c语言使用librdkafka库实现kafka的生产和消费实例(转)

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

librdkafka进行kafka生产操作的大致步骤如下：

1、创建kafka配置

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

2、配置kafka各项参数

```
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)
```

3、设置发送回调函数

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

4、创建producer实例

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

5、实例化topic

```
rd_kafka_topic_t *rd_kafka_topic_new (rd_kafka_t *rk, const char *topic, rd_kafka_topic_conf_t *conf)
```

6、异步调用将消息发送到指定的topic

```
int rd_kafka_produce (rd_kafka_topic_t *rkt, int32_t partition,
              int msgflags,
              void *payload, size_t len,
              const void *key, size_t keylen,
              void *msg_opaque)
```

7、阻塞等待消息发送完成

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

8、等待完成producer请求完成

```
rd_kafka_resp_err_t rd_kafka_flush (rd_kafka_t *rk, int timeout_ms)
```

9、销毁topic

```
void rd_kafka_topic_destroy (rd_kafka_topic_t *app_rkt)
```

10、销毁producer实例

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

完整代码如下my_producer.c:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

#include "../src/rdkafka.h"

static int run = 1;

static void stop(int sig){
    run = 0;
    fclose(stdin);
}

/*
    每条消息调用一次该回调函数，说明消息是传递成功(rkmessage->err == RD_KAFKA_RESP_ERR_NO_ERROR)
    还是传递失败(rkmessage->err != RD_KAFKA_RESP_ERR_NO_ERROR)
    该回调函数由rd_kafka_poll()触发，在应用程序的线程上执行
 */
static void dr_msg_cb(rd_kafka_t *rk,
                      const rd_kafka_message_t *rkmessage, void *opaque){
        if(rkmessage->err)
            fprintf(stderr, "%% Message delivery failed: %s\n",
                    rd_kafka_err2str(rkmessage->err));
        else
            fprintf(stderr,
                        "%% Message delivered (%zd bytes, "
                        "partition %"PRId32")\n",
                        rkmessage->len, rkmessage->partition);
        /* rkmessage被librdkafka自动销毁*/
}

int main(int argc, char **argv){
    rd_kafka_t *rk;            /*Producer instance handle*/
    rd_kafka_topic_t *rkt;     /*topic对象*/
    rd_kafka_conf_t *conf;     /*临时配置对象*/
    char errstr[512];
    char buf[512];
    const char *brokers;
    const char *topic;

    if(argc != 3){
        fprintf(stderr, "%% Usage: %s <broker> <topic>\n", argv[0]);
        return 1;
    }

    brokers = argv[1];
    topic = argv[2];

    /* 创建一个kafka配置占位 */
    conf = rd_kafka_conf_new();

    /*创建broker集群*/
    if (rd_kafka_conf_set(conf, "bootstrap.servers", brokers, errstr,
                sizeof(errstr)) != RD_KAFKA_CONF_OK){
        fprintf(stderr, "%s\n", errstr);
        return 1;
    }

    /*设置发送报告回调函数，rd_kafka_produce()接收的每条消息都会调用一次该回调函数
     *应用程序需要定期调用rd_kafka_poll()来服务排队的发送报告回调函数*/
    rd_kafka_conf_set_dr_msg_cb(conf, dr_msg_cb);

    /*创建producer实例
      rd_kafka_new()获取conf对象的所有权,应用程序在此调用之后不得再次引用它*/
    rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(errstr));
    if(!rk){
        fprintf(stderr, "%% Failed to create new producer:%s\n", errstr);
        return 1;
    }

    /*实例化一个或多个topics(`rd_kafka_topic_t`)来提供生产或消费，topic
    对象保存topic特定的配置，并在内部填充所有可用分区和leader brokers，*/
    rkt = rd_kafka_topic_new(rk, topic, NULL);
    if (!rkt){
        fprintf(stderr, "%% Failed to create topic object: %s\n",
                rd_kafka_err2str(rd_kafka_last_error()));
        rd_kafka_destroy(rk);
        return 1;
    }

    /*用于中断的信号*/
    signal(SIGINT, stop);

```

```
    fprintf(stderr,
                "%% Type some text and hit enter to produce message\n"
                "%% Or just hit enter to only serve delivery reports\n"
                "%% Press Ctrl-C or Ctrl-D to exit\n");

    while(run && fgets(buf, sizeof(buf), stdin)){
        size_t len = strlen(buf);

        if(buf[len-1] == '\n')
            buf[--len] = '\0';

        if(len == 0){
            /*轮询用于事件的kafka handle.
            事件将导致应用程序提供的回调函数被调用
            第二个参数是最大阻塞时间. 如果设为0, 将会是非阻塞的调用*/
            rd_kafka_poll(rk, 0);
            continue;
        }

    retry:
        /*Send/Produce message.
          这是一个异步调用, 在成功的情况下, 只会将消息排入内部producer队列.
          对broker的实际传递尝试由后台线程处理. 之前注册的传递回调函数(dr_msg_cb)
          用于在消息传递成功或失败时向应用程序发回信号*/
        if (rd_kafka_produce(
                    /* Topic object */
                    rkt,
                    /*使用内置的分区来选择分区*/
                    RD_KAFKA_PARTITION_UA,
                    /*生成payload的副本*/
                    RD_KAFKA_MSG_F_COPY,
                    /*消息体和长度*/
                    buf, len,
                    /*可选键及其长度*/
                    NULL, 0,
                    NULL) == -1){
            fprintf(stderr,
                "%% Failed to produce to topic %s: %s\n",
                rd_kafka_topic_name(rkt),
                rd_kafka_err2str(rd_kafka_last_error()));

            if (rd_kafka_last_error() == RD_KAFKA_RESP_ERR__QUEUE_FULL){
                /*如果内部队列满, 等待消息传输完成并retry,
                内部队列表示要发送的消息和已发送或失败的消息,
                内部队列受限于queue.buffering.max.messages配置项*/
                rd_kafka_poll(rk, 1000);
                goto retry;
            }
        }else{
            fprintf(stderr, "%% Enqueued message (%zd bytes) for topic %s\n",
                len, rd_kafka_topic_name(rkt));
        }

        /*producer应用程序应不断地通过以频繁的间隔调用rd_kafka_poll()来为
        传送报告队列提供服务. 在没有生成消息以确定先前生成的消息已发送了其
        发送报告回调函数(和其他注册过的回调函数)期间, 要确保rd_kafka_poll()
        仍然被调用*/
        rd_kafka_poll(rk, 0);
    }

    fprintf(stderr, "%% Flushing final message.. \n");
    /*rd_kafka_flush是rd_kafka_poll()的抽象化,
    等待所有未完成的produce请求完成, 通常在销毁producer实例前完成
    以确保所有排列中和正在传输的produce请求在销毁前完成*/
    rd_kafka_flush(rk, 10*1000);

    /* Destroy topic object */
    rd_kafka_topic_destroy(rkt);

    /* Destroy the producer instance */
    rd_kafka_destroy(rk);

    return 0;
}
```

二、consumer

librdkafka进行kafka消费操作的大致步骤如下:

1、创建kafka配置

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

2、创建kafka topic的配置

```
rd_kafka_topic_conf_t *rd_kafka_topic_conf_new (void)
```

3、配置kafka各项参数

```
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)
```

4、配置kafka topic各项参数

```
rd_kafka_conf_res_t rd_kafka_topic_conf_set (rd_kafka_topic_conf_t *conf,
                        const char *name,
                        const char *value,
                        char *errstr, size_t errstr_size)
```

5、创建consumer实例

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

6、为consumer实例添加brokerlist

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

7、开启consumer订阅

```
rd_kafka_subscribe (rd_kafka_t *rk, const rd_kafka_topic_partition_list_t *topics)
```

8、轮询消息或事件, 并调用回调函数

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

9、关闭consumer实例

```
rd_kafka_resp_err_t rd_kafka_consumer_close (rd_kafka_t *rk)
```

10、释放topic list资源

```
rd_kafka_topic_partition_list_destroy (rd_kafka_topic_partition_list_t *rktparlist)
```

11、销毁consumer实例

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

12、等待consumer对象的销毁

```
int rd_kafka_wait_destroyed (int timeout_ms)
```

完整代码如下my_consumer.c

```
#include <string.h>
#include <stdlib.h>
#include <syslog.h>
#include <signal.h>
#include <error.h>
#include <getopt.h>

#include "../src/rdkafka.h"

static int run = 1;
//`rd_kafka_t`自带一个可选的配置API. 如果没有调用API. Librdkafka将会使用CONFIGURATION.md中的默认配置.
static rd_kafka_t *rk;
static rd_kafka_topic_partition_list_t *topics;

static void stop (int sig) {
  if (!run)
    exit(1);
  run = 0;
  fclose(stdin); /* abort fgets() */
```

```
}

static void sig_usr1 (int sig) {
  rd_kafka_dump(stdout, rk);
}

/**
 * 处理并打印已消费的消息
 */
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);

      return;
    }

    if (rkmessage->rkt)
          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));
    else
          fprintf(stderr, "%% Consumer error: %s: %s\n",
                  rd_kafka_err2str(rkmessage->err),
                  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;
  }

  fprintf(stdout, "%% Message (topic %s [%"PRId32"], "
                  "offset %"PRId64", %zd bytes):\n",
                  rd_kafka_topic_name(rkmessage->rkt),
                  rkmessage->partition,
    rkmessage->offset, rkmessage->len);

  if (rkmessage->key_len) {
    printf("Key: %.*s\n",
           (int)rkmessage->key_len, (char *)rkmessage->key);
  }

  printf("%.*s\n",
         (int)rkmessage->len, (char *)rkmessage->payload);


}

/*
 init all configuration of kafka
 */
int initKafka(char *brokers, char *group,char *topic){
  rd_kafka_conf_t *conf;
  rd_kafka_topic_conf_t *topic_conf;
  rd_kafka_resp_err_t err;
  char tmp[16];
  char errstr[512];

  /* Kafka configuration */
  conf = rd_kafka_conf_new();

  //quick termination
  snprintf(tmp, sizeof(tmp), "%i", SIGIO);
  rd_kafka_conf_set(conf, "internal.termination.signal", tmp, NULL, 0);

  //topic configuration
  topic_conf = rd_kafka_topic_conf_new();

  /* Consumer groups require a group id */
  if (!group)
          group = "rdkafka_consumer_example";
  if (rd_kafka_conf_set(conf, "group.id", group,
                        errstr, sizeof(errstr)) !=
      RD_KAFKA_CONF_OK) {
          fprintf(stderr, "%% %s\n", errstr);
          return -1;
  }

  /* Consumer groups always use broker based offset storage */
  if (rd_kafka_topic_conf_set(topic_conf, "offset.store.method",
                              "broker",
                              errstr, sizeof(errstr)) !=
      RD_KAFKA_CONF_OK) {
          fprintf(stderr, "%% %s\n", errstr);
          return -1;
  }

  /* Set default topic config for pattern-matched topics. */
  rd_kafka_conf_set_default_topic_conf(conf, topic_conf);

  //实例化一个顶级对象rd_kafka_t作为基础容器，提供全局配置和共享状态
  rk = rd_kafka_new(RD_KAFKA_CONSUMER, conf, errstr, sizeof(errstr));
  if(!rk){
    fprintf(stderr, "%% Failed to create new consumer:%s\n", errstr);
    return -1;
  }

  //librdkafka需要至少一个brokers的初始化list
  if (rd_kafka_brokers_add(rk, brokers) == 0){
    fprintf(stderr, "%% No valid brokers specified\n");
    return -1;
  }

  //重定向 rd_kafka_poll()队列到consumer_poll()队列
  rd_kafka_poll_set_consumer(rk);

  //创建一个Topic+Partition的存储空间(list/vector)
  topics = rd_kafka_topic_partition_list_new(1);
  //把Topic+Partition加入list
  rd_kafka_topic_partition_list_add(topics, topic, -1);
  //开启consumer订阅，匹配的topic将被添加到订阅列表中
  if((err = rd_kafka_subscribe(rk, topics))){
      fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
      return -1;
  }

  return 1;
}

int main(int argc, char **argv){
  char *brokers = "localhost:9092";
  char *group = NULL;
  char *topic = NULL;

  int opt;
  rd_kafka_resp_err_t err;

  while ((opt = getopt(argc, argv, "g:b:t:qd:eX:As:DO")) != -1){
    switch (opt) {
      case 'b':
        brokers = optarg;
        break;
      case 'g':
        group = optarg;
        break;
      case 't':
        topic = optarg;
        break;
      default:
        break;
    }
  }

  signal(SIGINT, stop);
  signal(SIGUSR1, sig_usr1);

  if(!initKafka(brokers, group, topic)){
    fprintf(stderr, "kafka server initialize error\n");
  }else{
    while(run){
      rd_kafka_message_t *rkmessage;
      /*轮询消费者的消息或事件. 最多阻塞timeout_ms
        -应用程序应该定期调用consumer_poll(), 即使没有预期的消息, 以服务
        所有排队等待的回调函数. 当注册过rebalance_cb 该操作尤为重要.
        因为它需要被正确地调用和处理以同步内部消费者状态 */
      rkmessage = rd_kafka_consumer_poll(rk, 1000);
      if(rkmessage){
        msg_consume(rkmessage, NULL);
        /*释放rkmessage的资源, 并把所有权还给rdkafka*/
        rd_kafka_message_destroy(rkmessage);
      }
    }
  }

done:
    /*此调用将会阻塞, 直到consumer撤销其分配, 调用rebalance_cb(如果已设置),
    commit offset到broker,并离开consumer group
    最大阻塞时间被设置为session.timeout.ms
    */
    err = rd_kafka_consumer_close(rk);
    if(err){
      fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
    }else{
      fprintf(stderr, "%% Consumer closed\n");
    }

    //释放topics list使用的所有资源和它自己
    rd_kafka_topic_partition_list_destroy(topics);

    //destroy kafka handle
    rd_kafka_destroy(rk);
```

```
    run = 5;
    //等待所有rd_kafka_t对象销毁,所有kafka对象被销毁,返回0,超时返回-1
    while(run-- > 0 && rd_kafka_wait_destroyed(1000) == -1){
      printf("Waiting for librdkafka to decommission\n");
    }
    if(run <= 0){
      //dump rdkafka内部状态到stdout流
      rd_kafka_dump(stdout, rk);
    }

    return 0;
}
```

在linux下编译producer和consumer的代码：

```
gcc my_producer.c -o my_producer  -lrdkafka -lz -lpthread -lrt
gcc my_consumer.c -o my_consumer  -lrdkafka -lz -lpthread -lrt
```

在运行my_producer或my_consumer时可能会报错"error while loading shared libraries xxx.so", 此时需要在/etc/ld.so.conf中加入xxx.so所在的目录

在本地启动一个简单的kafka服务，设置broker集群为localhost：9092并创建一个叫“test_topic”的topic
启动方式可参考 kafka0.8.2集群的环境搭建并实现基本的生产消费

启动consumer:

```
lijinqi@ubuntu:/librdkafka$ ./my_consumer -b localhost:9092 -t test_topic
% Consumer reached end of test_topic [0] message queue at offset 0
```


启动producer，并发送一条数据“hello world”：

```
lijinqi@ubuntu:/librdkafka$ ./my_producer localhost:9092 test_topic
% Type some text and hit enter to produce message
% Or just hit enter to only serve delivery reports
% Press Ctrl-C or Ctrl-D to exit
hello world
% Enqueued message (11 bytes) for topic test_topic
```


consumer处成功收到producer发送的“hello world”：

```
lijinqi@ubuntu:/librdkafka$ ./my_consumer -b localhost:9092 -t test_topic
% Consumer reached end of test_topic [0] message queue at offset 0
% Message (topic test_topic [0], offset 0, 11 bytes):
hello world
% Consumer reached end of test_topic [0] message queue at offset 1
```


http://orchome.com/5

https://github.com/edenhill/librdkafka

https://github.com/mfontanini/cppkafka

https://github.com/zengyuxing007/kafka_test_cpp

aliyun活动 https://www.aliyun.com/acts/limit-buy?userCode=re2o7acl

分类: 大型系统架构

Dufe王彬
粉丝 - 52 关注 - 3
+加关注

« 上一篇： Nginx-Lua模块的执行顺序(转)
» 下一篇： Flume环境搭建_五种案例(转)

posted on 2018-01-04 11:55 Dufe王彬 阅读(16534) 评论(0) 编辑 收藏 举报