Tokyocabinet/Tokyotyrant文档大合集

[原文:整理于网络 整理:一米六二<xurenlu@gmail.com> 网络更新:http://www.162cm.com/p/tokyotyrant.html]

Last Updated: 2009-12-04

- 1. 前言
- 2. 参考资料链接
- 3. 使用介绍
  - 3.1. 基本概念
  - 3.2. Tokyo Cabinet 简介
  - 3.3. 性能介绍
  - 3.4. tokyotyrant和Memcached的优势比较
    - 3.4.1. 故障转移
    - 3.4.2. 日志文件体积小
    - 3.4.3. 超大数据量下表现出色
  - 3.5. 安装
    - 3.5.1. 编译安装tokyocabinet数据库
    - 3.5.2. 编译安装tokyotyrant

- 3.6. tokyotyrant的配置
  - 3.6.1. 创建tokyotyrant数据文件存放目录
  - 3.6.2. 启动tokyotyrant的主进程（ttserver）
  - 3.6.3. 停止tokyotyrant（ttserver）
- 3.7. 调用

- 4. 程序架构
  - 4.1. 流程介绍
    - 4.1.1. 多线程
    - 4.1.2. TokyoTyrant vs. Memcached
    - 4.1.3. 启动流程
    - 4.1.4. 请求处理
    - 4.1.5. 数据结构
- 5. 数据库存储基础
  - 5.1. tokyocabinet的源代码结构
  - 5.2. tokyotyrant的存储类型
  - 5.3. tokyotyrant的缓存
  - 5.4. 异步
  - 5.5. 索引
  - 5.6. 数据的Hash
    - 5.6.1. 冷存储
      - 5.6.1.1. 一级hash索引：bidx
      - 5.6.1.2. 二级hash索引：hash
      - 5.6.1.3. key值对比

        - 5.6.1.4. 存储时的主要逻辑
        - 5.6.1.5. 数据文件结构
        - 5.6.1.6. 内存映射的一级索引
        - 5.6.1.7. bnum参数
        - 5.6.1.8. 预告
- 6. 线程和事件
    - 6.1. 工作线程
        - 6.1.1. 什么是工作线程组
        - 6.1.2. 从这里开始
        - 6.1.3. ttservdeqtasks的工作过程
            - 6.1.3.1. **存取请求**
            - 6.1.3.2. 线程处理请求
- 7. MemcacheDB,Tokyo Tyrant和Redis 性能对比测试
    - 7.1. 测试环境
        - 7.1.1. 软件环境
        - 7.1.2. 配置
        - 7.1.3. 测试客户端
    - 7.2. 小数据量测试结果
    - 7.3. 大数据量测试结果
    - 7.4. Some notes about the test
- 8. Tokyo Tyrant 的问题和Bug
    - 8.1. Bug report
    - 8.2. tokyotyrant大规模出错的问题

- 8.3. Bugs
- 9. 延伸阅读:key-value-pair database的比较
  - 9.1. 满足极高读写性能需求的Kye-Value数据库：Redis，Tokyo Cabinet，Flare
    - 9.1.1. Redis
    - 9.1.2. Tokyo Cabinet和Tokoy Tyrant
    - 9.1.3. Flare
  - 9.2. 满足海量存储需求和访问的面向文档的数据库：MongoDB，CouchDB
    - 9.2.1. MongoDB
    - 9.2.2. CouchDB
  - 9.3. 满足高可扩展性和可用性的面向分布式计算的数据库：Cassandra，Voldemort
    - 9.3.1. Cassandra
    - 9.3.2. Voldemort

---

1. 前言

这里不是我个人原创,是我对网络上整理到的资料的再加工,以更成体系,更方便研究阅读.主要是对其中跟主题无关的文字删除,部分人称稍做修改;本人无版权,您可以将本页面视为对参考页面的镜像.第二部分已经给出所有的参考资料;

2. 参考资料链接

- 利用Tokyo Tyrant构建兼容Memcached协议、支持故障转移、高并发的分布式key-value持久存储系统[原创]》:[http://blog.s135.com/post/362/]
- tokyotyrant源代码研究-程序架构与运行流程:[http://lgone.com/blog/html/y2009/302.html]
- tokyocabinet的hash存储机制:[http://lgone.com/blog/html/y2009/529.html]
- tokyocabinet源代码研究：存储机制:[http://lgone.com/blog/html/y2009/505.html]
- tokyotyrant源代码研究-第三线程:[http://lgone.com/blog/html/y2009/342.html]
- tokyotyrant源代码研究-工作线程组:[http://lgone.com/blog/html/y2009/348.html]
- tokyotyrant大规模出错的问题:[http://lgone.com/blog/html/y2009/491.html]
- MemcacheDB, Tokyo Tyrant, Redis performance test: [http://timyang.net/data/mcdb-tt-redis/]
- robbin,为什么要用非关系数据库;[http://robbin.javaeye.com/blog/524977]

3. 使用介绍

3.1. 基本概念

- tokyocabinet :一个key-value的DBM数据库，但是没有提供网络接口，以下称TC。
- tokyotyrant :是为TC写的网络接口，他支持memcache协议，也可以通过HTTP操作，以下称TT。

3.2. Tokyo Cabinet 简介

1. 项目主页:http://tokyocabinet.sourceforge.net/

2. 简介

Tokyo Cabinet 是一个DBM的实现。这里的数据库由一系列key-value对的记录构成。key和value都可以是任意长度的字节序列,既可以是二进制也可以是字符串。这里没有数据类型和数据表的概念。 当做为Hash表数据库使用时，每个key必须是不同的,因此无法存储两个key相同的值。提供了以下访问方法:提供key,value参数来存储，按key删除记录，按key来读取记录，另外，遍历key也被支持，虽然顺序是任意的不能被保证。这些方法跟Unix标准的DBM,例如GDBM,NDBM等等是相同的，但是比它们的性能要好得多（因此可以替代它们)

当按B+树来存储时，拥用相同key的记录也能被存储。像hash表一样的读取，存储，删除函数也都有提供。记录按照用户提供的比较函数来存储。可以采用顺序或倒序的游标来读取每一条记录。依照这个原理，向前的字符串匹配搜索和整数区间搜索也实现了。另外，B＋树的事务也是可用的。 对于定长的数组，记录按自然数来标记存储。不能存储key相同的两条或更多记录。另外，每条记录的长度受到限 制。读取方法和hash表的一样。

Tokyo Cabinet是用C写的，同时提供c,perl,ruby,java的API。Tokyo Cabinet在提供了POSIX和C99的平台上都可用，它以GNU Lesser Public License协议发布。

3.3. 性能介绍

Tokyo Cabinet 是日本人 平林幹雄 开发的一款 DBM 数据库，该数据库读写非常快，哈希模式写入100万条数据只需0.643秒，读取100万条数据只需0.773秒，是 Berkeley DB 等 DBM 的几倍。

Benchmark Test of DBM Brothers

This benchmark test is to calculate processing time (real time) and file size of database.
Writing test is to store 1,000,000 records. Reading test is to fetch all of its records.
Both of the key and the value of each record are such 8-byte strings as `00000001', `00000002', `00000003'...
Tuning parameters of each DBM are set to display its best performance.
Platform: Linux 2.6.16 kernel, EXT3 file system (writeback), Intel Xeon quad core 2.3GHz CPU, 8GB RAM
Compilation: gcc 4.2.3 (using -O3), glibc 2.7

Result

NAME	DESCRIPTION	WRITE TIME	READ TIME	FILE SIZE
TC	Tokyo Cabinet 1.2.9	0.643	0.773	42,583,208
QDBM	Quick Database Manager 1.8.77	2.813	0.959	56,582,932
NDBM	New Database Manager 5.1	5.183	3.538	834,003,968
SDBM	Substitute Database Manager 1.0.2	6.202	0.000	621,281,280
GDBM	GNU Database Manager 1.8.3	18.878	3.119	88,137,728
TDB	Trivial Database 1.0.6	7.023	0.789	52,523,008
CDB	Tiny Constant Database 0.75	0.352	0.370	40,002,048
BDB	Berkeley DB 4.6.21	9.665	3.118	41,938,944
TC-BT-ASC	B+ tree API of TC (ascending order)	1.039	0.787	32,209,795
TC-BT-RND	B+ tree API of TC (at random)	4.075	3.114	12,466,925
QDBM-BT-ASC	B+ tree API of QDBM (ascending order)	1.401	0.955	40,620,715
QDBM-BT-RND	B+ tree API of QDBM (at random)	6.324	3.837	15,731,675
BDB-BT-ASC	B+ tree API of BDB (ascending order)	1.367	1.449	57,999,360
BDB-BT-RND	B+ tree API of BDB (at random)	3.659	2.316	29,818,880
TC-FIXED	Fixed-length API of TC	0.240	0.037	9,000,256

Unit of time is seconds. Unit of size is bytes.
Read time of SDBM can not be calculated because its database is broken when more than 100000 records.

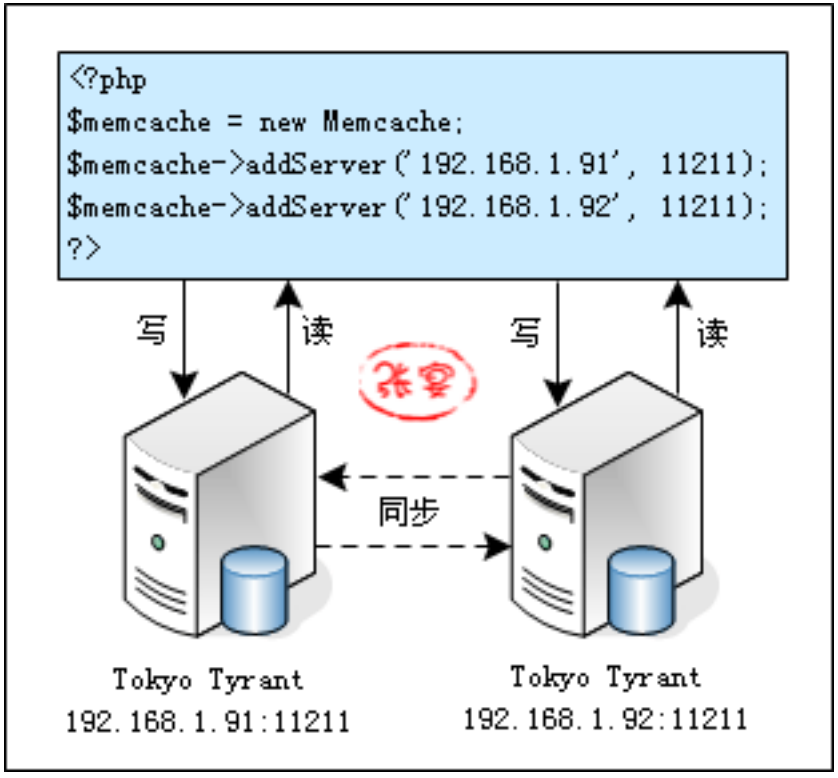
Tokyo Tyrant 加上 Tokyo Cabinet，构成了一款支持高并发的分布式持久存储系统，对任何原有Memcached客户端来讲，可以将Tokyo Tyrant看成是一个Memcached，但是，它的数据是可以持久存储的。这一点，跟新浪的Memcachedb性质一样。

3.4. tokyotyrant和Memcached的优势比较

相比Memcachedb而言，Tokyo Tyrant具有以下优势:

3.4.1. 故障转移

Tokyo Tyrant支持双机互为主辅模式，主辅库均可读写，而Memcachedb目前支持类似MySQL主辅库同步的方式实现读写分离，支持“主服务器可读写、辅助服务器只读”模式。



这里使用 $memcache->addServer 而不是 $memcache->connect 去连接 Tokyo Tyrant 服务器，是因为当 Memcache 客户端使用 addServer 服务器池时，是根据“crc32(key) % current_server_num”哈希算法将 key 哈希到不同的服务器的，PHP、C 和 python 的客户端都是如此的算法。Memcache 客户端的 addserver 具有故障转移机制，当 addserver 了2台 Memcached 服务器，而其中1台宕机了，那么 current_server_num 会

由原先的2变成1。

引用 memcached 官方网站和 PHP 手册中的两段话：

```
    http://www.danga.com/memcached/
    If a host goes down, the API re-maps that dead host's requests onto the
 servers that are available.



    http://cn.php.net/manual/zh/function.Memcache-addServer.php
    Failover may occur at any stage in any of the methods, as long as other
 servers are available the request the user won't notice. Any kind of socke
t or Memcached server level errors (except out-of-memory) may trigger the f
ailover. Normal client errors such as adding an existing key will not trigg
er a failover.
```

3.4.2. 日志文件体积小

Tokyo Tyrant用于主辅同步的日志文件比较小，大约是数据库文件的1.3倍，而Memcachedb的同步日志文件非常大，如果不定期清理，很容易将磁盘写满

3.4.3. 超大数据量下表现出色

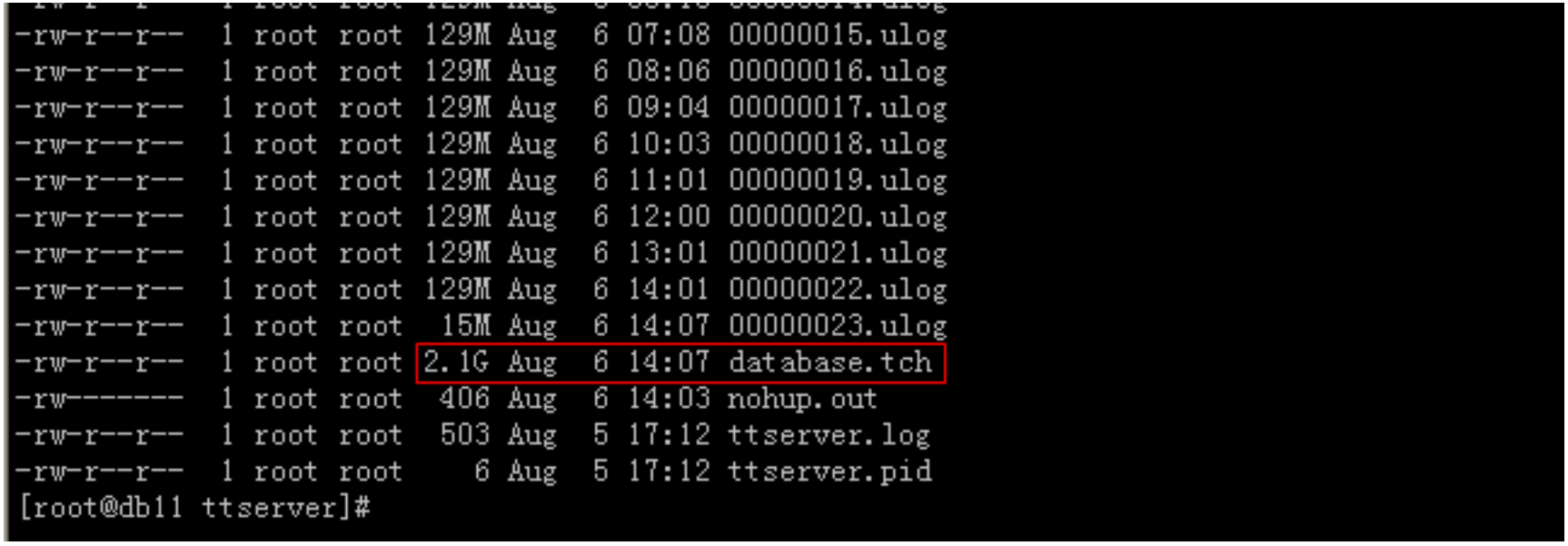
```
[root@db11 ttserver]# uname -a
Linux db11 2.6.9-67.ELsmp #1 SMP Wed Nov 7 13:56:44 EST 2007 x86_64 x86_64 x86_64 GNU/Linux
[root@db11 ttserver]# telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
stats
STAT pid 15047
STAT uptime 75305
STAT time 1218002842
STAT version 1.0.0
STAT rusage_user 898.872350
STAT rusage_system 3459.746038
STAT curr_items 67407431
STAT bytes 2157107744
END
get zhangyan
VALUE zhangyan 0 3
abc
END
quit
Connection closed by foreign host.
[root@db11 ttserver]# ls -lh
total 4.8G
-rw-r--r--  1 root root 129M Aug  5 17:45 00000001.ulog
-rw-r--r--  1 root root 129M Aug  5 18:36 00000002.ulog
-rw-r--r--  1 root root 129M Aug  5 19:32 00000003.ulog
-rw-r--r--  1 root root 129M Aug  5 20:30 00000004.ulog
-rw-r--r--  1 root root 129M Aug  5 21:27 00000005.ulog
-rw-r--r--  1 root root 129M Aug  5 22:25 00000006.ulog
-rw-r--r--  1 root root 129M Aug  5 23:24 00000007.ulog
-rw-r--r--  1 root root 129M Aug  6 00:21 00000008.ulog
-rw-r--r--  1 root root 129M Aug  6 01:19 00000009.ulog
-rw-r--r--  1 root root 129M Aug  6 02:21 00000010.ulog
-rw-r--r--  1 root root 129M Aug  6 03:20 00000011.ulog
-rw-r--r--  1 root root 129M Aug  6 04:18 00000012.ulog
-rw-r--r--  1 root root 129M Aug  6 05:14 00000013.ulog
-rw-r--r--  1 root root 129M Aug  6 06:10 00000014.ulog
```


图例：橙色下划线表示从键盘输入的命令，红色方框表示需要注意的屏幕显示信息。

tokyotyrant（ttserver）每秒可以处理10000次请求。

在64位Linux操作系统下，向tokyotyrant（ttserver）插入了接近7千万条（67407431）数据，数据库文件database.tch的大小达到2.1G（2157107744字节），用Memcache协议进行get、set等请求，速度仍然飞快。

而tokyotyrant（ttserver）的日志文件所占空间也很小，只是database.tch的1.3倍。



但是，Tokyo Tyrant 也有缺点： 在32位操作系统下，作为 Tokyo Tyrant 后端存储的 Tokyo Cabinet 数据库单个文件不能超过2G ，而64位操作系统则不受这一限制。所以，如果使用 Tokyo Tyrant，推荐在64位CPU、操作系统上安装运行。

3.5. 安装

notice

- ** 这里假定了tokyotyrant的版本是1.1.29,请注意.**
- ** 请先安装tokyocabinet,再安装tokyotyrant.后者依赖前者**

3.5.1. 编译安装tokyocabinet数据库

```
wget http://tokyocabinet.sourceforge.net/tokyocabinet-1.4.28.tar.gz
tar zxvf tokyocabinet-1.4.28.tar.gz
```

```
cd tokyocabinet-1.4.28/
./configure
make
make install
cd ../
```

3.5.2. 编译安装tokyotyrant

```
wget http://tokyocabinet.sourceforge.net/tyrantpkg/tokyotyrant-1.1.29.tar.gz
tar zxvf tokyotyrant-1.1.29.tar.gz
cd tokyotyrant-1.1.29/
./configure
make
make install
cd ../
```

3.6. tokyotyrant的配置

3.6.1. 创建tokyotyrant数据文件存放目录

```
```

```
mkdir -p /ttserver/
```

3.6.2. 启动tokyotyrant的主进程（ttserver）

1. 单机模式

```
ulimit -SHn 51200
ttserver -host 127.0.0.1 -port 11211 -thnum 8 -dmn -pid /ttserver/ttse
rver.pid -log /ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128m -
sid 1 -rts /ttserver/ttserver.rts /ttserver/database.tch
```

2. 双机互为主辅模式 服务器192.168.1.91:

```
ulimit -SHn 51200
ttserver -host 192.168.1.91 -port 11211 -thnum 8 -dmn -pid /ttserver/t
tserver.pid -log /ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128
m -sid 91 -mhost 192.168.1.92 -mport 11211 -rts /ttserver/ttserver.rts
 /ttserver/database.tch
```

服务器192.168.1.92:

```
ulimit -SHn 51200
ttserver -host 192.168.1.92 -port 11211 -thnum 8 -dmn -pid /ttserver/t
```

```
tserver.pid -log /ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128
m -sid 92 -mhost 192.168.1.91 -mport 11211 -rts /ttserver/ttserver.rts
 /ttserver/database.tch
```

3. 参数说明

ttserver -host -port -thnum -tout [-dmn] -pid -log [-ld|-le] -ulog -ulim [-uas] -sid -mhost -mport -rts [dbname]

-host name : 指定需要绑定的服务器域名或IP地址。默认绑定这台服务器上的所有IP地址。

-port num : 指定需要绑定的端口号。默认端口号为1978

-thnum num : 指定线程数。默认为8个线程。

-tout num : 指定每个会话的超时时间（单位为秒）。默认永不超时。

-dmn : 以守护进程方式运行。

-pid path : 输出进程ID到指定文件（这里指定文件名）。

-log path : 输出日志信息到指定文件（这里指定文件名）。

-ld : 在日志文件中还记录DEBUG调试信息。

-le : 在日志文件中仅记录错误信息。

-ulog path : 指定同步日志文件存放路径（这里指定目录名）。

-ulim num : 指定每个同步日志文件的大小（例如128m）。

-uas : 使用异步IO记录更新日志（使用此项会减少磁盘IO消耗，但是数据会先放在内存中，不会立即写入磁盘，如果重启服务器或ttserver进程被kill掉，将导致部分数据丢失。一般情况下不建议使用）。

-sid num : 指定服务器ID号（当使用主辅模式时，每台ttserver需要不同的ID号）

-mhost name : 指定主辅同步模式下，主服务器的域名或IP地址。

-mport num : 指定主辅同步模式下，主服务器的端口号。

-rts path : 指定用来存放同步时间戳的文件名。

如果使用的是哈希数据库，可以指定参数“#bnum=xxx”来提高性能。它可以指定bucket存储桶的数量。例如指定“#bnum=1000000”，就可以将最新最热的100万条记录缓存在内存中：

```
ttserver -host 127.0.0.1 -port 11211 -thnum 8 -dmn -pid /ttserver/ttse
rver.pid -log /ttserver/ttserver.log -le -ulog /ttserver/ -ulim 128m -
sid 1 -rts /ttserver/ttserver.rts /ttserver/database.tch#bnum=1000000
```

如果大量的客户端访问ttserver，请确保文件描述符够用。许多服务器的默认文件描述符为1024，可以在启动ttserver前使用ulimit命令提高这项值。例如：

```
ulimit -SHn 51200
```

3.6.3. 停止tokyotyrant（ttserver）

```
ps -ef | grep ttserver
```

找到ttserver的进程号并kill，例如：

```
kill -TERM 2159
```

3.7. 调用

1. 任何Memcached客户端均可直接调用tokyotyrant。
2. 还可以通过HTTP方式调用，下面以Linux的curl命令为例，介绍如何操作tokyotyrant;
   - 写数据:将数据“value”写入到“key”中：

     ```
     curl -X PUT http://127.0.0.1:11211/key -d "value"
     ```
   - 读数据，读取“key”中数据:

     ```
     curl http://127.0.0.1:11211/key
     ```
   - 删数据，删除“key”:

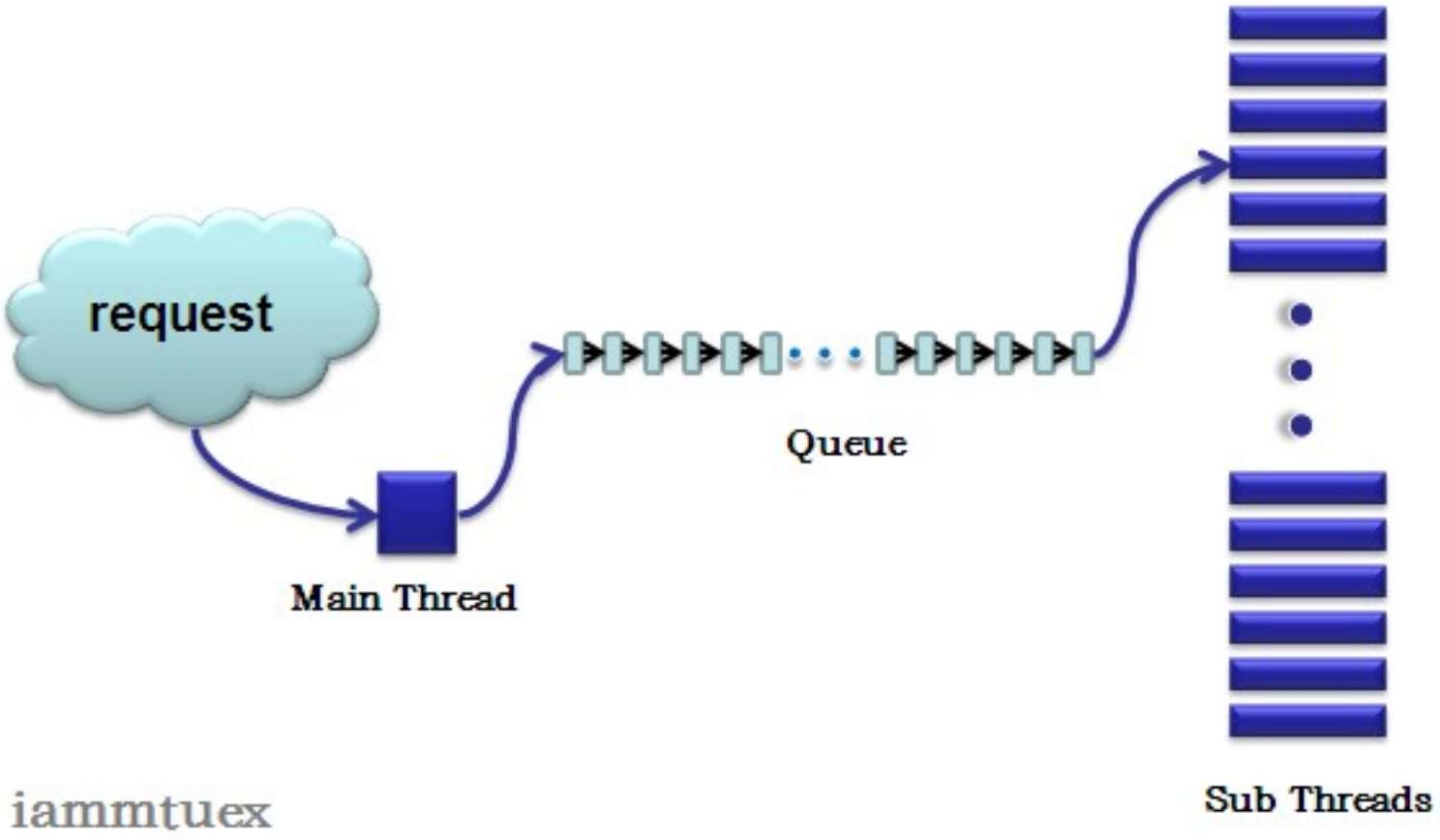
     ```
     curl -X DELETE http://127.0.0.1:11211/key
     ```

4. 程序架构

4.1. 流程介绍

总的说来，从编码方式上来看，tokyocabinet是一个很优秀的代码，整体架构条理十分

清楚，看起来不很费劲,下面先看一个图，大概说一下TT的工作流程。



4.1.1. 多线程

主线程通过网络接口侦听请求连接，然后将得到的请求放到一个全局的队列中，然后work线程从队列头取出请求进行处理并返回给用户。另外还有一个timer线程，他是用于全局做timeout检测的，这个后面再具体说。

4.1.2. TokyoTyrant vs. Memcached

现在对TT的处理流程有了一个大概了解，其实可以把它和memcache对比一下，他们都支持比较高的并发，他用的是linux原生的epoll.而memcache是用的libevent库，在linux下也是用的epoll。这种异步事件处理机制可以说是专为高并发而生。

不同的是memcache的实现上，每一个线程有一个自己的处理队列，而TT中是总的用一个队列，其实大家可以想像，肯定TT在处理上就会不如MC了。那是当然的，MC是存内存，所以他后面的线程们可以工作得很快，如果上面这样设计，那么如果队列同样长，就会导致经常空队列的情况，但是太长可能又会占用太多资源，而且线程间的同步调度又会需要浪费更多的时间在上面。总的来说是瓶颈不同，memcache基本不存在读写上的瓶颈，所以可以做成多队列的。这是我个人的一点理解。

4.1.3. 启动流程

好，言归正传。我们的TT这样设计是没有问题的。上面已经论述。那么他具体的实现是如何的呢，下面我先做一个简单的介绍，以后再对各部分进行深入分析。 整个运行流程的开始是在TT/ttutil.c文件的ttservstart函数开始的，他首先创建了主socket用于接受请求。接下来初始化所有线程并进入ttservdeqtasks函数入口运行。其实这时候已经开始不停地在队列的尾部处理请求了，不过现在没有请求，所以就是在空转。

然后构造epoll的描述符。 下一步是将上面创建的主socket放到epoll的侦听列表中，开始进入大循环进行整个接收请求工作。

4.1.4. 请求处理

当一个请求来到，epoll通知进行通知，如果是对主socket的连接请求，那么主线程会创

建一个新的socket然后将它一起放到epoll的侦听列表中。 如果这个请求是对派生socket的请求，那么这个派生的socket会被包装起来，放进我们上面那个队列中，这个过程是通过tclistpush完成的。 其实这已经是一个完整的流程了，子线程的处理是同步的在创建线程的时候进行的。现在我们回头来说一下。 当我们的队列有了数据，也就是有了具体的请求之后。我们的从线程们就可以读过来处理，当然，一个线程一次拿走一个进行处理。这个过程在代码中是怎么样的呢？ 我们知道从线程的入口是ttservdeqtasks。在这个函数中，也有一个大的循环，这个循环什么作用？当然就是不停地去队列尾上问，有没有任务。没有就再进行循环，有的话就调用tclistshift2函数将这个任务取出来。然后进行处理。

4.1.5. 数据结构

大体流程就是上面这样的，重要的数据结构有：

- TTREQ:其实就是代表每一个线程，多了一些附加数据。
- TTSERV:我们这个网络服务器就是一个他的实例，包含了一些参数的设置，比如host,port之流。
- TTSOCK:在具体的线程处理请求中，针对的是一个socket，这个socket就是这个类型。

  +++程序流程+++

  下面再说一下整个程序流程。

1. 先从ttserver.c的main函数开始，接收命令行参数做好了配置就进入同一个文件中

的proc函数。进行了一些初始化的工作。

2. 然后进入ttutil.c文件的ttservstart开始真正的启动过程，在这个函数中创建socket，启动线程，epoll侦听等等，如上面说的。

程序就是这么走过来的。大多东西也都是在上面说的两个文件里，有时候你会发现一些函数是tc开头的好像找不到，其实他是在TC的源代码中，还有一些数据结构也是。比如我们上面那个队列元素就是。 @startquote 其它收获： 想说一下自己看代码的感想，前段时间我一直在做TT的性能测试工作，测试我很不爽，因为不知道具体的流程，所以很多结果出了也不知道对不对，很多问题出了也不清楚是为什么。终于开始看代码，很爽。 首先是整个代码非常有条理。和我前一两周看的wordpress代码真是风格完全不同，当然读完代码的享受也不同。 第二是我总结了我看代码的一个比较好的方法，就是先看大流程，再看整体架构和数据结构。其它细节就很简单了。 大流程就是跟着main去看，如果有资料当然更好。整体架构就是从主要数据结构出发看过去，其实你会发现无非就是一些文件操作，网络编程。就是把数据结构倒过去倒过来。其实如果一开始你能知道每个数据结构是做什么的，那程序其实也不用看了。所以说主体数据结构很重要，有的人不喜欢看.h文件，我觉得是很不对的。 @endquote

5. 数据库存储基础

这里的数据库不是关系型数据库,是key-value数据库.tokyotyrant是在tokyocabinet的基础上开发的.进入tokyocabinet的源码目录,大致结构如下:

5.1. tokyocabinet的源代码结构

```
~/download/tokyocabinet-1.4.17@aragorn   $ ls -ahl *.c |awk '{print $8}'
md5.c
myconf.c
tcadb.c
tcamgr.c
tcatest.c
tcawmgr.c
tcbdb.c
tcbmgr.c
tcbmttest.c
tcbtest.c
tcfdb.c
tcfmgr.c
tcfmttest.c
tcftest.c
tchdb.c
tchmgr.c
tchmttest.c
tchtest.c
tctdb.c
tctmgr.c
tctmttest.c
tcttest.c
```

```
tcucodec.c
tcumttest.c
tcutest.c
tcutil.c
```

其中,tca*.c是抽象出来的数据库的代码，这个的库可以是hashtable,fix-length array database,b+ tree database 中的任何一种。tcb*.c是b+ tree数据库的相关代码,tcf*.c是fix-length array 数据库的相关代码,tch*.c是hashtable型数据库相关代码,tct*.c是table database相关的代码。理解了这些数据存储类型的区别,才能理解后来tokyotyrant的一些参数设置.在下一节就专门讲述这些; tcu*.c是工具类的杂项函数集。 基本上所有的数据库类型都有*test和*mttest,*mgr三个二进制程序。比如fix-length-array database有tcftest.c,tcftest.c,tcfmttest.c都有是main函数的。

5.2. tokyotyrant的存储类型

- If the name is “*”, the database will be an on-memory hash database. //类似于MC的内存HASH存储
- If it is “+”, the database will be an on-memory tree database. //内存中的B+树存储
- If its suffix is “.tch”, the database will be a hash database. //硬盘上的HASH存储
- If its suffix is “.tcb”, the database will be a B+ tree database. //硬盘上的B+树存储
- If its suffix is “.tcf”, the database will be a fixed-length database. //像数组一样的连续数字对应定长值的存储
- If its suffix is “.tct”, the database will be a table database. //一个key值下面对应很多个name->value的形式

- 这些资料是从tokyotyrant的文档得到的,跟tokyocabinet中的基本相对应;
- table型(tct)是后来加入的功能,早先的版本并无这个类型.

5.3. tokyotyrant的缓存

请大家参照"安装和使用"部分(来源:张宴)，基本也是中文世界的第一篇研究文章，里面讲到用tch表时，如果加上#bnum=***，就可以缓存多少数据在内存中。也就是说TC是一个可以缓存磁盘数据的存储系统，这个缓存可以配置，配置不仅是bnum这一个参数决定，还有rcnum， xmsiz这两个参数，具体意思请看官方和源代码注释。

5.4. 异步

TC支持异步的写入机制，就是说写入的数据可以不一定刷到磁盘上，可以等到一定条件后再进行同步操作。这个异步的好处当然就是快，坏处当然就是可能在突发情况下丢失一些数据。TC的大部分写入方法都是同步的，而且会在操作过程中进行异步数据的同步磁盘写入。

5.5. 索引

HASH方式存储的数据是如何索引的？这个想必看过MC和PHP 的HASH数据实现的人都应该设计得出来，那就是一个大的HASH表，一个key来了，做一下HASH，通过HASH值到大的HASH表中找到具体数据的位置，相同的HASH值再串成一个链，跟着这个链再对比key值就能找到你的数据了。TC的设计大概也是这样，不同的是他的这个位置不是一个指针，也是一个offset值，代表他相对于数据文件首位置的偏移量。然后具体在

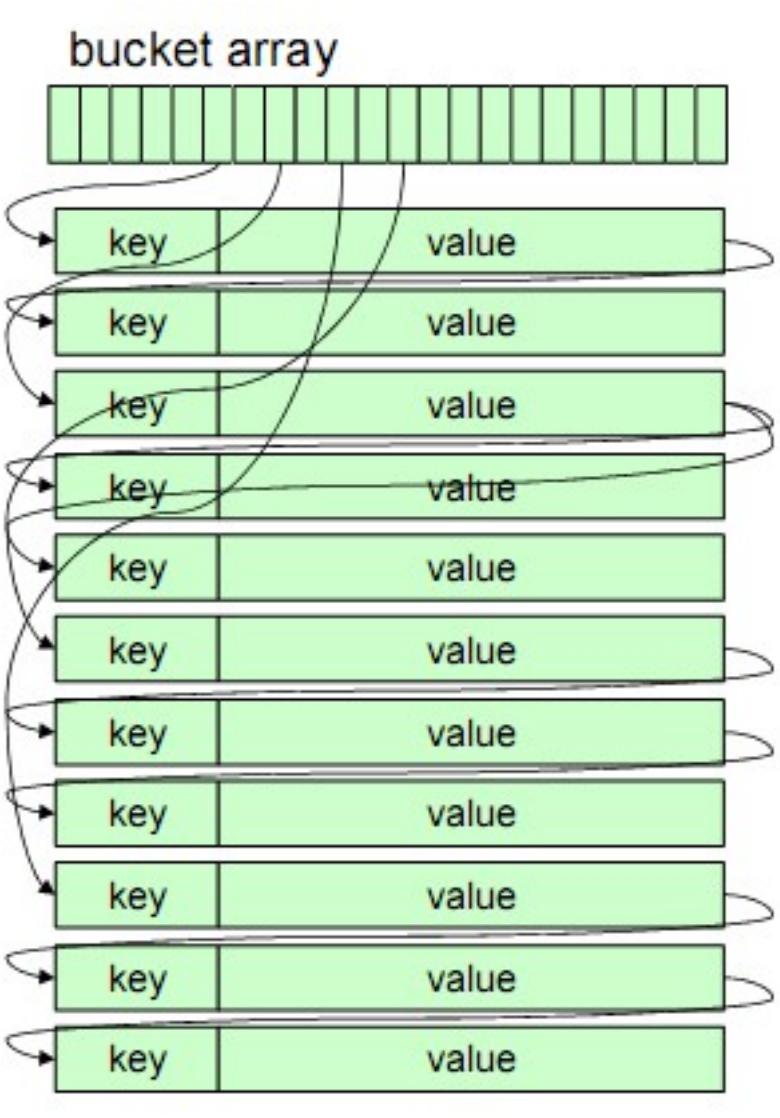
写入读取的时候，通过这个偏移量去调用pwrite和pread两个系统调用去做写入读取工作。

5.6. 数据的Hash

tokyocabinet的硬盘hash存储方式，又分为在数据文件中的冷存储和在内存中的热存储，冷存储用于保存所有key-value的对应值，热存储是对经常get的数据的一个缓存，达到对频繁使用的数据的快速读取的目的。

5.6.1. 冷存储

这是从讲tokyo系列产品的官方ppt上的截图:



5.6.1.1. 一级hash索引：bidx

冷存储的第一级索引是一个hash表，叫做bucket table，也就是上图第一行表示的，运用过程是通过key值用一个hash算法算出一个bidx值，然后在这个表里查这个bidx对应的key-value值存在哪里，再在文件中查找。这个bucket table的大小，就是由我们跟在数

据文件名后面的设置参数bnum设置的。

当然，一开始的时候这个bucket array是空的。比如当第一个数据来了的时候，他请求存储，比如这时计算到他的index值是1000，那么我们会找一下1000对应的位置，发现是NULL，那么我们就将这个值存在目前数据存储区的第一个可用位置。然后将这个位置记录到bucket array里。过程确实很简单。

5.6.1.2. 二级hash索引：hash

但是我们都知道，hash算法是会冲突的，当不同的key值算到了同样的hash，那我们仅用上面的一个bucket array是不能区分的，所以这里又加了第二层hash，其实加几层都是一样，冲突不可避免，最后还是要通过key的比较来判断，所以这个过程一共是三步。

第一个取到index，得到初始的存储位置。可能这个位置的这条记录的第二个hash值hash2并不与目标的hash2相等，也就是第一个hash冲突了。

因为第一个hash相等的所有记录都是互相用数据单元的left，right指针（其实就是一个offset值）连接起来的。而且按第二个hash值的大小按序排列的，所以当我们要存储的数据的第一个hash值index与该bidx指向的第一个数据单元的第二hash值hash2不同的时候，我们就按照大小分别在不同的方向上找到自己的位置然后将数据插入。

5.6.1.3. key值对比

在存储数据时，当已经存在这个key值时，我们可能会选择覆盖，可能会选择保留原值

，可能会将新值连接在老数据后面，到底进行哪一个操作，取决于我们调用的是哪个方法。因为所有的存储，包括replace，set等，都是调用一个叫tchdbputimpl的函数来实现数据存储的。

（数据缓存部分待后续）

5.6.1.4. 存储时的主要逻辑

1. 通过第一个hash值bidx在bucket array中找到对应的bidx的offset值。
2. 通过offset值取到一个数据单元。
3. 考查这个数据单元的hash值，是否与待处理数据一样。

    1. 如果一样，那再考查key是否一样。
        1. 如果key也一样，那么采用覆盖，替换，连接等几种方法来处理， 具体采用哪个方法取决于调用这个函数的一方为函数提供了怎样参数（具体在最后一个参数）。
    2. 如果不同，那对比这个数据元的hash值与待处理的数据的hash值的大小，如果大于待处理的值，就再对比这个数据单元右边的一个数据，如果小于则向左。如此循环，直到找到合适的位置。 +如果这个位置已经有数据，比较这个数据的key与这个位置的数据元的数据的key的大小。
        1. 如果一样大，那采用上面的方法，覆盖，替换，连接等几种方法来处理。
        2. 如果不一样大，那再通过大小的对比采用向左或向右的方法来寻找，直到找到相同的key，或者根本就没有相同的key已经存储了，那就在

中间插入一项。

不知道说清楚没有，其实就是一般的双向有序链表的查找，不同的是他在第一个标识的基础上又有第二第三个。第一个是bidx，第二个是hash，第三个是key。这里hash的算法我没有具体看，但是应该是与key按同样顺序增长的，也就是hash大的，key比较时也大，不然可以想一下，第一次如果找到一个单元，因为hash的大小比较而向左，向左后又因为key而让向右。这个时候可能就会死循环了。

下面再说一下数据文件的结构，及一部分的缓存数据。

5.6.1.5. 数据文件结构

数据文件的结构，首先是一个“ToKyO CaBiNeT”字串，他占一行，只是起一个前导标识的作用。 然后是bucket array的存储，就是一个长度为“bnum”（配置参数）的连续存储空间，和内存中的形式一样，后面是我现在还不是很明白的free block pool相关的一些数据，然后是填充空间的一些空白数据。

5.6.1.6. 内存映射的一级索引

其中在第一行的“ToKyO CaBiNeT”字串之后，开始进行的内存映射，映射的大小由配置参数xmsiz，与bucket array的大小，与上面说的文件的大小三者有关系，总的结果是会至少包含整个bucket array表。也就是说这个一级索引是存在内存中的。这里如果你配置了过小的xmsiz，也不会有影响。因为他第一步是取xmsiz和bucket array空间之中的大的那个。

5.6.1.7. bnum参数

总的说来，这里需要注意的是bnum参数。他决定了我们的bucket array的大小，基本上也决定了你数据的冲突情况，平均查找情况，所以这个数字的设定相当重要，在这方面我没有什么经验，看到有推荐为需要存储的记录总数的0.4-4倍。

5.6.1.8. 预告

这只是缓存的一部分，是完全存储的一级索引，另一部分是数据的缓存，也就是对常用数据直接可像memcached一样从内存取数据，这个我下面再接着说。

6. 线程和事件

tokyotyrant 使用多线程+epoll的基础架构.因此只能在linux或其他支持epoll的系统上运行,不支持windows;由于基于epoll,可以支持高并发,同时由于是多线程,相对多进程来说更省内存; tokyotyrant 在工作时会创建一定数量的工作线程和定时器线程(也可以叫做第三线程),下面将依次介绍;

6.1. 工作线程

6.1.1. 什么是工作线程组

其实本来想取名字叫第二线程，因为这个作线程组就是由主线程在epoll侦听前创建的那N个线程，这个N是由命令行启动参数设置的。

这个线程是处理所有请求的，如果说主线程是个推销员，那这类线程就是车间里的工人了。

6.1.2. 从这里开始

线程在ttservstart里被创建，就是那个循环次数为thnum的循环，他创建了reqs这个数组，其实每个数组就是一个工作线程的包装。这个数组的数据类型是TTREQ，下面还是将这个数据结构列一下：

```
typedef struct _TTREQ {                  /* type of structure for a server
*/
  pthread_t thid;                        /* thread ID */
  bool alive;                            /* alive flag */
  struct _TTSERV *serv;                  /* server object */
  int epfd;                              /* polling file descriptor */
  double mtime;                          /* last modified time */
  bool keep;                             /* keep-alive flag */
  int idx;                               /* ordinal index */
} TTREQ;
```

如上所言，他主要就是工作线程的包装。

上面线程创建的时候，入口地址是ttservdeqtasks()，同样是在ttutil.c文件中实现。这个函数的作用就是读取工作队列中的工作单元，然后处理这个单元。

6.1.3. ttservdeqtasks的工作过程

我们还记得，在前面说过主线程接受请求，然后将他压入到队列中，那时用了一个方法名叫tclistpush()，而现在我们的工作线程的任务正好相反，是要从队列的另一头取出请求，这时用的方法是tclistshift2()。

6.1.3.1. **存取请求**

这里必须得说一个这个队列的结构，这个队列是作为serv数据结构的一个属性存在的，就是那个TCLIST类型的queue指针。

好，我们看一下TCLIST类型：

```
typedef struct {                        /* type of structure for an array
list */
  TCLISTDATUM *array;                   /* array of data */
  int anum;                             /* number of the elements of the a
rray */
  int start;                            /* start index of used elements */
  int num;                              /* number of used elements */
} TCLIST;
```

第一个叫array的元素确实指向一个数组，其实这个数组就是队列的容器，后面就是队列

这种数据结构的起始位置等参数。

我们上面说的tclistpush()和tclistshift2()两个函数就是对这个数据结构进行操作的。

而存入和取出的东西，主要就是一个fd，就是epoll接收到连接的时候用accept创建的新的fd。当时接收到联接就将这个fd放入队列，然后工作线程从队列另一头取出它。过程就是这样。取出队列处理，相当于从这个fd读取请求，再将请求的返回写到这个fd中。

好，我们看tclistshift2()函数之后，取到fd然后再将这个fd与此时处理这个fd的req传给ttservtask()进行处理。到这一步，就相当于是将一个线程与一个具体的请求挂在一起了。

6.1.3.2. 线程处理请求

ttservtask()的实现也在当前文件中，他的实现极简单，就是调用了serv的do_task函数指针所指函数，此函数指针在前面已经赋值为do_task()函数了，这个函数位于ttserver.c中，和do_slave()是挨在一起的，这两个最核心的函数放在一起。

他主要是从fd中解析出请求内容，然后调用TC的接口处理，再返回。就是这样。

但是我们知道TT可是支持MC协议，telnet连接与HTTP的REST功能的，所以呢，这个do_task函数就分成几块来做的了。根据读出来的第一个字节不同，分别调用了不同的方法实现，而所有的方法无非都是调用TC的接口。具体看ttserver.c中的do_task函数中调用的函数的实现。

7 MemcacheDB Tokyo Tyrant和Redis

性能对比测试

7.1. 测试环境

+++硬件++ 2 Linux boxes in a LAN, 1 server and 1 test client Linux Centos 5.2 64bit Intel(R) Xeon(R) CPU E5410 @ 2.33GHz (L2 cache: 6M), Quad-Core * 2 8G memory SCSI disk (standalone disk, no other access)

7.1.1. 软件环境

db-4.7.25.tar.gz libevent-1.4.11-stable.tar.gz memcached-1.2.8.tar.gz memcachedb-1.2.1-beta.tar.gz redis-0.900_2.tar.gz tokyocabinet-1.4.9.tar.gz tokyotyrant-1.1.9.tar.gz

7.1.2. 配置

- Memcachedb 启动参数

```
Test 100 bytes
./memcachedb -H /data5/kvtest/bdb/data -d -p 11212 -m 2048 -N -L 8192
(Update: As mentioned by Steve, the 100-byte-test missed the -N paramter, so I added it and updated the data)
Test 20k bytes
./memcachedb -H /data5/kvtest/mcdb/data -d -p 11212 -b 21000 -N -m 2048
```

- Tokyo Tyrant (Tokyo Cabinet) configuration

```
Use default Tokyo Tyrant sbin/ttservctl
use .tch database, hashtable database


ulimsiz=”256m”
sid=1
dbname=”$basedir/casket.tch#bnum=50000000" # default 1M is not enough!
maxcon=”65536"
retval=0
```

- Redis configuration

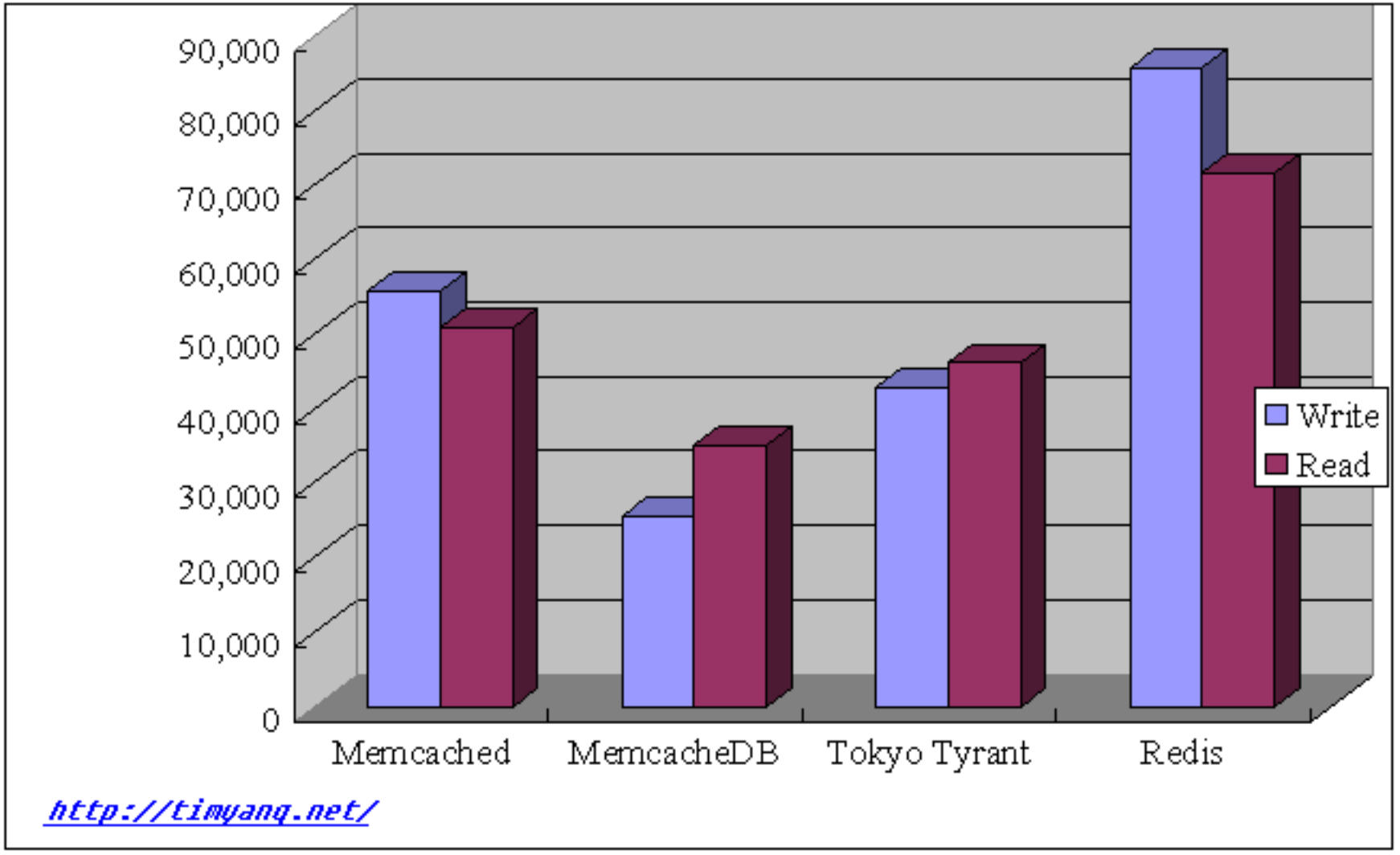
```
timeout 300
save 900 1
save 300 10
save 60 10000
# no maxmemory settings
```

7.1.3. 测试客户端

Client in Java, JDK1.6.0, 16 threads Use Memcached client java_memcached-release_2.0.1.jar JRedis client for Redis test, another JDBC-Redis has poor performance.

7.2. 小数据量测试结果

Test 1, 1-5,000,000 as key, 100 bytes string value, do set, then get test, all get test has re sult. Request per second(mean)



Store	Write	Read

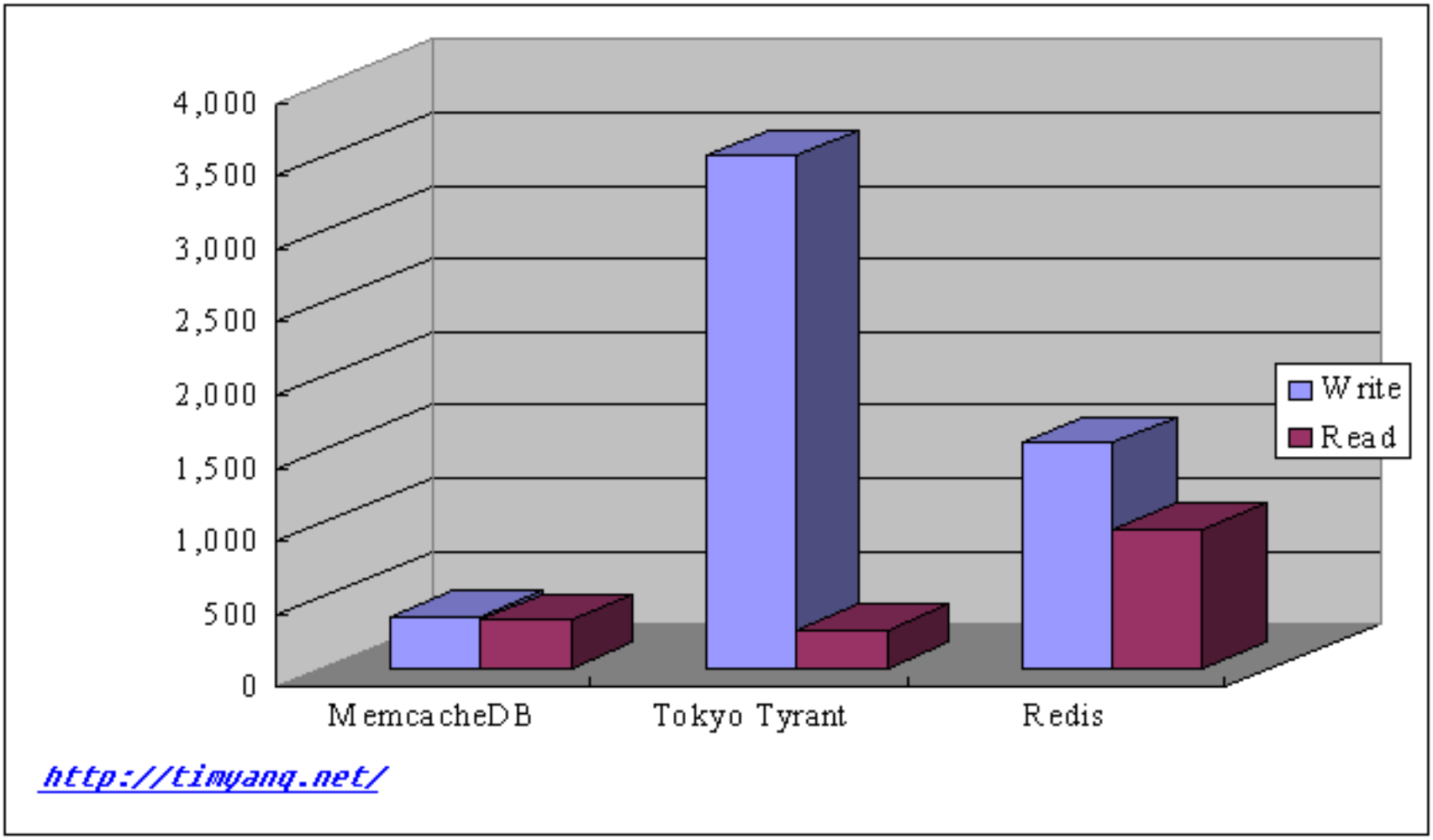
Memcached	55,989	50,974
memcachedb	25,583	35,260
Tokyo Tyrant	42,988	46,238
Redis	85,765	71,708

Server Load Average

Store	Write	Read
memcachedb	1.80, 1.53, 0.87	11.17, 1.16, 0.83
memcachedb	11.44, 0.93, 0.64	4.35, 1.94, 1.05
Tokyo Tyrant	3.70, 1.71, 1.14	2.98, 1.81, 1.26
Redis	1.06, 0.32, 0.181	1.56, 1.00, 0.54

7.3. 大数据量测试结果

Test 2, 1-500,000 as key, 20k bytes string value, do set, then get test, all get test has result. Request per second(mean) (Aug 13 Update: fixed a bug on get() that read non-exist key)



Store	Write	Read
memcachedb	357	327
Tokyo Tyrant	3,501	257
Redis	1,542	957

7.4. Some notes about the test

When test Redis server, the memory goes up steadily, consumed all 8G and then use swap(and write speed slow down), after all memory and swap space is used, the client will get exceptions. So use Redis in a productive environment should limit to a small data size. It is another cache solution rather than a persistent storage. So compare Redis together with MemcacheDB/TC may not fair because Redis actually does not save data to disk during the test.

Tokyo cabinet and memcachedb are very stable during heavy load, use very little memory in set test and less than physical memory in get test.

MemcacheDB peformance is poor for write large data size(20k).

The call response time was not monitored in this test.

8. Tokyo Tyrant 的问题和Bug

8.1. Bug report

请将Bug 汇报给 hirarin#gmail.com (您应当知道如何发邮件的);

8.2. tokyotyrant大规模出错的问题

1. 错误描述 在进行大量数据缺口插入时，eg.连接插入100W条1000字长的数据。key为0-999999的简单字串。在中途出现插入失败，并且后续插入全部失败的情况，比如在40多万之前插入成功，后面全部失败。

2. 原因 MC的php客户端设置有超时机制，如果服务器端处理时间过长，则从客户端主动关闭此TCP连接，当然连接关闭了，后面的操作也就全部失败了。

3. 解决方法 设置connect函数的第三个参数timeout为一个比较大的数字。

8.3. Bugs

我猜这是一个小小的bug:

```
file ttserver.c ,line 524:
     if(mhost){
      ttservlog(g_serv, TTLOGINFO,
                "warning: replication is omitted because the SID is not
specified");
      mhost = NULL;
    }
```

我认为应该是:

```
     if(mhost){
      ttservlog(g_serv, TTLOGINFO,
                "warning: replication is omitted because the mhost is
not specified");
```

```
    mhost = NULL;
  }
```

9. 延伸阅读:key-value-pair database的比较

9.1. 满足极高读写性能需求的Kye-Value数据库：Redis，Tokyo Cabinet， Flare

9.1.1. Redis

Redis是一个很新的项目，刚刚发布了1.0版本。Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过10万次读写操作，是我知道的性能最快的Key-Value DB。

Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存List链表和Set集合的数据结构，而且还支持对List进行各种操作，例如从List两端push和pop数据，取List区间，排序等等，对Set支持各种集合的并集交集操作，此外单个value的最大限制是1GB，不像memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一个功能加强版的memcached来用。

Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，并且它没有原生的可扩展机制，不具有scale（可扩展）能力，要依赖客户端来实现分布式读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。目前使用Redis的网站有github，Engine Yard。

9.1.2. Tokyo Cabinet和Tokoy Tyrant

TC和TT的开发者是日本人Mikio Hirabayashi，主要被用在日本最大的SNS网站mixi.jp上，TC发展的时间最早，现在已经是一个非常成熟的项目，也是Kye-Value数据库领域最大的热点，现在被广泛的应用在很多很多网站上。TC是一个高性能的存储引擎，而TT提供了多线程高并发服务器，性能也非常出色，每秒可以处理4-5万次读写操作。

TC除了支持Key-Value存储之外，还支持保存Hashtable数据类型，因此很像一个简单的数据库表，并且还支持基于column的条件查询，分页查询和排序功能，基本上相当于支持单表的基础查询功能了，所以可以简单的替代关系数据库的很多操作，这也是TC受到大家欢迎的主要原因之一，有一个Ruby的项目miyazakiresistance将TT的hashtable的操作封装成和ActiveRecord一样的操作，用起来非常爽。

TC/TT在mixi的实际应用当中，存储了2000万条以上的数据，同时支撑了上万个并发连接，是一个久经考验的项目。TC在保证了极高的并发读写性能的同时，具有可靠的数据持久化机制，同时还支持类似关系数据库表结构的hashtable以及简单的条件，分页和排序操作，是一个很棒的NoSQL数据库。

TC的主要缺点是在数据量达到上亿级别以后，并发写数据性能会大幅度下降，NoSQL: I f Only It Was That Easy提到，他们发现在TC里面插入1.6亿条2-20KB数据的时候，写入

性能开始急剧下降。看来是当数据量上亿条的时候，TC性能开始大幅度下降，从TC作者自己提供的mixi数据来看，至少上千万条数据量的时候还没有遇到这么明显的写入性能瓶颈。

9.1.3. Flare

TC是日本第一大SNS网站mixi开发的，而Flare是日本第二大SNS网站green.jp开发的，有意思吧。Flare简单的说就是给TC添加了scale功能。他替换掉了TT部分，自己另外给TC写了网络服务器，Flare的主要特点就是支持scale能力，他在网络服务端之前添加了一个node server，来管理后端的多个服务器节点，因此可以动态添加数据库服务节点，删除服务器节点，也支持failover。如果你的使用场景必须要让TC可以scale，那么可以考虑flare。

flare唯一的缺点就是他只支持memcached协议，因此当你使用flare的时候，就不能使用TC的table数据结构了，只能使用TC的key-value数据结构存储。

9.2. 满足海量存储需求和访问的面向文档的数据库：MongoDB，CouchDB

面向文档的非关系数据库主要解决的问题不是高性能的并发读写，而是保证海量数据存储的同时，具有良好的查询性能。MongoDB是用C++开发的，而CouchDB则是Erlang开发的.

9.2.1. MongoDB

MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似json的bjson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

Mongo主要解决的是海量数据的访问效率问题，根据官方的文档，当数据量达到50GB以上的时候，Mongo的数据库访问速度是MySQL的10倍以上。Mongo的并发读写效率不是特别出色，根据官方提供的性能测试表明，大约每秒可以处理0.5万－1.5次读写请求。对于Mongo的并发读写性能，我（robbin）也打算有空的时候好好测试一下。

因为Mongo主要是支持海量数据存储的，所以Mongo还自带了一个出色的分布式文件系统GridFS，可以支持海量的数据存储，但我也看到有些评论认为GridFS性能不佳，这一点还是有待亲自做点测试来验证了。

最后由于Mongo可以支持复杂的数据结构，而且带有强大的数据查询功能，因此非常受到欢迎，很多项目都考虑用MongoDB来替代MySQL来实现不是特别复杂的Web应用，比方说why we migrated from MySQL to MongoDB就是一个真实的从MySQL迁移到MongoDB的案例，由于数据量实在太大，所以迁移到了Mongo上面，数据查询的速度得到了非常显著的提升。

MongoDB也有一个ruby的项目MongoMapper，是模仿Merb的DataMapper编写的MongoDB的接口，使用起来非常简单，几乎和DataMapper一模一样，功能非常强大易用。

9.2.2. CouchDB

CouchDB现在是一个非常有名气的项目，似乎不用多介绍了。但是我却对CouchDB没有什么兴趣，主要是因为CouchDB仅仅提供了基于HTTP REST的接口，因此CouchDB单纯从并发读写性能来说，是非常糟糕的，这让我立刻抛弃了对CouchDB的兴趣。

这里一米六二插一句话:本人觉得,CounchDB这个玩意儿,光顾着加时尚元素，忘记自己是干什么的了;json又怎么样,erlang写的又怎么样?建议玩玩,千万别真用它....

9.3. 满足高可扩展性和可用性的面向分布式计算的数据库：Cassandra，Voldemort

面向scale能力的数据库其实主要解决的问题领域和上述两类数据库还不太一样，它首先必须是一个分布式的数据库系统，由分布在不同节点上面的数据库共同构成一个数据库服务系统，并且根据这种分布式架构来提供online的，具有弹性的可扩展能力，例如可以不停机的添加更多数据节点，删除数据节点等等。因此像Cassandra常常被看成是一个开源版本的Google BigTable的替代品。Cassandra和Voldemort都是用Java开发的：

9.3.1. Cassandra

Cassandra项目是Facebook在2008年开源出来的，随后Facebook自己使用Cassandra的另外一个不开源的分支，而开源出来的Cassandra主要被Amazon的Dynamite团队来维护，并且Cassandra被认为是Dynamite2.0版本。目前除了Facebook之外，twitter和digg.com都在使用Cassandra。

Cassandra的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个

分布式网络服务，对Cassandra的一个写操作，会被复制到其他节点上去，对Cassandra的读操作，也会被路由到某个节点上面去读取。对于一个Cassandra群集来说，扩展性能是比较简单的事情，只管在群集里面添加节点就可以了。我看到有文章说Facebook的Cassandra群集有超过100台服务器构成的数据库群集。

Cassandra也支持比较丰富的数据结构和功能强大的查询语言，和MongoDB比较类似，查询功能比MongoDB稍弱一些，twitter的平台架构部门领导Evan Weaver写了一篇文章介绍Cassandra：http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/，有非常详细的介绍。

Cassandra以单个节点来衡量，其节点的并发读写性能不是特别好，有文章说评测下来Cassandra每秒大约不到1万次读写请求，我也看到一些对这个问题进行质疑的评论，但是评价Cassandra单个节点的性能是没有意义的，真实的分布式数据库访问系统必然是n多个节点构成的系统，其并发性能取决于整个系统的节点数量，路由效率，而不仅仅是单节点的并发负载能力。

9.3.2. Voldemort

Voldemort是个和Cassandra类似的面向解决scale问题的分布式数据库系统，Cassandra来自于Facebook这个SNS网站，而Voldemort则来自于Linkedin这个SNS网站。说起来SNS网站为我们贡献了n多的NoSQL数据库，例如Cassandar，Voldemort，Tokyo Cabinet，Flare等等。Voldemort的资料不是很多，因此我没有特别仔细去钻研，Voldemort官方给出Voldemort的并发读写性能也很不错，每秒超过了1.5万次读写。