尚硅谷大数据技术之 Kafka（源码解析）

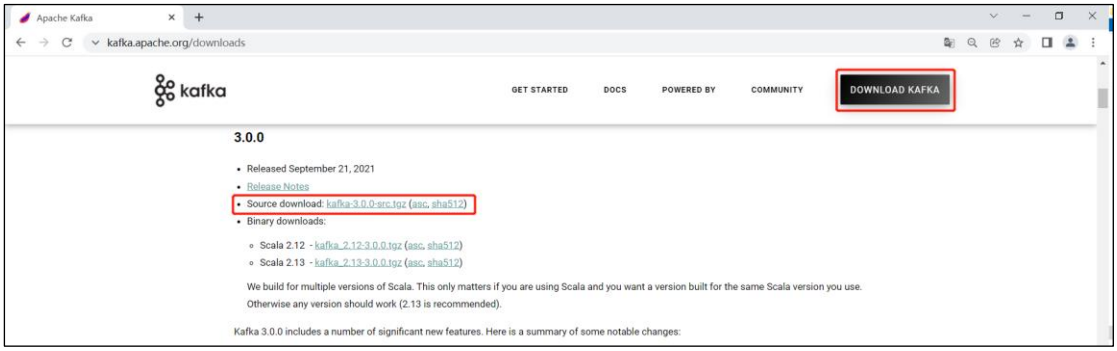
（作者：尚硅谷研究院）

版本：V3.3

第 1 章 源码环境准备

1.1 源码下载地址

http://kafka.apache.org/downloads



1.2 安装 JDK&Scala

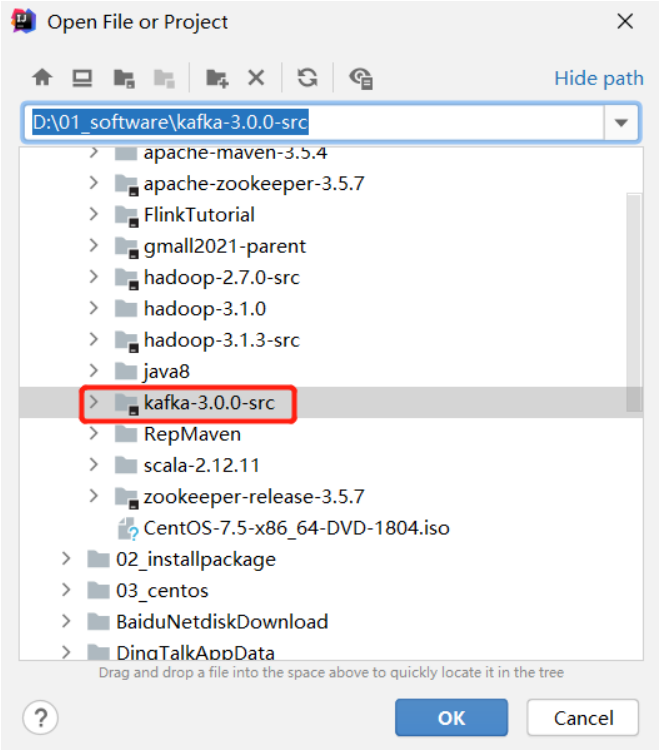
需要在 Windows 本地安装 JDK 8 或者 JDK8 以上版本。

需要在 Windows 本地安装 Scala2.12。

1.3 加载源码

将 kafka-3.0.0-src.tgz 源码包，解压到非中文目录。例如：D:\01_software\kafka-3.0.0-src。

打开 IDEA，点击 File->Open…->源码包解压的位置。

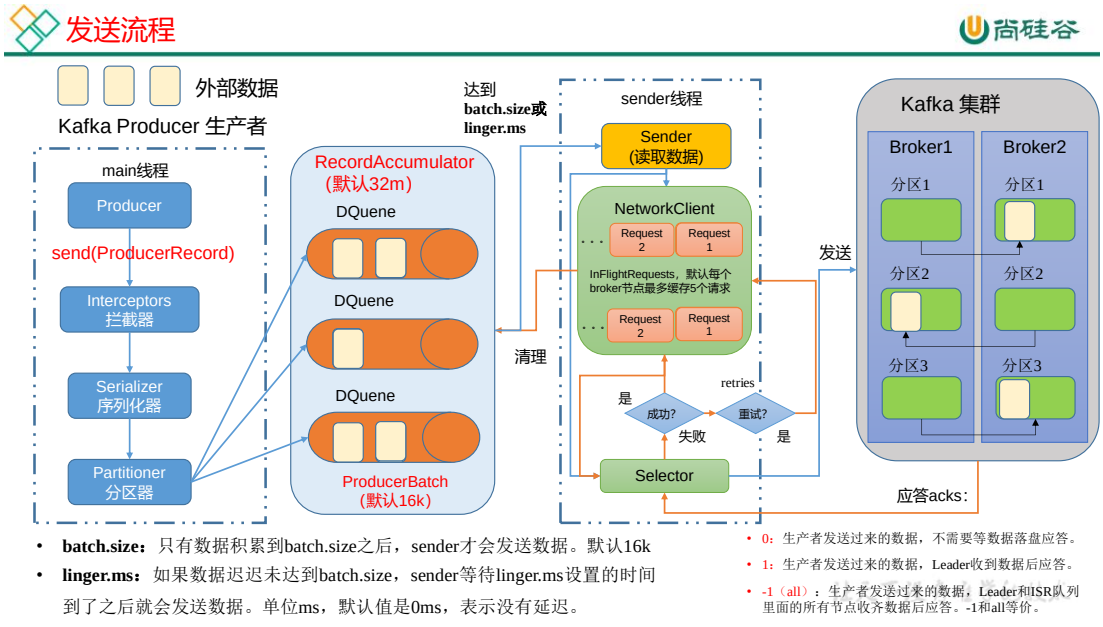


1.4 安装 gradle

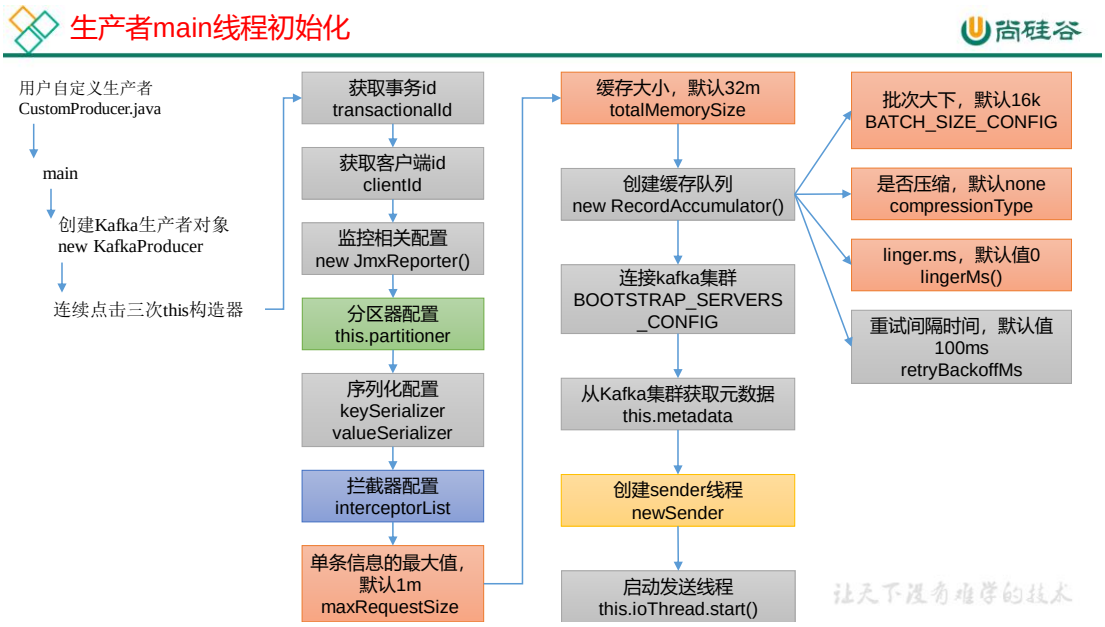
Gradle 是类似于 maven 的代码管理工具。安卓程序管理通常采用 Gradle。

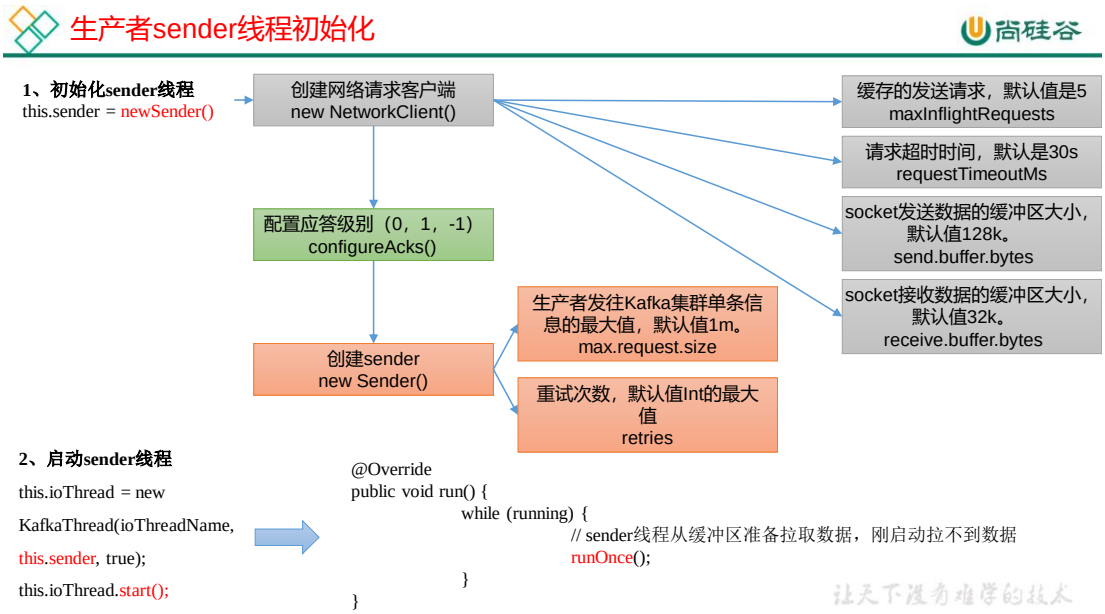
IDEA 自动帮你下载安装，下载的时间比较长（网络慢，需要 1 天时间，有 VPN 需要几分钟）。

第 2 章 生产者源码



2.1 初始化





2.1.1 程序入口

1）从用户自己编写的 main 方法开始阅读

```
package com.atguigu.kafka.producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class CustomProducer {

    public static void main(String[] args) {
```

```
        // 0 配置
        Properties properties = new Properties();

        // 连接集群 bootstrap.servers

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop102
:9092,hadoop103:9092");

        // 指定对应的 key 和 value 的序列化类型 key.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,Strin
gSerializer.class.getName());

        // 1 创建 kafka 生产者对象
        // "" hello
        KafkaProducer<String,   String>   kafkaProducer   =   new
KafkaProducer<>(properties);

        // 2 发送数据
        for (int i = 0; i < 5; i++) {
            kafkaProducer.send(new
ProducerRecord<>("first","atguigu"+i));
        }

        // 3 关闭资源
        kafkaProducer.close();
    }
}
```

2.1.2 生产者 main 线程初始化

点击 main()方法中的 KafkaProducer()。

KafkaProducer.java

```
public KafkaProducer(Properties properties) {
    this(properties, null, null);
}

public    KafkaProducer(Properties    properties,    Serializer<K>
keySerializer, Serializer<V> valueSerializer) {
    this(Utils.propsToMap(properties),              keySerializer,
valueSerializer);
}

public  KafkaProducer(Map<String,  Object>  configs,  Serializer<K>
keySerializer, Serializer<V> valueSerializer) {
    this(new
ProducerConfig(ProducerConfig.appendSerializerToConfig(configs,
keySerializer, valueSerializer)),
          keySerializer,   valueSerializer,   null,   null,   null,
Time.SYSTEM);
}
```

跳转到 KafkaProducer 构造方法。

```
KafkaProducer(ProducerConfig config,
              Serializer<K> keySerializer,
              Serializer<V> valueSerializer,
              ProducerMetadata metadata,
              KafkaClient kafkaClient,
              ProducerInterceptors<K, V> interceptors,
              Time time) {
    try {
        this.producerConfig = config;
        this.time = time;
        // 获取事务 id
        String                    transactionalId                    =
config.getString(ProducerConfig.TRANSACTIONAL_ID_CONFIG);
        // 获取客户端 id
        this.clientId                                                 =
config.getString(ProducerConfig.CLIENT_ID_CONFIG);

        LogContext logContext;
        if (transactionalId == null)
            logContext   =   new   LogContext(String.format("[Producer
clientId=%s] ", clientId));
        else
            logContext   =   new   LogContext(String.format("[Producer
clientId=%s, transactionalId=%s] ", clientId, transactionalId));
        log = logContext.logger(KafkaProducer.class);
        log.trace("Starting the Kafka producer");

        Map<String,           String>           metricTags           =
Collections.singletonMap("client-id", clientId);
        MetricConfig           metricConfig           =           new
MetricConfig().samples(config.getInt(ProducerConfig.METRICS_NUM_S
AMPLES_CONFIG))

    .timeWindow(config.getLong(ProducerConfig.METRICS_SAMPLE_WINDO
W_MS_CONFIG), TimeUnit.MILLISECONDS)

    .recordLevel(Sensor.RecordingLevel.forName(config.getString(Pr
oducerConfig.METRICS_RECORDING_LEVEL_CONFIG)))
                .tags(metricTags);
        List<MetricsReporter>               reporters               =
config.getConfiguredInstances(ProducerConfig.METRIC_REPORTER_CLAS
SES_CONFIG,
                MetricsReporter.class,

    Collections.singletonMap(ProducerConfig.CLIENT_ID_CONFIG,
clientId));
        // 监控相关配置
        JmxReporter jmxReporter = new JmxReporter();

    jmxReporter.configure(config.originals(Collections.singletonMa
p(ProducerConfig.CLIENT_ID_CONFIG, clientId)));
        reporters.add(jmxReporter);
        MetricsContext          metricsContext          =          new
KafkaMetricsContext(JMX_PREFIX,
```

```
    config.originalsWithPrefix(CommonClientConfigs.METRICS_CONTEXT
_PREFIX));
        this.metrics = new Metrics(metricConfig, reporters, time,
metricsContext);
        // 分区器配置
        this.partitioner = config.getConfiguredInstance(
                ProducerConfig.PARTITIONER_CLASS_CONFIG,
                Partitioner.class,

    Collections.singletonMap(ProducerConfig.CLIENT_ID_CONFIG,
clientId));
        // 重试时间间隔参数配置，默认值 100ms
        long                    retryBackoffMs                    =
config.getLong(ProducerConfig.RETRY_BACKOFF_MS_CONFIG);
        // 序列化配置
        if (keySerializer == null) {
            this.keySerializer                                    =
config.getConfiguredInstance(ProducerConfig.KEY_SERIALIZER_CLASS_
CONFIG,

         Serializer.class);

    this.keySerializer.configure(config.originals(Collections.sing
letonMap(ProducerConfig.CLIENT_ID_CONFIG, clientId)), true);
        } else {

    config.ignore(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG);
            this.keySerializer = keySerializer;
        }
        if (valueSerializer == null) {
            this.valueSerializer                                  =
config.getConfiguredInstance(ProducerConfig.VALUE_SERIALIZER_CLAS
S_CONFIG,Serializer.class);

    this.valueSerializer.configure(config.originals(Collections.si
ngletonMap(ProducerConfig.CLIENT_ID_CONFIG, clientId)), false);
        } else {

    config.ignore(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG);
            this.valueSerializer = valueSerializer;
        }

        // 拦截器配置
        List<ProducerInterceptor<K, V>> interceptorList = (List)
config.getConfiguredInstances(
                ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                ProducerInterceptor.class,

    Collections.singletonMap(ProducerConfig.CLIENT_ID_CONFIG,
clientId));
        if (interceptors != null)
            this.interceptors = interceptors;
        else
            this.interceptors                  =                  new
ProducerInterceptors<>(interceptorList);
        ClusterResourceListeners     clusterResourceListeners     =
configureClusterResourceListeners(keySerializer,
```

```
                valueSerializer, interceptorList, reporters);
        // 生产者发往 Kafka 集群单条信息的最大值，默认 1m
        this.maxRequestSize                                          =
config.getInt(ProducerConfig.MAX_REQUEST_SIZE_CONFIG);
        // 缓存大小，默认 32m
        this.totalMemorySize                                         =
config.getLong(ProducerConfig.BUFFER_MEMORY_CONFIG);
        // 压缩配置，默认 none
        this.compressionType                                         =
CompressionType.forName(config.getString(ProducerConfig.COMPRESSI
ON_TYPE_CONFIG));

        this.maxBlockTimeMs                                          =
config.getLong(ProducerConfig.MAX_BLOCK_MS_CONFIG);
        int  deliveryTimeoutMs  =  configureDeliveryTimeout(config,
log);

        this.apiVersions = new ApiVersions();
        this.transactionManager = configureTransactionState(config,
logContext);
        // 上下文环境
        // 批次大下，默认 16k
        // 是否压缩，默认 none
        // linger.ms，默认值 0。
        // 重试间隔时间，默认值 100ms。
        // delivery.timeout.ms 默认值 2 分钟。
        // request.timeout.ms 默认值 30s。
        this.accumulator = new RecordAccumulator(logContext,
                config.getInt(ProducerConfig.BATCH_SIZE_CONFIG),
                this.compressionType,
                lingerMs(config),
                retryBackoffMs,
                deliveryTimeoutMs,
                metrics,
                PRODUCER_METRIC_GROUP_NAME,
                time,
                apiVersions,
                transactionManager,
                new                  BufferPool(this.totalMemorySize,
config.getInt(ProducerConfig.BATCH_SIZE_CONFIG),  metrics,  time,
PRODUCER_METRIC_GROUP_NAME));

        // Kafka 集群地址
        List<InetSocketAddress>              addresses              =
ClientUtils.parseAndValidateAddresses(

    config.getList(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG),

    config.getString(ProducerConfig.CLIENT_DNS_LOOKUP_CONFIG));

        // 从 Kafka 集群获取元数据
        if (metadata != null) {
            this.metadata = metadata;
        } else {
            // metadata.max.age.ms 默认值 5 分钟。生产者每隔多久需要更新一
```

```
下自己的元数据
            // metadata.max.idle.ms 默认值 5 分钟。网络最多空闲时间设置，超
过该阈值，就关闭该网络
            this.metadata = new ProducerMetadata(retryBackoffMs,

    config.getLong(ProducerConfig.METADATA_MAX_AGE_CONFIG),

    config.getLong(ProducerConfig.METADATA_MAX_IDLE_CONFIG),
                    logContext,
                    clusterResourceListeners,
                    Time.SYSTEM);
            this.metadata.bootstrap(addresses);
        }
        this.errors = this.metrics.sensor("errors");
        // 初始化 sender 线程
        this.sender     =     newSender(logContext,     kafkaClient,
this.metadata);
        String  ioThreadName  =  NETWORK_THREAD_PREFIX  +  "  |  "  +
clientId;
        // 启动发送线程
        this.ioThread = new KafkaThread(ioThreadName, this.sender,
true);
        this.ioThread.start();

        config.logUnused();
       AppInfoParser.registerAppInfo(JMX_PREFIX, clientId, metrics,
time.milliseconds());
        log.debug("Kafka producer started");
    } catch (Throwable t) {
        ... ...
    }
}
```

2.1.3 生产者 sender 线程初始化

点击 newSender()方法，查看发送线程初始化。

KafkaProducer.java

```
Sender newSender(LogContext logContext, KafkaClient kafkaClient,
ProducerMetadata metadata) {
    // 缓存的发送请求，默认值是 5。
    int                    maxInflightRequests                    =
configureInflightRequests(producerConfig);
    // request.timeout.ms 默认值 30s。
    int                    requestTimeoutMs                       =
producerConfig.getInt(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG);
    ChannelBuilder                 channelBuilder                 =
ClientUtils.createChannelBuilder(producerConfig,             time,
logContext);
    ProducerMetrics          metricsRegistry          =          new
ProducerMetrics(this.metrics);
    Sensor                   throttleTimeSensor                     =
Sender.throttleTimeSensor(metricsRegistry.senderMetrics);
    // maxInflightRequests 缓存的发送请求，默认值是 5。
    // reconnect.backoff.ms 默认值 50ms。重试时间间隔
    // reconnect.backoff.max.ms 默认值 1000ms。重试的总时间。每次重试失败
```

```
时，呈指数增加重试时间，直至达到此最大值
    // send.buffer.bytes 默认值 128k。 socket 发送数据的缓冲区大小
    // receive.buffer.bytes 默认值 32k。socket 接收数据的缓冲区大小
    // request.timeout.ms 默认值 30s。
    // socket.connection.setup.timeout.ms 默认值 10s。生产者和服务器通信连接建立的时间。如果在超时之前没有建立连接，将关闭通信。
    // socket.connection.setup.timeout.max.ms 默认值 30s。生产者和服务器通信，每次连续连接失败时，连接建立超时将呈指数增加，直至达到此最大值。
    KafkaClient client = kafkaClient != null ? kafkaClient : new NetworkClient(
            new Selector(producerConfig.getLong(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG),
                    this.metrics, time, "producer", channelBuilder, logContext),
            metadata,
            clientId,
            maxInflightRequests,

    producerConfig.getLong(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG),

    producerConfig.getLong(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG),

    producerConfig.getInt(ProducerConfig.SEND_BUFFER_CONFIG),

    producerConfig.getInt(ProducerConfig.RECEIVE_BUFFER_CONFIG),
            requestTimeoutMs,

    producerConfig.getLong(ProducerConfig.SOCKET_CONNECTION_SETUP_TIMEOUT_MS_CONFIG),

    producerConfig.getLong(ProducerConfig.SOCKET_CONNECTION_SETUP_TIMEOUT_MAX_MS_CONFIG),
            time,
            true,
            apiVersions,
            throttleTimeSensor,
            logContext);

    // acks 默认值是-1。
    // acks=0，生产者发送给 Kafka 服务器后，不需要应答
    // acks=1，生产者发送给 Kafka 服务器后，Leader 接收后应答
    // acks=-1（all），生产者发送给 Kafka 服务器后，Leader 和在 ISR 队列的所有 Follower 共同应答
    short acks = configureAcks(producerConfig, log);

    // max.request.size 默认值 1m。 生产者发往 Kafka 集群单条信息的最大值
    // retries 重试次数，默认值 Int 的最大值
    // retry.backoff.ms 默认值 100ms。重试时间间隔
    return new Sender(logContext,
            client,
            metadata,
            this.accumulator,
```

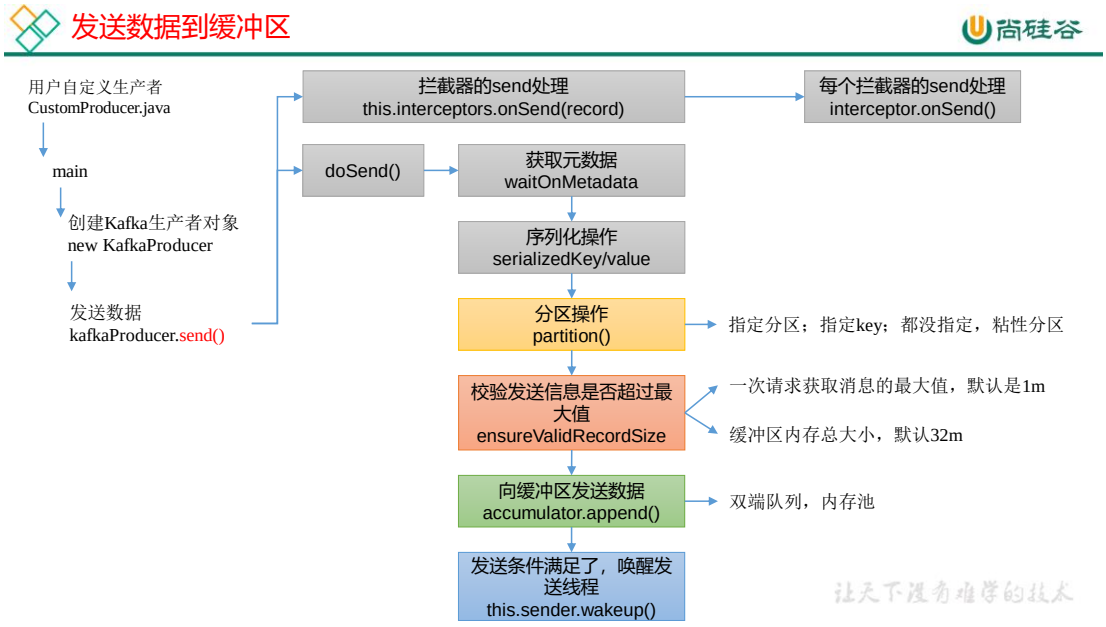
```
            maxInflightRequests == 1,

    producerConfig.getInt(ProducerConfig.MAX_REQUEST_SIZE_CONFIG),
            acks,
            producerConfig.getInt(ProducerConfig.RETRIES_CONFIG),
            metricsRegistry.senderMetrics,
            time,
            requestTimeoutMs,

    producerConfig.getLong(ProducerConfig.RETRY_BACKOFF_MS_CONFIG),
            this.transactionManager,
            apiVersions);
}
```

Sender 对象被放到了一个线程中启动，所有需要点击 newSender()方法中的 Sender，并找到 sender 对象中的 run()方法。

Sender.java

```
@Override
public void run() {
    log.debug("Starting Kafka producer I/O thread.");

    // main loop, runs until close is called
    while (running) {
        try {
            // sender 线程从缓冲区准备拉取数据，刚启动拉不到数据
            runOnce();
        } catch (Exception e) {
            log.error("Uncaught error in kafka producer I/O thread:
", e);
        }
    }

    ... ...
}
```

2.2 发送数据到缓冲区



2.2.1 发送总体流程

点击自己编写的 CustomProducer.java 中的 send()方法。

```
// 2 发送数据
for (int i = 0; i < 5; i++) {
    kafkaProducer.send(new ProducerRecord<>("first","atguigu"+i));
}
```

KafkaProducer.java

```
@Override
public Future<RecordMetadata> send(ProducerRecord<K, V> record) {
    return send(record, null);
}

@Override
public Future<RecordMetadata> send(ProducerRecord<K, V> record,
Callback callback) {
    // intercept the record, which can be potentially modified;
this method does not throw exceptions
    // 拦截器处理发送的数据
    ProducerRecord<K, V> interceptedRecord =
this.interceptors.onSend(record);
    return doSend(interceptedRecord, callback);
}
```

点击 onSend()方法，进行拦截器相关处理。

ProducerInterceptors.java

```
public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record) {
    ProducerRecord<K, V> interceptRecord = record;
   for (ProducerInterceptor<K, V> interceptor : this.interceptors)
{
       try {
           // 拦截器处理
           interceptRecord = interceptor.onSend(interceptRecord);
       } catch (Exception e) {
           // do not propagate interceptor exception, log and
continue calling other interceptors
           // be careful not to throw exception from here
           if (record != null)
               log.warn("Error executing interceptor onSend
callback for topic: {}, partition: {}", record.topic(),
record.partition(), e);
           else
               log.warn("Error executing interceptor onSend
callback", e);
       }
    }
    return interceptRecord;
}
```

从拦截器处理中返回，点击 doSend()方法。

KafkaProducer.java

```
private Future<RecordMetadata> doSend(ProducerRecord<K, V> record,
```

```
Callback callback) {
    TopicPartition tp = null;
    try {
        throwIfProducerClosed();
        // first make sure the metadata for the topic is available
        long nowMs = time.milliseconds();
        ClusterAndWaitTime clusterAndWaitTime;
        try {
            // 从 Kafka 拉取元数据。maxBlockTimeMs 表示最多能等待多长时间。
            clusterAndWaitTime    =    waitOnMetadata(record.topic(),
record.partition(), nowMs, maxBlockTimeMs);
        } catch (KafkaException e) {
            if (metadata.isClosed())
                throw new KafkaException("Producer closed while send
in progress", e);
            throw e;
        }
        nowMs += clusterAndWaitTime.waitedOnMetadataMs;
        // 剩余时间 = 最多能等待时间 - 用了多少时间;
        long   remainingWaitMs   =   Math.max(0,   maxBlockTimeMs   -
clusterAndWaitTime.waitedOnMetadataMs);
        // 更新集群元数据
        Cluster cluster = clusterAndWaitTime.cluster;

        // 序列化操作
        byte[] serializedKey;
        try {
            serializedKey = keySerializer.serialize(record.topic(),
record.headers(), record.key());
        } catch (ClassCastException cce) {
            throw new SerializationException("Can't convert key of
class " + record.key().getClass().getName() +
                    "          to          class          "          +
producerConfig.getClass(ProducerConfig.KEY_SERIALIZER_CLASS_CONFI
G).getName() +
                    " specified in key.serializer", cce);
        }
        byte[] serializedValue;
        try {
            serializedValue                                          =
valueSerializer.serialize(record.topic(),        record.headers(),
record.value());
        } catch (ClassCastException cce) {
            throw new SerializationException("Can't convert value
of class " + record.value().getClass().getName() +
                    "          to          class          "          +
producerConfig.getClass(ProducerConfig.VALUE_SERIALIZER_CLASS_CON
FIG).getName() +
                    " specified in value.serializer", cce);
        }

        // 分区操作（根据元数据信息）
        int    partition    =    partition(record,    serializedKey,
serializedValue, cluster);
        tp = new TopicPartition(record.topic(), partition);

        setReadOnly(record.headers());
```

```
        Header[] headers = record.headers().toArray();

        int                     serializedSize                     =
AbstractRecords.estimateSizeInBytesUpperBound(apiVersions.maxUsab
leProduceMagic(),
                compressionType,    serializedKey,    serializedValue,
headers);

        // 校验发送消息的大小是否超过最大值，默认是 1m
        ensureValidRecordSize(serializedSize);

        long  timestamp  =  record.timestamp()  ==  null  ?  nowMs  :
record.timestamp();
        if (log.isTraceEnabled()) {
            log.trace("Attempting to append record {} with callback
{} to topic {} partition {}", record, callback, record.topic(),
partition);
        }

        // 消息发送的回调函数
        // producer callback will make sure to call both 'callback'
and interceptor callback
        Callback          interceptCallback          =          new
InterceptorCallback<>(callback, this.interceptors, tp);

        if        (transactionManager        !=        null        &&
transactionManager.isTransactional()) {
            transactionManager.failIfNotReadyForSend();
        }

        // 内存，默认 32m，里面是默认 16k 一个批次
        RecordAccumulator.RecordAppendResult        result        =
accumulator.append(tp, timestamp, serializedKey,
                serializedValue,     headers,     interceptCallback,
remainingWaitMs, true, nowMs);

        if (result.abortForNewBatch) {
            int prevPartition = partition;
            partitioner.onNewBatch(record.topic(),          cluster,
prevPartition);
            partition     =     partition(record,     serializedKey,
serializedValue, cluster);
            tp = new TopicPartition(record.topic(), partition);
            if (log.isTraceEnabled()) {
                log.trace("Retrying append due to new batch creation
for  topic  {}  partition  {}.  The  old  partition  was  {}",
record.topic(), partition, prevPartition);
            }
            //  producer  callback  will  make  sure  to  call  both
'callback' and interceptor callback
            interceptCallback = new InterceptorCallback<>(callback,
this.interceptors, tp);

            result     =     accumulator.append(tp,     timestamp,
serializedKey,
                serializedValue,     headers,     interceptCallback,
remainingWaitMs, false, nowMs);
```

```
        }

        if (transactionManager != null &&
transactionManager.isTransactional())
            transactionManager.maybeAddPartitionToTransaction(tp);

        // 批次满了 或者  创建了一个新的批次，唤醒 sender 发送线程
        if (result.batchIsFull || result.newBatchCreated) {
            log.trace("Waking up the sender since topic {}
partition {} is either full or getting a new batch",
record.topic(), partition);
            this.sender.wakeup();
        }
        return result.future;
    } catch (ApiException e) {
        ... ...
    }
}
```

2.2.2 分区选择

KafkaProducer.java

详解默认分区规则。

```
int partition = partition(record, serializedKey, serializedValue,
cluster);
tp = new TopicPartition(record.topic(), partition);

private int partition(ProducerRecord<K, V> record, byte[]
serializedKey, byte[] serializedValue, Cluster cluster) {
    // 指定了分区，那就直接用该分区号
    Integer partition = record.partition();
    return partition != null ?
            partition :
            // 分区器选择分区
            partitioner.partition(
                    record.topic(), record.key(), serializedKey,
record.value(), serializedValue, cluster);
}
```

点击 partition，跳转到 Partitioner 接口。选中 partition，点击 ctrl+ h，查找接口实现类

```
int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster);
```

选择默认的分区器 DefaultPartitioner

```
public int partition(String topic, Object key, byte[] keyBytes,
Object value, byte[] valueBytes, Cluster cluster,
                     int numPartitions) {
    // 没有指定 key:
    if (keyBytes == null) {
        return stickyPartitionCache.partition(topic, cluster);
    }
    // 指定 key:按照 key 的 hash 值对分区取模
    // hash the keyBytes to choose a partition
    return Utils.toPositive(Utils.murmur2(keyBytes)) %
numPartitions;
```

```
}

// 没有指定 key 和分区的处理方式
public int partition(String topic, Cluster cluster) {
    Integer part = indexCache.get(topic);
    if (part == null) {
        return nextPartition(topic, cluster, -1);
    }
    return part;
}

public int nextPartition(String topic, Cluster cluster, int
prevPartition) {
    List<PartitionInfo>                partitions                =
cluster.partitionsForTopic(topic);
    Integer oldPart = indexCache.get(topic);
    Integer newPart = oldPart;
    // Check that the current sticky partition for the topic is
either not set or that the partition that
    // triggered the new batch matches the sticky partition that
needs to be changed.
    if (oldPart == null || oldPart == prevPartition) {
        List<PartitionInfo>          availablePartitions          =
cluster.availablePartitionsForTopic(topic);
        if (availablePartitions.size() < 1) {
            Integer                      random                   =
Utils.toPositive(ThreadLocalRandom.current().nextInt());
            newPart = random % partitions.size();
        } else if (availablePartitions.size() == 1) {
            newPart = availablePartitions.get(0).partition();
        } else {
            while (newPart == null || newPart.equals(oldPart)) {
                int                      random                   =
Utils.toPositive(ThreadLocalRandom.current().nextInt());
                newPart     =     availablePartitions.get(random     %
availablePartitions.size()).partition();
            }
        }
        // Only change the sticky partition if it is null or
prevPartition matches the current sticky partition.
        if (oldPart == null) {
            indexCache.putIfAbsent(topic, newPart);
        } else {
            indexCache.replace(topic, prevPartition, newPart);
        }
        return indexCache.get(topic);
    }
    return indexCache.get(topic);
}
```

2.2.3 发送消息大小校验

KafkaProducer.java

详解缓冲区大小

```
ensureValidRecordSize(serializedSize);
```

```
private void ensureValidRecordSize(int size) {
    // 一次请求获取消息的最大值，默认是 1m
    if (size > maxRequestSize)
        throw new RecordTooLargeException("The message is " + size
+             " bytes when serialized which is larger than " +
maxRequestSize + ", which is the value of the " +
              ProducerConfig.MAX_REQUEST_SIZE_CONFIG      +      "
configuration.");
    // 缓冲区内存总大小，默认 32m
    if (size > totalMemorySize)
        throw new RecordTooLargeException("The message is " + size
+
              " bytes when serialized which is larger than the
total memory buffer you have configured with the " +
              ProducerConfig.BUFFER_MEMORY_CONFIG +
              " configuration.");
}
```

2.2.4 内存池

KafkaProducer.java

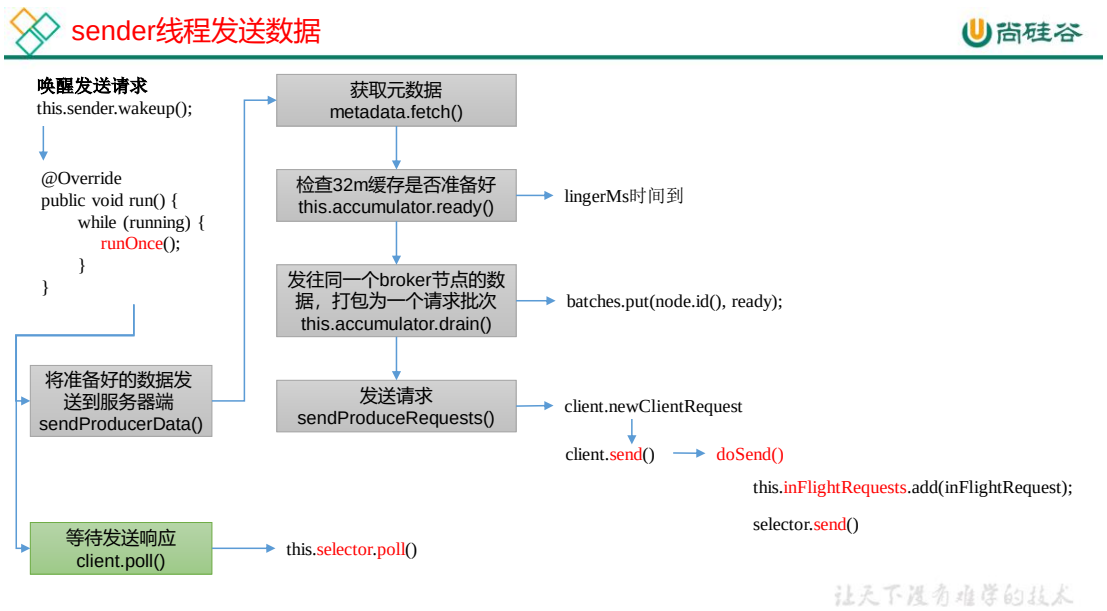
详解内存池。

```
RecordAccumulator.RecordAppendResult            result            =
accumulator.append(tp,          timestamp,          serializedKey,
        serializedValue,        headers,        interceptCallback,
remainingWaitMs, true, nowMs);


public RecordAppendResult append(TopicPartition tp,
                             long timestamp,
                             byte[] key,
                             byte[] value,
                             Header[] headers,
                             Callback callback,
                             long maxTimeToBlock,
                             boolean abortOnNewBatch,
                             long           nowMs)           throws
InterruptedException {
    // We keep track of the number of appending thread to make
sure we do not miss batches in
    // abortIncompleteBatches().
    appendsInProgress.incrementAndGet();
    ByteBuffer buffer = null;
    if (headers == null) headers = Record.EMPTY_HEADERS;
    try {
        // 每个分区，创建或者获取一个队列
        // check if we have an in-progress batch
        Deque<ProducerBatch> dq = getOrCreateDeque(tp);
        synchronized (dq) {
            if (closed)
                throw new KafkaException("Producer closed while send
in progress");
            // 尝试向队列里面添加数据（没有分配内存、批次对象，所以失败）
            RecordAppendResult appendResult = tryAppend(timestamp,
key, value, headers, callback, dq, nowMs);
```

```
            if (appendResult != null)
                return appendResult;
        }

        // we don't have an in-progress record batch try to
allocate a new batch
        if (abortOnNewBatch) {
            // Return a result that will cause another call to
append.
          return new RecordAppendResult(null, false, false, true);
        }

        byte maxUsableMagic = apiVersions.maxUsableProduceMagic();
        // 取批次大小（默认 16k）和消息大小的最大值（上限默认 1m）。这样设计的
主要原因是有可能一条消息的大小大于批次大小。
        int size = Math.max(this.batchSize,
AbstractRecords.estimateSizeInBytesUpperBound(maxUsableMagic,
compression, key, value, headers));

        log.trace("Allocating a new {} byte message buffer for
topic {} partition {} with remaining timeout {}ms", size,
tp.topic(), tp.partition(), maxTimeToBlock);
        // 根据批次大小（默认 16k）和消息大小中最大值，分配内存
        buffer = free.allocate(size, maxTimeToBlock);

        // Update the current time in case the buffer allocation
blocked above.
        nowMs = time.milliseconds();
        synchronized (dq) {
            // Need to check if producer is closed again after
grabbing the dequeue lock.
            if (closed)
                throw new KafkaException("Producer closed while send
in progress");
            // 尝试向队列里面添加数据（有内存，但是没有批次对象）
            RecordAppendResult appendResult = tryAppend(timestamp,
key, value, headers, callback, dq, nowMs);
            if (appendResult != null) {
                // Somebody else found us a batch, return the one we
waited for! Hopefully this doesn't happen often...
                return appendResult;
            }

            MemoryRecordsBuilder recordsBuilder =
recordsBuilder(buffer, maxUsableMagic);
            // 根据内存大小封装批次（有内存、有批次对象）
            ProducerBatch batch = new ProducerBatch(tp,
recordsBuilder, nowMs);
            // 尝试向队列里面添加数据
            FutureRecordMetadata future =
Objects.requireNonNull(batch.tryAppend(timestamp, key, value,
headers,callback, nowMs));

            // 把新创建的批次放到队列末尾
            dq.addLast(batch);
            incomplete.add(batch);
```

```
            // Don't deallocate this buffer in the finally block as
it's being used in the record batch
            buffer = null;
            return new RecordAppendResult(future, dq.size() > 1 ||
batch.isFull(), true, false);
        }
    } finally {
        // 如果发生异常，释放内存
        if (buffer != null)
            free.deallocate(buffer);
        appendsInProgress.decrementAndGet();
    }
}
```

2.3 sender 线程发送数据



KafkaProducer.java

详解发送线程。

```
if (result.batchIsFull || result.newBatchCreated) {
    log.trace("Waking up the sender since topic {} partition {}
is either full or getting a new batch", record.topic(),
partition);
    this.sender.wakeup();
}
```

进入 sender 发送线程的 run()方法。

```
@Override
public void run() {
    log.debug("Starting Kafka producer I/O thread.");

    // main loop, runs until close is called
    while (running) {
        try {
            runOnce();
        } catch (Exception e) {
```

```
            log.error("Uncaught error in kafka producer I/O thread:
", e);
        }
    }
    ... ...
}

void runOnce() {
    // 如果是事务操作，按照如下处理
    if (transactionManager != null) {
        try {
            transactionManager.maybeResolveSequences();

            // do not continue sending if the transaction manager
is in a failed state
            if (transactionManager.hasFatalError()) {
                RuntimeException             lastError             =
transactionManager.lastError();
                if (lastError != null)
                    maybeAbortBatches(lastError);
                client.poll(retryBackoffMs, time.milliseconds());
                return;
            }

            // Check whether we need a new producerId. If so, we
will enqueue an InitProducerId
            // request which will be sent below

    transactionManager.bumpIdempotentEpochAndResetIdIfNeeded();

            if (maybeSendAndPollTransactionalRequest()) {
                return;
            }
        } catch (AuthenticationException e) {
            // This is already logged as error, but propagated here
to perform any clean ups.
            log.trace("Authentication  exception  while  processing
transactional request", e);
            transactionManager.authenticationFailed(e);
        }
    }

    long currentTimeMs = time.milliseconds();
    // 将准备好的数据发送到服务器端
    long pollTimeout = sendProducerData(currentTimeMs);
    // 等待发送响应
    client.poll(pollTimeout, currentTimeMs);
}

// 获取要发送数据的细节
private long sendProducerData(long now) {
    // 获取元数据
    Cluster cluster = metadata.fetch();
    // get the list of partitions with data ready to send
    // 1、检查 32m 缓存是否准备好（linger.ms）
    RecordAccumulator.ReadyCheckResult          result          =
this.accumulator.ready(cluster, now);
```

```
    // 如果 Leader 信息不知道，是不能发送数据的
   // if there are any partitions whose leaders are not known yet,
force metadata update
    if (!result.unknownLeaderTopics.isEmpty()) {
        // The set of topics with unknown leader contains topics
with leader election pending as well as
        // topics which may have expired. Add the topic again to
metadata to ensure it is included
        // and request metadata update, since there are messages to
send to the topic.
        for (String topic : result.unknownLeaderTopics)
            this.metadata.add(topic, now);

        log.debug("Requesting metadata update due to unknown leader
topics from the batched records: {}",
            result.unknownLeaderTopics);
        this.metadata.requestUpdate();
    }

    // remove any nodes we aren't ready to send to
    // 删除掉没有准备好发送的数据
    Iterator<Node> iter = result.readyNodes.iterator();
    long notReadyTimeout = Long.MAX_VALUE;
    while (iter.hasNext()) {
        Node node = iter.next();
        if (!this.client.ready(node, now)) {
            iter.remove();
            notReadyTimeout        =        Math.min(notReadyTimeout,
this.client.pollDelayMs(node, now));
        }
    }

    // create produce requests
    // 2、发往同一个 broker 节点的数据，打包为一个请求批次
    Map<Integer,        List<ProducerBatch>>        batches        =
this.accumulator.drain(cluster,                result.readyNodes,
this.maxRequestSize, now);
    addToInflightBatches(batches);
    if (guaranteeMessageOrder) {
        // Mute all the partitions drained
        for (List<ProducerBatch> batchList : batches.values()) {
            for (ProducerBatch batch : batchList)
               this.accumulator.mutePartition(batch.topicPartition);
        }
    }

    accumulator.resetNextBatchExpiryTime();
    List<ProducerBatch>          expiredInflightBatches          =
getExpiredInflightBatches(now);
    List<ProducerBatch>              expiredBatches              =
this.accumulator.expiredBatches(now);
    expiredBatches.addAll(expiredInflightBatches);

    // Reset the producer id if an expired batch has previously
been sent to the broker. Also update the metrics
    //   for   expired   batches.   see   the   documentation   of
```

```
@TransactionState.resetIdempotentProducerId to understand why
    // we need to reset the producer id here.
    if (!expiredBatches.isEmpty())
        log.trace("Expired {} batches in accumulator",
expiredBatches.size());
    for (ProducerBatch expiredBatch : expiredBatches) {
        String errorMessage = "Expiring " +
expiredBatch.recordCount + " record(s) for " +
expiredBatch.topicPartition
            + ":" + (now - expiredBatch.createdMs) + " ms has
passed since batch creation";
        failBatch(expiredBatch, new TimeoutException(errorMessage),
false);
        if (transactionManager != null && expiredBatch.inRetry()) {
            // This ensures that no new batches are drained until
the current in flight batches are fully resolved.
            transactionManager.markSequenceUnresolved(expiredBatch);
        }
    }
    sensors.updateProduceRequestMetrics(batches);

    // If we have any nodes that are ready to send + have sendable
data, poll with 0 timeout so this can immediately
    // loop and try sending more data. Otherwise, the timeout will
be the smaller value between next batch expiry
    // time, and the delay time for checking data availability.
Note that the nodes may have data that isn't yet
    // sendable due to lingering, backing off, etc. This
specifically does not include nodes with sendable data
    // that aren't ready to send since they would cause busy
looping.
    long pollTimeout = Math.min(result.nextReadyCheckDelayMs,
notReadyTimeout);
    pollTimeout = Math.min(pollTimeout,
this.accumulator.nextExpiryTimeMs() - now);
    pollTimeout = Math.max(pollTimeout, 0);
    if (!result.readyNodes.isEmpty()) {
        log.trace("Nodes with data ready to send: {}",
result.readyNodes);
        // if some partitions are already ready to be sent, the
select time would be 0;
        // otherwise if some partition already has some data
accumulated but not ready yet,
        // the select time will be the time difference between now
and its linger expiry time;
        // otherwise the select time will be the time difference
between now and the metadata expiry time;
        pollTimeout = 0;
    }
    // 3、发送请求
    sendProduceRequests(batches, now);
    return pollTimeout;
}

    // 1、检查 32m 缓存是否准备好（linger.ms）
public ReadyCheckResult ready(Cluster cluster, long nowMs) {
    Set<Node> readyNodes = new HashSet<>();
```

```
    long nextReadyCheckDelayMs = Long.MAX_VALUE;
    Set<String> unknownLeaderTopics = new HashSet<>();

    boolean exhausted = this.free.queued() > 0;
    for (Map.Entry<TopicPartition, Deque<ProducerBatch>> entry :
this.batches.entrySet()) {
        Deque<ProducerBatch> deque = entry.getValue();
        synchronized (deque) {
            // When producing to a large number of partitions, this
path is hot and deques are often empty.
            // We check whether a batch exists first to avoid the
more expensive checks whenever possible.
            ProducerBatch batch = deque.peekFirst();
            if (batch != null) {
                TopicPartition part = entry.getKey();
                Node leader = cluster.leaderFor(part);
                if (leader == null) {
                    // This is a partition for which leader is not
known, but messages are available to send.
                    // Note that entries are currently not removed
from batches when deque is empty.
                    unknownLeaderTopics.add(part.topic());
                } else if (!readyNodes.contains(leader)
&& !isMuted(part)) {
                    long waitedTimeMs = batch.waitedTimeMs(nowMs);
                    // 如果不是第一次拉取该批次数据，且等待时间没有超过重试时
间，backingOff=true
                    boolean backingOff = batch.attempts() > 0 &&
waitedTimeMs < retryBackoffMs;
                    // 如果 backingOff=true，返回重试时间，如果不是重试，选
择 lingerMs
                    long timeToWaitMs = backingOff ? retryBackoffMs :
lingerMs;
                   boolean full = deque.size() > 1 || batch.isFull();
                    // 如果等待的时间超过了 timeToWaitMs，expired=true，表
示可以发送数据
                    boolean expired = waitedTimeMs >= timeToWaitMs;
                    boolean transactionCompleting =
transactionManager != null && transactionManager.isCompleting();
                    boolean sendable = full
                        || expired
                        || exhausted
                        || closed
                        || flushInProgress()
                        || transactionCompleting;
                    if (sendable && !backingOff) {
                        readyNodes.add(leader);
                    } else {
                        long timeLeftMs = Math.max(timeToWaitMs -
waitedTimeMs, 0);
                        // Note that this results in a conservative
estimate since an un-sendable partition may have
                        // a leader that will later be found to have
sendable data. However, this is good enough
                        // since we'll just wake up and then sleep
again for the remaining time.
                        nextReadyCheckDelayMs = Math.min(timeLeftMs,
```

```
nextReadyCheckDelayMs);
                    }
                }
            }
        }
    }
    return new ReadyCheckResult(readyNodes, nextReadyCheckDelayMs,
unknownLeaderTopics);
}


// 2、发往同一个 broker 节点的数据，打包为一个请求批次。
public Map<Integer, List<ProducerBatch>> drain(Cluster cluster,
Set<Node> nodes, int maxSize, long now) {
    if (nodes.isEmpty())
        return Collections.emptyMap();

    Map<Integer, List<ProducerBatch>> batches = new HashMap<>();
    for (Node node : nodes) {
        List<ProducerBatch> ready = drainBatchesForOneNode(cluster,
node, maxSize, now);
        batches.put(node.id(), ready);
    }
    return batches;
}

// 3、发送请求
private void sendProduceRequest(long now, int destination, short
acks, int timeout, List<ProducerBatch> batches) {
    if (batches.isEmpty())
        return;

    final Map<TopicPartition, ProducerBatch> recordsByPartition =
new HashMap<>(batches.size());

    // find the minimum magic version used when creating the
record sets
    byte minUsedMagic = apiVersions.maxUsableProduceMagic();
    for (ProducerBatch batch : batches) {
        if (batch.magic() < minUsedMagic)
            minUsedMagic = batch.magic();
    }
    ProduceRequestData.TopicProduceDataCollection tpd = new
ProduceRequestData.TopicProduceDataCollection();
    for (ProducerBatch batch : batches) {
        TopicPartition tp = batch.topicPartition;
        MemoryRecords records = batch.records();

        // down convert if necessary to the minimum magic used. In
general, there can be a delay between the time
        // that the producer starts building the batch and the time
that we send the request, and we may have
        // chosen the message format based on out-dated metadata.
In the worst case, we optimistically chose to use
        // the new message format, but found that the broker didn't
support it, so we need to down-convert on the
        // client before sending. This is intended to handle edge
```

```
cases around cluster upgrades where brokers may
       // not all support the same message format version. For
example, if a partition migrates from a broker
       // which is supporting the new magic version to one which
doesn't, then we will need to convert.
       if (!records.hasMatchingMagic(minUsedMagic))
          records = batch.records().downConvert(minUsedMagic, 0,
time).records();
       ProduceRequestData.TopicProduceData tpData =
tpd.find(tp.topic());
       if (tpData == null) {
          tpData = new
ProduceRequestData.TopicProduceData().setName(tp.topic());
          tpd.add(tpData);
       }
       tpData.partitionData().add(new
ProduceRequestData.PartitionProduceData()
             .setIndex(tp.partition())
             .setRecords(records));
       recordsByPartition.put(tp, batch);
    }

    String transactionalId = null;
    if (transactionManager != null &&
transactionManager.isTransactional()) {
       transactionalId = transactionManager.transactionalId();
    }

    ProduceRequest.Builder requestBuilder =
ProduceRequest.forMagic(minUsedMagic,
          new ProduceRequestData()
                .setAcks(acks)
                .setTimeoutMs(timeout)
                .setTransactionalId(transactionalId)
                .setTopicData(tpd));
    RequestCompletionHandler callback = response ->
handleProduceResponse(response, recordsByPartition,
time.milliseconds());

    String nodeId = Integer.toString(destination);
    // 创建发送请求对象
    ClientRequest clientRequest = client.newClientRequest(nodeId,
requestBuilder, now, acks != 0,
          requestTimeoutMs, callback);
    // 发送请求
    client.send(clientRequest, now);
    log.trace("Sent produce request to {}: {}", nodeId,
requestBuilder);
}
    // 选中 send，点击 ctrl + alt + b
@Override
public void send(ClientRequest request, long now) {
    doSend(request, false, now);
}

private void doSend(ClientRequest clientRequest, boolean
isInternalRequest, long now) {
```

```
    ensureActive();
    String nodeId = clientRequest.destination();
    if (!isInternalRequest) {
        // If this request came from outside the NetworkClient,
validate
        // that we can send data.  If the request is internal, we
trust
        // that internal code has done this validation.  Validation
        // will be slightly different for some internal requests
(for
        // example, ApiVersionsRequests can be sent prior to being
in
        // READY state.)
        if (!canSendRequest(nodeId, now))
            throw new IllegalStateException("Attempt to send a
request to node " + nodeId + " which is not ready.");
    }
    AbstractRequest.Builder<?>               builder              =
clientRequest.requestBuilder();
    try {
        NodeApiVersions versionInfo = apiVersions.get(nodeId);
        short version;
        // Note: if versionInfo is null, we have no server version
information. This would be
        // the case when sending the initial ApiVersionRequest
which fetches the version
        // information itself.    It is also the case when
discoverBrokerVersions is set to false.
        if (versionInfo == null) {
            version = builder.latestAllowedVersion();
            if (discoverBrokerVersions && log.isTraceEnabled())
                log.trace("No version information found when sending
{} with correlation id {} to node {}. " +
                        "Assuming            version            {}.",
clientRequest.apiKey(),   clientRequest.correlationId(),   nodeId,
version);
        } else {
            version                                                =
versionInfo.latestUsableVersion(clientRequest.apiKey(),
builder.oldestAllowedVersion(),
                    builder.latestAllowedVersion());
        }
        //    The    call    to    build    may    also    throw
UnsupportedVersionException, if there are essential
        // fields that cannot be represented in the chosen version.
        // 发送请求
        doSend(clientRequest,        isInternalRequest,        now,
builder.build(version));
    }              catch              (UnsupportedVersionException
unsupportedVersionException) {
        // If the version is not supported, skip sending the
request over the wire.
        // Instead, simply add it to the local queue of aborted
requests.
        log.debug("Version mismatch when attempting to send {} with
correlation id {} to {}", builder,
                clientRequest.correlationId(),
```

```
clientRequest.destination(), unsupportedVersionException);
       ClientResponse         clientResponse         =         new
ClientResponse(clientRequest.makeHeader(builder.latestAllowedVers
ion()),
              clientRequest.callback(),
clientRequest.destination(), now, now,
              false, unsupportedVersionException, null, null);

       if (!isInternalRequest)
          abortedSends.add(clientResponse);
       else if (clientRequest.apiKey() == ApiKeys.METADATA)
          metadataUpdater.handleFailedRequest(now,
Optional.of(unsupportedVersionException));
    }
}

private    void    doSend(ClientRequest    clientRequest,    boolean
isInternalRequest, long now, AbstractRequest request) {
    String destination = clientRequest.destination();
    RequestHeader                       header                       =
clientRequest.makeHeader(request.version());
    if (log.isDebugEnabled()) {
       log.debug("Sending {} request with header {} and timeout {}
to node {}: {}",
          clientRequest.apiKey(),                            header,
clientRequest.requestTimeoutMs(), destination, request);
    }
    Send send = request.toSend(header);
    InFlightRequest inFlightRequest = new InFlightRequest(
          clientRequest,
          header,
          isInternalRequest,
          request,
          send,
          now);
    // 添加请求到 inflint
    this.inFlightRequests.add(inFlightRequest);
    // 发送数据
    selector.send(new      NetworkSend(clientRequest.destination(),
send));
}


// 获取服务器端响应
client.poll(pollTimeout, currentTimeMs);

@Override
public List<ClientResponse> poll(long timeout, long now) {
    ensureActive();

    if (!abortedSends.isEmpty()) {
       // If there are aborted sends because of unsupported
version exceptions or disconnects,
       // handle them immediately without waiting for
Selector#poll.
       List<ClientResponse> responses = new ArrayList<>();
       handleAbortedSends(responses);
```

```
        completeResponses(responses);
        return responses;
    }

    long metadataTimeout = metadataUpdater.maybeUpdate(now);
    try {
        this.selector.poll(Utils.min(timeout,      metadataTimeout,
defaultRequestTimeoutMs));
    } catch (IOException e) {
        log.error("Unexpected error during I/O", e);
    }

    // process completed actions
    // 获取发送后的响应
    long updatedNow = this.time.milliseconds();
    List<ClientResponse> responses = new ArrayList<>();
    handleCompletedSends(responses, updatedNow);
    handleCompletedReceives(responses, updatedNow);
    handleDisconnections(responses, updatedNow);
    handleConnections();
    handleInitiateApiVersionRequests(updatedNow);
    handleTimedOutConnections(responses, updatedNow);
    handleTimedOutRequests(responses, updatedNow);
    completeResponses(responses);

    return responses;
}
```

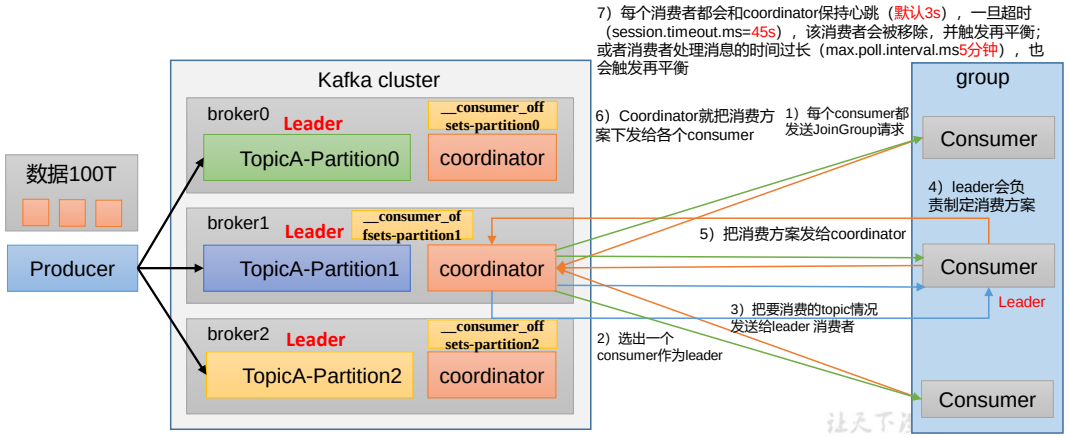
第 3 章 消费者源码

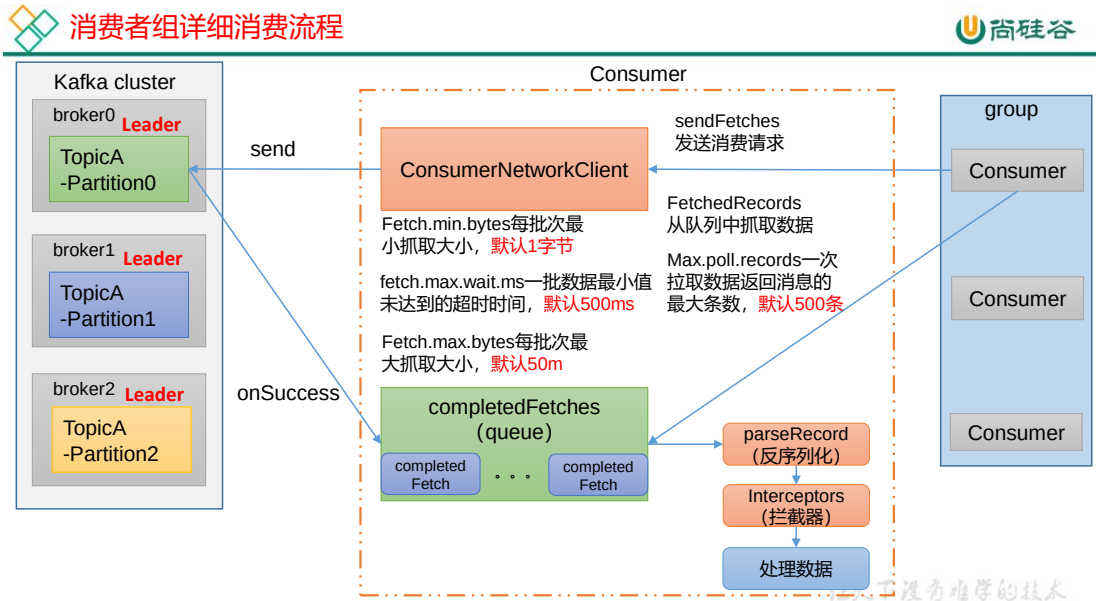
消费者组初始化流程

1、coordinator：辅助实现消费者组的初始化和分区的分配。

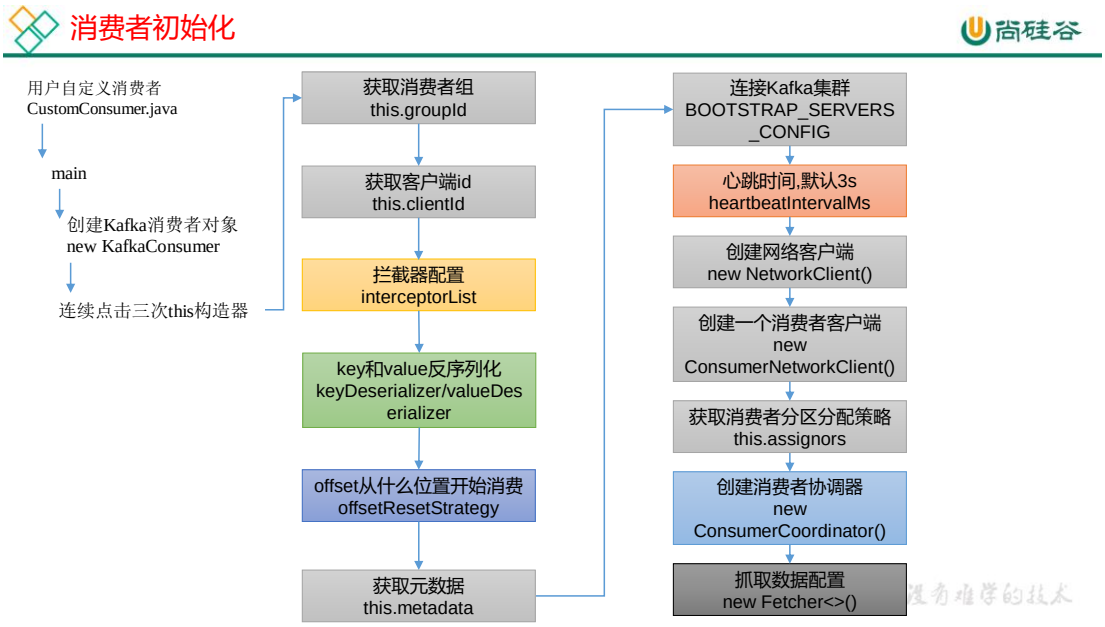
coordinator节点选择 = groupid的hashcode值 % 50（__consumer_offsets的分区数量）

例如：groupid的hashcode值 = 1，1% 50 = 1，那么__consumer_offsets 主题的1号分区，在哪个broker上，就选择这个节点的coordinator作为这个消费者组的老大。消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset。





3.1 初始化



3.1.1 程序入口

1）从用户自己编写的 main 方法开始阅读

```
package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;
```

```
public class CustomConsumer {

    public static void main(String[] args) {

        // 0 配置
        Properties properties = new Properties();

        // 连接 bootstrap.servers

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop102
:9092,hadoop103:9092");

        // 反序列化

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        // 配置消费者组 id
        properties.put(ConsumerConfig.GROUP_ID_CONFIG,"test");

        // 手动提交

properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,false);

        // 1 创建一个消费者  "", "hello"
        KafkaConsumer<String,   String>   kafkaConsumer   =   new
KafkaConsumer<>(properties);

        // 2 订阅主题 first
        ArrayList<String> topics = new ArrayList<>();
        topics.add("first");
        kafkaConsumer.subscribe(topics);

        // 3 消费数据
        while (true){

            ConsumerRecords<String,   String>   consumerRecords   =
kafkaConsumer.poll(Duration.ofSeconds(1));

            for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
                System.out.println(consumerRecord);
            }

            // 手动提交 offset
//            kafkaConsumer.commitSync();
            kafkaConsumer.commitAsync();
        }
    }
}
```

3.1.2 消费者初始化

点击 main()方法中的 KafkaConsumer ()。

KafkaConsumer.java

```
public KafkaConsumer(Properties properties) {
    this(properties, null, null);
}

public KafkaConsumer(Properties properties,
                     Deserializer<K> keyDeserializer,
                     Deserializer<V> valueDeserializer) {
    this(Utils.propsToMap(properties),             keyDeserializer,
valueDeserializer);
}

public KafkaConsumer(Map<String, Object> configs,
                     Deserializer<K> keyDeserializer,
                     Deserializer<V> valueDeserializer) {
    this(new
ConsumerConfig(ConsumerConfig.appendDeserializerToConfig(configs,
keyDeserializer, valueDeserializer)),
          keyDeserializer, valueDeserializer);
}
```

跳转到 KafkaConsumer 构造方法。

```
KafkaConsumer(ConsumerConfig        config,        Deserializer<K>
keyDeserializer, Deserializer<V> valueDeserializer) {
    try {
       GroupRebalanceConfig     groupRebalanceConfig     =     new
GroupRebalanceConfig(config,
             GroupRebalanceConfig.ProtocolType.CONSUMER);
       // 获取消费者组 id 和客户端 id
       this.groupId                                              =
Optional.ofNullable(groupRebalanceConfig.groupId);
       this.clientId                                             =
config.getString(CommonClientConfigs.CLIENT_ID_CONFIG);

       LogContext logContext;

       // If group.instance.id is set, we will append it to the
log context.
       if (groupRebalanceConfig.groupInstanceId.isPresent()) {
          logContext = new LogContext("[Consumer instanceId=" +
groupRebalanceConfig.groupInstanceId.get() +
                ", clientId=" + clientId + ", groupId=" +
groupId.orElse("null") + "] ");
       } else {
          logContext = new LogContext("[Consumer clientId=" +
clientId + ", groupId=" + groupId.orElse("null") + "] ");
       }

       this.log = logContext.logger(getClass());
       boolean                  enableAutoCommit                 =
config.maybeOverrideEnableAutoCommit();
       groupId.ifPresent(groupIdStr -> {
```

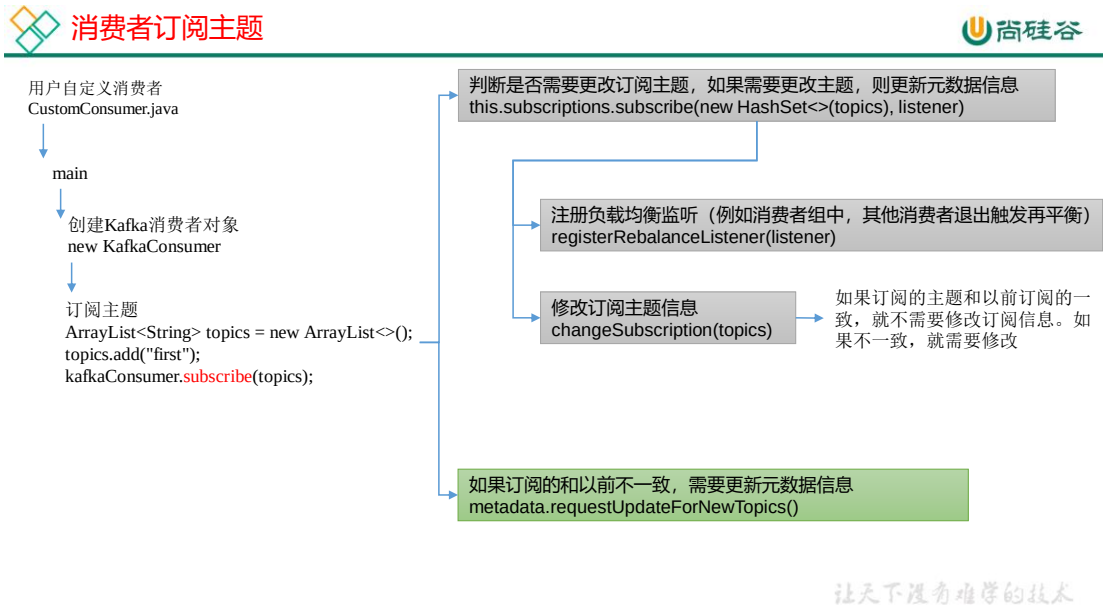
```
            if (groupIdStr.isEmpty()) {
                log.warn("Support for using the empty group id by
consumers is deprecated and will be removed in the next major
release.");
            }
        });

        log.debug("Initializing the Kafka consumer");
        // 等待服务端响应的最大等待时间，默认是 30s
        this.requestTimeoutMs                                    =
config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG);
        this.defaultApiTimeoutMs                                 =
config.getInt(ConsumerConfig.DEFAULT_API_TIMEOUT_MS_CONFIG);
        this.time = Time.SYSTEM;
        this.metrics = buildMetrics(config, time, clientId);
        // 重试时间间隔
        this.retryBackoffMs                                      =
config.getLong(ConsumerConfig.RETRY_BACKOFF_MS_CONFIG);

        // 拦截器配置
        List<ConsumerInterceptor<K,  V>>  interceptorList  =  (List)
config.getConfiguredInstances(
                ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
                ConsumerInterceptor.class,

    Collections.singletonMap(ConsumerConfig.CLIENT_ID_CONFIG,
clientId));
        this.interceptors                    =                   new
ConsumerInterceptors<>(interceptorList);

        // key 和 value 反序列化配置
        if (keyDeserializer == null) {
            this.keyDeserializer                                 =
config.getConfiguredInstance(ConsumerConfig.KEY_DESERIALIZER_CLAS
S_CONFIG, Deserializer.class);

    this.keyDeserializer.configure(config.originals(Collections.si
ngletonMap(ConsumerConfig.CLIENT_ID_CONFIG, clientId)), true);
        } else {

    config.ignore(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG);
            this.keyDeserializer = keyDeserializer;
        }
        if (valueDeserializer == null) {
            this.valueDeserializer                               =
config.getConfiguredInstance(ConsumerConfig.VALUE_DESERIALIZER_CL
ASS_CONFIG, Deserializer.class);

    this.valueDeserializer.configure(config.originals(Collections.
singletonMap(ConsumerConfig.CLIENT_ID_CONFIG, clientId)), false);
        } else {

    config.ignore(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG);
            this.valueDeserializer = valueDeserializer;
        }
        // offset 从什么位置开始消费，默认是 latest
        OffsetResetStrategy          offsetResetStrategy          =
```

```
OffsetResetStrategy.valueOf(config.getString(ConsumerConfig.AUTO_
OFFSET_RESET_CONFIG).toUpperCase(Locale.ROOT));
        this.subscriptions   =   new   SubscriptionState(logContext,
offsetResetStrategy);
        ClusterResourceListeners     clusterResourceListeners     =
configureClusterResourceListeners(keyDeserializer,
                valueDeserializer,              metrics.reporters(),
interceptorList);
        // 获取元数据
        // 配置是否可以消费系统主题数据
        // 配置是否允许自动创建主题
        this.metadata = new ConsumerMetadata(retryBackoffMs,

    config.getLong(ConsumerConfig.METADATA_MAX_AGE_CONFIG),

    !config.getBoolean(ConsumerConfig.EXCLUDE_INTERNAL_TOPICS_CONF
IG),

    config.getBoolean(ConsumerConfig.ALLOW_AUTO_CREATE_TOPICS_CONF
IG),
                subscriptions, logContext, clusterResourceListeners);
        // 配置连接 Kafka 集群
        List<InetSocketAddress>              addresses              =
ClientUtils.parseAndValidateAddresses(

    config.getList(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG),
config.getString(ConsumerConfig.CLIENT_DNS_LOOKUP_CONFIG));
        this.metadata.bootstrap(addresses);
        String metricGrpPrefix = "consumer";

        FetcherMetricsRegistry      metricsRegistry      =      new
FetcherMetricsRegistry(Collections.singleton(CLIENT_ID_METRIC_TAG
), metricGrpPrefix);
        ChannelBuilder                channelBuilder                =
ClientUtils.createChannelBuilder(config, time, logContext);
        this.isolationLevel = IsolationLevel.valueOf(

    config.getString(ConsumerConfig.ISOLATION_LEVEL_CONFIG).toUppe
rCase(Locale.ROOT));
        Sensor                  throttleTimeSensor                  =
Fetcher.throttleTimeSensor(metrics, metricsRegistry);
        // 心跳时间,默认 3s
        int                   heartbeatIntervalMs                   =
config.getInt(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG);

        ApiVersions apiVersions = new ApiVersions();
        // 创建网络客户端
        NetworkClient netClient = new NetworkClient(
                new
Selector(config.getLong(ConsumerConfig.CONNECTIONS_MAX_IDLE_MS_CO
NFIG),    metrics,    time,    metricGrpPrefix,    channelBuilder,
logContext),
                this.metadata,
                clientId,
                100, // a fixed large enough value will suffice for
max in-flight requests
```

```
    config.getLong(ConsumerConfig.RECONNECT_BACKOFF_MS_CONFIG),

    config.getLong(ConsumerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG),
                config.getInt(ConsumerConfig.SEND_BUFFER_CONFIG),
                config.getInt(ConsumerConfig.RECEIVE_BUFFER_CONFIG),

    config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG),

    config.getLong(ConsumerConfig.SOCKET_CONNECTION_SETUP_TIMEOUT_
MS_CONFIG),

    config.getLong(ConsumerConfig.SOCKET_CONNECTION_SETUP_TIMEOUT_
MAX_MS_CONFIG),
                time,
                true,
                apiVersions,
                throttleTimeSensor,
                logContext);
        // 创建一个消费者客户端
        this.client = new ConsumerNetworkClient(
                logContext,
                netClient,
                metadata,
                time,
                retryBackoffMs,

    config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG),
                heartbeatIntervalMs);  //Will  avoid  blocking  an
extended period of time to prevent heartbeat thread starvation
        // 获取消费者分区分配策略
        this.assignors                                          =
ConsumerPartitionAssignor.getAssignorInstances(

    config.getList(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CO
NFIG),

    config.originals(Collections.singletonMap(ConsumerConfig.CLIEN
T_ID_CONFIG, clientId))
        );

        // 创建消费者协调器
        // 自动提交 Offset 时间间隔，默认 5s
       // no coordinator will be constructed for the default (null)
group id
        this.coordinator = !groupId.isPresent() ? null :
            new ConsumerCoordinator(groupRebalanceConfig,
                    logContext,
                    this.client,
                    assignors,
                    this.metadata,
                    this.subscriptions,
                    metrics,
                    metricGrpPrefix,
                    this.time,
                    enableAutoCommit,

    config.getInt(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG),
```

```
                this.interceptors,

   config.getBoolean(ConsumerConfig.THROW_ON_FETCH_STABLE_OFFSET_
UNSUPPORTED));
       // 抓取数据配置
       // 一次抓取最小值，默认 1 个字节
       // 一次抓取最大值，默认 50m
       // 一次抓取最大等待时间，默认 500ms
       // 每个分区抓取的最大字节数，默认 1m
       // 一次 poll 拉取数据返回消息的最大条数，默认是 500 条。
       // key 和 value 的反序列化
       this.fetcher = new Fetcher<>(
             logContext,
             this.client,
            config.getInt(ConsumerConfig.FETCH_MIN_BYTES_CONFIG),
            config.getInt(ConsumerConfig.FETCH_MAX_BYTES_CONFIG),

   config.getInt(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG),

  config.getInt(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG),

   config.getInt(ConsumerConfig.MAX_POLL_RECORDS_CONFIG),
            config.getBoolean(ConsumerConfig.CHECK_CRCS_CONFIG),
            config.getString(ConsumerConfig.CLIENT_RACK_CONFIG),
            this.keyDeserializer,
            this.valueDeserializer,
            this.metadata,
            this.subscriptions,
            metrics,
            metricsRegistry,
            this.time,
            this.retryBackoffMs,
            this.requestTimeoutMs,
            isolationLevel,
            apiVersions);

       this.kafkaConsumerMetrics                =                new
KafkaConsumerMetrics(metrics, metricGrpPrefix);

       config.logUnused();
     AppInfoParser.registerAppInfo(JMX_PREFIX, clientId, metrics,
time.milliseconds());
       log.debug("Kafka consumer initialized");
    } catch (Throwable t) {
       ... ...
    }
}
```

3.2 消费者订阅主题



点击自己编写的 CustomConsumer.java 中的 subscribe ()方法。

CustomConsumer.java

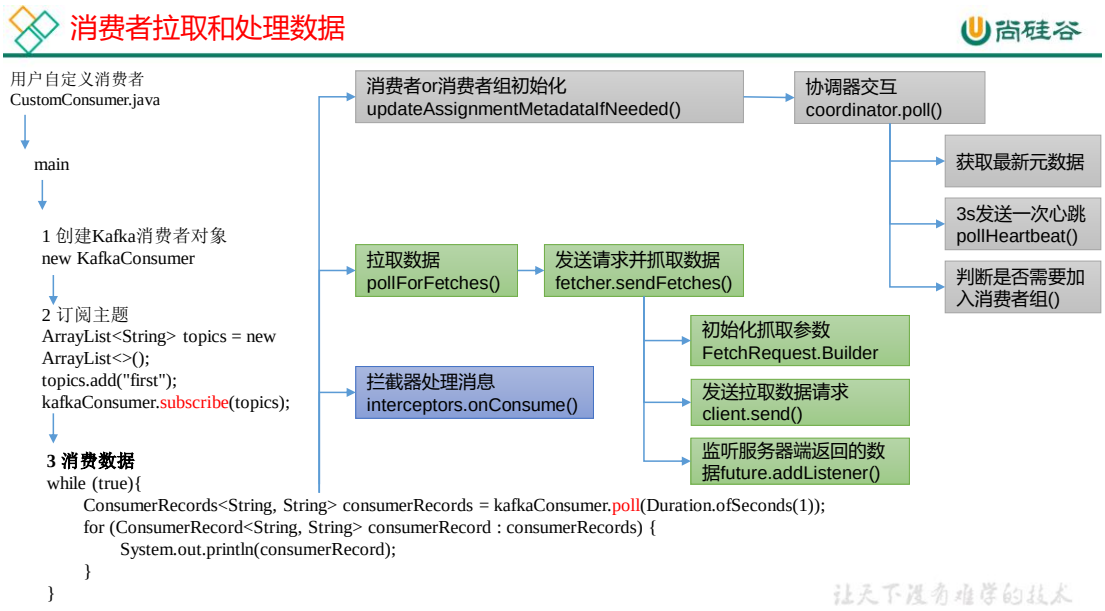
```
// 2 订阅主题 first
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);
```

KafkaConsumer.java

```
@Override
public void subscribe(Collection<String> topics) {
    subscribe(topics, new NoOpConsumerRebalanceListener());
}

@Override
public void subscribe(Collection<String> topics,
ConsumerRebalanceListener listener) {
    acquireAndEnsureOpen();
    try {
        maybeThrowInvalidGroupIdException();
        // 异常情况处理
        if (topics == null)
            throw new IllegalArgumentException("Topic collection to
subscribe to cannot be null");

        if (topics.isEmpty()) {
            // treat subscribing to empty topic list as the same as
unsubscribing
            this.unsubscribe();
        } else {
            for (String topic : topics) {
                if (Utils.isBlank(topic))
                    throw new IllegalArgumentException("Topic
collection to subscribe to cannot contain null or empty topic");
```

```
            }

            throwIfNoAssignorsConfigured();
            // 清空订阅异常主题的缓存数据
            fetcher.clearBufferedDataForUnassignedTopics(topics);
            log.info("Subscribed to topic(s): {}",
Utils.join(topics, ", "));

            // 判断是否需要更改订阅主题，如果需要更改主题，则更新元数据信息
            if (this.subscriptions.subscribe(new HashSet<>(topics),
listener))
                metadata.requestUpdateForNewTopics();
        }
    } finally {
        release();
    }
}

public synchronized boolean subscribe(Set<String> topics,
ConsumerRebalanceListener listener) {
    // 注册负载均衡监听（例如消费者组中，其他消费者退出触发再平衡）
    registerRebalanceListener(listener);
    // 按照设置的主题开始订阅，自动分配分区
    setSubscriptionType(SubscriptionType.AUTO_TOPICS);
    // 修改订阅主题信息
    return changeSubscription(topics);
}

private boolean changeSubscription(Set<String> topicsToSubscribe)
{
    // 如果订阅的主题和以前订阅的一致，就不需要修改订阅信息。如果不一致，就需
要修改。
    if (subscription.equals(topicsToSubscribe))
        return false;

    subscription = topicsToSubscribe;
    return true;
}

    // 如果订阅的和以前不一致，需要更新元数据信息
public synchronized int requestUpdateForNewTopics() {
    // Override the timestamp of last refresh to let immediate
update.
    this.lastRefreshMs = 0;
    this.needPartialUpdate = true;
    this.requestVersion++;
    return this.updateVersion;
}
```

3.3 消费者拉取和处理数据



3.3.1 消费总体流程

点击自己编写的 CustomConsumer.java 中的 poll ()方法。

CustomConsumer.java

```
// 3 消费数据
while (true){

    ConsumerRecords<String, String> consumerRecords = kafkaConsumer.poll(Duration.ofSeconds(1));

    for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
        System.out.println(consumerRecord);
    }
}
```

KafkaConsumer.java

```
@Override
public ConsumerRecords<K, V> poll(final Duration timeout) {
    return poll(time.timer(timeout), true);
}

private ConsumerRecords<K, V> poll(final Timer timer, final boolean includeMetadataInTimeout) {
    acquireAndEnsureOpen();
    try {
        // 记录开始拉取消息时间

    this.kafkaConsumerMetrics.recordPollStart(timer.currentTimeMs());

        if (this.subscriptions.hasNoSubscriptionOrUserAssignment())
```

```
{
          throw   new   IllegalStateException("Consumer   is   not
subscribed to any topics or assigned any partitions");
       }

       do {
          client.maybeTriggerWakeup();

          if (includeMetadataInTimeout) {
             // try to update assignment metadata BUT do not need
to block on the timer for join group
             // 1、消费者 or 消费者组初始化
             updateAssignmentMetadataIfNeeded(timer, false);
          } else {
             while
(!updateAssignmentMetadataIfNeeded(time.timer(Long.MAX_VALUE),
true)) {
                log.warn("Still waiting for metadata");
             }
          }

          // 2、开始拉取数据
          final  Map<TopicPartition,  List<ConsumerRecord<K,  V>>>
records = pollForFetches(timer);
          if (!records.isEmpty()) {
             // before returning the fetched records, we can send
off the next round of fetches
             // and avoid block waiting for their responses to
enable pipelining while the user
             // is handling the fetched records.
             //
             // NOTE: since the consumed position has already
been updated, we must not allow
             // wakeups or any other errors to be triggered prior
to returning the fetched records.
             if      (fetcher.sendFetches()      >      0      ||
client.hasPendingRequests()) {
                client.transmitSends();
             }

             // 3、拦截器处理消息
             return               this.interceptors.onConsume(new
ConsumerRecords<>(records));
          }
       } while (timer.notExpired());

       return ConsumerRecords.empty();
    } finally {
       release();

   this.kafkaConsumerMetrics.recordPollEnd(timer.currentTimeMs());
    }
}
```

3.3.2 消费者/消费者组初始化

```
// 1、消费者 or 消费者组初始化
```

```
boolean updateAssignmentMetadataIfNeeded(final Timer timer, final
boolean waitForJoinGroup) {
    if    (coordinator    !=    null    &&    !coordinator.poll(timer,
waitForJoinGroup)) {
        return false;
    }

    return updateFetchPositions(timer);
}

public boolean poll(Timer timer, boolean waitForJoinGroup) {
    // 获取最新元数据
    maybeUpdateSubscriptionMetadata();

    invokeCompletedOffsetCommitCallbacks();

    if (subscriptions.hasAutoAssignedPartitions()) {
        if (protocol == null) {
            throw new IllegalStateException("User configured " +
ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG +
                " to empty while trying to subscribe for group
protocol to auto assign partitions");
        }
        // Always update the heartbeat last poll time so that the
heartbeat thread does not leave the
        // group proactively due to application inactivity even if
(say) the coordinator cannot be found.
        // 3s 发送一次心跳
        pollHeartbeat(timer.currentTimeMs());
        // 保证和 Coordinator 正常通信（寻找服务器端的 coordinator）
        if (coordinatorUnknown() && !ensureCoordinatorReady(timer))
{
            return false;
        }

        // 判断是否需要加入消费者组
        if (rejoinNeededOrPending()) {
            // due to a race condition between the initial metadata
fetch and the initial rebalance,
            // we need to ensure that the metadata is fresh before
joining initially. This ensures
            // that we have matched the pattern against the
cluster's topics at least once before joining.
            if (subscriptions.hasPatternSubscription()) {
                … …
                if
(this.metadata.timeToAllowUpdate(timer.currentTimeMs()) == 0) {
                    this.metadata.requestUpdate();
                }

                if (!client.ensureFreshMetadata(timer)) {
                    return false;
                }

                maybeUpdateSubscriptionMetadata();
            }
```

```
            // if not wait for join group, we would just use a
timer of 0
            if (!ensureActiveGroup(waitForJoinGroup ? timer :
time.timer(0L))) {
                // since we may use a different timer in the callee,
we'd still need
                // to update the original timer's current time after
the call
                timer.update(time.milliseconds());

                return false;
            }
        }
    } else {
        … …
        if                                  (metadata.updateRequested()
&& !client.hasReadyNodes(timer.currentTimeMs())) {
            client.awaitMetadataUpdate(timer);
        }
    }

    // 是否自动提交 offset
    maybeAutoCommitOffsetsAsync(timer.currentTimeMs());
    return true;
}

protected synchronized boolean ensureCoordinatorReady(final Timer
timer) {
    // 如果找到 coordinator，直接返回
    if (!coordinatorUnknown())
        return true;
    // 如果没有找到，循环给服务器端发送请求，直到找到 coordinator
    do {

        if (fatalFindCoordinatorException != null) {
            final       RuntimeException       fatalException       =
fatalFindCoordinatorException;
            fatalFindCoordinatorException = null;
            throw fatalException;
        }
        // 创建寻找 coordinator 的请求
        final RequestFuture<Void> future = lookupCoordinator();
        // 发送寻找 coordinator 的请求给服务器端
        client.poll(future, timer);

        if (!future.isDone()) {
            // ran out of time
            break;
        }

        RuntimeException fatalException = null;

        if (future.failed()) {
            if (future.isRetriable()) {
                log.debug("Coordinator discovery failed, refreshing
metadata", future.exception());
                client.awaitMetadataUpdate(timer);
```

```
            } else {
                fatalException = future.exception();
                log.info("FindCoordinator request hit fatal
exception", fatalException);
            }
        } else if (coordinator != null &&
client.isUnavailable(coordinator)) {
            // we found the coordinator, but the connection has
failed, so mark
            // it dead and backoff before retrying discovery
            markCoordinatorUnknown("coordinator unavailable");
            timer.sleep(rebalanceConfig.retryBackoffMs);
        }

        clearFindCoordinatorFutureFuture();
        if (fatalException != null)
            throw fatalException;
    } while (coordinatorUnknown() && timer.notExpired());

    return !coordinatorUnknown();
}

protected synchronized RequestFuture<Void> lookupCoordinator() {
    if (findCoordinatorFuture == null) {
        // find a node to ask about the coordinator
        Node node = this.client.leastLoadedNode();
        if (node == null) {
            log.debug("No broker available to send FindCoordinator
request");
            return RequestFuture.noBrokersAvailable();
        } else {
            // 向服务器端发送，查找 Coordinator 请求
            findCoordinatorFuture =
sendFindCoordinatorRequest(node);
        }
    }
    return findCoordinatorFuture;
}

private RequestFuture<Void> sendFindCoordinatorRequest(Node node)
{
    // initiate the group metadata request
    log.debug("Sending FindCoordinator request to broker {}",
node);
    // 封装发送请求
    FindCoordinatorRequestData data = new
FindCoordinatorRequestData()
            .setKeyType(CoordinatorType.GROUP.id())
            .setKey(this.rebalanceConfig.groupId);
    FindCoordinatorRequest.Builder requestBuilder = new
FindCoordinatorRequest.Builder(data);
    // 消费者向服务器端发送请求
    return client.send(node, requestBuilder)
            .compose(new FindCoordinatorResponseHandler());
}
```

3.3.3 拉取数据

```
// 2、开始拉取数据
private Map<TopicPartition, List<ConsumerRecord<K, V>>>
pollForFetches(Timer timer) {
    long pollTimeout = coordinator == null ? timer.remainingMs() :

    Math.min(coordinator.timeToNextPoll(timer.currentTimeMs()),
timer.remainingMs());

    // if data is available already, return it immediately
    final Map<TopicPartition, List<ConsumerRecord<K, V>>> records
= fetcher.fetchedRecords();
    if (!records.isEmpty()) {
        return records;
    }

    // send any new fetches (won't resend pending fetches)
    // 2.1 发送请求并抓取数据
    fetcher.sendFetches();

    // We do not want to be stuck blocking in poll if we are
missing some positions
    // since the offset lookup may be backing off after a failure

    // NOTE: the use of cachedSubscriptionHashAllFetchPositions
means we MUST call
    // updateAssignmentMetadataIfNeeded before this method.
    if (!cachedSubscriptionHashAllFetchPositions && pollTimeout >
retryBackoffMs) {
        pollTimeout = retryBackoffMs;
    }

    log.trace("Polling for fetches with timeout {}", pollTimeout);

    Timer pollTimer = time.timer(pollTimeout);
    client.poll(pollTimer, () -> {
        // since a fetch might be completed by the background
thread, we need this poll condition
        // to ensure that we do not block unnecessarily in poll()
        return !fetcher.hasAvailableFetches();
    });
    timer.update(pollTimer.currentTimeMs());

    // 2.2 把数据按照分区封装好后，一次处理默认 500 条数据
    return fetcher.fetchedRecords();
}
```

2.1 发送请求并抓取数据

Fetcher.java

```
public synchronized int sendFetches() {
    // Update metrics in case there was an assignment change
    sensors.maybeUpdateAssignment(subscriptions);
```

```
    Map<Node,                  FetchSessionHandler.FetchRequestData>
fetchRequestMap = prepareFetchRequests();
    for    (Map.Entry<Node,    FetchSessionHandler.FetchRequestData>
entry : fetchRequestMap.entrySet()) {
        final Node fetchTarget = entry.getKey();
        final     FetchSessionHandler.FetchRequestData     data     =
entry.getValue();
        // 初始化抓取数据的参数：
        // 最大等待时间默认 500ms
        // 最小抓取一个字节
        // 最大抓取 50m 数据，
        final FetchRequest.Builder request = FetchRequest.Builder
                .forConsumer(this.maxWaitMs,           this.minBytes,
data.toSend())
                .isolationLevel(isolationLevel)
                .setMaxBytes(this.maxBytes)
                .metadata(data.metadata())
                .toForget(data.toForget())
                .rackId(clientRackId);

        if (log.isDebugEnabled()) {
            log.debug("Sending {} {} to broker {}", isolationLevel,
data.toString(), fetchTarget);
        }
        // 发送拉取数据请求
        RequestFuture<ClientResponse>            future            =
client.send(fetchTarget, request);
        // We add the node to the set of nodes with pending fetch
requests before adding the
        // listener because the future may have been fulfilled on
another thread (e.g. during a
        // disconnection being handled by the heartbeat thread)
which will mean the listener
        // will be invoked synchronously.
       this.nodesWithPendingFetchRequests.add(entry.getKey().id());
        // 监听服务器端返回的数据
        future.addListener(new
RequestFutureListener<ClientResponse>() {
            @Override
            public void onSuccess(ClientResponse resp) {
                // 成功接收服务器端数据
                synchronized (Fetcher.this) {
                    try {
                        // 获取服务器端响应数据
                        FetchResponse   response   =   (FetchResponse)
resp.responseBody();
                        FetchSessionHandler          handler          =
sessionHandler(fetchTarget.id());
                        if (handler == null) {
                            log.error("Unable          to          find
FetchSessionHandler for node {}. Ignoring fetch response.",
                                    fetchTarget.id());
                            return;
                        }
                        if (!handler.handleResponse(response)) {
                            return;
```

```
                    }

                    Set<TopicPartition>    partitions    =    new
HashSet<>(response.responseData().keySet());
                    FetchResponseMetricAggregator
metricAggregator   =   new   FetchResponseMetricAggregator(sensors,
partitions);

                    for                  (Map.Entry<TopicPartition,
FetchResponseData.PartitionData>             entry             :
response.responseData().entrySet()) {
                        TopicPartition partition = entry.getKey();
                        FetchRequest.PartitionData  requestData  =
data.sessionPartitions().get(partition);
                        if (requestData == null) {
                            String message;
                            if (data.metadata().isFull()) {
                                message                            =
MessageFormatter.arrayFormat(
                                        "Response  for  missing  full
request partition: partition={}; metadata={}",
                                        new       Object[]{partition,
data.metadata()}).getMessage();
                            } else {
                                message                            =
MessageFormatter.arrayFormat(
                                        "Response for missing session
request   partition:   partition={};   metadata={};   toSend={};
toForget={}",
                                        new       Object[]{partition,
data.metadata(), data.toSend(), data.toForget()}).getMessage();
                            }

                            // Received fetch response for missing
session partition
                            throw                                  new
IllegalStateException(message);
                        } else {
                            long           fetchOffset           =
requestData.fetchOffset;
                            FetchResponseData.PartitionData
partitionData = entry.getValue();

                            log.debug("Fetch  {}  at  offset  {}  for
partition {} returned fetch data {}",
                                    isolationLevel,     fetchOffset,
partition, partitionData);

                            Iterator<? extends RecordBatch> batches
=  FetchResponse.recordsOrFail(partitionData).batches().iterator();
                            short         responseVersion         =
resp.requestHeader().apiVersion();
                            // 把数据按照分区，添加到消息队列里面
                            //              private              final
ConcurrentLinkedQueue<CompletedFetch> completedFetches;
                            completedFetches.add(new
CompletedFetch(partition, partitionData,
```

```
                                    metricAggregator,        batches,
fetchOffset, responseVersion));
                        }
                    }


    sensors.fetchLatency.record(resp.requestLatencyMs());
                } finally {

    nodesWithPendingFetchRequests.remove(fetchTarget.id());
                }
            }
        }

        @Override
        public void onFailure(RuntimeException e) {
            synchronized (Fetcher.this) {
                try {
                    FetchSessionHandler          handler          =
sessionHandler(fetchTarget.id());
                    if (handler != null) {
                        handler.handleError(e);
                    }
                } finally {

    nodesWithPendingFetchRequests.remove(fetchTarget.id());
                }
            }
        }
    });

    }
    return fetchRequestMap.size();
}
```

2.2 把数据按照分区封装好后，一次处理最大条数默认 500 条数据

```
public     Map<TopicPartition,     List<ConsumerRecord<K,     V>>>
fetchedRecords() {
    Map<TopicPartition, List<ConsumerRecord<K, V>>> fetched = new
HashMap<>();
    Queue<CompletedFetch>     pausedCompletedFetches     =     new
ArrayDeque<>();
    // 一次处理的最大条数，默认 500 条
    int recordsRemaining = maxPollRecords;

    try {
        // 循环处理
        while (recordsRemaining > 0) {
            if        (nextInLineFetch        ==        null        ||
nextInLineFetch.isConsumed) {
                // 从缓存中获取数据
                CompletedFetch records = completedFetches.peek();
                // 缓存中数据为 null,直接跳出循环
                if (records == null) break;

                if (records.notInitialized()) {
                    try {
```

```
                    nextInLineFetch                                =
initializeCompletedFetch(records);
                } catch (Exception e) {
                    // Remove a completedFetch upon a parse with
exception if (1) it contains no records, and
                    //  (2)  there  are  no  fetched  records  with
actual content preceding this exception.
                    //  The  first  condition  ensures  that  the
completedFetches is not stuck with the same completedFetch
                    //      in      cases      such      as      the
TopicAuthorizationException,  and  the  second  condition  ensures
that no
                    // potential data loss due to an exception in
a following record.
                    FetchResponseData.PartitionData  partition  =
records.partitionData;
                    if            (fetched.isEmpty()            &&
FetchResponse.recordsOrFail(partition).sizeInBytes() == 0) {
                        completedFetches.poll();
                    }
                    throw e;
                }
            } else {
                nextInLineFetch = records;
            }
            // 从缓存中拉取数据
            completedFetches.poll();
        }                          else                          if
(subscriptions.isPaused(nextInLineFetch.partition)) {
            // when the partition is paused we add the records
back to the completedFetches queue instead of draining
            // them so that they can be returned on a subsequent
poll if the partition is resumed at that time
            log.debug("Skipping  fetching  records  for  assigned
partition {} because it is paused", nextInLineFetch.partition);
            pausedCompletedFetches.add(nextInLineFetch);
            nextInLineFetch = null;
        } else {
            List<ConsumerRecord<K,      V>>      records      =
fetchRecords(nextInLineFetch, recordsRemaining);

            if (!records.isEmpty()) {
                TopicPartition            partition            =
nextInLineFetch.partition;
                List<ConsumerRecord<K,   V>>   currentRecords   =
fetched.get(partition);
                if (currentRecords == null) {
                    fetched.put(partition, records);
                } else {
                    // this case shouldn't usually happen because
we only send one fetch at a time per partition,
                    // but it might conceivably happen in some
rare cases (such as partition leader changes).
                    // we have to copy to a new list because the
old one may be immutable
                    List<ConsumerRecord<K, V>> newRecords = new
ArrayList<>(records.size() + currentRecords.size());
```

```
                    newRecords.addAll(currentRecords);
                    newRecords.addAll(records);
                    fetched.put(partition, newRecords);
                }
                recordsRemaining -= records.size();
            }
        }
    }
} catch (KafkaException e) {
    if (fetched.isEmpty())
        throw e;
} finally {
    // add any polled completed fetches for paused partitions
back to the completed fetches queue to be
    // re-evaluated in the next poll
    completedFetches.addAll(pausedCompletedFetches);
}

return fetched;
}
```

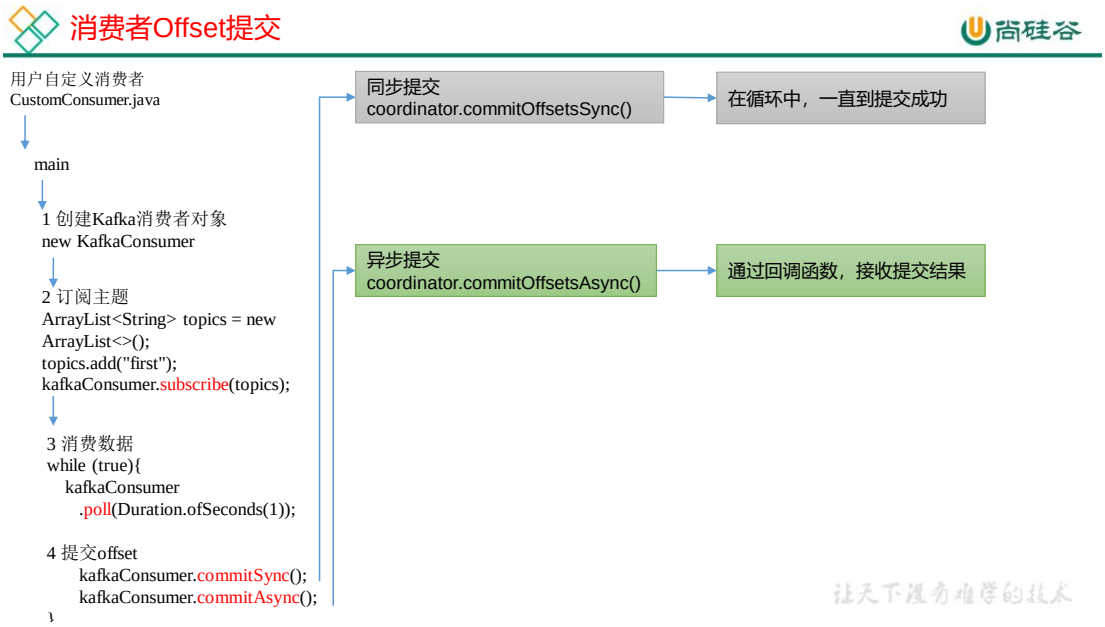
3.3.4 拦截器处理数据

在 poll()方法中点击 onConsume()方法。

```
// 3、拦截器处理消息
// 数据从服务器端，返回后，放入集合中缓存
final Map<TopicPartition, List<ConsumerRecord<K, V>>> records =
pollForFetches(timer);
… …
// 从集合中拉取数据处理，首先经过的是拦截器
return this.interceptors.onConsume(new
ConsumerRecords<>(records));
```

```
public ConsumerRecords<K, V> onConsume(ConsumerRecords<K, V>
records) {
    ConsumerRecords<K, V> interceptRecords = records;
   for (ConsumerInterceptor<K, V> interceptor : this.interceptors)
{
        try {
            interceptRecords =
interceptor.onConsume(interceptRecords);
        } catch (Exception e) {
            // do not propagate interceptor exception, log and
continue calling other interceptors
            log.warn("Error executing interceptor onConsume
callback", e);
        }
    }
    return interceptRecords;
}
```

3.4 消费者 Offset 提交



3.4.1 手动同步提交 Offset

手动同步提交 Offset

CustomConsumer.java

```
// 手动提交
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,false);

// 1 创建一个消费者  "", "hello"
KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<>(properties);

// 2 订阅主题 first
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 3 消费数据
while (true){

    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord);
    }

    // 手动提交 offset
    kafkaConsumer.commitSync();
}
```

KafkaConsumer.java

```
@Override
public void commitSync() {
```

```
    commitSync(Duration.ofMillis(defaultApiTimeoutMs));
}

@Override
public void commitSync(Duration timeout) {
    commitSync(subscriptions.allConsumed(), timeout);
}

@Override
public void commitSync(final Map<TopicPartition,
OffsetAndMetadata> offsets, final Duration timeout) {
    acquireAndEnsureOpen();
    try {
        maybeThrowInvalidGroupIdException();
        offsets.forEach(this::updateLastSeenEpochIfNewer);
        // 同步提交
        if (!coordinator.commitOffsetsSync(new HashMap<>(offsets),
time.timer(timeout))) {
            throw new TimeoutException("Timeout of " +
timeout.toMillis() + "ms expired before successfully " +
                "committing offsets " + offsets);
        }
    } finally {
        release();
    }
}


public boolean commitOffsetsSync(Map<TopicPartition,
OffsetAndMetadata> offsets, Timer timer) {
    invokeCompletedOffsetCommitCallbacks();

    if (offsets.isEmpty())
        return true;

    do {
        if (coordinatorUnknown() && !ensureCoordinatorReady(timer))
{
            return false;
        }

        // 发送提交请求
        RequestFuture<Void> future =
sendOffsetCommitRequest(offsets);
        client.poll(future, timer);

        // We may have had in-flight offset commits when the
synchronous commit began. If so, ensure that
        // the corresponding callbacks are invoked prior to
returning in order to preserve the order that
        // the offset commits were applied.
        invokeCompletedOffsetCommitCallbacks();

        // 提交成功
        if (future.succeeded()) {
            if (interceptors != null)
                interceptors.onCommit(offsets);
```

```
            return true;
        }

        if (future.failed() && !future.isRetriable())
            throw future.exception();

        timer.sleep(rebalanceConfig.retryBackoffMs);
    } while (timer.notExpired());

    return false;
}
```

3.4.2 手动异步提交 Offset

手动异步提交 Offset

CustomConsumer.java

```
// 手动提交
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,false);

// 1 创建一个消费者  "", "hello"
KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<>(properties);

// 2 订阅主题 first
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 3 消费数据
while (true){

    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord);
    }

    // 手动提交 offset
    // kafkaConsumer.commitSync();
    kafkaConsumer.commitAsync();
}
```

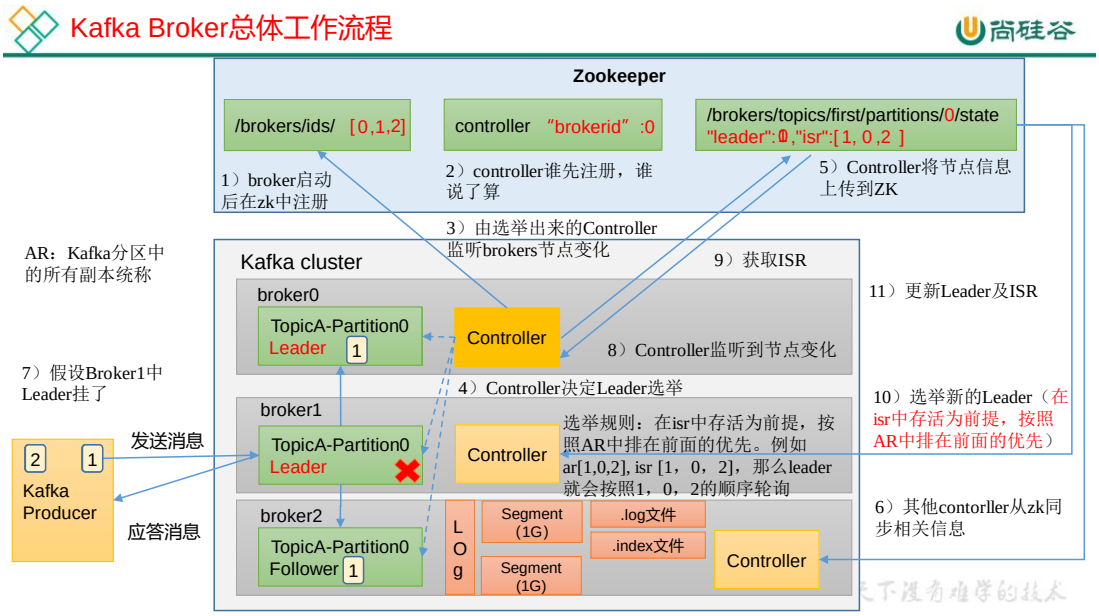
KafkaConsumer.java

```
@Override
public void commitAsync() {
    commitAsync(null);
}

@Override
public void commitAsync(OffsetCommitCallback callback) {
    commitAsync(subscriptions.allConsumed(), callback);
}

@Override
```

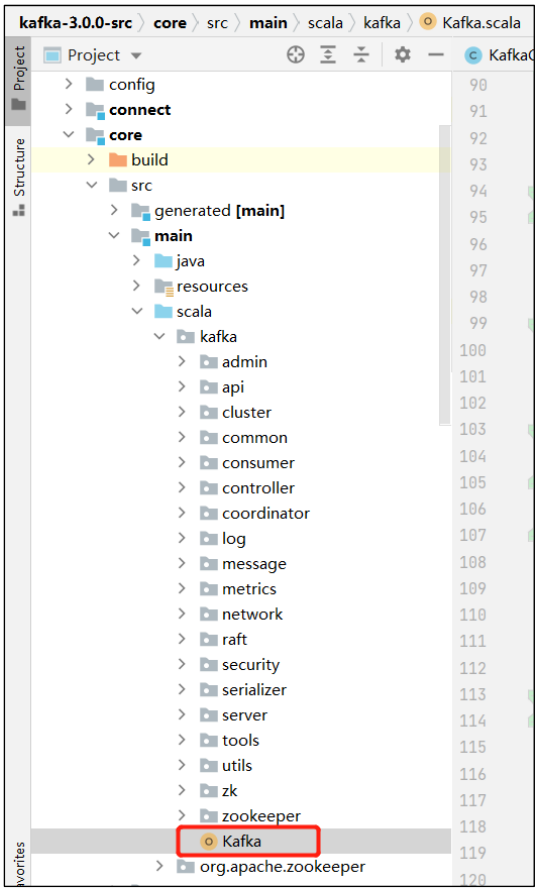
```
public void commitAsync(final Map<TopicPartition,
OffsetAndMetadata> offsets, OffsetCommitCallback callback) {
    acquireAndEnsureOpen();
    try {
        maybeThrowInvalidGroupIdException();
        log.debug("Committing offsets: {}", offsets);
        offsets.forEach(this::updateLastSeenEpochIfNewer);
        // 提交 offset
        coordinator.commitOffsetsAsync(new HashMap<>(offsets),
callback);
    } finally {
        release();
    }
}

public void commitOffsetsAsync(final Map<TopicPartition,
OffsetAndMetadata> offsets, final OffsetCommitCallback callback)
{
    invokeCompletedOffsetCommitCallbacks();

    if (!coordinatorUnknown()) {
        doCommitOffsetsAsync(offsets, callback);
    } else {
        // we don't know the current coordinator, so try to find it
and then send the commit
        // or fail (we don't want recursive retries which can cause
offset commits to arrive
        // out of order). Note that there may be multiple offset
commits chained to the same
        // coordinator lookup request. This is fine because the
listeners will be invoked in
        // the same order that they were added. Note also that
AbstractCoordinator prevents
        // multiple concurrent coordinator lookup requests.
        pendingAsyncCommits.incrementAndGet();
        // 监听提交 offset 的结果
        lookupCoordinator().addListener(new
RequestFutureListener<Void>() {
            @Override
            public void onSuccess(Void value) {
                pendingAsyncCommits.decrementAndGet();
                doCommitOffsetsAsync(offsets, callback);
                client.pollNoWakeup();
            }

            @Override
            public void onFailure(RuntimeException e) {
                pendingAsyncCommits.decrementAndGet();
                completedOffsetCommits.add(new
OffsetCommitCompletion(callback, offsets,
                        new RetriableCommitFailedException(e)));
            }
        });
    }

    // ensure the commit has a chance to be transmitted (without
blocking on its completion).
```

```
    // Note that commits are treated as heartbeats by the
coordinator, so there is no need to
    // explicitly allow heartbeats through delayed task execution.
    client.pollNoWakeup();
}
```

第 4 章 服务器源码



4.1 程序入口



Kafka.scala

程序的入口

```
def main(args: Array[String]): Unit = {
try {
  // 获取参数相关信息
  val serverProps = getPropsFromArgs(args)
  // 配置服务
  val server = buildServer(serverProps)

  try {
    if (!OperatingSystem.IS_WINDOWS && !Java.isIbmJdk)
      new LoggingSignalHandler().register()
  } catch {
    case e: ReflectiveOperationException =>
      warn("Failed to register optional signal handler that logs a
message when the process is terminated " +
        s"by a signal. Reason for registration failure is: $e", e)
  }

  // attach shutdown handler to catch terminating signals as well
as normal termination
  Exit.addShutdownHook("kafka-shutdown-hook", {
    try server.shutdown()
    catch {
      case _: Throwable =>
```

```
        fatal("Halting Kafka.")
        // Calling exit() can lead to deadlock as exit() can be
called multiple times. Force exit.
        Exit.halt(1)
    }
  })

  // 启动服务
  try server.startup()
  catch {
    case _: Throwable =>
      //  KafkaServer.startup()  calls  shutdown()  in  case  of
exceptions, so we invoke `exit` to set the status code
      fatal("Exiting Kafka.")
      Exit.exit(1)
  }

  server.awaitShutdown()
}
catch {
  case e: Throwable =>
    fatal("Exiting Kafka due to fatal exception", e)
    Exit.exit(1)
}
Exit.exit(0)
}
```