相比于0.3版本0.4版本下做了以下修改：

问题：

1. 客户端一直刷新请求资源 一段时间过后 服务器出现段错误
2. 在目录请求未完成时 继续请求则出现段错误
3. 一直都存在的问题是客户端存在一直在请求无法析构请求的情况

为不提供拷贝构造和赋值运算符的类添加了noncopyable基类

重写了RALL机制的锁

重写了单例模式，将双重加锁改为更简单而安全的pthread\_once()

1、关于枚举类型的变量的值的问题

/\*枚举类型中的值 当所有值都未赋值时第一次元素的值为0 后面的值依次加1

如果有赋值的话则 后面的变量则从该赋值位置开始逐渐加1

\*/

enum HeaderState

{

h\_start = 0,

h\_key,

h\_colon,

h\_spaces\_after\_colon,

h\_value,

h\_CR,

h\_LF,

h\_end\_CR,

h\_end\_LF

};

h\_key = 1; h\_colon =2; h\_spaces\_after\_colon =3;……..

2、关于关键字explicit的使用

//避免在对象初始化时隐式调用该构造函数 只能通过显示调用 即不支持隐式类型转换

/\*在调用该构造函数时 如果不是类似A a(19)就会报错 如果使用 a = 19就相当于使用隐式类型转换\*/

explicit MutexLockGuard(MutexLock &\_mutex);

3、关于使用非拷贝 非赋值基类的创建和使用方式

//创建非拷贝构造 非赋值基类 其它类继承该类即具有该类功能 参考陈硕的linux多线程服务器端编程

#pragma once

class nocopyable

{

protected:

nocopyable() {};

~nocopyable() {};

private:

nocopyable(const nocopyable&);

const nocopyable& operator=(const nocopyable&);

};

4、重新定义RALL锁类

/\*.hpp 文件表示 .cpp和.h文件都放在同一个文件中 表示为将定义和实现都放在同一个文件 这样编译时就可以不用再进行编译 直接生成.obj

文件进行调用即可\*/

#pragma once

#include "nocopyable.hpp"

#include <pthread.h>

#include <cstdio>

//nocopyable为非拷贝 非赋值类 因此继承该类可以拥有这种属性

class MutexLock : noncopyable

{

public:

MutexLock()

{

pthread\_mutex\_init(&mutex, *NULL*);

}

~MutexLock()

{

//在锁析构的时候进行加锁 是防止此时锁被使用

pthread\_mutex\_lock(&mutex);

pthread\_mutex\_destroy(&mutex);

}

void lock()

{

pthread\_mutex\_lock(&mutex);

}

void unlock()

{

pthread\_mutex\_unlock(&mutex);

}

private:

pthread\_mutex\_t mutex;

};

/\*详细来看 RALL锁机制 其实可以理解为我们在调用函数时 其实是入栈 在函数中调用锁对象 锁对象也是在栈中

因此在函数调用结束时 牵涉到出栈 出栈就会释放资源 因此我们不用担心另外的线程来使用资源

使用RALL方法能够成功的将对象和资源进行绑定

\*/

/\*RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，

使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。\*///管理资源

class MutexLockGuard : noncopyable

{

public:

//避免在对象初始化时隐式调用该构造函数 只能通过显示调用 即不支持隐式类型转换

/\*在调用该构造函数时 如果不是类似A a(19)就会报错 如果使用 a = 19就相当于使用隐式类型转换\*/

explicit MutexLockGuard(MutexLock &\_mutex):mutex(\_mutex)

{

mutex.*lock*();

}

~MutexLockGuard()

{

mutex.unlock();

}

private:

MutexLock &mutex;

};

5、对weak\_ptr类型的timer指针类型的lock()深度剖析

/\*对该段代码的分析是由于我们在执行程序的过程中 出现段错误，导致程序崩溃\*/

//分离定时器 -- 如果定时器 不为空 我们对定时器进行清空操作 设置其参数 deleted 为true

void RequestData::seperateTimer()

{

if (timer.*lock*())

{

*shared\_ptr*<TimerNode> my\_timer(timer.*lock*());

my\_timer->clearReq();

timer.*reset*();

}

}

timer.lock()的作用是weak\_ptr.lock() 会把weak\_ptr提升为一个 shared\_ptr 对象，当 引用计数为0 ，那么 shared\_ptr 为空。

timer.reset()的作用是将定时器清空

6、将双重加锁更改为更加安全的一次加锁的pthread\_once;

pthread\_once\_t MimeType::once\_control = PTHREAD\_ONCE\_INIT;

应用于函数getMime中：

*std*::*string* MimeType::getMime(const *std*::*string* &suffix)

{

//将变量和函数同时传入 判断是否该函数中有线程在执行

pthread\_once(&once\_control, MimeType::init);

//判断是否找到该类型

if (mime.*find*(suffix) == mime.*end*())

{

return mime["default"];

}

else

{

return mime[suffix];

}

}

Gdb使用的大前提 是程序是自己写的

遇到段错误，程序代码逻辑没有问题，实在无法解决当前问题，学习了解GDB调试程序：

-g：使用该参数编译可执行文件，得到调试表

gdb+可执行程序

list: list 1列出源码 根据源码指定 行号设置断点

b：b 20 在20行设置断点

run/r 运行程序

n/next 下一条指令(会越过函数)

s/step 下一条指令(会进入函数)

p/print p i 查看变量的值

continue 继续执行断点后续指令

quit 退出gdb当前调试

1. 首先在编译时在末尾加入 –g 可以加入调试信息
2. gdb + 可执行文件 进入调试
3. list展示源码信息 list 1从第一行开始显示 继续l/list
4. 设置断点 break + 行号 (b 52)
5. run(r)开始执行 程序停止在设置的断点处
6. n/s next/step(单步) s可以进入到函数内部 如果函数时系统函数则使用n跳过
7. p + 变量 可以打印变量的值

其他指令：

run：使用run找到出段错误的地方

finish: 结束当前函数的调用

set args: 设置mian函数参数

run 字串1 字串2：设置main命令行参数

info b: 查看断点信息表

b 20 if i = 5设置条件断点

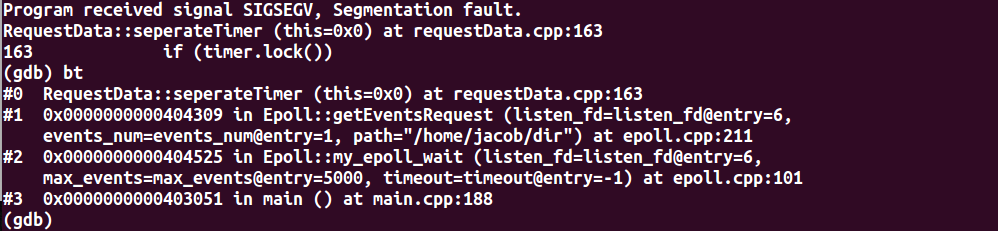
ptype查看变量类型

bt 列出当前程序中正存活着的栈帧

frame 根据栈帧编号，切换栈帧

display 设置跟踪变量

undisplay 取消设置跟踪变量 使用跟踪变量的编号



如上图所示，使用gdb调试工具找到出错误的位置，但是具体怎么调试暂时还不知道，正在进一步的查资料是timer.lock()的问题还是在调用seperateTime时未加锁，导致在执行函数seperateTimer时显示this=0x0；这个问题所导致的直接问题就是出现无法解决的段错误！