

原创

pyxllq

于 2018-06-07 15:08:03 发布

7507

收藏 10

版权

分类专栏: 编程

编程

专栏收录该内容

0 订阅 11 篇文章 订阅专栏

1.提供初始化以及收发消息接口，rd_kafka_msg.h

```
1 | #ifndef RDKAFKA_MSG_H_
2 | #define RDKAFKA_MSG_H_
3 |
4 | #include "rdkafka.h"
5 |
6 | int kafka_init(char mode, char* topic, int partition, char* brokers,
7 |               rd_kafka_topic_conf_t **topic_conf, rd_kafka_topic_t **rkt);
8 | int kafka_send_msg(rd_kafka_topic_t *rkt, rd_kafka_topic_conf_t *topic_conf, char* data, int len);
9 | void* kafka_receive_msg(void *arg);
10| #endif
```

2.kafka库头文件rdkafka.h，有点长

```
1 | /*
2 |  * Librdkafka - Apache Kafka C Library
3 |  *
4 |  * Copyright (c) 2012-2013 Magnus Edenhill
5 |  * All rights reserved.
6 |  *
7 |  * Redistribution and use in source and binary forms, with or without
8 |  * modification, are permitted provided that the following conditions are met:
9 |  *
10|  * 1. Redistributions of source code must retain the above copyright notice,
11|  *   this list of conditions and the following disclaimer.
12|  * 2. Redistributions in binary form must reproduce the above copyright notice,
13|  *   this list of conditions and the following disclaimer in the documentation
14|  *   and/or other materials provided with the distribution.
15|  *
16|  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
17|  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
18|  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
19|  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
20|  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
21|  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
22|  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
23|  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
24|  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
25|  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
26|  * POSSIBILITY OF SUCH DAMAGE.
27| */
28|
29| /**
30|  * @file rdkafka.h
31|  * @brief Apache Kafka C/C++ consumer and producer client library.
32|  *
33|  * rdkafka.h contains the public API for Librdkafka.
34|  * The API is documented in this file as comments prefixing the function, type,
```

分类专栏

	算法	1篇
	MySQL	2篇
	DPDK	1篇
	Vue开发	1篇
	编程	11篇
	python	2篇
	mysql常用命令	2篇
	归档	5篇
	分布式	2篇
	linux	7篇

```
35  * enum, define, etc.
36  *
37  * @sa For the C++ interface see rdkafkacpp.h
38  *
39  * @tableofcontents
40  */
41
42
43  /* @cond NO_DOC */
44  #pragma once
45
46  #include <stdio.h>
47  #include <inttypes.h>
48  #include <sys/types.h>
49
50  #ifdef __cplusplus
51  extern "C" {
52  #if 0
53  } /* Restore indent */
54  #endif
55  #endif
56
57  #ifdef _MSC_VER
58  #include <basetsd.h>
59  #ifndef WIN32_MEAN_AND_LEAN
60  #define WIN32_MEAN_AND_LEAN
61  #endif
62  #include <Winsock2.h> /* for sockaddr, .. */
63  typedef SSIZE_T ssize_t;
64  #define RD_UNUSED
65  #define RD_INLINE __inline
66  #define RD_DEPRECATED __declspec(deprecated)
67  #undef RD_EXPORT
68  #ifdef LIBRDKAFKA_STATICLIB
69  #define RD_EXPORT
70  #else
71  #ifdef LIBRDKAFKA_EXPORTS
72  #define RD_EXPORT __declspec(dllexport)
73  #else
74  #define RD_EXPORT __declspec(dllimport)
75  #endif
76  #ifndef LIBRDKAFKA_TYPECHECKS
77  #define LIBRDKAFKA_TYPECHECKS 0
78  #endif
79  #endif
80
81  #else
82  #include <sys/socket.h> /* for sockaddr, .. */
83
84  #define RD_UNUSED __attribute__((unused))
85  #define RD_INLINE inline
86  #define RD_EXPORT
87  #define RD_DEPRECATED __attribute__((deprecated))
88
89  #ifndef LIBRDKAFKA_TYPECHECKS
90  #define LIBRDKAFKA_TYPECHECKS 1
91  #endif
92  #endif
93
94
95  /**
```

```

96  * @brief Type-checking macros
97  * Compile-time checking that \p ARG is of type \p TYPE.
98  * @returns \p RET
99  */
100 #if LIBRDKAFKA_TYPECHECKS
101 #define _LRK_TYPECHECK(RET,TYPE,ARG) \
102     ({ if (0) { TYPE __t RD_UNUSED = (ARG); } RET; })
103
104 #define _LRK_TYPECHECK2(RET,TYPE,ARG,TYPE2,ARG2) \
105     ({ \
106         if (0) { \
107             TYPE __t RD_UNUSED = (ARG); \
108             TYPE2 __t2 RD_UNUSED = (ARG2); \
109         } \
110         RET; })
111 #else
112 #define _LRK_TYPECHECK(RET,TYPE,ARG) (RET)
113 #define _LRK_TYPECHECK2(RET,TYPE,ARG,TYPE2,ARG2) (RET)
114 #endif
115
116 /* @endcond */
117
118 /**
119  * @name Librdkafka version
120  * @{
121  *
122  *
123  *
124  */
125
126 /**
127  * @brief Librdkafka version
128  *
129  * Interpreted as hex \c MM.mm.rr.xx:
130  * - MM = Major
131  * - mm = minor
132  * - rr = revision
133  * - xx = pre-release id (0xff is the final release)
134  *
135  * E.g.: \c 0x000801ff = 0.8.1
136  *
137  * @remark This value should only be used during compile time,
138  *         for runtime checks of version use rd_kafka_version()
139  */
140 #define RD_KAFKA_VERSION 0x000b00ff
141
142 /**
143  * @brief Returns the Librdkafka version as integer.
144  *
145  * @returns Version integer.
146  *
147  * @sa See RD_KAFKA_VERSION for how to parse the integer format.
148  * @sa Use rd_kafka_version_str() to retrieve the version as a string.
149  */
150 RD_EXPORT
151 int rd_kafka_version(void);
152
153 /**
154  * @brief Returns the Librdkafka version as string.
155  *
156  * @returns Version string

```

```

157 */
158 RD_EXPORT
159 const char *rd_kafka_version_str (void);
160
161 /**@}*/
162
163
164 /**
165  * @name Constants, errors, types
166  * @{
167  *
168  *
169  */
170
171
172 /**
173  * @enum rd_kafka_type_t
174  *
175  * @brief rd_kafka_t handle type.
176  *
177  * @sa rd_kafka_new()
178  */
179 typedef enum rd_kafka_type_t {
180     RD_KAFKA_PRODUCER, /**< Producer client */
181     RD_KAFKA_CONSUMER /**< Consumer client */
182 } rd_kafka_type_t;
183
184
185 /**
186  * @enum Timestamp types
187  *
188  * @sa rd_kafka_message_timestamp()
189  */
190 typedef enum rd_kafka_timestamp_type_t {
191     RD_KAFKA_TIMESTAMP_NOT_AVAILABLE, /**< Timestamp not available */
192     RD_KAFKA_TIMESTAMP_CREATE_TIME, /**< Message creation time */
193     RD_KAFKA_TIMESTAMP_LOG_APPEND_TIME /**< Log append time */
194 } rd_kafka_timestamp_type_t;
195
196
197
198 /**
199  * @brief Retrieve supported debug contexts for use with the \c "debug"
200  *        configuration property. (runtime)
201  *
202  * @returns Comma-separated list of available debugging contexts.
203  */
204 RD_EXPORT
205 const char *rd_kafka_get_debug_contexts(void);
206
207 /**
208  * @brief Supported debug contexts. (compile time)
209  *
210  * @deprecated This compile time value may be outdated at runtime due to
211  *        linking another version of the library.
212  *        Use rd_kafka_get_debug_contexts() instead.
213  */
214 #define RD_KAFKA_DEBUG_CONTEXTS \
215     "all,generic,broker,topic,metadata,queue,msg,protocol,cgrp,security,fetch,feature"
216
217

```

```

218 /* @cond NO_DOC */
219 /* Private types to provide ABI compatibility */
220 typedef struct rd_kafka_s rd_kafka_t;
221 typedef struct rd_kafka_topic_s rd_kafka_topic_t;
222 typedef struct rd_kafka_conf_s rd_kafka_conf_t;
223 typedef struct rd_kafka_topic_conf_s rd_kafka_topic_conf_t;
224 typedef struct rd_kafka_queue_s rd_kafka_queue_t;
225 /* @endcond */
226
227
228 /**
229  * @enum rd_kafka_resp_err_t
230  * @brief Error codes.
231  *
232  * The negative error codes delimited by two underscores
233  * (\c RD_KAFKA_RESP_ERR__..) denotes errors internal to librdkafka and are
234  * displayed as \c "Local: <error string..>", while the error codes
235  * delimited by a single underscore (\c RD_KAFKA_RESP_ERR..) denote broker
236  * errors and are displayed as \c "Broker: <error string..>".
237  *
238  * @sa Use rd_kafka_err2str() to translate an error code a human readable string
239  */
240 typedef enum {
241     /* Internal errors to rdkafka: */
242     /** Begin internal error codes */
243     RD_KAFKA_RESP_ERR__BEGIN = -200,
244     /** Received message is incorrect */
245     RD_KAFKA_RESP_ERR__BAD_MSG = -199,
246     /** Bad/unknown compression */
247     RD_KAFKA_RESP_ERR__BAD_COMPRESSION = -198,
248     /** Broker is going away */
249     RD_KAFKA_RESP_ERR__DESTROY = -197,
250     /** Generic failure */
251     RD_KAFKA_RESP_ERR__FAIL = -196,
252     /** Broker transport failure */
253     RD_KAFKA_RESP_ERR__TRANSPORT = -195,
254     /** Critical system resource */
255     RD_KAFKA_RESP_ERR__CRIT_SYS_RESOURCE = -194,
256     /** Failed to resolve broker */
257     RD_KAFKA_RESP_ERR__RESOLVE = -193,
258     /** Produced message timed out*/
259     RD_KAFKA_RESP_ERR__MSG_TIMED_OUT = -192,
260     /** Reached the end of the topic+partition queue on
261      * the broker. Not really an error. */
262     RD_KAFKA_RESP_ERR__PARTITION_EOF = -191,
263     /** Permanent: Partition does not exist in cluster. */
264     RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION = -190,
265     /** File or filesystem error */
266     RD_KAFKA_RESP_ERR__FS = -189,
267     /** Permanent: Topic does not exist in cluster. */
268     RD_KAFKA_RESP_ERR__UNKNOWN_TOPIC = -188,
269     /** All broker connections are down. */
270     RD_KAFKA_RESP_ERR__ALL_BROKERS_DOWN = -187,
271     /** Invalid argument, or invalid configuration */
272     RD_KAFKA_RESP_ERR__INVALID_ARG = -186,
273     /** Operation timed out */
274     RD_KAFKA_RESP_ERR__TIMED_OUT = -185,
275     /** Queue is full */
276     RD_KAFKA_RESP_ERR__QUEUE_FULL = -184,
277     /** ISR count < required.acks */
278     RD_KAFKA_RESP_ERR__ISR_INSUFF = -183,

```

```
279      /** Broker node update */
280      RD_KAFKA_RESP_ERR__NODE_UPDATE = -182,
281      /** SSL error */
282      RD_KAFKA_RESP_ERR__SSL = -181,
283      /** Waiting for coordinator to become available. */
284      RD_KAFKA_RESP_ERR__WAIT_COORD = -180,
285      /** Unknown client group */
286      RD_KAFKA_RESP_ERR__UNKNOWN_GROUP = -179,
287      /** Operation in progress */
288      RD_KAFKA_RESP_ERR__IN_PROGRESS = -178,
289      /** Previous operation in progress, wait for it to finish. */
290      RD_KAFKA_RESP_ERR__PREV_IN_PROGRESS = -177,
291      /** This operation would interfere with an existing subscription */
292      RD_KAFKA_RESP_ERR__EXISTING_SUBSCRIPTION = -176,
293      /** Assigned partitions (rebalance_cb) */
294      RD_KAFKA_RESP_ERR__ASSIGN_PARTITIONS = -175,
295      /** Revoked partitions (rebalance_cb) */
296      RD_KAFKA_RESP_ERR__REVOKE_PARTITIONS = -174,
297      /** Conflicting use */
298      RD_KAFKA_RESP_ERR__CONFLICT = -173,
299      /** Wrong state */
300      RD_KAFKA_RESP_ERR__STATE = -172,
301      /** Unknown protocol */
302      RD_KAFKA_RESP_ERR__UNKNOWN_PROTOCOL = -171,
303      /** Not implemented */
304      RD_KAFKA_RESP_ERR__NOT_IMPLEMENTED = -170,
305      /** Authentication failure*/
306      RD_KAFKA_RESP_ERR__AUTHENTICATION = -169,
307      /** No stored offset */
308      RD_KAFKA_RESP_ERR__NO_OFFSET = -168,
309      /** Outdated */
310      RD_KAFKA_RESP_ERR__OUTDATED = -167,
311      /** Timed out in queue */
312      RD_KAFKA_RESP_ERR__TIMED_OUT_QUEUE = -166,
313      /** Feature not supported by broker */
314      RD_KAFKA_RESP_ERR__UNSUPPORTED_FEATURE = -165,
315      /** Awaiting cache update */
316      RD_KAFKA_RESP_ERR__WAIT_CACHE = -164,
317      /** Operation interrupted (e.g., due to yield)) */
318      RD_KAFKA_RESP_ERR__INTR = -163,
319      /** Key serialization error */
320      RD_KAFKA_RESP_ERR__KEY_SERIALIZATION = -162,
321      /** Value serialization error */
322      RD_KAFKA_RESP_ERR__VALUE_SERIALIZATION = -161,
323      /** Key deserialization error */
324      RD_KAFKA_RESP_ERR__KEY_DESERIALIZATION = -160,
325      /** Value deserialization error */
326      RD_KAFKA_RESP_ERR__VALUE_DESERIALIZATION = -159,
327
328      /** End internal error codes */
329      RD_KAFKA_RESP_ERR__END = -100,
330
331      /* Kafka broker errors: */
332      /** Unknown broker error */
333      RD_KAFKA_RESP_ERR__UNKNOWN = -1,
334      /** Success */
335      RD_KAFKA_RESP_ERR_NO_ERROR = 0,
336      /** Offset out of range */
337      RD_KAFKA_RESP_ERR_OFFSET_OUT_OF_RANGE = 1,
338      /** Invalid message */
339      RD_KAFKA_RESP_ERR_INVALID_MSG = 2,
```

```
340     /** Unknown topic or partition */
341     RD_KAFKA_RESP_ERR_UNKNOWN_TOPIC_OR_PART = 3,
342     /** Invalid message size */
343     RD_KAFKA_RESP_ERR_INVALID_MSG_SIZE = 4,
344     /** Leader not available */
345     RD_KAFKA_RESP_ERR_LEADER_NOT_AVAILABLE = 5,
346     /** Not leader for partition */
347     RD_KAFKA_RESP_ERR_NOT_LEADER_FOR_PARTITION = 6,
348     /** Request timed out */
349     RD_KAFKA_RESP_ERR_REQUEST_TIMED_OUT = 7,
350     /** Broker not available */
351     RD_KAFKA_RESP_ERR_BROKER_NOT_AVAILABLE = 8,
352     /** Replica not available */
353     RD_KAFKA_RESP_ERR_REPLICA_NOT_AVAILABLE = 9,
354     /** Message size too large */
355     RD_KAFKA_RESP_ERR_MSG_SIZE_TOO_LARGE = 10,
356     /** StaleControllerEpochCode */
357     RD_KAFKA_RESP_ERR_STALE_CTRL_EPOCH = 11,
358     /** Offset metadata string too large */
359     RD_KAFKA_RESP_ERR_OFFSET_METADATA_TOO_LARGE = 12,
360     /** Broker disconnected before response received */
361     RD_KAFKA_RESP_ERR_NETWORK_EXCEPTION = 13,
362     /** Group coordinator load in progress */
363     RD_KAFKA_RESP_ERR_GROUP_LOAD_IN_PROGRESS = 14,
364     /** Group coordinator not available */
365     RD_KAFKA_RESP_ERR_GROUP_COORDINATOR_NOT_AVAILABLE = 15,
366     /** Not coordinator for group */
367     RD_KAFKA_RESP_ERR_NOT_COORDINATOR_FOR_GROUP = 16,
368     /** Invalid topic */
369     RD_KAFKA_RESP_ERR_TOPIC_EXCEPTION = 17,
370     /** Message batch larger than configured server segment size */
371     RD_KAFKA_RESP_ERR_RECORD_LIST_TOO_LARGE = 18,
372     /** Not enough in-sync replicas */
373     RD_KAFKA_RESP_ERR_NOT_ENOUGH_REPLICAS = 19,
374     /** Message(s) written to insufficient number of in-sync replicas */
375     RD_KAFKA_RESP_ERR_NOT_ENOUGH_REPLICAS_AFTER_APPEND = 20,
376     /** Invalid required acks value */
377     RD_KAFKA_RESP_ERR_INVALID_REQUIRED_ACKS = 21,
378     /** Specified group generation id is not valid */
379     RD_KAFKA_RESP_ERR_ILLEGAL_GENERATION = 22,
380     /** Inconsistent group protocol */
381     RD_KAFKA_RESP_ERR_INCONSISTENT_GROUP_PROTOCOL = 23,
382     /** Invalid group.id */
383     RD_KAFKA_RESP_ERR_INVALID_GROUP_ID = 24,
384     /** Unknown member */
385     RD_KAFKA_RESP_ERR_UNKNOWN_MEMBER_ID = 25,
386     /** Invalid session timeout */
387     RD_KAFKA_RESP_ERR_INVALID_SESSION_TIMEOUT = 26,
388     /** Group rebalance in progress */
389     RD_KAFKA_RESP_ERR_REBALANCE_IN_PROGRESS = 27,
390     /** Commit offset data size is not valid */
391     RD_KAFKA_RESP_ERR_INVALID_COMMIT_OFFSET_SIZE = 28,
392     /** Topic authorization failed */
393     RD_KAFKA_RESP_ERR_TOPIC_AUTHORIZATION_FAILED = 29,
394     /** Group authorization failed */
395     RD_KAFKA_RESP_ERR_GROUP_AUTHORIZATION_FAILED = 30,
396     /** Cluster authorization failed */
397     RD_KAFKA_RESP_ERR_CLUSTER_AUTHORIZATION_FAILED = 31,
398     /** Invalid timestamp */
399     RD_KAFKA_RESP_ERR_INVALID_TIMESTAMP = 32,
400     /** Unsupported SASL mechanism */
```

```

401 RD_KAFKA_RESP_ERR_UNSUPPORTED_SASL_MECHANISM = 33,
402     /** Illegal SASL state */
403 RD_KAFKA_RESP_ERR_ILLEGAL_SASL_STATE = 34,
404     /** Unusported version */
405 RD_KAFKA_RESP_ERR_UNSUPPORTED_VERSION = 35,
406     /** Topic already exists */
407 RD_KAFKA_RESP_ERR_TOPIC_ALREADY_EXISTS = 36,
408     /** Invalid number of partitions */
409 RD_KAFKA_RESP_ERR_INVALID_PARTITIONS = 37,
410     /** Invalid replication factor */
411 RD_KAFKA_RESP_ERR_INVALID_REPLICATION_FACTOR = 38,
412     /** Invalid replica assignment */
413 RD_KAFKA_RESP_ERR_INVALID_REPLICA_ASSIGNMENT = 39,
414     /** Invalid config */
415 RD_KAFKA_RESP_ERR_INVALID_CONFIG = 40,
416     /** Not controller for cluster */
417 RD_KAFKA_RESP_ERR_NOT_CONTROLLER = 41,
418     /** Invalid request */
419 RD_KAFKA_RESP_ERR_INVALID_REQUEST = 42,
420     /** Message format on broker does not support request */
421 RD_KAFKA_RESP_ERR_UNSUPPORTED_FOR_MESSAGE_FORMAT = 43,
422     /** Isolation policy volution */
423     RD_KAFKA_RESP_ERR_POLICY_VIOLATION = 44,
424     /** Broker received an out of order sequence number */
425     RD_KAFKA_RESP_ERR_OUT_OF_ORDER_SEQUENCE_NUMBER = 45,
426     /** Broker received a duplicate sequence number */
427     RD_KAFKA_RESP_ERR_DUPLICATE_SEQUENCE_NUMBER = 46,
428     /** Producer attempted an operation with an old epoch */
429     RD_KAFKA_RESP_ERR_INVALID_PRODUCER_EPOCH = 47,
430     /** Producer attempted a transactional operation in an invalid state */
431     RD_KAFKA_RESP_ERR_INVALID_TXN_STATE = 48,
432     /** Producer attempted to use a producer id which is not
433      * currently assigned to its transactional id */
434     RD_KAFKA_RESP_ERR_INVALID_PRODUCER_ID_MAPPING = 49,
435     /** Transaction timeout is larger than the maximum
436      * value allowed by the broker's max.transaction.timeout.ms */
437     RD_KAFKA_RESP_ERR_INVALID_TRANSACTION_TIMEOUT = 50,
438     /** Producer attempted to update a transaction while another
439      * concurrent operation on the same transaction was ongoing */
440     RD_KAFKA_RESP_ERR_CONCURRENT_TRANSACTIONS = 51,
441     /** Indicates that the transaction coordinator sending a
442      * WriteTxnMarker is no longer the current coordinator for a
443      * given producer */
444     RD_KAFKA_RESP_ERR_TRANSACTION_COORDINATOR_FENCED = 52,
445     /** Transactional Id authorization failed */
446     RD_KAFKA_RESP_ERR_TRANSACTIONAL_ID_AUTHORIZATION_FAILED = 53,
447     /** Security features are disabled */
448     RD_KAFKA_RESP_ERR_SECURITY_DISABLED = 54,
449     /** Operation not attempted */
450     RD_KAFKA_RESP_ERR_OPERATION_NOT_ATTEMPTED = 55,
451
452     RD_KAFKA_RESP_ERR_END_ALL,
453 } rd_kafka_resp_err_t;
454
455
456 /**
457  * @brief Error code value, name and description.
458  * Typically for use with language bindings to automatically expose
459  * the full set of Librdkafka error codes.
460  */
461 struct rd_kafka_err_desc {

```



```

462     rd_kafka_resp_err_t code;/**< Error code */
463     const char *name;      /**< Error name, same as code enum sans prefix */
464     const char *desc;      /**< Human readable error description. */
465 };
466
467
468 /**
469  * @brief Returns the full list of error codes.
470  */
471 RD_EXPORT
472 void rd_kafka_get_err_descs (const struct rd_kafka_err_desc **errdescs,
473                             size_t *cntp);
474
475
476
477
478 /**
479  * @brief Returns a human readable representation of a kafka error.
480  *
481  * @param err Error code to translate
482  */
483 RD_EXPORT
484 const char *rd_kafka_err2str (rd_kafka_resp_err_t err);
485
486
487
488 /**
489  * @brief Returns the error code name (enum name).
490  *
491  * @param err Error code to translate
492  */
493 RD_EXPORT
494 const char *rd_kafka_err2name (rd_kafka_resp_err_t err);
495
496
497 /**
498  * @brief Returns the last error code generated by a Legacy API call
499  *        in the current thread.
500  *
501  * The legacy APIs are the ones using errno to propagate error value, namely:
502  * - rd_kafka_topic_new()
503  * - rd_kafka_consume_start()
504  * - rd_kafka_consume_stop()
505  * - rd_kafka_consume()
506  * - rd_kafka_consume_batch()
507  * - rd_kafka_consume_callback()
508  * - rd_kafka_consume_queue()
509  * - rd_kafka_produce()
510  *
511  * The main use for this function is to avoid converting system \p errno
512  * values to rd_kafka_resp_err_t codes for Legacy APIs.
513  *
514  * @remark The last error is stored per-thread, if multiple rd_kafka_t handles
515  *          are used in the same application thread the developer needs to
516  *          make sure rd_kafka_last_error() is called immediately after
517  *          a failed API call.
518  *
519  * @remark errno propagation from Librdkafka is not safe on Windows
520  *          and should not be used, use rd_kafka_last_error() instead.
521  */
522 RD_EXPORT

```

```

523 rd_kafka_resp_err_t rd_kafka_last_error (void);
524
525
526 /**
527  * @brief Converts the system errno value \p errnox to a rd_kafka_resp_err_t
528  *       error code upon failure from the following functions:
529  *       - rd_kafka_topic_new()
530  *       - rd_kafka_consume_start()
531  *       - rd_kafka_consume_stop()
532  *       - rd_kafka_consume()
533  *       - rd_kafka_consume_batch()
534  *       - rd_kafka_consume_callback()
535  *       - rd_kafka_consume_queue()
536  *       - rd_kafka_produce()
537  *
538  * @param errnox System errno value to convert
539  *
540  * @returns Appropriate error code for \p errnox
541  *
542  * @remark A better alternative is to call rd_kafka_last_error() immediately
543  *         after any of the above functions return -1 or NULL.
544  *
545  * @deprecated Use rd_kafka_last_error() to retrieve the last error code
546  *             set by the legacy Librdkafka APIs.
547  *
548  * @sa rd_kafka_last_error()
549  */
550 RD_EXPORT RD_DEPRECATED
551 rd_kafka_resp_err_t rd_kafka_errno2err(int errnox);
552
553
554 /**
555  * @brief Returns the thread-local system errno
556  *
557  * On most platforms this is the same as \p errno but in case of different
558  * runtimes between library and application (e.g., Windows static DLLs)
559  * this provides a means for exposing the errno Librdkafka uses.
560  *
561  * @remark The value is local to the current calling thread.
562  *
563  * @deprecated Use rd_kafka_last_error() to retrieve the last error code
564  *             set by the legacy Librdkafka APIs.
565  */
566 RD_EXPORT RD_DEPRECATED
567 int rd_kafka_errno (void);
568
569
570
571 /**
572  * @brief Topic+Partition place holder
573  *
574  * Generic place holder for a Topic+Partition and its related information
575  * used for multiple purposes:
576  * - consumer offset (see rd_kafka_commit(), et.al.)
577  * - group rebalancing callback (rd_kafka_conf_set_rebalance_cb())
578  * - offset commit result callback (rd_kafka_conf_set_offset_commit_cb())
579  */
580
581 /**
582  * @brief Generic place holder for a specific Topic+Partition.
583  *

```

```

584 * @sa rd_kafka_topic_partition_list_new()
585 */
586 typedef struct rd_kafka_topic_partition_s {
587     char        *topic;           /**< Topic name */
588     int32_t      partition;       /**< Partition */
589     int64_t      offset;          /**< Offset */
590     void         *metadata;       /**< Metadata */
591     size_t       metadata_size;   /**< Metadata size */
592     void         *opaque;         /**< Application opaque */
593     rd_kafka_resp_err_t err;      /**< Error code, depending on use. */
594     void         *_private;       /**< INTERNAL USE ONLY,
595                                   *   INITIALIZE TO ZERO, DO NOT TOUCH */
596 } rd_kafka_topic_partition_t;
597
598
599 /**
600  * @brief Destroy a rd_kafka_topic_partition_t.
601  * @remark This must not be called for elements in a topic partition list.
602  */
603 RD_EXPORT
604 void rd_kafka_topic_partition_destroy (rd_kafka_topic_partition_t *rktpar);
605
606
607 /**
608  * @brief A growable list of Topic+Partitions.
609  *
610  */
611 typedef struct rd_kafka_topic_partition_list_s {
612     int cnt;           /**< Current number of elements */
613     int size;          /**< Current allocated size */
614     rd_kafka_topic_partition_t *elems; /**< Element array[] */
615 } rd_kafka_topic_partition_list_t;
616
617
618 /**
619  * @brief Create a new list/vector Topic+Partition container.
620  *
621  * @param size Initial allocated size used when the expected number of
622  *             elements is known or can be estimated.
623  *             Avoids reallocation and possibly relocation of the
624  *             elems array.
625  *
626  * @returns A newly allocated Topic+Partition list.
627  *
628  * @remark Use rd_kafka_topic_partition_list_destroy() to free all resources
629  *          in use by a list and the list itself.
630  * @sa      rd_kafka_topic_partition_list_add()
631  */
632 RD_EXPORT
633 rd_kafka_topic_partition_list_t *rd_kafka_topic_partition_list_new (int size);
634
635
636 /**
637  * @brief Free all resources used by the list and the list itself.
638  */
639 RD_EXPORT
640 void
641 rd_kafka_topic_partition_list_destroy (rd_kafka_topic_partition_list_t *rkparlist);
642
643 /**
644  * @brief Add topic+partition to list

```

```

645  *
646  * @param rktparlist List to extend
647  * @param topic      Topic name (copied)
648  * @param partition  Partition id
649  *
650  * @returns The object which can be used to fill in additional fields.
651  */
652  RD_EXPORT
653  rd_kafka_topic_partition_t *
654  rd_kafka_topic_partition_list_add (rd_kafka_topic_partition_list_t *rktparlist,
655                                   const char *topic, int32_t partition);
656
657
658  /**
659   * @brief Add range of partitions from \p start to \p stop inclusive.
660   *
661   * @param rktparlist List to extend
662   * @param topic      Topic name (copied)
663   * @param start      Start partition of range
664   * @param stop       Last partition of range (inclusive)
665   */
666  RD_EXPORT
667  void
668  rd_kafka_topic_partition_list_add_range (rd_kafka_topic_partition_list_t
669                                          *rktparlist,
670                                          const char *topic,
671                                          int32_t start, int32_t stop);
672
673
674
675  /**
676   * @brief Delete partition from list.
677   *
678   * @param rktparlist List to modify
679   * @param topic      Topic name to match
680   * @param partition  Partition to match
681   *
682   * @returns 1 if partition was found (and removed), else 0.
683   *
684   * @remark Any held indices to elems[] are unusable after this call returns 1.
685   */
686  RD_EXPORT
687  int
688  rd_kafka_topic_partition_list_del (rd_kafka_topic_partition_list_t *rktparlist,
689                                   const char *topic, int32_t partition);
690
691
692  /**
693   * @brief Delete partition from list by elems[] index.
694   *
695   * @returns 1 if partition was found (and removed), else 0.
696   *
697   * @sa rd_kafka_topic_partition_list_del()
698   */
699  RD_EXPORT
700  int
701  rd_kafka_topic_partition_list_del_by_idx (
702      rd_kafka_topic_partition_list_t *rktparlist,
703      int idx);
704
705

```

```

706 /**
707  * @brief Make a copy of an existing List.
708  *
709  * @param src    The existing List to copy.
710  *
711  * @returns A new List fully populated to be identical to \p src
712  */
713 RD_EXPORT
714 rd_kafka_topic_partition_list_t *
715 rd_kafka_topic_partition_list_copy (const rd_kafka_topic_partition_list_t *src);
716
717
718
719
720 /**
721  * @brief Set offset to \p offset for \p topic and \p partition
722  *
723  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or
724  *          RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION if \p partition was not found
725  *          in the List.
726  */
727 RD_EXPORT
728 rd_kafka_resp_err_t rd_kafka_topic_partition_list_set_offset (
729     rd_kafka_topic_partition_list_t *rktparlist,
730     const char *topic, int32_t partition, int64_t offset);
731
732
733
734 /**
735  * @brief Find element by \p topic and \p partition.
736  *
737  * @returns a pointer to the first matching element, or NULL if not found.
738  */
739 RD_EXPORT
740 rd_kafka_topic_partition_t *
741 rd_kafka_topic_partition_list_find (rd_kafka_topic_partition_list_t *rktparlist,
742     const char *topic, int32_t partition);
743
744
745 /**
746  * @brief Sort List using comparator \p cmp.
747  *
748  * If \p cmp is NULL the default comparator will be used that
749  * sorts by ascending topic name and partition.
750  *
751  */
752 RD_EXPORT void
753 rd_kafka_topic_partition_list_sort (rd_kafka_topic_partition_list_t *rktparlist,
754     int (*cmp) (const void *a, const void *b,
755                 void *opaque),
756     void *opaque);
757
758
759 /**@}*/
760
761
762
763 /**
764  * @name Var-arg tag types
765  * @{
766  *

```

```

767 */
768
769 /**
770  * @enum rd_kafka_vtype_t
771  *
772  * @brief Var-arg tag types
773  *
774  * @sa rd_kafka_producev()
775  */
776 typedef enum rd_kafka_vtype_t {
777     RD_KAFKA_VTYPE_END,          /**< va-arg sentinel */
778     RD_KAFKA_VTYPE_TOPIC,        /**< (const char *) Topic name */
779     RD_KAFKA_VTYPE_RKT,          /**< (rd_kafka_topic_t *) Topic handle */
780     RD_KAFKA_VTYPE_PARTITION,    /**< (int32_t) Partition */
781     RD_KAFKA_VTYPE_VALUE,        /**< (void *, size_t) Message value (payload)*/
782     RD_KAFKA_VTYPE_KEY,          /**< (void *, size_t) Message key */
783     RD_KAFKA_VTYPE_OPAQUE,       /**< (void *) Application opaque */
784     RD_KAFKA_VTYPE_MSGFLAGS,     /**< (int) RD_KAFKA_MSG_F_.. flags */
785     RD_KAFKA_VTYPE_TIMESTAMP,    /**< (int64_t) Milliseconds since epoch UTC */
786 } rd_kafka_vtype_t;
787
788
789 /**
790  * @brief Convenience macros for rd_kafka_vtype_t that takes the
791  *       correct arguments for each vtype.
792  */
793
794 /*!
795  * va-arg end sentinel used to terminate the variable argument list
796  */
797 #define RD_KAFKA_V_END RD_KAFKA_VTYPE_END
798
799 /*!
800  * Topic name (const char *)
801  */
802 #define RD_KAFKA_V_TOPIC(topic)          \
803     _LRK_TYPECHECK(RD_KAFKA_VTYPE_TOPIC, const char *, topic), \
804     (const char *)topic
805 /*!
806  * Topic object (rd_kafka_topic_t *)
807  */
808 #define RD_KAFKA_V_RKT(rkt)              \
809     _LRK_TYPECHECK(RD_KAFKA_VTYPE_RKT, rd_kafka_topic_t *, rkt), \
810     (rd_kafka_topic_t *)rkt
811 /*!
812  * Partition (int32_t)
813  */
814 #define RD_KAFKA_V_PARTITION(partition)  \
815     _LRK_TYPECHECK(RD_KAFKA_VTYPE_PARTITION, int32_t, partition), \
816     (int32_t)partition
817 /*!
818  * Message value/payload pointer and length (void *, size_t)
819  */
820 #define RD_KAFKA_V_VALUE(VALUE,LEN)      \
821     _LRK_TYPECHECK2(RD_KAFKA_VTYPE_VALUE, void *, VALUE, size_t, LEN), \
822     (void *)VALUE, (size_t)LEN
823 /*!
824  * Message key pointer and length (const void *, size_t)
825  */
826 #define RD_KAFKA_V_KEY(KEY,LEN)          \
827     _LRK_TYPECHECK2(RD_KAFKA_VTYPE_KEY, const void *, KEY, size_t, LEN), \

```

```

828 (void *)KEY, (size_t)LEN
829 /*!
830  * Opaque pointer (void *)
831  */
832 #define RD_KAFKA_V_OPAQUE(opaque) \
833      _LRK_TYPECHECK(RD_KAFKA_VTYPE_OPAQUE, void *, opaque), \
834      (void *)opaque
835 /*!
836  * Message flags (int)
837  * @sa RD_KAFKA_MSG_F_COPY, et.al.
838  */
839 #define RD_KAFKA_V_MSGFLAGS(msgflags) \
840      _LRK_TYPECHECK(RD_KAFKA_VTYPE_MSGFLAGS, int, msgflags), \
841      (int)msgflags
842 /*!
843  * Timestamp (int64_t)
844  */
845 #define RD_KAFKA_V_TIMESTAMP(timestamp) \
846      _LRK_TYPECHECK(RD_KAFKA_VTYPE_TIMESTAMP, int64_t, timestamp), \
847      (int64_t)timestamp
848
849 /**@}*/
850
851
852 /**
853  * @name Kafka messages
854  * @{
855  *
856  */
857
858
859
860 // FIXME: This doesn't show up in docs for some reason
861 // "Compound rd_kafka_message_t is not documented."
862
863 /**
864  * @brief A Kafka message as returned by the \c rd_kafka_consume*() family
865  *       of functions as well as provided to the Producer \c dr_msg_cb().
866  *
867  * For the consumer this object has two purposes:
868  * - provide the application with a consumed message. (\c err == 0)
869  * - report per-topic+partition consumer errors (\c err != 0)
870  *
871  * The application must check \c err to decide what action to take.
872  *
873  * When the application is finished with a message it must call
874  * rd_kafka_message_destroy() unless otherwise noted.
875  */
876 typedef struct rd_kafka_message_s {
877      rd_kafka_resp_err_t err;    /**< Non-zero for error signaling. */
878      rd_kafka_topic_t *rkt;     /**< Topic */
879      int32_t partition;        /**< Partition */
880      void *payload;            /**< Producer: original message payload.
881      * Consumer: Depends on the value of \c err :
882      * - \c err==0: Message payload.
883      * - \c err!=0: Error string */
884      size_t len;                /**< Depends on the value of \c err :
885      * - \c err==0: Message payload length
886      * - \c err!=0: Error string length */
887      void *key;                 /**< Depends on the value of \c err :
888      * - \c err==0: Optional message key */

```

```

889     size_t key_len;                                /**< Depends on the value of \c err :
890     * - \c err==0: Optional message key length*/
891     int64_t offset;                                /**< Consume:
892     * - Message offset (or offset for error
893     *   if \c err!=0 if applicable).
894     * - dr_msg_cb:
895     *   Message offset assigned by broker.
896     *   If \c produce.offset.report is set then
897     *   each message will have this field set,
898     *   otherwise only the last message in
899     *   each produced internal batch will
900     *   have this field set, otherwise 0. */
901     void *_private;                                /**< Consume:
902     * - rdkafka private pointer: DO NOT MODIFY
903     *   - dr_msg_cb:
904     *     msg_opaque from produce() call */
905 } rd_kafka_message_t;
906
907
908 /**
909  * @brief Frees resources for \p rkmessage and hands ownership back to rdkafka.
910  */
911 RD_EXPORT
912 void rd_kafka_message_destroy(rd_kafka_message_t *rkmessage);
913
914
915
916
917 /**
918  * @brief Returns the error string for an errored rd_kafka_message_t or NULL if
919  *   there was no error.
920  *
921  * @remark This function MUST NOT be used with the producer.
922  */
923 static RD_INLINE const char *
924 RD_UNUSED
925 rd_kafka_message_errstr(const rd_kafka_message_t *rkmessage) {
926     if (!rkmessage->err)
927         return NULL;
928
929     if (rkmessage->payload)
930         return (const char *)rkmessage->payload;
931
932     return rd_kafka_err2str(rkmessage->err);
933 }
934
935
936
937 /**
938  * @brief Returns the message timestamp for a consumed message.
939  *
940  * The timestamp is the number of milliseconds since the epoch (UTC).
941  *
942  * \p tstype (if not NULL) is updated to indicate the type of timestamp.
943  *
944  * @returns message timestamp, or -1 if not available.
945  *
946  * @remark Message timestamps require broker version 0.10.0 or later.
947  */
948 RD_EXPORT
949 int64_t rd_kafka_message_timestamp (const rd_kafka_message_t *rkmessage,

```



```

950         rd_kafka_timestamp_type_t *tstype);
951
952
953
954 /**
955  * @brief Returns the latency for a produced message measured from
956  *       the produce() call.
957  *
958  * @returns the latency in microseconds, or -1 if not available.
959  */
960 RD_EXPORT
961 int64_t rd_kafka_message_latency (const rd_kafka_message_t *rkmessage);
962
963
964 /**@}*/
965
966
967 /**
968  * @name Configuration interface
969  * @{
970  *
971  * @brief Main/global configuration property interface
972  *
973  */
974
975 /**
976  * @enum rd_kafka_conf_res_t
977  * @brief Configuration result type
978  */
979 typedef enum {
980     RD_KAFKA_CONF_UNKNOWN = -2, /**< Unknown configuration name. */
981     RD_KAFKA_CONF_INVALID = -1, /**< Invalid configuration value. */
982     RD_KAFKA_CONF_OK = 0        /**< Configuration okay */
983 } rd_kafka_conf_res_t;
984
985
986 /**
987  * @brief Create configuration object.
988  *
989  * When providing your own configuration to the \c rd_kafka_*_new_*() calls
990  * the rd_kafka_conf_t objects needs to be created with this function
991  * which will set up the defaults.
992  * I.e.:
993  * @code
994  *     rd_kafka_conf_t *myconf;
995  *     rd_kafka_conf_res_t res;
996  *
997  *     myconf = rd_kafka_conf_new();
998  *     res = rd_kafka_conf_set(myconf, "socket.timeout.ms", "600",
999  *                             errstr, sizeof(errstr));
1000  *     if (res != RD_KAFKA_CONF_OK)
1001  *         die("%s\n", errstr);
1002  *
1003  *     rk = rd_kafka_new(..., myconf);
1004  * @endcode
1005  *
1006  * Please see CONFIGURATION.md for the default settings or use
1007  * rd_kafka_conf_properties_show() to provide the information at runtime.
1008  *
1009  * The properties are identical to the Apache Kafka configuration properties
1010  * whenever possible.

```

```

1011 *
1012 * @returns A new rd_kafka_conf_t object with defaults set.
1013 *
1014 * @sa rd_kafka_conf_set(), rd_kafka_conf_destroy()
1015 */
1016 RD_EXPORT
1017 rd_kafka_conf_t *rd_kafka_conf_new(void);
1018
1019
1020 /**
1021 * @brief Destroys a conf object.
1022 */
1023 RD_EXPORT
1024 void rd_kafka_conf_destroy(rd_kafka_conf_t *conf);
1025
1026
1027 /**
1028 * @brief Creates a copy/duplicate of configuration object \p conf
1029 *
1030 * @remark Interceptors are NOT copied to the new configuration object.
1031 * @sa rd_kafka_interceptor_f_on_conf_dup
1032 */
1033 RD_EXPORT
1034 rd_kafka_conf_t *rd_kafka_conf_dup(const rd_kafka_conf_t *conf);
1035
1036
1037 /**
1038 * @brief Same as rd_kafka_conf_dup() but with an array of property name
1039 * prefixes to filter out (ignore) when copying.
1040 */
1041 RD_EXPORT
1042 rd_kafka_conf_t *rd_kafka_conf_dup_filter (const rd_kafka_conf_t *conf,
1043                                           size_t filter_cnt,
1044                                           const char **filter);
1045
1046
1047
1048 /**
1049 * @brief Sets a configuration property.
1050 *
1051 * \p conf must have been previously created with rd_kafka_conf_new().
1052 *
1053 * Fallthrough:
1054 * Topic-level configuration properties may be set using this interface
1055 * in which case they are applied on the \c default_topic_conf.
1056 * If no \c default_topic_conf has been set one will be created.
1057 * Any sub-sequent rd_kafka_conf_set_default_topic_conf() calls will
1058 * replace the current default topic configuration.
1059 *
1060 * @returns \c rd_kafka_conf_res_t to indicate success or failure.
1061 * In case of failure \p errstr is updated to contain a human readable
1062 * error string.
1063 */
1064 RD_EXPORT
1065 rd_kafka_conf_res_t rd_kafka_conf_set(rd_kafka_conf_t *conf,
1066                                       const char *name,
1067                                       const char *value,
1068                                       char *errstr, size_t errstr_size);
1069
1070
1071 /**

```

```

1072 * @brief Enable event sourcing.
1073 * \p events is a bitmask of \c RD_KAFKA_EVENT_* of events to enable
1074 * for consumption by `rd_kafka_queue_poll()`.
1075 */
1076 RD_EXPORT
1077 void rd_kafka_conf_set_events(rd_kafka_conf_t *conf, int events);
1078
1079
1080 /**
1081  * @deprecated See rd_kafka_conf_set_dr_msg_cb()
1082  */
1083 RD_EXPORT
1084 void rd_kafka_conf_set_dr_cb(rd_kafka_conf_t *conf,
1085                             void (*dr_cb) (rd_kafka_t *rk,
1086                                             void *payload, size_t len,
1087                                             rd_kafka_resp_err_t err,
1088                                             void *opaque, void *msg_opaque));
1089
1090 /**
1091  * @brief \b Producer: Set delivery report callback in provided \p conf object.
1092  *
1093  * The delivery report callback will be called once for each message
1094  * accepted by rd_kafka_produce() (et.al) with \p err set to indicate
1095  * the result of the produce request.
1096  *
1097  * The callback is called when a message is succesfully produced or
1098  * if librdkafka encountered a permanent failure, or the retry counter for
1099  * temporary errors has been exhausted.
1100  *
1101  * An application must call rd_kafka_poll() at regular intervals to
1102  * serve queued delivery report callbacks.
1103  */
1104 RD_EXPORT
1105 void rd_kafka_conf_set_dr_msg_cb(rd_kafka_conf_t *conf,
1106                                  void (*dr_msg_cb) (rd_kafka_t *rk,
1107                                                       const rd_kafka_message_t *
1108                                                       rkmessage,
1109                                                       void *opaque));
1110
1111
1112 /**
1113  * @brief \b Consumer: Set consume callback for use with rd_kafka_consumer_poll()
1114  *
1115  */
1116 RD_EXPORT
1117 void rd_kafka_conf_set_consume_cb (rd_kafka_conf_t *conf,
1118                                    void (*consume_cb) (rd_kafka_message_t *
1119                                                         rkmessage,
1120                                                         void *opaque));
1121
1122 /**
1123  * @brief \b Consumer: Set rebalance callback for use with
1124  * coordinated consumer group balancing.
1125  *
1126  * The \p err field is set to either RD_KAFKA_RESP_ERR__ASSIGN_PARTITIONS
1127  * or RD_KAFKA_RESP_ERR__REVOKE_PARTITIONS and 'partitions'
1128  * contains the full partition set that was either assigned or revoked.
1129  *
1130  * Registering a \p rebalance_cb turns off librdkafka's automatic
1131  * partition assignment/revocation and instead delegates that responsibility
1132  * to the application's \p rebalance_cb.

```

```

1133 *
1134 * The rebalance callback is responsible for updating librdkafka's
1135 * assignment set based on the two events: RD_KAFKA_RESP_ERR__ASSIGN_PARTITIONS
1136 * and RD_KAFKA_RESP_ERR__REVOKE_PARTITIONS but should also be able to handle
1137 * arbitrary rebalancing failures where \p err is neither of those.
1138 * @remark In this latter case (arbitrary error), the application must
1139 * call rd_kafka_assign(rk, NULL) to synchronize state.
1140 *
1141 * Without a rebalance callback this is done automatically by Librdkafka
1142 * but registering a rebalance callback gives the application flexibility
1143 * in performing other operations along with the assigning/revocation,
1144 * such as fetching offsets from an alternate location (on assign)
1145 * or manually committing offsets (on revoke).
1146 *
1147 * @remark The \p partitions list is destroyed by Librdkafka on return
1148 * return from the rebalance_cb and must not be freed or
1149 * saved by the application.
1150 *
1151 * The following example shows the application's responsibilities:
1152 * @code
1153 * static void rebalance_cb (rd_kafka_t *rk, rd_kafka_resp_err_t err,
1154 *                          rd_kafka_topic_partition_list_t *partitions,
1155 *                          void *opaque) {
1156 *
1157 *     switch (err)
1158 *     {
1159 *     case RD_KAFKA_RESP_ERR__ASSIGN_PARTITIONS:
1160 *         // application may load offsets from arbitrary external
1161 *         // storage here and update \p partitions
1162 *
1163 *         rd_kafka_assign(rk, partitions);
1164 *         break;
1165 *
1166 *     case RD_KAFKA_RESP_ERR__REVOKE_PARTITIONS:
1167 *         if (manual_commits) // Optional explicit manual commit
1168 *             rd_kafka_commit(rk, partitions, 0); // sync commit
1169 *
1170 *         rd_kafka_assign(rk, NULL);
1171 *         break;
1172 *
1173 *     default:
1174 *         handle_unlikely_error(err);
1175 *         rd_kafka_assign(rk, NULL); // sync state
1176 *         break;
1177 *     }
1178 * }
1179 * @endcode
1180 */
1181 RD_EXPORT
1182 void rd_kafka_conf_set_rebalance_cb (
1183     rd_kafka_conf_t *conf,
1184     void (*rebalance_cb) (rd_kafka_t *rk,
1185                          rd_kafka_resp_err_t err,
1186                          rd_kafka_topic_partition_list_t *partitions,
1187                          void *opaque));
1188
1189
1190
1191 /**
1192 * @brief \b Consumer: Set offset commit callback for use with consumer groups.
1193 *

```

```

1194 * The results of automatic or manual offset commits will be scheduled
1195 * for this callback and is served by rd_kafka_consumer_poll().
1196 *
1197 * If no partitions had valid offsets to commit this callback will be called
1198 * with \p err == RD_KAFKA_RESP_ERR__NO_OFFSET which is not to be considered
1199 * an error.
1200 *
1201 * The \p offsets List contains per-partition information:
1202 * - \c offset: committed offset (attempted)
1203 * - \c err:    commit error
1204 */
1205 RD_EXPORT
1206 void rd_kafka_conf_set_offset_commit_cb (
1207     rd_kafka_conf_t *conf,
1208     void (*offset_commit_cb) (rd_kafka_t *rk,
1209                               rd_kafka_resp_err_t err,
1210                               rd_kafka_topic_partition_list_t *offsets,
1211                               void *opaque));
1212
1213 /**
1214 * @brief Set error callback in provided conf object.
1215 *
1216 * The error callback is used by Librdkafka to signal critical errors
1217 * back to the application.
1218 *
1219 * If no \p error_cb is registered then the errors will be logged instead.
1220 */
1221
1222 RD_EXPORT
1223 void rd_kafka_conf_set_error_cb(rd_kafka_conf_t *conf,
1224     void (*error_cb) (rd_kafka_t *rk, int err,
1225                      const char *reason,
1226                      void *opaque));
1227
1228 /**
1229 * @brief Set throttle callback.
1230 *
1231 * The throttle callback is used to forward broker throttle times to the
1232 * application for Produce and Fetch (consume) requests.
1233 *
1234 * Callbacks are triggered whenever a non-zero throttle time is returned by
1235 * the broker, or when the throttle time drops back to zero.
1236 *
1237 * An application must call rd_kafka_poll() or rd_kafka_consumer_poll() at
1238 * regular intervals to serve queued callbacks.
1239 *
1240 * @remark Requires broker version 0.9.0 or Later.
1241 */
1242 RD_EXPORT
1243 void rd_kafka_conf_set_throttle_cb (rd_kafka_conf_t *conf,
1244     void (*throttle_cb) (
1245         rd_kafka_t *rk,
1246         const char *broker_name,
1247         int32_t broker_id,
1248         int throttle_time_ms,
1249         void *opaque));
1250
1251 /**
1252 * @brief Set logger callback.
1253 *
1254 *

```

```

1255 * The default is to print to stderr, but a syslog logger is also available,
1256 * see rd_kafka_log_print and rd_kafka_log_syslog for the builtin alternatives.
1257 * Alternatively the application may provide its own logger callback.
1258 * Or pass \p func as NULL to disable logging.
1259 *
1260 * This is the configuration alternative to the deprecated rd_kafka_set_logger()
1261 *
1262 * @remark The log_cb will be called spontaneously from librdkafka's internal
1263 *         threads unless logs have been forwarded to a poll queue through
1264 *         \c rd_kafka_set_log_queue().
1265 *         An application MUST NOT call any librdkafka APIs or do any prolonged
1266 *         work in a non-forwarded \c log_cb.
1267 */
1268 RD_EXPORT
1269 void rd_kafka_conf_set_log_cb(rd_kafka_conf_t *conf,
1270                             void (*log_cb) (const rd_kafka_t *rk, int level,
1271                                             const char *fac, const char *buf));
1272
1273
1274 /**
1275 * @brief Set statistics callback in provided conf object.
1276 *
1277 * The statistics callback is triggered from rd_kafka_poll() every
1278 * \c statistics.interval.ms (needs to be configured separately).
1279 * Function arguments:
1280 *   - \p rk - Kafka handle
1281 *   - \p json - String containing the statistics data in JSON format
1282 *   - \p json_len - Length of \p json string.
1283 *   - \p opaque - Application-provided opaque.
1284 *
1285 * If the application wishes to hold on to the \p json pointer and free
1286 * it at a later time it must return 1 from the \p stats_cb.
1287 * If the application returns 0 from the \p stats_cb then librdkafka
1288 * will immediately free the \p json pointer.
1289 */
1290 RD_EXPORT
1291 void rd_kafka_conf_set_stats_cb(rd_kafka_conf_t *conf,
1292                                int (*stats_cb) (rd_kafka_t *rk,
1293                                                char *json,
1294                                                size_t json_len,
1295                                                void *opaque));
1296
1297
1298
1299 /**
1300 * @brief Set socket callback.
1301 *
1302 * The socket callback is responsible for opening a socket
1303 * according to the supplied \p domain, \p type and \p protocol.
1304 * The socket shall be created with \c CLOEXEC set in a racefree fashion, if
1305 * possible.
1306 *
1307 * Default:
1308 *   - on linux: racefree CLOEXEC
1309 *   - others : non-racefree CLOEXEC
1310 *
1311 * @remark The callback will be called from an internal librdkafka thread.
1312 */
1313 RD_EXPORT
1314 void rd_kafka_conf_set_socket_cb(rd_kafka_conf_t *conf,
1315                                 int (*socket_cb) (int domain, int type,

```



pyxllq

码龄12年 暂无认证

25 14万+ 13万+ 5万+
原创 周排名 总排名 访问 等级

```
1316         int protocol,  
1317         void *opaque));  
1318  
1319  
1320  
1321 /**  
1322  * @brief Set connect callback.  
1323  *  
1324  * The connect callback is responsible for connecting socket \p sockfd  
1325  * to peer address \p addr.  
1326  * The \p id field contains the broker identifier.  
1327  *  
1328  * \p connect_cb shall return 0 on success (socket connected) or an error  
1329  * number (errno) on error.  
1330  *  
1331  * @remark The callback will be called from an internal librdkafka thread.  
1332  */  
1333 RD_EXPORT void  
1334 rd_kafka_conf_set_connect_cb (rd_kafka_conf_t *conf,  
1335                               int (*connect_cb) (int sockfd,  
1336                                                  const struct sockaddr *addr,  
1337                                                  int addrlen,  
1338                                                  const char *id,  
1339                                                  void *opaque));  
1340  
1341 /**  
1342  * @brief Set close socket callback.  
1343  *  
1344  * Close a socket (optionally opened with socket_cb()).  
1345  *  
1346  * @remark The callback will be called from an internal librdkafka thread.  
1347  */  
1348 RD_EXPORT void  
1349 rd_kafka_conf_set_closesocket_cb (rd_kafka_conf_t *conf,  
1350                                   int (*closesocket_cb) (int sockfd,  
1351                                                         void *opaque));  
1352  
1353  
1354  
1355 #ifndef _MSC_VER  
1356 /**  
1357  * @brief Set open callback.  
1358  *  
1359  * The open callback is responsible for opening the file specified by  
1360  * pathname, flags and mode.  
1361  * The file shall be opened with \c CLOEXEC set in a racefree fashion, if  
1362  * possible.  
1363  *  
1364  * Default:  
1365  * - on linux: racefree CLOEXEC  
1366  * - others : non-racefree CLOEXEC  
1367  *  
1368  * @remark The callback will be called from an internal librdkafka thread.  
1369  */  
1370 RD_EXPORT  
1371 void rd_kafka_conf_set_open_cb (rd_kafka_conf_t *conf,  
1372                                 int (*open_cb) (const char *pathname,  
1373                                                  int flags, mode_t mode,  
1374                                                  void *opaque));  
1375 #endif  
1376
```

678

5

14

2

50

积分

粉丝

获赞

评论

收藏

恒

1

私信

关注

搜博主文章

热门文章

net.ipv4.tcp_timestamps引发的tcp syn无响应案

16493

udp使用connect建立连接与普通udp连接区别

8218

c语言使用kafka消息队列的实例

7506

优化--单台服务器支持百万级别tcp长连接

4806

vue项目：Cannot read property '_t' of null报错

3311

最新评论

优化--单台服务器支持百万级别tcp长连接_仰望星空_：内存需要多大

net.ipv4.tcp_timestamps引发的tcp syn无...

pan_shu: https://github.com/11061055/ph

p-7.3.0-ext-curl/wiki/linux-%E5%86%85%

您愿意向朋友推荐“博客详情页”吗?

强烈不推荐

不推荐

一般般

推荐

强烈推荐

最新文章

网卡收包中断的上半部和下半部

记录相关库或者工具以及知识点

循环队列 代码实现 (FIFO)

2021年 4篇

2020年 9篇

2018年 23篇

1377

/**

1378

** @brief Sets the application's opaque pointer that will be passed to callbacks*

1379

**/*

1380

RD_EXPORT

1381

*void rd_kafka_conf_set_opaque(rd_kafka_conf_t *conf, void *opaque);*

1382

1383

*/***

1384

** @brief Retrieves the opaque pointer previously set with rd_kafka_conf_set_opaque()*

1385

**/*

1386

RD_EXPORT

1387

*void *rd_kafka_opaque(const rd_kafka_t *rk);*

1388

1389

1390

1391

*/***

1392

** Sets the default topic configuration to use for automatically*

1393

** subscribed topics (e.g., through pattern-matched topics).*

1394

** The topic config object is not usable after this call.*

1395

**/*

1396

RD_EXPORT

1397

*void rd_kafka_conf_set_default_topic_conf (rd_kafka_conf_t *conf,*

1398

*rd_kafka_topic_conf_t *tconf);*

1399

1400

1401

1402

*/***

1403

** @brief Retrieve configuration value for property \p name.*

1404

1405

** If \p dest is non-NULL the value will be written to \p dest with at*

1406

** most \p dest_size.*

1407

1408

** \p *dest_size is updated to the full length of the value, thus if*

1409

** \p *dest_size initially is smaller than the full length the application*

1410

** may reallocate \p dest to fit the returned \p *dest_size and try again.*

1411

1412

** If \p dest is NULL only the full length of the value is returned.*

1413

1414

** Fallthrough:*

1415

** Topic-level configuration properties from the \c default_topic_conf*

1416

** may be retrieved using this interface.*

1417

1418

** @returns \p RD_KAFKA_CONF_OK if the property name matched, else*

1419

** \p RD_KAFKA_CONF_UNKNOWN.*

1420

**/*

1421

RD_EXPORT

1422

*rd_kafka_conf_res_t rd_kafka_conf_get (const rd_kafka_conf_t *conf,*

1423

*const char *name,*

1424

*char *dest, size_t *dest_size);*

1425

pyxlq

关注

0

0

10

专栏目录

会员中心 足迹 动态 消息 创作抽奖

分类专栏

C

算法

1篇

C

MySQL

2篇


```

1437
1438 /**
1439  * @brief Dump the configuration properties and values of \p conf to an array
1440  *       with \key\, \value\ pairs.
1441  *
1442  * The number of entries in the array is returned in \p *cntp.
1443  *
1444  * The dump must be freed with `rd_kafka_conf_dump_free()`.
1445  */
1446 RD_EXPORT
1447 const char **rd_kafka_conf_dump(rd_kafka_conf_t *conf, size_t *cntp);
1448
1449
1450 /**
1451  * @brief Dump the topic configuration properties and values of \p conf
1452  *       to an array with \key\, \value\ pairs.
1453  *
1454  * The number of entries in the array is returned in \p *cntp.
1455  *
1456  * The dump must be freed with `rd_kafka_conf_dump_free()`.
1457  */
1458 RD_EXPORT
1459 const char **rd_kafka_topic_conf_dump(rd_kafka_topic_conf_t *conf,
1460                                       size_t *cntp);
1461
1462 /**
1463  * @brief Frees a configuration dump returned from `rd_kafka_conf_dump()` or
1464  *       `rd_kafka_topic_conf_dump()`.
1465  */
1466 RD_EXPORT
1467 void rd_kafka_conf_dump_free(const char **arr, size_t cnt);
1468
1469 /**
1470  * @brief Prints a table to \p fp of all supported configuration properties,
1471  *       their default values as well as a description.
1472  */
1473 RD_EXPORT
1474 void rd_kafka_conf_properties_show(FILE *fp);
1475
1476 /**@}*/
1477
1478
1479 /**
1480  * @name Topic configuration
1481  * @{
1482  *
1483  * @brief Topic configuration property interface
1484  *
1485  */
1486
1487
1488 /**
1489  * @brief Create topic configuration object
1490  *
1491  * @sa Same semantics as for rd_kafka_conf_new().
1492  */
1493 RD_EXPORT
1494 rd_kafka_topic_conf_t *rd_kafka_topic_conf_new(void);
1495
1496
1497 /**

```

	DPDK	1篇
	Vue开发	1篇
	编程	11篇
	python	2篇
	mysql常用命令	2篇
	归档	5篇
	分布式	2篇
	linux	7篇

```

1498  * @brief Creates a copy/duplicate of topic configuration object \p conf.
1499  */
1500  RD_EXPORT
1501  rd_kafka_topic_conf_t *rd_kafka_topic_conf_dup(const rd_kafka_topic_conf_t
1502          * c o n f ) ;
1503
1504
1505  /**
1506   * @brief Destroys a topic conf object.
1507   */
1508  RD_EXPORT
1509  void rd_kafka_topic_conf_destroy(rd_kafka_topic_conf_t *topic_conf);
1510
1511
1512  /**
1513   * @brief Sets a single rd_kafka_topic_conf_t value by property name.
1514   *
1515   * \p topic_conf should have been previously set up
1516   * with `rd_kafka_topic_conf_new()`.
1517   *
1518   * @returns rd_kafka_conf_res_t to indicate success or failure.
1519   */
1520  RD_EXPORT
1521  rd_kafka_conf_res_t rd_kafka_topic_conf_set(rd_kafka_topic_conf_t *conf,
1522          const char *name,
1523          const char *value,
1524          char *errstr, size_t errstr_size);
1525
1526  /**
1527   * @brief Sets the application's opaque pointer that will be passed to all topic
1528   * callbacks as the \c rkt_opaque argument.
1529   */
1530  RD_EXPORT
1531  void rd_kafka_topic_conf_set_opaque(rd_kafka_topic_conf_t *conf, void *opaque);
1532
1533
1534  /**
1535   * @brief \b Producer: Set partitioner callback in provided topic conf object.
1536   *
1537   * The partitioner may be called in any thread at any time,
1538   * it may be called multiple times for the same message/key.
1539   *
1540   * Partitioner function constraints:
1541   * - MUST NOT call any rd_kafka_*( ) functions except:
1542   *   rd_kafka_topic_partition_available()
1543   * - MUST NOT block or execute for prolonged periods of time.
1544   * - MUST return a value between 0 and partition_cnt-1, or the
1545   *   special \c RD_KAFKA_PARTITION_UA value if partitioning
1546   *   could not be performed.
1547   */
1548  RD_EXPORT
1549  void
1550  rd_kafka_topic_conf_set_partitioner_cb (rd_kafka_topic_conf_t *topic_conf,
1551          int32_t (*partitioner) (
1552          const rd_kafka_topic_t *rkt,
1553          const void *keydata,
1554          size_t keylen,
1555          int32_t partition_cnt,
1556          void *rkt_opaque,
1557          void *msg_opaque));
1558

```

[illegible]

```

1620
1621
1622 /**@}*/
1623
1624
1625
1626 /**
1627  * @name Main Kafka and Topic object handles
1628  * @{
1629  *
1630  *
1631  */
1632
1633
1634
1635
1636 /**
1637  * @brief Creates a new Kafka handle and starts its operation according to the
1638  *        specified \p type (\p RD_KAFKA_CONSUMER or \p RD_KAFKA_PRODUCER).
1639  *
1640  * \p conf is an optional struct created with `rd_kafka_conf_new()` that will
1641  * be used instead of the default configuration.
1642  * The \p conf object is freed by this function on success and must not be used
1643  * or destroyed by the application sub-sequently.
1644  * See `rd_kafka_conf_set()` et.al for more information.
1645  *
1646  * \p errstr must be a pointer to memory of at least size \p errstr_size where
1647  * `rd_kafka_new()` may write a human readable error message in case the
1648  * creation of a new handle fails. In which case the function returns NULL.
1649  *
1650  * @remark \b RD_KAFKA_CONSUMER: When a new \p RD_KAFKA_CONSUMER
1651  *        rd_kafka_t handle is created it may either operate in the
1652  *        legacy simple consumer mode using the rd_kafka_consume_start()
1653  *        interface, or the High-Level KafkaConsumer API.
1654  * @remark An application must only use one of these groups of APIs on a given
1655  *        rd_kafka_t RD_KAFKA_CONSUMER handle.
1656  *
1657  *
1658  * @returns The Kafka handle on success or NULL on error (see \p errstr)
1659  *
1660  * @sa To destroy the Kafka handle, use rd_kafka_destroy().
1661  */
1662 RD_EXPORT
1663 rd_kafka_t *rd_kafka_new(rd_kafka_type_t type, rd_kafka_conf_t *conf,
1664                          char *errstr, size_t errstr_size);
1665
1666
1667 /**
1668  * @brief Destroy Kafka handle.
1669  *
1670  * @remark This is a blocking operation.
1671  */
1672 RD_EXPORT
1673 void rd_kafka_destroy(rd_kafka_t *rk);
1674
1675
1676
1677 /**
1678  * @brief Returns Kafka handle name.
1679  */
1680 RD_EXPORT

```

```

1681 const char *rd_kafka_name(const rd_kafka_t *rk);
1682
1683
1684 /**
1685  * @brief Returns Kafka handle type.
1686  */
1687 RD_EXPORT
1688 rd_kafka_type_t rd_kafka_type(const rd_kafka_t *rk);
1689
1690
1691 /**
1692  * @brief Returns this client's broker-assigned group member id
1693  *
1694  * @remark This currently requires the high-level KafkaConsumer
1695  *
1696  * @returns An allocated string containing the current broker-assigned group
1697  *           member id, or NULL if not available.
1698  *           The application must free the string with \p free() or
1699  *           rd_kafka_mem_free()
1700  */
1701 RD_EXPORT
1702 char *rd_kafka_memberid (const rd_kafka_t *rk);
1703
1704
1705
1706 /**
1707  * @brief Returns the ClusterId as reported in broker metadata.
1708  *
1709  * @param timeout_ms If there is no cached value from metadata retrieval
1710  *                   then this specifies the maximum amount of time
1711  *                   (in milliseconds) the call will block waiting
1712  *                   for metadata to be retrieved.
1713  *                   Use 0 for non-blocking calls.
1714  *
1715  * @remark Requires broker version >=0.10.0 and api.version.request=true.
1716  *
1717  * @remark The application must free the returned pointer
1718  *           using rd_kafka_mem_free().
1719  *
1720  * @returns a newly allocated string containing the ClusterId, or NULL
1721  *           if no ClusterId could be retrieved in the allotted timespan.
1722  */
1723 RD_EXPORT
1724 char *rd_kafka_clusterid (rd_kafka_t *rk, int timeout_ms);
1725
1726
1727 /**
1728  * @brief Creates a new topic handle for topic named \p topic.
1729  *
1730  * \p conf is an optional configuration for the topic created with
1731  * `rd_kafka_topic_conf_new()` that will be used instead of the default
1732  * topic configuration.
1733  * The \p conf object is freed by this function and must not be used or
1734  * destroyed by the application sub-sequently.
1735  * See `rd_kafka_topic_conf_set()` et.al for more information.
1736  *
1737  * Topic handles are refcounted internally and calling rd_kafka_topic_new()
1738  * again with the same topic name will return the previous topic handle
1739  * without updating the original handle's configuration.
1740  * Applications must eventually call rd_kafka_topic_destroy() for each
1741  * successful call to rd_kafka_topic_new() to clear up resources.

```

```

1742 *
1743 * @returns the new topic handle or NULL on error (use rd_kafka_errno2err()
1744 *         to convert system \p errno to an rd_kafka_resp_err_t error code.
1745 *
1746 * @sa rd_kafka_topic_destroy()
1747 */
1748 RD_EXPORT
1749 rd_kafka_topic_t *rd_kafka_topic_new(rd_kafka_t *rk, const char *topic,
1750                                     rd_kafka_topic_conf_t *conf);
1751
1752
1753
1754 /**
1755 * @brief Loose application's topic handle refcount as previously created
1756 *        with `rd_kafka_topic_new()`.
1757 *
1758 * @remark Since topic objects are refcounted (both internally and for the app)
1759 *        the topic object might not actually be destroyed by this call,
1760 *        but the application must consider the object destroyed.
1761 */
1762 RD_EXPORT
1763 void rd_kafka_topic_destroy(rd_kafka_topic_t *rkt);
1764
1765
1766 /**
1767 * @brief Returns the topic name.
1768 */
1769 RD_EXPORT
1770 const char *rd_kafka_topic_name(const rd_kafka_topic_t *rkt);
1771
1772
1773 /**
1774 * @brief Get the \p rkt_opaque pointer that was set in the topic configuration.
1775 */
1776 RD_EXPORT
1777 void *rd_kafka_topic_opaque (const rd_kafka_topic_t *rkt);
1778
1779
1780 /**
1781 * @brief Unassigned partition.
1782 *
1783 * The unassigned partition is used by the producer API for messages
1784 * that should be partitioned using the configured or default partitioner.
1785 */
1786 #define RD_KAFKA_PARTITION_UA ((int32_t)-1)
1787
1788
1789 /**
1790 * @brief Polls the provided kafka handle for events.
1791 *
1792 * Events will cause application provided callbacks to be called.
1793 *
1794 * The \p timeout_ms argument specifies the maximum amount of time
1795 * (in milliseconds) that the call will block waiting for events.
1796 * For non-blocking calls, provide 0 as \p timeout_ms.
1797 * To wait indefinitely for an event, provide -1.
1798 *
1799 * @remark An application should make sure to call poll() at regular
1800 *        intervals to serve any queued callbacks waiting to be called.
1801 *
1802 * Events:

```

```

1803 * - delivery report callbacks (if dr_cb/dr_msg_cb is configured) [producer]
1804 * - error callbacks (rd_kafka_conf_set_error_cb()) [all]
1805 * - stats callbacks (rd_kafka_conf_set_stats_cb()) [all]
1806 * - throttle callbacks (rd_kafka_conf_set_throttle_cb()) [all]
1807 *
1808 * @returns the number of events served.
1809 */
1810 RD_EXPORT
1811 int rd_kafka_poll(rd_kafka_t *rk, int timeout_ms);
1812
1813
1814 /**
1815 * @brief Cancels the current callback dispatcher (rd_kafka_poll(),
1816 * rd_kafka_consume_callback(), etc).
1817 *
1818 * A callback may use this to force an immediate return to the calling
1819 * code (caller of e.g. rd_kafka_poll()) without processing any further
1820 * events.
1821 *
1822 * @remark This function MUST ONLY be called from within a librdkafka callback.
1823 */
1824 RD_EXPORT
1825 void rd_kafka_yield (rd_kafka_t *rk);
1826
1827
1828
1829
1830 /**
1831 * @brief Pause producing or consumption for the provided list of partitions.
1832 *
1833 * Success or error is returned per-partition \p err in the \p partitions list.
1834 *
1835 * @returns RD_KAFKA_RESP_ERR_NO_ERROR
1836 */
1837 RD_EXPORT rd_kafka_resp_err_t
1838 rd_kafka_pause_partitions (rd_kafka_t *rk,
1839 rd_kafka_topic_partition_list_t *partitions);
1840
1841
1842
1843 /**
1844 * @brief Resume producing consumption for the provided list of partitions.
1845 *
1846 * Success or error is returned per-partition \p err in the \p partitions list.
1847 *
1848 * @returns RD_KAFKA_RESP_ERR_NO_ERROR
1849 */
1850 RD_EXPORT rd_kafka_resp_err_t
1851 rd_kafka_resume_partitions (rd_kafka_t *rk,
1852 rd_kafka_topic_partition_list_t *partitions);
1853
1854
1855
1856
1857 /**
1858 * @brief Query broker for low (oldest/beginning) and high (newest/end) offsets
1859 * for partition.
1860 *
1861 * Offsets are returned in \p *low and \p *high respectively.
1862 *
1863 * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code on failure.

```

```

1864 */
1865 RD_EXPORT rd_kafka_resp_err_t
1866 rd_kafka_query_watermark_offsets (rd_kafka_t *rk,
1867                                   const char *topic, int32_t partition,
1868                                   int64_t *low, int64_t *high, int timeout_ms);
1869
1870
1871 /**
1872  * @brief Get Last known Low (oldest/beginning) and high (newest/end) offsets
1873  *        for partition.
1874  *
1875  * The Low offset is updated periodically (if statistics.interval.ms is set)
1876  * while the high offset is updated on each fetched message set from the broker.
1877  *
1878  * If there is no cached offset (either low or high, or both) then
1879  * RD_KAFKA_OFFSET_INVALID will be returned for the respective offset.
1880  *
1881  * Offsets are returned in \p *Low and \p *high respectively.
1882  *
1883  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code on failure.
1884  *
1885  * @remark Shall only be used with an active consumer instance.
1886  */
1887 RD_EXPORT rd_kafka_resp_err_t
1888 rd_kafka_get_watermark_offsets (rd_kafka_t *rk,
1889                                 const char *topic, int32_t partition,
1890                                 int64_t *low, int64_t *high);
1891
1892
1893
1894 /**
1895  * @brief Look up the offsets for the given partitions by timestamp.
1896  *
1897  * The returned offset for each partition is the earliest offset whose
1898  * timestamp is greater than or equal to the given timestamp in the
1899  * corresponding partition.
1900  *
1901  * The timestamps to query are represented as \c offset in \p offsets
1902  * on input, and \c offset will contain the offset on output.
1903  *
1904  * The function will block for at most \p timeout_ms milliseconds.
1905  *
1906  * @remark Duplicate Topic+Partitions are not supported.
1907  * @remark Per-partition errors may be returned in \c rd_kafka_topic_partition_t.err
1908  *
1909  * @returns an error code for general errors, else RD_KAFKA_RESP_ERR_NO_ERROR
1910  *          in which case per-partition errors might be set.
1911  */
1912 RD_EXPORT rd_kafka_resp_err_t
1913 rd_kafka_offsets_for_times (rd_kafka_t *rk,
1914                             rd_kafka_topic_partition_list_t *offsets,
1915                             int timeout_ms);
1916
1917
1918 /**
1919  * @brief Free pointer returned by Librdkafka
1920  *
1921  * This is typically an abstraction for the free(3) call and makes sure
1922  * the application can use the same memory allocator as Librdkafka for
1923  * freeing pointers returned by Librdkafka.
1924  *

```



```

1925  * In standard setups it is usually not necessary to use this interface
1926  * rather than the free(3) function.
1927  *
1928  * @remark rd_kafka_mem_free() must only be used for pointers returned by APIs
1929  * that explicitly mention using this function for freeing.
1930  */
1931 RD_EXPORT
1932 void rd_kafka_mem_free (rd_kafka_t *rk, void *ptr);
1933
1934
1935 /**@}*/
1936
1937
1938
1939
1940
1941 /**
1942  * @name Queue API
1943  * @{
1944  *
1945  * Message queues allows the application to re-route consumed messages
1946  * from multiple topic+partitions into one single queue point.
1947  * This queue point containing messages from a number of topic+partitions
1948  * may then be served by a single rd_kafka_consume*_queue() call,
1949  * rather than one call per topic+partition combination.
1950  */
1951
1952
1953 /**
1954  * @brief Create a new message queue.
1955  *
1956  * See rd_kafka_consume_start_queue(), rd_kafka_consume_queue(), et.al.
1957  */
1958 RD_EXPORT
1959 rd_kafka_queue_t *rd_kafka_queue_new(rd_kafka_t *rk);
1960
1961 /**
1962  * Destroy a queue, purging all of its enqueued messages.
1963  */
1964 RD_EXPORT
1965 void rd_kafka_queue_destroy(rd_kafka_queue_t *rkqu);
1966
1967
1968 /**
1969  * @returns a reference to the main Librdkafka event queue.
1970  * This is the queue served by rd_kafka_poll().
1971  *
1972  * Use rd_kafka_queue_destroy() to loose the reference.
1973  */
1974 RD_EXPORT
1975 rd_kafka_queue_t *rd_kafka_queue_get_main (rd_kafka_t *rk);
1976
1977
1978 /**
1979  * @returns a reference to the Librdkafka consumer queue.
1980  * This is the queue served by rd_kafka_consumer_poll().
1981  *
1982  * Use rd_kafka_queue_destroy() to loose the reference.
1983  *
1984  * @remark rd_kafka_queue_destroy() MUST be called on this queue
1985  * prior to calling rd_kafka_consumer_close().

```

```

1986 */
1987 RD_EXPORT
1988 rd_kafka_queue_t *rd_kafka_queue_get_consumer (rd_kafka_t *rk);
1989
1990 /**
1991  * @returns a reference to the partition's queue, or NULL if
1992  *          partition is invalid.
1993  *
1994  * Use rd_kafka_queue_destroy() to loose the reference.
1995  *
1996  * @remark rd_kafka_queue_destroy() MUST be called on this queue
1997  *
1998  * @remark This function only works on consumers.
1999  */
2000 RD_EXPORT
2001 rd_kafka_queue_t *rd_kafka_queue_get_partition (rd_kafka_t *rk,
2002                                                  const char *topic,
2003                                                  int32_t partition);
2004
2005 /**
2006  * @brief Forward/re-route queue \p src to \p dst.
2007  * If \p dst is \c NULL the forwarding is removed.
2008  *
2009  * The internal refcounts for both queues are increased.
2010  *
2011  * @remark Regardless of whether \p dst is NULL or not, after calling this
2012  *          function, \p src will not forward it's fetch queue to the consumer
2013  *          queue.
2014  */
2015 RD_EXPORT
2016 void rd_kafka_queue_forward (rd_kafka_queue_t *src, rd_kafka_queue_t *dst);
2017
2018 /**
2019  * @brief Forward Librdkafka logs (and debug) to the specified queue
2020  *          for serving with one of the ..poll() calls.
2021  *
2022  *          This allows an application to serve log callbacks (\c log_cb)
2023  *          in its thread of choice.
2024  *
2025  * @param rkqu Queue to forward logs to. If the value is NULL the logs
2026  *          are forwarded to the main queue.
2027  *
2028  * @remark The configuration property \c log.queue MUST also be set to true.
2029  *
2030  * @remark Librdkafka maintains its own reference to the provided queue.
2031  *
2032  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code on error.
2033  */
2034 RD_EXPORT
2035 rd_kafka_resp_err_t rd_kafka_set_log_queue (rd_kafka_t *rk,
2036                                              rd_kafka_queue_t *rkqu);
2037
2038
2039 /**
2040  * @returns the current number of elements in queue.
2041  */
2042 RD_EXPORT
2043 size_t rd_kafka_queue_length (rd_kafka_queue_t *rkqu);
2044
2045
2046 /**

```

```

2047 * @brief Enable IO event triggering for queue.
2048 *
2049 * To ease integration with IO based polling loops this API
2050 * allows an application to create a separate file-descriptor
2051 * that librdkafka will write \p payload (of size \p size) to
2052 * whenever a new element is enqueued on a previously empty queue.
2053 *
2054 * To remove event triggering call with \p fd = -1.
2055 *
2056 * Librdkafka will maintain a copy of the \p payload.
2057 *
2058 * @remark When using forwarded queues the IO event must only be enabled
2059 * on the final forwarded-to (destination) queue.
2060 */
2061 RD_EXPORT
2062 void rd_kafka_queue_io_event_enable (rd_kafka_queue_t *rkqu, int fd,
2063                                     const void *payload, size_t size);
2064
2065 /**@}*/
2066
2067 /**
2068 *
2069 * @name Simple Consumer API (Legacy)
2070 * @{
2071 *
2072 */
2073
2074
2075 #define RD_KAFKA_OFFSET_BEGINNING -2 /**< Start consuming from beginning of
2076 * kafka partition queue: oldest msg */
2077 #define RD_KAFKA_OFFSET_END -1 /**< Start consuming from end of kafka
2078 * partition queue: next msg */
2079 #define RD_KAFKA_OFFSET_STORED -1000 /**< Start consuming from offset retrieved
2080 * from offset store */
2081 #define RD_KAFKA_OFFSET_INVALID -1001 /**< Invalid offset */
2082
2083
2084 /** @cond NO_DOC */
2085 #define RD_KAFKA_OFFSET_TAIL_BASE -2000 /* internal: do not use */
2086 /** @endcond */
2087
2088 /**
2089 * @brief Start consuming \p CNT messages from topic's current end offset.
2090 *
2091 * That is, if current end offset is 12345 and \p CNT is 200, it will start
2092 * consuming from offset \c 12345-200 = \c 12145. */
2093 #define RD_KAFKA_OFFSET_TAIL(CNT) (RD_KAFKA_OFFSET_TAIL_BASE - (CNT))
2094
2095 /**
2096 * @brief Start consuming messages for topic \p rkt and \p partition
2097 * at offset \p offset which may either be an absolute \c (0..N)
2098 * or one of the logical offsets:
2099 * - RD_KAFKA_OFFSET_BEGINNING
2100 * - RD_KAFKA_OFFSET_END
2101 * - RD_KAFKA_OFFSET_STORED
2102 * - RD_KAFKA_OFFSET_TAIL
2103 *
2104 * rdkafka will attempt to keep \c queued.min.messages (config property)
2105 * messages in the local queue by repeatedly fetching batches of messages
2106 * from the broker until the threshold is reached.
2107 *

```

```

2108 * The application shall use one of the `rd_kafka_consume*()` functions
2109 * to consume messages from the local queue, each kafka message being
2110 * represented as a `rd_kafka_message_t` object.
2111 *
2112 * `rd_kafka_consume_start()` must not be called multiple times for the same
2113 * topic and partition without stopping consumption first with
2114 * `rd_kafka_consume_stop()`.
2115 *
2116 * @returns 0 on success or -1 on error in which case errno is set accordingly:
2117 * - EBUSY - Conflicts with an existing or previous subscription
2118 *          (RD_KAFKA_RESP_ERR__CONFLICT)
2119 * - EINVAL - Invalid offset, or incomplete configuration (lacking group.id)
2120 *          (RD_KAFKA_RESP_ERR__INVALID_ARG)
2121 * - ESRCH - requested \p partition is invalid.
2122 *          (RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION)
2123 * - ENOENT - topic is unknown in the Kafka cluster.
2124 *          (RD_KAFKA_RESP_ERR__UNKNOWN_TOPIC)
2125 *
2126 * Use `rd_kafka_errno2err()` to convert sytem \c errno to `rd_kafka_resp_err_t`
2127 */
2128 RD_EXPORT
2129 int rd_kafka_consume_start(rd_kafka_topic_t *rkt, int32_t partition,
2130                           int64_t offset);
2131
2132 /**
2133  * @brief Same as rd_kafka_consume_start() but re-routes incoming messages to
2134  * the provided queue \p rkqu (which must have been previously allocated
2135  * with `rd_kafka_queue_new()`).
2136  *
2137  * The application must use one of the `rd_kafka_consume_*_queue()` functions
2138  * to receive fetched messages.
2139  *
2140  * `rd_kafka_consume_start_queue()` must not be called multiple times for the
2141  * same topic and partition without stopping consumption first with
2142  * `rd_kafka_consume_stop()`.
2143  * `rd_kafka_consume_start()` and `rd_kafka_consume_start_queue()` must not
2144  * be combined for the same topic and partition.
2145  */
2146 RD_EXPORT
2147 int rd_kafka_consume_start_queue(rd_kafka_topic_t *rkt, int32_t partition,
2148                                 int64_t offset, rd_kafka_queue_t *rkqu);
2149
2150 /**
2151  * @brief Stop consuming messages for topic \p rkt and \p partition, purging
2152  * all messages currently in the local queue.
2153  *
2154  * NOTE: To enforce synchronisation this call will block until the internal
2155  * fetcher has terminated and offsets are committed to configured
2156  * storage method.
2157  *
2158  * The application needs to stop all consumers before calling
2159  * `rd_kafka_destroy()` on the main object handle.
2160  *
2161  * @returns 0 on success or -1 on error (see `errno`).
2162  */
2163 RD_EXPORT
2164 int rd_kafka_consume_stop(rd_kafka_topic_t *rkt, int32_t partition);
2165
2166
2167
2168 /**

```

```

2169 * @brief Seek consumer for topic+partition to \p offset which is either an
2170 *      absolute or logical offset.
2171 *
2172 * If \p timeout_ms is not 0 the call will wait this long for the
2173 * seek to be performed. If the timeout is reached the internal state
2174 * will be unknown and this function returns `RD_KAFKA_RESP_ERR__TIMED_OUT`.
2175 * If \p timeout_ms is 0 it will initiate the seek but return
2176 * immediately without any error reporting (e.g., async).
2177 *
2178 * This call triggers a fetch queue barrier flush.
2179 *
2180 * @returns `RD_KAFKA_RESP_ERR__NO_ERROR` on success else an error code.
2181 */
2182 RD_EXPORT
2183 rd_kafka_resp_err_t rd_kafka_seek (rd_kafka_topic_t *rkt,
2184                                     int32_t partition,
2185                                     int64_t offset,
2186                                     int timeout_ms);
2187
2188
2189 /**
2190 * @brief Consume a single message from topic \p rkt and \p partition
2191 *
2192 * \p timeout_ms is maximum amount of time to wait for a message to be received.
2193 * Consumer must have been previously started with `rd_kafka_consume_start()`.
2194 *
2195 * @returns a message object on success or \c NULL on error.
2196 * The message object must be destroyed with `rd_kafka_message_destroy()`
2197 * when the application is done with it.
2198 *
2199 * Errors (when returning NULL):
2200 * - ETIMEDOUT - \p timeout_ms was reached with no new messages fetched.
2201 * - ENOENT    - \p rkt + \p partition is unknown.
2202 *              (no prior `rd_kafka_consume_start()` call)
2203 *
2204 * NOTE: The returned message's \c ..->err must be checked for errors.
2205 * NOTE: \c ..->err \c == \c RD_KAFKA_RESP_ERR__PARTITION_EOF signals that the
2206 * end of the partition has been reached, which should typically not be
2207 * considered an error. The application should handle this case
2208 * (e.g., ignore).
2209 *
2210 * @remark on_consume() interceptors may be called from this function prior to
2211 * passing message to application.
2212 */
2213 RD_EXPORT
2214 rd_kafka_message_t *rd_kafka_consume(rd_kafka_topic_t *rkt, int32_t partition,
2215                                       int timeout_ms);
2216
2217
2218
2219 /**
2220 * @brief Consume up to \p rkmessages_size from topic \p rkt and \p partition
2221 *      putting a pointer to each message in the application provided
2222 *      array \p rkmessages (of size \p rkmessages_size entries).
2223 *
2224 * `rd_kafka_consume_batch()` provides higher throughput performance
2225 * than `rd_kafka_consume()`.
2226 *
2227 * \p timeout_ms is the maximum amount of time to wait for all of
2228 * \p rkmessages_size messages to be put into \p rkmessages.
2229 * If no messages were available within the timeout period this function

```

```

2230 * returns 0 and \p rkmessages remains untouched.
2231 * This differs somewhat from `rd_kafka_consume()`.
2232 *
2233 * The message objects must be destroyed with `rd_kafka_message_destroy()`
2234 * when the application is done with it.
2235 *
2236 * @returns the number of rkmessages added in \p rkmessages,
2237 * or -1 on error (same error codes as for `rd_kafka_consume()`).
2238 *
2239 * @sa rd_kafka_consume()
2240 *
2241 * @remark on_consume() interceptors may be called from this function prior to
2242 * passing message to application.
2243 */
2244 RD_EXPORT
2245 ssize_t rd_kafka_consume_batch(rd_kafka_topic_t *rkt, int32_t partition,
2246                                int timeout_ms,
2247                                rd_kafka_message_t **rkmessages,
2248                                size_t rkmessages_size);
2249
2250
2251
2252 /**
2253 * @brief Consumes messages from topic \p rkt and \p partition, calling
2254 * the provided callback for each consumed message.
2255 *
2256 * `rd_kafka_consume_callback()` provides higher throughput performance
2257 * than both `rd_kafka_consume()` and `rd_kafka_consume_batch()`.
2258 *
2259 * \p timeout_ms is the maximum amount of time to wait for one or more messages
2260 * to arrive.
2261 *
2262 * The provided \p consume_cb function is called for each message,
2263 * the application \b MUST \b NOT call `rd_kafka_message_destroy()` on the
2264 * provided \p rkmessage.
2265 *
2266 * The \p opaque argument is passed to the 'consume_cb' as \p opaque.
2267 *
2268 * @returns the number of messages processed or -1 on error.
2269 *
2270 * @sa rd_kafka_consume()
2271 *
2272 * @remark on_consume() interceptors may be called from this function prior to
2273 * passing message to application.
2274 */
2275 RD_EXPORT
2276 int rd_kafka_consume_callback(rd_kafka_topic_t *rkt, int32_t partition,
2277                               int timeout_ms,
2278                               void (*consume_cb) (rd_kafka_message_t
2279 * r k m e s s a g e ,
2280                               void *opaque),
2281                               void *opaque);
2282
2283
2284 /**
2285 * @name Simple Consumer API (Legacy): Queue consumers
2286 * @{
2287 *
2288 * The following `...queue()` functions are analogue to the functions above
2289 * but reads messages from the provided queue \p rkqu instead.
2290 * \p rkqu must have been previously created with `rd_kafka_queue_new()`

```

```

2291 * and the topic consumer must have been started with
2292 * `rd_kafka_consume_start_queue()` utilising the the same queue.
2293 */
2294
2295 /**
2296 * @brief Consume from queue
2297 *
2298 * @sa rd_kafka_consume()
2299 */
2300 RD_EXPORT
2301 rd_kafka_message_t *rd_kafka_consume_queue(rd_kafka_queue_t *rkqu,
2302                                             int timeout_ms);
2303
2304 /**
2305 * @brief Consume batch of messages from queue
2306 *
2307 * @sa rd_kafka_consume_batch()
2308 */
2309 RD_EXPORT
2310 ssize_t rd_kafka_consume_batch_queue(rd_kafka_queue_t *rkqu,
2311                                       int timeout_ms,
2312                                       rd_kafka_message_t **rkmessages,
2313                                       size_t rkmessages_size);
2314
2315 /**
2316 * @brief Consume multiple messages from queue with callback
2317 *
2318 * @sa rd_kafka_consume_callback()
2319 */
2320 RD_EXPORT
2321 int rd_kafka_consume_callback_queue(rd_kafka_queue_t *rkqu,
2322                                     int timeout_ms,
2323                                     void (*consume_cb) (rd_kafka_message_t
2324                                                         * r k m e s s a g e ,
2325                                                         void *opaque),
2326                                     void *opaque);
2327
2328
2329 /**@}*/
2330
2331
2332
2333
2334 /**
2335 * @name Simple Consumer API (Legacy): Topic+partition offset store.
2336 * @{
2337 *
2338 * If \c auto.commit.enable is true the offset is stored automatically prior to
2339 * returning of the message(s) in each of the rd_kafka_consume*() functions
2340 * above.
2341 */
2342
2343
2344 /**
2345 * @brief Store offset \p offset for topic \p rkt partition \p partition.
2346 *
2347 * The offset will be committed (written) to the offset store according
2348 * to \c `auto.commit.interval.ms` or manual offset-less commit()
2349 *
2350 * @remark \c `enable.auto.offset.store` must be set to "false" when using this API.
2351 *

```

```
2352 * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code on error.
2353 */
2354 RD_EXPORT
2355 rd_kafka_resp_err_t rd_kafka_offset_store(rd_kafka_topic_t *rkt,
2356                                           int32_t partition, int64_t offset);
2357
2358
2359 /**
2360  * @brief Store offsets for one or more partitions.
2361  *
2362  * The offset will be committed (written) to the offset store according
2363  * to \c `auto.commit.interval.ms` or manual offset-less commit().
2364  *
2365  * Per-partition success/error status propagated through each partition's
2366  * \c .err field.
2367  *
2368  * @remark \c `enable.auto.offset.store` must be set to "false" when using this API.
2369  *
2370  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code if
2371  *          none of the offsets could be stored.
2372  */
2373 RD_EXPORT rd_kafka_resp_err_t
2374 rd_kafka_offsets_store(rd_kafka_t *rk,
2375                       rd_kafka_topic_partition_list_t *offsets);
2376 /**@}*/
2377
2378
2379
2380
2381 /**
2382  * @name KafkaConsumer (C)
2383  * @{
2384  * @brief High-Level KafkaConsumer C API
2385  *
2386  *
2387  *
2388  */
2389
2390 /**
2391  * @brief Subscribe to topic set using balanced consumer groups.
2392  *
2393  * Wildcard (regex) topics are supported by the Librdkafka assignor:
2394  * any topic name in the \p topics list that is prefixed with \c "\"" will
2395  * be regex-matched to the full list of topics in the cluster and matching
2396  * topics will be added to the subscription list.
2397  *
2398  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or
2399  *          RD_KAFKA_RESP_ERR__INVALID_ARG if list is empty, contains invalid
2400  *          topics or regexes.
2401  */
2402 RD_EXPORT rd_kafka_resp_err_t
2403 rd_kafka_subscribe (rd_kafka_t *rk,
2404                   const rd_kafka_topic_partition_list_t *topics);
2405
2406
2407 /**
2408  * @brief Unsubscribe from the current subscription set.
2409  */
2410 RD_EXPORT
2411 rd_kafka_resp_err_t rd_kafka_unsubscribe (rd_kafka_t *rk);
2412
```



```

2413
2414 /**
2415  * @brief Returns the current topic subscription
2416  *
2417  * @returns An error code on failure, otherwise \p topic is updated
2418  *          to point to a newly allocated topic list (possibly empty).
2419  *
2420  * @remark The application is responsible for calling
2421  *          rd_kafka_topic_partition_list_destroy on the returned list.
2422  */
2423 RD_EXPORT rd_kafka_resp_err_t
2424 rd_kafka_subscription (rd_kafka_t *rk,
2425                        rd_kafka_topic_partition_list_t **topics);
2426
2427
2428
2429 /**
2430  * @brief Poll the consumer for messages or events.
2431  *
2432  * Will block for at most \p timeout_ms milliseconds.
2433  *
2434  * @remark An application should make sure to call consumer_poll() at regular
2435  *          intervals, even if no messages are expected, to serve any
2436  *          queued callbacks waiting to be called. This is especially
2437  *          important when a rebalance_cb has been registered as it needs
2438  *          to be called and handled properly to synchronize internal
2439  *          consumer state.
2440  *
2441  * @returns A message object which is a proper message if \p ->err is
2442  *          RD_KAFKA_RESP_ERR_NO_ERROR, or an event or error for any other
2443  *          value.
2444  *
2445  * @remark on_consume() interceptors may be called from this function prior to
2446  *          passing message to application.
2447  *
2448  * @sa rd_kafka_message_t
2449  */
2450 RD_EXPORT
2451 rd_kafka_message_t *rd_kafka_consumer_poll (rd_kafka_t *rk, int timeout_ms);
2452
2453 /**
2454  * @brief Close down the KafkaConsumer.
2455  *
2456  * @remark This call will block until the consumer has revoked its assignment,
2457  *          calling the \c rebalance_cb if it is configured, committed offsets
2458  *          to broker, and left the consumer group.
2459  *          The maximum blocking time is roughly limited to session.timeout.ms.
2460  *
2461  * @returns An error code indicating if the consumer close was succesful
2462  *          or not.
2463  *
2464  * @remark The application still needs to call rd_kafka_destroy() after
2465  *          this call finishes to clean up the underlying handle resources.
2466  *
2467  */
2468 RD_EXPORT
2469 rd_kafka_resp_err_t rd_kafka_consumer_close (rd_kafka_t *rk);
2470
2471
2472
2473 /**

```

```
2474 * @brief Atomic assignment of partitions to consume.
2475 *
2476 * The new \p partitions will replace the existing assignment.
2477 *
2478 * When used from a rebalance callback the application shall pass the
2479 * partition list passed to the callback (or a copy of it) (even if the list
2480 * is empty) rather than NULL to maintain internal join state.
2481 *
2482 * A zero-length \p partitions will treat the partitions as a valid,
2483 * albeit empty, assignment, and maintain internal state, while a \c NULL
2484 * value for \p partitions will reset and clear the internal state.
2485 */
2486 RD_EXPORT rd_kafka_resp_err_t
2487 rd_kafka_assign (rd_kafka_t *rk,
2488                 const rd_kafka_topic_partition_list_t *partitions);
2489
2490 /**
2491 * @brief Returns the current partition assignment
2492 *
2493 * @returns An error code on failure, otherwise \p partitions is updated
2494 *          to point to a newly allocated partition list (possibly empty).
2495 *
2496 * @remark The application is responsible for calling
2497 *          rd_kafka_topic_partition_list_destroy on the returned list.
2498 */
2499 RD_EXPORT rd_kafka_resp_err_t
2500 rd_kafka_assignment (rd_kafka_t *rk,
2501                     rd_kafka_topic_partition_list_t **partitions);
2502
2503
2504
2505
2506 /**
2507 * @brief Commit offsets on broker for the provided list of partitions.
2508 *
2509 * \p offsets should contain \c topic, \c partition, \c offset and possibly
2510 * \c metadata.
2511 * If \p offsets is NULL the current partition assignment will be used instead.
2512 *
2513 * If \p async is false this operation will block until the broker offset commit
2514 * is done, returning the resulting success or error code.
2515 *
2516 * If a rd_kafka_conf_set_offset_commit_cb() offset commit callback has been
2517 * configured the callback will be enqueued for a future call to
2518 * rd_kafka_poll(), rd_kafka_consumer_poll() or similar.
2519 */
2520 RD_EXPORT rd_kafka_resp_err_t
2521 rd_kafka_commit (rd_kafka_t *rk, const rd_kafka_topic_partition_list_t *offsets,
2522                 int async);
2523
2524
2525 /**
2526 * @brief Commit message's offset on broker for the message's partition.
2527 *
2528 * @sa rd_kafka_commit
2529 */
2530 RD_EXPORT rd_kafka_resp_err_t
2531 rd_kafka_commit_message (rd_kafka_t *rk, const rd_kafka_message_t *rkmessage,
2532                          int async);
2533
2534
```

```

2535 /**
2536  * @brief Commit offsets on broker for the provided list of partitions.
2537  *
2538  * See rd_kafka_commit for \p offsets semantics.
2539  *
2540  * The result of the offset commit will be posted on the provided \p rkqu queue.
2541  *
2542  * If the application uses one of the poll APIs (rd_kafka_poll(),
2543  * rd_kafka_consumer_poll(), rd_kafka_queue_poll(), ..) to serve the queue
2544  * the \p cb callback is required. \p opaque is passed to the callback.
2545  *
2546  * If using the event API the callback is ignored and the offset commit result
2547  * will be returned as an RD_KAFKA_EVENT_COMMIT event. The \p opaque
2548  * value will be available with rd_kafka_event_opaque()
2549  *
2550  * If \p rkqu is NULL a temporary queue will be created and the callback will
2551  * be served by this call.
2552  *
2553  * @sa rd_kafka_commit()
2554  * @sa rd_kafka_conf_set_offset_commit_cb()
2555  */
2556 RD_EXPORT rd_kafka_resp_err_t
2557 rd_kafka_commit_queue (rd_kafka_t *rk,
2558                       const rd_kafka_topic_partition_list_t *offsets,
2559                       rd_kafka_queue_t *rkqu,
2560                       void (*cb) (rd_kafka_t *rk,
2561                                   rd_kafka_resp_err_t err,
2562                                   rd_kafka_topic_partition_list_t *offsets,
2563                                   void *opaque),
2564                       void *opaque);
2565
2566 /**
2567  * @brief Retrieve committed offsets for topics+partitions.
2568  *
2569  *
2570  * The \p offset field of each requested partition will either be set to
2571  * stored offset or to RD_KAFKA_OFFSET_INVALID in case there was no stored
2572  * offset for that partition.
2573  *
2574  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success in which case the
2575  *          \p offset or \p err field of each \p partitions' element is filled
2576  *          in with the stored offset, or a partition specific error.
2577  *          Else returns an error code.
2578  */
2579 RD_EXPORT rd_kafka_resp_err_t
2580 rd_kafka_committed (rd_kafka_t *rk,
2581                   rd_kafka_topic_partition_list_t *partitions,
2582                   int timeout_ms);
2583
2584
2585 /**
2586  * @brief Retrieve current positions (offsets) for topics+partitions.
2587  *
2588  *
2589  * The \p offset field of each requested partition will be set to the offset
2590  * of the last consumed message + 1, or RD_KAFKA_OFFSET_INVALID in case there was
2591  * no previous message.
2592  *
2593  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success in which case the
2594  *          \p offset or \p err field of each \p partitions' element is filled
2595  *          in with the stored offset, or a partition specific error.

```

```
2596 Else returns an error code.
2597 */
2598 RD_EXPORT rd_kafka_resp_err_t
2599 rd_kafka_position (rd_kafka_t *rk,
2600                   rd_kafka_topic_partition_list_t *partitions);
2601
2602
2603 /**@}*/
2604
2605
2606
2607 /**
2608  * @name Producer API
2609  * @{
2610  *
2611  *
2612  */
2613
2614
2615 /**
2616  * @brief Producer message flags
2617  */
2618 #define RD_KAFKA_MSG_F_FREE 0x1 /**< Delegate freeing of payload to rdkafka. */
2619 #define RD_KAFKA_MSG_F_COPY 0x2 /**< rdkafka will make a copy of the payload. */
2620 #define RD_KAFKA_MSG_F_BLOCK 0x4 /**< Block produce*() on message queue full.
2621     * WARNING: If a delivery report callback
2622     * is used the application MUST
2623     * call rd_kafka_poll() (or equiv.)
2624     * to make sure delivered messages
2625     * are drained from the internal
2626     * delivery report queue.
2627     * Failure to do so will result
2628     * in indefinitely blocking on
2629     * the produce() call when the
2630     * message queue is full.
2631     * /
2632
2633
2634
2635 /**
2636  * @brief Produce and send a single message to broker.
2637  *
2638  * \p rkt is the target topic which must have been previously created with
2639  * `rd_kafka_topic_new()`.
2640  *
2641  * `rd_kafka_produce()` is an asynch non-blocking API.
2642  *
2643  * \p partition is the target partition, either:
2644  * - RD_KAFKA_PARTITION_UA (unassigned) for
2645  *   automatic partitioning using the topic's partitioner function, or
2646  * - a fixed partition (0..N)
2647  *
2648  * \p msgflags is zero or more of the following flags OR:ed together:
2649  * RD_KAFKA_MSG_F_BLOCK - block \p produce*() call if
2650  *   \p queue.buffering.max.messages or
2651  *   \p queue.buffering.max.kbytes are exceeded.
2652  * Messages are considered in-queue from the point they
2653  * are accepted by produce() until their corresponding
2654  * delivery report callback/event returns.
2655  * It is thus a requirement to call
2656  * rd_kafka_poll() (or equiv.) from a separate
```

```

2657 *      thread when F_BLOCK is used.
2658 *      See WARNING on \c RD_KAFKA_MSG_F_BLOCK above.
2659 *
2660 *      RD_KAFKA_MSG_F_FREE - rdkafka will free(3) \p payload when it is done
2661 *      with it.
2662 *      RD_KAFKA_MSG_F_COPY - the \p payload data will be copied and the
2663 *      \p payload pointer will not be used by rdkafka
2664 *      after the call returns.
2665 *
2666 *      .._F_FREE and .._F_COPY are mutually exclusive.
2667 *
2668 *      If the function returns -1 and RD_KAFKA_MSG_F_FREE was specified, then
2669 *      the memory associated with the payload is still the caller's
2670 *      responsibility.
2671 *
2672 *      \p payload is the message payload of size \p len bytes.
2673 *
2674 *      \p key is an optional message key of size \p keylen bytes, if non-NULL it
2675 *      will be passed to the topic partitioner as well as be sent with the
2676 *      message to the broker and passed on to the consumer.
2677 *
2678 *      \p msg_opaque is an optional application-provided per-message opaque
2679 *      pointer that will be provided in the delivery report callback (`dr_cb`) for
2680 *      referencing this message.
2681 *
2682 *      @remark on_send() and on_acknowledgement() interceptors may be called
2683 *      from this function. on_acknowledgement() will only be called if the
2684 *      message fails partitioning.
2685 *
2686 *      @returns 0 on success or -1 on error in which case errno is set accordingly:
2687 *      - ENOBUFS - maximum number of outstanding messages has been reached:
2688 *          "queue.buffering.max.messages"
2689 *          (RD_KAFKA_RESP_ERR__QUEUE_FULL)
2690 *      - EMSGSIZE - message is larger than configured max size:
2691 *          "messages.max.bytes".
2692 *          (RD_KAFKA_RESP_ERR_MSG_SIZE_TOO_LARGE)
2693 *      - ESRCH - requested \p partition is unknown in the Kafka cluster.
2694 *          (RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION)
2695 *      - ENOENT - topic is unknown in the Kafka cluster.
2696 *          (RD_KAFKA_RESP_ERR__UNKNOWN_TOPIC)
2697 *
2698 *      @sa Use rd_kafka_errno2err() to convert `errno` to rdkafka error code.
2699 */
2700 RD_EXPORT
2701 int rd_kafka_produce(rd_kafka_topic_t *rkt, int32_t partition,
2702                     int msgflags,
2703                     void *payload, size_t len,
2704                     const void *key, size_t keylen,
2705                     void *msg_opaque);
2706
2707
2708 /**
2709 * @brief Produce and send a single message to broker.
2710 *
2711 * The message is defined by a va-arg list using \c rd_kafka_vtype_t
2712 * tag tuples which must be terminated with a single \c RD_KAFKA_V_END.
2713 *
2714 * @returns \c RD_KAFKA_RESP_ERR_NO_ERROR on success, else an error code.
2715 *
2716 * @sa rd_kafka_produce, RD_KAFKA_V_END
2717 */

```

```

2718 RD_EXPORT
2719 rd_kafka_resp_err_t rd_kafka_producev (rd_kafka_t *rk, ...);
2720
2721
2722 /**
2723  * @brief Produce multiple messages.
2724  *
2725  * If partition is RD_KAFKA_PARTITION_UA the configured partitioner will
2726  * be run for each message (slower), otherwise the messages will be enqueued
2727  * to the specified partition directly (faster).
2728  *
2729  * The messages are provided in the array \p rkmessages of count \p message_cnt
2730  * elements.
2731  * The \p partition and \p msgflags are used for all provided messages.
2732  *
2733  * Honoured \p rkmessages[] fields are:
2734  * - payload, len    Message payload and length
2735  * - key, key_len    Optional message key
2736  * - _private        Message opaque pointer (msg_opaque)
2737  * - err             Will be set according to success or failure.
2738  *                  Application only needs to check for errors if
2739  *                  return value != \p message_cnt.
2740  *
2741  * @returns the number of messages successfully enqueued for producing.
2742  */
2743 RD_EXPORT
2744 int rd_kafka_produce_batch(rd_kafka_topic_t *rkt, int32_t partition,
2745                           int msgflags,
2746                           rd_kafka_message_t *rkmessages, int message_cnt);
2747
2748
2749
2750
2751 /**
2752  * @brief Wait until all outstanding produce requests, et.al, are completed.
2753  *
2754  * This should typically be done prior to destroying a producer instance
2755  * to make sure all queued and in-flight produce requests are completed
2756  * before terminating.
2757  *
2758  * @remark This function will call rd_kafka_poll() and thus trigger callbacks.
2759  *
2760  * @returns RD_KAFKA_RESP_ERR__TIMED_OUT if \p timeout_ms was reached before all
2761  *          outstanding requests were completed, else RD_KAFKA_RESP_ERR_NO_ERROR
2762  */
2763 RD_EXPORT
2764 rd_kafka_resp_err_t rd_kafka_flush (rd_kafka_t *rk, int timeout_ms);
2765
2766 /** @} */
2767
2768
2769 /**
2770  * @name Metadata API
2771  * @{
2772  *
2773  *
2774  */
2775
2776
2777 /**
2778  * @brief Broker information

```



pyxllq

码龄12年

暂无认证

25	14万+	13万+	5万+	
原创	周排名	总排名	访问	等级

678	5	14	2	50
积分	粉丝	获赞	评论	收藏



私信



关注

```
2779 */
2780 typedef struct rd_kafka_metadata_broker {
2781     int32_t id;          /**< Broker Id */
2782     char *host;          /**< Broker hostname */
2783     int port;            /**< Broker listening port */
2784 } rd_kafka_metadata_broker_t;
2785
2786 /**
2787  * @brief Partition information
2788  */
2789 typedef struct rd_kafka_metadata_partition {
2790     int32_t id;          /**< Partition Id */
2791     rd_kafka_resp_err_t err; /**< Partition error reported by broker */
2792     int32_t leader;      /**< Leader broker */
2793     int replica_cnt;     /**< Number of brokers in \p replicas */
2794     int32_t *replicas;   /**< Replica brokers */
2795     int isr_cnt;         /**< Number of ISR brokers in \p isrs */
2796     int32_t *isrs;       /**< In-Sync-Replica brokers */
2797 } rd_kafka_metadata_partition_t;
2798
2799 /**
2800  * @brief Topic information
2801  */
2802 typedef struct rd_kafka_metadata_topic {
2803     char *topic;         /**< Topic name */
2804     int partition_cnt;   /**< Number of partitions in \p partitions*/
2805     struct rd_kafka_metadata_partition *partitions; /**< Partitions */
2806     rd_kafka_resp_err_t err; /**< Topic error reported by broker */
2807 } rd_kafka_metadata_topic_t;
2808
2809 /**
2810  * @brief Metadata container
2811  */
2812
2813 typedef struct rd_kafka_metadata {
2814     int broker_cnt;      /**< Number of brokers in \p brokers */
2815     struct rd_kafka_metadata_broker *brokers; /**< Brokers */
2816
2817     int topic_cnt;       /**< Number of topics in \p topics */
2818     struct rd_kafka_metadata_topic *topics; /**< Topics */
2819
2820     int32_t orig_broker_id; /**< Broker originating this metadata */
2821     char *orig_broker_name; /**< Name of originating broker */
2822 } rd_kafka_metadata_t;
2823
2824
2825 /**
2826  * @brief Request Metadata from broker.
2827  *
2828  * Parameters:
2829  * - \p all_topics if non-zero: request info about all topics in cluster,
2830  * if zero: only request info about locally known topics.
2831  * - \p only_rkt only request info about this topic
2832  * - \p metadatap pointer to hold metadata result.
2833  * The \p *metadatap pointer must be released
2834  * with rd_kafka_metadata_destroy().
2835  * - \p timeout_ms maximum response time before failing.
2836  *
2837  * Returns RD_KAFKA_RESP_ERR_NO_ERROR on success (in which case *metadatap)
2838  * will be set, else RD_KAFKA_RESP_ERR__TIMED_OUT on timeout or
2839  * other error code on error.
```

搜博文文章

热门文章

net.ipv4.tcp_timestamps引发的tcp syn无响应案

16493

udp使用connect建立连接与普通udp连接区别

8218

c语言使用kafka消息队列的实例

7506

优化--单台服务器支持百万级别tcp长连接

4806

vue项目：Cannot read property '_t' of null报错

3311

最新评论

优化--单台服务器支持百万级别tcp长连接_仰望星空_：内存需要多大

net.ipv4.tcp_timestamps引发的tcp syn无...

pan_shu: https://github.com/11061055/ph

p-7.3.0-ext-curl/wiki/linux-%E5%86%85%...

您愿意向朋友推荐“博客详情页”吗?

强烈不推荐

不推荐

一般般

推荐

强烈推荐

最新文章

网卡收包中断的上半部和下半部

记录相关库或者工具以及知识点

循环队列 代码实现 (FIFO)

2021年 4篇

2020年 9篇

2018年 23篇

2840 */

2841 RD_EXPORT

2842 rd_kafka_resp_err_t

2843 rd_kafka_metadata (rd_kafka_t *rk, int all_topics,

2844 rd_kafka_topic_t *only_rkt,

2845 const struct rd_kafka_metadata **metadatap,

2846 int timeout_ms);

2847

2848 /**

2849 * @brief Release metadata memory.

2850 */

2851 RD_EXPORT

2852 void rd_kafka_metadata_destroy(const struct rd_kafka_metadata *metadata);

2853

2854

2855 /**@*/

2856

2857

2858

2859 /**

2860 * @name Client group information

2861 * @{

2862 *

2863 *

2864 */

2865

2866

2867 /**

2868 * @brief Group member information

2869 *

2870 * For more information on \p member_metadata format, see

2871 * https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-Group

2872 *

2873 */

2874 struct rd_kafka_group_member_info {

2875 char *member_id; /**< Member id (generated by broker) */

2876 char *client_id; /**< Client's \p client.id */

2877 char *client_host; /**< Client's hostname */

2878 void *member_metadata; /**< Member metadata (binary),

2879 * format depends on \p protocol_type. */

pyxllq

关注

5

0

0

10

专栏目录

CSDN

博客

下载

课程

学习

PK

问答

社区

插件

认证

开源

c kafka

搜索

会员中心

足迹

动态

消息

创作抽大奖

分类专栏

算法

1篇

MySQL

2篇

DPDK

1篇

Vue开发

1篇

编程

11篇

python

2篇

mysql常用命令

2篇


```

2901 * @brief List of groups
2902 *
2903 * @sa rd_kafka_group_list_destroy() to release list memory.
2904 */
2905 struct rd_kafka_group_list {
2906     struct rd_kafka_group_info *groups;    /**< Groups */
2907     int group_cnt;                        /**< Group count */
2908 };
2909
2910
2911 /**
2912  * @brief List and describe client groups in cluster.
2913  *
2914  * \p group is an optional group name to describe, otherwise (\p NULL) all
2915  * groups are returned.
2916  *
2917  * \p timeout_ms is the (approximate) maximum time to wait for response
2918  * from brokers and must be a positive value.
2919  *
2920  * @returns \p RD_KAFKA_RESP_ERR_NO_ERROR on success and \p grplistp is
2921  *          updated to point to a newly allocated list of groups.
2922  *          Else returns an error code on failure and \p grplistp remains
2923  *          untouched.
2924  *
2925  * @sa Use rd_kafka_group_list_destroy() to release list memory.
2926  */
2927 RD_EXPORT
2928 rd_kafka_resp_err_t
2929 rd_kafka_list_groups (rd_kafka_t *rk, const char *group,
2930                      const struct rd_kafka_group_list **grplistp,
2931                      int timeout_ms);
2932
2933 /**
2934  * @brief Release list memory
2935  */
2936 RD_EXPORT
2937 void rd_kafka_group_list_destroy (const struct rd_kafka_group_list *grplist);
2938
2939
2940 /**@}*/
2941
2942
2943
2944 /**
2945  * @name Miscellaneous APIs
2946  * @{
2947  *
2948  */
2949
2950
2951 /**
2952  * @brief Adds one or more brokers to the kafka handle's list of initial
2953  *        bootstrap brokers.
2954  *
2955  * Additional brokers will be discovered automatically as soon as rdkafka
2956  * connects to a broker by querying the broker metadata.
2957  *
2958  * If a broker name resolves to multiple addresses (and possibly
2959  * address families) all will be used for connection attempts in
2960  * round-robin fashion.
2961  *

```

	归档	5篇
	分布式	2篇
	linux	7篇

```

2962 * \p brokerList is a ,-separated list of brokers in the format:
2963 * \c \<broker1\>,\<broker2\>,..
2964 * Where each broker is in either the host or URL based format:
2965 * \c \<host\>[:\<port\>]
2966 * \c \<proto\>://\<host\>[:port]
2967 * \c \<proto\> is either \c PLAINTEXT, \c SSL, \c SASL, \c SASL_PLAINTEXT
2968 * The two formats can be mixed but ultimately the value of the
2969 * `security.protocol` config property decides what brokers are allowed.
2970 *
2971 * Example:
2972 * brokerList = "broker1:10000,broker2"
2973 * brokerList = "SSL://broker3:9000,ssl://broker2"
2974 *
2975 * @returns the number of brokers successfully added.
2976 *
2977 * @remark Brokers may also be defined with the \c metadata.broker.list or
2978 * \c bootstrap.servers configuration property (preferred method).
2979 */
2980 RD_EXPORT
2981 int rd_kafka_brokers_add(rd_kafka_t *rk, const char *brokerlist);
2982
2983
2984
2985
2986 /**
2987 * @brief Set logger function.
2988 *
2989 * The default is to print to stderr, but a syslog logger is also available,
2990 * see rd_kafka_log(print/syslog) for the builtin alternatives.
2991 * Alternatively the application may provide its own logger callback.
2992 * Or pass 'func' as NULL to disable logging.
2993 *
2994 * @deprecated Use rd_kafka_conf_set_log_cb()
2995 *
2996 * @remark \p rk may be passed as NULL in the callback.
2997 */
2998 RD_EXPORT RD_DEPRECATED
2999 void rd_kafka_set_logger(rd_kafka_t *rk,
3000 void (*func) (const rd_kafka_t *rk, int level,
3001 const char *fac, const char *buf));
3002
3003
3004 /**
3005 * @brief Specifies the maximum logging level produced by
3006 * internal kafka logging and debugging.
3007 *
3008 * If the \p \"debug\" configuration property is set the level is automatically
3009 * adjusted to \c LOG_DEBUG (7).
3010 */
3011 RD_EXPORT
3012 void rd_kafka_set_log_level(rd_kafka_t *rk, int level);
3013
3014
3015 /**
3016 * @brief Builtin (default) log sink: print to stderr
3017 */
3018 RD_EXPORT
3019 void rd_kafka_log_print(const rd_kafka_t *rk, int level,
3020 const char *fac, const char *buf);
3021
3022

```

```

3023 /**
3024  * @brief Builtin log sink: print to syslog.
3025  */
3026 RD_EXPORT
3027 void rd_kafka_log_syslog(const rd_kafka_t *rk, int level,
3028                          const char *fac, const char *buf);
3029
3030
3031 /**
3032  * @brief Returns the current out queue length.
3033  *
3034  * The out queue contains messages waiting to be sent to, or acknowledged by,
3035  * the broker.
3036  *
3037  * An application should wait for this queue to reach zero before terminating
3038  * to make sure outstanding requests (such as offset commits) are fully
3039  * processed.
3040  *
3041  * @returns number of messages in the out queue.
3042  */
3043 RD_EXPORT
3044 int rd_kafka_outq_len(rd_kafka_t *rk);
3045
3046
3047
3048 /**
3049  * @brief Dumps rdkafka's internal state for handle \p rk to stream \p fp
3050  *
3051  * This is only useful for debugging rdkafka, showing state and statistics
3052  * for brokers, topics, partitions, etc.
3053  */
3054 RD_EXPORT
3055 void rd_kafka_dump(FILE *fp, rd_kafka_t *rk);
3056
3057
3058
3059 /**
3060  * @brief Retrieve the current number of threads in use by Librdkafka.
3061  *
3062  * Used by regression tests.
3063  */
3064 RD_EXPORT
3065 int rd_kafka_thread_cnt(void);
3066
3067
3068 /**
3069  * @brief Wait for all rd_kafka_t objects to be destroyed.
3070  *
3071  * Returns 0 if all kafka objects are now destroyed, or -1 if the
3072  * timeout was reached.
3073  *
3074  * @remark This function is deprecated.
3075  */
3076 RD_EXPORT
3077 int rd_kafka_wait_destroyed(int timeout_ms);
3078
3079
3080 /**
3081  * @brief Run Librdkafka's built-in unit-tests.
3082  *
3083  * @returns the number of failures, or 0 if all tests passed.

```

```

3084 */
3085 RD_EXPORT
3086 int rd_kafka_unittest (void);
3087
3088
3089 /**@}*/
3090
3091
3092
3093
3094 /**
3095  * @name Experimental APIs
3096  * @{
3097  */
3098
3099 /**
3100  * @brief Redirect the main (rd_kafka_poll()) queue to the KafkaConsumer's
3101  *        queue (rd_kafka_consumer_poll()).
3102  *
3103  * @warning It is not permitted to call rd_kafka_poll() after directing the
3104  *        main queue with rd_kafka_poll_set_consumer().
3105  */
3106 RD_EXPORT
3107 rd_kafka_resp_err_t rd_kafka_poll_set_consumer (rd_kafka_t *rk);
3108
3109
3110 /**@}*/
3111
3112 /**
3113  * @name Event interface
3114  *
3115  * @brief The event API provides an alternative pollable non-callback interface
3116  *        to librdkafka's message and event queues.
3117  *
3118  * @{
3119  */
3120
3121
3122 /**
3123  * @brief Event types
3124  */
3125 typedef int rd_kafka_event_type_t;
3126 #define RD_KAFKA_EVENT_NONE      0x0
3127 #define RD_KAFKA_EVENT_DR        0x1 /**< Producer Delivery report batch */
3128 #define RD_KAFKA_EVENT_FETCH     0x2 /**< Fetched message (consumer) */
3129 #define RD_KAFKA_EVENT_LOG       0x4 /**< Log message */
3130 #define RD_KAFKA_EVENT_ERROR     0x8 /**< Error */
3131 #define RD_KAFKA_EVENT_REBALANCE 0x10 /**< Group rebalance (consumer) */
3132 #define RD_KAFKA_EVENT_OFFSET_COMMIT 0x20 /**< Offset commit result */
3133 #define RD_KAFKA_EVENT_STATS     0x40 /**< Stats */
3134
3135
3136 typedef struct rd_kafka_op_s rd_kafka_event_t;
3137
3138
3139 /**
3140  * @returns the event type for the given event.
3141  *
3142  * @remark As a convenience it is okay to pass \p rkev as NULL in which case
3143  *        RD_KAFKA_EVENT_NONE is returned.
3144  */

```

```

3145 RD_EXPORT
3146 rd_kafka_event_type_t rd_kafka_event_type (const rd_kafka_event_t *rkev);
3147
3148 /**
3149  * @returns the event type's name for the given event.
3150  *
3151  * @remark As a convenience it is okay to pass \p rkev as NULL in which case
3152  *         the name for RD_KAFKA_EVENT_NONE is returned.
3153  */
3154 RD_EXPORT
3155 const char *rd_kafka_event_name (const rd_kafka_event_t *rkev);
3156
3157 /**
3158  * @brief Destroy an event.
3159  *
3160  *
3161  * @remark Any references to this event, such as extracted messages,
3162  *         will not be usable after this call.
3163  *
3164  * @remark As a convenience it is okay to pass \p rkev as NULL in which case
3165  *         no action is performed.
3166  */
3167 RD_EXPORT
3168 void rd_kafka_event_destroy (rd_kafka_event_t *rkev);
3169
3170 /**
3171  * @returns the next message from an event.
3172  *
3173  *
3174  * Call repeatedly until it returns NULL.
3175  *
3176  * Event types:
3177  * - RD_KAFKA_EVENT_FETCH (1 message)
3178  * - RD_KAFKA_EVENT_DR (>=1 message(s))
3179  *
3180  * @remark The returned message(s) MUST NOT be
3181  *         freed with rd_kafka_message_destroy().
3182  *
3183  * @remark on_consume() interceptor may be called
3184  *         from this function prior to passing message to application.
3185  */
3186 RD_EXPORT
3187 const rd_kafka_message_t *rd_kafka_event_message_next (rd_kafka_event_t *rkev);
3188
3189 /**
3190  * @brief Extracts \p size message(s) from the event into the
3191  *        pre-allocated array \p rkmessages.
3192  *
3193  *
3194  * Event types:
3195  * - RD_KAFKA_EVENT_FETCH (1 message)
3196  * - RD_KAFKA_EVENT_DR (>=1 message(s))
3197  *
3198  * @returns the number of messages extracted.
3199  *
3200  * @remark on_consume() interceptor may be called
3201  *         from this function prior to passing message to application.
3202  */
3203 RD_EXPORT
3204 size_t rd_kafka_event_message_array (rd_kafka_event_t *rkev,
3205                                     const rd_kafka_message_t **rkmessages,

```

```

3206         size_t size);
3207
3208
3209 /**
3210  * @returns the number of remaining messages in the event.
3211  *
3212  * Event types:
3213  * - RD_KAFKA_EVENT_FETCH (1 message)
3214  * - RD_KAFKA_EVENT_DR (>=1 message(s))
3215  */
3216 RD_EXPORT
3217 size_t rd_kafka_event_message_count (rd_kafka_event_t *rkev);
3218
3219
3220 /**
3221  * @returns the error code for the event.
3222  *
3223  * Event types:
3224  * - all
3225  */
3226 RD_EXPORT
3227 rd_kafka_resp_err_t rd_kafka_event_error (rd_kafka_event_t *rkev);
3228
3229
3230 /**
3231  * @returns the error string (if any).
3232  *      An application should check that rd_kafka_event_error() returns
3233  *      non-zero before calling this function.
3234  *
3235  * Event types:
3236  * - all
3237  */
3238 RD_EXPORT
3239 const char *rd_kafka_event_error_string (rd_kafka_event_t *rkev);
3240
3241
3242
3243 /**
3244  * @returns the user opaque (if any)
3245  *
3246  * Event types:
3247  * - RD_KAFKA_OFFSET_COMMIT
3248  */
3249 RD_EXPORT
3250 void *rd_kafka_event_opaque (rd_kafka_event_t *rkev);
3251
3252
3253 /**
3254  * @brief Extract log message from the event.
3255  *
3256  * Event types:
3257  * - RD_KAFKA_EVENT_LOG
3258  *
3259  * @returns 0 on success or -1 if unsupported event type.
3260  */
3261 RD_EXPORT
3262 int rd_kafka_event_log (rd_kafka_event_t *rkev,
3263                        const char **fac, const char **str, int *level);
3264
3265
3266 /**

```

```

3267 * @brief Extract stats from the event.
3268 *
3269 * Event types:
3270 * - RD_KAFKA_EVENT_STATS
3271 *
3272 * @returns stats json string.
3273 *
3274 * @remark the returned string will be freed automatically along with the event object
3275 *
3276 */
3277 RD_EXPORT
3278 const char *rd_kafka_event_stats (rd_kafka_event_t *rkev);
3279
3280
3281 /**
3282 * @returns the topic partition list from the event.
3283 *
3284 * @remark The list MUST NOT be freed with rd_kafka_topic_partition_list_destroy()
3285 *
3286 * Event types:
3287 * - RD_KAFKA_EVENT_REBALANCE
3288 * - RD_KAFKA_EVENT_OFFSET_COMMIT
3289 */
3290 RD_EXPORT rd_kafka_topic_partition_list_t *
3291 rd_kafka_event_topic_partition_list (rd_kafka_event_t *rkev);
3292
3293
3294 /**
3295 * @returns a newly allocated topic_partition container, if applicable for the event type,
3296 *         else NULL.
3297 *
3298 * @remark The returned pointer MUST be freed with rd_kafka_topic_partition_destroy().
3299 *
3300 * Event types:
3301 * RD_KAFKA_EVENT_ERROR (for partition level errors)
3302 */
3303 RD_EXPORT rd_kafka_topic_partition_t *
3304 rd_kafka_event_topic_partition (rd_kafka_event_t *rkev);
3305
3306
3307 /**
3308 * @brief Poll a queue for an event for max \p timeout_ms.
3309 *
3310 * @returns an event, or NULL.
3311 *
3312 * @remark Use rd_kafka_event_destroy() to free the event.
3313 */
3314 RD_EXPORT
3315 rd_kafka_event_t *rd_kafka_queue_poll (rd_kafka_queue_t *rkqu, int timeout_ms);
3316
3317 /**
3318 * @brief Poll a queue for events served through callbacks for max \p timeout_ms.
3319 *
3320 * @returns the number of events served.
3321 *
3322 * @remark This API must only be used for queues with callbacks registered
3323 *         for all expected event types. E.g., not a message queue.
3324 */
3325 RD_EXPORT
3326 int rd_kafka_queue_poll_callback (rd_kafka_queue_t *rkqu, int timeout_ms);
3327

```

```

3328
3329 /**@}*/
3330
3331
3332 /**
3333  * @name Plugin interface
3334  *
3335  * @brief A plugin interface that allows external runtime-loaded libraries
3336  * to integrate with a client instance without modifications to
3337  * the application code.
3338  *
3339  * Plugins are loaded when referenced through the `plugin.library.paths`
3340  * configuration property and operates on the \c rd_kafka_conf_t
3341  * object prior \c rd_kafka_t instance creation.
3342  *
3343  * @warning Plugins require the application to link librdkafka dynamically
3344  * and not statically. Failure to do so will lead to missing symbols
3345  * or finding symbols in another librdkafka library than the
3346  * application was linked with.
3347 */
3348
3349
3350 /**
3351  * @brief Plugin's configuration initializer method called each time the
3352  * library is referenced from configuration (even if previously loaded by
3353  * another client instance).
3354  *
3355  * @remark This method MUST be implemented by plugins and have the symbol name
3356  * \c conf_init
3357  *
3358  * @param conf Configuration set up to this point.
3359  * @param plug_opaque Plugin can set this pointer to a per-configuration
3360  * opaque pointer.
3361  * @param errstr String buffer of size \p errstr_size where plugin must write
3362  * a human readable error string in the case the initializer
3363  * fails (returns non-zero).
3364  *
3365  * @remark A plugin may add an on_conf_destroy() interceptor to clean up
3366  * plugin-specific resources created in the plugin's conf_init() method.
3367  *
3368  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code on error.
3369 */
3370 typedef rd_kafka_resp_err_t
3371 (rd_kafka_plugin_f_conf_init_t) (rd_kafka_conf_t *conf,
3372 void **plug_opaque,
3373 char *errstr, size_t errstr_size);
3374
3375 /**@}*/
3376
3377
3378
3379 /**
3380  * @name Interceptors
3381  *
3382  * @{
3383  *
3384  * @brief A callback interface that allows message interception for both
3385  * producer and consumer data pipelines.
3386  *
3387  * Except for the on_new(), on_conf_set(), on_conf_dup() and on_conf_destroy()
3388  * interceptors, interceptors are added to the

```



```
3389 * newly created rd_kafka_t client instance. These interceptors MUST only
3390 * be added from on_new() and MUST NOT be added after rd_kafka_new() returns.
3391 *
3392 * The on_new(), on_conf_set(), on_conf_dup() and on_conf_destroy() interceptors
3393 * are added to the configuration object which is later passed to
3394 * rd_kafka_new() where on_new() is called to allow addition of
3395 * other interceptors.
3396 *
3397 * Each interceptor reference consists of a display name (ic_name),
3398 * a callback function, and an application-specified opaque value that is
3399 * passed as-is to the callback.
3400 * The ic_name must be unique for the interceptor implementation and is used
3401 * to reject duplicate interceptor methods.
3402 *
3403 * Any number of interceptors can be added and they are called in the order
3404 * they were added, unless otherwise noted.
3405 * The list of registered interceptor methods are referred to as
3406 * interceptor chains.
3407 *
3408 * @remark Contrary to the Java client the Librdkafka interceptor interface
3409 * does not support message modification. Message mutability is
3410 * discouraged in the Java client and the combination of
3411 * serializers and headers cover most use-cases.
3412 *
3413 * @remark Interceptors are NOT copied to the new configuration on
3414 * rd_kafka_conf_dup() since it would be hard for interceptors to
3415 * track usage of the interceptor's opaque value.
3416 * An interceptor should rely on the plugin, which will be copied
3417 * in rd_kafka_conf_dup(), to set up the initial interceptors.
3418 * An interceptor should implement the on_conf_dup() method
3419 * to manually set up its internal configuration on the newly created
3420 * configuration object that is being copied-to based on the
3421 * interceptor-specific configuration properties.
3422 * conf_dup() should thus be treated the same as conf_init().
3423 *
3424 * @remark Interceptors are keyed by the interceptor type (on_..()), the
3425 * interceptor name (ic_name) and the interceptor method function.
3426 * Duplicates are not allowed and the .._add_on_..() method will
3427 * return RD_KAFKA_RESP_ERR__CONFLICT if attempting to add a duplicate
3428 * method.
3429 * The only exception is on_conf_destroy() which may be added multiple
3430 * times by the same interceptor to allow proper cleanup of
3431 * interceptor configuration state.
3432 */
3433
3434
3435 /**
3436 * @brief on_conf_set() is called from rd_kafka*_conf_set() in the order
3437 * the interceptors were added.
3438 *
3439 * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3440 * @param name The configuration property to set.
3441 * @param val The configuration value to set, or NULL for reverting to default
3442 * in which case the previous value should be freed.
3443 * @param errstr A human readable error string in case the interceptor fails.
3444 * @param errstr_size Maximum space (including \0) in \p errstr.
3445 *
3446 * @returns RD_KAFKA_CONF_RES_OK if the property was known and successfully
3447 * handled by the interceptor, RD_KAFKA_CONF_RES_INVALID if the
3448 * property was handled by the interceptor but the value was invalid,
3449 * or RD_KAFKA_CONF_RES_UNKNOWN if the interceptor did not handle
```

```

3450 *      this property, in which case the property is passed on on the
3451 *      interceptor in the chain, finally ending up at the built-in
3452 *      configuration handler.
3453 */
3454 typedef rd_kafka_conf_res_t
3455 (rd_kafka_interceptor_f_on_conf_set_t) (rd_kafka_conf_t *conf,
3456                                         const char *name, const char *val,
3457                                         char *errstr, size_t errstr_size,
3458                                         void *ic_opaque);
3459
3460
3461 /**
3462  * @brief on_conf_dup() is called from rd_kafka_conf_dup() in the
3463  *      order the interceptors were added and is used to let
3464  *      an interceptor re-register its conf interceptors with a new
3465  *      opaque value.
3466  *      The on_conf_dup() method is called prior to the configuration from
3467  *      \p old_conf being copied to \p new_conf.
3468  *
3469  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3470  *
3471  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or an error code
3472  *      on failure (which is logged but otherwise ignored).
3473  *
3474  * @remark No on_conf_* interceptors are copied to the new configuration
3475  *      object on rd_kafka_conf_dup().
3476  */
3477 typedef rd_kafka_resp_err_t
3478 (rd_kafka_interceptor_f_on_conf_dup_t) (rd_kafka_conf_t *new_conf,
3479                                         const rd_kafka_conf_t *old_conf,
3480                                         size_t filter_cnt,
3481                                         const char **filter,
3482                                         void *ic_opaque);
3483
3484
3485 /**
3486  * @brief on_conf_destroy() is called from rd_kafka_*_conf_destroy() in the
3487  *      order the interceptors were added.
3488  *
3489  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3490  */
3491 typedef rd_kafka_resp_err_t
3492 (rd_kafka_interceptor_f_on_conf_destroy_t) (void *ic_opaque);
3493
3494
3495 /**
3496  * @brief on_new() is called from rd_kafka_new() prior to returning
3497  *      the newly created client instance to the application.
3498  *
3499  * @param rk The client instance.
3500  * @param conf The client instance's final configuration.
3501  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3502  * @param errstr A human readable error string in case the interceptor fails.
3503  * @param errstr_size Maximum space (including \0) in \p errstr.
3504  *
3505  * @returns an error code on failure, the error is logged but otherwise ignored.
3506  *
3507  * @warning The \p rk client instance will not be fully set up when this
3508  *      interceptor is called and the interceptor MUST NOT call any
3509  *      other rk-specific APIs than rd_kafka_interceptor_add..().
3510  */

```

```

3511 */
3512 typedef rd_kafka_resp_err_t
3513 (rd_kafka_interceptor_f_on_new_t) (rd_kafka_t *rk, const rd_kafka_conf_t *conf,
3514                                     void *ic_opaque,
3515                                     char *errstr, size_t errstr_size);
3516
3517
3518 /**
3519  * @brief on_destroy() is called from rd_kafka_destroy() or (rd_kafka_new()
3520  *         if rd_kafka_new() fails during initialization).
3521  *
3522  * @param rk The client instance.
3523  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3524  */
3525 typedef rd_kafka_resp_err_t
3526 (rd_kafka_interceptor_f_on_destroy_t) (rd_kafka_t *rk, void *ic_opaque);
3527
3528
3529
3530
3531 /**
3532  * @brief on_send() is called from rd_kafka_produce*() (et.al) prior to
3533  *         the partitioner being called.
3534  *
3535  * @param rk The client instance.
3536  * @param rkmessage The message being produced. Immutable.
3537  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3538  *
3539  * @remark This interceptor is only used by producer instances.
3540  *
3541  * @remark The \p rkmessage object is NOT mutable and MUST NOT be modified
3542  *         by the interceptor.
3543  *
3544  * @remark If the partitioner fails or an unknown partition was specified,
3545  *         the on_acknowledgement() interceptor chain will be called from
3546  *         within the rd_kafka_produce*() call to maintain send-acknowledgement
3547  *         symmetry.
3548  *
3549  * @returns an error code on failure, the error is logged but otherwise ignored.
3550  */
3551 typedef rd_kafka_resp_err_t
3552 (rd_kafka_interceptor_f_on_send_t) (rd_kafka_t *rk,
3553                                     rd_kafka_message_t *rkmessage,
3554                                     void *ic_opaque);
3555
3556 /**
3557  * @brief on_acknowledgement() is called to inform interceptors that a message
3558  *         was succesfully delivered or permanently failed delivery.
3559  *         The interceptor chain is called from internal librdkafka background
3560  *         threads, or rd_kafka_produce*() if the partitioner failed.
3561  *
3562  * @param rk The client instance.
3563  * @param rkmessage The message being produced. Immutable.
3564  * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3565  *
3566  * @remark This interceptor is only used by producer instances.
3567  *
3568  * @remark The \p rkmessage object is NOT mutable and MUST NOT be modified
3569  *         by the interceptor.
3570  *
3571  * @warning The on_acknowledgement() method may be called from internal

```

```

3572 *      Librdkafka threads. An on_acknowledgement() interceptor MUST NOT
3573 *      call any Librdkafka API's associated with the \p rk, or perform
3574 *      any blocking or prolonged work.
3575 *
3576 * @returns an error code on failure, the error is logged but otherwise ignored.
3577 */
3578 typedef rd_kafka_resp_err_t
3579 (rd_kafka_interceptor_f_on_acknowledgement_t) (rd_kafka_t *rk,
3580                                               rd_kafka_message_t *rkmessage,
3581                                               void *ic_opaque);
3582
3583
3584 /**
3585 * @brief on_consume() is called just prior to passing the message to the
3586 *        application in rd_kafka_consumer_poll(), rd_kafka_consume*(),
3587 *        the event interface, etc.
3588 *
3589 * @param rk The client instance.
3590 * @param rkmessage The message being consumed. Immutable.
3591 * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3592 *
3593 * @remark This interceptor is only used by consumer instances.
3594 *
3595 * @remark The \p rkmessage object is NOT mutable and MUST NOT be modified
3596 *        by the interceptor.
3597 *
3598 * @returns an error code on failure, the error is logged but otherwise ignored.
3599 */
3600 typedef rd_kafka_resp_err_t
3601 (rd_kafka_interceptor_f_on_consume_t) (rd_kafka_t *rk,
3602                                       rd_kafka_message_t *rkmessage,
3603                                       void *ic_opaque);
3604
3605 /**
3606 * @brief on_commit() is called on completed or failed offset commit.
3607 *        It is called from internal Librdkafka threads.
3608 *
3609 * @param rk The client instance.
3610 * @param offsets List of topic+partition+offset+error that were committed.
3611 *        The error message of each partition should be checked for
3612 *        error.
3613 * @param ic_opaque The interceptor's opaque pointer specified in ..add..().
3614 *
3615 * @remark This interceptor is only used by consumer instances.
3616 *
3617 * @warning The on_commit() interceptor is called from internal
3618 *        Librdkafka threads. An on_commit() interceptor MUST NOT
3619 *        call any Librdkafka API's associated with the \p rk, or perform
3620 *        any blocking or prolonged work.
3621 *
3622 *
3623 * @returns an error code on failure, the error is logged but otherwise ignored.
3624 */
3625 typedef rd_kafka_resp_err_t
3626 (rd_kafka_interceptor_f_on_commit_t) (
3627     rd_kafka_t *rk,
3628     const rd_kafka_topic_partition_list_t *offsets,
3629     rd_kafka_resp_err_t err, void *ic_opaque);
3630
3631
3632

```

```
3633 /**
3634  * @brief Append an on_conf_set() interceptor.
3635  *
3636  * @param conf Configuration object.
3637  * @param ic_name Interceptor name, used in logging.
3638  * @param on_conf_set Function pointer.
3639  * @param ic_opaque Opaque value that will be passed to the function.
3640  *
3641  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3642  *          if an existing intercepted with the same \p ic_name and function
3643  *          has already been added to \p conf.
3644  */
3645 RD_EXPORT rd_kafka_resp_err_t
3646 rd_kafka_conf_interceptor_add_on_conf_set (
3647     rd_kafka_conf_t *conf, const char *ic_name,
3648     rd_kafka_interceptor_f_on_conf_set_t *on_conf_set,
3649     void *ic_opaque);
3650
3651
3652 /**
3653  * @brief Append an on_conf_dup() interceptor.
3654  *
3655  * @param conf Configuration object.
3656  * @param ic_name Interceptor name, used in logging.
3657  * @param on_conf_dup Function pointer.
3658  * @param ic_opaque Opaque value that will be passed to the function.
3659  *
3660  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3661  *          if an existing intercepted with the same \p ic_name and function
3662  *          has already been added to \p conf.
3663  */
3664 RD_EXPORT rd_kafka_resp_err_t
3665 rd_kafka_conf_interceptor_add_on_conf_dup (
3666     rd_kafka_conf_t *conf, const char *ic_name,
3667     rd_kafka_interceptor_f_on_conf_dup_t *on_conf_dup,
3668     void *ic_opaque);
3669
3670 /**
3671  * @brief Append an on_conf_destroy() interceptor.
3672  *
3673  * @param conf Configuration object.
3674  * @param ic_name Interceptor name, used in logging.
3675  * @param on_conf_destroy Function pointer.
3676  * @param ic_opaque Opaque value that will be passed to the function.
3677  *
3678  * @returns RD_KAFKA_RESP_ERR_NO_ERROR
3679  *
3680  * @remark Multiple on_conf_destroy() interceptors are allowed to be added
3681  *          to the same configuration object.
3682  */
3683 RD_EXPORT rd_kafka_resp_err_t
3684 rd_kafka_conf_interceptor_add_on_conf_destroy (
3685     rd_kafka_conf_t *conf, const char *ic_name,
3686     rd_kafka_interceptor_f_on_conf_destroy_t *on_conf_destroy,
3687     void *ic_opaque);
3688
3689
3690 /**
3691  * @brief Append an on_new() interceptor.
3692  *
3693  * @param conf Configuration object.
```

```

3694 * @param ic_name Interceptor name, used in logging.
3695 * @param on_send Function pointer.
3696 * @param ic_opaque Opaque value that will be passed to the function.
3697 *
3698 * @remark Since the on_new() interceptor is added to the configuration object
3699 * it may be copied by rd_kafka_conf_dup().
3700 * An interceptor implementation must thus be able to handle
3701 * the same interceptor,ic_opaque tuple to be used by multiple
3702 * client instances.
3703 *
3704 * @remark An interceptor plugin should check the return value to make sure it
3705 * has not already been added.
3706 *
3707 * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3708 * if an existing intercepted with the same \p ic_name and function
3709 * has already been added to \p conf.
3710 */
3711 RD_EXPORT rd_kafka_resp_err_t
3712 rd_kafka_conf_interceptor_add_on_new (
3713     rd_kafka_conf_t *conf, const char *ic_name,
3714     rd_kafka_interceptor_f_on_new_t *on_new,
3715     void *ic_opaque);
3716
3717
3718
3719 /**
3720 * @brief Append an on_destroy() interceptor.
3721 *
3722 * @param rk Client instance.
3723 * @param ic_name Interceptor name, used in logging.
3724 * @param on_destroy Function pointer.
3725 * @param ic_opaque Opaque value that will be passed to the function.
3726 *
3727 * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3728 * if an existing intercepted with the same \p ic_name and function
3729 * has already been added to \p conf.
3730 */
3731 RD_EXPORT rd_kafka_resp_err_t
3732 rd_kafka_interceptor_add_on_destroy (
3733     rd_kafka_t *rk, const char *ic_name,
3734     rd_kafka_interceptor_f_on_destroy_t *on_destroy,
3735     void *ic_opaque);
3736
3737
3738 /**
3739 * @brief Append an on_send() interceptor.
3740 *
3741 * @param rk Client instance.
3742 * @param ic_name Interceptor name, used in logging.
3743 * @param on_send Function pointer.
3744 * @param ic_opaque Opaque value that will be passed to the function.
3745 *
3746 * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3747 * if an existing intercepted with the same \p ic_name and function
3748 * has already been added to \p conf.
3749 */
3750 RD_EXPORT rd_kafka_resp_err_t
3751 rd_kafka_interceptor_add_on_send (
3752     rd_kafka_t *rk, const char *ic_name,
3753     rd_kafka_interceptor_f_on_send_t *on_send,
3754     void *ic_opaque);

```

```

3755
3756 /**
3757  * @brief Append an on_acknowledgement() interceptor.
3758  *
3759  * @param rk Client instance.
3760  * @param ic_name Interceptor name, used in Logging.
3761  * @param on_acknowledgement Function pointer.
3762  * @param ic_opaque Opaque value that will be passed to the function.
3763  *
3764  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3765  *          if an existing intercepted with the same \p ic_name and function
3766  *          has already been added to \p conf.
3767  */
3768 RD_EXPORT rd_kafka_resp_err_t
3769 rd_kafka_interceptor_add_on_acknowledgement (
3770     rd_kafka_t *rk, const char *ic_name,
3771     rd_kafka_interceptor_f_on_acknowledgement_t *on_acknowledgement,
3772     void *ic_opaque);
3773
3774
3775 /**
3776  * @brief Append an on_consume() interceptor.
3777  *
3778  * @param rk Client instance.
3779  * @param ic_name Interceptor name, used in Logging.
3780  * @param on_consume Function pointer.
3781  * @param ic_opaque Opaque value that will be passed to the function.
3782  *
3783  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3784  *          if an existing intercepted with the same \p ic_name and function
3785  *          has already been added to \p conf.
3786  */
3787 RD_EXPORT rd_kafka_resp_err_t
3788 rd_kafka_interceptor_add_on_consume (
3789     rd_kafka_t *rk, const char *ic_name,
3790     rd_kafka_interceptor_f_on_consume_t *on_consume,
3791     void *ic_opaque);
3792
3793
3794 /**
3795  * @brief Append an on_commit() interceptor.
3796  *
3797  * @param rk Client instance.
3798  * @param ic_name Interceptor name, used in Logging.
3799  * @param on_commit() Function pointer.
3800  * @param ic_opaque Opaque value that will be passed to the function.
3801  *
3802  * @returns RD_KAFKA_RESP_ERR_NO_ERROR on success or RD_KAFKA_RESP_ERR_CONFLICT
3803  *          if an existing intercepted with the same \p ic_name and function
3804  *          has already been added to \p conf.
3805  */
3806 RD_EXPORT rd_kafka_resp_err_t
3807 rd_kafka_interceptor_add_on_commit (
3808     rd_kafka_t *rk, const char *ic_name,
3809     rd_kafka_interceptor_f_on_commit_t *on_commit,
3810     void *ic_opaque);
3811
3812
3813
3814
3815 /**@}*/

```

```

3816
3817
3818 #ifdef __cplusplus
3819 }
3820 #endif

```

3. 收发消息接口等实现

```

1  /*
2   * Librdkafka - Apache Kafka C Library
3   *
4   * Copyright (c) 2012, Magnus Edenhill
5   * ALL rights reserved.
6   *
7   * Redistribution and use in source and binary forms, with or without
8   * modification, are permitted provided that the following conditions are met:
9   *
10  * 1. Redistributions of source code must retain the above copyright notice,
11  *   this list of conditions and the following disclaimer.
12  * 2. Redistributions in binary form must reproduce the above copyright notice,
13  *   this list of conditions and the following disclaimer in the documentation
14  *   and/or other materials provided with the distribution.
15  *
16  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
17  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
18  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
19  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
20  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
21  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
22  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
23  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
24  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
25  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
26  * POSSIBILITY OF SUCH DAMAGE.
27  */
28
29 /**
30  * Apache Kafka consumer & producer example programs
31  * using the Kafka driver from Librdkafka
32  * (https://github.com/edenhill/Librdkafka)
33  */
34
35 #include <ctype.h>
36 #include <signal.h>
37 #include <string.h>
38 #include <unistd.h>
39 #include <stdlib.h>
40 // #include <syslog.h>
41 #include <time.h>
42 #include <sys/time.h>
43 #include <getopt.h>
44
45 /* Typical include path would be <Librdkafka/rdkafka.h>, but this program
46  * is builtin from within the Librdkafka source tree and thus differs. */
47 #include "rdkafka.h" /* for Kafka driver */
48
49
50 static int run = 1;
51 static rd_kafka_t *rk;
52 static int exit_eof = 0;

```



```

53 static int quiet = 0;
54 static enum {
55     OUTPUT_HEXDUMP,
56     OUTPUT_RAW,
57 } output = OUTPUT_HEXDUMP;
58
59 static void stop (int sig) {
60     run = 0;
61     fclose(stdin); /* abort fgets() */
62 }
63
64
65 static void hexdump (FILE *fp, const char *name, const void *ptr, size_t len) {
66     const char *p = (const char *)ptr;
67     size_t of = 0;
68
69
70     if (name)
71         fprintf(fp, "%s hexdump (%zd bytes):\n", name, len);
72
73     for (of = 0 ; of < len ; of += 16) {
74         char hexen[16*3+1];
75         char charen[16+1];
76         int hof = 0;
77
78         int cof = 0;
79         int i;
80
81         for (i = of ; i < (int)of + 16 && i < (int)len ; i++) {
82             h o f  += sprintf(hexen+hof, "%02x ", p[i] & 0xff);
83             c o f  += sprintf(charen+cof, "%c",
84                             isprint((int)p[i]) ? p[i] : '.');
85         }
86         fprintf(fp, "%08zx: %-48s %-16s\n",
87             of, hexen, charen);
88     }
89 }
90
91 /**
92  * Kafka logger callback (optional)
93  */
94 static void logger (const rd_kafka_t *rk, int level,
95                     const char *fac, const char *buf) {
96     struct timeval tv;
97     gettimeofday(&tv, NULL);
98     fprintf(stderr, "%u.%03u RDKAFKA-%i-%s: %s: %s\n",
99         (int)tv.tv_sec, (int)(tv.tv_usec / 1000),
100     level, fac, rk ? rd_kafka_name(rk) : NULL, buf);
101 }
102
103 /**
104  * Message delivery report callback.
105  * Called once for each message.
106  * See rdkafka.h for more information.
107  */
108 static void msg_delivered (rd_kafka_t *rk,
109                             void *payload, size_t len,
110                             rd_kafka_resp_err_t error_code,
111                             void *opaque, void *msg_opaque) {
112
113     if (error_code)

```

```

114         fprintf(stderr, "%% Message delivery failed: %s\n",
115                 rd_kafka_err2str(error_code));
116         //else if (!quiet)
117         // fprintf(stderr, "%% Message delivered (%zd bytes): %.*s\n", Len,
118         //         (int)Len, (const char *)payload);
119     }
120
121     /**
122     * Message delivery report callback using the richer rd_kafka_message_t object.
123     */
124     static void msg_delivered2 (rd_kafka_t *rk,
125                                const rd_kafka_message_t *rkmessage, void *opaque) {
126         printf("del: %s: offset %"PRIu64"\n",
127                rd_kafka_err2str(rkmessage->err), rkmessage->offset);
128         if (rkmessage->err)
129             fprintf(stderr, "%% Message delivery failed: %s\n",
130                     rd_kafka_err2str(rkmessage->err));
131         else if (!quiet)
132             fprintf(stderr,
133                     "%% Message delivered (%zd bytes, offset %"PRIu64", "
134                     "partition %"PRIu32"): %.*s\n",
135                     rkmessage->len, rkmessage->offset,
136                     rkmessage->partition,
137                     (int)rkmessage->len, (const char *)rkmessage->payload);
138     }
139
140
141     static void msg_consume (rd_kafka_message_t *rkmessage,
142                             void *opaque) {
143         if (rkmessage->err) {
144             if (rkmessage->err == RD_KAFKA_RESP_ERR__PARTITION_EOF) {
145                 //fprintf(stderr,
146                 //    "%% Consumer reached end of %s [%"PRIu32"] "
147                 //    "message queue at offset %"PRIu64"\n",
148                 //    rd_kafka_topic_name(rkmessage->rkt),
149                 //    rkmessage->partition, rkmessage->offset);
150
151                 if (exit_eof)
152                     run = 0;
153
154                 return;
155             }
156
157             fprintf(stderr, "%% Consume error for topic \"%s\" [%"PRIu32"] "
158                     "offset %"PRIu64": %s\n",
159                     rd_kafka_topic_name(rkmessage->rkt),
160                     rkmessage->partition,
161                     rkmessage->offset,
162                     rd_kafka_message_errstr(rkmessage));
163
164             if (rkmessage->err == RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION ||
165                 rkmessage->err == RD_KAFKA_RESP_ERR__UNKNOWN_TOPIC)
166                 run = 0;
167
168             return;
169         }
170     }
171 #if 0
172     if (!quiet) {
173         rd_kafka_timestamp_type_t tstype;
174         int64_t timestamp;
175         fprintf(stdout, "%% Message (offset %"PRIu64", %zd bytes):\n",
176                 rkmessage->offset, rkmessage->len);

```

```

175
176     timestamp = rd_kafka_message_timestamp(rkmessage, &tstype);
177     if (tstype != RD_KAFKA_TIMESTAMP_NOT_AVAILABLE) {
178         const char *tsname = ">";
179         if (tstype == RD_KAFKA_TIMESTAMP_CREATE_TIME)
180             tsname = "create time";
181         else if (tstype == RD_KAFKA_TIMESTAMP_LOG_APPEND_TIME)
182             tsname = "log append time";
183
184         fprintf(stdout, "%% Message timestamp: %s %"PRIu64
185             " (%ds ago)\n",
186             tsname, timestamp,
187             !timestamp ? 0 :
188                 (int)time(NULL) - (int)(timestamp/1000));
189     }
190 }
191
192 if (rkmessage->key_len) {
193     if (output == OUTPUT_HEXDUMP)
194         hexdump(stdout, "Message Key",
195             rkmessage->key, rkmessage->key_len);
196     else
197         printf("Key: %.*s\n",
198             (int)rkmessage->key_len, (char *)rkmessage->key);
199 }
200
201 if (output == OUTPUT_HEXDUMP)
202     hexdump(stdout, "Message Payload",
203         rkmessage->payload, rkmessage->len);
204 else
205     printf("%.*s\n",
206         (int)rkmessage->len, (char *)rkmessage->payload);
207 #endif
208 #ifdef CONSUME /*process_balance_msg为自己的消息接收处理函数*/
209     process_balance_msg((sint8*)(rkmessage->payload), rkmessage->len);
210 #endif
211 }
212
213
214 static void metadata_print (const char *topic,
215                             const struct rd_kafka_metadata *metadata) {
216     int i, j, k;
217
218     printf("Metadata for %s (from broker %"PRIu32": %s):\n",
219         topic ? : "all topics",
220         metadata->orig_broker_id,
221         metadata->orig_broker_name);
222
223
224     /* Iterate brokers */
225     printf(" %i brokers:\n", metadata->broker_cnt);
226     for (i = 0 ; i < metadata->broker_cnt ; i++)
227         printf("   broker %"PRIu32" at %s:%i\n",
228             metadata->brokers[i].id,
229             metadata->brokers[i].host,
230             metadata->brokers[i].port);
231
232     /* Iterate topics */
233     printf(" %i topics:\n", metadata->topic_cnt);
234     for (i = 0 ; i < metadata->topic_cnt ; i++) {
235         const struct rd_kafka_metadata_topic *t = &metadata->topics[i];

```

```

236         printf("  topic \"%s\" with %i partitions:",
237                t->topic,
238                t->partition_cnt);
239         if (t->err) {
240             printf(" %s", rd_kafka_err2str(t->err));
241             if (t->err == RD_KAFKA_RESP_ERR_LEADER_NOT_AVAILABLE)
242                 printf(" (try again)");
243         }
244         printf("\n");
245
246         /* Iterate topic's partitions */
247         for (j = 0 ; j < t->partition_cnt ; j++) {
248             const struct rd_kafka_metadata_partition *p;
249             p = &t->partitions[j];
250             printf("    partition %"PRIu32", "
251                   "leader %"PRIu32", replicas: ",
252                   p->id, p->leader);
253
254             /* Iterate partition's replicas */
255             for (k = 0 ; k < p->replica_cnt ; k++)
256                 printf("%s%"PRIu32",
257                       k > 0 ? ",":":", p->replicas[k]);
258
259             /* Iterate partition's ISRs */
260             printf(", isrs: ");
261             for (k = 0 ; k < p->isr_cnt ; k++)
262                 printf("%s%"PRIu32",
263                       k > 0 ? ",":":", p->isrs[k]);
264
265             if (p->err)
266                 printf(", %s\n", rd_kafka_err2str(p->err));
267             else
268                 printf("\n");
269         }
270     }
271
272
273 static void sig_usr1 (int sig) {
274     rd_kafka_dump(stdout, rk);
275 }
276
277 int kafka_init(char mode, char* topic, int partion, char* brokers,
278               rd_kafka_topic_conf_t **topic_conf, rd_kafka_topic_t **rkt)
279 {
280     int opt;
281     char errstr[512];
282     rd_kafka_conf_t *conf;
283     int64_t start_offset = 0;
284     int report_offsets = 0;
285     int do_conf_dump = 0;
286     char tmp[16];
287
288     if (mode != 'C' && mode != 'P') {
289         fprintf(stderr, "mode err mode=%c\n", mode);
290         return -1;
291     }
292     if (NULL == topic || NULL == brokers || partion < 0){
293         fprintf(stderr, "input para err\n");
294         return -1;
295     }
296     /* Kafka configuration */

```

```

297 conf = rd_kafka_conf_new();
298
299 /* Set Logger */
300 rd_kafka_conf_set_log_cb(conf, logger);
301
302 /* Quick termination */
303 snprintf(tmp, sizeof(tmp), "%i", SIGIO);
304 rd_kafka_conf_set(conf, "internal.termination.signal", tmp, NULL, 0);
305
306 /* Topic configuration */
307 *topic_conf = rd_kafka_topic_conf_new();
308
309 signal(SIGINT, stop);
310 signal(SIGUSR1, sig_usr1);
311
312 if (mode == 'P') {
313     /*
314      * Producer
315      */
316     char buf[4096];
317     int sendcnt = 0;
318
319     /* Set up a message delivery report callback.
320      * It will be called once for each message, either on successful
321      * delivery to broker, or upon failure to deliver to broker. */
322
323     /* If offset reporting (-o report) is enabled, use the
324      * richer dr_msg_cb instead. */
325     if (report_offsets) {
326         rd_kafka_topic_conf_set(*topic_conf,
327                                 "produce.offset.report",
328                                 "true", errstr, sizeof(errstr));
329         rd_kafka_conf_set_dr_msg_cb(conf, msg_delivered2);
330     } else
331         rd_kafka_conf_set_dr_cb(conf, msg_delivered);
332
333     /* Create Kafka handle */
334     if (!(rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf,
335                             errstr, sizeof(errstr)))) {
336         fprintf(stderr,
337                 "%% Failed to create new producer: %s\n",
338                 errstr);
339         return -1;
340     }
341
342     /* Add brokers */
343     if (rd_kafka_brokers_add(rk, brokers) == 0) {
344         fprintf(stderr, "%% No valid brokers specified\n");
345         return -1;
346     }
347
348     /* Create topic */
349     *rkt = rd_kafka_topic_new(rk, topic, *topic_conf);
350     *topic_conf = NULL; /* Now owned by topic */
351 } else if (mode == 'C') {
352     /*
353      * Consumer
354      */
355
356     /* Create Kafka handle */
357     if (!(rk = rd_kafka_new(RD_KAFKA_CONSUMER, conf,

```

```

358         e r r s t r , sizeof(estr))) {
359             fprintf(stderr,
360                 "%s Failed to create new consumer: %s\n",
361                 e r r s t r ) ;
362             return -1;
363         }
364
365         /* Add brokers */
366         if (rd_kafka_brokers_add(rk, brokers) == 0) {
367             fprintf(stderr, "%s No valid brokers specified\n");
368             return -1;
369         }
370
371
372         /* Create topic */
373         * r k t = rd_kafka_topic_new(rk, topic, *topic_conf);
374         *topic_conf = NULL; /* Now owned by topic */
375
376         /* Start consuming */
377         if (rd_kafka_consume_start(*rkt, partion, -1/*start_offset*/) == -1){
378             rd_kafka_resp_err_t err = rd_kafka_last_error();
379             fprintf(stderr, "%s Failed to start consuming: %s\n",
380                 rd_kafka_err2str(err));
381             if (err == RD_KAFKA_RESP_ERR__INVALID_ARG)
382                 fprintf(stderr,
383                     "%s Broker based offset storage "
384                     "requires a group.id, "
385                     "add: -X group.id=yourGroup\n");
386             return -1;
387         }
388     }
389     return 0;
390 }
391
392 int kafka_send_msg(rd_kafka_topic_t *rkt, rd_kafka_topic_conf_t *topic_conf, char* data, int len)
393 {
394     /*
395      * Producer
396      */
397     LOG_ERROR("kafka_send_msg len=%d",len);
398     int sendcnt = 0;
399     int partition = 0;
400     if (NULL == rkt){
401         LOG_ERROR("rkt is null");
402         return -1;
403     }
404     if (NULL == data || 0 == len){
405         LOG_ERROR("send msg is null or length is 0");
406         return 0;
407     }
408
409     if (run) {
410
411         /* Send/Produce message. */
412         if (rd_kafka_produce(rkt, partition,
413             RD_KAFKA_MSG_F_COPY,
414             /* Payload and Length */
415             data, len,
416             /* Optional key and its length */
417             NULL, 0,
418             /* Message opaque, provided in

```



pyxlq

码龄12年

暂无认证

25 14万+ 13万+ 5万+
原创 周排名 总排名 访问 等级

678 5 14 2 50
积分 粉丝 获赞 评论 收藏



私信

关注

搜博文文章



热门文章

net.ipv4.tcp_timestamps引发的tcp syn无响
应案 16493

udp使用connect建立连接与普通udp连接区
别 8218

```
419         * delivery report callback as
420         * msg_opaque. */
421         NULL) == -1) {
422             fprintf(stderr,
423                 "%% Failed to produce to topic %s "
424                 "partition %i: %s\n",
425                 rd_kafka_topic_name(rkt), partition,
426                 rd_kafka_err2str(rd_kafka_last_error()));
427             /* Poll to handle delivery reports */
428             rd_kafka_poll(rk, 0);
429         }
430
431         //fprintf(stderr, "%% Sent %zd bytes to topic "
432         //            "%s partition %i\n",
433         //            len, rd_kafka_topic_name(rkt), partition);
434         sendcnt++;
435         /* Poll to handle delivery reports */
436         rd_kafka_poll(rk, 0);
437     }
438     else{
439         /* Poll to handle delivery reports */
440         rd_kafka_poll(rk, 0);
441
442         /* Wait for messages to be delivered */
443         while (run && rd_kafka_outq_len(rk) > 0)
444             rd_kafka_poll(rk, 100);
445
446         /* Destroy topic */
447         rd_kafka_topic_destroy(rkt);
448
449         /* Destroy the handle */
450         rd_kafka_destroy(rk);
451     }
452     return 0;
453 }
454
455 void* kafka_receive_msg(void *arg)
456 {
457     /*
458     * Consumer
459     */
460     if (NULL == arg)
461     {
462         LOG_ERROR("kafka_receive_msg rkt is null");
463         return NULL;
464     }
465     int partition = 0;
466     rd_kafka_topic_t* rkt = (rd_kafka_topic_t*)arg;
467     while (run) {
468         rd_kafka_message_t *rkmessage;
469         rd_kafka_resp_err_t err;
470
471         /* Poll for errors, etc. */
472         rd_kafka_poll(rk, 0);
473
474         /* Consume single message.
475          * See rdkafka_performance.c for high speed
476          * consuming of messages. */
477         rkmessage = rd_kafka_consume(rkt, partition, 1000);
478         if (!rkmessage) /* timeout */
```

c语言使用kafka消息队列的实例 7506

优化--单台服务器支持百万级别tcp长连接 4806

vue项目：Cannot read property '_t' of null 报错 3311

最新评论

优化--单台服务器支持百万级别tcp长连接
仰望星空: 内存需要多大

net.ipv4.tcp_timestamps引发的tcp syn无...
pan_shu: https://github.com/11061055/ph
p-7.3.0-ext-curl/wiki/linux-%E5%86%85%...

您愿意向朋友推荐“博客详情页”吗？

    

强烈不推荐 不推荐 一般般 推荐 强烈推荐

最新文章

网卡收包中断的上半部和下半部

记录相关库或者工具以及知识点

循环队列 代码实现（FIFO）

2021年 4篇 2020年 9篇

2018年 23篇

```
479         continue;
480
481         msg_consume(rkmessage, NULL);
482
483         /* Return message to rdkafka */
484         rd_kafka_message_destroy(rkmessage);
485     }
486
487     /* Stop consuming */
488     rd_kafka_consume_stop(rkt, partition);
489
490     while (rd_kafka_outq_len(rk) > 0)
491         rd_kafka_poll(rk, 10);
492
493     /* Destroy topic */
494     rd_kafka_topic_destroy(rkt);
495
496     /* Destroy handle */
497     rd_kafka_destroy(rk);
498 }
499
500 #if 0
501 int main (int argc, char **argv) {
502     rd_kafka_topic_t *rkt;
503     char *brokers = "localhost:9092";
504     char mode = 'C';
505     char *topic = NULL;
506     int partition = RD_KAFKA_PARTITION_UA;
507     int opt;
508     rd_kafka_topic_conf_t *topic_conf;
509     char errstr[512];
```



pyxlq

关注

👍 0



💬 0



★ 10



专栏目录

 CSDN 博客 下载 课程 学习 PK 问答 社区 插件 认证 开源

🔥 c kafka

🔍 搜索

```
515
516     if(0){
517         fprintf(stderr, "argc=%d\n", argc);
518         exit(1);
519     }
520     mode = 'C';
521     topic="msgtest";
522     partition=0;
523     brokers="169.0.1.198:9092";
524
525     if (kafka_init(mode,topic,partition,brokers,&topic_conf,&rkt) == -1){
526         exit(1);
527     }
528
529     if (NULL == rkt){
530         fprintf(stderr, "kafka_init rkt is null\n");
531         exit(1);
532     }
533     char data[4096];
534     kafka_receive_msg(rkt);
535     return 0;
536 }
537 #endif
```



举报



会员中心



足迹

动态

消息

创作独大奖

分类专栏

-  算法 1篇
-  MySQL 2篇
-  DPK 1篇
-  Vue开发 1篇
-  编程 11篇
-  python 2篇
-  mysql常用命令 2篇
-  归档 5篇
-  分布式 2篇
-  linux 7篇

上面的main是一个测试例子，消费侧的，即获取消息的例子，下面则是生产者例子，只需要把上面的main替换成下面的代码即可

```
1  int main (int argc, char **argv) {
2      rd_kafka_topic_t *rkt;
3      char *brokers = "localhost:9092";
4      char mode = 'C';
5      char *topic = NULL;
6      int partition = RD_KAFKA_PARTITION_UA;
7      int opt;
8      rd_kafka_topic_conf_t *topic_conf;
9      char errstr[512];
10     int64_t start_offset = 0;
11     int report_offsets = 0;
12     int do_conf_dump = 0;
13     char tmp[16];
14
15
16     if(0){
17         fprintf(stderr, "argc=%d\n", argc);
18         exit(1);
19     }
20     mode = 'P';
21     topic="msgtest";
22     partition=0;
23     brokers="169.0.1.198:9092";
24
25     if (kafka_init(mode,topic,partition,brokers,&topic_conf,&rkt) == -1){
26         exit(1);
27     }
28
29     if (NULL == rkt){
30         fprintf(stderr, "kafka_init rkt is null\n");
31         exit(1);
32     }
33     char data[4096];
34     while(1){
35         fgets(data, sizeof(data), stdin);
36         size_t len = strlen(data);
37         if (data[len-1] == '\n')
38             data[--len] = '\0';
39         kafka_send_msg(rkt,topic_conf,data,len);
40     }
41     return 0;
42 }
```

要测试还需下一个librdkafka.a文件，仅供参考，记录之

Kafka的简单介绍-刘宇

刘宇的博客 200

作者：刘宇。一、消息队列的介绍1、什么是消息队列2、消息队列的分类2.1、点对点(Peer-to-Peer)2.2、发布订阅(Pub/Sub)3、P2P和发布订阅的区别4...

kafka c语言编程,kafka C程序Producer堆积

weixin_30953869的博客 98

现象Mean message/sec : 3kPeak Messages/5Min: 130wMean Bytes in /sec: 600KB/s从Kafka Eagle页面观测，5分钟粒度消息数曲线在达到峰值130w条...

参与评论



请发表有价值的评论，博客评论欢迎灌水，良好的社区氛围需大家一起维护。



评论

抢沙发



pyxlq

码龄12年

 暂无认证

25
原创

14万+
周排名

13万+
总排名

5万+
访问


等级

678
积分

5
粉丝

14
获赞

2
评论

50
收藏





私信

关注

搜博文文章

热门文章

net.ipv4.tcp_timestamps引发的tcp syn无响
应案  16493

udp使用connect建立连接与普通udp连接区
别  8218

c语言使用kafka消息队列的实例  7506

优化--单台服务器支持百万级别tcp长连接
 4806

vue项目: Cannot read property '_t' of null
报错  3311

最新评论

优化-单台服务器支持百万级别tcp长连接
仰望星空: 内存需要多大

net.ipv4.tcp_timestamps引发的tcp syn...
pan_shu: <https://github.com/11061055/pan-7.3.0-ext-curl/wiki/linux-%E5%86%85%>...

浅谈开源Kafka和腾讯云cafka_老张的csdn的博客_cafka	4-2
1、为什么要设计kafka? 2、开源的kafka架构是怎么样的? 3、腾讯云的cafka架构是怎样的? 4、腾讯云的cafka架构解决了什么样的问题? 5、我对开源k...	
高性能消息队列 CKafka 核心原理介绍(上)_kware211的博...	3-15
CKafka是基础架构部开发的高性能、高可用消息中间件,其主要用于消息传输、网站活动追踪、运营监控、日志聚合、流式处理、事件追踪、提交日志等等...	
kafka的c/c++高性能客户端librdkafka简介	小麒麟的成长之路 2万+
Librdkafka是c语言实现的apachekafka的高性能客户端,为生产和使用kafka提供高效可靠的客户端,并且提供了c++接口 性能: Librdkafka 是一款专为...	
kafka2.2源码分析之幂等设计实现 (exactly once语义实现)	lzf的博客 307
概述 幂等性生产者能够实现单个producer对同一个<topic ,partition>的exactly once语义。 producer端 Producer 后台发送线程 Sender, 在 run() 方法中, ...	
CKafka如何助力腾讯课堂实现百万消息稳定互动?_QcloudC...	3-22
CKafka如何助力腾讯课堂实现百万消息稳定互动? 导语 疫情期间,为了保障国内学子的正常学习进度,腾讯课堂积极响应国家“停工不停学”的号召,紧急上线...	
高性能消息队列 CKafka 核心原理介绍(下)_DemonHunter2...	4-28
可以看出比v0版本的消息仅多了一个timestamp字段用于表明消息的生产(或添加到broker日志的时间),方便用户通过指定时间去获取消息。为了方便查看用...	
ckafka配置	DEAD_line9527的博客 191
一、注入生产者和消费者 1、创建ApplicationContext-mq-kafka.xml文件 2、将该配置文件在ApplicationContext中引用 <?xml version="1.0" encoding="UT...	
c语言开发kafka环境,c++ (11) 使用librdkafka库实现kafka的消费实例	weixin_35725559的博客 481
关于librdkafka库的介绍,可以参考kafka的c/c++高性能客户端librdkafka简介, 本文使用librdkafka库来进行kafka的简单的消费librdkafka在c语言的基础上...	
腾讯云CKafka上线DataHub,让数据流转更简便_腾讯云中间...	5-8
CKafka 推出 Datahub 数据中心接入服务模块,负责从业务数据源获取数据,并进行一些预处理工作,分发给离线/在线处理平台,构建数据源和数据处理系统间...	
Java小技巧:ckafka开源_m0_57285455的博客	3-12
jar --help usage: tabula [-a <AREA>] [-b <DIRECTORY>] [-c <COLUMNS>] [-f <FORMAT>] [-g] [-h] [-i] [-I] [Kafka的一些常用功能点 01-07 自己写的一些...	
【Flink】producer attempted to use a producer id which is not currently assigned to its transaction 最新发布	九师兄 370
1.场景1 1.1 概述 我是 cancel savepoint 之后重启不行,直接启动又没事 cancel savepoint 之后重启不行 指定了 从savepoint, -s 指定了 地址 就是要 恢复...	
kafka文档 (5) - - - 0.8.2 - C/C++客户端介绍 热门推荐	录事参军 4万+
文章来源地址: https://github.com/edenhill/librdkafka/blob/master/INTRODUCTION.md librdkafka 是Apache Kafka 客户端C语言的高性能实现,能够提供...	
...者如何读同一生产者消息_腾讯云消息队列(Ckafka)监...	5-5
消息队列 CKafka(CloudKafka)是基于开源 Apache Kafka 消息队列引擎,提供高吞吐性能、高可扩展性的消息队列服务。消息队列 CKafka 完美兼容 Apach...	
kafka c语言编程,史上最全、最详细的 kafka 学习笔记!	weixin_34774527的博客 421
一、为什么需要消息系统1.解耦:允许你独立的扩展或修改两边的处理过程,只要确保它们遵守同样的接口约束。2.冗余:消息队列把数据进行持久化直...	
c语言使用librdkafka库实现kafka的生产和消费实例	小麒麟的成长之路 3万+
关于librdkafka库的介绍,可以参考kafka的c/c++高性能客户端librdkafka简介, 本文使用librdkafka库来进行kafka的简单的生产、消费 一、producer librdk...	
想使用消息队列,先考虑下这些问题!	架构文摘 108
消息队列优势 消息队列 (Message Queue, 简称MQ), 其主要用于在复杂的微服务系统中进行消息通信, 它的优点可以大致整理成以下几点: 服务间解...	
java反序列化失败怎么处理_Apache Kafak如何处理消息反序列化失败等毒丸现象? ...	weixin_39605004的博客 713
在Kafka的场景下毒丸是:针对Kafka主题生产插入的记录,无论尝试多少次,消费者使用都会失败。因此,毒丸可以有不同的形式:记录已损坏(我自己从...	
高性能消息队列 CKafka 核心原理介绍 (上)	weixin_30776545的博客 164
欢迎大家前往腾讯云技术社区,获取更多腾讯海量技术实践干货哦~ 作者: 闫燕飞 1.背景 CKafka是基础架构部开发的高性能、高可用消息中间件,其主要...	
SpringSecurity版本不一致导致序列化失败	王大发的博客 5694
问题 今天将springcloud.sr2升级到g版后发现zuul去auth验证信息时失败, auth报以下错误 Handling error: SerializationException, Cannot deserialize; ne...	
Kafka 错误代码解释	Rao的博客 1万+
Kafka 原生API交互的Code代表 使用client 与 kafka 交互时, 会返回错误代码, 官方给出了, 代码表示的具体含义 官网地址 https://kafka.apache.org/prot...	
Kafka在消费者反序列化时出现问题	aw9165628的博客 1838
Kafka在消费者反序列化时出现问题 问题描述 今天在启动Kafka时, 出现了一些问题。Kafka启动后, 卡在了某一消费点, 报 Missing exception handling f...	
kafka api c语言,kafka的c/c++高性能客户端librdkafka简介	weixin_35871529的博客 187
Librdkafka是c语言实现的apachekafka的高性能客户端,为生产和使用kafka提供高效可靠的客户端,并且提供了c++接口性能: Librdkafka 是一款专为现...	

您愿意向朋友推荐“博客详情页”吗？



强烈不推荐 不推荐 一般般 推荐 强烈推荐

最新文章

网卡收包中断的上半部和下半部

记录相关库或者工具以及知识点

循环队列 代码实现（FIFO）

2021年 4篇

2020年 9篇

2018年 23篇

kafka系列之重试机制(15)

1万+

kafka 重试机制 重试源码 首先我们从KafkaProducer的send 方法入手我们看到其实客户端是不会直接发送数据的，而是将其加入到了在一个缓存里面去，然...

kafka c语言实现源码, Spring-Kafka源代码解析（消费者）

weixin_39554891的博客 92

根据配置文件一步步去查看：spring-consumer.xmlDefaultKafkaConsumerFactory就是根据参数配置生产consumer，略过。监听器类，实现了监听类一些...

Kafka生产者事务和幂等

mlhxn的专栏 2万+

1 生产者幂等性 1.1 引入 幂等性引入目的：生产者重复生产消息。生产者进行retry会产生重试时，会重复产生消息。有了幂等性之后，在进行retry重试时...

“相关推荐”对你有帮助么？



非常没帮助



没帮助



一般



有帮助



非常有帮助

©2022 CSDN 皮肤主题：大白 设计师：CSDN官方博客 返回首页

关于我们 招贤纳士 商务合作 寻求报道 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00



pyxlq

关注



0



0



10



专栏目录



举报

