# Lab2 记录管理器函数实现说明

**作者**: 何思远  
**学号**: 2023212224  
**日期**: 2025-10-19

## 目录

### 一、RmFileHandle类方法实现

1. [get\_record()](#X435fdcd6c4b1359106b30531c2143d2cd5f955d)
2. [insert\_record()](#X2652a9fd397054669826bf9250742cde4094510)
3. [delete\_record()](#X804e6d787058e77da0877bb1dae441648ab578c)
4. [update\_record()](#X6bcc4c0fa6b6c6d758910db9f06ea51586d5126)
5. [fetch\_page\_handle()](#X30719ad6a6448e40e6333f101904764f88bcfe1)
6. [create\_new\_page\_handle()](#Xaf1c2e50c1df9a8dacad39912a03cdc9b117775)
7. [create\_page\_handle()](#X8689f2fc8c16c65bccea450c29647b8f31d5691)
8. [release\_page\_handle()](#X1ee8ac8cad3401317e2264031c842ffa22e272b)

### 二、RmScan类方法实现

1. [RmScan() 构造函数](#Xf151b04896095d2ddce1e3cc6331e55a862d398)
2. [next()](#X3cf64beb0c40d4d48ce717e8752f1fbde2b35e2)
3. [is\_end()](#X4afb6ae8d5bcef0b3ed3d1b2e5b304eee67f111)

## 一、RmFileHandle类方法实现

RmFileHandle类负责文件级别的记录操作，包括记录的增删改查以及页面管理。每个RmFileHandle对象对应一个表数据文件。

### 1.1 get\_record()

#### 1. 方法声明

std::unique\_ptr<RmRecord> get\_record(const Rid& rid, Context\* context) const;

**方法名**: get\_record

**返回类型**: std::unique\_ptr<RmRecord>

**功能**: 获取表中指定记录号(Rid)对应的记录

**参数列表**:

|  | **rid** | **context** |
| --- | --- | --- |
| **类型** | const Rid& | Context\* |
| **含义** | 记录号，包含页面号(page\_no)和槽位号(slot\_no) | 上下文信息（暂未使用） |

**返回值**: 返回指向RmRecord的智能指针，若指定位置无记录则返回nullptr

#### 2. 方法实现思路

1. 调用fetch\_page\_handle(rid.page\_no)获取记录所在页面的页面句柄
2. 使用Bitmap::is\_set(page\_handle.bitmap, rid.slot\_no)检查该槽位的位图标记
3. 如果位图标记为0（无记录），直接返回nullptr
4. 如果位图标记为1（有记录），通过page\_handle.get\_slot(rid.slot\_no)获取槽位数据的内存地址
5. 创建RmRecord对象，将槽位数据和记录大小作为参数
6. 使用std::make\_unique返回智能指针

**代码位置**: src/record/rm\_file\_handle.cpp:19-36

#### 3. 方法实现难点

**难点**: 理解RmPageHandle的内存布局和槽位地址计算

**解决方法**: RmPageHandle结构体已经封装了get\_slot()方法，该方法通过slots + slot\_no \* file\_hdr->record\_size计算出指定槽位的起始地址。只需直接调用即可，无需手动计算偏移量。

### 1.2 insert\_record()

#### 1. 方法声明

Rid insert\_record(char\* buf, Context\* context);

**方法名**: insert\_record

**返回类型**: Rid

**功能**: 在表中插入一条新记录，系统自动选择插入位置

**参数列表**:

|  | **buf** | **context** |
| --- | --- | --- |
| **类型** | char\* | Context\* |
| **含义** | 指向要插入的记录数据的指针 | 上下文信息（暂未使用） |

**返回值**: 返回插入记录的Rid（记录号）

#### 2. 方法实现思路

1. 调用create\_page\_handle()获取一个有空闲空间的页面（该方法会自动判断是使用现有空闲页还是创建新页）
2. 使用Bitmap::first\_bit(false, page\_handle.bitmap, file\_hdr\_.num\_records\_per\_page)找到第一个位图值为0的槽位（即空闲槽位）
3. 通过page\_handle.get\_slot(slot\_no)获取槽位地址，使用memcpy()将buf的数据复制到该槽位
4. 调用Bitmap::set(page\_handle.bitmap, slot\_no)将位图对应位置置1，标记槽位已使用
5. 更新页面头：page\_handle.page\_hdr->num\_records++
6. 检查页面是否已满：如果page\_hdr->num\_records == file\_hdr\_.num\_records\_per\_page，说明页面已满，需要更新文件头的first\_free\_page\_no为当前页的next\_free\_page\_no，将当前页从空闲链表中移除
7. 调用buffer\_pool\_manager\_->unpin\_page()并传入true标记页面为脏页
8. 返回Rid{page\_no, slot\_no}

**代码位置**: src/record/rm\_file\_handle.cpp:44-80

#### 3. 方法实现难点

**难点1**: 正确维护空闲页面链表

当插入记录后页面变满时，需要将该页面从空闲链表中移除。具体操作是将文件头的first\_free\_page\_no更新为当前页面的next\_free\_page\_no。

**解决方法**: 在更新页面记录数后，增加判断条件：

if (page\_handle.page\_hdr->num\_records == file\_hdr\_.num\_records\_per\_page) {  
 file\_hdr\_.first\_free\_page\_no = page\_handle.page\_hdr->next\_free\_page\_no;  
}

**难点2**: 理解堆文件组织方式

堆文件不关心记录插入的具体位置，只需找到任意有空闲空间的页面即可。因此使用create\_page\_handle()统一获取可用页面，简化了实现。

### 1.3 delete\_record()

#### 1. 方法声明

void delete\_record(const Rid& rid, Context\* context);

**方法名**: delete\_record

**返回类型**: void

**功能**: 删除表中指定记录号对应的记录

**参数列表**:

|  | **rid** | **context** |
| --- | --- | --- |
| **类型** | const Rid& | Context\* |
| **含义** | 要删除的记录号 | 上下文信息（暂未使用） |

#### 2. 方法实现思路

1. 调用fetch\_page\_handle(rid.page\_no)获取记录所在页面
2. 使用Bitmap::is\_set()检查该位置是否有记录，若无则直接返回
3. 在删除前，记录页面当前是否已满：bool was\_full = (page\_hdr->num\_records == file\_hdr\_.num\_records\_per\_page)
4. 调用Bitmap::reset(page\_handle.bitmap, rid.slot\_no)将位图对应位置置0，标记槽位为空闲
5. 更新页面头：page\_handle.page\_hdr->num\_records--
6. 判断页面是否从"已满"变为"未满"：如果was\_full为true，说明删除前页面已满，删除后变为未满，需要调用release\_page\_handle(page\_handle)将页面重新加入空闲链表
7. 调用buffer\_pool\_manager\_->unpin\_page()并传入true标记页面为脏页

**代码位置**: src/record/rm\_file\_handle.cpp:96-127

#### 3. 方法实现难点

**难点**: 判断何时需要将页面加入空闲链表

只有当页面从"已满"状态变为"未满"状态时，才需要调用release\_page\_handle()。如果页面本来就未满，删除记录后仍然未满，不需要额外操作（因为它已经在空闲链表中）。

**解决方法**: 在删除操作前先记录页面是否已满的状态，删除后根据这个状态判断：

bool was\_full = (page\_handle.page\_hdr->num\_records == file\_hdr\_.num\_records\_per\_page);  
// ... 执行删除操作 ...  
if (was\_full) {  
 release\_page\_handle(page\_handle);  
}

### 1.4 update\_record()

#### 1. 方法声明

void update\_record(const Rid& rid, char\* buf, Context\* context);

**方法名**: update\_record

**返回类型**: void

**功能**: 更新表中指定记录号对应的记录数据

**参数列表**:

|  | **rid** | **buf** | **context** |
| --- | --- | --- | --- |
| **类型** | const Rid& | char\* | Context\* |
| **含义** | 要更新的记录号 | 新记录的数据指针 | 上下文信息（暂未使用） |

#### 2. 方法实现思路

1. 调用fetch\_page\_handle(rid.page\_no)获取记录所在页面
2. 使用Bitmap::is\_set()检查该位置是否有记录，若无则直接返回
3. 通过page\_handle.get\_slot(rid.slot\_no)获取槽位地址
4. 使用memcpy(slot\_data, buf, file\_hdr\_.record\_size)将新数据覆盖旧数据
5. 调用buffer\_pool\_manager\_->unpin\_page()并传入true标记页面为脏页

**代码位置**: src/record/rm\_file\_handle.cpp:136-156

#### 3. 方法实现难点

本方法实现较为简单，无特殊难点。只需注意：

* 更新操作不改变记录数量，因此不需要更新num\_records
* 不需要修改位图
* 不会影响页面的满/未满状态，不需要维护空闲链表

### 1.5 fetch\_page\_handle()

#### 1. 方法声明

RmPageHandle fetch\_page\_handle(int page\_no) const;

**方法名**: fetch\_page\_handle

**返回类型**: RmPageHandle

**功能**: 获取指定页面号对应的页面句柄

**参数列表**:

|  | **page\_no** |
| --- | --- |
| **类型** | int |
| **含义** | 页面号 |

**返回值**: 返回对应的RmPageHandle对象

#### 2. 方法实现思路

1. 检查页面号的有效性：如果page\_no >= file\_hdr\_.num\_pages，说明页面号超出范围，抛出PageNotExistError异常
2. 构造PageId对象：PageId{fd\_, page\_no}，其中fd\_是文件描述符
3. 调用buffer\_pool\_manager\_->fetch\_page(PageId{fd\_, page\_no})从缓冲池获取页面
4. 创建RmPageHandle对象：RmPageHandle(&file\_hdr\_, page)
5. 返回RmPageHandle对象（RmPageHandle的构造函数会自动解析页面内部结构，设置page\_hdr、bitmap、slots等指针）

**代码位置**: src/record/rm\_file\_handle.cpp:166-182

#### 3. 方法实现难点

**难点**: 理解RmPageHandle的构造过程

RmPageHandle构造函数会根据文件头信息，自动计算页面内部各部分的地址：

* page\_hdr指向页面头
* bitmap指向位图区域
* slots指向记录槽位区域

这些计算已经封装在RmPageHandle构造函数中，只需传入正确的参数即可。

### 1.6 create\_new\_page\_handle()

#### 1. 方法声明

RmPageHandle create\_new\_page\_handle();

**方法名**: create\_new\_page\_handle

**返回类型**: RmPageHandle

**功能**: 创建一个全新的页面句柄，会在磁盘上分配新页面

**返回值**: 返回新创建的RmPageHandle对象

#### 2. 方法实现思路

1. 构造新页面的PageId：PageId new\_page\_id = {fd\_, file\_hdr\_.num\_pages}，页面号为当前页面总数（即下一个可用页面号）
2. 调用buffer\_pool\_manager\_->new\_page(&new\_page\_id)在缓冲池中创建新页面
3. 创建RmPageHandle对象：RmPageHandle page\_handle(&file\_hdr\_, page)
4. 初始化页面头：
   * page\_handle.page\_hdr->next\_free\_page\_no = RM\_NO\_PAGE（-1，表示没有下一个空闲页）
   * page\_handle.page\_hdr->num\_records = 0（记录数初始化为0）
5. 初始化位图：调用Bitmap::init(page\_handle.bitmap, file\_hdr\_.bitmap\_size)将位图全部置0
6. 更新文件头：
   * file\_hdr\_.num\_pages++（页面总数加1）
   * file\_hdr\_.first\_free\_page\_no = page->get\_page\_id().page\_no（新页面成为第一个空闲页）
7. 返回page\_handle

**代码位置**: src/record/rm\_file\_handle.cpp:188-214

#### 3. 方法实现难点

**难点**: 正确初始化新页面的元数据

新页面分配后，必须初始化以下内容：

* 页面头的两个字段（next\_free\_page\_no和num\_records）
* 位图（全部置0表示所有槽位空闲）
* 文件头的页面总数和第一个空闲页号

**解决方法**: 按照顺序依次初始化，确保不遗漏任何字段。特别注意新页面自动成为第一个空闲页，因为它完全空闲。

### 1.7 create\_page\_handle()

#### 1. 方法声明

RmPageHandle create\_page\_handle();

**方法名**: create\_page\_handle

**返回类型**: RmPageHandle

**功能**: 创建或获取一个有空闲空间的页面句柄。如果有空闲页则返回空闲页，否则创建新页

**返回值**: 返回有空闲空间的RmPageHandle对象

#### 2. 方法实现思路

1. 检查文件头的first\_free\_page\_no字段
2. 如果first\_free\_page\_no == RM\_NO\_PAGE（值为-1），说明当前没有空闲页，调用create\_new\_page\_handle()创建新页面并返回
3. 如果first\_free\_page\_no不为-1，说明存在空闲页，调用fetch\_page\_handle(file\_hdr\_.first\_free\_page\_no)获取第一个空闲页并返回

**代码位置**: src/record/rm\_file\_handle.cpp:222-238

#### 3. 方法实现难点

本方法实现简单，无特殊难点。这是一个统一的接口方法，封装了"获取空闲页"的逻辑，调用者无需关心是使用现有空闲页还是创建新页，由该方法自动判断。

### 1.8 release\_page\_handle()

#### 1. 方法声明

void release\_page\_handle(RmPageHandle& page\_handle);

**方法名**: release\_page\_handle

**返回类型**: void

**功能**: 当页面从已满状态变为未满状态时调用，将页面重新加入空闲页面链表

**参数列表**:

|  | **page\_handle** |
| --- | --- |
| **类型** | RmPageHandle& |
| **含义** | 要释放的页面句柄引用 |

#### 2. 方法实现思路

采用头插法将页面插入到空闲链表头部：

1. 将当前页面的next\_free\_page\_no设置为文件头的first\_free\_page\_no：  
   page\_handle.page\_hdr->next\_free\_page\_no = file\_hdr\_.first\_free\_page\_no
2. 将文件头的first\_free\_page\_no更新为当前页面的页面号：  
   file\_hdr\_.first\_free\_page\_no = page\_handle.page->get\_page\_id().page\_no

这样，当前页面就成为了空闲链表的第一个节点。

**代码位置**: src/record/rm\_file\_handle.cpp:243-256

#### 3. 方法实现难点

**难点**: 理解空闲页面链表的维护机制

空闲页面链表是通过文件头和页面头中的页面号字段构成的单向链表：

* 文件头的first\_free\_page\_no指向链表头
* 每个页面头的next\_free\_page\_no指向链表中的下一个节点
* RM\_NO\_PAGE（-1）表示链表结束

**解决方法**: 采用头插法是最简单的方式，只需要两步操作即可完成插入。这种方法的时间复杂度为O(1)。

## 二、RmScan类方法实现

RmScan类负责遍历文件中的所有记录，提供迭代器功能。它内部维护一个Rid指向当前扫描位置。

### 2.1 RmScan() 构造函数

#### 1. 方法声明

RmScan(const RmFileHandle \*file\_handle);

**方法名**: RmScan（构造函数）

**功能**: 初始化记录扫描器，设置扫描的起始位置为第一个存放了记录的位置

**参数列表**:

|  | **file\_handle** |
| --- | --- |
| **类型** | const RmFileHandle\* |
| **含义** | 要扫描的文件句柄指针 |

#### 2. 方法实现思路

1. 保存文件句柄指针到成员变量file\_handle\_
2. 初始化内部Rid为第一个数据页的第一个槽位之前的位置：
   * rid\_.page\_no = RM\_FIRST\_RECORD\_PAGE（值为1，因为0号页是文件头页）
   * rid\_.slot\_no = -1（设为-1是为了让next()从第0个槽位开始查找）
3. 调用next()方法找到第一个实际存放了记录的位置

**代码位置**: src/record/rm\_scan.cpp:18-29

#### 3. 方法实现难点

**难点**: 理解为什么slot\_no要初始化为-1

因为next()方法会从rid\_.slot\_no + 1开始查找，所以如果要从第0个槽位开始查找，就需要将slot\_no初始化为-1。

### 2.2 next()

#### 1. 方法声明

void next() override;

**方法名**: next

**返回类型**: void

**功能**: 将扫描器移动到下一个存放了记录的位置

#### 2. 方法实现思路

使用循环遍历页面和槽位：

1. 外层循环：遍历所有数据页，条件为rid\_.page\_no < file\_handle\_->file\_hdr\_.num\_pages
2. 对于当前页面：
   * 调用file\_handle\_->fetch\_page\_handle(rid\_.page\_no)获取页面句柄
   * 调用Bitmap::next\_bit(true, page\_handle.bitmap, num\_records\_per\_page, rid\_.slot\_no)查找下一个位图值为1的槽位（true表示查找值为1的位）
   * 释放页面：buffer\_pool\_manager\_->unpin\_page(..., false)（false表示未修改页面）
   * 如果找到的slot\_no小于每页记录数，说明找到了记录，更新rid\_.slot\_no并返回
   * 如果没找到，说明当前页面没有更多记录
3. 移动到下一个页面：
   * rid\_.page\_no++
   * rid\_.slot\_no = -1（重置为-1，从下一页的第0个槽位开始）
4. 如果所有页面都遍历完，设置末尾标记：
   * rid\_.page\_no = RM\_NO\_PAGE（-1）
   * rid\_.slot\_no = -1

**代码位置**: src/record/rm\_scan.cpp:34-65

#### 3. 方法实现难点

**难点1**: 跨页面扫描的逻辑

需要正确处理页面边界：当当前页面没有更多记录时，移动到下一个页面并重置slot\_no为-1。

**解决方法**: 使用外层while循环遍历页面，内层使用Bitmap::next\_bit()查找记录。每次移动到新页面时重置slot\_no。

**难点2**: 正确释放页面

每次获取页面后必须释放（unpin），否则会导致缓冲池资源耗尽。

**解决方法**: 在每次循环中，使用完页面后立即调用unpin\_page()，传入false表示未修改页面。

### 2.3 is\_end()

#### 1. 方法声明

bool is\_end() const override;

**方法名**: is\_end

**返回类型**: bool

**功能**: 判断扫描器是否已经到达文件末尾

**返回值**: 如果到达末尾返回true，否则返回false

#### 2. 方法实现思路

1. 检查当前rid的page\_no是否等于RM\_NO\_PAGE（值为-1）
2. 如果相等，说明已经扫描完所有记录，返回true
3. 否则返回false

**代码位置**: src/record/rm\_scan.cpp:70-76

#### 3. 方法实现难点

本方法实现简单，无特殊难点。只需要判断一个条件即可。RM\_NO\_PAGE是在next()方法中设置的末尾标记。

## 三、数据结构说明

### 3.1 RmFileHdr (文件头)

位于src/record/rm\_defs.h:22-28

struct RmFileHdr {  
 int record\_size; // 每条记录的大小（字节）  
 int num\_pages; // 文件中的页面总数  
 int num\_records\_per\_page; // 每页最多存储的记录数  
 int first\_free\_page\_no; // 第一个有空闲空间的页面号  
 int bitmap\_size; // 每页位图的大小（字节）  
};

**说明**: 文件头存储在文件的第0号页面，记录文件的元数据信息。

### 3.2 RmPageHdr (页面头)

位于src/record/rm\_defs.h:31-34

struct RmPageHdr {  
 int next\_free\_page\_no; // 下一个有空闲空间的页面号  
 int num\_records; // 当前页面中的记录数  
};

**说明**: 每个数据页面的页面头，用于维护空闲链表和记录数统计。

### 3.3 RmPageHandle (页面句柄)

位于src/record/rm\_file\_handle.h:24-41

struct RmPageHandle {  
 const RmFileHdr \*file\_hdr; // 文件头指针  
 Page \*page; // 页面实际数据  
 RmPageHdr \*page\_hdr; // 页面头指针  
 char \*bitmap; // 位图指针  
 char \*slots; // 记录槽位区域指针  
  
 // 返回指定slot\_no的slot存储地址  
 char\* get\_slot(int slot\_no) const {  
 return slots + slot\_no \* file\_hdr->record\_size;  
 }  
};

**说明**: 页面句柄封装了页面的内部结构，提供便捷的访问接口。

### 3.4 Rid (记录标识符)

位于src/defs.h

struct Rid {  
 int page\_no; // 页面号  
 int slot\_no; // 槽位号  
};

**说明**: Rid唯一标识一条记录的位置。

## 四、核心机制说明

### 4.1 分槽页面布局

每个数据页面采用分槽布局，结构如下：

+------------------+  
| Page Header | <- RmPageHdr (8字节，包含next\_free\_page\_no和num\_records)  
+------------------+  
| Bitmap | <- 位图，每个bit对应一个槽位（1表示已用，0表示空闲）  
+------------------+  
| Slot 0 | <- 记录槽位0  
+------------------+  
| Slot 1 | <- 记录槽位1  
+------------------+  
| ... |  
+------------------+  
| Slot N-1 | <- 记录槽位N-1  
+------------------+

槽位数量由记录大小和页面大小决定，计算公式在文件创建时确定。

### 4.2 空闲页面链表

使用单向链表管理有空闲空间的页面：

file\_hdr.first\_free\_page\_no → Page A → Page C → Page D → RM\_NO\_PAGE(-1)  
 (next=C) (next=D) (next=-1)  
  
Page B (已满，不在链表中)

* 文件头的first\_free\_page\_no指向链表头
* 每个页面头的next\_free\_page\_no指向下一个空闲页
* 页面满时从链表中移除，变未满时重新加入链表头部（头插法）
* RM\_NO\_PAGE（-1）表示链表结束

### 4.3 位图机制

使用位图（Bitmap）标记每个槽位的使用状态：

* **位值为1**: 槽位已使用，存放了记录
* **位值为0**: 槽位空闲，可以插入记录

Bitmap类提供的关键静态方法：

* set(bitmap, pos): 将pos位置1
* reset(bitmap, pos): 将pos位置0
* is\_set(bitmap, pos): 判断pos位是否为1
* first\_bit(bit, bitmap, max\_n): 查找第一个值为bit的位
* next\_bit(bit, bitmap, max\_n, curr): 从curr+1开始查找下一个值为bit的位

## 五、测试说明

### 5.1 单元测试

**测试文件位置**: src/test/storage/record\_manager\_test.cpp

**编译和运行测试**:

cd build  
make record\_manager\_test  
./bin/record\_manager\_test

### 5.2 测试覆盖内容

* 记录的插入操作（单条、多条、跨页面）
* 记录的查询操作（存在的记录、不存在的记录）
* 记录的更新操作
* 记录的删除操作（单条、多条、删除后页面状态变化）
* 记录迭代器的遍历功能（空文件、单页、多页）
* 边界情况处理（页面满、文件空等）

## 六、实现总结

### 6.1 关键技术点

1. **位图管理**: 使用位图高效标记槽位使用状态，时间复杂度O(n)
2. **空闲链表**: 使用单向链表管理空闲页面，头插法操作时间复杂度O(1)
3. **堆文件组织**: 不关心记录插入位置，简化了实现逻辑
4. **缓冲池集成**: 所有页面访问通过缓冲池管理器，实现页面缓存

### 6.2 注意事项

1. **页面固定与释放**: 所有通过fetch\_page或new\_page获取的页面，使用完毕后必须unpin\_page
2. **脏页标记**: 修改页面数据后，必须在unpin\_page时传入true标记为脏页
3. **空闲链表维护**: 页面满/未满状态变化时，必须正确维护空闲链表
4. **元数据一致性**: 修改记录时，必须同步更新位图、页面头、文件头的相关字段
5. **异常处理**: 访问不存在的页面或记录时，应正确处理（返回nullptr或抛出异常）

**文档结束**