基于内存映射文件的复杂对象快速读取方法

黄向平[1,2],彭明田[1,2],杨永凯[1,2]
(1. 中国民航信息网络股份有限公司,北京 101318;
2. 民航旅客服务智能化应用技术重点实验室,北京 101318)

摘 要:当前内存数据库(NoSQL)、嵌入式数据库技术在高并发高性能系统中得到了广泛的应用,但对于复杂对象数据的读取效率仍然低下,研究发现主要性能瓶颈有两个:一是内核态与用户态间的内存拷贝,拷贝消耗时间与复杂对象的数据量成线性增长;二是从数据库数据到运行时数据对象的格式转化操作,不但需要开辟新的内存空间存储运行时数据对象,而且还需要解析原始数据并拷贝至新对象之中。为此,提出了一种基于内存映射(memory mapping)文件的复杂对象共享读取方法。借助内存映射文件与自定义内存分配器,实现了结构复杂的 C++标准模板库容器对象跨进程无拷贝、无格式转化的共享,有效降低了数据读取延时。通过性能的分析比较表明,与 NoSQL 内存数据库、嵌入式数据库比,读取性能效率提升 10 倍以上。再加上底层技术成熟稳定,复用了标准模板库,具有开发成本低、可维护性强、实用性高等优点,因此,适用于高并发高性能的高可用后台服务系统。

关键词:内存数据库;嵌入式数据库;复杂对象共享读取;内存映射;内存分配器
中图分类号:TP301 **文献标识码**:A **文章编号**:1673-629X(2020)03-0082-06
doi:10.3969/j.issn.1673-629X.2020.03.016

Fast Reading Method of Complex Objects Based on Memory Mapping Files

HUANG Xiang-ping[1,2],PENG Ming-tian[1,2],YANG Yong-kai[1,2]
(1. TravelSky Technology Limited,Beijing 101318,China;
2. Key Laboratory of Intelligent Passenger Service of Civil Aviation,Beijing 101318,China)

Abstract:The current NoSQL database and embedded database technology have been widely used in high concurrent and high performance systems,but the efficiency of reading complex object data is still low. It is found that there are two main performance bottlenecks. One is the memory copy between kernel mode and user mode,which consumes time linearly with the amount of data of complex objects. The other is format conversion operation from database data to runtime data object,which not only needs to allocate new memory space to store runtime data objects,but also needs to parse the original data and copy them to new objects. Therefore,we propose a method of shared reading of complex objects based on memory mapping file. By means of memory mapping file and custom memory allocator,the sharing of container objects of C++ standard template library with complex structure without copy and format conversion across processes is realized to effectively reduce the data reading delay. The performance analysis and comparison show that the efficiency of data reading is more than 10 times higher than that of NoSQL memory database and embedded database. In addition,the bottom technology is mature and stable and the standard template library is reused,with the advantages of low development cost,strong maintainability and high practicability. Therefore,it is suitable for high concurrent and high performance high-availability back-end service system.

Key words:NoSQL database;embedded database;shared reading of complex objects;memory mapping;memory allocator

0 引 言

后台高并发高性能系统设计过程中,经常碰到数据访问写少读多的场景。比如民航航班运价搜索平台,航班运价数据更新不频繁,但搜索过程需读取并处理大量复杂数据。如何解决复杂数据读取效率的问题,是系统性能优化的关键问题之一。

收稿日期:2019-04-22 **修回日期**:2019-08-23 **网络出版时间**:2019-12-05
基金项目:国家核高基课题(2014ZX010450101);国家发改委 2014 年云计算工程项目(发改办高技[2014]1799 号)
作者简介:黄向平(1982-),男,硕士,研究方向为民航信息化技术;彭明田,教授级高级工程师,研究方向为民航信息化技术。
网络出版地址:http://kns.cnki.net/kcms/detail/61.1450.TP.20191205.1133.046.html

Linux 环境下高并发高效率的数据访问技术有 NoSQL 内存数据库[1]、嵌入式数据库[2]等。NoSQL(not only SQL)内存数据库典型代表有 redis[3],它是一种 key-value 类型的数据库产品,在大数据平台[4]、缓存系统[5]中得到了广泛的应用,相对于关系型数据库,具有服务响应快、并发访问高、易于扩展、易于开发等优点。嵌入式数据库是一种轻量级的、与应用进程在同一个地址空间的数据库,Berkeley DB 是目前应用较广泛、较稳定的技术选择,不但适用于嵌入式系统,也同样适用于高性能后台服务系统[6],优点是不需要进程间通信,读取效率很高。

对上述两种高性能数据库的研究发现,当数据结构复杂、读取频繁时仍会有性能问题。鉴于此,文中提出了基于内存映射文件的复杂对象快速读取方法。主要从三个方面对读取过程进行改进:(1)通过内存映射文件[7],省掉了内核态到用户态的内存拷贝操作,从而提高数据读取效率;(2)优化文件内数据格式,无需转化即可直接使用,从而提高数据格式转化的效率;(3)复用模板库和容器,避免重新编码实现一些常用数据结构,从而提高开发效率。

1　传统数据库技术

1.1　内存数据库

近年来,随着计算机技术的不断发展,特别是互联网应用的扩张普及,对于高并发环境下的数据库技术提出了高可用、高性能的要求。在一些诸如交通、电信、金融、电子商务之类的应用中,对于响应时间有非常高的要求。当前随着半导体技术的发展,摩尔定律依然有效,内存芯片的价格不断降低,服务器内存存储容量已经向 TB 级别迈进。相应的 64 位 CPU 和 Linux 操作系统的普及,操作系统寻址空间达到了 2 的 64 次方。内存资源紧张昂贵的历史一去不复返,如何充分、有效地利用内存资源成了软件设计的重点。因此,对于那些数据访问实时性要求很高的应用来说,开始考虑把整个数据库或者部分数据存储于内存之中,从而产生了内存数据库的概念。

一般定义上的内存数据库,是指数据常驻内存,在数据读写执行过程中没有内外存之间的数据 I/O,显然,它需要较多的物理内存,但也不一定把整个数据库置于内存当中。和传统的基于机械磁盘来存储数据的数据库系统相比,拥有更快的数据读写速度,因为内存数据库在指令执行过程当中没有磁盘读写操作。同时,内存 I/O 性能远高于磁盘,所以基于内存的数据库读写性能也远高于传统磁盘数据库。

典型的内存数据库如 memcached 和 redis,是一种 nosql 数据库,也是一种键值对数据库。通过在内存里维护一个统一的巨大的哈希表,能够存储各种格式的数据。一般的使用方法是减少数据库访问次数,以此提高系统服务响应时间,也可以把全量数据存储于内存数据库当中,完全替代传统关系型数据库。通常情况下,受制于物理内存空间大小,在数据容量达到指定值后,系统就基于 LRU(least recent used)算法[8]自动删除不适用的数据。可以说这类内存数据库是为缓存而设计的服务器,并没有过多考虑数据的永久性问题。不过 redis 加入了 virtual memory 改进方案,把很少使用的 value 保存到磁盘,而所有 key 保存于内存当中。因此 redis 可以应用于海量复杂数据的高性能读写场景。

此类内存数据库是基于 epoll 网络事件处理模型的 C/S 服务架构,支持高并发的网络连接,充分利用了网络 I/O 能力。如果把客户端与服务端部署于同一台服务器,则能避免网络延迟,只需进程间通信即可完成数据访问,提升响应速度。

1.2　嵌入式数据库

嵌入式数据库通常与嵌入式操作系统及具体的应用集成在一起,无需独立运行数据库服务实例,由程序直接调用响应的 API 即可读写数据,不需要对某种查询语言进行解析,也不需要生成解析计划。不仅仅适用于嵌入式操作系统,在一些高性能系统当中也直接连接嵌入式数据库,和应用程序运行在同一个地址空间,避免了进程间通信的消耗,因此具有较高的运行效率。

Berkeley DB 是目前使用较多的一种嵌入式数据库,作为一种轻量级嵌入式数据库,为许多编程语言提供了 API 接口,包括 c、c++、java、python 等,所有数据库相关操作都由 Berkeley DB 库函数负责统一完成。这样在多进程读写并行访问,或者在同一个进程内多线程竞争访问,都可以保证数据的一致性、完整性。底层的数据加锁、事务日志和存储管理等都在 Berkeley DB 库函数中实现,它们对于应用程序而言是完全透明的。

Berkeley DB 对于任何存入的 value 数据都是原样拷贝到数据文件当中,无论是文本数据还是二进制数据。它提供了四种存储数据的模式:btree、hash、queue 和 recno,在打开数据库的时候,需要指定其中一种存储模式。虽然有多种存储模式,但键值对是数据库的基础数据结构。用 API 函数访问数据库时,只需提供关键字就能够访问相应的数据。键值对结构在 Berkeley DB 中都是用一个名叫 DBT 的简单数据结构来表示的,它的作用主要是保存相应的内存地址和长度。

1.3 数据传输转化过程

数据的读取过程,就是数据从数据库传输到应用进程,并转化为运行时数据结构的过程。为了获得比传统内存数据库、嵌入式数据库更高读取性能的读取方法,在此分解研究数据传输转化过程,并依此提出优化目标对象。

进程地址空间分为内核态和用户态[9],其中内核态地址空间为所有进程与内核共享,用户态地址空间为每个进程独有。不管是跨网络、跨进程,还是读取文件的数据传输方式,都涉及到内核态数据拷贝至用户态的过程。具体过程如图 1 所示。

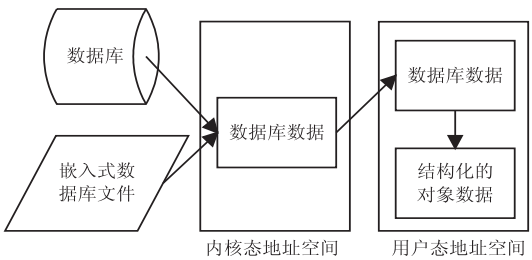

图 1 数据库数据传输转化过程

(1)内存数据库有独立运行的数据库引擎实例,数据依赖网络传输,数据包到来后网卡通过 DMA 将数据送入内核态地址空间,交由内核协议栈来处理,用户进程通过系统调用拷贝该数据包至用户态地址空间。如果内存数据库与用户进程部署于同一台服务器,则通过 Linux 回环接口(loopback)传输,它是一个虚拟网络接口,不与任何硬件相连,数据包的传输不经过网卡直接送入内核态地址空间,之后的协议处理传输与网络过程相同。

(2)嵌入式数据库把数据存储于数据文件当中。为了避免频繁重复的磁盘访问,Linux 操作系统在内核态地址空间内存里维护了一个页面缓存(page cache)[10],这样在从磁盘读取文件内容前查找这页面缓存是否已经装载了,只有没有命中缓存时才真正的从磁盘读取数据。页面缓存的价值有两个方面:第一,访问磁盘的速度远远低于访问内存的速度,因此,从内存访问数据比磁盘访问更快;第二,数据一旦被访问,就很可能在短期内再次被访问到。这种在短期内集中访问同一片数据的原理被称作临时局部原理(temporal locality)。临时局部原理能保证,如果在第一次访问数据时缓存它,那就极有可能在短期内再次被高速缓存命中。

数据格式转化一般由序列化和反序列化[11]过程组成。将程序运行过程中的对象转化成可以存储或运输的形式的过程,就是序列化(serialization)。反之,序列化结果转化成对象形式的过程,就是反序列化(de-serialization)。在传统的关系型数据库中,序列化和反序列化操作都隐含在数据库读写操作当中,对用户是透明的。内存数据库和嵌入式数据库的序列化和反序列化过程一般需要用户自定义[12]。由于各个系统的软硬件规格不同,一般需要编程语言无关的格式来序列化数据,常见的有 xml/json[13]等可见字符文本,以及效率更高的二进制数据流传输方式 protocol buffers[14]。当数据量很大,或者获取频次很高时,序列化和反序列化会大量消耗系统资源。

2 基于内存映射文件的快速读取方法

针对上述提出的内核态到用户态的内存拷贝,以及反序列化等带来的读取性能问题,文中针对性地提出改进方法。

2.1 减少内存拷贝

为了尽可能地减少拷贝操作,降低内存消耗,可采用系统调用 mmap,即内存映射技术。内存映射技术是 Linux 的一种内存管理方法,通过这种方法,在不占用额外的内存空间条件下,就能实现目标磁盘文件与进程用户态虚拟内存空间的对应关系,省去了文件 IO 等操作。

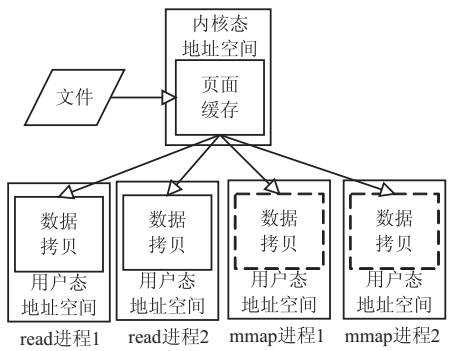

图 2 内核态与用户态的内存拷贝与映射

以读取性能较优的嵌入式数据库 Berkeley DB 为例,比较分析 mmap 的性能优势。Berkeley DB 以文件的形式存储数据,读取数据所采用的系统调用是 read。read 和 mmap 方式的差异比较如图 2 所示。磁盘文件装载于内核态的页面缓存区域,进程以 read 方式访问文件,在其用户态地址空间独立拷贝一份数据,每个 read 进程独享一份数据拷贝。也可以在多个 read 进程间共享一份用户态数据拷贝,其弊端是为了保证数据一致性需要额外的锁机制,读取频繁时多个进程竞争等待造成读取效率降低,由于此问题较复杂且与该文相关度不高,不再详述。相对而言,mmap 方式既简单又高效,不但没有拷贝操作,内存消耗也很低,整个物理内存只存在一份文件数据(在页面缓存区域)。应用程序不需要考虑文件映射内的数据是否已经装载,也不需要考虑如何在多个进程间共享,这些 Linux 内核都已处理,对用户透明。如果磁盘文件过大超过

了物理内存大小,Linux 内核会以 LRU(least recent used)算法把冷数据置换出来,从而释放内存来存储新数据,最终在页面缓存中的数据是经常被访问的热数据。

2.2　优化数据格式

复杂对象不但数据量大,且数据间关系复杂,体现在数据结构上就是指针引用较多。当把复杂对象序列化于文件时,指针就失效了,必须将相应的指针转化为指针指向的目标数据块在文件中的偏移量。如图 3 所示的哈希表,哈希桶内的指针指向键值对结构,当序列化于文件时,该指针需修改为键值对结构在文件中的偏移量。当用户进程读取该哈希表时,得到的文件偏移量加上文件映射首地址,即是目标键值对结构的内存地址。

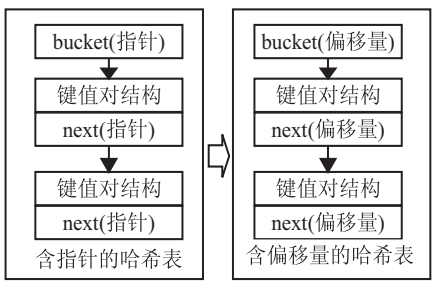

图 3　复杂对象的指针转化

在 64 位地址空间[15]当中,有大量的空闲空间可供程序使用。因此,经过如下步骤(见图 4),则可在文件中直接存储指针,避免文件偏移量的转化:

(1)序列化进程在指定虚拟内存区域内分配对象内存,直至整个复杂对象存储于该指定虚拟内存区域。

(2)把该内存区域内的二进制数据 dump 到内存镜像文件。

(3)由反序列化进程映射该内存镜像文件,映射地址为之前指定的虚拟内存区域。

(4)复杂对象不需要任何格式转化,即可被反序列化进程使用。

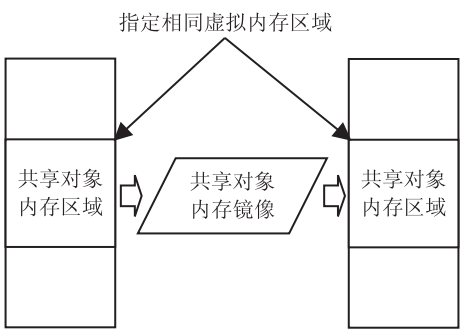

图 4　复杂对象共享过程

2.3　复用模板库和容器

如 2.2 节所述,优化数据格式的重点在于如何把一个复杂对象序列化于指定虚拟内存区域。假设某个复杂对象为多重 key-value 结构(value 部分又是一个 key-value 结构),这就要求开发人员开发一个特定的哈希表数据结构。当复杂对象使用的数据结构更多时,相应的特定数据结构开发工作也更多,开发效率低下。由此,引入了可复用组件 C++标准模板库[16]。

C++标准模板库 STL(standard template library)是一套标准化组件,由六大组件组成:

(1) container:泛型容器,基本数据结构有数组 vector,链表 list,二叉树 map,哈希表 unordered_map 等。

(2) algorithms:泛型算法,常见的排序 sort,搜索 search 等。

(3) iterator:迭代器,借此 container 和 algorithms 解耦,两者可以各自独立发展互不关联。

(4) function object:是一种模板函数对象,常见的有 map 容器中的比较对象 std::less。

(5) adaptor:适配器,修饰 container 或 function object 接口的一种组件。比如栈 stack 作为容器适配器,可以实现为基础容器类型(vector,list)的适配。

(6) allocator:内存分配器,STL 提供了默认内存分配器。

泛型容器提供了一些常用的数据结构,可以嵌套使用,比如借助 unordered_map 可实现多重 key-value 结构。内存分配器负责数据对象的内存分配,默认内存分配器是基于 gnu libc 的内存分配算法[17],内存分配分散于进程堆空间之中,无法区分共享数据与私有数据,两类数据交错存放。通过自定义内存分配器,使得需要共享的大小不一的各种 C++对象从指定的地址空间中分配,其他私有对象仍然从进程堆空间分配。最终,序列化进程把指定内存区域内的连续内存块数据 dump 出来保存,即为可共享的数据文件。

3　系统关键实现与对比验证

该方案的实现关键点在于如何把对象序列化于数据文件当中。数据的读取操作相对简单,把文件映射获得的内存区域首地址指针强制转化为 C++对象指针类型即可。

3.1　创建空洞文件

初始阶段创建空文件用以存放目标复杂对象。由于对象长度大小事先无法确定,因此一般做法是创建一个较小的空文件,随着复杂对象的扩张而不断延长文件长度。这个办法的缺点是实现相对复杂,延长操作过多的话会影响写入性能。

借助空洞文件,可以解决该问题。在 Linux 文件操作中,文件位移量是可以大于文件的当前长度的,在这种情况下,对该文件的下一次写将延长该文件,并在

文件中构成一个空洞,即为空洞文件。位于文件中但没有写过的字节都被预设为 0。空洞是否占用硬盘空间由文件系统决定,在一般文件系统中不占用硬盘空间,意味着不需要大量的写 0 操作,创建过程耗时极短。

在写 C++对象数据文件前,先创建一个空洞文件,文件长度为预估数据量的若干倍,比如 10 倍。无需担心浪费磁盘空间,因为最终序列化完毕时,可以根据实际数据量缩减文件长度。

3.2 基于内存映射技术的内存分配器

内存分配器是处理应用对于内存的分配和释放要求的,默认内存分配器所分配内存处于进程堆空间。该方案自定义的基于内存映射技术的内存分配器,可分配内存来至磁盘文件映射出的连续内存空间。其主要接口和成员变量如下:

```
class MemoryMapAllocator
{
public:
MemoryMapAllocator(const char * filePath, unsigned long maxSize, void * address);
~MemoryMapAllocator();
void * allocate(unsigned long size);
void release(const void * ptr);
private:
void * m_currentPointer;
};
```

构造函数的作用是根据数据文件目录位置,以及预估的数据最大大小,创建空洞文件,并映射入指定的用户态地址空间(函数参数 address),成员变量 m_currentPointer 初始化为映射空间的首地址。

析构函数的主要功能是以 msync 系统函数同步数据写入磁盘,再以 munmap 系统函数撤销内存映射,即数据持久化过程。

成员函数 allocate 是内存分配操作方法,首先 m_currentPointer 指针自增函数输入参数 size 字符数,然后函数返回自增前 m_currentPointer 的值。一般为了提升内存访问效率,对输入参数 size 做对齐工作,向上取整 8 的倍数(64 位操作系统环境)。

成员函数 release 是释放内存操作方法,在方案中并不必要。所有 C++对象都分配于映射地址空间,所有数据写入之后,撤销文件映射并把内存数据同步于磁盘文件,所有对象内存自然释放,因此 release 函数内部实现可以为空操作。一种更复杂的实现是重复利用释放的内存,借助精细的内存管理机制(如 slab 算法[18]),记录所有释放的内存,内存分配算法优先在其中寻找符合大小要求的闲置空间。实际应用当中,此类释放内存数量有限,重复利用价值不高,且代码复杂度高,可维护性差,不建议使用。

3.3 多线程支持

在创建数据文件时,如果数据量很大,可借助多线程并发处理能力来加速处理。为此内存分配器需设计成线程安全。

最简单的方法是在类 MemoryMapAllocator 中新增锁对象,每次调用 allocate 方法时加锁,函数返回时释放锁,保证只有一个线程在修改 m_currentPointer 指针,其余试图申请内存的线程处睡眠等待状态。当线程争抢锁剧烈时,频繁的线程切换就会成为性能瓶颈。

为了解决锁带来的弊端,可采用原子操作[19]。所谓的原子操作是指不会被线程调度机制打断的操作,一旦操作开始,就一直运行到结束,中间不会被其他线程操作打断。其实现原理是依靠硬件的支持,在 X86 平台上,CPU 提供了在指令执行期间对总线加锁的手段。避免了线程切换,提升处理速度。

GCC 编译器当前已经支持了原子操作,可以对 1、2、4、8 字节的数值或指针类型进行原子的加、减、与、或、异等操作。对于 m_currentPointer 的加法操作,可以靠原子操作函数__sync_fetch_and_add(type * ptr, type value)完成,其实现细节是将 value 加到 * ptr,结果更新到 * ptr,并返回操作之前 * ptr 的值。

3.4 读取效率对比

为了对改进复杂对象读取性能做进一步的分析,对传统内存数据库 redis、嵌入式数据库 berkeley DB 和基于内存映射文件的复杂对象读取方法进行了性能对比测试。测试目的是在读取相同数据量的前提下,比较各个方案的响应时间值。软件配置为 Linux 内核 2.6.32,redis libhiredis 0.13,berkeley DB libdb_cxx 4.7;硬件配置 CPU 频率 2.60 GHz,内存 256 G。

在以上测试环境中,创建一个进程顺序进行三次不同方案的数据读取操作,保证了软硬件资源一致。其中内存数据库 redis 服务端与客户端部署于同一台服务器,避免网络延时带来的误差。内存映射文件数据库采用了 C++标准模板库容器 unordered_map。

三个数据库存储相同 key-value 键值对 1 024 个,每个 key 平均 2.9 Bytes,每个 value 平均 2 986 Bytes,读取程序设计为遍历读取这 1 024 个 key 共 600 遍,即 614 400 次查询操作,实验结果见表 1。

表 1 三种数据库方案读取性能测试结果

数据库方案	耗时/s	相对比例
内存数据库 redis	12.59	153.2 倍
嵌入式数据库 berkeley DB	1.09	13.2 倍
内存映射文件数据库	0.08	1 倍

从结果上看,文中提出方法的读取性能相对于传统方案有非常显著的提升,与性能较优的嵌入式数据库相比其读取响应时间提升 10 倍以上。而且如前所述,内存数据库与嵌入式数据库的内存消耗量与读取进程个数成线性增长,因为每个进程都要拷贝存储数据,而内存映射文件数据库不需要拷贝,其内存消耗量与读取进程个数基本不相关。

该方案的另一个优点在于支持各种复杂对象,而不单限于 key-value 结构。由于内存数据库、嵌入式数据库支持的结构类型有限,故只比对了 key-value 结构。如果把一个复杂对象分拆成多个 key-value 键值对存储于普通数据库中,其性能更无法与面向对象的内存映射文件数据库相提并论。原因是前者需要多次索引搜索才能组装成完整的复杂对象,而后者没有这个过程,所有的对象内数据关系都已经通过指针关联。

4　结束语

传统数据库在高并发应用场景中,数据拷贝操作频繁,数据量较大时容易成为性能瓶颈。当数据对象结构复杂时,不但数据库表结构、索引设计复杂,而且序列化与反序列化过程耗时严重。该方案通过内存映射文件、自定义内存分配器、标准模板库容器等技术,有效解决了复杂对象读取性能问题。读取过程没有进程间通信,不需要数据拷贝,不需要反序列化过程即可直接引用对象,与传统数据库读取效率对比的实验表明性能可提升 10 倍以上。同时,还具有底层技术成熟稳定、开发代码量少、易于维护等优点。内存映射文件技术得到了广泛的支持,Linux 等类 Unix 操作系统中都提供了相关系统调用。标准模板库提供了诸多常用数据结构,大大降低了复杂对象结构设计与开发成本。借由此类成熟技术,开发过程周期短,易于实现,后期维护更新成本低,在实际应用中运行稳定。在数据库灾备设计方面,由于数据存储于文件,易于传输,如果数据丢失,可以很快从灾备数据文件中恢复。因此,该方案实用性高,适用于数据访问方式写少读多的高并发高性能搜索、计算等后台服务系统,在工程实践中有重要的典型意义。

参考文献:

[1] CHANDRA D G. BASE analysis of NoSQL database[J]. Future Generation Computer Systems,2015,52:13-21.

[2] 万玛宁,关　永,韩相军. 嵌入式数据库典型技术 SQLite 和 Berkeley DB 的研究[J]. 微计算机信息,2006,22(2):91-93.

[3] GAO X,FANG X. High-performance distributed cache architecture based on Redis[J]. Lecture Notes in Electrical Engineering,2014,270:105-111.

[4] 吕冬雪. 基于大数据环境的 NoSQL 技术分析[J]. 电子设计工程,2016,24(14):33-36.

[5] 曾超宇,李金香. Redis 在高速缓存系统中的应用[J]. 微型机与应用,2013,32(12):11-13.

[6] 刘启洪. 基于嵌入式数据库的海量存储技术分析研究[J]. 数字技术与应用,2013(4):80-81.

[7] 王祥雒,李　毅. Linux 中基于 mmap()的共享存储实现研究[J]. 计算机应用,2006,26(S2):307-309.

[8] 张震波,杨鹤标,马振华. 基于 LRU 算法的 Web 系统缓存机制[J]. 计算机工程,2006,32(19):68-70.

[9] 王兆文,蒋泽军,陈进朝. 一种提高 Linux 内存管理实时性的设计方案[J]. 计算机工程,2014,40(9):291-294.

[10] 张学亮,左小翠. Linux 页面缓存机制分析及其对磁盘 I/O 性能影响[J]. 计算机与现代化,2010(2):183-187.

[11] 赵　冬. 序列化技术的综述和展望[J]. 电脑与电信,2013(9):39-42.

[12] 孙杜靖,李玲娟. 面向 Redis 的数据序列化算法研究[J]. 计算机技术与发展,2017,27(5):77-81.

[13] 张　涛,黄　强,毛磊雅,等. 一个基于 JSON 的对象序列化算法[J]. 计算机工程与应用,2007,43(15):98-100.

[14] FENG J H,LI J H. Google protocol buffers research and application in online game[C]//IEEE conference anthology. China:IEEE,2013:1-4.

[15] 宿继奎,吴亚栋,吕必俊. 32 位到 64 位的移植[J]. 计算机应用与软件,2007,24(3):174-176.

[16] 葛建芳. C++标准模板库与代码重用[J]. 南通大学学报:自然科学版,2006,5(2):71-74.

[17] 张　会. C++语言内存分配研究[J]. 计算机时代,2014(5):44-46.

[18] 赵鲲鹏,苏葆光. Linux 内存管理中的 Slab 分配机制[J]. 现代计算机,2006(5):89-91.

[19] 潘圆圆,李德华. C++中的原子操作及其使用[J]. 计算机与数字工程,2013,41(11):1853-1855.