Hiredis是Redis数据库的C客户端库。

除了支持发送命令和接收回复之外，它还具有一个与IO层解耦的回复解析器。

Hiredis仅支持二进制安全的redis协议，因此适用于1.2.0以上的Redis版本。

Hiredis的API包含异步API，同步API，以及回复解析API。

一：同步API

使用同步API，仅仅需要以下三个API：

redisContext \*redisConnect(const char \*ip, int port);

void \*redisCommand(redisContext \*c, const char \*format, ...);

void freeReplyObject(void \*reply);

1：连接

redisConnect函数用于创建一个所谓的redis上下文：redisContext。该上下文用于保存一个连接的状态。redisContext结构中，整数成员err为非0时，表示连接发生了错误，成员errstr包含一个描述该错误的字符串。因此，在调用redisConnect函数连接Redis之后，需要检查err成员，判断连接是否成功：

redisContext \*c = redisConnect("127.0.0.1", 6379);

if (c != NULL && c->err) {

printf("Error: %s\n", c->errstr);

// handle error

}

2：发送命令

有多种向Redis发送命令的方式，第一种就是通过调用redisCommand函数。该函数接收类似于printf的参数，比如：

reply = redisCommand(context, "SET foo bar");

可以在格式字符串中使用’%s’，表示在命令中插入一个字符串，此时使用strlen判断字符串的长度：

reply = redisCommand(context, "SET foo %s", value);

如果需要在命令中传递二进制安全的字符串，可以使用”%b”，此时需要一个size\_t类型的参数表示该字符串的长度：

reply = redisCommand(context, "SET foo %b", value, (size\_t) valuelen);

在Hireds内部，使用一个或多个空格分隔命令参数，并将其转换成协议格式发送个Redis。可以使用多个%s：

reply = redisCommand(context, "SET key:%s %s", myid, value);

2：解析回复

当命令成功执行后，redisCommand函数返回该命令的回复信息。当错误发生时，返回值为NULL，并且会设置上下文中的err成员。一旦发生了错误，该上下文将不再可用，需要重新建立一个连接：

redisCommand函数的标准回复是一个redisReply类型的指针。该结构中的type成员表示回复信息的类型：

REDIS\_REPLY\_STATUS：状态回复。reply->str中保存Redis回复的状态字符串，该字符串的长度保存在reply->len中。

REDIS\_REPLY\_ERROR：错误回复。Redis回复的错误信息与状态信息一样，保存在reply->str中，字符串的长度在reply->len中。

REDIS\_REPLY\_INTEGER：整数回复。Redis回复的整数信息保存在long long类型的reply->integer成员中。

REDIS\_REPLY\_NIL：Redis回复”nil”。

REDIS\_REPLY\_STRING：单字符串回复。在reply->str中保存了Redis回复的字符串，该字符串的长度保存在reply->len中。

REDIS\_REPLY\_ARRAY：多字符串回复。Redis回复的字符串个数保存在reply->elements中，回复的每个字符串都是一个redisReply结构，可以通过reply->element[..index..]访问。

回复的redisReply结构需要使用freeReplyObject函数进行释放。如果该结构的reply->element中包含子回复的话，该函数也能很好的处理，因此用户无需特殊处理。

注意：当前的Hiredis版本（0.10.0），在使用异步API时会自动释放回复。也就是说无需手动调用freeReplyObject函数，回复信息会在hiredis中，调用回调函数之后自动释放。该行为或许会在以后的版本中有所改变。

3：清理

要想与Redis断链并且清理上下文，可以使用下列函数：

void redisFree(redisContext \*c);

该函数会立即关闭socket描述符，并且释放在创建上下文时申请的空间。

4：发送命令

除了redisCommand函数，还可以使用redisCommandArgv函数发送命令：

void \*redisCommandArgv(redisContext \*c, int argc, const char \*\*argv, const size\_t \*argvlen);

参数argc为命令参数的个数，数组argv中保存各个命令参数，整数数组argvlen中保存每个命令参数的长度。如果argvlen置为NULL，则该函数使用strlen判断每个命令参数的长度。因此，如果命令参数需要是二进制安全的，则需要提供argvlen参数。

该函数的返回值与redisCommand一样。

5：管道

当调用redisCommand类的函数时，Hiredis首先将命令字符串转换为Redis协议格式，然后将格式化后的命令放到上下文中的输出缓存中，该输出缓存是动态增长的，因此可以保存任意长度的命令。

将格式化的命令放到输出缓存之后，就会调用redisGetReply函数，该函数的执行流程如下：

a：如果输入缓存非空：解析输入缓存中的单个回复，并将其返回；如果没有可解析的回复了，则执行步骤b；

b：如果输入缓存为空：将整个输出缓存中的内容发送出去，从socket中读取内容，直到有一个回复可解析了。

redisGetReply函数作为Hiredis的API之一被暴露出来，可以在读取socket上的回复时使用该函数。

对于管道命令来说，唯一要做的事就是填充输出缓存。可以使用以下的函数：

void redisAppendCommand(redisContext \*c, const char \*format, ...);

void redisAppendCommandArgv(redisContext \*c, int argc, const char \*\*argv, const size\_t \*argvlen);

当调用以上函数若干次之后，可以调用redisGetReply函数接收回复。这俩函数的返回值要么是REDIS\_OK，要么是REDIS\_ERR。返回REDIS\_ERR表明读取回复时发生了错误，可以通过上下文中的err找出错误的原因。

下面的例子展示了管道的用法：

redisReply \*reply;

redisAppendCommand(context,"SET foo bar");

redisAppendCommand(context,"GET foo");

redisGetReply(context,&reply); // reply for SET

freeReplyObject(reply);

redisGetReply(context,&reply); // reply for GET

freeReplyObject(reply);

该API可以用于实现一个阻塞的订阅者：

reply = redisCommand(context,"SUBSCRIBE foo");

freeReplyObject(reply);

while(redisGetReply(context,&reply) == REDIS\_OK) {

// consume message

freeReplyObject(reply);

}

6：错误

当函数调用失败时，就会设置上下文中的err成员为非0值。err的取值可以是下列之一：

REDIS\_ERR\_IO：创建连接、写socket，读socket时发生了IO错误。可以使用全局变量errno判断错误原因。

REDIS\_ERR\_EOF：服务端关闭连接；

REDIS\_ERR\_PROTOCOL：解析协议时发生了错误；

REDIS\_ERR\_OTHER：其他错误。目前，该错误仅用于主机名无法解析的情况；

发生错误时，除了会设置err之外，还会设置errstr为描述错误信息的字符串。

二：异步API

Hiredis提供了异步API，这些API可以方便的用于各种事件库中，比如libev和libevent等。

1：建链

函数redisAsyncConnect用于创建一个到Redis的非阻塞的连接。该函数返回一个指向新创建的redisAsyncContext结构的指针。调用该函数后，需要检查其中的err属性来判断是否发生了错误。

redisAsyncContext \*c = redisAsyncConnect("127.0.0.1", 6379);

if (c->err) {

printf("Error: %s\n", c->errstr);

// handle error

}

异步上下文中，可以设置一个断链回调函数，当发生断链时就会调用该函数，该函数的原型如下：

void (const redisAsyncContext \*c, int status);

如果是用户主动断链，则参数status将被置为REDIS\_OK，如果是因为发生了错误而导致的断链，则该参数被置为REDIS\_ERR，这种情况下，可以通过上下文中的err属性判断错误的原因。

当断链回调函数触发后，上下文对象将会被释放。当需要重新建链时，可以在断链回调函数中执行重新建链操作。

每个上下文只能设置一次断链回调函数。后续的设置会直接返回REDIS\_ERR。设置断链回调函数，可以通过以下的函数实现：

int redisAsyncSetDisconnectCallback(redisAsyncContext \*ac, redisDisconnectCallback \*fn);

2：发送命令

在异步环境中，因为事件循环的特性，命令是自动发送的。发送命令时，需要设置收到回复时的回调函数。该回调函数的原型如下：

void(redisAsyncContext \*c, void \*reply, void \*privdata);

在异步环境中产生命令的函数如下：

int redisAsyncCommand(

redisAsyncContext \*ac, redisCallbackFn \*fn, void \*privdata,

const char \*format, ...);

int redisAsyncCommandArgv(

redisAsyncContext \*ac, redisCallbackFn \*fn, void \*privdata,

int argc, const char \*\*argv, const size\_t \*argvlen);

这两个函数类似于相应的阻塞版本。如果命令成功追加到输出缓存中，则函数返回REDIS\_OK，否则返回REDIS\_ERR。比如如果用户请求断链的话，就不会向输出缓存中追加新命令，因此需要返回REDIS\_ERR。

当命令回复的回调函数置为NULL时，该命令的回复会被立即释放。如果回调函数非NULL，则在回调函数调用之后，会释放命令回复。

如果调用回调函数之前，上下文发生了错误，则会以NULL调用回调。

3：断链

异步连接可以通过下面的函数终止：

void redisAsyncDisconnect(redisAsyncContext \*ac);

调用该函数后，连接不会立即断开。此时不会接受新的命令，只有当输出缓存中的命令全部发送出去后，读取到命令回复并调用回复回调函数之后，才会终止连接。这之后，断链回调函数会以REDIS\_OK为status参数被执行，然后就会释放上下文对象。

三：解析回复API

解析回复API包含下列函数：

redisReader \*redisReaderCreate(void);

void redisReaderFree(redisReader \*reader);

int redisReaderFeed(redisReader \*reader, const char \*buf, size\_t len);

int redisReaderGetReply(redisReader \*reader, void \*\*reply);

在Hiredis内部，创建上下文时也会使用这些函数。

1：用法

redisReaderCreate函数创建一个redisReader结构，该结构具有未解析数据缓存，以及协议解析器的状态属性。

socket上读取的数据，可以通过redisReaderFeed函数将其放入到redisReader中的内部缓存中。

当调用函数redisReaderGetReply时，就会解析数据。该函数通过void \*\*reply参数返回一个整数状态，以及一个回复结构。回复状态可以是REDIS\_OK或REDIS\_ERR。

解析器限制多字符串回复的嵌套最大层数为7，当回复的嵌套层数大于该值时，解析器会返回错误。

2：回复

redisReaderGetReply函数创建redisReply结构，并使参数reply指向该结构。比如，如果命令回复为REDIS\_REPLY\_STATUS，则redisReply结构的str属性将会置为C字符串。可以通过设置redisReader结构中的fn属性，设置创建redisReply结构的函数。只有在创建redisReader之后立即这样做。

3：读取的最大缓存

不管是使用Reader api，还是通过Redis上下文使用它，redisReader结构使用缓存接收Redis服务器发来的数据。

当数据为空，或者数据量大于16kb时，该缓存会被释放。

可以设置缓存的最大容量，如果置为0，表示无限制。

context->reader->maxbuf = 0;

这种方式只有需要在与大量回复的情况下设置，从而可以提高性能。只要在没有大回复的情况下，应该立即将其设置回REDIS\_READER\_MAX\_BUF，从而节约内存。

redisContext \*redisConnect(const char \*ip, int port);

redisContext \*redisConnectWithTimeout(const char \*ip, int port, const struct timeval tv);

函数创建一个redisContext结构，并且根据ip和port，向redis建链。这是阻塞版本的建联函数，因此该函数返回后，建链要么成功，要么失败。该函数返回redisContext结构，如果建链失败，则其中的err属性和errstr属性都会设置相应的错误码和错误信息。

（设置connect超时的方法）

void \*redisCommand(redisContext \*c, const char \*format, ...);

该函数接收命令字符串，发送给Redis服务器，等待Redis的回复。该函数返回后，如果发生错误，则函数返回NULL，否则，返回Redis服务器的回复信息。

该函数，以及redisvCommand函数，都是首先调用redisvAppendCommand，将命令转换成统一请求协议格式之后，追加到输出缓存c->obuf中。

然后调用\_\_redisBlockForReply函数将命令发送给Redis，并读取其回复。

int redisvFormatCommand(char \*\*target, const char \*format, va\_list ap);

解析命令字符串，将解析后的命令转换成统一请求协议格式，并使\*target指向转换后的请求报文。

int \_\_redisAppendCommand(redisContext \*c, const char \*cmd, size\_t len):

将长度为len的字符串cmd，追加到上下文的输出缓存c->obuf中。追加失败返回REDIS\_ERR，否则，返回REDIS\_OK。

int redisBufferWrite(redisContext \*c, int \*done);

将输出缓存c->obuf中的内容写入到socket描述符中。当输出参数down被置为1时，表明输出缓存c->obuf中的内容已经全部发送出去了，目前c->obuf为空了。

在同步连接中，该函数是在循环中调用的，直到down为1时才退出循环。

int redisReaderFeed(redisReader \*r, const char \*buf, size\_t len);

将长度为len的buf，追加到解析器redisReader结构中的输入缓存buf属性中，r->len表示r->buf当前的长度。r->pos表示缓存读索引，每次从缓存中读取数据时，都是从r->buf[r->pos]开始读取。

如果解析器设置了r->maxbuf，该属性表示该解析器中的buf，允许的最大闲置空间。为了节省内存空间，当buf为空，并且当前buf的闲置空间大于r->maxbuf时，就会释放r->buf，重新为其申请空间。

该属性的默认值为16K。如果置为0，表示无此限制。

int redisBufferRead(redisContext \*c);

该函数从socket描述符中读取数据，并调用redisReaderFeed函数，将读取到的数据追加到解析器输入缓存c->reader->buf中。

static char \*readBytes(redisReader \*r, unsigned int bytes);

从解析器缓存r->buf中，读取bytes个字节。首先判断当前缓存中是否有足够的数据，如果没有足够数据直接返回NULL，否则，返回r->buf+r->pos的值，并将r->pos增加bytes，表示读取了bytes个字节。

static char \*readLine(redisReader \*r, int \*\_len);

从解析器缓存r->buf中，读取1行数据。返回该行数据的首地址。

首先从r->buf+r->pos开始寻找”\r\n”，如果找不到，则直接返回NULL。如果找到了，则将输出参数\_len置为该行中有效数据的长度（不包括”\r\n”），然后将r->pos增加该行的长度（包括”\r\n”），返回该行首地址。