工作线程数究竟要设置为多少 | 架构师之路

一、需求缘起

Web-Server通常有个配置，**最大工作线程数**，后端服务一般也有个配置，工作线程池的**线程数量**，这个线程数的配置不同的业务架构师有不同的经验值，有些业务设置为CPU核数的2倍，有些业务设置为CPU核数的8倍，有些业务设置为CPU核数的32倍。

“工作线程数”的设置依据是什么，到底设置为多少能够最大化CPU性能，是本文要讨论的问题。

二、共性认知

在进行进一步深入讨论之前，先以提问的方式就一些共性认知达成一致。

问：工作线程数是不是设置的越大越好？

答：肯定不是的

- 服务器CPU核数有限，能够同时并发的线程数有限，单核CPU设置10000个工作线程没有意义
- 线程切换是有开销的，如果线程切换过于频繁，反而会使性能降低

问：调用sleep()函数的时候，线程是否一直占用CPU？

答：不占用，等待时会把CPU让出来，给其他需要CPU资源的线程使用。

不止sleep()函数，在进行一些阻塞调用时，例如网络编程中的：

- **阻塞accept()，等待客户端连接**
- **阻塞recv()，等待下游回包**

都不占用CPU资源。

问：单核CPU，设置多线程有意义么，是否能提高并发性能？

答：即使是单核，使用多线程也是有意义的，大多数情况也能提高并发

- **多线程编码可以让代码更加清晰，例如：IO线程收发包，Worker线程进行任务处理，Timeout线程进行超时检测**
- 如果有一个任务一直占用CPU资源在进行计算，此时增加线程并不能增加并发，例如以下代码会一直占用CPU，并使得CPU占用率达到100%：

```
while(1){ i++; }
```

- 通常来说，Worker线程一般不会一直占用CPU进行计算，此时即使CPU是单核，增加Worker线程也能够提高并发，因为这个线程在休息的时候，其他的线程可以继续工

作

三、常见服务线程模型

了解常见的服务线程模型，有助于理解服务并发的原理，一般来说互联网常见的服务线程模型有两种：

- IO线程与工作现场通过任务队列解耦
- 纯异步

IO线程与工作线程通过队列解耦类模型

如上图，大部分Web-Server与服务框架都是使用这样的一种“IO线程与Worker线程通过队列解耦”类线程模型：

- 有少数几个IO线程监听上游发过来的请求，并进行收发包（生产者）
- 有一个或者多个任务队列，作为IO线程与Worker线程异步解耦的数据传输通道（临界资源）
- 有多个工作线程执行正真的任务（消费者）

这个线程模型应用很广，符合大部分场景，这个线程模型的特点是，**工作线程内部是同步阻塞执行任务的**（回想一下tomcat线程中是怎么执行Java程序的，dubbo工作线程中是怎么执行任务的），因此可以通过增加Worker线程数来增加并发能力，今天要讨论的重点是“该模型Worker线程数设置为多少能达到最大的并发”。

纯异步线程模型

没有阻塞，这种线程模型只需要设置很少的线程数就能够做到很高的吞吐量，该模型的缺点是：

- 如果使用单线程模式，难以利用多CPU多核的优势
- 程序员更习惯写同步代码，callback的方式对代码的可读性有冲击，对程序员的要求也更高
- 框架更复杂，往往需要server端收发组件，server端队列，client端收发组件，client端队列，上下文管理组件，有限状态机组件，超时管理组件的支持

文章《RPC-client异步收发核心细节？》中有更详细的介绍，however，这个模型不是今天讨论的重点，

四、工作线程的工作模式

了解工作线程的工作模式，对量化分析线程数的设置非常有帮助：

上图是一个典型的工作线程的处理过程，从开始处理start到结束处理end，该任务的处理共有7个步骤：

- 从工作队列里拿出任务，进行一些**本地**初始化计算，例如http协议分析、参数解析、参数校验等
- **访问cache拿一些数据**
- 拿到cache里的数据后，再进行一些**本地**计算，这些计算和业务逻辑相关
- 通过**RPC调用下游service**再拿一些数据，或者让下游service去处理一些相关的任务
- RPC调用结束后，再进行一些**本地**计算，怎么计算和业务逻辑相关
- **访问DB进行一些数据操作**
- 操作完数据库之后做一些收尾工作，同样这些收尾工作也是**本地**计算，和业务逻辑相关

分析整个处理的时间轴，会发现：

- 其中1，3，5，7步骤中（上图中粉色时间轴），线程进行本地业务逻辑计算时**需要占用CPU**
- 而2，4，6步骤中（上图中橙色时间轴），访问cache、service、DB过程中线程处于一个等待结果的状态，**不需要占用CPU**，进一步的分解，这个“等待结果”的时间共分为三部分：

2.1）请求在网络上传输到下游的cache、service、DB
2.2）下游cache、service、DB进行任务处理
2.3）cache、service、DB将报文在网络上传回工作线程

五、量化分析并合理设置工作线程数

最后一起来回答工作线程数设置为多少合理的问题。

通过上面的分析，Worker线程在执行的过程中，有一部计算时间需要占用CPU，另一部分等待时间不需要占用CPU，通过量化分析，例如**打日志进行统计**，可以统计出整个**Worker线程执行过程中这两部分时间的比例**，例如：

- 执行计算，占用CPU的时间（粉色时间轴）是100ms
- 等待时间，不占用CPU的时间（橙色时间轴）也是100ms

得到的结果是，这个线程计算和等待的时间是1：1，即有50%的时间在计算（占用CPU），50%的时间在等待（不占用CPU）：

- 假设此时是单核，则设置为2个工作线程就可以把CPU充分利用起来，让CPU跑到100%
- 假设此时是N核，则设置为2N个工作现场就可以把CPU充分利用起来，让CPU跑到N*100%

结论：
N核服务器，通过执行业务的单线程分析出本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N*(x+y)/x，能让CPU的利用率最大化。

经验：
一般来说，非CPU密集型的业务（加解密、压缩解压缩、搜索排序等业务是CPU密集型的业务），瓶颈都在后端数据库访问或者RPC调用，本地CPU计算的时间很少，所以设置几十或者几百个工作线程是能够提升吞吐量的。

六、总结

- 线程数不是越多越好
- sleep()不占用CPU
- 单核设置多线程不但能使得代码清晰，还能提高吞吐量
- 站点和服务最常用的线程模型是“IO线程与工作现场通过任务队列解耦”，此时设置多工作线程可以提升吞吐量
- N核服务器，通过日志分析出任务执行过程中，本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N*(x+y)/x，能让CPU的利用率最大化