# 服务器设计框架

## 设计目标

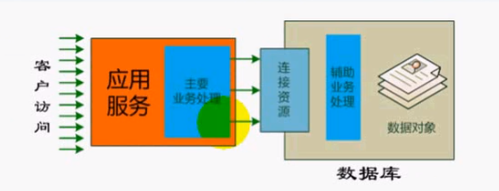
高并发服务器设计目标：

1. 高性能（High Performance）
2. 高可用（High Availability）
3. 伸缩性（Scalability）

## 架构演变

### C/S+B/S架构

一个典型的服务器结构：



技术：网络I/O（epoll）+服务器高性能编程技术+数据库

数据库可能出现的问题：

1. 超出数据库连接数：数据库并发连接数10个，应用服务器这边有1000个并发请求，将会有990个请求失败；
2. 超出时限：数据库并发连接数10个，数据库1秒中之内最能处理1000个请求，应用服务器这边有10000个并发请求，会出现0~10秒的等待。

针对上述问题，可以采用的解决方案：

1. DAL队列服务+连接池
2. 主要业务逻辑搬到应用服务器处理，数据库只做辅助的业务处理
3. 缓存

### 分布式架构

## 规范

服务器程序规范涉及：

1. 以后台形式运行
2. 有日志系统
3. 以非root身份运行
4. 可配置
5. 需要考虑资源的使用

## syslog

应用程序使用syslog函数和守护进程通信：

#include <syslog.h>

void syslog(int priority, const char\* message, …);



## 后台进程

服务器程序后台化的过程：

1. 创建子进程，关闭父进程
2. 设置文件权限掩码
3. 创建新的会话
4. 切换工作目录
5. 关闭标准输入、输出设备
6. 关闭其他打开的文件描述符

## 设计模型

### 循环式模式



这种模式没有充分利用多核CPU，不适合执行时间长的服务，比较适用于短连接的服务，如果需要长连接则需要在read和write之间不断循环，只能服务一个客户端。

高性能网络服务器不能采用循环式模式。

### 并发式模型

高并发服务器模型：

1. 多进程并发服务器

客户端操作基本步骤（客户端无特殊操作）：

1. socket创建监听套接字
2. bind绑定地址结构
3. listen设置监听上限
4. accept创建cfd负责实际的socket连接处理（每个accept都创建一个进程）

while(1)

{

accept(); //接收客户端请求

pid = fork();

if(0==pid)

//处理结果

}

1. 子进程：close(lfd)

父进程：close(cfd)，注册信号捕获函数SIGCHLD，在回调函数中完成子进程回收

1. 多线程并发服务器

服务端基本操作：

1. socket
2. bind
3. listen
4. accept

while(1)

{

cfd = accept(lfd);

pthread\_create(&tid,NULL,tfn,NULL);

}

1. 子线程

void \*tfn(void \*args)

{

close(lfd);

read();

}

1. 多路IO复用服务器：select/poll/epoll



并发式模型每一个连接都有一个进程或线程来处理连接，在建立和销毁线程或进程时需要建立和释放连接，消耗也比较大。

### Reactor模型(epoll)



Reactor模型可以在一个线程中并发处理多个请求，缺点是没有充分利用多核CPU，不适用执行时间比较长的服务。

注：Reactor是纯异步。

### Reactor+threads模型

Reactor模型只适用于执行时间短的服务，如果需要处理长时间连接请求，可以引入线程池：



该模型的缺点在于Reactor只能在一个线程中，不能充分利用多核CPU。

### Reactors in thread模型



### 对比



## 功能模块





# I/O模型

一个IO操作其实分成了两个步骤：发起IO请求和实际的IO操作。

阻塞IO和非阻塞IO的区别在于第一步：发起IO请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞IO；如果不阻塞，那么就是非阻塞IO。

同步IO和异步IO的区别就在于第二个步骤是否阻塞，如果实际的IO读写阻塞请求进程，那么就是同步IO，因此阻塞IO、非阻塞IO、IO复用、信号驱动IO都是同步IO；如果不阻塞，而是操作系统帮你做完IO操作再将结果返回给你，那么就是异步IO。

## BIO

**客户端代码：**

//Bind，Connect

Socket client = new Socket("127.0.0.1",7777);

//读写

PrintWriter pw = new PrintWriter(client.getOutputStream());

BufferedReader br=

new BufferedReader(new InputStreamReader(System.in));

pw.write(br.readLine());

//Close

pw.close();

br.close();

**服务端代码：**

Socket socket;

//Bind，Listen

ServerSocket ss = new ServerSocket(7777);

while (true) {

//Accept

socket = ss.accept();

//一般新建一个线程执行读写

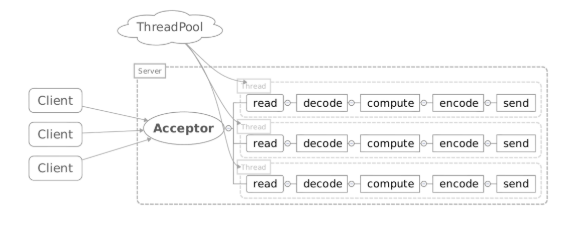
BufferedReader br = new BufferedReader(

new InputStreamReader(socket .getInputStream()));

System.out.println("you input is : " + br.readLine());

}

模型图如下所示：



**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！

## NIO

**NIO客户端代码（连接）：**

//获取socket通道

SocketChannel channel = SocketChannel.open();

channel.configureBlocking(false);

//获得通道管理器

selector=Selector.open();

channel.connect(new InetSocketAddress(serverIp, port));

//为该通道注册SelectionKey.OP\_CONNECT事件

channel.register(selector, SelectionKey.OP\_CONNECT);

**NIO客户端代码（监听）：**

while(true){

//选择注册过的io操作的事件(第一次为SelectionKey.OP\_CONNECT)

selector.select();

while(SelectionKey key : selector.selectedKeys()){

if(key.isConnectable()){

SocketChannel channel=(SocketChannel)key.channel();

if(channel.isConnectionPending()){

channel.finishConnect();//如果正在连接，则完成连接

}

channel.register(selector, SelectionKey.OP\_READ);

}else if(key.isReadable()){ //有可读数据事件。

SocketChannel channel = (SocketChannel)key.channel();

ByteBuffer buffer = ByteBuffer.allocate(10);

channel.read(buffer);

byte[] data = buffer.array();

String message = new String(data);

System.out.println("recevie message from server:, size:"

+ buffer.position() + " msg: " + message);

}

}

}

**NIO服务端代码（连接）：**

//获取一个ServerSocket通道

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

serverChannel.socket().bind(new InetSocketAddress(port));

//获取通道管理器

selector = Selector.open();

//将通道管理器与通道绑定，并为该通道注册SelectionKey.OP\_ACCEPT事件，

serverChannel.register(selector, SelectionKey.OP\_ACCEPT);

**NIO服务端代码（监听）：**

while(true){

//当有注册的事件到达时，方法返回，否则阻塞。

selector.select();

for(SelectionKey key : selector.selectedKeys()){

if(key.isAcceptable()){

ServerSocketChannel server =

(ServerSocketChannel)key.channel();

SocketChannel channel = server.accept();

channel.write(ByteBuffer.wrap(

new String("send message to client").getBytes()));

//在与客户端连接成功后，为客户端通道注册SelectionKey.OP\_READ事件。

channel.register(selector, SelectionKey.OP\_READ);

}else if(key.isReadable()){//有可读数据事件

SocketChannel channel = (SocketChannel)key.channel();

ByteBuffer buffer = ByteBuffer.allocate(10);

int read = channel.read(buffer);

byte[] data = buffer.array();

String message = new String(data);

System.out.println("receive message from client, size:"

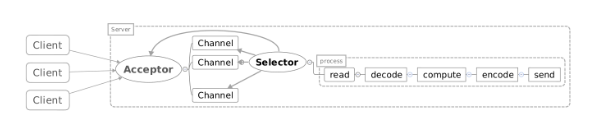
+ buffer.position() + " msg: " + message);

}

}

}

NIO模型示例如下：



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

**优点：**

性能瓶颈高

**缺点：**

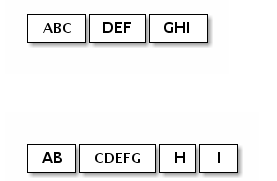
模型复杂

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！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！

## AIO

# 事件处理模型

Reactor模式是处理并发I/O比较常见的一种模式，用于同步I/O，中心思想是将所有要处理的I/O事件注册到一个中心I/O多路复用器上，同时主线程/进程阻塞在多路复用器上；一旦有I/O事件到来或是准备就绪(文件描述符或socket可读、写)，多路复用器返回并将事先注册的相应I/O事件分发到对应的处理器中。

Reactor是一种事件驱动机制，和普通函数调用的不同之处在于：应用程序不是主动的调用某个API完成处理，而是恰恰相反，Reactor逆置了事件处理流程，应用程序需要提供相应的接口并注册到Reactor上，如果相应的事件发生，Reactor将主动调用应用程序注册的接口，这些接口又称为“回调函数”。用“好莱坞原则”来形容Reactor再合适不过了：不要打电话给我们，我们会打电话通知你。

Reactor模式与Observer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 。

服务器一般处理三类事件：I/O，信号，定时器事件

## 背景

从计算机出现伊始，以Dijikstra为代表的早期程序员们准备拿它用来作一些正儿八经的事情的时候，就被事务交互所带来的并发性所困扰着。同时运行的并发事务会在不同时间或空间产生交互，无法进行直观的描述。为了保证并发系统设计的正确性人们试图通过形式化的模型来规范它。

包括有比较早出现的是曾经小圈子盛行的petri-net，它是60年代就提出来的，还有70年代提出了Actor、CSP，以及较晚提出的Pi算子等并发模型。形式化模型虽然可以保证正确性，但如何落地，特别是高效的实现它是个问题；有些模型随着系统的复杂，自身也会变得复杂起来，所以无法在整个系统里采用统一的模型；应用模型设计以及使用相关工具需要专业知识；等上述因素导致实际工程中，一般不会拿模型做指导。

比较早把并发模型作为一个通用的编程基础设施的，就是大名鼎鼎的Erlang了。它从语言层面上引入了Actor模型。

随后，Golang的设计者在Hoare的建议下，实现了CSP模型的一些概念。

其实这些较晚出现的并发模型之间有着比较多的相似之处，不仅可以相互模拟，而且实现中也使用了一些本质上相同的机制。比如Scala akka的作者受Erlang的启发，搞了一套基于多线程的Actor模型，也借用了CSP模型里的Channel的概念。这些语言上支持的并发模型－其实准确应该说是机制，因为不是绝对意义上的形式化模型，给开发大规模并发系统带来了很多便利，它们也被称为是统一线程和事件驱动的并发模型。

事件驱动和线程在实现并发系统中的优劣曾经是工程领域里长期争论的一个话题。的确不深入体体会的话很难在两者之间取舍。事件驱动是跟客观世界打交道必不可少的手段，比如计算机输入输出系统依赖的中断机制，就是一种典型的事件驱动，如果采用其他方式，比如轮询机制的话就丧失了并发性，或者造成处理上的繁琐。相反，回到计算机内部，独立执行的逻辑显然采用线程更加容易，事件驱动机制会把代码搞的支离破碎。

但从理论上说，本质上两者是等价的，它们都是调度代码执行的手段，或者说在计算机里模拟并发性的手段。事件驱动比较擅长处理的问题，一般叫做异步事件，就是事件发生和不受程序执行流程的影响；而线程适合处理事务之间有着明显因果关系或者相关性很强的场合；所以，当需要处理跨事件的事务逻辑时，事件驱动都会变得复杂难解；反之，处理那种跟代码当前处理流程不相关的东西，使用事件的方式会让更加简单明了，比如类Unix常见的signal()系统调用；还是比如异常，它本质是上也是一种事件。

相对线程而言，事务驱动有一点优势的地方，就是它与生俱来的并发性和灵活性，主要表现在处理外部输入输出上。线程，更加准确的说是分时调度或抢占式调度机制是通过模拟一个个独立的个体来表现并发性的；而事件驱动的并发性则可视做是和个体以外的其他客体交互表现出来的。所以在GUI场合下，为了可以模拟桌面上出现任意动作，采用了事件驱动。在事件驱动机制擅长的领域，会采用事件循环这样一种同步调度或者协作式的调度机制，所以大多数人不用考虑一个完整系统下的需要考虑的并发或者共享问题，这些问题已经被系统底层处理好了。同时，我们也知道大部分时候，尽管不是全部时候，这些事件之间不存在强逻辑关系。这种单一数据集，不可预测的多事件输入场景下几乎只能采用事件驱动，如果采用线程机制的话，不但享受不到顺序流程带来的便利性，反而会因为同步带来诸多问题。

知道两者的优劣之后，人们就自然会想到，能不能统一这两者，让我们在需要的时候使用趁手的机制。但是，一旦把事件驱动和线程结合起来，人们往往就会发现，一方面带来了同步问题或交互问题，还有一方面遭遇阻塞情况下，大量使用线程导致性能劣化。异步编程带来的复杂性是众所周知的，聪明的系统开发者是不会把这种问题交给上层应用的，所以，在事件驱动没那么有必要的年代，表现出来的系统调用都是同步的，当然也是阻塞的。这也是为什么后来线程大行其道的原因，事件驱动遭遇阻塞调用的情况下，就丧失它那些自然的优势。所以有人说大多数情况下线程不是个好主意，有人就说事件对高并发服务器场合来说就是个坏主意。现在人们知道，并发模型都可以视作是事件驱动的，它们大多依赖于一些消息队列，管道，邮箱这些设施，来保证并发事务间正确交互和同步所需要的特性，或者更底层一些，是事务操作这些同步设施时使用接口的相对原子性，保证了交互时不会出现使用共享内存中锁互斥带来了一些问题。

所以Erlang处理这个问题是粗野而霸道的，通过取消线程，直接断绝了共享的念头；再通过轻量级进程，允许大量的使用进程，或异步调度，来避免到阻塞调用的带来的问题；最后再提供一个良好的消息机制，满足事务交互或同步的需要。最终提供了一个既灵活又足够高效的并发体系。当然，在一般人看来Erlang的成功毕竟只是在特定场合下的，没人会感到，这会对系统层面和一些关键应用带来什么影响。但影响必定是有的，首先有人在haskell下实现了线程和事件驱动的统一。haskell因为不允许副作用，所以在任意时候采用线程都是可行而且对用户透明的。这个实现通过运行时系统提供应用级线程调度，然后使用monad来表达顺序逻辑状态演变，赋予do记号线程执行特性；同时在事件循环中可以使用映射或者模式匹配表达事件处理过程以及赋予事件处理过程的CPS monad同样的线程执行特性，来提供高并发性。紧接着不甘人后的Scala就照葫芦画瓢又搞了一个，虽然在这里事件和线程被赋予统一的actor语义。Golang的goroutine本质上也是一样的，只是它没有函数式那么多特性来限制用户而已。一般认为Actor模型更像事件驱动，因为消息发送和接收是异步的，但这不是关键。这些事件驱动和线程统一实现的最大共同点就在于它们都允许大量的线程/进程的存在，当然，为了避免性能问题，这语言都依赖于其运行时系统实现的所谓轻量级的线程/进程。由于这个因素，线程间的通信无论是异步还是阻塞的，除了是否同步外，差异已经非常小了。眼下，大多数流行的解释性的或者自带运行时的语言中具备上都具备了类似的特性。但除此之外，一些人也在跃跃欲试的给系统开发层面的工具包里添加上同样强大的功能东西，协程就是其中一个眼下普遍流行的概念。从调度角度而言，相对线程而言，协程其实就是一种同步调度器，或者协作式调度器，可以在模拟并发的同时不需要引入额外的并发控制机制，数据的一致性是程序主动提供的。另外一个角度，它也是事件驱动的一种特例，或者说是隐式事件驱动。然而，协程实现需要面对一些问题，首先是阻塞调用的问题，一些协程实现中，通过重写了一整套接口替换掉系统调用来实现应用级调度，能这么做源于系统调用对程序来说其实和管道一样是一个原子性的接口。另外一个问题就是想要享受多核CPU的并行好处的话，还要面对系统级线程间通信问题，就是需要和golang类似的管道机制。虽然实现一个管道并不困难，但毕竟也有不少细节问题需要考虑，包括额外带来的开销。直接使用管道其实还有一些问题，比如说它不是解耦的，需要预先的考量和设计；以及golang的管道或者说这个概念本身就存在一些问题，它一会是同步的，一会又是异步的，带缓冲的，但这个缓冲又是有限的，也会阻塞的。尽管上面说了，在非系统级调度上它们之间差异很小，但相比把调度开销视为必然的应用级调度而言系统线程的作用本来是希望提供并行性。一种具有完全异步保证的机制，才可能充分利用计算资源，哪怕对大多数单个任务而言，计算资源似乎早已过剩。可以看出，纯粹的actor模式在这个时候的更匹配需求，这源于它更接近真正的事件驱动。但很多实现中都忽略了这一优势，把事件驱动隐藏起来，甚至惧怕它的带来的灵活性，比如把解耦看作是导致事件驱动复杂化的因素，或者觉得有了映射和模式匹配就非常满意了。大多数人仍然沉浸在传统直接式编程风格带来的便利性中，把阻塞调用，线程视为是与生俱来的，而没有意识到这只不过是聪明的前人给傻瓜提供的便利，尽管它们也是有价值的，但是也遗留了下了包袱。巨大的惯性，让人忽视了语言的不足，感觉不到语言应该：有更好的表达事件驱动的方式；具备随心所欲的表达已经在大脑里构思好了的事件发生和对应处理流程上的关联的能力；而不仅仅是一股脑的开上成千上万的线程，要知道开发提供这种依赖的系统不可能依赖于还无法运行的系统自身。

## Reactor模型

**Reactor模型要求主线程（即I/O处理单元）只负责文件描述符上是否有事件发生**，如果有事件发生，则立即通知工作线程（即逻辑单元），除此之外，主线程不做其他实施性的工作，读写数据，接收新的连接以及处理客户请求的工作都在线程中完成。

**Nginx就是基于Reactor模型**。

### 组件

**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。

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

### 工作流程

Reactor模型的工作流程：

1. 主线程注册就绪事件（epoll往内核事件表中注册）
2. 主线程（epoll\_wait）等待连接上的读就绪事件
3. socket有可读事件，（epoll\_wait通知主线程），主线程将socket可读事件放入请求队列
4. 工作线程处理请求队列中的事件
5. 主线程等待连接可写
6. 可写时，主线程将可写事件放入请求队列
7. 工作线程往socket上写入服务器处理客户请求

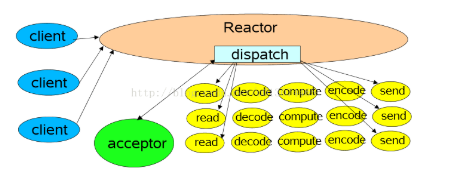


### 分类

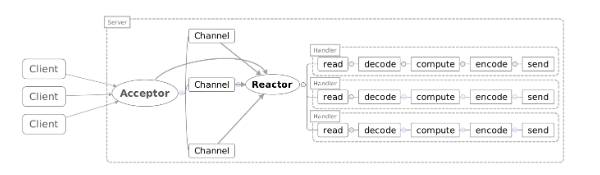
Reactor从线程池和Reactor的选择上可以细分为如下几种：

#### Reactor单线程模型

这是最简单的单Reactor单线程模型。Reactor线程是个多面手，负责多路分离套接字，Accept新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过这种单线程模型不能充分利用多核资源，所以实际使用的不多。



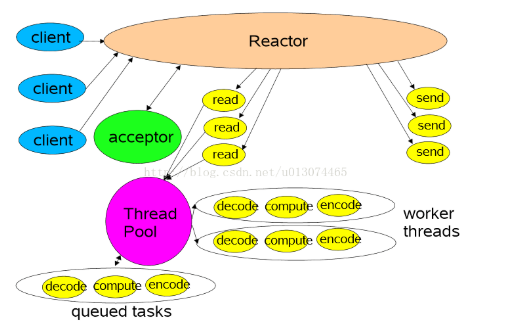
这个模型和上面的NIO流程很类似，只是将消息相关处理独立到了Handler中去了！

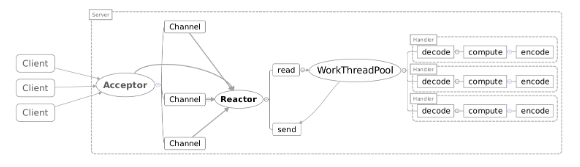


虽然上面说到NIO一个线程就可以支持所有的IO处理。但是瓶颈也是显而易见的！我们看一个客户端的情况，如果这个客户端多次进行请求，如果在Handler中的处理速度较慢，那么后续的客户端请求都会被积压，导致响应变慢！所以引入了Reactor多线程模型!

#### Reactor多线程模型

该模型在事件处理器（Handler）链部分采用了多线程（线程池），也是后端程序常用的模型。



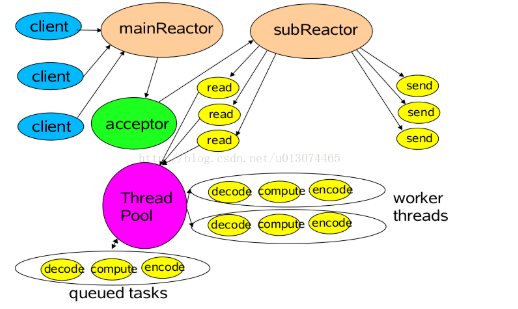


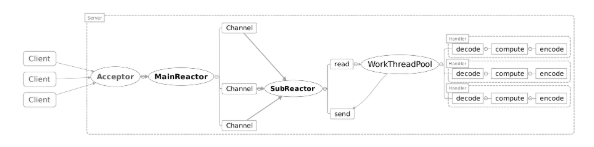
Reactor多线程模型就是将Handler中的IO操作和非IO操作分开，操作IO的线程称为IO线程，非IO操作的线程称为工作线程!这样的话，客户端的请求会直接被丢到线程池中，客户端发送请求就不会堵塞！

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

比起第二种模型，多线程模式（多个Reactor），它是将Reactor分成两部分，mainReactor负责监听并accept新连接，然后将建立的socket通过多路复用器（Acceptor）分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据；业务处理功能，其交给worker线程池完成。通常，subReactor个数上可与CPU个数等同。





主Reactor用于响应连接请求，从Reactor用于处理IO操作请求！

### Netty

## Proactor模型

Proactor是和异步I/O相关的。

在Reactor模式中，事件分离者等待某个事件或者可应用或个操作的状态发生（比如文件描述符可读写，或者是socket可读写），事件分离器就把这个事件传给事先注册的处理器（事件处理函数或者回调函数），由后者来做实际的读写操作。

在Proactor模式中，事件处理者(或者代由事件分离者发起)直接发起一个异步读写操作(相当于请求)，而实际的工作是由操作系统来完成的。发起时，需要提供的参数包括用于存放读到数据的缓存区，读的数据大小，或者用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。事件分离者得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。

可以看出两者的区别：Reactor是在事件发生时就通知事先注册的事件（读写由处理函数完成）；Proactor是在事件发生时进行异步I/O（读写由OS完成），待IO完成事件分离器才调度处理器来处理。

Proactor模型的工作流程：

1. 主线程注册socket上的读完成事件
2. 主线程处理其他逻辑
3. 读事件完成，发送信号
4. 应用程序调用信号处理函数
5. 主线程继续其他逻辑
6. 写入数据时，发送信号
7. 应用程序使用信号处理函数来善后



同步I/O模拟Proactor模型的原理：

1. 主线程执行数据的读写，同时负责通知
2. 工作线程获取结果，对逻辑的结果进行处理



Reactor模型和Proactor模型的共性：

1. 在连接或者监听socket上调用epoll\_wait
2. 工作线程处理连接上的事件
3. 主线程和工作线程的沟通通过工作队列

## 区别

以读操作为例（类操作类似）。

在Reactor（同步）中实现读：

- 注册读就绪事件和相应的事件处理器

- 事件分离器等待事件

- 事件到来，激活分离器，分离器调用事件对应的处理器。

- 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

Proactor（异步）中的读：

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。

- 事件分离器等待操作完成事件

- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。

- 事件分离器呼唤处理器。

- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

## 高效的并发模型

并发模型的特点：

1. 适用于I/O密集型
2. 有多线程和多进程两种
3. I/O处理单元和多个逻辑单元协调完成任务



# 提高服务器性能方法

服务器性能四大杀手：

1. 数据拷贝：可以使用缓存
2. 环境/上下文切换：该不该使用多线程，单线程好还是多线程好，单核服务器（采用状态机编程，效率最佳），多线程能够充分返回多核服务器的性能
3. 内存分配：增加内存池，减少向操作系统申请内存
4. 锁竞争

## 创建池

池：硬件资源充足，以硬件资源换取效率

池可以提高性能的原因：

1. 静态资源无需重新分配
2. 分配资源的系统调用很耗时
3. 避免服务器对内核的频繁访问

### 分类

池根据不同的资源类型，可以分为多种：

1. 内存池
2. 进程池
3. 线程池
4. 连接池

### 内存池

内存池的作用：

1. 存放大块数据
2. 存放数据缓存

内存池创建的方法：

1. 对于用户申请的大块内存使用内存映射
2. 对于小块内存从内存池合适的链表中取出

### 进程/线程池

进程池和线程池的作用：

1. 避免动态启动的时间开销
2. 使得处理更加单一
3. 充分利用硬件资源

进程池和线程池的注意事项：

1. 典型的生产者消费者问题
2. 注意访问共享资源存在的竞争

### 连接池

连接池的作用：

1. 为创建新连接提速
2. 可用于集群内部永久性连接

连接池创建的方法：

1. 预先分配固定数据的连接
2. 对每一个连接都分配相应的资源

## 减少数据复制

减少数据复制的原因：

1. 磁盘I/O操作非常耗时
2. 用户和内核之间的数据耗费系统资源

减少数据复制的方法：

1. 在合适的地方使用“零拷贝”函数
2. 使用共享内存传递信息

## 减少上下文切换和锁

减少上下文切换和锁的原因：

1. 任务的切换存在很大的系统开销
2. 锁使得并发程序编程串行执行

减少上下文切换和锁的方法：

1. 开启的线程不要多于CPU的个数
2. 减少并发程序的公共资源

# 服务器调试技术

## 调整系统参数

修改最大文件描述符：

ulimit –n

ulimit –SHn max-file-number

在/etc/security/limits.conf文件中永久性修改：

hard nofile max-file-number

soft nofile max-file-number

修改系统级文件描述符：

sysctl –w fs.file-max = max-file-number

永久性修改系统级文件描述符：

fs.file-max = max-file-number

## GDB调试器

步骤：gdb🡪attach pid🡪set follow-fork-mode mode

## 系统检测工具