HTTP客户端使用指南

概述

ACL提供了功能强大的HTTP客户端实现，支持同步和异步两种模式，提供了丰富的特性如连接池、SSL、压缩、Cookie管理等。

核心类

1. http_client - 底层HTTP协议处理

`http_client`类负责HTTP协议的底层处理，包括请求发送、响应接收、chunked编码等。

```
// 创建HTTP客户端
acl::socket_stream conn;
conn.open("www.example.com:80", 10, 10);

acl::http_client client(&conn);

// 构建请求头
acl::http_header header;
header.set_url("/api/test");
header.set_host("www.example.com");
header.set_method(acl::HTTP_METHOD_GET);

// 发送请求
client.write_head(header);

// 读取响应
if (client.read_head()) {
    // 读取响应体
    acl::string body;
    while (true) {
        int ret = client.read_body(body, true);
        if (ret <= 0) {
            break;
        }
    }
}
```

2. http_request - 同步HTTP请求

`http_request`类是对`http_client`的高层封装，使用更加简单。

2.1 基本使用

```cpp
#include "acl_cpp/lib_acl.hpp"

// 方式1：指定服务器地址
acl::http_request req("www.example.com:80", 10, 10);

// 方式2：使用已建立的连接
acl::socket_stream* conn = new acl::socket_stream();
conn->open("www.example.com:80", 10, 10);
acl::http_request req2(conn);

// 设置URL
req.request_header().set_url("/api/users");
req.request_header().set_host("www.example.com");

// 发送GET请求
if (req.request(NULL, 0)) {
    // 获取响应状态码
    int status = req.http_status();
    printf("HTTP状态码: %d\n", status);

    // 获取响应体
    acl::string body;
    if (req.get_body(body)) {
        printf("响应: %s\n", body.c_str());
    }
}
```

2.2 发送POST请求

```cpp
acl::http_request req("api.example.com:80");

// 设置请求头
req.request_header().set_url("/api/login");
req.request_header().set_host("api.example.com");
req.request_header().set_content_type("application/json");

// 构建JSON请求体
acl::json json;
json.add_text("username", "user123");
json.add_text("password", "pass456");
acl::string body = json.to_string();

// 发送POST请求
if (req.post(body.c_str(), body.length())) {
    acl::string response;
    req.get_body(response);
    printf("响应: %s\n", response.c_str());
}
```

2.3 添加请求参数

```
acl::http_request req("www.example.com:80");

// 方式1：URL中包含参数
req.request_header().set_url("/api/search?keyword=test&page=1");

// 方式2：使用add_param添加参数
req.request_header().set_url("/api/search");
req.request_header().add_param("keyword", "test");
req.request_header().add_int("page", 1);
req.request_header().add_int("size", 20);

// 发送请求
req.request(NULL, 0);
```

2.4 设置请求头

```
acl::http_request req("api.example.com:80");

// 设置各种请求头
req.request_header().set_url("/api/data");
req.request_header().set_host("api.example.com");
req.request_header().add_entry("User-Agent", "MyApp/1.0");
req.request_header().add_entry("Authorization", "Bearer token123");
req.request_header().set_content_type("application/json");
req.request_header().set_keep_alive(true);

// 启用GZIP压缩
req.request_header().accept_gzip(true);
```

2.5 Cookie管理

```
acl::http_request req("www.example.com:80");

// 添加Cookie
req.request_header().add_cookie("sessionid", "abc123");
req.request_header().add_cookie("user", "john");

// 发送请求
req.request(NULL, 0);

// 获取响应中的Cookie
const std::vector<acl::HttpCookie*>* cookies = req.get_cookies();
if (cookies) {
    for (auto cookie : *cookies) {
        printf("Cookie: %s=%s\n",
```

```
                cookie->getName(),
                cookie->getValue());
    }
}
```

2.6 断点续传（Range请求）

```
acl::http_request req("cdn.example.com:80");

// 设置Range头，请求部分内容
req.request_header().set_url("/files/large.zip");
req.request_header().set_range(1024000, 2048000);  // 字节1MB到2MB

if (req.request(NULL, 0)) {
    // 检查是否支持Range
    if (req.support_range()) {
        long long from = req.get_range_from();
        long long to = req.get_range_to();
        long long max = req.get_range_max();

        printf("Range: %lld-%lld/%lld\n", from, to, max);

        // 读取数据
        char buf[8192];
        while (true) {
            int ret = req.read_body(buf, sizeof(buf));
            if (ret <= 0) break;
            // 处理数据
        }
    }
}
```

2.7 流式读取响应

```
acl::http_request req("www.example.com:80");

req.request_header().set_url("/api/stream");

if (req.request(NULL, 0)) {
    // 逐行读取
    acl::string line;
    while (req.body_gets(line, true)) {
        printf("Line: %s\n", line.c_str());
    }
}
```

2.8 分步发送请求

```
acl::http_request req("upload.example.com:80");

// 设置请求头
req.request_header().set_url("/upload");
req.request_header().set_content_length(file_size);

// 先发送请求头
if (!req.write_head()) {
    printf("发送请求头失败\n");
    return false;
}

// 分块发送请求体
char buf[8192];
while (/* 有数据 */) {
    int n = read_data(buf, sizeof(buf));
    if (!req.write_body(buf, n)) {
        printf("发送数据失败\n");
        return false;
    }
}

// 发送结束标志
req.write_body(NULL, 0);

// 读取响应
acl::string response;
req.get_body(response);
```

3. http_aclient - 异步HTTP客户端

异步HTTP客户端适用于高并发场景，基于事件驱动模型。

```
#include "acl_cpp/lib_acl.hpp"

class MyHttpClient : public acl::http_aclient {
public:
    MyHttpClient(acl::aio_handle& handle)
        : http_aclient(handle) {}

    virtual ~MyHttpClient() {}

    // 必须实现destroy方法
    void destroy() override {
        delete this;
    }

protected:
    // 连接成功回调
    bool on_connect() override {
```

```
        printf("连接成功\n");

        // 设置请求头
        request_header().set_url("/api/test");
        request_header().set_host("www.example.com");

        // 发送请求
        send_request(NULL, 0);
        return true;
    }

    // 接收到响应头
    bool on_http_res_hdr(const acl::http_header& header) override {
        printf("状态码: %d\n", header.get_status());
        return true;
    }

    // 接收到响应体数据
    bool on_http_res_body(char* data, size_t dlen) override {
        printf("接收数据: %.*s\n", (int)dlen, data);
        return true;
    }

    // 响应接收完成
    bool on_http_res_finish(bool success) override {
        printf("请求完成: %s\n", success ? "成功" : "失败");
        return false;  // 返回false关闭连接
    }

    // 连接断开
    void on_disconnect() override {
        printf("连接已断开\n");
    }
};

// 使用示例
int main() {
    acl::aio_handle handle;

    MyHttpClient* client = new MyHttpClient(handle);

    // 连接服务器
    if (!client->open("www.example.com:80", 10, 10)) {
        printf("连接失败\n");
        delete client;
        return 1;
    }

    // 启动事件循环
    while (true) {
        handle.check();
    }
```

```
    return 0;
}
```

4. http_request_pool - 连接池

连接池可以复用连接，提高性能。

```
#include "acl_cpp/lib_acl.hpp"

// 创建连接池管理器
acl::connect_manager manager;

// 创建HTTP连接池
acl::http_request_pool* pool = new acl::http_request_pool(
    "www.example.com:80",  // 服务器地址
    100                    // 最大连接数
);

// 添加到管理器
manager.set(pool);

// 从连接池获取连接
acl::http_request* req = (acl::http_request*) pool->peek();
if (req == NULL) {
    printf("获取连接失败\n");
    return;
}

// 使用连接
req->request_header().reset();  // 重置状态
req->request_header().set_url("/api/test");
req->request_header().set_host("www.example.com");

if (req->request(NULL, 0)) {
    acl::string body;
    req->get_body(body);
    printf("响应: %s\n", body.c_str());
}

// 归还连接
pool->put(req, true);  // true表示保持连接
```

5. http_download - 文件下载

专门用于文件下载的类，支持断点续传。

```
#include "acl_cpp/lib_acl.hpp"

class MyDownloader : public acl::http_download {
```

```
public:
    MyDownloader(const char* url, const char* save_path)
        : http_download(url), save_path_(save_path) {
        fp_ = fopen(save_path, "wb");
    }

    ~MyDownloader() {
        if (fp_) fclose(fp_);
    }

protected:
    // 获取到文件大小
    bool on_length(long long int length) override {
        printf("文件大小: %lld bytes\n", length);
        return true;
    }

    // 接收数据
    bool on_save(const void* data, size_t len) override {
        if (fp_) {
            fwrite(data, 1, len, fp_);
            total_ += len;
            printf("已下载: %lld bytes\r", total_);
            fflush(stdout);
        }
        return true;
    }

private:
    std::string save_path_;
    FILE* fp_;
    long long total_ = 0;
};

// 使用示例
int main() {
    MyDownloader downloader(
        "http://example.com/files/large.zip",
        "/tmp/large.zip"
    );

    // 开始下载（支持断点续传）
    if (downloader.get(0, -1)) {
        printf("下载成功\n");
    } else {
        printf("下载失败\n");
    }

    return 0;
}
```

SSL/TLS支持

```
#include "acl_cpp/lib_acl.hpp"

// 创建SSL配置
acl::sslbase_conf* ssl_conf = new acl::mbedtls_conf(false);

// 创建HTTPS请求
acl::http_request req("www.example.com:443", 10, 10);

// 设置SSL
req.set_ssl(ssl_conf);

// 正常发送请求
req.request_header().set_url("/api/secure");
req.request(NULL, 0);

delete ssl_conf;
```

字符集转换

```
acl::http_request req("www.example.com:80");

// 设置本地字符集为GBK
req.set_local_charset("gbk");

req.request_header().set_url("/api/data");

if (req.request(NULL, 0)) {
    // 获取响应并自动转换为GBK
    acl::string body;
    req.get_body(body, "gbk");
}
```

GZIP压缩

```
acl::http_request req("www.example.com:80");

// 请求GZIP压缩
req.request_header().accept_gzip(true);

// 设置自动解压
req.set_unzip(true);

req.request_header().set_url("/api/large");

if (req.request(NULL, 0)) {
    // 响应会自动解压
    acl::string body;
```

```
    req.get_body(body);
}
```

超时设置

```
// 构造时设置超时
acl::http_request req(
    "www.example.com:80",
    10,  // 连接超时（秒）
    30   // 读写超时（秒）
);

// 或使用已有连接
acl::socket_stream conn;
conn.set_rw_timeout(30);  // 设置读写超时
conn.open("www.example.com:80", 10, 30);
```

重定向

```
acl::http_request req("www.example.com:80");

// 设置最大重定向次数
req.request_header().set_redirect(5);

req.request_header().set_url("/redirect");

if (req.request(NULL, 0)) {
    // 自动跟随重定向
    int redirect_count = req.request_header().get_redirect();
    printf("重定向次数: %d\n", redirect_count);
}
```

错误处理

```
acl::http_request req("www.example.com:80");

req.request_header().set_url("/api/test");

if (!req.request(NULL, 0)) {
    printf("请求失败\n");
    return;
}

// 检查HTTP状态码
int status = req.http_status();
if (status != 200) {
```

```
    printf("HTTP错误: %d\n", status);
    return;
}

// 检查连接状态
if (!req.keep_alive()) {
    printf("连接已关闭\n");
}
```

最佳实践

1. **使用连接池**: 对于频繁访问的服务器，使用连接池提高性能
2. **合理设置超时**: 根据网络环境和业务需求设置合理的超时时间
3. **复用对象**: 在循环中使用`reset()`方法复用http_request对象
4. **异步处理**: 高并发场景使用http_aclient异步客户端
5. **错误处理**: 完善的错误检查和异常处理
6. **内存管理**: 注意连接的生命周期管理
7. **SSL验证**: 生产环境启用证书验证

性能优化建议

1. 启用HTTP Keep-Alive
2. 使用连接池减少连接建立开销
3. 合理使用GZIP压缩
4. 异步模式处理大量并发
5. 流式处理大文件
6. 使用内存池优化内存分配

常见问题

Q: 如何处理HTTPS?

A: 使用`set_ssl()`方法设置SSL配置。

Q: 如何实现断点续传?

A: 使用`set_range()`设置Range头，或使用http_download类。

Q: 如何获取响应头信息?

A: 使用`header_value()`方法获取特定头部字段。

Q: 连接池如何设置大小?

A: 在创建http_request_pool时指定最大连接数。