03-高性能IO模型：为什么单线程Redis能那么快？

你好，我是蒋德钧。

今天，我们来探讨一个很多人都很关心的问题：“为什么单线程的Redis能那么快？”

首先，我要和你厘清一个事实，我们通常说，Redis是单线程，主要是指**Redis的网络IO和键值对读写是由一个线程来完成的，这也是Redis对外提供键值存储服务的主要流程**。但Redis的其他功能，比如持久化、异步删除、集群数据同步等，其实是由额外的线程执行的。

所以，严格来说，Redis并不是单线程，但是我们一般把Redis称为单线程高性能，这样显得“酷”些。接下来，我也会把Redis称为单线程模式。而且，这也会促使你紧接着提问：“为什么用单线程？为什么单线程能这么快？”

要弄明白这个问题，我们就要深入地学习下Redis的单线程设计机制以及多路复用机制。之后你在调优Redis性能时，也能更有针对性地避免会导致Redis单线程阻塞的操作，例如执行复杂度高的命令。

好了，话不多说，接下来，我们就先来学习下Redis采用单线程的原因。

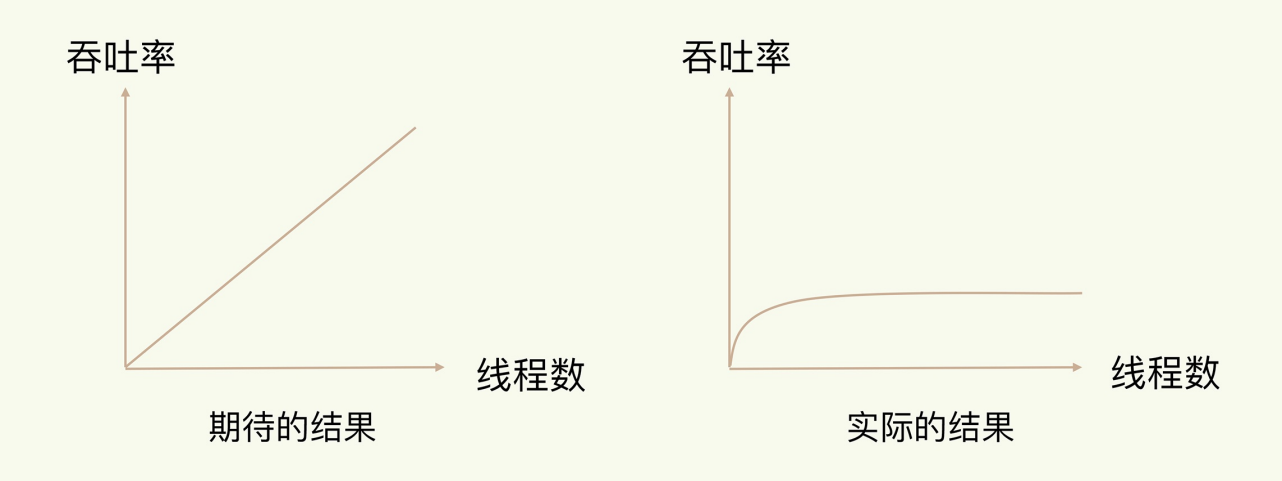
Redis为什么用单线程？

要更好地理解Redis为什么用单线程，我们就要先了解多线程的开销。

多线程的开销

日常写程序时，我们经常会听到一种说法：“使用多线程，可以增加系统吞吐率，或是可以增加系统扩展性。”的确，对于一个多线程的系统来说，在有合理的资源分配的情况下，可以增加系统中处理请求操作的资源实体，进而提升系统能够同时处理的请求数，即吞吐率。下面的左图是我们采用多线程时所期待的结果。

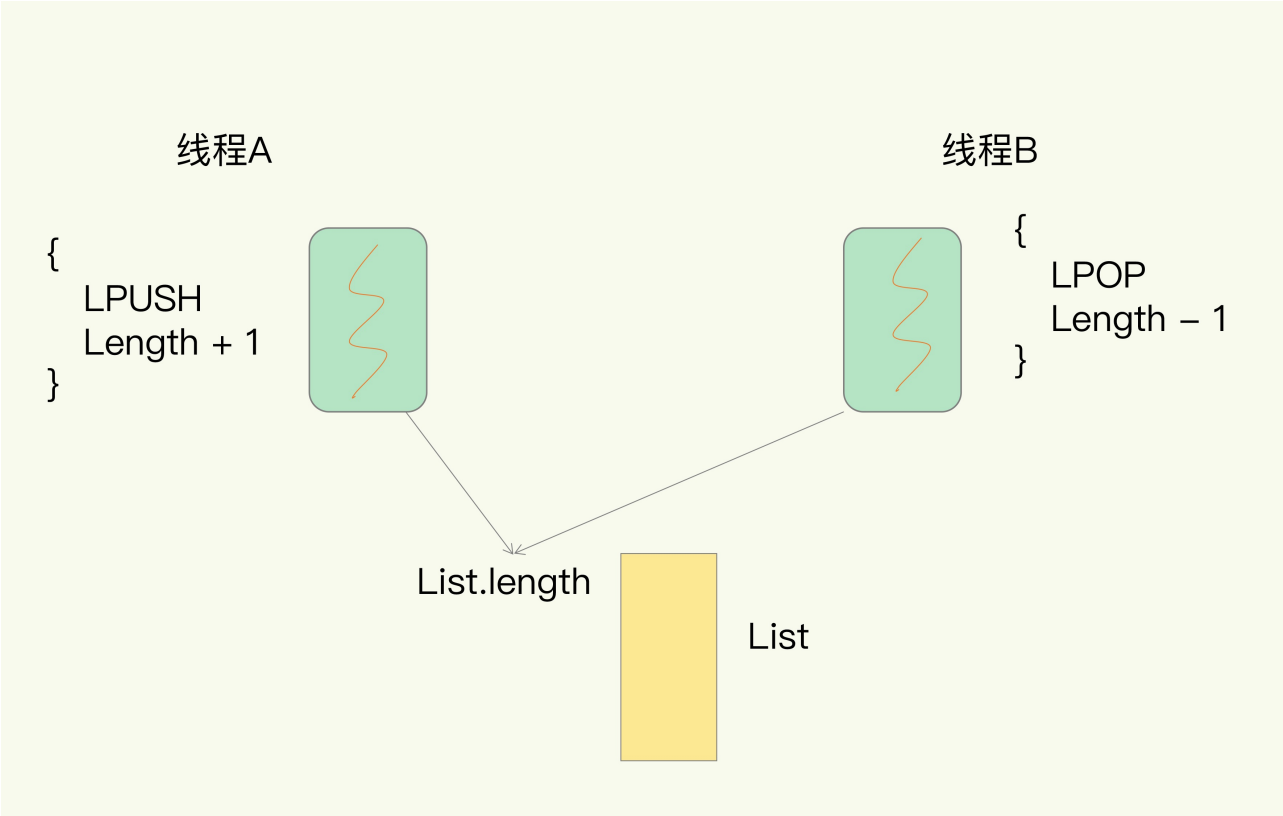
但是，请你注意，通常情况下，在我们采用多线程后，如果没有良好的系统设计，实际得到的结果，其实是右图所展示的那样。我们刚开始增加线程数时，系统吞吐率会增加，但是，再进一步增加线程时，系统吞吐率就增长迟缓了，有时甚至还会出现下降的情况。



为什么会出现这种情况呢？一个关键的瓶颈在于，系统中通常会存在被多线程同时访问的共享资源，比如一个共享的数据结构。当有多个线程要修改这个共享资源时，为了保证共享资源的正确性，就需要有额外的机

制进行保证，而这个额外的机制，就会带来额外的开销。

拿Redis来说，在上节课中，我提到过，Redis有List的数据类型，并提供出队（LPOP）和入队（LPUSH）操作。假设Redis采用多线程设计，如下图所示，现在有两个线程A和B，线程A对一个List做LPUSH操作，并对队列长度加1。同时，线程B对该List执行LPOP操作，并对队列长度减1。为了保证队列长度的正确性，Redis需要让线程A和B的LPUSH和LPOP串行执行，这样一来，Redis可以无误地记录它们对List长度的修改。否则，我们可能就会得到错误的长度结果。这就是**多线程编程模式面临的共享资源的并发访问控制问题**。



并发访问控制一直是多线程开发中的一个难点问题，如果没有精细的设计，比如说，只是简单地采用一个粗粒度互斥锁，就会出现不理想的结果：即使增加了线程，大部分线程也在等待获取访问共享资源的互斥锁，并行变串行，系统吞吐率并没有随着线程的增加而增加。

而且，采用多线程开发一般会引入同步原语来保护共享资源的并发访问，这也会降低系统代码的易调试性和可维护性。为了避免这些问题，Redis直接采用了单线程模式。

讲到这里，你应该已经明白了“Redis为什么用单线程”，那么，接下来，我们就来看看，为什么单线程Redis能获得高性能。

单线程Redis为什么那么快?

通常来说，单线程的处理能力要比多线程差很多，但是Redis却能使用单线程模型达到每秒数十万级别的处理能力，这是为什么呢？其实，这是Redis多方面设计选择的一个综合结果。

一方面，Redis的大部分操作在内存上完成，再加上它采用了高效的数据结构，例如哈希表和跳表，这是它实现高性能的一个重要原因。另一方面，就是Redis采用了**多路复用机制**，使其在网络IO操作中能并发处理大量的客户端请求，实现高吞吐率。接下来，我们就重点学习下多路复用机制。

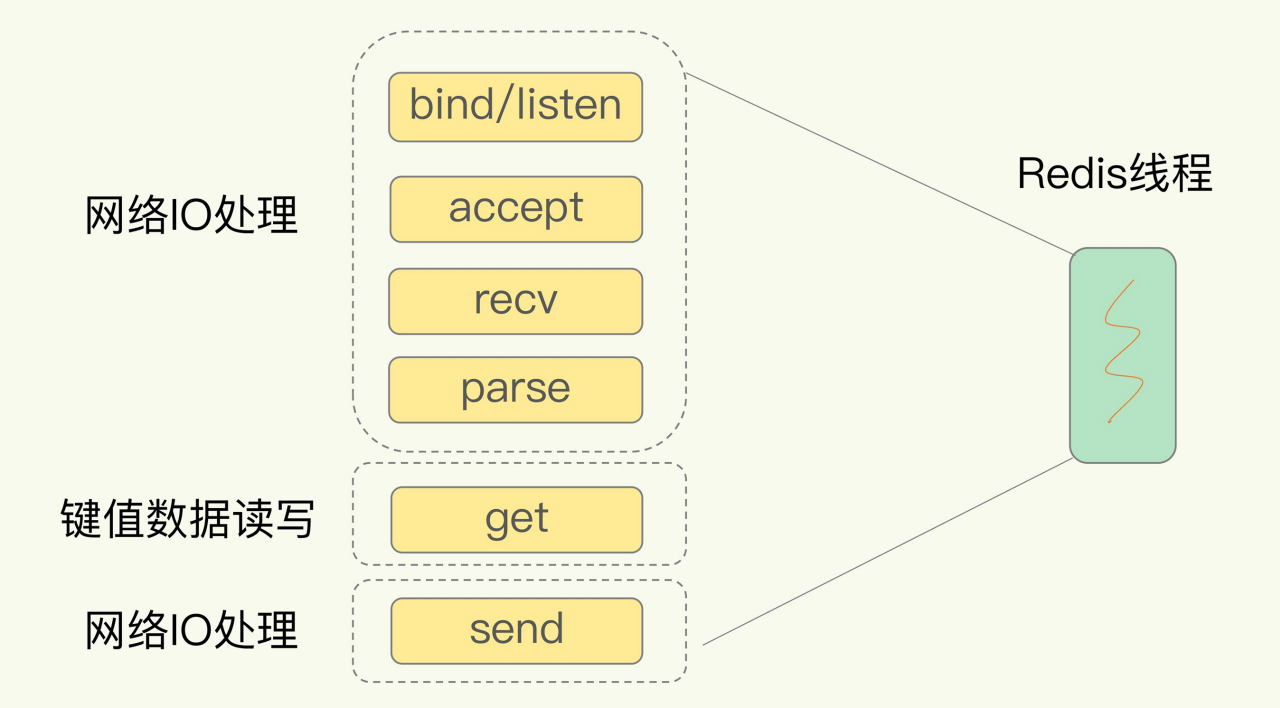
首先，我们要弄明白网络操作的基本IO模型和潜在的阻塞点。毕竟，Redis采用单线程进行IO，如果线程被阻塞了，就无法进行多路复用了。

基本IO模型与阻塞点

你还记得我在第一节课介绍的具有网络框架的SimpleKV吗？

以Get请求为例，SimpleKV为了处理一个Get请求，需要监听客户端请求（bind/listen），和客户端建立连接（accept），从socket中读取请求（recv），解析客户端发送请求（parse），根据请求类型读取键值数据（get），最后给客户端返回结果，即向socket中写回数据（send）。

下图显示了这一过程，其中，bind/listen、accept、recv、parse和send属于网络IO处理，而get属于键值数据操作。既然Redis是单线程，那么，最基本的一种实现是在一个线程中依次执行上面说的这些操作。



但是，在这里的网络IO操作中，有潜在的阻塞点，分别是accept()和recv()。当Redis监听到一个客户端有连接请求，但一直未能成功建立起连接时，会阻塞在accept()函数这里，导致其他客户端无法和Redis建立连接。类似的，当Redis通过recv()从一个客户端读取数据时，如果数据一直没有到达，Redis也会一直阻塞在recv()。

这就导致Redis整个线程阻塞，无法处理其他客户端请求，效率很低。不过，幸运的是，socket网络模型本身支持非阻塞模式。

非阻塞模式

Socket网络模型的非阻塞模式设置，主要体现在三个关键的函数调用上，如果想要使用socket非阻塞模式，就必须要了解这三个函数的调用返回类型和设置模式。接下来，我们就重点学习下它们。

在socket模型中，不同操作调用后会返回不同的套接字类型。socket()方法会返回主动套接字，然后调用listen()方法，将主动套接字转化为监听套接字，此时，可以监听来自客户端的连接请求。最后，调用accept()方法接收到达的客户端连接，并返回已连接套接字。

调用方法	返回套接字类型	非阻塞模式	效果
socket()	主动套接字		
listen()	监听套接字	可设置	accept()非阻塞
accept()	已连接套接字	可设置	send()/recv()非阻塞

针对监听套接字，我们可以设置非阻塞模式：当Redis调用accept()但一直未有连接请求到达时，Redis线程可以返回处理其他操作，而不用一直等待。但是，你要注意的是，调用accept()时，已经存在监听套接字了。

虽然Redis线程可以不用继续等待，但是总得有机制继续在监听套接字上等待后续连接请求，并在有请求时通知Redis。

类似的，我们也可以针对已连接套接字设置非阻塞模式：Redis调用recv()后，如果已连接套接字上一直没有数据到达，Redis线程同样可以返回处理其他操作。我们也需要有机制继续监听该已连接套接字，并在有数据达到时通知Redis。

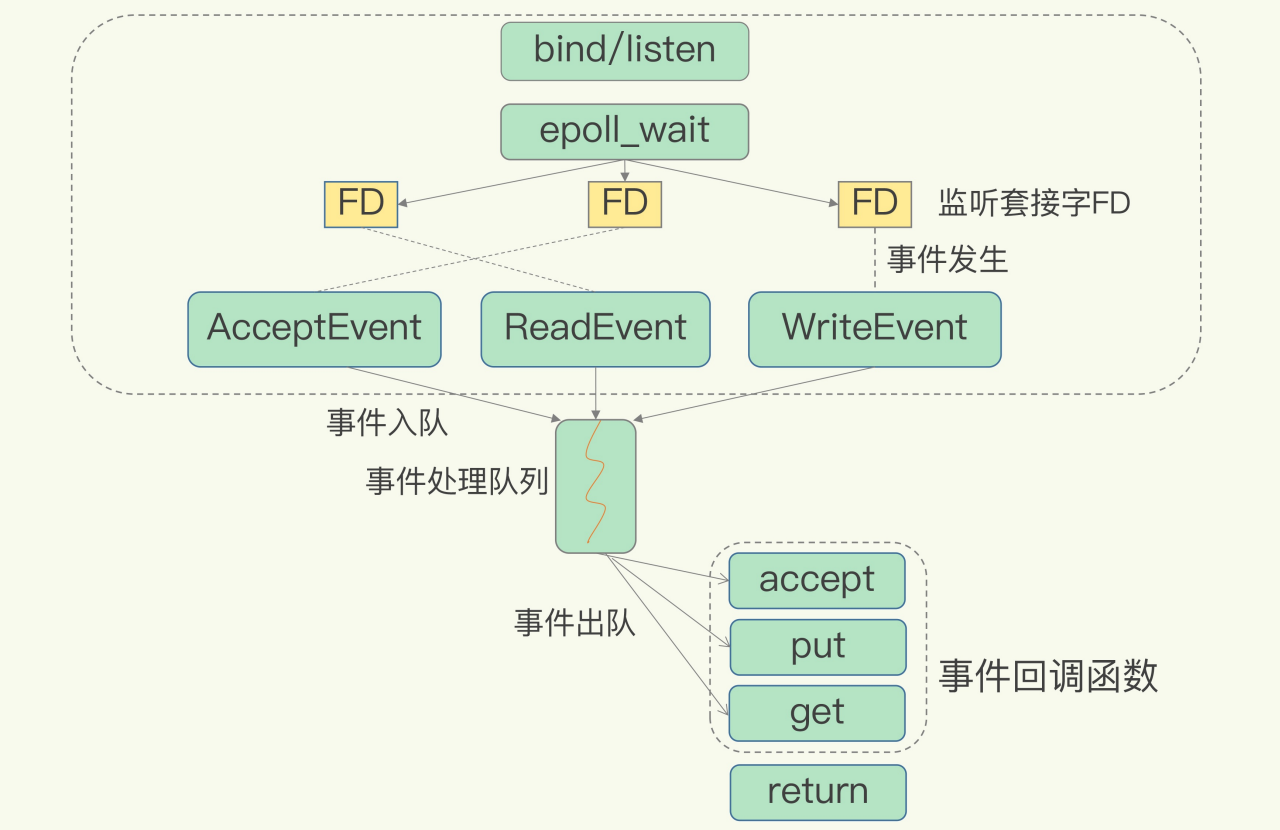
这样才能保证Redis线程，既不会像基本IO模型中一直在阻塞点等待，也不会导致Redis无法处理实际到达的连接请求或数据。

到此，Linux中的IO多路复用机制就要登场了。

基于多路复用的高性能I/O模型

Linux中的IO多路复用机制是指一个线程处理多个IO流，就是我们经常听到的select/epoll机制。简单来说，在Redis只运行单线程的情况下，**该机制允许内核中，同时存在多个监听套接字和已连接套接字**。内核会一直监听这些套接字上的连接请求或数据请求。一旦有请求到达，就会交给Redis线程处理，这就实现了一个Redis线程处理多个IO流的效果。

下图就是基于多路复用的Redis IO模型。图中的多个FD就是刚才所说的多个套接字。Redis网络框架调用epoll机制，让内核监听这些套接字。此时，Redis线程不会阻塞在某一个特定的监听或已连接套接字上，也就是说，不会阻塞在某一个特定的客户端请求处理上。正因为此，Redis可以同时和多个客户端连接并处理请求，从而提升并发性。



为了在请求到达时能通知到Redis线程，select/epoll提供了**基于事件的回调机制**，即**针对不同事件的发生，调用相应的处理函数**。

那么，回调机制是怎么工作的呢？其实，select/epoll一旦监测到FD上有请求到达时，就会触发相应的事件。

这些事件会被放进一个事件队列，Redis单线程对该事件队列不断进行处理。这样一来，Redis无需一直轮询是否有请求实际发生，这就可以避免造成CPU资源浪费。同时，Redis在对事件队列中的事件进行处理时，会调用相应的处理函数，这就实现了基于事件的回调。因为Redis一直在对事件队列进行处理，所以能及时响应客户端请求，提升Redis的响应性能。

为了方便你理解，我再以连接请求和读数据请求为例，具体解释一下。

这两个请求分别对应Accept事件和Read事件，Redis分别对这两个事件注册accept和get回调函数。当Linux内核监听到有连接请求或读数据请求时，就会触发Accept事件和Read事件，此时，内核就会回调Redis相应的accept和get函数进行处理。

这就像病人去医院瞧病。在医生实际诊断前，每个病人（等同于请求）都需要先分诊、测体温、登记等。如果这些工作都由医生来完成，医生的工作效率就会很低。所以，医院都设置了分诊台，分诊台会一直处理这些诊断前的工作（类似于Linux内核监听请求），然后再转交给医生做实际诊断。这样即使一个医生（相当于Redis单线程），效率也能提升。

不过，需要注意的是，即使你的应用场景中部署了不同的操作系统，多路复用机制也是适用的。因为这个机制的实现有很多种，既有基于Linux系统下的select和epoll实现，也有基于FreeBSD的kqueue实现，以及基于Solaris的evport实现，这样，你可以根据Redis实际运行的操作系统，选择相应的多路复用实现。

小结

今天，我们重点学习了Redis线程的三个问题：“Redis真的只有单线程吗？”“为什么用单线程？”“单线程为什么这么快？”

现在，我们知道了，Redis单线程是指它对网络IO和数据读写的操作采用了一个线程，而采用单线程的一个核心原因是避免多线程开发的并发控制问题。单线程的Redis也能获得高性能，跟多路复用的IO模型密切相关，因为这避免了accept()和send()/recv()潜在的网络IO操作阻塞点。

搞懂了这些，你就走在了很多人的前面。如果你身边还有不清楚这几个问题的朋友，欢迎你分享给他/她，解决他们的困惑。

另外，我也剧透下，可能你也注意到了，2020年5月，Redis 6.0的稳定版发布了，Redis 6.0中提出了多线程模型。那么，这个多线程模型和这节课所说的IO模型有什么关联？会引入复杂的并发控制问题吗？会给Redis 6.0带来多大提升？关于这些问题，我会在后面的课程中和你具体介绍。

每课一问

这节课，我给你提个小问题，在“Redis基本IO模型”图中，你觉得还有哪些潜在的性能瓶颈吗？欢迎在留言区写下你的思考和答案，我们一起交流讨论。

精选留言：

- Kaito 2020-08-10 11:37:32
  Redis单线程处理IO请求性能瓶颈主要包括2个方面：

  1、任意一个请求在server中一旦发生耗时，都会影响整个server的性能，也就是说后面的请求都要等前面这个耗时请求处理完成，自己才能被处理到。耗时的操作包括以下几种：
  a、操作bigkey：写入一个bigkey在分配内存时需要消耗更多的时间，同样，删除bigkey释放内存同样会产生耗时；
  b、使用复杂度过高的命令：例如SORT/SUNION/ZUNIONSTORE，或者O(N)命令，但是N很大，例如lrange key 0 -1一次查询全量数据；
  c、大量key集中过期：Redis的过期机制也是在主线程中执行的，大量key集中过期会导致处理一个请求时，耗时都在删除过期key，耗时变长；
  d、淘汰策略：淘汰策略也是在主线程执行的，当内存超过Redis内存上限后，每次写入都需要淘汰一些key，也会造成耗时变长；
  e、AOF刷盘开启always机制：每次写入都需要把这个操作刷到磁盘，写磁盘的速度远比写内存慢，会拖慢Redis的性能；
  f、主从全量同步生成RDB：虽然采用fork子进程生成数据快照，但fork这一瞬间也是会阻塞整个线程的，实例越大，阻塞时间越久；
  2、并发量非常大时，单线程读写客户端IO数据存在性能瓶颈，虽然采用IO多路复用机制，但是读写客户端数据依旧是同步IO，只能单线程依次读取客户端的数据，无法利用到CPU多核。

  针对问题1，一方面需要业务人员去规避，一方面Redis在4.0推出了lazy-free机制，把bigkey释放内存的耗时操作放在了异步线程中执行，降低对主线程的影响。

  针对问题2，Redis在6.0推出了多线程，可以在高并发场景下利用CPU多核多线程读写客户端数据，进一步提升server性能，当然，只是针对客户端的读写是并行的，每个命令的真正操作依旧是单线程的。 [23赞]

- Darren 2020-08-10 09:15:17

1.big key的操作。
2.潜在的大量数据操作，比如 key *或者get all之类的操作，所以才引入了scan的相关操作。
3.特殊的场景，大量的客户端接入。

简单介绍下select poll epoll的区别，select和poll本质上没啥区别，就是文件描述符数量的限制，select根据不同的系统，文件描述符限制为1024或者2048，poll没有数量限制。他两都是把文件描述符集合保存在用户态，每次把集合传入内核态，内核态返回ready的文件描述符。
epoll是通过epoll_create和epoll_ctl和epoll_await三个系统调用完成的，每当接入一个文件描述符，通过ctl添加到内核维护的红黑树中，通过事件机制，当数据ready后，从红黑树移动到链表，通过await获取链表中准备好数据的fd，程序去处理。 [10赞]

- 曾轼麟 2020-08-10 14:12:52
  虽然单线程很快，没有锁的单线程更快借助CPU的多级缓存可以把性能发挥到最大。但是随着访问量的增加，以及数据量的增加，IO的写入写出会成为性能瓶颈。10个socket的IO吞吐处理肯定比1000个socket吞吐处理的快，为了解决这个问题，Redis6引入了IO多线程的方式以及client缓冲区，在实际指令处理还是单线程模式。在IO上变成的了【主线程】带着众多【IO线程】进行IO，IO线程听从主线程的指挥是写入还是写出。Read的时候IO线程会和主线程一起读取并且解析命令（RESP协议）存入缓冲区，写的时候会从缓冲区写出到Socket。IO线程听从主线程的指挥，在同一个时间点上主线程和IO线程会一起写出或者读取，并且主线程会等待IO线程的结束。但是这种模式的多线程会面临一给NUMA陷阱的问题，在最近的Redis版本中加强了IO线程和CPU的亲和性解决了这个问题。（不过目前官方在默认情况下并不推荐使用多线程IO模式，需要手动开启） [1赞]

- test 2020-08-10 08:47:29
  单线程同步非阻塞读取网络IO的时候会有性能瓶颈，如果读取的内容过多的时候 [1赞]

- 每天晒白牙 2020-08-10 08:27:39
  理解的深入了 [1赞]

- williamcai 2020-08-11 06:45:42
  事件队列堆积了大量的请求，有些请求非常耗时，单线程处理就会发生性能问题

- 阳阳 2020-08-10 18:46:53
  服务端是非阻塞的，那客户端是否是阻塞的一直等待结果的呢？直到服务端返回结果？

- 末日，成欢 2020-08-10 18:45:08
  调用 accept() 时，已经存在监听套接字了。 如果客户端还没有请求过来， 也会有FD的存在吗？

- yyl 2020-08-10 12:37:44
  1. 对于不同的事件，都是进入 相同的事件处理队列吗？
  2. 事件处理队列是先入先出的，如果队列中请求一旦发生累积，请求的处理延时也会也会随之增大吧？

- 脱缰的野马__ 2020-08-10 12:16:26
  多线程之间的上下文切换也是影响性能的点

- MClink 2020-08-10 11:44:09
  想问个问题，如果我在代码层连接 Redis 时没有设置 timeout 参数，那么如果对应的某个业务进程执行 get/rPop 命令卡在了 revefrom 的系统调用（也就是等待Redis 返回数据），而该Redis还在正常的处理其他的客户端请求（看起来正常，没有挂掉），那么这种原因有可能是因为 Redis 已经正确处理并且返回了

数据，但是客户端没有正常接收到导致的嘛（丢包？），基于 tcp 的 Redis 应该是有重传机制的才对吧，看了很久，如果是 Redis 的内部发生了阻塞，照理说整个事件队列应该也是会被阻塞的才对。希望老师能给点处理该问题的方向。

- 努力努力再努力 2020-08-10 10:04:11
  老师，redis是通过一个线程来负责建立连接，这个线程也负责处理读时间和写事件吗？这种类似单reactor线程模型吗？一直以为redis是和netty类型的，一个线程专门处理连接，然后另外单个线程负责读写事件，然后再调用handler去处理的

- Jackey 2020-08-10 09:50:07
  我认为可能的瓶颈一个是在处理big key时网络传输的速率，再一个是每次都要重新建立连接感觉对性能是有些浪费的，不知道可不可以和一些常用客户端建立长连接

- yyl 2020-08-10 09:05:36
  “Redis 单线程是指它对网络 IO 和数据读写的操作采用了一个线程”
  老师，这句话如下理解是正确否？
  1. 响应redis客户端网络请求的线程、处理事件回调函数的线程、读写全局Hash表的线程 都是 同一个线程
  2. 若Redis采用多线程实现，全局Hash表成为多线程的竞争资源

- 徐鹏 2020-08-10 08:59:59
  在事件回掉函数中处理的数据量太大应该会非常影响性能

- 0bug 2020-08-10 08:10:51
  操作大key的时候，IO是性能瓶颈

- 滴流乱转小胖子 2020-08-10 07:21:16
  老师你好，单线程的处理事件队列中的事件，这样还是会遇到性能瓶颈吧？

- 来碗绿豆汤 2020-08-10 00:31:44
  如果把接收连接请求和接收数据分到两个线程里面是不是更好,毕竟他们两个是干两件事的

- 咸鱼 2020-08-10 00:31:33
  这章让我对IO多路复用的理解又深了些

- 张晗_Jeremy 2020-08-10 00:26:43
  第一时间打卡！