数据结构课程设计报告

银行业务模拟系统

学　院: 计算机学院
专　业: 计算机科学与技术
年　级: 2022 级
学　号: 3124009862
学生姓名: 杨恒熠
指导教师: 李小妹
提交日期: 2025 年 11 月

目录

1 问题描述

根据《数据结构实验指导书》题目 4 的要求，设计并实现一个银行业务的事件驱动模拟系统，通过模拟方法求出客户在银行内逗留的平均时间。

1.1 业务规则

1. 客户业务分为两种：
   - 第一种：申请从银行得到一笔资金（取款或借款）
   - 第二种：向银行投入一笔资金（存款或还款）
2. 银行有两个服务窗口，相应地有两个队列：
   - 客户到达银行后先排第一个队
   - 处理每个客户业务时，如果属于第一种业务且申请额超出银行现存资金总额而得不到满足，则立刻排入第二个队等候，直至满足时才离开银行；否则业务处理完后立刻离开银行
3. 每接待完一个第二种业务的客户，则顺序检查和处理第二个队列中的客户，对能满足的申请者予以满足，不能满足者重新排到第二个队列的队尾
4. 营业时间结束时所有客户立即离开银行

2 系统设计

2.1 架构设计

系统采用事件驱动架构，通过事件队列管理所有客户到达和离开事件，按时间顺序处理。系统主要包含以下几个模块：

1. **事件管理模块**：负责事件的创建、排序和调度
2. **队列管理模块**：管理客户队列，包括正在等待服务的客户队列和因资金不足而等待的客户队列
3. **客户管理模块**：负责客户的创建、销毁和业务处理
4. **银行系统模块**：协调各个模块，维护银行状态
5. **日志系统模块**：记录系统运行过程中的关键信息

2.2 数据结构设计

2.2.1 客户结构体

```
typedef struct Customer {
    int id;                 // 客户ID
    int arrive_time;        // 到达时间
    int duration;           // 服务时间
    int amount;             // 交易金额（正数表示存款，负数表示取款）
    struct Customer* next;  // 链表指针
} Customer;
```

Listing 1: 客户结构体

2.2.2 事件结构体

```
typedef struct Event {
    int occur_time;             // 事件发生时间
    EventType event_type;       // 事件类型
    struct Customer* customer;  // 相关客户
    struct Event* next;         // 链表指针
} Event;
```

Listing 2: 事件结构体

2.2.3 队列结构体

```
typedef struct Queue {
    Customer* front;  // 队首指针
    Customer* rear;   // 队尾指针
    int size;         // 队列大小
} Queue;
```

Listing 3: 队列结构体

2.2.4 银行系统结构体

```
typedef struct BankSystem {
    int total_money;        // 银行总资金
    int current_time;       // 当前时间
    int close_time;         // 关门时间
    int customer_count;     // 客户计数器
    Queue queue1;           // 第一个队列（正在等待服务的客户）
    Queue queue2;           // 第二个队列（因资金不足而等待的客户）
```

```
    Event* event_list;      // 事件列表
    int total_customers;    // 总客户数
    long long total_wait_time;  // 总等待时间
    FILE* log_fp;           // 日志文件指针
    LogLevel log_level;     // 日志级别
} BankSystem;
```

Listing 4: 银行系统结构体

2.3 核心算法

2.3.1 事件驱动模拟

系统采用事件驱动的方式进行模拟，主要处理两类事件：

1. 客户到达事件
2. 客户离开事件

事件按照发生时间顺序存储在事件列表中，每次取出最早发生的事件进行处理。

2.3.2 客户服务流程

1. 客户到达时，加入第一个队列
2. 队列首客户接受服务，创建离开事件
3. 根据业务类型和银行资金情况决定客户去向：
    - 存款业务：直接完成，更新银行资金
    - 取款业务：资金充足则完成，资金不足则加入等待队列
4. 存款业务完成后，检查并处理等待队列中的客户

2.3.3 等待队列处理

每当有存款业务完成时，系统会检查等待队列中的客户：

1. 按顺序检查每个客户是否可以获得服务
2. 能满足的客户完成交易
3. 不能满足的客户重新排队
4. 当银行资金不再增加或处理一定数量客户后停止检查

3 系统实现

3.1 错误处理机制

系统采用统一的错误码机制，所有函数返回值都遵循以下规范：

- BANK_SUCCESS: 操作成功
- BANK_ERROR_INVALID_PARAM: 无效参数
- BANK_ERROR_MEMORY_ALLOC: 内存分配失败
- BANK_ERROR_FILE_OP: 文件操作失败
- BANK_ERROR_QUEUE_OP: 队列操作失败
- BANK_ERROR_EVENT_OP: 事件操作失败

3.2 日志系统

系统实现了分级日志系统，支持 DEBUG、INFO、WARN、ERROR 四个级别，可通过配置文件或命令行参数控制日志级别和输出位置。

3.3 配置管理

系统支持通过配置文件进行参数配置，配置文件格式如下：
“‘ 银行模拟系统配置文件
初始资金 $\text{initial}_money = 10000$
营业时间（分钟）$\text{close}_time = 600$
是否启用日志 $\text{enable}_log = 1$
日志级别 (0=DEBUG, 1=INFO, 2=WARN, 3=ERROR) $\log_level = 1$
日志文件路径 $\log_file = bank_simulation.log$

3.4 主要功能模块

3.4.1 队列操作

- init_queue: 初始化队列
- is_queue_empty: 检查队列是否为空
- get_queue_size: 获取队列大小
- enqueue: 入队操作

- dequeue: 出队操作
- queue_front: 获取队首元素

3.4.2 事件管理

- insert_event: 插入事件（按时间顺序）
- get_next_event: 获取下一个事件

3.4.3 客户管理

- create_customer: 创建随机客户
- destroy_customer: 销毁客户
- handle_arrive_event: 处理客户到达事件
- handle_leave_event: 处理客户离开事件
- process_queue2: 处理等待队列

3.4.4 系统控制

- init_bank_system: 初始化银行系统
- destroy_bank_system: 销毁银行系统
- run_simulation: 运行模拟
- print_statistics: 打印统计信息
- bank_log: 写日志
- load_config: 加载配置文件

3.5 关键处理逻辑

3.5.1 客户到达处理

1. 创建客户对象，随机生成服务时间和交易金额
2. 将客户加入第一个队列
3. 生成下一个客户到达事件

3.5.2 客户离开处理

1. 计算客户逗留时间并累加
2. 根据业务类型处理：
   - 存款：增加银行资金，处理等待队列
   - 取款：检查资金是否充足，充足则完成，不足则加入等待队列

3.5.3 等待队列处理

1. 按顺序检查等待队列中的客户
2. 资金充足的客户完成交易
3. 资金仍不足的客户重新排队
4. 适当条件下停止处理以避免无限循环

4 测试与分析

4.1 测试方案

4.1.1 测试数据设置

- 初始资金：10000 元
- 营业时间：600 分钟
- 客户到达间隔：1-10 分钟随机
- 服务时间：1-20 分钟随机
- 交易金额：-5000 到 5000 元随机

4.1.2 极端情况测试

1. 客户到达间隔短，服务时间长
2. 客户到达间隔长，服务时间短

4.2 运行结果

经过测试运行，系统能够正确处理所有客户事件，准确计算客户平均逗留时间，并合理处理资金不足情况。

4.2.1 典型运行结果

```
Bank Simulation System
======================
Initial money: 10000
Close time: 600 minutes
Logging enabled, level: 0, file: bank_simulation.log

Running simulation...
========================
Simulation completed, statistics:
Total customers: 114
Total wait time: 33797 minutes
Average stay time: 296.46 minutes
Final bank money: 10000 yuan
Remaining customers in waiting queue: 0
```

4.3 性能分析

4.3.1 时间复杂度

- 事件插入操作：$O(n)$，其中 n 为当前事件列表长度
- 事件获取操作：$O(1)$
- 队列操作：$O(1)$
- 总体时间复杂度：$O(n^2)$，其中 n 为总事件数

4.3.2 空间复杂度

- 客户对象：$O(n)$，其中 n 为客户数
- 事件列表：$O(n)$，其中 n 为事件数
- 队列存储：$O(n)$，其中 n 为队列中客户数
- 总体空间复杂度：$O(n)$

5 总结与体会

5.1 系统特点

5.1.1 优点

1. **事件驱动**：采用事件驱动模拟，时间效率高
2. **动态内存管理**：使用动态内存分配，内存使用灵活
3. **模块化设计**：功能模块划分清晰，便于维护和扩展
4. **完整统计**：提供详细的统计信息，便于分析
5. **错误处理**：完善的错误处理机制，提高系统健壮性
6. **日志系统**：分级日志系统，便于调试和监控
7. **配置管理**：支持配置文件，提高系统灵活性

5.1.2 可能的改进

1. **图形界面**：可以添加图形界面以更直观地显示模拟过程
2. **参数配置文件**：可以通过配置文件设置模拟参数
3. **日志记录**：添加详细的日志记录功能
4. **多窗口支持**：实现真正的多窗口服务
5. **性能优化**：使用优先队列优化事件处理时间复杂度

5.2 收获与体会

通过本次课程设计，我深入理解了事件驱动模拟的思想，掌握了队列、链表等数据结构的实际应用。在实现过程中，我遇到了许多挑战，如事件顺序处理、等待队列管理等，通过不断调试和优化，最终完成了系统的设计与实现。

这次课程设计不仅提高了我的编程能力，更重要的是让我学会了如何将理论知识应用到实际问题中，对数据结构的理解也更加深刻。特别是在优化代码结构、完善错误处理机制和实现日志系统方面，我学到了很多大厂开发的实践经验。

6 参考文献

1. 吴伟民等. 《数据结构》. 广东工业大学计算机学院，2015.1

2. 严蔚敏, 吴伟民. 《数据结构（C 语言版）》. 清华大学出版社

3. 数据结构实验指导书（2015 春）

7 附录

7.1 源代码

7.1.1 bank_simulation.h

```c
/**
 * @file bank_simulation.h
 * @brief 银行模拟系统头文件
 * @author Yang Hengyi
 * @version 1.0
 * @date 2025-11-27
 */

#ifndef BANK_SIMULATION_H
#define BANK_SIMULATION_H

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <stdarg.h>

#ifdef __cplusplus
extern "C" {
#endif

// 常量定义
#define MAX_LINE_LENGTH 1024
#define DEFAULT_INITIAL_MONEY 10000
#define DEFAULT_CLOSE_TIME 600
#define MIN_SERVICE_TIME 1
#define MAX_SERVICE_TIME 20
#define MIN_ARRIVAL_INTERVAL 1
#define MAX_ARRIVAL_INTERVAL 10
#define MIN_TRANSACTION_AMOUNT -5000
```

```
#define MAX_TRANSACTION_AMOUNT 5000

// 错误码定义
#define BANK_SUCCESS 0
#define BANK_ERROR_INVALID_PARAM -1
#define BANK_ERROR_MEMORY_ALLOC -2
#define BANK_ERROR_FILE_OP -3
#define BANK_ERROR_QUEUE_OP -4
#define BANK_ERROR_EVENT_OP -5

// 事件类型枚举
typedef enum {
    EVENT_ARRIVE = 0,  // 客户到达事件
    EVENT_LEAVE = 1    // 客户离开事件
} EventType;

// 客户业务类型枚举
typedef enum {
    BUSINESS_WITHDRAW = 0,  // 取款/借款
    BUSINESS_DEPOSIT = 1    // 存款/还款
} BusinessType;

// 日志级别枚举
typedef enum {
    LOG_LEVEL_DEBUG = 0,
    LOG_LEVEL_INFO = 1,
    LOG_LEVEL_WARN = 2,
    LOG_LEVEL_ERROR = 3
} LogLevel;

// 客户结构体
typedef struct Customer {
    int id;                 // 客户ID
    int arrive_time;        // 到达时间
    int duration;           // 服务时间
    int amount;             // 交易金额（正数表示存款，负数表示取款）
    struct Customer* next;  // 链表指针
} Customer;

// 事件结构体
typedef struct Event {
    int occur_time;             // 事件发生时间
    EventType event_type;       // 事件类型
    struct Customer* customer;  // 相关客户
    struct Event* next;         // 链表指针
```

```
} Event;

// 队列结构体
typedef struct Queue {
    Customer* front;  // 队首指针
    Customer* rear;   // 队尾指针
    int size;         // 队列大小
} Queue;

// 银行系统配置结构体
typedef struct BankConfig {
    int initial_money;      // 初始资金
    int close_time;         // 关门时间
    int enable_log;         // 是否启用日志
    LogLevel log_level;     // 日志级别
    char log_file[256];     // 日志文件路径
} BankConfig;

// 银行系统结构体
typedef struct BankSystem {
    int total_money;        // 银行总资金
    int current_time;       // 当前时间
    int close_time;         // 关门时间
    int customer_count;     // 客户计数器
    Queue queue1;           // 第一个队列（正在等待服务的客户）
    Queue queue2;           // 第二个队列（因资金不足而等待的客户）
    Event* event_list;      // 事件列表
    int total_customers;    // 总客户数
    long long total_wait_time;  // 总等待时间
    FILE* log_fp;           // 日志文件指针
    LogLevel log_level;     // 日志级别
} BankSystem;

// 函数声明

/**
 * @brief 初始化银行系统
 * @param bank 银行系统指针
 * @param config 配置信息
 * @return 成功返回BANK_SUCCESS，失败返回错误码
 */
int init_bank_system(BankSystem* bank, const BankConfig* config);

/**
 * @brief 销毁银行系统
```

```
121  * @param bank 银行系统指针
122  * @return 成功返回BANK_SUCCESS, 失败返回错误码
123  */
124 int destroy_bank_system(BankSystem* bank);
125 
126 /**
127  * @brief 初始化队列
128  * @param q 队列指针
129  * @return 成功返回BANK_SUCCESS, 失败返回错误码
130  */
131 int init_queue(Queue* q);
132 
133 /**
134  * @brief 检查队列是否为空
135  * @param q 队列指针
136  * @return 空返回1, 非空返回0
137  */
138 int is_queue_empty(const Queue* q);
139 
140 /**
141  * @brief 获取队列大小
142  * @param q 队列指针
143  * @return 队列大小
144  */
145 int get_queue_size(const Queue* q);
146 
147 /**
148  * @brief 入队
149  * @param q 队列指针
150  * @param customer 客户指针
151  * @return 成功返回BANK_SUCCESS, 失败返回错误码
152  */
153 int enqueue(Queue* q, Customer* customer);
154 
155 /**
156  * @brief 出队
157  * @param q 队列指针
158  * @return 客户指针, 失败返回NULL
159  */
160 Customer* dequeue(Queue* q);
161 
162 /**
163  * @brief 获取队首元素
164  * @param q 队列指针
165  * @return 客户指针, 失败返回NULL
```

```
 */
Customer* queue_front(const Queue* q);

/**
 * @brief 插入事件（按时间顺序）
 * @param event_list 事件列表指针的指针
 * @param event 事件指针
 * @return 成功返回BANK_SUCCESS，失败返回错误码
 */
int insert_event(Event** event_list, Event* event);

/**
 * @brief 获取下一个事件
 * @param event_list 事件列表指针的指针
 * @return 事件指针，失败返回NULL
 */
Event* get_next_event(Event** event_list);

/**
 * @brief 创建客户
 * @param id 客户ID
 * @param arrive_time 到达时间
 * @return 客户指针，失败返回NULL
 */
Customer* create_customer(int id, int arrive_time);

/**
 * @brief 销毁客户
 * @param customer 客户指针
 * @return 成功返回BANK_SUCCESS，失败返回错误码
 */
int destroy_customer(Customer* customer);

/**
 * @brief 处理客户到达事件
 * @param bank 银行系统指针
 * @param event 事件指针
 * @return 成功返回BANK_SUCCESS，失败返回错误码
 */
int handle_arrive_event(BankSystem* bank, Event* event);

/**
 * @brief 处理客户离开事件
 * @param bank 银行系统指针
 * @param event 事件指针
```

```
211  * @return 成功返回BANK_SUCCESS, 失败返回错误码
212  */
213 int handle_leave_event(BankSystem* bank, Event* event);
214 
215 /**
216  * @brief 处理等待队列中的客户
217  * @param bank 银行系统指针
218  * @return 成功返回BANK_SUCCESS, 失败返回错误码
219  */
220 int process_queue2(BankSystem* bank);
221 
222 /**
223  * @brief 运行模拟
224  * @param bank 银行系统指针
225  * @return 成功返回BANK_SUCCESS, 失败返回错误码
226  */
227 int run_simulation(BankSystem* bank);
228 
229 /**
230  * @brief 打印统计信息
231  * @param bank 银行系统指针
232  * @return 成功返回BANK_SUCCESS, 失败返回错误码
233  */
234 int print_statistics(const BankSystem* bank);
235 
236 /**
237  * @brief 写日志
238  * @param bank 银行系统指针
239  * @param level 日志级别
240  * @param format 格式化字符串
241  * @param ... 可变参数
242  * @return 成功返回BANK_SUCCESS, 失败返回错误码
243  */
244 int bank_log(const BankSystem* bank, LogLevel level, const char* format,
     ...);
245 
246 /**
247  * @brief 加载配置文件
248  * @param config 配置结构体指针
249  * @param filename 配置文件名
250  * @return 成功返回BANK_SUCCESS, 失败返回错误码
251  */
252 int load_config(BankConfig* config, const char* filename);
253 
254 #ifdef __cplusplus
```

```
}
#endif

#endif // BANK_SIMULATION_H
```

Listing 5: bank_simulation.h

7.1.2 bank_simulation.c

```c
/**
 * @file bank_simulation.c
 * @brief 银行模拟系统实现文件
 * @author Yang Hengyi
 * @version 1.0
 * @date 2025-11-27
 */

#include "../include/bank_simulation.h"
#include <stdarg.h>
#include <math.h>

// 日志级别字符串映射
static const char* log_level_strings[] = {
    "DEBUG",
    "INFO",
    "WARN",
    "ERROR"
};

/**
 * @brief 初始化银行系统
 */
int init_bank_system(BankSystem* bank, const BankConfig* config) {
    if (!bank || !config) {
        return BANK_ERROR_INVALID_PARAM;
    }

    // 初始化银行系统参数
    bank->total_money = config->initial_money;
    bank->current_time = 0;
    bank->close_time = config->close_time;
    bank->total_customers = 0;
    bank->total_wait_time = 0;
    bank->customer_count = 0;
    bank->event_list = NULL;
```

```
37
38      // 初始化队列
39      if (init_queue(&bank->queue1) != BANK_SUCCESS) {
40          return BANK_ERROR_QUEUE_OP;
41      }
42
43      if (init_queue(&bank->queue2) != BANK_SUCCESS) {
44          return BANK_ERROR_QUEUE_OP;
45      }
46
47      // 初始化日志
48      bank->log_level = config->log_level;
49      if (config->enable_log && strlen(config->log_file) > 0) {
50          bank->log_fp = fopen(config->log_file, "w");
51          if (!bank->log_fp) {
52              fprintf(stderr, "Warning: Failed to open log file %s\n", config
    ->log_file);
53              bank->log_fp = stdout;
54          }
55      } else {
56          bank->log_fp = NULL;
57      }
58
59      // 创建第一个客户到达事件
60      Customer* first_customer = create_customer(bank->customer_count++, 0);
61      if (!first_customer) {
62          return BANK_ERROR_MEMORY_ALLOC;
63      }
64
65      Event* first_event = (Event*)malloc(sizeof(Event));
66      if (!first_event) {
67          destroy_customer(first_customer);
68          return BANK_ERROR_MEMORY_ALLOC;
69      }
70
71      first_event->occur_time = 0;
72      first_event->event_type = EVENT_ARRIVE;
73      first_event->customer = first_customer;
74      first_event->next = NULL;
75
76      if (insert_event(&bank->event_list, first_event) != BANK_SUCCESS) {
77          free(first_event);
78          destroy_customer(first_customer);
79          return BANK_ERROR_EVENT_OP;
80      }
```

```
81
82      bank_log(bank, LOG_LEVEL_INFO, "Bank system initialized with initial
        money: %d, close time: %d",
83               config->initial_money, config->close_time);
84
85      return BANK_SUCCESS;
86  }
87
88  /**
89   * @brief 销毁银行系统
90   */
91  int destroy_bank_system(BankSystem* bank) {
92      if (!bank) {
93          return BANK_ERROR_INVALID_PARAM;
94      }
95
96      // 释放所有事件
97      while (bank->event_list) {
98          Event* event = get_next_event(&bank->event_list);
99          if (event) {
100             if (event->customer) {
101                 destroy_customer(event->customer);
102             }
103             free(event);
104         }
105     }
106
107     // 释放队列中的所有客户
108     Customer* customer;
109     while ((customer = dequeue(&bank->queue1)) != NULL) {
110         destroy_customer(customer);
111     }
112
113     while ((customer = dequeue(&bank->queue2)) != NULL) {
114         destroy_customer(customer);
115     }
116
117     // 关闭日志文件
118     if (bank->log_fp && bank->log_fp != stdout) {
119         fclose(bank->log_fp);
120     }
121
122     bank_log(bank, LOG_LEVEL_INFO, "Bank system destroyed");
123     return BANK_SUCCESS;
124 }
```

```
125
126 /**
127  * @brief 初始化队列
128  */
129 int init_queue(Queue* q) {
130     if (!q) {
131         return BANK_ERROR_INVALID_PARAM;
132     }
133     q->front = NULL;
134     q->rear = NULL;
135     q->size = 0;
136     return BANK_SUCCESS;
137 }
138
139 /**
140  * @brief 检查队列是否为空
141  */
142 int is_queue_empty(const Queue* q) {
143     if (!q) {
144         return 1;
145     }
146     return q->front == NULL;
147 }
148
149 /**
150  * @brief 获取队列大小
151  */
152 int get_queue_size(const Queue* q) {
153     if (!q) {
154         return 0;
155     }
156     return q->size;
157 }
158
159 /**
160  * @brief 入队
161  */
162 int enqueue(Queue* q, Customer* customer) {
163     if (!q || !customer) {
164         return BANK_ERROR_INVALID_PARAM;
165     }
166
167     customer->next = NULL;
168
169     if (is_queue_empty(q)) {
```

```
170         q->front = customer;
171         q->rear = customer;
172     } else {
173         q->rear->next = customer;
174         q->rear = customer;
175     }
176 
177     q->size++;
178     return BANK_SUCCESS;
179 }
180 
181 /**
182  * @brief 出队
183  */
184 Customer* dequeue(Queue* q) {
185     if (!q || is_queue_empty(q)) {
186         return NULL;
187     }
188 
189     Customer* customer = q->front;
190     q->front = q->front->next;
191 
192     if (q->front == NULL) {
193         q->rear = NULL;
194     }
195 
196     customer->next = NULL;
197     q->size--;
198 
199     return customer;
200 }
201 
202 /**
203  * @brief 获取队首元素
204  */
205 Customer* queue_front(const Queue* q) {
206     if (!q || is_queue_empty(q)) {
207         return NULL;
208     }
209     return q->front;
210 }
211 
212 /**
213  * @brief 插入事件（按时间顺序）
214  */
```

```
215 int insert_event(Event** event_list, Event* event) {
216     if (!event_list || !event) {
217         return BANK_ERROR_INVALID_PARAM;
218     }
219 
220     // 如果事件列表为空或新事件时间早于第一个事件
221     if (*event_list == NULL || (*event_list)->occur_time > event->
        occur_time) {
222         event->next = *event_list;
223         *event_list = event;
224         return BANK_SUCCESS;
225     }
226 
227     // 查找插入位置
228     Event* current = *event_list;
229     while (current->next != NULL && current->next->occur_time <= event->
        occur_time) {
230         current = current->next;
231     }
232 
233     event->next = current->next;
234     current->next = event;
235     return BANK_SUCCESS;
236 }
237 
238 /**
239  * @brief 获取下一个事件
240  */
241 Event* get_next_event(Event** event_list) {
242     if (!event_list || *event_list == NULL) {
243         return NULL;
244     }
245 
246     Event* event = *event_list;
247     *event_list = (*event_list)->next;
248     event->next = NULL;
249     return event;
250 }
251 
252 /**
253  * @brief 创建客户
254  */
255 Customer* create_customer(int id, int arrive_time) {
256     Customer* customer = (Customer*)malloc(sizeof(Customer));
257     if (!customer) {
```

```
258         return NULL;
259     }
260 
261     customer->id = id;
262     customer->arrive_time = arrive_time;
263 
264     // 随机生成服务时间 (1-20分钟)
265     customer->duration = rand() % (MAX_SERVICE_TIME - MIN_SERVICE_TIME + 1)
        + MIN_SERVICE_TIME;
266 
267     // 随机生成交易金额 (-5000到5000元)
268     customer->amount = rand() % (MAX_TRANSACTION_AMOUNT -
    MIN_TRANSACTION_AMOUNT + 1) + MIN_TRANSACTION_AMOUNT;
269 
270     customer->next = NULL;
271 
272     return customer;
273 }
274 
275 /**
276  * @brief 销毁客户
277  */
278 int destroy_customer(Customer* customer) {
279     if (!customer) {
280         return BANK_ERROR_INVALID_PARAM;
281     }
282     free(customer);
283     return BANK_SUCCESS;
284 }
285 
286 /**
287  * @brief 处理客户到达事件
288  */
289 int handle_arrive_event(BankSystem* bank, Event* event) {
290     if (!bank || !event || !event->customer) {
291         return BANK_ERROR_INVALID_PARAM;
292     }
293 
294     Customer* customer = event->customer;
295 
296     // 将客户加入第一个队列
297     if (enqueue(&bank->queue1, customer) != BANK_SUCCESS) {
298         bank_log(bank, LOG_LEVEL_ERROR, "Failed to enqueue customer %d",
    customer->id);
299         destroy_customer(customer);
```

```
300         free(event);
301         return BANK_ERROR_QUEUE_OP;
302     }
303 
304     bank->total_customers++;
305 
306     bank_log(bank, LOG_LEVEL_INFO, "Time %d: Customer %d arrived,
    transaction amount %d, service time %d",
307              bank->current_time, customer->id, customer->amount, customer->
    duration);
308 
309     // 生成下一个客户到达事件
310     if (bank->current_time < bank->close_time) {
311         // 下一个客户到达的时间间隔 (1-10分钟)
312         int interval = rand() % (MAX_ARRIVAL_INTERVAL -
    MIN_ARRIVAL_INTERVAL + 1) + MIN_ARRIVAL_INTERVAL;
313         int next_arrive_time = bank->current_time + interval;
314 
315         if (next_arrive_time < bank->close_time) {
316             Customer* next_customer = create_customer(bank->customer_count
    ++, next_arrive_time);
317             if (next_customer) {
318                 Event* next_event = (Event*)malloc(sizeof(Event));
319                 if (next_event) {
320                     next_event->occur_time = next_arrive_time;
321                     next_event->event_type = EVENT_ARRIVE;
322                     next_event->customer = next_customer;
323                     next_event->next = NULL;
324 
325                     if (insert_event(&bank->event_list, next_event) !=
    BANK_SUCCESS) {
326                         bank_log(bank, LOG_LEVEL_WARN, "Failed to insert
    next arrival event");
327                         destroy_customer(next_customer);
328                         free(next_event);
329                         // 不返回错误，因为主流程仍然可以继续
330                     }
331                 } else {
332                     destroy_customer(next_customer);
333                 }
334             }
335         }
336     }
337 
338     free(event);
```

```
339     return BANK_SUCCESS;
340 }
341 
342 /**
343  * @brief 处理客户离开事件
344  */
345 int handle_leave_event(BankSystem* bank, Event* event) {
346     if (!bank || !event || !event->customer) {
347         return BANK_ERROR_INVALID_PARAM;
348     }
349 
350     Customer* customer = event->customer;
351 
352     // 计算客户在银行的总逗留时间
353     int stay_time = bank->current_time - customer->arrive_time;
354     bank->total_wait_time += stay_time;
355 
356     bank_log(bank, LOG_LEVEL_INFO, "Time %d: Customer %d leaving,
    transaction amount %d, total stay time %d",
357              bank->current_time, customer->id, customer->amount, stay_time)
    ;
358 
359     // 如果是存款或还款业务（金额为正）
360     if (customer->amount > 0) {
361         bank->total_money += customer->amount;
362         bank_log(bank, LOG_LEVEL_INFO, "  Bank money increased by %d,
    current total %d",
363                  customer->amount, bank->total_money);
364 
365         // 处理第二个队列中等待的客户
366         process_queue2(bank);
367     }
368     // 如果是取款或借款业务（金额为负）
369     else {
370         // 检查银行是否有足够资金
371         if (bank->total_money >= abs(customer->amount)) {
372             bank->total_money += customer->amount; // amount为负数
373             bank_log(bank, LOG_LEVEL_INFO, "  Bank money decreased by %d,
    current total %d",
374                      abs(customer->amount), bank->total_money);
375         } else {
376             // 资金不足，将客户加入第二个队列
377             bank_log(bank, LOG_LEVEL_INFO, "  Insufficient funds, customer
    %d added to waiting queue", customer->id);
378             if (enqueue(&bank->queue2, customer) != BANK_SUCCESS) {
```

```
                bank_log(bank, LOG_LEVEL_ERROR, "Failed to enqueue customer %d to waiting queue", customer->id);
                destroy_customer(customer);
                free(event);
                return BANK_ERROR_QUEUE_OP;
            }
            free(event);
            return BANK_SUCCESS;
        }
    }

    destroy_customer(customer);
    free(event);
    return BANK_SUCCESS;
}

/**
 * @brief 处理第二个队列中的客户
 */
int process_queue2(BankSystem* bank) {
    if (!bank) {
        return BANK_ERROR_INVALID_PARAM;
    }

    if (is_queue_empty(&bank->queue2)) {
        return BANK_SUCCESS;
    }

    bank_log(bank, LOG_LEVEL_DEBUG, "Processing waiting queue customers");

    // 记录处理前的银行资金
    int money_before = bank->total_money;
    int processed_count = 0;

    // 创建临时队列存储无法处理的客户
    Queue temp_queue;
    if (init_queue(&temp_queue) != BANK_SUCCESS) {
        bank_log(bank, LOG_LEVEL_ERROR, "Failed to initialize temp queue");
        return BANK_ERROR_QUEUE_OP;
    }

    Customer* customer = dequeue(&bank->queue2);
    while (customer != NULL) {
        // 检查银行是否有足够资金处理该客户
        if (bank->total_money >= abs(customer->amount)) {
```

```
423             // 可以处理
424             bank->total_money += customer->amount;
425             int stay_time = bank->current_time - customer->arrive_time;
426             bank->total_wait_time += stay_time;
427             processed_count++;
428 
429             bank_log(bank, LOG_LEVEL_INFO, "  Time %d: Waiting customer %d
    completed transaction, amount %d, total stay time %d",
430                     bank->current_time, customer->id, customer->amount,
    stay_time);
431             bank_log(bank, LOG_LEVEL_INFO, "  Bank money changed by %d,
    current total %d",
432                     customer->amount, bank->total_money);
433 
434             destroy_customer(customer);
435         } else {
436             // 仍然无法处理，放回临时队列
437             if (enqueue(&temp_queue, customer) != BANK_SUCCESS) {
438                 bank_log(bank, LOG_LEVEL_ERROR, "Failed to enqueue customer
     %d to temp queue", customer->id);
439                 destroy_customer(customer);
440             }
441         }
442 
443         // 检查是否应该停止处理
444         if (bank->total_money <= money_before || processed_count > 100) {
445             // 将当前客户重新放回队列
446             if (customer != NULL && enqueue(&temp_queue, customer) !=
    BANK_SUCCESS) {
447                 bank_log(bank, LOG_LEVEL_ERROR, "Failed to enqueue customer
     %d to temp queue", customer->id);
448                 destroy_customer(customer);
449             }
450             break;
451         }
452 
453         customer = dequeue(&bank->queue2);
454     }
455 
456     // 将未处理的客户移回等待队列
457     while (!is_queue_empty(&temp_queue)) {
458         Customer* temp_customer = dequeue(&temp_queue);
459         if (temp_customer && enqueue(&bank->queue2, temp_customer) !=
    BANK_SUCCESS) {
```

```
460             bank_log(bank, LOG_LEVEL_ERROR, "Failed to enqueue customer %d
    back to waiting queue", temp_customer->id);
461             destroy_customer(temp_customer);
462         }
463     }
464 
465     bank_log(bank, LOG_LEVEL_DEBUG, "Finished processing waiting queue");
466     return BANK_SUCCESS;
467 }
468 
469 /**
470  * @brief 运行模拟
471  */
472 int run_simulation(BankSystem* bank) {
473     if (!bank) {
474         return BANK_ERROR_INVALID_PARAM;
475     }
476 
477     bank_log(bank, LOG_LEVEL_INFO, "Bank simulation started");
478     bank_log(bank, LOG_LEVEL_INFO, "Initial money: %d", bank->total_money);
479     bank_log(bank, LOG_LEVEL_INFO, "Close time: %d", bank->close_time);
480     bank_log(bank, LOG_LEVEL_INFO, "========================");
481 
482     while (bank->event_list != NULL) {
483         Event* event = get_next_event(&bank->event_list);
484         if (event == NULL) {
485             break;
486         }
487 
488         bank->current_time = event->occur_time;
489 
490         // 如果已经超过关门时间，结束营业
491         if (bank->current_time >= bank->close_time) {
492             bank_log(bank, LOG_LEVEL_INFO, "Time %d: Bank closed, business
    ended", bank->current_time);
493             if (event->customer) {
494                 destroy_customer(event->customer);
495             }
496             free(event);
497             break;
498         }
499 
500         int result;
501         if (event->event_type == EVENT_ARRIVE) {
502             // 到达事件
```

```
503             result = handle_arrive_event(bank, event);
504         } else {
505             // 离开事件
506             result = handle_leave_event(bank, event);
507         }
508 
509         // 如果处理事件时出现严重错误，停止模拟
510         if (result != BANK_SUCCESS && result != BANK_ERROR_EVENT_OP) {
511             bank_log(bank, LOG_LEVEL_ERROR, "Error occurred during event
    processing, stopping simulation");
512             return result;
513         }
514     }
515 
516     // 处理完所有事件后，处理队列1中剩余的客户
517     bank_log(bank, LOG_LEVEL_INFO, "========================");
518     bank_log(bank, LOG_LEVEL_INFO, "Processing remaining customers");
519 
520     while (!is_queue_empty(&bank->queue1)) {
521         Customer* customer = dequeue(&bank->queue1);
522         if (customer) {
523             int stay_time = bank->current_time - customer->arrive_time;
524             bank->total_wait_time += stay_time;
525             bank_log(bank, LOG_LEVEL_INFO, "Time %d: Customer %d left (
    business ended), total stay time %d",
526                      bank->current_time, customer->id, stay_time);
527             destroy_customer(customer);
528         }
529     }
530 
531     return BANK_SUCCESS;
532 }
533 
534 /**
535  * @brief 打印统计信息
536  */
537 int print_statistics(const BankSystem* bank) {
538     if (!bank) {
539         return BANK_ERROR_INVALID_PARAM;
540     }
541 
542     printf("========================\n");
543     printf("Simulation completed, statistics:\n");
544     printf("Total customers: %d\n", bank->total_customers);
545     printf("Total wait time: %lld minutes\n", bank->total_wait_time);
```

```
546
547     if (bank->total_customers > 0) {
548         double avg_stay_time = (double)bank->total_wait_time / bank->
    total_customers;
549         printf("Average stay time: %.2f minutes\n", avg_stay_time);
550     }
551
552     printf("Final bank money: %d yuan\n", bank->total_money);
553     printf("Remaining customers in waiting queue: %d\n", get_queue_size(&
    bank->queue2));
554     return BANK_SUCCESS;
555 }
556
557 /**
558  * @brief 写日志
559  */
560 int bank_log(const BankSystem* bank, LogLevel level, const char* format,
    ...) {
561     if (!bank || level < bank->log_level) {
562         return BANK_SUCCESS;
563     }
564
565     // 如果没有启用日志，直接返回
566     if (!bank->log_fp) {
567         return BANK_SUCCESS;
568     }
569
570     // 获取当前时间
571     time_t now;
572     time(&now);
573     struct tm* local_time = localtime(&now);
574
575     // 写入日志级别和时间
576     fprintf(bank->log_fp, "[%04d-%02d-%02d %02d:%02d:%02d][%s] ",
577             local_time->tm_year + 1900,
578             local_time->tm_mon + 1,
579             local_time->tm_mday,
580             local_time->tm_hour,
581             local_time->tm_min,
582             local_time->tm_sec,
583             log_level_strings[level]);
584
585     // 写入日志内容
586     va_list args;
587     va_start(args, format);
```

```
588     vfprintf(bank->log_fp, format, args);
589     va_end(args);
590 
591     fprintf(bank->log_fp, "\n");
592     fflush(bank->log_fp);
593     return BANK_SUCCESS;
594 }
595 
596 /**
597  * @brief 加载配置文件
598  */
599 int load_config(BankConfig* config, const char* filename) {
600     if (!config || !filename) {
601         return BANK_ERROR_INVALID_PARAM;
602     }
603 
604     FILE* fp = fopen(filename, "r");
605     if (!fp) {
606         return BANK_ERROR_FILE_OP;
607     }
608 
609     char line[MAX_LINE_LENGTH];
610     while (fgets(line, sizeof(line), fp)) {
611         // 跳过注释和空行
612         if (line[0] == '#' || line[0] == '\n' || line[0] == '\r') {
613             continue;
614         }
615 
616         char key[MAX_LINE_LENGTH];
617         char value[MAX_LINE_LENGTH];
618 
619         if (sscanf(line, "%s = %s", key, value) == 2) {
620             if (strcmp(key, "initial_money") == 0) {
621                 config->initial_money = atoi(value);
622             } else if (strcmp(key, "close_time") == 0) {
623                 config->close_time = atoi(value);
624             } else if (strcmp(key, "enable_log") == 0) {
625                 config->enable_log = atoi(value);
626             } else if (strcmp(key, "log_level") == 0) {
627                 config->log_level = (LogLevel)atoi(value);
628             } else if (strcmp(key, "log_file") == 0) {
629                 strncpy(config->log_file, value, sizeof(config->log_file) -
    1);
630                 config->log_file[sizeof(config->log_file) - 1] = '\0';
631             }
```

```
632         }
633     }
634 
635     fclose(fp);
636     return BANK_SUCCESS;
637 }
```

Listing 6: bank_simulation.c

7.1.3 main.c

```
1  /**
2   * @file main.c
3   * @brief 银行模拟系统主程序
4   * @author Yang Hengyi
5   * @version 1.0
6   * @date 2025-11-27
7   */
8  
9  #include "../include/bank_simulation.h"
10 #include <unistd.h>
11 
12 /**
13  * @brief 打印帮助信息
14  */
15 void print_help(const char* program_name) {
16     printf("Usage: %s [options]\n", program_name);
17     printf("Options:\n");
18     printf("  -h, --help              Show this help message\n");
19     printf("  -c, --config <file>     Specify configuration file\n");
20     printf("  -m, --money <amount>    Set initial money (default: 10000)\n"
    );
21     printf("  -t, --time <minutes>    Set close time in minutes (default:
    600)\n");
22     printf("  -l, --log               Enable logging\n");
23     printf("  -v, --verbose           Enable verbose logging (DEBUG level)\
    n");
24     printf("\n");
25     printf("Examples:\n");
26     printf("  %s                          # Run with default settings\n",
    program_name);
27     printf("  %s -m 5000 -t 300          # Run with 5000 initial money and
    300 minutes\n", program_name);
28     printf("  %s -c config.ini           # Run with configuration file\n",
    program_name);
```

```
29      printf("  %s -l -v                    # Run with verbose logging\n",
        program_name);
30  }
31
32  /**
33   * @brief 创建默认配置文件
34   */
35  int create_default_config(const char* filename) {
36      FILE* fp = fopen(filename, "w");
37      if (!fp) {
38          return BANK_ERROR_FILE_OP;
39      }
40
41      fprintf(fp, "# 银行模拟系统配置文件\n");
42      fprintf(fp, "\n");
43      fprintf(fp, "# 初始资金\n");
44      fprintf(fp, "initial_money = 10000\n");
45      fprintf(fp, "\n");
46      fprintf(fp, "# 营业时间（分钟）\n");
47      fprintf(fp, "close_time = 600\n");
48      fprintf(fp, "\n");
49      fprintf(fp, "# 是否启用日志\n");
50      fprintf(fp, "enable_log = 1\n");
51      fprintf(fp, "\n");
52      fprintf(fp, "# 日志级别 (0=DEBUG, 1=INFO, 2=WARN, 3=ERROR)\n");
53      fprintf(fp, "log_level = 1\n");
54      fprintf(fp, "\n");
55      fprintf(fp, "# 日志文件路径\n");
56      fprintf(fp, "log_file = bank_simulation.log\n");
57
58      fclose(fp);
59      return BANK_SUCCESS;
60  }
61
62  /**
63   * @brief 主函数
64   */
65  int main(int argc, char* argv[]) {
66      // 设置随机数种子
67      srand((unsigned int)time(NULL));
68
69      // 初始化默认配置
70      BankConfig config = {
71          .initial_money = DEFAULT_INITIAL_MONEY,
72          .close_time = DEFAULT_CLOSE_TIME,
```

```
        .enable_log = 0,
        .log_level = LOG_LEVEL_INFO,
        .log_file = ""
    };

    // 解析命令行参数
    int opt;
    while ((opt = getopt(argc, argv, "hc:m:t:lv")) != -1) {
        switch (opt) {
            case 'h':
                print_help(argv[0]);
                return 0;
            case 'c':
                if (load_config(&config, optarg) != BANK_SUCCESS) {
                    fprintf(stderr, "Error: Failed to load config file %s\n
", optarg);
                    return 1;
                }
                break;
            case 'm':
                config.initial_money = atoi(optarg);
                break;
            case 't':
                config.close_time = atoi(optarg);
                break;
            case 'l':
                config.enable_log = 1;
                if (strlen(config.log_file) == 0) {
                    strncpy(config.log_file, "bank_simulation.log", sizeof(
config.log_file) - 1);
                }
                break;
            case 'v':
                config.log_level = LOG_LEVEL_DEBUG;
                break;
            default:
                print_help(argv[0]);
                return 1;
        }
    }

    // 如果启用了日志但没有指定日志文件，使用默认文件名
    if (config.enable_log && strlen(config.log_file) == 0) {
        strncpy(config.log_file, "bank_simulation.log", sizeof(config.
log_file) - 1);
```

```
115     }
116 
117     // 创建默认配置文件（如果不存在）
118     if (access("config.ini", F_OK) == -1) {
119         if (create_default_config("config.ini") != BANK_SUCCESS) {
120             fprintf(stderr, "Warning: Failed to create default config file\
    n");
121         }
122     }
123 
124     printf("Bank Simulation System\n");
125     printf("======================\n");
126     printf("Initial money: %d\n", config.initial_money);
127     printf("Close time: %d minutes\n", config.close_time);
128     if (config.enable_log) {
129         printf("Logging enabled, level: %d, file: %s\n", config.log_level,
    config.log_file);
130     }
131     printf("\n");
132 
133     // 初始化银行系统
134     BankSystem bank;
135     if (init_bank_system(&bank, &config) != BANK_SUCCESS) {
136         fprintf(stderr, "Error: Failed to initialize bank system\n");
137         return 1;
138     }
139 
140     // 运行模拟
141     printf("Running simulation...\n");
142     if (run_simulation(&bank) != BANK_SUCCESS) {
143         fprintf(stderr, "Error: Failed to run simulation\n");
144         destroy_bank_system(&bank);
145         return 1;
146     }
147 
148     // 打印统计信息
149     if (print_statistics(&bank) != BANK_SUCCESS) {
150         fprintf(stderr, "Error: Failed to print statistics\n");
151         destroy_bank_system(&bank);
152         return 1;
153     }
154 
155     // 销毁银行系统
156     if (destroy_bank_system(&bank) != BANK_SUCCESS) {
157         fprintf(stderr, "Error: Failed to destroy bank system\n");
```

```
158         return 1;
159     }
160 
161     return 0;
162 }
```

Listing 7: main.c

7.2 Makefile

```
1  # 银行模拟系统 Makefile
2  
3  # 编译器
4  CC = gcc
5  
6  # 编译选项
7  CFLAGS = -Wall -Wextra -std=c99 -g -O2 -pedantic
8  CFLAGS += -D_DEFAULT_SOURCE  # 用于兼容较新的Linux系统
9  
10 # 包含目录
11 INCLUDES = -Iinclude
12 
13 # 目标文件
14 TARGET = bin/bank_simulation
15 
16 # 源文件目录
17 SRC_DIR = src
18 INC_DIR = include
19 BIN_DIR = bin
20 
21 # 源文件
22 SRCS = $(SRC_DIR)/bank_simulation.c $(SRC_DIR)/main.c
23 
24 # 对象文件
25 OBJS = $(SRCS:$(SRC_DIR)/%.c=$(BIN_DIR)/%.o)
26 
27 # 依赖文件
28 DEPS = $(OBJS:.o=.d)
29 
30 # 默认目标
31 all: directories $(TARGET)
32 
33 # 创建所需目录
34 directories:
```

```
35   @mkdir -p $(BIN_DIR)
36
37 # 链接目标文件
38 $(TARGET): $(OBJS)
39   $(CC) $(OBJS) -o $@
40   @echo "Build successful: $@"
41
42 # 编译源文件（包含依赖关系）
43 $(BIN_DIR)/%.o: $(SRC_DIR)/%.c
44   $(CC) $(CFLAGS) $(INCLUDES) -MMD -MP -c $< -o $@
45
46 # 包含自动生成的依赖文件
47 -include $(DEPS)
48
49 # 清理生成文件
50 clean:
51   rm -rf $(BIN_DIR) $(TARGET)
52
53 # 重新编译
54 rebuild: clean all
55
56 # 运行程序
57 run: $(TARGET)
58   ./$(TARGET)
59
60 # 安装（示例）
61 install: $(TARGET)
62   cp $(TARGET) /usr/local/bin/
63
64 # 显示变量（调试用）
65 print-vars:
66   @echo "SRCS: $(SRCS)"
67   @echo "OBJS: $(OBJS)"
68   @echo "DEPS: $(DEPS)"
69
70 .PHONY: all clean rebuild run install print-vars directories
```

Listing 8: Makefile