KAFKA 系列解读

魏小军

2014-6-25

目 录

序论

apache kafka在数据处理中特别是日志和消息的处理上会有很多出色的表现.首先当然推荐的是kafka的官网 http://kafka.apache.org/。在官网最值得参考的文章就是 kafka design：http://kafka.apache.org/design.html，要特别重视这篇文章，里面有好多理念都特别好，推荐多读几遍。

在 OSC 的翻译频道有 kafka design 全中文的翻译，翻得挺好的，推荐一下：http://www.oschina.net/translate/kafka-design。kafka 的 wiki 是很不错的学习文档：https://cwiki.apache.org/confluence/display/KAFKA/Index

接下来就是一系列文章，文章都是循序渐进的方式带你了解 kafka：

- 关于 kafka 的基本知识，分布式的基础：《分布式消息系统 Kafka 初步》
- kafka 的分布式搭建，quick start：《kafka 分布式环境搭建》
- 关于 kafka 的实现细节，这主要就是讲 design 的部分：《细节上》、《细节下》
- 关于 kafka 开发环境，scala 环境的搭建：《开发环境搭建》
- 数据生产者，producer 的用法：《producer 的用法》、《producer 使用注意》
- 数据消费者，consumer 的用法：《consumer 的用法》
- 还有些零碎的，关于通信段的源码解读：《net 包源码解读》、《broker 配置》

扩展的阅读还有下面这些：

- 关于 kafka 和 jafka 的相关博客，特别好，有很多问题也都找他解决的，大神一般的存在：http://rockybean.github.com/@rockybean
- kafka 的 java 化版本 jafka：https://github.com/adyliu/jafka
- 淘宝的 metaQ：https://github.com/killme2008/Metamorphosis
- 最近在写的 inforQ，刚开始写，也纯粹是为了读下源码，不定期更新：https://github.com/ielts0909/inforq

0.8 版本的相关更新如下：

0.8 更新内容介绍：《kafka0.8 版本的一些更新》

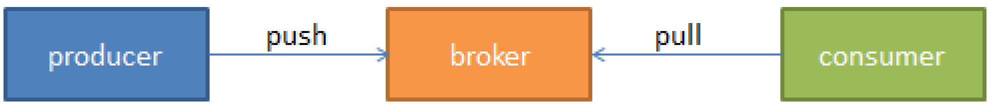
第一章　分布式消息系统 KAFKA 初识

从这一篇开始分布式消息系统的入门。在我们大量使用分布式数据库、分布式计算集群的时候，是否会遇到这样的一些问题：

- 我想分析一下用户行为（pageviews），以便我能设计出更好的广告位
- 我想对用户的搜索关键词进行统计，分析出当前的流行趋势。这个很有意思，在经济学上有个长裙理论，就是说，如果长裙的销量高了，说明经济不景气了，因为姑娘们没钱买各种丝袜了。
- 有些数据，我觉得存数据库浪费，直接存硬盘又怕到时候操作效率低。

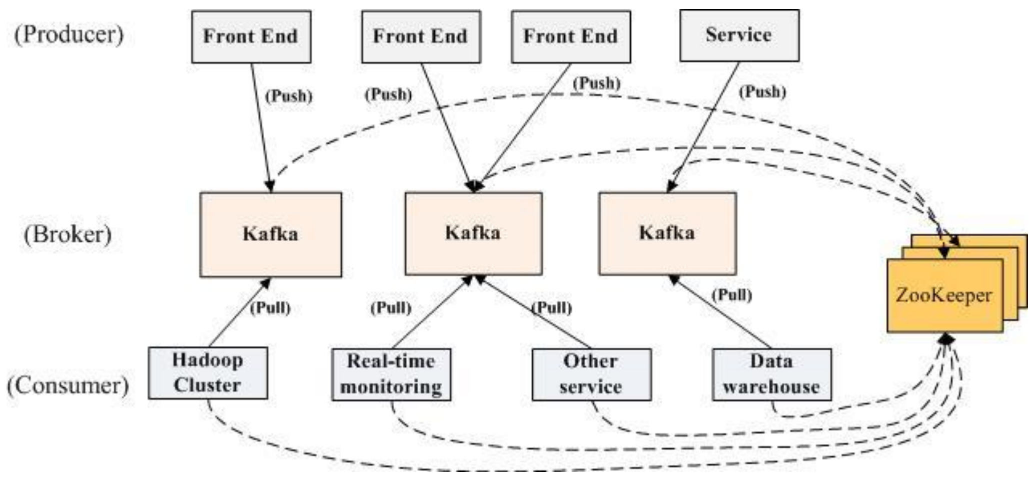
这个时候，我们就可以用到分布式消息系统了。虽然上面的描述更偏向于一个日志系统，但确实 kafka 在实际应用中被大量的用于日志系统。首先我们要明白什么是消息系统，在 kafka 官网上对 kafka 的定义叫：A distributed publish-subscribe messaging system(一个分布式发布-订阅消息传递系统)。publish-subscribe 是发布和订阅的意思，所以更准确的说 kafka 是一个消息订阅和发布的系统。publish- subscribe 这个概念很重要，因为 kafka 的设计理念就可以从这里说起。

我们将消息的发布（publish）暂时称作 producer，将消息的订阅（subscribe）表述为 consumer，将中间的存储阵列称作 broker(代理)，这样我们就可以大致描绘出这样一个场面：



生产者（蓝色，蓝领么，总是辛苦点儿）将数据生产出来，丢给 broker 进行存储，消费者需要消费数据了，就从 broker 中去拿出数据来，然后完成一系列对数据的处理。

乍一看这也太简单了，不是说了它是分布式么，难道把 producer、broker 和 consumer 放在三台不同的机器上就算是分布式了么。我们看 kafka 官方给出的图：



多个 broker 协同合作，producer 和 consumer 部署在各个业务逻辑中被频繁的调用，三者通过 zookeeper 管理协调请求和转发。这样一个高性能的分布式消息发布与订阅系统就完成了。图上有个细节需要注意，producer 到 broker 的过程是 push，也就是有数据就推送到 broker，而 consumer 到 broker 的过程是 pull，是通过 consumer 主动去拉数据的，而不是 broker 把数据主动发送到 consumer 端的。

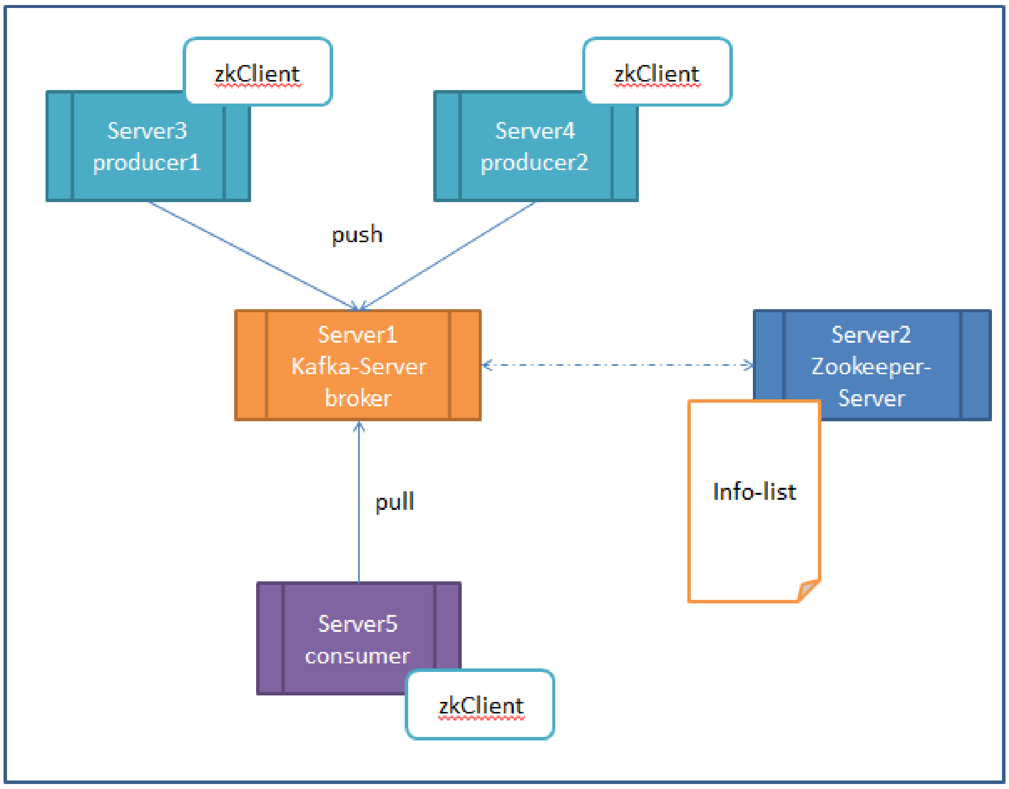
这样一个系统到底在哪里体现出了它的高性能，我们看官网上的描述：

- Persistent messaging with O(1) disk structures that provide constant time performance even with many TB of stored messages.
- High-throughput: even with very modest hardware Kafka can support hundreds of thousands of messages per second.
- Explicit support for partitioning messages over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
- Support for parallel data load into Hadoop.

至于为什么会有 O(1)这样的效率，为什么能有很高的吞吐量我们在后面的文章中都会讲述，今天我们主要关注的还是 kafka 的设计理念。了解完了性能，我们来看下 kafka 到底能用来做什么，除了我开始的时候提到的之外，我们看看 kafka 已经实际在跑的，用在哪些方面：

- LinkedIn - Apache Kafka is used at LinkedIn for activity stream data and operational metrics. This powers various products like LinkedIn Newsfeed, LinkedIn Today in addition to our offline analytics systems like Hadoop.
- Tumblr - http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html
- Mate1.com Inc. - Apache kafka is used at Mate1 as our main event bus that powers our news and activity feeds, automated review systems, and will soon power real time notifications and log distribution.
- Tagged - Apache Kafka drives our new pub sub system which delivers real-time events for users in our latest game - Deckadence. It will soon be used in a host of new use cases including group chat and back end stats and log collection.
- Boundary - Apache Kafka aggregates high-flow message streams into a unified distributed pubsub service, brokering the data for other internal systems as part of Boundary's real-time network analytics infrastructure.
- DataSift - Apache Kafka is used at DataSift as a collector of monitoring events and to track user's consumption of data streams in real time. http://highscalability.com/blog/2011/11/29/datasift-architecture-realtime-datamining-at-120000-tweets-p.html
- Wooga - We use Kafka to aggregate and process tracking data from all our facebook games (which are hosted at various providers) in a central location.
- AddThis - Apache Kafka is used at AddThis to collect events generated by our data network and broker that data to our analytics clusters and real-time web analytics platform.
- Urban Airship - At Urban Airship we use Kafka to buffer incoming data points from mobile devices for processing by our analytics infrastructure.
- Metamarkets - We use Kafka to collect realtime event data from clients, as well as our own internal service metrics, that feed our interactive analytics dashboards.
- SocialTwist - We use Kafka internally as part of our reliable email queueing system.
- Countandra - We use a hierarchical distributed counting engine, uses Kafka as a primary speedy interface as well as routing events for cascading counting
- FlyHajj.com - We use Kafka to collect all metrics and events generated by the users of the website.

至此你应该对 kafka 是一个什么样的系统有所体会，并能了解他的基本结构，还有就是他能用来做什么。那么接下来，我们再回到 producer、consumer、broker 以及 zookeeper 这四者的关系中来。



我们看上面的图，我们把 broker 的数量减少，只有一台。现在假设我们按照上图进行部署：

- Server-1 broker 其实就是 kafka 的 server，因为 producer 和 consumer 都要去连它。Broker 主要还是做存储用。
- Server-2 是 zookeeper 的 server 端，zookeeper 的具体作用你可以去官网查，在这里你可以先想象，它维持了一张表，记录了各个节点的 IP、端口等信息（以后还会讲到，它里面还存了 kafka 的相关信息）。
- Server-3、4、5 他们的共同之处就是都配置了 zkClient，更明确的说，就是运行前必须配置 zookeeper 的地址，道理也很简单，这之间的连接都是需要 zookeeper 来进行分发的。
- Server-1 和 Server-2 的关系，他们可以放在一台机器上，也可以分开放，zookeeper 也可以配集群。目的是防止某一台挂了。

简单说下整个系统运行的顺序：

1. 启动 zookeeper 的 server
2. 启动 kafka 的 server
3. Producer 如果生产了数据，会先通过 zookeeper 找到 broker，然后将数据存放进 broker
4. Consumer 如果要消费数据，会先通过 zookeeper 找对应的 broker，然后消费。

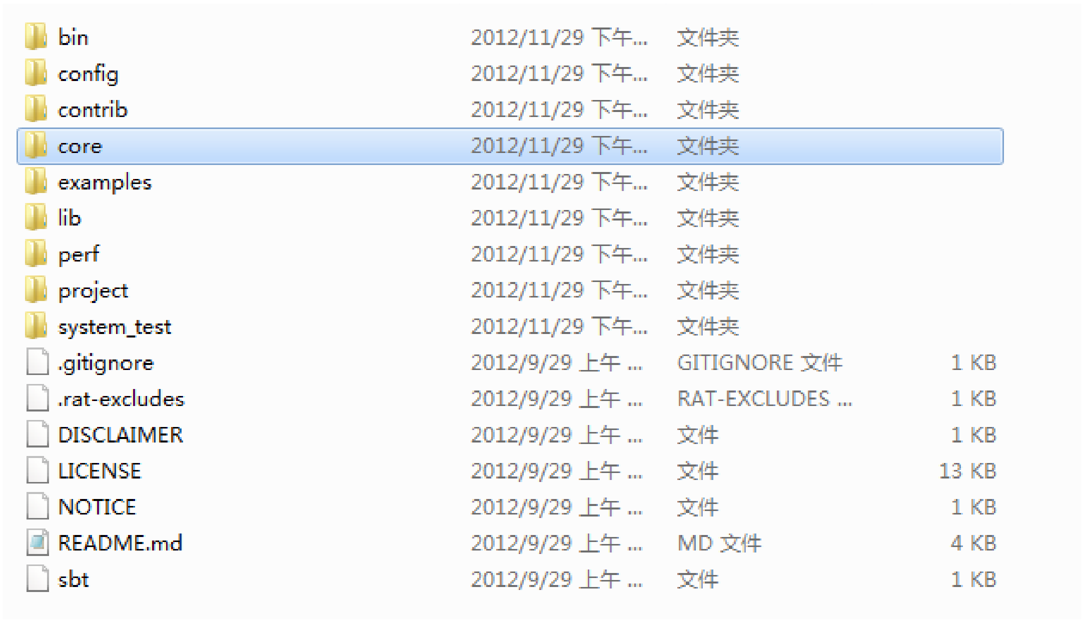
对 kafka 的初步认识就写到这里，接下去我会写如何搭建 kafka 的环境。

第二章　KAFKA 分布式环境搭建

这篇文章将介绍如何搭建 kafka 环境，我们会从单机版开始，然后逐渐往分布式扩展。单机版的搭建官网上就有，比较容易实现，这里我就简单介绍下即可，而分布式的搭建官网却没有描述，我们最终的目的还是用分布式来解决问题，所以这部分会是重点。

Kafka 的中文文档并不多，所以我们尽量详细点儿写。要交会你搭建分布式其实很简单，手把手的教程大不了我录个视频就好了，可我觉得那不是走这条路的方式。只有真正了解原理，并且理解的透彻了才能最大限度的发挥一个框架的作用。所以，如果你不了解什么事 kafka，请先看：《kafka 初步》

我们从搭建单机版的环境开始说起,如果你喜欢看英文版：这里有官方的《quick start》，单机版的部署很简单，我就讲几点比较重要的，首先 kafka 是用 scala 编写的，可以跑在 JVM 上，所以我们并不需要单独去搭建 scala 的环境，后面会涉及到编程的时候我们再说如何去配置 scala 的问题，这里用不到，当然你要知道这个是跑在 linux 上的。第二，我用的是最新版 0.7.2 的版本，你下载完 kafka 你可以打开它的目录浏览一下：



bin	2012/11/29 下午...	文件夹	
config	2012/11/29 下午...	文件夹	
contrib	2012/11/29 下午...	文件夹	
core	2012/11/29 下午...	文件夹	
examples	2012/11/29 下午...	文件夹	
lib	2012/11/29 下午...	文件夹	
perf	2012/11/29 下午...	文件夹	
project	2012/11/29 下午...	文件夹	
system_test	2012/11/29 下午...	文件夹	
.gitignore	2012/9/29 上午 ...	GITIGNORE 文件	1 KB
.rat-excludes	2012/9/29 上午 ...	RAT-EXCLUDES ...	1 KB
DISCLAIMER	2012/9/29 上午 ...	文件	1 KB
LICENSE	2012/9/29 上午 ...	文件	13 KB
NOTICE	2012/9/29 上午 ...	文件	1 KB
README.md	2012/9/29 上午 ...	MD 文件	4 KB
sbt	2012/9/29 上午 ...	文件	1 KB

我就不介绍每个包里的内容是干嘛的，我就着重说一点，你在这个文件夹里只能找到 3 个 jar 包，并且这 3 个还不能用于后面的编程，而且你也没法在里面找到 pom 这样用于构建的 xml。也别急，也别满世界找，这些依赖库得等你把它放到 linux 上才会出现（当然需要命令）。

搭建单机版环境，简单的说有那么几步：

1. 安装 java 环境，我用的是最新的版本 jdk7 的
2. 将下载下来的 kafka 扔到 linux 上，并解压。我用的 red het server 的 linux。
3. 接下来就是下载 kafka 的依赖包和构建 kafka 的环境。注意，这一步需要电脑联网。具体命令就是官网介绍的./sbt update 和 ./sbt package。

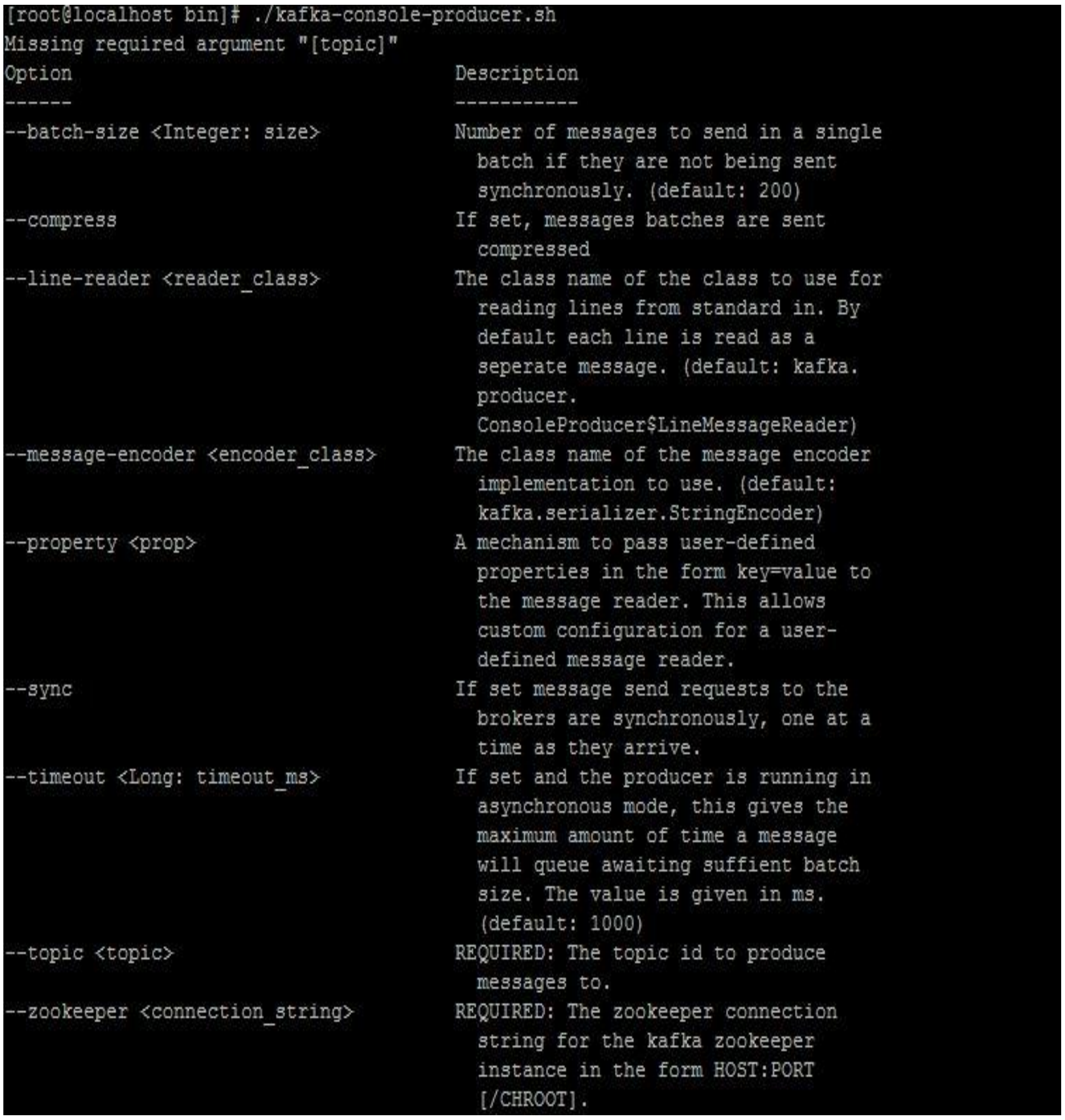
执行完上面这步大概会花个 10 多分钟吧，我在自己家里 ubuntu 没有成功，报了下载不到 jline 的错。单位里用虚拟机 ubuntu 成功了，我深刻怀疑是网的问题。上面这不执行完了有两点要注意，一是 sbt 帮你下载完了所有依赖库，但是这些 jar 都是分散在各个目录下的，注意区分。二是，这些 jar 一部分是 kafka 的编程包，一部分是 scala 的环境包，上面说了没必要自己去搭 scala 的环境道理就在这边，你自己去下一个 2.9 的 scala，但人家 kafka 只支持 2.8、2.7。所以编程的时候就用 sbt 给你下好的包即可。后面讲到编程的时候，会写怎么搭编程环境，很简单的。

上面的步骤都执行完了，环境算是好了，下面我们要测试下是否能成功运行 kafka：

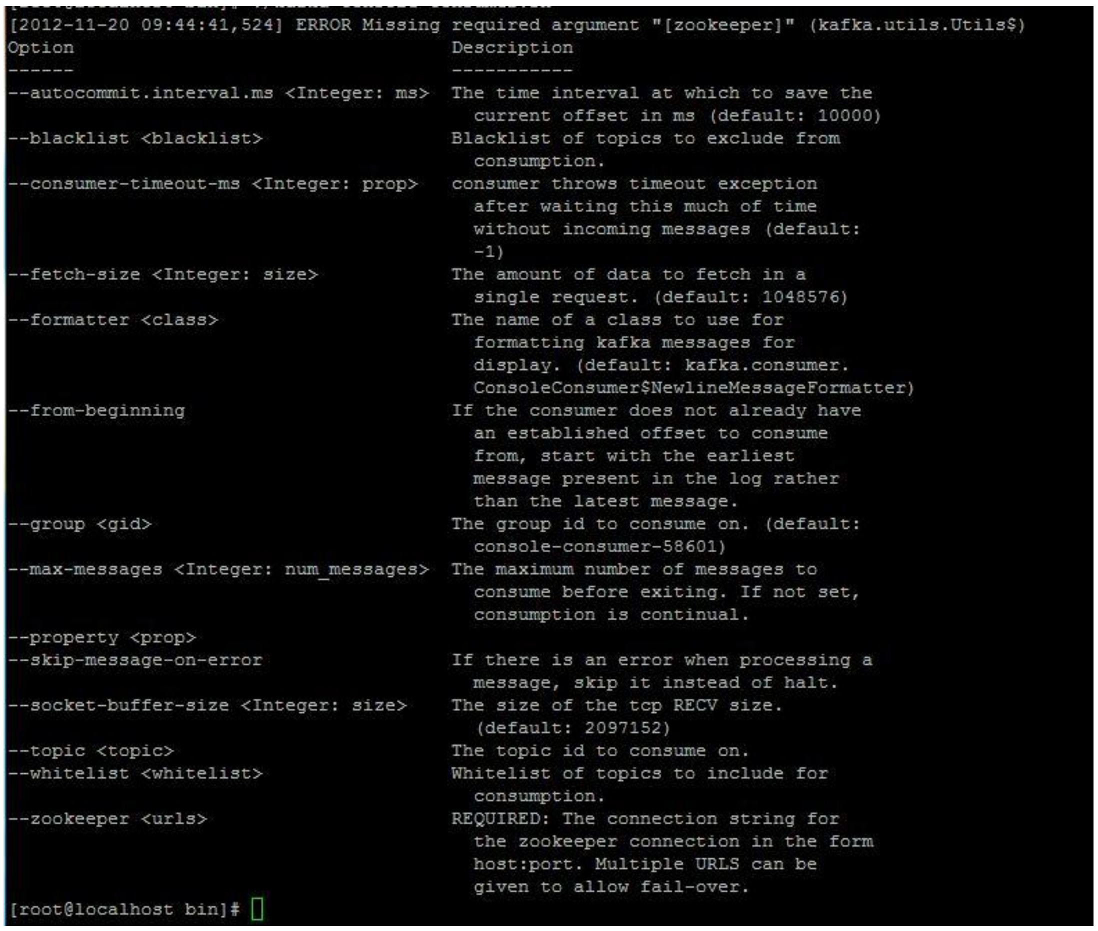
1. 启动 zookeeper server ：bin/zookeeper-server-start.sh ../config/zookeeper.properties & (用&是为了能退出命令行)
2. 启动 kafka server: bin/kafka-server-start.sh ../config/server.properties &
3. Kafka 为我们提供了一个 console 来做连通性测试，下面我们先运行 producer：bin/kafka-console-producer.sh --zookeeper localhost:2181 --topic test 这是相当于开启了一个 producer 的命令行。命令行的参数我们一会儿再解释。
4. 接下来运行 consumer，新启一个 terminal：bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
5. 执行完 consumer 的命令后，你可以在 producer 的 terminal 中输入信息，马上在 consumer 的 terminal 中就会出现你输的信息。有点儿像一个通信客户端。

具体可看《quick start》。

如果你能看到 5 执行了，说明你单机版部署成功了。下面解释下两条命令中参数的意思。--zookeeper localhost:2181 这个说明了去连本机 2181 端口的 zookeeper server，--topic test，在 kafka 里，消息按 topic 来区分，我们这里的 topic 叫 test，所以不管是 consumer 还是 producer 都指向了 test。其他的参数，我就截图了，首先是 producer 的参数：


```
[root@localhost bin]# ./kafka-console-producer.sh
Missing required argument "[topic]"
Option                                  Description
------                                  -----------
--batch-size <Integer: size>            Number of messages to send in a single
                                          batch if they are not being sent
                                          synchronously. (default: 200)
--compress                              If set, messages batches are sent
                                          compressed
--line-reader <reader_class>            The class name of the class to use for
                                          reading lines from standard in. By
                                          default each line is read as a
                                          seperate message. (default: kafka.
                                          producer.
                                          ConsoleProducer$LineMessageReader)
--message-encoder <encoder_class>       The class name of the message encoder
                                          implementation to use. (default:
                                          kafka.serializer.StringEncoder)
--property <prop>                       A mechanism to pass user-defined
                                          properties in the form key=value to
                                          the message reader. This allows
                                          custom configuration for a user-
                                          defined message reader.
--sync                                  If set message send requests to the
                                          brokers are synchronously, one at a
                                          time as they arrive.
--timeout <Long: timeout_ms>            If set and the producer is running in
                                          asynchronous mode, this gives the
                                          maximum amount of time a message
                                          will queue awaiting suffient batch
                                          size. The value is given in ms.
                                          (default: 1000)
--topic <topic>                         REQUIRED: The topic id to produce
                                          messages to.
--zookeeper <connection_string>         REQUIRED: The zookeeper connection
                                          string for the kafka zookeeper
                                          instance in the form HOST:PORT
                                          [/CHROOT].
```

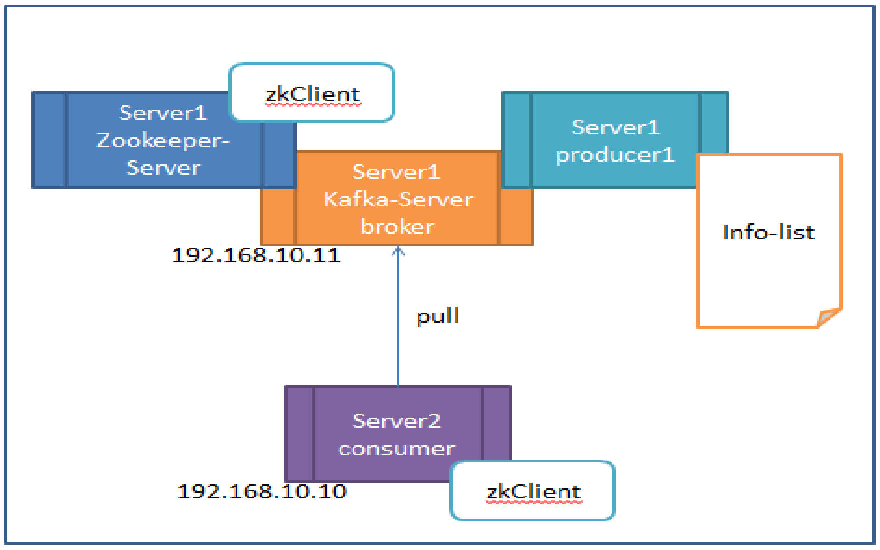
Consumer 的参数:



```
[2012-11-20 09:44:41,524] ERROR Missing required argument "[zookeeper]" (kafka.utils.Utils$)
Option                                  Description
------                                  -----------
--autocommit.interval.ms <Integer: ms>  The time interval at which to save the
                                          current offset in ms (default: 10000)
--blacklist <blacklist>                 Blacklist of topics to exclude from
                                          consumption.
--consumer-timeout-ms <Integer: prop>   consumer throws timeout exception
                                          after waiting this much of time
                                          without incoming messages (default:
                                          -1)
--fetch-size <Integer: size>            The amount of data to fetch in a
                                          single request. (default: 1048576)
--formatter <class>                     The name of a class to use for
                                          formatting kafka messages for
                                          display. (default: kafka.consumer.
                                          ConsoleConsumer$NewlineMessageFormatter)
--from-beginning                        If the consumer does not already have
                                          an established offset to consume
                                          from, start with the earliest
                                          message present in the log rather
                                          than the latest message.
--group <gid>                           The group id to consume on. (default:
                                          console-consumer-58601)
--max-messages <Integer: num_messages>  The maximum number of messages to
                                          consume before exiting. If not set,
                                          consumption is continual.
--property <prop>
--skip-message-on-error                 If there is an error when processing a
                                          message, skip it instead of halt.
--socket-buffer-size <Integer: size>    The size of the tcp RECV size.
                                          (default: 2097152)
--topic <topic>                         The topic id to consume on.
--whitelist <whitelist>                 Whitelist of topics to include for
                                          consumption.
--zookeeper <urls>                      REQUIRED: The connection string for
                                          the zookeeper connection in the form
                                          host:port. Multiple URLS can be
                                          given to allow fail-over.
[root@localhost bin]#
```

这些参数你可以先看个大概，之后会在编程中使用到，都可以动态的配置。

好了单机版就部署完了，那是不是把 consumer 的放到另一台机器上就算分布式了呢。是的，前提是，你还能运行到上面的第 5 步。在讲配置之前,我们还是将上篇写的分布式来回顾一下，当然我们简化一下情况，按照实际部署的分析：

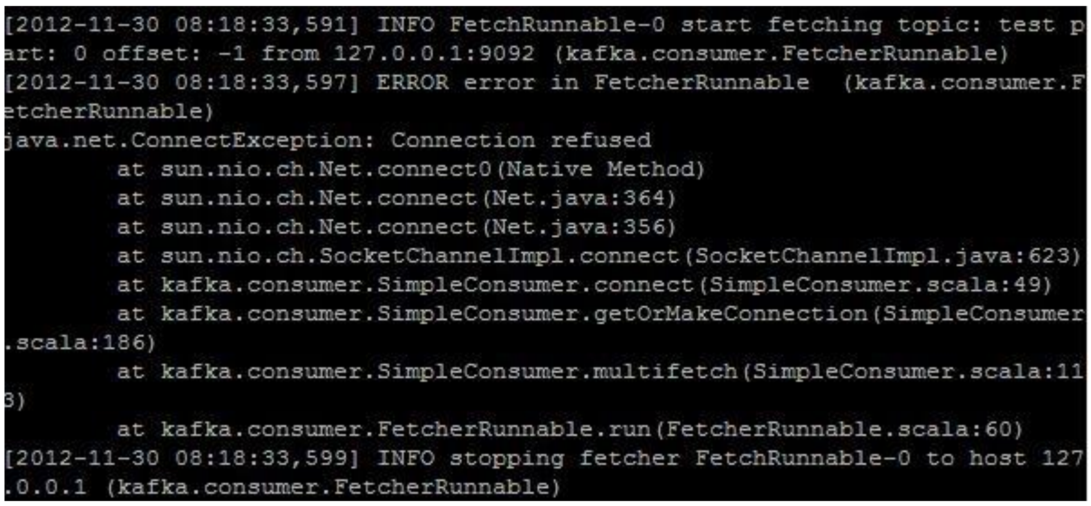


假设我只有两台机器，server1 和 server2。我现在想把 zookeeper server 和 kafka server 和 producer 都放在一台机器上 server1，把 consumer 放在 server2 上。这当然也叫分布式了，虽然机子不多，但是这个部署成功了，扩展是相当的容易。

我们还是按照那 5 个步骤来做，当然你肯定能知道，3、4 两步的参数要改了，我们假设 server1 的 IP 是 192.168.10.11 server2 的 IP 是 192.168.10.10：

- 启动 zookeeper server ：bin/zookeeper-server-start.sh ../config/zookeeper.properties & (用&是为了能退出命令行)
- 启动 kafka server: bin/kafka-server-start.sh ../config/server.properties &
- Kafka 为我们提供了一个 console 来做连通性测试，下面我们先运行 producer：bin/kafka-console-producer.sh --zookeeper 192.168.10.11:2181 --topic test 这是相当于开启了一个 producer 的命令行。
- 接下来运行 consumer，新启一个 terminal：bin/kafka-console-consumer.sh --zookeeper 192.168.10.11:2181 --topic test --from-beginning
- 执行完 consumer 的命令后，你可以在 producer 的 terminal 中输入信息，马上在 consumer 的 terminal 中就会出现你输的信息。

这个时候你能执行出第 5 步的效果么，是不是报了下面的错了：



```
[2012-11-30 08:18:33,591] INFO FetchRunnable-0 start fetching topic: test p
art: 0 offset: -1 from 127.0.0.1:9092 (kafka.consumer.FetcherRunnable)
[2012-11-30 08:18:33,597] ERROR error in FetcherRunnable  (kafka.consumer.F
etcherRunnable)
java.net.ConnectException: Connection refused
        at sun.nio.ch.Net.connect0(Native Method)
        at sun.nio.ch.Net.connect(Net.java:364)
        at sun.nio.ch.Net.connect(Net.java:356)
        at sun.nio.ch.SocketChannelImpl.connect(SocketChannelImpl.java:623)
        at kafka.consumer.SimpleConsumer.connect(SimpleConsumer.scala:49)
        at kafka.consumer.SimpleConsumer.getOrMakeConnection(SimpleConsumer
.scala:186)
        at kafka.consumer.SimpleConsumer.multifetch(SimpleConsumer.scala:11
3)
        at kafka.consumer.FetcherRunnable.run(FetcherRunnable.scala:60)
[2012-11-30 08:18:33,599] INFO stopping fetcher FetchRunnable-0 to host 127
.0.0.1 (kafka.consumer.FetcherRunnable)
```

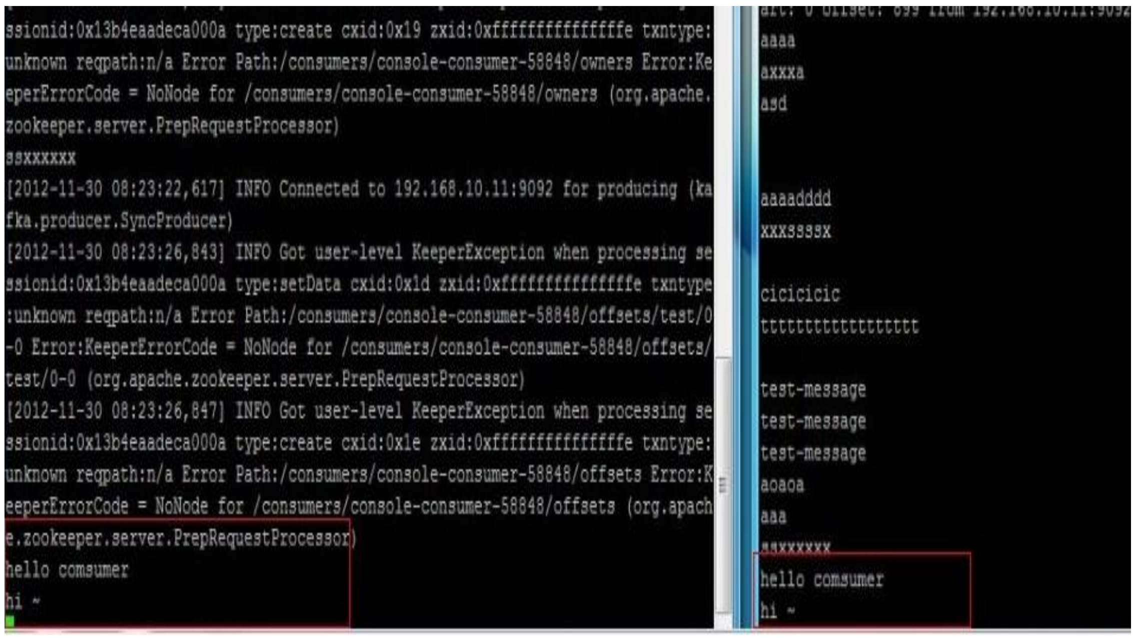
我来说原因，在这之前想请你再回去看看《kafka 初步》，看看里面讲分布式的内容。

这里的 kafka server 就是 broker，broker 是存数据的，producer 把数据给 broker，consumer 从 broker 取数据。那 zookeeper 是干嘛的，说的肤浅点儿，zookeeper 就是他们之间的选择分发器，所有的连接都要先注册到 zookeeper 上。你可以把它想象成 NIO，zookeeper 就是 selector，producer、consumer 和 broker 都要注册到 selector 上，并且留下了相应的 key。所以问题就出在了 kafka server 的配置 server.properties 上。Kafka 注册到 zookeeper 上的信息不对，才导致了上面的错误。我们看 config 中 server.properties 的配置就可以知道：

```
# Hostname the broker will advertise to consumers. If not set, kafka will use the value returned

# from InetAddress.getLocalHost().   If there are multiple interfaces getLocalHost

# may not be what you want.

#hostname=
```

默认的 hostname 如果你不设置，就是 127.0.0.1，所以你把这个 hostname 设置成 192.168.10.11 即可，这样你重启下 kafka server 端，就能执行第 5 步了。

成功配置的话，你能看到下面的效果，左边的是 producer，右边的是 consumer，看最下面两行好了，前面的是我之前测试用过的：



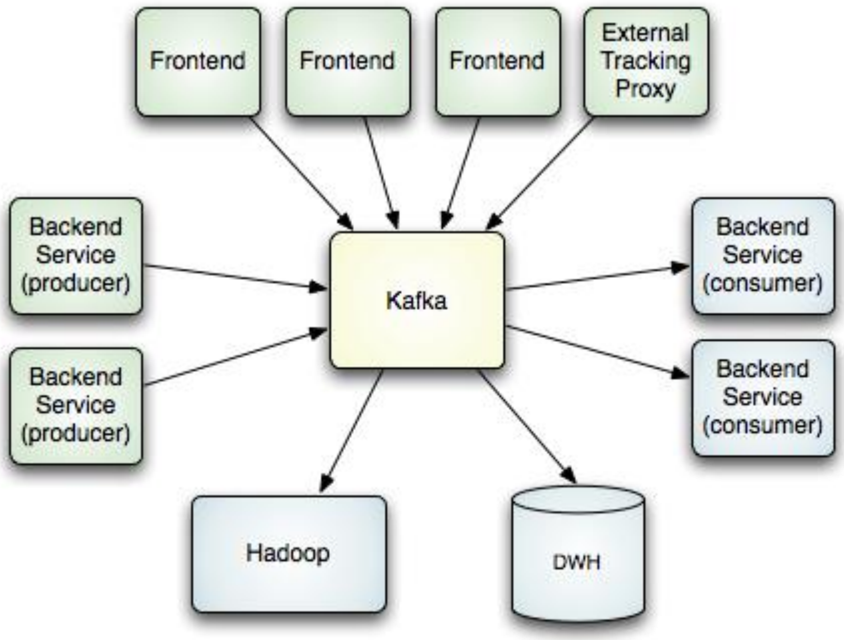
```
ssionid:0x13b4eaadeca000a type:create cxid:0x19 zxid:0xfffffffffffffffe txntype:
unknown reqpath:n/a Error Path:/consumers/console-consumer-58848/owners Error:Ke
eperErrorCode = NoNode for /consumers/console-consumer-58848/owners (org.apache.
zookeeper.server.PrepRequestProcessor)
ssxxxxxx
[2012-11-30 08:23:22,617] INFO Connected to 192.168.10.11:9092 for producing (ka
fka.producer.SyncProducer)
[2012-11-30 08:23:26,843] INFO Got user-level KeeperException when processing se
ssionid:0x13b4eaadeca000a type:setData cxid:0x1d zxid:0xfffffffffffffffe txntype
:unknown reqpath:n/a Error Path:/consumers/console-consumer-58848/offsets/test/0
-0 Error:KeeperErrorCode = NoNode for /consumers/console-consumer-58848/offsets/
test/0-0 (org.apache.zookeeper.server.PrepRequestProcessor)
[2012-11-30 08:23:26,847] INFO Got user-level KeeperException when processing se
ssionid:0x13b4eaadeca000a type:create cxid:0x1e zxid:0xfffffffffffffffe txntype:
unknown reqpath:n/a Error Path:/consumers/console-consumer-58848/offsets Error:K
eeperErrorCode = NoNode for /consumers/console-consumer-58848/offsets (org.apach
e.zookeeper.server.PrepRequestProcessor)
hello comsumer
hi ~
```

```
art: 0 offset: 899 from 192.168.10.11:9092
aaaa
axxxa
asd


aaaadddd
xxxsssx

cicicicic
tttttttttttttttttt

test-message
test-message
test-message
aoaoa
aaa
ssxxxxxx
hello comsumer
hi ~
```

如果你还是云里雾里的，建议你回头去看看上篇文章，将 kafka 分布式基本原理的，kafka 实际操作是要建立在对原理熟悉的情况下的。搭建完了环境，后面就要开始写程序去处理实际问题了。当然再写程序之前，下一篇我会先写一点 kafka 为什么会有如此高的性能，它是怎么保障这些性能的。

第三章　KAFKA 实现细节（上）

如果你第一次看 kafka 的文章，请先看《分布式消息系统 kafka 初步》，之前有人问 kafka 和一般的 MQ 之间的区别，这个问题挺难回答，我觉得不如从 kafka 的实现原理来分析更为透彻，这篇将依据官网上给出的 design 来详细的分析，kafka 是如何实现其高性能、高吞吐的。这一段应该会挺长的我想分两篇来写。今天这一篇主要从宏观上说 kafka 实现的细节，下一篇，在从具体的技术上去分析。


Frontend
Frontend
Frontend
External Tracking Proxy
Backend Service (producer)
Backend Service (producer)
Kafka
Backend Service (consumer)
Backend Service (consumer)
Hadoop
DWH

我们先看 kafka 的设计元素：

1. 通常来说，kafka 的使用是为了消息的持久化（persistent messages）
2. 吞吐量是 kafka 设计的主要目标
3. 关于消费的状态被记录为 consumer 的一部分，而不是 server。这点稍微解释下，这里的 server 还是只 broker，谁消费了多少数据都记录在消费者自己手中，不存在 broker 中。按理说，消费记录也是一个日志，可以放在 broker 中，至于为什么要这么设计，我们写下去了再说。
4. Kafka 的分布式可以表现在 producer、broker、consumer 都可以分布在多台机器上。

在讲实现原理之前，我们还得了解几个术语：

- Topic：其实官网上没有单独提这个词，但 topic 其实才是理解的关键，在 kafka 中，不同的数据可以按照不同的 topic 存储。
- Message：消息是 kafka 处理的对象，在 kafka 中，消息是被发布到 broker 的 topic 中。而 consumer 也是从相应的 topic 中拿数据。也就是说，message 是按 topic 存储的。
- Consumer Group：虽然上面的设计元素第四条，我们说三者都可以部署到多台机器上，三者分别并作为一个逻辑的 group，但对于 consumer 来说这样的部署需要特殊的支持。Consumer Group 就是让多个（相关的）进程（机器）在逻辑上扮演一个 consumer。这个 group 的定义其实是为了去支持 topic 这样的语义。在 JMS 中，大家最熟悉的是队列，我们将所有的 consumer 放到一个 group 中，这样就是队列。而 topic 则是将 consumer 放置到与它相关的 topic 中去。所以无论一个 topic 存在于多少个 consumer 中，a message is stored only a single time。你可能会有疑问，备份怎么办，接着看下去。

接下来，我们来看 kafka 的实现究竟依赖了哪些东西。

1. 硬件上，kafka 选用了硬盘直接读写，当然这里也有策略。一个 67200rpm STAT RAID5 的阵列，线性读写速度是 300MB/sec，如果是随机读写，速度则是 50K/sec。差距很明显，所以 kafka 选的策略就是利用线性存储，至于怎么存，我们在存储中会说到。
2. 关于缓存，kafka 没有使用内存作为缓存。操作系统用个特性，如果不用 direct I/O，那些闲置的 memory 会去做 disk caching，如果 a process maintains an in-process cache of the data，这样的情况下可能会产生双份的 pagecache，会存储两遍。另外 Kafka 跑在 JVM 上，本身 JVM 垃圾回收、创建对象都非常的耗内存，所以不再依赖于内存做缓存。All data is immediately written to a persistent log on the filesystem without any call to flush the data.当然内核自己的 flush 不算了。温泉做一次 32G 的内存缓存，需要大概 10 多分钟。
3. Liner writer/reader：这样做的虽然没有 B 树那样多样的变化，但却有 O（1）的操作，而且读写不会相互影响。同时，线性的读写也解耦了数据规模的问题。用廉价的存储就可以达到很高的性价比。
4. Zero-copy：将数据从硬盘写到 socket 一般需要经过...你可以自己算一下，这是操作系统里的知识，答案在文章末尾，具体也可以看这里：http://my.oschina.net/ielts0909/blog/85147。一句话，Zero-copy 减少了 IO 的操作步骤。
5. GZIP and Snappy compression：考虑到传输最大的瓶颈就在于网络上，kafka 提供了对数据压缩的各种协议。
6. 事务机制：虽然 kafka 对事务的处理比较薄弱，但是在 message 的分发上还是做了一定的策略来保证数据递送的准确性。

At most once—this handles the first case described. Messages are immediately marked as consumed, so they can't be given out twice, but many failure scenarios may lead to losing messages.

At least once—this is the second case where we guarantee each message will be delivered at least once, but in failure cases may be delivered twice.

Exactly once—this is what people actually want, each message is delivered once and only once.

上述就是关于 kafka 的实现细节，主要写了关于 kafka 采用到的技术和使用技术的原因，在后面一篇中，我将主要讲述 producer、broker、consumer 之间的配合以及 kafka 的存储问题。

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space
2. The application reads the data from kernel space into a user-space buffer
3. The application writes the data back into kernel space into a socket buffer
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network

其实 zero-copy 这个技术我们已经在使用了，在 NIO 中的 FileChannel 中的 transferTo 就是采用这样的原理的。

第四章　KAFKA 实现细节（下）

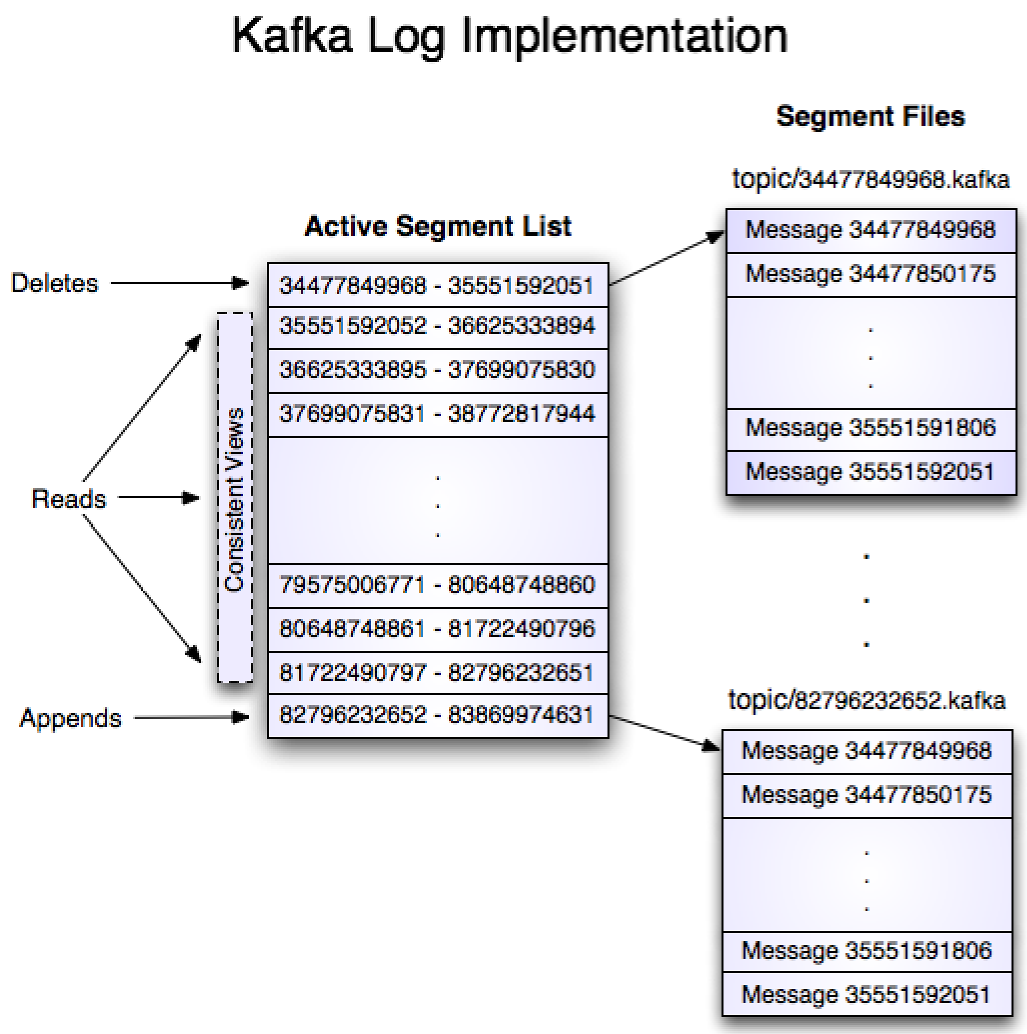
在这一篇，我想主要写点儿kafka的存储，以及对前文kafka的分布式一些补充，kafka的应用中，分布式使用是一个很关键的主题，更好的理解producer、broker和consumer的分布式构建有利于提高系统整体的性能。这部分理论其实很简单，所以就不花大精力去写了。

在上一篇中，我们说到了 kafka 直接使用硬盘作为存储，并且不使用内存缓存。我们还说到，之所以要这么应用，主要是考虑到硬盘在线性读写时候速度完全能满足要求，以及使用内存缓存会带来的一些负面影响。如果你不是很了解，可以先看看之前的那篇。

有关存储方面，我们要引进几个概念：

1. Partition：同一个 topic 下可以设置多个 partition，目的是为了提高并行处理的能力。可以将同一个 topic 下的 message 存储到不同的 partition 下。
2. Offset：kafka 的存储文件都是按照 offset.kafka 来命名，用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到 2048.kafka 的文件即可。当然 the first offset 就是 00000000000.kafka。
3. Messages：这里写下 message 的构成，a fixed-size header 和 variable length opaque byte array payload 组成。Header 由 version 和 checksum 组成，checksum 采用 CRC32。

下图就反应了日志都是 append 的这一个过程：



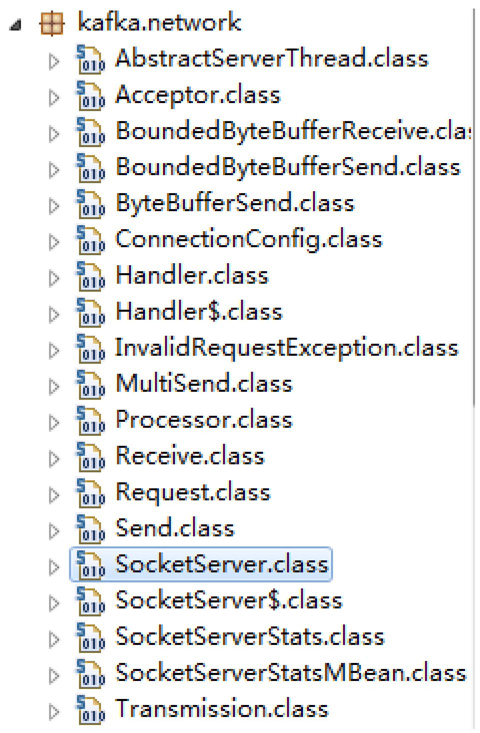
在写的时候会有两个参数需要注意：The log takes two configuration parameter *M* which gives the number of messages to write before forcing the OS to flush the file to disk, and *S* which gives a number of seconds after which a flush is forced.

在分布式方面：

1. broker 的部署是一种 no central master 的概念，并且每个节点都是同等的，节点的增加和减少都不需要改变任何配置。
2. producer 和 consumer 通过 zookeeper 去发现 topic，并且通过 zookeeper 来协调生产和消费的过程。
3. producer、consumer 和 broker 均采用 TCP 连接，通信基于 NIO 实现。Producer 和 consumer 能自动检测 broker 的增加和减少。

第五章　KAFKA.NETWORK 包源码解读

最近阅读了 kafka network 包的源码，主要是想了解下 kafka 底层通信的一些细节，这部分都是用 NIO 实现的，并且用的是最基本的 NIO 实现模板，代码阅读起来也比较简单。抛开 zookeeper 这部分的通信不看，我们就看最基本的 producer 和 consumer 之间的基于 NIO 的通信模块。在 network 中主要包含以下类：



```
kafka.network
    AbstractServerThread.class
    Acceptor.class
    BoundedByteBufferReceive.cla
    BoundedByteBufferSend.class
    ByteBufferSend.class
    ConnectionConfig.class
    Handler.class
    Handler$.class
    InvalidRequestException.class
    MultiSend.class
    Processor.class
    Receive.class
    Request.class
    Send.class
    SocketServer.class
    SocketServer$.class
    SocketServerStats.class
    SocketServerStatsMBean.class
    Transmission.class
```

我们挑选几个最主要的类说明，先从 SocketServer 的描述看起：

```
/**
* An NIO socket server. The thread model is
*    1 Acceptor thread that handles new connections
*    N Processor threads that each have their own selectors and handle all requests from their connections synchronously
*/
```

在 SocketServer 中采用 processors 数组保存 processor

```
Private val processors = new Array[Processor](numProcessorThreads);
```

在 AbstractServerThread 继承了 runnable，其中采用闭锁控制开始和结束，主要作用是为了实现同步。同时打开 selector，为后续的继承者使用。

```
protected val selector = Selector.open();
protected val logger = Logger.getLogger(getClass());
private val startupLatch = new CountDownLatch(1);
private val shutdownLatch = new CountDownLatch(1);
private val alive = new AtomicBoolean(false);
```

这个类是后续讲到的两个类的基类，并且闭锁的应用是整个同步作用实现的关键，我们看一组 stratup 的闭锁操作，其中 Unit 在 scala 语法中你可以把他认为是 void，也就是方法的返回值为空：

```
/**
 * Wait for the thread to completely start up
 */
  def awaitStartup(): Unit = startupLatch.await
/**
 * Record that the thread startup is complete
 */
protected def startupComplete() = {
   alive.set(true)
   startupLatch.countDown
}
```

Acceptor 继承了 AbstractServerThread，虽然叫 Acceptor，但是它并没有单独拿出来使用，而是直接被 socketServer 引用，这点在命名和使用上与一般的通信框架不同：

```
private[kafka] class Acceptor(
         val port: Int,
         private val processors: Array[Processor],
         val sendBufferSize: Int,
         val receiveBufferSize: Int) extends AbstractServerThread {
    ……
}
```

这个类中主要实现了 ServerSocketChannel 的相关工作：

```
val serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket.bind(new InetSocketAddress(port));
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
logger.info("Awaiting connections on port " + port);
startupComplete();
```

其内部操作和 NIO 一样：

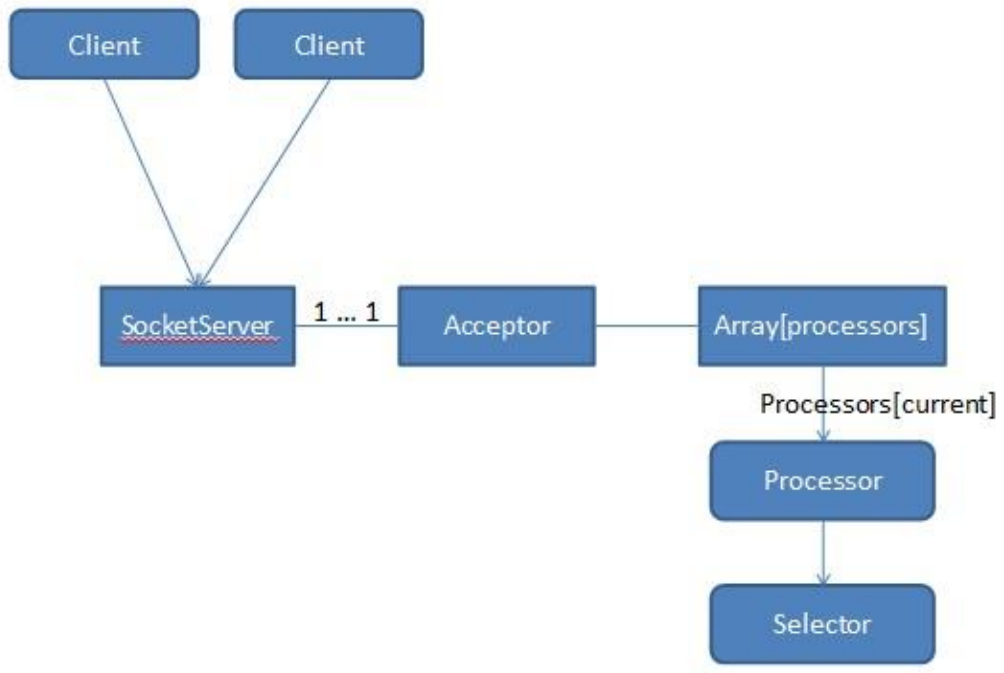
```
/*
* Accept a new connection
*/
def accept(key: SelectionKey, processor: Processor) {
    val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel];
    serverSocketChannel.socket().setReceiveBufferSize(receiveBufferSize);
    val socketChannel = serverSocketChannel.accept();
```

```
            socketChannel.configureBlocking(false);
            socketChannel.socket().setTcpNoDelay(true);
            socketChannel.socket().setSendBufferSize(sendBufferSize);
            if (logger.isDebugEnabled()) {
              logger.debug("sendBufferSize: [" + socketChannel.socket().getSendBufferSize();
                    + "] receiveBufferSize: [" + socketChannel.socket().getReceiveBufferSize()
                    + "]");
            }
            processor.accept(socketChannel);
        }
```

Procesor 类继承了 abstractServerThread，其实主要是在 Acceptor 类中的 accept 方法中，又新启一个线程来处理读写操作：

```
private[kafka] class Processor(val handlerMapping: Handler.HandlerMapping,
                               val time: Time,
                               val stats: SocketServerStats,
                               val maxRequestSize: Int) extends AbstractServerThread{
    ……
}
```

所以整个 kafka 中使用的 NIO 的模型可以归结为下图：



socketServer 中引用 Acceptor 处理多个 client 过来的 connector，并为每个 connection 创建出一个 processor 去单独处理，每个 processor 中均引用独立的 selector。

整体来说，这样的设计和我们在用 NIO 写传统的通信没有什么区别，只是这里在同步上稍微做了点儿文章。更详细的网络操作还是请看 mina 系列的分析。

第六章　KAFKA BROKER 配置介绍

这部分内容对了解系统和提高软件性能都有很大的帮助，kafka 官网上也给出了比较详细的配置详单，但是我们还是直接从代码来看 broker 到底有哪些配置需要我们去了解的，配置都有英文注释，所以每一部分是干什么的就不翻译了，都能看懂：

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
*    http://www.apache.org/licenses/LICENSE-2.0
* Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package kafka.server
import java.util.Properties
import kafka.utils.{Utils, ZKConfig}
import kafka.message.Message
/**
 * Configuration settings for the kafka server
 */
class KafkaConfig(props: Properties) extends ZKConfig(props) {
  /* the port to listen and accept connections on */
  val port: Int = Utils.getInt(props, "port", 6667)
  /* hostname of broker. If not set, will pick up from the value returned from getLocalHost.
If there are multiple interfaces getLocalHost may not be what you want. */
  val hostName: String = Utils.getString(props, "hostname", null)
  /* the broker id for this server */
  val brokerId: Int = Utils.getInt(props, "brokerid")
    /* the SO_SNDBUFF buffer of the socket sever sockets */
  val socketSendBuffer: Int = Utils.getInt(props, "socket.send.buffer", 100*1024)
    /* the SO_RCVBUFF buffer of the socket sever sockets */
  val socketReceiveBuffer: Int = Utils.getInt(props, "socket.receive.buffer", 100*1024)
    /* the maximum number of bytes in a socket request */
  val maxSocketRequestSize: Int = Utils.getIntInRange(props, "max.socket.request.bytes",
100*1024*1024, (1, Int.MaxValue))
  /* the maximum size of message that the server can receive */
  val maxMessageSize = Utils.getIntInRange(props, "max.message.size", 1000000, (0,
Int.MaxValue))
  /* the number of worker threads that the server uses for handling all client requests*/
  val numThreads = Utils.getIntInRange(props, "num.threads",
Runtime.getRuntime().availableProcessors, (1, Int.MaxValue))
   /* the interval in which to measure performance statistics */
  val monitoringPeriodSecs = Utils.getIntInRange(props, "monitoring.period.secs", 600,
(1, Int.MaxValue))
   /* the default number of log partitions per topic */
  val numPartitions = Utils.getIntInRange(props, "num.partitions", 1, (1, Int.MaxValue))
   /* the directory in which the log data is kept */
  val logDir = Utils.getString(props, "log.dir")
   /* the maximum size of a single log file */
  val logFileSize = Utils.getIntInRange(props, "log.file.size", 1*1024*1024*1024,
(Message.MinHeaderSize, Int.MaxValue))
 /* the maximum size of a single log file for some specific topic */
  val logFileSizeMap = Utils.getTopicFileSize(Utils.getString(props,
"topic.log.file.size", ""))
 /* the maximum time before a new log segment is rolled out */
  val logRollHours = Utils.getIntInRange(props, "log.roll.hours", 24*7, (1,
Int.MaxValue))
 /* the number of hours before rolling out a new log segment for some specific topic */
  val logRollHoursMap = Utils.getTopicRollHours(Utils.getString(props,
"topic.log.roll.hours", ""))
 /* the number of hours to keep a log file before deleting it */
  val logRetentionHours = Utils.getIntInRange(props, "log.retention.hours", 24*7, (1,
Int.MaxValue))
 /* the number of hours to keep a log file before deleting it for some specific topic*/
  val logRetentionHoursMap = Utils.getTopicRetentionHours(Utils.getString(props,
"topic.log.retention.hours", ""))
   /* the maximum size of the log before deleting it */
  val logRetentionSize = Utils.getLong(props, "log.retention.size", -1)
 /* the maximum size of the log for some specific topic before deleting it */
```

```
  val logRetentionSizeMap = Utils.getTopicRetentionSize(Utils.getString(props,
"topic.log.retention.size", ""))
 /* the frequency in minutes that the log cleaner checks whether any log is eligible for
deletion */
  val logCleanupIntervalMinutes = Utils.getIntInRange(props,
"log.cleanup.interval.mins", 10, (1, Int.MaxValue))
   /* enable zookeeper registration in the server */
  val enableZookeeper = Utils.getBoolean(props, "enable.zookeeper", true)
 /* the number of messages accumulated on a log partition before messages are flushed
to disk */
  val flushInterval = Utils.getIntInRange(props, "log.flush.interval", 500, (1,
Int.MaxValue))
 /* the maximum time in ms that a message in selected topics is kept in memory before
flushed to disk, e.g., topic1:3000,topic2: 6000  */
  val flushIntervalMap = Utils.getTopicFlushIntervals(Utils.getString(props,
"topic.flush.intervals.ms", ""))
 /* the frequency in ms that the log flusher checks whether any log needs to be flushed
to disk */
  val flushSchedulerThreadRate = Utils.getInt(props,
"log.default.flush.scheduler.interval.ms",  3000)
 /* the maximum time in ms that a message in any topic is kept in memory before flushed
to disk */
  val defaultFlushIntervalMs = Utils.getInt(props, "log.default.flush.interval.ms",
flushSchedulerThreadRate)
  /* the number of partitions for selected topics, e.g., topic1:8,topic2:16 */
  val topicPartitionsMap = Utils.getTopicPartitions(Utils.getString(props,
"topic.partition.count.map", ""))
 /* the maximum length of topic name*/
  val maxTopicNameLength = Utils.getIntInRange(props, "max.topic.name.length", 255, (1,
Int.MaxValue))
}
```

上面这段代码来自 kafka.server 包下的 KafkaConfig 类，之前我们就说过，broker 就是 kafka 中的 server，所以讲配置放在这个包中也不奇怪。这里我们顺着代码往下读，也顺便看看 scala 的语法。和 java 一样也要 import 相关的包，kafka 将同一包内的两个类写在大括号中：

```
import kafka.utils.{Utils, ZKConfig}
```

然后我们看类的写法：

```
class KafkaConfig(props: Properties) extends ZKConfig(props)
```

我们看到在加载 kafkaConfig 的时候会加载一个 properties 对象，同时也会加载有关 zookeeper 的 properties，这个时候我们可以回忆一下，之前我们启动 kafka broker 的命令：

1. 启动 zookeeper server ：bin/zookeeper-server-start.sh ../config/zookeeper.properties & (用&是为了能退出命令行)
2. 启动 kafka server: bin/kafka-server-start.sh ../config/server.properties &

所以你能明白，初始化 kafka broker 的时候程序一定是去加载位于 config 文件夹下的 properties，这个和 java 都一样没有区别。当然 properties 我们也可以通过程序来给出，这个我们后面再说，继续看我们的代码。既然找到了对应的 properties 文件，我们就结合代码和 properties 一起来看。

Kafka broker 的 properties 中，将配置分为以下六类：

1. Server Basics：关于 brokerid，hostname 等配置
2. Socket Server Settings：关于传输的配置，端口、buffer 的区间等。
3. Log Basics：配置 log 的位置和 partition 的数量。
4. Log Flush Policy：这部分是 kafka 配置中最重要的部分，决定了数据 flush 到 disk 的策略。
5. Log Retention Policy：这部分主要配置日志处理时的策略。
6. Zookeeper：配置 zookeeper 的相关信息。

在文件 properties 中的配置均出现在 kafkaConfig 这个类中，我们再看看 kafkaConfig 中的代码：

```
/* the broker id for this server */
val brokerId: Int = Utils.getInt(props, "brokerid");
/* the SO_SNDBUFF buffer of the socket sever sockets */
val socketSendBuffer: Int = Utils.getInt(props, "socket.send.buffer", 100*1024);
```

凡是参数中有三个的，最后一个是 default，而参数只有两个的则要求你一定要配置，否则的话则报错。当然在这么多参数中肯定是有一些经验参数的，至于这些参数怎么配置我确实没有一个特别的推荐，需要在不断的测试中才能磨合出来。

当然你也可以将配置写在程序里，然后通过程序去启动 broker，这样 kafka 的配置就可以像下面一样写：

```
Properties props = new Properties();
props.setProperty("port","9093");
props.setProperty("log.dir","/home/kafka/data1");
```

我倒是觉得配置还是直接写在配置文件中比较好，如果需要修改也不会影响正在运行的服务，写在内存中，总是会有些不方便的地方。所以还是建议大家都写配置好了，后面讲到的 producer 和 consumer 都一样。

这里再提两个参数一个是 brokerid，每个 broker 的 id 必须要区分；第二个参数是 hostname，这个是 broker 和 producer、consumer 联系的关键，这里记住一定要改成你的地址和端口，否则永远连得都是 localhost。

下一篇将写 producer 和 consumer 的配置了，涉及到这部分就要开始编程了，写着写着又往源码里看进去了，下篇会先讲如何搭建开发环境，然后再写两个简单那的例子去熟悉配置。

第七章　KAFKA 开发环境搭建

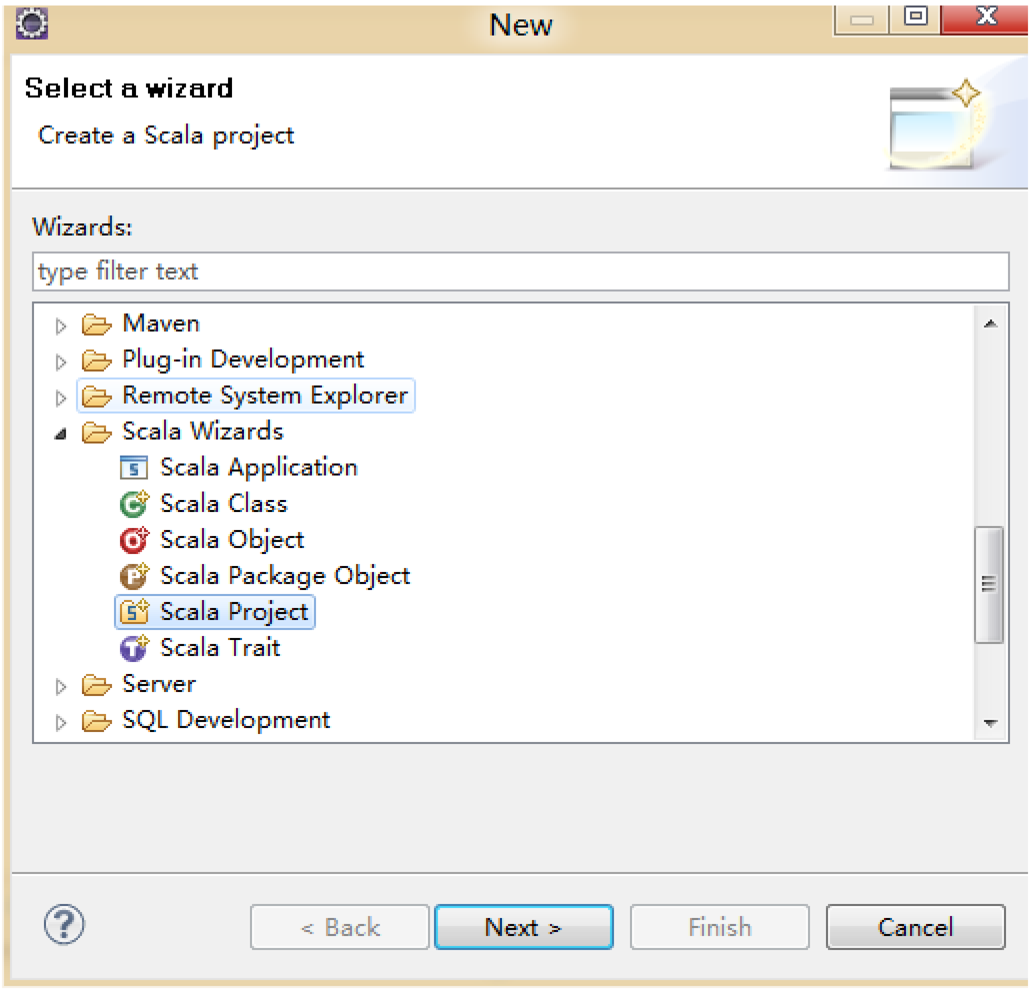
如果你要利用代码来跑 kafka 的应用，那你最好先把官网给出的 example 先在单机环境和分布式环境下跑通，然后再逐步将原有的 consumer、producer 和 broker 替换成自己写的代码。所以在阅读这篇文章前你需要具备以下前提：

1. 简单了解 kafka 功能，理解 kafka 的分布式原理
2. 能在分布式环境下成功运行—topic test。

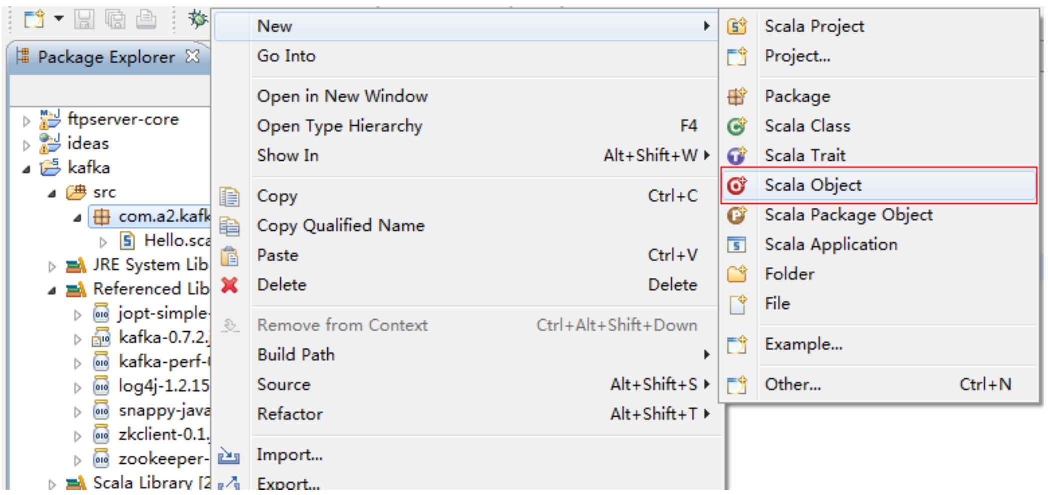
如果你还没有完成上述两个前提，请先看：kafka 分布式初步 kafka 搭建分布式环境

接下来我们就简单介绍下 kafka 开发环境的搭建。

1. 搭建 scala 环境，这里有两种方案，第一直接去 scala 官网下载 SDK，解压配置环境变量；第二种办法，你可以不用安装 SDK，直接在项目中引用 kafka 编译后下载下来的 scala 编译包和编码包（scala-compiler.jar 和 scala-library.jar）。我推荐第二种，因为通过 kafka 编译下载下来的 scala 版本和 kafka 版本都是匹配的（但是有时候可能会跟 eclipse 的插件需要的环境冲突，所以最好把第一种也安装一下，以防万一），而一般我们用的是 java 项目来写，所以直接导入相关依赖包就可以了，第一种方案有助于我们看源码和用 scala 开发。这些 jar 的路径位于 kafka-0.7.2-incubating-src\javatest\lib 目录下。
2. 为 eclipse 安装 scala 开发环境。这只是一个插件，可以在：http://scala-ide.org/中下载安装或者在线安装。装完之后，你就能在 eclipse 中创建 scala 的项目了。



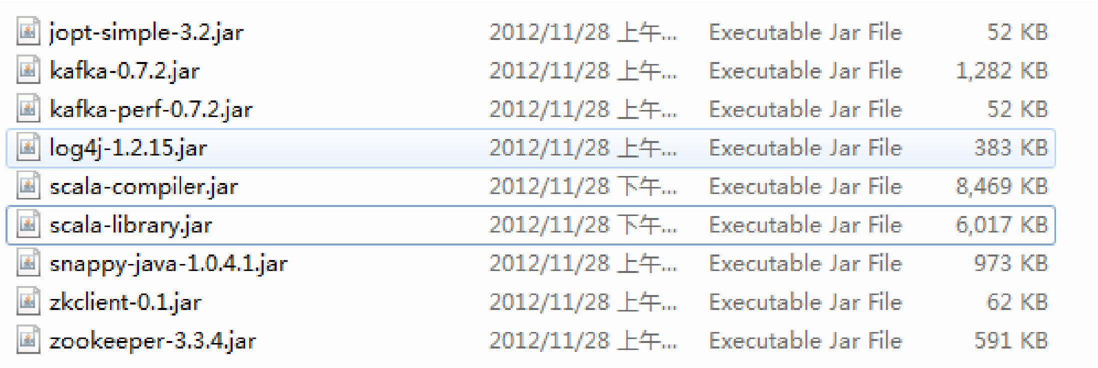
我们可以先写一个 hello world 试一下，这样一来，是不是又多了一种语言写 hello world 了。



有些人 new 的时候找不到 scala 相关的类，那是因为你 eclipse 的 perspective 不对，切换到 scala 的 perspective 下就可以了。注意 new 的是 object，然后输入：

```
package com.a2.kafka.scala.test

object Hello {

   def main(args: Array[String]): Unit = {
    printf("Hello Scala!!");
   }
}
```

3. 找到编码需要的依赖包。记住去你 linux 上经过 update 的 kafka 文件夹里找，不要从直接从官网上下载的文件里找。具体路径是：kafka-0.7.2-incubating-src\javatest\lib 这是你用 java 开发 kafka 相关程序用到的最基础的包，如果你用到了 hadoop，只要去相关的文件夹找一下就可以了。然后把这些包加到项目里即可。



jopt-simple-3.2.jar	2012/11/28 上午...	Executable Jar File	52 KB
kafka-0.7.2.jar	2012/11/28 上午...	Executable Jar File	1,282 KB
kafka-perf-0.7.2.jar	2012/11/28 上午...	Executable Jar File	52 KB
log4j-1.2.15.jar	2012/11/28 上午...	Executable Jar File	383 KB
scala-compiler.jar	2012/11/28 下午...	Executable Jar File	8,469 KB
scala-library.jar	2012/11/28 下午...	Executable Jar File	6,017 KB
snappy-java-1.0.4.1.jar	2012/11/28 上午...	Executable Jar File	973 KB
zkclient-0.1.jar	2012/11/28 上午...	Executable Jar File	62 KB
zookeeper-3.3.4.jar	2012/11/28 上午...	Executable Jar File	591 KB

到了这里，基本的开发环境应该是搭建完了，然后我们要开始写点儿简单的代码了。我们还是根据之前《分布式环境搭建》中给出的例子。稍微回忆下：

1. 启动 zookeeper server ：bin/zookeeper-server-start.sh ../config/zookeeper.properties & (用&是为了能退出命令行)
2. 启动 kafka server: bin/kafka-server-start.sh ../config/server.properties &
3. Kafka 为我们提供了一个 console 来做连通性测试，下面我们先运行 producer：bin/kafka-console-producer.sh --zookeeper 192.168.10.11:2181 --topic test 这是相当于开启了一个 producer 的命令行。

4. 接下来运行 consumer，新启一个 terminal：bin/kafka-console-consumer.sh --zookeeper 192.168.10.11:2181 --topic test --from-beginning
5. 执行完 consumer 的命令后，你可以在 producer 的 terminal 中输入信息，马上在 consumer 的 terminal 中就会出现你输的信息。

这个例子就是在分布式的环境下 producer 生产数据，然后 consumer 从 broker 抓取数据显示在 console 上。当然注意一点 server.properties 中的 hostname 需要换成你的对应地址，具体可以回去看《分布式环境搭建》。现在我们就用代码来模拟 producer 发送数据的过程：

这里我们建一个 java project 就可以了，导入依赖包。kafka-0.7.2-incubating-src\javatest\lib 目录下的 jar.

```
package com.a2.test.kafka;

import java.util.Properties;

import kafka.javaapi.producer.Producer;

import kafka.javaapi.producer.ProducerData;

import kafka.producer.ProducerConfig;

public class Producertest {

    public static void main(String[] args) {

        Properties props = newProperties();

        props.put("zk.connect", "192.168.10.11:2181");

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

        ProducerConfig config = newProducerConfig(props);

        Producer producer = newProducer(config);

        ProducerData data = newProducerData("test", "Hello");

        producer.send(data);

    }

}
```

这样我们就用代码代替了 console 去发送信息了,这里我们用了 utils 中的 properties 对象直接构建了配置，而不是直接读取，当然你也可以读配置。Data 里的两个参数，第一个是指定 topic，第二个是发送的内容。

接下来就是运行了，如果你是在 windows 下，可能会等待很久然后报 Unable to connect to zookeeper server within timeout: 6000，这个可能是网卡的原因，你可以直接放到 linux 上，然后用命令行运行（注意引包）：`Java -jar test.jar  Dclasspath=/lib`

Consumer 的代码以及实现和 producer 差不多，如果你感兴趣可以去官网找相关的代码，都很简单。当然还有一部分关于 producer 和 consumer 的配置，我们下篇再说。

第八章　KAFKA PRODUCER 端封装自定义消息

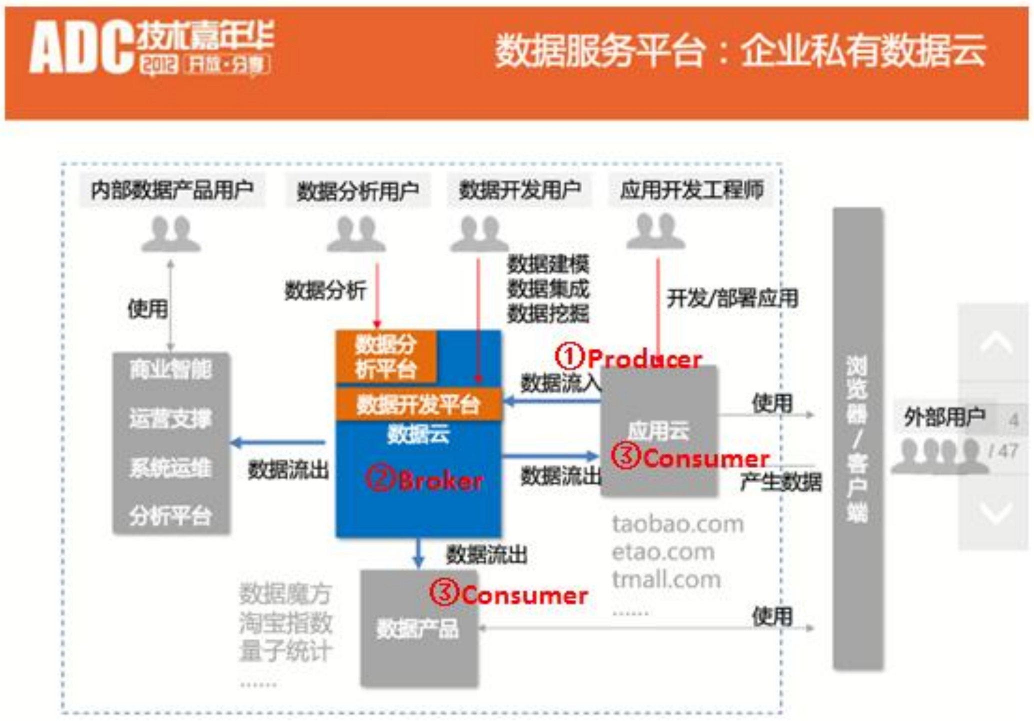
这篇文章主要讲 kafka producer 端的编程，通过一个应用案例来描述 kafka 在实际应用中的作用。如果你还没有搭建起 kafka 的开发环境，可以先参考：

首先描述一下应用的情况：一个站内的搜索引擎，运营人员想知道某一时段，各类用户对商品的不同需求。通过对这些数据的分析，从而获得更多有价值的市场分析报表。这样的情况，就需要我们对每次的搜索进行记录，当然，不太可能使用数据库区记录这些信息（数据库存这些数据我会觉得是种浪费，个人意见）。最好的办法是存日志。然后通过对日志的分析，计算出有用的数据。我们采用 kafka 这种分布式日志系统来实现这一过程。

完成上述一系列的工作，可以按照以下步骤来执行：

1. 搭建 kafka 系统运行环境。
2. 设计数据存储格式（按照自定义格式来封装消息）
3. Producer 端获取真实数据（搜索记录），并对数据按上述 2 中设计的格式进行编码。
4. Producer 将已经编码的数据发送到 broker 上，在 broker 上进行存储（分配存储策略）。
5. Consumer 端从 broker 中获取数据，分析计算。

如果用淘宝数据服务平台的架构来匹配这一过程，broker 就好比数据中心中存储的角色，producer 端基本是放在了应用中心的开放 API 中，consumer 端则一般用于数据产品和应用中心的获取数据中使用。



今天主要写的是 2、3、4 三个步骤。我们先看第二步。为了快速实现，这里就设计一个比较简单的消息格式，复杂的原理和这个一样。

ID | user | keyword | Date

用四个字段分别表示消息的 ID、用户、查询关键词和查询时间。当然你如果要设计的更复杂，可以加入 IP 这些信息。这些用 java 写就是一个简单的 pojo 类，这是 getter/setter 方法即可。由于在封转成 kafka 的 message 时需要将数据转化成 bytep[]类型，可以提供一个序列化的方法。

我在这里直接重写 toString 了：

```
@Override
public String toString() {
    String keyword = "[info kafka producer:]";
    keyword = keyword + this.getId() + "-"+ this.getUser() + "-"
    + this.getKeyword() + "-"+ this.getCurrent();
    return keyword;
}
```

这样还没有完成，这只是将数据格式用 java 对象表现出来，解析来要对其按照 kafka 的消息类型进行封装，在这里我们只需要实现 Encoder 类即可：

```
package org.gfg.kafka.message;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import kafka.message.Message;
public class KeywordMessage implements kafka.serializer.Encoder {
    public static final Logger LOG = LoggerFactory.getLogger(Keyword.class);
    @Override
    public Message toMessage(Keyword words) {
        LOG.info("start in encoding...");
        return new Message(words.toString().getBytes());
    }
}
```

注意泛型和返回类型即可。这样 KeywordMessage 就是一个可以被 kafka 发送和存储的对象了。

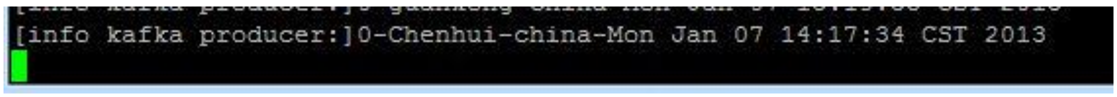
接下来，我们可以编写一部分 producer，获取业务系统的数据。要注意，producer 数据的推送到 broker 的，所以发起者还是业务系统，下面的代码就能直接发送一次数据，注释都很详细：

```
/**配置producer必要的参数*/
Properties props = newProperties();
props.put("zk.connect", "192.168.10.11:2181");
/**选择用哪个类来进行序列化*/
props.put("serializer.class", "org.gfg.kafka.message.KeywordMessage");
props.put("zk.connectiontimeout.ms", "6000");
ProducerConfig config=newProducerConfig(props);
/**制造数据*/
Keyword keyword=new Keyword();
```

```
keyword.setUser("Chenhui");
keyword.setId(0);
keyword.setKeyword("china");
List<Keyword> msg = new ArrayList<Keyword>();
msg.add(keyword);
/**构造数据发送对象*/
Producer<String, Keyword> producer = new  Producer<String, Keyword>(config);
ProducerData<String,Keyword> data = new  ProducerData<String, Keyword>("test", msg);
producer.send(data);
```

发送完之后，我们可以用 bin 目录下的 kafka-console-consumer 来看发送的结果（当然现在用的 topic 是 test）。可以用命令：

./kafka-console-consumer –zookeeper 192.168.10.11:2181 –topic test–from-beginning



```
[info kafka producer:]0-Chenhui-china-Mon Jan 07 14:17:34 CST 2013
```

如果是在使用 zookeeper 搭建分布式的情况下（zookeeper based broker discovery），我们可以执行第三个步骤，用编码来实现 partition 的分配策略。这里需要我们实现 Partitioner 对象：

```
package org.gfg.kafka.partitioner;
import org.gfg.kafka.message.Keyword;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import kafka.producer.Partitioner;
public class ProducerPartitioner implements Partitioner {
    public static final Logger LOG= LoggerFactory.getLogger(Keyword.class);
    @Override
    public intpartition(String key, intnumPartitions) {
        LOG.info("ProducerPartitioner key:"+key+" partitions:"+numPartitions);
        return key.length() % numPartitions;
    }
}
```

在上面的 partition 方法中，值得注意的是，key 我们是在构造数据发送对象时设置的，这个 key 是区分存储的关键，比如我想将我的数据按照不同的用户类别存储。Partition 的好处是可以并发的获取同类数据，提高效率，具体可以看之前的文章。

所以在第二部时的 producer 代码需要有所改进：

```
/**选择用哪个类来进行设置partition*/
props.put("partitioner.class", "org.gfg.kafka.partitioner.ProducerPartitioner");
ProducerData data=new ProducerData("test","developer", msg);
```

增加了对 partition 的配置，并且修改了 ProducerData 的参数，其中，中间的就是 key，如果不设置 partition，kafka 则随机的向 broker 中发送请求。

我们可以看一眼 ProducerData 的源码：

```
package kafka.javaapi.producer

import scala.collection.JavaConversions._

class ProducerData[K, V](

                         private val topic: String,

                         private val key: K,

                         private val data: java.util.List[V]) {

    def this(t: String, d: java.util.List[V]) =

        this(topic = t, key = null.asInstanceOf[K], data = d)

    def this(t: String, d: V) =

        this(topic = t, key = null.asInstanceOf[K], data = asList(List(d)))

    def getTopic: String = topic

    def getKey: K = key

    def getData: java.util.List[V] = data

}
```

至此，producer 端的事情都做完了，当然这就是个 demo，还有很多性能上的优化需要做，当然有了这个基础，我们就能将数据存储到 broker 上，下一步，就是用 consumer 来消费这些日志，形成有价值的数据产品。

第九章　KAFKA PRODUCER 使用注意

最近在测试 kafka 性能的时候特别对 kafka 的 producer 端进行了一些扩展，本想着针对多个业务开发多个 producer 进行并行的生产数据，并通过统一的线程池进行管理，结果在用 jconsole 进行观察的时候，发现线程数一路飙升。

本以为一个简单的发送端程序却花了不少精力。造成线程上涨的主要原因是有两个线程对象不断的被创建，并且暂时无法销毁。一个叫 sendThread，另一个叫 eventThread。并且由于这两类线程驻留在线程池内，也无法回收线程资源。

在 google 中很容易找到这两个线程的来源，zookeeper。ZkClient 去连接 zookeeper 的 server 时候都会创建 sendThread 和 eventThread 两个线程，其中 sendThread 主要用于 client 与 server 端之间的网络连接，真正的处理线程由 eventThread 来执行。Zookeeper 是一个分布式的协调框架，而分布式应用中经常会出现动态的增加或删除节点的操作，所以为了实时了解分布式整个节点的数量和基本信息，就有必要维护一个长连接的线程与服务端保持连接。另外 zookeeper 连接时占用的时间也比较长，如果每次生产数据时都连接发起一次连接势必造成了大量时间的耗费。

所以推荐将 producer 端改写成单例模式，有助于减少 zookeeper 端占用的资源。Producer 自身是线程安全的类，只要封装得当就能最恰当的发挥好 producer 的作用。

另外 producer 分同步 producer 和异步 producer 两种发送方式，基本代码都相似，只需要在创建对象的时候 new 不同的对象即可。如果需要较高的实时性，则推荐使用同步方式发送数据，如果对实时性要求并不高则可采用异步方式发送数据，降低系统的开销。

第十章　KAFKA CONSUMER 端的一些解惑

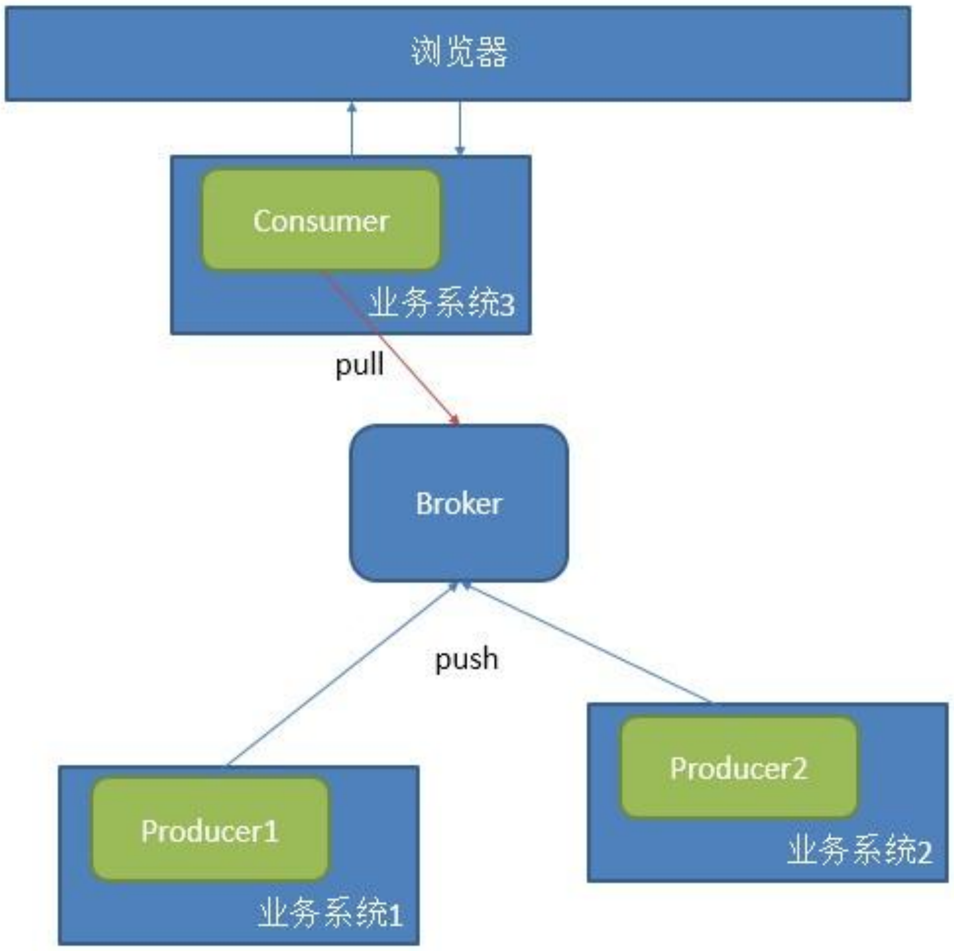
最近一直忙着各种设计和文档，终于有时间来更新一点儿关于 kafka 的东西。之前有一篇文章讲述的是 kafka Producer 端的程序，也就是日志的生产者，这部分比较容易理解，业务系统将运行日志或者业务日志发送到 broker 中，由 broker 代为存储。那讲的是如何收集日志，今天要写的是如何获取日志，然后再做相关的处理。

之前写过 kafka 是将日志按照 topic 的形式存储，一个 topic 会按照 partition 存在同一个文件夹下，目录在 config/server.properties 中指定（具体的存储规则可以查看之前的文章）：

```
# The directory under which to store log files
log.dir=/tmp/kafka-logs
```

Consumer 端的目的就是为了获取 log 日志，然后做进一步的处理。在这里我们可以将数据的处理按照需求分为两个方向，线上和线下，也可以叫实时和离线。实时处理部分类似于网站里的站短，有消息了马上就推送到前端，这是一种对实时性要求极高的模式，kafka 可以做到，当然针对站短这样的功能还有更好的处理方式，我主要将 kafka 线上消费功能用在了实时统计上，处理一些如实时流量汇总、各系统实时吞吐量汇总等。

这种应用，一般采用一个 consumer 中的一个 group 对应一个业务，配合多个 producer 提供数据，如下图模式：



采用这种方式处理很简单，采用官网上给的例子即可解决，只是由于版本的问题，代码稍作更改即可：

```
package com.a2.kafka.consumer;

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;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.Message;
import kafka.message.MessageAndMetadata;

public class CommonConsumer {
    public static void main(String[] args) {
        // specify some consumer properties
        Properties props = new Properties();
        props.put("zk.connect", "192.168.181.128:2181");
        props.put("zk.connectiontimeout.ms", "1000000");
        props.put("groupid", "test_group");
        // Create the connection to the cluster
        ConsumerConfig consumerConfig = new ConsumerConfig(props);
        ConsumerConnector consumerConnector =
        Consumer.createJavaConsumerConnector(consumerConfig);

        Map<String, Integer> map=new HashMap<String,Integer>();
        map.put("test", 2);
        // create 4 partitions of the stream for topic “test”, to allow 4 threads to
consume
        Map<String, List<KafkaStream<Message>>> topicMessageStreams =
            consumerConnector.createMessageStreams(map);
        List<KafkaStream<Message>> streams = topicMessageStreams.get("test");

        // create list of 4 threads to consume from each of the partitions
```

```
            ExecutorService executor = Executors.newFixedThreadPool(4);

            // consume the messages in the threads
            for(final KafkaStream<Message> stream: streams) {
              executor.submit(new Runnable() {
                public void run() {
                  for(MessageAndMetadata<Message> msgAndMetadata: stream) {
                    // process message (msgAndMetadata.message())
                   System.out.println(msgAndMetadata.message());
                  }
                }
              });
            }
        }
    }
```

这是一个 user level 的 API，还有一个 low level 的 API 可以从官网找到，这里就不贴出来了。这个 consumer 是底层采用的是一个阻塞队列，只要一有 producer 生产数据，那 consumer 就会将数据打印出来，这是不是十分符合实时性的要求。

当然这里会产生一个很严重的问题，如果你重启一下上面这个程序，那你连一条数据都抓不到，但是你去 log 文件中明明可以看到所有数据都好好的存在。换句话说，一旦你消费过这些数据，那你就无法再次用同一个 groupid 消费同一组数据了。我已经把结论说出来了，要消费同一组数据，你可以采用不同的 group。

简单说下产生这个问题的原因，这个问题类似于 transaction commit，在消息系统中都会有这样一个问题存在，数据消费状态这个信息到底存哪里。是存在 consumer 端，还是存在 broker 端。对于这样的争论，一般会出现三种情况：

- *At most once*—this handles the first case described. Messages are immediately marked as consumed, so they can't be given out twice, but many failure scenarios may lead to losing messages.
- *At least once*—this is the second case where we guarantee each message will be delivered at least once, but in failure cases may be delivered twice.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

第一种情况是将消费的状态存储在了 broker 端，一旦消费了就改变状态，但会因为网络原因少消费信息，第二种是存在两端，并且先在 broker 端将状态记为 send，等 consumer 处理完之后将状态标记为 consumed，但也有可能因为在处理消息时产生异常，导致状态标记错误等，并且会产生性能的问题。第三种当然是最好的结果。

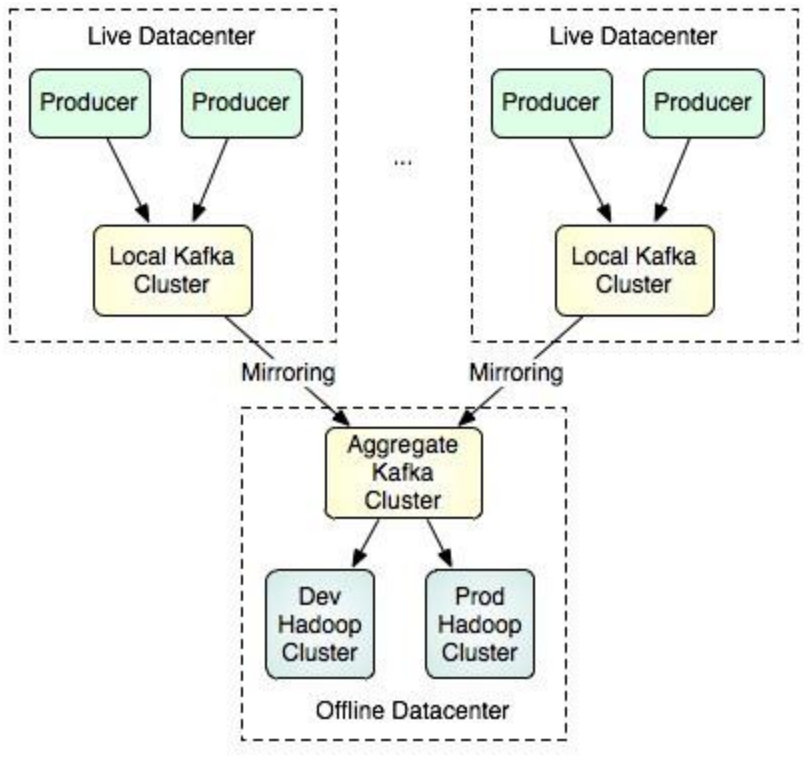
Kafka 解决这个问题采用 high water mark 这样的标记，也就是设置 offset：

Kafka does two unusual things with respect to metadata. First the stream is partitioned on the brokers into a setof distinct partitions. The semantic meaning of these partitions is left up to the producer and the producer specifies whichpartition a message belongs to. Within a partition messages are stored **in**the order **in**whichthey arrive at the broker, and will be given out to consumers **in**that same order. This means that rather than store metadata **for**each message (marking it as consumed, say), we just need to store the "high water mark"**for**each combination of consumer, topic, and partition. Hence the total metadata required to summarize the state of the consumer is actually quite small. In Kafka we refer to this high-water mark as "the offset"**for**reasons that will become clear**in**the implementation section.

所以在每次消费信息时，log4j 中都会输出不同的 offset：

```
[FetchRunnable-0] INFO : kafka.consumer.FetcherRunnable#info : FetchRunnable-0start
                        fetching topic: test part: 0offset: 0from 192.168.181.128:9092
[FetchRunnable-0] INFO : kafka.consumer.FetcherRunnable#info : FetchRunnable-0start
                        fetching topic: test part: 0offset: 15from 192.168.181.128:9092
```

除了采用不同的 groupid 去抓取已经消费过的数据，kafka 还提供了另一种思路，这种方式更适合线下的操作，镜像。

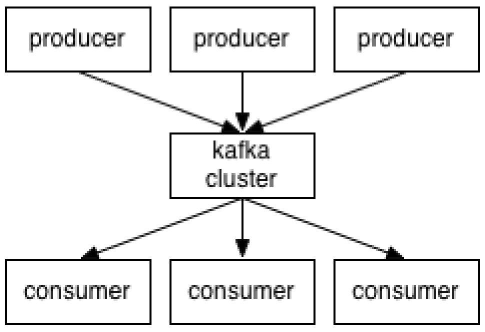


通过一些配置，就可以将线上产生的数据同步到镜像中去，然后再由特定的集群区处理大批量的数据，这种方式可以采用 low level 的 API 按照不同的 partition 和 offset 来抓取数据，以获得更好的并行处理效果。

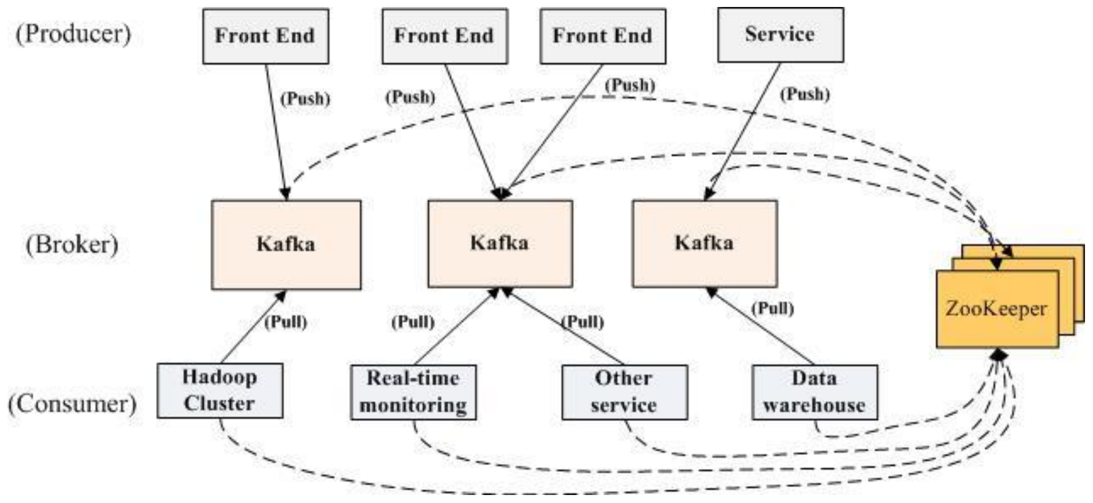
第十一章 KAFKA 0.8 的一些变化

之前分享了一个英文版的变化，一直没时间去翻译，今天上了下 kafka 的官网发现 0.8 的代码能下载了，更值得关注的是 0.8 的相关文档也更新上来了，上面的一些变化还是很可喜的，说明这套系统还是有很大的利用价值的。

最重要的一个变化体现在一张图上：



还记得之前 kafka0.7 版本的时候这张图的样子么：



箭头的指向不同了，之前版本的 kafka 的 consumer 只支持 pull 的模式来抓取数据，而现在在 consumer 端的数据获取方式改变了，可以支持 push 的方式了。这个改变是不是更贴近了现在一些主流的消息系统。另外，Consumer 已经支持”long poll”这种方式，这种方式的好处就是减少了不必要的轮询，使得端到端的数据传输更快捷。

对 consumer 的更新比较突出，特别是 0.7 版本，需要根据 partition 和 offset 获取数据都只能使用 low level 的 api，而在 0.8 版本中都支持了 high level 的 API 了，这样使得编程就更加方便了。不变的是，consumer 还是采用 consumer group 的方式来同时支持 queue 和 publish-subscribe 两种方式。每个 partition 每次只允许一个 consumer 消费来确保消费的顺序性。

Partitions 可以有独立的副本了，这使得之前在服务器宕机情况下，partition 丢失的情况不在发生，这些通过配置 replication factor 进行调整。

更多的内容可参考 kafka 官网关于 0.8 版本的叙述。之前的文章都是在 0.7x 版本上写的，所以如果你要按照之前文章进行参考学习的话请下载 0.7x 的版本。之后 kafka 所有的文章都会按照 0.8 版本的新内容来更新。

第十二章　KAFKA 框架设计

Why we built this

Kafka is a messaging system that was originally developed at LinkedIn to serve as the foundation for LinkedIn's activity stream and operational data processing pipeline. It is now used at a variety of different companies for various data pipeline and messaging uses.

Activity stream data is a normal part of any website for reporting on usage of the site. Activity data is things like page views, information about what content was shown, searches, etc. This kind of thing is usually handled by logging the activity out to some kind of file and then periodically aggregating these files for analysis. Operational data is data about the performance of servers (CPU, IO usage, request times, service logs, etc) and a variety of different approaches to aggregating operational data are used.

In recent years, activity and operational data has become a critical part of the production features of websites, and a slightly more sophisticated set of infrastructure is needed.

译者信息▽

我们为什么要搭建该系统

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

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

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

Use cases for activity stream and operational data

- "News feed" features that broadcast the activity of your friends.
- Relevance and ranking uses count ratings, votes, or click-through to determine which of a given set of items is most relevant.
- Security: Sites need to block abusive crawlers, rate-limit apis, detect spamming attempts, and maintain other detection and prevention systems that key off site activity.
- Operational monitoring: Most sites needs some kind of real-time, heads-up monitoring that can track performance and trigger alerts if something goes wrong.
- Reporting and Batch processing: It is common to load data into a data warehouse or Hadoop system for offline analysis and reporting on business activity

译者信息▽

活动流和运营数据的若干用例

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

- 报表和批处理: 将数据装载到数据仓库或者 Hadoop 系统中进行离线分析，然后针对业务行为做出相应的报表，这种做法很普遍。

Characteristics of activity stream data

This high-throughput stream of immutable activity data represents a real computational challenge as the volume may easily be 10x or 100x larger than the next largest data source on a site.

Traditional log file aggregation is a respectable and scalable approach to supporting offline use cases like reporting or batch processing; but is too high latency for real-time processing and tends to have rather high operational complexity. On the other hand, existing messaging and queuing systems are okay for real-time and near-real-time use-cases, but handle large unconsumed queues very poorly often treating persistence as an after thought. This creates problems for feeding the data to offline systems like Hadoop that may only consume some sources once per hour or per day. Kafka is intended to be a single queuing platform that can support both offline and online use cases.

Kafka supports fairly general messaging semantics. Nothing ties it to activity processing, though that was our motivating use case.

译者信息▽

活动流数据的特点

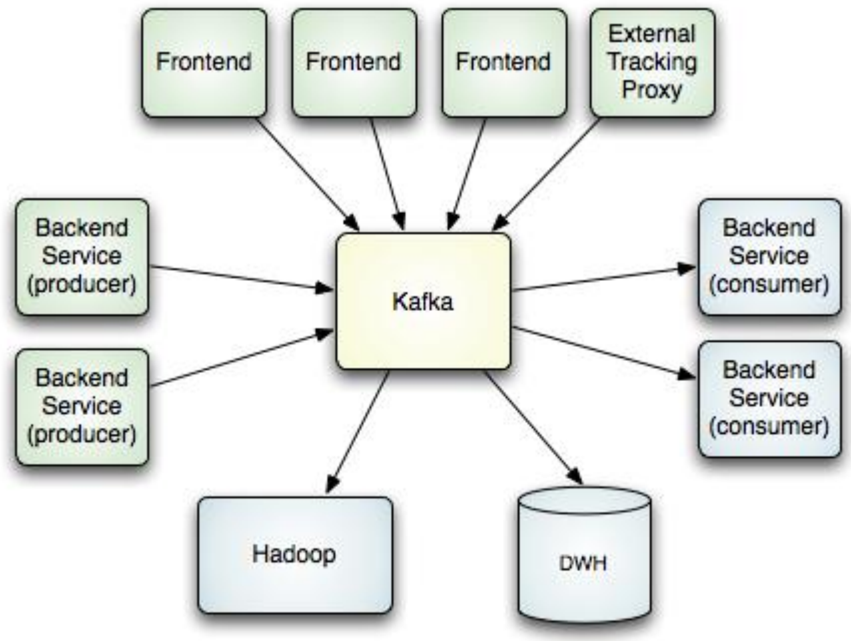
这种由不可变(immutable)的活动数据组成的高吞吐量数据流代表了对计算能力的一种真正的挑战，因其数据量很容易就可能会比网站中位于第二位的数据源的数据量大 10 到 100 倍。

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

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

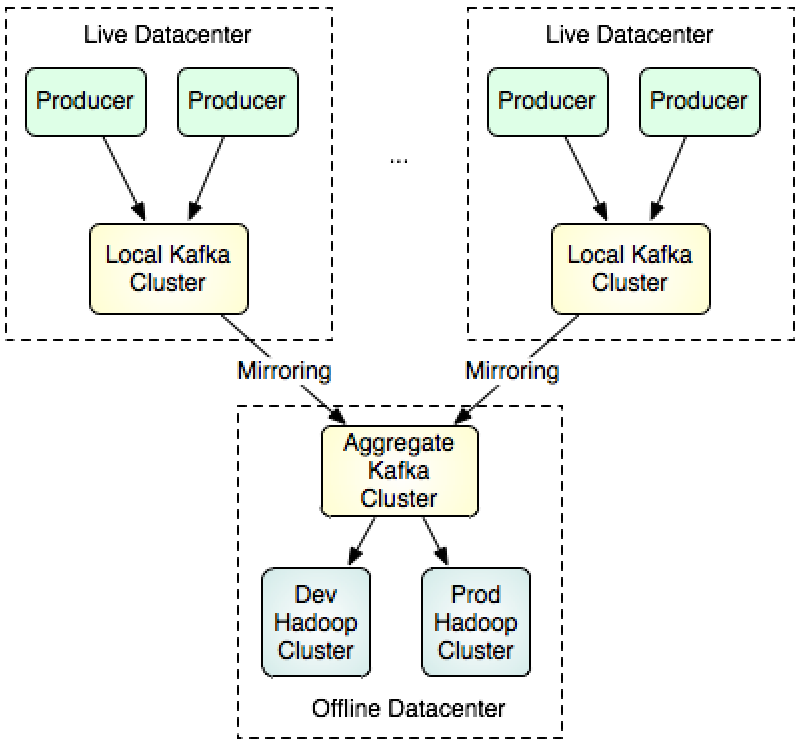
Deployment

The following diagram gives a simplified view of the deployment topology at LinkedIn.


Frontend
Frontend
Frontend
External Tracking Proxy
Backend Service (producer)
Backend Service (producer)
Kafka
Backend Service (consumer)
Backend Service (consumer)
Hadoop
DWH

Note that a single kafka cluster handles all activity data from all different sources. This provides a single pipeline of data for both online and offline consumers. This tier acts as a buffer between live activity and asynchronous processing. We also use kafka to replicate all data to a different datacenter for offline consumption.

It is not intended that a single Kafka cluster span data centers, but Kafka is intended to support multi-datacenter data flow topologies. This is done by allowing mirroring or "syncing" between clusters. This feature is very simple, the mirror cluster simply acts as a consumer of the source cluster. This means it is possible for a single cluster to join data from many datacenters into a single location. Here is an example of a possible multi-datacenter topology aimed at supporting batch loads:

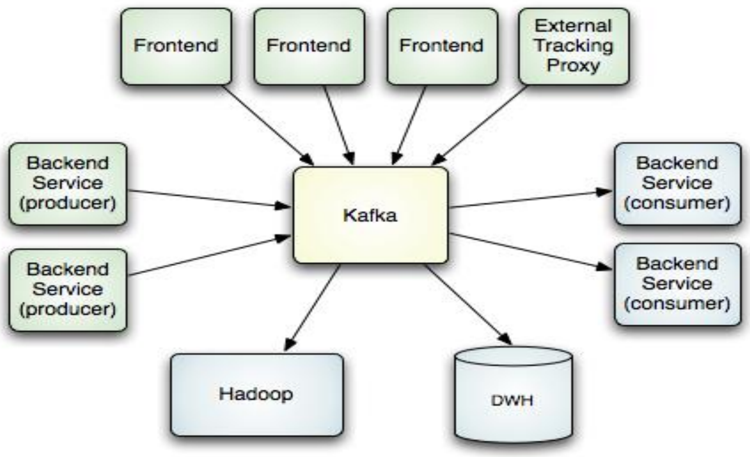


Note that there is no correspondence between nodes in the two clusters—they may be of different sizes, contain different number of nodes, and a single cluster can mirror any number of source clusters. More details on using the mirroring feature can be found here.

译者信息▽

部署

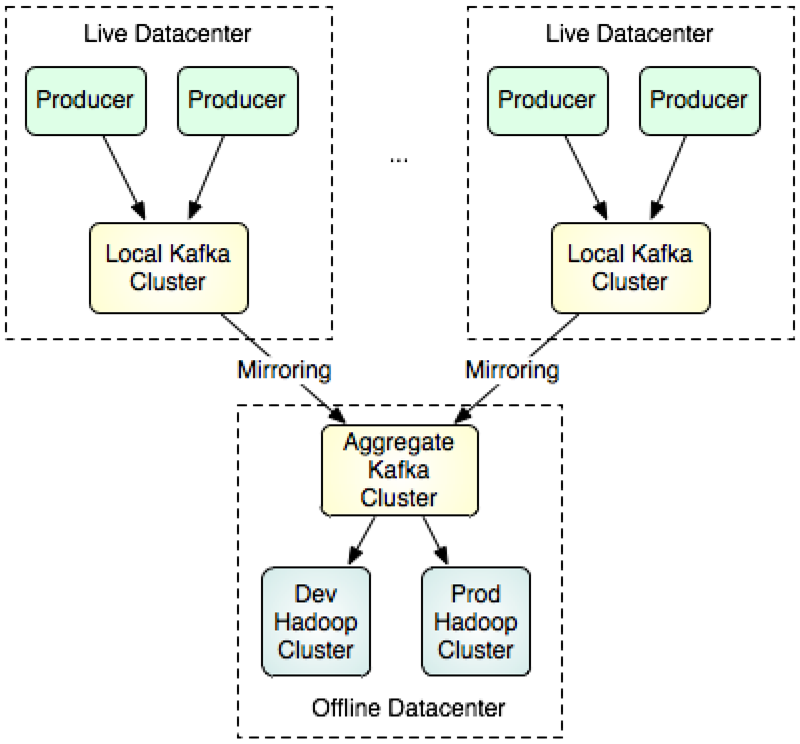
下面的示意图所示是在 LinkedIn 中部署后各系统形成的拓扑结构。



Frontend
Frontend
Frontend
External Tracking Proxy
Backend Service (producer)
Backend Service (producer)
Kafka
Backend Service (consumer)
Backend Service (consumer)
Hadoop
DWH

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

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



请注意，在图中上面部分的两个集群之间不存在通信连接，两者可能大小不同，具有不同数量的节点。下面部分中的这个单个的集群可以镜像任意数量的源集群。要了解镜像功能使用方面的更多细节，请访问这里.

Major Design Elements

There is a small number of major design decisions that make Kafka different from most other messaging systems:

1. Kafka is designed for persistent messages as the common case
2. Throughput rather than features are the primary design constraint
3. State about *what* has been consumed is maintained as part of the consumer not the server
4. Kafka is explicitly distributed. It is assumed that producers, brokers, and consumers are all spread over multiple machines.

Each of these decisions will be discussed in more detail below.

译者信息▽ 主要的设计元素

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

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

以上这些设计决策将在下文中进行逐条详述。

Basics

First some basic terminology and concepts.

Messages are the fundamental unit of communication. Messages are *published* to a *topic* by a *producer* which means they are physically sent to a server acting as a *broker* (probably another machine). Some number of *consumers* subscribe to a topic, and each published message is delivered to all the consumers.

Kafka is explicitly distributed—producers, consumers, and brokers can all be run on a cluster of machines that co-operate as a logical group. This happens fairly naturally for brokers and producers, but consumers require some particular support. Each consumer process belongs to a *consumer group* and each message is delivered to exactly one process within every consumer group. Hence a consumer group allows many processes or machines to logically act as a single consumer. The concept of consumer group is very powerful and can be used to support the semantics of either a *queue* or *topic* as found in JMS. To support *queue* semantics, we can put all consumers in a single consumer group, in which case each message will go to a single consumer. To support *topic* semantics, each consumer is put in its own consumer group, and then all consumers will receive each message. A more common case in our own usage is that we have multiple logical consumer groups, each consisting of a cluster of consuming machines that act as a logical whole. Kafka has the added benefit in the case of large data that no matter how many consumers a topic has, a message is stored only a single time.

译者信息▽

基础知识

首先来看一些基本的术语和概念。

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

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

Message Persistence and Caching

Don't fear the filesystem!

Kafka relies heavily on the filesystem for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a 6 7200rpm SATA RAID-5 array is about 300MB/sec but the performance of random writes is only about 50k/sec—a difference of nearly 10000X. These linear reads and writes are the most predictable of all usage patterns, and hence the one detected and optimized best by the operating system using read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this ACM Queue article; they actually find that sequential disk access can in some cases be faster than random memory access!

译者信息▽

消息持久化（Message Persistence）及其缓存

不要害怕文件系统！

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

磁盘性能方面最关键的一个事实是，在过去的十几年中，硬盘的吞吐量正在变得和磁盘寻道时间严重不一致了。结果，在一个由 6 个 7200rpm 的 SATA 硬盘组成的 RAID-5 磁盘阵列上，线性写入（linear write）的速度大约是 300MB/秒，但随即写入却只有 50k/秒，其中的差别接近 10000 倍。线性读取和写入是所有使用模式中最具可预计性的一种方式，因而操作系统采用预读（read-ahead）和后

写（write-behind）技术对磁盘读写进行探测并优化后效果也不错。预读就是提前将一个比较大的磁盘块中内容读入内存，后写是将一些较小的逻辑写入操作合并起来组成比较大的物理写入操作。关于这个问题更深入的讨论请参考这篇文章 ACM Queue article；实际上他们发现，在某些情况下，顺序磁盘访问能够比随即内存访问还要快！

To compensate for this performance divergence modern operating systems have become increasingly aggressive in their use of main memory for disk caching. Any modern OS will happily divert *all* free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice.

Furthermore we are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

1. The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
2. Java garbage collection becomes increasingly sketchy and expensive as the in-heap data increases.

译者信息▽

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

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

1. Java 对象的内存开销（overhead）非常大，往往是对象中存储的数据所占内存的两倍（或更糟）。

2. Java 中的内存垃圾回收会随着堆内数据不断增长而变得越来越不明确，回收所花费的代价也会越来越大。

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure—we at least double the available cache by having automatic access to all free memory, and likely double again by storing a compact byte structure rather than individual objects. Doing so will result in a cache of up to 28-30GB on a 32GB machine without GC penalties. Furthermore this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory (which for a 10GB cache may take 10 minutes) or else it will need to start with a completely cold cache (which likely means terrible initial performance). It also greatly simplifies the code as all logic for maintaining coherency between the cache and filesystem is now in the OS, which tends to do so more efficiently and more correctly than one-off in-process attempts. If your disk usage favors linear reads then read-ahead is effectively pre-populating this cache with useful data on each disk read.译者信息▽

由于这些因素，使用文件系统并依赖于页面缓存要优于自己在内存中维护一个缓存或者什么别的结构 —— 通过对所有空闲内存自动拥有访问权，我们至少将可用的缓存大小翻了一倍，然后通过保存压缩后的字节结构而非单个对象，缓存可用大小接着可能又翻了一倍。这么做下来，在 GC 性能不受损失的情况下，我们可在一台拥有 32G 内存的机器上获得高达 28 到 30G 的缓存。而且，这种缓存即使在服务重启之后会仍然保持有效，而不象进程内缓存，进程重启后还需要在内存中进行缓存重建（10G 的缓存重建时间可能需要 10 分钟），否则就需要以一个全空的缓存开始运行（这么做它的初

始性能会非常糟糕）。这还大大简化了代码，因为对缓存和文件系统之间的一致性进行维护的所有逻辑现在都是在 OS 中实现的，这事 OS 做起来要比我们在进程中做那种一次性的缓存更加高效，准确性也更高。如果你使用磁盘的方式更倾向于线性读取操作，那么随着每次磁盘读取操作，预读就能非常高效使用随后准能用得着的数据填充缓存。

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush to the filesystem only when necessary, we invert that. All data is immediately written to a persistent log on the filesystem without any call to flush the data. In effect this just means that it is transferred into the kernel's pagecache where the OS can flush it later. Then we add a configuration driven flush policy to allow the user of the system to control how often data is flushed to the physical disk (every N messages or every M seconds) to put a bound on the amount of data "at risk" in the event of a hard crash.

This style of pagecache-centric design is described in an article on the design of Varnish here (along with a healthy helping of arrogance).

译者信息▽

这就让人联想到一个非常简单的设计方案：不是要在内存中保存尽可能多的数据并在需要时将这些数据刷新（flush）到文件系统，而是我们要做完全相反的事情。所有数据都要立即写入文件系统中持久化的日志中但不进行刷新数据的任何调用。实际中这么做意味着，数据被传输到 OS 内核的页面缓存中了，OS 随后会将这些数据刷新到磁盘的。此外我们添加了一条基于配置的刷新策略，允许用户对把数据刷新到物理磁盘的频率进行控制（每当接收到 N 条消息或者每过 M 秒），从而可以为系统硬件崩溃时“处于危险之中”的数据在量上加个上限。

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

Constant Time Suffices

The persistent data structure used in messaging systems metadata is often a BTree. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. They do come with a fairly high cost, though: Btree operations are O(log N). Normally O(log N) is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at 10 ms a pop, and each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead. Since storage systems mix very fast cached operations with actual physical disk operations, the observed performance of tree structures is often superlinear. Furthermore BTrees require a very sophisticated page or row locking implementation to avoid locking the entire tree on each operation. The implementation must pay a fairly high price for row-locking or else effectively serialize all reads. Because of the heavy reliance on disk seeks it is not possible to effectively take advantage of the improvements in drive density, and one is forced to use small (< 100GB high RPM SAS drives to maintain a sane ratio of data to seek capacity p>译者信息▽

常量时长足矣

消息系统元数据的持久化数据结构往往采用 BTree。 BTree 是目前最通用的数据结构，在消息系统中它可以用来广泛支持多种不同的事务性或非事务性语义。 它的确也带来了一个非常高的处理开销，Btree 运算的时间复杂度为 O(log N)。一般 O(log N)被认为基本上等于常量时长，但对于磁盘操作来讲，情况就不同了。磁盘寻道时间一次要花 10ms 的时间，而且每个磁盘同时只能进行一个寻道操作，因而其并行程度很有限。因此，即使少量的磁盘寻道操作也会造成非常大的时间开销。因为存储系统混合了高速缓存操作和真正的物理磁盘操作，所以树型结构（tree structure）可观察到的性能往往是超线性的（superlinear）。更进一步讲，BTrees 需要一种非常复杂的页面级或行级锁定机制才能避免在每次操作时锁定一整颗树。实现这种机制就要为行级锁定付出非常高昂的代价，否则就必须对所有的读取操作进行串行化（serialize）。因为对磁盘寻道操作的高度依赖，就不太可能高效地从驱动器密度（drive density）的提高中获得改善，因而就不得不使用容量较小(< 100GB SASspan style='font-size:14px;font-style:normal;font-weight:400;font-family:微软雅黑,Verdana,sans-serif,宋体;color:rgb(0, 0, 0);' >维持一种比较合理的</span>数据与寻道容量之比。

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions. Though this structure would not support the rich semantics of a BTree implementation, but it has the advantage that all operations are O(1) and reads do not block writes or each other. This has obvious performance advantages since the performance is completely decoupled from the data size--one server can now take full advantage of a number of cheap, low-rotational speed 1+TB SATA drives. Though they have poor seek performance, these drives often have comparable performance for large reads and writes at 1/3 the price and 3x the capacity.

Having access to virtually unlimited disk space without penalty means that we can provide some features not usually found in a messaging system. For example, in kafka, instead of deleting a message immediately after consumption, we can retain messages for a relative long period (say a week).

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

Maximizing Efficiency

Our assumption is that the volume of messages is extremely high, indeed it is some multiple of the total number of page views for the site (since a page view is one of the activities we process). Furthermore we assume each message published is read at least once (and often multiple times), hence we optimize for consumption rather than production.

There are two common causes of inefficiency: too many network requests, and excessive byte copying.

To encourage efficiency, the APIs are built around a "message set" abstraction that naturally groups messages. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time.

译者信息▽

效率最大化

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

导致低效率的原因常见的有两个：过多的网络请求和大量的字节拷贝操作。

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

TheMessageSetimplementation is itself a very thin API that wraps a byte array or file. Hence there is no separate serialization or deserialization step required for message processing, message fields are lazily deserialized as needed (or not deserialized if not needed).

The message log maintained by the broker is itself just a directory of message sets that have been written to disk. This abstraction allows a single byte format to be shared by both the broker and the consumer (and to some degree the producer, though producer messages are checksumed and validated before being added to the log).

Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the sendfile system call. Java provides access to this system call with theFileChannel.transferToapi.

译者信息▽

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

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

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

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space
2. The application reads the data from kernel space into a user-space buffer

3. The application writes the data back into kernel space into a socket buffer
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network

This is clearly inefficient, there are four copies, two system calls. Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.

We expect a common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to kernel space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.

For more background on the sendfile and zero-copy support in Java, see this article on IBM developerworks.

译者信息▽

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

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

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

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

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

End-to-end Batch Compression

In many cases the bottleneck is actually not CPU but network. This is particularly true for a data pipeline that needs to send messages across data centers. Of course the user can always send compressed messages without any support needed from Kafka, but this can lead to very poor compression ratios as much of the redundancy is due to repetition between messages (e.g. field names in JSON or user agents in web logs or common string values). Efficient compression requires compressing multiple messages together rather than compressing each message individually. Ideally this would be possible in an end-to-end fashion—that is, data would be compressed prior to sending by the producer and remain compressed on the server, only being decompressed by the eventual consumers.

Kafka supports this by allowing recursive message sets. A batch of messages can be clumped together compressed and sent to the server in this form. This batch of messages will be delivered all to the same consumer and will remain in compressed form until it arrives there.

Kafka supports GZIP and Snappy compression protocols. More details on compression can be found here.

译者信息▽

端到端的批量压缩

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

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

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

Consumer state

Keeping track of *what* has been consumed is one of the key things a messaging system must provide. It is not intuitive, but recording this state is one of the key performance points for the system. State tracking requires updating a persistent entity and potentially causes random accesses. Hence it is likely to be bound by the seek time of the storage system not the write bandwidth (as described above).

Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker records that fact locally. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else it could go. Since the data structure used for storage in many messaging systems scale poorly, this is also a pragmatic choice--since the broker knows what is consumed it can immediately delete it, keeping the data size small.

译者信息▽

客户状态

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

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

What is perhaps not obvious, is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as **consumed** immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) then that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as **sent** not **consumed** when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as **consumed**. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

译者信息▽

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

Message delivery semantics

So clearly there are multiple possible message delivery guarantees that could be provided:

- *At most once*—this handles the first case described. Messages are immediately marked as consumed, so they can't be given out twice, but many failure scenarios may lead to losing messages.
- *At least once*—this is the second case where we guarantee each message will be delivered at least once, but in failure cases may be delivered twice.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

This problem is heavily studied, and is a variation of the "transaction commit" problem. Algorithms that provide exactly once semantics exist, two- or three-phase commits and Paxos variants being examples, but they come with some drawbacks. They typically require multiple round trips and may have poor guarantees of liveness (they can halt indefinitely). The FLP result provides some of the fundamental limitations on these algorithms.

译者信息▽

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

系统可以提供的几种可能的消息传递保障如下所示:

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

这个问题已得到广泛的研究，属于“事务提交”问题的一个变种。提供仅仅一次语义的算法已经有了，两阶段或者三阶段提交法以及 Paxos 算法的一些变种就是其中的一些例子，但它们都有与生俱来的的缺陷。这些算法往往需要多个网络往返（round trip），可能也无法很好的保证其活性（liveness）（它们可能会导致无限期停机）。FLP 结果给出了这些算法的一些基本的局限。

Kafka does two unusual things with respect to metadata. First the stream is partitioned on the brokers into a set of distinct partitions. The semantic meaning of these partitions is left up to the producer and the producer specifies which partition a message belongs to. Within a partition messages are stored in the order in which they arrive at the broker, and will be given out to consumers in that same order. This means that rather than store metadata for each message (marking it as consumed, say), we just need to store the "high water mark" for each combination of consumer, topic, and partition. Hence the total metadata required to summarize the state of the consumer is actually quite small. In Kafka we refer to this high-water mark as "the offset" for reasons that will become clear in the implementation section.

译者信息▽

Kafka 对元数据做了两件很不寻常的事情。一件是，代理将数据流划分为一组互相独立的分区。这些分区的语义由生产者定义，由生产者来指定每条消息属于哪个分区。一个分区内的消息以到达代理的时间为准进行排序，将来按此顺序将消息发送给使用者。这么一来，就用不着为每一天消息保存一条

元数据（比如说，将消息标示为已使用）了，我们只需为使用者、话题和分区的每种组合记录一个“最高水位标记”（high water mark）即可。因此，标示使用者状态所需的元数据总量实际上特别小。在 Kafka 中，我们将该最高水位标记称为“偏移量”（offset），这么叫的原因将在实现细节部分讲解。

Consumer state

In Kafka, the consumers are responsible for maintaining state information (offset) on what has been consumed. Typically, the Kafka consumer library writes their state data to zookeeper. However, it may be beneficial for consumers to write state data into the same datastore where they are writing the results of their processing. For example, the consumer may simply be entering some aggregate value into a centralized transactional OLTP database. In this case the consumer can store the state of what is consumed in the same transaction as the database modification. This solves a distributed consensus problem, by removing the distributed part! A similar trick works for some non-transactional systems as well. A search system can store its consumer state with its index segments. Though it may provide no durability guarantees, this means that the index is always in sync with the consumer state: if an unflushed index segment is lost in a crash, the indexes can always resume consumption from the latest checkpointed offset. Likewise our Hadoop load job which does parallel loads from Kafka, does a similar trick. Individual mappers write the offset of the last consumed message to HDFS at the end of the map task. If a job fails and gets restarted, each mapper simply restarts from the offsets stored in HDFS.

There is a side benefit of this decision. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

译者信息▽

使用者的状态

在 Kafka 中，由使用者负责维护反映哪些消息已被使用的状态信息（偏移量）。典型情况下，Kafka 使用者的 library 会把状态数据保存到 Zookeeper 之中。然而，让使用者将状态信息保存到保存它们的消息处理结果的那个数据存储（datastore）中也许会更佳。例如，使用者也许就是要把一些统计值存储到集中式事物 OLTP 数据库中，在这种情况下，使用者可以在进行那个数据库数据更改的同一个事务中将消息使用状态信息存储起来。这样就消除了分布式的部分，从而解决了分布式中的一致性

问题！这在非事务性系统中也有类似的技巧可用。搜索系统可用将使用者状态信息同它的索引段（index segment）存储到一起。尽管这么做可能无法保证数据的持久性（durability），但却可用让索引同使用者状态信息保存同步：如果由于宕机造成有一些没有刷新到磁盘的索引段信息丢了，我们总是可用从上次建立检查点（checkpoint）的偏移量处继续对索引进行处理。与此类似，Hadoop 的加载作业（load job）从 Kafka 中并行加载，也有相同的技巧可用。每个 Mapper 在 map 任务结束前，将它使用的最后一个消息的偏移量存入 HDFS。

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

Push vs. pull

A related question is whether consumers should pull data from brokers or brokers should push data to the subscriber. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some recent systems, such as scribe and flume, focusing on log aggregation, follow a very different push based path where each node acts as a broker and data is pushed downstream. There are pros and cons to both approaches. However a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal, is generally for the consumer to be able to consume at the maximum possible rate; unfortunately in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

译者信息▽

Push 和 Pull

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

Distribution

Kafka is built to be run across a cluster of machines as the common case. There is no central "master" node. Brokers are peers to each other and can be added and removed at anytime without any manual configuration changes. Similarly, producers and consumers can be started dynamically at any time. Each broker registers some metadata (e.g., available topics) in Zookeeper. Producers and consumers can use Zookeeper to discover topics and to co-ordinate the production and consumption. The details of producers and consumers will be described below.

译者信息▽

分发

Kafka 通常情况下是运行在集群中的服务器上。没有中央的“主”节点。代理彼此之间是对等的，不需要任何手动配置即可可随时添加和删除。同样，生产者和消费者可以在任何时候开启。 每个代理

都可以在 Zookeeper(分布式协调系统)中注册的一些元数据（例如，可用的主题）。生产者和消费者可以使用 Zookeeper 发现主题和相互协调。关于生产者和消费者的细节将在下面描述。

Producer

Automatic producer load balancing

Kafka supports client-side load balancing for message producers or use of a dedicated load balancer to balance TCP connections. A dedicated layer-4 load balancer works by balancing TCP connections over Kafka brokers. In this configuration all messages from a given producer go to a single broker. The advantage of using a level-4 load balancer is that each producer only needs a single TCP connection, and no connection to zookeeper is needed. The disadvantage is that the balancing is done at the TCP connection level, and hence it may not be well balanced (if some producers produce many more messages than others, evenly dividing up the connections per broker may not result in evenly dividing up the messages per broker).

译者信息▽

生产者

生产者自动负载均衡

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

Client-side zookeeper-based load balancing solves some of these problems. It allows the producer to dynamically discover new brokers, and balance load on a per-request basis. Likewise it allows the producer to partition data according to some key instead of randomly, which enables stickiness on the consumer (e.g. partitioning data consumption by user id). This feature is called "semantic partitioning", and is described in more detail below.

The working of the zookeeper-based load balancing is described below. Zookeeper watchers are registered on the following events—

- a new broker comes up

- a broker goes down
- a new topic is registered
- a broker gets registered for an existing topic

Internally, the producer maintains an elastic pool of connections to the brokers, one per broker. This pool is kept updated to establish/maintain connections to all the live brokers, through the zookeeper watcher callbacks. When a producer request for a particular topic comes in, a broker partition is picked by the partitioner (see section on semantic partitioning). The available producer connection is used from the pool to send the data to the selected broker partition.

译者信息▽

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

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

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

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

Asynchronous send

Asynchronous non-blocking operations are fundamental to scaling messaging systems. In Kafka, the producer provides an option to use asynchronous dispatch of produce requests (producer.type=async). This allows buffering of produce requests in a in-memory queue and batch sends that are triggered by a time interval or a pre-configured batch size. Since data is typically published from set of heterogenous machines producing data at variable rates, this asynchronous buffering helps generate uniform traffic to the brokers, leading to better network utilization and higher throughput.

译者信息▽

异步发送

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

Semantic partitioning

Consider an application that would like to maintain an aggregation of the number of profile visitors for each member. It would like to send all profile visit events for a member to a particular partition and, hence, have all updates for a member to appear in the same stream for the same consumer thread. The producer has the capability to be able to semantically map messages to the available kafka nodes and partitions. This allows partitioning the stream of messages with some semantic partition function based on some key in the message to spread them over broker machines. The partitioning function can be customized by providing an implementation of the kafka.producer.Partitioner interface, default being the random partitioner. For the example above, the key would be member_id and the partitioning function would be hash(member_id)%num_partitions.

译者信息▽

语义分区

下面看看一个想要为每个成员统计一个个人空间访客总数的程序该怎么做。应该把一个成员的所有个人空间访问事件发送给某特定分区，因此就可以把对一个成员的所有更新都放在同一个使用者线程中

的同一个事件流中。生产者具有从语义上将消息映射到有效的 Kafka 节点和分区之上的能力。这样就可以用一个语义分区函数将消息流按照消息中的某个键值进行分区，并将不同分区发送给各自相应的代理。通过实现 kafak.producer.Partitioner 接口，可以对分区函数进行定制。在缺省情况下使用的是随即分区函数。上例中，那个键值应该是 member_id，分区函数可以是

hash(member_id)%num_partitions。

Support for Hadoop and other batch data load

Scalable persistence allows for the possibility of supporting batch data loads that periodically snapshot data into an offline system for batch processing. We make use of this for loading data into our data warehouse and Hadoop clusters.

Batch processing happens in stages beginning with the data load stage and proceeding in an acyclic graph of processing and output stages (e.g. as supported here). An essential feature of support for this model is the ability to re-run the data load from a point in time (in case anything goes wrong).

In the case of Hadoop we parallelize the data load by splitting the load over individual map tasks, one for each node/topic/partition combination, allowing full parallelism in the loading. Hadoop provides the task management, and tasks which fail can restart without danger of duplicate data.

译者信息▽

对 Hadoop 以及其它批量数据装载的支持

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

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

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

Implementation Details

The following gives a brief description of some relevant lower-level implementation details for some parts of the system described in the above section.

API Design

Producer APIs

The Producer API that wraps the 2 low-level producers -kafka.producer.SyncProducerandkafka.producer.async.AsyncProducer.

```
class Producer {   /* Sends the data, partitioned by key to the topic using either the */  /* synchronous or the asynchronous producer */  public void send(kafka.javaapi.producer.ProducerData producerData);  /* Sends a list of data, partitioned by key to the topic using either */  /* the synchronous or the asynchronous producer */  public void send(java.util.List< kafka javaapiproducerProducerData> producerData);  /* Closes the producer and cleans up */     public void close();}
```

译者信息▽

实施细则

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

API 设计

生产者 APIs

生产者 API 是给两个底层生产者的再封装

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

```
class Producer {  /* Sends the data, partitioned by key to the topic using either the */  /* synchronous or the asynchronous producer */  public void send(kafka.javaapi.producer.ProducerData producerData);  /* Sends a list of data, partitioned by key to the topic using either */  /* the synchronous or the asynchronous producer */  public void send(java.util.List< kafka javaapiproducerProducerData> producerData);  /* Closes the producer and cleans up */      public void close();}
```

The goal is to expose all the producer functionality through a single API to the client. The

new producer -

- can handle queueing/buffering of multiple producer requests and asynchronous dispatch of the batched data -

  kafka.producer.Producerprovides the ability to batch multiple produce requests (producer.type=async), before serializing and dispatching them to the appropriate kafka broker partition. The size of the batch can be controlled by a few config parameters. As events enter a queue, they are buffered in a queue, until eitherqueue.timeorbatch.sizeis reached. A background thread (kafka.producer.async.ProducerSendThread) dequeues the batch of data and lets thekafka.producer.EventHandlerserialize and send the data to the appropriate kafka broker partition. A custom event handler can be plugged in through theevent.handlerconfig parameter. At various stages of this producer queue pipeline, it is helpful to be able to inject callbacks, either for plugging in custom logging/tracing code or custom monitoring logic. This is possible by implementing thekafka.producer.async.CallbackHandlerinterface and settingcallback.handlerconfig parameter to that class.

- handles the serialization of data through a user-specifiedEncoder-

  ```
  interface Encoder {  public Message toMessage(T data);}
  ```

  The default is the no-opkafka.serializer.DefaultEncoder

- provides zookeeper based automatic broker discovery -

  The zookeeper based broker discovery and load balancing can be used by specifying the zookeeper connection url through thezk.connectconfig parameter. For some applications, however, the dependence on zookeeper is inappropriate. In that case, the producer can take in a static list of brokers through thebroker.listconfig parameter. Each produce requests gets routed to a random broker partition in this case. If that broker is down, the produce request fails.

- provides software load balancing through an optionally user-specifiedPartitioner-

  The routing decision is influenced by thekafka.producer.Partitioner.

  ```
  interface Partitioner {   int partition(T key, int numPartitions);}
  ```

  The partition API uses the key and the number of available broker partitions to return a partition id. This id is used as an index into a sorted list of broker_ids and partitions to pick a broker partition for the producer request. The default partitioning strategy ishash(key)%numPartitions. If the key is null, then a random broker partition is picked. A custom partitioning strategy can also be plugged in using thepartitioner.classconfig parameter.

译者信息▽

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

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

```
interfaceEncoder {
publicMessage toMessage(T data);
}
```

- Encoder 的缺省值是一个什么活都不干的 kafka.serializer.DefaultEncoder。
- 提供基于 zookeeper 的代理自动发现功能 —— 通过使用 zk.connect 配置参数指定 zookeeper的连接 url，就能够使用基于zookeeper的代理发现和负载均衡功能。在有些应用场合，可能不太适合于依赖 zookeeper。在这种情况下，生产者可以从 broker.list 这个配置参数中获得一个代理的静态列表，每个生产请求会被随即的分配给各代理分区。如果相应的代理宕机，那么生产请求就会失败。
- 通过使用一个可选性的、由用户指定的 Partitioner，提供由软件实现的负载均衡功能 —— 数据发送路径选择决策受 kafka.producer.Partitioner 的影响。

```
interfacePartitioner {
intpartition(T key, intnumPartitions);
}
```

- 分区 API 根据相关的键值以及系统中具有的代理分区的数量返回一个分区 id。将该 id 用作索引，在 broker_id 和 partition 组成的经过排序的列表中为相应的生产者请求找出一个代理分区。缺省的分区策略是 hash(key)%numPartitions。如果 key 为 null，那就进行随机选择。使用 partitioner.class 这个配置参数也可以插入自定义的分区策略。
- Consumer APIs
- We have 2 levels of consumer APIs. The low-level "simple" API maintains a connection to a single broker and has a close correspondence to the network requests sent to the server. This API is completely stateless, with the offset being passed in on every request, allowing the user to maintain this metadata however they choose.

- The high-level API hides the details of brokers from the consumer and allows consuming off the cluster of machines without concern for the underlying topology. It also maintains the state of what has been consumed. The high-level API also provides the ability to subscribe to topics that match a filter expression (i.e., either a whitelist or a blacklist regular expression).
- 译者信息▽
- 使用者 API
- 我们有两个层次的使用者 API。底层比较简单的 API 维护了一个同单个代理建立的连接，完全同发送给服务器的网络请求相吻合。该 API 完全是无状态的，每个请求都带有一个偏移量作为参数，从而允许用户以自己选择的任意方式维护该元数据。
- 高层 API 对使用者隐藏了代理的具体细节，让使用者可运行于集群中的机器之上而无需关心底层的拓扑结构。它还维护着数据使用的状态。高层 API 还提供了订阅同一个过滤表达式(例如，白名单或黑名单的正则表达式)相匹配的多个话题的能力。

Low-level API

- `class SimpleConsumer {   /* Send fetch request to a broker and get back a set of messages. */   public ByteBufferMessageSet fetch(FetchRequest request); /* Send a list of fetch requests to a broker and get back a response set. */   public MultiFetchResponse multifetch(List fetches); /**   * Get a list of valid offsets (up to maxSize) before the given time.   * The result is a list of offsets, in descending order.   * @param time: time in millisecs,   *   if set to OffsetRequest$.MODULE$.LATIEST_TIME(), get from the latest offset available.   *   if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the earliest offset available.   */ public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);}`
- The low-level API is used to implement the high-level API as well as being used directly for some of our offline consumers (such as the hadoop consumer) which have particular requirements around maintaining state.

High-level API

- `/* create a connection to the cluster */ ConsumerConnector connector = Consumer.create(consumerConfig);interface ConsumerConnector {  /**   * This method is used to get a list of KafkaStreams, which are iterators over`

```
* MessageAndMetadata objects from which you can obtain messages and their
* associated metadata (currently only topic).   *  Input: a map of    *
Output: a map of    */  public Map< createMessageStreams(Map
topicCountMap);   /**   * You can also obtain a list of KafkaStreams, that
iterate over messages   * from topics that match a TopicFilter. (A
TopicFilter encapsulates a   * whitelist or a blacklist which is a standard
Java regex.)   */  public List
createMessageStreamsByFilter(     TopicFilter topicFilter, int
numStreams);  /* Commit the offsets of all messages consumed so far. */
public commitOffsets()    /* Shut down the connector */  public shutdown()}
```

- This API is centered around iterators, implemented by the KafkaStream class. Each KafkaStream represents the stream of messages from one or more partitions on one or more servers. Each stream is used for single threaded processing, so the client can provide the number of desired streams in the create call. Thus a stream may represent the merging of multiple server partitions (to correspond to the number of processing threads), but each partition only goes to one stream.
- The createMessageStreams call registers the consumer for the topic, which results in rebalancing the consumer/broker assignment. The API encourages creating many topic streams in a single call in order to minimize this rebalancing. The createMessageStreamsByFilter call (additionally) registers watchers to discover new topics that match its filter. Note that each stream that createMessageStreamsByFilter returns may iterate over messages from multiple topics (i.e., if multiple topics are allowed by the filter).
- 译者信息▽

底层 API

-
  ```
  class SimpleConsumer {      /* Send fetch request to a broker and get
  back a set of messages. */   public ByteBufferMessageSet
  fetch(FetchRequest request);  /* Send a list of fetch requests to a
  broker and get back a response set. */   public MultiFetchResponse
  multifetch(List fetches);  /**   * Get a list of valid offsets (up to
  maxSize) before the given time.   * The result is a list of offsets,
  in descending order.   * @param time: time in millisecs,   *
  if set to OffsetRequest$.MODULE$.LATIEST_TIME(), get from the latest
  offset available.   *             if set to
  OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the earliest offset
  available.   */  public long[] getOffsetsBefore(String topic, int
  partition, long time, int maxNumOffsets);}
  ```
- 底层 API 不但用于实现高层 API，而且还直接用于我们的离线使用者（比如 Hadoop 这个使用者），这些使用者还对状态的维护有比较特定的需求。
- 高层 API

-
  ```
  /* create a connection to the cluster */ ConsumerConnector connector
  = Consumer.create(consumerConfig);interface ConsumerConnector {
  /**   * This method is used to get a list of KafkaStreams, which are
  iterators over   * MessageAndMetadata objects from which you can obtain
  messages and their   * associated metadata (currently only topic).   *
  Input: a map of    *  Output: a map of    */  public Map<
  createMessageStreams(Map topicCountMap);   /**   * You can also obtain
  a list of KafkaStreams, that iterate over messages   * from topics that
  match a TopicFilter. (A TopicFilter encapsulates a   * whitelist or a
  blacklist which is a standard Java regex.)   */  public List
  createMessageStreamsByFilter(     TopicFilter topicFilter, int
  numStreams);  /* Commit the offsets of all messages consumed so far.
  */  public commitOffsets()   /* Shut down the connector */  public
  shutdown()}
  ```
- 该 API 的中心是一个由 KafkaStream 这个类实现的迭代器（iterator）。每个 KafkaStream 都代表着一个从一个或多个分区到一个或多个服务器的消息流。每个流都是使用单个线程进行处理的，所以，该 API 的使用者在该 API 的创建调用中可以提供所需的任意个数的流。这样，一个流可能会代表多个服务器分区的合并（同处理线程的数目相同），但每个分区只会把数据发送给一个流中。
- createMessageStreams 方法为使用者注册到相应的话题之上，这将导致需要对使用者/代理的分配情况进行重新平衡。为了将重新平衡操作减少到最小。该 API 鼓励在一次调用中就创建多个话题流。createMessageStreamsByFilter 方法为发现同其过滤条件想匹配的话题（额外地）注册了多个监视器（watchers）。应该注意，createMessageStreamsByFilter 方法所返回的每个流都可能会对多个话题进行迭代（比如，在满足过滤条件的话题有多个的情况下）。

Network Layer

- The network layer is a fairly straight-forward NIO server, and will not be described in great detail. The sendfile implementation is done by giving theMessageSetinterface awriteTomethod. This allows the file-backed message set to use the more efficienttransferToimplementation instead of an in-process buffered write. The threading model is a single acceptor thread and *N* processor threads which handle a fixed number of connections each. This design has been pretty thoroughly tested elsewhere and found to be simple to implement and fast. The protocol is kept quite simple to allow for future implementation of clients in other languages.

- 译者信息▽

网络层

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

Messages

- Messages consist of a fixed-size header and variable length opaque byte array payload. The header contains a format version and a CRC32 checksum to detect corruption or truncation. Leaving the payload opaque is the right decision: there is a great deal of progress being made on serialization libraries right now, and any particular choice is unlikely to be right for all uses. Needless to say a particular application using Kafka would likely mandate a particular serialization type as part of its usage. TheMessageSetinterface is simply an iterator over messages with specialized methods for bulk reading and writing to an NIOChannel.
- 译者信息▽

消息

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

MessageSet 接口就是一个使用特殊的方法对 NIOChannel 进行大宗数据读写（bulk reading and writing to an NIOChannel）的消息迭代器。

Message Format

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

Log

- A log for a topic named "my_topic" with two partitions consists of two directories (namelymy_topic_0andmy_topic_1) populated with data files containing the messages for that topic. The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer *N* storing the message length which is followed by the *N* message bytes. Each message is uniquely identified by a 64-bit integer *offset* giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition. The on-disk format of each message is given below. Each log file is named with the offset of the first message it contains. So the first file created will be 00000000000.kafka, and each additional file will have an integer name roughly *S* bytes from the previous file where *S* is the max log file size given in the configuration.
- 译者信息▽

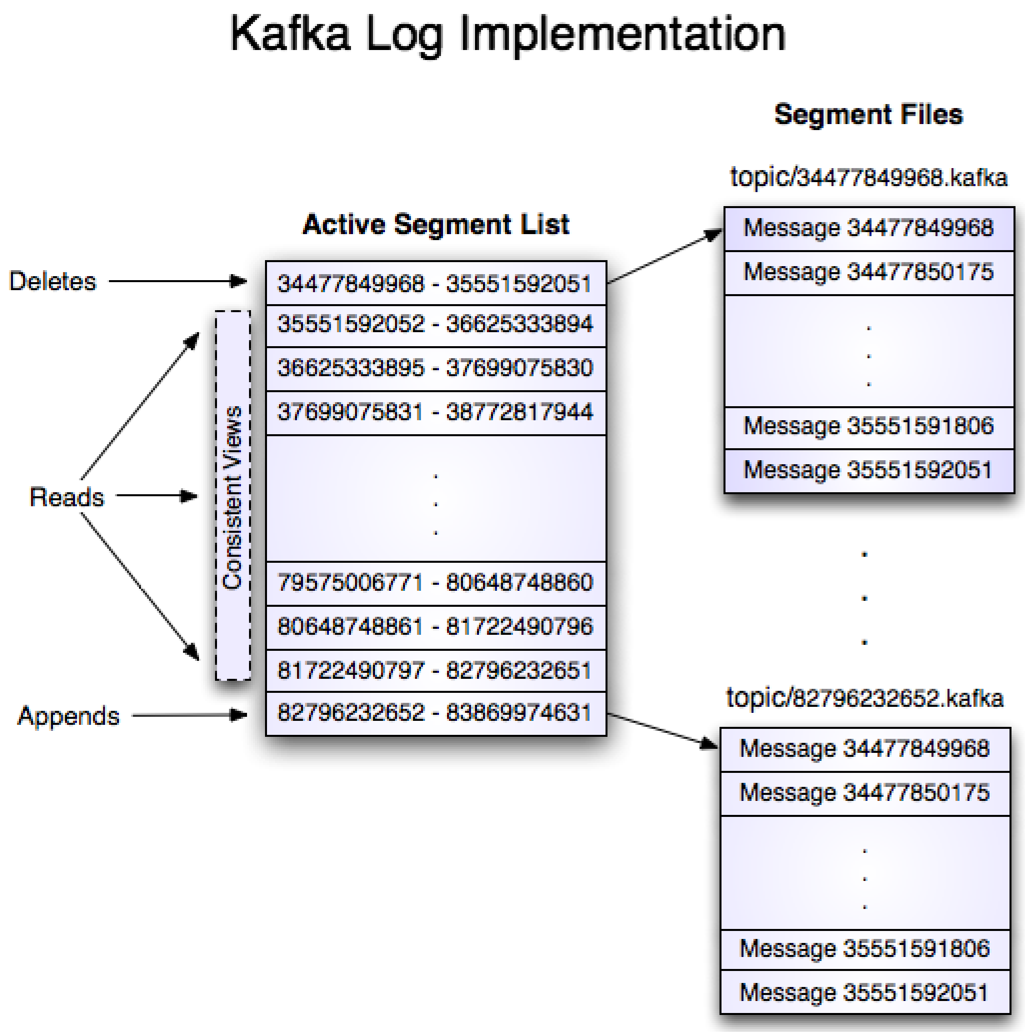
消息的格式

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

日志

- 具有两个分区的、名称为"my_topic"的话题的日志由两个目录组成（即：my_topic_0 和 my_topic_1)，目录中存储的是内容为该话题的消息的数据文件。日志的文件格式是一系列的“日志项”；每条日志项包含一个表示消息长度的 4 字节整数 N，其后接着保存的是 N 字节的消息。每条消息用一个 64 位的整数偏移量进行唯一性标示，该偏移量表示了该消息在那个分区中的那个话题下发送的所有消息组成的消息流中所处的字节位置。每条消息在磁盘上的格式如下文所示。每个日志文件的以它所包含的第一条消息的偏移量来命名。因此，第一个创建出来的文件的名字将为 00000000000.kafka，随后每个后加的文件的名字将是前一个文件的文件名大约再加 S 个字节所得的整数，其中，S 是配置文件中指定的最大日志文件的大小。
- The exact binary format for messages is versioned and maintained as a standard interface so message sets can be transfered between producer, broker, and client without recopying or conversion when desirable. This format is as follows:

-
  ```
  On-disk format of a messagemessage length : 4 bytes (value: 1+4+n) "magic" value  : 1 bytecrc           : 4 bytespayload       : n bytes
  ```
- The use of the message offset as the message id is unusual. Our original idea was to use a GUID generated by the producer, and maintain a mapping from GUID to offset on each broker. But since a consumer must maintain an ID for each server, the global uniqueness of the GUID provides no value. Furthermore the complexity of maintaining the mapping from a random id to an offset requires a heavy weight index structure which must be synchronized with disk, essentially requiring a full persistent random-access data structure. Thus to simplify the lookup structure we decided to use a simple per-partition atomic counter which could be coupled with the partition id and node id to uniquely identify a message; this makes the lookup structure simpler, though multiple seeks per consumer request are still likely. However once we settled on a counter, the jump to directly using the offset seemed natural—both after all are monotonically increasing integers unique to a partition. Since the offset is hidden from the consumer API this decision is ultimately an implementation detail and we went with the more efficient approach.



-

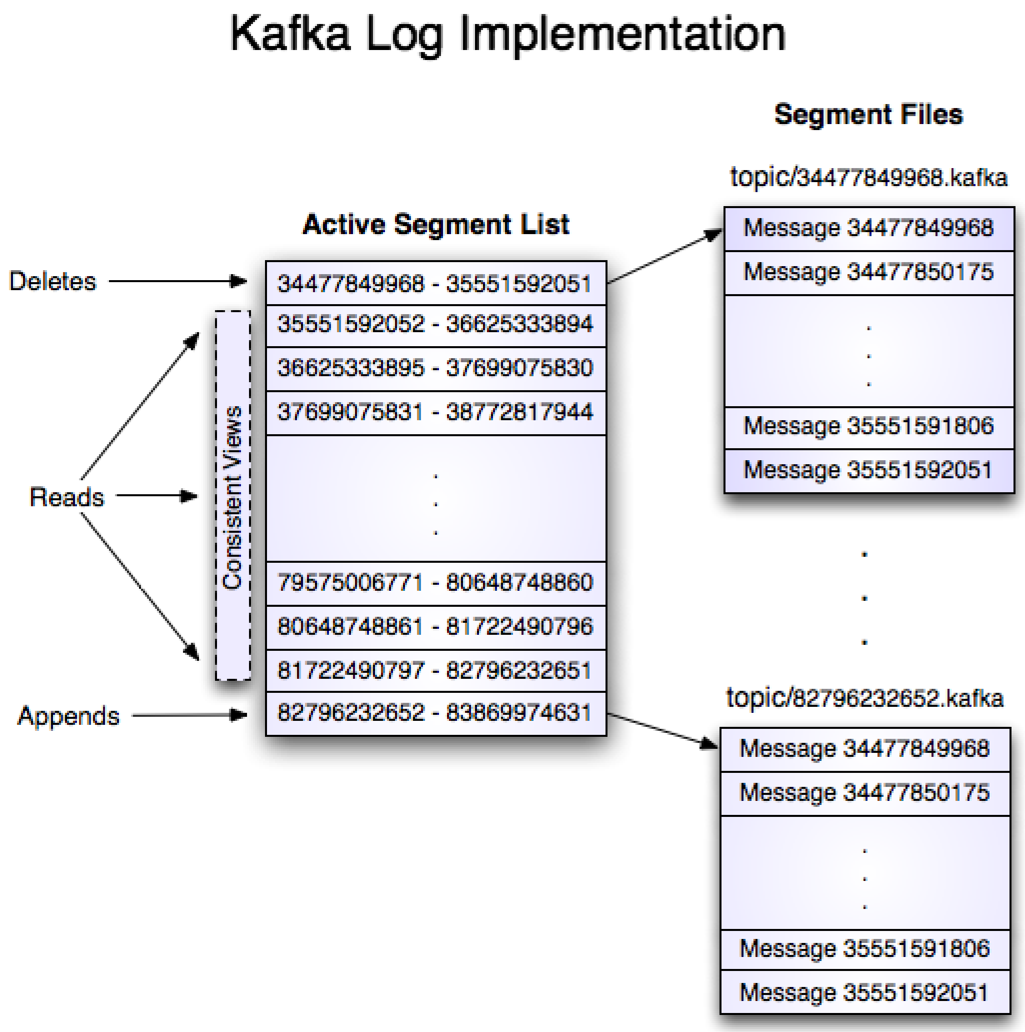
  译者信息▽

- 消息的确切的二进制格式都有版本，它保持为一个标准的接口，让消息集可以根据需要在生产者、代理、和使用者直接进行自由传输而无须重新拷贝或转换。其格式如下所示：

-
  ```
  On-disk format of a messagemessage length : 4 bytes (value: 1+4+n)
  "magic" value  : 1 bytecrc           : 4 bytespayload       : n bytes
  ```

- 将消息的偏移量作为消息的可不常见。我们原先的想法是使用由生产者产生的 GUID 作为消息 id，然后在每个代理上作一个从 GUID 到偏移量的映射。但是，既然使用者必须为每个服务器维护一个 ID，那么 GUID 所具有的全局唯一性就失去了价值。更有甚者，维护将从一个随机数到偏移量的映射关系带来的复杂性，使得我们必须使用一种重量级的索引结构，而且这种结构还必须与磁盘保持同步，这样我们还就必须使用一种完全持久化的、需随机访问的

数据结构。如此一来，为了简化查询结构，我们就决定使用一个简单的依分区的原子计数器（atomic counter）这个计数器可以同分区 id 以及节点 id 结合起来唯一的指定一条消息；这种方法使得查询结构简化不少，尽管每次在处理使用者请求时仍有可能会涉及多次磁盘寻道操作。然而，一旦我们决定使用计数器，跳向直接使用偏移量作为 id 就非常自然了，毕竟两者都是分区内具有唯一性的、单调增加的整数。既然偏移量是在使用者 API 中并不会体现出来，所以这个决策最终还是属于一个实现细节，进而我们就选择了这种更加高效的方式。



-

Writes

- The log allows serial appends which always go to the last file. This file is rolled over to a fresh file when it reaches a configurable size (say 1GB). The log takes two configuration parameter *M* which gives the number of messages to write before forcing the OS to flush the file to disk, and *S* which gives a number of seconds after which a flush is forced. This gives a durability guarantee of losing at most *M* messages or *S* seconds of data in the event of a system crash.

- 译者信息▽

写操作

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

Reads

- Reads are done by giving the 64-bit logical offset of a message and an *S*-byte max chunk size. This will return an iterator over the messages contained in the *S*-byte buffer. *S* is intended to be larger than any single message, but in the event of an abnormally large message, the read can be retried multiple times, each time doubling the buffer size, until the message is read successfully. A maximum message and buffer size can be specified to make the server reject messages larger than some size, and to give a bound to the client on the maximum it need ever read to get a complete message. It is likely that the read buffer ends with a partial message, this is easily detected by the size delimiting.
- 译者信息▽

读操作

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

and then reading from that file offset. The search is done as a simple binary search variation against an in-memory range maintained for each file.

- The log provides the capability of getting the most recently written message to allow clients to start subscribing as of "right now". This is also useful in the case the consumer fails to consume its data within its SLA-specified number of days. In this case when the client attempts to consume a non-existant offset it is given an OutOfRangeException and can either reset itself or fail as appropriate to the use case.
- 译者信息▽
- 从某偏移量开始进行日志读取的实际过程需要先找出存储所需数据的日志段文件，从全局偏移量计算出文件内偏移量，然后再从该文件偏移量处开始读取。搜索过程通过对每个文件保存在内存中的范围值进行一种变化后的二分查找完成。
- 日志提供了获取最新写入的消息的功能，从而允许从“当下”开始消息订阅。这个功能在使用者在 SLA 规定的天数内没能正常使用数据的情况下也很有用。当使用者企图从一个并不存在的偏移量开始使用数据时就会出现这种情况，此时使用者会得到一个 OutOfRangeException 异常，它可以根据具体的使用情况对自己进行重启或者仅仅失败而退出。
- The following is the format of the results sent to the consumer.
- `MessageSetSend (fetch result)total length     : 4 byteserror code      : 2 bytesmessage 1       : x bytes...message n       : x bytes`
- `MultiMessageSetSend (multiFetch result)total length      : 4 byteserror code       : 2 bytesmessageSetSend 1...messageSetSend n`

Deletes

- Data is deleted one log segment at a time. The log manager allows pluggable delete policies to choose which files are eligible for deletion. The current policy deletes any log with a modification time of more than *N* days ago, though a policy which retained the last *N* GB could also be useful. To avoid locking reads while still allowing deletes that modify the segment list we use a copy-on-write style segment list implementation that provides consistent views to allow a binary search to proceed on an immutable static snapshot view of the log segments while deletes are progressing.
- 译者信息▽
- 以下是发送给数据使用者（consumer）的结果的格式。
- `MessageSetSend (fetch result)total length     : 4 byteserror code       : 2 bytesmessage 1       : x bytes...message n       : x bytes`

- `MultiMessageSetSend (multiFetch result)total length      : 4 byteserror code        : 2 bytesmessageSetSend 1...messageSetSend n`

删除

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

Guarantees

The log provides a configuration parameter *M* which controls the maximum number of messages that are written before forcing a flush to disk. On startup a log recovery process is run that iterates over all messages in the newest log segment and verifies that each message entry is valid. A message entry is valid if the sum of its size and offset are less than the length of the file AND the CRC32 of the message payload matches the CRC stored with the message. In the event corruption is detected the log is truncated to the last valid offset.

Note that two kinds of corruption must be handled: truncation in which an unwritten block is lost due to a crash, and corruption in which a nonsense block is ADDED to the file. The reason for this is that in general the OS makes no guarantee of the write order between the file inode and the actual block data so in addition to losing written data the file can gain nonsense data if the inode is updated with a new size but a crash occurs before the block containing that data is not written. The CRC detects this corner case, and prevents it from corrupting the log (though the unwritten messages are, of course, lost).

译者信息▽

数据正确性保证

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

要注意，这里有两种必需处理的数据损坏情况：由于系统崩溃造成的未被正常写入的数据块（block）因而需要截断的情况以及由于文件中被加入了毫无意义的数据块而造成的数据损坏情况。造成数据损坏的原因是，一般来说 OS 并不能保证文件索引节点（inode）和实际数据块这两者的写入顺序，因

此，除了可能会丢失未刷新的已写入数据之外，在索引节点已经用新的文件大小更新了但在将数据块写入磁盘块之前发生了系统崩溃的情况下，文件就可能会获得一些毫无意义的数据。CRC 值就是用于这种极端情况，避免由此造成整个日志文件的损坏(尽管未得到保存的消息当然是真的找不回来了)。

Distribution

Zookeeper Directories

The following gives the zookeeper structures and algorithms used for co-ordination between consumers and brokers.

Notation

When an element in a path is denoted [xyz], that means that the value of xyz is not fixed and there is in fact a zookeeper znode for each possible value of xyz. For example /topics/[topic] would be a directory named /topics containing a sub-directory for each topic name. Numerical ranges are also given such as [0...5] to indicate the subdirectories 0, 1, 2, 3, 4. An arrow -< is used to indicate the contents of a znode. For example /hello -< world would indicate a znode /hello containing the value "world".

译者信息▽

分发

Zookeeper 目录

接下来讨论 zookeeper 用于在使用者和代理直接进行协调的结构和算法。

记法

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

Broker Node Registry

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

This is a list of all present broker nodes, each of which provides a unique logical broker id which identifies it to consumers (which must be given as part of its configuration). On startup, a broker node registers itself by creating a znode with the logical broker id under /brokers/ids. The purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) is an error.

Since the broker registers itself in zookeeper using ephemeral znodes, this registration is dynamic and will disappear if the broker is shutdown or dies (thus notifying consumers it is no longer available).

Broker Topic Registry

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

Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

译者信息▽

代理节点的注册

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

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

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

代理话题的注册

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

每个代理会都要注册在某话题之下，注册后它会维护并保存该话题的分区总数。

Consumers and Consumer Groups

Consumers of topics also register themselves in Zookeeper, in order to balance the consumption of data and track their offsets in each partition for each broker they consume from.

Multiple consumers can form a group and jointly consume a single topic. Each consumer in the same group is given a shared group_id. For example if one consumer is your foobar process, which is run across three machines, then you might assign this group of consumers the id "foobar". This group id is provided in the configuration of the consumer, and is your way to tell the consumer which group it belongs to.

The consumers in a group divide up the partitions as fairly as possible, each partition is consumed by exactly one consumer in a consumer group.

译者信息▽

使用者和使用者小组

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

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

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

Consumer Id Registry

In addition to the group_id which is shared by all consumers in a group, each consumer is given a transient, unique consumer_id (of the form hostname:uuid) for identification purposes. Consumer ids are registered in the following directory.

```
/consumers/[group_id]/ids/[consumer_id] --< {"topic1": #streams, ..., "topicN":
#streams} (ephemeral node)
```

Each of the consumers in the group registers under its group and creates a znode with its consumer_id. The value of the znode contains a map of . This id is simply used to identify each of the consumers which is currently active within a group. This is an ephemeral node so it will disappear if the consumer process dies.译者信息▽

使用者 ID 的注册

除了小组内的所有使用者都要共享一个 group_id 之外，每个使用者为了要同其它使用者区别开来，还要有一个非永久性的、具有唯一性的 consumer_id(采用 hostname:uuid 的形式）。 consumer_id 要在以下的目录中进行注册。

```
/consumers/[group_id]/ids/[consumer_id] --< {"topic1": #streams, ...,
"topicN": #streams} (ephemeral node)
```

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

Consumer Offset Tracking

Consumers track the maximum offset they have consumed in each partition. This value is stored in a zookeeper directory

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

Partition Owner registry

Each broker partition is consumed by a single consumer within a given consumer group. The consumer must establish its ownership of a given partition before any consumption can begin. To establish its ownership, a consumer writes its own id in an ephemeral node under the particular broker partition it is claiming.

```
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id] --<
consumer_node_id (ephemeral node)
```

译者信息▽

数据使用者偏移追踪

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

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

分区拥有者注册表

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

```
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id] --<
consumer_node_id (ephemeral node)
```

Broker node registration

The broker nodes are basically independent, so they only publish information about what they have. When a broker joins, it registers itself under the broker node registry directory and writes information about its host name and port. The broker also register the list of existing topics and their logical partitions in the broker topic registry. New topics are registered dynamically when they are created on the broker.

Consumer registration algorithm

When a consumer starts, it does the following:

1. Register itself in the consumer id registry under its group.

2. Register a watch on changes (new consumers joining or any existing consumers leaving) under the consumer id registry. (Each change triggers rebalancing among all consumers within the group to which the changed consumer belongs.)
3. Register a watch on changes (new brokers joining or any existing brokers leaving) under the broker id registry. (Each change triggers rebalancing among all consumers in all consumer groups.)
4. If the consumer creates a message stream using a topic filter, it also registers a watch on changes (new topics being added) under the broker topic registry. (Each change will trigger re-evaluation of the available topics to determine which topics are allowed by the topic filter. A new allowed topic will trigger rebalancing among all consumers within the consumer group.)
5. Force itself to rebalance within in its consumer group.

译者信息▽

代理节点的注册

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

使用者注册算法

当使用者启动时，它要做以下这些事情：

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

话题过滤器过滤出哪些话题。新过滤出来的话题将触发一次对该使用者所在的小组内所有的使用者负载均衡。）

5. 迫使自己在小组内进行重新负载均衡。

Consumer rebalancing algorithm

The consumer rebalancing algorithms allows all the consumers in a group to come into consensus on which consumer is consuming which partitions. Consumer rebalancing is triggered on each addition or removal of both broker nodes and other consumers within the same group. For a given topic and a given consumer group, broker partitions are divided evenly among consumers within the group. A partition is always consumed by a single consumer. This design simplifies the implementation. Had we allowed a partition to be concurrently consumed by multiple consumers, there would be contention on the partition and some kind of locking would be required. If there are more consumers than partitions, some consumers won't get any data at all. During rebalancing, we try to assign partitions to consumers in such a way that reduces the number of broker nodes each consumer has to connect to.

Each consumer does the following during rebalancing:

```
1. For each topic T that $C_i$ subscribes to
2.    let $P_T$ be all partitions producing topic T
3.    let $C_G$ be all consumers in the same group as $C_i$ that consume topic T
4.    sort $P_T$ (so partitions on the same broker are clustered together)
5.    sort $C_G$
6.    let i be the index position of $C_i$ in $C_G$ and let N = size($P_T$)/size($C_G$)
7.    assign partitions from i*N to (i+1)*N - 1 to consumer $C_i$
8.    remove current entries owned by $C_i$ from the partition owner registry
9.    add newly assigned partitions to the partition owner registry
         (we may need to re-try this until the original partition owner releases its ownership)
```

When rebalancing is triggered at one consumer, rebalancing should be triggered in other consumers within the same group about the same time.

译者信息▽

使用者重新负载均衡的算法

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

载均衡的过程中，我们按照尽量减少每个使用者需要连接的代理的个数的方式，尝尝试着将分区分配给使用者。

每个使用者在重新进行负载均衡时需要做下列的事情：

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

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