要想合理的配置线程池的大小，首先得分析任务的特性，可以从以下几个角度分析：

1. 任务的性质：CPU密集型任务、IO密集型任务、混合型任务。
2. 任务的优先级：高、中、低。
3. 任务的执行时间：长、中、短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接等。

性质不同的任务可以交给不同规模的线程池执行。

对于不同性质的任务来说，CPU密集型任务应配置尽可能小的线程，如配置CPU个数+1的线程数，IO密集型任务应配置尽可能多的线程，因为IO操作不占用CPU，不要让CPU闲下来，应加大线程数量，如配置两倍CPU个数+1，而对于混合型的任务，如果可以拆分，拆分成IO密集型和CPU密集型分别处理，前提是两者运行的时间是差不多的，如果处理时间相差很大，则没必要拆分了。

若任务对其他系统资源有依赖，如某个任务依赖数据库的连接返回的结果，这时候等待的时间越长，则CPU空闲的时间越长，那么线程数量应设置得越大，才能更好的利用CPU。  
当然具体合理线程池值大小，需要结合系统实际情况，在大量的尝试下比较才能得出，以上只是前人总结的规律。

在这篇[如何合理地估算线程池大小？](http://ifeve.com/how-to-calculate-threadpool-size/)文章中发现了一个估算合理值的公式

最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数目

比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)\*8=32。这个公式进一步转化为：

最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）\* CPU数目

可以得出一个结论：  
**线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。**  
以上公式与之前的CPU和IO密集型任务设置线程数基本吻合。

并发编程网上的一个问题  
高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？  
（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换  
（2）并发不高、任务执行时间长的业务要区分开看：  
　　a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以适当加大线程池中的线程数目，让CPU处理更多的业务  
　　b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换  
（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

**如何合理地估算线程池大小？**

这个问题虽然看起来很小，却并不那么容易回答。大家如果有更好的方法欢迎赐教，先来一个天真的估算方法：假设要求一个系统的TPS（Transaction Per Second或者Task Per Second）至少为20，然后假设每个Transaction由一个线程完成，继续假设平均每个线程处理一个Transaction的时间为4s。那么问题转化为：

**如何设计线程池大小，使得可以在1s内处理完20个Transaction？**

计算过程很简单，每个线程的处理能力为0.25TPS，那么要达到20TPS，显然需要20/0.25=80个线程。

很显然这个估算方法很天真，因为它没有考虑到CPU数目。一般服务器的CPU核数为16或者32，如果有80个线程，那么肯定会带来太多不必要的线程上下文切换开销。  
  
再来第二种简单的但不知是否可行的方法（N为CPU总核数）：

* 如果是CPU密集型应用，则线程池大小设置为N+1
* 如果是IO密集型应用，则线程池大小设置为2N+1

如果一台服务器上只部署这一个应用并且只有这一个线程池，那么这种估算或许合理，具体还需自行测试验证。

接下来在这个文档：服务器性能IO优化 中发现一个估算公式：

最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数目

比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)\*8=32。这个公式进一步转化为  
：

最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）\* CPU数目

可以得出一个结论：

**线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。**

上一种估算方法也和这个结论相合。

一个系统最快的部分是CPU，所以决定一个系统吞吐量上限的是CPU。增强CPU处理能力，可以提高系统吞吐量上限。但根据短板效应，真实的系统吞吐量并不能单纯根据CPU来计算。那要提高系统吞吐量，就需要从“系统短板”（比如网络延迟、IO）着手：

* 尽量提高短板操作的并行化比率，比如多线程下载技术
* 增强短板能力，比如用NIO替代IO

第一条可以联系到Amdahl定律，这条定律定义了串行系统并行化后的加速比计算公式：

加速比=优化前系统耗时 / 优化后系统耗时

加速比越大，表明系统并行化的优化效果越好。Addahl定律还给出了系统并行度、CPU数目和加速比的关系，加速比为Speedup，系统串行化比率（指串行执行代码所占比率）为F，CPU数目为N：

Speedup <= 1 / (F + (1-F)/N)

当N足够大时，串行化比率F越小，加速比Speedup越大。

写到这里，我突然冒出一个问题。

**是否使用线程池就一定比使用单线程高效呢？**

答案是否定的，比如Redis就是单线程的，但它却非常高效，基本操作都能达到十万量级/s。从线程这个角度来看，部分原因在于：

* 多线程带来线程上下文切换开销，单线程就没有这种开销
* 锁

当然“Redis很快”更本质的原因在于：Redis基本都是内存操作，这种情况下单线程可以很高效地利用CPU。而多线程适用场景一般是：存在相当比例的IO和网络操作。

所以即使有上面的简单估算方法，也许看似合理，但实际上也未必合理，都需要结合系统真实情况（比如是IO密集型或者是CPU密集型或者是纯内存操作）和硬件环境（CPU、内存、硬盘读写速度、网络状况等）来不断尝试达到一个符合实际的合理估算值。