浙江工业大学

本科毕业设计说明书（论文）

（2013 届）

论文题目　基于内存数据库的大数据应用系统的设计与实现

作者姓名	陈佳鹏
指导教师	陈　波
学科 (专业)	软件工程
所在学院	计算机科学与技术学院
提交日期	2013 年 06 月

摘　　要

内存数据库是最近很热门的技术，它将数据完全放在内存中，这样做的好处是对数据有极高的处理速度，而传统的硬盘只用做数据库的持久化备份。在这之中，具有代表性的 Redis 是一个高性能的 Key-Value 数据库系统，在此基础上，本内存数据库系统结合 SQLite 进行开发，从而在 Redis 的基础上实现了基础 SQL 语句的操作，上层 GUI 基于 PyQt4 开发。主要涉及的技术包括数据的存储，数据的持久化等。

本内存数据库应用系统在 Redis 原命令的基础上支持一些基础的 SQL 操作（Create，Select 等），同时根据 Redis Server 的配置，在持久化时根据 SQLite 的特性将整个数据库保存成单个文件的形式。与此同时，拿其与比较热门的关系型数据库 MySQL（5.5）进行了对比，比较了两者在读写速度上的差异。

最后一部分包含了本系统和 MySQL5.5 性能对比的结果，并做了一定的分析，评价和总结。

关键字： Redis，内存数据库，SQLite

Abstract

Recently, main memory database is a very hot technology, in a main memory database, the whole database is entirely in memory, which can provide extremely high data processing speed, and the hard disk is only used for persistent backup of the database. Among the main memory databases, Redis is a representative Key-Value database system with high-performance. Our memory database is based on Redis wihich combined with SQLite, which implements the basic SQL statements of operations on redis. Also, the GUI development is based on PyQt4. Mainly related technology includes data storage, data persistence, etc.

Our main memory database system supports not only Redis's original command but also some basic SQL operations (Create, Select, etc.). At the same time, according to the Redis Server configuration, the entire SQLite database dumps into a single file. Also, we compared the main memory databse with the popular relational database MySQL (5.5), including the read/write speed difference between them.

The last section contains the compared results of two database, followed by some analysis, evaluation and summary.

Keywords: Redis, Main Memory Database 2, SQLite

目　　录

图 目 录

表 目 录

第一章 绪论

1.1 研究背景

这几年是互联网的高速发展期，各种类型的应用层出不穷，这对相关技术方面提出了更多的要求。用于数据储存传统关系型数据库（比如 SQL Server 等）面临着越来越多的挑战，这些挑战主要体现在以下几个方面：

低延迟 I/O、海量级别的数据和流量、大规模集群监控管理、日益增长运营成本。

鉴于上文所提及的那些挑战，时下热门的内存数据库越来越展现其超强的能力。首先，把数据保存在内存中能极大地提高程序应用的性能，这也导致数据必须重新设计组织，使得数据在内存中以新的方式存储（下文会做详细的介绍）。其次，内存数据库的原理是通过内存资源作为牺牲来换取数据处理的实时性，简单的说就是“用空间来换取时间”。然而，内存数据库并不是完全的将所有数据都保存在内存中，但是，这个前提是需要大内存量的支持。

但是，并不能直接认为内存数据库就能取代关系型数据库。因为两者的出发点并不相同，或者说两者所针对的方面不同。对于关系型数据库，它的重点在于解决大容量数据的储存和分析问题，而内存数据库的重点在于解决数据的实时处理和高并发问题。两者是相辅相成的，内存数据库在事务的实时处理要比关系型数据库强，但数据安全稳定方面不能和其相比了。

所以，在实际应用中，通常是两种数据库结合使用，而不是完全以内存数据库代替传统数据库。

1.2 内存数据库产品

1.2.1 Oracle TimesTen

作为一个关系型的内存数据库，TimesTen 功能全面，它运行在应用层，从而缩短处理响应时间和提高吞吐量。话句话说，TimesTen 是磁盘数据库的“Cache”，通过物理内存中的数据存储区的直接操作，减少了到磁盘间的 I/O 交互。

1.2.2 SAP HANA

SAP HANA 是一款面向数据源的、灵活、多用途的内存应用设备，整合了基于硬件优化的 SAP 软件模块，通过 SAP 主要硬件合作伙伴提供给客户。SAP HANA 提供灵活、节约、高效、实时的方法管理海量数据。利用 HANA，企业可以不必运行

多个数据仓库、运营和分析系统，从而削减相关的硬件和维护成本。HANA 将在内存技术基础上，为新的创新应用程序奠定技术基础，支持更高效的业务应用程序，如：计划、预测、运营绩效和模拟解决方案 [8]。

1.2.3 Redis

作为一个 Key-Value 储存系统，Redis 支持存储的 value 类型很多，包括字符串、链表、集合和有序集合。为了保证效率，数据都是缓存在内存中，同时 Redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了主从同步 [22]。

1.2.4 SQLite

SQLite 是一个用 C 语言编写的小型的、轻量级的、绿色、开源、轻便的数据库。它最大的特点是没有类型的概念，说明可以保存任何类型的数据到表的任何位置中。此外，它具有很强的移植性，可以运行在 Windows、Linux、BSD、Mac OS X 和一些商用 UNIX 系统上 [23]。

1.3 本文主要工作

本文在内存数据库的相关基础上，拿 Redis 作为例子，分析了其底层相关的数据结构，以及其持久化方式。在底层数据结构方面，首先，结合部分伪代码介绍了最基础的内部数据结构，包括 char* 的“替代品”sds，双端列表，字典（映射），跳跃表等，这些都是底层数据结构，是上层字符串或者数据结构（如 Hash 表，列表，集合，有序集合等）的底层实现。此外，在此基础上了，介绍了两个 Redis 内存映射数据结构（整数集合，压缩列表），并结合伪代码分析其工作原理。然后，介绍了 Redis 目前支持的两种持久化方式，并对两种方式进行比较。

最后，本文阐述了基于 Redis 和 SQLite 开发的内存数据库系统的实现方案，以及主要功能模块的实现流程，以及本内存数据库在性能上和关系型数据库的差异。

1.4 本文的组织结构

本文共分为八章，以内存数据库为背景，研究讨论了基于 Redis 并支持 SQL 架构的内存数据库，详细阐述了如何利用该框架技术对系统的模块进行设计与实现，各章内容如下：

第一章，介绍了内存数据库研究的背景，一些相关产品和本文的主要工作。

第二章，详细介绍了内存数据库系统开发的方法与技术。

第三章，重点介绍了数据的存储结构。

第四章，具体介绍了数据的持久化。

第五章，详细介绍了基于 Redis 和 SQLite 的内存数据库的实现。

第六章，与 MySQL（5.5）进行性能对比。

第七章，对系统开发进行总结并提出下一步工作。

第二章 方法与技术

上文说到，Redis 并非是传统的关系型数据库，无法支持 SQL 语句解析，所以本系统在这基础上配合采用了 SQLite 的接口，同时为 Redis 新增加了一个基于 SQLite 的“sql”命令。至此，可以通过 sql 命令来进行数据库的表操作，表中的所有数据完全存储在内存中，此外，根据 Redis.conf 的配置，可以设置表中数据库每次保存到硬盘的间隔，这样做可以保证数据的正确性，防止出现可能的断电宕机使数据丢失的情况。

本内存数据库系统采用了后端 SQLite 的 C 语言接口嵌入 Redis，前端 GUI 使用 PyQT4 的架构进行开发，实现了支持一些基础 SQL 语句的内存数据库应用系统，本章将对上述知识进行简要的阐述，主要具体介绍 Redis，包括其一些原理和常用的应用场景，还有 PyQT4 模块的一些基本模块知识。

2.1 Redis

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。作为 Key-value 型数据库，Redis 也提供了键（Key）和键值（Value）的映射关系。但是，除了常规的数值或字符串，Redis 的键值还可以是以下形式之一：

Lists（列表）

Sets（集合）

Sorted sets（有序集合）

Hashes（哈希表）

键值的数据类型决定了该键值支持的操作。Redis 支持诸如列表、集合或有序集合的交集、并集、差集等高级原子操作；同时，如果键值的类型是普通数字，Redis 则提供自增等原子操作。通常，Redis 将数据存储于内存中，或被配置为使用虚拟内存。通过两种方式可以实现数据持久化：使用快照的方式，将内存中的数据不断写入磁盘；或使用类似 MySQL 的日志方式，记录每次更新的日志。前者性能较高，但是可能会引起一定程度的数据丢失；后者相反。

相比需要依赖磁盘记录每个更新的数据库，基于内存的特性无疑给 Redis 带来了非常优秀的性能。读写操作之间没有显著的性能差异，如果 Redis 将数据只存储于内存中。下文简单列举了 Redis 在当下 Web 应用开发中一些常用的场景，其中结合了相关的 Redis 命令作为具体场景使用的介绍：

（1）保存个人主页中内容列表（如微博新鲜事）。Redis 中的列表操作“LPUSH”用来插入一个内容 ID，作为关键字存储在列表头部，同时配合 LTRIM 来限制列表中的项目数。

（2）相关排行榜。Redis 中的有序集合操作“ZADD”命令可以直接实现这个功能，此外，命令“ZREVRANGE”可以用来按照得分来获取前几名的数据。

（3）队列。除了“push”和“pop”类型的命令之外，Redis 还有阻塞队列的命令，能够让一个程序在执行时被另一个程序添加到队列。

（4）缓存。对于 Redis 这种“Key-Value”系统而言，用它来实现缓存会很轻松、方便、高效。如果你想自己专门编写代码来完成，这样的开销相对而言可能太大。

2.2 SQLite

上文已经对 SQLite 做了的基本的介绍，由于本内存数据库系统基于 Redis 和 SQLite 的接口开发，所以这里主要介绍 SQLite 的 C 语言接口部分，几个主要并且常用的接口如下：

（1）

```
int sqlite3_open(
  const char *filename,
  sqlite3 **ppDb
);
```

打开指定数据库文件，将 **ppdb 绑定到数据库连接对象，返回打开结果代码。因为其他接口函数一般都需要一个指向数据库文件的指针，所以一般这个函数在最前面调用，为其他函数做准备工作，如果数据库文件不存在，则自动新建一个。

（2）

```
int sqlite3_prepare(
  sqlite3 *db,
  const char *zSql,
  int nByte,
  sqlite3_stmt **ppStmt,
  const char **pzTail
);
```

将 UTF-8 编码的 SQL 语句编译成字节码,将结果保存到 **ppStmt,以便后面的执行函数方便执行。而 sqlite3_prepare16() 则是 UTF-16 编码的版本。

(3)

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
```

撤销准备好的 SQL 声明(sqlite3_stmt),当数据库连接关闭的时候,所有准备好的 SQL 声明都必须被释放销毁。

(4)

```
int sqlite3_close(sqlite3*);
```

关闭之前通过 sqlite3_open() 打开的数据库文件连接对象。

(5)

```
int sqlite3_exec(
  sqlite3*,
  const char *sql,
  int (*callback)(void*,int,char**,char**),
  void *,
  char **errmsg
);
```

编译和执行多条 SQL 语句,将查询的结果返回给回调函数,如果执行错误,将错误记录到 errmsg。

2.3 PyQT4

PyQT 是一个生成图形应用程序的工具包。是 python 语言和成功的 Qt 库的绑定。Qt 库是这个世界上最强大的库之一,PyQT 作为一组 python 的模块来实现。它包含了超过 300 个类,将近 6000 个函数和方法。它是一个多平台的工具包,可以在所有的主流操作系统上运行,包括 Unic,Windows 和 Mac。PyQT 采用双协议,开发者可以在 GPL 和商业授权中选择。以前的版本中,GPL 版本只存在于 Unic 上。从 PyQT4 开始,GPL 协议支持所有的平台。QtCore 模块包含了核心的非图形功能,这个模块被用来实现时间,文件和目录,不同的数据格式,流,互联网地址,mime 类型,线程或进程等等。QtGui 模块包含了图形组件和类的描述,包括例如按钮,窗口,状态栏,滑块,位图,颜色,字体等等 [24]。

QtCore 模块包含了核心的非图形功能,这个模块被用来实现时间,文件和目录,不同的数据格式,流,互联网地址,mime 类型,线程或进程等等。

QtGui 模块包含了图形组件和类的描述，包括例如按钮，窗口，状态栏，滑块，位图，颜色，字体等等。

QtNetwork 模块包含了网络编程所需的类，这些类可以用来实现 TCP/IP 和 UDP 的客户端/服务器程序，使得网络编程更加简单更加可移植。

QtXml 模块提供了处理 xml 文件的类，这个模块包含了 SAX 和 DOM APIs 的实现。

QtSql 模块提供了处理数据库的类。

第三章 数据的存储

3.1 内部数据结构

本章主要介绍 Redis 的内部数据结构、内存映射数据结构以及一些与其相关的应用场景。Redis 和其他很多 key-value 数据库的不同之处在于,Redis 不仅支持简单的字符串键值对,它还提供了一系列数据结构类型值,比如列表、哈希、集合和有序集,并在这些数据结构类型上定义了一套强大的 API。通过对不同类型的值进行操作,Redis 可以很轻易地完成其他只支持字符串键值对的 key-value 数据库很难(或者无法)完成的任务。在 Redis 的内部,数据结构类型值由高效的数据结构和算法进行支持,并且在 Redis 自身的构建当中,也大量用到了这些数据结构。

本节将对其使用的数据结构和算法进行简单的介绍,并介绍了这些数据结构和算法的应用场景。

3.1.1 动态字符串

Redis 是一个“Key-Value”数据库,数据库的值可以是字符串、集合、列表等多种类型的对象,而数据库的键则总是字符串对象,其底层所使用的字符串对象是用 sds(Simple Dynamic String)表示,其结构如下:

```
typedef char *sds;
struct sdshdr {
    int len;
    int free;
    char buf[];
};
```

其中 len 表示已使用的长度,free 表示剩余的长度,使用 sds 而不是 char* 的原因主要是:char* 的功能单一,抽象层次低,不能高效地支持一些 Redis 常用的操作(比如追加操作和长度计算操作)。除此之外,通过 len 属性,sdshdr 可以实现复杂度为 $O(1)$ 的长度计算操作。另一方面,通过对 buf 分配一些额外的空间,并使用 free 记录未使用空间的大小,sdshdr 可以让执行追加操作所需的内存重分配次数大大减少。

3.1.2 双端列表

双端链表作为一种通用的数据结构，在 Redis 内部使用得非常多：它既是 Redis 列表结构的底层实现之一，还被大量 Redis 模块所使用，用于构建 Redis 的其他功能，由于双端列表的设计实现比较常见，下文简单的阐述其结构如下：

```
typedef struct listNode {
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;

typedef struct list {
    listNode *head;
    listNode *tail;
    unsigned long len;
    void *(*dup)(void *ptr);
    void (*free)(void *ptr);
    int (*match)(void *ptr, void *key);
} list;
```

对于一个链表节点 listNode，它包括了前驱和后驱节点以及节点值的指针，而链表本身为包括了列表头尾节点的指针、列表长度以及三个关键函数的指针（节点值的拷贝、释放、比较函数）。这三个函数指针设置的目的在于对于不同类型的值，有时候需要不同的函数来处理这些值，因此，这三个函数分别用来处理值的复制、释放和比较。

举个例子：当删除一个 listNode 时，如果包含这个节点的 list 的 list->free 函数不为空，那么删除函数就会先调用 list->free(listNode->value) 清空节点的值，再执行余下的删除操作（比如说，释放节点）。

综上所述，双端列表的优点可以归结为以下几点：

（1）节点带有前驱和后继指针，访问前驱节点和后继节点的复杂度为 O(1)，并且对链表的迭代可以在从表头到表尾和从表尾到表头两个方向进行。

（2）链表带有指向表头和表尾的指针，因此对表头和表尾进行处理的复杂度为 O(1)。

（3）链表带有记录节点数量的属性，所以可以在 O(1) 复杂度内返回链表的节点数量（长度）。

3.1.3 字典

字典,也就是我们常说的 map,是一种抽象数据结构,由系列键值对组成,各个键值对的键各不相同,程序可以将新的键值对添加到字典中,或者基于键进行查找、更新或删除等操作。下文简单的介绍了字典的结构:

```
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;

typedef struct dict {
    dictht ht[2];
    int rehashidx;
    //...
} dict;
```

一个字典由两个 hash 表组成,0 号 hash 表(ht[0])是字典主要使用的 hash 表,而 1 号 hash 表则只有在 Server 对 0 号 hash 表进行 rehash 时才使用,rehashidx 表示 rehash 进度的标志。而 rehash 的作用是为了维护 hash 表的效率,例如:当 hash 表的碰撞率很高并且 table[i] 挂着很长一个链表时,查找效率会大大下降,这个时候可以通过 rehash 对 hash 表进行扩容的操作。

字典在 Redis 中的应用广泛,主要用途有以下两个:

1. 实现数据库键空间。Redis 是一个键值对数据库,数据库中的键值对就由字典保存,当添加一个键值对到数据库时(不论键值对是什么类型),程序就将该键值对添加到这个字典,同理,当用户从数据库中删除一个键值对时,程序就会将这个键值对从字典中删除。

2. 用作 Hash 类型键的其中一种底层实现。

3.1.4 跳跃表

跳跃表的介绍这里就不再详细的阐述了,很多和数据结构有关的书上都有相关的介绍,这种数据结构以有序的方式在层次化的链表中保存元素,它的效率可以和平衡树媲美(查找、删除、添加等操作都可以在对数期望时间下完成),并且比起平衡树来说,跳跃表的实现要简单直观得多。下面对 Redis 中使用的跳跃表结构进行简单的介绍:

```
typedef struct zskiplistNode {
    robj *obj;
    double score;
    struct zskiplistNode *backward;
    struct zskiplistLevel {
        struct zskiplistNode *forward;
        unsigned int span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    struct zskiplistNode *header, *tail;
    unsigned long length;
    int level;
} zskiplist;
```

相比于传统的跳跃表的节点,redis 增加了一个前驱节点的指针,这样的好处使得对跳跃表进行反向遍历。对于跳跃表节点,包含了对象的指针(obj)和 score,以及跳跃层,每一个跳跃层包括了跳跃到的下一个节点以及这次跳跃所跨越的节点数。对于跳跃表本身,和双端列表一样,维护头尾指针,节点数量,还有每个节点的跳跃层的数量。

传统的跳跃表有个特点:不能包含相同的 score。为了满足自身的功能,跳跃表可以允许重复相同的 score 值。那么这样一来,在进行节点的对比操作的时候,如果 score 值相同,单靠 score 值无法判断一个元素的身份,需要连 obj 都一并检查才行。

和字典、链表或者 sds 这种大量使用的数据结构不同,跳跃表在 Redis 的唯一作用,就是实现有序集数据类型。跳跃表将指向有序集的 score 值和 obj 指针作为元素,并以 score 值为索引,对有序集元素进行排序。这个功能的实用场景很多,例如上文提及过的排行榜排序功能。

3.2 内存映射数据结构

上文阐述了内部数据结构和算法以及实际应用场景,虽然这些内部数据结构非常强大,但是创建一系列完整的数据结构本身也是一件相当耗费内存的工作,这就会产生一个问题:当一个对象包含的元素数量并不多,或者元素本身的体积并不大时,使用代价高昂的内部数据结构并不是最好的办法。

所以，为了解决这种问题，在这种情况下，Redis 会使用内存映射数据结构来代替内部数据结构。内存映射数据结构是一系列经过特殊编码的字节序列，创建它们所消耗的内存通常比作用类似的内部数据结构要少得多，可以为用户节省大量的内存。

但是，内存映射数据结构的编码和操作方式要比内部数据结构要复杂得多，所以内存映射数据结构所占用的处理时间会比作用类似的内部数据结构要多，简而言之，内存映射数据结构是一种牺牲时间换取空间的做法。下文将对两种内存映射数据结构进行介绍。

3.2.1 整数集合

整数集合以数组储存的方式有序、无重复地保存多个整数值，它会根据元素的值，自动选择该用什么长度的整数类型（int16_t、int32_t、int64_t）来保存元素。举个例子，如果在一个整数集合里面，最大的整数可以用 int16_t 类型来保存，那么这个整数集合的所有元素都应该以 int16_t 类型来保存。

这样也会产生出一个问题：如果有一个新元素要加入到这个 intset，并且这个元素不能用 int16_t 类型来保存（int_32t 或者 int64_t），那么这个 intset 就会自动进行扩展，也就是先将集合中现有的所有元素从 int16_t 类型转换为相应的类型，接着再将新元素加入到集合中。根据需要，整数集合可以自动从 int16_t 扩展到 int32_t 或 int64_t，或者从 int32_t 扩展到 int64_t。

整数集合的定义结构如下：

```
typedef struct intset {
    uint32_t encoding;
    uint32_t length;
    int8_t contents[];
} intset;
```

encoding 的值可以是以下三个常量的其中一个：

```
#define INTSET_ENC_INT16 (sizeof(int16_t))
#define INTSET_ENC_INT32 (sizeof(int32_t))
#define INTSET_ENC_INT64 (sizeof(int64_t))
```

contents 数组是实际保存整数的地方，数组中的元素有两个特性：没有重复元素；元素在数组中从小到大排列；这样一来，程序可以使用二分查找算法来实现查找操作，复杂度为 O(lgN)。现在，举个实际的例子，如果整数集合现在保存了 1,3,7，最大是 7，能用 int16_t 来保存，那么 contents 的结构如下所示：

```
value   |  1  |  3  |  7  |
bit     0    15    31    47
```

如果现在添加了一个新整数 999999，int16_t 必须扩展到 int32_t，所以扩展之后的 contents 的结构如下所示：

```
value   |    1     |    3     |    7     |  999999 |
bit     0   15     31   47     63         95        127
```

3.2.2 压缩列表

压缩列表（Ziplist）是由一系列特殊编码的内存块构成的列表，一个压缩列表可以包含多个节点（entry），每个节点可以保存一个长度受限的字符数组（不以 0 结尾的 char 数组）或者整数。更具体的说，压缩列表其实是用一个字符串来实现的双向链表结构，这样做的目的可以减少双向链表的存储空间，主要是节省了链表指针的存储，如果存储前驱和后驱节点的指针一共需要 8 个字节，而转化成存储前驱节点的长度和当前结点长度在大多数情况下可以节省很多空间。

但是，这样设计的储存方式也有不足：如果每次向链表增加元素，那么都需要重新分配内存的工作。压缩列表节点的基本结构如下文所示：

```
|----------------|--------|------|-------|
|pre_entry_length|encoding|length|content|
|----------------|--------|------|-------|
```

其中各个节点域表示的含义如下表 3.1 所示：

表 3.1: 压缩列表节点

域	含义
pre_entry_length	前一个节点的长度（可以用来访问前一个节点）。如果前一节点的长度小于 254 字节，那么只使用一个字节保存。反之，那么将第 1 个字节的值设为 254，然后用接下来的 4 个字节保存实际长度。
encoding	占两个 bit，00、01 和 10 说明 content 表示的是字符数组，11 说明 content 表示的是整数数组。
length	length 所占的 bit 和 encoding 有关。00：encoding 和 length 共占 1 个 byte，即 lenth 占 6 个 bit；01：encoding 和 length 共占 2 个 byte，即 lenth 占 14 个 bit；10：encoding 和 length 共占 5 个 byte，其中第 1 个 byte 剩余 6 个 bit 不记，即 lenth 占 32 个 bit
content	保存着节点的内容，它的类型和长度由 encoding 和 length 决定。

压缩列表本身是由列表头，节点，列表末尾表示符组成的，其中列表头又由列表总字节数（zlbytes），末节点偏移量（zltail）和节点数量（zllen）组成，如下图所示：

```
 4bytes   4bytes  2bytes                          1byte
|--------|-------|------|-------|-----|-------|------|
|zlbytes |zltail |zllen |entry1 | ... |entryN |zlend |
|--------|-------|------|-------|-----|-------|------|
```

其中，计算 zltail 偏移所得到的位置为 entryN 的首地址，由于每个节点内部包含了前一个节点的长度，所以这两者一起实现了类似双端列表中从后向前遍历的功能。最后的 zlend 用来标识列表的结尾，为固定值 1111 1111。

综上所述，类似整数集合，压缩列表也是一个牺牲时间换取空间的做法，当添加和删除 ziplist 节点时候，可能会引起连锁更新，因此，添加和删除操作的最坏复杂度为 $O(N^2)$。

第四章 数据的持久化

什么是持久化，简单来讲就是将数据放到断电后数据不会丢失的设备中，也就是我们通常理解的硬盘上。在运行情况下，Redis 以数据结构的形式将数据维持在内存中，为了让这些数据在 Redis 重启之后仍然可用，需要经常将内存中的数据同步到磁盘来保证持久化。这里有两种持久化模式，RDB（保存数据库快照）和 AOF（记录写命令）。

4.1 RDB

RDB 是默认的持久化方式，将内存中数据以快照的方式写入到二进制文件中，默认的文件名为 dump.rdb。在 Server 运行时，RDB 程序将当前内存中的数据库快照保存到磁盘文件中，在 Redis 重启动时，RDB 程序可以通过载入 RDB 文件来还原数据库的状态。RDB 最核心的是 rdbSave 和 rdbLoad 两个函数，前者用于生成 RDB 文件到磁盘，而后者则用于将 RDB 文件中的数据重新载入到内存中，如下图 4.1 所示：

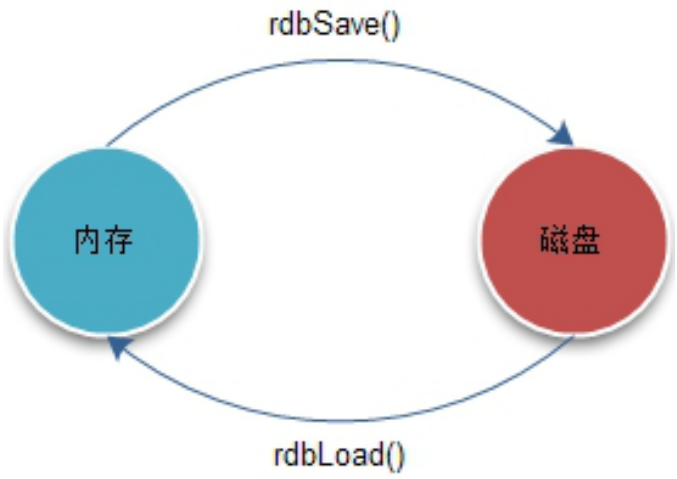

图 4.1: RDB 核心函数

4.1.1 保存

rdbSave 负责将内存中的数据库数据以.rdb 的格式保存到磁盘中。在保存 RDB 文件期间，主进程会被阻塞，直到保存完成为止。SAVE 和 BGSAVE 两个命令都会调用 rdbSave 函数，但它们调用的方式各有不同：

1. SAVE 直接调用 rdbSave，阻塞 Redis 主进程，直到保存完成。在主进程阻塞期间，服务器不能处理客户端的任何请求。

2. BGSAVE 则 fork 出一个子进程，子进程负责调用 rdbSave，并在保存完成之后向主进程发送信号，通知保存已完成。因为 rdbSave 在子进程被调用，所以 Server 在 BGSAVE 执行期间仍然可以继续处理客户端的请求。

下图的伪代码来描述这两个命令：

```
def SAVE():
    rdbSave()

def BGSAVE():
    pid = fork()
    if pid == 0:
        rdbSave()
    elif pid > 0:
        handle_request()
    else:
        # pid == -1
        handle_fork_error()
```

4.1.2 载入

当 Redis 服务器启动时，rdbLoad() 就会被执行，它读取 RDB 文件，并将文件中的数据库数据载入到内存中。在载入期间，请求命令一律返回错误。等到载入完成之后，服务器才会开始正常处理所有命令。此外，因为 AOF（下文会做阐述）文件的保存频率通常要高于 RDB 文件保存的频率，所以一般来说，AOF 文件中的数据会比 RDB 文件中的数据要新。因此，如果服务器在启动时，配置文件里说明打开了 AOF 功能，那么程序优先使用 AOF 文件来还原数据。只有在 AOF 功能未打开的情况下，Redis 才会使用 RDB 文件来还原数据。

4.1.3 RDB 文件结构

RDB 文件的结构也采用了编码的形式，并且相对于整数集合或者压缩列表更加复杂。RDB 文件的整体格式如下：

```
|-----|-------|---------|---------------|---|---------|
|REDIS|VERSION|SELECT-DB|KEY-VALUE-PAIRS|EOF|CHECK-SUM|
|-----|-------|---------|---------------|---|---------|
```

相关域的定义如 表4.1 所示：

表 4.1: RDB 文件结构

域	含义
REDIS	文件前 5 bytes，固定为“REDIS”，用来标识该文件为 RDB 文件。
RDB-VERSION	RDB 文件版本号。对 RDB 文件进行版本分类的原因是不同的版本可能数据编码方式不同，必须通过文件版本号来确定数据的读入方式。
SELECT-DB	数据库编号。
KEY-VALUE-PAIRS	数据库键值对集合（下文中做详细说明）。
EOF	标记结尾，此处的结尾指的是数据库结尾，而非文件结尾，为固定值 255。
CHECK-SUM	整个 RDB 文件内容的校验和，为一个 uint_64t 类型的整数。

从上文可以看出，最重要也是编码最复杂的部分即为数据库中键值对的编码方式，一个键值对的编码格式如下：

```
|--------------------|-------------|---|-----|
|OPTIONAL-EXPIRE-TIME|TYPE-OF-VALUE|KEY|VALUE|
|--------------------|-------------|---|-----|
```

OPTIONAL-EXPIRE-TIME 域是可选的，如果键没有设置过期时间，那么这个域就不会出现；反之，那么它记录着键的过期时间，在当前版本的 RDB 中，过期时间是一个以毫秒为单位的 UNIX 时间戳。

TYPE-OF-VALUE 域记录着 VALUE 域的值所使用的编码，根据这个域的指示，程序会使用不同的方式来保存和读取 VALUE 的值。

KEY 域保存着键，格式和 REDIS_ENCODING_RAW 编码的字符串对象一样（见下文）。

由于 VALUE 保存的数据类型很多，有 String，List，Set，Sorted Set，Hash 等，所以 VALUE 域保存的格式有跟多种，下文按照这个顺序来阐述具体的编码格式：

1. String（通过 REDIS_ENCODING_INT 编码）

   在这种情况下，String 能直接表示成 8 位、16 位或者 32 位的有符号整数，例如“9”可以直接用 0000 1001 来保存，如果超过了 int32_t 的大小，则退化成字符序列的形式保存。一个字符序列结构如下：

```
|---|-------|
|LEN|CONTENT|
|---|-------|
```

其中,LEN 保存了以 byte 为单位的字符长度,CONTENT 域保存了字符内容。当进行载入时,先读入 LEN,创建一个长度等于 LEN 的字符串对象,然后再从文件中读取 LEN 字节数据,并将这些数据设置为字符串对象的值。

2. String(通过 REDIS_ENCODING_RAW 编码)
   如果 String 不能被表示成整数,那么它就按正常的字符串序列的方式进行保存,方式和上文 String(通过 REDIS_ENCODING_INT 编码)中的方法一样,采用“LEN + CONTENT”的结构进行存储。

   但是,如果 Server 配置文件里说明了使用“LZF 压缩算法”的话,存储格式就变成如下所示:

```
|--------|--------------|------------------|
|LZF-FLAG|COMPRESSED-LEN|COMPRESSED-CONTENT|
|--------|--------------|------------------|
```

   最前面的LZF-FLAG标示符说明这是经过LZF算法压缩过的字符串，COMPRESSED-LEN是该字符串的字节长度，COMPRESSED-CONTENT是被压缩后的字符串数据。

3. LIST(通过 REDIS_ENCODING_LINKEDLIST 编码)
   这个通过 LINKEDLIST 的形式来保存一个 LIST,结构如下:

```
|---------|------------|------------|---|------------|
|LIST-SIZE|NODE-VALUE-1|NODE-VALUE-2|...|NODE-VALUE-N|
|---------|------------|------------|---|------------|
```

   其中 LIST-SIZE 保存链表节点数量,之后节点值的保存方式和字符串的保存方式一样。当进行载入时,先读取节点的数量 LIST-SIZE,然后创建一个新的链表,最后一直执行“载入结点,添加到链表”的步骤。

4. Set(通过 REDIS_ENCODING_HT 编码)
   Set 的表示结构和上文的 List 表示结构基本一致,如下所示:

```
|--------|---------|---------|---|---------|
|SET-SIZE|ELEMENT-1|ELEMENT-2|...|ELEMENT-N|
|--------|---------|---------|---|---------|
```

其中 SET-SIZE 记录了集合元素的数量，之后元素值的保存方式和字符串的保存方式一样。当进行载入时，先读入集合元素的数量 SET-SIZE，然后创建一个新的 Hash 表，最后一直执行“载入字符串，添加到 Hash 表”的步骤。

5. Sorted Set（通过 REDIS_ENCODING_SKIPLIST 编码）
   Sorted Set 的表示结构和上文的 List 表示结构基本一致，其中一个节点分成了成员和分数，如下所示：

```
|---------|--------|-------|---|--------|-------|
|ZSET-SIZE|MEMBER-1|SCORE-1|...|MEMBER-N|SCORE-N|
|---------|--------|-------|---|--------|-------|
```

   其中 ZSET-SIZE 记录了集合元素的数量。当进行载入时，先读取有序集元素数量，创建一个新的 Skiplist，最后一直执行“载入 member（字符串），载入 score（字符串），添加到新的 Skiplist”的步骤。

6. Hash（通过 REDIS_ENCODING_HT 编码）
   Hash 的表示结构和上文的 List 表示结构基本一致，其中一个节点分成了键和值，如下所示：

```
|---------|-----|-------|---|-----|-------|
|HASH-SIZE|KEY-1|VALUE-1|...|KEY-N|VALUE-N|
|---------|-----|-------|---|-----|-------|
```

   其中 HASH-SIZE 记录了 Hash 表键值对的数量。当进行载入时，先读取 Hash 表大小，创建一个新的 Hash 表，最后一直执行“载入 key（字符串），载入 value（字符串），添加到新的 Hash 表”的步骤。

7. List、Hash、Zset（通过 REDIS_ENCODING_ZIPLIST 编码）
   List、Hash、Zset 可以通过压缩列表（ziplist）来保存，保存方式如下：

```
|---|-------|
|LEN|ZIPLIST|
|---|-------|
```

   当进行载入时，先读入压缩列表长度 LEN，再根据 LEN 读入数据，最后将数据还原成一个 ziplist。

8. Set(通过 REDIS_ENCODING_INTSET 编码)
   当 Set 可以用 intset 来保存时,保存方式如下:

```
|---|------|
|LEN|INTSET|
|---|------|
```

   当进行载入时,先读入压缩列表长度 LEN,再根据 LEN 读入数据,最后将数据还原成一个 intset。

4.2 AOF

上文介绍了快照(RDB)形式的持久化方式,与之想比的 AOF 则以协议文本的方式,将所有对数据库进行过写入的命令(及其参数)记录到 AOF 文件,以此达到记录数据库状态的目的。AOF 比 RDB 有更好的持久化性,原因在于使用 AOF 持久化时,Server 会将收到的写命令追加到文件中(默认是 appendonly.aof)。当 redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。但是,由于操作系统会在内核中缓存修改,所以那些写命令可能不是立即写到硬盘上。这样 AOF 方式的持久化也还是有可能会丢失部分对数据的修改。整个 AOF 流程如图 4.2 所示:

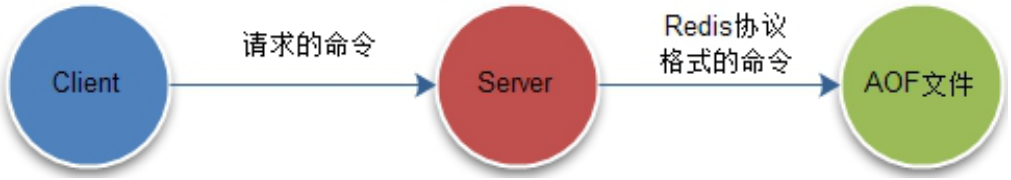

图 4.2: AOF 持久化流程

服务器将命令同步到 AOF 文件的整个过程可以分为三个阶段:

1. 命令传输:服务器将执行完的命令、命令的参数等信息发送到 AOF 程序中。
2. 追加缓存:AOF 程序根据接收到的命令数据,将命令转换为 Redis 协议(见下文)的格式,然后将协议内容追加到服务器的 AOF 缓存中。
3. 写入和保存:AOF 缓存中的内容被写入到 AOF 文件末尾,如果设定的 AOF 保存条件被满足的话,fsync() 或者 fdatasync() 会被调用,将写入的内容真正地保存到磁盘中。

4.2.1 命令传输

当一个客户端需要执行命令时,它通过网络连接,将协议文本发送给服务器。Redis 的协议标准如下:

*<参数数量>\r\n$<第1个参数字节数>\r\n<参数数据>\r\n...$<第N个参数字节数>\r\n<参数数据>\r\n

比如说,要执行命令“SET KEY VALUE”,客户端将向服务器发送文本*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n。服务器在接到客户端的请求之后,它会根据协议文本的内容,选择适当的命令函数,每当命令函数成功执行之后,命令参数都会被传播到 AOF 程序。

4.2.2 追加缓存

当命令被传输到 AOF 程序之后,程序会根据命令以及命令的参数,将命令从字符串对象转换回原来的协议文本。协议文本生成之后,它会被追加到服务器管理的缓存的末尾。

4.2.3 写入和保存

每当服务器常规任务函数被执行、或者事件处理器被执行时,aof.c/flushAppendOnlyFile 函数都会被调用,这个函数执行以下两个工作:

WRITE:根据条件,将 aof_buf 中的缓存写入到 AOF 文件。

SAVE:根据条件,调用 fsync 或 fdatasync 函数,将 AOF 文件保存到硬盘中。

两个步骤都需要根据一定的条件来执行,而这些条件由 AOF 所使用的保存模式来决定,具体条件由配置文件中 appendfsync 的值有关。

4.3 AOF 重写

在 AOF 模式下,每一条写命令都生成一条日志记录,那么这个 AOF 文件会越来越大。所以必须在某些条件下对 AOF 文件进行缩小体积的操作。因此,Redis 提供了 AOF 重写的功能。其功能就是重新生成一份 AOF 文件,新的 AOF 文件中一条记录的操作只会有一次(最新值),而不像一份老文件那样,可能记录了对同一个值的多次操作。其生成过程和 RDB 类似,也是 fork 一个进程,直接遍历所有数据,写入新的 AOF 临时文件。在写入新文件的过程中,所有的写操作日志还是会写到原来老的 AOF 文件中,同时还会记录在内存缓冲区中。当重完操作完成后,会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的 rename 命令用新的 AOF 文件取代老的 AOF 文件。

结合上文提到的 RDB 模式,两者操作都是顺序 I/O 操作,性能都很高。而同

时在通过 RDB 文件或者 AOF 文件进行数据库恢复的时候，也是顺序的读取数据加载到内存中。所以也不会造成磁盘的随机读，实现高效率的恢复方式。

第五章 系统实现

5.1 相关功能

本内存数据库系统功能主要分两部分：Redis 原命令部分，SQL 部分，系统的功能结构如图 5.1 所示。

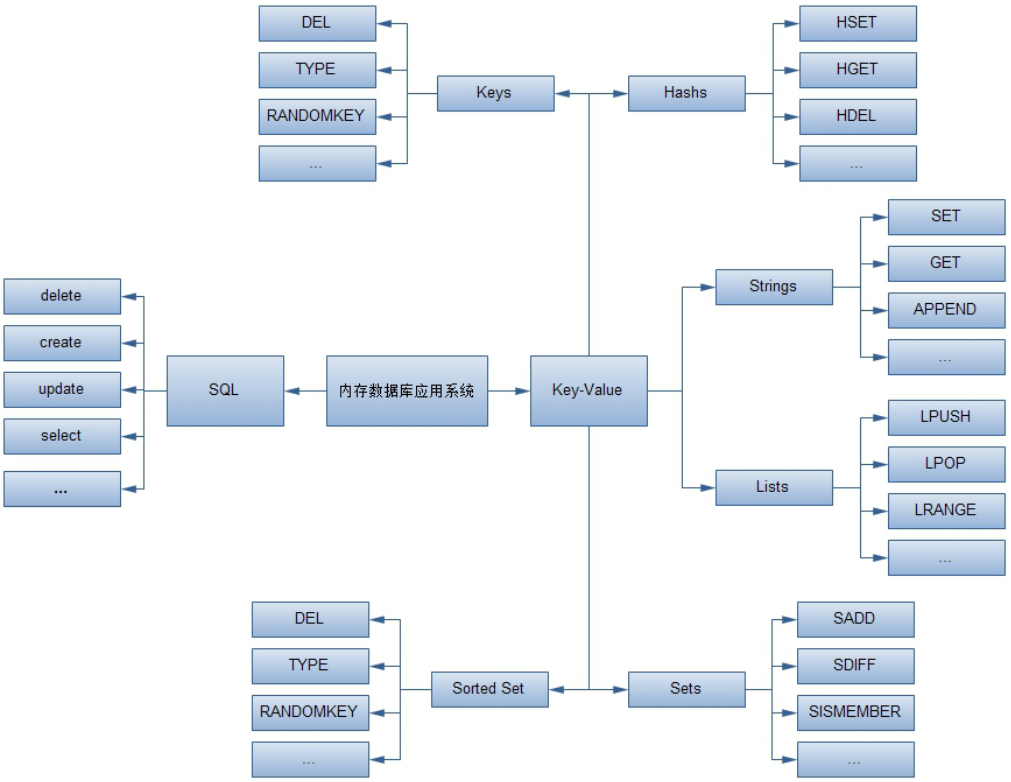

图 5.1: 系统功能结构图

系统根据用户输入的命令语句来选择具体的模块来执行。

5.2 主界面

本内存数据库系统界面采用 PyQT4 开发，包含了简单的执行命令，获得执行结果等功能，界面截图如下：

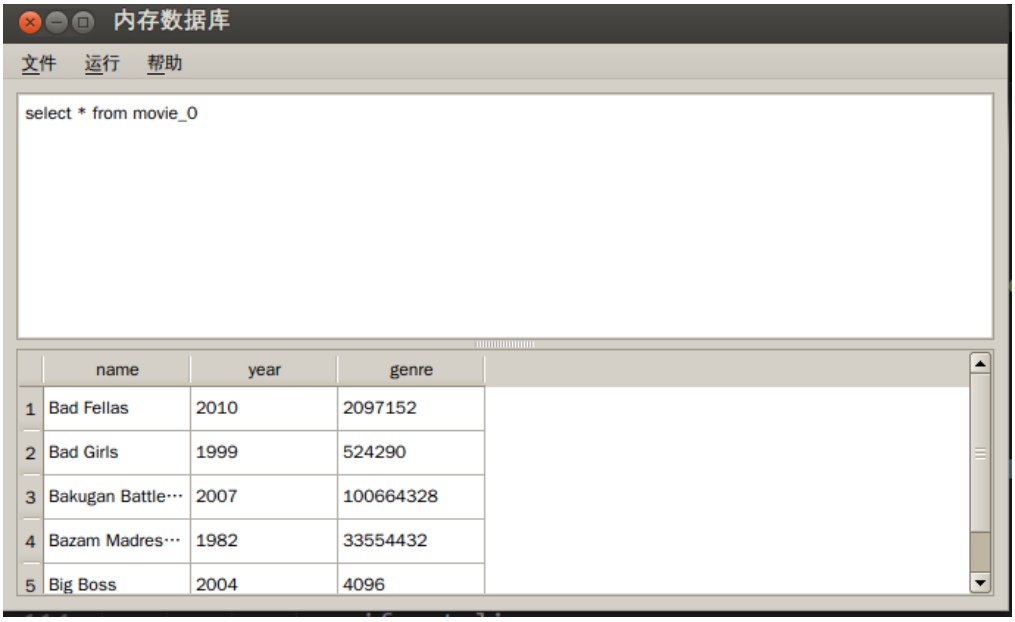

图 5.2: 系统界面截图

5.3 系统架构

在上文中，图 5.1 说明了主要的功能模块，包括 SQL 模块和 Redis 元命令模块，这两个模块的主要功能就是执行相关的命令，并返回相应的结果，但是两者的命令都是从同一数据源（GUI）获得的，必须通过“选择器”来确定一条命令是属于 SQL 模块还是 Redis 模块，具体设计流程如图 5.3 所示。

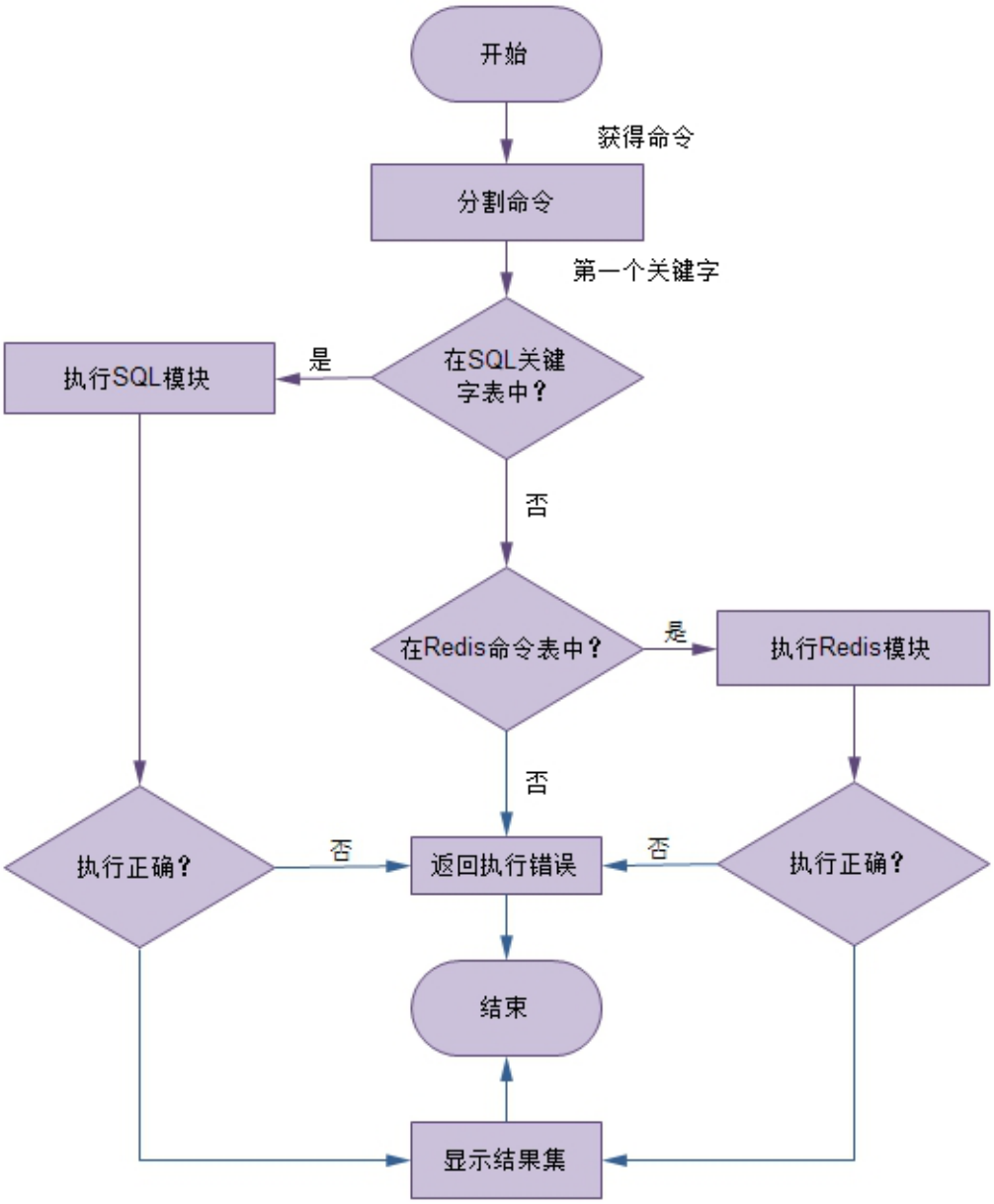

图 5.3: 选择处理模块的流程设计

第六章 性能对比

对于一个内存数据库系统而言，性能是非常重要的一个因素。为了测试本内存数据库系统的实际性能，将 MySQL（5.5）作为参照数据库，通过一定量的读写操作，来对比两者在性能上的差异，具体结果如表 6.1 所示：

表 6.1: 性能对比

	内存数据库（无索引）	MySQL	内存数据库（索引）
写的条数	3294	3294	3294
写的时间	0.851782083511 秒	2.95379686356 秒	0.939613819122 秒
写的速度	3867 条/秒	1115 条/秒	3505 条/秒
读的条数	10000	10000	10000
读的时间	10.441368103 秒	10.3408880234 秒	5.75849199295 秒
读的速度	957 条/秒	967 条/秒	1736 条/秒

从表 6.1 可以看出，性能的瓶颈在于是否进了索引。

第七章 总结

7.1 完成的工作

实现了基于 Redis 和 SQLite 开发的内存数据库系统的方案，并开发了功能较为简单的 GUI 界面，然后对其性能（和 MySQL 5. 5）进行了对比测试。

7.2 存在的问题及下一步工作

存在的问题主要是 GUI 的功能不全，缺少其他比较重要的功能，例如文件树等。接下去的工作主要是完善 GUI 中相关的功能并做测试，其次继续深入 Redis 的源代码，了解其其他模块的相关功能，例如对象处理机制，事务，频道订阅与发布，消息的发送等。

参考文献

[1] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[C]. ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007，14:205--220.

[2] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots[C]. Data Engineering (ICDE), 2011 IEEE 27th International Conference on, 2011:195--206.

[3] Bernet J. Dictionary compression for a scan-based, main-memory database system[D].[S.l.]: Master thesis, Eidgenössische Technische Hochschule, 2009-2010, 2010.

[4] Qureshi M K, Srinivasan V, Rivers J A. Scalable high performance main memory system using phase-change memory technology[C]. ACM SIGARCH Computer Architecture News, 2009，37:24--33.

[5] Liu M, Fei X D, Hu S, et al. Design and Implementation of Main Memory Database in ATC System[J]. Computer Engineering, 2010, 21:019.

[6] Zhao Y M, Zheng X F, Xu L Z. Design and implementation of index in main memory database system named SwiftMMDB[J]. Journal of Computer Applications, 2011, 9:024.

[7] Diaconu C, Freedman C S, Larson P A, et al. IN-MEMORY DATABASE SYSTEM[R], 2010. US Patent App. 12/756,185.

[8] Färber F, May N, Lehner W, et al. The SAP HANA database-an architecture overview[J]. IEEE Data Eng. Bull, 2012, 35(1).

[9] Ren K, Thomson A, Abadi D J. Lightweight locking for main memory database systems[C]. Proceedings of the 39th international conference on Very Large Data Bases, 2012:145--156.

[10] Niu X, Jin X, Han J, et al. A Cache-Sensitive Hash Indexing Structure for Main Memory Database[M]//Pervasive Computing and the Networked World.[S.l.]: Springer, 2013:400--404.

[11] Cattell R G, Russell C L. Systems and methods for a distributed in-memory database and distributed cache[R], 2012. EP Patent 1,840,766.

[12] Gurajada A P, Eluri A S, Nalawade V A, et al. Managing Data Storage as an In-Memory Database in a Database Management System[R], 2010. US Patent App. 12/726,063.

[13] Hoang C, Lahiri T, Neimat M A, et al. Distributed Consistent Grid of In-Memory Database Caches[R], 2009. US Patent App. 12/562,928.

[14] Han J, Song M, Song J. A Novel Solution of Distributed Memory NoSQL Database for Cloud Computing[C]. Computer and Information Science (ICIS), 2011 IEEE/ACIS 10th International Conference on, 2011:351--355.

[15] Plattner H. A common database approach for OLTP and OLAP using an in-memory column database[C]. Proceedings of the 35th SIGMOD international conference on Management of data, 2009:1--2.

[16] 王珊, 肖艳芹, 刘大为, 等. 内存数据库关键技术研究 [J]. 计算机应用, 2007, 27(10): 2353--2357.

[17] 陆宏. 一种高效内存数据库设计 [J]. 指挥信息系统与技术, 2012, 3(1):81--84.

[18] 郭超, 李坤, 王永炎, 等. 多核处理器环境下内存数据库索引性能分析 [J]. 计算机学报, 2010, 1:33.

[19] 袁培森, 皮德常. 用于内存数据库的 Hash 索引的设计与实现 [J]. 计算机工程, 2007, 33(18):69--71.

[20] 周游弋, 董道国, 金城. 高并发集群监控系统中内存数据库的设计与应用 [J]. 计算机应用与软件, 2011, 28(06):128--130.

[21] 张延松, 王占伟, 孙妍, 等. 内存数据库可控的 page-color 优化技术研究 [J]. 计算机研究与发展, 2011, 48(z2).

[22] antirez. Redis[R]. http://redis.io/.

[23] SQLite Home Page[R]. http://www.sqlite.org/.

[24] PyQT4 Home Page[R]. http://www.riverbankcomputing.co.uk/software/pyqt/intro.

致谢

由于自己本身的能力有限，虽然此次毕业设计已经基本完成，但是其中还有很多的不足和有待改进之处。

在这里特别感谢我的导师陈波老师在这整个过程中给予我的悉心的指导，以及一起努力奋斗的同学们的支持。感谢同组的杨道峰同学，同寝室的几位室友，感谢他们一直的支持和鼓励。感谢大学以来的老师，教会了我们很多重要的基础知识。最后感谢浙江工业大学四年来对我的大力培养。

附录

附录 **1** 毕业设计文献综述

附录 **2** 毕业设计开题报告

附录 **3** 毕业设计外文翻译（中文译文与外文原文）