1. 数据流

RedoFile

Transaction

End info

Transaction

Begin info

Change

RedoRecord

DmlChangeList

DmlRow

Sql

Transaction

1. 详细解释
2. RedoFile >>> RedoRecord

RedoFile >>> RedoRecord就是从RedoFile读出RedoRecord的过程。

RedoFile在逻辑上由FileHeader,RedoHeader, 多个RedoRecord组成。

RedoFile在物理上由多个日志数据块组成，简称日志块。

其中FileHeader位于0号块，占用整个日志块；RedoHeader位于1号块，占用整个日志块；剩下的就是RedoRecord的数据。

日志块分为: blkHeader 和 blkBody。

因为RedoFile在物理上有多个日志块组成，所以RedoRecord的数据也在日志块中，但是RedoRecord的数据忽略blkHeader的数据。

我们把从2号日志块到RedoFile的尾部称为RedoFileBody。

RedoFileBody由多个日志块的blkBody组成，忽略所有日志块的blkHeader。

RedoRecord的一个重要属性是rba,定义了RedoRecord的起始位置。

Rba的组成是redoFile的sequence号，日志块号，在日志块号中的偏移。

如果知道了RedoRecord的rba和RedoRecord的长度就能够把RedoRecord读出来。

blkHeader有个很重要的属性first\_new\_record\_offset，这表示在日志块中的第一个RedoRecord在日志块的偏移。first\_new\_record\_offset合法值是：等于零，大于等于blkHeader.size且小于blk.size。

如果first\_new\_record\_offset == 0 则表示该日志块的blkBody整体被包含在某一个RedoRecord中。

如果first\_new\_record\_offset >= blkHeader.size && first\_new\_record\_offset < blk.size 则至少有一个RedoRecord的起始位置位于该日志块，且第一个起始于该日志块的RedoRecord在该日志块中的偏移为first\_new\_record\_offset；

RedoFile的组成:

需要和下一个日志文件的第一个RedoRecord合并才能成为一个完整的redoRecord.

………..

无效数据

RedoRecord

无效数据

需要和上一个日志文件的最有一个

RedoRecord合并。

RedoHeader

FileHeader

RedoFileBody

读RedoRecord的逻辑:

::读第一个RedoRecord

>>>读取指针定位到2号块的blkBody处(第一个RedoRecord的起始位置)

>>>计算redoRecord的长度

>>>根据RedoRecod的起始位置和长度读出RedoRecord

>>>如果第一个RedoRecord需要合并，则和上一个日志文件最后一个RedoRecord合并

>>>跳过无效数据，使读取指针定位到下一个RedoRecord的起始位置

::读中间的RedoRecord

>>>读取指针已经定位到RedoRecord的起始位置

>>>计算RedoRecord的长度

>>>根据RedoRecord的起始位置和长度读出RedoRecord

>>>跳过无效数据，使读取指针跳到下一个RedoRecord的起始位置

::读取最后一个RedoRecord

>>>读取指针已经定位到RedoRecord的起始位置

>>>计算RedoRecord的长度

>>>根据RedoRecord的起始位置和长度读出RedoRecord

>>>如果RedoRecord的数据还没读完，则把它保留下来以便下一次的合并

::判断第一个RedoRecord是否需要合并

>>>读取指针已经定位到第一个RedoRecord的起始位置(2号块blkBody的起始处)

>>>如果2号块的blkHeader. first\_new\_record\_offset == blkHeader.size则不需要否则需要

::计算RedoRecord的长度

>>>如果是第一个RedoRecord且它需要合并 则计算第一个且需要合并的Record的长度

>>>否则通过尝试读RedoRecordHeader的方式计算Record的长度

::跳过无效数据

>>>读取指针已经定位到RedoRecord的末尾位置

>>>如果需要跳过当前日志块，则跳过当前日志块到下一块

::计算第一个且需要合并的的Record的长度

>>>读取指针已经地位到第一个RedoRecord的起始位置

>>>

recordLen = 0

while(blk.blkHeader. first\_new\_record\_offset == 0)

recordLen += blk.blkBody.size

redoFile.skipToNextBlk()

recordLen += blk.blkHeader. first\_new\_record\_offset – blk.blkHeader.size

>>>

(相同的缩进表示在相同的模块，和python相似)

::判断是否需要跳过当前日志块

>>>读取指针已经定位到RedoRecord的末尾位置

>>>如果当前日志块剩余的数据的长度小于Record.MinSize则需要

>>>如果当前日志块剩余的数据的长度大于等于Record.MinSize且尝试读出RedoRecord的长度后发现RedoRecord.length不在区间[RedoRecord.MinSize,RedoRecord.MaxSize] 则需要

(尝试读的意思是只读数据但是不移动读取指针)

对应的代码文件:

Lgc\_redoFile.cpp

Lgc\_RedoRecordInput.cpp

Lgc\_RedoRecord.cpp

2、RedoRecord >>>Change

RedoRecord >>>Change的过程就是从RedoRecord中读取Change的过程

RedoRecord的组成:

Record Header

Change 1

……

Change分为RedoChange和UndoChange, RedoChange依赖于UndoChange, UndoChange先出现，RedoChange后出现，相互依赖的UndoChange和RedoChange在同一个Record中且紧挨着。

从Record中读取Change的流程:

>>>如果ChangeList没有构建，则构建ChangeList

>>>从ChangeList中pop出Change

::构建ChangeList(record)

prevChange = NULL

For change in Record

changeList.add(change)

if PrevChange.isUndo()

change.addUndoRef(prevChange)

prevChange = change

(注解: for change in Record就是迭代地从Record中读出Change

Change.addUndoRef(prevChange) 为change添加Undo依赖

)

1. change >>> transaction begin info

change中有个特殊的类型我命名为事务开始型change, 简称BeginTChange

里面主要记录了事务开始时的一些信息，例如事务开始时的scn号。

1. change >>>transaction end info

change中有个特殊的类型我命名为事务结束型change,简称EndTChange.

里面主要的记录了事务结束时的信息，例如事务是否提交，事务结束时的scn号。

1. change >>>dmlChangeList

change >>>dmlChangeList 就是将多个dmlChange组装成dmlChangeList, 一个dmlChangeList代表了一个dml操作，dmlChangeList对应的dml操作一般只操作一行，但是只有一个情况dmlChangelist对应的dml操作操作了多行，就是多行直接插入且多行直接插入的数据在同一个数据块中。

一个dmlChange必须属于某个transaction, 因为dml操作不可能脱离事务而单独存在。

组装dmlChangeList的流程:

前提条件: dmlChangeList依赖的Transaction的所有的change都被添加到事务中

For change in transaction

If change.isDml()

dmlChangeList.addChange(change)

change.isEndOfDml()

dmlChangeList.setEnd()

parse(dmlChangeList)

dmlChangeList.clear()

(注解: for change in transaction 迭代地从事务中取出change

If change.isDml() 如果是dml型change

dmlChangeList.addChange(change) 将change添加到dmlChangeList

change.isEndOfDml() 该change中有个标志标志着dmlChangeList结束

parse(dmlChangeList) 解析dmlChangeList，得出一个或者多个dmlRow

)

Transaction的组成:

BeginTChange

EndTChange

Some other change that I don’t know

dmlChangeList

…………………………..

dmlChangeList

Some other change that I don’t kow

dmlChangeList的组成:

dmlChange

有dml结束标志的dmlChange

…..

dmlChange里的dmlChange根据dmlChange是RedoChange还是UndoChange

非为两个部分: redoDmlChangeList, undoDmlChangeList

1. dmlChangeList >>> DmlRow

dmlChangeList >>> DmlRow 解析dmlChangeList 得出一个或者多个dmlRow

根据dml操作类型将dmlChangeList的类型划分:insertDmlChangeList,

multiinsertDmlChangeList , deleteDmlChangeList, updateDmlChangeList.

dmlChange解析出来一般是多列数据，如果存在数据迁移(一列数据跨多个数据块)一列

数据可能要多个change才能存储，dmlChangeList一般情款是一行数据。但是有个特

情况MultiInsertDmlChangeList解析出来是多行数据。RedoChange解析出的是列的后镜

值，UndoChange解析出来的是列的前镜像值。前镜像值就是列修改之前的值，后镜像值就是列修改之后的值。

解析dmlChangeList大概流程：

>>>从dmlChangeList中获取object\_Id

>>>解析redoChangeList得出afterImageOfColumns

>>>解析undoChangeList得出beforeImageOfColumns

>>>根据object\_id, afterImageOfColumns, beforeImageOfColumns组装dmlRow

1. dmlRow >>> sql

解析dmlRow成dml语句，解析出的dml语句只操作一行。

dmlRow主要主城部分: object\_id, afterImageOfColumns, beforeImageOfColumns

解析dmlRow成dml语句 的伪代码:

if afterImageOfColumns.size() > 0 && beforeImageOfColumns.size() == 0

sqlText = “insert into “ + getTableName(object\_id) + “(“;

for column in afterImageOfColumns

sqlText +=column.name()

if 不是循环的最后一次

sqltext += “,”

sqlText += “) values(”

for column in beforeImageOfColumns

sqlText += column.value()

if 不是循环的最后一次

sqltext += “,”

sqlText += “)”

elif afterImageOfColumns.size() == 0 && beforeImageOfColumns.size() > 0

sqlText = “delete from “ + getTableName(object\_id) + “where　“

for column in beforeImageOfColumns:

sqlText += column.name()

sqlText += “=”

sqlText += columns.value()

if 不是循环的最后一次

sqlText += “ and “

elif afterImageOfColumns.size() > 0 && beforeImageOfColumns.size() > 0

sqlText = “update “ + getTableName(object\_id) + “ set “

for column in afterImageOfColumns:

sqlText += column.name()

sqlText += “=”

sqlText += column.value()

if 不是循环的最后一次

sqlText += “,”

sqlText += “ where “

for column in beforeImageOcColumns:

sqlText += column.name()

sqlText += “=”

sqlText += column.value()

if 不是循环的最后一次:

sqlText += “ and “

return sqlText