

主讲老师: Fox

课前须知:

- 课前预习: [BIO、NIO编程与直接内存、零拷贝深入辨析](#)
- 有道云笔记地址: <https://note.youdao.com/s/Yh24cs62>

## 1. Tomcat I/O模型详解

### 1.1 Linux I/O模型详解

#### I/O要解决什么问题

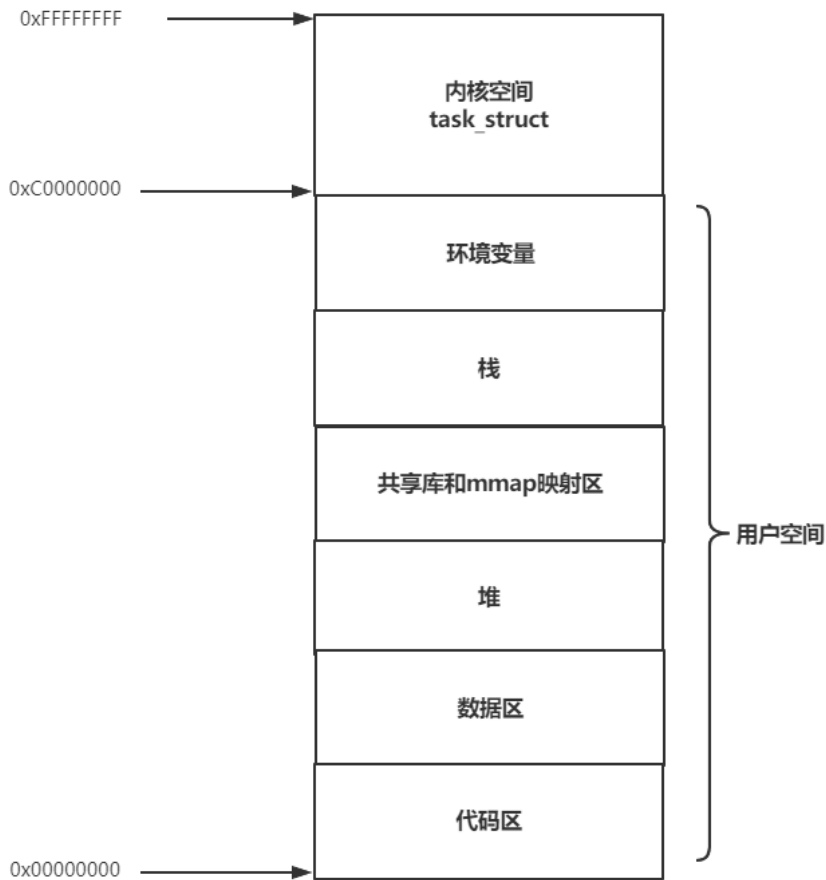
I/O: 在计算机内存与外部设备之间拷贝数据的过程。

程序通过CPU向外部设备发出读指令, 数据从外部设备拷贝至内存需要一段时间, 这段时间CPU就没事做了, 程序就会两种选择:

1. 让出CPU资源, 让其干其他事情
2. 继续让CPU不停地查询数据是否拷贝完成

到底采取何种选择就是I/O模型需要解决的事情了。

以网络数据读取为例来分析, 会涉及两个对象, 一个是调用这个I/O操作的用户线程, 另一个是操作系统内核。一个进程的地址空间分为用户空间和内核空间, 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, 只有内核可以直接访问各种硬件资源, 比如磁盘和网卡。



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：

- **数据准备阶段：** 用户线程等待内核将数据从网卡拷贝到内核空间。
- **数据拷贝阶段：** 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）。

不同的I/O模型对于这2个步骤有着不同的实现步骤。

## Linux的I/O模型分类

Linux 系统下的 I/O 模型有 5 种：

- 同步阻塞I/O (blocking 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

- 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。
- 同步或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。

请结合课上BIO,NIO,AIO的代码来理解

## 1.2 Tomcat支持的 I/O 模型

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， <b>适合连接数目小且固定架构</b> 。Tomcat8.5.x开始移除BIO。
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。 <b>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</b>
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。 <b>AprEndpoint 是通过 JNI 调用 APR 本地库而实现非阻塞 I/O 的</b> 。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 NIO2 底层都是通过 epoll 来实现的，但是 Java NIO 更加简单高效。

### Tomcat I/O 模型如何选型

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 NIO2，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。如果你的 Tomcat 跑在 Linux 平台上，建议使用 NIO。因为在 Linux 平台上，Java NIO 和 Java NIO2 底层都是通过 epoll 来实现的，但是 Java NIO 更加简单高效。


指定IO模型只需修改protocol配置

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

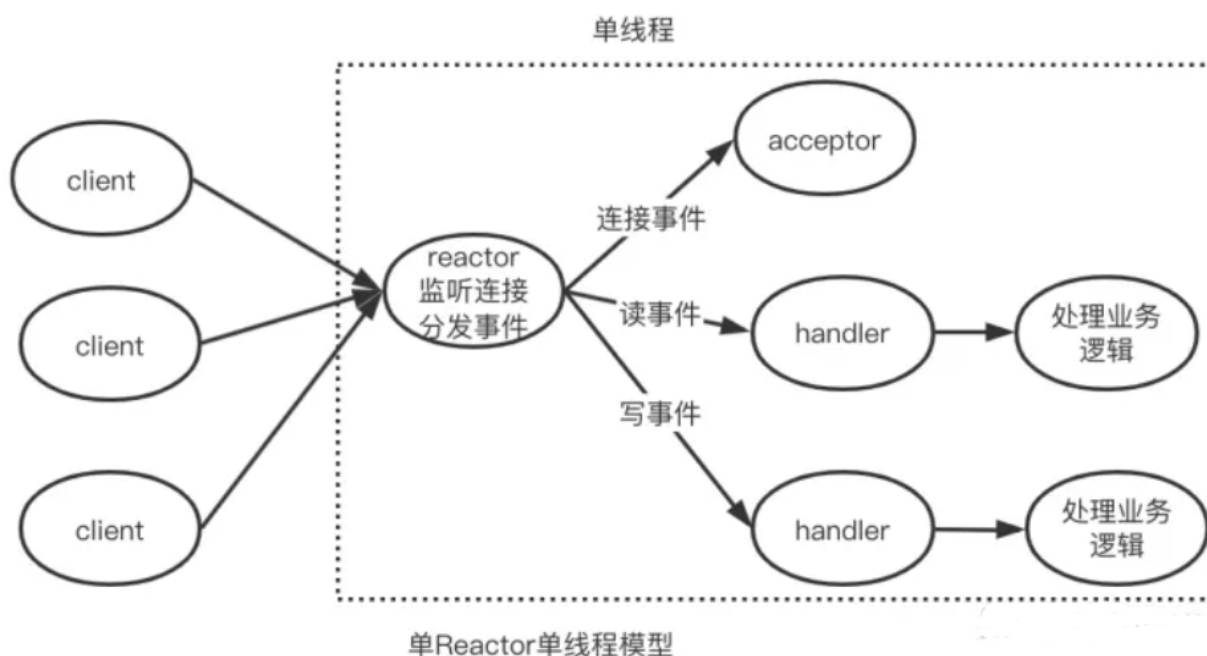
### 1.3 网络编程模型Reactor线程模型

Reactor 模型是网络服务器端用来处理高并发网络 IO 请求的一种编程模型。

该模型主要有三类处理事件：即连接事件、写事件、读事件；三个关键角色：即 reactor、acceptor、handler。acceptor负责连接事件，handler负责读写事件，reactor负责事件监听和事件分发。

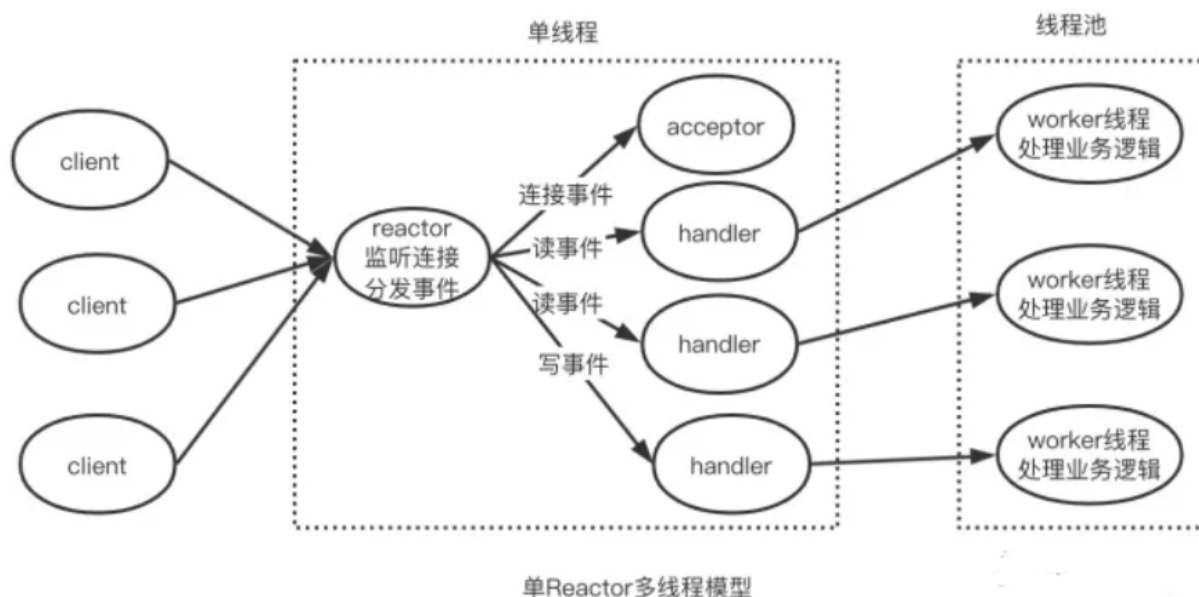
 Scalable IO in Ja....pdf  
270.55KB

#### 单 Reactor 单线程



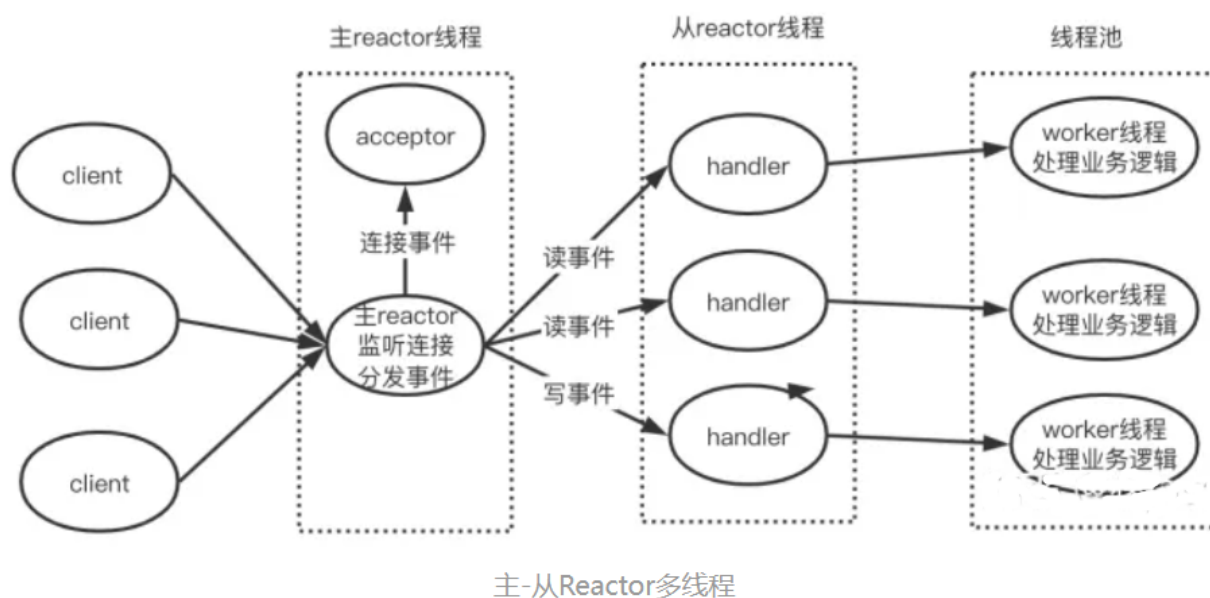
由上图可以看出，单Reactor单线程模型中的 reactor、acceptor 和 handler以及后续业务处理逻辑的功能都是由一个线程来执行的。reactor 负责监听客户端事件和事件分发，一旦有连接事件发生，它会分发给 acceptor，由 acceptor 负责建立连接，然后创建一个 handler。如果是读写事件，reactor 将事件分发给 handler 进行处理。handler 负责读取客户端请求，进行业务处理，并最终给客户端返回结果。

#### 单 Reactor 多线程



该模型中，reactor、acceptor 和 handler 的功能由一个线程来执行，与此同时，会有一个线程池，由若干 worker 线程组成。在监听客户端事件、连接事件处理方面，这个类型和单 reactor 单线程是相同的，但是不同之处在于，在单 reactor 多线程类型中，handler 只负责读取请求和写回结果，而具体的业务处理由 worker 线程来完成。

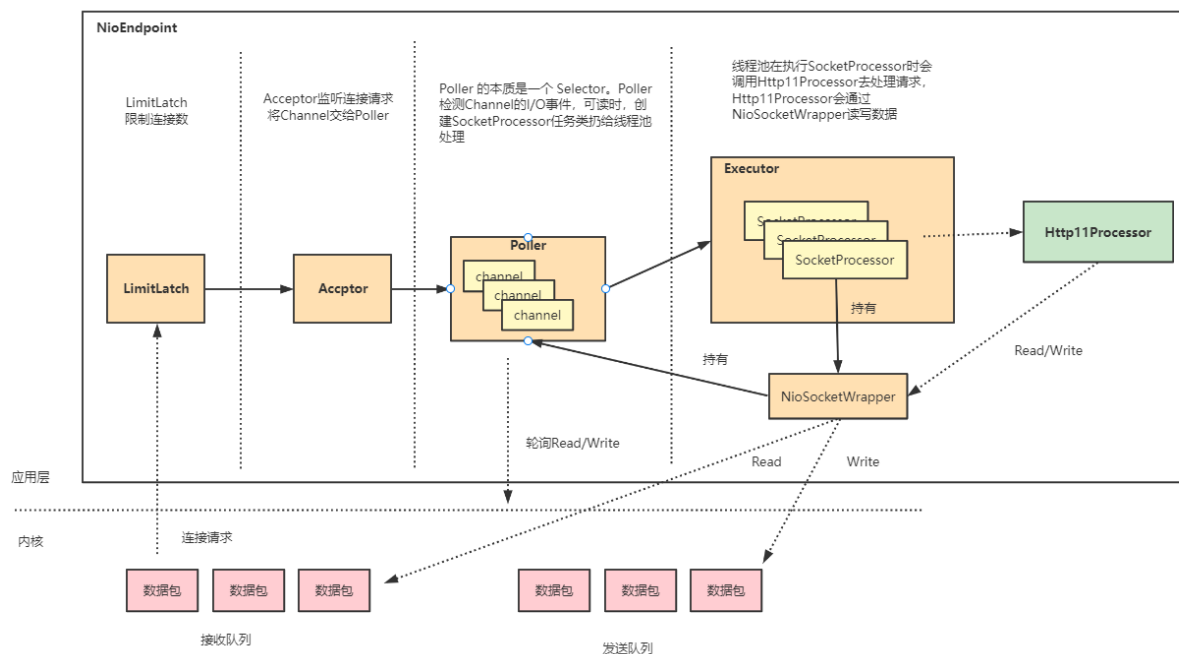
## 主从 Reactor 多线程



在这个类型中，会有一个主 reactor 线程、多个子 reactor 线程和多个 worker 线程组成的一个线程池。其中，主 reactor 负责监听客户端事件，并在同一个线程中让 acceptor 处理连接事件。一旦连接建立后，主 reactor 会把连接分发给子 reactor 线程，由子 reactor 负责这个连接上的后续事件处理。那么，子 reactor 会监听客户端连接上的后续事件，有读写事件发生时，它会让在同一个线程中的 handler 读取请求和返回结果，而和单 reactor 多线程类似，具体业务处理，它还是会让线程池中的 worker 线程处理。

## 1.4 Tomcat NIO实现

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**



- 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。

```
1 #NioEndpoint#initServerSocket
2
3 serverSock = ServerSocketChannel.open();
4 //第2个参数表示操作系统的等待队列长度，默认100
5 //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
6 serverSock.bind(addr, getAcceptCount());
7 //ServerSocketChannel 被设置成阻塞模式
8 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。

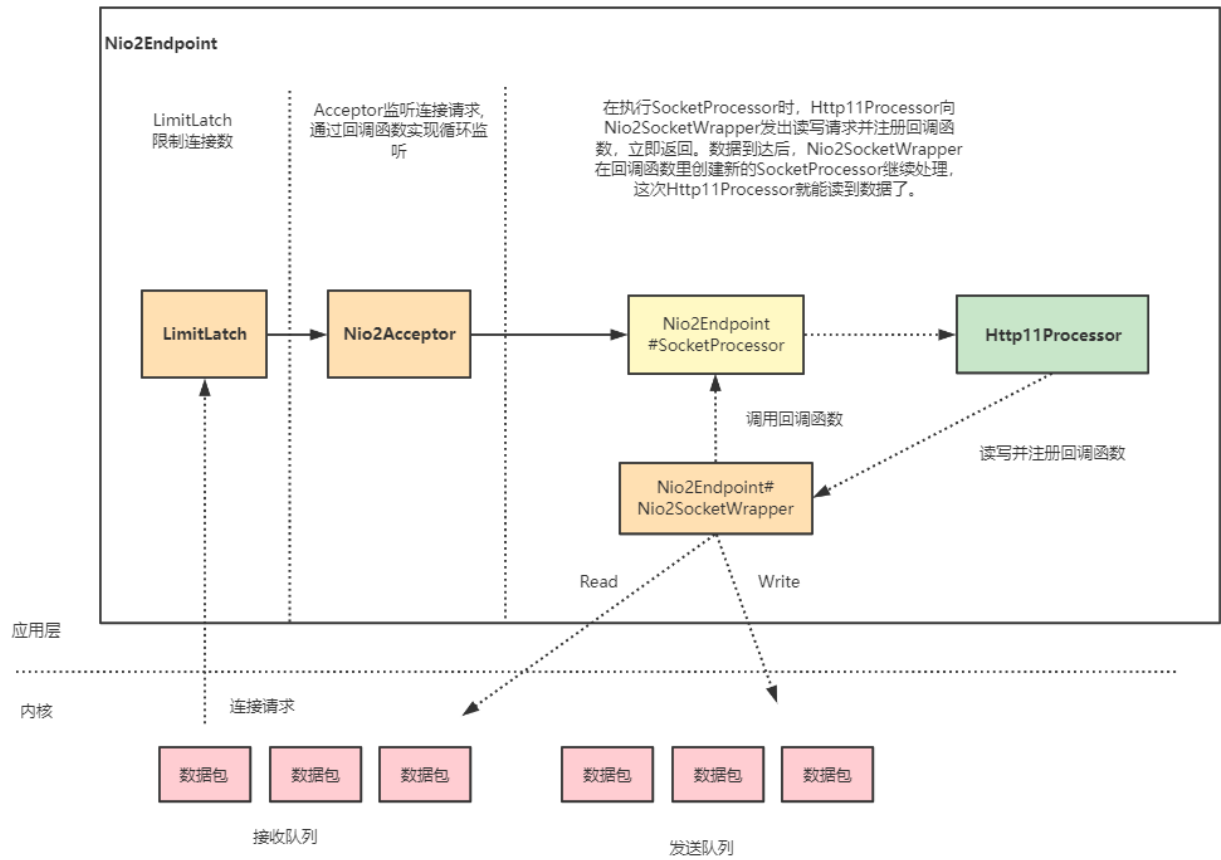
```
public class Poller implements Runnable {

    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。

## 1.5 Tomcat 异步IO实现

NIO 和 NIO2 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。



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。

## 2. Tomcat性能调优

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



## 2.1 如何监控Tomcat的性能

### 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，就是要能够又快又好地处理请求，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

JConsole是一款基于JMX的可视化监控和管理工具

#### 1) 开启 JMX 的远程监听端口

我们可以在 Tomcat 的 bin 目录下新建一个名为setenv.sh的文件（或者setenv.bat，根据你的操作系统类型），然后输入下面的内容：

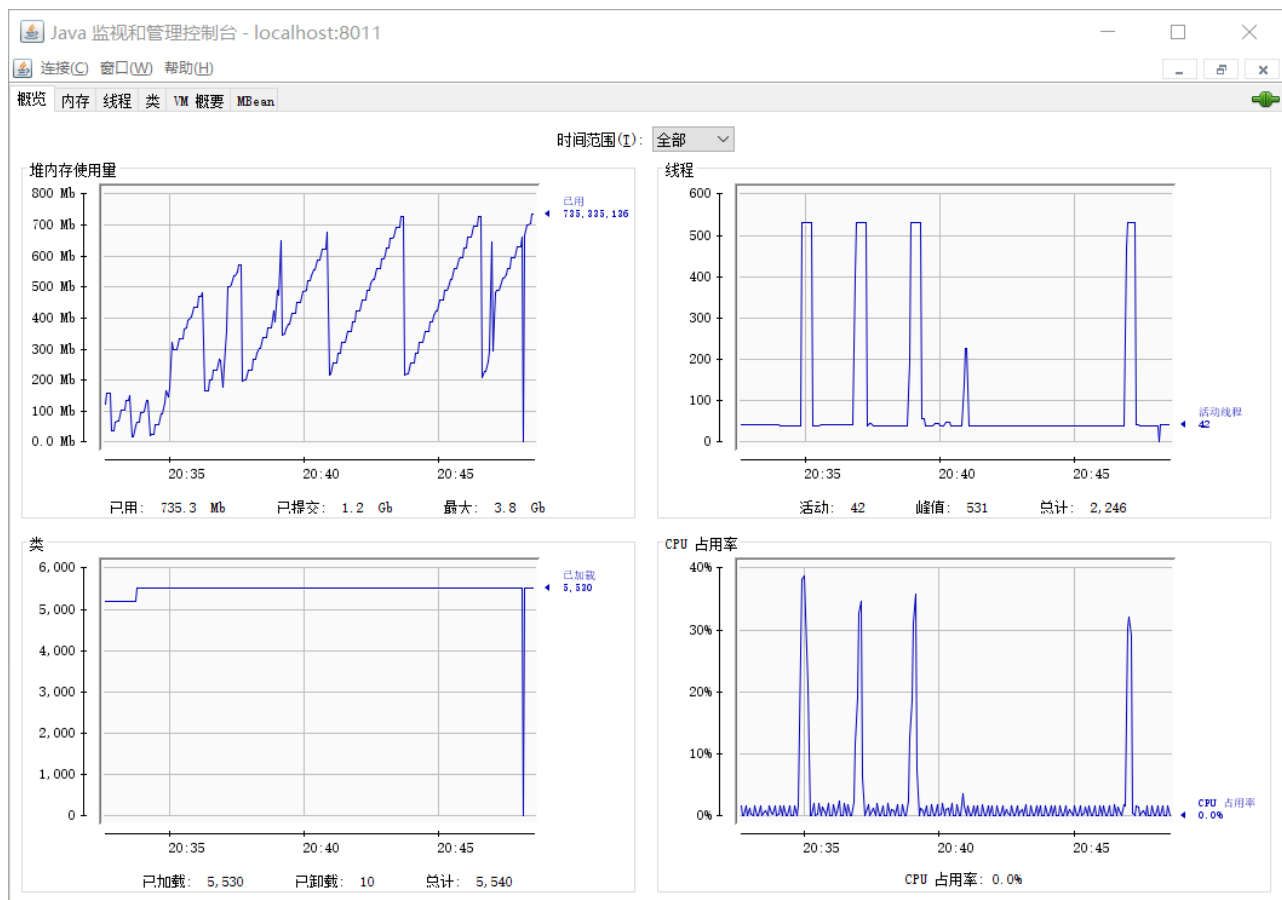
```
1 export JAVA_OPTS="${JAVA_OPTS} -Dcom.sun.management.jmxremote"
2 export JAVA_OPTS="${JAVA_OPTS} -Dcom.sun.management.jmxremote.port=8011"
3 export JAVA_OPTS="${JAVA_OPTS} -Djava.rmi.server.hostname=x.x.x.x"
4 export JAVA_OPTS="${JAVA_OPTS} -Dcom.sun.management.jmxremote.ssl=false"
5 export JAVA_OPTS="${JAVA_OPTS} -Dcom.sun.management.jmxremote.authenticate=false"
```

2)重启 Tomcat，这样 JMX 的监听端口 8011 就开启了，接下来通过 JConsole 来连接这个端口。

```
1 jconsole x.x.x.x:8011
```

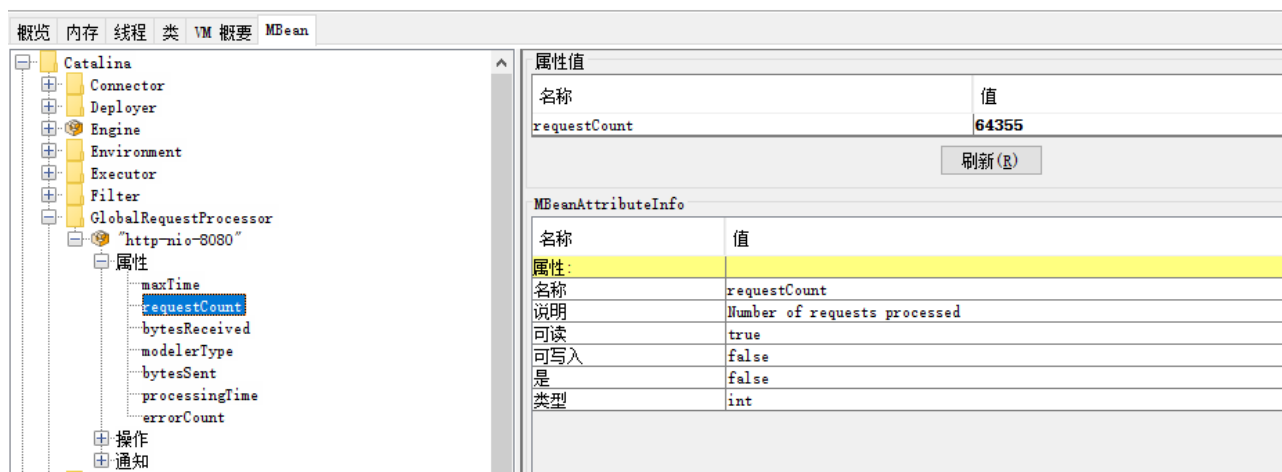
我们可以看到 JConsole 的主界面：





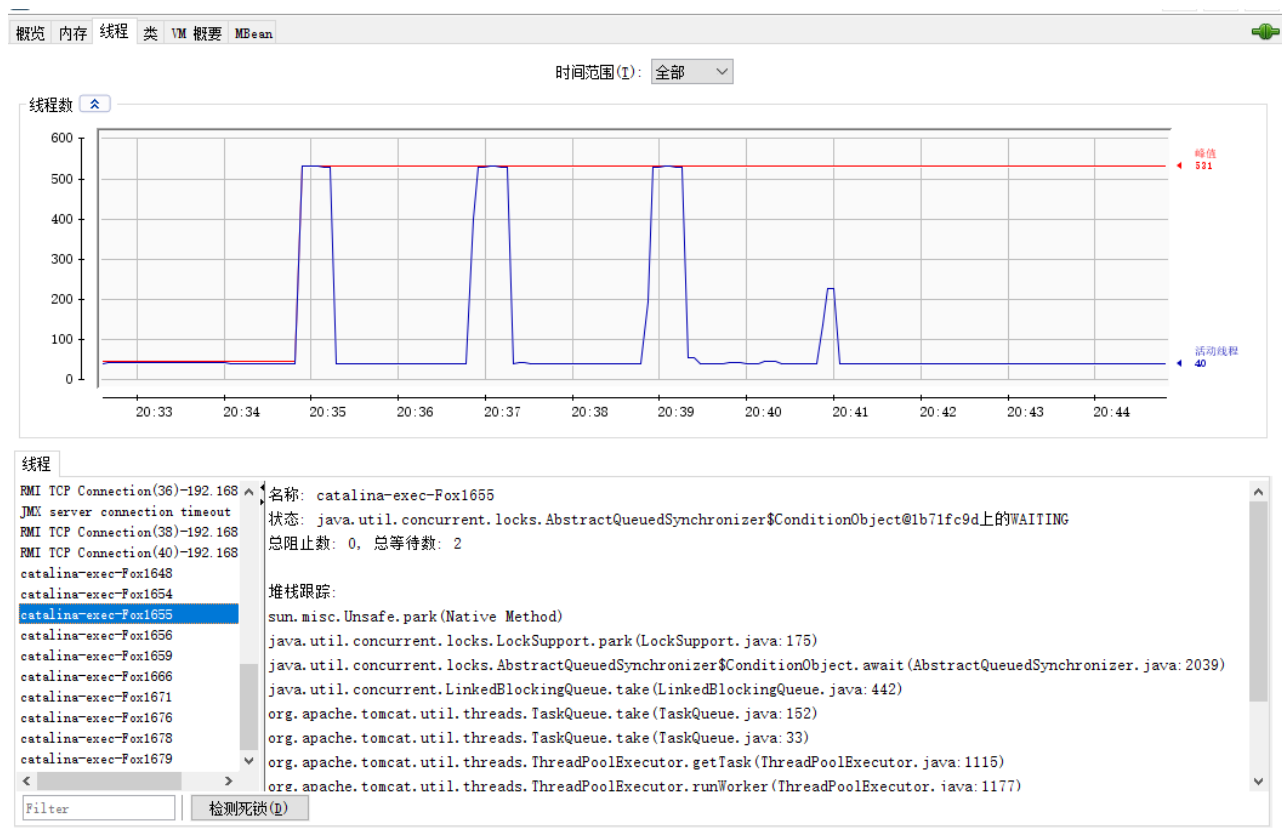
## 吞吐量、响应时间、错误数

在 MBeans 标签页下选择 GlobalRequestProcessor，这里有 Tomcat 请求处理的统计信息。你会看到 Tomcat 中的各种连接器，展开“http-nio-8080”，你会看到这个连接器上的统计信息，其中 maxTime 表示最长的响应时间，processingTime 表示平均响应时间，requestCount 表示吞吐量，errorCount 就是错误数。



## 线程池

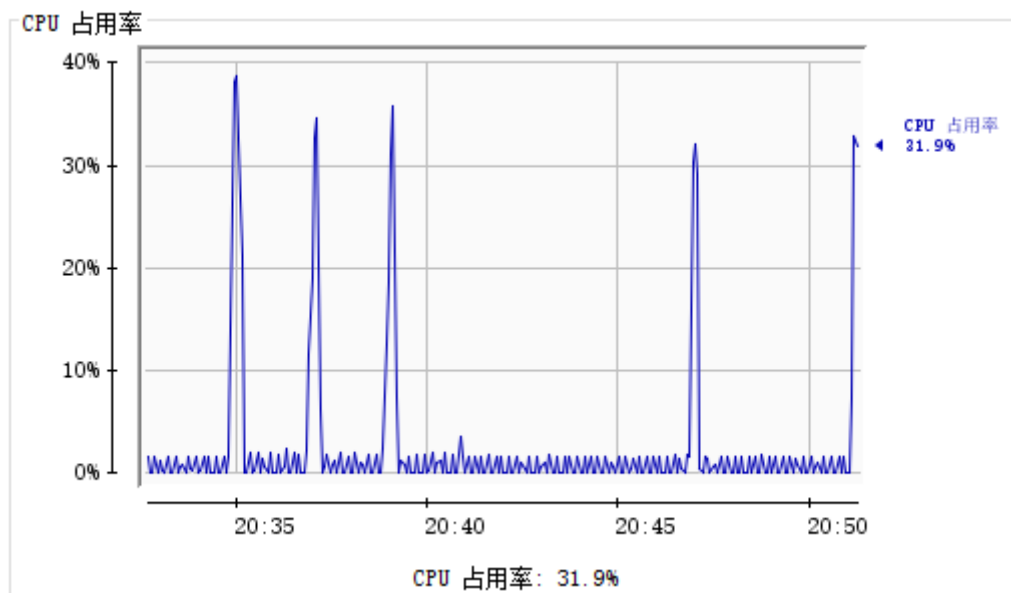
选择“线程”标签页，可以看到当前 Tomcat 进程中有多少线程，如下图所示：



图的左下方是线程列表，右边是线程的运行栈，这些都是非常有用的信息。如果大量线程阻塞，通过观察线程栈，能看到线程阻塞在哪个函数，有可能是 I/O 等待，或者是死锁。

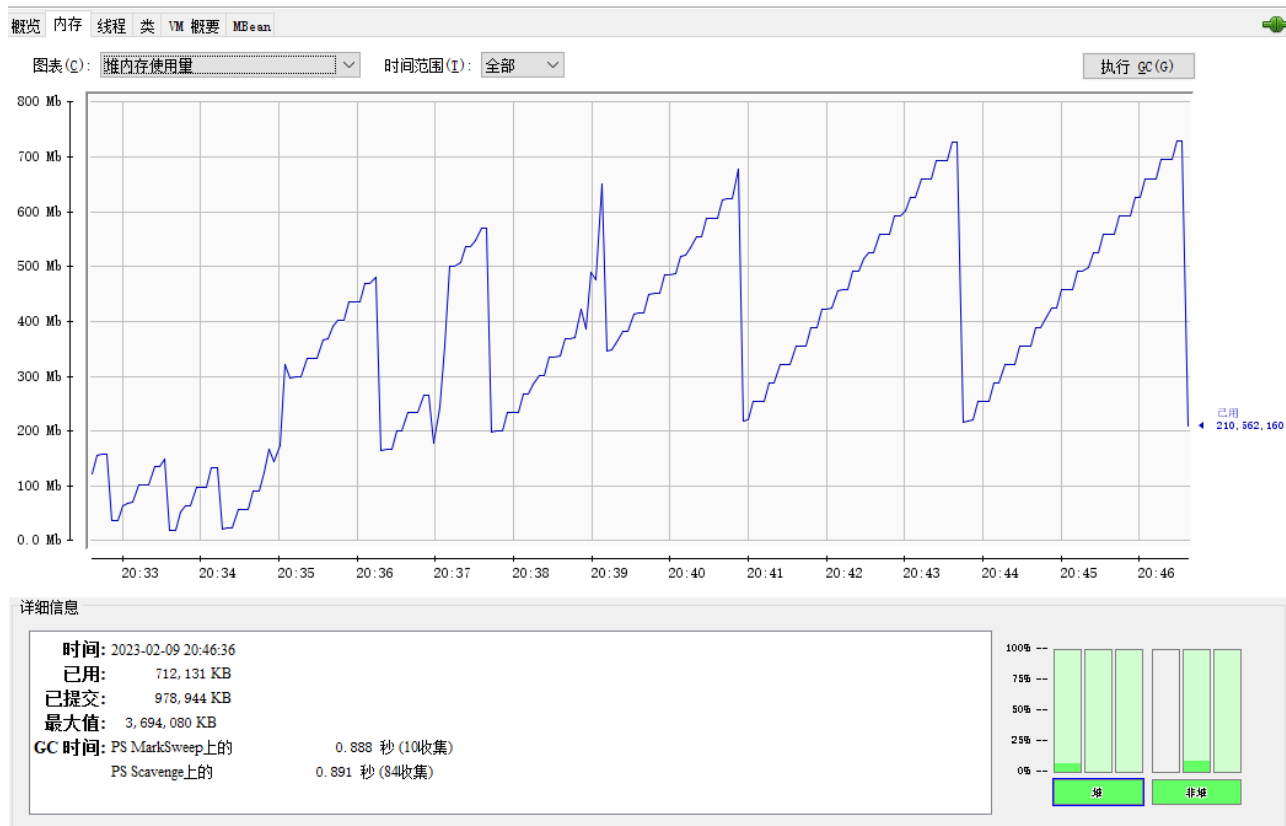
## CPU

在主界面可以找到 CPU 使用率指标，请注意这里的 CPU 使用率指的是 Tomcat 进程占用的 CPU，不是主机总的 CPU 使用率。



## JVM 内存

选择“内存”标签页，你能看到 Tomcat 进程的 JVM 内存使用情况。



## 命令行查看 Tomcat 指标

极端情况下如果 Web 应用占用过多 CPU 或者内存，又或者程序中发生了死锁，导致 Web 应用对外没有响应，监控系统上看不到数据，这个时候需要我们登陆到目标机器，通过命令行来查看各种指标。

1. 首先我们通过 ps 命令找到 Tomcat 进程，拿到进程 ID。

```
1 ps -ef|grep tomcat
```

2. 接着查看进程状态的大致信息，通过cat /proc/<pid>/status命令：

3. 监控进程的 CPU 和内存资源使用情况：

4. 查看 Tomcat 的网络连接，比如 Tomcat 在 8080 端口上监听连接请求，通过下面的命令查看连接列表：

你还可以分别统计处在“已连接”状态和“TIME\_WAIT”状态的连接数：

5. 通过 ifstat 来查看网络流量，大致可以看出 Tomcat 当前的请求数和负载状况。

## 2.2 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。

## server.xml中配置线程池

```
1  <!--
2  namePrefix: 线程前缀
3  maxThreads: 最大线程数，默认设置 200，一般建议在 500 ~ 1000，根据硬件设施和业务来判断
4  minSpareThreads: 核心线程数，默认设置 25
5  prestartminSpareThreads: 在 Tomcat 初始化的时候就初始化核心线程
6  maxQueueSize: 最大的等待队列数，超过则拒绝请求，默认 Integer.MAX_VALUE
7  maxIdleTime: 线程空闲时间，超过该时间，线程会被销毁，单位毫秒
8  className: 线程实现类，默认org.apache.catalina.core.StandardThreadExecutor
9  -->
10 <Executor name="tomcatThreadPool" namePrefix="catalina-exec-Fox"
11         prestartminSpareThreads="true"
12         maxThreads="500" minSpareThreads="10" maxIdleTime="10000"/>
13
14 <Connector port="8080" protocol="HTTP/1.1" executor="tomcatThreadPool"
15         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  $\text{线程数} = \text{CPU 核心数} * (1 + \text{平均等待时间} / \text{平均工作时间})$ ，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中调整Tomcat参数

方式1: yml中配置（属性配置类：ServerProperties）

```
1  server:
2    tomcat:
3      threads:
4        min-spare: 20
```

```
5     max: 500
6     connection-timeout: 5000ms
```

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

```
1  @Configuration
2  public class MyTomcatCustomizer implements
3      WebServerFactoryCustomizer<TomcatServletWebServerFactory> {
4
5      @Override
6      public void customize(TomcatServletWebServerFactory factory) {
7          factory.setPort(8090);
8          factory.setProtocol("org.apache.coyote.http11.Http11NioProtocol");
9          factory.addConnectorCustomizers(connectorCustomizer());
10     }
11
12     @Bean
13     public TomcatConnectorCustomizer connectorCustomizer(){
14         return new TomcatConnectorCustomizer() {
15             @Override
16             public void customize(Connector connector) {
17                 Http11NioProtocol protocol = (Http11NioProtocol)
connector.getProtocolHandler();
18                 protocol.setMaxThreads(500);
19                 protocol.setMinSpareThreads(20);
20                 protocol.setConnectionTimeout(5000);
21             }
22         };
23     }
24
25 }
```