**1、redis和memcache问题**

**1.1 基本概念**

Memcache是一个高性能，分布式内存对象缓存系统，通过在内存里维护一个统一的巨大的hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。

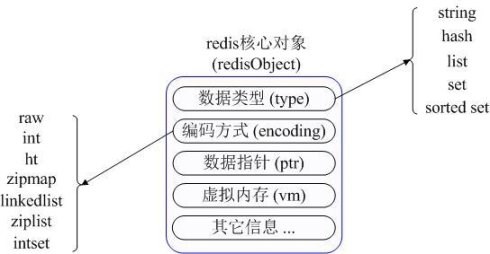
redis是一个开源的、基于内存的，多数据结构的key-value存储系统；其可以用作数据库、缓存和消息中间件。同memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

**1.2 redis和memcache支持的数据类型**

Memcached仅支持简单的key-value结构的数据记录；

Redis支持的数据结构中最为常用的主要有五种：String(字符串)、Hash(散列)、List(列表)、Set(集合)和Sorted Set(有序集合)；

Redis内部使用一个redisObject对象来表示所有的key和value。redisObject最主要的信息如图所示：

[](http://www.biaodianfu.com/wp-content/uploads/2014/01/redisObject.jpg)

type代表一个value对象具体是何种数据类型，encoding是不同数据类型在redis内部的存储方式。

比如：type=string代表value存储的是一个普通字符串，那么对应的encoding可以是raw或者是int，如果是int则代表实际redis内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如:”123″ “456”这样的字符串。

只有打开了Redis的虚拟内存功能，vm字段字段才会真正的分配内存，该功能默认是关闭状态的。

**1.3 Memcache的特性**

（1）使用物理内存作为缓存区，可独立运行在服务器上。每个进程最大2G，如果想缓存更多的数据，可以开辟更多的memcache进程（不同端口）或者使用分布式memcache进行缓存，将数据缓存到不同的物理机或者虚拟机上。

（2）使用key-value的方式来存储数据，这是一种单索引的结构化数据组织形式，可使数据项查询时间复杂度为O(1)。

（3）协议简单：基于文本行的协议，直接通过telnet在memcached服务器上可进行存取数据操作，简单，方便多种缓存参考此协议；

（4）基于libevent高性能通信：Libevent是一套利用C开发的程序库，它将BSD系统的kqueue,[Linux](http://lib.csdn.net/base/linux)系统的epoll等事件处理功能封装成一个接口，与传统的select相比，提高了性能。

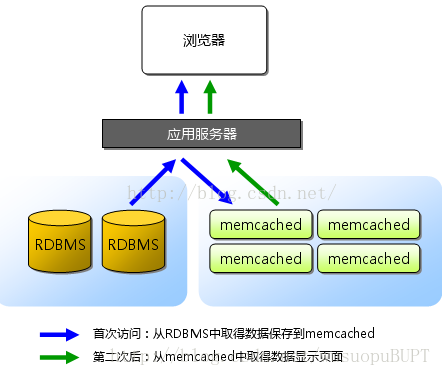
（5）内置的内存管理方式：所有数据都保存在内存中，存取数据比硬盘快，当内存满后，通过LRU[算法](http://lib.csdn.net/base/datastructure)自动删除不使用的缓存，但没有考虑数据的容灾问题，重启服务，所有数据会丢失。

（6）分布式：各个memcached服务器之间互不通信，各自独立存取数据，不共享任何信息。服务器并不具有分布式功能，分布式部署取决于memcache客户端。

（7）缓存策略：Memcached的缓存策略是LRU（最近最少使用）到期失效策略。

在memcached内存储数据项时，可以指定它在缓存的失效时间，默认为永久。当memcached服务器用完分配的内时，失效的数据被首先替换，然后也是最近未使用的数据。在LRU中，memcached使用的是一种Lazy Expiration策略，自己不会监控存入的key/vlue对是否过期，而是在获取key值时查看记录的时间戳，检查key/value对空间是否过期，这样可减轻服务器的负载。

Memcache工作原理

图1

MemCache的工作流程如下：

（1） 先检查客户端的请求数据是否在memcached中，如有，直接把请求数据返回，不再对数据库进行任何操作；

（2） 如果请求的数据不在memcached中，就去查数据库，把从数据库中获取的数据返回给客户端，同时把数据缓存一份到memcached中（memcached客户端不负责，需要程序实现）；

（3） 每次更新数据库的同时更新memcached中的数据，保证一致性；

（4） 当分配给memcached内存空间用完之后，会使用LRU（Least Recently Used，最近最少使用）策略加上到期失效策略，失效数据首先被替换，然后再替换掉最近未使用的数据。

**1.4 redis的数据类型详解**

（1）String

常用命令：set/get/decr/incr/mget等；

应用场景：String是最常用的一种数据类型，普通的key/value存储都可以归为此类；即可以完全实现目前 Memcached 的功能，并且效率更高

实现方式：String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr、decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

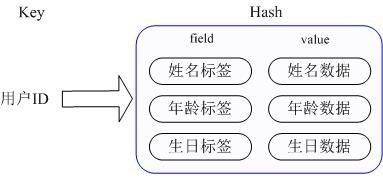
2）Hash

常用命令：hget/hset/hgetall等

应用场景：我们要存储一个用户信息对象数据，其中包括用户ID、用户姓名、年龄和生日，通过用户ID我们希望获取该用户的姓名或者年龄或者生日；

实现方式：Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口。如图所示，Key是用户ID, value是一个Map。这个Map的key是成员的属性名，value是属性值。这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过 key(用户ID) + field(属性标签) 就可以操作对应属性数据。

当前HashMap的实现有两种方式：当HashMap的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，这时对应的value的redisObject的encoding为zipmap，当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。

[](http://www.biaodianfu.com/wp-content/uploads/2014/01/hash.jpg)

3）List

常用命令：lpush/rpush/lpop/rpop/lrange等；

应用场景：Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现；

实现方式：Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

4）Set

常用命令：sadd/spop/smembers/sunion等；

应用场景：Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的；Set集合的概念就是一堆不重复值的组合。利用Redis提供的Set数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

实现方式：set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。

5）Sorted Set

常用命令：zadd/zrange/zrem/zcard等；

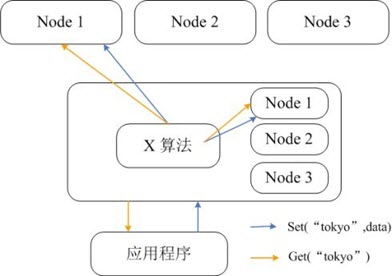
应用场景：Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

实现方式：Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

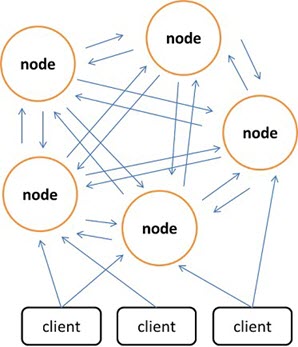
**1.5 集群管理的不同**

Memcached是全内存的数据缓冲系统，Redis虽然支持数据的持久化，但是全内存毕竟才是其高性能的本质。作为基于内存的存储系统来说，机器物理内存的大小就是系统能够容纳的最[大数据](http://lib.csdn.net/base/hadoop)量。如果需要处理的数据量超过了单台机器的物理内存大小，就需要构建分布式集群来扩展存储能力。

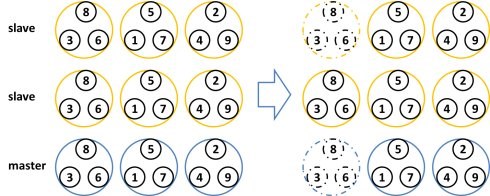
Memcached本身并不支持分布式，因此只能在客户端通过像一致性哈希这样的分布式[算法](http://lib.csdn.net/base/datastructure)来实现Memcached的分布式存储。下图给出了Memcached的分布式存储实现[架构](http://lib.csdn.net/base/architecture)。当客户端向Memcached集群发送数据之前，首先会通过内置的分布式算法计算出该条数据的目标节点，然后数据会直接发送到该节点上存储。但客户端查询数据时，同样要计算出查询数据所在的节点，然后直接向该节点发送查询请求以获取数据。

[](http://www.biaodianfu.com/wp-content/uploads/2014/01/Memcached-node.jpg)

相较于Memcached只能采用客户端实现分布式存储，Redis更偏向于在服务器端构建分布式存储。最新版本的Redis已经支持了分布式存储功能。Redis Cluster是一个实现了分布式且允许单点故障的Redis高级版本，它没有中心节点，具有线性可伸缩的功能。下图给出Redis Cluster的分布式存储架构，其中节点与节点之间通过二进制协议进行通信，节点与客户端之间通过ascii协议进行通信。在数据的放置策略上，Redis Cluster将整个key的数值域分成4096个哈希槽，每个节点上可以存储一个或多个哈希槽，也就是说当前Redis Cluster支持的最大节点数就是4096。Redis Cluster使用的分布式算法也很简单：crc16( key ) % HASH\_SLOTS\_NUMBER。

[](http://www.biaodianfu.com/wp-content/uploads/2014/01/Redis-Cluster.jpg)

为了保证单点故障下的数据可用性，Redis Cluster引入了Master节点和Slave节点。在Redis Cluster中，每个Master节点都会有对应的两个用于冗余的Slave节点。这样在整个集群中，任意两个节点的宕机都不会导致数据的不可用。当Master节点退出后，集群会自动选择一个Slave节点成为新的Master节点。

[](http://www.biaodianfu.com/wp-content/uploads/2014/01/Redis-Cluster-2.jpg)

**1.6 Memcache和Redis的比较总结**

（1）数据结构：Memcache只支持key value存储方式，Redis支持更多的数据类型，比如Key value，hash，list，set，zset；

（2）多线程：Memcache支持多线程，redis支持单线程；CPU利用方面Memcache优于redis；

（3）持久化：Memcache不支持持久化，Redis支持持久化；

（4）内存利用率：memcache高，redis低（采用压缩的情况下比memcache高）；

（5）过期策略：memcache过期后，不删除缓存，会导致下次取数据数据的问题，Redis有专门线程，清除缓存数据；

**2、UDP和TCP的对比**

 UDP协议在IP协议上增加了复用、分用和差错检测功能

TCP是在不可靠的IP层之上实现的可靠的数据传输协议，主要解决传输的可靠、有序、无丢失和不重复的问题。

|  |  |  |
| --- | --- | --- |
| 属性 | UDP | TCP |
| 建立连接 | 无需建立连接 | 面向连接 |
| 连接状态 | 无连接状态 | 连接状态（点对点） |
| 首部开销 | 8字节 | 20字节 |
| 交付 | 尽最大努力，不保证可靠交付 | 可靠交付 |
| 数据格式 | 面向报文 | 面向字节流 |
| 工作（通信）方式 | 1对1、1对多、多对1、多对多 | 双全工通信 |
| 拥塞控制 | 没有拥塞控制 | 具有拥塞控制 |
|  |  |  |
|  |  |  |

**3、PHP实现单例模式**

单例模式：

例：就是实例（instance），其实就是对象（object）

单例：就是一个对象

单例模式：就是设计出如下特性的一个类：这个类只能“创造”出它的一个对象（实例）；

<?php

class Single {

//第一步：私有化构造方法

private function \_\_construct() {

}

//第二步：定义一个静态属性，初始值为null

static private $instance = null;

//第三步：定义一个静态方法

static function getObject() {

//控制逻辑：只生产一个对象，然后返回给调用者

if(!isset(self::$instance)) { //还没有生产

$objSingle = new self(); //生产一个

self::$instance = $objSingle; //保存到static变量中

return self::$instance; //返回给调用者

}else {//已经生产

return self::$instance; //直接返回（之前生产过的）对象

}

}

//第四步：定义一个静态方法

Private function \_\_clone() {

//啥也不做

}

}

$objSingle1 = self::getObject();

$objSingle2 = self::getObject();

//此时$objSingle1和$objSingle1是一样的。

?>

（1）如果在外部修改静态变量的值呢？？？ 需要改进的地方---> static private $instance = null;

（2）如果使用clone（$obj3 = clone $obj1;是自动调用了该类中的魔术方法\_\_clone()）来复制对象时，那么又开始有多个了，那么该怎么解决呢？？

概括：禁止clone—>私有化 \_\_clone()这个魔术方法(那么在外界就不能调用了)

PHP基础补充：

PHP 构造方法 \_\_construct() 允许在实例化一个类之前先执行构造方法。

构造方法是类中的一个特殊方法。当使用 new 操作符创建一个类的实例时，构造方法将会自动调用，其名称必须是 \_\_construct() 。

在一个类中只能声明一个构造方法，而是只有在每次创建对象的时候都会去调用一次构造方法，不能主动的调用这个方法，所以通常用它执行一些有用的初始化任务。

该方法无返回值。

语法：

function \_\_construct(arg1,arg2,...) {

......

}

// 析构函数

function \_\_destruct() {

// ...

}

**对比Java：**

Java构造方法

类有一个特殊的成员方法叫作构造方法，它的作用是创建对象并初始化成员变量。在创建对象时，会自动调用类的构造方法。

构造方法定义规则：Java 中的构造方法必须与该类具有相同的名字，并且没有方法的返回类型（包括没有void）。

另外，构造方法一般都应用 public 类型来说明，这样才能在程序任意的位置创建类的实例－－对象。

示例：下面是一个 Rectangle 类的构造方法，它带有两个参数，分别表示矩形的长和宽：

public class Rectangle {

public Rectangle(int w, int h) {

width = w;

height = h;

}

public Rectangle() {}

}

每个类至少有一个构造方法。如果不写一个构造方法，Java 编程语言将提供一个默认的，该构造方法没有参数，而且方法体为空。

如果一个类中已经定义了构造方法则系统不再提供默认的构造方法。

Java析构方法

当垃圾回收器将要释放无用对象的内存时，先调用该对象的finalize（）方法。如果在程序终止前垃圾回收器始终没有执行垃圾回收操作，

那么垃圾回收器将始终不会调用无用对象的finalize（）方法。在Java的Object基类中提供了protected类型的finalize（）方法，

因此任何Java类都可以覆盖finalize（）方法，通常，在析构方法中进行释放对象占用的相关资源的操作。

Java 虚拟机的垃圾回收操作对程序完全是透明的，因此程序无法预料某个无用对象的finalize（）方法何时被调用。如果一个程序只占用少量内存，没有造成严重的内存需求，垃圾回收器可能没有释放那些无用对象占用的内存，因此这些对象的finalize（）方法还没有被调用，程序就终止了。

程序即使显式调用System.gc()或Runtime.gc()方法，也不能保证垃圾回收操作一定执行，也就不能保证对象的finalize（）方法一定被调用。

当垃圾回收器在执行finalize（）方法的时候，如果出现了异常，垃圾回收器不会报告异常，程序继续正常运行。

protected void finalize() {

System.out.println("in finalize");

}

在 Java 编程里面，一般不需要我们去写析构方法，这里只是了解一下就可以了。

**4、PHP实现工厂模式**

**工厂模式**

存在这样一个类（就是所谓的工厂）：这个类可以根据传递过来的类，生产出对应类的对象。

<?php

//设计一个工厂类：这个工厂类有一个静态方法，通过该方法可以获得指定类的对象

class factory {

static function getFactoryObject($className) {

$obj = new $className(); //这是可变类

return $obj;

}

}

//class A{}；

//class B{};

$obj1 = factory::getFactoryObject("A");

$obj2 = factory::getFactoryObject("B");

$obj3 = factory::getFactoryObject("A");

?>

注意结果：$obj1和$obj3的对象不是同一个。

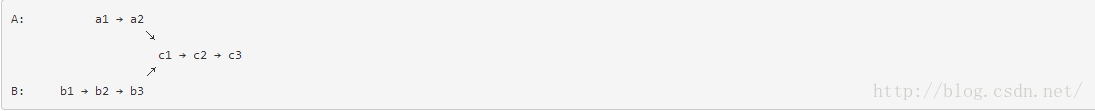


**5、求交叉链表的第一个交点（剑指offer）**

1. 问题描述

　　给定两个单链表，查找这两个单链表的交叉节点。例如：链表listA为：a1→a2→c1→c2→c3，链表listB为：b1→b2→b3→c1→c2→c3。那么这两个的第一个交叉节点为c1。

2. 方法与思路

　　首先，观察一下交叉节点的特点。如果两个链表有交叉节点的话，那么这个交叉节点之后的其他节点都是相同的，也就是说两个链表的结构应该是Y字型的。   
　　  
　　   
　　也就是说，c1之后的节点都是交叉节点。下面的问题就是如何确定c1这个节点，我们可以设两个指针分别遍历两个链表，然后对比节点的值，但是两个链表可能是不等长的，我们可以先让长度较大的链表指针先走|len(listA)−len(listB)|步，然后在同步进行。   
　　时间复杂度O(n)，空间复杂度O(1)。 

**6、树的递归和非递归遍历**

参考网址：<http://blog.csdn.net/rebirth_love/article/details/51312992>

1. /\*\* 递归实现前序遍历 \*/
2. **protected** **static** **void** preorder(BTNode p) {
3. **if** (p != **null**) {
4. visit(p);
5. preorder(p.getLeft());
6. preorder(p.getRight());
7. }
8. }
10. /\*\* 递归实现中序遍历 \*/
11. **protected** **static** **void** inorder(BTNode p) {
12. **if** (p != **null**) {
13. inorder(p.getLeft());
14. visit(p);
15. inorder(p.getRight());
16. }
17. }
19. /\*\* 递归实现后序遍历 \*/
20. **protected** **static** **void** postorder(BTNode p) {
21. **if** (p != **null**) {
22. postorder(p.getLeft());
23. postorder(p.getRight());
24. visit(p);
25. }
26. }
27. /\*\* 非递归实现前序遍历 \*/
28. **protected** **static** **void** iterativePreorder(BTNode p) {
29. Stack<BTNode> stack = **new** Stack<BTNode>();
30. **if** (p != **null**) {
31. stack.push(p);
32. **while** (!stack.empty()) {
33. p = stack.pop();
34. visit(p);
35. **if** (p.getRight() != **null**)
36. stack.push(p.getRight());
37. **if** (p.getLeft() != **null**)
38. stack.push(p.getLeft());
39. }
40. }
41. }
43. /\*\* 非递归实现后序遍历 \*/
44. **protected** **static** **void** iterativePostorder(BTNode p) {
45. BTNode q = p;
46. Stack<BTNode> stack = **new** Stack<BTNode>();
47. **while** (p != **null**) {
48. // 左子树入栈
49. **for** (; p.getLeft() != **null**; p = p.getLeft())
50. stack.push(p);
51. // 当前节点无右子或右子已经输出
52. **while** (p != **null** && (p.getRight() == **null** || p.getRight() == q)) {
53. visit(p);
54. q = p;// 记录上一个已输出节点
55. **if** (stack.empty())
56. **return**;
57. p = stack.pop();
58. }
59. // 处理右子
60. stack.push(p);
61. p = p.getRight();
62. }
63. }
65. /\*\* 非递归实现中序遍历 \*/
66. **protected** **static** **void** iterativeInorder(BTNode p) {
67. Stack<BTNode> stack = **new** Stack<BTNode>();
68. **while** (p != **null**) {
69. **while** (p != **null**) {
70. **if** (p.getRight() != **null**)
71. stack.push(p.getRight());// 当前节点右子入栈
72. stack.push(p);// 当前节点入栈
73. p = p.getLeft();
74. }
75. p = stack.pop();
76. **while** (!stack.empty() && p.getRight() == **null**) {//没有右子树，直接出栈
77. visit(p);
78. p = stack.pop();
79. }
80. visit(p);////有右子树，循环
81. **if** (!stack.empty())
82. p = stack.pop();
83. **else**
84. p = **null**;
85. }
86. }
88. //层次遍历
89. **public** **static** **void** levelOrder(BTNode p){
90. **if**(p==**null**)**return**;
91. LinkedList<BTNode> queue=**new** LinkedList<BTNode>();
92. queue.add(p);
93. **while**(!queue.isEmpty()){
94. BTNode temp=queue.remove();
95. visit(temp);
96. **if**(temp.getLeft()!=**null**){
97. queue.add(temp.getLeft());
98. }
99. **if**(temp.getRight() != **null**){
100. queue.add(temp.getRight());
101. }
102. }
103. }



第六题：给定a、b两个文件，各存放50亿个url，每个url各占用64字节，内存限制是4G，如何找出a、b文件共同的url？

思考：可以估计每个文件的大小为5G\*64=300G，远大于4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

思路：遍历文件a，对每个url求取hash(url)%1000，然后根据所得值将url分别存储到1000个小文件（设为a0,a1,...a999）当中。这样每个小文件的大小约为300M。

遍历文件b，采取和a相同的方法将url分别存储到1000个小文件(b0,b1....b999)中。

这样处理后，所有可能相同的url都在对应的小文件(a0 vs b0, a1 vs b1....a999 vs b999)当中，不对应的小文件（比如a0 vs b99）不可能有相同的url。

然后我们只要求出1000对小文件中相同的url即可。

比如对于a0 vs b0，我们可以遍历a0，将其中的url存储到hash\_map当中。

然后遍历b0，如果url在hash\_map中，则说明此url在a和b中同时存在，保存到文件中即可。