# 服务器设计框架

## 设计目标

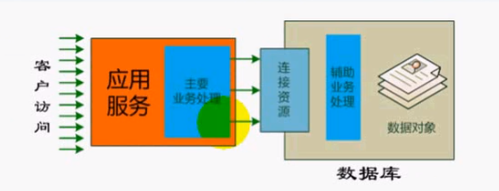
高并发服务器设计目标：

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

## 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
2. 多线程并发服务器
3. 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。

### 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





# 事件处理模型

服务器一般处理三类事件：I/O，信号，定时器事件

## 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**。

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



## Proactor模型

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

## 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