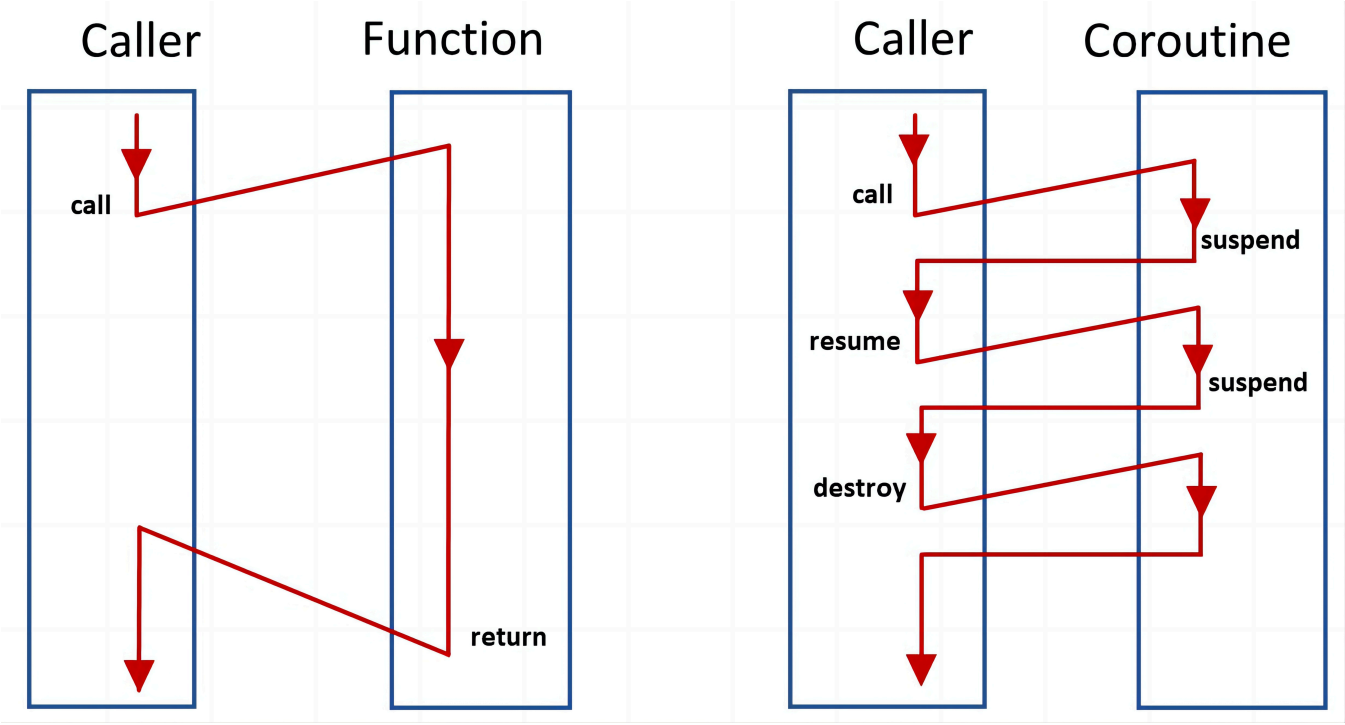
协程调度框架

协程调度框架基于C++20协程标准实现，实现基本的任务、定时器、网络IO、文件IO、通知调度功能。
SVN路径: https://172.16.32.113/svn/CSM2020/trunk/02设计与实现/02软件开发/09其他/个人工作目录/qinrui/代码/framework

协程(Coroutine)

协程其实也是一个函数，不同于普通函数，协程函数内部可以挂起、等待、恢复。
协程函数与普通函数比较如下图所示:



C++20协程

C++20的协程设计为无栈协程，相对于有栈协程，省掉了上下文切换开销，只能手动切换，效率更高，也不用管理复杂的寄存器状态，移植性更好，但这同时也导致了不能被非协程函数嵌套调用。

C++20引入了三个协程关键字:

```
co_yield 挂起并返回值
co_await 挂起
co_return 结束协程
```

Promise

Promise是协程的状态管理，在堆上管理本身的状态信息（形参，局部变量，自带数据，各个阶段点执行点）。

Future

Future对象主要是与Promise对象交互，进行调用者（caller）与被调用者（callee）之间的通信。

Awaitable

Awaitable是一个可等待的对象，可以使用co_await去触发该等待对象的await_ready、await_suspend、await_resume函数，进行控制权的恢复、转移。

具体机制

Promise/Future

当调用一个协程函数时，协程会创建Promise对象，并在编译器编译阶段会在协程函数各个阶段插入关键代码。

```
{
    // 1. 在堆上创建promise对象

    // 2. initial
    co_await promise.initial_suspend();
    try{
        // 3. 执行函数本身代码
        // 函数本身代码
    }
    catch(...)
    {
        // 4. 如果有异常，进入unhandle_exception阶段
        promise.unhandle_exception();
    }

    // 5. 进入final阶段
    promise.final_suspend();
}
```

Promise需要实现一下接口：

```
initial_suspend: 返回一个Awaitable对象
final_suspend: 返回一个Awaitable对象
get_return_object: 返回一个Future对象给调用者（caller）
unhandled_exception: 处理异常
return_value/return_void: co_return时返回值给调用者（caller）
yield_value: 挂起时返回值给调用者（caller）
```

Future对象持有Promise对象，使用std::coroutine_handle<Promise>协程句柄包装。
std::coroutine_handle<Promise>对象包含一下方法：

```
destroy: 销毁Promise对象
from_promise: 静态方法，从Promise对象返回其coroutine_handle协程句柄
done: 是否处于final_suspend阶段
promise: 返回Promise对象引用
resume/operator(): 恢复到协程
```

Awaitable对象

co_await 该关键词操作的对象就是一个Awaitable。
Awaitable对象需要实现以下方法:

```
await_ready()
await_suspend(coroutine_handle<>)
await_resume()
```

当执行到co_await <expr>代码时，编译器会生成如下代码:

```
{
    auto &&value = <expr>;
    // 1. 获取等待对象
    auto &&awaitable = get_awaitable(promise, static_cast<decltype(value)>(value));
    auto &&awaiter = get_awaiter(static_cast<decltype(awaitable)>(awaitable));
    // 2. 判断是否等待
    if(!awaiter.await_ready())
    {
        {
            // 3.1 如果awaiter.await_suspend的返回值是void
            try
            {
                awaiter.await_suspend(coroutine_handle);
            }
            catch(...)
            {
                exception = std::current_exception();
                goto resumePoint;
            }

            // 本身挂起，控制权交给调用者
        }

        {
            // 3.2 如果awaiter.await_suspend的返回值是bool
            bool result;
            try
            {
                result = awaiter.await_suspend(coroutine_handle);
            }
            catch(...)
            {
                exception = std::current_exception();
                goto resumePoint;
            }

            if(!result)
            {
                goto resumePoint;
            }

            // 本身挂起，控制权交给调用者
        }

        {
            // 3.3 如果awaiter.await_suspend的返回值是std::coroutine_handle<...>
```

```
            std::coroutine_handle<...> handle;
            try
            {
                handle = awaiter.await_suspend(coroutine_handle);
            }
            catch(...)
            {
                exception = std::current_exception();
                goto resumePoint;
            }

            handle.resume();
            // 本身挂起，控制权交给调用者
        }
    }

resumePoint:
    // 4. 如果有异常，重新抛出异常
    if(exception)
    {
        std::rethrow_exception(exception);
    }

    // 5. 调用await_resume 将返回值给co_await，co_await 执行完毕
    return awaiter.await_resume();
}
```

co_yield <expr> 其实就是调用了promise.yeild_value(expr);

```
co_await promise.yield_value(expr);
```

co_return <expr> 其实就是调用了promise.return_void/return_value函数

框架实现

Coroutine

```
#include "co/co_coroutine.h"
```

Coroutine对象是一个Future对象，包含了Promise对象。

Coroutine::promise_type对象是一个Promise对象，实现了协程各个阶段的控制。

```
// Promise 对象
template <typename T>
struct PromiseTypeBase
{

#ifdef _CO_USE_EXCEPTION
    bool interrupt;
    bool exception;
#endif

    size_t ref_count;
    // 持有的协程句柄
    std::coroutine_handle<> parent_handle;

    void reset()
    {
#ifdef _CO_USE_EXCEPTION
        exception = false;
        interrupt = false;
#endif
        ref_count = 1;
    }

    void retain()
    {
        ++ref_count;
    }

    void release(std::coroutine_handle<> co_handle)
    {
        assert(ref_count > 0);
        --ref_count;

        if (ref_count == 0)
        {
            assert(isLoopThread());
            CoHandle *handle = new CoHandle();
            handle->handle = co_handle;
            DLListTailAdd(&threadLoop()->coroutineQueue, &handle->node);
        }
    }

    // 生成Futuer对象
    auto get_return_object()
    {
        reset();
```

```
        return Coroutine<T>{ HandleType<T>::from_promise(static_cast<PromiseType<T> &>(*this)) };
    }

    // initial阶段的等待对象，永远不会挂起，当协程函数被调用时，不会挂起在initial阶段，直接执行用户代码。
    auto initial_suspend()
    {
        return std::suspend_never{};
    }

    // final阶段的等待对象
    auto final_suspend() noexcept
    {
        struct Awaitable
        {
            // 返回false 始终挂起协程，等待调用者调用destroy函数销毁协程。
            bool await_ready() noexcept
            {
                return false;
            }

            // 返回协程句柄 如果有父协程，则恢复父协程，否则恢复给调用者。
            auto await_suspend(HandleType<T> co_handle) noexcept
            {

#if __has_include(<coroutine>)
                auto parent = co_handle.promise().parent_handle;
                co_handle.promise().release(co_handle);
                return parent ? parent : std::noop_coroutine();
#else
                auto parent = co_handle.promise().parent_handle;
                co_handle.promise().release(co_handle);

                if (parent)
                {
                    parent.resume();
                }
#endif
            }

            // 无返回值
            void await_resume() noexcept
            {
            }
        };

        return Awaitable{};
```

```
    }

    // 终止异常程序
    void unhandled_exception()
    {

#ifdef _CO_USE_EXCEPTION
        exception = true;
        co_exception_handler();
#else
        std::terminate();
#endif

    }
};

template <typename T>
struct PromiseTypeNoVoid : public PromiseTypeBase<T>
{
    T value;

    // 返回协程的值，使用std::move减少拷贝
    void return_value(T v)
    {
        value = std::move(v);
    }
};

template <typename T>
struct PromiseTypeVoid : public PromiseTypeBase<T>
{
    // 无返回的的协程返回
    void return_void()
    {
    }
};


// Future对象
template <typename T>
struct Coroutine
{
    using promise_type = PromiseType<T>;

    Coroutine(HandleType<T> h) : handle{ h }
    {
```

```
        assert(isLoopThread());
        ++threadLoop()->activeCoroutines;

        handle.promise().retain();
    }

    ~Coroutine()
    {
        if (handle)
        {
            handle.promise().release(handle);
        }
    }

    Coroutine(const Coroutine &) = delete;

    Coroutine &operator=(const Coroutine &) = delete;

    Coroutine(Coroutine &&other) noexcept : handle{ other.handle }
    {
        other.handle = nullptr;
    }

    Coroutine &operator=(Coroutine &&other) noexcept
    {
        if (this == &other)
        {
            return *this;
        }

        handle = other.handle;
        other.handle = nullptr;

        return *this;
    }

    // 如果本协程已经处于final阶段，则不用再等待，直接返回值
    bool await_ready()
    {
        return handle.done();
    }

    // 记录父协程的句柄，然后挂起，返回给调用者
    void await_suspend(std::coroutine_handle<> co_handle)
    {
        handle.promise().parent_handle = co_handle;
```

```
    }

    // 返回值，使用std::move减少拷贝
    decltype(auto) await_resume()
    {

#ifdef _CO_USE_EXCEPTION
        if (handle.promise().exception)
        {
            throw CoInterruptException();
        }

        if (co_is_interrupt(handle.promise().parent_handle))
        {
            throw CoInterruptException();
        }
#endif

        if constexpr (std::is_void_v<T>)
        {
            return;
        }
        else
        {
            return std::move(handle.promise().value);
        }
    }

#ifdef _CO_USE_EXCEPTION
    void interrupt()
    {
        handle.promise().interrupt = true;
    }
#endif

    HandleType<T> handle;
};
```

使用方法: 返回值为Coroutine<T>的函数就时一个协程函数，该函数可以使用co_await关键字挂起。

```
Coroutine<int> add(int a, int b)
{
    co_return a + b;
}

Coroutine<void> cal()
{
    int a = co_await add(1, 2);
    assert(a == 3);

    auto co_add = add(2, 3);
    int b = co_await co_add;
    assert(b == 5);
}
```

事件循环(Loop)

事件循环实现了协程管理、定时器的计数、任务、IO任务调度等功能。

协程调度框架运行在事件循环中，所有的协程均运行在事件循环中，所以事件循环中的所有协程是同步的。

```
while (true)
{
    // 1. 更新事件
    loopUpdateTime(loop);

    // 2. 执行定时器
    loopRunTimers(loop);

    // 3. 执行异步请求、IO请求
    loopRunRequests(loop);

    // 4. 协程管理，协程的销毁
    loopRunCoroutines(loop);

    // 5. 判断是否退出 当程序中无协程运行时，事件循环将退出
    if (!loopAlive(loop))
    {
        break;
    }

    // 6. 轮询事件，比如文件IO、网络IO
    timeout = loopBackendTimeout(loop);

    if (pGetQueuedCompletionStatusEx)
    {
        loopPoll(loop, timeout);
    }
    else
    {
        loopPollWin(loop, timeout);
    }
}
```

使用方法:

```
// 1. 设置该线程为事件循环线程 也就是协程调度线程
setLoopThread();

// 2. 初始化事件循环
Loop *loop = threadLoop();
loopInit(loop);

// 3. 执行协程函数
// TODO: 调用协程函数

// 4. 开启事件循环
loopRun(loop);

// 5. 关闭事件循环
loopClose(loop);
```

协程管理

当协程创建协程计数加一。

```
Coroutine(handle_type h) : handle{h}
{
    // 添加计数
    assert(isLoopThread());
    ++threadLoop()->activeCoroutines;

    handle.promise().retain();
}
```

当协程进入final阶段并未被Coroutine<T>对象持有时，协程会被加入待销毁的协程链表。

```
// 协程销毁链表
assert(isLoopThread());
CoHandle *handle = new CoHandle();
handle->handle = co_handle;
DLListTailAdd(&threadLoop()->coroutineQueue, &handle->node);
```

在事件循环中统一被销毁（destroy）。

```
// 销毁协程
CoHandle *handle = (CoHandle *)DLListTakeFirst(&list);
handle->handle.destroy();
delete handle;
--loop->activeCoroutines;
```

定时器事件(LoopTimer)

定时器是在事件循环中计数，超时触发的事件。内部使用MinHeap。

开启定时器

```
// 1.设定定时器参数
timer->delayUs = delayUs;
timer->timeoutUs = timer->loop->timeUs + delayUs;
timer->callback = callback;
timer->id = timer->loop->timerCount++;

// 插入定时器MinHeap
heapInsert(&timer->loop->timerHeap, &timer->timerNode, timerLessThan);
```

执行定时器

```
HeapNode *node = nullptr;
LoopTimer *timer = nullptr;
while (nullptr != (node = heapMin(&loop->timerHeap)))
{
    // 1. 从MinHeap取出来最小的定时器
    timer = MEMBER_TO_TYPE(node, LoopTimer, timerNode);

    if (timer->timeoutUs > loop->timeUs)
    {
        // 2. 定时器未超时，不再进行定时器处理
        break;
    }

    // 2. 停止定时器
    loopTimerStop(timer);

    // 3. 调用定时器回调
    timer->callback(timer);
}
```

IO请求(LoopIo)

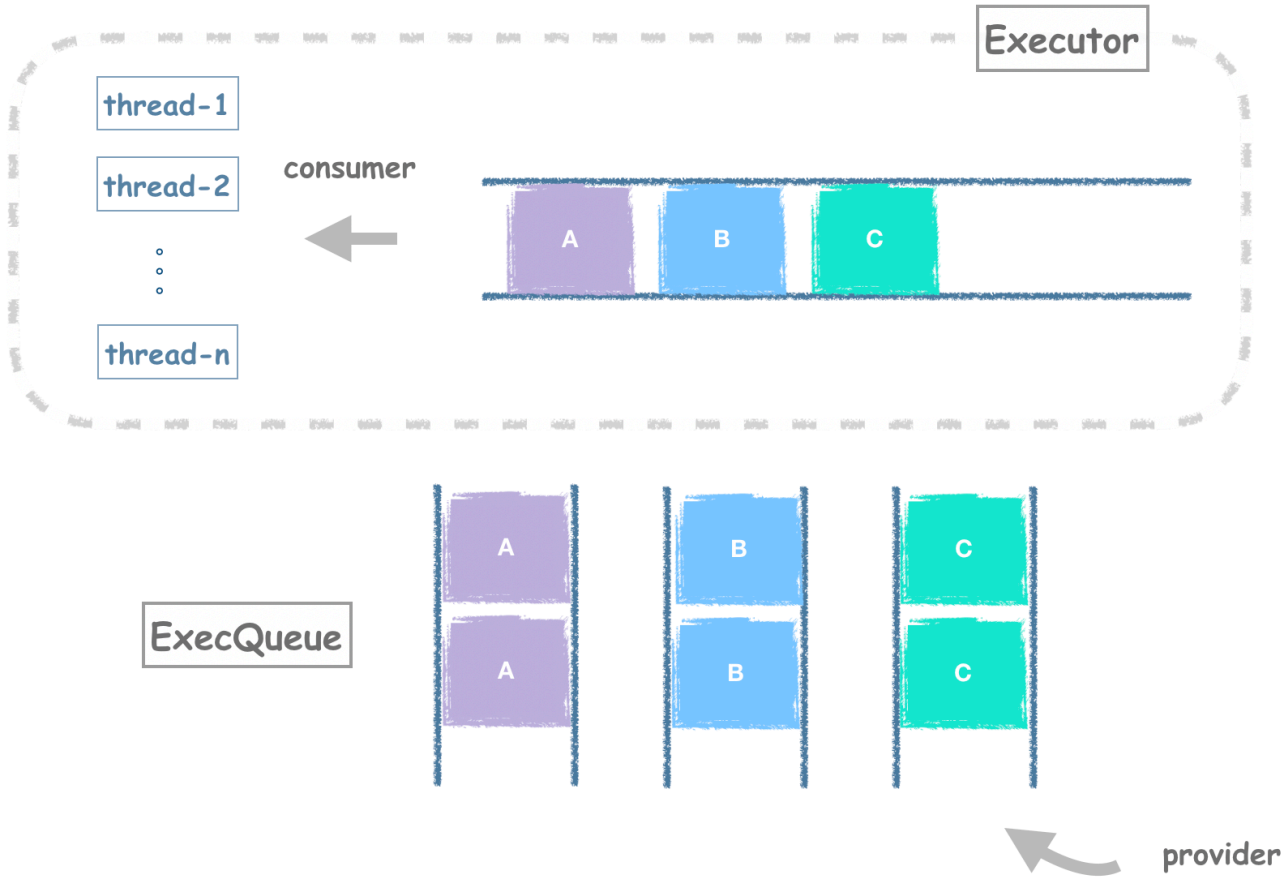
IO请求使用Windows重叠IO（Overlapped）和完成端口（IOCP）实现。

LoopIo需要与Windows异步IO请求函数共同协作，比如： ReadFile WriteFile WSASend WSARecv等函数。

任务处理(LoopWorker)

任务处理内部使用线程池、内存池实现，通过LoopAsync与事件循环(Loop)通信。

任务调度采用多维调度队列，做到对每个执行队列公平调度。线程池中有一个消费队列，线程池之外有N个执行队列，执行队列中永远最多只有一个任务会被放进消费队列中。如下图所示:



Workflow - Computing and Scheduling Architecture

使用多维调度算法可以满足以下特点:

1. 如果有空闲线程，同一执行队列的任务也能实时调度。
2. 如果没有空闲线程，则同一执行队列的任务按照FIFO顺序进入执行队列。执行队列之间平等对待。
3. 执行队列可以无数个。

任务调度(Worker)

CoWorker

CoWorker是一个Waitable对象。使用事件循环中的任务调度实现await_ready、await_suspend、await_resume接口。

co_async

返回一个异步执行的工作等待对象(CoWorker)。

```
#include "co/co_worker.h"

int add(int a, int b)
{
    return a + b;
}

WorkerQueue queue;
int result = co_await co_async(&queue, add, 1, 2);
assert(result == 3);
```

co_sync

返回一个同步执行的工作等待对象。

```
int add(int a, int b)
{
    return a + b;
}

WorkerQueue queue;
int result = co_await co_async(&queue, add, 1, 2);
assert(result == 3);
```

coro_async

返回一个异步执行的协程对象(Coroutine<T>)

```
int add(int a, int b)
{
    return a + b;
}

// 1. 直接等待结果 和 co_async效果一样
WorkerQueue queue;
int result = co_await coro_async(&queue, add, 1, 2);
assert(result == 3);

// 2. 稍后等待结果
auto co_add = coro_async(&queue, add, 1, 2);
// 做其他事情
co_await co_add;

// 3. 不等待结果
coro_async(&queue, add, 1, 2);
```

coro_sync

返回一个同步执行的协程对象(Coroutine<T>)

```
int add(int a, int b)
{
    return a + b;
}

// 1. 直接等待结果 和 co_sync效果一样
WorkerQueue queue;
int result = co_await coro_sync(&queue, add, 1, 2);
assert(result == 3);

// 2. 稍后等待结果
auto co_add = coro_sync(&queue, add, 1, 2);
// 做其他事情
co_await co_add;

// 3. 不等待结果
coro_sync(&queue, add, 1, 2);
```

注意：

1. 可等待对象必须co_await，不然不会触发协程。
2. 协程可以不等待结果。

定时器(Timer)

co_sleep

返回一个延时的等待对象。单位ms

```
#include "co/co_sleep.h"
// 延时1000ms
co_await co_sleep(1000);
```

coro_sleep

返回一个延时的协程。单位ms

```
// 1. 直接等待 延时1000ms 同co_sleep
co_await coro_sleep(1000);

// 2. 稍后等待 延时1000ms 同co_sleep
auto co_delay = coro_sleep(1000);
// 做其他事情
co_await co_delay;
```

通知(Notify)

CoNotify

通知使用Awaitable挂起、恢复特性实现。

```cpp
#include "co/notify/co_notify.h"
// 1. wait
CoNotify notify;
bool success = co_await notify.wait(-1);

// 其他的协程函数
// 2. notify
notify.notify(true);
```

文件(File)

CoFile

文件使用Windows重叠IO、完成端口和LoopIo实现。

```cpp
#include "co/file/co_file.h"
CoFile file;
// 1. open
if(!file.open("test.txt"))
{
    co_return;
}

// 2. read
int64_t offset = 0;
char buf[1024];
int len = co_await file.read(buf,sizeof(buf),offset);

// 3. write
int64_t offset = 0;
char buf[1024] = {0};
int len = co_await file.write(buf,sizeof(buf),offset);
```

网络(Network)

TCP

CoTcpClient

CoTcpClient使用Windows重叠IO、完成端口和LoopIo实现。

1.connect

```
CoTcpClient client;
bool success = co_await client.connect("127.0.0.1", 9798, 2000);
```

2.disconnect

```
bool success = co_await client.disconnect(2000);
```

3.recv

```
char buf[1024] = {0};
int len = co_await client.recv(buf,sizeof(buf),2000);
if(len < 0)
{
    if(errno == ETIMEDOUT)
    {
        // timeout
    }
    else
    {
        // error
    }
}

if(len == 0)
{
    // close
}

// success
```

4.send

```
char buf[1024] = {0};
int len = co_await client.send(buf,sizeof(buf),2000);
if(len < 0)
{
    if(errno == ETIMEDOUT)
    {
        // timeout
    }
    else
    {
        // error
    }
}

if(len == 0)
{
    // close
}

// success
```

CoTcpServer

CoTcpServer使用Windows重叠IO、完成端口和LoopIo实现。

```
CoTcpServer server;
if(!server.listen("0.0.0.0", 9798, 125))
{
    // failure
}

while(true)
{
    char ip[24] = {0};
    uint16_t port;
    auto sock = co_await server.accept(-1, ip, &port);

    if(sock == INVLAID_SOCKET)
    {
        if(errno == ETIMEDOUT)
        {
            // timeout
        }
        else
        {
            // error
        }
    }

    // success
}
```

数据库

Sqlite

SqliteBinder

```cpp
// bind signed number
template <class T>
SqliteBinder &operator<<(T val)
    requires(refl::traits::is_signed_integral_v<T>)

// bind unsigned number
template <class T>
SqliteBinder &operator<<(T val)
    requires(refl::traits::is_unsigned_integral_v<T>)

// bind floating point
template <class T>
SqliteBinder &operator<<(T val)
    requires(std::is_floating_point_v<T>)

// bind number array
template < uint32_t N, class T>
SqliteBinder &operator<<(T(&val)[N])
    requires(std::is_arithmetic_v<T>)

// bind number vector
template<class T>
SqliteBinder &operator<<(const std::vector<T> &bytes)
    requires(std::is_arithmetic_v<T>)

// bind text
template <uint32_t N>
SqliteBinder &operator<<(char(*val)[N])

SqliteBinder &operator<<(const char *val)

SqliteBinder &operator<<(const std::string &val)

// bind text vector
SqliteBinder &operator<<(const std::vector<std::string> &strings)

// bind base struct
template<class T>
SqliteBinder &operator<<(const T &val)
    requires(refl::traits::is_trivially_class_v<T>)

// bind null
SqliteBinder &operator<<(std::nullptr_t)
```

Sqlite DML

```
#include "co/sqlite/co_sqlite.h"
CoSqlite sqlite("path");
SqliteBinder binder(mode, sql); // mode SQLITE_BINDER_MODE_READ or SQLITE_BINDER_MODE_WRITE
// bind data...

auto result = co_await (binder >> splite);
if(!result.success)
{
    // error
}

// success
```

Sqlite Transaction

```
#include "co/sqlite/co_sqlite.h"
CoSqlite sqlite("path");
auto trans = sqlite.transaction();
auto result =  co_await trans.begin();
if(!result.success)
{
    // error
}

SqliteBinder binder(mode, sql); // mode SQLITE_BINDER_MODE_READ or SQLITE_BINDER_MODE_WRITE
// bind data...

auto result = co_await (binder >> trans);
if(!result.success)
{
    // error
    co_await trans.rollback();
}

result = co_await trans.commit();
```

Oracle

OciBinder

```

```

Oracle DML

```
#include "co/oci/co_oci.h"

CoOci oci("db","user","pwd");
OciBinder binder(mode,sql); // mode OCI_BINDER_MODE_READ OCI_BINDER_MODE_WRITE
// bind data...

auto result = co_await (binder >> oci);
if(!result.success)
{
    // errror
}
```

Oracle Batch

```
#include "co/oci/co_oci.h"

CoOci oci("db","user","pwd");
OciBatch batch(sql, size);
// bind data...

auto result = co_await (batch >> oci);
if(!result.success)
{
    // errror
}
```

Oracle Stream

```
#include "co/oci/co_oci.h"
CoOci oci("db","user","pwd");

OciStream stream(sql);
// bind data...

do{
    auto result = co_await (stream >> oci);
    if(!result.success)
    {
        // error
        break;
    }

    if(result.rows.empty())
    {
        // finish
        break;
    }

    // hande data

}while(stream.m_state == OCI_STREAM_STATE_QUERY);
```

Oracle Transaction

```
#include "co/oci/co_oci.h"

CoOci oci("db","user","pwd");
auto trans = oci.transaction();

auto result = co_await trans.begin();

OciBinder binder(mode,sql); // mode OCI_BINDER_MODE_READ OCI_BINDER_MODE_WRITE
// bind data...

result = co_await (binder >> trans);
if(!result.success)
{
    co_await trans.rollback();
    // errror
}

OciBatch batch(sql, size);
// bind data...

result = co_await (batch >> trans);
if(!result.success)
{
    co_await trans.rollback();
    // errror
}

result = co_await trans.commit();
```

MySql

Mysql DML

```
#include "co/mysql/co_mysql.h"

CoMysql mysql("host","user","pwd","db",port);

MysqlBinder binder(mode,sql); // mode MYSQL_BINDER_MODE_READ MYSQL_BINDER_MODE_WRITE
// bind data...

auto result = co_await (binder >> mysql);
if(!result.success)
{
    // error
}
```

Mysql Batch

//TODO: impl

Mysql Stream

//TODO: impl

Mysql Transaction

```
#include "co/mysql/co_mysql.h"

CoMysql mysql("host","user","pwd","db",port);

auto trans = mysql.transaction();
auto result = co_await trans.begion();

MysqlBinder binder(mode,sql); // mode MYSQL_BINDER_MODE_READ MYSQL_BINDER_MODE_WRITE
// bind data...

result = co_await (binder >> mysql);
if(!result.success)
{
    // error
    co_await trans.rollback();
}

result = co_await trans.commit();
```

网络框架

网络框架分为Channel层、Stream层、Protocol层

Channel

Channel层实现底层网络、串口数据等数据接收和发送，并解包为统一的消息ReadMessage

TcpChannel

Tcp协议的Channel实现。

SerialportChannel

串口协议的Channel实现

UdpChannel

UDP协议的Channel实现

Stream

Stream层基于Channel层, 实现对Channel状态的监听， 流的基本状态更新。具体功能得Protocol层实现

Protocol

WD20Protocol

20内部协议框架， 基于TcpChannel、Stream实现。
协议帧格式定义见站机、终端升级文档 第4节通信协议。
该协议实现了数据推送、命令响应、数据订阅、大包分包发送功能。

互联互通协议

```
// HLHT结构体需要满足以下条件
template <typename T>
concept is_wd20_hlht_struct_v = requires(T t) {
    t.GetHLHTLength(); // 包长度
    t.LoadHLHTBuff(nullptr); // 序列化
};
```

```
// 发送HLHT数据
template<typename Channel, typename DATA>
Coroutine<void> sendHLHTData(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8_t

// 发送HLHT数据 带系统类型 0xFF HisCall 0xFE 核心
template<typename Channel, typename DATA>
Coroutine<void> sendHLHTData(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8_t
```

```
// 发送HLHT命令
template<typename Channel>
Coroutine<void> sendHLHTCmd(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8_t

template<typename Channel, typename CMD>
Coroutine<void> sendHLHTCmd(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8_t

template<typename Channel>
Coroutine<void> sendHLHTCmdSys(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8

template<typename Channel, typename CMD>
Coroutine<void> sendHLHTCmdSys(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uint8
```

```
// 发送HLHT内部命令
template<typename Channel, typename CMD>
Coroutine<void> sendHLHTInnerCmd(Channel channel, uint16_t src, uint16_t dst, uint8_t version, uin

// 发送HLHT内部数据
template<typename Channel, typename CMD>
Coroutine<void> sendHLHTInnerData(Channel channel, uint16_t src, uint16_t dst, uint8_t version, ui
```

```
// HLHT数据处理
while(!m_stopFlag)
{
    auto messages = co_await protocol->getHLHTDatas();
    if(messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20HLHTData*data = (WD20HLHTData*)message->data;
        switch(data->dataType)
        {
            // dispatcher cmd
        }
    }
}
```

```
// HLHT命令处理
while(!m_stopFlag)
{
    auto messages = co_await protocol->getHLHTCmds();
    if(messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20HLHTCmd*data = (WD20HLHTCmd*)message->data;
        switch(data->dataType)
        {
            // dispatcher cmd
        }
    }
}
```

```
// HLHTInner数据处理
while(!m_stopFlag)
{
    auto messages = co_await protocol->getHLHTInnerDatas();
    if(messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20HLHTInnerCmd*data = (WD20HLHTInnerCmd*)message->data;
    }
}
```

```
// HLHTInner命令处理
while(!m_stopFlag)
{
    auto messages = co_await protocol->getHLHTInnerCmds();
    if(messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20HLHTInnerCmd*data = (WD20HLHTInnerCmd*)message->data;
    }
}
```

WD20内部协议

```
// 20内部结构体需要实现一下函数
template <typename T>
concept is_wd20_inner_struct_v = requires(T t) {
t.GetLength();
t.ExplainBuff(nullptr);
t.LoadBuff(nullptr);
};
```

数据发送

```
template<uint32_t CmdType, typename Channel, typename T>
Coroutine<void> send(Channel channel, uint16_t src, uint16_t dst, T &&val);
```

数据处理

```
while(!m_stopFlag)
{
    auto messages = co_await protocol->getInnerDatas();
    if (messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20InnerCmd* data = (WD20InnerCmd*)message->data;
        switch(data->type)
        {
            // TODO: dispatch data
        }
    }
}
```

命令与响应

发送命令，并获取返回结果

```
template <uint32_t CmdType, typename Channel>
Coroutine<std::shared_ptr<ReadMessage>> getResponse(Channel channel, uint16_t src, uint16_t dst, i

template <uint32_t CmdType, typename Channel, typename CMD>
Coroutine<std::shared_ptr<ReadMessage>> getResponse(Channel channel, uint16_t src, uint16_t dst, C

template <uint32_t CmdType, uint32_t RspType, typename Channel, typename RSP>
Coroutine<bool> getResponse(Channel channel, uint16_t src, uint16_t dst, RSP &rsp, int32_t timeout

template <uint32_t CmdType, uint32_t RspType, typename Channel, typename CMD, typename RSP>
Coroutine<bool> getResponse(Channel channel, uint16_t src, uint16_t dst, CMD &&val, RSP &rsp, int3

template <uint32_t RspType, typename Channel, typename Buffer, typename RSP>
Coroutine<bool> getResponse(Channel channel, Buffer buffer, RSP &rsp, int32_t timeout);
```

发送命令响应

```
template<uint32_t RspType>
Coroutine<void> sendResponse(std::shared_ptr<ReadMessage> message);

template<uint32_t RspType, typename T>
Coroutine<void> sendResponse(std::shared_ptr<ReadMessage> message, T &&val);

template<uint32_t RspType, typename Channel, typename T>
Coroutine<void> sendResponse(Channel channel, uint16_t src, uint16_t dst, uint32_t ackSeq, T &&val
```

命令处理

```
while(!m_stopFlag)
{
    auto messages = co_await protocol->getInnerCmds();
    if(messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        WD20InnerCmd* data = (WD20InnerCmd*)message->data;
        switch(data->type)
        {
            // TODO: dispatch cmd
        }
    }
}

Coroutine<void> handleCmd(std::shared_ptr<WD20Protocol> protocol, std::shared_ptr<ReadMessage> mes
{
    Cmd cmd;
    auto size = co_await ReadMessageParser::co_parser<TYPE>(message,cmd);
    if(size == 0)
    {
        co_return;
    }

    Rsp rsp;
    // TODO: handl cmd build rsp

    protocol->sendResponse<TYPE>(std::move(message), std::move(rsp));
}
```

命令与回执

发送数据，并等待回执

```
template <uint32_t CmdType, typename Channel, typename CMD>
Coroutine<bool> getAck(Channel channel, uint16_t src, uint16_t dst, CMD &&val, int32_t timeout);
```

消息订阅

数据订阅

```
// 订阅数据流
template <uint32_t CmdType>
Coroutine<std::shared_ptr<WD20MsgDIStream>> suscribe(std::shared_ptr<TcpChannel> channel, uint16_t

template <uint32_t CmdType, typename CMD>
Coroutine<std::shared_ptr<WD20MsgDIStream>> suscribe(std::shared_ptr<TcpChannel> channel, uint16_t

// WD20MsgDIStream:
// 获取订阅数据
Coroutine<std::list<std::shared_ptr<ReadMessage>>> getPushMessages(int32_t timeout);

// 取消订阅数据
template <uint32_t RspType>
Coroutine<bool> cancel();
```

使用数据输入流

```
Cmd cmd;
auto stream = protocol->suscribe<TYPE>(channel, 0, 0, cmd);
while(stream->state() == STREAM_STATE_CONNECTED)
{
    auto messages = co_await stream->getPushMessages(-1);
    if(messages.empty())
    {
        // end or channel is disconnect
        break;
    }

    // TODO: handle message
}
```

数据推送

```
// 获取输出流
Coroutine<std::list<std::pair<std::shared_ptr<ReadMessage>, std::shared_ptr<WD20MsgDOStream>>>> ge

// WD20MsgDOStream

// 数据推送
template <uint32_t RspType, typename T>
Coroutine<bool> push(const T &val);

// 推送完毕
template <uint32_t RspType>
Coroutine<bool> end();
```

处理数据输出流

```
while(!m_stopFlag)
{
    auto streams = co_await protocol->getMsgDOStreams();
    if (streams.empty())
    {
        break;
    }

    for (auto &[message, stream] : streams)
    {
        WD20InnerCmd *cmd = (WD20InnerCmd *)message->data;
        switch (cmd->type)
        {
            // dispatch stream
        }
    }
}

Coroutine<void> handleStream(std::shared_ptr<ReadMessage> message, std::shared_ptr<WD20MsgDOStream
{
    Cmd cmd;
    auto size = co_await ReadMessageParser::co_parser<TYPE>(message,cmd);
    if(size == 0)
    {
        co_return;
    }

    Rsp rsp;
    while(stream->state() == STREAM_STATE_CONNECTED)
    {
        // TODO: build rsp
        co_await stream->push<TYPE>(rsp);
    }
}
```

客户端输入流 该输入流是对订阅数据流的封装，实现自动重连服务器

```
WD20ClientDIStream(std::shared_ptr<WD20Protocol> protocol);

// 开始订阅
template <uint32_t CMD_TYPE, typename CMD>
void start(CMD cmd)

template <uint32_t CMD_TYPE>
void start()

    // 获取消息
Coroutine<std::list<std::shared_ptr<ReadMessage>>> getMessages(int32_t timeout)

// 停止订阅
Coroutine<void> stop()
```

使用客户端输入流

```
WD20ClientDIStream stream(protocol);

Cmd cmd;
stream.start<CMD_TYPE>(cmd);

while(!m_stopFlag)
{
    auto messages = co_await stream.getMessages(-1);
    if (messages.empty())
    {
        break;
    }

    for(auto &message : messages)
    {
        Rsp rsp;
        auto size = co_await ReadMessageParser::co_parser<RSP_TYPE>(message,rsp);
        if(size == 0)
        {
            continue;
        }

        // TODO: handle rsp
    }
}

co_await stream.stop();
```

MFC使用协程框架

MFC有自己的一套事件循环系统, 因此需要将协程嵌入MFC的事件循环中

```
// *app.cpp

#include "mfc/mfc_helper.h"

MFCHelper::getInstance()->init();
```

MFCHelper实现

init

```
// 初始化网络
WSAData version;
WSAStartup(WINSOCK_VERSION, &version);

// 初始化日志库
std::string logPath = ExeUtils::getExeDirPath() + "//..//日志文件";
FileUtils::createDirs(logPath);
WDLog::getInstance()->init(logPath.c_str(), LOG_LEVEL_DEBUG);

// 初始化事件循环
setLoopThread();

Loop *loop = threadLoop();
loopInit(loop);

// 开启定时器 每5ms执行一次
m_impl->timerId = SetTimer(nullptr, 0, 5, timerCallback);
```

callback

```
// 执行一次事件循环
Loop *loop = threadLoop();
loopRunOnce(loop);
```

release

```
// 关闭定时器
if (m_impl->timerId != 0)
{
    KillTimer(nullptr, m_impl->timerId);
}
setLoopStop();

// 等待所有协程执行完毕
Loop *loop = threadLoop();
loopRun(loop);
loopClose(loop);

// 关闭日志模块
WDLog::getInstance()->release();
```

Qt使用协程框架

Qt自有一套事件循环机制, 因此需要将协程嵌入事件循环中

```
#include "coroutineapplication.h"

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

    MainWindow w;
    w.showMaximized();

    int ret = a.exec();
    return ret;
}
```

CoroutineApplication实现

```
CoroutineApplication::CoroutineApplication(int argc, char *argv[])
    : QApplication(argc, argv)
{
    // init network
    WSAData version;
    WSAStartup(WINSOCK_VERSION, &version);

    QApplication::processEvents();

    // init log
    QString path = QApplication::applicationDirPath() + "//..//日志文件";
    std::string logPath = path.toLocal8Bit().data();
    FileUtils::createDirs(logPath);
    WDLog::getInstance()->init(logPath.c_str(), LOG_LEVEL_DEBUG);

    // init loop
    setLoopThread();

    Loop *loop = threadLoop();
    loopInit(loop);
    m_loop = loop;

    // init timer
    m_timer = new QTimer(this);
    connect(m_timer, &QTimer::timeout, this, &CoroutineApplication::timeCallback);

    // start timer 5ms
    m_timer->setInterval(5);
    m_timer->start();
}

CoroutineApplication::~CoroutineApplication()
{
    m_timer->stop();
    // stop loop
    setLoopStop();

    // wait loop stop
    loopRun(m_loop);
    loopClose(m_loop);

    // release log
    WDLog::getInstance()->release();
}

void CoroutineApplication::timeCallback()
```

```
{
    // run once
    loopRunOnce(m_loop);
}
```