FastDB学习日志

# 快速开始

1、创建类型和表描述符

2、在实现模块中通过REGISTER宏注册表描述符

3、如果重新定义默认的Fastdb错误处理程序，应该从dbDatabase派生；

4、打开数据库dbDatabase::Open()，检查返回码确定数据库是否正确打开；

5、如果使用多线程访问同一个数据库，通过dbDatabase::attach方法将每个线程附加到数据库。线程终止前，通过dbDatabase::detach将线程从数据库中分离出来；

6、如果使用引用遍历数据库对象，需要不使用引用找到一些类型的根对象。最佳对象为表中的第一个记录；

7、使用dbQuery和dbCursor访问数据库数据；多线程时，每个线程都应该有自己的查询和游标对象的实例；

查询对象用于缓存编译查询，因此延长查询变量的生命周期是明智的(例如静态变量)。

8、数据库的操作：insert不使用游标，通过全局重载模板函数插入。select、update、remove通过游标对象执行。

9、当执行第一次选择，插入和删除时，事务将自动启动。事务由dbDatabase::commit()提交，由dbDatabase::rollback()终止。

10、退出应用程序之前，应主动关闭数据库。dbDatabase::close()会自动提交最后一个事务，如果不想提交这个事务，应明确指定dbDatabase::rollback();

11、fastdb应用程序的通用模型为：

// 头文件

#include "fastdb.h"

extern dbDatabase db;

class MyTable {

char const\* someField;

......

public:

TYPE\_DESCRIPTOR((FIELD(someField)));

};

// 源文件

REGISTER(MyTable);

int main() {

if(db.open("mydatabase")) {

dbCursor<MyTable> cursor;

dbQuery q;

char value[bufSize];

q = "someField=",value;

gets(value);

if(cursor.select(q) > 0) {

do {

printf("%s\n", cursor->someField);

} while(cursor.next());

}

db.close();

return EXIT\_SUCCESS;

}

else {

return EXIT\_FAILURE;

}

}

# 查询优化

FastDB提供的几种查询优化方法：索引、逆引用、并行查询；

## 索引

FastDB提供了两种类型的索引：扩展的哈希表和T-Tree。哈希表对指定了关键字值的记录的访问是最快的。T-Tree提供了对数算法复杂度的搜索、插入和删除操作。

FastDB由程序员提供的表达式来决定什么时候使用索引及使用哪一种索引。下面的规则说明了FastDB应用索引的算法：

• 编译好的条件表达式总是从左到右检查.

• 如果最终(topmost)表达式是AND，则尝试在表达式的左半部分使用索引，右半部分作为过滤(filter).

• 如果最终表达式是OR，则如果左半可以使用索引则使用，然后测试右半使用索引的可能性.

• 此外，当下列条件满足时，则索引适用于表达式

1. 最终表达式是关系操作 (= < > <= >= between like)

2. 操作数的类型是布尔型，数值型，字符串和引用

3. 表达式的右操作数是文本常量或者C++变量，并且

4. 左操作数是记录的索引字段

5. 索引与关系操作兼容

在下面的情况下使用哈希表索引：

• 使用等于=的比较.

• 使用Between 操作并且两个端点操作数的值相等.

• 使用Like 操作并且模式串不包含特别字符(’%’或者’\_’)并且没有转义字符(在escape 部分指定).

哈希表不适合并且下面条件满足时，使用T-Tree索引：

• 比较运算( = < > <= >= between)被使用.

• 使用 Like 运算并且模式串包含非空前缀(也就是说模式的第一个字符不是’%’或者’\_’).

## 逆引用

FastDB在插入/删除/更新记录时及查询优化时使用逆引用的信息。记录之间的关系可以为：一对一、一对多、多对一、多对多；

当一个声明了关系的记录被插入表中，所有表中的与该记录关联的逆引用，都被更新至指向这个记录。当更新了一个记录并且一个指明了该记录的关系的字段发生变化，则该逆引用自动重构，删除那些不再与该被更新的记录关联的记录对该记录的引用，并且设置包含在该关系中的新记录的的逆引用至该更新的记录。当一个记录被删除，所有逆引用字段中指向其的引用都被删除。

## 并行查询

并行查询由FastDB自动完成，仅用于表的顺序扫描。

并行查询有两个参数控制：生成的线程数和并行搜索阈值。第一个参数在dbDatabase的构造函数中指定，或由dbDatabase::setConcurrency方法设置。该参数的零值要求FastDB自动检测系统中的在线CPU数量，并产生相同的线程数。默认情况下，线程数设置为1，因此不会执行并行查询。第二个参数指定表中的最小记录数，以便并行查询能够提高查询性能。该参数是dbDatabase类的静态组件，可以在任何时刻由应用程序修改。

下面情况下不能执行并行查询：

1. 索引被用于查询执行.

2. 表中的记录数小于dbDatabase :: dbParallelScanThreshold.

3. 为游标设置选择限值.

4. 该查询包括从部分开始.

# 容错支持 主从复制 自动同步

从2.49 版本开始fastdb 提供了可选的容错支持。可以启动一个主要的(活动的)和几个备用的结点，所有在主要结点发生的变化同时被复制到备用结点上。如果主结点崩溃，其中一个备用结点将变为活动的并成为主结点。一旦一个崩溃的结点重新启动，它要进行恢复，与主结点的状态同步，然后作为备用结点投入使用。结点通过套接字连接并规定放置在不同的计算机上。通信被假定为是可靠的。

要使用容错支持，应该使用REPLICATION\_SUPPORT 可选项来重新编译fastdb.在makefile开始把FAULT\_TOLERANT 变量设置为1 来把它打开。应该使用dbReplicatedDatabase 来代替dbDatabase。在open 方法的参数中，除了数据库名和文件名之外，应当指定这个结点的标志符(从0 到N-1 的整数)，包含所有结点地址(主机：端口)的数组以及结点数(N).然后就可以在N 个结点的每一个启动程序。一旦所有的实例都启动，ID=0 的结点成为活动的(主结点)。在这个实例中open 方法返回true.其他结点在open 方法堵塞。如果主结点崩溃，其中一个备用结点被激活(open 方法返回true),然后这个实例继续执行。如果崩溃的实例重新启动，它将尝试连接所有服务器，恢复其状态然后作为备用结点，等待其代替崩溃的主结点的机会。如果主结点正常终止，所有备用结点的close 方法返回false.

在容错模式下fastdb保留两个文件：一个包含数据库本身，另一个则是页更新计数器。带有页更新计数器的文件用于增量恢复。当崩溃结点重启动时，它将向主结点发送页计数器，并只接受这段时间在主结点发生变化的页(其时间戳大于被恢复的结点所发送的页)。

在复制模式中(在主结点)应用程序在事务提交期间并不阻塞直到所有的变化被刷新到硬盘。已改变的页由独立的线程异步的刷新到磁盘上。这样带来了显著的性能提升。但如果所有的结点都崩溃了，数据库就可能处于不一致状态。也可以指定向硬盘刷新数据的时间延迟：延迟越大，磁盘IO 开销越小。但在崩溃的情况下，需要从主结点发送更多的数据以进行恢复。如果所有节点都崩溃，系统管理员应选择具有最新版本的数据库的节点(对不起，它不能自动完成)，并在该节点启动应用程序，但不在其他节点。超时(5秒)到期后，会报告无法连接到其他节点。当与所有指定节点的连接失败时，程序将执行本地恢复并作为新的活动节点启动。然后，你可以启动从主动节点执行恢复的其他节点。

可以在无盘模式中使用容错模式(DISKLESS\_CONFIGURATION 构建选项)。在这种情况下，没有数据保存在磁盘上(没有数据库文件，也没有页更新计数器).假定至少有一个结点总是活动的。只要有一个在线结点数据就不会丢失。当崩溃结点恢复时，主结点向其发送完整的数据库快照(增量恢复是无法实现的因为崩溃结点的状态已经丢失)。由于这种模式没有磁盘操作，操作性能是非常高的并且只受限于网络吞吐量。

当复制结点启动后就开始在指定的时限内尝试连接所有其他的结点。如果在这个时间内无法建立连接，则该结点被假定为自主启动的并作为普通(非复制)数据库开始工作。如果结点与其它结点建立了连接，则具有最小ID的节点被选为复制主结点。所有的其他结点被切换到旁置模式并等待来自主结点的复制请求。如果主结点和从结点的数据库的内容不一致(使用页计数器数组来决定)，则主结点进行旁置结点的恢复，向其发送最近的页面。如果主结点崩溃，则旁置结点选择一个新的主结点(最小ID 的节点)。所有的旁置结点都在open方法堵塞直到下面的情况之一发生：

1. 主结点崩溃并且该节点被选为新的主结点。在这种情况下open方法返回true.

2. 主结点正常关闭数据库。在这种情况下所有复制结点的open方法返回false.

可以从其他应用程序中对复制数据库进行只读的访问。在这种情况，复制的结点必须通过dbReplicatedDatabase(dbDatabase::dbConcurrentUpdate)构造调用来创建。其他应用程序可以用dbDatabase(dbDatabase::dbConcurrentReadMode)实例来访问同一数据库

并非所有应用都需要容错。许多应用使用复制只是为了提高可测量性，在许多结点间分担负载。对于这些应用，fastdb提供了简化复制模型。在这种情况下，有两种结点：读者和写者。任何一个写者结点都可以作为复制主结点。而读者结点只能从主结点接收复制的数据而不能自己更新数据库。与上面所述的复制模型的最主要区别是读者永远不能变成主结点并且这个结点的open方法一旦与主结点建立了连接就马上归还控制权。来自主结点的更新通过单独的线程接收。读者结点要用dbReplicatedDatabase(dbDatabase::dbConcurrentRead)构造器来创建。必须使用预主结点同样的数据库模式(类)。当主结点关闭连接时来自读者结点的数据库连接并不自动关闭，其仍然保持打开并且应用仍然可以以只读模式访问数据库。一旦主结点重启，就会与所有的旁置结点建立连接并继续向它们发送更新。如果没有读者结点，则复制模型就等同于前面所述的容错模型，如果只有一个写者结点和一个或多个读者结点，这就是经典的主从复制。

# 延迟事务和在线备份调度

Fastdb 支持ACID 事务。也就是说当数据库得到事务已经提交的报告后，可以保证该数据库在系统出错时(除了硬盘上的数据库镜像损坏外)能够恢复。在标准配置(例如没有非易变RAM)和通用操作系统中(windows，unix….)提供这种特性的唯一方法是对硬盘进行同步写。在这里“同步”意味着操作系统直到数据被真正写到硬盘上之后才会把控制权交回应用程序。不幸的是同步写是非常耗时的操作—平均磁盘访问时间是10ms，因此每秒很难达到处理100个事务的性能。

但是在很多情况下，丢失最后几秒的变化是可以接受的(但是要与数据库保持一致性)。依照这个假定，数据库性能可以显著得到提高，fastdb 为这样的应用程序提供了“延迟事务提交模式”。当提交事务延迟非零时，数据库并不马上执行提交操作，而是根据一个指定的超时时间延迟操作。当超时时间过期，事务正常提交，这保证了在系统崩溃时只有在指定的超时时间内的变化才被丢失。

如果以延迟事务初始化的线程在被延迟的事务提交之前启动了新的事务，则延迟提交操作被忽略。因此fastdb 可以把一个客户端执行的许多继起(subsequent)的事务组成一个单一的大事务。这样就极大地提高了性能，因为其减少了同步写的次数和创建的映像页的个数。(参看事务一节)。

如果其他客户端试图在延迟提交超时时间过期前启动事务，则fastdb 强制进行延迟提交然后释放资源。因此同步不受延迟提交的影响。

延迟提交缺省是关闭的(超时时间为0)。你可以指定提交延迟参数作为dbDatabase::open方法的第二个可选参数。在SubSQL 工具中也可以通过设置FASTDB\_COMMIT\_DELAY 环境变量(秒)来指定事务提交延迟的值。

fastdb 使用的事务提交模式保证了在软硬件出现故障时只要磁盘上的数据库没有损坏(写到盘上的数据可以正确的读出来)的恢复。如果由于某些原因数据库文件损坏了，则恢复的唯一途径是使用备份(但愿在不久之前做过这样的操作)。

当数据库离线时可以通过拷贝数据库文件来备份。dbDatabase 类提供了backup 方法来进行在线备份而不需要停止数据库。程序员在任何时候都可以调用这个方法。不过更进一步，fastdb 提供了备份调度可以自动进行备份。唯一需要的是—备份文件名和备份之间的时间间隔。

dbDatabase::scheduleBackup(char const\* fileName, time\_t period)方法派生出单独的线程在指定的时间内(秒)向指定的位置进行备份。如果filename以"?"字符结尾，则备份初始化的时间被附加到文件名的末尾来产生唯一的文件名。在这种情况下所有的备份文件保存在磁盘上(把太老的备份文件移除或者把它们移到别的介质上是管理员的责任)。否则备份被写入到以fileName+".new"命名的文件中，备份完成后旧备份文件被删除新文件被重命名为filename。在后一种情况下，fastdb 也将检查旧备份文件(如果有的话)的创建日期然后按照这样的方式来调整等待时间，就是备份之间的时间差要等于指定的间隔(因此如果数据库服务器每天只启动8个小时，而备份间隔为24 小时，则备份将每天都进行，这与唯一文件名模式不同)。

可以通过设置FASTDB\_BACKUP\_NAME 环境变量在SubSQL 工具中进行备份调度。如果指定了FASTDB\_BACKUP\_NAME则间隔值依此取定，否则设置为每天。从备份中恢复只需要用一些备份文件替代损坏的数据库文件。

# C++接口

## 表Table

fastdb 中的数据保存在表中，这些表对应于c++类，其中表记录对应于类实例。下面的c++数据类型可以作为fastdb 记录的原子组件：

|  |  |
| --- | --- |
| **Type** | **Description** |
| bool | boolean type (true,false) |
| int1 | one byte signed integer (-128..127) |
| int2 | two bytes signed integer (-32768..32767) |
| int4 | four bytes signed integer (-2147483648..2147483647) |
| int8 | eight bytes signed integer (-2\*\*63..2\*\*63-1) |
| real4 | four bytes ANSI floating point type |
| real8 | eight bytes ANSI double precision floating point type |
| char const\* | zero terminated string |
| dbReference<T> | reference to class T |
| dbArray<T> | dynamic array of elements of type T |

除了上表定义的数据类型外，fastdb 记录还可以包括这些元组的嵌套结构。fastdb不支持无符号数据类型以简化查询语言，清除由于符号数/无符号数比较产生的错误，减少数据库引擎的大小。

不幸的是c++没有提供在运行时获得一个类的元信息（metainformation）的方法（RTTI 并不被所有编译器支持，并且也不能提供足够的信息）。因此程序员必须明确枚举包含在数据库表中的类字段（这也使得在类和表之间的映像更为灵活）。fastdb 提供了一个宏和相关的类的集合来使得这个映像尽可能的灵活。

### TYPE\_DESCRIPTOR

每一个要在数据库中使用的c++类或者结构，都包含一个特别的方法来描述其字段。宏TYPE\_DESCRIPTOR(field\_list)构成了这个方法。这个宏的用括号括起来的单一参数是一个类字段描述符的列表。如果要为这个类定义一些方法并使之可以用于对应的数据库，则用宏CLASS\_DESCRIPTOR(name, field\_list)来代替TYPE\_DESCRIPTOR。需要类名来取得成员函数的引用。

下面的宏可以用来构造字段描述符：

#### FIELD(name)

指定名字的非索引字段.

#### KEY(name, index\_type)

索引字段。index\_type 必须是HASHED 和INDEXED 标志的组合。当指定HASHED标志的时候，fastdb 将为用这个字段作为关键字的表创建一个hash 表。当指定INDEXED标志时，fastdb将为使用这个字段作为关键字的表创建一个T\_tree（一种特殊的索引）.

#### UDT(name, index\_type, comparator)

用户自定义原始二进制类型。数据库把这种类型作为指定大小的字节序列处理。这个字段可以用来查询（比较同一类型的查询参数），可以通过order 子句来索引和使用。通过程序员提供的comparator 函数来进行比较操作。比较函数接受3 个参数：两个指向待比较的原始二进制对象的指针及其大小。index\_type 的语义与KEY宏中的一致。

#### RAWKEY(name, index)

带有预定义比较算子的原始二进制类型。这个宏只是一个把memcmp 作为比较算子的UDT宏的特例。

#### RAWFIELD(name)

另一个UDT宏的特例，使用memcmp作为预定义比较算子，并且没有索引。

#### SUPERCLASS(name)

指定当前类的基类(父亲)的信息。

#### RELATION(reference, inverse\_reference)

指定类（表）之间的一对一、一对多或者多对多的关系。reference 和inverse\_reference 字段都必须是引用或者引用数组类型。inverse\_reference字段是一个包含了指向当前表的逆引用的引用表。逆引用自动由fastdb更新并用于查询优化。

#### OWNER(reference, inverse\_reference)

指定类(表)之间的一对多或者多对多的owner-member关系。当owner记录被删除时所有引用的member 记录也会被删除(层叠式删除)。如果member记录要引用owner就必须通过RELATION宏声明。

#### METHOD(name)

为类指定一个方法。该方法必须是无参的实例成员函数，返回bool 值、数值、引用或者字符串类型。方法必须在类的所有属性之后指定。

尽管只有原子字段可以被索引，但可以为一个结构指定一个索引类型。只有当该索引类型在该结构的索引mask中指定时才会为该结构的成员创建。这就允许程序员可以根据该结构在记录中的角色来设置或者取消索引。

下面的例子说明了头文件中类型描述符的创建过程：

class dbDateTime {

int4 stamp;

public:

int year() {

return localtime((time\_t\*)&stamp)->tm\_year + 1900;

}

...

CLASS\_DESCRIPTOR(dbDateTime,

(KEY(stamp,INDEXED|HASHED),

METHOD(year), METHOD(month), METHOD(day),

METHOD(dayOfYear), METHOD(dayOfWeek),

METHOD(hour), METHOD(minute), METHOD(second)));

};

class Detail {

public:

char const\* name;

char const\* material;

char const\* color;

real4 weight;

dbArray< dbReference<Contract> > contracts;

TYPE\_DESCRIPTOR((KEY(name, INDEXED|HASHED),

KEY(material, HASHED),

KEY(color, HASHED),

KEY(weight, INDEXED),

RELATION(contracts, detail)));

};

class Contract {

public:

dbDateTime delivery;

int4 quantity;

int8 price;

dbReference<Detail> detail;

dbReference<Supplier> supplier;

TYPE\_DESCRIPTOR((KEY(delivery, HASHED|INDEXED),

KEY(quantity, INDEXED),

KEY(price, INDEXED),

RELATION(detail, contracts),

RELATION(supplier, contracts)));

};

所有数据库中使用的类都要定义类型描述符。

### REGISTER

除了定义类型描述符外，还必须在C++类和数据库表之间建立一个映像。宏REGISTER(name)就做这个工作。与TYPE\_DESCRIPTOR 宏不同的是，REGISTER 宏应该在实现文件中使用而不是在头文件中。该宏构造一个与类相连的表的描述符。如果你要在一个应用中使用多个数据库，那么就可能使用REGISTER\_IN(name,database)宏在一个具体数据库中注册一个表。该宏的database 参数应该是一个指向dbDatabase 对象的指针。

你可以像下面这样注册数据库的表：

REGISTER(Detail);

REGISTER(Supplier);

REGISTER(Contract);

表（以及对应的类）在每一时刻只能对应于一个数据库。当你打开一个数据库，fastdb向数据库中导入所有在应用中定义的类。如果一个同名的类在数据库中已经存在，则会比较描述符在数据库中的类与描述符在应用中的类，如果两个类的定义不同，则fastdb试图将该表转换成新的格式。数值类型之间的任何转换(整形到实型，实型到整形，扩展或者截断)都是允许的。增加字段也很容易，但是只有对空表才可以删除字段(以避免偶然的数据崩溃)。

装载所有的类描述符后，fastdb 就检查在应用程序的类描述符中指定的索引是否存在于数据库中、创建新的索引并且删除不再使用的索引。只有在不超过一个应用程序访问数据库时才可以进行表的重构以及增加/删除索引。所以只有第一个与数据库关联的应用程序可以进行表的转换，所有其余的应用只能向数据库中增加新类。

有一个特殊的内部表Metatable，该表包含了数据库中所有其他表的信息。C++程序员不需要访问这个表，因为数据库表的结构是由C++类指定的。但在交互SQL 程序中，还是有必要检查这个表来获取记录字段的信息。

从版本2.30开始，fastdb支持自增字段(有数据自动赋值的值唯一的字段)。要使用自增字段必须：

1. 带上-DAUTOINCREMENT\_SUPPROT 标志重新编译fastdb 和你的应用程序。（在fastdb makefile 中的DEFS变量中增加这个标志）。

注意：不带该标记编译的fastdb创建的数据库文件与带标记编译的fastdb创建的数据库文件不兼容。

1. 如果你要使用初始值非0的计数器，则必须给dbTableDescriptor::initialAutoincrementCount赋个值。该变量由所有的表共享，因此所有的表都有一个共同初始值的自增计数器。
2. 自增字段必须是int4 类型，并且必须用AUTOINCREMENT标志声明

class Record {

int4 rid;

char const\* name;

...

TYPE\_DESCRIPTOR((KEY(rid, AUTOINCREMENT|INDEXED), FIELD(name), ...));

}

1. 当要在数据库中插入带有自增字段的记录时不需要为自增字段赋值(将会被忽略)。当插入成功后，该字段将被赋给一个唯一的值(这样确保在数据库中从未使用过)：

Record rec;

// no rec.rid should be specified

rec.name = "John Smith";

insert(rec);

// rec.rid now assigned unique value

int newRecordId = rec.rid; // and can be used to reference this record

1. 当记录被删除该值将不会再使用，当事务中止时，表的自增计数器也将回滚。

## 查询Query

query 类用于两个目的:

1. 构造一个查询并绑定查询参数
2. 作为已编译的查询的缓存

fastdb 提供重载的c++运算符'='和','来构造带参数的查询语句。参数可以在被使用的地方直接指定，消除了在参数占位符和c变量之间的任何映像，在下面的查询示例中，参数price和quantity的指针保存在查询中，因此该查询可以用不同的参数执行多次。c++函数重载使之可以自动的确定参数的类型，不需要程序员提供额外信息（从而减少了bug 的可能性）。

dbQuery q;

int price, quantity;

q = "price >=",price,"or quantity >=",quantity;

由于char \*可以用来指定一个查询的分片(fraction)(例如"price >=")和一个字符串类型的参数，fastdb 使用了一个特别的规则来解决这个模糊性。该规则基于这样一个假定即没有理由把一个查询文本分解成两个字符串如("price",">=")或者指定多于一个的参数序列("color=",color,color)。因此fastdb假定第一个字符串是该查询文本的一个分片并且随之转换到操作数模式。在操作数模式中，fastdb 认为char \* 参数是一个查询参数然后切换回到查询文本模式，依此类推。

也可以不用这个“句法糖”（syntax sugar）而是显示的通过dbQuery::append(dbQueryElement::ElementType type, void const\* ptr)方法来构造查询元素。在向查询添加元素之前，必须通过dbQuery::reset()方法来重置查询（'operator='自动作了这个事情）。

不能使用c++数值常量来作为查询参数，因为参数是通过引用来访问的。但可以使用字符串常量，因为字符串是传值的。有两种方法在一个查询中指定字符串参数：使用一个字符串缓冲或一个指向字符串指针的指针：

dbQuery q;

char\* type;

char name[256];

q = "name=",name,"and type=",&type;

scanf("%s", name);

type = "A";

cursor.select(q);

...

scanf("%s", name);

type = "B";

cursor.select(q);

...

查询变量既不能作为一个参数传给一个函数也不能赋给另一个变量。当fastdb 编译查询时，会把编译树存到该对象中。下一次使用该查询时，不需要再次编译并且可以使用已编译好的树。这样节省了一些编译查询的时间。

fastdb 提供了两个方法来集成数据库中的用户自定义类型。第一种方法-定义类方法-已经讨论过，另一个方法只处理查询构造。程序员要定义方法，该方法并不作确实的运算，而是返回一个表达式（根据预先定义的数据库类型）,该方法来执行必要的查询。最好通过例子来说明这点。fastdb 没有内置的日期时间类型，而是使用一个普通的c++类dbDateTime。该类定义了方法用来在有序列表中指定日期时间字段和使用通常的运算符来比较两个日期。

class dbDateTime {

int4 stamp;

public:

...

dbQueryExpression operator == (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),"=",stamp;

return expr;

}

dbQueryExpression operator != (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),"<>",stamp;

return expr;

}

dbQueryExpression operator < (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),">",stamp;

return expr;

}

dbQueryExpression operator <= (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),">=",stamp;

return expr;

}

dbQueryExpression operator > (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),"<",stamp;

return expr;

}

dbQueryExpression operator >= (char const\* field) {

dbQueryExpression expr;

expr = dbComponent(field,"stamp"),"<=",stamp;

return expr;

}

friend dbQueryExpression between(char const\* field, dbDateTime& from,

dbDateTime& till)

{

dbQueryExpression expr;

expr=dbComponent(field,"stamp"),"between",from.stamp,"and",till.stamp;

return expr;

}

friend dbQueryExpression ascent(char const\* field) {

dbQueryExpression expr;

expr=dbComponent(field,"stamp");

return expr;

}

friend dbQueryExpression descent(char const\* field) {

dbQueryExpression expr;

expr=dbComponent(field,"stamp"),"desc";

return expr;

}

};

所有这些方法接受参数作为一个记录的字段的名字，该名字用来构造一个记录组件的全名。使用类dbComponent来作这个事情，该类把结构字段的名字和结构组件的名字组合成一个用'.'符号分隔的复合名字。类dbQueryExpression用来收集表达式项，表达式自动的用圆括号括起来，消除了运算符优先级引起的冲突。

假定一个记录包含了一个字段dbDateTime 类型的字段delivery，可以如下构造查询：

dbDateTime from, till;

q1 = between("delivery", from, till),"order by",ascent("delivery");

q2 = till >= "delivery";

除了这些方法外，一些类指定方法也可以用这种方法定义，例如一个区域类型的overlaps方法。这种方法的好处是数据库引擎可以使用预定义的类型并且可以使用索引和其他的一些优化方法来执行查询。另一方面，这些类的实现的封装已保留，因此程序员在一个类的表示改变时不应该重写所有的查询。下面这些c++类型可以用作查询参数：

|  |  |
| --- | --- |
| int1 | bool |
| int2 | char const\* |
| int4 | char \*\* |
| int8 | char const\*\* |
| real4 | dbReference<T> |
| real8 | dbArray< dbReference<T> > |

## 游标Cursor

游标用来访问选择语句返回的记录。fastdb提供了有类型的游标，也就是说，与具体表相关的游标。fastdb 有两种游标：只读游标和用于更新的游标。fastdb中的游标用c++模板类dbCursor<T>来表示，其中T 为与数据库表相关的C++类的的名字。游标类型必须在构造游标的时候指定。缺省创建一个只读游标。要创建一个用于更新的游标，必须给构造函数传递一个dbCursorForUpdate 参数。

一个类T 的游标包含一个类T的实例，用来获得当前的记录。这就是为什么表类必须要有一个缺省构造函数(无参数的构造函数)而没有副作用。fastdb 优化了从数据库中获取记录，只从对象的固定部分复制记录。字符串本身并不复制，而是使相应的字段直接指向数据库中。数组也是如此：他们的组件在数据库中的表示跟在应用程序中的表示是一样的（标量类型的数组或者标量组件的嵌套结构的数组）。

应用程序不能直接改变数据库中的字符串和数据元素。当一个数组方法需要更新一个数组体时，先在内存中创建一个副本然后更新这个副本。如果程序员要更新字符串字段，他应该给这个指针赋一个新值，而不是直接修改数据库里的字符串。对于字符串元素建议使用char const \* 而不是char \*，以使编译器可以检查对字符串的非法使用。

### select(dbQuery &q) / select()

执行一个查询要么通过游标select（dbQuery &q）方法，要么通过select()方法，后者可以迭代表中的所有记录。两个方法都返回选中的记录的数量，并且把当前位置置于第一个记录(如果有的话)。

### next(),prev(),first(),last()

游标可以前滚或者后滚。next(),prev(),first(),last()方法可以用来改变游标的当前位置。如果由于没有（更多）的记录存在而使得操作无法进行，这些方法将返回NULL，并且游标的位置不改变。

### get()

游标类提供了get（）方法来获得指向当前记录（保存在游标中）的指针。重载的运算符' ->'也可以用来访问当前记录的元素。

### update()

如果一个游标以更新方式打开，就可以改变当前的记录并且用update()方法保存到数据库中或者被删除掉。如果当前记录被删除，下一个记录变为当前记录。如果没有下一个记录，则前一个记录（如果有的话）变为当前。

### removeAll()/removeAllSelected()

removeAll()方法删除表中的所有记录。而removeAllSelected方法只删除游标选择的所有记录。

### reset()

当更新记录后，数据库的大小可能会增加。从而虚拟存储器的数据库区域需要进行扩展。该重映射的后果之一就是，该区域的基地址可能发生变化，然后应用程序中保存的所有数据库指针都变得无效。当数据库区域重映象时fastdb自动更新所有打开的游标中的当前记录。因此，当一个数据库更新时，程序员应该通过游标的->方法来访问记录字段，而不应该使用指针变量。

当前选择使用的内存可以通过reset()方法来释放。该方法自动的由select()、dbDatabase::commit()、dbDatabase::rollback()方法以及游标的销毁（destructor）函数调用，因此大多数情况下不需要显式调用reset()方法。

### at(dbReference<T> const& ref)

游标也可以通过引用来访问记录。at(dbReference<T> const& ref)方法把游标指向引用所指的记录。在这种情况下，选择将只包含一个记录，而next(),prev()方法将总是返回NULL。

由于游标和引用在fastdb中是严格类型化的，所有必须的检查可以由编译器静态的进行而不需要动态类型检查。运行时唯一要进行的检查是对空引用的检查。

### currentId()

游标中当前记录的对象标识符可以用currentId()方法获得。

### setSelectionLimit(size\_t lim) / unsetSelectionLimit()

可以限制select 语句返回的记录的数目。游标类有两个方法setSelectionLimit(size\_t lim)和unsetSelectionLimit()用来设置/取消查询返回的记录数的限制。在某些情况下，程序员可能只需要几个记录或者头几个记录，从而查询的执行时间和消耗的内存大小可以通过限制选择的大小来降低。但如果你指定了被选记录的排序方式，只选择k 个记录的查询并不返回关键字最小的头k 个记录，而是返回任意k 个记录然后排序。

### insert()

所有数据库数据的操作都可以通过游标来进行，唯一的例外是插入操作，fastDB提供了一个重载的插入函数：

template<class T>

dbReference<T> insert(T const& record);

该函数将在表的末尾插入一个记录然后返回该对象的引用。fastdb 中插入的顺序是严格指定的因而应用程序可以使用表中记录排序方式的假定。

因为应用程序大量使用引用在对象之间导航，有必要提供一个根对象，从这个对象开始进行引用遍历。这样一个根对象的一个可取候选者就是表中的第一个记录（也是表中最老的记录）.该记录可以通过不带参数执行select（）方法来访问。游标中的当前记录就是表中的第一条记录。

### null

fastdb 的c++API 为引用类型定义了一个特殊的null 变量，可以用null 变量与引用比较或者赋给一个引用：

void update(dbReference<Contract> c) {

if (c != null) {

dbCursor<Contract> contract(dbCursorForUpdate);

contract.at(c);

contract->supplier = null;

}

}

### 延迟参数绑定

查询参数通常跟c++变量绑定。大多数情况下这是方便而且灵活的机制。但在多线程应用中，无法保证同一查询会在同一时刻不被另一线程以不同的参数执行。一个解决的方法是使用同步原语（临界区或者mutex）来排除查询的并发执行。但这种方法会导致性能退化。fastdb可以并行操作读操作而提高了整体系统吞吐量。另一个解决方法是使用延迟参数绑定。如下所示：

dbQuery q;

struct QueryParams {

int salary;

int age;

int rank;

};

void open()

{

QueryParams\* params = (QueryParams\*)NULL;

q = "salary > ", params->salary, "and age < ", params->age, "and rank =", params->rank;

}

void find(int salary, int age, int rank)

{

QueryParams params;

params.salary = salary;

params.age = age;

params.rank = rank;

dbCursor<Person> cusor;

if (cursor.select(q, &params) > 0) {

do {

cout << cursor->name << NL;

} while (cursor.next());

}

}

在这个例子中open 函数只为结构中的字段偏移绑定查询变量。然后再find 函数中，指向带有参数的结构的真实的指针传递给select 结构。find 函数可以被多个线程并发执行而只有一个编译好的查询被所有这些线程使用。这种机制从版本2.25 开始使用。

## 数据库dbDatabase

dbDatabase类控制应用与数据库的交互，进行数据库并发访问的同步，事务处理，内存分配，出错处理...dbDatabase对象的构造函数允许程序员制定一些数据库参数。

fastdb 可以并行的编译和执行查询，在多处理器系统中显著的提高了性能。但不能并发更新数据库(这是为了尽可能少的日志事务(log-less transaction)机制和0 等待恢复的代价).当一个应用程序试图改变数据库(打开一个更新游标或者在表中插入新记录)时，首先就以排他方式锁住数据库，禁止其他应用程序访问数据库，即使是只读的查询。这样来避免锁住数据库应用程序过长的时间，改变事务应当尽可能的短。在该事务中不能进行堵塞操作(如等待用户的输入).

在数据库层只使用共享锁和排它锁使得fastdb 几乎清除锁开销从而优化无冲突操作的执行速度。但是如果多个应用同时更新数据库的不同部分，则fastdb使用的方法将非常低效。这就是为什么fastdb 主要适用于单应用程序数据局部访问模式或者多应用读优势(read-dominated)访问模式模型。

在多线程应用中游标对象应该只由一个线程使用。如果在你的应用中有超过一个的线程，则在每个线程中使用局部游标变量。在线程间有可能共享查询变量，但要注意查询参数。查询要么没有参数，要么使用相关的参数绑定形式。

数据库对象由所有的线程共享，使用线程专有数据来进行查询的同步代价最小的并行编译和执行。需要同步的全局的东西很少：符号表，树结点池…..。但是扫描、解析和执行查询可以不需要任何的同步来进行，如果有多处理器系统的高层并发机制。一个数据库事务由第一个选择或者插入操作开始。如果使用用于更新的游标，则数据库以排他方式锁住，禁止其他应用和线程访问数据库。如果使用只读游标，这数据库以共享模式锁住，防止其他的应用或者线程改变数据库，但允许并发读请求的执行。一个事务必须显示终止，要么通过dbDatabase::commit()方法提交该事务对数据库的所有更改，或者通过dbDatabase::rollback()方法来取消事务所作的所有更改。dbDatabase::close()方法自动提交当前事务

如果你使用只读游标来执行选择从而启动一个事务然后又使用更新游标来对数据库作某些改变，则数据库将首先以共享模式锁住，然后锁变成排他模式。如果该数据被多个应用访问这种情况可能会造成死锁。想象一下应用A 启动了一个读事务而应用B 也启动了一个读事务。二者都拥有数据库的共享锁。如果二者都试图把它们的锁改变为排他模式，他们将永远被互相堵塞(另外一个进程的共享锁存在时不能授予排它锁)。为了避免这种情况，在事务开始就试着使用更新游标，或者显示的使用dbDatabase::lock()方法。关于fastdb 中事务实现的信息可以参见《事务》这一节。

### 构造函数

dbDatabase对象的构造函数允许程序员制定一些数据库参数：

dbDatabase(dbAccessType type = dbAllAccess,

size\_t dbInitSize = dbDefaultInitDatabaseSize,

size\_t dbExtensionQuantum = dbDefaultExtensionQuantum,

size\_t dbInitIndexSize = dbDefaultInitIndexSize,

int nThreads = 1);

#### type

支持下面的数据库访问类型：

|  |  |
| --- | --- |
| **Access type** | **Description** |
| dbDatabase::dbReadOnly | 只读模式 |
| dbDatabase::dbAllAccess | 正常模式 |
| dbDatabase::dbConcurrentRead | 与dbConcurrentUpdate结合使用，在更新数据库的同时可以以只读模式访问数据库 |
| dbDatabase::dbConcurrentUpdate | 与dbConcurrentRead结合使用的模式，可在数据库中执行更新，而不会长时间阻止读取应用程序 |

当数据库以只读方式打开时，不能向数据库添加新的类定义，不能改变已有的类的定义和索引。

在数据库主要用只读模式访问而更新不应该长时间堵塞读操作的情况下应该同时使用dbConcurrentUpdate 和dbConcurrentRead 模式。在这种模式下更新数据库可以与读访问并发执行(读将不会看到改变的数据直到事务提交)。只有在更新事务提交时，才会设置排他锁然后在当前对象索引自增改变(incremental change)之后马上释放掉。于是你可以使用dbConcurrentRead 模式启动一个或多个应用而其读事务将并发执行。也可以使用dbConcurrentUpdate 模式启动一个或多个应用。所有这些应用的事务将通过一个全局的mutex 来同步。因此这些事务(甚至是只读)将排他性的执行。但是dbConcurrentUpdate模式的应用的事务可以与dbConcurrentRead模式的应用的事务并发运行。请参阅testconc.cpp例子，里边说明了这些模式的使用方法。

注意！不要把dbConcurrentUpdate 和dbConcurrentRead模式与其他模式混合使用，也不要在一个进程中同时使用他们(因此不能启动两个线程其中一个用dbConcurrentUpdate模式打开数据库另一个用dbConcurrentRead 模式)。在dbConcurrentUpdate 模式下不要使用dbDatabase::precommit 方法。

#### dbInitSize

dbInitSize参数指定了数据库文件的初始大小。数据库文件按照需要增长；设置初始大小只是为了减少重新分配空间(会占用很多时间)的次数。在当前fastdb 数据库的实现中该大小在每次扩展时至少加倍。该参数的缺省值为4 兆字节。

#### dbExtensionQuantum

dbExtensionQuantum定了内存分配位图的扩展量子。简单的说，这个参数的值指定了在不需要试图重用被释放的对象的空间时分配多少连续内存空间。缺省值为4MB.

#### dbInitIndexSize

dbInitIndexSize 参数指定了初始的索引大小。fastdb 中的所有对象都通过一个对象索引访问。该对象索引有两个副本：当前的和已提交的。对象索引按照需要重新分配，设置一个初始值只是为了减少(或者增加)重新分配的次数。该参数的缺省值是64K个对象标识符。

#### nThreads

nThreads 控制并行查询的层次。如果大于1，则fastdb启动一些查询的并行执行(包括对结果排序)。在这种情况下，fastdb 引擎将派生指定数目的并行线程。通常为该参数指定超过系统中在线CPU数目的值是没有意义的。也可以为该参数传递0值，这时fastdb 将自动侦测系统中在线cpu 的数目。在任何时候都可以用dbDatabase::setConcurrency来指定线程数。

dbDatabase类包含一个静态字段dbParallelScanThreshold，该字段指定了在使用并行查询后表中记录数的一个阈值，缺省为1000。

### 打开函数open

可以用

open( char const\* databaseName, char const\* fileName = NULL,

unsigned waitLockTimeout = INFINITE

)

方法来打开数据库。

如果数据库成功打开，open方法返回true,否则返回false。在后面这种情况，数据库的handleError 方法将带上DatabaseOpenError 错误码被调用。

#### databaseName

数据库名应该是由除了‘\’之外的任意符号组成的标识符。

数据库名称用于构造同步原语和共享内存的名称。如果两个进程打开具有相同数据库名称但不同文件名的数据库，则会导致数据库文件损坏。请注意，Windows和Unix之间的命名系统资源有所不同。在Unix中，资源以与文件相同的方式解析(实际上要访问一些需要打开文件的命名系统资源，并使用ftok函数从文件句柄获取密钥)。在Windows资源中有自己的名称空间。因此，如果你使用文件/数据库名称的相对路径，例如open(“mydatabase”)并从两个不同的目录生成两个进程，则在Windows中，它们将创建不同的数据库文件，但使用相同的同步原语。这导致数据库损坏。请在Windows中使用相对路径非常小心。

#### fileName

如果文件名参数省略，则通过数据库名加一个后缀“.fdb"来创建一个文件。

#### waitLockTimeout

waitLockTimeout 可以设置用来防止当工作于该数据库的所有活动进程中的某些进程崩溃时把所有的进程锁住。如果崩溃的进程锁住了数据库，则其他的进程都将无法继续执行。为了防止这种情况，可以指定一个等待该锁的最大延迟，当该延迟过期后，系统将试图进行恢复并继续执行活动的进程。

### close

一个数据库会话可以用close方法中止，该方法隐含的提交当前事务。

### attach

在一个多线程的应用中，每一个要访问数据库的线程都应该首先与数据库粘附(attach)。dbDatabase::attach()方法分配线程指定数据然后把线程与数据库粘附起来。该方法自动由open()方法调用。因此没有理由为打开数据的线程调用attach()方法。

### detach

当该线程工作完毕，应当调用dbDatabase::detach() 方法。close 方法自动调用detach()方法。detach()方法隐含提交当前事务。一个已经分离的线程试图访问数据库时将产生断言错误(assertionfailure)。

### lock

可以使用lock()方法来显示的锁住数据库。锁通常是自动进行的。只有很少的情况下你才需要使用这个方法。它将以排他方式锁住数据库直到当前事务结束。

### backup

可以用dbDatabase::backup(char const\* file)方法来备份数据库。备份操作将以共享模式锁住数据然后从内存向指定的文件刷新数据库的映像。因为使用了影子对象索引，数据库文件总是处于一致状态，因此从备份恢复只需要把备份文件改一下名字(如果备份被放到磁带，则首先要把文件恢复到磁盘)。

### handleError

dbDatabase类也负责处理一些应用的错误，如编译查询时的句法错误，执行查询时的索引越界或者空引用访问。由一个虚方法dbDatabase::handleError来处理这些错误。

virtual void handleError(dbErrorClass error,

char const\* msg = NULL,

int arg = 0);

程序员可以从dbDatabase 类来派生出自定义的子类，并重定义缺省的错误处理。

|  |  |  |  |
| --- | --- | --- | --- |
| Error classes and default handling | | | |
| **Class** | **Description** | **Argument** | **Default reaction** |
| QueryError | 查询编译错误 | position in query string | abort compilation |
| ArithmeticError | arithmetic error during division or power operations | - | terminate application |
| IndexOutOfRangeError | 索引越界 | value of index | terminate application |
| DatabaseOpenError | 数据库打开时出错 | - | open method will return false |
| FileError | 文件IO操作失败 | error code | terminate application |
| OutOfMemoryError | 没有足够的内存用于对象分配 | requested allocation size | terminate application |
| Deadlock | 升级锁导致死锁 | - | terminate application |
| NullReferenceError | 查询执行期间访问空引用 | - | terminate application |

# 常用API

## 全局API

### insert

|  |  |
| --- | --- |
| 函数原型： | template<class T> deReference<T> insert(T const& record); |
| 函数参数： | record：[i] 待插入的记录 |
| 返回值： | 插入记录的引用 |
| 函数功能： | 将record插入对应类型的表中 |

### remove

|  |  |
| --- | --- |
| 函数原型： | template<class T> deReference<T> insert(T const& record); |
| 函数参数： | record：[i] 待插入的记录 |
| 返回值： | 插入记录的引用 |
| 函数功能： | 将record插入对应类型的表中 |

### USER\_FUNC

### REGISTER\_TEMPLATE

## dbDatabase

### dbDatabase

|  |  |
| --- | --- |
| 函数原型： | dbDatabase(dbAccessType type = dbAllAccess,  size\_t dbInitSize = dbDefaultInitDatabaseSize,  size\_t dbExtensionQuantum = dbDefaultExtensionQuantum,  size\_t dbInitIndexSize = dbDefaultInitIndexSize,  int nThreads = 1, …); |
| 函数参数： | type：[i] 访问类型 dbDatabase::dbReadOnly或dbDatabase::dbAllAccess |
| dbInitSize：[i]数据库的初始大小。如果Fastdb使用DISKLESS\_CONFIGURATION选项编译，应该在这个参数中设置MAXIMAL(这种方式下数据库不能重新分配) |
| dbExtensionQuantum：[i] 用于扩展内存分配位图的量子 |
| dbInitIndexSize：[i] 初始索引大小 |
| nThreads：[i] 顺序搜索和排序操作的并发级别 |
| ….：[i] 其余参数不需要指定，Fastdb内部用于检查编译选项 |
| 返回值： | 无 |
| 函数功能： | dbDatabase构造函数 |

### setFixedSizeAllocator

|  |  |
| --- | --- |
| 函数原型： | void setFixedSizeAllocator(size\_t minSize, size\_t maxSize, size\_t quantum, size\_t bufSize) |
| 函数参数： | minSize：[i] 分配器分配的最小对象大小 |
| maxSize：[i] 分配器分配的最大对象大小 |
| quantum：[i] 固定大小和分配器链之间的大小差异 |
| bufSize：[i] 该分配器拥有最大数量的空闲元素 |
| 返回值： | 无 |
| 函数功能： | 初始化固定大小分配器 |

### open

|  |  |
| --- | --- |
| 函数原型： | bool open(char\_t const\* databaseName,  char\_t const\* fileName = NULL,  time\_t waitLockTimeoutMsec = INFINITE,  time\_t commitDelaySec = 0); |
| 函数参数： | databaseName：[i] 数据库名称 |
| fileName：[i] 数据库文件的路径，如果为空，则文件名为databaseName + “.fdb” |
| waitLockTimeoutMsec：[i] 等待锁超时，默认禁用 |
| commitDelaySec：[i] 延迟提交超时，默认禁用 |
| 返回值： | 数据库成功打开返回true，否则返回false |
| 函数功能： | 打开数据库 |

|  |  |
| --- | --- |
| 函数原型： | bool open(OpenParameters& params); |
| 函数参数： | params：[i] 打开数据库的参数 |
| 返回值： | 数据库成功打开返回true，否则返回false |
| 函数功能： | 使用OpenParameters参数打开数据库 |

### close

|  |  |
| --- | --- |
| 函数原型： | void close() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 关闭数据库 |

### preCommit

|  |  |
| --- | --- |
| 函数原型： | void preCommit() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 释放事务的所有锁，允许其他客户端进行操作，但不刷新对磁盘的更改 |

### commit

|  |  |
| --- | --- |
| 函数原型： | void commit(); |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 提交事务 |

### rollback

|  |  |
| --- | --- |
| 函数原型： | void rollback(); |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 回滚事务 |

### scheduleBackup

|  |  |
| --- | --- |
| 函数原型： | void scheduleBackup(char\_t const\* fileName, time\_t periodSec); |
| 函数参数： | fileName：[i] 备份文件路径。如果名称以'？'结尾，那么每个备份将被放置在使用当前时间戳替换'？'的单独文件中 |
| periodSec：[i] 执行备份的时间，以秒为单位 |
| 返回值： | 无 |
| 函数功能： | 安排备份数据库 |

### bakcup

|  |  |
| --- | --- |
| 函数原型： | bool backup(char\_t const\* file, bool compactify); |
| 函数参数： | file：[i] 备份文件路径 |
| compactify：[i] 如果为true，那么数据库将在备份期间被压缩 - 即所有使用的对象将被放置在一起而没有孔; 如果为false，通过将内存映射对象写入备份文件来执行备份。 |
| 返回值： | 无 |
| 函数功能： | 以指定的名称对文件进行备份 |

|  |  |
| --- | --- |
| 函数原型： | bool backup(dbFile\* file, bool compactify); |
| 函数参数： | file：[i] 备份文件路径。备份完成后此文件不会被关闭 |
| compactify：[i] 如果为true，那么数据库将在备份期间被压缩 - 即所有使用的对象将被放置在一起而没有孔; 如果为false，通过将内存映射对象写入备份文件来执行备份。 |
| 返回值： | 无 |
| 函数功能： | 备份到指定文件 |

### setConcurrency

|  |  |
| --- | --- |
| 函数原型： | void setConcurrency(unsigned nThreads); |
| 函数参数： | nThreads：[i] 为顺序搜索和排序操作创建的最大线程数 |
| 返回值： | 无 |
| 函数功能： | 设置顺序搜索和排序操作的并发级别。默认情况下Fastdb尝试检测系统中的CPU数量并创建相同数量的线程 |

### attach

|  |  |
| --- | --- |
| 函数原型： | void attach(); |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 将当前线程附加到数据库。该方法应该由除打开数据库的线程外的其他所有线程执行。 |

|  |  |
| --- | --- |
| 函数原型： | void attach(dbDatabaseThreadContext\* ctx); |
| 函数参数： | ctx：[i] 将与当前线程相关联的事务上下文 |
| 返回值： | 无 |
| 函数功能： | 设置当前线程的事务上下文。 使用此方法可以在不同线程之间共享相同的事务 |

### detach

|  |  |
| --- | --- |
| 函数原型： | void detach(int flags = COMMIT|DESTROY\_CONTEXT); |
| 函数参数： | flags：[i] DetachFlags COMMIT和DESTROY\_CONTEXT的组合 |
| 返回值： | 无 |
| 函数功能： | 从数据库中删除线程 |

### lock

|  |  |
| --- | --- |
| 函数原型： | void lock(dbLockType lock = dbExclusiveLock) |
| 函数参数： | minSize：[i] 分配器分配的最小对象大小 |
| 返回值： | 无 |
| 函数功能： | 排他性的锁定数据库 |

### insert

|  |  |
| --- | --- |
| 函数原型： | template<class T> dbReference<T> insert(T const& record) |
| 函数参数： | record：[i] 要插入到数据库中的临时记录 |
| 返回值： | 创建对象的引用 |
| 函数功能： | 在数据库中插入记录 |

### OpenParameters

|  |  |
| --- | --- |
| struct OpenParameters | |
| char\_t const\* databaseName | 数据库名称 |
| char\_t const\* databaseFilePath | 数据库文件路径 |
| time\_t transactionCommitDelay | 事务提交延迟时间 |
| time\_t waitLockTimeoutMsec | 死锁检测超时（在此时间过期后锁定被撤销） |
| dbAccessType accessType | 数据库访问类型 |
| size\_t initSize | 初始数据库文件大小 |
| size\_t extensionQuantum | 扩展内存分配位图量子 |
| size\_t initIndexSize | 初始数据库索引大小 |
| int nThreads | 顺序搜索与排序操作的并行级别 |
| offs\_t freeSpaceReuseThreshold | 释放分配空间量的阈值，之后分配位图重新从开始使用释放对象 |
| int parallelScanThreshold | 指定表中最少的记录数量，启动并行执行顺序搜索 |
| int nodeId | 复制的数据库节点ID |
| char\*\* nodeAddresses | 复制的数据库节点地址 |
| int nNodes | 备份数据库中的节点数 |
| int pollInterval | 轮询节点间隔 |
| int waitReadyTimeout | 当新的主节点在启动时变为活动状态时，待机节点等待确认超时 |
| int waitStatusTimeout | 启动期间请求其他节点状态超时 |
| int recoveryConnectionAttempts | 在数据恢复时与其他节点建立连接的最大尝试次数 |
| int startupConnectionAttempts | 在启动时与其他节点建立连接的最大尝试次数 |
| int replicationWriteTimeout | 写入复制节点超时。如果在指定的超时时间内无法完成写入，则认为节点已经死机，连接被挂起。 |
| int maxAsyncRecoveryIterations | 异步恢复迭代的最大数量。 如果无法在指定的迭代次数内完成主数据库恢复的永久更新，则执行同步恢复（主机将不会执行任何事务提交直到恢复结束）。 |
| int fileOpenFlags | 在dbFile :: OpenFlags中定义的标志的位掩码。这些标志指定打开数据库文件的方式 |

### setTransactionLogger

|  |  |
| --- | --- |
| 函数原型： | void setTransactionLogger(dbTransactionLogger\* logger); |
| 函数参数： | logger：[i] 事务日志，设为NULL时禁用日志记录 |
| 返回值： | 无 |
| 函数功能： | 设置事务记录器。该方法用于将事务记录到某些外部目的地，当从这个源重播事务时，不应该保存日志。 |

## dbReplicatedDatabase

### dbReplicatedDatabase

### open

## dbQuery

## dbCursor

### dbCursor

|  |  |
| --- | --- |
| 函数原型： | dbCursor(dbCursorType type = dbCursorViewOnly) |
| 函数参数： | type：[i] 游标类型 |
| 返回值： | 无 |
| 函数功能： | 构造函数。 |

|  |  |
| --- | --- |
| 函数原型： | dbCursor(dbDatabase\* aDb, dbCursorType type = dbCursorViewOnly) |
| 函数参数： | aDb：[i] 执行表查找的数据库 |
|  | type：[i] 游标类型 |
| 返回值： | 无 |
| 函数功能： | 显式指定数据库的游标构造函数。此游标应用于未分配的表。 |

### currentId

|  |  |
| --- | --- |
| 函数原型： | dbReference<T> currentId() const |
| 函数参数： | 无 |
| 返回值： | 当前记录的引用 |
| 函数功能： | 获取当前对象标识 |

### setSelectionLimit

|  |  |
| --- | --- |
| 函数原型： | void setSelectionLimit(cardinality\_t lim) |
| 函数参数： | lim：[i] 最大记录个数 |
| 返回值： | 无 |
| 函数功能： | 指定要选择的最大记录数 |

### unsetSelectionLimit

|  |  |
| --- | --- |
| 函数原型： | void unsetSelectionLimit() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 删除选择限制 |

### select

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select(dbQuery& query, dbCursorType aType, void\* paramStruct = NULL) |
| 函数参数： | query：[i] 选择条件 |
| aType：[i] 游标类型：dbCursorForUpdate，dbCursorViewOnly |
| paramStruct：[i] 指向带有参数的结构的指针。如果要创建可重入的预编译查询，即在不同的线程同时使用同一query的情况下，应该避免使用静态变量，而是将参数放入一些结构中，在查询中指定相对于参数的偏移量，填充本地结构并将指针传递给select方法 |
| 返回值： | 所选记录的数量 |
| 函数功能： | 执行查询 |

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select(dbQuery& query, void\* paramStruct = NULL) |
| 函数参数： | query：[i] 选择条件 |
| paramStruct：[i] 指向带有参数的结构的指针。如果要创建可重入的预编译查询，即在不同的线程同时使用同一query的情况下，应该避免使用静态变量，而是将参数放入一些结构中，在查询中指定相对于参数的偏移量，填充本地结构并将指针传递给select方法 |
| 返回值： | 所选记录的数量 |
| 函数功能： | 使用默认游标类型执行查询 |

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select(char const\* condition, dbCursorType aType, void\* paramStruct = NULL) |
| 函数参数： | condition：[i] 选择标准 |
| aType：[i] 游标类型：dbCursorForUpdate，dbCursorViewOnly |
| paramStruct：[i] 指向带有参数的结构的指针。如果要创建可重入的预编译查询，即在不同的线程同时使用同一query的情况下，应该避免使用静态变量，而是将参数放入一些结构中，在查询中指定相对于参数的偏移量，填充本地结构并将指针传递给select方法 |
| 返回值： | 所选记录的数量 |
| 函数功能： | 执行查询 |

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select(char const\* condition, void\* paramStruct = NULL) |
| 函数参数： | condition：[i] 选择标准 |
| paramStruct：[i] 指向带有参数的结构的指针。如果要创建可重入的预编译查询，即在不同的线程同时使用同一query的情况下，应该避免使用静态变量，而是将参数放入一些结构中，在查询中指定相对于参数的偏移量，填充本地结构并将指针传递给select方法 |
| 返回值： | 所选记录的数量 |
| 函数功能： | 使用默认游标类型执行查询 |

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select(dbCursorType aType) |
| 函数参数： | aType：[i] 游标类型：dbCursorForUpdate，dbCursorViewOnly |
| 返回值： | 所选记录的数量 |
| 函数功能： | 查询表中所有记录 |

|  |  |
| --- | --- |
| 函数原型： | cardinality\_t select() |
| 函数参数： | 无 |
| 返回值： | 所选记录的数量 |
| 函数功能： | 使用默认游标类型查询表中所有记录 |

### update

|  |  |
| --- | --- |
| 函数原型： | void update() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 更新当前记录。更改当前记录的值，然后调用update方法将更改保存到数据库 |

### remove

|  |  |
| --- | --- |
| 函数原型： | void remove() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 删除当前记录 |

### removeAll

|  |  |
| --- | --- |
| 函数原型： | void update() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 删除表中所有记录 |

### removeAllSelected

|  |  |
| --- | --- |
| 函数原型： | void removeAllSelected() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 删除所有选择的记录 |

### at

|  |  |
| --- | --- |
| 函数原型： | T\* at(dbReference<T> const& ref) |
| 函数参数： | ref：[i] 引用记录 |
| 返回值： | T\*：指向引用记录的指针 |
| 函数功能： | 通过引用选择记录 |

### next

|  |  |
| --- | --- |
| 函数原型： | T\* next () |
| 函数参数： | 无 |
| 返回值： | T\*：指向下一记录的指针，或没有下一记录返回NULL |
| 函数功能： | 获取下一个记录的指针 |

类似函数：prev、first、last；

### get

|  |  |
| --- | --- |
| 函数原型： | T\* get() |
| 函数参数： | 无 |
| 返回值： | T\*：指向当前记录，或没有当前记录返回NULL |
| 函数功能： | 获取指向当前记录的指针 |

## dbThread

### create

|  |  |
| --- | --- |
| 函数原型： | void create(thread\_proc\_t f, void\* arg) |
| 函数参数： | f：[i] 线程执行函数 |
| arg：[i] 线程执行参数 |
| 返回值： | 无 |
| 函数功能： | 创建线程 |

### join

|  |  |
| --- | --- |
| 函数原型： | void join() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 等待线程执行结束 |

### dbThread::sleep

|  |  |
| --- | --- |
| 函数原型： | static void sleep(time\_t sec) |
| 函数参数： | sec：[i] 休眠时间 毫秒 |
| 返回值： | 无 |
| 函数功能： | 线程休眠指定时长 |

## dbMutex

### lock

|  |  |
| --- | --- |
| 函数原型： | void lock() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 锁定当前线程 |

### unlock

|  |  |
| --- | --- |
| 函数原型： | void unlock() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 解锁当前线程 |

## dbLocalEvent

### wait

|  |  |
| --- | --- |
| 函数原型： | bool wait(dbMutex& mutex, time\_t timeoutMsec) |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： |  |

|  |  |
| --- | --- |
| 函数原型： | void wait(dbMutex& mutex) |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： |  |

## dbFileTransactionLogger

### open

|  |  |
| --- | --- |
| 函数原型： | bool open(char\_t const\* path, int flags = dbFile::no\_sync, bool crc = true) |
| 函数参数： | path：[i] 事务日志文件的路径 |
| flags：[i] dbFile：：OpenFlags标志。使用dbFile::truncate标志创建一个新文件 |
| crc：[i] 是否计算事务体的CRC |
| 返回值： | 打开文件成功返回true，否则返回false |
| 函数功能： | 打开事务日志文件 |

### close

|  |  |
| --- | --- |
| 函数原型： | void close() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 关闭事务日志文件 |

### restore

|  |  |
| --- | --- |
| 函数原型： | RestoreStatus restore(dbDatabase& db, size\_t& nTrans) |
| 函数参数： | db：[io] 被恢复的数据库 |
| nTrans：[o] 恢复事务的个数 |
| 返回值： | 恢复状态 rsOK, rsCRCMismatch, rsReadFailed, rsTableNotFound, rsOIDMismatch |
| 函数功能： | 从日志中恢复事务 |

### insert

|  |  |
| --- | --- |
| 函数原型： | bool insert(dbTableDescriptor\* table, oid\_t oid, dbRecord const\* dbsObj, void const\* appObj) |
| 函数参数： | table：[i] 插入对象的表的表描述符 |
| oid：[i] 对象标识 |
| dbsObj：[i] 数据库对象体 |
| appObj：[i] 应用程序对象体 |
| 返回值： | 插入成功返回true，否则返回false |
| 函数功能： | 插入一个新对象 |

### remove

|  |  |
| --- | --- |
| 函数原型： | virtual bool remove(dbTableDescriptor\* table, oid\_t oid) |
| 函数参数： | table：[i] 删除对象的表的表描述符 |
| oid：[i] 对象标识 |
| 返回值： | 删除成功返回true，否则返回false |
| 函数功能： | 删除一个对象 |

### update

|  |  |
| --- | --- |
| 函数原型： | virtual bool update(dbTableDescriptor\* table, oid\_t oid, dbRecord const\* dbsObj, void const\* appObj) |
| 函数参数： | table：[i] 更新对象的表的表描述符 |
| oid：[i] 对象标识 |
| dbsObj：[i] 数据库对象体 |
| appObj：[i] 应用程序对象体 |
| 返回值： | 更新成功返回true，否则返回false |
| 函数功能： | 更新一个对象 |

### commitPhase1

|  |  |
| --- | --- |
| 函数原型： | virtual bool commitPhase1() |
| 函数参数： | 无 |
| 返回值： | 提交成功返回true，否则返回false |
| 函数功能： | 执行事务提交的第一阶段 |

### commitPhase2

|  |  |
| --- | --- |
| 函数原型： | virtual void commitPhase2() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 执行事务提交的第二阶段 |

### roolback

|  |  |
| --- | --- |
| 函数原型： | virtual void rollback() |
| 函数参数： | 无 |
| 返回值： | 无 |
| 函数功能： | 回滚事务 |

## dbTimeSeriesBlock

## dbTimeSeriesProcessor

### dbTimeSeriesProcessor

### add

### getInterval