# 基于协程的高并发回射服务器

## 切换上下文用了什么系统调用？

#include <stdio.h>

#include <ucontext.h>

#include <unistd.h>

int main(int argc, const char \*argv[]){

ucontext\_t context;

getcontext(&context); // 将当前上下文保存到context中

puts("Hello world");

sleep(1);

setcontext(&context); // 切换到context上下文处

return 0;

}

#include <ucontext.h>

#include <stdio.h>

void func1(void \* arg)

{

puts("1");

puts("11");

puts("111");

puts("1111");

}

void context\_test()

{

char stack[1024\*128];

ucontext\_t child, main;

getcontext(&child); //获取当前上下文

child.uc\_stack.ss\_sp = stack;//指定栈空间

child.uc\_stack.ss\_size = sizeof(stack);//指定栈空间大小

child.uc\_stack.ss\_flags = 0;

child.uc\_link = &main;//设置后继上下文

makecontext(&child, (void (\*)(void))func1, 0);

//修改上下文指向func1函数

swapcontext(&main, &child);

//切换到child上下文，保存当前上下文到main

puts("main");

//如果设置了后继上下文，func1函数指向完后会返回此处

}

int main()

{

context\_test();

return 0;

}

* getcontext(&ctx)保存上下文到ctx。
* setcontext(&ctx)切换上下文。
* makecontext(&ctx, (void(\*)(void))func)将ctx指向func函数。
* swapcontext(&ctx1, &ctx2)切换到ctx2并将当前上下文保存到ctx1。

## 具体说说协程上下文切换的过程

在创建协程时，将当前的上下文通过getcontext和makecontext与协程的运行函数绑定。当协程遇到某些事件时会指行Yield函数让出CPU，也就是使用swapcontext进行上下文切换（切换到调度器的上下文并将当前上下文保存到当前协程的上下文中）。此时由调度器从就绪队列中选择一个新的协程，也就是使用swapcontext进行上下文切换（切换到新的协程的上下文并将当前上下文保存到调度器的上下文中）。

## 什么时候进行上下文切换？

在程序中有一个主协程，就是一个服务器对象负责监听客户端的连接，同时将服务器和客户端的套接字设置为非阻塞。主协程中当服务器没有监听到客户端连接时会调用yield函数让出CPU。每当监听一个新的客户端连接，就创建一个新的协程负责读取客户端发来的数据并发送数据给客户端，当服务器读完客户端的数据（边沿触发）时会调用yield函数让出CPU，当服务器写完数据也会调用yield函数。

## 有栈协程和无栈协程

这里的栈是调用栈，有栈协程和无栈协程的区别在于有栈协程可以在任意嵌套函数中被挂起，但无栈协程不行。在这个项目中使用的是无栈协程，因为每个协程都有独立的功能，主协程负责服务器对客户端的监听，其他协程负责服务器对客户端数据的读写，因此协程没有嵌套调用。

# 基于B+树的小型SQL数据库

## 是如何实现从磁盘读写数据的？

数据库的每一条记录都有一个偏移量，这个偏移量是记录保存在磁盘文件位置的偏移量，先打开磁盘文件，然后通过fseek找到要读写记录的位置，再通过fread和fwrite进行读写。

## 为什么用B+树不用红黑树和B树？

## 是如何设计B+树头部、内部结点和叶子结点的数据结构的？

* 头部存有B+树的阶，key和value的大小，内部结点和叶子结点数量，树的高度，root指针，最左端叶子结点指针。
* 内部结点存有指向父结点，左兄弟，右兄弟的指针和一个长度为B+树阶的内部结点指针数组，父结点用于插入和删除操作中查找父结点，兄弟结点用于删除操作中向兄弟借记录和合并。
* 叶子结点存有指向父结点，左兄弟，右兄弟的指针和一个长度为B+树阶的记录数组，父结点用于插入和删除操作中查找父结点，兄弟结点用于删除操作中向兄弟借记录和合并。

## 是如何实现B+树的查找的？

从根节点开始，在每一个内部结点从左往右寻找，找到第一个大于等于key的下标，然后依次向下查找，直到找到叶子节点为止，记录保存在叶子结点上。

## 是如何实现B+树的插入的？

首先找到待插入记录应在的叶子结点，如果该结点记录数量小于B+树的阶则直接插入即可，否则表示叶子结点记录数量已满，则新建一个叶子结点并将原叶子结点的一半移动到新的叶子结点中。此时还递归需检查父结点的key数量有没有超过B+树的阶，如果超过则需要继续分裂。

## 是如何实现B+树的删除的？

首先找到待删除记录所在的叶子结点，如果删该结点记录数量大于B+树阶的一半则直接删除即可，否则表示删除后叶子结点记录数量不够，此时先向左兄弟借一个记录，没借到就向右兄弟，再没借到就只能和兄弟合并，如果该结点是父亲结点的最后一个孩子就和左兄弟合并，否则与右兄弟合并。此时还递归需检查父结点的key数量有没有小于B+树的阶的一半，如果小于则需要继续向左右兄弟借key或者合并。

## 可以实现表字段的插入吗？

不能，为了简化磁盘存储，我将记录的大小固定，字段也是固定的，不能修改。

## 是如何解析用户输入的命令的？

首先通过字符串的第一个单词判断是插入、删除、查找、更新还是非法操作，然后通过sscanf来读取命令中的id或是id的范围再进行相应的操作。

# 基于HTTP协议的在线数独求解器

## 是如何设计解数独的方案的？

采用的是改进的dfs搜索算法，原始的dfs算法是从数独格子的左上角开始，对每个格子可能填入的数字进行dfs搜索，直到出现数字冲突或者填满所有格子。但这样dfs搜索树的分枝太多，解一些复杂的数独需要很长时间，因此考虑从最少的分枝入手，也就是每次都选可填入数字最少的格子进行搜索，这样能有效的减少搜索树的分枝，提高搜索效率。

## 如何解析http请求头？

请求头包括方法，url和http版本。方法常见的有GET和POST，本项目中使用的是GET，GET方法主要通过URL来传输数据，使用GET方法后的URL格式为?key1=value1&key2=value2。

## http服务器响应头包括哪些内容？

响应头包括HTTP版本，状态号和响应字符串，比如200 OK和404 NOT FOUND。在传输文件前需要发送Content-Type传输文件的类型，长度是可选的。

## GET和POST过程分别是怎么样的？

### post请求的过程

1. 浏览器请求tcp连接（第一次握手）
2. 服务器答应进行tcp连接（第二次握手）
3. 浏览器确认，并发送post请求头（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）
4. 服务器返回100 Continue响应
5. 浏览器发送数据
6. 服务器返回200 OK响应

### get请求的过程

1. 浏览器请求tcp连接（第一次握手）
2. 服务器答应进行tcp连接（第二次握手）
3. 浏览器确认，并发送get请求头和数据（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）
4. 服务器返回200 OK响应

## GET和POST的区别

* post更安全（不会作为url的一部分，不会被缓存、保存在服务器日志、以及浏览器浏览记录中）。
* post发送的数据更大（get有url长度限制）。
* post能发送更多的数据类型（get只能发送ASCII字符）。
* post比get慢。
* post用于修改和写入数据，get一般用于搜索排序和筛选之类的操作（淘宝，支付宝的搜索查询都是get提交），目的是资源的获取，读取数据。

## 基础网络通信API

int lfd = socket(AF\_INET, SOCK\_STREAM, 0);

bind(lfd, (struct sockaddr\*)&addr, sizeof(addr));

listen(lfd, 128);

accept(lfd, (struct sockaddr\*)&cliaddr, &len);

write(cfd, buf, n);

read(cfd, buf, sizeof(buf));

close(lfd);

## libevent API

event\_base\_new创建根结点

evconnlistener\_new\_bind绑定IP、端口和监听回调函数，返回监听器

event\_base\_dispatch循环监听

event\_base\_free释放根节点

bufferevent\_socket\_new创建bufferevent根节点

bufferevent\_setcb设置回调

bufferevent\_enable(bev, EV\_READ | EV\_WRITE)启用读写

bufferevent\_write写

bufferevent\_read读

读写一般在回调函数中调用

## epoll API

### 创建红黑树

int epoll\_create(int size);

参数:

size：监听的文件描述符的上限，2.6版本之后写1即可

返回：返回树的句柄

### 上树 下树 修改节点

nt epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

参数:

epfd：树的句柄

op：EPOLL\_CTL\_ADD上树 EPOLL\_CTL\_DEL下树 EPOLL\_CTL\_MOD修改

fd：上树，下树的文件描述符

event：上树的节点

typedef union epoll\_data {

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

struct epoll\_event {

/\* Epoll events \*/ 需要监听的事件

uint32\_t events;

/\* User data variable \*/ 需要监听的文件描述符

epoll\_data\_t data;

};

将cfd上树

int epfd = epoll\_create(1);

struct epoll\_event ev;

ev. data.fd = cfd;

ev.events = EPOLLIN;

epoll\_ctl(epfd, EPOLL\_CTL\_ADD,cfd, &ev);

### 监听

int epoll\_wait(int epfd, struct epoll\_event \*events,

int maxevents, int timeout);

功能：监听树上文件描述符的变化

epfd：数的句柄

events：接收变化的节点的数组的首地址

maxevents：数组元素的个数

timeout：-1 永久监听 大于等于0 限时等待

返回值: 返回的是变化的文件描述符个数

### epoll工作方式

#### 水平触发 LT

* 读缓冲区有数据就会触发epoll\_wait
* 写缓冲区可以写就会触发epoll\_wait

#### 边沿触发 ET

* 读缓冲区数据发生变化会触发epoll\_wait，也就是数据来一次只触发一次
* 写缓冲区从有到无就会触发epoll\_wait