芯祥联科技 MQTT Broker 商用 SDK V2.0.0 使用手册

文档信息

项目	内容
文档版本	V1.0
适用SDK版本	MQTT Broker SDK V2.0.0
发布日期	2026-03-18
编制单位	芯祥联科技 MQTT 项目组
技术支持	hezuo@xxltech.cn
官方网站	www.xxltech.cn

1. 产品概述

1.1 产品简介

芯祥联科技 MQTT Broker 商用 SDK（以下简称“SDK”）是一款基于 MQTT 3.1.1/5.0 双协议标准研发的高性能、高可靠 Broker 核心接口库。该 SDK 支持多实例独立运行、跨平台部署，具备客户端认证、主题权限控制、事件通知、消息管理等核心能力，可快速集成到各类物联网、工业控制、智慧设备等场景，为用户提供稳定、高效的 MQTT 消息中转服务。

1.2 核心特性

- 多实例安全：支持多个 Broker 实例并行运行，端口、配置、回调完全隔离，无全局状态污染。
- 跨平台兼容：完美适配 Windows、Linux 操作系统，支持动态库（DLL/SO）、静态库两种集成方式。
- 协议兼容：严格遵循 OASIS MQTT 3.1.1/5.0 官方标准，支持 QoS 0/1/2 可靠投递、遗嘱消息、保留消息等核心特性。
- 高并发支持：基于静态内存池+分段锁架构，单实例可稳定支撑 5000+ 客户端并发连接。

- 灵活可控：提供完善的配置接口、客户端管理接口、统计接口，支持自定义认证、ACL 权限控制及事件回调。
- 商用级稳定：具备完善的异常处理、内存管理、超时清理机制，无内存泄漏、锁冲突等问题，可直接用于生产环境。

1.3 适用场景

- 物联网设备接入：智能家居、工业传感器、智能网关等设备的消息中转。
- 工业控制：工业设备数据采集、指令下发、设备状态监控。
- 智慧场景：智慧园区、智慧交通、智慧医疗等场景的消息协同。
- 定制化 Broker 开发：快速搭建符合自身业务需求的 MQTT 消息服务器。

2. 快速上手

2.1 环境准备

2.1.1 硬件环境

- CPU：Intel Core i5 及以上，或同等性能处理器。
- 内存：最低 4GB，推荐 8GB 及以上（高并发场景推荐 32GB）。
- 存储：至少 100MB 空闲空间（用于 SDK 部署及日志存储）。

2.1.2 软件环境

操作系统	编译环境	依赖库
Windows 10/11（32/64位）	Visual Studio 2019/2022	ws2_32.lib、user32.lib
Linux（Ubuntu 18.04+ / CentOS 7+）	GCC 7.0+	pthread

2.2 集成步骤

SDK 集成遵循“创建实例-配置-启动-业务-停止-销毁”的标准流程，以下为最简集成示例，详细接口说明见后续章节。

2.2.1 头文件与库文件引入

将 SDK 提供的 `mqtt_broker_sdk.h` 头文件放入项目头文件目录，将编译好的库文件（Windows：mqtt_broker_sdk.lib/mqtt_broker_sdk.dll；Linux：libmqtt_broker_sdk.a/libmqtt_broker_sdk.so）放入项目库目录，并在项目中配置库依赖。

2.2.2 最简集成代码示例

```c
#include "mqtt_broker_sdk.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 示例：空实现回调（实际业务需根据需求修改）
bool auth_cb(broker_handle_t handle, const char* client_id, const char* username, const char* password) {
    return true; // 允许所有客户端连接
}

bool acl_cb(broker_handle_t handle, const char* client_id, const char* topic, bool is_publish) {
    return true; // 允许所有主题操作
}

void event_cb(broker_handle_t handle, BrokerEventType type, const char* json_data) {
    printf("Broker Event: type=%d, data=%s\n", type, json_data);
}

int main() {
    // 1. 创建Broker实例
    broker_handle_t broker = sdk_mqtt_broker_create();
    if (!broker) {
        printf("创建Broker实例失败\n");
        return -1;
    }

    // 2. 注册回调（认证、ACL、事件）
    sdk_mqtt_broker_register_auth_callback(broker, auth_cb);
    sdk_mqtt_broker_register_acl_callback(broker, acl_cb);
    sdk_mqtt_broker_register_event_callback(broker, event_cb);

    // 3. 配置参数（可选，使用默认配置可跳过）
    sdk_mqtt_broker_set_keepalive_timeout(broker, 300); // 心跳超时300秒
    sdk_mqtt_broker_set_max_packet_size(broker, 4096);  // 最大报文长度4KB

    // 4. 初始化资源
    if (sdk_mqtt_broker_init(broker) != BROKER_OK) {
        printf("Broker初始化失败\n");
        sdk_mqtt_broker_destroy(broker);
        return -1;
    }
```

```
42
43      // 5. 启动Broker（监听1883端口，最大5000客户端）
44      if (sdk_mqtt_broker_start(broker, 1883, 5000) != BROKER_OK) {
45          printf("Broker启动失败\n");
46          sdk_mqtt_broker_deinit(broker);
47          sdk_mqtt_broker_destroy(broker);
48          return -1;
49      }
50
51      printf("MQTT Broker 启动成功，监听端口: 1883\n");
52      printf("按Enter键停止服务...\n");
53      getchar();
54
55      // 6. 停止服务
56      sdk_mqtt_broker_stop(broker);
57
58      // 7. 释放资源
59      sdk_mqtt_broker_deinit(broker);
60
61      // 8. 销毁实例
62      sdk_mqtt_broker_destroy(broker);
63
64      printf("Broker服务已停止\n");
65      return 0;
66  }
```

2.2.3 编译命令

Windows（VS命令行）：

```
1  cl your_demo.c mqtt_broker_sdk.c mqtt_client.c mqtt_packet_handler.c ^
2      persistence.c netip.c agent_log.c mqtt_stats.c ^
3      /link ws2_32.lib user32.lib
```

Linux（GCC）：

```
1  gcc your_demo.c mqtt_broker_sdk.c mqtt_client.c mqtt_packet_handler.c ^
2      persistence.c netip.c agent_log.c mqtt_stats.c -lpthread -o broker_demo
```

3. 核心概念说明

3.1 实例句柄

SDK 采用句柄模型管理 Broker 实例，每个 Broker 实例对应一个独立的 `broker_handle_t` 句柄，所有对外 API 均需传入该句柄，确保多实例隔离。

定义：`typedef void* broker_handle_t;`

说明：句柄由 `sdk_mqtt_broker_create` 函数创建，使用完成后需通过 `sdk_mqtt_broker_destroy` 销毁，避免内存泄漏。

3.2 返回值约定

所有 SDK 接口返回值遵循以下约定，错误码可通过 `sdk_mqtt_broker_error_str` 函数获取具体描述：

返回值	说明
BROKER_OK (0)	操作成功
BROKER_ERR_INIT (-1)	初始化失败
BROKER_ERR_PARAM (-2)	参数无效
BROKER_ERR_RESOURCE (-3)	资源分配失败（如内存不足）
BROKER_ERR_NOT_RUNNING (-4)	Broker 未运行
BROKER_ERR_NOT_FOUND (-5)	未找到指定客户端/主题
BROKER_ERR_DENIED (-6)	操作被拒绝（如认证失败、ACL 拒绝）
BROKER_ERR_BUSY (-7)	Broker 正忙（如已运行，无法重复启动）
BROKER_ERR_TIMEOUT (-8)	操作超时
BROKER_ERR_UNKNOWN (-99)	未知错误

3.3 线程安全说明

- 所有对外 API 均为线程安全，支持多线程同时调用（同一实例的不同 API 可在不同线程中调用）。

- 回调函数（认证、ACL、事件）由 Broker 内部线程调用，回调内禁止执行耗时操作（如阻塞等待），禁止重入 SDK 阻塞接口，避免死锁。
- 多实例场景下，不同实例的 API 调用完全独立，无线程安全冲突。

4. 接口详细说明

本章节按功能模块分类，详细说明所有 SDK 对外接口的功能、参数、返回值及使用注意事项，所有接口均支持多实例，需传入对应实例句柄。

4.1 实例生命周期管理接口

用于 Broker 实例的创建、销毁、初始化、启动、停止，是 SDK 使用的基础。

4.1.1 创建 Broker 实例

```
broker_handle_t sdk_mqtt_broker_create(void);
```

- 功能：创建一个全新的 Broker 实例，分配实例所需内存及基础资源。
- 参数：无。
- 返回值：成功返回实例句柄，失败返回 NULL。
- 注意事项：每个实例对应独立的配置和资源，创建多个实例时需确保端口不冲突。

4.1.2 销毁 Broker 实例

```
void sdk_mqtt_broker_destroy(broker_handle_t handle);
```

- 功能：销毁指定的 Broker 实例，释放实例占用的所有内存及资源。
- 参数：`handle` - 待销毁的 Broker 实例句柄。
- 返回值：无。
- 注意事项：销毁前必须先调用 `sdk_mqtt_broker_stop` 和 `sdk_mqtt_broker_deinit`，否则会导致资源泄漏。

4.1.3 初始化 Broker 资源

```
int sdk_mqtt_broker_init(broker_handle_t handle);
```

- 功能：初始化 Broker 内部资源，包括内存池、协议栈、统计模块、持久化模块等。
- 参数：`handle` - Broker 实例句柄。
- 返回值：成功返回 BROKER_OK，失败返回对应错误码。
- 注意事项：必须在 `sdk_mqtt_broker_start` 之前调用，且每个实例仅需初始化一次。

4.1.4 启动 Broker 服务

```
int sdk_mqtt_broker_start(
    broker_handle_t handle,
    int listen_port,
    int max_clients
);
```

- 功能：启动 Broker 监听服务，开始接收客户端连接及消息处理。
- 参数：
  - `handle` - Broker 实例句柄。
  - `listen_port` - 监听端口（如 1883，需确保端口未被占用）。
  - `max_clients` - 最大并发客户端连接数（最大不超过 MQTT_MAX_CLIENTS 宏定义值）。
- 返回值：成功返回 BROKER_OK，失败返回对应错误码（如端口占用返回 BROKER_ERR_RESOURCE）。
- 注意事项：启动后不可修改核心配置（如最大报文长度、心跳超时）；多实例启动时，需使用不同的监听端口。

4.1.5 停止 Broker 服务

```
int sdk_mqtt_broker_stop(broker_handle_t handle);
```

- 功能：停止 Broker 监听服务，关闭所有客户端连接，退出内部处理线程。
- 参数：`handle` - Broker 实例句柄。
- 返回值：成功返回 BROKER_OK，失败返回对应错误码（如未运行返回 BROKER_ERR_NOT_RUNNING）。
- 注意事项：停止后可重新调用 `sdk_mqtt_broker_start` 重启服务。

4.1.6 释放 Broker 资源

```
1   void sdk_mqtt_broker_deinit(broker_handle_t handle);
```

- 功能：释放 Broker 初始化时分配的资源（内存池、锁、协议栈等），与 `sdk_mqtt_broker_init` 成对调用。
- 参数： `handle` - Broker 实例句柄。
- 返回值：无。
- 注意事项：必须在 `sdk_mqtt_broker_stop` 之后、 `sdk_mqtt_broker_destroy` 之前调用。

4.1.7 查询 Broker 运行状态

```
1   bool sdk_mqtt_broker_is_running(broker_handle_t handle);
```

- 功能：查询指定 Broker 实例是否处于运行状态。
- 参数： `handle` - Broker 实例句柄。
- 返回值：运行中返回 true，已停止返回 false。

4.2 配置接口

用于配置 Broker 核心参数，需在 `sdk_mqtt_broker_start` 之前调用，运行中不可修改。

4.2.1 设置日志级别

```
1   int sdk_mqtt_broker_set_log_level(broker_handle_t handle, int level);
```

- 功能：设置 Broker 日志输出级别，控制日志详细程度。
- 参数：
  - `handle` - Broker 实例句柄。
  - `level` - 日志级别（0：关闭，1：错误，2：警告，3：信息，4：调试）。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（级别无效）。

4.2.2 设置心跳超时时间

```
int sdk_mqtt_broker_set_keepalive_timeout(
    broker_handle_t handle,
    int timeout_sec
);
```

- 功能：设置客户端心跳超时时间，超过该时间未收到客户端心跳，Broker 将主动断开连接。
- 参数：
  - `handle` - Broker 实例句柄。
  - `timeout_sec` - 超时时间（秒），默认 300 秒，推荐值 180~300 秒（符合公网 NAT 超时特性）。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（超时时间≤0）或 BROKER_ERR_BUSY（已运行）。
- 注意事项：超时时间遵循 MQTT 标准，实际超时阈值为客户端心跳时间 × 1.5；若客户端心跳设为 0（禁用心跳），则使用该超时时间作为默认阈值。

4.2.3 设置最大报文长度

```
int sdk_mqtt_broker_set_max_packet_size(
    broker_handle_t handle,
    int size_bytes
);
```

- 功能：设置客户端与 Broker 之间允许传输的最大报文长度，超过该长度的报文将被直接丢弃。
- 参数：
  - `handle` - Broker 实例句柄。
  - `size_bytes` - 最大报文长度（字节），范围 256~10×1024×1024（10MB），默认 4096 字节。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（长度超出范围）或 BROKER_ERR_BUSY（已运行）。

4.3 回调注册接口

用于注册自定义回调函数，实现客户端认证、主题权限控制、事件通知等自定义逻辑，是 SDK 灵活扩展的核心。

4.3.1 注册事件回调

```
void sdk_mqtt_broker_register_event_callback(
    broker_handle_t handle,
    broker_event_cb cb
);
```

- 功能：注册 Broker 事件回调函数，当发生客户端上下线、消息发布、订阅等事件时，SDK 将调用该回调通知业务层。
- 参数：
  - `handle` - Broker 实例句柄。
  - `cb` - 事件回调函数指针，原型如下：
    `typedef void (*broker_event_cb)(`
    `broker_handle_t handle,`
    `BrokerEventType type,`
    `const char* json_data`
    `);`
- 回调参数说明：
  - `handle` - 触发事件的 Broker 实例句柄。
  - `type` - 事件类型（枚举 `BrokerEventType`），具体值如下：
    - BROKER_EVENT_CLIENT_CONNECT（0）：客户端上线
    - BROKER_EVENT_CLIENT_DISCONNECT（1）：客户端下线
    - BROKER_EVENT_CLIENT_PUBLISH（2）：客户端发布消息
    - BROKER_EVENT_CLIENT_SUBSCRIBE（3）：客户端订阅主题
    - BROKER_EVENT_CLIENT_UNSUBSCRIBE（4）：客户端取消订阅
    - BROKER_EVENT_WILL_MESSAGE（5）：触发遗嘱消息
  - `json_data` - 事件数据（JSON 格式字符串），包含事件相关详细信息（如客户端 ID、主题、消息内容等）。
- 注意事项：回调函数由 SDK 内部线程调用，禁止耗时操作；若无需事件通知，可传入 NULL 取消注册。

4.3.2 注册认证回调

```
void sdk_mqtt_broker_register_auth_callback(
    broker_handle_t handle,
    broker_auth_cb cb
);
```

- 功能：注册客户端连接认证回调函数，客户端发送 CONNECT 报文时，SDK 将调用该回调校验客户端合法性。
- 参数：
  - `handle` - Broker 实例句柄。
  - `cb` - 认证回调函数指针，原型如下：
    `typedef bool (*broker_auth_cb)(broker_handle_t handle, const char* client_id, const char* username, const char* password);`
- 回调参数说明：
  - `handle` - Broker 实例句柄。
  - `client_id` - 客户端 ID（不可为空）。
  - `username` - 客户端用户名（可为 NULL）。
  - `password` - 客户端密码（可为 NULL）。
- 返回值：回调函数返回 true 表示允许客户端连接，返回 false 表示拒绝连接（Broker 将发送 CONNACK 报文拒绝连接）。
- 注意事项：若未注册该回调，默认允许所有客户端连接；回调内需快速完成认证逻辑，避免阻塞客户端连接。

4.3.3 注册 ACL 权限回调

```
void sdk_mqtt_broker_register_acl_callback(
    broker_handle_t handle,
    broker_acl_cb cb
```

```
);
```

- 功能：注册主题权限控制回调函数，客户端发布（PUBLISH）或订阅（SUBSCRIBE）主题时，SDK 将调用该回调校验权限。
- 参数：
  - `handle` - Broker 实例句柄。
  - `cb` - ACL 回调函数指针，原型如下：
    `typedef bool (*broker_acl_cb)(broker_handle_t handle, const char* client_id, const char* topic, bool is_publish);`
- 回调参数说明：
  - `handle` - Broker 实例句柄。
  - `client_id` - 客户端 ID。
  - `topic` - 客户端操作的主题。
  - `is_publish` - 操作类型：true 表示发布（PUBLISH），false 表示订阅（SUBSCRIBE）。
- 返回值：回调函数返回 true 表示允许操作，返回 false 表示拒绝操作（Broker 将拒绝该报文并返回对应错误）。
- 注意事项：若未注册该回调，默认允许所有主题操作；可根据客户端 ID、主题前缀等实现精细化权限控制。

4.4 客户端管理接口

用于查询客户端状态、管理客户端连接，支持获取在线客户端数量、客户端详情、强制踢人等操作。

4.4.1 获取在线客户端数量

```
int sdk_mqtt_broker_get_online_count(broker_handle_t handle);
```

- 功能：获取指定 Broker 实例的当前在线客户端数量。
- 参数： `handle` - Broker 实例句柄。

- 返回值：成功返回在线客户端数量（≥0），失败返回 BROKER_ERR_NOT_RUNNING（未运行）。

4.4.2 获取客户端 ID 列表

```
int sdk_mqtt_broker_get_client_id_list(
    broker_handle_t handle,
    char* out_list,
    int max_len
);
```

- 功能：获取指定 Broker 实例的所有在线客户端 ID 列表，以逗号分隔。
- 参数：
  - `handle` - Broker 实例句柄。
  - `out_list` - 输出缓冲区，用于存储客户端 ID 列表。
  - `max_len` - 输出缓冲区最大长度（需包含字符串结束符 '\0'）。
- 返回值：成功返回实际写入缓冲区的字符数（不含结束符），失败返回 BROKER_ERR_PARAM（缓冲区为空或长度≤0）或 BROKER_ERR_NOT_RUNNING（未运行）。
- 示例：若在线客户端 ID 为 "client001"、"client002"，则输出缓冲区内容为 "client001,client002"。

4.4.3 获取客户端详细信息

```
int sdk_mqtt_broker_get_client_info(
    broker_handle_t handle,
    const char* client_id,
    BrokerClientInfo* out_info
);
```

- 功能：获取指定客户端的详细信息（IP、端口、心跳、在线状态等）。
- 参数：
  - `handle` - Broker 实例句柄。
  - `client_id` - 待查询的客户端 ID。

- `out_info` - 输出结构体，用于存储客户端详细信息，结构体定义如下：

```
typedef struct {
    char client_id[128];        // 客户端ID
    char ip[64];                // 客户端IP地址
    uint16_t port;              // 客户端端口
    uint16_t keepalive;         // 心跳时间（秒）
    bool     is_connected;      // 是否在线
    uint32_t connect_time;      // 连接时间戳（秒，从1970-01-01开始）
} BrokerClientInfo;
```

- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（参数无效）、BROKER_ERR_NOT_FOUND（客户端不存在）或 BROKER_ERR_NOT_RUNNING（未运行）。

4.4.4 强制踢掉客户端

```
int sdk_mqtt_broker_kick_client(
    broker_handle_t handle,
    const char* client_id
);
```

- 功能：强制断开指定客户端的连接，客户端将收到 Broker 发送的 DISCONNECT 报文（若支持）。
- 参数：
  - `handle` - Broker 实例句柄。
  - `client_id` - 待踢掉的客户端 ID。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（客户端 ID 为空）、BROKER_ERR_NOT_FOUND（客户端不存在）或 BROKER_ERR_NOT_RUNNING（未运行）。
- 注意事项：踢掉客户端后，若客户端开启了遗嘱消息且未正常发送 DISCONNECT 报文，Broker 将触发遗嘱消息发送。

4.5 服务端消息发布接口

用于 Broker 主动向订阅对应主题的客户端发布消息，支持 QoS 0/1/2 可靠投递及保留消息。

```
int sdk_mqtt_broker_publish(
    broker_handle_t handle,
    const char* topic,
    const uint8_t* payload,
```

```
    int payload_len,
    int qos,
    bool retain
);
```

- 功能：Broker 主动发布消息，自动推送给所有匹配该主题的订阅客户端。
- 参数：
  - `handle` - Broker 实例句柄。
  - `topic` - 消息主题（不可为空）。
  - `payload` - 消息内容（不可为空）。
  - `payload_len` - 消息内容长度（≥1，且不超过最大报文长度）。
  - `qos` - QoS 等级（0、1、2），推荐根据业务需求选择：
    - QoS 0：最多一次投递，不保证消息到达。
    - QoS 1：至少一次投递，保证消息到达，可能重复。
    - QoS 2：恰好一次投递，保证消息到达且不重复（性能开销最高）。
  - `retain` - 是否为保留消息：true 表示保留，false 表示不保留。
    - 保留消息：新订阅该主题的客户端将立即收到该保留消息。
    - 若发布相同主题的保留消息，将覆盖原有保留消息；若 payload_len 为 0，将删除该主题的保留消息。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（参数无效）、BROKER_ERR_NOT_RUNNING（未运行）或 BROKER_ERR_RESOURCE（资源不足）。

4.6 保留消息管理接口

用于管理 Broker 中的保留消息，支持清除指定主题保留消息、获取保留消息主题列表。

4.6.1 清除指定主题保留消息

```
int sdk_mqtt_broker_clear_retain(
    broker_handle_t handle,
    const char* topic
);
```

- 功能：清除指定主题的保留消息（若存在）。
- 参数：
  - `handle` - Broker 实例句柄。
  - `topic` - 待清除保留消息的主题（不可为空）。
- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（参数无效）、BROKER_ERR_NOT_RUNNING（未运行）或 BROKER_ERR_NOT_FOUND（该主题无保留消息）。

4.6.2 获取保留消息主题列表

```
int sdk_mqtt_broker_get_retain_topic_list(
    broker_handle_t handle,
    char* out_list,
    int max_len
);
```

- 功能：获取指定 Broker 实例中所有存在保留消息的主题列表，以逗号分隔。
- 参数：
  - `handle` - Broker 实例句柄。
  - `out_list` - 输出缓冲区，用于存储主题列表。
  - `max_len` - 输出缓冲区最大长度（需包含字符串结束符 '\0'）。
- 返回值：成功返回实际写入缓冲区的字符数（不含结束符），失败返回 BROKER_ERR_PARAM（缓冲区为空或长度≤0）或 BROKER_ERR_NOT_RUNNING（未运行）。

4.7 统计监控接口

用于获取 Broker 运行状态统计信息，便于业务层监控 Broker 运行情况。

```
int sdk_mqtt_broker_get_stats(
    broker_handle_t handle,
    BrokerStats* out_stats
);
```

- 功能：获取指定 Broker 实例的运行统计信息，包括在线客户端数、消息总数、运行时间等。

- 参数：
  - `handle` - Broker 实例句柄。
  - `out_stats` - 输出结构体，用于存储统计信息，结构体定义如下：

```
typedef struct {
    int       online_count;          // 在线客户端数
    uint32_t total_publish;          // 消息总数（所有 QoS 等级）
    uint32_t total_subscribe;        // 订阅总数
    uint32_t uptime;                 // 运行时间（秒）
} BrokerStats;
```

- 返回值：成功返回 BROKER_OK，失败返回 BROKER_ERR_PARAM（参数无效）或 BROKER_ERR_NOT_RUNNING（未运行）。

4.8 工具接口

提供 SDK 版本查询、错误描述获取等工具功能，便于问题排查和版本管理。

4.8.1 获取 SDK 版本号

```
1   const char* sdk_mqtt_broker_version(void);
```

- 功能：获取 SDK 版本号字符串（如 "V2.0.0"）。
- 参数：无。
- 返回值：SDK 版本号字符串（只读，不可修改）。

4.8.2 获取错误描述字符串

```
1   const char* sdk_mqtt_broker_error_str(int err_code);
```

- 功能：根据错误码获取对应的错误描述字符串，便于问题排查。
- 参数：`err_code` - SDK 接口返回的错误码（负数）。
- 返回值：错误描述字符串（如 BROKER_ERR_INIT 对应 "Init failed"），未知错误返回 "Unknown error"。

5. 多实例使用说明

SDK 完全支持多 Broker 实例并行运行，每个实例独立监听端口、独立配置、独立回调，互不干扰。以下为多实例使用示例及注意事项。

5.1 多实例使用示例

```
#include "mqtt_broker_sdk.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h> // Windows平台需添加，Linux替换为#include <pthread.h>

// 实例1（1883端口）回调
bool auth1(broker_handle_t handle, const char* client_id, const char* username, const char* password) {
    printf("[1883] 认证: client_id=%s\n", client_id);
    return true; // 允许所有客户端连接
}

void event1(broker_handle_t handle, BrokerEventType type, const char* json_data) {
    printf("[1883] 事件: type=%d, data=%s\n", type, json_data);
}

bool acl1(broker_handle_t handle, const char* client_id, const char* topic, bool is_publish) {
    printf("[1883] ACL校验: client_id=%s, topic=%s, 发布=%d\n", client_id, topic, is_publish);
    return true; // 允许所有主题操作
}

// 实例2（1884端口）回调
bool auth2(broker_handle_t handle, const char* client_id, const char* username, const char* password) {
    printf("[1884] 认证: client_id=%s\n", client_id);
    // 示例：仅允许client_id以"dev_"开头的客户端连接
    if (client_id != NULL && strstr(client_id, "dev_") == client_id) {
        return true;
    }
    return false;
}

void event2(broker_handle_t handle, BrokerEventType type, const char* json_data) {
    printf("[1884] 事件: type=%d, data=%s\n", type, json_data);
}

```

```
bool acl2(broker_handle_t handle, const char* client_id, const char* topic, bool is_publish) {
    // 示例：仅允许发布/订阅以"device/"为前缀的主题
    if (topic != NULL && strstr(topic, "device/") == topic) {
        return true;
    }
    printf("[1884] ACL拒绝：client_id=%s, topic=%s, 发布=%d\n", client_id, topic, is_publish);
    return false;
}

// 线程函数：用于运行两个Broker实例（Windows用CreateThread，Linux用pthread_create）
DWORD WINAPI broker_thread1(LPVOID param) {
    broker_handle_t broker = (broker_handle_t)param;
    // 启动实例1（1883端口，最大5000客户端）
    if (sdk_mqtt_broker_start(broker, 1883, 5000) != BROKER_OK) {
        printf("[1883] Broker启动失败\n");
        sdk_mqtt_broker_deinit(broker);
        sdk_mqtt_broker_destroy(broker);
        return -1;
    }
    printf("[1883] MQTT Broker 启动成功，监听端口：1883\n");
    // 阻塞线程，保持Broker运行
    while (1) {
        Sleep(1000); // Windows延迟，Linux替换为sleep(1)
    }
    return 0;
}

DWORD WINAPI broker_thread2(LPVOID param) {
    broker_handle_t broker = (broker_handle_t)param;
    // 启动实例2（1884端口，最大3000客户端）
    if (sdk_mqtt_broker_start(broker, 1884, 3000) != BROKER_OK) {
        printf("[1884] Broker启动失败\n");
        sdk_mqtt_broker_deinit(broker);
        sdk_mqtt_broker_destroy(broker);
        return -1;
    }
    printf("[1884] MQTT Broker 启动成功，监听端口：1884\n");
    // 阻塞线程，保持Broker运行
    while (1) {
        Sleep(1000); // Windows延迟，Linux替换为sleep(1)
    }
    return 0;
}

```

```c
int main() {
    // 1. 创建两个独立的Broker实例
    broker_handle_t broker1 = sdk_mqtt_broker_create();
    broker_handle_t broker2 = sdk_mqtt_broker_create();
    if (!broker1 || !broker2) {
        printf("创建Broker实例失败\n");
        if (broker1) sdk_mqtt_broker_destroy(broker1);
        if (broker2) sdk_mqtt_broker_destroy(broker2);
        return -1;
    }

    // 2. 配置实例1（1883端口）并注册回调
    sdk_mqtt_broker_register_auth_callback(broker1, auth1);
    sdk_mqtt_broker_register_acl_callback(broker1, acl1);
    sdk_mqtt_broker_register_event_callback(broker1, event1);
    sdk_mqtt_broker_set_keepalive_timeout(broker1, 300);
    sdk_mqtt_broker_set_max_packet_size(broker1, 4096);
    // 初始化实例1资源
    if (sdk_mqtt_broker_init(broker1) != BROKER_OK) {
        printf("[1883] Broker初始化失败\n");
        sdk_mqtt_broker_destroy(broker1);
        sdk_mqtt_broker_destroy(broker2);
        return -1;
    }

    // 3. 配置实例2（1884端口）并注册回调
    sdk_mqtt_broker_register_auth_callback(broker2, auth2);
    sdk_mqtt_broker_register_acl_callback(broker2, acl2);
    sdk_mqtt_broker_register_event_callback(broker2, event2);
    sdk_mqtt_broker_set_keepalive_timeout(broker2, 180); // 与实例1不同的心跳超时
    sdk_mqtt_broker_set_max_packet_size(broker2, 8192);  // 与实例1不同的最大报文长度
    // 初始化实例2资源
    if (sdk_mqtt_broker_init(broker2) != BROKER_OK) {
        printf("[1884] Broker初始化失败\n");
        sdk_mqtt_broker_deinit(broker1);
        sdk_mqtt_broker_destroy(broker1);
        sdk_mqtt_broker_destroy(broker2);
        return -1;
    }

    // 4. 创建线程，分别运行两个Broker实例
    HANDLE thread1 = CreateThread(NULL, 0, broker_thread1, (LPVOID)broker1, 0, NULL);
    HANDLE thread2 = CreateThread(NULL, 0, broker_thread2, (LPVOID)broker2, 0, NULL);
```

```
123         if (!thread1 || !thread2) {
124             printf("创建线程失败\n");
125             sdk_mqtt_broker_deinit(broker1);
126             sdk_mqtt_broker_deinit(broker2);
127             sdk_mqtt_broker_destroy(broker1);
128             sdk_mqtt_broker_destroy(broker2);
129             return -1;
130         }
131 
132         printf("两个Broker实例已启动，按Enter键停止服务...\n");
133         getchar();
134 
135         // 5. 停止服务、释放资源、销毁实例
136         sdk_mqtt_broker_stop(broker1);
137         sdk_mqtt_broker_stop(broker2);
138         sdk_mqtt_broker_deinit(broker1);
139         sdk_mqtt_broker_deinit(broker2);
140         sdk_mqtt_broker_destroy(broker1);
141         sdk_mqtt_broker_destroy(broker2);
142 
143         // 关闭线程
144         TerminateThread(thread1, 0);
145         TerminateThread(thread2, 0);
146         CloseHandle(thread1);
147         CloseHandle(thread2);
148 
149         printf("两个Broker服务已停止\n");
150         return 0;
151     }
152 
153     // Linux平台适配说明（替换线程相关代码）：
154     // 1. 替换头文件: #include <pthread.h>
155     // 2. 替换线程函数定义:
156     //    void* broker_thread1(void* param) { ... }
157     //    void* broker_thread2(void* param) { ... }
158     // 3. 替换线程创建:
159     //    pthread_t thread1, thread2;
160     //    pthread_create(&thread1, NULL, broker_thread1, (void*)broker1);
161     //    pthread_create(&thread2, NULL, broker_thread2, (void*)broker2);
162     // 4. 替换线程阻塞与终止:
163     //    pthread_join(thread1, NULL);
164     //    pthread_join(thread2, NULL);
165     // 5. 替换延迟函数: sleep(1);
166 
```

（注：文档部分内容可能由 AI 生成）