Memcached全面剖析

Edward Lee, 2014-09-19

本文主要参考了charlee翻译的《[Memcached全面剖析](http://blog.charlee.li/memcached-pdf/)》

1. Memcached是什么？

许多Web应用都将数据保存到RDBMS中，应用程序从中**读取数据**并在浏览器中显示。但随着数据量的增大和访问的集中，就会出现(**问题出现：**)RDBMS的负载加重、数据库响应恶化、网站显示延迟等重大影响。

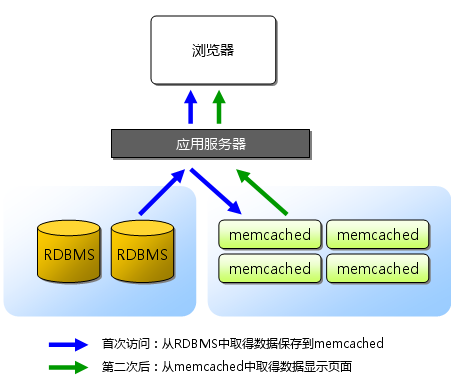
What is [Memcached](http://memcached.org/)?

Free & open source, high-performance, distributed **memory** object **caching system**, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.

Memcached is an in-memory **key-value** store for **small chunks of arbitrary data (strings, objects)** from results of **database calls, API calls, or page rendering**.

Memcached是一个高性能的分布式**内存对象缓存系统**。**目标**是，通过缓存**数据库查询**、**API调用**或**网页渲染结果**，减少**数据库**(**select**)访问次数，以加速**动态Web应用**的响应速度，提高**可扩展性**。（通过减轻数据库负载来使动态Web应用提速）

这时就该Memcached大显身手了。

图1.1 一般情况下Memcached的用途

适用场景

Memcached适用于**多查询(get)少更新**的场景。对于更新数据操作，数据每次还是会落地到DB；而对于**查询**操作，若缓存命中则直接返回，无需再请求DB。

1. 特征

* **Key-Value**
* Simple Protocol
* set/cas/get/gets
* incr/decr
* stats
* Lazy Expiration
* LRU
* Access
* telnet host port
* printf 'stats\r\n' | nc host port
* [Java client](http://blog.csdn.net/qqiabc521/article/details/6438429)

简单协议

目前，Memcached使用简单的[基于文本行的协议](https://github.com/memcached/memcached/blob/master/doc/protocol.txt)，还支持[二进制协议](https://github.com/memcached/memcached/blob/master/doc/protocol-binary.xml)。

常用操作命令

**保存数据**

* set：store this data（无论存不存在键相同的数据，都保存）
* **cas** (check and set)：store this data but only if no one else has updated since I last fetched it（仅当从我上次取来之后没有人更新它时，才保存）

**优势**：防止高并发下的数据不一致问题。

如果操作失败，可以增加**重试机制**(2次get-cas)。

**获取数据**

* get
* **gets**：需要一次取得多条数据时使用（批量操作）。它可以非同步地同时取得多个键值，(**优势**：)其速度要比循环调用get快数十倍。

**增一和减一操作**

可以将Memcached上特定的键值作为**计数器**使用。但在实际项目中很少使用这个特性，因为它没落地数据，一旦服务器崩溃就全完了。对于**单个维度**的计数，可以使用redis的[INCR](http://www.redis.cn/commands/incr.html)和[DECR](http://www.redis.cn/commands/decr.html)命令或MySQL的handlersocket；对于**用户的多维度**计数，可以使用redis的[hashes](http://www.redis.cn/topics/data-types.html#hashes)数据结构。

* incr
* decr

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。

**统计监控**

* stats：用于查询服务器维护的统计信息和其它内部数据

1. 内存存储

Slab Allocator机制：整理内存以便重复使用

目前，Memcached默认情况下使用Slab Allocator机制来分配、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，(**缺点**：)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，在最坏的情况下，会导致操作系统比Memcached进程本身还慢。**Slab Allocator**就是为了解决该问题而诞生的。

Slab源于Jeff Bonwick为SunOS操作系统首次引入的一种内存处理机制。Slab的**设计理念**是基于**对象缓冲**的，**基本想法**是**避免重复大量**的初始化和清理操作。(**用途**：)Slab主要用于频繁分配释放的内存对象。如果是采用系统自带的**malloc/free**的话，反复地操作会造成大量内存碎片，操作系统将会花费大量的时间去查找连续的内存块来满足malloc的请求。——摘自《[Memcached源码剖析笔记](http://www.cppblog.com/xguru/archive/2010/01/23/106265.html)》

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**。

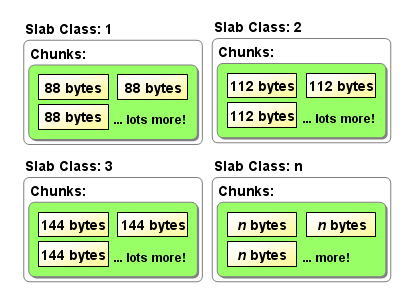
Slab Allocator的**原理**相当简单。将分配的内存分割成各种尺寸的**块(chunk)**，并把尺寸相同的块分成**组(chunk的集合，Slab Class)**（图2.1）。

图2.1 Slab Allocator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 Allocator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

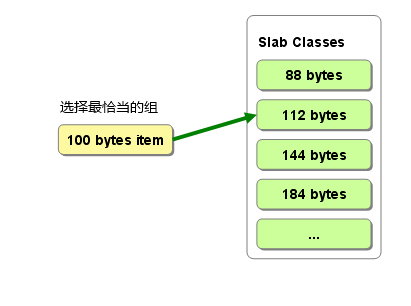
**Slab Class**

特定大小的**Chunk组**

在Slab中缓存记录的原理

下面说明Memcached是如何针对客户端发送的数据来选择Slab并缓存到Chunk中。

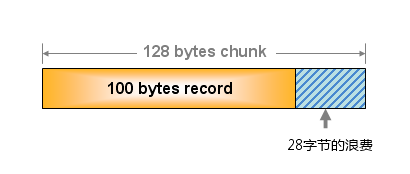
（第1步）Memcached根据收到的**数据的大小**，选择最适合该数据大小的Slab（图2.2）。（第2步）Memcached中保存着**Slab内空闲Chunk的列表**，根据该列表选择Chunk，然后将数据缓存于其中。

图2.2 选择存储记录的Chunk组的方法

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2.3）。

图2.3 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常用大小，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Chunk组的列表，就可以减少浪费。

使用Growth Factor进行调优

Memcached在启动时，指定Growth Factor因子(-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。

在直接使用Memcached默认值进行部署之前，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命令来查看slabs的利用率等各种各样的信息。

stats：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。

查看slabs的使用状况

使用Memcached的创造者Brad写的名为[memcached-tool](http://code.sixapart.com/svn/memcached/trunk/server/scripts/memcached-tool)的Perl脚本，可以方便地获得slab的使用情况 （它将返回值整理成容易阅读的格式）。

从这个脚本获得的信息对于调优非常方便，**强烈推荐使用**！

总结

本章简单说明了Memcached的缓存机制和调优方法。希望读者能够理解**Memcached的内存管理原理**及**其优缺点**。

1. 删除机制和发展方向

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。本章介绍Memcached的**数据删除机制**，以及Memcached的最新发展方向——二进制协议和外部引擎支持。

Memcached在数据删除方面有效利用资源

**懒惰删除机制**：数据不会真正从Memcached中消失

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)，其存储空间即可重复使用。

（源码剖析）在Memcached中删除一个item对象的时候，并不是从内存中释放，而是简单的进行标记处理，再将其指针放入slot回收插槽，下次分配的时候直接使用。

**Lazy Expiration**：**懒惰检测机制**

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 Expiration。（**优势**：）因此，Memcached不会在过期监视上耗费CPU时间。

（源码剖析）Memcached不花过多的时间检测各个item对象是否过期，当get获取数据时，才检查item对象是否应该删除。你不访问，我就不处理。

**LRU**：从缓存中有效删除数据的原理

Memcached会**优先使用**已过期的记录的空间。但即使如此，也会发生追加新纪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，（**实现**：）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。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**。

**外部引擎支持**

外部引擎支持的必要性

世界上有许多Memcached的派生系统，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。

**总结**

本章介绍了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起支持等Memcached的最新发展方向。这些功能要到1.3版才会支持，尽请期待！

1. 分布式算法

**Memcached的分布式**

Memcached虽然成为“分布式”缓存系统，但**服务器端**并没有“分布式”功能，各个Memcached不会相互通信以共享信息。**服务器端**仅包括前面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。这种分布式是Memcached的最大特点。（Memcached不互相通信的分布式）

**Memcached的分布式是什么意思？**

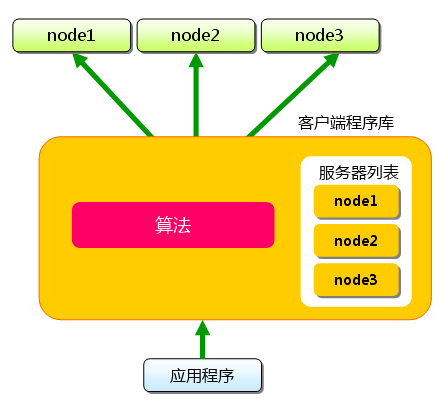
这里多次使用了“分布式”这个词，但并未做详细解释。现在开始简单地介绍一下其原理，各个客户端的实现基本相同。

图4.1 分布式简介：准备

**保存数据**

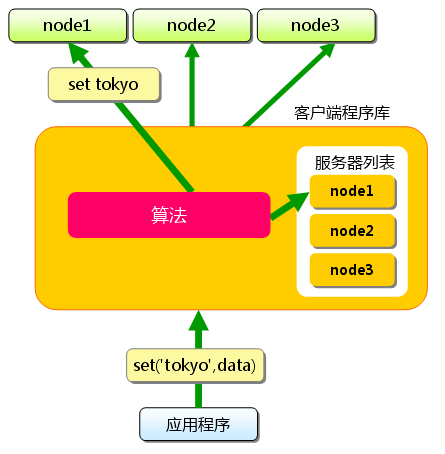
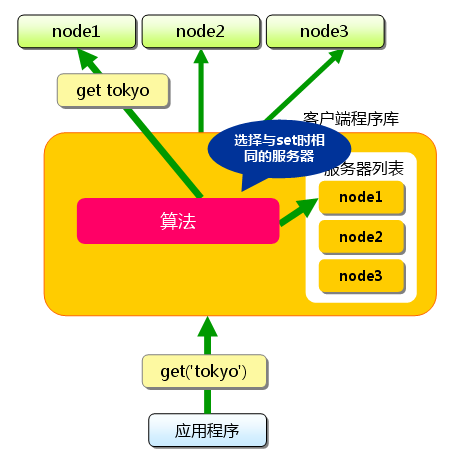
首先，向memcached中**添加**“tokyo”。（第1步）将“tokyo”传给客户端程序库后，客户端实现的算法就会根据“**键**”来决定保存数据的Memcached服务器。（第2步）服务器选定后，即(set/cas)命令它保存“tokyo”及其值。

图4.2 分布式简介：**添加**时

**获取数据**

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。（第1步）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，（第2步）然后发送获取命令(**get/gets**)。只要数据没有因为某些原因被删除，就能获得保存的值。

图4.3 分布式简介：**获取**时

**分布式算法**

**取模算法**

简单来说，就是“根据服务器台数的余数进行散列”。求得键的整数哈希值（CRC32），再除以服务器台数，根据其余数来选择服务器。

**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。

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**：**一致性哈希算法**

Consistent Hashing的简单说明

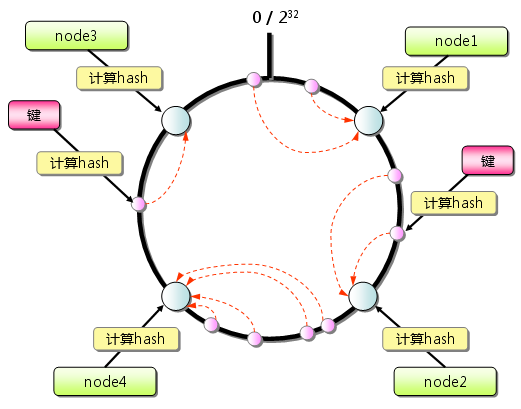
Consistent Hashing如下图所示：首先，求出Memcached服务器（节点）的哈希值，并将其配置到0～2SUP(32)的圆上。然后，用同样的方法求出存储数据的**键**的哈希值，并映射到圆上。然后，从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过2SUP(32)仍然找不到服务器，就会保存到第一台memcached服务器上。

图4.4 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，只有在圆上**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。

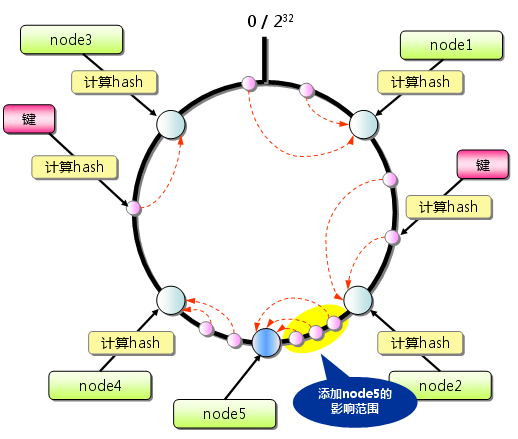


图4.5 Consistent Hashing：**添加服务器**

（**优势**：）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）在圆上分配100～200个点。（**优势**：）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。

**总结**

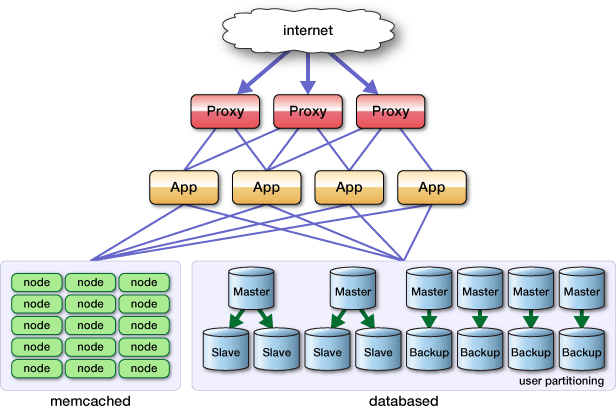
本章介绍了Memcached的分布式算法，主要有Memcached的分布式是由客户端函数库实现，以及高效率地分散数据的Consisting Hashing算法。

1. Memcached的应用和兼容程序

[**mixi案例研究**](http://blog.charlee.li/memcached-005/)

mixi在提供服务的初期阶段就使用了Memcached。

随着网站访问量的急剧增加，单纯为数据库添加slave已无法满足需要。因此引入了Memcached。此外，我们也从增加可扩展性的方面进行了验证，证明了Memcached的速度和稳定性都能满足需要。现在，Memcached已成为mixi服务中非常重要的组成部分。

图1 mixi现在的系统组件