async()和future<>：

async()：提供一个接口，让一个callable object，若是可能的话在后台运行，成为一个独立线程。

class future<>：允许你等待线程结束并获取其结果。

例：

int DoSomething(char c)

{

*default\_random\_engine* dre(c);

*uniform\_int\_distribution*<int> id(10, 1000);

// loop to print character after a random period of time

for (int i = 0; i < 10; ++i)

{

*this\_thread*::*sleep\_for*(*chrono*::*milliseconds*(id(dre)));

*cout*.*put*(c).*flush*();

}

return c;

}

int Func1()

{

return DoSomething('.');

}

int Func2()

{

return DoSomething('+');

}

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

{

*cout* << "starting Func1() in background"

<< " and Func2() in foreground:" << *endl*;

// start Func1() asynchronously

// (now or later or never)

// 尝试将Func1异步启动于

// 一个分离线程内

*future*<int> result1(*async*(Func1));

// call Func2() synchronously

int result2 = Func2();

// print result(wait for Func1()

// to finish and add its result to

// result2

int result = result1.*get*() + result2;

*cout* << "\nresult of Func1() + Func2(): " << result

<< *endl*;

return 0;

}

输出为：

starting Func1() in background and Func2() in foreground:

.++...+...+..++.++++

result of Func1() + Func2(): 89

从输出结果.和+交错出现可以看出Func1是异步运行的。

thread：

例：程序stl\_test91

void DoSomething(int num, char c)

{

try

{

*default\_random\_engine* dre(42 \* c);

*uniform\_int\_distribution*<int> id(10, 1000);

for (int i = 0; i < num; ++i)

{

*this\_thread*::*sleep\_for*(*chrono*::*milliseconds*(id(dre)));

*cout*.*put*(c).*flush*();

}

}

catch (const *exception*& e)

{

*cerr* << "Thread-exception (thread "

<< *this\_thread*::*get\_id*() << ") : " << e.*what*() << *endl*;

}

catch (...)

{

*cerr* << "Thread-exception (thread "

<< *this\_thread*::*get\_id*() << ")" << *endl*;

}

}

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

{

try {

// 前台线程

thread t1(DoSomething, 5, '.');

*cout* << "- started fg thread " << t1.*get\_id*() << *endl*;

// print other characters in other background threads

// 后台线程

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

{

thread t(DoSomething, 10, 'a' + i);

*cout* << "- detach started bg thread " << t.*get\_id*() << *endl*;

t.*detach*();

}

*cin*.*get*();

*cout* << "- join fg thread " << t1.*get\_id*() << *endl*;

t1.*join*(); // wait for t1 finish

}

catch (const *exception*& e)

{

*cerr* << "exception: " << e.*what*() << *endl*;

}

*cout* << *endl*;

*cout* << "main thread end" << *endl*;

return 0;

}

输出可能的结果：

- started fg thread 44380

- detach started bg thread 47088

- detach started bg thread 34176

- detach started bg thread 48496

- detach started bg thread 30844

- detach started bg thread 42036

acdea.d

- join fg thread 44380

bec.baeddbdcbce.adb.ddbecca.

main thread end

Mutex和Lock

例：程序stl\_test92

*mutex* print\_mutex;

void Print(const *string*& s)

{

// 如果没有mutex锁

// 输出将混乱

*lock\_guard*<*mutex*> locker(print\_mutex);

for (char c : s)

{

*cout*.*put*(c);

}

*cout* << *endl*;

}

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

{

auto f1 = *async*(*launch*::*async*, Print,

"Hello from a first thread");

auto f2 = *async*(*launch*::*async*, Print,

"Hello from a second thread");

Print("Hello from the main thread");

return 0;

}

输出为：

Hello from the main thread

Hello from a first thread

Hello from a second thread

递归锁：recursive\_mutex，允许多次锁定lock

普通的mutex，如果之前的lock未释放，第二次lock会造成deadlock死锁的问题，递归是recursive\_mutex则不存在这种问题，在最后一次（last）unlock时彻底释放lock。

尝试性lock和带时间性的lock

尝试性lock：

std::mutex m;

// try to acquire a lock and do other stuff

while (m.try\_lock() == false) {

DoSomeOtherStuff();

}

std::lock\_guard<std::mutex> lg(m, std::adopt\_lock);

带有时间性的lock

std::timed\_mutex m;

// try for one second to acquire a lock

if (m.try\_lock\_for(std::chrono::seconds(1))) {

std::lock\_guard<std::timed\_mutex> lg(m, std::adopt\_lock);

}

else {

CouldNotGetTheLock();

}

条件变量：condition variable

例：程序stl\_test93

bool ready\_flag;

*mutex* ready\_mutex;

*condition\_variable* ready\_condition\_variable;

// 激发条件满足线程

void Thread1()

{

*cout* << "<return>" << *endl*;

*cin*.*get*();

// 添加{}用来限定lock\_guard的作用域

{

*lock\_guard*<*mutex*> lg(ready\_mutex);

ready\_flag = true;

} // release lock

ready\_condition\_variable.*notify\_one*();

}

// 等待条件被满足的线程

void Thread2()

{

{

*unique\_lock*<*mutex*> ul(ready\_mutex);

ready\_condition\_variable.*wait*(ul, [] {return ready\_flag; });

} // release lock

*cout* << "done" << *endl*;

}

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

{

auto f1 = *async*(*launch*::*async*, Thread1);

auto f2 = *async*(*launch*::*async*, Thread2);

return 0;

}

输出为：

<return>

done

条件变量的多线程队列

例：程序stl\_test94

*queue*<int> queue\_;

*mutex* queue\_mutex;

*condition\_variable* queue\_condition\_variable;

void Provider(int value)

{

for (int i = 0; i < 6; ++i)

{

{

*lock\_guard*<*mutex*> lg(queue\_mutex);

queue\_.*push*(value + i);

} // release lock

queue\_condition\_variable.*notify\_one*();

//queue\_condition\_variable.notify\_all();

*this\_thread*::*sleep\_for*(*chrono*::*microseconds*(value));

}

}

void Consumer(int number)

{

while (true)

{

int value;

{

*unique\_lock*<*mutex*> ul(queue\_mutex);

queue\_condition\_variable.*wait*(ul, [] {return !queue\_.*empty*(); });

value = queue\_.*front*();

queue\_.*pop*();

} // release lock

*cout* << "consumer " << number << ": " << value << *endl*;

}

}

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

{

// start three providers for values 100+, 300+, and 500+

auto p1 = *async*(*launch*::*async*, Provider, 100);

auto p2 = *async*(*launch*::*async*, Provider, 300);

auto p3 = *async*(*launch*::*async*, Provider, 500);

// start two consumers printing the values

auto c1 = *async*(*launch*::*async*, Consumer, 1);

auto c2 = *async*(*launch*::*async*, Consumer, 2);

return 0;

}

可能输出为：

consumer 1: 100

consumer 1: 300

consumer 1: 500

consumer 1: 501

consumer 1: 301

consumer 1: 101

consumer 1: consumer 2: 502

102

consumer 2: 302

consumer 1: 103

consumer 1: 303

consumer 1: 503

consumer 2: 104

consumer 2: 304

consumer 2: 504

consumer 1: 105

consumer 1: 505

consumer 2: 305

从输出可以看出，两个consumer的输出并非同步的，而且并发等待者的被通知顺序也是不确定的。

atomic，原子操作

例：程序stl\_test95

long data\_;

*atomic*<bool> ready\_flag(false);

void Provider()

{

// after reading a character

*cout* << "<return>" << *endl*;

*cin*.*get*();

data\_ = 42;

// atomic不需要mutex和lock

ready\_flag.*store*(true);

}

void Consumer()

{

while (!ready\_flag.*load*())

{

*cout*.*put*('.').*flush*();

*this\_thread*::*sleep\_for*(*chrono*::*milliseconds*(500));

}

*cout* << "\nvalue : " << data\_ << *endl*;

}

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

{

auto p = *async*(*launch*::*async*, Provider);

auto c = *async*(*launch*::*async*, Consumer);

return 0;

}

可能的输出为：

<return>

.....

value : 42