**KafkaMQ**

**学习手册**

**V1.0**

**张镇**

**目 录**

[1 简介 1](#_Toc491105974)

[1.1 Kafka创建背景 1](#_Toc491105975)

[1.2 Kafka简介 1](#_Toc491105976)

[1.3 为什么需要消息系统 1](#_Toc491105977)

[1.4 常用Message Queue对比 2](#_Toc491105978)

[2 Kafka架构 3](#_Toc491105979)

[2.1 拓扑结构 3](#_Toc491105980)

[2.2 相关概念 4](#_Toc491105981)

[2.3 zookeeper 节点 5](#_Toc491105982)

[3 Kafka安装和启动 5](#_Toc491105983)

[3.1 Step 1: 下载代码 5](#_Toc491105984)

[3.2 Step 2: 启动服务 5](#_Toc491105985)

[3.3 Step 3: 创建一个主题(topic) 6](#_Toc491105986)

[3.4 Step 4: 发送消息 6](#_Toc491105987)

[3.5 Step 5: 消费消息 6](#_Toc491105988)

[3.6 Step 6: 设置多个broker集群 7](#_Toc491105989)

[3.7 Step 7: 使用 Kafka Connect导入/导出数据 8](#_Toc491105990)

[3.8 Step 8: 使用Kafka Stream来处理数据 9](#_Toc491105991)

[4 producer发布消息 11](#_Toc491105992)

[4.1 写入方式 11](#_Toc491105993)

[4.2 消息路由 11](#_Toc491105994)

[4.3 写入流程 13](#_Toc491105995)

[4.4 producer delivery guarantee 14](#_Toc491105996)

[5 broker保存消息 14](#_Toc491105997)

[5.1 存储方式 14](#_Toc491105998)

[5.2 存储策略 15](#_Toc491105999)

[5.3 topic 创建与删除 15](#_Toc491106000)

[5.3.1 创建 topic 15](#_Toc491106001)

[5.3.2 删除 topic 16](#_Toc491106002)

[6 Kafka High Availability 16](#_Toc491106003)

[6.1 Kafka为何需要High Available 17](#_Toc491106004)

[6.1.1 为何需要Replication 17](#_Toc491106005)

[6.1.2 为何需要Leader Election 17](#_Toc491106006)

[6.2 Kafka HA设计解析 18](#_Toc491106007)

[6.2.1 如何将所有Replica均匀分布到整个集群 18](#_Toc491106008)

[6.2.2 Data Replication 18](#_Toc491106009)

[6.2.2.1 Propagate消息 18](#_Toc491106010)

[6.2.2.2 ACK前需要保证有多少个备份 19](#_Toc491106011)

[6.2.2.3 Leader Election算法 20](#_Toc491106012)

[6.2.2.4 如何处理所有Replica都不工作 21](#_Toc491106013)

[6.2.2.5 如何选举Leader 21](#_Toc491106014)

[6.2.3 HA相关ZooKeeper结构 22](#_Toc491106015)

[6.2.3.1 admin 22](#_Toc491106016)

[6.2.3.2 brokers 26](#_Toc491106017)

[6.2.3.3 controller 28](#_Toc491106018)

[6.2.4 broker failover过程简介 28](#_Toc491106019)

[6.3 Broker Failover过程 30](#_Toc491106020)

[6.3.1 Controller对Broker failure的处理过程 30](#_Toc491106021)

[6.3.2 创建/删除Topic 32](#_Toc491106022)

[6.3.3 Broker响应请求流程 33](#_Toc491106023)

[6.3.4 LeaderAndIsrRequest响应过程 34](#_Toc491106024)

[6.3.5 Broker启动过程 36](#_Toc491106025)

[6.3.6 Controller Failover 37](#_Toc491106026)

[6.3.7 Partition重新分配 38](#_Toc491106027)

[6.3.8 Follower从Leader Fetch数据 39](#_Toc491106028)

[6.4 Replication工具 40](#_Toc491106029)

[6.4.1 Topic Tool 40](#_Toc491106030)

[6.4.2 Replica Verification Tool 41](#_Toc491106031)

[6.4.3 Preferred Replica Leader Election Tool 41](#_Toc491106032)

[6.4.3.1 用途 41](#_Toc491106033)

[6.4.3.2 原理 41](#_Toc491106034)

[6.4.3.3 用法 42](#_Toc491106035)

[6.4.4 Kafka Reassign Partitions Tool 43](#_Toc491106036)

[6.4.4.1 用途 43](#_Toc491106037)

[6.4.4.2 原理 44](#_Toc491106038)

[6.4.4.3 用法 44](#_Toc491106039)

[6.4.5 State Change Log Merge Tool 47](#_Toc491106040)

[6.4.5.1 用途 47](#_Toc491106041)

[6.4.5.2 用法 47](#_Toc491106042)

[6.5 leader failover 47](#_Toc491106043)

[6.6 controller failover 48](#_Toc491106044)

[7 consumer 消费消息 49](#_Toc491106045)

[7.1 consumer API 49](#_Toc491106046)

[7.1.1 The high-level consumer API 49](#_Toc491106047)

[7.1.2 The SimpleConsumer API 49](#_Toc491106048)

[7.2 consumer group 50](#_Toc491106049)

[7.3 消费方式 51](#_Toc491106050)

[7.4 consumer delivery guarantee 51](#_Toc491106051)

[7.5 consumer rebalance 52](#_Toc491106052)

[8 borker配置 53](#_Toc491106053)

[8.1 最小配置 53](#_Toc491106054)

[8.2 kafka提供的borker配置 53](#_Toc491106055)

[8.3 配置详细说明 55](#_Toc491106056)

[8.4 topic level 配置 65](#_Toc491106057)

[9 注意事项 68](#_Toc491106058)

[9.1 producer 无法发送消息的问题 68](#_Toc491106059)

[9.2 kafka consumer防止数据丢失 69](#_Toc491106060)

# 简介

## Kafka创建背景

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。现在它已被[多家不同类型的公司](https://cwiki.apache.org/confluence/display/KAFKA/Powered+By) 作为多种类型的数据管道和消息系统使用。

活动流数据是几乎所有站点在对其网站使用情况做报表时都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。

## Kafka简介

Kafka是一种分布式的，基于发布/订阅的消息系统。主要设计目标如下：

* 以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间复杂度的访问性能。
* 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条以上消息的传输。
* 支持Kafka Server间的消息分区，及分布式消费，同时保证每个Partition内的消息顺序传输。
* 同时支持离线数据处理和实时数据处理。
* Scale out：支持在线水平扩展。

## 为什么需要消息系统

**1.解耦：**允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

**2.冗余：**消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

**3.扩展性：**因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

**4.灵活性 & 峰值处理能力：**在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

**5.可恢复性：**系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

**6.顺序保证：**在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）

**7.缓冲：**有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

**8.异步通信：**很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

## 常用Message Queue对比

* **RabbitMQ**

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

* **Redis**

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

* **ZeroMQ**

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演这个服务器角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

* **ActiveMQ**

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

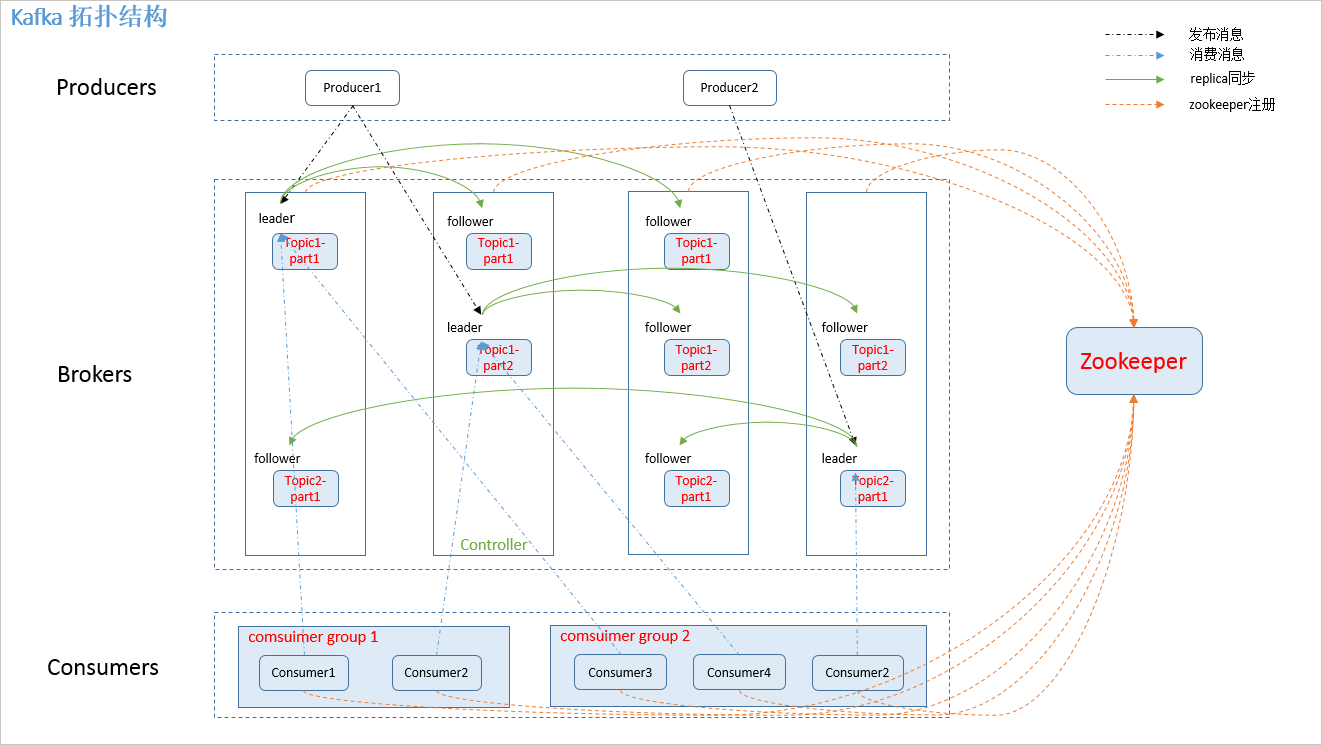
* **Kafka/Jafka**

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

# Kafka架构

## 拓扑结构

如下图：



## 相关概念

kafka 相关名词解释如下：

**1.producer：**消息生产者，发布消息到 kafka 集群的终端或服务。

**2.broker：**kafka 集群中包含的服务器。

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

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

**5.consumer：**从 kafka 集群中消费消息的终端或服务。

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

**7.replica：**partition 的副本，保障 partition 的高可用。

**8.leader：**replica 中的一个角色，producer和consumer 只跟leader 交互。

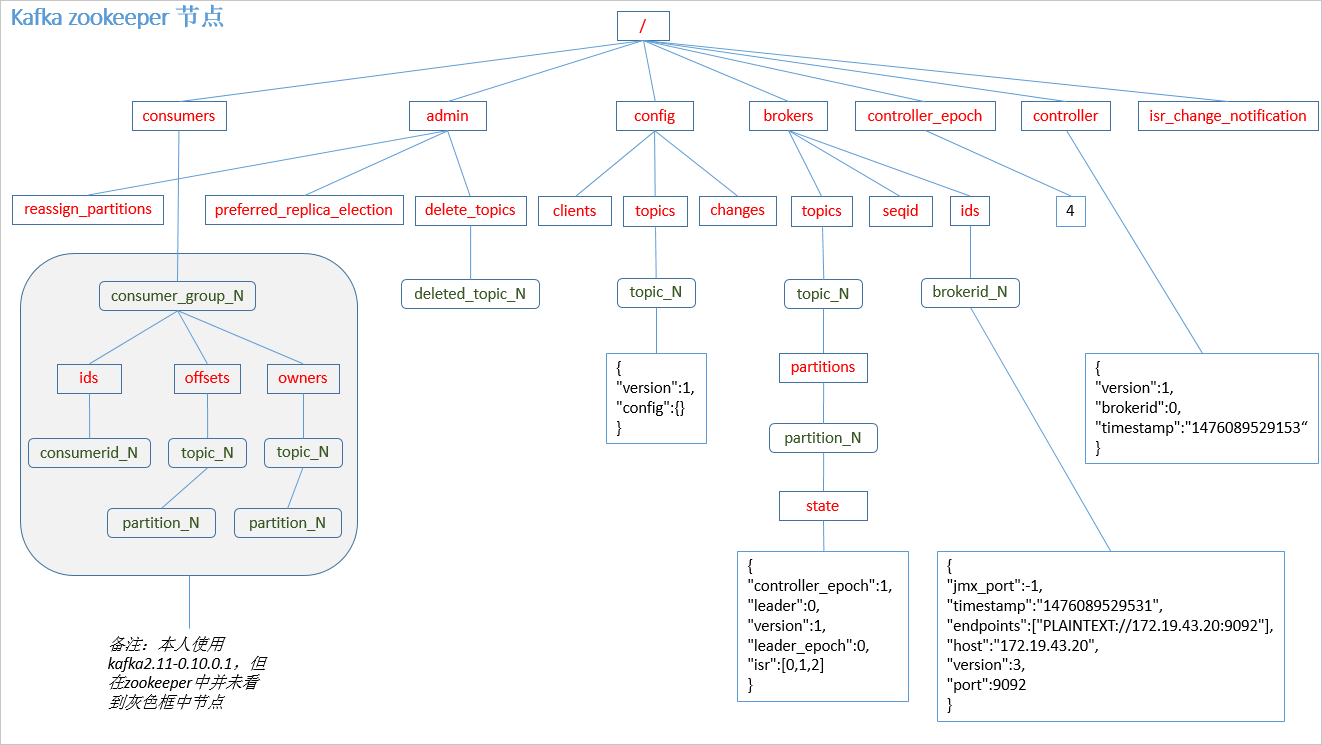
**9.follower：**replica 中的一个角色，从leader 中复制数据。

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

**12.zookeeper：**kafka 通过 zookeeper 来存储集群的 meta 信息。

## zookeeper 节点

kafka 在 zookeeper 中的存储结构如下图所示：



# Kafka安装和启动

kafka的背景知识已经讲了很多了，让我们现在开始实践吧，假设你现在没有Kafka和ZooKeeper环境。

## Step 1: 下载代码

下载0.10.0.0版本并且解压它。

> tar -xzf kafka\_2.11-0.10.0.0.tgz

> cd kafka\_2.11-0.10.0.0

## Step 2: 启动服务

运行kafka需要使用Zookeeper，所以你需要先启动Zookeeper，如果你没有Zookeeper，你可以使用kafka自带打包和配置好的Zookeeper。

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

[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

...

现在启动kafka服务：

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

[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)

[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)

...

## Step 3: 创建一个主题(topic)

创建一个名为“test”的Topic，只有一个分区和一个备份：

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

创建好之后，可以通过运行以下命令，查看已创建的topic信息：

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

test

或者，除了手工创建topic外，你也可以配置你的broker，当发布一个不存在的topic时自动创建topic。

## Step 4: 发送消息

Kafka提供了一个命令行的工具，可以从输入文件或者命令行中读取消息并发送给Kafka集群，每一行是一条消息。

运行producer（生产者）,然后在控制台输入几条消息到服务器。

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

This is a message

This is another message

## Step 5: 消费消息

Kafka也提供了一个消费消息的命令行工具，将存储的信息输出出来。

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

This is a message

This is another message

如果你有2台不同的终端上运行上述命令，那么当你在运行生产者时，消费者就能消费到生产者发送的消息。

所有的命令行工具有很多的选项，你可以查看文档来了解更多的功能。

## Step 6: 设置多个broker集群

到目前，我们只是单一的运行一个broker,，没什么意思。对于Kafka,一个broker仅仅只是一个集群的大小, 所有让我们多设几个broker。

首先为每个broker创建一个配置文件:

> cp config/server.properties config/server-1.properties

> cp config/server.properties config/server-2.properties

现在编辑这些新建的文件，设置以下属性：

config/server-1.properties:

broker.id=1

listeners=PLAINTEXT://:9093

log.dir=/tmp/kafka-logs-1

config/server-2.properties:

broker.id=2

listeners=PLAINTEXT://:9094

log.dir=/tmp/kafka-logs-2

broker.id是集群中每个节点的唯一且永久的名称，我们修改端口和日志分区是因为我们现在在同一台机器上运行，我们要防止broker在同一端口上注册和覆盖对方的数据。

我们已经运行了zookeeper和刚才的一个kafka节点，所有我们只需要在启动2个新的kafka节点。

> bin/kafka-server-start.sh config/server-1.properties &

...

> bin/kafka-server-start.sh config/server-2.properties &

...

现在，我们创建一个新topic，把备份设置为：3

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

好了，现在我们已经有了一个集群了，我们怎么知道每个集群在做什么呢？运行命令“describe topics”。

> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic

Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:

Topic: my-replicated-topic Partition: 0 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0

这是一个解释输出，第一行是所有分区的摘要，每一个线提供一个分区信息，因为我们只有一个分区，所有只有一条线。

* "leader"：该节点负责所有指定分区的读和写，每个节点的领导都是随机选择的。
* "replicas":备份的节点，无论该节点是否是leader或者目前是否还活着，只是显示。
* "isr"：备份节点的集合，也就是活着的节点集合。

我们运行这个命令，看看一开始我们创建的那个节点：

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

Topic:test PartitionCount:1 ReplicationFactor:1 Configs:

Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0

没有惊喜，刚才创建的topic（主题）没有Replicas，所以是0。

让我们来发布一些信息在新的topic上：

> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic

...

my test message 1

my test message 2

^C

现在，消费这些消息。

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic

...

my test message 1

my test message 2

^C

我们要测试集群的容错，kill掉leader，Broker1作为当前的leader，也就是kill掉Broker1。

> ps | grep server-1.properties

7564 ttys002 0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bin/java...

> kill -9 7564

备份节点之一成为新的leader，而broker1已经不在同步备份集合里了。

> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic

Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:

Topic: my-replicated-topic Partition: 0 Leader: 2 Replicas: 1,2,0 Isr: 2,0

但是，消息仍然没丢：

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic

...

my test message 1

my test message 2

^C

## Step 7: 使用 Kafka Connect导入/导出数据

从控制台写入和写回数据是一个方便的开始，但你可能想要从其他来源导入或导出数据到其他系统。对于大多数系统，可以使用kafka Connect，而不需要编写自定义集成代码。Kafka Connect是导入和导出数据的一个工具。它是一个可扩展的工具，运行连接器，实现与自定义的逻辑的外部系统交互。我们将看到如何运行Kafka Connect用简单的连接器从文件导入数据到Kafka主题，再从Kafka主题导出数据到文件，首先，我们首先创建一些种子数据用来测试：

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

接下来，我们开始2个连接器运行在独立的模式，这意味着它们运行在一个单一的，本地的，专用的进程。我们提供3个配置文件作为参数。第一个始终是kafka Connect进程，如kafka broker连接和数据库序列化格式，剩下的配置文件每个指定的连接器来创建，这些文件包括一个独特的连接器名称，连接器类来实例化和任何其他配置要求的。

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

这是示例的配置文件，使用默认的本地集群配置并创建了2个连接器：第一个是导入连接器，从导入文件中读取并发布到Kafka主题，第二个是导出连接器，从kafka主题读取消息输出到外部文件，在启动过程中，你会看到一些日志消息，包括一些连接器实例化的说明。

一旦kafka Connect进程已经开始，导入连接器应该从：test.txt读取；写入到topic：connect-test；从主题导出连接器：connect-test；读取消息写入到文件：test.sink.txt。

我们可以通过验证输出文件的内容来验证数据数据已经全部导出：

cat test.sink.txt

foo

bar

注意，导入的数据也已经在Kafka主题：connect-test里,所以我们可以使用该命令查看这个主题：

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

{"schema":{"type":"string","optional":false},"payload":"foo"}

{"schema":{"type":"string","optional":false},"payload":"bar"}

...

连接器继续处理数据，因此我们可以添加数据到文件并通过管道移动：

echo "Another line" >> test.txt

你应该会看到出现在消费者控台输出一行信息并导出到文件。

## Step 8: 使用Kafka Stream来处理数据

Kafka Stream是kafka的客户端库，用于实时流处理和分析存储在kafka broker的数据。一个WordCountDemo的例子（为了方便阅读，使用的是java8 lambda表达式）。

KTable wordCounts = textLines

// Split each text line, by whitespace, into words.

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("W+")))

// Ensure the words are available as record keys for the next aggregate operation.

.map((key, value) -> new KeyValue<>(value, value))

// Count the occurrences of each word (record key) and store the results into a table named "Counts".

.countByKey("Counts")

它实现了wordcount算法，从输入的文本计算出一个词出现的次数。然而，不像其他的WordCount的例子，你可能会看到，在有限的数据之前，执行的演示应用程序的行为略有不同，因为它的目的是在一个无限的操作，数据流。类似的有界变量，它是一种动态算法，跟踪和更新的单词计数。然而，由于它必须假设潜在的无界输入数据，它会定期输出其当前状态和结果，同时继续处理更多的数据，因为它不知道什么时候它处理过的“所有”的输入数据。

现在准备输入数据到kafka的topic中，随后kafka Stream应用处理这个topic的数据。

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

接下来，使用控制台的producer 将输入的数据发送到指定的topic（streams-file-input）中（在实践中，stream数据可能会持续流入，其中kafka的应用将启动并运行）。

> bin/kafka-topics.sh --create \

--zookeeper localhost:2181 \

--replication-factor 1 \

--partitions 1 \

--topic streams-file-input

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

现在，我们运行 WordCount 处理输入的数据：

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

不会有任何的STDOUT输出，除了日志，结果不断地写回另一个topic（streams-wordcount-output），demo运行几秒，然后，不像典型的流处理应用程序，自动终止。

现在我们检查WordCountDemo应用，从输出的topic读取。

> ./bin/kafka-console-consumer --zookeeper localhost:2181

--topic streams-wordcount-output

--from-beginning

--formatter kafka.tools.DefaultMessageFormatter

--property print.key=true

--property print.key=true

--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer

--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

输出数据打印到控台（你可以使用Ctrl-C停止）：

all 1

streams 1

lead 1

to 1

kafka 1

hello 1

kafka 2

streams 2

join 1

kafka 3

summit 1

^C

第一列是message的key，第二列是message的value，要注意，输出的实际是一个连续的更新流，其中每条数据（即：原始输出的每行）是一个单词的最新的count，又叫记录键“kafka”。对于同一个key有多个记录，每个记录之后是前一个的更新。

# producer发布消息

## 写入方式

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。

## 消息路由

producer 发送消息到 broker 时，会根据分区算法选择将其存储到哪一个 partition。其路由机制为：

* 1. 指定了 patition，则直接使用；
* 2. 未指定 patition 但指定 key，通过对 key 的 value 进行hash 选出一个 patition；
* 3. patition 和 key 都未指定，使用轮询选出一个 patition。

 附上 java 客户端分区源码，一目了然：

//创建消息实例

**public** ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) {

**if** (topic == **null**)

**throw** **new** IllegalArgumentException("Topic cannot be null");

**if** (timestamp != **null** && timestamp < 0)

**throw** **new** IllegalArgumentException("Invalid timestamp " + timestamp);

**this**.topic = topic;

**this**.partition = partition;

**this**.key = key;

**this**.value = value;

**this**.timestamp = timestamp;

}

//计算 patition，如果指定了 patition 则直接使用，否则使用 key 计算

**private** **int** partition(ProducerRecord<K, V> record, **byte**[] serializedKey , **byte**[] serializedValue, Cluster cluster) {

Integer partition = record.partition();

**if** (partition != **null**) {

List<PartitionInfo> partitions = cluster.partitionsForTopic(record.topic());

**int** lastPartition = partitions.size() - 1;

**if** (partition < 0 || partition > lastPartition) {

**throw** **new** IllegalArgumentException(String.format("Invalid partition given with record: %d is not in the range [0...%d].", partition, lastPartition));

}

**return** partition;

}

**return** **this**.partitioner.partition(record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster);

}

//使用 key 选取 patition

**public** **int** partition(String topic, Object key, **byte**[] keyBytes, Object value, **byte**[] valueBytes, Cluster cluster) {

List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);

**int** numPartitions = partitions.size();

**if** (keyBytes == **null**) {

**int** nextValue = counter.getAndIncrement();

List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);

**if** (availablePartitions.size() > 0) {

**int** part = DefaultPartitioner.toPositive(nextValue) % availablePartitions.size();

**return** availablePartitions.get(part).partition();

} **else** {

**return** DefaultPartitioner.toPositive(nextValue) % numPartitions;

}

} **else** {

//对 keyBytes 进行 hash 选出一个 patition

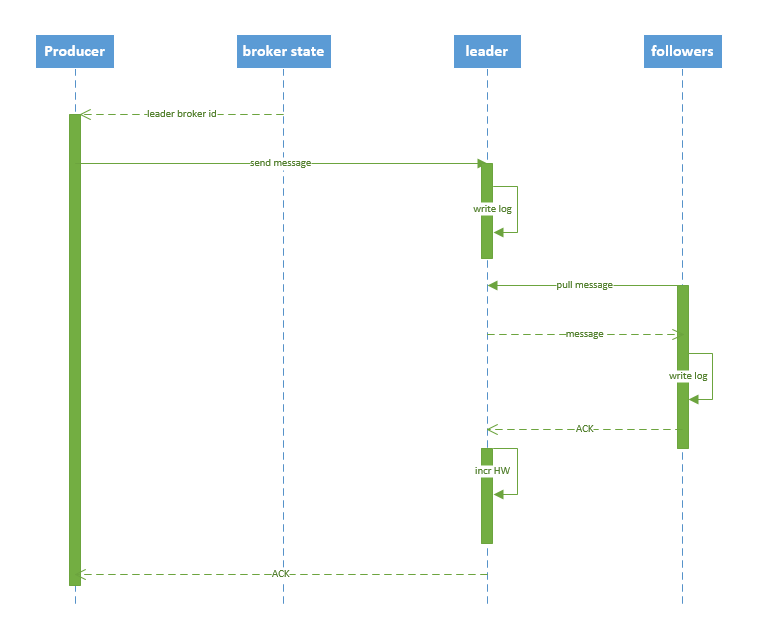
**return** DefaultPartitioner.toPositive(Utils.murmur2(keyBytes)) % numPartitions;

}

}

## 写入流程

producer 写入消息序列图如下所示：



流程说明：

* 1. producer 先从 zookeeper 的 "/brokers/.../state" 节点找到该 partition 的 leader。
* 2. producer 将消息发送给该 leader。
* 3. leader 将消息写入本地 log。
* 4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK。
* 5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW（high watermark，最后 commit 的 offset） 并向 producer 发送 ACK。

## producer delivery guarantee

 一般情况下存在三种情况：

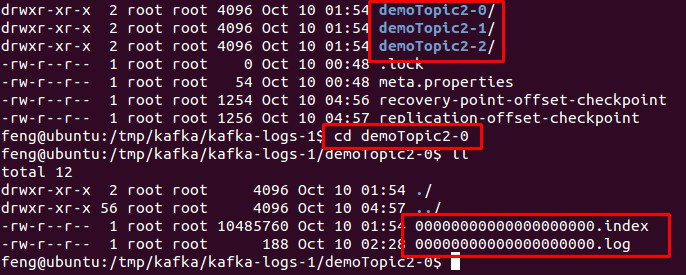
* 1. At most once 消息可能会丢，但绝不会重复传输.
* 2. At least one 消息绝不会丢，但可能会重复传输。
* 3. Exactly once 每条消息肯定会被传输一次且仅传输一次。

当 producer 向 broker 发送消息时，一旦这条消息被 commit，由于 replication 的存在，它就不会丢。但是如果 producer 发送数据给 broker 后，遇到网络问题而造成通信中断，那 Producer 就无法判断该条消息是否已经 commit。虽然 Kafka 无法确定网络故障期间发生了什么，但是 producer 可以生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了 Exactly once，但目前还并未实现。所以目前默认情况下一条消息从 producer 到 broker 是确保了 At least once，可通过设置 producer 异步发送实现At most once。

# broker保存消息

## 存储方式

物理上把 topic 分成一个或多个 patition（对应 server.properties 中的 num.partitions=3 配置），每个 patition 物理上对应一个文件夹（该文件夹存储该 patition 的所有消息和索引文件），如下：



## 存储策略

无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

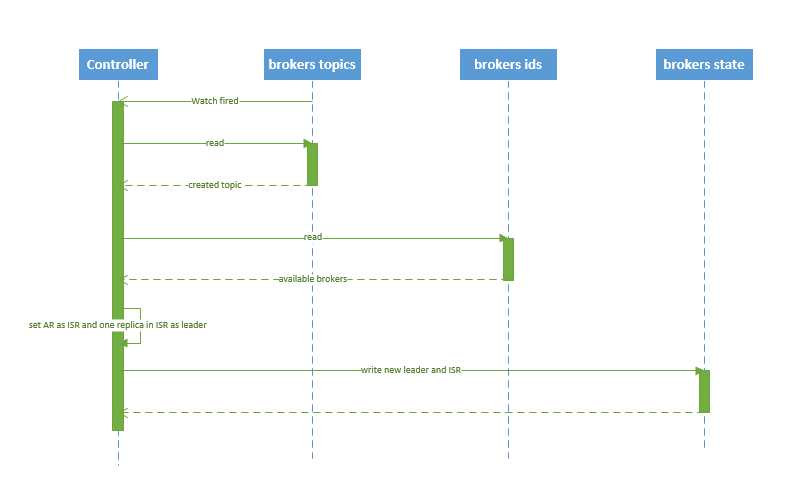
* 1. 基于时间：log.retention.hours=168
* 2. 基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。

## topic 创建与删除

### 创建 topic

创建 topic 的序列图如下所示：



流程说明：

* 1. controller 在 ZooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被创建，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。
* 2. controller从 /brokers/ids 读取当前所有可用的 broker 列表，对于 set\_p 中的每一个 partition：

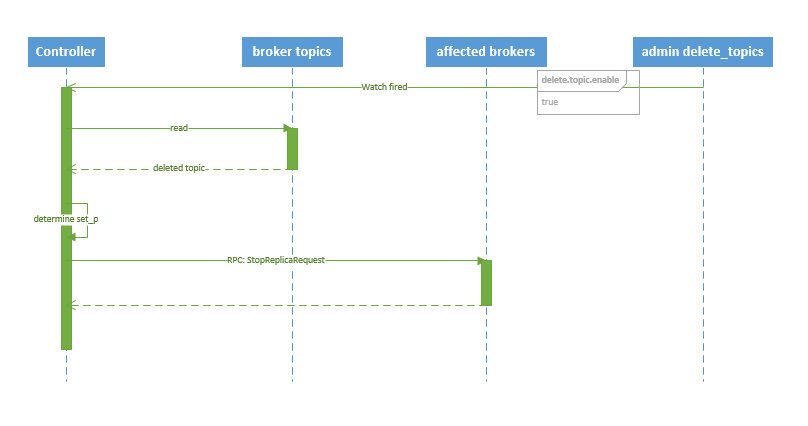
2.1 从分配给该 partition 的所有 replica（称为AR）中任选一个可用的 broker 作为新的 leader，并将AR设置为新的 ISR

2.2 将新的 leader 和 ISR 写入 /brokers/topics/[topic]/partitions/[partition]/state

* 3. controller 通过 RPC 向相关的 broker 发送 LeaderAndISRRequest。

### 删除 topic

删除 topic 的序列图如下所示：



流程说明：

* 1. controller 在 zooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被删除，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。
* 2. 若 delete.topic.enable=false，结束；否则 controller 注册在 /admin/delete\_topics 上的 watch 被 fire，controller 通过回调向对应的 broker 发送 StopReplicaRequest。

# Kafka High Availability

Kafka在0.8以前的版本中，并不提供High Availablity机制，一旦一个或多个Broker宕机，则宕机期间其上所有Partition都无法继续提供服务。若该Broker永远不能再恢复，亦或磁盘故障，则其上数据将丢失。而Kafka的设计目标之一即是提供数据持久化，同时对于分布式系统来说，尤其当集群规模上升到一定程度后，一台或者多台机器宕机的可能性大大提高，对Failover要求非常高。因此，Kafka从0.8开始提供High Availability机制。本文从Data Replication和Leader Election两方面介绍了Kafka的HA机制。

## Kafka为何需要High Available

### 为何需要Replication

在Kafka在0.8以前的版本中，是没有Replication的，一旦某一个Broker宕机，则其上所有的Partition数据都不可被消费，这与Kafka数据持久性及Delivery Guarantee的设计目标相悖。同时Producer都不能再将数据存于这些Partition中。

* 如果Producer使用同步模式则Producer会在尝试重新发送message.send.max.retries（默认值为3）次后抛出Exception，用户可以选择停止发送后续数据也可选择继续选择发送。而前者会造成数据的阻塞，后者会造成本应发往该Broker的数据的丢失。
* 如果Producer使用异步模式，则Producer会尝试重新发送message.send.max.retries（默认值为3）次后记录该异常并继续发送后续数据，这会造成数据丢失并且用户只能通过日志发现该问题。同时，Kafka的Producer并未对异步模式提供callback接口。

由此可见，在没有Replication的情况下，一旦某机器宕机或者某个Broker停止工作则会造成整个系统的可用性降低。随着集群规模的增加，整个集群中出现该类异常的几率大大增加，因此对于生产系统而言Replication机制的引入非常重要。

### 为何需要Leader Election

引入Replication之后，同一个Partition可能会有多个Replica，而这时需要在这些Replication之间选出一个Leader，Producer和Consumer只与这个Leader交互，其它Replica作为Follower从Leader中复制数据。

因为需要保证同一个Partition的多个Replica之间的数据一致性（其中一个宕机后其它Replica必须要能继续服务并且即不能造成数据重复也不能造成数据丢失）。如果没有一个Leader，所有Replica都可同时读/写数据，那就需要保证多个Replica之间互相（N×N条通路）同步数据，数据的一致性和有序性非常难保证，大大增加了Replication实现的复杂性，同时也增加了出现异常的几率。而引入Leader后，只有Leader负责数据读写，Follower只向Leader顺序Fetch数据（N条通路），系统更加简单且高效。

## Kafka HA设计解析

### 如何将所有Replica均匀分布到整个集群

为了更好的做负载均衡，Kafka尽量将所有的Partition均匀分配到整个集群上。一个典型的部署方式是一个Topic的Partition数量大于Broker的数量。同时为了提高Kafka的容错能力，也需要将同一个Partition的Replica尽量分散到不同的机器。实际上，如果所有的Replica都在同一个Broker上，那一旦该Broker宕机，该Partition的所有Replica都无法工作，也就达不到HA的效果。同时，如果某个Broker宕机了，需要保证它上面的负载可以被均匀的分配到其它幸存的所有Broker上。

Kafka分配Replica的算法如下：

* 将所有Broker（假设共n个Broker）和待分配的Partition排序。
* 将第i个Partition分配到第（i mod n）个Broker上。
* 将第i个Partition的第j个Replica分配到第（(i + j) mode n）个Broker上。

### Data Replication

Kafka的Data Replication需要解决如下问题：

* 怎样Propagate消息。
* 在向Producer发送ACK前需要保证有多少个Replica已经收到该消息。
* 怎样处理某个Replica不工作的情况。
* 怎样处理Failed Replica恢复回来的情况。

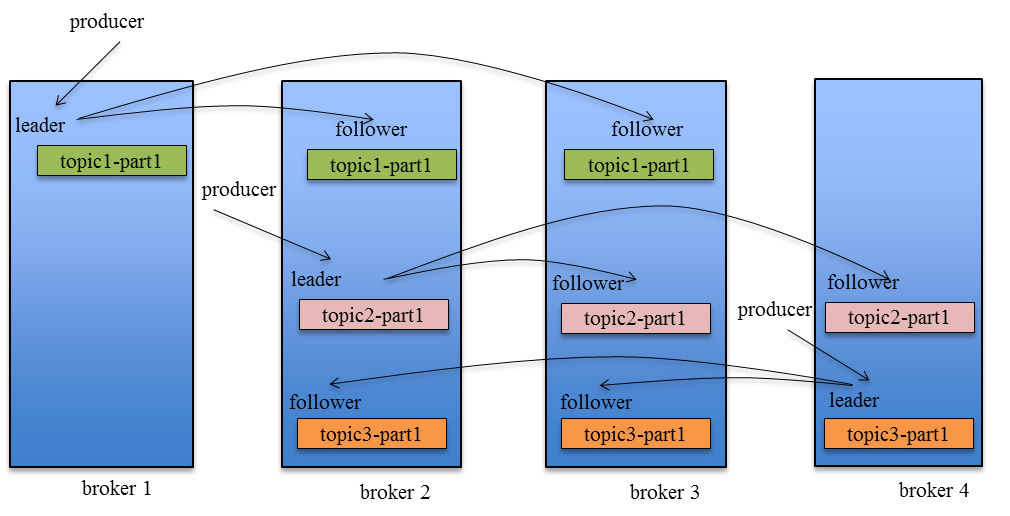
#### Propagate消息

Producer在发布消息到某个Partition时，先通过ZooKeeper找到该Partition的Leader，然后无论该Topic的Replication Factor为多少（也即该Partition有多少个Replica），Producer只将该消息发送到该Partition的Leader。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。Follower在收到该消息并写入其Log后，向Leader发送ACK。一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经commit了，Leader将增加HW并且向Producer发送ACK。

为了提高性能，每个Follower在接收到数据后就立马向Leader发送ACK，而非等到数据写入Log中。因此，对于已经commit的消息，Kafka只能保证它被存于多个Replica的内存中，而不能保证它们被持久化到磁盘中，也就不能完全保证异常发生后该条消息一定能被Consumer消费。但考虑到这种场景非常少见，可以认为这种方式在性能和数据持久化上做了一个比较好的平衡。在将来的版本中，Kafka会考虑提供更高的持久性。

Consumer读消息也是从Leader读取，只有被commit过的消息（offset低于HW的消息）才会暴露给Consumer。

Kafka Replication的数据流如下图所示：



#### ACK前需要保证有多少个备份

和大部分分布式系统一样，Kafka处理失败需要明确定义一个Broker是否“活着”。对于Kafka而言，Kafka存活包含两个条件，一是它必须维护与ZooKeeper的session（这个通过ZooKeeper的Heartbeat机制来实现）。二是Follower必须能够及时将Leader的消息复制过来，不能“落后太多”。

Leader会跟踪与其保持同步的Replica列表，该列表称为ISR（即in-sync Replica）。如果一个Follower宕机，或者落后太多，Leader将把它从ISR中移除。这里所描述的“落后太多”指Follower复制的消息落后于Leader后的条数超过预定值（该值可在$KAFKA\_HOME/config/server.properties中通过replica.lag.max.messages配置，其默认值是4000）或者Follower超过一定时间（该值可在$KAFKA\_HOME/config/server.properties中通过replica.lag.time.max.ms来配置，其默认值是10000）未向Leader发送fetch请求。

Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，完全同步复制要求所有能工作的Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率（高吞吐率是Kafka非常重要的一个特性）。而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下如果Follower都复制完都落后于Leader，而如果Leader突然宕机，则会丢失数据。而Kafka的这种使用ISR的方式则很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，这样极大的提高复制性能（批量写磁盘），极大减少了Follower与Leader的差距。

需要说明的是，Kafka只解决fail/recover，不处理“Byzantine”（“拜占庭”）问题。一条消息只有被ISR里的所有Follower都从Leader复制过去才会被认为已提交。这样就避免了部分数据被写进了Leader，还没来得及被任何Follower复制就宕机了，而造成数据丢失（Consumer无法消费这些数据）。而对于Producer而言，它可以选择是否等待消息commit，这可以通过request.required.acks来设置。这种机制确保了只要ISR有一个或以上的Follower，一条被commit的消息就不会丢失。

#### Leader Election算法

上文说明了Kafka是如何做Replication的，另外一个很重要的问题是当Leader宕机了，怎样在Follower中选举出新的Leader。因为Follower可能落后许多或者crash了，所以必须确保选择“最新”的Follower作为新的Leader。一个基本的原则就是，如果Leader不在了，新的Leader必须拥有原来的Leader commit过的所有消息。这就需要作一个折衷，如果Leader在标明一条消息被commit前等待更多的Follower确认，那在它宕机之后就有更多的Follower可以作为新的Leader，但这也会造成吞吐率的下降。

一种非常常用的选举leader的方式是“Majority Vote”（“少数服从多数”），但Kafka并未采用这种方式。这种模式下，如果我们有2f+1个Replica（包含Leader和Follower），那在commit之前必须保证有f+1个Replica复制完消息，为了保证正确选出新的Leader，fail的Replica不能超过f个。因为在剩下的任意f+1个Replica里，至少有一个Replica包含有最新的所有消息。这种方式有个很大的优势，系统的latency只取决于最快的几个Broker，而非最慢那个。Majority Vote也有一些劣势，为了保证Leader Election的正常进行，它所能容忍的fail的follower个数比较少。如果要容忍1个follower挂掉，必须要有3个以上的Replica，如果要容忍2个Follower挂掉，必须要有5个以上的Replica。也就是说，在生产环境下为了保证较高的容错程度，必须要有大量的Replica，而大量的Replica又会在大数据量下导致性能的急剧下降。这就是这种算法更多用在[ZooKeeper](http://zookeeper.apache.org/)这种共享集群配置的系统中而很少在需要存储大量数据的系统中使用的原因。例如HDFS的HA Feature是基于[majority-vote-based journal](http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4-1)，但是它的数据存储并没有使用这种方式。

实际上，Leader Election算法非常多，比如ZooKeeper的[Zab](http://web.stanford.edu/class/cs347/reading/zab.pdf), [Raft](https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf)和[Viewstamped Replication](http://pmg.csail.mit.edu/papers/vr-revisited.pdf)。而Kafka所使用的Leader Election算法更像微软的[PacificA](http://research.microsoft.com/apps/pubs/default.aspx?id=66814)算法。

Kafka在ZooKeeper中动态维护了一个ISR（in-sync replicas），这个ISR里的所有Replica都跟上了leader，只有ISR里的成员才有被选为Leader的可能。在这种模式下，对于f+1个Replica，一个Partition能在保证不丢失已经commit的消息的前提下容忍f个Replica的失败。在大多数使用场景中，这种模式是非常有利的。事实上，为了容忍f个Replica的失败，Majority Vote和ISR在commit前需要等待的Replica数量是一样的，但是ISR需要的总的Replica的个数几乎是Majority Vote的一半。

虽然Majority Vote与ISR相比有不需等待最慢的Broker这一优势，但是Kafka作者认为Kafka可以通过Producer选择是否被commit阻塞来改善这一问题，并且节省下来的Replica和磁盘使得ISR模式仍然值得。

#### 如何处理所有Replica都不工作

上文提到，在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某个Partition的所有Replica都宕机了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

* 等待ISR中的任一个Replica“活”过来，并且选它作为Leader。
* 选择第一个“活”过来的Replica（不一定是ISR中的）作为Leader。

这就需要在可用性和一致性当中作出一个简单的折衷。如果一定要等待ISR中的Replica“活”过来，那不可用的时间就可能会相对较长。而且如果ISR中的所有Replica都无法“活”过来了，或者数据都丢失了，这个Partition将永远不可用。选择第一个“活”过来的Replica作为Leader，而这个Replica不是ISR中的Replica，那即使它并不保证已经包含了所有已commit的消息，它也会成为Leader而作为consumer的数据源（前文有说明，所有读写都由Leader完成）。Kafka0.8.\*使用了第二种方式。根据Kafka的文档，在以后的版本中，Kafka支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性。

#### 如何选举Leader

最简单最直观的方案是，所有Follower都在ZooKeeper上设置一个Watch，一旦Leader宕机，其对应的ephemeral znode会自动删除，此时所有Follower都尝试创建该节点，而创建成功者（ZooKeeper保证只有一个能创建成功）即是新的Leader，其它Replica即为Follower。

但是该方法会有3个问题：

* split-brain 这是由ZooKeeper的特性引起的，虽然ZooKeeper能保证所有Watch按顺序触发，但并不能保证同一时刻所有Replica“看”到的状态是一样的，这就可能造成不同Replica的响应不一致。
* herd effect 如果宕机的那个Broker上的Partition比较多，会造成多个Watch被触发，造成集群内大量的调整。
* ZooKeeper负载过重 每个Replica都要为此在ZooKeeper上注册一个Watch，当集群规模增加到几千个Partition时ZooKeeper负载会过重。

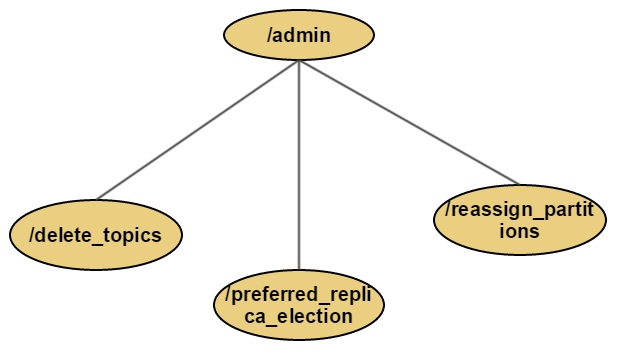
Kafka 0.8.\*的Leader Election方案解决了上述问题，它在所有broker中选出一个controller，所有Partition的Leader选举都由controller决定。controller会将Leader的改变直接通过RPC的方式（比ZooKeeper Queue的方式更高效）通知需为为此作为响应的Broker。同时controller也负责增删Topic以及Replica的重新分配。

### HA相关ZooKeeper结构

首先声明本节所示ZooKeeper结构中，实线框代表路径名是固定的，而虚线框代表路径名与业务相关。

#### admin

该目录下znode只有在有相关操作时才会存在，操作结束时会将其删除。



/admin/preferred\_replica\_election数据结构

{

"fields":[

{

"name":"version",

"type":"int",

"doc":"version id"

},

{

"name":"partitions",

"type":{

"type":"array",

"items":{

"fields":[

{

"name":"topic",

"type":"string",

"doc":"topic of the partition for which preferred replica election should be triggered"

},

{

"name":"partition",

"type":"int",

"doc":"the partition for which preferred replica election should be triggered"

}

],

}

"doc":"an array of partitions for which preferred replica election should be triggered"

}

}

]

}

Example:

{

"version": 1,

"partitions":

[

{

"topic": "topic1",

"partition": 8

},

{

"topic": "topic2",

"partition": 16

}

]

}

/admin/reassign\_partitions用于将一些Partition分配到不同的broker集合上。对于每个待重新分配的Partition，Kafka会在该znode上存储其所有的Replica和相应的Broker id。该znode由管理进程创建并且一旦重新分配成功它将会被自动移除。其数据结构如下：

{

"fields":[

{

"name":"version",

"type":"int",

"doc":"version id"

},

{

"name":"partitions",

"type":{

"type":"array",

"items":{

"fields":[

{

"name":"topic",

"type":"string",

"doc":"topic of the partition to be reassigned"

},

{

"name":"partition",

"type":"int",

"doc":"the partition to be reassigned"

},

{

"name":"replicas",

"type":"array",

"items":"int",

"doc":"a list of replica ids"

}

],

}

"doc":"an array of partitions to be reassigned to new replicas"

}

}

]

}

Example:

{

"version": 1,

"partitions":

[

{

"topic": "topic3",

"partition": 1,

"replicas": [1, 2, 3]

}

]

}

/admin/delete\_topics数据结构：

Schema:

{ "fields":

[ {"name": "version", "type": "int", "doc": "version id"},

{"name": "topics",

"type": { "type": "array", "items": "string", "doc": "an array of topics to be deleted"}

} ]

}

Example:

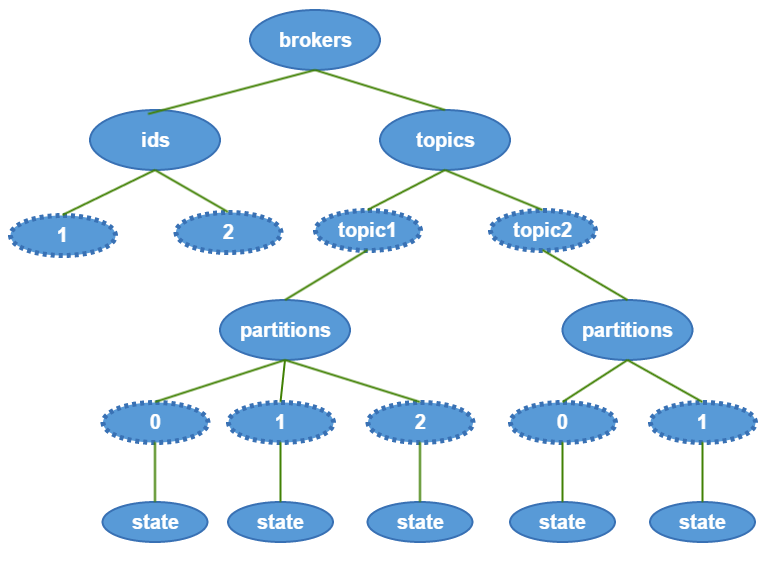
{

"version": 1,

"topics": ["topic4", "topic5"]

}

#### brokers



broker（即/brokers/ids/[brokerId]）存储“活着”的broker信息。数据结构如下：

Schema:

{ "fields":

[ {"name": "version", "type": "int", "doc": "version id"},

{"name": "host", "type": "string", "doc": "ip address or host name of the broker"},

{"name": "port", "type": "int", "doc": "port of the broker"},

{"name": "jmx\_port", "type": "int", "doc": "port for jmx"}

]

}

Example:

{

"jmx\_port":-1,

"host":"node1",

"version":1,

"port":9092

}

topic注册信息（/brokers/topics/[topic]），存储该topic的所有partition的所有replica所在的broker id，第一个replica即为preferred replica，对一个给定的partition，它在同一个broker上最多只有一个replica,因此broker id可作为replica id。数据结构如下：

Schema:

{ "fields" :

[ {"name": "version", "type": "int", "doc": "version id"},

{"name": "partitions",

"type": {"type": "map",

"values": {"type": "array", "items": "int", "doc": "a list of replica ids"},

"doc": "a map from partition id to replica list"},

}

]

}

Example:

{

"version":1,

"partitions":

{"12":[6],

"8":[2],

"4":[6],

"11":[5],

"9":[3],

"5":[7],

"10":[4],

"6":[8],

"1":[3],

"0":[2],

"2":[4],

"7":[1],

"3":[5]}

}

partition state（/brokers/topics/[topic]/partitions/[partitionId]/state） 结构如下：

Schema:

{ "fields":

[ {"name": "version", "type": "int", "doc": "version id"},

{"name": "isr",

"type": {"type": "array",

"items": "int",

"doc": "an array of the id of replicas in isr"}

},

{"name": "leader", "type": "int", "doc": "id of the leader replica"},

{"name": "controller\_epoch", "type": "int", "doc": "epoch of the controller that last updated the leader and isr info"},

{"name": "leader\_epoch", "type": "int", "doc": "epoch of the leader"}

]

}

Example:

{

"controller\_epoch":29,

"leader":2,

"version":1,

"leader\_epoch":48,

"isr":[2]

}

#### controller

/controller -> int (broker id of the controller)存储当前controller的信息

Schema:

{ "fields":

[ {"name": "version", "type": "int", "doc": "version id"},

{"name": "brokerid", "type": "int", "doc": "broker id of the controller"}

]

}

Example:

{

"version":1,

　　"brokerid":8

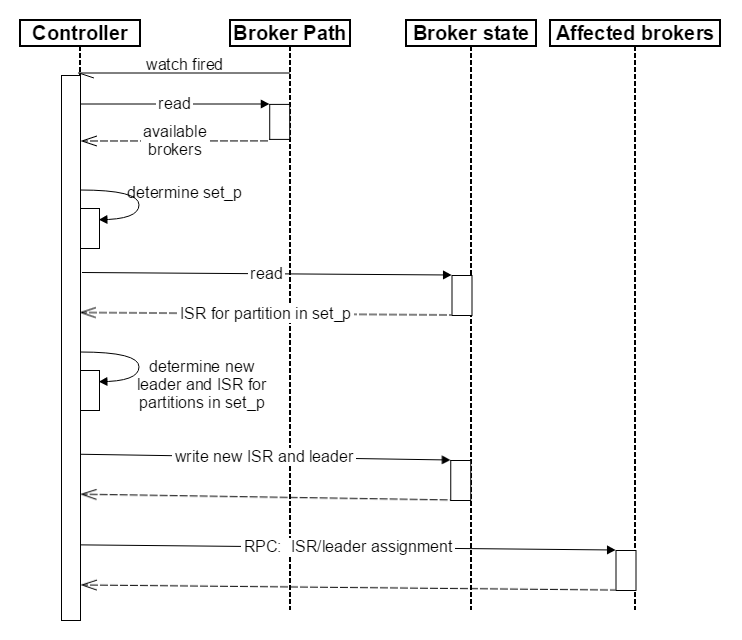
}

/controller\_epoch -> int (epoch)直接以整数形式存储controller epoch，而非像其它znode一样以JSON字符串形式存储。

### broker failover过程简介

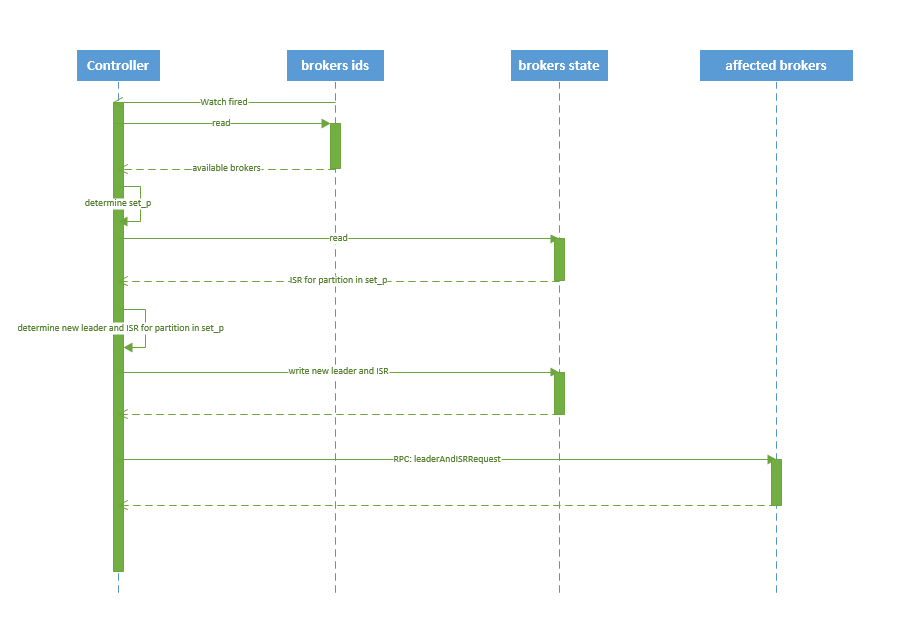
* Controller在ZooKeeper注册Watch，一旦有Broker宕机（这是用宕机代表任何让系统认为其die的情景，包括但不限于机器断电，网络不可用，GC导致的Stop The World，进程crash等），其在ZooKeeper对应的znode会自动被删除，ZooKeeper会fire Controller注册的watch，Controller读取最新的幸存的Broker。
* Controller决定set\_p，该集合包含了宕机的所有Broker上的所有Partition。
* 对set\_p中的每一个Partition：
  + 3.1 从/brokers/topics/[topic]/partitions/[partition]/state读取该Partition当前的ISR
  + 3.2 决定该Partition的新Leader。如果当前ISR中有至少一个Replica还幸存，则选择其中一个作为新Leader，新的ISR则包含当前ISR中所有幸存的Replica。否则选择该Partition中任意一个幸存的Replica作为新的Leader以及ISR（该场景下可能会有潜在的数据丢失）。如果该Partition的所有Replica都宕机了，则将新的Leader设置为-1。
  + 3.3 将新的Leader，ISR和新的leader\_epoch及controller\_epoch写入/brokers/topics/[topic]/partitions/[partition]/state。注意，该操作只有其version在3.1至3.3的过程中无变化时才会执行，否则跳转到3.1
* 直接通过RPC向set\_p相关的Broker发送LeaderAndISRRequest命令。Controller可以在一个RPC操作中发送多个命令从而提高效率。

broker failover顺序图如下所示。



## Broker Failover过程

kafka broker failover 序列图如下所示：



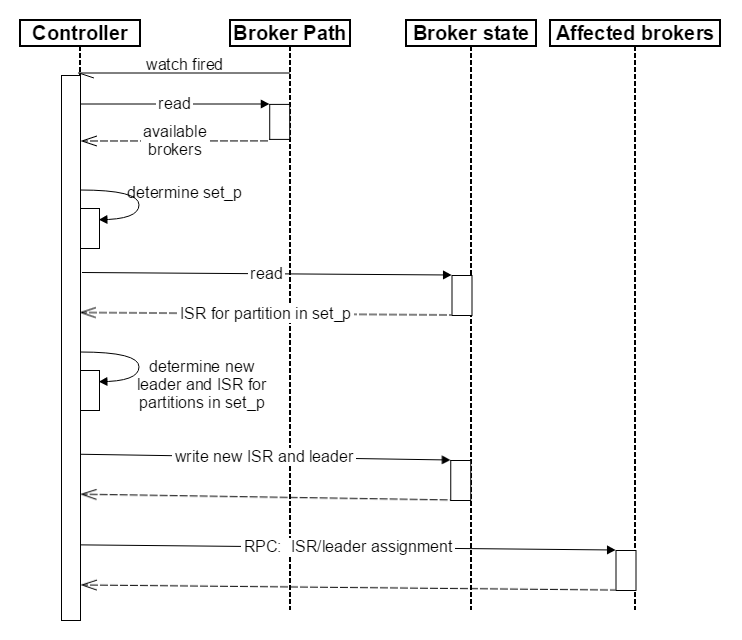
流程说明：

* 1. controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker 宕机时 zookeeper 会 fire watch。
* 2. controller 从 /brokers/ids 节点读取可用broker。
* 3. controller决定set\_p，该集合包含宕机 broker 上的所有 partition。
* 4. 对 set\_p 中的每一个 partition：
  + 4.1 从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR。
  + 4.2 决定新 leader。
  + 4.3 将新 leader、ISR、controller\_epoch 和 leader\_epoch 等信息写入 state 节点。
* 5. 通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令。

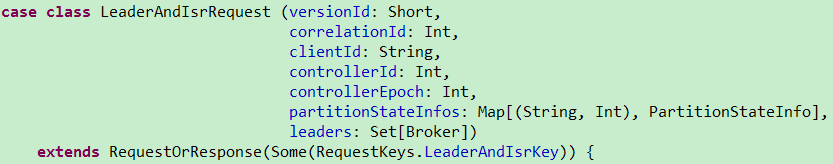
### Controller对Broker failure的处理过程

* Controller在ZooKeeper的/brokers/ids节点上注册Watch。一旦有Broker宕机（本文用宕机代表任何让Kafka认为其Broker die的情景，包括但不限于机器断电，网络不可用，GC导致的Stop The World，进程crash等），其在ZooKeeper对应的Znode会自动被删除，ZooKeeper会fire Controller注册的Watch，Controller即可获取最新的幸存的Broker列表。
* Controller决定set\_p，该集合包含了宕机的所有Broker上的所有Partition。
* 对set\_p中的每一个Partition：
  + 3.1 从/brokers/topics/[topic]/partitions/[partition]/state读取该Partition当前的ISR。
  + 3.2 决定该Partition的新Leader。如果当前ISR中有至少一个Replica还幸存，则选择其中一个作为新Leader，新的ISR则包含当前ISR中所有幸存的Replica。否则选择该Partition中任意一个幸存的Replica作为新的Leader以及ISR（该场景下可能会有潜在的数据丢失）。如果该Partition的所有Replica都宕机了，则将新的Leader设置为-1。
  + 3.3 将新的Leader，ISR和新的leader\_epoch及controller\_epoch写入/brokers/topics/[topic]/partitions/[partition]/state。注意，该操作只有Controller版本在3.1至3.3的过程中无变化时才会执行，否则跳转到3.1。
* 直接通过RPC向set\_p相关的Broker发送LeaderAndISRRequest命令。Controller可以在一个RPC操作中发送多个命令从而提高效率。

Broker failover顺序图如下所示。



LeaderAndIsrRequest结构如下



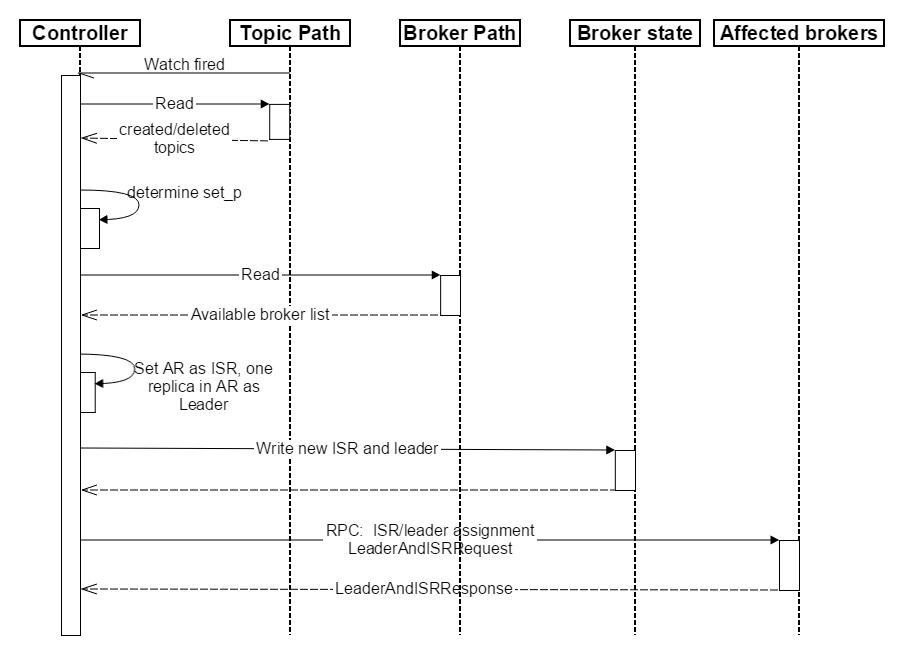
LeaderAndIsrResponse结构如下

http://cdn.infoqstatic.com/statics_s2_20170816-0412u2/resource/articles/kafka-analysis-part-3/zh/resources/0606002.png

### 创建/删除Topic

* Controller在ZooKeeper的/brokers/topics节点上注册Watch，一旦某个Topic被创建或删除，则Controller会通过Watch得到新创建/删除的Topic的Partition/Replica分配。
* 对于删除Topic操作，Topic工具会将该Topic名字存于/admin/delete\_topics。若delete.topic.enable为true，则Controller注册在/admin/delete\_topics上的Watch被fire，Controller通过回调向对应的Broker发送StopReplicaRequest，若为false则Controller不会在/admin/delete\_topics上注册Watch，也就不会对该事件作出反应。
* 对于创建Topic操作，Controller从/brokers/ids读取当前所有可用的Broker列表，对于set\_p中的每一个Partition：
  + 3.1 从分配给该Partition的所有Replica（称为AR）中任选一个可用的Broker作为新的Leader，并将AR设置为新的ISR（因为该Topic是新创建的，所以AR中所有的Replica都没有数据，可认为它们都是同步的，也即都在ISR中，任意一个Replica都可作为Leader）
  + 3.2 将新的Leader和ISR写入/brokers/topics/[topic]/partitions/[partition]
* 直接通过RPC向相关的Broker发送LeaderAndISRRequest。

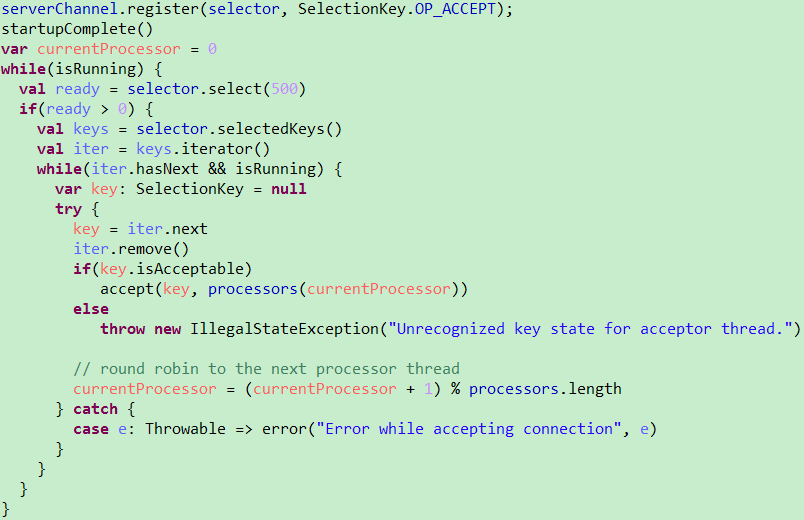
创建Topic顺序图如下所示。



### Broker响应请求流程

Broker通过kafka.network.SocketServer及相关模块接受各种请求并作出响应。整个网络通信模块基于Java NIO开发，并采用Reactor模式，其中包含1个Acceptor负责接受客户请求，N个Processor负责读写数据，M个Handler处理业务逻辑。

Acceptor的主要职责是监听并接受客户端（请求发起方，包括但不限于Producer，Consumer，Controller，Admin Tool）的连接请求，并建立和客户端的数据传输通道，然后为该客户端指定一个Processor，至此它对该客户端该次请求的任务就结束了，它可以去响应下一个客户端的连接请求了。其核心代码如下。



Processor主要负责从客户端读取数据并将响应返回给客户端，它本身并不处理具体的业务逻辑，并且其内部维护了一个队列来保存分配给它的所有SocketChannel。Processor的run方法会循环从队列中取出新的SocketChannel并将其SelectionKey.OP\_READ注册到selector上，然后循环处理已就绪的读（请求）和写（响应）。Processor读取完数据后，将其封装成Request对象并将其交给RequestChannel。

RequestChannel是Processor和KafkaRequestHandler交换数据的地方，它包含一个队列requestQueue用来存放Processor加入的Request，KafkaRequestHandler会从里面取出Request来处理；同时它还包含一个respondQueue，用来存放KafkaRequestHandler处理完Request后返还给客户端的Response。

Processor会通过processNewResponses方法依次将requestChannel中responseQueue保存的Response取出，并将对应的SelectionKey.OP\_WRITE事件注册到selector上。当selector的select方法返回时，对检测到的可写通道，调用write方法将Response返回给客户端。

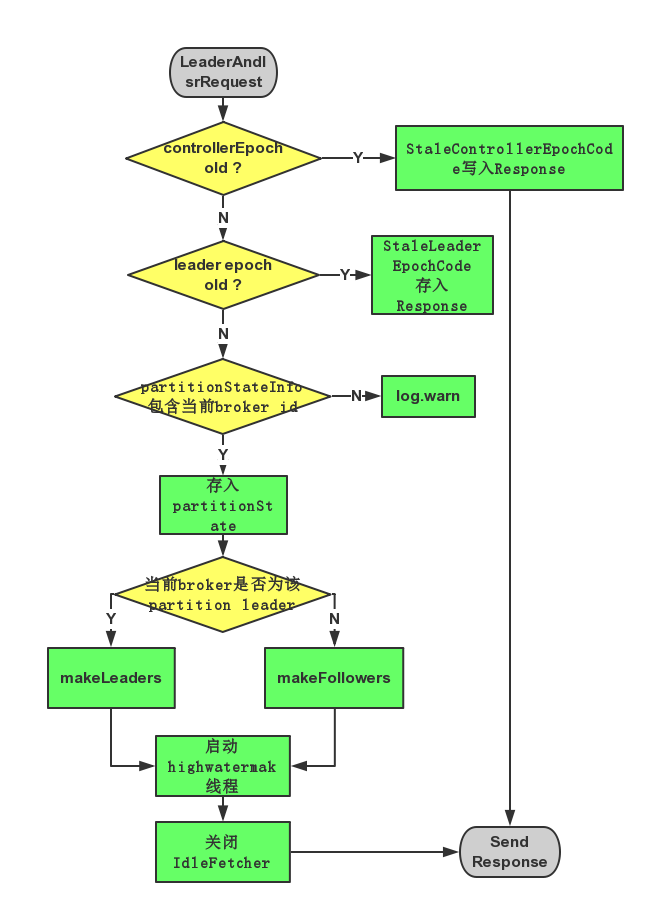
KafkaRequestHandler循环从RequestChannel中取Request并交给kafka.server.KafkaApis处理具体的业务逻辑。

### LeaderAndIsrRequest响应过程

对于收到的LeaderAndIsrRequest，Broker主要通过ReplicaManager的becomeLeaderOrFollower处理，流程如下：

* 若请求中controllerEpoch小于当前最新的controllerEpoch，则直接返回ErrorMapping.StaleControllerEpochCode。
* 对于请求中partitionStateInfos中的每一个元素，即（(topic, partitionId), partitionStateInfo)：
  + 2.1 若partitionStateInfo中的leader epoch大于当前ReplicManager中存储的(topic, partitionId)对应的partition的leader epoch，则：
    - 2.1.1 若当前brokerid（或者说replica id）在partitionStateInfo中，则将该partition及partitionStateInfo存入一个名为partitionState的HashMap中
    - 2.1.2 否则说明该Broker不在该Partition分配的Replica list中，将该信息记录于log中
  + 2.2 否则将相应的Error code（ErrorMapping.StaleLeaderEpochCode）存入Response中
* 筛选出partitionState中Leader与当前Broker ID相等的所有记录存入partitionsTobeLeader中，其它记录存入partitionsToBeFollower中。
* 若partitionsTobeLeader不为空，则对其执行makeLeaders方。
* 若partitionsToBeFollower不为空，则对其执行makeFollowers方法。
* 若highwatermak线程还未启动，则将其启动，并将hwThreadInitialized设为true。
* 关闭所有Idle状态的Fetcher。

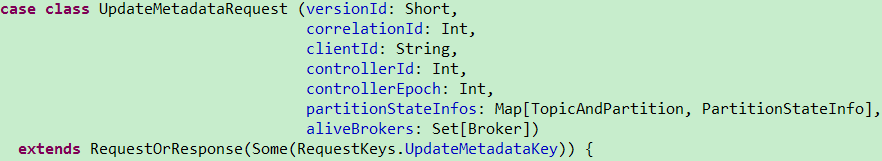
LeaderAndIsrRequest处理过程如下图所示



### Broker启动过程

Broker启动后首先根据其ID在ZooKeeper的/brokers/idszonde下创建临时子节点（[Ephemeral node](http://zookeeper.apache.org/doc/trunk/zookeeperOver.html#Nodes+and+ephemeral+nodes)），创建成功后Controller的ReplicaStateMachine注册其上的Broker Change Watch会被fire，从而通过回调KafkaController.onBrokerStartup方法完成以下步骤：

* 向所有新启动的Broker发送UpdateMetadataRequest，其定义如下。



* 将新启动的Broker上的所有Replica设置为OnlineReplica状态，同时这些Broker会为这些Partition启动high watermark线程。
* 通过partitionStateMachine触发OnlinePartitionStateChange。

### Controller Failover

Controller也需要Failover。每个Broker都会在Controller Path (/controller)上注册一个Watch。当前Controller失败时，对应的Controller Path会自动消失（因为它是Ephemeral Node），此时该Watch被fire，所有“活”着的Broker都会去竞选成为新的Controller（创建新的Controller Path），但是只会有一个竞选成功（这点由ZooKeeper保证）。竞选成功者即为新的Leader，竞选失败者则重新在新的Controller Path上注册Watch。因为[ZooKeeper的Watch是一次性的，被fire一次之后即失效](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkWatches)，所以需要重新注册。

Broker成功竞选为新Controller后会触发KafkaController.onControllerFailover方法，并在该方法中完成如下操作：

* 读取并增加Controller Epoch。
* 在ReassignedPartitions Patch(/admin/reassign\_partitions)上注册Watch。
* 在PreferredReplicaElection Path(/admin/preferred\_replica\_election)上注册Watch。
* 通过partitionStateMachine在Broker Topics Patch(/brokers/topics)上注册Watch。
* 若delete.topic.enable设置为true（默认值是false），则partitionStateMachine在Delete Topic Patch(/admin/delete\_topics)上注册Watch。
* 通过replicaStateMachine在Broker Ids Patch(/brokers/ids)上注册Watch。
* 初始化ControllerContext对象，设置当前所有Topic，“活”着的Broker列表，所有Partition的Leader及ISR等。
* 启动replicaStateMachine和partitionStateMachine。
* 将brokerState状态设置为RunningAsController。
* 将每个Partition的Leadership信息发送给所有“活”着的Broker。
* 若auto.leader.rebalance.enable配置为true（默认值是true），则启动partition-rebalance线程。
* 若delete.topic.enable设置为true且Delete Topic Patch(/admin/delete\_topics)中有值，则删除相应的Topic。

### Partition重新分配

管理工具发出重新分配Partition请求后，会将相应信息写到/admin/reassign\_partitions上，而该操作会触发ReassignedPartitionsIsrChangeListener，从而通过执行回调函数KafkaController.onPartitionReassignment来完成以下操作：

* 将ZooKeeper中的AR（Current Assigned Replicas）更新为OAR（Original list of replicas for partition） + RAR（Reassigned replicas）。
* 强制更新ZooKeeper中的leader epoch，向AR中的每个Replica发送LeaderAndIsrRequest。
* 将RAR - OAR中的Replica设置为NewReplica状态。
* 等待直到RAR中所有的Replica都与其Leader同步。
* 将RAR中所有的Replica都设置为OnlineReplica状态。
* 将Cache中的AR设置为RAR。
* 若Leader不在RAR中，则从RAR中重新选举出一个新的Leader并发送LeaderAndIsrRequest。若新的Leader不是从RAR中选举而出，则还要增加ZooKeeper中的leader epoch。
* 将OAR - RAR中的所有Replica设置为OfflineReplica状态，该过程包含两部分。第一，将ZooKeeper上ISR中的OAR - RAR移除并向Leader发送LeaderAndIsrRequest从而通知这些Replica已经从ISR中移除；第二，向OAR - RAR中的Replica发送StopReplicaRequest从而停止不再分配给该Partition的Replica。
* 将OAR - RAR中的所有Replica设置为NonExistentReplica状态从而将其从磁盘上删除。
* 将ZooKeeper中的AR设置为RAR。
* 删除/admin/reassign\_partition。

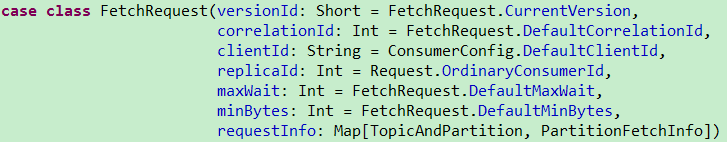
注意：最后一步才将ZooKeeper中的AR更新，因为这是唯一一个持久存储AR的地方，如果Controller在这一步之前crash，新的Controller仍然能够继续完成该过程。

以下是Partition重新分配的案例，OAR = ｛1，2，3｝，RAR = ｛4，5，6｝，Partition重新分配过程中ZooKeeper中的AR和Leader/ISR路径如下

| **AR** | **leader/isr** | **Sttep** |
| --- | --- | --- |
| {1,2,3} | 1/{1,2,3} | (initial state) |
| {1,2,3,4,5,6} | 1/{1,2,3} | (step 2) |
| {1,2,3,4,5,6} | 1/{1,2,3,4,5,6} | (step 4) |
| {1,2,3,4,5,6} | 4/{1,2,3,4,5,6} | (step 7) |
| {1,2,3,4,5,6} | 4/{4,5,6} | (step 8) |
| {4,5,6} | 4/{4,5,6} | (step 10) |

### Follower从Leader Fetch数据

Follower通过向Leader发送FetchRequest获取消息，FetchRequest结构如下



从FetchRequest的结构可以看出，每个Fetch请求都要指定最大等待时间和最小获取字节数，以及由TopicAndPartition和PartitionFetchInfo构成的Map。实际上，Follower从Leader数据和Consumer从Broker Fetch数据，都是通过FetchRequest请求完成，所以在FetchRequest结构中，其中一个字段是clientID，并且其默认值是ConsumerConfig.DefaultClientId。

Leader收到Fetch请求后，Kafka通过KafkaApis.handleFetchRequest响应该请求，响应过程如下：

* replicaManager根据请求读出数据存入dataRead中。
* 如果该请求来自Follower则更新其相应的LEO（log end offset）以及相应Partition的High Watermark
* 根据dataRead算出可读消息长度（单位为字节）并存入bytesReadable中。
* 满足下面4个条件中的1个，则立即将相应的数据返回
  + Fetch请求不希望等待，即fetchRequest.macWait <= 0
  + Fetch请求不要求一定能取到消息，即fetchRequest.numPartitions <= 0，也即requestInfo为空
  + 有足够的数据可供返回，即bytesReadable >= fetchRequest.minBytes
  + 读取数据时发生异常
* 若不满足以上4个条件，FetchRequest将不会立即返回，并将该请求封装成DelayedFetch。检查该DeplayedFetch是否满足，若满足则返回请求，否则将该请求加入Watch列表

Leader通过以FetchResponse的形式将消息返回给Follower，FetchResponse结构如下

http://cdn.infoqstatic.com/statics_s2_20170816-0412u2/resource/articles/kafka-analysis-part-3/zh/resources/0606008.png

## Replication工具

同一个partition可能会有多个replica（对应server.properties配置中的default.replication.factor=N）。没有 replica 的情况下，一旦 broker 宕机，其上所有 patition 的数据都不可被消费，同时 producer 也不能再将数据存于其上的 patition。引入replication 之后，同一个 partition 可能会有多个 replica，而这时需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。

Kafka 分配 Replica 的算法如下：

* 1. 将所有 broker（假设共 n 个 broker）和待分配的 partition 排序。
* 2. 将第 i 个 partition 分配到第（i mod n）个 broker 上。
* 3. 将第 i 个 partition 的第 j 个 replica 分配到第（(i + j) mode n）个 broker上。

### Topic Tool

$KAFKA\_HOME/bin/kafka-topics.sh，该工具可用于创建、删除、修改、查看某个Topic，也可用于列出所有Topic。另外，该工具还可修改某个Topic的以下配置。

unclean.leader.election.enable

delete.retention.ms

segment.jitter.ms

retention.ms

flush.ms

segment.bytes

flush.messages

segment.ms

retention.bytes

cleanup.policy

segment.index.bytes

min.cleanable.dirty.ratio

max.message.bytes

file.delete.delay.ms

min.insync.replicas

index.interval.bytes

### Replica Verification Tool

$KAFKA\_HOME/bin/kafka-replica-verification.sh，该工具用来验证所指定的一个或多个Topic下每个Partition对应的所有Replica是否都同步。可通过topic-white-list这一参数指定所需要验证的所有Topic，支持正则表达式。

### Preferred Replica Leader Election Tool

#### 用途

有了Replication机制后，每个Partition可能有多个备份。某个Partition的Replica列表叫作AR（Assigned Replicas），AR中的第一个Replica即为“Preferred Replica”。创建一个新的Topic或者给已有Topic增加Partition时，Kafka保证Preferred Replica被均匀分布到集群中的所有Broker上。理想情况下，Preferred Replica会被选为Leader。以上两点保证了所有Partition的Leader被均匀分布到了集群当中，这一点非常重要，因为所有的读写操作都由Leader完成，若Leader分布过于集中，会造成集群负载不均衡。但是，随着集群的运行，该平衡可能会因为Broker的宕机而被打破，该工具就是用来帮助恢复Leader分配的平衡。

事实上，每个Topic从失败中恢复过来后，它默认会被设置为Follower角色，除非某个Partition的Replica全部宕机，而当前Broker是该Partition的AR中第一个恢复回来的Replica。因此，某个Partition的Leader（Preferred Replica）宕机并恢复后，它很可能不再是该Partition的Leader，但仍然是Preferred Replica。

#### 原理

* 在ZooKeeper上创建/admin/preferred\_replica\_election节点，并存入需要调整Preferred Replica的Partition信息。
* Controller一直Watch该节点，一旦该节点被创建，Controller会收到通知，并获取该内容。
* Controller读取Preferred Replica，如果发现该Replica当前并非是Leader并且它在该Partition的ISR中，Controller向该Replica发送LeaderAndIsrRequest，使该Replica成为Leader。如果该Replica当前并非是Leader，且不在ISR中，Controller为了保证没有数据丢失，并不会将其设置为Leader。

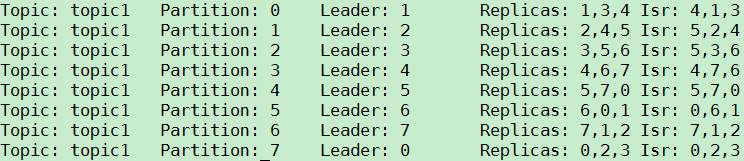
#### 用法

$KAFKA\_HOME/bin/kafka-preferred-replica-election.sh --zookeeper localhost:2181

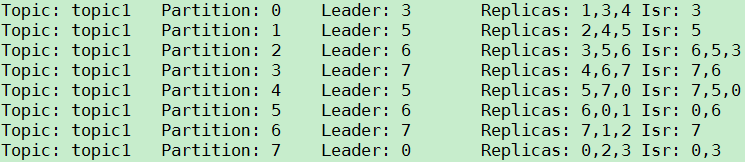
在包含8个Broker的Kafka集群上，创建1个名为topic1，replication-factor为3，Partition数为8的Topic，使用如下命令查看其Partition/Replica分布。

$KAFKA\_HOME/bin/kafka-topics.sh --describe --topic topic1 --zookeeper localhost:2181

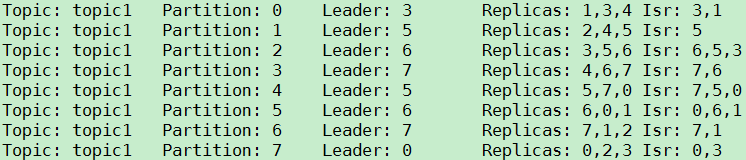
查询结果如下图所示，从图中可以看到，Kafka将所有Replica均匀分布到了整个集群，并且Leader也均匀分布。



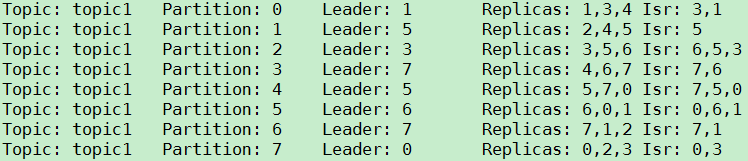
手动停止部分Broker，topic1的Partition/Replica分布如下图所示。从图中可以看到，由于Broker 1/2/4都被停止，Partition 0的Leader由原来的1变为3，Partition 1的Leader由原来的2变为5，Partition 2的Leader由原来的3变为6，Partition 3的Leader由原来的4变为7。



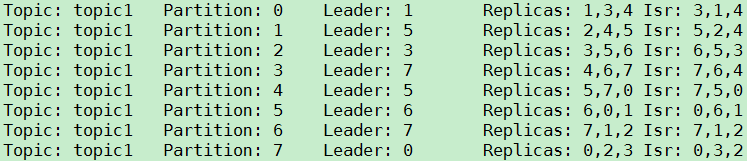
再重新启动ID为1的Broker，topic1的Partition/Replica分布如下。可以看到，虽然Broker 1已经启动（Partition 0和Partition5的ISR中有1），但是1并不是任何一个Parititon的Leader，而Broker 5/6/7都是2个Partition的Leader，即Leader的分布不均衡——一个Broker最多是2个Partition的Leader，而最少是0个Partition的Leader。



运行该工具后，topic1的Partition/Replica分布如下图所示。由图可见，除了Partition 1和Partition 3由于Broker 2和Broker 4还未启动，所以其Leader不是其Preferred Repliac外，其它所有Partition的Leader都是其Preferred Replica。同时，与运行该工具前相比，Leader的分配更均匀——一个Broker最多是2个Parittion的Leader，最少是1个Partition的Leader。



启动Broker 2和Broker 4，Leader分布与上一步相比并未变化，如下图所示。



再次运行该工具，所有Partition的Leader都由其Preferred Replica承担，Leader分布更均匀——每个Broker承担1个Partition的Leader角色。

除了手动运行该工具使Leader分配均匀外，Kafka还提供了自动平衡Leader分配的功能，该功能可通过将auto.leader.rebalance.enable设置为true开启，它将周期性检查Leader分配是否平衡，若不平衡度超过一定阈值则自动由Controller尝试将各Partition的Leader设置为其Preferred Replica。检查周期由leader.imbalance.check.interval.seconds指定，不平衡度阈值由leader.imbalance.per.broker.percentage指定。

### Kafka Reassign Partitions Tool

#### 用途

该工具的设计目标与Preferred Replica Leader Election Tool有些类似，都旨在促进Kafka集群的负载均衡。不同的是，Preferred Replica Leader Election只能在Partition的AR范围内调整其Leader，使Leader分布均匀，而该工具还可以调整Partition的AR。

Follower需要从Leader Fetch数据以保持与Leader同步，所以仅仅保持Leader分布的平衡对整个集群的负载均衡来说是不够的。另外，生产环境下，随着负载的增大，可能需要给Kafka集群扩容。向Kafka集群中增加Broker非常简单方便，但是对于已有的Topic，并不会自动将其Partition迁移到新加入的Broker上，此时可用该工具达到此目的。某些场景下，实际负载可能远小于最初预期负载，此时可用该工具将分布在整个集群上的Partition重装分配到某些机器上，然后可以停止不需要的Broker从而实现节约资源的目的。

需要说明的是，该工具不仅可以调整Partition的AR位置，还可调整其AR数量，即改变该Topic的replication factor。

#### 原理

该工具只负责将所需信息存入ZooKeeper中相应节点，然后退出，不负责相关的具体操作，所有调整都由Controller完成。

* 在ZooKeeper上创建/admin/reassign\_partitions节点，并存入目标Partition列表及其对应的目标AR列表。
* Controller注册在/admin/reassign\_partitions上的Watch被fire，Controller获取该列表。
* 对列表中的所有Partition，Controller会做如下操作：
  + 启动RAR - AR中的Replica，即新分配的Replica。（RAR = Reassigned Replicas， AR = Assigned Replicas）。
  + 等待新的Replica与Leader同步。
  + 如果Leader不在RAR中，从RAR中选出新的Leader。
  + 停止并删除AR - RAR中的Replica，即不再需要的Replica。
  + 删除/admin/reassign\_partitions节点。

#### 用法

该工具有三种使用模式：

* generate模式，给定需要重新分配的Topic，自动生成reassign plan（并不执行）。
* execute模式，根据指定的reassign plan重新分配Partition。
* verify模式，验证重新分配Partition是否成功。

下面这个例子将使用该工具将Topic的所有Partition重新分配到Broker 4/5/6/7上，步骤如下：

* 使用generate模式，生成reassign plan。

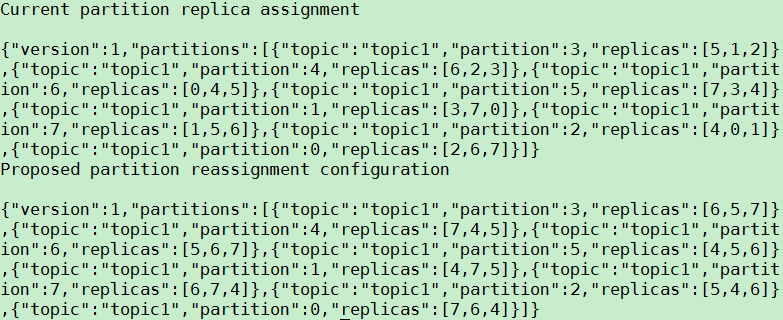
指定需要重新分配的Topic （{"topics":[{"topic":"topic1"}],"version":1}），并存入/tmp/topics-to-move.json文件中，然后执行如下命令：

$KAFKA\_HOME/bin/kafka-reassign-partitions.sh --zookeeper localhost:2181

--topics-to-move-json-file /tmp/topics-to-move.json

--broker-list "4,5,6,7" --generate

结果如下图所示：

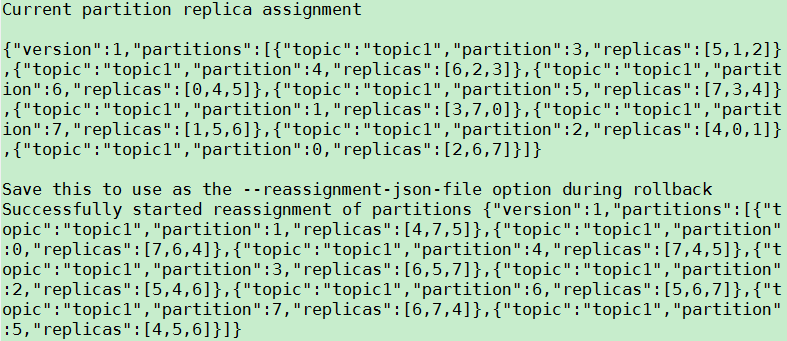


* 使用execute模式，执行reassign plan。

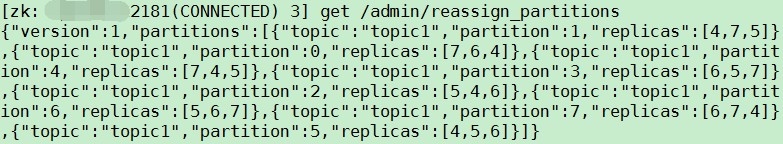
将上一步生成的reassignment plan存入/tmp/reassign-plan.json文件中，并执行：

$KAFKA\_HOME/bin/kafka-reassign-partitions.sh --zookeeper localhost:2181

--reassignment-json-file /tmp/reassign-plan.json --execute



此时，ZooKeeper上/admin/reassign\_partitions节点被创建，且其值与/tmp/reassign-plan.json文件的内容一致。



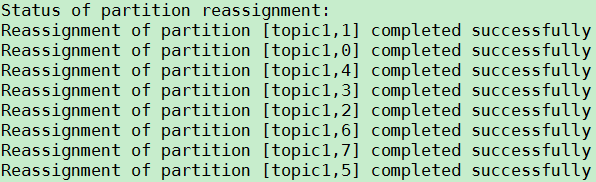
* 使用verify模式，验证reassign是否完成。

执行verify命令：

$KAFKA\_HOME/bin/kafka-reassign-partitions.sh --zookeeper localhost:2181

--reassignment-json-file /tmp/reassign-plan.json --verify

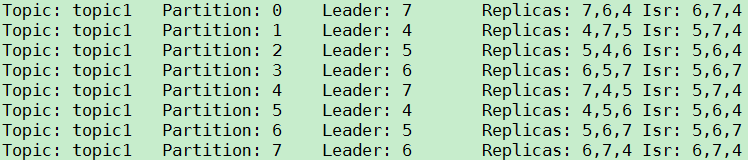
结果如下所示，从图中可看出topic1的所有Partititon都根据reassign plan重新分配成功。



接下来用Topic Tool再次验证。

bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic topic1

结果如下图所示，从图中可看出topic1的所有Partition都被重新分配到Broker 4/5/6/7，且每个Partition的AR与reassign plan一致。



需要说明的是，在使用execute之前，并不一定要使用generate模式自动生成reassign plan，使用generate模式只是为了方便。事实上，某些场景下，generate模式生成的reassign plan并不一定能满足需求，此时用户可以自己设置reassign plan。

### State Change Log Merge Tool

#### 用途

该工具旨在从整个集群的Broker上收集状态改变日志，并生成一个集中的格式化的日志以帮助诊断状态改变相关的故障。每个Broker都会将其收到的状态改变相关的的指令存于名为state-change.log的日志文件中。某些情况下，Partition的Leader election可能会出现问题，此时我们需要对整个集群的状态改变有个全局的了解从而诊断故障并解决问题。该工具将集群中相关的state-change.log日志按时间顺序合并，同时支持用户输入时间范围和目标Topic及Partition作为过滤条件，最终将格式化的结果输出。

#### 用法

bin/kafka-run-class.sh kafka.tools.StateChangeLogMerger

--logs /opt/kafka\_2.11-0.8.2.1/logs/state-change.log

--topic topic1 --partitions 0,1,2,3,4,5,6,7

## leader failover

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

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

当所有 replica 都不工作时，有两种可行的方案：

* 1. 等待 ISR 中的任一个 replica 活过来，并选它作为 leader。可保障数据不丢失，但时间可能相对较长。
* 2. 选择第一个活过来的 replica（不一定是 ISR 成员）作为 leader。无法保障数据不丢失，但相对不可用时间较短。

kafka 0.8.\* 使用第二种方式。kafka 通过 Controller 来选举 leader 。

## controller failover

当 controller 宕机时会触发 controller failover。每个 broker 都会在 zookeeper 的 “/controller”节点注册 watcher，当 controller 宕机时 zookeeper 中的临时节点消失，所有存活的 broker 收到 fire 的通知，每个 broker 都尝试创建新的 controller path，只有一个竞选成功并当选为 controller。

当新的 controller 当选时，会触发 KafkaController.onControllerFailover 方法，在该方法中完成如下操作：

* 1. 读取并增加 Controller Epoch。
* 2. 在 reassignedPartitions Patch(/admin/reassign\_partitions) 上注册 watcher。
* 3. 在 preferredReplicaElection Path(/admin/preferred\_replica\_election) 上注册 watcher。
* 4. 通过 partitionStateMachine 在 broker Topics Patch(/brokers/topics) 上注册 watcher。
* 5. 若 delete.topic.enable=true（默认值是 false），则 partitionStateMachine 在 Delete Topic Patch(/admin/delete\_topics) 上注册 watcher。
* 6. 通过 replicaStateMachine在 Broker Ids Patch(/brokers/ids)上注册Watch。
* 7. 初始化 ControllerContext 对象，设置当前所有 topic，“活”着的 broker 列表，所有 partition 的 leader 及 ISR等。
* 8. 启动 replicaStateMachine 和 partitionStateMachine。
* 9. 将 brokerState 状态设置为 RunningAsController。
* 10. 将每个 partition 的 Leadership 信息发送给所有“活”着的 broker。
* 11. 若 auto.leader.rebalance.enable=true（默认值是true），则启动 partition-rebalance 线程。
* 12. 若 delete.topic.enable=true 且Delete Topic Patch(/admin/delete\_topics)中有值，则删除相应的Topic。

# consumer 消费消息

## consumer API

kafka 提供了两套 consumer API：

* 1. The high-level Consumer API
* 2. The SimpleConsumer API

其中 high-level consumer API 提供了一个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。

### The high-level consumer API

high-level consumer API 提供了 consumer group 的语义，一个消息只能被 group 内的一个 consumer 所消费，且 consumer 消费消息时不关注 offset，最后一个 offset 由 zookeeper 保存。

使用 high-level consumer API 可以是多线程的应用，应当注意：

* 1. 如果消费线程大于 patition 数量，则有些线程将收不到消息。
* 2. 如果 patition 数量大于线程数，则有些线程多收到多个 patition 的消息。
* 3. 如果一个线程消费多个 patition，则无法保证你收到的消息的顺序，而一个 patition 内的消息是有序的。

### The SimpleConsumer API

如果你想要对 patition 有更多的控制权，那就应该使用 SimpleConsumer API，比如：

* 1. 多次读取一个消息。
* 2. 只消费一个 patition 中的部分消息。
* 3. 使用事务来保证一个消息仅被消费一次。

但是使用此 API 时，partition、offset、broker、leader 等对你不再透明，需要自己去管理。你需要做大量的额外工作：

* 1. 必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息。
* 2. 应用程序需要通过程序获知每个 Partition 的 leader 是谁。
* 3. 需要处理 leader 的变更。

 使用 SimpleConsumer API 的一般流程如下：

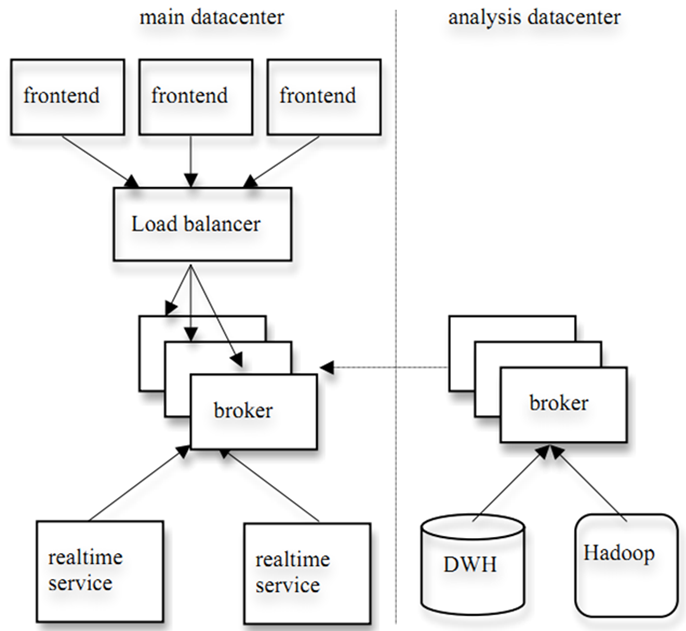
* 1. 查找到一个“活着”的 broker，并且找出每个 partition 的 leader。
* 2. 找出每个 partition 的 follower。
* 3. 定义好请求，该请求应该能描述应用程序需要哪些数据。
* 4. fetch 数据。
* 5. 识别 leader 的变化，并对之作出必要的响应。

以下针对 high-level Consumer API 进行说明。

## consumer group

kafka 的分配单位是 patition。每个 consumer 都属于一个 group，一个 partition 只能被同一个 group 内的一个 consumer 所消费（也就保障了一个消息只能被 group 内的一个 consuemr 所消费），但是多个 group 可以同时消费这个 partition。

kafka 的设计目标之一就是同时实现离线处理和实时处理，根据这一特性，可以使用 spark/Storm 这些实时处理系统对消息在线处理，同时使用 Hadoop 批处理系统进行离线处理，还可以将数据备份到另一个数据中心，只需要保证这三者属于不同的 consumer group。如下图所示：



## 消费方式

consumer 采用 pull 模式从 broker 中读取数据。

push 模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。它的目标是尽可能以最快速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 consumer 的消费能力以适当的速率消费消息。

对于 Kafka 而言，pull 模式更合适，它可简化 broker 的设计，consumer 可自主控制消费消息的速率，同时 consumer 可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

## consumer delivery guarantee

如果将 consumer 设置为 autocommit，consumer 一旦读到数据立即自动 commit。如果只讨论这一读取消息的过程，那 Kafka 确保了 Exactly once。

但实际使用中应用程序并非在 consumer 读取完数据就结束了，而是要进行进一步处理，而数据处理与 commit 的顺序在很大程度上决定了consumer delivery guarantee：

**1.读完消息先 commit 再处理消息。**这种模式下，如果 consumer 在 commit 后还没来得及处理消息就 crash 了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对应于 At most once

**2.读完消息先处理再 commit。**这种模式下，如果在处理完消息之后 commit 之前 consumer crash 了，下次重新开始工作时还会处理刚刚未 commit 的消息，实际上该消息已经被处理过了。这就对应于 At least once。

**3.如果一定要做到 Exactly once，就需要协调 offset 和实际操作的输出。**精典的做法是引入两阶段提交。如果能让 offset 和操作输入存在同一个地方，会更简洁和通用。这种方式可能更好，因为许多输出系统可能不支持两阶段提交。比如，consumer 拿到数据后可能把数据放到 HDFS，如果把最新的 offset 和数据本身一起写到 HDFS，那就可以保证数据的输出和 offset 的更新要么都完成，要么都不完成，间接实现 Exactly once。（目前就 high-level API而言，offset 是存于Zookeeper 中的，无法存于HDFS，而SimpleConsuemr API的 offset 是由自己去维护的，可以将之存于 HDFS 中）

总之，Kafka 默认保证 At least once，并且允许通过设置 producer 异步提交来实现 At most once。而 Exactly once 要求与外部存储系统协作，幸运的是 kafka 提供的 offset 可以非常直接非常容易得使用这种方式。

## consumer rebalance

当有 consumer 加入或退出、以及 partition 的改变（如 broker 加入或退出）时会触发 rebalance。consumer rebalance算法如下：

* 1. 将目标 topic 下的所有 partirtion 排序，存于PT。
* 2. 对某 consumer group 下所有 consumer 排序，存于 CG，第 i 个consumer 记为 Ci。
* 3. N=size(PT)/size(CG)，向上取整。
* 4. 解除 Ci 对原来分配的 partition 的消费权（i从0开始）。
* 5. 将第i\*N到（i+1）\*N-1个 partition 分配给 Ci。

在 0.8.\*版本，每个 consumer 都只负责调整自己所消费的 partition，为了保证整个consumer group 的一致性，当一个 consumer 触发了 rebalance 时，该 consumer group 内的其它所有其它 consumer 也应该同时触发 rebalance。这会导致以下几个问题：

**1.Herd effect。**任何 broker 或者 consumer 的增减都会触发所有的 consumer 的 rebalance

**2.Split Brain。**每个 consumer 分别单独通过 zookeeper 判断哪些 broker 和 consumer 宕机了，那么不同 consumer 在同一时刻从 zookeeper 看到的 view 就可能不一样，这是由 zookeeper 的特性决定的，这就会造成不正确的 reblance 尝试。

**3. 调整结果不可控。**所有的 consumer 都并不知道其它 consumer 的 rebalance 是否成功，这可能会导致 kafka 工作在一个不正确的状态。

基于以上问题，kafka 设计者考虑在0.9.\*版本开始使用中心 coordinator 来控制 consumer rebalance，然后又从简便性和验证要求两方面考虑，计划在 consumer 客户端实现分配方案。

# borker配置

## 最小配置

通常情况下需要在减压缩kafka后，修改config/server.properties 配置文件中的以下两项

log.dirs = kafka-logs

zookeeper.connect = localhost:9092

listeners = PLAINTEXT://ip:9092

* log.dirs 指的是kafka的log Data保存的目录，默认为Null。如果不指定log Data会保存到log.dir设置的目录中，log.dir默认为/tmp/kafka-logs。**需要保证启动KafKaServer的用户对log.dirs或log.dir设置的目录具有读与写的权限**。
* zookeeper.connect 指的是zookeeper集群的地址，可以是多个，多个之间用逗号分割hostname1:port1,hostname2:port2,hostname3:port3
* listeners 监听列表(以逗号分隔 不同的协议(如plaintext,trace,ssl、不同的IP和端口))

## kafka提供的borker配置

# Replication configurations

num.replica.fetchers=4

replica.fetch.max.bytes=1048576

replica.fetch.wait.max.ms=500

replica.high.watermark.checkpoint.interval.ms=5000

replica.socket.timeout.ms=30000

replica.socket.receive.buffer.bytes=65536

replica.lag.time.max.ms=10000

controller.socket.timeout.ms=30000

controller.message.queue.size=10

# Log configuration

num.partitions=8

message.max.bytes=1000000

auto.create.topics.enable=true

log.index.interval.bytes=4096

log.index.size.max.bytes=10485760

log.retention.hours=168

log.flush.interval.ms=10000

log.flush.interval.messages=20000

log.flush.scheduler.interval.ms=2000

log.roll.hours=168

log.retention.check.interval.ms=300000

log.segment.bytes=1073741824

# ZK configuration

zookeeper.connection.timeout.ms=6000

zookeeper.sync.time.ms=2000

# Socket server configuration

num.io.threads=8

num.network.threads=8

socket.request.max.bytes=104857600

socket.receive.buffer.bytes=1048576

socket.send.buffer.bytes=1048576

queued.max.requests=16

fetch.purgatory.purge.interval.requests=100

producer.purgatory.purge.interval.requests=100

## 配置详细说明

| **名称** | **说明** | **类型** | **默认值** | **有效值** | **重要性** |
| --- | --- | --- | --- | --- | --- |
| zookeeper.connect | zookeeper集群的地址，可以是多个，多个之间用逗号分割 | string | localhost:2181 | ip1:port1,ip2:port2 | 高 |
| zookeeper.connection.timeout.ms | 客户端在建立通zookeeper连接中的最大等待时间 | int | null | 6000 | 高 |
| zookeeper.session.timeout.ms | ZooKeeper的session的超时时间，如果在这段时间内没有收到ZK的心跳，则会被认为该Kafka server挂掉了。如果把这个值设置得过低可能被误认为挂掉，如果设置得过高，如果真的挂了，则需要很长时间才能被server得知 | int | 6000 |  | 高 |
| zookeeper.sync.time.ms | 一个ZK follower能落后leader的时间 | int | 2000 |  | 高 |
| listeners | 监听列表(以逗号分隔 不同的协议(如plaintext,trace,ssl、不同的IP和端口)),hostname如果设置为0.0.0.0则绑定所有的网卡地址；如果hostname为空则绑定默认的网卡。如果没有配置则默认为java.net.InetAddress.getCanonicalHostName() | string | null | 如：PLAINTEXT://myhost:9092,TRACE://:9091 或 PLAINTEXT://0.0.0.0:9092, | 高 |
| host.name | 。如果设置了它，会仅绑定这个地址。如果没有设置，则会绑定所有的网络接口，并提交一个给ZK。**不推荐使用 只有当listeners没有设置时才有必要使用。** | string | “’ | 如：”localhost” | 高 |
| port | server用来接受client连接的端口。**不推荐使用,使用listeners配置项代替；只有在listeners没有配置时才使用。** | int | 9092 |  | 高 |
| advertised.host.name | 会将hostname通知给生产者和消费者，在多网卡时需要设置该值为另一个ip地址。如果没有设置该值，则返回 配置项host.name设置的值，如果host.name没有设置则返回java.net.InetAddress.getCanonicalHostName()**不推荐使用 只有当advertised.listeners或listeners没有设置时才有必要使用。** | string | null |  | 高 |
| advertised.listeners | 设置不同于listeners配置的监听列表即不同于listeners设置的网卡地址及端口；如果没有配置，会使用listeners的值 | string | null |  | 高 |
| advertised.port | 分发这个端口给所有的producer，consumer和其他broker来建立连接。如果此端口跟server绑定的端口不同，则才有必要设置。**不推荐使用 只有当advertised.listeners或listeners没有设置时才有必要使用。** | int | null |  | 高 |
| auto.create.topics.enable | 是否允许自动创建topic。如果设为true，那么produce，consume或者fetch metadata一个不存在的topic时，就会自动创建一个默认replication factor和partition number的topic。 | boolean | true |  | 高 |
| background.threads | 一些后台任务处理的线程数，例如过期消息文件的删除等，一般情况下不需要去做修改 | int | 10 |  | 高 |
| broker.id | 每一个broker在集群中的唯一表示，要求是正数。当该服务器的IP地址发生改变时，broker.id没有变化，则不会影响consumers的消息情况。 | int | -1 |  | 高 |
| compression.type | 指定topic的压缩类型。除了支持’gzip’, ‘snappy’, ‘lz4’外，还支持”uncompressed(不压缩)”以及producer(由producer来指定) | string | producer |  | 高 |
| delete.topic.enable | 是否启动删除topic。如果设置为false,你在删除topic的时候无法删除，但是会打上一个你将删除该topic的标记，等到你修改这一属性的值为true后重新启动Kafka集群的时候，集群自动将那些标记删除的topic删除掉，对应的log.dirs目录下的topic目录和数据也会被删除。而将这一属性设置为true之后，你就能成功删除你想要删除的topic了 | boolean | false |  | 高 |
| auto.leader.rebalance.enable | 一个后台线程会周期性的自动尝试，为所有的broker的每个partition平衡leadership，使kafka的leader均衡。 | boolean | true |  | 高 |
| leader.imbalance.check.interval.seconds | 检查leader是否均衡的时间间隔(秒) | long | 300 |  | 高 |
| leader.imbalance.per.broker.percentage | 每个broker允许的不平衡的leader的百分比。如果每个broker超过了这个百分比，复制控制器会重新平衡leadership。 | int | 10 |  | 高 |
| log.flush.interval.messages | 数据flush(sync)到硬盘前之前累积的消息条数，因为磁盘IO操作是一个慢操作,但又是一个”数据可靠性”的必要手段,所以此参数的设置,需要在”数据可靠性”与”性能”之间做必要的权衡.如果此值过大,将会导致每次”fsync”的时间较长(IO阻塞),如果此值过小,将会导致”fsync”的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失 | long | 9223372036854775807 |  | 高 |
| log.flush.interval.ms | 当达到下面的时间(ms)时，执行一次强制的flush操作。interval.ms和interval.messages无论哪个达到，都会flush。 | long | null |  | 高 |
| log.flush.offset.checkpoint.interval.ms | 记录上次把log刷到磁盘的时间点的频率，用来日后的recovery。通常不需要改变 | long | 60000 |  | 高 |
| log.flush.scheduler.interval.ms | 检查是否需要固化到硬盘的时间间隔 | long | 9223372036854775807 |  | 高 |
| log.retention.bytes | topic每个分区的最大文件大小，一个topic的大小限制 = 分区数\*log.retention.bytes。-1没有大小限log.retention.bytes和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 | loong | -1 |  | 高 |
| log.retention.hours | 日志保存时间，默认为7天（168小时）。超过这个时间会根据policy处理数据。bytes和minutes无论哪个先达到都会触发 | int | 168 |  | 高 |
| log.retention.minutes | 数据存储的最大时间超过这个时间会根据log.cleanup.policy设置的策略处理数据，也就是消费端能够多久去消费数据 |  |  |  |  |
| log.retention.bytes和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 | int | null |  | 高 |  |
| log.roll.hous | 当达到下面时间，会强制新建一个segment。这个参数会在日志segment没有达到log.segment.bytes设置的大小，也会强制新建一个segment会 | int | 168 |  | 高 |
| log.roll.jitter.{ms,hours} | 从logRollTimeMillis抽离的jitter最大数目 | int | 0 |  | 高 |
| log.segment.bytes | topic partition的日志存放在某个目录下诸多文件中，这些文件将partition的日志切分成一段一段的；这个属性就是每个文件的最大尺寸；当尺寸达到这个数值时，就会创建新文件。此设置可以由每个topic基础设置时进行覆盖 | long | 1G=1024\*1024\*1024 |  | 高 |
| log.segment.delet.delay.ms | 删除文件系统上文件的等待时间，默认是1分钟 | long | 6000 |  | 高 |
| message.max.bytes | 表示一个服务器能够接收处理的消息的最大字节数，注意这个值producer和consumer必须设置一致，且不要大于fetch.message.max.bytes属性的值该值默认是1000012字节，大概900KB |  |  |  |  |
| int | 1000012 |  | 高 |  |  |
| min.insync.replicas | 该属性规定了最小的ISR数。当producer设置request.required.acks为all或-1时，指定副本(replicas)的最小数目（必须确认每一个repica的写数据都是成功的），如果这个数目没有达到，producer会产生异常。 | int | 1 |  | 高 |
| num.io.threads | 服务器用来处理请求的I/O线程的数目；这个线程数目至少要等于硬盘的个数。 | int | 8 |  | 高 |
| num.network.threads | 服务器用来处理网络请求的网络线程数目；一般你不需要更改这个属性 | int | 3 |  | 高 |
| num.recovery.threads.per.data.dir | 每数据目录用于日志恢复启动和关闭冲洗时的线程数量 | int | 1 |  | 高 |
| num.replica.fetchers | 从leader进行复制消息的线程数，增大这个数值会增加follower的IO | int | 1 |  | 高 |
| offset.metadata.max.bytes | 允许client(消费者)保存它们元数据(offset)的最大的数据量 | int | 4096(4kb) |  |  |
| offsets.commit.required.acks | 在offset commit可以接受之前，需要设置确认的数目，一般不需要更改 | int | -1 |  | 高 |
| offsets.commit.timeout.ms | offset commit会延迟直至此超时或所需的副本数都收到offset commit，这类似于producer请求的超时 | int | 5000 |  | 高 |
| offsets.load.buffer.size | 此设置对应于offset manager在读取缓存offset segment的批量大小（以字节为单位). | int | 5242880 |  | 高 |
| offsets.retention.check.interval.ms | offset管理器检查陈旧offsets的频率 | long | 600000(10分钟) |  | 高 |
| offsets.topic.num.partitions | 偏移的提交topic的分区数目。 由于目前不支持部署之后改变，我们建议您使用生产较高的设置（例如，100-200） | int | 50 |  | 高 |
| offsets.topic.replication.factor | 复制因子的offset提交topic。较高的设置（例如三个或四个），建议以确保更高的可用性。如果offset topic创建时，broker比复制因子少，offset topic将以较少的副本创建。 | short | 3 |  | 高 |
| offsets.topic.segment.bytes | offset topic的Segment大小。因为它使用压缩的topic，所有Sgment的大小应该保持小一点，以促进更快的日志压实和负载 | int | 104857600 |  | 高 |
| queued.max.requests | 在网络线程(network threads)停止读取新请求之前，可以排队等待I/O线程处理的最大请求个数。若是等待IO的请求超过这个数值，那么会停止接受外部消息 | int | 500 |  | 高 |
| quota.consumer.default | 以clientid或consumer group区分的consumer端每秒可以抓取的最大byte | long | 9223372036854775807 |  | 高 |
| quota.producer.default | producer端每秒可以产生的最大byte | long | 9223372036854775807 |  | 高 |
| replica.fetch.max.bytes | replicas每次获取数据的最大字节数 | int | 1048576 |  | 高 |
| replica.fetch.min.bytes | fetch的最小数据尺寸,如果leader中尚未同步的数据不足此值,将会阻塞,直到满足条件 | int | 1 |  | 高 |
| replica.fetch.wait.max.ms | replicas同leader之间通信的最大等待时间，失败了会重试。这个值须小于replica.lag.time.max.ms，以防止低吞吐量主题ISR频繁收缩 | int | 500 |  | 高 |
| replica.high.watermark.checkpoint.interval.ms | 每一个replica存储自己的high watermark到磁盘的频率，用来日后的recovery | int | 5000 |  | 高 |
| replica.socket.timeout.ms | 复制数据过程中，replica发送给leader的网络请求的socket超时时间,至少等于replica.fetch.wait.max.ms | int | 30000 |  | 高 |
| replica.socket.receive.buffer.bytes | 复制过程leader接受请求的buffer大小 | int | 65536(64\*1024) |  | 高 |
| replica.lag.time.max.ms | replicas响应partition leader的最长等待时间，若是超过这个时间，就将replicas列入ISR(in-sync replicas)，并认为它是死的，不会再加入管理中 | long | 10000 |  | 高 |
| replica.lag.max.messages | 如果follower落后与leader太多,将会认为此follower[或者说partition relicas]已经失效。 通常,在follower与leader通讯时,因为网络延迟或者链接断开,总会导致replicas中消息同步滞后如果消息之后太多,leader将认为此follower网络延迟较大或者消息吞吐能力有限,将会把此replicas迁移到其他follower中.在broker数量较少,或者网络不足的环境中,建议提高此值. | int | 4000 |  | 高 |
| request.timeout.ms | producer等待响应的最长时间，如果超时将重发几次，最终报错 | int | 30000 |  | 高 |
| socket.receive.buffer.bytes | socket用于接收网络请求的缓存大小 | int | 102400 |  | 高 |
| socket.request.max.bytes | server能接受的请求的最大的大小，这是为了防止server跑光内存，不能大于Java堆的大小。 | int | 104857600(100\*1024\*1024) |  | 高 |
| socket.send.buffer.bytes | server端用来处理socket连接的SO\_SNDBUFF缓冲大小 | int | 102400 |  | 高 |
| controller.socket.timeout.ms | partition管理控制器进行备份时，socket的超时时间 | int | 30000 |  | 高 |
| controller.message.queue.size | partition leader与replicas数据同步时,消息的队列大小 | int | 10 |  | 高 |
| num.partitions | 每个topic的分区个数，若是在topic创建时候没有指定的话会被topic创建时的指定参数覆盖 | int | 1 | 推荐设为8 | 高 |
| log.index.interval.bytes | 当执行一次fetch后，需要一定的空间扫描最近的offset，设置的越大越好，但是也更耗内存一般使用默认值就可以 | int | 4096 |  | 中 |
| log.index.size.max.bytes | 每个log segment的最大尺寸。注意，如果log尺寸达到这个数值，即使尺寸没有超过log.segment.bytes限制，也需要产生新的log segment。 | int | 10485760 |  | 中 |
| fetch.purgatory.purge.interval.requests | 非立即答复请求放入purgatory中，当到达或超出interval时认为request complete | int | 1000 |  | 中 |
| producer.purgatory.purge.interval.requests | producer请求清除时间 | int | 1000 |  | 中 |
| default.replication.factor | 一个topic ，默认分区的replication个数 ，不能大于集群中broker的个数。 | int | 1 |  | 中 |
| group.max.session.timeout.ms | 注册consumer允许的最大超时时间 | int | 300000 |  | 中 |
| group.min.session.timeout.ms | 注册consumer允许的最小超时时间 | int | 6000 |  | 中 |
| inter.broker.protocol.version | broker协议版本 | string | 0.10.0 |  | 中 |
| log.cleaner.backoff.ms | 检查log是否需要clean的时间间隔 | long | 15000 |  | 中 |
| log.cleaner.dedupe.buffer.size | 日志压缩去重时候的缓存空间，在空间允许的情况下，越大越好 | long | 134217728 |  | 中 |
| log.cleaner.delete.retention.ms | 保存时间；保存压缩日志的最长时间；也是客户端消费消息的最长时间，同log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据；会被topic创建时的指定时间覆盖。 | long | 86400000(一天) |  | 中 |
| log.cleaner.enable | 是否启动压缩日志,当这个属性设置为false时，一旦日志的保存时间或者大小达到上限时，就会被删除；如果设置为true，则当保存属性达到上限时，就会进行压缩 | boolean | false |  | 中 |
| log.cleaner.threads | 日志压缩运行的线程数 | int | 1 |  | 中 |
| log.cleaner.io.buffer.load.factor | 日志清理中hash表的扩大因子，一般不需要修改 | double | 0.9 |  | 中 |
| log.cleaner.io.buffer.size | log cleaner清除过程中针对日志进行索引化以及精简化所用到的缓存大小。最好设置大点，以提供充足的内存 | int | 524288 |  | 中 |
| log.cleaner.io.max.bytes.per.second | 进行log compaction时，log cleaner可以拥有的最大I/O数目。这项设置限制了cleaner，以避免干扰活动的请求服务。 | double | 1.7976931348623157E308 |  | 中 |
| log.cleaner.min.cleanable.ratio | 这项配置控制log compactor试图清理日志的频率（假定[log compaction]是打开的）。默认避免清理压缩超过50%的日志。这个比率绑定了备份日志所消耗的最大空间（50%的日志备份时压缩率为50%）。更高的比率则意味着浪费消耗更少，也就可以更有效的清理更多的空间。这项设置在每个topic设置中可以覆盖 | double | 0.5 |  | 中 |
| log.preallocate | 是否预创建新文件，windows推荐使用 | boolean | false |  | 中 |
| log.retention.check.interval.ms | 检查日志分段文件的间隔时间，以确定是否文件属性是否到达删除要求。 | long | 300000 |  | 中 |
| max.connections.per.ip | 一个broker允许从每个ip地址连接的最大数目 | int | 2147483647=Int.MaxValue |  | 中 |
| max.connections.per.ip.overrides | 每个IP或主机名覆盖连接的默认最大数量 | string | “” |  | 中 |
| replica.fetch.backoff.ms | 复制数据时失败等待时间 | int | 1000 |  | 中 |
| reserved.broker.max.id | broker可以使用的最大ID值 | int | 1000 |  | 中 |

## topic level 配置

broker级别的参数可以由topic级别的覆写，不是所有的broker参数在topic级别都有对应值。

以下是topic-level的配置选项。server的默认配置在Server Default Property列下给出了，设定这些默认值不会改变原有的设置。

| **Property** | **Default** | **Server Default Property** | **Description** |
| --- | --- | --- | --- |
| cleanup.policy | delete | log.cleanup.policy | 要么是”delete“要么是”compact“； 这个字符串指明了针对旧日志部分的利用方式；默认方式（”delete”）将会丢弃旧的部分当他们的回收时间或者尺寸限制到达时。”compact“将会进行日志压缩 |
| delete.retention.ms | 86400000 (24 hours) | log.cleaner.delete.retention.ms | 对于压缩日志保留的最长时间，也是客户端消费消息的最长时间，通log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据。此项配置可以在topic创建时的置顶参数覆盖 |
| flush.messages | none | log.flush.interval.messages | 此项配置指定时间间隔：强制进行fsync日志。例如，如果这个选项设置为1，那么每条消息之后都需要进行fsync，如果设置为5，则每5条消息就需要进行一次fsync。一般来说，建议你不要设置这个值。此参数的设置,需要在”数据可靠性”与”性能”之间做必要的权衡.如果此值过大,将会导致每次”fsync”的时间较长(IO阻塞),如果此值过小,将会导致”fsync”的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失. |
| flush.ms | None | log.flush.interval.ms | 此项配置用来置顶强制进行fsync日志到磁盘的时间间隔；例如，如果设置为1000，那么每1000ms就需要进行一次fsync。一般不建议使用这个选项 |
| index.interval.bytes | 4096 | log.index.interval.bytes | 默认设置保证了我们每4096个字节就对消息添加一个索引，更多的索引使得阅读的消息更加靠近，但是索引规模却会由此增大；一般不需要改变这个选项 |
| max.message.bytes | 1000000 | max.message.bytes | kafka追加消息的最大尺寸。注意如果你增大这个尺寸，你也必须增大你consumer的fetch 尺寸，这样consumer才能fetch到这些最大尺寸的消息。 |
| min.cleanable.dirty.ratio | 0.5 | in.cleanable.dirty.ratio | 此项配置控制log压缩器试图进行清除日志的频率。默认情况下，将避免清除压缩率超过50%的日志。这个比率避免了最大的空间浪费 |
| min.insync.replicas | 1 | min.insync.replicas | 当producer设置 acks为-1时，min.insync.replicas指定replicas的最小数目（必须确认每一个repica的写数据都是成功的），如果这个数目没有达到，producer会产生异常。 |
| retention.bytes | None | log.retention.bytes | 如果使用“delete”的retention 策略，这项配置就是指在删除日志之前，日志所能达到的最大尺寸。默认情况下，没有尺寸限制而只有时间限制 |
| retention.ms | 7 days | log.retention.minutes | 如果使用“delete”的retention策略，这项配置就是指删除日志前日志保存的时间。 |
| segment.bytes | 1GB | log.segment.bytes | kafka中log日志是分成一块块存储的，此配置是指log日志划分成块的大小 |
| segment.index.bytes | 10MB | log.index.size.max.bytes | 此配置是有关offsets和文件位置之间映射的索引文件的大小；一般不需要修改这个配置 |
| segment.ms | 7 days | log.roll.hours | 即使log的分块(segment)文件没有达到需要删除、压缩的大小，一旦log 的时间达到这个上限，就会强制新建一个log分块文件 |
| segment.jitter.ms | 0 | log.roll.jitter.{ms,hours} | The maximum jitter to subtract from logRollTimeMillis. |

# 注意事项

## producer 无法发送消息的问题

最开始在本机搭建了kafka伪集群，本地 producer 客户端成功发布消息至 broker。随后在服务器上搭建了 kafka 集群，在本机连接该集群，producer 却无法发布消息到 broker（奇怪也没有抛错）。最开始怀疑是 iptables 没开放，于是开放端口，结果还不行（又开始是代码问题、版本问题等等，倒腾了很久）。最后没办法，一项一项查看 server.properties 配置，发现以下两个配置：

# The address the socket server listens on. It will get the value returned from

# java.net.InetAddress.getCanonicalHostName() if not configured.

# FORMAT:

# listeners = security\_protocol://host\_name:port

# EXAMPLE:

# listeners = PLAINTEXT://your.host.name:9092

listeners=PLAINTEXT://:9092

　# Hostname and port the broker will advertise to producers and consumers. If not set,

　# it uses the value for "listeners" if configured. Otherwise, it will use the value

　# returned from java.net.InetAddress.getCanonicalHostName().

　#advertised.listeners=PLAINTEXT://your.host.name:9092

以上说的就是 advertised.listeners 是 broker 给 producer 和 consumer 连接使用的，如果没有设置，就使用 listeners，而如果 host\_name 没有设置的话，就使用 java.net.InetAddress.getCanonicalHostName() 方法返回的主机名。

修改方法：

1. listeners=PLAINTEXT://121.10.26.XXX:9092

2. advertised.listeners=PLAINTEXT://121.10.26.XXX:9092

修改后重启服务，正常工作。

## [kafka consumer防止数据丢失](http://kane-xie.iteye.com/blog/2225085)

kafka最初是被LinkedIn设计用来处理log的分布式消息系统，因此它的着眼点不在数据的安全性（log偶尔丢几条无所谓），换句话说kafka并不能完全保证数据不丢失。

尽管kafka官网声称能够保证at-least-once，但如果consumer进程数小于partition\_num，这个结论不一定成立。

考虑这样一个case，partiton\_num=2，启动一个consumer进程订阅这个topic，对应的，stream\_num设为2，也就是说启两个线程并行处理message。

如果auto.commit.enable=true，当consumer fetch了一些数据但还没有完全处理掉的时候，刚好到commit interval出发了提交offset操作，接着consumer crash掉了。这时已经fetch的数据还没有处理完成但已经被commit掉，因此没有机会再次被处理，数据丢失。

如果auto.commit.enable=false，假设consumer的两个fetcher各自拿了一条数据，并且由两个线程同时处理，这时线程t1处理完partition1的数据，手动提交offset，这里需要着重说明的是，当手动执行commit的时候，实际上是对这个consumer进程所占有的所有partition进行commit，kafka暂时还没有提供更细粒度的commit方式，也就是说，即使t2没有处理完partition2的数据，offset也被t1提交掉了。如果这时consumer crash掉，t2正在处理的这条数据就丢失了。

如果希望能够严格的不丢数据，解决办法有两个：

* 手动commit offset，并针对partition\_num启同样数目的consumer进程，这样就能保证一个consumer进程占有一个partition，commit offset的时候不会影响别的partition的offset。但这个方法比较局限，因为partition和consumer进程的数目必须严格对应。
* 另一个方法同样需要手动commit offset，另外在consumer端再将所有fetch到的数据缓存到queue里，当把queue里所有的数据处理完之后，再批量提交offset，这样就能保证只有处理完的数据才被commit。