Resume function实践

兰征鹏([tearshark@163.com](mailto:tearshark@163.com))

# 从Simulated function说起

在C++11之前，有一种类，用来模仿函数行为的类，学名叫做“functor”。Functor这种范式成熟的结果，就是大家觉得还是编译器来实现比较好，于是有了C++11的lambda。我们来看看lambda与functor之间的关系，这有助于理解下面的Resume function。

|  |  |
| --- | --- |
| struct functor  {  mutable int **value**;  functor(int **val\_**) :**value**(**val\_**) {}  int operator ()(int **b**) const  {  return **value** + **b**;  }  };  auto **a** = lambda(5);  std::*cout* << **a**(4) << std::*endl*; | int **value** = 5;  auto **a** = [**value**]  (int **b**) mutable  {  return **value** + **b**;  };  std::*cout* << **a**(4) << std::*endl*; |

Lambda通过capture，将捕获的变量变成了类成员，lambda函数体，其实就是functor的operator()符号重载，编译器辅助生成了仿函数的需要的所有一切元素。

Lambda如此的好用，以至于，要在C++11的所有特性中，只选择一个作为最重要的特性，我首选lambda。Lambda使得需要callback的异步编程相对于C++11之前，容易了很多。以前怕麻烦不敢想不敢用的callback，现今都可以通过简单的capture规划，使用lambda完成。lambda甚至可以说改变了编程的习惯与思路。

于是，lambda被应用得越来越多，越来越多……

然而，万事万物皆非完美，当lambda作为callback被大量滥用的时候，另外一个东西则冒出来了：

**CALLBACK HELL**

# 进入callback hell

以下图例是笔者在某个项目真实的经历。下面每一个+号后面分支出来的过程，都是一个callback。也是这个代码让笔者下定了决心要找到一个合理的解决callback hell的方案。

--main thread

--|

--+---+switch ui thread

--| |

--| |----start switch ui

--| +--------------------+show section layer

--| |----end switch ui |

--| L----finishied |----show GameArmySectionLayer

--| |

--| |----open section

--| +--------------------+show fight result

--| |----update star UI |

--| L----finished |----create defense army

--| |----army initialize by section

--| |----switch ui thread

--| +----------------------+show fighting

--| |----switch ui thread |

--| L----finished |----init battle info

--| +---------------+show FightWorldLayer

--| |----end battle |

--| L----finished |

--| +---------------+show GameFightChapterView

--| | |

--| L----finished |----chapter passed

--| L----finished

当这种代码一而再，再而三出现的时候，越来越觉得代码不可读，维护起来也非常累。笔者也想过用future-then范式，把递归的callback转成线性的callback代码，但初略考察后，觉得还不是解决问题的终极办法。于是，协程这个古老的概念被想起。然而，限于现有的一些协程功能，比如Windows下的Fiber，Boost实现的coroutine，以及一些开源的协程实现，如libgo，几乎都是stackfull的协程。对内存敏感的客户端程序来说（笔者目前做移动端游戏，对内存使用还是很谨慎），stackfull的协程用起来还是很心虚。在这过程中，也有网友给我介绍他自己的stakless的协程库，使用宏来实现的，用起来似乎也不是很愉快。

寻寻觅觅中，2017年到来，C++17 coroutine的提案也进入笔者视野。

# 再说coroutine

如果要手工实现一个stackless的协程库，不使用宏的话，大致代码就长得像这个样子：

struct coroutine

{

int **result**;

int **value**;

coroutine(int **val\_**) :**value**(**val\_**) {}

int **step\_** = 0;

void goNext()

{

switch (**step\_**)

{

case 0:

++**step\_**;

std::*cout* << "step 0" << std::*endl*;

**result** = **value** \* (*rand*() % 4);

break;

case 1:

++**step\_**;

std::*cout* << "step 1" << std::*endl*;

**result** = **value** \* (*rand*() % 4);

break;

case 2:

++**step\_**;

std::*cout* << "step 2" << std::*endl*;

**result** = **value** \* (*rand*() % 4);

break;

default:

**step\_** = -1;

break;

}

}

bool done() const { return **step\_** < 0; }

int currentValue() const { return **result**; }

};

使用这个协程类的代码大概就是这样子：

coroutine **c**(5);

for (**c**.goNext(); !**c**.done(); **c**.goNext())

{

std::*cout* << **c**.currentValue() << std::*endl*;

}

理论上，每个stackless的协程，我们都可以这么完成；有了前面lambda的经验后，我们很容易想到，把这种重复性的劳动，交给编译器去做。Resume function正好就是这种东西。

# 说到Resume function

因为是介绍Resume function实践的文章，所以，我直接抛出现成的语法，而不去详细介绍Resume function的由来了。用Resume function来写这个代码的话，大致代码如下，注意co\_yield关键字：

auto coroutine(int **value**)

{

std::*cout* << "step 0" << std::*endl*;

co\_yield **value** \* (*rand*() % 4);

std::*cout* << "step 1" << std::*endl*;

co\_yield **value** \* (*rand*() % 4);

std::*cout* << "step 2" << std::*endl*;

return **value** \* (*rand*() % 4);

}

auto **c** = coroutine(5);

for (auto **v** : **c**)

std::*cout* << **v** << std::*endl*;

然后我们将手工写的coroutine代码，和编译器版本的coroutine放在一起来对比下：

|  |  |
| --- | --- |
| struct coroutine  {  int **result**;  int **value**;  coroutine(int **val\_**) :**value**(**val\_**) {}  int **step\_** = 0;  void goNext()  {  switch (**step\_**)  {  case 0:  ++**step\_**;  std::*cout* << "step 0" << std::*endl*;  **result** = **value** \* (*rand*() % 4);  break;  case 1:  ++**step\_**;  std::*cout* << "step 1" << std::*endl*;  **result** = **value** \* (*rand*() % 4);  break;  case 2:  ++**step\_**;  std::*cout* << "step 2" << std::*endl*;  **result** = **value** \* (*rand*() % 4);  break;  default:  **step\_** = -1;  break;  }  }  bool done() const { return **step\_** < 0; }  int currentValue() const { return **result**; }  };  coroutine **c**(5);  for (**c**.goNext(); !**c**.done(); **c**.goNext()){  std::*cout* << **c**.currentValue() << std::*endl*;  } | auto coroutine  (int **value**)  {  std::*cout* << "step 0" << std::*endl*;  co\_yield **value** \* (*rand*() % 4);  std::*cout* << "step 1" << std::*endl*;  co\_yield **value** \* (*rand*() % 4);  std::*cout* << "step 2" << std::*endl*;  return **value** \* (*rand*() % 4);  }  auto **c** = coroutine(5);  for (auto **v** : **c**){  std::*cout* << **v** << std::*endl*;  } |

可以看出，只要编译器帮我们把函数入参，局部变量转成类成员变量，同时额外添加一个step\_变量来指示当前运行到那一步骤，就可以把一个函数，转成一个协程类。而程序员就可以较为专注与业务代码，而不用去考虑手写协程的细节问题。

# Resume function的细节

那么，编译器可能如何去实现这些细节的呢？下面以一个手写的函数为例:

*R* foo(T1 a, T2 b) {

T3 c;

body-containing-*suspend*-resume-points

}

编译器可以像下面这样的代码，构造出一个“函数”：

*R* foo(T1 **a**, T2 **b**) {

using \_\_traits = std::resumable\_traits<*R*, T1, T2>;

struct \_\_Context {

\_\_traits::promise\_type **\_Promise**;

T1 **a**;

T2 **b**;

T3 c;

template <typename U1, typename U2>

\_\_Context(U1&& **a**, U2&& **b**) : **a**(forward<U1>(**a**)), **b**(forward<U2>(**b**)) {}

void operator()() noexcept {

await **\_Promise**.initial\_suspend();

try { body-containing-*suspend*-resume-points-with-some-changes }

catch (...) { **\_Promise**.set\_exception(std::*current\_exception*()); }

**\_\_return\_label**:

await **\_Promise**.final\_suspend();

<deallocate - frame> (this, sizeof(\_\_Context) + <X>);

}

};

auto **mem** = <allocate - frame>(sizeof(\_\_Context) + <X>);

\_\_Context \* **coro** = nullptr;

try {

**coro** = new (**mem**) \_\_Context(**a**, **b**);

auto **result** = \_\_traits::get\_return\_object( std::resumable\_handle<\_\_traits::promise\_type>::from\_promise(&**coro**->\_\_Promise));

(\***coro**)();

return **result**;

}

catch (...) {

if (**coro**) **coro**->~\_\_Context();

<deallocate - frame> (**mem**, sizeof(\_\_Context) + <X>);

throw;

}

}

struct \_\_Context

\_\_Context就是这个函数对应的类，成员变量a,b是函数入参；a,b赋值通过\_\_Context的构造函数。c是一个函数的局部变量。\_Promise是Resume function与用户代码交互的类，我们可以通过写自己的promise\_type来完成Resume function的一些细节。

operator()()重载是真正的函数体，这个函数体被主要包含四部分：

初始化：**\_Promise**.initial\_suspend()；

原始业务逻辑代码；

异常处理：**\_Promise**.set\_exception(std::*current\_exception*());

终止代码：**\_Promise**.final\_suspend()

foo函数

foo函数就变成了构造一个\_\_Context实例，并返回与之相关的awaitable。<X>代表Resume function在编译器内部(\*)必须的帧数据，通过inplacement new构造好\_\_Context实例，通过std::resumable\_handle<>::from\_promise构造好<X>代表的帧数据，整个resume function就算构造完成。然后返回与之关联的awaitable。

这个awaitable里，可以通过await\_suspend()获得一个coroutine\_handle<>。保存这个handle，在合适的时候，调用这个handle，就可以将resume function进行到下一步。反复这个过程，最终执行到**\_\_return\_label**:指示的代码处，整个resume function被销毁，内存被回收。

我们要在promise和awaitable之间，引入一个State概念。这个State，负责将promise, awaitable和逻辑代码串起来，最终完成一个协调的整体。

此范例代码引用自：N4134-<Resumable Functions v2>

(\*)不去详细区分是编译器需要还是CRT需要

用一张表来总结下这个过程：

|  |  |  |  |
| --- | --- | --- | --- |
| R | \_\_Context |  |  |
|  | Coroutine frame |  |  |
| Awaitable<R> <- | Promise | ->State<- | Logic |
|  | Argument |  |  |
|  | Local Variable |  |  |

浅绿色背景色：是程序要完成的工作

红色文字的Logic：是逻辑代码要完成的工作，其它三部分可以写成通用的代码

蓝色文字：是编译器自动生成的代码

Promise概念模型:

为了在用户代码和编译器之间交互，因此，Resume function提出了Promise概念：

concept promise<typename T>

{

awaitable<void> initial\_suspend();

awaitable<void> final\_suspend();

bool cancellation\_requested();

awaitable<T> get\_return\_object();

void return\_value(T); //when T is not void

void yield\_value(T); //when T is not void

void return\_void(); //when T is void

void set\_exception(std::*exception\_ptr &&*);

};

initial\_suspend()用于初始化;

final\_suspend()用于清理析构;

cancellation\_requested()用于判断是否要终止Resume function；

get\_return\_object()用于返回与之关联的awaitable；

return\_value(T)/ yield\_value(T)/ return\_void()用于获取Resume function上一次执行的结果，然后用这个结果，继续后续步骤的代码；

set\_exception()用于处理异常。

至于返回的awaitable，则是另外一个概念模型。我把这个awaitable放到下面一起讲。

# Awaitable function的细节

与Promise相关关联的awaitable又是一个什么东西呢？实际上是一个“可等待对象(Awaitable object)”。因为Resume function是被拆分成多个步骤执行的，每个步骤执行完毕，都会返回一个“可等待对象”，然后等待下一次执行。

同时，这个“可等待对象”可不是只能在Resume function里能用，而是在任何“可等待的函数(Awaitable function)”里使用，只要这个函数返回一个“可等待对象”，并且行为遵循“可等待对象”的行为即可。

这个“可等待对象”被定义为如下：

concept awaitable<typename T>{

//这件事你准备好了吗？

bool await\_ready();

//准备好了，那值是什么呢？

T await\_resume();

//没准备好？那准备好了后再来叫我，我先去干别的事情。这是呼叫我的暗号，记住了

void await\_suspend(std::experimental::coroutine\_handle<>);

};

任何一个返回“可等待对象”的函数，我们都称为“Awaitable function”。这个概念非常重要，是将callback转为协程的重要手段。

“Awaitable function”是如何被使用的呢？当我们写下下面的代码的时候：

co\_await awaitable\_function();

...;

编译器实际会生成如下代码，来完成co\_await操作：

auto **\_\_awaitable** = awaitable\_function();

if (!**\_\_awaitable**.await\_ready())

{

**\_\_awaitable**.await\_suspend(**label\_resume**);

return **<other awaitable object>;**

**label\_resume:**

**<resume execute...>;**

**}**

auto val = **\_\_awaitable**.await\_resume();

<deallocate>(**\_\_awaitable**);

...;

对于每一个返回的awaitable object，编译器：

1. 首先检查await\_ready()；
2. 如果已经准备好了结果，则调用await\_resume()获得值，然后不暂停继续执行后续代码；
3. 如果还未准备好，则调用await\_suspend()，给出执行下一段代码的协程句柄coroutine\_handle<>(看作一个仿函数)；
4. 当在其它时候，工作完成之后，我们可以调用这个coroutine\_handle<>，继续协程的执行。
5. coroutine\_handle<>只能执行一次，也不能不调用----否则，整个Resume function就被挂起来了，也无法得到清理和析构。

concept awaitable非常重要，将callback转为coroutine就是靠awaitable完成的。将阻塞/异步操作转化为支持协程的过程，叫做“green”。这个green也有一个通用的范式，大概如下代码所示：

template<typename T>

awaitable\_t<T> green\_async\_operator()

{

awaitable\_state\_t<T> **st =** {...};

if (condition *is* true)

**st**.set\_value();

else

async\_operator\_with\_callback([**st**](...)

{

if (have exception)

**st.**set\_exception(std::*make\_exception\_ptr*(**ex**));

else

**st**.set\_value(...);

});

return awaitable\_t**<T>(st)**;

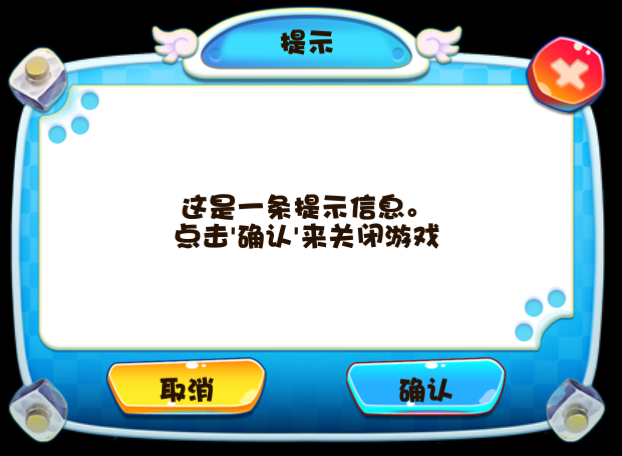
}

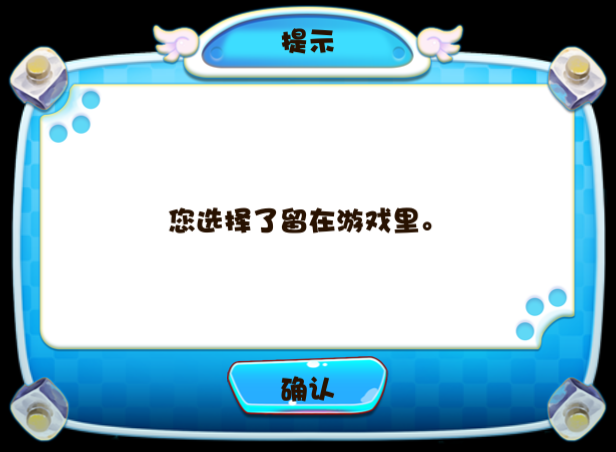
这中间，有看到了熟悉的State概念。State负责关联业务代码和Awaitable Object，是真正负责功能的类。这段代码执行过程大概如下：

1. 生成State实例，用于关联业务代码和即将返回的awaitable；
2. 如果条件已经满足了，则直接给State设置值；随后的awaitable<T>::await\_ready()返回true；
3. 如果条件不满足，则开启异步操作，或者新开线程执行阻塞操作，并将State作为回调必须的变量保存起来
4. 在回调里面，对State要么设置异常，要么设置值
5. 调用awaitable<T>::await\_suspend()设置的coroutine\_handle<>，继续执行Resume function后续代码。

至此，Resume function的各方面的细节就介绍得差不多了，下面，我们要用一些范例来说明如何使用Resume function。

# UI范例





## 延迟回调接口的MsgBox

enum struct MsgButton : uint32\_t{

Default = 0x7fffffff,

OK = 1U,

Cancel = 2U,

Close = 4U,

OKCancel = 3U,

};

using MessageBoxCallback = std::*function*<void(MsgButton)>;

class MessageBoxLayer : public CommonTitleLayer {

public:

virtual bool init(const char \* **msg**, const MessageBoxCallback & **cb**);

private:

MessageBoxCallback m\_Callback;

};

MessageBoxLayer \* showMessage\_CB(const char \* **msg**, const MessageBoxCallback & **cb**,

cocos2d::Scene \* **pScene**)

{

MessageBoxLayer \* **layer** = new MessageBoxLayer;

**layer**->init(**msg**, **cb**);

**pScene**->addChild(**layer**, 999999);

**layer**->autorelease();

return **layer**;

}

showMessage\_CB(u8"这是一条提示信息。\n点击'确认'来关闭游戏",

[=](MsgButton **ok**){

if (**ok** == MsgButton::OK) {

Director::getInstance()->end();

}

else if(**ok** > MsgButton(0)){

showMessage\_CB(u8"您选择了留在游戏里。",

[=](MsgButton){

CCLOG("end message box");

});

}

}

);

## 协程接口的MsgBox

resumef::awaitable\_t<MsgButton>

showMessage(const char \* **msg**, cocos2d::Scene \* **pScene**)

{

resumef::awaitable\_t<MsgButton> **awaitable**;

auto **box** = showMessage\_CB(**msg**,

[st = **awaitable**.\_state](MsgButton **ok**)

{

st->set\_value(**ok**);

}, p**Scene**);

return **awaitable**;

}

void HelloWorld::showMessageBox()

{

GO

{

auto **ok** = co\_await **showMessage**(u8"这是一条提示信息。\n点击'确认'来关闭游戏");

if (**ok** == MsgButton::OK)

{

this->menuCloseCallback(nullptr);

}

else

{

co\_await **showMessage**(u8"您选择了留在游戏里。", {}, nullptr);

CCLOG("end message box");

}

};

}

# MySQL范例

MySQL是一个应用广泛的数据库，其客户端API并没有提供异步接口，是一个典型的阻塞接口。通常来说，不能在主逻辑线程里直接调用MySQL的阻塞API，而是选择将MySQL的阻塞API放在其它线程去调用。MySQL的连接数量又是有限的，所以，一个线程池+MySQL连接池就能很好的处理MySQL的操作了。下面就以一个笔者用到的异步回调接口的MySQL功能为例。

## 异步回调接口的MySQL

MySQL查询功能，基本上分为三个：Select，Update，Insert，三个操作的返回值不一样。所以定义了下面的枚举。但限于篇幅，这里只以Update为例。

enum struct AsyncOperator{

Select, Update, Insert

};

然后定义异步操作队列**m\_listAsyncQuery。**

struct AsyncNode;

std::list<AsyncNode> **m\_listAsyncQuery**;

再定义操作完毕的结果队列。留意结果队列的顺序，是否要跟操作队列的顺序一一对应。

struct SynchResult;

std::list<SynchResult> **m\_listSynchResult**;

那么，一个异步的MySQL查询代码就是这样子的：

typedef std::*function*<void(bool, uint64\_t, std::*exception\_ptr* &&)> async\_update\_callback;

bool AsynUpdate(const std::*string* & **str**, const async\_update\_callback & **callback**){

**m\_listAsyncQuery**.*emplace\_back*(AsyncOperator::Update, **str**, **callback**);

return true;

}

另外的线程里，执行操作队列的代码大概如下：

void AsyncQueryThread()

{

//一个同步的MySQL连接实现

SynchMysqlConnect **impl**;

//循环等待队列

for (;;)

{

//忽略异步操作队列的锁啊条件变量啊信号量啊之类的东西......

AsyncNode **an** = <pop front>(**m\_listAsyncQuery**);

try

{

//同步执行MySQL的Update查询

auto **ret** = **impl**.Query(**an**.**sql**.*data*(), **an**.**sql**.*size*());

//获得Update结果

uint64\_t **iid** = **ret** ? **impl**.affected\_rows() : 0;

//iid放入结果队列，再次忽略锁之类的玩意儿

**m\_listSynchResult**.*emplace\_back*(std::move(**an**), **ret**, nullptr, **iid**, nullptr);

}

catch (...)

{

//发生了异常，将异常放入结果队列

**m\_listSynchResult**.*emplace\_back*(std::move(**an**), false, nullptr, 0, std::*current\_exception*());

}

}

}

循环结果队列，调用回调函数

void MysqlConnectLoop()

{

std::list<SynchResult> **listSynchResult = <move>(m\_listSynchResult);**

for (SynchResult & **sr** : **listSynchResult**)

**sr**.cb(**sr**.*result*, **sr**.**iid**, std::move(**sr**.ex));

}

最后回调接口的异步MySQL功能，用起来就是这个样子：

auto strSql = "UPDATE world.city SET Population = Population + 1 WHERE `Name`='Kabul'"s;

AsynUpdate(strSql,

[](bool, uint64\_t **effectCnt**, std::*exception\_ptr* && **ex**)

{

std::*cout* << **effectCnt** << std::*endl*;

});

## 协程接口的MySQL

那么，要如何做到MySQL支持协程呢？是不是要把整个回调接口的代码改写一次呢？回答显然是否定的，只需要增加一小段代码，充分利用现有的回调接口即可:

resumef::awaitable\_t<int64\_t>

mysql\_update(const std::*string* & **str**)

{

resumef::awaitable\_t <int64\_t> **awaitable**;

auto **callback**=[st=**awaitable**.\_state](bool **result**, uint64\_t **cnt**, std::*exception\_ptr* && **ex**)

{

if (!**ex**)

st->set\_value(**result** ? **cnt** : 0);

else

st->set\_exception (std::move(**e**));

st->resume(); //如果不配合librf使用

};

try

{

if (!AsynUpdate(**str**, **callback**))

**awaitable**.\_state->set\_value(0);

}

catch (...)

{

**awaitable**.\_state->set\_exception(std::*current\_exception*());

}

return **awaitable**;

}

利用绿化后的mysql\_update函数，就可以开始在协程里执行MySQL语句了：

auto strSql = ...;

uint64\_t **effectCnt =** co\_awaitmysql\_update(strSql);

std::*cout* << **effectCnt** << std::*endl*;

对比下代码，嗯，优雅了很多。

AsynUpdate(strSql,

[](bool, uint64\_t **effectCnt**, std::*exception\_ptr* && **ex**)

{

std::*cout* << **effectCnt** << std::*endl*;

});

# ASIO(Network)范例

## 回调接口的ASIO

无协程的情况下：

void start()

{

do\_read([this](*size\_t* **size**) {

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

do\_write(prepare\_write\_msg("first logic result : ", **size**), [this]

{

do\_read([this](*size\_t* **size**) {

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

do\_write(prepare\_write\_msg("second logic result : ", **size**), [this]

{

do\_read([this](*size\_t* **size**) {

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

do\_write(prepare\_write\_msg("third logic result : ", **size**), [this]

{

//无限不循环......

});

});

});

});

});

});

}

template<class \_Fx>

void do\_read(\_Fx && **fn**)

{

auto **self**(*shared\_from\_this*());

asio::async\_read\_until(**socket\_**, **read\_stream\_**, 0,

[this, **self**, **fn** = std::*forward*<\_Fx>(**fn**)](const asio::error\_code& **ec**, std::*size\_t* **size**)

{

if (!**ec** && **size** > 0) {

auto **bufs** = **read\_stream\_**.*data*();

std::*copy*(asio::buffers\_begin(**bufs**), asio::buffers\_end(**bufs**), **read\_buff\_**.*begin*());

**read\_stream\_**.consume(asio::buffer\_size(**bufs**));

**fn**(asio::buffer\_size(**bufs**));

}

});

}

template<class \_Fx>

void do\_write(*size\_t* **size**, \_Fx && **fn**)

{

...  
}

## ASIO对协程的支持

背景知识、ASIO的异步操作的返回值：

auto asio::async\_xxx(**…**, handler) -> ???

template <typename Handler, typename Signature>

struct handler\_type

{

typedef Handler type;

};

其返回值是通过handler\_type + async\_result两个模板决定的，因此，我们想办法做一个特殊的handler, 使得handler\_type + async\_result返回resume awaitable需要的awaitable\_t<>接口。

template<typename Allocator = std::*allocator*<void> >

class use\_task\_t{……};

\_\_declspec(selectany) use\_task\_t<> ***use\_task***;

template<typename Allocator, typename *ReturnType*, typename Arg2>

struct handler\_type<use\_task\_t<Allocator>, *ReturnType*(asio::error\_code, Arg2)> {

typedef detail::promise\_handler<Arg2> type;

};

template<typename T>

class async\_result<detail::promise\_handler<T> > {

awaituv::awaitable\_t<T> get() { return std::*move*(task\_); }

awaituv::awaitable\_t<T> task\_;

};

template<typename T>

class promise\_handler;

负责生成resume function必须的promise\_t<>,并通过async\_result<>返回的awaitable\_t<>。

最终使用的时候，就这样使用了：

auto result = co\_await asio::async\_xxx(**……**, asio::***use\_task***);

## 协程接口的ASIO

如果使用协程，该如何做：

awaituv::awaitable\_t<void> start()

{

auto **self** = this->*shared\_from\_this*();

auto **size** = co\_await do\_read();

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

co\_await do\_write(prepare\_write\_msg("first logic result : ", **size**));

**size** = co\_await do\_read();

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

co\_await do\_write(prepare\_write\_msg("second logic result : ", **size**));

**size** = co\_await do\_read();

std::*cout* << **read\_buff\_**.*data*() << std::*endl*;

co\_await do\_write(prepare\_write\_msg("third logic result : ", **size**));

//无限不循环......

}

awaituv::awaitable\_t<*size\_t*> do\_read()

{

auto **size** = co\_await asio::async\_read\_until(**socket\_**, **read\_stream\_**, 0, asio::use\_task);

auto **bufs** = **read\_stream\_**.*data*();

std::*copy*(asio::buffers\_begin(**bufs**), asio::buffers\_end(**bufs**), **read\_buff\_**.*begin*());

**read\_stream\_**.consume(asio::buffer\_size(**bufs**));

return asio::buffer\_size(**bufs**);

}

awaituv::awaitable\_t<*size\_t*> do\_write(*size\_t* **size**)

{

return asio::async\_write(**socket\_**, asio::buffer(**write\_buff\_**.*data*(), **size**), asio::use\_task);

}

# 问题一、跨线程调度

协程被跨线程了

//这是一个重度计算任务，只能单开线程来避免主线程被阻塞

auto async\_heavy\_computing\_tasks(*int64\_t* **val**)

{

awaituv::awaitable\_t<*int64\_t*> **awaitable**;

std::thread([**val**, st = **awaitable**.\_state]

{

std::*this\_thread*::sleep\_for(500ms);

st->set\_value(**val** \* **val**);

}).*detach*();

return **awaitable**;

}

awaituv::awaitable\_t<void> heavy\_computing\_sequential(*int64\_t* **val**)

{

std::*cout* << **val** << " @" << std::*this\_thread*::*get\_id*() << std::*endl*;

**val** = co\_await async\_heavy\_computing\_tasks(**val**);

std::*cout* << **val** << " @" << std::*this\_thread*::*get\_id*() << std::*endl*;

**val** = co\_await async\_heavy\_computing\_tasks(**val**);

std::*cout* << **val** << " @" << std::*this\_thread*::*get\_id*() << std::*endl*;

**val** = co\_await async\_heavy\_computing\_tasks(**val**);

std::*cout* << **val** << " @" << std::*this\_thread*::*get\_id*() << std::*endl*;

}

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

{

std::*cout* << "main thread id is " << std::*this\_thread*::*get\_id*() << std::*endl*;

heavy\_computing\_sequential(2);

return 0;

}

main thread id is 11716

2 @11716

4 @12688

16 @264

256 @10472

# 问题二、协程的通用限制

如errno，ctime等基于TLS的函数，阻塞线程的sleep等函数，以及mutex这样的线程同步原语，在协程里均要小心使用或者不能使用。在扩展Awaitable function的时候，也要注意，不要返回诸如ctime()的字符串这样的“引用”语义的内容，还是老老实实通过std::string构造一个“拷贝”语义的内容，再返回给协程。而诸如errno这样的错误信息，推荐使用“exception”返回给协程。

# Exception的处理

对于异步模型，当发生了异步不能处理的异常的时候，通常是要将异常想办法送回回调函数去处理。对于ASIO来说，就是几乎每个回调函数，都会给一个asio::error\_code表示错误。因此，在处理resume function的时候，也要考虑这个问题。

Librf选择了通过void state\_t::set\_exception(std::exception\_ptr && ex)来设置异常。并在获取值的时候，在函数auto awaitable\_t::await\_resume()里将这个异常抛出，从而将异常带入到resume function里。

template <typename T = void>

struct awaitable\_t

{

T await\_resume()

{

\_state->rethrow\_if\_exception();

return \_state->get\_value();

}

};

在整个异常处理里，std::exception\_ptr担任了传递异常的重要任务。当异步功能要返回一个异常时，可以通过构造一个std::exception或派生自std::exception 的异常，通过std::make\_exception\_ptr()获得exception\_ptr；当异步功能捕获到一个异常时，可以通过std::current\_exception()获得当前异常的exception\_ptr。

然后，在await\_resume()函数里，通过std::rethrow\_exception()抛出异常。

最后，resume function代码，就跟正常写顺序代码一样的方式处理异常了：

awaitable\_vt test\_signal\_exception()

{

try

{

auto r = co\_await async\_signal\_exception(10);

std::cout << "result is " << r << std::endl;

}

catch (const std::exception& e)

{

std::cout << "exception signal : " << e.what() << std::endl;

}

}

//请打开结构化异常(/EHa)

auto async\_signal\_exception(const *intptr\_t* dividend)

{

awaitable\_t<*int64\_t*> awaitable;

std::*thread*([dividend, st = awaitable.\_state]

{

std::*this\_thread*::sleep\_for(std::*chrono*::*milliseconds*(50));

try

{

//也可以注释掉这个判断，使用结构化异常。但就获得不了具体描述信息了

if (dividend == 0)

throw std::*logic\_error*("divided by zero");

st->set\_value(10000 / dividend);

}

catch (...)

{

st->set\_exception(std::*current\_exception*());

}

}).*detach*();

return awaitable;

}

auto async\_signal\_exception2(const *intptr\_t* dividend)

{

awaitable\_t<*int64\_t*> awaitable;

std::*thread*([dividend, st = awaitable.\_state]

{

std::*this\_thread*::sleep\_for(std::*chrono*::*milliseconds*(50));

if (dividend == 0)

st->throw\_exception(std::*logic\_error*("divided by zero"));

else

st->set\_value(10000 / dividend);

}).*detach*();

return awaitable;

}

# Resume function的总结

上述的范例代码，分别给出了两个函数：

Resume function

Awaitable function

分别用于协程逻辑代码，和绿化阻塞代码。

为了操作Resume function，提供了四个概念：

Promise/Awaitable

State

Awaitable

Promise/Awaitable用于Resume function，对于用户来说，Promise几乎不可见。只需要申明返回Awaitable<T>就可以了，然后在代码里使用co\_await关键字。

Awaitable/State用于Awaitable function。这是一种通用的方法，来绿化阻塞操作，使得阻塞功能也能应用到Resume function上的方法。

还分别演示了resume function如何应用在

MySQL数据库

UI

网络ASIO

通过这些范例，看得出，Resume function能跟现有callback范式的，异步/延迟代码很好的结合，完美解决回调地狱问题，同时也能很容易的跟第三方异步代码的集成。

# 协程的一些现状

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | stackfull | stackcopy | stackless | |
|  | 传统 | resume function |
|  | 每一个协程单独一个栈 | 所有协程共享一个栈 | 不需要栈控件，换成堆空间 | |
| 内存占用 | 高 | 低 | 低 | 低 |
| 切换代价 | 小 | 大 | 小 | 小 |
| 编码难度 | 简单 | C++下及其困难 | 困难 | 简单 |
| 操作系统 | 提供支持 | 不支持(\*) | 不需要特殊支持 | 不需要特殊支持 |
| 历史 | 悠久 |  | 有久远的应用 | NEW |
| 可靠性 | 高 | ???(不了解) | 莫名担心 | ? |
| 借鉴性 |  |  |  | C#,… |
| 范例 | libgo,… | libco,… |  | librf, awaitable\_tasks |

(\*):虽然也利用了操作系统提供的协程的栈，但协程切换时候，栈的拷贝交换需要手工完成

# librf的目标

综上，resume function被称作C++ coroutine是名至实归。但是，resume function离现代coroutine需要的一些功能，又还差了一点点。因此，librf出生了。

最开始我的设计目标，是用librf来解决异步回调的，包括解决异步回调的怎么驱动着往下走；解决回调发生在另外一个线程的时候，协程被多个线程之间跳转执行，导致显示加锁带来的复杂度提升。所以，当前librf被设计成一个单线程单实例的样子。

为了更好的同步多个协程之间的执行顺序，librf提供了mutex，channel原语。mutex用于互斥访问；channel用于协程通信，用于消费/生产者模型等。同时提供了定时器，用于暂停协程，和支持mutex等的超时功能。

librf推荐将异常通过state\_t::set\_exception()/throw\_exception()函数，抛给协程来处理。而不是由callback代码直接处理掉。

目前，librf还未过多关注性能问题，只是由于stackless的天性，对内存占用很有信心。所以，下一步的目标，是稳定librf，将librf用于对性能不是那么关注的服务器领域，以及C++客户端逻辑。

欢迎大家关注librf，并对librf提出改进意见。下面是librf在GitHub上的地址：

<https://github.com/tearshark/librf>

感谢看到这里。至此，resume function实践一文就讲到这里。下面是本文相关的资料和范例代码地址：

<https://github.com/tearshark/resumef>