

## Writing system software: code comments.

antirez 1997 days ago. 317682 views.

For quite some time I've wanted to record a new video talking about code comments for my "writing system software" series on YouTube. However, after giving it some thought, I realized that the topic was better suited for a blog post, so here we are. In this post I analyze Redis comments, trying to categorize them. Along the way I try to show why, in my opinion, writing comments is of paramount importance in order to produce good code, that is maintainable in the long run and understandable by others and by the authors during modifications and debugging activities.

Not everybody thinks likewise. Many believe that comments are useless if the code is solid enough. The idea is that when everything is well designed, the code itself documents what the code is doing, hence code comments are superfluous. I disagree with that vision for two main reasons:

1. Many comments don't explain what the code is doing. They explain what you can't understand just from what the code does. Often this missing information is *\*why\** the code is doing a certain action, or why it's doing something that is clear instead of something else that would feel more natural.
2. While it is not generally useful to document, line by line, what the code is doing, because it is understandable just by reading it, a key goal in writing readable code is to lower the amount of effort and the number of details the reader should take into her or his head while reading some code. So comments can be, for me, a tool for lowering the cognitive load of the reader.

The following code snippet is a good example of the second point above. Note that all the code snippets in this blog post are obtained from the Redis source code. Every code snippet is presented prefixed by the file name it was extracted from. The branch used is the current "unstable" with hash 32e0d237.

```
scripting.c:
    /* Initial Stack: array */
    lua_getglobal(lua,"table");
    lua_pushstring(lua,"sort");
    lua_gettable(lua,-2);          /* Stack: array, table, table.sort */
    lua_pushvalue(lua,-3);        /* Stack: array, table, table.sort, array */
    if (lua_pcall(lua,1,0,0)) {
        /* Stack: array, table, error */
```

```

    /* We are not interested in the error, we assume that the problem is
    * that there are 'false' elements inside the array, so we try
    * again with a slower function but able to handle this case, that
    * is: table.sort(table, __redis__compare_helper) */
lua_pop(lua,1);          /* Stack: array, table */
lua_pushstring(lua,"sort"); /* Stack: array, table, sort */
lua_gettable(lua,-2);    /* Stack: array, table, table.sort */
lua_pushvalue(lua,-3);  /* Stack: array, table, table.sort, array */
lua_getglobal(lua,"__redis__compare_helper");
/* Stack: array, table, table.sort, array, __redis__compare_helper */
lua_call(lua,2,0);
}

```

Lua uses a stack based API. A reader following each call in the function above, having also a Lua API reference at hand, will be able to mentally reconstruct the stack layout at every given moment. But why to force the reader to do such effort? While writing the code, the original author had to do that mental effort anyway. What I did there was just to annotate every line with the current stack layout after every call. Reading this code is now trivial, regardless of the fact the Lua API is otherwise non trivial to follow.

My goal here is not just to offer my point of view on the usefulness of comments as a tool to provide a background that is not clearly available reading a local section of the source code. But also to also provide some evidence about the usefulness of the kind of comments that are historically considered useless or even dangerous, that is, comments stating *\*what\** the code is doing, and not why.

## # Classification of comments

The way I started this work was by reading random parts of the Redis source code, to check if and why comments were useful in different contexts. What quickly emerged was that comments are useful for very different reasons, since they tend to be very different in function, writing style, length and update frequency. I eventually turned the work into a classification task.

During my research I identified nine types of comments:

- \* Function comments
- \* Design comments
- \* Why comments
- \* Teacher comments
- \* Checklist comments
- \* Guide comments
- \* Trivial comments

- \* Debt comments
- \* Backup comments

The first six are, in my opinion, mostly very positive forms of commenting, while the final three are somewhat questionable. In the next sections each type will be analyzed with examples from the Redis source code.

## FUNCTION COMMENTS

The goal of a function comment is to prevent the reader from reading code in the first place. Instead, after reading the comment, it should be possible to consider some code as a black box that should obey certain rules. Normally function comments are at the top of functions definitions, but they may be at other places, documenting classes, macros, or other functionally isolated blocks of code that define some interface.

rax.c:

```
/* Seek the gretest key in the subtree at the current node. Return 0 on
 * out of memory, otherwise 1. This is an helper function for different
 * iteration functions below. */
int raxSeekGreatest(raxIterator *it) {
    ...
}
```

Function comments are actually a form of in-line API documentation. If the function comment is written well enough, the user should be able most of the times to jump back to what she was reading (reading the code calling such API) without having to read the implementation of a function, a class, a macro, or whatever.

Among all the kinds of comments, these are the ones most widely accepted by the programming community at large as needed. The only point to analyze is if it is a good idea to place comments that are largely API reference documentation inside the code itself. For me the answer is simple: I want the API documentation to exactly match the code. As the code is changed, the documentation should be changed. For this reason, by using function comments as a prologue of functions or other elements, we make the API documentation close to the code, accomplishing three results:

- \* As the code is changed, the documentation can be easily changed at the same time, without the risk of making the API reference stale.
- \* This approach maximizes the probability that the author of the change, that should be the one better understanding the change, will also be the author of the API documentation change.
- \* Reading the code is handy to find the documentation of functions or methods

directly where they are defined, so that the reader of the code can focus solely on the code, instead of context switching between code and documentation.

## DESIGN COMMENTS

While a "function comment" is usually located at the start of a function, a design comment is more often located at the start of a file. The design comment basically states how and why a given piece of code uses certain algorithms, techniques, tricks, and implementation. It is an higher level overview of what you'll see implemented in the code. With such background, reading the code will be simpler. Moreover I tend to trust more code where I can find design notes. At least I know that some kind of explicit design phase happened, at some point, during the development process.

In my experience design comments are also very useful in order to state, in case the solution proposed by the implementation looks a bit too trivial, what were the competing solutions and why a very simple solution was considered to be enough for the case at hand. If the design is correct, the reader will convince herself that the solution is appropriate and that such simplicity comes from a process, not from being lazy or only knowing how to code basic things.

bio.c:

```
* DESIGN
* -----
*
* The design is trivial, we have a structure representing a job to perform
* and a different thread and job queue for every job type.
* Every thread waits for new jobs in its queue, and process every job
* sequentially.
...
```

## WHY COMMENTS

Why comments explain the reason why the code is doing something, even if what the code is doing is crystal clear. See the following example from the Redis replication code.

replication.c:

```
if (idle > server.repl_backlog_time_limit) {
    /* When we free the backlog, we always use a new
     * replication ID and clear the ID2. This is needed
     * because when there is no backlog, the master_repl_offset
     * is not updated, but we would still retain our replication
     * ID, leading to the following problem:
     *
     */
}
```

```

* 1. We are a master instance.
* 2. Our replica is promoted to master. It's repl-id-2 will
*    be the same as our repl-id.
* 3. We, yet as master, receive some updates, that will not
*    increment the master_repl_offset.
* 4. Later we are turned into a replica, connect to the new
*    master that will accept our PSYNC request by second
*    replication ID, but there will be data inconsistency
*    because we received writes. */
changeReplicationId();
clearReplicationId2();
freeReplicationBacklog();
serverLog(LL_NOTICE,
          "Replication backlog freed after %d seconds "
          "without connected replicas.",
          (int) server.repl_backlog_time_limit);
}

```

If I check just the function calls there is very little to wonder: if a timeout is reached, change the main replication ID, clear the secondary ID, and finally free the replication backlog. However what is not exactly clear is why we need to change the replication IDs when freeing the backlog.

Now this is the kind of thing that happens continuously in software once it has reached a given level of complexity. Regardless of the code involved, the replication protocol has some level of complexity itself, so we need to do certain things in order to make sure that other bad things can't happen. Probably these kind of comments are, in some way, opportunities to reason about the system and check if it should be improved, so that such complexity is no longer needed, hence also the comment can be removed. However often making something simpler may make something else harder or is simply not viable, or requires future work breaking backward compatibility.

Here is another one.

replication.c:

```

/* SYNC can't be issued when the server has pending data to send to
* the client about already issued commands. We need a fresh reply
* buffer registering the differences between the BGSAVE and the current
* dataset, so that we can copy to other replicas if needed. */
if (clientHasPendingReplies(c)) {
    addReplyError(c,"SYNC and PSYNC are invalid with pending output");
    return;
}

```

If you run SYNC while there is still pending output (from a past command) to send to the client, the command should fail because during the replication handshake the output buffer of the client is used to accumulate changes, and may be later duplicated to serve other replicas connecting while we are already creating the RDB file for the full sync with the first replica. This is the why we do that. What we do is trivial. Pending replies? Emit an error. Why is rather obscure without the comment.

One may think that such comments are needed only when describing complex protocols and interactions, like in the case of replication. Is that the case? Let's change completely file and goals, and we see still such comments everywhere.

expire.c:

```
for (j = 0; j < dbs_per_call && timelimit_exit == 0; j++) {
    int expired;
    redisDb *db = server.db+(current_db % server.dbnum);

    /* Increment the DB now so we are sure if we run out of time
     * in the current DB we'll restart from the next. This allows to
     * distribute the time evenly across DBs. */
    current_db++;
    ...
}
```

That's an interesting one. We want to expire keys from different DBs, as long as we have some time. However instead of incrementing the “database ID” to process next at the end of the loop processing the current database, we do it differently: we select the current DB in the `db` variable, but then we immediately increment the ID of the next database to process (at the next call of this function). This way if the function terminates because too much effort was spent in a single call, we don't have the problem of restarting again from the same database, letting logically expired keys accumulating in the other databases since we are too focused in processing the same database again and again.

With such comment we both explain why we increment at that stage, and that the next person going to modify the code, should preserve such quality. Note that without the comment the code looks completely harmless. Select, increment, go to do some work. There is no evident reason for not relocating the increment at the end of the loop where it could look more natural.

Trivia: the loop increment was indeed at the end in the original code. It was moved there during a fix: at the same time the comment was added. So let's say this is kinda of a "regression comment".

TEACHER COMMENTS

Teacher comments don't try to explain the code itself or certain side effects we should be aware of. They teach instead the \*domain\* (for example math, computer graphics, networking, statistics, complex data structures) in which the code is operating, that may be one outside of the reader skills set, or is simply too full of details to recall all them from memory.

The LOLWUT command in version 5 needs to display rotated squares on the screen (<http://antirez.com/news/123>). In order to do so it uses some basic trigonometry: despite the fact that the math used is simple, many programmers reading the Redis source code may not have any math background, so the comment at the top of the function explains what's going to happen inside the function itself.

lolwut5.c:

```
/* Draw a square centered at the specified x,y coordinates, with the specified
 * rotation angle and size. In order to write a rotated square, we use the
 * trivial fact that the parametric equation:
 *
 * x = sin(k)
 * y = cos(k)
 *
 * Describes a circle for values going from 0 to 2*PI. So basically if we
start
 * at 45 degrees, that is k = PI/4, with the first point, and then we find
 * the other three points incrementing K by PI/2 (90 degrees), we'll have the
 * points of the square. In order to rotate the square, we just start with
 * k = PI/4 + rotation_angle, and we are done.
 *
 * Of course the vanilla equations above will describe the square inside a
 * circle of radius 1, so in order to draw larger squares we'll have to
 * multiply the obtained coordinates, and then translate them. However this
 * is much simpler than implementing the abstract concept of 2D shape and then
 * performing the rotation/translation transformation, so for LOLWUT it's
 * a good approach. */
```

The comment does not contain anything that is related to the code of the function itself, or its side effects, or the technical details related to the function. The description is only limited to the mathematical concept that is used inside the function in order to reach a given goal.

I think teacher comments are of huge value. They teach something in case the reader is not aware of such concepts, or at least provide a starting point for further investigation. But this in turn means that a teacher comment increases the amount of programmers that can read some code path: writing code that can be read by many programmers is a major goal of mine. There are developers that may not have math skills but are very solid programmers that can contribute some wonderful

fix or optimization. And in general code should be read other than being executed, since is written by humans for other humans.

There are cases where teacher comments are almost impossible to avoid in order to write decent code. A good example is the Redis radix tree implementation. Radix trees are articulated data structures. The Redis implementation re-states the whole data structure theory as it implements it, showing the different cases and what the algorithm does to merge or split nodes and so forth. Immediately after each section of comment, we have the code implementing what was written before. After months of not touching the file implementing the radix tree, I was able to open it, fix a bug in a few minutes, and continue doing something else. There is no need to study again how a radix tree works, since the explanation is the same thing as the code itself, all mixed together.

The comments are too long, so I'll just show certain snippets.

rax.c:

```
/* If the node we stopped at is a compressed node, we need to
 * split it before to continue.
 *
 * Splitting a compressed node have a few possible cases.
 * Imagine that the node 'h' we are currently at is a compressed
 * node containing the string "ANNIBALE" (it means that it represents
 * nodes A -> N -> N -> I -> B -> A -> L -> E with the only child
 * pointer of this node pointing at the 'E' node, because remember that
 * we have characters at the edges of the graph, not inside the nodes
 * themselves.
 *
 * In order to show a real case imagine our node to also point to
 * another compressed node, that finally points at the node without
 * children, representing '0':
 *
 *     "ANNIBALE" -> "SCO" -> []
 *
 * ... snip ...
 *
 * 3a. IF $SPLITPOS == 0:
 *     Replace the old node with the split node, by copying the auxiliary
 *     data if any. Fix parent's reference. Free old node eventually
 *     (we still need its data for the next steps of the algorithm).
 *
 * 3b. IF $SPLITPOS != 0:
 *     Trim the compressed node (reallocating it as well) in order to
 *     contain $splitpos characters. Change child pointer in order to link
 *     to the split node. If new compressed node len is just 1, set
```



```
* iscompr to 0 (layout is the same). Fix parent's reference.
```

```
... snip ...
```

```
if (j == 0) {
    /* 3a: Replace the old node with the split node. */
    if (h->iskey) {
        void *ndata = raxGetData(h);
        raxSetData(splitnode,ndata);
    }
    memcpy(parentlink,&splitnode,sizeof(splitnode));
} else {
    /* 3b: Trim the compressed node. */
    trimmed->size = j;
    memcpy(trimmed->data,h->data,j);
    trimmed->iscompr = j > 1 ? 1 : 0;
    trimmed->iskey = h->iskey;
    trimmed->isnull = h->isnull;
    if (h->iskey && !h->isnull) {
        void *ndata = raxGetData(h);
        raxSetData(trimmed,ndata);
    }
    raxNode **cp = raxNodeLastChildPtr(trimmed);
    ...
}
```

As you can see the description in the comment is then matched with the same labels in the code. It's hard to show it all in this form so if you want to get the whole idea just check the full file at:

<https://github.com/antirez/redis/blob/unstable/src/rax.c>

This level of commenting is not needed for everything, but things like radix trees are really full of little details and corner cases. They are hard to recall, and certain details are *specific* to a given implementation. Doing this for a linked list does not make much sense of course. It's a matter of personal sensibility to understand when it's worth it or not.

## CHECKLIST COMMENTS

This is a very common and odd one: sometimes because of language limitations, design issues, or simply because of the natural complexity arising in systems, it is not possible to centralize a given concept or interface in one piece, so there are places in the code that tells you to remember to do things in some other place of the code. The general concept is:

```
/* Warning: if you add a type ID here, make sure to modify the
```

```
* function getTypeNameByID() as well. */
```

In a perfect world this should never be needed, but in practice sometimes there are no escapes from that. For example Redis types could be represented using an "object type" structure, and every object could link to the type the object it belongs, so you could do:

```
printf("Type is %s\n", myobject->type->name);
```

But guess what? It's too expensive for us, because a Redis object is represented like this:

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                           * LFU data (least significant 8 bits frequency
                           * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

We use 4 bits instead of 64 to represent the type. This is just to show why sometimes things are not as centralized and natural as they should be. When the situation is like that, sometimes what helps is to use defensive commenting in order to make sure that if a given code section is touched, it reminds you to make sure to also modify other parts of the code. Specifically a checklist comment does one or both of the following things:

- \* It tells you a set of actions to do when something is modified.

- \* It warns you about the way certain changes should be operated.

Another example in `blocked.c`, when a new blocking type is introduced.

`blocked.c`:

```
* When implementing a new type of blocking operation, the implementation
* should modify unblockClient() and replyToBlockedClientTimedOut() in order
* to handle the btype-specific behavior of this two functions.
* If the blocking operation waits for certain keys to change state, the
* clusterRedirectBlockedClientIfNeeded() function should also be updated.
```

The checklist comment is also useful in a context similar to when certain "why comments" are used: when it is not obvious why some code must be executed at a given place, after or before something. But while the why comment may tell you why

a statement is there, the checklist comment used in the same case is more biased towards telling you what rules to follow if you want to modify it (in this case the rule is, follow a given ordering), without breaking the code behavior.

cluster.c:

```
/* Update our info about served slots.
 *
 * Note: this MUST happen after we update the master/replica state
 * so that CLUSTER_NODE_MASTER flag will be set. */
```

Checklist comments are very common inside the Linux kernel, where the order of certain operations is extremely important.

#### GUIDE COMMENT

I abuse guide comments at such a level that probably, the majority of comments in Redis are guide comments. Moreover guide comments are exactly what most people believe to be completely useless comments.

- \* They don't state what is not clear from the code.

- \* There are no design hints in guide comments.

Guide comments do a single thing: they babysit the reader, assist him or her while processing what is written in the source code by providing clear division, rhythm, and introducing what you are going to read.

Guide comments' sole reason to exist is to lower the cognitive load of the programmer reading some code.

rax.c:

```
/* Call the node callback if any, and replace the node pointer
 * if the callback returns true. */
if (it->node_cb && it->node_cb(&it->node))
    memcpy(cp,&it->node,sizeof(it->node));
```

```
/* For "next" step, stop every time we find a key along the
 * way, since the key is lexicographically smaller compared to
 * what follows in the sub-children. */
```

```
if (it->node->iskey) {
    it->data = raxGetData(it->node);
```

```
    return 1;
```

```
}
```

There is nothing that the comments are adding to the code above. The guide comments above will assist you reading the code, moreover they'll acknowledge you about the fact you are understanding it right. More examples.

networking.c:

```
/* Log link disconnection with replica */
if ((c->flags & CLIENT_SLAVE) && !(c->flags & CLIENT_MONITOR)) {
    serverLog(LL_WARNING,"Connection with replica %s lost.",
        replicationGetSlaveName(c));
}

/* Free the query buffer */
sdsfree(c->querybuf);
sdsfree(c->pending_querybuf);
c->querybuf = NULL;

/* Deallocate structures used to block on blocking ops. */
if (c->flags & CLIENT_BLOCKED) unblockClient(c);
dictRelease(c->bpop.keys);

/* UNWATCH all the keys */
unwatchAllKeys(c);
listRelease(c->watched_keys);

/* Unsubscribe from all the pubsub channels */
pubsubUnsubscribeAllChannels(c,0);
pubsubUnsubscribeAllPatterns(c,0);
dictRelease(c->pubsub_channels);
listRelease(c->pubsub_patterns);

/* Free data structures. */
listRelease(c->reply);
freeClientArgv(c);

/* Unlink the client: this will close the socket, remove the I/O
 * handlers, and remove references of the client from different
 * places where active clients may be referenced. */
unlinkClient(c);
```

Redis is *literally* ridden of guide comments, so basically every file you open will contain plenty of them. Why bother? Of all the comment types I analyzed so far in this blog post, I'll admit that this is absolutely the most subjective one. I don't value code without such comments as less good, yet I firmly believe that if people regard the Redis code as readable, some part of the reason is because of

all the guide comments.

Guide comments have some usefulness other than the stated ones. Since they clearly divide the code in isolated sections, an addition to the code is very likely to be inserted in the appropriate section, instead of ending in some random part. To have related statements nearby is a big readability win.

Also make sure to check the guide comment above before the `unlinkClient()` function is called. The guide comment briefly tells the reader what the function is going to do, avoiding the need to jump back into the function if you are only interested in the big picture.

## TRIVIAL COMMENTS

Guide comments are very subjective tools. You may like them or not. I love them. However, a guide comment can degenerate into a a very bad comment: it can easily turn into a "trivial comment". A trivial comment is a guide comment where the cognitive load of reading the comment is the same or higher than just reading the associated code. The following form of trivial comment is exactly what many books will tell you to avoid.

```
array_len++;          /* Increment the length of our array. */
```

So if you write guide comments, make sure you avoid writing trivial ones.

## DEBT COMMENTS

Debt comments are technical debts statements hard coded inside the source code itself:

`t_stream.c`:

```
/* Here we should perform garbage collection in case at this point
 * there are too many entries deleted inside the listpack. */
entries -= to_delete;
marked_deleted += to_delete;
if (entries + marked_deleted > 10 && marked_deleted > entries/2) {
    /* TODO: perform a garbage collection. */
}
```

The snippet above is extracted from the Redis streams implementation. Redis streams allow to delete elements from the middle using the XDEL command. This may be useful in different ways, especially in the context of privacy regulations where certain data cannot be retained no matter what data structure or system you are using in order to store them. It is a very odd use case for a mostly append only data structure, but if users start to delete more than 50% of items in the

middle, the stream starts to fragment, being composed of "macro nodes". Entries are just flagged as deleted, but are only reclaimed once all the entries in a given macro node are freed. So your mass deletions will change the memory behavior of streams.

Right now, this looks like a non issue, since I don't expect users to delete most history in a stream. However it is possible that in the future we may want to introduce garbage collection: the macro node could be compacted once the ratio between the deleted entries and the existing entries reach a given level. Moreover nearby nodes may be glued together after the garbage collection. I was kind of afraid that later I would no longer remember what were the entry points to do the garbage collection, so I put TODO comments, and even wrote the trigger condition.

This is probably not great. A better idea was instead to write, in the design comment at the top of the file, why we are currently not performing GC. And what are the entry points for GC, if we want to add it later.

FIXME, TODO, XXX, "This is a hack", are all forms of debt comments. They are not great in general, I try to avoid them, but it's not always possible, and sometimes instead of forgetting forever about a problem, I prefer to put a note inside the source code. At least one should periodically grep for such comments, and see if it is possible to put the notes in a better place, or if the problem is no longer relevant or could be fixed right away.

## BACKUP COMMENTS

Finally backup comments are the ones where the developer comments older versions of some code block or even a whole function, because she or he is insecure about the change that was operated in the new one. What is puzzling is that this still happens now that we have Git. I guess people have an uneasy feeling about losing that code fragment, considered more sane or stable, in some years old commit.

But source code is not for making backups. If you want to save an older version of a function or code part, your work is not finished and cannot be committed. Either make sure the new function is better than the past one, or take it just in your development tree until you are sure.

Backup comments end my classification. Let's try some conclusion.

# Comments as an analysis tool.

Comments are rubber duck debugging on steroids, except you are not talking with a rubber duck, but with the future reader of the code, which is more intimidating than a rubber duck, and can use Twitter. So in the process you really try to understand if what you are stating *is acceptable*, honorable, good enough. And if it is not, you make your homework, and come up with something more decent.


It is the same process that happens while writing documentation: the writer attempts to provide the gist of what a given piece of code does, what are the guarantees, the side effects. This is often a bug hunting opportunity. It is very easy while describing something to find that it has holes... You can't really describe it all because you are not sure about a given behavior: such behavior is just emerging from complexity, at random. You really don't want that, so you go back and fix it all. I find this a splendid reason to write comments.

# Writing good comments is harder than writing good code

You may think that writing comments is a lesser noble form of work. After all you *\*can code\**! However consider this: code is a set of statement and function calls, or whatever your programming paradigm is. Sometimes such statements do not make much sense, honestly, if the code is not good. Comments require always to have some design process ongoing, and to understand the code you are writing in a deeper sense. On top of that, in order to write good comments, you have to develop your writing skills. The same writing skills will assist you writing emails, documentation, design documents, blog posts, and commit messages.

I write code because I have an urgent sense to share and communicate more than anything else. Comments coadiuvate the code, assist it, describe our efforts, and after all I love writing them as much as I love writing code itself.

(Thanks to Michel Martens for giving feedbacks during the writing of this blog post)

 Dear reader, the first six chapters of my AI sci-fi novel, WOHPE, are now available as a free eBook. [Click here to get it.](#)