

中图分类号:

UDC:

学校代码: 10055

密级: 公开

南开大学
硕士学位论文

一种基于分布式内存缓存系统的高效容错
架构

An Efficient Fault Tolerance Framework for
Distributed In-memory Caching Systems

论文作者	赵帅兵	指导教师	王刚 教授
申请学位	工学硕士	培养单位	计算机学院
学科专业	计算机科学与技术	研究方向	分布式存储
答辩委员会主席	刘璟	评阅人	刘璟、王刚

南开大学研究生院

二〇一九年五月

南开大学学位论文使用授权书

本人完全了解《南开大学关于研究生学位论文收藏和利用管理办法》关于南开大学(简称“学校”)研究生学位论文收藏和利用的管理规定,同意向南开大学提交本人的学位论文电子版及相应的纸质本。

本人了解南开大学拥有在《中华人民共和国著作权法》规定范围内的学位论文使用权,同意在以下几方面向学校授权。即:

1. 学校将学位论文编入《南开大学博硕士学位论文全文数据库》,并作为资料在学校图书馆等场所提供阅览,在校园网上提供论文目录检索、文摘及前16页的浏览等信息服务;
2. 学校可以采用影印、缩印或其他复制手段保存学位论文;学校根据规定向教育部指定的收藏和存档单位提交学位论文;
3. 非公开学位论文在解密后的使用权同公开论文。

本人承诺:本人的学位论文是在南开大学学习期间创作完成的作品,并已通过论文答辩;提交的学位论文电子版与纸质本论文的内容一致,如因不同造成不良后果由本人自负。

本人签署本授权书一份(此授权书为论文中一页),交图书馆留存。

学位论文作者暨授权人(亲笔)签字: _____

20____年____月____日

南开大学研究生学位论文作者信息

论 文 题 目	一种基于分布式内存缓存系统的高效容错架构				
姓 名	赵帅兵	学号	2120160437	答辩日期	2018年5月19日
论 文 类 别	博士 <input type="checkbox"/> 学历硕士 <input checked="" type="checkbox"/> 专业学位硕士 <input type="checkbox"/> 同等学力硕士 <input type="checkbox"/> 划 <input checked="" type="checkbox"/> 选择				
学 院 (单 位)	计算机学院		学科/专业(专业学位)名称		计算机科学与技术
联 系 电 话	18697309570		电 子 邮 箱	zhaoshb@nbjl.nankai.edu.cn	
通讯地址(邮编): 南开大学津南校区计控学院 409					
非公开论文编号			备 注		

注:本授权书适用我校授予的所有博士、硕士的学位论文。如已批准为非公开学位论文,须向图书馆提供批准通过的《南开大学研究生申请非公开学位论文审批表》复印件和“非公开学位论文标注说明”页原件。

南开大学学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下进行研究工作所取得的研究成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或者没有公开发表的作品的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：_____年 月 日

非公开学位论文标注说明

根据南开大学有关规定，非公开学位论文须经指导教师同意、作者本人申请和相关部门批准方能标注。未经批准的均为公开学位论文，公开学位论文本说明为空白。

论文题目			
申请密级	<input type="checkbox"/> 限制 (≤2 年) <input type="checkbox"/> 秘密 (≤10 年) <input type="checkbox"/> 机密 (≤20 年)		
保密期限	20 年 月 日至 20 年 月 日		
审批表编号		批准日期	20 年 月 日

南开大学学位评定委员会办公室盖章 (有效)

注：限制 ★ 2 年（可少于 2 年）；秘密 ★ 10 年（可少于 10 年）；机密 ★ 20 年（可少于 20 年）

摘要

随着信息时代的高速发展，数据库系统应该为我们提供实时的、准确的和高性能的服务。为了满足人们对性能日益增长的需求，许多大型数据库应用开始采用分布式内存缓存系统来提高用户体验。其中，Memcached 就是一种典型的分布式内存缓存系统。分布式内存缓存系统将用户经常访问到的数据存放在内存中。这样，当用户需要访问这部分数据时，就不需要进行远程数据库操作，极大地提高了这部分数据的访问效率。然而，传统的分布式内存缓存系统没有容错能力。当系统中的一个节点发生故障时，该节点存放的数据将会丢失。用户如果想访问这部分数据，就需要重新从远端服务器或者磁盘中读取。这是一个延迟高，耗时长过程。为了提升分布式内存缓存系统的可靠性和可用性，Cocytus 将 Reed-Solomon 编码和分布式协议加入到了分布式 Memcached 中，使得 Memcached 具有容错能力。相比三副本技术，采用 Reed-Solomon 编码在具有相同容错能力的情况下可以节省大量的内存开销。

然而，Reed-Solomon 编码涉及到复杂的有限域运算，因此计算性能较低；另外，在数据恢复过程中，Reed-Solomon 编码需要大量的数据传输开销。Reed-Solomon 编码的两个缺点成为了分布式 Memcached 新的瓶颈。

本文将 Row-Diagonal Parity (RDP) 编码加入到分布式 Memcached 系统中来优化 Cocytus 的计算性能。然后，本文采用两种方案 Row-Diagonal Optimal Recovery (RDOR) 模型和 Collective Reconstruction Read (CRR) 模型来加速数据恢复过程。这将极大地提高用户体验和系统可靠性。本文所采用的方案在数据更新和数据正常访问的性能和 Cocytus 接近。在 4 个数据节点和 2 个校验节点的情况下，与 Cocytus 相比，本文所采用的方案可以降低 31% 的数据恢复时间。

关键词：Memcached；纠删码；恢复优化；并行优化

Abstract

With the development of the information age, the database systems are expected to provide real-time, accurate and high-performance services. In order to meet the growing need of people, many large database applications have introduced distributed in-memory object caching systems, of which Memcached is one of the most typical. These systems put the most frequently accessed data in memory so that user requests are processed without remote database operations. However, the traditional distributed Memcached does not have the fault-tolerant capability. If one server node is crashed, the lost data need to be reloaded from the remote servers or disks, which hurts the system performance. In order to make the distributed Memcached more reliable and available, Cocytus introduced Reed-Solomon codes and distributed protocol to the distributed Memcached which implement the fault-tolerant mechanism. Cocytus can save much memory compared to primary-backup replication when tolerating the same number of failures.

However, the relatively complex finite field calculation used by RS codes and the high network transmission cost during data reconstruction are becoming a new performance bottleneck of Memcached.

This paper introduced RDP codes into distributed Memcached to optimize the calculation performance of Cocytus. In addition, this paper adopted RDOR codes and Collective Reconstruction Read to speed up the rate of data recovery. The new distributed Memcached has a similar performance with Cocytus in terms of data update and normal data access. Compared with Cocytus which uses RS codes for fault-tolerant, the new distributed Memcached with 4 data nodes and 2 check parity nodes reduces reconstruction time by up to 31%.

Key Words: Memcached; erasure codes; optimal recovery; parallel recovery

目 录

第一章 绪论	1
第一节 研究背景和意义	1
第二节 本文主要工作	3
第三节 本文组织结构	4
第二章 背景知识	5
第一节 Memcached	5
2.1.1 Memcached 的发展	5
2.1.2 Memcached 的工作原理	6
2.1.3 Memcached 的内存管理	6
2.1.4 分布式 Memcached	9
第二节 Cocytus	12
2.2.1 Reed-Solomon Codes	12
2.2.2 Cocytus 系统	14
2.2.3 Cocytus 相关研究	17
第三节 本章小结	17
第三章 高效容错 Memcached 实现	18
第一节 基于 RDP 编码的分布式 Memcached 系统架构	18
3.1.1 RDP 编码的编解码原理分析	18
3.1.2 基于 RDP 编码的分布式 Memcached 系统架构	22
第二节 基于 RDP 编码的增量数据更新	25
3.2.1 基于 RDP 编码的增量数据更新	25
3.2.2 分布式 Memcached 的数据一致性	29
第三节 基于 RDOR 的数据恢复优化	31
3.3.1 RDOR 恢复优化原理	32
3.3.2 基于 RDOR 的数据恢复流程	34
第四节 基于 CRR 的数据恢复优化	39
3.4.1 CRR 恢复优化原理	39
3.4.2 基于 CRR 的数据恢复流程	41

第五节 RDOR 与 CRR 结合的可行性探讨	43
第六节 本章小结	47
第四章 实验与分析	48
第一节 实验环境描述.....	48
第二节 系统功能实验.....	49
第三节 系统性能实验.....	52
4.3.1 系统正常运行时的性能实验	52
4.3.2 系统故障模式下的重构性能实验	55
第四节 本章小结	59
第五章 总结与展望	60
第一节 本文的总结	60
第二节 未来的展望	61
参考文献	62
致谢	66
个人简历	67

第一章 绪论

第一节 研究背景和意义

在这个信息爆炸的年代，数据量的指数型增长^[1]导致大型数据库应用的并发流量急剧增加^[2]。关系型数据库管理系统（Relational database management systems, RDBMS）由于自身存储架构的缺陷^[3]，已经逐渐无法满足人们日益增长的性能需求^[2]。为了解决这个问题，分布式内存缓存系统应运而生。常见的分布式内存缓存系统有 Memcached^[4, 5] 和 Redis^[6]。这类系统的基本思想是将多台服务器的内存联合起来，作为一个大的内存池，然后将这整个内存池视作远程服务器或者磁盘的缓存。分布式内存缓存系统将用户经常访问的数据存放到内存池中。当用户需要访问这部分数据时，就可以直接从内存池中读取，而不需要经过复杂而漫长的数据库操作，这样就极大地提升了这部分数据的访问效率。因此，很多大型互联网公司^[7-9]就采用了内存缓存系统来增加系统响应速度，提升用户体验。

图1.1就是一个典型的分布式内存缓存系统示意图。该系统将3台服务器的内存联合起来。初始时，数据节点是没有数据的，用户根据需求从远程服务器获取数据，通过客户端上传到数据节点。随后用户对这些数据的再次请求就不需要涉及到远程服务器了。用户就可以直接向客户端发送数据请求。客户端根据映射算法将请求提交给某一个数据节点。数据节点收到请求后，从内存中取出请求的数据，返回给客户端。现在假设在系统正常运行过程中，某一个数据节点出现了故障。由于该系统没有容错功能，因此该节点上存放的所有数据就会丢失。用户对这部分数据的请求将会无效，因此需要重新从远程服务器读取。而远程服务器由于各种未知性，延迟可能非常高。因此，在原始的分布式内存缓存系统中，如果某个节点出现故障，系统性能可能急剧下降，极大地降低用户体验。

因此，我们需要有一个机制来保障分布式内存缓存系统即使有节点发生故障，也可以继续提供服务，也就是说我们需要容错。一个传统而且简单的容错方法就是副本（Primary-backup replication, PBR）^[10]。在副本方式中，每个主数

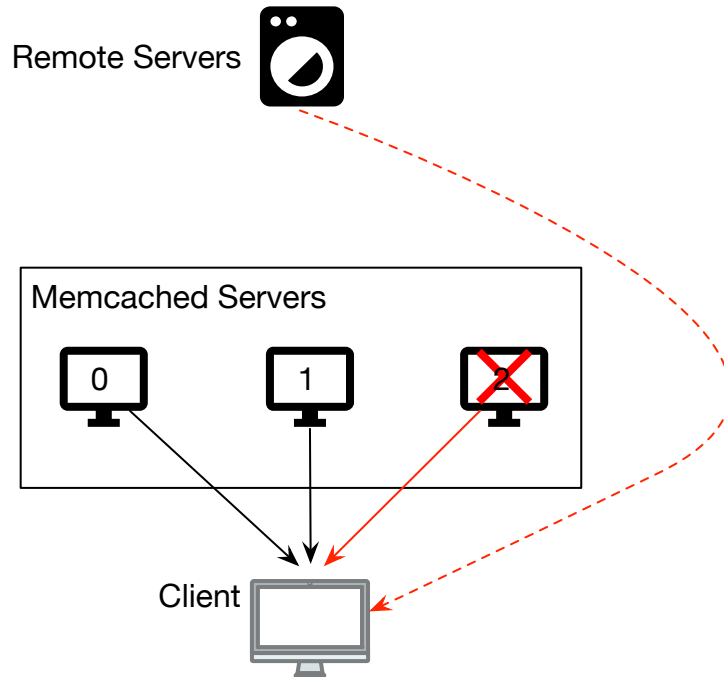


图 1.1 分布式内存缓存系统故障示意图

据节点都有多个备用数据节点。这些备用数据节点存放着主数据节点的数据备份。当一个主数据节点发生故障时，其中一个备用数据节点将作为新的主数据节点进行工作。虽然副本方式能够为分布式系统提供容错能力，但是它所带来的数据冗余是非常大的。一个分布式系统要能够容忍 M 个故障，数据就需要存放原数据量的 $M + 1$ 倍。

相比副本方式而言，纠删码 (Erasure codes) 在提供相同容错能力的情况下，需要更少的数据冗余。因此，纠删码在如今的分布式系统中应用非常广泛^[11-14]。存储领域众所周知的 Reed-Solomon codes (RS codes)^[15, 16] 就是一种典型的纠删码。在基于纠删码的分布式系统中，服务器节点被划分成数据节点和校验节点两种。数据节点存放原始数据，而校验节点存放原始数据的校验数据。校验数据通常由原始数据计算得到。几个数据节点和校验节点划分到一起构成一个编码组。每个节点上的数据被切分成多个大小相同的编码单元 (*unit*)。如果某个节点上的部分 *unit* 由于节点故障而丢失，这些丢失的 *unit* 可以由同个编码组中的其他节点上的 *unit* 通过计算恢复出来。

Memagent^[17] 是一个简单而有效的代理程序。它工作在 Memcached 的数据

节点和客户端之间，为 Memcached 提供了基于副本方式的容错能力。如果一个数据节点发生故障，客户端可以从其备用数据节点读取数据。但是，采用副本方式将造成大量的存储冗余，而内存资源非常珍贵，因此这种方式是非常浪费资源的。

Cocytus^[18] 首次将 RS 编码加入到 Memcached 中，实现了一个存储开销极小的分布式内存缓存系统。与副本方式相比，在能容忍两个节点故障的情况下，Cocytus 可以节省 46% 的内存空间，而且不会有明显的性能下降。

然而，Cocytus 所采用的 RS 编码虽然应用广泛，而且扩展性强，但是它也有两个不容忽视的缺点。首先，RS 编码在编解码的过程中需要复杂的基于有限域 (Finite field)^[19] 的运算。然后，在数据恢复的过程中，RS 编码涉及到大量的数据传输，会导致延迟偏高，性能下降。

综上所述，原始的分布式内存缓存系统在发生节点故障的时候，可能会导致性能急剧下降，影响用户体验。因此，探索如何利用纠删码来构造一个低冗余，高性能的分布式内存缓存系统是非常有意义的。

第二节 本文主要工作

首先，本文将利用一种更为高效的纠删码 Row-Diagonal Parity (RDP)^[20] 编码来取代 Cocytus 中的 RS 编码，实现一种具有高效容错能力的分布式内存缓存系统。RDP 编码的编解码计算过程只需要进行简单的 XOR 运算。我们知道 XOR 运算远比有限域的运算要高效而且更容易理解（可以帮助避免实现上的错误），因此采用 RDP 编码将比采用 RS 编码效率更高。

然后，针对 RS 编码数据恢复时数据传输量大的问题，本文在 RDP 编码的基础上提出了两种方案来进行恢复优化，分别是 Row-diagonal Optimal Recovery (RDOR)^[21] 和 Collective Reconstruction Read (CRR)^[22]。RDOR 是 RDP 编码的单故障优化恢复模型，它可以降低 RDP 编码在数据恢复过程中传输的数据量。而 CRR 是一种并行优化算法，它可以让 RDP 编码在数据恢复过程中并行传输所需数据。经实验验证，这两种方案都可以有效地减少分布式内存缓存系统在数据恢复过程中的所消耗的时间，提升系统的性能。

第三节 本文组织结构

本文将按如下结构进行组织：

第一章绪论部分主要阐述了本文的研究背景和意义，并概括性地介绍本文的主要工作和组织结构。

第二章将介绍与本文相关的背景知识，主要包括 Memcached 系统，RS 编码以及 Cocytus 系统。

第三章重点阐述了本文的核心工作，主要包括五个方面：一是基于 RDP 编码的 Memcached 系统架构；二是基于 RDP 编码的增量数据更新；三是基于 RDOR 的数据恢复优化方案；四是基于 CRR 的数据恢复优化方案；五是将 RDOR 和 CRR 结合的可行性分析。

第四章是实验部分，主要是将本文实现的基于 RDP 编码，RDOR，CRR 的系统与基于 RS 编码的 Cocytus 系统做性能上的对比分析。

第五章首先总结了本文研究所取得的成果，然后提出了未来可能的研究方向。

第二章 背景知识

第一节 Memcached

2.1.1 Memcached 的发展

随着互联网技术的高速发展，人们越来越多地通过互联网来获取信息。根据“We Are Social”披露的最新数据，在 2018 年全球互联网用户数已经突破了 40 亿，世界上一半的人口已经开始使用互联网，而且其中有 30 亿人口每个月都会使用社交媒体^[23]。据 CNNIC 发布的第 42 次《中国互联网络发展状况统计报告》称，截止 2018 年 6 月，我国网民规模已达 8.02 亿，互联网普及率达 57.7%^[24]。

互联网普及率的提高必然会导致数据量的爆炸式增长。根据国际数据公司 (IDC) 发布的大数据白皮书《Data Age 2025》预测，2025 年全球大数据规模将增长到 163ZB，相当于 2016 年所产生 16.1ZB 的 10 倍^[25]。

在如今的大数据时代下，数据量的暴增给数据库带来了巨大的挑战。现在的应用程序大多将数据存储在关系型数据库中。应用程序收到用户请求后，从数据库中获取相应的数据并返回给用户。但是，传统的关系型数据库响应速度较慢，并发处理能力也比较弱，导致应用程序响应延迟较高，影响用户体验。因此，随着用户数目不断增加，数据量不断变大，关系型数据库逐渐无法满足应用程序这种高并发的需求。于是，分布式内存缓存系统便顺势而生。而本文所主要研究的 Memcached 正是最流行的分布式内存缓存系统之一。

Memcached 最初是由 Brad Fitzpatrick 在 2003 年为自己的网站 LiveJournal 开发的缓存程序^[26]。Memcached 将用户经常需要访问的数据存放在内存中作一个缓存。当用户访问这部分数据的时候，就可以直接从 Memcached 服务器的内存中读取，而不需要再访问远程数据库。这样不仅可以加速这部分数据的访问效率，还降低了远程数据库的访问压力。随着 Memcached 不断完善，现在，Memcached 已经被很多大型互联网公司所使用，比如 YouTube^[7]，Facebook^[8] 和 Twitter^[9] 等。

近年来，随着硬件工艺技术的不断进步，内存容量不断提高，内存价格不断下跌，持续地刺激着开发人员在内存技术上有所作为。另外，64 位操作系统

的普及也使得服务器支持更大的地址空间。这些都给分布式内存缓存系统的进一步发展创造了条件。

2.1.2 Memcached 的工作原理

Memcached 是一个自由、开源、高性能、分布式的内存对象缓存系统。Memcached 用于加速基于数据库的应用程序的响应速度，降低数据库的负载。如图1.1，Memcached 作为数据库的缓存，工作在应用程序和数据库之间。如果用户请求的数据已经缓存在 Memcached 中，应用程序可以直接从 Memcached 中取得这部分数据，返回给用户，而不需要从远端数据库中读取。

使用 Memcached 有两个好处：第一，由于 Memcached 一般部署在离应用程序更近的位置，而且 Memcached 的数据都保存在内存中，因此从 Memcached 中读取数据将比从远端数据库中读取所消耗的时间短很多。这样就可以加速应用程序的响应速度，优化性能。第二，由于上述过程不需要涉及到数据库，因此就降低了数据库的负载，使得应用程序可以支持更高的并发量。

Memcached 用 HashMap 来存储写入的键值对 (key-value map)。用户在向 Memcached 写入数据时，可以指定数据的有效期 (exptime)。Memcached 对过期数据的处理方式是惰性删除：Memcached 不会主动地去检测数据是否过期，而是在访问到这个数据的时候才会去检测它的过期时间。另外，除了在写入数据时指定过期时间，用户还可以通过客户端发送清除数据 (*flush_all*) 的命令来批量地让所有数据过期。Memcached 对清除数据命令的处理方式也是惰性删除。

上面是 Memcached 对过期数据的处理方式。但是内存作为一种存储介质，空间是有限的。如果内存被未过期的数据占满，Memcached 需要置换掉较旧的数据来为新写入的数据腾出空间。Memcached 采用了最近最少使用 (least recently used, LRU) 算法^[27] 来保证新数据一定有空间进行存储。在写入新数据时，如果发现已分配的空间已经被未过期的数据占满，且无法再分配新空间时，Memcached 就会利用 LRU 算法将最近最少使用的数据删除，在这片内存空间写入新数据。

2.1.3 Memcached 的内存管理

作为一个内存缓存系统，Memcached 将所有的数据都存放在内存中。而内存相比硬盘而言还是更加稀少而且昂贵的。因此，研究如何高效利用内存，减少内存碎片，对于 Memcached 是一个巨大的挑战。

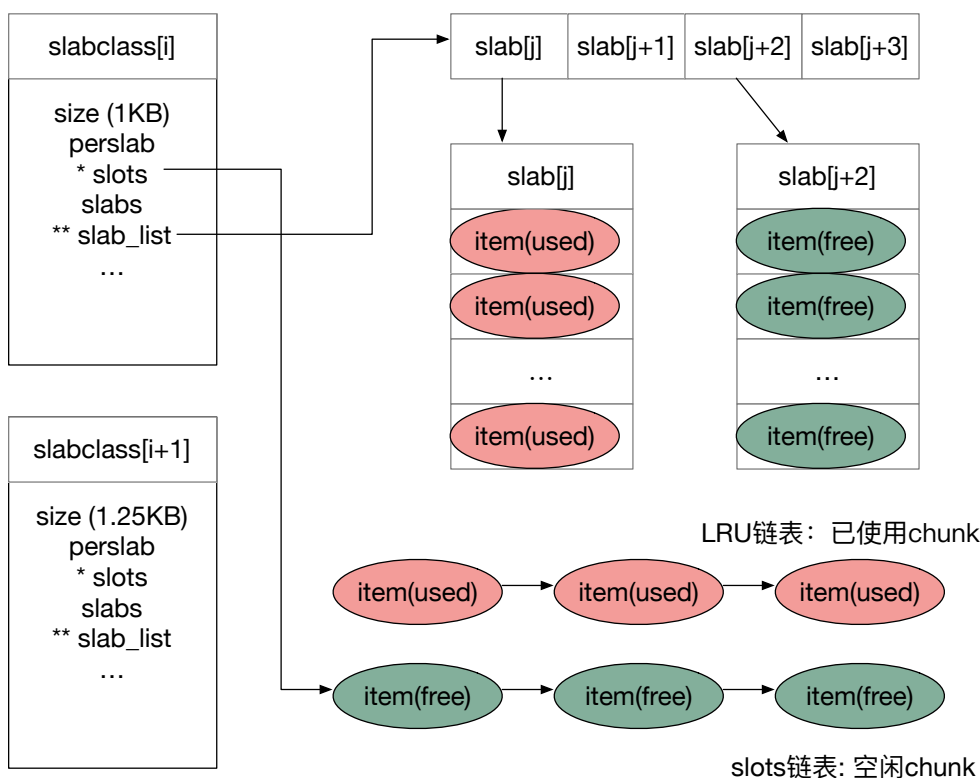


图 2.1 Memcached 内存管理示意图

如图2.1所示，Memcached 的内存管理机制主要涉及到四个重要的概念：

- **slabclass**: **slabclass** 是 Memcached 进行内存管理的基本单位。每个 **slabclass** 主要由多个具有相同 **chunk** 大小的 **slab**，一个空闲链表 **slots** 和一个 **LRU** 链表构成。
- **slab**: 每个 **slab** 由多个相同大小的 **chunk** 构成。当 **slabclass** 的所有 **slab** 都占满后，**slabclass** 可以向系统申请新的 **slab**，然后将该 **slab** 划分成相同大小的 **chunk**。因此，可以说 **slab** 是申请内存空间的最小单位。
- **chunk**: **chunk** 是 Memcached 内存中最小的存储单位。Memcached 会根据 **item** 的大小找到最适合该 **item** 的 **chunk**，然后写入进去。
- **item**: Memcached 中用户存储的数据对象。大小不固定。

四者之间的关系是：每个 **item** 写入到一个 **chunk** 中。多个相同大小的 **chunk** 构成一个 **slab**。多个基本 **chunk** 大小相同的 **slab** 构成一个 **slabclass**。如果两个 **slab** 属于同一个 **slabclass**，则他们的 **chunk** 大小相同；如果两个 **slab** 分别属于不同的 **slabclass**，则他们的 **chunk** 大小不同。形象地说，如果说 **slabclass** 是一个作

业本，slab 就是本子中的页，而 chunk 就是每一页中的格子。我们有多多个不同格子大小的作业本，但是一个作业本中的每一页的每一个格子的大小都是相同的。

每一个 slabclass 都有一个 size 属性。这个 size 就是该 slabclass 中每一个 chunk 的大小。而不同的 slabclass 的 size 是不同的。这样不同的 slabclass 所产生的 chunk 的大小就是不同的，可以供不同大小的 item 选择。当然，这个 size 不是随机生成的大小。不同的 slabclass 的 size 的差别由参数递增因子（growth factor）决定。在 Memcached 中，递增因子默认值为 1.25。也就是说，如果最小的 chunk 大小为 1KB，那么其他的 chunk 大小依次为 1.25KB，1.5625KB（需要取整，此处方便演示），以此类推。为了满足不同用户的需求，递增因子和最小的 chunk 大小都是可以自行配置的。

当用户要向 Memcached 写入一个 item 时，Memcached 会选择恰好大于等于该 item 大小的 chunk 进行写入。比如，我们有一个大小为 1.21KB 的 item，由于 1KB 的 chunk 无法容纳该 item，而 1.5625KB 大小对于该 item 又过于浪费，因此 Memcached 会选择一个 1.25KB 大小的 chunk 写入该 item。虽然这种方式会产生 0.04KB 的内存碎片，但是已经尽可能地减少内存浪费了。

2.1.2提到，Memcached 采用 LRU 算法来进行新旧数据的置换。如图2.1所示，每一个 slabclass 有一个 slots 链表和一个 LRU 链表。slots 链表是空闲的 chunk 链表。而 LRU 链表是该 slabclass 已经使用的 chunk 链表。每使用一个空闲的 chunk，都会将该 chunk 从 slots 链表移除，并加入到 LRU 链表的相应位置。每当一个已写入的 item 被访问时，都将会根据 LRU 算法更新其在 LRU 链表中的相应位置，以保证 LRU 链表从尾到头的权重是越来越高的。

假设 Memcached 收到了一个写入数据的指令，那么一次真实的内存分配步骤如下：

1. 首先根据 item 的大小，选择合适的 slabclass，准备写入数据。
2. 然后从选定的 slabclass 的 LRU 链表尾部开始向前遍历，查看是否有已经过期的 item。如果有，就直接利用这个 item 所占空间；否则继续下一步。
3. 如果上面没有找到过期的 item，则尝试从该 slabclass 的 slots 链表中获取空闲的 chunk。
4. 如果上面没有找到空闲的 chunk，则该 slabclass 尝试申请新的 slab。如果申请成功，新的 slab 就可以划分成多个空闲 chunk，然后拿出一个写

入该 item。

5. 如果上面申请 slab 失败，即没有内存可供分配，此时只能置换掉旧的 item。Memcached 将把该 slabclass 的 LRU 链表尾部的一个 item 置换成新的 item。

由此可见，Memcached 写入新数据的基本思路是优先利用已分配的无用空间，然后考虑分配新空间，最后在无可奈何的情况下置换掉旧数据。

从上面的内存分配流程中可以看出，slabclass 获取 slab 是有竞争关系的。不同的 slabclass 的 slab 数目根据 Memcached 使用情况是不同的，而 Memcached 总的内存大小是一定的，那么这种竞争就会造成一个问题。考虑一个实际使用时的情况，刚开始使用 Memcached 时，应用程序的数据大小都比较小，比如 1KB。这样就会导致 size 为 1KB 的 slabclass 非常大，而其他的 size 的 slabclass 非常小。此时，如果由于业务变更等原因，该应用程序的数据都变大 10 倍，变成 10KB。但是，size 为 10KB 的 slabclass 非常小，就会导致该 slabclass 频繁的置换旧数据，Memcached 命中率降低，系统延迟增大，系统性能下降。为了解决这个问题，Memcached 提供了内存页重分配的功能，该功能可以将原来 size 为 1KB 的 slabclass 中的 slab 重新分配给 size 为 10KB 的 slabclass。用户可以选择手动或者让 Memcached 自动进行内存页重分配。如果用户开启自动内存页重分配，Memcached 就可以根据 slabclass 中发生 LRU 置换的频率，判断该 slabclass 的 slab 数是供给不足还是供给过剩，然后将供给过剩的 slabclass 的部分 slab 重分配给供给不足的 slabclass。Memcached 的内存页重分配策略解决了不同 size 的 slabclass 之间的竞争问题。

2.1.4 分布式 Memcached

在这个数据量呈爆炸式增长的今天，单个节点因为内存、CPU 受限，不可能完成如此大规模的存储计算任务，通常我们必须部署一个分布式的 Memcached 才能满足应用程序存储量大、并发度高的需求。

Memcached 的设计理念就是小而强大，它是一个简单、易于部署的分布式内存缓存系统。Memcached 虽然被称为一个“分布式”系统，但是 Memcached 本身并没有任何分布式的功能。Memcached 的服务器之间不会相互通信，其所谓的“分布式”，需要依赖客户端的实现。如图2.2，客户端根据用户数据的 key 值决定该数据应该保存在哪一台服务器上。当用户读取数据时，客户端再根据用户指定的 key 值重新计算出该 key 值所对应的数据在哪台服务器上，然后向这台

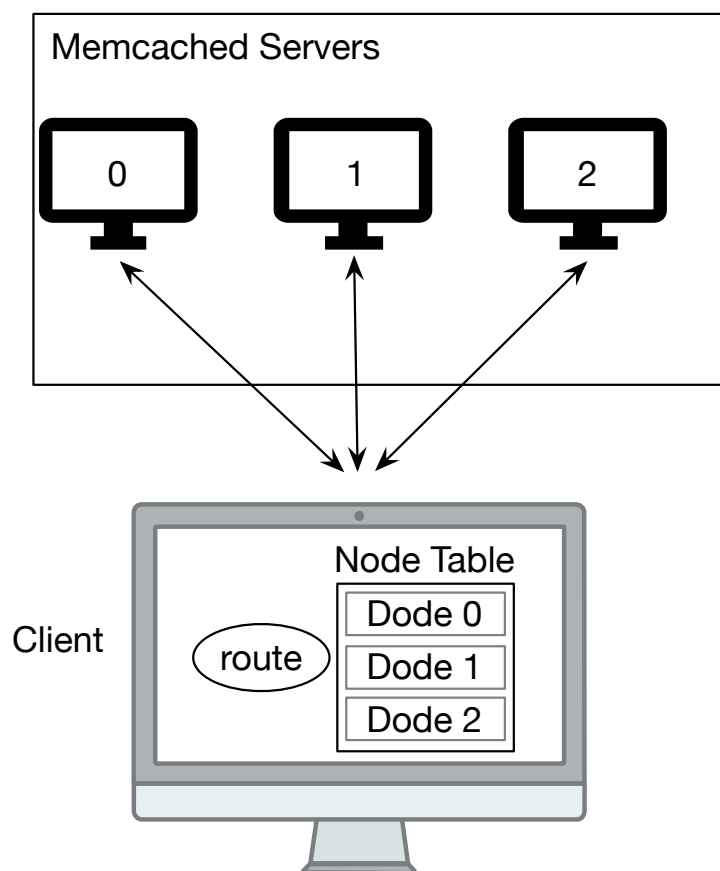


图 2.2 分布式 Memcached 架构图

服务器发送读取请求。只要客户端写入和读取数据的路由算法是一致的，就能保证用户数据被正确的写入和读出。路由算法决定着客户端要把用户数据存储在哪台服务器上，关系到服务器间的负载均衡，因此选择一个合适的路由算法对 Memcached 来说至关重要。

最简单直观的路由算法莫过于余数 Hash。余数 Hash 的算法是先对 key 值进行一次 Hash（如 SHA-1^[28]、MD5^[29] 等），然后对此 Hash 值进行取余操作，将该余数作为最终的 Hash 值。假设我们有 n 台 Memcached 节点，以 i 表示客户端为 key 选择的存储节点，则余数 Hash 可以表示成公式 2.1：

$$i = \text{Hash}(\text{key}) \% n \quad (2.1)$$

一般情况下，第一层 Hash 得到的数字比较均匀，因此客户端可以将用户数据较为均匀地映射到各个节点上，保证了负载均衡。但是，当 Memcached 集

群容量改变时，这种路由方法将会造成缓存丢失的问题。假设某个 Memcached 集群的服务器数量由 3 变成 4，某个用户的键值对 (“key0”, ”value0”) 的 key 的 Hash 值为 5。则在集群服务器数量未改变之前， $5\%3 = 2$ ，故该数据存放在 Node2 上。在服务器数量变成 4 之后，如果我们想要读取该数据，因为 $5\%4 = 1$ ，因此客户端选择从 Node1 去读取数据，结果当然是无法读出。而如果我们不想重新手动插入数据，就需要客户端做大量数据迁移，而这也是相当浪费时间的。

为了解决这个问题，Memcached 选择了一致性哈希（Consistent Hash^[30]）算法来进行数据路由。

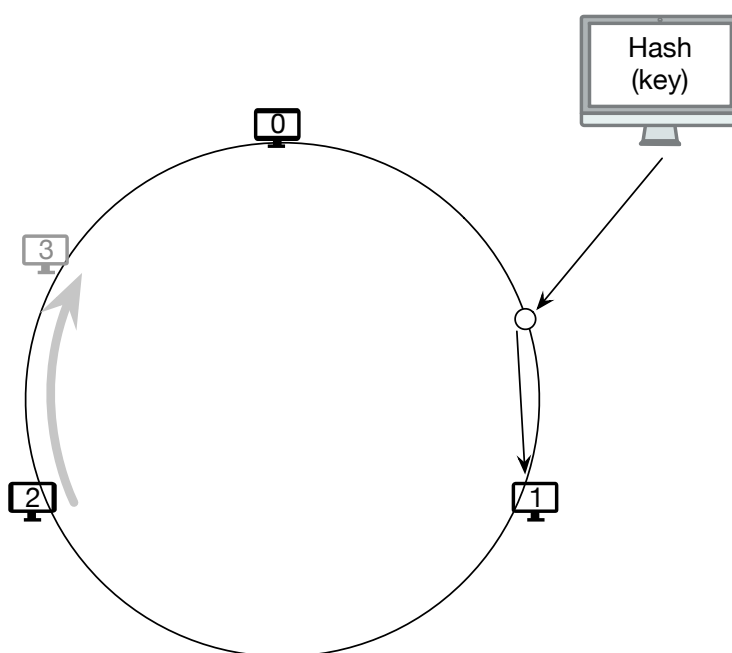


图 2.3 Memcached 一致性哈希算法示意图

如上图，一致性哈希算法将用户数据和 Memcached 节点统一映射到一个范围是 $[0, 2^{32}]$ 的环上。当用户插入一个数据时，客户端根据该数据的 Hash 值，将它存放在它在环上后面顺时针方向离它最近的节点上。也就是说，每一个节点保存着上一个节点的 Hash 值到本节点 Hash 值之间的数据。假设 Node1 的 Hash 值为 234，Node2 的 Hash 值为 567，则 Node2 保存 Hash(key) 值在 $(234, 567]$ 内的数据。

当 Memcached 集群添加一个节点 Node3 时，只有 Node2 到 Node3 这个范围内的数据会丢失，需要从 Node0 迁移到 Node3，而其他的数据都存放在原节

点，不会造成数据丢失或迁移。

由此可见，一致性哈希算法可以使 Memcached 在改变集群规模时产生尽可能少的数据丢失或者数据迁移，减少因增加或减少机器而造成系统进入降级模式的时间，提高系统的整体运行效率。

第二节 Cocytus

2.2.1 Reed-Solomon Codes

Reed-Solomon 编码是由 Reed, Solomon 两人共同在 1960 年研发出的一组编码^[31]。RS 编码通常可以表示成 $RS(n, k)$ 的形式，表示该 RS 编码的码长为 n ，有效信息长度为 k ，最多能够纠正 $n - k$ 个错误。RS 编码是目前世界上应用最广泛的纠删码之一^[32, 33]。

RS 编码是一种线性码，它的编码方式可以由一个生成矩阵 \mathbf{A} 来唯一确定。RS 编码的生成矩阵 \mathbf{A} 通常由一个单位矩阵拼上一个范德蒙矩阵 (Vandermonde Matrix)^[34] 构成。如果把要进行编码的信息表示成 \mathbf{D} ，把编码后的信息表示成 \mathbf{E} ，则 RS 编码的编码过程可以如下表示：

$$\mathbf{A} \times \mathbf{D} = \mathbf{E} \quad (2.2)$$

如果原始数据有 k 块，编码之后的数据有 n 块，校验数据有 m 块，则上式2.2可以展开如下：

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1^1 & 2^1 & 3^1 & \cdots & k^1 \\ 1^2 & 2^2 & 3^2 & \cdots & k^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1^{m-1} & 2^{m-1} & 3^{m-1} & \cdots & k^{m-1} \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{k-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{k-1} \\ p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{m-1} \end{bmatrix} \quad (2.3)$$

在上式2.3中，我们可以看到，生成矩阵 \mathbf{A} 的上半部分（前 k 行）是一个单

位矩阵，下半部分（后 m 行）是一个范德蒙矩阵。假设原始信息字长是 1 字节，则 d_i 取值范围是 $[0, 255]$ 。如果直接拿生成矩阵 \mathbf{A} 与原始数据 \mathbf{D} 相乘， p_i 可能会非常大，会出现数位溢出的情况，显然是不可以的。因此 RS 编码的运算通常是建立在一个有 2^w 个元素的有限域上的，其中 w 是字长。

对于任意给定的 n 和 k ($k < n$)，我们都可以按照上面的方法，由 k 个数据块生成一共 n 个数据块，构造一个 RS 编码。由此可见，RS 编码的扩展性非常强。

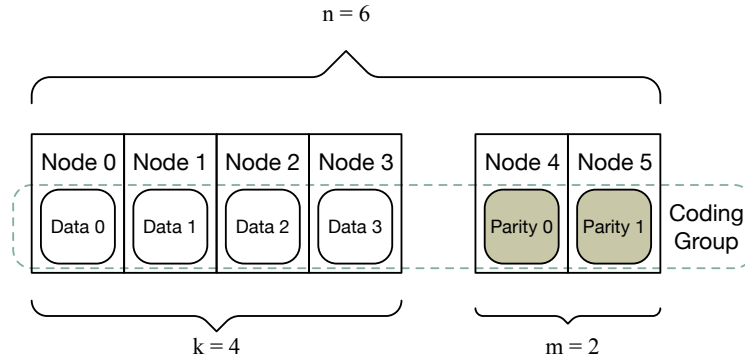


图 2.4 RS(6, 4) 编码示意图

如图2.4是 $RS(6, 4)$ 的编码示意图。在这样的一个编码组中，我们有 4 个原始数据块，2 个校验块，码长为 6。如果其中任意 1 个或 2 个块丢失，RS 编码都可以通过剩下的任意 4 个块将原始信息恢复出来。而它的存储开销仅为 $1.5X$ ，这个数字在具有双容错能力的文件系统中是非常可观的。

$$\mathbf{D} = \mathbf{A}'^{-1} \times \mathbf{E}' \quad (2.4)$$

更一般地，对于 $RS(n, k)$ 编码系统来说，如果其中任意最多 $n - k$ 个块丢失， $RS(n, k)$ 都可以根据公式2.4通过剩下的任意 k 个块将原始信息恢复出来。也就是说， $RS(n, k)$ 具有 $n - k$ 位容错能力。其中， \mathbf{E}' 是从剩余数据中任意选出来的 k 个数据块对应的矩阵， \mathbf{A}' 是生成矩阵 \mathbf{A} 与前面选出的 k 个数据块相对应的行构成的新矩阵， \mathbf{A}'^{-1} 是 \mathbf{A}' 的逆矩阵。该式2.4的推理也非常简单，可以直接由式2.2进行缩减后左右两遍同时乘上 \mathbf{A}'^{-1} 得到。实际上，RS 编码达到了相同容错能力下理论上的最小冗余。也就是说，对于采用不同编码方案且具有相同容错能力的存储系统而言，采用 RS 编码的存储系统所引入的存储开销是最小的。

虽然对于 $RS(n, k)$ 而言，丢失任意 $n - k$ 个数据块，都可以恢复出原始数据。

但是即使是只丢失 1 个数据块, $RS(n, k)$ 也需要获取 k 个数据块的全部数据才能恢复原始数据。在图 2.4 所示的 $RS(6, 4)$ 中, 假如 *Node0* 损坏, 如果我们想恢复 *Node0* 上面的数据, 我们需要获取到 *Node1*、*Node2*、*Node3* 和 *Node4* 上面的所有数据, 然后通过解码算法, 计算出原始数据。由此可见, RS 编码在数据恢复过程中需要大量的数据传输。

综上所述, 虽然 RS 编码具有扩展性好、冗余度低等优点, 但是, 它也有两大缺点:

1. 计算复杂度高。RS 编码的运算是基于有限域的, 它的编码和解码过程都需要复杂的查表操作, 因此 RS 编码的计算复杂度较高, 效率较低。
2. 恢复成本高。在数据恢复过程中, RS 编码需要获取大量数据块才能进行解码, 将原始数据恢复出来, 这个过程的网络传输开销较大, 消耗时间较长, 会降低系统整体性能。

虽然针对 RS 编码的优化方案层出不穷, 例如采用柯西矩阵来进行优化^[35], 基于二进制矩阵进行优化^[36]等, 但是采用 RS 编码所引入的各种开销还是比本文所采用的 RDP 编码要高很多。

2.2.2 Cocytus 系统

在分布式系统中, 节点失效和硬件故障被认为是一种常态^[37]。而对于内存而言, 只要机器由于某种原因重启, 就会造成数据丢失, 因此内存相比硬盘更加具有易失性。因此, 像 Memcached 这样简单地将用户数据保存在内存中, 且没有采用类似 Redis^[6] 那样的持久化机制, 很容易造成数据丢失。而当 Memcached 节点故障后, 如果没有相应的容错机制, 就需要像第一节描述的那样从远程数据库读取所需数据, 这造成的读延迟将是难以预估的, 可能会严重地损害系统性能, 降低用户体验。由此可见, 为 Memcached 添加容错能力是非常有意义的。

陈海波等人在 2017 年首次提出了将纠删码用于分布式内存缓存系统, 并在 Memcached 的基础上加入了 RS 编码, 实现了一个具有容错能力的分布式 Memcached, 即 Cocytus。Cocytus 采用的是混合编码^[38, 39]的方式, 也就是说 Cocytus 将用户数据和元数据分离, 对大小较小的元数据信息采用的是副本的方式, 而对较大的实际数据采用的是 RS 编码。这种方式就意味着如果在一个应用中, 用户的数据都是字节级别的数据, 元数据信息甚至比实际数据还要大, 那么 Cocytus 的存储冗余度将会很高。当然, 这种情况在实际情况下非常少见。

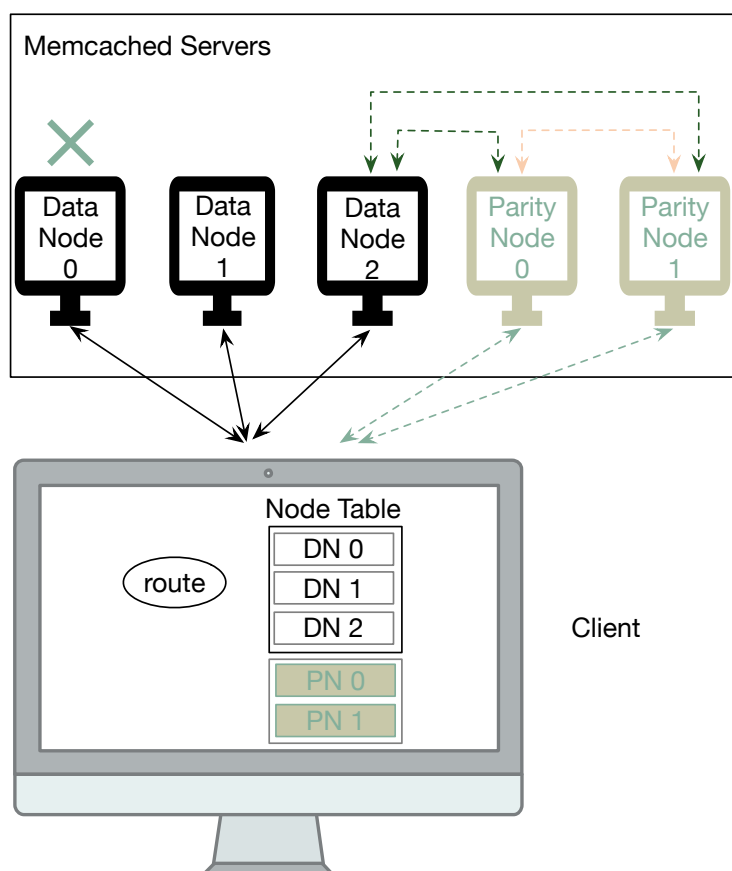


图 2.5 采用 RS(5, 3) 编码的分布式 Memcached 示意图

如图2.5是在分布式 Memcached 中采用 RS(5,3) 编码后的架构图。从图中可以看出，该采用了 RS 编码之后的分布式 Memcached 架构图与原架构图1.1最大的差别在于将原来等价的节点划分成了数据节点（Data Node）和校验节点（Parity Node）。在采用 RS(5,3) 编码的分布式 Memcached 中，有 3 个数据节点和 2 个校验节点，分别存储 RS(5,3) 编码中的原始数据和校验数据。注意本文所讨论的 RS 编码和 RDP 编码都属于系统码^[40]，也即编码之后的码字包含全部原始数据。在这个例子中，3 个数据节点存放的都是用户写入的原始数据，未经任何修改，因此用户的读取操作可以直接从数据节点中读取到原始信息，不需要任何网络传输或计算操作。

在原始的分布式 Memcached 中，节点之间不需要任何通信。而当引入了 RS 编码之后，由于需要计算和存储校验数据，同步版本号，恢复数据等操作，Cocytus 在不同的节点之间加入了大量的通信。这些通信主要发生在数据节点与

校验节点之间以及校验节点与校验节点之间。而数据节点与数据节点之间的通信主要是为了监听节点状态。

当用户要向 Cocytus 写入一个数据时，客户端将数据直接发送给数据节点，数据节点在本地存放一份原始数据，然后计算出校验值并发送到校验节点。校验节点收到后保存相应的校验数据。而当一个数据节点发生故障时，Cocytus 会从校验节点中选出一个作为该数据节点的替代节点，然后选择一个该恢复过程的 leader，由该 leader 来启动恢复流程。leader 从其他节点获取到所需的数据后，进行计算，并将计算出的结果存放到替代节点上。至此，数据恢复流程结束。注意恢复流程中的 leader 和替代节点通常是同一个节点。但也有不是的情况，例如如果有两个节点发生故障，这时 Cocytus 需要 1 个 leader 和 2 个替代节点，他们就不可能是相同的。值得一提的是，Cocytus 只会对数据节点的故障进行修复。而如果是校验节点发生故障，为了系统性能着想，Cocytus 将选择忽视该故障。当然如果这个校验节点正在作为一个数据节点的替代节点工作，Cocytus 将为该数据节点选择一个新的替代节点并恢复它上面的数据。

另外，由于节点性质发生变化，因此客户端也需要做相应的修改。如图2.5下半部分所示，客户端的路由表不仅包含数据节点，也包含校验节点。在系统正常运行情况下，客户端只会将用户写入的数据路由到数据节点。而当某一个数据节点发生故障时，客户端会将原来发送给该数据节点的数据转而发送给它的替代节点。由此可见，对客户端而言，其哈希算法中的节点数目是不变的。在某个数据节点发生故障时，替代节点将取代该数据节点原来的位置和功能，客户端只需要更改它要发送数据的目的节点即可。

Memcached 是基于事件驱动的。它内部实现了一个状态机，而这个状态机正是 Memcached 处理逻辑的核心，简单的说，Memcached 就是根据不同的状态去完成不同的任务。一个任务可能由多个状态机节点构成。例如在一个最简单的读操作过程中，节点可能会经历待机，解析命令，读取数据等多个状态节点。而 Cocytus 由于引入了 RS 编码，因此需要扩展原来的状态机，增加更多的状态节点。例如，由于校验节点与数据节点功能不同，因此需要增加校验节点的各种状态，比如等待数据节点传输数据的状态、恢复数据时向数据节点请求数据的状态等。

2.2.3 Cocytus 相关研究

仿佛 Cocytus 点燃了一把火，最近关于用纠删码为分布式内存缓存系统增加容错能力的相关研究如雨后春笋般涌现。例如，BCStore^[41] 和 GU^[42] 采用批处理的方式来减少数据写入分布式内存缓存系统时的带宽消耗，提升写入数据的效率。WPS^[43] 基于负载感知的方法来解决负载不均的问题，提升集群整体的访问性能。针对 Cocytus 不适用于小数据的情况，MemEC^[44] 采用全编码的方式，将用户数据和元数据都用纠删码进行编码，使得分布式 Memcached 即使全是小数据，也能高效存储。

本文所做的工作也是基于开源的 Cocytus^[45] 的。

第三节 本章小结

本章主要介绍了与本文相关的背景知识。首先介绍了一种常见的分布式内存缓存系统 Memcached，包括它的发展，工作原理，内存管理，以及它是如何实现分布式的。然后介绍了本文所依赖的开源软件 Cocytus，包括它所采用的 RS 编码，以及它是如何把 RS 编码引入到 Memcached 中的。最后介绍了近期关于容错 Memcached 的一些相关的研究工作。

第三章 高效容错 Memcached 实现

第一节 基于 RDP 编码的分布式 Memcached 系统架构

3.1.1 RDP 编码的编解码原理分析

第2.2.1节提到，RS 编码的计算依赖于有限域，一次乘法运算要经过一次查表，会降低分布式 Memcached 的性能。因此，本文提出使用更加高效的 RDP 编码^[20] 来代替 RS 编码进行容错。RDP 编码的编码和解码操作都只需要简单的 XOR 运算，而不涉及到复杂的有限域，因此它的运算性能较高，可以提高分布式 Memcached 的性能。

RDP 编码与 EVENODD 编码^[46] 类似，都属于阵列码^[47] 的范畴。阵列码是一种常见的容错编码，它将原始数据划分成大小一致的数据块，将这些数据块排布成二维阵列，然后对阵列中的数据块进行编码运算得到校验阵列，以此达到容错的目的。

RDP 编码的编码原理分析

RDP 编码是由 Corbett 等人在 2004 年提出的一种阵列码。RDP 编码阵列由一个控制参数 p 决定。该参数 p 必须是一个大于 2 的素数。但是我们通过删除一些列，并假设其值为 0 来达到缩减 RDP 编码的目的。

一个 RDP 编码阵列由 $p+1$ 列， $p-1$ 行组成。通常情况下，将 RDP 编码应用到分布式系统中时，会将 $p+1$ 列分别映射到 $p+1$ 个节点上。在这 $p+1$ 个节点中，有 $p-1$ 个数据节点存放用户的原始数据，还有 2 个校验节点存放原始数据的校验数据。然后，每个节点上的数据会被划分成最小的编码单元 (*unit*)。每个 *unit* 又被划分成 $p-1$ 个大小相同的数据块（对应 RDP 编码阵列中的 $p-1$ 行）。2 个校验节点分别称为行校验节点和对角校验节点。行校验节点存放不同列相同行的原始数据块的异或值。对角校验节点存放不同列相同对角线的数据块（包括原始数据块和校验数据块）的异或值。

如图3.1所示是控制参数 p 为 5 时的 RDP 编码阵列示意图。该图有 6 个节点组成，其中 4 个数据节点，2 个校验节点。每个节点中的 1 个 *unit* 被划分成 4 个

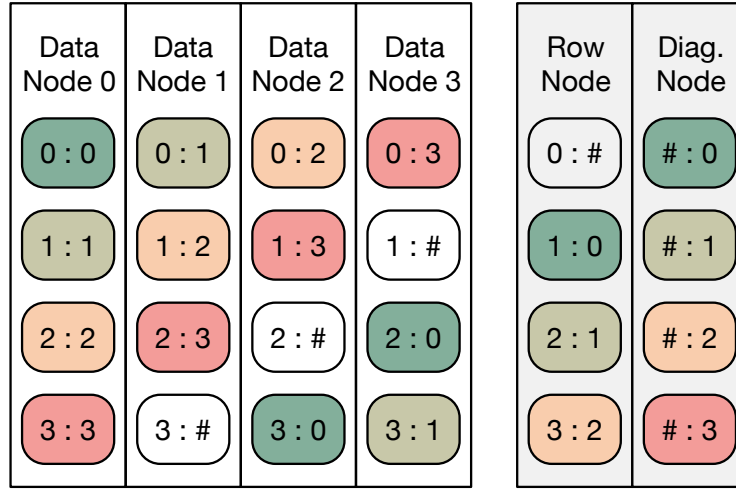


图 3.1 RDP(6,4) 编码示意图

相同大小的数据块。每个数据块都属于某一行，某一对角线。而图中每个数据块中的数字就代表着该数据块属于哪一行，哪一对角线，# 表示该数据块不属于任何行或者对角线。例如，第 1 行第 2 列（从 0 开始计数）的数据块 1:3 属于行 1，对角线 3。行校验节点的每一个数据块都是属于同一行的各个数据节点的数据块的异或值。例如，数据块 1:0 是由数据块 1:1、数据块 1:2、数据块 1:3 和数据块 1:# 通过异或运算得到的。对角校验节点的每一个数据块都是属于同一对角线的各个数据节点和行校验节点的数据块的异或值。在该图中，同一颜色的数据块属于同一条对角线。例如，数据块 #:1 是由数据块 1:1、数据块 0:1、数据块 3:1 和数据块 2:1 通过异或运算得到的。值得一提的是，对于一个 $(p-1) \times p$ 的阵列来说，应该有 p 条对角线，而 RDP 编码只用到了其中的 $p-1$ 条对角线，剩下的 1 条（即图中未上色）并未使用。

从上面的分析中，我们不难推出 RDP 编码的数学公式。假设以 $d_{i,j}$ 表示第 i 行，第 j 列的数据块（ $0 \leq i \leq p-2$ 且 $0 \leq j \leq p$ ），可以看出当 $j = p-1$ 时，即 $d_{i,p-1}$ 表示行校验数据块；当 $j = p$ 时，即 $d_{i,p}$ 表示对角校验数据块。注意这里的 i 和 j 和图 3.1 中数据块中的数字是不同的含义。因此，RDP 编码行校验数据块的计算公式为：

$$d_{i,p-1} = \bigoplus_{j=0}^{p-2} d_{i,j} \quad (0 \leq i \leq p-2) \quad (3.1)$$

RDP 编码对角校验数据块的计算公式为：

$$d_{i,p} = \bigoplus_{j=0}^{p-1} d_{(i-j)\%p,j} \quad (0 \leq i \leq p-2) \quad (3.2)$$

注意这里当 $(i-j)\%p$ 的值为 $p-1$ 时，RDP 阵列中这一列并没有对应的数据块参与该对角校验数据块的运算。例如 $d_{1,5} = d_{1,0} \oplus d_{0,1} \oplus d_{3,3} \oplus d_{2,4}$ ，其中并没有 $d_{4,2}$ ，因为阵列中并没有第 4 行（从 0 开始计数）。

RDP 编码的解码原理分析

RDP 编码是一种双容错编码方案。它最多可以容忍两个节点出现故障。下面将分别讨论 1 个节点和 2 个节点出现故障的情况下的修复方案。

如果 RDP 编码中有 1 个节点出现故障，我们需要修复这 1 列的 $p-1$ 个数据块：

1. 如果该节点是数据节点，根据异或运算的性质 $A \oplus A = 0$ 和公式 3.1 可知，该数据节点的数据可由其他数据节点数据和行校验节点数据进行异或得到。因此，数据节点 k 的第 i 行的数据块可由下面的公式得出：

$$d_{i,k} = \bigoplus_{j=0}^{p-1} d_{i,j} \quad (0 \leq i \leq p-2, \quad j \neq k) \quad (3.3)$$

2. 如果该节点是校验节点，则该节点的数据可以根据公式 3.1 或公式 3.2 重新计算得出。

如果 RDP 编码中有 2 个节点出现故障，我们需要修复这 2 列的 $2(p-1)$ 个数据块。

从上面 RDP 编码的编码方式，如图 3.1，我们知道一个 RDP 编码阵列有 $p-1$ 个行校验组和 $p-1$ 个对角校验组。每个编码组包含 p 个小块。例如数据块 0:0、数据块 0:1、数据块 0:2、数据块 0:3 和数据块 0:# 属于同一个行校验组 0；数据块 1:1、数据块 0:1、数据块 3:1、数据块 2:1 和数据块 #:1 属于同一个对角校验组 1。一个编码组中的任何 1 个数据块丢失，都可以由该校验组组中的其他 $p-1$ 个数据块进行异或得出（注意有些数据块属于 2 个校验组，如数据块 1:2 既属于行校验组 1，又属于对角校验组 2；而有些数据块只属于 1 个校验组，如数据块 1:# 只属于行校验组 1，而不参与对角校验的生成）。

为了恢复 2 个节点的故障，首先我们要从这 2 个节点丢失的 $2(p-1)$ 个数据块中找出这样的数据块，在这丢失的 $2(p-1)$ 个数据块中，只有该数据块参与

了某一个校验组的计算。根据该数据块所属的校验组恢复出该数据块。然后再从剩下的 $2p-3$ 个数据块中找出具有上述特征的数据块，同理恢复出这个数据块。随后再从剩下的 $2p-4$ 个数据块中找出具有上述特征的数据块，同理恢复出这个数据块。依此循环，直到所有的 $2(p-1)$ 个数据块都被恢复出来。

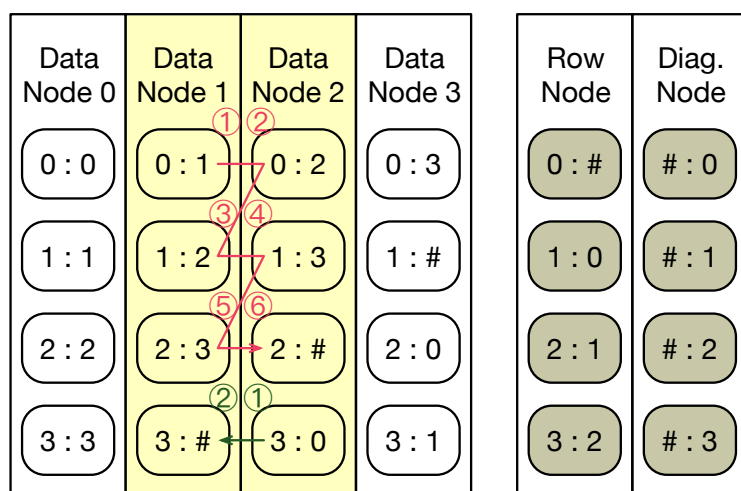


图 3.2 RDP(6,4) 双故障解码示意图

如图3.2所示是 $p=5$ 时的 RDP 编码的双故障解码示意图。在这个例子中，数据节点 1 和数据节点 2 发生故障，需要进行修复。通过观察，我们发现在这 8 个数据块中，只有数据块 0:1 参与了对角校验组 1 的计算，只有数据块 3:0 参与了对角校验组 0 的计算。因此，在第一轮我们首先修复这两个数据块。数据块 0:1 可以由同对角校验组的数据块 1:1、数据块 3:1、数据块 2:1 和数据块 #:1 通过异或运算得出。数据块 3:0 可以由同对角校验组的数据块 0:0、数据块 2:0、数据块 1:0 和数据块 #:0 通过异或运算得出。然后，我们发现由于数据块 0:1 和数据块 3:0 已经得到修复，因此在剩下的 6 个数据块中，只有数据块 0:2 参与了对角校验组 2 的计算，只有数据块 3:# 参与了对角校验组 3 的计算。因此，在第二轮我们可以修复这两个数据块。数据块 0:2 可以由同行校验组的数据块 0:0、数据块 0:1、数据块 0:3 和数据块 0:# 通过异或运算得出。数据块 3:0 可以由同行校验组的数据块 3:3、数据块 3:#、数据块 3:1 和数据块 3:2 通过异或运算得出。随后，由于数据块 0:2 得到修复，我们可以根据对角校验组 2 来修复数据块 1:2。随后，我们就可以依次修复数据块 1:3、数据块 2:3 和数据块 2:#。至此，数据节点 1 和数据节点 2 的所有 8 个数据块都得到了修复。

3.1.2 基于 RDP 编码的分布式 Memcached 系统架构

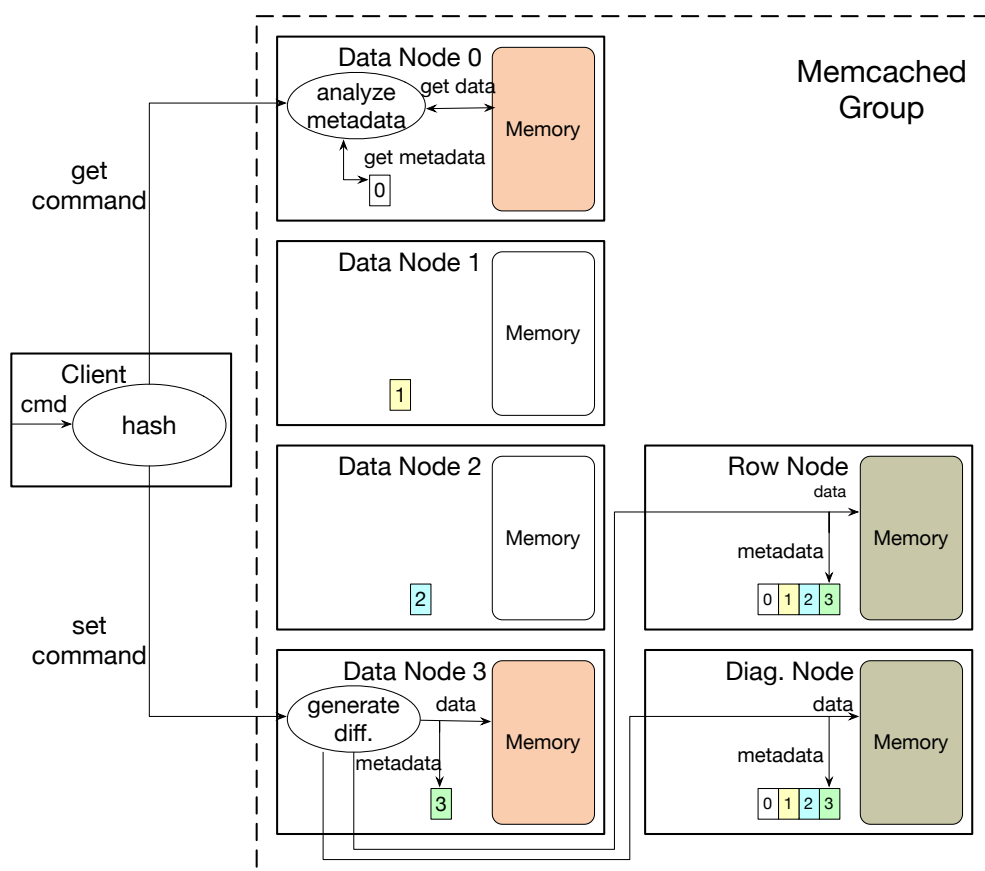


图 3.3 基于 RDP(6,4) 编码的分布式 Memcached 系统架构

如图3.3是控制参数 p 为 5 时，基于 RDP 编码的分布式 Memcached 系统架构。这个 Memcached 服务器集群一共有 6 个 Memcached 节点。这 6 个节点有 4 个数据节点，1 个行校验节点和 1 个对角校验节点。这 6 个节点构成一个分布式 Memcached 组。如果一个 Memcached 集群有很多台机器，需要将每 $p+1$ 台划分成一个 Memcached 组。

在该分布式 Memcached 中，每个节点中的内存空间被划分成同等大小的 *unit*（默认大小为 4KB）。用户写入的数据就存放在这些 *unit* 中。每个节点上的不同 *unit* 被从 0 开始递增地赋一个 *unit* 标号。各个节点上具有相同 *unit* 标号的 *unit* 构成一个编码组。RDP 编码的编码和解码操作就在一个个的编码组内进行，不会跨越编码组。

除了用户的实际数据外，分布式 Memcached 还需要保存这些数据的元数据

信息。图3.3中带有数字的矩形表示各个节点的元数据。元数据信息包括写入对象的大小、过期时间、内存地址等。元数据在分布式 Memcached 中的存储方式是副本方式。即数据节点存储各自节点上数据对象的元数据，校验节点存储所有数据节点上数据对象的元数据。如图中数据节点 1 内只有 1 个矩形表示自身存储的数据对象的元数据，而行校验节点内有 4 个矩形表示 4 个数据节点存储的数据对象的元数据。

在该分布式 Memcached 架构中，有两种基本操作： $set(key, value)$ 和 $get(key)$ 。 $update(key, value)$ 和 $set(key, value)$ 共用一套写入流程。我们称用户读写的对象为 $item$ 。这两种基本操作都需要经过用户编写的客户端。

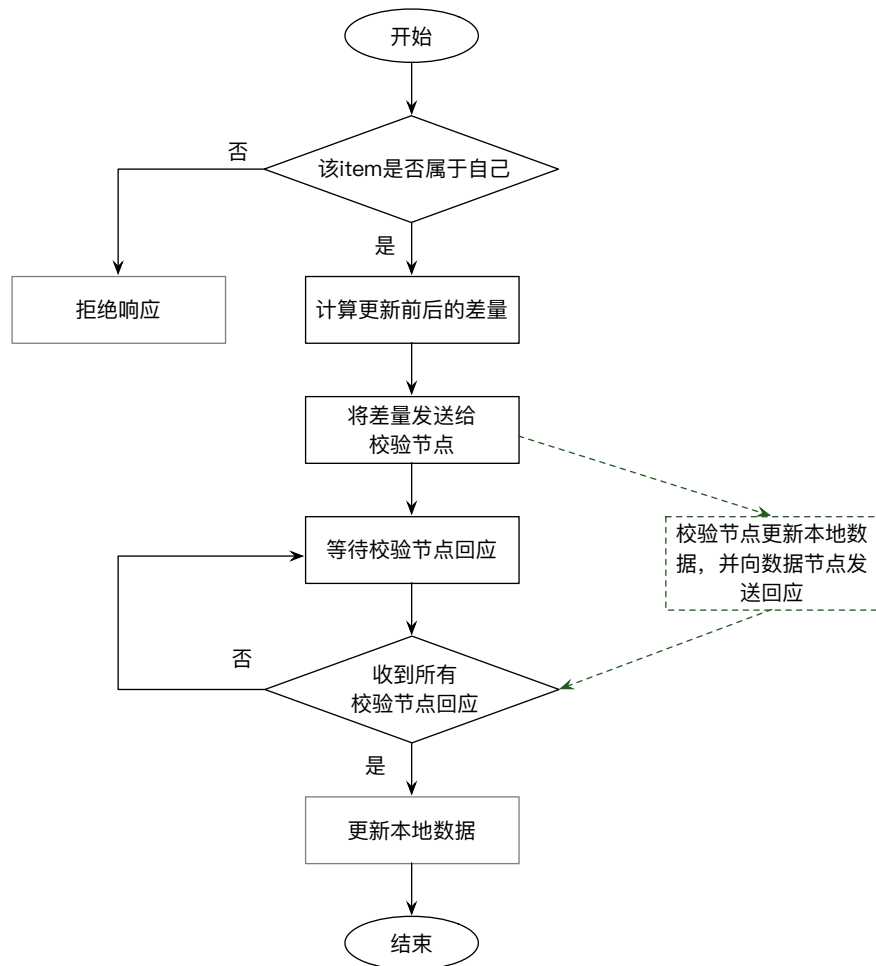


图 3.4 分布式 Memcached 的写数据流程图

如图3.4所示是分布式 Memcached 的常规写数据流程图。图中实线部分是数

据节点的流程，虚线部分则有校验节点的参与。一次简单 *set* 操作由用户向客户端发送写数据请求，客户端根据该 *item* 的 *key* 的哈希值来判定这个 *item* 应该发送给哪个数据节点。数据节点收到来自客户端的写数据请求后，首先判断这个 *item* 是否应该发送给自己。如果这个 *item* 不属于自己，可以拒绝接收。如果这个 *item* 属于自己，数据节点计算两个校验节点因为这次更新所产生的差值，并分别发送给两个校验节点。如果是新写入的 *item*，则原值是 0，差值就是新值。校验节点收到来自数据节点的差值后，更新本地的相关数据并向数据节点发送回应。数据节点收到所有校验节点的回应后，更新本地数据。当然，数据节点和校验节点在更新数据的同时，也会对相应的元数据进行更新。关于实际的数据更新算法，将在第二节详细介绍。

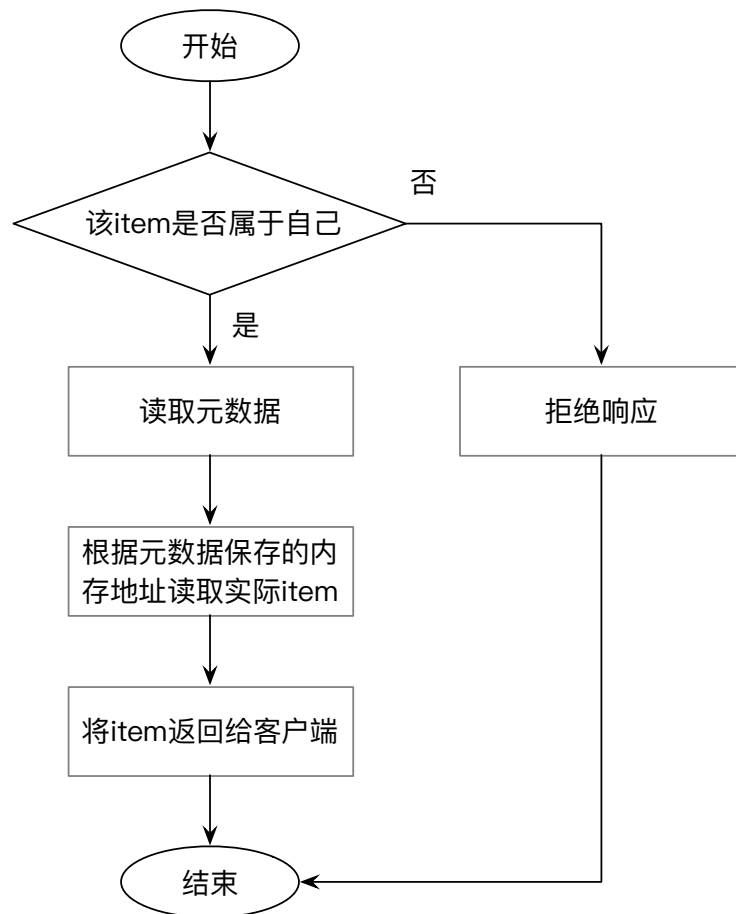


图 3.5 分布式 Memcached 的读数据流程图

一次 *get* 操作由用户向客户端发送读数据请求。与写数据类似，客户端根据

要读取的 *item* 的 *key* 的哈希值来判断应该从哪个数据节点读取这个 *item*。然后如图3.5，数据节点收到来自客户端的读数据请求后，同样判断这个 *item* 是否在自己节点上存储。如果这个 *item* 不属于自己，则直接拒绝响应。如果这个 *item* 属于自己，首先读取这个 *item* 的元数据，根据元数据判断这个 *item* 的实际存储位置。最后再从内存中取出这个 *item* 的值，返回给客户端。

值得一提的是，由于用户的不确定性，用户写入的 *item* 大小不可能都恰好与 *unit* 的大小相同。在该分布式 Memcached 中，一个 *unit* 中可以写入多个 *item*，一个 *item* 也可以跨越多个 *unit*，但是由于 C 语言的特性，为了保证 CPU 访问内存的效率，需要保证这些 *item* 在写入时是 16 字节对齐的。也就是说，即使一个 *item* 不足 16 字节，在写入到分布式 Memcached 中时，也会占据 16 字节的位置。而如果一个 *item* 的实际大小是 55 字节，那么在写入到分布式 Memcached 中时，将会占据 64 字节的位置。

第二节 基于 RDP 编码的增量数据更新

3.2.1 基于 RDP 编码的增量数据更新

本节主要讨论如何将增量编码^[48]的思想应用到 RDP 编码上。

首先，我们看一下在没有增量编码的情况下，基于 RDP 编码的分布式 Memcached 是如何进行数据更新的。由3.1.1，我们知道 RDP 编码的行校验节点的值是各个数据节点的值异或。因此，如果一个数据节点要更新它所存储的数据，那么所有数据节点都需要将该编码组内的值发送到行校验节点，然后由行校验节点重新计算其上的行校验数据并进行更新。RDP 编码的对角校验节点的值是由所有的数据节点和行校验节点上的数据计算得出的。因此，如果一个数据节点要更新它所存储的数据，那么所有的数据节点和行校验节点都需要将该编码组内的值发送到对角校验节点，然后由对角校验节点重新计算其上的对角校验数据并进行更新。当然，对角校验节点上的数据可以由行校验节点计算得出后直接发送到对角校验节点，这样可以减少部分数据传输。即使如此，我们依然认为这种数据更新的方式会产生大量的数据传输，会极大地影响数据更新的效率，降低系统整体性能。因此，本文采用了增量编码的方式来对 RDP 编码的数据更新进行优化。

增量编码的核心思想在于，在数据更新的时候，数据节点并不是简单的把更新后的数据直接发送给校验节点，而是把更新后的数据和更新前的旧数据的

差量（异或值）发送给校验节点。对于校验节点而言，只有数据更新的数据节点的数据发生了变化，其他数据节点的数据还是原来的数据，因此只需要根据该差量就可以计算出更新后自身应该保存的新校验的值。以此，来达到降低数据更新时数据传输的目的。

举个例子，假设 r 是由 d_0 、 d_1 、 d_2 和 d_3 四个数据通过异或计算得出的结果，即

$$r = d_0 \oplus d_1 \oplus d_2 \oplus d_3 \quad (3.4)$$

此时， d_1 发生更新变成了 d_1' 。假设 Δ_{d_1} 表示 d_1 和 d_1' 的差量，根据下面的推导：

$$\begin{aligned} r' &= d_0 \oplus d_1' \oplus d_2 \oplus d_3 \\ &= d_0 \oplus d_1 \oplus \Delta_{d_1} \oplus d_2 \oplus d_3 \\ &= d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus \Delta_{d_1} \\ &= r \oplus \Delta_{d_1} \end{aligned}$$

我们可以得出如下公式3.5：

$$r' = r \oplus \Delta_{d_1} \quad (3.5)$$

因此，要把 r 更新为新的 r' ，只需要知道 d_1 与其更新后的 d_1' 的差值即可。由此可见，采用差量编码，可以极大地降低数据更新时的数据传输。

如图3.6，我们以 $RDP(6,4)$ 编码为例，展示差量编码是如何应用到 RDP 编码中的。首先，我们假设数据节点 1 的 1 个 *unit* 即将发生更新操作。这个 *unit* 的 4 个数据块 $d_{i,1}$ ($i \in \{0,1,2,3\}$) 都会发生变化。

我们用 Δ_{node1} 、 Δ_{row} 和 Δ_{diag} 分别表示数据节点 1，行校验节点和对角校验节点更新前后的差值。然后我们把数据节点 1 中该 *unit* 更新前后的差量表示为：

$$\Delta_{node1} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} \quad (3.6)$$

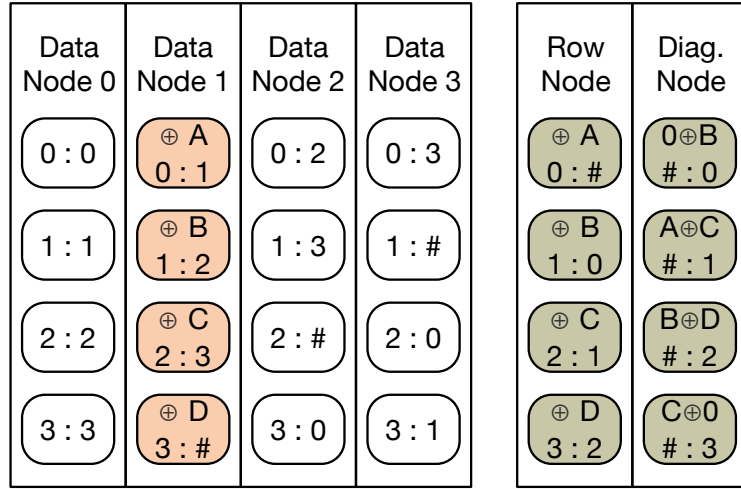


图 3.6 RDP(6,4) 差量编码示意图

那么，这个 *unit* 的变化可以表示成：

$$\begin{bmatrix} d_{0,1} \\ d_{1,1} \\ d_{2,1} \\ d_{3,1} \end{bmatrix} \mapsto \begin{bmatrix} d_{0,1} \oplus A \\ d_{1,1} \oplus B \\ d_{2,1} \oplus C \\ d_{3,1} \oplus D \end{bmatrix} \quad (3.7)$$

在 RDP 编码中，行校验节点的各个数据块是各个数据节点上对应行的数据块的异或，根据公式3.5，我们可以如下式更新行校验节点上对应的 *unit*：

$$\begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{bmatrix} \mapsto \begin{bmatrix} R_0 \oplus A \\ R_1 \oplus B \\ R_2 \oplus C \\ R_3 \oplus D \end{bmatrix}. \quad (3.8)$$

而 RDP 编码中对角校验节点上数据的计算稍微复杂一些。对角校验节点上对应的 *unit* 可以由数据节点 1 和行校验节点上的 *unit* 计算得出。对角校验节点上的数据块是属于同一对角线的各个数据节点和行校验节点的数据块的异或值。通过图3.1，我们看出，数据节点 1 中的 $d_{0,1}$ 参与了对角校验节点中 D_1 的计算；数据节点 1 中的 $d_{1,1}$ 参与了对角校验节点中 D_2 的计算。而 $d_{0,1}$ 在第 0 行， D_1 在第 1 行； $d_{1,1}$ 在第 1 行， D_2 在第 2 行。因此在计算对角校验节点上的数据块时，并不能直接拿数据节点 1 上的差值 Δ_{node1} 进行计算，需要先把数据节点 1 上的

差值进行旋转操作。

我们定义这个旋转操作为 $rotate(\Delta, nodeID)$ 。其中， Δ 为某个节点更新前后的差值， $nodeID$ 为该节点的 ID 。

我们发现 $d_{0,1}$ 应该旋转到第 1 行； $d_{1,1}$ 应该旋转到第 2 行； $d_{2,1}$ 应该旋转到第 3 行；而 $d_{3,1}$ 由于不参与任何对角校验节点中数据块的计算，因此不需要进行旋转。

我们用 Δ'_{node1} 表示数据节点 1 中 $unit$ 旋转后的结果，0 表示数据节点 1 中没有数据块参与对角校验节点中该数据块的计算，则：

$$\Delta'_{node1} = rotate(\Delta_{node1}, node1) = \begin{bmatrix} 0 \\ A \\ B \\ C \end{bmatrix} \quad (3.9)$$

同理，行校验节点上的差值也需要进行旋转操作才能用于对角校验节点上的数据块的计算。我们发现 R_0 应该不参与任何对角校验节点中数据块的计算，因此不需要旋转； R_1 应该旋转到第 0 行； R_2 应该旋转到第 1 行； R_3 应该旋转到第 2 行。

我们用 Δ'_{row} 表示行校验节点中该 $unit$ 旋转后的结果，0 表示行校验节点中没有数据块参与对角校验节点中该数据块的计算，则：

$$\Delta'_{row} = rotate(\Delta_{row}, row) = \begin{bmatrix} B \\ C \\ D \\ 0 \end{bmatrix} \quad (3.10)$$

易证，对角校验节点上对应 $unit$ 的差值是数据节点 1 和行校验节点上对应 $unit$ 差值旋转后的结果的异或，即：

$$\Delta_{Diag} = \Delta'_{node1} \oplus \Delta'_{row} \quad (3.11)$$

因此，我们可以如下式更新对角校验节点上对应的 *unit*：

$$\begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} \mapsto \begin{bmatrix} D_0 \oplus B \\ D_1 \oplus A \oplus C \\ D_2 \oplus B \oplus D \\ D_3 \oplus C \end{bmatrix}. \quad (3.12)$$

事实上，任何数据节点上发生数据更新，只要知道了更新前后的差值，都可以很容易得出行校验节点上更新前后的差值，进而通过将该数据节点和行校验节点的差值进行旋转得到对角校验节点上更新前后的差值。这样，我们就可以在数据节点上计算出行校验节点和对角校验节点上更新前后的差值，然后直接把差值发送到对应的校验节点。校验节点收到差值后，与自身原来的数据进行异或得到更新后的值，并保存在自己的内存中。由于任何数据节点发生更新操作，都需要向两个校验节点发送数据，因此校验节点的负载相比数据节点要高一些。因此，相比数据节点将数据节点的差值发送到校验节点，然后由校验节点计算自身更新前后的差值并进行存储的方法而言，前者可以降低校验节点的负载，进而达到负载均衡的目的。

本文在实现基于差量编码的 RDP 编码时，将数据节点和行校验节点需要旋转到位置做成了如下公式3.13的一个旋转矩阵并写入配置文件。

$$RM = \begin{bmatrix} 0 & 1 & 2 & 3 & \# \\ 1 & 2 & 3 & \# & 0 \\ 2 & 3 & \# & 0 & 1 \\ 3 & \# & 0 & 1 & 2 \end{bmatrix} \quad (3.13)$$

$RM_{i,j}$ 表示数据节点或行校验节点的数据块 $d_{i,j}$ 应该旋转到位置。 $\#$ 表示忽略该数据块。

综上所述，数据节点计算校验节点的差值的算法如算法1所示：

3.2.2 分布式 Memcached 的数据一致性

由于本文将 RDP 编码引入到分布式 Memcached 中来实现容错功能，因此分布式 Memcached 中的节点分为了数据节点和校验节点两种。一次数据更新不仅涉及某一个数据节点，还会涉及到所有的校验节点。如果一次数据更新在数据节点上更新成功了，但是在校验节点上更新失败了，这就会导致不同节点上

算法 1 基于 RDP 编码的差量数据更新算法

输入：数据节点更新前数据 $oldData$ ，数据节点更新后数据 $newData$ ，数据节点编号 $nodeID$ ，行数据节点编号 $rowID$

输出：行校验节点差值 $rowDiff$ ，对角校验节点差值 $diagDiff$

```

1:  $rowDiff \leftarrow oldData \oplus newData$ 
2: for each check node  $i$  do
3:   if  $i$  is the row check node then
4:      $rowDiff \leftarrow rowDiff$ 
5:   else
6:      $tmp1 \leftarrow rotate(rowDiff, nodeID)$ 
7:      $tmp2 \leftarrow rotate(rowDiff, rowID)$ 
8:      $diagDiff \leftarrow tmp1 \oplus tmp2$ 
9:   end if
10: end for
11: return ( $rowDiff$ ,  $diagDiff$ )

```

数据不一致的问题。为了解决数据一致性问题，最简单的方法就是两阶段提交协议（2PC）^[49]了。但是使用两阶段提交协议，每次数据更新都需要两轮的消息通信，会导致延迟增大。本文采用了与 Cocytus 类似的捎带更新的方法来保证分布式 Memcached 中的数据一致性。

如图3.7所示就是本文在分布式 Memcached 中采用的捎带更新策略的示意图。该图以 $RDP(6,4)$ 编码为例，展示本文是如何维护分布式 Memcached 中的数据一致性的。

首先，在基于 RDP 编码的分布式 Memcached 中，我们为每一次更新（包含写入）操作赋一个 id 号。这个 id 会随着更新操作次数的增长自动递增，就像一个逻辑时钟一样。在本例中，数据节点 1 进行了一次数据更新，这次更新的 id 为 6。数据节点 1 收到更新请求后，首先按照第3.2.1节所述计算两个校验节点所需的差量，然后分别发送给两个校验节点。校验节点收到数据节点 1 发送来的差量后，首先将此差量保存在缓冲区中，然后立即向数据节点 1 发送确认消息，告诉数据节点 1 自己已经接收到了其发来的更新请求。在本例中，校验节点缓冲区内包含了 4 个数据节点的缓冲差量，在收到来自数据节点 1 的差量后，缓冲区中的 id 由 5 变为 6。当数据节点 1 收到所有校验节点发送回来的确认消息后，这次更新操作就可以被视为是稳定的，因此图最下面的稳定的 id 是 6。然后数据节点 1 就可以将这次更新写入到自己的内存中了。我们将最新的稳定的 id 视为该数据节点版本号。

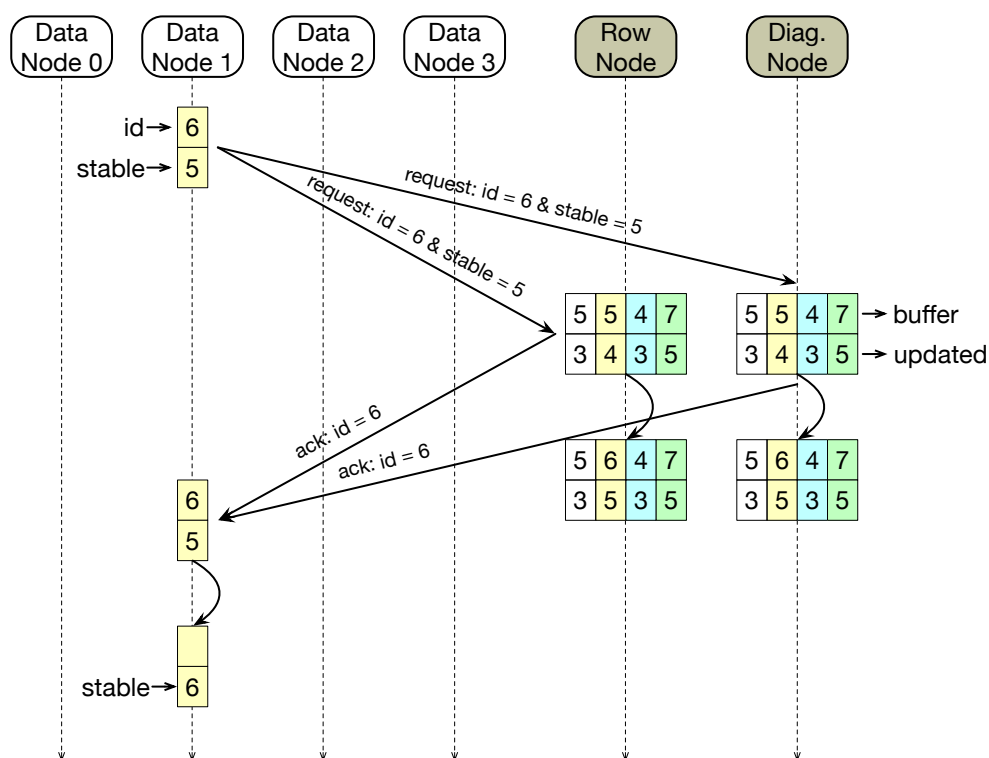


图 3.7 分布式 Memcached 的捎带更新

当数据节点 1 向校验节点发送下一次更新请求时，会捎带着这个版本号一起发送给校验节点。现在我们假设上次更新的 `id` 为 5，现在要更新的 `id` 为 6。当校验节点收到这次更新后，发现这个请求捎带了版本号为 5，说明数据节点 1 已经写入了 `id` 为 5 的更新操作，因此校验节点依次将 `id` 小于等于 5 的操作都从缓冲区写入自己的内存中。在本例中，校验节点将 `id` 为 5 的更新操作写入内存中。当故障发生时，未被所有校验节点接收或者 `id` 大于版本号的操作均会被无视。

值得一提的是，采用这种捎带更新的方式，数据节点第一次写入数据时，校验节点只是把数据放入缓冲区。而后的每一次更新，校验节点最多只会更新到上一次数据更新，最新的数据只会存放在缓冲区中。因此校验节点已更新的数据 `id` 永远比数据节点少 1。

第三节 基于 RDOR 的数据恢复优化

RDP 编码是一种双容错编码方案，因此本文所设计的基于 RDP 编码的分布式 Memcached 最多可以容忍两个节点出现故障。但是，在实际应用中，通常

两个节点同时发生故障的概率要远远低于一个节点发生故障的概率^[50]。因此，研究如何高效地恢复一个节点的故障是非常有意义的。本文采用了 RDOR 和 CRR 两种方案来对 RDP 编码单个节点发生故障时的修复过程进行优化。本章的第三节和第四节将分别介绍 RDOR 和 CRR 是如何应用到本文所设计的分布式 Memcached 的。

3.3.1 RDOR 恢复优化原理

Row-diagonal Optimal Recovery (RDOR)^[21] 是由 Xiang 等人在 2010 年提出的一种 RDP 编码的单故障优化模型。相比 RDP 编码而言，RDOR 可以减少单故障恢复过程中所需的数据量，进而减少分布式 Memcached 恢复某个数据节点时的网络传输量，加快数据恢复的速率。

如第3.1.1节所述，在传统的 RDP 编码中，要恢复单个数据节点的故障，我们需要获取到所有剩余数据节点和行校验节点的数据，然后进行异或计算得出该数据节点的数据。如图3.1所示的 $RDP(6,4)$ 编码中，如果数据节点 1 发生故障，那么我们需要将数据节点 0、数据节点 2、数据节点 3 和行校验节点的数据进行异或，才能得出数据节点 1 的数据。由此可见，传统的 RDP 编码在单故障情况下数据恢复的数据传输开销和 RS 编码是一样的。

然而，不同于 RS 编码的是，RDP 编码是一种阵列码。RDP 编码的两个校验节点中其中一个是由同行的数据块计算得出的；另一个是由同对角线的数据块计算得出的。因此，RDP 编码的大部分数据块既参与了行校验组的计算，又参与了对角校验组的计算。例如在图3.8中，数据块 1:2 既参与了行校验组 1 的计算，又参与了对角校验组 2 的计算。RDOR 就是利用了 RDP 编码的这一特性来降低单节点故障的修复过程中的数据传输量的。相比 RDP 编码的传统修复方案，RDOR 可以实现从更多的节点上读取数据，但是从每个节点上获取的数据量都更小，而且最后获取的数据总量更小。

首先，我们用一个例子来说明 RDOR 具体是如何来优化 RDP 编码的故障修复过程的。如图3.8所示是基于 $RDP(6,4)$ 的 RDOR 解码模型示意图。在这个例子中，我们假设数据节点 0 发生故障。为了恢复数据节点 0 的 4 个数据块，对于传统的 RDP 编码而言，我们需要获取数据节点 1、数据节点 2、数据节点 3 和行校验节点所有的共 16 个数据块，通过计算得出。

同样地，为了恢复数据节点 0 的 4 个数据块，对于 RDOR 恢复模型而言，我们只需要获取如图标示的 12 个数据块即可。我们可以通过数据块 0:1、数据

Data Node 0	Data Node 1	Data Node 2	Data Node 3	Row Node	Diag. Node
0:0	0:1	0:2	0:3	0:#	#:0
1:1	1:2	1:3	1:#	1:0	#:1
2:2	2:3	2:#	2:0	2:1	#:2
3:3	3:#	3:0	3:1	3:2	#:3

图 3.8 基于 RDP(6,4) 的 RDOR 解码模型

块 0:2、数据块 0:3 和数据块 0:# 计算得出数据块 0:0；通过数据块 1:2、数据块 1:3、数据块 1:# 和数据块 1:0 计算得出数据块 1:1；通过数据块 1:2、数据块 0:2、数据块 3:2 和数据块 #:2 计算得出数据块 2:2；通过数据块 2:3、数据块 1:3、数据块 0:3 和数据块 #:3 计算得出数据块 3:3。通过观察，我们可以发现，数据块 1:2、数据块 0:2、数据块 1:3、数据块 0:3 这 4 个数据块均使用了两次，既参与了某个行校验组的解码，又参与了某个对角校验组的解码。例如，数据块 0:2 既和数据块 0:1、数据块 0:3、数据块 0:# 一起根据行校验组 0 计算得出了数据块 0:0，又和数据块 1:2、数据块 3:2、数据块 #:2 一起根据对角校验组 2 计算得出了数据块 2:2。

经过证明，对于给定控制参数 p 的 RDP 编码的任意数据节点的恢复，RDOR 最少只需要获取 $\frac{3}{4}(p-1)^2$ 个数据块即可。而且只要该 RDOR 恢复策略由 $\frac{1}{2}(p-1)$ 个行校验组和 $\frac{1}{2}(p-1)$ 个对角校验组构成，就可以使恢复所需数据块数达到这个最小值。^[21]

注意，除了数据节点 0 外，其他数据节点都会包含一个特殊的数据块 $i:#$ ($1 \leq i \leq p-1$)。而这种数据块 $i:#$ 由于不参与任何对角校验组的计算，只可以根据行校验组进行恢复。例如，在图 3.8 的例子中，如果数据节点 1 发生故障，那么数据节点 1 中的数据块 3:# 由于并没有参与任何对角校验组的计算，因此只能根据行校验组 3 计算得出。

因此，RDOR 模型在挑选恢复某个数据节点时的数据块时，首先在待恢复

的数据块中选择 $\frac{1}{2}(p-1)$ 个数据块 $i:j$ ($j \neq \#$)，这些数据块根据对角校验组进行恢复；然后在待恢复的数据块中选择 $\frac{1}{2}(p-1)$ 个数据块，这些数据块根据行校验组进行恢复。

根据第3.1.1节的分析，为了恢复 RDP 编码中一个数据节点的全部数据块，传统的 RDP 编码需要获取一共 $(p-1)^2$ 个数据块，然后通过计算得出各个数据块；而 RDOR 只需要获取 $\frac{3}{4}(p-1)^2$ 个数据块即可计算得出各个数据块。相比传统的 RDP 编码的恢复方案，采用 RDOR 的恢复方案在理论上可以降低 25% 的数据传输量。

3.3.2 基于 RDOR 的数据恢复流程

接下来将展示本文是如何利用 RDOR 来对基于 RDP 编码的分布式 Memcached 中的数据恢复进行优化的。

一旦一个数据节点发生故障并被检测到，系统就会选出一个校验节点来对这个故障进行处理，这个校验节点我们称为 *leader*，后面的恢复流程主要由这个 *leader* 带领其他节点完成。在基于 RDP 编码的分布式 Memcached 中，我们通常选择行校验节点作为恢复流程的 *leader*。另外，由于某一个数据节点发生故障，因此在这个节点的数据恢复完成后，我们需要对这些数据进行存储。这里，我们会顺次地选择某一个校验节点来存储恢复后的数据，这个校验节点我们称为替代节点。也就是说，在基于 RDP 编码的分布式 Memcached 中，如果一个数据节点发生故障，系统将选择行校验节点作为其替代节点。如果有第二个数据节点发生故障，系统将选择对角校验节点作为第二个故障数据节点的替代节点。综上所述，在基于 RDP 编码的分布式 Memcached 中，单节点故障的恢复流程的 *leader* 和数据节点的替代节点通常都由行校验节点担任。

RDOR 策略的版本同步阶段

基于 RDOR 的分布式 Memcached 的单故障数据恢复流程主要包含两个阶段：版本同步阶段和数据恢复阶段。回顾第3.2.2节，本文采用了捎带更新的策略来高效地进行数据更新。而由于采用了捎带更新，最后各个节点的数据版本可能不一致，因此在实际的数据恢复之前要先同步各个节点之间的数据版本。版本同步阶段又分为两个小阶段：故障数据节点的数据版本同步阶段和其他数据节点的数据版本同步阶段。

故障数据节点的数据版本同步较为简单，主要用来对故障数据节点存放在各个

校验节点的数据进行同步。该阶段发生在为故障数据节点找到替代节点之后，具体的流程如下：

1. 在确定了替代节点之后，由替代节点向其他校验节点宣告自己将作为故障节点的替代节点。
2. 其他校验节点收到该消息后，将自身存储的故障数据节点的最新操作的 *id* 发送给替代节点。
3. 替代节点收到该 *id* 后，与自身存储的故障数据节点的最新操作的 *id* 做比较，选出较小的一个 *id* 作为故障数据节点的最终版本号。然后将故障数据节点所有不大于该版本号的更新操作从缓冲区取出并写入内存中。随后向所有校验节点广播该版本号，让他们也更新到同样的版本。
4. 其他校验节点收到替代节点发来的版本号后，将故障数据节点所有不大于该版本号的更新操作从缓冲区取出并写入内存。

以上便是故障数据节点的版本同步流程。其他数据节点的版本同步流程稍微复杂一点。主要差别在于故障数据节点的版本同步只发生在校验节点之间；而其他数据节点的版本同步需要所有节点合作完成。另外，故障数据节点的版本同步只在故障数据节点发生故障之后触发一次。而其他数据节点的版本同步则会嵌入到数据传输请求与其一起发送，因此每一轮数据修复都会触发一次其他数据节点的版本同步。这是因为故障数据节点发生故障后，该故障数据节点就不会再次发生数据更新，因此一次版本同步就可以了。但是在节点恢复的过程中，其他数据节点还是有可能发生数据更新，也就是所谓的降级模式。在降级模式中，如果某个数据节点发生了数据更新，就有可能导致该数据节点和校验节点间的数据版本不一致，因此在计算故障数据节点的故障之前，一定要先同步其他数据节点的数据版本。否则，计算出来的结果就有可能比真实值多出一个刚更新数据的差值，导致结果出现问题。

如图3.9所示是基于 RDP(6,4) 的分布式 Memcached 数据恢复中的版本同步流程。在该例中，数据节点 0 发生故障需要修复，行校验节点作为恢复 *leader* 发起版本同步。*leader* 向数据节点 1、数据节点 2 和数据节点 3 分别发送同步版本号的请求，数据节点收到请求后，将自己的最新版本号发送给各个校验节点。校验节点收到后，分别根据来自各个数据节点版本号更新自身数据。最后对角校验节点发送完成回应返回给行校验节点。

更一般地，本文将其他数据节点的版本同步流程总结如下：

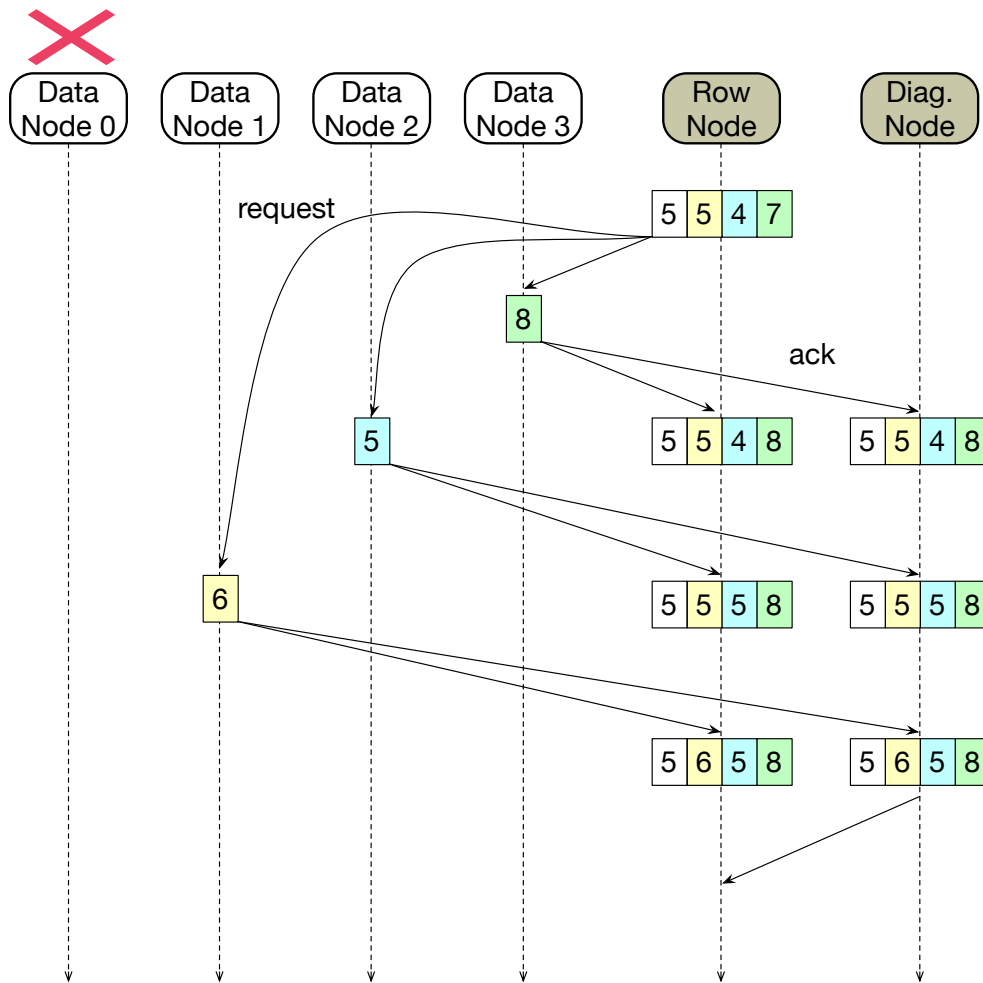


图 3.9 分布式 Memcached 数据恢复中的版本同步

1. 首先，恢复 *leader* 向参与恢复故障数据节点的其他所有数据节点发送同步版本号的请求。
2. 数据节点收到来自 *leader* 的同步版本号的请求后，发送自身最新版本号给所有参与恢复故障数据节点的校验节点。
3. 校验节点收到来自数据节点的版本号后，将对应节点所有不大于该版本号的更新操作从缓冲区取出并写入到内存中。如果该校验节点不是恢复流程的 *leader*，则在更新完所有参与恢复故障数据节点的其他数据节点的数据后，向恢复 *leader* 发送完成回应。

RDOR 策略的数据恢复阶段

从第3.1.2节，我们知道，本文所设计的分布式 Memcached 是以 *unit* 为最基本单元进行编码和存储的。同样地，在故障恢复的过程中，也是以 *unit* 为最基本单元一个个进行解码的。在实际的实现中，每个 *unit* 设置为 4KB。分布式 Memcached 中的每个节点的每个 *unit* 都有一个从 0 开始递增的标号 *unitID*。不同的节点独立计数。这个 *unitID* 与前面第3.2.2节所述的操作的 *id* 是不同的。一个节点上会有很多个 *unit*，每一个 *unit* 都会有一个不同的 *unitID*。而对于操作 *id* 而言，每一次写入或者更新数据都有一个不同的操作 *id*。而不同的操作 *id* 可能是在同一个 *unit* 上进行操作的。在恢复某一个数据节点上的某一个 *unit* 时，我们需要从同一个编码组中的其他参与恢复的节点上获取与该 *unit* 具有相同 *unitID* 的 *unit*，然后进行计算得出该丢失的 *unit*。

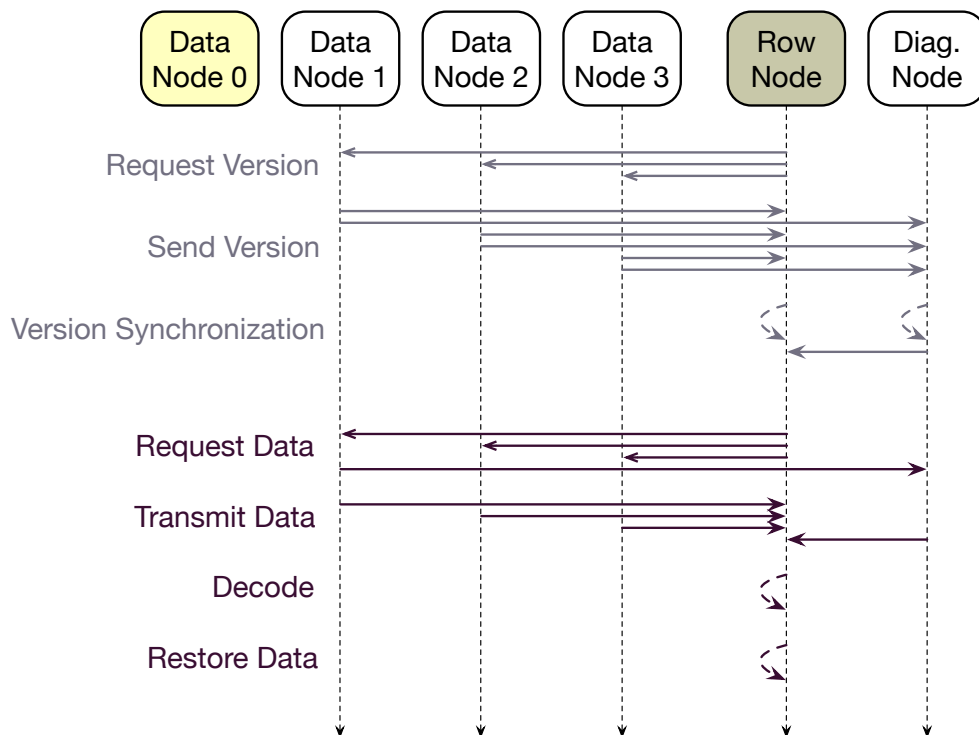


图 3.10 基于 RDOR 的分布式 Memcached 数据恢复流程

要恢复一个数据节点上的数据，根据第3.3.1节所述的 RDOR 恢复模型，需要所有剩余节点一起参与。如图3.10所示是基于 $RDP(6,4)$ 的分布式 Memcached 下的 RDOR 恢复模型示意图。假设数据节点 0 发生故障，行校验节点作为恢复

的 *leader* 和数据节点 0 的替代节点。

首先在版本同步阶段，行校验节点发送同步命令给所有剩余的数据节点。由于行校验节点和对角校验节点都需要参与后续的恢复计算工作，因此数据节点收到同步命令后发送最新版本号给两个校验节点。各个校验节点收到最新版本号后，将对应节点不大于所收到的版本号的操作从缓冲区中取出并写入内存中。然后，对角校验节点发送同步完成回应给行校验节点。

在数据恢复阶段，行校验节点发送数据请求命令给所有参与恢复计算的节点。注意这里对角校验节点也参与数据恢复的计算，但是行校验节点并不直接向对角校验节点发送数据请求命令，而是由数据节点 1 在向对角校验节点发送最新版本号时，捎带该数据请求命令。这样做可以减少一次网络通信。各个节点在收到数据请求命令后，向行校验节点发送计算数据节点 0 所需的数据。注意这里各个节点并不需要发送某个 *unit* 的全部数据，而只需要发送像第 3.3.1 节所述的部分数据块即可。行校验节点收到恢复某个 *unit* 所需的全部数据块后，即可开始对该 *unit* 进行解码计算。一个 *unit* 修复完成后，即可保存到自身的内存中。数据节点 0 的数据恢复完成后，行校验节点就可以作为数据节点 0 的替代节点进行工作了。即原来发向数据节点 0 的所有操作，如写入和读取，就可以直接发向行校验节点了，由行校验节点替代数据节点 0 进行响应。

根据上述分析，基于 RDOR 的分布式 Memcached 故障恢复流程可以总结为以下 6 步：

1. 选择一个校验节点作为恢复的 *leader*，选择一个校验节点作为故障数据节点的替代节点，由 *leader* 向所有参与故障数据节点恢复工作的数据节点发送同步版本号请求。
2. 数据节点收到同步版本号请求后，向所有校验节点发送本节点的最新版本号。
3. 校验节点收到最新版本号后，将对应节点的不大于该版本号的操作从缓冲区取出并写入内存中。随后，将同步完成回应发送给 *leader*。
4. *leader* 发送数据请求指令给所有参与故障数据节点恢复工作的节点。
5. 当一个节点收到数据请求指令后，将所需数据从自身内存中取出并发送给 *leader*。
6. 当 *leader* 收到恢复一个 *unit* 所需的全部数据后，通过计算得出丢失的数据，并将恢复后的结果发送到故障数据节点的替代节点。

图3.10展示的恢复流程包括其他节点的版本同步阶段和数据恢复阶段。其他节点的版本同步阶段包括前3步，数据恢复阶段包括后3步。在具体的实现中，这两个阶段是同步进行的，因此会有所谓的捎带优化。

注意，在通常情况下的单故障恢复过程中，恢复 *leader* 和故障节点的替代节点都会是行校验节点。因此，上面最后一步中的恢复 *leader* 将恢复后的结果发送到替代节点这一步通常不需要进行。行校验节点直接将恢复后的数据保存到自身内存中即可。这样，对于传统的 RDP 编码而言，恢复过程中总的数据传输量为 $(p-1) \times (p-1)$ 。而对于 RDOR 而言，要使数据传输量最小，需要让恢复 *leader*，也就是行校验节点，上所需的数据块数目尽可能地多。而在 RDOR 修复流程中，一个节点最多也只能是 $p-2$ 个数据块。因此，恢复过程中总的数据传输量最小为 $\frac{3}{4}(p-1)^2 - p + 2$ 。这相比 RDP 编码而言，至少可以减少 25% 的数据传输量。

第四节 基于 CRR 的数据恢复优化

3.4.1 CRR 恢复优化原理

由第3.1.1节可知，要恢复一个数据节点的数据，需要把剩余各个数据节点上的数据发送到恢复 *leader* 上。一般地说，不同节点上的数据是串行地一个一个地发送到恢复 *leader* 上的。当 *leader* 从所有所需节点上收到数据，就可以根据 RDP 编码的解码方法计算出丢失的数据。由于每个节点的带宽都是有限的，而 *leader* 要从各个数据节点收取大量的数据。因此，*leader* 的网络传输很容易成为系统恢复性能的瓶颈。为了解决这个问题，南开百度联合实验室的李鹏师兄提出了 Collective Reconstruction Read (CRR)^[22]。CRR 可以将 RDP 编码的恢复流程以并行和分布式的方式进行。

本文用一个例子来说明 CRR 的基本恢复流程。如图3.11所示是基于 CRR 的数据恢复示意图。在该例中，节点 0 发生故障，一共有 8 个节点参与恢复流程，编号从 1 到 8，其中节点 8 作为数据恢复的 *leader*，其他 7 个节点要将数据传输到节点 8 上。CRR 解码流程包括多个轮次。每一轮中的多个操作是可以并行进行的。在第一轮中，每一个满足 $i \% 2 = 1$ 的节点 i 将自身的数据发送到其后一个满足 $i \% 2 = 0$ 的节点。然后，满足 $i \% 2 = 0$ 的节点将其自身的数据和收到的数据进行计算并保存中间结果。在本例中，节点 1、节点 3、节点 5、节点 7 将自身数据分别发送到节点 2、节点 4、节点 6 和节点 8 上。在第二轮中，每一个满足

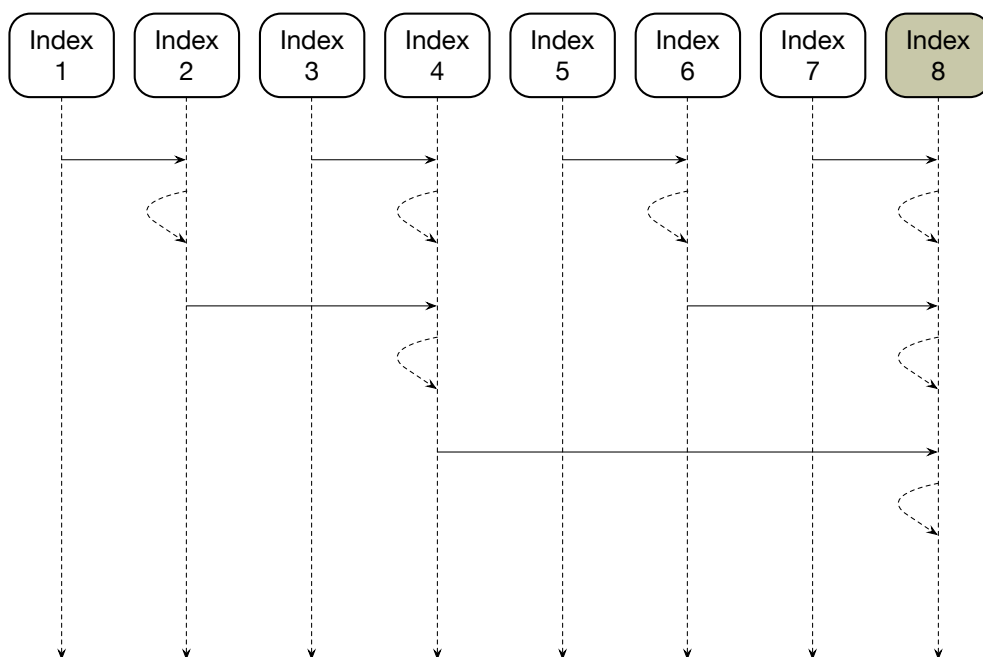


图 3.11 基于 CRR 的数据恢复示意图

$i\%4 = 2$ 的节点 i 将上一轮保存的中间结果发送到其后一个满足 $i\%4 = 0$ 的节点。然后，满足 $i\%4 = 0$ 的节点将其自身保存的中间结果和收到的中间结果进行计算并保存中间结果。在本例中，节点 2 和节点 6 将上一轮保存的中间结果分别发送到节点 4 和节点 8 上。在第三轮中，每一个满足 $i\%8 = 4$ 的节点 i 将上一轮保存的中间结果发送到其后一个满足 $i\%8 = 0$ 的节点。然后，满足 $i\%8 = 0$ 的节点将其自身保存的中间结果和收到的中间结果进行计算并保存中间结果。在本例中，节点 4 将上一轮保存的中间结果发送到节点 8 上。依此循环，直到 *leader* 收到所有节点的数据，或者说没有节点参与下一轮的传输。

由第 3.1.1 节可知，RDP 编码单数据节点故障的修复非常简单，只需要将剩余的数据节点和行校验节点对应的数据进行异或即可。而异或运算满足结合律。例如，对于类似 $d_0 \oplus d_1 \oplus d_2 \oplus d_3$ 这样的运算而言，可以先分别计算 $d_0 \oplus d_1$ 和 $d_2 \oplus d_3$ 作为中间结果，再把两个中间结果进行异或。因此，CRR 可以用来对 RDP 的解码过程进行优化。

从上面所述的 CRR 数据恢复流程来看，对于一个有 k 个节点参与的恢复流程而言，第一轮有 $\lfloor \frac{k}{2} \rfloor$ 个节点参与数据传输。之后的每一轮，参与数据传输的节点数目减半。因此，整个恢复流程一共有 $\lceil \log_2 k \rceil$ 轮。假设每一轮传输 m 个数

据块，传输每一个数据块需要的时间是 t 。则对于 CRR 数据恢复模型而言，总共需要的时间为 $m \lceil \log_2 k \rceil t$ 。而对于传统的 RDP 编码的恢复方案而言，串行地传输全部的数据需要 $k-1$ 轮，因此总共需要的时间为 $m(k-1)t$ 。由此可见，CRR 优化模型将传统的 RDP 编码的单故障恢复时间由多项式阶变成了对数阶，极大地提升了恢复的效率。

3.4.2 基于 CRR 的数据恢复流程

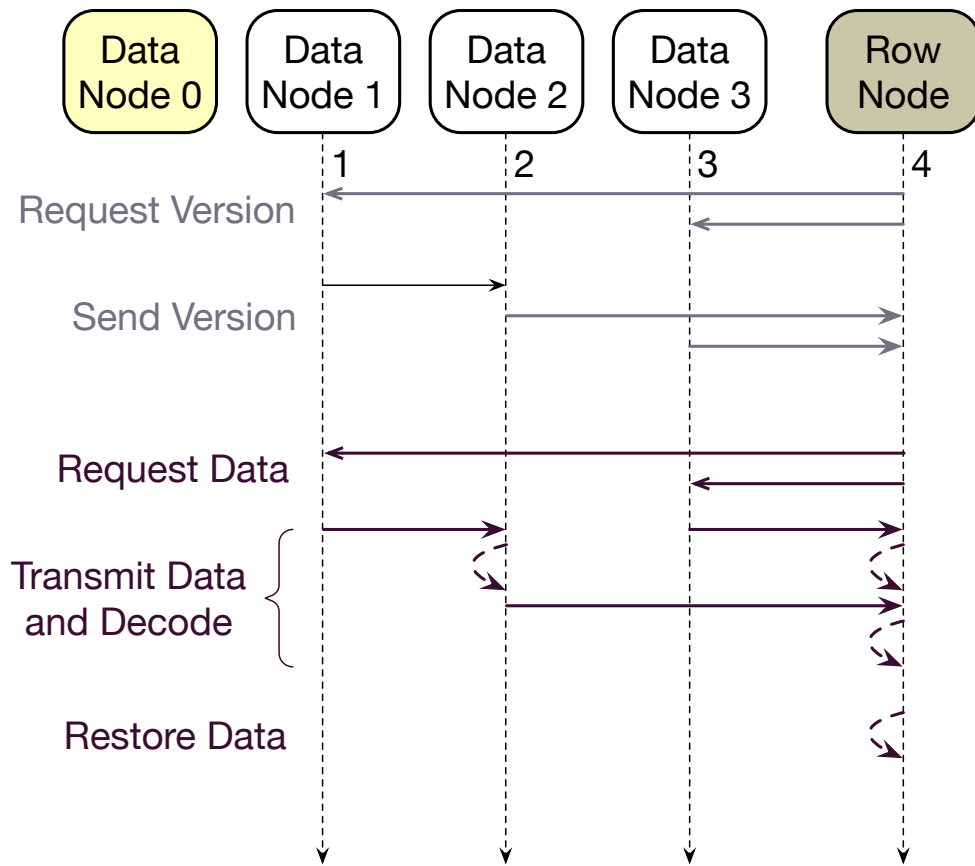


图 3.12 基于 CRR 的分布式 Memcached 数据恢复流程

基于 CRR 解码优化模型，本文提出了一种分布式 Memcached 中的并行优化方案。如图 3.12，考虑一个基于 $RDP(6,4)$ 编码的分布式 Memcached 编码组，假设数据节点 0 发生故障需要修复，行校验节点作为恢复的 *leader* 和数据节点 0 的替代节点。要修复数据节点 0 的数据，需要数据节点 1、数据节点 2、数据节点 3 和行校验节点参与解码，我们将上面参与解码运算的 4 个节点按照从 1 到 4

进行标号。与基于 RDOR 的数据恢复流程类似，CRR 恢复流程也分为版本同步阶段和数据恢复阶段。

首先在剩余节点版本同步阶段，行校验节点本应该发送版本同步命令给所有剩余数据节点。但是由于数据节点 1 并不需要直接向行校验节点发送数据块，为了减少网络传输，我们将数据节点 1 和数据节点 2 的回应合二为一。即行校验节点只向数据节点 1 发送版本同步命令，数据节点 1 收到该命令后，将最新版本号发送给数据节点 2，然后数据节点 2 将两个数据节点的最新版本号统一发送给行校验节点。由于在 CRR 恢复模型中，对角校验节点并不参与恢复流程，所以此时没有必要对对角校验节点上的版本进行同步，因此，该图所示的流程中没有对角校验节点。行校验节点收到某个数据节点的最新版本号后，将对应节点的不大于该版本号的操作从缓冲区取出并写入内存中。在同步完所有数据节点版本号后，行校验节点上的数据版本就和数据节点上是一致的了。

然后在数据恢复阶段，行校验节点发送数据请求命令给所有参与恢复计算的节点。而根据 CRR 的恢复流程，数据节点 1 需要向数据节点 2 发送数据，因此可以在数据节点 1 向数据节点 2 发送数据时，捎带着行校验节点向数据节点 2 的数据请求指令。因此，行校验节点只需要向数据节点 1 和数据节点 3 发送数据请求命令。接下来就可以按照第 3.4.1 节所述进行数据传输和计算了。在本例的第一轮中，数据节点 1 和数据节点 3 分别将自身的数据发送给数据节点 2 和行校验节点。数据节点 2 和行校验节点收到数据节点 1 和数据节点 3 的数据后，将自身数据和收到的数据进行异或，得到中间结果并临时存储。在第二轮中，数据节点 2 将上一轮得到的中间结果发送给行校验节点。行校验节点收到数据节点 2 的中间结果后，将自身上一轮保存的中间结果和收到的数据节点 2 的中间结果进行异或得到最终结果，即数据节点 0 的数据。最后，行校验节点将计算得到的数据节点 0 的值保存到自身的内存中。

根据上述分析，基于 CRR 的分布式 Memcached 故障恢复流程可以总结为以下 6 步：

1. 选择一个校验节点作为恢复的 *leader*，选择一个校验节点作为故障数据节点的替代节点，由 *leader* 向所有参与故障数据节点恢复工作的数据节点发送同步版本号请求。
2. 数据节点收到同步版本号请求后，向行校验节点发送本节点的最新版本号。

3. 行校验节点收到最新版本号后，将对应节点的不大于该版本号的操作从缓冲区取出并写入内存中。
4. *leader* 发送数据请求指令给 CRR 恢复流程中参与第一轮数据传输的节点。
5. 按照 CRR 并行恢复流程，将所有数据节点上的数据传输给 *leader*。
6. 当 *leader* 收到 CRR 最后一轮数据后，通过计算得出丢失的数据，并将恢复后的结果发送到故障数据节点的替代节点。

对基于 $RDP(6,4)$ 编码的分布式 Memcached 的恢复流程而言，传统的 RDP 编码的解码算法需要 3 轮的数据传输，而 CRR 优化模型仅需要 2 轮的数据传输，而且每一轮传输的数据量都是 4 个数据块。因此，对于 $RDP(6,4)$ 编码而言，理论上 CRR 优化模型可以降低传统的 RDP 编码 33.3% 的故障恢复时间。

第五节 RDOR 与 CRR 结合的可行性探讨

从第三节和第四节我们知道，在基于 RDP 编码的分布式 Memcached 中，RDOR 和 CRR 都可以减少发生故障时的恢复时间。但是，他们是从不同的角度出发来优化恢复过程，进而提升系统性能的。RDOR 恢复优化模型通过降低网络传输量来减少故障恢复时间，但是这个网络传输的过程仍然是串行地传输。与此对应，CRR 恢复优化模型通过提升传输并发度来减少故障恢复时间，但是 CRR 传输的总的数据量相比传统的 RDP 编码故障恢复过程并没有减少。那么，一个很自然的问题就是，是否可以将 RDOR 和 CRR 的优点合而为一，即既可以减少总的数据传输量，又提升数据传输的并发度。本节将主要探讨 RDOR 和 CRR 结合（本文表示为 RDOR-CRR）的可行性。

如图3.13所示是基于 RDOR-CRR 的分布式 Memcached 数据恢复流程。该图的上半部分是一个 $RDP(8,6)$ 编码的示意图， $RDP(8,6)$ 的控制参数 p 为 7，一个编码组具有 6 个数据节点和 2 个校验节点。在该例中，数据节点 0 发生故障需要修复，行校验节点作为恢复 *leader* 和数据节点 0 的替代节点。而采用 RDOR 恢复优化模型进行优化所需的数据块已经标示出来。该图的下半部分表示采用 CRR 恢复优化模型的网络传输流程。实线表示网络传输过程，实线上方的第一个数字是此次数据传输过程的序号，第二个数字表示此次传输的数据块数量，虚线表示计算过程。

采用 RDOR-CRR 的分布式 Memcached 数据恢复流程如下：在第一轮中，数

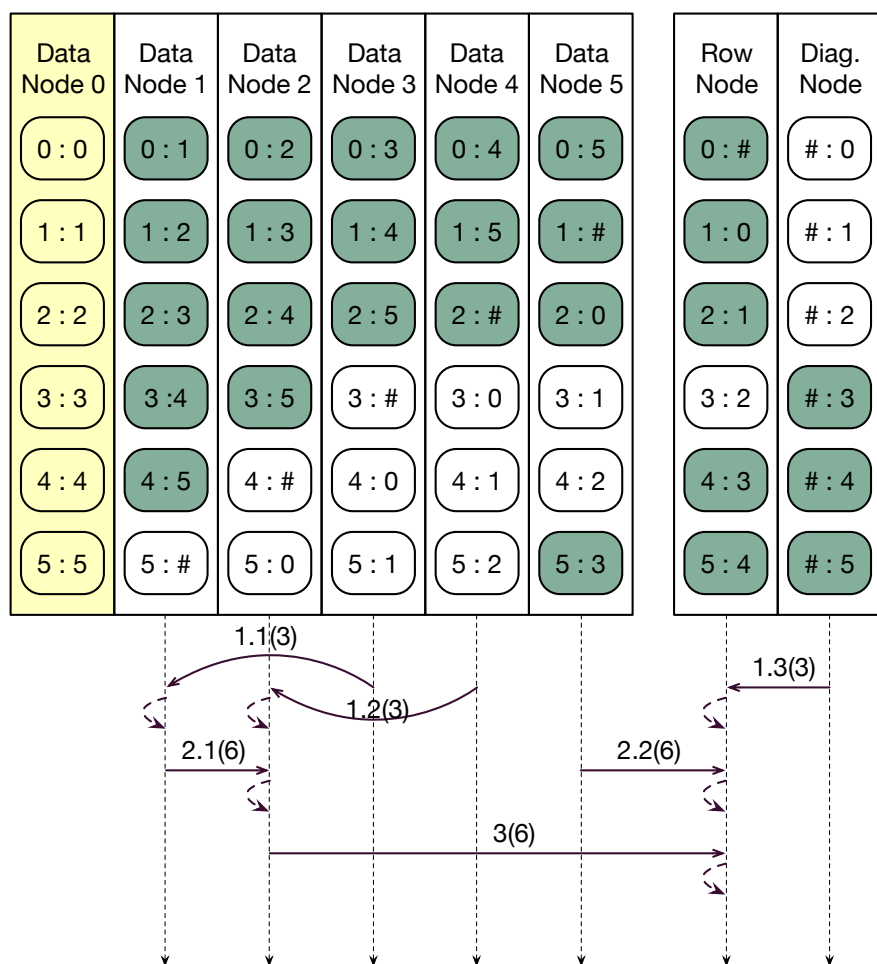


图 3.13 基于 RDOR-CRR 的分布式 Memcached 数据恢复流程

据节点 3 发送它的 3 个数据块给数据节点 1，数据节点 4 发送它的 3 个数据块给数据节点 2，对角校验节点发送它的 3 个数据块给行校验节点。注意，该轮以及后面的每一轮中的多个数据传输都是可以同时进行的，不存在先后顺序。对于每一个接收到数据的节点，如果它自身的数据块和收到的数据块在同一行校验组或对角校验组，它就需要将处于同一校验组的自身数据块和收到的数据块做异或运算，并将结果保存为中间结果。另外，如果某个数据块没有与其匹配的数据块在同一校验组，也需要将其保存到中间结果中，该数据块在后续的计算中将会用到。在下一轮中，把该中间结果传输给其他节点。在该例中，数据节点 1 在第一轮收到来自数据节点 3 的 3 个数据块后，将会计算出 6 个数据块作为中间结果，分别是数据块 $(0:1) \oplus (0:3)$ 、数据块 $(1:2) \oplus (1:4)$ 、数据块 $(2:3) \oplus (2:5)$ 、

数据块 $(2:3) \oplus (0:3)$ 、数据块 $(3:4) \oplus (1:4)$ 和数据块 $(4:5) \oplus (2:5)$ 。同理，数据节点 2 在第一轮收到来自数据节点 4 的 3 个数据块后，也会计算出 6 个数据块作为中间结果，分别是数据块 $(0:2) \oplus (0:4)$ 、数据块 $(1:3) \oplus (1:5)$ 、数据块 $(2:4) \oplus (2:\#)$ 、数据块 $1:3$ 、数据块 $(2:4) \oplus (0:4)$ 、数据块 $(3:5) \oplus (1:5)$ 。这里数据节点 2 的数据块 $1:3$ 属于对角校验组 3，而在数据节点 4 中没有数据块参与对角校验组 3 的计算，因此将数据块 $1:3$ 单独保存到中间结果中，以供后续计算之用。实际上，行校验节点在第一轮收到来自对角校验节点的 3 个数据块后，也会计算出 6 个数据块作为中间结果，但是行校验节点不需要向其他节点发送中间结果。在第二轮中，数据节点 1 发送它的 6 个中间结果块给数据节点 2，数据节点 5 发送它的 4 个数据块给行校验节点。数据节点 2 在第二轮收到来自数据节点 1 的 6 个中间结果块后，会将这 6 个中间结果块和自己在第一轮计算出的 6 个中间结果块进行计算得出新的 6 个中间结果块，分别是数据块 $(0:1) \oplus (0:3) \oplus (0:2) \oplus (0:4)$ 、数据块 $(1:2) \oplus (1:4) \oplus (1:3) \oplus (1:5)$ 、数据块 $(2:3) \oplus (2:5) \oplus (2:4) \oplus (2:\#)$ 、数据块 $(2:3) \oplus (0:3) \oplus (1:3)$ 、数据块 $(3:4) \oplus (1:4) \oplus (2:4) \oplus (0:4)$ 和数据块 $(4:5) \oplus (2:5) \oplus (3:5) \oplus (1:5)$ 。同样地，行校验节点在第二轮收到来自数据节点 5 的 4 个数据块后，也会将这 4 个数据块和自己在第一轮计算出的 6 个中间结果块计算得出新的 6 个中间结果块。在这一轮中，虽然数据节点 5 只传输了 4 个数据块，但是数据节点 1 需要传输 6 个数据块。对于并行的数据传输而言，瓶颈会在数据节点 1 这边，因此本文视这一轮的传输量为 6 个数据块。在最后一轮中，数据节点 2 发送它的 6 个中间结果块给行校验节点。行校验节点收到来自数据节点 2 的 6 个中间结果块后，将这 6 个中间结果块和自己在第二轮计算出的 6 个中间结果块计算得出新的 6 个数据块，即数据节点 0 丢失的 6 个数据块。

从上面的例子中我们发现，在基于 RDOR-CRR 的分布式 Memcached 数据恢复流程中，每个节点计算出的中间结果都是 6 个数据块。本文将该发现一般化为定理 3.1。

定理 3.1 在一个拥有 k 个数据节点的 RDP 编码阵列中，如果采用 RDOR-CRR 恢复模型，各个节点生成的中间结果都会包含 k 个中间结果数据块。

证明. 考虑一个具有 k 个数据节点的 RDP 编码阵列。该阵列还有 2 个校验节点，所有节点记为节点 0 到 $k+1$ 。该阵列具有 k 条对角线，记为对角线 0 到 $k-1$ 。节点 i 上的 k 个数据块参与除了对角线 $i-1$ 外的所有对角校验组的计

算。例如在图3.13中，节点 1 只没有覆盖对角线 0，节点 2 只没有覆盖对角线 1，节点 6（行校验节点）只没有覆盖对角线 5。因此，任何两个节点结合在一起，它们的数据块就会覆盖所有的对角线。根据 RDOR 恢复优化模型的规则，要恢复一个数据节点，我们需要 $\frac{k}{2}$ 个行校验组和 $\frac{k}{2}$ 个对角校验组。这就意味着，在 RDOR-CRR 恢复模型中，当一个节点收到另一个节点发来的数据后，会恰好计算出 $\frac{k}{2}$ 个对角中间结果数据块。类似地，可以看出在 RDOR-CRR 恢复模型中，当一个节点收到另一个节点发来的数据后，会恰好计算出 $\frac{k}{2}$ 个行中间结果数据块。因此，在 RDOR-CRR 恢复模型中，各个节点生成的中间结果都会包含一共 k 个中间结果数据块。□

定理3.1说明了在 RDOR-CRR 恢复模型中，除了第一轮外，之后的每一轮都需要传输 k 个数据块。根据 CRR 恢复优化模型的规则，在第一轮中，会有 $\lfloor \frac{k+1}{2} \rfloor$ 个节点需要向其他节点传输数据。根据 RDOR 恢复优化模型的规则，在任何 RDP 阵列中，都有 3 个节点具有 $\frac{k}{2}$ 个数据块参与 RDOR 计算，2 个节点具有 $\frac{k}{2} + 1$ 个数据块参与 RDOR 计算，2 个节点具有 $\frac{k}{2} + 2$ 个数据块参与 RDOR 计算，等等。易证，在 RDOR-CRR 恢复模型的第一轮中，最小数据传输块数为 $\lfloor (\lfloor \frac{k+1}{2} \rfloor - 2) / 2 \rfloor + \frac{k}{2}$ 块。根据第四节，我们知道在 CRR 恢复模型中，第一轮每个节点传输 k 个数据块。因此，相比 CRR 恢复模型而言，RDOR-CRR 恢复模型在第一轮可以节省 $\frac{k}{2} - \lfloor (\lfloor \frac{k+1}{2} \rfloor - 2) / 2 \rfloor$ 个数据块的传输时间。而对于其他轮次而言，RDOR-CRR 恢复模型和 CRR 模型都需要传输 k 个数据块。因此相比 CRR 恢复模型，RDOR-CRR 恢复模型只在第一轮次有优化效果。

另外，RDOR-CRR 恢复模型比 CRR 恢复模型多 1 个节点参与数据恢复，这就意味着 RDOR-CRR 恢复模型的数据传输轮次是 $\lceil \log_2(k+1) \rceil$ 轮。如果满足 $k = 2^m$ ，那么 RDOR-CRR 恢复模型会比 CRR 恢复模型多一轮数据传输。例如，对于 $RDP(6,4)$ 而言，采用 RDOR-CRR 恢复模型需要 3 轮数据传输，而采用 CRR 恢复模型只需要 2 轮数据传输。此时，RDOR-CRR 恢复模型不仅没有优化效果，反而会增加恢复所需时间，降低系统性能。

基于上述分析，RDOR-CRR 恢复模型相比 CRR 恢复模型而言的优化是非常有限的。而且这种优化是建立在将分布式 Memcached 的网络传输协议复杂化，将计算复杂化的基础之上的。因此，本文并没有将 RDOR-CRR 应用到我们的高性能分布式 Memcached 中。

第六节 本章小结

本章介绍了本文的主要工作，即实现了一个高效容错的分布式 Memcached。其中，第一节介绍了本文如何将 RDP 编码应用到分布式 Memcached；第二节介绍了基于 RDP 编码的增量更新；第三节和第四节分别介绍了本文如何用 RDOR 恢复优化模型和 CRR 恢复优化模型对分布式 Memcached 的故障恢复流程进行优化；第五节探讨了将 RDOR 和 CRR 结合起来的可行性。

第四章 实验与分析

第一节 实验环境描述

本文的主要工作是设计并实现了一个高效容错的分布式 Memcached 系统，因此实验内容主要分为功能实验和性能实验两部分。本文的性能实验主要与 Cocyus 进行对比分析。本文对各项实验内容均进行两组实验。第一组实验用控制参数 p 为 5 时的 $RDP(6,4)$ 编码与 $RS(6,4)$ 编码进行对比，这两种编码均由 4 个数据节点和 2 个校验节点组成；第二组实验用控制参数 p 为 7 时的 $RDP(8,6)$ 编码与 $RS(8,6)$ 编码进行对比，这两种编码均由 6 个数据节点和 2 个校验节点组成。如表 4.1 所示是本文实验的环境配置。

表 4.1 实验环境配置

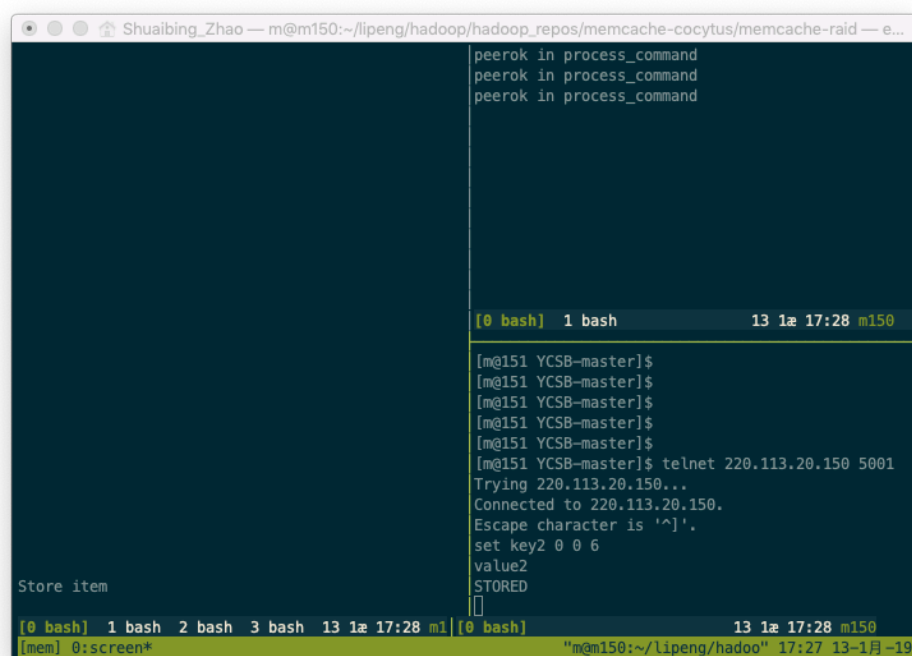
系统配置	
Operation System	CentOS 6.5
CPU Processor	Intel(R) Xeon(TM) CPU 2.80GHz
Memory	2GB
Memcached	
Memcached Version	1.4.3
Number of Nodes	6/8

本文使用 Yahoo! Cloud Serving Benchmark (YCSB)^[51] 为分布式 Memcached 生成随机的数据集。第 2.1.4 节所说的分布式 Memcached 客户端就集成在 YCSB 中。YCSB 测试工具自带了各种流行的分布式系统的客户端，包括 Memcached 客户端。但是本文所设计的分布式 Memcached 将原来的 Memcached 节点划分成了数据节点和校验节点两种，因此并不能直接用 YCSB 测试工具默认的客户端。本文在 YCSB 中实现了用于该分布式 Memcached 的客户端。在分布式 Memcached 系统正常运行的情况下，该客户端会根据哈希算法将用户的请求发送到某一台数据节点上，而当该数据节点发生故障时，该客户端会将该请求发送到其替代节点上。YCSB 生成的数据是一个个的键值对。其中键的长度通常小

于 16 字节。值是一个大小可以自定义的对象。在本文中，为了实验结果的可靠性，我们用 5 种不同大小的 item 来对分布式 Memcached 的性能进行测试，分别是 4KB、8KB、16KB、32KB 和 64KB。

另外，为了测试不同网络状态下的系统性能，本文使用 iproute2^[52] 来对分布式 Memcached 服务器之间的网络带宽进行限制。在本文的实验中，我们测试了三种网络环境，分别是高带宽（1Gbps）、中带宽（500Mbps）和低带宽（100Mbps）。

第二节 系统功能实验



```
Shuaibing_Zhao — m@m150:~/lipeng/hadoop/hadoop_repos/memcache-cocytus/memcache-raid — e...
peerok in process_command
peerok in process_command
peerok in process_command

[0 bash] 1 bash 13 17:28 m150

[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$ telnet 220.113.20.150 5001
Trying 220.113.20.150...
Connected to 220.113.20.150.
Escape character is '^'.
set key2 0 0 6
value2
STORED

Store item

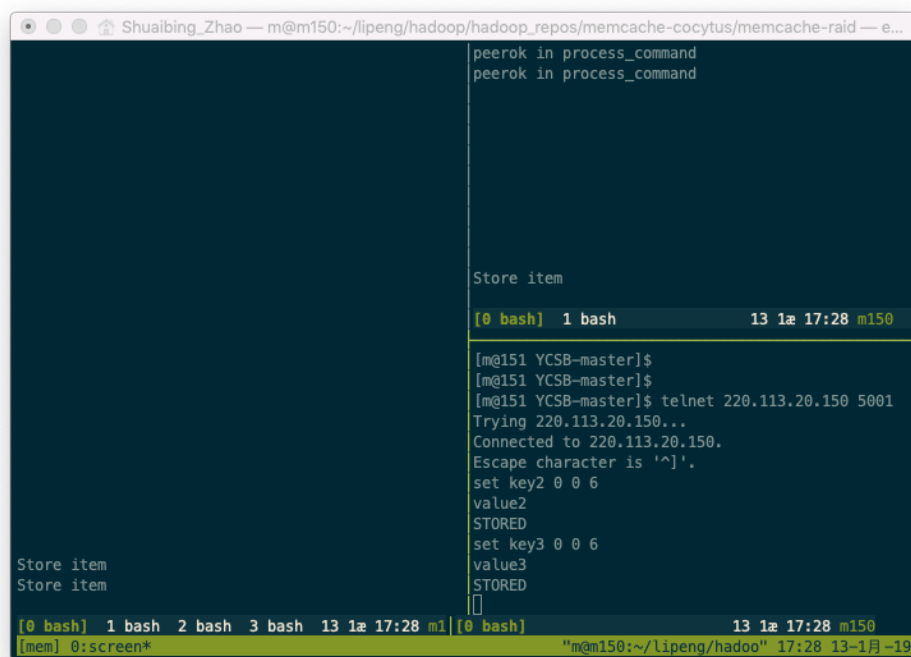
[0 bash] 1 bash 2 bash 3 bash 13 17:28 m1 [0 bash] 13 17:28 m150
[mem] 0:screen* "m@m150:~/lipeng/hadoo" 17:27 13-1月-19
```

图 4.1 向分布式 Memcached 的数据节点写入 1 个数据

作为一个缓存系统，分布式 Memcached 最重要的功能当然是存储和读取数据。本节以控制参数 p 为 5 时的 $RDP(6,4)$ 编码为例，来对分布式 Memcached 的读写功能进行验证。基于 $RDP(6,4)$ 编码的分布式 Memcached 一共有 6 个节点，其中包含 4 个数据节点和 2 个校验节点。另外还有 1 个客户端用于向分布式 Memcached 发送读写请求。本文的所有代码修改工作都是针对系统层次的，这

些修改对用户基本是透明的。并且本文并没有修改对外接口。因此为了简单起见，本节直接使用 Telnet 客户端向分布式 Memcached 发送读写请求。

如图4.1所示是由客户端向分布式 Memcached 的数据节点写入 1 个数据的示意图。该图被划分成 3 个面板，左边面板是 4 个数据节点，右上边面板是 2 个校验节点，右下边面板是 Telnet 客户端。从图中可以看出，客户端向数据节点 0 写入 1 个数据，其中键为“key2”，值为“value2”，其他为标志信息，本文暂不讨论。然后数据节点 0 收到该写入命令后，对数据进行存储。



```
Shuaibing_Zhao — m@m150:~/lipeng/hadoop/hadoop_repos/memcache-cocytus/memcache-raid — e...
peerok in process_command
peerok in process_command

Store item
[0 bash] 1 bash 13 17:28 m150

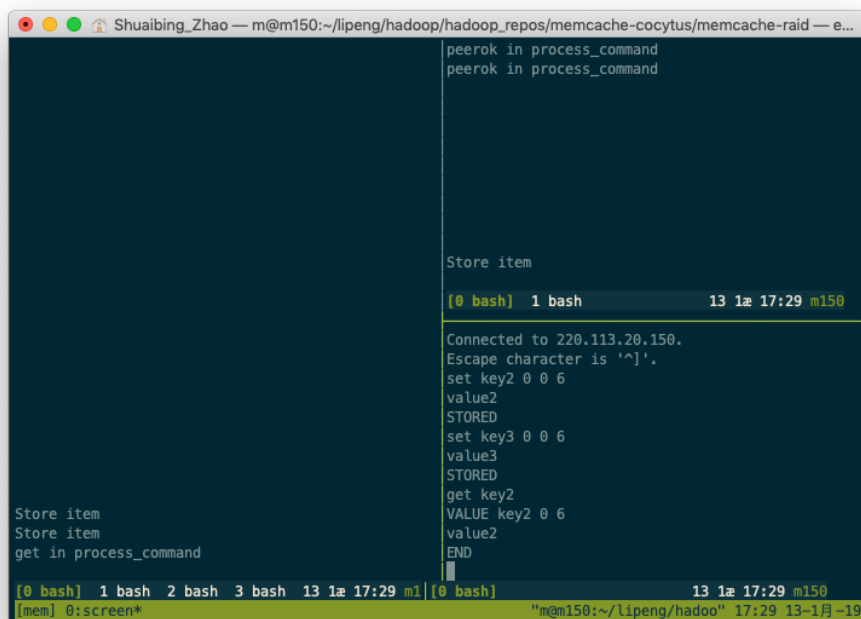
[m@151 YCSB-master]$
[m@151 YCSB-master]$
[m@151 YCSB-master]$ telnet 220.113.20.150 5001
Trying 220.113.20.150...
Connected to 220.113.20.150.
Escape character is '^]'.
set key2 0 0 6
value2
STORED
set key3 0 0 6
value3
STORED
[]

Store item
Store item
[0 bash] 1 bash 2 bash 3 bash 13 17:28 m1 [0 bash] 13 17:28 m150
[mem] 0:screen* "m@m150:~/lipeng/hadoo" 17:28 13-1月-19
```

图 4.2 再次向分布式 Memcached 的数据节点写入 1 个数据

如图4.2所示是由客户端再次向分布式 Memcached 的数据节点写入 1 个数据的示意图。这次，客户端向数据节点 0 又写入 1 个数据，其中键为“key3”，值为“value3”。数据节点 0 收到该写入命令后，对数据进行存储。值得注意的是，第一次向数据节点 0 写入数据时，校验节点并没有对数据的校验数据进行存储，而是如第3.2.2节所述，将其保存到校验节点的缓冲区。在第二次向数据节点 0 写入数据时，才会把上一次临时存放在缓冲区中的校验取出写入内存中。

如图4.3所示是由客户端向分布式 Memcached 的数据节点读取 1 个数据的示意图。客户端向数据节点 0 发送一条读指令，要求读取键为“key2”的值。数据



```
peerok in process_command
peerok in process_command

Store item
[0 bash] 1 bash 13 17:29 m150

Connected to 220.113.20.150.
Escape character is '^'.
set key2 0 0 6
value2
STORED
set key3 0 0 6
value3
STORED
get key2
VALUE key2 0 6
value2
END

Store item
Store item
get in process_command
[0 bash] 1 bash 2 bash 3 bash 13 17:29 m1 [0 bash] 13 17:29 m150
[mem] 0:screen* "m@m150:~/lipeng/hadoo" 17:29 13-1月-18
```

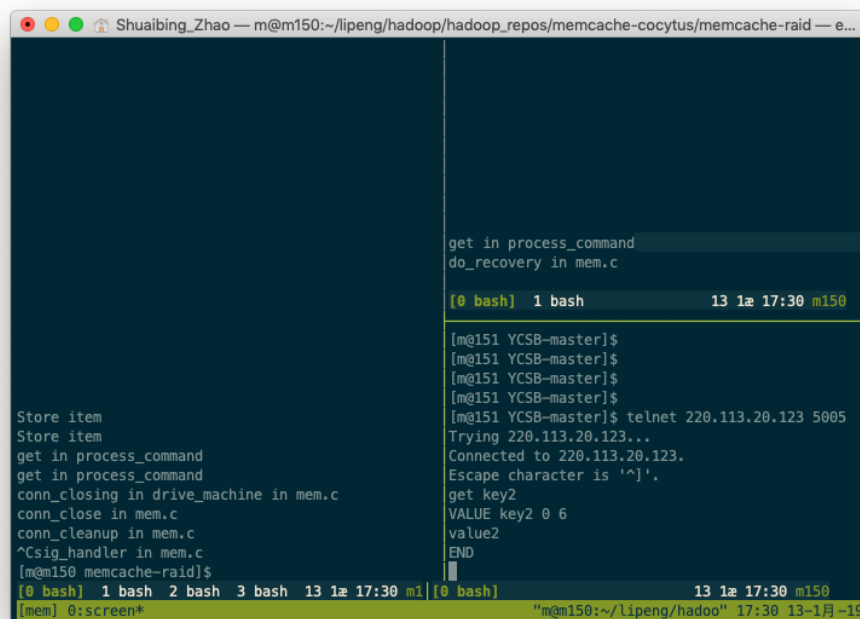
图 4.3 从分布式 Memcached 的数据节点读取 1 个数据

节点 0 收到该读取指令后，从自身内存中查找对应的值，并向客户端返回结果“value2”。读取数据的过程没有数据节点的参与。

上面是分布式 Memcached 正常运行模式下的功能实验，下面本文将展示在某数据节点发生故障时的功能实验。

如图4.4所示，数据节点 0 发生故障，行校验节点被选作数据节点 0 的替代节点，原来发向数据节点 0 的请求都应发向行校验节点。客户端向行校验节点发送一条读指令，要求读取键为“key2”的值。行校验节点收到该读取指令后，进行恢复工作得到对应的值，将计算得到的数据保存到自身内存中并向客户端返回结果“value2”。

如图4.5所示，客户端再次向行校验节点发送读取键为“key2”的值，由于行校验节点已经恢复过这个数据，因此内存中已经有了该值，不需要再次进行恢复工作，可以直接向客户端返回结果“value2”。



```
Shuaibing_Zhao — m@m150:~/lipeng/hadoop/hadoop_repos/memcache-cocytus/memcache-raid — e...

Store item
Store item
get in process_command
get in process_command
conn_closing in drive_machine in mem.c
conn_close in mem.c
conn_cleanup in mem.c
^Csig_handler in mem.c
[m@m150 memcache-raid]$

[0 bash] 1 bash 13 1a 17:30 m150

[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$
[m@m151 YCSB-master]$ telnet 220.113.20.123 5005
Trying 220.113.20.123...
Connected to 220.113.20.123.
Escape character is '^]'.
get key2
VALUE key2 0 6
value2
END
^C
[m@m150 memcache-raid]$

[0 bash] 1 bash 2 bash 3 bash 13 1a 17:30 m1 [0 bash] 13 1a 17:30 m150
[mem] 0:screen* "m@m150:~/lipeng/hadoo" 17:30 13-1月-15
```

图 4.4 从分布式 Memcached 的替代节点读取 1 个数据

第三节 系统性能实验

第二节展示了本文所设计并实现的分布式 Memcached 的功能，本节将对这些功能的实际性能进行测试。本文主要与 Cocytus 的性能做对比分析。在本节的实验结果图中，我们把 Cocytus 的结果记为“RS Codes”，把基于 RDP 编码的结果记为“RDP Codes”，把基于 RDOR 恢复优化模型的结果记为“RDOR”，把基于 CRR 恢复优化模型的结果记为“CRR”。

4.3.1 系统正常运行时的性能实验

首先，对于分布式 Memcached 这样一个缓存系统而言，写入数据自然是最重要的功能。而对于基于 RDP 编码的分布式 Memcached 而言，编码过程是写入过程的一个重要的组成部分。图4.6展示了向分布式 Memcached 写入不同大小的 item 时的编码计算开销。对于每一种大小，本文都向分布式 Memcached 中写入 2000 个 item。从图中可以看出，采用 RDP 编码的分布式 Memcached 的编码计算时间明显比采用 RS 编码的分布式 Memcached 的编码计算时间要短。具

```

Store item
Store item
get in process_command
get in process_command
conn_closing in drive_machine in mem.c
conn_close in mem.c
conn_cleanup in mem.c
^Csig_handler in mem.c
[m@m150 memcache-raid]$
[0 bash] 1 bash 2 bash 3 bash 13 1a 17:30 m1
[mem] 0:screen*

get in process_command
do_recovery in mem.c
get in process_command

[0 bash] 1 bash 13 1a 17:30 m150

[m@m151 YCSB-master]$ telnet 220.113.20.123 5005
Trying 220.113.20.123...
Connected to 220.113.20.123.
Escape character is '^]'.
get key2
VALUE key2 0 6
value2
END
get key2
VALUE key2 0 6
value2
END
[0 bash] 1 bash 13 1a 17:30 m150
[m@m150:~/lipeng/hadoop 17:30 13-1月-15]

```

图 4.5 再次从分布式 Memcached 的替代节点读取 1 个数据数据

体地说, $RDP(6,4)$ 编码的计算开销比 $RS(6,4)$ 编码要少 46% 到 54%; $RDP(8,6)$ 编码的计算开销比 $RS(8,6)$ 编码要少 39% 到 47%。一般情况下, RS 编码的计算是基于有限域的, 而 RDP 编码的计算是纯异或的。RDP 编码的计算速度应该比 RS 编码快很多, 甚至不在同一个数量级。但是, 一方面, 本文所实现的分布式 Memcached 和 Cocytus 均使用了增量编码的思想去写入数据; 另一方面, 二者又都采用了 Jerasure 库函数来对计算进行加速^[53]。所以最终的结果就是采用 RDP 编码的分布式 Memcached 的编码计算开销是采用 RS 编码的分布式 Memcached 的一半左右。

接下来, 图4.7展示的是向分布式 Memcached 写入数据时的吞吐率。本实验一共向分布式 Memcached 中写入了 20000 个 4KB 大小的 item。从图中可以看出, 无论控制参数 p 为 5 还是 7, 采用 RDP 编码的分布式 Memcached 的吞吐率都和采用 RS 编码的分布式 Memcached 的吞吐率接近。这是因为本文基于 RDP 编码的分布式 Memcached 的数据写入流程与基于 RS 编码的 Cocytus 类似, 都需要由数据节点向各个校验节点发送校验增量并等待校验节点返回其收到该差

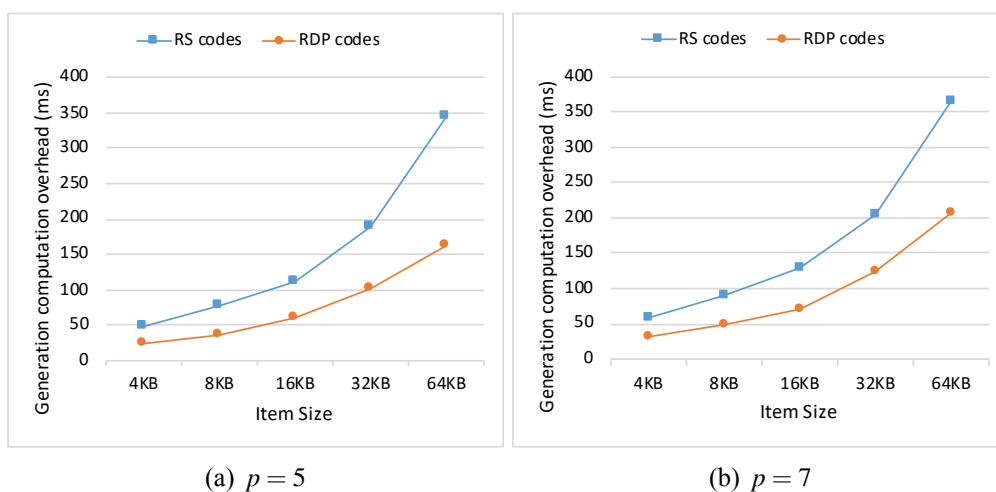


图 4.6 向分布式 Memcached 写入不同大小的 item 时的编码计算开销

量的回应。由图可知，采用 $RDP(6,4)$ 编码和 $RS(6,4)$ 编码的分布式 Memcached 的吞吐率约为每秒 1100 次写入；而采用 $RDP(8,6)$ 编码和 $RS(8,6)$ 编码的分布式 Memcached 的吞吐率约为每秒 985 次写入。值得一提的是，图 4.6 表明 RDP 编码的计算开销约为 RS 编码的一半，但是向分布式 Memcached 写入 item，不仅仅需要编码计算，还需要大量的数据传输。众所周知，网络传输的速度远远低于内存计算的速度。根据测试，在本文的分布式 Memcached 中，编码计算的时间开销仅为写入流程总开销的 2% 左右。因此，RDP 编码相比 RS 编码在计算方面的优势被漫长的数据传输过程给抹平了。所以最终的结果就是采用 RDP 编码的分布式 Memcached 的写入速度与采用 RS 编码的分布式 Memcached 接近。

图 4.8 展示的从向分布式 Memcached 读取数据时的吞吐率。本实验一共向分布式 Memcached 发送了 20000 次读请求，分布式 Memcached 中的数据的大小均为 4KB。从图中可以看出，无论控制参数 p 为 5 还是 7，采用 RDP 编码的分布式 Memcached 的吞吐率都和采用 RS 编码的分布式 Memcached 的吞吐率接近。这是因为读取数据只涉及到数据节点，与校验节点没有任何关系。数据节点收到一次读数据请求后，只需从自身内存中查找到该数据并返回给客户端即可。由图可知，采用 $RDP(6,4)$ 编码和 $RS(6,4)$ 编码的分布式 Memcached 的吞吐率约为每秒 3250 次读取；而采用 $RDP(8,6)$ 编码和 $RS(8,6)$ 编码的分布式 Memcached 的吞吐率约为每秒 3050 次读取。

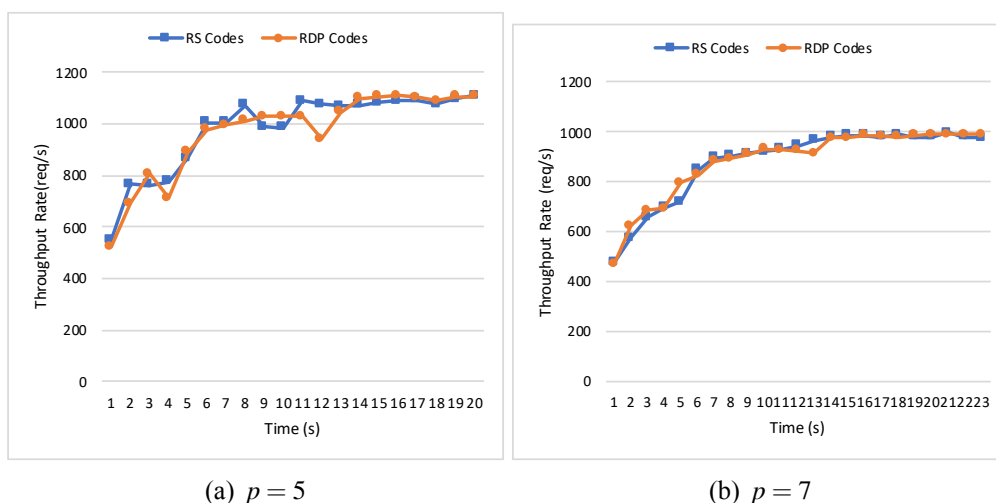


图 4.7 向分布式 Memcached 写入数据时的吞吐率

4.3.2 系统故障模式下的重构性能实验

第4.3.1节展示了本文所实现的分布式 Memcached 在正常运行时的性能实验。但是，在分布式系统中，节点发生故障是不可避免的。当节点发生故障时，为了保证系统仍能提供服务，需要把故障节点的数据恢复出来。而数据的恢复速度直接影响系统的性能。本文采用了 RDOR 和 CRR 两种方案来对故障节点的恢复过程进行加速。本节将对数据节点的故障恢复性能进行测试。

在分布式 Memcached 中，解码计算过程是数据恢复过程中的一个重要组成部分。图4.9展示的是分布式 Memcached 在恢复某一个数据节点过程中的解码计算开销。本实验一共分为 5 组。每组向分布式 Memcached 中插入 2000 个大小相同的 item。不同的组插入的 item 的大小不同，分别为 4KB、8KB、16KB、32KB 和 64KB。具体的实验结果是，对于控制参数 p 为 5 的各个方案而言，RS 编码的解码时间比 RDP 编码要多 45% 左右，RDOR 恢复模型的解码时间与 RS 编码相当，CRR 恢复模型的解码时间比 RDP 编码要少 22% 左右；而对于控制参数 p 为 7 的各个方案而言，RS 编码的解码时间比 RDP 编码要多 51% 左右，RDOR 恢复模型的解码时间与 RS 编码相当，CRR 恢复模型的解码时间比 RDP 编码要少 35% 左右。总体而言，我们可以得出三点结论。第一，我们可以看出 RS 编码的解码时间比 RDP 编码要长。同第4.3.1节所述的编码过程类似，RS 编码和 RDP 编码的解码都采用了 Jerasure 库函数来对解码的计算过程进行并行优化，因此，二者的解码时间相差没有太过明显。第二，RDOR 恢复模型的解码时间与

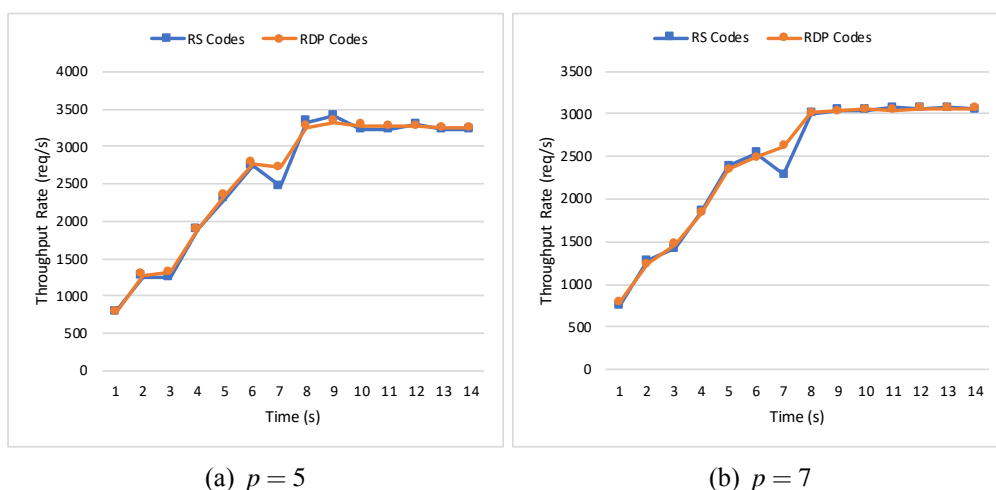


图 4.8 从分布式 Memcached 读取数据时的吞吐率

RS 编码相当。这是因为虽然 RDOR 恢复模型的解码过程需要的数据量比 RDP 编码减少了，但是 RDOR 恢复模型在解码时的计算压力却比 RDP 编码要大。如第3.3.1节所述，RDOR 恢复模型的计算不仅涉及行校验组的计算，还涉及对角校验组的计算。这种行与对角线交叉的计算对于并行计算而言是一种压力。最终的结果就是 RDOR 恢复模型的解码时间与 RS 编码相当。第三，CRR 恢复模型的解码时间比 RDP 编码要少。这是因为 CRR 恢复模型将原来 RDP 编码的解码过程划分成多个子过程，并将这些子过程分到不同的节点上并行计算，最终减少了原 RDP 编码的解码时间。

图4.10展示的是分布式 Memcached 在恢复某一个数据节点过程中的总开销。与图4.9相同，该实验分为 5 组，每组向分布式 Memcached 中插入 2000 个大小相同的 item。对比图4.9与图4.10可以发现，在分布式 Memcached 中的节点恢复过程中，解码计算时间占据总恢复时间的不到 10%。具体的实验结果是，对于控制参数 p 为 5 的各个方案而言，采用 RDP 编码的分布式 Memcached 的恢复总开销与采用 RS 编码相当，采用 RDOR 恢复模型的恢复总开销比采用 RS 编码少 10% 左右，采用 CRR 恢复模型的恢复总开销比采用 RS 编码少 30% 左右。对于控制参数 p 为 7 的各个方案而言，采用 RDP 编码的分布式 Memcached 的恢复总开销与采用 RS 编码相当，采用 RDOR 恢复模型的恢复总开销比采用 RS 编码少 14% 左右，采用 CRR 恢复模型的恢复总开销比采用 RS 编码少 37% 左右。总体而言，我们可以得出三点结论。第一，采用 RDP 编码的分布式 Memcached 的

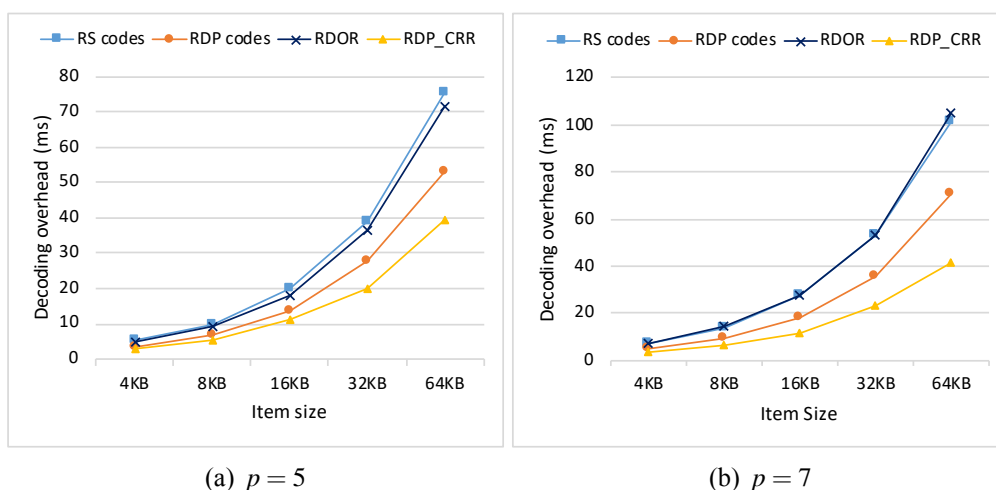


图 4.9 分布式 Memcached 在恢复数据节点过程中的解码计算开销

恢复总开销与采用 RS 编码相当。由第3.1.1节，我们直到 RDP 编码的解码恢复过程所需要的数据量与 RS 编码是一样的，而且二者的数据传输方式都是串行传输。而在故障节点的恢复过程中，数据传输的时间占据总恢复时间的很大一部分。因此，最终采用 RDP 编码的分布式 Memcached 的恢复总开销与采用 RS 编码相当。第二，采用 RDOR 恢复模型的恢复总开销比 RS 编码减少了一部分。第三，采用 CRR 恢复模型的恢复总开销比 RS 编码减少很多。理论上，在控制参数 p 为 5 时，采用 RDOR 恢复模型的恢复总开销应该比 RS 编码减少 25%，采用 CRR 恢复模型的恢复总开销应该比 RS 编码减少 33.3%。根据实验结果，CRR 恢复模型的恢复时间符合预期，但是 RDOR 恢复模型的恢复总开销优化比却比理论值少了很多。我们猜想，这是因为 RDOR 恢复模型在数据恢复过程中，虽然需要获取的数据总量减少了，但是却需要从更多的节点获取数据。

为了验证上面的猜想，我们把网络带宽限制到一个较低的值时，来看 RDOR 恢复模型是否会表现得更好。对于一个分布式系统而言，网络带宽不可能一直是很高的。其他进程的网络通信，恢复过程中的读写操作等都会占据一定的带宽。因此，测试不同网络状态下的系统恢复性能是有意义的。图4.11展示的是在不同的网络状态下，分布式 Memcached 在恢复某一个数据节点过程中的总开销。在该实验中，每次向分布式 Memcached 中插入 2000 个 4KB 大小的 item。具体的实验结果是，对于控制参数 p 为 5 的各个方案而言，在高带宽、中带宽、低带宽的网络状态下，采用 RDP 编码的分布式 Memcached 的恢复总开销均与采

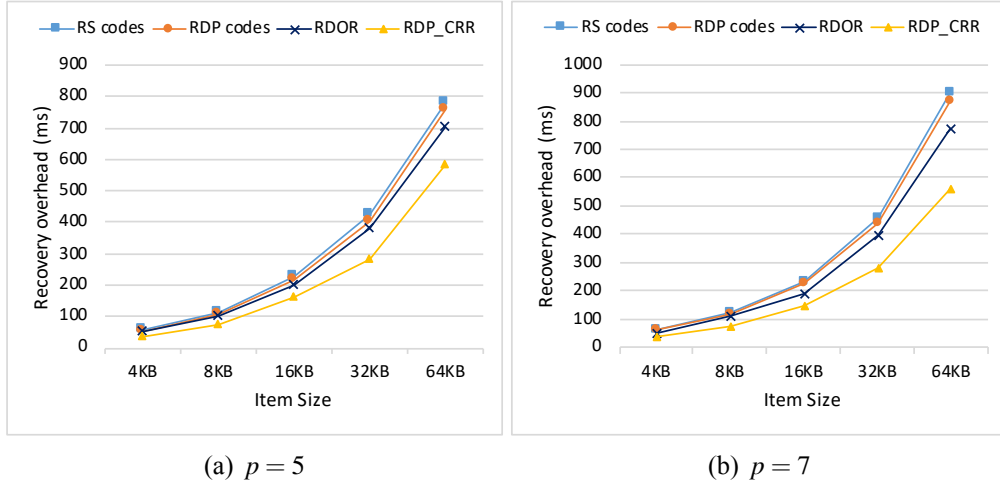


图 4.10 分布式 Memcached 在恢复数据节点过程中的总开销

用 RS 编码相当, 采用 RDOR 恢复模型的恢复总开销比采用 RS 编码分别减少 10%、20%、25% 左右, 采用 CRR 恢复模型的恢复总开销比采用 RS 编码分别减少 31%、18%、10% 左右。对于控制参数 p 为 7 的各个方案而言, 在高带宽、中带宽、低带宽的网络状态下, 采用 RDP 编码的分布式 Memcached 的恢复总开销均与采用 RS 编码相当, 采用 RDOR 恢复模型的恢复总开销比采用 RS 编码分别减少 15%、20%、25% 左右, 采用 CRR 恢复模型的恢复总开销比采用 RS 编码分别减少 35%、20%、13% 左右。总体而言, 我们可以得出三点结论。第一, 无论网络状态如何, 采用 RDP 编码的分布式 Memcached 的恢复总开销均与采用 RS 编码相当。第二, 网络带宽越低, 采用 RDOR 恢复模型的分布式 Memcached 的恢复总开销相比 RS 编码越好。第三, 网络带宽越高, 采用 CRR 恢复模型的分布式 Memcached 的恢复总开销相比 RS 编码越好。这与我们的猜测相符。

综上所述, 我们可以得出结论, CRR 恢复模型比较适用于正常带宽下的分布式 Memcached, 而 RDOR 恢复模型则更适用于低网络带宽下的分布式 Memcached。

据证明, 在相同的系统配置下, 平均数据丢失时间 (MTTDL) 与故障修复时间 (MTTR) 成反比^[54]。如公式所示 4.1:

$$MTTDL = \frac{MTTF^3}{N \times (G-1) \times (G-2) \times MTTR^2} \quad (4.1)$$

其中, MTTF 表示节点平均故障时间, G 表示条纹大小, 即本文中一个编码

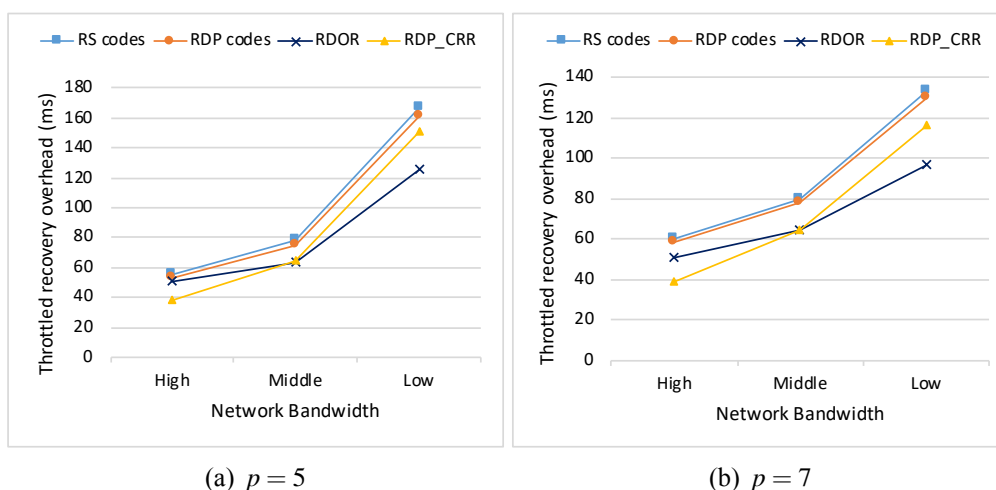


图 4.11 在不同的网络状态下，分布式 Memcached 在恢复数据节点过程中的总开销

组中的节点个数， N 表示整个分布式集群中的节点总数。

由公式4.1可知，本文所实现的分布式 Memcached 的故障恢复时间相比 Cocytus 至多减少 31%，所以 MTDDL 可以提升 210%。这显著地提升了分布式 Memcached 系统的可靠性。另一方面，如果我们想要维护一个与 RS 编码相当的可靠性，那么我们可以使用 $RDP(8,6)$ 编码来取代 $RS(6,4)$ 编码。这意味着降低 11% 的存储开销。

综上所述，减少分布式 Memcached 的故障恢复开销有以下优点：第一可以提升系统的可靠性，第二如果我们想要维持原来的可靠性，可以降低系统的存储开销。当然，还有一个显而易见的优点，就是可以提升用户体验。

第四节 本章小结

本章主要是对本文所设计并实现的分布式 Memcached 的功能及性能的测试。第一节说明了实验环境的配置。第二节展示了分布式 Memcached 的读写功能。第三节展示了分布式 Memcached 在正常运行时的读写性能，以及在系统故障模式下的重构性能。实验结果表明，RDP 编码在数据写入时的性能与 RS 编码相当；在网络带宽正常的情况下，CRR 恢复优化模型在数据恢复方面表现良好；在网络带宽较慢的情况下，RDOR 恢复优化模型在数据恢复方面表现更佳。

第五章 总结与展望

第一节 本文的总结

信息时代的到来在为我们提供大量机会的同时，也给我们带来了越来越多的挑战。大型数据库应用并发请求急剧增加，关系型数据库管理系统已经无法满足人们的性能需求。分布式内存缓存系统应运而生，而 Memcached 就是当前最流行的分布式内存缓存系统之一。分布式内存缓存系统将多台机器的内存联合成一个大型内存池，用作外部数据库的缓存。用户可以将经常访问的数据存放到分布式内存缓存系统中，这样，这部分数据的访问就可以直接从该系统的内存池中读取，而不需要经过漫长的数据库操作，这样就极大地提升了这部分数据的访问效率。

然而，在分布式系统中，节点故障是常态。而分布式内存缓存系统通常没有容错能力。当分布式内存缓存系统中的某个节点发生故障，该节点上存放的所有数据就会丢失。用户对这些数据的再次访问将不得不重新从远程数据库中读取。而远程服务器由于各种未知性，延迟可能非常高。这就会导致系统性能急剧下降，严重影响用户体验。因此，分布式内存缓存系统需要一个容错机制来保证系统的可靠性。

Cocytus 首次将 RS 编码应用到分布式 Memcached 中，实现了一个存储开销最优的具有容错能力的分布式内存缓存系统。但是，RS 编码虽然应用广泛，但是却有两个不容忽视的缺点。第一，RS 编码在编解码时的计算是在有限域上进行的，计算效率较低。第二，在数据恢复的过程中，RS 编码需要进行大量的数据传输，这在分布式系统中会导致延迟增高，性能下降。

针对 RS 编码的两个缺点，本文设计并实现了一种高效容错的分布式 Memcached。首先，本文采用 RDP 编码取代 RS 编码来实现容错。RDP 编码是一种双容错纠删码，其编解码运算只有简单的异或计算，因此性能更高。其次，本文采用了增量编码的方式来对基于 RDP 编码的分布式 Memcached 的数据更新流程进行优化，并使用了捎带更新的方案来维护数据的一致性。然后，本文采用 RDOR 恢复优化模型和 CRR 恢复优化模型来对分布式 Memcached 的数据

恢复流程进行加速。其中 RDOR 恢复优化模型可以减少数据恢复过程中传输的数据量，而 CRR 恢复模型可以将数据传输过程分配到不同的节点上并行完成，最终二者都可以减少数据恢复流程所需的时间。

最后，经过实验验证，与 RS 编码相比，RDP 编码虽然在编解码的计算时间上占优，但是由于编解码时间只占数据写入过程的一小部分，因此在 RDP 编码在数据写入时的性能与 RS 编码相当。而在数据恢复方面，经过实验验证，在网络带宽正常的情况下，CRR 恢复优化模型表现良好；在网络带宽较慢的情况下，RDOR 恢复优化模型表现更佳。

第二节 未来的展望

本文设计并实现了一个高效容错的分布式 Memcached，并对其功能及性能进行了测试。但是，针对本文工作，未来还有很多方面可以进一步探索。

1. 基于节点感知技术，对参与恢复的节点进行筛选。例如，在一个在线的系统中，某个数据节点可能正在忙碌着响应用户的其他请求，此时，如果让他参与数据恢复工作，不仅会影响到其他正在响应的请求，也会降低数据恢复的速度。因此，在数据恢复的过程中，我们应该避免选择这种繁忙的节点参与数据恢复过程。
2. 为本文设计的分布式 Memcached 加入数据迁移能力。在本文所设计的分布式 Memcached 中，故障节点的数据恢复后会存放在其替代节点上。后续可以考虑增加数据迁移能力，当有新的可用节点或者故障节点恢复连接后，将恢复后数据迁移到这些节点上。
3. 采用容错能力更高的阵列码，进一步提高系统可靠性。本文所采用的 RDP 编码是一种双容错的阵列码，最多可以容忍两个节点同时发生故障。而 STAR^[55]、HoVer^[56] 等编码却是可以实现多容错的阵列码。用户如果想要更高的可靠性，则可以引入这些编码方案。
4. 将高效容错技术应用到更多分布式内存缓存系统中。本文是基于一种典型的分布式内存缓存系统，即 Memcached，实现了高效容错。但是，分布式内存缓存系统琳琅满目且各有千秋。后续可以考虑将高效容错技术应用到更多的分布式内存缓存系统中，供用户选择。

参考文献

- [1] J Gantz, D Reinsel. Extracting value from chaos[J]. IDC iview, 2011, 1142(2011): 1 ~ 12.
- [2] S John Walker. Big data: A revolution that will transform how we live, work, and think. 2014.
- [3] T Bakuya, M Matsui. Relational database management system. US Patent 5,680,614. 1997.
- [4] B Fitzpatrick. Distributed caching with memcached[J]. Linux journal, 2004, 2004(124): 5.
- [5] J Jose, H Subramoni, M Luo, et al. Memcached design on high performance rdma capable interconnects[C]//2011 International Conference on Parallel Processing. IEEE. [S.l.]: [s.n.], 2011: 743 ~ 752.
- [6] J Zawodny. Redis: Lightweight key/value store that goes the extra mile[J]. Linux Magazine, 2009, 79.
- [7] C Do Cuong. Seattle conference on scalability: Youtube scalability[J]. Video, June, 2007.
- [8] R Nishtala, H Fugal, S Grimm, et al. Scaling Memcache at Facebook.[C]//Nsd. Vol. 13. [S.l.]: [s.n.], 2013: 385 ~ 398.
- [9] C Aniszczyk. Caching with twemcache[J]. Twitter Blog, Engineering Blog, 2012: 1 ~ 7.
- [10] N Budhiraja, K Marzullo, F B Schneider, et al. The primary-backup approach[J]. Distributed systems, 1993, 2: 199 ~ 216.
- [11] H.-Y Lin, W.-G Tzeng. A secure erasure code-based cloud storage system with secure data forwarding[J]. IEEE transactions on parallel and distributed systems, 2012, 23(6): 995 ~ 1003.
- [12] D Shankar, X Lu, D K Panda. High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads[C]//Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE. [S.l.]: [s.n.], 2017: 527 ~ 537.
- [13] J Harshan, F Oggier, A Datta. Sparsity exploiting erasure coding for resilient storage and efficient i/o access in delta based versioning systems[C]//Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on. IEEE. [S.l.]: [s.n.], 2015: 798 ~ 799.
- [14] C Huang, H Simitci, Y Xu, et al. Erasure Coding in Windows Azure Storage.[C]//Usenix annual technical conference. Boston, MA. [S.l.]: [s.n.], 2012: 15 ~ 26.
- [15] S B Wicker, V K Bhargava. Reed-Solomon codes and their applications[M]. [S.l.]: John Wiley & Sons, 1999.

-
- [16] B Sklar. Reed-solomon codes[J]. Downloaded from URL <http://www.informit.com/content/images/art.sub.-sklar7.sub.-reed-solomon-n/elementLinks/art.sub.-sklar7.sub.-reed-solomon.pdf>, 2001: 1 ~ 33.
- [17] D.-c QU, Y.-s LIU. Design of a Distributed MEM-Agent System[J]. Journal of Beijing Institute of Technology, 2005, 10: 010.
- [18] H Chen, H Zhang, M Dong, et al. Efficient and available in-memory KV-store with hybrid erasure coding and replication[J]. ACM Transactions on Storage (TOS), 2017, 13(3): 25.
- [19] R Lidl, H Niederreiter. Finite fields[M]. Vol. 20. [S.l.]: Cambridge university press, 1997.
- [20] P Corbett, B English, A Goel, et al. Row-diagonal parity for double disk failure correction[C]//Proceedings of the 3rd USENIX Conference on File and Storage Technologies. USENIX Association Berkeley, CA, USA. [S.l.]: [s.n.], 2004: 1 ~ 14.
- [21] L Xiang, Y Xu, J Lui, et al. Optimal recovery of single disk failure in RDP code storage systems[J]. ACM SIGMETRICS Performance Evaluation Review, 2010, 38(1): 119 ~ 130.
- [22] P Li, X Jin, R J Stones, et al. Parallelizing degraded read for erasure coded cloud storage systems using collective communications[C]//Trustcom/BigDataSE/I SPA, 2016 IEEE. IEEE. [S.l.]: [s.n.], 2016: 1272 ~ 1279.
- [23] S Kemp. Digital in 2018: world's Internet users pass the 4 billion mark[J]. New York, We Are Social, 2018, 30.
- [24] 中国互联网络信息中心. 第 42 次《中国互联网络发展状况统计报告》. http://www.cac.gov.cn/2018-08/20/c_1123296882.htm. Accessed August 20, 2018.
- [25] D Reinsel, J Gantz, J Rydning. Data age 2025: The evolution of data to life-critical[J]. Don't Focus on Big Data, 2017.
- [26] W contributors. Memcached. <https://en.wikipedia.org/w/index.php?title=Memcached&oldid=874291797>. Accessed December 21, 2018.
- [27] D Lee, J Choi, J.-H Kim, et al. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies[C]//ACM SIGMETRICS Performance Evaluation Review. Vol. 27. 1. ACM. [S.l.]: [s.n.], 1999: 134 ~ 143.
- [28] D Eastlake 3rd, P Jones. US secure hash algorithm 1 (SHA1). Tech. rep. 2001.
- [29] R Rivest. The MD5 message-digest algorithm. Tech. rep. 1992.
- [30] D Karger, E Lehman, T Leighton, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web[C]//Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM. [S.l.]: [s.n.], 1997: 654 ~ 663.
- [31] I S Reed, G Solomon. Polynomial codes over certain finite fields[J]. Journal of the society for industrial and applied mathematics, 1960, 8(2): 300 ~ 304.

- [32] M Sathiamoorthy, M Asteris, D Papailiopoulos, et al. Xoring elephants: Novel erasure codes for big data[C]//Proceedings of the VLDB Endowment. Vol. 6. 5. VLDB Endowment. [S.l.]: [s.n.], 2013: 325 ~ 336.
- [33] W Halbawi, N Azizan, F Salehi, et al. Improving distributed gradient descent using reed-solomon codes[C]//2018 IEEE International Symposium on Information Theory (ISIT). IEEE. [S.l.]: [s.n.], 2018: 2027 ~ 2031.
- [34] D Kalman. The generalized Vandermonde matrix[J]. Mathematics Magazine, 1984, 57(1): 15 ~ 21.
- [35] J S Plank, L Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications[C]//Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on. IEEE. [S.l.]: [s.n.], 2006: 173 ~ 180.
- [36] 朱卫卫, 杨金民. 基于二进制矩阵的 RS 编码优化算法[J]. 计算机工程, 2011, 37(23): 57 ~ 59.
- [37] T White. Hadoop-The Definitive Guide: Storage and Analysis at Internet Scale (revised and updated). 2012.
- [38] W Litwin, R Moussa, T Schwarz. LH* RS—a highly-available scalable distributed data structure[J]. ACM Transactions on Database Systems (TODS), 2005, 30(3): 769 ~ 811.
- [39] C Lai, S Jiang, L Yang, et al. Atlas: Baidu's key-value storage system for cloud data[C]//Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on. IEEE. [S.l.]: [s.n.], 2015: 1 ~ 14.
- [40] D J Costello, S Lin. Error Control Coding: Fundamentals and Applications. 1982.
- [41] S Li, Q Zhang, Z Yang, et al. BCStore: bandwidth-efficient in-memory KV-store with batch coding[C]//Proceedings of International Conference on Massive Storage Systems and Technology, MSST. [S.l.]: [s.n.], 2017: 1 ~ 13.
- [42] J Xia, J Huang, X Qin, et al. Revisiting Updating Schemes for Erasure-Coded In-Memory Stores[C]//Networking, Architecture, and Storage (NAS), 2017 International Conference on. IEEE. [S.l.]: [s.n.], 2017: 1 ~ 6.
- [43] S Wang, J Huang, X Qin, et al. WPS: A Workload-Aware Placement Scheme for Erasure-Coded In-Memory Stores[C]//Networking, Architecture, and Storage (NAS), 2017 International Conference on. IEEE. [S.l.]: [s.n.], 2017: 1 ~ 10.
- [44] M M Yiu, H H Chan, P P Lee. Erasure coding for small objects in in-memory KV storage[C]//Proceedings of the 10th ACM International Systems and Storage Conference. ACM. [S.l.]: [s.n.], 2017: 14.
- [45] initrdmk. Cocytus. <https://ipads.se.sjtu.edu.cn/pub/projects/cocytus>. Accessed February 22, 2016.
- [46] M Blaum, J Brady, J Bruck, et al. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures[J]. IEEE Transactions on computers, 1995, 44(2): 192 ~ 202.
- [47] M Blaum, J Bruck, A Vardy. MDS array codes with independent parity symbols[J]. IEEE Transactions on Information Theory, 1996, 42(2): 529 ~ 542.

- [48] C Canudas-de-Wit, F R Rubio, J Fornes, et al. Differential coding in networked controlled linear systems[C]//American Control Conference, 2006. IEEE. [S.l.]: [s.n.], 2006: 6 ~ pp.
- [49] Y Al-Houmaily, P Chrysanthis. Two-phase commit in gigabit-networked distributed databases[C]//Int. Conf. on Parallel and Distributed Computing Systems (PDCS). [S.l.]: [s.n.], 1995.
- [50] K Rashmi, N B Shah, D Gu, et al. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster.[C]//HotStorage. [S.l.]: [s.n.], 2013.
- [51] B F Cooper, A Silberstein, E Tam, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. ACM. [S.l.]: [s.n.], 2010: 143 ~ 154.
- [52] A Kuznetsov, S Hemminger. Iproute2: Utilities for Controlling TCP[J]. IP Networking and Traffic, 2012.
- [53] J S Plank, S Simmerman, C D Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications[J]. Technical Report CS-07-603, University of Tennessee, 2007.
- [54] P M Chen, E K Lee, G A Gibson, et al. RAID: High-performance, reliable secondary storage[J]. ACM Computing Surveys (CSUR), 1994, 26(2): 145 ~ 185.
- [55] C Huang, L Xu. STAR: An efficient coding scheme for correcting triple storage node failures[J]. IEEE Transactions on Computers, 2008, 57(7): 889 ~ 901.
- [56] J L Hafner. HoVer erasure codes for disk arrays[C]//International Conference on Dependable Systems and Networks (DSN'06). IEEE. [S.l.]: [s.n.], 2006: 217 ~ 226.

致谢

距离走出南开，步入社会的日子越来越近了，心中百感交集。有对同窗的不舍，有对校园的依赖，有对工作的抵触，又有对新生活的好奇。纵使前方大雾弥漫，我们依然要勇敢地走过去。从 2012 年到如今的 2019 年，从八里台校区到如今的津南校区，从刚入大学的本科生到如今的研究生毕业，作为一个 7 年的南开人，我无比地自豪。今后无论走到哪里，我都会坚定南开“允公允能，日新月异”的校训；无论走到哪里，我都要大声地说一声“我是爱南开的”。2019 年是南开大学建校 100 周年。我们这届学生将带着这 100 年的历史，怀着 100 分的信心，扬帆起航。

在这里，我想感谢我的导师王刚教授，您渊博的学识和对学术一丝不苟的态度深深地影响了我，感谢您为我的人生指明了方向。感谢刘晓光教授和李雨森老师，你们对互联网发展的大局观大大地开阔了我的眼界。感谢李忠伟老师、任明明老师、苏明老师和 Rebecca 老师，你们在学习和生活上给予我许多帮助，使我受益匪浅。

另外，我想感谢实验室的师兄师姐，师弟师妹，你们的陪伴使我的生活充满乐趣。感谢李鹏师兄和沈璐师姐，是你们的耐心讲解和无私帮助，使我在学术的道路上一步步前进。感谢同届的黄晓敏、郜姝妮、刘博和刘学达，我们在这三年里一起欢笑，一起学习，一起进步。感谢于朝阳师妹和张佳辰师弟，我们一起出国开会的三天时光就像冬日里一股暖阳，对我弥足珍贵。

然后，我想感谢我的爸爸妈妈。感谢你们这 20 多年的养育之恩，是你们的不断鼓励和支持，让我充满勇气，离开家庭，走向校园，走向社会。因为我知道，我的背后永远有一个避风港。

最后，再次对母校、对老师、对同学、对家人表示感谢。祝母校日新月异，桃李满天下；祝老师们家庭和睦，工作顺利；祝各位同学学业有成，前程似锦；祝爸爸妈妈身体健康，生活美满。

个人简历

基本信息

姓 名：赵帅兵 性别：男 出生年月：1994 年 3 月
电子邮箱：zhaoshb@nbjl.nankai.edu.cn
通信地址：天津市津南区同砚路 38 号南开大学津南校区计算机学院 409 室
(300350)

教育背景

2016.9-2019.6 南开大学计算机学院 计算机科学与技术 工学硕士
2012.9-2016.6 南开大学计算机学院 计算机科学与技术 工学学士

工作经历

2018.6-2019.10 阿里巴巴 搜索事业部 后台研发

学术论文

孔琰, 赵帅兵, 刘若琳, 梁爽, 庄园, 冯世舫, 王刚, 刘晓光, 李忠伟. 基于安卓平台的多云存储系统 [J]. 计算机应用, 2017, 37(A01): 39-44.

Shuaibing Zhao, Lu Shen, Yusen Li, Rebecca J. Stones, Gang Wang and Xiaoguang Liu.
An Efficient Fault Tolerance Framework for Distributed In-memory Caching Systems.
International Conference on Parallel and Distributed Systems, 2017