# 源码目录介绍

## 概念

  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。

## 目录介绍

  这里我使用的是redis-2.2.2.tar.gz版本的redis首先要对它进行安装，这里我选择使用cygwin工具进行安装，加入我把该压缩包放在F盘下，使用cygwin工具进行shell命令：

|  |  |
| --- | --- |
| 1  2  3 | $ tar xzf redis-2.2.2.tar.gz  $ cd redis-2.2.2  $ make |

  这里的make实际上操作的是Makefile文件，Makefile按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个shell脚本一样，其中也可以执行操作系统的命令。

浏览下Redis根目录中的Makefile文件：

# Top level makefile, the real shit is at src/Makefile

TARGETS=32bit noopt test

all:

    cd src && $(MAKE) $@

install: dummy

    cd src && $(MAKE) $@

clean:

    cd src && $(MAKE) $@

    cd deps/hiredis && $(MAKE) $@

    cd deps/linenoise && $(MAKE) $@

$(TARGETS):

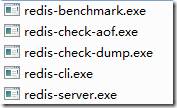
    cd src && $(MAKE) $@

src/help.h:

    @./utils/generate-command-help.rb > $@

dummy:

  通过make命令可以执行“cd src && make all”。而此时的make all实际上已经开始执行src目录中的Makefile文件。这个文件比较复杂，大致就是将一系列的c文件以及h文件链接起来，通过cc/gcc编译器将文件生成目标文件o，接着将相应的o目标文件在通过编译器生成exe文件，当你编译完毕后，在src的目录上将产生5个exe文件：

[](http://images.cnblogs.com/cnblogs_com/liping13599168/201104/20110412000057697.png)

redis-benchmark.exe：用于做性能测试；

redis-check-aof.exe：更新日志检查；

redis-check-dump.exe：用于本地数据库检查；

redis-cli.exe：客户端程序；

redis-server.exe：服务端程序；

具体用法这里不多说了，可以参考（[http://www.cnblogs.com/daizhj/articles/1956681.html](http://www.cnblogs.com/daizhj/articles/1956681.html" \o "http://www.cnblogs.com/daizhj/articles/1956681.html)）

  现在来看下src包含的文件（按照首字母顺序来讲）：

**adlist.h/adlist.c**：用于对list的定义，它是个双向链表结构，从头文件可以找到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // list节点  typedef struct listNode {      struct listNode \*prev;      struct listNode \*next;      void \*value;  } listNode;    // list迭代器  typedef struct listIter {      listNode \*next;  **int** direction;  } listIter;    // list数据结构  typedef struct list {      listNode \*head;      listNode \*tail;      void \*(\*dup)(void \*ptr);      void (\***free**)(void \*ptr);  **int** (\*match)(void \*ptr, void \*key);      unsigned **int** len;  } list; |

在ListNode节点下包含prev指针和next指针，说明它通过指针将节点进行双向链接。并且从adlist.h的头文件可以找到非常丰富的方法声明，包括list创建，list释放，list头部/尾部添加节点等等，具体在后面的系列会做出介绍。

**ae.h/ae.c**：用于Redis的事件处理，包括句柄事件和超时事件。

在ae.c中的头部可以发现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #ifdef HAVE\_EPOLL  #include "ae\_epoll.c"  #else      #ifdef HAVE\_KQUEUE      #include "ae\_kqueue.c"      #else      #include "ae\_select.c"      #endif  #endif |

  在网络相关操作中，定义了一组公共操作接口：aeApiCreate，aeApiFree，aeApiAddEvent，aeApiDelEvent，aeApiPoll，aeApiName方法。在**ae\_epoll.c、ae\_kqueue.c和ae\_select.c**中，分别实现了基于epoll/kqueue和select系统调用的接口。系统调用的选择顺序依次为epoll，kqueue，select。

**anet.h/anet.c**：这两个文件非常重要，作为Server/Client通信的基础封装，包括anetTcpServer,anetTcpConnect,anetTcpAccept,anetRead,anetWrite等等方法。

**aof.c**：aof，全称为append only file，作用就是记录每次的写操作,在遇到断电等问题时可以用它来恢复数据库状态。但是他不是bin的,而是text的。一行一行,写得很规范.如果你是一台redis,那你也能人肉通过它恢复数据。

**config.h/config.c**：用于将配置文件redis.conf文件中的配置读取出来的属性通过程序放到server对象中。在main函数（server服务主入口点处）可以发现里面调用loadServerConfig(char \*filename)方法，这个方法就是使用config.c里面的方法实现。具体会在后面的系列中详细介绍。

**db.c**：对于Redis内存数据库的相关操作。

**debug.c**：用于调试使用。

**dict.h/dict.c**：也是很重要的两个文件，主要对于内存中的hash进行管理：

typedef struct dictEntry {

    void \*key;

    void \*val;

    struct dictEntry \*next;

} dictEntry;

typedef struct dictType {

    unsigned int (\*hashFunction)(const void \*key);

    void \*(\*keyDup)(void \*privdata, const void \*key);

    void \*(\*valDup)(void \*privdata, const void \*obj);

    int (\*keyCompare)(void \*privdata, const void \*key1, const void \*key2);

    void (\*keyDestructor)(void \*privdata, void \*key);

    void (\*valDestructor)(void \*privdata, void \*obj);

} dictType;

typedef struct dict {

    dictType \*type;

    void \*privdata;

    dictht ht[2];

    int rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/

    int iterators; /\* number of iterators currently running \*/

} dict;

这里dictEntry作为一个dict字段结构，里面包括key以及value，已经指向下一个dictEntry的指针。dictType作为一些dict的操作结构。dict作为一个hash结构。后面的文章会具体介绍。

**fmacros.h**：用于Mac下的兼容性处理。

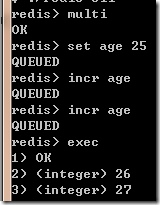
**help.h**：辅助于命令的提示信息，作用于redis-cli.exe可执行文件中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | struct commandHelp {  **char** \*name;  **char** \*params;  **char** \*summary;  **int** group;  **char** \*since;  } commandHelp[] = {      { "APPEND",      "key value",      "Append a value to a key",      1,      "1.3.3" },      { "AUTH",      "password",      "Authenticate to the server",      8,      "0.08" },      { "BGREWRITEAOF",      "-",      "Asynchronously rewrite the append-only file",      9,      ....  }; |
|  |  |

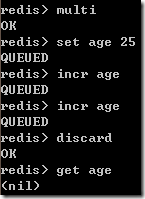
**intset.h/intset.c**：整数范围内的使用set，并包含相关set操作。

**lzf.h/lzf\_c.c/lzf\_d.c/lzfP.h**：对于本地数据库的保存，使用的是LZF压缩算法，很神奇，算法只有200-300行的代码。

**multi.c**：用于事务处理操作。请看这样的一个例子：

[](http://images.cnblogs.com/cnblogs_com/liping13599168/201104/201104120201525473.png)

通过执行exec，可以提交整个事务过程，如果你想撤销整个事务过程，你可以使用discard命令：

[](http://images.cnblogs.com/cnblogs_com/liping13599168/201104/201104120201534111.png)

可以发现get age已经取不到值了，说明discard命令让事务失效。

**networking.c**：网络协议传输方法定义相关的都放在这个文件里面了。包括让Client连接上Server，让Slave挂接到Master，已经Server/Client之间的信息交互的实现等等。

**object.c**：用于创建和释放redisObject对象，redisObject结构为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | typedef struct redisObject {      unsigned type:4;      unsigned storage:2;     /\* REDIS\_VM\_MEMORY or REDIS\_VM\_SWAPPING \*/      unsigned encoding:4;      unsigned lru:22;        /\* lru time (relative to server.lruclock) \*/  **int** refcount;      void \*ptr;      /\* VM fields are only allocated if VM is active, otherwise the       \* object allocation function will just allocate       \* sizeof(redisObjct) minus sizeof(redisObjectVM), so using       \* Redis without VM active will not have any overhead. \*/  } robj; |

**pqsort.h/pqsort.c/sort.c**：关于排序算法，sort.c具体作为Redis场景下的排序实现。

**pubsub.c**：用于订阅模式的实现，有点类似于Client广播发送的方式。

**rdb.c**：对于Redis本地数据库的相关操作，默认文件是dump.rdb（通过配置文件获得），包括的操作包括保存，移除，查询等等。

**redis-benchmark.c**：用于redis性能测试的实现。请看main方法以下设置：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | config.debug = 0;  config.numclients = 50;  config.requests = 10000;  config.liveclients = 0;  config.el = aeCreateEventLoop();  aeCreateTimeEvent(config.el,1,showThroughput,NULL,NULL);  config.keepalive = 1;  config.donerequests = 0;  config.datasize = 3;  config.randomkeys = 0;  config.randomkeys\_keyspacelen = 0;  config.quiet = 0;  config.loop = 0;  config.idlemode = 0;  config.latency = NULL;  config.clients = listCreate();  config.hostip = "127.0.0.1";  config.hostport = 6379;  config.hostsocket = NULL;    parseOptions(argc,argv);  config.latency = zmalloc(sizeof(**long** **long**)\*config.requests); |

默认性能测试中的客户端数量为50个，并行发送的请求有10000条，也可以通过redis-benchmark命令行参数进行设置。

**redis-check-aof.c**：用于更新日志检查的实现。

**redis-check-dump.c**：用于本地数据库检查的实现。

**redis-cli.c**：客户端程序的实现。具体会在后面的文章详细介绍。

**redis.h/redis.c**：服务端程序的实现。具体会在后面的文章详细介绍。

**release.h/release.c**：用于发布使用。

**replication.c**：用于主从数据库的复制操作的实现。

**sds.h/sds.c**：用于对字符串的定义，从头文件可以找到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //字符串  struct sdshdr {  **int** len;  **int** **free**;  **char** buf[];  }; |

还可以看到对于字符串的相关操作，包括复制，连接，清零等等。

**sha1.h/sha1.c**：有关于sha算法的实现。

**solarisfixes.h**：Solaris系统的兼容性实现。

**syncio.c**：用于同步Socket和文件I/O操作。

**t\_hash.c/t\_list.c/t\_set.c/t\_string.c/t\_zset.c**：hash，list，set，string，zset在Server/Client中的应答操作。主要通过redisObject进行类型转换。

**testhelp.h**：一个C风格的小型测试框架。

**util.c**：关于通用工具的方法实现。

**version.h**：Redis版本号定义。

**vm.c**：关于虚拟内存的管理实现。

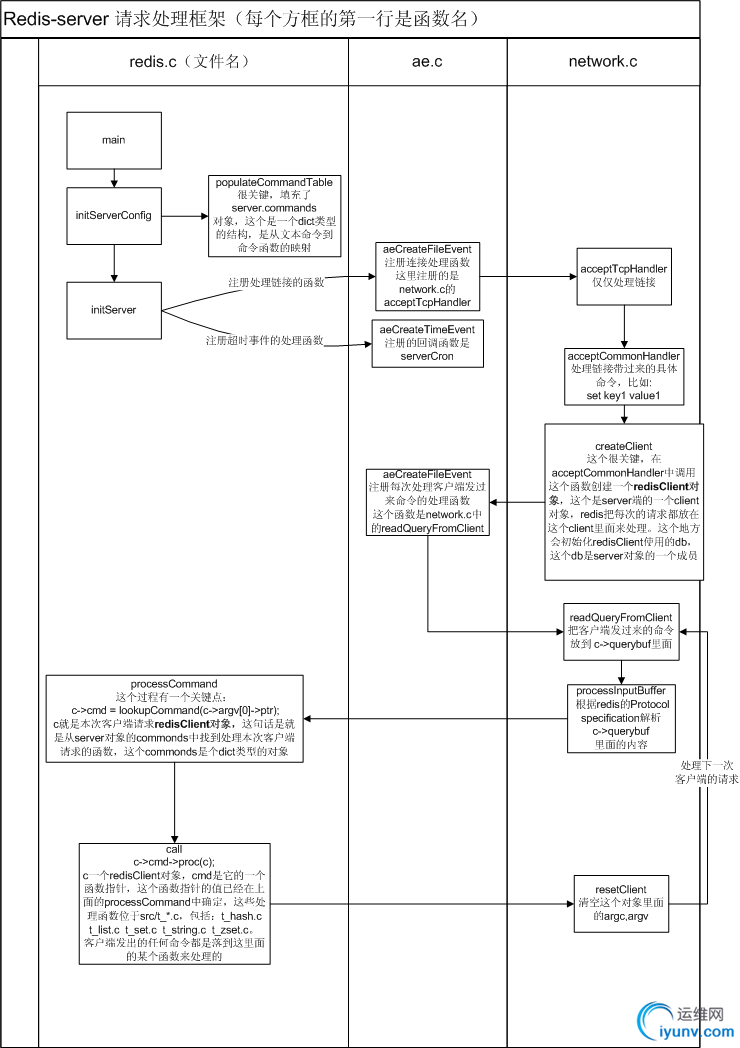
**zipmap.h/zipmap.c**：zipmap是一个类似于hash的存储对象。在新建一个hash对象时开始是用zipmap（又称为small hash）来存储的。这个zipmap其实并不是hash table但是zipmap相比正常的hash实现可以节省不少hash本身需要的一些元数据存储开销，如果field或者value的大小超出一定限制后，redis会在内部自动将zipmap替换成正常的hash实现。

**ziplist.h/ziplist.c**：ziplist是一个类似于list的存储对象。它的原理类似于zipmap。

**zmalloc.h/zmalloc.c**：关于Redis的内存分配的封装实现。

下一篇我会介绍下redis-server以及redis-cli的源码实现。

# 架构分析



## 事件机制

1. 使用ae的时候要初始化一个aeEventLoop，这个保存了一个进程要关注的所有事件相关信息

/\* State of an event based program \*/

typedef struct aeEventLoop {

int maxfd;

long long timeEventNextId;

aeFileEvent events[AE\_SETSIZE]; /\* Registered events \*/ AE\_SETSIZE是10\*1024 限制epoll能最多关注的事件最大个数

aeFiredEvent fired[AE\_SETSIZE]; /\* Fired events \*/

aeTimeEvent \*timeEventHead; 这个是注册的所有定时事件

int stop;

void \*apidata; /\* This is used for polling API specific data \*/

aeBeforeSleepProc \*beforesleep; //一个函数指针，指定每次aeMain每次轮询要做的事情

} aeEventLoop;

2. 这个aeEventLoop两个非常重要的数据结构，一个是aeFileEvent 还有一个是aeTimeEvent。

前者关注的个数用AE\_SETSIZE做了限制，是一个固定大小的数据，这个里面放了一个文件事件的所有相关信息；

后者关注的定时事件，这个的个数不好提前确定，这里的实现方式是通过一个Linked List，所以不支持随机访问。这几个相关的定义如下：

/\* File event structure \*/

typedef struct aeFileEvent {

int mask; /\* one of AE\_(READABLE|WRITABLE) \*/

aeFileProc \*rfileProc;

aeFileProc \*wfileProc;

void \*clientData;

} aeFileEvent;

/\* Time event structure \*/

typedef struct aeTimeEvent {

long long id; /\* time event identifier. \*/

long when\_sec; /\* seconds \*/

long when\_ms; /\* milliseconds \*/

aeTimeProc \*timeProc;

aeEventFinalizerProc \*finalizerProc;

void \*clientData;

struct aeTimeEvent \*next;

} aeTimeEvent;

/\* A fired event \*/

typedef struct aeFiredEvent {

int fd;

int mask;

} aeFiredEvent;

3. 然后使用aeCreateFileEvent和aeCreateTimeEvent，向aeEventLoop 注册事件，可以注册AE\_SETSIZE个与FD相关的事件和无数个定时事件

FD方式要指定监听的FD，定时方式要指定轮询间隔

int aeCreateFileEvent(aeEventLoop \*eventLoop, int fd, int mask, aeFileProc \*proc, void \*clientData)

long long aeCreateTimeEvent(aeEventLoop \*eventLoop, long long milliseconds, aeTimeProc \*proc, void \*clientData, aeEventFinalizerProc \*finalizerProc)

4. 最后是调用aeMain，aeMain会调用aeProcessEvents，这个aeProcessEvents就是处理事件的核心函数

5. 从几个重要的方面来说明aeProcessEvents处理事件的方式：

a. aeProcessEvents即会处理FD类型的事件，也会处理定时类型的时间。处理顺序是先处理FD类型的，然后处理定时类型的

b. 需要处理的FD类型事件是通过static int aeApiPoll(aeEventLoop \*eventLoop, struct timeval \*tvp)来获得的，这里的tvp的用意后面会提到。

需要注意的是这个aeApiPoll根据OS的不同有多个实现，在不同的文件中，具体使用哪个取决于下面的代码：

/\* Include the best multiplexing layer supported by this system.

\* The following should be ordered by performances, descending.

\* 出现在ae.c中，自动选择使用哪种事件机制 \*/

#ifdef HAVE\_EPOLL

#include "ae\_epoll.c"

#else

#ifdef HAVE\_KQUEUE

#include "ae\_kqueue.c"

#else

#include "ae\_select.c"

#endif

#endif

c. 由于，FD事件是提前于定时事件处理的，所以，这里有一种可能就是，在对FD类型事件做epoll\_wait的时候，有一些定时事件的期限已经到了，但是由于epoll\_wait是阻塞住的，所以没有办法得到及时处理。ae库采取的解决方式是：设置epoll\_wait的超时时间，但是这个时间设置成多少合适呢，函数aeSearchNearestTimer就是要解决这个问题的，它会便利计算出aeTimeEvent这个链表中所有定时事件的触发时间，然后返回下一个要触发事件的时间。

然后aeProcessEvents会计算当前时间举例这个时间的时间差，这个时间差就是epoll\_wait的超时时间，通过aeApiPoll会传给epoll\_wait。

如果这个时间差是零或者是负数，那么epoll\_wait将会立即返回；如果没有超时事件，超时时间就会设置成-1，epoll\_wait就会一直等

d. 再说一下processTimeEvents这个处理定时事件的函数。这个函数要一个一个的扫描链表里面的时间，发现一个到了时间就处理一个，否则，就把超时间增加1ms。

如果是成功处理完一个，则要重新扫描整个链表（因为在处理的同时，有可能前面的时间有到期了）；如果一个没有处理，则跳到下一个看条件是否满足

这个地方是个低效的方法，只要处理成功下次就要进行复杂度为O(n)的查找

protocol

　　\* CR LF

　　$ CR LF

　　 CR LF

　　...

　　$ CR LF

　　 CR LF

Bulk Reply

　　The format used for every argument $6\r\nmydata\r\n is called a Bulk Reply

　　\*3

　　$3

　　SET

　　$5

　　mykey

　　$7

　　myvalue

　　"\*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"

Replies

　　Redis will reply to commands with different kinds of replies. It is possible to check the kind of reply from the first byte sent by the server:

　　With a single line reply the first byte of the reply will be "+"

　　With an error message the first byte of the reply will be "-"

　　With an integer number the first byte of the reply will be ":"

　　With bulk reply the first byte of the reply will be "$"

　　With multi-bulk reply the first byte of the reply will be "\*"

Multi-bulk replies

　　C: LRANGE mylist 0 3

　　s: \*4

　　s: $3

　　s: foo

　　s: $3

　　s: bar

　　s: $5

　　s: Hello

　　s: $5

　　s: World

sds.h

　　struct sdshdr {

　　 int len;

　　 int free;

　　 char buf[];

　　};

　　adlist.h

　　typedef struct listNode {

　　 struct listNode \*prev;

　　 struct listNode \*next;

　　 void \*value;

　　} listNode;

　　typedef struct listIter {

　　 listNode \*next;

　　 int direction;

　　} listIter;

　　typedef struct list {

　　 listNode \*head;

　　 listNode \*tail;

　　 void \*(\*dup)(void \*ptr);

　　 void (\*free)(void \*ptr);

　　 int (\*match)(void \*ptr, void \*key);

　　 unsigned int len;

　　} list;

dict.h

　　typedef struct dictEntry {

　　 void \*key;

　　 void \*val;

　　 struct dictEntry \*next;

　　} dictEntry;

　　typedef struct dictType {

　　 unsigned int (\*hashFunction)(const void \*key);

　　 void \*(\*keyDup)(void \*privdata, const void \*key);

　　 void \*(\*valDup)(void \*privdata, const void \*obj);

　　 int (\*keyCompare)(void \*privdata, const void \*key1, const void \*key2);

　　 void (\*keyDestructor)(void \*privdata, void \*key);

　　 void (\*valDestructor)(void \*privdata, void \*obj);

　　} dictType;

　　/\* This is our hash table structure. Every dictionary has two of this as we

　　 \* implement incremental rehashing, for the old to the new table. \*/

　　typedef struct dictht {

　　 dictEntry \*\*table;

　　 unsigned long size;

　　 unsigned long sizemask;

　　 unsigned long used;

　　} dictht;

　　typedef struct dict {

　　 dictType \*type;

　　 void \*privdata;

　　 dictht ht[2];

　　 int rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/

　　 int iterators; /\* number of iterators currently running \*/

　　} dict;

redis.h

　　typedef struct redisObject {

　　 unsigned type:4;

　　 unsigned storage:2; /\* REDIS\_VM\_MEMORY or REDIS\_VM\_SWAPPING \*/

　　 unsigned encoding:4;

　　 unsigned lru:22; /\* lru time (relative to server.lruclock) \*/

　　 int refcount;

　　 void \*ptr;

　　 /\* VM fields are only allocated if VM is active, otherwise the

　　 \* object allocation function will just allocate

　　 \* sizeof(redisObjct) minus sizeof(redisObjectVM), so using

　　 \* Redis without VM active will not have any overhead. \*/

　　} robj;

关于存储的结构分析，公司五竹写的很好，网上可以查到http://www.searchtb.com/，在这个博客里面

## 基本数据结构

### Redis server基本数据结构

redisServer主要记录了server的全局信息，如数据库，连入的client，支持的所有操作，从配置文件中读取的配置信息等。

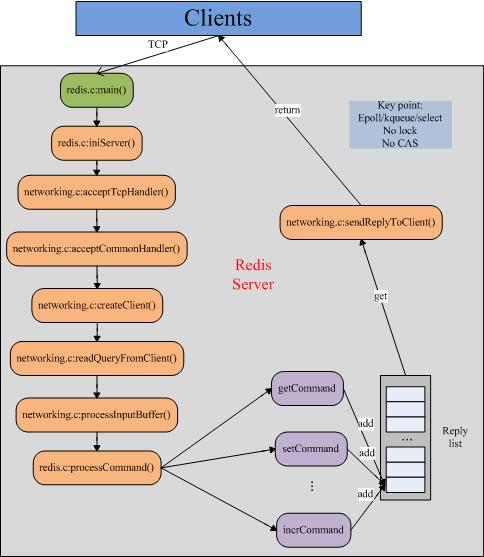
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //redis.h    **struct** redisServer {      pthread\_t mainthread; //主线程    **int** port; //端口号    **char** \*bindaddr; //地址      …….    **int** ipfd; //主线程的文件描述符      ……      redisDb \*db;      ……      list \*clients; /\* 当前接入的client列表 \*/      dict \*commands;             /\*支持的所有操作\*/      …    }; |

redisClient主要记录了某个接入客户端的状态信息，如客户端连接句柄，操作的数据库，发送的命令，返回的结果列表等。为每个client建立一个这种数据结构，可以很方便的支持多用户并发访问。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | **typedef** **struct** redisClient {    **int** fd;  //客户端连接句柄      redisDb \*db; //操作的数据库      ……      sds querybuf; //用户命令缓冲区    **int** argc;      robj \*\*argv;    **struct** redisCommand \*cmd;    **int** reqtype;    **int** multibulklen;       /\* number of multi bulk arguments left to read \*/    **long** bulklen;           /\* length of bulk argument in multi bulk request \*/      list \*reply; //用户命令的执行结果，会被异步的反馈给用户      ……  }; |

### Redis代码架构

下图描述了client与redis server的整个交互过程，图中只描述了代码关键路径。



（1）首先，从redis.c文件中的main函数开始：

Main函数首先调用：

|  |  |
| --- | --- |
| 1 | initServerConfig(); |

该函数主要完成以下功能：设定默认的参数值，并读取配置文件redis.conf，若用户配置了某个参数，则用该参数值替换默认值。

接下来，调用：

|  |  |
| --- | --- |
| 1 | initServer(); |

该函数主要对server进行初始化，初始化内容包括：

调用anetTcpServer函数创建socket server作为redis server，并将该server的句柄加到epoll/kqueue的监听队列中。一旦有client接入，便会对该client触发操作acceptTcpHandler，该操作是调用aeCreateFileEvent注册的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | …..  **if** (server.port != 0) {      server.ipfd = anetTcpServer(server.neterr,server.port,server.bindaddr);    **if** (server.ipfd == ANET\_ERR) {        redisLog(REDIS\_WARNING, "Opening port %d: %s",          server.port, server.neterr);    **exit**(1);      }    }  ……  aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL);    **if** (server.ipfd > 0 && aeCreateFileEvent(server.el,server.ipfd,AE\_READABLE,      acceptTcpHandler,NULL) == AE\_ERR) oom("creating file event");      …… |

【注】aeCreateFileEvent函数用于注册监听连接事件，即某个client向server发起连接或者发出一个命令后，会触发这个事件；而aeCreateTimeEvent函数用于注册定时任务serverCron，该函数每隔100 毫秒执行一次，主要进行一些后台处理，如：记日志，清除无效key，清除无效链接。

acceptTcpHandler函数会调用acceptCommonHander，而acceptCommonHander又会调用createClient来为该client创建一个redisClient对象….，最终，redis会根据用户输入的命令调用已经写好的命令执行函数，这些函数已经被写死，保存到一个全局只读表中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **struct** redisCommand readonlyCommandTable[] = {      {"get",getCommand,2,0,NULL,1,1,1},      {"set",setCommand,3,REDIS\_CMD\_DENYOOM,NULL,0,0,0},      {"setnx",setnxCommand,3,REDIS\_CMD\_DENYOOM,NULL,0,0,0},      {"setex",setexCommand,4,REDIS\_CMD\_DENYOOM,NULL,0,0,0},      {"append",appendCommand,3,REDIS\_CMD\_DENYOOM,NULL,1,1,1},      ……    }; |

用户输入get命令，redis最终会调用getCommand函数，用户输入set命令，redis最终会调用setCommand函数…..。该表是在initServerConfig()函数中，被加载到一个hash table中的，以便于后面的查找：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | //initServerConfig()    ……    populateCommandTable();    ……    // populateCommandTable()    **int** j;    **int** numcommands = **sizeof**(readonlyCommandTable)/**sizeof**(**struct** redisCommand);    **for** (j = 0; j < numcommands; j++) {    **struct** redisCommand \*c = readonlyCommandTable+j;    **int** retval;      retval = dictAdd(server.commands, sdsnew(c->name), c);    **assert**(retval == DICT\_OK);    }    } |

（2）redis执行完用户的一个命令后，会将结果写入到redisClient对象中的reply list中，而sendReplyToClient函数会不断的从该list中数据，异步地发送给client。需要注意的是，sendReplyToClient函数也是通过aeCreateFileEvent注册的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **int** \_installWriteEvent(redisClient \*c) {    **if** (c->fd <= 0) **return** REDIS\_ERR;    **if** (c->bufpos == 0 && listLength(c->reply) == 0 &&        (c->replstate == REDIS\_REPL\_NONE ||          c->replstate == REDIS\_REPL\_ONLINE) &&            aeCreateFileEvent(server.el, c->fd, AE\_WRITABLE,              sendReplyToClient, c) == AE\_ERR) **return** REDIS\_ERR;    **return** REDIS\_OK;    }    **void** addReply(redisClient \*c, robj \*obj) {    **if** (\_installWriteEvent(c) != REDIS\_OK) **return**;      ……    } |

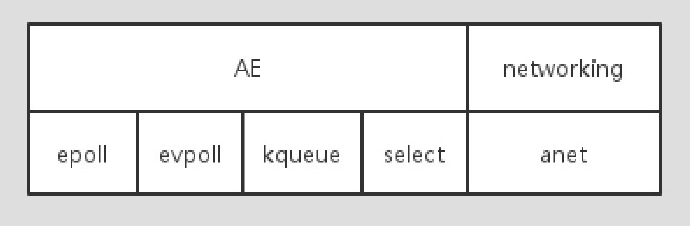
上面只是粗略讲了一下代码架构，只算是抛砖引玉了，如果读者想更进一步了解redis代码架构，最好亲自读一下代码。总体而言，redis代码结构很清晰，架构也是比较简单。

# 网络通信架构

最近略有闲暇时间，于是对Redis进行了一些学习，学习途径除了官方文档还有Redis源代码，我看的版本是2.8.13，Redis源码总行数不到5W行，不同组件拆分非常细致，阅读起来也很清晰。这篇博客主要介绍我对Redis网络层架构以及线程模型的一些了解，希望能对大家有所帮助。

## 网络基础架构

网络编程离不开Socket，网络I/O模型最常用的无非是同步阻塞、同步非阻塞、异步阻塞、异步非阻塞，高性能网络服务器最常见的线程模型也就是基于EventLoop模式的单线程模型。我们看看Redis的网络架构是怎么样的：



Redis基础组建结构

这里解释下上图涉及的组件，Redis网络层基础组件主要包括四个部分：

1. EventLoop事件轮训器，这部分实现在AE里面。
2. 提供Socket句柄事件的多路复用器，这部分分别对于不同平台提供了不同的实现，比如epoll和select可以用于linux平台、kqueue可以用于苹果平台、evpoll可以用于Solaris平台，这里并没有看到iocp，也就是Redis对于Windows支持并不是很好。
3. 包括网络事件处理器实现的networking，这部分主要包括两个重要的今天要讲的事件处理器：acceptTcpHandler和acceptCommonHandler。
4. 处理网络比较底层的部分，比如网络句柄创建、网络的读写等。

### Redis单线程模型

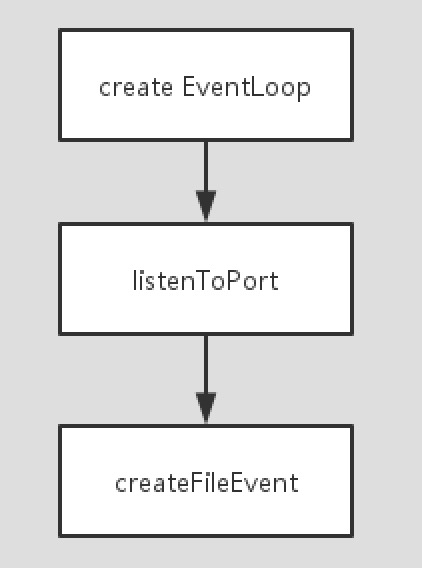
要理解Redis的单线程模型，我们先抛出一些问题，当我们有多个客户端同时去跟Redis Server建立连接，之后又同时对某个key进行操作，这个过程中发生了什么呢？会不会有并发问题？

### 网络初始化

好了，这些问题先丢在这了，我们看看Redis启动初始化的过程中会做什么事情，这里尽量省略了与本文无关的部分：

1. 初始化Redis Server参数，这部分代码通过initServerConfig实现。
2. 初始化Redis Server，这部分代码在initServer里面。
3. 启动事件轮训器。

对，这里我们就把Redis的启动部分简化为三步，跟网络操作有关的主要在第二步和第三步里面，来看看initServer里面发生了什么：

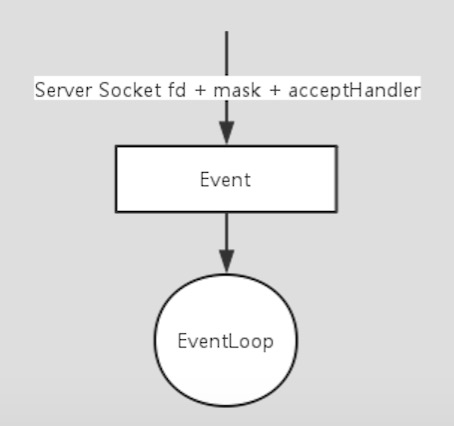


initServer流程

initServer里面首先创建了一个EventLoop，然后监听Server的IP对应的端口号，假设我们监听的是127.0.0.1：3333这个IP:端口对，我们得到的一个Server Socket句柄，最后通过createFileEvent将我们得到的Server Socket句柄和我们关心的网络事件mask注册到EventLoop上面。EventLoop是什么呢，我们看看它的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **typedef** **struct** aeEventLoop {  **int** maxfd;   */\* highest file descriptor currently registered \*/*  **int** setsize; */\* max number of file descriptors tracked \*/*  **long** **long** timeEventNextId;      time\_t lastTime;     */\* Used to detect system clock skew \*/*      aeFileEvent \*events; */\* Registered events \*/*      aeFiredEvent \*fired; */\* Fired events \*/*      aeTimeEvent \*timeEventHead;  **int** stop;  **void** \*apidata; */\* This is used for polling API specific data \*/*      aeBeforeSleepProc \*beforesleep;  } aeEventLoop; |

上面我们关注的主要是两个东西：events和fired。他们分别是两个数组，events用于存放被注册的事件以及相应的句柄，fired用于存放当EventLoop线程从多路复用器轮训到有事件的句柄的时候，EventLoop线程会把它放入fired数组里面，然后处理。



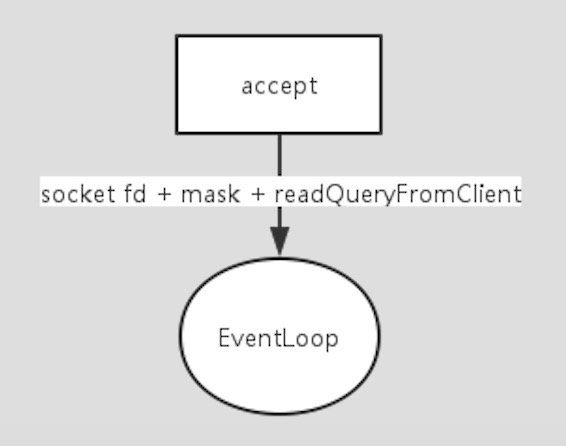
****事件注册示意图****

我用上面的示意图描述createFileEvent做的事情，就是将Server Socket句柄和关心的事件mask以及当事件产生的时候的事件处理器accptHandler生成一个aeFileEvent注册到EventLoop的events的数组里面，当然在这之前会首先将事件注册到多路复用器上，也就是epoll、kqueue等这些组件上。事件注册完之后需要对多路复用器进行轮训，来分离我们关心切发生的事件，那就是最后一步，启动事件轮询器。

### 接收网络连接

上面的步骤完成了服务端的网络初始化，而且事件轮询器已经开始工作了，事件轮询器做什么事情呢，就是不断轮训多路复用器，看看之前注册的事件有没有发生，如果有发生，则将会将事件分离出来，放入EventLoop的fired数组中，然后处理这些事件。

很显然，上面注册的事件是客户端建立连接这个事件，因此当有两个客户端同时连接Redis服务器的时候，事件轮询器会从多路复用器上面分离出这个事件，同时调用acceptHandler来处理。acceptHandler做的事情主要是accept客户端的连接，创建socket句柄，然后将socket句柄和读事件注册到EventLoop的events数组里面，不一样的是对于客户端的事件处理器是readQueryClient。

****

****accept客户端连接以及注册客户端连接句柄示意图****

上面示意图表示了acceptHandler处理客户端连接，得到句柄之后再将这个句柄注册到多路复用器以及EventLoop上的示意图。之后再同样再处理下一个客户端的连接，这些都是串行的。

### 事件轮训

上面接收客户端这部分其实都发生在事件轮训的主循环里面：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **void** aeMain(aeEventLoop \*eventLoop) {      eventLoop->stop = 0;  **while** (!eventLoop->stop) {  **if** (eventLoop->beforesleep != **NULL**)              eventLoop->beforesleep(eventLoop);          aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);      }  } |

Redis会不断的轮训多路复用器，将网络事件分离出来，如果是accept事件，则新接收客户端连接并将其注册到多路复用器以及EventLoop中，如果是查询事件，则通过读取客户端的命令进行相应的处理，这一切都是单线程，顺序的执行的，因此不会发生并发问题。

## 应用分析

Redis官网对Redis的读写性能测试结果达到10左右，这是非常吸引人的。Redis的单线程的行为主要是对内存的读写，这些操作其实用不了多少时间，因此瓶颈在网络I/O上面，我们一般提供较好的网络环境就可以提升Redis的吞吐量，比如提高网络带宽，除此之外还可以通过合并命令提交批处理请求来代替单条命令一次次请求从而减少网络开销，提高吞吐量。