

Kafka介绍

2017年11月9日 10:11

概述



官方网址：<http://kafka.apache.org/>

以下摘自官网的介绍：

Apache Kafka® is a *distributed streaming platform*. What exactly does that mean?

We think of a streaming platform as having three key capabilities:

1. It lets you publish and subscribe to streams of records. In this respect it is similar to a **message queue** or enterprise messaging system.
2. It lets you **store** streams of records in a **fault-tolerant** way.
3. It lets you process streams of records as they occur.

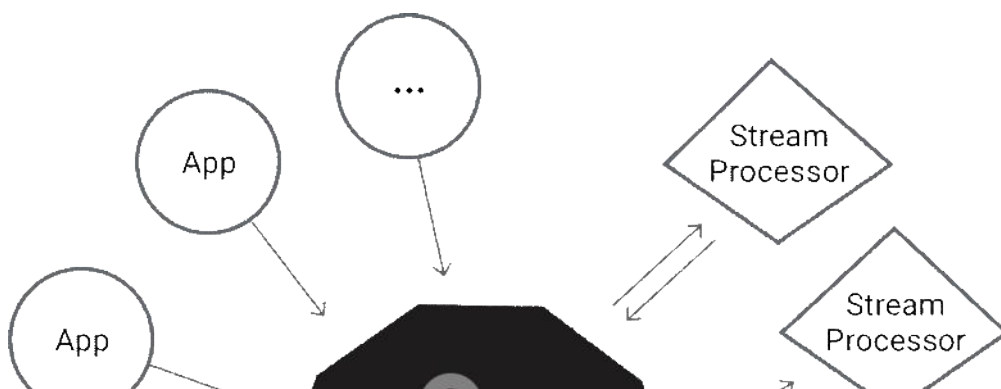
What is Kafka good for?

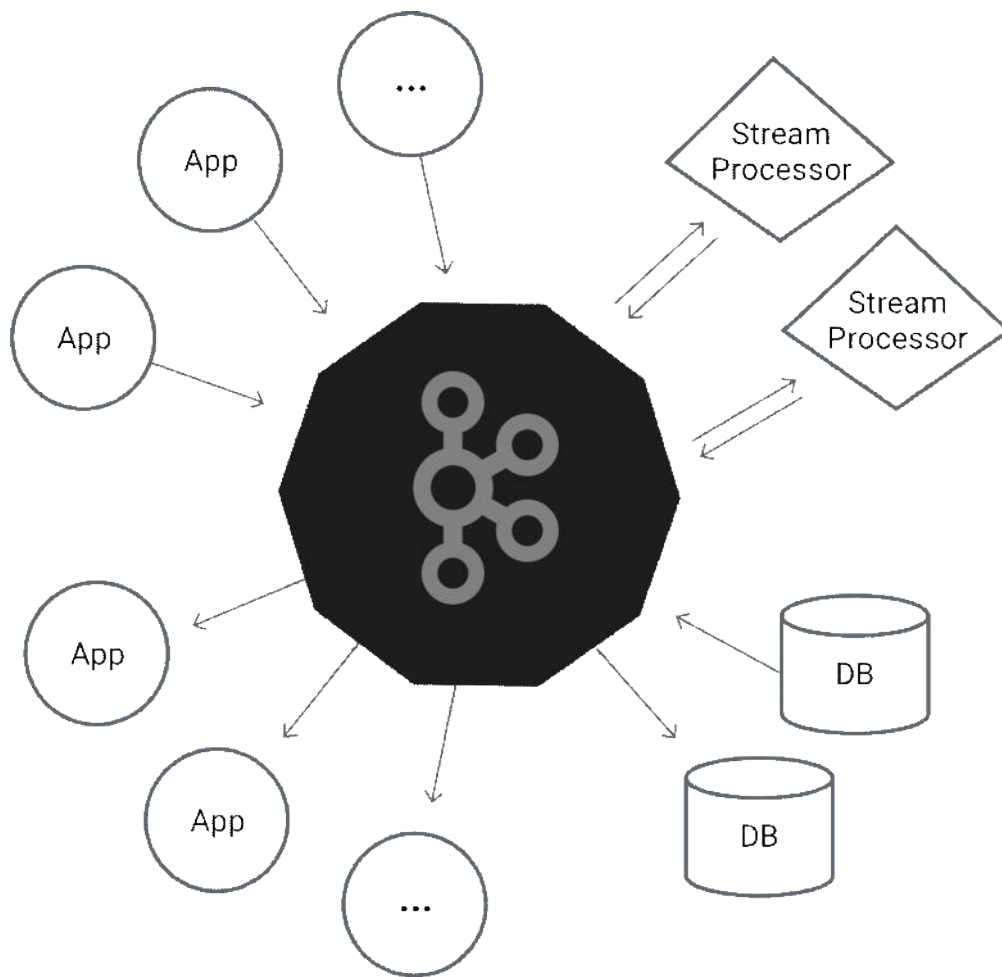
It gets used for two broad classes of application:

1. Building **real-time streaming** data pipelines that reliably **get data** between systems or applications
2. Building **real-time streaming** applications that **transform** or **react** to the streams of data

First a few concepts:

- Kafka is run as a cluster on one or more servers.
- The Kafka cluster stores streams of *records* in categories called *topics*.
- Each record consists of a key, a value, and a timestamp.





Kafka是由LinkedIn开发的一个分布式的消息系统，最初是用作LinkedIn的**活动流**（Activity Stream）和**运营数据**处理的基础。

活动流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。

Kafka使用Scala编写，它以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布

式处理系统如Cloudera、Apache Storm、Spark都支持与Kafka集成。

综上，Kafka是一种分布式的，基于发布/订阅的消息系统，能够高效并实时的吞吐数据，以及通过分布式集群及数据复制冗余机制（副本冗余机制）实现数据的安全

常用Message Queue对比

RabbitMQ

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

Redis

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。

ZeroMQ

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty（NIO）作为传输模块）。

ActiveMQ

ActiveMQ是Apache下的一个子项目。类似于ZeroMQ，它能够以代理人和点对点的技术实

现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

适用场景

Messaging

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.不过到目前为止,我们应该很清楚认识到,kafka并没有提供JMS中的"事务性""消息传输担保(消息确认机制)""消息分组"等企业级特性;kafka只能使用作为"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)

Website 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

Metric

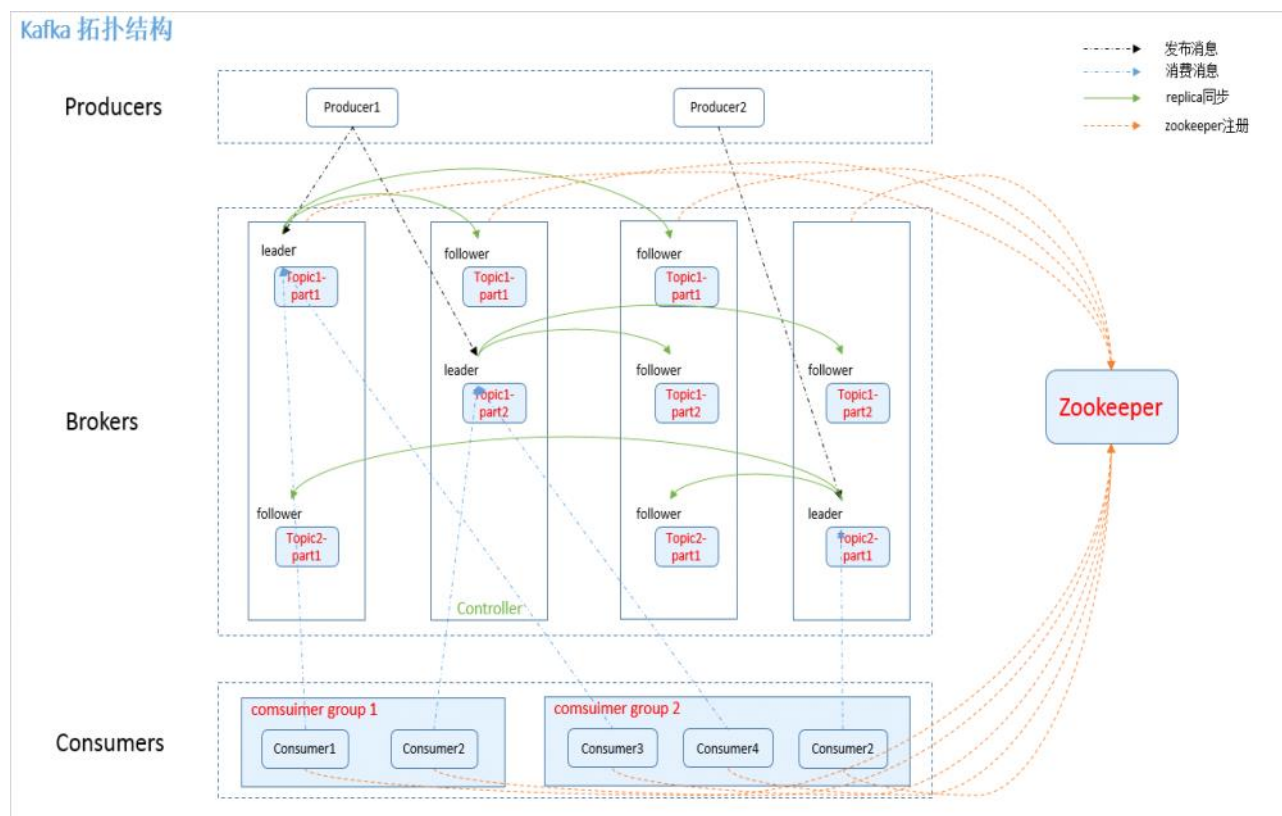
Kafka通常被用于可操作的监控数据。这包括从分布式应用程序来的聚合统计用来生产集中的运营数据提要。

Log Aggregatio

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

Kafka架构

2017年11月21日 19:14



1.producer :

消息生产者，发布消息到 kafka 集群的终端或服务。

2.broker :

kafka 集群中包含的服务器。broker (经纪人，消费转发服务)

3.topic :

每条发布到 kafka 集群的消息属于的类别，即 kafka 是面向 topic 的。

4.partition :

partition 是物理上的概念，每个 topic 包含一个或多个 partition。kafka 分配的单位是 partition。

5.consumer :

从 kafka 集群中消费消息的终端或服务。

6.Consumer group :

high-level consumer API 中，每个 consumer 都属于一个 consumer group，每条消息只能被 consumer group 中的一个 Consumer 消费，但可以被多个 consumer group 消费。

即组间数据是共享的，组内数据是竞争的。

7.replica :

partition 的副本，保障 partition 的高可用。

8.leader :

replica 中的一个角色，producer 和 consumer 只跟 leader 交互。

9.follower :

replica 中的一个角色，从 leader 中复制数据。

10.controller :

kafka 集群中的其中一个服务器，用来进行 leader election 以及 各种 failover。

11.zookeeper :

kafka 通过 zookeeper 来存储集群的 meta 信息。

Kafka集群配置

2017年11月16日 15:09

实现步骤：

0.准备3台虚拟机

1.从官网下载安装包 <http://kafka.apache.org/downloads>

2.上传到01虚拟机，解压

3.进入安装目录下的config目录

4.对server.properties进行配置

配置示例：

broker.id=0

log.dirs=/home/software/kafka/kafka-logs

zookeeper.connect=hadoop01:2181,hadoop02:2181,hadoop03:2181

delete.topic.enable=true

advertised.host.name=192.168.234.21

advertised.port=9092

5.保存退出后，别忘了在安装目录下创建 kafka-logs目录

6.配置其他两台虚拟机，更改配置文件的broker.id编号（不重复即可）

7.先启动zookeeper集群

8.启动kafka集群

进入bin目录

执行：sh kafka-server-start.sh ../config/server.properties

Kafka在Zookeeper下的路径：

```
rmr /cluster
rmr /brokers
rmr /admin
rmr /isr_change_notification
rmr /log_dir_event_notification
rmr /controller_epoch
```

```
rmr /consumers
rmr /latest_producer_id_block
rmr /config
```

Kafka使用

2017年11月16日 15:45

1.创建自定义的topic

在bin目录下执行：

```
sh kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1 --partitions 1 --topic enbook
```

注:副本数量要小于等于节点数量

2.查看所有的topic

执行：sh kafka-topics.sh --list --zookeeper hadoop01:2181

3.启动producer

执行：sh kafka-console-producer.sh --broker-list hadoop01:9092 --topic enbook

4.启动consumer

执行：[root@hadoop01 bin]# sh kafka-console-consumer.sh --zookeeper hadoop01:2181 --topic enbook --from-beginning

5.可以通过producer和consumer模拟消息的发送和接收

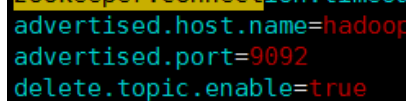
6.删除topic指令：

进入bin目录，执行：sh kafka-topics.sh --delete --zookeeper hadoop01:2181 --topic enbook

可以通过配置 config目录下的 server.properties文件，加入如下的配置：

 配置示例：

delete.topic.enable=true

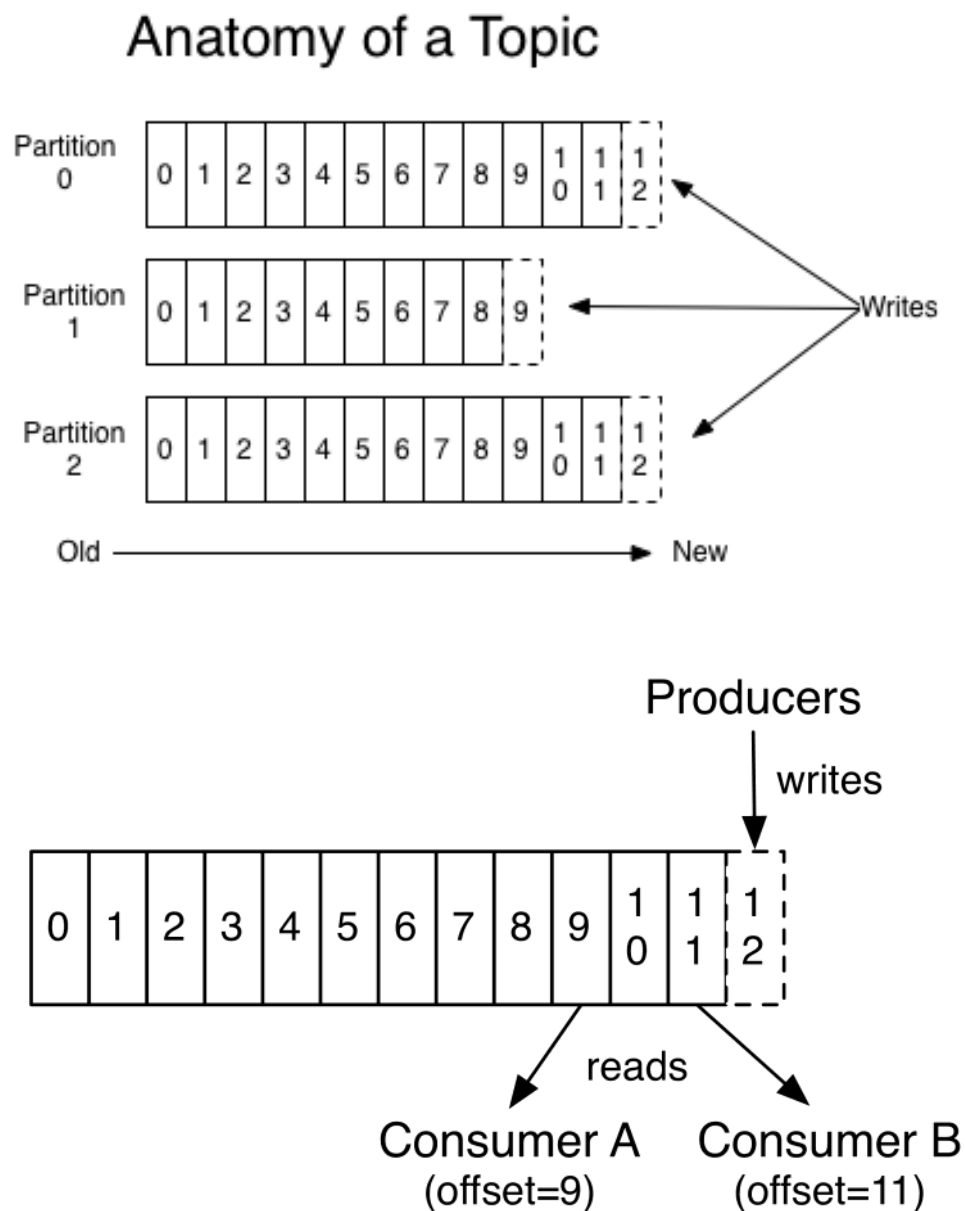


```
advertised.host.name=hadoop01
advertised.port=9092
delete.topic.enable=true
```


Topic与Partition

2017年11月22日 19:28

示意图



Topic

每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic（主题）。

Partition

Partition是物理上的概念，每个Topic包含一个或多个Partition.

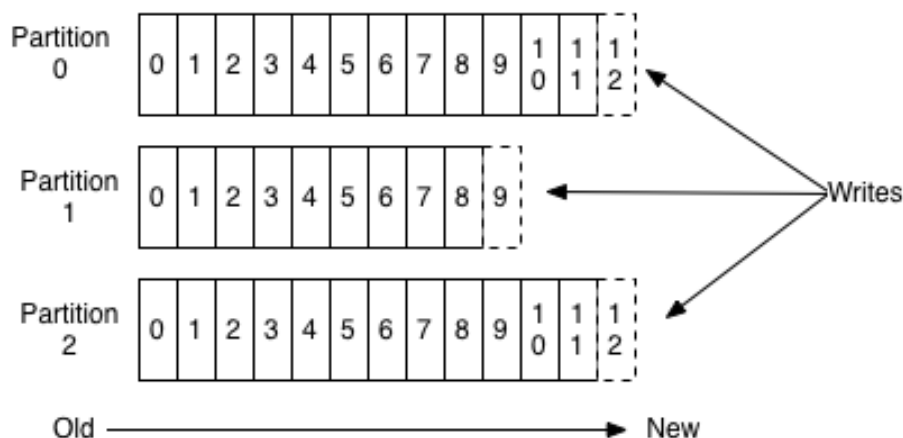
Topic在逻辑上可以被认为是一个queue，每条消息都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。

为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，**每个Partition在物理上对应一个文件夹**，该文件夹下存储这个Partition的所有消息和索引文件。若创建topic1和topic2两个topic，且分别有13个和19个分区，如下图所示。

```
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-2
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-18
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-2
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-8
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-16
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-8
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-6
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-14
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-6
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-5
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-13
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-5
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-9
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-17
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-9
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-4
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-4
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-3
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-3
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-7
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-15
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-7
```

因为每条消息都被append到该Partition中，属于**顺序写磁盘**，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。

Anatomy of a Topic



对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略删除旧数据。一是基于时间，二是基于Partition文件大小。例如可以通过配置 `$KAFKA_HOME/config/server.properties`，让Kafka删除一周前的数据，也可在Partition文件超过1GB时删除旧数据，配置如下所示。



配置示例：

The minimum age of a log file to be eligible for deletion

`log.retention.hours=168`

The maximum size of a log segment file. When this size is reached a new log segment will be created.

`log.segment.bytes=1073741824`

The interval at which log segments are checked to see if they can be deleted according to the

retention policies

`log.retention.check.interval.ms=300000`

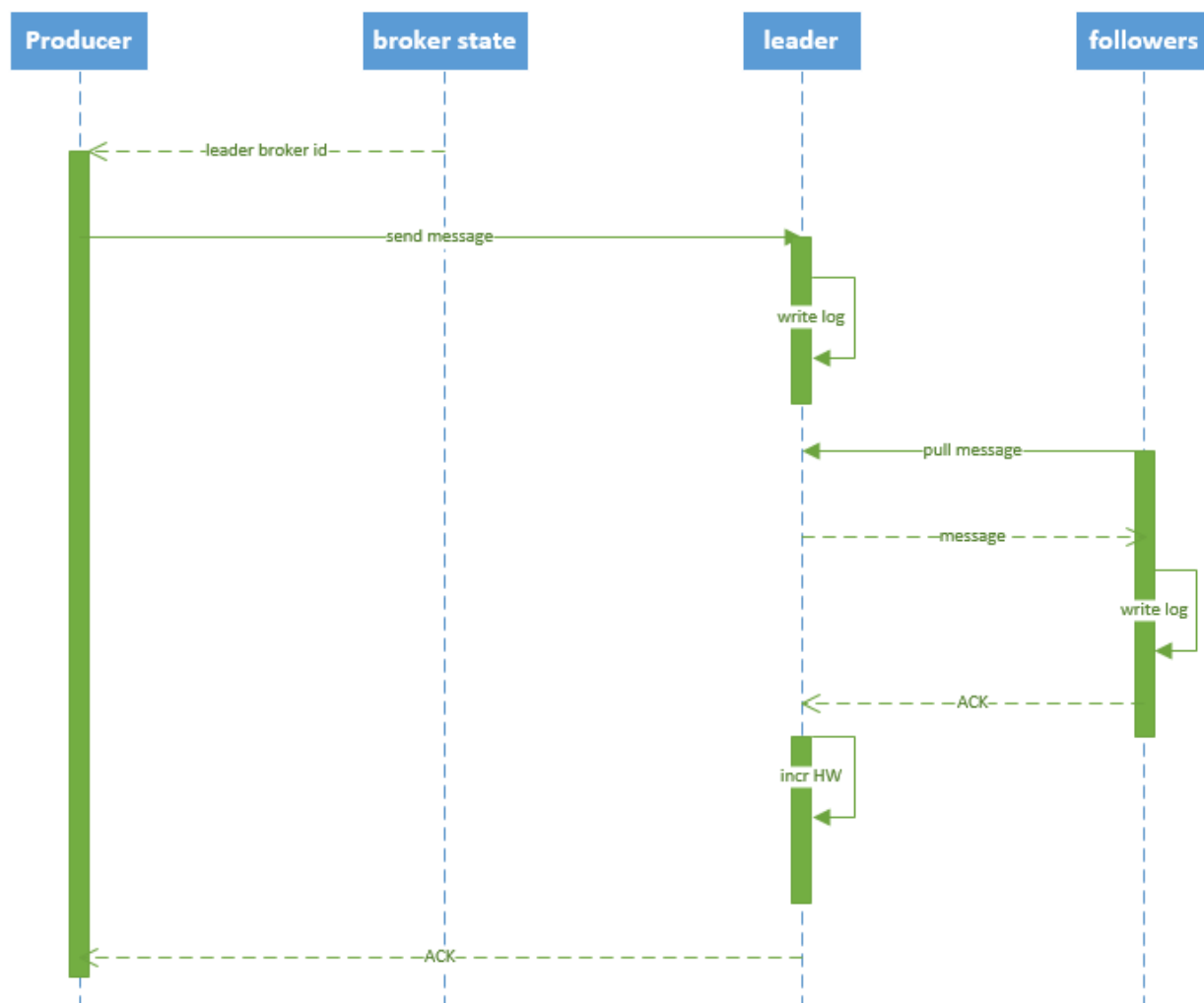
If `log.cleaner.enable=true` is set the cleaner will be enabled and individual logs can then be marked for log compaction.

`log.cleaner.enable=false`

Kafka消息流处理

2017年11月22日 19:35

Producer 写入消息序列图



流程说明：

1. producer 先从 zookeeper 的 `"/brokers/.../state"` 节点找到该 partition 的 leader
2. producer 将消息发送给该 leader
3. leader 将消息写入本地 log
4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK

5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW (high watermark , 最后 commit 的 offset) 并向 producer 发送 ACK。

ISR指的是：比如有三个副本，编号是① ② ③，其中② 是Leader ① ③是Follower。假设在数据同步过程中，①跟上Leader,但是③出现故障或没有及时同步，则① ②是一个ISR，而③不是ISR成员。后期在Leader选举时，会用到ISR机制。会优先从ISR中选择Leader

kafka HA

2017年11月22日 19:47

概述

同一个 partition 可能会有多个 replica (对应 server.properties 配置中的 `default.replication.factor=N`) 。

没有 replica 的情况下，一旦 broker 宕机，其上所有 partition 的数据都不可被消费，同时 producer 也不能再将数据存于其上的 partition。

引入replication 之后，同一个 partition 可能会有多个 replica，而这时需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。

leader failover

当 partition 对应的 leader 宕机时，需要从 follower 中选举出新 leader。在选举新leader时，一个基本的原则是，新的 leader 必须拥有旧 leader commit 过的所有消息。

kafka 在 zookeeper 中 (/brokers/.../state) 动态维护了一个 ISR (in-sync replicas)，由写入流程可知 ISR 里面的所有 replica 都跟上了 leader，只有 ISR 里面的成员才能选为 leader。**对于 $f+1$ 个 replica，一个 partition 可以在容忍 f 个 replica 失效的情况下保证消息不丢失。**

比如 一个分区 有5个副本，挂了4个，剩一个副本，依然可以工作。

注意：kafka的选举不同于zookeeper，用的不是过半选举。

当所有 replica 都不工作时，有两种可行的方案：

1. 等待 ISR 中的任一个 replica 活过来，并选它作为 leader。可保障数据不丢失，但时间可能相对较长。
 2. 选择第一个活过来的 replica (不一定是 ISR 成员) 作为 leader。无法保障数据不丢失，但相对不可用时间较短。
- kafka 0.8.* 使用第二种方式。此外，kafka 通过 Controller 来选举 leader。

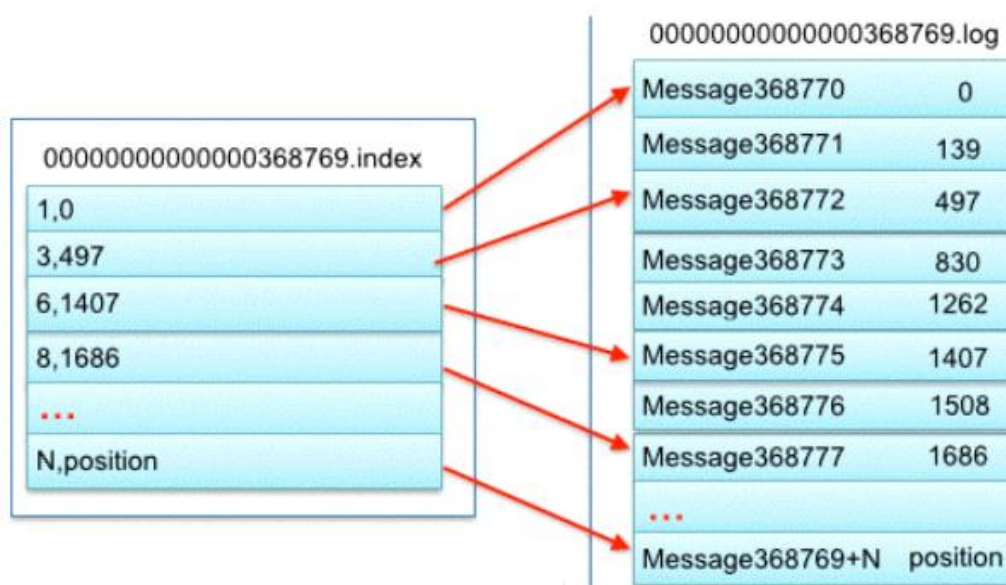
Kafka offset机制

2018年9月6日 20:21

分区数据的Offset存储机制

一个分区在文件系统里存储为一个文件夹。文件夹里包含**日志文件**和**索引文件**。其文件名是其包含的offset的**最小**的条目的offset。

```
00000000000000000000.index
00000000000000000000.log
00000000000000368769.index
00000000000000368769.log
00000000000000737337.index
00000000000000737337.log
```



实际上offset的存储采用了稀疏索引，这样对于稠密索引来说节省了存储空间，但代价是查找费点时间。

Consumer消费者的offset存储机制

Consumer在从broker读消息后，可以选择**commit**，该操作会在kafka中保存该Consumer在该Partition中读取的消息的**offset**。该Consumer下一次再读该Partition时会从下一条开始读取。

通过这一特性可以保证同一消费者从Kafka中不会重复消费数据。

底层实现原理：

执行：sh kafka-console-consumer.sh --bootstrap-server hadoop01:9092 --topic enbook --from-beginning --new-consumer

执行：sh kafka-consumer-groups.sh --bootstrap-server hadoop01:9092 --list --new-consumer

查询得到的group id

console-consumer-60350

进入kafka-logs目录查看，会发现多个很多目录，这是因为kafka默认会生成50个

__consumer_offsets 的目录，用于存储消费者消费的offset位置。

__consumer_offsets-0	__consumer_offsets-34
__consumer_offsets-1	__consumer_offsets-35
__consumer_offsets-10	__consumer_offsets-36
__consumer_offsets-11	__consumer_offsets-37
__consumer_offsets-12	__consumer_offsets-38
__consumer_offsets-13	__consumer_offsets-39
__consumer_offsets-14	__consumer_offsets-4
__consumer_offsets-15	__consumer_offsets-40
__consumer_offsets-16	__consumer_offsets-41
__consumer_offsets-17	__consumer_offsets-42
__consumer_offsets-18	__consumer_offsets-43
__consumer_offsets-19	__consumer_offsets-44
__consumer_offsets-2	__consumer_offsets-45
__consumer_offsets-20	__consumer_offsets-46
__consumer_offsets-21	__consumer_offsets-47

Kafka会使用下面公式计算该消费者group位移保存在__consumer_offsets的哪个目录上：

$\text{Math.abs}(\text{groupID.hashCode()}) \% \text{numPartitions}$

Kafka API使用

2017年12月5日 20:38



旧版代码示意：

```
public class TestDemo {

    @Test

    public void producer(){

        Properties props=new Properties();

        props.put("serializer.class","kafka.serializer.StringEncoder");

        props.put("metadata.broker.list","192.168.234.11:9092");

        Producer<Integer,String> producer=new Producer<>(new ProducerConfig(props));

        producer.send(new KeyedMessage<Integer, String>("enbook","Think in java"));

    }

}
```



新版代码示意：

```
import java.util.Properties;

import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.junit.Test;
```

```

public class TestDemo {

    @Test

    public void producer() throws InterruptedException, ExecutionException{

        Properties props=new Properties();

        props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");

        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");


        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"192.168.234.11:9092");


        Producer<Integer, String> kafkaProducer = new KafkaProducer<Integer, String>(props);

        for(int i=0;i<100;i++){

            ProducerRecord<Integer, String> message = new ProducerRecord<Integer, String>

                ("enbook","",i);

            kafkaProducer.send(message);

        }


        while(true);

    }

}

```



创建Topic代码：

```
@Test
```

```

public void create_topic(){

    ZkUtils zkUtils =

    ZkUtils.apply("192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181",

    30000, 30000, JaasUtils.isZkSecurityEnabled());

    // 创建一个单分区单副本名为t1的topic

    AdminUtils.createTopic(zkUtils, "t1", 1, 1, new Properties(), RackAwareMode.Enforced

    $.MODULE$);

    zkUtils.close();

}

```



删除Topic代码：

@Test

```

public void delete_topic(){

    ZkUtils zkUtils =

    ZkUtils.apply("192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181",

    30000, 30000, JaasUtils.isZkSecurityEnabled());

    // 删除topic 't1'

    AdminUtils.deleteTopic(zkUtils, "t1");

    zkUtils.close();

}

```



创建消费者线程并指定消费者组：

@Test

```

public void consumer_1(){

```

```

Properties props = new Properties();

props.put("bootstrap.servers", "192.168.234.11:9092");

props.put("group.id", "consumer-tutorial");

props.put("key.deserializer", StringDeserializer.class.getName());

props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);


consumer.subscribe(Arrays.asList("enbook", "t2"));


try {

    while (true) {

        ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);

        for (ConsumerRecord<String, String> record : records)

            System.out.println("c1消费:" + record.offset() + ":" + record.value());

    }

} catch (Exception e) {

} finally {

    consumer.close();

}

}

```

@Test

```

public void consumer_2(){

    Properties props = new Properties();

    props.put("bootstrap.servers", "192.168.234.11:9092");

```

```

props.put("group.id", "consumer-tutorial");

props.put("key.deserializer", StringDeserializer.class.getName());

props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);


consumer.subscribe(Arrays.asList("enbook", "t2"));


try {

    while (true) {

        ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);

        for (ConsumerRecord<String, String> record : records)

            System.out.println("c2消费:" + record.offset() + ":" + record.value());

    }

} catch (Exception e) {

} finally {

    consumer.close();

}

}

}

```

扩展：Zero Copy

2017年12月5日 22:09

概述

首先来看一下维基百科对Zero Copy的定义：

(来自 <<https://en.wikipedia.org/wiki/Zero-copy>>)

"**Zero-copy**" describes computer operations in which the [CPU](#) does not perform the task of copying data from one [memory](#) area to another. This is frequently used to **save CPU cycles and memory bandwidth when transmitting a file over a network.**

Zero-copy versions of operating system elements, such as device drivers, file systems, and network protocol stacks, greatly increase the performance of certain application programs and more efficiently utilize system resources. Also, **zero-copy operations reduce the number of time-consuming mode switches between user space and kernel space.**

原理概述

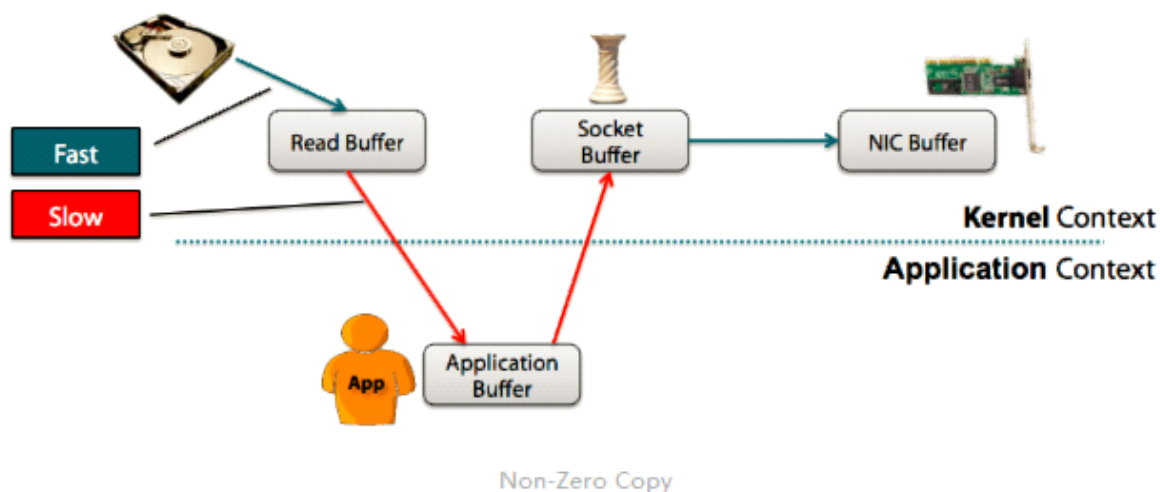
zero copy (零复制) 是一种特殊形式的内存映射，它允许你将Kernel内存直接映射到设备内存空间上。其实就是设备可以通过直接内存访问 (direct memory access , DMA) 方式来访问Kernal Space。

我们拿Kafka举例，Kafka会对流数据做持久化存储以实现容错，比如说一个topic会对应多个Partition，每个Partition实际上最后会落地为一个文件存储在磁盘上，

这意味着当Consumer从Kafka消费数据时，会有很多data从硬盘读出之后，会原封不动的通过socket传输给用户。

这种操作看起来可能不会怎么消耗CPU，但是实际上它是低效的：kernal把数据从disk读出来，然后把它传输给user级的application，然后application再次把同样的内容再传回给处于kernal级的socket。这种场景下，application实际上只是作为一种低效的中间介质，用来把disk file的data传给socket。

如下图所示：



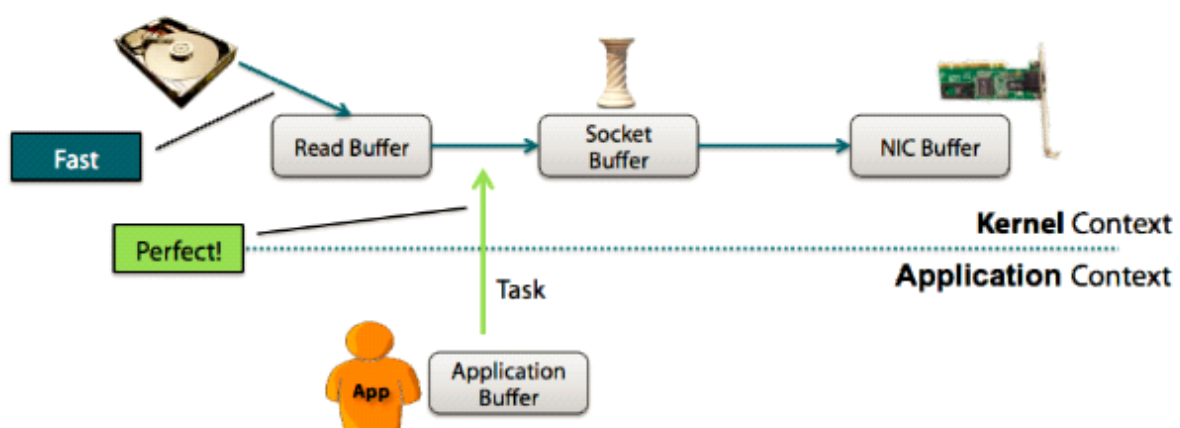
所以，当我们通过网络传输数据时，尽管看起来很简单，但是在OS的内部，这个copy操作要经历四次user mode和kernel mode之间的上下文切换，而**数据都被拷贝了四次**！

此外，data每次穿过user-kernel boundary，都会被copy，这会消耗cpu，并且占用RAM的带宽。

注：随机存取存储器(random access memory，RAM)又称作"随机存储器"，是与CPU直接交换数据的内部存储器，也叫主存(内存)。它可以随时读写，而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储媒介。

Zero Copy的具体实现

实际上第二次和第三次copy是毫无意义的。应用程序仅仅缓存了一下data就原封不动的把它发回给socket buffer。实际上，data应该直接在read buffer和socket buffer之间传输，如下图：

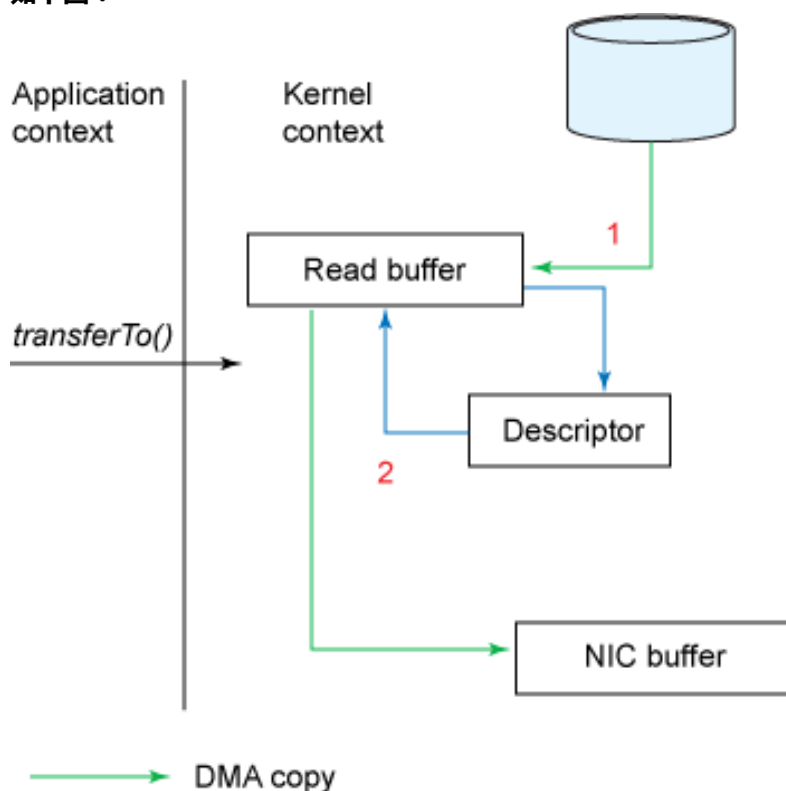


从上图中可以清楚的看到，Zero Copy的模式中，避免了数据在用户空间和内存空间之间的拷贝，从而提高了系统的整体性能。Linux中的sendfile()以及Java NIO中的FileChannel.transferTo()方法都实现了零拷贝的功能，而在Netty中也通过在FileRegion中包装了NIO的FileChannel.transferTo()方法实现了零拷贝。

这是一个很明显的进步：我们把context switch的次数从4次减少到了2次，同时也把data copy的次数从4次降低到了3次（而且其中只有一次占用了CPU，另外两次由DMA完成）。但是，要做到真正的zero copy，这还差一些。

如果网卡支持 gather operation，我们可以通过kernel进一步减少数据的拷贝操作。在2.4及以上版本的linux内核中，开发者修改了socket buffer descriptor来适应这一需求。这个方法不仅减少了context switch，还消除了和CPU有关的数据拷贝。

如下图：



最新的Zero Copy的机制是追加了一些descriptor的信息，包括data的位置和长度。然后DMA 直接把data从kernel buffer传输到protocol engine，这样就消除了唯一的一次需要占用CPU的拷贝操作。

为什么要使用kernel buffer做中介

使用kernel buffer做中介(而不是直接把data传到user buffer中)看起来比较低效(多了一次copy)。然而实际上kernel buffer是用来提高性能的。在进行读操作的时候，kernel buffer起到了预读cache的作用。当写请求的data size比kernel buffer的size小的时候，这能够显著的提升性能。在进行写操作时，kernel buffer的存在可以使得写请求完全异步。

但悲剧的是，当读请求的data size远大于kernel buffer size的时候，这个方法本身变成了性能的瓶颈。

扩展：信息检索技术

2017年9月30日 10:47

概念介绍

全文检索是一种将文件中所有文本与检索项匹配的文字资料检索方法。全文检索系统是按照全文检索理论建立起来的用于提供全文检索服务的软件系统。

全文检索主要对**非结构化数据**的数据检索。

结构化数据和非结构化数据

结构化数据：指具有固定格式或有限长度的数据，如数据库，元数据等。

非结构化数据：指不定长或无固定格式的数据，如邮件，word文档，网页等。

当然有的地方还会提到第三种，半结构化数据，如XML，HTML等，当根据需要可按结构化数据来处理，也可抽取纯文本按非结构化数据来处理。

注：非结构化数据另外一种叫法叫：**全文数据**。

数据搜索

按照数据的分类，搜索也分为两种：

对结构化数据的搜索：如对数据库的搜索，用SQL语句。再如对元数据的搜索，如利用windows搜索对文件名，类型，修改时间进行搜索等。

对非结构化数据的搜索：如利用windows的搜索也可以搜索文件内容，Linux下的grep命令，再如用Google和百度可以搜索大量内容数据。

我们重点来探讨对非结构化数据的搜索。

顺序扫描法

所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。

比如：利用windows的搜索也可以搜索文件内容，如果做全盘文件的检索，速度会相当的慢，因为硬盘上的数据很大。Linux下的grep命令也是这一种方式。

大家可能觉得这种方法比较原始，但对于小数据量的文件，这种方法还是最直接，最方便的。但是对于大量的文件，这种方法就很慢了。

有人可能会说，对非结构化数据顺序扫描很慢，对结构化数据的搜索却相对较快（由于结构化数据有一定的结构可以采取一定的搜索算法加快速度），那么把我们的非结构化数据想办法弄得有一定结构不就行了吗？

这种想法很天然，却构成了全文检索的基本思路，也即将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。

这部分从非结构化数据中提取出的然后重新组织的信息，我们称之为索引。

索引与全文检索



比如字典，对每一个字的解释是非结构化的，如果字典没有音节表和部首检字表，在茫茫辞海中找一个字只能顺序扫描。

所以，字典的拼音表和部首检字表就相当于字典的索引，每一项读音都指向此字的详细解释的页数。我们搜索时按结构化的拼音搜到读音，然后按其指向的页数，便可找到我们的非结构化数据——也即对字的解释。

这种先建立索引，再对索引进行搜索的过程就叫全文检索(Full-text Search)。

下面这幅图来自《Lucene in action》，但却不仅仅描述了Lucene的检索过程，而是描述了全文检索的一般过程。

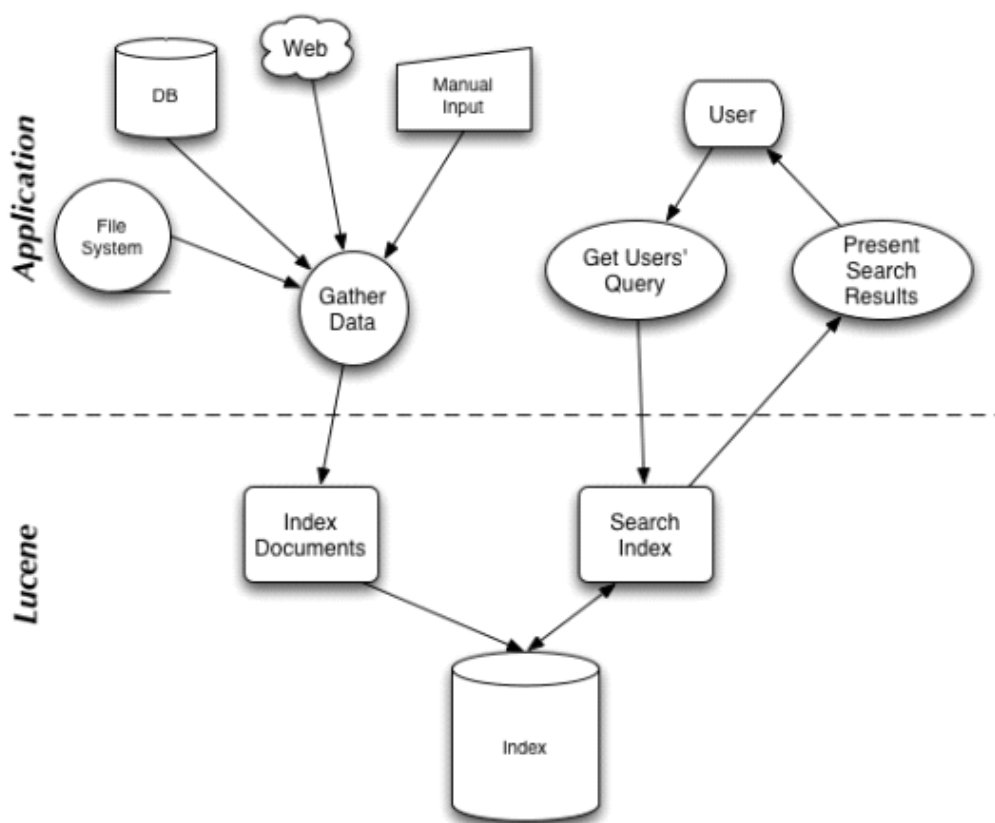


Figure 1.5 A typical application integration with Lucene

全文检索原理

全文检索大体分两个过程，**创建索引(Indexing)**和**搜索索引(Search)**。

索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。

搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

正向索引

已知文件，欲检索数据，这是建立：文件——数据的映射，称为正向索引，比如下图：

data.txt的索引表



data.txt

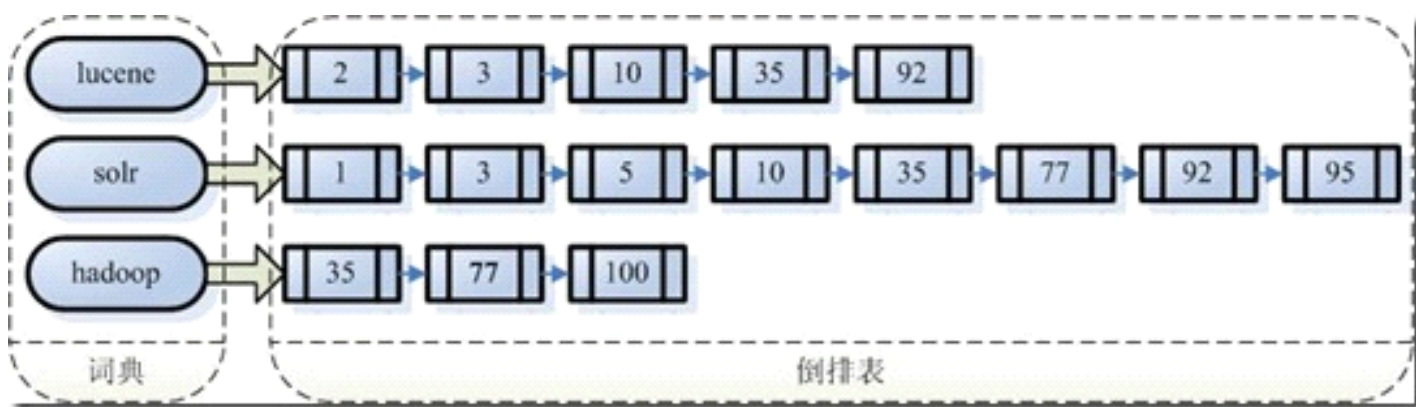
单词	出现的频次
hello	5
hadoop	6
lucene	2
the	20
this	14

反向索引

在大多数的应用中，我们想做的是搜索某个数据都出现在了哪些文件里或网页里

这是已知数据，欲检索文件，这是建立：数据——文件的映射，称为**反向索引**，又称**倒排索引**。

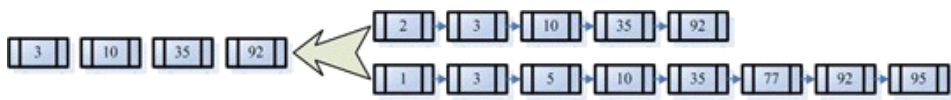
假如我们有100篇文章，想查看一下lucene,hadoop,solr 在哪些文章中出现过，如下图：



左边保存的是一系列字符数据，称为**词典**。每个字符串都指向包含此字符串的文档(Document)链表，此文档链表称为**倒排表**(Posting List)。

比如我们要寻找既包含字符串“lucene”又包含字符串“solr”的文档，我们只需要以下步骤：

1. 取出包含字符串“lucene”的文档链表。
2. 取出包含字符串“solr”的文档链表。
3. 通过合并链表，找出既包含“lucene”又包含“solr”的文件。



注意：全文检索的确加快了搜索的速度，但是多了索引的过程，两者加起来不一定比顺序扫描快多少。尤其是在数据量小的时候更是如此。并且对一个很大的数据创建索引也是一个很慢的过程。

然而两者还是有区别的，顺序扫描是每次都要扫描，而创建索引的过程仅仅需要一次，以后便是一劳永逸的了，每次搜索，创建索引的过程不必经过，仅仅搜索创建好的索引就可以了。

这也是全文搜索相对于顺序扫描的优势之一：一次索引，多次使用。

如何创建索引

全文检索的索引创建过程一般有以下几步：

1.第一步：一些要索引的原文档(Document)。

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to go out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

2.第二步：将原文档传给分词组件(Tokenizer)。

分词组件(Tokenizer)会做以下几件事情(此过程称为Tokenize)：

1. 将文档分成一个一个单独的单词。
2. 去除标点符号。
3. 去除停词(Stop word)。

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中挺词(Stop word)如：“the”，“a”，“this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

经过分词(Tokenizer)后得到的结果称为词元(Token)。

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，
“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，
“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

对于英语，语言处理组件(Linguistic Processor)一般做以下几点：

1. 变为小写(Lowercase)。
2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。
3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。

补充：语言处理组件(linguistic processor)的结果称为词(Term)。

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，
“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，
“student”，“find”，“them”，“drink”，“allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

第四步：将得到的词(Term)传给索引组件(Indexer)。

索引组件(Indexer)主要做以下几件事情：

- 4.1. 利用得到的词(Term)创建一个字典。

在我们的例子中字典如下：

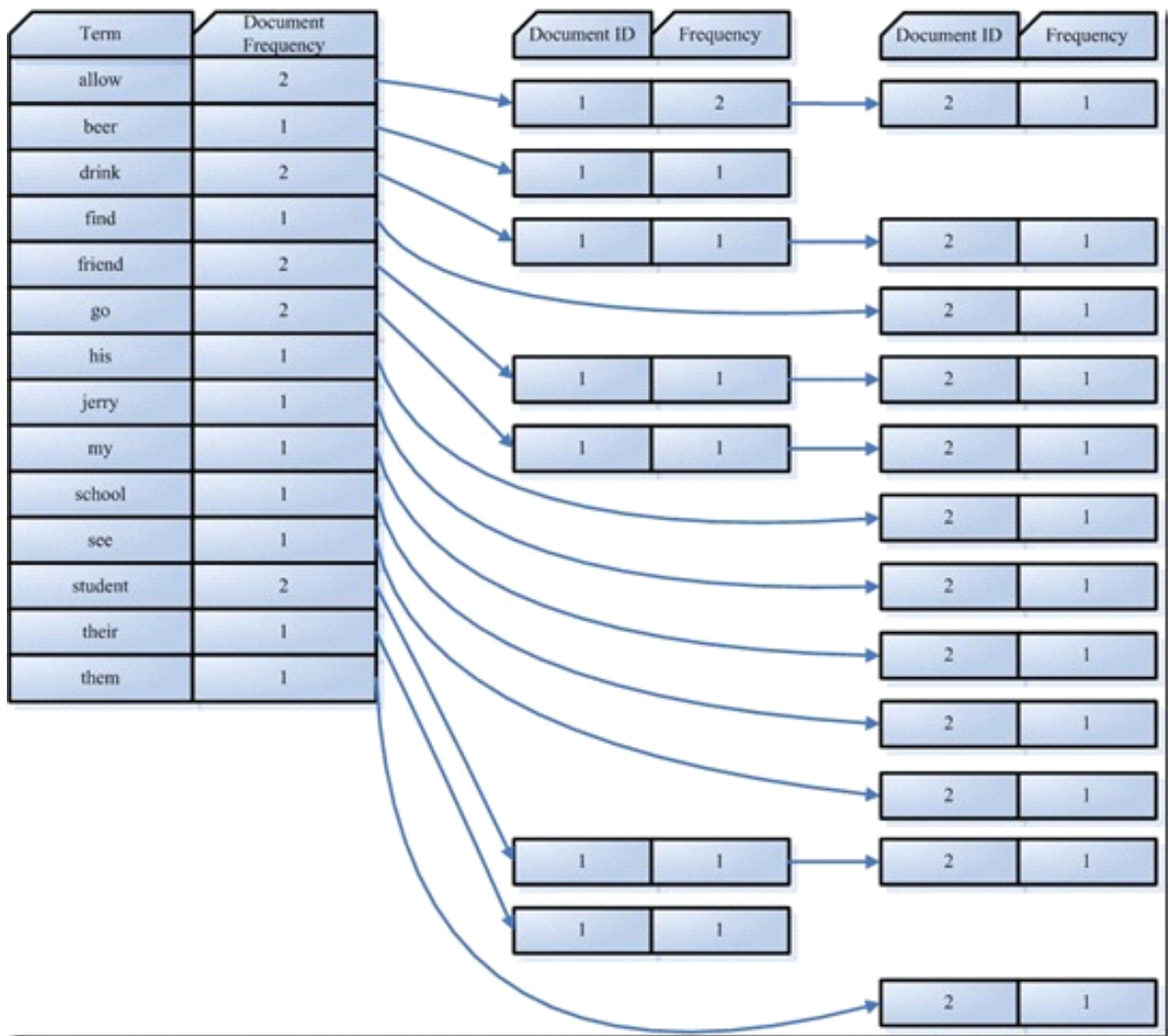
Term	Document ID
student	1
allow	1
go	1
their	1
friend	1
allow	1
drink	1
beer	1
my	2
friend	2
jerry	2

go	2
school	2
see	2
his	2
student	2
find	2
them	2
drink	2
allow	2

4.2 对字典按字母顺序进行排序。

Term	Document ID
allow	1
allow	1
allow	2
beer	1
drink	1
drink	2
find	2
friend	1
friend	2
go	1
go	2
his	2
jerry	2
my	2
school	2
see	2
student	1
student	2
their	1
them	2

4.3 合并相同的词(Term)成为文档倒排(Posting List)链表。



在此表中，有几个定义：

Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。

Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow” 来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

扩展：向量空间模型算法(Vector Space Model)

2017年9月30日 17:20

概念介绍

向量空间模型 (VSM: Vector Space Model) 由Salton等人于20世纪70年代提出，并成功地应用于文本检索系统。

VSM概念简单，把对文本内容的处理简化为向量空间中的向量运算，并且它以**空间上的相似度表达语义的相似度，直观易懂**。当文档被表示为文档空间的向量，**就可以通过计算向量之间的相似性来度量文档间的相似性**。文本处理中最常用的相似性度量方式是**余弦距离**。

M个无序特征项 t_i ，词根/词/短语/其他每个文档 d_j 可以用特征项向量来表示 ($a_{1j}, a_{2j}, \dots, a_{Mj}$) 权重计算，N个训练文档 $A_{M \times N} = (a_{ij})$ 文档相似度比较

向量空间模型 (或词组向量模型) 是一个应用于信息过滤，信息抽取，索引以及评估相关性的代数模型。

算法原理

1. 计算权重(Term weight)的过程。

影响一个词(Term)在一篇文档中的重要性主要有两个因素：

Term Frequency (tf)：即此Term在此文档中出现了多少次。tf 越大说明越重要。

Document Frequency (df)：即有多少文档包含此Term。df 越大说明越不重要。

词(Term)在文档中出现的次数越多，说明此词(Term)对该文档越重要，如“搜索”这个词，在本文档中出现的次数很多，说明本文档主要就是讲这方面的事的。然而在一篇英语文档中，this出现的次数更多，就说明更重要吗？不是的，这是由第二个因素进行调整，第二个因素说

明，有越多的文档包含此词 (Term)，说明此词 (Term) 太普通，不足以区分这些文档，因而重要性越低。

我们来看一下模型公式：

$$W_{t,d} = tf_{t,d} \log\left(\frac{n}{df_t}\right)$$

说明：

$w_{t,d}$ = the weight of the term t in document d

$tf_{t,d}$ = frequency of term t in document d

n = total number of documents

df_t = the number of documents that contain term t

这仅仅只term weight计算公式的简单典型实现。实现全文检索系统的人会有自己的实现，Lucene就与此稍有不同。

2. 判断Term之间的关系从而得到文档相关性的过程，也即向量空间模型的算法 (VSM)。

我们把文档看作一系列词 (Term)，每一个词 (Term) 都有一个权重 (Term weight)，不同的词 (Term) 根据自己在文档中的权重来影响文档相关性的打分计算。

于是我们把所有此文档中词 (term) 的权重 (term weight) 看作一个向量。

Document = {term1, term2, , term N}

Document Vector = {weight1, weight2, , weight N}

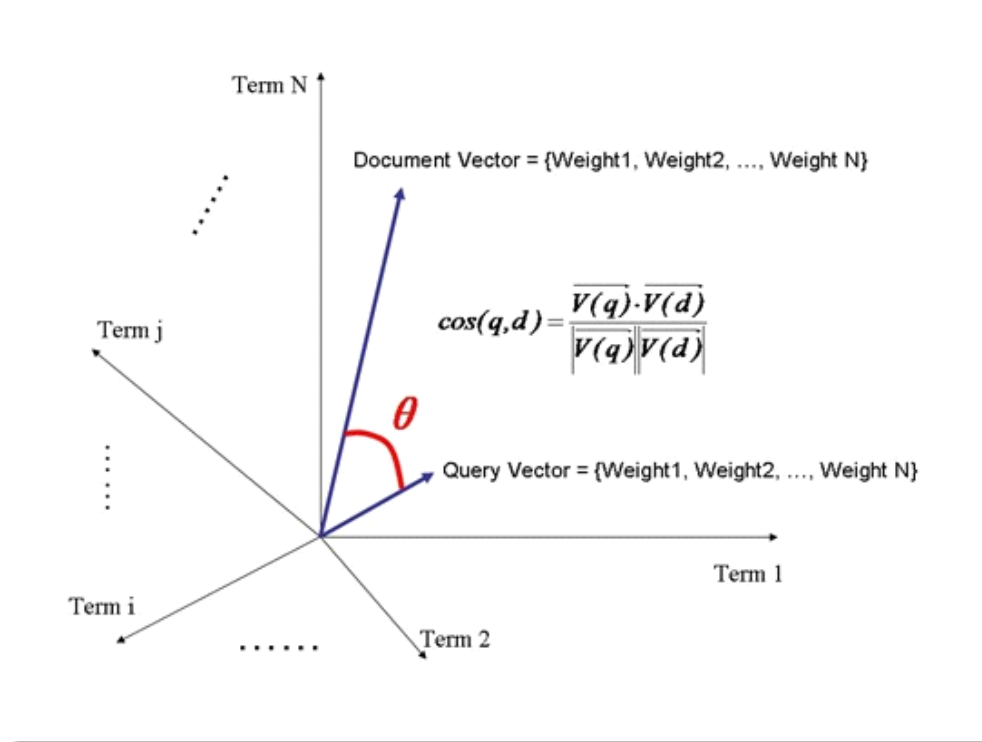
同样我们把查询语句看作一个简单的文档，也用向量来表示。

Query = {term1, term 2, , term N}

Query Vector = {weight1, weight2, , weight N}

我们把所有搜索出的文档向量及查询向量放到一个N维空间中，每个词 (term) 是一维。

如图：



我们认为两个向量之间的夹角越小，相关性越大。

所以我们计算夹角的余弦值作为相关性的打分，夹角越小，余弦值越大，打分越高，相关性越大。

相关性打分公式如下

$$score(q,d) = \frac{\vec{V}_q \cdot \vec{V}_d}{\|\vec{V}_q\| \|\vec{V}_d\|} = \frac{\sum_{i=1}^n w_{i,q} w_{i,d}}{\sqrt{\sum_{i=1}^n w_{i,q}^2} \sqrt{\sum_{i=1}^n w_{i,d}^2}}$$
$$\cos(\theta) = \frac{\sum_{k=1}^n x_{1k} x_{2k}}{\sqrt{\sum_{k=1}^n x_{1k}^2} \sqrt{\sum_{k=1}^n x_{2k}^2}}$$

举个例子，查询语句有11个Term，共有三篇文档搜索出来。其中各自的权重(Term weight)，如下表格。

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
D1	0	0	.477	0	.477	.176	0	0	0	.176	0
D2	0	.176	0	.477	0	0	0	0	.954	0	.176

D3	0	.176	0	0	0	.176	0	0	0	.176	.176
Q	0	0	0	0	0	.176	0	0	.477	0	.176

于是计算，三篇文档同查询语句的相关性打分分别为：

$$SC(Q,D_1) = \frac{(0.176)(0.176)}{\sqrt{0.477^2 + 0.477^2 + 0.176^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.08$$

$$SC(Q,D_2) = \frac{(0.954)(0.477) + (0.176)^2}{\sqrt{0.176^2 + 0.477^2 + 0.954^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.825$$

$$SC(Q,D_3) = \frac{(0.176)^2 + (0.176)^2}{\sqrt{0.176^2 + 0.176^2 + 0.176^2 + 0.176^2} \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.327$$

于是文档二相关性最高，先返回，其次是文档一，最后是文档三。

到此为止，我们可以找到我们最想要的文档了。