**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联201

姓名： 赵方程 学号： 42024137 实验日期： 2021 年 9 月 23 日

**实验名称：个人银行账户管理系统的改进**

**实验目的：**

完善银行账户系统，并实现以下功能：

(简述你的系统新增了哪些“银行1-6”没有的功能，每个功能用1句话简述)

1. 使用C++的异常处理机制来处理日期不合法，额度超出等错误。

2. 每个用户登陆后可以创建自己的两类账户，并进行操作

3. 每个用户登陆后需要提示该用户需要处理的信息,如还款的提示及当月的账户 统计，包括收入和支出的统计。

1. 可以查询某个月按照时间排序的账户查询信息。
2. 可以查询某个月按照交易金额从大到小排序的账户查询信息。
3. 直接存储用户数据,每次运行时从硬盘中读取
4. 记录了必要的错误日志,存储于BankSystemError.log

**实验仪器：**

计算机：ROG Zephyrus G14

CPU：2.9GHz AMD RYZEN R7 4800HS

内存：16GB 3200MHz DDR4

硬盘：512GB

显卡：NVIDIA RTX 2060 with max-q Design 6GB

操作系统：Windows 10

编译器：MSVC 16.11.3

C++语言标准: ISO C++20

**实验内容与步骤：**

**在实现具体功能前首先实现基础组件:**

sf::default\_contention\_free\_shared\_mutex: 一个C++11的读写锁

**(https://github.com/AlexeyAB/object\_threadsafe)**

**用于读写文件的基类**

class BankIOBase

{

protected:

//bind file with lock

static std::unordered\_map

<std::string, sf::default\_contention\_free\_shared\_mutex> file\_mutex;

static sf::default\_contention\_free\_shared\_mutex mapLock;

class MapGuard

{

private:

std::string name;

public:

MapGuard() = delete;

MapGuard(const MapGuard&) = delete;

auto operator =(const MapGuard&) = delete;

MapGuard(const std::string& name): name(name)

{

mapLock.lock();

file\_mutex[name].lock();

mapLock.unlock();

}

~MapGuard()

{

mapLock.lock();

file\_mutex[name].unlock();

mapLock.unlock();

}

};

class MapSharedGuard

{

private:

std::string name;

public:

MapSharedGuard() = delete;

MapSharedGuard(const MapSharedGuard&) = delete;

auto operator =(const MapSharedGuard&) = delete;

MapSharedGuard(const std::string& name) : name(name)

{

mapLock.lock();

file\_mutex[name].lock\_shared();

mapLock.unlock();

}

~MapSharedGuard()

{

mapLock.lock();

file\_mutex[name].unlock\_shared();

mapLock.unlock();

}

};

std::fstream fs{};

std::string name;

BankIOBase() = default;

explicit BankIOBase(std::string name) : name(std::move(name))

{

}

public:

BankIOBase(const BankIOBase&) = delete;

BankIOBase(BankIOBase&&) = delete;

virtual ~BankIOBase() = default;};

**用于保存/加载文件的派生类(s)**

class Saver : public BankIOBase

{

private:

static bool status;

std::ios::openmode mode;

public:

explicit Saver(std::string fileName, const std::ios::openmode& mode = std::ios::app) :

BankIOBase(std::move(fileName)),

mode(mode)

{

}

Saver(const Saver&) = delete;

auto operator=(const Saver&) -> void = delete;

~Saver() override = default;

auto save(const std::string& msg) -> void

{

if (!status)

{

return;

}

MapSharedGuard guard(name);

fs.open(name, mode);

if (fs.is\_open())

{

fs << '\n' << msg << '\n';

}

fs.close();

}

auto save(const char\* msg) -> void

{

if (!status)

{

return;

}

MapSharedGuard guard(name);

fs.open(name, mode);

if (fs.is\_open())

{

fs << '\n' << msg << '\n';

}

fs.close();

}

auto save(const std::vector<std::string>& vMsg) -> void

{

if (!status)

{

return;

}

MapSharedGuard guard(name);

fs.open(name, mode);

for (const auto& i : vMsg)

{

fs << '\n' << i << '\n';

}

fs.close();

}

static auto statusOn() -> void

{

status = true;

}

static auto statusOff() -> void

{

status = false;

}

static auto Status() -> bool

{

return status;

}

};

class Loader : public BankIOBase

{

public:

explicit Loader(std::string fileName) : BankIOBase(std::move(fileName))

{

}

Loader(const Loader&) = delete;

auto operator=(const Loader&) -> void = delete;

~Loader() override = default;

auto load() -> std::vector<std::string>

{

MapSharedGuard guard(name);

std::vector<std::string> buffer;

buffer.reserve(20);

fs.open(name, std::ios::in);

if (fs.is\_open())

{

std::string temp;

while (fs >> temp)

{

buffer.emplace\_back(temp);

}

}

fs.close();

return buffer;

}

auto load(std::vector<std::string>& buffer) -> void

{

buffer.reserve(20);

MapSharedGuard guard(name);

fs.open(name, std::ios::in);

if (fs.is\_open())

{

std::string temp;

while (fs >> temp)

{

buffer.emplace\_back(temp);

}

}

fs.close();

}

auto loadByLine() -> std::vector<std::string>

{

std::vector<std::string> buffer;

buffer.reserve(20);

MapSharedGuard guard(name);

fs.open(name, std::ios::in);

if (fs.is\_open())

{

std::string temp;

while (std::getline(fs, temp))

{

buffer.emplace\_back(temp);

}

}

fs.close();

return buffer;

}

auto loadOneLine() -> std::string

{

std::string content;

MapSharedGuard guard(name);

fs.open(name, std::ios::in);

if (fs.is\_open())

{

std::string temp;

std::getline(fs, content);

}

fs.close();

return content;

}

};

**储存/加载账户信息的函数(s)**

**saveAccounts()**

**loadAccounts()**

**实现代码比较长,详见附件**

**【**功能1：建立简单的储蓄账户**】**

1.设计

数据成员:

储蓄账户拥有id, balance, rate, account\_record 等基本属性

利息计算由成员 accumulator 实现

成员函数:

Record 记录属性

settel

Getters/Setters

Query(s)

1. 实现
2. 功能主要有：

建立账户,设置密码,显示信息，存款，取款，结算利息等

声明

|  |  |
| --- | --- |
| using AccountMode = int;  class Account  {  private:  std::string password;  protected:  const std::string id;  const Date initialDate;  std::atomic<double> balance;  std::atomic<double> rate;  static std::atomic<double> total;  static std::multiset<AccountRecord> recordSet;//记录全体账单  static sf::default\_contention\_free\_shared\_mutex recordSetMutex;  std::shared\_ptr<Accumulator> accumulator;//计算利息  std::multiset<AccountRecord> myRecordSet;//记录本账户账单  sf::default\_contention\_free\_shared\_mutex myRecordSetMutex;  Account(const Date& date, const std::string& id, const double& rate, const std::string& password,   AccountMode mode);  virtual auto checkSufficiency(const double& amount) -> bool = 0;//检查余额  auto record(const Date& date, const double& amount, const char\* msg) -> void;  static auto error(const std::string& msg) -> void;  auto getPassword() const -> std::string;  public:  Account() = delete;  Account(const Account&) = delete;  auto operator =(const Account&) = delete;  virtual ~Account() = default;  auto setPassword(const std::string& newPass) -> void;  //function todo change Password  auto checkPassword(const std::string& password) const -> bool;  auto getId() const noexcept -> std::string;  auto getBalance() const noexcept -> double;  auto getRate() const noexcept -> double;  auto setBalance(const double& balance) -> void;  virtual auto info() const -> std::string =0;  virtual auto saveInfo() const -> std::string = 0;//保存所需的基本信息  virtual auto show(std::ostream& os = std::cout) const -> void =0;//显示账户  virtual auto deposit(const Date& date, const double& amount, const char\* msg = nullptr) -> void =0;//存款  virtual auto withdraw(const Date& date, const double& amount, const char\* msg = nullptr) -> void =0;//取款  virtual auto settle(const Date& date) -> void =0;;//结算  virtual auto notice() -> std::string;//登录提示  static auto getTotal() noexcept -> double;  static auto addTotal(const double& value) -> void;  static auto query(const Date& date1, const Date& date2) -> std::string;//查询  static auto insertIntoTotalRecord(const AccountRecord& ar) -> void;//添加至总账单  static auto insertIntoTotalRecord(const Date& date, const std::string& accountID, const double& amount,                                     const double& balance,const std::string& desc) -> void; auto insertIntoMyRecord(const AccountRecord& ar) -> void;//添加至本账户账单  auto insertIntoMyRecord(const Date& date, const std::string& accountID, const double& amount,  const double& balance, const std::string& desc) -> void;  auto getMyAccountRecord() -> std::vector<AccountRecord>;  auto getMyAccountRecord(std::vector<AccountRecord>& myAccountRecord) -> void;  auto query\_this(const Date& d1, const Date& d2) -> std::string;//查询本账户账单  auto query\_this\_by\_order(const Date& d1,                           const Date& d2,                           const std::function<bool(const AccountRecord&, const AccountRecord&)>& order)  -> std::vector<AccountRecord>;//按某顺序查询账单  }; |  |

代码片段1 账户类的声明

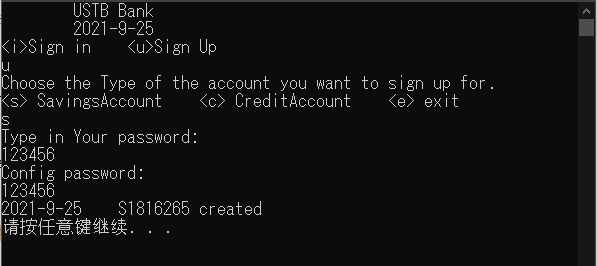
部分实现

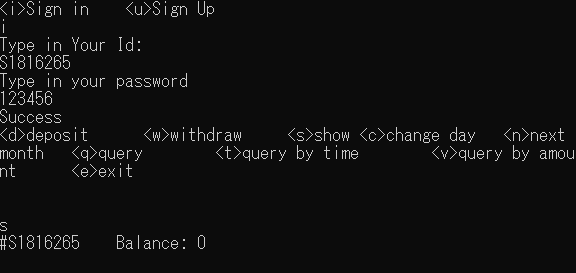
|  |
| --- |
| /\* \* 保存数据，输出msg，amount区分正负 \* recordSet插入失败时抛出异常,恢复record前的状态 \*/  auto Account::record(const Date& date, const double& amount, const char\* msg) -> void  {  const auto newBalance = getBalance() + amount;  setBalance(newBalance);  total = total + amount;    try  {  recordSetMutex.lock();  const AccountRecord ar(date, id, amount, getBalance(), msg);  recordSet.insert(ar);  recordSetMutex.unlock();  insertIntoMyRecord(ar);  Saver(id + ".txt").save(ar.to\_string() + "\n");  }  catch (const std::exception&) //注意异常安全  {  recordSetMutex.unlock();  setBalance(getBalance() - amount);  total = total - amount;  throw AccountException(std::string("failed,recordSet insert error").append("    ").append(this->info()));  }  std::cout << date<< "    #" << this->id << "    " << amount << "    " << getBalance() << "     " << msg << '\n';  }  //获取登录时信息  auto Account::notice() -> std::string  {  const auto sysDate = DATE::getSystemDate();  const auto d1 = getDate(sysDate.getYear(), sysDate.getMonth(), 1);    const auto notice = this->query\_this\_by\_order(d1, sysDate, AccountRecord::ascendAmountComparator());  std::string income, outcome;  for (const auto& ar : notice)  {  if (ar.getAmount() > 0)  {  income += ar.to\_string() + "\n";  }  else  {  outcome += ar.to\_string() + "\n";  }  }    return income + "\n\n" + outcome;  }  /\*  \* 返回 string  \* 查询到的信息  \*/  auto Account::query(const Date& date1, const Date& date2) -> std::string  {  bool found = false;  std::string result;  recordSetMutex.lock\_shared();  for (const auto& i : recordSet)  {  if ((i.getDate() >= date1) && (i.getDate() <= date2))  {  found = true;  result += i.to\_string().append("\n");  }  }  recordSetMutex.unlock\_shared();  if (!found)  {  result = "Not Found";  }  return result;  }  //添加至总账单  auto Account::insertIntoTotalRecord(const AccountRecord& ar) -> void  {  recordSetMutex.lock();  try  {  recordSet.insert(ar);  }  catch (...)  {  recordSetMutex.unlock();  throw;  }    recordSetMutex.unlock();  }  //插入本账户账单  auto Account::insertIntoMyRecord(const AccountRecord& ar) -> void  {  myRecordSetMutex.lock();  try  {  myRecordSet.insert(ar);  }  catch (...)  {  myRecordSetMutex.unlock();  throw;  }    myRecordSetMutex.unlock();  }  //获取本账户账单  auto Account::getMyAccountRecord() -> std::vector<AccountRecord>  {  std::vector<AccountRecord> myAccountRecord;  myAccountRecord.reserve(20);  try  {  myRecordSetMutex.lock\_shared();  for (const auto& i : myRecordSet)  {  myAccountRecord.emplace\_back(i);  }  myRecordSetMutex.unlock\_shared();  }  catch (const std::exception&)  {  myRecordSetMutex.unlock\_shared();  throw;  }    return myAccountRecord;  }  //以某规则查询账单  auto Account::query\_this\_by\_order(const Date& d1, const Date& d2,                                    const std::function<bool(const AccountRecord&,                                                             const AccountRecord&)>& order) -> std::vector<AccountRecord>  {  auto&& arVector = getMyAccountRecord();    std::vector<AccountRecord> res;  res.reserve(20);  for (const auto& ar : arVector)  {  if ((ar.getDate() >= d1) && (ar.getDate() <= d2))  {  res.emplace\_back(ar);  }  }    std::sort(res.begin(), res.begin() + res.size(), order);  return res;  } |
| - |

3.测试

（1）测试用例1：u s 123456 123456（用户输入正确数据）

得到的运行截图如下：

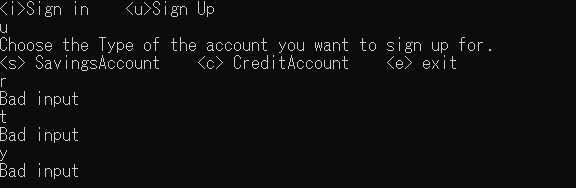




测试了创建账户的过程,输出的结果表示创建了一个S1816265的账户,密码为123456

考虑到现实中基本不存在自定义利率以及自定义ID的情况,故设定利率为0.015,ID由id generator自动生成.

1. 测试用例2：用户输入了错误的选项



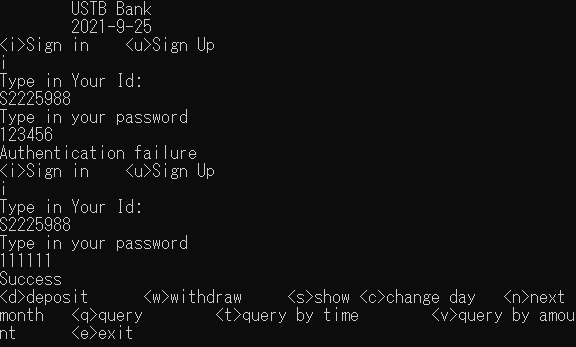
用户输入错误选项,报错提示

1. 测试用例3: 用户输入了不符合规则的密码

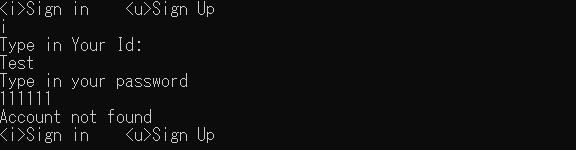


用户输入不符合规定的密码,或密码校验失败,报错

1. 测试用例4: 用户输入错误的密码

 密码不一致,报错

1. 测试用例5: 用户输入错误的账户



账户不存在,报错

**【**功能2：使用C++异常处理机制来处理日期不合法,额度超出等错误**】**

1. 日期非法处理

由于在构造函数中最好不要抛出异常(影响对象资源的释放),故Date的构造函数均为noexcept

那么在处理日期非法的时候需要引入工厂函数getDate,在此函数中判断日期是否合法,合法则构造Date对象并返回,否则抛出异常并在外层进行处理或者继续向外抛出

/\*    返回Date类时间,不合法则抛出异常 \*/

inline auto getDate(const int& year, const int& month, const int& day) -> Date

{

if (!Date::isValid(year, month, day))

{

throw std::runtime\_error("invalid date");

}

return {year, month, day};

}

至于如何判断日期的合法性(即isValid的实现),见下

auto Date::isValid(const int& year, const int& month, const int& day) -> bool

{

if (day < 1 || month > 12 || month < 1)

{

return false;

}

int d = DATE::dayOfMonth[month - 1];

if (month == 2 && isLeapYear(year))

{

++d;

}

return d >= day;

}

1. 余额不足处理

每次取款前判断额度是否充足,若不足则抛出异常

/\* \* 取款，amount为正，余额或者信用卡额度不足时抛出异常  \*/

auto CreditAccount::withdraw(const Date&date,const double&amount,const char\*msg)->void

{

if (!checkSufficiency(amount))

{

throw std::runtime\_error("insufficient balance");

}

record(date, -amount, msg);

this->accumulator->calculate(date, rate, -amount);

}

auto SavingsAccount::withdraw(const Date&date,const double&amount,const char\*msg)->void

{

if (!checkSufficiency(amount))

{

throw std::runtime\_error("insufficient balance");

}

record(date, -amount, msg);

this->accumulator->calculate(date, rate, -amount);

}

结算时,若余额不足抛出异常

/\* \* 结算利息，判断是否需要交纳年费 ，余额不足抛出异常  \*/

auto CreditAccount::settle(const Date& date) -> void

{

if (date.getDay() != 1) //不是每月一号，不结算

{

return;

}

this->accumulator->calculate(date, rate, 0);

if (fabs(accumulator->getSum()) > 1e-6)

{

record(date, accumulator->getInterest(), "interest");

}

if (date.getMonth() == 1 && date.getDay() == 1) //一月一号扣年费

{

record(date, -fee, "annual fee");

}

accumulator->reset(date, balance);

if (!checkSufficiency())

{

throw std::runtime\_error("out of credit");

}

if (!checkSufficiency(0))

{

throw std::runtime\_error("insufficient balance");

}

}

1. 其他

除了上述两种情况需要异常处理,还有很大一部分用户输入错误情况需要使用异常处理,比如 输入数字时输入了其他字符, 命令参数数量错误等需要在其他地方进行异常处理.

例如:

inline auto query\_by\_order\_time(const std::vector<std::string>& parameters) ->Command\*

{

const auto size = parameters.size();

std::string id, password;

Date date1, date2;

if (size == 2) //date1 date2

{

date1 = getDate(parameters.at(0), CommandCreator::getDateSplit());

date2 = getDate(parameters.at(1), CommandCreator::getDateSplit());

}

else if (size == 4) //id password date1 date2

{

id = parameters.at(0);

password = parameters.at(1);

date1 = getDate(parameters.at(2), CommandCreator::getDateSplit());

date2 = getDate(parameters.at(3), CommandCreator::getDateSplit());

}

else

{

throw std::runtime\_error("Bad input");//参数数量错误

}

if (date1 > date2)

{

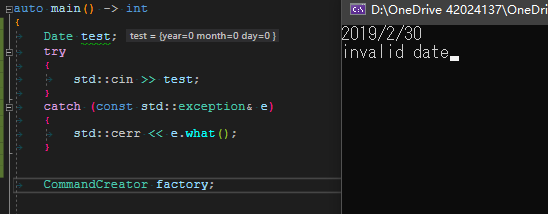
throw std::runtime\_error("invalid date range");

}

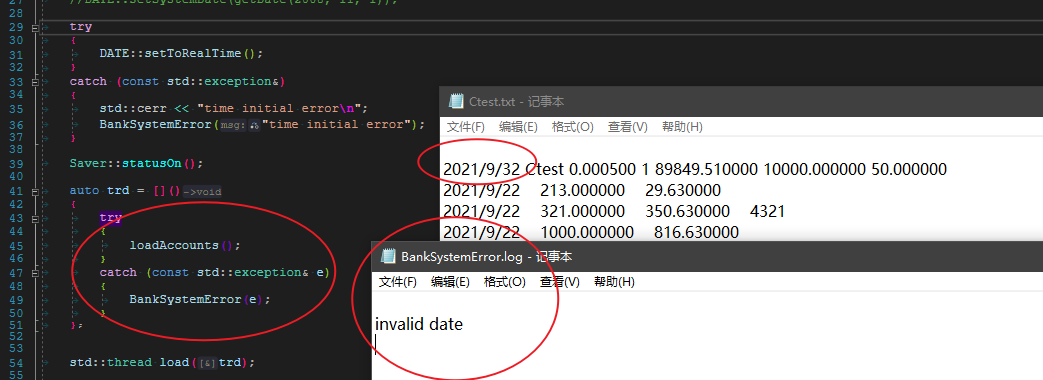
return new QueryBy(date1, date2, AccountRecord::ascendDateComparator(), id, password);

}

测试用例1: 输入错误时间



测试用例2: 记录的数据有误,读取

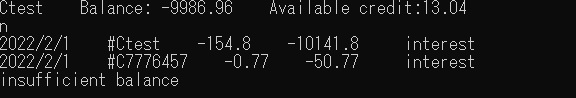


考虑到系统读取时间等操作对用户来说是没有感知的,所以单独设置了测试用例

测试用例3: 取款时余额不足



测试用例4: 结算利息时余额不足



**【**功能3：每个用户登陆后需要提示该用户需要处理的信息,如还款的提示及当月的账户 统计，包括收入和支出的统计。**】**

Account基类提供方法即可

auto Account::notice() -> std::string

{

const auto sysDate = DATE::getSystemDate();

const auto d1 = getDate(sysDate.getYear(), sysDate.getMonth(), 1);

const auto notice = this->query\_this\_by\_order

(d1,sysDate,AccountRecord::ascendAmountComparator());

std::string income, outcome;

for (const auto& ar : notice)

{

if (ar.getAmount() > 0)

{

income += ar.to\_string() + "\n";

}

else

{

outcome += ar.to\_string() + "\n";

}

}

return income + "\n\n" + outcome;

}

Override

auto CreditAccount::notice() -> std::string

{

std::string payment;

if (getAvailableCredit() <= 0)

{

payment += "Insufficient credit\n";

}

if (getDebt())

{

payment += "Debt: " + std::to\_string(getDebt()) + "\n";

}

return payment + Account::notice();

}

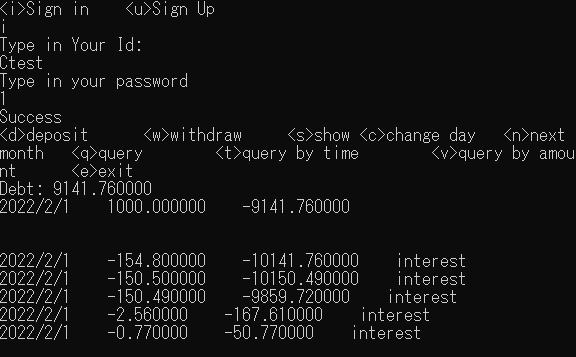
auto SavingsAccount::notice() -> std::string

{

return Account::notice();

}

测试用例: 登录账户



【功能4：查询按时间排序的账单信息】

提供查询函数即可

auto Account::query\_this\_by\_order(const Date& d1, const Date& d2,

const std::function<bool(const AccountRecord&,const AccountRecord&)>& order)

-> std::vector<AccountRecord>

{

auto&& arVector = getMyAccountRecord();

std::vector<AccountRecord> res;

res.reserve(20);

for (const auto& ar : arVector)

{

if ((ar.getDate() >= d1) && (ar.getDate() <= d2))

{

res.emplace\_back(ar);

}

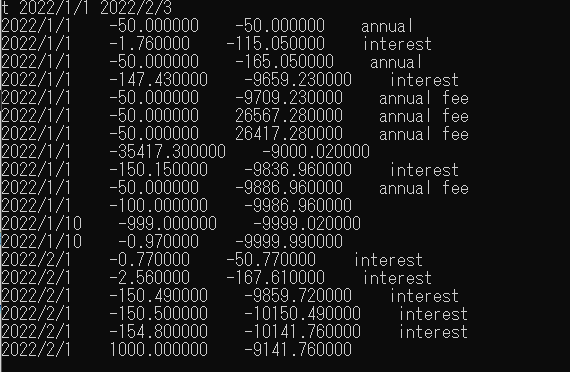
}

std::sort(res.begin(), res.begin() + res.size(), order);

return res;

}

测试用例:

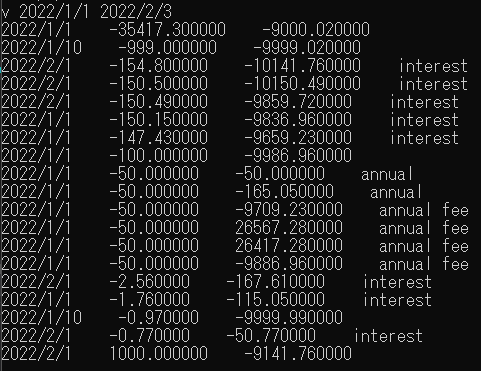


【功能5：查询按交易金额排序的账单信息】

实现同上

只需要在调用时使第三个参数为按交易金额排序的函数或lambda即可

测试用例:



【功能6：直接储存用户数据,每次运行时从硬盘中读取】

详见基础组件部分的loadAccount及saveAccount

【功能7：记录必要的错误日志】

/\*

发送错误信息至BankSystemError.log

\*/

inline auto BankSystemError(const std::exception& exc) -> void

{

const bool status = Saver::Status();

Saver::statusOn();

Saver("BankSystemError.log").save(exc.what());

status ? Saver::statusOn() : Saver::statusOff();

}

inline auto BankSystemError(const std::string& msg) -> void

{

const bool status = Saver::Status();

Saver::statusOn();

Saver("BankSystemError.log").save(msg);

status ? Saver::statusOn() : Saver::statusOff(); // **issue**

}

此处Saver的status设计很不好,有机会重构的话再改吧

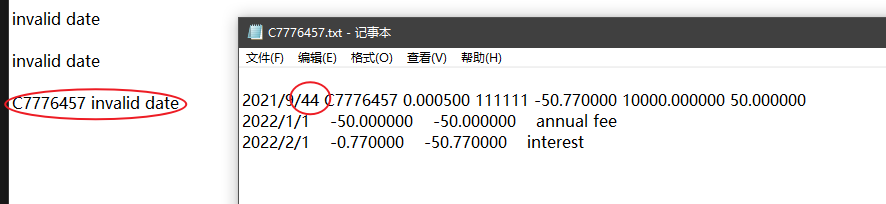
因为当时step\_6时为了复用代码,从Command.txt 和 std::cin 里读取的命令都从CommandCreator 里执行,但是这样的话从Command.txt读取的命令仍会保存到文件中,所以设置了一个saver的static变量来标识是否保存.

但是在多线程的情况下只要有另一个saver确实需要保存,这里就会影响到另一个saver,使之也无法保存.

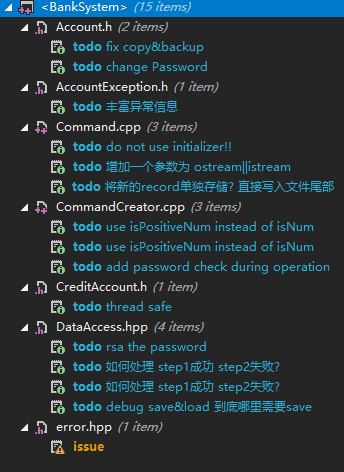
但考虑到服务开始时基本不会有其他的线程,所以暂时先这样设计.

测试用例:

用户一般的操作不会引起较为严重的错误,只有当记录的数据有误时会引发较为严重的错误,所以测试时人为修改了记录的数据

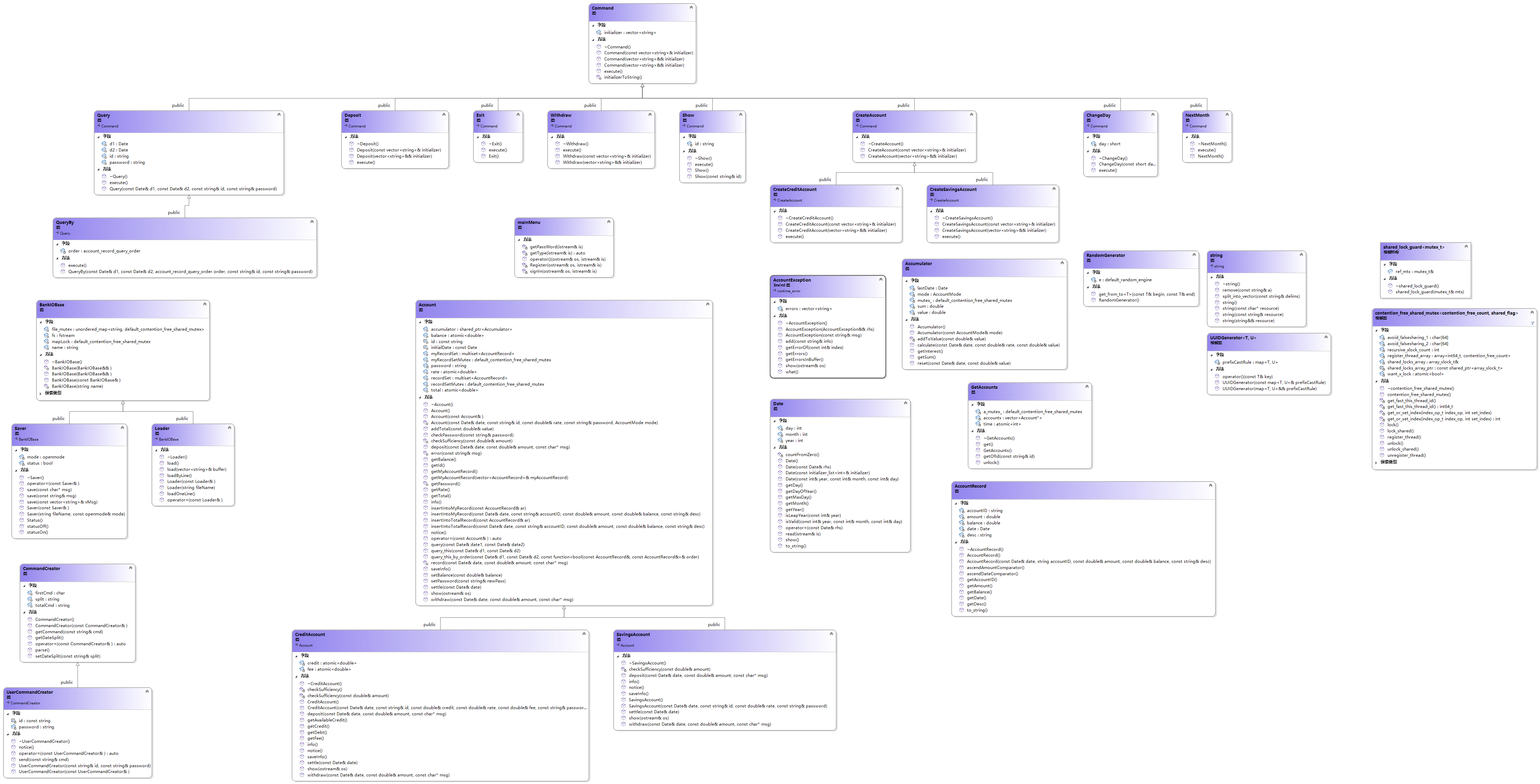


**实验结论与感想**

Todo-list:

最好可以加密数据,至少不能明文存储密码吧(啊喂

或许后续还能加个转账功能?



类图