1. NoSQL

NoSQL（Not Only SQL），泛指非关系型的数据库（mysql、oracle、sqlserver都是关系型数据库）。

1.1 NoSQL特点

- 数据之间无关系，随意扩展
- 数据存储简单，可以存在内存中，读写速度快
- 不需要建表、字段。自定义格式

1.2 NoSQL数据库的分类

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值（key-value）	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB，memcache	**内容缓存，主要用于处理大量数据的高访问负载**，也用于一些日志系统等等。	Key 指向 Value 的键值对，通常用hash table来实现	查找速度快	数据无
列存储数据库	Cassandra, HBase, Riak	分布式的文件系统	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相
文档型数据库	CouchDB, MongoDb	Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容）	Key-Value对应的键值对，Value为结构化数据	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	查询性

图形数据库	Neo4J, InfoGrid, Infinite Graph	社交网络，推荐系统等。专注于构建关系图谱	图结构	利用图结构相关算法。比如最短路径寻址，N度关系查找等	很多时 而且这

1.2.1 共同特征

1. **不需要预定义模式**：不需要事先定义数据模式，预定义表结构。数据中的每条记录都可能有不同的属性和格式。当插入数据时，并不需要预先定义它们的模式。
2. **无共享架构**：相对于将所有数据存储的存储区域网络中的全共享架构。NoSQL往往将数据划分后存储在各个本地服务器上。因为从本地磁盘读取数据的性能往往好于通过网络传输读取数据的性能，从而提高了系统的性能。
3. **弹性可扩展**：可以在系统运行的时候，动态增加或者删除结点。不需要停机维护，数据可以自动迁移。
4. **分区**：相对于将数据存放于同一个节点，NoSQL数据库需要将数据进行分区，将记录分散在多个节点上面。并且通常分区的同时还要做复制。这样既提高了并行性能，又能保证没有单点失效的问题。
5. **异步复制**：和RAID存储系统不同的是，NoSQL中的复制，往往是**基于日志的异步复制**。这样，数据就可以尽快地写入一个节点，而不会被网络传输引起迟延。缺点是**并不总是能保证一致性**，这样的方式在出现故障的时候，可能会丢失少量的数据。
6. BASE：相对于事务严格的ACID(原子性，一致性，隔离性，持久性)特性，NoSQL数据库保证的是BASE（BA——基本可用，S——软状态，柔性事务，E——最终一致性）特性。【注】CAP原理（C——一致性，A——可用性，P——分区容错性，当前NoSQL大部分满足了AP原理）

1.2.2 适用场景

NoSQL数据库在以下的这几种情况下比较适用：1、数据模型比较简单；2、需要灵活性更强的IT系统；3、对数据库性能要求较高；4、不需要高度的数据一致性；5、对于给定key，比较容易映射复杂值的环境。

2. 常用NoSQL介绍

2.1 Memcached

2.1.1 Memcached基础

2.1.1.1 什么是Memcached?

MemCache是一个开源的**高性能的分布式的内存对象缓存系统**，用于各种动态应用以减轻数据库负担。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**，从而提高动态、数据库驱动应用速度。MemCache会在内存中开辟一块空间，建立一个统一的巨大的hash**表**，hash表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。

【注】MemCache 和 MemCached：MemCache是这个项目的名称，而MemCached是服务器端的主程序名称。

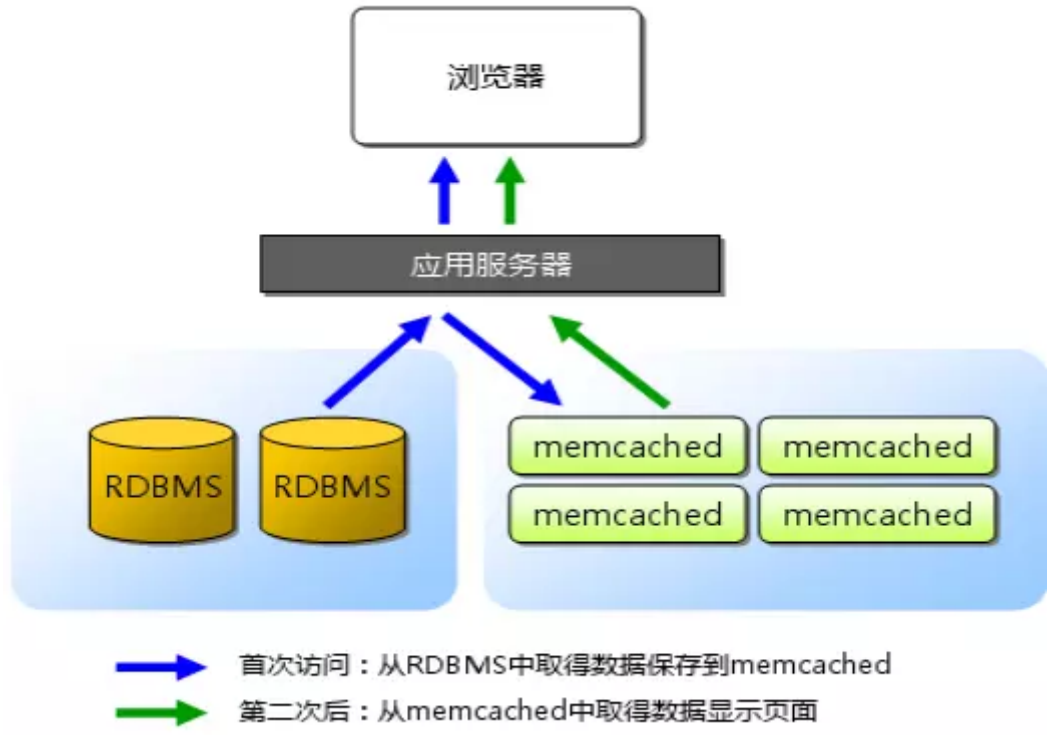
2.1.1.2 Memcached应用场景

通常在**访问量高**的Web网站和应用中使用MemCache，用来缓解数据库的压力，并且**提升网站和应用的响应速度。**

在应用程序中，我们通常在以下节点来使用 MemCached：

1. 访问频繁的数据库数据（身份token、首页动态）
2. 访问频繁的查询条件和结果
3. 作为Session的存储方式（提升Session存取性能）
4. 页面缓存
5. 更新频繁的非重要数据（访客量、点击次数）
6. 大量的hot数据

常用工作流程（如下图）：



1. 客户端请求数据
2. 检查MemCached中是否有对应数据
3. 有的话直接返回，结束
4. 没有的话，去数据库里请求数据
5. 将数据写入MemCached，供下次请求时使用
6. 返回数据，结束

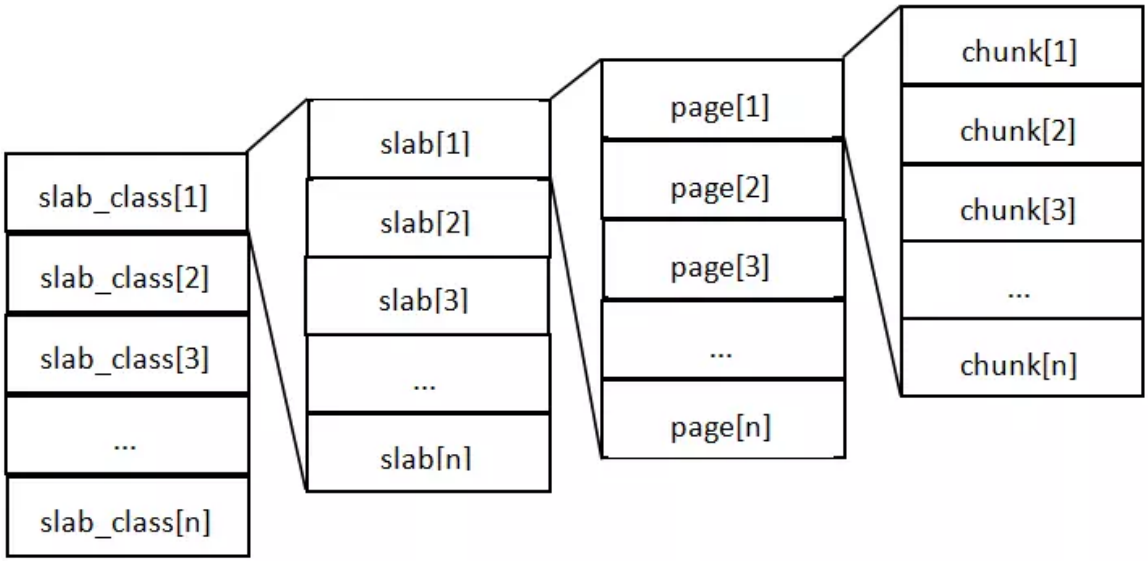
缓存到MemCached中的数据库数据，在更新数据库时要同时更新MemCached。

2.1.2 Memcached工作原理

MemCached采用了**C/S架构**，在Server端启动后，以守护程序的方式，监听客户端的请求。启动时可以指定监听的IP（服务器的内网ip/外网ip）、端口号（所以做分布式测试时，一台服务器上可以启动多个不同端口号的MemCached进程）、使用的内存大小等关键参数。一旦启动，服务就会一直处于可用状态。
为了提高性能，MemCached**缓存的数据全部存储在**MemCached**管理的内存中，所以重启服务器之后缓存数据会清空，不支持持久化。**

2.1.2.1 Memcached内存管理

1. **内存结构**



1. slab_class里，存放的是一组组chunk大小相同的slab
2. 每个slab里面包含若干个page，page的默认大小是1M，如果slab大小100M，就包含100个page
3. 每个page里面包含若干个chunk，chunk是数据的实际存放单位，每个slab里面的chunk大小相同

2. 内存分配方式

1. Memcached利用slab allocation机制来分配和管理内存，它按照预先规定的大小，将分配的内存分割成特定长度的内存块，再把尺寸相同的内存块分成组，数据在存放时，根据键值大小去匹配slab大小，找就近的slab存放，所以存在空间浪费现象。而传统的内存管理方式是，使用完通过malloc分配的内存后通过free来回收内存，这种方式容易产生内存碎片并降低操作系统对内存的管理效率。
2. 存放数据时，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。

（图一） （图二）

MemCache中的value存放位置是由value的大小决定，value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88字节、slab[2]的chunk大小为112字节、slab[3]的chunk大小为144字节（*默认相邻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*），那么一个100字节的value，将被放到2号slab中。

3. 内存回收方式

1. 当数据容量用完MemCached分配的内存后，就会基于LRU(Least Recently Used，**最近最少使用**)算法清理失效的缓存数据（*放入数据时可设置失效时间*），或者清理最近最少使用的缓存数据，然后放入新的数据。
2. 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。
3. 需要注意的是，如果如果MemCache启动**没有追加**-M，**则表示禁止**LRU，这种情况下内存不够会报Out Of Memory错误。

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

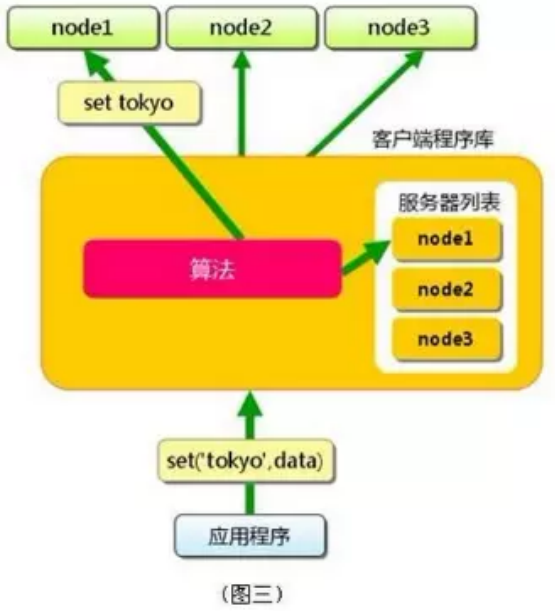
2.1.2.2 Memcached分布式

为了提升MemCached的存储容量和性能，我们应用的客户端可能会对应多个MemCached服务器来提供服务，这就是MemCached的分布式。

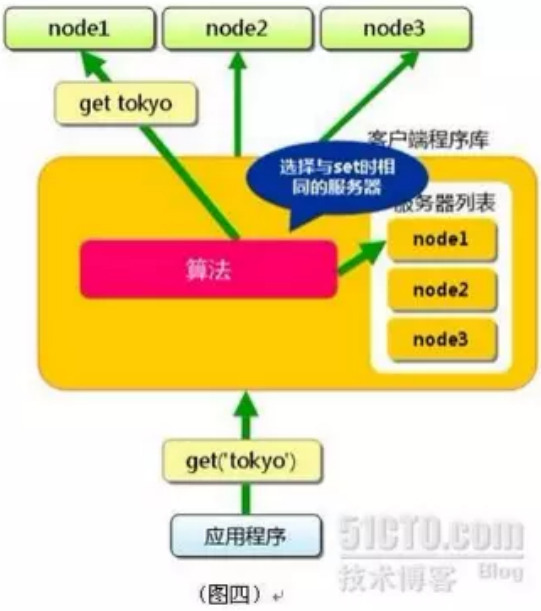
1. 分布式实现原理

MemCached的目前版本是通过C实现，采用了**单进程、多线程、异步**I/O，**基于事件**(event_based)的服务方式.使用libevent作为事件通知实现。多个Server可以协同工作，但这些 Server **之间保存的数据各不相同，而且并不通信**（与之形成对比的，比如JBoss Cache，某台服务器有缓存数据更新时，会通知集群中其他机器更新缓存或清除缓存数据），**每个**Server**只是对自己的数据进行管理。**

Client端通过IP地址和端口号指定Server端，将需要缓存的数据是以key->value**对**的形式保存在Server端。key的值通过hash进行转换，根据hash值把value传递到对应的具体的某个Server上。当需要获取对象数据时，也根据key进行。首先**对**key**进行**hash，**通过获得的值可以确定它被保存在了哪台**Server**上**，然后再向该Server发出请求。Client**端只需要知道保存**hash(key)**的值在哪台服务器上就可以了。**



（图三）



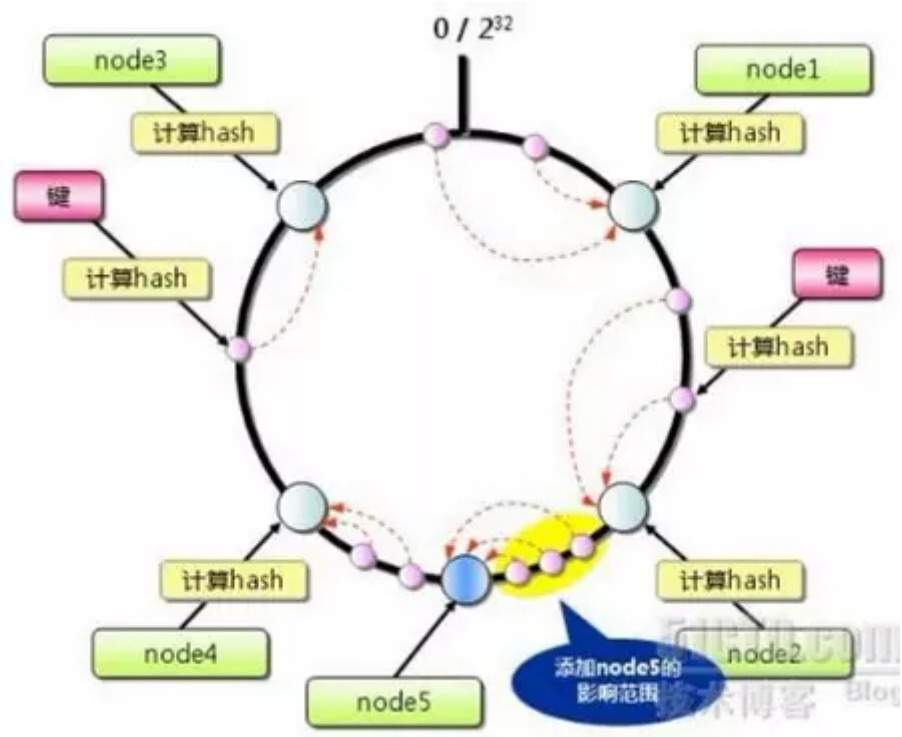
（图四）

2. 分布式算法解析

1. **余数算法：**
   先求得键的整数散列值（也是就键string的HashCode值 什么是HashCode），再除以服务器台

数，**根据余数确定存取服务器**，这种方法计算简单，高效，但**在**memcached**服务器增加或减少时，几乎所有的缓存都会失效。**

2. **散列算法(一致性**hash):
   先算出MemCached**服务器的散列值，并将其分布到**0**到**2^32**的圆上**，然后用同样的方法算出**存储数据**的键的散列值并映射至圆上**，最后**从数据映射到的位置开始顺时针查找，将数据保存到查找到的第一个服务器上**，如果超过2的32次方，依然找不到服务器，就将数据保存到第一台MemCached服务器上。**如果添加了一台**MemCached**服务器，只在圆上增加服务器的逆时针方向的第一台服务器上的键会受到影响。**



2.1.2.3 Memcached线程管理

MemCached网络模型是典型的**单进程多线程模型，采用**libevent**处理网络请求**，**主进程负责将新来的连接分配给**work**线程，**work**线程负责处理连接**，有点类似与负载均衡，通过主进程分发到对应的工作线程。

MemCached**默认有**7**个线程，**4**个主要的工作线程，**3**个辅助线程**，线程可划分为以下4种：

1. **主线程**，负责MemCached服务器初始化，监听TCP、Unix Domain连接请求；
2. **工作线程池**，MemCached默认4个工作线程，可通过启动参数修改，负责处理TCP、UDP，Unix域套接口链路上的请求；
3. assoc**维护线程**，MemCached内存中维护一张巨大的hash表，该线程**负责**hash**表动态增长**；
4. slab**维护线程**，即内存管理模块维护线程，负责class中slab的平衡，MemCached启动选项中可关闭该线程。

2.1.3 MemCached特性与限制

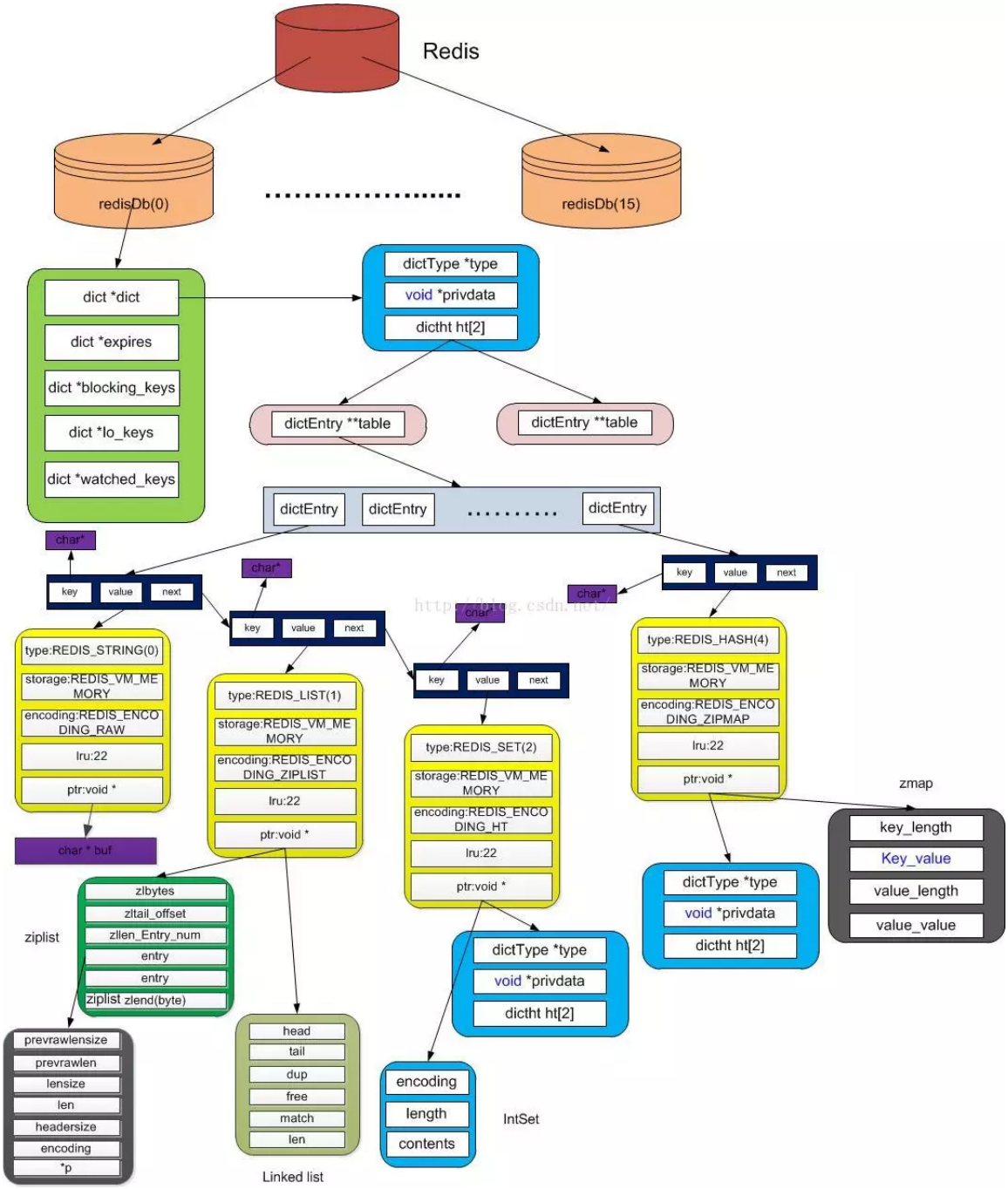
1. MemCache中可以保存的item数据量是没有限制的，只要内存足够
2. MemCache单进程**在32位机中最大使用内存为**2G，64**位机则没有限制**
3. Key**最大为**250**个字节**，超过该长度无法存储
4. 单个item最大数据是1MB，超过1MB的数据不予存储
5. **MemCache服务端是不安全的**，比如已知某个MemCache节点，可以直接telnet过去，并通过flush_all让已经存在的键值对立即失效，所以MemCache服务器最好配置到内网环境，通过防火墙制定可访问客户端
6. **不能够遍历**MemCache**中所有的**item，因为这个操作的速度相对缓慢且会阻塞其他的操作
7. MemCached**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。

2.2 Redis

Redis是一个key-value**存储系统。**和Memcached类似，它支持存储的value类型相对更多，包括string**(字符串)、** list**(链表)、**set**(集合)和**zset**(有序集合)。**这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是**原子性**的。在此基础上，redis支持各种不同方式的排序。与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,当前 Redis的应用已经非常广泛，国内像新浪、淘宝，国外像 Flickr、Github等均在使用Redis的缓存服务。

2.2.1 Redis数据结构

Redis支持丰富的数据类型，并提供了大量简单高效的功能。Redis的底层数据结构总览图：



上图列出了Redis内部底层的一些重要数据结构，包括List, Set, Hash, String等。

1. Redis Object

redisObject定义了**类型，编码方式，**LRU**时间，引用计数，***ptr**指向实际保存值指针。**

- type: redisObject的类型，字符串，列表，集合，有序集，哈希表等
- encoding: 底层实现结构，字符串，整数，跳跃表，压缩列表等
- ptr：实际指向保存值的数据结构

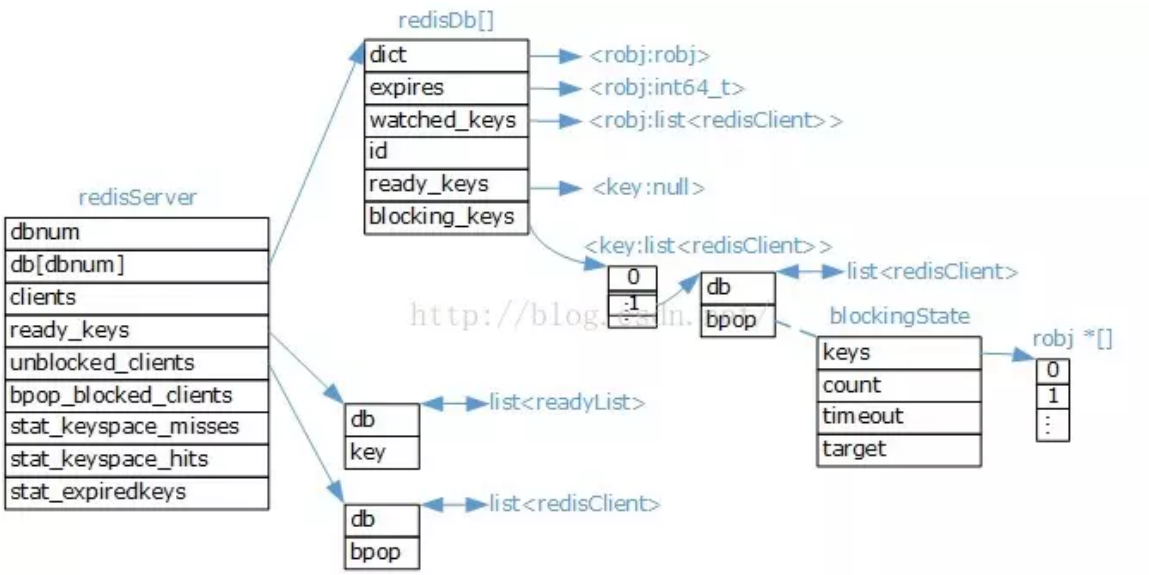
举个具体例子，redisObject{type: REDIS_LIST, encoding:REDIS_ENCODING_LINKEDLIST},

这个对象是Redis列表，其值保存在一个链表中，ptr指针指向这个列表。Redis自己实现对象管理机制，并**基于引用计数的垃圾回收。**Redis提供了incrRefCount与decrRefCount来管理对象跟踪对象的

引用，当减少引用时检测计数器为是否需要释放内存对象。

2. RedisDB

RedisDB内部数据结构，封装了数据库层面的信息：



从redisDB来看，几个重要属性：

- id：数据库内部编号，仅供内部操作使用，如AOF等
- dict：存放整个数据库的键值对，键为字符串，值为Redis的数据结构，如List, Set, Hash等。
- expires：键的过期时间

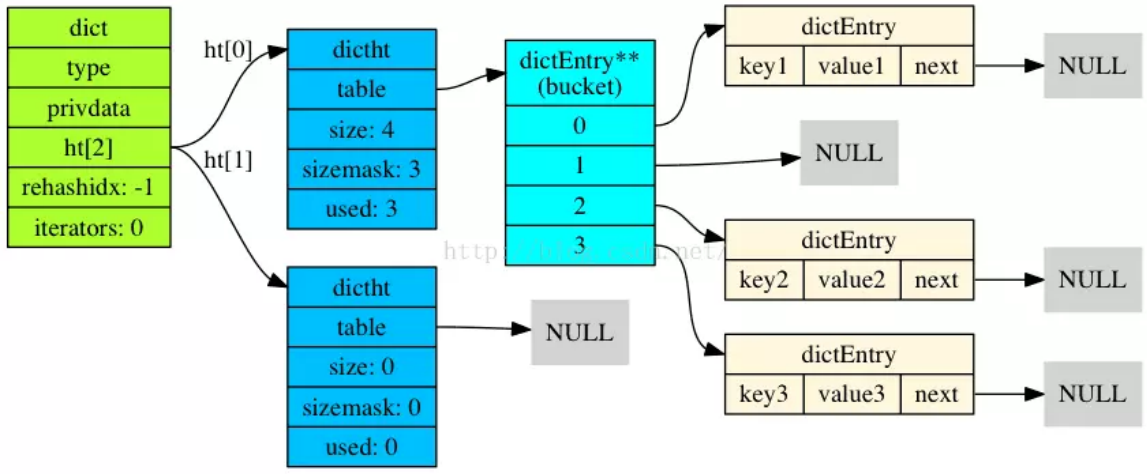
3. RedisServer

RedisServer代码在Redis 3.0支持Cluster后变得较复杂，总体来说包含如下几个大部分：

- 通用部分：如pid进程id，数据库指针，命令字典表，Sentinel模式标志位等
- 网络信息：如port TCP监听端口，Cluster Bus监听socket, 已使用slot数量，Active的客户端列表, 当前客户端，Slaves列表等。
- 其它信息：如AOF信息，统计信息, 配置信息（如已经配置总db数量dbnum等），日志信息，Replication配置，Pub/Sub, Cluster信息，Lua脚本信息配置等等。

4. Redis Hash

Redis的哈希表/字典是其核心数据结构之一，值得深入研究。Redis Hash数据结构， Hash新创建时，在不影响效率情况下，Redis默认使用zipmap作为底层实现以节省空间，只有当size超出一定限制后（hash-max-zipmap-entries ），Redis才会自动把zipmap转换为下图Hash Table。



上图字典的底层实现为哈希表，每个字典包含2个哈希表，ht[0], ht[1], 1号哈希表是在rehash过程中才使用的。而哈希表则由dictEntry构成。

每个字典包含了3个内部数据结构：

- Dict：字典的根结构，包含了2个dictht，其中2作为rehashing之用
- Dictht：包含了linkedlist dictEntry
- DictEntry：包含了3个数据结构（double/uint64_6/int64t）的链表，类似Java HashMap中的Entry结构

5. Hash**算法**

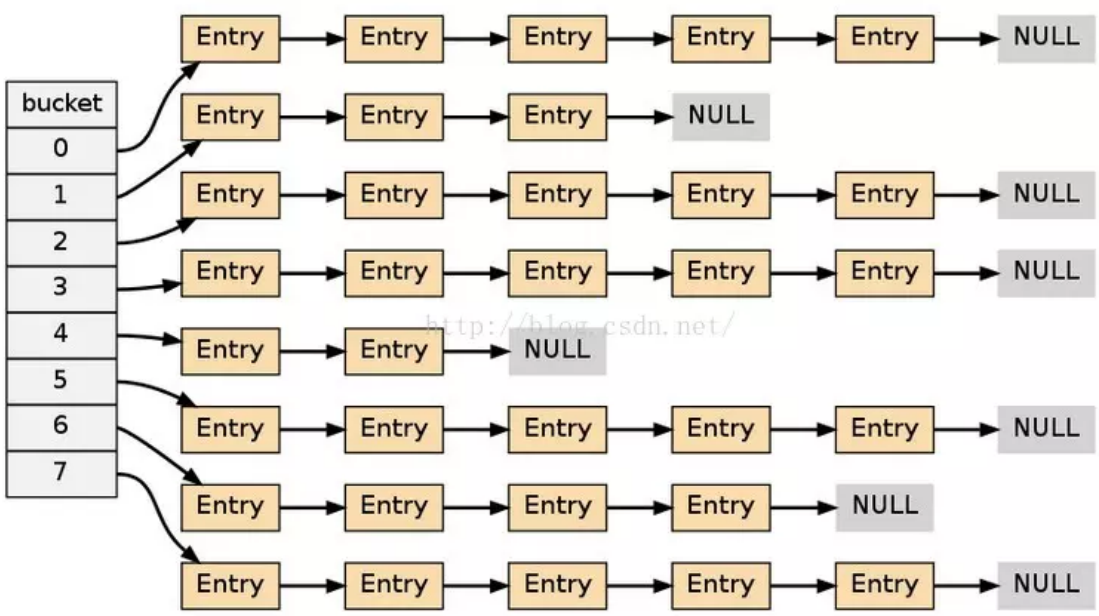
目前Redis中引入了一些经典哈希算法，而HashTable则主要为以下两种：

- MurmurHash2 32bit算法：著名的非加密型哈希函数，能产生32位或64位哈希值，最新版本为MurmurHash3。该算法针对一个字符串进行哈希，可表现较强离散性。
- 基于djb**算法实现散列算法**：该算法较为简单，同样是将字符串转换为哈希值。主要利用字符串中的ASCII码与一个随机seed，进行变换得到哈希值。

评估一个哈希算法的优劣，**主要看其哈希值的离散均匀效果以及消除冲突程度。**

6. Rehash

类似Java中的HashMap, 当有新键值对添加到Redis字典时，有可能会触发rehash。Redis中处理哈希碰撞的方法与Java一样，都是采用**链表法**，整个哈希表的性能则依赖于它的大小size和它已经保存节点数量used的比率。**比率在1:1时，哈希表的性能最好**，如果节点数量比哈希表大小大很多的话，则整个哈希表就退化成多个链表，其性能优势全无。



上图的哈希表，平均每次失败查找需要访问5个节点。为了保持高效性能，在不修改键值对情况下，需要进行rehash，目标是将ratio比率维持在1:1左右。Ratio = Used / Size

rehash**触发条件**：

- 自然rehash：ratio >= 1, 且变量dict_can_resize为真
- 强制rehash：ratio大于dict_force_resize_ratio（v 3.2.1版本为5）

rehash**执行过程**：

- 创建ht[1]并分配至少2**倍于**ht[0] table**的空间**
- 将ht[0] table中的所有键值对迁移到ht[1] table
- 将ht[0]数据清空，并将ht[1]替换为新的ht[0]

Redis哈希为了避免整个rehash过程中服务被阻塞，采用了渐进式的rehash，即**rehash程序激活后，并不是马上执行直到完成，而是分多次，渐进式（**incremental**）的完成。**同时，为了保证并发安全，**在执行**rehash**中间执行添加时，新的节点会直接添加到**ht[1]**而不是**ht[0], **这样保证了数据的完整性与安全性。**另一方面，哈希的Rehash在还提供了创新的（相对于Java HashMap**）收缩（**shrink**）字典**，当可用节点远远大于已用节点的时候，rehash**会自动进行收缩**，具体过程与上面类似以保证比率始终高效使用。

2.2.2 Redis持久化

持久化简单来讲就是将数据放到断电后数据不会丢失的设备中，也就是我们通常理解的硬盘上。

首先我们来看一下数据库在进行写操作时到底做了哪些事，主要有下面五个过程：

- 客户端向服务端发送写操作（数据在客户端的内存中）。

- 数据库服务端接收到写请求的数据（数据在服务端的内存中）。
- 服务端调用write这个系统调用，将数据往磁盘上写（数据在系统内存的缓冲区中）。
- 操作系统将缓冲区中的数据转移到磁盘控制器上（数据在磁盘缓存中）。
- 磁盘控制器将数据写到磁盘的物理介质中（数据真正落到磁盘上）。

通过上面5步的了解，可能我们会希望搞清下面一些问题：

- 数据库多长时间调用一次write，将数据写到内核缓冲区?
- 内核多长时间会将系统缓冲区中的数据写到磁盘控制器?
- 磁盘控制器又在什么时候把缓存中的数据写到物理介质上?
- 对于第一个问题，通常数据库层面会进行全面控制。
- 对第二个问题，操作系统有其默认的策略，但是我们也可以通过POSIX API提供的fsync系列命令强制操作系统将数据从内核区写到磁盘控制器上。
- 对于第三个问题，好像数据库已经无法触及，但实际上，大多数情况下磁盘缓存是被设置关闭的，或者是只开启为读缓存，也就是说写操作不会进行缓存，直接写到磁盘。**建议的做法是仅仅当你的磁盘设备有备用电池时才开启写缓存。**

数据损坏

所谓数据损坏，就是数据无法恢复，上面我们讲的都是如何保证数据是确实写到磁盘上去，但是写到磁盘上可能并不意味着数据不会损坏。比如我们可能一次写请求会进行两次不同的写操作，当意外发生时，可能会导致一次写操作安全完成，但是另一次还没有进行。如果数据库的数据文件结构组织不合理，可能就会导致数据完全不能恢复的状况出现。

这里通常也有三种策略来组织数据，以防止数据文件损坏到无法恢复的情况：

- 第一种是最粗糙的处理，就是不通过数据的组织形式保证数据的可恢复性。而是通过配置**数据同步备份**的方式，在数据文件损坏后通过数据备份来进行恢复。实际上MongoDB在不开启操作日志，通过配置Replica Sets时就是这种情况。
- 另一种是在上面基础上添加一个**操作日志**，每次操作时记一下操作的行为，这样我们可以通过操作日志来进行数据恢复。因为操作日志是顺序追加的方式写的，所以不会出现操作日志也无法恢复的情况。这也类似于MongoDB开启了操作日志的情况。
- 更保险的做法是数据库不进行旧数据的修改，只是**以追加方式去完成写操作**，这样数据本身就是一份日志，这样就永远不会出现数据无法恢复的情况了。实际上CouchDB就是此做法的优秀范例。

2.2.2.1 Redis提供了RDB持久化和AOF持久化

RDB持久化是指在指定的时间间隔内**将内存中的数据集快照写入磁盘。**

也是默认的持久化方式，这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

可以通过配置设置自动做快照持久化的方式。我们可以**配置**redis**在**n**秒内如果超过**m**个**key**被修改就自动做快照**，下面是默认的快照保存配置

```
save 900 1    #900秒内如果超过1个key被修改，则发起快照保存
save 300 10   #300秒内容如超过10个key被修改，则发起快照保存
save 60 10000
```

1. RDB文件保存过程

- redis调用fork,现在有了子进程和父进程，此时服务进程将有短暂的停顿。
- 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的**写时复制机制（copy on write)**父子进程会共享相同的物理页面，当父进程处理写请求时os会**为父进程要修改的页面创建副本**，而不是写共享的页面。所以子进程的地址空间内的数据是fork时刻整个数据库的一个快照。
- 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

client 也可以**使用**save**或者**bgsave**命令通知**redis**做一次快照持久化。**save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。

另一点需要注意的是，**每次快照持久化都是将内存数据完整写入到磁盘一次**，并不是增量的只同步脏数据。**如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘**io**操作，可能会严重影响性能。**

优势

- 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这样非常方便进行备份。比如你可能打算没1天归档一些数据。
- 方便备份，我们可以很容易的将一个一个RDB文件移动到其他的存储介质上
- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

劣势

- 如果你需要尽量避免在**服务器故障时丢失数据**，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。
- 每次保存 RDB 的时候，Redis **都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时，** fork() **可能会非常耗时，造成服务器在某某毫秒内停止处理客户端**； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

2. AOF文件保存过程

redis会将每一个收到的**写命令都通过**write**函数追加到文件中**(默认是 appendonly.aof)。

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化**也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉**redis**我们想要 通过**fsync**函数强制**os**写入到磁盘的时机**。有三种方式如下（默认是：每秒fsync一次）

```
appendonly yes //启用aof持久化方式

# appendfsync always //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用

# appendfsync everysec //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐

# appendfsync no //完全依赖os，性能最好,持久化没保证
```

aof 的方式也同时带来了另一个问题。**持久化文件会变的越来越大。**例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis**提供了**bgrewriteaof**命令。收到此命令**redis**将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。**具体过程如下

- redis调用fork ，现在有父子两个进程
- 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

- 父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。
- 当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。
- 现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。

优势

- 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync，每秒钟一次 fsync，或者每次执行写入命令时 fsync。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
- AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof **工具也可以轻易地修复这种问题。**
  Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。
- AOF **文件有序地保存了对数据库执行的所有写入操**作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

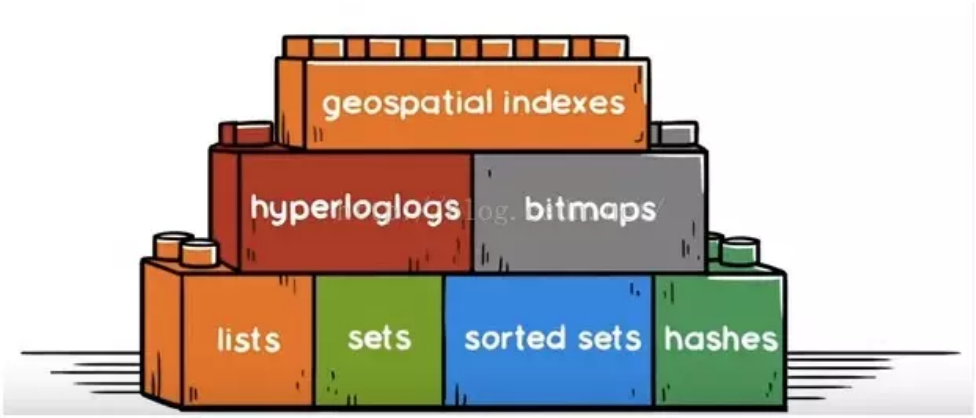
劣势

- 对于相同的数据集来说，AOF **文件的体积通常要大于** RDB **文件的体积。**
- 根据所**使用的** fsync **策略**，AOF **的速度可能会慢于** RDB。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。

- AOF 在过去曾经发生过这样的 bug： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

Redis v.s. Memcached

- **数据类型与操作**：Redis拥有更多丰富的数据结构支持与操作，而Memcached则需客户端自己处理并进行网络交互



- **内存使用率**：简单K/V存储，Memcached内存利用率更高（使用了slab与大小不同的chunk来管理内存），而如果采用Redis Hash来存储则其组合压缩，内存利用率高于Memcached
- **性能**：总体来说，二者性能接近；Redis使用了单核（单线程IO复用，封装了AeEvent事件处理框架，实现了epoll,kqueue,select），Memcached采用了多核，各有利弊；当数据大于100K的时候，Memcached性能高于Redis
- **数据持久化**：Redis支持数据文件持久化，RDB与AOF两种策略；Memcached则不支持
- **分布式**：Memcached本身并不支持服务器端分布式，客户端只能借助一致性哈希分布式算法来实现Memcached分布式存储；当然Redis也是从3.0版本开始才支持服务器端cluster的，重要的是现在支持了。
- **其他方面**：Redis提供其他一些功能，如Pub/Sub, Queue, 简单Transacation, Replication等。

Redis v.s. HashMap

HashMap**单机受限于内存容量**，而正是Redis分布式之优势

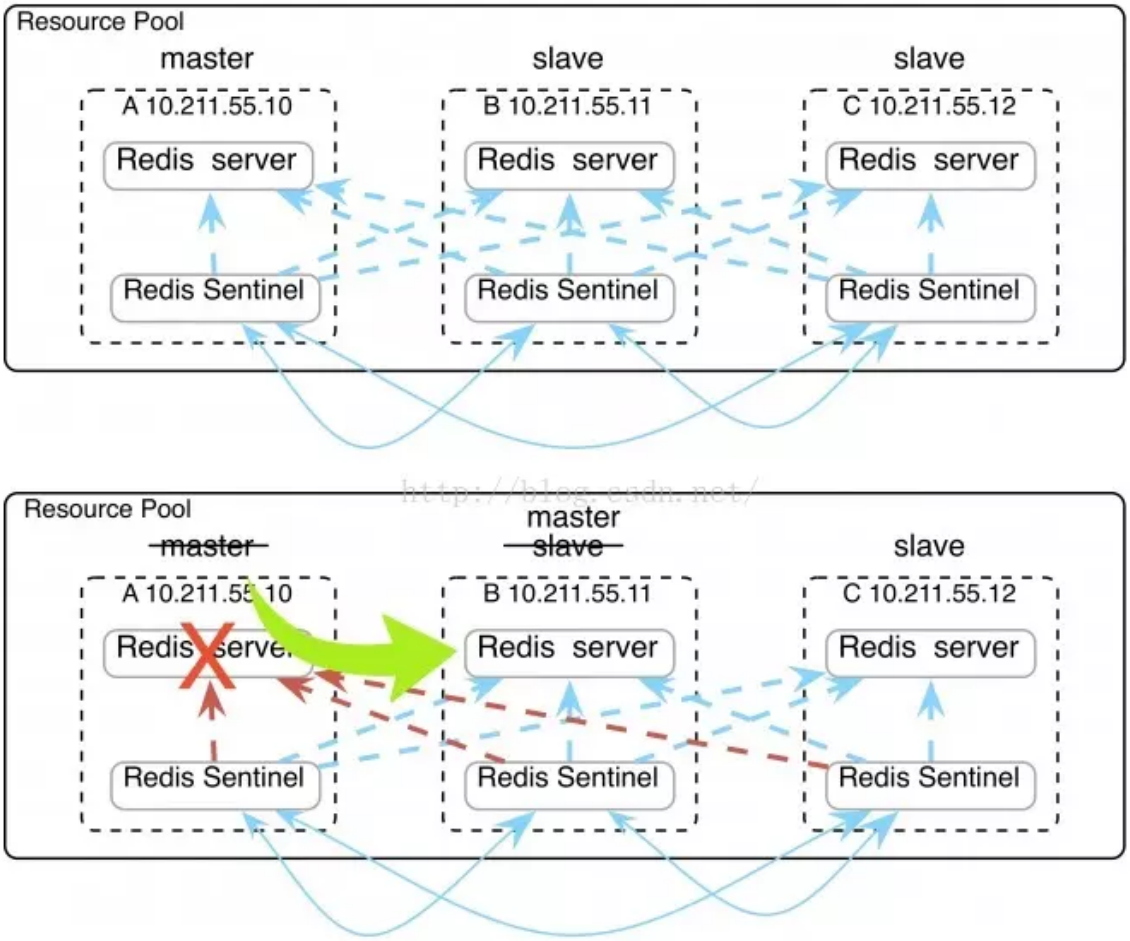
- HashMap当数据量超过一定限制后，需要妥善管理堆内存，不然会造成内存溢出或者Memory Leak；Redis则具备了文件持久性，以及Failover达到HA.
- HashMap只能受限于本机，而Redis天生分布式，可以让多个App Server访问，负载均衡。

所以Redis适合全部数据都在内存的场景包括需要临时持久化，尤其作为缓存来使用，并支持对缓存数据进行简单处理计算；如涉及Redis与RDBMS双向同步的话，则需要引入一些复杂度。

2.2.3 Redis系统架构

Sentinel架构

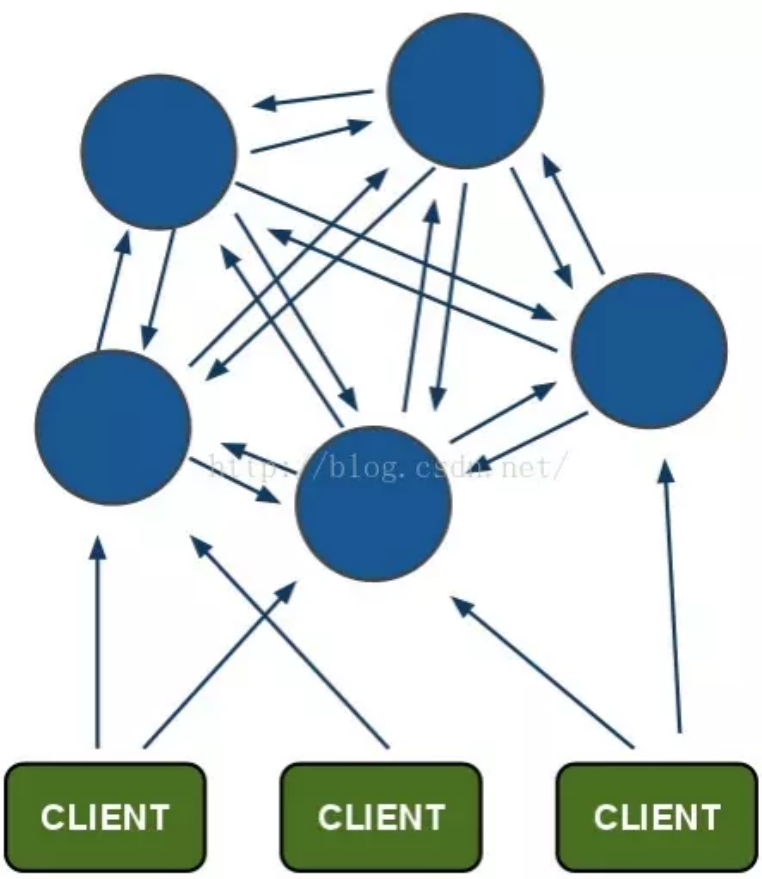
Reids Sentinel诞生于2012年（Redis 2.4版本），建议在单机Redis或者客户端模式Cluster的时候（非 3.0版本Redis Cluster）采用，作为HA, Failover来使用。



Sentinel**主要提供了集群管理，包括监控，通知，自动故障恢复。**如上图，当其中一个master无法正常工作时，Sentinel将把一个Slave提升为Master, 从而自动恢复故障。而Sentinel本身也做到了分布式，可以部署多个Sentinel实例来监控Redis实例（建议基数，至少3个Sentinel实例来监控一组Redis Master/Slaves），多个Sentinel进程间通过Gossip协议来确定Master是否宕机，通过Agreement协议来决定是否执行故障自动迁移以及重新选主，整体设计类似ZooKeeper）。

Cluster架构

所谓分布式即支持数据分片，并且自动管理故障恢复（failover）与备份（replication）。



如上图Redis Cluster采用了无中心结构，每个节点都保存，共享数据和集群状态，每个节点与其它所有节点通信，**使用Gossip协议来传播及发现新节点，通过分区来提供一定程度可用性，当某个node的Master宕机时，Cluster会自动选举一个Slave形成一个新的Master**，这里应该是

借鉴，重用了Sentinel的功能。另外，Redis Cluster**并没有使用通常的一致性哈希，而引入哈希槽的概念**，Cluster中固定有16384个slot，每个key通过CRC16校验后对16384取模来决定其对应slot的位置，而每个node负责一部分的slot管理，**当node变化时，动态调整slot的分布，而数据则无须挪动。对于客户端来说，client可以向任意一个实例请求，Cluster会自动定位需要访问的slot**。

上图查询路由过程中，我们随机发送到任意一个Redis实例，这个实例会按照上文提到的CRC16校验后取模定位，并转发至正确的Redis实例中。

```
slot = crc16("foo") mod NUMER_SLOTS
```