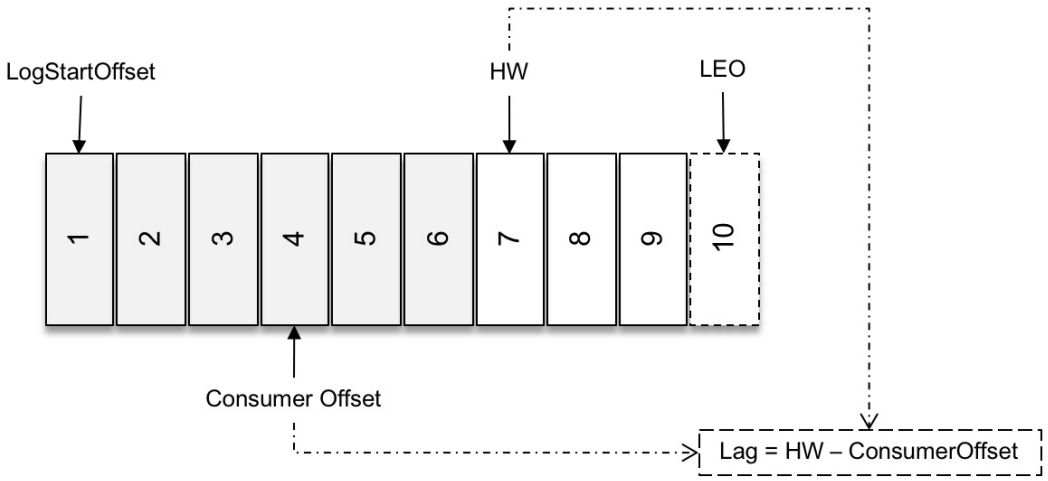
# 消费滞后

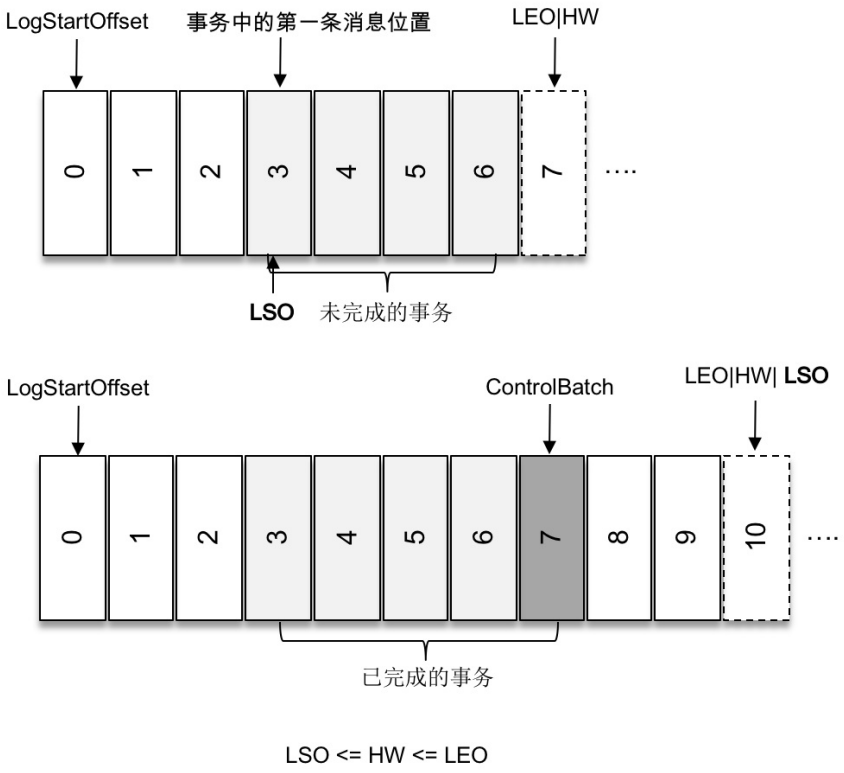
消息堆积是消息中间件的一大特色，消息中间件的流量削峰、冗余存储等功能正是得益于消息中间件的消息堆积能力。然而消息堆积是一把亦正亦邪的“双刃剑”，如果应用场合不恰当，反而会对上下游的业务造成不必要的麻烦，比如消息堆积势必会影响上下游整个调用链的时效性。在某些情况下，有些中间件如 RabbitMQ 在发生消息堆积时还会影响自身的性能。对 Kafka 而言，虽然消息堆积不会给其自身性能带来太大的困扰，但难免会影响上下游的业务，堆积过多有可能造成磁盘爆满，或者触发日志清除操作而造成消息丢失的情况。如何利用好消息堆积这把双刃剑，监控是其中关键的一步。

消息堆积是消费滞后（Lag）的一种表现形式，消息中间件中留存的消息与消费的消息之间的差值即为消息堆积量，也称为消费滞后（Lag）量。对 Kafka 的使用者而言，消费 Lag 是他们非常关心的一个指标。




通过前面章节的内容，我们了解 logStartOffset、HW、LEO 这些分区中消息位置的概念，如上图所示。对每一个分区而言，它的 Lag 等于 HW - ConsumerOffset 的值，其中 ConsumerOffset 表示当前的消费位移。

以上针对的都是普通的情况，如果为消息引入了事务，那么 Lag 的计算方式就会有所不同。如果消费者客户端的 isolation.level 参数配置为“read_uncommitted”（默认），那么 Lag 的计算方式不受影响；如果这个参数配置为“read_committed”，那么就要引入 LSO 来进行计算了。LSO 是 LastStableOffset 的缩写，如图下图所示。对未完成的事务而言，LSO 的值等于事务中第一条消息的位置（firstUnstableOffset），对已完成的事务而言，它的值同 HW 相同，所以我们可以得出一个结论：LSO≤HW≤LEO。




对于分区中有未完成的事务，并且消费者客户端的 isolation.level 参数配置为“read_committed”的情况，它对应的 Lag 等于 LSO - ConsumerOffset 的值。

为了便于说明问题，在下面的陈述中如无特殊说明，Lag 的计算都针对没有事务的情况。虽然使用事务的场景远没有非事务的场景多，但读者对 LSO 的概念也要有一定的认知，避免在真正使用事务的时候对 Lag 的理解造成偏差。

要计算 Lag，首先得获取 ConsumerOffset 和 HW 的值，ConsumerOffset 保存在内部主题__consumer_offsets 中，HW 又时刻在变化，那么这两个变量该如何获取呢？在27节中我们讲述了 kafka-consumer-groups.sh 脚本的用法，这个脚本可以让我们很方便地查看消费组内每个分区所对应的 Lag，我们不妨借鉴一下它的实现方法：

- 首先通过 DescribeGroupsRequest 请求获取当前消费组的元数据信息，当然在这之前还会通过 FindCoordinatorRequest 请求查找消费组对应的 GroupCoordinator。
- 接着通过 OffsetFetchRequest 请求获取消费位移 ConsumerOffset。
- 然后通过 KafkaConsumer 的 endOffsets(Collection partitions)方法（对应于 ListOffsetRequest 请求）获取 HW（LSO）的值。
- 最后通过 HW 与 ConsumerOffset 相减得到分区的 Lag，要获得主题的总体 Lag 只需对旗下的各个分区累加即可。

除了 Lag，我们发现 kafka-consumer-groups.sh 脚本中打印的其他信息也很重要，下面的示例程序（代码清单32-1）演示了如何实现同“bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group groupIdMonitor”一样的效果，其中还包含了它对应的 TOPIC、PARTITION、CURRENT-OFFSET、Log-END-OFFSET 等信息。如果读者只想关注其中的 Lag 信息，则可以自行缩减一下代码。

代码较长，可以在 Github 上获取，请点击[这里](https://github.com/hiddenzzh/kafka_book_demo/blob/mast

```
///代码清单32-1 消息堆积计算示例
import lombok.Builder;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import
```

```
org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.admin.*;
import
org.apache.kafka.clients.consumer.ConsumerConfig;
import
org.apache.kafka.clients.consumer.KafkaConsumer;
import
org.apache.kafka.clients.consumer.OffsetAndMetada
ta;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.TopicPartition;
import
org.apache.kafka.common.serialization.StringDeser
ializer;

import java.util.*;
import java.util.concurrent.ExecutionException;

import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

@Slf4j
public class KafkaConsumerGroupService {
    private String brokerList;
    private AdminClient adminClient;
    private KafkaConsumer<String, String>
kafkaConsumer;

    public KafkaConsumerGroupService(String
brokerList) {
        this.brokerList = brokerList;
    }
    //初始化
```

```
    public void init(){
        Properties props = new Properties();

props.put(CommonClientConfigs.BOOTSTRAP_SERVERS_C
ONFIG, brokerList);
        adminClient = AdminClient.create(props);
        kafkaConsumer =
ConsumerGroupUtils.createNewConsumer(brokerList,
                "kafkaAdminClientDemoGroupId");
    }
    //释放资源
    public void close(){
        if (adminClient != null) {
            adminClient.close();
        }
        if (kafkaConsumer != null) {
            kafkaConsumer.close();
        }
    }
    //收集消费信息的方法
    public List<PartitionAssignmentState>
collectGroupAssignment(
            String group) throws
ExecutionException, InterruptedException {
        //通过DescribeGroupsRequest请求获取当前消费组
的元数据信息
        DescribeConsumerGroupsResult groupResult
= adminClient

.describeConsumerGroups(Collections.singleton(gro
up));
        ConsumerGroupDescription description =
```

```
groupResult.all().get().get(group);

        List<TopicPartition> assignedTps = new
ArrayList<>();
        List<PartitionAssignmentState>
rowsWithConsumer = new ArrayList<>();
        Collection<MemberDescription> members =
description.members();
        if (members != null) {
            //通过OffsetFetchRequest请求获取消费位移
ConsumerOffset
            ListConsumerGroupOffsetsResult
offsetResult = adminClient

.listConsumerGroupOffsets(group);
            Map<TopicPartition,
OffsetAndMetadata> offsets = offsetResult

.partitionsToOffsetAndMetadata().get();
            if (offsets != null &&
!offsets.isEmpty()) {
                String state =
description.state().toString();
                if (state.equals("Stable")) {
                    rowsWithConsumer =
getRowsWithConsumer(description, offsets,
                            members, assignedTps,
group);
                }
            }
            List<PartitionAssignmentState>
rowsWithoutConsumer =
```

```
getRowsWithoutConsumer(description, offsets,
                                assignedTps, group);

rowsWithConsumer.addAll(rowsWithoutConsumer);
        }
        return rowsWithConsumer;
    }
    //有消费者成员信息的处理
    private List<PartitionAssignmentState>
getRowsWithConsumer(
            ConsumerGroupDescription description,
            Map<TopicPartition,
OffsetAndMetadata> offsets,
            Collection<MemberDescription>
members,
            List<TopicPartition> assignedTps,
String group) {
        List<PartitionAssignmentState>
rowsWithConsumer = new ArrayList<>();
        for (MemberDescription member : members)
{
            MemberAssignment assignment =
member.assignment();
            if (assignment == null) {
                continue;
            }
            Set<TopicPartition> tpSet =
assignment.topicPartitions();
            if (tpSet.isEmpty()) {

rowsWithConsumer.add(PartitionAssignmentState.bui
lder()
```

```
.group(group).coordinator(description.coordinator
())

.consumerId(member.consumerId()).host(member.host
())

.clientId(member.clientId()).build());

            } else {
                Map<TopicPartition, Long>
logSizes =

kafkaConsumer.endOffsets(tpSet);
                assignedTps.addAll(tpSet);
                List<PartitionAssignmentState>
tempList = tpSet.stream()

.sorted(comparing(TopicPartition::partition))
                        .map(tp ->
getPasWithConsumer(logSizes, offsets, tp,
                                group, member,
description)).collect(toList());

rowsWithConsumer.addAll(tempList);
            }
        }
        return rowsWithConsumer;
    }

    private PartitionAssignmentState
getPasWithConsumer(
            Map<TopicPartition, Long> logSizes,
            Map<TopicPartition,
```

```
OffsetAndMetadata> offsets,
            TopicPartition tp, String group,
            MemberDescription member,
            ConsumerGroupDescription description)
{
        long logSize = logSizes.get(tp);
        if (offsets.containsKey(tp)) {
            long offset =
offsets.get(tp).offset();
            long lag = getLag(offset, logSize);
            return
PartitionAssignmentState.builder().group(group)

.coordinator(description.coordinator()).lag(lag)

.topic(tp.topic()).partition(tp.partition())

.offset(offset).consumerId(member.consumerId())

.host(member.host()).clientId(member.clientId())
                    .logSize(logSize).build();
        }else {
            return
PartitionAssignmentState.builder()

.group(group).coordinator(description.coordinator
())

.topic(tp.topic()).partition(tp.partition())

.consumerId(member.consumerId()).host(member.host
())
```

```
.clientId(member.clientId()).logSize(logSize).bui
ld();
            }
        }
        //计算Lag
        private static long getLag(long offset, long
logSize) {
            long lag = logSize - offset;
            return lag < 0 ? 0 : lag;
        }
        //没有消费者成员信息的处理
        private List<PartitionAssignmentState>
getRowsWithoutConsumer(
                ConsumerGroupDescription description,
                Map<TopicPartition,
OffsetAndMetadata> offsets,
                List<TopicPartition> assignedTps,
String group) {
            Set<TopicPartition> tpSet =
offsets.keySet();

            return tpSet.stream()
                    .filter(tp ->
!assignedTps.contains(tp))
                    .map(tp -> {
                        long logSize = 0;
                        Long endOffset =
kafkaConsumer.

endOffsets(Collections.singleton(tp)).get(tp);
                        if (endOffset != null) {
                            logSize = endOffset;
                        }
```

```
                    long offset =
offsets.get(tp).offset();
                    return
PartitionAssignmentState.builder().group(group)

.coordinator(description.coordinator())

.topic(tp.topic()).partition(tp.partition())

.logSize(logSize).lag(getLag(offset, logSize))

.offset(offset).build();

}).sorted(comparing(PartitionAssignmentState::get
Partition))
                .collect(toList());
    }
}

class ConsumerGroupUtils{
    //创建KafkaConsumer实例，因为要通过
KafkaConsumer.endOffsets()方法获取HW(LSO)
    static KafkaConsumer<String, String>
createNewConsumer(
            String brokerUrl, String groupId) {
        Properties props = new Properties();

props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG
, brokerUrl);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
groupId);

props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFI
```

```
G, "false");

props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_C
ONFIG,

StringDeserializer.class.getName());

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS
_CONFIG,

StringDeserializer.class.getName());
        return new KafkaConsumer<>(props);
    }
    //打印最终的输出结果，如果要展示到页面上，则可以将
List<PartitionAssignmentState> list
    //转换成JSON之类的输出到前端，然后通过页面展示
    static void
printPasList(List<PartitionAssignmentState> list)
{
        System.out.println(String.format("%-40s
%-10s %-15s %-15s %-10s" +
                        " %-50s%-30s %s",
"TOPIC", "PARTITION",
                "CURRENT-OFFSET", "LOG-END-
OFFSET", "LAG",
                "CONSUMER-ID", "HOST", "CLIENT-
ID"));

        list.forEach(item ->

System.out.println(String.format("%-40s %-10s
%-15s " +
                                "%-15s %-10s
```

```
%-50s%-30s %s",
                            item.getTopic(),
item.getPartition(), item.getOffset(),
                            item.getLogSize(),
item.getLag(),

Optional.ofNullable(item.getConsumerId()).orElse(
"-"),

Optional.ofNullable(item.getHost()).orElse("-"),

Optional.ofNullable(item.getClientId()).orElse("-
")));
    }
}
//最终展示结果所需的JavaBean
@Data
@Builder
class PartitionAssignmentState {
    private String group;
    private Node coordinator;
    private String topic;
    private int partition;
    private long offset;
    private long lag;
    private String consumerId;
    private String host;
    private String clientId;
    private long logSize;
}
```

使用上面这段示例程序时需要导入与使用 Java 客户端时相同的 Maven 依赖（kafka-clients）。上面示例程序的主函数如下：

```
public static void main(String[] args) throws
ExecutionException,
        InterruptedException {
    KafkaConsumerGroupService service =
            new
KafkaConsumerGroupService("localhost:9092");
    service.init();
    List<PartitionAssignmentState> list =

service.collectGroupAssignment("groupIdMonitor");
    ConsumerGroupUtils.printPasList(list);
    service.close();
}
```

读者可以运行这个程序并对比与 kafka-consumer-groups.sh --describe 有何不同。

kafka-consumer-groups.sh 脚本的功能是通过 kafka.admin.ConsumerGroupCommand 类实现的，而上面的示例就是用 Java 语言和 KafkaAdminClient 作为辅助来重写由 Scala 语言编写的 ConsumerGroupCommand 类中的 collectGroupOffsets() 方法。代码清单32-1的代码量偏多，建议读者按照它和 collectGroupOffsets() 方法中的源码重新写一遍，相信会让你对 Kafka 的认知更加深刻。

我们可不可以直接调用collectGroupOffsets()方法而不需要这么复杂的重写过程呢？很遗憾的是不可以，这是由于collectGroupOffsets()方法中调用的PartitionAssignmentState类的权限问题（private[admin]）而导致的。

不过事情也不是绝对的，我们可以借助 jackson-module-scala 工具包来通过序列化的手段绕过 PartitionAssignmentState 类的权限问题，对应的 Maven 依赖如下：

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.4</version>
</dependency>
<dependency>

<groupId>com.fasterxml.jackson.module</groupId>
    <artifactId>jackson-module-
scala_2.11</artifactId>
    <version>2.9.5</version>
</dependency>
```

注意如果本地安装的 Scala 版本与所配置的 jackson-module-scala 版本不一致，则会报出一些异常。由于我们还会调用 Kafka 服务端（ConsumerGroupCommand 类就是服务端的代码，而不是客户端的）的代码，所以还需要导入对应的 Maven 依赖：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>2.0.0</version>
</dependency>
```

对应的示例如代码清单32-2所示。

```
//代码清单32-2 直接调用ConsumerGroupCommand实现
import
com.fasterxml.jackson.databind.DeserializationFea
ture;
import com.fasterxml.jackson.databind.JavaType;
import
com.fasterxml.jackson.databind.ObjectMapper;

```

```
import
com.fasterxml.jackson.module.scala.DefaultScalaMo
dule;
import kafka.admin.ConsumerGroupCommand;
import lombok.Builder;
import lombok.Data;

import java.io.IOException;
import java.util.List;
import java.util.Optional;

public class KafkaConsumerGroupAnother {
    public static void main(String[] args) throws
IOException {
        String[] agrs = {"--describe", "--
bootstrap-server",
                "localhost:9092", "--group",
"groupIdMonitor"};

ConsumerGroupCommand.ConsumerGroupCommandOptions
options =
                new
ConsumerGroupCommand.ConsumerGroupCommandOptions(
agrs);
        ConsumerGroupCommand.ConsumerGroupService
kafkaConsumerGroupService =
                new
ConsumerGroupCommand.ConsumerGroupService(options
);

        ObjectMapper mapper = new ObjectMapper();
        //1. 使用jackson-module-scala_2.11
        mapper.registerModule(new
```

```
DefaultScalaModule());
        //2. 反序列化时忽略对象不存在的属性

mapper.configure(DeserializationFeature.FAIL_ON_U
NKNOWN_PROPERTIES,
false);
        //3. 将Scala对象序列化成JSON字符串
        //这里原本会有权限问题，通过序列化绕过
        String source =
mapper.writeValueAsString(kafkaConsumerGroupServi
ce.
                collectGroupOffsets()._2.get());
        //4. 将JSON字符串反序列化成Java对象
        List<PartitionAssignmentStateAnother>
target = mapper.readValue(source,

getCollectionType(mapper,List.class,

PartitionAssignmentStateAnother.class));
        //5. 排序
        target.sort((o1, o2) -> o1.getPartition()
- o2.getPartition());
        //6. 打印
        //这个方法参考前面ConsumerGroupUtils的
printPasList()方法
        printPasList(target);
    }

    public static JavaType
getCollectionType(ObjectMapper mapper,

Class<?> collectionClass,
```

```
Class<?>... elementClasses) {
        return mapper.getTypeFactory()

.constructParametricType(collectionClass,
elementClasses);
    }

@Data
@Builder
class PartitionAssignmentStateAnother {
    private String group;
    private Node coordinator;
    private String topic;
    private int partition;
    private long offset;
    private long lag;
    private String consumerId;
    private String host;
    private String clientId;
    private long logSize;

    @Data
    public static class Node{
        public int id;
        public String idString;
        public String host;
        public int port;
        public String rack;
    }
}
```

本段代码可以在 Github 上下载，请点击这里
(https://github.com/hiddenzzh/kafka_book_demo/blob/mast

在原本的代码清单32-1中，PartitionAssignmentState 中的 coordinator 类型是 Node，这个类型需要自定义，否则会报错，所以在代码清单32-2中又重写了 PartitionAssignmentState 类为 PartitionAssignmentStateAnother，读者需要注意其中的区别（建议读者跟着写一遍，这样能够深刻地体会到其中的细节问题）。如果页面中需要展示这些信息，那么我们甚至可以直接返回代码清单32-2中第3步骤的 source 字符串给页面，方便快捷。

# 监控指标说明

Kafka 自身提供的 JMX 监控指标已经超过了500个，本书不可能一一将其罗列，只能挑选部分重要及常用的指标来进行说明。

上一节的第一张图中除了展示了消息流入速度（MessagesInPerSec），还展示了网络流入/流出速度，这2个指标对应的 MBean 名称如下表所示。

| 指 标 名 称 | MBean 名称 |
| --- | --- |
| 网络流入速率（bytesIn） | kafka.server:type=BrokerTopicMetrics,name=Byt |
| 网络流出速率（bytesOut） | kafka.server:type=BrokerTopicMetrics,name=Byt |

这两个指标都是 broker 端的指标，分别对应前面章节中提及的 byteIn 和 byteOut。它们的属性列表同 MessagesInPerSec 的类似，与 MessagesInPerSec 指标不同的是，这2个指标的单位为 B/s，具体的使用方式可以参考第42节。

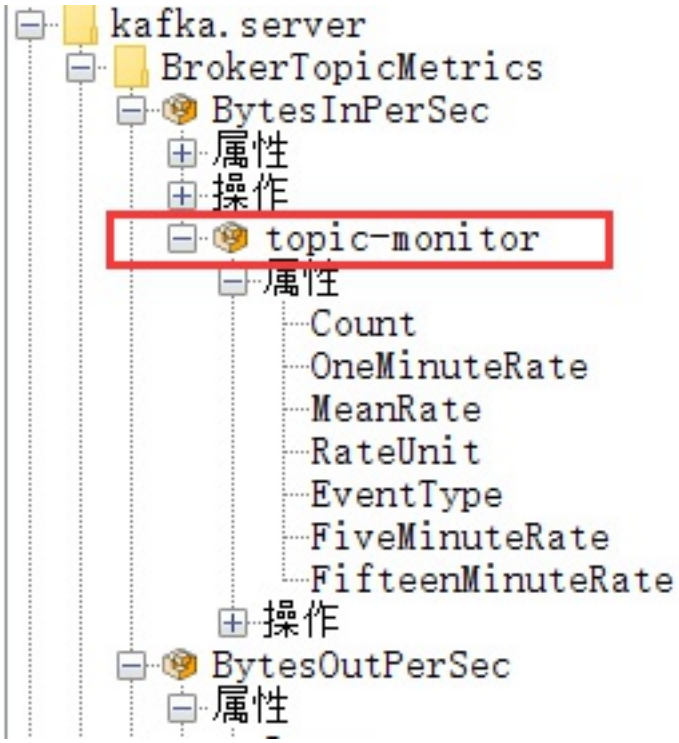
Kafka 并没有提供类似 MessagesOutPerSec 的指标，这是为什么呢？因为消息是以批次的形式发送给消费者的，在这个过程中并不会再展开（展开会严重影响性能，如果仅仅为了统计一个普通的指标而

展开，则会显得非常得不偿失）这些批次的内容来统计消息的个数，所以对 Kafka 而言，它也不知道发送了多少条消息，也就不会有类似 MessagesOutPerSec 这样的指标了。

不过在 Kafka 中有一个 TotalFetchRequestsPerSec 指标用于统计每秒拉取请求的次数，它可以从侧面反映出消息被拉取的多少。这个指标还有一个对应的 TotalProduceRequestsPerSec，用于统计每秒写入请求的次数。这2个指标对应的 MBean 名称如下表所示。

| 指 标 名 称 | M |
| --- | --- |
| TotalFetchRequestsPerSec | kafka.server:type=BrokerTopicMe |
| TotalProduceRequestsPerSec | kafka.server:type=BrokerTopicMe |

这些指标还有对应的与主题相关的指标，如下图所示。




主题 topic-monitor 的 MBean 名称为：

```
kafka.server:type=BrokerTopicMetrics,name=BytesIn
PerSec,topic=topic-monitor
```

由此可以归纳出主题端的 BytesInPerSec 指标的 MBean 名称为：

```
kafka.server:type=BrokerTopicMetrics,name=BytesIn
PerSec,topic=<topic>
```

这个规则对于其他主题端的指标同样适用。注意并不是每个 broker 端的指标都有其对应的主题端的指标，比如 ActiveControllerCount，它表示当前 broker 是否是集群的控制器。这个指标只有2个可选值，要么为0，要么为1。如果为1，则表示当前 broker 就是集群的控制器。任何时刻一个集群中有且仅有一个控制器，如果集群中所有 broker 的 ActiveControllerCount 指标之和不为1，则说明发生了异常情况，需要及时地告警以通知相关人员排查故障。ActiveControllerCount 对应于41节第二张图中的 Controller 标记，它的 MBean 名称为：

```
kafka.controller:type=KafkaController,name=Active
ControllerCount
```

与 UnderReplicatedPartitions 指标同级的还有 LeaderCount、PartitionCount、IsrExpandPerSec 和 IsrShrinksPerSec 这4个重要的指标，它们分别表征了 broker 中 leader 副本的总数、分区的总数、ISR 集合扩张速度和 ISR 集合收缩速度。这4个指标对应的 MBean 名称如下表所示。

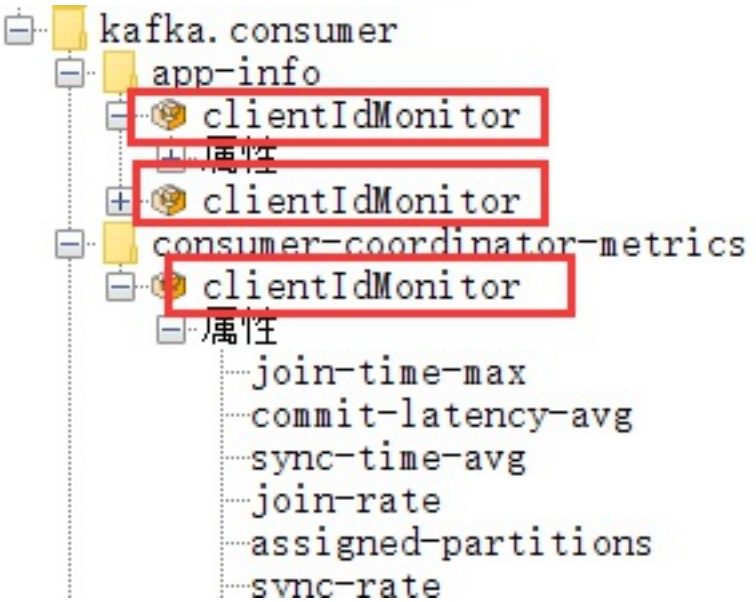
| 指 标 名 称 | MBean 名称 |
| --- | --- |
| LeaderCount | kafka.server:type=ReplicaManager,name=Le |
| PartitionCount | kafka.server:type=ReplicaManager,name=Pa |
| IsrShrinksPerSec | kafka.server:type=ReplicaManager,name=Isr |
| IsrExpandsPerSec | kafka.server:type=ReplicaManager,name=Isr |

对 LeaderCount 和 PartitionCount 而言，在前面的篇幅中已经有所提及，尤其是 LeaderCount，它牵涉集群的负载是否均衡。而 IsrExpandPerSec 和 IsrShrinksPerSec 这2个代表 ISR 集合变化速度的指标可以用来监测 Kafka 集群的性能问题。

对 Kafka 的客户端而言，它同样提供了可供 JMX 获取的监控指标，我们在运行 Kafka 客户端的时候同样需要显式地打开 JMX 功能，比如添加以下运行参数：

```
-Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

相比于 broker 端，客户端的指标就少了很多，不过每一个客户端指标都有一个对应的 clientId，如下图所示，其中 clientIdMonitor 就是客户端的 clientId。




Kafka 还提供了许多其他重要的指标，但笔者并不打算再多赘述，读者可以通过 JConsole 工具和 Kafka 官方文档(http://kafka.apache.org/documentation/#monitoring)来一一探索指标的奥秘。

# 监控模块

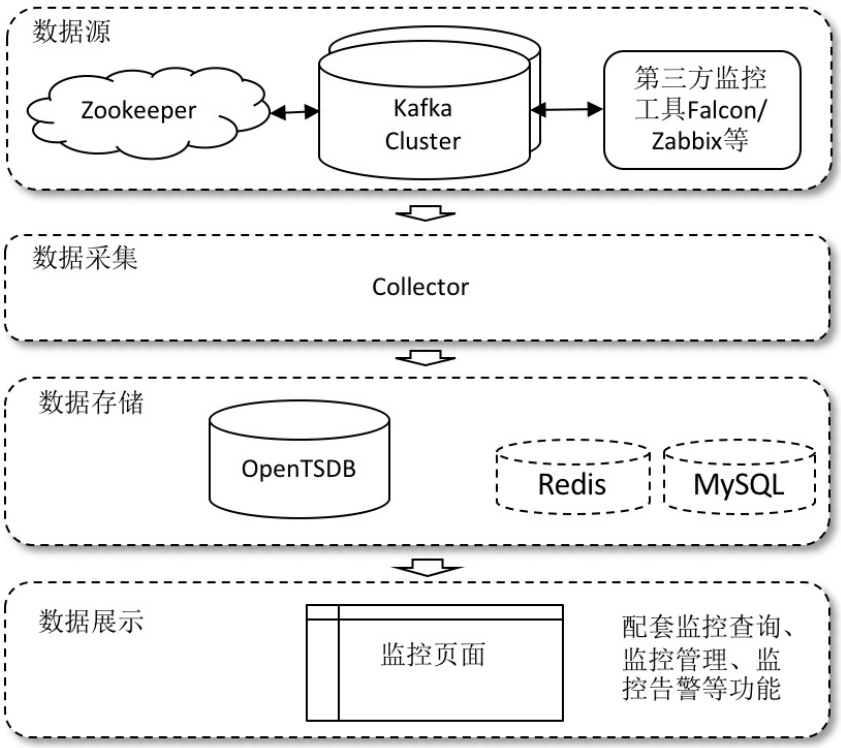
Kafka 的监控架构主要分为数据采集、数据存储和数据展示这3个部分。数据采集主要指从各个数据源采集监控数据并做一些必要的运算，然后发送给数据存储模块进行存储。数据源可以是 Kafka 配套

的 ZooKeeper、Kafka 自身提供的内部运行指标（通过 JMX 获取）、Kafka 内部的一些数据（比如__consumer_offset 中存储的信息，通过 Kafka 自定义协议获取）、Falcon/Zabbix 等第三方工具（或者其他类似的工具，主要用来监控集群的硬件指标）。

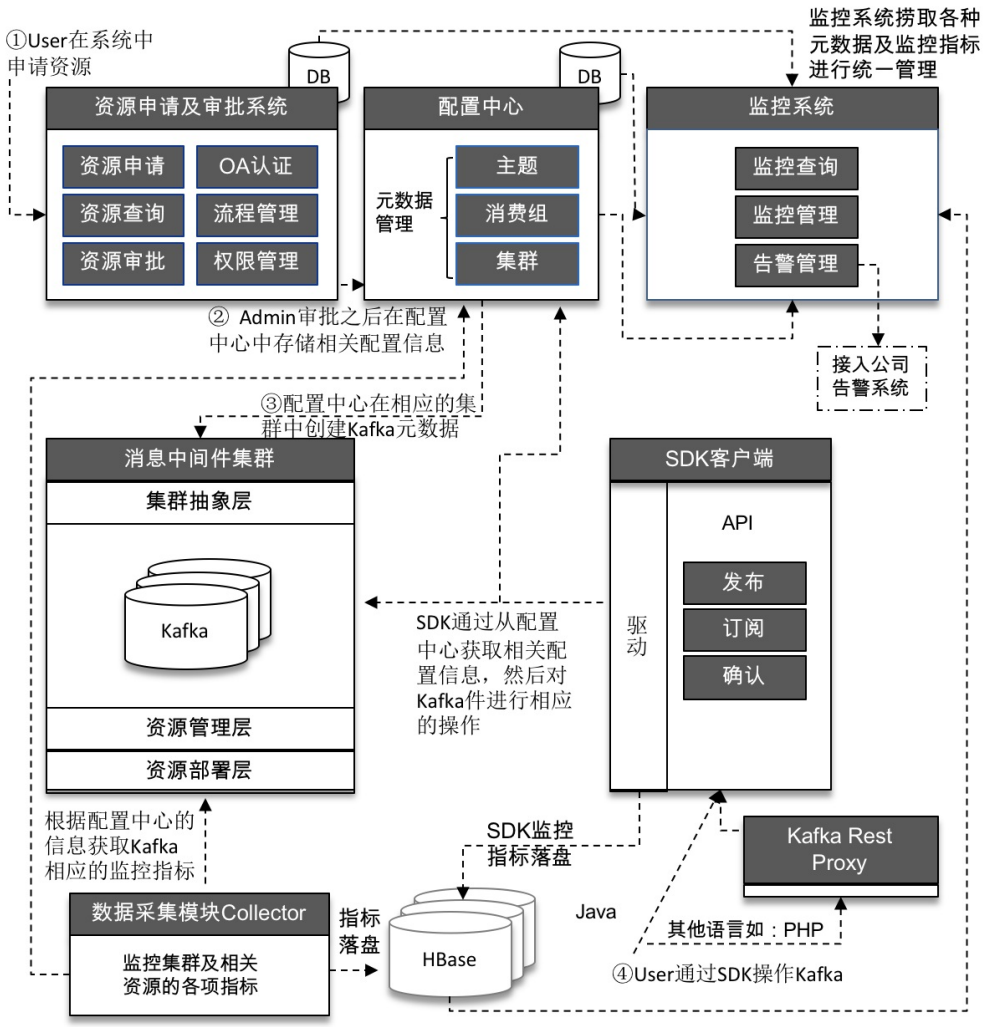
数据存储指将采集的原始数据经过一定的预处理后进行相应的存储，方便数据清洗（这个步骤可以省略）和数据展示。数据存储可以采用 OpenTSDB 之类的基于时间序列的数据库，方便做一些聚合计算，也可以附加采用 Redis、MySQL 等存储特定数据。

顾名思义，数据展示是将经过预处理的、存储的数据展示到监控页面上，以便提供丰富的 UI 给用户使用。当然数据展示模块也可以绕过数据存储模块直接通向数据采集模块，或者从数据源直接拉取数据。

整个监控系统的模型架构如下图所示。




监控模块是 Kafka 生态链中的重要一环，它是查看 Kafka 运行状态的主要依据，是排查故障的重要参考，同时是触发告警的源头，以便及时预防或修复故障。下图展示了 Kafka 的某种应用生态，监控系统及监控数据的采集都在其中。




首先，用户在资源申请审批系统中申请所需要使用的 Kafka 资源。管理员在审批完用户的申请之后，将相应的配置信息存储在配置中心，然后由配置中心负责创建相应的 Kafka 资源（比如根据预先申请的分区数、副本因子数创建对应的主题）。在资源创建成功之后会触发数据采集模块（Collector）对监控指标进行收集，最终存入预先设定的存储模块，比如 HBase。

用户通过封装后的 SDK 进行生产消费。SDK 中除了包含原生的 Kafka 客户端的功能，还包含了与应用生态中各个其他模块的互动功能，比如监听配置中心配置的变更以便及时进行相应的处理。如果用户采用的编程语言与 SDK 的实现语言互不相通，则可以使用 Kafka REST Proxy 来作为跨语言应用的补救措施。与此同时，SDK 中也有其相应的指标，比如业务相关的消息发送和消费的速度、重试的次数等，牵涉 SDK 的地方需要自定义原本 Kafka 所没有的监控指标。

无论通过 Collector 采集的指标数据，还是 SDK 上送的指标数据，在存入存储模块之前都可以做一定的预处理，比如在 Collector 中可以根据收集到的数据对各个 broker 节点的负载进行归一化的处理，然后将处理后的计算值保存到存储模块中，进而方便页面的展示。

在上图展示的应用生态中还缺失了运维这一环，前面的章节中多多少少都提到了一些运维相关的内容。有兴趣的读者还可以关注一下 LinkedIn 开源的 [Kafka Cruise Control (https://github.com/linkedin/cruise-control)](https://github.com/linkedin/cruise-control)——旨在使 Kafka 实现大规模自动化运维。

# 总结

这两节主要讲述如何自定义实现一个 Kafka 监控系统，其中包括页面整体的布局把控、监控数据的来源、监控指标的说明，以及监控模块在整个 Kafka 应用生态中所处的地位。这里并不讲述如何使用某款 Kafka 监控产品，而是给读者提供一个实现监控产品的思路。如果读者不想耗费精力实现一款监控产品而是想直接使用开源现成的，那么本章的内容也可以帮助读者更好地理解这些监控产品的实现原理。