

---

# 分布式消息系统 Kafka

课程讲义

主讲: Reythor 雷

2019

# 分布式消息系统 Kafka

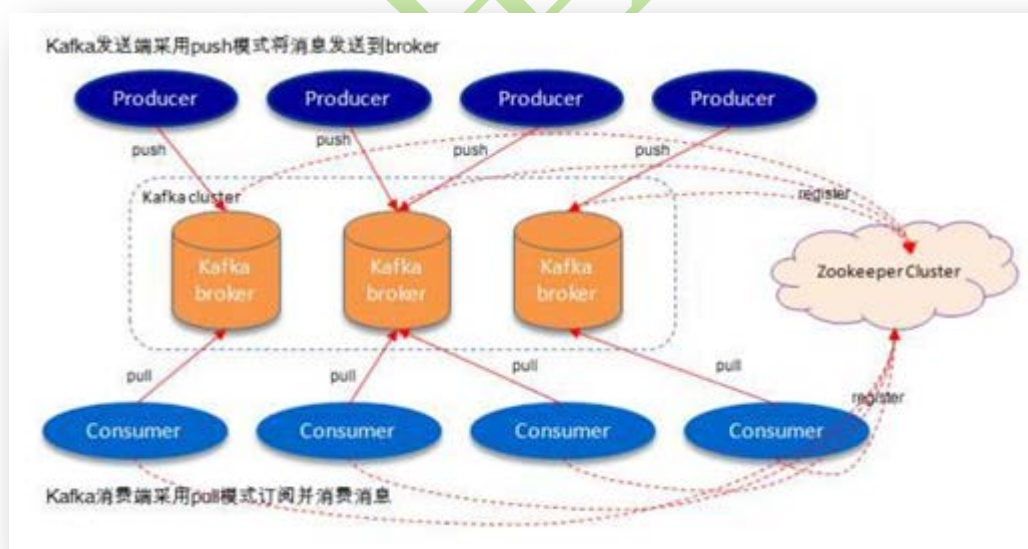
## 第1章 Kafka 概述

### 1.1 kafaka 简介

Apache Kafka 是一个快速、可扩展的、高吞吐的、可容错的分布式“发布-订阅”消息系统，使用 Scala 与 Java 语言编写，能够将消息从一个端点传递到另一个端点，较之传统的消息中间件（例如 ActiveMQ、RabbitMQ），Kafka 具有高吞吐量、内置分区、支持消息副本和高容错的特性，非常适合大规模消息处理应用程序。

Kafka 官网：<http://kafka.apache.org/>

### 1.2 Kafa 系统架构



### 1.3 应用场景

Kafka 的应用场景很多，这里就举几个最常见的场景。

#### 1.3.1 用户的活动追踪

用户在网站的不同活动消息发布到不同的主题中心，然后可以对这些消息进行实时监测

实时处理。当然，也可加载到 Hadoop 或离线处理数据仓库，对用户进行画像。像淘宝、京东这些大型的电商平台，用户的所有活动都是要进行追踪的。

### 1.3.2 日志聚合



### 1.3.3 限流削峰



## 1.4 kafka 高吞吐率实现

Kafka 与其它 MQ 相比，其最大的特点就是高吞吐率。为了增加存储能力，Kafka 将所有的消息都写入到了**低速大容量的硬盘**。按理说，这将导致性能损失，但实际上，kafka 仍可保持超高的吞吐率，性能并未受到影响。其主要采用了如下的方式实现了高吞吐率。

- 顺序读写：Kafka 将消息写入到了分区 partition 中，而分区中消息是顺序读写的。顺序读写要远快于随机读写。
- 零拷贝：生产者、消费者对于 kafka 中消息的操作是采用零拷贝实现的。
- 批量发送：Kafka 允许使用批量消息发送模式。
- 消息压缩：Kafka 支持对消息集合进行压缩。

## 第2章 Kafka 工作原理与工作过程

### 2.1 Kafka 基本术语

对于 Kafka 基本原理的介绍，可以通过对以下基本术语的介绍进行。

#### 2.1.1 Topic

主题。在 Kafka 中，使用一个类别属性来划分消息的所属类，划分消息的这个类称为 topic。topic 相当于消息的分类标签，是一个逻辑概念。

#### 2.1.2 Partition

分区。topic 中的消息被分割为一个或多个 partition，其是一个物理概念，对应到系统上就是一个或若干个目录。

若一个 Topic 包含多个 partition，消息在写入到 partition 时是无法保证其写入顺序与生产顺序的一致性的。若需要严格保证这个一致性，则 partition 数量设置为 1。

若要使消费者严格按照进入到 kafka 中的消息顺序进行消费，则要使 partition 设置为 1。

#### 2.1.3 segment

段。将 partition 进一步细分为若干的 segment，每个 segment 文件的最大大小相等。

#### 2.1.4 Broker

Kafka 集群包含一个或多个服务器，每个服务器节点称为一个 broker。

假设某个 topic 中有 N 个 partition，集群中有 M 个 broker，则 broker 与 partition 的数量关系是：

- 若  $N \geq M$ ，且  $N \% M = 0$ ，每个 broker 上会平均分配  $N/M$  个 partition
- 若  $N > M$ ，但  $N \% M \neq 0$ ，此时会出现各个 broker 上 partition 数量不平均的情况，此时各个 broker 的消息负载是不均衡的。该情况要避免。
- 若  $N < M$ ，此时会出现某些 broker 上没有 partition，此时各个 broker 的消息负载是不均衡的。该情况要避免。

#### 2.1.5 Producer

生产者。即消息的发布者，其会将某 topic 的消息发布到相应的 partition 中。

## 2.1.6 Consumer

消费者。可以从 broker 中读取消息。

对于消费者需要注意：

- 一个消费者可以消费多个 topic 的消息
- 一个消费者也可以消费同一个 topic 中的多个 partition 中的消息
- 反过来说也成立，即一个 partition 允许多个无关的消费者（不同组的消费者）同时消费

## 2.1.7 Consumer Group

consumer group 是 kafka 提供的可扩展且具有容错性的消费者机制。组内可以有多个消费者，它们共享一个公共的 ID，即 group ID。组内的所有消费者会协调在一起平均消费订阅主题的所有分区。

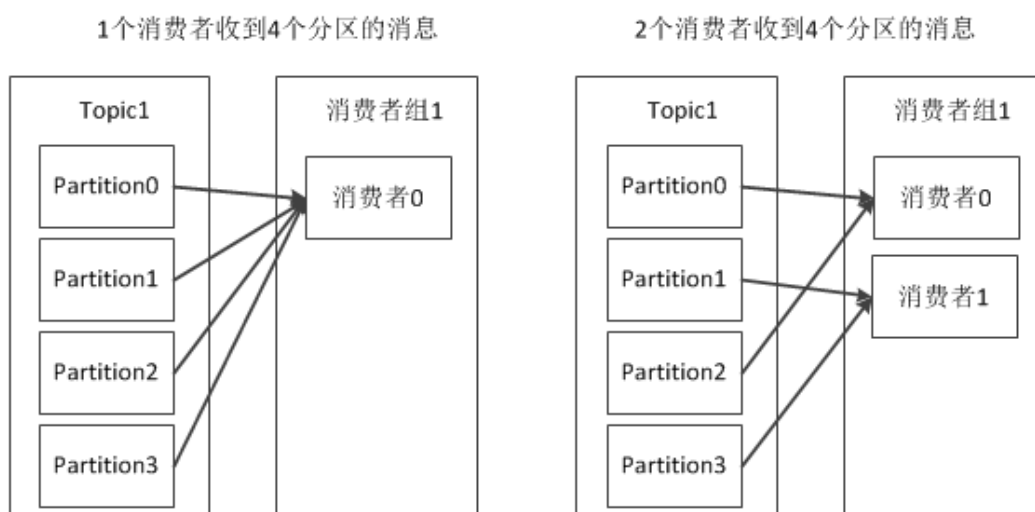
Kafka 可以保证在稳定状态下，一个 partition 中的消息只能被同一个 consumer group 中的一个 consumer 消费，而一个组内 consumer 只会消费某一个或几个特定的 partition。当然，一个消息可以同时被多个 consumer group 消费。

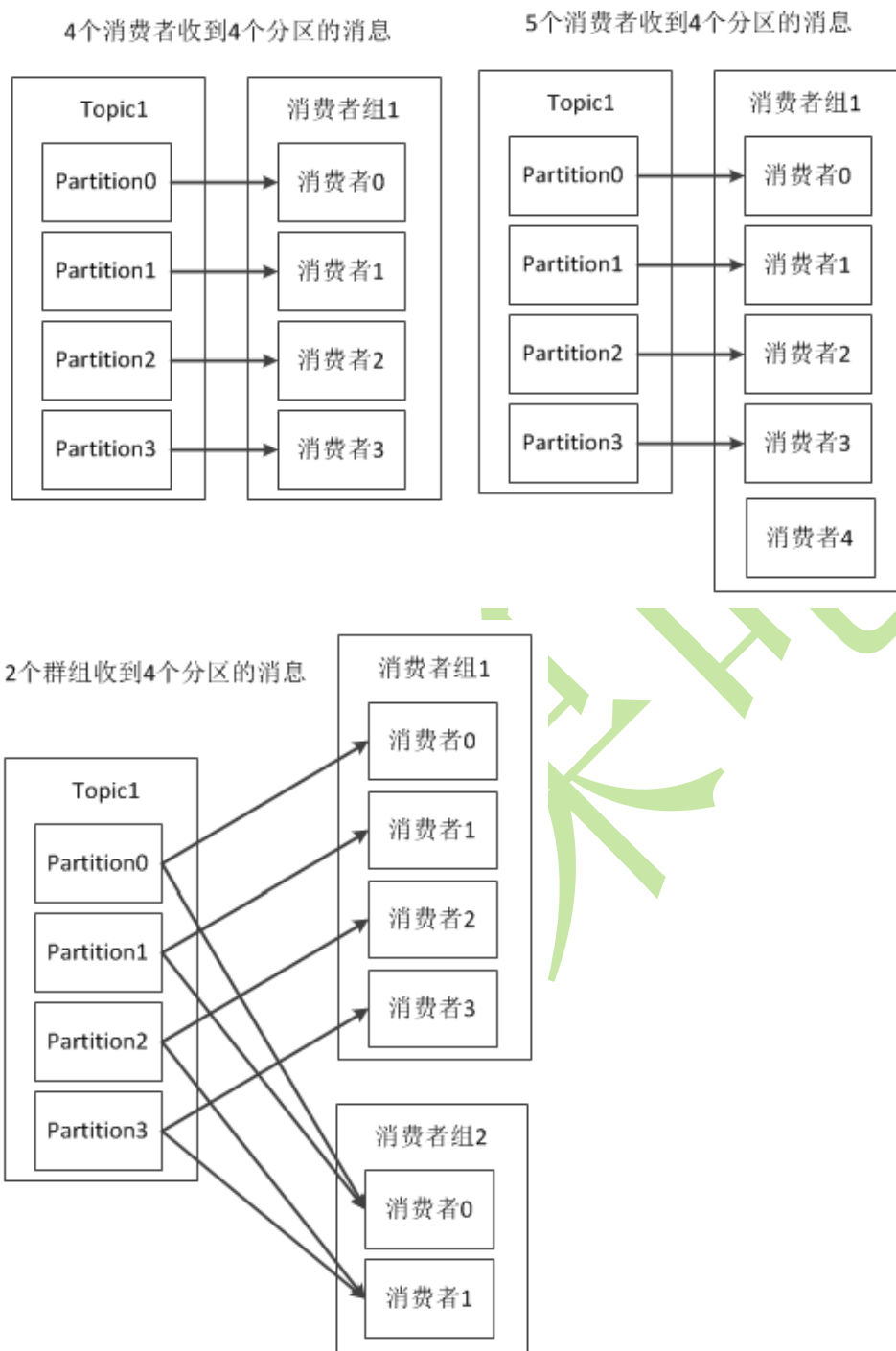
总结：

组内 consumer 与 partition 的关系是 1:n，partition 与组内 consumer 的关系则是 1:1 也就是说，在稳定状态下，一旦为某组内 consumer 分配了某一个/几个 partition 后，就不会变化了。反过来说，一旦为某 partition 分配了组内 consumer，就不会再为其分配其它组内 consumer 了。

这种设计方案最大的好处：简单。但也存在不足：组内 consumer 消息的不平均。

组中 consumer 数量与 partition 数量的对应关系如下。





## 2.1.8 Replicas of partition

分区副本。副本是一个分区的备份，是为了防止消息丢失而创建的分区的备份。

### 2.1.9 Partition Leader

每个 partition 有多个副本，其中有且仅有一个作为 Leader，Leader 是当前负责消息读写的 partition。即所有读写操作只能发生于 Leader 分区上。

broker controller 负责 partition leader 的选举。

### 2.1.10 Partition Follower

所有 Follower 都需要从 Leader 同步消息，Follower 与 Leader 始终保持消息同步。  
partition leader 与 follower 是主备关系，非主从关系。

### 2.1.11 ISR

ISR, In-Sync Replicas, 是指副本同步列表。

AR, Assigned Replicas, 指定的副本。在最初时,  $AR = ISR$ , 但在运行过程中, 若出现了 ISR 中的某些 follower 同步超时, 则 leader 会将这些 follower 从 ISR 列表中踢除, 进入到 OSR 集合。即  $AR = ISR + OSR$

OSR, Outof-Sync Replicas

### 2.1.12 offset

偏移量。每条消息都有一个当前 Partition 下唯一的 64 字节的 offset, 它是相对于当前分区第一条消息的偏移量。

### 2.1.13 offset commit

Consumer 从 partition 中取出一批消息写入到 buffer 对其进行消费, 在规定时间内消费完消息后, 会自动将其消费消息的 offset 提交给 broker, 以让 broker 记录下哪些消息是消费过的。当然, 若在时限内没有消费完毕, 其是不会提交 offset 的。

### 2.1.14 Rebalance

当消费者组中消费者数量发生变化, 或 Topic 中的 partition 数量发生了变化时, partition 的所有权会在消费者间转移, 即 partition 会重新分配, 这个过程称为再均衡 Rebalance。

再均衡能够给消费者组及 broker 集群带来高可用性和伸缩性, 但在再均衡期间消费者是无法读取消息的, 即整个 broker 集群有一小段时间是不可用的。因此要避免不必要的再均衡。

### 2.1.15 Broker Controller

Kafka 集群的多个 broker 中,有一个会被选举为 controller,负责管理整个集群中 partition 和副本 replicas 的状态。

### 2.1.16 Zookeeper

Zookeeper 负责维护和协调 broker, 负责 Broker Controller 的选举。

总结对比:

- partition leader 是由 broker controller 负责选举的, 选举算法是“论资排辈”。
- broker controller 是由 zk 负责选举的, 选举算法是我们曾在 zk 的典型应用场景中讲过的 Master 选举

### 2.1.17 Coordinator

Coordinator 一般指的是运行在每个 broker 上的 group Coordinator 进程, 用于管理 Consumer Group 中的各个成员, 主要用于 offset 位移管理和 Rebalance。一个 Coordinator 可以同时管理多个消费者组。

## 2.2 Kafka 工作原理与过程

### 2.2.1 消息路由策略

在通过 API 方式发布消息时,生产者是以 Record 为消息进行发布的。Record 中包含 key 与 value, value 才是我们真正的消息本身, 而 key 用于路由消息所要存放的 Partition。消息要写入到哪个 Partition 并不是随机的, 而是有路由策略的。

- 1) 若指定了 partition, 则直接写入到指定的 partition;
- 2) 若未指定 partition 但指定了 key, 则通过对 key 的 hash 值与 partition 数量取模, 该取模结果就是要选出的 partition 索引;
- 3) 若 partition 和 key 都未指定, 则使用轮询算法选出一个 partition。

### 2.2.2 消息写入算法

消息生产者将消息发送给 broker, 并形成最终的可供消费者消费的 log, 是一个比较复杂的过程。

- 1) producer 向 broker 集群提交连接请求, 其所连接上的任意 broker 都会向其发送 broker controller 的通信 URL, 即 broker controller 主机配置文件中的 listeners 地址
- 2) 当 producer 指定了要生产消息的 topic 后, 其会向 broker controller 发送请求, 请求当前 topic 中所有 partition 的 leader 列表地址
- 3) broker controller 在接收到请求后, 会从 zk 中查找到指定 topic 的所有 partition 的 leader,



并返回给 producer

- 4) producer 在接收到 leader 列表地址后, 根据消息路由策略找到当前要发送消息所要发送的 partition leader, 然后将消息发送给该 leader
- 5) leader 将消息写入本地 log, 并通知 ISR 中的 followers
- 6) ISR 中的 followers 从 leader 中同步消息后向 leader 发送 ACK
- 7) leader 收到所有 ISR 中的 followers 的 ACK 后, 增加 HW, 表示消费者已经可以消费到该位置了
- 8) 若 leader 在等待 followers 的 ACK 时超时了, 发现还有 follower 没有发送 ACK, 则 leader 会将这些 follower 从 ISR 中清除, 然后再增加 HW

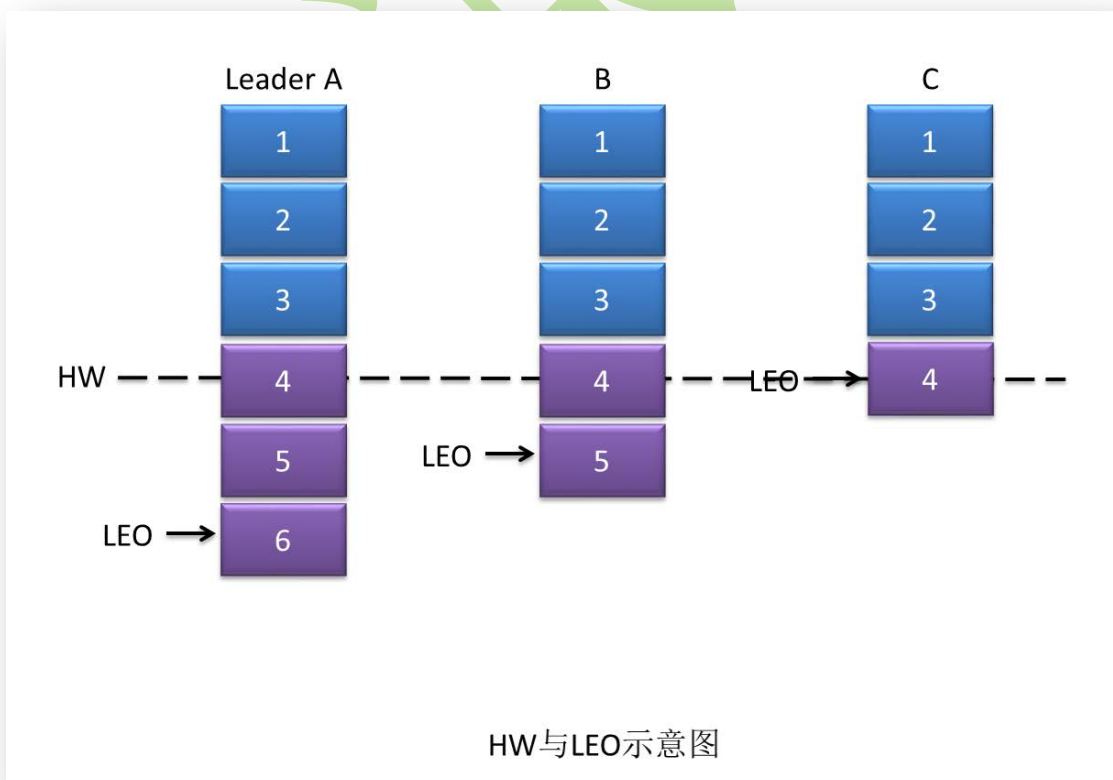
### 2.2.3 HW 机制

HW, HighWatermark, 高水位, 表示 Consumer 可以消费到的最高 partition 偏移量。HW 保证了 Kafka 集群中消息的一致性。确切地说, 是在 broker 集群正常运转的状态下, 保证了 partition 的 Follower 与 Leader 间数据的一致性 (HW 机制的作用)。

在 HW 机制中还有一个很重要的概念, LEO。

LEO, Log End Offset, 日志最后消息的偏移量。消息是被写入到 Kafka 的日志文件中的, 这是当前最后一个写入的消息在 Partition 中的偏移量。

对于 leader 新写入的消息, consumer 是不能立刻消费的。leader 会等待该消息被所有 ISR 中的 partition follower 同步后才会更新 HW, 此时消息才能被 consumer 消费。



## 2.2.4 HW 截断机制

如果 partition leader 接收到了新的消息，ISR 中其它 Follower 正在同步过程中，还未同步完毕时 leader 挂了。此时就需要选举出新的 leader。若没有 HW 截断机制，将会导致 partition 中 leader 与 follower 数据的不一致。

当原 Leader 宕机后又恢复时，将其 LEO 回退到其宕机时的 HW，然后再与新的 Leader 进行数据同步，这种机制称为 HW 截断机制。

注意，这个过程中是有可能引发消息丢失的。

总结对比：

- HW 机制：是为了保证在 broker 集群正常运转状态下的 partition leader 与 follower 的数据的一致性。
- HW 截断机制：是为了保证在 broker 集群中 partition leader 出现宕机情况下，partition leader 与 follower 的数据的一致性。

## 2.2.5 消息发送的可靠性机制

生产者向 kafka 发送消息时，可以选择需要的可靠性级别。通过 acks 参数的值进行设置。

### (1) 0 值

异步发送。生产者向 kafka 发送消息而不需要 kafka 反馈成功 ack。该方式效率最高，但可靠性最低。其可能会存在消息丢失的情况。

### (2) 1 值

默认值。同步发送，默认值。生产者发送消息给 kafka，broker 的 partition leader 在收到消息后马上发送成功 ack（无需等待 ISR 中的 follower 同步完成），生产者收到后知道消息发送成功，然后会再发送消息。如果一直未收到 kafka 的 ack，则生产者会认为消息发送失败，会重发消息。其同样存在消息丢失的可能性。

- 问题 1：该方法能否使 producer 确认其发送的消息发送成功（该消息写入到了 broker）？不能。
- 问题 2：该方法能否使 producer 确认其发送的消息发送失败了？能。只要 producer 没有收到 ack，一定是发送失败了。
- 简单说就是，收到 ack 了，不一定成功，但没有收到 ack，一定失败

### (3) -1 值

同步发送。其值等同于 all。生产者发送消息给 kafka，kafka 收到消息后要等到 ISR 列表

中的所有副本都同步消息完成后，才向生产者发送成功 **ack**。如果一直未收到 kafka 的 **ack**，则认为消息发送失败，会自动重发消息。

该方式存在 **follower** 重复接收的情况。注意，重复接收，与重复消费是两个概念。

## 2.2.6 消费者消费过程解析

生产者将消息发送到 **topic** 中，消费者即可对其进行消费，其消费过程如下：

- 1) **consumer** 向 **broker** 集群提交连接请求，其所连接上的任意 **broker** 都会向其发送 **broker controller** 的通信 URL，即 **broker controller** 主机配置文件中的 **listeners** 地址
- 2) 当 **consumer** 指定了要消费的 **topic** 后，其会向 **broker controller** 发送 **poll** 请求
- 3) **broker controller** 会为 **consumer** 分配一个或几个 **partition leader**，并将该 **partition** 的当前 **offset** 发送给 **consumer**
- 4) **consumer** 会按照 **broker controller** 分配的 **partition** 对其中的消息进行消费
- 5) 当消费者消费完该条消息后，消费者会向 **broker** 发送一个该消息已被消费的反馈，即该消息的 **offset**

- 若为手动提交：可以是消费完一条消息就提交一个 **offset**，也可以是消费完这一批消息后，提交最后一个消息的 **offset**。关键看代码怎么写。
- 若为自动提交：提交最后一个消息的 **offset**。

- 6) 当 **broker** 接到消费者的 **offset** 后，会更新到相应的 **\_\_consumer\_offset** 中
- 7) 以上过程一直重复，直到消费者停止请求消息
- 8) 消费者可以重置 **offset**，从而可以灵活消费存储在 **broker** 上的消息

## 2.2.7 Partition Leader 选举范围

当 **leader** 挂了后 **broker controller** 会从 **ISR** 中选一个 **follower** 成为新的 **leader**。但，若 **ISR** 中的所有副本都挂了怎么办？可以通过 **unclean.leader.election.enable** 的取值来设置 **Leader** 选举的范围。

### (1) false

必须等待 **ISR** 列表中有副本活过来才进行新的选举。该策略可靠性有保证，但可用性低。

### (2) true

在 **ISR** 中没有副本的情况下可以选择任何一个没有宕机主机中该 **topic** 的 **partition** 副本（即 **OSR** 中的 **partition** 副本）作为新的 **leader**，该策略可用性高，但可靠性没有保证。

## 2.2.8 重复消费问题及解决方案

以下讨论的重复消费问题都是基于 **offset** 自动提交的。

最常见的重复消费有两种：

### (1) 同一个 consumer 重复消费

当 Consumer 由于消费能力较低而引发了消费超时，则可能会形成重复消费。

解决方案：

- 减少消费量
- 延长自动提交的时长
- 自动提交变为手动提交

### (2) 不同的 consumer 重复消费

当 Consumer 消费了消息但还未提交 offset 时宕机，则这些已被消费过的消息会被重复消费。

解决方案：

- 自动提交变为手动提交

## 2.3 Kafka 集群搭建

在生产环境中为了防止单点问题，Kafka 都是以集群方式出现的。下面要搭建一个 Kafka 集群，包含三个 Kafka 主机，即三个 Broker。

## 2.3.1 Kafka 的下载

# Download

2.2.0 is the latest release. The current stable version is 2.2.0.

You can verify your download by following these [procedures](#) and using these [KEYS](#).

## 2.2.0

- Released Mar 22, 2019
- [Release Notes](#)
- Source download: [kafka-2.2.0-src.tgz](#) ([asc](#), [sha512](#))
- Binary downloads:
  - Scala 2.11 - [kafka\\_2.11-2.2.0.tgz](#) ([asc](#), [sha512](#))
  - Scala 2.12 - [kafka\\_2.12-2.2.0.tgz](#) ([asc](#), [sha512](#))

## 2.3.2 安装并配置第一台主机

### (1) 上传并解压

将下载好的 Kafka 压缩包上传至 CentOS 虚拟机，并解压。

```
✓ kafkaos1 [x]
[root@kafkaos1 ~]# ll
总用量 232576
-rw-r--r--. 1 root root 174157387 1月 18 2018 jdk-8u161-linux-x64.rpm
-rw-r--r-- 1 root root 63999924 3月 23 08:57 kafka_2.11-2.2.0.tgz
[root@kafkaos1 ~]#
[root@kafkaos1 ~]# tar -zxvf kafka_2.11-2.2.0.tgz -C /opt/apps
```

## (2) 创建软链接

```
✓ kafkaos1 [root@kafka0S1 apps]# ln -s kafka_2.11-2.2.0/ kafka
[root@kafka0S1 apps]# ll
总用量 0
lrwxrwxrwx 1 root root 17 4月  1 17:34 kafka -> kafka_2.11-2.2.0/
drwxr-xr-x 6 root root 89 3月 10 03:47 kafka_2.11-2.2.0
[root@kafka0S1 apps]#
```

## (3) 修改配置文件

在 kafka 安装目录下有一个 config/server.properties 文件，修改该文件。

```
✓ kafkaos1 17
18 ##### Server Basics #####
19
20 # The id of the broker. This must be set to a unique int
21 broker.id=0
22
23 ##### Socket Server Settings #####
24
25 # The address the socket server listens on. It will get
26 # java.net.InetAddress.getCanonicalHostName() if not co
27 #   FORMAT:
28 #     listeners = listener_name://host_name:port
29 #   EXAMPLE:
30 #     listeners = PLAINTEXT://your_host_name:9092
31 listeners=PLAINTEXT://192.168.59.151:9092
32
33 # Hostname and port the broker will advertise to produce
34 # it uses the value for "listeners" if configured. Other
35 # returned from java.net.InetAddress.getCanonicalHostNa
36 #advertised.listeners=PLAINTEXT://your_host_name:9092
37
38 # Maps listener names to security protocols, the default
```

```

56
57 ##### Log Basics #####
58
59 # A comma separated list of directories under wh
60 log.dirs=/tmp/kafka-logs
61
62 # The default number of log partitions per topic
63 # parallelism for consumption, but this will als
64 # the brokers.
65 num.partitions=1
66
67 # The number of threads per data directory to be
68 # flushing at shutdown.
69 # This value is recommended to be increased for
70 # n RAID array.
71 num.recovery.threads.per.data.dir=1

```

```

116 ##### Zookeeper #####
117
118 # Zookeeper connection string (see zookeeper docs for details).
119 # This is a comma separated host:port pairs, each corresponding
120 # server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
121 # You can also append an optional chroot string to the urls to s
122 # root directory for all kafka znodes.
123 zookeeper.connect=192.168.59.117:2181
124
125 # Timeout in ms for connecting to zookeeper
126 zookeeper.connection.timeout.ms=6000
127

```

### 2.3.3 再克隆两台 Kafka

以 kafkaOS1 为母机再克隆两台 Kafka 主机。在克隆完毕后，需要修改 server.properties 中的 broker.id、listeners 与 advertised.listeners。

```
✓ kafkaos1 ✓ kafkaos2 ✕
17
18 ##### Server Basics #####
19
20 # The id of the broker. This must be set to a
21 broker.id=1
22
23 ##### Socket Server Se
24
25 # The address the socket server listens on. It
26 # java.net.InetAddress.getCanonicalHostName()
27 #   FORMAT:
28 #     listeners = listener_name://host_name:po
29 #   EXAMPLE:
30 #     listeners = PLAINTEXT://your.host.name:9
31 listeners=PLAINTEXT://192.168.59.152:9092
32
33 # Hostname and port the broker will advertise
34 # it uses the value for "listeners" if configu
```

```
✓ kafkaos1 ✓ kafkaos2 ✓ kafkaos3 ✕
17
18 ##### Server Basics #####
19
20 # The id of the broker. This must be set to a unique
21 broker.id=2
22
23 ##### Socket Server Setting
24
25 # The address the socket server listens on. It will
26 # java.net.InetAddress.getCanonicalHostName() if no
27 #   FORMAT:
28 #     listeners = listener_name://host_name:port
29 #   EXAMPLE:
30 #     listeners = PLAINTEXT://your.host.name:9092
31 listeners=PLAINTEXT://192.168.59.153:9092
32
33 # Hostname and port the broker will advertise to pr
34 # it uses the value for "listeners" if configu
```



## 2.3.4 kafka 的启动与停止

### (1) 启动 zookeeper

```
✓ zkOS × ✓ kafkaos1 ✓ kafkaos2 ✓ kafkaos3
[root@zkOS ~]# zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /opt/apps/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@zkOS ~]#
```

### (2) 启动 kafka

在命令后添加-daemon 参数，可以使 kafka 以守护进程方式启动，即不占用窗口。

```
kafka]# bin/kafka-server-start.sh -daemon config/server.properties
kafka]#
```

### (3) 停止 kafka

```
✓ zkOS ✓ kafkaos ×
[root@kafkaos kafka]# bin/kafka-server-stop.sh
[root@kafkaos kafka]#
```

## 2.3.5 kafka 操作

### (1) 创建 topic

```
✓ zkos ✓ kafkaos1 ✗ kafkaos2 ✓ kafkaos3
[root@kafka0S1 kafka]# bin/kafka-topics.sh --create --bootstrap-server
192.168.59.151:9092 --replication-factor 1 --partitions 1 --topic test
[root@kafka0S1 kafka]#
```

### (2) 查看 topic

```
✓ zkos ✓ kafkaos1 ✗ kafkaos2 ✓ kafkaos3
[root@kafka0S1 kafka]# bin/kafka-topics.sh --list --bootstrap-server
192.168.59.151:9092
__consumer_offsets
test
[root@kafka0S1 kafka]#
```

### (3) 发送消息

该命令会创建一个生产者，然后由其生产消息。

```
✓ zkos ✓ kafkaos1 ✗ kafkaos2 ✓ kafkaos3
[root@kafka0S1 kafka]# bin/kafka-console-producer.sh --broker-list
192.168.59.151:9092 --topic test
>beijing
>shanghai
>guangzhou
>
```

## (4) 消费消息

```
zcos kafkaos1 kafkaos2 kafkaos3
[root@kafkaos2 kafka]# bin/kafka-console-consumer.sh --bootstrap-
server 192.168.59.151:9092 --topic test --from-beginning
beijing
shanghai
guangzhou
```

## (5) 继续生产消费

```
zcos kafkaos1 kafkaos2 kafkaos3
[root@kafkaos1 kafka]# bin/kafka-console-producer.sh --broker-list
192.168.59.151:9092 --topic test
>beijing
>shanghai
>guangzhou
>shenzhen
>
```

```
zcos kafkaos1 kafkaos2 kafkaos3
[root@kafkaos2 kafka]# bin/kafka-console-consumer.sh --bootstrap-
server 192.168.59.151:9092 --topic test --from-beginning
beijing
shanghai
guangzhou
shenzhen
```

## (6) 删除 topic

```
✓ zkos ✓ kafkaos1 ✗ kafkaos2 ✓ kafkaos3
[root@kafka0S1 kafka]# bin/kafka-topics.sh --list --bootstrap-se
rver 192.168.59.151:9092
__consumer_offsets
test
[root@kafka0S1 kafka]#
[root@kafka0S1 kafka]# bin/kafka-topics.sh --delete --bootstrap-
server 192.168.59.151:9092 --topic test
[root@kafka0S1 kafka]#
[root@kafka0S1 kafka]# bin/kafka-topics.sh --list --bootstrap-se
rver 192.168.59.151:9092
__consumer_offsets
[root@kafka0S1 kafka]#
```

## 2.4 日志查看

我们这里说的日志不是 Kafka 的启动日志, 启动日志在 Kafka 安装目录下的 logs/server.log 中。消息在磁盘上都是以日志的形式保存的。我们这里说的日志是存放在/tmp/kafka\_logs 目录中的消息日志, 即 partition 与 segment。

### 2.4.1 查看分区与备份

#### (1) 1 个分区 1 个备份

我们前面创建的 test 主题是 1 个分区 1 个备份。

```

[root@kafka0s3 kafka-logs]# ll
总用量 16
-rw-r--r-- 1 root root 0 1月 1 10:48 cleaner-offset-checkpoint
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-11
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-14
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-17
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-2
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-20
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-23
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-26
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-29
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-32
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-35
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-38
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-41
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-44
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-47
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-5
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-8
-rw-r--r-- 1 root root 4 1月 1 18:13 log-start-offset-checkpoint
-rw-r--r-- 1 root root 54 1月 1 10:48 meta.properties
-rw-r--r-- 1 root root 395 1月 1 18:13 recovery-point-offset-checkpoint
-rw-r--r-- 1 root root 395 1月 1 18:13 replication-offset-checkpoint
drwxr-xr-x 2 root root 141 1月 1 10:51 test-0
[root@kafka0s3 kafka-logs]#

```

## (2) 3 个分区 1 个备份

再次创建一个主题，命名为 one，创建三个分区，但仍为一个备份。依次查看三台 broker，可以看到每台 broker 中都有一个 one 主题的分区。

```

[root@kafka0s1 kafka-logs]# ll
总用量 16
-rw-r--r-- 1 root root 0 1月 1 10:48 cleaner-offset-checkpoint
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-0
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-12
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-15
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-18
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-21
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-24
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-27
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-3
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-30
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-33
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-36
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-39
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-42
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-45
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-48
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-6
drwxr-xr-x 2 root root 141 1月 1 10:53 __consumer_offsets-9
-rw-r--r-- 1 root root 4 1月 1 18:29 log-start-offset-checkpoint
-rw-r--r-- 1 root root 54 1月 1 10:48 meta.properties
drwxr-xr-x 2 root root 141 1月 1 18:28 one-1
-rw-r--r-- 1 root root 417 1月 1 18:29 recovery-point-offset-checkpoint
-rw-r--r-- 1 root root 417 1月 1 18:29 replication-offset-checkpoint
[root@kafka0s1 kafka-logs]#

```

### (3) 3 个分区 3 个备份

再次创建一个主题，命名为 **two**，创建三个分区，三个备份。依次查看三台 broker，可以看到每台 broker 中都有三份 **two** 主题的分区。

```

[root@kafkaos1 kafka-logs]# ls
cleaner-offset-checkpoint  __consumer_offsets-30  log-start-of
__consumer_offsets-0      __consumer_offsets-33  meta.properties
__consumer_offsets-12     __consumer_offsets-36  one-1
__consumer_offsets-15     __consumer_offsets-39  recovery-point
__consumer_offsets-18     __consumer_offsets-42  replication-
__consumer_offsets-21     __consumer_offsets-45  two-0
__consumer_offsets-24     __consumer_offsets-48  two-1
__consumer_offsets-27     __consumer_offsets-6   two-2
__consumer_offsets-3      __consumer_offsets-9
[root@kafkaos1 kafka-logs]#
  
```

## 2.4.2 查看分区与备份在 zk 中的信息

使用 `zkCli.sh` 命令连接上 zk，可以查看到 kafka 在 zk 的信息。

### (1) /brokers 目录

```

[zk: localhost:2181(CONNECTED) 2] ls /brokers
[ids, topics, seqid]
[zk: localhost:2181(CONNECTED) 3]
  
```

### (2) /brokers/ids 目录

存放的是 kafka 集群中各个主机的 broker-id 列表。

```

✓ zkos x ✓ kafkaos1 ✓ kafkaos1 (1) ✓ kafkaos2 ✓ kafkaos2 (1) ✓ kafkaos3 ✓ kafkaos3 (1)
[zk: localhost:2181(CONNECTED) 9] ls /brokers/ids
[0, 1, 2]
[zk: localhost:2181(CONNECTED) 10]

```

每个 id 的数据内容为当前主机的信息。

```

zkos - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+R>
x ✓ zkos x kafkaos1 kafkaos1 (1) ✓ kafkaos2 kafkaos2 (1) kafkaos3 kafkaos3 (1)
>> [zk: localhost:2181(CONNECTED) 13] get /brokers/ids/0
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},
"endpoints":["PLAINTEXT://192.168.59.151:9092"],"jmx_port":
-1,"host":"192.168.59.151","timestamp":"1558792160008",
"port":9092,"version":4}
cZxid = 0x1f1
ctime = Sat May 25 21:49:20 CST 2019
mZxid = 0x1f1
mtime = Sat May 25 21:49:20 CST 2019
pZxid = 0x1f1
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x100000e3f470004
dataLength = 198
numChildren = 0
[zk: localhost:2181(CONNECTED) 14]

```

### (3) /brokers/topics

```

✓ zkos x ✓ kafkaos1 ✓ kafkaos1 (1) ✓ kafkaos2 ✓ kafkaos2 (1) ✓ kafkaos3 ✓ kafkaos3 (1)
[zk: localhost:2181(CONNECTED) 15] ls /brokers/topics
[country, test, city, __consumer_offsets]
[zk: localhost:2181(CONNECTED) 16]

```

/brokers/topics/city/partitions 中存放的是 city 主题下所包含的 partition。这里的 0、1、2，在/tmp/kafka-logs 目录中即为 city-0，city-1，city-2。

```

[zk: localhost:2181(CONNECTED) 18] ls /brokers/topics/city/partitions
[0, 1, 2]
[zk: localhost:2181(CONNECTED) 19]

```

```

[zk: localhost:2181(CONNECTED) 24] get /brokers/topics/city/partitions/0/state
{"controller_epoch":7,"leader":2,"version":1,"leader_epoch":0,"isr":[2,1]}
cZxid = 0x20d
ctime = Sat May 25 22:00:40 CST 2019
mZxid = 0x20d
mtime = Sat May 25 22:00:40 CST 2019
pZxid = 0x20d
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 74
numChildren = 0
[zk: localhost:2181(CONNECTED) 25]

```

## 2.4.3 查看段 segment

### (1) segment 文件

segment 是一个逻辑概念，其由两类物理文件组成，分别为“.index”文件和“.log”文件。“log”文件中存放的是消息，而“.index”文件中存放的是“.log”文件中消息的索引。





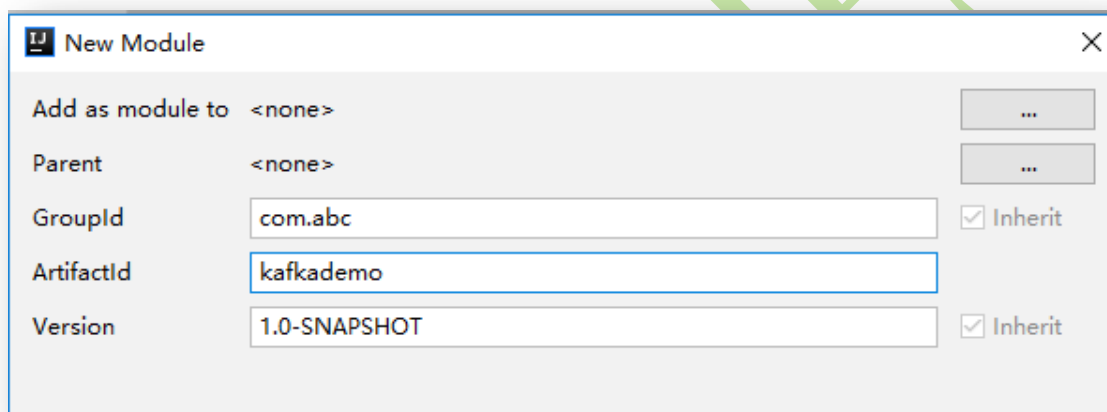
## 第3章 Kafka API

首先在命令行创建一个名称为 `cities` 的主题，并创建该主题的订阅者。

### 3.1 使用 kafka 原生 API

#### 3.1.1 创建工程

创建一个 Maven 的 Java 工程，命名为 `kafkaDemo`。创建时无需导入依赖。为了简单，后面的发布者与消费者均创建在该工程中。



#### 3.1.2 导入依赖

```
<!-- kafka 依赖 -->
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.12</artifactId>
  <version>1.1.1</version>
</dependency>
```

### 3.1.3 创建发布者 OneProducer

#### (1) 创建发布者类 OneProducer

```
public class OneProducer {  
    // 第一个泛型为key的类型，第二个泛型为消息本身的类型  
    private KafkaProducer<Integer, String> producer;  
  
    public OneProducer() {  
        Properties properties = new Properties();  
        properties.put("bootstrap.servers",  
            "kafka0S1:9092,kafka0S2:9092,kafka0S3:9092");  
        properties.put("key.serializer",  
            "org.apache.kafka.common.serialization.IntegerSerializer");  
        properties.put("value.serializer",  
            "org.apache.kafka.common.serialization.StringSerializer");  
  
        this.producer = new KafkaProducer<Integer, String>(properties);  
    }  
}
```

```
public void sendMsg() {  
    // 创建记录（消息）  
    // 指定主题及消息本身  
    // ProducerRecord<Integer, String> record =  
    //     new ProducerRecord<>("cities", "shanghai");  
    // 指定主题、key，及消息本身  
    // ProducerRecord<Integer, String> record =  
    //     new ProducerRecord<>("cities", 1, "shanghai");  
    // 指定主题、要写入的partition、key，及消息本身  
    ProducerRecord<Integer, String> record =  
        new ProducerRecord<>("cities", 0, 1, "shanghai");  
  
    // 发布消息，其返回值为Future对象，表示其发送过程为异步，不过这里不使用该返回结果  
    // Future<RecordMetadata> future = producer.send(record);  
    producer.send(record);  
}
```

## (2) 创建测试类 OneProducerTest

```
public class OneProducerTest {  
  
    public static void main(String[] args) throws IOException {  
        OneProducer producer = new OneProducer();  
        producer.sendMsg();  
        System.in.read();  
    }  
}
```

### 3.1.4 创建发布者 TwoProducer

前面的方式在消息发送成功后，代码中没有任何提示，这里可以使用回调方式，即发送成功后，会触发回调方法的执行。

#### (1) 创建发布者类 TwoProducer

复制 OneProducer 类，仅修改 sendMsg() 方法。

```
// 发布消息，其返回值为Future对象，表示其发送过程为异步，不过这里不使用该返回结果  
// Future<RecordMetadata> future = producer.send(record);  
// producer.send(record);  
  
// 可以调用以下两个参数的send()方法，可以在消息发布成功后触发回调的执行  
producer.send(record, new Callback() {  
    // RecordMetadata, 消息元数据，即主题、消息的key、消息本身等的封装对象  
    @Override  
    public void onCompletion(RecordMetadata metadata, Exception exception) {  
        System.out.print("partition = " + metadata.partition());  
        System.out.print(", topic = " + metadata.topic());  
        System.out.println(", offset = " + metadata.offset());  
    }  
});
```

## (2) 创建测试类 TwoProducerTest

```
public class TwoProducerTest {

    public static void main(String[] args) throws IOException {
        TwoProducer producer = new TwoProducer();
        producer.sendMsg();
        System.in.read();
    }
}
```

### 3.1.5 批量发送消息

#### (1) 创建发布者类 SomeProducerBatch

复制前面的发布者类，在其基础上进行修改。

```
public class SomeProducerBatch {
    // 第一个泛型为key的类型，第二个泛型为消息本身的类型
    private KafkaProducer<Integer, String> producer;

    public SomeProducerBatch() {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", "kafka0S1:9092,kafka0S2:9092");
        properties.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
        properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        properties.put("batch.size", 16384); // 16K
        properties.put("linger.ms", 50); // 50ms
        this.producer = new KafkaProducer<Integer, String>(properties);
    }
}
```

```
public void sendMsg() {
    for (int i=0; i<50; i++) {
        ProducerRecord<Integer, String> record =
            new ProducerRecord<>("cities", 0, i * 10, "city-" + i*100);

        producer.send(record, new Callback() {
            // RecordMetadata, 消息元数据, 即主题、消息的key、消息本身等的封装对象
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                System.out.print("partition = " + metadata.partition());
                System.out.print(", topic = " + metadata.topic());
                System.out.println(", offset = " + metadata.offset());
            }
        });
    }
}
```

## (2) 创建测试类 ProducerBatchTest

```
public class ProducerBatchTest {

    public static void main(String[] args) throws IOException {
        SomeProducerBatch producer = new SomeProducerBatch();
        producer.sendMsg();
        System.in.read();
    }
}
```

### 3.1.6 消费者组

#### (1) 创建消费者类 SomeConsumer

```
public class SomeConsumer extends ShutdownableThread {
    private KafkaConsumer<Integer, String> consumer;

    public SomeConsumer() {
        super("KafkaConsumerTest", false);

        Properties properties = new Properties();
        String brokers = "kafka0S1:9092,kafka0S2:9092,kafka0S3:9092";
        properties.put("bootstrap.servers", brokers);
        properties.put("group.id", "cityGroup1");
        properties.put("enable.auto.commit", "true");
        properties.put("max.poll.records", "500");
        properties.put("auto.commit.interval.ms", "1000");
        properties.put("session.timeout.ms", "30000");
        properties.put("heartbeat.interval.ms", "10000");
        properties.put("auto.offset.reset", "earliest");
        properties.put("key.deserializer",
            "org.apache.kafka.common.serialization.IntegerDeserializer");
        properties.put("value.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");

        this.consumer = new KafkaConsumer<Integer, String>(properties);
    }
}
```

```

@Override
public void doWork() {
    // 指定要消费的主题
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.print("topic = " + record.topic());
        System.out.print(" partition = " + record.partition());
        System.out.print(" key = " + record.key());
        System.out.println(" value = " + record.value());
    }
}
}

```

## (2) 创建测试类 ConsumerTest

```

public class ConsumerTest {
    public static void main(String[] args) {
        SomeConsumer consumer = new SomeConsumer();
        consumer.start();
    }
}

```

### 3.1.7 消费者同步手动提交

#### (1) 自动提交的问题

前面的消费者都是以自动提交 offset 的方式对 broker 中的消息进行消费的,但自动提交可能会出现消息重复消费的情况。所以在生产环境下,很多时候需要对 offset 进行手动提交,以解决重复消费的问题。



## (2) 手动提交分类

手动提交又可以划分为同步提交、异步提交、同异步联合提交。这些提交方式仅仅是 doWork() 方法不相同，其构造器是相同的。所以下面首先在前面消费者类的基础上进行构造器的修改，然后再分别实现三种不同的提交方式。

## (3) 创建消费者类 SyncManualConsumer

### A、原理

同步提交方式是，消费者向 broker 提交 offset 后等待 broker 成功响应。若没有收到响应，则会重新提交，直到获取到响应。而在这个等待过程中，消费者是阻塞的。其严重影响了消费者的吞吐量。

### B、修改构造器

直接复制前面的 SomeConsumer，在其基础上进行修改。

```
public class SyncManualConsumer extends ShutdownableThread {
    private KafkaConsumer<Integer, String> consumer;

    public SyncManualConsumer() {
        super("KafkaConsumerTest", false);

        Properties properties = new Properties();
        String brokers = "kafka0S1:9092,kafka0S2:9092,kafka0S3:9092";
        properties.put("bootstrap.servers", brokers);
        properties.put("group.id", "cityGroup1");

        properties.put("enable.auto.commit", "false");
        // properties.put("auto.commit.interval.ms", "1000");
    }
}
```

## C、修改 doWork()方法

```
@Override
public void doWork() {
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.println("topic = " + record.topic());
        System.out.println("partition = " + record.partition());
        System.out.println("key = " + record.key());
        System.out.println("value = " + record.value());
    }
    // 手动同步提交
    consumer.commitSync();
}
```

## (4) 创建测试类 SyncManualTest

```
public class SyncManualTest {
    public static void main(String[] args) {
        SyncManualConsumer consumer = new SyncManualConsumer();
        consumer.start();
    }
}
```

### 3.1.8 消费者异步手动提交

#### (1) 原理

手动同步提交方式需要等待 broker 的成功响应，效率太低，影响消费者的吞吐量。异步提交方式是，消费者向 broker 提交 offset 后不用等待成功响应，所以其增加了消费者的吞吐量。

## (2) 创建消费者类 AsyncManualConsumer

复制前面的 SyncManualConsumer 类，在其基础上进行修改。

```
@Override
public void doWork() {
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.println("topic = " + record.topic());
        System.out.println("partition = " + record.partition());
        System.out.println("key = " + record.key());
        System.out.println("value = " + record.value());
    }

    // 手动异步提交
    // consumer.commitSync();
    consumer.commitAsync((offsets, ex) -> {
        if(ex != null) {
            System.out.print("提交失败, offsets = " + offsets);
            System.out.println(", exception = " + ex);
        }
    });
}
```

## (3) 创建测试类 AsyncManualTest

```
public class AsyncManualTest {
    public static void main(String[] args) {
        AsyncManualConsumer consumer = new AsyncManualConsumer();
        consumer.start();
    }
}
```

### 3.1.9 消费者同异步手动提交

#### (1) 原理

同异步提交，即同步提交与异步提交组合使用。一般情况下，在异步手动提交时，若偶尔出现提交失败，其也不会影响消费者的消费。因为后续提交最终会将这次提交失败的 offset 给提交了。

但异步手动提交会产生重复消费，为了防止重复消费，可以将同步提交与异常提交联合使用。

#### (2) 创建消费者类 SyncAsyncManualConsumer

复制前面的 AsyncManualConsumer 类，在其基础上进行修改。

```
@Override
public void doWork() {
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.println("topic = " + record.topic());
        System.out.println("partition = " + record.partition());
        System.out.println("key = " + record.key());
        System.out.println("value = " + record.value());
    }
    consumer.commitAsync((offsets, ex) -> {
        if(ex != null) {
            System.out.print("提交失败, offsets = " + offsets);
            System.out.println(", exception = " + ex);

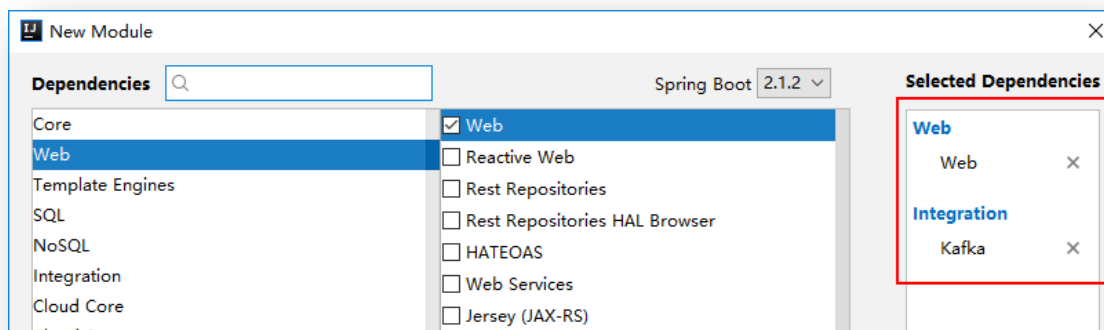
            // 同步提交
            consumer.commitSync();
        }
    });
}
```

## 3.2 Spring Boot Kafka

为了简单，以下代码是将消息发布者与订阅者定义到了一个工程中的。

### 3.2.1 创建工程

创建一个 Spring Boot 工程，导入如下依赖。



### 3.2.2 定义发布者

Spring 是通过 `KafkaTemplate` 来完成对 Kafka 的操作的。

#### (1) 修改配置文件



#### (2) 定义发布者处理器

Spring Kafka 通过 `KafkaTemplate` 完成消息的发布。

```
@RestController
public class SomeProducer {
    @Autowired
    private KafkaTemplate<String, String> template;

    // 从配置文件读取自定义属性
    @Value("${kafka.topic}")
    private String topic;

    // 由于是提交数据，所以使用Post方式
    @PostMapping("/msg/send")
    public String sendMsg(@RequestParam("message") String message) {
        template.send(topic, message);
        return "send success";
    }
}
```

### 3.2.3 定义消费者

Spring 是通过监听方式实现消费者的。

#### (1) 修改配置文件

在配置文件中添加如下内容。注意，Spring 中要求必须为消费者指定组。

```

1  # 自定义属性
2  kafka:
3    topic: cities
4
5  # 配置Kafka
6  spring:
7    kafka:
8      bootstrap-servers: kafka0S1:9092,kafka0S2:9092,kafka0S3:9092
9      producer: # 配置生产者
10       key-serializer: org.apache.kafka.common.serialization.StringSerializer
11       value-serializer: org.apache.kafka.common.serialization.StringSerializer
12
13       consumer: # 配置消费者
14       group-id: group0 # 消费者组
15       key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
16       value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
17
  
```

## (2) 定义消费者

Spring Kafka 是通过 `KafkaListener` 监听方式来完成消息订阅与接收的。当监听到有指定主题的消息时，就会触发 `@KafkaListener` 注解所标注的方法的执行。

```

@Component
public class SomeConsumer {

    @KafkaListener(topics = "${kafka.topic}")
    public void onMsg(String message) {
        System.out.println("Kafka消费者接受到消息 " + message);
    }

}
  
```