Resumef实践

一、从lambda说起，编译器如何完成仿函数

二、再说coroutine，编译器可能如何完成协程类

三、resume function的两种形式：yield+await

四、编译器实现await的细节

五、一个await的范例

六、一个mysql+callback的范例

七、通用的绿化callback

八、具体绿化mysql+callback

九、一个message box的范例

十、asio+callback的传统应用

十一、resume function如何与asio系统工作

十二、总结resume function要解决的痛点

十三、对比三种协程的优缺点

十四、librf的协程调度

十五、librf如何处理异常

十六、librf如何处理多线程

十七、librf之sleep()

十八、librf之mutex

十九、librf之channel

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

演讲议题形式  
演讲时长：40 分钟，包括提问环节  
演讲内容标准  
主题：申请者需要有明确的主题观点，内容新颖，有深度，能够帮助听众在知识、实操等方面有所启发。  
经历：申请者在相关领域的成就以及从业经历，我们也会作为评审的参考标准。  
感受：我们非常重视听众在参加演讲分享过后的收获，这同时也是举办技术会议的重要意义。

有什么特色吗  
你可以深入讲一下，对比一下  
做这个的动机

Resumef+asio

Resumef+messageBox

Resumef+net应答

Resumef+异步阻塞操作

Resumef实现概要

Resumef+提前准备好该怎么做

Resumef+协程

优化协程调度

asio 服务器，使用lambda完成do\_something\_1/2/3，  
asio 服务器，使用rf完成do\_something\_1/2/3  
cocos2d-x 客户端，使用rf完成msgbox  
多线程下的rf  
librf解决什么  
librf如何处理成协程  
librf如何处理异常  
librf如何处理多线程  
librf如何处理先完成/后调用的情况《？》  
librf如何实现mutex

对现有callback代码的保护

跟第三方异步代码的集成难易度

从lambda说起

|  |  |
| --- | --- |
| struct lambda  {  mutable int **value**;  lambda(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*; |

显然，编译器辅助生成了仿函数的所有一切。

再说说coroutine

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*;

|  |  |
| --- | --- |
| 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*;  } |

Resume function

Yield return

Await

Await的应用

auto async\_ heavy\_computing\_tasks(int64\_t **val**)

{

using namespace std::*chrono*;

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

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

{

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

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

}).*detach*();

return **awaitable**.get\_future();

}

resumef::future\_vt heavy\_computing\_ sequential(int64\_t **val**)

{

std::*cout* << **val** << std::*endl*;

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

std::*cout* << **val** << std::*endl*;

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

std::*cout* << **val** << std::*endl*;

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

std::*cout* << **val** << std::*endl*;

}

resumef::future\_t<int64\_t> heavy\_computing\_loop(int64\_t **val**)

{

std::*cout* << **val** << std::*endl*;

for (int **i** = 0; **i** < 5; ++**i**)

{

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

std::*cout* << **val** << std::*endl*;

}

return **val**;

}

Resume function的实现细节

Given a user authored function :

*R* foo(T1 a, T2 b) { T2 c;body - containing - *suspend* - resume - points }

Compiler can constructs **a** function that behaves as if the following code was generated :

*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;

}

}

引用自：N4134-<Resumable Functions v2>

阻塞操作：阻塞io/mysql client

struct AsyncNode;

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

struct SynchResult;

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

enum struct AsyncOperator{

Select, Update, Insert

};

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;

}

void AsyncQueryThread(){

SynchMysqlConnect **impl**;

for (;;) {

AsyncNode **an** = std::move(**m\_listAsyncQuery**.*front*());

**m\_listAsyncQuery**.*pop\_front*();

try {

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

if (**an**.cb){

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

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

}

}

catch (...){

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

}

}

}

void MysqlConnectLoop(){

for (SynchResult & **sr** : **m\_listSynchResult**){

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

}

**m\_listSynchResult**.*clear*();

}

AsynUpdate("UPDATE world.city SET Population = Population + 1 WHERE `Name`='Kabul'"s,

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

{

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

});

如何green：

future<T> async\_function()

{

promise<T> prom;

auto st = prom.get\_state();

if (condition is true)

st.set\_value();

else

async\_operator([st]

{

st.set\_value();

})

return prom.get\_future();

}

如何做成支持协程:

inline resumef::future\_t<int64\_t>

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

{

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

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

{

if (!**e**)

st->set\_value(**result** ? **id** : -1);

else

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

};

try {

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

**awaitable**.\_state->set\_value(-1);

}

catch (...) {

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

}

return **awaitable**.get\_future();

}

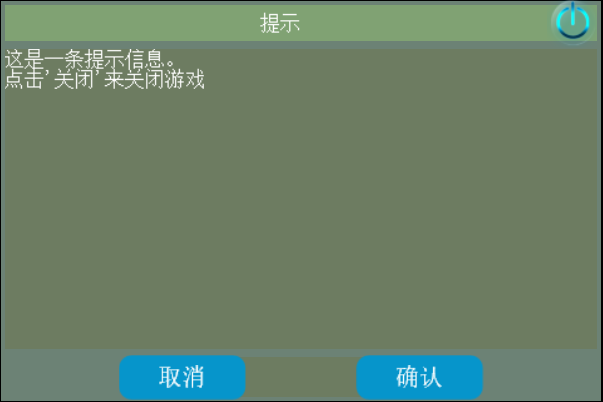
uint64\_t **effectCnt =** co\_awaitmysql\_update(

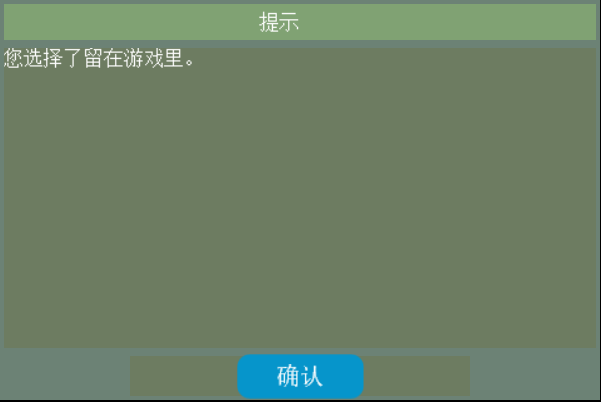
"UPDATE world.city SET Population = Population + 1 WHERE `Name`='Kabul'"s);

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

如何转化为协程使用：

等待操作：delay/wait confirm





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");

});

}

}

);

无协程的情况下：

异步操作：如io/net

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**));

}

});

}

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

{

/// The handler type for the specific signature.

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 future需要的future\_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::future\_t<T> get() { return std::*move*(task\_); }

awaituv::future\_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future\_t<>。

最终使用的时候，就这样使用了：

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

如果使用协程，该如何做：

awaituv::future\_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::future\_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**);

}

当所有这些混合在一起的时候：

Resume function要解决的痛点是

Block operation

Callback hell

Async

Hook

Green

协程的一些现状(这个靠后)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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