Librdkafka对kafka topic的封装

扫帚的影子
弱水三千 只取一瓢

赞同 2

2 人赞同了该文章

- 上一节我们讲了librdkakfa对topic-partition的封装, 任何一个partition都必须要属于一下topic;
- 我们这节就来分析一上librdkafka对topic的封装

rd_kafka_itopic_s

- 所在文件: src/rdkafka_topic.h
- 这里还有一个类型 `rd_kafka_topic_t` ,定义: `typedef struct rd_kafka_topic_s rd_kafka_topic_t;` ,这是个空定义没有现实, 其实就是 `rd_kafka_itopic_s` , 这个类型主要是面向librdkafka的使用者,sdk里称作 `app topic` , 它有自己的引用计数. 在librdkafka内部使用 `rd_kafka_itopic` , 它也有自己的引用计数, 有点罗嗦啊~
- 定义:

```
struct rd_kafka_itopic_s {
        // 定义成tailq的元素
    TAILQ_ENTRY(rd_kafka_itopic_s) rkt_link;

        //  引用计数
    rd_refcnt_t        rkt_refcnt;

    rwlock_t           rkt_lock;

        // 所属topic的名字
    rd_kafkap_str_t   *rkt_topic;

        // 表示一个未assigned的partition
    shptr_rd_kafka_toppar_t  *rkt_ua;  /* unassigned partition */

        // 拥有的partition列表
    shptr_rd_kafka_toppar_t **rkt_p;
    int32_t            rkt_partition_cnt;

        // 期望操作的partition, 但还没有从broker获取到其信息的partition列表
        rd_list_t          rkt_desp;              /* Desired partitions
                                                   * that are not yet seen
                                                   * in the cluster. */
        // 最近一次更新metadata的时间
    rd_ts_t            rkt_ts_metadata; /* Timestamp of last metadata
                         * update for this topic. */

        mtx_t              rkt_app_lock;    /* Protects rkt_app_* */
,
       // 在application层一个rd_kafka_itopic_s对外表现为一个 rd_kafka_topic_t类型的对象,
        rd_kafka_topic_t *rkt_app_rkt;      /* A shared topic pointer
                                             * to be used for callbacks
                                             * to the application. */
    int               rkt_app_refcnt;   /* Number of active rkt's new()ed
                         * by application. */
        //  topic的三种状态:未知, 已存在, 不存在
    enum {
        RD_KAFKA_TOPIC_S_UNKNOWN,   /* No cluster information yet */
        RD_KAFKA_TOPIC_S_EXISTS,    /* Topic exists in cluster */
        RD_KAFKA_TOPIC_S_NOTEXISTS, /* Topic is not known in cluster */
    } rkt_state;

        int               rkt_flags;
#define RD_KAFKA_TOPIC_F_LEADER_UNAVAIL   0x1 /* Leader lost/unavailable
                                               * for at least one partition. */
        // 所属的rd_kafka_t
    rd_kafka_t       *rkt_rk;

        shptr_rd_kafka_itopic_t *rkt_shptr_app; /* Application's topic_new() */

    rd_kafka_topic_conf_t rkt_conf;
};
```

- 创建一个 `rd_kafka_itopic_s` 对象 `rd_kafka_topic_new0` , 这是一个内部调用函数

```
shptr_rd_kafka_itopic_t *rd_kafka_topic_new0 (rd_kafka_t *rk,
                                              const char *topic,
                                              rd_kafka_topic_conf_t *conf,
                                              int *existing,
                                              int do_lock) {
    rd_kafka_itopic_t *rkt;
        shptr_rd_kafka_itopic_t *s_rkt;
        const struct rd_kafka_metadata_cache_entry *rkmce;

       // topic名字check , 长度不能超512
    if (!topic || strlen(topic) > 512) {
        if (conf)
            rd_kafka_topic_conf_destroy(conf);
        rd_kafka_set_last_error(RD_KAFKA_RESP_ERR__INVALID_ARG,
                    EINVAL);
        return NULL;
    }

    if (do_lock)
                rd_kafka_wrlock(rk);
        // 所有创建的rd_kafka_itopic对象都会加入到对应的topic的rk->rk_topics中, 先从中查找,
    if ((s_rkt = rd_kafka_topic_find(rk, topic, 0/*no lock*/))) {
                if (do_lock)
                        rd_kafka_wrunlock(rk);
        if (conf)
            rd_kafka_topic_conf_destroy(conf);
                if (existing)
                        *existing = 1;
        return s_rkt;
        }

        if (existing)
                *existing = 0;

        // 分配对应的内存, 设置各属性
    rkt = rd_calloc(1, sizeof(*rkt));

    rkt->rkt_topic     = rd_kafkap_str_new(topic, -1);
    rkt->rkt_rk        = rk;

    if (!conf) {
                if (rk->rk_conf.topic_conf)
                        conf = rd_kafka_topic_conf_dup(rk->rk_conf.topic_conf);
                else
                        conf = rd_kafka_topic_conf_new();
        }
    rkt->rkt_conf = *conf;
    rd_free(conf); /* explicitly not rd_kafka_topic_destroy()
                    * since we dont want to rd_free internal members,
                    * just the placeholder. The internal members
                    * were copied on the line above. */

    /* Default partitioner: consistent_random */
    if (!rkt->rkt_conf.partitioner)
        rkt->rkt_conf.partitioner = rd_kafka_msg_partitioner_consistent_random;

    if (rkt->rkt_conf.compression_codec == RD_KAFKA_COMPRESSION_INHERIT)
        rkt->rkt_conf.compression_codec = rk->rk_conf.compression_codec;

        rd_list_init(&rkt->rkt_desp, 16, NULL);
        rd_refcnt_init(&rkt->rkt_refcnt, 0);

        s_rkt = rd_kafka_topic_keep(rkt);

    rwlock_init(&rkt->rkt_lock);
        mtx_init(&rkt->rkt_app_lock, mtx_plain);

    /* Create unassigned partition */
    rkt->rkt_ua = rd_kafka_toppar_new(rkt, RD_KAFKA_PARTITION_UA);

        // 加入到对应的rk_kafka_t中的topic列表
    TAILQ_INSERT_TAIL(&rk->rk_topics, rkt, rkt_link);
    rk->rk_topic_cnt++;

        /* Populate from metadata cache. */
        // 加入或更新到metadata cache
        if ((rkmce = rd_kafka_metadata_cache_find(rk, topic, 1/*valid*/))) {
                if (existing)
                        *existing = 1;
                rd_kafka_topic_metadata_update(rkt, &rkmce->rkmce_mtopic,
                                               rkmce->rkmce_ts_insert);
        }

        if (do_lock)
                rd_kafka_wrunlock(rk);

    return s_rkt;
```

- 创建 `rd_kafka_topic_t` 对象, 对外的接口 `rd_kafka_topic_new`

```
rd_kafka_topic_t *rd_kafka_topic_new (rd_kafka_t *rk, const char *topic,
                                      rd_kafka_topic_conf_t *conf) {
        shptr_rd_kafka_itopic_t *s_rkt;
        rd_kafka_itopic_t *rkt;
        rd_kafka_topic_t *app_rkt;
        int existing;

       // 创建一个`shptr_rd_kafka_itopic_t`对象
        s_rkt = rd_kafka_topic_new0(rk, topic, conf, &existing, 1/*lock*/);
        if (!s_rkt)
                return NULL;

        // 指针转换, 从`shptr_rd_kafka_itopic_t`到`rd_kafka_itopic_t`, 引用计数不变
        rkt = rd_kafka_topic_s2i(s_rkt);

        /* Save a shared pointer to be used in callbacks. */
        // 引数计数加1, 指针转换成一个`rd_kafka_topic_t`
        // app相对应的引用计数也加1
    app_rkt = rd_kafka_topic_keep_app(rkt);

        /* Query for the topic leader (async) */
        if (!existing)
               // 发metadata request, 获取leader等相关信息
                rd_kafka_topic_leader_query(rk, rkt);

        /* Drop our reference since there is already/now a rkt_app_rkt */
        rd_kafka_topic_destroy0(s_rkt);

        return app_rkt;
}
```

- 获取当前 `rd_kafka_t` 对象持有的所有topic的名字,保存在一个 `rd_list` 中

```
void rd_kafka_local_topics_to_list (rd_kafka_t *rk, rd_list_t *topics) {
        rd_kafka_itopic_t *rkt;

        rd_kafka_rdlock(rk);
        rd_list_grow(topics, rk->rk_topic_cnt);
        TAILQ_FOREACH(rkt, &rk->rk_topics, rkt_link)
                rd_list_add(topics, rd_strdup(rkt->rkt_topic->str));
        rd_kafka_rdunlock(rk);
}
```

- 判断parition是否是有效的,就是判断其leader是否有效

```
int rd_kafka_topic_partition_available (const rd_kafka_topic_t *app_rkt,
                    int32_t partition) {
    int avail;
    shptr_rd_kafka_toppar_t *s_rktp;
        rd_kafka_toppar_t *rktp;
        rd_kafka_broker_t *rkb;

    s_rktp = rd_kafka_toppar_get(rd_kafka_topic_a2i(app_rkt),
                                     partition, 0/*no ua-on-miss*/);
    if (unlikely(!s_rktp))
        return 0;

        rktp = rd_kafka_toppar_s2i(s_rktp);
        rkb = rd_kafka_toppar_leader(rktp, 1/*proper broker*/);
        avail = rkb ? 1 : 0;
        if (rkb)
                rd_kafka_broker_destroy(rkb);
    rd_kafka_toppar_destroy(s_rktp);
    return avail;
}
```

- 扫描所有topic的patitions:
- 筛出 kafka message过期的, 回调application层
- 找出需要刷新metadata的, 发送metadata request

```
int rd_kafka_topic_scan_all (rd_kafka_t *rk, rd_ts_t now) {
    rd_kafka_itopic_t *rkt;
    rd_kafka_toppar_t *rktp;
        shptr_rd_kafka_toppar_t *s_rktp;
    int totcnt = 0;
        rd_list_t query_topics;

        rd_list_init(&query_topics, 0, rd_free);

    rd_kafka_rdlock(rk);
    TAILQ_FOREACH(rkt, &rk->rk_topics, rkt_link) {
        int p;
                int cnt = 0, tpcnt = 0;
                rd_kafka_msgq_t timedout;
                int query_this = 0;

                rd_kafka_msgq_init(&timedout);

        rd_kafka_topic_wrlock(rkt);

                /* Check if metadata information has timed out. */
               // metadata cache中没有缓存,需要query metadata
                if (rkt->rkt_state != RD_KAFKA_TOPIC_S_UNKNOWN &&
                    !rd_kafka_metadata_cache_topic_get(
                            rk, rkt->rkt_topic->str, 1/*only valid*/)) {
                        rd_kafka_topic_set_state(rkt, RD_KAFKA_TOPIC_S_UNKNOWN);

                        query_this = 1;
                }

                /* Just need a read-lock from here on. */
                rd_kafka_topic_wrunlock(rkt);
                rd_kafka_topic_rdlock(rkt);

                if (rkt->rkt_partition_cnt == 0) {
                        query_this = 1;
                }

        for (p = RD_KAFKA_PARTITION_UA ;
             p < rkt->rkt_partition_cnt ; p++) {
            int did_tmout = 0;

            if (!(s_rktp = rd_kafka_toppar_get(rkt, p, 0)))
                continue;

                        rktp = rd_kafka_toppar_s2i(s_rktp);
            rd_kafka_toppar_lock(rktp);

                        /* Check that partition has a leader that is up,
                         * else add topic to query list. */
                       // partition leader无效时, 要request metadata
                        if (p != RD_KAFKA_PARTITION_UA &&
                            (!rktp->rktp_leader ||
                             rktp->rktp_leader->rkb_source ==
                             RD_KAFKA_INTERNAL ||
                             rd_kafka_broker_get_state(rktp->rktp_leader) <
                             RD_KAFKA_BROKER_STATE_UP)) {
                                query_this = 1;
                        }

            /* Scan toppar's message queues for timeouts */
            if (rd_kafka_msgq_age_scan(&rktp->rktp_xmit_msgq,
                           &timedout, now) > 0)
                did_tmout = 1;

            if (rd_kafka_msgq_age_scan(&rktp->rktp_msgq,
                           &timedout, now) > 0)
                did_tmout = 1;

            tpcnt += did_tmout;

            rd_kafka_toppar_unlock(rktp);
            rd_kafka_toppar_destroy(s_rktp);
        }

                rd_kafka_topic_rdunlock(rkt);
                if ((cnt = rd_atomic32_get(&timedout.rkmq_msg_cnt)) > 0) {
                        totcnt += cnt;
                        // kafka mesage过期, 则需要回调到application层
                        rd_kafka_dr_msgq(rkt, &timedout,
                                         RD_KAFKA_RESP_ERR__MSG_TIMED_OUT);
                }

                /* Need to re-query this topic's leader. */
                if (query_this &&
                    !rd_list_find(&query_topics, rkt->rkt_topic->str,
                                  (void *)strcmp))
                        rd_list_add(&query_topics,
                                    rd_strdup(rkt->rkt_topic->str));

        }
        rd_kafka_rdunlock(rk);

        if (!rd_list_empty(&query_topics))
                // 发送 metadata request
                rd_kafka_metadata_refresh_topics(rk, NULL, &query_topics,
                                                 1/*force even if cached
                                                    * info exists*/,
                                                 "refresh unavailable topics");
        rd_list_destroy(&query_topics);

        return totcnt;
}
```

- 更新topic的partition个数, partition个数可能增加, 也可能减少 `rd_kafka_topic_partition_cnt_update` , 简单讲:
- 新增的partition, 创建;
- 老的partition, 删除;

```
static int rd_kafka_topic_partition_cnt_update (rd_kafka_itopic_t *rkt,
                        int32_t partition_cnt) {
    rd_kafka_t *rk = rkt->rkt_rk;
    shptr_rd_kafka_toppar_t **rktps;
    shptr_rd_kafka_toppar_t *rktp_ua;
        shptr_rd_kafka_toppar_t *s_rktp;
    rd_kafka_toppar_t *rktp;
    rd_kafka_msgq_t tmpq = RD_KAFKA_MSGQ_INITIALIZER(tmpq);
    int32_t i;

        更新前后partition数量相同的话, 不作任何处理
    if (likely(rkt->rkt_partition_cnt == partition_cnt))
        return 0; /* No change in partition count */

    /* Create and assign new partition list */
       // 创建新的partition list, 分配内存
    if (partition_cnt > 0)
        rktps = rd_calloc(partition_cnt, sizeof(*rktps));
    else
        rktps = NULL;

        // 如果新个数大于老个数
    for (i = 0 ; i < partition_cnt ; i++) {
                // 多出来的都是新扩容的partition
        if (i >= rkt->rkt_partition_cnt) {
            /* New partition. Check if its in the list of
             * desired partitions first. */
                        // 检查是否在desired partition 列表中
                        s_rktp = rd_kafka_toppar_desired_get(rkt, i);

                        rktp = s_rktp ? rd_kafka_toppar_s2i(s_rktp) : NULL;
                        if (rktp) {
                                // 在desired partition 列表中,则移除它
                rd_kafka_toppar_lock(rktp);
                                rktp->rktp_flags &= ~RD_KAFKA_TOPPAR_F_UNKNOWN;

                                /* Remove from desp list since the
                                 * partition is now known. */
                                rd_kafka_toppar_desired_unlink(rktp);
                                rd_kafka_toppar_unlock(rktp);
            } else
                s_rktp = rd_kafka_toppar_new(rkt, i);
                        // 赋值rktps[i]
            rktps[i] = s_rktp;
        } else {
                        // 如果是已经存在的partition, 放到rktps[i], 并且作引用计数的增减
            /* Existing partition, grab our own reference. */
            rktps[i] = rd_kafka_toppar_keep(
                rd_kafka_toppar_s2i(rkt->rkt_p[i]));
            /* Loose previous ref */
            rd_kafka_toppar_destroy(rkt->rkt_p[i]);
        }
    }

    rktp_ua = rd_kafka_toppar_get(rkt, RD_KAFKA_PARTITION_UA, 0);

        /* Propagate notexist errors for desired partitions */
        // 扫描desired partition 列表中, 还余下的都是无主的, 集群中不存在的partition, 回调e
        RD_LIST_FOREACH(s_rktp, &rkt->rkt_desp, i) {
                rd_kafka_toppar_enq_error(rd_kafka_toppar_s2i(s_rktp),
                                          RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION);
        }

    /* Remove excessive partitions */
        // 处理更新后的partition个数小于更新前的情况, 需要删除一部分partition
    for (i = partition_cnt ; i < rkt->rkt_partition_cnt ; i++) {
        s_rktp = rkt->rkt_p[i];
                rktp = rd_kafka_toppar_s2i(s_rktp);
        rd_kafka_toppar_lock(rktp);

        if (rktp->rktp_flags & RD_KAFKA_TOPPAR_F_DESIRED) {
                        rd_kafka_dbg(rkt->rkt_rk, TOPIC, "DESIRED",
                                     "Topic %s [%"PRId32"] is desired "
                                     "but no longer known: "
                                     "moving back on desired list",
                                     rkt->rkt_topic->str, rktp->rktp_partition);
                        // 是DESIRED状态的话, 再放回到desired列表
            rktp->rktp_flags |= RD_KAFKA_TOPPAR_F_UNKNOWN;
                        rd_kafka_toppar_desired_link(rktp);

                        if (!rd_kafka_terminating(rkt->rkt_rk))
                                rd_kafka_toppar_enq_error(
                                        rktp,
                                        RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION);
                        // 解除和broker的联系, 实际上是关联到内部的UA broker
            rd_kafka_toppar_broker_delegate(rktp, NULL, 0);

        } else {
            /* Tell handling broker to let go of the toppar */
            rktp->rktp_flags |= RD_KAFKA_TOPPAR_F_REMOVE;
            rd_kafka_toppar_broker_leave_for_remove(rktp);
        }

        rd_kafka_toppar_unlock(rktp);

        rd_kafka_toppar_destroy(s_rktp);
    }


    if (rkt->rkt_p)
        rd_free(rkt->rkt_p);

    rkt->rkt_p = rktps;
    rkt->rkt_partition_cnt = partition_cnt;

    return 1;
}
```

- 将在UA partition上待发送的kafka message重新分配到有效的patition上 `rd_kafka_topic_assign_uas` :

```
static void rd_kafka_topic_assign_uas (rd_kafka_itopic_t *rkt,
                                       rd_kafka_resp_err_t err) {
    rd_kafka_t *rk = rkt->rkt_rk;
    shptr_rd_kafka_toppar_t *s_rktp_ua;
        rd_kafka_toppar_t *rktp_ua;
    rd_kafka_msg_t *rkm, *tmp;
    rd_kafka_msgq_t uas = RD_KAFKA_MSGQ_INITIALIZER(uas);
    rd_kafka_msgq_t failed = RD_KAFKA_MSGQ_INITIALIZER(failed);
    int cnt;

    if (rkt->rkt_rk->rk_type != RD_KAFKA_PRODUCER)
        return;

        // 没有UA partition,就直接返回了
    s_rktp_ua = rd_kafka_toppar_get(rkt, RD_KAFKA_PARTITION_UA, 0);
    if (unlikely(!s_rktp_ua)) {
        return;
    }

        rktp_ua = rd_kafka_toppar_s2i(s_rktp_ua);

        // 将ua partition上的msg移动到临时队列上
    rd_kafka_toppar_lock(rktp_ua);
    rd_kafka_msgq_move(&uas, &rktp_ua->rktp_msgq);
    cnt = rd_atomic32_get(&uas.rkmq_msg_cnt);
    rd_kafka_toppar_unlock(rktp_ua);

    TAILQ_FOREACH_SAFE(rkm, &uas.rkmq_msgs, rkm_link, tmp) {
        /* Fast-path for failing messages with forced partition */
               // 无效的msg放到failed 队列
        if (rkm->rkm_partition != RD_KAFKA_PARTITION_UA &&
            rkm->rkm_partition >= rkt->rkt_partition_cnt &&
            rkt->rkt_state != RD_KAFKA_TOPIC_S_UNKNOWN) {
            rd_kafka_msgq_enq(&failed, rkm);
            continue;
        }

               // 重新路由kafka message到相应的partition, 失败则放入failed 队列
        if (unlikely(rd_kafka_msg_partitioner(rkt, rkm, 0) != 0)) {
            /* Desired partition not available */
            rd_kafka_msgq_enq(&failed, rkm);
        }
    }

        // 失败的msg, 都回调给application层
    if (rd_atomic32_get(&failed.rkmq_msg_cnt) > 0) {
        /* Fail the messages */
        rd_kafka_dr_msgq(rkt, &failed,
                 rkt->rkt_state == RD_KAFKA_TOPIC_S_NOTEXISTS ?
                 err :
                 RD_KAFKA_RESP_ERR__UNKNOWN_PARTITION);
    }

    rd_kafka_toppar_destroy(s_rktp_ua); /* from get() */
}
```

* 关于metadata相关的操作, 我们介绍metadata时再来分析

Librdkafka源码分析-Content Table

发布于 2019-01-18 09:01

Kafka 分布式存储 大数据

还没有评论

写下你的评论...

文章被以下专栏收录

消息系统kafka的那些事儿
分享分布式消息系统kafka的点点滴滴

推荐阅读

搞透Kafka的存储架构，看这篇就够了

华仔聊技术 发表于kafka...

Kafka设计解析（一） Kafka背景及架构介绍

原创文章，转载请务必将下面这段话置于文章开头处。（已授权InfoQ中文站发布） 本文转发自技术世界，原文链接 Kafka设计解析（一）- Kafka背景及架构介绍1. 摘要Kafka是由LinkedIn开发并开...

郭俊 Jason

图解Kafka

Enjoy... 发表于Enjoy...

Kafka零拷贝

消失er

× 登录即可查看 超5亿 专业优质内容
超 5 千万创作者的优质提问、专业回答、深度文章和精彩视频尽在知乎。
立即登录/注册

▲ 赞同 2 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···