

从零开始重写sylar C++高性能分布式服务器框架

由 zhongluqiang创建, 最后修改于12月 10, 2021

项目地址: <https://github.com/zhongluqiang/sylar-from-scratch>

C++服务器框架, 包括日志模块, 配置模块, 线程模块, 协程模块, 协程调度模块, IO协程调度模块, hook模块, socket模块, bytearray序列化, tcpserver模块, http模块, websocket模块, https模块等。

- [项目依赖](#)
- [日志模块](#)
- [环境变量模块](#)
- [配置模块](#)
- [线程模块](#)
- [协程模块](#)
- [协程调度模块](#)
- [IO协程调度模块](#)
- [定时器模块](#)
- [hook模块](#)
- [Address模块](#)
- [Socket模块](#)
- [ByteArray类](#)
- [Stream模块](#)
- [TcpServer类](#)
- [HTTP模块](#)
- [守护进程](#)

无标签

浙ICP备2021003588号 ·

日志模块

由 zhongluqiang 创建, 最后修改于 6月 29, 2021

日志模块概述

用于格式化输出程序日志, 方便从日志中定位程序运行过程中出现的问题。这里的日志除了日志内容本身之外, 还应该包括文件名/行号, 时间戳, 线程/协程号, 模块名称, 日志级别等额外信息, 甚至在打印致命的日志时, 还应该附加程序的栈回溯信息, 以便于分析和排查问题。

从设计上看, 一个完整的日志模块应该具备以下功能:

1. 区分不同的级别, 比如常的DEBUG/INFO/WARN/ERROR等级别。日志模块可以通过指定级别实现只输出某个级别的日志, 这样可以灵活开关一些不重要的日志输出, 比如程序在调试阶段可以设置一个较低的级别, 以便看到更多的调度日志信息, 程序发布之后可以设置一个较高的级别, 以减少日志输出, 提高性能。
2. 区分不同的输出地。不同的日志可以输出到不同的位置, 比如可以输出到标准输出, 输出到文件, 输出到syslog, 输出到网络上的日志服务器等, 甚至同一条日志可以同时输出到多个输出地。
3. 区分不同的类别。日志可以分类并命名, 一个程序的各个模块可以使用不同的名称来输出日志, 这样可以很方便地判断出当前日志是哪个程序模块输出的。
4. 日志格式可灵活配置。可以按需指定每条日志是否包含文件名/行号、时间戳、线程/协程号、日志级别、启动时间等内容。
5. 可通过配置文件的方式配置以上功能。

log4cpp示例

这里先通过一个典型的C++日志框架log4cpp来分析以上几点的运行。关于log4cpp的介绍和使用可参考官网: <http://log4cpp.sourceforge.net>, 下面是一个它的示例:

```
#include <log4cpp/Category.hh>
#include <log4cpp/Appender.hh>
#include <log4cpp/OstreamAppender.hh>
#include <log4cpp/FileAppender.hh>
#include <log4cpp/Layout.hh>
#include <log4cpp/PatternLayout.hh>
#include <log4cpp/PropertyConfigurator.hh>

int main() {
    // 日志类别
    log4cpp::Category &root_logger = log4cpp::Category::getRoot(); // 获取root日志器, root日志器名称默认为空
    log4cpp::Category &file_logger = log4cpp::Category::getInstance("file_logger"); // 获取指定名称为file_logger的日志器

    // 日志输出地
    log4cpp::Appender *coutAppender = new log4cpp::OstreamAppender("cout", &std::cout); // 输出到终端
    log4cpp::FileAppender *fileAppender = new log4cpp::FileAppender("file", "./log.txt"); // 输出到文件

    // 日志格式
    log4cpp::PatternLayout *coutPattern = new log4cpp::PatternLayout();
    coutPattern->setConversionPattern("[%d{%Y-%m-%d %H:%M:%S:%1}] %p %m%n"); // 格式: [年:月:日 时:分:秒:毫秒] 日志级别 日志内容
    log4cpp::PatternLayout *filePattern = new log4cpp::PatternLayout();
    filePattern->setConversionPattern("[%d{%Y-%m-%d %H:%M:%S:%1}] %R %c %p %m%n"); // 格式: [年:月:日 时:分:秒:毫秒] UTC秒数 日志器名称 日志内容

    // 绑定日志器和appender, 设置日志器的输出级别
    coutAppender->setLayout(coutPattern);
    root_logger.addAppender(coutAppender);
    root_logger.setPriority(log4cpp::Priority::INFO);
    fileAppender->setLayout(filePattern);
    file_logger.addAppender(fileAppender);
    file_logger.setPriority(log4cpp::Priority::ERROR);

    // 支持c风格的日志打印和流式的日志打印
    root_logger.debug("debug msg");
    root_logger.debugStream() << "debug msg by stream";
    root_logger.info("info msg");
    root_logger.infoStream() << "info msg by stream";
    root_logger.error("error msg");
    root_logger.errorStream() << "err msg by stream";

    file_logger.debug("debug msg");
    file_logger.debugStream() << "debug msg by stream";
    file_logger.info("info msg");
    file_logger.infoStream() << "info msg by stream";
    file_logger.error("error msg");
    file_logger.errorStream() << "err msg by stream";

    return 0;
}
```

上面的代码在终端输出如下：

```
[2021-06-25 09:33:59:734] INFO info msg
[2021-06-25 09:33:59:734] INFO info msg by stream
[2021-06-25 09:33:59:734] ERROR error msg
[2021-06-25 09:33:59:734] ERROR err msg by stream
[2021-06-25 09:33:59:734] ERROR error msg
[2021-06-25 09:33:59:734] ERROR err msg by stream
```

日志文件内容如下：

log.txt

```
[2021-06-25 09:33:59:734] 1624584839 file_logger ERROR error msg
[2021-06-25 09:33:59:734] 1624584839 file_logger ERROR err msg by stream
```

对于log4cpp总结如下：

1. Category对应日志类别，使用不同的名称来区别日志器，使用 log4cpp::Category::getInstance() 获取指定名称的日志器实例，如果两个日志器名称相同，那么对应同一个日志器实例。
2. 使用Category的setPriority()方法设置日志器的日志级别，日志级别使用log4cpp::Priority枚举值来表示，一共有FATAL/ALERT/CRIT/ERROR/WARN/NOTICE/INFO/DEBUG/NOTSET几个等级。
3. 使用Appender来表示日志输出地，Appender可以细分为OstreamAppender和FileAppender等不同类型。一个Category可以有多个Appender，这样一条日志就可以输出到多个地方，通过addAppender()方法为Category新增Appender。
4. 使用PatternLayout来表示日志的格式，格式通过模板字符串来指定，比如%d表示时间（后面可用{}指定具体的时间格式），%R表示UTC秒数，%c表示日志器名称，%p表示日志级别，%m表示日志消息等。
5. PatternLayout和Appender绑定，Priority和Category绑定，一条日志经过Category判断级别通过后由Appender输出，Appender输出的格式由PatternLayout指定。

除了在程序中指定日志配置，log4cpp也支持通过配置文件指定，这点参考[官方示例](#)即可。

sylar日志模块设计

下面开始sylar的日志模块设计。

首先是日志级别，这个参考log4cpp即可，一共定义以下几个级别：

```
enum Level {
    /// 致命情况，系统不可用
    FATAL = 0,
    /// 高优先级情况，例如数据库系统崩溃
    ALERT = 100,
    /// 严重错误，例如硬盘错误
    CRIT = 200,
    /// 错误
    ERROR = 300,
    /// 警告
    WARN = 400,
    /// 正常但值得注意
    NOTICE = 500,
    /// 一般信息
    INFO = 600,
    /// 调试信息
    DEBUG = 700,
    /// 未设置
    NOTSET = 800,
};
```

接下来是几个关键的类：

```
class LogFormatter;
class LogAppender;
class Logger;
class LogEvent;
class LogEventWrap;
class LogManager;
```

关于这几个类的设计如下：

LogFormatter：日志格式器，与log4cpp的PatternLayout对应，用于格式化一个日志事件。该类构建时可以指定pattern，表示如何进行格式化。提供format方法，用于将日志事件格式化成字符串。

LogAppender：日志输出器，用于将一个日志事件输出到对应的输出地。该类内部包含一个**LogFormatter**成员和一个**log**方法，日志事件先经过**LogFormatter**格式化后再输出到对应的输出地。从这个类可以派生出不同的**Appender**类型，比如**StdoutLogAppender**和**FileLogAppender**，分别表示输出到终端和文件。

Logger：日志器，负责进行日志输出。一个**Logger**包含多个**LogAppender**和一个日志级别，提供**log**方法，传入日志事件，判断该日志事件的级别高于日志器本身的级别之后调用**LogAppender**将日志进行输出，否则该日志被抛弃。

LogEvent：日志事件，用于记录日志现场，比如该日志的级别，文件名/行号，日志消息，线程/协程号，所属日志器名称等。

LogEventWrap：日志事件包装类，其实就是将日志事件和日志器包装到一起，因为一条日志只会在一个日志器上进行输出。将日志事件和日志器包装到一起后，方便通过宏定义来简化日志模块的使用。另外，**LogEventWrap**还负责在构建时指定日志事件和日志器，在析构时调用日志器的**log**方法将日志事件进行输出。

LogManager：日志器管理类，单例模式，用于统一管理所有的日志器，提供日志器的创建与获取方法。**LogManager**自带一个root Logger，用于为日志模块提供一个初始可用的日志器。

至此，日志模块的设计就基本结束了，总结一下日志模块的工作流程：

1. 初始化**LogFormatter**, **LogAppender**, **Logger**。
2. 通过宏定义提供流式风格和格式化风格的日志接口。每次写日志时，通过宏自动生成对应的日志事件**LogEvent**，并且将日志事件和日志器**Logger**包装到一起，生成一个**LogEventWrap**对象。
3. 日志接口执行结束后，**LogEventWrap**对象析构，在析构函数里调用**Logger**的**log**方法将日志事件进行输出。

sylar日志模块关键实现

LogEvent

日志事件，用于记录日志现场，具体包括以下内容：

- 日志内容
- 日志器名称
- 日志级别
- 文件名，对应 `_FILE_` 宏
- 行号，对应 `LINE` 宏
- 程序运行时间，通过 `sylar::GetElapsedMS()` 获取
- 线程ID
- 协程ID
- UTC时间戳，对应 `time(0)`
- 线程名称

日志事件的构造需要通过宏来简化，否则，每次生成一个日志事件时都要对上面这些内容进行赋值，够麻烦的。

LogFormatter

日志格式器，用于格式化一个日志事件，将其转化成一串字符串。由于一个日志事件包括了很多的内容（参考上一节，这里将每个内容称为一个item），但实际上用户并不希望每次输出日志时都将这些items全部进行输出，而是希望可以自由地选择要输出的item。并且，用户还可能需要在每条日志里增加一些指定的字符，比如在文件名和行号之间加上一个冒号的情况。为了实现这项功能，**LogFormatter**使用了一个模板字符串来指定格式化的方式，这个模板字符串是一串像下面这样的字符：

```
"%d{%-m-%d %H:%M:%S} [%rms] %T%t%N%T%F%T[%p] %T[%c] %T%F:%l%T%m%n"
```

模板字符串由普通字符和转义字符构成，转义字符以%开头，比如%*m*, %*p*等。除了转义字符，剩下的全部都是普通字符，包括空格。

LogFormatter根据模板字符串来格式化日志事件。首先，在构造**LogFormatter**对象时会指定一串模板字符，**LogFormatter**会首先解析该模板字符串，将其中的转义字符和普通字符解析出来。然后，在格式化日志事件时，根据模板字符串，将其中的转义字符替换成日志事件的具体内容，普通字符保持不变。

当前实现支持以下转义字符：

% <i>m</i>	消息
% <i>p</i>	日志级别
% <i>c</i>	日志器名称
% <i>d</i>	日期时间，后面可跟一对括号指定时间格式，比如 <code>%d{%-m-%d %H:%M:%S}</code> ，这里的格式字符与C语言 <code>strftime</code> 一致
% <i>r</i>	该日志器创建后的累计运行毫秒数
% <i>f</i>	文件名
% <i>l</i>	行号
% <i>t</i>	线程id
% <i>F</i>	协程id
% <i>N</i>	线程名称
% <i>%</i>	百分号
% <i>T</i>	制表符
% <i>n</i>	换行

假设一个**LogFormatter**的模板字符串为 `[%c] [%p] %t %F %f:%l %m%n`，那么它输出的日志可能是像下面这样的一行字符串：

```
[root] [info] 1000 0 main.cpp:12 hello world
```

LogFormatter的实现重点是解析模板字符串，提取出其中的转义字符和普通字符，这里可以参考<https://github.com/zongluoqiang/sylar-from-scratch/blob/main/sylar/log.cpp>，其中的LogFormatter::init()提供了一个简单的基于状态机的解析方案，支持解析上面列举的转义字符，并且支持将连续的普通字符合并成一个字符串。

LogAppender

日志输出器，用于输出一个日志事件。这是一个虚类，可以派生出不同的具体实现，比如往输出到终端的StdoutLogAppender，以及输出到文件的FileLogAppender。

LogAppender的实现包含了一个用户指定的LogFormatter和一个默认的LogFormatter，以及log方法，不同类型的Appender通过重载log方法来实现往不同的目的地进行输出，这部分直接阅读源码即可，不难理解。

Logger

日志器，用于输出日志。这个类是直接与用户进行交互的类，提供log方法用于输出日志事件。

Logger的实现包含了日志级别，日志器名称，创建时间，以及一个LogAppender数组，日志事件由log方法输出，log方法首先判断日志级别是否达到本Logger的级别要求，是则将日志传给各个LogAppender进行输出，否则抛弃这条日志。

LogEventWrap

日志事件包装类，在日志现场构造，包装了日志器和日志事件两个对象，在日志记录结束后，LogEventWrap析构时，调用日志器的log方法输出日志事件。

LogManager

日志器管理类，单例模式，用于统一管理全部的日志器，提供getLogger()方法用于创建/获取日志器。内部维护一个名称到日志器的map，当获取的日志器存在时，直接返回对应的日志器指针，否则创建对应日志器并返回。

工具宏

sylar定义了一系列工具宏用于简化编码以及实现流式风格的日志输出和格式化风格的日志输出，下面以流式风格的实现为例分析一下宏的设计：

```
/**
 * @brief 使用流式方式将日志级别level的日志写入到logger
 * @details 构造一个LogEventWrap对象，包裹包含日志器和日志事件，在对象析构时调用日志器写日志事件
 */
#define SYLAR_LOG_LEVEL(logger , level) \
    if(level <= logger->getLevel()) \
        sylar::LogEventWrap(logger, sylar::LogEvent::ptr(new sylar::LogEvent(logger->getName(), \
            level, __FILE__, __LINE__, sylar::GetElapsedMS() - logger->getCreateTime(), \
            sylar::GetThreadId(), sylar::GetFiberId(), time(0), sylar::GetThreadName()))).getLogEvent()->getSS()

#define SYLAR_LOG_FATAL(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::FATAL)

#define SYLAR_LOG_ALERT(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::ALERT)

#define SYLAR_LOG_CRIT(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::CRIT)

#define SYLAR_LOG_ERROR(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::ERROR)

#define SYLAR_LOG_WARN(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::WARN)

#define SYLAR_LOG_NOTICE(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::NOTICE)

#define SYLAR_LOG_INFO(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::INFO)

#define SYLAR_LOG_DEBUG(logger) SYLAR_LOG_LEVEL(logger, sylar::LogLevel::DEBUG)
```

假如调用语句如下：

```
sylar::Logger::ptr g_logger = SYLAR_LOG_ROOT();
SYLAR_LOG_INFO(g_logger) << "info msg";
```

那么，这个宏将展开成下面的形式：

```
sylar::Logger::ptr g_logger = SYLAR_LOG_ROOT();
if(sylar::LogLevel::INFO <= g_logger->getLevel())
    sylar::LogEventWrap(logger, sylar::LogEvent::ptr(new sylar::LogEvent(logger->getName(),
        level, __FILE__, __LINE__, sylar::GetElapsedMS() - logger->getCreateTime(),
        sylar::GetThreadId(), sylar::GetFiberId(), time(0), sylar::GetThreadName()))).getLogEvent()->getSS() << "info msg";
```

这里要实现通过g_logger打印一条INFO级别的消息。那么，首先判断INFO级别是否高于g_logger本身的日志级别（这里的实现与原版sylar相反，数字越小，优先级越高），如果不高于，那if语句执行不到，这条日志也不会打印，否则，临时构造一个LogEventWrap对象，传入日志器g_logger，以及现场构造的日志事件。通过LogEventWrap的getLogEvent()方法拿到日志事件，再用日志事件的流式日志消息成员输出日志消息。由于

LogEventWrap是在if语句内部构建的，一旦if语句执行结束，LogEventWrap对象就会析构，日志事件也就会被g_logger进行输出，这个设计可以说是非常巧妙。

待补充与完善

目前来看，sylar日志模块已经实现了一个完整的日志框架，并且配合后面的配置模块，可用性很高，待补充与完善的地方主要存在于LogAppender，目前只提供了输出到终端与输出到文件两类LogAppender，但从实际项目来看，以下几种类型的LogAppender都是非常有必要的：

1. Rolling File Appender, 循环覆盖写文件
2. Rolling Memory Appender, 循环覆盖写内存缓冲区
3. 支持日志文件按大小分片或是按日期分片
4. 支持网络日志服务器，比如syslog

无标签

浙ICP备2021003588号

环境变量模块

由 zhongluqiang 创建, 最后修改于 6月 30, 2021

概述

提供程序运行时的环境变量管理功能。这里的环境变量不仅包括系统环境变量，还包括程序自定义环境变量，命令行参数，帮助选项与描述，以及程序运行路径相关的信息。

所谓环境变量就是程序运行时可直接获取和设置的一组变量，它们往往代表一些特定的含义。所有的环境变量都以key-value的形式存储，key和value都是字符串形式。这里可以参考系统环境变量来理解，在程序运行时，可以通过调用getenv()/setenv()接口来获取/设置系统环境变量，比如getenv("PWD")来获取当前路径。在shell中可以通过printenv命令来打印当前所有的环境变量，并且在当前shell中运行的所有程序都共享这组环境变量值。

其他类型的环境变量也可以类比系统环境变量，只不过系统环境变量由shell来保存，而其他类型的环境变量由程序自己内部存储，但两者效果是一样的。具体地，sylar定义了以下几类环境变量：

1. 系统环境变量，由shell保存，sylar环境变量模块提供getEnv()/setEnv()方法用于操作系统环境变量。
2. 程序自定义环境变量，对应get()/add()/has()/del()接口，自定义环境变量保存在程序自己的内存空间中，在内部实现为一个`std::map<std::string, std::string>`结构。
3. 命令行参数，通过解析main函数的参数得到。所有参数都被解析成选项-选项值的形式，选项只能以`-`开头，后面跟选项值。如果一个参数只有选项没有值，那么值为空字符串。命令行参数也保存在程序自定义环境变量中。
4. 帮助选项与描述。这里是统一生成程序的命令行帮助信息，在执行程序时如果指定了`-h`选项，那么就打印这些帮助信息。帮助选项与描述也是存储在程序自己的内存空间中，在内部实现为一个`std::vector<std::pair<std::string, std::string>>`结构。
5. 与程序运行路径相关的信息，包括记录程序名，程序路径，当前路径，这些由单独的成员变量来存储。

sylar环境变量模块设计

与环境变量相关的类只有一个`class Env`，并且这个类被包装成了单例模式。通过单例可以保证程序的环境变量是全局唯一的，便于统一管理。`Env`类提供以下方法：

1. init: 环境变量模块初始化，需要将main函数的参数原样传入init接口中，以便于从main函数参数中提取命令行选项与值，以及通过`argv[0]`参数获取命令行程序名称。
2. add/get/has/del: 用于操作程序自定义环境变量，参数为key-value，get操作支持传入默认值，在对应的环境变量不存在时，返回这个默认值。
3. setEnv/getEnv: 用于操作系统环境变量，对应标准库的`setenv/getenv`操作。
3. addHelp/removeHelp/printHelp: 用于操作帮助选项和描述信息。
4. getExe/getCwd/getAbsolutePath: 用于获取程序名称，程序路径，绝对路径。
5. getConfigPath: 获取配置文件夹路径，配置文件夹路径由命令行-c选项传入。

sylar环境变量模块实现

这里只描述环境变量环境的一些实现细节。

1. 获取程序的bin文件绝对路径是通过`/proc/$pid/`目录下exe软链接文件指向的路径来确定的，用到了`readlink(2)`系统调用。
2. 通过bin文件绝对路径可以得到bin文件所在的目录，只需要将最后的文件名部分去掉即可。
3. 通过`argv[0]`获得命令行输入的程序路径，注意这里的路径可能是以`./`开头的相对路径。
4. 通过`setenv/getenv`操作系统环境变量，参考`setenv(3), getenv(3)`。
5. 提供`getAbsolutePath`方法，传入一个相对于bin文件的路径，返回这个路径的绝对路径。比如默认的配置文件路径就是通过`getAbsolutePath(get("c", "conf"))`来获取的，也就是配置文件夹默认在bin文件所在目录的`conf`文件夹。
6. 按使用惯例，main函数执行的第一条语句应该就是调用Env的init方法初始化命令行参数。

待补充和完善

sylar在解析命令行参数时，没有使用`getopt()`/`getopt_long()`接口，而是使用了自己编写的解析代码，这就导致sylar的命令行参数不支持长选项和选项合并，像`ps -aux`这样的多个选项组合在一起的命令行参数以及`ps --help`这样的长选项是不支持的。

无标签

浙ICP备2021003588号

配置模块

由 zhongluqiang 创建, 最后修改于 7月 01, 2021

配置模块概述

用于定义/声明配置项，并且从配置文件中加载用户配置。

一般而言，一项配置应该包括以下要素：

1. 名称，对应一个字符串，必须唯一，不能与其他配置项产生冲突。
2. 类型，可以是基本类型，但也应该支持复杂类型和自定义类型。
3. 值。
3. 默认值，考虑到用户不一定总是会显式地给配置项赋值，所以配置项最好有一个默认值。
4. 配置变更通知，一旦用户更新了配置值，那么应该通知所有使用了这项配置的代码，以便于进行一些具体的操作，比如重新打开文件，重新起监听端口等。
5. 校验方法，更新配置时会调用校验方法进行校验，以保证用户不会给配置项设置一个非法的值。

一个配置模块应具备的基本功能：

1. 支持定义/声明配置项，也就是在提供配置名称、类型以及可选的默认值的情况下生成一个可用的配置项。由于一项配置可能在多个源文件中使用，所以配置模块还应该支持跨文件声明配置项的方法。
2. 支持更新配置项的值。这点很好理解，配置项刚被定义时可能有一个初始默认值，但用户可能会有新的值来覆盖掉原来的值。
2. 支持从预置的途径中加载配置项，一般是配置文件，也可以是命令行参数，或是网络服务器。这里不仅应该支持基本数据类型的加载，也应该支持复杂数据类型的加载，比如直接从配置文件中加载一个map类型的配置项，或是直接从一个预定格式的配置文件中加载一个自定义结构体。
3. 支持给配置项注册配置变更通知。配置模块应该提供方法让程序知道某项配置被修改了，以便于进行一些操作。比如对于网络服务器而言，如果服务器端口配置变化了，那程序应该重新起监听端口。这个功能一般是通过注册回调函数来实现的，配置使用方预先给配置项注册一个配置变更回调函数，配置项发生变化时，触发对应的回调函数以通知调用方。由于一项配置可能在多个地方引用，所以配置变更回调函数应该是一个数组的形式。
4. 支持给配置项设置校验方法。配置项在定义时也可以指定一个校验方法，以保证该项配置不会被设置成一个非法的值，比如对于文件路径类的配置，可以通过校验方法来确保该路径一定存在。
5. 支持导出当前配置。

gflags介绍

gflags是谷歌开源的一个基于命令行的C++配置库，项目地址：

<https://github.com/gflags/gflags>，文档地址：<https://gflags.github.io/gflags/>。

gflags在配置定义和使用上与sylar的有几分类似，除了侧重于命令行这点外，其他功能差不多。并且，gflags也支持从配置文件中加载配置。sylar与gflags的另外一个不同点是，gflags支持配置参数校验，但sylar不支持。在学习sylar的配置模块之前，可以先通过gflags完整的文档和示例代码感受一下配置模块的使用。

sylar配置模块设计

采用约定优于配置的思想，关于约定优于配置的描述可参考以下链接：

[维基百科：约定优于配置](#)

[如何理解 SpringBoot 中的约定优于配置 - 云+社区 - 腾讯云](#)

简单来说，约定优于配置的背景条件是，一般来说，程序所依赖的配置项都有一个公认的默认值，也就是所谓的约定。这点有可许多可以参考的例子，比如对于一个http网络服务器，服务端口通常都是80端口，对于配置文件夹路径，一般都是conf文件夹，对于数据库目录，一般都是db或data文件夹。对于这些具有公认约定的配置，就不需要麻烦程序员在程序跑起来后再一项一项地指定了，而是可以在初始时就将配置项设置成对应的值。这样，程序员就可以只修改那些约定之外的配置项，然后以最小的代价让程序跑起来。

约定优于配置的方式可以减少程序员做决定的数量，获得简单的好处，同时兼顾灵活性。

在代码上，约定优于配置的思路体现为所有的配置项在定义时都带一个的默认值，以下是一个sylar配置项的示例，这是一个int类型的配置项，名称为 `tcp.connect.timeout`，初始值为5000。

```
static sylar::ConfigVar<int>::ptr g_tcp_connect_timeout = sylar::Config::Lookup("tcp.connect.timeout")
```

sylar的配置项定义之后即可使用，比如上面的配置项可以直接使用 `g_tcp_connect_timeout->getValue()` 获取参数的值，这里获取的为默认值5000。

sylar使用YAML做为配置文件，配置名称大小写不敏感，并且支持级别格式的数据类型，比如上面的配置项对应的YAML配置文件内容如下：

```
tcp:
  connect:
    timeout: 10000
```

这里指定了配置名称为 `tcp.connect.timeout` 的配置项的值为10000。由于配置文件指定的值与默认值不一样，当配置文件加载后，对应的配置项值会被自动更新为10000，如果配置项还注册了配置变更回调函数的话，会一并调用配置变更回调函数以通知配置使用方。

sylar支持STL容器（vector, list, set, map等等），支持自定义类型（需要实现序列化和反序列化方法）。

与配置模块相关的类：

`ConfigVarBase`：配置项基类，虚基类，定义了配置项公有的成员和方法。sylar对每个配置项都包括名称和描述两项成员，以及`toString/fromString`两个纯虚函数方法。`ConfigVarBase`并不包含配置项类型和值，这些由继承类实现，由继承类实现的还包括具体类型的`toString/fromString`方法，用于和YAML字符串进行相互转换。

`ConfigVar`：具体的配置参数类，继承自`ConfigVarBase`，并且是一个模板类，有3个模板参数。第一个模板参数是类型T，表示配置项的类型。另外两个模板参数是FromStr和ToStr，这两个参数是仿函数，FromStr用于将YAML字符串转类型T，ToStr用于将T转YAML字符串。这两个模板参数具有默认值 `LexicalCast<std::string, T>` 和 `LexicalCast<T, std::string>`，根据不同的类型T，FromStr和ToStr具有不同的偏特化实现。

`ConfigVar`类在`ConfigVarBase`基础上包含了一个T类型的成员和一个变更回调函数数组，此外，`ConfigVar`还提供了`setValue/getValue`方法用于获取/更新配置值（更新配置时会一并触发全部的配置变更回调函数），以及`addListener/delListener`方法用于添加或删除配置变更回调函数。

`Config`：`ConfigVar`的管理类，负责托管全部的`ConfigVar`对象，单例模式。提供`Lookup`方法，用于根据配置名称查询配置项。如果调用`Lookup`查询时同时提供了默认值和配置项的描述信息，那么在未找到对应的配置时，会自动创建一个对应的配置项，这样就保证了配置模块定义即可用的特性。除此

外，Config类还提供了LoadFromYaml和LoadFromConfDir两个方法，用于从YAML对象或从命令行-c选项指定的配置文件路径中加载配置。Config的全部成员变量和方法都是static类型，保证了全局只有一个实例。

sylar配置模块实现

挑几个重点讲讲。

sylar的配置模块使用了yaml-cpp作为YAML解析库，关于yaml-cpp的编译和使用可参考GitHub链接：[jbeder/yaml-cpp: A YAML parser and emitter in C++](#)，关于YAML格式的介绍可参考[YAML入门教程 | 菜鸟教程](#)。

sylar的配置模块实现的一大难点是类型转换类（仿函数）的偏特化实现。对于每种类型的配置，在对应的ConfigVar模板类实例化时都要提供其FromStr和ToStr两个仿函数，用于实现该类型和YAML字符串的相互转换。由于配置项的类型众多，包括全部的基本数据类型（int, float, double, string等），以及vector/list/set/unordered_set/map/unordered_map这几个复杂数据类型，还有用户自定义的类型。为了简化代码编写，sylar从一个基本类型的转换类开始，特化出了剩余类型的转换类，这个基本类型如下：

```
/*
 * @brief 类型转换模板类(F 源类型, T 目标类型)
 */
template <class F, class T>
class LexicalCast {
public:
    /**
     * @brief 类型转换
     * @param[in] v 源类型值
     * @return 返回v转换后的目标类型
     * @exception 当类型不可转换时抛出异常
     */
    T operator()(const F &v) {
        return boost::lexical_cast<T>(v);
    }
};
```

这里的LexicalCast类是一个仿函数，它支持 LexicalCast<F, T>()(const F &v) 调用，可将传入的F类型的参数v进行转换，并返回T类型的结果。实际的转换语句是 boost::lexical_cast<T>(v)。但是，受限于 boost::lexical_cast，LexicalCast当前只能实现基本数据类型和std::string的相互转换，不能实现复杂类型的转换，下面的代码可用于演示当前LexicalCast的功能：

```
std::string str1 = LexicalCast<int, std::string>()(123);      // ok, str1等于"123"
int int1 = LexicalCast<std::string, int>()"123");           // ok, int1等于123
std::string str2 = LexicalCast<float, std::string>()(3.14);   // ok, str2等于"3.14"
float float2 = LexicalCast<std::string, float>()"3.14");    // ok, float2等于3.14

vector<int> v = LexicalCast<std::string, vector<int>>()(...); // 错误, LexicalCast目前还不支持
std::string s = LexicalCast<vector<int>, std::string>()(...); // 错误, 同上
```

为了实现YAML字符串和vector/list/set/unordered_set/map/unordered_map的相互转换，就要对每个类型都进行特化，分别实现其转换类，下面是YAML字符串和vector的相互转换实现：

```

/**
 * @brief 类型转换模板类片特化(YAML String 转换成 std::vector<T>)
 */
template <class T>
class LexicalCast<std::string, std::vector<T>> {
public:
    std::vector<T> operator()(const std::string &v) {
        YAML::Node node = YAML::Load(v);
        typename std::vector<T> vec;
        std::stringstream ss;
        for (size_t i = 0; i < node.size(); ++i) {
            ss.str("");
            ss << node[i];
            vec.push_back(LexicalCast<std::string, T>()(ss.str()));
        }
        return vec;
    }
};

/** 
 * @brief 类型转换模板类片特化(std::vector<T> 转换成 YAML String)
 */
template <class T>
class LexicalCast<std::vector<T>, std::string> {
public:
    std::string operator()(const std::vector<T> &v) {
        YAML::Node node(YAML::NodeType::Sequence);
        for (auto &i : v) {
            node.push_back(YAML::Load(LexicalCast<T, std::string>()(i)));
        }
        std::stringstream ss;
        ss << node;
        return ss.str();
    }
};

```

上面分别实现了 `LexicalCast<std::string, std::vector<T>>` 和 `LexicalCast<std::vector<T>, std::string>`，其中在转换单个的数组元素时，再次用到了 `LexicalCast<std::string, T>` 和 `LexicalCast<T, std::string>`，如果这里 `T` 是基本数据类型，那么就可以用最开始的基本类型的转换类进行模板实例化并完成转换了，下面是针对 `vector` 和 `YAML` 字符串相互转换的示例：

```

std::vector<int> v = LexicalCast<std::string, std::vector<int>>("[1, 2, 3]"); // ok, v等
std::string s = LexicalCast<std::vector<int>, std::string>()(std::vector<int>{1, 2, 3}); //
// //
// //
// 
```

另外，由于这里的模板实例化是可以嵌套的，由 `vector` 和 `vector` 组合出来的全部类型都可以顺利地实现和 `YAML` 的转化，以下是一个二维数组的示例：

```
std::vector<std::vector<int>> vv = LexicalCast<std::string, std::vector<std::vector<int>>>(std::string ss = LexicalCast<std::vector<std::vector<int>>, std::string>()(vv);
```

上面的代码运行之后，`vv`将具有二维数组 `{{1,2,3},{4,5,6}}` 的值，而字符串`ss`则是与之对应 YAML 格式的二维数组，如下：

```
-  
- 1  
- 2  
- 3  
  
-  
- 4  
- 5  
- 6
```

其他复杂类型的偏特化与`vector`类型，参考源码理解即可。

每实现一个新类型的转换，那这个类型和之前已实现的类型组合出的数据类型也可以顺利实现转换，比如`vector<set>`, `set<vector>`, `set<map>`, `map<set>`, `map<map>`这种。这种基于偏特化实现类型转换的方法可以说是非常巧妙了，代码可以做到高度简化，但功能却非常强大，这也变相展示了泛型程序设计的强大之处吧。

待补充与完善

整合配置文件与命令行参数，配置项可以用命令行选项进行覆盖，并且在导出时进行标注。

配置项支持校验，参考`gflags`。

无标签

浙ICP备2021003588号 ·

线程模块

由 zhongluqiang 创建, 最后修改于 7月 01, 2021

提供线程类和线程同步类, 基于pthread实现, 包括以下类:

Thread : 线程类, 构造函数传入线程入口函数和线程名称, 线程入口函数类型为void(), 如果带参数, 则需要用std::bind进行绑定。线程类构造之后线程即开始运行, 构造函数在线程真正开始运行之后返回。

线程同步类 (这部分被拆分到mutex.h) 中:

Semaphore : 计数信号量, 基于 sem_t 实现

Mutex : 互斥锁, 基于 pthread_mutex_t 实现

RWMutex : 读写锁, 基于 pthread_rwlock_t 实现

Spinlock : 自旋锁, 基于 pthread_spinlock_t 实现

CASLock : 原子锁, 基于 std::atomic_flag 实现

线程模块总体比较简单, 在对pthread相关的接口有一定了解的情况下, 参考源码应该不难理解, 这里说几个重点:

1. 为什么不直接使用C++11提供的thread类。按sylar的描述, 因为thread其实也是基于pthread实现的。并且C++11里面没有提供读写互斥量, RWMutex, Spinlock等, 在高并发场景, 这些对象是经常需要用到的, 所以选择自己封装pthread。
2. 关于线程入口函数。sylar的线程只支持void(void)类型的入口函数, 不支持给线程传参数, 但实际使用时可以结合std::bind来绑定参数, 这样就相当于支持任何类型和数量的参数。
3. 关于子线程的执行时机。sylar的线程类可以保证在构造完成之后线程函数一定已经处于运行状态, 这是通过一个信号量来实现的, 构造函数在创建线程后会一直阻塞, 直到线程函数运行并且通知信号量, 构造函数才会返回, 而构造函数一旦返回, 就说明线程函数已经在执行了。
4. 关于线程局部变量。sylar的每个线程都有两个线程局部变量, 一个用于存储当前线程的Thread指针, 另一个存储线程名称, 通过Thread::GetThis()可以拿到当前线程的指针。
5. 关于范围锁。sylar大量使用了范围锁来实现互斥, 范围锁是指用类的构造函数来加锁, 用析构函数来释放锁。这种方式可以简化锁的操作, 也可以避免忘记解锁导致的死锁问题, 以下是一个范围锁的示例和说明:

```
sylar::Mutex mutex;

{
    sylar::Mutex::Lock lock(mutex); // 定义lock对象, 类型为sylar::Mutex::Lock, 传入互斥量, 在构造函数中完成加锁操作, 如果该锁已经被持有, 那构造lock对象时会抛出异常
    // 临界区操作
    ...
    // 大括号范围结束, 所有在该范围内定义的自动变量都会被回收, lock对象被回收时触发析构函数, 在析构函数中释放锁
}
```

无标签

浙ICP备2021003588号 ·

协程模块

由 zhongluqiang 创建, 最后修改于 7月 07, 2021

基于ucontext_t实现非对称协程。本章描述的内容只和协程有关, 不涉及协程调度, 请对照源码理解: <https://github.com/zhongluqiang/sylar-from-scratch/releases/tag/v1.4.0>。

协程概述

参考以下链接:

[【协程第一话】协程到底是怎样的存在? _哔哩哔哩_bilibili](#)

[【协程第二话】协程和IO多路复用更配哦~_哔哩哔哩_bilibili](#)

[C++ 协程的近况、设计与实现中的细节和决策 - 简书](#)

[微信开源C++Libco介绍与应用（一） - 知乎](#)

[微信开源C++Libco介绍与应用（二） - 知乎](#)

[libco: 一个极速的轻量级 C 非对称协程库 \(10 ns/ctxsw + 一千万协程并发仅耗内存 2.8GB + Github Trending\) - 燕云 - 博客园](#)

[一文彻底弄懂C++开源协程库libco——原理及应用 - 知乎](#)

[NtyCo的实现 · wangbojing/NtyCo Wiki](#)

以上协程库最好跑一下测试用例, 感受一下协程的使用。

[幼麟实验室的个人空间_哔哩哔哩_Bilibili](#) 这个频道出的一系列视频非常精品, 一定要多看看。

建议初学者在开始学习协程时, 不要尝试深入x86/x64体系结构和汇编语言去了解协程上下文的内容和协程切换的具体操作, 只需要了解协程是什么, 协程上下文和协程切换是怎么回事即可。另外, 特别说明, NtyCo的配套视频又臭又长, 不要去看。

看了上面的链接还不了解协程的可以再往下看看。

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, 所以协程也称为轻量级线程。

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。在Linux系统里这个上下文用ucontext_t结构体来表示, 通过getcontext()来获取。

搞清楚协程和线程的区别。协程虽然被称为轻量级线程, 但在单线程内, 协程并不能并发执行, 只能是一个协程结束或yield后, 再执行另一个协程, 而线程则是可以真正并发执行的。其实这点也好理解, 毕竟协程只是以一种花里胡哨的方式去运行一个函数, 不管实现得如何巧妙, 也不可能在单线程里做到同时运行两个函数, 否则还要多线程有何用?

因为单线程下协程并不是并发执行, 而是顺序执行的, 所以不要在协程里使用线程级别的锁来做协程同步, 比如pthread_mutex_t。如果一个协程在持有锁之后让出执行, 那么同线程的其他任何协程一旦尝试再次持有这个锁, 整个线程就锁死了, 这和单线程环境下, 连续两次对同一个锁进行加锁导致的死锁道理完全一样。

同样是单线程环境下, 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。

所谓创建协程, 其实就是把一个函数包装成一个协程对象, 然后再用协程的方式把这个函数跑起来; 所谓协程调度, 其实就是创建一批的协程对象, 然后再创建一个调度协程, 通过调度协程把这些协程对象一个一个消化掉(协程可以在被调度时继续向调度器添加新的调度任务); 所谓IO协程调度, 其实就是在调度协程时, 如果发现这个协程在等待IO就绪, 那就先让这个协程让出执行权, 等对应的IO就绪后再重新恢复这个协程的运行; 所谓定时器, 就是给调度协程预设一个协程对象, 等定时时间到了就恢复预设的协程对象。

ucontext_t接口

sylar的协程模块基于ucontext_t实现, 在学习之前, 必须对ucontext_t和ucontext_t的操作函数非常熟悉。关于ucontext_t的定义和相关的接口如下:

```
// 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, ...);

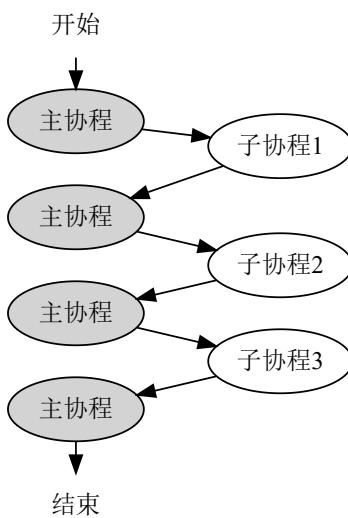
// 恢复ucp指向的上下文，同时将当前的上下文存储到oucp中，
// 和setcontext一样，swapcontext也不会返回，而是会跳转到ucp上下文对应的函数中执行，相当于调用了函数
// swapcontext是sylar非对称协程实现的关键，线程主协程和子协程用这个接口进行上下文切换
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);

```

ucontext_t的示例代码可参考：<https://github.com/zhongluqiang/playground/tree/main/examples/ucontext>，这里不再详细介绍。

sylar协程模块设计

sylar使用非对称协程模型，也就是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，并且在程序结束时，一定要再切回主协程，以保证程序能正常结束，像下面这样：



关于协程模型和什么是对称与非对称协程可参考下面的链接：

[C++ 协程的近况、设计与实现中的细节和决策 - 简书](#)

[协程学习（对称和非对称） - 知乎](#)

从网上的参考资料来看，相对于对称协程，非对称协程具有代码实现简单的特点。

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。

非对称协程的行为与函数类似，因为函数在运行结束后也总是会返回调用者。

虽然目前还没有涉及到协程调度，但这里其实可以将线程的主协程想像成线程的调度协程，每个子协程执行完了，都必须切回线程主协程，由主协程负责选出下一个要执行的子协程。如果子协程可以和子协程切换，那就相当于变相赋予了子协程调度的权利，这在非对称协程里是不允许的。

sylar借助了线程局部变量的功能来实现协程模块。线程局部变量与全局变量类似，不同之处在于声明的线程局部变量在每个线程都独有一份，而全局变量是全部线程共享一份。

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。又由于sylar只使用swapcontext来做协程切换，那就意味着，这两个线程局部变量必须至少有一个是用来保存线程主协程的上下文的，如果这两个线程局部变量存储的都是子协程的上下文，那么不管怎么调用swapcontext，都没法恢复主协程的上下文，也就意味着程序最终无法回到主协程去执行，程序也就跑飞了。

如果将线程的局部变量设置成一个类似链表的数据结构，那理论上应该也可以实现对称协程，也就是子协程可以直接和子协程切换，但代码复杂度上肯定会增加不少，因为要考虑多线程和公平调度的问题。

sylar的非对称协程代码实现简单，并且在后面实现协程调度时可以做到公平调度，缺点是子协程只能和线程主协程切换，意味着子协程无法创建并运行新的子协程，并且在后面实现协程调度时，完成一次子协程调度需要额外多切换一次上下文。

sylar协程模块实现

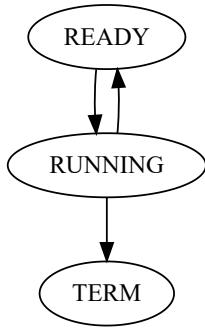
协程状态

这里在sylar的基础上进行简化，对每个协程，只设计了3种状态，分别是READY，代表就绪态，RUNNING，代表正在运行，TERM，代表运行结束。

与sylar版本的实现相比，去掉了INIT状态，HOLD状态，和EXCEPT状态。

sylar的INIT状态是协程对象刚创建时的状态，这个状态可以直接归到READY状态里，sylar的HOLD状态和READY状态与协程调度有关，READY状态的协程会被调度器自动重新调度，而HOLD状态的协程需要显式地再次将协程加入调度，这两个状态也可以归到READY状态里，反正都表示可执行状态。sylar还给协程设计了一个EXCEPT状态，表示协程入口函数执行时出现异常的状态，这个状态可以不管，具体到协程调度模块再讨论。

去掉这几个状态后，协程的状态模型就简单得一目了然了，一个协程要么正在运行（RUNNING），要么准备运行（READY），要运行结束（TERM）。



状态简化后，唯一的缺陷是无法区分一个READY状态的协程对象是刚创建，还是已经运行到一半yield了，这在重置协程对象时有影响。重置协程时，如果协程对象只是刚创建但一次都没运行过，那应该是允许重置的，但如果协程的状态是运行到一半yield了，那应该不允许重置。虽然可以把INIT状态加上以区分READY状态，但既然简化了状态，那就简化到底，让协程只有在TERM状态下才允许重置，问题迎刃而解。

协程原语

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。

协程类实现

sylar的协程通过Fiber类来表示，这个类包含以下成员变量：

```

/// 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;
  
```

接下来是与协程有关的全局变量和线程局部变量。

Fiber的源码定义了两个全局静态变量，用于生成协程id和统计当前的协程数，如下：

```

/// 全局静态变量，用于生成协程id
static std::atomic<uint64_t> s_fiber_id{0};
/// 全局静态变量，用于统计当前的协程数
static std::atomic<uint64_t> s_fiber_count{0};
  
```

然后是线程局部变量，对于每个线程，sylar设计了以下两个线程局部变量用于保存协程上下文信息：

```

/// 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，当前线程的主线程，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;
  
```

这两个线程局部变量保存的协程上下文对协程的实现至关重要，它们的用途如下：

`t_fiber`：保存当前正在运行的协程指针，必须时刻指向当前正在运行的协程对象。协程模块初始化时，`t_fiber`指向线程主协程对象。

`t_thread_fiber`：保存线程主协程指针，智能指针形式。协程模块初始化时，`t_thread_fiber`指向线程主协程对象。当子协程`resume`时，通过`swapcontext`将主协程的上下文保存到`t_thread_fiber`的`ucontext_t`成员中，同时激活子协程的`ucontext_t`上下文。当子协程`yield`时，从`t_thread_fiber`中取得主协程的上下文并恢复运行。

接下来是协程类Fiber的成员方法。

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。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();
}
```

个人感觉这里的GetThis()方法设计得不太好，用函数的副作用掩盖了线程协程初始化这一重要的操作步骤，更好的设计应该是把线程协程初始化和GetThis()分开，让用户知道自己真正在做的事。

接下来是协程原语的实现，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;
}
```

其他实现细节

关于协程id。sylar通过全局静态变量s_fiber_id的自增来生成协程id，每创建一个新协程，s_fiber_id自增1，并作为新协程的id（实际是先取值，再自增1）。

关于线程主协程的构建。线程主协程代表线程入口函数或是main函数所在的协程，这两种函数都不是以协程的手段创建的，所以它们只有ucontext_t上下文，但没有入口函数，也没有分配栈空间。

关于协程切换。子协程的resume操作一定是在主协程里执行的，主协程的resume操作一定是在子协程里执行的，这点完美和swapcontext匹配，参考上面协程原语的实现。

关于智能指针的引用计数，由于t_fiber和t_thread_fiber一个是原始指针一个是智能指针，混用时要注意智能指针的引用计数问题，不恰当的混用可能导致协程对象已经运行结束，但未析构问题。关于协程对象的智能指针引用计数跟踪可参考test_fiber.cc。

注意事项

子协程不能直接resume另一个子协程，像下面这样的代码会直接让程序跑飞：

test_fiber2.cc

```
/*
 * @file test_fiber2.cc
 * @brief 协程测试，用于演示非对称协程
 * @version 0.1
 * @date 2021-06-15
 */
#include "sylar/fiber.h"
#include "sylar/sylar.h"
#include <string>
#include <vector>

sylar::Logger::ptr g_logger = SYLAR_LOG_ROOT();

void run_in_fiber2() {
    SYLAR_LOG_INFO(g_logger) << "run_in_fiber2 begin";
    SYLAR_LOG_INFO(g_logger) << "run_in_fiber2 end";
}

void run_in_fiber() {
    SYLAR_LOG_INFO(g_logger) << "run_in_fiber begin";

    /**
     * 非对称协程，子协程不能创建并运行新的子协程，下面的操作是有问题的，
     * 子协程再创建子协程，原来的主协程就跑飞了
     */
    sylar::Fiber::ptr fiber(new sylar::Fiber(run_in_fiber2));
    fiber->resume();

    SYLAR_LOG_INFO(g_logger) << "run_in_fiber end";
}

int main(int argc, char *argv[]) {
    sylar::EnvMgr::GetInstance()->init(argc, argv);
    sylar::Config::LoadFromConfDir(sylar::EnvMgr::GetInstance()->getConfigPath());

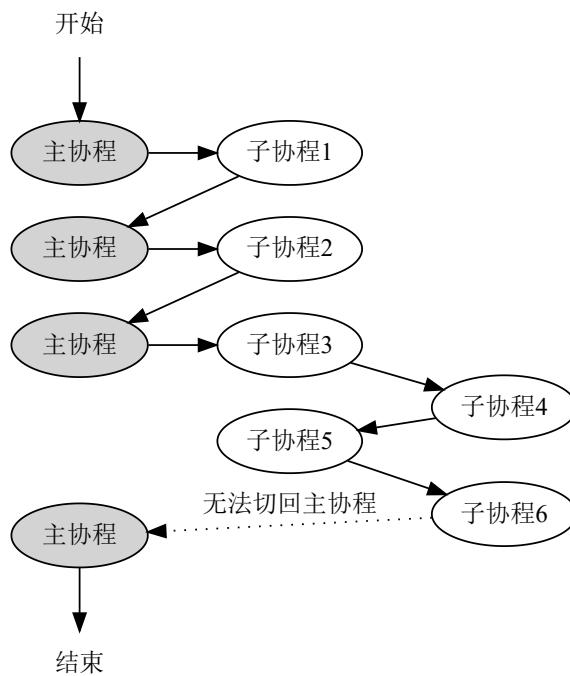
    SYLAR_LOG_INFO(g_logger) << "main begin";

    sylar::Fiber::GetThis();

    sylar::Fiber::ptr fiber(new sylar::Fiber(run_in_fiber));
    fiber->resume();

    SYLAR_LOG_INFO(g_logger) << "main end";
    return 0;
}
```

究其原因，在于上面的run_in_fiber本身是一个子协程，在其内部执行另一个协程的resume时，swapcontext会把run_in_fiber的上下文保存到t_thread_fiber中，导致t_thread_fiber不再指向main函数的上下文，导致程序跑飞。



无标签

浙ICP备2021003588号

协程调度模块

由 zhongluqiang 创建, 最后修改于 7月 12, 2021

实现了一个N-M的协程调度器, N个线程运行M个协程, 协程可以在线程之间进行切换, 也可以绑定到指定线程运行。本章对应源码:
<https://github.com/zhongluqiang/sylar-from-scratch/releases/tag/v1.5.0>。

学习协程调度之前必须完全掌握sylar的协程模块, 参考[协程模块](#)。

实现协程调度之后, 可以解决前一章协程模块中子协程不能运行另一个子协程的缺陷, 子协程可以通过向调度器添加调度任务的方式来运行另一个子协程。

协程调度最难理解的地方是当caller线程也参与调度时调度协程和主线程切换的情况, 注意对照源码进行理解。

协程调度概述

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。

从某种程度来看, 协程调度其实非常简单, 简单到用下面的代码就可以实现一个调度器, 这个调度器可以添加调度任务, 运行调度任务, 并且还是完全公平调度的, 先添加的任务先执行, 后添加的任务后执行。

```
/*
 * @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;
}

```

不要觉得上面这个调度器扯淡，除了不支持多线程，sylar的协程调度器和它的设计思路完全相同，甚至，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，只不过代码看起来更花里胡哨一些。

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。

sylar协程调度模块设计

sylar的协程调度模块支持多线程，支持使用caller线程进行调度，支持添加函数或协程作为调度对象，并且支持将函数或协程绑定到一个具体的线程上执行。

首先是协程调度器的初始化。sylar的协程调度器在初始化时支持传入线程数和一个布尔型的use_caller参数，表示是否使用caller线程。在使用caller线程的情况下，线程数自动减一，并且调度器内部会初始化一个属于caller线程的调度协程并保存起来（比如，在main函数中创建的调度器，如果use_caller为true，那调度器会初始化一个属于main函数线程的调度协程）。

调度器创建好后，即可调用调度器的schedule方法向调度器添加调度任务，但此时调度器并不会立刻执行这些任务，而是将它们保存到内部的一个任务队列中。

接下来是调用start方法启动调度。start方法调用后会创建调度线程池，线程数量由初始化时的线程数和use_caller确定。调度线程一旦创建，就会立刻从任务队列里取任务执行。比较特殊的一点是，如果初始化时指定线程数为1且use_caller为true，那么start方法什么也不做，因为不需要创建新线程用于调度。并且，由于没有创建新的调度线程，那只能由caller线程的调度协程来负责调度协程，而caller线程的调度协程的执行时机与start方法并不在同一个地方。

接下来是调度协程，对应run方法。调度协程负责从调度器的任务队列中取任务执行。取出的任务即子协程，这里调度协程和子协程的切换模型即为前一章介绍的非对称模型，每个子协程执行完后都必须返回调度协程，由调度协程重新从任务队列中取新的协程并执行。如果任务队列空了，那么调度协程会切换到一个idle协程，这个idle协程什么也不做，等有新任务进来时，idle协程才会退出并回到调度协程，重新开始下一轮调度。

在非caller线程里，调度协程就是调度线程的主线程，但在caller线程里，调度协程并不是caller线程的主协程，而是相当于caller线程的子协程，这在协程切换时会有大麻烦（这点是sylar协程调度模块最难理解的地方），如何处理这个问题将在下面的章节专门进行讨论。

接下来是添加调度任务，对应schedule方法，这个方法支持传入协程或函数，并且支持一个线程号参数，表示是否将这个协程或函数绑定到一个具体的线程上执行。如果任务队列为空，那么在添加任务之后，要调用一次tickle方法以通知各调度线程的调度协程有新任务来了。

在执行调度任务时，还可以通过调度器的GetThis()方法获取到当前调度器，再通过schedule方法继续添加新的任务，这就变相实现了在子协程中创建并运行新的子协程的功能。

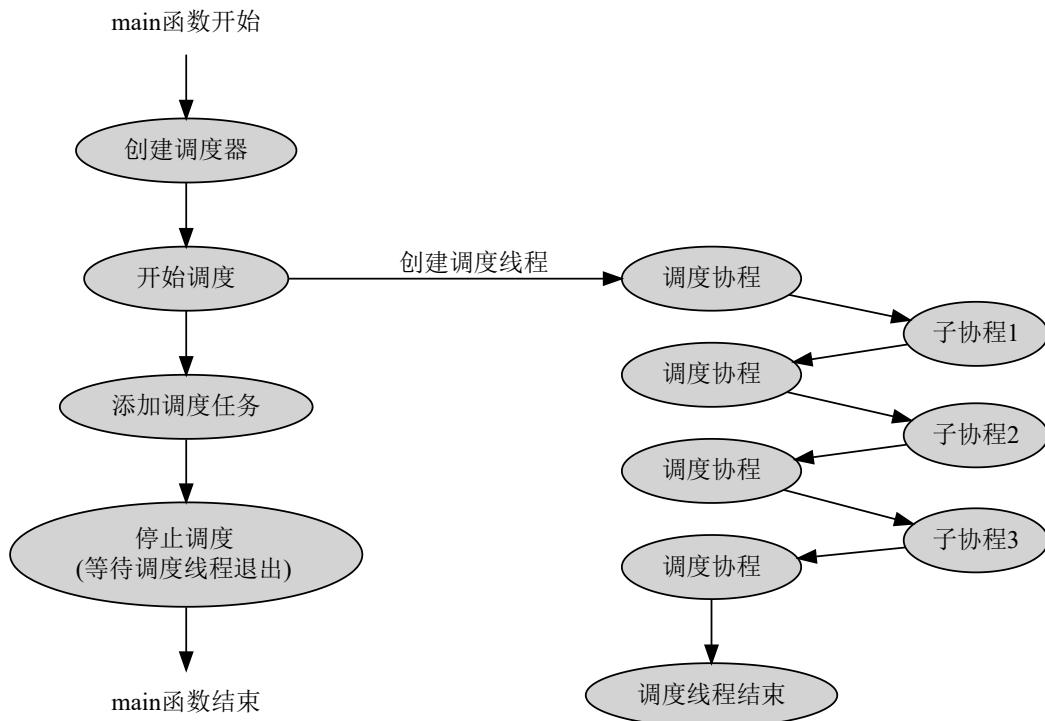
接下来是调度器的停止。调度器的停止行为要分两种情况讨论，首先是use_caller为false的情况，这种情况下，由于没有使用caller线程进行调度，那么只需要简单地等各个调度线程的调度协程退出就行了。如果use_caller为true，表示caller线程也要参与调度，这时，调度器初始化时记录的属于caller线程的调度协程就要起作用了，在调度器停止前，应该让这个caller线程的调度协程也运行一次，让caller线程完成调度工作后再退出。如果调度器只使用了caller线程进行调度，那么所有的调度任务要在调度器停止时才会被调度。

调度协程切换问题

这里分两种典型情况来讨论一下调度协程的切换情况，其他情况可以看成以下两种情况的组合，原理是一样的。

1. 线程数为1，且use_caller为true，对应只使用main函数线程进行协程调度的情况。
2. 线程数为1，且use_caller为false，对应额外创建一个线程进行协程调度、main函数线程不参与调度的情况。

这里先说情况2。情况2比较好理解，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，参考下面的图示：



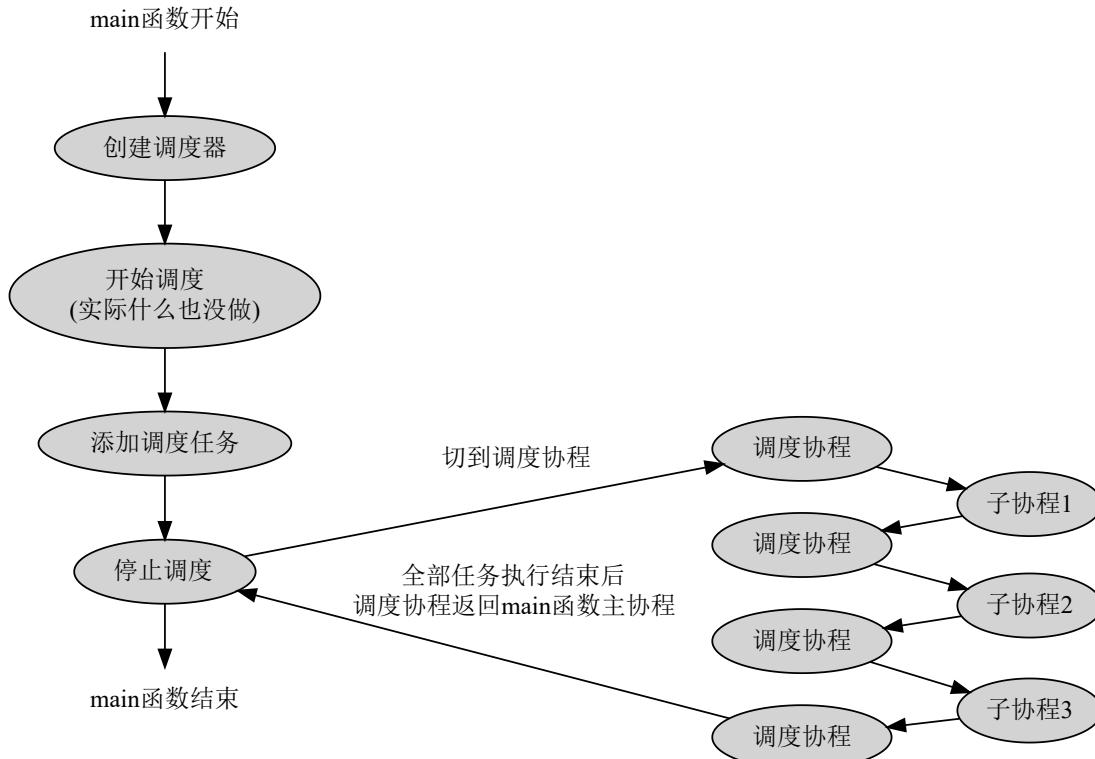
情况1则比较复杂，因为没有额外的线程进行协程调度，那只能用main函数所在的线程来进行调度，而梳理一下main函数线程要运行的协程，会发现有以下三类协程：

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：

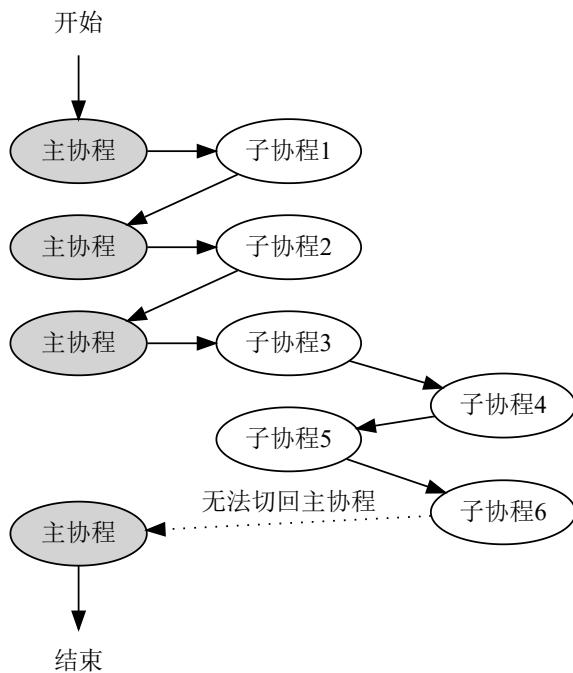
1. main函数主协程运行，创建调度器
2. 仍然是main函数主协程运行，向调度器添加一些调度任务
3. 开始协程调度，main函数主协程让出执行权，切换到调度协程，调度协程从任务队列里按顺序执行所有的任务
4. 每次执行一个任务，调度协程都要让出执行权，再切到该任务的协程里去执行，任务执行结束后，还要再切回调度协程，继续下一个任务的调度
5. 所有任务都执行完后，调度协程还要让出执行权并切回main函数主协程，以保证程序能顺利结束。

上面的过程也可以总结为：main函数先攒下一波协程，然后切到调度协程里去执行，等把这些协程都消耗完后，再从调度协程切回来，像下面这样：



从这个图看，是不是本文开头的简单调度器实现有几分神似？

然而，回顾一下前面协程模块就会发现，上面这种协程切换实际是有问题的，参考下面这张图：



在非对称协程里，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，最后一个也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：

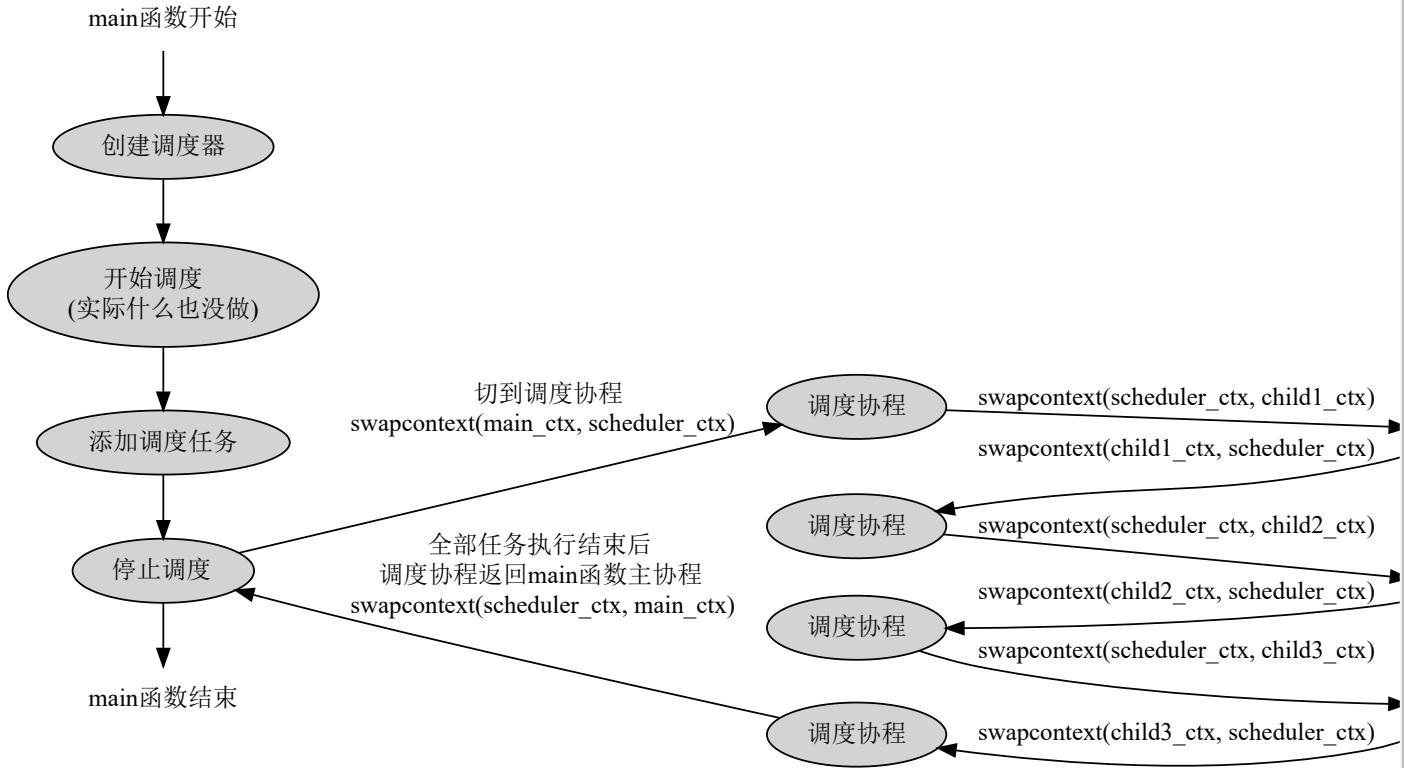
1. 给协程类增加一个bool类型的成员m_runInScheduler，用于记录该协程是否通过调度器来运行。

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。

经过上面的改造了，就可以解决单线程环境下caller线程主协程-调度协程-任务协程之间的上下文切换问题了，假设caller线程主协程的上下文为main_ctx，调度协程的上下文为scheduler_ctx，任务协程上下文为child_ctx，那么单线程下的协程切换将像下面这样（图片显示不全的话请缩小显示比例）：



其他情况的讨论

讨论几种情况：

1. 调度器的退出问题。调度器内部有一个协程任务队列，调度器调度的实质就是内部的线程池从这个任务队列拿任务并执行，那么，停止调度时，如果任务队列还有任务剩余，要怎么处理？这里可以简化处理，强制规定只有所有的任务都完成调度时，调度器才可以退出，如果有一个任务没有执行完，那调度器就不能退出。

2. 任务协程执行过程中主动调用yield让出了执行权，调度器要怎么处理？半路yield的协程显然并没有执行完，一种处理方法是调度器来帮协程擦屁股，在检测到协程从resume返回时，如果状态仍为READY，那么就把协程重新扔回任务队列，使其可以再次被调度，这样保证一个协程可以执行结束。但这种策略是画蛇添足的，从生活经验的角度来看，一个成熟的协程肯定要学会自我管理，既然你自己yield了，那么你就应该自己管理好自己，而不是让别人来帮你，这样才算是一个成熟的协程。对于主动yield的协程，我们的策略是，调度器直接认为这个任务已经调度完了，不再将其加入任务队列。如果协程想完整地运行，那么在yield之前，协程必须先把自己再扔回当前调度器的任务队列里，然后再执行yield，这样才能确保后面还会再来调度这个协程。

这里规定了一点，协程在主动执行yield前，必须先将自己重新添加到调度器的任务队列中。如果协程不顾后果地执行yield，最后的后果就是协程将永远无法再被执行，也就是所说的逃逸状态。（sylar的处理方法比较折衷一些，sylar定义了两种yield操作，一种是yield to ready，这种yield调度器会再次将协程加入任务队列并等待调度，另一种是yield to hold，这种yield调度器不会再将协程加入任务队列，协程在yield之前必须自己先将自己加入到协程的调度队列中，否则协程就处于逃逸状态。再说一点，sylar定义的yield to ready，在整个sylar框架内一次都没用到，看来sylar也同意，一个成熟的协程要学会自我管理。）

3. 只使用调度器所在的线程进行调度，典型的就是main函数中定义调度器并且只使用main函数线程执行调度任务。这种场景下，可以认为是main函数先攒下一波协程，然后切到调度协程，把这些协程消耗完后再从调度协程切回main函数协程。每个协程在运行时也可以继续创建新的协程并加入调度。如果所有协程都调度完了，并且没有创建新的调度任务，那么下一步就是讨论idle该如何处理。

4. idle如何处理，也就是当调度器没有协程可调度时，调度协程该怎么办。直觉上来看这里应该有一些同步手段，比如，没有调度任务时，调度协程阻塞住，比如阻塞在一个idle协程上，等待新任务加入后退出idle协程，恢复调度。然而这种方案是无法实现的，因为每个线程同一时间只能有一个协程在执行，如果调度线程阻塞在idle协程上，那么除非idle协程自行让出执行权，否则其他的协程都得不到执行，这里就造成了一个先有鸡还是先有蛋的问题：只有创建新任务idle协程才会退出，只有idle协程退出才能创建新任务。为了解决这个问题，sylar采取了一个简单粗暴的办法，如果任务队列空了，调度协程会不停地检测任务队列，看有没有新任务，俗称忙等待，CPU使用率爆表。这点可以从sylar的源码上发现，一是Scheduler的tickle函数什么也不做，因为根本不需要通知调度线程是否有新任务，二是idle协程在协程调度器未停止的情况下只会yield to hold，而调度协程又会将idle协程重新swapIn，相当于idle啥也不做直接返回。这个问题在sylar框架内无解，只有一种方法可以规避掉，那就是设置autostop标志，这个标志会使得调

度器在调度完所有任务后自动退出。在后续的IOManager中，上面的问题会得到一定的改善，并且tickle和idle可以实现得更加巧妙一些，以应对IO事件。

5. 只有main函数线程参与调度时的调度执行时机。前面说过，当只有main函数线程参与调度时，可以认为是主线程先攒下一波协程，然后切到调度协程开始调度这些协程，等所有的协程都调度完了，调度协程进idle状态，这个状态下调度器只能执行忙等待，啥也做不了。这也就是说，主线程main函数一旦开启了协程调度，就无法回头了，位于开始调度点之后的代码都执行不到。对于这个问题，sylar把调度器的开始点放在了stop方法中，也就是，调度开始即结束，干完活就下班。IOManager也是类似，除了可以调用stop方法外，IOManager类的析构函数也有一个stop方法，可以保证所有的任务都会被调度到。

6. 额外创建了调度线程时的调度执行时机。如果不额外创建线程，也就是线程数为1并且use caller，那所有的调度任务都在stop()时才会进行调度。但如果额外创建了线程，那么，在添加完调度任务之后任务马上就可以在另一个线程中调度执行。归纳起来，如果只使用caller线程进行调度，那所有的任务协程都在stop之后排队调度，如果有额外线程，那任务协程在刚添加到任务队列时就可以得到调度。

7. 协程中的异常要怎么处理，子协程抛出了异常该怎么办？这点其实非常好办，类比一下线程即可，你会在线程外面处理线程抛出的异常吗？答案是不会，所以协程抛出的异常我们也不处理，直接让程序按默认的处理方式来处理即可。一个成熟的协程应该自己处理掉自己的异常，而不是让调度器来帮忙。顺便说一下，sylar的协程调度器处理了协程抛出的异常，并且给异常结束的协程设置了一个EXCEPT状态，这看似贴心，但从长远的角度来看，其实是非常不利于协程的健康成长的。

8. 关于协程调度器的优雅停止。sylar停止调度器的策略如下：

1. 设置m_stopping标志，该标志表示正在停止
2. 检测是否使用了caller线程进行调度，如果使用了caller线程进行调度，那要保证stop方法是由caller线程发起的
3. 通知其他调度线程的调度协程退出调度
4. 通知当前线程的调度协程退出调度
5. 如果使用了caller线程进行调度，那执行一次caller线程的调度协程（只使用caller线程时的协程调度全仰仗这个操作）
6. 等caller线程的调度协程返回
7. 等所有调度线程结束

Sylar协程调度模块实现

这部分请对照源码：<https://github.com/zhongluqiang/sylar-from-scratch/releases/tag/v1.5.0>。

首先是对协程模块的改造，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");
    }
}
}
}

```

然后是对调度任务的定义，如下，这里任务类型可以是协程/函数二选一，并且可指定调度线程。

```

/**
 * @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;
    }
};

```

接下来是调度器的成员变量，包括以下成员：

```

/// 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;
```

接下来是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Fiber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。

接下来是调度器的构造方法，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use_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;
}
```

接下来是两个get方法，用于获取当前线程的调度器的调度协程，这两个都是静态方法：

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

Fiber *Scheduler::GetMainFiber() {
    return t_scheduler_fiber;
}
```

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());
    }
}

```

接下来判断调度器是否已经停止的方法，只有当所有的任务都被执行完了，调度器才可以停止：

```

bool Scheduler::stopping() {
    MutexType::Lock lock(m_mutex);
    return m_stopping && m_tasks.empty() && m_activeThreadCount == 0;
}

```

接下来是调度器的tickle和idle实现，可以看到这两个方法并没有什么卵用：

```

void Scheduler::tickle() {
    SYLAR_LOG_DEBUG(g_logger) << "tickle";
}

void Scheduler::idle() {
    SYLAR_LOG_DEBUG(g_logger) << "idle";
    while (!stopping()) {
        sylar::Fiber::GetThis()->yield();
    }
}

```

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，所以这里其实就是一个忙等待，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();
    }
}
```

注意事项

sylar的协程调度模块因为存任务队列空闲时调度线程忙等待的问题，所以实际上并不实用，真正实用的是后面基于Scheduler实现的IOManager。由于任务队列的任务是按顺序执行的，如果有一个任务占用了比较长时间，那其他任务的执行会受到影响，如果任务执行的是像while(1)这样的循环，那线程数不够时，后面的任務都不会得到执行。另外，当前还没有实现hook功能，像sleep和等待IO就绪这样的操作也会阻塞协程调度。

无标签

浙ICP备2021003588号 ·

IO协程调度模块

由 zhongluqiang 创建, 最后修改于 7月 16, 2021

继承自协程调度器, 封装了epoll, 支持为socket fd注册读写事件回调函数。本章对应源码: <https://github.com/zhongluqiang/sylar-from-scratch/releases/tag/v1.6.0>。

IO协程调度还解决了调度器在idle状态下忙等待导致CPU占用率高的问题。IO协程调度器使用一对管道fd来tickle调度协程, 当调度器空闲时, idle协程通过epoll_wait阻塞在管道的读描述符上, 等管道的可读事件。添加新任务时, tickle方法写管道, idle协程检测到管道可读后退出, 调度器执行调度。

学习本章内容之前必须对sylar的协程模块和协程调度模块非常熟悉, 参考链接 [协程模块 协程调度模块](#), 此外, 对epoll接口也要非常熟悉, 参考 man 7 epoll 。

IO协程调度概述

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(一般是一套接字描述符)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)

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。后续的socket api hook模块也依赖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。这里可以参考一个基于epoll实现的简单事件库: [3.2 epoll的反应堆模式实现 · libevent深入浅出 · 看云](#), sylar的IO调度和这种写法类似。

sylar IO协程调度模块设计

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。

sylar IO协程调度器实现

sylar的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Scheduler, 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, 那就没有必要发通知
*/
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Scheduler::schedule(), 二是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; 三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_wait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;
    }

    // 删除已经发生的事件，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();
}

```

syclar IO协程调度的几点总结

1. 总得来说，syclar的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。

2. 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。

3. 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。
4. 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。
5. cancelEvent和cancelAll都会触发一次事件，但delEvent不会。
6. FdContext的寻址问题，sylar直接使用fd的值作为FdContext数组的下标，这样可以快速找到一个fd对应的FdContext。由于关闭的fd会被重复利用，所以这里也不用担心FdContext数组膨胀太快，或是利用率低的问题。
7. IO协程调度器的退出，不但所有协程要完成调度，所有IO事件也要完成调度。
8. sylar的IO协程调度器应该配合非阻塞IO来使用，如果使用阻塞模式，可能会阻塞进程，参考[为什么 IO 多路复用要搭配非阻塞 IO? - 知乎](#)。

无标签

定时器模块

由 zhongluqiang 创建, 最后修改于 7月 17, 2021

基于epoll超时实现定时器功能, 精度毫秒级, 支持在指定超时时间结束之后执行回调函数。本章对应源码: <https://github.com/zhongluqiang/sylar-from-scratch/releases/tag/v1.7.0>。

《Linux高性能服务器编程.pdf》第11章对定时器有完整而详细地介绍, 包括原理与代码实现, 本文关于定时器的介绍与实现都是从这章摘抄的, 目的只为了加深本人对定时器的理解。建议读者先阅读这章的内容, 然后直接看sylar定时器的设计与实现, 下面的定时器概述和几种定时器实现随便看看就好了。

定时器概述

通过定时器可以实现给服务器注册定时事件, 这是服务器上经常要处理的一类事件, 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。

定时事件依赖于Linux提供的定时机制, 它是驱动定时事件的原动力, 目前Linux提供了以下几种可供程序利用的定时机制:

1. alarm()或setitimer(), 这俩的本质都是先设置一个超时时间, 然后等SIGALARM信号触发, 通过捕获信号来判断超时
2. 套接字超时选项, 对应SO_RECVTIMEO和SO_SNDFTIMEO, 通过errno来判断超时
3. 多路复用超时参数, select/poll/epoll都支持设置超时参数, 通过判断返回值为0来判断超时
4. timer_create系统接口, 实质也是借助信号, 参考man 2 timer_create
5. timerfd_create系列接口, 通过判断文件描述符可读来判断超时, 可配合IO多路复用, 参考man 2 timerfd_create

服务器程序通常需要处理众多定时事件, 如何有效地组织与管理这些定时事件对服务器的性能至关重要。为此, 我们要将每个定时事件分别封装成定时器, 并使用某种容器类数据结构, 比如链表、排序链表和时间轮, 将所有定时器串联起来, 以实现对定时事件的统一管理。

每个定时器通常至少包含两个成员: 一个超时时间(相对时间或绝对时间)和一个任务回调函数。除此外, 定时器还可以包括回调函数参数及是否自动重启等信息。

有两种高效管理定时器的容器: 时间轮和时间堆, sylar使用时间堆的方式管理定时器。

几种定时器实现

基于升序链表的定时器

1. 所有定时器组织成链表结构, 链表成员包含超时时间, 回调函数, 回调函数参数, 以及链表指针域。
2. 定时器在链表中按超时时间进行升序排列, 超时时间短的在前, 长的在后。每次添加定时器时, 都要按超时时间将定时器插入到链表的指定位置。
3. 程序运行后维护一个周期性触发的tick信号, 比如利用alarm函数周期性触发ALARM信号, 在信号处理函数中从头遍历定时器链表, 判断定时器是否超时。如果定时器超时, 则记录下该定时器, 然后将其从链表中删除。
4. 执行所有超时的定时器的回调函数。

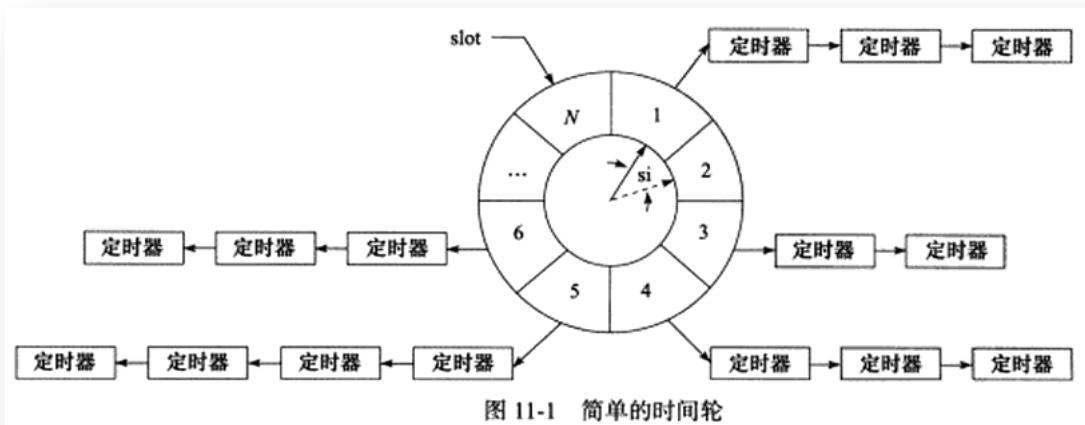
以上就是一个基于升序链表的定时器实现, 这种方式添加定时器的时间复杂度是O(n), 删除定时器的时间复杂度是O(1), 执行定时任务的时间复杂度是O(1)。

tick信号的周期对定时器的性能有较大的影响, 当tick信号周期较小时, 定时器精度高, 但CPU负担较高, 因为要频繁执行信号处理函数; 当tick信号周期较大时, CPU负担小, 但定时精度差。

当定时器数量较多时, 链表插入操作开销比较大。

时间轮

与上面的升序链表实现方式类似, 也需要维护一个周期性触发的tick信号, 但不同的是, 定时器不再组织成单链表结构, 而是按照超时时间, 通过散列分布到不同的时间轮上, 像下面这样:



上面的时间轮包含N个槽位，每个槽位上都有一个定时器链表。时间轮以恒定的速度顺时针转动，每转一步，表盘上的指针就指向下一个槽位。每次转动对应一个tick，它的周期为si，一个共有N个槽，所以它运转一周的时间是N*si。

每个槽位都有一条定时器链表，同一条链表上的每个定时器都具有相同的特征：前后节点的定时时间相差N*si的整数倍。时间轮正是利用这个关系将定时器散列到不同的链表上。假如现在指针指向槽cs，我们要添加一个定时时间为ti的定时器，则该定时器将被插入槽ts(time slot)对应的链表中：

```
ts = (cs + (ti / si)) % N
```

时间轮通过哈希表的思想，将定时器散列到不同的链表上，每个链表的定时器数目都明显少于原来的排序链表，插入效率基本不受定时器数目的影响。

和升序链表一样，tick的周期将影响定时器精度和CPU负载，除此外，时间轮上的槽数量N还对定时器的效率有影响，N越大，则散列越均匀，插入效率越高，N越小，则散列越容易冲突，至N等于1时，时间轮将完全退化成升序链表。

上面的时间轮只有一个轮子，而复杂的时间轮可能有多个轮子，不同的轮子拥有不同的粒度。相邻的两个轮子，精度高的转一圈，精度低的仅往前移动一槽，就像水表一样。

注意点：

单个槽上的定时器链表仍然是按升序链表来组织的，只不过前后两个节点的时间差一定是N*si的整数倍。注意这里前后节点的时间差不一定是1个N*si，也有可能是好几个N*si，所以不能通过定时器所在的槽位和链表位置直接推算出定时器的超时时间。或者换个说法，表盘指针转到某个槽时，仍需要按升序链表的方式遍历这个链表的节点，并判断是否超时。

时间堆

上面的两种定时器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，并将这个最短时间作为下一个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定时器设计

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定时器实现

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;
/**
 * @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
     */
    Timer();
}
```

```

* @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_wait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();
}
...
}

```

几点实现细节的讨论

- 创建定时器时只传入了相对超时时间，内部要先进行转换，根据当前时间把相对时间转化成绝对时间。
- sylar支持创建条件定时器，也就是在创建定时器时绑定一个变量，在定时器触发时判断一下该变量是否仍然有效，如果变量无效，那就取消触发。
- 关于onTimerInsertedAtFront()方法的作用。这个方法是IOManager提供给TimerManager使用的，当TimerManager检测到新添加的定时器的超时时间比当前最小的定时器还要小时，TimerManager通过这个方法来通知IOManager立刻更新当前的epoll_wait超时，否则新添加的定时器的执行时间将不准确。实际实现时，只需要在onTimerInsertedAtFront()方法内执行一次tickle就行了，tickle之后，epoll_wait会立即退出，并重新从TimerManager中获取最近的超时时间，这时拿到的超时时间就是新添加的最小定时器的超时时间了。
- 关于校时问题。sylar的定时器以gettimeofday()来获取绝对时间点并判断超时，所以依赖于系统时间，如果系统进行了校时，比如NTP时间同步，那这套定时机制就失效了。sylar的解决办法是设置一个较小的超时步长，比如3秒钟，也就是epoll_wait最多3秒超时，假设最近一个定时器的超时时间是10秒以后，那epoll_wait需要超时3秒才会触发。每次超时之后除了要检查有没有要触发的定时器，还顺便检查一下系统时间有没有被往回调。如果系统时间往回调了1个小时以上，那就触发全部定时器。个人感觉这个办法有些粗糙，其实只需要换个时间源就可以解决校时问题，换成clock_gettime(CLOCK_MONOTONIC_RAW)的方式获取系统的单调时间，就可以解决这个问题了。

无标签

hook模块

由 zhongluqiang 创建, 最后修改于 9月 28, 2021

hook系统底层和socket相关的API, socket IO相关的API, 以及sleep系列的API。hook的开启控制是线程粒度的, 可以自由选择。通过hook模块, 可以使一些不具异步功能的API, 展现出异步的性能, 如MySQL。

注意: 本篇提到的系统调用接口实际是指C标准函数库提供的接口, 而不是单指Linux提供的系统调用, 比如malloc和free就不是系统调用, 它们是C标准函数库提供的接口。

hook概述

理解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。

还可以用C++的子类重载来理解hook。在C++中, 子类在重载父类的同名方法时, 一种常见的实现方式是子类先完成自己的操作, 再调用父类的操作, 如下:

```
class Base {
public:
    void Print() {
        cout << "This is Base" << endl;
    }
};

class Child : public Base {
public:
    /// 子类重载时先实现自己的操作, 再调用父类的操作
    void Print() {
        cout << "This is Child" << endl;
        Base::Print();
    }
};
```

在上面的代码实现中, 调用子类的Print方法, 会先执行子类的语句, 然后再调用父类的Print方法, 这就相当于子类hook了父类的Print方法。

由于hook之后的系统调用与原始的系统调用同名, 所以对于程序开发者来说也很方便, 不需要重新学习新的接口, 只需要按老的接口调用惯例直接写代码就行了。

hook功能

hook的目的是在不重新编写代码的情况下, 把老代码中的socket IO相关的API都转成异步, 以提高性能。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:

1. 协程1: sleep(2) 睡眠两秒后返回。
2. 协程2: 在socket fd1 上send 100k数据。
3. 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:

1. 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。
2. 调度协程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。
3. 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:

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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实现

这里只讲解动态链接中的hook实现，静态链接以及基于内核模块的hook不在本章讨论范围。

在学习hook之前需要对Linux的动态链接有一定的了解，建议阅读《程序员的自我修养——链接、装载与库》第7章。本站[关于链接与装载的几个测试代码](#)提供了一些示例，有助于理解动态链接的具体行为。

hook的实现机制非常简单，就是通过动态库的全局符号介入功能，用自定义的接口来替换掉同名的系统调用接口。由于系统调用接口基本上是由C标准函数库libc提供的，所以这里要做的事情就是用自定义的动态库来覆盖掉libc中的同名符号。

基于动态链接的hook有两种方式，第一种是外挂式hook，也称为非侵入式hook，通过优先加载自定义动态库来实现对后加载的动态库进行hook，这种hook方式不需要重新编译代码，考虑以下例子：

main.c

```
#include <unistd.h>
#include <string.h>

int main() {
    write(STDOUT_FILENO, "hello world\n", strlen("hello world\n")); // 调用系统调用write写标准输出文件描述符
    return 0;
}
```

在这个例子中，可执行程序调用write向标准输出文件描述符写数据。对这个程序进行编译和执行，效果如下：

```
# gcc main.c  
# ./a.out  
hello world
```

使用ldd命令查看可执行程序的依赖的共享库，如下：

```
# ldd a.out
        linux-vdso.so.1 (0x00007ffc96519000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd40a61000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd40c62000)
```

可以看到其依赖libc共享库，write系统调用就是由libc提供的。

gcc 编译生成可执行文件时会默认链接 libc 库，所以不需要显式指定链接参数，这点可以在编译时给 gcc 增加一个 “-v” 参数，将整个编译流程详细地打印出来进行验证，如下：

```
# gcc -v main.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
...
/usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
COLLECT_GCC_OPTIONS=''-v' '-mtune=generic' '-march=x86-64'
```

注意上面的 "/usr/lib/gcc/x86_64-linux-gnu/9/collect2 ... -pop-state **-lc** -lgcc ..."，这里的 **-lc** 就说明程序在进行链接时会自动链接一次 libc。

下面在不重新编译代码的情况下，用自定义的动态库来替换掉可执行程序a.out中的write实现，新建hook.c，内容如下：

hook.c

```
#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"));
}
```

这里实现了一个write函数，这个函数的签名和libc提供的write函数完全一样，函数内容是用syscall的方式直接调用编号为SYS_write的系统调用，实现的效果也是往标准输出写内容，只不过这里我们将输出内容替换成其他值。将hook.c编译成动态库：

```
gcc -fPIC -shared hook.c -o libhook.so
```

通过设置 `LD_PRELOAD` 环境变量，将 `libhook.so` 设置成优先加载，从而覆盖掉 `libc` 中的 `write` 函数，如下：

```
# LD_PRELOAD="./libhook.so" ./a.out
12345
```

这里我们并没有重新编译可执行程序 `a.out`，但是可以看到，`write` 的实现已经替换了我们自己的实现。究其原因，就是 `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。

第二种方式的 `hook` 是侵入式的，需要改造代码或是重新编译一次以指定动态库加载顺序。如果是以改造代码的方式来实现 `hook`，那么可以像下面这样直接将 `write` 函数的实现放在 `main.c` 里，那么编译时全局符号表里先出现的必然是 `main.c` 中的 `write` 符号：

```
main.c

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

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

int main() {
    write(STDOUT_FILENO, "hello world\n", strlen("hello world\n")); // 这里调用的是上面的write实现
    return 0;
}
```

如果不改造代码，那么可以重新编译一次，通过编译参数将自定义的动态库放在 `libc` 之前进行链接。由于默认情况下 `gcc` 总会链接一次 `libc`，并且 `libc` 的位置也总在命令行所有参数后面，所以只需要像下面这样操作就可以了：

```
# gcc main.c -L . -lhook -Wl,-rpath=.
# ./a.out
12345
```

这里显式指定了链接 `libhook.so`（`-Wl,-rpath=.` 用于指定运行时的动态库搜索路径，避免找不到动态库的问题），由于 `libhook.so` 的链接位置比 `libc` 要靠前（可以通过 `gcc -v` 进行验证），所以运行时会先加载 `libhook.so`，从而实现全局符号介入，这点也可以通过 `ldd` 命令来查看：

```
# ldd a.out
linux-vdso.so.1 (0x00007ffe615f9000)
libhook.so => ./libhook.so (0x00007fab4bae3000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fab4b8e9000)
/lib64/ld-linux-x86-64.so.2 (0x00007fab4baef000)
```

关于 `hook` 的另一个讨论点是如何找回已经被全局符号介入机制覆盖的系统调用接口，这个功能非常实用，因为大部分情况下，系统调用提供的功能都是无可替代的，我们虽然可以用 `hook` 的方式将其替换成自己的实现，但是最终要实现的功能，还是得由原始的系统调用接口来完成。

以 `malloc` 和 `free` 为例，假如我们要 `hook` 标准库提供的 `malloc` 和 `free` 接口，以跟踪每次分配和释放的内存地址，判断有无内存泄漏问题，那么具体的实现方式应该是，先调用自定义的 `malloc` 和 `free` 实现，在分配和释放内存之前，记录下内存地址，然后再调用标准库里的 `malloc` 和 `free`，以真正实现内存申请和释放。

上面的过程涉及到了查找后加载的动态库里被覆盖的符号地址问题。首先，这个操作本身就具有合理性，因为程序运行时，依赖的动态库无论是先加载还是后加载，最终都会被加载到程序的进程空间中，也就是说，那些因为加载顺序靠后而被覆盖的符号，它们只是被“雪藏”了而已，实际还是存在于程序的进程空间中的，通过一定的办法，可以把它们再找回来。在 Linux 中，这个方法就是 `dlsym`，它的函数原型如下：

```
#define _GNU_SOURCE
#include <dlfcn.h>

void *dlsym(void *handle, const char *symbol);
```

关于 `dlsym` 的使用可参考 `man 3 dlsym`，在链接时需要指定 `-ldl` 参数。使用 `dlsym` 找回被覆盖的符号时，第一个参数固定为 `RTLD_NEXT`，第二个参数为符号的名称，下面通过 `dlsym` 来实现上面的内存跟踪功能：

```
hook_malloc.c

#define _GNU_SOURCE
#include <dlfcn.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

typedef void* (*malloc_func_t)(size_t size);
typedef void (*free_func_t)(void *ptr);
```

```

// 这两个指针用于保存libc中的malloc和free的地址
malloc_func_t sys_malloc = NULL;
free_func_t sys_free = NULL;

// 重定义malloc和free，在这里重定义会导致libc中的同名符号被覆盖
// 这里不能调用带缓冲的printf接口，否则会出段错误
void *malloc(size_t size) {
    // 先调用标准库里的malloc申请内存，再记录内存分配信息，这里只是简单地将内存地址和长度打印出来
    void *ptr = sys_malloc(size);
    fprintf(stderr, "malloc: ptr=%p, length=%ld\n", ptr, size);
    return ptr;
}

void free(void *ptr) {
    // 打印内存释放信息，再调用标准库里的free释放内存
    fprintf(stderr, "free: ptr=%p\n", ptr);
    sys_free(ptr);
}

int main() {
    // 通过dlsym找到标准库中的malloc和free的符号地址
    sys_malloc = dlsym(RTLD_NEXT, "malloc");
    assert(dlerror() == NULL);
    sys_free = dlsym(RTLD_NEXT, "free");
    assert(dlerror() == NULL);

    char *ptrs[5];

    for(int i = 0; i < 5; i++) {
        ptrs[i] = malloc(100 + i);
        memset(ptrs[i], 0, 100 + i);
    }

    for(int i = 0; i < 5; i++) {
        free(ptrs[i]);
    }
    return 0;
}

```

编译运行以上代码，效果如下：

```

# gcc hook_malloc.c -ldl
# ./a.out
malloc: ptr=0x55775fa8e2a0, length=100
malloc: ptr=0x55775fa8e310, length=101
malloc: ptr=0x55775fa8e380, length=102
malloc: ptr=0x55775fa8e3f0, length=103
malloc: ptr=0x55775fa8e460, length=104
free: ptr=0x55775fa8e2a0
free: ptr=0x55775fa8e310
free: ptr=0x55775fa8e380
free: ptr=0x55775fa8e3f0
free: ptr=0x55775fa8e460

```

sylar hook模块设计

sylar的hook功能以线程为单位，可自由设置当前线程是否使用hook。默认情况下，协程调度器的调度线程会开启hook，而其他线程则不会开启。sylar对以下函数进行了hook，并且只对socket fd进行了hook，如果操作的不是socket fd，那会直接调用系统原本的API，而不是hook之后的API：

```

sleep
usleep
nanosleep
socket
connect
accept
read
readv
recv
recvfrom
recvmsg
write
writev
send
sendto

```

```
sendmsg
close
fcntl
ioctl
getsockopt
setsockopt
```

此外，sylar还增加了一个 `connect_with_timeout` 接口用于实现带超时的connect。

为了管理所有的socket fd，sylar设计了一个FdManager类来记录所有分配过的fd的上下文，这是一个单例类，每个socket fd上下文记录了当前fd的读写超时，是否设置非阻塞等信息。

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。

sylar hook模块实现

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_initied = false;
    if(is_initied) {
        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; \
    ... \
    setsockopt_fun setsocket_f = nullptr;
};

hook_init() {
    ...

```

```

sleep_f = (sleep_fun)dlsym(RTLD_NEXT, "sleep"); \
usleep_f = (usleep_fun)dlsym(RTLD_NEXT, "usleep"); \
...
setsockopt_f = (setsockopt_fun)dlsym(RTLD_NEXT, "setsockopt");
}

```

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。这里是sylar条件定时器的巧妙应用，自行体会，感觉说得不是很清楚。

接下来是accept和read/write/recv/send等IO接口的hook实现，这里sylar又一次充分发挥了懒得写代码的本事，用一个do_io模板函数将全部情况都囊括了进来。do_io模板函数的实现与上面的connect_with_timeout实现基本一致，都借助了条件定时器和READ/WRITE事件，这里我也懒得写了，自行看代码。

最后是一些边边角角的情况，有以下几个要注意：

- close，这里除了要删除fd的上下文，还要取消掉fd上的全部事件，这会让fd的读写事件回调都执行一次。
- fcntl，这里的O_NONBLOCK标志要特殊处理，因为所有参与协程调度的fd都会被设置成非阻塞模式，所以要在应用层维护好用户设置的非阻塞标志。
- ioctl，同样要特殊处理FIONBIO命令，这个命令用于设置非阻塞，处理方式和上面的fcntl一样。
- setsockopt，这里要特殊处理SO_RECVTIMEO和SO_SNDFTIMEO，在应用层记录套接字的读写超时，方便协程调度器获取。

注意事项

- 由于定时器模块只支持毫秒级定时，所以被hook后的nanosleep()实际精度只能达到毫秒级，而不是纳秒级。
- 按照 man 2 socket 的描述，自2.6.27版本的内核开始socket函数支持直接在type中位或SOCK_NONBLOCK标志位以创建非阻塞套接字，sylar的hook模块未处理这种情况。
- 按sylar hook模块的实现，非调度线程不支持启用hook。

无标签

Address模块

由 zhongluqiang 创建, 最后修改于 12月 07, 2021

Address模块概述

提供网络地址相关的类, 支持与网络地址相关的操作, 一共有以下几个类:

Address : 所有网络地址的基类, 抽象类, 对应sockaddr类型, 但只包含抽象方法, 不包含具体的成员。除此外, Address作为地址类还提供了网络地址查询及网卡地址查询功能。

IPAddress : IP地址的基类, 抽象类, 在Address基础上, 增加了IP地址相关的端口以及子网掩码、广播地址、网段地址操作, 同样是只包含抽象方法, 不包含具体的成员。

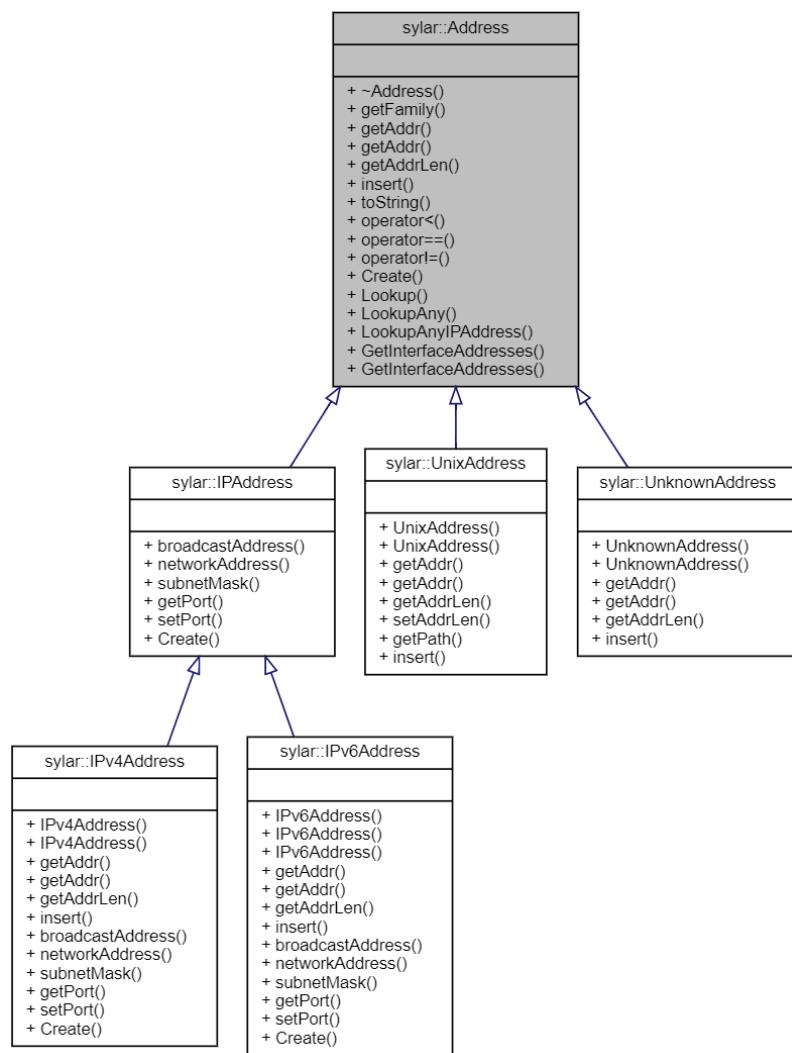
IPv4Address : IPv4地址类, 实体类, 表示一个IPv4地址, 对应sockaddr_in类型, 包含一个sockaddr_in成员, 可以操作该成员的网络地址和端口, 以及获取子网掩码等操作。

IPv6Address : IPv6地址类, 实体类, 与IPv4Address类似, 表示一个IPv6地址, 对应sockaddr_in6类型, 包含一个sockaddr_in6成员。

UnixAddress : Unix域套接字类, 对应sockaddr_un类型, 同上。

UnknownAddress : 表示一个未知类型的套接字地址, 实体类, 对应sockaddr类型, 这个类型与Address类型的区别是它包含一个sockaddr成员, 并且是一个实体类。

整个网络地址模块的类继承关系图如下:



关于套接字地址

Linux使用**Berkeley套接字接口**进行网络编程, 这套接口是事实上的标准网络套接字编程接口, 在基本所有的系统上都支持。Berkeley套接字接口提供了一系列用于网络编程的通用API, 通过这些API可以实现跨主机之间网络通信, 或是在本机上通过Unix域套接字进行进程间通信。

几乎所有的(Berkeley)套接字接口都需要传入一个地址参数(比如在connect或send时指定对端的地址), 用于表示网络中的一台主机的通信地址。不同的协议类型对应的地址类型不一样, 比如IPv4协议对应IPv4地址, 长度是32位, 而IPv6协议对应IPv6地址, 长度是128位, 又比如Unix域套接字地址是一个路径字符串。

如果针对每种类型的地址都制定一套对应的API接口, 那么最终的套接字API接口数量规模一定会非常庞大, 这对开发和维护都没好处。我们希望的是只使用一套通用的API接口就能实现各种地址类型的操作, 比如针对IPv4地址的代码, 能够在不修改或尽量少修改的前提下, 就可用于IPv6地址。对此, Berkeley套接字接口拟定了一个通用套接字地址结构sockaddr, 用于表示任意类型的地址, 所有的套接字API在传入地址参数时都只需要传入sockaddr类型, 以保证接口的通用性。除通用地址结构sockaddr外, 还有一系列表示具体的网络地址的结构, 这些具体的网络地址结构用于用户赋值, 但在使用时, 都要转化成sockaddr的形式。

sockaddr表示通用套接字地址结构, 其定义如下:

```
struct sockaddr
{
    unsigned short sa_family; // 地址族, 也就是地址类型
    char sa_data[14];         // 地址内容
};
```

所有的套接字API都是以指针形式接收sockaddr参数，并且额外需要一个地址长度参数，这可以保证当sockaddr本身不足以容纳一个具体的地址时，可以通过指针取到全部的内容。比如上面的地址内容占14字节，这并不足以容纳一个128位16字节的IPv6地址。但当以指针形式传入时，完全可以通过指针取到适合IPv6的长度。

除sockaddr外，套接字接口还定义了一系列具体的网络地址结构，比如sockaddr_in表示IPv4地址，sockaddr_in6表示IPv6地址，sockaddr_un表示Unix域套接字地址，它们的定义如下：

```
struct sockaddr_in
{
    unsigned short sin_family; // 地址族, IPv4的地址族为AF_INET
    unsigned short sin_port;   // 端口
    struct in_addr sin_addr;  // IP地址, IPv4的地址用一个32位整数来表示
    char sin_zero[8];          // 填充位, 填零即可
};

struct sockaddr_in6
{
    unsigned short sin6_family; // 地址族, IPv6的地址族为AF_INET6
    in_port_t sin6_port;        // 端口
    uint32_t sin6_flowinfo;     // IPv6流控信息
    struct in6_addr sin6_addr;  // IPv6地址, 实际为一个128位的结构体
    uint32_t sin6_scope_id;     // IPv6 scope-id
};

struct sockaddr_un
{
    unsigned short sun_family; // 地址族, Unix域套字地址族为AF_UNIX
    char sun_path[108];        // 路径字符串
};
```

通过上面的定义也可以发现，除sockaddr_in可以无缝转换为sockaddr外，sockaddr_in6和sockaddr_un都不能转换为sockaddr，因为大小不一样。但这并不影响套接字接口的通用性，因为在使用时，所有类型的地址都会转换成**sockaddr指针**形式，又由于以上所有的地址结构的前两个字节都表示地址族，所以通过**sockaddr指针总能拿到传入地址的地址类型**，通过地址类型判断出地址长度后，再通过sockaddr指针取适合该地址的长度即可拿到地址内容。

实际使用时，一般都是先定义具体的网络地址结构并初始化，然后在传入套接字接口时，通过指针强制转化成sockaddr类型，以下分别是使用IPv4地址和IPv6地址发起bind的代码：

IPv4

```
int sockfd;
sockaddr_in addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);

memset(&addr, 0, sizeof(sockaddr_in));
addr.sa_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(80);

bind(sockfd, (sockaddr*)&addr, sizeof(addr)); // bind支持所有的地址类型
                                                // 这里要将IPv4地址结构强制转化成sockaddr*类型，并传入地址长度
```

IPv6

```
int sockfd;
sockaddr_in6 addr;

sockfd = socket(AF_INET6, SOCK_STREAM, 0);

memset(&addr, 0, sizeof(sockaddr_in6));
addr.sa_family = AF_INET6;
addr.sin6_addr.s_addr = in6addr_any;
addr.sin6_port = htons(80);

bind(sockfd, (sockaddr*)&addr, sizeof(addr)); // 这里将IPv6地址也转化成sockaddr*再传入bind
                                                // 虽然sockaddr结构体并不足以容纳一个IPv6地址，但是传入的是指针和地址长度，
                                                // bind内部在判断出当前地址是IPv6地址的情况下，
                                                // 仍然可以通过指针和长度取到一个完整的IPv6地址
```

通过这两段代码也可以看出，虽然IPv4和IPv6的地址结构不一样，但它们在进行bind操作时的形式是一样的，Berkeley套接字接口正是通过这种形式保证了接口在大部分情况下都可以通用。

Address类

这个类是所有网络地址类的基类，并且是一个抽象类，对应的是sockaddr，表示通用网络地址。对于一个通用的网络地址，需要关注它的地址类型，sockaddr指针，地址长度这几项内容。

除此外，Address类还提供了地址解析与本机网卡地址查询的功能，地址解析功能可以实现域名解析，网卡地址查询可以获取本机指定网卡的IP地址。

IPAddress类

继承自Address类，表示一个IP地址，同样是一个抽象类，因为IP地址包含IPv4地址和IPv6地址。IPAddress类提供了IP地址相关的端口和掩码、网段地址、网络地址操作，无论是IPv4还是IPv6都支持这些操作，但这些方法都是抽象方法，需要由继承类来实现。

IPv4Address类

继承自IPAddress类，表示一个IPv4地址，到这一步，IPv4Address就是一个实体类了，它包含一个sockaddr_in类型的成员，并且提供具体的端口设置/获取，掩码、网段、网络地址设置/获取操作。

IPv6Address类

继承自IPAddress类，表示一个IPv6地址，也是一个实体类，实现思路和IPv4Address一致。

UnixAddress类

继承自Address类，表示一个Unix域套接字地址，是一个实体类，可以用于实例化对象。UnixAddress类包含一个sockaddr_un对象以及一个路径字符串长度。

UnknownAddress类

继承自Address类，包含一个sockaddr成员，表示未知的地址类型。

无标签

浙ICP备2021003588号

Socket模块

由 zhongluqiang 创建, 最后修改于 12月 07, 2021

套接字类, 表示一个套接字对象。

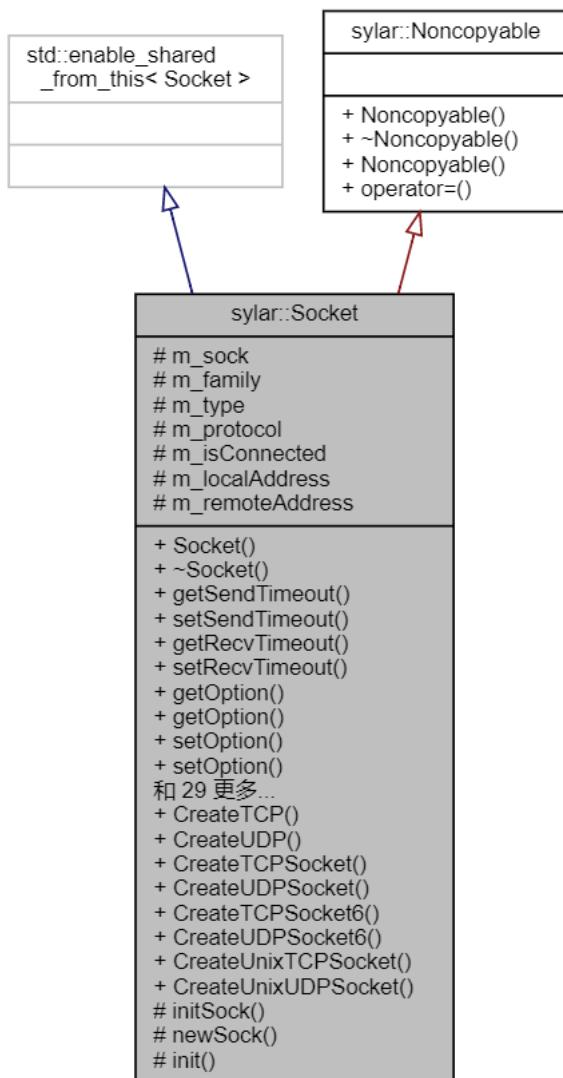
对于套接字类, 需要关注以下属性:

1. 文件描述符
2. 地址类型 (AF_INET, AF_INET6等)
3. 套接字类型 (SOCK_STREAM, SOCK_DGRAM等)
4. 协议类型 (这项其实可以忽略)
5. 是否连接 (针对TCP套接字, 如果是UDP套接字, 则默认已连接)
6. 本地地址和对端的地址

套接字类应提供以下方法:

1. 创建各种类型的套接字对象的方法 (TCP套接字, UDP套接字, Unix域套接字)
2. 设置套接字选项, 比如超时参数
3. bind/connect/listen方法, 实现绑定地址、发起连接、发起监听功能
4. accept方法, 返回连入的套接字对象
5. 发送、接收数据的方法
6. 获取本地地址、远端地址的方法
7. 获取套接字类型、地址类型、协议类型的方法
8. 取消套接字读、写的方法

以下是Socket类的继承关系图:



无标签

ByteArray类

由 zhongluqiang 创建, 最后修改于 11月 28, 2021

字节数组容器, 提供基础类型的序列化与反序列化功能。

ByteArray的底层存储是固定大小的块, 以链表形式组织。每次写入数据时, 将数据写入到链表最后一个块中, 如果最后一个块不足以容纳数据, 则分配一个新的块并添加到链表结尾, 再写入数据。ByteArray会记录当前的操作位置, 每次写入数据时, 该操作位置按写入大小往后偏移, 如果要读取数据, 则必须调用setPosition重新设置当前的操作位置。

ByteArray支持基础类型的序列化与反序列化功能, 并且支持将序列化后的结果写入文件, 以及从文件中读取内容进行反序列化。ByteArray支持以下类型的序列化与反序列化:

1. 固定长度的有符号/无符号8位、16位、32位、64位整数
2. 不固定长度的有符号/无符号32位、64位整数
3. float, double类型
4. 字符串, 包含字符串长度, 长度范围支持16位、32位、64位。
5. 字符串, 不包含长度。

以上所有的类型都支持读写。

ByteArray还支持设置序列化时的大小端顺序。

浅谈序列化

zigzag算法

用于压缩较小的整数, 参考: [小而巧的数字压缩算法: zigzag_简单的老王-CSDN博客_zigzag编码](#)。

ByteArray在序列化不固定长度的有符号/无符号32位、64位整数时使用了zigzag算法。

TLV编码结构

用于序列化和消息传递, 指Tag (类型), Length (长度), Value (值), 参考: [TLV编码通信协议设计 - Tango 博客 | Tango Blog](#)。

ByteArray在序列化字符串时使用TLV中的Length和Value。

无标签

浙ICP备2021003588号

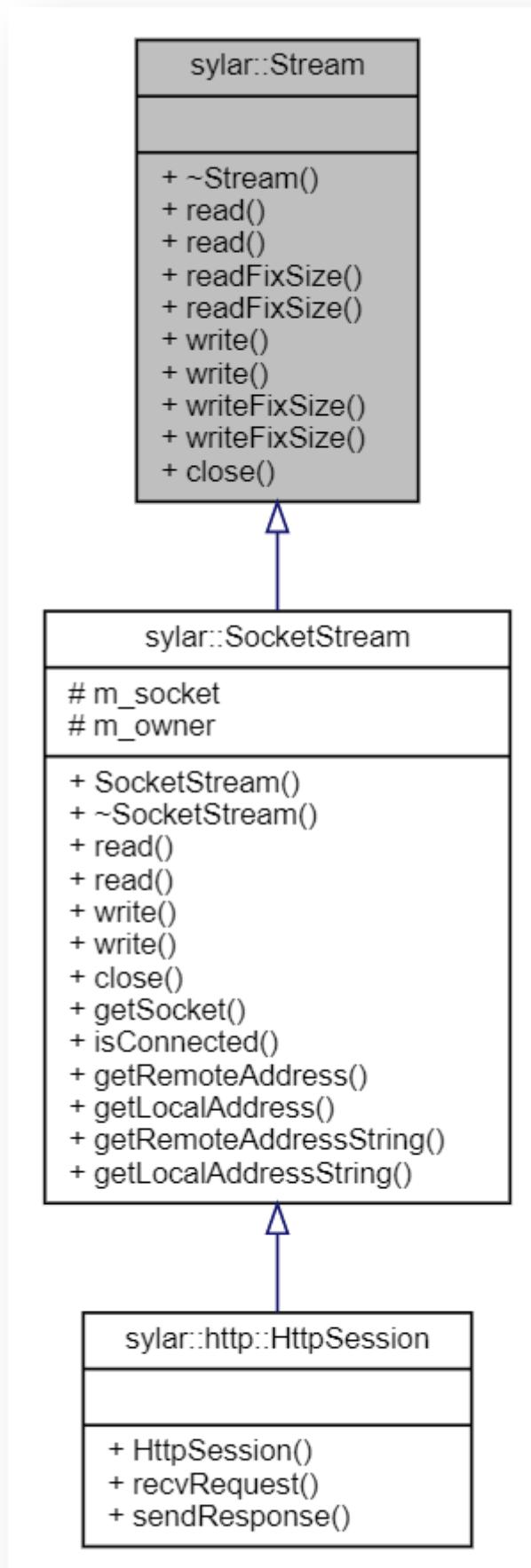
Stream模块

由 zhongluqiang创建, 最后修改于12月 16, 2021

流结构，提供字节流读写接口。

所有的流结构都继承自抽象类Stream，Stream类规定了一个流必须具备read/write接口和readFixSize/writeFixSize接口，继承自Stream的类必须实现这些接口。

以下是Stream模块的继承关系图：



SocketStream类

套接字流结构，将套接字封装成流结构，以支持Stream接口规范，除此外，SocketStream还支持套接字关闭操作以及获取本地/远端地址的操作。

无标签

浙ICP备2021003588号 ·

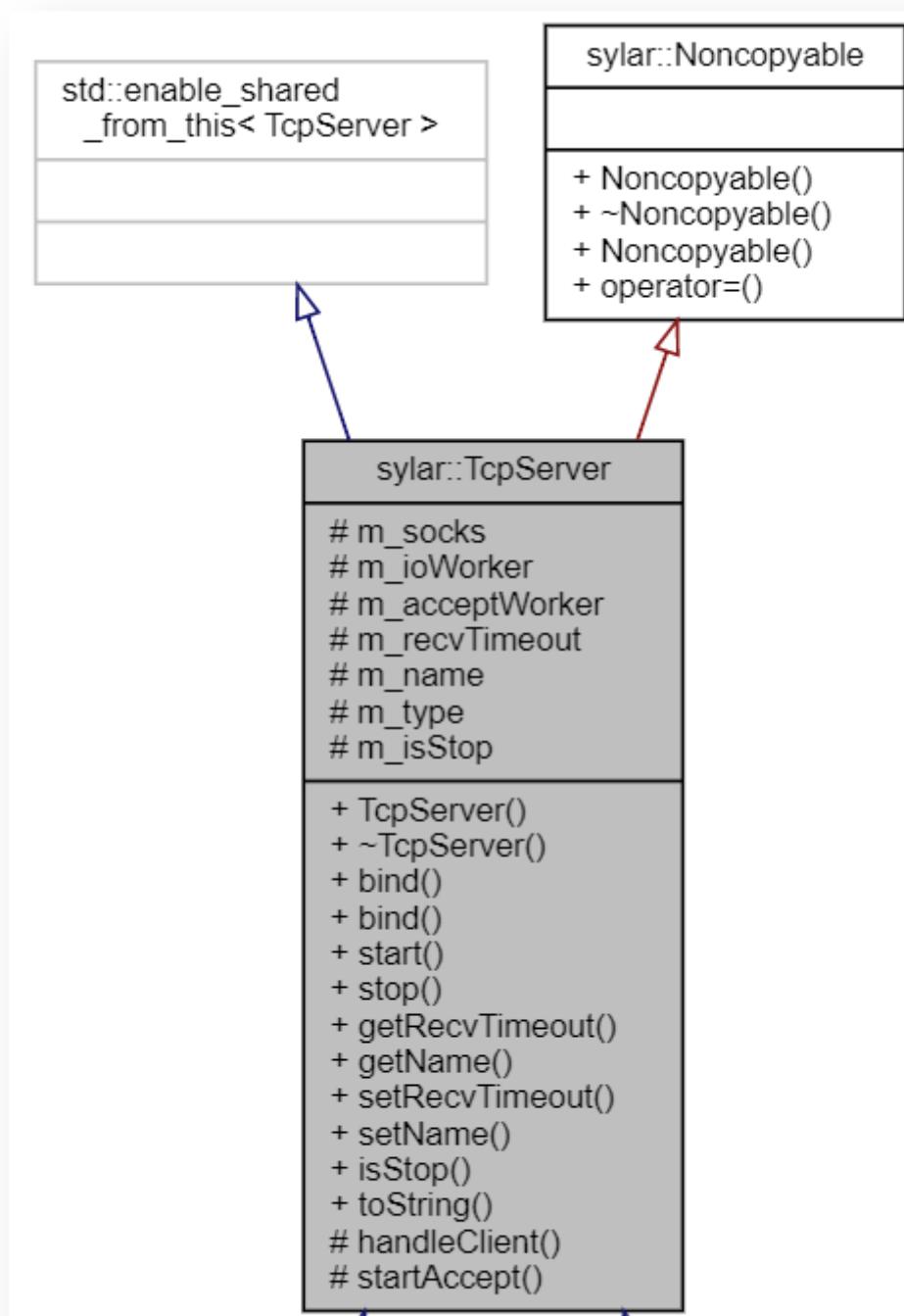
TcpServer类

由 zhongluqiang 创建于 11月 28, 2021

TCP服务器封装。

TcpServer类支持同时绑定多个地址进行监听，只需要在绑定时传入地址数组即可。 TcpServer还可以分别指定接收客户端和处理客户端的协程调度器。

以下是TcpServer的继承关系图：



TcpServer类采用了Template Pattern设计模式，它的HandleClient是交由继承类来实现的。使用 TcpServer时，必须从TcpServer派生一个新类，并重新实现子类的handleClient操作，这点可以参考 test_tcp_server.cc。

无标签

浙ICP备2021003588号 ·

HTTP模块

由 zhongluqiang 创建, 最后修改于 12月 09, 2021

提供HTTP服务，主要包含以下几个模块：

1. HTTP常量定义，包括HTTP方法 HttpMethod 与HTTP状态 HttpStatus。
2. HTTP请求与响应结构，对应 HttpRequest 和 HttpResponse。
3. HTTP解析器，包含HTTP请求解析器与HTTP响应解析器，对应 HttpRequestParser 和 HttpResponseParser。
4. HTTP会话结构，对应 HttpSession。
5. HTTP服务器。
6. HTTP Servlet。
7. HTTP客户端 HttpConnection，用于发起GET/POST等请求，支持连接池。

HTTP模块依赖[nodejs/http-parser](#)提供的HTTP解析器，并且直接复用了nodejs/http-parser中定义的HTTP方法与状态枚举。

HTTP常量定义

包括HttpMethod和HttpStatus两个定义，如下：

http_parser/http_parser.h

```
/* Request Methods */
#define HTTP_METHOD_MAP(XX) \
    XX(0, DELETE, DELETE) \
    XX(1, GET, GET) \
    XX(2, HEAD, HEAD) \
    XX(3, POST, POST) \
    XX(4, PUT, PUT) \
    ... \
    \
/* Status Codes */
#define HTTP_STATUS_MAP(XX) \
    XX(100, CONTINUE, Continue) \
    XX(101, SWITCHING_PROTOCOLS, Switching Protocols) \
    XX(102, PROCESSING, Processing) \
    XX(200, OK, OK) \
    XX(201, CREATED, Created) \
    XX(202, ACCEPTED, Accepted) \
    XX(203, NON_AUTHORITATIVE_INFORMATION, Non-Authoritative Information) \
    ...
```

http.h

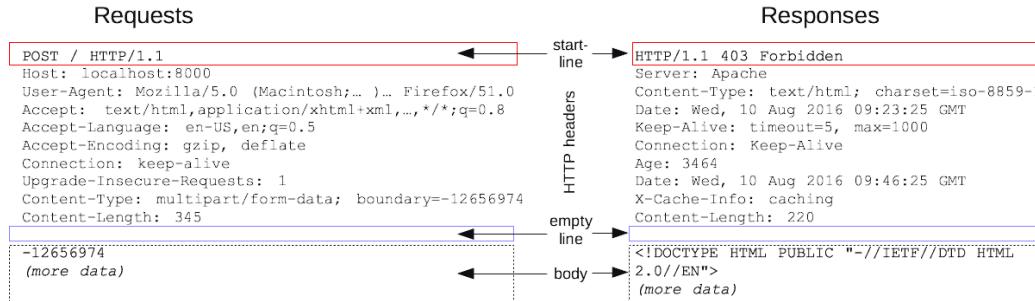
```
/** 
 * @brief HTTP方法枚举
 */
enum class HttpMethod {
#define XX(num, name, string) name = num,
    HTTP_METHOD_MAP(XX)
#undef XX
    INVALID_METHOD
};

/** 
 * @brief HTTP状态枚举
 */
enum class HttpStatus {
#define XX(code, name, desc) name = code,
    HTTP_STATUS_MAP(XX)
#undef XX
};
```

HTTP请求与响应结构

包括HttpRequest和HttpResponse两个结构，用于封装HTTP请求与响应。

关于HTTP请求和响应的格式可参考[HTTP消息 - HTTP | MDN](#)，以下是一个HTTP请求与响应的示例：



对于HTTP请求，需要关注HTTP方法，请求路径和参数，HTTP版本，HTTP头部的key-value结构，Cookies，以及HTTP Body内容。

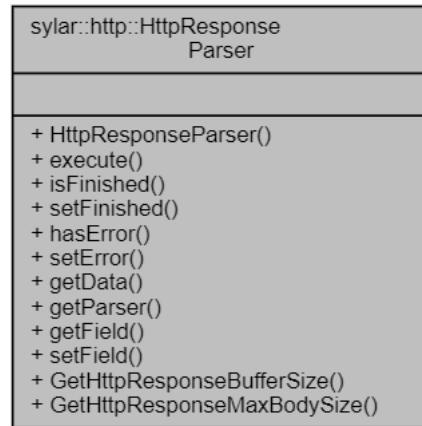
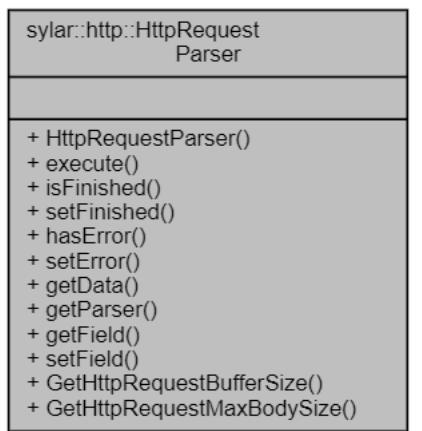
对于HTTP响应，需要关注HTTP版本，响应状态码，响应字符串，响应头部的key-value结构，以及响应的Body内容。

HTTP解析器

输入字节流，解析HTTP消息，包括HttpRequestParser和HttpResponseParser两个结构。

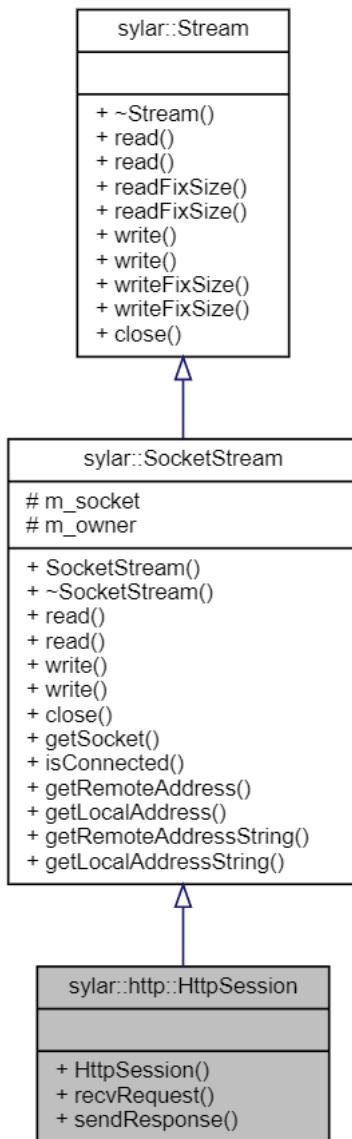
HTTP解析器基于[nodejs/http-parser](#)实现，通过套接字读到HTTP消息后将消息内容传递给解析器，解析器通过回调的形式通知调用方HTTP解析的内容。

以下是HTTP解析器的类协作图：



HTTP会话结构 HttpSession

继承自`SocketStream`，实现了在套接字流上读取HTTP请求与发送HTTP响应的功能，在读取HTTP请求时需要借助HTTP解析器，以便于将套接字流上的内容解析成HTTP请求。以下是HttpSession的继承关系图：



HTTP服务器

继承自TcpServer，重载handleClient方法，将accept后得到的客户端套接字封装成 HttpSession 结构，以便于接收和发送HTTP消息。

```

29 ~ void HttpServer::handleClient(Socket::ptr client) {
30     SYLAR_LOG_DEBUG(g_logger) << "handleClient " << *client;
31     HttpSession::ptr session(new HttpSession(client));
32     do {
33         auto req = session->recvRequest();
34         if(!req) {
35             SYLAR_LOG_DEBUG(g_logger) << "recv http request fail, errno="
36             << errno << " errstr=" << strerror(errno)
37             << " client:" << *client << " keep_alive=" << m_isKeepalive;
38             break;
39         }
40
41         HttpResponse::ptr rsp(new HttpResponse(req->getVersion()
42             ,req->isClose() || !m_isKeepalive));
43         rsp->setHeader("Server", getName());
44         m_dispatch->handle(req, rsp, session);
45         session->sendResponse(rsp);
46
47         if(!m_isKeepalive || req->isClose()) {
48             break;
49         }
50     } while(true);
51     session->close();
52 }
  
```

HTTP Servlet

提供HTTP请求路径到处理类的映射，用于规范化的HTTP消息处理流程。

HTTP Servlet包括两部分，第一部分是Servlet对象，每个Servlet对象表示一种处理HTTP消息的方法，第二部分是ServletDispatch，它包含一个请求路径到Servlet对象的映射，用于指定一个请求路径该用哪个Servlet来处理。

以下是Servlet对象用于处理请求的声明：

```
45  /**
46   * @brief 处理请求
47   * @param[in] request HTTP请求
48   * @param[in] response HTTP响应
49   * @param[in] session HTTP连接
50   * @return 是否处理成功
51   */
52   virtual int32_t handle(sylar::http::HttpRequest::ptr request
53   , sylar::http::HttpResponse::ptr response
54   , sylar::http::HttpSession::ptr session) = 0;
```

HTTP客户端HttpConnection

用于发起GET/POST等请求并获取响应，支持设置超时，keep-alive，支持连接池。

HTTP服务端的业务模型是接收请求→发送响应，而HTTP客户端的业务模型是发送请求→接收响应。

关于连接池，是指提前准备好一系列已接建立连接的socket，这样，在发起请求时，可以直接从中选择一个进行通信，而不用重复创建套接字→发起connect→发起请求的流程。

连接池与发起请求时的keep-alive参数有关，如果使用连接池来发起GET/POST请求，在未设置keep-alive时，连接池并没有什么卵用。

无标签

守护进程

由 zhongluqiang 创建于 12月 10, 2021

将进程与终端解绑，转到后台运行，除此外，sylar还实现了双进程唤醒功能，父进程作为守护进程的同时会检测子进程是否退出，如果子进程退出，则会定时重新拉起子进程。

以下是守护进程的实现步骤：

1. 调用daemon(1, 0)将当前进程以守护进程的形式运行；
2. 守护进程fork子进程，在子进程运行主营业务；
3. 父进程通过waitpid()检测子进程是否退出，如果子进程退出，则重新拉起子进程；

```

37     static int real_daemon(int argc, char** argv,
38     | | | | std::function<int(int argc, char** argv)> main_cb) {
39     | | | | | daemon(1, 0);
40     | | | | | ProcessInfoMgr::GetInstance()->parent_id = getpid();
41     | | | | | ProcessInfoMgr::GetInstance()->parent_start_time = time(0);
42     | | | | | while(true) {
43     | | | | | | pid_t pid = fork();
44     | | | | | | if(pid == 0) {
45     | | | | | | | //子进程返回
46     | | | | | | | ProcessInfoMgr::GetInstance()->main_id = getpid();
47     | | | | | | | ProcessInfoMgr::GetInstance()->main_start_time = time(0);
48     | | | | | | | SYLAR_LOG_INFO(g_logger) << "process start pid=" << getpid();
49     | | | | | | | return real_start(argc, argv, main_cb);
50     | | | | | } else if(pid < 0) {
51     | | | | | | SYLAR_LOG_ERROR(g_logger) << "fork fail return=" << pid
52     | | | | | | | << " errno=" << errno << " errstr=" << strerror(errno);
53     | | | | | | | return -1;
54     | | | | | } else {
55     | | | | | | //父进程返回
56     | | | | | | | int status = 0;
57     | | | | | | | waitpid(pid, &status, 0);
58     | | | | | | | if(status) {
59     | | | | | | | | SYLAR_LOG_ERROR(g_logger) << "child crash pid=" << pid
60     | | | | | | | | << " status=" << status;
61     | | | | | | } else {
62     | | | | | | | SYLAR_LOG_INFO(g_logger) << "child finished pid=" << pid;
63     | | | | | | | break;
64     | | | | | | } }
65     | | | | | | | ProcessInfoMgr::GetInstance()->restart_count += 1;
66     | | | | | | | sleep(g_daemon_restart_interval->getValue());
67     | | | | | | }
68     | | | | | }
69     | | | | | return 0;
70   }

```

无标签

