南京大學

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 **20** 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 20, 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

1.2.1　规约

List 支持的常见简单操作 (insert, del, read)

规约: convergence (strong eventual consistency) Def2 and Def3 of “SSS” SIGMOD89 Page 4

1.2.2　OT

Operational Transformation(OT) 是一种为了支持协作功能，在协作软件系统中所采用的技术。OT 最早在 1989 年被提出，是为了在纯文本文档的协同编辑中实现一致性和并发控制所发明的，经过二十余年的研究，OT 的能力已经得到了拓展，在 2009 年 OT 做为一种核心技术被 Apache Wave 和 Google Docs 所采用来实现其合作特点。OT 的基本思想是根据先前执行的并发操作的影响，对正在编辑的操作的参数进行转换或调整，是转换后的操作能够达到正确地操作，并且保持文档的一致性。

1.3 OT-based 协议

client-server 结构

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。

structure.jpg structure.bb

图 1–1: OT 系统结构

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。

由于这种分层结构，我们可以单独考虑 OT 函数的设计，而不必关心控制算法。

1.4 本文研究工作: 面向 Redis List 的 OT 函数的设计与验证

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。

在 OT 函数的设计方面，本文使用数学公式表示出所有 OT 函数的基本形式，并且绘制图片和表格进行相应说明。

在 OT 函数的验证方面，使用 TLA + 完成了对所设计 OT 函数的验证，证明其满足 CP1 正确性，并且对验证代码的复杂度进行了相应分析。

1.5 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。

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，对实验结果进行了分析

第 5 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　OT 函数的性质

OT 函数需要满足的性质:

- Convergence Property 1(CP1): 这是 Jupiter 协议正确性的必要条件。对于定义在文档状态 S 上的给定操作 O1 和 O2，满足 CP1 等式:$S \circ O1 \circ OT(O2, O1) = S \circ O2 \circ OT(O1, O)$, 也就是说在 S 上按顺序实施 O1,OT(O2,O1) 操作和实施 O2,OT(O1,O2) 操作效果相同。
- Convergence Property 2(CP2): 对于定义在文档状态 S 上的给定操作 O1 和 O2，满足 CP2 等式：$OT(OT(O1, O2), OT(O3, O2)) = OT(OT(O1, O3), OT(O2, O3))$
- Inverse Property 1(IP1)
- Inverse Property 2(IP1)
- Inverse Property 3(IP3)

2.2　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。

2.2.1　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。

$$Set\begin{cases} OT(set(i,x),set(j,y)) = \begin{cases} no-op & pr1>pr2 \quad i=j \\ set(i,x) & else \end{cases} \\ OT(Set(i,x),Ins(j,y)) = \begin{cases} Set(i,x) & i<j \\ Set(i+1,x) & i\ge j \end{cases} \\ OT(Set(i,x),Del(j)) = \begin{cases} Set(i,x) & i<j \\ no-op & i=j \\ Set(i-1,x) & i>j \end{cases} \end{cases} \tag{2-1}$$

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

$$Del\begin{cases} OT(Del(i),Set(j,x)) = Del(i) \\ OT(Del(i),Ins(j,x)) = \begin{cases} Del(i+1) & i\ge j \\ Del(i) & i<j \end{cases} \\ OT(del(i),del(j)) = \begin{cases} Del(i-1) & i>j \\ Del(i) & i<j \\ no-op & i=j \end{cases} \end{cases} \tag{2-3}$$

2.2.2 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。

$$OT(Ins(p1,s1),Ins(p1,s2)) = \begin{cases} Ins(p1,s1) & p1 < p2 \\ Ins(p1+|s2|,s1) & p1 > p2 \\ Ins(p1+|s2|,s1) & p1 = p2 \quad pr1 < pr2 \\ Ins(p1,s1) & p1 = p2 \quad pr1 > pr2 \end{cases} \tag{2-4}$$

$$OT(Ins(p1,s1),Del(p2,l1)) = \begin{cases} Ins(p1,s1) & p1 \le p2 \\ no-op & p2 < p1 < p2+l1 \\ Ins(p1-l1,s1) & p1 \ge p2+l1 \end{cases} \tag{2-5}$$

$$OT(Del(p1,l1),Ins(p2,s1)) = \begin{cases} Del(p1,l1) & p1+l1 \le p2 \\ Del(p1,l1+|s1|) & p1 < p2 < p1+l1 \\ Ins(p1+|s1|,l1) & p1 \ge p2 \end{cases} \tag{2-6}$$

$$OT(Del(p1,l1),Del(p2,l2)) = \begin{cases} Del(p1,l1) & p1 < p2 \quad p1+l1 \le p2 \\ Del(p1,p2-p1) & p1 < p2 \quad p2 < p1+l1 \le p2+l2 \\ Del(p1,l1-l2) & p1 < p2 \quad p2+l2 < p1+l1 \\ no-op & p2 \le p1 < p2+l2 \quad p1+l1 \le p2+l2 \\ Del(p2,p1+l1-p2-l2) & p2 \le p1 < p2+l2 \quad p1+l1 > p2+l2 \\ Del(p1-l2,l1) & p1 \ge p2+l2 \end{cases} \tag{2-7}$$

第二类简单例子

2.3 OT 函数的验证

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

本文针对 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。(可使用表格)

- 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 第三类 OT 函数设计

(画图/做表格)

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。

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

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

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

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

$$= \begin{cases} Del(p_1 + |s_{k+1}|, l_1; p_2 + |s_{k+1}|, l_2; ...; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_{k+1} \leq p_1 \\ Del(p_1, l_1; p_2, l_2; ...; p_{i-1}, l_{i-1}; p_i, l_i + |s_{k+1}|; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_i < p_{k+1} < pi + \\ Del(p_1, l_1; p_2, l_2; ...; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; ...; p_k + |s_{k+1}|, l_k) & p_i + l_i \leq p_{k+1} \leq \\ Del(p_1, l_1; p_2, l_2; ...; p_k, l_k) & P_k + l_k \leq p_{k+1} \end{cases} \quad (3\text{-}2)$$

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。

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

3.3 其它 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。

$$OT(Ins(i,s), Set(j,x)) = Ins(i,s) \tag{3-4}$$

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。

$$OT(Set(i,x), Ins(j,s)) = \begin{cases} Set(i,x) & i < j \\ Set(i+|s|,x) & i \geq j \end{cases} \tag{3-5}$$

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。

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

$$= \begin{cases} Ins(p_{k+1}, s_{k+1}) & p_{k+1} < p_1 \\ no-op & p_i \leq p_{k+1} < p_i + l_i \\ Set(p_{k+1} - l_1 - l_2 - ... - l_i, x) & p_i + l_i \leq p_{k+1} < p_{i+1} \\ Set(p_{k+1} - l_1 - l_2 - ... - l_k, x) & p_{k+1} \geq pk + lk \end{cases} \tag{3-6}$$

第四章　基于 TLA+ 的 OT 函数验证

4.1　TLA+ 简介

TLA+ 是一种形式化的规约语言。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。TLA+ model checker 与经典的模型检验工具类似，通过遍历系统模型的所有可能的行为，验证其正确性。

figure: assume 删除; 英文注释; 类似宏删除; 空行

```
----- 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

4.1.1 TLA+ Modules

TLA+ 提供了多种数据结构的实现，包括在本次实验中使用的 sequecnce 和 record 等。

record 数据结构类似于 C 语言中的 struct 结构，即一个 record 包含若干个 field，每个 field 可以在给定的集合中选取元素填充 field。sequence 表示一个序列，如果引入 Sequences 模块，也可以将数组当作序列来进行处理，并且使用模块里的相应函数在本次实验中，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

以上的操作都不会使原有的操作序列发生变化。

4.2 使用 TLA+ 描述 OT 函数

解释同时描述并验证 OT 函数。

4.2.1 LIST 相关操作表示

4.2.1.1 第一、二类操作的表示

采用 record 的数据结构来表示具体的操作。第一类操作的 record 分为两种，set 和 ins 有三个域:pos 位置，ch 操作字符，pr 优先级。而 del 操作少了 ch 字符域，这样定义可以减少无效重复的操作数量，使验证代码的效率得到

提高。

$$OP\_1\_set \triangleq [type : "set", pos : POS, ch : CH, pr : PR]$$
$$OP\_1\_ins \triangleq [type : "ins", pos : POS, ch : CH, pr : PR]$$
$$OP\_1\_del \triangleq [type : "del", pos : POS, pr : PR]$$
$$OP\_1 \triangleq OP\_1\_set \cup OP\_1\_ins \cup OP\_1\_del$$

第二类操作的 record 也分为两种,ins 操作有三个域:pos 插入位置,pr 优先级，str 要插入的字符串。del 操作也有三个域:pos 插入位置,pr 优先级，len 删除区间的长度

$$OP\_2\_ins \triangleq [type : "ins\_r", pos : POS, pr : PR, str : STR]$$
$$OP\_2\_del \triangleq [type : "del\_r", pos : POS, pr : PR, len : LEN]$$
$$OP\_2 \triangleq OP\_2\_ins \cup OP\_2\_del$$

4.2.1.2 第三类操作的表示

第三类操作同样是使用 record 来表示，包含两个 filed，即 type 类型和 ints 删除区间的集合。

$$OP\_3 \triangleq [type : "del\_m", ints : NoncoSeq]$$

在表示第三类 OT 函数时，我们首先遇到的问题便是如何表示第三类 List 命令，由于第三类 List 命令为删除多个不相交的集合区间，因此我们需要使用 TLA+ 遍历在 List 长度上所有可以表示出来的区间，一个命令的删除区间即为若干个不相交的符合条件的区间。在 TLA+ 中支持两个集合的笛卡尔积操作，因此所有符合条件的区间的集合 Intervals 可以如下定义：

$$Intervals \triangleq \{< a, b >\in POS \times POS : a + b \leq Maxnum(POS) + 1\}$$

a 为区间的起始断点,b 为区间长度, 因为一个命令的删除区间为若干个不相交的

符合条件的区间，定义区间后，便可以生成操作的删除区间。将一个删除操作的删除区间称为一个区间组合，那么有两种方式表示这种区间组合，一种是区间二元组的集合，另一种是二维数组，由于在接下来的 OT 函数中，需要较为方便地对操作进行转换，所以在本次实验中采用第二种表示方法，先生成第一种表示方法，即二元数组的集合。

$$NoncoIntervals \triangleq$$
$$\{ints \in SUBSET\ Intervals : \forall i, j \in ints : i[2] + i[1] \leq j[1] \vee j[2] + j[1] \leq i[1] \vee i = j\} \backslash \{\{\}\}$$

这样，*NoncoIntervals* 中每个元素都是一个区间组合，下面我们需要将这个集合变成一个二维数组，由于在 TLA+ 中没有执行某个指令固定次数的相应代码，所以在本次实验中需要多次使用递归来表示这种循环操作。一个符号在使用前必须先声明或者定义，所以使用递归要在函数或定义之前加上 recursive 关键字。那么比如在这里我们需要将区间组合的集合转换为区间组合的数组，就需要使用两个递归的定义。首先定义了将集合转化为序列的 SetTOSeq()：

$$RECURSIVE\ SetTOSeq(\_)$$
$$SetTOSeq(T) \triangleq IF\ T = \{\}\ THEN <<>>$$
$$ELSE\ LET\ t \triangleq CHOOSE\ x \in T : TRUE$$
$$IN\ <t>\ \backslash o\ SetTOSeq(T \backslash t)$$

接下来便可以定义 Seqset(T), 将二元组的集合的集合 T 转化为二维数组的集合：

$$RECURSIVESeqset(\_)$$
$$Seqset(T) \triangleq$$
$$IF\ T =\ THEN$$
$$ELSE\ LET\ t \triangleq\ CHOOSE\ x\ \in T : TRUE$$
$$IN\ Seqset(T \backslash t) \cup SetTOSeq(t)$$
$$NoncoSeq\ \triangleq\ Seqset(NoncoIntervals)$$

得到的集合 *NoncoSeq* 便为 ints 域的取值范围，这样遍历时得到的具体操作的

ints 域便为一个二维数组，该数组中横坐标表示第几个删除区间，纵坐标 1 的值表示该区间开始端带点，纵坐标为 2 的值表示区间长度。

4.2.2 命令执行的表示

命令的执行与 LIST 和具体操作相关，所以定义中使用的参数便是 LIST 和具体操作中的参数，使用 Sequences 模块中的 API 进行代码的编写。以第二类的 del 操作为例，调用了 $\backslash o$ 和 $SubSeq$ 定义:

$$del\_ran\_op(list, pos, len) \triangleq SubSeq(list, 1, pos-1) \quad \backslash o \quad SubSeq(list, pos+len, Len(list))$$

唯一不同的是第三类操作的执行函数，同样是由于需要采用循环操作，所以还是要使用递归的定义完成命令的执行。

$$RECURSIVE\ del\_mulran\_op(\_,\_,\_)$$

$$del\_mulran\_op(list, ints, num) \triangleq$$

$$IF\ num = 0\ THEN\ list$$

$$ELSE\ del\_mulran\_op(SubSeq(list, 1, ints[num][1]-1)\ \backslash o\ SubSeq(list, ints[num][2]$$

$$+ints[num[1], Len(list)), ints, num-1)$$

num 是 ints 的区间个数，从后往前进行删除，这样不会影响下面要操作的区间，每次递归删除一个区间，并且值就是删除后的 list，当 num 为 0 时就表示删除完成了，直接返回当前的参数中的 list 即可。

4.2.3 OT 函数的描述

使用定义 Xform 表示 OT 函数，即 Xform(lop, rop) 为 lop 对于 rop 转换后的新操作。以第一类函数为例：

$$Xform(lop, rop) \triangleq$$

$$CASE \quad lop.type = \text{"}ins\text{"} \wedge rop.type = \text{"}ins\text{"} -> Xform\_ins\_ins(lop, rop)$$

$$[] \quad lop.type = \text{"}ins\text{"} \wedge rop.type = \text{"}del\text{"} -> Xform\_ins\_del(lop, rop)$$

$$[] \quad lop.type = \text{"}ins\text{"} \wedge rop.type = \text{"}set\text{"} -> Xform\_ins\_set(lop, rop)$$

4.2.3.1 第一类函数的表示

第一类函数的表示较为简单，不需要使用递归结构。以 ins 操作对于 ins 操作的转换来举例说明，使用定义 $Xform\_ins\_ins$ 表示 OT 函数，即 $Xform\_ins\_ins(lins, rins)$ 为 ins 操作对于 ins 操作转换后的新操作。

$$
\begin{aligned}
&Xform\_ins\_ins(lins, rins) \triangleq \\
&\quad IF \quad lins.pos < rins.pos \\
&\quad THEN \quad lins \\
&\qquad ELSE \quad IF \quad lins.pos > rins.pos \\
&\qquad\quad THEN \quad [lins \quad EXCEPT !.pos = @ + 1] \\
&\qquad\quad ELSE \quad IF \quad lins.pr < rins.pr \\
&\qquad\qquad THEN \quad [lins \quad EXCEPT !.pos = @ + 1] \\
&\qquad\qquad ELSE \quad lins
\end{aligned}
$$

其中 EXCEPT ! 的意思为除了特定的域发生变化，其他域保持不变。这样参照着之前第一类函数的数学公式，便可以类似地将其他第一类函数用同样的方式表示出来。

4.2.3.2 第二类函数的表示

第二类函数的表示也比较简单，同样不需要使用递归结构。以 ins 操作对于 del 操作的转换来举例说明，使用定义 $Xform\_ins\_del\_r$ 表示 OT 函数，即 $Xform\_ins\_del\_r(ins, del)$ 为 ins 操作对于 del 操作转换后的新操作。

$$
\begin{aligned}
&Xform\_ins\_del\_r(ins, del) \triangleq \\
&\quad CASE ins.pos \leq del.pos -> ins \\
&\quad [] ins.pos > del.pos \wedge ins.pos < del.pos + del.len -> NOP \\
&\quad [] ins.pos \geq del.pos + del.len -> [ins \quad EXCEPT !.pos = @ - del.len]
\end{aligned}
$$

EXCEPT ! 的意思为除了特定的域发生变化，其他域保持不变。这样参照着之前第二类函数的数学公式，便可以类似地将其他第二类函数用同样的方式表示出来。

4.2.3.3 第三类函数的表示

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。这类函数的设计代码量占到了整个设计的一半，大量使用了递归定义，这里拿最为复杂的 Del 命令自身的 OT 函数来举例。

$RECURSIVE\ Xform\_del\_del\_m(\_,\_,\_)$

$Xform\_del\_del\_m(ldel, rdel, i) \triangleq$

$\quad IF\ i > Len(ldel.ints)\ THEN\ ldel$

$\quad\quad ELSE\ Xform\_del\_del\_m([ldel\ EXCEPT\ !.ints[i] = transdel4(@, rdel.ints)], rdel, i+1)$

使用定义 $Xform\_del\_del\_m$ 表示 OT 函数, 这里设置 i 为递归变量,i 为 ldel 中当前正在进行转换的区间，转换函数为 transdel4，按照公式 (3-10) 进行编写, 即进行一个删除区间相对于 rdel 的转换，这样当 i 的值大于 ldel 中删除区间的数量时，所有删除区间都完成了转换，将转换后的新区间合并起来，便是 ldel 转换后的新操作。

$$transdel4(int, ints) \triangleq$$
$$<< newpos(int[1], ints, 1), newlen(int[1], int[2], ints, 1, 0) >>$$

$newpos$ 和 $newlen$ 便是这个删除区间对于 rdel 的转换后的新区间端点和区间长度。$newpos$ 和 $newlen$ 两个定义同样使用递归结构来表示。$newpos$:

$RECURSIVEnewpos(\_,\_,\_)$

$snewpos(pos, ints, i) \triangleq$

$\quad IFpos < ints[1][1]THENpos$

$\quad ELSEIFi = Len(ints) \wedge pos \geq ints[i][1] + ints[i][2]THENpos - Dlen(ints, i, 0)$

$\quad ELSEIFpos \geq ints[i][1] \wedge pos < ints[i][1] + ints[i][2]THENints[i][1] - Dlen(ints, i-1, 0)$

$\quad ELSEIFints[i][1] + ints[i][2] \leq pos \wedge pos < ints[i+1][1]THENpos - Dlen(ints, i, 0)$

$\quad ELSEnewpos(pos, ints, i+1)$

递归变量为 i，代表当前在对 rdel 中第几个区间进行 newpos 的计算，每递归一次,i 的值加一，如果满足条件要求则返回相应 newpos 的值。其中 Dlen 的作用为统计 rdel 中前 i 个删除区间的长度之和 (代码详见附录）。*newlen*:

$$RECURSIVE newlen(\_,\_,\_,\_,\_)$$

$$newlen(pos, len, ints, i, sum) \triangleq$$

$$\quad IF\ i > Len(ints)\ THEN\quad len - sum$$

$$\quad ELSE\ IF\ pos + len < ints[i][1]\ THEN\ newlen(pos, len, ints, i + 1, sum)$$

$$\quad ELSE\ IF\ pos < ints[i][1] \land ints[i][1] < pos + len \land pos + len \leq ints[i][1] + ints[i][2]$$

$$\qquad THEN\ newlen(pos, len, ints, i + 1, sum + pos + len - ints[i][1])$$

$$\quad ELSE\ IF\ pos < ints[i][1] \land pos + len > ints[i][1] + ints[i][2]$$

$$\qquad THEN\ newlen(pos, len, ints, i + 1, sum + ints[i][2])$$

$$\quad ELSE\ IF\ ints[i][1] \leq pos \land pos < ints[i][1] + ints[i][2] \land pos + len \leq ints[i][1] + ints[i][2]$$

$$\quad THEN\ 0$$

$$\quad ELSE\ IF\ ints[i][1] \leq pos \land pos < ints[i][1] + ints[i][2] \land pos + len > ints[i][1] + ints[i][2]$$

$$\qquad THEN\ newlen(pos, len, ints, i + 1, sum + ints[i][1] + ints[i][2] - pos)$$

$$\quad ELSE\ newlen(pos, len, ints, i + 1, sum)$$

递归变量为 i 和 sum,i 表示当前在对 rdel 中第几个区间进行 newlen 的计算,sum 表示在当前计算完成的所有区间中 len 转换为 newlen 需要减去的长度。这样便根据公式 (3-10) 完成了 Del 命令自身的 OT 函数，第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数可以用类似地方式表示出来，同时，剩余的 OT 函数也可以用递归或非递归的形式根据公式进行表示，在此不再赘述。

4.3 正确性验证

上节中已经完成了 OT 函数的 TLA+ 描述，剩下的就是决定验证的目标和方式。验证的目标: CP1 如何用 TLA+ 表示。

我们验证 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。结合以上给出的

定义，用 TLA+ 来描述所设计 OT 函数的 CP1 正确性：

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

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。

因为在本次实验中不涉及系统状态的变化，只需证明等式对于所有操作都成立即可，因此没有 behavior spec，即没有初始状态 Init 和状态关系 Next。不过，在 TLC 中提供了 Evaluate Constant Expression 的功能，即可以计算常量表达式的值。以第一类函数的 CP1 验证为例，我们给出如下正确性定义：

$$\begin{aligned}
&correctness_1(list) == \\
&\quad \forall op1, op2 \in OP_1 : \\
&\qquad \lor op1.pr = op2.pr \\
&\qquad \lor apply(apply(list, op1), Xform(op2, op1)) = apply(apply(list, op2), Xform(op1, op2))
\end{aligned}$$

这样的话，$op1$,$op2$ 就实现了对 OP_1 中所有操作的遍历，并且对于满足要求的任意 $op1$ 和 $op2$，只要优先级满足要求（两个操作的优先级不相等），则其一定满足 CP1 等式。在 Evaluate Constant Expression 内填写这个表达式，我们便可以 check model 并根据 value 的值来检验函数的正确性了。

4.3.1 TLA+ Model Checker 设置及实验环境

在本次实验中，我们定义 Model Checker 中常量的参数值如下：

STR <- ”like”,”enjoy”,”fond”,”love”,”fantasy”

CH <- ”a”,”b”,”c”,”d”,”e”

PR <- 1,2

STR 为第二类插入操作中 str 域的可选择范围，CH 为第二类插入、设置操作中 ch 域的可选择范围，PR 为操作的优先级，在此处我们设定两个操作的优先级不相同，否则为违规操作变换。通过调整 LIST 常量的长度，实验在不同长度的 LIST 下完成各类函数的验证所需要的时间。

实验环境：

Number of worker threads: 12

Fraction of physical memory allocated to TLC: 16173 mb

4.3.2 实验结果

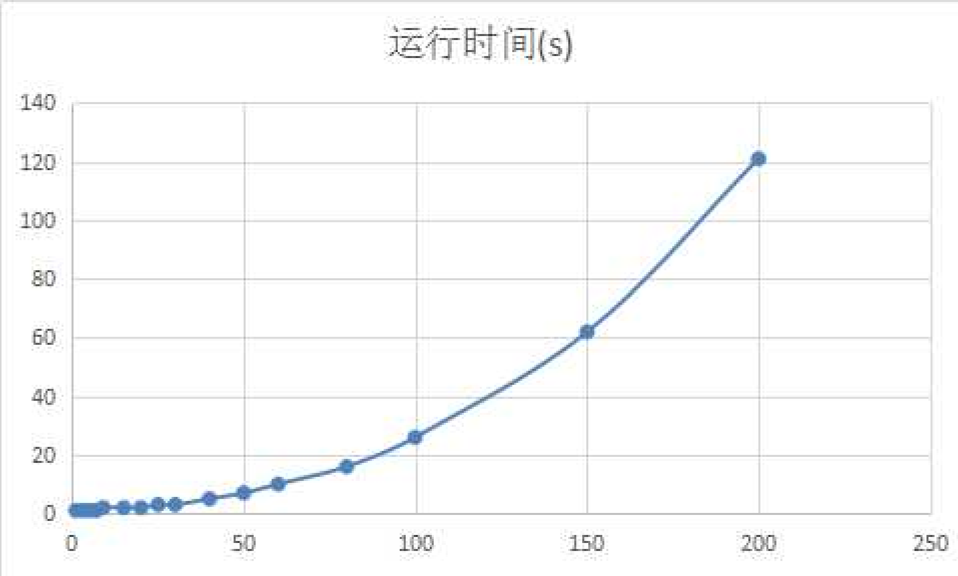

图 4–2: 验证第一类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：若列表长度为 n，则第一类操作的个数:$5*2*n(ins)+5*2*n(set)+2*n(del)=22n$ 任意选取其中两个操作验证 CP1 正确性，所以验证算法的时间复杂度为 $O(n^2)$ 从实验结果看，与复杂度的分析相一致。

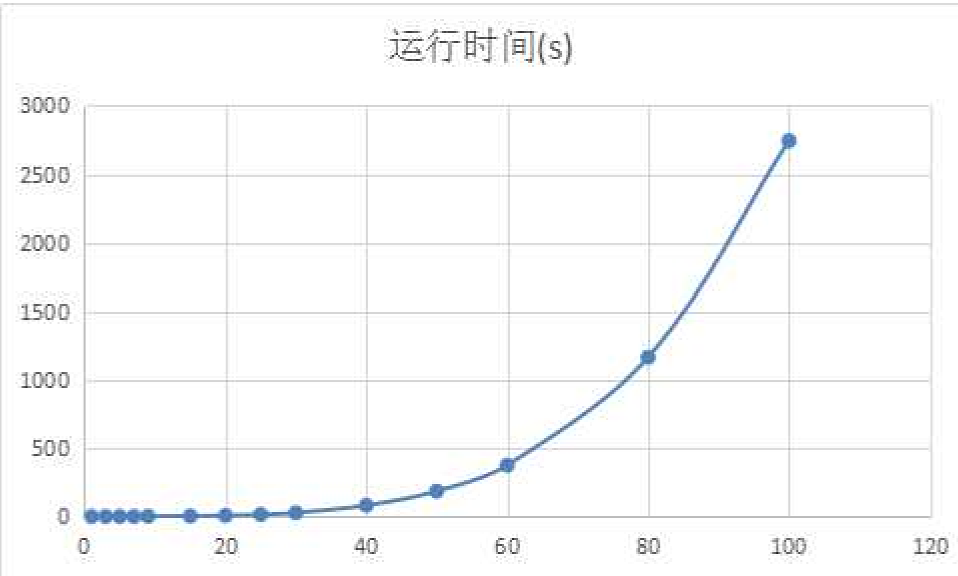

图 4–3: 验证第二类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：若列表长度为 n，则第二类操作的个数:$5*2*n(ins))+n*2*$

$n(del) = 2n^2 + 10n$ 任意选取其中两个操作验证 CP1 正确性，所以验证算法的时间副复杂度为 $O(n^4)$ 从实验结果看，与复杂度的分析相一致。

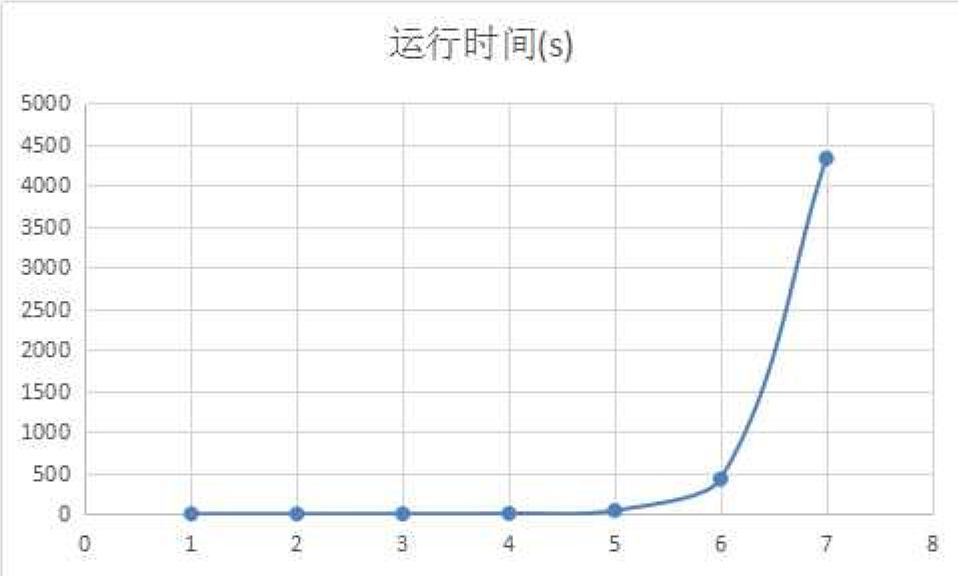

图 4–4: 验证第三类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：目前第三类函数的验证结果只能到 LIST 长度为 7。因为第三类操作的个数为指数级别 $O(3^n)$，并且在生成操作和操作的转换过程中多次使用了递归，更大大增加了验证算法的复杂程度。

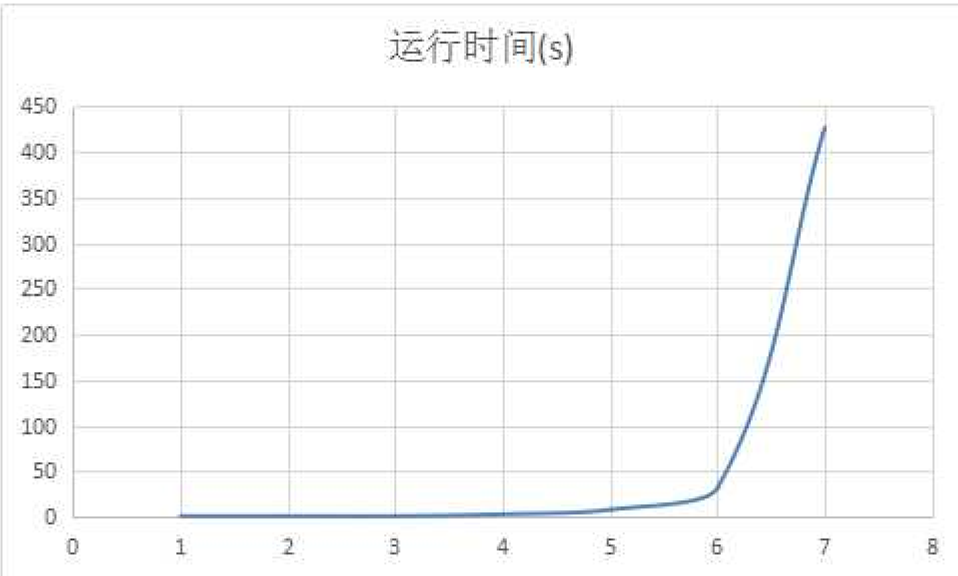

图 4–5: 验证其余函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：目前其余函数的验证结果只能到 LIST 长度为 7。与第三类函数

的验证相同，在其余函数的验证算法中也需要生成 $O(3^n)$ 级别的第三类操作和使用递归代码实现操作的转换，因此复杂度与第三类函数相仿。

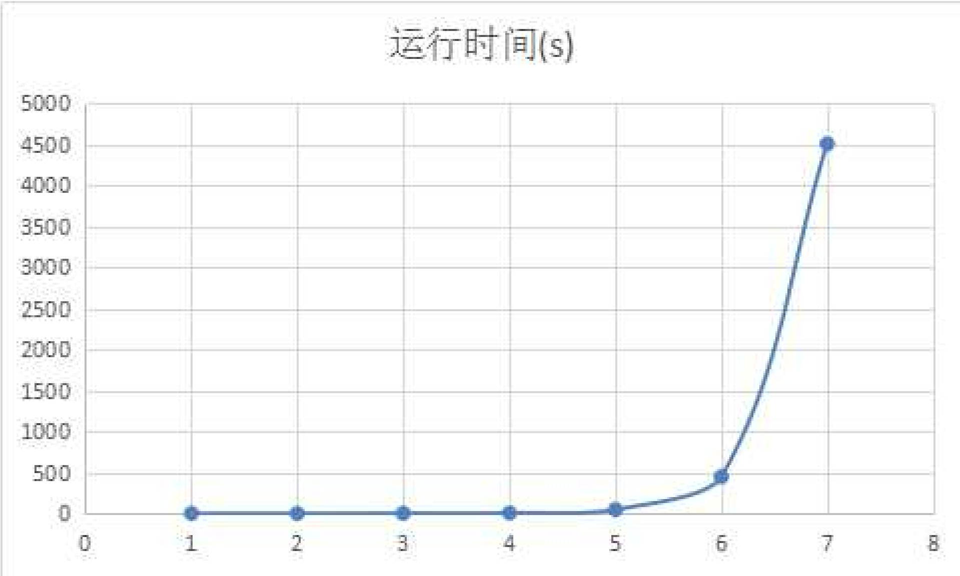

图 4-6: 验证所有 OT 函数正确性与列表长度（横坐标）的关系

结果分析：时间复杂度与第三类函数的复杂度相同，目前可以验证所设计的 OT 函数在 LIST 长度为 7 之内时的正确性。

第五章　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，其满足 CP1 正确性。

5.1　工作总结

本文所做实验论证了 Redis 系统中 List 相关命令的 OT 函数的设计与验证

5.2　研究展望

验证所设计的 OT 函数正确性只是一个起点，要真正能够在 Redis 系统中实现 List 的 OT 函数，还有许多工作亟待完成。

5.2.1　OT 函数的实现

由于 Reids 系统的主从结构和我们的要求不符合，要想在 Redis 中实现 OT 函数，还需要修改 Redis 的系统通信框架，使其变为 P2P 的通信结构。

5.2.2　验证代码的改进

此外，由于在本次实验的 TLA+ 代码中大量使用了递归的函数或者定义，使得对于第三类函数以及全部函数正确性验证的过程时间较长，效率较低，代码还存在着改进的空间。之所以采用 TLA+ 来完成 OT 函数的验证，正是因为其可以遍历所有操作的特性，改进验证代码目前有两个想法：一是使用语法糖 Pluascal 从而避免对于递归定义的依赖，二是使用其他编程语言来编写验证代码。

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