Kafka C++客户端库 librdkafka 笔记

一见 2018/4/26

目录

1. 前言

librdkafka 提供的异步的生产接口，异步的消费接口和同步的消息接口，没有同步的生产接口。

2. 缩略语

缩略语	缩略语全称	示例或说明
rd	Rapid Development	rd.h
rk	RdKafka	

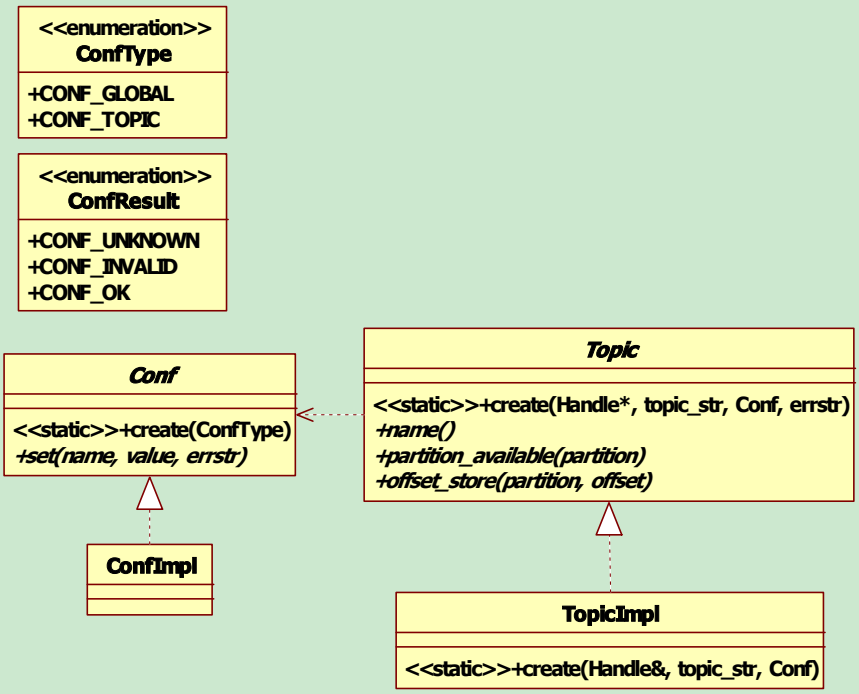
toppar	Topic Partition	struct rd_kafka_**toppar**_t
		{
		};

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）

3. 配置和主题

3.1. 配置和主题结构



3.1.1.Conf

配置接口，配置分两种：全局的和主题的。

3.1.2.ConfImpl

配置的实现。

3.1.3.Topic

主题接口。

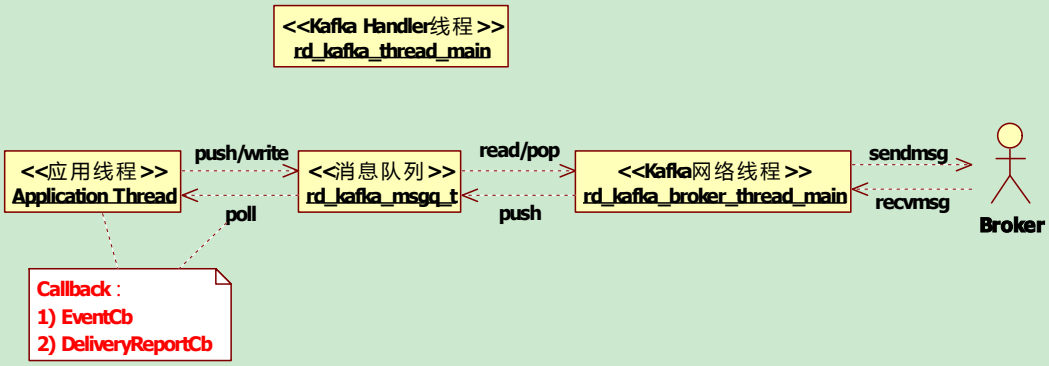
3.1.4.TopicImpl

主题的实现。

4. 线程

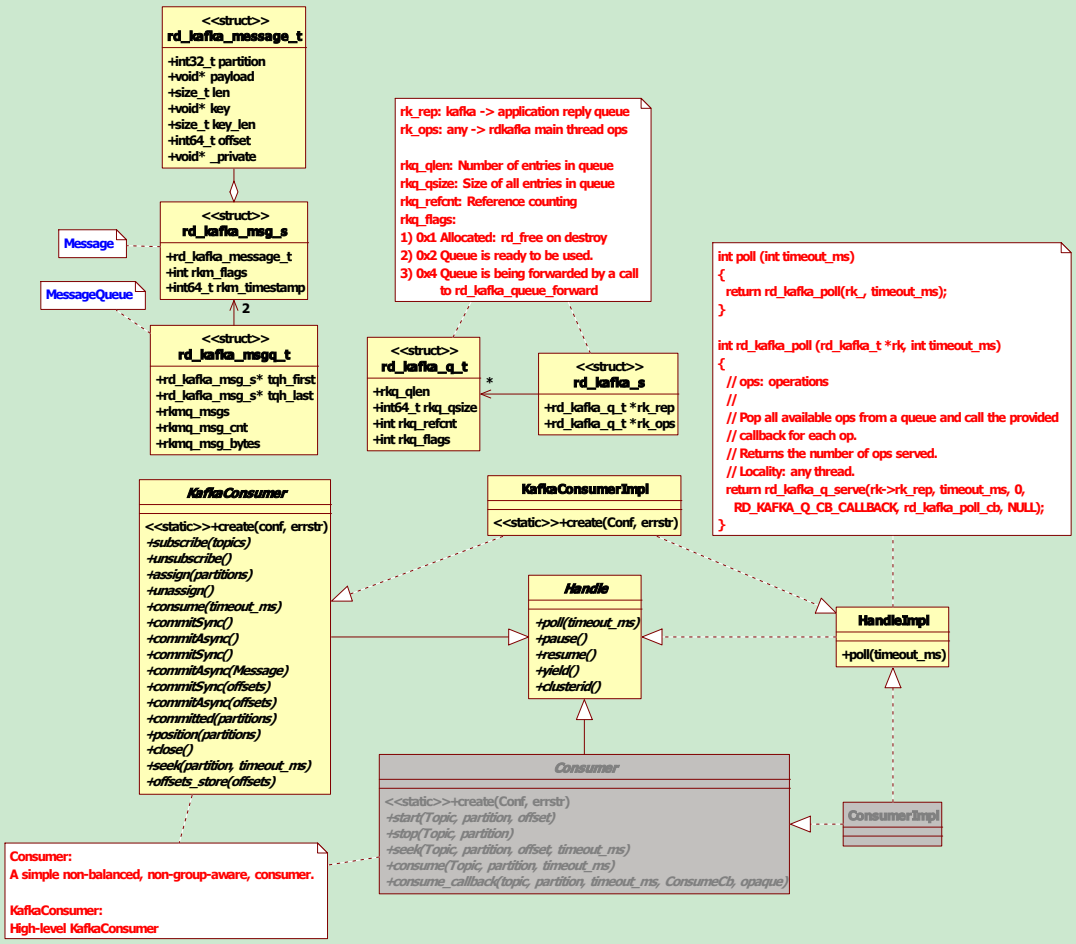
RdKafka 编程涉及到三类线程：

1) 应用线程，业务代码的实现
2) Kafka Broker 线程 rd_kafka_broker_thread_main，负责与 Broker 通讯，多个
3) Kafka Handler 线程 rd_kafka_thread_main，每创建一个 consumer 或 producer 即会创建一个 Handler 线程。



5. 消费者

5.1. 消费者结构



5.1.1. Handle

定义了 poll 等接口，它的实现者为 HandleImpl。

5.1.2. HandleImpl

实现了消费者和生产者均使用的 poll 等，其中 poll 的作用为：

1) 为生产者回调消息发送结果；
2) 为生产者和消费者回调事件。

```
class Handle {
  /**
   * @brief Polls the provided kafka handle for events.
```

```
   *
   * Events will trigger application provided callbacks to be called.
   *
   * The \p timeout_ms argument specifies the maximum amount of time
   * (in milliseconds) that the call will block waiting for events.
   * For non-blocking calls, provide 0 as \p timeout_ms.
   * To wait indefinately for events, provide -1.
   *
   * Events:
   *   - delivery report callbacks (if an RdKafka::DeliveryCb is configured) [producer]
   *   - event callbacks (if an RdKafka::EventCb is configured) [producer & consumer]
   *
   * @remark  An application should make sure to call poll() at regular
   *          intervals to serve any queued callbacks waiting to be called.
   *
   * @warning This method MUST NOT be used with the RdKafka::KafkaConsumer,
   *          use its RdKafka::KafkaConsumer::consume() instead.
   *
   * @returns the number of events served.
   */
  virtual int poll(int timeout_ms) = 0;
};
```

5.1.3.ConsumeCb

只针对消费者的 Callback。

5.1.4.RebalanceCb

只针对消费者的 Callback。

5.1.5.EventCb

消费者和生产者均可设置 EventCb，如：_global_conf->set("**event_cb**", &_event_cb, errmsg);。

```
/**
 * @brief Event callback class
 *
 * Events are a generic interface for propagating errors, statistics, logs, etc
 * from librdkafka to the application.
 *
 * @sa RdKafka::Event
```

```
 */
class RD_EXPORT EventCb {
 public:
  /**
   * @brief Event callback
   *
   * @sa RdKafka::Event
   */
  virtual void event_cb (Event &event) = 0;

  virtual ~EventCb() { }
};

/**
 * @brief Event object class as passed to the EventCb callback.
 */
class RD_EXPORT Event {
 public:
  /** @brief Event type */
  enum Type {
    EVENT_ERROR,     /**< Event is an error condition */
    EVENT_STATS,     /**< Event is a statistics JSON document */
    EVENT_LOG,       /**< Event is a log message */
    EVENT_THROTTLE   /**< Event is a throttle level signaling from the broker */
  };
};
```

5.1.6.Consumer

简单消息者，一般不使用，而是使用 KafkaConsumer。

5.1.7.KafkaConsumer

消费者和生产者均采用多重继承方式，其中 KafkaConsumer 为消费者接口，KafkaConsumerImpl 为消费者实现。

5.1.8.KafkaConsumerImpl

KafkaConsumerImpl 为消费者实现。

5.1.9.rd_kafka_message_t

消息结构。

5.1.10. rd_kafka_msg_s

消息结构，但消息数据实际存储在 rd_kafka_message_t，结构大致如下：

```
struct rd_kafka_msg_s
{
  rd_kafka_message_t rkm_rkmessage;
  struct
  {
    rd_kafka_msg_s* tqe_next;
    rd_kafka_msg_s** tqe_prev;
    int64_t rkm_timestamp;
    rd_kafka_timestamp_type_t rkm_tstype;
  }rkm_link;
};
```

5.1.11. rd_kafka_msgq_t

存储消息的消息队列，生产者生产的消息并不直接 socket 发送到 brokers，而是放入了这个队列，结构大致如下：

```
struct rd_kafka_msgq_t
{
  struct
  {
    rd_kafka_msg_s* tqh_first; // 队首
    rd_kafka_msg_s* tqh_last;  // 队尾
  };

  // 消息个数
  rd_atomic32_t rkmq_msg_cnt;
  // 所有消息加起来的字节数
  rd_atomic64_t rkmq_msg_bytes;
};
```

5.1.12. rd_kafka_toppar_t

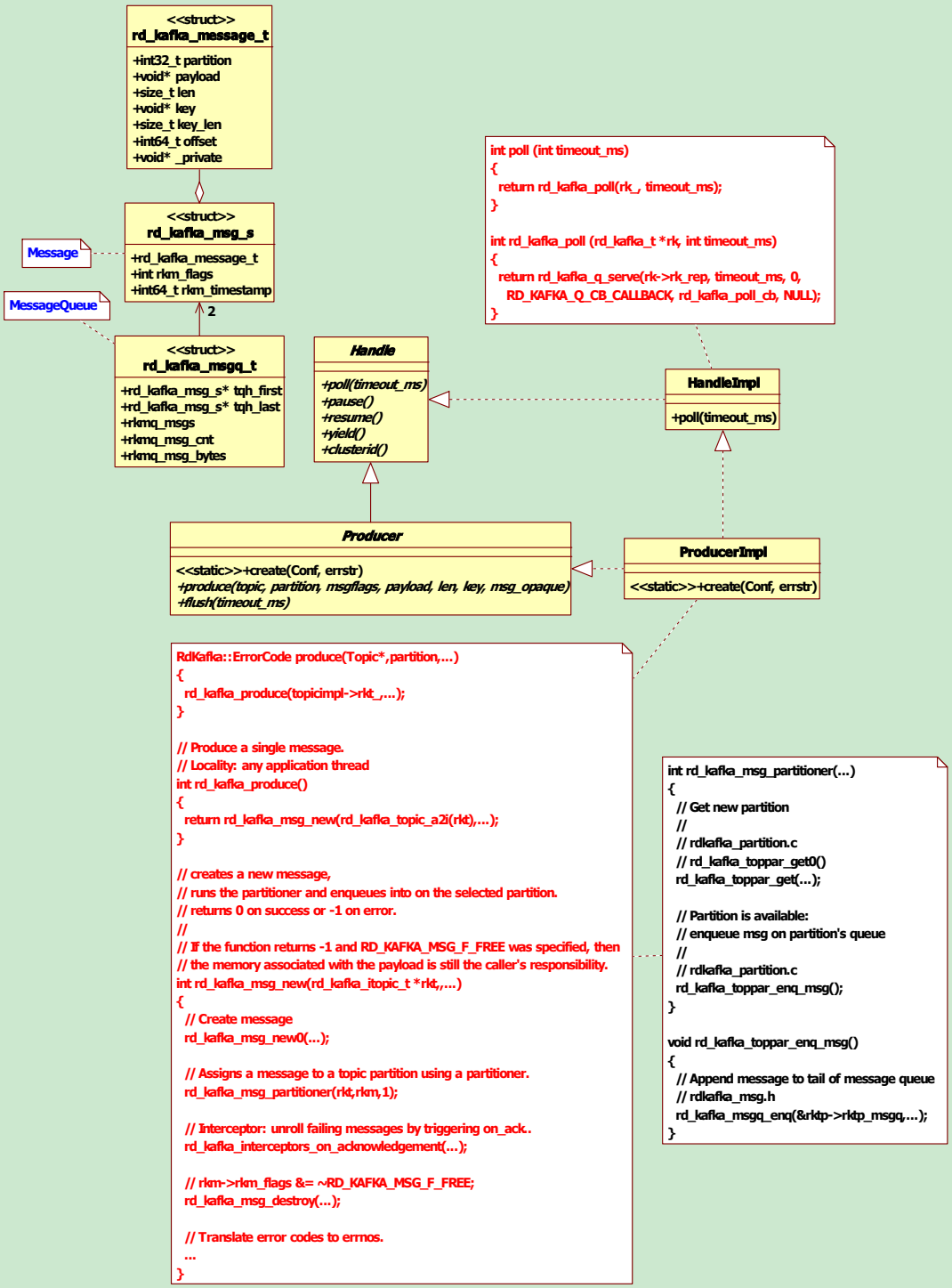
Topic-Partition 队列，很复杂的一个结构，部分内容如下：

```
// Topic + Partition combination
```

```
typedef struct rd_kafka_toppar_s
{
  struct
  {
    rd_kafka_toppar_s* tqe_next;
    rd_kafka_toppar_s** tqe_prev;
  }rktp_rklink;

  struct
  {
    rd_kafka_toppar_s* tqe_next;
    rd_kafka_toppar_s** tqe_prev;
  }rktp_rkblink;

  struct
  {
    rd_kafka_toppar_s* cqe_next;
    rd_kafka_toppar_s* cqe_prev;
  }rktp_fetchlink;

  struct
  {
    rd_kafka_toppar_s* tqe_next;
    rd_kafka_toppar_s** tqe_prev;
  }rktp_rktlink;

  struct
  {
    rd_kafka_toppar_s* tqe_next;
    rd_kafka_toppar_s** tqe_prev;
  }rktp_cgrplink;

  rd_kafka_itopic_t* rktp_rkt;
  int32_t rktp_partition;
  int32_t rktp_leader_id;
  rd_kafka_broker_t* rktp_leader;
  rd_kafka_broker_t* rktp_next_leader;
  rd_refcnt_t rktp_refcnt;
  rd_kafka_msgq_t rktp_msgq; // application->rdkafka queue
}rd_kafka_toppar_t;
```

6. 生产者

6.1. 生产者结构



6.1.1.DeliveryReportCb

消息已经成功递送到 Broker 时回调，只针对生产者有效。

6.1.2.PartitionerCb

计算分区号回调函数，只针对生产者有效。

6.1.3.Producer

Producer 为生产者接口，它的实现者为 ProducerImpl。

6.1.4.ProduceImpl

ProducerImpl 为生产者的实现。

6.2. 生产者启动过程 1

启动时会创建两组线程：一组 Broker 线程（rd_kafka_**broker**_thread_main，多个），实为与 Broker 间的网络 IO 线程；一组 Handler 线程（rd_kafka_thread_**main**，单个），每调用一次 RdKafka::Producer::create 或 rd_kafka_new 即创建一 Handler 线程。

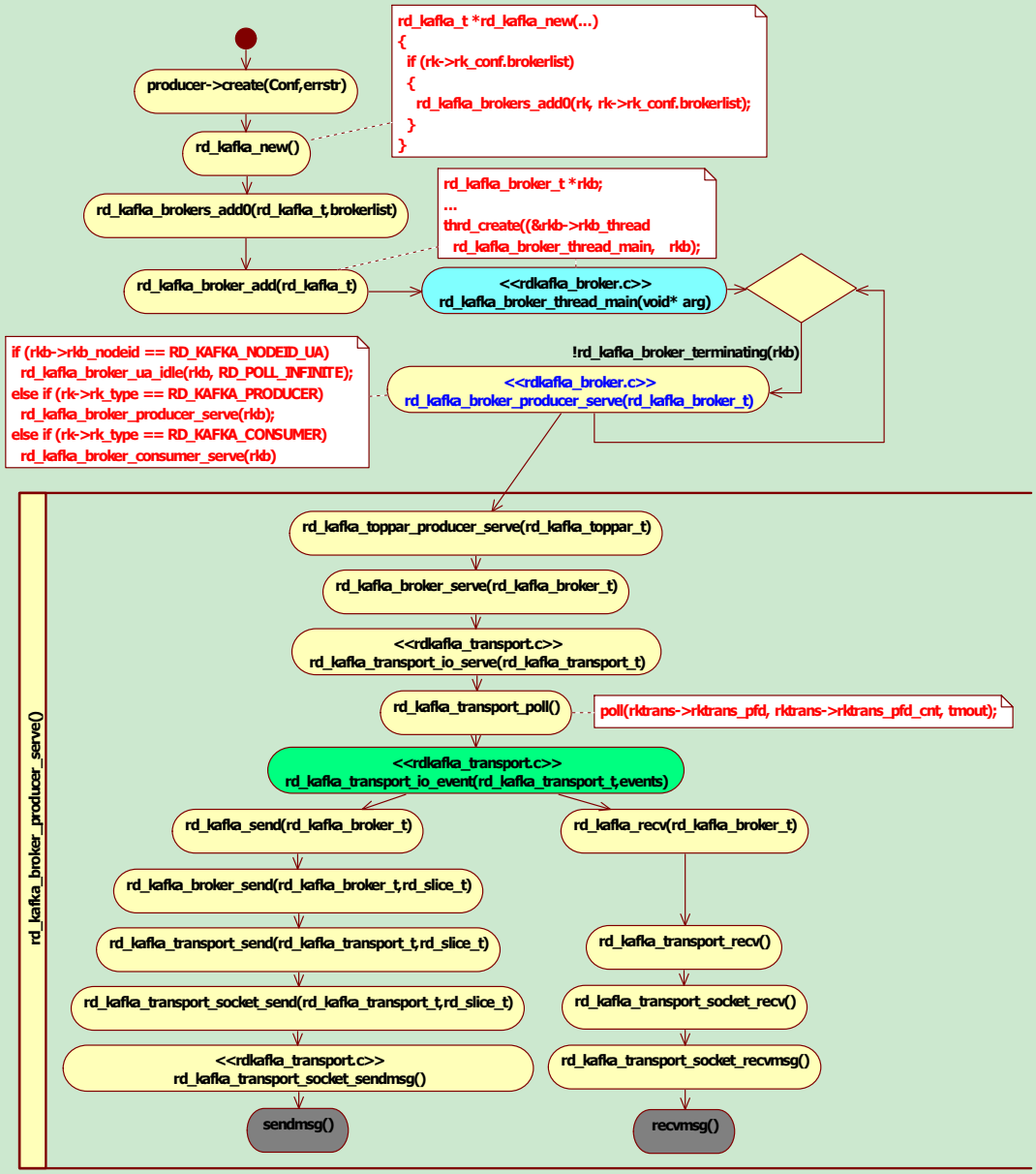
Handler 线程调用栈：

```
(gdb) t 17
[Switching to thread 17 (Thread 0x7ff7059d3700 (LWP 16765))]
#0  0x00007ff7091e6cf2 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
(gdb) bt
#0  0x00007ff7091e6cf2 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
#1  0x00000000005b4d2f in cnd_timedwait_ms (cnd=0x1517748, mtx=0x1517720, timeout_ms=898) at tinycthread.c:501
#2  0x0000000000580e16 in rd_kafka_q_serve (rkq=0x1517720, timeout_ms=898, max_cnt=0, cb_type=RD_KAFKA_Q_CB_CALLBACK, callback=0x0, opaque=0x0) at rdkafka_queue.c:440
#3  0x000000000054ee9b in rd_kafka_thread_main (arg=0x1516df0) at rdkafka.c:1227
#4  0x00000000005b4e0f in _thrd_wrapper_function (aArg=0x15179d0) at tinycthread.c:624
#5  0x00007ff7091e2e25 in start_thread () from /lib64/libpthread.so.0
#6  0x00007ff7082d135d in clone () from /lib64/libc.so.6
```

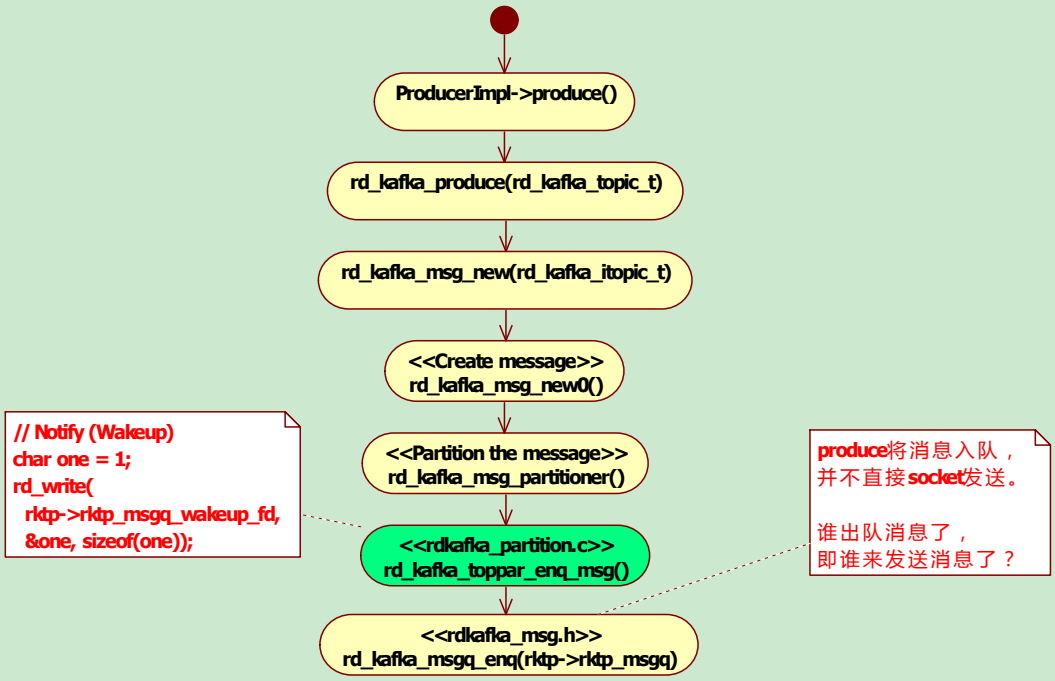
6.3. 生产者启动过程 2

创建网络 IO 线程，消费者启动过程类似，只是一个调用 rd_kafka_broker_producer_serve(rkb)，另一个调用 rd_kafka_broker_consumer_serve(rkb)。

IO 线程负责消息的收和发，发送底层调用的是 sendmsg，收调用的是 recvmsg（但 MSVC 平台调用 send 和 recv）。



6.4. 生产者生产过程



生产者生产的消息并不直接 socket 发送到 brokers，而是放入队列 **rd_kafka_msgq_t** 中。Broker 线程（rd_kafka_broker_thread_main）消费这个队列。

Broker 线程同时监控与 Broker 间的网络连接，又要监控队列中是否有数据，如何实现的？这个队列和管道绑定在一起的，绑定的是管道写端（rktp->rktp_msgq_wakeup_fd = rkb->rkb_toppar_wakeup_fd; rkb->rkb_toppar_wakeup_fd=rkb->rkb_wakeup_fd[1]）。

这样 Broker 线程即可同时监听网络数据和管道数据。

```
// int rd_kafka_msg_partitioner(rd_kafka_itopic_t *rkt, rd_kafka_msg_t *rkm,int do_lock)
(gdb) p *rkm
$7 = {rkm_rkmessage = {err = RD_KAFKA_RESP_ERR_NO_ERROR, rkt = 0x1590c10, partition = 1, payload
= 0x7f48c4001260, len = 203, key = 0x7f48c400132b, key_len = 14, offset = 0,
    _private = 0x0}, rkm_link = {tqe_next = 0x5b5d47554245445b, tqe_prev = 0x6361667265746e69},
rkm_flags = 196610, rkm_timestamp = 1524829399009,
  rkm_tstype = RD_KAFKA_TIMESTAMP_CREATE_TIME, rkm_u = {producer = {ts_timeout = 16074575505526,
ts_enq = 16074275505526}}}
(gdb) p rkm->rkm_rkmessage
$8 = {err = RD_KAFKA_RESP_ERR_NO_ERROR, rkt = 0x1590c10, partition = 1, payload = 0x7f48c4001260,
len = 203, key = 0x7f48c400132b, key_len = 14, offset = 0, _private = 0x0}
(gdb) p rkm->rkm_rkmessage->payload
$9 = (void *) 0x7f48c4001260
(gdb) p (char*)rkm->rkm_rkmessage->payload
$10                                    =                                    0x7f48c4001260
"{\"p\":\"f\",\"o\":1,\"d\":\"m\",\"d\":\"m\",\"i\":\"f2\",\"ip\":\"127.0.0.1\",\"pt\":2018,
\"sc\":0,\"fc\":1,\"tc\":0,\"acc\":395,\"mcc\":395,\"cd\":\"test\",\"cmd\":\"tester\",\"cf\"
```

:\"main\",\"cp\":\"1.49.16.9\"...

7. poll 过程

poll 的作用是触发回调，生产者即使不调用 poll，消息也会发送出去，但是如果不通过 poll 触发回调，则不能确定消息发送状态（成功或失败等）。

消费队列 rd_kafka_t->rk_rep，rk_rep 为响应队列，类型为 rd_kafka_q_t 或 rd_kafka_q_s：