer 客户端决定。比如可以采用"random""key-hash""轮询"

等,如果一个 topic 中有多个 partitions,那么在 producer 端实现"消息均衡分发"是必要的。

其中 partition leader 的位置(host:port)注册在 zookeeper 中,producer 作为 zookeeper 的 client,已经注册了 watch 用来监听 partition leader 的变更事件。

异步发送:将多条消息暂且在客户端 buffer 起来,并将他们批量的发送到 broker,小数据 IO 太多,会拖慢整体的网络延迟,批量延迟发送事实上提升了网络效率。不过这也有一定的隐患,比如说当 producer 失效时,那些尚未发送的消息将会丢失。

利用 Java 进行编程,需要引入相关 jar 包,具体如下:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.9.2</artifactId>
  <version>0.8.1.1</version>
  <scope>compile</scope>
  <exclusions>
    <exclusion>
      <artifactId>jmxri</artifactId>
      <groupId>com.sun.jmx</groupId>
    </exclusion>
    <exclusion>
      <artifactId>jms</artifactId>
      <groupId>javax.jms</groupId>
    </exclusion>
    <exclusion>
      <artifactId>jmxtools</artifactId>
      <groupId>com.sun.jdmk</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

如下是一个 Java 例子:

```
import java.util.*;
import kafka.javaapi.producer.Producer;
```

```
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class TestProducer {
    public static void main(String[] args) {
        long events = 100l;
        Random rnd = new Random();

        /**生产者相关配置文件项**/
        Properties props = new Properties();
        props.put("metadata.broker.list", "broker1:9092,broker2:9092 ");
        props.put("serializer.class", "kafka.serializer.StringEncoder");
        props.put("partitioner.class",
"com.gds.kafa.kafa.partitioner.SimplePartitioner");
        props.put("request.required.acks", "1");
        //创建生产者配置
        ProducerConfig config = new ProducerConfig(props);
        //创建生产者
        Producer<String, String> producer = new Producer<String, String>(config);

        for (long nEvents = 0; nEvents < events; nEvents++) {
               long runtime = new Date().getTime();
               String ip = "192.168.2." + rnd.nextInt(255);
               String msg = runtime + ",www.example.com," + ip;
               KeyedMessage<String,   String>   data   =   new   KeyedMessage<String,
String>("page_visits", ip, msg);
               producer.send(data);
        }
        producer.close();
    }
}
package com.gds.kafa.kafa.partitioner;
import kafka.producer.Partitioner;
import kafka.utils.VerifiableProperties;

public class SimplePartitioner implements Partitioner {
    public SimplePartitioner (VerifiableProperties props) {

    }
    /**
     * 确定生产者向broker的topic的那个partition中插入数据
     */
    public int partition(Object key, int a_numPartitions) {
        int partition = 0;
```

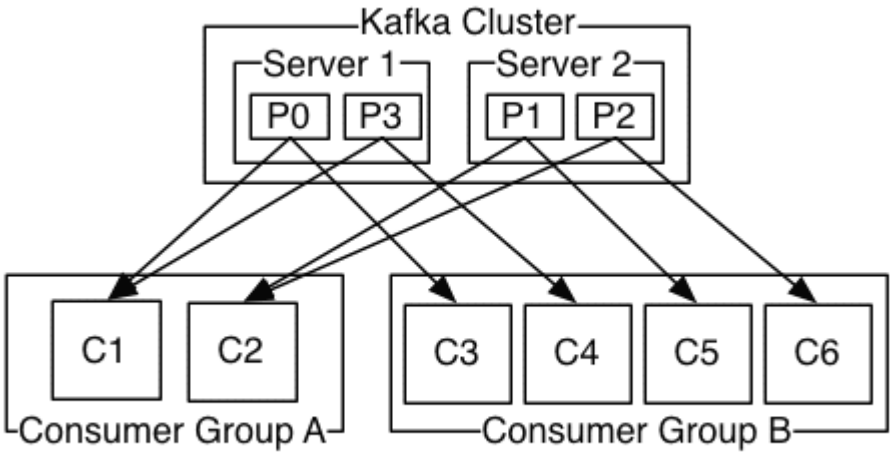
```
        String stringKey = (String) key;
        int offset = stringKey.lastIndexOf('.');
        if (offset > 0) {
           partition    =    Integer.parseInt(    stringKey.substring(offset+1))    %
a_numPartitions;
        }
       return partition;
  }
}
```

4、消费者

consumer 端向 broker 发送"fetch"请求，并告知其获取消息的 offset；此后 consumer 将会获得一定条数的消息；consumer 端也可以重置 offset 来重新消费消息。

在 kafka 中，producers 将消息推送给 broker 端，consumer 在和 broker 建立连接之后，主动去 pull（或者说 fetch）消息；这中模式有些优点，首先 consumer 端可以根据自己的消费能力适时的去 fetch 消息并处理，且可以控制消息消费的进度（offset）；此外，消费者可以良好的控制消息消费的数量，batch fetch。

在 kafka 中，partition 中的消息只有一个 consumer 在消费，且不存在消息状态的控制，也没有复杂的消息确认机制，可见 kafka broker 端是相当轻量级的。当消息被 consumer 接收之后，consumer 可以在本地保存最后消息的 offset，并间歇性的向 zookeeper 注册 offset。由此可见，consumer 客户端也很轻量级。



(摘自官网)

Kafka 提供了两套 API 给 Consumer：

✓ The high-level Consumer API

✓ The SimpleConsumer API

第一种高度抽象的 Consumer API，它使用起来简单、方便，但是对于某些特殊的需求我们可能要用到第二种更底层的 API。

High Level Consumer

在某些应用场景，我们希望通过多线程读取消息，而我们并不关心从 Kafka 消费消息的顺序，我们仅仅关心数据能被消费就行。High Level 就是用于抽象这类消费动作的。

消息消费以 Consumer Group 为单位，每个 Consumer Group 中可以有多个 consumer，每个 consumer 是一个线程，topic 的每个 partition 同时只能被某一个 consumer 读取，Consumer Group 对应的每个 partition 都有一个最新的 offset 的值，存储在

zookeeper 上的。所以不会出现重复消费的情况。

High Level Consumer 可以并且应该被使用在多线程的环境，线程模型中线程的数量(也代表 group 中 consumer 的数量)和 topic 的 partition 数量有关，下面列举一些规则：

- 当提供的线程数量多于 partition 的数量，则部分线程将不会接收到消息；
- 当提供的线程数量少于 partition 的数量，则部分线程将从多个 partition 接收消息；
- 当某个线程从多个 partition 接收消息时，不保证接收消息的顺序；可能出现从 partition3接收5条消息，从 partition4接收6条消息，接着又从 partition3接收10条消息；
- 当添加更多线程时，会引起 kafka 做 re-balance，可能改变 partition 和线程的对应关系。

代码示例如下：

```
package com.gds.kafa.kafa.consumer;

import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConsumerGroupExample {
    private final ConsumerConnector consumer;
```

```
    private final String topic;
    private ExecutorService executor;

    public ConsumerGroupExample(String a_zookeeper, String a_groupId,
            String a_topic) {
        consumer = kafka.consumer.Consumer
                .createJavaConsumerConnector(createConsumerConfig(a_zookeeper,
                        a_groupId));
        this.topic = a_topic;
    }

    public void shutdown() {
        if (consumer != null)
            consumer.shutdown();
        if (executor != null)
            executor.shutdown();
    }

    public void run(int a_numThreads) {
        Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
        topicCountMap.put(topic, new Integer(a_numThreads));
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer
                .createMessageStreams(topicCountMap);
        List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

        // now launch all the threads
        //
        executor = Executors.newFixedThreadPool(a_numThreads);

        // now create an object to consume the messages
        //
        int threadNumber = 0;
        for (final KafkaStream<byte[], byte[]> stream : streams) {
            executor.submit(new ConsumerTest(stream, threadNumber));
            threadNumber++;
        }
    }

    private ConsumerConfig createConsumerConfig(String a_zookeeper,
            String a_groupId) {
        Properties props = new Properties();
        props.put("zookeeper.connect", a_zookeeper);
        props.put("group.id", a_groupId);
        props.put("zookeeper.session.timeout.ms", "400");
```

```
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");

        return new ConsumerConfig(props);
    }

    public static void main(String[] args) {
        String zooKeeper = args[0];
        String groupId = args[1];
        String topic = args[2];
        int threads = Integer.parseInt(args[3]);

        ConsumerGroupExample example = new ConsumerGroupExample(zooKeeper,
                groupId, topic);
        example.run(threads);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException ie) {

        }
        example.shutdown();
    }
}



package com.gds.kafa.kafa.consumer;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;

public class ConsumerTest implements Runnable {
    private KafkaStream<byte[], byte[]> m_stream;
    private int m_threadNumber;

    public ConsumerTest(KafkaStream<byte[], byte[]> a_stream, int a_threadNumber) {
        m_threadNumber = a_threadNumber;
        m_stream = a_stream;
    }

    public void run() {
        ConsumerIterator<byte[], byte[]> it = m_stream.iterator();
        while (it.hasNext())
            System.out.println("Thread " + m_threadNumber + ": " + new
```

```
String(it.next().message()));
        System.out.println("Shutting down Thread: " + m_threadNumber);
    }
}
```

The SimpleConsumer API

这种 API 能够帮助我们做哪些事情：

- ✓ 一个消息读取多次
- ✓ 在一个处理过程中只消费 Partition 其中的一部分消息
- ✓ 添加事务管理机制以保证消息被处理且仅被处理一次

使用 SimpleConsumer 有哪些弊端呢？

- ✓ 必须在程序中跟踪 offset 值
- ✓ 必须找出指定 Topic Partition 中的 lead broker
- ✓ 必须处理 broker 的变动

使用 SimpleConsumer 的步骤

- ✓ 从所有活跃的 broker 中找出哪个是指定 Topic Partition 中的 leader broker
- ✓ 找出指定 Topic Partition 中的所有备份 broker
- ✓ 构造请求
- ✓ 发送请求查询数据
- ✓ 处理 leader broker 变更

示例代码如下：

```
package com.gds.kafa.kafa.consumer;

import kafka.api.FetchRequest;
import kafka.api.FetchRequestBuilder;
```

```
import kafka.api.PartitionOffsetRequestInfo;
import kafka.common.ErrorMapping;
import kafka.common.TopicAndPartition;
import kafka.javaapi.*;
import kafka.javaapi.consumer.SimpleConsumer;
import kafka.message.MessageAndOffset;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SimpleExample {
    public static void main(String args[]) {
        SimpleExample example = new SimpleExample();
        long maxReads = Long.parseLong(args[0]);
        String topic = args[1];
        int partition = Integer.parseInt(args[2]);
        List<String> seeds = new ArrayList<String>();
        seeds.add(args[3]);
        int port = Integer.parseInt(args[4]);
        try {
            example.run(maxReads, topic, partition, seeds, port);
        } catch (Exception e) {
            System.out.println("Oops:" + e);
            e.printStackTrace();
        }
    }

    private List<String> m_replicaBrokers = new ArrayList<String>();

    public SimpleExample() {
        m_replicaBrokers = new ArrayList<String>();
    }

    public void run(long a_maxReads, String a_topic, int a_partition,
            List<String> a_seedBrokers, int a_port) throws Exception {
        // find the meta data about the topic and partition we are interested in
        PartitionMetadata metadata = findLeader(a_seedBrokers, a_port, a_topic,
                a_partition);
        if (metadata == null) {
            System.out
                    .println("Can't find metadata for Topic and Partition. Exiting");
```

```
            return;
        }
        if (metadata.leader() == null) {
            System.out
                    .println("Can't find Leader for Topic and Partition. Exiting");
            return;
        }
        String leadBroker = metadata.leader().host();
        String clientName = "Client_" + a_topic + "_" + a_partition;

        SimpleConsumer consumer = new SimpleConsumer(leadBroker, a_port,
                100000, 64 * 1024, clientName);
        long readOffset = getLastOffset(consumer, a_topic, a_partition,
                kafka.api.OffsetRequest.EarliestTime(), clientName);

        int numErrors = 0;
        while (a_maxReads > 0) {
            if (consumer == null) {
                consumer = new SimpleConsumer(leadBroker, a_port, 100000,
                        64 * 1024, clientName);
            }
            FetchRequest req = new FetchRequestBuilder().clientId(clientName)
                    .addFetch(a_topic, a_partition, readOffset, 100000).build();
            // Note: this fetchSize of 100000 might need to be increased if large batches
are written to Kafka
            FetchResponse fetchResponse = consumer.fetch(req);

            if (fetchResponse.hasError()) {
                numErrors++;
                // Something went wrong!
                short code = fetchResponse.errorCode(a_topic, a_partition);
                System.out.println("Error fetching data from the Broker:"
                        + leadBroker + " Reason: " + code);
                if (numErrors > 5)
                    break;
                if (code == ErrorMapping.OffsetOutOfRangeCode()) {
                    // We asked for an invalid offset. For simple case ask for
                    // the last element to reset
                    readOffset = getLastOffset(consumer, a_topic, a_partition,
                            kafka.api.OffsetRequest.LatestTime(), clientName);
                    continue;
                }
                consumer.close();
                consumer = null;
```

```
                leadBroker = findNewLeader(leadBroker, a_topic, a_partition,
                        a_port);
                continue;
            }
            numErrors = 0;

            long numRead = 0;
            for (MessageAndOffset messageAndOffset : fetchResponse.messageSet(
                    a_topic, a_partition)) {
                long currentOffset = messageAndOffset.offset();
                if (currentOffset < readOffset) {
                    System.out.println("Found an old offset: " + currentOffset
                            + " Expecting: " + readOffset);
                    continue;
                }
                readOffset = messageAndOffset.nextOffset();
                ByteBuffer payload = messageAndOffset.message().payload();

                byte[] bytes = new byte[payload.limit()];
                payload.get(bytes);
                System.out.println(String.valueOf(messageAndOffset.offset())
                        + ": " + new String(bytes, "UTF-8"));
                numRead++;
                a_maxReads--;
            }

            if (numRead == 0) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ie) {
                }
            }
        }
        if (consumer != null)
            consumer.close();
    }

    public static long getLastOffset(SimpleConsumer consumer, String topic,
            int partition, long whichTime, String clientName) {
        TopicAndPartition topicAndPartition = new TopicAndPartition(topic,
                partition);
        Map<TopicAndPartition,   PartitionOffsetRequestInfo>   requestInfo   =   new
HashMap<TopicAndPartition, PartitionOffsetRequestInfo>();
        requestInfo.put(topicAndPartition, new PartitionOffsetRequestInfo(
```

```
                whichTime, 1));
        kafka.javaapi.OffsetRequest request = new kafka.javaapi.OffsetRequest(
                requestInfo, kafka.api.OffsetRequest.CurrentVersion(),
                clientName);
        OffsetResponse response = consumer.getOffsetsBefore(request);

        if (response.hasError()) {
            System.out
                    .println("Error fetching data Offset Data the Broker. Reason: "
                            + response.errorCode(topic, partition));
            return 0;
        }
        long[] offsets = response.offsets(topic, partition);
        return offsets[0];
    }

    private String findNewLeader(String a_oldLeader, String a_topic,
            int a_partition, int a_port) throws Exception {
        for (int i = 0; i < 3; i++) {
            boolean goToSleep = false;
            PartitionMetadata metadata = findLeader(m_replicaBrokers, a_port,
                    a_topic, a_partition);
            if (metadata == null) {
                goToSleep = true;
            } else if (metadata.leader() == null) {
                goToSleep = true;
            } else if (a_oldLeader.equalsIgnoreCase(metadata.leader().host())
                    && i == 0) {
                // first time through if the leader hasn't changed give
                // ZooKeeper a second to recover
                // second time, assume the broker did recover before failover,
                // or it was a non-Broker issue
                //
                goToSleep = true;
            } else {
                return metadata.leader().host();
            }
            if (goToSleep) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ie) {
                }
            }
        }
```

```
        System.out
                .println("Unable to find new leader after Broker failure. Exiting");
        throw new Exception(
                "Unable to find new leader after Broker failure. Exiting");
    }

    private PartitionMetadata findLeader(List<String> a_seedBrokers,
            int a_port, String a_topic, int a_partition) {
        PartitionMetadata returnMetaData = null;
        loop: for (String seed : a_seedBrokers) {
            SimpleConsumer consumer = null;
            try {
                consumer = new SimpleConsumer(seed, a_port, 100000, 64 * 1024,
                        "leaderLookup");
                List<String> topics = Collections.singletonList(a_topic);
                TopicMetadataRequest req = new TopicMetadataRequest(topics);
                kafka.javaapi.TopicMetadataResponse resp = consumer.send(req);

                List<TopicMetadata> metaData = resp.topicsMetadata();
                for (TopicMetadata item : metaData) {
                    for (PartitionMetadata part : item.partitionsMetadata()) {
                        if (part.partitionId() == a_partition) {
                            returnMetaData = part;
                            break loop;
                        }
                    }
                }
            } catch (Exception e) {
                System.out.println("Error communicating with Broker [" + seed
                        + "] to find Leader for [" + a_topic + ", "
                        + a_partition + "] Reason: " + e);
            } finally {
                if (consumer != null)
                    consumer.close();
            }
        }
        if (returnMetaData != null) {
            m_replicaBrokers.clear();
            for (kafka.cluster.Broker replica : returnMetaData.replicas()) {
                m_replicaBrokers.add(replica.host());
            }
        }
        return returnMetaData;
    }
```

```
}
```

5、消息传送机制

对于 JMS 实现,消息传输担保非常直接:有且只有一次(exactly once)。在 kafka 中稍有不同:

- ✓ at most once: 最多一次,发送一次,无论成败,将不会重发。消费者 fetch 消息,然后保存 offset,然后处理消息;当 client 保存 offset 之后,但是在消息处理过程中出现了异常,导致部分消息未能继续处理。那么此后"未处理"的消息将不能被 fetch 到,这就是"at most once"。
- ✓ at least once: 消息至少发送一次,如果消息未能接受成功,可能会重发,直到接收成功。消费者 fetch 消息,然后处理消息,然后保存 offset。如果消息处理成功之后,但是在保存 offset 阶段 zookeeper 异常导致保存操作未能执行成功,这就导致接下来再次 fetch 时可能获得上次已经处理过的消息,这就是"at least once",原因 offset 没有及时的提交给 zookeeper,zookeeper 恢复正常还是之前 offset 状态。
- ✓ exactly once: 消息只会发送一次。kafka 中并没有严格的去实现(基于 2 阶段提交,事务),我们认为这种策略在 kafka 中是没有必要的。

通常情况下"at-least-once"是我们首选。(相比 at most once 而言,重复接收数据总比丢失数据要好)。

6、复制备份

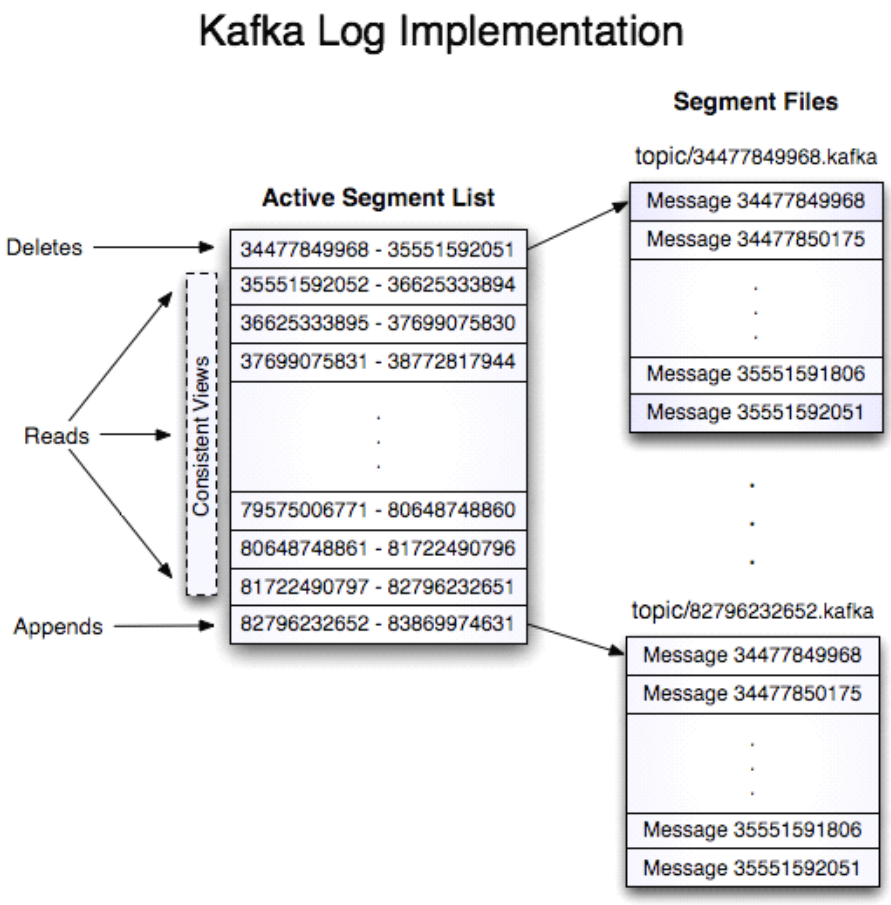
kafka 将每个 partition 数据复制到多个 server 上,任何一个 partition 有一个 leader 和多个 follower(可以没有);备份的个数可以通过 broker 配置文件来设定。leader 处理所有的 read-write 请求,follower 需要和 leader 保持同步。Follower 和 consumer 一样,消费消息并保存在本地日志中;leader 负责跟踪所有的 follower 状态,如果 follower"落后"太多或者失效,leader 将会把它从 replicas 同步列表中删除。当所有的 follower 都将一条消息保存成功,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"。

7、日志

如果一个 topic 的名称为"my_topic",它有 2 个 partitions,那么日志将会保存在 my_topic_0 和 my_topic_1 两个目录中;日志文件

中保存了一序列"log entries"(日志条目),每个 log entry 格式为"4 个字节的数字 N 表示消息的长度" + "N 个字节的消息内容";每个日志都有一个 offset 来唯一的标记一条消息,offset 的值为 8 个字节的数字,表示此消息在此 partition 中所处的起始位置。每个 partition 在物理存储层面,有多个 log file 组成(称为 segment)。segment file 的命名为"最小 offset".kafka 。例如"00000000000.kafka";其中"最小 offset"表示此 segment 中起始消息的 offset。



(摘自官网)

其中每个 partiton 中所持有的 segments 列表信息会存储在 zookeeper 中。

当 segment 文件尺寸达到一定阀值时(可以通过配置文件设定,默认 1G),将会创建一个新的文件;当 buffer 中消息的条数达到阀值时将会触发日志信息 flush 到日志文件中,同时如果"距离最近一次 flush 的时间差"达到阀值时,也会触发 flush 到日志文件。如果 broker 失效,极有可能会丢失那些尚未 flush 到文件的消息。因为 server 意外实现,仍然会导致 log 文件格式的破坏(文件尾部),那么就要求当 server 启东是需要检测最后一个 segment 的文件结构是否合法并进行必要的修复。

获取消息时,需要指定 offset 和最大 chunk 尺寸,offset 用来表示消息的起始位置,chunk size 用来表示最大获取消息的总长度(间接的表示消息的条数)。根据 offset,可以找到此消息所在 segment 文件,然后根据 segment 的最小 offset 取差值,得到它在 file 中的相对位置,直接读取输出即可。

日志文件的删除策略非常简单:启动一个后台线程定期扫描 log file 列表,把保存时间超过阀值的文件直接删除(根据文件的创建时间)。为了避免删除文件时仍然有 read 操作(consumer 消费),采取 copy-on-write 方式。

8、zookeeper

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)

- 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: /brokers/ids/[0。。。N], 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。

- 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: /brokers/topics/[topic]/partitions/[0。。。N] 其中[0。。N]表示 partition 索引号。

- 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。

- 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。

- 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。

- Partition Owner registry: 用来标记 partition 被哪个 consumer 消费。临时 znode。

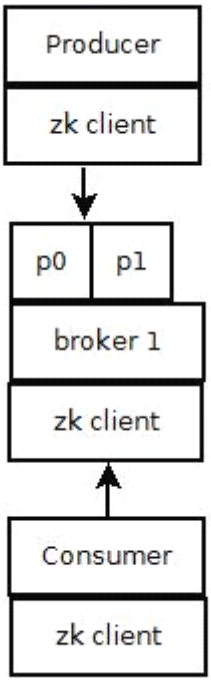
格式:

/consumers/[group_id]/owners/[topic]/[broker_id-partition_id] -->consumer_node_id

当 consumer 启动时,所触发的操作:

- ✓ 首先进行"Consumer id Registry";
- ✓ 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)。
- ✓ 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。



- 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。
- Broker 端使用 zookeeper 用来注册 broker 信息,已经监测 partition leader 存活性。
- 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 partition leader 建立 socket 连接,并获取消息。

四、主要配置

1、Broker 配置

```
##broker标识,id为正数,且全局不得重复.
broker.id=1
##日志文件保存的目录
log.dirs=~/kafka/logs
##broker需要使用zookeeper保存meata信息,因此broker为zk client;
##此处为zookeeper集群的connectString,后面可以跟上path,比如
##hostname:port/chroot/kafka
##不过需要注意,path的全路径需要有自己来创建(使用zookeeper脚本工具)
zookeeper.connect=hostname1:port1,hostname2:port2
##用来侦听链接的端口,prudcer或consumer将在此端口建立链接
port=6667
##指定broker实例绑定的网络接口地址
host.name=
##每个partition的备份个数,默认为1,建议根据实际条件选择
##此致值大意味着消息各个server上同步时需要的延迟较高
num.partitions=2
##日志文件中每个segment文件的尺寸,默认为1G
##log.segment.bytes=1024*1024*1024
##滚动生成新的segment文件的最大时间
##log.roll.hours=24*7
##segment文件保留的最长时间,超时将被删除
##log.retention.hours=24*7
##partiton中buffer中,消息的条数,达到阀值,将触发flush到磁盘.
log.flush.interval.messages=10000
#消息buffer的时间,达到阀值,将触发flush到磁盘.
log.flush.interval.ms=3000
##partition leader等待follower同步消息的最大时间,
##如果超时,leader将follower移除同步列表
replica.lag.time.max.ms=10000
##允许follower落后的最大消息条数,如果达到阀值,将follower移除同步列表
##replica.lag.max.message=4000
##消息的备份的个数,默认为1
num.replica.fetchers=1
```

2、Consumer 主要配置

```
##当前消费者的group名称,需要指定
group.id=
##consumer作为zookeeper client,需要通过zk保存一些meta信息,此处为zk connectString
zookeeper.connect=hostname1:port,hostname2:port2
##当前consumer的标识,可以设定,也可以有系统生成.
conusmer.id=
##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
fetch.messages.max.bytes=1024*1024
##当consumer消费一定里的消息之后,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,就像zk提交一次,而是现在本地保存,并定期提交
auto.commit.enable=true
##自动提交的时间间隔,默认为1分钟.
auto.commit.interval.ms=60*1000
```

3、Producer 主要配置

```
##对于开发者而言,需要通过broker.list指定当前producer需要关注的broker列表
##producer通过和每个broker链接,并获取partitions,
##如果某个broker链接失败,将导致此上的partitons无法继续发布消息
##格式:host1:port,host2:port2,其中host:port需要参考broker配置文件.
##对于producer而言没有使用zookeeper自动发现broker列表，非常奇怪。(0.8V和0.7有区别)
metadata.broker.list=
##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
##producer消息发送的模式,同步或异步.
##异步意味着消息将会在本地buffer,并适时批里发送
##默认为sync,建议async
producer.type=sync
##消息序列化类,将消息实体转换成byte[]
serializer.class=kafka.serializer.DefaultEncoder
key.serializer.class=${serializer.class}
##partitions路由类,消息在发送时将根据此实例的方法获得partition索引号.
partitioner.class=kafka.producer.DefaultPartitioner

##消息压缩算法,none,gzip,snappy
compression.codec=none
##消息在producer端buffer的条数.仅在producer.type=async下有效
##batch.num.messages=200
```

以上是关于 kafka 一些基础说明，在其中我们知道如果要 kafka 正常运行，必须配置 zookeeper，否则无论是 kafka 集群还是客户端的生存者和消费者都无法正常的工作的。

关于各个角色的配置详情，请参考官方文档：

http://kafka.apache.org/08/configuration.html

五、broker 集群的搭建

1、单机环境的搭建部署

1. 下载 kafka 压缩包，并解压

```
tar -xzf kafka_2.9.2-0.8.1.1.tgz
Mv kafka_2.9.2-0.8.1.1 kafka
cd kafka
```

2. 配置 zookeeper 集群、并启动 zookeeper 集群

3. 启动 kafka

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

4. 创建 topic

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

5. 查询 topic 列表和 topic 描述信息

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
Test
bin/kafka-topics.sh  --describe  --zookeeper  localhost:2181
--topic test
```

```
[root@master kafka]# bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
Topic:test      PartitionCount:6        ReplicationFactor:3     Configs:
        Topic: test     Partition: 0    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: test     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: test     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: test     Partition: 3    Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: test     Partition: 4    Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: test     Partition: 5    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
```

6. 向 topic 发送测试消息

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

This is a message

This is another message

7. 读取 topic 里的消息

bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

This is a message

This is another message

2、集群环境的搭建部署

和单机环境一样，只是需要修改下 broker 的配置文件而已。

1、将单机版的 kafka 目录复制到其他几台电脑上。

2、修改每台电脑上的 kafka 目录下的 server.properties 文件。

broker.id=1//这个参数在 kafka 的 broker 集群中必须唯一，且为正整数。

3、启动每台电脑上的 kafka 即可。

补充说明：

1、public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String, Integer> topicCountMap)，其中该方法的参数 Map 的 key 为 topic 名称，value 为 topic 对应的分区数，譬如说如果在 kafka 中不存在相应的 topic 时，则会创建一个 topic，分区数为 value，如果存在的话，该处的 value 则不起什么作用

2、关于生产者向指定的分区发送数据，通过设置 partitioner。class 的属性来指定向那个分区发送数据，如果自己指定必须编写相应的程序，默认是 kafka。producer。DefaultPartitioner,分区程序是基于散列的键。

3、在多个消费者读取同一个 topic 的数据，为了保证每个消费者读取数据的唯一性，必须将这些消费者 group_id 定义为同一个值，这样就构建了一个类似队列的数据结构，如果定义不同，则类似一种广播结构的。

4、在 consumer api 中，参数设计到数字部分，类似 Map<String,Integer>,

numStream,指的都是在 topic 不存在的时，会创建一个 topic，并且分区个数为 Integer,numStream,注意如果数字大于 broker 的配置中 num。partitions 属性，会以 num。partitions 为依据创建分区个数的。

5、producer api，调用 send 时，如果不存在 topic，也会创建 topic，在该方法中没有提供分区个数的参数，在这里分区个数是由服务端

broker 的配置中 num。partitions 属性决定的。

http://kafka.apache.org/documentation.html