# Netty篇

## netty简述

* Netty 是一个基于 JAVA **NIO** 类库的**异步通信框架**，用于创建**异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性的**网络**客户端**和**服务器端**
* netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read和write事件

### Reactor中的组件

* Reactor:Reactor是IO事件的派发者。
* Acceptor:Acceptor接受client连接，建立对应client的Handler，并向Reactor注册此Handler。
* Handler:和一个client通讯的实体，按这样的过程实现业务的处理。一般在基本的Handler基础上还会有更进一步的层次划分， 用来抽象诸如decode，process和encoder这些过程。比如对Web Server而言，decode通常是HTTP请求的解析， process的过程会进一步涉及到Listener和Servlet的调用。业务逻辑的处理在Reactor模式里被分散的IO事件所打破， 所以Handler需要有适当的机制在所需的信息还不全（读到一半）的时候保存上下文，并在下一次IO事件到来的时候（另一半可读了）能继续中断的处理。为了简化设计，Handler通常被设计成状态机，按GoF的state pattern来实现。
* Reactor模型和AWT事件模型很像，就是将消息放到了一个队列中，通过异步线程池对其进行消费
* **特点**
  + **异步、非阻塞、基于事件驱动**的NIO框架
  + 支持**多种传输层通信协议**，包括TCP、UDP等
  + 开发异步HTTP服务端和客户端应用程序
  + 提供对**多种应用层协议**的支持，包括TCP私有协议、HTTP协议、WebSocket协议、文件传输等
  + 默认提供多种编解码能力，包括**Java序列化、Google的ProtoBuf、二进制编解码、Jboss marshalling、文本字符串、base64、简单XML**等，这些编解码框架可以被用户直接使用
  + 提供形式**多样的编解码基础类库**，可以非常方便的实现私有协议栈编解码框架的二次定制和开发
  + 经典的**ChannelFuture-listener**机制，所有的异步IO操作都可以设置listener进行监听和获取操作结果
  + 基于**ChannelPipeline-ChannelHandler**的责任链模式，可以方便的**自定义业务拦截器**用于**业务逻辑定制**
  + 安全性：支持SSL、HTTPS
  + 可靠性：流量整形、读写超时控制机制、缓冲区最大容量限制、资源的优雅释放等
  + **简洁的API和启动辅助类，简化开发难度，减少代码量**
* 为什么**不是传统的IO**
  + **线程模型存在致命缺陷**：一连接一线程的模型导致服务端无法承受大量客户端的并发连接
  + **性能差**：频繁的线程上下文切换导致 CPU 利用效率不高
  + **可靠性差**：由于所有的 IO 操作都是同步的，所以业务线程只要进行 IO 操作，也会存在被同步阻塞的风险，这会导致系统的可靠性差，依赖外部组件的处理能力和网络的情况
* **什么是上下文切换**
* 即使是单核CPU也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。
* CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。
* 这就像我们同时读两本书，当我们在读一本英文的技术书籍时，发现某个单词不认识， 于是便打开中英文词典，但是在放下英文书籍之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读 书效率的，同样上下文切换也会影响多线程的执行速度。
* 使用**NIO**，同步阻塞 IO 的**三个缺陷**都将**迎刃而解**
  + **NIO 采用 Reactor 模式，一个 Reactor 线程聚合一个多路复用器 Selector，它可以同时注册、监听和轮询成百上千个 Channel，一个 IO 线程可以同时并发处理N个客户端连接，线程模型优化为1：N（N < 进程可用的最大句柄数）或者 M : N (M通常为 CPU 核数 + 1， N < 进程可用的最大句柄数)**
  + 由于 IO 线程总数有限，不会存在**频繁的 IO 线程之间上下文切换和竞争，CPU 利用率高**
  + 所有的 IO 操作都是异步的，即使业务线程直接进行 IO 操作，也不会被同步阻塞，系统不再依赖外部的网络环境和外部应用程序的处理性能
  + 切换到 NIO 编程之后可以为系统带来巨大的可靠性、性能提升，所以，目前采用 NIO 进行通信已经逐渐成为主流
* 为什么**不直接使用JDK NIO**类库
  + NIO的**类库和API繁杂**，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等
  + 需要具备其它的额外技能做铺垫，例如**熟悉Java多线程**编程，因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序
  + **可靠性能力补齐，工作量和难度都非常大**。例如客户端面临**断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流**的处理等等，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大
  + JDK NIO的BUG，例如臭名昭著的**epoll bug**，它会导致**Selector空轮询**，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该bug发生概率降低了一些而已，它并没有被根本解决
  + Netty解决策略:
    - 1) 根据该BUG的特征，首先侦测该BUG是否发生；
    - 2) 将问题Selector上注册的Channel转移到新建的Selector上；
    - 3) 老的问题Selector关闭，使用新建的Selector替换。
  + 一个高性能、高可靠性的 NIO 服务端**开发和维护成本**都是非常**高**的，开发者需要具有丰富的 NIO 编程经验和网络维护经验，很多时候甚至需要通过抓包来定位问题
  + 开发出一套 NIO 程序需要 1 个月，但是它的稳定很可能需要 1 年甚至更长的时间
  + Netty 的**健壮性、功能、性能、可定制性和可扩展性**在同类框架中都是**首屈一指**

## 2. netty原理

* **Netty逻辑架构**
  + **第一层**
    - **Reactor 通信调度层**，它由一系列辅助类组成，包括 Reactor 线程NioEventLoop 以及其父类、NioSocketChannel/NioServerSocketChannel 以及其父类、ByteBuffer 以及由其衍生出来的各种 Buffer、Unsafe 以及其衍生出的各种内部子类等
  + **第二层**
    - **职责链 ChannelPipeLine**，它负责调度事件在职责链中的传播，支持动态的编排职责链，职责链可以选择性的拦截自己关心的事件，对于其它IO操作和事件忽略，Handler同时支持inbound和outbound事件
  + **第三层**
    - **业务逻辑编排层**，业务逻辑编排层通常有两类：一类是纯粹的业务逻辑编排，还有一类是应用层协议插件，用于协议相关的编解码和链路管理，例如 CMPP 协议插件
* **灵拷贝**
  + “零拷贝”是指计算机操作的过程中，**CPU不需要为数据在内存之间的拷贝消耗资源**。而它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User Space）而**直接在内核空间（Kernel Space）中传输到网络的方式**
* **Netty的“零拷贝”主要体现在三个方面**
  + Netty的**接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝**。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝
  + Netty提供了**组合Buffer对象**，可以聚合多个ByteBuffer对象，用户可以**像操作一个Buffer那样方便的对组合Buffer进行操作**，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer
  + Netty的文件传输采用了**transferTo方法**，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题
* **Reactor线程模型**
  + **Reactor单线程模型**
    - 一个线程中，Acceptor进行请求派发，处理连接请求，验证， 通过Dispatch将对应的ByteBuffer派发到指定的Handler上进行消息解码，进行业务逻辑Handler工作
    - 小容量应用场景，可以使用单线程模型，对于高负载、大并发的应用却不合适
      * 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送
      * 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈
      * 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障
  + **Reactor多线程模型**
    - 一个专门NIO Acceptor线程**监听服务端，接受客户端TCP连接请求**
    - 由一个NIO线程池（ 可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程）进行**IO操作，读写，编码，消息发送接受**
    - 1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题
    - 绝大多数场景下，Reactor多线程模型都可以满足性能需求， 在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题
      * 例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能
  + **主从Reactor多线程模型**
    - 服务端用于**接收客户端连接**的不再是个1个单独的NIO线程，而是**一个独立的NIO线程池**
    - **Acceptor**接收到**客户端TCP连接请求**处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到IO线程池（sub reactor线程池）的某个IO线程上，由它负责**SocketChannel的读写和编解码工作**
    - **Acceptor线程池**仅仅只用于**客户端的登陆、握手和安全认证**，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作
    - **子线程池**进行**消息接受，发送，编码等业务处理**
  + Netty的线程模型可以通过创建不同的**EventLoopGroup**实例并通过**适当的参数配置**，就可以支持上述**三种Reactor线程模型**
  + Netty 对Reactor线程模型的支持提供了灵活的定制能力，可以满足不同业务场景的性能诉求
* **buffer**
  + **取代JDK NIO的java.nio.ByteBuffer**，相比ByteBuffer
    - 可以根据需要自定义buffer type
    - 内置混合的buffer type，实现zero-copy
    - 提供类似StringBuffer的动态dynamic buffer
    - 不需要调用flip方法
    - 更快的性能

## 3. netty应用场景

* **弹性伸缩的分布式服务架构**
  + 阿里巴巴**Dubbo**内部私有通信协议-dubbo协议默认使用Netty作为高性能异步通信框架，为分布式服务节点之间提供**高性能的NIO客户端和服务端通信**
  + 大众点评服务框架Pigeon和消息中间件Swallow
* **大数据领域**
  + **Hadoop的子系统Apache Avro**
    - 丰富的数据结构
    - 压缩、高效、二进制的序列化框
    - 远程服务调用（RPC）
    - 多语言、灵活的集成能力
* **游戏行业**
  + **高并发**：由于采用异步非阻塞模式，一个Netty游戏服务端可以同时处理成千上万的游戏玩家登陆和在线
  + **高性能**：Netty的性能在各个NIO框架中最高，它的单节点吞吐量非常大，适合海量玩家同时在线游戏
  + **安全性**：支持HTTPS、SSL等，可以在传输层进行安全控制
  + **定制性**：可以方便的实现业务逻辑的定制、游戏编解码的定制，可以方便的与第三方进行集成，例如amf3等

参考: [Netty(RPC高性能之道)](http://blog.csdn.net/zhiguozhu/article/details/50517551)

# IO篇

IO的方式通常分为几种，**同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO。**

## 一、BIO

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

## 二、NIO

    NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题： 在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

    NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。  也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

   BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。

NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

      在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。

　　HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的.当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

## 三、AIO

     与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。  即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。  在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

* AsynchronousSocketChannel
* AsynchronousServerSocketChannel
* AsynchronousFileChannel
* AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

## 四、总结

**BIO是一个连接一个线程。**

**NIO是一个请求一个线程。**

**AIO是一个有效请求一个线程。**

先来个例子理解一下概念，以银行取款为例：

* 同步 ： 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；
* 异步 ： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）；
* 阻塞 ： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；
* 非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

Java对BIO、NIO、AIO的支持：

* Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
* Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
* Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO、NIO、AIO适用场景分析:

* BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
* 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，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
* AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

另外，I/O属于底层操作，需要操作系统支持，并发也需要操作系统的支持，所以性能方面不同操作系统差异会比较明显。

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。

    在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

 一般来说I/O模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞IO

**同步阻塞IO**：在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

**同步非阻塞IO**:在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

**异步阻塞IO**：此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！

**异步非阻塞IO**:在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

# RPC篇

## Thrift(facebook)

静态数据结构,修改数据结构需重新编译IDL文件. 开发简单. 性能高于GRPC

Thrift序列协议

0.9.3使用BUG:传递Map时如果value为空在service.write报错

## GRPC(google)

支持HTTP2

主要面向移动端

Protobuf序列协议

本身非分布式,需自己实现

15年开源 不够成熟

## DUBBO(Ali)

支持多种协议:http/hessian2/thrift等

## AVRO(hadoop)

主要针对大数据

轻量级框架

动态模式,不生成代码,避免侵入性,利于扩展

不支持协议扩展

# Zookeeper篇

Zookeeper 分布式服务框架是 Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。本文将从使用者角度详细介绍 Zookeeper 的安装和配置文件中各个配置项的意义，以及分析 Zookeeper 的典型的应用场景（配置文件的管理、集群管理、同步锁、Leader 选举、队列管理等），用 Java 实现它们并给出示例代码。

# 消息篇

## RocketMQ

稳定(支付宝集群50台,日处理百亿消息)

定时消费:

回溯消费: 重新将某一个时刻之前的消息重新消费一遍

消息过滤:broker端过滤,节省大量网络传输是否有消息重发造成的重复消费

消息堆积: 就是当缓存消息的内存满了之后的解决方案，一种是丢弃策略，这种不会影响吞吐量，还有一种就是将消息持久化到磁盘，这种会影响吞吐量，在评估影响程度上，RocketMQ的成绩稍微好一点

客户端不在线：RocketMQ可以在客户端上线后继续将未消费的消息推送到客户端

持久化

事务

顺序：我们的消息总线中的消息应该都是无状态的，所以对消息的处理顺序没有严格的要求，如果有特殊要求的话可以在业务层进行控制，activeMQ无法保证严格的顺序，RocketMQ可以保证严格的消费顺序

## ActiveMQ

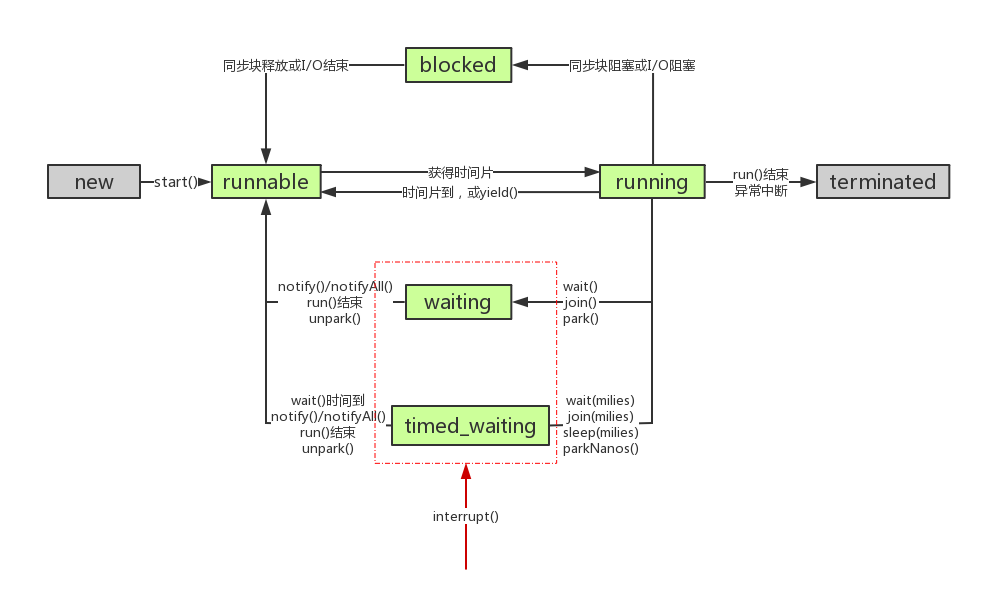
RocketMQ需要新建一个特殊队列来接收优先级高的队列，无法实现从0-65535这种细粒度的控制，ActiveMQ可以精细控制

## Kafka

# 线程篇

## 基础篇

### 线程状态



public enum State {  
 */\*\*  
 \* 新建  
 \*/  
 NEW*,  
  
 */\*\*  
 \* 就绪  
 \*/  
 RUNNABLE*,  
  
 */\*\*  
 \* 阻塞  
 \* {****@link*** *Object#wait() Object.wait}.  
 \*/  
 BLOCKED*,  
  
 */\*\*  
 \* 等待  
 \** ***@important*** *调用wait()方法引起"当前线程等待",直到另外一个线程调用notify()或notifyAll()唤醒该线程。  
 \* 换句话说，这个方法和wait(0)的效果一样！。  
 \* “当前线程”在调用wait()时，必须拥有该对象的同步锁。该线程调用wait()之后，会释放该锁；  
 \* 然后一直等待直到“其它线程”调用对象的同步锁的notify()或notifyAll()方法。  
 \* 然后，该线程继续等待直到它重新获取“该对象的同步锁”，然后就可以接着运行。  
 \* following methods:  
 \* {****@link*** *Object#wait() Object.wait} with no timeout  
 \* {****@link*** *#join(long) Thread.join} with no timeout  
 \* {****@link*** *LockSupport#park() LockSupport.park}  
 \*/  
 WAITING*,  
  
 */\*\*  
 \* 带等待时间的WAITING  
 \* <ul>  
 \* <li>{****@link*** *#sleep Thread.sleep}</li>  
 \* <li>{****@link*** *Object#wait(long) Object.wait} with timeout</li>  
 \* <li>{****@link*** *#join(long) Thread.join} with timeout</li>  
 \* <li>{****@link*** *LockSupport#parkNanos LockSupport.parkNanos}</li>  
 \* <li>{****@link*** *LockSupport#parkUntil LockSupport.parkUntil}</li>  
 \* </ul>  
 \*/  
 TIMED\_WAITING*,  
  
 */\*\*  
 \* 销毁  
 \*/  
 TERMINATED*;  
}

### 线程启动

*/\*\*  
 \* 启动一个新线程，新线程会执行相应的run()方法。start()不能被重复调用。  
 \* start()实际上是通过本地方法start0()启动线程的。  
 \* 而start0()会新运行一个线程，一旦得到CPU时间片,新线程会调用run()方法。  
 \*/*public synchronized void start() {  
 */\*\*  
 \* 如果线程不是"就绪状态"，则抛出异常！  
 \*/* if (threadStatus != 0)  
 throw new IllegalThreadStateException();  
  
 /\* Notify the group that this thread is about to be started  
 \* so that it can be added to the group's list of threads  
 \* and the group's unstarted count can be decremented. \*/  
 group.add(this);  
  
 boolean started = false;  
 try {  
 start0();  
 started = true;  
 } finally {  
 try {  
 if (!started) {  
 group.threadStartFailed(this);  
 }  
 } catch (Throwable ignore) {  
 /\* do nothing. If start0 threw a Throwable then  
 it will be passed up the call stack \*/  
 }  
 }  
}  
  
@Override  
public void run() {  
 if (target != null) {  
 target.run();  
 }  
}  
  
*/\*\*  
 \* 调用JNI接口  
 \*/*private native void start0();  
*/\*\*  
 \*获取当前执行线程  
 \*/*public static native Thread currentThread();

### wait()/notify()

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“**当前线程**”等待，因为线程进入等待状态，所以线程应该释放它所持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
OK，线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“**唤醒线程**”)，它只有在获取“该对象的同步锁”(**这里的同步锁必须和等待线程的同步锁是同一个**)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

 使用wait()、notify()和notifyAll()时需要先对调用对象加锁，调用wait()方法后会释放锁。

 调用wait()方法之后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列中。

 notify()或notifyAll()方法调用后，等待线程不会立刻从wait()中返回，需要等该线程释放锁之后，才有机会获取锁之后从wait()返回。

 notify()方法将等待队列中的一个等待线程从等待队列中移动到同步队列中；notifyAll()方法则是把等待队列中的所有线程都移动到同步队列中；被移动的线程状态从WAITING变为BLOCKED。

 从wait()方法返回的前提是，改线程获得了调用对象的锁。

**阻塞与等待的区别**

阻塞：当一个线程试图获取对象锁（非java.util.concurrent库中的锁，即synchronized），而该锁被其他线程持有，则该线程进入阻塞状态。它的特点是**使用简单，由JVM调度器来决定唤醒自己，而不需要由另一个线程来显式唤醒自己，不响应中断**。  
等待：当一个线程等待另一个线程通知调度器一个条件时，该线程进入等待状态。它的特点是**需要等待另一个线程显式地唤醒自己，实现灵活，语义更丰富，可响应中断**。例如调用：Object.wait()、Thread.join()以及等待Lock或Condition。

　　需要强调的是虽然synchronized和JUC里的Lock都实现锁的功能，但线程进入的状态是不一样的。**synchronized会让线程进入阻塞态，而JUC里的Lock是用LockSupport.park()/unpark()来实现阻塞/唤醒的，会让线程进入等待态**。但话又说回来，虽然等锁时进入的状态不一样，但被唤醒后又都进入runnable态，从行为效果来看又是一样的。

源码:

*/\*\*  
 \** ***@important*** *调用wait()方法引起"***当前线程等待***",直到另外一个线程调用notify()或notifyAll()唤醒该线程。  
 \* 换句话说，这个方法和wait(0)的效果一样！。  
 \* “当前线程”在调用wait()时，必须拥有该对象的同步锁。该线程调用wait()之后，会释放该锁；  
 \* 然后一直等待直到“其它线程”调用对象的同步锁的notify()或notifyAll()方法。  
 \* 然后，该线程继续等待直到它重新获取“该对象的同步锁”，然后就可以接着运行。  
 \*/*public final void wait() throws InterruptedException {  
 wait(0);  
}  
public final native void wait(long timeout) throws InterruptedException;  
public final native void notify();  
public final native void notifyAll();

### yield()/sleep()/join()/interrupt()

* 1. yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后， 其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！

public static native void yield();

* 1. 使当前线程休眠，即当前线程会从“运行状态”进入到“休眠(阻塞)状态”。sleep()会指定休眠时间，线程休眠的时间会大于/等于该休眠时间；在线程重新被唤醒时， 它会由“阻塞状态”变成“就绪状态”，从而等待cpu的调度执行。不会释放锁

public static native void sleep(long millis) throws InterruptedException;

* 1. 让“主线程”等待“子线程”结束之后才能继续运行。wait()的作用是让“当前线程”等待，而这里的“当前线程”是指当前在CPU上运行的线程。所以，虽然是调用子线程的wait()方法，但是它是通过“主线程”去调用的；所以，休眠的是主线程，而不是“子线程”！

public final synchronized void join(long millis) throws InterruptedException {  
 long base = System.*currentTimeMillis*();  
 long now = 0;  
  
 if (millis < 0) {  
 throw new IllegalArgumentException("timeout value is negative");  
 }  
  
 if (millis == 0) {  
 while (isAlive()) {  
 wait(0);  
 }  
 } else {  
 while (isAlive()) {  
 long delay = millis - now;  
 if (delay <= 0) {  
 break;  
 }  
 wait(delay);  
 now = System.*currentTimeMillis*() - base;  
 }  
 }  
}

* 1. interrupt()的作用是中断本线程。本线程中断自己是被允许的；其它线程调用本线程的interrupt()方法时，会通过checkAccess()检查权限。这有可能抛出SecurityException异常。如果本线程是处于阻塞状态：调用线程的wait(), wait(long)或wait(long, int)会让它进入等待(阻塞)状态， 或者调用线程的join(), join(long), join(long, int), sleep(long), sleep(long, int)也会让它进入阻塞状态。若线程在阻塞状态时，调用了它的interrupt()方法，那么它的“中断状态”会被清除并且会收到一个InterruptedException异常。例如，线程通过wait()进入阻塞状态，此时通过interrupt()中断该线程；调用interrupt()会立即将线程的中断标记设为“true”，但是由于线程处于阻塞状态，所以该“中断标记”会立即被清除为“false”，同时，会产生一个InterruptedException的异常。如果线程被阻塞在一个Selector选择器中，那么通过interrupt()中断它时；线程的中断标记会被设置为true，并且它会立即从选择操作中返回。如果不属于前面所说的情况，那么通过interrupt()中断线程时，它的中断标记会被设置为“true”。中断一个“已终止的线程”不会产生任何操作。

interrupted() 和 isInterrupted()都能够用于检测对象的“中断标记”。

1.interrupted 是作用于当前线程，isInterrupted 是作用于调用该方法的线程对象所对应的线程。（线程对象对应的线程不一定是当前运行的线程。例如我们可以在A线程中去调用B线程对象的isInterrupted方法。）

2.interrupted()会清除中断标记,只有当前线程才能清除自己的标记位

interrupt（）是用来设置中断状态的。返回true说明中断状态被设置了而不是被清除了。我们调用sleep、wait等此类可中断（throw InterruptedException）方法时，一旦方法抛出InterruptedException，当前调用该方法的线程的中断状态就会被jvm自动清除了，就是说我们调用该线程的isInterrupted 方法时是返回false。如果你想保持中断状态，可以再次调用interrupt方法设置中断状态。这样做的原因是，java的中断并不是真正的中断线程，而只设置标志位（中断位）来通知用户。如果你捕获到中断异常，说明当前线程已经被中断，不需要继续保持中断位。  
interrupted是静态方法，返回的是当前线程的中断状态。例如，如果当前线程被中断（没有抛出中断异常，否则中断状态就会被清除），你调用interrupted方法，第一次会返回true。然后，当前线程的中断状态被方法内部清除了。第二次调用时就会返回false。如果你刚开始一直调用isInterrupted，则会一直返回true，除非中间线程的中断状态被其他操作清除了。

public void interrupt() {  
 if (this != Thread.*currentThread*())  
 checkAccess();  
  
 synchronized (blockerLock) {  
 Interruptible b = blocker;  
 if (b != null) {  
 interrupt0(); // Just to set the interrupt flag  
 b.interrupt(this);  
 return;  
 }  
 }  
 interrupt0();  
}

public static boolean interrupted() {  
 return *currentThread*().isInterrupted(true);  
}  
public boolean isInterrupted() {  
 return isInterrupted(false);  
}

* 1. 其他JNI方法

/\* Some private helper methods \*/  
private native void setPriority0(int newPriority);  
private native void stop0(Object o);  
private native void suspend0();  
private native void resume0();  
private native void interrupt0();  
private native void setNativeName(String name);  
public final native boolean isAlive();

### Thread.stop()

**1. 即刻抛出ThreadDeath异常，在线程的run()方法内，任何一点都有可能抛出ThreadDeath Error，包括在catch或finally语句中。**

**2. 会释放该线程所持有的所有的锁，而这种释放是不可控制的，非预期的。**

**Java中多线程锁释放的条件：**

1）执行完同步代码块，就会释放锁。（synchronized）  
2）在执行同步代码块的过程中，遇到异常而导致线程终止，锁也会被释放。（exception）  
3）在执行同步代码块的过程中，执行了锁所属对象的wait()方法，这个线程会释放锁，进入对象的等待池。(wait)

从上面的三点我就可以看到stop方法释放锁是在第二点的，通过抛出异常来释放锁，释放该线程所持有的所有的锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。这种方式是不安全的，不可靠的。

### 线程优先级和守护线程

java 中的线程优先级的范围是1～10，默认的优先级是5。“高优先级线程”会优先于“低优先级线程”执行。

java 中有两种线程：**用户线程**和**守护线程**。可以通过isDaemon()方法来区别它们：如果返回false，则说明该线程是“用户线程”；否则就是“守护线程”。  
用户线程一般用户执行用户级任务，而守护线程也就是“后台线程”，一般用来执行后台任务。需要注意的是：Java虚拟机在“用户线程”都结束后会后退出。

JDK 中关于线程优先级和守护线程的介绍如下：

每个线程都有一个优先级。“高优先级线程”会优先于“低优先级线程”执行。每个线程都可以被标记为一个守护进程或非守护进程。在一些运行的主线程中创建新的子线程时，子线程的优先级被设置为等于“创建它的主线程的优先级”，当且仅当“创建它的主线程是守护线程”时“子线程才会是守护线程”。

当Java虚拟机启动时，通常有一个单一的非守护线程（该线程通过是通过main()方法启动）。JVM会一直运行直到下面的任意一个条件发生，JVM就会终止运行：

(01) 调用了exit()方法，并且exit()有权限被正常执行。

(02) 所有的“非守护线程”都死了(即JVM中仅仅只有“守护线程”)。

每一个线程都被标记为“守护线程”或“用户线程”。当只有守护线程运行时，JVM会自动退出。

//最小优先级  
public final static int *MIN\_PRIORITY* = 1;  
//默认优先级  
public final static int *NORM\_PRIORITY* = 5;  
//最大优先级  
public final static int *MAX\_PRIORITY* = 10;  
  
public final void setPriority(int newPriority) {  
 ThreadGroup g;  
 checkAccess();  
 if (newPriority > *MAX\_PRIORITY* || newPriority < *MIN\_PRIORITY*) {  
 throw new IllegalArgumentException();  
 }  
 if((g = getThreadGroup()) != null) {  
 if (newPriority > g.getMaxPriority()) {  
 newPriority = g.getMaxPriority();  
 }  
 setPriority0(priority = newPriority);  
 }  
}

cpu在执行t1和t2的时候，根据时间片轮循调度，所以能够并发执行。

public final void setDaemon(boolean on) {  
 checkAccess();  
 if (isAlive()) {  
 throw new IllegalThreadStateException();  
 }  
 daemon = on;  
}

(01) 主线程main是用户线程，它创建的子线程t1也是用户线程。  
(02) t2是守护线程。在“主线程main”和“子线程t1”(它们都是用户线程)执行完毕，只剩t2这个守护线程的时候，JVM自动退出。

### 生产者消费者问题

### Synchronized

* 1. **原理**

**在java中，每一个对象有且仅有一个同步锁。这也意味着，同步锁是依赖于对象而存在。**  
**当我们调用某对象的synchronized方法时，就获取了该对象的同步锁。**例如，synchronized(obj)就获取了“obj这个对象”的同步锁。  
**不同线程对同步锁的访问是互斥的。**也就是说，某时间点，对象的同步锁只能被一个线程获取到！通过同步锁，我们就能在多线程中，实现对“对象/方法”的互斥访问。 例如，现在有两个线程A和线程B，它们都会访问“对象obj的同步锁”。假设，在某一时刻，线程A获取到“obj的同步锁”并在执行一些操作；而此时，线程B也企图获取“obj的同步锁” —— 线程B会获取失败，它必须等待，直到线程A释放了“该对象的同步锁”之后线程B才能获取到“obj的同步锁”从而才可以运行。

* 1. **基本规则**

[第一条](http://www.cnblogs.com/skywang12345/p/3479202.html#a21): 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程对**“该对象”的该“synchronized方法”或者“synchronized代码块”的访问**将被阻塞。  
第二条: 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程仍然**可以访问“该对象”的非同步代码块**。  
第三条: 当一个线程访问“某对象”的“synchronized方法”或者“synchronized代码块”时，其他线程对**“该对象”的其他的“synchronized方法”或者“synchronized代码块”的访问**将被阻塞。

* 1. **实例锁和全局锁**

**实例锁**-- 锁在某一个实例对象上。**如果该类是单例，那么该锁也具有全局锁的概念**。实例锁对应的就是synchronized关键字。  
**全局锁** -- 该锁针对的是类，无论实例多少个对象，那么线程都共享该锁。全局锁对应的就是static synchronized（或者是锁在该类的class或者classloader对象上）。

* 1. **内存可见性**

线程获取锁时,JMM会吧该线程对应的本地内存置为无效,从而使得被监视器保护的临界区代码必须从主内存中读取共享变量值

线程释放锁时,JMM把该线程对应的工作内存中的数据刷新到主内存中

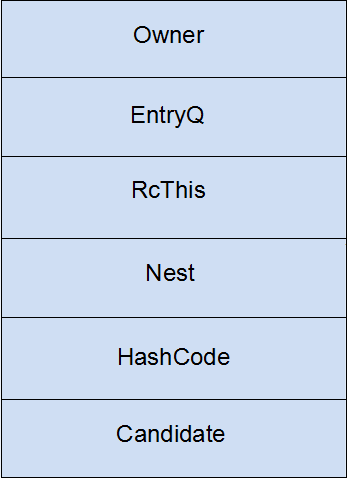
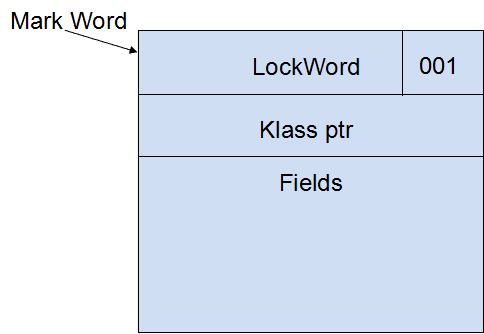
* 1. **个人理解**

同步代码使用**monitoenter**和**monitorexit**指令实现

同步方法使用**ACC\_SYNCHRONIZED** 实现

本质上都是对指定对象相关联的monitor的获取.

线程一旦进入被synchronized修饰的方法或代码块,指定锁对象将对象头中的lockword与monitor关联起来,monitor中的owner存放拥有该锁的线程的唯一标识,确保一次只有一个线程拥有代码的执行权.



对象头 **monitor**

### 死锁

* + - * 1. **概念**

所谓死锁,就是多个线程循环等待它方占用的资源而无限期的僵持下去的局面

* + - * 1. **产生死锁的必要条件**
* 〈1〉互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如CD-ROM驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。
* 〈2〉不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。
* 〈3〉占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。
* 〈4〉循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，......，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

# Java8

## Lambda

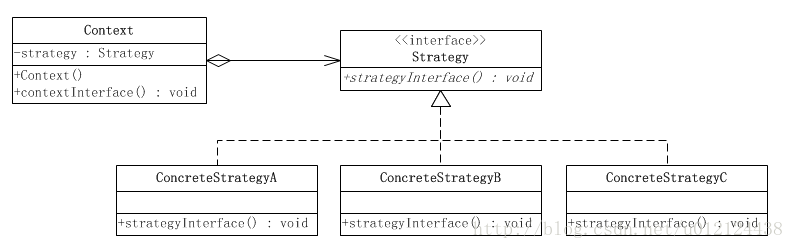
## Java.util.function

# Java9

# 设计模式

## 策略模式

**UML类图**



这个模式涉及到三个角色：

环境(Context)角色：持有一个Strategy的引用。

抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

具体策略(ConcreteStrategy)角色：包装了相关的算法或行为。

项目实例:

AbstractProcessHandler.doProcess0

## 责任链模式

使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系，

将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理他为止。

角色

抽象处理者角色(Handler)：定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个Java抽象类或者Java接口实现。

具体处理者角色(ConcreteHandler)：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

实例: netty **ChannelPipeline-ChannelHandler**

## 反应器(reactor)/观察者(observer)模式

[**Java**](http://lib.csdn.net/base/java)NIO非堵塞技术实际是采取反应器模式，或者说是观察者(observer)模式为我们监察I/O端口，如果有内容进来，会自动通知我们，这样，我们就不必开启多个线程死等，从外界看，实现了流畅的I/O读写，不堵塞了。

同步和异步区别：有无通知（是否轮询）  
堵塞和非堵塞区别：操作结果是否等待（是否马上有返回值），只是设计方式的不同

NIO 有一个主要的类Selector，这个类似一个观察者，只要我们把需要探知的socketchannel告诉Selector，我们接着做别的事情，当有事件发生时，他会通知我们，传回一组SelectionKey，我们读取这些Key，就会获得我们刚刚注册过的socketchannel，然后，我们从这个Channel中读取数据，接着我们可以处理这些数据。

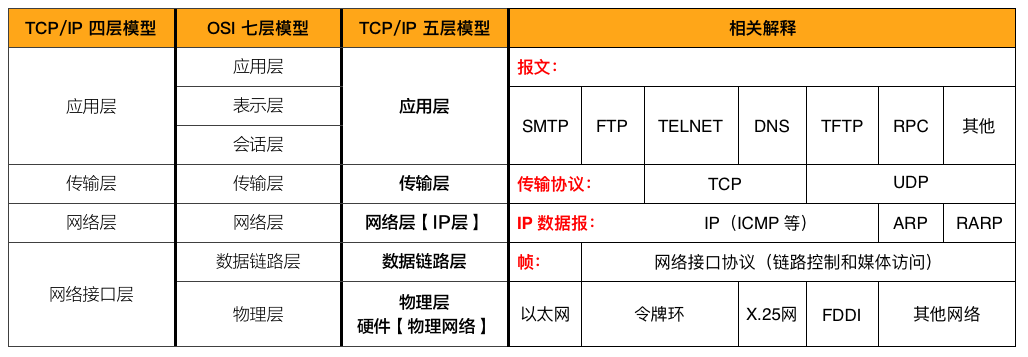
反应器模式与观察者模式在某些方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。

# OpenResty(nginx+lua)

编程式负载均衡, 配置指定服务器接口

# Tcp/Ip Http Socket篇

## 1. 简介



## HTTP「Hypertext Transfer Protocol」超文本传输协议

作用：规定客户端和服务器之间的数据传输格式  
特点：

* 简单快速：HTTP协议简单，HTTP服务器程序小，通信速度快
* 允许传输各种各样的数据
* HTTP 0.9 和 1.0使用 非持续连接，限制每次连接只处理一个请求，请求做出响应后，马上断开，节省传输时间

### 1）发送请求的方法「不同的方法对资源有不同的操作方式」

* PUT：增，给服务器添加资源
* DELETE：删，给服务器删除资源
* POST：改，修改服务器资源
* GET：查，查找服务器资源

### 2）请求参数

传递给服务器的具体数据，比如：帐号、密码  
浏览器和服务器对URL的长度有限制，一般不超过 1K

GET请求参数

* 格式：URL +?+ 参数1=值1 +&+ 参数2=值2「参数间用 & 隔开」  
  多值参数，格式：URL +?+ 参数1=值1 +&+ 参数1=值2
* 安全性差「直接将请求暴露在 URL 里」  
  GET请求的URL一般会记录在服务器的访问日志里，服务器的访问日志是黑客攻击的重点对象之一
* 仅仅是查询数据使用

POST请求参数

* 服务器参数全部放在 **请求体** 中
* 理论上，没有大小限制。实际上，取决于服务器的处理能力
* **安全性好** 即便是 POST也要加密后提交
* 增、删、改服务器数据使用

### 3）HTTP 通信过程 - 请求

I. 请求头：对客户端环境，请求信息的描述

* Host: 120.25.226.186:32812 格式：服务器主机地址: 端口
* GET/minion.png HTTP/1.1 格式：请求方法**/**请求资源路径 HTTP**/**协议版本
* User-Agent: Mozilla/5.0 客户端的类型，客户端的软件环境
* Accept: text/html, \*/\* 客户端所能接收的数据类型
* Accept-Language: zh-cn 客户端的语言环境
* Accept-Encoding: gzip 客户端支持的数据压缩格式

II. 请求体：客户端发给服务器的具体数据，比如：文件数据「POST才会有」

### 4）HTTP 通信过程 - 响应

I. 响应头：服务器的描述，返回数据的描述

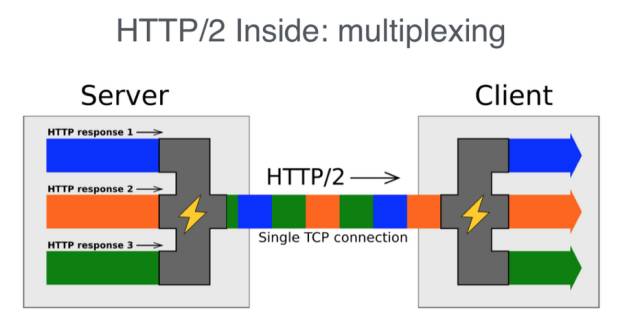
* HTTP/1.1 200 OK 格式：HTTP协议版本 **状态码** 状态名称
* Server: Apache-Coyote/1.1 服务器的类型
* Content-Type: image/jpeg 返回数据的类型
* Content-Length: 56811 返回数据的长度
* Date: Mon, 23 Jun 2014 12:54:52 GMT 响应的时间

II. 响应体：服务器返回客户端的具体数据，比如：文件数据  
III. 常见的响应状态码

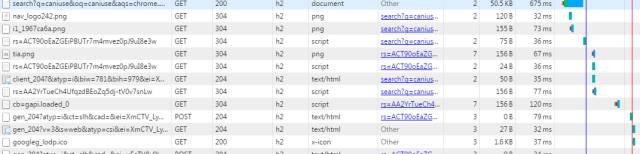
| **状态码** | **英文名称** | **中文描述** |
| --- | --- | --- |
| 200 | OK | 请求成功 |
| 300 |  | 重定向 |
| 400 | Bad Request | 客户端请求语法错误，服务器无法解析 |
| 404 | NotFound | 根据客户端的请求，找不到资源 |
| 500 | Internal Server Error | 服务器内部错误无法完成请求 |

### 5 ) HTTP 2.0

* **新的二进制格式**（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
* HTTP 2.0 二进制分帧层，封装HTTP 消息并在客户端与服务器之间传输
* HTTP2.0 将所有传输的信息分割为更小的消息和帧，并对它们采用二进制格式的编码。
* 注：HTTPS 是二进制分帧的另一个典型示例：所有HTTP 消息都以透明的方式为我们编码和解码，不必对应用进行任何修改。HTTP2.0工作原理有点类似
* **多路复用**（MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。**多路复用原理图**：



* **header压缩，**如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
* **服务端\*\***（server push），同SPDY一样，HTTP2.0也具有server push功能。目前，有大多数网站已经启用HTTP2.0，例如YouTuBe，淘宝网等网站，利用chrome控制台可以查看是否启用H2：



### 6 ) HTTPS

1. HTTPS协议需要到ca申请证书，一般免费证书很少，需要交费。
2. HTTP是超文本传输协议，信息是明文传输，HTTPS 则是具有安全性的ssl加密传输协议。
3. HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. HTTPS的连接很简单，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## TCP Transmission Control Protocol」传输控制协议

简介

* 面向连接的通信协议，通过**三次握手**建立连接，通讯完成时要拆除连接，**四次挥手**断开连接
* TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地
* 使用 TCP 的协议  
  FTP（文件传输协议）、Telnet（远程登录协议）、SMTP（简单邮件传输协议）  
  POP3（和 SMTP 相对，用于接收邮件）、HTTP 协议等

特点

* 每条传输连接只能有两个端点，只能进行点对点的连接，不支持多播和广播的传输方式
* 连接中可以进行大数据传输「数据不受限制」
* 通信双方可以同时发数据和接收数据
* 每次发送请求使用 TCP 协议  
  必须建立连接「效率会稍低」，是**可靠协议**，送达安全

### I. 建立连接-三次握手

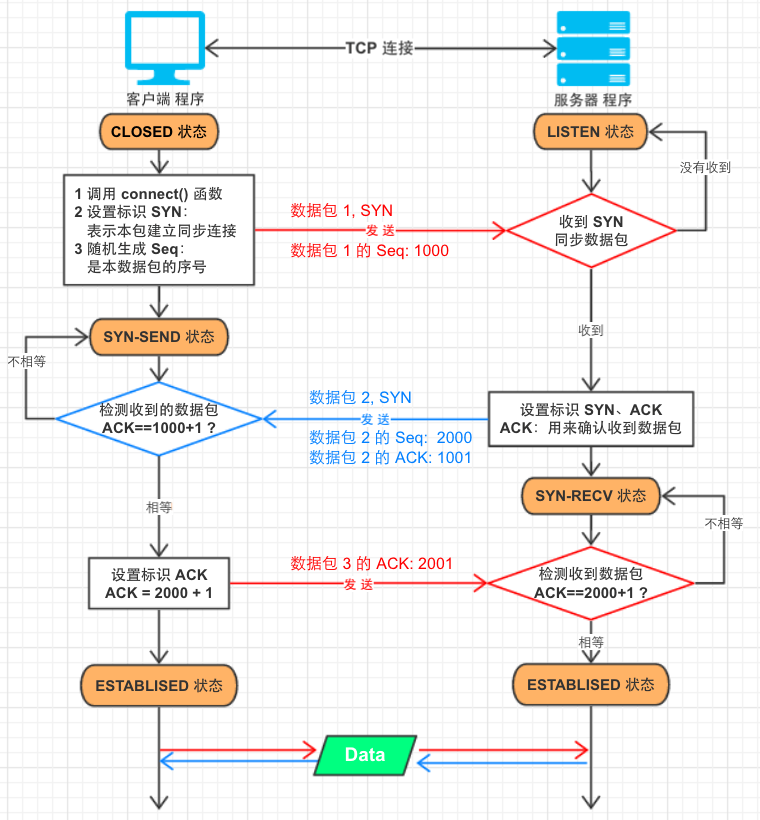
建立连接非常重要，是数据正确传输的前提

模拟步骤

* [Shake 1] 套接字A：“你好，套接字 B，我这里有数据要传送给你，建立连接吧。”
* [Shake 2] 套接字B：“好的，我这边已准备就绪。”
* [Shake 3] 套接字A：“谢谢你受理我的请求。”

具体步骤

* CLOSED 状态：没有建立连接，客户端已经调用 socket() 函数创建了套接字
* LISTEN 状态：服务器端调用 listen() 函数后，开始监听客户端请求  
  服务器没有回传 ACK 包，客户端会重新发送，直到服务器回传 ACK 包
* SYN-SEND 状态：向服务器端发送过了数据包
* SYN-RECV 状态：服务器已经将数据包发出
* ESTABLISED 状态：连接已经成功建立



必要性

* 在「两次握手」的情形下，假设 Client 想跟 Server 建立连接，但是却因为中途连接请求的数据报丢失了，故Client端不得不重新发送一遍，这个时候 Server 端仅收到一个连接请求，因此可以正常的建立连接
* 有时候 Client 端重新发送请求不是因为数据报丢失了，而是有可能数据传输过程因为网络并发量很大在某结点被阻塞了，这种情形下Server端将先后收到 2 次请求，并持续等待两个 Client 请求向他发送数据...
* 问题就在这里，Cient端实际上只有一次请求，而 Server 端却有 2 个响应，极端的情况可能由于 Client 端多次重新发送请求数据而导致 Server端最后建立了 N 多个响应在等待，因而造成极大的资源浪费！

### II. 断开连接-四次挥手

释放不再使用的资源  
不能正常断开，会造成数据传输错误，套接字不能关闭，持续占用资源，如果并发量高，服务器压力堪忧

模拟步骤

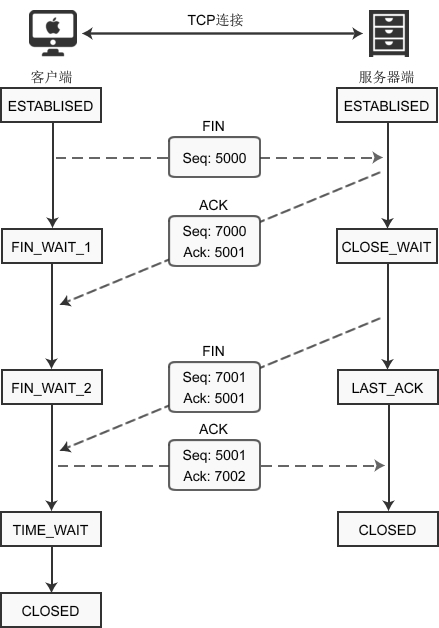
* [Shake 1] 套接字A：“任务处理完毕，我希望断开连接。”
* [Shake 2] 套接字B：“哦，是吗？请稍等，我准备一下。”
* 等待片刻后……
* [Shake 3] 套接字B：“我准备好了，可以断开连接了。”
* [Shake 4] 套接字A：“好的，谢谢合作。”

具体步骤

* FIN 状态：Finish，表示断开连接
* TIME\_WAIT 状态：TIME\_WAIT 要等待 2MSL 才会进入 CLOSED 状态，如果 2MSL 后还未收到服务器重传的 FIN 包，就说明服务器已经收到了 ACK 包  
  客户端最后一次发送 ACK 包时，若服务器收不到，服务器会再次发送 FIN 包  
  如果这时客户端完全关闭了连接，那么服务器无论如何也收不到 ACK 包了，所以客户端需要等待片刻、确认后，才能进入 CLOSED 状态

报文最大生存时间（MSL，Maximum Segment Lifetime）  
数据包在网络中是有生存时间的，超过这个时间还未到达目标主机就会被丢弃，并通知源主机

* CLOSED 状态：断开连接状态



## UDP「User Datagram Protocol」用户数据协议

简介

* 将数据及源和目的封装成数据包中，不需要建立连接
* 使用 UDP协议包括：TFTP（简单文件传输协议）、SNMP（简单网络管理协议）、DNS（域名解析协议）、NFS、BOOTP

特点

* 每个数据报的大小限制在 64K 之内
* 支持单播，组播，广播等多种通信方式
* 每次发送请求使用 UDP 协议  
  无需建立连接，是**不可靠协议**「效率高」

## socket

socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。



* **Socket 是 应用层 和 传输层 之间的桥梁**

HTTP 与 Socket 的区别

1. HTTP 是基于 Socket 的实现；HTTP 应用层协议，主要解决如何包装数据
2. HTTP 传输的数据格式是规定好的，Socket 实现数据传输是最原始，Socket 实现的数据传输格式可自定义
3. Socket 是对 TCP/IP 协议的封装，Socket 本身并不是协议，而是一个调用接口「API」  
   通过 Socket 我们才能使用 TCP/IP 协议

## **HTTP tcp/ip对比**

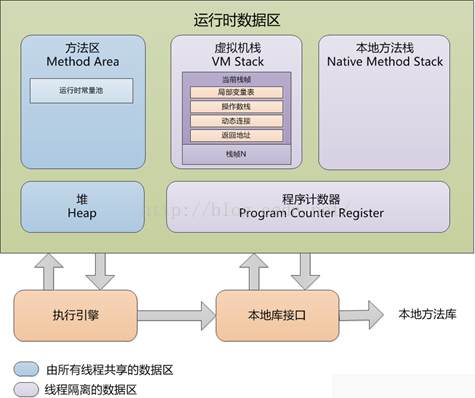
**TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。**关于TCP/IP和HTTP协议的关系，网络有一段比较容易理解的介绍：“我们在传输数据时，可以只使用（传输层）TCP/IP协议，但是那样的话，如果没有应用层，便无法识别数据内容，如果想要使传输的数据有意义，则必须使用到应用层协议，应用层协议有很多，比如HTTP、FTP、TELNET等，也可以自己定义应用层协议。WEB使用HTTP协议作应用层协议，以封装HTTP 文本信息，然后使用TCP/IP做传输层协议将它发到网络上。”

术语TCP/IP代表传输控制协议/网际协议，指的是一系列协议。“IP”代表网际协议，TCP和UDP使用该协议从一个网络传送数据包到另一个网络。把**IP想像成一种高速公路**，它允许其它协议在上面行驶并找到到其它电脑的出口。**TCP和UDP是高速公路上的“卡车”，它们携带的货物就是像HTTP**，文件传输协议FTP这样的协议等。

下面的图表试图显示不同的TCP/IP和其他的协议在最初OSI模型中的位置：

|  |  |  |
| --- | --- | --- |
| 7 | **应用层** | 例如[HTTP](http://zh.wikipedia.org/wiki/HTTP)、[SMTP](http://zh.wikipedia.org/wiki/SMTP)、[SNMP](http://zh.wikipedia.org/wiki/SNMP)、[FTP](http://zh.wikipedia.org/wiki/FTP)、[Telnet](http://zh.wikipedia.org/wiki/Telnet)、[SIP](http://zh.wikipedia.org/wiki/SIP)、[SSH](http://zh.wikipedia.org/wiki/SSH)、[NFS](http://zh.wikipedia.org/wiki/NFS)、[RTSP](http://zh.wikipedia.org/wiki/RTSP)、[XMPP](http://zh.wikipedia.org/wiki/XMPP)、[Whois](http://zh.wikipedia.org/wiki/Whois)、[ENRP](http://zh.wikipedia.org/w/index.php?title=ENRP&action=edit&redlink=1) |
| 6 | **表示层** | 例如[XDR](http://zh.wikipedia.org/w/index.php?title=External_Data_Representation&action=edit&redlink=1)、[ASN.1](http://zh.wikipedia.org/w/index.php?title=Abstract_Syntax_Notation_1&action=edit&redlink=1)、[SMB](http://zh.wikipedia.org/w/index.php?title=Server_message_block&action=edit&redlink=1)、[AFP](http://zh.wikipedia.org/w/index.php?title=Apple_Filing_Protocol&action=edit&redlink=1)、[NCP](http://zh.wikipedia.org/w/index.php?title=NetWare_Core_Protocol&action=edit&redlink=1) |
| 5 | **会话层** | 例如[ASAP](http://zh.wikipedia.org/w/index.php?title=Aggregate_Server_Access_Protocol&action=edit&redlink=1)、[TLS](http://zh.wikipedia.org/wiki/Transport_Layer_Security)、[SSH](http://zh.wikipedia.org/wiki/SSH)、ISO 8327 / CCITT X.225、[RPC](http://zh.wikipedia.org/w/index.php?title=Remote_procedure_call&action=edit&redlink=1)、[NetBIOS](http://zh.wikipedia.org/w/index.php?title=NetBIOS&action=edit&redlink=1)、[ASP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[Winsock](http://zh.wikipedia.org/w/index.php?title=Winsock&action=edit&redlink=1)、[BSD sockets](http://zh.wikipedia.org/wiki/Berkeley_sockets) |
| 4 | **传输层** | 例如[TCP](http://zh.wikipedia.org/wiki/TCP)、[UDP](http://zh.wikipedia.org/wiki/User_Datagram_Protocol)、[RTP](http://zh.wikipedia.org/w/index.php?title=Real-time_Transport_Protocol&action=edit&redlink=1)、[SCTP](http://zh.wikipedia.org/w/index.php?title=Stream_Control_Transmission_Protocol&action=edit&redlink=1)、[SPX](http://zh.wikipedia.org/w/index.php?title=Sequenced_packet_exchange&action=edit&redlink=1)、[ATP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[IL](http://zh.wikipedia.org/w/index.php?title=IL_Protocol&action=edit&redlink=1) |
| 3 | **网络层** | 例如[IP](http://zh.wikipedia.org/wiki/%E7%BD%91%E9%99%85%E5%8D%8F%E8%AE%AE)、[ICMP](http://zh.wikipedia.org/wiki/ICMP)、[IGMP](http://zh.wikipedia.org/wiki/IGMP)、[IPX](http://zh.wikipedia.org/wiki/IPX)、[BGP](http://zh.wikipedia.org/wiki/BGP)、[OSPF](http://zh.wikipedia.org/wiki/OSPF)、[RIP](http://zh.wikipedia.org/wiki/RIP)、[IGRP](http://zh.wikipedia.org/wiki/IGRP)、[EIGRP](http://zh.wikipedia.org/wiki/EIGRP)、[ARP](http://zh.wikipedia.org/wiki/ARP)、[RARP](http://zh.wikipedia.org/wiki/RARP)、 [X.25](http://zh.wikipedia.org/wiki/X.25) |
| 2 | **数据链路层** | 例如[以太网](http://zh.wikipedia.org/wiki/%E4%BB%A5%E5%A4%AA%E7%BD%91)、[令牌环](http://zh.wikipedia.org/wiki/%E4%BB%A4%E7%89%8C%E7%8E%AF)、[HDLC](http://zh.wikipedia.org/wiki/HDLC)、[帧中继](http://zh.wikipedia.org/wiki/%E5%B8%A7%E4%B8%AD%E7%BB%A7)、[ISDN](http://zh.wikipedia.org/wiki/ISDN)、[ATM](http://zh.wikipedia.org/wiki/%E5%BC%82%E6%AD%A5%E4%BC%A0%E8%BE%93%E6%A8%A1%E5%BC%8F)、[IEEE 802.11](http://zh.wikipedia.org/wiki/IEEE_802.11)、[FDDI](http://zh.wikipedia.org/wiki/FDDI)、[PPP](http://zh.wikipedia.org/wiki/PPP) |
| 1 | **物理层** | 例如[线路](http://zh.wikipedia.org/w/index.php?title=%E7%BA%BF%E8%B7%AF&action=edit&redlink=1)、[无线电](http://zh.wikipedia.org/wiki/%E6%97%A0%E7%BA%BF%E7%94%B5)、[光纤](http://zh.wikipedia.org/wiki/%E5%85%89%E7%BA%A4)、[信鸽](http://zh.wikipedia.org/wiki/%E4%BF%A1%E9%B8%BD) |

# JVM内存模型



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **名称** | **特征** | **作用** | **配置参数** | **异常** |
| 程序计数器 | 占用内存小，线程私有，  生命周期与线程相同 | 大致为字节码行号指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。 | 无 | 无 |
| 虚拟机栈 | 线程私有，生命周期与线程相同，使用连续的内存空间 | Java 方法执行的内存模型，存储局部变量表、操作栈、动态链接、方法出口等信息 | -Xss | StackOverflowError  OutOfMemoryError |
| java堆 | 线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址 | 保存对象实例，所有对象实例（包括数组）都要在堆上分配 | -Xms  -Xsx  -Xmn | OutOfMemoryError |
| 方法区 | 线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址 | 存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据 | -XX:PermSize:  16M  -XX:MaxPermSize  64M | OutOfMemoryError |
| 运行时常量池 | 方法区的一部分，具有动态性 | 存放字面量及符号引用 |  |  |

## 内存分配过程

1、JVM 会试图为相关Java对象在Eden Space中初始化一块内存区域。

2、当Eden空间足够时，内存申请结束；否则到下一步。

3、JVM 试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）。释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区。

4、Survivor区被用来作为Eden及Old的中间交换区域，当Old区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区。

5、当Old区空间不够时，JVM 会在Old区进行完全的垃圾收集（0级）。

6、完全垃圾收集后，若Survivor及Old区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现“outofmemory”错误。

# 二进制及运算篇

<< 向左移位

>> 向右移位

>>> 无符号右移 ,忽略符号位并总是以”0”作为填充位

/\* 00000001 << 1 = 00000010 \*/  
 1 << 1 == 2   
   
/\* 00000001 << 3 = 00001000 \*/  
 1 << 3 == 8  
   
/\* 11111111 11111111 11111111 11110000 >> 4 = 11111111 11111111 11111111 11111111 \*/  
 0xFFFFFFF0 >> 4 == 0xFFFFFFFF   
   
/\* 00001111 11111111 11111111 11111111 >> 4 = 00000000 11111111 11111111 11111111 \*/  
 0x0FFFFFFF >> 4 == 0x00FFFFFF

/\* 10000000 00000000 00000000 00000000 >>> 1 = 01000000 00000000 00000000 00000000 \*/  
 0x80000000 >>> 1 == 0x40000000  
   
/\* 10000000 00000000 00000000 00000000 >> 1 = 11000000 00000000 00000000 00000000 \*/  
 0x80000000 >> 1 == 0xC0000000

**注意：**向右移位是有符号操作符。和许多语言一样，Java使用最高位来表示数值的正负，负数的最高位永远为1。一个以1开头的二进制数移位后还将以1开头，一个以0开头的二进制树移位后还将以0开头。所以**要小心**：Java是可以在整数中进行位运算的。

最大的用途之一是迅速求2的幂。1向左移位1位是2，移2位是4，移3位是8…… 相似的，向右移1位相当于是把该数除以2。

另一个用途便是创建掩码。位掩码可用于屏蔽或者修改一个二进制数中的某些指定位，下一部分会进行详细讲解。假如我们想要创建一个  
*00001000*的掩码，代码十分简单：

|  |  |
| --- | --- |
|  | int bitmask = 1 << 3; |

n<<1 = n\*2

n>>1 = n/2取整数

n<<m = n\*2m

n>>m = n/2m

~n = -(n+1) ~10 = -11

* ~ 按位取反
* ~1111 == 0000
* ~0011 == 1100
* & 按位与 取0
* 1010 & 0101 == 0000
* 1100 & 0110 == 0100
* ^ 按位异或 同0异1
* 1010 ^ 0101 == 1111
* 1100 ^ 0110 == 1010
* | 按位或 取1
* 1010 | 0101 == 1111
* 1100 | 0110 == 1110