【代码随想录知识星球】项目精讲-协程库（CPP）

代码随想录知识星球在23年5月份发布了项目精讲 webserver（C++） 精讲

不少录友反馈，webserver 现在有点 “烂大街”。

坦白来说 webserver 这个项目对于小白，还是相对友好的，而且webserver可以集成很多服务。

这次再带大家做一个 CPP的项目，实现 协程库，相对于带大家造轮子。

项目文档依然非常齐全，从 前置知识 到 理论基础，从 动手实现 到 项目拓展 再到最后 简历如何写，面试会问的问题，都给大家安排了。

- 前序
    - 为什么要做协程库?
    - 所需要的基础知识
    - 编程语言
    - 操作系统&Linux
    - 计算机网络
    - 参考书籍&开源项目&博客
- 动手前先了解一下协程
    - 协程基础知识
        - 什么是协程?
        - 对称协程与非对称协程
        - 有栈协程与无栈协程
        - 独立栈与共享栈
        - 协程的优缺点
    - C++有哪些协程库?
- 开始动手
    - 协程类的实现
    - 协程调度
    - 协程+IO
    - 定时器
    - hook
- 写好了就完了吗?
    - 项目扩展
    - 协程+
    - 性能测试
- 如何应对面试?
    - 简历怎么写?
    - 面试会问哪些问题呢?

前序

为什么要做协程库?

1、**不“烂大街”，主打一个差异化**

cpp选手简历上基本人手一个WebServer项目，这个项目可以串联很大部分的基础知识，包括 C/C++，计算机网络，操作系统基础知识。

还有Linux环境下的网络编程、系统编程，数据库等知识，且有很大的扩展空间，可以扩展更多功能模块，cpp选手用这个项目作为巩固基础知识的入门项目是非常合适的。

但是这个项目确实过于“烂大街”，如果不能做出新意，只是简单的照搬，或是简单做个功能扩展，拿这个项目作为简历的主打项目是毫无竞争力的。

2、**协程确实有用**

面试中，面试官经常会问这样的问题，“你知道线程和进程区别吗？”然后会紧接着追问“你了解协程吗?

协程和进程、线程又有什么区别?

”我们通过基础知识的学习和如WebServer此类项目，已经对进程和线程有了比较深的理解，但对协程相关知识却知之甚少。

协程作为一种强大的并发编程技术，可以在需要处理大量I/O操作或者并发任务的情况下提高程序的性能和可维护性。

在许多场景应用广泛，如果我们能做一个协程库的项目，不但可以让简历更加出彩，对以后的工作也大有帮助。

3、**协程库只是一个轮子，可以方便的应用在其他项目中**，增加其他项目的“新意”

自己手动完成一个协程库，还可以直接将我们自己编写的协程库用在其他项目里，就比如“烂大街”的WebServer，引入协程技术，不但可以提高并发和资源利用率，还大大简化了异步编程的复杂性，使代码更易于理解和维护，这样这个“烂大街”的项目也就有了新意。

4、**增加知识的深度和广度，提高面试通过率**

深入理解了协程技术后，即使面试官不主动问协程技术，就算问进程与线程，我们也可以主动提及协程，与线程和进程对比，引导面试官问协程相关的问题，主动展示自己知识的深度和广度，这会大大提高我们面试的通过率。

所需要的基础知识

整体来说，本项目对基础还是有一定要求的，需要有基本的linux网络编程和系统编程基础，技术栈上其实和Webserver是高度重合的，建议在做过webserver这样的入门项目后再做后会更容易上手一些。

编程语言

语言主要是c++，要求不高，掌握基本的 C/C++ 语法，**熟悉C++11 的常用特**性即可，像**智能指针、还有匿名函数、function对象**等都会用到，其他一些不常用到的特性可以在项目里学习。

操作系统&Linux

操作系统的所有基础知识都是要了解的，尤其是**进程管理部分，要深入了解进程和线程。**

本项目是在Linux环境下开发，**Linux的基本知识尤其是基础操作命令**是必须要会的。**Linux常见的系统调用**也要会用，要有基本的L**inux系统编程基础**。

计算机网络

计算机网络基础知识是必须的。本项目主要涉及还是Linux下的网络编程，常见**网络编程API、网络IO，IO多路复用、高性能网络模式：Reactor 和 Proactor**都是要了解的基础知识。

参考书籍&开源项目&博客

书籍

- 游双. Linux高性能服务器编程[M]. 机械工业出版社, 2013.
- 陈硕. Linux多线程服务端编程：使用muduo C++网络库[M]. 电子工业出版社, 2013.

主要就是Linux服务端编程的相关书籍，这两本是比较热门的，第一本相对基础，也都是Linux服务端编程的重点，建议从头开始精读；第二本相对进阶，可以有选择的阅读。

开源项目

- https://github.com/sylar-yin/sylar

一个c++服务器框架，包括日志、配置、协程、tcpserver、http等模块，我们要做的项目就是就是这个项目的协程模块。建议有时间全都做一下，会对整个服务器开发有完整深入的了解。B站上有完整的教学视频 https://www.bilibili.com/video/av53602631/?from=www.sylar.top，视频有点枯燥，需要耐着性子慢慢看。

- https://github.com/zhongluqiang/sylar-from-scratch

对上面sylar项目的重写，部分地方有适当简化，有更详细的博客讲解，https://www.midlane.top/wiki/pages/viewpage.action?pageId=10060952，写的非常详细

- https://github.com/Tencent/libco

腾讯开源的c++协程库，是微信后台大规模使用的c/c++协程库，2013年至今稳定运行在微信后台的数万台机器上，代码量不多，难度不大，非常适合学习。

博客

- http://www.sylar.top/blog/

sylar项目的作者博客，有项目各个部分的介绍

- https://www.midlane.top/wiki/pages/viewpage.action?pageId=10060952

sylar-from-scratch的介绍，从零开始重写sylar项目，介绍比sylar作者博客更加详细

- https://blog.csdn.net/affrewg/article/details/126449237?spm=1001.2014.3001.5501
- https://blog.csdn.net/affrewg/article/details/126804064?spm=1001.2014.3001.5501
- https://blog.csdn.net/affrewg/article/details/126804124?spm=1001.2014.3001.5501

以上三篇是我在学习libco协程库中做的记录，对libco协程库库关键功能做的源码剖析，在学习libco时可以参考。

动手前先了解一下协程

协程基础知识

什么是协程?

对协程概念的理解可以对比线程，通用的说法是协程是一种**“轻量级线程”，用户态线程**”。

最简单的理解，可以将协程当成一种看起来花里胡哨，并且使用起来也花里胡哨的函数。每个协程在创建时都会指定一个入口函数，这点可以类比线程。**协程的本质就是函数和函数运行状态的组合** 。协程和函数的不同之处是，函数一旦被调用，只能从头开始执行，直到函数执行结束退出，而协程则可以执行到一半就退出（称为yield），但此时协程并未真正结束，只是暂时让出CPU执行权，在后面适当的时机协程可以重新恢复运行（称为resume），在这段时间里其他的协程可以获得CPU并运行，所以协程被描述称为**“轻量级线程”。**

协程能够半路yield、再重新resume的关键是协程存储了函数在yield时间点的执行状态，这个状态称为**协程上下文**。协程上下文包含了函数在当前执行状态下的全部CPU寄存器的值，这些寄存器值记录了函数栈帧、代码的执行位置等信息，如果将这些寄存器的值重新设置给CPU，就相当于重新恢复了函数的运行。**单线程环境下，协程的yield和resume一定是同步进行的，一个协程的yield，必然对应另一个协程的resume**，因为线程不可能没有执行主体。并且，协程的yield和resume是完全由应用程序来控制的。与线程不同，线程创建之后，线程的运行和调度也是由操作系统自动完成的，但协程创建后，**协程的运行和调度都要由应用程序来完成**，就和调用函数一样，所以协程也被称为**“用户态线程”**。

更多协程的介绍可以参考：

https://www.bilibili.com/video/BV1b5411b7SD/

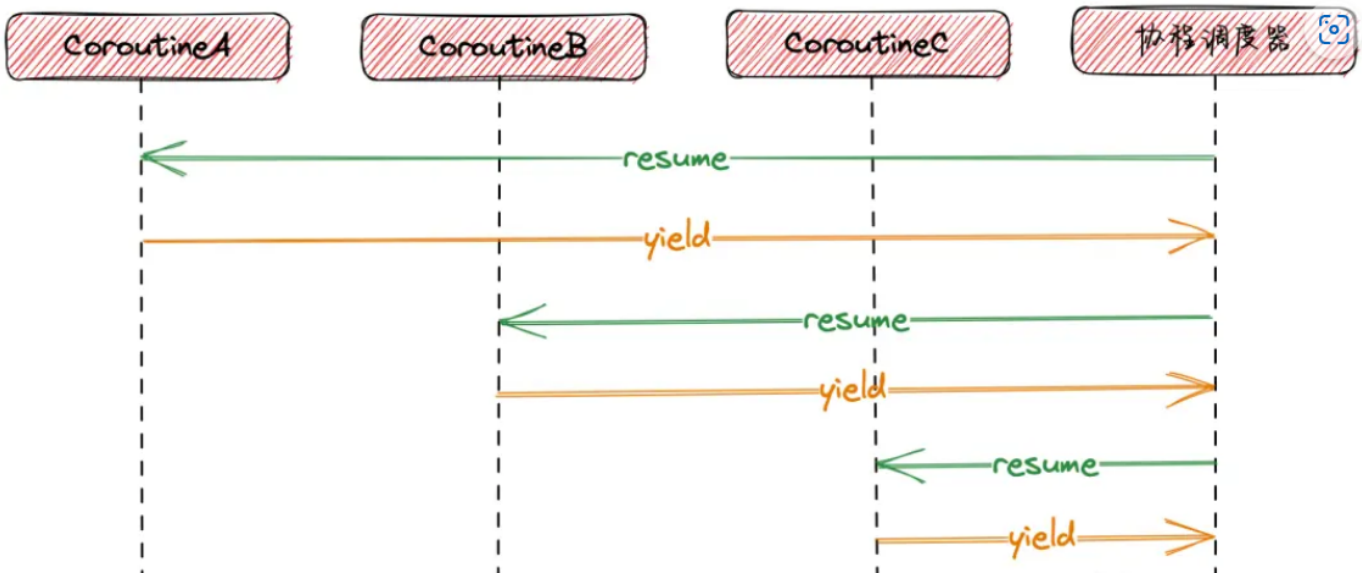
https://www.bilibili.com/video/BV1b5411b7SD/

https://jasonkayzk.github.io/2022/06/03/%E6%B5%85%E8%B0%88%E5%8D%8F%E7%A8%8B/

对称协程与非对称协程

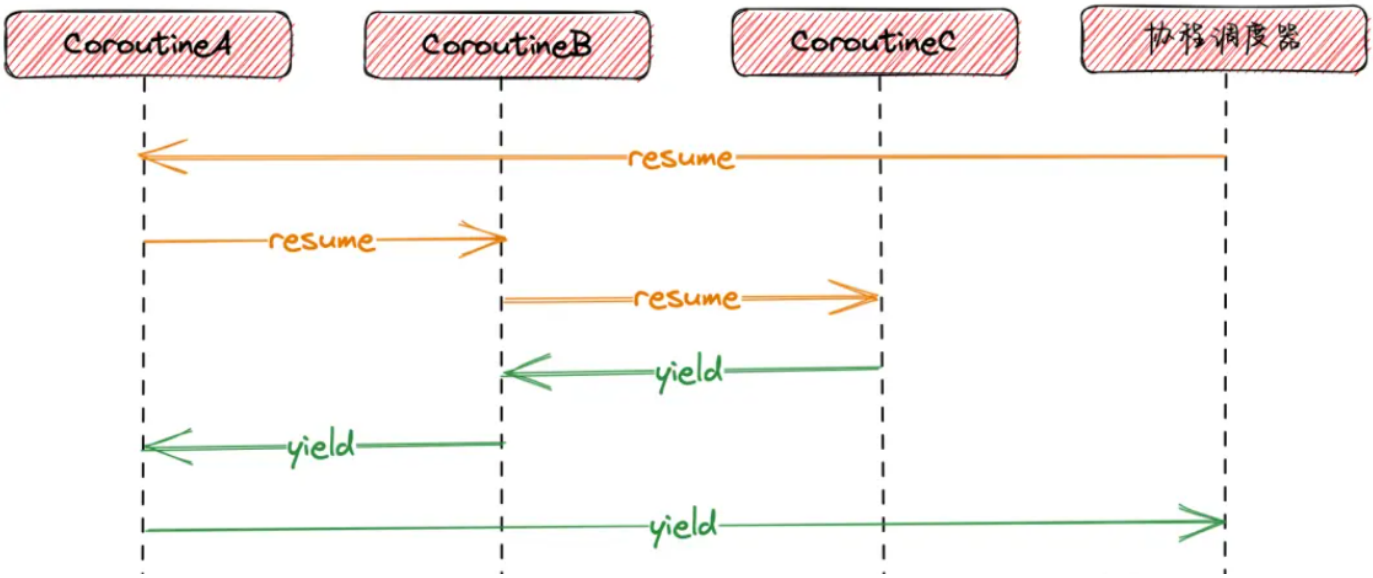
对称协程，协程可以不受限制地将控制权交给任何其他协程。任何一个协程都是相互独立且平等的，调度权可以在任意协程之间转移。

CoroutineA，CoroutineB，CoroutineC之间是可以通过协程调度器可以切换到任意协程。



非对称协程，是指协程之间存在类似堆栈的调用方-被调用方关系。协程出让调度权的目标只能是它的调用者。

CoroutineA，CoroutineB，CoroutineC之间比如与调用者成对出现，比如resume的调用者返回的位置，必须是被调用者yield



对称协程更灵活，非对称协程实现更简单。在对称协程中，子协程可以直接和子协程切换，也就是说每个协程不仅要运行自己的入口函数代码，还要负责选出下一个合适的协程进行切换，相当于每个协程都要充当调度器的角色，这样程序设计起来会比较麻烦，并且程序的控制流也会变得复杂和难以管理。而在非对称协程中，可以借助专门的调度器来负责调度协程，每个协程只需要运行自己的入口函数，然后结束时将运行权交回给调度器，由调度器来选出下一个要执行的协程即可。

有栈协程与无栈协程

有栈协程：用独立的执行栈来保存协程的上下文信息。当协程被挂起时，栈协程会保存当前执行状态（例如函数调用栈、局部变量等），并将控制权交还给调度器。当协程被恢复时，栈协程会将之前保存的执行状态恢复，从上次挂起的地方继续执行。类似于内核态线程的实现，不同协程间切换还是要切换对应的栈上下文，只是不用陷入内核而已。

无栈协程：它不需要独立的执行栈来保存协程的上下文信息，协程的上下文都放到公共内存中，当协程被挂起时，无栈协程会将协程的状态保存在堆上的数据结构中，并将控制权交还给调度器。当协程被恢复时，无栈协程会将之前保存的状态从堆中取出，并从上次挂起的地方继续执行。协程切换时，使用状态机来切换，就不用切换对应的上下文了，因为都在堆里的。比有栈协程都要轻量许多。

更多详细介绍可以参考：

https://mthli.xyz/stackful-stackless/

https://zhuanlan.zhihu.com/p/347445164

独立栈与共享栈

独立栈和共享栈都是有栈协程。

共享栈本质就是所有的协程在运行的时候都使用同一个栈空间，每次协程切换时要把自身用的共享栈空间拷贝。对协程调用 yield 的时候，该协程栈内容暂时保存起来，保存的时候需要用到多少内存就开辟多少，这样就减少了内存的浪费， resume 该协程的时候，协程之前保存的栈内容，会被重新拷贝到运行时栈中。

独立栈，也就是每个协程的栈空间都是独立的，固定大小。好处是协程切换的时候，内存不用拷贝来拷贝去。坏处则是 内存空间浪费。因为栈空间在运行时不能随时扩容，否则如果有指针操作执行了栈内存，扩容后将导致指针失效。为了防止栈内存不够，每个协程都要预先开一个足够的栈空间使用。当然很多协程在实际运行中也用不了这么大的空间，就必然造成内存的浪费和开辟大内存造成的性能损耗。

独立栈相对简单，但废内存，容易栈溢出。

共享栈使用公共资源，公共资源内存空间比较大，相对安全，节省内存空间，但是协程频繁切换需要进行内存拷贝，废 CPU。

协程的优缺点

经过对协程概念的介绍，想必都对协程的优点有了些许认识，这个问题见仁见智，这里根据我的理解总结两点：

- **提高资源利用率，提高程序并发性能。**协程允许开发者编写异步代码，实现非阻塞的并发操作，通过在适当的时候挂起和恢复协程，可以有效地管理多个任务的执行，提高程序的并发性能。与线程相比，协程是轻量级的，它们的创建和上下文切换开销较小，可以同时执行大量的协程，而不会导致系统负载过重，**可以在单线程下实现异步**，使程序不存在阻塞阶段，充分利用cpu资源。
- **简化异步编程逻辑。**使用协程可以简化并发编程的复杂性，通过使用适当的协程库或语言特性，可以避免显式的线程同步、锁和互斥量等并发编程的常见问题，**用同步的思想就可以编写成异步的程序。**

协程有什么缺点呢，细分析起来还是可以总结不少的，比如难以调试、占用更多内存，学习成本相对较高等，但是最明显的缺点是：

- **无法利用多核资源。**线程才是系统调度的基本单位，单线程下的多协程本质上还是串行执行的，只能用到单核计算资源，所以协程往往要与多线程、多进程一起使用。

C++有哪些协程库?

C++20引入了原生的协程支持作为C++标准库的一部分，可以用于编写异步代码。具体介绍可参考https://www.bennyhuo.com/2022/03/09/cpp-coroutines-01-intro/

一些有名的第三方协程库有：

Boost.Coroutine2：这是Boost库中的一个模块，提供了一组灵活的协程实现，包括对称协程和非对称协程。它提供了一些强大的特性，如支持多个栈和自定义栈大小。

libco：腾讯微信团队开源的一个C/C++协程库，据说微信后台大量在使用这个库，通过几个简单的接口就能实现协程的创建/调度，同时基于epoll/kqueue实现了一个事件反应堆，再加上sys_call（系统调用）hook技术，以此给开发者提供同步/异步的开发方法
libgo ： 是一个使用 C++ 编写的协作式调度的stackful有栈协程库, 同时也是一个强大的并行编程库。支持linux平台，MacOS和windows平台，在c++11以上的环境中都能用。

还有很多不同功能和适用场景的C++协程库可自行搜索。

开始动手

下面主要以sylar为例介绍一下一个完整的协程库具体应该怎么实现。

协程类的实现

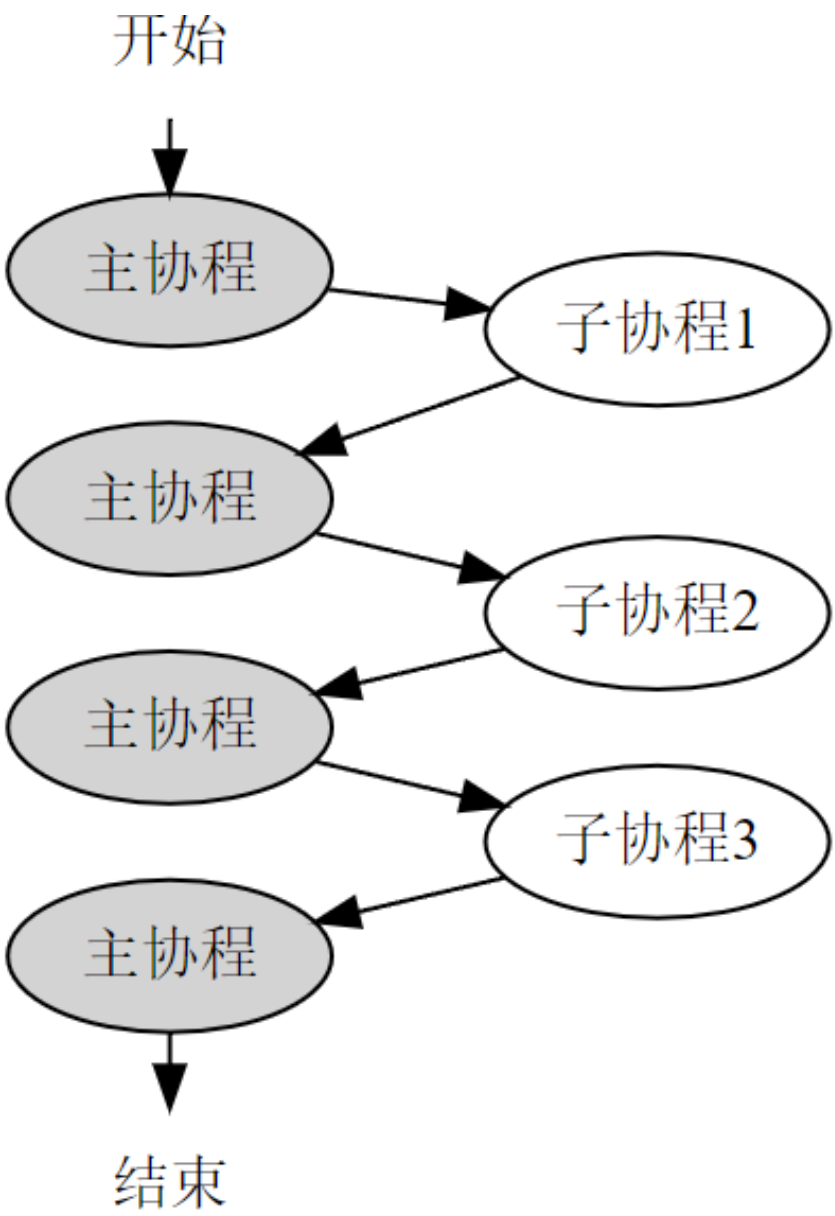
在正式编写协程类之前，首先需要学习一下Linux下的ucontext族函数，ucontext机制是GNU C库提供的一组创建，保存，切换用户态执行上下文的API，这是协程能够随时切换和恢复的关键。

项目里主要会用到的API如下：

```
// 上下文结构体定义
// 这个结构体是平台相关的，因为不同平台的寄存器不一样
// 下面列出的是所有平台都至少会包含的4个成员
typedef struct ucontext_t {
    // 当前上下文结束后，下一个激活的上下文对象的指针，只在当前上下文是由makecontext创建时有效
    struct ucontext_t *uc_link;
    // 当前上下文的信号屏蔽掩码
    sigset_t          uc_sigmask;
    // 当前上下文使用的栈内存空间，只在当前上下文是由makecontext创建时有效
    stack_t           uc_stack;
    // 平台相关的上下文具体内容，包含寄存器的值
    mcontext_t        uc_mcontext;
    ...
} ucontext_t;

// 获取当前的上下文
int getcontext(ucontext_t *ucp);

// 恢复ucp指向的上下文，这个函数不会返回，而是会跳转到ucp上下文对应的函数中执行，相当于变相调用了函数
int setcontext(const ucontext_t *ucp);

// 修改由getcontext获取到的上下文指针ucp，将其与一个函数func进行绑定，支持指定func运行时的参数，
// 在调用makecontext之前，必须手动给ucp分配一段内存空间，存储在ucp->uc_stack中，这段内存空间将作为func函数运行时的栈空间，
// 同时也可以指定ucp->uc_link，表示函数运行结束后恢复uc_link指向的上下文，
// 如果不赋值uc_link，那func函数结束时必须调用setcontext或swapcontext以重新指定一个有效的上下文，否则程序就跑飞了
// makecontext执行完后，ucp就与函数func绑定了，调用setcontext或swapcontext激活ucp时，func就会被运行
void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);

// 恢复ucp指向的上下文，同时将当前的上下文存储到oucp中，
// 和setcontext一样，swapcontext也不会返回，而是会跳转到ucp上下文对应的函数中执行，相当于调用了函数
// swapcontext是sylar非对称协程实现的关键，线程主协程和子协程用这个接口进行上下文切换
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

更详细的介绍和示例可以参考：https://developer.aliyun.com/article/52886

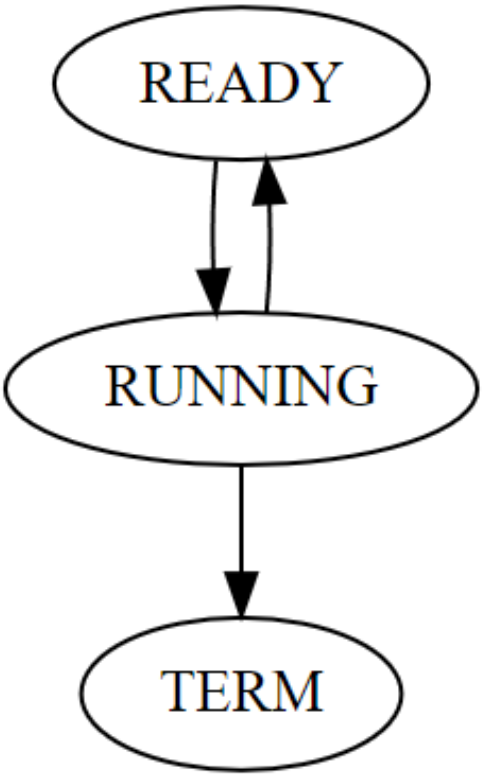
sylar的协程实现使用了非对称模型，且保证子协程不能再创建新的协程，即协程不能嵌套调用，子协程只能与线程主协程进行切换，这种模型简单，非常容易理解。



具体实现上，sylar使用线程局部变量（C++11 thread_local变量）来保存协程上下文对象，这点很好理解，因为协程是在线程里运行的，不同线程的协程相互不影响，每个线程都要独自处理当前线程的协程切换问题。对于每个线程的协程上下文，sylar设计了两个线程局部变量来存储上下文信息（对应源码的t_fiber和t_thread_fiber），也就是说，一个线程在任何时候最多只能知道两个协程的上下文。

```
/// 线程局部变量，当前线程正在运行的协程
static thread_local Fiber *t_fiber = nullptr;
/// 线程局部变量，当前线程的主协程，切换到这个协程，就相当于切换到了主线程中运行，智能指针形式
static thread_local Fiber::ptr t_thread_fiber = nullptr;
```

我们都知道进程有就绪、阻塞等各种状态，要实现协程也要考虑协程的状态，这里我们也尽可能简化，只设置三种协程状态：就绪态、运行态和结束态，一个协程要么正在运行（RUNNING），要么准备运行（READY），要运行结束（TERM）。



对于非对称协程来说，协程除了创建语句外，只有两种操作，一种是resume，表示恢复协程运行，一种是yield，表示让出执行。协程的结束没有专门的操作，协程函数运行结束时协程即结束，协程结束时会自动调用一次yield以返回主协程。

在介绍协程时，我们提到协程相当于一个可以随意切换的函数，在实现协程时当然要给协程绑定一个运行函数，除此之外还要给协程一个运行的栈空间。

到这我们基本都猜出了一个协程类大概应该有哪些成员变量和操作接口了：

```
/**
 * @brief 协程类
 */
class Fiber : public std::enable_shared_from_this<Fiber> {
public:
    typedef std::shared_ptr<Fiber> ptr;

    /**
     * @brief 协程状态
     * @details 在sylar基础上进行了状态简化，只定义三态转换关系，也就是协程要么正在运行(RUNNING)，
     * 要么准备运行(READY)，要么运行结束(TERM)。不区分协程的初始状态，初始即READY。不区分协程是异常结束还是正常结束，
     * 只要结束就是TERM状态。也不区别HOLD状态，协程只要未结束也非运行态，那就是READY状态。
     */
    enum State {
        /// 就绪态，刚创建或者yield之后的状态
        READY,
        /// 运行态，resume之后的状态
        RUNNING,
```

```
        /// 结束态，协程的回调函数执行完之后为TERM状态
        TERM
    };

private:
    /**
     * @brief 构造函数
     * @attention 无参构造函数只用于创建线程的第一个协程，也就是线程主函数对应的协程，
     * 这个协程只能由GetThis()方法调用，所以定义成私有方法
     */
    Fiber();

public:
    /**
     * @brief 构造函数，用于创建用户协程
     * @param[in] cb 协程入口函数
     * @param[in] stacksize 栈大小
     * @param[in] run_in_scheduler 本协程是否参与调度器调度，默认为true
     */
    Fiber(std::function<void()> cb, size_t stacksize = 0, bool run_in_scheduler =
true);

    /**
     * @brief 析构函数
     */
    ~Fiber();

    /**
     * @brief 重置协程状态和入口函数，复用栈空间，不重新创建栈
     * @param[in] cb
     */
    void reset(std::function<void()> cb);

    /**
     * @brief 将当前协程切到到执行状态
     * @details 当前协程和正在运行的协程进行交换，前者状态变为RUNNING，后者状态变为READY
     */
    void resume();

    /**
     * @brief 当前协程让出执行权
     * @details 当前协程与上次resume时退到后台的协程进行交换，前者状态变为READY，后者状态变为
RUNNING
     */
    void yield();

    /**
     * @brief 获取协程ID
     */
```

```
    uint64_t getId() const { return m_id; }

    /**
     * @brief 获取协程状态
     */
    State getState() const { return m_state; }

public:
    /**
     * @brief 设置当前正在运行的协程，即设置线程局部变量t_fiber的值
     */
    static void SetThis(Fiber *f);

    /**
     * @brief 返回当前线程正在执行的协程
     * @details 如果当前线程还未创建协程，则创建线程的第一个协程，
     * 且该协程为当前线程的主协程，其他协程都通过这个协程来调度，也就是说，其他协程
     * 结束时,都要切回到主协程，由主协程重新选择新的协程进行resume
     * @attention 线程如果要创建协程，那么应该首先执行一下Fiber::GetThis()操作，以初始化主函数协程
     */
    static Fiber::ptr GetThis();

    /**
     * @brief 获取总协程数
     */
    static uint64_t TotalFibers();

    /**
     * @brief 协程入口函数
     */
    static void MainFunc();

    /**
     * @brief 获取当前协程id
     */
    static uint64_t GetFiberId();

private:
    /// 协程id
    uint64_t m_id         = 0;
    /// 协程栈大小
    uint32_t m_stacksize = 0;
    /// 协程状态
    State m_state         = READY;
    /// 协程上下文
    ucontext_t m_ctx;
    /// 协程栈地址
    void *m_stack = nullptr;
    /// 协程入口函数
```

```
    std::function<void()> m_cb;
    /// 本协程是否参与调度器调度
    bool m_runInScheduler;
};
```

下面来详细看一下一些关键操作的具体实现:

首先是协程的构建函数。Fiber类提供了两个构造函数，带参数的构造函数用于构造子协程，初始化子协程的ucontext_t上下文和栈空间，要求必须传入协程的入口函数，以及可选的协程栈大小，**sylar采取的是独立栈的形式，每个协程都自己固定大小的栈空间，**协程构造函数要负责分配栈内存空间。不带参的构造函数用于初始化当前线程的协程功能，构造线程主协程对象，以及对t_fiber和t_thread_fiber进行赋值。这个构造函数被定义成私有方法，不允许在类外部调用，只能通过GetThis()方法，在返回当前正在运行的协程时，如果发现当前线程的主协程未被初始化，那就用不带参的构造函数初始化线程主协程。因为GetThis()兼具初始化主协程的功能，在使用协程之前必须显式调用一次GetThis()。

```
/**
 * @brief 构造函数
 * @attention 无参构造函数只用于创建线程的第一个协程，也就是线程主函数对应的协程，
 * 这个协程只能由GetThis()方法调用，所以定义成私有方法
 */
Fiber::Fiber(){
    SetThis(this);
    m_state = RUNNING;

    if (getcontext(&m_ctx)) {
        SYLAR_ASSERT2(false, "getcontext");
    }

    ++s_fiber_count;
    m_id = s_fiber_id++; // 协程id从0开始，用完加1

    SYLAR_LOG_DEBUG(g_logger) << "Fiber::Fiber() main id = " << m_id;
}

/**
 * @brief 构造函数，用于创建用户协程
 * @param[] cb 协程入口函数
 * @param[] stacksize 栈大小，默认为128k
 */
Fiber::Fiber(std::function<void()> cb, size_t stacksize)
    : m_id(s_fiber_id++)
    , m_cb(cb) {
    ++s_fiber_count;
    m_stacksize = stacksize ? stacksize : g_fiber_stack_size->getValue();
    m_stack     = StackAllocator::Alloc(m_stacksize);

    if (getcontext(&m_ctx)) {
        SYLAR_ASSERT2(false, "getcontext");
    }
```

```
    m_ctx.uc_link          = nullptr;
    m_ctx.uc_stack.ss_sp   = m_stack;
    m_ctx.uc_stack.ss_size = m_stacksize;

    makecontext(&m_ctx, &Fiber::MainFunc, 0);

    SYLAR_LOG_DEBUG(g_logger) << "Fiber::Fiber() id = " << m_id;
}

/**
 * @brief 返回当前线程正在执行的协程
 * @details 如果当前线程还未创建协程，则创建线程的第一个协程，
 * 且该协程为当前线程的主协程，其他协程都通过这个协程来调度，也就是说，其他协程
 * 结束时,都要切回到主协程，由主协程重新选择新的协程进行resume
 * @attention 线程如果要创建协程，那么应该首先执行一下Fiber::GetThis()操作，以初始化主函数协程
 */
Fiber::ptr GetThis(){
    if (t_fiber) {
        return t_fiber->shared_from_this();
    }

    Fiber::ptr main_fiber(new Fiber);
    SYLAR_ASSERT(t_fiber == main_fiber.get());
    t_thread_fiber = main_fiber;
    return t_fiber->shared_from_this();
}
```

接下来是协程切换的实现，也就是resume和yield：

```
/**
 * @brief 将当前协程切到到执行状态
 * @details 当前协程和正在运行的协程进行交换，前者状态变为RUNNING，后者状态变为READY
 */
void Fiber::resume() {
    SYLAR_ASSERT(m_state != TERM && m_state != RUNNING);
    SetThis(this);
    m_state = RUNNING;

    if (swapcontext(&(t_thread_fiber->m_ctx), &m_ctx)) {
        SYLAR_ASSERT2(false, "swapcontext");
    }
}

/**
 * @brief 当前协程让出执行权
 * @details 当前协程与上次resume时退到后台的协程进行交换，前者状态变为READY，后者状态变为RUNNING
 */
void Fiber::yield() {
```

```
    /// 协程运行完之后会自动yield一次，用于回到主协程，此时状态已为结束状态
    SYLAR_ASSERT(m_state == RUNNING || m_state == TERM);
    SetThis(t_thread_fiber.get());
    if (m_state != TERM) {
        m_state = READY;
    }

    if (swapcontext(&m_ctx, &(t_thread_fiber->m_ctx))) {
        SYLAR_ASSERT2(false, "swapcontext");
    }
}
```

在非对称协程里，执行resume时的当前执行环境一定是位于线程主协程里，所以这里的swapcontext操作的结果把主协程的上下文保存到t_thread_fiber->m_ctx中，并且激活子协程的上下文；而执行yield时，当前执行环境一定是位于子协程里，所以这里的swapcontext操作的结果是把子协程的上下文保存到协程自己的m_ctx中，同时从t_thread_fiber获得主协程的上下文并激活。

接下来是协程入口函数，sylar在用户传入的协程入口函数上进行了一次封装，这个封装类似于线程模块的对线程入口函数的封装。通过封装协程入口函数，可以实现协程在结束自动执行yield的操作。

```
/**
 * @brief 协程入口函数
 * @note 这里没有处理协程函数出现异常的情况，同样是为了简化状态管理，并且个人认为协程的异常不应该由框架处理，应该由开发者自行处理
 */
void Fiber::MainFunc() {
    Fiber::ptr cur = GetThis(); // GetThis()的shared_from_this()方法让引用计数加1
    SYLAR_ASSERT(cur);

    cur->m_cb(); // 这里真正执行协程的入口函数
    cur->m_cb    = nullptr;
    cur->m_state = TERM;

    auto raw_ptr = cur.get(); // 手动让t_fiber的引用计数减1
    cur.reset();
    raw_ptr->yield(); // 协程结束时自动yield，以回到主协程
}
```

接下来是协程的重置，重置协程就是重复利用已结束的协程，复用其栈空间，创建新协程，实现如下：

```
/**
 * 这里为了简化状态管理，强制只有TERM状态的协程才可以重置，但其实刚创建好但没执行过的协程也应该允许重置的
 */
void Fiber::reset(std::function<void()> cb) {
    SYLAR_ASSERT(m_stack);
    SYLAR_ASSERT(m_state == TERM);
    m_cb = cb;
    if (getcontext(&m_ctx)) {
```

```
        SYLAR_ASSERT2(false, "getcontext");
    }

    m_ctx.uc_link          = nullptr;
    m_ctx.uc_stack.ss_sp   = m_stack;
    m_ctx.uc_stack.ss_size = m_stacksize;

    makecontext(&m_ctx, &Fiber::MainFunc, 0);
    m_state = READY;
}
```

协程调度

当你有很多协程时，如何把这些协程都消耗掉，这就是协程调度。在前面的协程模块中，对于每个协程，都需要用户手动调用协程的resume方法将协程运行起来，然后等协程运行结束并返回，再运行下一个协程。这种运行协程的方式其实是用户自己在挑选协程执行，相当于用户在充当调度器，显然不够灵活。引入协程调度后，则可以先创建一个协程调度器，然后把这些要调度的协程传递给调度器，由调度器负责把这些协程一个一个消耗掉。
我们都学过进程的调度算法：先来先服务、最短作业优先、最高响应比优先、时间片轮转等，协程的调度也类似，我们可以随便选择调度协程的算法，sylar里使用的就是最简单的先来先服务。

来看一个最简单的调度实现：

```
/**
 * @file simple_fiber_scheduler.cc
 * @brief 一个简单的协程调度器实现
 * @version 0.1
 * @date 2021-07-10
 */

#include "sylar/sylar.h"

/**
 * @brief 简单协程调度类，支持添加调度任务以及运行调度任务
 */
class Scheduler {
public:
    /**
     * @brief 添加协程调度任务
     */
    void schedule(sylar::Fiber::ptr task) {
        m_tasks.push_back(task);
    }

    /**
     * @brief 执行调度任务
     */
    void run() {
        sylar::Fiber::ptr task;
        auto it = m_tasks.begin();
```

```
        while(it != m_tasks.end()) {
            task = *it;
            m_tasks.erase(it++);
            task->resume();
        }
    }
private:
    /// 任务队列
    std::list<sylar::Fiber::ptr> m_tasks;
};

void test_fiber(int i) {
    std::cout << "hello world " << i << std::endl;
}

int main() {
    /// 初始化当前线程的主协程
    sylar::Fiber::GetThis();

    /// 创建调度器
    Scheduler sc;

    /// 添加调度任务
    for(auto i = 0; i < 10; i++) {
        sylar::Fiber::ptr fiber(new sylar::Fiber(
            std::bind(test_fiber, i)
        ));
        sc.schedule(fiber);
    }

    /// 执行调度任务
    sc.run();

    return 0;
}
```

是不是非常简单，sylar的协程调度器和上面代码思路完全相同，上面的实现可以看成是sylar的协程调度器的一个特例，当sylar的协程调度器只使用main函数所在线程进行调度时，它的工作原理和上面的完全一样。

接下来将从上面这个调度器开始，来分析一些和协程调度器相关的概念。

首先是关于调度任务的定义，对于协程调度器来说，协程当然可以作为调度任务，但实际上，函数也应可以，因为函数也是可执行的对象，调度器应当支持直接调度一个函数。这在代码实现上也很简单，只需要将函数包装成协程即可，协程调度器的实现重点还是以协程为基础。

接下来是多线程，通过前面协程模块的知识我们可以知道，一个线程同一时刻只能运行一个协程，所以，作为协程调度器，势必要用到多线程来提高调度的效率，因为有多个线程就意味着有多个协程可以同时执行，这显然是要好过单线程的。

既然多线程可以提高协程调度的效率，那么，能不能把调度器所在的线程（称为caller线程）也加入进来作为调度线程呢？比如典型地，在main函数中定义的调度器，能不能把main函数所在的线程也用来执行调度任务呢？答案是肯定的，在实现相同调度能力的情况下（指能够同时调度的协程数量），线程数越小，线程切换的开销也就越小，效率就更高一些，所以，**调度器所在的线程，也应该支持用来执行调度任务。甚至，调度器完全可以不创建新的线程，而只使用caller线程来进行协程调度，比如只使用main函数所在的线程来进行协程调度。**

接下来是调度器如何运行，这里可以简单地认为，调度器创建后，内部首先会创建一个调度线程池，调度开始后，所有调度线程按顺序从任务队列里取任务执行，调度线程数越多，能够同时调度的任务也就越多，当所有任务都调度完后，调度线程就停下来等新的任务进来。

接下来是添加调度任务，添加调度任务的本质就是往调度器的任务队列里塞任务，但是，只添加调度任务是不够的，还应该有一种方式用于通知调度线程有新的任务加进来了，因为调度线程并不一定知道有新任务进来了。当然调度线程也可以不停地轮询有没有新任务，但是这样CPU占用率会很高。

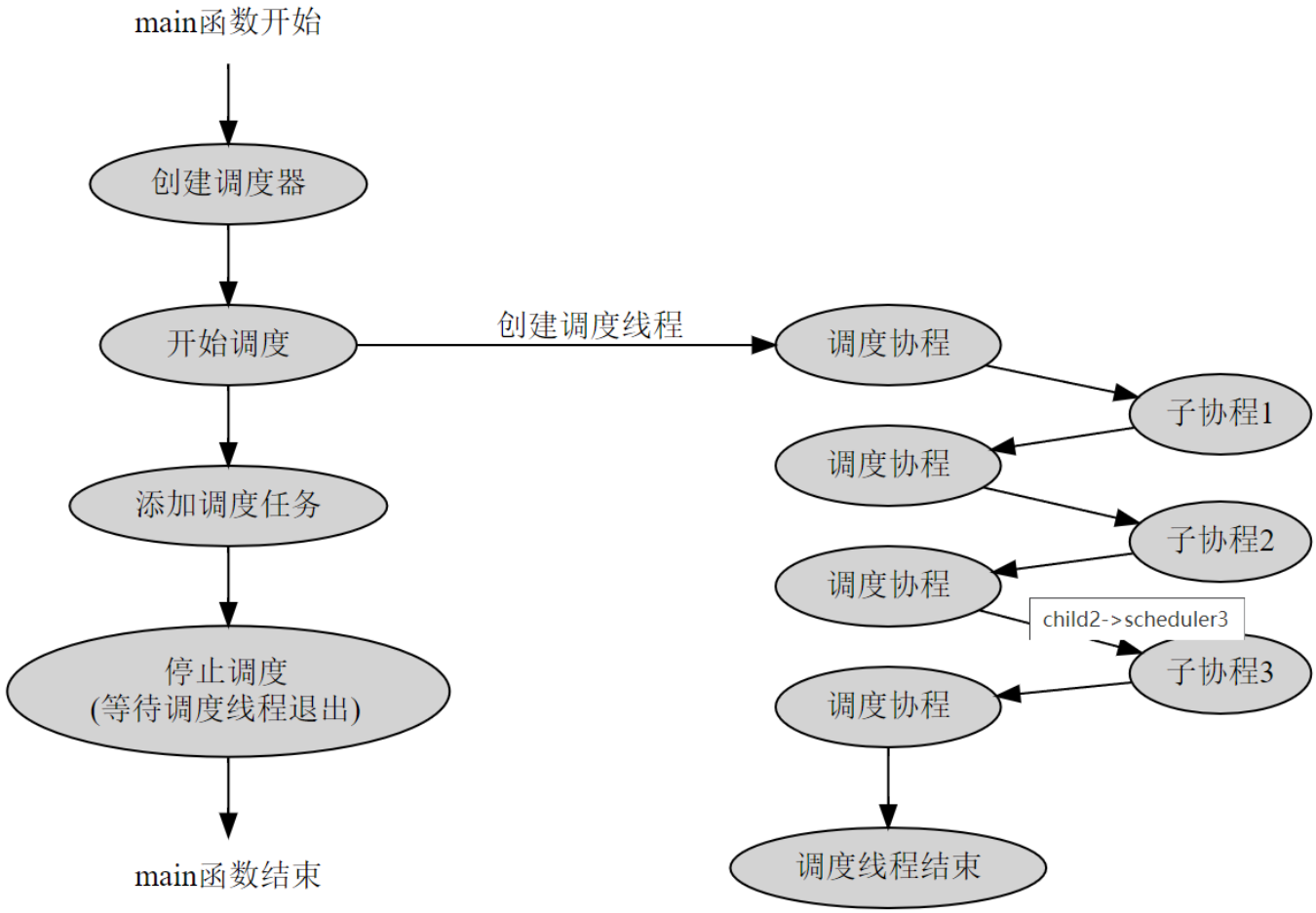
接下来是调度器的停止。调度器应该支持停止调度的功能，以便回收调度线程的资源，只有当所有的调度线程都结束后，调度器才算真正停止。

通过上面的描述，一个协程调度器的大概设计也就出炉了：

调度器内部维护一个任务队列和一个调度线程池。开始调度后，线程池从任务队列里按顺序取任务执行。调度线程可以包含caller线程。当全部任务都执行完了，线程池停止调度，等新的任务进来。添加新任务后，通知线程池有新的任务进来了，线程池重新开始运行调度。停止调度时，各调度线程退出，调度器停止工作。

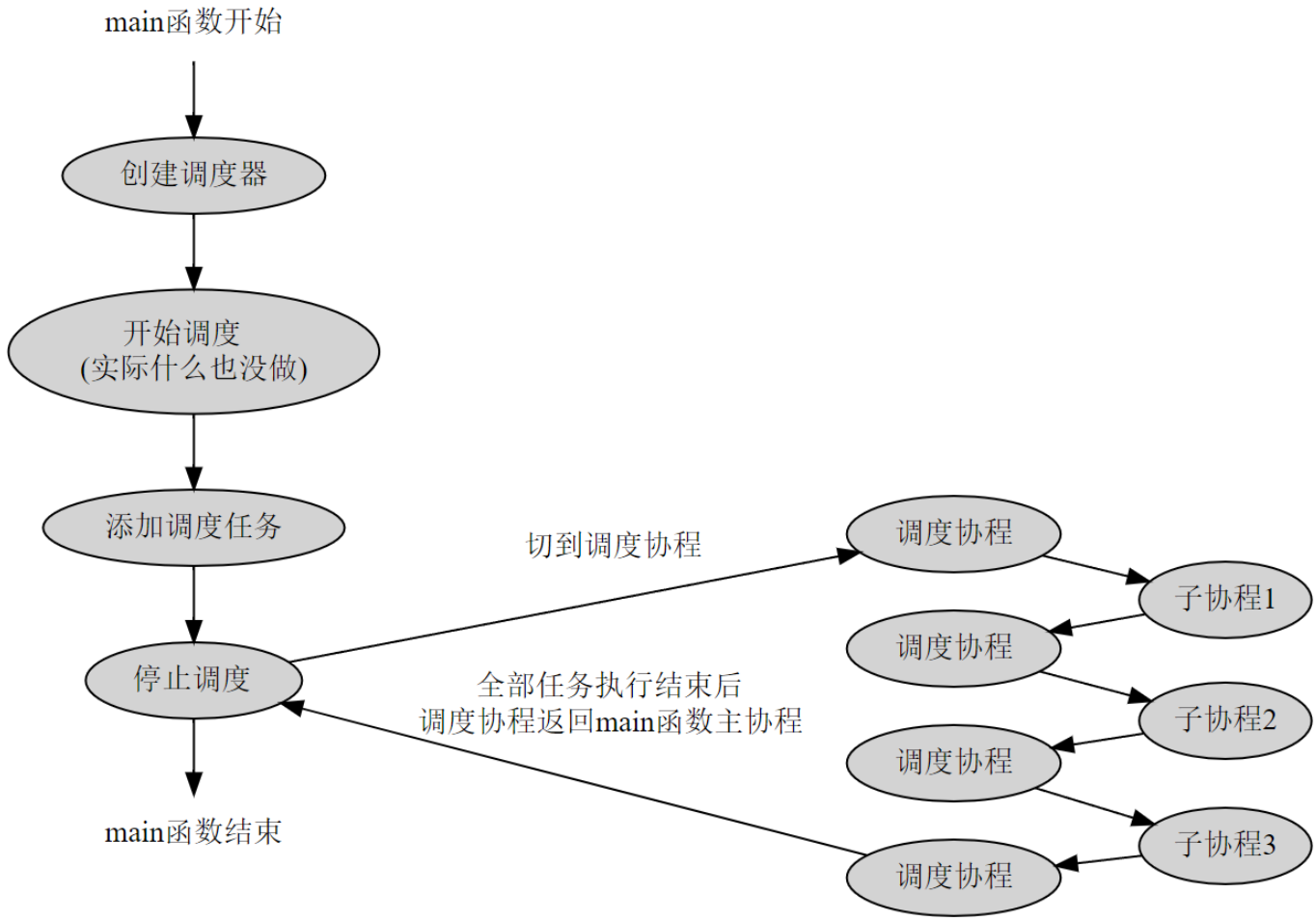
在看具体实现之前要重点理解一下调度时的协程切换问题：

当主线程（调度器线程）不参与调度时，即use_caller为false时，就必须要创建其他线程进行协程调度：



因为有单独的线程用于协程调度，那只需要让新线程的入口函数作为调度协程，从任务队列里取任务执行就行了，main函数与调度协程完全不相关，main函数只需要向调度器添加任务，然后在适当的时机停止调度器即可。当调度器停止时，main函数要等待调度线程结束后再退出。

当主线程也参与调度时，，即use_caller为true时，可以是多线程，也可以是单线程，多线程时调度线程的协程切换如上，那主线程和单线程的时候协程是怎样切换的呢：



梳理一下main函数线程要运行的协程，会发现有以下三类协程：

1. main函数对应的主协程
2. 调度协程
3. 待调度的任务协程

在main函数线程里这三类协程运行的顺序是这样的：

1. main函数主协程运行，创建调度器
2. 仍然是main函数主协程运行，向调度器添加一些调度任务
3. 开始协程调度，main函数主协程让出执行权，切换到调度协程，调度协程从任务队列里按顺序执行所有的任务
4. 每次执行一个任务，调度协程都要让出执行权，再切到该任务的协程里去执行，任务执行结束后，还要再切回调度协程，继续下一个任务的调度
5. 所有任务都执行完后，调度协程还要让出执行权并切回main函数主协程，以保证程序能顺利结束。

在具体的实现上，前面提到sylar的子协程只能和线程主协程切换，而不能和另一个子协程切换。在上面的情况1中，线程主协程是main函数对应的协程，另外的两类协程，也就是调度协程和任务协程，都是子协程，也就是说，调度协程不能直接和任务协程切换，一旦切换，程序的main函数协程就跑飞了。

解决单线程环境下caller线程主协程-调度协程-任务协程之间的上下文切换，是sylar协程调度实现的关键。

其实，子协程和子协程切换导致线程主协程跑飞的关键原因在于，每个线程只有两个线程局部变量用于保存当前的协程上下文信息。也就是说线程任何时候都最多只能知道两个协程的上下文，其中一个是当前正在运行协程的上下文，另一个是线程主协程的上下文，如果子协程和子协程切换，那这两个上下文都会变成子协程的上下文，线程主协程的上下文丢失了，程序也就跑飞了。如果不改变这种局部，就只能线程主协程去充当调度协程，这就相当于又回到了让用户充当调度器的情况。

那么，如何改变这种情况呢？其实非常简单，只需要给每个线程增加一个线程局部变量用于保存调度协程的上下文就可以了，这样，每个线程可以同时保存三个协程的上下文，一个是当前正在执行的协程上下文，另一个是线程主协程的上下文，最后一个是调度协程的上下文。有了这三个上下文，协程就可以根据自己的身份来选择和每次和哪个协程进行交换，具体操作如下：

1. 给协程类增加一个bool类型的成员m_runInScheduler，用于记录该协程是否通过调度器来运行。
2. 创建协程时，根据协程的身份指定对应的协程类型，具体来说，只有想让调度器调度的协程的m_runInScheduler值为true，线程主协程和线程调度协程的m_runInScheduler都为false。
3. resume一个协程时，如果如果这个协程的m_runInScheduler值为true，表示这个协程参与调度器调度，那它应该和三个线程局部变量中的调度协程上下文进行切换，同理，在协程yield时，也应该恢复调度协程的上下文，表示从子协程切换回调度协程；
4. 如果协程的m_runInScheduler值为false，表示这个协程不参与调度器调度，那么在resume协程时，直接和线程主协程切换就可以了，yield也一样，应该恢复线程主协程的上下文。m_runInScheduler值为false的协程上下文切换完全和调度协程无关，可以脱离调度器使用。

下面就可以来看一下协程调度的源码实现了：

首先是要对上面协程类的实现做一些改造，增加m_runInScheduler成员，表示当前协程是否参与调度器调度，在协程的resume和yield时，根据协程的运行环境确定是和线程主协程进行交换还是和调度协程进行交换：

```
Fiber::Fiber(std::function<void()> cb, size_t stacksize, bool run_in_scheduler)
    : m_id(s_fiber_id++)
    , m_cb(cb)
    , m_runInScheduler(run_in_scheduler) {
    ++s_fiber_count;
    m_stacksize = stacksize ? stacksize : g_fiber_stack_size->getValue();
    m_stack     = StackAllocator::Alloc(m_stacksize);

    if (getcontext(&m_ctx)) {
        SYLAR_ASSERT2(false, "getcontext");
    }

    m_ctx.uc_link          = nullptr;
    m_ctx.uc_stack.ss_sp   = m_stack;
    m_ctx.uc_stack.ss_size = m_stacksize;

    makecontext(&m_ctx, &Fiber::MainFunc, 0);

    SYLAR_LOG_DEBUG(g_logger) << "Fiber::Fiber() id = " << m_id;
}

void Fiber::resume() {
    SYLAR_ASSERT(m_state != TERM && m_state != RUNNING);
    SetThis(this);
    m_state = RUNNING;

    // 如果协程参与调度器调度，那么应该和调度器的主协程进行swap，而不是线程主协程
    if (m_runInScheduler) {
        if (swapcontext(&(Scheduler::GetMainFiber()->m_ctx), &m_ctx)) {
            SYLAR_ASSERT2(false, "swapcontext");
```

```
        }
    } else {
        if (swapcontext(&(t_thread_fiber->m_ctx), &m_ctx)) {
            SYLAR_ASSERT2(false, "swapcontext");
        }
    }
}

void Fiber::yield() {
    /// 协程运行完之后会自动yield一次，用于回到主协程，此时状态已为结束状态
    SYLAR_ASSERT(m_state == RUNNING || m_state == TERM);
    SetThis(t_thread_fiber.get());
    if (m_state != TERM) {
        m_state = READY;
    }

    // 如果协程参与调度器调度，那么应该和调度器的主协程进行swap，而不是线程主协程
    if (m_runInScheduler) {
        if (swapcontext(&m_ctx, &(Scheduler::GetMainFiber()->m_ctx))) {
            SYLAR_ASSERT2(false, "swapcontext");
        }
    } else {
        if (swapcontext(&m_ctx, &(t_thread_fiber->m_ctx))) {
            SYLAR_ASSERT2(false, "swapcontext");
        }
    }
}
```

下面就来看一下调度器类的成员：

```
/**
 * @brief 协程调度器
 * @details 封装的是N-M的协程调度器
 *          内部有一个线程池,支持协程在线程池里面切换
 */
class Scheduler {
public:
    typedef std::shared_ptr<Scheduler> ptr;
    typedef Mutex MutexType;

    /**
     * @brief 创建调度器
     * @param[in] threads 线程数
     * @param[in] use_caller 是否将当前线程也作为调度线程
     * @param[in] name 名称
     */
    Scheduler(size_t threads = 1, bool use_caller = true, const std::string &name =
"Scheduler");
```

```
    /**
     * @brief 析构函数
     */
    virtual ~Scheduler();

    /**
     * @brief 获取调度器名称
     */
    const std::string &getName() const { return m_name; }

    /**
     * @brief 获取当前线程调度器指针
     */
    static Scheduler *GetThis();

    /**
     * @brief 获取当前线程的主协程
     */
    static Fiber *GetMainFiber();

    /**
     * @brief 添加调度任务
     * @tparam FiberOrCb 调度任务类型，可以是协程对象或函数指针
     * @param[] fc 协程对象或指针
     * @param[] thread 指定运行该任务的线程号，-1表示任意线程
     */
    template <class FiberOrCb>
    void schedule(FiberOrCb fc, int thread = -1) {
        bool need_tickle = false;
        {
            MutexType::Lock lock(m_mutex);
            need_tickle = scheduleNoLock(fc, thread);
        }

        if (need_tickle) {
            tickle(); // 唤醒idle协程
        }
    }

    /**
     * @brief 启动调度器
     */
    void start();

    /**
     * @brief 停止调度器，等所有调度任务都执行完了再返回
     */
    void stop();
```

```
protected:
    /**
     * @brief 通知协程调度器有任务了
     */
    virtual void tickle();

    /**
     * @brief 协程调度函数
     */
    void run();

    /**
     * @brief 无任务调度时执行idle协程
     */
    virtual void idle();

    /**
     * @brief 返回是否可以停止
     */
    virtual bool stopping();

    /**
     * @brief 设置当前的协程调度器
     */
    void setThis();

    /**
     * @brief 返回是否有空闲线程
     * @details 当调度协程进入idle时空闲线程数加1，从idle协程返回时空闲线程数减1
     */
    bool hasIdleThreads() { return m_idleThreadCount > 0; }
private:
    /**
     * @brief 调度任务，协程/函数二选一，可指定在哪个线程上调度
     */
    struct ScheduleTask {
        Fiber::ptr fiber;
        std::function<void()> cb;
        int thread;

        ScheduleTask(Fiber::ptr f, int thr) {
            fiber  = f;
            thread = thr;
        }
        ScheduleTask(Fiber::ptr *f, int thr) {
            fiber.swap(*f);
            thread = thr;
        }
        ScheduleTask(std::function<void()> f, int thr) {
```

```
            cb     = f;
            thread = thr;
        }
        ScheduleTask() { thread = -1; }

        void reset() {
            fiber  = nullptr;
            cb     = nullptr;
            thread = -1;
        }
    };

private:
    /// 协程调度器名称
    std::string m_name;
    /// 互斥锁
    MutexType m_mutex;
    /// 线程池
    std::vector<Thread::ptr> m_threads;
    /// 任务队列
    std::list<ScheduleTask> m_tasks;
    /// 线程池的线程ID数组
    std::vector<int> m_threadIds;
    /// 工作线程数量，不包含use_caller的主线程
    size_t m_threadCount = 0;
    /// 活跃线程数
    std::atomic<size_t> m_activeThreadCount = {0};
    /// idle线程数
    std::atomic<size_t> m_idleThreadCount = {0};

    /// 是否use caller
    bool m_useCaller;
    /// use_caller为true时，调度器所在线程的调度协程
    Fiber::ptr m_rootFiber;
    /// use_caller为true时，调度器所在线程的id
    int m_rootThread = 0;

    /// 是否正在停止
    bool m_stopping = false;
};
```

下面来看一下协程调度模块的全局变量和线程局部变量，这里只有以下两个线程局部变量：

```
/// 当前线程的调度器，同一个调度器下的所有线程指同同一个调度器实例
static thread_local Scheduler *t_scheduler = nullptr;
/// 当前线程的调度协程，每个线程都独有一份，包括caller线程
static thread_local Fiber *t_scheduler_fiber = nullptr;
```

t_scheduler_fiber保存当前线程的调度协程，加上前面协程模块的t_fiber和t_thread_fiber，每个线程总共可以记录三个协程的上下文信息。

调度器的构造方法如下：

```
/**
 * @brief 创建调度器
 * @param[in] threads 线程数
 * @param[in] use_caller 是否将当前线程也作为调度线程
 * @param[in] name 名称
 */
Scheduler::Scheduler(size_t threads, bool use_caller, const std::string &name) {
    SYLAR_ASSERT(threads > 0);

    m_useCaller = use_caller;
    m_name      = name;

    if (use_caller) {
        --threads;
        sylar::Fiber::GetThis();
        SYLAR_ASSERT(GetThis() == nullptr);
        t_scheduler = this;

        /**
         * 在user_caller为true的情况下，初始化caller线程的调度协程
         * caller线程的调度协程不会被调度器调度，而且，caller线程的调度协程停止时，应该返回caller线程的主协程
         */
        m_rootFiber.reset(new Fiber(std::bind(&Scheduler::run, this), 0, false));

        sylar::Thread::SetName(m_name);
        t_scheduler_fiber = m_rootFiber.get();
        m_rootThread      = sylar::GetThreadId();
        m_threadIds.push_back(m_rootThread);
    } else {
        m_rootThread = -1;
    }
    m_threadCount = threads;
}

Scheduler *Scheduler::GetThis() {
    return t_scheduler;
}
```

接下来是协程调度器的start方法实现，这里主要初始化调度线程池，如果只使用caller线程进行调度，那这个方法啥也不做：

```
void Scheduler::start() {
    SYLAR_LOG_DEBUG(g_logger) << "start";
```

```
    MutexType::Lock lock(m_mutex);
    if (m_stopping) {
        SYLAR_LOG_ERROR(g_logger) << "Scheduler is stopped";
        return;
    }
    SYLAR_ASSERT(m_threads.empty());
    m_threads.resize(m_threadCount);
    for (size_t i = 0; i < m_threadCount; i++) {
        m_threads[i].reset(new Thread(std::bind(&Scheduler::run, this),
                                      m_name + "_" + std::to_string(i)));
        m_threadIds.push_back(m_threads[i]->getId());
    }
}
```

接下来是调度协程的实现，内部有一个while(true)循环，不停地从任务队列取任务并执行，由于Fiber类改造过，每个被调度器执行的协程在结束时都会回到调度协程，所以这里不用担心跑飞问题，当任务队列为空时，代码会进idle协程，但idle协程啥也不做直接就yield了，状态还是READY状态，所以这里其实就是个忙等待，CPU占用率爆炸，只有当调度器检测到停止标志时，idle协程才会真正结束，调度协程也会检测到idle协程状态为TERM，并且随之退出整个调度协程。这里还可以看出一点，对于一个任务协程，只要其从resume中返回了，那不管它的状态是TERM还是READY，调度器都不会自动将其再次加入调度，因为前面说过，一个成熟的协程是要学会自我管理的。

```
void Scheduler::run() {
    SYLAR_LOG_DEBUG(g_logger) << "run";
    setThis();
    if (sylar::GetThreadId() != m_rootThread) {
        t_scheduler_fiber = sylar::Fiber::GetThis().get();
    }

    Fiber::ptr idle_fiber(new Fiber(std::bind(&Scheduler::idle, this)));
    Fiber::ptr cb_fiber;

    ScheduleTask task;
    while (true) {
        task.reset();
        bool tickle_me = false; // 是否tickle其他线程进行任务调度
        {
            MutexType::Lock lock(m_mutex);
            auto it = m_tasks.begin();
            // 遍历所有调度任务
            while (it != m_tasks.end()) {
                if (it->thread != -1 && it->thread != sylar::GetThreadId()) {
                    // 指定了调度线程，但不是在当前线程上调度，标记一下需要通知其他线程进行调度，然
后跳过这个任务，继续下一个
                    ++it;
                    tickle_me = true;
                    continue;
                }

                // 找到一个未指定线程，或是指定了当前线程的任务
```

```
                SYLAR_ASSERT(it->fiber || it->cb);
                if (it->fiber) {
                    // 任务队列时的协程一定是READY状态，谁会把RUNNING或TERM状态的协程加入调度呢?
                    SYLAR_ASSERT(it->fiber->getState() == Fiber::READY);
                }
                // 当前调度线程找到一个任务，准备开始调度，将其从任务队列中剔除，活动线程数加1
                task = *it;
                m_tasks.erase(it++);
                ++m_activeThreadCount;
                break;
            }
            // 当前线程拿完一个任务后，发现任务队列还有剩余，那么tickle一下其他线程
            tickle_me |= (it != m_tasks.end());
        }

        if (tickle_me) {
            tickle();
        }

        if (task.fiber) {
            // resume协程，resume返回时，协程要么执行完了，要么半路yield了，总之这个任务就算完成了，活跃线程数减一
            task.fiber->resume();
            --m_activeThreadCount;
            task.reset();
        } else if (task.cb) {
            if (cb_fiber) {
                cb_fiber->reset(task.cb);
            } else {
                cb_fiber.reset(new Fiber(task.cb));
            }
            task.reset();
            cb_fiber->resume();
            --m_activeThreadCount;
            cb_fiber.reset();
        } else {
            // 进到这个分支情况一定是任务队列空了，调度idle协程即可
            if (idle_fiber->getState() == Fiber::TERM) {
                // 如果调度器没有调度任务，那么idle协程会不停地resume/yield，不会结束，如果idle协程结束了，那一定是调度器停止了
                SYLAR_LOG_DEBUG(g_logger) << "idle fiber term";
                break;
            }
            ++m_idleThreadCount;
            idle_fiber->resume();
            --m_idleThreadCount;
        }
    }
    SYLAR_LOG_DEBUG(g_logger) << "Scheduler::run() exit";
```

```
}
```

最后是调度器的stop方法，在使用了caller线程的情况下，调度器依赖stop方法来执行caller线程的调度协程，如果调度器只使用了caller线程来调度，那调度器真正开始执行调度的位置就是这个stop方法。

```
void Scheduler::stop() {
    SYLAR_LOG_DEBUG(g_logger) << "stop";
    if (stopping()) {
        return;
    }
    m_stopping = true;

    /// 如果use caller，那只能由caller线程发起stop
    if (m_useCaller) {
        SYLAR_ASSERT(GetThis() == this);
    } else {
        SYLAR_ASSERT(GetThis() != this);
    }

    for (size_t i = 0; i < m_threadCount; i++) {
        tickle();
    }

    if (m_rootFiber) {
        tickle();
    }

    /// 在use caller情况下，调度器协程结束时，应该返回caller协程
    if (m_rootFiber) {
        m_rootFiber->resume();
        SYLAR_LOG_DEBUG(g_logger) << "m_rootFiber end";
    }

    std::vector<Thread::ptr> thrs;
    {
        MutexType::Lock lock(m_mutex);
        thrs.swap(m_threads);
    }
    for (auto &i : thrs) {
        i->join();
    }
}
```

协程+IO

IO事件调度功能对服务器开发至关重要，因为服务器通常需要处理大量来自客户端的socket fd，使用IO事件调度可以将开发者从判断socket fd是否可读或可写的工作中解放出来，使得程序员只需要关心socket fd的IO操作，实现IO协程调度意义重大。

IO协程调度可以看成是增强版的协程调度。在前面的协程调度模块中，调度器对协程的调度是无条件执行的，在调度器已经启动调度的情况下，任务一旦添加成功，就会排队等待调度器执行。调度器不支持删除调度任务，并且调度器在正常退出之前一定会执行完全部的调度任务，所以在某种程度上可以认为，把一个协程添加到调度器的任务队列，就相当于调用了协程的resume方法。

IO协程调度支持协程调度的全部功能，因为IO协程调度器是直接继承协程调度器实现的。除了协程调度，IO协程调度还增加了IO事件调度的功能，这个功能是针对描述符（一般是套接字描述符）的。IO协程调度支持为描述符注册可读和可写事件的回调函数，当描述符可读或可写时，执行对应的回调函数。（这里可以直接把回调函数等效成协程，所以这个功能被称为IO协程调度）

很多的库都可以实现类似的工作，比如libevent，libuv，libev等，这些库被称为异步事件库或异步IO库，从网上可以很容易地找到大把的资料介绍这类库。有的库不仅可以处理socket fd事件，还可以处理定时器事件和信号事件。这些事件库的实现原理基本类似，都是先将套接字设置成非阻塞状态，然后将套接字与回调函数绑定，接下来进入一个基于IO多路复用的事件循环，等待事件发生，然后调用对应的回调函数。

sylar的IO协程调度模块基于epoll实现，只支持Linux平台。对每个fd，sylar支持两类事件，一类是可读事件，对应EPOLLIN，一类是可写事件，对应EPOLLOUT，sylar的事件枚举值直接继承自epoll。

当然epoll本身除了支持了EPOLLIN和EPOLLOUT两类事件外，还支持其他事件，比如EPOLLRDHUP, EPOLLERR, EPOLLHUP等，对于这些事件，sylar的做法是将其进行归类，分别对应到EPOLLIN和EPOLLOUT中，也就是所有的事件都可以表示为可读或可写事件，甚至有的事件还可以同时表示可读及可写事件，比如EPOLLERR事件发生时，fd将同时触发可读和可写事件。

对于IO协程调度来说，每次调度都包含一个三元组信息，分别是描述符-事件类型（可读或可写）-回调函数，调度器记录全部需要调度的三元组信息，其中描述符和事件类型用于epoll_wait，回调函数用于协程调度。这个三元组信息在源码上通过FdContext结构体来存储，在执行epoll_wait时通过epoll_event的私有数据指针data.ptr来保存FdContext结构体信息。

IO协程调度器在idle时会epoll_wait所有注册的fd，如果有fd满足条件，epoll_wait返回，从私有数据中拿到fd的上下文信息，并且执行其中的回调函数。（实际是idle协程只负责收集所有已触发的fd的回调函数并将其加入调度器的任务队列，真正的执行时机是idle协程退出后，调度器在下一轮调度时执行）。

与协程调度器不一样的是，IO协程调度器支持取消事件。取消事件表示不关心某个fd的某个事件了，如果某个fd的可读或可写事件都被取消了，那这个fd会从调度器的epoll_wait中删除。

IO协程调度器对应IOManager，这个类直接继承自Scheduler：

```
class IOManager : public Scheduler {
public:
    typedef std::shared_ptr<IOManager> ptr;
    typedef RWMutex RWMutexType;
...
}
```

首先是读写事件的定义，这里直接继承epoll的枚举值，如下：

```
/**
 * @brief IO事件，继承自epoll对事件的定义
 * @details 这里只关心socket fd的读和写事件，其他epoll事件会归类到这两类事件中
 */
enum Event {
    /// 无事件
    NONE = 0x0,
    /// 读事件(EPOLLIN)
    READ = 0x1,
    /// 写事件(EPOLLOUT)
    WRITE = 0x4,
};
```

接下来是对描述符-事件类型-回调函数三元组的定义，这个三元组也称为fd上下文，使用结构体FdContext来表示。由于fd有可读和可写两种事件，每种事件的回调函数也可以不一样，所以每个fd都需要保存两个事件类型-回调函数组合。FdContext结构体定义如下：

```
/**
 * @brief socket fd上下文类
 * @details 每个socket fd都对应一个FdContext，包括fd的值，fd上的事件，以及fd的读写事件上下文
 */
struct FdContext {
    typedef Mutex MutexType;
    /**
     * @brief 事件上下文类
     * @details fd的每个事件都有一个事件上下文，保存这个事件的回调函数以及执行回调函数的调度器
     *          sylar对fd事件做了简化，只预留了读事件和写事件，所有的事件都被归类到这两类事件中
     */
    struct EventContext {
        /// 执行事件回调的调度器
        Scheduler *scheduler = nullptr;
        /// 事件回调协程
        Fiber::ptr fiber;
        /// 事件回调函数
        std::function<void()> cb;
    };

    /**
     * @brief 获取事件上下文类
     * @param[in] event 事件类型
     * @return 返回对应事件的上下文
     */
    EventContext &getEventContext(Event event);

    /**
     * @brief 重置事件上下文
     * @param[in, out] ctx 待重置的事件上下文对象
     */
```

```
    void resetEventContext(EventContext &ctx);

    /**
     * @brief 触发事件
     * @details 根据事件类型调用对应上下文结构中的调度器去调度回调协程或回调函数
     * @param[in] event 事件类型
     */
    void triggerEvent(Event event);

    /// 读事件上下文
    EventContext read;
    /// 写事件上下文
    EventContext write;
    /// 事件关联的句柄
    int fd = 0;
    /// 该fd添加了哪些事件的回调函数，或者说该fd关心哪些事件
    Event events = NONE;
    /// 事件的Mutex
    MutexType mutex;
};
```

接下来是IOManager的成员变量。IOManager包含一个epoll实例的句柄m_epfd以及用于tickle的一对pipe fd，还有全部的fd上下文m_fdContexts，如下：

```
/// epoll 文件句柄
int m_epfd = 0;
/// pipe 文件句柄，fd[0]读端，fd[1]写端
int m_tickleFds[2];
/// 当前等待执行的IO事件数量
std::atomic<size_t> m_pendingEventCount = {0};
/// IOManager的Mutex
RWMutexType m_mutex;
/// socket事件上下文的容器
std::vector<FdContext *> m_fdContexts;
```

接下来是在继承类IOManager中改造协程调度器，使其支持epoll，并重载tickle和idle，实现通知调度协程和IO协程调度功能：

```
/**
 * @brief 构造函数
 * @param[in] threads 线程数量
 * @param[in] use_caller 是否将调用线程包含进去
 * @param[in] name 调度器的名称
 */
IOManager::IOManager(size_t threads, bool use_caller, const std::string &name)
    : Scheduler(threads, use_caller, name) {
    // 创建epoll实例
    m_epfd = epoll_create(5000);
```

```
    SYLAR_ASSERT(m_epfd > 0);

    // 创建pipe，获取m_tickleFds[2]，其中m_tickleFds[0]是管道的读端，m_tickleFds[1]是管道的写端
    int rt = pipe(m_tickleFds);
    SYLAR_ASSERT(!rt);

    // 注册pipe读句柄的可读事件，用于tickle调度协程，通过epoll_event.data.fd保存描述符
    epoll_event event;
    memset(&event, 0, sizeof(epoll_event));
    event.events  = EPOLLIN | EPOLLET;
    event.data.fd = m_tickleFds[0];

    // 非阻塞方式，配合边缘触发
    rt = fcntl(m_tickleFds[0], F_SETFL, O_NONBLOCK);
    SYLAR_ASSERT(!rt);

    // 将管道的读描述符加入epoll多路复用，如果管道可读，idle中的epoll_wait会返回
    rt = epoll_ctl(m_epfd, EPOLL_CTL_ADD, m_tickleFds[0], &event);
    SYLAR_ASSERT(!rt);

    contextResize(32);

    // 这里直接开启了Schedluer，也就是说IOManager创建即可调度协程
    start();
}

/**
 * @brief 通知调度器有任务要调度
 * @details 写pipe让idle协程从epoll_wait退出，待idle协程yield之后Scheduler::run就可以调度其他任务
 * 如果当前没有空闲调度线程，那就没必要发通知
 */
void IOManager::tickle() {
    SYLAR_LOG_DEBUG(g_logger) << "tickle";
    if(!hasIdleThreads()) {
        return;
    }
    int rt = write(m_tickleFds[1], "T", 1);
    SYLAR_ASSERT(rt == 1);
}

/**
 * @brief idle协程
 * @details 对于IO协程调度来说，应阻塞在等待IO事件上，idle退出的时机是epoll_wait返回，对应的操作是tickle或注册的IO事件就绪
 * 调度器无调度任务时会阻塞idle协程上，对IO调度器而言，idle状态应该关注两件事，一是有没有新的调度任务，对应Schduler::schedule(),
```

```
 * 如果有新的调度任务，那应该立即退出idle状态，并执行对应的任务；二是关注当前注册的所有IO事件有没有触发，如果有触发，那么应该执行
 * IO事件对应的回调函数
 */
void IOManager::idle() {
    SYLAR_LOG_DEBUG(g_logger) << "idle";

    // 一次epoll_wait最多检测256个就绪事件，如果就绪事件超过了这个数，那么会在下轮epoll_wati继续处理
    const uint64_t MAX_EVNETS = 256;
    epoll_event *events       = new epoll_event[MAX_EVNETS]();
    std::shared_ptr<epoll_event> shared_events(events, [](epoll_event *ptr) {
        delete[] ptr;
    });

    while (true) {
        if(stopping()) {
            SYLAR_LOG_DEBUG(g_logger) << "name=" << getName() << "idle stopping exit";
            break;
        }
        // 阻塞在epoll_wait上，等待事件发生
        static const int MAX_TIMEOUT = 5000;
        int rt = epoll_wait(m_epfd, events, MAX_EVNETS, MAX_TIMEOUT);
        if(rt < 0) {
            if(errno == EINTR) {
                continue;
            }
            SYLAR_LOG_ERROR(g_logger) << "epoll_wait(" << m_epfd << ") (rt="
                                      << rt << ") (errno=" << errno << ") (errstr:" << strerror(errno) << ")";
            break;
        }

        // 遍历所有发生的事件，根据epoll_event的私有指针找到对应的FdContext，进行事件处理
        for (int i = 0; i < rt; ++i) {
            epoll_event &event = events[i];
            if (event.data.fd == m_tickleFds[0]) {
                // ticklefd[0]用于通知协程调度，这时只需要把管道里的内容读完即可，本轮idle结束Scheduler::run会重新执行协程调度
                uint8_t dummy[256];
                while (read(m_tickleFds[0], dummy, sizeof(dummy)) > 0)
                    ;
                continue;
            }

            // 通过epoll_event的私有指针获取FdContext
            FdContext *fd_ctx = (FdContext *)event.data.ptr;
            FdContext::MutexType::Lock lock(fd_ctx->mutex);
            /**
```

```
             * EPOLLERR: 出错，比如写读端已经关闭的pipe
             * EPOLLHUP: 套接字对端关闭
             * 出现这两种事件，应该同时触发fd的读和写事件，否则有可能出现注册的事件永远执行不到的情况
             */
            if (event.events & (EPOLLERR | EPOLLHUP)) {
                event.events |= (EPOLLIN | EPOLLOUT) & fd_ctx->events;
            }
            int real_events = NONE;
            if (event.events & EPOLLIN) {
                real_events |= READ;
            }
            if (event.events & EPOLLOUT) {
                real_events |= WRITE;
            }

            if ((fd_ctx->events & real_events) == NONE) {
                continue;
            }

            // 剔除已经发生的事件，将剩下的事件重新加入epoll_wait,
            // 如果剩下的事件为0，表示这个fd已经不需要关注了，直接从epoll中删除
            int left_events = (fd_ctx->events & ~real_events);
            int op          = left_events ? EPOLL_CTL_MOD : EPOLL_CTL_DEL;
            event.events    = EPOLLET | left_events;

            int rt2 = epoll_ctl(m_epfd, op, fd_ctx->fd, &event);
            if (rt2) {
                SYLAR_LOG_ERROR(g_logger) << "epoll_ctl(" << m_epfd << ", "
                                          << (EpollCtlOp)op << ", " << fd_ctx->fd << ",
" << (EPOLL_EVENTS)event.events << "):"
                                          << rt2 << " (" << errno << ") (" <<
strerror(errno) << ")";
                continue;
            }

            // 处理已经发生的事件，也就是让调度器调度指定的函数或协程
            if (real_events & READ) {
                fd_ctx->triggerEvent(READ);
                --m_pendingEventCount;
            }
            if (real_events & WRITE) {
                fd_ctx->triggerEvent(WRITE);
                --m_pendingEventCount;
            }
        } // end for

        /**
         * 一旦处理完所有的事件，idle协程yield，这样可以让调度协程(Scheduler::run)重新检查是否有新
任务要调度
```

```
            * 上面triggerEvent实际也只是把对应的fiber重新加入调度，要执行的话还要等idle协程退出
            */
            Fiber::ptr cur = Fiber::GetThis();
            auto raw_ptr   = cur.get();
            cur.reset();

            raw_ptr->yield();
        } // end while(true)
    }
```

接下来是注册事件回调addEvent，删除事件回调delEvent，取消事件回调cancelEvent，以及取消全部事件cancelAll：

```
/**
 * @brief 添加事件
 * @details fd描述符发生了event事件时执行cb函数
 * @param[in] fd socket句柄
 * @param[in] event 事件类型
 * @param[in] cb 事件回调函数，如果为空，则默认把当前协程作为回调执行体
 * @return 添加成功返回0,失败返回-1
 */
int IOManager::addEvent(int fd, Event event, std::function<void()> cb) {
    // 找到fd对应的FdContext，如果不存在，那就分配一个
    FdContext *fd_ctx = nullptr;
    RWMutexType::ReadLock lock(m_mutex);
    if ((int)m_fdContexts.size() > fd) {
        fd_ctx = m_fdContexts[fd];
        lock.unlock();
    } else {
        lock.unlock();
        RWMutexType::WriteLock lock2(m_mutex);
        contextResize(fd * 1.5);
        fd_ctx = m_fdContexts[fd];
    }

    // 同一个fd不允许重复添加相同的事件
    FdContext::MutexType::Lock lock2(fd_ctx->mutex);
    if (SYLAR_UNLIKELY(fd_ctx->events & event)) {
        SYLAR_LOG_ERROR(g_logger) << "addEvent assert fd=" << fd
                                  << " event=" << (EPOLL_EVENTS)event
                                  << " fd_ctx.event=" << (EPOLL_EVENTS)fd_ctx->events;
        SYLAR_ASSERT(!(fd_ctx->events & event));
    }

    // 将新的事件加入epoll_wait，使用epoll_event的私有指针存储FdContext的位置
    int op = fd_ctx->events ? EPOLL_CTL_MOD : EPOLL_CTL_ADD;
    epoll_event epevent;
    epevent.events   = EPOLLET | fd_ctx->events | event;
    epevent.data.ptr = fd_ctx;
```

```
    int rt = epoll_ctl(m_epfd, op, fd, &epevent);
    if (rt) {
        SYLAR_LOG_ERROR(g_logger) << "epoll_ctl(" << m_epfd << ", "
                                  << (EpollCtlOp)op << ", " << fd << ", " <<
(EPOLL_EVENTS)epevent.events << "):"
                                  << rt << " (" << errno << ") (" << strerror(errno) <<
") fd_ctx->events="
                                  << (EPOLL_EVENTS)fd_ctx->events;
        return -1;
    }

    // 待执行IO事件数加1
    ++m_pendingEventCount;

    // 找到这个fd的event事件对应的EventContext，对其中的scheduler，cb，fiber进行赋值
    fd_ctx->events                      = (Event)(fd_ctx->events | event);
    FdContext::EventContext &event_ctx = fd_ctx->getEventContext(event);
    SYLAR_ASSERT(!event_ctx.scheduler && !event_ctx.fiber && !event_ctx.cb);

    // 赋值scheduler和回调函数，如果回调函数为空，则把当前协程当成回调执行体
    event_ctx.scheduler = Scheduler::GetThis();
    if (cb) {
        event_ctx.cb.swap(cb);
    } else {
        event_ctx.fiber = Fiber::GetThis();
        SYLAR_ASSERT2(event_ctx.fiber->getState() == Fiber::RUNNING, "state=" <<
event_ctx.fiber->getState());
    }
    return 0;
}
/**
 * @brief 删除事件
 * @param[in] fd socket句柄
 * @param[in] event 事件类型
 * @attention 不会触发事件
 * @return 是否删除成功
 */
bool IOManager::delEvent(int fd, Event event) {
    // 找到fd对应的FdContext
    RWMutexType::ReadLock lock(m_mutex);
    if ((int)m_fdContexts.size() <= fd) {
        return false;
    }
    FdContext *fd_ctx = m_fdContexts[fd];
    lock.unlock();

    FdContext::MutexType::Lock lock2(fd_ctx->mutex);
    if (SYLAR_UNLIKELY(!(fd_ctx->events & event))) {
```

```
        return false;
    }

    // 清除指定的事件，表示不关心这个事件了，如果清除之后结果为0，则从epoll_wait中删除该文件描述符
    Event new_events = (Event)(fd_ctx->events & ~event);
    int op           = new_events ? EPOLL_CTL_MOD : EPOLL_CTL_DEL;
    epoll_event epevent;
    epevent.events   = EPOLLET | new_events;
    epevent.data.ptr = fd_ctx;

    int rt = epoll_ctl(m_epfd, op, fd, &epevent);
    if (rt) {
        SYLAR_LOG_ERROR(g_logger) << "epoll_ctl(" << m_epfd << ", "
                                  << (EpollCtlOp)op << ", " << fd << ", " <<
(EPOLL_EVENTS)epevent.events << "):"
                                  << rt << " (" << errno << ") (" << strerror(errno) <<
")";
        return false;
    }

    // 待执行事件数减1
    --m_pendingEventCount;
    // 重置该fd对应的event事件上下文
    fd_ctx->events                     = new_events;
    FdContext::EventContext &event_ctx = fd_ctx->getEventContext(event);
    fd_ctx->resetEventContext(event_ctx);
    return true;
}


/**
 * @brief 取消事件
 * @param[in] fd socket句柄
 * @param[in] event 事件类型
 * @attention 如果该事件被注册过回调，那就触发一次回调事件
 * @return 是否删除成功
 */
bool IOManager::cancelEvent(int fd, Event event) {
    // 找到fd对应的FdContext
    RWMutexType::ReadLock lock(m_mutex);
    if ((int)m_fdContexts.size() <= fd) {
        return false;
    }
    FdContext *fd_ctx = m_fdContexts[fd];
    lock.unlock();

    FdContext::MutexType::Lock lock2(fd_ctx->mutex);
    if (SYLAR_UNLIKELY(!(fd_ctx->events & event))) {
        return false;
```

```
    }

    // 删除事件
    Event new_events = (Event)(fd_ctx->events & ~event);
    int op           = new_events ? EPOLL_CTL_MOD : EPOLL_CTL_DEL;
    epoll_event epevent;
    epevent.events   = EPOLLET | new_events;
    epevent.data.ptr = fd_ctx;

    int rt = epoll_ctl(m_epfd, op, fd, &epevent);
    if (rt) {
        SYLAR_LOG_ERROR(g_logger) << "epoll_ctl(" << m_epfd << ", "
                                  << (EpollCtlOp)op << ", " << fd << ", " <<
(EPOLL_EVENTS)epevent.events << "):"
                                  << rt << " (" << errno << ") (" << strerror(errno) <<
")";
        return false;
    }

    // 删除之前触发一次事件
    fd_ctx->triggerEvent(event);
    // 活跃事件数减1
    --m_pendingEventCount;
    return true;
}

/**
 * @brief 取消所有事件
 * @details 所有被注册的回调事件在cancel之前都会被执行一次
 * @param[in] fd socket句柄
 * @return 是否删除成功
 */
bool IOManager::cancelAll(int fd) {
    // 找到fd对应的FdContext
    RWMutexType::ReadLock lock(m_mutex);
    if ((int)m_fdContexts.size() <= fd) {
        return false;
    }
    FdContext *fd_ctx = m_fdContexts[fd];
    lock.unlock();

    FdContext::MutexType::Lock lock2(fd_ctx->mutex);
    if (!fd_ctx->events) {
        return false;
    }

    // 删除全部事件
    int op = EPOLL_CTL_DEL;
    epoll_event epevent;
```

```
    epevent.events   = 0;
    epevent.data.ptr = fd_ctx;

    int rt = epoll_ctl(m_epfd, op, fd, &epevent);
    if (rt) {
        SYLAR_LOG_ERROR(g_logger) << "epoll_ctl(" << m_epfd << ", "
                                  << (EpollCtlOp)op << ", " << fd << ", " <<
(EPOLL_EVENTS)epevent.events << "):"
                                  << rt << " (" << errno << ") (" << strerror(errno) <<
")";
        return false;
    }

    // 触发全部已注册的事件
    if (fd_ctx->events & READ) {
        fd_ctx->triggerEvent(READ);
        --m_pendingEventCount;
    }
    if (fd_ctx->events & WRITE) {
        fd_ctx->triggerEvent(WRITE);
        --m_pendingEventCount;
    }

    SYLAR_ASSERT(fd_ctx->events == 0);
    return true;
}
```

接下来是IOManager的析构函数实现和stopping重载。对于IOManager的析构，首先要等Scheduler调度完所有的任务，然后再关闭epoll句柄和pipe句柄，然后释放所有的FdContext；对于stopping，IOManager在判断是否可退出时，还要加上所有IO事件都完成调度的条件：

```
IOManager::~IOManager() {
    stop();
    close(m_epfd);
    close(m_tickleFds[0]);
    close(m_tickleFds[1]);

    for (size_t i = 0; i < m_fdContexts.size(); ++i) {
        if (m_fdContexts[i]) {
            delete m_fdContexts[i];
        }
    }
}
bool IOManager::stopping() {
    // 对于IOManager而言，必须等所有待调度的IO事件都执行完了才可以退出
    return m_pendingEventCount == 0 && Scheduler::stopping();
}
```

总得来说，IO协程调度模块可分为两部分，第一部分是对协程调度器的改造，将epoll与协程调度融合，重新实现

tickle和idle，并保证原有的功能不变。第二部分是基于epoll实现IO事件的添加、删除、调度、取消等功能。

IO协程调度关注的是FdContext信息，也就是描述符-事件-回调函数三元组，IOManager需要保存所有关注的三元组，并且在epoll_wait检测到描述符事件就绪时执行对应的回调函数。epoll是线程安全的，即使调度器有多个调度线程，它们也可以共用同一个epoll实例，而不用担心互斥。由于空闲时所有线程都阻塞的epoll_wait上，所以也不用担心CPU占用问题。

addEvent是一次性的，比如说，注册了一个读事件，当fd可读时会触发该事件，但触发完之后，这次注册的事件就失效了，后面fd再次可读时，并不会继续执行该事件回调，如果要持续触发事件的回调，那每次事件处理完都要手动再addEvent。这样在应对fd的WRITE事件时会比较好处理，因为fd可写是常态，如果注册一次就一直有效，那么可写事件就必须在执行完之后就删除掉。

定时器

本项目的定时器是为了实现协程调度器对定时任务的调度，服务器上经常要处理定时事件，比如3秒后关闭一个连接，或是定期检测一个客户端的连接状态。，定时器也是后面hook模块的基础。

定时器大家都很熟悉了，在WebServer里也用到了，常见的定时器实现有升序链表、高性能时间轮、时间堆等，《Linux高性能服务器编程》第11章对定时器有完整而详细地介绍，网上各种也有各种介绍，这里就不展开介绍了，只看一下本项目里定时器的实现。

无论是升序链表还是时间轮的设计都依赖一个固定周期触发的tick信号。设计定时器的另一种实现思路是直接将超时时间当作tick周期，具体操作是每次都取出所有定时器中超时时间最小的超时值作为一个tick，这样，一旦tick触发，超时时间最小的定时器必然到期。处理完已超时的定时器后，再从剩余的定时器中找出超时时间最小的一个，并将这个最小时间作为下一个tick，如此反复，就可以实现较为精确的定时。最小堆很适合处理这种定时方案，将所有定时器按最小堆来组织，可以很方便地获取到当前的最小超时时间，sylar采取的即是这种方案。

sylar的定时器采用最小堆设计，所有定时器根据绝对的超时时间点进行排序，每次取出离当前时间最近的一个超时时间点，计算出超时需要等待的时间，然后等待超时。超时时间到后，获取当前的绝对时间点，然后把最小堆里超时时间点小于这个时间点的定时器都收集起来，执行它们的回调函数。

注意，在注册定时事件时，一般提供的是相对时间，比如相对当前时间3秒后执行。sylar会根据传入的相对时间和当前的绝对时间计算出定时器超时时的绝对时间点，然后根据这个绝对时间点对定时器进行最小堆排序。因为依赖的是系统绝对时间，所以需要考虑校时因素，这点会在后面讨论。

sylar定时器的超时等待基于epoll_wait，精度只支持毫秒级，因为epoll_wait的超时精度也只有毫秒级。

关于定时器和IO协程调度器的整合。IO协程调度器的idle协程会在调度器空闲时阻塞在epoll_wait上，等待IO事件发生。在之前的代码里，epoll_wait具有固定的超时时间，这个值是5秒钟。加入定时器功能后，epoll_wait的超时时间改用当前定时器的最小超时时间来代替。epoll_wait返回后，根据当前的绝对时间把已超时的所有定时器收集起来，执行它们的回调函数。

由于epoll_wait的返回并不一定是超时引起的，也有可能是IO事件唤醒的，所以在epoll_wait返回后不能想当然地假设定时器已经超时了，而是要再判断一下定时器有没有超时，这时绝对时间的好处就体现出来了，通过比较当前的绝对时间和定时器的绝对超时时间，就可以确定一个定时器到底有没有超时。

sylar的定时器对应Timer类，这个类的成员变量包括定时器的绝对超时时间点，是否重复执行，回调函数，以及一个指向TimerManager的指针，提供cancel/reset/refresh方法用于操作定时器。构造Timer时可以传入超时时间，也可以直接传入一个绝对时间。Timer的构造函数被定义成私有方式，只能通过TimerManager类来创建Timer对象。除此外，Timer类还提供了一个仿函数Comparator，用于比较两个Timer对象，比较的依据是绝对超时时间。

```
class TimerManager;
```

```cpp
/**
 * @brief 定时器
 */
class Timer : public std::enable_shared_from_this<Timer> {
friend class TimerManager;
public:
    /// 定时器的智能指针类型
    typedef std::shared_ptr<Timer> ptr;

    /**
     * @brief 取消定时器
     */
    bool cancel();

    /**
     * @brief 刷新设置定时器的执行时间
     */
    bool refresh();

    /**
     * @brief 重置定时器时间
     * @param[in] ms 定时器执行间隔时间(毫秒)
     * @param[in] from_now 是否从当前时间开始计算
     */
    bool reset(uint64_t ms, bool from_now);
private:
    /**
     * @brief 构造函数
     * @param[in] ms 定时器执行间隔时间
     * @param[in] cb 回调函数
     * @param[in] recurring 是否循环
     * @param[in] manager 定时器管理器
     */
    Timer(uint64_t ms, std::function<void()> cb,
          bool recurring, TimerManager* manager);
    /**
     * @brief 构造函数
     * @param[in] next 执行的时间戳(毫秒)
     */
    Timer(uint64_t next);
private:
    /// 是否循环定时器
    bool m_recurring = false;
    /// 执行周期
    uint64_t m_ms = 0;
    /// 精确的执行时间
    uint64_t m_next = 0;
    /// 回调函数
    std::function<void()> m_cb;
```

```
    /// 定时器管理器
    TimerManager* m_manager = nullptr;
private:
    /**
     * @brief 定时器比较仿函数
     */
    struct Comparator {
        /**
         * @brief 比较定时器的智能指针的大小(按执行时间排序)
         * @param[in] lhs 定时器智能指针
         * @param[in] rhs 定时器智能指针
         */
        bool operator()(const Timer::ptr& lhs, const Timer::ptr& rhs) const;
    };
};
```

所有的Timer对象都由TimerManager类进行管理，TimerManager包含一个std::set类型的Timer集合，这个集合就是定时器的最小堆结构，因为set里的元素总是排序过的，所以总是可以很方便地获取到当前的最小定时器。TimerManager提供创建定时器，获取最近一个定时器的超时时间，以及获取全部已经超时的定时器回调函数的方法，并且提供了一个onTimerInsertedAtFront()方法，这是一个虚函数，由IOManager继承时实现，当新的定时器插入到Timer集合的首部时，TimerManager通过该方法来通知IOManager立刻更新当前的epoll_wait超时。TimerManager还负责检测是否发生了校时，由detectClockRollover方法实现。

```
**
 * @brief 定时器管理器
 */
class TimerManager {
friend class Timer;
public:
    /// 读写锁类型
    typedef RWMutex RWMutexType;

    /**
     * @brief 构造函数
     */
    TimerManager();

    /**
     * @brief 析构函数
     */
    virtual ~TimerManager();

    /**
     * @brief 添加定时器
     * @param[in] ms 定时器执行间隔时间
     * @param[in] cb 定时器回调函数
     * @param[in] recurring 是否循环定时器
     */
    Timer::ptr addTimer(uint64_t ms, std::function<void()> cb
```

```
                        ,bool recurring = false);

    /**
     * @brief 添加条件定时器
     * @param[in] ms 定时器执行间隔时间
     * @param[in] cb 定时器回调函数
     * @param[in] weak_cond 条件
     * @param[in] recurring 是否循环
     */
    Timer::ptr addConditionTimer(uint64_t ms, std::function<void()> cb
                        ,std::weak_ptr<void> weak_cond
                        ,bool recurring = false);

    /**
     * @brief 到最近一个定时器执行的时间间隔(毫秒)
     */
    uint64_t getNextTimer();

    /**
     * @brief 获取需要执行的定时器的回调函数列表
     * @param[out] cbs 回调函数数组
     */
    void listExpiredCb(std::vector<std::function<void()> >& cbs);

    /**
     * @brief 是否有定时器
     */
    bool hasTimer();
protected:

    /**
     * @brief 当有新的定时器插入到定时器的首部,执行该函数
     */
    virtual void onTimerInsertedAtFront() = 0;

    /**
     * @brief 将定时器添加到管理器中
     */
    void addTimer(Timer::ptr val, RWMutexType::WriteLock& lock);
private:
    /**
     * @brief 检测服务器时间是否被调后了
     */
    bool detectClockRollover(uint64_t now_ms);
private:
    /// Mutex
    RWMutexType m_mutex;
    /// 定时器集合
    std::set<Timer::ptr, Timer::Comparator> m_timers;
```

```
    /// 是否触发onTimerInsertedAtFront
    bool m_tickled = false;
    /// 上次执行时间
    uint64_t m_previousTime = 0;
};
```

IOManager通过继承的方式获得TimerManager类的所有方法，这种方式相当于给IOManager外挂了一个定时器管理模块。为支持定时器功能，需要重新改造idle协程的实现，epoll_wait应该根据下一个定时器的超时时间来设置超时参数。

```
class IOManager : public Scheduler, public TimerManager {
...
}

void IOManager::idle() {
    SYLAR_LOG_DEBUG(g_logger) << "idle";

    // 一次epoll_wait最多检测256个就绪事件，如果就绪事件超过了这个数，那么会在下轮epoll_wati继续处理
    const uint64_t MAX_EVNETS = 256;
    epoll_event *events       = new epoll_event[MAX_EVNETS]();
    std::shared_ptr<epoll_event> shared_events(events, [](epoll_event *ptr) {
        delete[] ptr;
    });

    while (true) {
        // 获取下一个定时器的超时时间，顺便判断调度器是否停止
        uint64_t next_timeout = 0;
        if( SYLAR_UNLIKELY(stopping(next_timeout))) {
            SYLAR_LOG_DEBUG(g_logger) << "name=" << getName() << "idle stopping exit";
            break;
        }

        // 阻塞在epoll_wait上，等待事件发生或定时器超时
        int rt = 0;
        do{
            // 默认超时时间5秒，如果下一个定时器的超时时间大于5秒，仍以5秒来计算超时，避免定时器超时时间太大时，epoll_wait一直阻塞
            static const int MAX_TIMEOUT = 5000;
            if(next_timeout != ~0ull) {
                next_timeout = std::min((int)next_timeout, MAX_TIMEOUT);
            } else {
                next_timeout = MAX_TIMEOUT;
            }
            rt = epoll_wait(m_epfd, events, MAX_EVNETS, (int)next_timeout);
            if(rt < 0 && errno == EINTR) {
                continue;
            } else {
```

```
                break;
            }
        } while(true);

        // 收集所有已超时的定时器，执行回调函数
        std::vector<std::function<void()>> cbs;
        listExpiredCb(cbs);
        if(!cbs.empty()) {
            for(const auto &cb : cbs) {
                schedule(cb);
            }
            cbs.clear();
        }
        ...
}
```

hook

什么是hook？hook实际上就是对系统调用API进行一次封装，将其封装成一个与原始的系统调用API同名的接口，应用在调用这个接口时，会先执行封装中的操作，再执行原始的系统调用API。hook技术可以使应用程序在执行系统调用之前进行一些隐藏的操作，比如可以对系统提供malloc()和free()进行hook，在真正进行内存分配和释放之前，统计内存的引用计数，以排查内存泄露问题。

本项目的hook有什么用呢？前面提到，协程的一个很大的好处就是发生阻塞时可以随时切换、提供资源利用率，我们的协程库就是要实现使用我们协程库编写的服务器程序没有阻塞阶段，也就是当前协程发生阻塞就切换到其他协程，当阻塞协程就绪的时候在再切换回来。hook和IO协程调度是密切相关的，如果不使用IO协程调度器，那hook没有任何意义，考虑IOManager要在一个线程上按顺序调度以下协程：

协程1：sleep(2) 睡眠两秒后返回。

协程2：在scoket fd1 上send 100k数据。

协程3：在socket fd2 上recv直到数据接收成功。

在未hook的情况下，IOManager要调度上面的协程，流程是下面这样的：

调度协程1，协程阻塞在sleep上，等2秒后返回，这两秒内调度线程是被协程1占用的，其他协程无法在当前线程上调度。

调度协徨2，协程阻塞send 100k数据上，这个操作一般问题不大，因为send数据无论如何都要占用时间，但如果fd迟迟不可写，那send会阻塞直到套接字可写，同样，在阻塞期间，其他协程也无法在当前线程上调度。

调度协程3，协程阻塞在recv上，这个操作要直到recv超时或是有数据时才返回，期间调度器也无法调度其他协程。

上面的调度流程最终总结起来就是，协程只能按顺序调度，一旦有一个协程阻塞住了，那整个调度线程也就阻塞住了，其他的协程都无法在当前线程上执行。像这种一条路走到黑的方式其实并不是完全不可避免，以sleep为例，调度器完全可以在检测到协程sleep后，将协程yield以让出执行权，同时设置一个定时器，2秒后再将协程重新resume。这样，调度器就可以在这2秒期间调度其他的任务，同时还可以顺利的实现sleep 2秒后再继续执行协程的效果，send/recv与此类似。在完全实现hook后，IOManager的执行流程将变成下面的方式：

调度协程1，检测到协程sleep，那么先添加一个2秒的定时器，定时器回调函数是在调度器上继续调度本协程，接着协程yield，等定时器超时。

因为上一步协程1已经yield了，所以协程2并不需要等2秒后才可以执行，而是立刻可以执行。同样，调度器检测到协程send，由于不知道fd是不是马上可写，所以先在IOManager上给fd注册一个写事件，回调函数是让当前协程resume并执行实际的send操作，然后当前协程yield，等可写事件发生。

上一步协程2也yield了，可以马上调度协程3。协程3与协程2类似，也是给fd注册一个读事件，回调函数是让当前协程resume并继续recv，然后本协程yield，等事件发生。

等2秒超时后，执行定时器回调函数，将协程1 resume以便继续执行。

等协程2的fd可写，一旦可写，调用写事件回调函数将协程2 resume以便继续执行send。

等协程3的fd可读，一旦可读，调用回调函数将协程3 resume以便继续执行recv。

上面的4、5、6步都是异步的，调度线程并不会阻塞，IOManager仍然可以调度其他的任务，只在相关的事件发生后，再继续执行对应的任务即可。并且，由于hook的函数签名与原函数一样，所以对调用方也很方便，只需要以同步的方式编写代码，实现的效果却是异步执行的，效率很高。

总而言之，在IO协程调度中对相关的系统调用进行hook，可以让调度线程尽可能得把时间片都花在有意义的操作上，而不是浪费在阻塞等待中。

hook的重点是在替换API的底层实现的同时完全模拟其原本的行为，因为调用方是不知道hook的细节的，在调用被hook的API时，如果其行为与原本的行为不一致，就会给调用方造成困惑。比如，所有的socket fd在进行IO调度时都会被设置成NONBLOCK模式，如果用户未显式地对fd设置NONBLOCK，那就要处理好fcntl，不要对用户暴露fd已经是NONBLOCK的事实，这点也说明，除了IO相关的函数要进行hook外，对fcntl, setsockopt之类的功能函数也要进行hook，才能保证API的一致性。

hook有两种方式侵入式和外挂式，侵入式很简单，就是直接改造代码，将目标函数的入口点替换为自定义的代码，从而在函数执行之前或之后注入自定义逻辑。外挂式hook是优先加自定义载动态库来实现对后加载的动态库进行hook。比如我们自己实现了write函数：

```
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>

ssize_t write(int fd, const void *buf, size_t count) {
    syscall(SYS_write, STDOUT_FILENO, "12345\n", strlen("12345\n"));
}
```

将其比编译成libhook.so动态库，通过设置 LD_PRELOAD环境变量，将libhoook.so设置成优先加载。

```
 LD_PRELOAD="./libhook.so" ./a.out
```

LD_PRELOAD环境变量，它指明了在运行a.out之前，系统会优先把libhook.so加载到了程序的进程空间，使得在a.out运行之前，其全局符号表中就已经有了一个write符号，这样在后续加载libc共享库时，由于全局符号介入机制，libc中的write符号不会再被加入全局符号表，所以全局符号表中的write就变成了我们自己的实现。
关于hook模块和IO协程调度的整合。一共有三类接口需要hook，如下：

1. sleep延时系列接口，包括sleep/usleep/nanosleep。对于这些接口的hook，只需要给IO协程调度器注册一个

定时事件，在定时事件触发后再继续执行当前协程即可。当前协程在注册完定时事件后即可yield让出执行权。

2. socket IO系列接口，包括read/write/recv/send...等，connect及accept也可以归到这类接口中。这类接口的hook首先需要判断操作的fd是否是socket fd，以及用户是否显式地对该fd设置过非阻塞模式，如果不是socket fd或是用户显式设置过非阻塞模式，那么就不需要hook了，直接调用操作系统的IO接口即可。如果需要hook，那么首先在IO协程调度器上注册对应的读写事件，等事件发生后再继续执行当前协程。当前协程在注册完IO事件即可yield让出执行权
3. socket/fcntl/ioctl/close等接口，这类接口主要处理的是边缘情况，比如分配fd上下文，处理超时及用户显式设置非阻塞问题。首先是socket fd上下文和FdManager的实现，这两个类用于记录fd上下文和保存全部的fd上下文，它们的关键实现如下：

```
/**
 * @brief 文件句柄上下文类
 * @details 管理文件句柄类型(是否socket)
 *          是否阻塞,是否关闭,读/写超时时间
 */
class FdCtx : public std::enable_shared_from_this<FdCtx> {
public:
    typedef std::shared_ptr<FdCtx> ptr;
    /**
     * @brief 通过文件句柄构造FdCtx
     */
    FdCtx(int fd);
    /**
     * @brief 析构函数
     */
    ~FdCtx();
    ....
private:
    /// 是否初始化
    bool m_isInit: 1;
    /// 是否socket
    bool m_isSocket: 1;
    /// 是否hook非阻塞
    bool m_sysNonblock: 1;
    /// 是否用户主动设置非阻塞
    bool m_userNonblock: 1;
    /// 是否关闭
    bool m_isClosed: 1;
    /// 文件句柄
    int m_fd;
    /// 读超时时间毫秒
    uint64_t m_recvTimeout;
    /// 写超时时间毫秒
    uint64_t m_sendTimeout;
};

/**
 * @brief 文件句柄管理类
```

```
 */
class FdManager {
public:
    typedef RWMutex RWMutexType;
    /**
     * @brief 无参构造函数
     */
    FdManager();

    /**
     * @brief 获取/创建文件句柄类FdCtx
     * @param[in] fd 文件句柄
     * @param[in] auto_create 是否自动创建
     * @return 返回对应文件句柄类FdCtx::ptr
     */
    FdCtx::ptr get(int fd, bool auto_create = false);

    /**
     * @brief 删除文件句柄类
     * @param[in] fd 文件句柄
     */
    void del(int fd);
private:
    /// 读写锁
    RWMutexType m_mutex;
    /// 文件句柄集合
    std::vector<FdCtx::ptr> m_datas;
};

/// 文件句柄单例
typedef Singleton<FdManager> FdMgr;
```

FdCtx类在用户态记录了fd的读写超时和非阻塞信息，其中非阻塞包括用户显式设置的非阻塞和hook内部设置的非阻塞，区分这两种非阻塞可以有效应对用户对fd设置/获取NONBLOCK模式的情形。

另外注意一点，FdManager类对FdCtx的寻址采用了和IOManager中对FdContext的寻址一样的寻址方式，直接用fd作为数组下标进行寻址。

接下来是hook的整体实现。首先定义线程局部变量t_hook_enable，用于表示当前线程是否启用hook，使用线程局部变量表示hook模块是线程粒度的，各个线程可单独启用或关闭hook。然后是获取各个被hook的接口的原始地址， 这里要借助dlsym来获取。sylar使用了一套宏来简化编码，这套宏的实现如下：

```
#define HOOK_FUN(XX) \
    XX(sleep) \
    XX(usleep) \
    XX(nanosleep) \
    XX(socket) \
    XX(connect) \
    XX(accept) \
```

```
    XX(read) \
    XX(readv) \
    XX(recv) \
    XX(recvfrom) \
    XX(recvmsg) \
    XX(write) \
    XX(writev) \
    XX(send) \
    XX(sendto) \
    XX(sendmsg) \
    XX(close) \
    XX(fcntl) \
    XX(ioctl) \
    XX(getsockopt) \
    XX(setsockopt)

extern "C" {
#define XX(name) name ## _fun name ## _f = nullptr;
    HOOK_FUN(XX);
#undef XX
}

void hook_init() {
    static bool is_inited = false;
    if(is_inited) {
        return;
    }
#define XX(name) name ## _f = (name ## _fun)dlsym(RTLD_NEXT, #name);
    HOOK_FUN(XX);
#undef XX
}
```

上面的宏展开之后的效果如下：

```
extern "C" {
    sleep_fun sleep_f = nullptr; \
    usleep_fun usleep_f = nullptr; \
    ....
    setsocketopt_fun setsocket_f = nullptr;
};

hook_init() {
    ...

    sleep_f = (sleep_fun)dlsym(RTLD_NEXT, "sleep"); \
    usleep_f = (usleep_fun)dlsym(RTLD_NEXT, "usleep"); \
    ...
    setsocketopt_f = (setsocketopt_fun)dlsym(RTLD_NEXT, "setsocketopt");
}
```

hook_init() 放在一个静态对象的构造函数中调用，这表示在main函数运行之前就会获取各个符号的地址并保存在全局变量中。

最后是各个接口的hook实现，这部分和上面的全局变量定义要放在extern "C"中，以防止C++编译器对符号名称添加修饰。由于被hook的接口要完全模拟原接口的行为，所以这里要小心处理好各种边界情况以及返回值和errno问题。

首先是sleep/usleep/nanosleep的hook实现，它们的实现思路完全一样，即先添加定时器再yield，比如sleep函数的hook代码如下：

```
unsigned int sleep(unsigned int seconds) {
    if(!sylar::t_hook_enable) {
        return sleep_f(seconds);
    }

    sylar::Fiber::ptr fiber = sylar::Fiber::GetThis();
    sylar::IOManager* iom = sylar::IOManager::GetThis();
    iom->addTimer(seconds * 1000, std::bind((void(sylar::Scheduler::*)
            (sylar::Fiber::ptr, int thread))&sylar::IOManager::schedule
            ,iom, fiber, -1));
    sylar::Fiber::GetThis()->yield();
    return 0;
}
```

接下来是socket接口的hook实现，socket用于创建套接字，需要在拿到fd后将其添加到FdManager中，代码实现如下：

```
int socket(int domain, int type, int protocol) {
    if(!sylar::t_hook_enable) {
        return socket_f(domain, type, protocol);
    }
    int fd = socket_f(domain, type, protocol);
    if(fd == -1) {
        return fd;
    }
    sylar::FdMgr::GetInstance()->get(fd, true);
    return fd;
}
```

接下来是connect和connect_with_timeout的实现，由于connect有默认的超时，所以这里只需要实现connect_with_timeout即可：

```
int connect_with_timeout(int fd, const struct sockaddr* addr, socklen_t addrlen,
uint64_t timeout_ms) {
    if(!sylar::t_hook_enable) {
        return connect_f(fd, addr, addrlen);
    }
    sylar::FdCtx::ptr ctx = sylar::FdMgr::GetInstance()->get(fd);
```

```
    if(!ctx || ctx->isClose()) {
        errno = EBADF;
        return -1;
    }

    if(!ctx->isSocket()) {
        return connect_f(fd, addr, addrlen);
    }

    if(ctx->getUserNonblock()) {
        return connect_f(fd, addr, addrlen);
    }

    int n = connect_f(fd, addr, addrlen);
    if(n == 0) {
        return 0;
    } else if(n != -1 || errno != EINPROGRESS) {
        return n;
    }

    sylar::IOManager* iom = sylar::IOManager::GetThis();
    sylar::Timer::ptr timer;
    std::shared_ptr<timer_info> tinfo(new timer_info);
    std::weak_ptr<timer_info> winfo(tinfo);

    if(timeout_ms != (uint64_t)-1) {
        timer = iom->addConditionTimer(timeout_ms, [winfo, fd, iom]() {
                auto t = winfo.lock();
                if(!t || t->cancelled) {
                    return;
                }
                t->cancelled = ETIMEDOUT;
                iom->cancelEvent(fd, sylar::IOManager::WRITE);
        }, winfo);
    }

    int rt = iom->addEvent(fd, sylar::IOManager::WRITE);
    if(rt == 0) {
        sylar::Fiber::GetThis()->yield();
        if(timer) {
            timer->cancel();
        }
        if(tinfo->cancelled) {
            errno = tinfo->cancelled;
            return -1;
        }
    } else {
        if(timer) {
            timer->cancel();
```

```
        }
        SYLAR_LOG_ERROR(g_logger) << "connect addEvent(" << fd << ", WRITE) error";
    }

    int error = 0;
    socklen_t len = sizeof(int);
    if(-1 == getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &len)) {
        return -1;
    }
    if(!error) {
        return 0;
    } else {
        errno = error;
        return -1;
    }
}
```

上面的实现重点如下：

判断传入的fd是否为套接字，如果不为套接字，则调用系统的connect函数并返回。判断fd是否被显式设置为了非阻塞模式，如果是则调用系统的connect函数并返回。

调用系统的connect函数，由于套接字是非阻塞的，这里会直接返回EINPROGRESS错误。

如果超时参数有效，则添加一个条件定时器，在定时时间到后通过t->cancelled设置超时标志并触发一次WRITE事件。添加WRITE事件并yield，等待WRITE事件触发再往下执行。

等待超时或套接字可写，如果先超时，则条件变量winfo仍然有效，通过winfo来设置超时标志并触发WRITE事件，协程从yield点返回，返回之后通过超时标志设置errno并返回-1；如果在未超时之前套接字就可写了，那么直接取消定时器并返回成功。

取消定时器会导致定时器回调被强制执行一次，但这并不会导致问题，因为只有当前协程结束后，定时器回调才会在接下来被调度，由于定时器回调被执行时connect_with_timeout协程已经执行完了，所以理所当然地条件变量也被释放了，所以实际上定时器回调函数什么也没做。

写好了就完了吗?

项目扩展

本项目还有由于很多可以扩展的地方的，这里提供几个思路：

- 增加对共享栈的支持

  本项目的协程库是独立栈的形式，每个协程有自己的栈空间，优点是实现简单，协程切换开销相对较低，但是比较耗内存，我们可以增加本项目对共享栈的支持，让协程在运行的时候都使用同一个栈空间，每次协程切换时要把自身共享栈空间拷贝。回到该协程的时候，将之前保存的数据，重新拷贝到运行时栈中。具体的实现可以参考一下libco里的实现：

  ```
  /**
  * 一个共享栈的结构体，每个共享栈的内存所在
  ```

```
* 一个进程或者线程栈的地址，是从高位到低位安排数据的，所以stack_bp是栈底，stack_buffer是栈顶
*/
struct stStackMem_t
{
  stCoRoutine_t* occupy_co; // 当前正在使用该共享栈的协程
  int stack_size;   // 栈的大小
  char* stack_bp;   // stack_buffer + stack_size 栈底
  char* stack_buffer; // 栈的内容，也就是栈顶
};

/*
* 所有共享栈的结构体，这里的共享栈是个数组，每个元素分别是个共享栈
*/
struct stShareStack_t
{
  unsigned int alloc_idx; // 目前正在使用的那个共享栈的index
  int stack_size; // 共享栈的大小，这里的大小指的是一个stStackMem_t*的大小
  int count;   // 共享栈的个数，共享栈可以为多个，所以以下为共享栈的数组
  stStackMem_t** stack_array; //栈的内容，这里是个数组，元素是stStackMem_t*
};
```

下面两个函数用来创建共享栈以及分配内存：

```
/**
* 创建一个共享栈

* @param count 创建共享栈的个数
* @param stack_size 每个共享栈的大小
*/
stShareStack_t* co_alloc_sharestack(int count, int stack_size)
{
  stShareStack_t* share_stack = (stShareStack_t*)malloc(sizeof(stShareStack_t));
  share_stack->alloc_idx = 0;
  share_stack->stack_size = stack_size;

  //alloc stack array
  share_stack->count = count;
  stStackMem_t** stack_array = (stStackMem_t**)calloc(count,
sizeof(stStackMem_t*)); //见下文介绍
  for (int i = 0; i < count; i++)
  {
    stack_array[i] = co_alloc_stackmem(stack_size); //co_alloc_stackmem用于分配每个共
享栈内存，实现见下
  }
  share_stack->stack_array = stack_array;
  return share_stack;
}
```

```
/**
* 分配一个栈内存
* @param stack_size的大小
*/
stStackMem_t* co_alloc_stackmem(unsigned int stack_size)
{
  stStackMem_t* stack_mem = (stStackMem_t*)malloc(sizeof(stStackMem_t));
  stack_mem->occupy_co= NULL; //当前没有协程使用该共享栈
  stack_mem->stack_size = stack_size;
  stack_mem->stack_buffer = (char*)malloc(stack_size); //栈顶，低内存空间
  stack_mem->stack_bp = stack_mem->stack_buffer + stack_size; //栈底，高地址空间
  return stack_mem;
}
```

在协程切换时，会将共享栈的数据保存到上一个协程（occupy_co）的save_buffer中，将接下来要执行的协程（pending_co）save_buffer中的数据拷贝到共享栈中。

注意，在coctx_swap(&(curr->ctx),&(pending_co->ctx)); 函数前后，协程经历了切换，函数之前，coctx_swap会进入到pending_co的协程环境中运行，函数之后已经yield回此协程了，才会执行接下来的语句：

```
*
* 1. 将当前的运行上下文保存到curr中
* 2. 将当前的运行上下文替换为pending_co中的上下文

* @param curr
* @param pending_co
*/
void co_swap(stCoRoutine_t* curr, stCoRoutine_t* pending_co)
{
  stCoRoutineEnv_t* env = co_get_curr_thread_env();

  //get curr stack sp
  //c变量的作用是为了找到目前的栈顶，因为c变量是最后一个放入栈中的内容。
  char c;
  curr->stack_sp= &c;

  if (!pending_co->cIsShareStack)
  {
    // 如果没有采用共享栈，清空pending_co和occupy_co
    env->pending_co = NULL;
    env->occupy_co = NULL;
  }
  else
  {
    // 如果采用了共享栈
    env->pending_co = pending_co;
```

```
    //get last occupy co on the same stack mem
    // occupy_co指的是，和pending_co共同使用一个共享栈的协程
    // 把它取出来是为了先把occupy_co的内存保存起来
    stCoRoutine_t* occupy_co = pending_co->stack_mem->occupy_co;

    //set pending co to occupy thest stack mem;
    // 将该共享栈的占用者改为pending_co
    pending_co->stack_mem->occupy_co = pending_co;

    env->occupy_co = occupy_co;

    if (occupy_co && occupy_co != pending_co)
    {
      // 如果上一个使用协程不为空，则需要把它的栈内容保存起来，见下个函数。
      save_stack_buffer(occupy_co);
    }
  }

  // swap context
  coctx_swap(&(curr->ctx),&(pending_co->ctx) );

  // 上一步coctx_swap会进入到pending_co的协程环境中运行
  // 到这一步，已经yield回此协程了，才会执行下面的语句
  // 而yield回此协程之前，env->pending_co会被上一层协程设置为此协程
  // 因此可以顺利执行：将之前保存起来的栈内容，恢复到运行栈上

  //stack buffer may be overwrite, so get again;
  stCoRoutineEnv_t* curr_env = co_get_curr_thread_env();
  stCoRoutine_t* update_occupy_co =  curr_env->occupy_co;
  stCoRoutine_t* update_pending_co = curr_env->pending_co;

  // 将栈的内容恢复，如果不是共享栈的话，每个协程都有自己独立的栈空间，则不用恢复。
  if (update_occupy_co && update_pending_co && update_occupy_co !=
update_pending_co)
  {
    // resume stack buffer
    if (update_pending_co->save_buffer && update_pending_co->save_size > 0)
    {
      // 将之前保存起来的栈内容，恢复到运行栈上
      memcpy(update_pending_co->stack_sp, update_pending_co->save_buffer,
update_pending_co->save_size);
    }
  }
}

/**
* 将原本占用共享栈的协程的内存保存起来。
* @param occupy_co 原本占用共享栈的协程
*/
```

```
void save_stack_buffer(stCoRoutine_t* occupy_co)
{
  ///copy out
  stStackMem_t* stack_mem = occupy_co->stack_mem;
  // 计算出栈的大小
  int len = stack_mem->stack_bp - occupy_co->stack_sp;

  if (occupy_co->save_buffer)
  {
    free(occupy_co->save_buffer), occupy_co->save_buffer = NULL;
  }

  occupy_co->save_buffer = (char*)malloc(len); //malloc buf;
  occupy_co->save_size = len;

  // 将当前运行栈的内容，拷贝到save_buffer中
  memcpy(occupy_co->save_buffer, occupy_co->stack_sp, len);
}
```

- 支持协程嵌套

  本项目的协程只支持线程主协程与任务协程之间切换，无法在子协程"调用"新一层的子协程，也就是无法进行协程的嵌套调用，可以增加对这个功能的支持，具体实现也可以参考libco，每个线程都有一份 stCoRoutineEnv_t 对象（协程的运行环境），在线程第一次创建协程时被自动创建，同时也会创建主协程，并将指向主协程的指针放到 pCallStack[0] 里：最多支持128层的协程调用，

```
/*
* 线程所管理的协程的运行环境
* 一个线程只有一个这个属性
*/
struct stCoRoutineEnv_t
{
  // 这里实际上维护的是个调用栈
  // 最后一位是当前运行的协程，前一位是当前协程的父协程(即，resume该协程的协程)
  // 可以看出来，libco只能支持128层协程的嵌套调用。这个绝对够了
  stCoRoutine_t *pCallStack[ 128 ];

  int iCallStackSize; // 当前调用栈长度

  stCoEpoll_t *pEpoll;  //主要是epoll，作为协程的调度器

  //当采用共享栈模式时，用于共享栈数据的保存与回复
  stCoRoutine_t* pending_co;
  stCoRoutine_t* occupy_co;
};
```

引入了pCallStack数组以后，协程的切换就变成了下面这样：

```
void co_resume( stCoRoutine_t *co )
{
  stCoRoutineEnv_t *env = co->env;

  // 找到当前运行的协程，从数组最后一位拿出当前运行的协程，如果目前没有协程，那就是主线程
  stCoRoutine_t *lpCurrRoutine = env->pCallStack[ env->iCallStackSize - 1 ];

  if( !co->cStart )
  {
    // 如果当前协程还没有开始运行，为其构建上下文
    coctx_make( &co->ctx,(coctx_pfn_t)CoRoutineFunc,co, 0 );
    co->cStart = 1;
  }

  // 将指定协程放入线程的协程队列末尾
  env->pCallStack[ env->iCallStackSize++ ] = co;

  // 将当前运行的上下文保存到lpCurrRoutine中，同时将协程co的上下文替换进去
  // 执行完这一句，当前的运行环境就被替换为 co 了
  co_swap( lpCurrRoutine, co );
}
```

```
/*
*
* 主动将当前运行的协程挂起，并恢复到上一层的协程
*
* @param env 协程管理器
*/
void co_yield_env( stCoRoutineEnv_t *env )
{
  // 这里直接取了iCallStackSize - 2，那么万一icallstacksize < 2呢?
  // 所以这里实际上有个约束，就是co_yield之前必须先co_resume，这样就不会造成这个问题了

  // last就是 找到上次调用co_resume(curr)的协程
  stCoRoutine_t *last = env->pCallStack[ env->iCallStackSize - 2 ];

  // 当前栈
  stCoRoutine_t *curr = env->pCallStack[ env->iCallStackSize - 1 ];

  env->iCallStackSize--;

  // 把上下文当前的存储到curr中，并切换成last的上下文
  co_swap( curr, last);
}
```

- 增加对更复杂的协程调度算法的支持

  本项目的协程调度算法是最简单的先来先服务，我们可以参考操作系统对进程的调度算法，引入优先级，相应比、时间片等结构，实现更多相对复杂一些的调度算法以应对更多的需求场景。

协程+

我们写完WebServer后可以进行部署，提供http服务，但本项目写完了好像什么也干不了，本项目只是一个协程库。

我们可以此基础上用我们自己写的写协程库开发各种各样的服务，**比如用我们协程库项目创建一个IO协程调度器，将监听、建立连接等步骤都封装成协程添加到调度器队列里**，再加上http服务，这就是一个改编的WebServer项目。

这里还是十分建议将本协程库项目应用在我们其他项目中，尤其是一些需要提供网络服务的后端项目。

使用本项目作为基础的网络服务模块也证明了本项目的可用性，不至于让面试官认为我们的协程库只是一个玩具项目。

性能测试

写完了协程库，我们势必要对其进行一些测试来验证我们写的协程库是否有用，相比于其他的库有什么优势，分析出适用场景和性能瓶颈：

- “你是否对你写的协程库进行过测试？”
- “你这个协程库相比于其他库有什么优势？”

这些问题也是面试官经常会问到的，性能测试也是比较容易体现出我们思考和能力的环节，进行详细的性能测试也会成为项目的一大亮点。

我们的协程库主要用于网络服务器后端，所以开发不同场景的简易服务器，用其他的网络库来进行对比。

比如就可以用使用我们编写的协程库改编的WebServer和原始的WebServer分别进行压测，对比一下qps。

值得注意的是我们程序是单线程还是多线程，是计算密集型还是IO密集型，业务逻辑的复杂程度都会影响测试结果，我们最好在更重情况下进行多次测试，以验证我们协程库项目的优势和适用场景。

这里，我以本项目、libco协程库、libevent网络库以及原生epoll分别编写单线程回声服务器，使用ApacheBench测试工具分别进行压力测试。

ApacheBench的使用可以参考 https://www.jianshu.com/p/43d04d8baaf7，基本的命令如下：

```
ab -n 100 -c 10 http://test.com/
```

其中－n表示请求数，－c表示并发数。

下面给出三个服务程序的示例代码：

```
#include "sylar.h"
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

```cpp
#include <fcntl.h>
#include <iostream>
#include <stack>

static int sock_listen_fd = -1;

void test_accept();

void error(char* msg)
{
    perror(msg);
    printf("erreur...\n");
    exit(1);
}

void watch_io_read() {
    sylarr::IOManager::GetThis()->
        addEvent(sock_listen_fd, sylarr::IOManager::READ, test_accept);
}

void test_accept() {
    struct sockaddr_in addr; //maybe sockaddr_un;
  memset( &addr,0,sizeof(addr) );
  socklen_t len = sizeof(addr);
    int fd = accept(sock_listen_fd, (struct sockaddr*)&addr, &len);
    if (fd < 0) {
        std::cout << "fd = "<< fd << "accept false" << std::endl;
    } else {
        fcntl(fd, F_SETFL, O_NONBLOCK);
        sylarr::IOManager::GetThis()->
            addEvent(fd, sylarr::IOManager::READ, [fd](){
                char buffer[1024];
                memset(buffer, 0, sizeof(buffer));
                while (true) {
                    int ret = recv(fd, buffer, sizeof(buffer), 0);
                    if (ret > 0) {
                        ret = send(fd, buffer, ret, 0);
                    }
                    if (ret <= 0) {
                        if (errno == EAGAIN) continue;
                        close(fd);
                        break;
                    }
                }

        });

    }
    sylarr::IOManager::GetThis()->schedule(watch_io_read);
```

```
}

void test_iomanager() {
    int portno = 8080;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    // setup socket
    sock_listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_listen_fd < 0) {
        error("Error creating socket..\n");
    }
    int yes = 1;
    // lose the pesky "address already in use" error message
    setsockopt(sock_listen_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));

    memset((char *)&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(portno);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // bind socket and listen for connections
    if (bind(sock_listen_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
        error("Error binding socket..\n");

    if (listen(sock_listen_fd, 1024) < 0) {
        error("Error listening..\n");
    }
    printf("epoll echo server listening for connections on port: %d\n", portno);

    fcntl(sock_listen_fd, F_SETFL, O_NONBLOCK);

    sylarr::IOManager iom;

    iom.addEvent(sock_listen_fd, sylarr::IOManager::READ, test_accept);
}

int main(int argc, char *argv[]) {
    test_iomanager();

    return 0;
}
```

```
#include "co_routine.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/time.h>
```

```
#include <stack>

#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

#ifdef __FreeBSD__
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>
#endif

using namespace std;
struct task_t
{
  stCoRoutine_t *co;
  int fd;
};

static stack<task_t*> g_readwrite;
static int g_listen_fd = -1;
static int SetNonBlock(int iSock)
{
    int iFlags;

    iFlags = fcntl(iSock, F_GETFL, 0);
    iFlags |= O_NONBLOCK;
    iFlags |= O_NDELAY;
    int ret = fcntl(iSock, F_SETFL, iFlags);
    return ret;
}

static void *readwrite_routine( void *arg )
{
  co_enable_hook_sys();

  task_t *co = (task_t*)arg;
  char buf[ 1024 * 16 ];
  for(;;)
  {
    if( -1 == co->fd )
    {
      // push进去
      g_readwrite.push( co );
```

```
      // 切出
      co_yield_ct();
      continue;
    }

    int fd = co->fd;
    co->fd = -1;

    for(;;)
    {
      // 将该fd的可读事件，注册到epoll中
      // co_accept里面libco并没有将其设置为 O_NONBLOCK
      // 是用户主动设置的 O_NONBLOCK
      // 所以read函数不走hook逻辑，需要自行进行poll切出
      struct pollfd pf = { 0 };
      pf.fd = fd;
      pf.events = (POLLIN|POLLERR|POLLHUP);

      co_poll( co_get_epoll_ct(),&pf,1,1000);

      // 当超时或者可读事件到达时，进行read。所以read不一定成功，有可能是超时造成的
      int ret = read( fd,buf,sizeof(buf) );

      // 读多少就写多少
      if( ret > 0 )
      {
        ret = write( fd,buf,ret );
      }

      if( ret <= 0 )
      {
        // accept_routine->SetNonBlock(fd) cause EAGAIN, we should continue
        if (errno == EAGAIN)
          continue;
        close( fd );
        break;
      }
    }

  }
  return 0;
}
int co_accept(int fd, struct sockaddr *addr, socklen_t *len );
static void *accept_routine( void * )
{
  co_enable_hook_sys();
  printf("accept_routine\n");
  fflush(stdout);
  for(;;)
```

```
  {
    //printf("pid %ld g_readwrite.size %ld\n",getpid(),g_readwrite.size());
    if( g_readwrite.empty() )
    {
      printf("empty\n"); //sleep
      struct pollfd pf = { 0 };
      pf.fd = -1;

      // sleep 1秒，等待有空余的协程
      poll( &pf,1,1000);

      continue;
    }

    struct sockaddr_in addr; //maybe sockaddr_un;
    memset( &addr,0,sizeof(addr) );
    socklen_t len = sizeof(addr);

    // accept
    int fd = co_accept(g_listen_fd, (struct sockaddr *)&addr, &len);
    if( fd < 0 )
    {
      // 意思是，如果accept失败了，没办法，暂时切出去
      struct pollfd pf = { 0 };
      pf.fd = g_listen_fd;
      pf.events = (POLLIN|POLLERR|POLLHUP);
      co_poll( co_get_epoll_ct(),&pf,1,1000 );

      continue;
    }

    if( g_readwrite.empty() )
    {
      close( fd );
      continue;
    }

    // 这里需要手动将其变成非阻塞的
    SetNonBlock( fd );

    task_t *co = g_readwrite.top();
    co->fd = fd;
    g_readwrite.pop();
    co_resume( co->co );
  }
  return 0;
}
```

```
static void SetAddr(const char *pszIP,const unsigned short shPort,struct sockaddr_in
&addr)
{
  bzero(&addr,sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(shPort);
  int nIP = 0;
  if( !pszIP || '\0' == *pszIP
      || 0 == strcmp(pszIP,"0") || 0 == strcmp(pszIP,"0.0.0.0")
    || 0 == strcmp(pszIP,"*")
    )
  {
    nIP = htonl(INADDR_ANY);
  }
  else
  {
    nIP = inet_addr(pszIP);
  }
  addr.sin_addr.s_addr = nIP;

}

static int CreateTcpSocket(const unsigned short shPort /* = 0 */,const char *pszIP /* =
"*" */,bool bReuse /* = false */)
{
  int fd = socket(AF_INET,SOCK_STREAM, IPPROTO_TCP);
  if( fd >= 0 )
  {
    if(shPort != 0)
    {
      if(bReuse)
      {
        int nReuseAddr = 1;
        setsockopt(fd,SOL_SOCKET,SO_REUSEADDR,&nReuseAddr,sizeof(nReuseAddr));
      }
      struct sockaddr_in addr ;
      SetAddr(pszIP,shPort,addr);
      int ret = bind(fd,(struct sockaddr*)&addr,sizeof(addr));
      if( ret != 0)
      {
        close(fd);
        return -1;
      }
    }
  }
  return fd;
}
```

```
int main(int argc,char *argv[])
{
  if(argc<5){
    printf("Usage:\n"
               "example_echosvr [IP] [PORT] [TASK_COUNT] [PROCESS_COUNT]\n"
               "example_echosvr [IP] [PORT] [TASK_COUNT] [PROCESS_COUNT] -d   #
daemonize mode\n");
    return -1;
  }
  const char *ip = argv[1];
  int port = atoi( argv[2] );
  int cnt = atoi( argv[3] ); // task_count 协程数
  int proccnt = atoi( argv[4] ); // 进程数
  bool deamonize = argc >= 6 && strcmp(argv[5], "-d") == 0;

  g_listen_fd = CreateTcpSocket( port,ip,true );
  listen( g_listen_fd,1024 );
  if(g_listen_fd==-1){
    printf("Port %d is in use\n", port);
    return -1;
  }
  printf("listen %d %s:%d\n",g_listen_fd,ip,port);

  SetNonBlock( g_listen_fd );

  for(int k=0;k<proccnt;k++)
  {

    pid_t pid = fork();
    if( pid > 0 )
    {
      continue;
    }
    else if( pid < 0 )
    {
      break;
    }
    for(int i=0;i<cnt;i++)
    {
      task_t * task = (task_t*)calloc( 1,sizeof(task_t) );
      task->fd = -1;

      // 创建一个协程
      co_create( &(task->co),NULL,readwrite_routine,task );

      // 启动协程
      co_resume( task->co );
    }
```

```
    // 启动listen协程
    stCoRoutine_t *accept_co = NULL;
    co_create( &accept_co,NULL,accept_routine,0 );
    // 启动协程
    co_resume( accept_co );

    // 启动事件循环
    co_eventloop( co_get_epoll_ct(),0,0 );

    exit(0);
  }
  if(!deamonize) wait(NULL);
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <event2/event.h>

#define PORT 8888

// 回声处理函数
void echo_read_cb(evutil_socket_t fd, short events, void *arg) {
    char buf[1024];
    int len;
    len = recv(fd, buf, sizeof(buf)-1, 0);
    if (len <= 0) {
        // 发生错误或连接关闭，关闭连接并释放事件资源
        close(fd);
        event_free((struct event*)arg);
        return;
    }

    buf[len] = '\0';
    printf("接收到消息: %s\n", buf);

    // 发送回声消息给客户端
    send(fd, buf, len, 0);
}

// 接受连接回调函数
void accept_conn_cb(evutil_socket_t listener, short event, void *arg) {
    struct event_base *base = (struct event_base*)arg;
    struct sockaddr_storage ss;
    socklen_t slen = sizeof(ss);
    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
```

```
    if (fd < 0) {
        perror("accept");
    } else if (fd > FD_SETSIZE) {
        close(fd);
    } else {
        // 创建一个新的事件结构体
        struct event *ev = event_new(NULL, -1, 0, NULL, NULL);
        // 将新的事件添加到事件循环中
        event_assign(ev, base, fd, EV_READ|EV_PERSIST, echo_read_cb, (void*)ev);
        event_add(ev, NULL);
    }
}

int main() {
    struct event_base *base;
    struct event listener;

    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(PORT);

    // 初始化Libevent库
    base = event_base_new();

    // 创建一个监听事件
    listener = *event_new(base, -1, EV_READ|EV_PERSIST, accept_conn_cb, (void*)base);
    // 将监听事件绑定到指定的地址和端口
    if (event_add(&listener, NULL) == -1) {
        perror("event_add");
        return -1;
    }

    // 开始事件循环
    event_base_dispatch(base);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/epoll.h>

#define MAX_EVENTS 10
```

```c
#define PORT 8888

int main() {
    int listen_fd, conn_fd, epoll_fd, event_count;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    struct epoll_event events[MAX_EVENTS], event;

    // 创建监听套接字
    if ((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return -1;
    }

    // 设置服务器地址和端口
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // 绑定监听套接字到服务器地址和端口
    if (bind(listen_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        return -1;
    }

    // 监听连接
    if (listen(listen_fd, 5) == -1) {
        perror("listen");
        return -1;
    }

    // 创建 epoll 实例
    if ((epoll_fd = epoll_create1(0)) == -1) {
        perror("epoll_create1");
        return -1;
    }

    // 添加监听套接字到 epoll 实例中
    event.events = EPOLLIN;
    event.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &event) == -1) {
        perror("epoll_ctl");
        return -1;
    }

    while (1) {
        // 等待事件发生
        event_count = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
```

```
        if (event_count == -1) {
            perror("epoll_wait");
            return -1;
        }

        // 处理事件
        for (int i = 0; i < event_count; i++) {
            if (events[i].data.fd == listen_fd) {
                // 有新连接到达
                conn_fd = accept(listen_fd, (struct sockaddr*)&client_addr, &addr_len);
                if (conn_fd == -1) {
                    perror("accept");
                    continue;
                }

                // 将新连接的套接字添加到 epoll 实例中
                event.events = EPOLLIN;
                event.data.fd = conn_fd;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn_fd, &event) == -1) {
                    perror("epoll_ctl");
                    return -1;
                }
            } else {
                // 有数据可读
                char buf[1024];
                int len = read(events[i].data.fd, buf, sizeof(buf) - 1);
                if (len <= 0) {
                    // 发生错误或连接关闭，关闭连接
                    close(events[i].data.fd);
                } else {
                    buf[len] = '\0';
                    printf("接收到消息：%s\n", buf);
                    // 发送回声消息给客户端
                    write(events[i].data.fd, buf, len);
                }
            }
        }
    }

    // 关闭监听套接字和 epoll 实例
    close(listen_fd);
    close(epoll_fd);

    return 0;
}
```

其实，只使用qps来衡量我么项目编写的好坏是不完整的，除了关注最大qps之外，还需要关注服务器的负载负载情况，CPU、内存等，分析我们的协程库是否耗用了太多的系统资源，是否有效的利用到了系统资源。这里具体的分析过程和结果不再展开，大家可自行探索。

其他多线程、更复杂的业务逻辑可自行编写测试，每个人的测试结果可能会由于环境、项目的具体实现、测试方法的不同而存在差异，这里给出我的测试结论：单线程下本项目相比于于原生epoll几乎没有性能损失，在大流量、多线程、IO密集条件，使用本项目编写的网络服务器相比于其他的网络服务库具有明显的的性能优势，经分析本项目的性能瓶颈存在于服务器运行时协程的创建、切换、析构，同时协程独立栈的设计也使得本项目占用更多的内存。

为了解决本项目的性能瓶颈，这里提供一种优化思路：服务器使用"协程池"方法，引入对共享栈的支持。

如何应对面试?

简历怎么写?

简历的写法卡哥星球有详细的说明，这里先给出我秋招时这个项目的简历：

c++服务器框架-协程库 **独立开发** 2022.04 - 2022.09

项目背景： 本项目为在完成webserver等基础项目后的进阶项目，参考开源项目sylar，用于巩固基础知识，锻炼实操能力。

项目描述： 本项目是在Linux环境下使用C++从零开发的部分服务器框架，**主要实现了协程库的编写，基于ucontext_t实现了协程类、结合epoll和定时器实现了N-M协程调度器，支持IO事件、定时器事件的回调，基于协程调度，对常见系统API进行hook封装实现异步。** 同时还完成了日志功能、配置功能、多线程和锁的封装等工作。

- **线程封装：** 封装了pthread，互斥量，信号量，读写锁、自旋锁，实现范围锁。
- **协程实现：** **基于ucontext_t实现了非对称协程，**设计三种协程状态，使子协程可以和线程主协程相互切换。
- **定时器：** **基于时间堆实现了定时器功能，**支持定时事件的添加、删除、更新。
- **协程调度：** **实现了N-M协程调度器，**支持main函数线程参与调度，在基本的协程调度器基础上结合epoll和定时器实现了IO协程调度，**支持IO事件和定时事件的注册和回调。**
- **Hook：** 基于IO协程调度器**对sleep系列、SocketIO系列、fd操作系列系统调用进行hook封装实现阻塞调用的异步。**
- **测试：** 使用原生epoll、libevent和本项目分别编写单线程简易服务器，利用ApacheBench进行压力测试，消息条数1000000，并发连接数量1000，RPS分别为27082.70、25432.28、26040.80，性能并无太大差异，说明**单线程下本项目几乎没有性能损失，又多次在不同条件下测试，证明本项目在大流量、多线程、IO密集条件下工作具有一定的性能优势。**

迫于简历篇幅，还缺少项目难点、个人收获部分内容，这里给出参考：

项目难点：

- 基于epoll和定时器实现多线程IO协程调度器，支持对定时任务协程和IO任务协程的调度，且支持主线程（创建调度器的线程）参与调度。
- 对seelp、IO等阻塞系统调用进行hook封装，在函数内部进行协程切换，将阻塞系统调用改造为非阻塞。

个人收获：

- 深入了解了协程技术，熟悉了共享栈、对称/非对称协程等概念
- 了解了主要协程库的优缺点以及适用场景，对进程、线程、协程的区别有了更深入的了解
- 熟练掌握了Linux网络编程、系统编程接口，对IO多路复用、事件驱动模型有了一定的了解。

这里的示例仅供参考，还是要根据自己的理解来写，建议根据上面介绍的内容，把项目做出亮点，在简历上重点体现。

面试会问哪些问题呢?

下面都是我在秋招中被问到的真实面试题，有些场次没有记录，但大部分问题基本都是协程的基础知识和项目里的一些细节，本文介绍里基本都覆盖到了，还是要认真做过项目，深入学习一下协程，有自己的理解，才能在面试的时候面对各种问题从容不迫。

- **【所有公司】简要介绍一下协程库这个项目。**

  参考简历。

- **【大部分公司】对比一下进程、线程和协程。**

进程是操作系统进行资源分配的基本单位，每个进程都有自己的独立内存空间。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

线程又叫做轻量级进程，是进程的一个实体，是处理器任务调度和执行的基本单位位。它是比进程更小的能独立运行的基本单位。线程只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

协程，又称微线程，是一种用户态的轻量级线程，协程的调度完全由用户控制（也就是在用户态执行）。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到线程的堆区，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

- **【蔚来】协程的缺点是什么?**

  难以调试、占用更多内存，学习成本相对较高等，最明显的缺点是：**无法利用多核资源**。线程才是系统调度的基本单位，单线程下的多协程本质上还是串行执行的，只能用到单核计算资源，所以协程往往要与多线程、多进程一起使用。

- **【阿里巴巴本地生活、京东、快手】你觉得协程有什么用?**

  提高资源利用率，提高程序并发性能。协程允许开发者编写异步代码，实现非阻塞的并发操作，通过在适当的时候挂起和恢复协程，可以有效地管理多个任务的执行，提高程序的并发性能。与线程相比，协程是轻量级的，它们的创建和上下文切换开销较小，可以同时执行大量的协程，而不会导致系统负载过重，可以在单线程下实现异步，使程序不存在阻塞阶段，充分利用cpu资源。

  简化异步编程逻辑。使用协程可以简化并发编程的复杂性，通过使用适当的协程库或语言特性，可以避免显式的线程同步、锁和互斥量等并发编程的常见问题，用同步的思想就可以编写成异步的程序。

- **【滴滴】衡量一个协程库性能的标准有哪些?**

  响应时间：衡量在给定负载下协程调度的快慢。较低的响应时间表示更高的性能。
  吞吐量：指在单位时间内执行的任务数量。较高的吞吐量表示更高的性能。
  并发能力：指同时处理的协程数量。更高的并发能力表示更好的性能。
  上下文切换开销：在协程库中，上下文切换是指从一个协程切换到另一个协程的操作。较低的上下文切换开销表示更高的性能，因为过多的上下文切换会浪费时间和资源。

  资源利用率：一个好的协程库应该有效地利用系统资源，例如处理器、内存和网络等。协程库在资源利用方面的优化程度可以影响性能。

- **【滴滴、华为】你是怎样做的性能测试?**

  参考上文性能测试部分。

- **【京东】项目中有没有加锁的场景，加的是什么锁?**

  项目中有很多地方需要加锁，比如在多线程的协程调度器在从协程的任务队列取协程任务执行的时候需要加互斥锁，协程调度器全局变量访问时加读写锁。

- **【滴滴】了解Go语言的协程吗?**

  Go语言最大的特色就是从语言层面支持并发（Goroutine），Goroutine是Go中最基本的执行单元。事实上每一个Go程序至少有一个Goroutine：main Goroutine。Go 程序从 main 包的 main() 函数开始，在程序启动时，Go 程序就会为 main() 函数创建一个默认的 goroutine。更多内容参考https://juejin.cn/post/7044741465930465311

- **【京东、快手】了解brpc吗，你觉得里面用到协程了吗？（因为有在百度实习的经历，所以被问到这个问题）**

是的，brpc引入m:n的线程模型，固定的内核线程调度运行大量的bthread以避免内核线程上下文切换带来的开销。bthread类似协程，即用户态线程，bthread的切换不会陷入内核，不会进行一系列内存同步等耗时操作，因此bthread的切换在100-200ns，相比内核线程的微秒级别有着数量级的提升。

- **【科大讯飞、中望软件】你这个项目遇到的最大的困难是什么?**

  参考简历项目难点的描述。

- **【华为】服务器运行时每建立一个用户连接就要创建一个协程，不会影响性能吗?**

  是的，会存在这个问题，在承受高并发时，大量的协程的创建、析构都会消耗较大的系统资源，可以采用"协程池"的方法解决，提前创建指定数量的协程，有新的任务执行时，选择一个协程进行任务函数绑定，任务完成将协程重置返回到协程中。

- **【科大讯飞】为什要有空闲协程，和主协程合成一个不行吗?**

  在IO协程调度器空闲协程里要进行epoll监听，添加协程任务，最基本的协程调度器包含idle协程，IO协程调度器对其进行了重写，主协程只进行任务调度，idle只进行任务添加，降低了不同功能之间的耦合，便于后续扩展维护。

- **【滴滴】你的协程是怎样调度的?**

  参考协程调度部分。

- **【阿里云、华为】了解c++的协程吗?**

  C++的协程是从C++20标准开始引入的一种语言特性，旨在提供一种方便、高效处理异步任务和并发编程的机制。C++的协程通过关键字co_await和co_yield来实现，结合协程类型和协程函数来定义和使用协程。更多细节可参考https://www.bennyhuo.com/2022/03/09/cpp-coroutines-01-intro/