# 背景

之前所述，select、poll、epoll等IO多路复用都是客户端连接服务器技术，**线程池位于客户端**，即客户端已经完成与服务端的连接，可以使用高效地方法接收数据，接下来就是如何高效地处理数据。

# 作用

**线程池作用：**

1. 解决任务处理
2. 阻塞IO
3. 解决线程创建与销毁的成本问题
4. 管理线程
5. 异步解耦（写日志和落盘不同时）

在开发过程中，合理地利用线程池有以下好处：

1. 降低资源消耗，通过重复利用已创建的线程降低线程创建和销毁造成的消耗；
2. 提高响应速度（**异步解耦**），当任务到达时，任务可以不需要等到线程创建就能立即执行；

注：所谓的异步解耦，就是将任务加入到任务队列不需要立即执行，与具体执行分开。

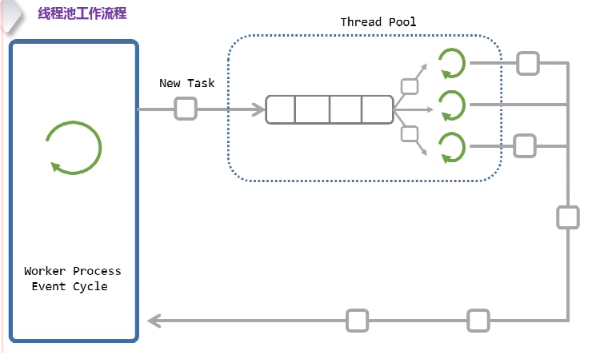
1. 提高线程的可管理性，线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

# 原理

提前创建好一堆线程，**客户端连接的时候直接使用，不需要频繁地创建和销毁线程**（创建和销毁线程是很耗时的），提高响应客户端的时间。

客户端发过来的数据对应一个事件，在服务端维护着事件的任务队列（这里使用条件变量判断是否需要启动线程相应事件，即任务队列不为空时线程池调用pthread\_sinal或pthread\_brodcast取任务，否则调用pthread\_cond\_wait阻塞），客户端发过来的事件在服务端会启动相应的线程处理。

线程池主要用于异步解耦，线程池如何做到异步解耦？



注：左侧对应的是工作线程，工作线程将新的任务加入到线程池的任务队列中，右侧线程池包括任务队列和实际的各个线程，各个线程负责任务的执行。所谓的异步解耦，实际上就是任务加入到任务队列和任务的执行是异步解耦的。

## 实现一

线程池如何实现？

1. 任务队列：多个线程共享的临界资源
2. 执行队列：多个执行线程的集合
3. 管理组件：用于管理任务队列和执行队列有秩序的工作

### 线程池结构体

代码：

#### 任务队列

1. 定义任务队列

// 任务队列

struct NJOB{

void (\*func)(void \*arg); //回调函数（每个任务执行需要的参数）

void \*user\_data; //用户信息

struct NJOB \*prev;

struct NJOB \*next;

};

注：一般创建一个独立的监听线程去监听客户端的请求，然后将任务加入到任务队列中。比如采用epoll监听socket，如果监听的事件发生了，则将任务队列中的任务删除并添加到执行队列中（执行队列有专门的独立线程执行相关操作）。

#### 工作队列

1. 定义执行队列

// 工作队列

struct NWORKER{

pthread\_t id;

struct NWORKER \*prev;

struct NWORKER \*next;

};

#### 管理组件

1. 定义管理组件

struct NMANAGER{

pthread\_mutex\_t mtx; //每次从任务队列取数据需要加锁

pthread\_cond\_t cond;

//定义线程挂起状态(如果任务队列不存在任务的时候，则线程挂起等待)

struct NJOB \*jobs;

struct NWORKER \*workers;

}nThreadPool;

注：因为执行队列需要从任务队列取数据，所以任务队列是多个线程的临界资源，所以需要引入锁对邻临界资源保护。

条件变量可以理解为任务队列不为空的时候触发执行队列获取任务，然后执行具体操作。

### 创建线程池

// 创建线程池(就是对结构体nThreadPool里面的变量初始化)

int nThreadPoolCreate(nThreadPool pool, int numWorkers){

//判断大小

if (numWorkers < 1) numWorkers = 1;

if (pool == NULL) return -1;

memset(pool,0,sizeof(nThreadPool));

pthread\_mutex\_t blank\_mutex = PTHREAD\_MUTEX\_INITILIZER;

memcpy(&pool->mtx, &blank\_mutex, sizeof(pthread\_mutex\_t));

pthread\_cond\_t blank\_cond = PTHREAD\_COND\_INITIALIZER;

memcpy(&pool->cond, &blank\_cond, sizeof(pthread\_cond\_t));//初始化

//workers

int i = 0;

for (i=0; i<numWorkers; i++){

struct NWORKER \*worker = (struct NWORKER\*)malloc(sizeof(struct NWORKER));

if(NULL==worker)

{

perror("malloc error");

return -2;

}

memset(worker,0,sizeof(struct NWORKER));

worker->pool = pool;

pthread\_create(&worker->id,NULL,thread\_cb,worker);//多个工作线程

LL\_ADD(worker,pool->workers);//将该工作线程加入到线程池

}

return 0;

}

### 销毁线程池

销毁线程有两个方法：

1. 不断循环，销毁创建的线程
2. 在工作线程中设置超时时间

### 任务队列添加任务

// 队列中添加任务

int nThreadPoolPushTask(nThreadPool \*pool, struct NJOB \*job){

pthread\_mutex\_lock(pool->mtx);

LL\_ADD(job,pool->jobs);

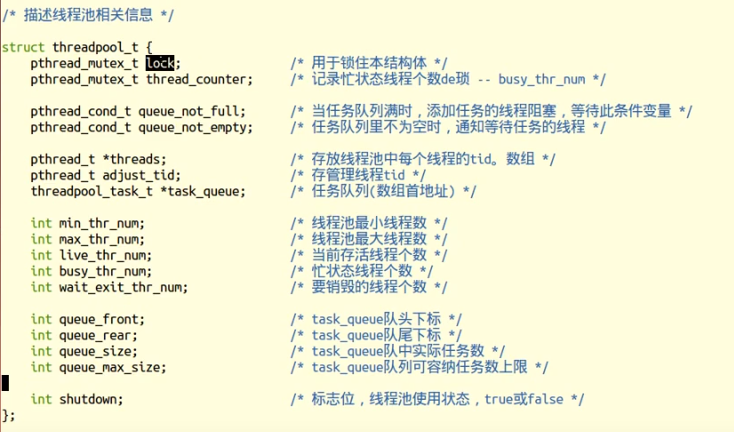
pthread\_cond\_signal(pool->cond);

pthread\_mutex\_unlock(pool->mtx);

}

## 实现二

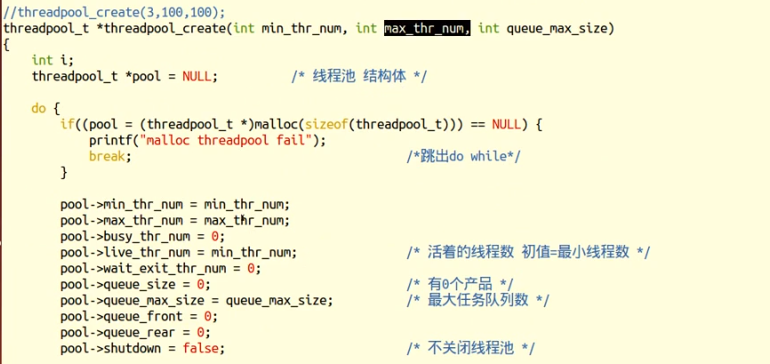
### 线程池结构体



### 创建线程池

基本操作：

1. 创建线程池结构体的指针
2. 初始化线程池结构体（N个成员变量）
3. 创建N个任务线程
4. 创建1个管理者线程
5. 失败时，销毁开辟的所有空间







### 管理线程

### 向线程池中添加任务

# Java线程池

# 线程池与消息队列

1、两者内部都使用了队列，如阻塞队列、优先级队列；

2、使用线程池时应用服务器既充当生产者又充当消费者，也是消息队列中间件的实现者，使用消息队列时中间件、生产者、消费者可以部署在不同的应用机器上（当然也可以部署在一台服务器上但很少有人这么用）；

3、出于第2点线程池更适合非分布式的系统，但在分布式架构下消息队列明显是更优项；

4、使用消息队列会带来额外的网络开销；

5、消息队列的耦合性更低，可扩展性更好,适用于弱一致性的场景，如对log日志的解耦;

6、消息队列自动实现消息的持久化，中间已经实现了大量功能，如消息转发、消息拒绝、消息重试，以及对消息的一些监控，例如消息的消费状态、消息的消费速率等，使用线程池如果需要很多功能还要自己去实现，例如需要执行状态需要打印队列数量、计算消息消费速度；

7、在不同系统间的服务调用（调用协议也可能不一致）线程池很难实现或开销很大，这时候消息队列可以屏蔽不同机器或不同协议的问题；

8、使用消息队列会提升系统的复杂度，网络抖动怎么办？最大队列长度怎么设置？超时时间又设置多少？Qos又设置为多少？消费者多少个比较合适？Channel cache size又该设置为多少？业务线可能都是用同一个Mq，你占资源太多，或者设计不当可能会导致整个Mq故障。

# 应用

Nginx：日志落盘，采用线程池，主要用于异步解耦。