17-为什么CPU结构也会影响Redis的性能?

你好，我是蒋德钧。

很多人都认为Redis和CPU的关系很简单，就是Redis的线程在CPU上运行，CPU快，Redis处理请求的速度也很快。

这种认知其实是片面的。CPU的多核架构以及多CPU架构，也会影响到Redis的性能。如果不了解CPU对Redis的影响，在对Redis的性能进行调优时，就可能会遗漏一些调优方法，不能把Redis的性能发挥到极限。

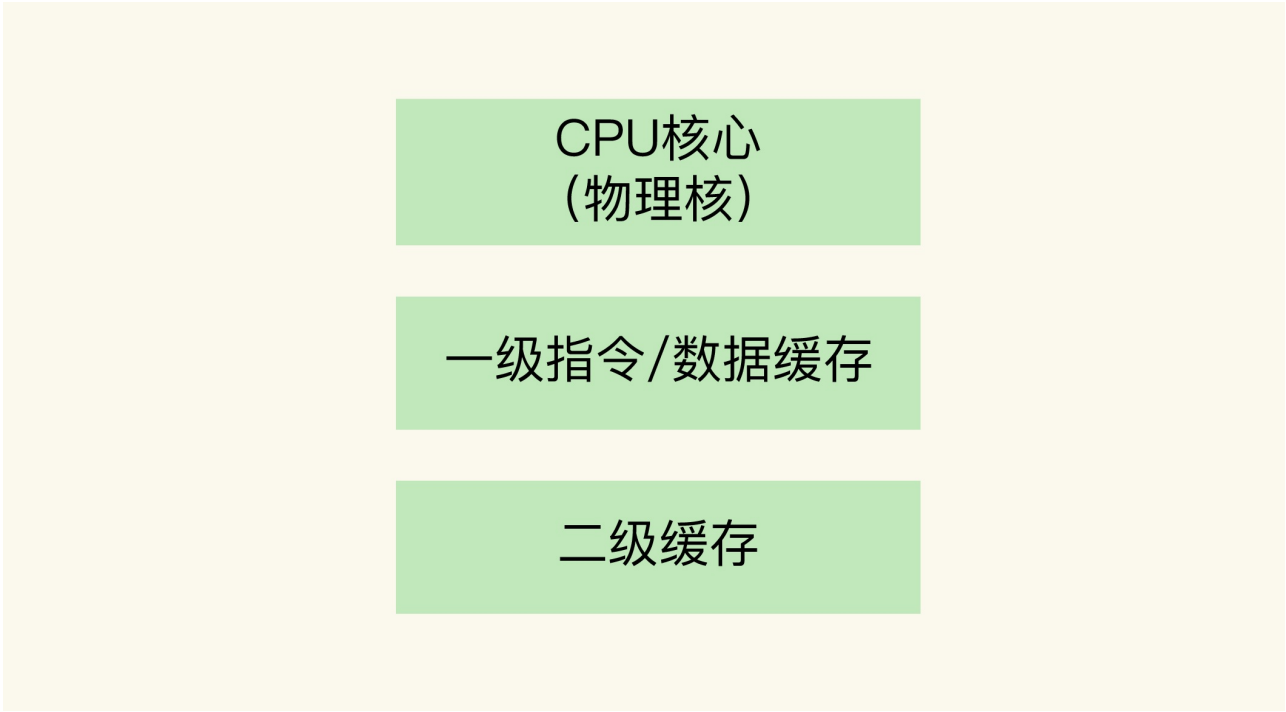
今天，我们就来学习下目前主流服务器的CPU架构，以及基于CPU多核架构和多CPU架构优化Redis性能的方法。

主流的CPU架构

要了解CPU对Redis具体有什么影响，我们得先了解一下CPU架构。

一个CPU处理器中一般有多个运行核心，我们把一个运行核心称为一个物理核，每个物理核都可以运行应用程序。每个物理核都拥有私有的一级缓存（Level 1 cache，简称L1 cache），包括一级指令缓存和一级数据缓存，以及私有的二级缓存（Level 2 cache，简称L2 cache）。

这里提到了一个概念，就是物理核的私有缓存。它其实是指缓存空间只能被当前的这个物理核使用，其他的物理核无法对这个核的缓存空间进行数据存取。我们来看一下CPU物理核的架构。



因为L1和L2缓存是每个物理核私有的，所以，当数据或指令保存在L1、L2缓存时，物理核访问它们的延迟不超过10纳秒，速度非常快。那么，如果Redis把要运行的指令或存取的数据保存在L1和L2缓存的话，就能高速地访问这些指令和数据。

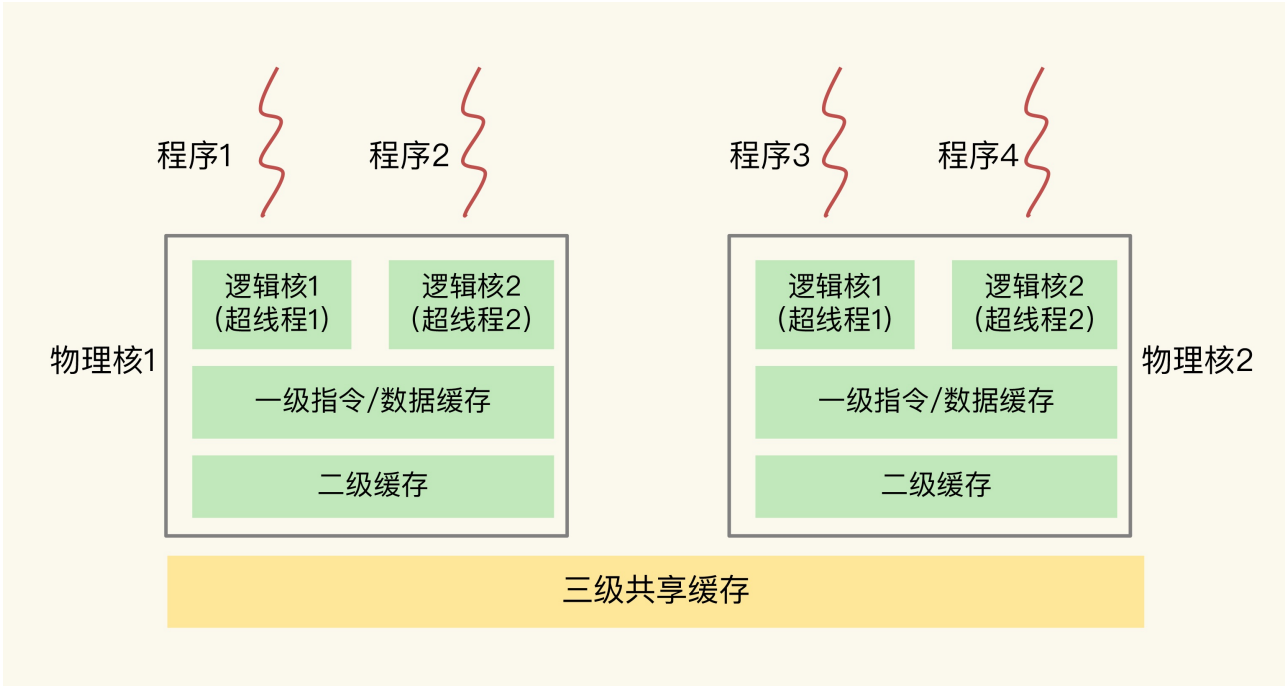
但是，这些L1和L2缓存的大小受限于处理器的制造技术，一般只有KB级别，存不下太多的数据。如果L1、L2缓存中没有所需的数据，应用程序就需要访问内存来获取数据。而应用程序的访存延迟一般在百纳秒级

别，是访问L1、L2缓存的延迟的近10倍，不可避免地会对性能造成影响。

所以，不同的物理核还会共享一个共同的三级缓存（Level 3 cache，简称为L3 cache）。L3缓存能够使用的存储资源比较多，所以一般比较大，能达到几MB到几十MB，这就能让应用程序缓存更多的数据。当L1、L2缓存中没有数据缓存时，可以访问L3，尽可能避免访问内存。

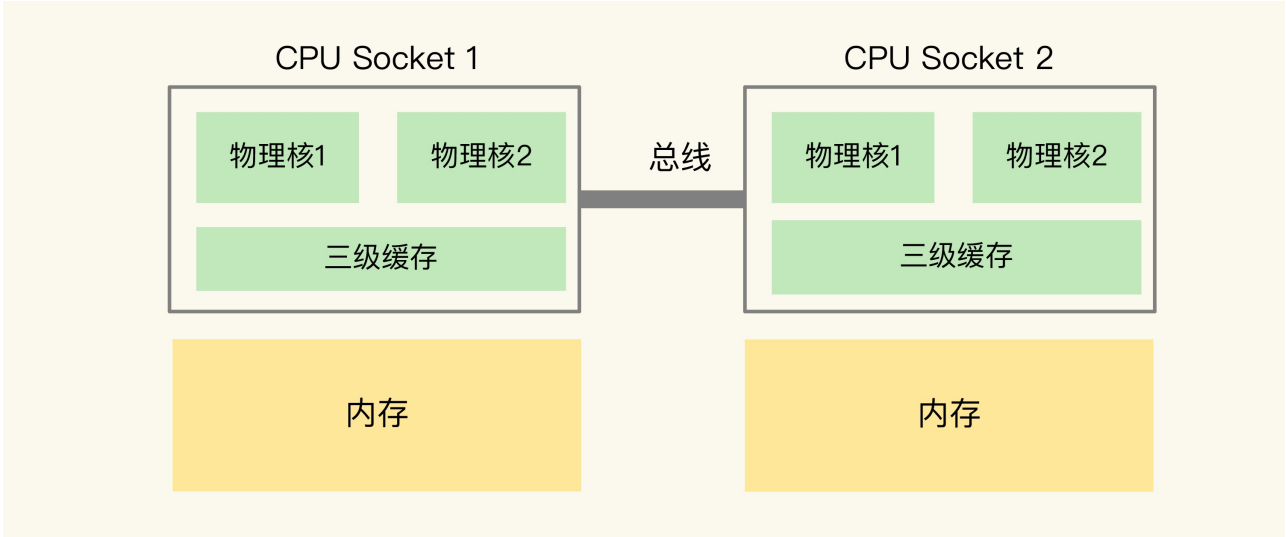
另外，现在主流的CPU处理器中，每个物理核通常都会运行两个超线程，也叫作逻辑核。同一个物理核的逻辑核会共享使用L1、L2缓存。

为了方便你理解，我用一张图展示一下物理核和逻辑核，以及一级、二级缓存的关系。



在主流的服务器上，一个CPU处理器会有10到20多个物理核。同时，为了提升服务器的处理能力，服务器上通常还会有多个CPU处理器（也称为多CPU Socket），每个处理器有自己的物理核（包括L1、L2缓存），L3缓存，以及连接的内存，同时，不同处理器间通过总线连接。

下图显示的就是多CPU Socket的架构，图中有两个Socket，每个Socket有两个物理核。



在多CPU架构上，应用程序可以在不同的处理器上运行。在刚才的图中，Redis可以先在Socket 1上运行一

段时间，然后再被调度到Socket 2上运行。

但是，有个地方需要你注意一下：如果应用程序先在一个Socket上运行，并且把数据保存到了内存，然后被调度到另一个Socket上运行，此时，应用程序再进行内存访问时，就需要访问之前Socket上连接的内存，这种访问属于**远端内存访问**。**和访问Socket直接连接的内存相比，远端内存访问会增加应用程序的延迟。**

在多CPU架构下，一个应用程序访问所在Socket的本地内存和访问远端内存的延迟并不一致，所以，我们也把这个架构称为非统一内存访问架构（Non-Uniform Memory Access，NUMA架构）。

到这里，我们就知道了主流的CPU多核架构和多CPU架构，我们来简单总结下CPU架构对应用程序运行的影响。

- L1、L2缓存中的指令和数据的访问速度很快，所以，充分利用L1、L2缓存，可以有效缩短应用程序的执行时间；
- 在NUMA架构下，如果应用程序从一个Socket上调度到另一个Socket上，就可能会出现远端内存访问的情况，这会直接增加应用程序的执行时间。

接下来，我们就先来了解下CPU多核是如何影响Redis性能的。

CPU多核对Redis性能的影响

在一个CPU核上运行时，应用程序需要记录自身使用的软硬件资源信息（例如栈指针、CPU核的寄存器值等），我们把这些信息称为**运行时信息**。同时，应用程序访问最频繁的指令和数据还会被缓存到L1、L2缓存上，以便提升执行速度。

但是，在多核CPU的场景下，一旦应用程序需要在一个新的CPU核上运行，那么，运行时信息就需要重新加载到新的CPU核上。而且，新的CPU核的L1、L2缓存也需要重新加载数据和指令，这会导致程序的运行时间增加。

说到这儿，我想跟你分享一个我曾经在多核CPU环境下对Redis性能进行调优的案例。希望借助这个案例，帮你全方位地了解到多核CPU对Redis的性能的影响。

当时，我们的项目需求是要对Redis的99%尾延迟进行优化，要求GET尾延迟小于300微秒，PUT尾延迟小于500微秒。

可能有同学不太清楚99%尾延迟是啥，我先解释一下。我们把所有请求的处理延迟从小到大排个序，**99%的请求延迟小于的值就是99%尾延迟**。比如说，我们有1000个请求，假设按请求延迟从小到大排序后，第991个请求的延迟实测值是1ms，而前990个请求的延迟都小于1ms，所以，这里的99%尾延迟就是1ms。

刚开始的时候，我们使用GET/PUT复杂度为O(1)的String类型进行数据存取，同时关闭了RDB和AOF，而且，Redis实例中没有保存集合类型的其他数据，也就没有bigkey操作，避免了可能导致延迟增加的许多情况。

但是，即使这样，我们在一台有24个CPU核的服务器上运行Redis实例，GET和PUT的99%尾延迟分别是504微秒和1175微秒，明显大于我们设定的目标。

后来，我们仔细检测了Redis实例运行时的服务器CPU的状态指标值，这才发现，CPU的context switch次数比较多。

context switch是指线程的上下文切换，这里的上下文就是线程的运行时信息。在CPU多核的环境中，一个线程先在一个CPU核上运行，之后又切换到另一个CPU核上运行，这时就会发生context switch。

当context switch发生后，Redis主线程的运行时信息需要被重新加载到另一个CPU核上，而且，此时，另一个CPU核上的L1、L2缓存中，并没有Redis实例之前运行时频繁访问的指令和数据，所以，这些指令和数据都需要重新从L3缓存，甚至是内存中加载。这个重新加载的过程是需要花费一定时间的。而且，Redis实例需要等待这个重新加载的过程完成后，才能开始处理请求，所以，这也会导致一些请求的处理时间增加。

如果在CPU多核场景下，Redis实例被频繁调度到不同CPU核上运行的话，那么，对Redis实例的请求处理时间影响就更大了。**每调度一次，一些请求就会受到运行时信息、指令和数据重新加载过程的影响，这就会导致某些请求的延迟明显高于其他请求**。分析到这里，我们就知道了刚刚的例子中99%尾延迟的值始终降不下来的原因。

所以，我们要避免Redis总是在不同CPU核上来回调度执行。于是，我们尝试着把Redis实例和CPU核绑定了，让一个Redis实例固定运行在一个CPU核上。我们可以使用**taskset命令**把一个程序绑定在一个核上运行。

比如说，我们执行下面的命令，就把Redis实例绑在了0号核上，其中，“-c”选项用于设置要绑定的核编号。

```
taskset -c 0 ./redis-server
```

绑定以后，我们进行了测试。我们发现，Redis实例的GET和PUT的99%尾延迟一下子就分别降到了260微秒和482微秒，达到了我们期望的目标。

我们来看一下绑核前后的Redis的99%尾延迟。

命令	未绑核运行Redis的99%尾延迟	绑核运行Redis的99%尾延迟
GET	504us	260us
PUT	1175us	482us

可以看到，在CPU多核的环境下，通过绑定Redis实例和CPU核，可以有效降低Redis的尾延迟。当然，绑核不仅对降低尾延迟有好处，同样也能降低平均延迟、提升吞吐率，进而提升Redis性能。

接下来，我们再来看看多CPU架构，也就是NUMA架构，对Redis性能的影响。

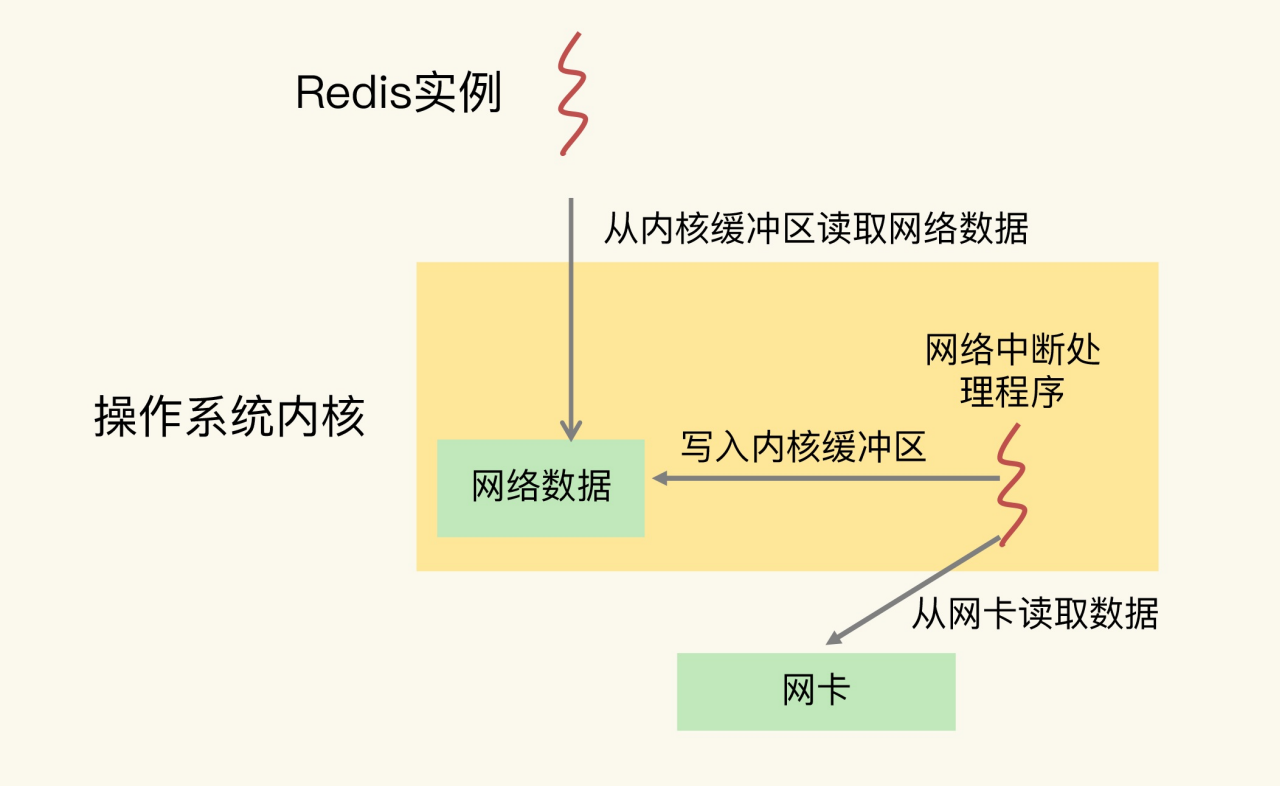
CPU的NUMA架构对Redis性能的影响

在实际应用Redis时，我经常看到一种做法，为了提升Redis的网络性能，把操作系统的网络中断处理程序和

CPU核绑定。这个做法可以避免网络中断处理程序在不同核上来回调度执行，的确能有效提升Redis的网络处理性能。

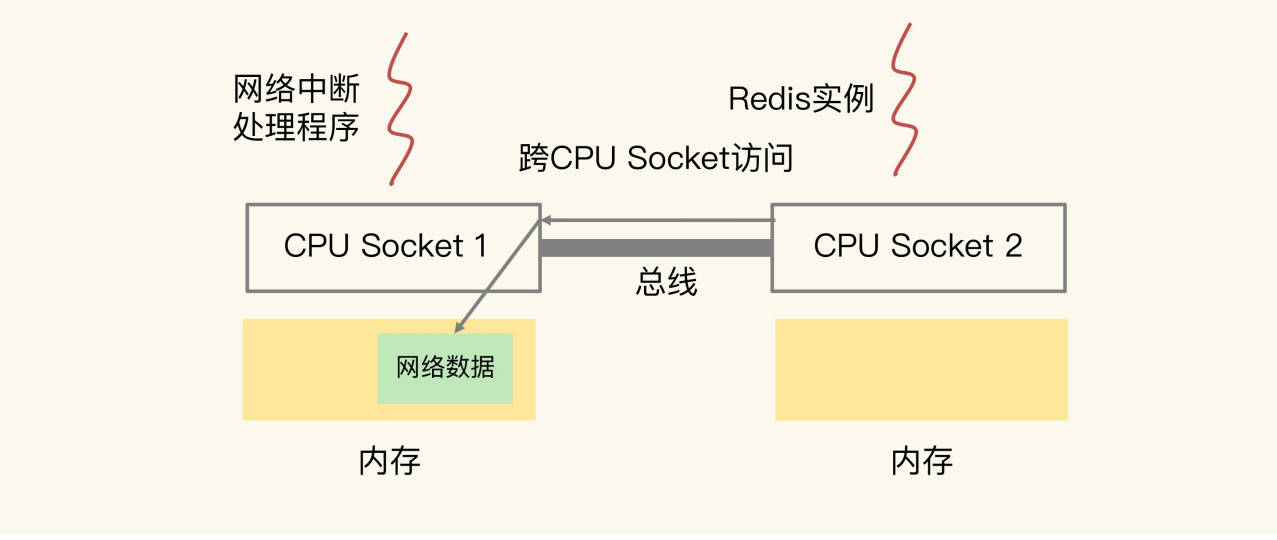
但是，网络中断程序是要和Redis实例进行网络数据交互的，一旦把网络中断程序绑核后，我们就需要注意Redis实例是绑在哪个核上了，这会关系到Redis访问网络数据的效率高低。

我们先来看下Redis实例和网络中断程序的数据交互：网络中断处理程序从网卡硬件中读取数据，并把数据写入到操作系统内核维护的一块内存缓冲区。内核会通过epoll机制触发事件，通知Redis实例，Redis实例再把数据从内核的内存缓冲区拷贝到自己的内存空间，如下图所示：



那么，在CPU的NUMA架构下，当网络中断处理程序、Redis实例分别和CPU核绑定后，就会有一个潜在的风险：**如果网络中断处理程序和Redis实例各自所绑的CPU核不在同一个CPU Socket上，那么，Redis实例读取网络数据时，就需要跨CPU Socket访问内存，这个过程会花费较多时间。**

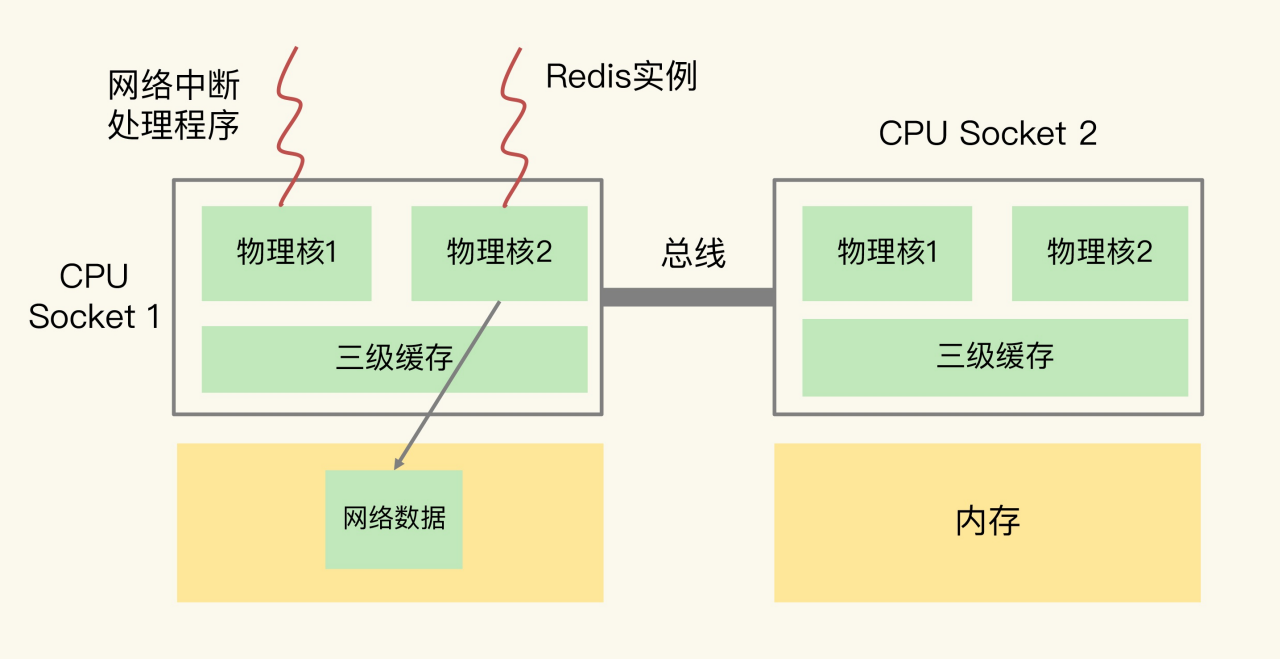
这么说可能有点抽象，我再借助一张图来解释下。



可以看到，图中的网络中断处理程序被绑在了CPU Socket 1的某个核上，而Redis实例则被绑在了CPU Socket 2上。此时，网络中断处理程序读取到的网络数据，被保存在CPU Socket 1的本地内存中，当Redis实例要访问网络数据时，就需要Socket 2通过总线把内存访问命令发送到 Socket 1上，进行远程访问，时间开销比较大。

我们曾经做过测试，和访问CPU Socket本地内存相比，跨CPU Socket的内存访问延迟增加了18%，这自然会导致Redis处理请求的延迟增加。

所以，为了避免Redis跨CPU Socket访问网络数据，我们最好把网络中断程序和Redis实例绑在同一个CPU Socket上，这样一来，Redis实例就可以直接从本地内存读取网络数据了，如下图所示：



不过，需要注意的是，**在CPU的NUMA架构下，对CPU核的编号规则，并不是先把一个CPU Socket中的所有逻辑核编完，再对下一个CPU Socket中的逻辑核编码，而是先给每个CPU Socket中每个物理核的第一个逻辑核依次编号，再给每个CPU Socket中的物理核的第二个逻辑核依次编号。**

我给你举个例子。假设有2个CPU Socket，每个Socket上有6个物理核，每个物理核又有2个逻辑核，总共24个逻辑核。我们可以执行**lscpu命令**，查看到这些核的编号：

```
lscpu

Architecture: x86_64
...
NUMA node0 CPU(s): 0-5,12-17
NUMA node1 CPU(s): 6-11,18-23
...
```

可以看到，NUMA node0的CPU核编号是0到5、12到17。其中，0到5是node0上的6个物理核中的第一个逻辑核的编号，12到17是相应物理核中的第二个逻辑核编号。NUMA node1的CPU核编号规则和node0一样。

所以，在绑核时，我们一定要注意，不能想当然地认为第一个Socket上的12个逻辑核的编号就是0到11。否

则，网络中断程序和Redis实例就可能绑在了不同的CPU Socket上。

比如说，如果我们把网络中断程序和Redis实例分别绑到编号为1和7的CPU核上，此时，它们仍然是在2个CPU Socket上，Redis实例仍然需要跨Socket读取网络数据。

所以，你一定要注意NUMA架构下CPU核的编号方法，这样才不会绑错核。

我们先简单地总结下刚刚学习的内容。在CPU多核的场景下，用taskset命令把Redis实例和一个核绑定，可以减少Redis实例在不同核上被来回调度执行的开销，避免较高的尾延迟；在多CPU的NUMA架构下，如果你对网络中断程序做了绑核操作，建议你同时把Redis实例和网络中断程序绑在同一个CPU Socket的不同核上，这样可以避免Redis跨Socket访问内存中的网络数据的时间开销。

不过，“硬币都是有两面的”，绑核也存在一定的风险。接下来，我们就来了解下它的潜在风险点和解决方案。

绑核的风险和解决方案

Redis除了主线程以外，还有用于RDB生成和AOF重写的子进程（可以回顾看下第4讲和第5讲）。此外，我们还在第16讲学习了Redis的后台线程。

当我们把Redis实例绑到一个CPU逻辑核上时，就会导致子进程、后台线程和Redis主线程竞争CPU资源，一旦子进程或后台线程占用CPU时，主线程就会被阻塞，导致Redis请求延迟增加。

针对这种情况，我来给你介绍两种解决方案，分别是**一个Redis实例对应绑一个物理核和优化Redis源码。**

方案一：一个Redis实例对应绑一个物理核

在给Redis实例绑核时，我们不要把一个实例和一个逻辑核绑定，而要和一个物理核绑定，也就是说，把一个物理核的2个逻辑核都用上。

我们还是以刚才的NUMA架构为例，NUMA node0的CPU核编号是0到5、12到17。其中，编号0和12、1和13、2和14等都是表示一个物理核的2个逻辑核。所以，在绑核时，我们使用属于同一个物理核的2个逻辑核进行绑核操作。例如，我们执行下面的命令，就把Redis实例绑定到了逻辑核0和12上，而这两个核正好都属于物理核1。

```
taskset -c 0,12 ./redis-server
```

和只绑一个逻辑核相比，把Redis实例和物理核绑定，可以让主线程、子进程、后台线程共享使用2个逻辑核，可以在一定程度上缓解CPU资源竞争。但是，因为只用了2个逻辑核，它们相互之间的CPU竞争仍然还会存在。如果你还想进一步减少CPU竞争，我再给你介绍一种方案。

方案二：优化Redis源码

这个方案就是通过修改Redis源码，把子进程和后台线程绑到不同的CPU核上。

如果你对Redis的源码不太熟悉，也没关系，因为这是通过编程实现绑核的一个通用做法。学会了这个方案，你可以在熟悉了源码之后把它用上，也可以应用在其他需要绑核的场景中。

接下来，我先介绍一下通用的做法，然后，再具体说说可以把这个做法对应到Redis的哪部分源码中。

通过编程实现绑核时，要用到操作系统提供的1个数据结构cpu_set_t和3个函数CPU_ZERO、CPU_SET和sched_setaffinity，我先来解释下它们。

- cpu_set_t数据结构：是一个位图，每一位用来表示服务器上的一个CPU逻辑核。
- CPU_ZERO函数：以cpu_set_t结构的位图为输入参数，把位图中所有的位设置为0。
- CPU_SET函数：以CPU逻辑核编号和cpu_set_t位图为参数，把位图中和输入的逻辑核编号对应的位设置为1。
- sched_setaffinity函数：以进程/线程ID号和cpu_set_t为参数，检查cpu_set_t中哪一位为1，就把输入的ID号所代表的进程/线程绑在对应的逻辑核上。

那么，怎么在编程时把这三个函数结合起来实现绑核呢？很简单，我们分四步走就行。

- 第一步：创建一个cpu_set_t结构的位图变量；
- 第二步：使用CPU_ZERO函数，把cpu_set_t结构的位图所有的位都设置为0；
- 第三步：根据要绑定的逻辑核编号，使用CPU_SET函数，把cpu_set_t结构的位图相应位设置为1；
- 第四步：使用sched_setaffinity函数，把程序绑定在cpu_set_t结构位图中为1的逻辑核上。

下面，我就具体介绍下，分别把后台线程、子进程绑到不同的核上的做法。

先说后台线程。为了让你更好地理解编程实现绑核，你可以看下这段示例代码，它实现了为线程绑核的操作：

```
//线程函数
void worker(int bind_cpu){
    cpu_set_t cpuset;  //创建位图变量
    CPU_ZERO(&cpu_set); //位图变量所有位设置0
    CPU_SET(bind_cpu, &cpuset); //根据输入的bind_cpu编号，把位图对应为设置为1
    sched_setaffinity(0, sizeof(cpuset), &cpuset); //把程序绑定在cpu_set_t结构位图中为1的逻辑核

    //实际线程函数工作
}

int main(){
    pthread_t pthread1
    //把创建的pthread1绑在编号为3的逻辑核上
    pthread_create(&pthread1, NULL, (void *)worker, 3);
}
```

对于Redis来说，它是在bio.c文件中的bioProcessBackgroundJobs函数中创建了后台线程。bioProcessBackgroundJobs函数类似于刚刚的例子中的worker函数，在这个函数中实现绑核四步操作，就可以把后台线程绑到和主线程不同的核上了。

和给线程绑核类似，当我们使用fork创建子进程时，也可以把刚刚说的四步操作实现在fork后的子进程代码中，示例代码如下：

```
int main(){
   //用fork创建一个子进程
   pid_t p = fork();
   if(p < 0){
      printf(" fork error\n");
   }
   //子进程代码部分
   else if(!p){
      cpu_set_t cpuset;  //创建位图变量
      CPU_ZERO(&cpu_set); //位图变量所有位设置0
      CPU_SET(3, &cpuset); //把位图的第3位设置为1
      sched_setaffinity(0, sizeof(cpuset), &cpuset);  //把程序绑定在3号逻辑核
      //实际子进程工作
      exit(0);
   }
   ...
}
```

对于Redis来说，生成RDB和AOF日志重写的子进程分别是下面两个文件的函数中实现的。

- rdb.c文件：rdbSaveBackground函数；
- aof.c文件：rewriteAppendOnlyFileBackground函数。

这两个函数中都调用了fork创建子进程，所以，我们可以在子进程代码部分加上绑核的四步操作。

使用源码优化方案，我们既可以实现Redis实例绑核，避免切换核带来的性能影响，还可以让子进程、后台线程和主线程不在同一个核上运行，避免了它们之间的CPU资源竞争。相比使用taskset绑核来说，这个方案可以进一步降低绑核的风险。

小结

这节课，我们学习了CPU架构对Redis性能的影响。首先，我们了解了目前主流的多核CPU架构，以及NUMA架构。

在多核CPU架构下，Redis如果在不同的核上运行，就需要频繁地进行上下文切换，这个过程会增加Redis的执行时间，客户端也会观察到较高的尾延迟了。所以，建议你在Redis运行时，把实例和某个核绑定，这样，就能重复利用核上的L1、L2缓存，可以降低响应延迟。

为了提升Redis的网络性能，我们有时还会把网络中断处理程序和CPU核绑定。在这种情况下，如果服务器使用的是NUMA架构，Redis实例一旦被调度到和中断处理程序不在同一个CPU Socket，就要跨CPU Socket访问网络数据，这就会降低Redis的性能。所以，我建议你把Redis实例和网络中断处理程序绑在同一个CPU Socket下的不同核上，这样可以提升Redis的运行性能。

虽然绑核可以帮助Redis降低请求执行时间，但是，除了主线程，Redis还有用于RDB和AOF重写的子进程，以及4.0版本之后提供的用于惰性删除的后台线程。当Redis实例和一个逻辑核绑定后，这些子进程和后台线

程会和主线程竞争CPU资源，也会对Redis性能造成影响。所以，我给了你两个建议：

- 如果你不想修改Redis代码，可以把按一个Redis实例一个物理核方式进行绑定，这样，Redis的主线程、子进程和后台线程可以共享使用一个物理核上的两个逻辑核。
- 如果你很熟悉Redis的源码，就可以在源码中增加绑核操作，把子进程和后台线程绑到不同的核上，这样可以避免对主线程的CPU资源竞争。不过，如果你不熟悉Redis源码，也不用太担心，Redis 6.0出来后，可以支持CPU核绑定的配置操作了，我将在第38讲中向你介绍Redis 6.0的最新特性。

Redis的低延迟是我们永恒的追求目标，而多核CPU和NUMA架构已经成为了目前服务器的主流配置，所以，希望你能掌握绑核优化方案，并把它应用到实践中。

每课一问

按照惯例，我给你提个小问题。

在一台有2个CPU Socket（每个Socket 8个物理核）的服务器上，我们部署了有8个实例的Redis切片集群（8个实例都为主节点，没有主备关系），现在有两个方案：

1. 在同一个CPU Socket上运行8个实例，并和8个CPU核绑定；
2. 在2个CPU Socket上各运行4个实例，并和相应Socket上的核绑定。

如果不考虑网络数据读取的影响，你会选择哪个方案呢？

欢迎在留言区写下你的思考和答案，如果你觉得有所收获，也欢迎你帮我把今天的内容分享给你的朋友。我们下节课见。

精选留言：

- Kaito 2020-09-16 00:07:35
  这篇文章收获很大！对于CPU结构和如何绑核有了进一步了解。其实在NUMA架构下，不光对于CPU的绑核需要注意，对于内存的使用，也有很多注意点，下面回答课后问题，也会提到NUMA架构下内存方面的注意事项。

  在一台有2个CPU Socket（每个Socket 8个物理核）的服务器上，我们部署了有8个实例的Redis切片集群（8个实例都为主节点，没有主备关系），采用哪种方案绑核最佳？

  我更倾向于的方案是：在两个CPU Socket上各运行4个实例，并和相应Socket上的核绑定。这么做的原因主要从L3 Cache的命中率、内存利用率、避免使用到Swap这三个方面考虑：

  1、由于CPU Socket1和2分别有自己的L3 Cache，如果把所有实例都绑定在同一个CPU Socket上，相当于这些实例共用这一个L3 Cache，另一个CPU Socket的L3 Cache浪费了。这些实例共用一个L3 Cache，会导致Cache中的数据频繁被替换，访问命中率下降，之后只能从内存中读取数据，这会增加访问的延迟。而8个实例分别绑定CPU Socket，可以充分使用2个L3 Cache，提高L3 Cache的命中率，减少从内存读取数据的开销，从而降低延迟。

  2、如果这些实例都绑定在一个CPU Socket，由于采用NUMA架构的原因，所有实例会优先使用这一个节点的内存，当这个节点内存不足时，再经过总线去申请另一个CPU Socket下的内存，此时也会增加延迟。而8个实例分别使用2个CPU Socket，各自在访问内存时都是就近访问，延迟最低。

3、如果这些实例都绑定在一个CPU Socket，还有一个比较大的风险是：用到Swap的概率将会大大提高。如果这个CPU Socket对应的内存不够了，也可能不会去另一个节点申请内存（操作系统可以配置内存回收策略和Swap使用倾向：本节点回收内存/其他节点申请内存/内存数据换到Swap的倾向程度），而操作系统可能会把这个节点的一部分内存数据换到Swap上从而释放出内存给进程使用（如果没开启Swap可会导致直接OOM）。因为Redis要求性能非常高，如果从Swap中读取数据，此时Redis的性能就会急剧下降，延迟变大。所以8个实例分别绑定CPU Socket，既可以充分使用2个节点的内存，提高内存使用率，而且触发使用Swap的风险也会降低。

其实我们可以查一下，在NUMA架构下，也经常发生某一个节点内存不够，但其他节点内存充足的情况下，依旧使用到了Swap，进而导致软件性能急剧下降的例子。所以在运维层面，我们也需要关注NUMA架构下的内存使用情况（多个内存节点使用可能不均衡），并合理配置系统参数（内存回收策略/Swap使用倾向），尽量去避免使用到Swap。 [31赞]

- test 2020-09-16 08:49:53
  课后问题：我会选择方案二。首先一个实例不止有一个线程需要运行，所以方案一肯定会有CPU竞争问题；其次切片集群的通信不是通过内存，而是通过网络IO。 [1赞]

- 土豆白菜 2020-09-16 23:10:29
  老师，我也想问下比如azure redis 能否做这些优化

- 游弋云端 2020-09-16 15:29:04
  有两套房子，就不用挤着睡吧，优选方案二。老师实验用的X86的CPU吧，对于ARM架构来讲，存在着跨DIE和跨P的说法，跨P的访问时延会更高，且多个P之间的访问存在着NUMA distances的说法，不同的布局导致的跨P访问时延也不相同。

- zhou 2020-09-16 14:54:52
  在 NUMA 架构下，比如有两个 CPU Socket：CPU Socket 1 和 CPU Socket 2，每个 CPU Socket 都有自己的内存，CPU Socket 1 有自己的内存 Mem1，CPU Socket 2 有自己的内存 Mem2。

  Redis 实例在 CPU Socket 1 上执行，网络中断处理程序在 CPU Socket 2 上执行，所以 Redis 实例的数据在内存 Mem1 上，网络中断处理程序的数据在 Mem2上。

  因此 Redis 实例读取网络中断处理程序的内存数据（Mem2）时，是需要远端访问的，比直接访问自己的内存数据（Mem1）要慢。

- 那时刻 2020-09-16 10:25:05
  请问老师，您文中提到我们仔细检测了 Redis 实例运行时的服务器 CPU 的状态指标值，这才发现，CPU 的 context switch 次数比较多。再遇到这样的问题的时候，排查的点有哪些呢？

- 写点啥呢 2020-09-16 08:54:20
  请问蒋老师，文章的例子代码是硬编码了子进程绑定的CPU编号，这样因为不知道运行时主进程绑定的CPU还是会有一定竞争的风险。那么有没有可以避免这种情况的方案，能够动态根据主进程绑定的情况分配子进程应该使用的CPU编号的实现呢？

- jacky 2020-09-16 08:08:38
  绑核用的都是逻辑核编号吧，那么在云虚机进行相关操作也是一样的了？