|  |
| --- |
| 完美时空 |
| XDB使用文档 |
| 面向XDB使用者 |
|  |
| **孙长明** |
| **2010/10/8** |

|  |
| --- |
| 本文档主要介绍完美时空自主研发的xdb数据库的使用接口，以及常见问题和模式。供应用开发者阅读。 |

目录

[基本信息 3](#_Toc274313308)

[连接方式 3](#_Toc274313309)

[基本元素：database、table、record 3](#_Toc274313310)

[关系数据库与对象数据库的区别 3](#_Toc274313311)

[如何学习xdb? 4](#_Toc274313312)

[哪里有例子？ 4](#_Toc274313313)

[数据定义 4](#_Toc274313314)

[表 4](#_Toc274313315)

[自定义数据类型：xbean 5](#_Toc274313316)

[any类型的xbean 7](#_Toc274313317)

[xbean变量的capacity 7](#_Toc274313318)

[xdb.xml 7](#_Toc274313319)

[根节点属性含义 8](#_Toc274313320)

[ProcedureConf节点属性含义 9](#_Toc274313321)

[生成代码 10](#_Toc274313322)

[启动xdb 10](#_Toc274313323)

[更改现有的表定义 10](#_Toc274313324)

[数据操作 10](#_Toc274313325)

[从表读取并（或）修改数据 10](#_Toc274313326)

[存储过程的调用方式 11](#_Toc274313327)

[存储过程的执行结果 12](#_Toc274313328)

[向表插入新的数据 12](#_Toc274313329)

[从表删除数据 12](#_Toc274313330)

[xbean的生命周期 12](#_Toc274313331)

[遍历一张硬盘表 14](#_Toc274313332)

[遍历一张内存表（或硬盘表在xdb中的缓存） 15](#_Toc274313333)

[批量修改数据 15](#_Toc274313334)

[存储过程嵌套 15](#_Toc274313335)

[关于持久性/checkpoint 16](#_Toc274313336)

[超时保护 16](#_Toc274313337)

[锁 17](#_Toc274313338)

[隔离度 17](#_Toc274313339)

[实现模型 17](#_Toc274313340)

[死锁怎么发生？ 18](#_Toc274313341)

[发生死锁怎么办？ 19](#_Toc274313342)

[怎么避免死锁 19](#_Toc274313343)

[在死锁的问题上，我们常常做不到最完美 19](#_Toc274313344)

[触发器 21](#_Toc274313345)

[注册触发器 21](#_Toc274313346)

[触发器的执行时刻 21](#_Toc274313347)

[触发器的粒度 21](#_Toc274313348)

[避免链式反应 21](#_Toc274313349)

[唯一名 21](#_Toc274313350)

[游戏服务器配置示例 22](#_Toc274313351)

[localid分配规则 23](#_Toc274313352)

[功能说明 23](#_Toc274313353)

[分配自增长ID 23](#_Toc274313354)

[分配名字 23](#_Toc274313355)

[分配ID 23](#_Toc274313356)

[编译唯一名 24](#_Toc274313357)

[获取发布版本 24](#_Toc274313358)

[设置编译脚本 24](#_Toc274313359)

[配置 25](#_Toc274313360)

[生成 25](#_Toc274313361)

[其它 25](#_Toc274313362)

[使用存储过程简化逻辑代码 25](#_Toc274313363)

[GS的线程模型 26](#_Toc274313364)

[协议的顺序 26](#_Toc274313365)

[定时任务 27](#_Toc274313366)

[随机数 27](#_Toc274313367)

[在存储过程中发送协议以及执行其它任务 27](#_Toc274313368)

[xdb.inuse文件 27](#_Toc274313369)

[xdb是数据库吗？ 28](#_Toc274313370)

# 基本信息

xdb是一个内存型数据库，不支持SQL，表与表之间的关系需要应用自己去维护。

它是一个数据存储的编程接口。

存储实现底层采用db.h，这个类似于Berkeley DB。

xdb基于JAVA,它内部使用JNI访问db.h。xdb只允许在java或groovy语言中使用，不提供其它语言的使用接口。

xdb是多线程的，同步访问数据。

xdb支持事务。可以回滚修改。

目前支持的平台有：

(1).win32：仅用于开发测试

(2).linux 64位: 推荐

(3).linux 32位:用户太少，未经充分测试。

特别强调：xdb是李成华写的，这个文档不是李成华写的。所以未必准确，并且一切有冲突的地方以李成华讲的为准。

## 连接方式

通过内存访问。数据库管理程序（DBMS）和应用逻辑在同一个进程内。类似的例子有：Berkeley DB，MS Office Access。

与之相反的有：完美的wdb，oracle的MySQL。 它通过TCP或其它网络方式连接到数据库管理程序，然后发送查询指令，然后把结果传回来。

## 基本元素：database、table、record

每个进程只能打开一个database，每个database里由很多table组成。每个table由很多record组成。

从程序的角度看，每个table就是一个map,维护key—>value这样的映射。每条映射就是一个record。map的key不可重复，key通常是int、long、string这样的简单类型。例如roleid、teamid、rolename等等。

value可以是简单类型，也可以是由简单类型组成的复杂类型（下面要讲的xbean）

## 关系数据库与对象数据库的区别

xdb不是一个关系型数据库。它是一个对象数据库。

关系数据库的基础是关系演算。关系数据库包含关系，即元组的集合。而对象数据库包含类，即对象的集合。  
在关系数据库中，元组的组成部分必须是基本类型。而对象数据库中，对象的组成部分可以是复杂类型（如vector,set）。

xdb的对象并不支持继承、多态。

关系数据库的基础是三值逻辑：true，false，null，而xdb不能往表里存储null.

## 如何学习xdb?

我的建议是：读文档，读xdb的实现代码，然后针对xdb的实现代码写测试用例。

多开口问。

## 哪里有例子？

snail/uniqname: 基于xdb的唯一名服务器

snail/gnet/gsd: 基于xdb的游戏逻辑服务器。

# 数据定义

这一章主要讲如何定义数据的表结构。对xdb而言，每个database由一个xml描述它。xdb第一次启动的时候，会根据这个xml创建相应的表。

它的基本结构如下：

<?xml version="1.0" encoding="utf-8"?>

<xdb>

<xbean name="Bag">

…

</xbean>

<table name="bag" key="long" value="Bag" cacheCapacity="10240" cachehigh="512" cachelow="256" lock="rolelock"/>

<xbean name="Pet">

…

</xbean>

<table name="pet" key="long" value="Pet" cacheCapacity="10240" cachehigh="512" cachelow="256"/>

</xdb>

xdb里数据类型分这么几类：

1. 基本的数据类型。有short、int、long、boolean、string 、binary 。（这些在后面介绍xbean的地方有详细介绍）
2. 容器：set/vector/list/map/treemap。容器里可以放基本数据类型和xbean。
3. xbean 用户自定义的，几种类型组合起来的复合类型。
4. cbean 用户自定义的，常量的xbean。

## 表

表通过这样的方式来定义：

<table name="produce" key="long" value="Produce" cacheCapacity="15000" cachehigh="512" cachelow="256" lock="rolelock" foreign="key:basic" autoIncrement="true" persistence="MEMORY" />

|  |  |  |
| --- | --- | --- |
| 属性名 | 说明 | 默认值 |
| name | 表名。必须是英文字母，全小写。 | 无 |
| key | key的类型。应该为int、long、string或cbean。 | 无 |
| value | value的类型。必须是基本类型或xbean。 | 无 |
| persistence | 表的存储类型分两种：硬盘表和内存表。persistence="MEMORY"代表内存表，不写则是硬盘表。硬盘表的数据会定时刷新到硬盘上，内存表则永远不会。对于金钱、元宝、物品、经验这些数据必然需要硬盘表。而对于组队、战斗这样的采用内存表即可，因为它更快。同时，我们约定，在服务器启动完毕开始接收协议之后，最好不要再遍历硬盘表。但是允许遍历内存表。any类型的xbean只能用于内存表,因为它不能被序列化。 | 硬盘表 |
| cacheCapacity | cache容量(单位：条)。对于内存表，万万不要让这个表的记录数超过cacheCapacity，如果超出，一些记录将被丢弃。 | 10240 |
| cacheCleanTry | 执行clean操作的时候如果没有清除到capacity以下，最多重试几次 | 256 |
| cachehigh | db.h的cachehigh（单位：页）。一般为512，不需要修改。 | 512 |
| cachelow | db.h的cachelow（单位：页）。一般为256，不需要修改。 | 256 |
| lock | lockname。在后面介绍锁的章节有详细介绍 | 和name字段相同 |
| foreign | 外键信息。key:basic是指，key指向basic表。该属性对内存表无效，对硬盘表只有合服时才有影响。 | "" |
| autoIncrement | key是不是自增长。如果为true，那么key采用自增长(要求key的类型必须是long) 。如果为false,那么每次插入的时候必须提供key。 | false |

## 自定义数据类型：xbean

xbean就是能回滚的java bean。每个xbean由一个或多个variable组成

xbean像协议一样，有marshal/unmarshal方法。定义xbean的方式和我们在协议里定义bean很类似。

范例：

<xbean name="Bag">

<variable name="money" type="long"/>

<variable name="capacity" type="int"/>

<variable name="nextid" type="int"/>

<variable name="items" type="map" key="int" value="Item" />

</xbean>

xbean内部除了variable节点之外不会包含其它任何节点。

属性说明：

name: 变量名。程序中会用

type: 变量类型

目前支持的数据类型。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名字 | 内部实现类型 | 装箱类型(1) | 访问接口 | 其他 |
| short | short | short | get/set | 可配置默认值 |
| int | int | Integer | get/set | 可配置默认值 |
| long | long | Long | get/set | 可配置默认值 |
| float | float | Float | get/set | 可配置默认值 |
| boolean | boolean | Boolean | get/set | 可配置默认值 |
| string | String | String | get/set | 可配置默认值 |
| binary | byte[] | 无 | get/set(2) |  |
| set | HashSet(3) | 无 | Set | 示例：  <variable name="items" type="set" value="Item"/> |
| list | LinkedList | 无 | List | 示例：  <variable name="items" type="list" value="Item"/> |
| vector | ArrayList | 无 | List | 示例：  <variable name="items" type="vector" value="Item"/> |
| map | HashMap | 无 | Map | 示例：  <variable name="items" type="map" key="int" value="Item"/>。  key必须是常变量。其中 Map.entrySet().remove(Object)操作不支持。 |
| treemap | TreeMap | 无 | NavigableMap |  |

(1) 装箱类型决定了这个类型能不能放到容器中，简单类型和自定义个xbean都可以放到容器中。

(2) 接口比较特殊，看生成出来的代码。

(3) 实际上是用 xdb.logs.SetX，这个实现和java.util.HashSet基本一样。

capacity: 不是强行的限制,程序运行的时候不会关心这个值。他仅仅是用来辅助预估内存占用量。(后面会详细解释这个)

default: 默认值。对于简单类型，可以设置一个默认值。

### any类型的xbean

有时候我们需要把一些特殊的对象也交给xdb管理，如网络连接、文件句柄、Future。这时候就需要把xbean设置成any类型。

<xbean name="fdinfo" any="true">

<variable name="file" type="vector" value="java.io.File"/>

</xbean>

any类型的xbean允许引用任何java类型。但是无回滚支持，且不能往硬盘上存储。

### xbean变量的capacity

主要用途：通过配置变长变量的最大容量，来分析和估算内存的使用状况。

xgen生成的时候有详细的WARN提示那些类型需要配置。基本规则如下：

已知固定长度的类型不需要配置，如int,short,long。需要配置的有

1. 未知类型以及包含固定类型的容器类型，如，any,string,binary。格式: capacity=”number”
2. 容器类型，如,list,vector,map。格式：capacity=”number[;key:keynumber;value:valuenumber]”。如果容器内包含的类型需要配置，就是用key:或value:进一步配置。

## xdb.xml

每个database由一个xml描述它。

这个文件在编译和启动的时候都需要。编译的时候需要根据它生成代码，启动的时候需要根据它来读取运行时参数。

下面是一个例子：

<xdb xgenOutput="src" trace="info" traceTo=":file" corePoolSize="100" dbhome="xdb" logpages="4096" backupDir="xbackup" checkpointPeriod="60000" backupIncPeriod="600000" backupFullPeriod="3600000" angelPeriod="5000">

<ProcedureConf executionTime="3000" maxExecutionTime="0" retryTimes="3" retryDelay="100"/>

<xbean name="Bag">

<variable name="money" type="long"/>

<variable name="capacity" type="int"/>

<variable name="nextid" type="int"/>

<variable name="items" type="vector" value="int" />

</xbean>

<table name="bag" key="long" value="Bag" cacheCapacity="10240" cachehigh="512" cachelow="256" lock="rolelock"/>

<TableSysConf name="\_sys\_" cacheCapacity="1" cachehigh="512" cachelow="256"/>

</xdb>

数据库的结构是静态的。程序在运行的时候无法动态的创建表和数据类型。

参照下面的表格可知：数据库将被放置在xdb目录下，数据库的备份将会被放在xbackup目录下，备份间隔是3600000毫秒一次。

每个database可以定义很多个table，很多个xbean。但是TableSysConf只有一个，里面放置一些元数据，一般不需要管。

根节点属性含义：

|  |  |
| --- | --- |
| 属性名 | 说明 |
| xgenOutput | 指定相对xgen的运行目录，不会自动创建 |
| 日志配置 | |
| trace | 设定错误输出级别，共有五个级别：DEBUG, INFO, WARN, ERROR, FATAL |
| traceTo | ":file"代表输出到文件，":out"代表输出到stdout |
| traceRotateHourOfDay | trace.log每天几点回滚。默认为6 |
| traceRotateMinute | 默认为30。即每天的6:30am回滚trace.log |
| 线程池配置 | |
| corePoolSize | 数据库默认线程池大小（即xdb.Executor的大小），主要用来执行协议的process方法 |
| procPoolSize | 数据库procedure线程池大小 xdb.Procedure的execute,submit进入事务时，使用这个执行器。 |
| schedPoolSize | 数据库scheduled线程池大小。这个线程池用来调度需要延迟或者定时执行的任务。 |
| timeoutPeriod | xdb.util.TimeoutManager多久执行一次。单位:毫秒,默认为2000，这个只是影响内部定时器的精度，一般无须修改。 |
| 存储配置 | |
| dbhome | 数据库目录。不会自动创建 |
| logpages | 数据库日志页面配置，单位是pages。用来控制logdir下单个日志文件的最大大小。参见db.h中GlobalLogger的构造函数 |
|  |  |
| CheckPoint配置 | |
| allowCheckpointXXX | 是否开启checkpoint,默认为true。修改此参数会导致危险，仅供测试。 |
| checkpointPeriod | checkpoint间隔，毫秒 |
| 备份配置 | |
| allowBackup | 是否开启备份功能。默认为true |
| backupDir | 备份目录。不会自动创建 |
| backupIncPeriod | （功能尚未开启）增量备份间隔，必须小于backupFullPeriod才有增量意义，否则这种备份不会执行。 |
| backupFullPeriod | 全备份间隔 |
| backupDelay | 默认为0 |
| 其它设置 | |
| angelPeriod | 死锁检测间隔，毫秒 |
| xdbVerify | 默认为false。只有这个选项打开，并且xgen生成代码的时候也生了verify代码，检查才有效。 |
| snapshotFatalTime | 默认为2000 |
| marshalPeriod | 默认为-1 |
| marshalN | 默认为1 |
|  |  |
| serialKey | 默认为1024 |

### ProcedureConf节点属性含义

|  |  |
| --- | --- |
| 属性名 | 说明 |
| executionTime | 不晓得有什么用 |
| maxExecutionTime | 最大执行时间 |
| retryTimes | 发生死锁时，重试几次 |
| retryDelay | 发生死锁时，延迟多久重试 |
| trace | 日志级别 |

## 生成代码

java -cp xdb.jar xgen.Main -outputEncoding utf-8 xdb.xml

## 启动xdb

//先载入数据定义文件

xdb.Xdb.getInstance().setConf(new xdb.XdbConf(“xdb.xml”));

//然后启动

xdb.Xdb.getInstance().start();

程序退出的时候，会自动调用xdb.Xdb.getInstance().stop()。也可在main函数的末尾手动调用。

## 更改现有的表定义

在开发的过程中经常需要添加新表或者给现有的表添字段。

如果尚未进入运营阶段，那么最简单的方式就是把全部清空重来。

如果已经开始运营，可使用snail/bin/xtransform.sh进行数据转换。

注意：允许添表、添字段、删字段。但是不允许对现有的字段改名。如果添字段和删字段发生在同一次更新中，那么新添的字段不允许使用要删除的字段的名字。

# 数据操作

这一章主要介绍如何通过xdb提供的函数读写数据。

## 从表读取并（或）修改数据

有两种方式。

方法一：存储过程。可读可改

处理数据最直接的方式写一个存储过程然后调用它。在存储过程内，每个table提供get/insert/delete方法。

示例：

//定义一个类，从xdb.Procedure继承，然后实现process方法。

class MyProc extends xdb.Procedure {

//return true代表commit这个存储过程。return false代表执行rollback

//存储过程只有这两种执行结果

boolean process(){

long roleid=4096L; //roleid可从protocol对象上通过Onlines类获得

xbean.Bag bag=xtable.Bag.get(roleid);

int money=bag.getMoney();

//....

bag.setMoney(bag.getMoney()+10);

bag.getItems().add(xbean.Pod.newItem());

return true;

}

};

...

//然后再这么使用。(比如在协议的处理函数里这么写)

MyProc proc=new MyProc();//此处可以把一些参数通过构造函数传递进去

proc.submit(); //向数据库提交这个存储过程

方法二：select。只能读

如果是在存储过程外，那么可以通过xtable.Bag.selectMoney(key)这样的方式根据key查找value。例如根据roleid去bag表查找这个人身上有多少钱。这种方式只能查询，不能修改或删除。

### 存储过程的调用方式

存储过程的调用方式有这么几种：

1. proc.call(): 在当前线程同步执行。如果当前线程正在执行某个存储过程，那么通过这个方法嵌套执行另一个存储过程，否则不推荐这么用。
2. proc.execute(): 提交到xdb的线程池中异步执行。调用者不关心proc的执行结果。
3. proc.submit(): 提交到xdb的线程池中异步执行。要求当前不能在存储过程中。与proc.execute()的不同是它会返回一个java的future。通过这个future对象可以等待proc执行完或获取proc的执行结果。

如果有多个存储过程被异步提交到线程池中执行，xdb未必会按照提交顺序执行。如果一定要有顺序，那么有两种方式：

1. 应用自己来保证。

顺序执行的例子：

proc1.submit().get();

proc2.submit().get();

那么proc2一定在proc1全部执行完之后才开始执行。

1. 使用特殊的线程池如SerialKeyExecutor,提交任务的时候需要添加一个参数用来排序。

### 存储过程的执行结果

存储过程只有两种执行结果，成功提交或回滚。

process函数最终return false，或者在执行的过程中抛出任何未捕获的异常，都会导致存储过程被回滚。那么对于表的所有的数据修改都会被回退。

## 向表插入新的数据

只有在存储过程之中才可以添加数据。

由于底层存储引擎的限制，任何table都不允许key重复。

按key的来源，table分两种，自增长的和非自增长的。

有两个接口可以向数据库中插入新的数据。xtable.add和xtable.insert。假设key不是自增长的，示例如下：

xbean.Bag bag=xbean.Pod.newBag();

xtable.Bag.insert(4096L,bag);

如果已经有key=4096L的记录存在，那么insert方法会抛出异常，而add方法会return false,这就是它们的区别。

假设key是自增长的，那么这个时候add/insert接口语义是一样的。并且，插入的时候不需要提供key.

xtable.Bag.insert(bag);

## 从表删除数据

只有在存储过程之中才可以删除数据。

有两个函数可以用来删除数据。xtable.remove和xtable.delete。如

xtable.Bag.remove( 4096L);

如果这个记录存在并且被成功删除，那么return true。否则，return false。但是，无论是否删除成功，它都会保持这条记录的锁，一直到存储过程结束。

## xbean的生命周期

xbean分两种，受xdb管理的和不受xdb管理的。下面分别讨论这种xbean的生命周期。

1. 受xdb管理的xbean

创建：

xbean.Bag bag=xbean.Pod.newBag();

通过xtable.get方法得到的xbean都是受xdb管理的xbean。以前面的例子来说

xbean.Bag bag=xtable.Bag.get(roleid);

那么bag以及bag.getItems()以及bag.getItems.get(0)得到的都是受xdb管理的xbean。

受xdb管理的xbean的生命周期不能超过存储过程的生命周期。永远不要存储过程外读写受xdb管理的xbean。

也就是说，不能这样：

class MyProc extends xdb.Procedure {

private xbean.Bag bag;

boolean process(){

long roleid=4096L;

bag=xtable.Bag.get(roleid);

return true;

}

public xbean.Bag getBag(){

return bag;

}

};

然后在存储过程外执行这样的语句：

MyProc proc=new MyProc();

proc.call();

proc.getBag();

因为存储过程执行完之后，这个xbean就无效了。但是上面三句如果在另一个存储过程内执行，那么是对的。

1. 不受xdb管理的xbean

创建：

xbean.Bag bag=xbean.Pod.newBagData();

或者xbean.toData()能把一个受xdb管理的xbean变成普通的java bean。这个java bean可以在存储过程外使用。

例如：

xbean.Bag bag=xtable.Bag.get(roleid);

xbean.Bag data=bag.toData();

不受xdb管理的xbean，无回滚支持。

select返回的xbean是不受xdb管理的xbean。

## 遍历一张硬盘表

想知道一个表有多少条记录？听起来很简单，但是做起来不那么容易，因为你需要把这个表遍历一遍。遍历一张硬盘表不需要在存储过程内执行。并且为了避免死锁，建议永远不要在存储过程内执行。

为了遍历一张硬盘表，可以传递一个回调（继承自TTable.IWalk）给TTable.walk或者TTable.browse方法。

示例代码如下：

xtable.Basic.*getTable*().browse(

**new** xdb.TTable.IWalk<Long, xbean.Basic>(){

@Override

**public** **boolean** onRecord(Long k, Basic v) {

System.out.Println(“roleid=”+k);

**return** **true**;

}

}

);

walk和browse的区别是：

首先，这两个函数都浏览不到xdb的cache中的脏页。

其次，这两个函数的区别是，walk方法能看到已经flush到db.h中的所有数据（这些数据不一定已经写入到硬盘上），但是browse方法跳过了db.h的缓存，只能看到做完checkpoint已经写入到硬盘上的数据，并且browse方法可以与checkpoint函数的flush过程并发执行而walk方法不行。browse的效率远高于walk方法，但是browse方法有很低的概率会产生误判，比如一个数据库本来有一万条记录但是用browse方法读出来了10100条。这两个方法的具体区别涉及到db.h的底层实现，所以这里不再细说。

除非你能把onRecord函数里面的逻辑非常简单并且表的行数不大，否则GS启动完毕，开始接受外部请求之后，最好不要再walk硬盘表。使用walk/browse方法最大的风险是程序的复杂度很容易跳出你的预想。

如果要运行时walk硬盘表并且获得最新的数据，也可以这样

假设shop是一张硬盘表，存储了所有的shop信息。那么再创建一个表，叫mirrorOfShop，key的类型和shop表相同，但是mirrorOfShop是内存表。

然后，启动的时候，遍历硬盘表，把所有的key插入到内存表中。

之后，所有对硬盘表的插入和删除操作，都对应着该内存表执行一次。

需要遍历的时候，遍历那个内存表。

## 遍历一张内存表（或硬盘表在xdb中的缓存）

遍历一张内存表（或硬盘表在xdb中的缓存）不需要在存储过程内执行。为了避免死锁，也永远不要在存储过程内执行。

每个Table都有一个getCache方法。因为内存表的所有数据都在Cache当中，所以可以这样遍历：

xtable.MyTable.getCache().walk(

new xdb.CacheQuery<Long, xbean.MyBean>() {

@Override

public void onQuery(Long k, xbean.MyBean value) {

System.out.Println(“roleid=”+k); }

}

);

walk内存表不存在硬盘表所面临的拿不到最新数据的问题。但是无论怎样，都看不到存储过程内部已修改但是尚未提交的数据。

逻辑应用一般不会去walk硬盘表在xdb中的缓存。

## 批量修改数据

禁止用xdb批量修改数据，即使用遍历表的方式修改大量数据。例如执行类似于”update mytable set colume1=10”这样的操作。

如果有这样的需求，使用dbx在GameServer未启动的时候做。

作为例外，如果这个表的数据量很小，那么可以在启动的时候用xdb做。

## 存储过程嵌套

假如在执行存储过程P1的时候需要执行另一个存储过程P2。那么现在首先考虑这么几个问题：

1. 在当前线程执行，还是另换一个线程执行

如果在当前线程执行，那么P1已经拿到的锁会继承给P2。只有在最外层的存储过程执行完之后才会释放所有的锁。（锁的问题在后面的章节具体讲）

例如：class P1 extends xdb.Procedure {

boolean process(){

…

Proc2 p2=new Proc2();

p2.call();

…

return true;

}

};

这就是平时通常所说的嵌套。由于p2在执行中所有的逻辑异常都会被xdb吃掉，对外只是成功或失败。所以可通过这样的方式，把逻辑操作封装成一个又一个的小单元。

如果不希望把现有的锁传递给它，那么就换到另一个线程执行。方法有很多种，例如：Procedure.execute(proc2)。这种方式概念上一般被认为是触发，而不是嵌套。

1. 同步执行，还是异步执行？

在当前线程执行p2.call()这样的方式是同步执行。只有当p2执行完之后，下面其它的代码才会被执行。

而Procedure.execute(proc2)这样的方式是异步执行。

如果要切换到另一个线程里执行，但是当前线程要等待P2执行完。那么可以往线程池扔一个Runnable实例，P1等待在这个Runnable的future上，另一个线程在这个Runnable的run方法里执行Proc2.call()。

1. 如果P1回滚了，P2需要回滚吗？

如果P1直接在当前线程执行p2.call()：

如果p2失败了，那么p2.call()会返回false,p1可以忽略这个返回值，也可以根据这个返回值执行其它操作。

如果p2成功了，但是p1在后面的逻辑执行中失败所以回滚，那么p2也会被回滚。

如果P2是被扔到另一个线程中执行，如果P1回滚了，无法直接影响到P2。

## 关于持久性/checkpoint

传统来说，一个存储过程只有把已修改的数据统统写入后备存储（如硬盘）后，存储过程才算已提交。（这里的写操作可能是写数据也可能是写日志或者二者都有）

xdb在修改数据的时候并非真的把新数据写入硬盘了，commit的时候也没有,xdb只有checkpoint线程在做checkpoint的时候才会写硬盘。在一个商业数据库中，一个存储过程修改几千万条记录也没有问题，但是我们不行，因为任何被修改的数据在做checkpoint前都不可能被写入硬盘。

## 超时保护

当用xdb默认的线程池执行协议或者在xdb的线程池里执行存储过程的时候，都是有超时保护的。意即，如果process方法执行的时间超过了时限，那么就会被打断。这个时间限制在xdb.xml中设置。但是，只有处于可被打断的状态的线程才会被打断，例如，while(true);这样的代码是无法被打断的，但是等锁的状态可被打断。什么状态可被打断具体参见java.lang.Thread.interrupt的文档。

# 锁

db.h是不考虑多线程问题的，没有锁。而Xdb是在多线程的环境下运行，所以它必须采用一定手段来达到隔离性。

## 隔离度

按照数据库的理论，隔离级别可以被分为以下四种，从上向下依次增强：

1. Read Uncommitted: 允许读到其它存储过程已修改但尚未提交的数据。
2. Read Committed：读到的数据一定是已经被提交的，但是可能会产生不可重复的读。Xdb的table的select方法提供的就是这样的隔离级别。对于给定的key,在同一个存储过程中两次调用select得到的可能是不同的值。
3. Repeatable Read：在同一个存储过程中用相同的条件查找，得到的永远是相同的值。但是有可能会产生幻影。但是对于key-value的数据库，一般不考虑这个。
4. Serializable：事务的执行是可串行化的。抛开select方法不谈，xdb采用的就是这样的隔离级别。

## 实现模型

　　Xdb采用基于锁的方式实现事务的隔离型。此处主要考虑两种锁，表锁和记录锁。应用只需要考虑记录锁。

每个table有一个属性叫lockname，是一个字符串。默认情况下lockname就是table name。每条record的锁(xdb.Lockey)由lockname+key组成。如果这两个都相同，那么就是同一把锁。

存储过程在执行get/insert(add)/delete(remove)这些方法的时候，会首先获取这条记录对应的锁，然后一直维持到存储过程结束。特别强调的是remove方法即使失败，也会一直保持锁。

xtable.Bag.selectMoney这样的东西既可在存储过程外调用也可在存储过程内调用，它会首先获取锁，然后取数据，然后立刻释放，然后返回。它与get方法的区别是它不保持锁，所以这样的语句一定不会导致死锁。

在存储过程内可以通过this.lock(Lockey[])的方式显式的加锁。这些锁一直被保持到存储过程执行完。注意：只有lock方法，没有unlock。永远没有办法手动的释放一把锁。lock方法一般是在存储过程一开始的时候执行。

写应用逻辑的时候脑袋必须很清楚，我现在拥有哪些锁，在提交之前，可能还会获取哪些锁？

xdb获取记录锁时，永远都是以互斥的方式获取（不区分读写）。

## 死锁怎么发生？

存储过程P1，在已经拿到锁A的情况下，请求锁B。

此时存储过程P2，在已经拿到锁B的情况下，请求锁A。

它们便会陷入相互的无限等待中。

例如下面这段代码，几乎必然会死锁：

public synchronized void doDeadLock() {

final java.util.concurrent.CyclicBarrier cb=new java.util.concurrent.CyclicBarrier(2);

new xdb.Procedure() {

@Override

protected boolean process() throws Exception {

xtable.Basic.get(4096L);

cb.await(10, java.util.concurrent.TimeUnit.SECONDS);

xtable.Basic.get(8192L);

return true;

}

}.submit();

new xdb.Procedure() {

@Override

protected boolean process() throws Exception {

xtable.Basic.get(8192L);

cb.await(10, java.util.concurrent.TimeUnit.SECONDS);

xtable.Basic.get(4096L);

return true;

}

}.submit();

}

线程A去访问Basic这个表的key=4096的这条记录。然后故意等10秒。

线程B去访问Basic这个表的key=8192的这条记录。然后故意等10秒。

然后线程A去访问8192这条记录（它的锁已经被B持有）。线程B去访问4096这条记录（它的锁已经被A持有）。就会陷入相互等待。

这里举的例子是死锁最简单的例子。因为这里都是互斥锁。当涉及读写锁、条件变量时，问题就复杂的多了。那些复杂的问题此处不考虑。

## 发生死锁怎么办？

xdb有一个线程，负责自动检测死锁，发现并打断它。在xdb的trace.log中有死锁打断日志。（可以拿我上面的代码做个实验）

被打断的存储过程能够被自动重做。

## 怎么避免死锁

所有人以同样的顺序获取锁！

看这么三种场景：

A：某人使用一个物品，给他自己增加1000经验。

此时需要访问两个表，Bag和Properties。key都是roleid。如果这两个表的lockname也一样，那么就太完美了。从头到尾只需要一个锁，所以它永远不会和任何人死锁。于是我们把所有单人数据的表，lockname设计成rolelock。

B：某玩家给予另外一个玩家1000游戏币

游戏币的数据存储在Bag表内。此时需要访问Bag这张表的key=roleid1和key=roleid2两条记录。

方法是：存储过程的一开始，先把这两个人的roleid排序，然后依次访问Bag表。由于在交易中访问到的所有表，lockname都等于rolelock,所以只要大家都保持这样的原则，那么永远不会有死锁。

C：某人申请加入一个队伍

首先有一个表叫做team。每条记录对应着每个队伍。lockname=team

然后还有一个表叫做roleid2team。key是roleid,value是team表的key。lockname=rolelock。

我们称team lock是大锁，rolelock是小锁。那么现在就有两种策略，先大后小或者先小后大。

先大后小：已获得小锁(rolelock)的情况下不得申请大锁(teamlock)。

先小后大：已获得大锁(teamlock)的情况下不得申请小锁(rolelock)。

以先小后大为例：

例如，A向B申请加入队伍，那么通过协议和协议的上下文很容易知道A、B的roleid。在process函数刚进入的时候，就手动锁定A和B的rolelock，然后再访问team表。

但某些事情会变得棘手。比如A说他要解散队伍。

## 在死锁的问题上，我们常常做不到最完美

继续说，比如A说他要解散队伍。按照最普通的逻辑，是这样：根据roleid到roleid2team查到teamid,然后根据teamid到team表查到当前有那些人，并删除这条记录。然后根据查到的teamid去roleid2team表挨个删除。可是，这么做违反了我们之前定下的加锁原则。

假设我们采用的是先小后大的加锁原则。（即先锁rolelock，后锁team表）

解决方案1：

写成２个存储过程。第一个存储过程根据roleid查到当前队伍所有人的roleid,然后执行第二个存储过程：

this.lock(roleids,“rolelock”);

final long leaderid=roleids[0];//人员列表中第一个就是队长

final long teamid= xtable.Role2team.get(leaderid);

xbean.Team team=xtable.Team.remove(teamid);

for (final long roleid : team.getRoles())

xtable.Role2team.remove(roleid);

可是，如果这两个存储过程之间，新添一个人会怎样？对于上面的代码，就会造成少锁了一个rolelock，就意味着可能会引发死锁。

解决方案２：

在第二个存储过程中，检查传进来的roleid和team表当前存的是否一致。如果不一致，则rollback。也就是说解散操作偶然会失败。

final long teamid= xtable.Role2team.get(leaderid);

xbean.Team team=xtable.Team.remove(teamid);

if(team.getRoles().size()!=roleids.length 　 || !team.getRoles().containsAll(roleids))

return false; //发现不一致，立即回滚。

for (final long roleid : team.getRoles())

xtable.Role2team.remove(roleid);

解决方案３：

select的好处是，可以让上面的两个存储过程写成一个。先用select方法去查teamid和roleids。然后下面用get/remove方法读写。但是这么做只是让代码看起来更紧凑更简洁一些，前面说的风险依然存在。

两次读到的东西不一致，就算发现了，又该怎么办？上面解决方案1的代码忽略这种不一致，它带来的是潜在的死锁。而其它有时候带来的可能是表与表之间的引用关系被破坏。例如根据roleid查到一个teamid,然后这个teamid在team表根本就不存在。

解决方案2的代码不能被用在关键场合。例如，我们需要用定时器在7月7日的某个时间点开启一项庆典活动，但是这个开启操作被“偶然性失败”了。这个可被接受吗？（这是个疑问句不是反问句）

或者我们把加锁的策略改成先大后小呢？

# 触发器

触发器的本意不是用来写游戏逻辑的。是用来在服务器和客户端之间同步数据的。

（可以完全不用这个功能。触发器的具体介绍待续）

## 注册触发器

每个表都有一个addListener/removeListener方法。

使用方法例如：

xtable.Bag.getTable().addListener(new BagChanged(), "value", "items");

其中第一个参数是某个实现了xdb.logs.Listener接口的类的实例。

## 触发器的执行时刻

触发器在存储过程成功commit之后执行。

## 触发器的粒度

这里是指事件的组成。例如在一个存储过程中多次修改了同一行数据，那么是触发一次还是触发多次呢？1次。在一个存储过程中修改了多行数据，那么是触发一次还是触发多次呢？每行1次。

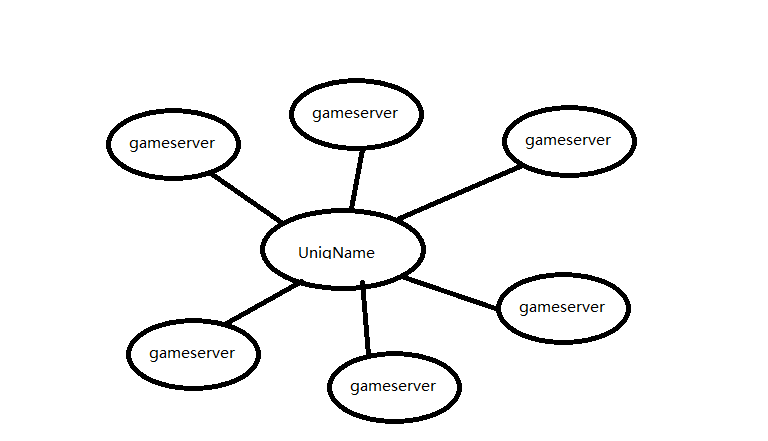
## 避免链式反应

在触发器执行中出现永远没有结束的链式反应是一个十分严重的问题。最稳妥的做法就是触发器的事件代码一定在存储过程之外并且执行中永远不要调用任何存储过程。

# 唯一名

唯一名主要是为以后数据库合并（合服）设计的。唯一名用来保证特定的数字（如角色id）或字符串（如角色名）在某些服务器是唯一的。

Uniqname分两部分：服务器和客户端。服务器端需要作为一个单独的进程启动，客户端内嵌在xdb中。



gameserver只有归同一个uniqname管理才能合并。

gameserver即便不考虑合服的问题，只要xdb任何一个表用到了自增长属性，那么就必须在xdb.xml中配置唯一名。

## 游戏服务器配置示例

在gameserver的xdb的配置文件中加入以下内容（与table节点同级）：

<UniqNameConf localId="0">

<XioConf name="xdb.util.UniqName">

<Manager name="Client" keepSize="1" maxSize="1">

<Coder>

<Rpc class="xdb.util.UniqName$Allocate"/>

<Rpc class="xdb.util.UniqName$Confirm"/>

<Rpc class="xdb.util.UniqName$Release"/>

<Rpc class="xdb.util.UniqName$Exist"/>

<Rpc class="xdb.util.UniqName$AllocateId"/>

<Rpc class="xdb.util.UniqName$ReleaseId"/>

</Coder>

<Connector remoteIp="172.16.32.45" remotePort="29024" sendBufferSize="16384" receiveBufferSize="16384" tcpNoDelay="true" inputBufferSize="131072" outputBufferSize="131072"/>

</Manager>

</XioConf>

</UniqNameConf>

其中Connector节点需要配置Uniqname服务器的IP和端口号。xdb在启动的时候会去连接这个服务器。对于Uniqname服务器管辖的每个client，都必须用一个唯一的整数来标识它（0-4095），这个整数就是UniqNameConf节点的localid属性。

其它的设置不需要改，照抄即可。

### localid分配规则

切记：localid不可重复，并且不可重用。例如：服务器A的localid是0，服务器B的localid是1，然后A和B合并成一个新的服务器，那么这个新服的localid可以是0，也可以是1，也可以是任何一个尚未使用的数。不可重用的意思是说以后如果要开新服，新服的localid即不能是0，也不能是1。

## 功能说明

唯一名主要涉及三个功能：给自增长的表分配key、在存储过程中通过rpc请求向唯一名服务器申请名字（字符串）或申请ID（整数）。

### 分配自增长ID

如果一个表是自增长的，那么它的key的类型必须是long。它的第一条记录的key应该是4096L+localid。第二条记录应该是4096\*2L+localid，以此类推。这样就可以保证在同一个uniqname管辖的范围内，所有自增长的表在合并的时候不会发生冲突。

这个分配过程不需要访问网络。只需要在设计表的时候配置自增长属性即可，不需要写什么额外代码。

### 分配名字

要向uniqname申请名字必须在存储过程中执行。在存储过程中，可以通过xdb.util.UniqName.allocate(String, String)这个静态方法向uniqname申请一个名字。其中第一个参数是名字所在分组（即名字空间）。如："role"(角色名)，"family"(家族名)，"faction"(帮派名)。第二个参数是所要申请的名字。这个函数返回true，代表分配成功，返回false代表分配失败。名字空间是在编译uniqname的时候在uniqname.xdb.xml指定。

因为这个分配操作需要访问网络，所以必须在xdb已启动网络的状态下执行。

如果执行allocate方法的这个存储过程最终没有commit，而是rollback了。那么已分配的名字会通过RPC请求被释放。

### 分配ID

在这样的情况下需要通过uniqname分配ID：

1. 这个ID需要是全服唯一的。因为合服后也需要唯一，所以在uniqname管辖的范围内必须是唯一的。
2. 这个ID生成的规则是特殊的。不是4096\*n+localid这样的格式。
3. 合服时这个ID不能变。

参见xdb.util.UniqName.allocateId(String)和xdb.util.UniqName.allocateId(String, long)这两个接口。

（待续）

## 编译唯一名

唯一名服务器现在不是作为一个编译好的jar包发布。需要各个项目组自行单独编译。如果你不是从头搭建一个项目，那么这一节先忽略。

### 获取发布版本

发布目录

http://172.16.10.57/repos/snail/bin

文件列表

uniqname.jar 可运行发布版本（二进制）

uniqname.xdb.xml 唯一名服务器初始配置，初次安装可以拷贝这个配置当作初始配置，更新时忽略她。

uniqname.install.xml 唯一名服务器更新脚本。

uniqname.class 安装程序。 （下面用不到）

### 设置编译脚本

因为现在的项目普遍以ant作为编译脚本。所以下面以ant为例讲如何编译唯一名。

首先，在自己的项目目录下创建一个空目录,叫uniq

然后，在build.xml中加入：

<!-- 相对于本build.xml -->

<property name="snail.bin" value="../snail/bin"/>

<!-- 相对于本build.xml -->

<property name="uniqname.dir" value="uniq"/>

<target name="uniqname">

<!-- 拷贝重新打包脚本 -->

<copy overwrite="true" file="${snail.bin}/uniqname.install.xml" todir="${uniqname.dir}"/>

<ant dir="${uniqname.dir}" antfile="uniqname.install.xml" target="install">

<!--新的uniqname.jar的生成目录 -->

<property name="output.dir" value="."/>

<property name="snail.bin" value="../${snail.bin}"/>

<property name="xdb.xml" value="../uniqname.xdb.xml"/>

</ant>

<delete file="${uniqname.dir}/uniqname.install.xml"/>

</target>

### 配置

把snail/bin/uniqname.xdb.xml复制到自己的项目的目录下，并按需求修改

1.如果要增加名字组，那么添加这样的table

<table name="role" key="string" value="NameState" cacheCapacity="10240" cachehigh="128" cachelow="64"/>

2. 如果要增加Id组(不推荐使用) ，那么添加这样的table

<table name="familyid" idmin="10000" idmax="1000000" key="long" value="IdState" cacheCapacity="10240" cachehigh="128" cachelow="64"/>

### 生成

ant uniqname

生好的jar包在uniq目录下。

# 其它

一些杂七杂八的东西，我想不到该归在哪一类中。

## 使用存储过程简化逻辑代码

通常来说，一个游戏操作包含这么几步：检查条件->执行操作->通知客户端。

设想需要完成这样一个游戏功能：

使用一个药品，这个药品的功能是把hp补满。

在没有事务保障的情况下，可能会这么写：

if(包裹里有这个物品){

if(hp满) return ;

把hp补满;

删除物品;

}

使用了xdb后，可以把第一行条件判断删除，不用判断包裹里是否有这个物品。

执行操作->扣除物品

扣除物品如果失败，回滚即可。

## GS的线程模型

(不同的游戏不一样)

GS是单进程，大概有这些线程:

(1) main：主线程。执行初始化操作，然后等待一个条件变量，被唤醒后执行关服逻辑。

(2) xio.Engine：维持到link的tcp连接。unmarshal收到的每个Protocol，并扔到xdb的线程池执行。

(3) xdb线程池：一共三个线程池，执行协议和存储过程的process方法等等。

(4) map: 以单线程方式顺序处理地图相关请求。

(5) xdb管理线程：负责写硬盘的checkpoint线程、负责检测死锁的等等。

Protocol的.process通常只有一行，就是new XXXXProc(this).submit()。把自己传递到某个存储过程的构造函数中，然后提交给xdb执行。一般来说，给客户端回协议的代码，会直接写在存储过程中，并在当前的worker线程直接发送。总的来说，一条协议被GS收到，然后到全部处理完毕，一般经过了3个线程：IO、某个worker线程执行process、某个worker线程执行procedure。

## 协议的顺序

先发送的协议一定先被执行吗？先提交的存储过程一定比后提交的先被执行吗？按上面的模型,不! 因为它被切换到另一个线程中执行，而那个线程不一定什么时候才能拿到CPU。

在开发网络应用时，经常会听到“协议顺序的问题”，这个问题的简单描述就是需要保证协议的执行顺序。 比如服务器先后收到两个协议 Open，Close，在并发的执行时，可能 Close 先被执行并完成，此时 最后的状态是打开的。对于客户端发送者来说，得到的最终结果就不是他所期望的。即使 Close 执行时 需要作必要的判断，会发现本来就是关闭的，并报告了错误，此时客户端能得到错误提示，但已经于事无补了。

协议顺序问题的本质是协议设计的不好，正确的解决方法在协议层解决，如增加协议，OpenResult, CloseResult。 客户端总是都在得到请求结果以后才发后续的请求。很显然，这样自然就没有协议顺序问题了，这是正确的解决方法。如果嫌这么做很麻烦，可以考虑换用xdb.util.SerialKeyExecutor作为执行协议的executor。

另一个问题是，先发出去的协议，客户端一定能先收到吗？简单来说，是的。例如在同一个存储过程里用psend发协议，那么先发的一定比后发的先到。

简单来说，对服务器而言，接收是无序的，而发送是有序的。

## 定时任务

假如策划有这样一个需求：

玩家在游戏世界里时，每分钟恢复1%的活力。不在线不恢复。

对程序而言，就是要按固定频率执行某个任务，并且可手动取消这个任务（下线时需要取消）。

xdb的线程池支持定时任务。所以上面一个问题的最佳实现方式就是扔一个定时任务到xdb的线程池里，然后把返回的future存下来（例如保存在内存表里）。需要取消的时候调用future.cancel()。

如：

java.util.concurrent.ScheduledFuture<?> future=xdb.Xdb.executor().scheduleAtFixedRate(new GameTimeTask(), 0,1, TimeUnit.SECONDS);

第一次执行从0秒后开始，之后每隔一秒，执行一次GameTimeTask.

## 随机数

程序中需要用到随机数的地方，请一律用xdb.Xdb.random()获得Random实例。不要自己设置随机数种子。

## 在存储过程中发送协议以及执行其它任务

变量的修改可以被回滚，但是网络包呢？已经发出去的网络消息能被回滚吗？不能。所以xdb后来提供了psend方法。

首先需要应用实现一个类，继承自xdb.Procedure.IOnlines接口。然后通过xdb.Procedure.setOlines()设置进去。

Procedure类提供了三组发包函数：

psend：在存储过程执行完时一定发送。无视存储过程是否执行成功。

psendWhileCommit： 在存储过程成功提交后发送。即成功才发送。

psendWhileRollback：在存储过程被回滚时发送。即失败才发送。

另外还有一个方法：

ppost: 添加任务到列表中，当存储过程结束时执行，而不管存储过程执行的结果。

## xdb.inuse文件

如果启动的时候发现dbhome目录下有xdb.inuse文件在，那么说明上次使用xdb的时候，程序未能正常退出（可能是被kill -9杀死的）。

手动把这个文件删除就能起来了。

## xdb是数据库吗？

你可以认为xdb是数据库，也可以认为它不是数据库。

我倾向于认为它不是一个数据库。

XDB是一个Trasactional Cache。并且xdb的代码中不仅包括数据访问的代码，还包括网络库XIO，还包括应用协议如UniqName。