Kafka客户端：c/c++

Kafka作为消息中间件，能够从客户端收消息和向客户端发消息。运行kafka的服务器和客户端之间的交互同样通过”报文“完成。

Kafka提供了不同语言的客户端收发报文的接口。

librdkafka.h提供了部分接口的文档，但rdkafka库包含所有的rd_kafka_xxxx函数

数据结构

rd_kafka_message_t

在生产者的回调函数中会用到，也作为consume函数的返回类型。

```
//DataFields
rd_kafka_resp_err_t     err //非零值代表出错
rd_kafka_topic_t *  rkt
int32_t     partition
void *  payload //Depends on the value of err : err==0: Message payload. err!=0:
Error string
size_t  len
void *  key
size_t  key_len
int64_t     offset
void *  _private //dr_msg_cb: msg_opaque from produce() call
```

rd_kafka_metadata_t

```
//Data Fields
int     broker_cnt
struct rd_kafka_metadata_broker *   brokers
int     topic_cnt
struct rd_kafka_metadata_topic *    topics
int32_t     orig_broker_id
char *  orig_broker_name
```

配合`rd_kafka_metadata`一起使用

创建环境

消费者角色必须创建一个`rd_kafka_t`结构作为”容器“。

全局配置和共享状态和这个容器相绑定。

```
//rd_kafka_t使用范例
rd_kafka_t *rk;
//配置全局配置conf
rd_kafka_new（RD_KAFKA_PRODUCER，conf）;
//配置topic并创建rkt

//生产消息

rd_kafka_topic_destroy(rkt);
rd_kafka_destroy(rk);
```

还需要实例化一个或多个主题（`rd_kafka_topic_t`），以用于产生或消费。可以对主题进行配置，比如是否记录偏移值到文件，比如是否等到所有的副本收到消息再给producer一个回应。

对`rd_kafka_t`和`rd_kafka_topic_t`的配置是可选的。不使用配置函数会导致librdkafka使用其默认值（已记录在*`CONFIGURATION.md`*）。

注意：应用程序可以创建多个`rd_kafka_t`对象，并且它们不共享任何状态。

注意：`rd_kafka_topic_t`对象的创建在`rd_kafka_t`之后。且`rd_kafka_topic_t`只能和创建它时用到的`rd_kafka_t`对象一起使用。

rd_kafka_conf_new

```
rd_kafka_conf_t* rd_kafka_conf_new(void )
```

不需要对应一个destroy

使用相应的set配置属性

```
rd_kafka_conf_t *conf = rd_kafka_conf_new();
```

Returns `rd_kafka_conf_res_t` to indicate success or failure. In case of failure `errstr` is updated to contain a human readable error string.

rd_kafka_conf_set

```
rd_kafka_conf_res_t rd_kafka_conf_set   (   rd_kafka_conf_t *   conf,const char
*   name,const char *   value,char *    errstr,size_t   errstr_size )
```

```
    char errstr[512];
    const char *brokers = "ckafka-6q2jz9en.ap-
guangzhou.ckafka.tencentcloudmq.com:6006";
    /*所有的set操作都长这样
        第一项为指向绑定的全局配置结构的指针
        第二项是配置的名字
        第三项是配置的值
        第四、五项报错信息的容器 长度                  */
    rd_kafka_conf_set(conf, "brokers", brokers, errstr, 512);
```

rd_kafka_brokers_add

效果与设置metadata.broker.list和bootstrap.servers一样

```
int rd_kafka_brokers_add(rd_kafka_t * rk,const char * brokerlist )
```

rd_kafka_new

消费者角色类型为RD_KAFKA_CONSUMER

生产者角色类型为RD_KAFKA_PRODUCER

```
rd_kafka_t* rd_kafka_new(rd_kafka_type_t type,rd_kafka_conf_t * conf,char *
errstr*,size_t errstr_size )
```

```
        /* Create Kafka handle */
        if (!(rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf,
                    errstr, sizeof(errstr)))) {
            fprintf(stderr,
                "%% Failed to create new producer: %s\n",
                errstr);
            exit(1);
        }
```

rd_kafka_conf_dup

```
rd_kafka_conf_t* rd_kafka_conf_dup (const rd_kafka_conf_t *conf)
```

Creates a copy/duplicate of configuration object conf

查看元数据

打印metadata示例

```
    const struct rd_kafka_metadata *metadata;
    err = rd_kafka_metadata(rk, rkt ? 0 : 1, rkt,
                            &metadata, 5000);
    if (err != RD_KAFKA_RESP_ERR_NO_ERROR) {
            fprintf(stderr,
                    "%% Failed to acquire metadata: %s\n",
                    rd_kafka_err2str(err));
            run = 0;
            break;
    }

    metadata_print(topic, metadata);

    rd_kafka_metadata_destroy(metadata);
```

rd_kafka_metadata

```
rd_kafka_resp_err_t rd_kafka_metadata ( rd_kafka_t * rk,
                                        int all_topics,rd_kafka_topic_t *
only_rkt,
                                        const struct rd_kafka_metadata **
metadatap,int timeout_ms )
```

all_topics:为0代表只获取only_rkt指定的topic的数据

结果由metadatap指向

*metadatap指针必须由rd_kafka_metadata_destroy()释放

```
/*一个打印metadata的示例*/
static void metadata_print (const char *topic,
                            const struct rd_kafka_metadata *metadata) {
        int i, j, k;
        int32_t controllerid;

        printf("Metadata for %s (from broker %"PRId32": %s):\n",
               topic ? : "all topics",
               metadata->orig_broker_id,
               metadata->orig_broker_name);
        //整个kafka集群只有一个controller 负责partition leader的选举 但是这个函数并没有
写在官方文档上
        controllerid = rd_kafka_controllerid(rk, 0);
        /*遍历broker */
        printf(" %i brokers:\n", metadata->broker_cnt);
        for (i = 0 ; i < metadata->broker_cnt ; i++)
                printf("  broker %"PRId32" at %s:%i%s\n",
                       metadata->brokers[i].id,
                       metadata->brokers[i].host,
                       metadata->brokers[i].port,
                       controllerid == metadata->brokers[i].id ?
                       " (controller)" : "");

        /*遍历topics */
        printf(" %i topics:\n", metadata->topic_cnt);
        for (i = 0 ; i < metadata->topic_cnt ; i++) {
                const struct rd_kafka_metadata_topic *t = &metadata->topics[i];
                printf("  topic \"%s\" with %i partitions:",
                       t->topic,
                       t->partition_cnt);
                if (t->err) {
                        printf(" %s", rd_kafka_err2str(t->err));
                        if (t->err == RD_KAFKA_RESP_ERR_LEADER_NOT_AVAILABLE)
                                printf(" (try again)");
                }
                printf("\n");

                /* Iterate topic's partitions */
                for (j = 0 ; j < t->partition_cnt ; j++) {
                        const struct rd_kafka_metadata_partition *p;
                        p = &t->partitions[j];
                        printf("    partition %"PRId32", "
                               "leader %"PRId32", replicas: ",
```

```
                                p->id, p->leader);

                        /* Iterate partition's replicas */
                        for (k = 0 ; k < p->replica_cnt ; k++)
                                printf("%s%"PRId32,
                                       k > 0 ? ",":"", p->replicas[k]);

                        /* Iterate partition's ISRs */
                        printf(", isrs: ");
                        for (k = 0 ; k < p->isr_cnt ; k++)
                                printf("%s%"PRId32,
                                       k > 0 ? ",":"", p->isrs[k]);
                        if (p->err)
                                printf(", %s\n", rd_kafka_err2str(p->err));
                        else
                                printf("\n");
                }
        }
}
```

rd_kafka_metadata_destroy()

生产者

rd_kafka_produce

```
if (rd_kafka_produce(rkt, partition,RD_KAFKA_MSG_F_COPY,
         /* Payload and length */
         buf, len,
         /* Optional key and its length */
         NULL, 0,
         /* Message opaque, provided in
            delivery report callback as msg_opaque. */
            NULL) == -1) {
            err = rd_kafka_last_error();
            }

                    if (err)
                    {fprintf(stderr,
                    "%% Failed to produce to topic %s "
                    "partition %i: %s\n",
                    rd_kafka_topic_name(rkt), partition,
                    rd_kafka_err2str(err));
```

rd_kafka_conf_set_dr_msg_cb

用于设置发送一条消息后的回调函数

```
void rd_kafka_conf_set_dr_msg_cb    (   rd_kafka_conf_t *   conf,
void(*)(rd_kafka_t *rk, const rd_kafka_message_t *rkmessage, void *opaque)
dr_msg_cb )
```

```
    /* 注册回调函数
         *对于每个消息，不管是否发送成功，都会执行 */
    /*
        对于每个使用rd_kafka_produce()发送的消息，执行回调函数，此时err已经被设置。
        必须在produce消息之后主动调用rd_kafka_poll()，执行回调函数
    */
    rd_kafka_conf_set_dr_msg_cb(conf, msg_delivered);
```

```
/*
将会使用 rd_kafka_conf_set_dr_msg_cb(conf, msg_delivered);
注册为回调函数,在生产一条消息之后调用
这个函数的内容：如果出错，打印错误码，如果没错，打印已发送的内容
*/
static void msg_delivered(rd_kafka_t *rk,
                          const rd_kafka_message_t *rkmessage, void *opaque)
{
    if (rkmessage->err)
        fprintf(stderr, "%% Message delivery failed: %s\n",
        /*一个被发送的消息 被设置了错误状态码*/
                rd_kafka_err2str(rkmessage->err));
    else if (!quiet)
        fprintf(stderr,
                "%% Message delivered (%zd bytes, offset %" PRId64 ", "
                "partition %" PRId32 "): %.*s\n",
                rkmessage->len, rkmessage->offset,
                rkmessage->partition,
                (int)rkmessage->len, (const char *)rkmessage->payload);
}
```

rd_kafka_poll

```
int rd_kafka_poll   (   rd_kafka_t * rk, int timeout_ms )
```

触发回调函数 数值是如果发生阻塞在回调函数中等待的时间

For non-blocking calls, provide 0 as `timeout_ms`. To wait indefinately for an event, provide -1.

Events:

- delivery report callbacks (if dr_cb/dr_msg_cb is configured) [producer]
- error callbacks (rd_kafka_conf_set_error_cb()) [all]
- stats callbacks (rd_kafka_conf_set_stats_cb()) [all]
- throttle callbacks (rd_kafka_conf_set_throttle_cb()) [all]

- Returns

  the number of events served.

rd_kafka_outq_len

```
int rd_kafka_outq_len   (   rd_kafka_t *    rk  )
```

返回out queue中消息的个数

out queue中有等待发送给broker或等待broker应答的消息

```
while (rd_kafka_outq_len(rk) > 0)
    rd_kafka_poll(rk, 10);
```

rd_kafka_topic_destroy

```
void rd_kafka_topic_destroy (rd_kafka_topic_t *rkt)
```

rd_kafka_destroy

```
void rd_kafka_destroy   (   rd_kafka_t *    rk  )
```

消费者

```
    /*
     * 一个消费者示例
     * 假设已经有rd_kafka_t*类型的全局变量rk
     */

    rd_kafka_conf_set(conf, "enable.partition.eof", "true",
                              NULL, 0);

    /* 使用配置好的环境创建消费者 */
    if (!(rk = rd_kafka_new(RD_KAFKA_CONSUMER, conf,
                errstr, sizeof(errstr)))) {
        fprintf(stderr,
            "%% Failed to create new consumer: %s\n",
            errstr);
        exit(1);
    }

    /* 可以手动添加broker，也可以使用rd_kafka_conf_set设置metadata.broker.list和
bootstrap.servers属性 */
    if (rd_kafka_brokers_add(rk, brokers) == 0) {
        fprintf(stderr, "%% No valid brokers specified\n");
        exit(1);
    }

    /* 创建一个topic */
    rkt = rd_kafka_topic_new(rk, topic, topic_conf);
            topic_conf = NULL; /* Now owned by topic */

    /* 开始消费 */
    if (rd_kafka_consume_start(rkt, partition, start_offset) == -1){
        rd_kafka_resp_err_t err = rd_kafka_last_error();
        fprintf(stderr, "%% Failed to start consuming: %s\n",
            rd_kafka_err2str(err));
                if (err == RD_KAFKA_RESP_ERR__INVALID_ARG)
                        fprintf(stderr,
                                "%% Broker based offset storage "
                                "requires a group.id, "
                                "add: -X group.id=yourGroup\n");
        exit(1);
    }
    //run是一个指示运行状态的变量
```

```
    while (run) {
        rd_kafka_message_t *rkmessage;
                    rd_kafka_resp_err_t err;

                    /* Poll for errors, etc. */
                    rd_kafka_poll(rk, 0);

        /* 消费单个消息*/
        rkmessage = rd_kafka_consume(rkt, partition, 1000);
        if (!rkmessage) /* timeout */
            continue;

        //自定义的消费函数 也可以使用回调

        /* 资源释放工作 */
        rd_kafka_message_destroy(rkmessage);
    }

    /* 结束消费 */
    rd_kafka_consume_stop(rkt, partition);

            while (rd_kafka_outq_len(rk) > 0)
                    rd_kafka_poll(rk, 10);

    /* 清理工作*/
    rd_kafka_topic_destroy(rkt);
    rd_kafka_destroy(rk);
```

rd_kafka_consume_start

！必须调用的内容

对同一个topic&partition 只调用一次 和rd_kafka_consume_stop配合使用

返回0则表示成功 -1表示出错

```
int rd_kafka_consume_start  (   rd_kafka_topic_t * rkt,int32_t partition,int64_t
offset )
```

```
        //使用范例
        if (rd_kafka_consume_start(rkt, partition, start_offset) == -1){
            rd_kafka_resp_err_t err = rd_kafka_last_error();
            fprintf(stderr, "%% Failed to start consuming: %s\n",
                rd_kafka_err2str(err));
                    if (err == RD_KAFKA_RESP_ERR__INVALID_ARG)
                            fprintf(stderr,
                                    "%% Broker based offset storage "
                                    "requires a group.id, "
                                    "add: -X group.id=yourGroup\n");
            exit(1);
        }
```

rd_kafka_consume

```
rd_kafka_message_t* rd_kafka_consume    (   rd_kafka_topic_t *  rkt,int32_t
partition,int   timeout_ms )
```

timeout_ms指明等待一条消息最多等待的时间

如果获取失败则不会返回该rd_kafka_message_t，而是返回NULL

一个消息获取成功并打印消息内容的范例如下所示

简略版

```
static void msg_consume (rd_kafka_message_t *rkmessage,
              void *opaque) {
    if (rkmessage->err) {
        if (rkmessage->err == RD_KAFKA_RESP_ERR__PARTITION_EOF) {
        //分区尾
        return;
        }
        //其他错误
        return;
    }

    if (rkmessage->key_len) {
        //打印key
    }
    //打印消息
}
```

详细版

```
//定义的打印格式
static  enum {
    OUTPUT_HEXDUMP,
    OUTPUT_RAW,
} output = OUTPUT_HEXDUMP;
```

在这个例子中我们将会使用 `hexdump` 函数 直接已十六进制打印二进制内容

```
static void msg_consume (rd_kafka_message_t *rkmessage,
              void *opaque) {
    if (rkmessage->err) {
        if (rkmessage->err == RD_KAFKA_RESP_ERR__PARTITION_EOF) {
            fprintf(stderr,
                "%% Consumer reached end of %s [%"PRId32"] "
                   "message queue at offset %"PRId64"\n",
                   rd_kafka_topic_name(rkmessage->rkt),
                   rkmessage->partition, rkmessage->offset);

            if (exit_eof)
                run = 0;

            return;
        }
```

```
        fprintf(stderr, "%% Consume error for topic \"%s\" [%"PRId32"] "
               "offset %"PRId64": %s\n",
               rd_kafka_topic_name(rkmessage->rkt),
               rkmessage->partition,
               rkmessage->offset,
               rd_kafka_message_errstr(rkmessage));

                if (rkmessage->err == RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION ||
                    rkmessage->err == RD_KAFKA_RESP_ERR__UNKNOWN_TOPIC)
                        run = 0;
        return;
    }

    if (!quiet) {
        rd_kafka_timestamp_type_t tstype;
        int64_t timestamp;
                rd_kafka_headers_t *hdrs;

        fprintf(stdout, "%% Message (offset %"PRId64", %zd bytes):\n",
            rkmessage->offset, rkmessage->len);

        timestamp = rd_kafka_message_timestamp(rkmessage, &tstype);
        if (tstype != RD_KAFKA_TIMESTAMP_NOT_AVAILABLE) {
            const char *tsname = "?";
            if (tstype == RD_KAFKA_TIMESTAMP_CREATE_TIME)
                tsname = "create time";
            else if (tstype == RD_KAFKA_TIMESTAMP_LOG_APPEND_TIME)
                tsname = "log append time";

            fprintf(stdout, "%% Message timestamp: %s %"PRId64
                " (%ds ago)\n",
                tsname, timestamp,
                !timestamp ? 0 :
                (int)time(NULL) - (int)(timestamp/1000));
        }

                if (!rd_kafka_message_headers(rkmessage, &hdrs)) {
                        size_t idx = 0;
                        const char *name;
                        const void *val;
                        size_t size;

                        fprintf(stdout, "%% Headers:");

                        while (!rd_kafka_header_get_all(hdrs, idx++,
                                                        &name, &val, &size)) {
                                fprintf(stdout, "%s%s=",
                                        idx == 1 ? " " : ", ", name);
                                if (val)
                                        fprintf(stdout, "\"%.*s\"",
                                                (int)size, (const char *)val);
                                else
                                        fprintf(stdout, "NULL");
                        }
                        fprintf(stdout, "\n");
                }
    }
```

```
    if (rkmessage->key_len) {
        if (output == OUTPUT_HEXDUMP)
            hexdump(stdout, "Message Key",
                rkmessage->key, rkmessage->key_len);
        else
            printf("Key: %.*s\n",
                   (int)rkmessage->key_len, (char *)rkmessage->key);
    }

    if (output == OUTPUT_HEXDUMP)
        hexdump(stdout, "Message Payload",
            rkmessage->payload, rkmessage->len);
    else
        printf("%.*s\n",
               (int)rkmessage->len, (char *)rkmessage->payload);
}
```

rd_kafka_message_destroy

消费完一条消息之后，释放资源

```
void rd_kafka_message_destroy(rd_kafka_message_t *rkmessage);
```

rd_kafka_conf_set_consume_cb

```
void rd_kafka_conf_set_consume_cb (rd_kafka_conf_t *conf, void(*consume_cb)
(rd_kafka_message_t *rkmessage, void *opaque))
```

消费一条消息的回调函数

配合rd_kafka_consumer_poll()一起使用

rd_kafka_consumer_poll

```
rd_kafka_message_t* rd_kafka_consumer_poll  (   rd_kafka_t *    rk,int
timeout_ms )
```

将最多阻塞 `timeout_ms` 毫秒

rd_kafka_outq_len

```
int rd_kafka_outq_len   (   rd_kafka_t *    rk  )
```

返回out queue中消息的个数

out queue中有等待发送给broker或等待broker应答的消息

由于消费者可能会有提交偏移量等发送消息的行为，故在调用destroy之前使用

```
while (rd_kafka_outq_len(rk) > 0)
    rd_kafka_poll(rk, 10);
```

rd_kafka_consume_stop

使用rd_kafka_destroy()之前使用其停止所有消费

```
int rd_kafka_consume_stop   (   rd_kafka_topic_t *  rkt,int32_t     partition )
```

统计数据

rdkafka提供了一个参数statistics.interval.ms来设置统计数据的间隔

在消息发送回调函数和统计回调函数中更新对事件和发送数据量的统计

由 **void rd_kafka_conf_set_stats_cb**设置统计回调函数，统计回调函数即（函数名是自定义的）

```
static int stats_cb (rd_kafka_t *rk, char *json, size_t json_len,
             void *opaque) {
    //  操作
    return 0;
}
```

json字符串是由kafka服务器提供的