主讲老师：Fox

课前须知：

- 理解I/O模型的本质：是为了解决什么问题的，为什么会有不同的I/O模型设计，重点理解I/O模型中的IO多路复用和异步IO。
- Java NIO编程复杂不要求掌握，重点是理解网络编程模型中的主从Reactor多线程模型在Tomcat中的实现

```
文档：2.Tomcat 线程模型详解&性能调优.note
链接：http://note.youdao.com/noteshare?id=f1960c96070b09982a6002c17a91bfe7&sub=7651077F416843C0BF80C70B4CBBB2F1
```

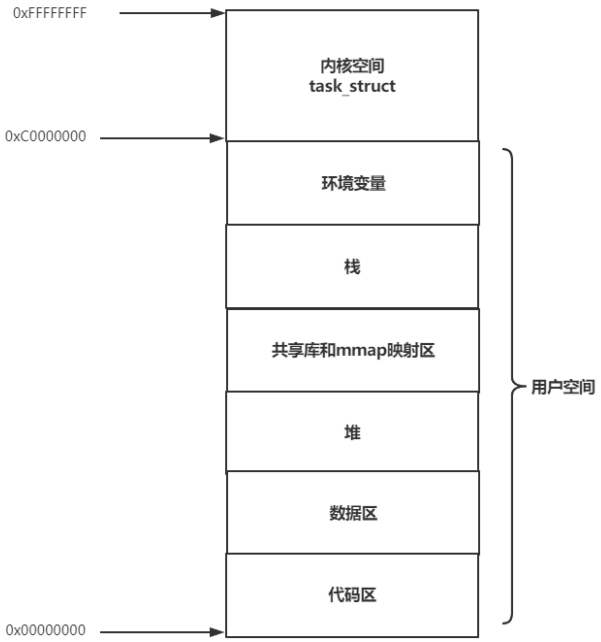
Tomcat I/O模型详解

I/O模型详解

内核空间和用户空间

思考：物理内存是有限的（比如16G内存），怎么把有限的内存分配给不同的进程？

Linux 给每个进程虚拟出一块很大的地址空间，比如 32 位机器上进程的虚拟内存地址空间是 4GB，从 0x00000000 到 0xFFFFFFFF。但这 4GB 并不是真实的物理内存，而是进程访问到了某个虚拟地址，如果这个地址还没有对应的物理内存页，就会产生缺页中断，分配物理内存，MMU（内存管理单元）会将虚拟地址与物理内存页的映射关系保存在页表中，再次访问这个虚拟地址，就能找到相应的物理内存页。每个进程的这 4GB 虚拟地址空间分布如下图所示：



> 用户空间从低到高依次是代码区、数据区、堆、共享库与 mmap 内存映射区、栈、环境变量。其中堆向高地址增长，栈向低地址增长。
> 用户空间上还有一个共享库和 mmap 映射区，Linux 提供了内存映射函数 mmap， 它可将文件内容映射到这个内存区域，用户通过读写这段内存，从而实现对文件的读取和修改，无需通过 read/write 系统调用来读写文件，省去了用户空间和内核空间之间的数据拷贝，Java 的

> MappedByteBuffer 就是通过它来实现的；用户程序用到的系统共享库也是通过 mmap 映射到了这个区域。
>
> task_struct结构体本身是分配在内核空间，它的vm_struct成员变量保存了各内存区域的起始和终止地址，此外task_struct中还保存了进程的其他信息，比如进程号、打开的文件、创建的 Socket 以及 CPU 运行上下文

进程的虚拟地址空间总体分为用户空间和内核空间，低地址上的 3GB 属于用户空间，高地址的 1GB 是内核空间，这是基于安全上的考虑，用户程序只能访问用户空间，内核程序可以访问整个进程空间，并且只有内核可以直接访问各种硬件资源，比如磁盘和网卡。

那用户程序需要访问这些硬件资源该怎么办呢？答案是通过系统调用，系统调用可以理解为内核实现的函数，比如应用程序要通过网卡接收数据，会调用 Socket 的 read 函数：

```
ssize_t read(int fd,void *buf,size_t nbyte)
```

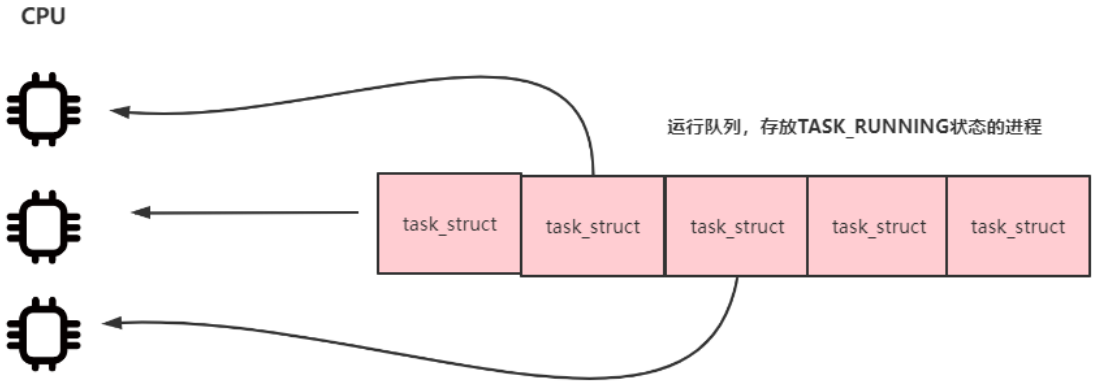
CPU 在执行系统调用的过程中会从用户态切换到内核态，CPU 在用户态下执行用户程序，使用的是用户空间的栈，访问用户空间的内存；当 CPU 切换到内核态后，执行内核代码，使用的是内核空间上的栈。

在 Linux 中，线程是一个轻量级的进程，轻量级说的是线程只是一个 CPU 调度单元，因此线程有自己的task_struct结构体和运行栈区，但是线程的其他资源都是跟父进程共用的，比如虚拟地址空间、打开的文件和 Socket 等。

阻塞与唤醒

思考：当用户线程发起一个阻塞式的 read 调用，数据未就绪时，线程就会阻塞，那阻塞具体是如何实现的呢？

Linux 内核将线程当作一个进程进行 CPU 调度，内核维护了一个可运行的进程队列，所有处于TASK_RUNNING状态的进程都会被放入运行队列中，本质是用双向链表将task_struct链接起来，排队使用 CPU 时间片，时间片用完重新调度 CPU。所谓调度就是在可运行进程列表中选择一个进程，再从 CPU 列表中选择一个可用的 CPU，将进程的上下文恢复到这个 CPU 的寄存器中，然后执行进程上下文指定的下一条指令。

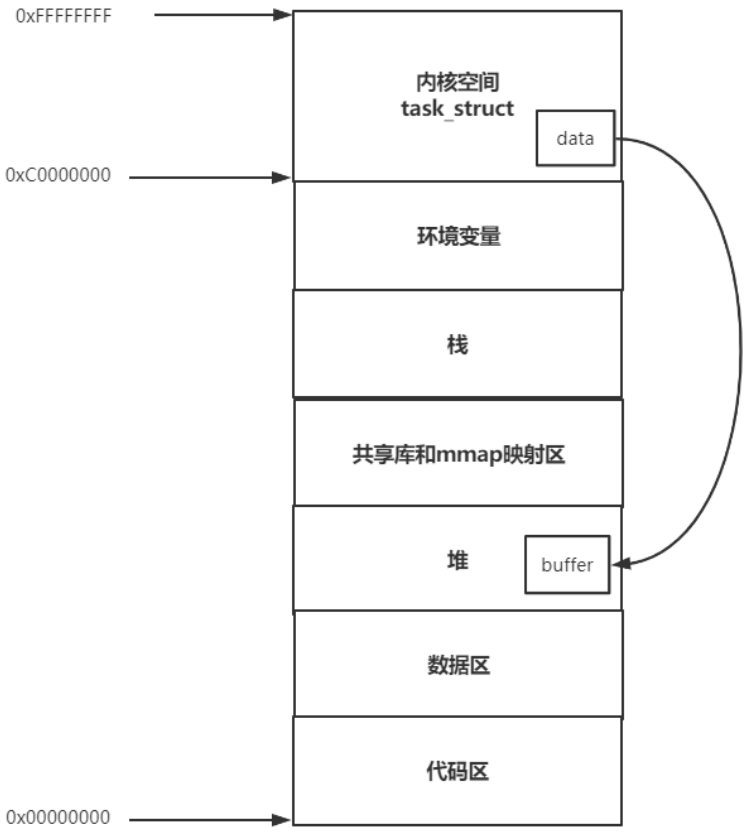


而阻塞的本质就是将进程的task_struct移出运行队列，添加到等待队列，并且将进程的状态的置为TASK_UNINTERRUPTIBLE或者TASK_INTERRUPTIBLE，重新触发一次 CPU 调度让出 CPU。

思考：线程是如何唤醒的呢?

线程在加入到等待队列的同时向内核注册了一个回调函数，告诉内核我在等待这个 Socket 上的数据，如果数据到了就唤醒我。这样当网卡接收到数据时，产生硬件中断，内核再通过调用回调函数唤醒进程。唤醒的过程就是将进程的task_struct从等待队列移到运行队列，并且将task_struct的状态置为TASK_RUNNING，这样进程就有机会重新获得 CPU 时间片。

这个过程中，内核还会将数据从内核空间拷贝到用户空间的堆上。



当 read 系统调用返回时，CPU 又从内核态切换到用户态，继续执行 read 调用的下一行代码，并且能从用户空间上的 Buffer 读到数据了。

Socket Read 系统调用的过程

以Linux操作系统为例，一次socket read 系统调用的过程：

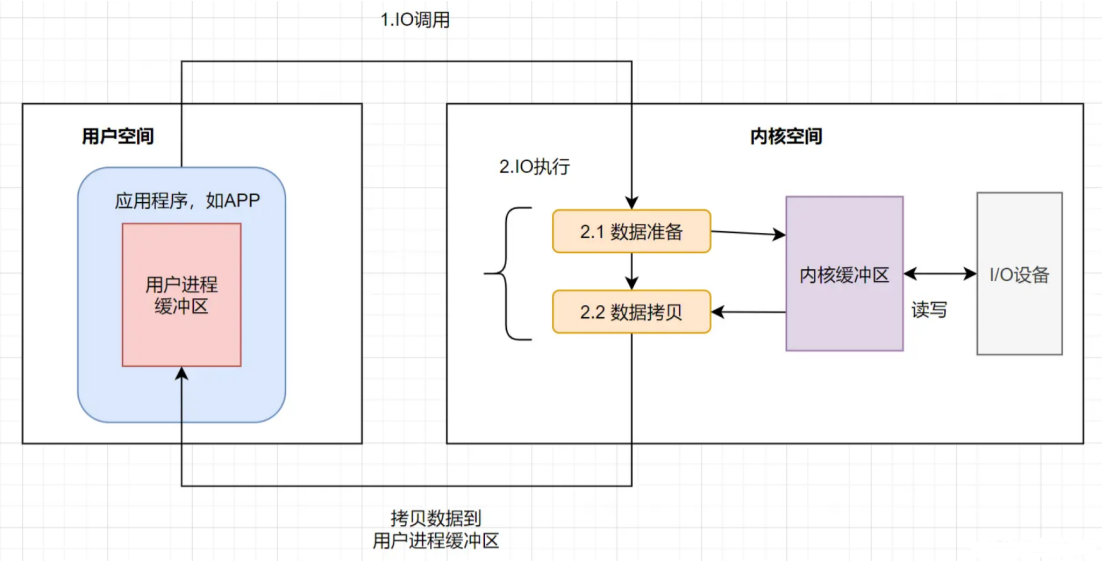
- 首先 CPU 在用户态执行应用程序的代码，访问进程虚拟地址空间的用户空间；
- read 系统调用时 CPU 从用户态切换到内核态，执行内核代码，内核检测到 Socket 上的数据未就绪时，将进程的task_struct结构体从运行队列中移到等待队列，并触发一次 CPU 调度，这时进程会让出 CPU；
- 当网卡数据到达时，内核将数据从内核空间拷贝到用户空间的 Buffer，接着将进程的task_struct结构体重新移到运行队列，这样进程就有机会重新获得 CPU 时间片，系统调用返回，CPU 又从内核态切换到用户态，访问用户空间的数据。

总结

当用户线程发起 I/O 调用后，网络数据读取操作会经历两个步骤：

- 用户线程等待内核将数据从网卡拷贝到内核空间。
- 内核将数据从内核空间拷贝到用户空间（应用进程的缓冲区）。

各种 I/O 模型的区别就是：它们实现这两个步骤的方式是不一样的。



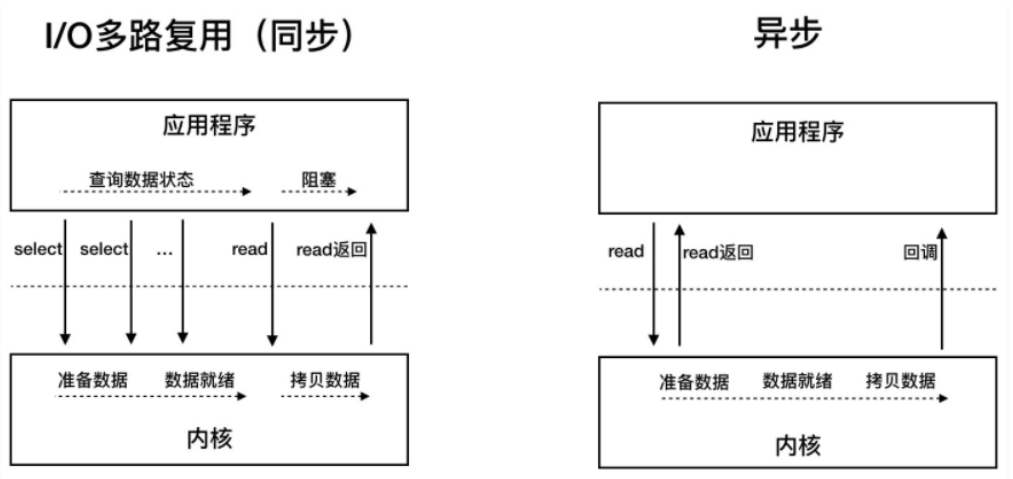
Unix(linux)下5种I/O模型

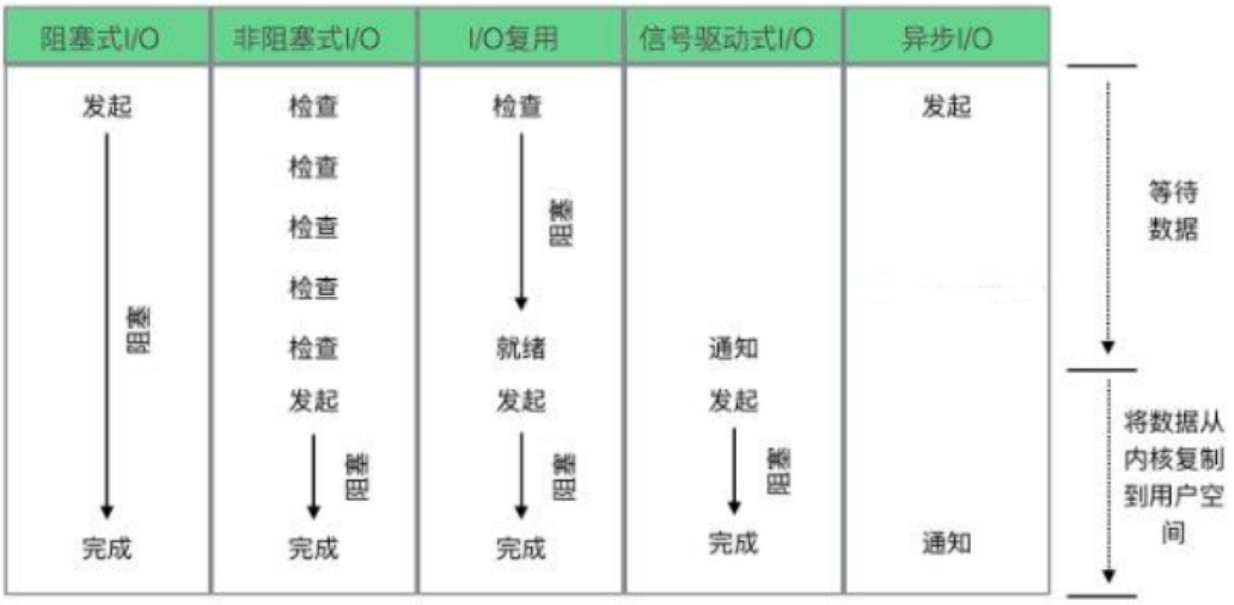
I/O 模型是为了解决内存和外部设备速度差异的问题。我们平时说的阻塞或非阻塞是指应用程序在发起 I/O 操作时，是立即返回还是等待。而同步和异步，是指应用程序在与内核通信时，数据从内核空间到应用空间的拷贝，是由内核主动发起还是由应用程序来触发。

Linux 系统下的 I/O 模型有 5 种：

- 同步阻塞I/O（bloking I/O）
- 同步非阻塞I/O（non-blocking I/O）
- I/O多路复用（multiplexing I/O）
- 信号驱动式I/O（signal-driven I/O）
- 异步I/O（asynchronous I/O）

其中信号驱动式IO在实际中并不常用





Tomcat的 I/O 模型

Tomcat 支持的多种 I/O 模型和应用层协议。Tomcat 支持的 I/O 模型有：

IO模型	描述
BIO (JIoEndpoint)	同步阻塞式IO，即Tomcat使用传统的java.io进行操作。该模式下每个请求都会创建一个线程，对性能开销大，不适合高并发场景。优点是稳定，适合

	连接数目小且固定架构。
NIO（NioEndpoint）	同步非阻塞式IO，jdk1.4 之后实现的新IO。该模式基于多路复用选择器监测连接状态再同步通知线程处理，从而达到非阻塞的目的。比传统BIO能更好的支持并发性能。Tomcat 8.0之后默认采用该模式。NIO方式适用于连接数目多且连接比较短（轻操作） 的架构， 比如聊天服务器， 弹幕系统， 服务器间通讯，编程比较复杂
AIO (Nio2Endpoint)	异步非阻塞式IO，jdk1.7后之支持 。与nio不同在于不需要多路复用选择器，而是请求处理线程执行完成进行回调通知，继续执行后续操作。Tomcat 8之后支持。一般适用于连接数较多且连接时间较长的应用
APR（AprEndpoint）	全称是 Apache Portable Runtime/Apache可移植运行库)，是Apache HTTP服务器的支持库。AprEndpoint 是通过 JNI 调用 APR 本地库而实现非阻塞 I/O 的。使用需要编译安装APR 库

注意： Linux 内核没有很完善地支持异步 I/O 模型，因此 JVM 并没有采用原生的 Linux 异步 I/O，而是在应用层面通过 epoll 模拟了异步 I/O 模型。因此在 Linux 平台上，Java NIO 和 Java NIO.2 底层都是通过 epoll 来实现的，但是 Java NIO 更加简单高效。

Tomcat I/O 模型的选择

I/O 调优实际上是连接器类型的选择，一般情况下默认都是 NIO，在绝大多数情况下都是够用的，除非你的 Web 应用用到了 TLS 加密传输，而且对性能要求极高，这个时候可以考虑 APR，因为 APR 通过 OpenSSL 来处理 TLS 握手和加密 / 解密。OpenSSL 本身用 C 语言实现，它还对 TLS 通信做了优化，所以性能比 Java 要高。如果你的 Tomcat 跑在 Windows 平台上，并且 HTTP 请求的数据量比较大，可以考虑 NIO.2，这是因为 Windows 从操作系统层面实现了真正意义上的异步 I/O，如果传输的数据量比较大，异步 I/O 的效果就能显现出来。

指定IO模型只需修改protocol配置

```
<!-- 修改protocol属性，使用NIO2 -->
<Connector port="8080" protocol="org.apache.coyote.http11.Http11Nio2Protocol"
  connectionTimeout="20000"
  redirectPort="8443" />
```

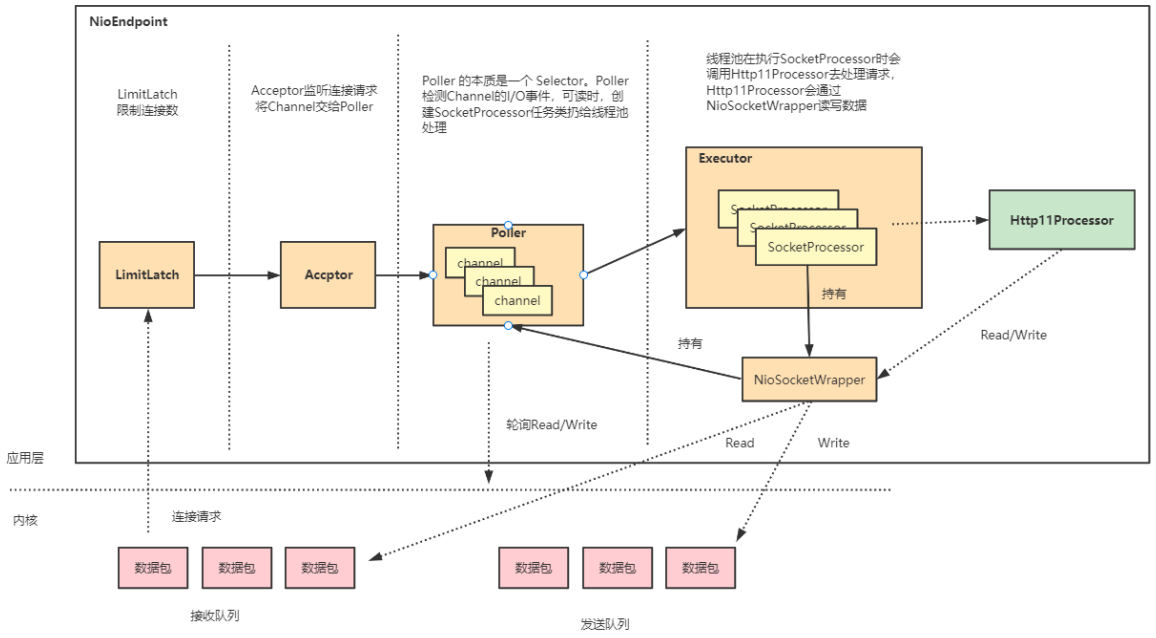
NioEndpoint

思考：Tomcat是如何实现非阻塞I/O的?

在 Tomcat 中，EndPoint 组件的主要工作就是处理 I/O，而 NioEndpoint 利用 Java NIO API 实现了多路复用 I/O 模型。Tomcat的NioEndpoint 是基于主从Reactor多线程模型设计的

Scalable IO in Java.pdf
270.55KB

NioEndpoint的设计思路



- LimitLatch 是连接控制器，它负责控制最大连接数，NIO 模式下默认是10000(tomcat9中8192)，当连接数到达最大时阻塞线程，直到后续组件处理完一个连接后将连接数减 1。注意到达最大连接数后操作系统底层还是会接收客户端连接，但用户层已经不再接收。
- Acceptor 跑在一个单独的线程里，它在一个死循环里调用 accept 方法来接收新连接，一旦有新的连接请求到来，accept 方法返回一个 Channel 对象，接着把 Channel 对象交给 Poller 去处理。

```
#NioEndpoint#initServerSocket

serverSock = ServerSocketChannel.open();
//第2个参数表示操作系统的等待队列长度，默认100
//当应用层面的连接数到达最大值时，操作系统可以继续接收的最大连接数
serverSock.bind(addr, getAcceptCount());
//ServerSocketChannel 被设置成阻塞模式
serverSock.configureBlocking(true);
```

ServerSocketChannel 通过 accept() 接受新的连接，accept() 方法返回获得 SocketChannel 对象，然后将 SocketChannel 对象封装在一个 PollerEvent 对象中，并将

PollerEvent 对象压入 Poller 的 SynchronizedQueue 里，这是个典型的生产者 - 消费者模式，Acceptor 与 Poller 线程之间通过 SynchronizedQueue 通信。

- Poller 的本质是一个 Selector，也跑在单独线程里。Poller 在内部维护一个 Channel 数组，它在一个死循环里不断检测 Channel 的数据就绪状态，一旦有 Channel 可读，就生成一个 SocketProcessor 任务对象扔给 Executor 去处理。

```
public class Poller implements Runnable {

    private Selector selector;
    private final SynchronizedQueue<PollerEvent> events =
            new SynchronizedQueue<>();
```

- Executor 就是线程池，负责运行 SocketProcessor 任务类，SocketProcessor 的 run 方法会调用 Http11Processor 来读取和解析请求数据。Http11Processor 是应用层协议的封装，它会调用容器获得响应，再把响应通过 Channel 写出。

设计精髓：Tomcat线程池扩展

思考：Tomcat是如何扩展java线程池的?

Tomcat线程池默认实现StandardThreadExecutor。Tomcat 线程池和 Java 原生线程池的区别：

- 自定义了拒绝策略，Tomcat 在线程总数达到最大数时，不是立即执行拒绝策略，而是再尝试向任务队列添加任务，添加失败后再执行拒绝策略。
- TaskQueue 重写了 LinkedBlockingQueue 的 offer 方法。只有当前线程数大于核心线程数、小于最大线程数，并且已提交的任务个数大于当前线程数时，也就是说线程不够用了，但是线程数又没达到极限，才会去创建新的线程。目的：在任务队列的长度无限制的情况下，让线程池有机会创建新的线程。

设计精髓：NIO中涉及的对象池技术

Java 对象，特别是一个比较大、比较复杂的 Java 对象，它们的创建、初始化和 GC 都需要耗费 CPU 和内存资源，为了减少这些开销，Tomcat 使用了对象池技术。对象池技术可以减少频繁创建和销毁对象带来的成本，实现对象的缓存和复用，是典型的以空间换时间的设计思路。

思考： PollerEvent为什么采用SynchronizedStack缓存?

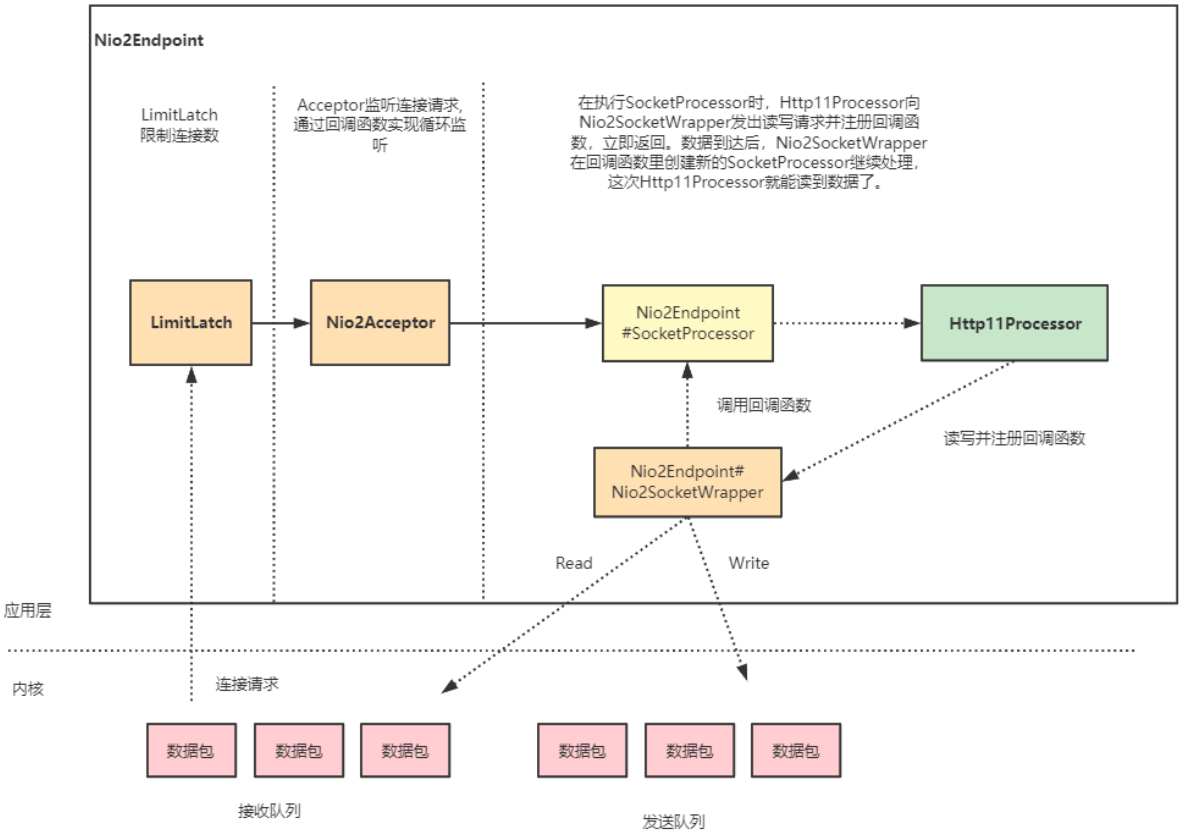
```
1 /**
2  * Cache for poller events
3  */
4 private SynchronizedStack<PollerEvent> eventCache;
```

SynchronizedStack 内部维护了一个对象数组，并且用数组来实现栈的接口：push 和 pop 方法，这两个方法分别用来归还对象和获取对象。SynchronizedStack 用数组而不是链表来维护对象，可以减少结点维护的内存开销，并且它本身只支持扩容不支持缩容，也就是说数组对象在使用过程中不会被重新赋值，也就不会被 GC。这样设计的目的是用最低的内存和 GC 的代价来实现无界容器，同时 Tomcat 的最大同时请求数是有限制的，因此不需要担心对象的数量会无限膨胀。

Nio2Endpoint

NIO 和 NIO.2 最大的区别是，一个是同步一个是异步。异步最大的特点是，应用程序不需要自己去触发数据从内核空间到用户空间的拷贝。

思考：Tomcat如何实现异步I/O的?



Nio2Endpoint 中没有 Poller 组件，也就是没有 Selector。在异步 I/O 模式下，Selector 的工作交给内核来做了。

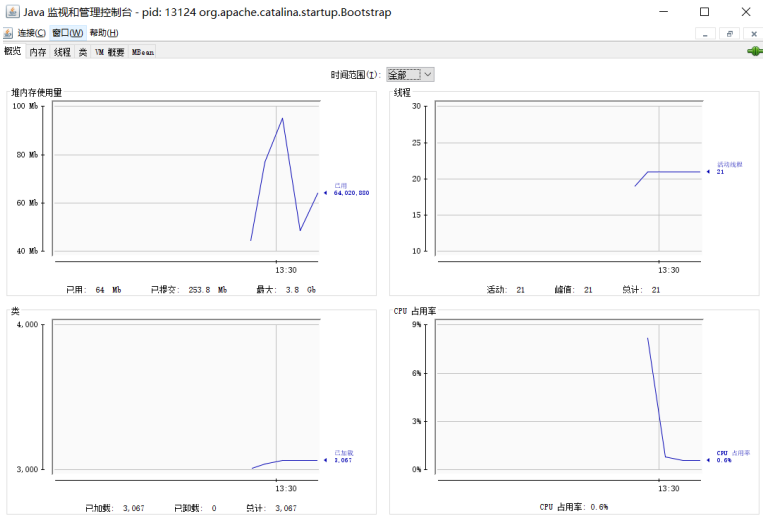
Tomcat调优

Tomcat9参数配置：https://tomcat.apache.org/tomcat-9.0-doc/config/http.html

Tomcat 的关键指标

Tomcat 的关键指标有吞吐量、响应时间、错误数、线程池、CPU 以及 JVM 内存。前三个指标是我们最关心的业务指标，Tomcat 作为服务器，就是要能够又快有好地处理请求，因此吞吐量要大、响应时间要短，并且错误数要少。后面三个指标是跟系统资源有关的，当某个资源出现瓶颈就会影响前面的业务指标，比如线程池中的线程数量不足会影响吞吐量和响应时间；但是线程数太多会耗费大量 CPU，也会影响吞吐量；当内存不足时会触发频繁地 GC，耗费 CPU，最后也会反映到业务指标上来。

通过 JConsole 监控 Tomcat



连接远程tomcat

linux上，在tomcat安装目录的bin下，新建setenv.sh,重启tomcat

```
export JAVA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=9999 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
```

命令行查看 Tomcat 指标

```
#通过 ps 命令找到 Tomcat 进程，拿到进程 ID
ps -ef|grep tomcat
#查看进程状态的大致信息
```

```
cat/proc/<pid>/status
#监控进程的 CPU 和内存资源使用情况
top -p pid
# 查看 Tomcat 的网络连接,比如 Tomcat 在 8080 端口上监听连接请求
netstat -na|grep 8080
```

线程池的并发调优

线程池调优指的是给 Tomcat 的线程池设置合适的参数，使得 Tomcat 能够又快又好地处理请求。

参数	描述
threadPriority	(int) 线程优先级，默认是5 (Thread.NORM_PRIORITY)
daemon	(boolean) 是否deamon线程，默认为true
namePrefix	(String) 线程前缀
maxThreads	(int) 线程池中的最大线程数，默认是200
minSpareThreads	(int) 最小线程数（线程空闲超过一段时间会被回收），默认是25
maxIdleTime	(int) 线程最大的空闲时间，超过这个时间线程就会回收，直到线程数剩下minSpareThreads个，默认值是一分钟
maxQueueSize	(int) 线程池中任务队列的最大长度，默认是Integer.MAX_VALUE
prestartminSpareThreads	(boolean) 是否在线程池启动时就创建minSpareThreads 个线程，默认为false

sever.xml中配置线程池

```
<!--
namePrefix: 线程前缀
maxThreads: 最大线程数，默认设置 200，一般建议在 500 ~ 800，根据硬件设施和业务来判断
minSpareThreads: 核心线程数，默认设置 25
prestartminSpareThreads: 在 Tomcat 初始化的时候就初始化核心线程
maxQueueSize: 最大的等待队列数，超过则拒绝请求 ，默认 Integer.MAX_VALUE
maxIdleTime: 线程空闲时间，超过该时间，线程会被销毁，单位毫秒
className: 线程实现类,默认org.apache.catalina.core.StandardThreadExecutor

 <Executor name="tomcatThreadPool" namePrefix="catalina-exec-Fox"
  prestartminSpareThreads="true"
  maxThreads="500" minSpareThreads="8" maxIdleTime="10000"/>

 <Connector port="8080" protocol="HTTP/1.1" executor="tomcatThreadPool"
  connectionTimeout="20000"
```

```
16   redirectPort="8443" URIEncoding="UTF-8"/>
```

这里面最核心的就是如何确定 maxThreads 的值，如果这个参数设置小了，Tomcat 会发生线程饥饿，并且请求的处理会在队列中排队等待，导致响应时间变长；如果 maxThreads 参数值过大，同样也会有问题，因为服务器的 CPU 的核数有限，线程数太多会导致线程在 CPU 上来回切换，耗费大量的切换开销。

理论上我们可以通过公式 线程数 = CPU 核心数 * （1+平均等待时间/平均工作时间），计算出一个理想值，这个值只具有指导意义，因为它受到各种资源的限制，实际场景中，我们需要在理想值的基础上进行压测，来获得最佳线程数。

SpringBoot应用中调整Tomcat参数

方式1： yml中配置 （属性配置类：ServerProperties）

```yaml
server:
 tomcat:
 threads:
 min-spare: 20
 max: 200
 connection-timeout: 5000ms
```

SpringBoot中的TomcatConnectorCustomizer类可用于对Connector进行定制化修改。

```
@Configuration
public class MyTomcatCustomizer implements
 WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

 @Override
 public void customize(TomcatServletWebServerFactory factory) {
 factory.setPort(8090);
 factory.setProtocol("org.apache.coyote.http11.Http11NioProtocol");
 factory.addConnectorCustomizers(connectorCustomizer());
  }

 @Bean
 public TomcatConnectorCustomizer connectorCustomizer(){
 return new TomcatConnectorCustomizer() {
 @Override
 public void customize(Connector connector) {
 Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolH
andler();
 protocol.setMaxThreads(500);
```

```
19   protocol.setMinSpareThreads(20);
20   protocol.setConnectionTimeout(5000);
21   }
22   };
23   }
24
25 }
```