南京大学

本 科 生 毕 业 论 文

（申 请 学 士 学 位）

论　文　题　目　面向 **Redis list** 的 **OT** 函数的设计与验证

作　者　姓　名　纪业

学科、专业方向　计算机软件与理论

指　导　教　师　魏恒峰 讲师

研　究　方　向　分布式算法

2018 年 **5** 月 **14** 日

学　　号：**141220044**

论文答辩日期：**2018** 年 **6** 月 **7** 日

指 导 教 师：　　（签字）

Design and verification of OT function for Redis List

by

Ji Ye

Supervised by

Professor Wei Henfeng

A dissertation submitted to
the undergraduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
BACHELOR
in
Computer Software and Theory

Department of Computer Science and Technology
Nanjing University

May 14, 2018

南京大学本科生毕业论文中文摘要首页用纸

毕业论文题目：面向 Redis list 的 OT 函数的设计与验证

计算机软件与理论 专业 2014 级学士生姓名：纪业

指导教师（姓名、职称）：魏恒峰 讲师

摘　要

协同编辑系统，可以允许不同地点的用户同时编辑同一份文档。为了获得较快的响应和较高的实用性，系统会在不同的地点或设备进行文档的复制。一个用户可以在某个副本上进行文档的编辑，并将做出的修改异步地传递给其他副本。不必要等待服务器处理完再响应用户操作，本地操作可以立即执行。同时系统必须保证编辑的一致性，即在所有用户完成文档的编辑后，所有的副本内容一致。

可以设计 OT 函数，并通过控制算法的调用来保证最终结果的一致性，现在 ins,del,set 等简单 operation 的 OT 函数已经基本实现，本次毕业设计的目标是实现 Redis List 所支持的 14 种非阻塞操作的 OT 函数，并且对实现函数的正确性进行验证。阿里云和 RedisLab 的团队目前都在对 Redis List 的操作进行开发，Redis List 操作的 OT 函数实现具有应用前景和商业前途。

针对上述问题，本文的贡献包括：

首先介绍了 Redis List 的相关命令，并将 Redis List 的相关命令进行基于坐标和操作方式的分类，能够更方便地实现 OT 函数的设计。

其次对于所有简化后的命令进行了 OT 函数的数学公式设计。

最后介绍了 TLA +，并通过 TLA+ 实现了对于所设计 OT 函数的证明，以及对实验结果的分析。

关键词： 协同编辑；操作变换；Redis 系统；TLA+ 验证

南京大学本科生毕业论文英文摘要首页用纸

THESIS: Design and verification of OT function for Redis List

SPECIALIZATION: Computer Software and Theory

POSTGRADUATE: Ji Ye

MENTOR: Professor Wei Henfeng

Abstract

To be filled.

keywords:

目　　次

第一章　前言

1.1　应用背景：协同编辑应用

协同编辑系统，可以允许不同地点的用户同时编辑同一份文档。为了获得较快的响应和较高的实用性，系统会在不同的地点或设备进行文档的复制。一个用户可以在某个副本上进行文档的编辑，并将做出的修改异步地传递给其他副本。不必要等待服务器处理完再响应用户操作，本地操作可以立即执行。同时系统必须保证编辑的一致性，即在所有用户完成文档的编辑后，所有的副本内容一致。

1.2　技术背景：Replicated List 规约及其基于 OT 的 Replicated List 算法

1.3　本文研究工作: 面向 Redis List 的 OT 函数的设计、验证与实现

本次毕业设计的目标是实现 Redis List 所支持的 14 种非阻塞操作的 OT(Operational Transformation) 函数，并且对实现函数的正确性进行验证。阿里云和 RedisLab 的团队目前都在对 Redis List 的操作进行开发，Redis List 操作的 OT 函数实现具有应用前景和商业前途。

1.4　论文组织

本文后续内容组织如下：

第 2 章介绍本文的相关工作，包括系统模型和已有的相关 OT 函数的设计。

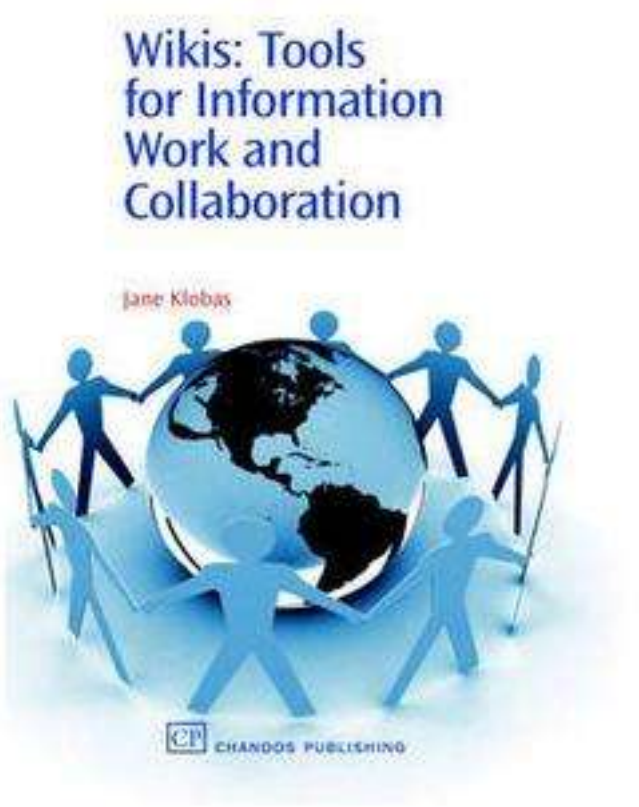

图 1-1: 系统编辑实例

第 3 章介绍了 Redis 列表相关的基本命令，并对其进行了分类，然后进行了对应 OT 函数的设计。

第 4 章介绍了 TLA+, 并使用 TLA+ 完成了对上述设计好的 OT 函数的验证。

第 5 章分析实验结果，并对实验结果进行分析。

第 6 章是本论文的结论和以后工作的相关展望。

第二章　相关工作

2.1　系统模型

一个 OT 系统包含 2 个关键的部分，上层的控制算法和底层的 OT 函数。控制算法负责决定哪些操作以何种顺序进行转换，而具体的 OT 函数则实施具体的两个操作之间的变化。OT 函数的数量由 OT 系统的模型所支持的数据和操作类型所决定。这两者由一系列转换条件和属性结合在一起，所以整个 OT 系统的正确性就是由控制算法和 OT 函数的正确性以及协议的正确性所共同决定的。

2.1.1　转换协议

已有的 OT 研究建立了一系列协议作为 OT 算法正确性的标准，违反任意一种协议都会导致不正确的转换结果，从而使得文档的编辑不能实现一致。有如下协议：

- Convergence Property 1(CP1):
- Convergence Property 2(CP2):
- Inverse Property 1(IP1):
- Inverse Property 2(IP1):
- Inverse Property 3(IP3):

2.2　已有 OT 函数的设计

可以设计 OT 函数，并通过控制算法的调用来保证最终结果的一致性，现在 ins,del,set 等简单 operation 的 OT 函数已经基本实现,ins,del 单个区间的 OT 函数也已经基本上设计完成，详见第三章中的第一类和第二类 OT 函数设计。

第三章　Redis List OT 函数设计

3.1　Redis List API 分类

3.1.1　Redis List API 简介

由于本次毕业设计是针对 Redis 系统的 List 命令进行的，首先对 Redis 中的列表 (List) 及其相关命令进行一个简要的介绍。Redis 中的列表是字符串列表，按照插入的顺序进行排序，一个列表可以包含的最多元素为 $2^{32}-1(4294967295)$ 个。

列表相关的基本命令共有 17 个，其中 3 个为阻塞性操作（即没有元素时会阻塞列表直至等待到有元素可以执行该命令），即其余 14 种为非阻塞性操作，即下面列出的 14 个命令。

- LINDEX key index: 通过索引 index 来获取列表 key 中的元素
- LINSERT key BEFORE(AFTER) pivot value: 在列表 key 某个元素 pivot 的前面 (BEFORE) 或者后面（AFTER) 插入元素 value，若元素不在列表中或者列表不存在则不执行任何操作
- LLEN key: 获取列表 key 的长度, 若 key 不存在则返回 0，如果 key 不为列表类型则出错
- LPOP key: 用于移除并返回列表的第一个元素
- LPUSH key value1 [value2]: 将一个值 (value1) 或者多个值 (value1,value2..) 插入到列表 key 头部，若 key 不存在则创建一个新列表并执行操作
- LPUSHX key value: 将一个值 (value) 插入到列表 key 头部，若列表 key 不存在则操作无效
- LRANGE key start stop: 返回列表 key 中指定区间 [start,stop] 内的元素
- LREM key count value: 根据 count 的值，移除列表 key 中与 value 值相等的元素（即删除 count 个）若 $count > 0$ 从表头开始搜索，若 $count < 0$ 从表尾开始搜索，若 $count = 0$ 则移除 key 中所有与 value 相等的元素

- LSET key index value: 通过索引 index 来设置列表 key 中元素的值为 value
- LTRIM key start stop: 对列表 key 进行修剪，只保留 [start,stop] 之间的元素，不在该区间的元素全部删除
- RPOP key: 移除并获取列表 key 中的最后一个元素
- RPOPLPUSH source destination: 移除并获取列表 source 中的最后一个元素，并添加到另一个列表 destination 中，返回该列表
- RPUSH key value1 [value2]: 将一个值 (value1) 或者多个值 (value1,value2..) 插入到列表 key 尾部，若 key 不存在则创建一个新列表并执行操作
- RPUSHX key value: 将一个值 (value) 插入到列表 key 尾部，若列表 key 不存在则操作无效

而其中 LRANGE,LLEN 和 LINDEX 这三个命令与 list 的内容修改无关，因此在本文中不考虑这两个命令的相关 OT 操作。本文中只考虑剩余这 11 个命令的 OT 函数的设计和验证。

3.1.2 Redis List API 分类

经过分析，这 12 个命令可以根据操作类型和作用范围分为以下这三类。

- 单个元素的删除、修改、插入：$Ins(pos, ele), Del(pos), Set(pos, ele)$
- 单个区间的删除、插入：$Ins(pos, str), Del(pos, len)$
- 多个区间的删除：$Del(pos1, len1; pos2, len2; ...; posk, lenk)$

具体来说，就是

- 第一类命令：
  - $LPUSHX -> Ins(0, ele)$
  - $RPUSHX -> Ins(len, ele)$
  - $LINSERT -> Ins(pos, ele)$
  - $LPOP -> Del(0)$
  - $RPOP -> Del(len - 1)$
  - $RPOPLPUSH -> Del(len - 1)$
  - $LSET -> Set(pos, ele)$

- 第二类命令：
  - $LPUSH->Ins(0, str)$
  - $RPUSH->Ins(len, str)$
- 第三类命令：
  - $LTRIM->Del(0, pos1-1; pos2+1, len-pos2-1)$
  - $LREM->Del(pos1, len1; pos2, len2; ...; posk, lenk)$

3.2 第一类 OT 函数的设计

我们定义第一类函数为第一类命令之间的 OT 函数，即 Ins,Del,Set 三种操作之间的 OT 函数，共有 3*3=9 个。

$$Set\begin{cases} OT(set(i,x), set(j,y)) = \begin{cases} no-op & pr1 > pr2 \quad i=j \\ set(i,x) & else \end{cases} \\ OT(Set(i,x), Ins(j,y)) = \begin{cases} Set(i,x) & i<j \\ Set(i+1,x) & i \geq j \end{cases} \\ OT(Set(i,x), Del(j)) = \begin{cases} Set(i,x) & i<j \\ no-op & i=j \\ Set(i-1,x) & i>j \end{cases} \end{cases} \tag{3-1}$$

$$Ins\begin{cases} OT(Ins(i,x), set(j,y)) = Ins(i,x) \\ OT(ins(i,x), ins(j,y)) = \begin{cases} ins(i+1,x) & i>j \\ ins(i,x) & i<j \\ ins(i+1,x) & i=j \quad pr1<pr2 \\ ins(i,x) & i=j \quad pr1>pr2 \end{cases} \\ OT(Ins(i,x), Del(j)) = \begin{cases} Ins(i,x) & i \leq j \\ Ins(i-1,x) & i>j \end{cases} \end{cases} \tag{3-2}$$

$$Del\begin{cases} OT(Del(i), Set(j,x)) = Del(i) \\ OT(Del(i), Ins(j,x)) = \begin{cases} Del(i+1) & i \geq j \\ Del(i) & i < j \end{cases} \\ OT(del(i), del(j)) = \begin{cases} Del(i-1) & i > j \\ Del(i) & i < j \\ no-op & i = j \end{cases} \end{cases} \tag{3-3}$$

3.3 第二类 OT 函数设计

我们定义第二类函数为第二类命令之间的 OT 函数，即 Ins,Del 两种命令之间的 OT 函数，共 2*2=4 个。

$$OT(Ins(p1,s1), Ins(p1,s2)) = \begin{cases} Ins(p1,s1) & p1 < p2 \\ Ins(p1+|s2|,s1) & p1 > p2 \\ Ins(p1+|s2|,s1) & p1 = p2 \quad pr1 < pr2 \\ Ins(p1,s1) & p1 = p2 \quad pr1 > pr2 \end{cases} \tag{3-4}$$

$$OT(Ins(p1,s1), Del(p2,l1)) = \begin{cases} Ins(p1,s1) & p1 \leq p2 \\ no-op & p2 < p1 < p2+l1 \\ Ins(p1-l1,s1) & p1 \geq p2+l1 \end{cases} \tag{3-5}$$

$$OT(Del(p1,l1), Ins(p2,s1)) = \begin{cases} Del(p1,l1) & p1+l1 \leq p2 \\ Del(p1,l1+|s1|) & p1 < p2 < p1+l1 \\ Ins(p1+|s1|,l1) & p1 \geq p2 \end{cases} \tag{3-6}$$

$$OT(Del(p1,l1),Del(p2,l2)) = \begin{cases} Del(p1,l1) & p1 < p2 \quad p1+l1 \le p2 \\ Del(p1,p2-p1) & p1 < p2 \quad p2 < p1+l1 \le p2+l2 \\ Del(p1,l1-l2) & p1 < p2 \quad p2+l2 < p1+l1 \\ no-op & p2 \le p1 < p2+l2 \quad p1+l1 \le p2+l2 \\ Del(p2,p1+l1-p2-l2) & p2 \le p1 < p2+l2 \quad p1+l1 > p2+l2 \\ Del(p1-l2,l1) & p1 \ge p2+l2 \end{cases} \tag{3-7}$$

3.4 第三类 OT 函数设计

我们定义第三类函数为第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数，以及第三类 Del 命令自身的 OT 函数，共 2+1=3 个。

首先考虑第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数，显然，我们需要按 Ins 操作的插入位置和 Del 操作的删除区间之间的关系进行分类，进行函数的设计。

Ins 操作对于 Del 操作的转换，如果是插入位置位于删除区间中，则 Ins 操作转换为 NOP，否则插入的位置要减去删除操作在该位置之前删除区间的总长度。

$OT(Ins(p_{k+1},s_{k+1}),Del(p_1,l_1;p_2,l_2;...;p_k,l_k))$

$$= \begin{cases} Ins(p_{k+1},s_{k+1}) & p_{k+1} \le p_1 \\ no-op & p_i < p_{k+1} < p_i+l_i \\ Ins(p_{k+1}-l_1-l_2-...-l_i,s_{k+1}) & p_i+l_i \le p_{k+1} \le p_{i+1} \\ Ins(p_{k+1}-l_1-l_2-...-l_k,s_{k+1}) & p_{k+1} \ge pk+lk \end{cases} \tag{3-8}$$

Del 操作对于 Del 操作的转换，如果是插入位置位于删除区间中，则该区间删除长度增加 $|s|$, 之前的区间不变，之后的区间都向后移动 $|s|$ 单位长度

如果插入位置不在删除区间中，那么在插入位置之前的区间不变，之后的区间都向后移动 $|s|$ 单位长度

$OT(Del(p_1,l_1;p_2,l_2;...;p_k,l_k),Ins(p_{k+1},s_{k+1}))$

$$= \begin{cases} Del(p_1 + |s_{k+1}|, l_1; p_2 + |s_{k+1}|, l_2; ...; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_{k+1} \le p_1 \\ Del(p_1, l_1; p_2, l_2; ...; p_{i-1}, l_{i-1}; p_i, l_i + |s_{k+1}|; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_i < p_{k+1} < pi + \\ Del(p_1, l_1; p_2, l_2; ...; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_i + l_i \le p_{k+1} \le \\ Del(p_1, l_1; p_2, l_2; ...; p_k, l_k) & P_k + l_k \le p_{k+1} \end{cases} \tag{3-9}$$

Del 操作自身的转换时最为复杂的，一开始想要将要转换的 Del 操作中所有的区间一起转换，发现这样不仅做起来难度很大，而且写公式和用代码表达都很容易出错，但如果是将每个区间都用某个公式来转换，然后将转换后的新区间合并为新的删除操作，就可以方便的实现第三类 Del 操作自身的转换了。

转变思路后，第三类 Del 自身的转换就可以用第二类 Del 操作对于第三类 Del 操作的转换来代替了，减少了公式的复杂度，同时也增加了可读性。

与前面的设计类似，由删除区间与删除区间之间的位置关系进行函数的设计。

$$OT(Del(p_{k+1},l_{k+1}),Del(p_1,l_1;p_2,l_2;...;p_k,l_k))$$

$$=\begin{cases}
Del(p_{k+1},l_{k+1}) & p_{k+1}<p_1 \quad p_{k+1}+l_{k+1}\le p_1 \\
Del(p_{k+1},p_j-l_1-l_2-...-l_{j-1}-p_{k+1}) & p_{k+1}<p_1 \quad p_j<p_{k+1}+l_{k+1}\le p_j \\
Del(p_{k+1},l_{k+1}-l_1-l_2-...-l_j) & p_{k+1}<p_1 \quad p_j+l_j<p_{k+1}+l_{k+1}\le \\
Del(p_{k+1},l_{k+1}-l_1-l_2-...-l_k) & p_{k+1}<p_1 \quad p_{k+1}+l_{k+1}>P_k+l_k \\
Del(p_i-l_1-l_2-...-l_{i-1},p_j-p_i-l_i-l_{i+1}...-l_{j-1}) & p_i\le p_{k+1}<p_i+l_i \\
 & p_j<p_{k+1}+l_{k+1}\le p_j+l_j \\
Del(p_i-l_1-l_2-...-l_{i-1},p_{k+1}+l_{k+1}-p_i-l_i-l_{i+1}-...-l_j) & p_i\le p_{k+1}<p_i+l_i \\
 & p_j+l_j<p_{k+1}+l_{k+1}\le p_{j+1} \\
Del(p_i-l_1-l_2-...-l_{i-1},p_{k+1}+l_{k+1}-p_i-l_i-l_{i+1}-...-l_k) & p_i\le p_{k+1}<p_i+l_i \\
 & p_{k+1}+l_{k+1}>P_k+l_k \\
Del(p_{k+1}-l_1-l_2-...-l_{i-1},p_j-p_{k+1}-l_{i+1}-l_{i+2}-...-l_{j-1}) & p_i+l_i\le p_{k+1}<p_{i+1} \\
 & p_j<p_{k+1}+l_{k+1}\le p_j+l_j \\
Del(p_{k+1}-l_1-l_2-...-l_{i-1},l_{k+1}-l_{i+1}-l_{i+2}-...-l_j) & p_i+l_i\le p_{k+1}<p_{i+1} \\
 & p_j+l_j<p_{k+1}+l_{k+1}\le p_{j+1} \\
Del(p_{k+1}-l_1-l_2-...-l_{i-1},l_{k+1}-l_{i+1}-l_{i+2}-...-l_k) & p_i+l_i\le p_{k+1}<p_{i+1} \\
 & p_{k+1}+l_{k+1}>P_k+l_k \\
Del(p_{k+1}-p_1-p_2...-p_k,l_{k+1}) & p_{k+1}\ge p_k+l_k \\
 & (i\ge j)
\end{cases} \tag{3-10}$$

3.5 剩余 OT 函数设计

显然，第三类的 del 命令可以涵盖第一类和第二类的 del 命令，第二类的 ins 命令可以涵盖第一类的 ins 命令，因此我们最后只要考虑 set(pos,ele) 命令和第二类的 ins 操作、第三类的 del 操作之间的 OT 函数关系，即可覆盖完全所有的 OT 函数设计。共有 2+2=4 个函数。

Ins 操作对于 Set 操作的转换就是其本身。

$$OT(Ins(i,s),Set(j,x)) = Ins(i,s) \tag{3-11}$$

Set 操作对于 Ins 操作的转换, 如果插入位置在 Set 位置之前则 Set 位置要增加 $|s|$ 单位长度，否则 Set 操作不变。

$$OT(Set(i,x),Ins(j,s)) = \begin{cases} Set(i,x) & i < j \\ Set(i+|s|,x) & i \geq j \end{cases} \tag{3-12}$$

Del 操作对于 Set 操作的转换就是其本身。

$$OT(Del(p_1,l_1;p_2,l_2;...;p_k,l_k),Set(p_{k+1},x)) = Del(p_1,l_1;p_2,l_2;...;p_k,l_k)$$

Set 操作对于 Del 操作的转换, 如果是 Set 位置位于删除区间中，则 Ins 操作转换为 NOP，否则插入的位置要减去删除操作在该位置之前删除区间的总长度。

$$OT(Set(p_{k+1},x),Del(p_1,l_1;p_2,l_2;...;p_k,l_k))$$

$$= \begin{cases} Ins(p_{k+1},s_{k+1}) & p_{k+1} < p_1 \\ no-op & p_i \leq p_{k+1} < p_i + l_i \\ Set(p_{k+1}-l_1-l_2-...-l_i,x) & p_i + l_i \leq p_{k+1} < p_{i+1} \\ Set(p_{k+1}-l_1-l_2-...-l_k,x) & p_{k+1} \geq pk + lk \end{cases} \tag{3-13}$$

第四章　基于 TLA+ 的 OT 函数验证

4.1　TLA+ 简介

TLA+ 是一种形式化的规范语言。它是一种设计系统和算法的工具，并且用来验证这些系统有没有关键错误。

正确性，是一个系统最为重要的性质，同时，正确性是比较难以证明的，特别是并发系统的正确性，因为存在着数目众多的状态变化，而 TLA+ 可以将系统的行为或者状态抽象为时态逻辑，即系统的行为或者状态会随着时间反生变化，然后通过一些数学分析的方法，来判断系统是否正确。

TLA+ 并不同于一般传统意义上的编程语言，更类似于一种数学语言，因为其语法大部分来自于实际的数理逻辑。

TLA+ 提供了工具集 TLAToolbox/TLC，同时还可以使用 TLA+ 的语法糖 PLUSCAL 来完成代码的编写，由于本文中并未涉及，在此不做展开。

```
----- MODULE M ----
EXTENDS M1, ..., Mn    \\* 引入其他模块，类似#include
CONSTANTS C1, ..., Cn   \\* 定义常量
VARUABLE x1, ..., xn    \\* 定义变量
ASSUME P1           \\* 假设、假定
\\* Definitions()  类似宏
OP(x1, ..., xn) == exp
\\* Functions 函数
f(x \in S) == exp
```

图 4-1: TLA+ 编码模板

4.2 使用 TLA+ 描述 OT 函数

4.2.1 调用模块的简单介绍

在本次实验中，我们引入了 TLA+ 中的 Sequences 模块，将 List 表示为一个 Sequecnce，并且使用了如下的 API：

operator	operation	example
Head	First element	Head(«1, 2») = 1
Tail	Sequence aside from head	Tail(«1, 2») = «2»
Append	Add element to end of sequence	Append(«1», 2) = «1, 2»
Len	Length of sequence	Len(«1, 2») = 2

4.2.2 OT 函数设计

第一、二类函数的表示

第三、四类函数的表示

命令执行的表示

4.3 正确性验证

在第二章中，我们提到过 CP1 和 CP2 两个协议，在这里我们只验证 CP1 性质的正确性。即同一个 List 经过 OT(OP2,OP1),OP1 或者 OT(OP1,OP2),OP1 这两种操作序列后，最终的结果是一致的。

用 TLA+ 来描述就是：

$$apply(apply(list, op1), Xform(op2, op1)) = apply(apply(list, op2), Xform(op1, op2))$$

只要对于任意的两个操作，这个等式都成立的话，那么 CP1 正确性即可得到验证。

第五章　实验结果

第六章　结论与未来工作

Redis List 命令的 OT 函数设计经过验证是成功的。

6.1　工作总结

6.2　研究展望

参考文献

[1] ARMBRUST M, FOX A, GRIFFITH R, et al. A View of Cloud Computing[J/OL]. Commun. ACM, 2010, 53(4): 50–58. http://doi.acm.org/10.1145/1721654.1721672.

[2] ZHANG I, SZEKERES A, VAN AKEN D, et al. Customizable and Extensible Deployment for Mobile/Cloud Applications[C/OL] // OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2014: 97–112. http://dl.acm.org/citation.cfm?id=2685048.2685057.

[3] ARDEKANI M S, TERRY D B. A self-configurable geo-replicated cloud storage system[C] // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014: 367–381.

[4] MOIR M, SHAVIT N. Concurrent Data Structures[C] // CRC. 2004.

[5] HUNT G C, MICHAEL M M, PARTHASARATHY S, et al. An Efficient Algorithm for Concurrent Priority Queue Heaps.[R]. [S.l.]: DTIC Document, 1994.

[6] HUANG Q, WEIHL W E. An evaluation of concurrent priority queue algorithms[C] // Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on. 1991: 518–525.

[7] SUNDELL H, TSIGAS P. Fast and lock-free concurrent priority queues for multi-thread systems[C] // Parallel and Distributed Processing Symposium, 2003. Proceedings. International. 2003: 11–pp.

[8] ATTIYA H, WELCH J. Distributed computing: fundamentals, simulations, and advanced topics: Vol 19[M]. [S.l.]: John Wiley & Sons, 2004.

[9] STEINKE R C, NUTT G J. A Unified Theory of Shared Memory Consistency[J/OL]. J. ACM, 2004, 51(5): 800–849.
http://doi.acm.org/10.1145/1017460.1017464.

[10] LAMPORT L. How to make a multiprocessor computer that correctly executes multiprocess programs[J]. Computers, IEEE Transactions on, 1979, 100(9): 690–691.

[11] ATTIYA H, WELCH J L. Sequential Consistency Versus Linearizability[J/OL]. ACM Trans. Comput. Syst., 1994, 12(2): 91–122.
http://doi.acm.org/10.1145/176575.176576.

[12] LIPTON R J, SANDBERG J S. PRAM: A scalable shared memory[M]. [S.l.]: Princeton University, Department of Computer Science, 1988.

[13] HERLIHY M P, WING J M. Linearizability: A correctness condition for concurrent objects[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1990, 12(3): 463–492.

[14] AHAMAD M, NEIGER G, BURNS J E, et al. Causal memory: Definitions, implementation, and programming[J]. Distributed Computing, 1995, 9(1): 37–49.

[15] VOGELS W. Eventually consistent[J]. Communications of the ACM, 2009, 52(1): 40–44.

致　　谢

感谢南京大学