

AsiaInfo **项目管理文档**

**高性能容器(XC)**

**开发手册**

|  |  |  |  |
| --- | --- | --- | --- |
| 编写 | 樊潇俊 | 编写时间 | 2011-6-25 |
| 审批 | 审批者姓名（及其职务） | 审批时间 |  |
| 版本 | V3.2.0 | | |

**亚信科技（中国）有限公司版权所有**

文档中的全部内容属亚信科技（中国）有限公司所有，

未经允许，不可全部或部分发表、复制、使用于任何目的。

**文档修订摘要**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **日期** | **修订号** | **描述** | **著者** | **审阅者** |
| 2011-06-25 | 无 | 新建文档 | 樊潇俊 |  |
| 2011-06-27 | 1.1.0 | 1.修改CEditHolder写法  2.修改index为multi\_index | 樊潇俊 |  |
| 2011-06-30 | 2.0.0 | 1.增加容器组概念  2.增加容器组快照 | 樊潇俊 |  |
| 2011-07-05 | 2.0.1 | 1.修改CTransaction事务是否全组更新为false | 樊潇俊 |  |
| 2011-07-13 | 2.1.0 | 1.增加容器组空间名  2.修改默认路径为$(HOME)/ipc/xc  3.增加Detach接口  4.增加容器组空间名约定  5.增加系统是否启动校验 | 樊潇俊 |  |
| 2011-07-20 | 2.1.8 | 1.增加快照返回容器生效时间  2.快照支持当成const传入 | 樊潇俊 |  |
| 2011-07-22 | 2.1.9 | 1.增加事务中Reopen接口,对已关闭的事务重新打开  2.修改事务原有接口名IsBeginTrans为IsOpen | 樊潇俊 |  |
| 2011-07-27 | 2.2.0 | 1.增加两个api  double GetAllMemSize();  double GetFreeMemSize(); | 樊潇俊 |  |
| 2011-10-10 | 2.2.9 | 1.增加数据装载代码自动生成说明 | 樊潇俊 |  |
| 2011-10-17 | 2.3.0 | 1.增加自动装载代码中容器类型配置  2.增加自动装载代码中Load接口 | 樊潇俊 |  |
| 2011-10-25 | 2.3.1 | 1.增加自动装载代码中Key类的生成 | 樊潇俊 |  |
| 2011-10-27 | 2.3.3 | 1.增加容器类型说明 | 樊潇俊 |  |
| 2011-10-31 | 2.3.4 | 1.增加QA章节 | 樊潇俊 |  |
| 2011-12-20 | 2.4.0 | 1.增加前台刷新配置 | 樊潇俊 |  |
| 2012-3-22 | 2.6.0 | 1.增加HASH\_MAP  2.增加HASH\_MULTIMAP | 樊潇俊 |  |
| 2012-6-25 | 2.6.8 | 1、使用预置用户列表更新或修改表名  2、xc2code添加配置文件指定连接支持。 | 张帆 |  |
| 2013-2-28 | 3.0.0 | 1. 增加进程异常CORE时,释放未提交的页内存 | 樊潇俊 |  |
| 2013-4-22 | 3.1.0 | 1.增加自定义类型自动生成 | 樊潇俊 |  |
| 2013-8-16 | 3.2.0 | 1.增加mysql数据源支持 | 樊潇俊 |  |

**目录**

第1章 引言 2

1.1 文档用途 2

1.2 阅读对象 2

1.3 名词术语 2

1.4 参考资料 2

第2章 概述 3

2.1 什么是XC？ 3

2.2 XC的功能定义 4

2.3 XC的使用场景 4

第3章 如何用XC开发应用？ 5

3.1 总述 5

3.1.1 空间名 5

3.1.2 依赖 5

3.1.2.1 头文件 5

3.1.2.2 库 5

3.2 框架开发流程 6

3.2.1 接口说明 6

3.2.2 开发说明 7

3.3 装载模块开发流程 7

3.3.1 接口说明 7

3.3.1.1 CEditHolder 7

3.3.1.2 CTransaction 7

3.3.2 开发说明 8

3.3.2.1 创建多个容器，在一个事务中提交生效 8

3.4 业务模块开发流程 10

3.4.1 接口说明 10

3.4.1.1 CSnapshot 10

3.4.1.2 CQueryHolder 10

3.4.2 开发说明 10

3.4.2.1 从容器中查询数据 10

第4章 STL标准容器介绍 12

4.1 仿函数介绍 12

第5章 扩展容器介绍 13

5.1 multi\_index 13

5.1.1 接口说明 13

5.1.1.1 multi\_index 13

5.1.1.2 multi\_index::iterator 14

5.1.1.3 multi\_index::value\_iterator 14

5.1.2 开发说明 14

5.2 match 17

5.2.1 接口说明 17

5.2.1.1 match 17

5.2.1.2 match::iterator 18

5.2.1.3 match::value\_iterator 18

5.2.2 开发说明 19

第6章 XC控制台使用 21

6.1 启动参数 21

6.2 功能 22

6.3 备注 23

第7章 组空间名约定 24

第8章 数据装载代码自动生成 25

8.1 配置文件说明 26

8.1.1 配置文件schema 26

8.1.2 配置文件例子 26

8.1.3 sql关联表配置 28

8.1.4 sql运行时变量 28

8.1.5 xcd插入自定义代码 29

8.1.6 自定义类型配置 29

8.1.7 容器类型配置 30

8.2 调用xc2code生成代码 31

sqlite dump文件名说明 33

8.2.1 生成代码名 33

8.2.2 生成代码空间名为 33

8.2.3 生成代码结构 33

8.2.4 自动生成接口 34

8.2.4.1 Load4Db:从数据库装载数据到XC 34

8.2.4.2 Load4Sqlite:从指定Sqlite文件中装载数据到XC 34

8.2.4.3 Load: 装载数据到XC(通用接口，这个接口会先调用Load4Db，如果不成功，那么再调用Load4Sqlite) 34

8.2.5 例子 35

8.2.5.1 从数据库装载数据 35

8.2.5.2 从sqlite的dump文件中装载数据 36

8.2.5.3 从xc中查询数据 37

8.3 获取sqlite dump文件路径 38

8.3.1 通过xc\_console获取 38

8.3.2 通过xc的api获取 39

8.4 通过xc\_console刷新数据 39

第9章 MYSQL支持 40

9.1 xcd编写支持mysql标签 40

9.2 xcd中宏支持 41

9.3 xc2code命令行 42

9.4 修改xc2code.cfg 43

第10章 JAVA缓存支持 44

10.1 描述 44

10.2 XCD编写 44

10.3 编译 44

10.4 模板 45

10.5 java客户端实用方法 46

第11章 通用前台刷新TS服务 47

11.1 配置说明 47

11.2 DBM配置图解 47

第12章 使用技巧 51

12.1 如何查看xc中加载的数据 51

第13章 Q&A 52

# 引言

## 文档用途

此文档是对XC（extreme container，高性能容器）的开发手册

## 阅读对象

此文档适合下列人员阅读：

* 产品设计工程师
* 产品开发工程师
* 产品测试工程师
* 现场维护工程师

## 名词术语

|  |  |
| --- | --- |
| XC | extreme container，高性能容器  XC是把原有进程私有的静态的内存数据，通过统一的容器的方式，存储到共享内存中，减少每个进程对内存的要求，并支持高性能防问要求。 |
| 容器组 | 一个XC中允许有多个容器组存在，一个容器组中有多个容器，系统可以对一个容器组做快照，保证一个容器组内的多个容器在同一快照下的数据版本是一致的，容器组名必须有组空间名(用于保证容器组名不冲突)，例如（jf::group1），组空间名约定看[组空间名约定](#_组空间名约定) |
| 组快照 | 同一快照下，保证同一容器组中多个容器版本是一致的 |

## 参考资料

# 概述

## 什么是XC？

XC是extreme container（高性能容器）的缩写。

XC是把原有进程私有的静态的内存数据，通过统一的容器的方式，存储到共享内存中，减少减少每个进程对内存的要求，并支持高性能访问要求。



## XC的功能定义

XC应该具备这样的功能集合：

1. 提供事务控制
   1. 对新增，更新容器组支持事务
   2. 同一事务支持多个容器同时提交或则回退
   3. 对同一容器做多版本控制，保证更新时不会影响原进程的读取（更新数据不需要重启进程）
   4. 支持事务的异常恢复(更新进程更新过程中出现异常，下次更新保证这些异常可以被处理)
2. 提供读取权限控制
   1. 对于业务进程，支持对容器的防问设置只读取权限，保证容器的安全，避免因为业务代码越界等原因，修改了容器的内存地址空间，造成整个容器不可用。
   2. 对于装载进程，支持对容器的防问设置读写权限
3. 支持下列容器

|  |  |
| --- | --- |
| STL标准 | 扩展 |
| string | match<T,CmpFunc> |
| list<T> | index<KEY,T, CmpFunc> |
| vector<T> |  |
| map<KEY,T> |  |
| multimap<KEY,T> |  |
| set<KEY> |  |
| multiset<KEY> |  |
| hash\_map<KEY,T> |  |
| hash\_multimap<KEY,T> |  |

1. 提供管理控制台
2. 支持容器组
3. 支持容器组快照
4. 容器组名，与容器名，都是大小写敏感的

## XC的使用场景

* 以读取为主
* 刷新频率少
* 事先知道需要的内存大小
* 数据初始化，在业务进程启动前完成

# 如何用XC开发应用？

## 总述

1. XC接口通过抛异常的方式返回错误
2. 初始化与关闭动作由框架完成
3. XC加载类是线程不安全的,快照类是线程安全的(多线程同时操作不用加锁处理)

### 空间名

高性能容器所有对外接口都在xc开头的空间名下面

### 依赖

#### 头文件

路径：$(OB\_REL)/include/public/xc

文件：#include “xc.h”

#### 库

USER\_LIBRARIES = public\_xc$(BUILDTYPE)

## 框架开发流程

### 接口说明

|  |  |
| --- | --- |
| 方法名 | 说明 |
| void Startup(  int32 iAllSemNum,  int32 iPerSemSetMaxNum,  int32 iChunkNum,  int32 iPageNum,  const char \* pIpcKeyFilePath=” /tmp/xc\_ipcskey/”,  const char \* pSqliteDumpPath = XC\_DEFAULT\_SQLITE\_DUMP\_PATH)  throw (XC\_EXCEPTION); | 系统初始化(启动完内存状态默认为只读)  iAllSemNum 系统信号量个数(信号量个数必须大于系统中的容器组数)  iPerSemSetMaxNum系统中允许每一个信号集中信号量的最大个数  iChunkNum系统内存块个数 (每个块大小=iPageNum\*XC\_PAGE\_SIZE) (系统总的空间大小=iChunkNum\*iPageNum\*XC\_PAGE\_SIZE)  iPageNum系统中每个内存块包含的页个数  pIpcKeyFilePath系统中IPC key文件路径（默认为$(HOME)/ipc/xc）  pSqliteDumpPath 系统sqlite dump文件路径（默认为$(HOME)/xc/sqlite\_dump） |
| void Shutdown(); | 系统关闭(释放相关资源) |
| void Attach(  const char \* pIpcKeyFilePath = “/tmp/xc\_ipcskey/”)  throw (XC\_EXCEPTION); | 系统attach(退出时不需要调用Shutdown)  pIpcKeyFilePath 系统中IPC key文件路径（默认为$(HOME)/ipc/xc） |
| void Detach(); | 系统detach(对应attach) |
| bool IsReadOnly(); | 判断系统是否为只读状态 |
| void SetReadOnly() throw (XC\_EXCEPTION); | 设置内存块为只读状态(业务进程中必须设置，避免因为业务进程异常导致整个内存块不可用) |
| void SetReadWrite() throw (XC\_EXCEPTION); | 设置内存块为读写状态(数据装载进程与数据更新进程必须设置) |
| void State(const char \* pInfo, FILE \* fd = stdout); | 返回当前系统内存与信号量使用信息  pInfo 需要增加的附加信息  fd 输出文件句柄，默认屏幕输出 |
| void State2File(); | 输出当前系统内存与信号量使用信息到文件  (可以通过XC\_STATE\_OUT\_PATH环境变量来指定输出路径) |
| int32 GetMemIdleRate(); | 获取系统内存空闲比率(返回百分比，最大不超过100) |
| double GetAllMemSize(); | 获取总内存大小(单位GB) |
| AISTD string GetSqliteDumpPath(); | 获取sqlite dump path路径信息 |
| double GetFreeMemSize(); | 获取空闲内存大小(单位GB) |
| CContainerGroupStlMapGetAllContainer(); | 返回系统中所有容器信息 |
| AISTD string GetVersion(); | 获取版本信息 |
| void SetPassword(AISTD string strNewPassword) throw (XC\_EXCEPTION); | 设置密码  strNewPassword 新密码 |
| bool CheckPasswork(AISTD string strPassword); | 检查输入的密码是否正确  strPassword 输入的密码 |
| AISTD string GetRevision(); | 获取SVN版本信息 |

### 开发说明

* 框架在启动时调用xc::Startup完成xc的共享内存创建，内部初始化等动作
* 框架在退出时需要调用xc::Shutdown完成xc的资源释放

PS:业务程序不需要在退出时调用xc::Shutdown

## 装载模块开发流程

装载模块指的是那些需要定时或则手工更新容器内部数据的模块

### 接口说明

#### CEditHolder

|  |  |
| --- | --- |
| 方法名 | 说明 |
| template<typename \_Tp>  CEditHolder(CTransaction & cTrans, const char \* pName) throw (XC\_EXCEPTION) | 构造一个待编辑的容器holder  cTrans 事务  pName 需要使用的容器名 |
| \_Tp & GetContainer() throw (XC\_EXCEPTION); | 获取这个容器的引用 |
| void Release(); | 手动释放这个容器的引用 |
| bool IsValid(); | 判断这个容器引用是否有效 |
| const char \* GetName(); | 获取holder的容器名 |
| ~CEditHolder() | 自动释放引用 |

#### CTransaction

|  |  |
| --- | --- |
| 方法名 | 说明 |
| CTransaction(const char \* pGroupName,  bool bAllGroupUpdate = false,  int32 iIntervalSleepTime = 10); | 创建更新事务  pGroupName 容器组名  bAllGroupUpdate 是否全组更新(默认为false)  iIntervalSleepTime 更新等待间隔时间(默认10秒) |
| AISTD string GetGroupName() | 获取事务名 |
| bool IsOpen() | 事务是否已开始 |
| bool IsValid(); | 判断这个容器引用是否有效 |
| void Reopen(const char \* pGroupName,  bool bAllGroupUpdate = false,  int32 iIntervalSleepTime = 10); | 重新打开事务  pGroupName 容器组名  bAllGroupUpdate 是否全组更新(默认为false)  iIntervalSleepTime 更新等待间隔时间(默认10秒) |
| void Rollback(); | 回退这次更新事务 |
| void Commit(); | 提交这次更新事务  (因为更新过程会是一个阻塞动作，所以更新进程调用本函数时可以使用线程机制去完成) |
| ~CTransaction() | 构析时如果更新未提交，自动调用Rollback操作 |

### 开发说明

* 一个事务可以包含多个容器的更新与创建
* 系统同一时间对同一容器只允许一个更新事务
* 系统支持更新异常时，下次自动恢复
* 一个事务支持同时对多个容器进行更新，并一次提交或则回退

#### 创建多个容器，在一个事务中提交生效

|  |
| --- |
| typedef xc::list<int> CXcIntList;  //设置为读写模式  xc::SetReadWrite();  //创建一个事务名为trans1的事务  xc::CTransaction cTrans (“demo::group”);  //更新或则创建容器list1  xc::CEditHolder<CXcIntList> cEditHolder1( cTrans, “list1”);  //调用cEditHolder1插入数据  cEditHolder1.GetContainer().push\_back(1);  //更新或则创建容器list2  xc::CEditHolder<CXcIntList> cEditHolder1( cTrans, “list2”);  //调用cEditHolder2插入数据  cEditHolder2.GetContainer().push\_back(1);  //提交事务  cTrans.Commit(); |

## 业务模块开发流程

业务模块指的是那些只需要读取容器内部数据不需要更新数据的模块

### 接口说明

#### CSnapshot

|  |  |
| --- | --- |
| 方法名 | 说明 |
| CSnapshot(const char \* pGroupName)  throw (XC\_EXCEPTION); | 创建容器组快照(通过同一快照获取到的容器数据版本是保证一致的)  pGroupName 容器组名 |
| AISTD string GetGroupName() const; | 获取组名 |
| void Release(); | 手动释放这个容器组快照 |
| time\_t GetValidTime() const; | 获取这个快照下容器组的生效时间 |
| ~CSnapshot () | 自动释放引用 |

#### CQueryHolder

|  |  |
| --- | --- |
| 方法名 | 说明 |
| template<typename \_Tp>  CQueryHolder(CSnapshot & cSnapshot,  const char \* pName) throw (XC\_EXCEPTION) | 构造一个容器查询引用类  cSnapshot 组快照  pName 需要使用的容器名 |
| \_Tp & GetContainer() throw (XC\_EXCEPTION); | 获取这个容器的引用 |
| void Release(); | 手动释放这个容器的引用 |
| bool IsValid(); | 判断这个容器引用是否有效 |
| const char \* GetName(); | 获取holder的容器名 |
| ~ CQueryHolder() | 自动释放引用 |

### 开发说明

#### 从容器中查询数据

|  |
| --- |
| typedef xc::list<int> CXcIntList;  //获取组快照  xc::CSnapshot cSnap("demo::group");  //获取这个容器  xc::CQueryHolder<CXcIntList> cQueryHolder(cSnap, “list1”);  //使用容器  CXcIntList::iterator itr = cQueryHolder.GetContainer().begin();  while(itr != cQueryHolder.GetContainer().end())  {  printf(“value:%d\n”, (\*itr++));  }  itr = cQueryHolder.GetContainer().find(1); |

# STL标准容器介绍

参考STL相关手册

## 仿函数介绍

<http://www.baidu.com/s?wd=stl+%B7%C2%BA%AF%CA%FD&rsv_bp=0&rsv_spt=3&n=2&inputT=661>

# 扩展容器介绍

## multi\_index

multi\_index容器：有多个索引列，同一索引列支持多个KEY值存放，V只存放一份

### 接口说明

#### multi\_index

|  |  |
| --- | --- |
| 方法名 | 说明 |
| template<typename \_Key, typename \_Tp, typename \_CmpFunc = base\_cmp\_func<\_Tp>>  class index; | \_Key key类型  \_Tp 值类型  \_CmpFunc可嵌入值比对函数，默认使用base\_cmp\_func |
| void insert(key\_list\_type & key, const \_Tp & value, int32 iIdx = -1) | 插入一堆key与一个值到index中  key key的list  value 值  iIdx 是否指定index的位置，默认不指定，通过key\_list的位置决定index的位置 |
| int32 size() | 返回值个数 |
| int32 count(int32 iIdx, const \_Key & key) | 计算满足这个key 的值个数 |
| int32 key\_size() | key的总大小 |
| int32 index\_size() | index大小 |
| iterator begin(int32 iIdx) | 获取起始位置迭代器 |
| iterator find(int32 iIdx, const \_Key & key, \_CmpFunc func = \_CmpFunc()) | 根据key值查询value，返回迭代器 |
| void clear() | 清空 |
| value\_iterator value\_begin() | 返回值的迭代器，用于迭代容器中的所有值 |

#### multi\_index::iterator

|  |  |
| --- | --- |
| 方法名 | 说明 |
| operator++() | 移动下一个元素的位置 |
| operator++(int) | 移动下一个元素的位置 |
| const \_Key & key() | 返回KEY的引用 |
| const \_Tp & value() | 返回值的引用 |
| bool eof() | 是否到结尾 |

#### multi\_index::value\_iterator

|  |  |
| --- | --- |
| 方法名 | 说明 |
| operator++() | 移动下一个元素的位置 |
| operator++(int) | 移动下一个元素的位置 |
| const \_Tp & value() | 返回值的引用 |
| bool eof() | 是否到结尾 |

### 开发说明

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | idx | 0 | 1 | 2 |  | | name | a | b | c | value | | data | ‘1’ | ‘2’ | ‘3’ | ‘v1’ | | ‘1’ | ‘3’ | ‘4’ | ‘v2’ | | ‘2’ | ‘3’ | ‘5’ | ‘v3’ | | ‘3’ | ‘2’ | ‘6’ | ‘v4 |   PS:  1.table:本table有3个索引列a,b,c，下标分别为0,1,2  2.数据的value为简单的String  typedef xc::multiindex<xc:string, xc::string> CIndexTable1;  CIndexTable1 cIndex;  CIndexType::key\_list keyList1, keyList2, keyList3, keyList4;  //先构造第一个记录的key  keyList1.push\_back("1");  keyList1.push\_back("2");  keyList1.push\_back("3");  //插入到index中  cIndex.insert(keyList1, "v1");  //用同样的方法构造其它数据  keyList2.push\_back("1");  keyList2.push\_back("3");  keyList2.push\_back("4");  cIndex.insert(keyList2, "v2");  keyList3.push\_back("2");  keyList3.push\_back("3");  keyList3.push\_back("5");  cIndex.insert(keyList3, "v3");  keyList4.push\_back("3");  keyList4.push\_back("2");  keyList4.push\_back("6");  cIndex.insert(keyList4, "v4");  //1.查询b索引中值为"3"的所有记录(多条)  //b 对应的下标为1  xc::string key = "3";  CIndexTable1::iterator itr = cIndex.find(1, key);  for (; !itr.eof(); itr++)  {  //打印出记录KEY与内容  printf("%d:%s\n", itr.key(), itr.value().c\_str());  }  //2.查询c索引中值为"4"的记录(单条)  key = "4";  itr = cIndex.find(2, key);  if(itr.eof())  {  //打印出记录KEY与内容  printf("%d:%s\n", itr.key(), itr.value().c\_str());  }  //3.计算出a索引中值为"1"的记录个数  key = "1";  int32 iCount = cIndex.count(0, key);  //迭代所有值  CIndexTable1::value\_iterator vitr = cIndex.value\_begin();  for (; !vitr.eof(); vitr++)  {  printf("data:%s\n", vitr.value().c\_str());  } |

## match

最大化匹配容器



### 接口说明

#### match

|  |  |
| --- | --- |
| 方法名 | 说明 |
| template<typename \_Tp,  typename \_CmpFunc = base\_cmp\_func<\_Tp>>  class match; | \_Tp 值类型  \_CmpFunc可嵌入值比对函数，默认使用base\_cmp\_func |
| void insert(key\_list\_type & key, const \_Tp & value, int32 iIdx = -1) | 插入一堆key与一个值到match中  key key的list  value 值  iIdx 是否指定index的位置，默认不指定，通过key\_list的位置决定index的位置 |
| int32 size() | 返回值个数 |
| iterator find(int32 iIdx, const \_Key & key, \_CmpFunc func = \_CmpFunc(), bool bBackTrack = true) | 根据key值查询value，返回迭代器  bBackTrack 当找不到值是是否需要回朔到上一节点查询 |
| iterator find(int32 iIdx, const \_Key & key, bool bBackTrack = true) | 根据key值查询value，返回迭代器  bBackTrack 当找不到值是是否需要回朔到上一节点查询 |
| void clear() | 清空 |
| value\_iterator value\_begin() | 返回值的迭代器，用于迭代容器中的所有值 |

#### match::iterator

|  |  |
| --- | --- |
| 方法名 | 说明 |
| operator++() | 移动下一个元素的位置 |
| operator++(int) | 移动下一个元素的位置 |
| const \_Key & key() | 返回KEY的引用 |
| const \_Tp & value() | 返回值的引用 |
| bool eof() | 是否到结尾 |

#### match::value\_iterator

|  |  |
| --- | --- |
| 方法名 | 说明 |
| operator++() | 移动下一个元素的位置 |
| operator++(int) | 移动下一个元素的位置 |
| const \_Tp & value() | 返回值的引用 |
| bool eof() | 是否到结尾 |

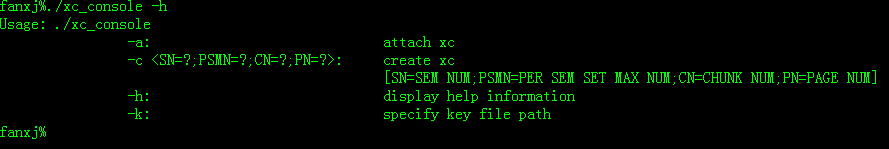
### 开发说明

|  |
| --- |
| struct base\_cmp\_funca  {  bool operator()(const xc::string & t1)  {  //过滤掉值为d1,d2的值  if (strcmp(t1.c\_str(), "d1") != 0 && strcmp(t1.c\_str(), "d2") != 0)  {  return true;  }  return false;  }  };  typedef xc::match<xc::string, base\_cmp\_funca> CMatchType;  CMatchType cMatch;  {  CMatchType::key\_list keyList;  keyList.push\_back("abcde");  cMatch.insert(keyList, "d1");  }  {  CMatchType::key\_list keyList;  keyList.push\_back("abcde");  cMatch.insert(keyList, "d2");  }  {  CMatchType::key\_list keyList;  keyList.push\_back("abc");  cMatch.insert(keyList, "d3");  }  {  CMatchType::key\_list keyList;  keyList.push\_back("abc");  cMatch.insert(keyList, "d4");  }  {  CMatchType::key\_list keyList;  keyList.push\_back("abcdefd");  cMatch.insert(keyList, "d5");  }  {  CMatchType::key\_list keyList;  keyList.push\_back("abcdefk");  cMatch.insert(keyList, "d6");  }  CMatchType::iterator itr = cMatch.find(0, "abcdef");  for (; !itr.eof(); itr++)  {  printf("%s:%s\n", itr.key().c\_str(), itr.value().c\_str());  }  //迭代所有的值  CMatchType::value\_iterator vitr = cMatch.value\_begin();  for (; !vitr.eof(); vitr++)  {  printf("value:%s\n", vitr.value().c\_str());  }  }  //运行结果  abc:d3  abc:d4 |

# XC控制台使用

提供对XC的管理与状态查询

## 启动参数



-a attach到一个已存在的XC上

-c 创建一个新的XC

SN 信号量总个数

PSMN 系统中允许的每一个信量集最大的信号量个数

CN 共享内存块个数

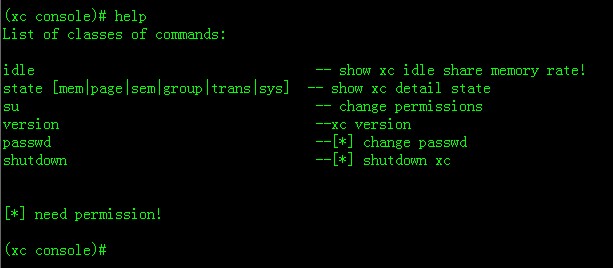
PN 一个共享内存块中的页数

ps:

1. 一个页大小为64K
2. 系统总的内存大小为CN\*PN\*64KB

-k 指定共享内存key文件存放路径（也就是区分不同XC的标识）

## 功能



idle 显示当前xc的内存空闲率

state 显示系统详细状态信息

mem 内存

page 内存页

sem 信号量

group 容器组信息

trans 事务信息

sys 系统参数信息

su 切换到超级权限

version 显示xc版本信息

passwd 修改超级权限密码

shutdown关闭xc容器

## 备注

1. 缺省密码为”xc”

# 组空间名约定

|  |  |
| --- | --- |
| **使用者** | **空间名** |
| 计费组 | JF:: |
| 帐务组 | ZW:: |
| 公共组 | COMMON:: |

**约定组名与容器名，都为大写，用\_分割**

# 数据装载代码自动生成

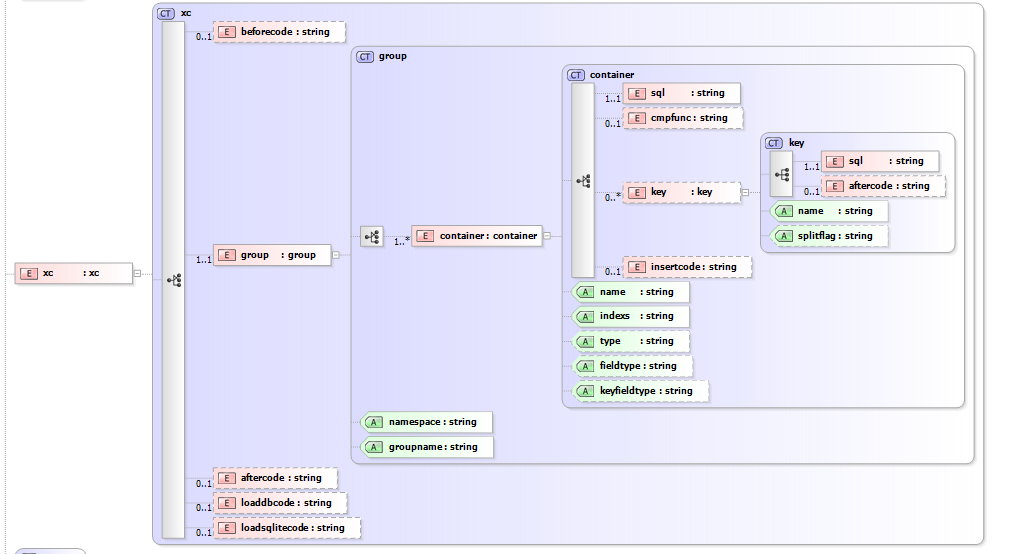






## 配置文件说明

### 配置文件schema



### 配置文件例子

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

<xc>

<group namespace="test" groupname="g2">

<container indexs="field\_no1;field\_no2;FIELD\_NO3" name="container1">

<sql>select field\_no1,field\_no2,field\_no3,field\_no4 from ^[xc\_load\_test2^]</sql>

</container>

<container indexs="f1" name="container2">

<sql>select f1 from ^[xc\_load\_test^]</sql>

<key name="key1">

<!--下面sql只是用于生成查询类结构用，不是正真的查询条件-->

<sql>select field\_no1 as no1,field\_no2 as no2,field\_no3 as no3,field\_no4 as no4 from ^[xc\_load\_test2^]</sql>

</key>

<key name="key2" splitflag="$">

<!--下面sql只是用于生成查询类结构用，不是正真的查询条件-->

<sql>select field\_no1 as abc1,field\_no2 as abc2,field\_no3 as abc3,field\_no4 as abc4 from xc\_load\_test2</sql>

</key>

</container>

</group>

</xc>

### sql关联表配置

使用^[^]标识出每一个sql中关联的表名，用于分用户时可以动态替换与前台显示容器与表的关系

<container indexs="field\_no1;field\_no2;FIELD\_NO3" name="container1">

    <sql>select field\_no1,field\_no2,field\_no3,field\_no4 from ^[table\_name1^]</sql>

  </container>

如果一个sql中，有多个表，请都标识出来，否则以后在分用户的情况下面，会识别不了

<container indexs="field\_no1;field\_no2;FIELD\_NO3" name="container1">

    <sql>select field\_no1,field\_no2,field\_no3,field\_no4 from ^[table\_name1^] , ^[table\_name2^]</sql>

  </container>

如果一个sql中，有多个表，请都标识出来，否则以后在分用户的情况下面，会识别不了。

xc在查询表时，先查找all\_tables表查找指定表对应的用户列表。并与XC\_OWNERS环境变量做匹配，寻找最优匹配对表名以user.tablename格式进行更新或修改。

XC\_OWNERS包含多个以分号做间隔的用户名。用户名先后顺序表示优先顺序。匹配时，排列在前的用户名优先。

### sql运行时变量

以^{^}标识变名量，在运行时传入这个变量，例如：

<container indexs="field\_no1;field\_no2;FIELD\_NO3" name="container1">

<sql>select field\_no1,field\_no2,field\_no3,field\_no4 from ^[table\_name1^] where 1=1 ^{where\_cond^}</sql>

 </container>

* api在调用接口时传入where\_cond的值

xc::CLoadOption cLoadOption;

cLoadOption.m\_mapSqlVariable["where\_cond"] = "and field\_no3 = 1";

g\_strXcDumpFileName = test::g2::Load4Db(cDbConn, cLoadOption, "container1");

* TS配置sql变量

    <sql\_variables>

                            <sql\_variable>

                                     <name>where\_cond</name>

                                     <value>and field\_no3 = 1</value>

                            </sql\_variable>

                   </sql\_variables>

### xcd插入自定义代码

插入代码有下面地方:

* **XC节点**

<xc>

<beforecode></beforecode><!—在生成的.cpp开始时插入代码-->

<aftercode></aftercode ><!—在生成的.cpp结束时插入代码-->

<loaddbcode></loaddbcode><!—在生成的Load4Db函数开始插入代码-->

<loadsqlitecode></loadsqlitecode><!—在生成的Load4Sqlite函数开始插入代码-->

</xc>

* **container节点**

<container>

<insertcode></insertcode><!—替换从数据库或则sqlite获取到数据后，插入xc的默认代码-->

</container>

红色的地方，就是insertcode替换的地方

bool CContainer16Desc::InsertBean2Xc(CContainer16::Type & cEditContainer, CContainer16 & cBean)

{

cEditContainer.insert(std::pair<xc::string, CContainer16>(cBean.GetFieldNo1(), cBean));

return true;

}

在替换代码的这些xml里面，请使用CDATA标签，以支持代码中的特殊符号

<![CDATA[代码内容]]>

### 自定义类型配置

<container indexs="f2" name="custom1" type="map" custom="true" fieldtype="F2=int32;f4=int32">

            <loaddbcode>

                static bool bPrint = true;

                if(bPrint)

                {

                    printf("$$$$$$$$$loaddbcode\n");

                    bPrint = false;

                }

                </loaddbcode>

            <sql>select f2,f4 from ^[xc\_load\_test^]</sql>

                <customtype><![CDATA[xc::map<int32, CCustom1> m\_mapData;]]></customtype>

                <customtype><![CDATA[xc::list<CCustom1> m\_lstData;]]></customtype>

            <loadallcode><![CDATA[

                printf("$$$$$$$$$loadallcode\n");

                return true;

            ]]></loadallcode>

        </container>

说明:

1.custom需要为true

2.分别有loaddbcode与loadallcode两个XML节点

3.loaddbcode:是单独每一条记录会调用业务代码

4.loadallcode:是所有记录从数据库中导完后再统一调用业务代码,如果同一个容器有loaddbcode存在,那么loadallcode会失效

5.customtype:是一个1-\*的节点,里面存放自定义类型

6.可以在容器中指定自定义类的名字,请修改customtypename这个属性,如果不指定那么有默认的名字来替换

### 容器类型配置

<container indexs="f1" name="container2"type=”map”>

支持的类型有

* multi\_index （默认）
* match
* map
* multimap
* vector
* list
* hash\_map
* hash\_multimap

## 调用xc2code生成代码

xc2code支持两种操作方式，一种在命令参数中指定连接串信息，另一种使用配置文件指定连接串信息。

使用命令参数指定连接串信息如下例：

xc2code -i xc\_load\_test\_g2.xml-u user -p password -c "db\_ip:db\_port:sid"

参数说明：

-i xc描述文件

-u 数据库连接用户名

-p 数据库连接用户密码

-c 数据库连接串格式为”db\_ip:db\_port:sid”

使用配置文件指定连接串信息如下例：

xc2code –i xc\_load\_test\_g2.xcd –C /usr/ob\_rel/config/xc2code.cfg –D aisconn

参数说明：

-i xc描述文件

-C 配置文件路径

-D 数据库连接别名

配置文件默认为$OB\_REL/config/xc2code.cfg，配置项如下例：

[db\_connections]

OBCONN = user/pass@db\_ip:db\_port:sid

xc2code使用别名指定连接，连接由用户名，密码，数据库连接串组成。格式为user/pass@db\_ip:db\_port:sid。

user为用户名。

pass为密码。

剩余部分为数据库连接串，格式为”db\_ip:db\_port:sid”

gmake支持自动重编xc相关文件，在makefile文件中指定以下变量即可。

IMPL\_XCD：该选项指定xcd描述文件名称，必须指定。即xc2code工具-i选项对应的内容。

XC\_DBCONN：该选项指定数据库连接别名，必须指定。即xc2code工具-D选项对应的内容。

RE\_GEN\_XC\_CODE：该选项指定是否强制重新生成xc相关文件。默认为强制生成。可选指定。

XC\_CFG\_PATH：该选项可指定配置文件路径，默认为$OB\_REL/config/xc2code.cfg。配置文件格式如上所述。

Makefile文件示例如下：

########################################################

#

# 后台动态库Makefile的写法

#

########################################################

# 包含基本的系统参数定义【不能更改】

include $(OB\_REL)/etc/NGbasedefine.mk

########################################################

# 目标名称，最终的程序文件名是 lib$(DEST)$(DLLTAIL)【必须修改】

DEST = xc\_ob\_kernel

# 目标的类型，必须是 DLL

DEST\_TYPE = DLL

# 编译目标程序需要的源代码文件，可以带路径（但必须是绝对路径）【必须修改】

#DEST\_SOURCES = xc\_ob\_kernel.cpp

IMPL\_XCD=xc\_ob\_kernel.xcd

XC\_DBCONN=OBCONN

RE\_GEN\_XC\_CODE = 1

USER\_INC\_PATH = \

$(OB\_REL)/include/public/xc \

$(OB\_REL)/include/3rd/otl \

########################################################

# 用户的宏定义，为编译器添加其它的 -D 参数，注意不要自行添加 -D 【可选】

# 需要连接的其它库文件，应使用 $(BUILDTYPE) 作为后缀【可选】

USER\_LIBRARIES += \

public\_xc$(BUILDTYPE) \

MAKEFILE\_NAME = Makefile.xc

########################################################

# 包含基本的 Makefile 规则文件【不能更改】

#include $(OB\_REL)/etc/NGCmake

include ../../etc/NGCmake

## sqlite dump文件名说明

【容器组空间名】.【容器名】.【事务创建时间】.xcdump

例如：

test.g2.20111010160144.xcdump

生成代码说明

默认生成一个对应的.h文件，文件中包涵了这个容器组下面所有容器的实体信息与描述信息，并提供个api，一个是从数据库装载数据，一个是从sqlite 的dump文件中装载数据到XC中，所有从数据库加载过来的类型在XC中都为xc::multi\_index（多KEY对多VALUE的），就是一个数据据有多纬索引，同时不是唯一索引

### 生成代码名

【配置文件名】.h

### 生成代码空间名为

【容器组空间名】::【容器组名】

### 生成代码结构

代码sql定义，生成数据实体类，与数据描述类

实体类名格式：C【容器名】

描述类名格式：C【容器名】Desc

查询类名格式：C【Key名】

实体类提供方法

* 实体属性的默认Get与Set方法
* 实体类内容打印方法Print，输出类信息到日志文件

查询类提供方法

* 实体属性的默认Get方法，不提供Set方法
* 实体类内容打印方法Print，输出类信息到日志文件
* 获取查询条件GenKey

### 自动生成接口

#### Load4Db:从数据库装载数据到XC

接口原型:

/\*!

\* \brief

\* 从数据库装载数据到XC

\* \param

\* cDbConn 数据库连接串

\* strContainerList 需要加载的容器列表，通过;号分割，如果为空表示加载所有容器

\* 例如：container1;container2

\* \exception 需要捕获XC\_EXCEPTION与otl\_exception信息

\* \return

\* dump到sqlite的文件名（全路径信息）

\*/

AISTD string Load4Db(otl\_connect & cDbConn, AISTD string strContainerList = "")

#### Load4Sqlite:从指定Sqlite文件中装载数据到XC

/\*!

\* \brief

\* 从sqlite装载数据到XC

\* \param

\* pDumpFullName sqlite dump文件名

\* strContainerList 需要加载的容器列表，通过;号分割，如果为空表示加载所有容器

\* 例如：container1;container2

\* \exception 需要捕获XC\_EXCEPTION信息

\*/

void Load4Sqlite(const char \* pDumpFullName, AISTD string strContainerList = "") \*/

#### Load:装载数据到XC(通用接口，这个接口会先调用Load4Db，如果不成功，那么再调用Load4Sqlite)

/\*!

\* \brief

\* 装载数据到XC(通用接口，这个接口会先调用Load4Db，如果不成功，那么再调用Load4Sqlite)

\* \param

\* cDbConn 数据库连接串

\* strDumpFullName sqlite dump文件名（默认为最后一次dump文件）

\* strContainerList 需要加载的容器列表，通过;号分割，如果为空表示加载所有容器

\* 例如：container1;container2

\*

\* \exception 需要捕获XC\_EXCEPTION信息

\*/

AISTD string Load(otl\_connect & cDbConn, AISTD string strDumpFullName = "test.g2.xcdump", AISTD string strContainerList = "")

### 例子

#### 从数据库装载数据

try

{

otl\_connect cDbConn;

cDbConn.rlogon("kt3/kt3@myoracle");

AISTD string strXcDumpFileName = test::g2::Load4Db(cDbConn);

printf(“sqlite dump file path:%s\n”, strXcDumpFileName.c\_str());

}

catch(XC\_EXCEPTION & xce)

{

printf("%s\n", xce.get\_message().c\_str());

}

catch(otl\_exception & otle)

{

printf("oracle exception!\n");

}

#### 从sqlite的dump文件中装载数据

try

{

test::g2::Load4Sqlite(“test.g2.20111006145833.xcdump”, "container1");

}

catch(XC\_EXCEPTION & xce)

{

printf("%s\n", xce.get\_message().c\_str());

}

xc::State("loadsqlite");

#### 从xc中查询数据

try

{

//查询数据

xc::CSnapshot cSnap(test::g2::CContainer1::GetGroupFullName());

xc::CQueryHolder<test::g2::CContainer1::Type> cQueryHolder(cSnap, test::g2::CContainer1::GetContainerName());

//通过INDEX=field\_no3查询数据等于“1”的记录

test::g2::CContainer1::Type::iterator itr =

cQueryHolder.GetContainer().find(test::g2::CContainer1::FIELD\_NO3, "1");

//循环打印数据内容

for(;!itr.eof(); itr++)

{

itr.value().Print();

}

}

catch(XC\_EXCEPTION & xce)

{

printf("%s\n", xce.get\_message().c\_str());

}

## 获取sqlite dump文件路径

### 通过xc\_console获取

(xc console)# state sys

XC SYS INFO:

VERSION:2.3.0

REVISION:$Revision: 494 $

CREATE\_TIME:2011-10-06 15:06:56

CHUNK\_NUM:1

PAGE\_NUM:10

ALL\_SEM\_NUM:12

PER\_SEM\_SET\_MAX\_NUM:12

ALL\_SEM\_SET\_NUM:1

TRANS\_MGR\_LOCK\_IDX:2

MEM\_MGR\_LOCK\_IDX:1

MEM\_PAGE\_MGR\_LOCK\_IDX:-1

SEM\_MGR\_LOCK\_IDX:0

XC\_START\_ADDR:0x40003000

**XC\_SQLITE\_DUMP\_PATH:/home/fanxj/xc/sqlite\_dump/**

XC\_IPC\_KEY\_PATH:/home/fanxj/ipc/xc/

(xc console)#

### 通过xc的api获取

/\*!

\* \brief

\* 获取sqlite dump path路径信息

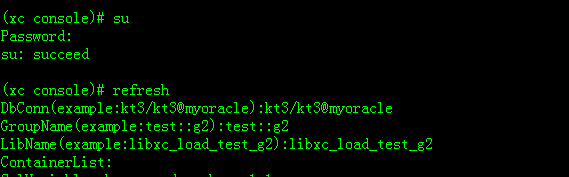
\* \return

\* 是否设置成功

\*/

AISTD string GetSqliteDumpPath();

## 通过xc\_console刷新数据



# MYSQL支持

## xcd编写支持mysql标签

<container indexs="t\_int64" name="map1" type="map" relationtable="">

        <sql>select sysdate from ^[dual^]</sql>

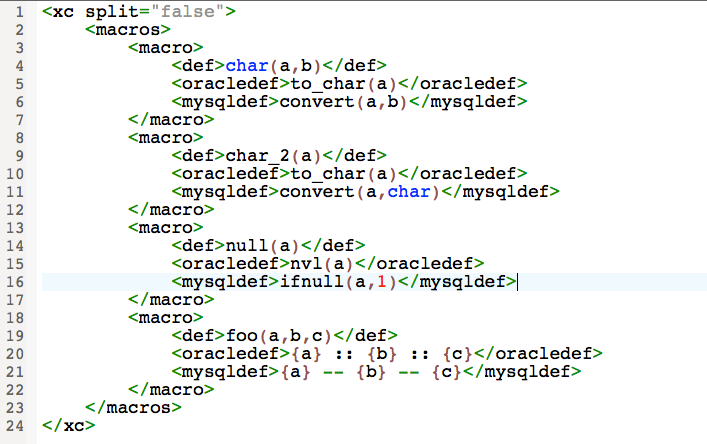
        <mysql>select sysdate()</mysql><!--mysql特定语法-->

  </container>

  a.如果一个容器在使用的在不同的数据库类型中会使用不同的sql,那么可以使用mysql这个标签来添加一个在mysql环境下面的sql语句

     如果这个sql是标准的,那么就不需要增加,直接支持在mysql与oracle两个环境下面运行

## xcd中宏支持



支持sql 中宏替换,在sql中写宏必须^$宏名字,这样的格式,例如: ^$char(t\_int32, char)

原始:

select t\_int16, t\_int32, t\_int64, t\_vchar, t\_char, t\_text,^$char(t\_int32, char) as macro , t\_bigint, t\_int from ^[xc\_mysql\_test^]

根据上面的宏定义:

1.在mysql环境下面最终生成的语句为:

select t\_int16, t\_int32, t\_int64, t\_vchar, t\_char, t\_text,convert (t\_int32, char) as macro , t\_bigint, t\_int from ^[xc\_mysql\_test^]

2.在oracle环境下面最终生成的语句为:

select t\_int16, t\_int32, t\_int64, t\_vchar, t\_char, t\_text,to\_char(t\_int32) as macro , t\_bigint, t\_int from ^[xc\_mysql\_test^]

## xc2code命令行

增加-t mysql,用于表示当前运行环境是mysql的,如果使用框架提供的Makefile来生成,那么可以不用关心这个参数,(需要更新框架的NGCmake)

  例如:xc2code -i xc\_load\_test\_g2\_mysql.xcd -u fanxj -p 123456 -c "localhost:3306/fanxjdb"  -t mysql -o .

## 修改xc2code.cfg

[db\_connections]

OBCONN=fanxj/123456@ip:3306/fanxjdb

# JAVA缓存支持

## 描述

java通过JNI的方式来访问XC共享内存

1.java访问不提供快照功能

2.java容器只提供hash\_map与hash\_multimap 2类，别的暂不提供

## XCD编写

<xc split="false" forjava="true">

增加forjava标签，如果是为java提供的XC缓存时，必须写为true

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

<xc split="false" forjava="true">

<group namespace="test\_xc" groupname="example\_jni">

<container indexs="f1" name="hashmap1" type="hash\_map" relationtable="">

<sql>select f1,f2,f3,f4 from ^[test\_xc\_oracle\_example\_base\_01^]</sql>

</container>

<container indexs="f2" name="mhashmap1" type="hash\_multimap" relationtable="">

<sql>select f1,f2,f3,f4 from ^[test\_xc\_oracle\_example\_base\_01^]</sql>

</container>

</group>

</xc>

## 编译

修改Makefile.xcd与对应的xcd文件，

直接gmake install;会自动生成jni文件与Makefile.jni，用于编译jni库

同时编译出对应xcd的刷新库

## 模板

########################################################

#

# 后台动态库Makefile的写法

#

########################################################

# 包含基本的系统参数定义【不能更改】

include $(OB\_REL)/etc/NGbasedefine.mk

########################################################

# 目标名称，最终的程序文件名是 lib$(DEST)$(DLLTAIL)【必须修改】

DEST = test\_xc\_example\_jni

# 目标的类型，必须是 DLL

DEST\_TYPE = DLL

# 编译目标程序需要的源代码文件，可以带路径（但必须是绝对路径）【必须修改】

IMPL\_XCD=test\_xc\_example\_jni.xcd

XC\_DBCONN=OBCONN

USER\_INC\_PATH = \

$(OB\_REL)/include/public/xc \

$(OB\_REL)/include/3rd/otl

########################################################

# 用户的宏定义，为编译器添加其它的 -D 参数，注意不要自行添加 -D 【可选】

# 需要连接的其它库文件，应使用 $(BUILDTYPE) 作为后缀【可选】

USER\_LIBRARIES += \

public\_xc$(BUILDTYPE)

########################################################

NEED\_DATABASE=1

# 包含基本的 Makefile 规则文件【不能更改】

include $(OB\_REL)/etc/NGCmake

修改Makefile.xcd红色字体的地方

## java客户端调用方式示例

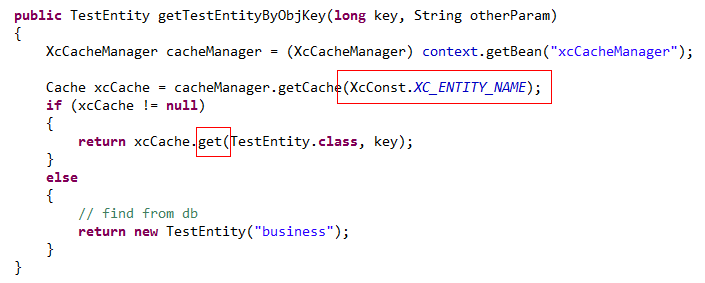
目前业务侧有两种调用方式。

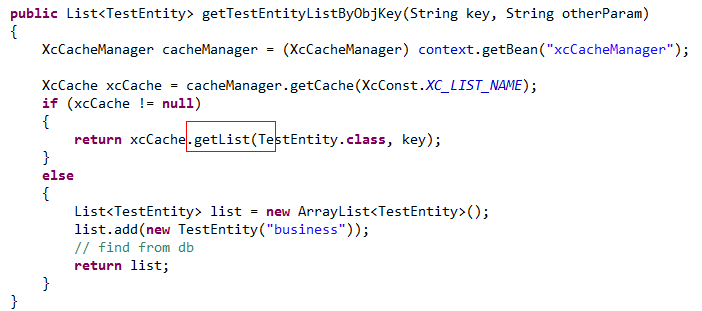
### 调用框架API实现

可以从spring上下文中获取XcCacheManager和XcCache，然后调用get和getList方法来获得XC缓存结果。如果XC不可用，则可以由业务侧自己实现获得结果方法，如从数据库获取等。



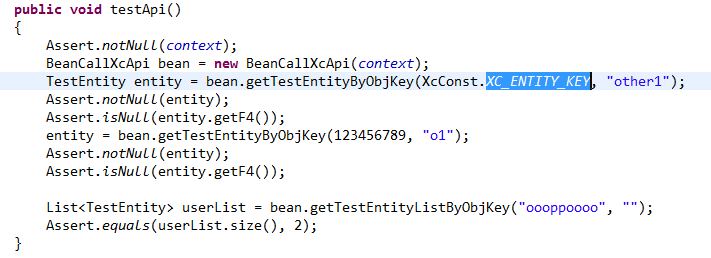






get方法获取单个缓存结果；getList方法获取缓存列表。

调用方示例：





对getCache方法的参数格式说明：



必须为group&container形式，用&分隔。

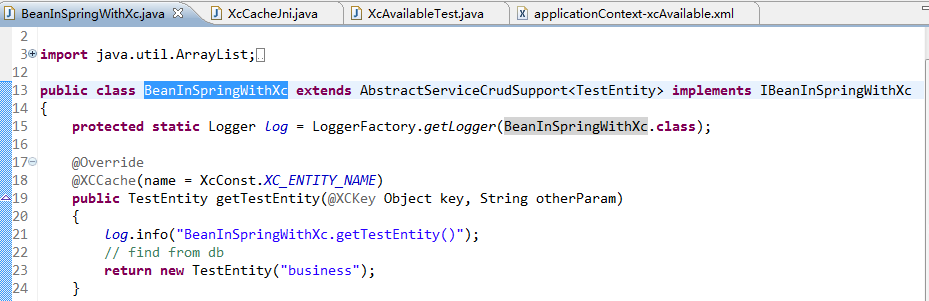
对XCKey的说明：



必须为short、int、long、String 这4种格式之一。

### 通过方法注解获取缓存

定义需要调用XC缓存的业务类，框架通过切面来实现XC缓存的获取，如果获取不到，则由业务自行实现逻辑，如从数据库获取等。



对于需要调用XC缓存的方法加上注解。

name格式如下：



其中格式为group&container，必须用&分隔。

ｋｅｙ的类型只能是ｓｈｏｒｔ，ｉｎｔ，ｌｏｎｇ，Ｓｔｒｉｎｇ类型之一。

spring定义：

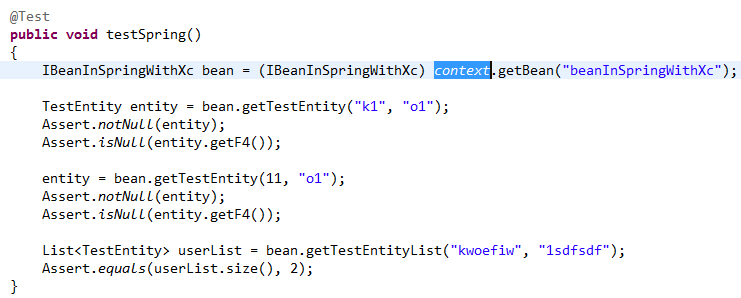


另外，目前框架采用JDK动态代理，因此对于方法请提取接口。

可以不继承其他类。

调用代码如下：





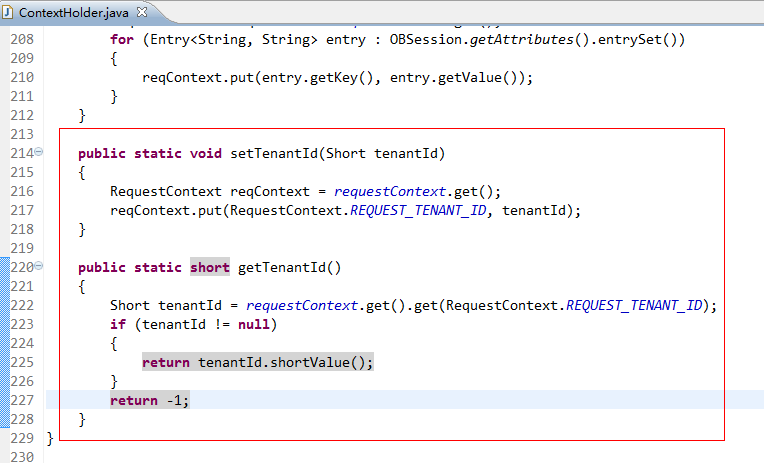
## Spring中XC缓存配置

在更新框架1.8.1的资源包之后，在业务侧生成代码之后的\generated\applicationContext-frame-server.xml文件中，会包含XC的配置：



## TenantId设置

com.ailk.easyframe.web.common.session.ContextHolder中提供方法用来在请求中设置tenantId：



该tenantId的值，在调用JNI方法时会传给XC。

# 通用前台刷新TS服务

## 配置说明

* **每一个对应的库需要单独配置一个TS服务**
* 统一TS库名为libpublic\_xc\_refresh\_tsD，函数名为createXcRefreshTs

         需要的配置为

         <xc\_refresh>

                   <dbapi\_name></dbapi\_name><!--[[必选]]数据库链接DBAPI别名，例如:OBK-->

                   <load\_lib\_name></load\_lib\_name><!--[[必选]]含有xc装载代码(含有编译自动生成的.cpp)的库名(就是上面生成的库) ，例如:libtest\_g2-->

                   <group\_name></group\_name><!--[[必选]]容器组名，例如:test::g2-->

<container\_list></container\_list><!--[[可选]]加载容器列表,以分号分割.例如:CONTAINER1;CONTAINER2;-->

                   <sql\_variables><!--[[可选]]sql变量-->

                            <sql\_variable>

                                     <name></name>

                                     <value></value>

                            </sql\_variable>

                            <sql\_variable>

                                     <name></name>

                                     <value></value>

                            </sql\_variable>

                   </sql\_variables>

         </xc\_refresh>

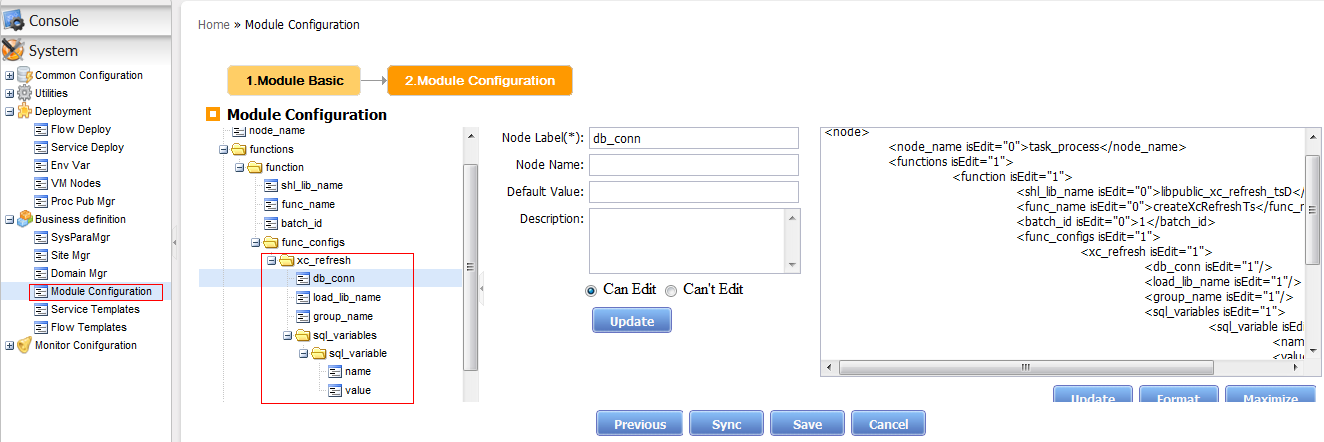
## DBM配置图解

1、配置说明

配置只有单个TS的流程。

2、组件配置

在组件中创建XC刷新所需要的配置：



也可通过文件module\_xc\_refresh.xml，导入组件：



3、服务模板配置

配置方法参考：

<http://10.10.10.141/svn/svnfiles/doc/product/cloudplatform/ver1/test/test_report/AIOpenBilling操作手册-流程配置.doc>

4、流程模板配置

配置方法参考：

<http://10.10.10.141/svn/svnfiles/doc/product/cloudplatform/ver1/test/test_report/AIOpenBilling操作手册-流程配置.doc>

5、流程实例化

配置方法参考：

<http://10.10.10.141/svn/svnfiles/doc/product/cloudplatform/ver1/test/test_report/AIOpenBilling操作手册-流程配置.doc>

主要配置项说明：

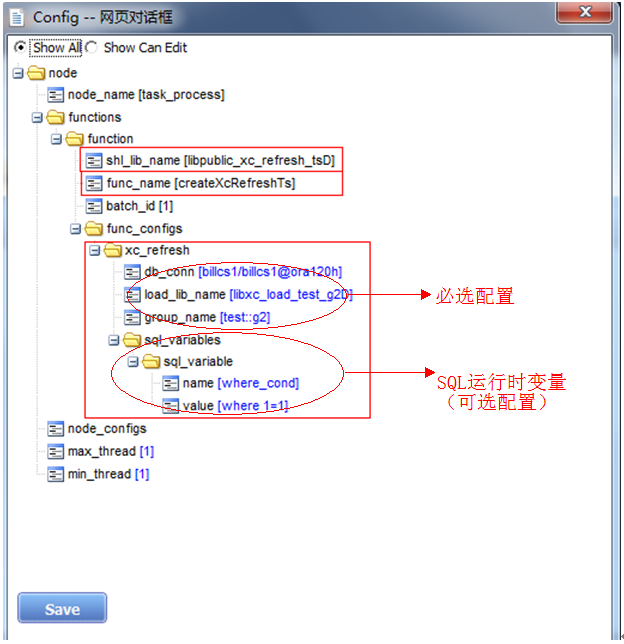
shl\_lib\_name：TS库名

func\_name：TS函数名

db\_conn：数据库连接信息

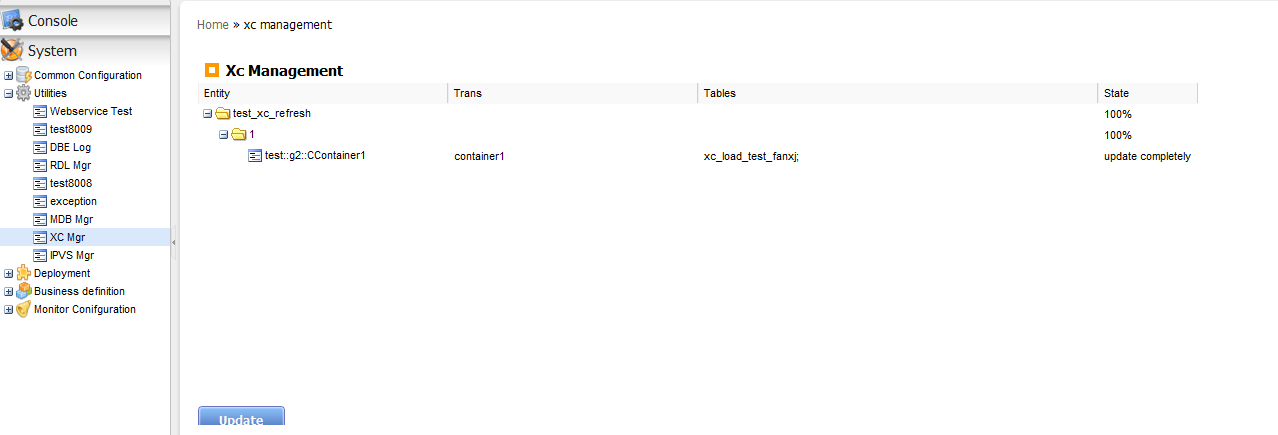
load\_lib\_name：装载数据的库名(库名最后的D不需要增加)

group\_name：容器组名

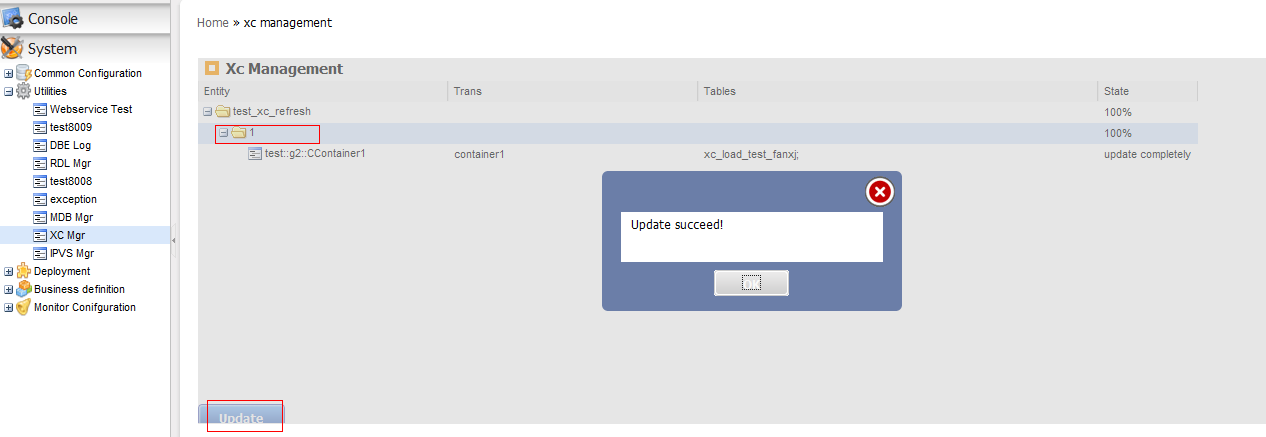


6、通过DBM前台进行XC刷新

启动流程成功后，点击System-->Utilities-->XC Mgr进入刷新界面



选择需要刷新的服务，点击”Update”按钮进行刷新



# 测试说明

## java客户端调用测试

XC缓存在window下不可用，因此在window下，只能进行模拟测试。

### 模拟测试

在window下，由于无法通过jni接口调用xc，因此通过模拟jni接口来测试。同时分为XC不可用和XC可用两种情况测试。单元测试类分别为：com.ailk.easyframe.xc.test.XcAvailableTest和com.ailk.easyframe.xc.test.XcNotAvailableTest，对应的配置文件为applicationContext-xcAvailable.xml和 applicationContext-xcNotAvailable.xml 。测试案例包含调用框架api和通过注解两种方式。

测试关键类介绍：

1、定义JNI类com.ailk.easyframe.xc.jni.XcCacheJni的子类，实现其中的逻辑，如：

**public** **class** XcCacheJniTemp **extends** XcCacheJni

{

@Override

**public** **boolean** isTest()

{

**return** **true**;

}

@Override

**public** **void** loadLibraryAndAttach(String ipcKeyPath, String xcLibraryPath, String jniCharsetName)

{

System.*out*.println("######loadLibraryAndAttach######");

}

@Override

**public** <T> List<T> get(String group, String container, **short** tenantId, Class<T> clz, String key) **throws** XcException

{

**return** newInstance(clz);

}

@Override

**public** <T> List<T> get(String group, String container, **short** tenantId, Class<T> clz, **short** key) **throws** XcException

{

**return** newInstance(clz);

}

@Override

**public** <T> List<T> get(String group, String container, **short** tenantId, Class<T> clz, **int** key) **throws** XcException

{

**return** newInstance(clz);

}

@Override

**public** <T> List<T> get(String group, String container, **short** tenantId, Class<T> clz, **long** key) **throws** XcException

{

**return** newInstance(clz);

}

**private** <T> List<T> newInstance(Class<T> clz)

{

List<T> list = **new** ArrayList<T>();

**try**

{

list.add(clz.newInstance());

list.add(clz.newInstance());

**return** list;

}

**catch** (Exception e)

{

e.printStackTrace();

**return** **null**;

}

}

}

把该类配置到spring中，并注入到XcCacheManager中：

<!-- 配置XC缓存 start 2013-10-23 15:34:39 -->

<!-- 测试需要 -->

<bean id=*"xcCacheJniTemp"* class=*"com.ailk.easyframe.xc.jni.XcCacheJniTemp"* />

<bean id=*"xcCacheManager"* class=*"com.ailk.easyframe.xc.cache.XcCacheManager"*>

<!-- 关闭所有缓存功能包括调用api实现的缓存 <property name="cacheOff" value="true"/> -->

<constructor-arg ref=*"xcCacheJniTemp"* />

</bean>

<bean id=*"xcCacheAspectj"* class=*"com.ailk.easyframe.xc.aspect.XcCacheAspectj"*>

<property name=*"xcCacheManager"* ref=*"xcCacheManager"* />

</bean>

<!-- 配置XC缓存 end 2013-10-23 15:34:44 -->

这样，业务侧在获取XC缓存数据时，则会根据XcCacheJniTemp中定义的逻辑来返回结果。

### 集成测试

框架集成测试在Linux下完成，目的就是测试java通过JNI调用XC是否可用。

测试环境：主机为：10.10.12.150 devbm1/devbm1

测试步骤：

把框架包打成一个大的jar（common-web\_fat.jar包含所有涉及到的jar包），其中的main类为com.ailk.easyframe.xc.main.CallCppXc，放到主机上，在XC可用的情况下，执行： java -DIPC\_KEY\_PATH=$HOME/ipc/xc -jar common-web\_fat.jar

能打印出从XC中配置的数据，则表示XC测试成功。

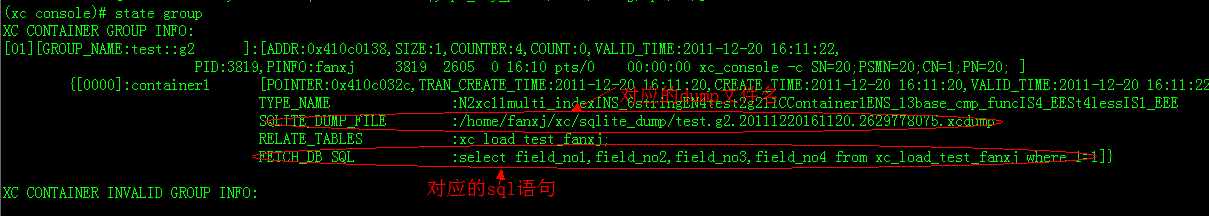
业务集成测试则需要自行配置XC数据，同时用业务代码测试。

# 使用技巧

## 如何查看xc中加载的数据

数据加载到xc中，大家可能有查看XC中数据的需求，现将方法告知

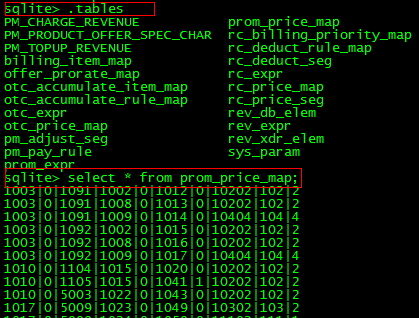
1. 进入到xc中，state group一下，找到对应的容器



1. 进入到dump的目录，里面会有很多xcdump的文件，账务的是zw.billing为前缀的dump,大家然后根据时间戳打开相应的dump文件

命令为：sqlite3  dump文件名  (sqlite是一种轻量级的数据库，sqlite3是这个数据库的命令)

1. 进入之后，大家可以.help 来查看相关的使用命令，我们最常用的就是.tables 来看有哪些表，即XC中的容器
2. 确定容器名之后，就可以用标准SQL来查询相关数据了



   5  退出   .exit

# Q&A

Q:XC本身有进程吗？

A:XC自身没有任何进程，是由若干个信号量与共享内存块组成的一个静态空间，它的数据是靠数据装载进程，装载进来

Q:XC中内存空间需要申请多大？

A:在系统集成环境下面，一台运行环境只运行一个XC，这个XC通过PP进程创建，XC容器的空间大小会根据每一个应用系统的要求统一做一个预分配，

在自己单元测试的环境下面，自己根据需要去创建XC空间大就即可

Q:数据装载如果完成？

A:XC内的数据应该是业务代码启动之前完成装载，数据装载进程在设计的时候必须与业务进程分离，因为数据装载进程对xc的访问是据有读写权限的，而业务进程是只有读权限的

Q:为什么要分读取与只读权限？

A:避免因为业务代码异常导致修改XC内存数据，造成整个系统数据出错而崩溃。

Q:快照创建会占用内存空间吗？

A:不会，快照的创建成本很低，速度很快，只是在这个容器组的一个引用计数上面加1

Q:快照支持多线程同时访问不加锁吗?

A:支持

Q:快照可以长期不释放吗？

A:快照用完必须立即释放（这个很重要），否则会影响这个容器组的数据更新，快照类试与你在数据库中select \* from table for update一张表，如果你不释放，那么数据装载进程更新时这个容器组数据回一直等待。

正确的快照使用方法

try

{

//创建快照（最好使用栈的方式，**不要用new与delete**）

xc::CSnapshot cSnap(test::g2::CContainer1::GetGroupFullName());

//查询容器组中的具体那一个容器

xc::CQueryHolder<test::g2::CContainer1::Type> cQueryHolder(cSnap, test::g2::CContainer1::GetContainerName());

test::g2::CContainer1::Type::iterator itr =

cQueryHolder.GetContainer().find(test::g2::CContainer1::FIELD\_NO3, "1");

//处理业务代码

for(;!itr.eof(); itr++)

{

itr.value().Print();

}

//退出这个函数时，快照会自动释放

}

catch(XC\_EXCEPTION & xce)

{

printf("%s\n", xce.get\_message().c\_str());

}

总结：

1. 快照需要用时创建，用完马上释放
2. 在快照期内避免很费时的代码处理（超过1分种的逻辑处理），可以考虑把这类代码放到快照期之外处理
3. 创建快照（最好使用栈的方式，不要用new与delete），如果快照是new出来的，必须有地方delete

Q:我如何查询一个XC的使用情况与基本信息?

A:通过[xc\_console](#_XC控制台使用)