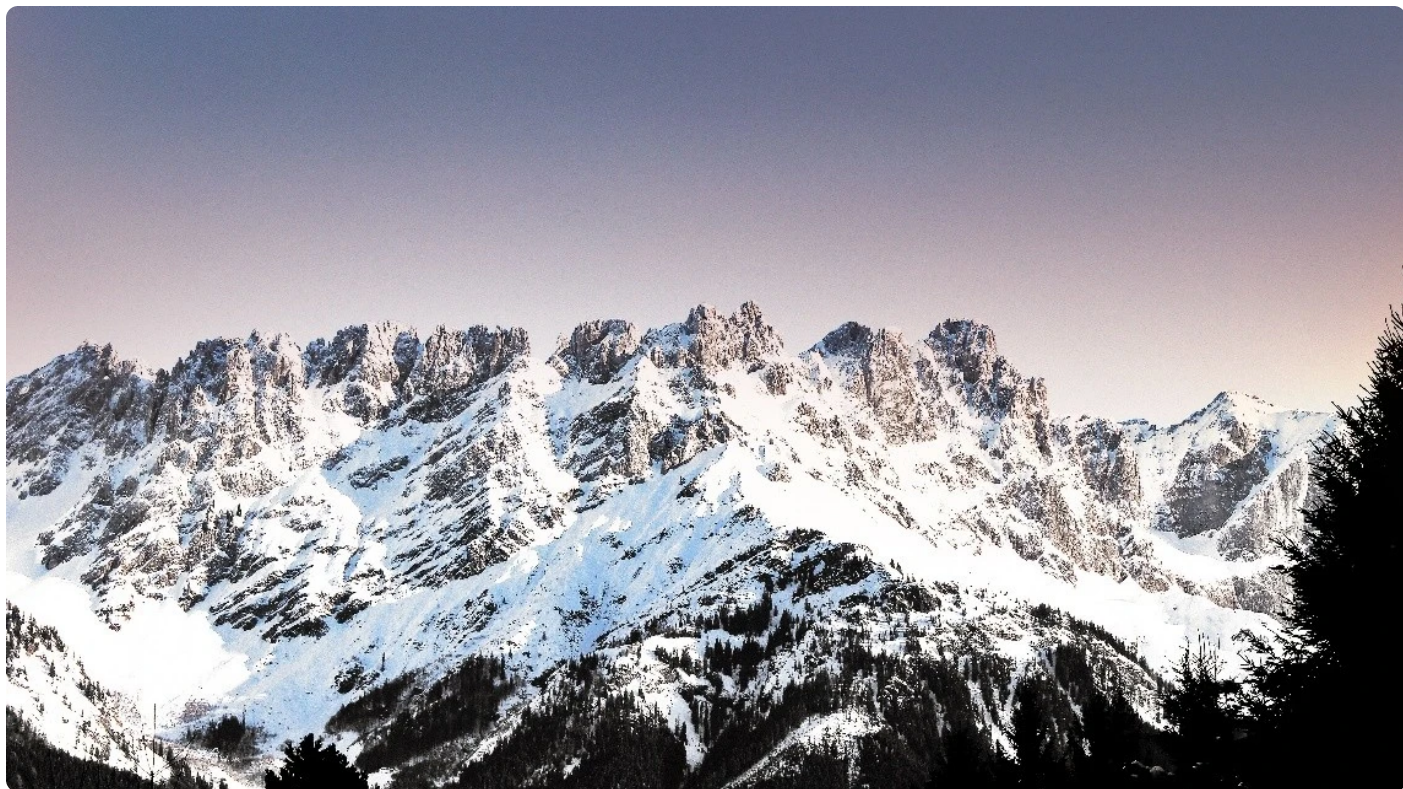


答疑2 | 第7~12讲课后思考题答案及常见问题答疑

2021-10-28 蒋德钧

《Redis源码剖析与实战》

[课程介绍 >](#)



讲述：蒋德钧

时长 15:08 大小 13.87M



你好，我是蒋德钧。

在今天的答疑中，我除了会解答课程之前的思考题以外，还会带你再进一步了解和数据结构设计、进程管理、高性能线程模型、IO 复用、预定义宏等相关的开发知识。希望你能通过这节课的答疑，进一步扩展你的知识面。

🔗 第 7 讲

问题：作为有序索引，Radix Tree 也能提供范围查询，那么与我们日常使用的 B+ 树，以及 🔗 第 5 讲中介绍的跳表相比，你觉得 Radix Tree 有什么优势和不足吗？

对于这道题，有不少同学比如 @Kaito、@曾轼麟等同学，都对 Radix Tree、B+ 树和跳表做了对比，这里我就来总结一下。

Radix Tree 的优势


- 因为 Radix Tree 是前缀树，所以，当保存数据的 key 具有相同前缀时，Radix Tree 会在不同的 key 间共享这些前缀，这样一来，和 B+ 树、跳表相比，就节省内存空间。
- Radix Tree 在查询单个 key 时，其查询复杂度 $O(K)$ 只和 key 的长度 k 有关，和现存的总数据量无关。而 B+ 树、跳表的查询复杂度和数据规模有关，所以 Radix Tree 查询单个 key 的效率要高于 B+ 树、跳表。
- Radix Tree 适合保存大量具有相同前缀的数据。比如一个典型场景，就是 Linux 内核中的 page cache，使用了 Radix Tree 保存文件内部偏移位置和缓存页的对应关系，其中树上的 key 就是文件中的偏移值。

Radix Tree 的不足

- 一般在实现 Radix Tree 时，每个叶子节点就保存一个 key，它的范围查询性能没有 B+ 树和跳表好。这是因为 B+ 树，它的叶子节点可以保存多个 key，而对于跳表来说，它可以遍历有序链表。因此，它们可以更快地支持范围查询。
- Radix Tree 的原理较为复杂，实现复杂度要高于 B+ 树和跳表。

🔗 第 8 讲

问题：Redis 源码的 main 函数在调用 initServer 函数之前，会执行如下的代码片段，你知道这个代码片段的作用是什么吗？

 复制代码

```
1 int main(int argc, char **argv) {
2     ...
3     server.supervised = redisIsSupervised(server.supervised_mode);
4     int background = server.daemonize && !server.supervised;
5     if (background) daemonize();
6     ...
7 }
```

这段代码的目的呢，是先检查 Redis 是否设置成让 upstart 或 systemd 这样的系统管理工具，来启停 Redis。这是由 **redisIsSupervised 函数**，来检查 redis.conf 配置文件中的 supervised 选项，而 supervised 选项的可用配置值，包括 no、upstart、systemd、auto，其中 no 就表示不用系统管理工具来启停 Redis，其他选项会用系统管理工具。

而如果 Redis 没有设置用系统管理工具，同时又设置了使用守护进程方式（对应配置项 `daemonize=yes`，`server.daemonize` 值为 1），那么，这段代码就调用 **daemonize 函数** 以守护进程的方式，启动 Redis。

🔗第 9 讲

问题：在 Redis 事件驱动框架代码中，分别使用了 Linux 系统上的 select 和 epoll 两种机制，你知道为什么 Redis 没有使用 poll 这一机制吗？

这道题呢，主要是希望你对 select 和 poll 这两个 IO 多路复用机制，有进一步的了解。课程的留言区中有不少同学也都回答正确了，我在这里说下我的答案。

select 机制的本质，是**阻塞式监听存放文件描述符的集合**，当监测到有描述符就绪时，select 会结束监测返回就绪的描述符个数。而 select 机制的不足有两个：一是它对单个进程能监听的描述符数量是有限制的，默认是 1024 个；二是即使 select 监测到有文件描述符就绪，程序还是需要线性扫描描述符集合，才能知道具体是哪些文件描述符就绪了。

而 poll 机制相比于 select 机制，本质其实没有太大区别，它只是把 select 机制中文件描述符数量的限制给取消了，允许进程一次监听超过 1024 个描述符。**在线性扫描描述符集合获得就绪的具体描述符这个操作上，poll 并没有优化改进**。所以，poll 相比 select 改进比较有限。而且，就像 @可怜大灰狼、@Kaito 等同学提到的，select 机制的兼容性好，可以在 Linux 和 Windows 上使用。

也正是因为 poll 机制改进有限，而且它对运行平台的支持度不及 select，所以 Redis 的事件驱动框架就没有使用 poll 机制。在 Linux 上，事件驱动框架直接使用了 epoll，而在 Windows 上，框架则使用的是 select。

不过，这里你也要注意的，Redis 的 `ae.c` 文件实现了 **aeWait 函数**，这个函数实际会使用 poll 机制来监测文件描述符。而 `aeWait` 函数会被 `rewriteAppendOnlyFile` 函数（在 `aof.c` 文件中）和 `migrateGetSocket` 函数（在 `cluster.c` 文件中）调用。@可怜大灰狼同学在回答思考题时，就提到了这一点。

此外，在解答这道题的时候，@Darren、@陌等同学还进一步回答了 epoll 机制的实现细节，我在这里也简单总结下他们的答案，分享给你。

当某进程调用 `epoll_create` 方法时，Linux 内核会创建一个 `eventpoll` 结构体，这个结构体中包含了一个红黑树 `rbr` 和一个双链表 `rdlist`，如下所示：

 复制代码

```
1 struct eventpoll{
2     //红黑树的根节点，树中存放着所有添加到epoll中需要监控的描述符
3     struct rb_root rbr;
4     //双链表中存放着已经就绪的描述符，会通过epoll_wait返回给调用程序
5     struct list_head rdlist;
6     ...
7 }
```

`epoll_create` 创建了 `eventpoll` 结构体后，程序调用 `epoll_ctl` 函数，添加要监听的文件描述符时，这些描述符会被保存在红黑树上。

同时，当有描述符上有事件发生时，一个名为 `ep_poll_callback` 的函数会被调用。这个函数会把就绪的描述符添加到 `rdlist` 链表中。而 `epoll_wait` 函数，会检查 `rdlist` 链表中是否有描述符添加进来。如果 `rdlist` 链表不为空，那么 `epoll_wait` 函数，就会把就绪的描述符返回给调用程序了。

🔗第 10 讲

问题：这节课我们学习了 Reactor 模型，除了 Redis，你还了解什么软件系统使用了 Reactor 模型吗？

对于这道题，不少同学都给出了使用 Reactor 模型的其他软件系统，比如 @Darren、@Kaito、@曾轼麟、@结冰的水滴等同学。那么，使用 Reator 模型的常见软件系统，实际上还包括 Netty、Nginx、Memcached、Kafka 等等。

在解答这道题的时候，我看到 @Darren 同学做了很好的扩展，回答了 Reactor 模型的三种类型。在这里，我也总结下分享给你。

- **类型一：单 reactor 单线程**

在这个类型中，Reactor 模型中的 reactor、acceptor 和 handler 的功能都是由一个线程来执行的。reactor 负责监听客户端事件，一旦有连接事件发生，它会分发给 acceptor，由 acceptor 负责建立连接，然后创建一个 handler，负责处理连接建立后的事件。如果是有非

连接的读写事件发生，reactor 将事件分发给 handler 进行处理。handler 负责读取客户端请求，进行业务处理，并最终给客户端返回结果。Redis 就是典型的单 reactor 单线程类型。

- **类型二：单 reactor 多线程**

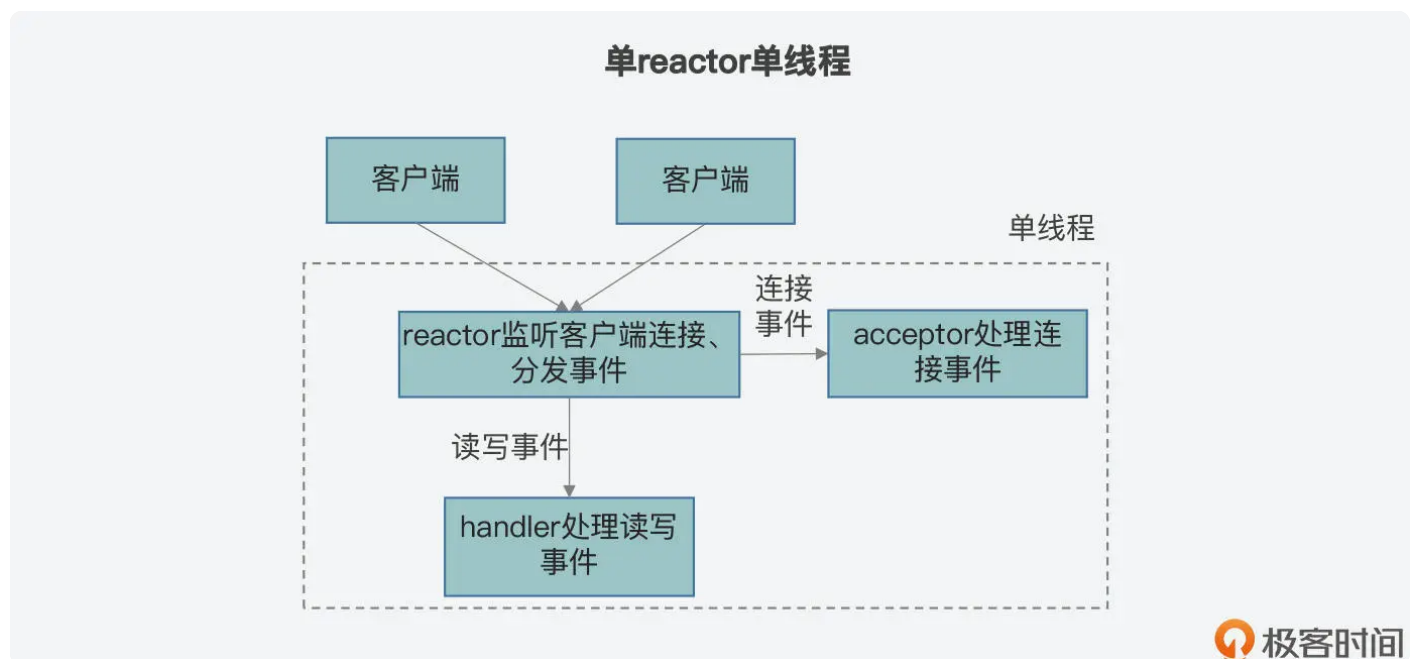
在这个类型中，reactor、acceptor 和 handler 的功能由一个线程来执行，与此同时，会有一个线程池，由若干 worker 线程组成。在监听客户端事件、连接事件处理方面，这个类型和单 reactor 单线程是相同的，但是不同之处在于，在单 reactor 多线程类型中，handler 只负责读取请求和写回结果，而具体的业务处理由 worker 线程来完成。

- **类型三：主 - 从 Reactor 多线程**

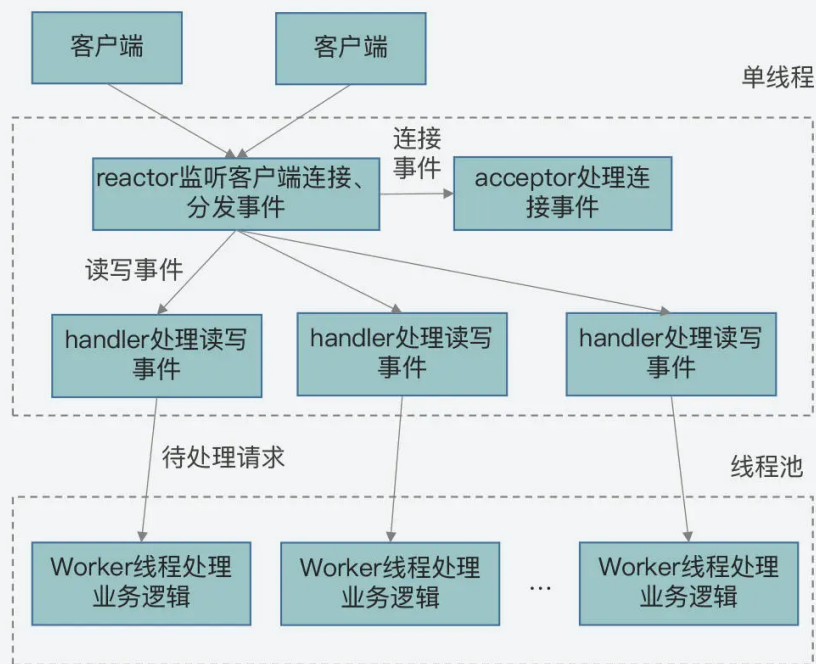
在这个类型中，会有一个主 reactor 线程、多个子 reactor 线程和多个 worker 线程组成的一个线程池。其中，主 reactor 负责监听客户端事件，并在同一个线程中让 acceptor 处理连接事件。一旦连接建立后，主 reactor 会把连接分发给子 reactor 线程，由子 reactor 负责这个连接上的后续事件处理。

那么，子 reactor 会监听客户端连接上的后续事件，有读写事件发生时，它会让在同一个线程中的 handler 读取请求和返回结果，而和单 reactor 多线程类似，具体业务处理，它还是会让线程池中的 worker 线程处理。刚才介绍的 Netty 使用的就是这个类型。

我在下面画了三张图，展示了刚才介绍的三个类型的区别，你可以再整体回顾下。

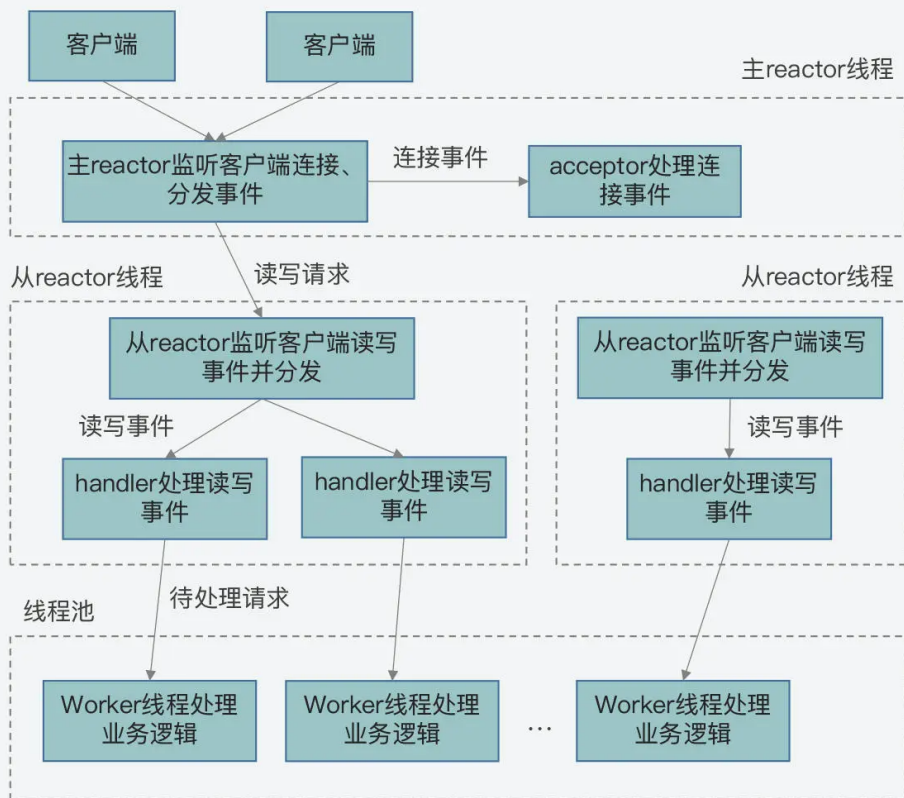


单reactor多线程



极客时间

主从reactor多线程



极客时间

第 11 讲

问题：已知，Redis 事件驱动框架的 aeApiCreate、aeApiAddEvent 等等这些函数，是对操作系统提供的 IO 多路复用函数进行了封装，具体的 IO 多路复用函数分别是在

[ae_epoll.c](#), [ae_evport.c](#), [ae_kqueue.c](#), [ae_select.c](#)四个代码文件中定义的。那么你知道，Redis 在调用 `aeApiCreate`、`aeApiAddEvent` 这些函数时，是根据什么条件来决定，具体调用哪个文件中的 IO 多路复用函数的吗？

其实，这道题的目的，主要是希望你能通过它进一步了解如何进行跨平台的编程开发。在实际业务场景中，我们开发的系统可能需要在不同的平台上运行，比如 Linux 和 Windows。那么，我们在开发时，就需要用同一套代码支持在不同平台上的执行。

就像 Redis 中使用的 IO 多路复用机制一样，不同平台上支持的 IO 多路复用函数是不一样的。但是，使用这些函数的事件驱动整体框架又可以用一套框架来实现。所以，我们就需要在同一套代码中区分底层平台，从而可以正确地使用该平台对应函数。

对应 Redis 事件驱动框架来说，它是用 `aeApiCreate`、`aeApiAddEven` 等函数，封装了不同的 IO 多路复用函数，而在 `ae.c` 文件的开头部分，它使用了 `#ifdef`、`#else`、`#endif` 等**条件编译指令**，来区分封装的函数应该具体使用哪种 IO 多路复用函数。

下面的代码就展示了刚才介绍的条件编译。

 复制代码

```
1  #ifdef HAVE_EVPORT
2  #include "ae_evport.c"
3  #else
4      #ifdef HAVE_EPOLL
5      #include "ae_epoll.c"
6      #else
7          #ifdef HAVE_KQUEUE
8          #include "ae_kqueue.c"
9          #else
10         #include "ae_select.c"
11         #endif
12     #endif
13 #endif
```

从这段代码中我们可以看到，如果 `HAVE_EPOLL` 宏被定义了，那么，代码就会包含 `ae_epoll.c` 文件，这也就是说，`aeApiCreate`、`aeApiAddEven`、`aeApiPoll` 这些函数就会调用 `epoll_create`、`epoll_ctl`、`epoll_wait` 这些机制。

类似的，如果 HAVE_KQUEUE 宏被定义了，那么，代码就会包含 ae_kqueue.c 文件，框架函数也会实际调用 kqueue 的机制。

那么，接下来的一个问题就是，HAVE_EPOLL、HAVE_KQUEUE 这些宏又是在哪里被定义的呢？

其实，它们是在 config.h 文件中定义的。

在 config.h 文件中，代码会判断是否定义了 __linux__ 宏，如果有的话，那么，代码就会定义 HAVE_EPOLL 宏。而如果定义了 __FreeBSD__、__OpenBSD__ 等宏，那么代码就会定义 HAVE_KQUEUE 宏。

下面的代码展示了 config.h 中的这部分逻辑，你可以看下。

 复制代码

```
1 #ifndef __linux__
2 #define HAVE_EPOLL 1
3 #endif
4
5 #if (defined(__APPLE__) && defined(MAC_OS_X_VERSION_10_6)) || defined(__FreeBSD__
6 #define HAVE_KQUEUE 1
7 #endif
```

好了，到这里，我们就知道了，Redis 源码中是根据 __linux__、__FreeBSD__、__OpenBSD__ 这些宏，来决定当前的运行平台是哪个平台，然后再设置相应的 IO 多路复用函数的宏。而 __linux__、__FreeBSD__、__OpenBSD__ 这些宏，又是如何定义的呢？

其实，这就和运行平台上的编译器有关了。编译器会根据所运行的平台提供预定义宏。像刚才的 __linux__、__FreeBSD__ 这些都是属于预定义宏，这些预定义宏的名称都是以 “__” 两条下划线开头和结尾的。你在 Linux 的系统中，比如 CentOS 或者 Ubuntu，运行如下所示的 gcc 命令，你就可以看到 Linux 中运行的 gcc 编译器，已经提供了 __linux__ 这个预定义宏了。

 复制代码

```
1 gcc -dM -E -x c /dev/null | grep linux
2
3 #define __linux 1
```



```
4 #define __linux__ 1
5 #define __gnu_linux__ 1
6 #define linux 1
```

而如果你在 macOS 的系统中运行如下所示的 gcc 命令，你也能看到 macOS 中运行的 gcc 编译器，已经提供了 __APPLE__ 预定义宏。

 复制代码

```
1 gcc -dM -E -x c /dev/null | grep APPLE
2 #define __APPLE__ 1
```

这样一来，当我们在某个系统平台上使用 gcc 编译 Redis 源码时，就可以根据编译器提供的预定义宏，来决定当前究竟该使用哪个 IO 多路复用函数了。而此时使用的 IO 多路复用函数，也是和 Redis 当前运行的平台是匹配的。

🔗 第 12 讲

问题：Redis 后台任务使用了 bio_job 结构体来描述，该结构体用了三个指针变量来表示任务参数，如下所示。那么，如果你创建的任务所需要的参数大于 3 个，你有什么应对方法来传参吗？

 复制代码

```
1 struct bio_job {
2     time_t time;
3     void *arg1, *arg2, *arg3; //传递给任务的参数
4 };
```

这道题其实是需要你了解在 C 函数开发时，如果想传递很多参数该如何处理。

其实，这里我们可以充分利用函数参数中的指针，让指针指向一个结构体，比如数组或哈希表。而数组或哈希表这样的结构体中，就可以保存很多参数了。这样一来，我们就可以通过指针指向结构体来传递多个参数了。

不过，你要注意的是，在函数使用参数时，还需要解析指针指向的结构体，这个会产生一些开销。

小结

这节课，我们解答了第 7 讲到第 12 讲的课后思考题。在设计这些思考题时，有些题我希望你能通过它们了解一些 C 语言编程开发的技巧，比如使用编译器提供的预定义宏实现跨平台的开发，或者是通过指针给 C 函数传递批量参数等。而有些题，我是希望你能对计算机系统的一些关键机制设计有更多的了解，比如 IO 多路复用机制的对比。

其实，在回答这些思考题的时候，你有没有感受到，Redis 就像一个小宝藏一样，我们可以从中学到从编程开发、到系统管理、再到系统设计等多方面的知识。希望你能在阅读 Redis 源码的道路上充分挖掘这个宝藏，充实自己的知识财富。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 答疑1 | 第1~6讲课后思考题答案及常见问题答疑

[下一篇](#) 答疑3 | 第13~18讲课后思考题答案及常见问题答疑

更多学习推荐

最新 Java 面试加油包 重磅上线! 限时免费




6 家大厂面试常考题

4 位资深专家视频课

15 个 Java 核心技术点

100 道算法必会题+详细解析

免费去领 

686 道 Java 面试高频题+详细解析

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。