翻译一篇技术评论社的文章，是讲memcached的连载。fcicq同学说这个东西很有用，希望大家喜欢。

memcached完全剖析 1- memcached的基础

发表日：2008/7/2
作者：长野雅广(Masahiro Nagano)
原文链接：http://gihyo.jp/dev/feature/01/memcached/0001

我是mixi株式会社开发部系统运营组的长野。 日常负责程序的运营。从今天开始，将分几次针对最近在Web应用的可扩展性领域 的热门话题memcached，与我公司开发部研究开发组的前坂一起， 说明其内部结构和使用。

memcached是什么？

memcached 是以LiveJournal 旗下Danga Interactive 公司的Brad Fitzpatric 为首开发的一款软件。现在已成为 mixi、hatena、Facebook、Vox、LiveJournal 等众多服务中 提高Web应用扩展性的重要因素。

许多 Web 应用都将数据保存到 RDBMS 中，应用服务器从中读取数据并在浏览器中显示。 但随着数据量的增大、访问的集中，就会出现 RDBMS 的负担加重、数据库响应恶化、 网站显示延迟等重大影响。

这时就该 memcached 大显身手了。memcached 是高性能的分布式内存缓存服务器。一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态 Web 应用的速度、 提高可扩展性。

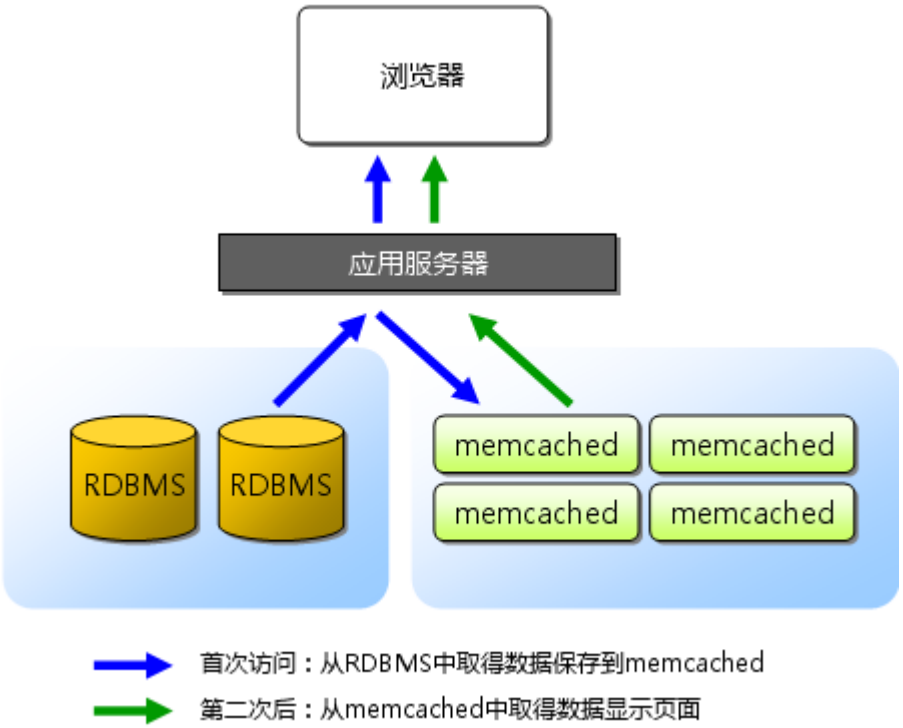


图 1 一般情况下 memcached 的用途

memcached的特征

memcached 作为高速运行的分布式缓存服务器，具有以下的特点。

- 协议简单
- 基于 libevent 的事件处理
- 内置内存存储方式
- memcached 不互相通信的分布式

协议简单

memcached 的服务器客户端通信并不使用复杂的 XML 等格式， 而使用简单的基于文本行的协议。因此，通过 telnet 也能在 memcached 上保存数据、取得数据。下面是例子。

```
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
set foo 0 0 3      （保存命令）
bar                （数据）
STORED             （结果）
get foo            （取得命令）
VALUE foo 0 3      （数据）
bar                （数据）
```

协议文档位于 memcached 的源代码内，也可以参考以下的 URL。

- http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt

基于 libevent 的事件处理

libevent 是个程序库，它将 Linux 的 epoll、BSD 类操作系统的 kqueue 等事件处理功能 封装成统一的接口。即使对服务器的连接数增加，也能发挥 O(1)的性能。 memcached 使用这个 libevent 库，因此能在 Linux、BSD、Solaris 等操作系统上发挥其高性能。 关于事件处理这里就不再详细介绍，可以参考 Dan Kegel 的 The C10K Problem。

- libevent: http://www.monkey.org/~provos/libevent/
- The C10K Problem: http://www.kegel.com/c10k.html

内置内存存储方式

为了提高性能，memcached 中保存的数据都存储在 memcached 内置的内存存储空间中。 由于数据仅存在于内存中，因此重启 memcached、重启操作系统会导致全部数据消失。 另外，内容容量达到指定值之后，就基于 LRU(Least Recently Used)算法自动删除不使用的缓存。 memcached 本身是为缓存而设计的服务器，因此并没有过多考虑数据的永久性问题。 关于内存存储的详细信息，本连载的第二讲以后前坂会进行介绍，请届时参考。

memcached 不互相通信的分布式

memcached 尽管是“分布式”缓存服务器，但服务器端并没有分布式功能。 各个 memcached 不会互相通信以共享信息。那么，怎样进行分布式呢？ 这完全取决于客户端的实现。本连载也将介绍 memcached 的分布式。

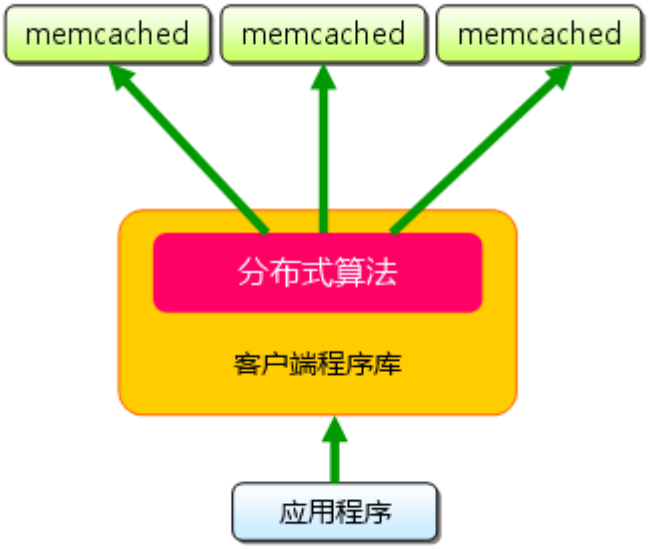


图 2 memcached 的分布式

接下来简单介绍一下 memcached 的使用方法。

安装memcached

memcached 的安装比较简单，这里稍加说明。

memcached 支持许多平台。

- Linux
- FreeBSD
- Solaris (memcached 1.2.5 以上版本)
- Mac OS X

另外也能安装在 Windows 上。这里使用 Fedora Core 8 进行说明。

memcached 的安装

运行 memcached 需要本文开头介绍的 libevent 库。Fedora 8 中有现成的 rpm 包，通过 yum 命令安装即可。

```
$ sudo yum install libevent libevent-devel
```

memcached 的源代码可以从 memcached 网站上下载。本文执笔时的最新版本为 1.2.5。 Fedora 8 虽然也包含了 memcached 的 rpm，但版本比较老。因为源代码安装并不困难， 这里就不使用 rpm 了。

- **下载memcached：**http://www.danga.com/memcached/download.bml

memcached 安装与一般应用程序相同，configure、make、make install 就行了。

```
$ wget http://www.danga.com/memcached/dist/memcached-1.2.5.tar.gz
$ tar zxf memcached-1.2.5.tar.gz
$ cd memcached-1.2.5
$ ./configure
$ make
$ sudo make install
```

默认情况下 memcached 安装到/usr/local/bin 下。

memcached 的启动

从终端输入以下命令，启动 memcached。

```
$ /usr/local/bin/memcached -p 11211 -m 64m -vv
slab class   1: chunk size     88 perslab 11915
slab class   2: chunk size    112 perslab  9362
slab class   3: chunk size    144 perslab  7281
中间省略
slab class  38: chunk size 391224 perslab     2
slab class  39: chunk size 489032 perslab     2
<23 server listening
<24 send buffer was 110592, now 268435456
<24 server listening (udp)
<24 server listening (udp)
<24 server listening (udp)
<24 server listening (udp)
```

这里显示了调试信息。这样就在前台启动了 memcached，监听 TCP 端口 11211 最大内存使用量为 64M。调试信息的内容大部分是关于存储的信息， 下次连载时具体说明。

作为 daemon 后台启动时，只需

```
$ /usr/local/bin/memcached -p 11211 -m 64m -d
```

这里使用的 memcached 启动选项的内容如下。

选项 说明
-p　使用的 TCP 端口。默认为 11211
-m　最大内存大小。默认为 64M
-vv　用 very vrebose 模式启动，调试信息和错误输出到控制台
-d　作为 daemon 在后台启动

上面四个是常用的启动选项，其他还有很多，通过

$ /usr/local/bin/memcached -h

命令可以显示。许多选项可以改变 memcached 的各种行为， 推荐读一读。

用客户端连接

许多语言都实现了连接 memcached 的客户端，其中以 Perl、PHP 为主。 仅仅 memcached 网站上列出的语言就有

- Perl
- PHP
- Python
- Ruby
- C#
- C/C++
- Lua

等等。

- **memcached客户端API：**http://www.danga.com/memcached/apis.bml

这里介绍通过 mixi 正在使用的 Perl 库链接 memcached 的方法。

使用 Cache::Memcached

Perl 的 memcached 客户端有

- Cache::Memcached
- Cache::Memcached::Fast
- Cache::Memcached::libmemcached

等几个 CPAN 模块。这里介绍的 Cache::Memcached 是 memcached 的作者 Brad Fitzpatric 的作品， 应该算是 memcached 的客户端中应用最为广泛的模块了。

- Cache::Memcached - search.cpan.org:
  http://search.cpan.org/dist/Cache-Memcached/

使用 Cache::Memcached 连接 memcached

下面的源代码为通过 Cache::Memcached 连接刚才启动的 memcached 的例子。

```
#!/usr/bin/perl

use strict;
use warnings;
use Cache::Memcached;

my $key = "foo";
my $value = "bar";
my $expires = 3600; # 1 hour
my $memcached = Cache::Memcached->new({
    servers => ["127.0.0.1:11211"],
    compress_threshold => 10_000
});

$memcached->add($key, $value, $expires);
my $ret = $memcached->get($key);
print "$ret\n";
```

在这里，为 Cache::Memcached 指定了 memcached 服务器的 IP 地址和一个选项，以生成实例。 Cache::Memcached 常用的选项如下所示。

选项	说明
servers	用数组指定 memcached 服务器和端口
compress_threshold	数据压缩时使用的值
namespace	指定添加到键的前缀

另外，Cache::Memcached 通过 Storable 模块可以将 Perl 的复杂数据序列化之后再保存， 因此散列、数组、对象等都可以直接保存到 memcached 中。

保存数据

向 memcached 保存数据的方法有

- add
- replace
- set

它们的使用方法都相同：

```
my $add = $memcached->add( '键', '值', '期限' );
my $replace = $memcached->replace( '键', '值', '期限' );
my $set = $memcached->set( '键', '值', '期限' );
```

向 memcached 保存数据时可以指定期限(秒)。不指定期限时，memcached 按照 LRU 算法保存数据。 这三个方法的区别如下：

选项	说明
add	仅当存储空间中不存在键相同的数据时才保存
replace	仅当存储空间中存在键相同的数据时才保存
set	与 add 和 replace 不同，无论何时都保存

获取数据

获取数据可以使用 get 和 get_multi 方法。

```
my $val = $memcached->get('键');
my $val = $memcached->get_multi('键 1', '键 2', '键 3', '键 4', '键 5');
```

一次取得多条数据时使用 get_multi。get_multi 可以非同步地同时取得多个键值， 其速度要比循环调用 get 快数十倍。

删除数据

删除数据使用 delete 方法，不过它有个独特的功能。

```
$memcached->delete('键', '阻塞时间(秒)');
```

删除第一个参数指定的键的数据。第二个参数指定一个时间值，可以禁止使用同样的键保存新数据。 此功能可以用于防止缓存数据的不完整。但是要注意，**set 函数忽视该阻塞，照常保存数据**

增一和减一操作

可以将 memcached 上特定的键值作为计数器使用。

```
my $ret = $memcached->incr('键');
$memcached->add('键', 0) unless defined $ret;
```

增一和减一是原子操作，但未设置初始值时，不会自动赋成 0。因此， 应当进行错误检查，必要时加入初始化操作。而且，服务器端也不会对 超过 2³²时的行为进行检查。

总结

这次简单介绍了 memcached，以及它的安装方法、Perl 客户端 Cache::Memcached 的用法。 只要知道，memcached 的使用方法十分简单就足够了。

下次由前坂来说明 memcached 的内部结构。了解 memcached 的内部构造， 就能知道如何使用 memcached 才能使 Web 应用的速度更上一层楼。 欢迎继续阅读下一章。

Memcached完全剖析 2-Slab Allocation

发表日：2008/7/9
作者：前坂徹(Toru Maesaka)
原文链接：http://gihyo.jp/dev/feature/01/memcached/0002

我是mixi株式会社研究开发组的前坂徹。上次的文章介绍了memcached是分布式的高速缓存服务器。 本次将介绍memcached的内部构造的实现方式，以及内存的管理方式。 另外，memcached的内部构造导致的弱点也将加以说明。

Slab Allocation机制：整理内存以便重复使用

最近的 memcached 默认情况下采用了名为 Slab Allocator 的机制分配、管理内存。 在该机制出现以前，内存的分配是通过对所有记录简单地进行 malloc 和 free 来进行的。 但是，这种方式会导致内存碎片，加重操作系统内存管理器的负担，最坏的情况下， 会导致操作系统比 memcached 进程本身还慢。Slab Allocator 就是为解决该问题而诞生的。

下面来看看 Slab Allocator 的原理。下面是 memcached 文档中的 slab allocator 的目标：

the primary goal of the slabs subsystem in memcached was to eliminate memory fragmentation issues totally by using fixed-size memory chunks coming from a few predetermined size classes.

也就是说，Slab Allocator 的基本原理是按照预先规定的大小，将分配的内存分割成特定长度的块， 以完全解决内存碎片问题。

Slab Allocation 的原理相当简单。将分配的内存分割成各种尺寸的块(chunk)，并把尺寸相同的块分成组（chunk 的集合）（图 1）。

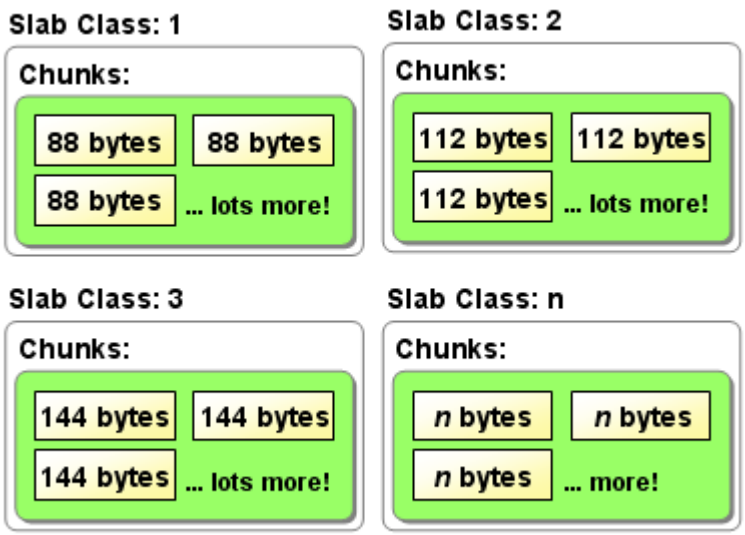


图 1 Slab Allocation 的构造图

而且，slab allocator 还有重复使用已分配的内存的目的。 也就是说，分配到的内存不会释放，而是重复利用。

Slab Allocation 的主要术语

Page

分配给 Slab 的内存空间，默认是 1MB。分配给 Slab 之后根据 slab 的大小切分成 chunk。

Chunk

用于缓存记录的内存空间。

Slab Class

特定大小的 chunk 的组。

在Slab中缓存记录的原理

下面说明 memcached 如何针对客户端发送的数据选择 slab 并缓存到 chunk 中。

memcached 根据收到的数据的大小，选择最适合数据大小的 slab（图 2）。memcached 中保存着 slab 内空闲 chunk 的列表，根据该列表选择 chunk， 然后将数据缓存于其中。

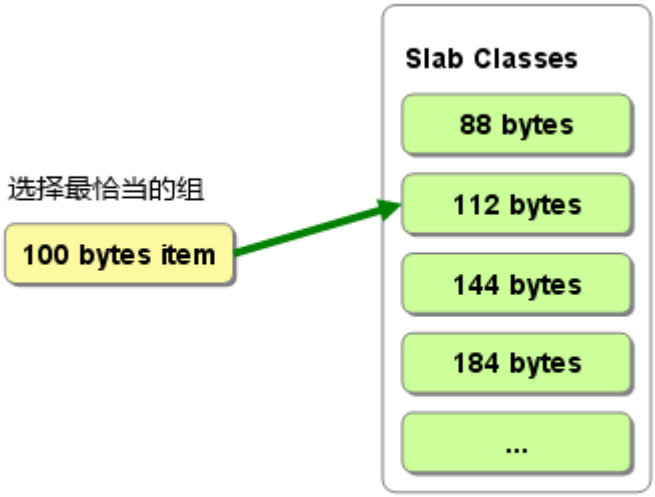


图 2 选择存储记录的组的方法

实际上，Slab Allocator 也是有利也有弊。下面介绍一下它的缺点。

Slab Allocator的缺点

Slab Allocator 解决了当初的内存碎片问题，但新的机制也给 memcached 带来了新的问题。

这个问题就是，由于分配的是特定长度的内存，因此无法有效利用分配的内存。例如，将 100 字节的数据缓存到 128 字节的 chunk 中，剩余的 28 字节就浪费了（图 3）。

图 3 chunk 空间的使用

对于该问题目前还没有完美的解决方案，但在文档中记载了比较有效的解决方案。

The most efficient way to reduce the waste is to use a list of size classes that closely matches (if that's at all possible) common sizes of objects that the clients of this particular installation of memcached are likely to store.

就是说，如果预先知道客户端发送的数据的公用大小，或者仅缓存大小相同的数据的情况下， 只要使用适合数据大小的组的列表，就可以减少浪费。

但是很遗憾，现在还不能进行任何调优，只能期待以后的版本了。 但是，我们可以调节 slab class 的大小的差别。 接下来说明 growth factor 选项。

使用Growth Factor进行调优

memcached 在启动时指定 Growth Factor 因子（通过-f 选项）， 就可以在某种程度上控制 slab 之间的差异。默认值为 1.25。 但是，在该选项出现之前，这个因子曾经固定为 2，称为“powers of 2”策略。

让我们用以前的设置，以 verbose 模式启动 memcached 试试看：

```
$ memcached -f 2 -vv
```

下面是启动后的 verbose 输出：

```
slab class   1: chunk size     128 perslab  8192
slab class   2: chunk size     256 perslab  4096
slab class   3: chunk size     512 perslab  2048
slab class   4: chunk size    1024 perslab  1024
slab class   5: chunk size    2048 perslab   512
slab class   6: chunk size    4096 perslab   256
```

```
slab class   7: chunk size   8192 perslab   128
slab class   8: chunk size  16384 perslab    64
slab class   9: chunk size  32768 perslab    32
slab class  10: chunk size  65536 perslab    16
slab class  11: chunk size 131072 perslab     8
slab class  12: chunk size 262144 perslab     4
slab class  13: chunk size 524288 perslab     2
```

可见，从 128 字节的组开始，组的大小依次增大为原来的 2 倍。 这样设置的问题是，slab 之间的差别比较大，有些情况下就相当浪费内存。 因此，为尽量减少内存浪费，两年前追加了 growth factor 这个选项。

来看看现在的默认设置（f=1.25）时的输出（篇幅所限，这里只写到第 10 组）：

```
slab class   1: chunk size     88 perslab 11915
slab class   2: chunk size    112 perslab  9362
slab class   3: chunk size    144 perslab  7281
slab class   4: chunk size    184 perslab  5698
slab class   5: chunk size    232 perslab  4519
slab class   6: chunk size    296 perslab  3542
slab class   7: chunk size    376 perslab  2788
slab class   8: chunk size    472 perslab  2221
slab class   9: chunk size    592 perslab  1771
slab class  10: chunk size    744 perslab  1409
```

可见，组间差距比因子为 2 时小得多，更适合缓存几百字节的记录。 从上面的输出结果来看，可能会觉得有些计算误差， 这些误差是为了保持字节数的对齐而故意设置的。

将 memcached 引入产品，或是直接使用默认值进行部署时， 最好是重新计算一下数据的预期平均长度，调整 growth factor， 以获得最恰当的设置。内存是珍贵的资源，浪费就太可惜了。

接下来介绍一下如何使用 memcached 的 stats 命令查看 slabs 的利用率等各种各样的信息。

查看memcached的内部状态

memcached 有个名为 stats 的命令，使用它可以获得各种各样的信息。 执行命令的方法很多，用 telnet 最为简单：

```
$ telnet 主机名 端口号
```

连接到 memcached 之后，输入 stats 再按回车，即可获得包括资源利用率在内的各种信息。 此外，输入"stats slabs"或"stats items"还可以获得关于缓存记录的信息。 结束程序请输入 quit。

这些命令的详细信息可以参考 memcached 软件包内的 protocol.txt 文档。

```
$ telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
stats
STAT pid 481
STAT uptime 16574
STAT time 1213687612
STAT version 1.2.5
STAT pointer_size 32
STAT rusage_user 0.102297
STAT rusage_system 0.214317
STAT curr_items 0
STAT total_items 0
STAT bytes 0
STAT curr_connections 6
STAT total_connections 8
STAT connection_structures 7
STAT cmd_get 0
STAT cmd_set 0
STAT get_hits 0
STAT get_misses 0
STAT evictions 0
STAT bytes_read 20
STAT bytes_written 465
STAT limit_maxbytes 67108864
STAT threads 4
END
quit
```

另外，如果安装了 libmemcached 这个面向 C/C++语言的客户端库，就会安装 memstat 这个命令。 使用方法很简单，可以用更少的步骤获得与 telnet 相同的信息，还能一次性从多台服务器获得信息。

```
$ memstat --servers=server1,server2,server3,...
```

libmemcached 可以从下面的地址获得：

- http://tangent.org/552/libmemcached.html

查看slabs的使用状况

使用 memcached 的创造着 Brad 写的名为 memcached-tool 的 Perl 脚本，可以方便地获得 slab 的使用情况（它将 memcached 的返回值整理成容易阅读的格式）。可以从下面的地址获得脚本：

- http://code.sixapart.com/svn/memcached/trunk/server/scripts/memcached-tool

使用方法也极其简单：

$ memcached-tool 主机名:端口 选项

查看 slabs 使用状况时无需指定选项，因此用下面的命令即可：

$ memcached-tool 主机名:端口

获得的信息如下所示：

```
 #  Item_Size   Max_age  1MB_pages Count   Full?
 1     104 B  1394292 s    1215 12249628    yes
 2     136 B  1456795 s      52  400919     yes
 3     176 B  1339587 s      33  196567     yes
 4     224 B  1360926 s     109  510221     yes
 5     280 B  1570071 s      49  183452     yes
 6     352 B  1592051 s      77  229197     yes
 7     440 B  1517732 s      66  157183     yes
 8     552 B  1460821 s      62  117697     yes
 9     696 B  1521917 s     143  215308     yes
10     872 B  1695035 s     205  246162     yes
11     1.1 kB 1681650 s     233  221968     yes
12     1.3 kB 1603363 s     241  183621     yes
13     1.7 kB 1634218 s      94   57197     yes
14     2.1 kB 1695038 s      75   36488     yes
15     2.6 kB 1747075 s      65   25203     yes
16     3.3 kB 1760661 s      78   24167     yes
```

各列的含义为：

列	含义
#	slab class 编号
Item_Size	Chunk 大小
Max_age	LRU 内最旧的记录的生存时间
1MB_pages	分配给 Slab 的页数

Count　　Slab 内的记录数
Full?　　Slab 内是否含有空闲 chunk

从这个脚本获得的信息对于调优非常方便，强烈推荐使用。

内存存储的总结

本次简单说明了 memcached 的缓存机制和调优方法。希望读者能理解 memcached 的内存管理原理及其优缺点。

下次将继续说明 LRU 和 Expire 等原理，以及 memcached 的最新发展方向—— 可扩充体系（pluggable architecher））。

memcached全面剖析 3-memcached的删除机制和发展方向

发表日：2008/7/16
作者：前坂徹(Toru Maesaka)
原文链接：http://gihyo.jp/dev/feature/01/memcached/0003

memcached 是缓存，所以数据不会永久保存在服务器上，这是向系统中引入 memcached 的前提。 本次介绍 memcached 的数据删除机制，以及 memcached 的最新发展方向——二进制协议（Binary Protocol） 和外部引擎支持。

memcached在数据删除方面有效利用资源

数据不会真正从 memcached 中消失

上次介绍过， memcached不会释放已分配的内存。记录超时后，客户端就无法再看见该记录（invisible，透明）， 其存储空间即可重复使用。

Lazy Expiration

memcached 内部不会监视记录是否过期，而是在 get 时查看记录的时间戳，检查记录是否过期。 这种技术被称为 lazy（惰性）expiration。因此，memcached 不会在过期监视上耗费 CPU 时间。

LRU：从缓存中有效删除数据的原理

memcached会优先使用已超时的记录的空间，但即使如此，也会发生追加新记录时空间不足的情况， 此时就要使用名为 Least Recently Used（LRU）机制来分配空间。 顾名思义，这是删除“最近最少使用”的记录的机制。 因此，当 memcached的内存空间不足时（无法从slab class 获取到新的空间时），就从最近未被使用的记录中搜索，并将其空间分配给新的记录。 从缓存的实用角度来看，该模型十分理想。

不过，有些情况下 LRU 机制反倒会造成麻烦。memcached 启动时通过“-M”参数可以禁止 LRU，如下所示：

```
$ memcached -M -m 1024
```

启动时必须注意的是，小写的“-m”选项是用来指定最大内存大小的。不指定具体数值则使用默认值 64MB。

指定“-M”参数启动后，内存用尽时 memcached 会返回错误。 话说回来，memcached 毕竟不是存储器，而是缓存，所以推荐使用 LRU。

memcached的最新发展方向

memcached 的 roadmap 上有两个大的目标。一个是二进制协议的策划和实现，另一个是外部引擎的加载功能。

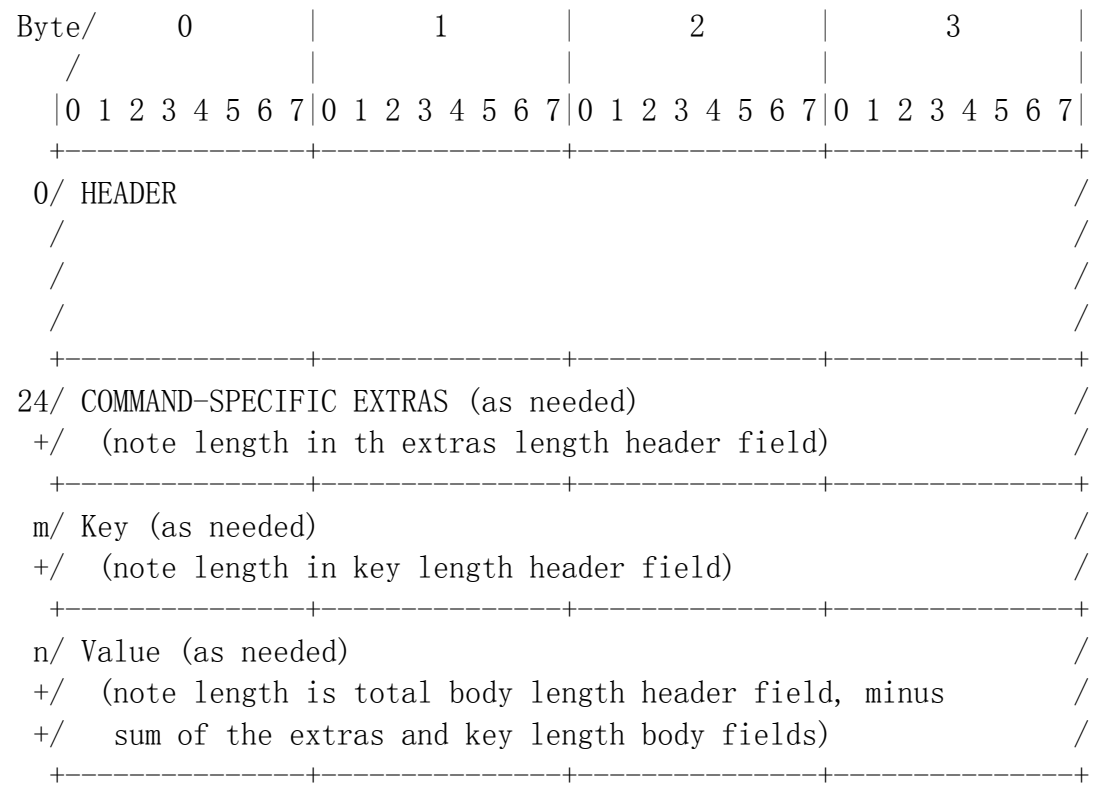
关于二进制协议

使用二进制协议的理由是它不需要文本协议的解析处理，使得原本高速的 memcached 的性能更上一层楼， 还能减少文本协议的漏洞。目前已大部分实现，开发用的代码库中已包含了该功能。 memcached 的下载页面上有代码库的链接。

- http://danga.com/memcached/download.bml

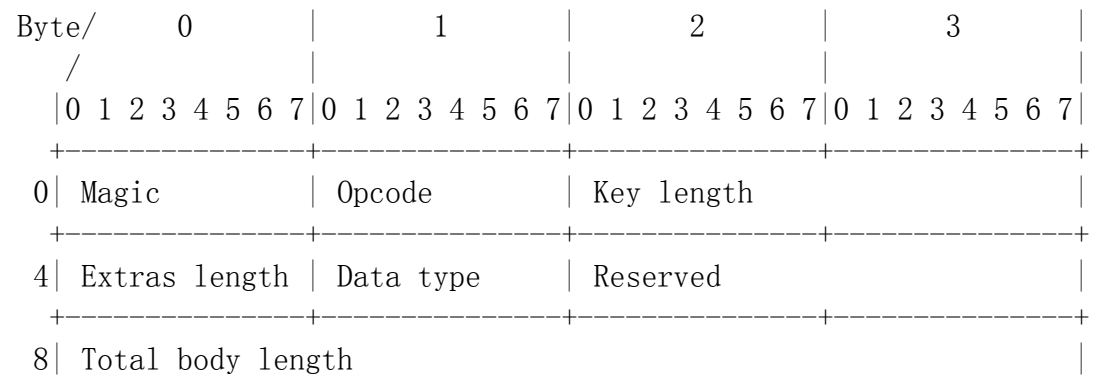
二进制协议的格式

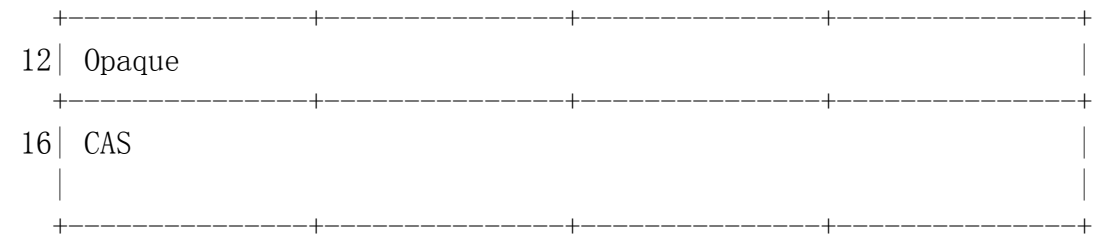
协议的包为 24 字节的帧，其后面是键和无结构数据（Unstructured Data）。 实际的格式如下（引自协议文档）：



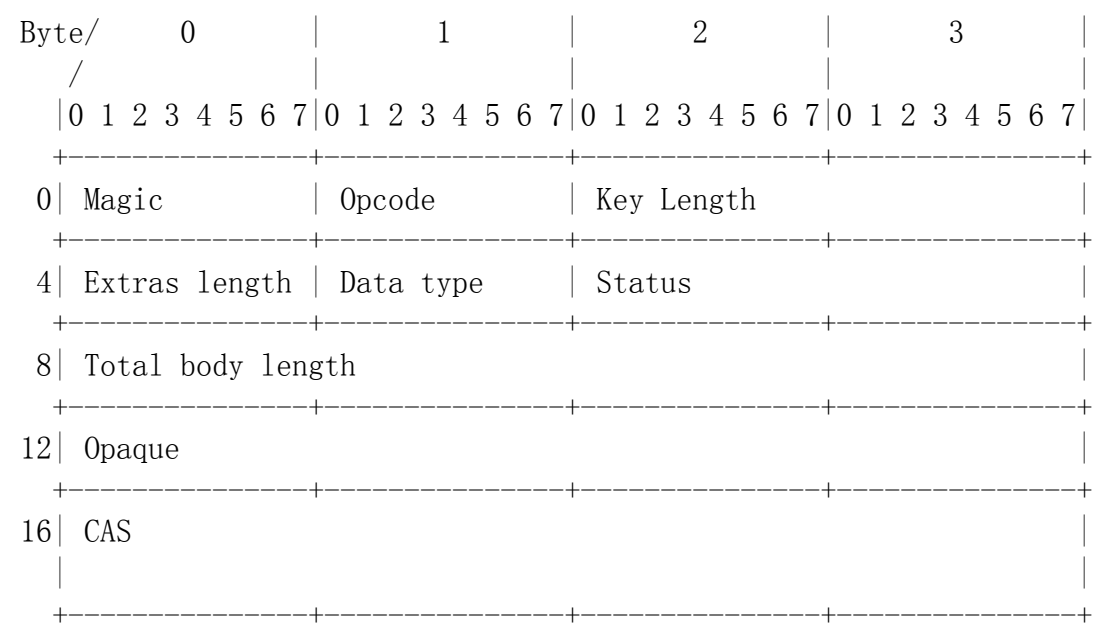
如上所示，包格式十分简单。需要注意的是，占据了 16 字节的头部(HEADER)分为 请求头（Request Header）和响应头（Response Header）两种。 头部中包含了表示包的有效性的 Magic 字节、命令种类、键长度、值长度等信息，格式如下：

Request Header





Response Header



如希望了解各个部分的详细内容，可以 checkout 出 memcached 的二进制协议的代码树， 参考其中的 docs 文件夹中的 protocol_binary.txt 文档。

HEADER 中引人注目的地方

看到 HEADER 格式后我的感想是，键的上限太大了！现在的 memcached 规格中，键长度最大为 250 字节， 但二进制协议中键的大小用 2 字节表示。因此，理论上最大可使用 65536 字节（2¹⁶）长的键。 尽管 250 字节以上的键并不会太常用，二进制协议发布之后就可以使用巨大的键了。

二进制协议从下一版本 1.3 系列开始支持。

外部引擎支持

我去年曾经试验性地将 memcached 的存储层改造成了可扩展的（pluggable）。

- http://alpha.mixi.co.jp/blog/?p=129

MySQL的Brian Aker看到这个改造之后，就将代码发到了memcached的邮件列表。memcached的开发者也十分感兴趣，就放到了roadmap中。现在由我和 memcached 的开发者Trond Norbye协同开发（规格设计、实现和测试）。 和国外协同开发时时差是个大问题，但抱着相同的愿景， 最后终于可以将可扩展架构的原型公布了。 代码库可以从memcached的下载页面 上访问。

外部引擎支持的必要性

世界上有许多 memcached 的派生软件，其理由是希望永久保存数据、实现数据冗余等， 即使牺牲一些性能也在所不惜。我在开发 memcached 之前，在 mixi 的研发部也曾经 考虑过重新发明 memcached。

外部引擎的加载机制能封装 memcached 的网络功能、事件处理等复杂的处理。因此，现阶段通过强制手段或重新设计等方式使 memcached 和存储引擎合作的困难就会烟消云散，尝试各种引擎就会变得轻而易举了。

简单 API 设计的成功的关键

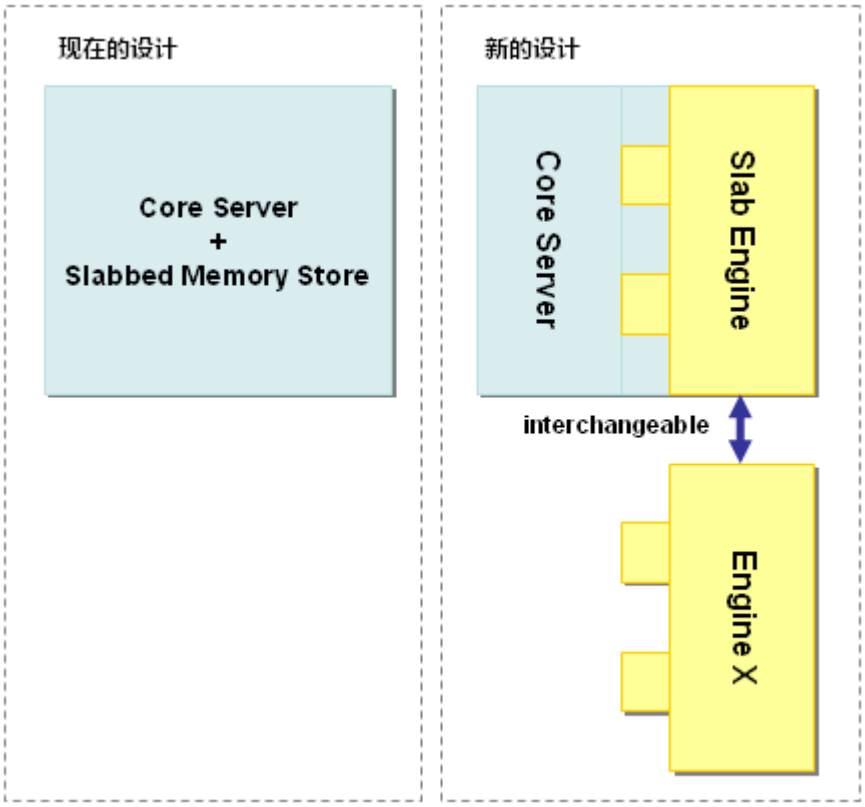
该项目中我们最重视的是 API 设计。函数过多，会使引擎开发者感到麻烦； 过于复杂，实现引擎的门槛就会过高。因此，最初版本的接口函数只有 13 个。 具体内容限于篇幅，这里就省略了，仅说明一下引擎应当完成的操作：

- 引擎信息（版本等）
- 引擎初始化
- 引擎关闭
- 引擎的统计信息
- 在容量方面，测试给定记录能否保存
- 为 item（记录）结构分配内存
- 释放 item（记录）的内存
- 删除记录
- 保存记录
- 回收记录
- 更新记录的时间戳
- 数学运算处理
- 数据的 flush

对详细规格有兴趣的读者，可以 checkout engine 项目的代码，阅读器中的 engine.h。

重新审视现在的体系

memcached 支持外部存储的难点是，网络和事件处理相关的代码（核心服务器）与 内存存储的代码紧密关联。这种现象也称为 tightly coupled（紧密耦合）。必须将内存存储的代码从核心服务器中独立出来，才能灵活地支持外部引擎。因此，基于我们设计的 API，memcached 被重构成下面的样子：



重构之后，我们与 1.2.5 版、二进制协议支持版等进行了性能对比，证实了它不会造成性能影响。

在考虑如何支持外部引擎加载时，让 memcached 进行并行控制（concurrency control）的方案是最为容易的， 但是对于引擎而言，并行控制正是性能的真谛，因此我们采用了将多线程支持完全交给引擎的设计方案。

以后的改进，会使得 memcached 的应用范围更为广泛。

总结

本次介绍了 memcached 的超时原理、内部如何删除数据等，在此之上又介绍了二进制协议和 外部引擎支持等 memcached 的最新发展方向。这些功能要到 1.3 版才会支持，敬请期待！

这是我在本连载中的最后一篇。感谢大家阅读我的文章！

下次由长野来介绍 memcached 的应用知识和应用程序兼容性等内容。

memcached全面剖析 4. memcached的分布式算法

我是Mixi的长野。 第 2 次、 第 3 次 由前坂介绍了memcached的内部情况。本次不再介绍memcached的内部结构， 开始介绍memcached的分布式。

memcached的分布式

正如第 1 次中介绍的那样， memcached虽然称为“分布式”缓存服务器，但服务器端并没有“分布式”功能。 服务器端仅包括 第 2 次、 第 3 次 前坂介绍的内存存储功能，其实现非常简单。 至于memcached的分布式，则是完全由客户端程序库实现的。 这种分布式是memcached的最大特点。

memcached 的分布式是什么意思？

这里多次使用了“分布式”这个词，但并未做详细解释。 现在开始简单地介绍一下其原理，各个客户端的实现基本相同。

下面假设 memcached 服务器有 node1～node3 三台， 应用程序要保存键名为“tokyo”“kanagawa”“chiba”“saitama”“gunma” 的数据。

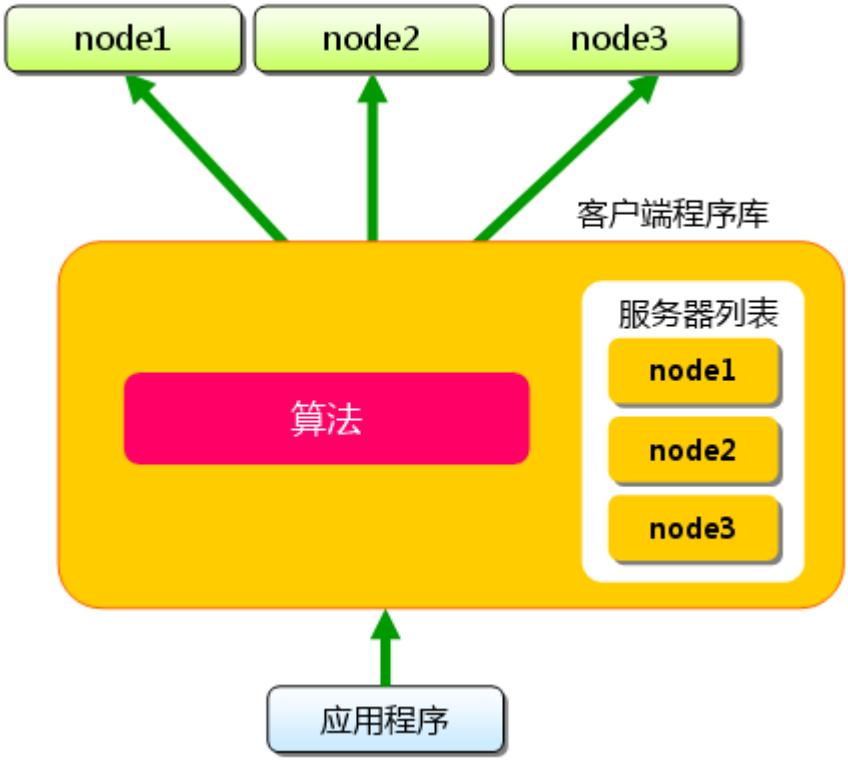


图 1 分布式简介：准备

首先向 memcached 中添加“tokyo”。将“tokyo”传给客户端程序库后， 客户端实现的算法就会根据“键”来决定保存数据的 memcached 服务器。 服务器选定后，即命令它保存“tokyo”及其值。

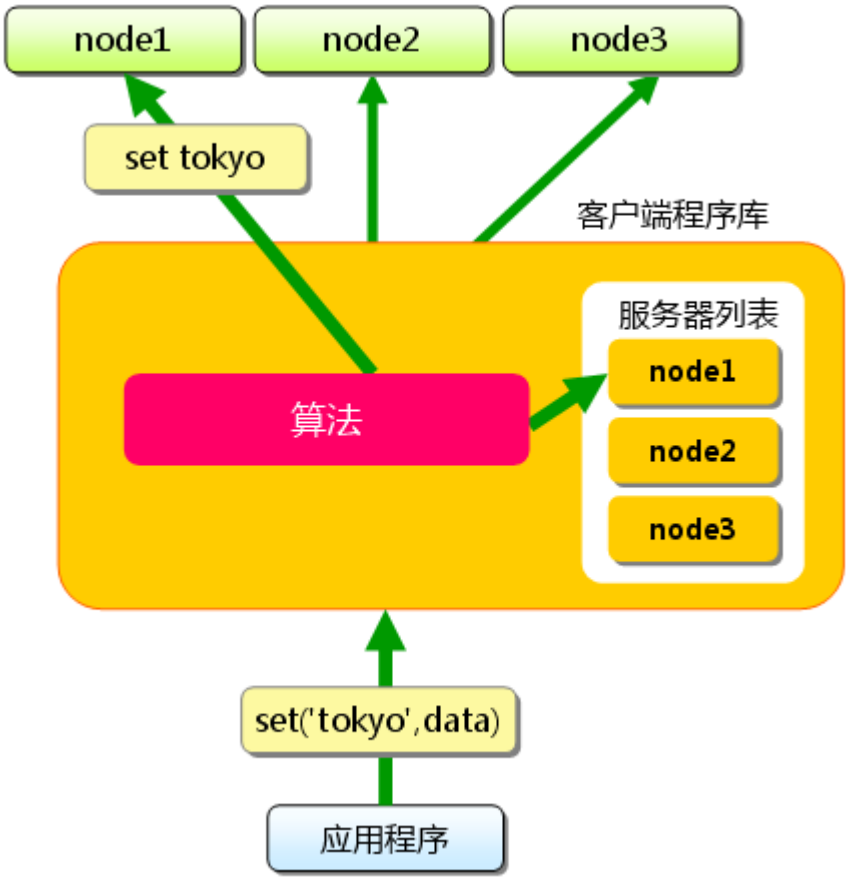


图 2 分布式简介：添加时

同样，“kanagawa”“chiba”“saitama”“gunma”都是先选择服务器再保存。

接下来获取保存的数据。获取时也要将要获取的键“tokyo”传递给函数库。 函数库通过与数据保存时相同的算法，根据“键”选择服务器。 使用的算法相同，就能选中与保存时相同的服务器，然后发送 get 命令。 只要数据没有因为某些原因被删除，就能获得保存的值。

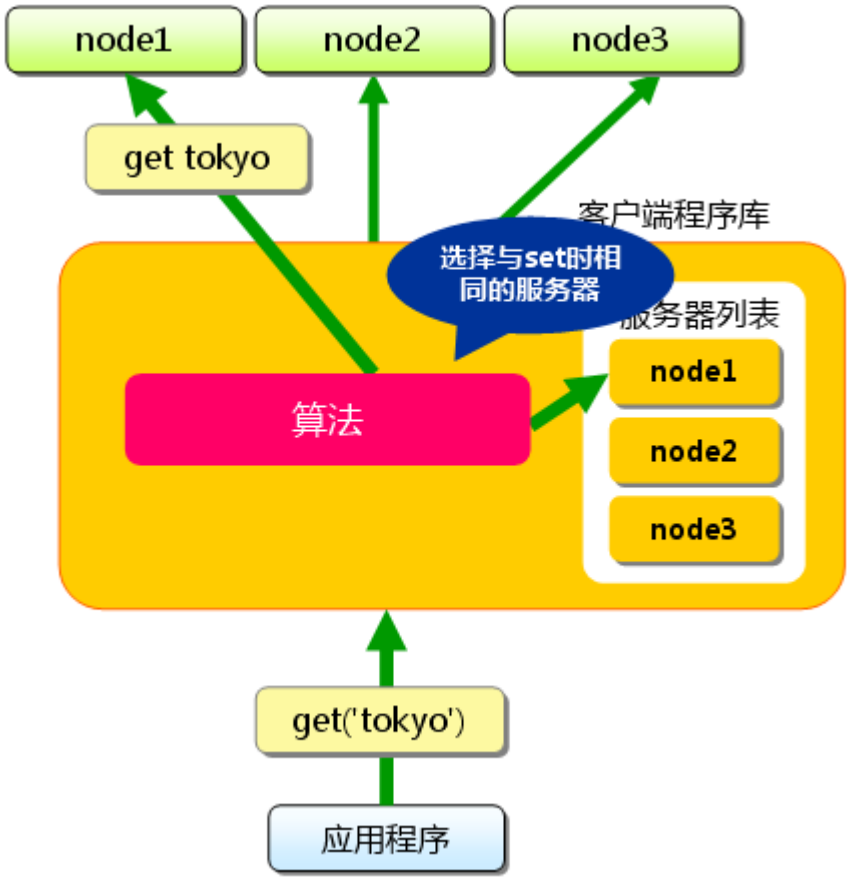


图 3 分布式简介：获取时

这样，将不同的键保存到不同的服务器上，就实现了 memcached 的分布式。memcached 服务器增多后，键就会分散，即使一台 memcached 服务器发生故障 无法连接，也不会影响其他的缓存，系统依然能继续运行。

接下来介绍第 1 次 中提到的Perl客户端函数库Cache::Memcached实现的分布式方法。

Cache::Memcached的分布式方法

Perl 的 memcached 客户端函数库 Cache::Memcached 是 memcached 的作者 Brad Fitzpatrick 的作品，可以说是原装的函数库了。

- Cache::Memcached - search.cpan.org

该函数库实现了分布式功能，是 memcached 标准的分布式方法。

根据余数计算分散

Cache::Memcached 的分布式方法简单来说，就是“根据服务器台数的余数进行分散”。 求得键的整数哈希值，再除以服务器台数，根据其余数来选择服务器。

下面将 Cache::Memcached 简化成以下的 Perl 脚本来进行说明。

```
use strict;
use warnings;
use String::CRC32;

my @nodes = ('node1','node2','node3');
my @keys = ('tokyo', 'kanagawa', 'chiba', 'saitama', 'gunma');

foreach my $key (@keys) {
    my $crc = crc32($key);              # CRC 值
    my $mod = $crc % ( $#nodes + 1 );
    my $server = $nodes[ $mod ];        # 根据余数选择服务器
    printf "%s => %s\n", $key, $server;
}
```

Cache::Memcached 在求哈希值时使用了 CRC。

- String::CRC32 - search.cpan.org

首先求得字符串的 CRC 值，根据该值除以服务器节点数目得到的余数决定服务器。 上面的代码执行后输入以下结果：

```
tokyo       => node2
kanagawa => node3
chiba       => node2
saitama    => node1
gunma      => node1
```

根据该结果，“tokyo”分散到 node2，“kanagawa”分散到 node3 等。 多说一句，当选择的服务器无法连接时，Cache::Memcached 会将连接次数 添加到键之后，再次计算哈希值并尝试连接。这个动作称为 rehash。 不希望 rehash 时可以在生成 Cache::Memcached 对象时指定“rehash => 0”选项。

根据余数计算分散的缺点

余数计算的方法简单，数据的分散性也相当优秀，但也有其缺点。 那就是当添加或移除服务器时，缓存重组的代价相当巨大。 添加服务器后，余数就会产生巨变，这样就无法获取与保存时相同的服务器，从而影响缓存的命中率。用 Perl 写段代码来验证其代价。

```
use strict;
use warnings;
use String::CRC32;

my @nodes = @ARGV;
my @keys = ('a'..'z');
my %nodes;

foreach my $key ( @keys ) {
    my $hash = crc32($key);
    my $mod = $hash % ( $#nodes + 1 );
    my $server = $nodes[ $mod ];
    push @{ $nodes{ $server } }, $key;
}

foreach my $node ( sort keys %nodes ) {
    printf "%s: %s\n", $node,  join ",", @{ $nodes{$node} };
}
```

这段 Perl 脚本演示了将“a”到“z”的键保存到 memcached 并访问的情况。将其保存为 mod.pl 并执行。

首先，当服务器只有三台时：

```
$ mod.pl node1 node2 nod3
node1: a,c,d,e,h,j,n,u,w,x
node2: g,i,k,l,p,r,s,y
node3: b,f,m,o,q,t,v,z
```

结果如上，node1 保存 a、c、d、e……，node2 保存 g、i、k……， 每台服务器都保存了 8 个到 10 个数据。

接下来增加一台 memcached 服务器。

```
$ mod.pl node1 node2 node3 node4
node1: d,f,m,o,t,v
node2: b,i,k,p,r,y
node3: e,g,l,n,u,w
node4: a,c,h,j,q,s,x,z
```

添加了 node4。可见，只有 d、i、k、p、r、y 命中了。像这样，添加节点后 键分散到的服务器会发生巨大变化。26 个键中只有六个在访问原来的服务器， 其他的全都移到了其他服务器。命中率降低到 23%。在 Web 应用程序中使用 memcached 时， 在添加 memcached 服务器的瞬间缓存效率会大幅度下降，负载会集中到数据库服务器上， 有可能会发生无法提供正常服务的情况。

mixi 的 Web 应用程序运用中也有这个问题，导致无法添加 memcached 服务器。但由于使用了新的分布式方法，现在可以轻而易举地添加 memcached 服务器了。这种分布式方法称为 Consistent Hashing。

Consistent Hashing

关于 Consistent Hashing 的思想，mixi 株式会社的开发 blog 等许多地方都介绍过， 这里只简单地说明一下。

- mixi Engineers' Blog – スマートな分散で快適キャッシュライフ
- ConsistentHashing – コンシステント ハッシュ法

Consistent Hashing 的简单说明

Consistent Hashing如下所示：首先求出memcached服务器（节点）的哈希值， 并将其配置到 0～2^{32}的圆（continuum）上。 然后用同样的方法求出存储数据的键的哈希值，并映射到圆上。 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。 如果超过 2^{32}仍然找不到服务器，就会保存到第一台memcached服务器上。

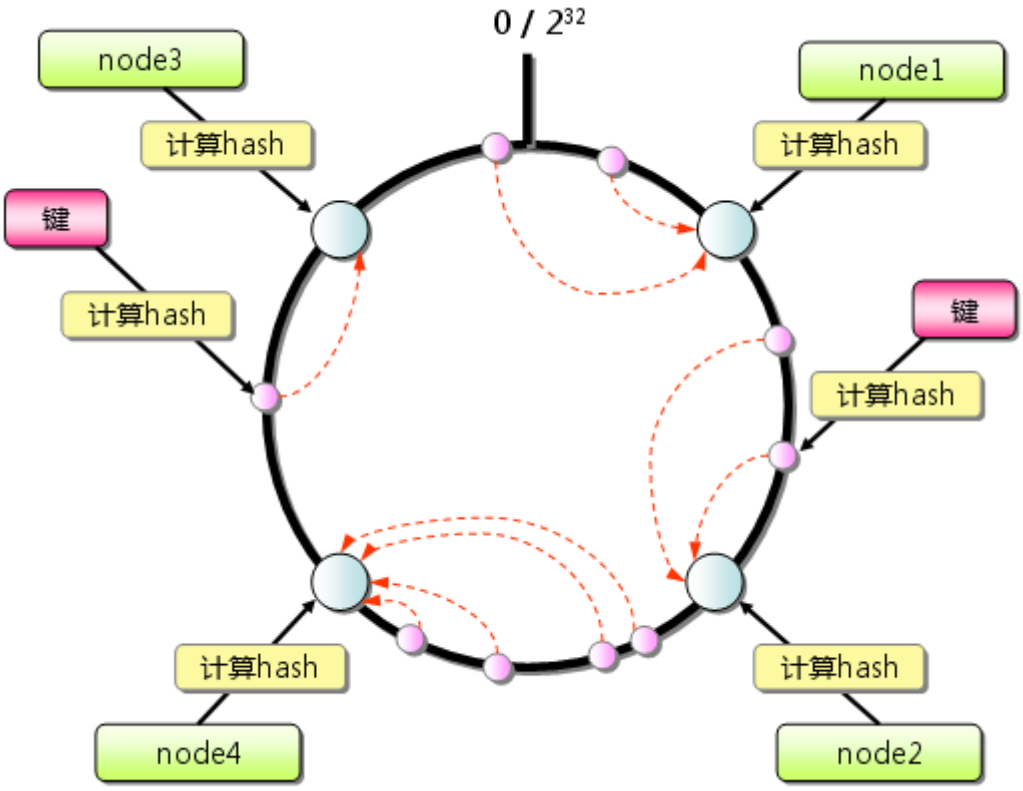


图 4 Consistent Hashing：基本原理

从上图的状态中添加一台 memcached 服务器。余数分布式算法由于保存键的服务器会发生巨大变化 而影响缓存的命中率，但 Consistent Hashing 中，只有在 continuum 上增加服务器的地点逆时针方向的 第一台服务器上的键会受到影响。

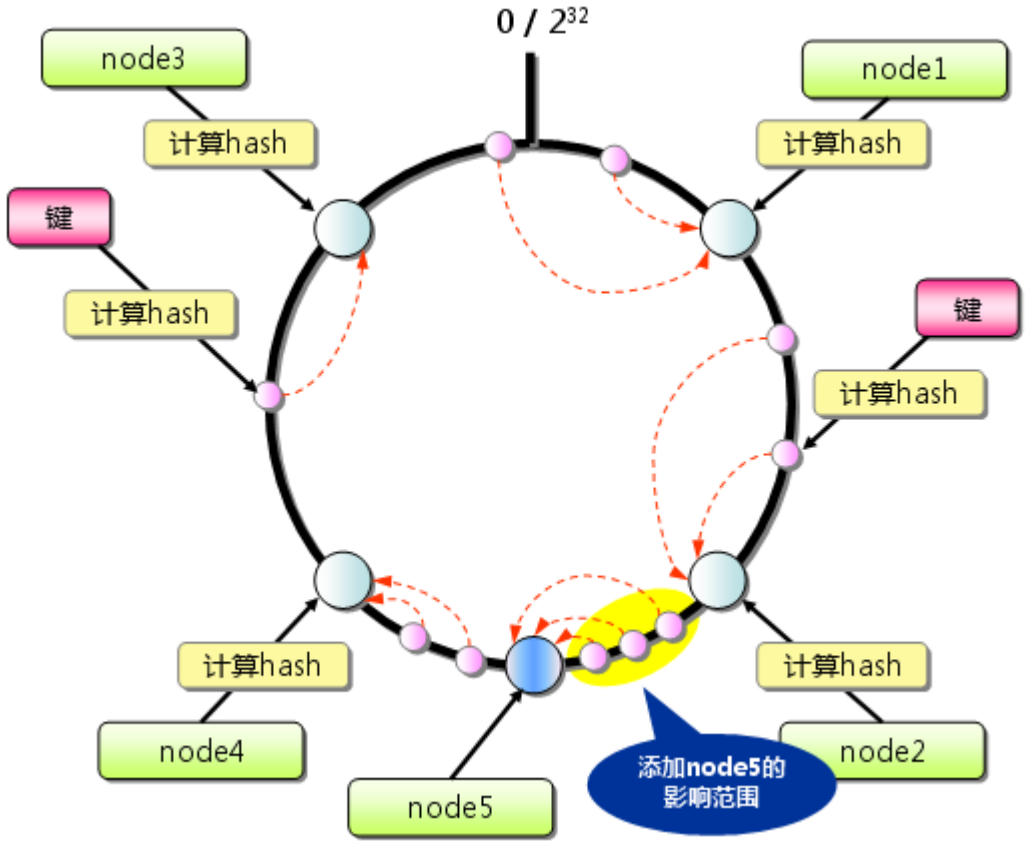


图 5 Consistent Hashing：添加服务器

因此，Consistent Hashing 最大限度地抑制了键的重新分布。 而且，有的 Consistent Hashing 的实现方法还采用了虚拟节点的思想。 使用一般的 hash 函数的话，服务器的映射地点的分布非常不均匀。 因此，使用虚拟节点的思想，为每个物理节点（服务器） 在 continuum 上分配 100～200 个点。这样就能抑制分布不均匀， 最大限度地减小服务器增减时的缓存重新分布。

通过下文中介绍的使用 Consistent Hashing 算法的 memcached 客户端函数库进行测试的结果是， 由服务器台数（n）和增加的服务器台数（m）计算增加服务器后的命中率计算公式如下：

(1 - n/(n+m)) * 100

支持 Consistent Hashing 的函数库

本连载中多次介绍的 Cache::Memcached 虽然不支持 Consistent Hashing， 但已有几个客户端函数库支持了这种新的分布式算法。 第一个支持 Consistent Hashing 和虚拟节点的 memcached 客户端函数库是 名为 libketama 的 PHP 库，由 last.fm 开发。

- libketama - a consistent hashing algo for memcache clients - RJ ブログ - Users at Last.fm

至于Perl客户端，连载的第 1 次 中介绍过的Cache::Memcached::Fast和Cache::Memcached::libmemcached支持 Consistent Hashing。

- Cache::Memcached::Fast - search.cpan.org
- Cache::Memcached::libmemcached - search.cpan.org

两者的接口都与 Cache::Memcached 几乎相同，如果正在使用 Cache::Memcached，那么就可以方便地替换过来。Cache::Memcached::Fast 重新实现了 libketama，使用 Consistent Hashing 创建对象时可以指定 ketama_points 选项。

```
my $memcached = Cache::Memcached::Fast->new({
    servers => ["192.168.0.1:11211","192.168.0.2:11211"],
    ketama_points => 150
});
```

另外，Cache::Memcached::libmemcached 是一个使用了 Brain Aker 开发的 C 函数库 libmemcached 的 Perl 模块。 libmemcached 本身支持几种分布式算法，也支持 Consistent Hashing， 其 Perl 绑定也支持 Consistent Hashing。

- Tangent Software: libmemcached

总结

本次介绍了 memcached 的分布式算法，主要有 memcached 的分布式是由客户端函数库实现， 以及高效率地分散数据的 Consistent Hashing 算法。下次将介绍 mixi 在 memcached 应用方面的一些经验， 和相关的兼容应用程序。