为什么 Redis 选择单线程模型

2019-10-18 Redis 多线程 系统设计 为什么这么设计

> 为什么这么设计（Why's THE Design）是一系列关于计算机领域中程序设计决策的文章，我们在这个系列的每一篇文章中都会提出一个具体的问题并从不同的角度讨论这种设计的优缺点、对具体实现造成的影响。如果你有想要了解的问题，可以在文章下面留言。

Redis 作为广为人知的内存数据库，在玩具项目和复杂的工业级别项目中都看到它的身影，然而 Redis 却是使用单线程模型进行设计的，这与很多人固有的观念有所冲突，为什么单线程的程序能够抗住每秒几百万的请求量呢？这也是我们今天要讨论的问题之一。

除此之外，Redis 4.0 之后的版本却抛弃了单线程模型这一设计，原本使用单线程运行的 Redis 也开始选择性使用多线程模型，这一看似有些矛盾的设计决策是今天需要讨论的另一个问题。

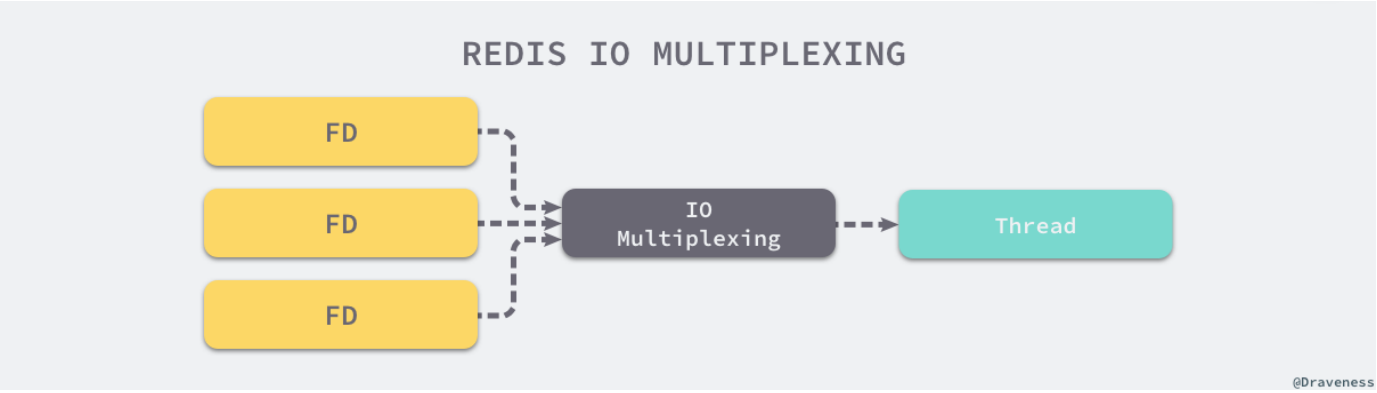
> However with Redis 4.0 we started to make Redis more threaded. For now this is limited to deleting objects in the background, and to blocking commands implemented via Redis modules. For the next releases, the plan is to make Redis more and more threaded.

概述

就像在介绍中说的，这一篇文章想要讨论的两个与 Redis 有关的问题就是：

- 为什么 Redis 在最初的版本中选择单线程模型？
- 为什么 Redis 在 4.0 之后的版本中加入了多线程的支持？

这两个看起来有些矛盾的问题实际上并不冲突，我们会分别阐述对这个看起来完全相反的设计决策作出分析和解释，不过在具体分析它们的设计之前，我们先来看一下不同版本 Redis 顶层的设计：



Redis 作为一个内存服务器，它需要处理很多来自外部的网络请求，它使用 I/O 多路复用机制同时监听多个文件描述符的可读和可写状态，一旦收到网络请求就会在内存中快速处理，由于绝大多数的操作都是纯内存的，所以处理的速度会非常地快。

在 Redis 4.0 之后的版本，情况就有了一些变动，新版的 Redis 服务在执行一些命令时就会使用『主处理线程』之外的其他线程，例如 UNLINK、FLUSHALL ASYNC、FLUSHDB ASYNC 等非阻塞的删除操作。

设计

无论是使用单线程模型还是多线程模型，这两个设计上的决定都是为了更好地提升 Redis 的开发效率、运行性能，想要理解两个看起来矛盾的设计决策，我们首先需要重新梳理做出决定的上下文和大前提，从下面的角度来看，使用单线程模型和多线程模型其实也并不矛盾。

虽然 Redis 在较新的版本中引入了多线程，不过是在部分命令上引入的，其中包括非阻塞的删除操作，在整体的架构设计上，主处理程序还是单线程模型的；由此看来，我们今天想要分析的两个问题可以简化成：

- 为什么 Redis 服务使用单线程模型处理绝大多数的网络请求？
- 为什么 Redis 服务增加了多个非阻塞的删除操作，例如：UNLINK、FLUSHALL ASYNC 和 FLUSHDB ASYNC？

接下来的两个小节将从多个角度分析这两个问题。

单线程模型

Redis 从一开始就选择使用单线程模型处理来自客户端的绝大多数网络请求，这种考虑其实是多方面的，作者分析了相关的资料，发现其中最重要的几个原因如下：

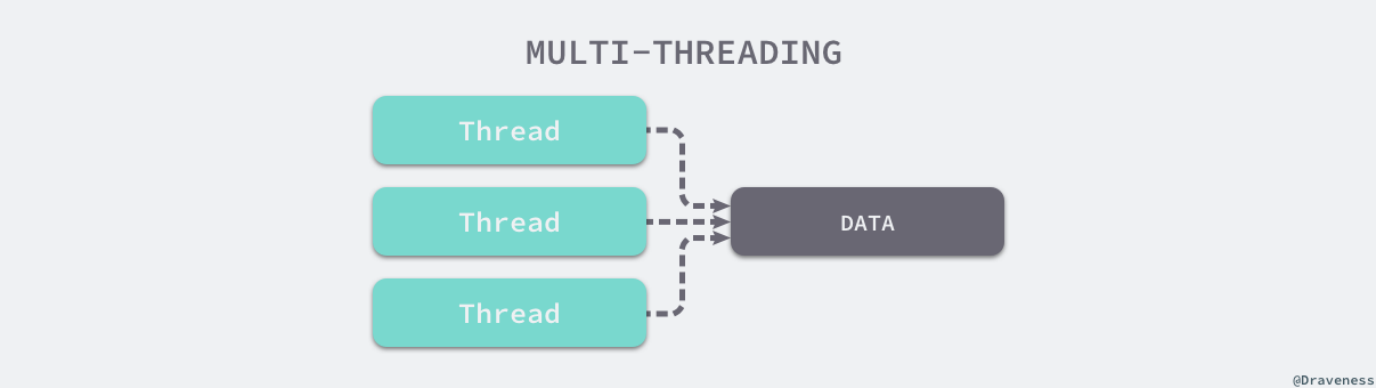
1. 使用单线程模型能带来更好的可维护性，方便开发和调试；

2. 使用单线程模型也能并发的处理客户端的请求；
3. Redis 服务中运行的绝大多数操作的性能瓶颈都不是 CPU；

上述三个原因中的最后一个是最终使用单线程模型的决定性因素，其他的两个原因都是使用单线程模型额外带来的好处，在这里我们会按顺序介绍上述的几个原因。

可维护性

可维护性对于一个项目来说非常重要，如果代码难以调试和测试，问题也经常难以复现，这对于任何一个项目来说都会严重地影响项目的可维护性。多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，代码的执行过程不再是串行的，多个线程同时访问的变量如果没有谨慎处理就会带来诡异的问题。



在网络上有一个调侃多线程模型的段子，就很好地展示了多线程模型带来的潜在问题：竞争条件 (race condition) —— 如果计算机中的两个进程（线程同理）同时尝试修改一个共享内存的内容，在没有并发控制的情况下，最终的结果依赖于两个进程的执行顺序和时机，如果发生了并发访问冲突，最后的结果就会是不正确的。

> Some people, when confronted with a problem, think, “I know, I’ll use threads,” and then two they hav erpoblesms.

引入了多线程，我们就必须要同时引入并发控制来保证在多个线程同时访问数据时程序行为的正确性，这就需要工程师额外维护并发控制的相关代码，例如，我们会需要在可能被并发读写的变量上增加互斥锁：

```
var (
    mu Mutex // cost
    data int
)

// thread 1
```

```
func() {
    mu.Lock()
    data += 1
    mu.Unlock()
}

// thread 2
func() {
    mu.Lock()
    data -= 1
    mu.Unlock()
}
```

在访问这些变量或者内存之前也需要先对获取互斥锁，一旦忘记获取锁或者忘记释放锁就可能会导致各种诡异的问题，管理相关的并发控制机制也需要付出额外的研发成本和负担。

并发处理

使用单线程模型也并不意味着程序不能并发的处理任务，Redis 虽然使用单线程模型处理用户的请求，但是它却使用 I/O 多路复用机制并发处理来自客户端的多个连接，同时等待多个连接发送的请求。

在 I/O 多路复用模型中，最重要的函数调用就是 `select` 以及类似函数，该方法的能够同时监控多个文件描述符（也就是客户端的连接）的可读可写情况，当其中的某些文件描述符可读或者可写时，`select` 方法就会返回可读以及可写的文件描述符个数。

使用 I/O 多路复用技术能够极大地减少系统的开销，系统不再需要额外创建和维护进程和线程来监听来自客户端的大量连接，减少了服务器的开发成本和维护成本。

性能瓶颈

最后要介绍的其实就是 Redis 选择单线程模型的决定性原因 —— 多线程技术能够帮助我们充分利用 CPU 的计算资源来并发的执行不同的任务，但是 CPU 资源往往都不是 Redis 服务器的性能瓶颈。哪怕我们在一个普通的 Linux 服务器上启动 Redis 服务，它也能在 1s 的时间内处理 1,000,000 个用户请求。

> It’s not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, using pipelining Redis running on an average Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.

如果这种吞吐量不能满足我们的需求，更推荐的做法是使用分片的方式将不同的请求交给不同的 Redis 服务器来处理，而不是在同一个 Redis 服务中引入大量的多线程操作。

简单总结一下，Redis 并不是 CPU 密集型的服务，如果不开启 AOF 备份，所有 Redis 的操作都会在内存中完成不会涉及任何的 I/O 操作，这些数据的读写由于只发生在内存中，所以处理速度是非常快的；整个服务的瓶颈在于网络传输带来的延迟和等待客户端的数据传输，也就是网络 I/O，所以使用多线程模型处理全部的外部请求可能不是一个好的方案。

> AOF 是 Redis 的一种持久化机制，它会在每次收到来自客户端的写请求时，将其记录到日志中，每次 Redis 服务器启动时都会重放 AOF 日志构建原始的数据集，保证数据的持久性。

多线程虽然会帮助我们更充分地利用 CPU 资源，但是操作系统上线程的切换也不是免费的，线程切换其实会带来额外的开销，其中包括：

1. 保存线程 1 的执行上下文；
2. 加载线程 2 的执行上下文；

频繁的对线程的上下文进行切换可能还会导致性能地急剧下降，这可能会导致我们不仅没有提升请求处理的平均速度，反而进行了负优化，所以这也是为什么 Redis 对于使用多线程技术非常谨慎。

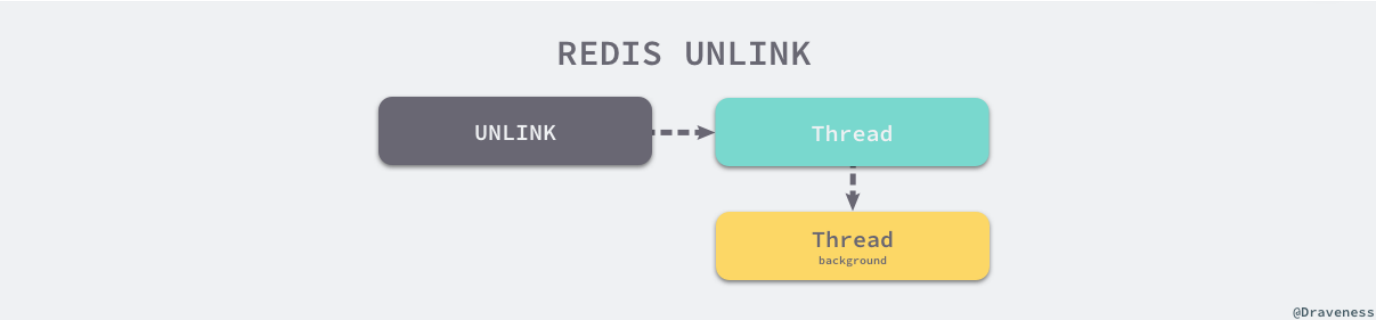
引入多线程

Redis 在最新的几个版本中加入了一些可以被其他线程异步处理的删除操作，也就是我们在上面提到的 UNLINK、FLUSHALL ASYNC 和 FLUSHDB ASYNC，我们为什么会需要这些删除操作，而它们为什么需要通过多线程的方式异步处理？

删除操作

我们可以在 Redis 在中使用 DEL 命令来删除一个键对应的值，如果待删除的键值对占用了较小的内存空间，那么哪怕是同步地删除这些键值对也不会消耗太多的时间。

但是对于 Redis 中的一些超大键值对，几十 MB 或者几百 MB 的数据并不能在几毫秒的时间内处理完，Redis 可能会需要在释放内存空间上消耗较多的时间，这些操作就会阻塞待处理的任务，影响 Redis 服务处理请求的 PCT99 和可用性。



然而释放内存空间的工作其实可以由后台线程异步进行处理，这也就是 UNLINK 命令的实现原理，它只会将键从元数据中删除，真正的删除操作会在后台异步执行。

总结

Redis 选择使用单线程模型处理客户端的请求主要还是因为 CPU 不是 Redis 服务器的瓶颈，所以使用多线程模型带来的性能提升并不能抵消它带来的开发成本和维护成本，系统的性能瓶颈也主要在网络 I/O 操作上；而 Redis 引入多线程操作也是出于性能上的考虑，对于一些大键值对的删除操作，通过多线程非阻塞地释放内存空间也能减少对 Redis 主线程阻塞的时间，提高执行的效率。

> 如果对文章中的内容有疑问或者想要了解更多软件工程上一些设计决策背后的原因，可以在博客下面留言，作者会及时回复本文相关的疑问并选择其中合适的主题作为后续的内容。

Reference

- Redis is single threaded. How can I exploit multiple CPU / cores?
- Redis is single-threaded, then how does it do concurrent I/O?
- Why isn't Redis designed to benefit from multi-threading?
- Concurrency is not parallelism
- The little-known feature of Redis 4.0 that will speed up your applications
- Redis 和 I/O 多路复用
- CPU Scheduling

{% include related/whys-the-design.md %}

转载申请

Go 语言设计与实现

各位读者朋友，很高兴大家通过本博客学习 Go 语言，感谢一路相伴！ 《Go语言设计与实现》 的纸质版图书已经上架京东，本书目前已经四印，印数超过 10,000 册，有需要的朋友请点击 链接 或者下面的图片购买。

文章图片

你可以在 技术文章配图指南 中找到画图的方法和素材。

0 个反应

54 条评论 – *由 giscus 驱动*

最旧 最新

输入 预览 Aa

登录以评论

支持 Markdown GitHub 登录

Zingphoy 2019 年 10 月 19 日

文章说， `CPU 资源往往都不是 Redis 服务器的性能瓶颈` ，这是不使用多线程的理由之一，没弄明白是什么意思

1. 如果不是 `cpu-bound` ，那是否代表添加一定的线程数是没问题的?
2. 如果是 `network I/O-bound` ，那为什么不通过添加线程来减少阻塞等待呢?
   懵了

↑ 1 0 条回复

draveness 2019 年 10 月 19 日 所有者

> 文章说， `CPU 资源往往都不是 Redis 服务器的性能瓶颈` ，这是不使用多线程的理由之一，没弄明白是什么意思
>
> 1. 如果不是 `cpu-bound` ，那是否代表添加一定的线程数是没问题的?
> 2. 如果是 `network I/O-bound` ，那为什么不通过添加线程来减少阻塞等待呢?
>   懵了

引入 IO 多路复用可以同时监听多个连接的可读和可写，不需要多个线程来做这个事情，引入了不必要的复杂性

↑ 1 0 条回复

zheng-bobo 2019 年 10 月 19 日 已编辑

大佬能探讨下Golang为什么要使用Plan 9的汇编器么？这篇文章的观点把Plan 9贬得一无是处? 实际上真[illegible]是这样么

draveness 2019 年 10 月 19 日 所有者

> 大佬能探讨下Golang为什么要使用Plan 9的汇编器么？这篇文章的观点把Plan 9贬得一无是处? 实际上真的是这样么

可以的，我看一下这个

↑ 1 ☺ 0 条回复

polunzh 2019 年 10 月 22 日

`DEL` 这样的 `IO` 操作是异步执行的，那么怎么判断这些操作是否执行完成了呢?

↑ 1 ☺ 0 条回复

draveness 2019 年 10 月 22 日 所有者

> `DEL` 这样的 `IO` 操作是异步执行的，那么怎么判断这些操作是否执行完成了呢?

你说的是 UNLINK 吧，我觉得调用方不会在乎这个操作什么时候会执行成功，如果在乎的话就会调用 DEL 了

↑ 1 ☺ 0 条回复

wenmoxiao 2019 年 10 月 22 日

不是因为作者不想过多考虑多线程引入的复杂性么？ ~ ~ 将部分问题丢给用户了

↑ 0 ☺ 0 条回复

draveness 2019 年 10 月 22 日 所有者

> 不是因为作者不想过多考虑多线程引入的复杂性么？ ~ ~ 将部分问题丢给用户了

还是因为纯内存服务够快了，不然处理能力很差，我们也不会用 Redis

↑ 1 ☺ 0 条回复

yuanjize 2019 年 10 月 22 日

你在做基础架构么

↑ 1 ☺ 0 条回复

moshiale 2019 年 10 月 23 日

大佬，请问你的画图工具是什么?

↑ 1 ☺ 0 条回复

OrdinaryYZH 2019 年 10 月 23 日

看了这篇文章，说到Redis6.0引入了多线程，是为了解决网络的 IO 消耗，能解释下这个具体是什么意思吗?还有这个跟IO多路复用要解决的问题有什么关系呢?

↑ 1 ☺ 0 条回复

draveness 2019 年 10 月 23 日 所有者 已编辑

> 看了这篇文章，说到Redis6.0引入了多线程，是为了解决网络的 IO 消耗，能解释下这个具体是什么意思吗?还有这个跟IO多路复用要解决的问题有什么关系呢?

这是一个非常好的问题，I/O 多路复用的主要作用是让我们可以使用一个线程来监控多个连接是否可读或者可写，但是从网络另一头发的数据包需要**先解序列化成 Redis 内部其他模块可以理解的命令**，这个过程就是 Redis 6.0 引入多线程来并发处理的。

I/O 多路复用模块收到数据包之后将其丢给后面多个 I/O 线程进行解析，I/O 线程处理结束后，主线程会负责串行的执行这些命令，由于向客户端发回数据包的过程也是比较耗时的，所以执行之后的结果也会交给多个 I/O 线程发送回客户端。

↑ 1 ☺ 0 条回复

chinaran 2019 年 10 月 24 日

文章写得很棒!

↑ 1 ☺ ❤ 1 0 条回复

liangdas 2019 年 10 月 28 日

> @polunzh

> DEL 这样的 IO 操作是异步执行的，那么怎么判断这些操作是否执行完成了呢？

只要从元数据删除是在主线程，后续的读取操作就都认为这个key已经被删除了

↑ 1 ☺ 👍 1 0 条回复

tanteng 2019 年 11 月 9 日

> **@moshiale**
> 大佬，请问你的画图工具是什么？

同问

↑ 1 ☺ 0 条回复

24 个隐藏项
加载更多…

fenghaojiang 2020 年 12 月 8 日

Some people, when confronted with a problem, think, “I know, I’ll use threads,” and then two they hav erpoblesms.

typo

↑ 1 ☺ 0 条回复

draveness 2020 年 12 月 8 日 所有者

> **@fenghaojiang**
> Some people, when confronted with a problem, think, “I know, I’ll use threads,” and then two they hav erpoblesms.
>
> typo

这并不是 typo

↑ 1 ☺ 0 条回复

dawncold 2020 年 12 月 8 日

因为用了threads ---- Sincerely, Zhen Tian

...

↑ 1 ☺ 0 条回复

ZMbiubiubiu 2021 年 1 月 5 日

> **@Aiome**
> 你文章解释性能瓶颈感觉解释的有问题。
> 开启多线程是为了提高CPU利用率。CPU密集型的操作不需要开启多线程提高CPU利用率，CPU本来就忙啊，线程数与CPU核心数相同就好了。IO密集型才需要开启多线程，A线程IO操作时CPU需要等待时间很长，这时切换到B线程进行计算操作，提高CPU利用率。
> 我感觉Reids设计应该是认为单线程IO操作时我CPU等会就等会了，比切换其他线程进行计算还要快。

我大概理解你意思，我理解的意思：Redis主处理程序主要与内存打交道，没有什么I/O处理，所以多线程并不适用，也就是说Redis的瓶颈不在于I/O，Redis不是I/O-bound吧，我读到原文，也感觉怪怪的。

↑ 1 ☺ 0 条回复

SpikeWong 2021 年 1 月 20 日

```
{% include related/whys-the-design.md %}
```

你的 template 失效了

↑ 1 ☺ 0 条回复

mlbjay 2021 年 3 月 15 日

虽然 看过Redis文档，虽然 也写过 IO多路复用，但是 这一篇文章 真是把这个问题讲透了！！！

↑ 1 ☺ 0 条回复

Jessezhao920 2021 年 4 月 21 日

在这里，我认为与其和多线程的去做对比，不如直接搞明白NIO和IO的，因为redis既然是一种缓存，直接操作与内存中，而我们一直在那多线程的复杂性，是初看上去，与单线程对应的就是多线程，但我认为并不是提升速度的核心，这是我的一点见解。

↑ 1 ☺ 0 条回复

Jigsawk 2021 年 6 月 15 日

> @Aiome

@Aiome

> 你文章解释性能瓶颈感觉解释的有问题。
> 开启多线程是为了提高CPU利用率。CPU密集型的操作不需要开启多线程提高CPU利用率，CPU本来就忙啊，线程数与CPU核心数相同就好了。IO密集型才需要开启多线程，A线程IO操作时CPU需要等待时间很长，这时切换到B线程进行计算操作，提高CPU利用率。
> 我感觉Reids设计应该是认为单线程IO操作时我CPU等会就等会了，比切换其他线程进行计算还要快。

我大致也看懂了你的意思，刚读完文章我也觉得作者由“Redis 服务中运行的绝大多数操作的性能瓶颈都不是 CPU”得出“没必要开启多线程”的这个说法有点问题。诚然可以通过selector来处理网络连接，但是如果开启多线程处理指令，应该依然能够进一步提升效率（提升了CPU的利用率）

我觉得因为redis在单线程处理连接与指令方面已经很快了，如果要引入多线程处理指令，虽然也可能一定程度上提升效率，但是却要考虑共享数据的线程安全问题，以及引入事务等概念？这在开发维护上带来了很高的复杂度。redis可能并不想做一个大而全的东西，只要能够在自己专注的场景中，获得高性能就可以了。如果对读写有更高要求，那就横向拓展。

↑ 1 ☺ ❤ 1 0 条回复

 **Johnson-jjy** 2021 年 7 月 23 日 已编辑

很喜欢作者的文章，看了都觉得受益匪浅。。以前我还会忽略评论，今天发现评论也是满满的精华

↑ 1 ☺ 0 条回复

 **887412** 2021 年 9 月 15 日

> @Zingphoy
> 文章说， `CPU 资源往往都不是 Redis 服务器的性能瓶颈` ，这是不使用多线程的理由之一，没弄明白是什么意思
>
> 1. 如果不是 `cpu-bound` ，那是否代表添加一定的线程数是没问题的?
> 2. 如果是 `network I/O-bound` ，那为什么不通过添加线程来减少阻塞等待呢?
>
> 懵了

我们想一下木桶效应，最终的性能是由哪个短木板决定的；cpu的处理速度远高于内存，而redis的父进程运行在内存中，我们知道引入多线程的目的是为了把比较耗时的任务非阻塞式处理来提高利用率，在内存中处理速度都非常快，比较耗时redis的主要性能瓶颈是内存和网络IO， 我们不会因为引入了多线程，内存和网络IO的处理速度就会变快，比较好的方式

↑ 1 ☺ 0 条回复

 **887412** 2021 年 9 月 15 日

> @Jigsawk

> **@Aiome**
> 你文章解释性能瓶颈感觉解释的有问题。
> 开启多线程是为了提高CPU利用率。CPU密集型的操作不需要开启多线程提高CPU利用率，CPU本来就忙啊，线程数与CPU核心数相同就好了。IO密集型才需要开启多线程，A线程IO操作时CPU需要等待时间很长，这时切换到B线程进行计算操作，提高CPU利用率。
> 我感觉Reids设计应该是认为单线程IO操作时我CPU等会就等会了，比切换其他线程进行计算还要快。
>
> 我大致也看懂了你的意思，刚读完文章我也觉得作者由“Redis 服务中运行的绝大多数操作的性能瓶颈都不是 CPU”得出“没必要开启多线程”的这个说法有点问题。诚然可以通过selector来处理网络连接，但是如果开启多线程处理指令，应该依然能够进一步提升效率（提升了CPU的利用率）
>
> 我觉得因为redis在单线程处理连接与指令方面已经很快了，如果要引入多线程处理指令，虽然也可能一定程度上提升效率，但是却要考虑共享数据的线程安全问题，以及引入事务等概念？这在开发维护上带来了很高的复杂度。redis可能并不想做一个大而全的东西，只要能够在自己专注的场景中，获得高性能就可以了。如果对读写有更高要求，那就横向拓展。

因为引入多线程是可以达到部分优化的，比如redis4.0的对垃圾的处理和redis6.0对命令的处理，但是redis的性能瓶颈是内存和网络IO，引入再多的线程也没办法解决根本矛盾，所以注定redis中多线程只能用于优化非阻塞式处理比较耗时的任务，但是根本上还是沿着单线程的思路，即使是redis6.0，它命令的执行采用单线程，只有在命令的处理上使用多线程。

↑ 1 ☺ 0 条回复

BrianQy 2021 年 11 月 18 日

单线程是主工作，那引入了多线程，还有纯粹的单线程吗？处理IO和备份或是删除的多线程就不会和主工作的单线程之间有切换的情况出现吗？那文中提到线程切换是有开销的，那这个开销怎么看待呢？

↑ 1 ☺ 0 条回复

zhangmuwuge 2022 年 1 月 6 日

是不是可以增加一些写出数据到网络的线程呢?充分利用cpu....这个时候已经没有数据互斥关系的成本,只有上下文切换的成本,只需要控制这个写出线程数而不会让cpu压力太大就行了

突如其来的一个想法....

↑ 1 ☺ 0 条回复

pqf773 2022 年 2 月 27 日

> **@donnior**
>
> > **@Crearns**
> > 你好，想请问一下，这种模型是不是类似Java的Netty中负责处理连接的BossEventLoopGroup和负责网络I/O的WorkEventLoopGroup，BossEventLoopGroup线程数一般为1，而

> 和负责网络I/O的WorkEventLoopGroup，BossEventLoopGroup线程数一般为1，而WorkEventLoopGroup一般为多线程?
>
> 还是不一样的，只能说两者都用到了io复用的技术；但是Netty采用的是一种reactor模式，你说的这个主线程来负责处理连接，另外一个线程池来处理读写事件，所以netty在整个处理io的时候实际上是使用了多个线程的（当然第二个线程池你也可以选择继续用来处理自己的业务）；而redis在不一样，它虽然使用了io复用，但是并不区分连接和读写，所有的io操作都变成事件由一个线程来处理，甚至于后面的命令执行也可以在这个线程中。
>
> redis之所以可以采用这种模型的原因完全是因为它可以非常快，因为它的数据都在内存中，操作都是纳秒级的，同时它的网络协议也会简单，读写解码都很快；而netty作为一个通用的网络框架并不能假定用户在上面执行什么操作，所以需要用这种线程模型来达到极致的效率； 而实际上，在redis这种模型下，如果你在高并发场景下来测试读写一些大数据（增加io的读写时间），或者执行一些耗时的命令，是可以看到比较明显的性能差异的。

mark

↑ 1 ☺ 0 条回复

zhangyuanxue 2022 年 7 月 21 日

使用单线程模型也并不意味着程序不能并发的处理任务，`Redis` 虽然使用单线程模型处理用户的请求，但是它却使用 `I/`

大佬，我理解的是redis虽然能够通过IO多路复用并发地处理用户的请求，但是实际后面涉及到的对所有连接的内存数据的读取还是通过单线程来处理的对吧?