FriendServer 设计

版权说明

对比特项目感兴趣，可以联系这个微信。

代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

功能设计

好友管理子服务，主要用于管理好友相关的数据与操作，因此主要负责以下接口：

1. 好友列表的获取：当用户登录成功之后，获取自己好友列表进行展示
2. 申请好友：搜索用户之后，点击申请好友，向对方发送好友申请
3. 待处理申请的获取：当用户登录成功之后，会获取离线的好友申请请求以待处理
4. 好友申请的处理：针对收到的好友申请进行同意/拒绝的处理
5. 删除好友：删除当前好友列表中的好友
6. 用户搜索：可以进行用户的搜索用于申请好友
7. 聊天会话列表的获取：每个单人/多人聊天都有一个聊天会话，在登录成功后可以获取聊天会话，查看历史的消息以及对方的各项信息
8. 多人聊天会话的创建：单人聊天会话在对方同意好友时创建，而多人会话需要调用该接口进行手动创建
9. 聊天成员列表的获取：多人聊天会话中，可以点击查看群成员按钮，查看群成员信息

模块划分

1. 参数/配置文件解析模块：基于 gflags 框架直接使用进行参数/配置文件解析。
2. 日志模块：基于 spdlog 框架封装的模块直接使用进行日志输出。
3. 服务注册模块：基于 etcd 框架封装的注册模块直接使用，进行聊天消息存储子服务的注册。
4. 数据库数据操作模块：基于 odb-mysql 数据管理封装的模块，实现数据库中数据的操作。
    a. 申请好友的时候，根据数据库中的数据判断两人是否已经是好友关系
    b. 申请好友的时候，根据数据库中的数据判断是否已经申请过好友
    c. 申请好友的时候，针对两位用户 ID 建立好友申请事件信息
    d. 好友信息处理的时候，找到申请事件，进行删除。
    e. 获取待处理好友申请事件的时候，从数据库根据用户 ID 查询出所有的申请信

息

f. 同意好友申请的时候，需要创建单聊会话，向数据库中插入会话信息

g. 从数据库根据指定用户 ID 获取所有好友 ID

h. 创建群聊的时候，需要创建群聊会话，向数据库中插入会话信息

i. 查看群聊成员的时候，从数据库根据会话 ID 获取所有会话成员 ID

j. 获取会话列表的时候，从数据库根据用户 ID 获取到所有会话信息

k. 删除好友的时候，从数据库中删除两人的好友关系，以及单聊会话，以及会话成员信息

5. rpc 服务模块：基于 brpc 框架搭建 rpc 服务器。

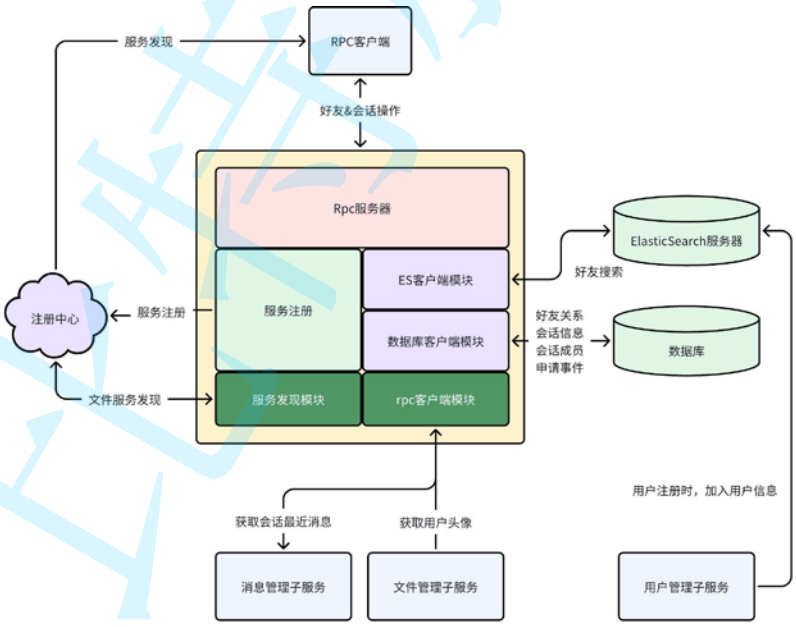
6. rpc 服务发现与调用模块：基于 etcd 框架与 brpc 框架封装的服务发现与调用模块，

a. 连接用户管理子服务：获取好友列表，会话成员，好友申请事件的时候获取用户信息。

b. 连接消息管理子服务：在打开聊天会话的时候，需要获取最近的一条消息进行展示。

7. ES 客户端模块：基于 elasticsearch 框架实现访问客户端，从 es 服务器进行用户的关键字搜索（用户信息由用户子服务在用户注册的时候添加进去）。

功能模块示意图：



数据管理

数据库数据管理：

根据好友相关操作分析，好友操作相关所需要有以下数据表：

用户信息表

该表由用户操作服务进行创建，并在用户注册时添加数据，好友这里只进行查询。

- 通过用户 ID 获取详细用户信息

用户关系表

> 因为本身用户服务器已经管理了用户个人信息，因此没必要再整一份用户信息出来，，也因为当前用户之间只有好友关系（目前未实现：黑名单，陌生人....），因此这里是一个好友关系表，表示谁和谁是好友
>
> 包含字段：
>
> - ID：作为主键
> - 用户 ID:
> - 好友 ID:
>
> 需要注意的是两个用户结为好友时，需要添加 (1,2),(2,1) 两条数据
>
> 提供的操作：
>
> - 新增用户关系
>   - 新增好友，通常伴随着新增会话，新增会话伴随着新增会话成员
> - 移除用户关系
>   - 移除好友，通常伴随着移除会话，移除会话伴随着移除会话成员
> - 判断两人是否是好友关系
> - 以用户 ID 获取用户的所有好友 ID
> - 与用户表连接，以用户 ID 获取所有好友详细信息

ODB 映射结构

```C++
#pragma once
#include <odb/core.hxx>
#include <odb/nullable.hxx>
```

```
#pragma db object
class friend_relation {
    public:
        friend_relation(){}
    private:
        friend class odb::access;
        #pragma db id auto
        long int _id;
        #pragma db index type("VARCHAR(127)")
        std::string _user_id;
        #pragma db type("VARCHAR(127)")
        std::string _friend_id;
};
```

会话信息

在多人聊天中，舍弃了群的概念，添加了聊天会话的概念，因为会话既可以是两人单聊会话，也可以是多人聊天会话，这样就可以统一管理了。

包含字段：

- ID：作为主键
- 会话 ID：会话标识
- 会话名称：单聊会话则设置为'单聊会话'或直接为空就行，因为单聊会话名称就是对方名称，头像就是对方头像
- 会话类型：SINGLE-单聊 / GROUP-多人（单聊由服务器在同意好友时创建，多人由用户申请创建）

提供的操作：

- 新增会话
  - 向会话成员表中新增会话成员信息
  - 向会话表中新增会话信息
- 删除会话
  - 删除会话成员表中的所有会话成员信息
  - 删除会话表中的会话信息
- 通过会话 ID，获取会话的详细信息
- 通过用户 ID 获取所有的好友单聊会话（连接会话成员表和用户表）

    - 所需字段：
        - 会话 ID
        - 会话名称：好友的昵称
        - 会话类型：单聊类型
        - 会话头像 ID：好友的头像 ID
        - 好友 ID：
- 通过用户 ID 获取所有自己的群聊会话（连接会话成员表和用户表）
    - 所需字段：
        - 会话 ID，
        - 会话名称
        - 会话类型：群聊类型

ODB 映射结构

```C++
#pragma once
#include <odb/core.hxx>
#include <odb/nullable.hxx>

enum class session_type_t {
    SINGLE = 1,
    GROUP = 2
};

#pragma db object
class chat_session {
    public:
        chat_session() {}
    private:
        friend class odb::access;
        #pragma db id auto
        long int _id;
        #pragma db unique type("VARCHAR(127)")
        std::string _session_id;
        #pragma db type("VARCHAR(127)")
        odb::nullable<std::string> _session_name;
        #pragma db type("TINYINT")
        session_type_t _session_type;
```

```
};
```

会话成员

> 每个会话中都会有两个及以上的成员，只有两个成员的会话是单聊会话，超过两个是多人聊天会话，为了明确哪个用户属于哪个会话，或者说会话中有哪些成员，因此需要有会话成员的数据管理
>
> 包含字段：
>
> - ID：作为主键
> - 会话 ID：会话标识
> - 用户 ID：用户标识
>
> 有了这张表就可以轻松的找出哪个用户属于哪个会话了，也可以根据会话 ID 获取所有成员 ID。
>
> 提供的操作：
>
> - 向指定会话中添加单个成员
> - 向指定会话中添加多个成员。
> - 从指定会话中删除单个成员
> - 通过会话 ID，获取会话的所有成员 ID
> - 删除会话所有成员：在删除会话的时候使用。

ODB 映射结构

```C++
#pragma once
#include <odb/core.hxx>
#include <odb/nullable.hxx>

#pragma db object
class chat_session_member
{
    public:
        chat_session_member (){}
    private:
        friend class odb::access;

        #pragma db id auto
```

```
        unsigned long _id;
        #pragma db index type("VARCHAR(127)")
        std::string _session_id;
        #pragma db type("VARCHAR(127)")
        std::string _user_id;
};
```

好友申请事件

> 在好友的操作中有个操作需要额外的管理，那就是申请好友的事件，因为用户 A 申请用户 B 为好友，并非一次性完成，需要用户 B 对本次申请进行处理，同意后才算是一次完整的流程。而在两次操作之间我们就需要为两次操作建立起相匹配的关系映射。
>
> 包含字段:
>
> - ID：作为主键
> - 事件 ID
> - 请求者用户 ID
> - 响应者用户 ID
> - 状态：用于表示本次请求的处理阶段，其包含三种状态：待处理-todo，同意-accept，拒绝-reject。
>
> 提供的操作:
>
> - 新增好友申请事件：申请的时候新增
> - 删除好友申请事件：处理完毕（同意/拒绝）的时候删除
> - 获取指定用户的所有待处理事件及关联申请者用户信息（连接用户表）

ODB 映射结构

```
C++
enum class fevent_status{
    PENDING = 1,
    ACCEPT = 2,
    REJECT = 3
};

#pragma db object
class friend_event {
```

```
    public:
        friend_event() {}
    private:
        friend class odb::access;
        #pragma db id auto
        long int _id;
        #pragma db unique type("VARCHAR(127)")
        std::string _event_id;
        #pragma db type("VARCHAR(127)")
        std::string _req_user_id;
        #pragma db type("VARCHAR(127)")
        std::string _rsp_user_id;
        #pragma db type("TINYINT")
        fevent_status _status;
};
```

ES 用户信息管理

创建用户索引

```
C++
POST /user/_doc
{
    "settings" : {
        "analysis" : {
            "analyzer" : {
                "ik" : {
                    "tokenizer" : "ik_max_word"
                }
            }
        }
    },
    "mappings" : {
        "dynamic" : true,
        "properties" : {
            "nickname" : {
                "type" : "text",
                "analyzer" : "ik_max_word"
            },
            "user_id" : {
                "type" : "keyword",
                "analyzer" : "standard"
            },
```

```
            "phone" : {
                "type" : "keyword",
                "analyzer" : "standard"
            },
            "description" : {
                "type" : "text",
                "index": "not_analyzed"
            },
            "avatar_id" : {
                "type" : "text",
                "index": "not_analyzed"
            }
        }
    }
}
```

新增测试数据

```
C++
POST /user/_doc/_bulk
{"index":{"_id":"1"}}
{"user_id" : "USER4b862aaa-2df8654a-7eb4bb65-
e3507f66","nickname" : "昵称 1","phone" : "手机号 1","description" :
"签名 1","avatar_id" : "头像 1"}
{"index":{"_id":"2"}}
{"user_id" : "USER14eeeaa5-442771b9-0262e455-
e4663d1d","nickname" : "昵称 2","phone" : "手机号 2","description" :
"签名 2","avatar_id" : "头像 2"}
{"index":{"_id":"3"}}
{"user_id" : "USER484a6734-03a124f0-996c169d-
d05c1869","nickname" : "昵称 3","phone" : "手机号 3","description" :
"签名 3","avatar_id" : "头像 3"}
{"index":{"_id":"4"}}
{"user_id" : "USER186ade83-4460d4a6-8c08068f-
83127b5d","nickname" : "昵称 4","phone" : "手机号 4","description" :
"签名 4","avatar_id" : "头像 4"}
{"index":{"_id":"5"}}
{"user_id" : "USER6f19d074-c33891cf-23bf5a83-
57189a19","nickname" : "昵称 5","phone" : "手机号 5","description" :
"签名 5","avatar_id" : "头像 5"}
{"index":{"_id":"6"}}
{"user_id" : "USER97605c64-9833ebb7-d0455353-
```

```
35a59195","nickname" : "昵称 6","phone" : "手机号 6","description" :
"签名 6","avatar_id" : "头像 6"}
```

进行搜索测试

```C++
GET /user/_doc/_search?pretty
{
    "query": {
        "match_all": {}
    }
}
```

```C++
GET /user/_doc/_search?pretty
{
    "query" : {
        "bool" : {
            "must_not" : [
                {
                    "terms" : {
                        "user_id.keyword" : [
                            "USER4b862aaa-2df8654a-7eb4bb65-
e3507f66",
                            "USER14eeeaa5-442771b9-0262e455-
e4663d1d",
                            "USER484a6734-03a124f0-996c169d-
d05c1869"
                        ]
                    }
                }
            ],
            "should" : [
                {
                    "match" : {
                        "user_id" : "昵称"
                    }
                },
                {
                    "match" : {
                        "nickname" : "昵称"
```

```
                    }
                },
                {
                    "match" : {
                        "phone" : "昵称"
                    }
                }
            ]
        }
    }
}
```

删除用户索引

```
C++
DELETE /user
```

接口实现流程

获取好友列表

1. 获取请求中的用户 ID
2. 根据用户 ID，从数据库的好友关系表&用户表中取出该用户所有的好友简息
3. 根据好友简息中的好友头像 ID，批量获取头像数据，组织完整用户信息结构
4. 组织响应，将好友列表返回给网关。

申请添加好友

1. 取出请求中的请求者 ID，和被请求者 ID
2. 判断两人是否已经是好友
3. 判断该用户是否已经申请过好友关系
4. 向好友申请事件表中，新增申请信息
5. 组织响应，将事件 ID 信息响应给网关

获取待处理好友申请事件

1. 取出请求中的用户 ID

2. 根据用户 ID，从申请事件表&用户表中找到该用户所有状态为 PENDING 的待处理事件关联申请人用户简息

3. 根据申请人用户头像 ID，从文件存储子服务器获取所有用户头像信息，组织用户信息结构

4. 组织响应，将申请事件列表响应给网关

处理好友申请

1. 取出请求中的申请人 ID，和被申请人 ID，以及处理结果

2. 根据两人 ID 在申请事件表中查询判断是否存在申请事件

3. 判断两人是否已经是好友（互相加好友的情况）

4. 不管拒绝还是同意，删除申请事件表中的事件信息（该事件处理完毕）

5. 若同意申请，则向用户关系表中添加好友关系数据，向会话表中新增会话信息，向会话成员表中新增成员信息

6. 组织响应，将新生成的会话 ID 响应给网关。

删除好友

1. 取出请求中的删除者 ID 和被删除者 ID

2. 从用户好友关系表中删除相关关系数据，从会话表中删除单聊会话，从会话成员表中删除会话成员信息

3. 组织响应，返回给网关

搜索好友

1. 取出请求中的用户 ID，和搜索关键字

2. 从好友关系表中取出该用户所有好友 ID

3. 根据关键字从 ES 服务器中进行用户搜索，搜索的时候需要将关键字作为用户 ID/手机号/昵称的搜索关键字进行搜索，且需要根据自己的 ID 和好友 ID 过滤掉自己和自己的好友。

4. 根据搜索到的用户简息中的头像 ID，从文件服务器批量获取用户头像数据

5. 组织响应，将搜索到的用户列表响应给网关

创建会话

1. 从请求中取出用户 ID 与会话名称，以及会话的成员 ID 列表

2. 生成会话 ID，并向会话表中新增会话信息数据，会话为群聊会话（单聊会话是同意好友申请的时候创建的）

3. 向会话成员表中新增所有的成员信息

4. 组织响应，将组织好的会话信息响应给网关。

获取会话列表

1. 从请求中取出用户 ID

2. 根据用户 ID，从会话表&会话成员表&用户表中取出好友的单聊会话列表（会话 ID，好友用户 ID，好友昵称，好友头像 ID），并组织会话信息结构对象

    a. 单聊会话中，对方的昵称就是会话名称，对方的头像就是会话头像，会话类型为单聊类型

3. 根据单聊会话 ID，从消息存储子服务获取会话的最后一条消息

4. 根据好友头像 ID，从文件存储子服务批量获取好友头像数据，

5. 组织好单聊会话结构数据

6. 根据用户 ID，从会话表&会话成员表中取出群聊会话列表（会话 ID，会话名称）

7. 根据群聊会话 ID，从消息存储子服务获取会话的最后一条消息

8. 组织好群聊会话结构数据

9. 将单聊会话数据和群聊会话数据组织到一起，响应给网关。

获取会话成员

1. 取出请求中用户 ID，和会话 ID

2. 根据会话 ID，从会话成员表&用户表中取出所有的成员用户信息

3. 根据成员信息中的头像 ID，从文件存储子服务批量获取头像数据组织用户信息结构

4. 组织响应，将会话的成员用户信息列表响应给网关