**5、掌握PHP操作memcache的方式和必备条件(环境)，并能理解缓存原理和在项目中使用**

1. **Memcache简介**   
   MemCache是一个自由、源码开放、高性能、分布式的分布式内存对象缓存系统，用于动态Web应用以减轻数据库的负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高了网站访问的速度。MemCaChe是一个存储键值对的HashMap，在内存中对任意的数据（比如字符串、对象等）所使用的key-value存储，数据可以来自数据库调用、API调用，或者页面渲染的结果。MemCache设计理念就是小而强大，它简单的设计促进了快速部署、易于开发并解决面对大规模的数据缓存的许多难题，而所开放的API使得MemCache能用于Java、C/C++/C#、Perl、Python、PHP、Ruby等大部分流行的程序语言。   
   另外，说一下MemCache和MemCached的区别：

1、MemCache是项目的名称

2、MemCached是MemCache服务器端可以执行文件的名称

MemCache的官方网站为[http://memcached.org/](http://memcached.org/" \t "http://www.mamicode.com/_blank)

**再总结MemCache的特性和限制**

1、MemCache中可以保存的item数据量是没有限制的，只要内存足够

2、MemCache单进程在32位机中最大使用内存为2G，这个之前的文章提了多次了，64位机则没有限制

3、Key最大为250个字节，超过该长度无法存储

4、单个item最大数据是1MB，超过1MB的数据不予存储

5、MemCache服务端是不安全的，比如已知某个MemCache节点，可以直接telnet过去，并通过flush\_all让已经存在的键值对立即失效

6、不能够遍历MemCache中所有的item，因为这个操作的速度相对缓慢且会阻塞其他的操作

7、MemCache的高性能源自于两阶段哈希结构：第一阶段在客户端，通过Hash算法根据Key值算出一个节点；第二阶段在服务端，通过一个内部的Hash算法，查找真正的item并返回给客户端。从实现的角度看，MemCache是一个非阻塞的、基于事件的服务器程序

8、MemCache设置添加某一个Key值的时候，传入expiry为0表示这个Key值永久有效，这个Key值也会在30天之后失效，见memcache.c的源代码：

IMG_256

#define REALTIME\_MAXDELTA 60\*60\*24\*30

static rel\_time\_t realtime(const time\_t exptime) {

if (exptime == 0) return 0;

if (exptime > REALTIME\_MAXDELTA) {

if (exptime <= process\_started)

return (rel\_time\_t)1;

return (rel\_time\_t)(exptime - process\_started);

} else {

return (rel\_time\_t)(exptime + current\_time);

}

}

IMG_257

这个失效的时间是memcache源码里面写的，开发者没有办法改变MemCache的Key值失效时间为30天这个限制

|  |  |
| --- | --- |
| **命    令** | **作    用** |
| get | 返回Key对应的Value值 |
| add | 无条件地设置一个Key值，没有就增加，有就覆盖 |
| set | 按照相应的Key值添加数据，如果Key已经存在则会操作失败 |
| replace | 按照相应的Key值替换数据，如果Key值不存在则会操作失败 |
| stats | 返回MemCache通用统计信息（下面有详细解读） |
| stats items | 返回各个slab中item的数目和最老的item的年龄（最后一次访问距离现在的秒数） |
| stats slabs | 返回MemCache运行期间创建的每个slab的信息（下面有详细解读） |
| version | 返回当前MemCache版本号 |
| flush\_all | 清空所有键值，但不会删除items，所以此时MemCache依旧占用内存 |
| quit | 关闭连接 |

**MemCache实现原理**

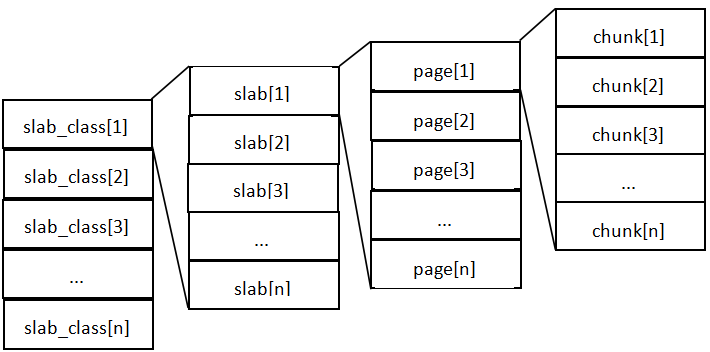
首先要说明一点，MemCache的数据存放在**内存**中，存放在内存中个人认为意味着几点：

1、访问数据的速度比传统的关系型数据库要快，因为Oracle、MySQL这些传统的关系型数据库为了保持数据的持久性，数据存放在硬盘中，IO操作速度慢

2、MemCache的数据存放在内存中同时意味着只要MemCache重启了，数据就会消失

3、既然MemCache的数据存放在内存中，那么势必受到机器位数的限制，这个之前的文章写过很多次了，32位机器最多只能使用2GB的内存空间，64位机器没有上限

然后我们来看一下MemCache的原理，MemCache最重要的莫不是内存分配的内容了，MemCache采用的内存分配方式是固定空间分配，还是自己画一张图说明：



这张图片里面涉及了slab\_class、slab、page、chunk四个概念，它们之间的关系是：

1、MemCache将内存空间分为一组slab

2、每个slab下又有若干个page，每个page默认是1M，如果一个slab占用100M内存的话，那么这个slab下应该有100个page

3、每个page里面包含一组chunk，chunk是真正存放数据的地方，同一个slab里面的chunk的大小是固定的

4、有相同大小chunk的slab被组织在一起，称为slab\_class

MemCache内存分配的方式称为allocator，slab的数量是有限的，几个、十几个或者几十个，这个和启动参数的配置相关。

MemCache中的value过来存放的地方是由value的大小决定的，value总是会被存放到与chunk大小最接近的一个slab中，比如slab[1]的chunk大小为80字节、slab[2]的chunk大小为100字节、slab[3]的chunk大小为128字节（**相邻slab内的chunk基本以1.25为比例进行增长，MemCache启动时可以用-f指定这个比例**），那么过来一个88字节的value，这个value将被放到2号slab中。放slab的时候，首先slab要申请内存，申请内存是以page为单位的，所以在放入第一个数据的时候，无论大小为多少，都会有1M大小的page被分配给该slab。申请到page后，slab会将这个page的内存按chunk的大小进行切分，这样就变成了一个chunk数组，最后从这个chunk数组中选择一个用于存储数据。

如果这个slab中没有chunk可以分配了怎么办，如果MemCache启动没有追加-M（禁止LRU，这种情况下内存不够会报Out Of Memory错误），那么MemCache会把这个slab中最近最少使用的chunk中的数据清理掉，然后放上最新的数据。针对MemCache的内存分配及回收算法，总结三点：

1、MemCache的内存分配chunk里面会有内存浪费，88字节的value分配在128字节（紧接着大的用）的chunk中，就损失了30字节，但是这也避免了管理内存碎片的问题

**2、MemCache的LRU算法不是针对全局的，是针对slab的**

3、应该可以理解为什么MemCache存放的value大小是限制的，因为一个新数据过来，slab会先以page为单位申请一块内存，申请的内存最多就只有1M，所以value大小自然不能大于1M了

**八、什么时候使用Memcache和Memcache的使用环境**   
使用Memcache的网站一般流量都是比较大的，为了缓解数据库的压力，让Memcache作为一个缓存区域，把部分信息保存在内存中，在前端能够迅速的进行存取。那么一般的焦点就是集中在如何分担数据库压力和进行分布式，毕竟单台Memcache的内存容量的有限的。我这里简单提出我的个人看法，未经实践，权当参考。   
  
**分布式应用**   
Memcache本来支持分布式，我们客户端稍加改造，更好的支持。我们的key可以适当进行有规律的封装，比如以user为主的网站来说，每个用户都有User ID，那么可以按照固定的ID来进行提取和存取，比如1开头的用户保存在第一台Memcache服务器上，以2开头的用户的数据保存在第二胎Mecache服务器上，存取数据都先按照User ID来进行相应的转换和存取。   
但是这个有缺点，就是需要对User ID进行判断，如果业务不一致，或者其他类型的应用，可能不是那么合适，那么可以根据自己的实际业务来进行考虑，或者去想更合适的方法。   
  
**减少数据库压力**   
这个算是比较重要的，所有的数据基本上都是保存在数据库当中的，每次频繁的存取数据库，导致数据库性能极具下降，无法同时服务更多的用户，比如MySQL，特别频繁的锁表，那么让Memcache来分担数据库的压力吧。我们需要一种改动比较小，并且能够不会大规模改变前端的方式来进行改变目前的架构。   
我考虑的一种简单方法：   
后端的数据库操作模块，把所有的Select操作提取出来（update/delete/insert不管），然后把对应的SQL进行相应的hash算法计算得出一个hash数据key（比如MD5或者SHA），然后把这个key去Memcache中查找数据，如果这个数据不存在，说明还没写入到缓存中，那么从数据库把数据提取出来，一个是数组类格式，然后把数据在set到Memcache中，key就是这个SQL的hash值，然后相应的设置一个失效时间，比如一个小时，那么一个小时中的数据都是从缓存中提取的，有效减少数据库的压力。缺点是数据不实时，当数据做了修改以后，无法实时到前端显示，并且还有可能对内存占用比较大，毕竟每次select出来的数据数量可能比较巨大，这个是需要考虑的因素。   
  
**九、Memcache的安全**   
  
我们上面的Memcache服务器端都是直接通过客户端连接后直接操作，没有任何的验证过程，这样如果服务器是直接暴露在互联网上的话是比较危险，轻则数据泄露被其他无关人员查看，重则服务器被入侵，因为Mecache是以root权限运行的，况且里面可能存在一些我们未知的bug或者是缓冲区溢出的情况，这些都是我们未知的，所以危险性是可以预见的。为了安全起见，我做两点建议，能够稍微的防止黑客的入侵或者数据的泄露。   
  
**内网访问**   
  
最好把两台服务器之间的访问是内网形态的，一般是Web服务器跟Memcache服务器之间。普遍的服务器都是有两块网卡，一块指向互联网，一块指向内网，那么就让Web服务器通过内网的网卡来访问Memcache服务器，我们Memcache的服务器上启动的时候就监听内网的IP地址和端口，内网间的访问能够有效阻止其他非法的访问。

复制代码代码如下:

# memcached -d -m 1024 -u root -l 192.168.0.200 -p 11211 -c 1024 -P /tmp/memcached.pid

Memcache服务器端设置监听通过内网的192.168.0.200的ip的11211端口，占用1024MB内存，并且允许最大1024个并发连接。   
  
**设置防火墙** 防火墙是简单有效的方式，如果却是两台服务器都是挂在网的，并且需要通过外网IP来访问Memcache的话，那么可以考虑使用防火墙或者代理程序来过滤非法访问。一般我们在Linux下可以使用iptables或者FreeBSD下的ipfw来指定一些规则防止一些非法的访问，比如我们可以设置只允许我们的Web服务器来访问我们Memcache服务器，同时阻止其他的访问。

复制代码代码如下:

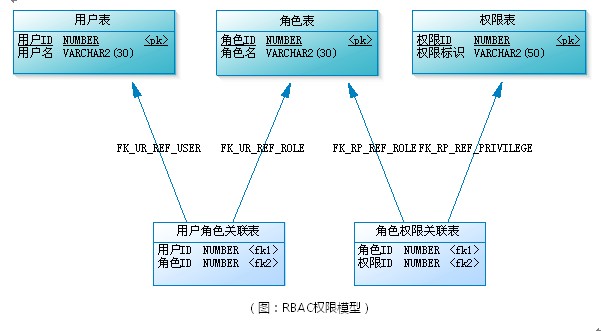
# iptables -F   
# iptables -P INPUT DROP   
# iptables -A INPUT -p tcp -s 192.168.0.2 –dport 11211 -j ACCEPT   
# iptables -A INPUT -p udp -s 192.168.0.2 –dport 11211 -j ACCEPT

**6、掌握RBAC权限模型原理，并能实现权限的控制.**

        举个简单例子：一个老师在学校教室她就拥有教书育人的权利义务，一个丈夫在家就有呵护妻子支撑家庭的权利义务，而一个父亲在孩子面前就有保护孩子，教育孩子的权利义务……而作为一个男生，我们很可能在不同的场所，成为这些角色，从而拥有了这些权利义务。而抽象出来就是用户，角色和权利三个方面。

         所以经过前人对权限方面的总结抽象，总结出来RBAC（Role-Based Access Control ）基于角色的访问控制。在以前做项目的时候，就听组长说权限管理模型的运用，但是自己但是只是接触了一下，直到后来在项目中真正的使用，才对此有深刻的理解。

           RBAC认为授权实际就是who,what,how三者之间的关系，即who对what进行how的操作。Who,权限的拥用者或主体（如Principal、User、Group、Role、Actor等等）；what,权限针对的对象或资源（Resource、Class） ；How,具体的权限（Privilege,正向授权与负向授权）。简单一点说吧就是，我们通过给角色授权，然后将附有权利的角色施加到某个用户身上，这样用户就可以实施相应的权利了。通过中间角色的身份，是权限管理更加灵活：角色的权利可以灵活改变，用户的角色的身份可以随着场所的不同而发生改变等。这样这套RBAC就几乎可以运用到所有的权限管理的模块上了。



角色是什么？可以理解为一定数量的权限的集合，权限的载体。例如：一个论坛系统，“超级管理员”、“版主”都是角色。版主可管理版内的帖子、可管理版内的用户等，这些是权限。要给某个用户授予这些权限，不需要直接将权限授予用户，可将“版主”这个角色赋予该用户。