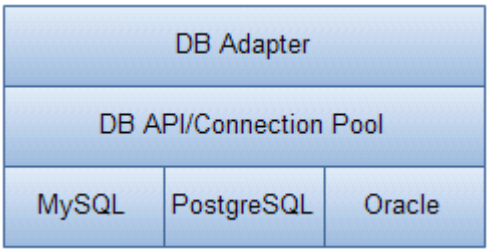
mooon db wrapper

一见

1. 前言

mooon db wrapper 不是一个 DB，仅是对现有的 DB API 的封装，使得使用更为简单。项目地址：http://code.google.com/p/mooon，可使用 SVN 下载最新代码。开发和交流论坛：http://bbs.hadoopor.com/index.php?gid=67，可了解项目最新动态。

2. 分层结构

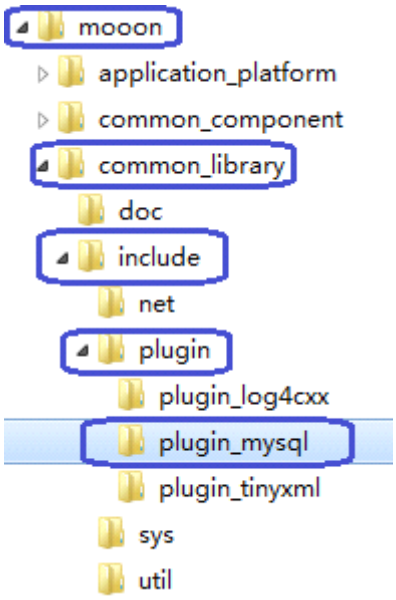


共分三层，最底层为各类数据库提供的 API，在它之上封装成 mooon db api，提供数据库连接池功能。mooon db API 本身与具体的数据库无关，通过不同的实现可支持不同的数据库，目前已经有 MySQL 的实现。

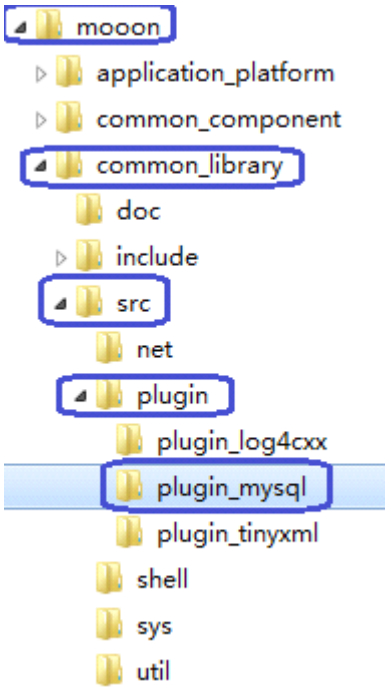
DB Adapter 是基于 DB API 的更高级抽象，内置数据库操作线程和队列，专门负责与数据库的读写交互，为上层应用提供一个异步回调的数据库操作，从而可解除应用和 DB 之间的一个强耦合。

3. 源码目录结构

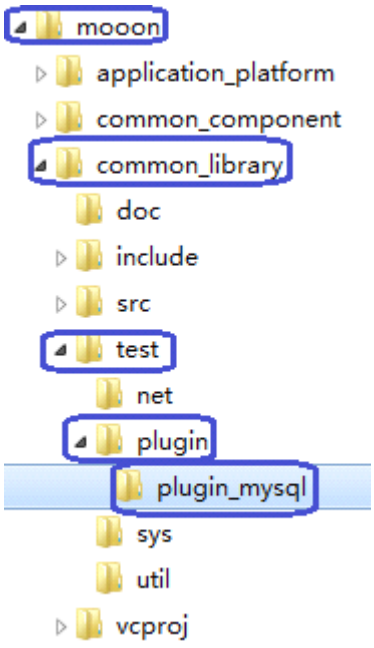
3.1. 头文件



3.2. CPP 文件



3.3. 测试代码



4. DB API

4.1. 异常类

数据库错误不能错误码的形式返回，而是采用异常的方式，可使得代码结构变得更为简洁。

4.1.1. CDBException

```
class CDBException
{
public:
    /***
      * 构造一个异常对象
      * 请注意不应当显示调用构造函数
      */
    CDBException(const char* sql, const char* error_message, int error_number=0, const char*
filename=__FILE__, int line_number=__LINE__);

    /** 返回执行出错的 SQL 语句，如果不是执行 SQL 语句，则仅返回一个字符串结尾符
*/
```

```
    const char* get_sql() const;

    /** 返回数据库的出错信息 */
    const char* get_error_message() const;

    /** 返回数据库的出错代码 */
    int get_error_number() const;

    /** 返回执行数据库操作时出错的文件名 */
    const char* get_filename() const;

    /** 返回执行数据库操作时出错的代码行 */
    int get_line_number() const;
};
```

4.2. 抽象接口

提供基础的数据库操作功能，而且也数据库无关，通过不同的实现，可支持不再的数据库。

4.2.1. IRecordrow

```
/***
  * 记录行接口
  */
class IRecordrow
{
public:
    /***
      * 通过字段编号取得字段的值
      */
    virtual const char* get_field_value(uint16_t index) const = 0;
};
```

4.2.2. IRecordset

```
/***
  * 记录集接口
  */
class IRecordset
{
public:
    /***
```

```
        * 得到记录集的行数
        * 对于 MySQL，如果 query 时，参数 is_stored 为 false，则该函数不能返回正确的值，
        * 所以应当只有在 is_stored 为 true，才使用该函数
        */
    virtual size_t get_row_number() const = 0;

    /***
        * 得到字段个数
        */
    virtual uint16_t get_field_number() const = 0;

    /***
        * 判断记录集是否为空
        */
    virtual bool is_empty() const = 0;

    /***
        * 检索结果集的下一行
        * @return: 如果没有要检索的行返回 NULL，否则返回指向记录行的指针，这时必须
调用 release_recordrow，否则有内存泄漏
        */
    virtual IRecordrow* get_next_recordrow() const = 0;

    /***
        * 释放 get_next_recordrow 得到的记录行
        */
    virtual void free_recordrow(IRecordrow* recordrow) = 0;
};
```

4.2.3. IDBConnection

```
/***
  * 数据库连接接口
  */
class IDBConnection
{
public:
    /** 是否允许自动提交 */
    virtual void enable_autocommit(bool enabled) = 0;

    /***
        * 用来判断数据库连接是否正建立着
        */
    virtual bool is_established() const = 0;
```

```
    /***
     * 数据库查询类操作，包括：select, show, describe, explain 和 check table 等
     * @is_stored: 是否将所有记录集拉到本地存储
     * @return: 如成功返回记录集的指针，这时必须调用 release_recordset，否则有内存泄
漏
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual IRecordset* query(bool is_stored, const char* format, ...) = 0;

    /***
     * 释放 query 得到的记录集
     */
    virtual void free_recordset(IRecordset* recordset) = 0;

    /***
     * 数据库 insert 和 update 更新操作
     * @return: 如成功返回受影响的记录个数
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual size_t update(const char* format, ...) = 0;
};
```

4.2.4. IDBConnectionPool

```
/***
 * 数据库连接池接口
 */
class IDBConnectionPool
{
public:
    /***
     * 得到全小写形式的数据库类型名，如：mysql 和 postgresql 等
     */
    virtual const char* get_type_name() const = 0;

    /***
     * 线程安全函数
     * 从数据库连接池中获取一个连接
     * @return: 如果当前无可用的连接，则返回 NULL，否则返回指向数据库连接的指针
     * @exception: 不会抛出任何异常
     */
    virtual IDBConnection* get_connection() = 0;
```

```
    /***
      * 线程安全函数
      * 将已经获取的数据库连接放回到数据库连接池中
      * @exception: 不会抛出任何异常
      */
    virtual void put_connection(IDBConnection* db_connection) = 0;

    /***
      * 创建连接池
      * @pool_size: 数据库连接池中的数据库连接个数
      * @db_ip: 需要连接的数据库 IP 地址
      * @db_port: 需要连接的数据库服务端口号
      * @db_name: 需要连接的数据库池
      * @db_user: 连接数据库用的用户名
      * @db_password: 连接数据库用的密码
      * @exception: 如出错抛出 CDBException 异常
      */
    virtual void create(uint16_t pool_size, const char* db_ip, uint16_t db_port, const char*
db_name, const char* db_user, const char* db_password) = 0;

    /***
      * 销毁已经创建的数据库连接池
      */
    virtual void destroy() = 0;

    /***
      * 得到连接池中的连接个数
      */
    virtual uint16_t get_connection_number() const = 0;
};
```

4.3. 助手类

强烈建议：
能使用助手类的时候尽可能地使用它，可以带来不必要的麻烦，而且可以简化代码结构。

4.3.1. DBConnectionHelper

```
/***
  * DB 连接助手类，用于自动释放已经获取的 DB 连接
  */
class DBConnectionHelper
```

```
{
public:
    DBConnectionHelper(IDBConnectionPool* db_connection_pool, IDBConnection*&
db_connection);

    ~DBConnectionHelper();
};
```

4.3.2. RecordsetHelper

```
/***
  * 记录集助手类，用于自动释放已经获取的记录集
  */
class RecordsetHelper
{
public:
    RecordsetHelper(IDBConnection* db_connection, IRecordset* recordset);
    ~RecordsetHelper();
};
```

4.3.3. RecordrowHelper

```
/***
  * 记录行助手类，用于自动释放已经获取的记录行
  */
class RecordrowHelper
{
public:
    RecordrowHelper(IRecordset* recordset, IRecordrow* recordrow);

    ~RecordrowHelper();
};
```

4.4. 示例

4.4.1. 源代码

```
#include "sys/db.h"
#include "plugin/plugin_mysql/plugin_mysql.h"
using namespace sys;
using namespace plugin;
```

```
int main()
{
    std::string sql = "SELECT * FROM test"; // 需要查询的 SQL 语句
    std::string db_ip = "127.0.0.1";
    std::string db_name = "test";
    std::string db_user = "root";
    std::string db_password = "";

    // create_mysql_connection_pool 和 destroy_mysql_connection_pool 两个全局函数
    // 在文件 plugin/plugin_mysql/plugin_mysql.h 中声明
    IDBConnectionPool* db_connection_pool = create_mysql_connection_pool();

    try
    {
        // 创建数据库连接池
        db_connection_pool->create(10, db_ip.c_str(), 3306, db_name.c_str(), db_user.c_str(),
db_password.c_str());
    }
    catch (sys::CDBException& ex)
    {
        fprintf(stderr, "Create database connection pool error: %s.\n", ex.get_error_message());
        exit(1);
    }

    do // 这个循环无实际意义，仅为简化代码结构
    {
        // 从数据库连接池中取一个连接
        IDBConnection* db_connection = db_connection_pool->get_connection();
        if (NULL == db_connection)
        {
            fprintf(stderr, "Database pool is empty.\n");
            break;
        }

        // 自动释放
        DBConnectionHelper db_connection_helper(db_connection_pool, db_connection);

        try
        {
            size_t row = 0; // 当前行数
            // 执行一条查询语句
            IRecordset* recordset = db_connection->query(false, "%s", sql.c_str());
            uint16_t field_number = recordset->get_field_number();
```

```
            // 自动释放
            RecordsetHelper recordset_helper(db_connection, recordset);

            for (;;)
            {
                // 取下一行记录
                IRecordrow* recordrow = recordset->get_next_recordrow();
                if (NULL == recordrow) break;

                // 自动释放
                RecordrowHelper recordrow_helper(recordset, recordrow);

                // 循环打印出所有字段值
                fprintf(stdout, "ROW[%04d] ==>\t", row++);
                for (uint16_t col=0; col<field_number; ++col)
                {
                    const char* field_value = recordrow->get_field_value(col);
                    fprintf(stdout, "%s\t", field_value);
                }
                fprintf(stdout, "\n");
            }
        }
        catch (sys::CDBException& ex)
        {
            fprintf(stderr, "Query %s error: %s.\n", ex.get_sql(), ex.get_error_message());
        }
    } while(false);

    // 销毁数据库连接池
    destroy_mysql_connection_pool(db_connection_pool);
}
```

4.4.2. 编译运行

进入$mooon/common_library/test/plugin/plugin_mysql 目录（其中$mooon 为 mooon 源代码所在目录），运行 Make 编译，成功后执行 run.sh 即可运行测试代码，如：sh run.sh。

5. DB Adapter

暂未实现。

6. 附录：如何编译 common 库？

编译测试代码之前，需要先编译好 common 库，common 库包含两大部分：基础类库和插件类库，它们的编译是自动进行的，而且会自动探测 MySQL 的安装目录。

要求 MySQL 安装在/usr/local 或用户主目录下，或者由环境变量 MYSQL_HOME 指定安装路径，而且 MySQL 的目录结构应当如下：

MySQL 安装目录

|---------include

|----------lib

在 include 目录下要求有 mysql.h 头文件，在 lib 目录下要求有 libmysqlclient_r.so 库文件。

当然，如果您不使用 mooon db wrapper，可以不用安装 MySQL，因为编译脚本会自动检测，如果没有安装，不会执行编译。

言归正传，先从 SVN 上取最新的 mooon common-library 代码，上传到 Linux 后，进入 src 目录，然后依次：

1) 执行 first_once.sh，该文件默认无执行权限，所以可 sh first_once.sh 方式执行（注：first_once.sh 只有在从 SVN 上取出代码时执行一次，这也是取 first_once 的意思）
2) 接下来，完全和普通的 automake 操作步骤一样，就不多说了
3) 编译好 mooon common-library 后，就可以进入 test 目录，编译测试代码了。在测试代码目录中，Makefile 是用来编译测试代码的，所以只需要执行 make 即可，而且 run.sh 是用来运行测试代码的，每一个 CPP 文件都会被编译成一个可执行的文件，所以必须保证目录下所有的 CPP 文件都实现了 main 函数。