kafka笔记

定义

Kafka 是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域

kafka架构

1）Producer ：消息生产者，就是向 kafka broker 发消息的客户端；
2）Consumer ：消息消费者，向 kafka broker 取消息的客户端；
3）Consumer Group （CG）：消费者组，由多个 consumer 组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
4）Broker ：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker可以容纳多个 topic。
5）Topic ：可以理解为一个队列，生产者和消费者面向的都是一个 topic；
6）Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列；
7）Replica：副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，一个 leader 和若干个 follower。
8）leader：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 leader。
9）follower：每个分区多个副本中的“从”，实时从 leader 中同步数据，保持和 leader 数据的同步。leader 发生故障时，某个 follower 会成为新的 follower。

kafka术语

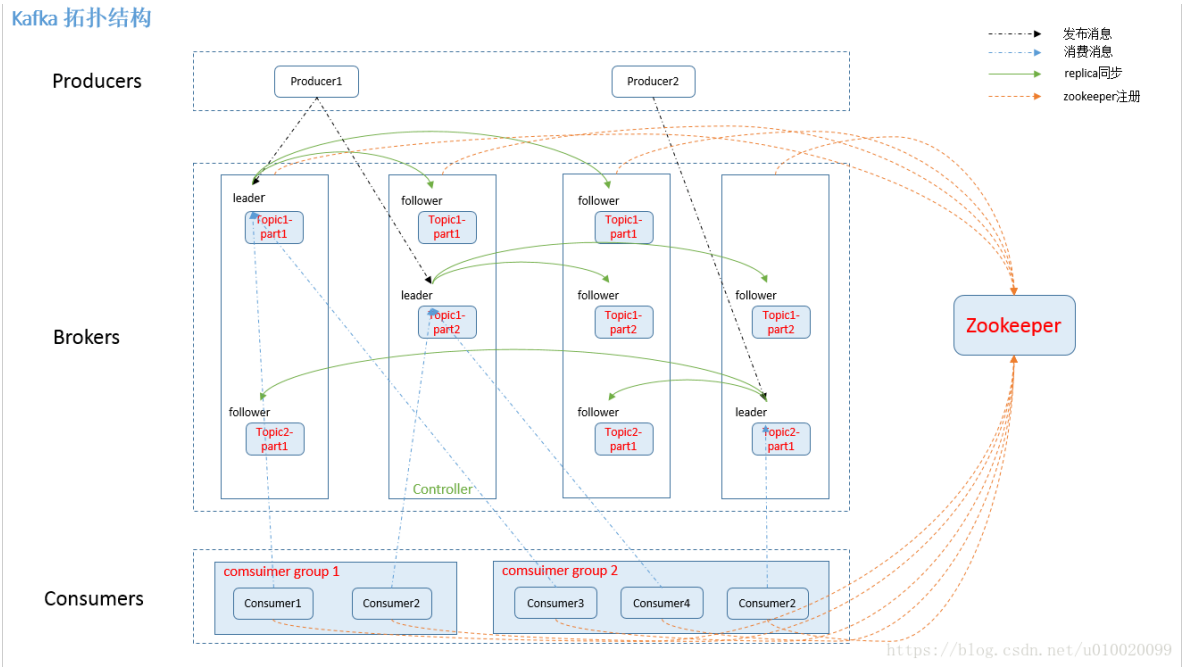
massage： kafka中最基本的传递对象，有固定格式。
topic： 一类消息，如page view，click行为等。
producer： 产生信息的主体，可以是服务器日志信息等。
consumer： 消费producer产生话题消息的主体。
broker： 消息处理结点，多个broker组成kafka集群。
partition： topic的物理分组，每个partition都是一个有序队列。
segment： 多个大小相等的段组成了一个partition。
offset： 一个连续的用于定位被追加到分区的每一个消息的序列号，最大值为64位的long大小，19位数字字符长度。
偏移量记录消息的每个消息所在分区的偏移量

kafka配置

```
#broker 的全局唯一编号，不能重复
broker.id=0
#删除 topic 功能使能
delete.topic.enable=true
#处理网络请求的线程数量
num.network.threads=3
```

```
#用来处理磁盘 IO 的现成数量
num.io.threads=8
#发送套接字的缓冲区大小
socket.send.buffer.bytes=102400
#接收套接字的缓冲区大小
socket.receive.buffer.bytes=102400
#请求套接字的缓冲区大小
socket.request.max.bytes=104857600
#kafka 运行日志存放的路径
log.dirs=/opt/module/kafka/logs
#topic 在当前 broker 上的分区个数
num.partitions=1
#用来恢复和清理 data 下数据的线程数量
num.recovery.threads.per.data.dir=1
#segment 文件保留的最长时间，超时将被删除
log.retention.hours=168
#配置连接 Zookeeper 集群地址
zookeeper.connect=huan01:2181,huan02:2181,huan03:2181
```

kafka拓扑结构



Kafka的特点

1.同时为分布和订阅提供高吞吐量。据了解，Kafka每秒可以生产约25万条消息（50MB），每秒处理55万条消息（110MB）　　这里说条数，可能不上特别准确，因为消息的大小可能不一致；
2.可进行持久化操作，将消息持久化到到磁盘，以日志的形式存储，因此可用于批量消费，例如ETL，以及实时应用程　　序。 通过将数据持久化到硬盘以及replication防止数据丢失。
3.分布式系统，易于向外拓展。所有的Producer、broker和consumer都会有多个，均为分布式。无需停机即可拓展机器。
4.消息被处理的状态是在consumer端维护，而不是由server端维护，当失败时能自动平衡。
5.支持Online和offline的场景

Kafka的核心概念

名词	解释
Producer	消息的生成者
Consumer	消息的消费者
ConsumerGroup	消费者组，可以并行消费Topic中的partition的消息
Broker	缓存代理，Kafka集群中的一台或者多台服务器统称为Broker
Topic	Kafka处理资源的消息源(feeds of messages)的不同分类
Partition	Topic物理上的分组，一个topic可以分为多个partion,每个partion是一个有序的队列。partion中每条消息都会被分配一个 有序的Id(offset)
Message	消息，是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息
Producers	消息和数据生成者，向Kafka的一个topic发布消息的 过程叫做producers
Consumers	消息和数据的消费者，订阅topic并处理其发布的消费过程叫做consumers

Producers的概念

1.消息和数据生成者，向Kafka的一个topic发布消息的过程叫做producers
2.Producer将消息发布到指定的Topic中，同时Producer也能决定将此消息归属于哪个partition；比如基于round-robin方　　式或者通过其他的一些算法等；
3.异步发送批量发送可以很有效的提高发送效率。kafka producer的异步发送模式允许进行批量发送，先将消息缓存到内　　存中，然后一次请求批量发送出去。

broker的概念

1.Broker没有副本机制，一旦broker宕机，该broker的消息将都不可用。
2.Broker不保存订阅者的状态，由订阅者自己保存。
3.无状态导致消息的删除成为难题（可能删除的消息正在被订阅），Kafka采用基于时间的SLA（服务保证），消息保存一　　定时间（通常7天）后会删除。
4.消费订阅者可以rewind back到任意位置重新进行消费，当订阅者故障时，可以选择最小的offset(id)进行重新读取消费　　消息

Message组成

1.Message消息：是通信的基本单位，每个producer可以向一个topic发布消息。
2.Kafka中的Message是以topic为基本单位组织的，不同的topic之间是相互独立的，每个topic又可以分成不同的partition每个partition储存一部分

partion中的每条Message包含以下三个属性：

offset	long
MessageSize	int32
data	messages的具体内容

Consumers的概念

1．消息和数据消费者，订阅topic并处理其发布的消息的过程叫做consumers.
2．在kafka中，我们可以认为一个group是一个“订阅者”，一个topic中的每个partions只会被一个“订阅者”中的一个 consumer消费，不过一个consumer可以消费多个partitions中的消息
注:Kafka的设计原理决定，对于一个topic，同一个group不能多于partition个数的consumer同时消费，否则将意味着某 些consumer无法得到消息

启动集群

依次在 huan01、huan02、huan03节点上启动 kafka

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

关闭集群

```
bin/kafka-server-stop.sh stop
bin/kafka-server-stop.sh stop
bin/kafka-server-stop.sh stop
```

Kafka 命令行操作

0)帮助

```
bin/kafka-topics.sh --list
```

1）查看当前服务器中的所有 topic

```
bin/kafka-topics.sh --zookeeper huan01:2181 --list
```

2）创建 topic

```
bin/kafka-topics.sh --zookeeper huan01:2181 --create --replication-factor 3 --
partitions 1 --topic first
选项说明:
--topic 定义 topic 名
--replication-factor 定义副本数
--partitions 定义分区数
```

3）删除 topic

```
bin/kafka-topics.sh --zookeeper huan01:2181 --delete --topic first
需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除。
```

4）发送消息

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh --broker-list
huan01:9092 --topic first
\>hello world
\>huan huan
```

5）消费消息

```
bin/kafka-console-consumer.sh --zookeeper huan01:2181 --topic first
bin/kafka-console-consumer.sh --bootstrap-server huan01:9092 --topic first
bin/kafka-console-consumer.sh --bootstrap-server huan01:9092 --from-beginning --
topic first
注意:--from-beginning：会把主题中以往所有的数据都读取出来
```

6）查看某个 Topic 的详情

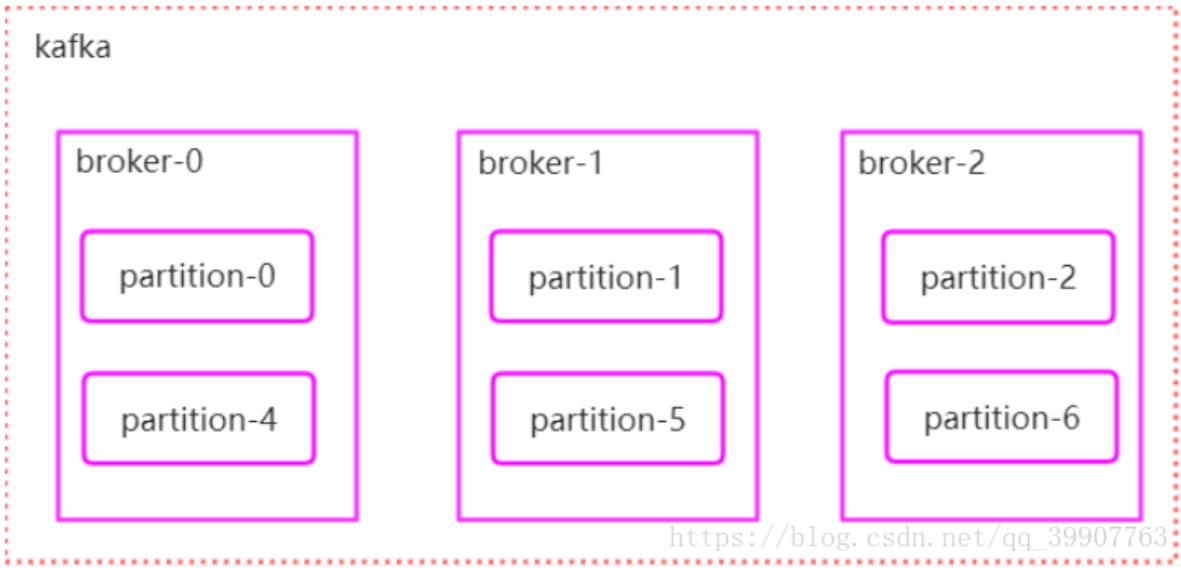
```
bin/kafka-topics.sh --zookeeper huan01:2181 --describe --topic first
```

7）修改分区数

```
bin/kafka-topics.sh --zookeeper huan01:2181 --alter --topic first --partitions 6
```

kafka的分区分配策略

1. 将所有broker（n个）和partition排序
2. 将第i个Partition分配到第（i mode n）个broker上



Producer如何把消息发送给对应分区

```
1.当key为空时，消息随机发送到各个分区（各个版本会有不同，有的是采用轮询的方式，有的是随机，有
的是一定时间内     只发送给固定partition,隔一段时间后随机换一个）
2.用key的ha'sh值对partion个数取模，决定要把消息发送到哪个partition上
```

消费者分区分配策略

Range 范围分区(默认的)

假如有10个分区，3个消费者，把分区按照序号排列0，1，2，3，4，5，6，7，8，9；消费者为C1,C2,C3，那么用分区数除以消费者数来决定每个Consumer消费几个Partition，除不尽的前面几个消费者将会多消费一个
最后分配结果如下

```
C1：0，1，2，3
C2：4，5，6
C3：7，8，9
```

如果有11个分区将会是：

```
C1：0，1，2，3
C2：4，5，6，7
C3：8，9，10
```

假如我们有两个主题T1,T2，分别有10个分区，最后的分配结果将会是这样：

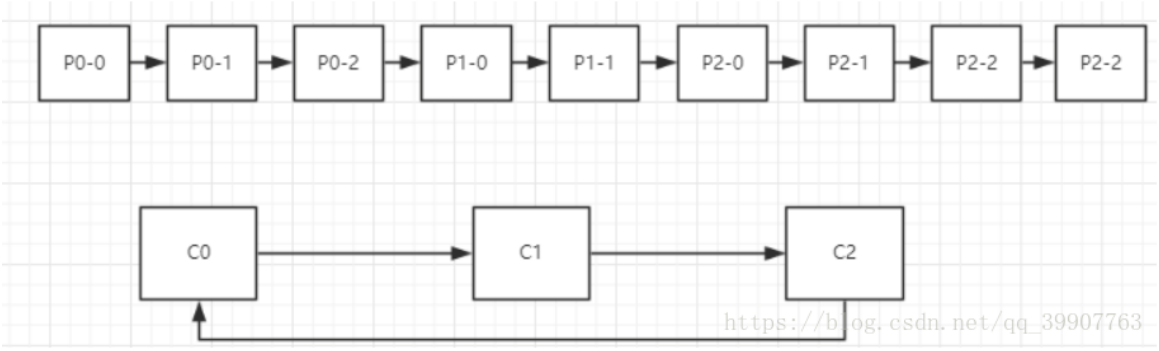
```
C1：T1（0，1，2，3） T2（0，1，2，3）
C2：T1（4，5，6） T2（4，5，6）
C3：T1（7，8，9） T2（7，8，9）
```

在这种情况下，C1多消费了两个分区

RoundRobin 轮询分区

```
把所有的partition和consumer列出来，然后轮询consumer和partition，尽可能的让把partition均匀的分配给consumer
假如有3个Topic T0（三个分区P0-0，P0-1,P0-2），T1(两个分区P1-0,P1-1)，T2(四个分区P2-0，P2-1，P2-2，P2-3)
有三个消费者：C0(订阅了T0，T1),C1（订阅了T1，T2），C2(订阅了T0,T2)
```

那么分区过程如下图所示



分区将会按照一定的顺序排列起来，消费者将会组成一个环状的结构，然后开始轮询。
P0-0分配给C0
P0-1分配给C1但是C1并没订阅T0，于是跳过C1把P0-1分配给C2,
P0-2分配给C0
P1-0分配给C1,
P1-1分配给C0,
P2-0分配给C1,
P2-1分配给C2,
P2-2分配给C1,
p2-3分配给C2

C0: P0-0，P0-2，P1-1
C1：P1-0，P2-0，P2-2
C2：P0-1，P2-1，P2-3

什么时候触发分区分配策略：

1.同一个Consumer Group内新增或减少Consumer
2.Topic分区发生变化

Rebalance的执行

kafka提供了一个角色Coordinator来执行。当Consumer Group的第一个Consumer启动的时候，他会向kafka集群中的任意一台broker发送GroupCoordinatorRequest请求，broker会返回一个负载最小的broker设置为coordinator，之后该group的所有成员都会和coordinator进行协调通信

整个Rebalance分为两个过程 jionGroup和sysncJion

joinGroup过程

在这一步中，所有的成员都会向coordinator发送JionGroup请求，请求内容包括group_id,member_id.protocol_metadata等，coordinator会从中选出一个consumer作为leader，并且把组成员信息和订阅消息，leader信息，rebanlance的版本信息发送给consumer

Synchronizing Group State阶段

组成员向coordinator发送SysnGroupRequet请求，但是只有leader会发送分区分配的方案（分区分配的方案其实是在消费者确定的），当coordinator收到leader发送的分区分配方案后，会通过SysnGroupResponse把方案同步到各个consumer中

offset 的维护

```
1）修改配置文件 consumer.properties
exclude.internal.topics=false
```

2）读取 offset

0.11.0.0 之前版本:

```
bin/kafka-console-consumer.sh --topic __consumer_offsets --zookeeper huan01:2181
--formatter
"kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter" --
consumer.config config/consumer.properties --from-beginning
```

0.11.0.0 之后版本(含):

```
bin/kafka-console-consumer.sh --topic __consumer_offsets --zookeeper huan01:2181
--formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --
consumer.config config/consumer.properties --from-beginning
```

消费者组案例

测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

案例实操
（1）在 hadoop102、hadoop103 上修改/opt/module/kafka/config/consumer.properties 配置文件中的 group.id 属性为任意组名。
vi consumer.properties -> group.id=atguigu
（2）在 huan01、huan02 上分别启动消费者
 bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first --consumer.config   config/consumer.properties

 bin/kafka-console-consumer.sh --bootstrap-server hadoop102:9092 --topic first --consumer.config
 config/consumer.properties

（3）在 hadoop104 上启动生产者
bin/kafka-console-producer.sh --broker-list hadoop102:9092 --topic first
\>hello world

（4）查看 hadoop102 和 hadoop103 的接收者。
 同一时刻只有一个消费者接收到消息

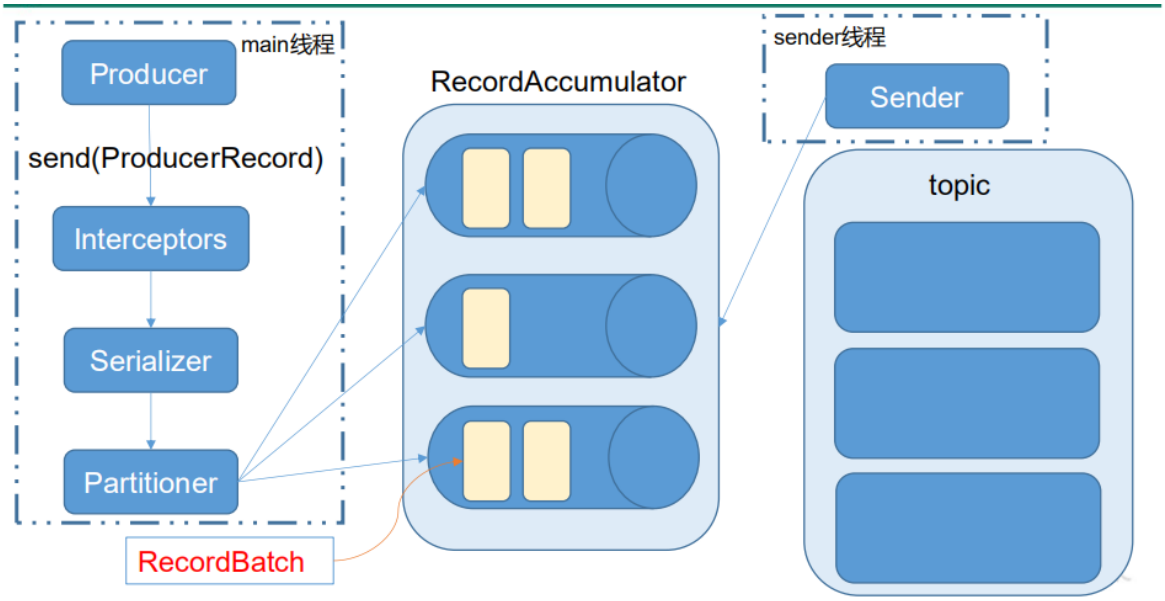
Zookeeper 在 Kafka 中的作用

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。
Controller 的管理工作都是依赖于 Zookeeper 的。

Kafka消息流程

生产者

Kafka 的 Producer 发送消息采用的是**异步发送**的方式。在消息发送的过程中，涉及到了两个线程,main线程和Sender线程，以及一个线程共享变量——RecordAccumulator。
main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker



相关参数：

batch.size： 只有数据积累到 batch.size 之后，sender 才会发送数据。

linger.ms： 如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

KafkaAPI分类

生产者API

```
KafkaProducer：需要创建一个生产者对象，用来发送数据
ProducerConfig：获取所需的一系列配置参数
ProducerRecord：每条数据都要封装成一个 ProducerRecord 对象
```

带回调函数的 API

```
回调函数(Callback)会在 producer 收到 ack 时调用，为异步调用，该方法有两个参数，分别是
RecordMetadata 和 Exception，如果 Exception 为 null，说明消息发送成功，如果
Exception 不为 null，说明消息发送失败。
注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。
```

同步发送 API

```
同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回 ack。由于 send 方法返回的是一
个 Future 对象，根据 Futrue 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对
象的 get 方法即可
```

消费者API

需要用到的类：
KafkaConsumer：需要创建一个消费者对象，用来消费数据
ConsumerConfig：获取所需的一系列配置参数
ConsuemrRecord：每条数据都要封装成一个 ConsumerRecord 对象
为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。
自动提交 offset 的相关参数：
enable.auto.commit：是否开启自动提交 offset 功能
auto.commit.interval.ms：自动提交 offset 的时间间隔

以下为自动提交 offset 的代码：

```
public class CustomConsumer {
public static void main(String[] args) {
 Properties props = new Properties();
 props.put("bootstrap.servers", "hadoop102:9092");
 props.put("group.id", "test");
 props.put("enable.auto.commit", "true");
 props.put("auto.commit.interval.ms", "1000");
 props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
 consumer.subscribe(Arrays.asList("first"));
 while (true) {
 ConsumerRecords<String, String> records =
consumer.poll(100);
 for (ConsumerRecord<String, String> record : records)
 System.out.printf("offset = %d, key = %s, value
= %s%n", record.offset(), record.key(), record.value());
 }
 } }
```

手动提交 offset

虽然自动提交 offset 十分简介便利，但由于其是基于时间提交的，开发人员难以把握 offset 提交的时机。因此 Kafka 还提供了手动提交 offset 的 API。
手动提交 offset 的方法有两种：分别是 commitSync（同步提交）和 commitAsync（异步提交）。两者的相同点是，都会将**本次** **poll** **的一批数据最高的偏移量提交**；不同点是，commitSync 阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而 commitAsync 则没有失败重试机制，故有可能提交失败。

1**）同步提交** offset

由于同步提交 offset 有失败重试机制，故更加可靠，以下为同步提交 offset 的示例。

```
public class CustomComsumer {
 public static void main(String[] args) {
 Properties props = new Properties();
//Kafka 集群
 props.put("bootstrap.servers", "hadoop102:9092");
//消费者组，只要 group.id 相同，就属于同一个消费者组
 props.put("group.id", "test");
 props.put("enable.auto.commit", "false");//关闭自动提交 offset
```

```
 props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(props);
 consumer.subscribe(Arrays.asList("first"));//消费者订阅主题
 while (true) {
//消费者拉取数据
 ConsumerRecords<String, String> records =
consumer.poll(100);
 for (ConsumerRecord<String, String> record : records) {
 System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),
record.key(), record.value());
 }
//同步提交，当前线程会阻塞直到 offset 提交成功
 consumer.commitSync();
 }
 } }
```

异步提交 offset

> 虽然同步提交 offset 更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞吐量会收到很大的影响。因此更多的情况下，会选用异步提交 offset 的方式。

以下为异步提交 offset 的示例：

```
public class CustomConsumer {
 public static void main(String[] args) {
 Properties props = new Properties();
 //Kafka 集群
 props.put("bootstrap.servers", "hadoop102:9092");
 //消费者组，只要 group.id 相同，就属于同一个消费者组
 props.put("group.id", "test");
 //关闭自动提交 offset
 props.put("enable.auto.commit", "false");
 props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
 consumer.subscribe(Arrays.asList("first"));//消费者订阅主题
 while (true) {
 ConsumerRecords<String, String> records = consumer.poll(100);//消费者拉取数据
 for (ConsumerRecord<String, String> record : records) {
 System.out.printf("offset = %d, key = %s, value
= %s%n", record.offset(), record.key(), record.value());
 }
//异步提交
 consumer.commitAsync(new OffsetCommitCallback() {
 @Override
 public void onComplete(Map<TopicPartition,
OffsetAndMetadata> offsets, Exception exception) {
 if (exception != null) {
 System.err.println("Commit failed for" +
offsets);
```

```
}
}
});
}
} }
```

自定义 Interceptor

拦截器原理

Producer 拦截器(interceptor)是在 Kafka 0.10 版本被引入的，主要用于实现 clients 端的定制化控制逻辑。
对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor 按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor 的实现接口是 org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括:
（1）configure(configs)
获取配置信息和初始化数据时调用。
（2）onSend(ProducerRecord)：
该方法封装进 KafkaProducer.send 方法中，即它运行在用户主线程中。Producer 确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。
（3）onAcknowledgement(RecordMetadata, Exception)：
该方法会在消息从 RecordAccumulator 成功发送到 Kafka Broker 之后，或者在发送过程中失败时调用。并且通常都是在 producer 回调逻辑触发之前。onAcknowledgement 运行在 producer 的 IO 线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢 producer 的消息发送效率。
（4）close：
关闭 interceptor，主要用于执行一些资源清理工作如前所述，interceptor 可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，并仅仅是捕获每个 interceptor 可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中
要特别留意