腾讯课堂零声教育：https://ke.qq.com/course/420945?tuin=137bb271

基于原文链接：https://blog.51cto.com/u_9291927/2502063修改。
网页版本：https://www.yuque.com/docs/share/c7e58ccb-62b2-4e14-a0d9-dfe08cd4793e?# 《1-4 C++操作kafka》

官方C/C++ API：https://docs.confluent.io/2.0.0/clients/librdkafka/index.html

源码：cplus_kafka压缩包。

0 安装librdkafka

https://github.com/edenhill/librdkafka

```
git clone https://github.com/edenhill/librdkafka.git
cd librdkafka
git checkout v1.7.0
./configure
make
sudo make install
sudo ldconfig
```

缩略语

缩略语	缩略语全称	示例或说明
rd	Rapid Development	rd.h
rk	RdKafka	
toppar	Topic Partition	struct rd_kafka_toppar_t { };
rep	Reply,	struct rd_kafka_t { rd_kafka_q_t *rk_rep };
msgq	Message Queue	struct rd_kafka_msgq_t { };
rkb	RdKafka Broker	Kafka代理
rko	RdKafka Operation	Kafka操作
rkm	RdKafka Message	Kafka消息
payload		存在Kafka上的消息（或叫Log)

一、C++ API

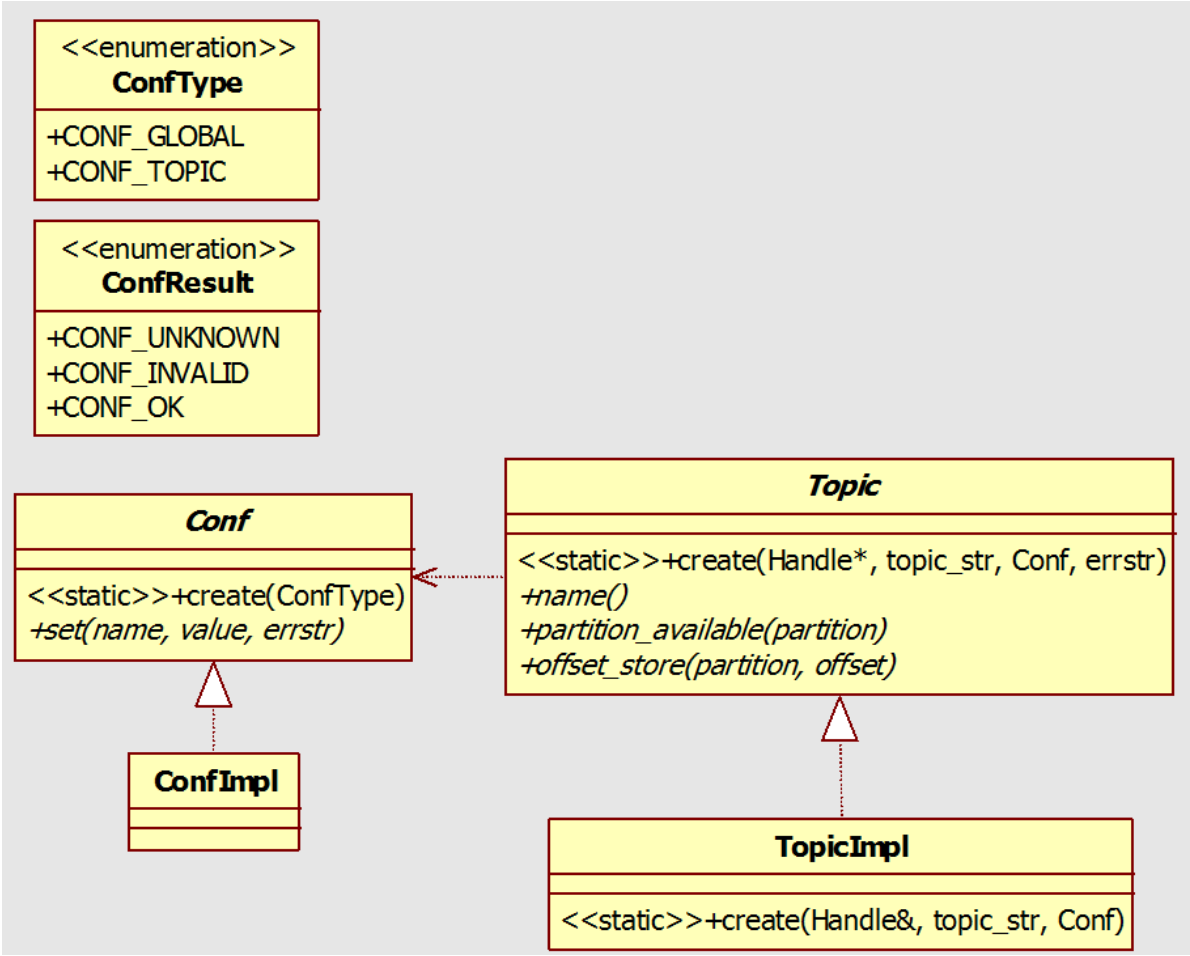
1、数据结构

RdKafka::DeliveryReportCb：Delivery Report回调类
RdKafka::PartitionerCb：Partitioner回调类
RdKafka::PartitionerKeyPointerCb：带key指针的Partitioner回调类
RdKafka::EventCb：Event回调类
RdKafka::Event：Event类
RdKafka::ConsumeCb：Consume回调类
RdKafka::RebalanceCb：KafkaConsunmer: Rebalance回调类
RdKafka::OffsetCommitCb：Offset Commit回调类
RdKafka::SocketCb：Socket回调类
RdKafka::OpenCb：Open回调类
RdKafka::Conf：配置接口类
RdKafka::Handle：客户端基类
RdKafka::TopicPartition：Topic+Partion类
RdKafka::Topic：Topic Handle
RdKafka::Message：消息对象类
RdKafka::Queue：队列接口
RdKafka::KafkaConsumer：KafkaConsumer高级接口
RdKafka::Consumer：简单Consumer类
RdKafka::Producer：Producer类
RdKafka::BrokerMetadata：Broker元数据信息类
RdKafka::PartitionMetadata：Partition元数据信息类
RdKafka::TopicMetadata：Topic元数据信息类
RdKafka::Metadata：元数据容器
librdkafka C++ API定义在rdkafkacpp.h文件中，兼容STD C++ 03标准，遵循Google编码规范。

2、通用API

通用API	说明
int RdKafka::version ();	获取librdkafka版本
std::string RdKafka::version_str();	获取librdkafka版本
std::string RdKafka::get_debug_contexts ();	获取librdkafka调试环境
int RdKafka::wait_destroyed(int timeout_ms);	等待所有的 rd_kafka_t对象销毁
std::string RdKafka::err2str(RdKafka::ErrorCode err);	将Kafka错误代码转换成可读字符串

3、RdKafka::Conf




```
enum  ConfType{
  CONF_GLOBAL, // 全局配置
  CONF_TOPIC // Topic配置
};
enum  ConfResult{
  CONF_UNKNOWN = -2,
  CONF_INVALID = -1,
  CONF_OK = 0
};
```

static Conf * create(ConfType type);
创建配置对象

Conf::ConfResult set(const std::string &name, const std::string &value, std::string &errstr);
设置配置对象的属性值，成功返回CONF_OK，错误时错误信息输出到errstr。

Conf::ConfResult set(const std::string &name, DeliveryReportCb *dr_cb, std::string &errstr);
设置dr_cb属性值

Conf::ConfResult set(const std::string &name, EventCb *event_cb, std::string &errstr);
设置event_cb属性值

Conf::ConfResult set(const std::string &name, const Conf *topic_conf, std::string &errstr);
设置用于自动订阅Topic的默认Topic配置

Conf::ConfResult set(const std::string &name, PartitionerCb *partitioner_cb, std::string &errstr);
设置partitioner_cb属性值，配置对象必须是CONF_TOPIC类型。

Conf::ConfResult set(const std::string &name, PartitionerKeyPointerCb *partitioner_kp_cb, std::string &errstr);
设置partitioner_key_pointer_cb属性值

Conf::ConfResult set(const std::string &name, SocketCb *socket_cb, std::string &errstr);
设置socket_cb属性值

Conf::ConfResult set(const std::string &name, OpenCb *open_cb, std::string &errstr);
设置open_cb属性值

Conf::ConfResult set(const std::string &name, RebalanceCb *rebalance_cb, std::string &errstr);
设置rebalance_cb属性值

Conf::ConfResult set(const std::string &name, OffsetCommitCb *offset_commit_cb, std::string &errstr);
设置offset_commit_cb属性值

Conf::ConfResult get(const std::string &name, std::string &value) const;
查询单条属性配置值

std::liststd::string * dump ();
按name,value元组序列化配置对象的属性名称和属性值到链表

virtual struct rd_kafka_conf_s *c_ptr_global () = 0;
如果是CONF_GLOBAL类型配置对象，返回底层数据结构rd_kafka_conf_t句柄，否则返回NULL。

virtual struct rd_kafka_topic_conf_s *c_ptr_topic () = 0;
如果是CONF_TOPIC类型配置对象，返回底层数据结构的rd_kafka_topic_conf_t句柄，否则返回0。

4、RdKafka::Topic

- static Topic * create(Handle *base, const std::string &topic_str, Conf *conf, std::string &errstr);
  使用conf配置创建名为topic_str的Topic句柄
- const std::string name ();
  获取Topic名称
- bool partition_available(int32_t partition) const;
  获取parition分区是否可用，只能在 RdKafka::PartitionerCb回调函数内被调用。
- ErrorCode offset_store(int32_t partition, int64_t offset);
  存储Topic的partition分区的offset位移，只能用于RdKafka::Consumer，不能用于RdKafka::KafkaConsumer高级接口类。使用本接口时，auto.commit.enable参数必须设置为false。
- virtual struct rd_kafka_topic_s *c_ptr () = 0;
  返回底层数据结构的rd_kafka_topic_t句柄，不推荐利用rd_kafka_topic_t句柄调用C API，但如果C++ API没有提供相应功能，可以直接使用C API和librdkafka核心交互。
- static const int32_t PARTITION_UA = -1;
  未赋值分区
- static const int64_t OFFSET_BEGINNING = -2;
  特殊位移，从开始消费
- static const int64_t OFFSET_END = -1;
  特殊位移，从末尾消费
- static const int64_t OFFSET_STORED = -1000;
  使用offset存储

5、RdKafka::Message

Message表示一条消费或生产的消息，或是事件。

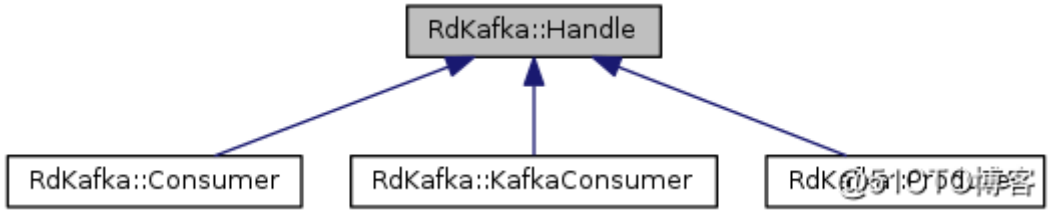
- std::string errstr() const;
  如果消息是一条错误事件，返回错误字符串，否则返回控字符串。
- ErrorCode err() const;
  如果消息是一条错误事件，返回错误代码，否则返回0
- Topic * topic() const;
  返回消息的Topic对象。如果消息的Topic对象没有显示使用RdKafka::Topic::create()创建，需要使用topic_name函数。
- std::string topic_name() const;
  返回消息的Topic名称
- int32_t partition() const;
  如果分区可用，返回分区号
- void * payload() const;
  返回消息数据
- size_t len() const;
  返回消息数据的长度
- const std::string * key() const;
  返回字符串类型的消息key
- const void * key_pointer() const;
  返回void类型的消息key
- size_t key_len() const;
  返回消息key的二进制长度
- int64_t offset () const;
  返回消息或错误的位移
- void * msg_opaque() const;
  返回RdKafka::Producer::produce()提供的msg_opaque
- virtual MessageTimestamp timestamp() const = 0;
  返回消息时间戳
- virtual int64_t latency() const = 0;
  返回produce函数内生产消息的微秒级时间延迟，如果延迟不可用，返回-1。
- virtual struct rd_kafka_message_s *c_ptr () = 0;
  返回底层数据结构的C rd_kafka_message_t句柄
- virtual Status status () const = 0;
  返回消息在Topic Log的持久化状态
- virtual RdKafka::Headers *headers () = 0;
  返回消息头
- virtual RdKafka::Headers *headers (RdKafka::ErrorCode *err) = 0;
  返回消息头，错误信息会输出到err

6、RdKafka::TopicPartition

- static TopicPartition * create(const std::string &topic, int partition);
  创建一个TopicPartition对象
- static TopicPartition *create (const std::string &topic, int partition,int64_t offset);
  创建TopicPartition对象
- static void destroy (std::vector<TopicPartition*> &partitions);
  销毁所有TopicPartition对象
- const std::string & topic () const;
  返回Topic名称

- int partition ();
  返回分区号
- int64_t offset();
  返回位移
- void set_offset(int64_t offset);
  设置位移
- ErrorCode err();
  返回错误码

7、RdKafka::Handle



客户端handle基类

- const std::string name();
  返回Handle的名称
- const std::string memberid() const;
  返回客户端组成员ID
- int poll (int timeout_ms);
  轮询处理指定的Kafka句柄的Event，返回事件数量。事件会触发应用程序提供的回调函数调用。timeout_ms参数指定回调函数指定阻塞等待的最大时间间隔；对于非阻塞调用，指定timeout_ms参数为0；永远等待事件，设置timeout_ms参数为-1。RdKafka::KafkaConsumer实例禁止使用poll方法，使用RdKafka::KafkaConsumer::consume()方法代替。
- int outq_len();
  返回当前出队列的长度，出队列包含等待发送到Broker的消息、请求和Broker要确认的消息、请求。
- ErrorCode metadata(bool all_topics, const Topic *only_rkt, Metadata **metadatap, int timeout_ms);
  从Broker请求元数据，成功返回RdKafka::ERR_NO_ERROR，超时返回RdKafka::ERR_TIMED_OUT，错误返回其它错误码。
- virtual ErrorCode pause (std::vector<TopicPartition*> &partitions) = 0;
  暂停分区链表中分区的消费和生产，返回ErrorCode::NO_ERROR。partitions中分区会返回成功或错误信息。
- virtual ErrorCode resume (std::vector<TopicPartition*> &partitions) = 0;
  恢复分区链表中分区的生产和消费，返回ErrorCode::NO_ERROR。
  partitions中分区会返回成功或错误信息。
  virtual ErrorCode query_watermark_offsets (const std::string &topic,int32_t partition,int64_t *low, int64_t *high,int timeout_ms) = 0;
  查询topic主题partition分区的高水位和低水位，高水位输出到high，低水位输出到low，成功返回RdKafka::ERR_NO_ERROR，失败返回错误码。
- virtual ErrorCode get_watermark_offsets (const std::string &topic,int32_t partition,int64_t *low, int64_t *high) = 0;
  获取topic主题partition分区的高水位和低水位，高水位输出到high，低水位输出到low，成功返回RdKafka::ERR_NO_ERROR，失败返回错误码。
- virtual ErrorCode offsetsForTimes (std::vector<TopicPartition*> &offsets,int timeout_ms) = 0;
  通过时间戳查询给定分区的位移，每个分区返回的位移是最新的位移，阻塞timeout_ms。
- virtual Queue *get_partition_queue (const TopicPartition *partition) = 0;
  获取指定TopicPartition的消息队列，成功返回从指定分区获取的队列，否则返回NULL。

- virtual ErrorCode set_log_queue (Queue *queue) = 0;
  将rdkafka logs转移到指定消息队列。queue是要转移rdkafka logs到的消息队列，如果为NULL，则转移到主消息队列。Log.queue属性必须设置为true。
- virtual void yield () = 0;
  取消当前回调函数调度器，如Handle::poll()，KafkaConsumer::consume()。只能再RdKafka回调函数内调用。
- virtual const std::string clusterid (int timeout_ms) = 0;
  返回Broker元数据报告的集群ID，要求Kafka 0.10.0以上版本。
- virtual struct rd_kafka_s *c_ptr () = 0;
  返回底层数据的rd_kafka_t句柄
- virtual int32_t controllerid (int timeout_ms) = 0;
  返回Broker元数据报告的当前控制器ID，要求Kafka 0.10.0以上版本，并且api.version.request=true
- virtual ErrorCode fatal_error (std::string &errstr) = 0;
  返回客户端实例的第一个fatal错误的错误代码
- virtual ErrorCode oauthbearer_set_token (const std::string &token_value,

int64_t md_lifetime_ms,
const std::string &md_principal_name,
const std::liststd::string &extensions,
std::string &errstr) = 0;

设置SASL/OAUTHBEARER令牌和元数据

- virtual ErrorCode oauthbearer_set_token_failure (const std::string &errstr) = 0;
  设置SASL/OAUTHBEARER刷新失败指示器

8、RdKafka::Producer（重点）

- static Producer * create(Conf *conf, std::string &errstr);
  创建一个新的Producer客户端对象，conf用于替换默认配置对象，本函数调用后conf可以重用。成功返回新的Producer客户端对象，失败返回NULL，errstr可读错误信息。
- ErrorCode produce(Topic *topic, int32_t partition, int msgflags, void *payload, size_t len, const std::string *key, void *msg_opaque);
  生产和发送单条消息到Broker。
  - topic：主题
  - partition：分区
  - msgflags：可选项为RK_MSG_BLOCK、RK_MSG_FREE、RK_MSG_COPY。RK_MSG_FREE表示RdKafka调用produce完成后会释放payload数据；RK_MSG_COPY表示payload数据会被拷贝，在produce调用完成后RdKafka不会使用payload指针；RK_MSG_BLOCK表示在消息队列满时阻塞produce函数，如果dr_cb回调函数被使用，应用程序必须调用rd_kafka_poll函数确保投递消息队列的投递消息投递完。当消息队列满时，失败会导致produce函数的永久阻塞。RK_MSG_FREE和RK_MSG_COPY是互斥操作。
    如果produce函数调用时指定了RK_MSG_FREE，并返回了错误码，
    与payload指针相关的内存数据必须由使用者负责释放。
  - payload：长度为len的消息负载数据
  - len：payload消息数据的长度。
  - key：key是可选的消息key，如果非NULL，会被传递给主题partitioner，并被随消息发送到Broker和传递给Consumer。
  - msg_opaque：msg_opaque是可选的应用程序提供给每条消息的opaque指针，opaque指针会在dr_cb回调函数内提供。
  - 返回错误码：
    ERR_NO_ERROR：消息成功发送并入对列。

ERR_QUEUE_FULL：最大消息数量达到queue.buffering.max.message。
ERR_MSG_SIZE_TOO_LARGE：消息数据大小太大，超过messages.max.bytes配置的值。
ERR_UNKNOWN_PARTITION：请求一个Kafka集群内的未知分区。
ERR_UNKNOWN_TOPIC：topic是Kafka集群的未知主题。

- ErrorCode produce(Topic *topic, int32_t partition, int msgflags, void *payload, size_t len, const void *key, size_t key_len, void *msg_opaque);
  生产和发送单条消息到Broker，传递key数据指针和key长度。
- ErrorCode produce(Topic *topic, int32_t partition, const std::vector< char > *payload, const std::vector< char > *key, void *msg_opaque);
  生产和发送单条消息到Broker，传递消息数组和key数组。

```
ErrorCode produce (Topic *topic, int32_t partition,
                               const std::vector<char> *payload,
                               const std::vector<char> *key,
                               void *msg_opaque)
```

生产和发送消息到Broker，接受数组类型的key和payload，数组会被复制。

- ErrorCode flush (int timeout_ms)
  等待所有未完成的所有Produce请求完成。
  为了确保所有队列和已经执行的Produce请求在中止前完成，flush操作优先于销毁生产者实例完成。
  本函数会调用Producer::poll()函数，因此会触发回调函数。
- ErrorCode purge (int purge_flags)
  清理生产者当前处理的消息。本函数调用时可能会阻塞一定时间，当后台线程队列在清理时。
  应用程序需要在调用poll或flush函数后，执行清理消息的dr_cb回调函数。
- virtual Error *init_transactions (int timeout_ms) = 0;
  初始化Producer实例的事务。
  失败返回RdKafka::Error错误对象，成功返回NULL。
  通过调用RdKafka::Error::is_retriable()函数可以检查返回的错误对象是否有权限重试，调用RdKafka::Error::is_fatal()检查返回的错误对象是否是严重错误。返回的错误对象必须delete。
- virtual Error *begin_transaction () = 0;
  启动事务。
  本函数调用前，init_transactions()函数必须被成功调用。
  成功返回NULL，失败返回错误对象。通过调用RdKafka::Error::is_fatal_error()函数可以检查是否是严重错误，返回的错误对象必须delete。
- virtual Error *send_offsets_to_transaction (const std::vector<TopicPartition*> &offsets,const ConsumerGroupMetadata *group_metadata,int timeout_ms) = 0;
  发送TopicPartition位移链表到由group_metadata指定的Consumer Group协调器，如果事务提交成功，位移才会被提交。
- virtual Error *commit_transaction (int timeout_ms) = 0;
  提交当前事务。在实际提交事务时，任何未完成的消息会被完成投递。
  成功返回NULL，失败返回错误对象。通过调用错误对象的方法可以检查是否有权限重试，是否是严重错误、可中止错误等。
- virtual Error *abort_transaction (int timeout_ms) = 0;
  停止事务。本函数从非严重错误、可终止事务中用于恢复。
  未完成消息会被清理。

9、RdKafka::Consumer（重点）

RdKafka::Consumer是简单的非Rebalance、非Group的消费者。

- static Consumer * create(Conf *conf, std::string &errstr);
  创建一个Kafka Consumer客户端对象
- static int64_t OffsetTail(int64_t offset);
  从Topic尾部转换位移为逻辑位移
- ErrorCode start(Topic *topic, int32_t partition, int64_t offset);
  从topic主题partition分区的offset位移开始消费消息，offset可以是普通位移，也可以是OFFSET_BEGINNING或OFFSET_END，rdkafka会试图从Broker重复拉取批量消息到本地队列使其维持queued.min.messages参数值数量的消息。start函数在没有调用stop函数停止消费时不能对同一个TopicPartition调用多次。
  应用程序会使用consume函数从本地队列消费消息。
- ErrorCode start(Topic *topic, int32_t partition, int64_t offset, Queue *queue);
  在消息队列queue的topic主题的partition分区开始消费。
- ErrorCode stop(Topic *topic, int32_t partition);
  停止从topic主题的partition分区消费消息，并清理本地队列的所有消息。应用程序需要在销毁所有Consumer对象前停止所有消费者。
- ErrorCode seek (Topic *topic, int32_t partition, int64_t offset, int timeout_ms)
  定位topic的partition分区的Consumer位移到offset
- Message * consume(Topic *topic, int32_t partition, int timeout_ms);
  从topic主题和partition分区消费一条消息。timeout_ms是等待获取消息的最大时间。消费者必须提前调用start函数。应用程序需要检查消费的消息是正常消息还是错误消息。应用程序完成时消息对象必须销毁。
- Message * consume(Queue *queue, int timeout_ms);
  从指定消息队列queue消费一条消息
- int consume_callback(Topic *topic, int32_t partition, int timeout_ms, ConsumeCb *consume_cb, void *opaque);
  从topic主题和partition分区消费消息，并对每条消费的消息使用指定回调函数处理。
  consume_callback提供了比consume更高的吞吐量。
  opaque参数回被传递给consume_cb的回调函数。
- int consume_callback(Queue *queue, int timeout_ms, RdKafka::ConsumeCb *consume_cb, void *opaque);
  从消息队列queue消费消息，并对每条消费的消息使用指定回调函数处理。

10、RdKafka::KafkaConsumer(重点)

KafkaConsumer是高级API，要求Kafka 0.9.0以上版本，当前支持range和roundrobin分区分配策略。

- static KafkaConsumer * create(Conf *conf, std::string &errstr);
  创建KafkaConsumer对象，conf对象必须配置Consumer要加入的消费者组。使用KafkaConsumer::close()进行关闭。
- ErrorCode assignment(std::vector< RdKafka::TopicPartition * > &partitions);
  返回由RdKafka::KafkaConsumer::assign() 设置的当前分区
- ErrorCode subscription(std::vector< std::string > &topics);
  返回由RdKafka::KafkaConsumer::subscribe() 设置的当前订阅Topic
- ErrorCode subscribe(const std::vector< std::string > &topics);
  更新订阅Topic分区
- ErrorCode unsubscribe();
  将当前订阅Topic取消订阅分区
- ErrorCode assign(const std::vector< TopicPartition * > &partitions);
  将分配分区更新为partitions

- ErrorCode unassign();
  停止消费并删除当前分配的分区
- Message * consume(int timeout_ms);
  消费消息或获取错误事件，触发回调函数，会自动调用注册的回调函数，包括RebalanceCb、EventCb、OffsetCommitCb等。需要使用delete释放消息。应用程序必须确保consume在指定时间间隔内调用，为了执行等待调用的回调函数，即使没有消息。当RebalanceCb被注册时，在需要调用和适当处理内部Consumer同步状态时，确保consume在指定时间间隔内调用极为重要。应用程序必须禁止对KafkaConsumer对象调用poll函数。
  如果RdKafka::Message::err()是ERR_NO_ERROR，则返回正常的消息；如果RdKafka::Message::err()是ERR_NO_ERRO，返回错误事件；如果RdKafka::Message::err()是ERR_TIMED_OUT，则超时。
- ErrorCode commitSync();
  提交当前分配分区的位移，同步操作，会阻塞直到位移被提交或提交失败。如果注册了RdKafka::OffsetCommitCb回调函数，其会在KafkaConsumer::consume()函数内调用并提交位移。
- ErrorCode commitAsync();
  异步提交位移
- ErrorCode commitSync(Message *message);
  基于消息对单个topic+partition对象同步提交位移
- virtual ErrorCode commitSync (std::vector<TopicPartition*> &offsets) = 0;
  对指定多个TopicPartition同步提交位移
- ErrorCode commitAsync(Message *message);
  基于消息对单个TopicPartition异步提交位移
- virtual ErrorCode commitAsync (const std::vector<TopicPartition*> &offsets) = 0;
  对多个TopicPartition异步提交位移
- ErrorCode close();
  正常关闭，会阻塞直到四个操作完成（触发避免当前分区分配的局部再平衡，停止当前赋值消费，提交位移，离开分组）
- virtual ConsumerGroupMetadata *groupMetadata () = 0;
  返回本Consumer实例的Consumer Group的元数据
- ErrorCode position (std::vector<TopicPartition*> &partitions)
  获取TopicPartition对象中当前位移，会别填充TopicPartition对象的offset字段。
- ErrorCode seek (const TopicPartition &partition, int timeout_ms)
  定位TopicPartition的Consumer到位移。
  timeout_ms为0，会开始Seek并立即返回；timeout_ms非0，Seek会等待timeout_ms时间。
- ErrorCode offsets_store (std::vector<TopicPartition*> &offsets)
  为TopicPartition存储位移，位移会在auto.commit.interval.ms时提交或是被手动提交。
  enable.auto.offset.store属性必须设置为fasle。

11、RdKafka::Event

```
enum  Type{
  EVENT_ERROR, //错误条件事件
  EVENT_STATS, // Json文档统计事件
  EVENT_LOG, // Log消息事件
  EVENT_THROTTLE // 来自Broker的throttle级信号事件
};
```

- virtual Type type() const =0;　　返回事件类型
- virtual ErrorCode err() const =0;　　返回事件错误代码
- virtual Severity severity() const =0;　　返回log严重级别

- virtual std::string fac() const =0;　　返回log基础字符串
- virtual std::string str () const =0;　　返回Log消息字符串
- virtual int throttle_time() const =0;　　返回throttle时间
- virtual std::string broker_name() const =0; 返回Broker名称
- virtual int broker_id() const =0;　　返回Broker ID

12、RdKafka::Queue

创建新的消息队列，消息队列运行客户端从多个topic+partitions对象重新路由消费消息到单个消息队列。包含多个topic+partitions对象的消息队列会运行一次consume()，而不是针对每个topic+partitions对象都执行。

- static Queue * create(Handle *handle);
  创建Kafka客户端的消息队列
  消息队列允许应用程序转发从多个Topic+Partition消费的消息到队列点。
- virtual ErrorCode forward (Queue *dst) = 0;
  将消息队列消息转移到dst消息队列。无论dst是否为NULL，调用本函数后，src不会转移其fetch队列到消费者队列。
- virtual Message *consume (int timeout_ms) = 0;
  从消息队列中消费消息或获取错误事件。释放消息需要使用delete。
- virtual int poll (int timeout_ms) = 0;
  poll消息队列，在任何入队回调函数会运行。禁止对包含消息的队列使用。返回事件数量，超时返回0。
- virtual void io_event_enable (int fd, const void *payload, size_t size) = 0;
  开启消息队列的IO事件触发。fd=-1，关闭事件触发。RdKafka会维护一个payload的拷贝。使用转移队列时，IO事件触发必须打开。

13、RdKafka::BrokerMetadata

- virtual int32_t id() const =0;
  返回Broker的ID
- virtual const std::string host() const =0;
  返回Broker主机
- virtual int port() const =0;
  返回Broker监听端口

14、RdKafka::Metadata

```
typedef std::vector<const BrokerMetadata*> BrokerMetadataVector;
typedef std::vector<const TopicMetadata*> TopicMetadataVector;
typedef BrokerMetadataVector::const_iterator BrokerMetadataIterator;
typedef TopicMetadataVector::const_iterator TopicMetadataIterator;
```

- virtual const BrokerMetadataVector * brokers() const =0;
  返回Broker链表
- virtual const TopicMetadataVector * topics() const =0;
  返回Topic链表
- virtual int32_t orig_broker_id() const =0;
  返回metadata所在Broker的ID
- virtual const std::string orig_broker_name() const =0;
  返回metadata所在Broker的名称

15、RdKafka::ConsumeCb

- virtual void consume_cb(Message &message, void *opaque)=0;
  ConsumeCb用于RdKafka::Consumer::consume_callback()接口，对消费的每条消息会调用ConsumeCb回调函数。

16、RdKafka::DeliveryReportCb

每收到一条RdKafka::Producer::produce()函数生产的消息，调用一次投递报告回调函数，RdKafka::Message::err()将会标识Produce请求的结果。为了使用队列化的投递报告回调函数，必须调用RdKafka::poll()函数。

- virtual void dr_cb(Message &message)=0;
  当一条消息成功生产或是rdkafka遇到永久失败或是重试次数耗尽，
  投递报告回调函数会被调用。

17、RdKafka::EventCb

事件是从RdKafka传递错误、统计信息、日志等消息到应用程序的通用接口。

- virtual void event_cb(Event &event)=0;
  事件回调函数

18、RdKafka::OffsetCommitCb

- virtual void offset_commit_cb(RdKafka::ErrorCode err, std::vector< TopicPartition * > &offsets)=0;
  用于消费者组的位移提交回调函数。
  自动或手动提交位移的结果回被位移提交回调函数并被RdKafka::KafkaConsumer::consume()函数使用。
  如果没有分区有合法的位移要提交，位移提交回调函数会被调用，此时err为ERR_NO_OFFSET。
  offsets链表包含每个分区的信息，提交的Topic、Partition、offset、提交错误。

19、RdKafka::OpenCb

- virtual int open_cb(const std::string &path, int flags, int mode)=0;
  Open回调函数用于使用flags、mode打开指定path的文件。

20、RdKafka::PartitionerCb

PartitionerCb用实现自定义分区策略，需要使用RdKafka::Conf::set()设置partitioner_cb属性。

- virtual int32_t partitioner_cb(const Topic *topic, const std::string *key, int32_t partition_cnt, void *msg_opaque)=0;
  Partitioner回调函数
  返回topic主题中使用key的分区，key可以是NULL或字符串。
  返回值必须在0到partition_cnt间，如果分区失败可能返回RD_KAFKA_PARTITION_UA(-1)。
  msg_opaque与RdKafka::Producer::produce()调用提供的msg_opaque相同。

21、RdKafka::PartitionerKeyPointerCb

- virtual int32_t partitioner_cb(const Topic *topic, const void *key, size_t key_len, int32_t partition_cnt, void *msg_opaque)=0;
  变体partitioner回调函数

使用key指针及其长度替代字符串类型key。
key可以为NULL，key_len可以为0。

22、RdKafka::PartitionMetadata

```
typedef std::vector<int32_t> ReplicasVector;
typedef std::vector<int32_t> ISRSVector;
typedef ReplicasVector::const_iterator ReplicasIterator;
typedef ISRSVector::const_iterator ISRSIterator;
```

- virtual int32_t id() const =0;
  返回分区ID
- virtual ErrorCode err() const =0;
  返回Broker报告的分区错误
- virtual int32_t leader() const =0;
  返回分区Leader的Broker ID
- virtual const std::vector<int32_t> * replicas() const =0;
  返回备份Broker链表
- virtual const std::vector<int32_t> * isrs() const =0;
  返回ISR Broker链表，Broker可能会返回一个缓存或过期的ISR链表。

23、RdKafka::RebalanceCb

- virtual void rebalance_cb(RdKafka::KafkaConsumer *consumer, RdKafka::ErrorCode err, std::vector< TopicPartition * > &partitions)=0;
  用于RdKafka::KafkaConsunmer的组再平衡回调函数
  注册rebalance_cb回调函数会关闭rdkafka的自动分区赋值和再分配并替换应用程序的rebalance_cb回调函数。
  再平衡回调函数负责对基于RdKafka::ERR_ASSIGN_PARTITIONS和RdKafka::ERR_REVOKE_PARTITIONS事件更新rdkafka的分区分配，也能处理任意前两者错误除外其它再平衡失败错误。对于RdKafka::ERR_ASSIGN_PARTITIONS和RdKafka::ERR_REVOKE_PARTITIONS事件之外的其它再平衡失败错误，必须调用unassign()同步状态。
  没有再平衡回调函数，rdkafka也能自动完成再平衡过程，但注册一个再平衡回调函数可以使应用程序在执行其它操作时拥有更大的灵活性，例如从指定位置获取位移或手动提交位移。

```
class MyRebalanceCb : public RdKafka::RebalanceCb
{
public:
    void rebalance_cb (RdKafka::KafkaConsumer *consumer,
                       RdKafka::ErrorCode err,
                       std::vector<RdKafka::TopicPartition*> &partitions)
    {
        if (err == RdKafka::ERR__ASSIGN_PARTITIONS)
        {
            // application may load offets from arbitrary external
            // storage here and update \p partitions
            consumer->assign(partitions);

        }
        else if (err == RdKafka::ERR__REVOKE_PARTITIONS)
        {
            // Application may commit offsets manually here
```

```
            // if auto.commit.enable=false
            consumer->unassign();
        }
        else
        {
            std::cerr << "Rebalancing error: " <<
                         RdKafka::err2str(err) << std::endl;
            consumer->unassign();
        }
    }
};
```

24、RdKafka::SocketCb

SocketCb回调函数用于打开一个Socket套接字。
virtual int socket_cb(int domain, int type, int protocol)=0;
Socket回调函数
用于打开使用domain、type、protocol创建的Socket连接。

25、RdKafka::Error

static Error *create (ErrorCode code, const std::string *errstr);
创建Kafka错误对象，RdKafka::Error对象必须要显示释放。
virtual ErrorCode code () const = 0;
返回Kafka错误的错误码
virtual std::string name () const = 0;
返回Kafka错误的错误码名称
virtual std::string str () const = 0;
返回Kafka错误的错误描述
virtual bool is_fatal () const = 0;
如果Kafka错误会导致客户端不可用的fatal错误，返回1，否则返回0。
virtual bool is_retriable () const = 0;
如果操作可重试，返回1，否则返回0。
virtual bool txn_requires_abort () const = 0;
如果Kafka错误是可终止的事务型错误，返回1，否则返回0。

26、RdKafka::TopicMetadata

```
typedef std::vector<const PartitionMetadata*> PartitionMetadataVector;
typedef PartitionMetadataVector::const_iterator PartitionMetadataIterator;
```

virtual const std::string topic() const = 0;
返回Topic名称。
virtual const PartitionMetadataVector *partitions() const = 0;
返回Partition列表
virtual ErrorCode err() const = 0;
返回Broker报告的Topic错误。

二、Kafka Producer C++ API封装

1、Kafka Producer使用流程

（1）创建Kafka配置实例
RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL)
（2）创建Topic配置实例
RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC)
（3）设置Kafka配置实例Broker属性
RdKafka::Conf::ConfResult RdKafka::Conf::set(const std::string &name, const std::string &value, std::string &errstr)
（4）设置Topic配置实例属性
RdKafka::Conf::ConfResult RdKafka::Conf::set (const std::string &name, const std::string &value, std::string &errstr)
（5）注册回调函数

```
//事件是从RdKafka传递错误、统计信息、日志等消息到应用程序的通用接口
Conf::ConfResult RdKafka::Conf::set ("event_cb", RdKafka::EventCb *dr_cb,
std::string &errstr);
//SocketCb回调函数用于打开一个Socket套接字
Conf::ConfResult RdKafka::Conf::set ("socket_cb", RdKafka::SocketCb *socket_cb,
std::string &errstr);
//Open回调函数用于使用flags、mode打开指定path的文件
Conf::ConfResult RdKafka::Conf::set ("open_cb", RdKafka::OpenCb *open_cb,
std::string &errstr);
//用于RdKafka::KafkaConsunmer的组再平衡回调函数
Conf::ConfResult RdKafka::Conf::set ("rebalance_cb", RdKafka::RebalanceCb
*rebalance_cb, std::string &errstr);
//用于消费者组的位移提交回调函数
Conf::ConfResult RdKafka::Conf::set ("offset_commit_cb", RdKafka::OffsetCommitCb
*offset_commit_cb, std::string &errstr);
// 对消费的每条消息会调用ConsumeCb回调函数
Conf::ConfResult RdKafka::Conf::set ("consume_cb", RdKafka::ConsumeCb
*consume_cb, std::string &errstr);
```

分区策略回调函数需要注册到Topic配置实例：

```
Conf::ConfResult RdKafka::Conf::set ("partitioner_cb", RdKafka::PartitionerCb
*dr_cb, std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("partitioner_key_pointer_cb",
RdKafka::PartitionerKeyPointerCb *dr_cb, std::string &errstr);
```

（6）创建Kafka Producer客户端实例
static RdKafka::Producer* RdKafka::Producer::create(RdKafka::Conf *conf, std::string &errstr);
conf为Kafka配置实例
（7）创建Topic实例

```
static RdKafka::Topic* RdKafka::Topic::create(RdKafka::Handle *base,
                                              const std::string &topic_str,
                                              RdKafka::Conf *conf,
                                              std::string &errstr);
```

conf为Topic配置实例
（8）生产消息

```
RdKafka::ErrorCode RdKafka::Producer::produce(RdKafka::Topic *topic,
                                              int32_t partition,
                                              int msgflags,
                                              void *payload,
                                              size_t len,
                                              const std::string *key,
                                              void *msg_opaque);
```

（9）阻塞等待Producer生产消息完成
int RdKafka::Producer::poll (int timeout_ms);
（10）等待Produce请求完成
RdKafka::ErrorCode RdKafka::Producer::flush(int timeout_ms);
（11）销毁Kafka Producer客户端实例
int RdKafka::wait_destroyed(int timeout_ms);

2、Kafka Producer实例

KafkaProducer.h文件：

```
#ifndef KAFKAPRODUCER_H
#define KAFKAPRODUCER_H

#pragma once
#include <string>
#include <iostream>
#include "rdkafkacpp.h"

class ProducerDeliveryReportCb : public RdKafka::DeliveryReportCb
{
public:
    void dr_cb(RdKafka::Message &message)
    {
        if (message.err())
            std::cerr << "Message delivery failed: " << message.errstr() <<
std::endl;
        else
            std::cerr << "Message delivered to topic " << message.topic_name()
                      << " [" << message.partition() << "] at offset "
                      << message.offset() << std::endl;
    }
};

class ProducerEventCb : public RdKafka::EventCb
{
public:
    void event_cb(RdKafka::Event &event)
    {
        switch(event.type())
        {
        case RdKafka::Event::EVENT_ERROR:
            std::cout << "RdKafka::Event::EVENT_ERROR: " <<
RdKafka::err2str(event.err()) << std::endl;
            break;
        case RdKafka::Event::EVENT_STATS:
```

```
            std::cout << "RdKafka::Event::EVENT_STATS: " << event.str() <<
std::endl;
            break;
        case RdKafka::Event::EVENT_LOG:
            std::cout << "RdKafka::Event::EVENT_LOG " << event.fac() <<
std::endl;
            break;
        case RdKafka::Event::EVENT_THROTTLE:
            std::cout << "RdKafka::Event::EVENT_THROTTLE " <<
event.broker_name() << std::endl;
            break;
        }
    }
};

class HashPartitionerCb : public RdKafka::PartitionerCb
{
public:
    int32_t partitioner_cb (const RdKafka::Topic *topic, const std::string *key,
                            int32_t partition_cnt, void *msg_opaque)
    {
        char msg[128] = {0};
        sprintf(msg, "HashPartitionerCb:[%s][%s][%d]", topic->name().c_str(),
                key->c_str(), partition_cnt);
        std::cout << msg << std::endl;
        return generate_hash(key->c_str(), key->size()) % partition_cnt;
    }
private:

    static inline unsigned int generate_hash(const char *str, size_t len)
    {
        unsigned int hash = 5381;
        for (size_t i = 0 ; i < len ; i++)
            hash = ((hash << 5) + hash) + str[i];
        return hash;
    }
};

class KafkaProducer
{
public:
    /**
     * @brief KafkaProducer
     * @param brokers
     * @param topic
     * @param partition
     */
    explicit KafkaProducer(const std::string& brokers, const std::string& topic,
                           int partition);
    /**
     * @brief push Message to Kafka
     * @param str, message data
     */
    void pushMessage(const std::string& str, const std::string& key);
    ~KafkaProducer();

protected:
    std::string m_brokers;//Broker列表，多个使用逗号分隔
```

```
    std::string m_topicStr;// Topic名称
    int m_partition;// 分区
    RdKafka::Conf* m_config;// Kafka Conf对象
    RdKafka::Conf* m_topicConfig;// Topic Conf对象
    RdKafka::Topic* m_topic;// Topic对象
    RdKafka::Producer* m_producer;// Producer对象
    RdKafka::DeliveryReportCb* m_dr_cb;
    RdKafka::EventCb* m_event_cb;
    RdKafka::PartitionerCb* m_partitioner_cb;
};

#endif // KAFKAPRODUCER_H
```

KafkaProducer.cpp文件:

```
#include "KafkaProducer.h"

KafkaProducer::KafkaProducer(const std::string& brokers, const std::string&
topic, int partition)
{
    m_brokers = brokers;
    m_topicStr = topic;
    m_partition = partition;
    // 创建Kafka Conf对象
    m_config = RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL);
    if(m_config == NULL)
    {
        std::cout << "Create RdKafka Conf failed." << std::endl;
    }
    // 创建Topic Conf对象
    m_topicConfig = RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC);
    if(m_topicConfig == NULL)
    {
        std::cout << "Create RdKafka Topic Conf failed." << std::endl;
    }
    // 设置Broker属性
    RdKafka::Conf::ConfResult errCode;
    m_dr_cb = new ProducerDeliveryReportCb;
    std::string errorStr;
    errCode = m_config->set("dr_cb", m_dr_cb, errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
    }
    m_event_cb = new ProducerEventCb;
    errCode = m_config->set("event_cb", m_event_cb, errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
    }

    m_partitioner_cb = new HashPartitionerCb;
    errCode = m_topicConfig->set("partitioner_cb", m_partitioner_cb, errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
```

```
    }

    errCode = m_config->set("statistics.interval.ms", "10000", errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
    }

    errCode = m_config->set("message.max.bytes", "10240000", errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
    }
    errCode = m_config->set("bootstrap.servers", m_brokers, errorStr);
    if(errCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed:" << errorStr << std::endl;
    }
    // 创建Producer
    m_producer = RdKafka::Producer::create(m_config, errorStr);
    if(m_producer == NULL)
    {
        std::cout << "Create Producer failed:" << errorStr << std::endl;
    }
    // 创建Topic对象
    m_topic = RdKafka::Topic::create(m_producer, m_topicStr, m_topicConfig,
errorStr);
    if(m_topic == NULL)
    {
        std::cout << "Create Topic failed:" << errorStr << std::endl;
    }
}

void KafkaProducer::pushMessage(const std::string& str, const std::string& key)
{
    int32_t len = str.length();
    void* payload = const_cast<void*>(static_cast<const void*>(str.data()));
    RdKafka::ErrorCode errorCode = m_producer->produce(m_topic,
RdKafka::Topic::PARTITION_UA,
                                    RdKafka::Producer::RK_MSG_COPY,
                                    payload, len, &key, NULL);
    m_producer->poll(0);
    if (errorCode != RdKafka::ERR_NO_ERROR)
    {
        std::cerr << "Produce failed: " << RdKafka::err2str(errorCode) <<
std::endl;
        if(errorCode ==  RdKafka::ERR__QUEUE_FULL)
        {
            m_producer->poll(1000);
        }
    }
}

KafkaProducer::~KafkaProducer()
{
    while (m_producer->outq_len() > 0)
    {
        std::cerr << "Waiting for " << m_producer->outq_len() << std::endl;
```

```
        m_producer->flush(5000);
    }
    delete m_config;
    delete m_topicConfig;
    delete m_topic;
    delete m_producer;
    delete m_dr_cb;
    delete m_event_cb;
    delete m_partitioner_cb;
}
```

main.cpp:

```
#include <iostream>
#include "KafkaProducer.h"
using namespace std;

int main()
{
    // 创建Producer
    KafkaProducer producer("192.168.0.105:9092", "test", 0);
    for(int i = 0; i < 10000; i++)
    {
        char msg[64] = {0};
        sprintf(msg, "%s%4d", "Hello RdKafka", i);
        // 生产消息
        char key[8] = {0};
        sprintf(key, "%d", i);
        producer.pushMessage(msg, key);
    }
    RdKafka::wait_destroyed(5000);
}
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)

project(KafkaProducer)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_COMPILER "g++")
set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS}")
set(CMAKE_INCLUDE_CURRENT_DIR ON)

# Kafka头文件路径
include_directories(/usr/local/include/librdkafka)
# Kafka库路径
link_directories(/usr/local/lib)

aux_source_directory(. SOURCE)

add_executable(${PROJECT_NAME} ${SOURCE})
TARGET_LINK_LIBRARIES(${PROJECT_NAME} rdkafka++)
```

3、Kafka消息查看

进入kafka容器：
docker exec -it kafka-test /bin/bash
查看Topic的消息：
kafka-console-consumer.sh --bootstrap-server kafka-test:9092 --topic test --from-beginning

三、Kafka Consumer C++ API封装

1、Kafka Consumer使用流程

RdKafka提供了两种消费者API，低级API的Consumer和高级API的KafkaConsumer，本文使用KafkaConsumer。
（1）创建Kafka配置实例
RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL)
（2）创建Topic配置实例
RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC)
（3）设置Kafka配置实例Broker属性
RdKafka::Conf::ConfResult RdKafka::Conf::set(const std::string &name, const std::string &value, std::string &errstr)
（4）设置Topic配置实例属性
RdKafka::Conf::ConfResult RdKafka::Conf::set (const std::string &name, const std::string &value, std::string &errstr)
（5）注册回调函数

```
Conf::ConfResult RdKafka::Conf::set ("event_cb", RdKafka::EventCb *dr_cb,
std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("socket_cb", RdKafka::SocketCb *socket_cb,
std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("open_cb", RdKafka::OpenCb *open_cb,
std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("rebalance_cb", RdKafka::RebalanceCb
*rebalance_cb, std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("offset_commit_cb", RdKafka::OffsetCommitCb
*offset_commit_cb, std::string &errstr);
Conf::ConfResult RdKafka::Conf::set ("consume_cb", RdKafka::ConsumeCb
*consume_cb, std::string &errstr);
```

（6）创建Kafka Consumer客户端实例
static RdKafka::KafkaConsumer* RdKafka::KafkaConsumer::create(RdKafka::Conf *conf, std::string &errstr);
conf为Kafka配置实例
（7）创建Topic实例

```
static RdKafka::Topic* RdKafka::Topic::create(RdKafka::Handle *base,
                                              const std::string &topic_str,
                                              RdKafka::Conf *conf,
                                              std::string &errstr);
```

conf为Topic配置实例
（8）订阅主题
RdKafka::ErrorCode RdKafka::KafkaConsumer::subscribe(const std::vectorstd::string &topics);
（9）消费消息
RdKafka::Message* RdKafka::KafkaConsumer::consume (int timeout_ms);
（10）关闭消费者实例
RdKafka::ErrorCode RdKafka::KafkaConsumer::close();
（11）销毁释放RdKafka资源
int RdKafka::wait_destroyed(int timeout_ms);

2、Kafka Consumer实例

KafkaConsumer.h文件：

```
#ifndef KAFKACONSUMER_H
#define KAFKACONSUMER_H

#pragma once

#include <string>
#include <iostream>
#include <vector>
#include <stdio.h>
#include "rdkafkacpp.h"

class ConsumerEventCb : public RdKafka::EventCb
{
public:
    void event_cb (RdKafka::Event &event)
    {
        switch (event.type())
        {
        case RdKafka::Event::EVENT_ERROR:
            if (event.fatal())
            {
                std::cerr << "FATAL ";
            }
            std::cerr << "ERROR (" << RdKafka::err2str(event.err()) << "): " <<
                      event.str() << std::endl;
            break;

        case RdKafka::Event::EVENT_STATS:
            std::cerr << "\"STATS\": " << event.str() << std::endl;
            break;

        case RdKafka::Event::EVENT_LOG:
            fprintf(stderr, "LOG-%i-%s: %s\n",
                    event.severity(), event.fac().c_str(), event.str().c_str());
            break;

        case RdKafka::Event::EVENT_THROTTLE:
            std::cerr << "THROTTLED: " << event.throttle_time() << "ms by " <<
                      event.broker_name() << " id " << (int)event.broker_id() <<
std::endl;
            break;
```

```
        default:
            std::cerr << "EVENT " << event.type() <<
                      " (" << RdKafka::err2str(event.err()) << "): " <<
                      event.str() << std::endl;
            break;
        }
    }
};

class ConsumerRebalanceCb : public RdKafka::RebalanceCb
{
private:
    static void printTopicPartition (const
std::vector<RdKafka::TopicPartition*>&partitions)
    {
        for (unsigned int i = 0 ; i < partitions.size() ; i++)
            std::cerr << partitions[i]->topic() <<
                      "[" << partitions[i]->partition() << "], ";
        std::cerr << "\n";
    }

public:
    void rebalance_cb (RdKafka::KafkaConsumer *consumer,
                       RdKafka::ErrorCode err,
                       std::vector<RdKafka::TopicPartition*> &partitions)
    {
        std::cerr << "RebalanceCb: " << RdKafka::err2str(err) << ": ";
        printTopicPartition(partitions);
        if (err == RdKafka::ERR__ASSIGN_PARTITIONS)
        {
            consumer->assign(partitions);
            partition_count = (int)partitions.size();
        }
        else
        {
            consumer->unassign();
            partition_count = 0;
        }
    }
private:
    int partition_count;
};

class KafkaConsumer
{
public:/**
     * @brief KafkaConsumer
     * @param brokers
     * @param groupID
     * @param topics
     * @param partition
     */
    explicit KafkaConsumer(const std::string& brokers, const std::string&
groupID,
                           const std::vector<std::string>& topics, int
partition);
    void pullMessage();
    ~KafkaConsumer();
```

```
protected:
    std::string m_brokers;
    std::string m_groupID;
    std::vector<std::string> m_topicVector;
    int m_partition;
    RdKafka::Conf* m_config;
    RdKafka::Conf* m_topicConfig;
    RdKafka::KafkaConsumer* m_consumer;
    RdKafka::EventCb* m_event_cb;
    RdKafka::RebalanceCb* m_rebalance_cb;
};

#endif // KAFKACONSUMER_H
```

KafkaConsumer.cpp文件:

```
#include "KafkaConsumer.h"

KafkaConsumer::KafkaConsumer(const std::string& brokers, const std::string&
groupID,
                             const std::vector<std::string>& topics, int
partition)
{
    m_brokers = brokers;
    m_groupID = groupID;
    m_topicVector = topics;
    m_partition = partition;

    std::string errorStr;
    RdKafka::Conf::ConfResult errorCode;
    m_config = RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL);

    m_event_cb = new ConsumerEventCb;
    errorCode = m_config->set("event_cb", m_event_cb, errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }

    m_rebalance_cb = new ConsumerRebalanceCb;
    errorCode = m_config->set("rebalance_cb", m_rebalance_cb, errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }

    errorCode = m_config->set("enable.partition.eof", "false", errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }

    errorCode = m_config->set("group.id", m_groupID, errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
```

```
    }
    errorCode = m_config->set("bootstrap.servers", m_brokers, errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }
    errorCode = m_config->set("max.partition.fetch.bytes", "1024000", errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }

    m_topicConfig = RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC);
    // 获取最新的消息数据
    errorCode = m_topicConfig->set("auto.offset.reset", "latest", errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Topic Conf set failed: " << errorStr << std::endl;
    }
    errorCode = m_config->set("default_topic_conf", m_topicConfig, errorStr);
    if(errorCode != RdKafka::Conf::CONF_OK)
    {
        std::cout << "Conf set failed: " << errorStr << std::endl;
    }
    m_consumer = RdKafka::KafkaConsumer::create(m_config, errorStr);
    if(m_consumer == NULL)
    {
        std::cout << "Create KafkaConsumer failed: " << errorStr << std::endl;
    }
    std::cout << "Created consumer " << m_consumer->name() << std::endl;
}

void msg_consume(RdKafka::Message* msg, void* opaque)
{
    switch (msg->err())
    {
    case RdKafka::ERR__TIMED_OUT:
        std::cerr << "Consumer error: " << msg->errstr() << std::endl;
        break;
    case RdKafka::ERR_NO_ERROR:
        std::cout << " Message in " << msg->topic_name() << " ["
                  << msg->partition() << "] at offset " << msg->offset()
                  << "key: " << msg->key() << " payload: "
                  << (char*)msg->payload() << std::endl;
        break;
    default:
        std::cerr << "Consumer error: " << msg->errstr() << std::endl;
        break;
    }
}

void KafkaConsumer::pullMessage()
{
    // 订阅Topic
    RdKafka::ErrorCode errorCode = m_consumer->subscribe(m_topicVector);
    if (errorCode != RdKafka::ERR_NO_ERROR)
    {
```

```
        std::cout << "subscribe failed: " << RdKafka::err2str(errorCode) <<
std::endl;
    }
    // 消费消息
    while(true)
    {
        RdKafka::Message *msg = m_consumer->consume(1000);
        msg_consume(msg, NULL);
        delete msg;
    }
}

KafkaConsumer::~KafkaConsumer()
{
    m_consumer->close();
    delete m_config;
    delete m_topicConfig;
    delete m_consumer;
    delete m_event_cb;
    delete m_rebalance_cb;

}
```

main.cpp文件:

```
#include "KafkaConsumer.h"

int main()
{
    std::string brokers = "192.168.0.105:9092";
    std::vector<std::string> topics;
    topics.push_back("test");
    topics.push_back("test2");
    std::string group = "testGroup";
    KafkaConsumer consumer(brokers, group, topics,
RdKafka::Topic::OFFSET_BEGINNING);
    consumer.pullMessage();

    RdKafka::wait_destroyed(5000);
    return 0;
}
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)

project(KafkaConsumer)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_COMPILER "g++")
set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS}")
set(CMAKE_INCLUDE_CURRENT_DIR ON)

# Kafka头文件路径
```

```
include_directories(/usr/local/include/librdkafka)
# Kafka库路径
link_directories(/usr/local/lib)

aux_source_directory(. SOURCE)

add_executable(${PROJECT_NAME} ${SOURCE})
TARGET_LINK_LIBRARIES(${PROJECT_NAME} rdkafka++)
```

https://github.com/scorpiostudio/Kafka

四、librdkafka配置参数

官方对应的配置参数：https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md

1、全局配置属性

属性	C/P	范围	默认值	描述
builtin.features	*		gzip, snappy, ssl, sasl, regex, lz4	标示该librdkafka的支持的内建特性。应用程序可以查看或设置这些值来检查是否支持这些特性。 *Type: CSV flags*

属性	C/P	范围	默认值	描述
builtin.features	*		gzip, snappy, ssl, sasl, regex, lz4	标示该librdkafka的支持的内建特性。应用程序可以查看或设置这些值来检查是否支持这些特性。
Type: CSV flags				
client.id	*		rdkafka	客户端标示。*Type: string*
metadata.broker.list	*			初始化的broker列表。应用程序也可以使用 rd_kafka_brokers_add() 在运行时添加 broker。
Type: string				
bootstrap.servers	*			参考 metadata.broker.list
message.max.bytes	*	1000 .. 1000000000	1000000	最大发送消息大小。
Type: integer				
message.copy.max.bytes	*	0 .. 1000000000	65535	消息拷贝到缓存的最大大小。如果消息大于这个值，将会消耗更多的iovec而采用引用（零拷贝）方式。
Type: integer				
receive.message.max.bytes	*	1000 .. 1000000000	100000000	最大接收消息大小。这是一个安全预防措施，防止协议饱和时内存耗尽。这个值至少为 fetch.message.max.bytes * 消费者分区数 + 消息头大小 (e.g. 200000 bytes).
Type: integer				
max.in.flight.requests.per.connection	*	1 .. 1000000	1000000	客户端保持的最大发送请求数。 该配置应用于每一个 broker 连接.
Type: integer				
metadata.request.timeout.ms	*	10 .. 900000	60000	无数据请求超时时间，毫秒。 适用于 metadata 请求等。
Type: integer				
topic.metadata.refresh.interval.ms	*	-1 .. 3600000	300000	Topic metadata 刷新间隔，毫秒。metadata 自动刷新错误和连接。设置为 -1 关闭刷新间隔。
Type: integer				
metadata.max.age.ms	*			参考 topic.metadata.refresh.interval.ms
topic.metadata.refresh.fast.cnt	*	0 .. 1000	10	当 topic 丢失 leader， metadata 请求的发送次数，发送间隔是 topic.metadata.refresh.fast.interval.ms 而不是 topic.metadata.refresh.interval.ms。 该配置用于快速修复broker leader。
Type: integer				
topic.metadata.refresh.fast.interval.ms	*	1 .. 60000	250	参考 topic.metadata.refresh.fast.cnt。
Type: integer				
topic.metadata.refresh.sparse	*	true, false	true	极少的 metadata 请求 (消费者的网络带宽很小)
Type: boolean				
topic.blacklist	*			Topic 黑名单，逗号分隔的正则表达式列表，匹配 topic名字，匹配到的 topic 如果不存在，就在 broker metadata 信息中忽略。
Type: pattern list				
debug	*	generic, broker, topic, metadata, queue, msg, protocol, cgrp, security, fetch, feature, all		逗号分隔的列表，控制 debug 上下文。调试生产者：broker,topic,msg。调试消费者：cgrp,topic,fetch
Type: CSV flags				
socket.timeout.ms	*	10 .. 300000	60000	网络请求超时时间。
Type: integer				
socket.blocking.max.ms	*	1 .. 60000	100	broker 在 socket 操作时最大阻塞时间。值越低，响应越快，但会略微提高CPU使用率。
Type: integer				
socket.send.buffer.bytes	*	0 .. 100000000	0	Broker socket 发送缓冲大小。系统默认为 0。
Type: integer				
socket.receive.buffer.bytes	*	0 .. 100000000	0	Broker socket 接收缓冲大小。系统默认为 0。
Type: integer				
socket.keepalive.enable	*	true, false	false	Broker sockets 允许 TCP 保持活力 (SO_KEEPALIVE)。

属性	C/P	范围	默认值	描述
Type: boolean				
socket.max.fails	*	0 .. 1000000	3	Broker 关闭连接的最大错误次数(e.g., timed out requests)。0不关闭。提示：连接自动重新建立。
Type: integer				
broker.address.ttl	*	0 .. 86400000	1000	保存 broker 地址响应结果的时间 (毫秒)。
Type: integer				
broker.address.family	*	any, v4, v6	any	允许的 broker IP 地址族：any, v4, v6。
Type: enum value				
reconnect.backoff.jitter.ms	*	0 .. 3600000	500	通过这个值调节 broker 重连尝试 +-50%。
Type: integer				
statistics.interval.ms	*	0 .. 86400000	0	librdkafka 统计间隔。应用程序需要通过 rd_kafka_conf_set_stats_cb()设置统计的回调函数。粒度是 1000ms. 0 关闭统计。
Type: integer				
enabled_events	*	0 .. 2147483647	0	参考 rd_kafka_conf_set_events()。
Type: integer				
error_cb	*			错误回调函数 (参考 rd_kafka_conf_set_error_cb())
Type: pointer				
throttle_cb	*			调节回调函数 (参考 rd_kafka_conf_set_throttle_cb())
Type: pointer				
stats_cb	*			统计回调函数 (参考 rd_kafka_conf_set_stats_cb())
Type: pointer				
log_cb	*			日志回调函数 (参考 rd_kafka_conf_set_log_cb())
Type: pointer				
log_level	*	0 .. 7	6	日志界别 (syslog(3) levels)
Type: integer				
log.thread.name	*	true, false	false	在日志消息中打印内部线程名。(useful for debugging librdkafka internals)
Type: boolean				
log.connection.close	*	true, false	true	记录 broker 断开连接。由于受 0.9 版本 broker 的 connection.max.idle.ms 的影响，最好关闭。
Type: boolean				
socket_cb	*			为Socket创建回调函数提供无缝 CLOEXEC
Type: pointer				
open_cb	*			为文件打开回调函数提供无缝 CLOEXEC
Type: pointer				
opaque	*			对应用程序不开放 (set with rd_kafka_conf_set_opaque())
Type: pointer				
default_topic_conf	*			默认 topic 配置，用于自动订阅 topics
Type: pointer				
internal.termination.signal	*	0 .. 128	0	用于 librdkafka 调用 rd_kafka_destroy() 快速终止的信号。如果没有设置信号， 终止过程会延迟直到所有内部线程的系统调用超时返回，且 rd_kafka_wait_destroyed() 返回 true。如果设置了信号，延迟会最小化。应用程序需要屏蔽该信号，而作为内部信号句柄。
Type: integer				
api.version.request	*	true, false	false	请求 broker 支持的API版本，调整可用协议特性的功能。如果设置为false，将使用 broker.version.fallback设置的回退版本。 **提示**: 依赖的 broker 版本 >=0.10.0。如果 broker（老版本）不支持该请求，使用 broker.version.fallback设置的回退版本。
Type: boolean				
api.version.fallback.ms	*	0 .. 604800000	1200000	配置 ApiVersionRequest 失败多长时间后，使用 broker.version.fallback 回退版本。**提示**: ApiVersionRequest 只用新的 broker 能使用。
Type: integer				

属性	C/P	范围	默认值	描述
broker.version.fallback	*		0.9.0	老版本的 broker（<0.10.0）不支持客户端查询支持协议特性(ApiVersionRequest, see api.version.request)，所以要客户端不知道什么特性可以使用。 用户使用本属性指示 broker 版本，如果 ApiVersionRequest 失败（或不可用），客户端据此属性自动调整特性。与 api.version.fallback.ms 配合使用。有效值：0.9.0, 0.8.2, 0.8.1, 0.8.0.
Type: string				
security.protocol	*	plaintext, ssl, sasl_plaintext, sasl_ssl	plaintext	与 broker 通讯的协议。
Type: enum value				
ssl.cipher.suites	*			密码套件是个组合体，包括鉴权，加密，认证和秘钥交换程序，用于网络连接的安全设置交换，使用 TLS 或 SSL 网络协议。查看手册 ciphers(1) 和 `SSL_CTX_set_cipher_list(3)。
Type: string				
ssl.key.location	*			客户端的私钥(PEM)路径，用于鉴权。
Type: string				
ssl.key.password	*			私钥密码。
Type: string				
ssl.certificate.location	*			客户端的公钥(PEM)路径，用于鉴权。
Type: string				
ssl.ca.location	*			CA 证书文件或路径，用于校验 broker key。
Type: string				
ssl.crl.location	*			CRL 路径，用于 broker 的证书校验。
Type: string				
sasl.mechanisms	*	GSSAPI, PLAIN	GSSAPI	使用 SASL 机制鉴权。 支持：GSSAPI, PLAIN. **提示**: 只能配置一种机制名。
Type: string				
sasl.kerberos.service.name	*		kafka	Kafka 运行的 Kerberos 首要名。
Type: string				
sasl.kerberos.principal	*		kafkaclient	客户端的 Kerberos 首要名。
Type: string				
sasl.kerberos.kinit.cmd	*		kinit -S “%{sasl.kerberos.service.name}/%{broker.name}” -k -t “%{sasl.kerberos.keytab}” %{sasl.kerberos.principal}	完整的 kerberos kinit 命令串，%{config.prop.name} 替换为与配置对象一直的值，%{broker.name} broker 的主机名。
Type: string				
sasl.kerberos.keytab	*			Kerberos keytab 文件的路径。如果不设置，则使用系统默认的。**提示**：不会自动使用，必须在 sasl.kerberos.kinit.cmd 中添加到模板，如 ... -t %{sasl.kerberos.keytab}。
Type: string				
sasl.kerberos.min.time.before.relogin	*	1 .. 86400000	60000	Key 恢复尝试的最小时间，毫秒。
Type: integer				
sasl.username	*			使用 PLAIN 机制时，SASL 用户名。
Type: string				
sasl.password	*			使用 PLAIN 机制时，SASL 密码。
Type: string				
group.id	*			客户端分组字符串。同组的客户端使用相同的 group.id。
Type: string				
partition.assignment.strategy	*		range,roundrobin	partition 分配策略，当选举组 leader 时，分配 partition 给组成员的策略。
Type: string				
session.timeout.ms	*	1 .. 3600000	30000	客户端组会话探测失败超市时间。
Type: integer				
heartbeat.interval.ms	*	1 .. 3600000	1000	组会话保活心跳间隔。
Type: integer				

属性	C/P	范围	默认值	描述
group.protocol.type	*		consumer	组协议类型。
Type: string				
coordinator.query.interval.ms	*	1 .. 3600000	600000	多久查询一次当前的客户端组协调人。如果当前的分配协调人挂了，为了更快的恢复协调人，探测时间间隔会除以 10。
Type: integer				
enable.auto.commit	C	true, false	true	在后台周期性的自动提交偏移量。
Type: boolean				
auto.commit.interval.ms	C	0 .. 86400000	5000	消费者偏移量提交（写入）到存储的频率，毫秒。(0 = 不可用)
Type: integer				
enable.auto.offset.store	C	true, false	true	为应用程序自动保存最后消息的偏移量。
Type: boolean				
queued.min.messages	C	1 .. 10000000	100000	每一个 topic+partition，本地消费者队列的最小消息数。
Type: integer				
queued.max.messages.kbytes	C	1 .. 1000000000	1000000	每一个 topic+partition，本地消费者队列的最大大小，单位kilobytes。该值应该大于 fetch.message.max.bytes。
Type: integer				
fetch.wait.max.ms	C	0 .. 300000	100	为写满fetch.min.bytes，broker 的最大等待时间。
Type: integer				
fetch.message.max.bytes	C	1 .. 1000000000	1048576	每一个 topic+partition 初始化的最大大小（bytes）用于从 broker 读消息。如果客户端遇到消息大于这个值，会逐步扩大直到塞下这个消息。
Type: integer				
max.partition.fetch.bytes	C			参考 fetch.message.max.bytes
fetch.min.bytes	C	1 .. 100000000	1	broker 请求的最小数据大小，单位bytes。如果达到 fetch.wait.max.ms 时间，则不管这个配置，将已收到的数据发送给客户端。
Type: integer				
fetch.error.backoff.ms	C	0 .. 300000	500	对于 topic+partition，如果接受错误，下一个接受请求间隔多长时间。
Type: integer				
offset.store.method	C	none, file, broker	broker	偏移量存储方式：'file' - 本地文件存储 (offset.store.path, et.al), 'broker' - 在 broker 上提交存储 (要求 Apache Kafka 0.8.2 或以后版本)。
Type: enum value				
consume_cb	C			消息消费回调函数 (参考 rd_kafka_conf_set_consume_cb())
Type: pointer				
rebalance_cb	C			消费者组重新分配后调用 (参考 rd_kafka_conf_set_rebalance_cb())
Type: pointer				
offset_commit_cb	C			偏移量提交结果回调函数 (参考 rd_kafka_conf_set_offset_commit_cb())
Type: pointer				
enable.partition.eof	C	true, false	true	当消费者到达分区结尾，发送 RD_KAFKA_RESP_ERR__PARTITION_EOF 事件。
Type: boolean				
queue.buffering.max.messages	P	1 .. 10000000	100000	生产者队列允许的最大消息数。
Type: integer				
queue.buffering.max.kbytes	P	1 .. 2147483647	4000000	生产者队列允许的最大大小，单位kb。
Type: integer				
queue.buffering.max.ms	P	1 .. 900000	1000	生产者队列缓存数据的最大时间，毫秒。
Type: integer				
message.send.max.retries	P	0 .. 10000000	2	消息集发送失败重试次数。**提示** 重试会导致重排。
Type: integer				
retries	P			参考 message.send.max.retries

属性	C/P	范围	默认值	描述
retry.backoff.ms	P	1 .. 300000	100	重试消息发送前的补偿时间。
Type: integer				
compression.codec	P	none, gzip, snappy, lz4	none	压缩消息集使用的压缩编解码器。这里配置的是所有 topic 的默认值，可能会被 topic 上的 compression.codec 属性覆盖。
Type: enum value				
batch.num.messages	P	1 .. 1000000	10000	一个消息集最大打包消息数量。整个消息集的大小仍受限于 message.max.bytes。
Type: integer				
delivery.report.only.error	P	true, false	false	只对失败的消息提供分发报告。
Type: boolean				
dr_cb	P			分发报告回调函数 (参考 rd_kafka_conf_set_dr_cb())
Type: pointer				
dr_msg_cb	P			分发报告回调函数 (参考 rd_kafka_conf_set_dr_msg_cb())
Type: pointer				

2、Topic 配置属性

属性	C/P	范围	默认值	描述
request.required.acks	P	-1 .. 1000	1	这个字段标示 leader broker 要从 ISR broker 接收多少个 ack，然后才确认发送请求：*0*=不发送任何 response/ack 给客户端, *1*=只有 leader broker 需要 ack 消息, *-1* or *all*=broker 阻塞直到所有的同步备份(ISRs)或in.sync.replicas设置的备份返回消息提交的确认应答。
Type: integer				
acks	P			参考 request.required.acks
request.timeout.ms	P	1 .. 900000	5000	生产者请求等待应答的超时时间，毫秒。这个值仅在 broker 上强制执行。参考 request.required.acks，不能等于 0.
Type: integer				
message.timeout.ms	P	0 .. 900000	300000	本地消息超时时间。这个值仅在本地强制执行，限制生产的消息等待被成功发送的等待时间，0 是不限制。
Type: integer				
produce.offset.report	P	true, false	false	报告生产消息的偏移量给应用程序。应用程序必须使用 dr_msg_cb 从 rd_kafka_message_t.offset 中获取偏移量。
Type: boolean				
partitioner_cb	P			分区方法回调函数 (参考 rd_kafka_topic_conf_set_partitioner_cb())
Type: pointer				
opaque	*			应用程序不可见 (参考 rd_kafka_topic_conf_set_opaque())
Type: pointer				
compression.codec	P	none, gzip, snappy, lz4, inherit	inherit	压缩消息集的压缩编解码器。
Type: enum value				
auto.commit.enable	C	true, false	true	如果是 true，周期性的提交最后一个消息的偏移量。用于当程序重启时抛弃不用的消息。如果是 false，应用程序需要调用 rd_kafka_offset_store() 保存偏移量 (可选). **提示** 这个属性时能用于简单消费者，high-level KafkaConsumer 会被全局的 enable.auto.commit 属性替代。**提示** 目前没有整合 zookeeper，根据 offset.store.method 的配置 偏移量将写入 broker 或本地文件 file according to offset.store.method.
Type: boolean				
enable.auto.commit	C			参考 auto.commit.enable
auto.commit.interval.ms	C	10 .. 86400000	60000	消费者偏移量提交（写入）到存储的频率，毫秒。
Type: integer				
auto.offset.reset	C	smallest, earliest, beginning, largest, latest, end, error	largest	如果偏移量存储还没有初始化或偏移量超过范围时的处理方式：Action to take when there is no initial offset in offset store or the desired offset is out of range: 'smallest','earliest' - 自动重设偏移量为最小偏移量，'largest','latest' - 自动重设偏移量为最大偏移量，'error' - 通过消费消息触发一个错误，请检查 message->err。
Type: enum value				

属性	C/P	范围	默认值	描述
offset.store.path	C		.	存储偏移量的本地文件路径。如果路径是个目录，在目录下自动创建基于 topic 和 partition 的文件名。
Type: string				
offset.store.sync.interval.ms	C	-1 .. 86400000	-1	偏移量文件 fsync() 的间隔，毫秒。-1 不同步，0 每次写入后立即同步。
Type: integer				
offset.store.method	C	file, broker	broker	偏移量存储方式：'file' - 本地文件存储 (offset.store.path, et.al), 'broker' - 在 broker 上提交存储 (要求 Apache Kafka 0.8.2 或以后版本)。
Type: enum value				
consume.callback.max.messages	C	0 .. 1000000	0	一次 rd_kafka_consume_callback*() 调配的最大消息数 (0 = 无限制)
Type: integer				

3、C/P 含义：C = 生产者, P = 消费者, * = 二者都有

五、参考

Kafka C++客户端库librdkafka详解
https://blog.csdn.net/weixin_43778179/article/details/104839206