TURING 图灵程序设计丛书

NoSQL

NOT ONLY SQL

数据库入门

[日] 佐佐木达也 著
罗勇 译

- 了解当今最炙手可热的NoSQL新型数据库技术
- 介绍memcached、Tokyo Tyrant、Redis、MongoDB
- 如何基于MySQL应用NoSQL技术特性

人民邮电出版社
POSTS & TELECOM PRESS

NoSQL

NOT ONLY SQL

数据库入门

在云计算时代，传统的关系型数据库的不足凸显出来，尤其是它无法应对大数据量的处理需求。为了弥补这些不足，NoSQL型数据库应运而生，以MongoDB、Hadoop为代表的NoSQL产品以其高性能、强扩展性和高容错性为大家所称道，并在数据库领域掀起了一场新的革命。

本书是一本NoSQL入门书，从最基本的NoSQL发展史开始，介绍了memcached、Tokyo Tyrant、Redis和MongoDB这4种NoSQL数据库的使用背景、优缺点和具体应用实例，并对这4种数据库进行了互相比较，旨在让读者全面了解NoSQL能解决的具体问题，为读者开发数据库提供更多选择。书中最后还介绍了如何将MySQL数据库NoSQL化。

本书适合所有数据库开发人员。

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/数据库/NoSQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-27950-7

ISBN 978-7-115-27950-7
定价：45.00元

TURING 图灵程序设计丛书

NoSQL

NOT ONLY SQL

数据库入门

[日] 佐佐木达也 著
罗勇 译

人民邮电出版社
北京

前　言

最近，到处都能听到“NoSQL”，这个词到底是什么意思呢？“NoSQL”到底能给我们带来什么好处呢？

现在，提起数据存储，一般都是针对关系型数据库来说的。但是，关系型数据库并不是万能的，它对于某些处理依然是很吃力的。本书所讲述的 NoSQL 数据库就是为了弥补关系型数据库的不足应运而生的。在适当的情况下使用 NoSQL 数据库，可以为关系型数据库需要耗费大量时间才能完成的处理，提供高速、合理的解决方案。

预想的适用情况

NoSQL 数据库可用于：

- **取代关系型数据库的弱势处理（比如大量数据的写入处理等）**
- **作为关系型数据库之外的另一种选择**

NoSQL 数据库虽然可以替关系型数据库分担一些难题（如大量数据的写入等），但这其中的操作具有相当的难度。因此本书仅仅将 NoSQL 定位为“关系型数据库之外的另一种选择”。

读者对象

本书面向的读者群为有 1 年关系型数据库开发经验的软件工程师和程序员。因为他们对数据量的增大给关系型数据库的检索和更新处理带来的剧烈性能恶化有切身感受，也能更深刻地理解 NoSQL 数据库的优势。

本书将为读者介绍 NoSQL 数据库以及使用 NoSQL 数据库所带来的便利。

本书内容

本书共由 5 章内容组成，下面对各章内容做一个简单的介绍。

第 1 章首先介绍什么是 NoSQL 以及这个词的来源。之后，会介绍关系型数据库的发展历史、关系型数据库的优势和不足，以及 NoSQL 数据库诞生的背景，以便使读者了解 NoSQL 数据库的发展历史。

这一章亦会对 NoSQL 数据库的种类和特征做简单讲解。同时阐述“怎么样才能更好地区别使用关系型数据库和 NoSQL 数据库”，为 NoSQL 数据库的引入奠定基础。

第 2 章将会讲述 memcached、Tokyo Tyrant、Redis 和 MongoDB 这 4 种 NoSQL 数据库，介绍这些 NoSQL 数据库各自的使用背景、特征和用例，以及它们的实际应用。通过本章的介绍，读者能够了解 NoSQL 数据库的基本使用方法。

第 3 章对上述 4 种 NoSQL 数据库的应用实例以及实现代码进行具体的介绍。这一章将使大家对这 4 种 NoSQL 数据库能解决的具体、实际的问题有所了解，让 NoSQL 数据库成为解决问题的选择之一。

虽然本章涉及的各个实例都可以用关系型数据库来实现，但是使用 NoSQL 数据库能够获得更快的响应，同时简单的操作也给使用者带来更大的便利。

第 4 章对上述 4 种 NoSQL 数据库的性能进行比较。本章不仅会进行与基本 CRUD 处理（创建、检索、更新、删除）相关的性能比较，而且还会着眼于一些具体实例，例如像 Tokyo Tyrant 的 addint 方法和 incr 方法的性能比较，Redis 的 list 类型的插入和删除的性能比较，以及 MySQL 的 join 和 MongoDB 的 embed 的性能比较等。

第 5 章对 NoSQL 数据库在实际应用中的问题点，以及 HandlerSocket 解决方案进行介绍。它虽然和 NoSQL 数据库略有一点不同，却是个非常有意思的解决方案，有利于读者们开阔视野。

版 权 声 明

目　　录

NoSQL 数据库的基础知识

本章首先介绍关系型数据库的起源、特征，以及它的优缺点，进而引出 NoSQL 数据库。

然后介绍 NoSQL 数据库的种类，以及如何更好地区别使用关系型数据库和 NoSQL 数据库。

1.1 关系型数据库和NoSQL数据库

1.1.1 什么是NoSQL

大家有没有听说过“NoSQL”呢？近年，这个词极受关注。看到“NoSQL”这个词，大家可能会误以为是“No! SQL”的缩写，并深感诧异：“SQL怎么会没有必要了呢?”但实际上，它是“Not Only SQL”的缩写。它的意义是：**适用关系型数据库[①]的时候就使用关系型数据库，不适用的时候也没有必要非使用关系型数据库不可，可以考虑使用更加合适的数据存储。**

为弥补关系型数据库的不足，各种各样的NoSQL数据库应运而生。

为了更好地了解本书所介绍的NoSQL数据库，对关系型数据库的理解是必不可少的。那么，就让我们先来看一看关系型数据库的历史、分类和特征吧。

1.1.2 关系型数据库简史

1969年，埃德加·弗兰克·科德（Edgar Frank Codd）发表了一篇划时代的论文，首次提出了关系数据模型的概念。但可惜的是，刊登论文的“IBM Research Report”只是IBM公司的内部刊物，因此论文反响平平。1970年，他再次在刊物《*Communication of the ACM*》上发表了题为“A Relational Model of Data for Large Shared Data banks”（大型共享数据库的关系模型）的论文，终于引起了大家的关注。

科德所提出的关系数据模型的概念成为了现今关系型数据库的基础。当时的关系型数据库由于硬件性能低劣、处理速度过慢而迟迟没有得到实际应用。但之后随着硬件性能的提升，加之使用简单、性能优越等优点，关系型数据库得到了广泛的应用。

① 关系数据库包括Oracle、Microsoft SQL、DB2、PostgreSQL等，本书主要指MySQL。

1.1.3 数据库的分类

数据库根据不同的数据模型（数据的表现形式）主要分成阶层型、网络型和关系型 3 种。

阶层型数据库

早期的数据库称为阶层型数据库，数据的关系都是以简单的树形结构来定义的。程序也通过树形结构对数据进行访问。这种结构，父记录（上层的记录）同时拥有多个子记录（下层记录），子记录只有唯一的父记录。正因为如此，这种非常简单的构造在碰到复杂数据的时候往往会造成数据的重复（同一数据在数据库内重复出现），出现数据冗余的问题。图 1-1 所示为阶层型数据库。

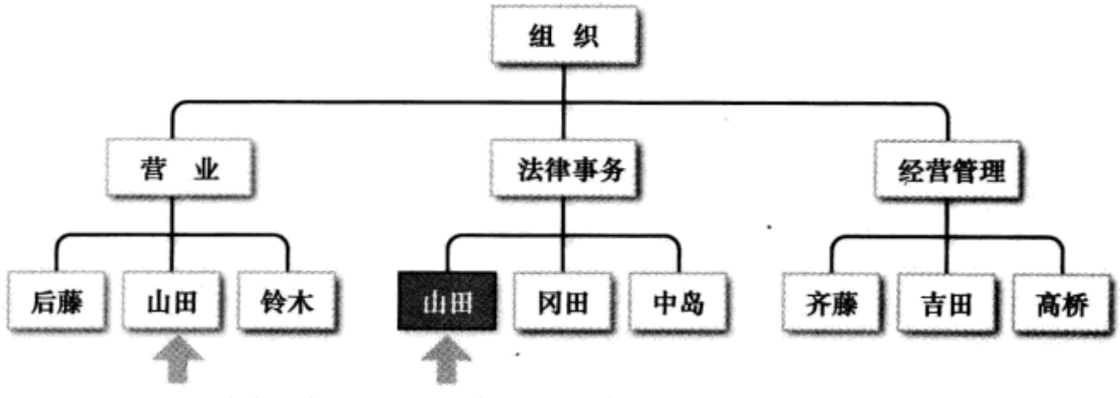


图 1-1 阶层型数据库示例

阶层型数据库把数据通过阶层结构的方式表现出来，虽然这样的结构有利于提高查询效率，但与此相对应的是，不理解数据结构就无法进行高效的查询。当然，在阶层结构发生变更的时候，程序也需要进行相应的变更。

网络型数据库

如前所述，阶层型数据库会带来数据重复的问题。为了解决这个问题，就出现了网络型数据库。它拥有同阶层型数据库相近的数据结构，同时各种数据又如同网状交织在一起，因此而得名。

阶层型数据库只能通过父子关系来表现数据之间的关系。针对这一不足，网络型数据库可以使子记录同时拥有多个父记录，从而解决了数据冗余的问题。图 1-2 所示为网络型数据库。

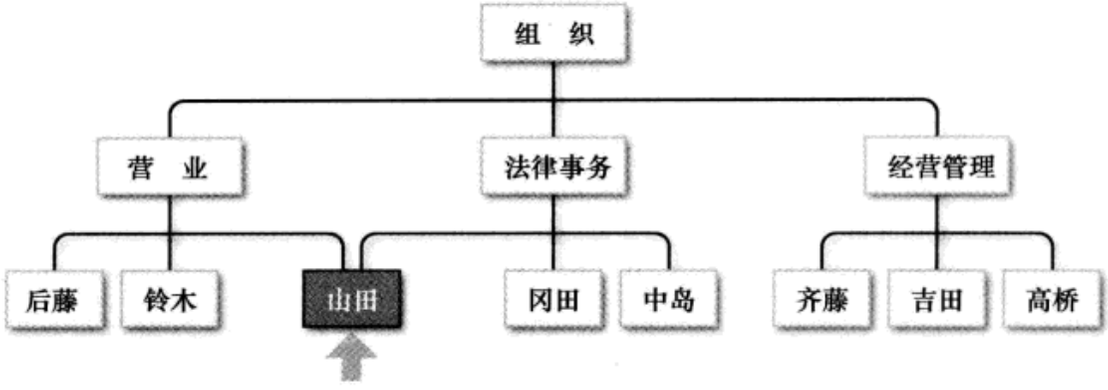


图 1-2 网络型数据库事例

但是，在网络型数据库中，数据间比较复杂的网络关系使得数据结构的更新变得比较困难。另外，与阶层型数据库一样，网络型数据库对数据结构有很强的依赖性，不理解数据结构就无法进行相应的数据访问。

关系型数据库

最后要向大家介绍的是以科德提出的关系数据模型为基础的关系型数据库。关系型数据库把所有的数据都通过行和列的二元表现形式表示出来，给人更容易理解的直观感受。网络型数据库存在着数据结构变更困难的问题，而关系型数据库可以使多条数据根据值来进行关联，这样就使数据可以独立存在，使得数据结构的变更变得简单易行。

对于阶层型数据库和网络型数据库，如果不理解相应的数据结构，就无法对数据进行读取，它们对数据结构的依赖性很强。因此，它们往往需要专业的工程师使用特定的计算机程序进行操作处理。相反，关系型数据库将作为操作对象的数据和操作方法（数据之间的关联）分离开来，消除了对数据结构的依赖性，让数据和程序的分离成为可能。这使得数据库可以广泛应用于各个不同领域，进一步扩大了数据库的应用范围。

参考资料

《RDB 技术》（日经 BP《ITpro》“技术广场” 2004/07/27）
http://itpro.nikkeibp.co.jp/members/NBY/techsquare/20040716/1/
沖冠吾《从零开始学数据建模》（ITmedia《@IT》“Database Expert” 2008/9/10）
http://www.atmarkit.co.jp/fdb/index/subindex/basic_modeling.html
铃木浩司《数据库有很多种，你能全都说出来吗?》（朝日互动 ZDNet Japan | builder 2007/09/11）
http://builder.japan.zdnet.com/sp/07database/story/0,3800082819,20356199,00.htm

1.1.4 关系型数据库的优势

通用性及高性能

虽然本书是讲解 NoSQL 数据库的，但有一个重要的大前提，请大家一定不要误解。这个大前提就是“关系型数据库的性能绝对不低，它具有非常好的通用性和非常高的性能”。毫无疑问，对于绝大多数的应用来说它都是最有效的解决方案。

突出的优势

关系型数据库作为应用广泛的通用型数据库，它的突出优势主要有以下几点：

- **保持数据的一致性（事务处理）**
- **由于以标准化为前提，数据更新的开销很小（相同的字段基本上都只有一处）**
- **可以进行 JOIN 等复杂查询**
- **存在很多实际成果和专业技术信息（成熟的技术）**

这其中，能够保持数据的一致性是关系型数据库的最大优势。在需要严格保证数据一致性和处理完整性的情况下，用关系型数据库是肯定没有错的。但是有些情况不需要 JOIN，对上述关系型数据库的优点也没有什么特别需要，这时似乎也就没有必要拘泥于关系型数据库了。

1.1.5 关系型数据库的不足

不擅长的处理

就像之前提到的那样，关系型数据库的性能非常高。但是它毕竟是一个通用型的数据库，并不能完全适应所有的用途。具体来说它并不擅长以下处理：

- **大量数据的写入处理**
- **为有数据更新的表做索引或表结构（schema）变更**
- **字段不固定时应用**
- **对简单查询需要快速返回结果的处理**

下面逐一进行详细的说明。

大量数据的写入处理

在数据读入方面，由复制产生的主从模式（数据的写入由主数据库负责，数据的读入由从数据库负责），可以比较简单地通过增加从数据库来实现规模化。但是，在数据的写入方面却完全没有简单的方法来解决规模化问题[①]。例如，要想将数据的写入规模化，可以考虑把主数据库从一台增加到两台，作为互相关联复制的二元主数据库来使用。确实这样似乎可以把每台主数据库的负荷减少一半，但是更新处理会发生冲突（同样的数据在两台服务器同时更新成其他值），可能会造成数据的不一致。为了避免这样的问题，就需要把对每个表的请求分别分配给合适的主数据库来处理，这就不那么简单了。图 1-3 所示为两台主机问题。图 1-4 所示为二元主数据库问题的解决办法。

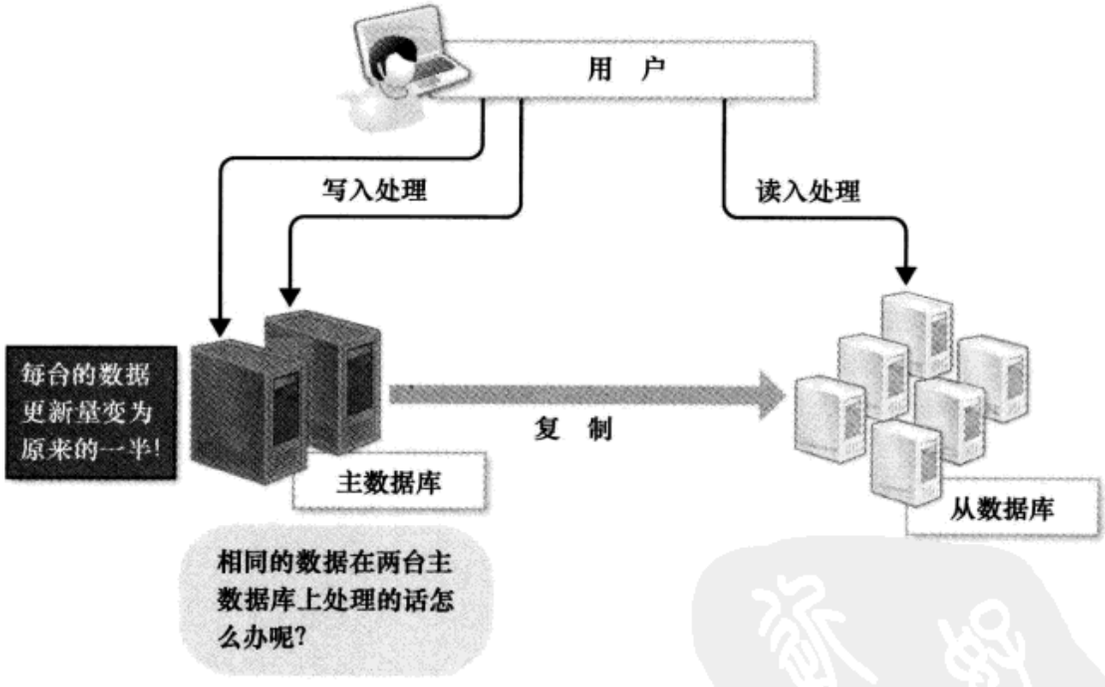


图 1-3 两台主机问题

另外也可以考虑把数据库分割开来，分别放在不同的数据库服务器上，比如将这个表放在这个数据库服务器上，那个表放在那个数据库服务器上。数据库分割可以减少每台数据库服务器上的数据量，以便减少硬盘 I/O（输入/输出）处理，实现内存上的高速处理，效果非常显著。但是，由于分别存储在不同服务器上的表之间无法进行 JOIN 处理，数据库分割的时候就需要预先考虑这些问题。数据库分割之

① 读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来实现读写分离，以提高读写性能和读库的可扩展性。Mysql 的 master-slave 模式成为了这个时候的网站标配。——译者注

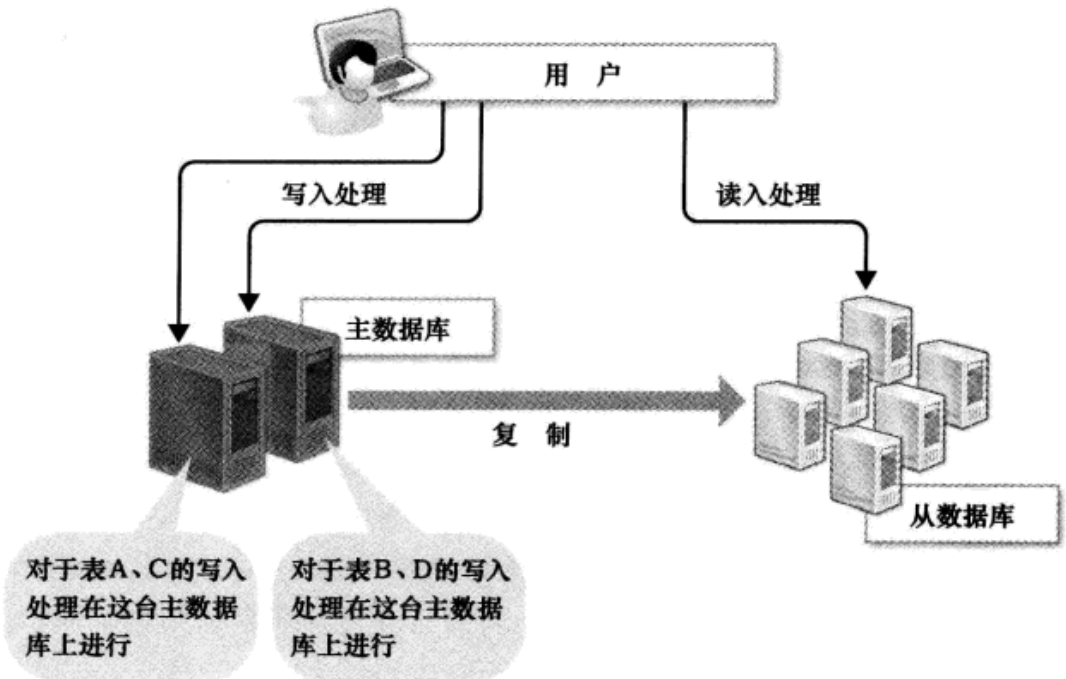


图 1-4 二元主数据库问题的解决办法

后，如果一定要进行 JOIN 处理，就必须要在程序中进行关联，这是非常困难的。图 1-5所示为数据库分割。

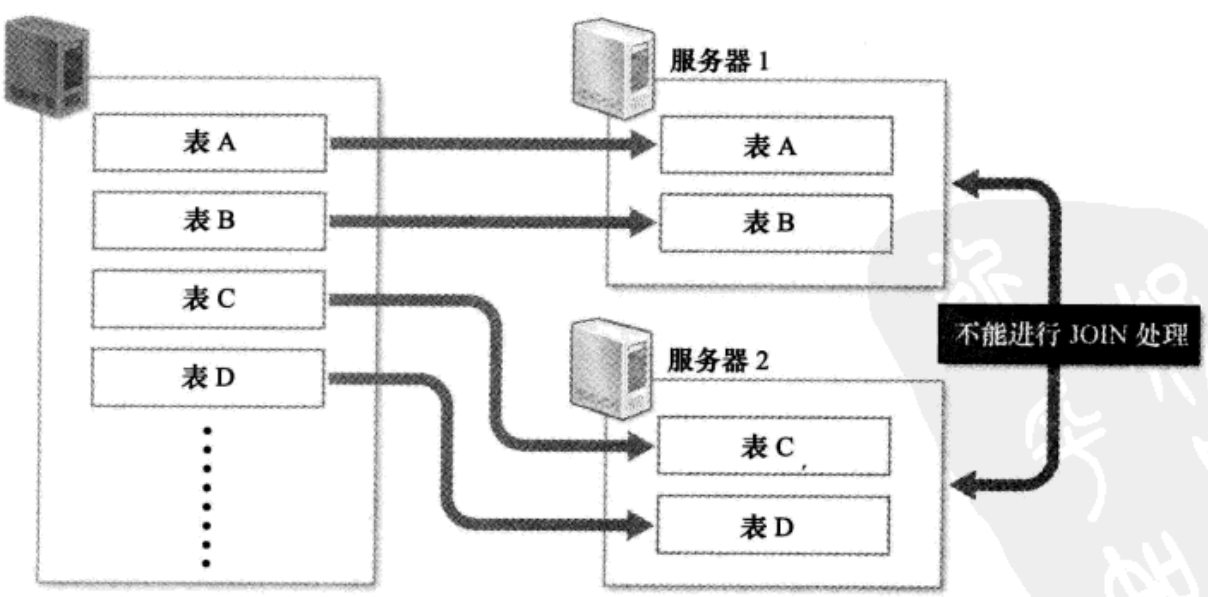


图 1-5 数据库分割

为有数据更新的表做索引或表结构（schema）变更

在使用关系型数据库时，为了加快查询速度需要创建索引，为了增加必要的字段就一定需要改变表结构。为了进行这些处理，需要对表进行共享锁定，这期间数

据变更（更新、插入、删除等）是无法进行的。如果需要进行一些耗时操作（例如为数据量比较大的表创建索引或者是变更其表结构），就需要特别注意：长时间内数据可能无法进行更新。表 1-1 所示为共享锁和排他锁。

表 1-1 共享锁和排他锁

名称	锁的影响范围	别名
共享锁	其他连接可以对数据进行读取但是不能修改数据	读锁
排他锁	其他连接无法对数据进行读取和修改操作	写锁

字段不固定时的应用

如果字段不固定，利用关系型数据库也是比较困难的。有人会说“需要的时候，加个字段就可以了”，这样的方法也不是不可以，但在实际运用中每次都进行反复的表结构变更是非常痛苦的。你也可以预先设定大量的预备字段，但这样的话，时间一长很容易弄不清楚字段和数据的对应状态（即哪个字段保存哪些数据），所以并不推荐使用。图 1-6 所示为使用预备字段的情况。

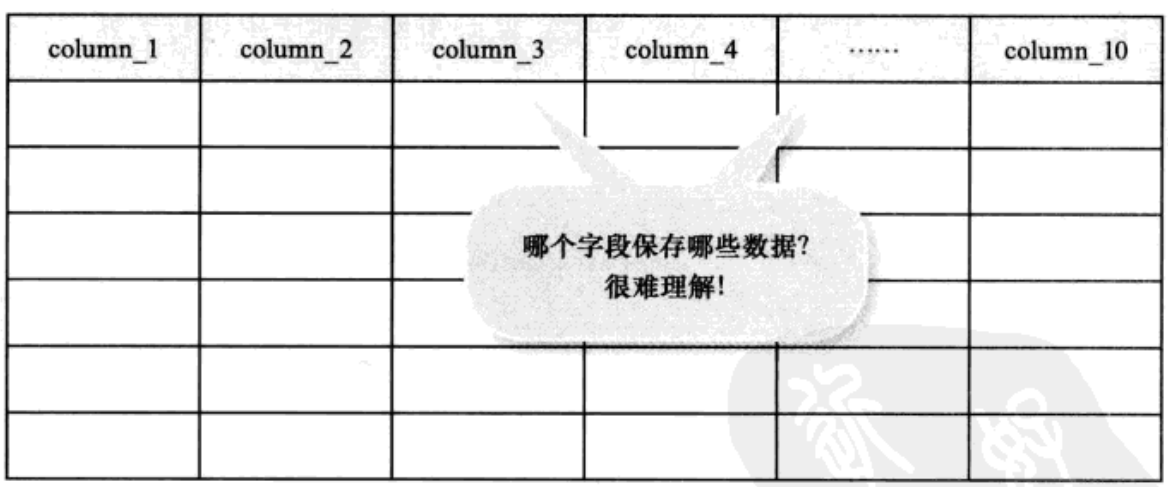


图 1-6 在使用预备字段的情况下

对简单查询需要快速返回结果的处理

最后还有一点，这点似乎称不上是缺点，但不管怎样，关系型数据库并不擅长对简单[①]的查询快速返回结果。因为关系型数据库是使用专门的 SQL 语言进行数据读取的，它需要对 SQL 语言进行解析，同时还有对表的锁定和解锁这样的额外开销。这里并不是说关系型数据库的速度太慢，而只是想告诉大家若希望对简单查询

① 这里所说的“简单”指的是没有复杂的查询条件，而不是用 JOIN 的意思。

进行高速处理，则没有必要非用关系型数据库不可。

在这种情况下，我想推荐大家使用 NoSQL 数据库。但是像 MySQL 提供了利用 HandlerSocket 这样的变通方法，也是可行的。虽然使用的是关系型数据库 MySQL，但并没有利用 SQL 而是直接进行数据访问。这样的方法是非常快速的。关于 HandlerSocket，将在第 5 章进行详细介绍。图 1-7 所示为 HandlerSocket 的概要。

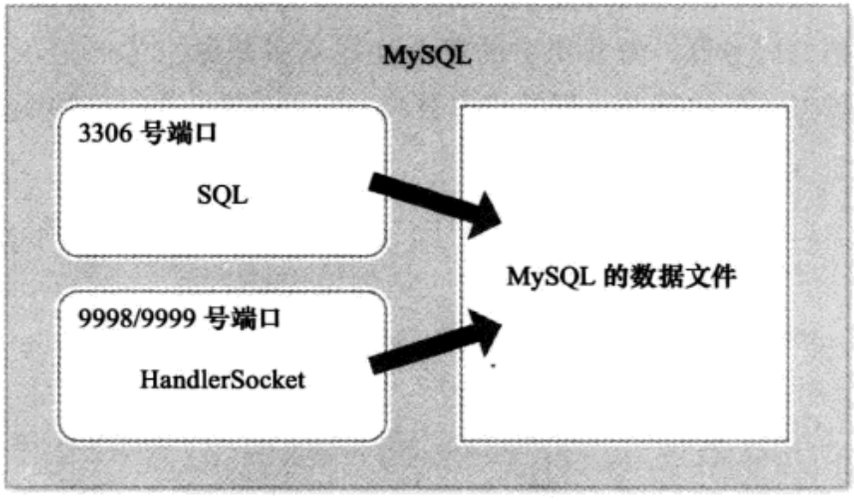


图 1-7 HandlerSocket 概要

1.1.6 NoSQL 数据库

1.1.5 节介绍了关系型数据库的不足之处。为了弥补这些不足（特别是最近几年），NoSQL 数据库出现了。关系型数据库应用广泛，能进行事务处理和 JOIN 等复杂处理。相对地，NoSQL 数据库只应用在特定领域，基本上不进行复杂的处理，但它恰恰弥补了之前所列举的关系型数据库的不足之处。

易于数据的分散

如前所述，关系型数据库并不擅长大量数据的写入处理。原本关系型数据库就是以 JOIN 为前提的，就是说，各个数据之间存在关联是关系型数据库得名的主要原因。为了进行 JOIN 处理，关系型数据库不得不把数据存储在同一个服务器内，这不利于数据的分散。相反，NoSQL 数据库原本就不支持 JOIN 处理，各个数据都是独立设计的，很容易把数据分散到多个服务器上。由于数据被分散到了多个服务器上，减少了每个服务器上的数据量，即使要进行大量数据的写入操作，处理起来也更加容易。同理，数据的读入操作当然也同样容易。

提升性能和增大规模

下面说一点题外话，如果想要使服务器能够轻松地处理更大量的数据，那么只有两个选择：一是提升性能，二是增大规模。下面我们来整理一下这两者的不同。

首先，提升性能指的就是通过提升现行服务器自身的性能来提高处理能力。这是非常简单的方法，程序方面也不需要进行变更，但需要一些费用。若要购买性能翻倍的服务器，需要花费的资金往往不只是原来的 2 倍，可能需要多达 5～10 倍。这种方法虽然简单，但是成本较高。图 1-8 所示为提升性能的费用与性能曲线。

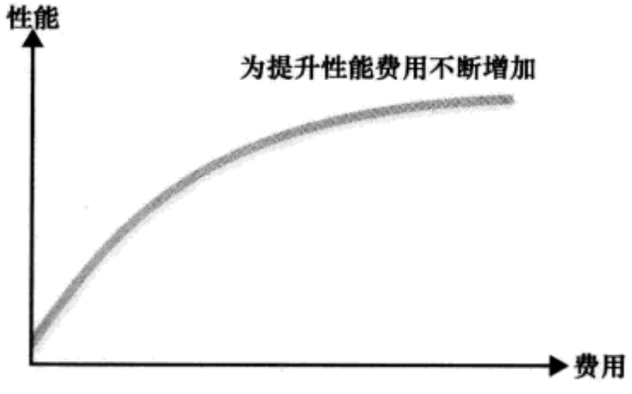


图 1-8 提升性能的费用与性能曲线

另一方面，增大规模指的是使用多台廉价的服务器来提高处理能力。它需要对程序进行变更，但由于使用廉价的服务器，可以控制成本。另外，以后只要依葫芦画瓢增加廉价服务器的数量就可以了。图 1-9 所示为提升性能和增大规模。

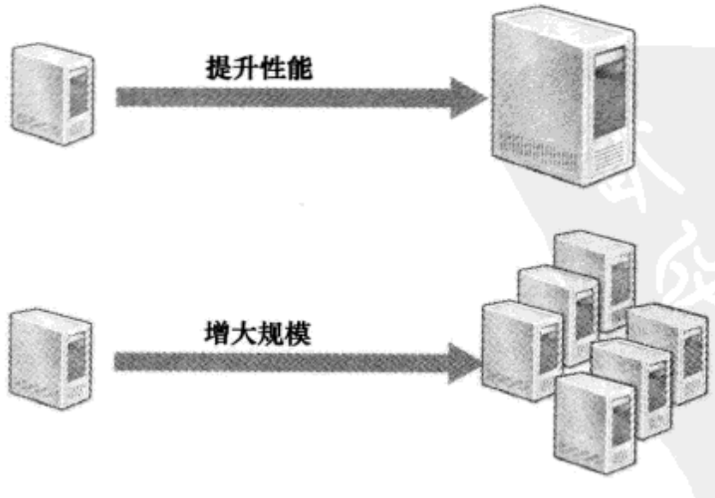


图 1-9 提升性能和增大规模

不对大量数据进行处理的话就没有使用的必要吗?

NoSQL 数据库基本上来说为了“使大量数据的写入处理更加容易（让增加服务器数量更容易）”而设计的。但如果不是对大量数据进行操作的话，NoSQL 数据库的应用就没有意义吗?

答案是否定的。的确，它在处理大量数据方面很有优势。但实际上 NoSQL 数据库还有各种各样的特点，如果能够恰当地利用这些特点，它就会非常有用。具体的例子将会在第 2 章和第 3 章进行介绍，这些用途将会让你感受到利用 NoSQL 的好处。

- **希望顺畅地对数据进行缓存（Cache）处理**
- **希望对数组类型的数据进行高速处理**
- **希望进行全部保存**

多样的 NoSQL 数据库

NoSQL 数据库存在着“键值存储”、“文档型数据库”、“列存储数据库”等各种各样的种类，每种数据库又包含各自的特点。下一节让我们一起来了解一下 NoSQL 数据库的种类和特点。

1.2 NoSQL数据库是什么

Chapter 1

NoSQL说起来简单，但实际上到底有多少种呢？我在提笔的时候，到NoSQL的官方网站上确认了一下，竟然已经有122种了。另外官方网站上也介绍了本书没有涉及到的图形数据库和对象数据库等各个类别。不知不觉间，原来已经出现了这么多的NoSQL数据库啊。

本节将为大家介绍具有代表性的NoSQL数据库。图1-10所示为http://nosql-database，表1-2所示为典型的NoSQL数据库。

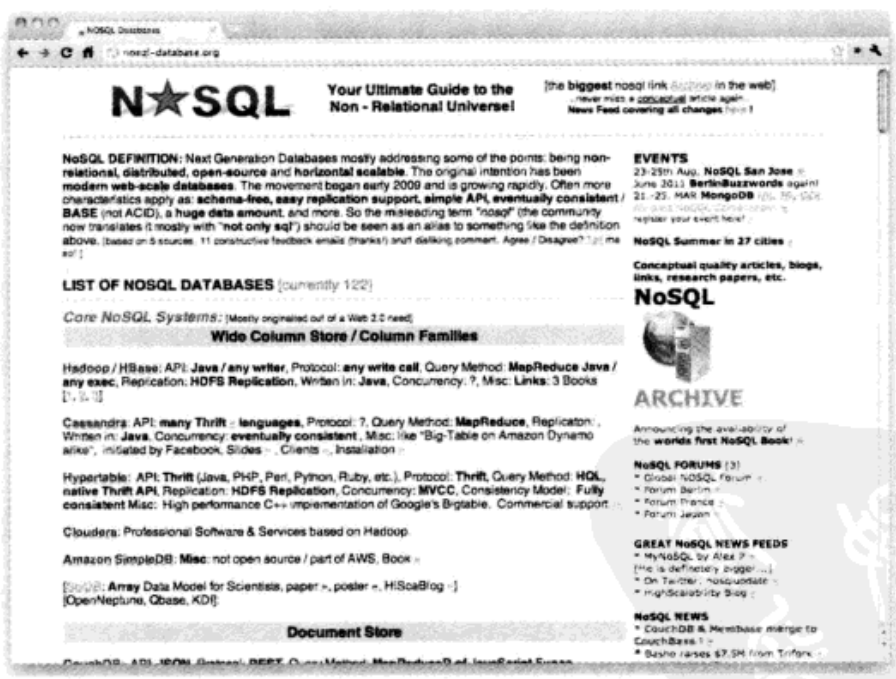


图1-10 http://nosql-database.org/

表1-2 典型的NoSQL数据库

临时性键值存储	永久性键值存储	面向文档的数据库	面向列的数据库
memcached	Tokyo Tyrant	MongoDB	Cassandra
(Redis)	Flare	CouchDB	HBase
	ROMA		HyperTable
	(Redis)		

1.2.1 键值存储

这是最常见的 NoSQL 数据库，它的数据是以键值的形式存储的。虽然它的处理速度非常快，但是基本上只能通过键的完全一致查询获取数据[①]。根据数据的保存方式可以分为临时性、永久性和两者兼具 3 种。

临时性

属于这种类型。所谓临时性就是“数据有可能丢失”的意思。memcached 把所有数据都保存在内存中，这样保存和读取的速度非常快，但是当 memcached 停止的时候，数据就不存在了。由于数据保存在内存中，所以无法操作超出内存容量的数据（旧数据会丢失）。

- **在内存中保存数据**
- **可以进行非常快速的保存和读取处理**
- **数据有可能丢失**

永久性

Tokyo Tyrant、Flare、ROMA 等属于这种类型。和临时性相反，所谓永久性就是“数据不会丢失”的意思。这里的键值存储不像 memcached 那样在内存中保存数据，而是把数据保存在硬盘上。与 memcached 在内存中处理数据比起来，由于必然要发生对硬盘的 IO 操作，所以性能上还是有差距的。但数据不会丢失是它最大的优势。

- **在硬盘上保存数据**
- **可以进行非常快速的保存和读取处理（但无法与 memcached 相比）**
- **数据不会丢失**

两者兼具

Redis 属于这种类型。Redis 有些特殊，临时性和永久性兼具，且集合了临时性键值存储和永久性键值存储的优点键值。Redis 首先把数据保存到内存中，在满足特定条件（默认是 15 分钟一次以上，5 分钟内 10 个以上，1 分钟内 10 000 个以上的键发生变更）的时候将数据写入到硬盘中。这样既确保了内存中数据的处理速度，又

① Tokyo Tyrant 的一部分数据类型和 Redis 也可以通过正则表达式来进行查询。

可以通过写入硬盘来保证数据的永久性。这种类型的数据库特别适合于处理数组类型的数据。

- 同时在内存和硬盘上保存数据
- 可以进行非常快速的保存和读取处理
- 保存在硬盘上的数据不会消失（可以恢复）
- 适合于处理数组类型的数据

1.2.2 面向文档的数据库

MongoDB、CouchDB属于这种类型。它们属于NoSQL数据库，但与键值存储相异。

不定义表结构

面向文档的数据库具有以下特征：即使不定义表结构，也可以像定义了表结构一样使用。关系型数据库在变更表结构时比较费事，而且为了保持一致性还需修改程序。然而NoSQL数据库则可省去这些麻烦（通常程序都是正确的），确实是方便快捷。

可以使用复杂的查询条件

跟键值存储不同的是，面向文档的数据库可以通过复杂的查询条件来获取数据。虽然不具备事务处理和JOIN这些关系型数据库所具有的处理能力，但除此以外的其他处理基本上都能实现。这是非常容易使用的NoSQL数据库。

- 不需要定义表结构
- 可以利用复杂的查询条件

1.2.3 面向列的数据库

Cassandra、Hbase、HyperTable属于这种类型。由于近年来数据量出现爆发性增长，这种类型的NoSQL数据库尤其引人注目。

面向行的数据库和面向列的数据库

普通的关系型数据库都是以行为单位来存储数据的，擅长进行以行为单位的读

入处理，比如特定条件数据的获取。因此，关系型数据库也被称为面向行的数据库。相反，面向列的数据库是以列为单位来存储数据的，擅长以列为单位读入数据。表 1-3所示为面向行的数据库和面向列的数据库比较。

表 1-3 面向行的数据库和面向列的数据库比较

数据类型	数据存储方式	优势
面向行的数据库	以行为单位	对少量行进行读取和更新
面向列的数据库	以列为单位	对大量行少数列进行读取，对所有行的特定列进行同时更新

高扩展性

面向列的数据库具有高扩展性，即使数据增加也不会降低相应的处理速度（特别是写入速度），所以它主要应用于需要处理大量数据的情况。另外，利用面向列的数据库的优势，把它作为批处理程序的存储器来对大量数据进行更新也是非常有用的。但由于面向列的数据库跟现行数据库存储的思维方式有很大不同，应用起来十分困难。

- **高扩展性（特别是写入处理）**
- **应用十分困难**

最近，像 Twitter 和 Facebook 这样需要对大量数据进行更新和查询的网络服务不断增加，面向列的数据库的优势对其中一些服务是非常有用的，但是由于这与本书所要介绍的内容关系不大，就不进行详细介绍了。

1.3 如何导入NoSQL数据库

Chapter 1

1.3.1 始终只是其中一种选择

并非对立而是互补的关系

关系型数据库和NoSQL数据库与其说是对立的（替代关系），倒不如说是互补的。笔者认为，与目前应用广泛的关系型数据库相对应，在有些情况下使用特定的NoSQL数据库，将会使处理更加简单。

这里并不是说“只使用NoSQL数据库”或者“只使用关系型数据库”，而是“通常情况下使用关系型数据库，在适合使用NoSQL的时候使用NoSQL数据库”，即让NoSQL数据库对关系型数据库的不足进行弥补。图1-11所示为在引入NoSQL数据库时的思维方法。

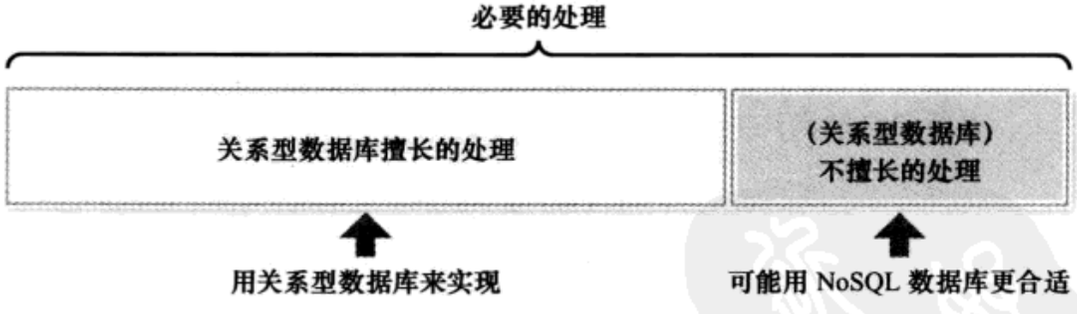


图1-11 在引入NoSQL数据库时的思维方法

量材适用

当然，如果用错了话，可能会发生使用NoSQL数据库反而比使用关系型数据库效果更差的情况。NoSQL数据库只是对关系型数据库不擅长的某些特定处理进行了优化，做到量材适用是非常重要的。

例如，若想获得“更高的处理速度”和“更恰当的数据存储”，那么NoSQL数据库是最佳选择。但一定不要在关系型数据库擅长的领域使用NoSQL数据库。

增加了数据存储的方式

原来一提到数据存储，就是关系型数据库，别无选择。现在 NoSQL 数据库给我们提供了另一种选择（当然要根据二者的优点和不足区别使用）。有些情况下，同样的处理若用 NoSQL 数据库来实现可以变得“更简单、更高速”。而且，NoSQL 数据库的种类有很多，它们都拥有各自不同的优势。

试着去参加学习研讨会吧

随着人们对 NoSQL 的认知度越来越高，最近出现了许多关于 NoSQL 的学习研讨会。我只是简单地搜索了一下，就发现有如此众多的学习研讨会。

如果对 NoSQL 很感兴趣请一定去参加这样的学习研讨会，肯定会得到很多新的收获。若能把自己感兴趣、了解的内容通过博客或者演讲的方式传播出去，那就更不起了。

1.3.2 在何种程度上信赖它

NoSQL数据库是一门新兴的技术，大家可能会觉得实际的操作经验还不多，还可能会碰到新的程序错误（bug）等，无法放心使用。

实际上，memcached已经相当地成熟了（错误和故障都已经被发现，且有明确的应对方法）。由于有丰富的事例和技术信息，所以不用担心会遇到上述问题。但是，在其他的NoSQL数据库的应用过程中遇到问题的可能性还是存在的。特别是实际应用的时候可以参考的经验、信息太少了。虽然NoSQL数据库能带来很多便利，但是在应用的时候也要考虑这些风险。

反过来说，如果不希望遇到此类问题，还是继续使用关系型数据库吧。它积累了很多的成熟经验，更让人放心。

NoSQL 数据库的种类和特征

前一章我们提到了 NoSQL 数据库的种类，并且对键值存储、面向文档的数据库、面向列的数据库进行了简单的介绍。本章将对 memcached、Tokyo Tyrant、Reds、MongoDB 这 4 种数据库做详细的说明。

另外，关于本章的各个导入说明和程序实例，都已在 CentOS 5.4 /Ruby 1.8.7 环境下进行了检验。

2.1 memcached（临时性键值存储）

Chapter 2

2.1.1 什么是 memcached

memcached是由Danga Interactive公司开发的开源软件，属于临时性键值存储的NoSQL数据库。（官方网站：http://memcached.org/）

2.1.2 为什么要使用 memcached

高速响应

大多数的Web应用程序通过关系型数据库来保存数据，并从中读取出必要的数据显示在用户的浏览器上。当数据量比较少时，应用程序可以很快读取出结果并显示出来。但当数据量增加，或者需要返回比较复杂的数据合计时，响应时间（用户访问Web应用程序得到返回结果的速度）就会变长，用户也只能被迫等待结果的返回。

根据Aberdeen Group公司2008年的调查，页面的显示速度每延迟1秒，网站访问量就会降低11%，从而导致营业额或者注册减少7%，顾客满意度下降16%[①]。类似的数据比比皆是，例如Google提到的“响应时间每延迟0.5秒，查询数将会减少20%”，Amazon提到的“响应时间延迟0.1秒，营业额下降1%”。可见响应速度是直接影响到用户行为和营业额的非常重要的指标。

作为高速缓存使用

那么，到底怎样才能获得高速的响应呢？当然如果是简单处理的话，利用关系型数据库的索引也能获得高速响应。虽然memcached会更快一些，但如果合理地使用索引，关系型数据库就足够快了。

但是，如果要对多个表的数据进行计算，情况又会怎样呢？若使用关系型数据

① 出自：http://www.gomez.com/wp-content/downloads/aberdeen-gomez-best-in-class.pdf。

库，我们需要从每个表中取出数据然后进行最后的组合处理，或者每次都要使用JOIN等处理。虽然我们可以通过事前用批处理制作数据来解决这个问题，但是这样又会增加需要管理的表，花费我们更多的精力。

“由于准备数据本身需要关系型数据库花费几十秒到几分钟的时间才能计算出来，因此实时计算就会显得比较慢。”这个时候memcached就可以大显身手了。因为memcached可以把从关系型数据库中读取出的数据保存到缓存中，所以即使是需要处理大量数据或者是访问非常集中的情况下，它也能非常快速地返回响应数据。这是因为memcached对于（第二次以后的）相同处理，只要它发现有数据保存在缓存里，就不用通过关系型数据库而直接进行处理。memcached存储的数据有可能丢失，但如果这些数据可以马上重新读取出来的话，那么即使因memcached停止而导致数据丢失，也不会有什么问题。

2.1.3 特征和用例

特征

memcached是通过大家都很熟悉的散列表（关联数组）来存储各种格式数据的键值存储，所有的数据都被保存在内存中。

memcached利用简单的文本协议来进行数据通信[①]，数据操作也只是类似于保存与键相对应的值这样的简单处理。因此，通过telnet连接memcached，就可以进行数据的保存和读取。但是，由于它利用的是文本协议，所以无法对构造体类型的数据进行操作，而只能对字符串类型的值进行操作。

当然，通过利用为各种语言准备的程序库来使用memcached的时候，值并不仅仅是字符串和数值，数组和散列表这样的构造体也可以进行保存和读取处理。但是，如前所述，因为使用了文本协议来进行通信，要完成这样的处理，就必须要把它们转换成序列化的字节（Byte）数组。通常情况下这些处理是在程序库内部进行的，因此不需要特别在意，但是需要在保存数据的时候进行序列化处理，读取数据的时候进行反序列化处理。例如使用Ruby的时候，保存数据需要执行Marshal.dump方法，读取数据需要执行Marshal.load方法。

序列化对开发语言的依赖

因为序列化依赖于开发语言，所以在某种开发语言环境下进行的序列化结果是

① 从1.4版本开始也采用二进制协议，并且保持了和文本协议的互换性。

无法在其他开发语言环境下使用的（由于无法进行适当的复原处理，数据往往会被破坏）。如果要进行这样的处理，就需要通过 JSON 或者 MessagePack 这样不存在语言依赖关系的格式化方法来进行明确的序列化和反序列化处理。图 2-1 所示为 memcached 的序列化处理和反序列化处理。

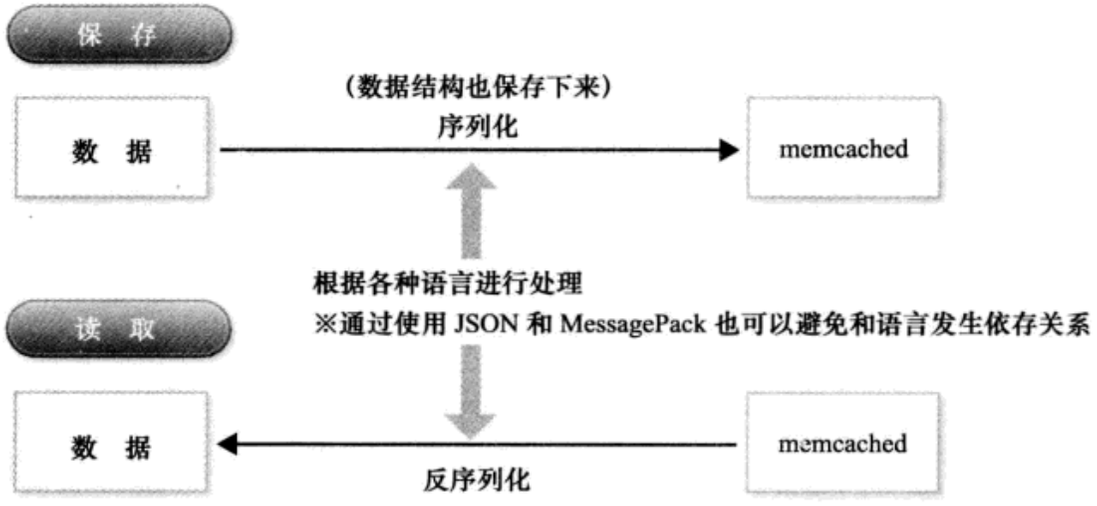


图 2-1 memcached 的序列化处理和反序列化处理

优势

说到 Memcached 的优势，要数其极其快速的处理速度。由于数据全都存储在内存中，没有硬盘的 IO 处理发生，所以它能以比关系型数据库高很多的速度进行处理（因为内存中对数据的访问速度是硬盘中的 10～100 万倍）。

另外一个容易被忽略的优势是它的简单易用性。由于它是通过键值这种散列表形式来操作数据的，所以只要用过散列表的人就能很容易地使用它。由于 memcached 停止的时候所有数据都会丢失，所以不论遇到什么奇怪的问题，只要重新启动 memcached 就可以恢复到初始状态。可能大家都有这样的经验：无论工具多么优秀，若是其难以使用，则用户也不会太多。

另外，由于现在很多的 Web 服务都在应用 memcached，不但拥有了成熟的技术，而且有很多成功经验被公开，这样我们在心理和技术上遇到的困难就会小得多。

随着数据量的增加，当 memcached 的内存无法保存所有数据的时候，可能就需要多台服务器来运行 memcached，实际上通过多台服务器来运行 memcached 也是非常简单的。在使用多台服务器来运行 memcached 的时候，会使用一致性散列（Consistent Hashing）算法[①]来分散数据。这个算法已经实际应用到 memcached 的客户端

① 从 1.4 版本开始也采用二进制协议，并且保持了和文本协议的互换性。

程序库中了。

一致性散列（Consistent Hashing）

如果只是使用多台服务器，就可以用键的散列值除以服务器台数，通过余数简单地决定哪台服务器处理哪条数据。例如使用 4 台 memcached 服务器，键的散列值是 516113735，516113735 除以 4 的余数是 3。由于余数只能是 0、1、2、3（每个数字都与 4 台服务器相对应），所以能够根据键来自动地决定使用这 4 台服务器中的哪一台。

但是，这种方法在服务器台数发生变化时就会遇到问题。如果增加 1 台服务器的时候会怎么样呢？这时服务器变成了 5 台，还是使用刚才的键，516113735 除以 5 的余数是 0，于是数据对应的服务器就跟刚才的不同了（从 3 号变成了 0 号）。这样一来会使处理数据的服务器发生很大变化，方法虽然简单但并不是很好。处理数据的服务器一旦发生变化，数据就无法正常地进行读取了（这里指的是缓存错误）。为了解决服务器台数增减带来的缓存错误问题，就出现了一致性散列（Consistent Hashing）这样的分散算法。

这个算法首先对各个服务器对应的散列值进行计算，把它们配置到一个圆周上。同时对各个数据对应的键的散列值进行计算，从键的散列值出发沿圆周向右，由距离该散列值最近的服务器来负责处理这条数据（数据的保存、读取都由这个服务器来执行）。下图可以帮助大家理解这个算法。图 2-2 所示为一致性散列（Consistent Hashing）的分配方式。

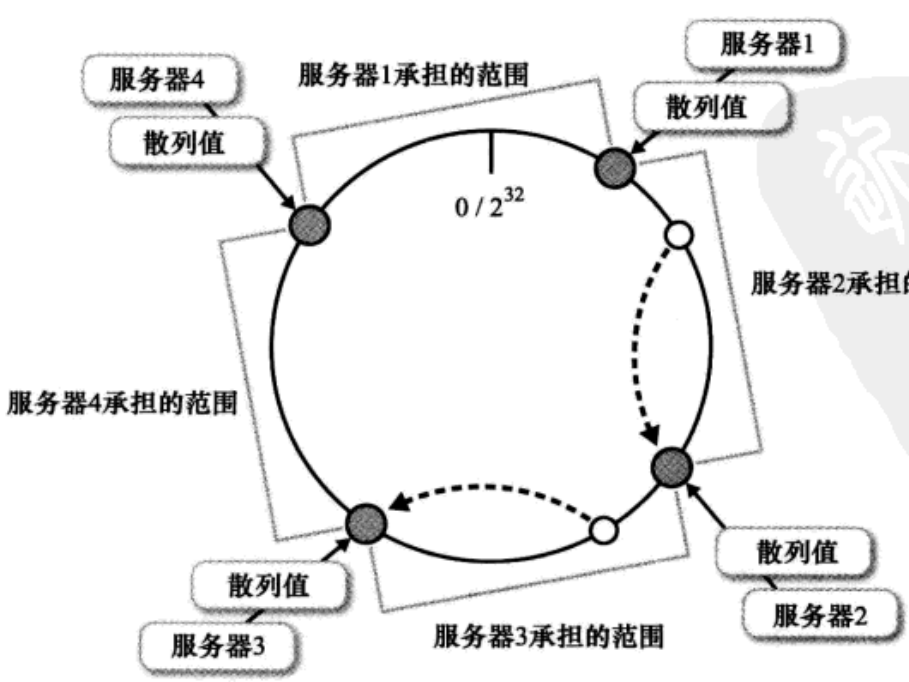


图 2-2 一致性散列（Consistent Hashing）的分配方式

这样的分配方式能够减小服务器增减带来的影响。

让我们来考虑一下具体的情况。例如刚才的例子中增加的那台5号服务器，假定它的散列值位于3号服务器和4号服务器之间，它对1号服务器、2号服务器和3号服务器负责处理的数据没有任何影响，这些服务器负责的数据还可以像以前一样继续由原来的服务器负责处理，唯一受到影响的是4号服务器负责的范围。

3号服务器到5号服务器之间的数据，本来应该由4号服务器负责处理，现在变成了由5号服务器来进行处理。负责处理这个范围内数据的服务器由4号变成了5号，就使得这个范围内的数据发生暂时性缓存错误的几率增大。但是，5号服务器和4号服务器之间的数据还是由4号服务器负责处理，这样只有一部分数据会受到影响。由此可见这个算法将会把服务器增减带来的缓存错误的影响减小到非常低的水平。图2-3所示为服务器增加时的变化。

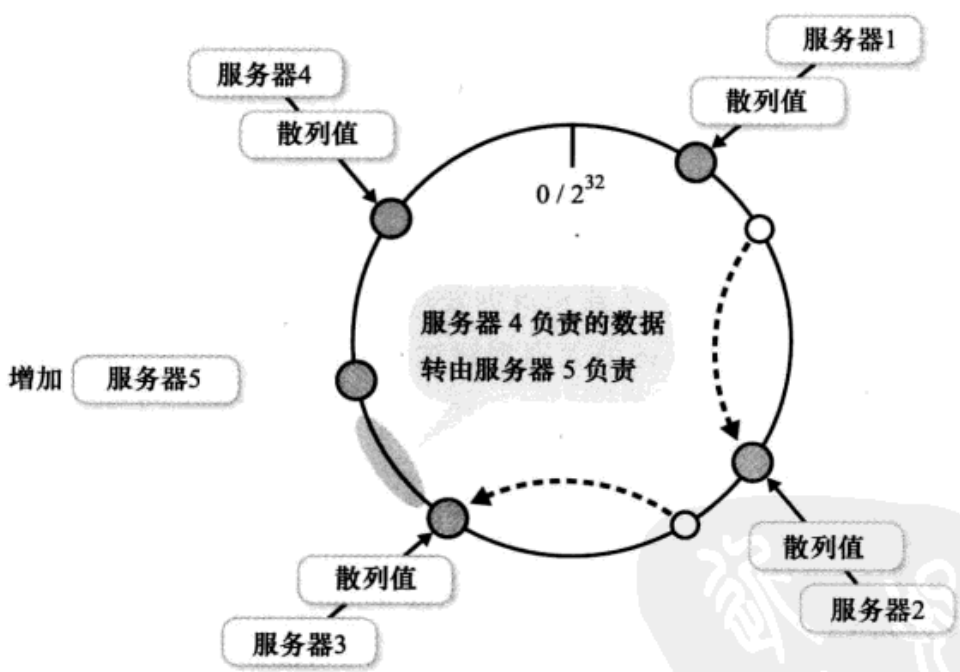


图2-3 服务器增加时的变化

不足

memcached有什么不足之处呢？其实就是大家非常在意的数据临时性（数据有可能会丢失）问题。

由于memcached把数据都保存在内存中，当memcached由于故障等原因停止的时候，所有的数据都会丢失。也正因为如此，用它来处理那些重要数据是比较危险的，绝对不要使用。最好的处理方式还是把原始数据保存在其他地方，而只是用memcached来处理原始数据的复制或者是通过原始数据计算出的结果。

另外，它还存在只能通过指定键来读取数据这样的局限性。例如，要取得键是“data _1”的数据，就不能用“包含 data”这样的模糊查询（LIKE）来进行，而必须通过完全匹配“data _1”来查询。从 SQL 的角度说就是只能通过下面这样的语句进行查询。

```
SELECT value FROM table WHERE key = XXX;
```

这些操作原本用关系型数据库就很容易实现，若要用 NoSQL 数据库就有必要在程序部分做相应的处理。

例如用日期作为键来保存数据的时候，关系型数据库可以通过 BETWEEN '2011-02-01'AND'2011-02-05'这样的条件来查询数据，但 memcached 却只能通过指定'2011-02-01'、'2011-02-02'、'2011-02-03、'2011-02-04'、'2011-02-05'这样具体的键来查询数据。图 2-4 所示为使用 memcached 读取数据时需要注意的地方。

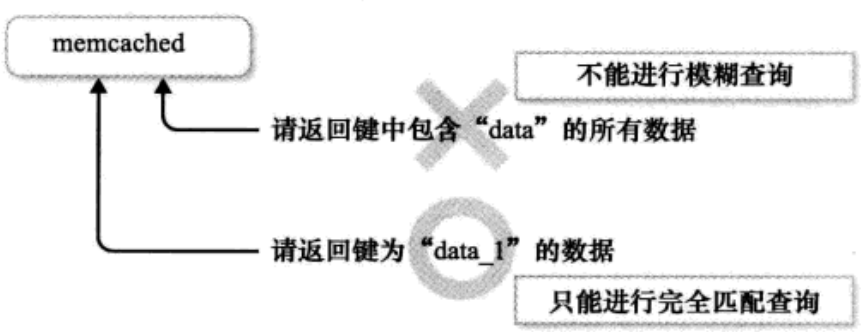


图 2-4 使用 memcached 读取数据时需要注意的地方

memcached 停止时的保障措施

如果数据库的访问量比较大，就需要提前做好准备，以便应对在 memcached 停止时发生的负载问题。也就是说在 memcached 停止、数据丢失的时候，之前一直没有通过关系型数据库而直接由 memcached 处理的大量请求，将会同时转向关系型数据库，这会大大增加关系型数据库的负荷。结果就会导致响应延迟，甚至会导致服务器崩溃。

使用后面将要提到的 repcached 可以提高 memcached 的可用性，使用多台服务器可以让每台服务器上的 memcached 处理较少的数据（即使有一台停止的时候也不会造成很大的影响），这些都是值得我们花功夫去研究的。

应用实例

以下介绍几个 memcached 的应用实例。这些例子是从官方网站上选取出来的。实际上除此之外还有非常多的 Web 服务在使用 memcached，即使是那些访问量很大的 Web 服务，memcached 亦能轻松应对。

- **Wikipedia**
- **Flickr**
- **Twitter**
- **Youtube**
- **Mixi**

由于 memcached 使用简单，并且可以改善响应延迟的问题，最近国内外的 Web 服务基本上都引入了 memcached。

用例

memcached 主要用于缓存从关系型数据库中取出的数据。[①] 图 2-5 所示为 memcached 的基本使用方法。

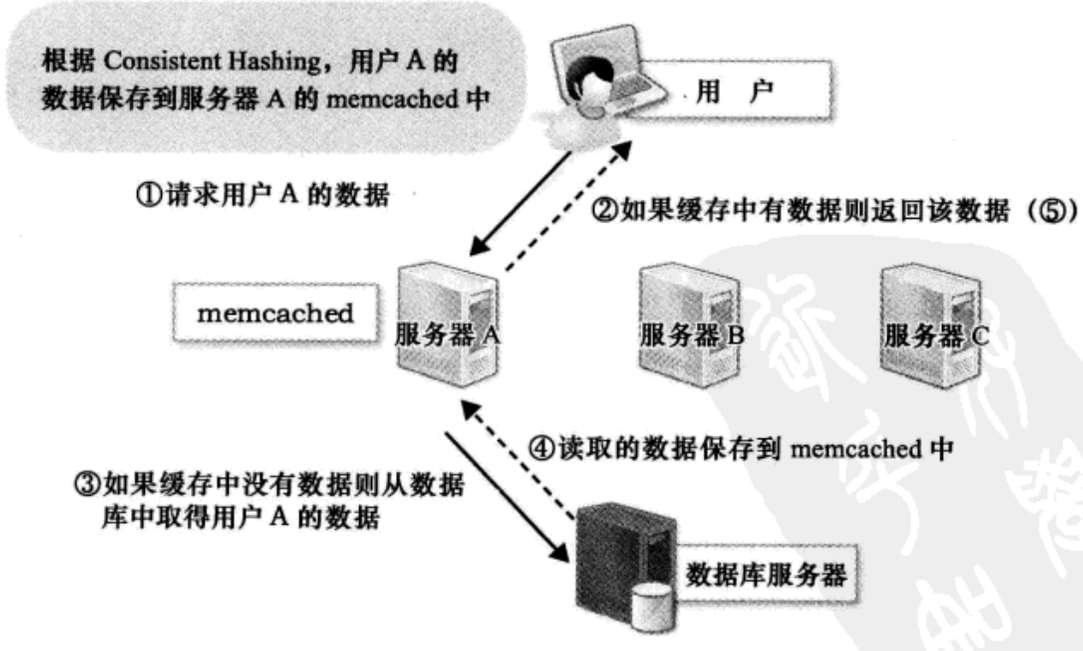


图 2-5 memcached 的基本使用方法

① 用户的 session 信息亦可保存在缓存中，但请注意倘若数据丢失，就会对登录状态产生的影响。

具体来说它在以下两种情况下的作用不可小觑：

- **有些网页需要执行一些耗费时间的 SQL 文，响应很慢**
- **有些网页访问频繁，负载很大**

当然也可以通过事前的批处理程序来创建所需要的数据，在一定程度上缓解这些问题。但是执行批处理程序不得不增加相应的临时表，这样就增加了运用和管理上的开销。如果数据需要保存，当然最好是先用批处理进行创建，然后再保存到关系型数据库中。但是，如果只是临时使用，即使数据丢失了也能在几分钟内完成恢复，用 memcached 不就足够了吗？

第 3 章将会介绍排名数据和利用外部 API 读取数据的实例。

2.1.4 安装步骤

安装

memcached 的安装非常简单。如果使用 CentOS（Community ENTerprise Operating System），可以通过 yum（软件包管理器）一次完成。但是由于 CentOS 的标准存储库中不包含 memcached[①]，所以在安装之前必须要添加 yum 的存储库。请创建/etc/yum.repos.d/dag.repo 文件，然后写入下列内容。

/etc/yum.repos.d/dag.repo

```
[dag]
name = Dag RPM Repository for Redhat EL5
baseurl = http://apt.sw.be/redhat/el$releasever/en/$basearch/dag
gpgcheck = 1
enabled = 0
gpgkey = http://dag.wieers.com/packages/RPM-GPG-KEY.dag.txt
```

这样就可以使用 dag 这个新的存储库了。由于它不是 CentOS 自带的官方存储库，所以一般都会设置成不可用状态（enabled＝0）。通过在 yum 的命令中加入“–enablerepo＝dag”这个参数，可以使得在查询和安装时把这个存储库也做为处理对象。例如 memcached 就可以通过下面这样的命令来进行安装。

① 参考 URL：http://www.atmarkit.co.jp/flinux/rensai/linuxtips/794uofrepo.html

```
sudo yum install memcached --enablerepo = dag
```

本书编写时的版本是1.4.5。当然也可以通过下载代码的方式进行安装，请选择自己习惯的安装方式。

启动

memcached的启动方法如下。

```
sudo /etc/init.d/memcached start
```

让我们来确认一下已经启动的进程，如下的进程应该已经启动。

```
$ ps aux |  grep memcached
memcached -d -p 11211 -u nobody -c 1024 -m 64
```

下面解释一下各个参数的含义，表2-1所示为memcached启动时的参数。

表2-1 memcached启动时的参数

参数	说明	默认值
-d	作为后台程序（Demon）在后台启动	-
-p	待机端口号	11211
-u	用户名（只在使用root运行的时候）	nobody
-c	最大连接数	1024
-m	最大使用内存量（单位是MB）	64

当保存的数据量超过-m参数设定的时候，它会以LUR（Least Recently Used—最近最少使用算法）的顺序来丢弃数据。这是一种丢弃最近最少使用数据的算法，考虑到memcached是作为缓存来使用的，这是最效的方法。

若想要增加内存的存储量，或者希望改变待机端口号，可以通过改变/etc/init.d/memcached的CACHESIAZE和PORT项目来实现，或者像下面这样直接通过设定参数来启动。

```
memcached -d -p 11212 -u nobody -c 1024 -m 128
```

2.1.5 动作确认

通过 telnet 确认

memcached 默认的待机端口号是 11211（所谓待机，指的是开启这个端口，等待对这个端口的访问），可以通过 telnet 来连接 memcached 的待机端口。赶快来试试看吧。表 2-2 所示为数据保存时的参数。


```
# 连接 localhost 的11211号端口
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain(127.0.0.1).
Escape character is '^]'.

# 保存数据
# set <key> <flag> <expires> <byte> [换行]
# <value>
set foo 0 10 3
bar
STORED

# 数据读取
get foo
VALUE foo 0 3
bar
END
```

表 2-2 数据保存时的参数

名称	说明	注意事项
flag	用来指定是否压缩数据 0：不压缩；1：压缩	-
expires	使用 UNIX 的时间戳或者从当前开始计算的秒数来指定数据保存的时间	从当前开始计算的秒数不能超过 30 天（60 * 60 * 24 * 30=2592000 秒） 设置成 0 的时候表示有效期无限（半永久的保存）
byte	用来指定作为值保存的数据的字节数	-

通过 telnet 来连接本地启动的 memcached 的 11211 号端口，可以确认数据的保存和读取状况。数据的保存在这里可能理解起来有点困难，其实最开始输入的并不

是数据，而是数据的字节数，换行之后输入的才是实际的数据。

另外 memcached 还可以使用 incr、delete、append、flush _all 等各种各样的命令。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain(127.0.0.1).
Escape character is '^]'.

set counter 0 0 1
1
STORED

# 数值的加法(减法的标识是 decr)
# incr <key> <num>
incr counter 1
2
incr counter 2
4

# 数据的删除
delete counter
DELETED

# 确认删除的数据
get counter
END

set test 0 0 4
test
STORED

# 在数据末尾添加内容(在数据开头添加内容的时候使用 prepend 方法)
# append <key> <flag> <expires> <byte> [换行]
# <value>
append test 0 0 3
add
STORED

get test
VALUE test 0 7
testadd
END

# 删除 memcached 上的所有数据
flush_all
OK
```

在 Ruby 程序上使用（1 台）

那么，这次让我们试试在 Ruby 的程序上使用 memcached。我们需要使用 gem 的程序库 memcache-client。安装方法如下，本书编写时的版本是 1.8.5。

```
gem install memcache-client
```

首先让我们试着进行简单的数据保存和读取。

```
$KCODE = 'u'

require "rubygems"
require "memcache"

server = ['localhost:11211'] # 指定要使用的 memcached
option = {}

cache = MemCache.new(server, option)

# 保存数据
cache['key1'] = 123                          # 数字
cache['key2'] = "ABCDE"                      # 字符串
cache['key3'] = %w(hoge fuga)                # 数组
cache['key4'] = {:foo => 1, :bar => "a"}     # 散列表

# 读取数据
p cache['key1'] # 123
p cache['key2'] # "ABCDE"
p cache['key3'] # ["hoge", "fuga"]
p cache['key4'] # {:bar => "a", :foo => 1}
```

感觉怎么样？即使不了解 Ruby 的人也能轻松操作，更不用说那些熟悉 Ruby 的人了。对于 cache 实例来说，可以通过适当的键，对数字、字符串、数组、散列表等各种类型的值进行保存和读取操作。经确认，数字、字符串、数组、散列表等都能被正确地读取出来。

另外，还可以通过散列表的形式来给构造函数传递参数。可以设定的参数如表2-3所示。

表2-3 可以传递给memcache-client构造函数的参数

名称	设定内容	初始值
namespace	保存数据用的命名空间，设置在键前面	nil
readonly	读取专用，写入的时候会发生MemCacheError	false
multithread	线程安全	true
failover	服务器崩溃时能否访问其他服务器	true
timeout	过期设定，设定成nil的时候不会过期	0.5秒
logger	指定输出日志用的logger	nil
no_reply	写入数据时是否接受应答，设置成不接受应答（false）的时候能够提升性能。 * memcached1.2.5之后的版本可以使用	false
check_size	值超过1MB的时候会发生MemCacheError	true
autofix_keys	键超过250个文字或者包含空格的时候，会用Digest::SHA1.hexdigest（key）来代替键	false

使用memcached和使用散列表非常相似，实际对比一下就一目了然了，是不是感觉很像呢？

```
# 使用memcached
cache = MemCache.new(['localhost:11211'])
cache['key'] = 'value'
p cache['key'] # value

# 使用散列表
hash = {}
hash['key'] = 'value'
p hash['key'] # value
```

memcache-client为数据的保存和读取分别提供了两种方法：数据保存用到的set方法和[]=方法，数据读取用到的get方法和[]方法。（其实[]=方法和[]方法分别是set方法和get方法的别名）。

一般情况下可以根据个人习惯来选择使用方法，不过请注意保存数据用到的set方法和[]=方法还是有一些区别的。数据保存的时候，使用set方法可以自由设定expires（失效时间），但是[]=方法不能设定expires（失效时间），而是使用0（也就

是半永久性的数据保存）作为默认设定。这些操作会根据程序库的不同而发生变化，建议大家通过各个程序库提供的文档和实际使用情况来判定。

让我们来看一下使用 set 方法保存数据时如何设定 expires（失效时间）。例如我们设定 expires（失效时间）的值为 10 秒，确认一下是不是 10 秒过后数据就无法取出了。

```
require "rubygems"
require "memcache"

cache = MemCache.new(['localhost:11211'])

cache.set('key', 'value', 10) # expires 设定为10秒
p cache['key'] # 'value'

sleep 10

p cache['key'] # nil(10秒后已经过期)
```


在 Ruby 程序上使用（多台）

目前为止我们已经尝试了使用 1 台服务器运行 memcached 的情况，接下来让我们试试看使用多台服务器的情况。

例如我们使用 3 台 memcached 服务器，通过一致性散列（Consistent Hashing）算法把数据分配到 3 台服务器上。说到使用多台服务器的好处，首先单纯从 memcached 的内存分配来看，每台服务器是 64MB（默认情况），合计就是 192MB，这是单个服务器的 3 倍。这样就可以把 3 倍于 1 台服务器的数据量在 memcached 的内存上进行缓存处理。再者，如果其中 1 台服务器停止的时候，虽然有影响，但也只限于小范围的数据。如果只有 1 台服务器，当 memcached 停止的时候，memcached 上所有的数据都会丢失。利用 3 台服务器来分散数据的话，最多只会丢失 1/3 的数据，其他 2/3 还是可以用的。而这 1/3 的数据也还可以通过一致性散列（Consistent Hashing）算法重新分散到剩余的服务器上。

接下来就让我们实际尝试一下。首先，分别设定待机端口号（这次我们分别设置为 11211、11212、11213）启动 3 台 memcached 服务器。

```
memcached -d -p 11211 -u nobody -c 1024 -m 64
memcached -d -p 11212 -u nobody -c 1024 -m 64
memcached -d -p 11213 -u nobody -c 1024 -m 64
```

然后程序部分只需要定义一个如下的数组，设定传递给构造函数的参数。使用多台服务器运行 memcached 所需要的操作基本上就是这些。但是，大家还需要注意：所使用的程序库必须要支持一致性散列算法。这次我们使用的 memcache-client，与一致性散列对应的版本是 1.6.0[①]。这样我们就可以在增加服务器数量的同时，通过一致性散列算法来抑制缓存错误的发生，不断地扩展缓存服务器的规模。


```
$KCODE = 'u'

require "rubygems"
require "memcache"
require "logger"

server = [
  'localhost:11211',
  'localhost:11212',
  'localhost:11213'
]

option = {
  :logger => Logger.new(STDOUT)
}

cache = MemCache.new(server, option)

# 数据的保存
cache['key1'] = 123                          # 数字
cache['key2'] = "ABCDE"                      # 字符串
cache['key3'] = %w(hoge fuga)                # 数组
cache['key4'] = {:foo => 1, :bar => "a"}     # 散列表

# 数据的读取
p cache['key1'] # 123
p cache['key2'] # "ABCDE"
p cache['key3'] # ["hoge", "fuga"]
p cache['key4'] # {:bar => "a", :foo => 1}
```

为了确认具体的操作内容，我们在构造函数中指定了 logger 参数。指定 logger 可以使 memcached 把所有的处理都详细地记录在日志中。在开发过程中，如果希望掌握 memcached 的具体操作，充分利用 logger 是个很好的途径。这次我们输出了如下的日志。

① https://github.com/mperham/memcache-client/blob/master/History.rdoc

```
# 作为对象的 memcached 的信息
 INFO --:memcache-client 1.8.5 ["localhost:11211", "localhost:11212",
"localhost:11213"]
DEBUG --:Servers now:[<MemCache::Server:localhost:11211 [1]
(NOT CONNECTED)> , <MemCache::Server:localhost:11212 [1] (NOT CONNECTED)> ,
<MemCache::Server:localhost:11213 [1] (NOT CONNECTED)> ]

# 数据的保存
DEBUG --:set key1 to <MemCache::Server:localhost:11212 [1] (CONNECTED)> :5
DEBUG --:set key2 to <MemCache::Server:localhost:11211 [1] (CONNECTED)> :19
DEBUG --:set key3 to <MemCache::Server:localhost:11211 [1] (CONNECTED)> :16
DEBUG --:set key4 to <MemCache::Server:localhost:11213 [1] (CONNECTED)> :19

# 数据的读取
DEBUG --:get key1 from <MemCache::Server:localhost:11212 [1] (CONNECTED)>
DEBUG --:get key2 from <MemCache::Server:localhost:11211 [1] (CONNECTED)>
DEBUG --:get key3 from <MemCache::Server:localhost:11211 [1] (CONNECTED)>
DEBUG --:get key4 from <MemCache::Server:localhost:11213 [1] (CONNECTED)>
```

这样我们就利用 3 台服务器成功启动了 memcached，也清楚地知道了数据通过一致性散列算法被分散到了不同的服务器上。

那么，如果 11211 号端口的 memcached 停止，服务器从 3 台变为 2 台的时候会发生什么样的情况呢？在刚才的程序中，如果不使用 11211 号端口的 memcached，结果如下所示。

```
# 作为对象的 memcached 的信息
 INFO --:memcache-client 1.8.5 ["localhost:11212", "localhost:11213"]
DEBUG --:Servers now:[<MemCache::Server:localhost:11212 [1] (NOT
CONNECTED)> , <MemCache::Server:localhost:11213 [1] (NOT CONNECTED)> ]

# 数据的保存
DEBUG --:set key1 to <MemCache::Server:localhost:11212 [1] (CONNECTED)> :5
DEBUG --:set key2 to <MemCache::Server:localhost:11212 [1] (CONNECTED)> :19
DEBUG --:set key3 to <MemCache::Server:localhost:11212 [1] (CONNECTED)> :16
DEBUG --:set key4 to <MemCache::Server:localhost:11213 [1] (CONNECTED)> :19

# 数据的读取
DEBUG --:get key1 from <MemCache::Server:localhost:11212 [1] (CONNECTED)>
DEBUG --:get key2 from <MemCache::Server:localhost:11212 [1] (CONNECTED)>
DEBUG --:get key3 from <MemCache::Server:localhost:11212 [1] (CONNECTED)>
DEBUG --:get key4 from <MemCache::Server:localhost:11213 [1] (CONNECTED)>
```

刚才通过11211号端口的memcached操作的键2和键3的数据，已经被分配到11212号端口的memcached上了，大家都明白了吧？由于memcached通过一致性散列（Consistent Hashing）算法来分散数据，所以数据会根据服务器的台数进行自动分散。

设定复制

单台的memcached是无法进行复制的。若要进行复制，则需要利用后面将要提到的repcached[①]等工具。之后我们将会使用到repcached，到时会进行详细介绍。

2.1.6 各种开发语言需要用到的程序库

连接memcached的客户端程序库，包含了Ruby、Perl等各种开发语言适用的程序库，下面列举了其中的一部分[②]，大家可以放心使用。你亦可使用JSON和MessagePack这样不受开发语言限制的方法将其进行序列化，使不同开发语言之间也能进行数据交互。表2-4所示为memcached的各种语言用的库（精选）。

表2-4 memcached的各种语言用的库（精选）

开发语言	程序库	开发语言	程序库
C/C++	libmemcached	Python	python-memcached
	libmemcache		MemcachePool
	apr_memcache		python-libmemcached
	memcacheclient		Django
	libketama		twisted. protocols. memcache
PHP	PECL/memcached	Ruby	cache_fu
	PECL/memcache		memcache-client
	php-libmemcached		Ruby-MemCache
Java	spymemcached		fauna
	Java memcached client		caffeine
	javamemcachedclient	Pert	Cache::Memcached
	memcache-client-forjava		Cache::Memcached::Fast
	xmemcached		Cache::Memcached::linbmemcached
	simple-spring-memcached		perl-libmemcached

① repcached是日本人开发的，可以实现memcached的复制功能。——译者注

② 详见：http://code.google.com/p/memcached/wiki/Clients。

2.1.7 相关工具

repcached

大家在使用 memcached 的时候总会担心数据丢失的问题。例如使用3台 memcached 服务器，如果其中一台停止服务，1/3的数据就会丢失。虽然说可以进行恢复，但是那段时间内对关系型数据库的访问就会比较集中，使其负荷加大，也很让人担心。memcached 能不能像关系型数据库那样通过主从复制技术来提高可用性呢？

答案是肯定的。这个时候可以使用 repcached[①] 来实现 memcached 的复制功能。如果只是想提高可用性的话，我们还有其他一些方法，比如利用 VIP（虚拟 IP 地址）技术，把多台 memcached 虚拟成一台假想的单一服务器，但相对来说使用 repcached 更容易一些。

使用 repcached 需要另外安装一些程序库。

```
sudo yum install libevent-devel --enablerepo = dag
```

repcached 可以通过源代码直接安装，本书编写时的版本是 2.2，它对应的 memcached 版本是 1.2.8，有一些偏低，请大家注意。

另外，关于配置（configure）时的参数，第一个选项是使复制有效的选项，第二个选项指明了安装命令的名字是 repcached，而不是 memcached。这样设置就可以在保留 memcached 的基础上安装 repcached 了。这里推荐大家选择 repcached 和 memcached 共存的方式。

```
wget http://downloads.sourceforge.net/repcached/memcached-1.2.8-repcached-
2.2.tar.gz
tar zxvf memcached-1.2.8-repcached-2.2.tar.gz
cd memcached-1.2.8-repcached-2.2
./configure --enable-replication --program-transform-name = s/memcached/
repcached/make
sudo make install
```

那么，就让我们来实际试用一下吧。首先使用 11211 号端口启动作为主服务器

① http://lab.klab.org/wiki/Repcached

的repcached。另外，为了更容易地理解这次的操作，启动的时候附带了-v这个参数。

```
$ repcached -p 11211 -v
replication:listen
replication:accept
```


然后，使用跟刚才相异的端口号（如果使用不同的服务器的话，也可以使用相同的端口号）来启动作为从服务器的repcached。启动的时候通过-x参数来指定对应的主服务器名或者是IP地址。另外，-x参数也可以用来指定复制的主服务器端口号，默认的端口号是11211。

```
$ repcached -p 11212 -x localhost -v
replication:connect (peer = 127.0.0.1:11212)
replication:marugoto copying
replication:start
```

我们只需要通过日志就能确认复制已经设置成功了。图2-6所示为repcached的复制。

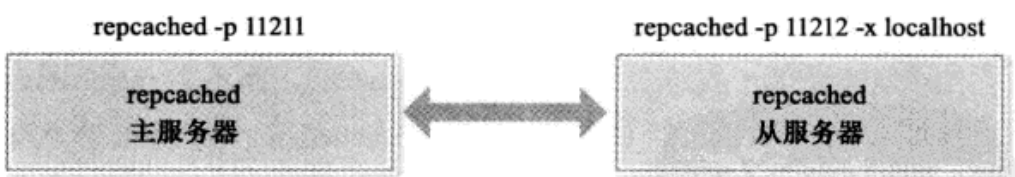


图2-6 repcached的复制

准备工作完成了，接下来让我们确认一下复制是否已经执行了。首先，通过telnet连接11211号端口的repcached（master），尝试保存数据。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set hello 0 0 5
world
STORED
```

```
get hello
VALUE hello 0 5
world
END
```

然后通过 telnet 连接 11212 号端口的 repcached（slave），确认一下刚才通过 11211 号端口的 repcached 保存的数据是否已经被复制了。


```
$ telnet localhost 11212
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

get hello
VALUE hello 0 5
world
END
```

没问题，已经可以从 repcached（slave）中读取数据了。只需要这样简单的配置，就可以在使用 memcached 的同时，完成数据复制，这样大家就可以放心地使用了吧。

顺便说一下，大家也可以对 11212 号端口的 repcached（slave）进行写入处理，因为 repcached 的复制是由二元主机构成的。

对 11212 号端口的 repcached（slave）进行写入处理，这些数据也可以顺利从 11211 号端口的 repcached（master）读取出来，据此你可以确认复制没有任何问题。

```
$ telnet localhost 11212
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set hello2 0 0 4
ruby
STORED

get hello2
VALUE hello2 0 4
ruby
END
```

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

get hello2
VALUE hello2 0 4
ruby
END
```


掌握 memcached 的使用情况

虽然 memcached 很方便，但是并没有简单的方法能确认当前保存数据、内存使用量等信息。例如假设在启动 memcached 后，通过 memcache-client 保存了以下 20 条数据。

```
require 'rubygems'
require 'memcache'

cache = MemCache.new(['localhost:11211'])

(1..10).each do | i|
  cache["key_# {i}"] = "value_# {i}"
end

(11..20).each do | i|
  cache["key_# {i}"] = "value_# {('a'..'z').to_a.join('')}_# {i}"
end
```

要确认当前 memcached 保存的数据，需要通过 telnet 连接，然后执行如下的命令。若使用 memcached 附带的 memcache-tool 脚本（script），也能在一定程度上进行确认。但是这些方法都比较繁琐，我们还是希望能更轻松地取得我们想要的数据。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'
```

```
stats items
STAT items:1:number 10
STAT items:1:age 87
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 0
STAT items:2:number 10
STAT items:2:age 87
STAT items:2:evicted 0
STAT items:2:evicted_nonzero 0
STAT items:2:evicted_time 0
STAT items:2:outofmemory 0
STAT items:2:tailrepairs 0
STAT items:2:reclaimed 0
END

# stats cachedump <slab_id> <num>
stats cachedump 1 10
ITEM key_10 [12 b; 1299367762 s]
ITEM key_9 [11 b; 1299367762 s]
ITEM key_8 [11 b; 1299367762 s]
ITEM key_7 [11 b; 1299367762 s]
ITEM key_6 [11 b; 1299367762 s]
ITEM key_5 [11 b; 1299367762 s]
ITEM key_4 [11 b; 1299367762 s]
ITEM key_3 [11 b; 1299367762 s]
ITEM key_2 [11 b; 1299367762 s]
ITEM key_1 [11 b; 1299367762 s]
END

stats cachedump 2 10
ITEM key_20 [39 b; 1299367762 s]
ITEM key_19 [39 b; 1299367762 s]
ITEM key_18 [39 b; 1299367762 s]
ITEM key_17 [39 b; 1299367762 s]
ITEM key_16 [39 b; 1299367762 s]
ITEM key_15 [39 b; 1299367762 s]
ITEM key_14 [39 b; 1299367762 s]
ITEM key_13 [39 b; 1299367762 s]
ITEM key_12 [39 b; 1299367762 s]
ITEM key_11 [39 b; 1299367762 s]
END
```

这里，笔者使用了自己做成的 memdump. rb 脚本（script）。

memdump.rb

```
require 'socket'

if ARGV.size != 2
  puts "Usage:ruby memdump.rb localhost 11211"
  exit
end

socket = TCPSocket.new(*ARGV)

puts "---"

items_h = {}
socket.puts "stats items"
while s = socket.gets
  break if s =~ /^END/
  if s =~ /^STAT items:(\d+):number (\d+)/
    items_h[$1] = $2;
  end
end

mem = 0
items_h.keys.each do |key|
  socket.puts "stats cachedump # {key} # {items_h[key]}"

  list = []
  while s = socket.gets
    break if s =~ /^END/
  list << $1 if s =~ /^ITEM (\w+) .+ /
end

list.sort{|a,b|a.split(/_/)[-1].to_i <=> b.split(/_/)[-1].to_i}.each do |key|
  socket.puts "get # {key}"
  while s = socket.gets
    next if s =~ /^VALUE/
    break if s =~ /^END/
    puts "# {key}:# {Marshal.load(s)}"
  end
end

socket.puts "stats slabs"
while s = socket.gets
    break if s =~ /^END/
    mem += $1.to_i if s =~ /^STAT \ d+ :mem_requested (\d+)/
  end
end

puts "--- "
puts "内存使用量:# {mem}byte"

socket.close
```

运行这个脚本程序时，通过如下的格式指定主机名和端口号，就可以得到该memcached服务器上保存的键值列表，并获得内存使用量。

```
ruby memdump.rb <hostname> <port>
```

由于这段脚本程序把memcache-client保存的数据作为对象（也就是说，数据保存的时候被序列化了），所以读取数据的时候需要通过Marshal.dump进行反序列化。

这样一来，就可以很方便地知道现在保存的数据和内存的使用量，也可以更容易地把握memcached的使用情况。


```
$ ruby memdump.rb localhost 11211
---


key_1 : value_1
key_2 : value_2
key_3 : value_3
key_4 : value_4
key_5 : value_5
key_6 : value_6
key_7 : value_7
key_8 : value_8
key_9 : value_9
key_10 : value_10
key_11 : value_abcdefghijklmnopqrstuvwxyz_11
key_12 : value_abcdefghijklmnopqrstuvwxyz_12
key_13 : value_abcdefghijklmnopqrstuvwxyz_13
key_14 : value_abcdefghijklmnopqrstuvwxyz_14
key_15 : value_abcdefghijklmnopqrstuvwxyz_15
key_16 : value_abcdefghijklmnopqrstuvwxyz_16
key_17 : value_abcdefghijklmnopqrstuvwxyz_17
key_18 : value_abcdefghijklmnopqrstuvwxyz_18
key_19 : value_abcdefghijklmnopqrstuvwxyz_19
key_20 : value_abcdefghijklmnopqrstuvwxyz_20
---
内存使用量:3224byte
```

2.2 Tokyo Tyrant（永久性键值存储）

Chapter 2

2.2.1 什么是 Tokyo Tyrant

Tokyo Tyrant是平林干雄（Mikio Hirabayashi）开发的开源软件，它属于NoSQL分类中的永久性键值存储。

官方网站：http://1978th.net/tokyotyrant

2.2.2 为什么要使用 Tokyo Tyrant

数据丢失的困扰

在前一节中，我们知道了使用memcached可以高速地返回Web应用响应。但memcached的使用毕竟只能在“允许数据丢失”和“即使数据丢失也可以进行复原”这样的前提下进行，局限性非常明显。我们当然希望可以有一种既能解决数据丢失问题，又能高速返回响应的办法，这应如何实现呢？

这里就需要用到Tokyo Tyrant。与memcached把数据保存在内存中不同，Tokyo Tyrant把数据保存在硬盘中。这样就保证了数据的持久性，不用再担心数据丢失的问题了[①]。另外，由于它是由C语言编写的，因此可以对硬盘访问进行优化，从而实现数据的高速处理。

2.2.3 特征和用例

特征

要理解Tokyo Tyrant，首先要先了解平林干雄开发的Tokyo Cabinet。

① 当然，由于硬盘损坏造成数据丢失的可能性还是存在的。

Tokyo Cabinet 是一款具有键值存储功能的数据库，它可以对数据进行保存和读取。但是 Tokyo Cabinet 不能通过网络使用，只能在本地间进行通讯。如果把它作为 NoSQL 数据库来使用的话，存在着不支持网络等很多不便之处。因为很多情况下，当我们需要增加应用服务器时，不只需要增加本地应用服务器，还要通过远程连接对特定的 Tokyo Cabinet/Tokyo Tyrant 进行访问。

实际上，这里要介绍的 Tokyo Tyrant 就是为了让 Tokyo Cabinet 能够支持网络（也就是可以通过其他服务器访问）而进行的封装[①]。Cabinet（内阁）是 Tyrant（专制君主）的傀儡，由此得名。[②] 说到底主体还是 Tokyo Cabinet，Tokyo Tyrant 只是对它的封装而已。图 2-7 所示为 Tokyo Cabinet 和 Tokyo Tyrant 的关系。

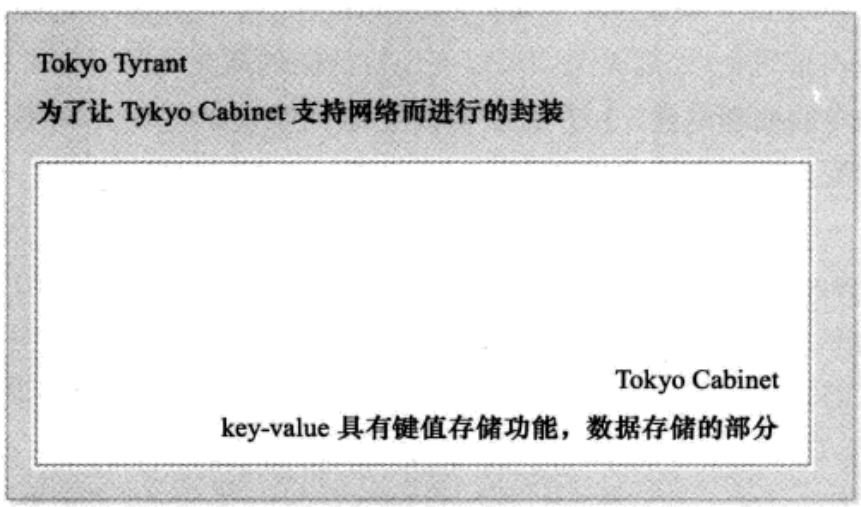


图 2-7 Tokyo Cabinet 和 Tokyo Tyrant 的关系

Tokyo Tyrant 和 memcached 一样，通过键值这样的散列表结构保存数据。但是，数据的保存地点却不一样，memcached 是把数据保存在内存中，而 Tokyo Tyrant 则是把数据保存在硬盘上。[③]

另外，Tokyo Tyrant 还引入了数据类型的概念。可以根据选择的数据类型，在缓存数据库（on memory database）、散列数据库（hash database）、B-tree 数据库（B-tree database）和表格数据库（table database）等数据保存方式间进行转换。例如表结构数据库就可以使用跟关系型数据库相似的方法进行操作（可以使用复杂的查询条件来抽取数据等）。

① Tokyo Tyrant 是为 Tokyo Cabinet 编写的网络接口，它支持 memcache 协议，也可以通过 HTTP 操作。——译者注

② http://alpha.mixi.co.jp/blog?p=31

③ 根据数据类型的不同，也存在和 memcached 同样的把数据保存在内存中的情况。

Tokyo Tyrant不但可以使用自带的协议进行通信，而且还为HTTP协议和memcached兼容协议提供了支持。但是，Tokyo Tyrant并不支持expires（失效时间）。由于使用简便，并且可以顺利地通过memcached兼容协议，获得与memcached一样的使用体验，确实是一个不错的选择。如果使用memcached兼容协议还可以非常简单地把数据从memcached中转移出来，关于这一点之后还会进行详细介绍。

优势

由于数据保存在硬盘上，Tokyo Tyrant的最大优势就是在它停止的时候数据也不会丢失。当然，关系型数据库也不存在数据丢失的问题，所以从某种程度上来说，也算不上什么优势。Tokyo Tyrant的另一个优势是：它在保存和读取数据的时候，与硬盘的IO处理无关，可以实现对数据的高速访问。它可以获得比关系型数据库快很多的处理速度。用户在获得高速返回响应的同时，又不必担心数据丢失的烦恼，真的是非常方便。

关于数据的读取方式，memcached只能通过与键完全一致的条件进行查询，而Tokyo Cabinet/Tokyo Tyrant存在不同的数据库类型，可以进行范围查询（B-tree数据库）或者像关系型数据库那样进行复杂条件的查询（表格数据库）。同一个产品，能够根据用途的不同而在数据库类型间进行切换实在是方便极了。表2-5所示为Tokyo Cabinet/Tokyo Tyrant的数据类型和特征。

表2-5 Tokyo Cabinet/Tokyo Tyrant的数据类型和特征

数据类型	数据的保存方式	特征
缓存数据库	以键值的形式把数据保存在内存中	和memcached相同
散列数据库	以键值的形式把数据保存在硬盘上	擅长随机存取
B-tree数据库	一个键对应多个值的保存和读取。数据保存在硬盘上	擅长范围查询和连续访问
表格数据库	像关系型数据库那样通过多个字段把数据保存在硬盘上	可以进行复杂条件的查询

另外Tokyo Tyrant还把复制作为一项标准功能。memcached不将复制作为标准支持，必须通过repcached等其他方法才能实现复制。而Tokyo Tyrant将复制作为标准支持，所以可以非常容易地进行数据复制处理。

不足

Tokyo Tyrant既不会发生数据丢失，访问的速度也非常快，看上去好像没有什么不足。

但是有一点：它的安装难度比较大。之前我们也介绍了 Tokyo Cabinet/Tokyo Tyrant 有不同的数据库类型，可以根据实际情况进行选择。作为单纯键值存储（一个键对应一个值）的散列数据库，通过 memcached 兼容协议使用起来非常方便，但是它还是要受到 memcached 使用功能的限制。

另一方面，虽然它可以通过自带的协议，根据不同的用途很容易地使用 B-tree 数据库或者表格数据库等，但是这其中也有一些特殊处理。

Tokyo Tyrant 通过 Tokyo Cabinet/Tokyo Tyrant 启动时指定的文件名来指定使用哪种数据库类型，这种方式较为奇特。当文件名是像“hoge. tch”这样以“. tch”结尾的时候使用散列数据库；像“fuga. tcb”这样以“. tcb”结尾的时候使用 B-tree 数据库；像“fuga. tct”这样以“. tct”结尾的时候使用表格数据库。除此之外的情况都使用缓存数据库。若想要使用协调（Tuning）参数，需要在文件名后面通过 # 分割符来指定，例如“hoge. tch # bnum＝1000000 # fpow＝12”等。表 2-6 所示为各种数据类型对应的类名和文件名。

表 2-6 各种数据类型对应的类名和文件名

数据类型	Tokyo Tyrant 的类名	Tokyo Cabinet 的类名	启动时的文件名
缓存数据库	-	-	*
散列数据库	RDB	HDB	*. tch
B-tree 数据库	-	BDB	*. tcb
表格数据库	RDBTBL	TDB	*. tct

另外，需要注意的是它没有 memcached 那么多的成熟技术，安装实例也没有 memcached 那么多，可能会在使用过程中遇到一些原因不明的问题。

应用实例

Tokyo Tyrant 适用于需要高速处理的重要文件。

- **mixi 最后登录时刻的数据处理**[①]
- **通过点餐记录来统计各餐馆的访问数量**[②]

由于这是在日本国内开发的键值存储，所以这里以日本国内的实例为主。据说最近在其他国家亦屡现相关应用实例。[③]

① http://alpha. mixi. co. jp/blog? p=166
② http://www. slideshare. net/tsukasa. oishi/miyazaki-resistance
③ http://www. publickey1. jp/blog/10/nosqltokyo _tyrantnetvibes. html

用例

Tokyo Tyrant的主要用途是为需要获得像memcached那样高速响应的用户提供数据持久化服务。另外，它也提供了和关系型数据库同样通过多个字段把数据保存在硬盘上，且具有高速处理能力的表格数据库。具体来说实际应用如下。

- **需要对重要数据进行频繁的访问**
- **虽不是单纯的键值形式存储，但有些页面处理较慢**

memcached用于处理非重要数据，Tokyo Tyrant用于处理重要数据。虽然二者的基本功能相似，有时也被应用在相同的用途上，但是由于对expires（失效时间）的支持以及复制功能相异，需要大家具体情况具体分析。

一般来说，因为memcached支持expires（失效时间），所以被用来处理需要定期删除的数据。而Tokyo Tyrant由于是把数据保存在硬盘上，所以被用来处理需要半永久性保存的数据。表2-7所示为memcached和Tokyo Tyrant的不同。

表2-7 memcached和Tokyo Tyrant的不同

名称	数据的持久性	expires（失效时间）	复制
memcached	无（保存在内存中）	有	不作为标准支持（通过repcached等支持）
Tokyo Tyrant	有（保存在硬盘上）	无	标准支持

第3章将会以页面访问计数器为例进行详细介绍。

2.2.4 安装步骤

正式安装之前需要通过yum来安装一些程序库。

```
sudo yum install zlib-devel bzip2-devel
```

安装

首先来安装Tokyo Cabinet，本书编写时的版本是1.4.46。

```
wget http://1978th.net/tokyocabinet/tokyocabinet-1.4.46.tar.gz
tar zxvf tokyocabinet-1.4.46.tar.gz
cd tokyocabinet-1.4.46
./configure
make
sudo make install
```

接下来安装 Tokyo Tyrant，本书编写时的版本是 1.1.41。


```
wget http://1978th.net/tokyotyrant/tokyotyrant-1.1.41.tar.gz
tar zxvf tokyotyrant-1.1.41.tar.gz
cd tokyotyrant-1.1.41
./configure
make
sudo make install
```

这样就安装完成了。

启动

马上启动一下试试看吧。我们可以通过其自带的 ttserver 的命令来启动。此时默认设定的待机端口号是 1978。

```
ttserver
```

只需要这样就可以成功启动了，但是由于我们没有指定文件名，所以这次是作为缓存数据库启动的。下面让我们以 hoge.tch 为文件名（只要扩展名是 .tch，文件名任意），作为散列数据库来启动一下。

```
ttserver hoge.tch
```

默认的待机端口号是 1978。大家可以在启动时通过参数来指定待机端口号、日志输出路径和数据类型，等等。启动时可指定的主要参数如表 2-8 所示。

表 2-8 Tokyo Tyrant 启动时的参数

参数	说明	默认值
-port	待机端口号	1978
-pid	输出进程 ID 到指定的文件，作为后台程序运行时必须指定	-
-dmn	作为后台程序运行	-
-log	指定日志输出的路径	-
-ulog	指定更新日志文件（复制时使用）	-
-ulim	指定更新日志的大小（超过指定大小时会覆盖之前的日志）	-
-sid	指定服务器 ID	-
-mhost	指定 master 的主机名（复制时使用）	-
-mport	指定 master 的端口号（复制时使用）	-
-rts	指定复制时的时间戳文件	-

2.2.5 动作确认

通过 telnet 确认

那么，首先让我们通过 telnet 来确认一下。由于它支持 memcached 兼容协议，所以可以像 memcached 一样，通过 set 和 get 这样的基本命令来使用。①

```
$ telnet localhost 1978
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

# set <key> <flag> <expires> <byte> [换行]
# <value>
set foo 0 0 3
bar
STORED

get foo
VALUE foo 0 3
bar
```

① 由于只是部分兼容 memcached，所以像 expires 等一部分功能并没有实现。

```
END

# 这里让我们临时停止 Tokyo Tyrant，
# 然后再次通过相同的命令(ttserver hoge.tch)启动。

get foo
VALUE foo 0 3
bar
END
```


通过这段程序我们能够确认 Tokyo Tyrant 可以像 memcached 一样来使用，但不同的是即使 Tokyo Tyrant 停止了数据也不会丢失。但是需要注意的是由于它不支持 expires（失效时间），即使设定了也不起作用。

在 Ruby 程序上使用（1 台）

接下来让我们在 Ruby 程序上使用 Tokyo Tyrant 试试看。首先安装 Ruby 的 Tokyo Tyrant 客户端，本书编写时的版本是 1.13。

```
wget http://1978th.net/tokyotyrant/rubypkg/tokyotyrant-ruby-1.13.tar.gz
tar zxvf tokyotyrant-ruby-1.13.tar.gz
cd tokyotyrant-ruby-1.13
sudo ruby install.rb
```

这样客户端就安装好了。接下来让我们看看 Tokyo Tyrant 的散列数据库和表格数据库这两种数据库类型的具体程序实例。

首先是散列数据库，让我们通过 Tokyo Tyrant 自带的协议来试用一下。通过 Tokyo Tyrant 自带协议的处理流程是这样的，首先打开数据库，然后进行数据的保存和读取（put 和 get）操作，最后不要忘记关闭数据库。

```
require 'tokyotyrant'

rdb = TokyoTyrant::RDB.new
rdb.open('localhost', 1978)

# 保存数据
rdb.put('hoge', 'fuga')

# 也可以通过这种散列方式保存数据；
# rdb['hoge'] = 'fuga'
```

```
# 数据读取(也可以通过散列方式进行读取)
p rdb.get('hoge') # 'fuga'

# 也可以通过这种散列方式读取;
# p rdb['hoge']

rdb.close
```


散列数据库也可以通过 memcached 兼容协议来使用，让我们也试试这种方式。可以使用与 memcached 相同的方法，只是需要改变一下端口号。使用 memcached 兼容协议的时候，由于可以使用 memcached 的客户端程序库，所以也可以通过 Consistent Hashing 算法来分散数据。使用 memcached 的时候，如果使用 memcached 兼容协议，就可以很容易地把数据导入 Tokyo Tyrant。

```
require 'rubygems'
require 'memcache'

tt = MemCache.new(['localhost:1978']) # 端口号发生了改变;

tt['hoge'] = 'fuga'
p tt['hoge'] # 'fuga'
```

目前为止介绍的都是基本的数据保存和读取处理，Tokyo Tyrant 还有各种方法可供使用，接下来介绍其中一部分。

```
require 'tokyotyrant'

rdb = TokyoTyrant::RDB.new
rdb.open('localhost', 1978)

rdb.put('hoge', 'fuga')
rdb.put('foo', 'bar')
rdb.put('test_1', 'data_1')
rdb.put('test_2', 'data_2')
rdb.put('test_3', 'data_3')

# 是否存在数据
p rdb.has_key?('test_3') # true
p rdb.has_key?('test_4') # false
```

```
# 返回存在的键的一览；
# ["test_1", "test_2", "test_3", "hoge", "foo"]
p rdb.keys

# 返回前方一致的键的一览；
# ["test_1", "test_2", "test_3"]
p rdb.fwmkeys('test')

# 删除所有数据；
p rdb.vanish

rdb.close
```


keys 和 vanish 方法在程序开发时似乎特别有用。

让我们通过另一个例子来看看表格数据库的使用情况。与关系型数据比较相似，除了不支持 JOIN 和事务处理外，其他像正则表达式、合计处理、排序处理这些都能实现。

例如，创建如下三个人的信息数据，通过程序取出按照生日降序排列且姓名以“299”结尾的数据。

```
require 'tokyotyrant'
include TokyoTyrant

rdb = RDBTBL.new
rdb.open('localhost', 1978)

rdb['1'] = {:name => "sasata299", :birthday => "19830308"}
rdb['2'] = {:name => "tarou", :birthday => "19801119"}
rdb['3'] = {:name => "hana299", :birthday => "19860730"}

query = RDBQRY.new(rdb)
query.addcond("name", RDBQRY::QCSTREW, "299")
query.setorder("birthday", RDBQRY::QONUMDESC)
res = query.search()

# 只返回键值；
p res # ["3", "1"]

# 通过 get 方法可以取到具体的值；
p rdb.get(res[0]) # {"name" => "hana299", "birthday" => "19860730"}
p rdb.get(res[1]) # {"name" => "sasata299", "birthday" => "19830308"}
```

```
# 知道键的情况下也可以直接取得数据；
p rdb.get('2') # {"name" => "tarou", "birthday" => "19801119"}

rdb.close
```

在程序中出现了几个常量。表格数据库虽然使用起来非常方便，但其使用过程中会出现各种常量，这里介绍几个比较典型的。表 2-9 所示为表格数据库常量的含义（精选）。

表 2-9 表格数据库常量的含义（精选）

常量	含义
TokyoTyrant::RDBQRY::QCSTREQ	字符串完全匹配查询
TokyoTyrant::RDBQRY::QCSTRINC	字符串模糊（LIKE）查询
TokyoTyrant::RDBQRY::QCSTRBW	字符串前方一致
TokyoTyrant::RDBQRY::QCSTREW	字符串后方一致
TokyoTyrant::RDBQRY::QCNUMEQ	数值比较（=）
TokyoTyrant::RDBQRY::QCNUMGT	数值比较（>）
TokyoTyrant::RDBQRY::QCNUMGE	数值比较（>=）
TokyoTyrant::RDBQRY::QCNUMLT	数值比较（<）
TokyoTyrant::RDBQRY::QCNUMLE	数值比较（<=）
TokyoTyrant::RDBQRY::QOSTRDESC	文字降序排列
TokyoTyrant::RDBQRY::QOSTRASC	文字升序排列
TokyoTyrant::RDBQRY::QONUMDESC	数值降序排列
TokyoTyrant::RDBQRY::QONUMASC	数值升序排列

这些常量所代表的含义，若不去查询是很难完全理解的为了省去这些麻烦，可使用 Tokyo Tyrant 的表格数据库，以及后面将要介绍的 MiyazakiResistance 和 ActiveTokyoCabinet 等封装方法，来隐藏这些常量。

在 Ruby 程序上使用（多台）

接下来让我们在多台服务器上试用一下 Tokyo Tyrant。由于使用了 memcached 兼容协议，所以也可以像 memcached 一样使用 ConsistentHashing 算法来进行数据分散。我们可以分别指定待机端口号（这次使用 1978、1979、1980）来启动 Tokyo Tyrant，具体操作如下。

```
ttserver data_1978.tch
ttserver -port 1979 data_1979.tch
ttserver -port 1980 data_1980.tch
```

程序方面，除了端口号不同之外，其他都跟 memcached 的使用方法相同。


```
$KCODE = 'u'

require "rubygems"
require "memcache"
require "logger"

server = [
  'localhost:1978',
  'localhost:1979',
  'localhost:1980'
]
option = {
  :logger => Logger.new(STDOUT)
}

cache = MemCache.new(server, option)

# 数据的保存；
cache['key1'] = 123                                    # 数值；
cache['key2'] = "ABCDE"                                # 字符串；
cache['key3'] = %w(hoge fuga)                          # 数组；
cache['key4'] = {:foo => 1, :bar => "a"}               # 散列表；

# 数据的读取；
p cache['key1'] # 123
p cache['key2'] # "ABCDE;"
p cache['key3'] # ["hoge", "fuga"]
p cache['key4'] # {:bar => "a", :foo => 1}
```

为了更容易地理解数据是如何被分散到各个服务器上的，我们通过设定构造函数的参数来指定 logger，事实表明，数据的确是通过 Consistent Hashing 算法被分散到各个服务器上的。

```
# 作为处理对象的 memcached 的信息；
I, [2011-03-06T19:07:02.751315 # 20507] INFO--: memcache-client 1.8.5
```

```
["localhost:1978", "localhost:1979", "localhost:1980"]
D, [2011-03-06T19:07:02.751506 # 20507] DEBUG --:Servers now:
[<MemCache::Server:localhost:1978 [1] (NOT CONNECTED)> ,
<MemCache::Server:localhost:1979 [1] (NOT CONNECTED)> ,
<MemCache::Server:localhost:1980 [1] (NOT CONNECTED)> ]

# 数据的保存;
D, [2011-03-06T19:07:02.761738 # 20507] DEBUG--:set key1 to <MemCache::Server:
localhost:1980 [1] (CONNECTED)> :5
D, [2011-03-06T19:07:02.763158 # 20507] DEBUG--:set key2 to <MemCache::Server:
localhost:1979 [1] (CONNECTED)> :19
D, [2011-03-06T19:07:02.763714 # 20507] DEBUG--:set key3 to <MemCache::Server:
localhost:1979 [1] (CONNECTED)> :16
D, [2011-03-06T19:07:02.764439 # 20507] DEBUG--:set key4 to <MemCache::Server:
localhost:1978 [1] (CONNECTED)> :19

# 数据的读取;
D, [2011-03-06T19:07:02.764831 # 20507] DEBUG--:get key1 from
<MemCache::Server:localhost:1980 [1] (CONNECTED)>
123
D, [2011-03-06T19:07:02.765529 # 20507] DEBUG--:get key2 from
<MemCache::Server:localhost:1979 [1] (CONNECTED)>
"ABCDE"
D, [2011-03-06T19:07:02.765871 # 20507] DEBUG--:get key3 from
<MemCache::Server:localhost:1979 [1] (CONNECTED)>
["hoge", "fuga"]
D, [2011-03-06T19:07:02.766294 # 20507] DEBUG--:get key4 from
<MemCache::Server:localhost:1978 [1] (CONNECTED)>
{:bar => "a", :foo => 1}
```

设定复制

Tokyo Tyrant将复制处理作为一项标准功能。下面让我们来实际操作一下。首先分别启动一台作为主数据库和一台作为从数据库的服务器。

```
# 建立更新日志的保存目录;
mkdir ulog_1978
mkdir ulog_1979

# 主数据库的启动;
ttserver-ulog ulog_1978 -sid 1 data_1978.tch

# 从数据库的启动;
ttserver -port 1979 -ulog ulog_1979 -sid 2 -mhost localhost -mport 1978 -rts
rts_1979.rts data_1979.tch
```

复制的时候通过-ulog参数来指定更新日志文件的保存目录，然后通过从数据库来指定主数据库的主机名（-mhost）、端口号（-mport），以及用来记录复制时间点的复制时间戳文件(-rts)，这样复制的准备工作就完成了。

使用 telnet 通过 1978 号端口连接 Tokyo Tyrant（主数据库），尝试保存数据。

```
$ telnet localhost 1978
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set hoge 0 0 4
fuga
STORED

get hoge
VALUE hoge 0 4
fuga
END
```

数据被成功保存，读取也没有任何问题。这时，在通过-ulog参数指定的 ulog_1978 目录中更新日志也被创建出来了，这样应该就已经把这条数据复制到 1979 号端口的 Tokyo Tyrant（从数据库）中了。赶快来确认一下吧。

```
$ telnet localhost 1979
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

get hoge
VALUE hoge 0 4
fuga
END
```

向 1979 号端口的 Tokyo Tyrant（从数据库）中复制数据成功，通过-rts参数指定的时间戳文件也被创建出来了。但是这样进行下去的话，更新日志文件会无限增大，所以需要大家通过-ulim参数来设定更新日志文件的大小，恰当地进行循环使用。

2.2.6 各种开发语言需要用到的程序库

Tokyo Tyrant为C、Perl、Ruby、Java、Lua、Python、PHP、Erlang、Haskell等各种开发语言提供了相应的程序库。表2-10所示为Tokyo Tyrant启动时的参数。

表2-10 Tokyo Tyrant启动时的参数

开发语言	程序库
Ruby	tokyotyrant-ruby
Perl	tokyotyrant-perl
Java	tokyotyrant-java
Python	pytyrant、python-tokyotyrant
PHP	Net _TokyoTyrant、php-tokyo _tyrant

2.2.7 相关工具

下面介绍MiyazakiResistance和ActiveTokyoCabinet，它们都是对表格数据库进行的封装，是非常方便的程序库，能够使表格数据库更易于操作。

MiyazakiResistance

首先我们通过如下命令来安装MiyazakiResistance，本书编写时的版本是0.1.6。

```
gem install miyazakiresistance
```

使用方法如下所示。原本表格数据库是没有表结构信息（表中保存了哪些字段这样的信息）的，但是MiyazakiResistance却需要指定表结构信息并加以管理。通过指定表结构信息可以使管理更加容易，还可以在取得数据时把它们映射为各种不同的类（例如datetime类型的字段会自动变换为Time类）。能够处理的字段类型有string、integer、date、datetime 4种。

另外，由于二元主机结构（双机热备—Active standby）和主从结构都很容易构筑，这样就可以很方便地应用到各种用途中了。[①]

① http://www.kearuspoon.net/artieles/706

使用 MiyazakiResistance 的例子

```
require 'rubygems'
require 'miyazakiresistance'

class User <MiyazakiResistance::Base
  set_server "localhost", 1978, :write

  set_column :name, :string
  set_column :age, :string
  set_column :created_at, :datetime
end

user = User.new(:name => "sasata299")
user.age = 27
user.save

p User.count # 1

# 查询条件必须用 AND 连接；
user = User.find(
  :first,
  :conditions => ["name = ? age = ?", "sasata299", 27],
  :order => "created_at DESC",
  :limit => 10
)
p user # # <User:0xb7e373dc @created_at = Fri Feb 04 06:19:27 +0900 2011, @
name = "sasata299", @id = 1, @age = "27">
```

ActiveTokyoCabinet

同样还是先来安装 ActiveTokyoCabinet，本书编写时的版本是 0.2.1。另外，ActiveTokyoCabinet 需要使用 ActiveSupport，但由于 ActiveSupport3 系列的版本无法运行，所以需要使用 ActiveSupport2 系列的版本。

```
gem install activetokyocabinet
```

ActiveTokyoCabinet 和 MiyazakiResistance 一样，也定义了表结构信息。ActiveTokyoCabinet 是作为 ActiveRecord（构成 Ruby on Rails 的程序库之一）的一个适配器来实现的，能够获得和 ActiveRecord 同样的使用体验。数据的校验（validation）也可以简单地完成。ActiveTokyoCabinet 能够处理的字段类型有 string、int、float 三种。

使用 ActiveTokyoCabinet 的例子

```
require 'rubygems'
require 'active_record'
require 'active_tokyocabinet/tdb'

ActiveRecord::Base.establish_connection(
  :adapter => 'tokyotyrant',
  :database => {
    :users => {:host => 'localhost', :port => 1978},
  }
)

class User <ActiveRecord::Base
  include ActiveTokyoCabinet::TDB

  string :name
  int :age
end

user = User.new(:name => "sasata299")
user.age = 27
user.save

p User.count # 1
user = User.find(
  :first,
  :conditions => ["name = ? AND age = ?", "sasata299", 27], # 不能进行 OR 查询;
  :order => "age DESC",
  :limit => 10
)
p user # # <User id:1, name:"sasata299", age:"27">
```

虽然 MiyazakiResistance 和 ActiveTokyoCabinet 非常相似，但还是存在是否支持 date/datetime 类型、是否支持校验等一些不同之处，请大家在使用时具体问题具体分析。另外，还需注意它们都只支持 AND 查询（无法进行 OR 查询）。表 2-11 所示为 MiyazakiResistance 和 ActiveTokyoCabinet 的主要不同。

表 2-11 MiyazakiResistance 和 ActiveTokyoCabinet 的主要不同

名称	支持date/datetime 类型	二元主机	数据校验
MiyazakiResistance	○	○	×
Tokyo ActiveTokyoCabinet	×	×	○

2.3 Redis（临时性/持久性键值存储）

2.3.1 什么是 Redis

Redis 是 Salvatore Sanfilippo 开发的开源软件。由于它兼具临时性和永久性，所以它是 NoSQL 数据库中介于 memcached 和 Tokyo Tyrant 之间的键值存储。

官方网站：http://redis.io/

2.3.2 为什么要使用 Redis

处理数组形式的数据

目前为止，我们提到了高速返回响应的重要性，也介绍了引入 memcached 和 Tokyo Tyrant 等键值存储的效果。确实，如果要处理字符串数据和标准的散列数据，memcached 和 Tokyo Tyrant 这样的键值存储可能已经足够了。但是，根据用途的不同，也不乏对快速处理数值和数组类型数据的要求。最近像 Twitter 的时间线这样，根据时间序列添加新数据，并且频率非常高的服务也越来越多了。

数据的一致性

这种情况下如果使用关系型数据库，随着数据量和访问量的增加，响应情况肯定会越来越差。当然，可以像前面介绍的那样，对数组形式的数据进行序列化，然后通过 memcached 或者 Tokyo Tyrant 来进行处理，就能获得高速的响应。但是，在操作数组形式数据的时候，一定要注意保持数据的一致性，这在实际应用中会比较困难。

通常情况下，使用 memcached 或者 Tokyo Tyrant 更新数据的时候，必须完成下述的三个处理：

- 取得数据
- 变更数据
- 保存数据

虽然每一步只需要很短的时间，但由于都是单独的处理，所以不管怎样，数据的读取、变更和更新之间都会存在时间延迟。当多个请求同时对同一个键对应的数据进行更新的时候就可能发生数据不一致的情况。而 Redis 会把这样的一连串的操作定义为原子操作[①]，并且提供很多命令来保证数据的一致性。

说到原子操作，可能理解起来不是那么形象，简单地说就是一连串的操作通过一次命令来执行。即使有多个客户端同时操作字符串类型的数据，基本上也只是在最新的数据上进行更新，所以不会有什么问题。但是，如果处理的数据是计数器这样可以进行加减的数值，或者是对数组形式的数据进行添加或者读取操作的时候，是否需要使用原子操作就变得非常关键了。对于数值的加减处理，memcached 提供了 incr 和 decr 方法，Tokyo Tyrant 提供了 addint 方法，这些都是原子操作的方法，但是对于这些之外的处理来说，要保证数据的一致性就非常困难了。

让我们来看一些具体的例子。

memcached 提供的原子操作（计数器）

让我们使用 memcached 来实现一个计数器功能，假定有两个进程并行，首先考虑一下不使用原子操作的时候会是什么样的情况。

进程 1 **从 memcached 中取得当前计数器的值（取到的值是 3）**

进程 1 **把计数器的值加 1（计数器的值变成了 4）**

进程 2 **从 memcached 中取得当前计数器的值（当前时点取到的值还是 3）**

进程 1 **把计数器的值保存到 memcached 中（数值 4 被保存到了 memcached 中）**

进程 2 **把计数器的值加 1（计数器的值变成了 4）**

进程 2 **把计数器的值保存到 memcached 中（数值 4 被保存到了 memcached 中）**

本来计数器的值应该加 2，最终变成 5，但实际上却变成了 4。像这样在某个进程取出数据到保存数据期间，其他进程也对同一条数据进行处理的话，就会造成数据不一致的发生。这个时候，如果使用 memcached，就需要通过 CAS（Compare-and-Swap 或者 Check and Set 的简称）操作来保证数据的一致性。

CAS 操作

CAS 操作就是使用 gets 和 cas 这两个命令来避免数据不一致的发生，其中 gets 是读取数据的命令，cas 是保存的命令。

首先，使用 gets 命令不仅可以读取数据，还能取得 cas id 的值。cas id 是数据的表

① 请参考 URL：http://ja.wikipedia.0org/wiki/不可分操作。

示值，在更新数据的时候是肯定会发生改变的，并且在保存数据的时候，会与被保存的数据一起传递给 cas 命令。这时，只有传递给 cas 命令的 cas id 和表示数据时用到的 cas id 一致，数据才会被保存。cas id 不一致的时候，就说明通过 gets 命令取到的数据发生了改变，数据保存就会失败。这样就可以在保存的时候确认数据是否被其他进程改变了，从而避免数据不一致的发生。图 2-8 所示为 memcached 的 CAS 操作。

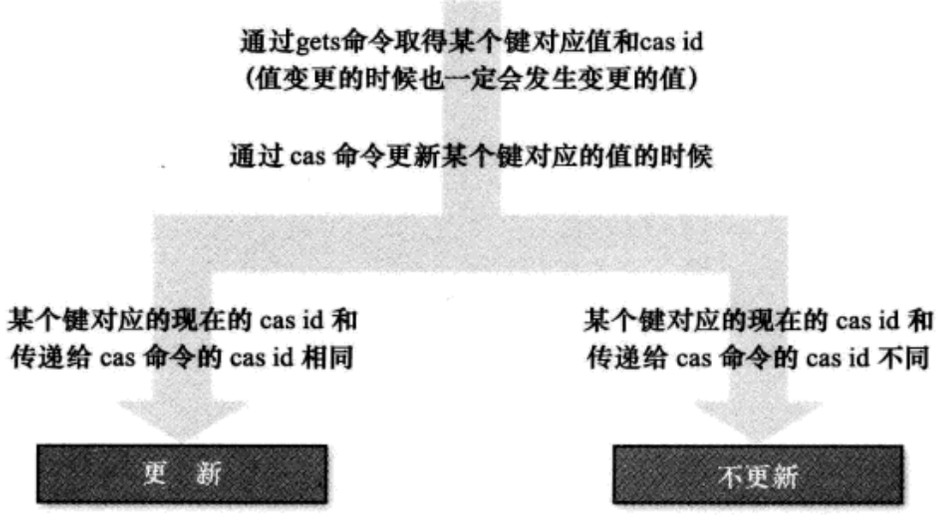


图 2-8 memcached 的 CAS 操作

下面让我们通过计数器这个例子来验证一下 CAS 操作[①]。首先，从某个终端连接 memcached（进程 1），把计数器的值设定为 3。然后，使用 gets 命令取出计数器的值和 cas id。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set counter 0 0 1
3
STORED

# 这时 cas id 的值是9
gets counter
VALUE counter 0 1 9
3
END
```

① 用数值加法作为例子很容易就能明白，即使不特意的进行 CAS 操作，由于使用 incr 和 decr 方法就能进行原子操作，在实际应用中也没有什么问题。

此时，进程1还没有对数据进行变更，这时通过另外一个终端连接memcached（进程2），把计数器更新为其他的数值。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set counter 0 0 1
4
STORED

# 这时 cas id 的值是10;
gets counter
VALUE counter 0 1 10
4
END
```

这时计数器的值变成了4，cas id的值也从9变成了10。进程1并不知道进程2把计数器的值更新了。在这样的状态下，当进程1试图使用cas命令保存计数器的值的时候会发生什么呢?

```
# 这个时候计数器的 cas id 已经变成了10,所以失败了;
# 失败的时候会返回 EXISTS;
# cas <key> <flag> <expires> <byte> <casid> [换行]
# <value>
cas counter 0 0 1 9
4
EXISTS

# 再次取得 cas id;
# 同时也得到了现在计数器的值是4;
gets counter
VALUE counter 0 1 10
4
END

# 使用正确的 cas id 更新成功;
cas counter 0 0 1 10
5
STORED
```

由于cas id不同，最初更新计数器的时候会失败，返回EXISTS。但再次取得最

新的 cas id，就可以通过它成功地更新计数器的值了。通过再次的读取操作就能得到最新的数据，因此就可以把这个值加 1 并正常地进行更新了。虽然每次更新的时候都确认返回值比较费功夫，但这样就可以避免数据不一致的发生。

memcached 提供的原子操作（数组类型）

我们再来看另外一个例子，看看 memcached 是如何对数组类型的数据进行处理的。假设通过多个客户端进行连接，尝试处理如下的例子。我们首先不使用原子性的 CAS 操作。

非原子操作

```
require 'rubygems'
require 'memcache'

cache = MemCache.new(['localhost:11211'])

# 保存一个[1,2,3]这样的数组数据；
cache['hoge'] = [1, 2, 3]

# 假设有多个客户端连接；
# 进程1取得了数组 array_1,进程2取得了数组 array_2；
array_1 = cache['hoge']
array_2 = cache['hoge']

array_1 <<4 # 进程1向数组添加了4；
array_2 <<5 # 进程2向数组添加了5；

cache['hoge'] = array_1
p cache['hoge'] # [1, 2, 3, 4]

cache['hoge'] = array_2
p cache['hoge'] # [1, 2, 3, 5]
```

虽然进程 1 向数组中添加了元素 4，但并没有最终被反应到数组中。进程 1 的处理被忽略了，这就造成了数据的不一致。

接下来让我们尝试一下使用原子性的 CAS 操作吧。这次我们使用 memcache-client，在 Ruby 的程序上进行 CAS 操作。memcache-client 从 1.7.0 版本开始支持 CAS 操作，所以必须使用 1.7.0 以上的版本[①]。

① https://github.com/mperham/memcache-client/blob/master/History.rdoc

原子操作

```
require 'rubygems'
require 'memcache'

cache = MemCache.new(['localhost:11211'])

# 保存一个[1,2,3]这样的数组数据;
cache['hoge'] = [1, 2, 3]

# 假设有多个客户端连接;
retry if cache.cas('hoge') {| value| value << 4} !~ /^STORED/
retry if cache.cas('hoge') {| value| value << 5} !~ /^STORED/

p cache['hoge'] # [1, 2, 3, 4, 5]
```


使用 memcache-client 不需要明确地执行 gets 命令，在 cas 命令执行的时候会自动调用 gets 命令，并对取到的 cas id 的一致性进行校验。

下面的程序使用的就是是原子操作，由于在保存数据时一定要校验当前的 cas id 和最初取得的 cas id 是否一致，所以在多个进程同时对同一个键进行处理的时候肯定有一些会失败。处理失败的时候（不是返回 STORED 而是返回 EXISTS）将会再次执行保存数据的处理。这样就完成了原子性的 CAS 操作。

```
retry if cache.cas('hoge') {| value| value << 4} !~ /^STORED/
```

完整的 cas 操作如下所示。

memcache-client-1.8.5/lib/memcache.rb

```
def cas(key, expiry = 0, raw = false)
  raise MemCacheError, "Update of readonly cache" if @ readonly
  raise MemCacheError, "A block is required" unless block_given?

  (value, token) = gets(key, raw)
  return nil unless value
  updated = yield value
  value = raw ? updated :Marshal.dump(updated)

  with_server(key) do | server, cache_key|
    logger.debug { "cas #{key} to #{server.inspect}:#{value.to_s.size}" } if
logger
    command = "cas #{cache_key} 0 #{expiry} #{value.to_s.size}
#{token}#{noreply}\ r\ n#{value}\ r\ n"
```

```
        with_socket_management(server) do | socket|
          socket.write command
          break nil if @ no_reply
          result = socket.gets
          raise_on_error_response! result

          if result.nil?
            server.close
            raise MemCacheError, "lost connection to #{server.host}:#{server.port}"
          end

          result
        end
      end
    end
```

memcached 在进行 CAS 操作的时候，由于这三个处理（数据读取、变更、保存）还是独立执行的，所以数据被其他进程改变的可能性还是存在的。因此，必须要对返回值进行校验，数据保存失败的时候需要加入纠正处理的逻辑。尽管这样就能实现原子操作，但是这样的逻辑处理起来还是需要花一定工夫的。

对数组类型数据进行的优化处理

这时就轮到 Redis 出场了。Redis 是键值存储的一种，但是它对链表（list）和集合（set）等数组类型的数据进行了优化处理，可以对数组类型的数据进行高速的插入和读取处理。另外，Redis 包含很多可以把这些处理原子化的命令，所以可以非常容易地保证数据的一致性。

如前所述，memcached 虽然也可以通过 CAS 操作来保证数据的一致性，但是仍然需要考虑多进程的问题，处理失败的时候还要考虑重新处理的逻辑，比较让人头疼。如果想处理数组类型的数据，Redis 真的是不错的选择。

2.3.3 特征和用例

特征

由于 Redis 通常是把数据保存在内存中，所以是处理速度非常快的键值存储。虽然已经存在了同样是在内存中保存数据的键值存储 memcached，但这两者的用途却大不相同。memcached 主要是用作关系型数据库的缓存，与关系型数据结合使用，

对简单的操作进行优化处理。与之相对，Redis本身就是作为数据存储而设计出来的，它的操作命令非常多（超过100种），其中很多很多都支持原子操作。

另外，memcached按照LRU（Least Recently Used）的顺序删除不使用的数据，而Redis通过命令明确地删除数据，除非设定了expires（失效时间），数据是不会被自动清除的。但是，Redis从2.1版本也开始支持LRU，设定之后也可以像memcached那样做为缓存进行使用。这样，就可以在保持数据永久性的同时，对expires（失效时间）过期的数据进行自动删除了。

数据快照

Redis在向硬盘写入数据的时候，还提供了数据快照这样的永久化功能。内存中数据的快照被写入到文件（*.rdb）中，再启动的时候数据快照中的内容就会被读入到内存中，这样就可以恢复到上次数据快照时的状态了。写入的时间（默认是15分钟1次以上，5分钟10个以上，1分钟1万个以上的键被修改的情况下发起快照）可以通过设定文件任意改变。如果把Append Only File选项设定为有效，那么所有的更新命令都可以随时记录到Append Only File中。即使服务器被迫停止的时候也能把所有的更新命令都记录到Append Only File中，因此不会造成数据丢失。

优势

Redis可以处理字符串、链表（list）、集合（set）、有序集合（zset）、散列表等各种类型的值数据，但需要注意的是所有数据都会被当作字符串来进行处理（数值的保存也是一样）。

```
require 'rubygems'
require 'redis'

redis = Redis.new

redis.set :number, 1
result = redis.get :number

p result # "1"
p result.class # String
```

如果想要处理数值类型的数据，就需要通过JSON等方法对其进行序列化。

```
require 'rubygems'
require 'redis'
require 'json'

redis = Redis.new

json = [1, 2, 3].to_json
redis.set :number, json
p JSON.parse(redis.get :number) # [1, 2, 3]
```


这些数据类型具有如下特征：

- **字符串类型**

  字符串是 Redis 可以处理的最基本的类型。字符串类型也可以用来保存 JPEG 图片和被序列化的 Ruby 对象。

- **链表（list）类型**

  链表类型指的是字符串链表。新元素可以添加在链表的表头（左侧）或者表尾（右侧）。与链表的长度无关，都可以保证 PUSH 和 POP 这样操作的复杂度为 O（l）（也就是说，与链表的长度无关，都能够在一定时间内进行处理）。

- **集合（set）类型**

  集合类型指的是字符串类型的无序集合，它的特点是拥有多个不重复的元素。元素的添加、删除、确认操作的复杂度都能达到 O（l）。它和链表类型非常相似，但是有些许相异：链表类型一般是以字符串类型数据作为元素，并且可以通过 PUSH 或者 POP 进行变更不同，集合类型只是字符串类型数据的集合。另外，集合类型的数据也可以通过交集或者并集等集合运算符来进行简单的集合运算。

- **有序集合（zset）类型**

  有序集合类型和集合类型非常相似。但与集合类型不同的是，有序集合类型的元素都分别拥有一个被称为 score 的值，并且按照这个值进行排序。有序集合类型和链表类型也很相似，但是与链表类型的元素按照添加的先后顺序来排序不同，有序集合类型是按照 score 的顺序来进行排序的。由于 score 可以设定为任意值，所以它可以非常灵活地进行排序。

- **散列表（hash）类型**

散列表类型是没有顺序的字符串类型的字段和值的映射，也就是一般所说的散

列表的形式。

只看这些说明可能理解起来比较困难，让我们用图表的形式来展示一下这些数据类型的实际结构，作为参考。图 2-9 所示为每个数据类型对应的数据结构。

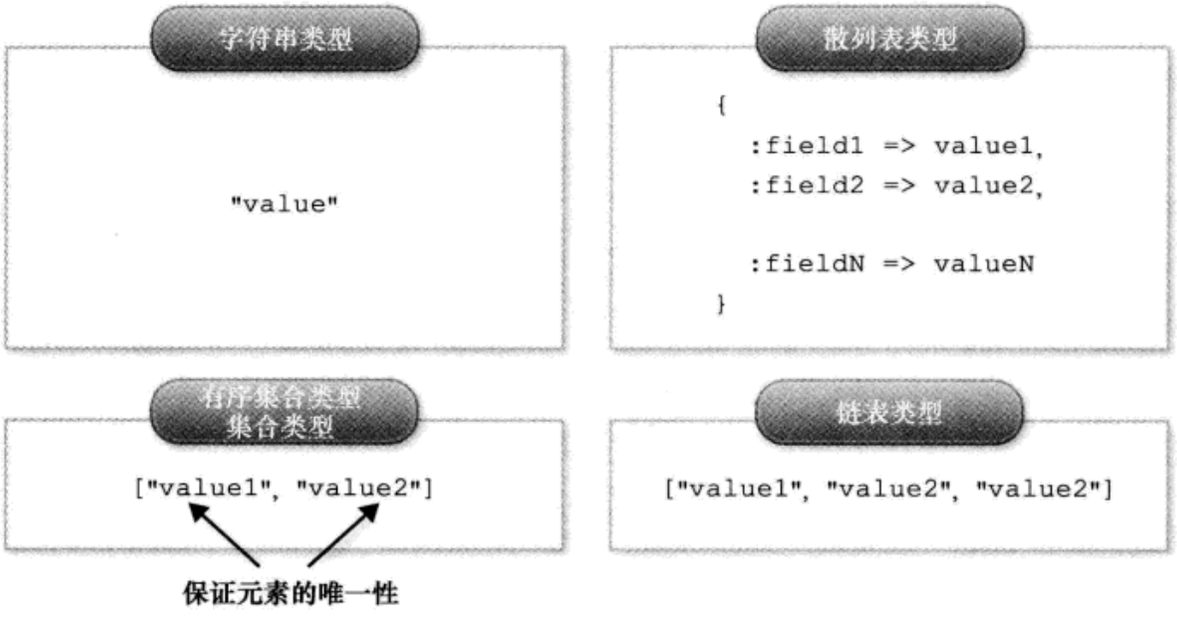


图 2-9 每个数据类型对应的数据结构

当然，memcached 或者 Tokyo Tyrant 也可以处理数组类型或者散列表类型的值数据，但是如前所述，Redis 能够很简单地进行原子处理是它最大的优势。不仅仅是数据的添加和读取，类似于向索引列表中添加数据、从索引列表中删除数据、模式匹配和对数组类型的数据进行的各种处理，都可以在保证数据一致性的前提下完成。

同时，由于数据通常保存在内存中，所以处理速度非常快。Redis 会定期对数据进行快照处理，除了一部分当前的更新之外，数据都不会丢失。虽然进行数据快照的时候会增加负荷（需要对所有数据进行 IO 处理），但是由于数据快照的 IO 处理通常都是序列化访问，所以非常高效。

不足

Redis 最大的问题就是这项新技术的使用实例比较少。它的使用实例甚至少于其他 NoSQL 数据库，使用者很可能会遇到不清楚使用方法，以及不知该如何选择更合适的解决方案等问题。

此外，以前的版本只能操作实际内存中的数据，有一定的局限性。但是，2.0 以后的版本实现了独立的虚拟内存结构，可以把实际内存中保存不下的数据写入到硬盘中，这样就可以实现从内存中读取频繁使用的数据，从硬盘上读取其他数据，扩

大了它的使用范围。

虚拟内存结构

虚拟内存是可以和操作系统进行互换的结构，是 Redis 在参考 Linux 内核的基础上独立实现的。通过虚拟内存的独立实现，提高了数据的压缩率，即使数据量超过了实际内存的容量，也可以通过在内存中保存所有数据的键来实现高效的处理。因为键保存在内存中，所以可以十分迅速地完成通过键读取数据的处理，这也是 Redis 可以进行高速处理的原因。在内存中保存 100 万个键只需要 160M 的内存就足够了。

使用实例

Redis 已经被应用到若干大规模服务中了。

- **github**
- **digg**
- **微笑直播（ニコニコ生放送）的观众人数统计和新节目一览等**[①]

特别是在微笑直播中得到了相当大规模的应用。

用例

Redis 可以进行原子操作，非常适合用来处理链表和集合这样的数组类型数据。具体的实例如下所示。

- **实现需要表示时间线的 web 服务**
- **对数组形式的数据进行频繁地添加和删除操作**

第 3 章将会以 Twitter 风格的时间线表示等简单的 web 应用和查询语句为例，进行具体的介绍。

2.3.4 安装步骤

安装

首先来安装 Redis，本书编写时的版本是 2.0.4。

① http://gihyo.jp/dev/feature/01/redis/0001?page=2

```
wget http://redis.googlecode.com/files/redis-2.0.4.tar.gz
tar zxvf redis-2.0.4.tar.gz
cd redis-2.0.4
make
sudo cp redis-server /usr/local/bin
sudo cp redis.conf /etc
```


启动

如下所示，通过参数指定配置文件来启动。默认的待机端口号是6379。待机端口号以及数据快照等设定可以通过配置文件来完成。表2-12所示为Redis配置文件中的项目。

```
redis-server /etc/redis.conf
```

表2-12 Redis配置文件中的项目

项目名	说明	默认值
daemonize	需要作为后台程序运行的时候设置成yes	no
port	待机端口号	6379
timeout	在这里指定多少秒之内没有响应就关闭连接	300
save <sec> <change>	在sec秒内至少有change次变更的时候写入硬盘	save 900 1 save 300 10 save 60 10000
dir	数据快照写入的目录	./
dbfilename	数据快照的文件名	dump. rdb
slaveof	使用复制的时候，在这里指定主数据库的主机名和端口号	-

2.3.5 动作确认

通过telnet确认

首先让我们通过telnet来进行动作确认，其方法和memcached以及Tokyo Ty-

rant 不尽相同。

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

# set <key> <byte> [换行]
# <value>
set hoge 4
fuga
+ OK

# $ 4的意思是4字节的数据；
get hoge
$ 4
fuga

# 利用 expires(失效时间)的时候使用的不是 set 而是 setex；
# setex <key> <expires> <byte> [换行]
# <value>
setex foo 5 3
bar
+ OK

get foo
$ 3
bar

# 5秒之后数据就消失了；
# 数据不存在的时候返回$ -1；
get foo
$ -1
```


它包含各种各样的命令，这里让我们试验一下 set、get、setex 这几个命令。get 就不用再说明了吧，set 和 setex 看上去有些相似，它们的区别是：可以使用 expires 的是 setex，不能使用 expires（也就是说，数据是半永久性的，不会丢失）的是 set。

除了字符串类型数据，我们也来试着操作一下链表类型的数据吧。

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

# 因为链表 data 并不存在,所以会自动的创建链表 data,
# 并在开头加入数据 foo;
# lpush <key> <byte> [换行]
# <value>
lpush data 3
foo
:1

# 在链表 data 的开头加入数据 bar;
lpush data 3
bar
:2

# 取得链表 data 从开头(0)到末尾(-1)的数据;
# *2表示链表有2个元素的意思;
# lrange <key> <start> <end>
lrange data 0-1
*2
$3
bar
$3
foo

# 删除与 data 关联的数据;
del data
:1
```

以上试用了 lpush、lrange、del 命令。lpush 是在链表（数组）表头位置添加数据的命令，如果键不存在的话会在添加数据之前自动创建一个空的链表。lrange 是针对链表类型的数据，返回包含指定键的任意索引的元素。del 是对任何数据类型都能使用的命令，它可以删除指定键关联的数据。图 2-10 所示为针对链表类型的基本命令。

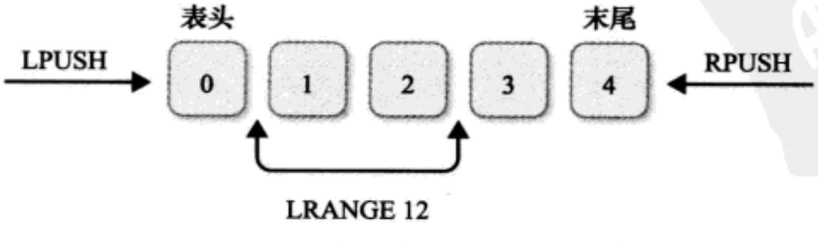


图 2-10 针对链表类型数据的基本命令

最后让我们尝试一下有序集合类型。

```
# 向有序集合 sets 中添加数据；
# zadd <key> <score> <byte> [换行]
# <value>
zadd sets 1 4
hoge
:1

zadd sets 2 4
fuga
:1

zadd sets 3 3
foo
:1

# 返回索引从0(开头)到- 1(末尾)的数据；
# zrange <key> <start> <end>
zrange sets 0- 1
*3
$ 4
hoge
$ 4
fuga
$ 3
foo

# 返回 score 从1到2的数据；
# zrangebyscore <key> <min> <max>
zrangebyscore sets 1 2
*2
$ 4
hoge
$ 4
fuga
```

以上试用了 zadd、zrange、zrangebyscore 这几个命令。zadd 是向作为有序集合的特定链表中添加数据的命令。zrangebyscore 是取得指定范围内数据的命令。zrange 命令可以在索引的范围内，通过 zrangebyscore 指定 score 的最小值和最大值，取出包含边界值的数据。

有序集合不仅能通过指定索引，还可以通过指定 score 取得任意范围内的数据。如果能恰当地利用这些特点，就可以将其进行广泛的应用。

对于不同的数据类型，Redis 还可使用许多其他命令，表 2-13 给大家介绍其中一部分。

表 2-13 Redis 可以使用的命令（精选）

数据类型	命令名称	命令内容	参数
字符串类型	SET	把值保存到键中	key，value
	SETEX	把带有 expires 的值保存到键中	key，expires，value
	GET	取得键对应的值	key
链表类型	LPUSH	在键对应的链表开头加入值	key，value
	RPUSH	在键对应的链表末尾加入值	key，value
	LLEN	返回键对应的链表中的元素	key
	LRANGE	返回键对应的链表中指定索引范围（start..end）内的元素	key，start，end
	LTRIM	对键对应的链表中指定索引范围（start..end）内的元素进行 trim 处理	key，start，end
	LINDEX	返回键对应的链表中指定索引的元素	key，index
集合类型	SADD	把指定的 member 保存到键中	key，member
	SCARD	返回键关联的元素个数	key
	SMEMBERS	返回键关联的集合内的所有成员	key
	SUNION	返回指定的键 1，键 2，…的并集	key1，key2，…
	SINTER	返回指定的键 1，键 2，…的交集（共通成员）	key1，key2，…
有序集合类型	ZADD	通过 score 保存与键对应的指定的成员	key，score，member
	ZRANGE	在键对应的元素中，返回指定索引范围（start..end）内的元素	key，start，end
	ZRANGEBYSCORE	在键对应的元素中，返回指定 score 范围（min..max）内的元素	key，min，max

（续）

数据类型	命令名称	命令内容	参数
散列表类型	HSET	把值保存到键对应的散列表中指定的 field 中	key，field，value
	HGET	返回键对应的散列表中指定 field 对应的值	key，field
	HKEYS	使用 glob 形式的模型返回存在的 field 的一览	glob 形式的 field 模型
	HEXISTS	键对应的散列表内存在指定的 field 的时候返回“1”，不存在的时候返回“0”	key，field
	HDEL	删除键对应的散列表中的 field	key，field
类型无关	DEL	删除键对应的数据	key
	KEYS	使用 glob 形式的模型返回存在的 key 的一览	glob 形式的模型 例如：user：*
	EXISTS	指定的键存在的场合返回“1”，不存在的时候返回“0”	key
	RENAME	把键的名字从 old _key 改成 new _key	old _key，new _key
	TYPE	返回指定键关联的数据的类型	key

在 Ruby 程序上使用（1 台）

在 Ruby 上使用 Redis 需要安装 redis-rb 程序库[①]，下面我们就尝试通过它来使用 Redis。通过下面的命令来安装 redis-rb，本书编写时的版本是 2.3.5。为了避免实际使用中出现警告信息，还需要安装 system _timer 这个程序库。

```
gem install redis
gem install system_timer
```

首先让我们确认一下 Redis 处理链表类型数据的时候是否能够保证数据的一致性。

① http://github.com/ezmobius/redis-rb

```
require 'rubygems'
require 'redis'

# 同样假设有多个客户端连接；
redis = Redis.new
redis2 = Redis.new

# 保存["1", "2", "3"]这样的数组数据；
redis.lpush :hoge, 3
redis.lpush :hoge, 2
redis.lpush :hoge, 1

redis.rpush :hoge, 4
p redis.lrange :hoge, 0, -1# ["1", "2", "3", "4"]

redis2.rpush :hoge, 5
p redis2.lrange :hoge, 0, -1 # ["1", "2", "3", "4", "5"]
```

由于数据的添加是原子处理，所以可以确认数据被正确地更新了。接下来让我们试着操作一下集合类型的数据。

```
require 'rubygems'
require 'redis'

redis = Redis.new

redis.sadd :fuga, 'm1'
redis.sadd :fuga, 'm2'
redis.sadd :fuga, 'm3'

p redis.smembers :fuga # ["m2", "m3", "m1"]
p redis.scard :fuga    # 3
p redis.spop :fuga     # 返回"m1", "m2", "m3"中的任意一个；
```

同样可以十分简单地进行处理。

在 Ruby 程序上使用（多台）

接下来让我们在多台服务器上试用 Redis。为了在多台服务器上启动 Redis，需要通过配置文件分别指定待机端口号。这次我们使用 6379、6380、6381 号端口启动 3 台服务器。

```
redis-server /etc/redis_6379.conf
redis-server /etc/redis_6380.conf
redis-server /etc/redis_6381.conf
```

程序方面，需要使用 redis/distributed 这个程序库。关于数据的分配方法，还是通过 Consistent Hashing 算法来进行的。

redis_multi_sample.rb

```
require 'rubygems'
require 'redis'
require 'redis/distributed'

redis = Redis::Distributed.new %W(redis://localhost:6379 redis://localhost:6380
redis://localhost:6381)

(1..20).each do | n|
  redis.set "key_# {n}", "value_# {n}"
end

redis.ring.nodes.each do | node|
  p "# {node.client.host}:# {node.client.port}:# {node.keys("*").join(', ')}"
end
```

数据最终的分散结果如下所示。虽然有些分散不均，但是可以确认数据已经被分散到每台服务器上了。

```
localhost:6379: key_18, key_9, key_20
localhost:6380: key_10, key_1, key_11, key_2, key_3, key_4, key_5,
                key_6, key_7,key_12, key_13, key_14, key_15, key_16, key_17
localhost:6381: key_8, key_19
```

设定复制

Redis 的复制非常简单，只需要在从数据库的 Redis 的配置文件中添加 slaveof 这个项目就可以实现了。

/etc/redis _slave. conf

```
slaveof localhost 6379
```

然后正常启动就可以了。

```
redis-server /etc/redis_master.conf
redis-server /etc/redis_slave.conf
```


那么，就让我们试着进行一下复制处理吧。首先连接6379端口的Redis（主数据库），保存数据。按照如下步骤保存两条链表类型的数据。

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

rpush data 1
1
:1

rpush data 1
2
:2

lrange data 0-1
*2
$1
1
$1
2
```

然后连接6380端口的Redis，确认一下复制是否正确地完成了。

```
$ telnet localhost 6380
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
```

```
lrange data 0-1
*2
$1
1
$1
2
```

这个例子非常简单，但能够有效确认 Redis 正确地进行了复制处理。根据 Redis 的说明文档记载，在 Amazon EC2 的实际应用中，从拥有 1000 万个键的主服务器同步到从服务器仅仅需要 21 秒就能完成[①]，确实是非常高速的复制处理。

2.3.6 各种开发语言需要用到的程序库

Redis 为 C、C#、C++、Clojure、Common Lisp、Erlang、Go、Haskell、Java、Lua、Node.js、Objective-C、Perl、PHP、Python、Ruby、Scala 等各种各样的开发语言提供了程序库[②]。如表 2-14 所示。

表 2-14 Redis 为各种开发语言提供的程序库（精选）

开发语言	程序库
Ruby	redis-rb
Perl	Redis
Java	Jedis，JRedis
C	hiredis
PHP	Predis，phpredis
Scala	scala-redis
Objective-C	objCRedis
node.js	node_redis，redis-node-client
Clojure	redis-clojure

① http://redis.shibu.jp/features.html

② http://redis.io/clients

2.4 MongoDB（面向文档的数据库）

Chapter 2

2.4.1 什么是 MongoDB

MongoDB是10gen公司开发的一款以高性能和可扩展性为特征的开源软件，它是NoSQL数据库中面向文档的数据库。

官方网站：http://www.mongodb.org/

2.4.2 为什么要使用 MongoDB

最后要介绍的NoSQL数据库就是MongoDB。MongoDB和之前介绍的memcached、Tokyo Tyrant、Redis这些键值存储在类型上略有不同，但却是一种非常实用的NoSQL数据库。

不能确定的表结构信息

关系型数据库虽然性能卓越，但是由于被设计成可以应对各种情况的通用型数据库，所以也存在一些不足之处。

例如在使用关系型数据库的时候，表结构（表中所保存的字段信息）都必须事先定义好，碰到很难定义表结构的时候就比较麻烦了。难以定义却又必须定义，这时恐怕就只能采用折中的方法：先定义最低限度的必要字段，需要的时候再添加其他字段。在这种情况下，肯定会发生添加字段等需要变更表结构的操作，势必要花费更多的工夫。但如果能接受这些额外的开销，也是个不错的解决方案。或者还可以考虑使用一些其他方法，例如事先定义一些像魔术数字一样的字段作为备用，在需要使用的时候加以利用等。

关系型数据可以在事先定义好表结构的前提下高效地处理数据。但是对于调查问卷数据和分析结果数据（通过解析日志数据得到的数据），我们很难知道哪些字段是必要的，这必然会带来反复的表结构变更操作，因此也就无需固执地非要使用关

系型数据库不可。

序列化可以解决一切问题吗

如果只是表结构的定义比较棘手的话，大家可能会觉得通过 JSON 等工具对数据进行序列化之后再保存到关系型数据库中就能解决问题。确实，若能忍受保存数据时的序列化处理以及读取数据时的反序列化处理所带来的额外开销，以及数据不易理解等问题，这也不失为一个好的解决方案。但是这种方法有可能会导致效率低下。例如，我们把如下数据通过 JSON 进行序列化，然后保存到关系型数据库的某个字段中。


```
require 'rubygems'
require 'json'

json = {
  :key1 => "value1",
  :key2 => "value2",
  :key3 => "value3"
}.to_json
```

即使存在多个键（本例中有键 1、键 2、键 3 三个），也可以顺利地通过 JSON 进行序列化，然后保存到关系型数据库中，并在读取的时候通过反序列化得到原来的散列表数据。

但是，若想要从所有数据中取得键 1 等于 xxx 的数据，应该怎么办才好呢？如果键 1 是保存在关系型数据库的字段中的话，就可以很容易地通过 SQL 读取出来。但是，那些被 JSON 序列化之后的数据却无法这样读取。因此，只能把所有数据都取出来进行反序列化，再从中抽取出符合条件的数据。如果一开始就把所有数据都取出来不仅浪费时间和资源，而且还必须对通过 SQL 取出的数据进行再次抽取。随着数据量的增大，处理所需要的时间也会越来越长。

无需定义表结构的数据库

这时就轮到 MongoDB 出场了。由于它是无表结构的数据库，所以使用 MongoDB 的时候是不需要定义表结构的。而且，由于它无需定义表结构，所以对于任何 key 都可以像关系型数据库那样进行复杂查询等操作。MongoDB 拥有比关系型数据库更快的处理速度，而且可以像关系型数据库那样通过添加索引来进行高速处理。

2.4.3 特征和用例

特征

毫无疑问，MongoDB的最大特征就是无表结构（没有必要定义表结构），，它无需像关系型数据那样定义表结构。但是，它到底是如何保存数据的呢？MongoDB在保存数据的时候会把数据和数据结构都完整地以BSON（JSON的二进制化产物）的形式保存起来，并把它作为值和特定的键进行关联。正是由于这样的设计所以它不需要定义表结构因而被称为面向文档数据库。

由于数据的处理方式不同，所以面向文档数据库的用语也发生了变化。刚开始的时候大家可能会因为用语的不同而不太适应。例如，关系型数据库中的表在面向文档数据库中称为集合（collection），关系型数据库中的记录在面向文档数据库中被称为文档（document）。表2-15所示为面向文档数据库用语。

表2-15 面向文档数据库用语

	数据库	表	记录
面向文档数据库用语	数据库	集合	文档

memcached或者Tokyo Tyrant是对关系型数据库弱点的补足，而MongoDB基本上是单独使用的，无需和关系型数据库配合使用，甚至会让人觉得只要能够通过关系型数据库进行的处理MongoDB同样也可以完成。

MongoDB跟关系型数据库不同的之处在于它无法进行JOIN查询，但它可以在标准的对象中事先嵌入（embed）其他对象，这样也能获得同样的效果。但需要注意的是，被嵌入的文档一般来说都是从嵌入元的数据中取得的。数据以数组的形式进行保存，查询起来比较容易。

但是，由于MongoDB在保存数据时需要预留出很大的空间，因此对硬盘的空间需求量呈逐渐增大的趋势[①]。

优势

无表结构是MongoDB最大的优势。由于不需要定义表结构，减少了添加字段等表结构变更所需要的开销。除此之外，它还有一些非常便利的地方。

让我们来看一个比较常见的例子：假设需要添加新字段，在这种情况下，对于

① http://www.mongodb.org/pages/viewpage.action?pageId=5079223#鏅開認者FAQ-なぜ私のデータはそんなに巨大なんですか%3F

关系型数据库来说，首先要进行表结构变更，然后在程序中针对这个新字段进行相应的修改。而 MongoDB 原本就没有定义表结构，所以只需要对程序进行相应的修改就可以了。

MongoDB 给我们带来的最大便利就是不必再去关心表结构和程序之间的一致性。使用关系型数据库时往往会发生表结构和程序之间不一致的问题，所以估计很多人在添加字段时往往只修改了程序，忘了修改表结构，从而导致出错。如果使用像 MongoDB 这样没有表结构的数据库，就不会发生类似问题了，只需保证程序的正确性即可。

MongoDB 的另一个优势，是可以灵活地指定查询条件。比如正则表达式查询，或者对数组中特定数据的判断都可以完成。

不足

MongoDB 不支持 JOIN 查询和事务处理，但实际上事务处理一般来说都是通过关系型数据库来完成的，很少会涉及到 MongoDB。虽然不能进行 JOIN 查询确实不太方便，但是也可以通过一些方法来规避。例如，可以在不需要 JOIN 查询的地方使用 MongoDB，或者是在初始设计中就避免使用 JOIN 查询等等。另外，还可以在一开始就把必要的数据全都嵌入到文档中去（详见第 3 章中的相关内容）。

还有一点需要注意的是，使用 MongoDB 创建和更新数据的时候，数据是不会实时写入到硬盘中的（参考一下 http://www.mongodb.org/pages/viewpage.action?pageId=5079223 的说明大家就能明白其中的原因了）。由于不能实时向硬盘中写入数据，所以就有可能出现数据丢失的情况。大家在使用的时候一定要谨慎。

分组处理

在介绍 MongoDB 的优势时，我们说它可以灵活地指定查询条件，但实际上它的分组处理（GROUP BY）有些特殊，使用起来并不是很容易。下面让我们通过 MongoDB 提供的 JavaScript 的 mongo shell 来实际演示一下分组处理。

```
$ mongo
MongoDB shell version:1.6.5
connecting to:test

# 创建数据；
> db.users.save({ age: 20 })
> db.users.save({ age: 24 })
> db.users.save({ age: 31 })
```

```
> db.users.save({ age: 34 })
> db.users.save({ age: 35 })
> db.users.save({ age: 28 })
> db.users.save({ age: 20 })
> db.users.save({ age: 31 })

# 这时保存的数据如下所示;
> db.users.find()
{ "_id" : ObjectId("4d755903192824438e5e2958"), "age" : 20 }
{ "_id" : ObjectId("4d75590a192824438e5e2959"), "age" : 24 }
{ "_id" : ObjectId("4d75590f192824438e5e295a"), "age" : 31 }
{ "_id" : ObjectId("4d755910192824438e5e295b"), "age" : 34 }
{ "_id" : ObjectId("4d755914192824438e5e295c"), "age" : 35 }
{ "_id" : ObjectId("4d755917192824438e5e295d"), "age" : 28 }
{ "_id" : ObjectId("4d755919192824438e5e295e"), "age" : 20 }
{ "_id" : ObjectId("4d75591c192824438e5e295f"), "age" : 31 }

# 按年龄统计;
# 使用的 SQL 文是 SELECT age, COUNT(*) from users GROUP BY age;;
> var result = db.users.group({
... key: { age :true }, # 通过 age 进行分组;
... cond: null, # 没有 WHERE 条件;
... reduce: function(obj, v) { v.count + = 1 }, # obj 是与条件匹配的数据,
v 是统计用的对象;

... initial: { count : 0 } # 初始值;
... })

# 结果可以以游标形式返回;
> result.forEach(function(x){
... printjson(x)
... })
{ "age" : 31, "count" : 2 }
{ "age" : 24, "count" : 1 }
{ "age" : 34, "count" : 1 }
{ "age" : 35, "count" : 1 }
{ "age" : 20, "count" : 2 }
{ "age" : 28, "count" : 1 }
```

MongoDB包含各种各样的处理和构成方式，虽然GROUP BY这样的处理有一些特殊，但还是希望大家能够理解和掌握，并能够扬长避短地使用MongoDB。

应用实例

目前国外使用MongoDB的案例非常多，日本国内亦呈增加趋势。

下面这些都是国内外需要处理大量数据时的实际应用实例。

- **foursquare**
- **Preferred Infrastructure**[①]
- **ameba pigg**[②]

实例

确实在字段无法确定的情况下使用 MongoDB 是非常有效的。例如问卷调查的回答数据，答案的个数是因调查问卷而不同，要确定字段非常困难。这时使用 MongoDB 就非常合适了。同理，它也非常适合用来保存分析结果数据。

- **字段不确定的情况。如调查问卷。**
- **字段可能会发生流动变化的情况。如分析结果数据。**

第 3 章将会对调查问卷的回答以及分析结果数据的存储实例进行具体的介绍。

2.4.4 安装步骤

安装

下面让我们来安装 MongoDB。虽然 MongoDB 可以通过 yum 来安装，但是由于没有可用的标准 yum 软件包，所以必须添加新的软件包（当然也可以通过程序安装）才能进行安装[③]。请创建/etc/yum.repos.d/10gen.repo 文件，并向其中写入如下内容。

/etc/yum.repos.d/10gen.repo

```
[10gen]
name = 10gen Repository
baseurl = http://downloads.mongodb.org/distros/centos/5.4/os/i386/
gpgcheck = 0
enabled = 0
```

① http://research.preferred.jp/2011/03/mongotokyo_2011_presentation/
② http://www.slideshare.net/snamura/mongo-db-couchdb20101214
③ 参考 URL：http://www.mongodb.org/display/DOCS/CentOS+and+Fedora+Packages

通过如下命令就可以一次安装完成了，本书编写时的版本是1.6.5。

```
sudo yum install mongo-stable-* --enablerepo = 10gen
```

启动

MongoDB服务器的启动方式如下所示。


```
sudo /etc/init.d/mongod start
```

启动成功后如下进程将会运行。

```
$ ps aux | grep mongodb
/usr/bin/mongod -f /etc/mongod.conf
```

启动时可以通过设定文件，或者是命令行参数来指定保存数据的目录和待机端口号等项目。下面总结一下主要的设定选项，如表2-16所示。

表2-16 MongoDB启动时的选项

选项	说明	默认值
--port	待机端口号	27017
--dbpath	保存数据文件的目录	-
--master	主数据库模式	-
--slave	从数据库模式	-
--source	复制时指定参照的主数据库	-
--only	复制时指定作为复制对象的数据库	-
-f	指定设定文件	-
--logpath	日志输入路径	标准输出
--rest	启动REST接口	-

2.4.5 动作确认

试用 mongo shell

memcached、Tokyo Tyrant、Redis都是通过telnet来确认处理过程的，而Mon-

goDB 提供了 JavaScript 的 mongo shell，让我们用 mongo shell 来确认一下它的处理过程吧。

```
$ mongo
MongoDB shell version: 1.6.5
connecting to: test # 如果没有指定,最开始会连接 test 数据库;

> show dbs # 显示数据库的命令;
admin
local

> show collections # 显示 collection 的命令;
```

此时 test 数据库并不存在，只存在 admin 数据库和 local 数据库。与关系型数据库的表相对应的 collection 中也什么都没有。下面就让我们试着来保存一些数据吧。

```
> db.users.save({ name :"sasata299" }) # 写入数据;
> db.users.find()
{ "_id" : ObjectId("4d4d3e13d8bca4f2ba32ba4f"), "name" : "sasata299" }

> show dbs
admin
local
test

> show collections
users
system.indexes
```

以上对 test 数据库中的 users collection 进行写入处理。我们并不需要事先完成创建 users collection 等操作。原本这时应该对 test 数据库进行操作，但其实这个 test 数据库本身并不存在。

虽然如此，数据依然被准确地创建出来，之后通过对数据库和 collection 的确认，test 数据库和 users collection 也已经创建出来了。能够在必要的时候自动地创建数据库和 collection，显得非常灵活。

另外，大家是否注意到 MongoDB 包含一个称为 _id 的值。它是用来识别自身的 12 字节的唯一存在的值。这个值如果不指定的话会自动生成，在 collection 中是唯一存在的。大家可以把它和关系型数据库中的连续 ID 对应起来理解。表 2-17 所示为 mongo shell 启动时的选项。

表 2-17 mongo shell 启动时的选项

选项	说明	默认值
–port	连接的端口号	27017
–host	连接的主机名	localhost

mongo shell 可以像下面这样来指定参数，这样就可以对特定的 collection 进行连接了。


```
$ mongo new
MongoDB shell version: 1.6.5
connecting to: new
```

在 Ruby 程序上使用（1 台）

接下来让我们尝试一下通过官方提供的 Ruby 程序库（mongo-ruby-driver）来使用 MongoDB。大家可以通过如下命令进行安装，本书编写时的版本是 1.2.0。

```
gem install mongo bson_ext
```

下面让我们使用这个程序库，简单地尝试一下 collection 一览、数据保存和读取等处理。通过对任意字段的数据进行保存、对任意字段进行正规表达式查询等操作，就能让大家掌握这些简单的处理。这些处理并不难，大家在使用后面将要介绍的 MongoMapper 等程序库时，将会获得更加直观的感受。

```
require 'rubygems'
require 'mongo'

connection = Mongo::Connection.new("localhost", 27017)
db = connection.db("mydb") # 连接 mydb 数据库；
p db.collection_names # [] 这时 collection 并不存在；

coll = db.collection("users")

# 写入数据；
(201..300).each {|n|  coll.insert(:name => "sasata#{n}")}

p db.collection_names # ["users", "system.indexes"]
```

```
p coll.count # 100

# 取出 name 是 sasata299的数据；
coll.find(:name => "sasata299").each {| row|  p row}
# {"_id" => BSON::ObjectId('4d4f04a061da6a5b81000064'), "name" => "sasata299"}

# 取出5条 name 以0结尾的数据；
coll.find(:name => /0$ /).limit(5).each {| row|  p row}
# {"_id" => BSON::ObjectId('4d4f076161da6a5bac00000a'), "name" => "sasata210"}
# {"_id" => BSON::ObjectId('4d4f076161da6a5bac000014'), "name" => "sasata220"}
# {"_id" => BSON::ObjectId('4d4f076161da6a5bac00001e'), "name" => "sasata230"}
# {"_id" => BSON::ObjectId('4d4f076161da6a5bac000028'), "name" => "sasata240"}
# {"_id" => BSON::ObjectId('4d4f076161da6a5bac000032'), "name" => "sasata250"}

# 删除 mydb 数据库；
connection.drop_database("mydb")
```


在 Ruby 程序上使用（多台）

接下来让我们试用多台 MongoDB。MongoDB 通过碎片（sharding）算法来进行数据分散。首先根据功能不同，启动 2 台 mongod，1 台 config 和 1 台 mongos。

```
mkdir /var/tmp/mongodb/27018
mkdir /var/tmp/mongodb/27019
mkdir /var/tmp/mongodb/config
mongod --dbpath /var/tmp/mongodb/27018 --port 27018 # mongod
mongod --dbpath /var/tmp/mongodb/27019 --port 27019 # mongod
mongod --dbpath /var/tmp/mongodb/config --port 27020 # config
mongos --configdb localhost:27020 --chunkSize 1 # mongos
```

mongod 是在后台实际进行数据保存和读取的服务器，它们可以组成一个集群来使用。mongos 是在前台直接与客户端进行交互的代理服务器。config 服务器 config 保存了 mongod 集群中 mongod、以及 mongod 存储数据范围等元信息。图 2-11 所示为 MongoDB 的 sharding 的构成。

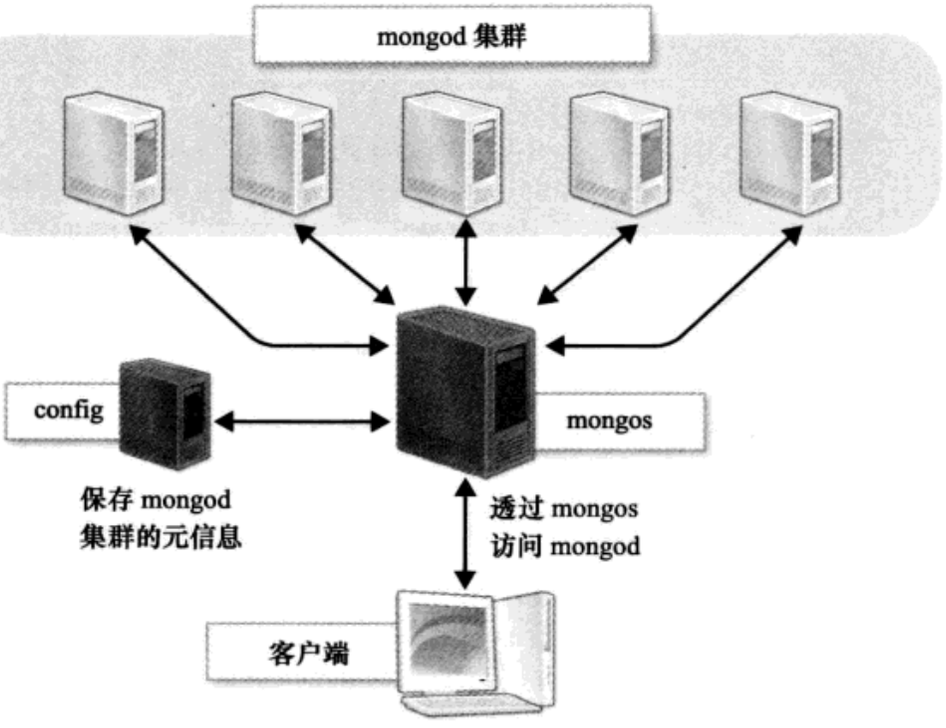


图 2-11 MongoDB 的 sharding 的构成

这时需要对 sharding 进行执行设定才能进行 sharding 处理。即需要连接 mongos 的 admin 数据库，把作为 mongod 启动的 27018 端口和 27019 端口上的 MongoDB 设定为 shard。

```
$ mongo
MongoDB shell version: 1.6.5
connecting to: test

> use admin
switched to db admin

# 把27018端口和27019端口的 mongod 作为 shard 进行注册；
> db.runCommand({ addshard : "localhost:27018", allowLocal : true })
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand({ addshard : "localhost:27019", allowLocal : true })
{ "shardAdded" : "shard0001", "ok" : 1 }

# 首先对 test 数据库进行 sharding 设定；
# 这样就可以通过任意一个 shard 管理同一个 collection 中的数据；
> db.runCommand({ enablesharding : "test" })
{ "ok" : 1 }

# 执行这个命令可以让多个 shard 管理同一个 collection 内的数据；
```

```
# 通过 user_id 字段对 test 数据库的 users collection 进行分配；
# 会出现请创建索引这样的错误消息；
> db.runCommand({ shardcollection : "test.users", key : {user_id : 1} })
{
    "ok" :0,
    "errmsg" : "please create an index over the sharding key before sharding."
}

# 移动到 test 数据库，为 user_id 字段创建索引；
> use test
switched to db test
> db.users.ensureIndex({ user_id : 1 })

# 再次回到 admin 数据库，进行 sharding 设定；
> use admin
switched to db admin
> db.runCommand({ shardcollection : "test.users", key : {user_id : 1} })
{ "collectionsharded" : "test.users", "ok" : 1 }
```

这样所有的准备工作就完成了。这些设定都会记录在 config 服务器上（通过 27020 端口启动的 MongoDB）。在当前状态下，通过 mongos 对 test 数据库的 users collection 进行的操作，都会被分散到 27018 端口或者 27019 端口的 mongod 上。

通过如下程序插入 30 万条数据，这里插入的都是 user _id 和 name 这样结构简单的数据。

```
require 'rubygems'
require 'mongo'

connection = Mongo::Connection.new("localhost", 27017)
db = connection.db("test")
coll = db.collection("users")

(1..300000).each {| n|  coll.insert(:user_id => n, :name => "user_# {n}")}
```

数据插入以后，可以通过如下程序对数据的分散情况进行确认。

```
$ mongo
MongoDB shell version:1.6.5
connecting to:test
```

```
> use admin
switched to db admin

> db.printShardingStatus() # 确认 sharding 的状态;
---Sharding Status---
     sharding version:{ "_id" :1, "version" :3 }
     shards:
          { "_id" :"shard0000", "host" :"localhost:27018" }
          { "_id" :"shard0001", "host" :"localhost:27019" }
     databases:
          { "_id" :"admin", "partitioned" :false, "primary" :"config" }
          { "_id" :"test", "partitioned" :true, "primary" :"shard0000" }
               test.users chunks:
                    { "user_id" :{$minKey :1 } }-->>{ "user_id" :1 } on :
shard0001 { "t" :2000, "i" :0 }
                    { "user_id" :1 }-->>{ "user_id" :11898 } on :shard0000 {
"t" :2000, "i" :4 }
                    { "user_id" :11898 }-->>{ "user_id" :23795 } on :shard0000 {"t" :
2000, "i" :5 }
                    { "user_id" :23795 }-->>{ "user_id" :43929 } on :shard0000 {"t" :
2000, "i" :6 }
                    { "user_id" :43929 }-->>{ "user_id" :64155 } on :shard0000 {"t" :
2000, "i" :8 }
                    { "user_id" :64155 }-->>{ "user_id" :84381 } on :shard0000 {"t" :
2000, "i" :10 }
                    { "user_id" :84381 }-->>{ "user_id" :104526 } on :shard0000{ "t" :
2000, "i" :12 }
                    { "user_id" :104526 }-->>{ "user_id" :124398 } on :
shard0000 { "t" :2000, "i" :14 }
                    { "user_id" :124398 }-->>{ "user_id" :140958 } on :
shard0000 { "t" :2000, "i" :16 }
                    { "user_id" :140958 }-->>{ "user_id" :157518 } on :
shard0000 { "t" :3000, "i" :1 }
                    { "user_id" :157518 }-->>{ "user_id" :174078 } on :
shard0001 { "t" :3000, "i" :2 }
                    { "user_id" :174078 }-->>{ "user_id" :190638 } on :
shard0001 { "t" :3000, "i" :4 }
                    { "user_id" :190638 }-->>{ "user_id" :207198 } on :
shard0001 { "t" :3000, "i" :6 }
                    { "user_id" :207198 }-->>{ "user_id" :223758 } on :
shard0001 { "t" :3000, "i" :8 }
                    { "user_id" :223758 }-->>{ "user_id" :240318 } on :
shard0001 { "t" :3000, "i" :10 }
                    { "user_id" :240318 }-->>{ "user_id" :256878 } on :
shard0001 { "t" :3000, "i" :12 }
                    { "user_id" :256878 }-->>{ "user_id" :273438 } on :
shard0001 { "t" :3000, "i" :14 }
                    { "user_id" :273438 }-->>{ "user_id" :289998 } on :
```

```
shard0001 { "t" :3000, "i" :16 }
                    { "user_id" :289998 }-->>{ "user_id" :{ $maxKey :1 } } on :
shard0001 { "t" :3000, "i" :17 }
```

MongoDB 的 sharding 是通过范围分割（Range partitioning）算法进行的。之前我们介绍了 memcached、Tokyo Tyrant、Redis 是通过 Consistent Hashing 算法，根据键的散列值，以键为单位随机分散到不同的服务器上的。MongoDB 的数据分散和它们不同，它是根据 user _id 的范围把数据分散到不同的服务器上的（从某个值到某个值分配到服务器 1 上，把某个值到某个值分配到服务器 2 上）。范围分割是以范围为单位把数据集中分散到某一台服务器上，因此比较擅长进行范围查询。表 2-18 所示为由数据分散算法造成的不同。

表 2-18 由数据分散算法造成的不同

数据的分配算法（sharding）	优点	缺点
Consistent Hashing	根据服务器的增加进行自动分配	缓存错误问题
范围分割	擅长范围查询	元信息需要在其他服务器管理

MongoDB 会以 chunk 为单位把数据集中起来进行处理。为了更加容易地确认 sharding 的结果，我们在 mongos 启动时把--chunkSize 设定为 1，实际应用中是没有必要设定这么小的值的（默认值为 200）。

关于 MongoDB 的 sharding，还有一点需要注意：数据分散只能针对 s chunk 值小于 8 的特定 shard（即被设定为 primary 的服务器）进行，也就是会存在一些不能进行数据分散的需求[①]。之前的例子也是把 user _id 截止到 157518 之前的数据全都分散到作为 primary 的 shard0000 上了。为了确认数据分散情况，这次我们插入比 30 万稍微多一点的数据。

看上去有点复杂，这里我们使用多台 MongoDB，对数据进行了分散处理。mongod 上所对应的数据信息都保存在 config 服务器上，因此那些对分散后数据的访问也可以通过 mongos 进行分散。图 2-12 所示为 sharding 操作图。

① 这里有对平衡方面的介绍：http://www.mongodb.org/pages/viewpage.action?pageId=5538048

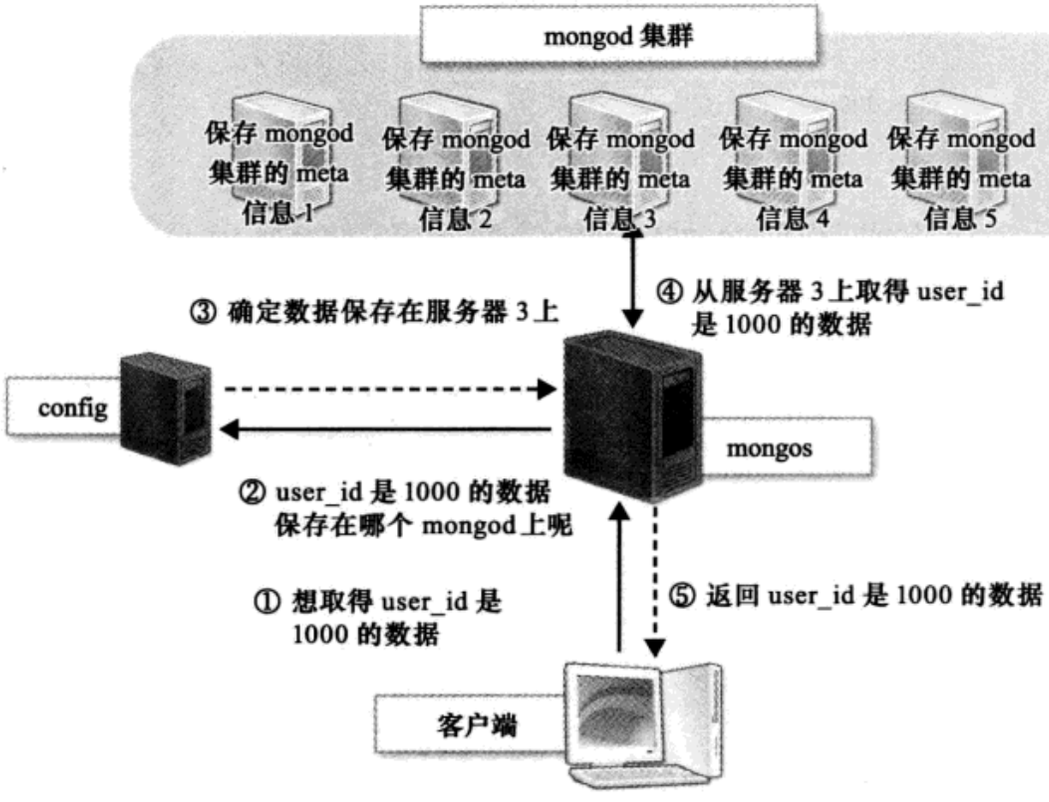


图 2-12 sharding 操作图

设定复制

下面让我们看一下 MongoDB 的复制处理。首先如下所示通过附带–master 选项启动主数据库 mongod。

```
mongod --dbpath /var/tmp/mongodb/27017 --master
```

然后通过 27018 端口附带–slave 选项启动作为从服务器的 MongoDB，通过–source选项指定主数据库的主机名和端口号，就可以开始复制了。

```
mongod --port 27018 --dbpath /var/tmp/mongodb/27018 \
       --slave --source localhost:27017
```

在这种状态下，连接 27017 端口的 MongoDB（主数据库），保存数据。

```
$ mongo
MongoDB shell version: 1.6.5
connecting to: test

> db.cities.save({ name : "Tokyo" })

> db.cities.find()
{ "_id" : ObjectId("4d749ae6296b1e9fb4d062dc"), "name" : "Tokyo" }
```


数据保存成功，接下来再确认一下数据是否已经成功复制到 27018 端口的 MongoDB（从数据库）上了。

```
$ mongo --port 27018
MongoDB shell version: 1.6.5
connecting to: 127.0.0.1: 27018/test

> db.cities.find()
{ "_id" : ObjectId("4d749ae6296b1e9fb4d062dc"), "name" : "Tokyo" }
```

这样就非常简单地完成了复制处理。这次我们尝试了主-从类型的数据复制，MongoDB从 1.6.0 版本开始已经可以对 Replica Sets 这种二元主数据库结构进行复制了。

为了使用 Replica Sets，我们首先启动 3 台 MongoDB，请不要忘记设定—replSet 选项。

```
mkdir /var/tmp/mongodb/replSet_1
mkdir /var/tmp/mongodb/replSet_2
mkdir /var/tmp/mongodb/replSet_3
mongod --replSet hoge --dbpath /var/tmp/mongodb/replSet_1
mongod --replSet hoge --dbpath /var/tmp/mongodb/replSet_2 --port 27018
mongod --replSet hoge --dbpath /var/tmp/mongodb/replSet_3 --port 27019
```

但是，这个时候会出现如下所示的报错信息，这是由于没有进行初始化，不能使用 Replica Sets 造成的。

```
replSet can't get local.system.replset config from self or any seed (EMPTYCONFIG)
```

请对 Replica Sets 进行初始化，连接任意的 mongod，执行下述处理。

```
MongoDB shell version: 1.6.5
connecting to: test

# 创建一个名为"hoge"的 Replica Sets;
> config = {_id: 'hoge', members:[
... {_id:0, host: 'localhost:27017'},
... {_id:1, host: 'localhost:27018'},
... {_id:2, host: 'localhost:27019'}]
... }
{
    "_id" : "hoge",
    "members" : [
        {
            "_id" : 0,
            "host" : "localhost:27017"
        },
        {
            "_id" : 1,
            "host" : "localhost:27018"
        },
        {
            "_id" : 2,
            "host" : "localhost:27019"
        }
    ]
}

# 进行初始化;
> rs.initiate(config)
{
    "info" : "Config now saved locally. Should come online in about a minute.",
    "ok" : 1
}

# 确认 Replica Sets 的状态;
> rs.status()
{
    "set" : "hoge",
    "date" : "Mon Apr 04 2011 14:33:28 GMT+ 0900 (JST)",
    "myState" : 1, # 主数据库;
    "members" : [
        {
            "_id" : 0,
            "name" : "localhost.localdomain2:27017",
```

```
            "health" : 1,
            "state" : 1,
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "localhost:27018",
            "health" : 1,
            "state" : 2,
            "uptime" : 105,
            "lastHeartbeat" : "Mon Apr 04 2011 14:33:27 GMT+ 0900 (JST)"
        },
        {
            "_id" : 2,
            "name" : "localhost:27019",
            "health" : 1,
            "state" : 2,
            "uptime" : 107,
            "lastHeartbeat" : "Mon Apr 04 2011 14:33:27 GMT+ 0900 (JST)"
        }
    ],
    "ok" : 1
}
```

通过 myState 的值就可以知道现在连接的 mongod 是主数据库还是从数据库（如果是 1 就是主数据库，如果是 2 就是从数据库）。下面就对主数据库的 27017 端口 mongod 进行数据写入操作（写入操作只能在主数据库中进行）。

```
$ mongo
MongoDB shell version: 1.6.5
connecting to: test

> db.messages.save({ body: "replset test" })
> db.messages.find()
{ "_id" : ObjectId("4d9ab05f32ddd93dff38115f"), "body" : "replset test" }
```

写入操作顺利完成。当然，此次变更也被复制到 27018 端口和 27019 端口的 mongod 中了，数据被成功保存。

那么，若停止主数据库的 27017 端口 mongod 将会怎样呢？Replica Sets 会在主数据库停止的时候自动进行失效转移（fail-over），把任意一个从数据库升级为主数据库。当 27017 端口的 mongod 停止时，27019 端口的 mongod 就升级成主数据库了。

我们可以通过myState来确认当前哪个mongod是主数据库。

```
$ mongo --port 27019
MongoDB shell version: 1.6.5
connecting to: test

> db.messages.update({ body: "replset test" }, { body: "replset test2" })
> db.messages.find()
{ "_id" : ObjectId("4d9ab05f32ddd93dff38115f"), "body" : "replset test2" }
```


由于27019端口的mongod升级成了主数据库，所以便可以通过它进行数据更新。这样MongoDB就能用Replica Sets简单地构建二元主数据库，提高自身的可用性。

2.4.6 各种开发语言需要用到的程序库

MongoDB为C、C#、C++、.NET、ColdFusion、Erlang、Java、JavaScript、PHP、Python、Ruby、Perl等各种开发语言提供了程序库[①]，如表2-19所示。

表2-19 MongoDB为各种开发语言提供的程序库（精选）

开发语言	程序库
Ruby	rmongo、mongo-ruby-driver
Perl	MongoDB
Java	Morphia
PHP	mongo
node.js	Mongoose
Objective C	NuMongoDB
R	RMongo
Lua	LuaMongo

2.4.7 相关工具

MongoMapper

之前我们介绍了通过Ruby使用MongoDB需要用到的程序库mongo-ruby-driver，

① http://www.mongodb.org/display/DOCSJP/Home

下面再给大家介绍一个 ActiveRecord 类的程序库 MongoMapper。通过 Rails 使用 MongoDB 的时候利用 MongoMapper 或者 Mongoid 程序库非常方便，它们全都在内部使用了 mongo-ruby-driver。

首先来安装 MongoMapper，本书编写时的版本是 0.8.6。

```
gem install mongo_mapper
```

可以像下面的程序这样进行的操作。


```
require 'rubygems'
require 'mongo_mapper'

MongoMapper.database = 'mydb'

class User
  include MongoMapper::Document

  key :name, String
  key :age, Integer
  key :created_at, Time
end

User.create(:name => "sasata299", :age => "27ab", :created_at => Time.now)
p User.first # # <User name:"sasata299", created_at:Sun Feb 06 21:11:35 UTC
2011, _id:BSON::ObjectId('4d4f0e8761da6a5c20000001'), age:27>
```

可能有人会问："无表结构的数据库怎么还要定义字段呢?"其中自有其道理。虽然不定义字段也可以使用，但是定义了字段之后就可以进行校验（validation）处理，还可以自动设定对该字段的访问（setter，getter）。另外，像上面的例子这样定义字段，还可以把 age 转换为 Integer 类型，把 created _at 转换为 Time 类型（类型转换）。

在刚才的例子中，虽然我们把字符串"27ab"赋给了 age，但由于它对应的是 Integer 类，所以只把其中的数字 27 保存起来了。

试用 NoSQL 数据库

第 2 章介绍了 memcached、Tokyo Tyrant、Redis 和 MongoDB 的特征和用例，以及简单的使用方法。对于这些 NoSQL 数据库的适用范围以及使用时的安装方法，你应该基本上掌握了吧?

但是，只是这样进行说明而没有具体的应用实例，估计大家还是无法理解实际中的使用方法，本章将会介绍各种 NoSQL 数据库的安装实例。

3.1 memcached 的具体使用实例

3.1.1 例① 关系型数据库的缓存

在下面这个例子中，memcached 用来缓存从关系型数据库中读取出的数据。也许，这是 memcached 最常见的使用实例了。

安装必要的程序库

下面我们就使用 Rails2.3.5 来创建一个实例，请根据具体情况进行安装。

```
gem install rails -v '2.3.5'
```

好啦，当我们准备使用的时候会发现，如果没有 Ruby 对应的 openssl 就会发生错误，所以需要通过安装 openssl 来把它加入到 Ruby 中。另外，由于笔者使用了 rvm[①]，所以使用的是如下命令，如果没有使用 rvm，请大家根据具体情况进行操作。

```
sudo yum install openssl-devel
rvm package install openssl
rvm remove 1.8.7
rvm install 1.8.7 --with-openssl-dir=$HOME/.rvm/usr
```

另外，这里将 MySQL 数据库置于可使用状态，其版本为 5.0.77。

```
sudo yum install mysql mysql-server mysql-devel
gem install mysql -- --with-mysql-config=/usr/bin/mysql_config
```

① http://blog.livedoor.jp/sasata299/archives/51558777.html

缓存博客文章

下面我们来看一个例子，看看用 memcached 来缓存博客数据的情况吧。另外，假设我们把博客文章保存在 articles 表中，这个表的字段包括 ID、标题、正文、创建时间、更新时间。

```
mysql> DESC articles;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
title	varchar(255)	YES		NULL	
body	text	YES		NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	

让我们试着读取一下博客文章的数据，处理流程如下：

- **确认 memcached 是否把该数据进行了缓存处理**
- **若数据被缓存，从 memcached 中读取数据**
- **若数据没有被缓存，从关系型数据库中读取数据，同时把数据缓存到 memcached 中**

首先需要确认该数据是否被缓存到 memcached 中了。数据若缓存到 memcached 中，只需要从 memcached 中读取数据就可以了，但是若数据没有被缓存到 memcached 中则需要注意，这时不仅需要从关系型数据库中读取数据，同时还要把数据缓存到 memcached 中。这样，下次就可以从 memcached 中读取数据了，同时也能减轻关系型数据库的负荷。

具体的程序如下所示：

```
require 'memcache'

def show
  cache = MemCache.new(['localhost:11211'])

  # 首先确认数据是否被缓存到 memcached 中了；
```

```
    data = cache[params[:id]]

    # 若数据没有被缓存到 memcached 中，
    # 则从关系型数据库中读取，并把数据缓存到 memcached 中；
    unless data
      data = Article.find(params[:id])
      cache.set(params[:id], data, 3600) # 失效时间是60*60，也就是1个小时；
    end
  end
```


这个例子太简单了，还不能充分说明 memcached 的优点。其实在博客中，文章是由标签、分类、留言、链接自动通知功能等各种数据组合在一起的，这些内容并非频繁更新，而且每次读取都需要花费很多时间。

例如，当用 memcached 保存标签和留言时，tags 表和 comments 表分别通过 article_id 和 article 表关联起来，如下所示：

```
mysql> DESC tags;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(255)	YES		NULL	
article_id	int(11)	NO		NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	

```
mysql> DESC comments;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
body	varchar(255)	YES		NULL	
article_id	int(11)	NO		NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	

程序代码如下所示。

```
require 'memcache'

def show
  cache = MemCache.new(['localhost:11211'])

  data = cache[params[:id]]

  # 若数据没有被缓存到 memcached 中，
  # 或者数据被更新；
  if !data || data_change?
    article = Article.find(id)
    tags = Tag.find_all_by_article_id(id).map(&:name)
    comments = Comment.find_all_by_article_id(id).map(&:body)
    @ data = {
       :title => article.title,
       :body => article.body,
       :created_at => article.created_at.strftime("%Y-%m-%d %H:%M:%S"),
       :tags => tags,
       :comments => comments
    }

    cache.set(id, @data, 86400) # 失效时间是60*60*24,也就是24个小时；
  end
end

def data_change?
  !!@change_flag
end

def update_article
  # 更新博客的文章；

  @change_flag = true
end

def update_tags(article_id)
  # 更新标签；

  @change_flag = true
end

def update_comments(article_id)
  # 更新评论；

  @change_flag = true
end
```

update_* 方法可以分别用来更新 articles 表、tags 表、comments 表。更新之后

把@change _flag 设置成 true，这时数据库会重新读入数据。除了这种情况之外，基本上不需要清除缓存，所以 expires（失效时间）会设置得比较长，如 24 小时。另外，缓存数据的粒度会在下一节进行详细介绍，articles 表、tags 表、comments 表都可以分别进行缓存，但是对于本例来说通过这种稍大的粒度进行缓存不是更好吗？

本例中使用的程序如下所示：

memcached _controller. rb

```
class MemcachedController <ApplicationController
  before_filter :init

  require 'memcache'

  def create
    (1..10).each do | n|
     Article.create!(:title => "title_# {n}", :body => "body_# {n}")
      (1 + rand(3)).times do | i|
        Tag.create!(:name => "tag_# {n}_# {i + 1}", :article_id => n)
      end
      (1 + rand(5)).times do | j|
        Comment.create!(:body => "comment_# {n}_# {j + 1}", :article_id => n)
      end
    end

    render :text => "hoge"
  end

  def show
    @ data = @ cache[params[:id]]

    if !@ data | | data_change?
      create_cache(params[:id])
    end
  end

  private

  def init
    @ cache = MemCache.new(['localhost:11211'])
  end

  def create_cache(id)
    article = Article.find(id)
    tags = Tag.find_all_by_article_id(id).map(&:name)
    comments = Comment.find_all_by_article_id(id).map(&:body)
    @ data = {
```

```
        :title => article.title,
        :body => article.body,
        :created_at => article.created_at.strftime("%Y-%m-%d %H:%M:%S"),
        :tags => tags,
        :comments => comments
      }

      @ cache.set(id, @ data, 86400)
    end

    def data_change?
      !!@ change_flag
    end

    def update_article
      # 更新博客文章;

      @ change_flag = true
    end

    def update_tags(article_id)
      # 更新标签;

      @ change_flag = true
    end

    def update_comments(article_id)
      # 更新评论;

      @ change_flag = true
    end
  end
```

memcached/show. rhtml

```
<style>
  table th{
    text-align:right;
    background-color:silver;
    padding:5px;
  }
  table td{
    padding:5px;
  }
</style>

<h2> 使用 memcached 缓存博客文章;</h2>
```

```
<table border = "1" cellspacing = "0">
  <tr>
    <th> 标题</th>
    <td> <% = @ data[:title] % >
  </tr>
  <tr>
    <th> 正文</th>
    <td> <% = @ data[:body] % >
  </tr>
  <tr>
    <th> 发表时间</th>
    <td> <% = @ data[:created_at] % >
  </tr>
  <tr>
    <th> 标签</th>
    <td> <% = @ data[:tags].join(', ') % >
  </tr>
  <tr>
    <th> 评论</th>
    <td> <% = @ data[:comments].join(', ') % >
  </tr>
</table>
```


缓存数据的粒度

但是，在 memcached 中缓存数据的时候，用多大的粒度比较好呢？是像本例这样，把所有必要的数据汇总起来通过一个键进行缓存（缓存数据的粒度比较大）好呢，还是分别对各个表（articles 表、tags 表、comments 表）中的数据通过各自的 key 进行缓存（缓存数据的粒度比较小）好呢？图 3-1 所示为缓存数据的粒度。

事实证明，将所有互相关联的数据汇集起来，通过一个键进行缓存（缓存数据的粒度比较大），这种方式更好。memcached 终究还是通过缓存来使用的，更新关系型数据库数据的时候必须要清除缓存，如果缓存数据的粒度太小，在数据更新时对于缓存的清除管理（即考虑哪部分缓存需要清除）就会非常繁琐。

例如，若要将 comments 表中的数据单独放入 memcached 内存，可通过“文章 ID：comment”这样的键来保存评论数据。评论更新之后，由于需要把“文章 ID：comment”这个键对应的数据清除掉，粒度比较小的话，创建缓存所需空间和缓存的键都会增加，要想弄清楚“什么数据通过什么键保存在什么地方”就会变得十分困难。当然，粒度小亦有一些优点，例如文章数据更新的时候评论数据不会受到影响，缓存不会被浪费等。

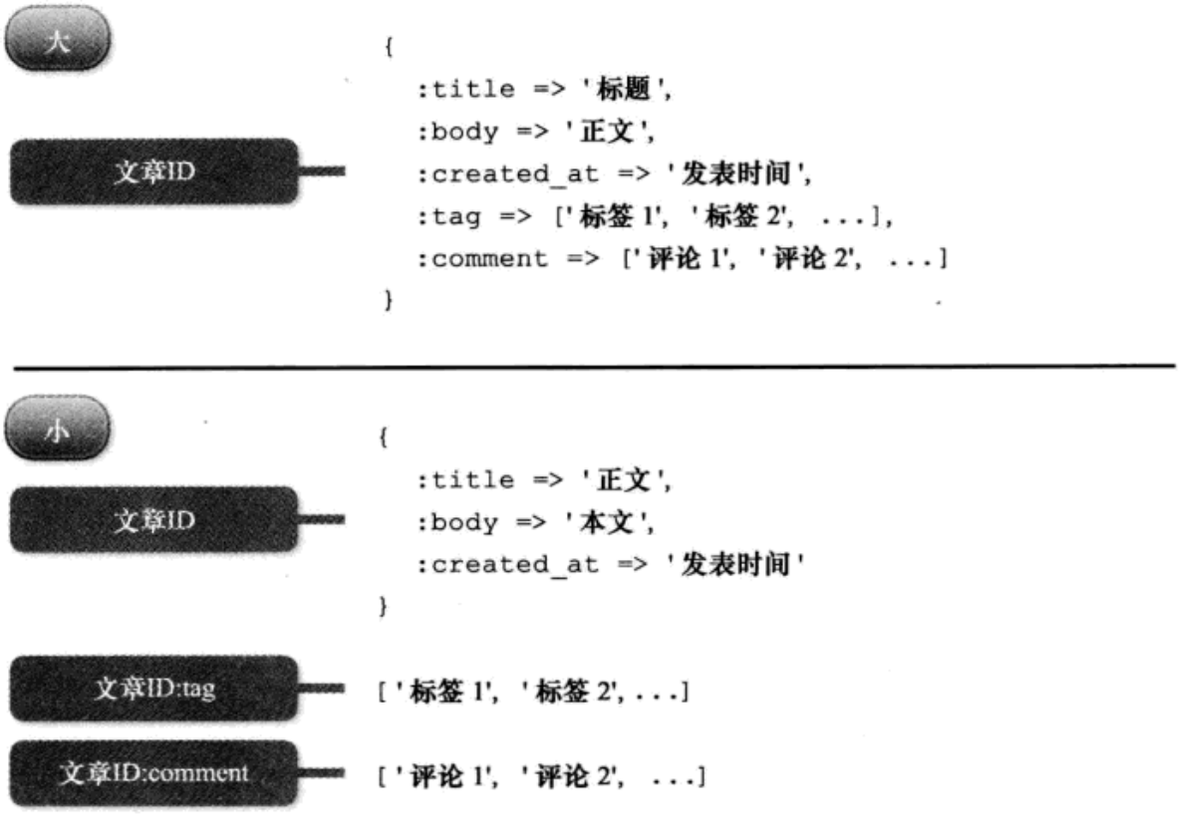


图 3-1 缓存数据的粒度

反之，若把数据汇总起来进行缓存，缓存的清除就会变得十分简单。articles 表、tags 表、comments 表不论进行什么样的更新，都可以通过清除同一个缓存来完成。另外，像本例这样把 articles 表、tags 表、comments 表等多个表中的数据汇总起来使用的时候，把数据汇集起来进行缓存操作也非常简单。但是，由于这种方法会因为各种各样的原因清除缓存，效率并不高。表 3-1 所示为各种缓存数据的优缺点。

表 3-1 各种缓存数据粒度的优缺点

缓存数据的粒度	优点	缺点
小	可以充分发挥缓存的效率	缓存的管理和清除比较繁琐
大	缓存的清除很简单	缓存的效率不高

关于缓存数据的粒度上面已经进行了说明，memcached 毕竟是通过缓存来使用的，能够恰当地清除缓存才是最重要的事情。即使通过使用 memcached 可以获得高速的响应，也并不意味着缓存中要一直保留过时的数据。

提高性能

通过 web 应用进行的处理大多都属于这种模式：从关系型数据库中读取数据，

进行某种处理然后再返回。这样的处理费时甚多，在需要高速返回响应的时候令人头疼不已。即使开始的时候响应速度很快，但随着数据的增加，也可能逐渐变慢。此时，若将数据缓存到 memcached 中，理论上就可以不通过关系型数据库而返回结果，进而提高处理速度。

3.1.2 例② 音乐视听排行网站

接下来，让我们再来看一个使用 memcached 的例子：利用 Gyao 每日音乐视听排行榜的 RSS[①] 和 YouTube 的视频查询 API[②]，尝试创建一个 Web 应用。

安装必要的程序库

这次的例子需要安装 XML 解析用的 nokogiri 程序库[③]。

```
sudo yum install libxml2 libxml2-devel libxslt libxslt-devel
gem install nokogiri
```

从 RSS 中取出排名前三十位的数据

首先从 Gyao 每日音乐视听排行榜的 RSS 中，抽取出排名前三十位的数据。若使用 Ruby，RSS 的解析可以通过使用作为标准程序库的 rss 程序库，像处理本地文件那样简单地进行处理。

```
require 'rss'

RSS_URL = 'http://gyao.yahoo.co.jp/rss/ranking/c/daily/music/'

rss = open(RSS_URL) {| f| RSS::Parser.parse(f.read)}
rss.items.each do | item|
  # 数据的格式是歌手名《歌曲名》;
  # 若数据名过长,则省略最后的右括号;
  if item.title = ~ /(.+ )「([^」]+ )/
    artist = $1
    title = $2
  end
end
```

① http://my. yahoo. co. jp/promo _jp/rsslist. html

② http://code. google. com/intl/ja/apis/youtube/developers _guide _protocol. html # Searching _for _Videos

③ http://nokogiri. org/tutorials/installng _nokogiri. html

通过这样解析 RSS，就能知道排行榜上第几位是哪位歌手唱的哪首歌曲。

查询 YouTube 的视频

接下来使用 YouTube 的视频查询 API，对之前得到的歌手名和歌曲名通过“歌手名 歌曲名”的形式进行查询。API 的查询结果以 Atom[①] 形式的 XML 返回，然后通过 nokogiri 程序库进行解析。

```
require 'rss'
require 'nokogiri'

API_URL = 'http://gdata.youtube.com/feeds/api/videos'
RSS_URL = 'http://gyao.yahoo.co.jp/rss/ranking/c/daily/music/'

@ data = []

rss = open(RSS_URL) {| f| RSS::Parser.parse(f.read)}
rss.items.each do | item|
  if item.title = ~ /(.+ )「([^」]+ )/
    artist = $1
    title = $2
  end

  _options = {
    :vq => URI.encode("# {artist} # {title}"),
    :format => 5
  }

  options = _options.map{| k,v| "# {k} = # {v}"}.join('&')
  uri = URI("# {API_URL}?# {options}")

  doc = Nokogiri::XML(uri.read)

  if doc.search('entry').blank? # 查询不到视频的时候；
    @ data<<{
      :artist => artist,
      :title => title
    }
  else # 查询到视频的时候；
    doc.search('entry').each do | entry|
      @ data<<{
        :artist => artist,
```

① Atom 是一种基于 XML 的文档格式以及基于 HTTP 的协议，它被站点和客户工具等用来聚合网络内容，包括 weblog 和新闻标题等。——译者注

```
                :title => title,
                :url => entry.xpath('media:group/media:content').first['url'],
                :type => entry.xpath('media:group/media:content').first['type'],
                :count => entry.xpath('yt:statistics').first['viewCount']
            }
            break # 查询到视频信息后就结束这个关键字的查询;
        end
    end
end
```

使用 YouTube 的视频查询 API 的时候，可以通过 format 参数来指定视频的格式。当 format 参数设定为 5 时，只取出可以作为 SWF 嵌入的视频（设定为 1 和 6 时只会取出手机可用的视频）。另外，可以通过 orderby 参数对查询结果进行排序，这里默认的是按 relevance（相关视频顺序）排序。

这里取出了每天排名 TOP30 的歌手名、歌曲名、视频的 URL 等全部必要的数据。表 3-2 简要总结一下 YouTube 的视频查询 API 的参数。

表 3-2 YouTube 的视频查询 API 的主要参数

名称	指定内容	有效值	默认值
vq	指定查询语句		
orderby	指定视频查询结果集的排序方式	relevance，published，viewCount，rating	relevance
start-index	指定结果集中第一个结果的索引		0
max-results	指定结果集的最大数量		25
alt	指定返回的格式	atom，rss，json，json-in-script	atom
format	指定视频的格式	1、6 是手机用 5 是可以作为 SWF 嵌入	

把数据保存到 memcached 中

最后，把得到的数据保存到 memcached 中（日期为键），这次我们把 expires（失效时间）设置为 24 个小时。作为键使用的日期一般都是当天的日期，当然也可以指定为任意的日期，这样就可以很方便地通过 cron（计划任务）等事前创建下一天的缓存数据了。

```
require 'memcache'

date = params[:date] || Date.today.strftime("%Y%m%d")
cache = MemCache.new

cache.set(date, @ data, 86400) # 失效时间是60*60*24,也就是24个小时;
```

这样就完成了。虽然还存在一些问题，比如一部分视频查询不到，或是查询出不相关的视频等，不过一个 Gyao 的每日音乐视听排行榜和 YouTube 视频组合的 Web 应用已经创建起来了。

本例中使用的程序如下所示：

memcached2 _controller. rb

```
class Memcached2Controller <ApplicationController
  before_filter :init

  require 'memcache'
  require 'rss'
  require 'nokogiri'

  API_URL = 'http://gdata.youtube.com/feeds/api/videos'
```

```
RSS_URL = 'http://gyao.yahoo.co.jp/rss/ranking/c/daily/music/'

def index
  @ date = params[:date] ? Date.parse(params[:date]) :Date.today
  date = @ date.strftime("%Y%m%d")

  unless @ data = @ cache[date]
    create_cache(date)
  end
end

private

def init
  @ cache = MemCache.new(['localhost:11211'])
end

def create_cache(date)
  @ data = []

  rss = open(RSS_URL) {| f| RSS::Parser.parse(f.read)}
  rss.items.each do | item|
    if item.title =~ /(.+ )「([^」]+ )/
      artist = $1
      title = $2
    end

    _options = {
      :vq => URI.encode("# {artist} # {title}"),
      :format => 5
  }

  options = _options.map{| k,v| "# {k} = # {v}"}.join('&')
  uri = URI("# {API_URL}?# {options}")

  doc = Nokogiri::XML(uri.read)

  if doc.search('entry').blank? # 查询不到视频的时候;
    @ data<<{
      :artist => artist,
      :title => title
    }
  else # 查询到视频的时候;
    doc.search('entry').each do | entry|
      @ data<<{
          :artist => artist,
          :title => title,
          :url => entry.xpath('media:group/media:content').first['url'],
```

```
            :type => entry.xpath('media:group/media:content').first['type'],
            :count => entry.xpath('yt:statistics').first['viewCount']
          }
          break
        end
      end
    end

    @ cache.set(date, @ data, 86400)
  end
end
```


memcached2/index. rhtml

```
<style>
  div.ranking{
    font-size:0.8em;
    color:gray;
    font-weight:bold;
  }
  div# video{
    margin-top:20px;
  }
  span.rank{
    font-size:1.5em;
    color:darkorange;
  }
  div.not_found{
    height:140px;
    width:303px;
    background-color:gray;
    color:# ffffff;
    text-align:center;
    font-weight:bold;
    font-size:240% ;
    padding-top:90px;
  }
  div.play_count{
    font-size:0.8em;
    color:gray;
    text-align:right;
    background-color:lightyellow;
    width:303px;
  }
</style>

<h2> <% = link_to "Gyao 点播排行榜前三十位", {:action => "index"},
```

```
{:style => "text-decoration:none"} % > </h2>

<div> "<% = @ date.strftime("%Y-%m-%d") %> "的数据</div>
<div id = "video">
<% @ data.each_with_index do | d,i| -%>
  <div class = "ranking">
    <span class = "rank"> <% = i + 1 %> 位</span> <% = d[:artist] %> -<% = d[:title]
%>
  </div>
  <% if d[:url].blank? -%>
    <div class = "not_found">
      NOT FOUND
    </div>
  <% else -%>
    <object width = "303" height = "250">
    <param name = "movie" value = "<% = d[:url] %> "> </param>
    <embed src = "<% = d[:url] %> " type = "<% = d[:type] %> " width = "303"
height = "250"> </embed>
    </object>
    <div class = "play_count">
      <span style = "margin-right:2px"> <% = number_with_delimiter(d[:count])
%> 次</span> 播放中;
    </div>
  <% end -%>
  </div>
  <div style = "margin-bottom:20px"> </div>
<% end -%>
```


与关系型数据库相异的性能

这次的例子完全没有使用关系型数据库，而只是用 memcached 来进行数据存储。虽然这样的使用方法可能有些特殊，但是由于可以从 memcached 中取出全部的数据，所以能获得非常高的响应速度。在不使用 memcached 的情况下，通过 RSS 取得排名数据，然后利用 YouTube 的视频查询 API 取得相应视频所需要的时间大约是 10 秒；但是如果使用 memcached，从第二次开始（含第二次）从 memcached 中取得数据只需要 0.005 秒就能返回响应。

本次使用的 Gyao 的每日音乐视听排行数据每天只更新一次，因此这次创建的音乐视听排行网站每天也只需要更新一次就足够了。每天读取一次数据保存到 memcached 中，并将 expires（失效时间）设定为 86400 秒（60 * 60 * 24，也就是 24 个小时），这样一来除了最开始的一次之外，数据都能从 memcached 中获得。如果通过 cron（计划任务）在前一天事先创建数据，之后的数据访问就可以全部都由 mem-

cached 来完成了。

如果数据不慎丢失，只需要 10 秒就能恢复。由此看来，在类似本例的情况下使用 memcached 确实很方便。

3.1.3 例③ 外部 API 的缓存

使用 memcached 把从外部 API 取得的数据进行缓存也是一种很好的方法。

减少开发时的过度访问

在有些情况下，特别是开发时，由于不清楚数据的结构和操作方法，往往需要多次取出相同的数据进行尝试。这时可以把外部 API 的响应缓存到 memcached 中，然后从这个 memcached 中读取数据。这样就可以减少外部 API 不必要的负荷，放心地进行重复操作了。

尽量把数据缓存到 memcached 中

另外，使用外部 API 发布后也必须注意一些情况。随着关注度增加，服务的访问量也会逐渐增加，如果每次都访问外部 API 的话，就可能超过访问次数的限制。这时在限制解除之前都无法使用外部 API。当然，即使没有限制，也应该把访问量控制在一个必要的、符合常理的最小限度之内。为了避免不必要的访问，应该尽量把外部 API 返回的响应缓存到 memcached 中。把数据缓存到 memcached 中，读取数据的速度是外部 API 无法比拟的，这也是它的优势。

有效利用 expires（失效时间）

但是，外部 API 的响应缓存，随着时间的推移也会变成过时的数据。因此，需要定期地清除缓存，这时就需要通过 memcached 的 expires 来进行管理了。例如把 expires 设定为 5 分钟，经过 5 分钟之后就会自动地清除缓存。这样就不用担心过期的缓存的问题了。

3.2 Tokyo Tyrant 的具体使用实例

例① 在每个页面显示访问量

我们继续用之前提到的博客来举例。博客会给每一篇文章分配单独的 URL，让我们试用 Tokyo Tyrant 每个页面设置一个访问计数器，来了解一下每个页面的访问数量。

获取每个页面的访问量

每个页面的访问计数器会在每次页面显示的时候更新，显示当时的访问量。虽然关系型数据库也能实现这样的功能，但是“访问计数器”只需要键值这样简单的数据结构就足够了，没有必要使用关系型数据那样复杂的结构。另外，由于还需要频繁地更新和读取数据，最好还是通过键值存储来实现高速的处理。

访问数据一旦丢失则很难恢复，因此并不适合使用 memcached 键值（因为它存在丢失数据的风险），这种情况下使用持久性的键值 Tokyo Tyrant 是更好的选择。

访问计数器的处理流程是“取得当前的访问数量，然后加 1，最后保存起来”。这个流程要经过数据读取、数据加算和数据保存三个步骤。为了保证处理的原子性，还必须在应用端进行对应处理，真的是非常麻烦。因此，这次我们使用 Tokyo Tyrant 来实现这个功能，同时使用了可以对加算进行原子处理的 addint 方法。

```
require 'tokyotyrant'

def show
  # 使用 Tokyo Tyrant 独自的协议；
  tt = TokyoTyrant::RDB.new
  tt.open('localhost', 1978)
  @ data = Marshal.load(tt.get(params[:id])) if tt.get(params[:id])

  if !@ data
```

```
    article = Article.find(params[:id])
    tags = Tag.find_all_by_article_id(params[:id]).map(&:name)
    comments = Comment.find_all_by_article_id(params[:id]).map(&:body)
    @ data = {
      :title => article.title,
      :body => article.body,
      :created_at => article.created_at.strftime("% Y-% m-% d % H:% M:% S"),
      :tags => tags,
      :comments => comments
    }

    tt.put(params[:id], Marshal.dump(@ data))
  end

  # 把访问数加1;
  tt.addint("counter:# {params[:id]}", 1)

  # 取得访问数;
  @ access_count = tt.get("counter:# {params[:id]}").unpack('i').first

  tt.close
end
```

由于 Tokyo Tyrant 只能保存字符串类型的值值，所以要通过 Marshal. dump 对数据进行序列化之后再保存起来，读取数据的时候再通过 Marshal. load 进行反序列化。使用 memcached 或 Tokyo Tyrant 的 memcached 兼容协议时可以自动地进行这些处理。不过在使用 Tokyo Tyrant 独立协议时必须明确地执行这样的处理，请特别注意。

虽然用 addint 方法读取数据时需要进行 unpack 等一些技术性处理，但这样就可以保证对访问数进行加算处理时的原子性了。

取得每个页面的访问量（incr 方法）

另外，由于 Tokyo Tyrant 支持 memcached 兼容协议，也可以用它来完成同样的处理。通过 1979 端口启动另一个的 Tokyo Tyrant，来确认 incr 方法的动作。

```
require 'tokyotyrant'

def show_mem
  mem = MemCache.new('localhost:1979')
```

```
    @data = mem.get(params[:id])

    if !@data
      article = Article.find(params[:id])
      tags = Tag.find_all_by_article_id(params[:id]).map(&:name)
      comments = Comment.find_all_by_article_id(params[:id]).map(&:body)
      @data = {
        :title => article.title,
        :body => article.body,
        :created_at => article.created_at.strftime("%Y-%m-%d %H:%M:%S"),
        :tags => tags,
        :comments => comments
      }

      mem.set(params[:id], @data)
    end

    # 取得访问量;
    # 若数据不存在需要明确地指定为0;
    @access_count = mem.incr("counter:# {params[:id]}")
    if !@access_count
      mem.set("counter:# {params[:id]}", 0)
      @access_count = mem.incr("counter:# {params[:id]}")
    end

    render :action => "show"
  end
```

这样就能实现跟刚才一样的访问计数功能了。

使用 Tokyo Tyrant 的 addint 方法时，若键不存在，会自动设置成 1，但是使用 memcached 的 incr 方法时，没有这样的自动处理。因此，需要手动保存为 0。大家可能比较在意哪种方法的效率更高，这个会在第 4 章进行说明。

本例中使用的程序如下所示：

tokyo _tyrant _controller. rb

```
class TokyoTyrant3Controller <ApplicationController
  before_filter :init
  after_filter :finalize

  require 'tokyotyrant'
  require 'memcache'

  def show
```

```
    @data = Marshal.load(@tt.get(params[:id])) if @tt.get(params[:id])

    if !@data
      create_cache(params[:id])
    end

    # 获取访问量;
    @access_count = access_count(params[:id])
  end

  def show_mem
    @data = @mem.get(params[:id])

    if !@data
      create_cache(params[:id], true)
    end

    # 获取访问量;
    @access_count = access_count(params[:id], true)

    render :action => "show"
  end

  private

  def init
    @tt = TokyoTyrant::RDB.new
    @tt.open('localhost', 1978)

    @mem = MemCache.new('localhost:1979')
  end

  def finalize
    @tt.close
  end

  def create_cache(id, mem_flag = false)
    article = Article.find(params[:id])
    tags = Tag.find_all_by_article_id(params[:id]).map(&:name)
    comments = Comment.find_all_by_article_id(params[:id]).map(&:body)
    @data = {
      :title => article.title,
      :body => article.body,
      :created_at => article.created_at.strftime("%Y-%m-%d %H:%M:%S"),
      :tags => tags,
      :comments => comments
    }
```

```
    if mem_flag
      @ mem.set(params[:id], @ data)
    else
      @ tt.put(params[:id], Marshal.dump(@ data))
    end
  end

  def access_count(id, mem_flag = false)
    _key = "counter:# {id}"

    if mem_flag
      result = @ mem.incr(_key)
      if !result
       @ mem.set(_key, 0)
       result = @ mem.incr(_key)
      end
      result
    else
      @ tt.addint(_key, 1)
      @ tt.get(_key).unpack('i').first
    end
  end
end
```

tokyo _tyrant/show. rhtml

```
<style>
  table th{
    text-align:right;
    background-color:silver;
    padding:5px;
  }
  table td{
    padding:5px;
  }
</style>

<h2> 使用 Tokyo Tyrant 完成博客文章的缓存和访问数量的记录</h2>

<table border = "1" cellspacing = "0">
  <tr>
    <th> 标题</th>
    <td> <% = @ data[:title] % > </td>
  </tr>
  <tr>
    <th> 正文</th>
    <td> <% = @ data[:body] % > </td>
  </tr>
```

```
    <tr>
      <th> 发表时间</th>
      <td> <% = @ data[:created_at] % > </td>
    </tr>
    <tr>
      <th> 标签</th>
      <td> <% = @ data[:tags].join(', ') % > </td>
    </tr>
    <tr>
      <th> 评论</th>
      <td> <% = @ data[:comments].join(', ') % > </td>
    </tr>
    <tr>
      <th> 显示次数</th>
      <td> <% = @ access_count % > </td>
    </tr>
</table>
```


按日期保存访问量

到目前为止，我们已经实现了在每个页面显示访问量的功能，接下来通过对每个页面每天的访问量进行计算，就可以知道每天的访问量了。通过“文章 ID：日期”这样包含日期形式的键就可以简单地实现这样的功能了。Tokyo Tyrant 的 addint 方法的使用步骤如下所示：

```
require 'tokyotyrant'

def show
  tt = TokyoTyrant::RDB.new
  tt.open('localhost', 1978)

  # ..snip..

  date = params[:date] | | Date.today.strftime("%Y%m%d")
  _key = "counter:# {params[:id]}:# {date}" # "文章 ID:日期"

  tt.addint(_key, 1)
  @ access_count = tt.get(_key).unpack('i').first

  tt.close
end
```

取得数据和重新排序

通过这样的处理，就可以获得文章的当日访问量了。

```
require 'tokyotyrant'

def access_log
  tt = TokyoTyrant::RDB.new
  tt.open('localhost', 1978)

  # 返回所有前方匹配的键；
  # 例如传递参数2的时候，返回文章 ID 是2的访问数；
  keys = @ tt.fwmkeys("counter:# {params[:id]}")
  keys.sort {| a,b|  a.split(/:/)[-1] <=> b.split(/:/)[-1]}.each do | k|
    p "# {k.split(/:/)[-1]} -# {@ tt.get(k).unpack('i').first}"
  end

  tt.close
end
```

结果显示如下：

```
20110313-12
20110314-23
20110315-33
20110316-17
20110317-21
```

但是，用 Tokyo Tyrant 的散列数据库时，只取得某个特定期间的数据是比较困难的，当然，这种情况因数据保持方法而异。因为只能进行前方匹配和后方匹配这种程度的查询，所以若要取得某个特定区间的数据，则需要先一次性取得某个页面对应的全部数据，然后再利用必要的部分（将不需要的部分剔除出去）。并且重新排序也需要在程序端来实现，这些处理如果使用关系型数据库的话可以很简单地实现，因此这也是在使用键值存储基础上的一种折衷方法吧。

但是，由于只是更新当天的数据，所以可以通过 cron（计划任务）等定期地把之前的数据保存到关系型数据库中。把数据保存到关系型数据库之后，就可以简单地完成具体查询和重新排序了。

本例中使用的程序如下所示：

tokyo _tyrant2 _controller. rb

```
class TokyoTyrant2Controller <ApplicationController
  before_filter :init
  after_filter :finalize

  require 'tokyotyrant'

  def show
    @ data = Marshal.load(@ tt.get(params[:id])) if @ tt.get(params[:id])

    if !@ data
      create_cache(params[:id])
    end

    # 取得访问数量;
    @ access_count = access_count(params[:id])
  end

  def access_log
    # 返回前方匹配的键;
    # 例如传递参数2的时候,返回文章 ID 是2的访问数;
    keys = @ tt.fwmkeys("counter:# {params[:id]}")
    keys.sort {| a,b|  a.split(/:/)[-1] <=> b.split(/:/)[-1]}.each do | k|
      p "# {k.split(/:/)[-1]}-# {@ tt.get(k).unpack('i').first}"
    end

    render :text => "hoge"
  end

  private

  def init
    @ tt = TokyoTyrant::RDB.new
    @ tt.open('localhost', 1978)
  end

  def finalize
    @ tt.close
  end

  def create_cache(id, mem_flag = false)
    article = Article.find(params[:id])
    tags = Tag.find_all_by_article_id(params[:id]).map(&:name)
    comments = Comment.find_all_by_article_id(params[:id]).map(&:body)
    @ data = {
      :title => article.title,
      :body => article.body,
      :created_at => article.created_at.strftime("%Y-%m-%d %H:%M:%S"),
```

```
        :tags => tags,
        :comments => comments
      }

      @ tt.put(params[:id], Marshal.dump(@ data))
    end

    def access_count(id)
      date = params[:date] | | Date.today.strftime("%Y%m%d")
      _key = "counter:# {id}:# {date}"

      @ tt.addint(_key, 1) # 把访问数量加1;
      @ tt.get(_key).unpack('i').first # 取得加算之后的数值;
    end
  end
```


简单的性能比较

之前提到了关系型数据库也能完成同样的处理，大家可能比较关心在性能方面会有怎样的差异。那么，让我们用对某个 key 进行 10000 次加 1 运算这样简单的基准测试程序（benchmark）来进行一下测试。

benchmark _controller. rb

```
class BenchmarkController <ApplicationController
  before_filter :init
  after_filter :finalize

  require 'benchmark'
  require 'tokyotyrant'

  def test
    Benchmark.bm do | x|
      x.report('MySQL') {
       1.upto(10000) do | num|
        counter = Counter.find(1) rescue Counter.create!
        counter.count + = 1
        counter.save!
       end
      }
      x.report('Tokyo Tyrant') {
       1.upto(10000) do | num|
        @ tt.addint(1, 1)
       end
```

```
      }
    end

    render :text => "hoge"
  end

  private

  def init
    @ tt = TokyoTyrant::RDB.new
    @ tt.open('localhost', 1978)
  end

  def finalize
    @ tt.close
  end
end
```

```
                  user      system      total        real
MySQL         5.160000    0.170000    5.330000   ( 27.506656)
Tokyo Tyrant  0.180000    0.100000    0.280000   ( 0.712789)
```

通过简单的基准测试可以发现，Tokyo Tyrant 的性能是关系型数据库（MySQL 的 InnoDB）的 30 倍以上。如果只是想要获得较高的处理速度，使用 Tokyo Tyrant 不失为最佳选择。

3.3 Redis 的具体应用实例

3.3.1 例① 时间线（Time Line）形式的 Web 应用

让我们来看一个 Redis 的例子，考察一下类似于 Twitter 的时间线形式（用户发言后以时间序列的形式显示）的 Web 应用吧。虽然这是一个 Python 的例子，但是日语版本的 Redis 文档介绍了该实例研究[①]，所以让我们以这个例子为参考尝试一下吧。接下来的内容，都是通过 Redis 来实现的（未使用关系型数据库）。

任务主要包括实现用户登录、登录/注销、关注、发表微博、显示微博等功能。下面让我们仔细来研究一下这些功能的实现。

用户登录

首先来看一下用户登录的部分，流程如下所示：

- **从用户登录表单中取得用户名和密码**
- **通过原子方法 incr 生成用户 ID**
- **生成认证需要的标记字符串（本例使用 32 位的随机字符串）**
- **通过 user _info 这个键以散列表形式保存以下数据**

  用户 ID => 用户名
  用户 ID => 密码的散列值
  用户 ID => 标记字符串
  标记字符串 => 用户 ID
  用户名 => 用户 ID

- **最后把标记字符串保存到 cookie 中**

虽然通过 user _info 这个键保存多个用户数据可能会随着用户数的增加而出现问题，但由于这次只是进行一些简单的处理，所以还是采用这样的设计来实现。另外，

① http://redis.shibu.jp/tutorial/index.html

这里通过判断是否存在保存标记字符串的 cookie 来确认是否登录。

具体的实现方法如下所示：

```
require 'redis'
require 'digest/sha2'

def signup
  redis = Redis.new

  if params[:commit]
    # 创建用户 ID;
    next_user_id = redis.incr :next_user_id

    auth = create_auth

    # 以"取得值:希望值"这样的散列形式保存;
    redis.hset :user_info, "# {next_user_id}:user_name", params[:user_name]
    redis.hset :user_info, "# {next_user_id}:password", Digest::SHA256.
hexdigest(params[:password])
    redis.hset :user_info, "# {next_user_id}:auth", auth
    redis.hset :user_info, "# {auth}:user_id", next_user_id
    redis.hset :user_info, "# {params[:user_name]}:user_id", next_user_id

    # ["1:user_name", "1:password", "1:auth", "jAklkoHn8VF9wtk1no3Qoh8HEiDiLhQG:
user_id", "sasata299:user_id"]
    p redis.hkeys :user_info

    store_cookie(auth)

    flash[:msg] = 登录完成;
    redirect_to :action => "index"
  end
end

# 创建32位的随机字符串;
def create_auth
  a = ('a'..'z').to_a + ('A'..'Z').to_a + ('0'..'9').to_a
  Array.new(32){a[rand(a.size)]}.join
end

def store_cookie(auth, delete_flag = false)
  # 将失效时间设定为24小时,通过 auth 这个键保存到 cookie 中;
  # 删除的时候需要设置为过去的日期;
  expire = delete_flag ? Time.now- 1.day :Time.now + 1.day
  cookies[:auth] = {
    :value => auth,
    :expires => Time.gm(expire.year, expire.month, expire.day, expire.hour,
expire.min, expire.sec)
  }
end
```

由于生成用户 ID 的 incr 方法是一个原子操作，所以可以保证其唯一性。

登录/注销

接下来让我们看一下登录/注销部分的处理。首先，注销部分的处理如下所示：

- 从 user_id 这个键保存的散列形式的数据中删除包含标记字符串的数据

  用户 ID => 用户名

  用户 ID => 密码的散列值

- 删除保存标记字符串的 cookie

具体的实现方法如下所示，通过将失效时间（expires）设定为过去的日期，即可删除 cookie。

```
require 'redis'

def logout
  redis = Redis.new
  redis.hdel :user_info, "# {@ user_id}:auth"
  redis.hdel :user_info, "# {cookies[:auth]}:user_id"
  store_cookie(nil, true) # 删除 cookie;

  flash[:msg] = 注销成功;
  redirect_to :action => "index"
end
```

删除全部包含标记字符串的数据就可以清除登录状态了。接着让我们看一下登录的部分，下面这样的处理流程是一种比较好的选择。

- 从登录表单中取得用户名和密码
- 通过用户名取得用户 ID
- 确认 Redis 中保存的密码的散列值和从登录表单中取得的散列值是否一致
- 如果一致则表明认证成功，并创建标记字符串
- 通过 user_info 这个键保存下列数据

  用户 ID => 标记字符串

  标记字符串 => 用户 ID

- 把标记字符串保存到 cookie 中

不仅仅是 Redis，对所有 NoSQL 数据库来说，类似于某个值只有通过特定值才能取得这样的情况，取得数据的方法与关系型数据库比较起来会有很多限制。

在本例中，为了判定能否登录，需要判断通过登录表单输入的密码的散列值是否正确。但是，通过登录表单只能取到用户名和密码，而无法取得用户登录时保存的密码的散列值和用户 ID，因此，还需要通过用户名取得用户 ID。

```
require 'redis'

def login
  redis = Redis.new

  if params[:commit]
    user_id = redis.hget :user_info, "# {params[:user_name]}:user_id"
    stored_password = redis.hget :user_info, "# {user_id}:password"

    if stored_password == Digest::SHA256.hexdigest(params[:password])
      auth = create_auth
      redis.hset :user_info, "# {user_id}:auth", auth
      redis.hset :user_info, "# {auth}:user_id", user_id
      store_cookie(auth)
    else
       # 输入错误时的处理;
    end

    flash[:msg] = "已登录"
    redirect_to :action => "index"
  end
end
```

判断是否登录的实现方式如下所示。如果已经登录，即标记字符串已经被保存到 cookie 中的时候，通过这个值来查询用户 ID。若可以取得用户 ID 就说明标记值是正确的，且用户已经登录。

```
require 'redis'

def login?
  redis = Redis.new

  if cookies[:auth]
    @ user_id = redis.hget :user_info, "# {cookies[:auth]}:user_id"
  end

  !@ user_id.blank?
end
```

关注

下面，让我们来看一下关注其他用户的处理。这里使用不允许重复成员出现的集合（set）类型是比较好的选择。使用集合（set）类型，可以避免已经关注的用户进行重复关注这样的问题。另外，Redis 包含很多用来进行集合运算的命令，例如使用 sinter 方法，可以简单地实现在查看用户简介的时候，显示“有○○人共同关注 TA”这样的功能。

让我们考虑一下如下的需求：

- 只有登录用户可以关注其他用户
- 打开用户个人页面的时候可以对其进行关注
- 关注某个用户的同时你将成为该用户的粉丝

具体的实现方式如下所示：

```
require 'redis'

def follow
  redis = Redis.new

  if cookies[:auth]
    @ user_id = redis.hget :user_info, "# {cookies[:auth]}:user_id"
  end

  # 在 params[:id]中保存关注用户的用户名；
  target_user_id = redis.hget :user_info, "# {params[:id]}:user_id"
  redis.sadd "# {@ user_id}:following", target_user_id
  redis.sadd "# {target_user_id}:followers", @ user_id

  render :text => "<span style = 'color:red;font- weight:bold'> 已关注</span> "
end
```

关注用户的用户 ID 列表以用户 ID：following 的方式保存，关注新的用户的时候会把该用户的 ID（target _user _id）追加到这个列表中。

不过，虽然这样可以了解“你都关注了谁”，但并不能很容易地知道“你关注的那个用户（target _user _id）还被谁关注了”。这里我们把用户 ID 也追加到被关注用户的用户 ID：following 列表中，虽然这样会花费一些功夫，但是由于可以很容易地得到“被谁关注了”的数据，因此可以对简单的数据结构进行高速的处理。

发表微博

接下来让我们看一下发表微博这部分的处理。微博是以时序列的形式显示的，从这点上考虑，最好用列表的形式保存微博数据。这样不仅可以快速地追加数据，而且可以进行原子操作。我们使用 lpush 方法在列表的表头追加新发表的微博数据。

处理流程如下所示：

- 只有登录用户可以发表微博
- 使用 incr 方法创建投稿 ID
- 通过 post：投稿 ID 这样的键，以“用户 ID | 微博发表时间 | 微博内容”这种用“|”进行分割的方式来保存投稿数据
- 向 timeline 这个键对应的列表中追加投稿 ID（包含该用户，以及被该用户关注的用户的 timeline）
- 向用户 ID：posts 这个键对应的列表中追加投稿 ID（只包含该用户的微博）

看起来有些复杂，具体的实现方式如下所示：

```
require 'redis'

def post
  redis = Redis.new

  if cookies[:auth]
    @ user_id = redis.hget :user_info, "# {cookies[:auth]}:user_id"
  end

  # 创建投稿 ID;
  next_post_id = redis.incr :next_post_id

  # 保存投稿数据;
  redis.set "post:# {next_post_id}", "# {@ user_id}| # {Time.now.to_s(:db)}|
# {params[:message]}"

  # 在包含全部用户微博的时间线中追加投稿 ID;
  redis.lpush :timeline, next_post_id

  # 取得粉丝(关注你的用户)的用户 ID;
  # 并向包含你本人在内的用户时间线中追加投稿 ID;
  followers = redis.smembers "# {@ user_id}:followers"
  followers <<@ user_id
  followers.each do | follower|
    redis.lpush "timeline:# {follower}", next_post_id
```

```
    end

    # 在你自己的微博数据中也追加投稿 ID:
    redis.lpush "# {@ user_id}:posts", next_post_id

    redirect_to :action => "index"
  end
```

由于这样下去列表的长度会逐渐增加，所以在实际使用过程中，通过 ltrim 方法来限制列表的大小是个不错的办法。这种情况下我们必须要考虑过时的投稿数据该如何处理。

显示微博

微博的显示是通过 lrange 方法来实现的。它只显示从对象微博最新的 10 条微博。在首页显示的数据，需要根据用户是否登录加以区分，在用户页面只显示该用户发表的微博。具体操作如下所示：

- **首页**
  - **已登录则显示该用户的时间线（timeline：用户 ID）**
  - **未登录则显示全部的时间线（timeline）**
- **用户页面**
  - **只显示该用户的微博（用户 ID：posts）**

让我们来看一下程序。首先是首页的处理，使用 lrange 方法取出最开始的 10 条记录。

```
require 'redis'

def index
  redis = Redis.new

  fetch_timeline
end

def fetch_timeline
  if login?
    post_ids = redis.lrange "timeline:# {@ user_id}", 0, 9
  else
```

```
    post_ids = redis.lrange :timeline, 0, 9
  end

  @ timeline = []
  post_ids.each do | post_id|
    @ timeline <<redis.get("post:# {post_id}")
  end
end
```

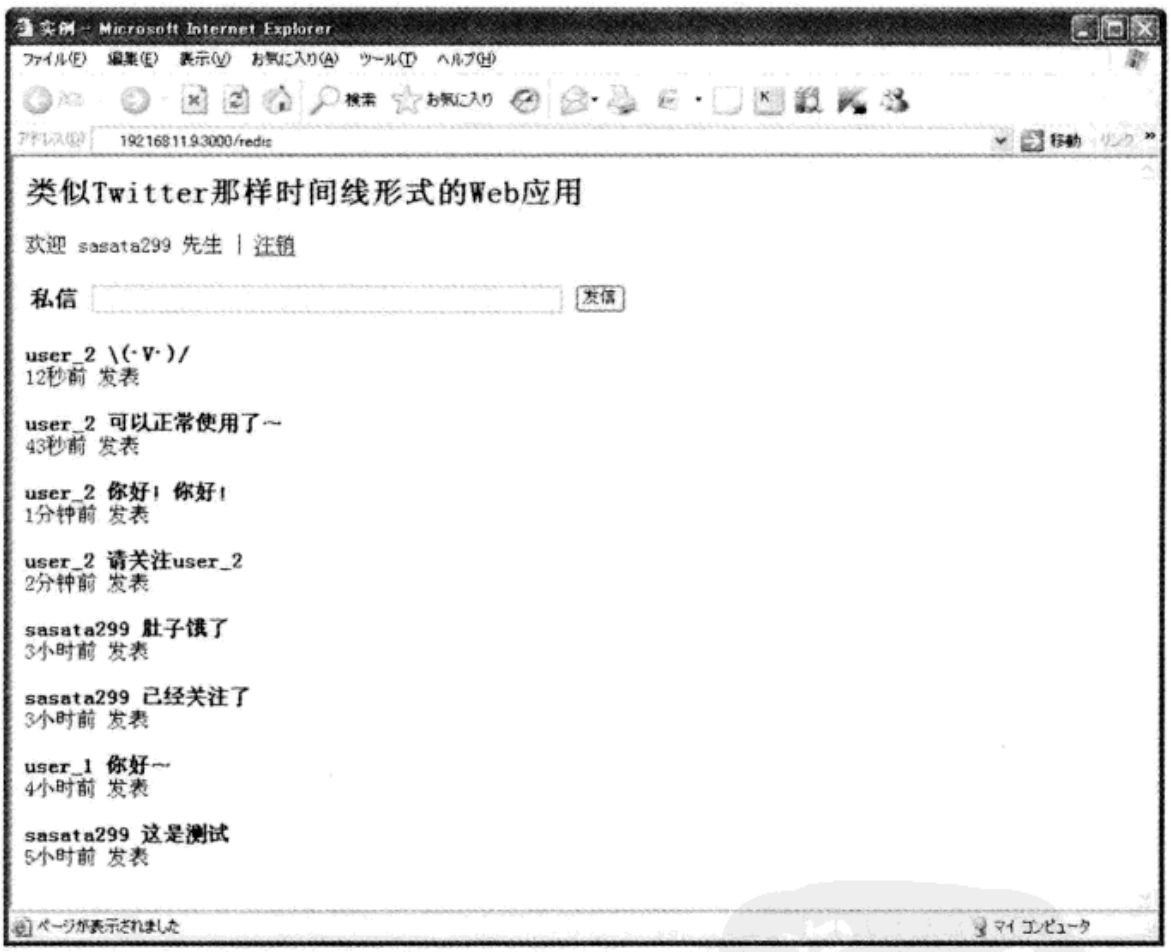
接下来，用户页面的处理如下所示，通过用户 ID：posts 取得用户的微博，并显示出来。


```
require 'redis'

def show
  redis = Redis.new

  # 在 params[:id]中保存打开页面的用户名；
  # http://192. 168. 11. 9: 3000/redis/show/sasata299 的话就是 sasata299
  user_id = redis. hget : user_info, "#{params [: id]}: user_id"

  if user_id
    if user_id == @ user_id
      @ page_title = "当前用户页面"
    else
      @ page_title = "#{params [:id]} 的页面"
    end
    fetch_user_posts (user_id)
    end

    render : action => "index"
end

def fetch_user_posts (user_id)
  post_ids = redis. lrange " #{user_id}: posts", 0, 9

  @ timeline = []
  post_ids. each do | post_id|
    @ timeline <<redis. get ("post: #{post_id}")
  end
end
```

尽管细节部分还存在各种各样的问题，但是我们只通过使用 Redis 就已经实现了像 Twitter 那样的时间线形式的 Web 应用。对于不同的数据类型，Redis 有不同的

处理风格。它可以对列表类型的数据进行高速的追加和读取处理，亦可以很容易地对集合（set）类型的数据进行集合运算（交集和并集等运算）。如果能针对这些特点恰当地加以利用，就可以简单地实现像本例这样的 Web 应用了。

Redis 的性能非常优越，这将会在第 4 章进行详细介绍。



本例中使用的程序如下所示：

redis _controller. rb

```
class RedisController <ApplicationController
  require 'redis'
  require 'digest/sha2'

  before_filter :init

  def index
    fetch_timeline
```

```
  end

  def show
    user_id = @ redis.hget :user_info, "#{params[:id]}:user_id"

    if user_id
      if user_id == @ user_id
       @ page_title = "当前用户页面;"
    else
      @ page_title = "#{params[:id]}的页面;
    end
    fetch_user_posts(user_id)
  end

  render :action => "index"
end

def post
  next_post_id = @ redis.incr :next_post_id
  @ redis.set "post:#{next_post_id}", "#{@ user_id}| #{Time.now.to_s(:db)}| #
{params[:message]}"

  @ redis.lpush :timeline, next_post_id
  # @ redis.ltrim :timeline, 0, 1000

  followers = @ redis.smembers "#{@ user_id}:followers"
  followers <<@ user_id
  followers.each do | follower|
    @ redis.lpush "timeline:#{follower}", next_post_id
  end
  @ redis.lpush "#{@ user_id}:posts", next_post_id

  redirect_to :action => "index"
end

def follow
  target_user_id = @ redis.hget :user_info, "#{params[:id]}:user_id"
  @ redis.sadd "#{@ user_id}:following", target_user_id
  @ redis.sadd "#{target_user_id}:followers", @ user_id

  render :text => "<span style = 'color:red;font- weight:bold'> 关注成功</span> "
end

def signup
  if params[:commit]
    next_user_id = @ redis.incr :next_user_id
```

```
      auth = create_auth

      # 以"取得值:希望值"这样的散列形式保存;
      @ redis.hset :user_info, "#{next_user_id}:user_name", params[:user_name]
      @ redis.hset :user_info, "#{next_user_id}:password", Digest::SHA256.
hexdigest(params[:password])
      @ redis.hset :user_info, "#{next_user_id}:auth", auth
      @ redis.hset :user_info, "#{auth}:user_id", next_user_id
      @ redis.hset :user_info, "#{params[:user_name]}:user_id", next_user_id

      store_cookie(auth)

      flash[:msg] = "保存成功"
      redirect_to :action => "index"
    end
  end

  def login
    if params[:commit]
      user_id = @ redis.hget :user_info, "#{params[:user_name]}:user_id"
      stored_password = @ redis.hget :user_info, "#{user_id}:password"

      if stored_password == Digest::SHA256.hexdigest(params[:password])
        auth = create_auth
        @ redis.hset :user_info, "#{user_id}:auth", auth
        @ redis.hset :user_info, "#{auth}:user_id", user_id
        store_cookie(auth)
      else
        # 输入错误时的处理;
      end

      flash[:msg] = "登录成功"
      redirect_to :action => "index"
    end
  end

  def logout
    @ redis.hdel :user_info, "#{@ user_id}:auth"
    @ redis.hdel :user_info, "#{cookies[:auth]}:user_id"
    store_cookie(nil, true)

    flash[:msg] = "注销"
    redirect_to :action => "index"
  end

  private

  def init
```

```
    @ redis = Redis.new

    if cookies[:auth]
      @ user_id = @ redis.hget :user_info, "#{cookies[:auth]}:user_id"
      @ user_name = @ redis.hget :user_info, "#{@ user_id}:user_name"
    end
  end

  def login?
    !@ user_id.blank?
  end

  def fetch_timeline
    if login?
      post_ids = @ redis.lrange "timeline:#{@ user_id}", 0, 9
    else
      post_ids = @ redis.lrange :timeline, 0, 9
    end

    @ timeline = []
    post_ids.each do | post_id|
      @ timeline <<@ redis.get("post:#{post_id}")
    end
  end

  def fetch_user_posts(user_id)
    post_ids = @ redis.lrange "#{user_id}:posts", 0, 9

    @ timeline = []
    post_ids.each do | post_id|
      @ timeline <<@ redis.get("post:#{post_id}")
    end
  end

  def create_auth
    a = ('a'..'z').to_a + ('A'..'Z').to_a + ('0'..'9').to_a
    Array.new(32){a[rand(a.size)]}.join
  end

  def store_cookie(auth, delete_flag = false)
    # 把失效时间设定为24小时,通过 auth 这个键保存到 cookie 中;
    # 删除的时候需要设置为过去的日期;
    expire = delete_flag ? Time.now- 1.day :Time.now + 1.day
    cookies[:auth] = {
      :value => auth,
      :expires => Time.gm(expire.year, expire.month, expire.day, expire.hour,
expire.min, expire.sec)
          }
    end
  end
```

redis/index. rhtml

```
<style>
  div.msg{
    background-color:lightblue;
    text-align:center;
    margin-bottom:10px;
  }
  div.timeline_user{
    color:gray;
    float:left;
  }
  div.timeline_user a{
    color:# 2276BB;
    font-weight:bold;
    text-decoration:none;
  }
  div.timeline_body{
    padding-left:5px;
    float:left;
  }
  div.timeline_date{
    color:gray;
    font-size:0.8em;
    padding-bottom:15px;
  }
  div.info{
    color:gray;
    font-size:0.8em;
    margin-bottom:10px;
  }
  div# page_title{
    color:darkolivegreen;
    border-top:1px solid gray;
    border-bottom:1px solid gray;
    padding:5px;
    font-weight:bold;
    margin-bottom:20px;
  }
  span# follow{
    font-size:0.8em;
  }
</style>

<h2> <% = link_to "类似 Twitter 那样时间线形式的 Web 应用", {:action
=> "index"}, {:style => "text- decoration:none"} % > </h2>

<% if flash[:msg] -% >
  <div class = "msg"> <% = flash[:msg] % > </div>
```

```
<% end -% >

<div class = "info">
  <% if @ user_id -% >
    欢迎 <% = @ user_name % > さん | < % = link_to "注销", :action =>
"logout" % >
  <% else -% >
    <% = link_to "用户登录", :action => "signup" % > | < % = link_to "登录",
:action => "login" % >
  <% end -% >
</div>

<% if @ user_id -% >
  <% form_tag :action => "post" do -% >
    <span style = "padding-right:3px"> 私信</span>
    < % = text_field_tag :message, nil, :size => 60 % >
    < % = submit_tag '发信' % > </th>
    <% end -% >
<% end -% >

<% if @ page_title -% >
  <div id = "page_title">
    < % = @ page_title % >
    <% if @ page_title ! = "当前用户页面" && @ user_id -% >
      <span id = "follow">
        <% user_id = @redis.hget :user_info, "#{params[:id]}:user_id" -% >
        <% if @redis.smembers("#{@user_id}:following").include?(user_id)-% >
          <span style = "color:pink"> 已关注</span>
        <% else -% >
          < % = link_to_remote "关注", :update => "follow", :url => {:action
=> "follow", :id => params[:id]} % >
        <% end -% >
      </span>
    <% end -% >
  </div>
<% end -% >

<% @ timeline.each do | tl| -% >
  <% user_id, created_at, message = tl.split(/\ | /) -% >
  <%
    sec = Time.now- Time.parse(created_at)
    if sec < 60
      diff = "#{sec.round}秒前"
    elsif sec < 60 * 60
      diff = "#{(sec / 60).round}分前"
    elsif sec < 60 * 60 * 24
      diff = "#{(sec / (60 * 60)).round}小时前"
    else
      diff = "#{(sec / (60 * 60 * 24)).round}日前"
    end
```

```
  -% >
  <% user_name = @ redis.hget :user_info, "#{user_id}:user_name"-% >
  <div class = "timeline_user"> < % = link_to user_name, :action => "show", :id
=> user_name % > </div>
  <div class = "timeline_body"> <% = message -% > </div>
  <div style = "clear:left"> </div>
  <div class = "timeline_date"> <% = diff % > 投稿成功</div>
<% end -% >
```


3.3.2 例② 查询历史记录

让我们来看另外一个例子，创建一个针对 ATND（举办活动的辅助工具[①]）的查询 Web 应用，然后使用 Redis 为它追加一个查询历史记录的功能。因为历史记录也是时序列类型的数据，所以我们使用和之前一样的列表类型来保存数据。

通过 ATND 的 API 取得活动信息

那么，首先我们使用 ATND 的 API[②] 来创建一个 ATND 的查询应用。这次的例子我们要查询那些通过关键字和 Twitter ID[③] 登录到 ATND 中的活动数据。由于返回的是 XML 类型的数据，我们需要使用在介绍 memcached 时提到的 nokogiri 程序库来进行解析。

另外，由于活动简介的内容比较多，所以只显示开头的 200 个文字，这样就可以通过文字单位对指定 KCODE 的活动概要进行分割了。

```
require 'nokogiri'
require 'open-uri'

KCODE = 'u'

ATND_API_URL = 'http://api.atnd.org/events/'

def index
  if params[:k] | | params[:t]
```

① http://atnd.org/

② http://api.atnd.org/

③ 只有通过 Twitter ID 登录 ATND，或者账户与 Twitter ID 进行关联的场合才作为对象。

```
    uri = URI("#{ATND_API_URL}?#{construct_search_query}")
    doc = Nokogiri::XML(uri.read)

    @ data = []
    if doc.search('event')
      doc.search('event').each do | event|
       @ data <<{
         :title => event.xpath('title').text,
         :url => event.xpath('event_url').text,
         :description => event.xpath('description').text.split(//)[0..199].
join(''),
         :started_at => event.xpath('started_at').text
       }
      end
    end
  end
end

def construct_search_query
  _options = {
    :keyword => params[:k],
    :twitter_id => params[:t],
    :count => 20
  }

  _options.map{| k,v|  "#{k} = #{v}"}.join("&")
end
```


追加查询历史记录功能

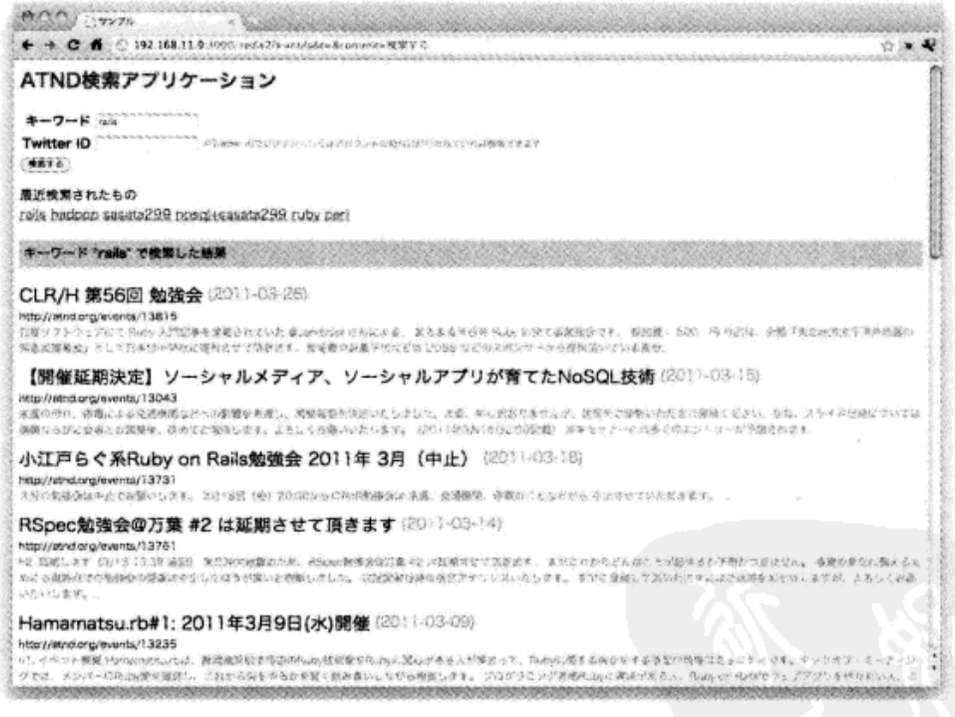
接下来让我们尝试使用 Redis 为这个 ATND 查询应用追加查询历史记录的功能。这个功能需要把访问 ATND 的 API 的数据作为历史记录全部保存起来。首先，查询历史记录保存部分的程序如下所示，关键字和 Twitter ID 与之前一样通过“|”分割进行保存。

```
require 'redis'

def store_history
  redis = Redis.new

  redis.lpush :history, "#{params[:k]}| #{params[:t]}"
end
```

然后是显示查询历史记录的部分，这里只显示 10 条最近的历史记录。

```
require 'redis'

def get_recent_history
  redis = Redis.new

  @ histories = redis.lrange :history, 0, 9
end
```


这样 ATND 查询应用就完成了。仅使用 Redis 就能非常简单地把查询历史记录的功能追加进来。当需要高速响应，并且操作对象是历史记录这样的列表类型数据的时候，没有必要去刻意使用关系型数据库，使用 Redis 就足够了。



本例中使用的程序如下所示：

redis2 _controller. rb

```
class Redis2Controller <ApplicationController
  require 'redis'
  require 'nokogiri'
  require 'open-uri'
```

```
  KCODE = 'u'

  before_filter :init
  ATND_API_URL = 'http://api.atnd.org/events/'

  def index
    if params[:k] || params[:t]
      uri = URI("#{ATND_API_URL}?#{construct_search_query}")

      doc = Nokogiri::XML(uri.read)

      @data = []
      if doc.search('event')
       doc.search('event').each do |event|
        @data << {
          :title => event.xpath('title').text,
          :url => event.xpath('event_url').text,
          :description => event.xpath('description').text.split(//)[0..199].
join(''),
          :started_at => event.xpath('started_at').text
        }
       end
      end

      store_history
    end

    get_recent_history
  end

  private

  def init
    @redis = Redis.new
  end

  def store_history
    @redis.lpush :history, "#{params[:k]}| #{params[:t]}"
  end

  def get_recent_history
    @histories = @redis.lrange :history, 0, 9
  end

  def construct_search_query
    _options = {
      :keyword => params[:k],
      :twitter_id => params[:t],
```

```
        :count => 20
      }
      _options.map{| k,v|  "#{k} = #{v}"}.join("&")
    end
end
```

redis2/index. rhtml

```
<style>
  table th{
    text-align:right;
  }
  div.msg{
    background-color:lightblue;
    text-align:center;
    margin-bottom:10px;
  }
  span.red{
    color:red;
    font-size:0.7em;
  }
  div.info{
    color:gray;
    font-size:0.8em;
    margin-bottom:10px;
  }
  div.event_title{
    font-size:1.4em;
    float:left;
  }
  div.event_date{
    font-size:1.2em;
    color:# b48a76;
    padding-left:5px;
    float:left;
  }
  div.event_url{
    font-size:0.8em;
  }
  div.event_description{
    color:gray;
    font-size:0.8em;
    margin-bottom:10px;
  }
  div.condition{
    padding:5px;
```

```
    background-color:silver;
    margin-bottom:20px;
  }
  div# history{
    margin-bottom:20px;
  }
</style>
<h2> <% = link_to "ATND查询应用", {:action => "index"}, {:style =>
"text-decoration:none"} % > </h2>

<% if flash[:msg] -% >
  <div class = "msg"> <% = flash[:msg] % > </div>
<% end -% >

<% form_tag({}, {:method => "get"}) do -% >
  <table>
    <tr>
      <th> 关键字</th>
      <td> <% = text_field_tag :k, params[:k] -% > </td>
      <td> </td>
    </tr>
    <tr>
      <th> Twitter ID</th>
      <td> <% = text_field_tag :t, params[:t] -% > </td>
      <td> <span class = "red"> ※通过 Twitter ID登录,或者对用户进行关联时可以进行
查询
    </tr>
  </table>
  <% = submit_tag '查询' % > </th>
<% end -% >

<% if !@ histories.blank? -% >
  <div id = "history">
    <div> 最近查询过的数据</div>
    <% @ histories.each do | history| -% >
      <% keyword, twitter_id = history.split(/\ | /) -% >

      <% if !keyword.blank? && !twitter_id.blank? -% >
       <% body = "#{keyword}+ #{twitter_id}" -% >
      <% else -% >
       <% if keyword.blank? -% >
        <% body = "#{twitter_id}" -% >
       <% elsif twitter_id.blank? -% >
        <% body = "#{keyword}" -% >
       <% end -% >
      <% end -% >

      <% = link_to body, :action => "index", :k => keyword, :t => twitter_id % >
```

```
    <% end -% >
  </div>
<% end -% >

<% if !@ data.blank? -% >
  <div class = "condition">
    <% list = [] -%>
    <% list <<%Q!关键字"<span style = 'font-weight:bold'> #{params[:k]}</
span> "! if !params[:k].blank? %>
    <% list <<%Q!Twitter_id "<span style = 'font- weight:bold'> #{params[:t]}
</span> "! if !params[:t].blank? %>
    <% = list.join(", ") %> 查询结果
  </div>
  <% @ data.each do | d| -%>
    <div class = "event_title"> <% = d[:title] %> </div>
    <div class = "event_date"> (<% = Date.parse(d[:started_at]).strftime
("%Y-%m-%d")%> )</div>
    <div style = "clear:left"> </div>
    <div class = "event_url"<% = link_to d[:url], d[:url], :popup => true %>
</div>
    <div class = "event_description"> <% = strip_tags(d[:description]) %> ...
</div>
  <% end -%>
<% end -%>
```

3.4 MongoDB的具体使用实例

3.4.1 例① 问卷调查数据的保存

使用MongoDB

让我们来看一个使用MongoDB的具体实例，尝试处理问卷调查数据。我们通过使用第2章最后介绍的MongoMapper程序库来实现这些处理。首先，为操作问卷调查数据的集合answers定义一个对象模型。在类定义中除了需要引入MongoMapper::Document之外，并没有什么特别的不同之处。

```
class Answer
  include MongoMapper::Document

  key :user_id, Integer, :required => true
  key :enquete_id, Integer, :required => true # 问卷调查ID
  timestamps!
end
```

虽然就像第2章说明的那样，使用MongoDB并不需要定义具体的字段，但是本例中还是明确地定义了user _id和enquete _id。这是因为定义字段能带来一定的好处，例如可以进行validation校验（本例中的ser _id和enquete _id是必须项目），可以设定访问，可以和对应的类（本例中用到的是Integer类）进行映射等。当然，有时候定义字段只是单纯地为了便于理解和管理。

对于本例中的问卷调查数据来说，user _id和enquete _id是必须项目。这种情况下，明确地定义字段是个非常好的选择。

虽然只是非常简单的设计，但是这样就可以用来保存任意形式的回答数据了。在同一个字段中，可以同时保存字符串数据、多行字符串数据、数值数据、数组数据等各种各样的数据。

另外，本例中只使用了Integer类，大家还可以使用其他各种各样的类。有关类

的信息如表 3-3 所示：

表 3-3 可以通过 MongoMapper 匹配使用的类

类	效果	备考
String	对字符串类型进行匹配	nil（无值）还是当作 nil（无值）来处理
Integer	对数字类型进行匹配	
Date	对 Date 类型进行匹配	
Time	对 Time 类型进行匹配	
Array	对数组类型进行匹配	如果没有值的话就当作是空数组来处理
Boolean	对 Boolean 类型进行匹配	nil（无值）还是当作 nil（无值）来处理
Object	不进行匹配（直接使用原值）	

可以保存各种类型的数据

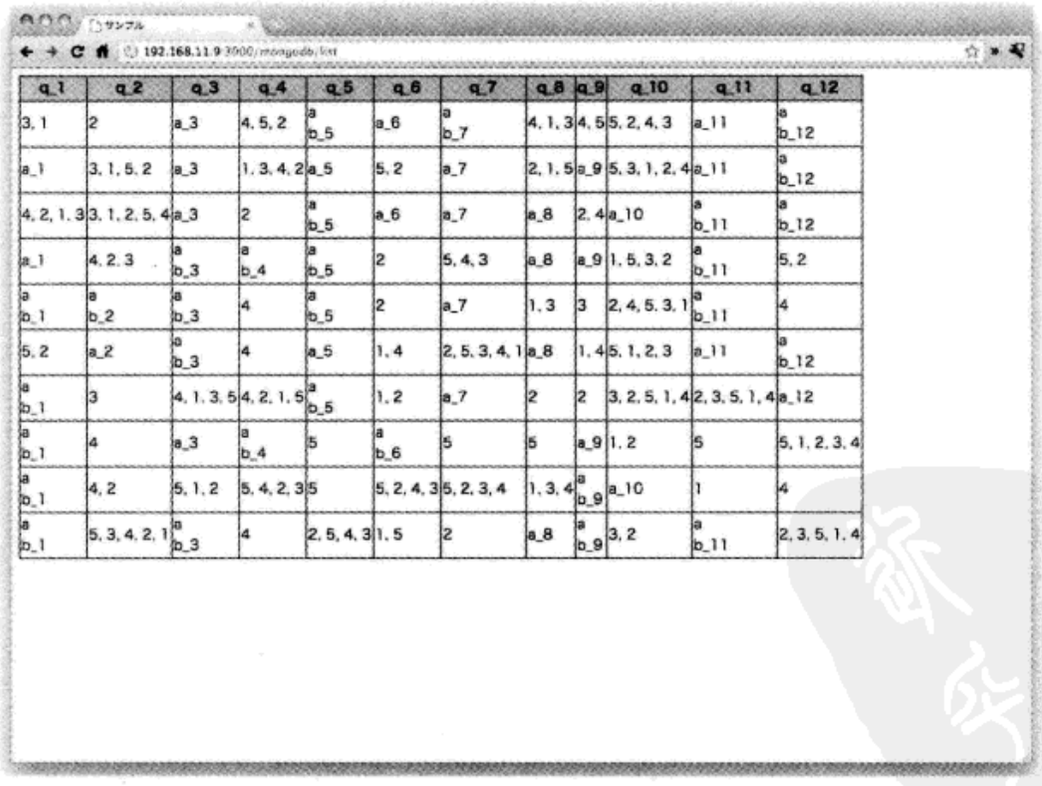
首先让我们追加一些合适的数据，来确认一下是不是任何类型的数据都可以保存。作为测试，我们假定有 12 个字段，尝试保存 25%的字符串类型数据，25%的多行字符串类型数据，25%的数字类型数据，25%的数组类型数据。实际使用的程序代码如下所示：

```
10.times do
  condition = {}
  (1..12).each do | n|
    if rand(100) %  4 == 1
      condition["q_#{n}"] = "a_#{n}"
    elsif rand(100) %  4 == 2
      condition["q_#{n}"] = "a\ nb_#{n}"
    elsif rand(100) %  4 == 3
      condition["q_#{n}"] = 1 +  rand(5)
    else
      list = [1, 2, 3, 4, 5]
      condition["q_#{n}"] = list.sort_by{rand}[0..rand(5)]
    end
  end

  Answer.create!(
    condition.merge(
      :user_id => 1,
      :enquete_id => 1
    )
  )
end
```

创建问卷调查数据的时候，user _id 和 enquete _id 通过其他途径来指定。由于之前把 user _id 和 enquete _id 设定为必须数据，所以如果 enquete _id 为空，创建数据的时候就会出错。

```
# 如果不指定 enquete_id 就会发生错误
MongoMapper::DocumentNotValid (Validation failed:Enquete can't be empty):
Answer.create!(
  :user_id => 1
)
```


让我们确认一下创建好的数据，确实在同一个字段中，同时保存了字符串类型、多行字符串类型、数字类型和数组类型等各种各样的数据。由于使用的是没有数据结构的 MongoDB 数据库，也可以非常容易地实现这样的处理。



q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12
3, 1	2	a_3	4, 5, 2	a b_5	a_6	a b_7	4, 1, 3	4, 5	5, 2, 4, 3	a_11	a b_12
a_1	3, 1, 5, 2	a_3	1, 3, 4, 2	a_5	5, 2	a_7	2, 1, 5	a_9	5, 3, 1, 2, 4	a_11	a b_12
4, 2, 1, 3	3, 1, 2, 5, 4	a_3	2	a b_5	a_6	a_7	a_8	2, 4	a_10	a b_11	a b_12
a_1	4, 2, 3	a b_3	a b_4	a b_5	2	5, 4, 3	a_8	a_9	1, 5, 3, 2	a b_11	5, 2
a b_1	a b_2	a b_3	4	a b_5	2	a_7	1, 3	3	2, 4, 5, 3, 1	a b_11	4
5, 2	a_2	a b_3	4	a_5	1, 4	2, 5, 3, 4, 1	a_8	1, 4	5, 1, 2, 3	a_11	a b_12
a b_1	3	4, 1, 3, 5	4, 2, 1, 5	a b_5	1, 2	a_7	2	2	3, 2, 5, 1, 4	2, 3, 5, 1, 4	a_12
a b_1	4	a_3	a b_4	5	a b_6	5	5	a_9	1, 2	5	5, 1, 2, 3, 4
a b_1	4, 2	5, 1, 2	5, 4, 2, 3	5	5, 2, 4, 3	5, 2, 3, 4	1, 3, 4	a b_9	a_10	1	4
a b_1	5, 3, 4, 2, 1	a b_3	4	2, 5, 4, 3	1, 5	2	a_8	a b_9	3, 2	a b_11	2, 3, 5, 1, 4

将集合嵌入到另一个集合中

那么让我们来看一下问卷调查数据的保存处理吧。在保存问卷调查数据时，会根据字段的不同来设定是必须回答还是可以选择回答。另外，在显示问卷调查的时

候，问题和回答方式的信息是必须的。即该问题是必须回答还是可以选择回答，回答方式是文本框输入还是单选按钮这样的信息都必须要进行设定，这也可以通过 MongoDB 来完成。例如，可以使用集合 enquetes 和集合 questions，用集合 enquetes 来保存问卷调查的基本数据（问卷调查的时间和调查对象的人数等），用集合 questions 来保存问卷调查的问题数据（问题的回答方式和选项信息等）。

如果使用关系型数据库，在 questions 表中保存关联 enquetes 表用的 enquete _id，通过 JOIN 进行一次查询就可以把 enquetes 表和 questions 表中的数据一起取出来，非常简单。而 MongoDB 不支持 JOIN 查询，即使在集合 questions 中保存了 enquete _id，也无法通过一次查询得到全部的数据[①]。这时，可以使用 MongoDB 的 embed（嵌入）方法，也就是在一个集合中嵌入另外一个集合的数据，来达到一次查询的效果。

embed 的程序如下所示。首先，定义集合 enquetes 对应的类（模型），这里要注意的是，需要指定集合 enquetes 中包含多个 questions 集合。与之前不同的是，需要在集合 questions 对应的类（模型）中引入（include）MongoMapper::EmbeddedDocument，即声明这个集合是被嵌入的集合。

```
class Enquete
  include MongoMapper::Document

  key :name, String, :required => true
  timestamps!

  # 嵌入多个 questions 集合
  many :questions
end
class Question
  include MongoMapper::EmbeddedDocument

  key :body, Object, :required => true
  key :type, String, :required => true
  key :choices, Array # 回答的选项
  key :required_column, Boolean, :default => false
end
```

本例中，通过上述设定，就可以在 enquetes 集合中指定作为 questions 字段的 questions 集合，并保存下来。由于 questions 集合被嵌入到 enquetes 集合中，这样就

① 首先需要查询集合 enquetes，然后通过 enquete _id 对集合 questions 进行查询。

可以通过一次查询把 enquetes 集合和 questions 集合中的数据全都取出来了。但是，必须要通过嵌入集合（本例中的 enquetes 集合）才能取得被嵌入其中的集合（本例中的 questions 集合）。请注意，不通过 enquetes 集合是无法取得 questions 集合的。同时被嵌入的集合无法使用 find 等方法。

让我们使用这些方法实际来创建一个问卷调查看一下吧。我们把问题 1 设定成文本框，问题 2 设定成单选按钮，问题 3 设定成复选框，问题 4 设定成下拉框，问题 5 设定成文本域，同时还需要设定问题和各种选项，以及是否必须回答等内容。

```
def create_question
  enquete = Enquete.new(
    :name => "关于体育运动和酒的调查"
  )

  question_list = []
  type_h = {
    :q_1 => 'text',
    :q_2 => 'checkbox',
    :q_3 => 'radio',
    :q_4 => 'pulldown',
    :q_5 => 'textarea'
  }
  body_h = {
    :q_1 => '您喜欢的体育运动',
    :q_2 => '您经常饮用的酒的种类',
    :q_3 => '您的性别',
    :q_4 => '您的年龄',
    :q_5 => '您是通过什么途径知道这次问卷调查的'
  }
  choices_h = {
    :q_1 => nil,
    :q_2 => %w(
      不喝酒
      啤酒
      日本酒
      烧酒
      红葡萄酒
      白葡萄酒
      鸡尾酒
      威士忌;
    ),
    :q_3 => %w(男性 女性),
    :q_4 => %w(20~29岁  30~39岁  40~49岁  50~59岁  60岁以上),
    :q_5 => nil
  }
```

```
    require_h = {
      :q_1 => true,
      :q_2 => true,
      :q_3 => true,
      :q_4 => true,
      :q_5 => false
    }
    (1..5).each do | n|
      question_list <<Question.new(
        :question_number => "q_#{n}",
        :body => body_h["q_#{n}".to_sym],
        :type => type_h["q_#{n}".to_sym],
        :choices => choices_h["q_#{n}".to_sym],
        :required_column => require_h["q_#{n}".to_sym]
      )
    end

    enquete.questions = question_list
    enquete.save!

    render :text => "hoge"
  end
```

保存回答数据

接下来，让我们实际保存一下回答的数据。我们需要确认一下该问题是否是必须回答的，如果必须回答而没有回答，就会显示一条错误信息。

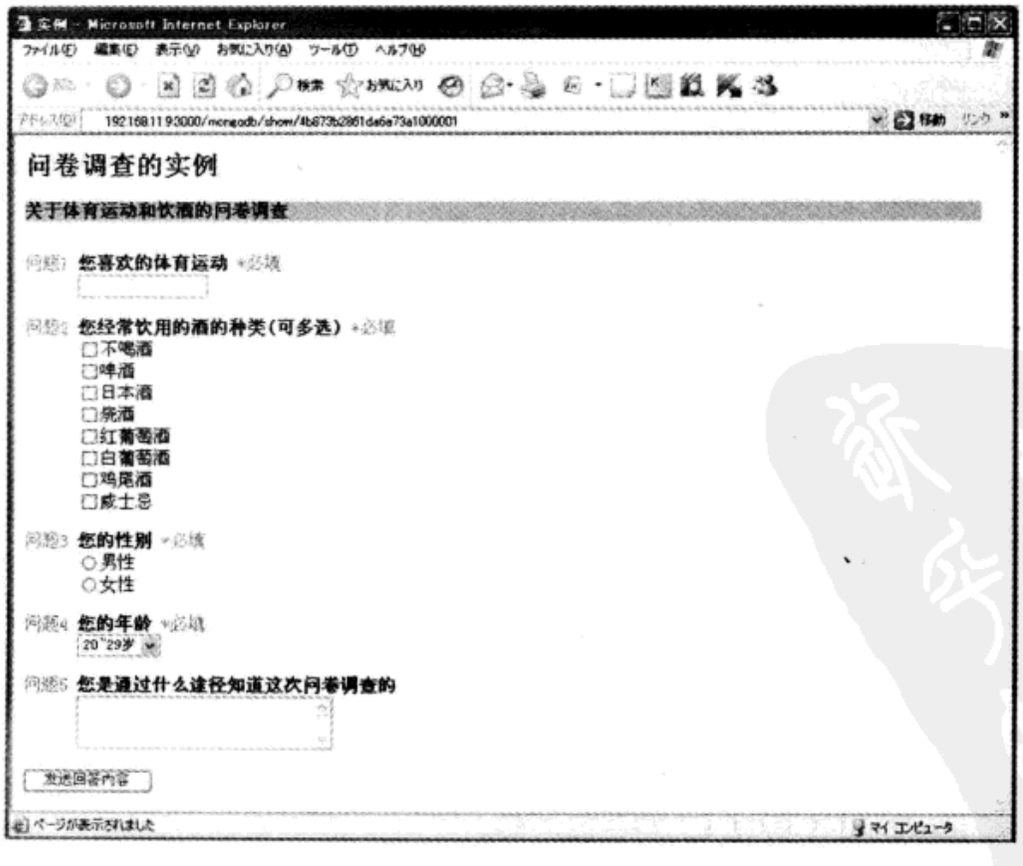
```
  def create
    if params[:commit]
      questions = Enquete.find(params[:answer][:enquete_id]).questions

      @ err_msg = {}
      questions.each do | q|
        if q.required_column && params[:answer][q.question_number].blank?
          @ err_msg[q.question_number] = "未输入"
        end
      end

      if @ err_msg.blank?
        answer = Answer.new(params[:answer])
        answer.save!
        flash[:msg] = "您的回答已经被接受"
```

```
        redirect_to :action => 'show', :id => params[:answer][:enquete_id]
      else
        @ enquete = Enquete.find(params[:answer][:enquete_id])
        render :action => 'show'
      end
    end
end
```

无需变更数据结构

到此我们完成了这个例子，它可以保存任何类型的调查问卷。即使问题的数量很多，也无需变更数据结构（追加字段等）。

问卷调查的回答有多种形式，问题的数量也不固定，因此如果使用关系型数据来完成的话就会非常让人头疼（我们已经在第2章进行了说明）。特别是每次追加字段的时候都需要变更表的数据结构，还需要花时间来保证数据库和程序代码的一致性。这种情况如果使用无数据结构的MongoDB就会感觉非常简单。



本例中使用的程序代码如下所示。

mongodb _enquete. rb

```
class Enquete
  include MongoMapper::Document

  key :name, String, :required => true
  timestamps!
  many :questions
end
```

mongodb _question. rb

```
class Question
  include MongoMapper::EmbeddedDocument

  key :body, Object, :required => true
  key :type, String, :required => true
  key :choices, Array
  key :required_column, Boolean, :default => false
end
```

mongodb _answer. rb

```
class Answer
  include MongoMapper::Document

  key :user_id, Integer, :required => true
  key :enquete_id, String, :required => true
  timestamps!
end
```

mongodb _controller. rb

```
class MongodbController <ApplicationController
  require 'mongo_mapper'
  MongoMapper::database = 'mydb'

  def index
    @ enquetes = Enquete.all
  end

  def show
    @ enquete = Enquete.find(params[:id])
```

```
end

def create
  if params[:commit]
    questions = Enquete.find(params[:answer][:enquete_id]).questions

    @ err_msg = {}
    questions.each do | q|
     if q.required_column && params[:answer][q.question_number].blank?
      @ err_msg[q.question_number] = "未输入"
     end
    end

    if @ err_msg.blank?
     answer = Answer.new(params[:answer])
     answer.save!
     flash[:msg] = "您的回答已经被接受"
     redirect_to :action => 'show', :id => params[:answer][:enquete_id]
    else
     @ enquete = Enquete.find(params[:answer][:enquete_id])
     render :action => 'show'
    end
  end
end

def list
  @ answers = Answer.all
end

def create_answer
  10.times do
    condition = {}
    (1..12).each do | n|
     if rand(100) %  4 == 1
      condition["q_#{n}"] = "a_#{n}"
     elsif rand(100) %  4 == 2
      condition["q_#{n}"] = "a\ nb_#{n}"
     elsif rand(100) %  4 == 3
      condition["q_#{n}"] = 1 +  rand(5)
     else
      list = [1, 2, 3, 4, 5]
      condition["q_#{n}"] = list.sort_by{rand}[0..rand(5)]
     end
    end

    Answer.create!(
     condition.merge(
      :user_id => 1,
```

```
        :enquete_id => 1
      )
    )
  end

  render :text => "hoge"
end

def create_question
  enquete = Enquete.new(
    :name => "关于体育运动和酒的调查"
  )

  question_list = []
  type_h = {
    :q_1 => 'text',
    :q_2 => 'checkbox',
    :q_3 => 'radio',
    :q_4 => 'pulldown',
    :q_5 => 'textarea'
  }
  body_h = {
    :q_1 => '您喜欢的体育运动',
    :q_2 => '您经常饮用的酒的种类',
    :q_3 => '您的性别',
    :q_4 => '您的年龄',
    :q_5 => '您是通过什么途径知道这次问卷调查的'
  }
  choices_h = {
    :q_1 => nil,
    :q_2 => %w(
      不喝酒
      啤酒
      日本酒
      烧酒
      红葡萄酒
      白葡萄酒
      鸡尾酒
      威士忌
    ),
    :q_3 => %w(男性  女性),
    :q_4 => %w(20~29岁  30~39岁  40~49岁  50~59岁  60岁以上),
    :q_5 => nil
  }
  require_h = {
    :q_1 => true,
    :q_2 => true,
    :q_3 => true,
```

```
        :q_4 => true,
        :q_5 => false
      }
      (1..5).each do | n|
        question_list <<Question.new(
          :question_number => "q_#{n}",
          :body => body_h["q_#{n}".to_sym],
          :type => type_h["q_#{n}".to_sym],
          :choices => choices_h["q_#{n}".to_sym],
          :required_column => require_h["q_#{n}".to_sym]
        )
      end

      enquete.questions = question_list
      enquete.save!

      render :text => "hoge"
    end
  end
```

mongodb/index.rhtml

```
<style>
  div.header{
    color:gray;
    text-align:right;
    padding-right:20px;
    vertical-align:text-top;
    float:left;
  }
  div.question{
    float:left;
  }
  div.title{
    font-weight:bold;
  }
  div.clear{
    clear:left;
    padding-bottom:20px;
  }
  span.require{
    color:red;
    font-size:0.8em;
  }
  div.enquete_title{
    font-weight:bold;
```

```
    background-color:silver;
    margin-bottom:30px;
  }
</style>

  <h2> <% = link_to "问卷调查实例", {:action => "index"}, {:style => "textdecoration:
none"} % > </h2>

<% @ enquetes.each do | enquete| -% >
  <% = link_to enquete.name, :action => 'show', :id => enquete.id % > created at
<% = enquete.created_at % > <br />
<% end -% >
```

Chapter 3

mongodb/show. rhtml

```
<style>
  div.header{
    color:gray;
    text-align:right;
    padding-right:20px;
    vertical-align:text-top;
    float:left;
  }
  div.question{
    float:left;
  }
  div.title{
    font-weight:bold;
  }
  div.clear{
    clear:left;
    padding-bottom:20px;
  }
  span.require{
    color:red;
    font-size:0.8em;
  }
  div.enquete_title{
    font-weight:bold;
    background-color:silver;
    margin-bottom:30px;
  }
  div.msg{
    background-color:lightblue;
    text-align:center;
```

```
    margin-bottom:10px;
  }
  span.err_msg{
    color:red;
    background-color:lightyellow;
    font-weight:bold;
  }
</style>

<h2> <% = link_to "问卷调查实例", {:action => 'index'}, {:style => "textdecora-
tion:none"} % > </h2>

<% if flash[:msg]-% >
  <div class = "msg"> <% = flash[:msg] % > </div>
<% end -% >

<div class = "enquete_title"> <% = @ enquete.name % > </div>

<% form_for :answer, :url => {:action => "create"} do | f| -% >
  <% @ enquete.questions.sort{| a,b| a.question_number.split(/_/)[1].to_i
<=> b.question_number.split(/_/)[1].to_i}.each do | q| -% >
  <% n = q.question_number.split(/_/)[1].to_i -% >
  <div class = "header"> 问题<% = n % > </div>
  <div class = "question">
    <% if @ err_msg && @ err_msg[q.question_number] -% >
      <span class = "err_msg"> <% = @ err_msg[q.question_number] % > </span>
    <% end -% >
    <div class = "title"> <% = q.body % > <% = % Q!<span class = "require"> ※必须
</span> ! if q.required_column % > </div>
    <% if q.type == 'text' -% >
      <% = f.text_field q.question_number % >
    <% elsif q.type == 'radio'-% >
      <% q.choices.each_with_index do | choice, i| -% >
       <% = check_box_tag "answer[#{q.question_number}][]", i, false, :id =>
"answer_#{q.question_number}_#{i}" % >
       <label for = "answer_<% = q.question_number % > _<% = i % > "> <% =
choice % > </label> <br />
      <% end -% >
    <% elsif q.type == 'checkbox' -% >
      <% q.choices.each_with_index do | choice, i| -% >
       <% = f.radio_button q.question_number, i % >
       <label for = "answer_<% = q.question_number % > _<% = i % > "> <% =
choice % > </label> <br />
      <% end -% >
    <% elsif q.type == 'pulldown' -% >
      <%
       list = []
       q.choices.each_with_index do | choice, i|
```

```
          list << [choice, i]
        end
      -% >
      <% = f.select q.question_number, list % >
    <%  elsif q.type == 'textarea' -% >
      <% = f.text_area q.question_number, :size => '40x5' % >
    <%  end -% >
  </div>
  <div class = "clear"> </div>
<%  end -% >

<% = f.hidden_field :user_id, :value => @ user.id % >
<% = f.hidden_field :enquete_id, :value => @ enquete.id % >

<div> <% = f.submit '发送回答内容', :style => "font- size:20px" % > </div>
<%  end -% >
```

Chapter 3

mongodb/list. rhtml

```
<style>
  table th{
    background-color:silver;
  }
</style>

<table border = "1" cellspacing = "0">
  <%  @ answers.each_with_index do | answer, i| -% >
    <%  _keys = answer.keys.keys.select{| k|  k = ~  /^q/}.sort{| a,b|  a.split
(/_/)[1].to_i <=> b.split(/_/)[1].to_i} -% >
    <%  if i.zero? -% >
      <tr>
        <%  _keys.each do | key|  -% >
          <th> <% = key % > </th>
        <%  end -% >
      </tr>
    <%  end -% >
    <tr>
      <%  _keys.each do | key|  -% >
        <td>
          <%  if answer[key].instance_of?(Array) -% >
            <% = answer[key].join(', ') % >
          <%  else -% >
            <%  if answer[key].instance_of?(String) -% >
              <% = answer[key].gsub("\ n", "<br /> ") % >
            <%  else -% >
```

```
            <% = answer[key] % >
          <% end -% >
        <% end -% >
      </td>
    <% end -% >
  </tr>
<% end -% >
</table>
```


3.4.2 例②解析数据的存储

让我来看另外一个例子，尝试把MongoDB作为解析数据的存储器，也就是把日志数据进行解析，然后使用MongoDB来保存解析结果。为了检查服务的使用情况，提供更好的服务，可以通过解析日志数据，获取PV[①]和UU[②]，以及其他各类数据，这些数据应该如何保存呢？

使用关系型数据库

当然使用关系型数据库来管理这些数据也不是不可以，但是，对于分析日志数据来说，我们并不清楚哪些字段是必须的。可能有些数据刚开始并不是必须的，但在进行某些分析时就变成必须的了。因此，可以考虑在创建解析用表时把所有解析数据都保存起来，在必要的时候添加一列，或者把每条解析数据保存到单独的表中。

如果把解析数据全都保存在同一个表中，每添加一列需要花费很多工夫，而且我们还要考虑到，只有在某些特定分析中才用得到的字段会越来越多。如果把每条解析数据都保存到单独的表中，表的数量就会在短时间内骤然增加，也很让人头疼。

使用MongoDB

这种情况下尝试MongoDB不失为明智之选。解析日志数据的时候会得到各种各样的数据，若使用MongoDB便可以不用考虑表结构的问题，直接把所有数据都保存下来即可。需要添加新数据的时候，也无需变更表结构，直接就能进行添加，这就是使用MongoDB的好处。对于那些以后可能会用到的临时数据，保存起来也很简单。

① page view即页面浏览量，或点击量。——译者注

② Unique Users指在单位时间内访问某一站点的所有不同的用户的数量。——译者注

MongoDB 的地理空间索引

MongoDB 还可以使用地理空间索引[①]，将经纬度信息作为索引来进行查询。

最近使用 GPS 的服务逐渐增加，同时 HTML5 还推出了 Geolocation API。估计今后定位信息将会越来越重要。在定位信息查询领域，GeoHash 是非常出名的，它能把经纬度信息汇总成一个字符串[②]。

例如，东京塔的经度是 35.65861，纬度是 139.745447，用来表示包含东京塔的 19m * 31m 的长方形范围的字符串就是 xn76ggrw26。

GeoHash 的另外一个特点就是可以根据字符串的长度来改变位置的精度。例如，我们取出之前代表东京塔位置的字符串的前 6 位（xn76gg），就可以表示出 609m * 989m 这样更大范围的位置。虽然通过改变字符串长度来改变精度大小是非常方便的，但是它还存在着难以通过直线距离进行排序，以及难以指定查询范围的问题。

与此相对，MongoDB 的地理空间索引直接利用经纬度信息，可以非常简单地进行使用。

```
# 把使用地理空间索引的字段指定为“2d”
# 也可以创建复合索引
> db.places.ensureIndex({ location:"2d" })

# 插入数据

# 返回距离[139.66,35.65]最近的5条记录
> db.places.find({ location :{$near :[139.66, 35.65] } }).limit(5)
{ "_id" :ObjectId("4d9ba63e2232a4d86ebd4c21"), "location" :[ 139.65984947,
35.64900162 ]}
{ "_id" :ObjectId("4d9ba63e2232a4d86ebd4c4a"), "location" :[ 139.66440523,
35.65416243 ]}
{ "_id" :ObjectId("4d9ba63e2232a4d86ebd4c70"), "location" :[ 139.66039999,
35.64228010 ]}
{ "_id" :ObjectId("4d9ba63e2232a4d86ebd4c64"), "location" :[ 139.65993267,
35.65967964 ]}
{ "_id" :ObjectId("4d9ba63e2232a4d86ebd4c50"), "location" :[ 139.66983844,
35.64824236 ]}
```

① 1.3.3 版本开始可以使用地理空间索引，另外 1.7.0 以上的版本可以使用正确的球体模型。

② http://blog.masuidrive.jp/index.php/2010/01/13/geohash/

性能验证

目前为止我们介绍了很多种 NoSQL 数据库，并了解了它们具体的应用实例和使用方法。那么在实际使用的过程中，它们的性能究竟如何呢？也许这是大家最关心的事。

本章将会对各种 NoSQL 数据库的实际性能进行简单地验证。

4.1 基本的插入和查询处理的性能

首先让我们比较一下最基本的插入和查询处理的性能。NoSQL 方面我们使用 memcached、Tokyo Tyrant（memcached 兼容协议和独立协议）、Redis、MongoDB 作为比较对象，关系型数据库方面我们使用 MySQL（InnoDB 和 MyISAM）作为比较对象。

4.1.1 假定案例

假定保存用户 ID 和某个其他数据，然后我们需要通过用户 ID 读取出这个数据。应该怎么办呢？对于 MySQL 和 MongoDB，我们创建 user _id 和 value 这两个字段，读取数据的时候通过 user _id 进行查询，取得 value 的值。对于 memcached、Tokyo Tyrant 和 Redis，我们对 user _id 对应的 value 进行操作。

我们以每完成 2 万次处理作为标准进行比较。实际的验证程序如下所示，验证插入性能的是 insert 方法，验证查询性能的是 select 方法。

new _benchmark _controller. rb

```
class NewBenchmarkController <ApplicationController
  before_filter :init
  after_filter :finalize

  require 'benchmark'
  require 'memcache'
  require 'tokyotyrant'
  require 'redis'
  require 'mongo_mapper'
  MongoMapper::database = 'mydb'

  LIMIT = 20000

  class Hoge <ActiveRecord::Base
  end

  class Fuga <ActiveRecord::Base
  end
```

```
class ::Foo
  include MongoMapper::Document
end

def insert
  Benchmark.bm do | x|
    x.report('InnoDB') {
     1.upto(LIMIT) do | user_id|
      Hoge.create!(:user_id => user_id, :value => "value_#{user_id}")
     end
    }
    x.report('MyISAM') {
     1.upto(LIMIT) do | user_id|
      Fuga.create!(:user_id => user_id, :value => "value_#{user_id}")
     end
    }
    x.report('memcached') {
     1.upto(LIMIT) do | user_id|
      @ mem[user_id.to_s] = "value_#{user_id}"
     end
    }
    x.report('Tokyo Tyrant (mem)') {
     1.upto(LIMIT) do | user_id|
      @ tt_mem[user_id.to_s] = "value_#{user_id}"
     end
    }
    x.report('Tokyo Tyrant') {
     1.upto(LIMIT) do | user_id|
      @ tt[user_id.to_s] = "value_#{user_id}"
     end
    }
    x.report('Redis') {
     1.upto(LIMIT) do | user_id|
      @ redis.set user_id.to_s, "value_#{user_id}"
     end
    }
    x.report('MongoDB') {
     1.upto(LIMIT) do | user_id|
      Foo.create!(:user_id => user_id, :value => "value_#{user_id}")
     end
    }
  end

  render :text => "hoge"
end

def select
  Benchmark.bm do | x|
    x.report('InnoDB') {
     1.upto(LIMIT) do
      user_id = 1 + rand(LIMIT)
```

```
        Hoge.first(:conditions => ["user_id = ?", user_id]).value
      end
    }
    x.report('MyISAM') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        Fuga.first(:conditions => ["user_id = ?", user_id]).value
      end
    }
    x.report('memcached') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        @ mem[user_id.to_s]
      end
    }
    x.report('Tokyo Tyrant (mem)') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        @ tt_mem[user_id.to_s]
      end
    }
    x.report('Tokyo Tyrant') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        @ tt[user_id.to_s]
      end
    }
    x.report('Redis') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        @ redis.get user_id.to_s
      end
    }
    x.report('MongoDB') {
      1.upto(LIMIT) do
        user_id = 1 + rand(LIMIT)
        Foo.first(:user_id => user_id).value
      end
    }
  end

  render :text => "hoge"
end

private

def init
  @ mem = MemCache.new(['localhost:11211'])
```

```
    @ tt_mem = MemCache.new('localhost:1979')

    @ tt = TokyoTyrant::RDB.new
    @ tt.open('localhost', 1978)

    @ redis = Redis.new
  end

  def finalize
    @ tt.close
  end
end
```


4.1.2 准备工作

首先，创建 MySQL 需要使用的 hoges 表和 fugas 表。其中，hoges 表是 InnoDB 类型的，fugas 表是 MyISAM 类型的。另外，为了验证 Tokyo Tyrant 的独立协议和 memcached 兼容协议，需要启动 1978 和 1979 两个端口。其中，1978 端口用来验证独立协议，1979 端口用来验证 memcached 兼容协议。

启动方法如下所示：

```
# 启动 MySQL
sudo /etc/init.d/mysqld start

# 启动 memcached
sudo /etc/init.d/memcached start

# 启动 Tokyo Tyrant
ttserver data_1978.tch
ttserver-port 1979 data_1979.tch

# 启动 Redis
redis-server /etc/redis.conf

# 启动 MongoDB
sudo /etc/init.d/mongod start
```

4.1.3 插入处理的性能

让我们来看一下性能测试的结果。首先来比较一下插入处理的性能。

	user	system	total	real
InnoDB	8.060000	0.240000	8.300000	(47.546858)
MyISAM	14.720000	0.500000	15.220000	(27.786782)
memcached	1.480000	0.210000	1.690000	(3.189919)
Tokyo Tyrant (mem)	1.520000	0.200000	1.720000	(3.374147)
Tokyo Tyrant	0.360000	0.180000	0.540000	(1.357315)
Redis	0.460000	0.350000	0.810000	(2.213919)
MongoDB	5.650000	0.190000	5.840000	(11.350683)

由此可以看到使用独立协议的 Tokyo Tyrant 和 Redis 的处理速度非常快。相反，由于程序库的影响[①]，memcached 和使用 memcached 兼容协议的 Tokyo Tyrant 却并没有表现出预想的性能。当然，跟关系型数据比起来，它们都表现出了更快的处理速度。

MongoDB 的性能介于键值存储和关系型数据库之间。虽然没有键值存储那样高的性能，但是它拥有比关系型数据库更快的处理速度，也是相当实用的。但是，MongoDB 在数据创建和更新时写入磁盘的处理是异步的，需要特别注意[②]。

这是一个非常粗略的验证标准，只是想让大家体会一下 memcached、Tokyo Tyrant、Redis、MongoDB 这些 NoSQL 数据库的处理速度有多快。

4.1.4 查询的性能

接下来让我们比较一下查询的性能。MySQL 和 MongoDB 并未使用索引功能。

	user	system	total	real
InnoDB	4.570000	0.240000	4.810000	(120.209437)
MyISAM	5.060000	0.390000	5.450000	(73.479136)
memcached	4.550000	0.250000	4.800000	(8.430305)
TokyoTyrant (mem)	3.920000	0.210000	4.130000	(7.506806)
TokyoTyrant	0.450000	0.220000	0.670000	(1.630736)
Redis	0.480000	0.440000	0.920000	(2.319238)
MongoDB	9.920000	0.320000	10.240000	(109.883533)

① http://blog.katsuma.tv/2010/06/memcached_vs_tokyocabinet_vs_tokyo_tyrant_vs_redis.html

② http://www.mongodb.org/pages/viewpage.action?pageId=5079223

在这个测试中，果然还是使用独立协议的 Tokyo Tyrant 和 Redis 拥有压倒性的处理速度。跟插入处理时一样，memcached 和使用 memcached 兼容协议的 Tokyo Tyrant 由于程序库的影响，处理速度略有延迟。据此，大家不仅能体会到 memcached、Tokyo Tyrant、Redis 这些键值存储的高速处理能力，同时也能看到 MongoDB 和 MySQL 的性能相当。

顺便说一下，如果 MySQL 和 MongoDB 创建了索引，也能获得理想的查询速度。下面就让我们试着给 MySQL 和 MongoDB 创建索引。

```
mysql> ALTER TABLE hoges ADD INDEX user_id_value (user_id, value);
Query OK, 10000 rows affected (0.14 sec)
Records:10000 Duplicates:0 Warnings:0

mysql> ALTER TABLE fugas ADD INDEX user_id_value (user_id, value);
Query OK, 10000 rows affected (0.06 sec)
Records:10000 Duplicates:0 Warnings:0
```

```
$ mongo mydb
MongoDB shell version:1.6.5
connecting to:mydb
> db.foos.ensureIndex({ user_id:1, value:1 })
```

结果如下所示：

	user	system	total	real
InnoDB	4.150000	0.170000	4.320000	(9.032817)
MyISAM	4.770000	0.160000	4.930000	(13.362070)
memcached	4.690000	0.280000	4.970000	(8.561159)
TokyoTyrant(mem)	3.930000	0.180000	4.110000	(7.479404)
TokyoTyrant	0.520000	0.250000	0.770000	(1.828418)
Redis	0.520000	0.330000	0.850000	(2.130677)
MongoDB	8.650000	0.240000	8.890000	(16.686684)

通过创建索引，MySQL 和 MongoDB 都可以获得非常高的处理速度。虽然适当地创建索引可以提高查询的性能，但是请大家一定注意，不要过多地创建索引，不然在添加和更新数据的时候，会由于需要更新索引而造成延迟。

最后，让我们以 MySQL 为参照，总结一下这些 NoSQL 数据库的性能吧。如

表 4-1所示。

表 4-1 与 MySQL 进行比较的 NoSQL 数据库的性能对比

	memcached	Tokyo Tyrant（mem）	Tokyo Tyrant	Redis	MongoDB
插入	快速	快速	非常快速	非常快速	较快速
查询	快速	快速	非常快速	非常快速	速度相同

4.2 不同实例的性能比较

插入和查询处理的基本性能大家应该都已经清楚了，接下来让我们通过具体的实例，对它们的性能进行进一步的比较。

4.2.1 Tokyo Tyrant 的 addint 方法和 incr 方法

使用 Tokyo Tyrant 对访问计数器这样的数值进行加算时，可以通过使用独立协议的 addint 方法或者是通过使用 memcached 兼容协议的 incr 方法来实现（这两种方法在第 3 章的时候进行过说明）。那么，二者在性能上究竟有什么不同呢？

准备工作

首先，通过 1978 号端口和 1979 号端口启动 2 台 Tokyo Tyrant。其中，1978 号端口的 Tokyo Tyrant 使用独立协议，1979 号端口的 Tokyo Tyrant 使用 memcached 兼容协议。

```
ttserver data_1978.tch
ttserver -port 1979 data_1979.tch
```

数值加算的性能

我们通过对数值进行 10 万次加算来进行测试，结果如下所示：

```
                  user     system      total        real
addint        3.950000   1.940000   5.890000  (15.201939)
incr         11.920000   1.140000  13.060000  (27.067998)
```

抛开程序库的影响不说，addint 方法的处理速度还是更快的。虽然 addint 方法（特别是读取数据的时候）多少有一些不足，但是在需要高速返回响应时表现非常出色。反观 incr 方法，它可能在处理速度上多少有一些劣势，却可以像 memcached 一

样进行操作，使用起来非常简单。

验证所用的数据如下所示。虽然使用 memcached 兼容协议时通常会进行序列化（Marshal. dump）处理，但是 incr 方法并不会进行序列化处理，所以单纯取得数据时，请不要进行序列化处理。

Benchemark _p1 _controller. rb

```
class BenchmarkP1Controller <ApplicationController
  before_filter :init
  after_filter :finalize

  require 'benchmark'
  require 'memcache'
  require 'tokyotyrant'

  LIMIT = 100000

  def index
    Benchmark.bm do | x|
      x.report('addint') {
       1.upto(LIMIT) do
        @ tt.addint("counter", 1)
        @ tt.get("counter").unpack('i').first
       end
      }
      x.report('incr') {
       1.upto(LIMIT) do
        _counter = @ tt_mem.incr("counter")
        if !_counter
         @ tt_mem.set("counter", 0)
         @ tt_mem.incr("counter")
        end
       end
      }
    end

    render :text => "hoge"
  end

  private

  def init
    @ tt = TokyoTyrant::RDB.new
    @ tt.open('localhost', 1978)

    @ tt_mem = MemCache.new('localhost:1979')
  end

  def finalize
    @ tt.close
  end
end
```

4.2.2 对 Redis 的列表类型的数据进行添加和删除

Redis 可以对列表类型的数据进行高速的添加和删除。不过，实际应用中到底能达到多高的速度呢？让我们来见识一下。

准备工作

启动 Redis。

```
redis-server /etc/redis.conf
```


列表操作的性能

我们对列表进行数据添加并读取出最新的 10 条数据，由此反复 10 万次。为了对比，我们对 MySQL 的 InnoDB 也进行同样的处理。结果可以发现使用 Redis 可以获得优于后二者 10 倍以上的处理速度。

	user	system	total	real
MySQL	68.090000	2.450000	70.540000	(330.203288)
Redis	8.590000	3.480000	12.070000	(28.179236)

通过测试结果 Redis 出色的处理速度显而易见。鉴于此，在处理这种时间序列的数据时，非常值得考虑使用 Redis 来存储数据。

验证使用的程序如下所示：

benchmark _p2 _controller. rb

```
class BenchmarkP2Controller <ApplicationController
  before_filter :init

  require 'benchmark'
  require 'redis'

  class List <ActiveRecord::Base
  end

  LIMIT = 100000

  def index
```

```
    Benchmark.bm do | x|
      x.report('MySQL') {
       1.upto(LIMIT) do | num|
        List.create!(:data => "data_#{num}")
        List.all(:order => "id DESC", :limit => 10)
       end
      }
      x.report('Redis') {
       1.upto(LIMIT) do | num|
        @ redis.lpush :list, "data_#{num}"
        @ redis.lrange :list, 0, 9
       end
      }
    end

    render :text => "hoge"
  end

  private

  def init
    @ redis = Redis.new
  end
end
```

Chapter 4

4.2.3 MySQL 的 JOIN 和 MongoDB 的 embed

刚才测试了 MongoDB 进行单纯的插入和查询处理时的性能，但是实际的应用往往是更加复杂的。比如对于关系型数据库来说，使用 JOIN 的实例还是占多数。至于 MongoDB，它的 embed 处理的性能如何呢？

让我们尝试对 MySQL 的 InnoDB 和 MongoDB 进行比较。

准备工作

启动 MongoDB。

```
sudo /etc/init.d/mongod start
```

插入处理的性能

首先让我们插入 2 万条数据。二者性能的比较结果如下所示，可以看到与

MySQL 比起来，MongoDB 拥有压倒性的处理速度。其中的原因之一就是 MySQL 进行一次处理的同时需要进行多次（Blog 和 Aricle）保存，而 MongoDB 只需要进行一次（BlogMongo）保存就可以了。

	user	system	total	real
MySQL	67.780000	1.690000	69.470000	(502.815060)
MongoDB	32.470000	0.250000	32.720000	(61.089533)

查询处理的性能

更重要的是查询处理的性能。由于这样的处理比较费时，所以我们只对执行 1000 次的结果进行比较。MySQL 的 JOIN 处理与 MongoDB 的 embed 处理，在读取数据方面的性能有什么不同呢？测试结果如下所示：

	user	system	total	real
MySQL	0.780000	0.020000	0.800000	(221.800680)
MongoDB	0.420000	0.010000	0.430000	(5.619436)

与 JOIN 比起来，embed 的处理速度具有压倒性的优势。这是因为 embed 处理在取得集合 blog _mongos 时会同时获得被 embed 的集合 article _mongos。不过，这样的 embed 处理在取得 embed 元集合 blog _mongos 的数据时没有什么问题，但是在取得 embed 对象集合 article _mongos 时，会把 embed 元集合 blog _mongos 也取出来，效率并不理想。

验证用的程序如下所示：

benchmark _p3 _controller. rb

```
class BenchmarkP3Controller <ApplicationController
  require 'benchmark'
  require 'mongo_mapper'
  MongoMapper::database = 'mydb'

  LIMIT = 20000
  LIMIT_S = 1000

  class ::Blog <ActiveRecord::Base
    has_many :articles
  end
```

```
class ::Article <ActiveRecord::Base
end

class ::BlogMongo
  include MongoMapper::Document
  many :article_mongos
end

class ::ArticleMongo
  include MongoMapper::EmbeddedDocument
end

def insert
Benchmark.bm do | x|
  x.report('MySQL') {
    1.upto(LIMIT) do | num|
     @ blog = Blog.create!(:title => "title_#{num}")
     1.upto(10) do | num2|
      @ article = Article.create!({
       :title => "title_#{num2}",
       :body => "body_#{num2}",
       :blog_id => @ blog.id
      })
     end
    end
  }
  x.report('MongoDB') {
    1.upto(LIMIT) do | num|
     @ blog_mongo = BlogMongo.new(
      :blog_id => num, # 用来查询数据
      :title => "title_#{num}"
     )
     1.upto(10) do | num2|
      @ blog_mongo.article_mongos <<ArticleMongo.new(
       :title => "title_#{num2}",
       :body => "body_#{num2}"
      )
     end
     @ blog_mongo.save!
    end
  }
end

  render :text => "hoge"
end

def select
  Benchmark.bm do | x|
    x.report('MySQL') {
```

```
        1.upto(LIMIT_S) do
         Blog.first(
          :include => :articles,
          :conditions => ["id = ?", (1 + rand(LIMIT))]
         ).articles
          end
         }
         x.report('MongoDB') {
          1.upto(LIMIT_S) do
           BlogMongo.first(
            :blog_id => (1 + rand(LIMIT))
           ).article_mongos
          end
        }
      end

      render :text => "hoge"
    end
  end
```


NoSQL 化的关系型数据库

NoSQL 在特定用途下体现出了优秀的性能，但是，大家可能会担心在实际使用过程中所需要的开销，以及使用经验不足等问题。

本章将转换视角，介绍一下将 MySQL 数据库 NoSQL 化的 HandlerSocket 方法。

5.1 关于NoSQL数据库

5.1.1 各种NoSQL数据库的特征

从第2章到第4章，我们介绍了memcached、Tokyo Tyrant、Redis、MongoDB等NoSQL数据库的各种特征和用例，以及具体的使用实例和性能。这里我们再进行一下简单的整理。

memcached

- 挥发性的键值存储
- 一般作为关系型数据库的缓存来使用
- 具有非常快的处理速度
- 由于存在数据丢失的可能，所以一般用来处理不需要持久保存的数据
- 用于需要使用expires时（需要定期清除数据）
- 使用Consistent Hashing算法来分散数据

Tokyo Tyrant

- 持久性的键值存储
- 用来处理需要持久保存，高速处理的数据
- 具有非常快的处理速度
- 用于不需要定期清除数据时
- 使用Consistent Hashing算法来分配数据

Redis

- 兼具memcached和Tokyo Tyrant优势的键值存储
- 擅长处理数组类型的数据
- 具有非常快的处理速度
- 可以高速处理时间序列的数据，易于处理集合运算

- 拥有很多可以进行原子处理的方法
- 使用 Consistent Hashing 算法来分散数据

MongoDB

- 面向无需定义表结构的文档数据
- 具有非常快的处理速度
- 通过 BSON 的形式可以保存和查询任何类型的数据
- 无法进行 JOIN 处理，但是可以通过嵌入（embed）来实现同样的功能
- 使用 sharding（范围分割）算法来分散数据

5.1.2 运行时的开销以及经验不足的问题

确实 NoSQL 数据库对关系型数据库不擅长的某些特定处理进行了补足，性能优秀，使用简便。但是，大家可能更担心实际应用的开销巨大，以及成熟经验不足等问题。一直使用关系型数据库进行的处理，换成 NoSQL 又该怎么办呢？此类适应性问题可能是引入 NoSQL 数据库最大的障碍吧。与此相对的是，关系型数据拥有十分成熟的技术，丰富的实例和经验，并不存在上述问题。

5.1.3 将 MySQL 数据库 NoSQL 化的方法

对于关系型数据库 MySQL 来说，DeNA[①]的樋口先生开发出了具有革命性的插件 HandlerSocket[②]。简单地说，它就是 MySQL 的非 SQL 接口，通过它可以不使用 SQL 就能读取和更新 MySQL 的数据。在第 1 章中我们提到了关系型数据库必须要对 SQL 进行解析，还会增加锁表和解锁这样的额外开销。HandlerSocket 虽然不支持事务处理，但是这种单纯的处理避免了使用 SQL 带来的额外开销，可以高速地完成数据处理，是将 MySQL 数据库 NoSQL 化的捷径。

NoSQL 数据库原本是为了弥补关系型数据的不足而开发的，而 HandlerSocket 却是完全不同的解决方案，它可以使关系型数据库本身具有 NoSQL 的功能，颇具创新性。同在 DeNA 公司的松信先生也在自己的博客中谈到了 HandlerSocket 读取数据

① DeNA 是一家日本移动互联网公司。以 mobage（梦宝谷）为核心，主要有 PlATFORM 和社区游戏等业务。——译者注

② http://engineer.dena.jp/2010/08/handlersocket-plugin-for-mysql.html

时的性能[①]，他认为 HandlerSocket 的处理速度比 memcached 还要快。同时，由于数据库还是 MySQL，只是在读取数据方面有所不同，不仅使用方便而且还有丰富的经验可供参考。

本章将对 HandlerSocket 进行详细的介绍。

① http://yoshinorimatsunobu.blogspot.com/2010/10/using-mysql-as-nosql-story-for.html

5.2 尝试使用 HandlerSocket

5.2.1 特征

HandlerSocket 的特征如下所示。不管怎么说，能高速地插入和读取数据真是令人皆大欢喜。

- 可以高速地完成单纯数据的插入和读取处理
- 可以使用 SQL 或者 HandlerSocket 对同一数据进行访问
- 可以通过独立的协议直接访问 MySQL 的数据
- 不支持事务处理

另外，HandlerSocket 的结构如图 5-1 所示。

图 5-1 HandlerSocket 的结构

由于 Handler 接口的访问级别要低于 SQL，因此免去了解析 SQL 以及锁表、解锁等额外的开销[①]，而 HandlerSocket 可以通过直接访问 Handler 接口获得非常高的处理速度。通常情况下，需要通过 SQL 层来访问 Handler 接口，而 HandlerSocket 却可以直接访问 Handler 接口。

① http://dev.mysql.com/doc/refman/5.1/ja/handler.html

5.2.2 为 MySQL 安装 HandlerSocket

插件的安装

尽管 HandlerSocket 听起来非常优秀，但实际使用的效果究竟如何呢？让我们来安装一下试试看吧。

首先，HandlerSocket 是为 MySQL 准备的，因此需要按照第 3 章介绍的那样，使用 yum 通过 CentOS 的标准库来安装 5.0 系列的版本。由于 MySQL5.1 系列的版本才支持 HandlerSocket，所以我们需要通过导入新的库 remi 来安装 MySQL5.1 系列版本。remi 库的导入方法如下所示。

```
sudo rpm -ivh http://rpms.famillecollet.com/enterprise/remi-release-5.rpm
```


这里需要在/etc/yum.repos.d/下创建包含如下内容的文件 remi.repo，这样就可以使用 remi 库了。本例中我们只使用了 remi 库，若使用 remi-test 库就可以安装最新的 MySQL5.5 系列版本了。

/etc/yum.repos.d/remi.repo

```
[remi]
name = Les RPM de remi pour Enterprise Linux 5-$basearch
baseurl = http://rpms.famillecollet.com/enterprise/5/remi/$basearch/
    http://iut-info.univ-reims.fr/remirpms/enterprise/5/remi/$basearch/
enabled = 0
gpgcheck = 1
gpgkey = file:///etc/pki/rpm-gpg/RPM-GPG-KEY-remi
failovermethod = priority

[remi-test]
name = Les RPM de remi en test pour Enterprise Linux 5-$basearch
baseurl = http://rpms.famillecollet.com//enterprise/5/test/$basearch/
enabled = 0
gpgcheck = 1
gpgkey = file:///etc/pki/rpm-gpg/RPM-GPG-KEY-remi
```

这样就可以通过 remi 来安装 MySQL5.1 系列版本了。另外，我们还需要提前安装 mysqlclient15。

```
sudo yum install mysqlclient15 --enablerepo = remi
```

准备工作完成了，接下来让我们来安装 MySQL。

```
sudo yum install mysql mysql-server mysql-devel --enablerepo = remi
```

MySQL5.1.56 就安装完成。大家最好还是通过程序来进行安装，本例为了更易于管理，所以使用 yum 来进行安装。

接下来，我们为 MySQL 安装 HandlerSocket 插件。首先从 github 取得 HandlerSocket 的源代码。

```
git clone https://github.com/ahiguti/HandlerSocket-Plugin-for-MySQL.git
cd HandlerSocket-Plugin-for-MySQL
```

在进行实际安装之前，需要先安装 libtool 和 g++这两个程序库。


```
sudo yum install libtool gcc-c++
```

安装 HandlerSocket 的时候需要用到 MySQL 的源代码，由于我们是通过 yum 来安装的，无法直接获得源代码，所以需要从其他途径获得相同版本（5.1.56）的 MySQL 的源代码。

```
cd /usr/local/src
wget http://ftp.up.ac.za/pub/linux/MySQL/Downloads/MySQL-5.1/mysql-5.1.56.tar.
gz tar zxvf mysql-5.1.56.tar.gz
```

把取得的源代码解压缩到/usr/local/src/mysql-5.1.56 目录下，接下来就可以安装 HandlerSocket 了。可以通过如下参数来指定源代码的路径，安装 HandlerSocket。

```
./autogen.sh
./configure --with-mysql-source = /usr/local/src/mysql-5.1.56 --with-mysql-
bindir = /usr/bin-with-mysql-plugindir = /usr/lib/mysql/plugin
make
sudo make install
```

接下来我们起动 MySQL。在起动的状态下，向配置文件的［mysqld］部分中添加如下的设定内容。

```
# 连接 handlersocket 的待机端口号(用来接收查询请求)
handlersocket_port = 9998

# 连接 handlersocket 的待机端口号(用来接收更新请求)
handlersocket_port_wr = 9999

# handlersocket 的绑定地址(可以为空)
handlersocket_address =

# 调试用
handlersocket_verbose = 0

# 通信超时时间(秒)
handlersocket_timeout = 300

# handlersocket 的工作线程数
handlersocket_threads = 16

# 指定 handlersocket 的最大线程数
thread_concurrency = 128

# 指定最大的接口数(socket)
open_files_limit = 65535
```


最后连接 MySQL，安装 HandlerSocket 插件。

```
mysql> INSTALL PLUGIN handlersocket SONAME 'handlersocket.so';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SHOW PLUGINS;
```

Name	Status	Type	Library	License
binlog	ACTIVE	STORAGE ENGINE	NULL	GPL
partition	ACTIVE	STORAGE ENGINE	NULL	GPL
ARCHIVE	ACTIVE	STORAGE ENGINE	NULL	GPL
BLACKHOLE	ACTIVE	STORAGE ENGINE	NULL	GPL
CSV	ACTIVE	STORAGE ENGINE	NULL	GPL
FEDERATED	DISABLED	STORAGE ENGINE	NULL	GPL
MEMORY	ACTIVE	STORAGE ENGINE	NULL	GPL
InnoDB	ACTIVE	STORAGE ENGINE	NULL	GPL
MyISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
MRG_MYISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
handlersocket	ACTIVE	DAEMON	handlersocket.so	BSD

```
11 rows in set (0.00 sec)
```

这样就将 HandlerSocket 装入 MySQL 了，我们可以在插件列表中找到它。

客户端程序库

让我们继续来安装客户端的程序库。

```
./autogen.sh
./configure --disable-handlersocket-server
make
sudo make install
```

5.2.3 动作确认

现在就让我们通过 telnet 来实际确认一下 HandlerSocket 的动作。首先尝试操作 handlersocket 数据库中的 test 表。


```
mysql> CREATE DATABASE handlersocket;

mysql> USE handlersocket;

mysql> CREATE TABLE test (
    ->   id INT NOT NULL PRIMARY KEY,
    ->   data VARCHAR(255)
    -> ) ENGINE= INNODB;
```

插入数据

首先让我们尝试进行数据的插入操作。连接处理更新请求专用的 9999 号端口，使用制表键作为 HandlerSocket 命令的分隔符。

```
$ telnet localhost 9999
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

# open_index
# P <indexid> <db> <table> <index> <col_list>
```

```
# <indexid> 设定为1
P 1    handlersocket  test  PRIMARY id,data
0 1

# insert
# <indexid> + <col_num> <col_1> .. <col_n>
# 插入 id 是1,data 是"data_1"的数据;
1 +    2    1    data_1
0 1
```

首先，HandlerSocket 的处理全都要使用索引，因此在最开始的时候需要为数据库和表创建特定的索引。这时，可以把任意值作为本次连接的＜indexid＞，之后的处理都会针对这个＜indexid＞来执行。

接下来就是数据插入的部分了，通过＜col _num＞来指定需要插入的字段个数。本例中要插入 id 和 data 两个字段的数据，因此设定为 2。倘若是单独插入 id（data 不指定任何值）则设定为 1。


```
1 +    1    1
0 1
```

这时可以通过其他连接来连接 MySQL，通过 SQL 来确认 test 表中的数据。这是因为 HandlerSocket 和 SQL 只是接口不同，但是却可以访问相同的数据。

```
mysql> SELECT * FROM test;
+------+--------+
| id   | data   |
+------+--------+
|    1 | data_1 |
+------+--------+
```

同样的数据，既可以通过 HandlerSocket 取得，也可以通过 SQL 取得。接下来让我们再插入两条数据。

```
1 +    2    2    data_2
0 1

1 +    2    3    data_3
0 1
```

读取数据

接下来我们进行数据的读取操作。虽然 9999 号端口也能完成查询处理，但是这次我们还是使用查询专用的 9998 号端口来完成。

```
P 1    handlersocket  test  PRIMARY id,data
0 1

# select
# <indexid> <op> <col_num> <col_1> .. <col_n> <limit> <offset>
# 取出1条 id 是1的数据
1 =    1    1
0 2    1    data_1

# 取出3条 id 大于1的数据

1 >    1    1         3
0 2    2    data_2    3    data_3
```

可以通过<col_num> 来设定作为查询条件的字段的个数，<limit> 和<offset> 也可以不设定，默认的设置为 1 和 0。另外，<op> 可以进行的设定如下所示：

- '=' 取出与<col_1> .. <col_n> 相同的数据
- '>' 按照升序取出比<col_1> .. <col_n> 大的数据
- '>=' 按照升序取出取出与<col_1> .. <col_n> 相同或者比<col_1> .. <col_n> 大的数据
- '<' 按照降序取出比<col_1> .. <col_n> 小的数据
- '<=' 按照降序取出取出与<col_1> .. <col_n> 相同或者比<col_1> .. <col_n> 小的数据

另外需要注意的是，HandlerSocket 只支持查询带有索引的数据。之前的例子由于设定了 PRIMARY（把 id 设定为主索引），所以只能通过 id 进行查询。那么，如果想通过 data 进行查询应该怎么办呢？

这就需要为 data 创建索引了，使用复合索引也是可以的。

```
mysql> ALTER TAbLE test ADD INDEX data (data);
Query OK, 3 rows affected (1.99 sec)
Records:3 Duplicates:0 Warnings:0
```

```
mysql> ALTER TAbLE test ADD INDEX id_data (id, data);
Query OK, 3 rows affected (0.01 sec)
Records:3 Duplicates:0 Warnings:0
```

借助这个索引，就可以通过data进行查询，或者是通过id和data进行联合查询了。让我们来试一下吧。

```
# 不使用主键作为索引,而是为 data 创建索引
# <indexid> 把<indexid> 设定为2
P 2    handlersocket  test  data  id,data
0 1

# 由于查询处理需要使用索引,所以这里要为 data 创建索引
2 =    1    data_1
0 2    1    data_1
```


为data创建了索引之后就可以通过data来查询数据了。这时由于没有为id创建索引，因此无法通过id进行查询。

接下来我们尝试一下复合索引。

```
# 使用 id_data 作为索引
# 把<indexid> 设定为3
P 3    handlersocket  test  id_data id,data
0 1

# 这里<field_num> 的值是2,所以 id 和 data 都可以指定
3 =    2    3    data_3
0 2    3    data_3
```

这样，我们就可以通过创建id和data的复合索引，使得通过id和data进行查询的操作称谓可能。

更新数据

接下来我们尝试更新操作。

```
P 1    handlersocket  test  PRIMARY id,data
0 1
```

```
# update
# <indexid> <op> <col_num> <col_1> .. <col_n> <limit> <offset> U <mod_1>
..<mod_n>
# 取出id是3的数据,然后把它的id更新为3,data更新为"data_03"
1 =    1    3    1    0    U    3    data_03
0 1    1

# 数据更新成功
1 =    1    3
0 2    3    data_03
```

删除数据

最后我们来看一下删除数据的处理。


```
# delete
# <indexid> <op> <col_num> <col_1> .. <col_n> <limit> <offset> D
# 取出1条id是1的数据,然后把它删除
1 =    1    1    1    0    D
0 1    1

# 删除数据
1 =    1    1
0 2
```

这时我们可以通过SQL来确认test表的状态。通过SQL取出的数据，我们能够确认目前为止的操作结果全都是正确的。

```
mysql> SELECT * FROM test;
+----+---------+
| id | data    |
+----+---------+
| 3  | data_03 |
| 2  | data_2  |
+----+---------+
```

一系列操作（插入、读取、更新、删除）确认完毕。

在 Ruby 程序上使用

接下来我们尝试通过 ruby-handlersocket 程序库[①]，在 Ruby 程序上使用 HandlerSocket。首先来安装 ruby-handlersocket，安装时需要使用 HandlerSocket 源程序中的 libhsclient 目录，我们将它复制出来。

```
wget https://bitbucket.org/winebarrel/ruby-handlersocket/get/tip.tar.gz
tar zxvf tip.tar.gz
cd winebarrel-ruby-handlersocket-c19841e47ea2/
ruby extconf.rb
cp -R /path/to/libhsclient .
make
sudo make install
```


这样安装就完成了，赶快来试试看吧。单纯查询数据的处理如下所示。

```
require 'handlersocket'

hs = HandlerSocket.new

hs.open_index(1, 'handlersocket', 'test', 'PRIMARY', 'id,data')
res = hs.execute_single(1, '= ', [2]) #  id = 读取 id = 2的数据
p res #  [0, "2", "data_2"]

hs.close
```

插入和删除数据的处理也可以像下面这样顺利地完成。

```
require 'handlersocket'

hs_write = HandlerSocket.new('localhost', 9999)

hs_write.open_index(1, 'handlersocket', 'test', 'PRIMARY', 'id,data')
hs_write.execute_insert(1, [4, 'data_4'])
hs_write.execute_delete(1, '= ', [2], 1, 0)
res2 = hs_write.execute_single(1, '> = ', [1], 10, 0)
p res2 #  [0, "3", "data_03", "4", "data_4"]

hs_write.close
```

① https：//bitbucket. org/winebarrel/ruby-handlersocket/src

虽然 HandlerSocket 使用起来有一些限制，但是基本的处理都能够完成，它还可以使 MySQL 获得像 NoSQL 数据库那样的操作体验。那么它的性能究竟如何呢？

5.2.4 HandlerSocket 的性能

我们还是使用第 4 章用到的例子，创建如下的表，然后向其中插入 2 万条数据。

```
mysql> CREATE TABLE users (
    ->   id int not null primary key,
    ->   user_id int not null,
    ->   value varchar(255) not null
    -> ) ENGINE = INNODB;
```

首先来看一下数据插入的部分，与 MySQL 的 InnoDB 和 memcached 的比较结果如下所示。

	user	system	total	real
HandlerSocket	0.010000	0.020000	0.030000	(11.038587)
InnoDB	8.060000	0.240000	8.300000	(47.546858)
memcached	1.480000	0.210000	1.690000	(3.189919)

虽然比不上 memcached，但是跟一般的 InnoDB 比起来，处理速度还是非常快的。性能验证用到的程序代码如下所示。

```
require 'handlersocket'
require 'benchmark'

LIMIT = 20000

def init
  @ hs = HandlerSocket.new('localhost', 9999)
  @ hs.open_index(1, 'handlersocket', 'users', 'PRIMARY', 'id,user_id,value')
end

def finalize
  @ hs.close
end

init
```

```
Benchmark.bm do | x|
  x.report('HandlerSocket') {
    1.upto(LIMIT) do | user_id|
      @ hs.execute_insert(1, [user_id, user_id, "value_#{user_id}"])
    end
  }
end

finalize
```

接着来看一下读取数据的部分，本例中要通过 user _id 进行查询，因此需要事先为 user _id 创建索引。

```
mysql> ALTER TAbLE users ADD INDEX user_id (user_id);
Query OK, 20000 rows affected (0.16 sec)
Records:20000 Duplicates:0 Warnings:0
```


比较结果如下所示，其中 InnoDB 开启了索引功能。

	user	system	total	real
HandlerSocket	0.090000	0.120000	0.210000	(1.033974)
InnoDB	4.150000	0.170000	4.320000	(9.032817)
memcached	4.690000	0.280000	4.970000	(8.561159)

确实还是 HandlerSocket 的处理速度更快一些。与开启索引功能的 InnoDB 和 memcached 比起来它具有压倒性的优势，能够获得与 Tokyo Tyrant 和 Redis 相仿的处理速度。性能验证用到的程序代码如下所示。

```
require 'handlersocket'
require 'benchmark'

LIMIT = 20000

def init
  @ hs = HandlerSocket.new('localhost', 9998)
  @ hs.open_index(1, 'handlersocket', 'users', 'user_id', 'id,user_id,value')
end

def finalize
```

```
  @ hs.close
end

init

Benchmark.bm do | x|
  x.report('HandlerSocket') {
    1.upto(LIMIT) do | user_id|
      user_id = 1 + rand(LIMIT)
      @ hs.execute_single(1, ' = ', [user_id])[3]
    end
  }
end

finalize
```

通过验证我们知道了 HandlerSocket 具有非常高的处理速度。而且它是以 MySQL 为基础的，用起来一点儿都不费劲。虽然它并不是很成熟，稳定性仍需观望，但据说 DeNA 公司已经开始正式使用 HandlerSocke 了 t。倘若条件合适，HandlerSocket 不失为明智之选。