Reactor模型介绍

# 无处不在的C/S架构

在这个充斥着**云**的时代,我们使用的软件可以说99%都是C/S架构的！

* 你发邮件用的Outlook,Foxmail等
* 你看视频用的优酷，土豆等
* 你写文档用的Office365,googleDoc，Evernote等
* 你浏览网页用的IE,Chrome等(**B/S是特殊的C/S**)
* ......

C/S架构的软件带来的一个明显的好处就是：只要有网络，你可以在任何地方干同一件事。

例如：你在家里使用Office365编写了文档。到了公司，只要打开编辑地址就可以看到在家里编写的文档，进行展示或者继续编辑。甚至在手机上进行阅读与编辑。不再需要U盘拷来拷去了。

C/S架构模型：



* C就是Client(客户端),上面的B是Browser(浏览器)
* S就是Server(服务器)：服务器管理某种资源，并且通过操作这种资源来为它的客户端提供某种服务；

C/S架构之所以能够流行的一个主要原因就是网速的提高以及费用的降低，特别是无线网络速度的提高。试想在2G时代，大家最多就是看看文字网页，小说什么的。看图片，那简直就是奢侈！更别说看视频了！

网速的提高，使得越来越多的人使用网络，例如：优酷，微信都是上亿用户量，更别说天猫双11的瞬间访问量了！这就对服务器有很高的要求！能够快速处理海量的用户请求！那服务器如何能快速的处理用户的请求呢？

# 高性能服务器

高性能服务器至少要满足如下几个需求：

* **效率高**：既然是高性能，那处理客户端请求的效率当然要很高了；
* **高可用**：不能随便就挂掉了；
* **编程简单**：基于此服务器进行业务开发需要足够简单；
* **可扩展**：可方便的扩展功能；
* **可伸缩**：可简单的通过部署的方式进行容量的伸缩，也就是服务需要无状态；

而满足如上需求的一个基础就是高性能的IO!

# Socket

无论你是发邮件，浏览网页，还是看视频～**实际底层都是使用的TCP/IP，而TCP/IP的编程抽象就是Socket**!

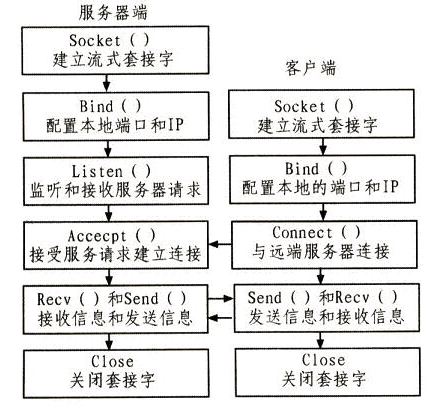
我一直对Socket的中文翻译很困惑，个人觉得是我所接触的技术名词翻译里最莫名其妙的，没有之一！**Socket**中文翻译为"**套接字**"！什么鬼？在很长的时间里我都无法将其和网络编程关联上！后来专门找了一些资料，最后在知乎上找到了一个还算满意的答案(具体链接<https://www.zhihu.com/question/21383903>)！

Socket的原意是插口，想表达的意思是插口与插槽的关系！"send socket"插到"receive socket"里，建立了链接，然后就可以通信了！

套接字的翻译，应该是参考了套接管(如下图)！从这个层面上来看，是有那么点意思！

套接字这个翻译已经是标准了，不纠结这个了！

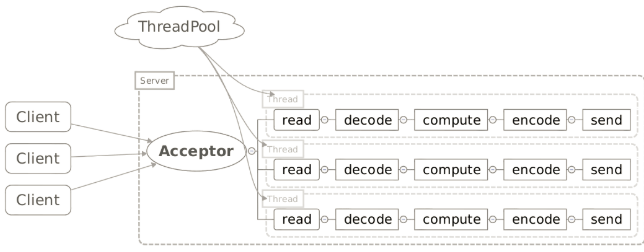
我们看一下Socket之间建立链接及通信的过程！实际上就是对TCP/IP连接与通信过程的抽象:



* 服务端Socket会bind到指定的端口上，Listen客户端的"插入"
* 客户端Socket会Connect到服务端
* 当服务端Accept到客户端连接后
* 就可以进行发送与接收消息了
* 通信完成后即可Close

# IO流分类：BIO、NIO、AIO

## BIO



BIO优缺点

优点：模型简单；编码简单；

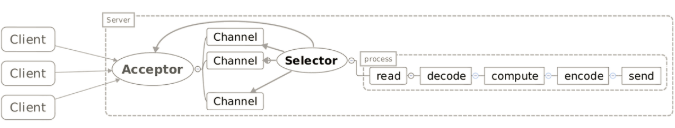
缺点：性能瓶颈低。

优缺点很明显。这里主要说下缺点：主要瓶颈在线程上。每个连接都会建立一个线程。虽然线程消耗比进程小，但是一台机器实际上能建立的有效线程有限，以Java来说，1.5以后，一个线程大致消耗1M内存！且随着线程数量的增加，CPU切换线程上下文的消耗也随之增加，在高过某个阀值后，继续增加线程，性能不增反降！而同样因为一个连接就新建一个线程，所以编码模型很简单！

就性能瓶颈这一点，就确定了BIO并不适合进行高性能服务器的开发！像Tomcat这样的Web服务器，从7开始就从BIO改成了NIO，来提高服务器性能！

后来引入了线程池和消息队列作为改进。

## NIO模型



* Acceptor注册Selector，监听accept事件
* 当客户端连接后，触发accept事件
* 服务器构建对应的Channel，并在其上注册Selector，监听读写事件
* 当发生读写事件后，进行相应的读写处理

NIO优缺点

优点：性能瓶颈高

缺点：模型复杂；编码复杂；**需处理半包问题**

NIO的优缺点和BIO就完全相反了!性能高，不用一个连接就建一个线程，可以一个线程处理所有的连接！相应的，编码就复杂很多，从上面的代码就可以明显体会到了。还有一个问题，**由于是非阻塞的，应用无法知道什么时候消息读完了，就存在了半包问题**！

## 半包问题



我们知道TCP/IP在发送消息的时候，可能会拆包(如上图1)！这就导致接收端无法知道什么时候收到的数据是一个完整的数据。例如:发送端分别发送了ABC,DEF,GHI三条信息，发送时被拆成了AB,CDRFG,H,I这四个包进行发送，接受端如何将其进行还原呢？在BIO模型中，当读不到数据后会阻塞，而NIO中不会!所以需要自行进行处理!例如，以换行符作为判断依据，或者定长消息发生，或者自定义协议！

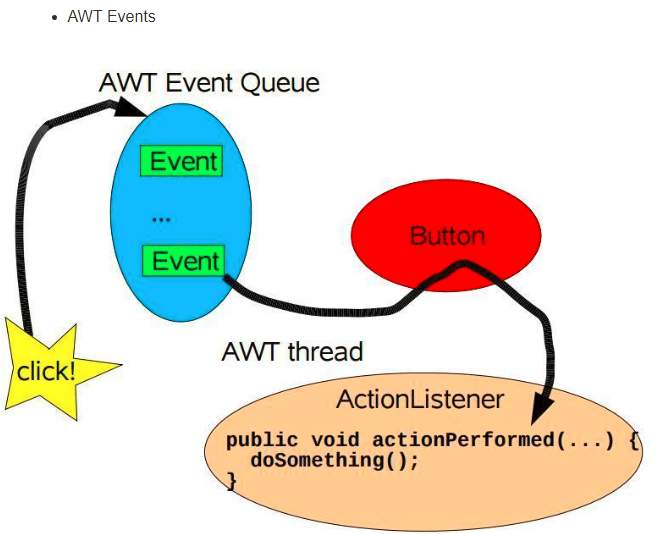
NIO虽然性能高，但是编码复杂，且需要处理半包问题！为了方便的进行NIO开发，就有了**Reactor模型**!

# Reactor模型

首先要需要说明的是，**reactor线程模型并不是netty所独有，其是一种并发编程模型，更确切的或者说一种思想**，其具有的是指导意义，开发者需要在这种编程模型思想的指导下，结合自己的实际场景，来进行合理的设计。在不同的场景下，可能设计出来的reactor线程模型是不一样的，例如scala中的akka框架，就是基于reactor线程模型的思想设计的。换句话说，**netty只是结合了nio网络编程的特点，合理的应用了reactor线程模型**。

现在我们还是需要简单的介绍一下有**哪些典型的reactor线程模型设计方式**。回顾串行工作者模型和并行工作者模型，它们主要的关注点是：**划分任务的接受阶段与任务的处理阶段**。也正是因为如此，我们通常将接受任务的线程称之为Accpet Thread。而任务的处理过程都是一个线程(worker thread)内完成的。

reactor线程模型关注的是：任务接受之后，对处理过程继续进行切分，划分为多个不同的步骤，每个步骤用不同的线程来处理，也就是原本由一个线程处理的任务现在由多个线程来处理，每个线程在处理完自己的步骤之后，还需要将任务转发到线程继续进行处理。为了进行区分，在reactor线程模型中，处理任务并且分发的线程，不再称之为worker thread，而是**reactor thread**。



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！

## 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**。

对应上面的NIO代码来看:

**Reactor**：相当于有分发功能的Selector

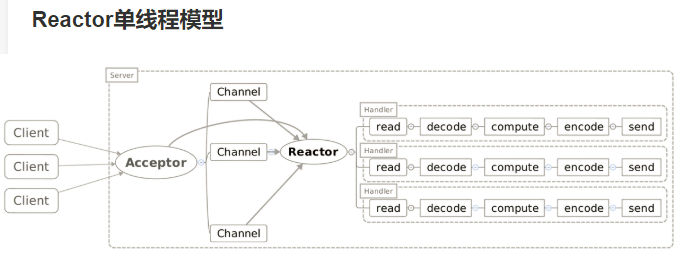
**Acceptor**：NIO中建立连接的那个判断分支

**Handler**：消息读写处理等操作类

Reactor从线程池和Reactor的选择上可以细分为如下几种：

## Reactor单线程模型

单线程reactor线程模型，之所以称之为单线程，还是因为**只有一个accpet Thread接受任务**，之后转发到reactor线程中进行处理。

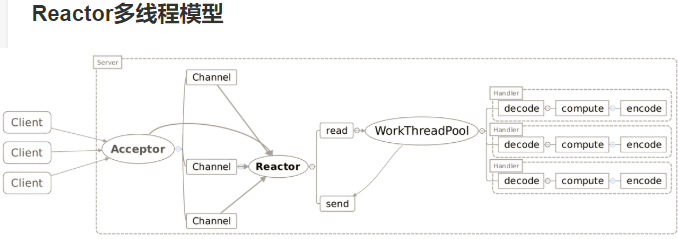


这个模型和上面的NIO流程很类似，只是将消息相关处理独立到了Handler中去了！

虽然上面说到**NIO一个线程就可以支持所有的IO处理**。但是瓶颈也是显而易见的！我们看一个客户端的情况，如果这个客户端多次进行请求，如果在Handler中的处理速度较慢，那么后续的客户端请求都会被积压，导致响应变慢！所以引入了Reactor多线程模型!

## Reactor多线程模型

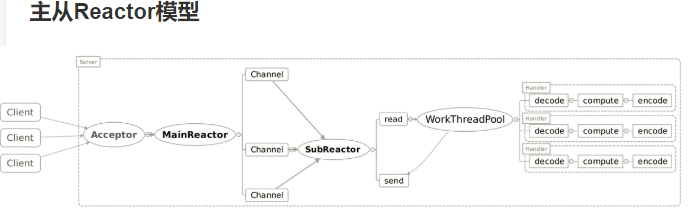
所谓多线程reactor线程模型，无非就是有多个accpet线程。



Reactor多线程模型就是将Handler中的IO操作和非IO操作分开，操作IO的线程称为IO线程，非IO操作的线程称为工作线程!这样的话，客户端的请求会直接被丢到线程池中，客户端发送请求就不会堵塞！

但是当用户进一步增加的时候，Reactor会出现瓶颈！因为Reactor既要处理IO操作请求，又要响应连接请求！为了分担Reactor的负担，所以引入了主从Reactor模型!

## 主从Reactor模型



主Reactor用于响应连接请求，从Reactor用于处理IO操作请求！

**混合型reactor线程模型，**实际上最能体现reactor线程模型的本质：

将任务处理切分成多个阶段进行，每个阶段处理完自己的部分之后，转发到下一个阶段进行处理。不同的阶段之间的执行是异步的，可以认为每个阶段都有一个独立的线程池。

不同的类型的任务，有着不同的处理流程，划分时需要划分成不同的阶段

# Netty

Netty是一个高性能NIO框架，其是对Reactor模型的一个实现！

从Netty服务器代码来看，与Reactor模型进行对应！

EventLoopGroup就相当于是Reactor，bossGroup对应主Reactor,workerGroup对应从Reactor

TimeServerHandler就是Handler

child开头的方法配置的是客户端channel，非child开头的方法配置的是服务端channel

