13 | 实战：单机如何实现管理百万主机的心跳服务？

2020-05-27 陶辉

讲述：陶辉
时长 13:31 大小 12.39M

你好，我是陶辉。

这一讲我们将结合前 12 讲，以一个可管理百万主机集群的心跳服务作为实战案例，看看所有高性能服务的设计思路。

首先解释下什么是心跳服务。集群中的主机如果宕机，那么管理服务必须及时发现，并做相应的容灾处理，比如将宕机主机的业务迁移到新的虚拟机上等等。怎么做到及时发现呢？可以要求每台主机定时上报心跳包，考虑到网络报文的延迟，如果管理服务在几个上报周期未收到心跳，则认为主机宕机。当新主机加入集群后，心跳服务也可以及时识别并告知管理服务。

这就是心跳服务要解决的核心问题，虽然很简单，可是如果集群规模达到百万台虚拟机或者微服务进程，这就不再简单了。多核 CPU、内存使用效率、网络带宽时延积等都必须纳入你的考虑，因为此时心跳包占用的网络带宽已经接近网卡上限，仅调动一颗 CPU 的计算力去处理就会大量丢包，百万级的对象、网络连接也很容易造成内存 OOM。甚至判断宕机的算法都要重新设计，降低时间复杂度后才能够应对超大集群的心跳管理。

解决这种集群规模下的性能问题，需要深入掌握底层基础知识，用系统化的全局思维去解决问题，这也是程序员薪资差异的重要分水岭。接下来，我们先实现更高效的核心算法，再设计高并发服务的架构，最后再来看传输层协议的选择。

如何设计更快的宕机判断算法？

通过心跳包找到宕机的主机需要一套算法，比如用 for 循环做一次遍历，找到停止上报心跳的主机就可以实现，然而正如[第 3 讲] 所说，**当管理的对象数量级很大时，算法复杂度会严重影响程序性能**，遍历算法此时并不可取。我们先分析下这个算法的时间复杂度。

如果用红黑树（这里用红黑树是因为它既支持遍历，也可以实现对数时间复杂度的查询操作）存放主机及其最近一次上报时间，那么，新主机上报心跳被发现的流程，时间复杂度仅为 O(logN)，这是查询红黑树的成本。寻找宕机服务的流程，需要对红黑树做全量遍历，用当前时间去比较每个主机的上次心跳时间，时间复杂度就是 O(N)！

如果业务对时间灵敏度要求很高，就意味着需要频繁地执行 O(N) 级的遍历，当 N 也就是主机数量很大时，耗时就很可观了。而且寻找宕机服务和接收心跳包是两个流程，如果它们都在单线程中执行，那么寻找宕机服务的那段时间就不能接收心跳包，会导致丢包！如果使用多线程并发执行，因为两个流程都需要操作红黑树，所以要使用到互斥锁，而当这两个流程争抢锁的频率很高时，性能也会急剧下降。

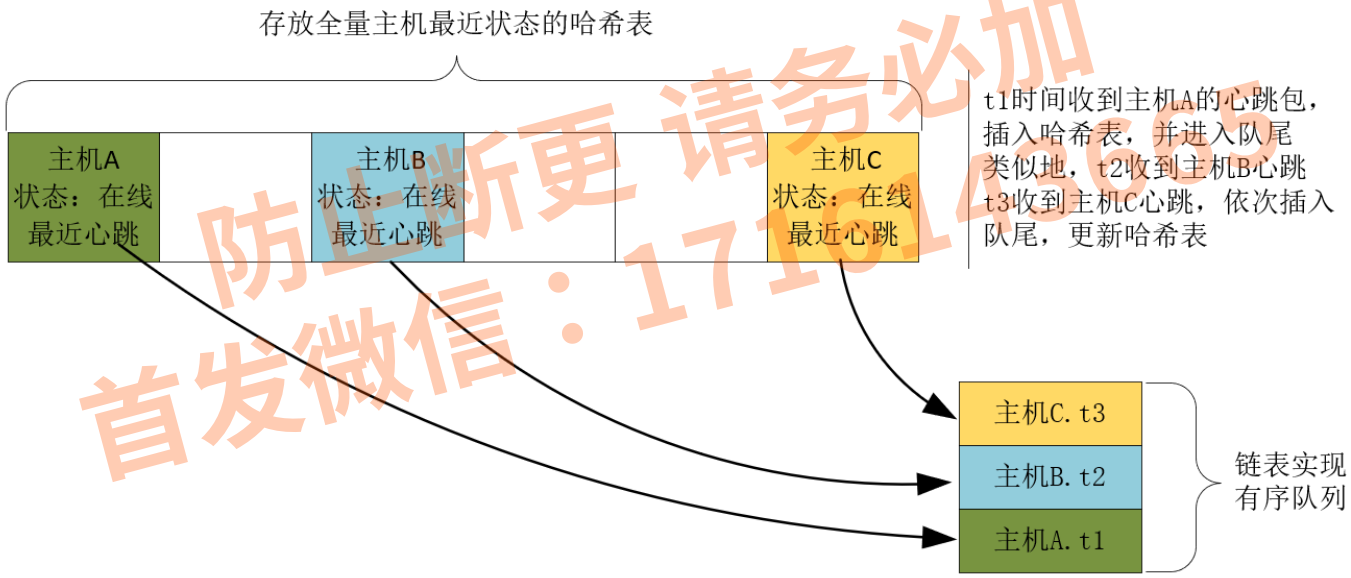
其实这个算法的根本问题在于，判断宕机的流程做了大量的重复工作。比如，主机每隔 1 秒上报一次心跳，而考虑到网络可能丢包，故 5 秒内失去心跳就认为宕机，这种情况下，如果主机 A 在第 10 秒时失去心跳，那么第 11、12、13、14 这 4 秒对主机 A 的遍历，都是多余的，只有第 15 秒对主机 A 的遍历才有意义。于是，每次遍历平均浪费了 4/5 的计算量。

如何设计快速的宕机判断算法呢？其实，这是一个从一堆主机中寻找宕机服务的信息题。**根据香农的理论，引入更多的信息，才能减少不确定性降低信息熵，从而减少计算量。就像心**

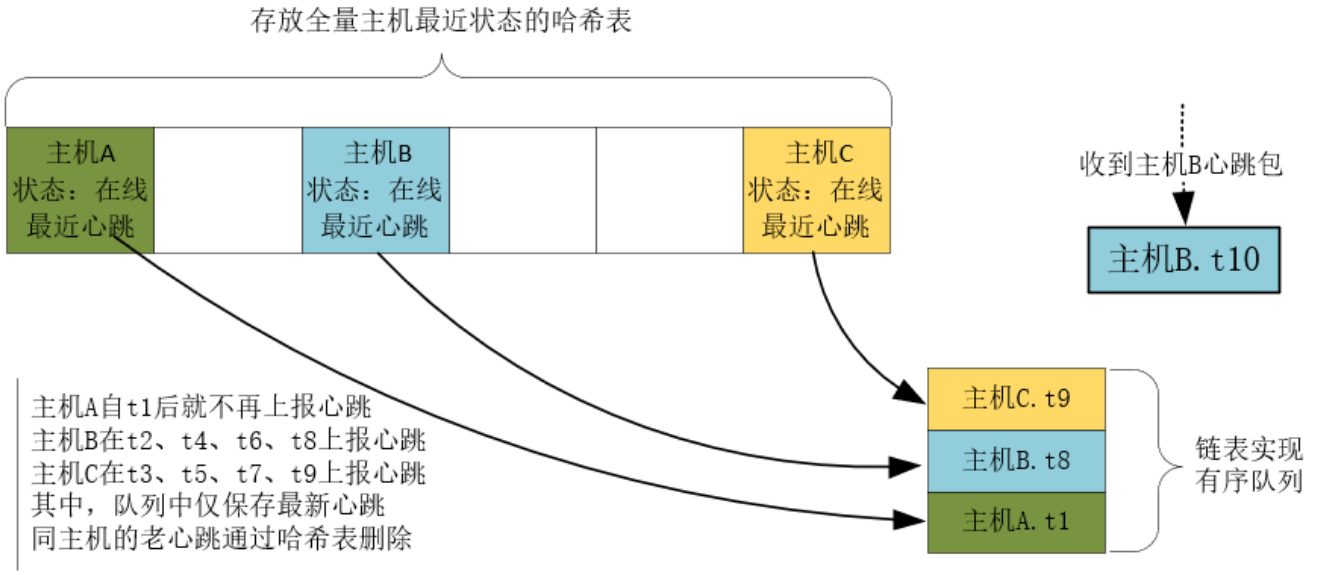
跳包间是有时间顺序的，上面的宕机判断算法显然忽略了接收到它们的顺序。比如主机 A 的上次心跳包距现在 4 秒了，而主机 B 距现在只有 1 秒，显然不应同等对待。

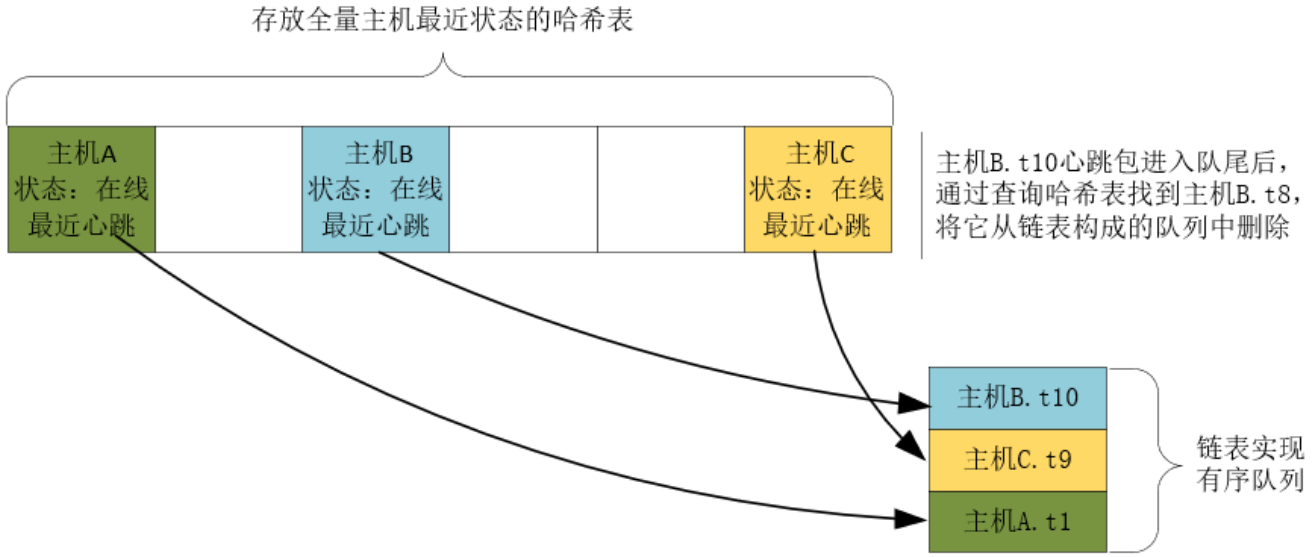
于是，我们引入存放心跳包的先入先出队列，这就保存了心跳包的时序关系。新的心跳包进入队列尾部，而老的心跳包则从队列首部退出，这样，寻找宕机服务时，只要看队列首部最老的心跳包，距现在是否超过 5 秒，如果超过 5 秒就认定宕机，同时把它取出队列，否则证明队列中不存在宕机服务，维持队列不变。

当然，这里并没有解决如何发现新主机的问题。我们还需要一个能够执行高效查询的容器，存放所有主机及其状态。红黑树虽然不慢，但我们不再需要遍历容器，所以可以选择更快的、查询时间复杂度为 O(1) 的哈希表存放主机信息（非标哈希表的实现参见[第 3 讲]）。如下图所示。

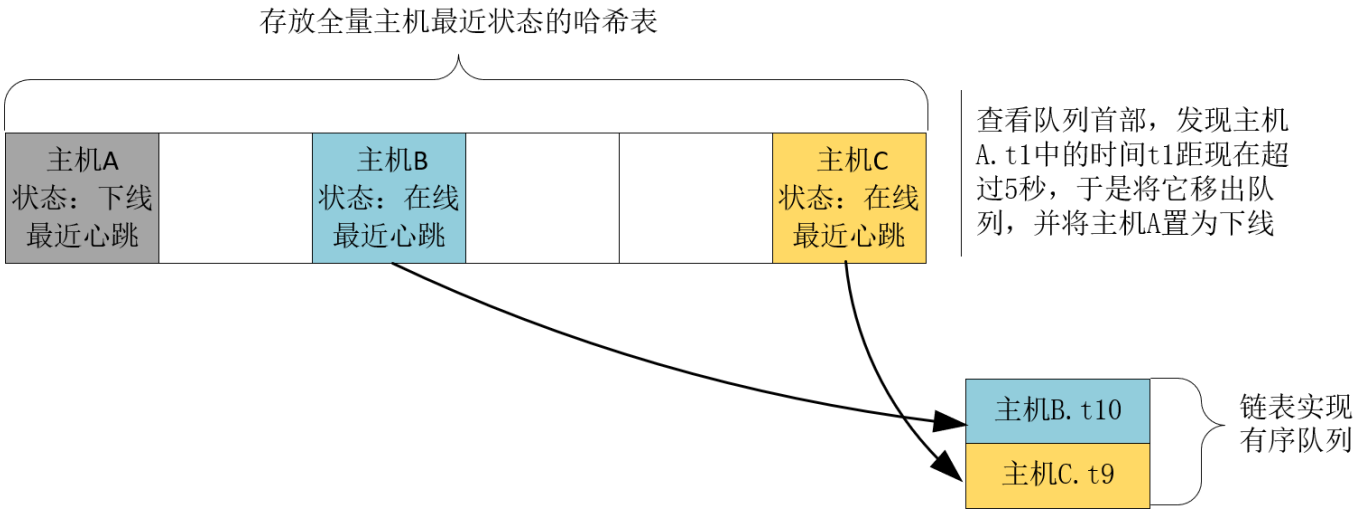


当然，队列中的心跳包并不是只能从队首删除，否则判断宕机流程的时间复杂度仍然是 O(N)。实际上，每当收到心跳包时，如果对应主机的上一个心跳包还在队列中，那么可以直接把它从队列中删除。显然，计算在线主机何时宕机，只需要最新的心跳包，老的心跳包没有必要存在。因此，这个队列为每个主机仅保留最新的那个心跳包。如下图所示：





这样，判断宕机的速度会非常快，它的计算量等于实际发生宕机的主机数量。同时，接收心跳包并发现新主机的流程，因为只需要做一次哈希表查询，时间复杂度也只有 O(1)。



这样，新算法通过**以空间换时间**的思想，虽然使用了更加占用空间的哈希表，并新增了有序队列容器，但将宕机和新主机发现这两个流程都优化到了常量级的时间复杂度。尤其是宕机流程的计算量非常小，它仅与实际宕机服务的数量有关，这就允许我们将宕机判断流程插入到心跳包的处理流程中，以微观上的分时任务实现宏观上的并发，同时也避免了对哈希表的加锁。

如何设计高并发架构?

有了核心算法，还需要充分利用服务器资源的架构，才能实现高并发。

一颗 1GHZ 主频的 CPU，意味着一秒钟只有 10 亿个时钟周期可以工作，如果心跳服务每秒接收到 100 万心跳包，就要求它必须在 1000 个时钟周期内处理完一个心跳包。这无法做到，因为每一个汇编指令的执行需要多个时钟周期（参见CPI），一条高级语言的语句又由多条汇编指令构成，而中间件提供的反序列化等函数又需要很多条语句才能完成。另外，内核从网卡上读取报文，执行协议分析需要的时钟周期也要算到这 1000 个时钟周期里。

因此，选择只用一颗 CPU 为核心的单线程开发模式，一定会出现计算力不足，不能及时接收报文从而使得缓冲区溢出的问题，最终导致大量丢包。所以，我们必须选择多线程或者多进程开发模式。多进程之间干扰更小，但内存不是共享的，数据同步较为困难，因此案例中我们还是选择多线程开发模式。

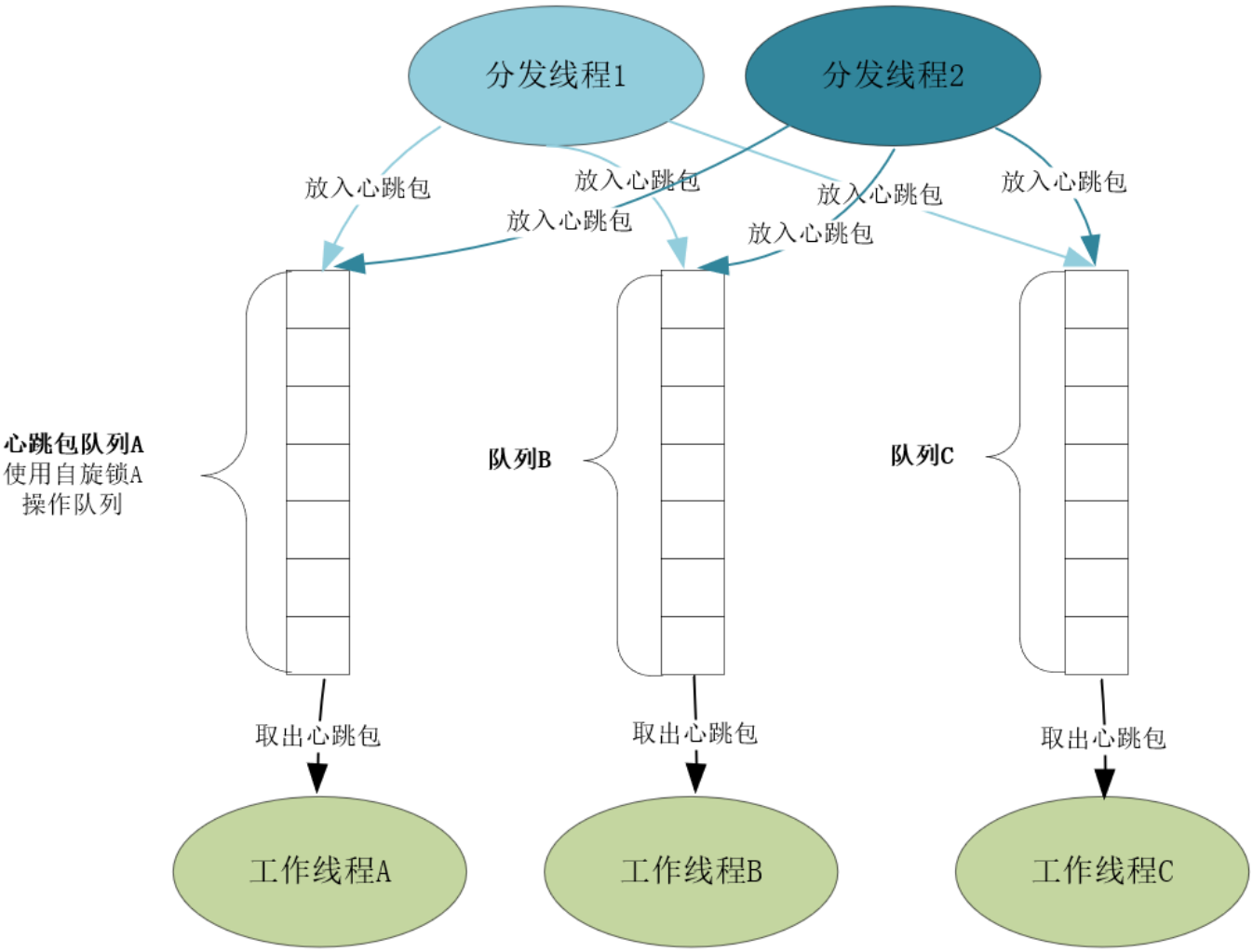
使用多线程后我们需要解决 3 个问题。

第一是负载均衡，我们应当把心跳包尽量均匀分配到不同的工作线程上处理。比如，接收网络报文的线程基于主机名或者 IP 地址，用哈希算法将心跳包分发给工作线程处理，这样每个工作线程只处理特定主机的心跳，相互间不会互相干扰，从而可以无锁编程。

第二是多线程同步。分发线程与工作线程间可以采用生产者 - 消费者模型传递心跳包，然而多线程间传递数据要加锁，为了减少争抢锁对系统资源的消耗，需要做到以下两点：

- 由于工作线程多过分发线程（接收心跳包消耗的资源更少），所以每个工作线程都配发独立的缓冲队列及操作队列的互斥锁；

为避免线程执行主动切换，必须使用自旋锁，关于锁的选择你可以看[第 6 讲]。如下图所示：



第三要解决 CPU 亲和性问题。从[第 1 讲]我们可以看到，CPU 缓存对计算速度的影响很大，如果线程频繁地切换 CPU 会导致缓存命中率下降，降低性能，此时将线程绑定到特定的 CPU 就是一个解决方案（NUMA 架构也会对 CPU 亲和性产生影响，这里略过）。

这样，通过上述的多线程架构就可以有效地使用 CPU。当然，除了 CPU，内存的使用效率也很重要。[第 2 讲]中我们提到，TCMalloc 相比 Linux 默认的 PtMalloc2 内存池，在多线程下分配小块内存的速度要快得多，所以对于心跳服务应当改用 TCMalloc 申请内存。而且，如果心跳包对象的格式已经固定，你还可以建立一个心跳包资源池，循环往复的使用，这进一步减少了构造、销毁心跳包对象所消耗的计算力。

由于服务重启后一个心跳周期内就可以获得所有心跳包，所以并不需要将数据持久化到磁盘上。如果你想进一步了解磁盘优化，可以再看下[第 4 讲]。

如何选择心跳包网络协议？

最后我们再来看看心跳包的协议该选择 TCP 还是 UDP 实现。

网络报文的长度是受限的，MTU（Maximum Transmission Unit）定义了最大值。比如以太网中 MTU 是 1500 字节，如果 TCP 或者 UDP 试图传送大于 1500 字节的报文，IP 协议就会把报文拆分后再发到网络中，并在接收方组装回原来的报文。然而，IP 协议并不擅长做这件事，拆包组包的效率很低，因此 TCP 协议宁愿自己拆包（详见[第 11 讲]）。

所以，如果心跳包长度小于 MTU，那么 UDP 协议是最佳选择。如果心跳包长度大于 MTU，那么最好选择 TCP 协议，面对复杂的 TCP 协议，还需要解决以下问题。

首先，一台服务器到底能同时建立多少 TCP 连接？要回答这个问题，得先从 TCP 四元组谈起，它唯一确定一个 TCP 连接。TCP 四元组分别是 < 源 IP、目的 IP、源端口、目的端口 >，其中前两者在 IP 头部中，后两者在 TCP 头部中。

<table>
<tr><td>版本号</td><td>头部长度</td><td colspan="2">服务类型</td><td colspan="4">总长度</td></tr>
<tr><td colspan="4">标识</td><td>标志位</td><td colspan="3">分片偏移</td></tr>
<tr><td colspan="2">TTL 生存时间</td><td colspan="2">上层协议</td><td colspan="4">首部校验和</td></tr>
<tr><td colspan="7">源 IP 地址</td></tr>
<tr><td colspan="7">目的 IP 地址</td></tr>
<tr><td colspan="4">源端口号</td><td colspan="3">目的端口号</td></tr>
<tr><td colspan="7">序列号</td></tr>
<tr><td colspan="7">确认号</td></tr>
<tr><td>首部长度</td><td>保留位</td><td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>F</td><td colspan="2">窗口大小</td></tr>
<tr><td colspan="4">校验和</td><td colspan="3">紧急指针</td></tr>
</table>

由于 IPv4 地址为 4 个字节（参见[第 7 讲]）、端口为 2 个字节，所以当服务器 IP 地址和监听端口固定时，并发连接数的上限则是 $2^{(32+16)}$。

当然，这么高的并发连接需要很多条件，其中之一就是增加单个进程允许打开的最大句柄数（包括操作系统允许的最大句柄数 /proc/sys/fs/file-nr），因为 Linux 下每个连接都要用掉一个文件句柄。当然，作为客户端的主机如果想用足 2^{16} 端口，还得修改 ip_local_port_range 配置，扩大客户端的端口范围：

复制代码

```
1  net.ipv4.ip_local_port_range = 32768    60999
```

其次，基于 TCP 协议实现百万级别的高并发，必须使用基于事件驱动的全异步开发模式（参见[第 8 讲]）。而且，TCP 协议的默认配置并没有考虑高并发场景，所以我们还得在以下 4 个方面优化 TCP 协议：

1. 三次握手建立连接的过程需要优化，详见[第 9 讲]；
2. 四次挥手关闭连接的过程也需要优化，详见[第 10 讲]；
3. 依据网络带宽时延积重新设置 TCP 缓冲区，详见[第 11 讲]；
4. 优化拥塞控制算法，详见[第 12 讲]。

最后还有一个问题需要我们考虑。网络中断时并没有任何信息通知服务器，此时该如何发现并清理服务器上的这些僵死连接呢？KeepAlive 机制允许服务器定时向客户端探测连接是否存活。其中，每隔 tcp_keepalive_time 秒执行一次探测。

```
net.ipv4.tcp_keepalive_time = 7200
```

复制代码

每次探测的最大等待时间是 tcp_keepalive_intvl 秒。

```
net.ipv4.tcp_keepalive_intvl = 75
```

复制代码

超时后，内核最多尝试 tcp_keepalive_probes 次，仍然没有反应就会及时关闭连接。

```
net.ipv4.tcp_keepalive_probes = 9
```

复制代码

当然，如果在应用层通过心跳能及时清理僵死 TCP 连接，效果会更好。

从上述优化方案可见，TCP 协议的高并发优化方案还是比较复杂的，这也是享受 TCP 优势时我们必须要付出的代价。

小结

这一讲以我实践过的项目为案例，介绍了高并发服务的设计思路。

核心算法对性能的影响最大，为了设计出高效的算法，我们必须分析出时间复杂度，充分寻找、利用已知信息，减少算法的计算量。在心跳服务这个案例中，利用好心跳包的时序，就可以把计算宕机的时间复杂度从 O(N) 降为 O(1)。

有了好的算法，还需要好的架构，才能高效地调动系统资源。当摩尔定律在 CPU 频率上失效后，CPU 都在向多核发展，所以高性能必须充分使用多核的计算力。此时，我们需要谨慎设计多线程间的负载均衡和数据同步，尽量减少访问共享资源带来的损耗。选择与业务场景匹配的内存池也很重要，对于 RPS 上百万的服务来说，申请内存的时间不再是一个忽略项。

选择网络协议时，如果消息长度大于 MTU，那么选择 TCP 更有利，但 TCP 解决了流控、可靠性等很多问题，优化起来较为困难。对于不要求可靠传输，长度通常不大的心跳包来说，UDP 协议通常是更好的选择。

思考题

最后，还是留给你点思考题。你遇到过心跳服务吗？它是怎么设计的？还有哪些优化空间？欢迎你在留言区与我探讨。

感谢阅读，如果你觉得这节课对你有一些启发，也欢迎把它分享给你的朋友。

上一篇 12 | 如何调整TCP拥塞控制的性能?

下一篇 14 | 优化TLS/SSL性能该从何下手?

精选留言 (13)

写留言

忆水寒

2020-05-27

本文收获蛮多的，可是没有遇到相关的场景，也希望多看看别人的思考和总结。
关于寻找宕机的节点，核心思路就是收到心跳包先插入队列尾部，然后通过哈希表找到队列中之前的位置进行判断是否宕机，并将队列中之前的心跳信息删除。这样用空间换时间，降低时间复杂度。
关于如何在单机模拟百万连接，可以参考https://mp.weixin.qq.com/s/6K9YDpdvsPbc...

展开

1

那时刻

2020-05-27

老师在文中提到的超过五秒没有收到心跳消息就会把这台主机从状态列表里删除了。可能有这种情况，比如主机A因为网络抖动超过五秒发来心跳消息，尽管它是alive状态，但是我们会误认为它下线了。为了应对这种情况，我们之前会采取一段时间内收到消息的百分数来计算，比如每秒一个心跳消息，在20秒内应该收到20个心跳消息，比如收到10个，比例是50%。以这个比例作为是否下线的阈值。请问老师，不知是否还有更好办法？

展开

作者回复: 你好那时刻，生产环境中必须容忍网络的抖动，因此容忍一定比例的丢包，是正常的，你的这个方法是可以的，当然，如果有些主机频繁的出现网络不稳定的话，也可以考虑用更平滑的函数来判断宕机

1 1

几米夜空
2020-05-30

有几个问题:
1.若使用UDP，有的系统不支持端口重用，只能打开一个相同UDP端口，这对于百万心跳，这有可能内核缓冲区就丢包，这怎么处理?
2.这还是需要去遍历哈希表，检查是否超时?若哈希函数不好，这链表也很长，这时间复杂度还是跟前面提到的一样，有没有什么好的哈希函数保证冲突少呢?

展开

作者回复: 1、子进程（包括线程）才可以复用父进程打开的端口，无关的进程是不能打开已经在监听的端口的。
2、可以看下第3课，里面有提到好的哈希函数该如何设计，另外装载因子也会影响冲突率

起飞的鸭子
2020-05-29

深度好文，大开眼界

展开

侠影
2020-05-28

有序队列中替换一个节点的心跳信息，不也需要遍历由于队列吗？这里有什么优化方案吗?

凉人。
2020-05-28

如果主机处理得速度不同，我想会不会心跳时间会有区别。这样用时间轮会不会好点。

作者回复: 你好凉人，如果是一个企业内部云上的主机，都会有NTP等服务来同步时间的，此时这套算法总体的计算成本最低。如果时间确实无法同步，需要应用代码来处理，那么时间轮也是一个不错的算法。


野性力量
2020-05-27

有什么开源的管理服务是这样设计的吗?

展开


野性力量
2020-05-27

提个和本文内容无关的问题，如果管理中心和服务网络断开，但是服务实际没有宕机，而且调用方也能访问服务，这种应该判定为宕机吗?

展开

作者回复: 应该判定为宕机


449038
2020-05-27

很棒的内容

展开


有铭
2020-05-27

寻找宕机服务时，只要看队列首部最老的心跳包，距现在是否超过 5 秒，如果超过 5 秒就认定宕机

==========

这里的逻辑无法理解：如果要用这种方式检测心跳，那么肯定要不停的把队列首部的心跳

包移除，让新的心跳包从尾部加入，那么如果这个加入的过程卡一点。岂不是就会误...

展开

作者回复: 你好有铭，这种设计下，还必须限制每次移除心跳包的数量（分多次执行），以防止加入过程长时间得不到执行。
而且，在这种极限场景下，必须监控CPU的使用率，如果长期维持在高占用率（可能你的集群规模已经超大，要每秒处理数百万心跳包），那么应当通过扩容更多的CPU核、分布式系统等其他方案来解决，这已经不是单台机器能解决的了。

3

🎧重返归途
2020-05-27

你好，MTU和MSS 有什么关联么

展开

作者回复: MSS必须小于MTU，MSS应用在多个网络构成的TCP链路中，而MTU应用在一个网络中

lpzheng
2020-05-27

如果宕机的主机一直没有发心跳包，那就是发现不了宕机了吗?

作者回复: 宕机并不是指机器断电、断网等故障，如果应用进程出现了问题，不上报心跳，虽然操作系统仍然正常，但业务也需要做容灾迁移

1

Ken
2020-05-27

烧脑，还在消化，容我二刷回来评论😂

展开

作者回复: ^_^