

## 12 | 高性能优化：单机Java极致优化

2021-10-22 余志东

《手把手带你搭建秒杀系统》

课程介绍 >



讲述：余志东

时长 13:24 大小 12.29M



你好，我是志东，欢迎和我一起从零打造秒杀系统。

今天这节课我们主要是聊一聊和 Java 相关的一些技术点的优化方向，包括 Tomcat、RPC 框架、JVM 以及 CDN 等。但在开始之前呢，我们先来说个基本知识点，那就是关于程序代码的两种运行模式，即 **CPU 密集型与 IO 密集型**。

CPU 密集型操作，顾名思义就是需要持续依赖 CPU 资源来执行的操作，比如各种逻辑计算、解析、判断等等。在这种情况下，我们的优化方向是尽可能地利用多核 CPU 资源，并且避免让 CPU 做无效的切换，因为 CPU 已经在不停地工作了，谁来干都一样，同时 CPU 还浪费资源。所以这个时候，我们最好让任务线程数和 CPU 核数保持一致，从而最大限度地利用 CPU 资源。

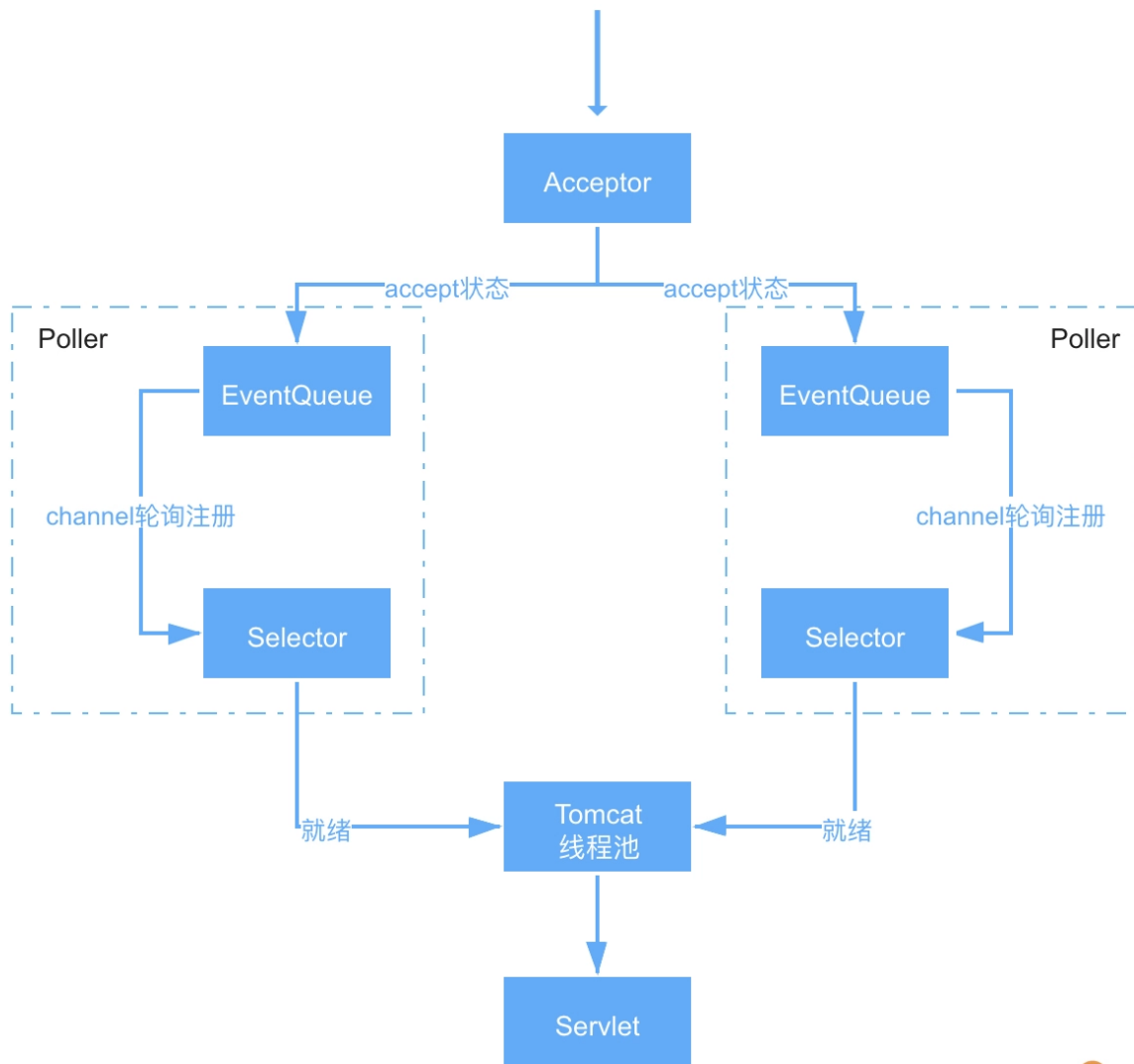


和 CPU 密集型操作相对的，就是 IO 密集型操作了，比如磁盘 IO 或者网络 IO，这个过程操作系统会挂起任务线程，让出 CPU 资源。此时如果任务线程较少，同时 IO 时间相对较长，那可能会出现所有线程都被挂起，然后 CPU 资源都在闲着的情况，所以此时我们需要适当地增加任务线程数量，来提高吞吐量，同时将 CPU 资源利用起来。

那为什么要说这个呢？因为这是做程序优化的基本原则。通过前面课程的学习，我们知道，秒杀系统里有提供两种类型的服务，一个是 Web 服务，一个是 RPC 服务，前者一般提供 HTTP 接口，后者提供 RPC 接口。当然这两种服务我们一般都是通过 Tomcat 来启动发布，但它们两者之间还是有些不同的。Web 服务接受和处理请求走的是 Tomcat 那套线程模型，而 RPC 服务则是根据选择的 RPC 框架的不同而有所变化，所以这节课我们首先来了解一下 Tomcat 相关的知识。

## Tomcat

根据我们以往“知己知彼”的学习方式，先看下 **Tomcat 在 NIO 线程模型下是怎么工作的**，简图如下所示：



简单来说就是：

Tomcat 启动时，会创建一个 Server 端的 Socket，来监控我们配置的端口号；  
 之后使用一个 Acceptor 来接受请求，然后将请求放到一个 Poller 下的事件队列中；  
 Poller 会轮询取出事件队列中的 Channel，并将其注册到自身下的 Selector；  
 而 Selector 也会不停轮询检查就绪的 Channel，然后将其交给 Tomcat 线程池；  
 Tomcat 线程池会拿出一个线程来进行处理，包括解析请求头、请求体等，并将其封装进 HttpServletRequest；  
 最后执行自定义的 Servlet 业务逻辑，执行完毕将响应结果返回。

所以从上图可以看出，所谓的非阻塞，其实就是相对以前的 BIO，Tomcat 不再是用一个线程将一个请求从头处理到尾，而是分阶段来执行了。好处显而易见，那就是提高了系统

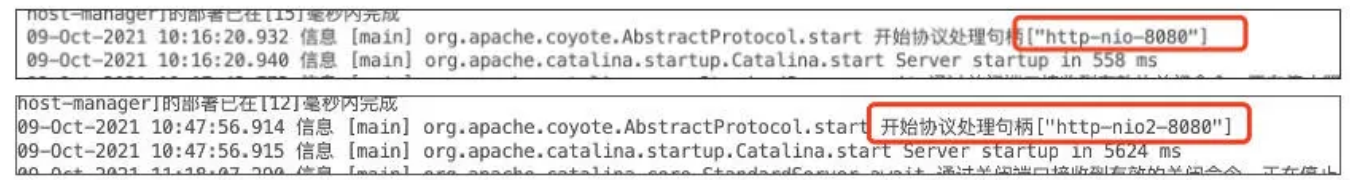
吞吐量。

在了解了 Tomcat 基本原理之后，我们再回过头来看下有什么地方是我们可以入手优化的。先看下 Tomcat 给我们开放了哪些可配置项：

复制代码

```
1 <Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectP
```

上面是 Tomcat 的 Connector 默认配置，首先是端口号，其次是 protocol，也就是上面说到的线程模型。Tomcat 8 之后默认使用的都是 NIO 模式，这个也可以通过我们服务的启动来查看：



如上图所示，就代表分别使用的是 NIO 模式和 NIO2(AIO) 模式，当然还可以选择 BIO 模式以及 APR 模式。具体对比可参考下表：

类型	Protocol值	说明
BIO	org.apache.coyote.http11.Http11Protocol	阻塞模式，如果你的 Tomcat 版本是 8.5.x，将不再支持 BIO 模式，会自动转成 NIO 模式
NIO	org.apache.coyote.http11.Http11NioProtocol	非阻塞，并发性较 BIO 有较大提升
NIO2	org.apache.coyote.http11.Http11Nio2Protocol	异步非阻塞，异步主要体现在 Acceptor 阶段读取网络数据流，不再是等待，而是完成后执行回调
APR	org.apache.coyote.http11.Http11AprProtocol	需要额外安装工具包，优势在于对静态文件和 SSL 的处理

那说完线程模型的选择，从上图中我们可以看到有个 Tomcat 线程池的概念，它是通过哪些配置来控制的呢？这里我们只摘几个重要的配置说一下，详细信息如下表所示：

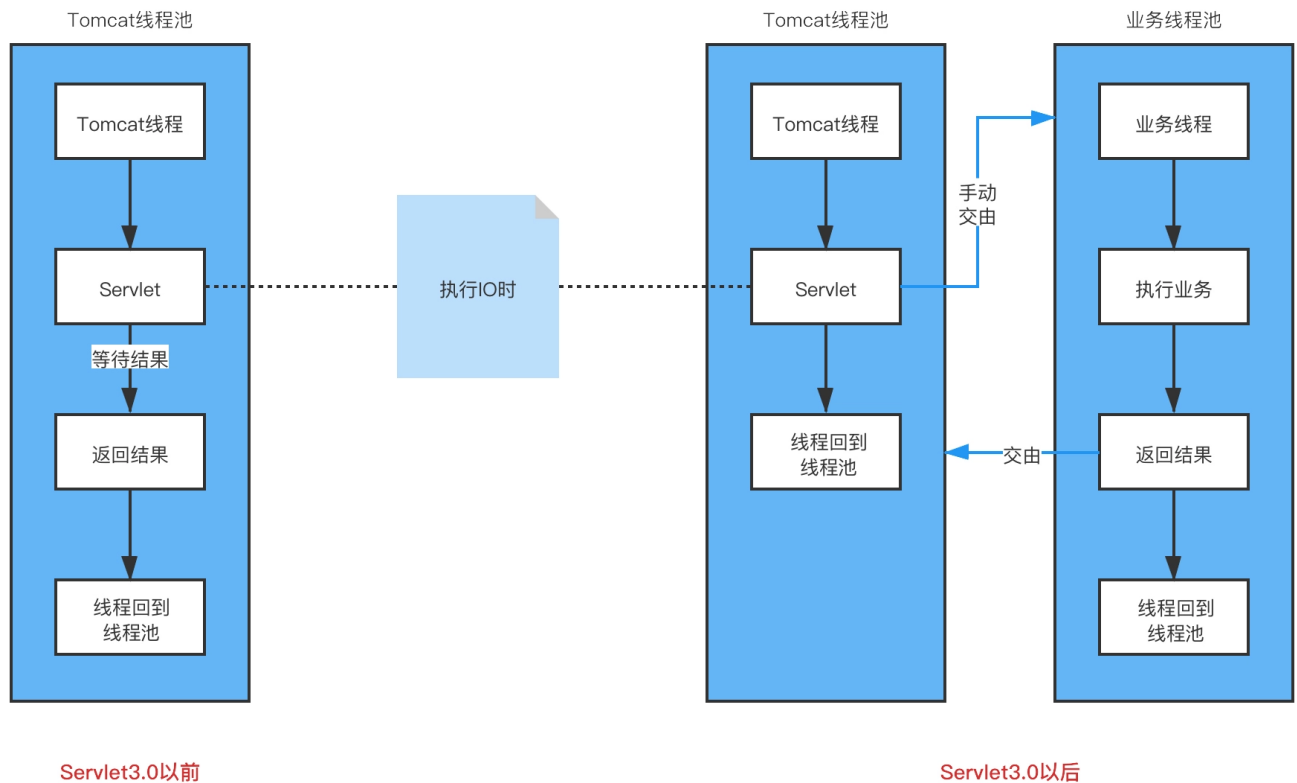
属性名	说明	默认值	推荐配置
maxConnections	最大连接数，达到限制值之后，会继续接受请求，但是会放到等待队列里	8192	8192
keepAliveTimeout	空闲连接的存活时间，默认是使用 connectionTimeout 的设置值，如果设为 0，则该连接变成短连接，根据我们上一章的介绍，这里要使用长连	20s	20s
connectionTimeout	连接超时时间，与 keepAliveTimeout 共用，同时也是读取消息体的超时时间，Tomcat 里默认是设置了 20s	20s	10s
acceptCount	等待队列长度，和 maxConnections 配合使用	100	1000
maxThreads	最大线程数，如果使用了自定义线程池，则该配置忽略	200	150-200
minSpareThreads	类似线程池的核心线程数概念，Tomcat 启动时就创建的线程数，不管是否闲置都不会被销毁	10	50-100

说完了 Tomcat 的配置，这里再简单说说 Servlet 的部分知识。我们都知道 Servlet 从 3.0 开始加入了异步，从 3.1 开始又新增了对 IO 非阻塞的支持，那么这个和 Tomcat 线程模型中提到的异步非阻塞是一个概念吗？这里我们就来捋一捋。

首先从上面的 Tomcat 线程模型图中，我们可以清晰地看到，NIO 或 AIO 的概念是针对请求的接收来说，而 Servlet 的异步非阻塞主要是针对请求的处理，已经是到了 Tomcat 线程池那里了。

我们先来看下 Servlet3.0 前后的变化对比，如下图所示：





概述一下就是，Servlet3.0 之前，Tomcat 线程在执行自定义 Servlet 时，如果过程中发生了 IO，那么 Tomcat 线程只能在那等着结果，这时线程是被挂起的，如果被挂起的多了，自然会影响对其他请求的处理。

所以在 Servlet3.0 之后，支持在这种情况下将这种等待的任务交给一个自定义的业务线程池去做，这样 Tomcat 线程可以很快地回到线程池，处理其他请求。而业务线程在执行完业务逻辑以后，通过调用指定的方法，告诉 Tomcat 线程池接下来可以将业务线程执行的结果返回给调用方，这样就实现了同步转异步的效果。

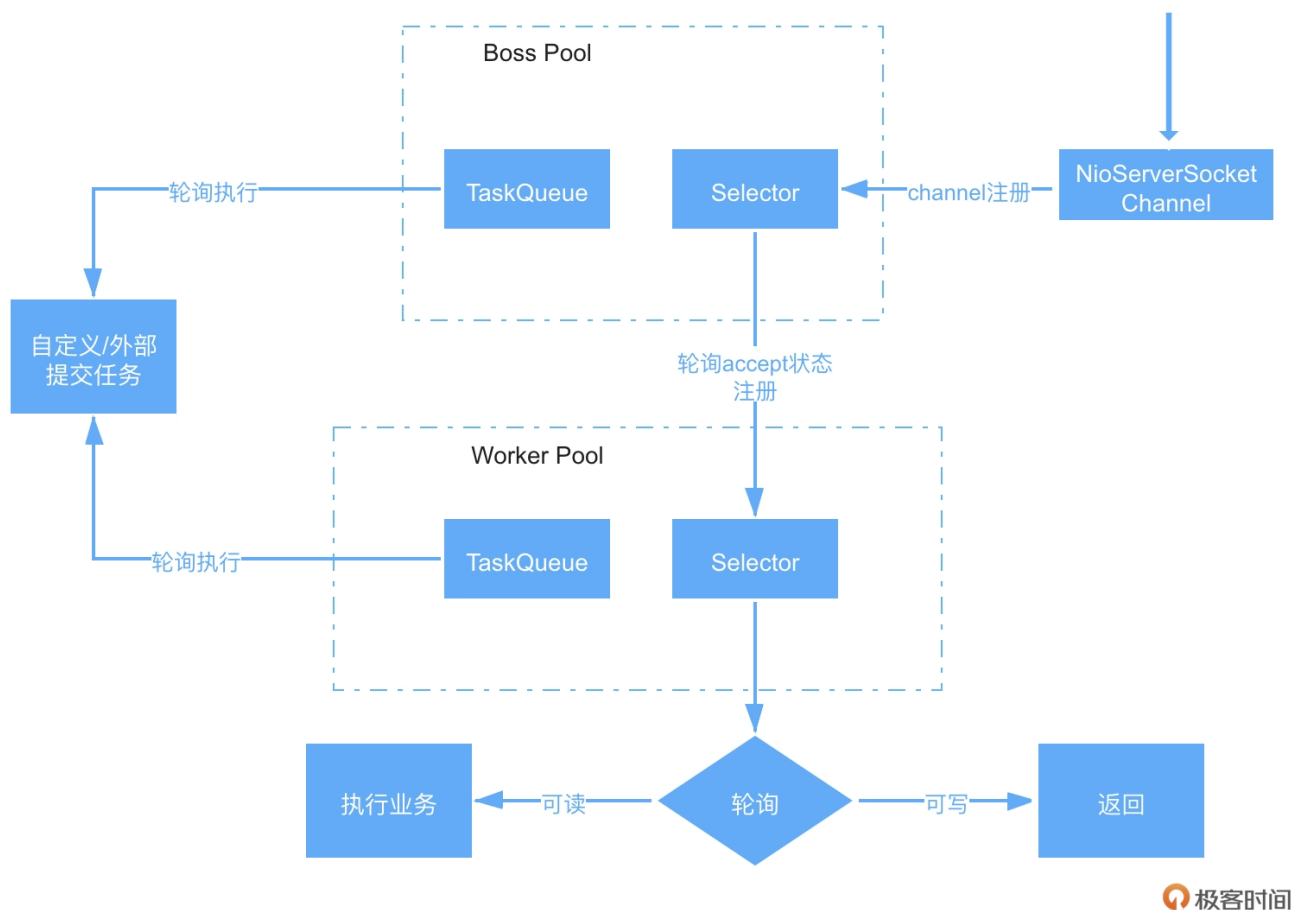
这样做的好处，可能对提高系统的吞吐量有一定帮助，但从 JVM 层面来说，并没有减少工作量。业务线程在执行任务遇到 IO 时，依然会阻塞，现在只是由业务线程池代替了 Tomcat 线程池做了最耗时的那部分工作，这样也许可以将原来的 200 个 Tomcat 线程，拆分成 20 个 Tomcat 线程、180 个业务线程来配合工作。这里原生 Servlet 以及 SpringMVC 对异步功能支持的测试代码，你可以看 GitHub 代码库中的 AsyncServlet 类和 TestAsyncController 类，相信你一看就明白了。

接着我们再聊一下 Servlet3.1 的非阻塞，这块简单来说，就是针对请求消息体的读取，这是个 IO 过程，以前是阻塞式读取，现在支持非阻塞读取了。实现的大致原理就是在读取数据时，新增一个监听事件，在读取完成后由 Tomcat 线程执行回调。

在了解了 Tomcat 线程模型之后，我们接着再说下 RPC 框架相关的知识。

## RPC 框架

虽然 RPC 服务处理请求的过程，会依据选用的 RPC 框架而有所不同，但绝大部分 RPC 框架底层使用的都是 Netty，而 Netty 则是基于 NIO 开发的一种网络通信框架，支持多种通信协议，其服务端线程模型简略图如下所示：



极客时间

简单描述就是：

在服务启动时，会创建一个 Server 端 Socket，监控我们配置的端口号；

然后将 NioServerSocketChannel 注册到 Boss Pool 中的一个 Selector 上；

再之后对 Selector 做轮询，将就绪状态的连接封装成 NioSocketChannel 并注册到 Worker Pool 下的一个 Selector 上；

而 Worker Pool 下的 Selector 也是同样轮询，找出可读和可写状态的分别执行不同操作。

同时两个 Pool 中都有任务队列，是不同场景下用户自定义或外部通过特定方式提交过去的任务，都会被依次执行。

所以当我们的应用只提供 RPC 服务时，我们可以将 Tomcat 的核心线程池配置，也就是 minSpareThreads 配置成 1，因为用不到。而我们主要需要调整的是 RPC 框架的相关配置，以 Dubbo 为例，我们看下 <dubbo:protocol> 的主要配置项：

配置项	说明	默认值	推荐值
dispatcher	协议的消息派发方式，用于指定线程模型：all,direct,message,execution,connection	all	all或者message
threadpool	线程池类型, 可选：fixed/cached	fix	fix
threads	业务线程池大小，也需要看dispatcher 配置，如果是 direct，就用不到这个了	200	150-200
iothreads	IO 线程池大小	CPU核数+1	-
queues	等待队列，业务线程池满后，新来事件放入该队列	0	-

在 Netty 中，虽然只有一个 Worker Pool，但会做两种类型的事情，一个是做 IO 处理，包括请求消息的读写，另一个是做业务逻辑处理。

而 Dubbo 将其分成了两个线程池，也就是上面表格中的两个线程池配置。这两个线程池做的事情，会根据 Dispatcher 的配置而有所不同。Netty 是以事件驱动的形式来工作的，像请求、响应、连接、断开、异常等操作都是事件；而 Dubbo 中的 Dispatcher 就是将不同的事件类型分给不同的线程池来处理，如果你感兴趣的话可以去看下 Dubbo 中 WrappedChannelHandler 类的 5 个实现类，分别对应 Dispatcher 的 5 个选项。

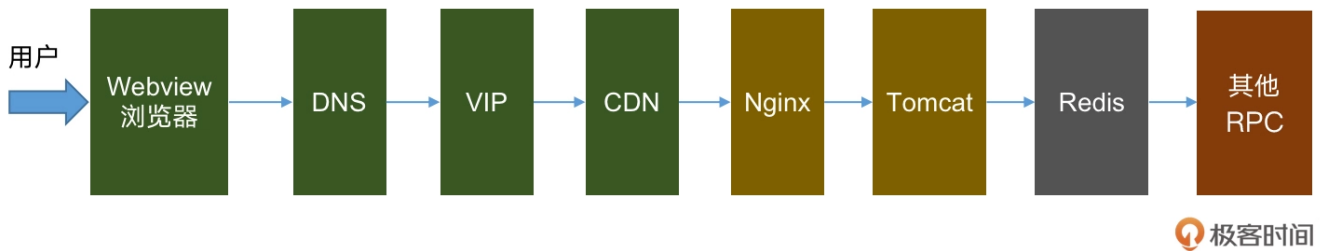
最后一个配置项 Queues，这个默认值是 0，也就是不接受等待，如果没有空闲线程处理任务，将会直接返回。这个得和客户端配置配合使用，如果这里配置了 0，那客户端最好配置重试。

讲完了两种服务的底层线程模型之后，我们再来介绍一下静态资源相关的优化。

## 静态资源



我们知道在秒杀系统中，客户端与服务端既有动态数据交互，也有静态数据交互，而我们做系统优化有个基本的原则，即**前后端交互越少，数据越小，链路越短，数据离用户越近，响应就越快。**

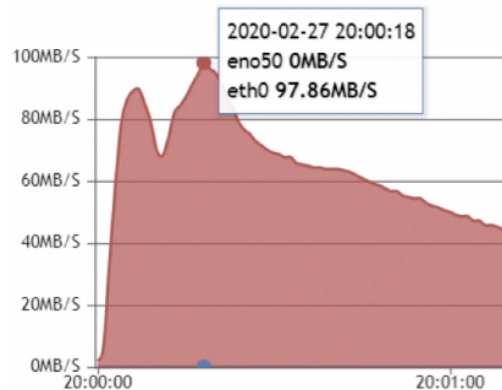
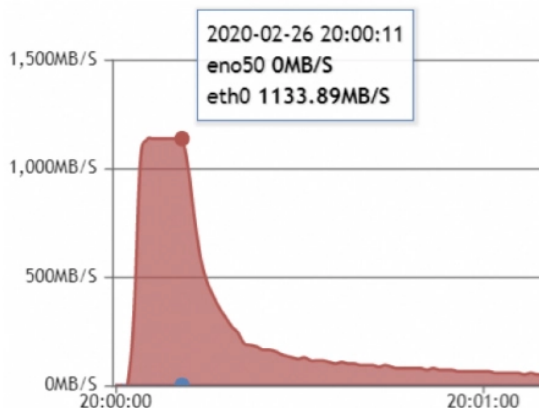


基于这个原则，针对以上的静态数据，我们就可以把静态文件 CDN 化，资源前移到全国各地的 CDN 节点上，用户秒杀的时候就近进行下载，就不需要都挤到中心的 Tomcat 服务器上了。

静态资源前移，大家平常也会做，感受比较深的是不是就是客户端的页面加载更快了，但除了性能的提升外，其实它对系统稳定也至关重要。

试想一下，当几百万人同时来拉取这些较大的资源文件时，对中心的 Tomcat 服务器以及公司的网络带宽都是巨大的压力。京东当初在进行口罩抢购的时候，这些静态资源就差点把公司的出口带宽打满，影响交易大盘，后来紧急扩容才避免了危机。

另外，这些静态资源对 Tomcat 所在物理机的网卡挑战也很大，京东在资源 CDN 化前，物理机的万兆网卡曾被打满，后来经过优化之后，网卡的流量只有原来的 10% 了。



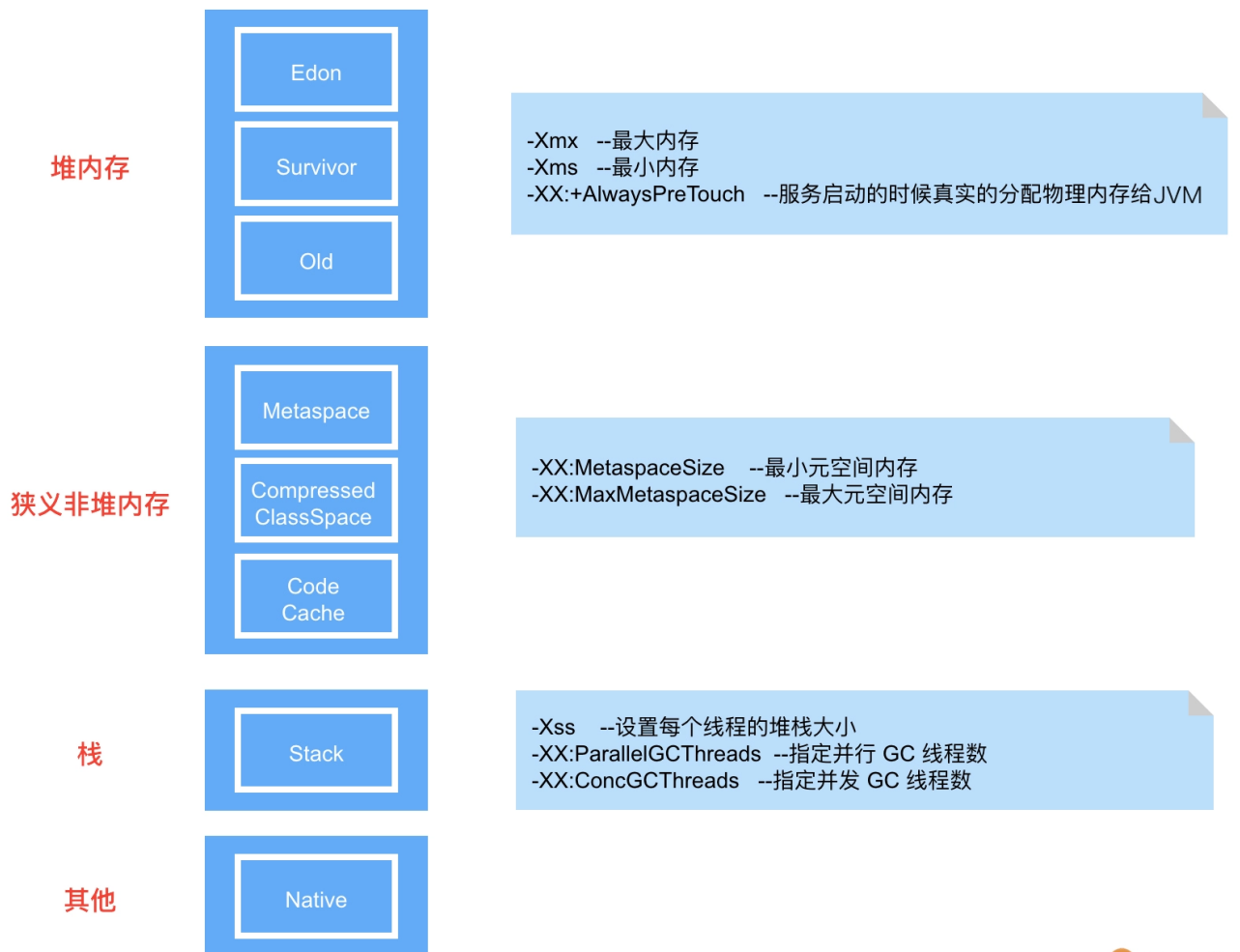
极客时间

在最后，我们再说下 Java 运行的基础环境，JVM 相关的知识以及优化。

# JVM

这里如果你对一些基本概念，比如 JVM 内存结构、GC 原理、垃圾收集器类型等还不太了解，那建议你先了解一下，会有事半功倍的效果。这块的内容比较多，又比较重要，但我们没办法一一展开，只说最核心的优化点。

先看个 JVM 内存模型以及常用配置，如下图所示：



极客时间

其实针对 JVM 的优化，我们最关心的无非就两个问题，一个是垃圾回收器怎么选择，另一个就是对选择的垃圾回收器如何做优化。这里我们分别讲一下。

对于垃圾回收器的选择，是需要分业务场景的。如果我们提供的服务对响应时间敏感，并且堆内存能够给到 8G 以上的，那建议选择 G1；堆内存较小或 JDK 版本较低的，可以选择 CMS。相反如果对响应时间不敏感，追求一定的吞吐量的，则建议选择 ParallelGC，同时这也是 JDK8 的默认垃圾回收器。

**选择完垃圾回收器之后，接下来就针对不同的垃圾回收器，分别做不同的参数优化。**

首先是 ParallelGC，其主要配置参数如下：

配置项	说明	建议
-Xmx	最大堆内存	容器内存的一半
-Xms	最小堆内存	容器内存的一半
-XX:MetaspaceSize	元空间内存	256M~512M
-XX:ParallelGCThreads	指定并行 GC 线程数	容器 CPU 核数
-XX:ConcGCThreads	指定并发 GC 线程数	容器 CPU 核数 / 2

然后是 CMS，在 ParallelGC 配置参数的基础上增加以下配置：

配置项	说明	建议
XX:+UseConcMarkSweepGC	启用 CMS 垃圾回收器	-
-Xmn	设置年轻代大小，配置了 ParallelGCThreads 后必须配置此参数	堆内存的 1/3~1/2
-XX:+UseCMSInitiatingOccupancyOnly	用设定的回收阈值	-
-XX:CMSInitiatingOccupancyFraction	即达到设置的阈值时触发 GC	70

再说下 G1 的优化配置（在使用了 G1 的情况下，就不要设置 -Xmn 和 XX:NewRatio 了），同样是在 ParallelGC 配置参数的基础上增加以下配置：

配置项	说明	建议
-XX:+UseG1GC	使用 G1 回收器	-
-XX:G1HeapRegionSize	堆内存的每个 region 的大小	8m

因为我们秒杀的业务场景更适合选择 G1 来做垃圾回收器，那这里也给一个在 8 核 16G 容器下的 JVM 配置，具体如下：

```
1 -Xms8192m -Xmx8192m -XX:MaxMetaspaceSize=512m -XX:+UseG1GC -XX:ParallelGCThrea
```

## 总结

今天主要围绕着 Java，对与其息息相关的 Tomcat、JVM、RPC 框架以及静态资源的优化，做了分析和讲解。

对于 Tomcat 的优化，在秒杀的特定业务场景下针对线程模型的选择，从理论和实际压测上看，NIO2 比 NIO 是有吞吐量的提升，但不是很大，如果为了省事，选择默认的 NIO 即可。而 APR 的话，因为我们静态资源都上到 CDN 了，并且 Web 服务并不直接对外（请求由 Nginx 转发过来），也不要求是 HTTPS 方式，所以这里也不考虑了，和线程池相关的配置，最好按照这节课中的建议做适当的调整。

同时我们也提到了 Servlet 在 3.0 和 3.1 版本提供的异步非阻塞功能，由于秒杀的接口入参不涉及文件之类的较大消息体，所以 IO 非阻塞可以不用。而异步功能这块，其实可以有更好的选择，那就是 Vertx 技术，这也是我们在下节课中，将会单独介绍的一种异步化编程思想技术。

而对于 RPC 框架，我们主要介绍了基于 NIO 开发的一种网络通信框架 Netty，了解了 Netty 主要使用两个池子，即 Boss Pool 和 Worker Pool 来实现 Reactor 模式。同时选择了一个具体的 RPC 框架 Dubbo，来做了详细的配置优化讲解。

在聊完了两种服务的底层线程模型与优化后，我们介绍了静态资源的优化方案，即将静态资源上到 CDN，以减轻对秒杀域名流量的压力，同时可以依靠 CDN 的全国部署，快速加载到对应的静态资源。

另外，我们还提到了 Java 运行的环境 JVM，包括垃圾回收器的选择与优化，即如果我们提供的服务对响应时间敏感，并且堆内存能够给到 8G 以上的，那就选择 G1；而堆内存较小或 JDK 版本较低的，可以选择 CMS。相反如果对响应时间不敏感，追求一定的吞吐量的，则建议选择 ParallelGC。同时针对不同的垃圾回收器，也给出了对应的优化配置。


当然以上所有的优化建议，在调整后都需要做实际业务场景下的压测，毕竟实践才是检测真理的唯一标准！

## 思考题

这节课我们介绍了通过 Tomcat 发布的 Web 服务和 RPC 服务，两者走的底层线程模型是不同的，如果我们的服务既提供 HTTP 接口，也提供 RPC 接口，我们该通过何种方式才能将二者的相互影响降至最低呢？

期待你的思考，也欢迎在留言区中分享讨论。我们下节课再见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 高性能优化：物理机极致优化

下一篇 13 | 优化番外篇：Vertx 介绍及快速入门

## 1024 活动特惠

# VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999 



精选留言 (1)

写留言



清风

2021-10-26

可以部署两个服务，将http服务和RPC服务分开，http服务使用独立的域名对外服务，Rpc使用独立的group对外服务

展开

