

C++ boost

2022年11月28日 9:49

C++ boost

<https://www.boost.org/>

<https://www.boost.org/doc/>

太多了。。。

但是找不到 asio, coroutine2。。找到了：

https://www.boost.org/doc/libs/1_80_0/libs/libraries.htm

=====

=====

https://www.boost.org/doc/libs/1_80_0/doc/html/lambda.html

Chapter 18. Boost.Lambda

Boost Lambda Library (BLL) 是一个 C++模板库，为C++实现了一种 lambda 抽象。
术语(。。应该是指 lambda abstractions, lambda 抽象)来源于 函数式编程 和 lambda演算，lambda抽象 定义了一个 无名函数。
BLL的主要动机是 提供 灵活 方便 的方式 来 为 STL算法 定义 无名函数对象。

a line of code says more than a thousand words;

下面的代码 输出 STL容器中的 元素 并 以 空格 分隔

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

表达式 std::cout << _1 << ' ' 定义了一个 一元函数对象。 变量 _1 是 这个函数的 参数，一个 实参的 占位符。 在 for_each 的 每次 迭代中，这个 函数 被调用， a的一个 元素作为实参。这个实参 代替 占位符，函数 就会推导了。

BLL 的本质是 让你 直接在 STL 算法中 定义 短的匿名函数对象。

。。。这个。。C++已经有 lambda 了， 我看文档 Copyright © 1999–2004 Jaakko Järvi, Gary Powell 。 所以是以前的。。估计 标准没有完全使用这套。

。。但是很coool

=====

Chapter 14. Boost.Function

https://www.boost.org/doc/libs/1_80_0/doc/html/function.html

Boost.Function 库 包含了 函数对象wrapper的一组类模板。

类似于 广义的callback。

它和 函数指针的 共同之处在于，两者 都定义了一个 调用接口， 通过这个接口 可以调用 某些实现，并且 调用的 实现 在 整个程序过程中 可能会发生变化。

一般来说，任何 使用 函数指针 来 defer 一个调用 或 make一个回调 的地方，
Boost.Function 都可以被用来 替代 函数指针 来允许 用户 对目标的 更灵活的 实现。 目
标 可以是 任何 'compatible' 函数对象 (或函数 指针)，意味着 Boost.Function 指定的
接口的参数 可以被转换为 目标函数对象的 参数。

Boost.Function 有2种语法形式: preferred 形式和 portable 形式。

preferred形式 更适合 C++语言，减少了 需要考虑的 单独模板参数的 数量，提供了 可读性。

但是， preferred 形式 不支持 所有的 平台，由于 编译器 bug。

compatible form 在 Boost.Function 支持的 所有编译器上 可用。

根据下面的表 来决定 哪种语法形式 适用于 你的 电脑。

Preferred syntax

>= GNU C++ 2.95x, 3.0.x
Comeau C++ 4.2.45.2
SGI MIPSpro 7.3.0
Intel C++ 5.0, 6.0
>= Microsoft Visual C++ 7.1

Portable syntax

所有支持 preferred 语法的 编译器
Microsoft Visual C++ 6.0, 7.0
Borland C++ 5.5.1
Sun WorkShop 6 update 2 C++ 5.3
Metrowerks CodeWarrior 8.1

基本用法

function wrapper 的定义很简单， function 是用 所需的 返回类型 和 参数类型 实例化

函数类模板，并将其公式化为 C++ function 类型。

可以提供任意数量的参数，最多达到一些impl 的限制(10是默认的最大值)。

下面的描述了一个接受 2个 int 参数，返回一个 float 的函数对象封装器 f
Preferred syntax

```
boost::function<float (int x, int y)> f;
```

Portable syntax

```
boost::function2<float, int, int> f;
```

默认下，函数对象封装器是空的，所以我们创建一个函数对象，赋值给 f

```
struct int_div {  
    float operator()(int x, int y) const { return ((float)x)/y; };  
};  
  
f = int_div();
```

现在，我们可以使用 f 来执行 int_div

```
std::cout << f(5, 3) << std::endl;
```

我们可以自由滴将任何兼容的函数对象分配给 f。如果 int_div 被声明为接受 2个 long，则隐式转换将应用于参数。参数的类型的唯一限制是它们是 CopyConstructible，所以我们可以使用 ref 和数组：

Preferred syntax

```
boost::function<void (int values[], int n, int& sum, float& avg)> sum_avg;
```

Portable syntax

```
boost::function4<void, int*, int, int&, float&> sum_avg;
```

```
void do_sum_avg(int values[], int n, int& sum, float& avg)  
{  
    sum = 0;  
    for (int i = 0; i < n; i++)  
        sum += values[i];  
    avg = (float)sum / n;  
}  
  
sum_avg = &do_sum_avg;
```

调用实际没有函数对象的函数对象wrapper是一个违反，和调用了值为null的函数指针类似，会抛出一个 bad_function_call 异常。

我们可以先检查函数对象wrapper是否为空（如果非空，则返回 true），或者和 0 比较。

```
if (f)  
    std::cout << f(5, 3) << std::endl;  
else  
    std::cout << "f has no target, so it is unsafe to call" << std::endl;
```

另一方面，empty() 方法 会 返回 wrapper 是否为空

最后，我们可以 清除 函数target，通过 赋值为 0 或 调用 clear() 方法

```
f = 0;
```

Free functions

free 函数指针 可以被视为 具有 常量函数 调用运算符 的单例函数对象，因此可以直接与函数对象包装器 一起使用

```
float mul_ints(int x, int y) { return ((float)x) * y; }
```

```
f = &mul_ints;
```

注意 & 可以省略，除非 你使用 Microsoft Visual C++ version 6

Member functions

许多系统中，回调 经常调用 特定对象 的成员函数。这通常被称为 argument binding，超出了 Boost.Function 的范围。

只 支持 直接使用 成员函数，下面的代码是 有效的：

```
struct X {
    int foo(int);
};
```

Preferred syntax

```
boost::function<int (X*, int)> f;
f = &X::foo;
X x;
f(&x, 5);
```

Portable syntax

```
boost::function2<int, X*, int> f;
f = &X::foo;
X x;
f(&x, 5);
```

有几个库支持 参数绑定，下面的总结了 3个这样的库

Bind。这个库允许 为任何函数对象 绑定参数。轻量级。

C++标准库。将 std::bindlst 和 std::mem_fun 一起使用，可以将 指针的对象 绑定到 成员函数，以便于 Boost 一起使用

Preferred syntax

```
boost::function<int (int)> f;
X x;
f = std::bindlst(
    std::mem_fun(&X::foo), &x);
f(5); // Call x.foo(5)
```

Portable syntax

```

boost::function<int, int> f;
X x;
f = std::bind1st(
    std::mem_fun(&X::foo), &x);
f(5); // Call x.foo(5)

```

Lambda库(boost的)。这个库 提供了 类似C++语法的 构建 函数对象的 机制。

对Function 对象的 ref

有时, 让 Boost.Function 克隆一个 函数对象 是 昂贵的(或语义错误的)。
这种情况下, 就要求 Boost.Function 对 真实的 函数对象 只保留一个ref。
这通过 ref 或 cref 函数 来 封装 函数对象的 引用 来完成。

Preferred syntax

```

stateful_type a_function_object;
boost::function<int (int)> f;
f = boost::ref(a_function_object);
boost::function<int (int)> f2(f);

```

Portable syntax

```

stateful_type a_function_object;
boost::function<int, int> f;
f = boost::ref(a_function_object);
boost::function<int, int> f2(f);

```

这里, f 不会创建 a_function_object 的 克隆, f2 也不会, f 和 f2 都指向了 a_function_object。

另外, 在使用 函数对象的引用时, Boost.Function 在 赋值 和 构造 期间 不会 抛出异常。

Boost.Function function objects 的 比较

函数对象封装器 可以通过 == 或 != 来和 wrapper 中的 任何函数对象 比较。
如果 封装器 包含了 那个类型的 函数对象, 它会和 给与的 函数对象 比较 (必须 要么是 EqualityComparable 或 boost::function_equal):

```

int compute_with_X(X*, int);

f = &X::foo;
assert(f == &X::foo);
assert(&compute_with_X != f);

```

当和 reference_wrapper 的实例 比较时, reference_wrapper 中的 对象的 地址 和 函数 对象封装器 中的 对象的 地址 相比。

```

a_stateful_object so1, so2;
f = boost::ref(so1);
assert(f == boost::ref(so1));

```

```
assert(f == so1); // Only if a_stateful_object is EqualityComparable
assert(f != boost::ref(so2));
```

```
// In header: <boost/function.hpp>
class bad_function_call : public std::runtime_error {
public:
    // construct/copy/destruct
    bad_function_call();
};
```

```
// In header: <boost/function.hpp>
class function_base {
public:
    // capacity
    bool empty() const;
    // target access
    template<typename Functor> Functor* target();
    template<typename Functor> const Functor* target() const;
    template<typename Functor> bool contains(const Functor&) const;
    const std::type_info& target_type() const;
};
```

```
=====
```

Chapter 1. Boost.Bind

https://www.boost.org/doc/libs/1_80_0/libs/bind/doc/html/bind.html

boost::bind 是 标准函数 std::bind1st 和 std::bind2nd 的 generalization。

它支持 任意的 函数对象，函数，函数指针，成员函数指针，可以绑定 任意参数 到 一个 特定值 或 将输入参数 路由到 任意位置。

bind 不会对 函数对象 提出任何 要求。特别是，它不需要 result_type, first_argument_type, second_argument_type 这些标准typedefs

和函数，函数指针 使用 bind

现有下面的 定义

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

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

bind(f, 1, 2) 会提供 一个 “nullary”(。。零元) 函数对象，不接受参数，返回 f(1, 2)。
类似的 bind(g, 1, 2, 3) () 等价于 g(1, 2, 3)

可以选择性地 仅绑定某些参数。 bind(f, _1, 5)(x) 等价于 f(x, 5) 。 _1 是 占位符，
意味着 第一个实参。

。。。不知道有没有人问过，为什么 不是 _0

为了进行比较，下面是 标准库 提供的 相同效果的 操作

```
std::bind2nd(std::ptr_fun(f), 5)(x);
```

bind 也提供了 std::bind1st 的功能

```
std::bind1st(std::ptr_fun(f), 5)(x); // f(5, x)
bind(f, 5, _1)(x); // f(5, x)
```

bind 能处理 多于2个形参 的 函数，并且 参数替换机制 更普遍

```
bind(f, _2, _1)(x, y); // f(y, x)
bind(g, _1, 9, _1)(x); // g(x, 9, x)
bind(g, _3, _3, _3)(x, y, z); // g(z, z, z)
bind(g, _1, _1, _1)(x, y, z); // g(x, x, x)
```

注意，在最后一个例子中， bind(g, _1, _1, _1) 生成的 函数对象 不会包含 任何参数的 ref， 除了第一个，但它仍然可以 在 多于1个参数的 情况下使用。

任何外部参数可以被 安静滴忽视，就像上面的 第三个例子中 忽视了 第一个 和 第二个参数。

bind 获得的 参数 是 复制后的，被 返回的 函数对象 hold。 例如，下面的代码

```
int i = 5;
bind(f, i, _1);
```

函数对象中 保存的是 i的值 的 copy。

boost::ref, boost:: cref 能被用来 创建 保存 对象的ref (而不是copy) 的 函数对象：

```
int i = 5;
bind(f, ref(i), _1);
bind(f, cref(i), _1);
```

和函数对象 使用 bind

bind 不限于 和函数一起。 它接受各种 函数对象。

在一般情况下，必须显式指定生成 的函数对象的 operator() 的返回类型（如果没有 typeid，则无法推断返回类型）

```
struct F
{
    int operator()(int a, int b) { return a - b; }
    bool operator()(long a, long b) { return a == b; }
};

F f;
int x = 104;
bind<int>(f, _1, _1)(x); // f(x, x), i.e. zero
```

一些编译器 bind<R>(f, ...) 语法 会有问题，一个可选的方法 是：

```
boost::bind(boost::type<int>(), f, _1, _1)(x);
```

注意这个语法只是 解决方案，不是 接口的一部分。

当 函数对象 暴露了一个 名为 result_type 的 内嵌类型，显式的return type 可以省略

```
int x = 8;
bind(std::less<int>(), _1, 9)(x); // x < 9
```

注意，省略 返回类型 的 能力 不是 所有 编译器 都有

默认下，bind 生成 给定的函数对象的 副本。boost::ref, boost::cref 可以被用来 使得 它 保存 函数对象 的 ref，而不是 copy。当 函数对象 是 不可复制，复制消耗大，包含状态的 时是有用的。当然，这种情况下，程序员 被期望 能 确保 函数对象 在 运行时，函数对象 没有被 销毁。

```
struct F2
{
    int s;

    typedef void result_type;
    void operator()(int x) { s += x; }
};

F2 f2 = { 0 };
int a[] = { 1, 2, 3 };

std::for_each(a, a+3, bind(ref(f2), _1));

assert(f2.s == 6);
```

和指向成员的 指针 使用 bind

指向 成员函数的 指针 和 指向 数据成员的 指针 不是 函数对象，因为它们 不支持

operator()。

bind 接受 成员 指针 作为 它的第一个参数， 其行为 就像 使用了 boost::mem_fn 来 转换 成员指针 到 函数对象。 换句话说，表达式

bind(&X::f, args)

等价于

bind<R>(mem_fn(&X::f), args)

对于 成员函数 R是 X::f 的 返回类型，

对于 数据成员 R 是 成员的类型

例子

```
struct X
{
    bool f(int a);
};

X x;
shared_ptr<X> p(new X);
int i = 5;

bind(&X::f, ref(x), _1)(i);           // x.f(i)
bind(&X::f, &x, _1)(i);               // (&x)->f(i)
bind(&X::f, x, _1)(i);                // (internal copy of x).f(i)
bind(&X::f, p, _1)(i);                // (internal copy of p)->f(i)
```

最后2个例子有趣于 它们生成了 'self-contained' 函数对象。

bind(&X::f, x, _1) 保存了 x 的副本。

bind(&X::f, p, _1) 保存了 p 的副本，由于 p 是 boost::shared_ptr，函数对象 保持了 对 X 实例的 引用，即使 p 超出了 范围 或 reset()。

Using nested binds for function composition

bind可以内嵌一个bind

```
bind(f, bind(g, _1))(x);           // f(g(x))
```

当 函数对象被调用时，内部的bind 在 外部的outer 之前 被推导，(多个内部bind之间的推导顺序是 未定义的)， 外部outer 推导时，内部bind 的结果 取代占位符。

bind的特性 可以用来 执行 函数组合。 可以使用 bind 完成 类似 Boost.Compose 的功能

注意，第一个参数，被绑定的函数对象 没有被推导，即使它是 bind 产生的 函数对象 或 一个 占位符参数， 所以下面的代码 不会 如预期般工作：

```
typedef void (*pf)(int);
```

```
std::vector<pf> v;
std::for_each(v.begin(), v.end(), bind(_1, 5));
```

可以通过 一个 助手函数对象 apply 来完成 期望的效果， apply 方法 会 将其 第一个参

数 作为 函数对象 应用于其参数列表的 其余部分。

为了方便起见, apply.hpp 头文件中提供了 apply 的实现:

```
typedef void (*pf)(int);

std::vector<pf> v;
std::for_each(v.begin(), v.end(), bind(apply<void>(), _1, 5));
```

默认下, 第一个参数不会被推导, 其他参数会。

有时, 我们 不要 eval 第一个参数 后的 参数, 即使它们是 嵌套的 bind 子表达式。

这个可以通过另一个 助手函数对象 protect 来实现, 它屏蔽了类型, 所以 bind 无法 识别 和 eval 它。当被调用时, protect 只是 转发参数列表 给 其他函数对象, 不会进行修改。

protect.hpp 头文件包含了 protect 的实现。

Overloaded operators (new in Boost 1.33)

为了方便, bind 生成的 函数对象 重载了 逻辑非operator!, 和 关系与逻辑操作符 ==, !=, <, <=, >, >=, &&, ||。

!bind(f, ...) 等价于 bind(logical_not(), bind(f, ...)), logical_not() 是一个 函数对象: 接受一个参数 x, 返回 !x

bind(f, ...) op x, 其中 op 是一个 关系 或 逻辑 运算符, 等价于 bind(relation(), bind(f, ...), x), relation 是一个 函数对象, 接受2个参数 a,b, 返回 a op b

你可以方便地 对 bind 的 结果取反

```
std::remove_if(first, last, !bind(&X::visible, _1)); // remove invisible
objects
```

将bind 的结果 和 值 进行比较

```
std::find_if(first, last, bind(&X::name, _1) == "Peter");
std::find_if(first, last, bind(&X::name, _1) == "Peter" || bind(&X::name, _1) == "Paul");
```

和 占位符 比较

```
bind(&X::name, _1) == _2
```

和另一个 bind 表达式比较

```
std::sort(first, last, bind(&X::name, _1) < bind(&X::name, _2)); // sort by
name
```

例子

和标准算法一起使用 bind

```
class image;
```

```
class animation
```

```

{
public:
    void advance(int ms);
    bool inactive() const;
    void render(image & target) const;
};

std::vector<animation> anims;

template<class C, class P> void erase_if(C & c, P pred)
{
    c.erase(std::remove_if(c.begin(), c.end(), pred), c.end());
}

void update(int ms)
{
    std::for_each(anims.begin(), anims.end(), boost::bind(&animation::advance,
1, ms));
    erase_if(anims, boost::mem_fn(&animation::inactive));
}

void render(image & target)
{
    std::for_each(anims.begin(), anims.end(), boost::bind(&animation::render,
1, boost::ref(target)));
}

```

◦ ◦ ◦

和Boost.Function 使用 bind

```

class button
{
public:
    boost::function<void()> onClick;
};
```

```

class player
{
public:
    void play();
    void stop();
};
```

```

button playButton, stopButton;
player thePlayer;
```

```

void connect()
{
    playButton.onClick = boost::bind(&player::play, &thePlayer);
```

```
    stopButton.onClick = boost::bind(&player::stop, &thePlayer);  
}
```

限制

bind 生成的 函数对象 通过 ref 来获得 参数， 所以无法 接收 非const临时变量 或 字面常量。 这是 C++ 2003 固有的局限性，被称为 转发问题， 它会在 C++0x 中修复。 。 。 那现在 C++17， bind 可以接收了吗？ 感觉 还是不行啊， ref的问题啊。

库使用了下面的签名格式

```
template<class T> void f(T & t);
```

来接受 任意类型的 参数，并 不做修改地 传送它们。注意，这个 不能用于 非const的右值。

在支持 部分排序 函数模板 的编译器中，一个可能的 解决方案是 增加一个 重载

```
template<class T> void f(T & t);  
template<class T> void f(T const & t);
```

不幸的是，如果有9个参数，那么需要提供 512个重载，这是不切合实际的。

库选择了一个 小的子集：对于最多 2个参数，它提供了完整的 常量重载，对于 3个 或更多 参数，它提供了一个额外的 重载，其中包含 常量引用 所取的 所有参数。这覆盖了用例的 合理部分

故障排除

不正确的参数个数

bind(f, a1, a2, ..., aN) 表达式中，f 必须接受 N 个参数。这个错误通常在 bind time 时 被发现， 换句话说，在 bind() 被调用的 那行上 会出现 编译错误。

```
int f(int, int);  
  
int main()  
{  
    boost::bind(f, 1);      // error, f takes two arguments  
    boost::bind(f, 1, 2); // OK  
}
```

这个错误的 一个变形是 忘记了 成员函数 有一个 隐式的 this 参数

```
struct X  
{  
    int f(int);  
}  
  
int main()  
{  
    boost::bind(&X::f, 1);      // error, X::f takes two arguments  
    boost::bind(&X::f, _1, 1); // OK
```

```
}
```

The function object cannot be called with the specified arguments
在普通函数调用时， 被绑定的函数对象 必须能 兼容 参数列表。

编译器会在 call time 时 发现 不兼容，在 bind.hpp 的 某行 抛出error，这行类似：

```
    return f(a[a1_], a[a2_]);  
。。这个是 bind.hpp 中的。
```

这种错误的例子：

```
int f(int);  
  
int main()  
{  
    boost::bind(f, "incompatible");      // OK so far, no call  
    boost::bind(f, "incompatible")();    // error, "incompatible" is not an  
    int  
        boost::bind(f, _1);            // OK  
        boost::bind(f, _1)("incompatible"); // error, "incompatible" is not an  
    int  
}  
。。带() 就是 call， 不带 只是声明/定义。
```

Accessing an argument that does not exist

在call time时， _N 占位符 选择 参数列表中 第 N 个， 向后越界的话 会 error

```
int f(int);  
  
int main()  
{  
    boost::bind(f, _1);                  // OK  
    boost::bind(f, _1)();                // error, there is no argument  
    number 1  
}
```

错误是在 bind.hpp 的 下面这行 爆出的

```
return f(a[a1_]);
```

当模仿 std::bind1st(f, a) 时， 一个常见的错误是 使用 bind(f, a, _2) 而不是 正确的 bind(f, a, _1)

Inappropriate use of bind(f, ...)

不恰当的使用

bind(f, a1, a2, a3 .. aN) 格式 导致 f的类型的 自动识别。 它不能对 所有 函数对象起效； f 必须是 函数 或 成员函数 指针

可以将这种格式 和 定义了 result_type 的 函数对象 一起使用，但 仅适用于 支持 partial specialization 和 partial ordering 的 编译器。 MSVC 在7.0之前 不支持这种函数对象的 语法

Inappropriate use of bind<R>(f, ...)

这种形式 支持 所有 函数对象。

可以(但不推荐) 对 函数 和 成员函数 指针 使用这种形式， 但只有在 那些 支持 partial ordering 的编译器上才可以。

MSVC 7.0 之前 不支持 函数 和 成员函数 指针 的 语法

Binding a nonstandard function

Binding an overloaded function

尝试绑定重载的函数， 经常会error， 而且 不会告诉你 绑定了 哪个。

这是一个常见的问题：一个成员函数 有 2个 重载， const 和 非const

```
struct X
{
    int& get();
    int const& get() const;
};

int main()
{
    boost::bind(&X::get, _1);
}
```

这种二义性 可以通过 手动 强转 (成员)函数 指针 为 期望的类型 来解决

```
int main()
{
    boost::bind(static_cast< int const& (X::*) () const >(&X::get), _1);
}
```

另一种 可读性更佳 的方式 是 引入一个 临时变量

```
int main()
{
    int const& (X::*get) () const = &X::get;
    boost::bind(get, _1);
}
```

Modeling STL function object concepts

=====

Chapter 29. Boost.Program_options

https://www.boost.org/doc/libs/1_80_0/doc/html/program_options.html

program_options 库 允许 开发人员 获得 program options，这个是 用户的 (name, value)对，通过 常规的方法：如 命令行 和 配置文件

为什么你需要这么一个库，为什么这个库比你通过手写代码来转换命令行 更好？

1. 更简单，声明option 的语法很简单，库本身很小。 将option的值转为想要的类型，保存到 代码的变量中，这些事情是自动处理的
2. 错误报告更好。命令行中的所有问题 都会被报告。手写的代码可能 不能正确地 转换输入。而且，可以自动生成 使用消息，以避免 和 实际的 option 不同。
3. 可以从任何地方读取 option。命令行可能不够用，你可能希望 配置文件 或 环境变量。

本节 会展示 program_options 库的一般用法。

完整的代码可以从 BOOST_ROOT/libs/program_options/example 中找到。

在下面所有的例子中，我们假设 下面的 命名空间别名 起效

```
namespace po = boost::program_options;
```

第一个例子是最简单的，只处理2个 option，下面是源码

```
// Declare the supported options.  
po::options_description desc("Allowed options");  
desc.add_options()  
    ("help", "produce help message")  
    ("compression", po::value<int>(), "set compression level")  
;
```

。。这个写法是 把自己(函数) 返回了？？？但是 2次 调用 的参数 不同啊。
。。这里应该是 声明 这个 desc 要抓那些 option，下面说了
。。。但是 连operator() 确实没有看懂。

```
po::variables_map vm;  
po::store(po::parse_command_line(ac, av, desc), vm);  
po::notify(vm);  
  
if (vm.count("help")) {  
    cout << desc << "\n";  
    return 1;  
}
```

```

if (vm.count("compression")) {
    cout << "Compression level was set to "
    << vm["compression"].as<int>() << ".\n";
} else {
    cout << "Compression level was not set.\n";
}

```

一开始 我们使用 options_description类 声明了 所有 允许的 options。
这个类的 add_options 方法 返回 一个 特殊的 代理对象，它定义了operator()。
调用这个 operator 来 实际声明 options。 参数是 option的名字，value的信息，描述。
上面的 例子中，第一个 option 没有 value， 第二个 有一个 int类型的 value

然后，定义一个 variables_map 类型的 对象。 这个类 用来 存储 option 的值，值的 类型不限。
然后，调用 store, parse_command_line 和 notify 函数 来使得 vm 包含 命令行 中所有 找到的 option

最后，我们可以 使用 option. variables_map 类 能像 std::map 一样使用， 除了 存储 的 value 必须 通过 as 方法来检索出来。 (如果 as 声明的 类型 和 实际存储的类型 不同，会抛出 异常)

使用：

```

$ bin/gcc/debug/first
Compression level was not set.
$ bin/gcc/debug/first --help
Allowed options:
  --help                  : produce help message
  --compression arg       : set compression level
$ bin/gcc/debug/first --compression 10
Compression level was set to 10.

```

一个option的value 的类型 肯定不止int。

假设 我们在写一个 编译器，它应该有 优化等级，include的路径，输入文件。

下面描述 这些 option

```

int opt;
po::options_description desc("Allowed options");
desc.add_options()
    ("help", "produce help message")
    ("optimization", po::value<int>(&opt)->default_value(10),
     "optimization level")
    ("include-path, I", po::value< vector<string> >(),
     "include path")
    ("input-file", po::value< vector<string> >(), "input file")
;

```

help 和之前的 类似。这是一个 应该在所有 case 都有的idea

optimization 选项 展示了2个 新的 功能，

第一，我们 指定了 变量的 地址(&opt)。在存储value后，这个 变量 会有 option的 值。

第二，我们制定了一个 默认值10，如果用户没有定义 value ， 那么就使用这个

include-path 选项 是 options_description 类的 接口 仅服务于 一个 源(命令行) 的唯一情况的 例子。

用户通常 喜欢 对常用 选项 使用 短选项名称， include-path, I 指定短命是 I。所以 “**--include-path” 和 “-I” 都可以使用。**

注意 include-path 的类型是 std::vector。 库 对 vector 有特殊支持： 可以多次 定义 option，这些 指定的value 会被 收集到 一个 vector 中。

input-file 选项指定了 要处理 哪些文件：

```
compiler --input-file=a.cpp
```

对比： compiler a.cpp 上面 的 有些不太标准。

库中 将那些 不需要 name 就可以 输入的 option 称为 positional options。 要 使用 a.cpp 代替 --input-file=a.cpp， 我们需要下面的 代码：

```
po::positional_options_description p;
p.add("input-file", -1);

po::variables_map vm;
po::store(po::command_line_parser(ac, av).
          options(desc).positional(p).run(), vm);
po::notify(vm);
```

头2行表明： 所有 positional option 都应该被 转为 input-file 选项。

注意，我们使用 command_line_parser 类来 转换 命令行， 而不是
parse_command_line。后者是 对简单用例的方便的封装，但是现在 我们需要传递 附加信息。

现在，所有的 选项 被描述 和 转换。

打印 options：

```
if (vm.count("include-path"))
{
    cout << "Include paths are: "
        << vm["include-path"].as<vector<string>>() << "\n";
}

if (vm.count("input-file"))
{
    cout << "Input files are: "
        << vm["input-file"].as<vector<string>>() << "\n";
}

cout << "Optimization level is " << opt << "\n";
```

用法:

```
$ bin/gcc/debug/options_description --help
Usage: options_description [options]
Allowed options:
  --help           : produce help message
  --optimization arg   : optimization level
  -I [ --include-path ] arg : include path
  --input-file arg    : input file
$ bin/gcc/debug/options_description
Optimization level is 10
$ bin/gcc/debug/options_description --optimization 4 -I foo -I
another/path --include-path third/include/path a.cpp b.cpp
Include paths are: foo another/path third/include/path
Input files are: a.cpp b.cpp
Optimization level is 4
```

有一个小问题，仍然可以指定 `--input-file`，并且 使用信息 也会这样说，会使得用户困惑。最好是 隐藏这些信息。

Multiple Sources

让用户使用命令行输入options。。。

最好是提供一个 默认配置文件

当然，需要合并 命令行 和 配置文件的 值。例如，优化等级 应该是命令行 覆盖 配置文件。 `include-path` 应该是 合并。

让我们看下面的代码，有2个有趣的细节

1. 我们声明了 多个 `options_description` 类的 实例。因为，一些option，比如 `input-file`，不应该 出现在 自动帮助信息 中。一些option，仅在配置文件中 才有意义。最后，在 帮助信息中 包含 一些 结构，而不是 option 的 长的 list，这很好。

让我们声明几个 option group

```
// Declare a group of options that will be
// allowed only on command line
po::options_description generic("Generic options");
generic.add_options()
  ("version,v", "print version string")
  ("help", "produce help message")
;

// Declare a group of options that will be
// allowed both on command line and in
// config file
po::options_description config("Configuration");
config.add_options()
  ("optimization", po::value<int>(&opt)->default_value(10),
   "optimization level")
```

```

    ("include-path, I",
     po::value< vector<string> >() ->composing(),
     "include path")
;

// Hidden options, will be allowed both on command line and
// in config file, but will not be shown to the user.
po::options_description hidden("Hidden options");
hidden.add_options()
    ("input-file", po::value< vector<string> >(), "input file")
;

```

注意 include-path 的 composing 方法。它告诉库，来自不同source 的 value 应该被组合起来。

options_description 类的 add 方法能用来 group 这些 option。

```

po::options_description cmdline_options;
cmdline_options.add(generic).add(config).add(hidden);

po::options_description config_file_options;
config_file_options.add(config).add(hidden);

po::options_description visible("Allowed options");
visible.add(generic).add(config);

```

value 的解析和存储按照通常的模式，除了我们额外调用了 parse_config_file，并且调用 store 2次。

但是如果相同的值在命令行和配置文件中都被指定了，会发生什么？通常，使用先保存的值，这是 --optimization 的处理逻辑，对于 include-file，值被合并起来

```

$ bin/gcc/debug/multiple_sources
Include paths are: /opt
Optimization level is 1
$ bin/gcc/debug/multiple_sources --help
Allows options:

```

```

Generic options:
  -v [ --version ]      : print version string
  --help                 : produce help message

```

```

Configuration:
  --optimization n       : optimization level
  -I [ --include-path ] path : include path

```

```

$ bin/gcc/debug/multiple_sources --optimization=4 -I foo a.cpp b.cpp
Include paths are: foo /opt
Input files are: a.cpp b.cpp
Optimization level is 4

```

第一次调用 使用 配置文件中的 值，第二次 使用 命令行的值。

我们可以看到 命令行 和 配置文件 include path 合并到一起了。 optimization 是从 命令行获取

在上面，我们看到了一些 库用法的例子。 现在我们来讨论 整体设计，包括 主要组件 和 它们的功能

库有3个主要组件

1. options description 组件，描述了 允许的option 和 怎么处理这些 option的 value。
2. parsers 组件，用这些信息 来 从 输入中 找到 option名字 和 值 并返回它们
3. storage组件，提供接口 来获得 option 的值。也会 将 parser返回的 string形式的 值 转换为 期望的 c++类型。

更具体一点：options_description 类是 options description 组件，
parse_command_line 函数 来自 parsers 组件， variables_map 类来自 storage 组件

Options Description Component

options description 组件 有 3个主要的 类： option_description, value_semantic, options_description。

前两个一起 描述了一个 单独的option，option_description类包含 option的名字，描述 和 一个 value_semantic 的 指针， value_semantic用于 知道 option的value的类型 和 可以parse值，应用 默认value 等。

options_description 是 option_description 实例的 容器

对于几何每个库来说，这些类 可以方便地创建：你可以 创建新的 option 通过 使用 构造器 然后 调用 options_description 的 add 方法。

但是对于 20个option来说，声明方式 太过冗长。这导致了下面的语法(你之前已经见过)

```
options_description desc;
desc.add_options()
    ("help", "produce help")
    ("optimization", value<int>() -> default_value(10), "optimization level")
;
```

。。这个语法是。。。ok，我知道了，老式的写法是 add + 构造器， 上面的新式写法 就是一行 一个 option。。

调用 value函数 创建了一个 从value_semantic派生的 类的 实例： typed_value。 那个类 包含了 parse 值 到一个 指定类型的 代码，和 一些方法（用户可以调用这些方法 来 指定额外的信息）（这实际上 模拟了 构造器的 具名参数）。

对 add_options 返回的 对象 调用 operator() 会 转发参数 到 options_description 的 构造器 并 增加 新的 instance。

注意，除了 value函数， 库 还提供了 bool_switch 函数，用户 可以写他 自己的 函数， 可以返回 value_semantic 的 其他子类。

在本章中，只会讨论 value 函数。

option 的信息 被分为 语法和语义 (syntactic and semantic)。

语法信息 包含 option 的名字 和 可以用于指定值 的token(记号?)数量。这个信息 被 parser 用来 group token 到 (name, pair)对，这里 value 只是 vector<string>。语义层 负责 转换 optison 的 value 到 C++类型。

这个分离 是 库设计 的一个重要部分。parser 只使用 语法层，这剥夺了 一些 使用复杂 结构的 自由。例如，很难转换 下面的语句

```
calc --expression=1 + 2/3
```

因为，无法转换

```
1 + 2/3
```

因为不知道它是 C 表达式。在用户的帮助下，可以更加清晰

```
calc --expression="1 + 2/3"
```

Syntactic Information

语法信息 通过 boost::program_options::options_description 类 和 boost::program_options::value_semantic 类的一些方法 提供，包括：

1. option 的名字，用来 在程序中 定位 option
2. option 的描述，展现给 用户
3. 解析期间 包含的option 的值的 token(？记号) 的 允许的 数量。

考虑下面的例子

```
options_description desc;
desc.add_options()
    ("help", "produce help message")
    ("compression", value<string>(), "compression level")
    ("verbose", value<string>()>implicit_value("0"), "verbosity level")
    ("email", value<string>()>multitoken(), "email to send to")
;
```

第一个参数，我们只定义了 名字和 描述。不能在 被parse的源 中指定 值。

对于第一(？二)个option，用户必须 指定一个值，使用 一个 token。

对于第三个option，用户 可以提供一个 token 来提供value，或 不需要提供 token。

对于最后一个option，值可以 有多个token。例如，下面的命令行是 ok 的

```
test --help --compression 10 --verbose --email beadle@mars beadle2@mars
```

Description formatting

有时，描述 很长，例如，当 多个 option 的值 需要 文档时。

下面 我们介绍一些可以使用的 简单格式化机制。

```
... . . .
```

Semantic Information

语义信息 完全由 boost::program_options::value_semantic 类 提供：

```
options_description desc;
desc.add_options()
    ("compression", value<int>()>default_value(10), "compression level")
    ("email", value<vector<string>>()
        ->composing()>notifier(&your_function), "email")
```

;

这些声明 定义了 第一个option的默认值是 10，第二个option可以 出现多次，且 所有的实例 应该被 合并。 在 parse 完成后，库会调用 &your_function， 传递 "email" 这个 option 的值 作为参数。

Positional Options

我们将 option 定义为 (名字, 值)，这是简单 且 有用的，但是在命令行的 一个特殊情况下，会有问题。 命令行 可以包含 positional option，它不会指定 任何名字：

```
archiver --compression=9 /etc/passwd
```

这里，"/etc/passwd" 元素 没有 option 名字

一个解决方案是 要求用户自己提取 positional option，并根据 自己的喜好 进行处理。但是 有一种更好的 方法： 提供一个 方法 来自动 为 positional option 提供名字，这样使得 上面的 命令行 可以 下面的一样 被解释

```
archiver --compression=9 --input-file=/etc/passwd
```

positional_options_description 类 允许 命令行parser 来 赋值名字。 类指定了 允许多少 positional option，对于每个 允许的 option，指定名字：

```
positional_options_description pd; pd.add("input-file", 1);
```

指定了 只能一个，且是第一个positional option 的 positional option 的名字 是 input-file

可以指定 数量，甚至 所有的 positional option，被赋值 相同的 名字

```
positional_options_description pd;
pd.add("output-file", 2).add("input-file", -1);
```

上面的例子中， 前2个 positional option 会被 关联到 output-file， 所有其他 会被关联到 input-file

Parsers Component

parsers组件 将 输入 切分成 (名字, 值) 对。

每个parser 都会查找 可能的 选项，并 查询 option descrition组件 来决定 option是否已知，它的值怎么指定。

在最简单的情况下，名字被显式指定，这会允许 库 来决定 这样的 option 是否已知。如果 已知，value_semantic 实例 决定 值如何被指定。(如果不是已知，则抛出异常)

常见的情况是，用户指定 值，或 存在这个option就意味着一些事情。所以 parser 检查 值在需要时被指定 并且 在不需要时不被指定，然后返回 新的 (名字, 值) 对。

要调用parser，通常要 调用一个 函数，传递 option描述，命令行 和 配置文件等。 解析结果 作为 parsed_option 类 的实例 返回。 通常，该对象直接传递给 存储组件，然而，它也可以直接使用，或 进行一些 额外的 处理。

上述的模型 有3个 例外，都和 命令行的 传统用法有关。 虽然它们需要 options

`description` 组件的一些额外支持，但是这种额外的复杂性是可以忍受的。

1. 命令行中指定的名字可能和 option名字不同：通常会对长的名字提供一个短的option名字作为别名。允许在命令行上指定缩写也是常见的。
2. 有时需要将值指定为几个token，例如`--email-recipient`选项可以后面有多个邮箱，每个都是独立的命令行token。这个行为是被支持的，尽管它可能会导致解析歧义，并且默认情况下不会启用。
3. 命令行可以包含positional options。命令行parser提供了一个机制来猜测这种option的名字。

Storage Component

存储组件负责

1. 保存option的最终的值到一个特殊的类和正则变量中
2. 处理不同source的优先级
3. 使用final值调用用于指定的notify函数

考虑一个例子

```
variables_map vm;
store(parse_command_line(argc, argv, desc), vm);
store(parse_config_file("example.cfg", desc), vm);
notify(vm);
```

`variables_map` 被用来存储option值。2次调用`store`函数来增加从命令行和配置文件中找到的值。最后调用`notify`函数执行用户定义的`notify`函数，并存储值到正则变量中，如果需要的话。

优先级的处理很简单：`store`不会改变一个已经被赋值的option的值。在上面的例子中，如果命令行指定了option的值，那么配置文件中的值会被忽略。

警告：在你存储完所有转换后的值后，不要忘记调用`notify`函数

Specific parsers

Configuration file parser

Environment variables parser

Configuration file parser

`parse_config_file`函数实现了对类似INI的配置文件的parse。配置文件的语法：

`name=value`

为option指定一个value

`[section name]`

在配置文件中引入一个新的section

#字符引入一行注释

option名字和section名字有关，如下面的配置：

`[gui.accessibility]`

`visual_bell=yes`

等价于

```
gui.accessibility.visual_bell=yes
```

。。java 中 properties文件有这种 section name 吗？

Environment variables parser

环境变量 是 通过 C运行时库 的 getenv 函数 获得 string 值。

操作系统 允许 为 每个 user 设置 初始值， 这些值 可以在 命令行中 被修改。例如，在 windows中，可以使用 autoexec.bat 文件 或 (在最新版本中) Control Panel/System/Advanced/Environment Variables。

在unix中， /etc/profile, ~/.profile, ~/.bash_profile 文件。

由于 环境变量 是对 整个OS起效的，所以 特别适合于那些 作用于 所有 程序的 option。

环境变量 可以通过 parse_environment 函数 来 parse。这个函数有 多个重载版本。

第一个参数 始终都是 options_description 实例，

第二个参数 指定了 哪些环境变量 必须被处理，以及 这些环境变量 对应了 什么 option 名字。要描述 第二个参数，我们需要 考虑 环境变量 的 命名规则。

如果你需要 通过 环境变量 指定 option，你需要 编写 变量的名字。为了避免 名字冲突，我们建议 你 为环境变量 使用一个 足够独特的前缀。

并且，虽然 option名字 通常是小写，但是 环境变量 通常是 大写。因此，对于 一个名字为 proxy 的 option， 环境变量 可以 是 BOOST_PROXY。

在parse期间，我们需要 执行 名字的 转换。这是通过 将前缀 作为 第二个参数 传入 给 parse_environment 函数 来实现的。比如说， 你传递 BOOST_ 作为前缀，对于 2个 环境变量 CVSROOT 和 BOOST_PROXY， 第一个参数 会被 忽略，第二个 会被 转换为 proxy 选项。

上面的逻辑 在许多情况下 已经足够了， 但也可以 将 一个 (接受string，返回string的) 函数 作为 parse_environment 的 第二个参数。在处理 每个 环境变量时，会调用 这个函数，这个函数 要么 返回 option的名字，要么返回 "" 如果这个变量 应该被 忽略。这个函数的一个例子 可以在 example/env_options.cpp 中看到

类型

在命令行，环境变量，配置文件 中 传递的所有东西 都是 string。对于类型是 非string 的 值，variables_map 中的值 将会尝试 转为 正确的类型。

整型 和 浮点 变量 通过 Boost的 lexical_cast 进行转换。但是不支持 非 10进制。因此无法使用 program_options

有多种方法 可以实现 bool：

(“my-option”， value<bool>()) 类似于其他的option
example --my-option=true

(“other-option”， bool_switch()) 是否存在，存在就是 true
example --other-switch

true, yes, on, 1 是 真， false, no, off, 0 是假。

当从 配置文件读取是，带等号的名字 后没有value 的话 认为 是 true

Annotated List of Symbols

下面的表格 描述了 重要的 符号

Symbol	Description
Options description component	
options_description	describes a number of options
value	defines the option's value
Parsers component	
parse_command_line	parses command line (simplified interface)
basic_command_line_parser	parses command line (extended interface)
parse_config_file	parses config file
parse_environment	parses environment
Storage component	
variables_map	storage for option values

How To

- Non-conventional Syntax
- Response Files
- Winmain Command Line
- Option Groups and Hidden Options
- Custom Validators
- Unicode Support
- Allowing Unknown Options
- Testing Option Presence

Design Discussion

- Unicode Support

=====

=====

=====

https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/index.html

Chapter 1. Coroutine2

Boost.Coroutine2 提供了 模板 来 提供 协程(subroutine)， 允许 挂起 和 恢复 执行。它保存了 执行的 本地状态， 允许 多次 重入协程。

。。。上古时期的计算机科学家们早就给出了概念, coroutine就是可以中断并恢复执行的 subroutine

Coroutines 能被视为 语言级别的 概念， 提供了 特殊的控制流。

和 thread 相比， 协程switch 是 协作的(程序员控制 何时 发生 switch)。
kernel 不参与 switch

本实现 使用了 Boost.Context 作为 上下文切换。

有一个 master library header 来导入所需的 class, function

```
#include <boost/coroutine2/all.hpp>
```

所有的 function, class 都包含于 命名空间: boost::coroutines2

库需要 C++11

Windows using `fcontext_t`: turn off global program optimization (/GL) and change /EHsc (compiler assumes that functions declared as extern "C" never throw a C++ exception) to /EHs (tells compiler assumes that functions declared as extern "C" may throw an exception).

Introduction

计算机科学中， routine 被定义为 一系列操作。

routine 的执行 形成 父-子 关系，并且 子routine 总在 父routine 之前 结束。

Coroutine 是 routine 的 generalization (概括，一般化)

coroutine 和 coutine 的 主要区别是: coroutine 通过保持 执行状态， 通过 附加操作 实现 其进程 的 显式 暂停和恢复，从而提供 增强的 控制流 (维护 执行上下文)

如果 coroutine 像 routine 一样被 调用， stack 会 保存 每个 call， 不会 pop。 jump into coroutine 的middle 是不可能的，因为 return 地址 是在 stack 的 顶部。

解决方案是，让每个 coroutine 有它自己的 stack 和 control-back (Boost.Context 的 fcontext_t)。

在 coroutine 被挂起前， 当前激活 的 coroutine 的 non-volatile 寄存器 (包括 栈 和 程序指针) 被 保存在 coroutine 的 control-back 中。

在 coroutine 恢复前， 它的 control-back 中的 寄存器 必须被 恢复。

。。非易失性寄存器(Non-volatile register)

上下文切换 不需要 系统特权，并且 提供了 C++ 的 方便的 协作多任务处理。

coroutine 提供了 准并行性。

当一个程序应该 同时 做几件事时， coroutine 可以比 只使用 一个 控制流 更简单，更 优雅地 完成。

使用 递归的函数 能特别清晰地看到 优点。

coroutine 的特点是

1. 2次连续的 call 之间 local data 是 保持不变的。
2. 当 控制 离开 coroutine 时 执行被挂起，并在 稍后某个时间 恢复。
3. 对称 或 非对称 控制传递 机制，见下文
4. first-class object (可以作为参数 传递， 由过程返回， 存储在 数据结构中 供 以后使用 或 由开发人员自由操作)
5. stack-ful 或 stack-less

coroutine 在 模拟，AI， 并发编程，文本处理，数据处理 中 非常有用，支持 实现： cooperative task, iterator, generator, infinite list, pipe

execution-transfer mechanism

coroutine 存在 2个类别： 对称 和 非对称 coroutine

非对称coroutine 知道 它的invoker， 使用一个 特殊的操作 来 隐式 yield 控制权 到 它的invoker。

相反，所有 对称coroutine 是 等价的， 一个 对称coroutine 可以传递 控制权 到 任何 其他的 对称coroutine。

因此，对称coroutine 必须 定义 指定 控制权 转移给谁。

通用的coroutine 库 提供 对称 或 非对称 coroutine

stackfulness

和 stackless coroutine 相反， stackful coroutine 可以从 嵌套的栈帧 中 挂起。

使用stackless coroutine，只有 顶层的 routine 可以被 挂起。

被顶层 routine 调用的 任何 routine 本身不能挂起。

这静止在 通用库的 routine 中提供 挂起/恢复 操作。

first-class continuation

第一类成员 可以作为 参数 传递， 被函数返回， 保存到 数据结构中(稍后使用)。
在一些实现中(C# 的 yield)， 不能直接访问 或 直接操作。

没有 stackful 和 first-class语义， 一些有用的 执行控制flow 就不能 被支持 (例如，
协作多任务 和 检查点)

Motivation (动机)

为了支持广泛的执行控制行为， coroutine<> 的协程类型 能被用来 escape-and-reenter loop， 还可以 用来 escape-and-reenter 递归计算， 用来 协助多任务处理， 这样可以 比单控制流 以 更简单，更优雅的方式 解决问题。

event-driven model

事件驱动模型 是一个 编程范式(paradigm)， 程序的 flow 被event 决定。

event 由多个独立的source 生成。

event-dispatcher 等待 外部source， 当有 任意event达到时， 触发 callback 函数
(event-handler)

APP 被划分为 event selection(detection) 和 event handling。

最终的APP 有高 scalable, flexible(可扩展性， 灵活性)， 高响应性 和 松耦合。

这使得 event-driven model 适用于 用户界面程序， 基于规则的生产者系统， 处理异步IO
的应用(如 网络服务器)

event-based asynchronous paradigm

经典的 sync 控制台程序 发出IO请求 (如， 等待用户输入 或 文件数据)， 然后阻塞 直到
请求完成。

相反， 一个 异步 IO 方法， 启动物理操作 后 立即返回给 caller， 即使 操作没有完成。
使用这种功能编写的 程序 不会 block： 它可以在 原始操作 依然挂起时 执行其他工作(包括并行的其他IO请求)。

当操作完成， 程序被 通知。由于 异步应用 等待操作 的总时间较少， 所以 它们的性能优
于 同步程序。

event 是 异步处理的 一种范式， 但不是 所有的 异步系统 都使用 event。

虽然 异步编程 可以 通过 thread 来完成， 但是 它们有它们自己的cost

1. 编程实现难
2. 内存要求高
3. 创建和 维护 线程状态的 开销很大
4. 线程间 切换的 代价大。

基于事件的 异步模型 避免了 这些问题

1. 更简单， 因为， 只有 一个 指令流
2. 上下文切换 更 便宜。

这种模式的 缺点是 程序结构并非最优。

事件驱动程序 需要 将 代码拆分为 多个 小的回调函数，即代码 按照 一系列 间歇 执行的 小步骤 组织。

平常 使用 函数和循环 的 层次结构 必须转换为 回调。

当控制流 返回到 event-loop 时，必须将 完整状态 存储到 数据结构中。

因此，事件驱动 的 APP 通常 编写起来 乏味 又 令人困惑。

每个回调 都会引入 一个新的 范文，错误回调等。

算法的 顺序性质 被划分为 多个 调用堆栈，使得 APP 很难调试。

异常处理程序 被局限于本地 处理程序：不可能 将 一系列事件 打包到 一个 try-catch 中。

不能将局部变量，while/for循环，递归等 与 event-loop 一起使用。

代表的 表达能力 不够。

。 。 很多代码。 。

COROUTINE

Boost.Coroutine2 提供 非对称 coroutine

生成 值序列的 实现 通常使用 非对称 协程。

stackful

协程的 每个实例 都有它自己的 stack

和 stackless 协程 对比， stackful 协程 允许 从 任意子 堆栈帧 调用 suspend 操作， 从而 实现 escape-and-reenter 递归操作。

move-only

协程 是 moveable-only (只能移动)

如果 它是 可复制的，那么 它的 stack 中的 所有 对象 也会被 复制。这会 导致 未定义 行为，如果 一些对象 是 RAIU 类。 第二个完成的对象 会再次 释放资源，会导致 未定义 行为。

clean-up

在协程 析构时，关联的 stack 会被 展开(unwound)。

协程的 构造器 允许你 传递 自定义的 stack-allocator。栈分配器 可以自由释放stack 或 cache stack (用于 稍后创建的 协程)。

segmented stack

coroutine<>::push_type 和 coroutine<>::pull_type 支持 分段stack

无法准确 估计 所需的 stack size， 大多数情况下，申请了 太多 的内存 (造成 虚拟地址空间的 浪费)

在构造时，协程 以默认(最小) stack size 开始。 最小stack size 是 max(page size, 信号堆栈的大小)

析构函数 释放了 关联的 stack。 实现者 可以自由滴 选择 释放stack 或 cache 它以备 后续使用。

context switch

协程根据 每个 context switch 上的 底层 ABI (使用 Boost.context) 保存和恢复 寄存器。

context switch 通过 `coroutine<>::push_type::operator()` 和 `coroutine<>::pull_type::operator()` 来完成。

在同一个 协程中 调用 `push_type`的`operator()` 和 `pull_type`的`operator()` 会导致 未定义行为。

Asymmetric coroutine

```
Class coroutine<>::pull_type  
Class coroutine<>::push_type
```

2个 非对称 协程类型 `coroutine<>::push_type` 和 `coroutine<>::pull_type`, 提供了 数据的 单向传输。

注意: `asymmetric_coroutine<>` is a `typedef` of `coroutine`.

`coroutine<>::pull_type`

从另一个 执行上下文 传输数据。

模板参数 定义了 传输的参数类型。

`coroutine<>::pull_type` 的 构造器 接受一个函数 (协程函数), 这个函数 接受 一个 `coroutine<>::push_type` 的 `ref` 作为参数。

实例化 一个 `coroutine<>::pull_type` 将 传递 执行的控制权 给 协程函数, 库 会 生成 互补的 `coroutine<>::push_type` , 并作为 `ref` 传递给 协程函数。

这种协程 提供了 `coroutine<>::pull_type::operator()`, 这个方法 只能 切换上下文, 不 传递数据。

`coroutine<>::pull_type` 提供了 `input iterator`

(`coroutine<>::pull_type::iterator`) 和 `std::begin() | std::end()` 的重载。
`increment-operation` 切换 上下文 和 传输数据。

```
typedef boost::coroutines2::coroutine<int> coro_t;
```

```
coro_t::pull_type source(  
    [&] (coro_t::push_type& sink) {  
        int first=1, second=1;  
        sink(first);  
        sink(second);  
        for (int i=0; i<8; ++i) {  
            int third=first+second;  
            first=second;  
            second=third;  
            sink(third);  
        }  
    });
```

```
for (auto i:source)  
    std::cout << i << " ";
```

output:

1 1 2 3 5 8 13 21 34 55

这个例子中，在主执行上下文中 创建了一个 `coroutine<>::pull_type`，接收一个 `lambda`（等于 协程函数），在 `for` 循环中 计算 Fibonacci。

协程函数 在 新创建的 被`coroutine<>::pull_type`的实例 管理的 执行上下文中 执行。`coroutine<>::push_type` 被 库 自动生成，并传递 一个 `ref` 到 `lambda`中。

每次 `lambda` 用 另一个 Fibonacci数字 调用 `coroutine<>::push_type::operator()`，`coroutine<>::push_type` 传递它 返回 到 主 执行上下文。

协程函数的 `local`状态 被保留，并在 将 执行控制 转移回 协程函数(以计算下一个 Fibonacci)时 被恢复

因为 `coroutine<>::pull_type` 提供了 `input iterator` 和 `std::begin() | end()` 的重载，基于 范围的 `for`循环 可以被用于 迭代 生成的 Fibonacci数。

`coroutine<>::push_type`

传输数据到 其他 执行上下文。

模板参数定义了 传输的 参数类型。

`coroutine<>::push_type` 的构造器 接受一个 函数（协程函数），它接受 `coroutine<>::pull_type` 的 `ref` 作为参数。

和 `pull_type` 不同， 实例化一个 `push_type` 不会 传递 执行控制权 给 协程函数， 而是 第一次调用 `push_type::operator()` 时 生成 互补的 `coroutine<>::pull_type`，并把它 作为 `ref` 传递给 协程函数。

`coroutine<>::push_type` 接口不包含 `get()`函数： 不能使用这种 协程 从 另一个 执行上下文 中 检索 值。

`coroutine<>::push_type` 提供了 输出迭代器 (`coroutine<>::push_type::iterator`) 和 `std::begin() | end()` 的重载。

`increment-operation` 切换上下文 和 传递数据

```
typedef boost::coroutines2::coroutine<std::string> coro_t;

struct FinalEOL{
    ~FinalEOL() {
        std::cout << std::endl;
    }
};

const int num=5, width=15;
coro_t::push_type writer(
    [&] (coro_t::pull_type& in) {
        // finish the last line when we leave by whatever means
        FinalEOL eol;
        // pull values from upstream, lay them out 'num' to a line
        for (;;) {
            for(int i=0;i<num;++i) {
                // when we exhaust the input, stop
                if(!in) return;
                std::cout << std::setw(width) << in.get();
                // now that we've handled this item, advance to next
            }
        }
    }
);
```

```

        in();
    }
    // after 'num' items, line break
    std::cout << std::endl;
}
);

std::vector<std::string> words{
    "peas", "porridge", "hot", "peas",
    "porridge", "cold", "peas", "porridge",
    "in", "the", "pot", "nine",
    "days", "old" };

std::copy(begin(words), end(words), begin(writer));

```

output:

peas	porridge	hot	peas	porridge
cold	peas	porridge	in	the
pot	nine	days	old	

在上面的例子中，主执行上下文中 创建 `coroutine<>::push_type`，接受一个 `lambda`（等于 协程函数），接受 `strings` 并在 每行上 显示 `'num'` 个 `string`。

这说明了 协程 允许的 控制反转。如果 没有协程，执行相同任务的 `util`函数 必须接受 每个 新值 作为 函数参数，并在 处理 这个 单值后 返回。

该函数 取决于 静态状态变量。

但是，协程函数 可以像 普通函数 那样 请求 每个新值 -- 即使 `caller` 也像 调用函数 那样 传递值。

协程函数 在 新创建的 受 `push_type` 实例 管理的 执行上下文 中 执行。

主执行上下文 传递 `string` 到 协程函数 通过 调用 `push_type::operator()`。

库 自动生成一个 `pull_type` 实例，并 作为 `ref` 传递给 `lambda`。

协程函数 使用 `pull_type::get()` 访问 从 主执行上下文 传递过来的 `strings`，并且根据 参数 `num` 和 `width` 输出 `string`到 `std::cout`。

协程函数的 `local state` 被保存，并且 在 将 执行控制 返回给 协程函数 后 恢复。

由于 `push_type` 提供了 输出迭代器 和 `std::begin()|end()` 的重载，`std::copy` 算法 可以用来 迭代 包含 `string`的 `vector`，且 按个传递 给 协程。

coroutine-function

协程函数 返回 `void`，并且 接受 它的 `counterpart-coroutine` 作为参数，所以 作为参数 传递给 协程函数 的 协程 是 唯一的 方式 来 传输数据 和 执行控制权 返回给 `caller`。

2 个 `coroutine type` 都接受 相同的 模板参数。

对于 `coroutine<>::pull_type`，协程函数 在 `pull_type` 构造时 传入。

对于 `push_type`，协程函数 不是在 `push_type`构造时传入， 而是在 第一次调用 `coroutine<>::push_type::operator()` 时 传入。

在 执行控制权 从 协程函数 中返回后，协程的 状态可以通过

`coroutine<>::pull_type::operator bool` 来检查，如果 返回 `true`，`coroutine` 是 `valid` (协程函数 没有被 `terminate`)。 除非 第一个模板参数 是 `void`，`true` 也意味着 `data value` 是 `available`。

passing data from a pull-coroutine to main-context

为了从 pull_type 传输 数据 到 main-context，框架在 main-context 中 合成了 (synthesizes) 一个 关联到 pull_type 实例的 push_type。

合成的 push_type 被 作为参数 传递到 coroutine-function。

coroutine-function 必须调用push_type::operator() 以便 将数据 传回 main-context。在 main-context, coroutine<>::pull_type::operator bool 决定了 是否 coroutine 是 valid，数据是否available，还是说 coroutine-function 被terminated了（此时 pull_type 是 invalid，没有数据可获得）。

获得 传输的 数据，通过 coroutine<>::pull_type::get()

```
typedef boost::coroutines2::coroutine<int> coro_t;

coro_t::pull_type source( // constructor enters coroutine-function
    [&](coro_t::push_type& sink) {
        sink(1); // push {1} back to main-context
        sink(1); // push {1} back to main-context
        sink(2); // push {2} back to main-context
        sink(3); // push {3} back to main-context
        sink(5); // push {5} back to main-context
        sink(8); // push {8} back to main-context
    });

while(source){           // test if pull-coroutine is valid
    int ret=source.get(); // access data value
    source();            // context-switch to coroutine-function
}
```

passing data from main-context to a push-coroutine

为了从 main-context 传输数据 到 coroutine<>::push_type，框架 在 main-context 中 合成了一个 关联到 push_type实例 的 pull_type。

合成的 pull_type 被作为参数 传递给 coroutine-function。

main-context 必须调用 push_type::operator() 以便 将 数据传递到 协程函数。

获得传输的数据，通过 pull_type::get()

```
typedef boost::coroutines2::coroutine<int> coro_t;

coro_t::push_type sink( // constructor does NOT enter coroutine-function
    [&](coro_t::pull_type& source) {
        for (int i:source) {
            std::cout << i << " ";
        }
    });

std::vector<int> v{1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
for( int i:v){
    sink(i); // push {i} to coroutine-function
```

```
}
```

accessing parameters

可以通过 `coroutine<>::pull_type::get()` 来获得 从 协程函数 返回，或 发送到协程函数的 参数。

区分 来自 context switch function 的 参数的 访问 可以用来：检查 在 从 `pull_type::operator()` 返回后 `pull_type` 是否 valid (有value 且 协程函数没有被 终止)。

```
typedef boost::coroutines2::coroutine<boost::tuple<int, int>> coro_t;

coro_t::push_type sink(
    [&](coro_t::pull_type& source) {
        // access tuple {7, 11}; x==7 y==1
        int x, y;
        boost::tie(x, y)=source.get();
    });
sink(boost::make_tuple(7, 11));
```

exceptions

在 首次调用 `coroutine<>::push_type::operator()` 之前，`pull_type` 的 协程函数 抛出的 异常，会被 `pull_type` 的构造器 重新抛出。

在 `pull_type`的 协程函数 第一次调用 `push_type::operator()` 后，所有 协程函数中的 异常 会被 `pull_type::operator()` 重新抛出。

`pull_type::get()` 不会抛出异常。

重要：协程函数 执行的 代码 不能阻止 `detail::forced_unwind` 异常的 传播。吸收这个 异常 将导致 栈展开失败 。 因此，所有 `catch` 了 所有异常的 代码 必须 重新 抛出 任何 pending 的 `detail::forced_unwind` 异常

```
try {
    // code that might throw
} catch(const boost::coroutines2::detail::forced_unwind&) {
    throw;
} catch(...) {
    // possibly not re-throw pending exception
}
```

重要：不要从 一个 `catch` 块 中跳出， 然后在 另一个 执行上下文 中 重新抛出异常。

Stack unwinding

有时，需要 `unwind`(展开) 一个 未完成 的 协程的 stack 来 销毁local stack 变量，以便 可以释放 资源 (RAII 模式)。

协程构造器 的 `attributes` 参数 用于 指明 析构器 是否应该 `unwind` stack (默认不会展开)。

stack unwinding 假设以下 前提条件

1. coroutine 不是 not-a-coroutine
2. coroutine 没有 complete
3. coroutine 不在running
4. coroutine 拥有一个栈

在展开后，coroutine 是完成的。

```
struct X {  
    X() {  
        std::cout<<"X()"<<std::endl;  
    }  
  
    ~X() {  
        std::cout<<"~X()"<<std::endl;  
    }  
};  
  
{  
    typedef boost::coroutines2::coroutine<void>::push_type coro_t;  
  
    coro_t::push_type sink(  
        [&] (coro_t::pull_type& source) {  
            X x;  
            for(int=0;;++i){  
                std::cout<<"fn(): "<<i<<std::endl;  
                // transfer execution control back to main()  
                source();  
            }  
        }  
    );  
  
    sink();  
    sink();  
    sink();  
    sink();  
    sink();  
  
    std::cout<<"sink is complete: "<<std::boolalpha<<!sink<<"\n";  
}  
  
output:  
X()  
fn(): 0  
fn(): 1  
fn(): 2  
fn(): 3  
fn(): 4  
fn(): 5
```

```
sink is complete: false
~X()
```

Range iterators

Boost.Coroutine2 提供了 output 和 input iterator，通过使用 `_boost_range_.coroutine<>::pull_type` 也可以使用 `std::begin()``end()` 作为 input-iterator

```
typedef boost::coroutines2::coroutine< int > coro_t;
```

```
int number=2, exponent=8;
coro_t::pull_type source(
    [&] (coro_t::push_type & sink) {
        int counter=0, result=1;
        while(counter++<exponent) {
            result=result*number;
            sink(result);
        }
    });
});
```

```
for (auto i:source)
    std::cout << i << " ";
```

output:

```
2 4 8 16 32 64 128 256
```

`coroutine<>::pull_type::iterator::operator++()` 相当于 `pull_type::operator()`。
`pull_type::iterator::operator*()` 大致相当于 `pull_type::get()`。

一个源于 `pull_type` 的 `std::begin()` 的 iterator 等于这个 `pull_type` 实例的
`std::end()`，当它的 `pull_type::operator bool` 返回 `false` 时。

注意

如果 T 是一个 move-only 类型，那么 `coroutine<T>::pull_type::iterator` 在递增前 只能 反引用一次。

output-iterator 能从 `coroutine<>::push_type` 中 创建。

```
typedef boost::coroutines2::coroutine<int> coro_t;
```

```
coro_t::push_type sink(
    [&] (coro_t::pull_type& source) {
        while(source) {
            std::cout << source.get() << " ";
            source();
        }
    });
});
```

```
std::vector<int> v{1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
std::copy(begin(v), end(v), begin(sink));
```

`push_type::iterator::operator*()` 大致相当于 `push_type::operator()`。
当 `push_type::operator bool` 返回 `false` 时，来自 `push_type` 的 `std::begin()` 的 iterator 等于这个实例的 `std::end()`。

Exit a coroutine-function

协程函数 通过 简单的 `return` 语句 退出 到 calling routine。

`pull_type, push_type` 变成 `complete`，即 `pull_type::operator bool`,
`push_type::operator bool` 会 返回 `false`

重要：在从 协程函数中 `return` 后，协程变成完成（不能使用 `push_type::operator()`,
`pull_type::operator()`）

https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/coroutine2/coroutine/asymmetric/pull_coro.html

```
#include <boost/coroutine2/coroutine.hpp>
```

```
template< typename R >
class coroutine<>::pull_type
{
public:
    template< typename Fn >
    pull_type( Fn && fn);

    template< typename StackAllocator, typename Fn >
    pull_type( StackAllocator stack_alloc, Fn && fn);

    pull_type( pull_type const& other)=delete;

    pull_type & operator=( pull_type const& other)=delete;

    ~pull_type();

    pull_type( pull_type && other) noexcept;

    pull_type & operator=( pull_type && other) noexcept;

    pull_coroutine & operator()();

    explicit operator bool() const noexcept;

    bool operator!() const noexcept;

    R get() noexcept;
};
```

```
template< typename R >
range_iterator< pull_type< R > >::type begin( pull_type< R > &);
```

```
template< typename R >
range_iterator< pull_type< R > >::type end( pull_type< R > &);
```

https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/coroutine2/coroutine/asymmetric/push_coro.html
#include <boost/coroutine2/coroutine.hpp>

```
template< typename Arg >
class coroutine<>::push_type
{
public:
```

```
    template< typename Fn >
    push_type( Fn && fn);
```

```
    template< typename StackAllocator, typename Fn >
    push_type( StackAllocator stack_alloc, Fn && fn);
```

```
    push_type( push_type const& other)=delete;
```

```
    push_type & operator=( push_type const& other)=delete;
```

```
    ~push_type();
```

```
    push_type( push_type && other) noexcept;
```

```
    push_type & operator=( push_type && other) noexcept;
```

```
    explicit operator bool() const noexcept;
```

```
    bool operator!() const noexcept;
```

```
    push_type & operator()( Arg arg);
```

```
} ;
```

```
template< typename Arg >
range_iterator< push_type< Arg > >::type begin( push_type< Arg > &);
```

```
template< typename Arg >
range_iterator< push_type< Arg > >::type end( push_type< Arg > &);
```

https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/coroutine2/coroutine/implementations_fcontext_t_uco.html

[ucontext_t and winfiber.html](#)

Implementations: fcontext_t, ucontext_t and WinFiber

fcontext_t

默认使用 fcontext_t，是基于 汇编的，不是所有平台通用。提供了 比 ucontext_t 和 WinFiber 更好的 性能。（上下文切换 消耗的 CPU时钟 是 ucontext_t 的 2个数量级 (magnitude) 的 小）。

ucontext_t

使用 BOOST_USE_UCONTEXT 和 context-impl=ucontext 来编译，会使用 ucontext_t，可以在 更广泛的 POSIX 平台上使用。

WinFiber

使用 BOOST_USE_WINFIB 和 context-impl=winfib。

[https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/coroutine2/stack.html](#)

Stack allocation

- Class protected_fixedsize
- Class pooled_fixedsize_stack
- Class fixedsize_stack
- Class segmented_stack
- Class stack_traits
- Class stack_context
- Support for valgrind
- Support for sanitizers

[https://www.boost.org/doc/libs/1_80_0/libs/coroutine2/doc/html/coroutine2/performance.html](#)

Performance

Table 1.1. Performance of context switch

using fcontext_t	using ucontext_t
26 ns / 56 CPU cycles	542 ns / 1146 CPU cycles

=====

[https://www.boost.org/doc/libs/1_80_0/doc/html/boost_asio.html](#)

Boost.Asio

Overview

- Rationale
- Basic Boost.Asio Anatomy
- Asynchronous Model
- Asynchronous Operations

- Asynchronous Agents
- Associated Characteristics and Associators
- Child Agents
- Executors
- Allocators
- Cancellation
- Completion Tokens
- Supporting Library Elements
- Higher Level Abstractions
- Core Concepts and Functionality
 - The Proactor Design Pattern: Concurrency Without Threads
 - Threads and Boost.Aasio
 - Strands: Use Threads Without Explicit Locking
 - Buffers
 - Streams, Short Reads and Short Writes
 - Reactor-Style Operations
 - Line-Based Operations
 - Custom Memory Allocation
 - Per-Operation Cancellation
 - Handler Tracking
 - Concurrency Hints
- Composition and Completion Tokens
 - Stackless Coroutines
 - Stackful Coroutines
 - Futures
 - C++20 Coroutines Support
 - Resumable C++20 Coroutines (experimental)
 - Deferred Operations (experimental)
 - Promises (experimental)
 - Co-ordinating Parallel Operations (experimental)
 - Compositions as Asynchronous Operations
 - Completion Token Adapters
- Networking
 - TCP, UDP and ICMP
 - Support for Other Protocols
 - Socket Iostreams
 - The BSD Socket API and Boost.Aasio
- Timers
- Files
- Pipes
- Serial Ports
- Signal Handling
- Channels (experimental)
- POSIX-Specific Functionality
 - UNIX Domain Sockets
 - Stream-Oriented File Descriptors
 - Fork
- Windows-Specific Functionality
 - Stream-Oriented HANDLES

Random-Access HANDLEs
Object HANDLEs
SSL
C++ 2011 Support
 Movable I/O Objects
 Movable Handlers
 Variadic Templates
 Array Container
 Atomics
 Shared Pointers
 Chrono
Platform-Specific Implementation Notes

Rationale 原理

大部分程序 都需要和外部交互，无论是通过 文件，互联网，串行总线，控制台。
有时，通过 网络时， IO会消耗很长时间。

Boost.Asio 提供了 工具 用来 管理这些 长时间的 操作，而不要求程序使用 基于线程和 显式锁的 并发模型。

Boost.Asio 面向 使用C++ 进行 系统编程的 程序员，通常需要 访问 网络 等 系统功能。
特别是，Asio 实现了以下目标

1. portability, 可移植性，库支持一系列常用的OS，并为这些 OS 提供了一致的行为
2. scalability, 可扩展性，库促进了 网络应用程序的开发，使得它们可以扩展到 数千个 并发连接。每个OS的 库实现 应该使用 最能实现这种 可伸缩性的 机制。
3. efficiency, 高效，库应该支持以下技术：scatter-gather IO，允许 程序 最小化 数据 copying。
4. model concepts from established APIs, such as BSD sockets, BSD socket API得到了广泛的实现和理解，其他编程语言使用类似的 网络API接口。
5. ease of use。库应该采用 工具包 而不是框架 的方法，为新用户提供 较低的入门门槛。也就是说，只需要学习一些 基本规则和准则，之后，库用户 只需要 了解正在使用的 特定功能。
6. basis for further abstraction, 库应该允许 其他库的开发 来提供 更高层的 抽象。例如，经常使用的 协议 (如HTTP)的实现。

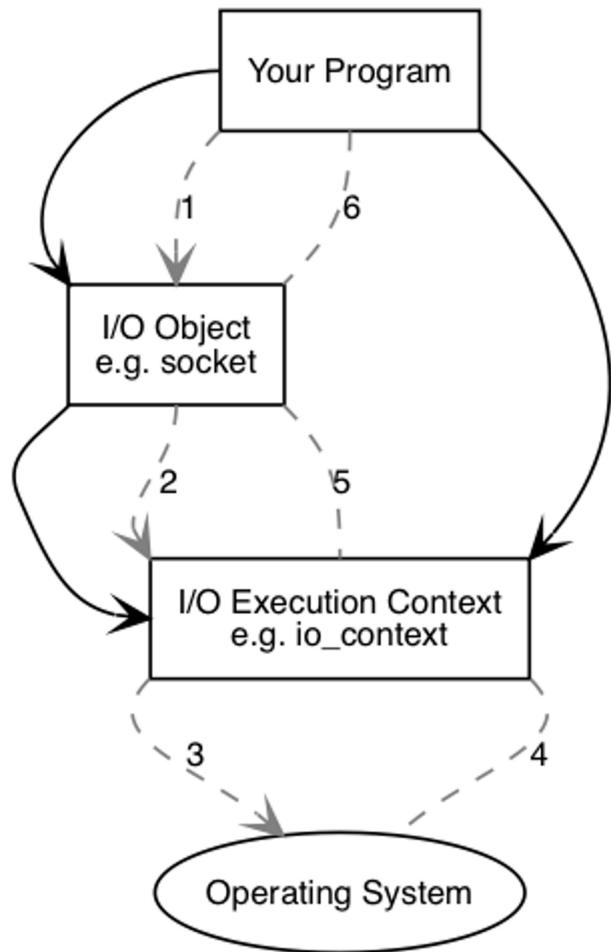
。。。Scatter/Gather I/O, 翻译过来是分散/聚集 I/O (又称为Vectored I/O)。是一种可以单次调用中对多个缓冲区进行输入/输出的方式，可以把多个缓冲区数据一次写到数据流中，也可以把一个数据流读取到多个缓冲区

尽管Asio 一开始主要专注于网络， 其异步IO的概念已经 扩展到 其他的OS资源，如 串行端口，文件描述符 等

Basic Boost.Asio Anatomy (剖析)

Asio 可以用来 执行 在 IO对象(如 socket)上 执行 同步和异步 操作。

根据下图，让我们思考，当你在 socket 上执行 连接操作时，发生了什么。
我们应该从 **同步操作开始**。



你的程序将至少有一个 I/O 执行上下文，比如 `boost::asio::io_context`, `boost::asio::thread_pool`, `boost::asio::system_context` 等。

I/O 执行上下文 代表了 你程序 中 对 OS 的 I/O 服务的 link。

```
boost::asio::io_context io_context;
```

要执行 I/O 操作，你的程序 需要 一个 I/O 对象，比如 TCP socket

```
boost::asio::ip::tcp::socket socket(io_context);
```

。 。 。 下面的序号 就是 图上的 数字

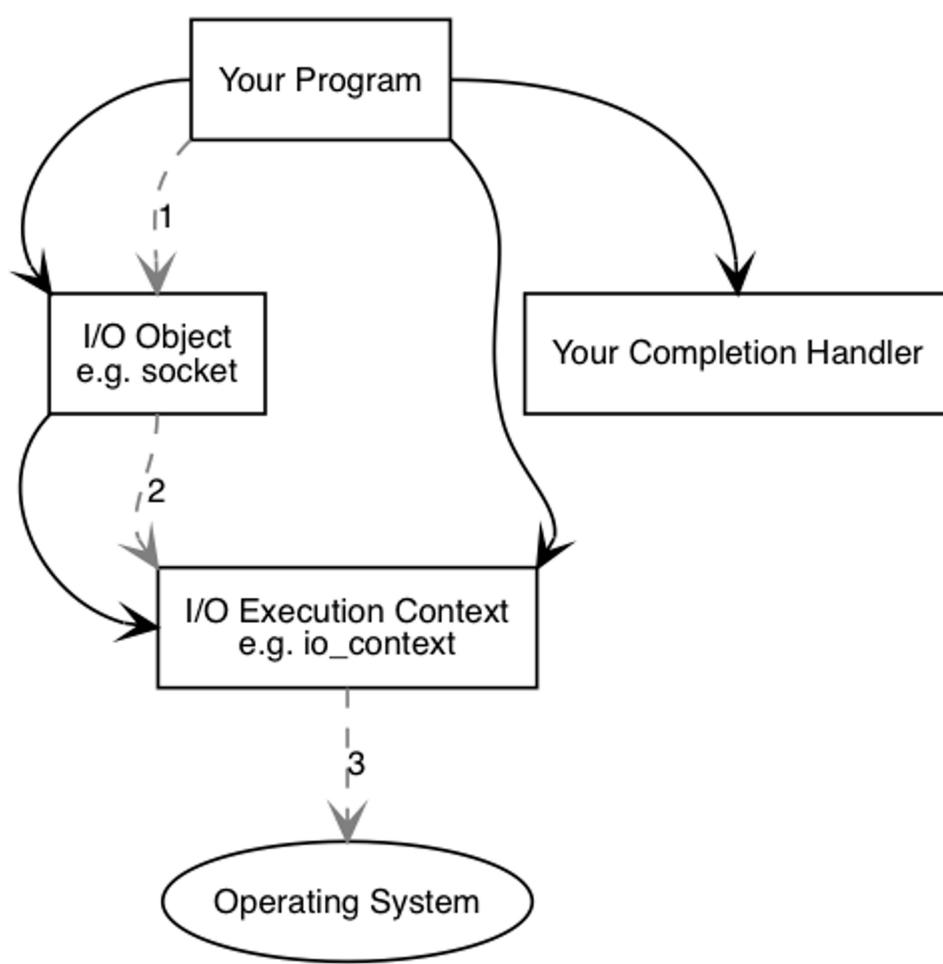
当执行了 同步连接操作 (synchronous connect operation) 后，下面的 事件序列 发生。

1. 你的程序 初始化 connect operation 通过 调用 I/O 对象
`socket.connect(server_endpoint);`
2. I/O 对象 转发 request 到 I/O 执行上下文
3. I/O 执行上下文 调用 OS 来执行 connect 操作
4. OS 返回 操作结果 给 I/O 执行上下文
5. I/O 执行上下文 转换 操作的 任何错误结果 为 一个 `boost::system::error_code` 类型 的对象。 `error_code` 可以和 特殊值 比较 或 作为 boolean 进行测试 (`false` 意味着 没有错误)。 结果 转发回 I/O 对象
6. 如果操作失败，I/O 对象 抛出 `boost::system::system_error` 类型的 异常。如果 初始化操作的 代码 被写为：

```
boost::system::error_code ec;
```

```
socket.connect(server_endpoint, ec);  
那么, ec 会被设置为 操作的结果, 不会抛出异常
```

当使用 异步操作时, 事件发生的序列 并不同。



1. 你的程序 初始化一个 connect operation 通过调用 IO对象。

```
socket.async_connect(server_endpoint, your_completion_handler);
```

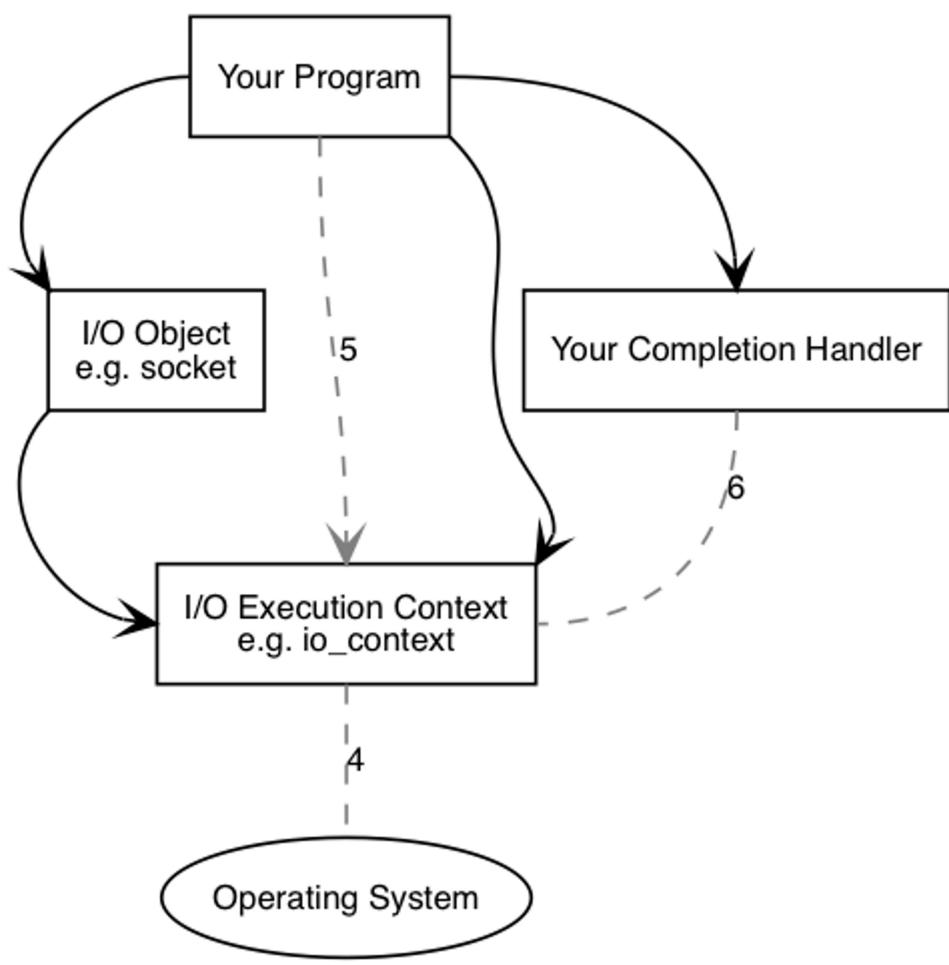
当 your_completion_handler 是 函数 或 带有签名的函数对象 时:

```
void your_completion_handler(const boost::system::error_code& ec);
```

所需的 准确的 签名 取决于 正在执行的 异步操作。

2. IO对象转发request 到 IO执行上下文
3. IO执行上下文 给 OS 发信号, 告诉OS 应该启动一个 异步connect

过段时间 (在 同步流程中, 这个 等待 是包含在 connect operation 中的)



4. OS 通过 将 结果放到 队列中，以告知 connect operation 已完成，并 准备好 被 I/O 执行上下文 获取
5. 当使用 io_context 作为 I/O执行上下文时， 你的程序 必须 调用 io_context::run() (或 其他类似io_context成员函数) 以 获得 结果。对 io_context::run() 的调用 会 block， 当 异步操作 未完成时， 所以通常 在你 第一个 异步操作后 立刻调用它。
6. 在对 io_context::run() 的调用中， I/O执行上下文 将操作的 结果 出队， 转换它为 error_code， 然后 传递给 你的 completion handler。

这是一个简化的 Asio 操作。

如果要 更多 高级特性，你需要 看文档。

Asynchronous Model

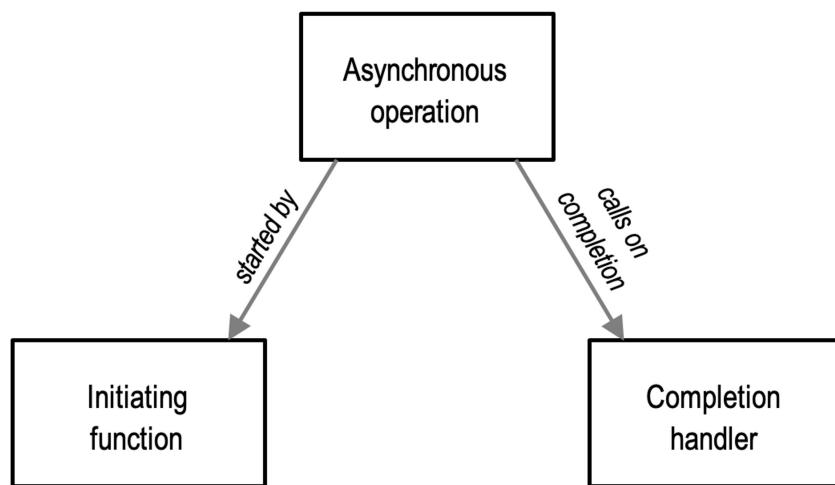
本章描述了 Asio 库的 核心的 异步模型 的 高层概括。

该模型将 异步操作 作为 异步合成 的 基本构成块，但是 方式 是将 它们与 合成机制 分离。

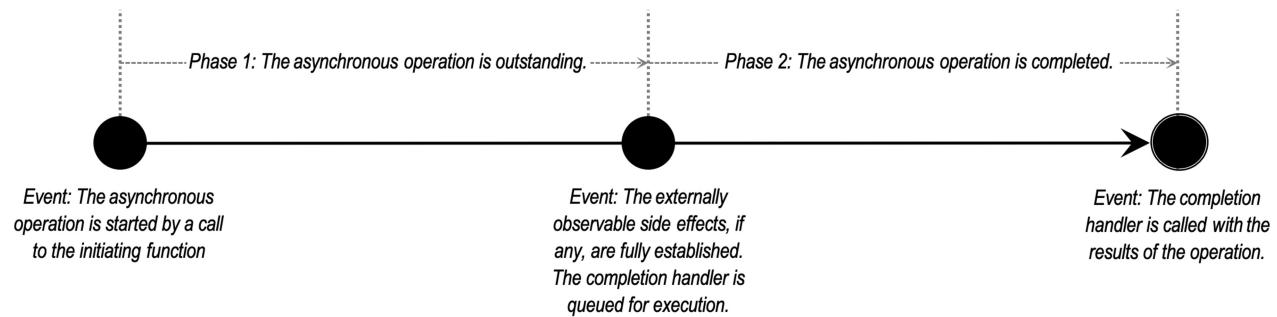
Boost.Asio 中的 异步操作 支持 callback, future (包括 eager 和 lazy)， fiber, coroutine, 等。

这是一个基础单元，用于组成 Boost.Asio 异步模型。

异步操作 代表了 在后台启动 和 执行的工作，而 启动工作的 用户代码 可以继续执行 其他工作。



从概念上讲，异步操作的 生命周期 可以通过 以下 事件 和 阶段 来描述



initiating function 是 用户 用来 启动一个 异步操作的 函数
completion handler 是 用户提供的，move-only function object， 它会被 调用，最多一次，调用时 获得 异步操作的结果。对 completion handler 的调用 告知用户 一些事情已经发生：操作已完成，操作的副作用已经建立。

initiating function 和 completion handler 组成了 用户的代码

synchronous 操作，体现为单个函数，因此有 几个固定的语义属性，
异步操作，采用了 同步操作 中的一些语义属性，以便于 灵活高效地 组合。

同步操作的属性	异步操作中的等价属性
当一个同步操作是 generic (例如, template)，返回参数的类型 由 函数和它的参数 决定	当一个 异步操作是 genric, completion handler 的参数类型 和 命令 由 initiating function 和它的参数 决定。
如果同步操作 需要 临时资源 (如 内存, 文件描述符, 线程)，这个资源会在 function return 前被释放	如果 异步操作 需要 临时资源(内存, 文件描述符, 线程)， 这个资源 会在 调用 completion hander 之前 释放。

后者是 异步操作的一个重要属性，因为它允许 completion handler 再初始化一个 异步操作 而不会 导致 资源使用的 累加。

考虑在 链中 反复执行 相同的操作，通过 确保 在完成处理程序运行之前 释放资源，我们 避免了 将操作链 的 资源使用量 加倍。

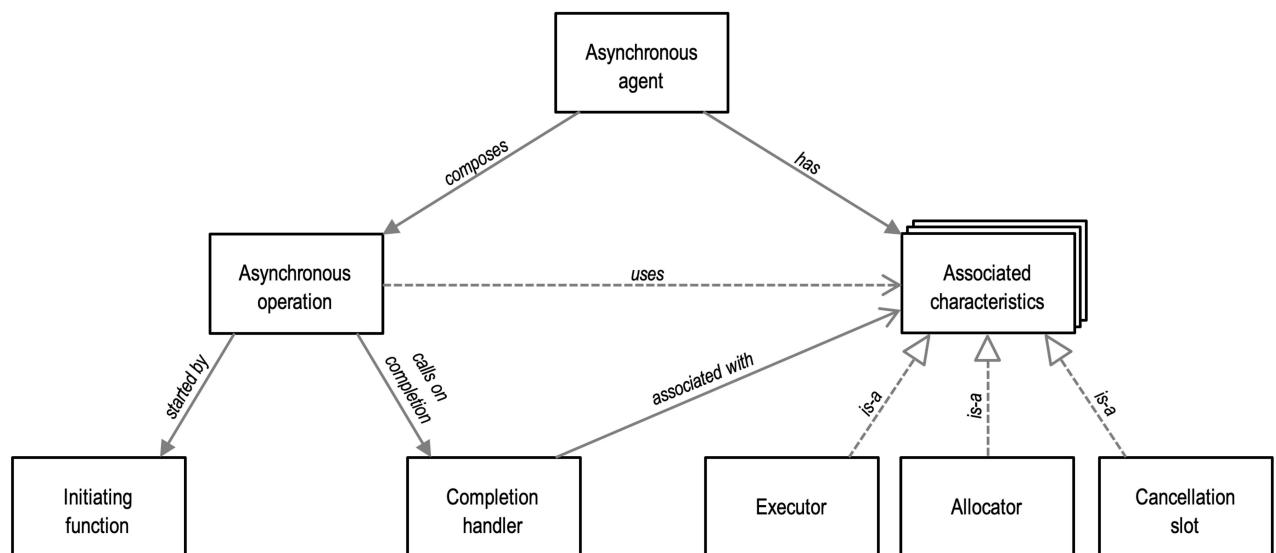
Asynchronous Agents

异步代理 是 异步操作的 顺序组合。

每个异步操作 被视为 异步代理的 一部分，即使 代理 只包含 一个 操作。

异步代理 是一个 实体，可以 和其他 agent 并发工作。

异步代理 对于 异步操作 就如同 线程 对于 同步操作 一样。



但是，异步代理 是一个 纯粹的 概念性 构造， 它允许 我们推理 程序中 异步操作的 上下文 和 组成。

"asynchronous agent" 这个名字 没有在 库 中存在。在代理中 使用 什么机制 来组成 异步操作 也不重要。

异步代理 交替地 等待 异步操作完成，然后 为该操作 运行 completion handler。在 代理的 上下文中， 这些 completion handler 表示 可调度工作的 的 不可分割单元。

Associated Characteristics and Associators 相关特征和关联

异步代理 有 相关特征，用来 指明 当作为agent的 一部分时，异步操作 应该有怎样的行为：

1. 作为 allocator，它决定 代理的 异步操作 如何 获得内存资源
2. 作为 cancellation slot， 它决定 代理的 异步操作 如何 支持 cancellation
3. 作为 executor，它决定 代理的 completion handler 如何 处理 入队 和 run。

Specification of an Associator

给定一个 associator，名为 associated_R，有：

1. 类型S 的 source值 s，在这里代表 completion handler 和它的类型。
2. 类型requirement 的集合 R，定义了相关的特征的句法和语义要求。
3. 满足类型需求R 的类型C 的候选值 c，表示异步操作提供的相关特征的默认实现

异步操作 使用 associator 来计算：

1. 类型： associated_R<S, C>::type
2. 值： associated_R<R, C>::get(s, c)

这些满足 定义在 R 中的要求。

为了方便起见，也可以通过 类型别名associated_R_t<S, C> 和 函数 get_associated_R(s, c) 来访问。

特征的 主模板 被指定为：

1. 如果 S::R_type 格式良好，则将 嵌入的 类型别名 定义为 S::R_type， 和 一个 静态成员函数 返回 s.get_R()。
2. 否则，如果 associator<associated_R, S, C>::type 格式良好，继承自 associator<associated_R, S, C>
3. 否则，定义一个 嵌入的 类型别名 为 C，和 一个 静态成员函数 返回 c

Child Agents

agent中的异步操作 可以是一个 child agent。对于父agent 而言，它只是在等待一个 异步操作的完成。组成子代理的 异步操作 按照顺序执行，当 final completion handler 完成时，父代理 恢复。

和单个异步操作一样，在 子代理上构建的 异步操作 必须在 调用 completion handler 之前 释放 它们的临时资源。我们也可以 将这些 子代理 视为一种 在completion handler 被调用之前 结束生命周期的 资源，

当异步操作创建子代理时，它可以向 子代理 传播 父代理的 相关特性。然后，可以通过 异步操作 和 子代理的 其他层 递归地传播这些 相关特性。 异步操作的 stacking 复制了 同步操作的 另一个属性。

Property of synchronous operations	Equivalent property of asynchronous operations
可以重构同步操作的组合，以使用在 同一个线程上运行的 子函数(即简单调用)，而不改变功能	异步操作 可以被 重构为 异步操作，和 共享了父代理相关特性的子代理， 而不改变功能。

最后，一些异步操作 可以通过 并发的多个子代理 来实现，这种情况下，异步操作可以 选择性地 传播 父代理 的相关特性。

Executors

https://www.boost.org/doc/libs/1_80_0/doc/html/boost_asio/overview/model/executors.html

每个 异步代理 都有一个 关联的 executor。

一个 agent 的 executor 决定了 agent 的 completion handler 如何 queue 并最终 run。

executor 的 示例用途包括：

1. 协调 在 共享的数据结构 上 运行的一组 异步agent， 确保 agent 的 completion handler 不会 并发 run。
2. 确保 agent 在 接近数据 或 事件源(如NIC (网卡)) 的 指定处理资源(如CPU) 上 run。
3. 表示一组相关的代理， 从而使动态线程池能够做出 更智能的 调度决策 (例如， 将 代理作为 一个单元 在 执行资源之间 移动)
4. 将 所有 completion handler 排队 以 在一个GUI应用线程 上 run， 以便 它们可以 安全地更新用户界面元素。
5. 返回 异步操作的 默认executor， 来使得 completion handler 足够地接近 触发 操作的completion 的 event
6. 调整 异步操作的 默认 executor， 以便在 每个 completion handler 之前 和 之后 运行代码， 如 日志， 用户授权， 异常处理。
7. 指定 异步agent 和 它的 completion handler 的 优先级。

异步agent 中的 异步操作 使用 agent 的关联的 executor 来：

1. 跟踪 异步操作 所代表的工作， 当操作未完成时。
2. 将 completion handler 排队 以便在 操作完成时 执行。
3. 确保 completion handler 不会以 可重入方式(re-entrantly) 运行， 如果可重入的话， 会导致 意外的递归 和 堆栈溢出。

因此， 异步agent 的 关联的 executor 代表了一个策略 来： how, where, when, agent 应该 run， 该策略 被指定为 构造代理的 代码的 交叉关注点。(。。。切面？)

Allocators

每个异步**agent** 有一个 关联的 allocator， 一个 agent 的 allocator 是 一个接口， 被 agent 的 异步操作 使用 来 获得 per-operation stable memory resources (POSMs) (每个操作的 稳定内存资源)。 这个名字反应了一个事实： 内存 是 per-operation， 因为 内存只在 操作的 生命周期中 保留，并且是 stable， 因为 内存被保证 在 整个操作期间 在 该位置可用。

异步操作可以以 多种不同方式 运用 POSMs：

1. 操作不需要 任何 POSMs。 例如， 操作 封装了 一个 已存在的 执行它自己的内存管理的API， 或 复制 长期存活的 state 到 已有的内存(如， circule buffer)。
2. 操作 使用一个 single, fixed-size POSM， 只要 操作未完成。 例如， 操作 存储一些 state 到 linked list 中。
3. 操作使用了一个 single, 运行时确定size的 POSM， 比如， 操作 保存 用户提供的数据的 副本， 或 一个 运行时size 的 iovec 结构。
4. 操作 并发地 使用 多个 POSMs。 例如， 一个固定size的 POSM 作为 链表， 一个 runtime-sized POSM 作为 buffer。

5. 操作 连续使用多个 POSM，大小可能不同。

关联的allocator 允许用户将 POSM优化 视为 异步操作组合的 交叉关注点。
此外，使用分配器作为获取 POSM的接口 为 异步操作的 实现者 和 用户提供极大的灵活性：

1. 用户可以忽视 allocator，如果 对于应用，使用任何默认策略 都可以。
2. implementers 可以忽视 allocator，特别是在操作 不被 视为性能敏感的情况下。
3. 用户可以 共同定位 POSM，以获得 更好的 引用位置。
4. 对于包含 不同大小的 串行POSM的 组合，内存使用量只需要 与当前现存的 POSM一样大。例如，考虑一个组合，它包含一个使用 大型POSM（连接建立 和 握手）的短时间操作，然后是 一个 使用 小型POSM(与对等端 进行读写传输) 的长时间操作

如前所述，在 调用 completion handler 之前，所有的资源都必须被释放。这使得 内存能被 agent 的 后续异步操作 使用。这允许 长期存活的 异步代理的 应用 没有 hot-path memory allocator，即使 用户代码 不清楚 关联的 allocator。

。。。。看着看着，就变成 1.81 了。。。

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/model/cancellation.html

Cancellation

在 Boost.Aasio，许多对象，比如 socket, timer， 支持 通过 它们的 close 或 cancel 成员函数 来 在对象范围内 取消 未完成的 异步操作。

但是，某些异步操作也支持 单独的，带目标的 cancellation。这种 per-operation 的 cancellation 被启用 通过 指定 每个异步agent 有一个 关联的 cancellation slot。

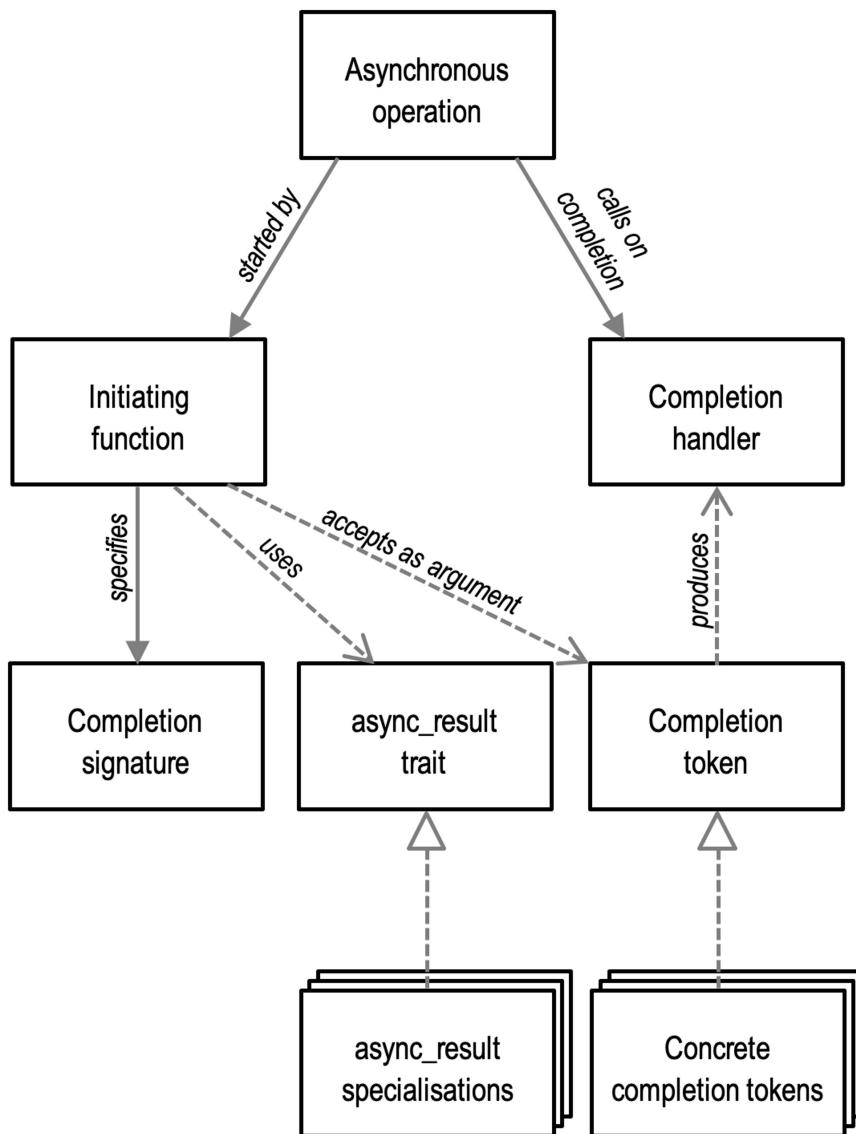
为了支持 cancellation，一个异步操作 安装 一个 cancellation handler 到 agent的 slot中。这个 cancellation handler 是一个 function object，当用户将 cancel信号 发送到 slot 中时 被 调用。

由于 一个 cancellation slot 只和 一个agent 关联，所以 slot 最多 hold 一个 handler， 安装一个 新的 handler 会 覆盖 之前 安装的 handler。因此，同一个slot 被重新用于 agent 内的 后续 异步操作

当异步操作 包含多个 子操作时， cancel 特别有用。例如，一个 子代理 可能已完成，另一个子代理 则 被 cancel，因为 不再需要 它的副作用。

Completion Tokens

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/model/completion_tokens.html



Boost.Asio 的 异步模型的一个 关键目标是 支持 多组成机制(multiple composition mechanisms)。

这是通过 completion token 来实现的，用户把 completion token 传递给 异步操作的 起始函数 来 消费 库的API。

按照惯例，completion token 是 异步操作的 起始函数的 最后一个参数。

例如，如果用户 传递一个lambda (或其他 函数对象) 作为 completion token， 异步操作的行为 就像 之前描述的： 操作开始，and 当操作完成时，result 被传递给 lambda。

```

socket.async_read_some(buffer,
    [](error_code e, size_t)
{
    // ...
});
  
```

当用户传递 use_future 这个completion token 时，operation 的行为就像是 根据 promise 和 future 对(pair) 实现的一样。

```

future<size_t> f =
socket.async_read_some(
    buffer, use_future
  
```

```

);
// ...
size_t n = f.get();

```

类似的，当用户传递 `use_awaitable` 这个completion token 时，起始函数的行为 就像 基于 `awaitable` 的 协程。但是，这种情况下，起始(initiating)函数 不会 直接启动 异步操作。它只会 `return awaitable`，当它是 `co_await-ed` 时，它会 启动 操作。

```

awaitable<void> foo()
{
    size_t n =
        co_await socket.async_read_some(
            buffer, use_awaitable
        );
    // ...
}

```

最后，`yield_context`，这个completion token 导致 起始函数 的行为 就像 `stackful` 协程的 上下文中的 `sync`操作。

```

void foo(boost::asio::yield_context yield)
{
    size_t n = socket.async_read_some(buffer, yield);

    // ...
}

```

所有的这些用法 都由 `async_read_some` 起始函数 的 一个 单个实现 支持。

为了实现这个，一个 异步操作 必须首先 指定 一个 completion signature (or, possibly, signatures)，它介绍了 将被传递给 completion handler 的 参数。然后，`operation` 的 起始函数 获得 completion signature, completion token, 和 它内部的实现 并 传递它们到 `async_result` 特质(trait)。 `async_result` 特质 是一个 客制化点，它将这些特性结合起来，首先生成一个 具体的 completion handler，然后启动操作。

`template <...>, class CompletionToken>`

`DEDUCED async_read_some(<...>, CompletionToken&& token);`

Completion signature: `void(error_code, size_t);`

User calls `async_read_some` and supplies a concrete completion token

`async_read_some` initiating function provides internal implementation

`async_result` trait

`ReturnType async_read_some(<...>, ConcreteToken&& token);`

为了在实践中看到这个，我们使用一个 `detached thread`(分离线程)，以适应 一个 同步操

作 到 一 个 异步操作:

```
template <
    completion_token_for<void(error_code, size_t)>[1]
    CompletionToken>
auto async_read_some(
    tcp::socket& s,
    const mutable_buffer& b,
    CompletionToken&& token)
{
    auto init = []([2]
        auto completion_handler,
        tcp::socket* s,
        const mutable_buffer& b)
    {
        std::thread([3]
            [](
                auto completion_handler,
                tcp::socket* s,
                const mutable_buffer& b
            )
        {
            error_code ec;
            size_t n = s->read_some(b, ec);
            std::move(completion_handler)(ec, n); [4]
        },
        std::move(completion_handler),
        s,
        b
    ).detach();
};

return async_result<[5]
    decay_t<CompletionToken>,
    void(error_code, size_t)
>::initiate(
    init, [6]
    std::forward<CompletionToken>(token), [7]
    &s, [8]
    b
);
}
```

1. `completion_token_for` 检查 用户提供的 `completion token` 是否满足 指定的 `completion signature`。对于老版本的C++，使用 `typename` 来代替
2. 定义一个 函数对象，包含了 代码来启动 异步操作。它会被传递给 具体的 `completion handler`，后面跟着 通过 `async_result` 传递的 任何其他参数。
3. 函数对象的 `body` 生成一个新的`thread` 来执行 操作。
4. 一旦操作完成，结果被传递到 `completion handler`。
5. `async_result trait` 被传递了 (decayed) `completion token type` 和 异步操作的

- completion signature。
6. 调用 trait 的 initiate 成员函数，首先 传递 启动操作的 函数对象。
 7. 然后 是转发的 completion token。trait的实现 会将 它 转换为 一个具体的 completion handler。
 8. 最后，传递 函数对象的 任何其他的参数。假设它们可以被 trait实现 decay-copied and moved

实际上，我们应该调用 async_initiate 帮助函数，而不是 直接 使用 async_result。async_initiate 函数 自动 执行 completion token 的 必要的 decay 和转发，且 支持 遗留的 completion token 实现的 向后兼容。

```
template <
    completion_token_for<void(error_code, size_t)>
    CompletionToken>
auto async_read_some(
    tcp::socket& s,
    const mutable_buffer& b,
    CompletionToken&& token)
{
    auto init = /* ... as above ... */;

    return async_initiate<
        CompletionToken,
        void(error_code, size_t)
    >(init, token, &s, b);
}
```

我们可以 将 completion token 视为 一个 原型 completion handler。

这种情况下，我们传递一个 函数对象（比如lambda）作为 completion token， 它已经满足了 completion handler 的要求。

async_result 主模板 通过 简单地转发参数 来处理这种情况：

```
template <class CompletionToken, completion_signature... Signatures>
struct async_result
{
    template <
        class Initiation,
        completion_handler_for<Signatures...> CompletionHandler,
        class... Args>
    static void initiate(
        Initiation&& initiation,
        CompletionHandler&& completion_handler,
        Args&&... args)
    {
        std::forward<Initiation>(initiation)(
            std::forward<CompletionHandler>(completion_handler),
            std::forward<Args>(args)...);
    }
};
```

我们可以看到 这个默认实现 避免了 所有参数的copy，从而确保了 急启动的 高效。

另一方面， lazy completion token (比如 `use_awaitable`) 可能 捕获这些参数 以 延迟启动 operation。例如，一个 微不足道的 延迟token 的 专门化(只是将operation打包以供后续使用) 可能看起来：

```
template <completion_signature... Signatures>
struct async_result<deferred_t, Signatures...>
{
    template <class Initiation, class... Args>
    static auto initiate(Initiation initiation, deferred_t, Args... args)
    {
        return [
            initiation = std::move(initiation),
            arg_pack = std::make_tuple(std::move(args)...)
        ](auto&& token) mutable
        {
            return std::apply(
                [&](auto&&... args)
                {
                    return async_result<decay_t<decltype(token)>,
Signatures...>::initiate(
                        std::move(initiation),
                        std::forward<decltype(token)>(token),
                        std::forward<decltype(args)>(args)...);
                });
        },
        std::move(arg_pack)
    );
}
};
```

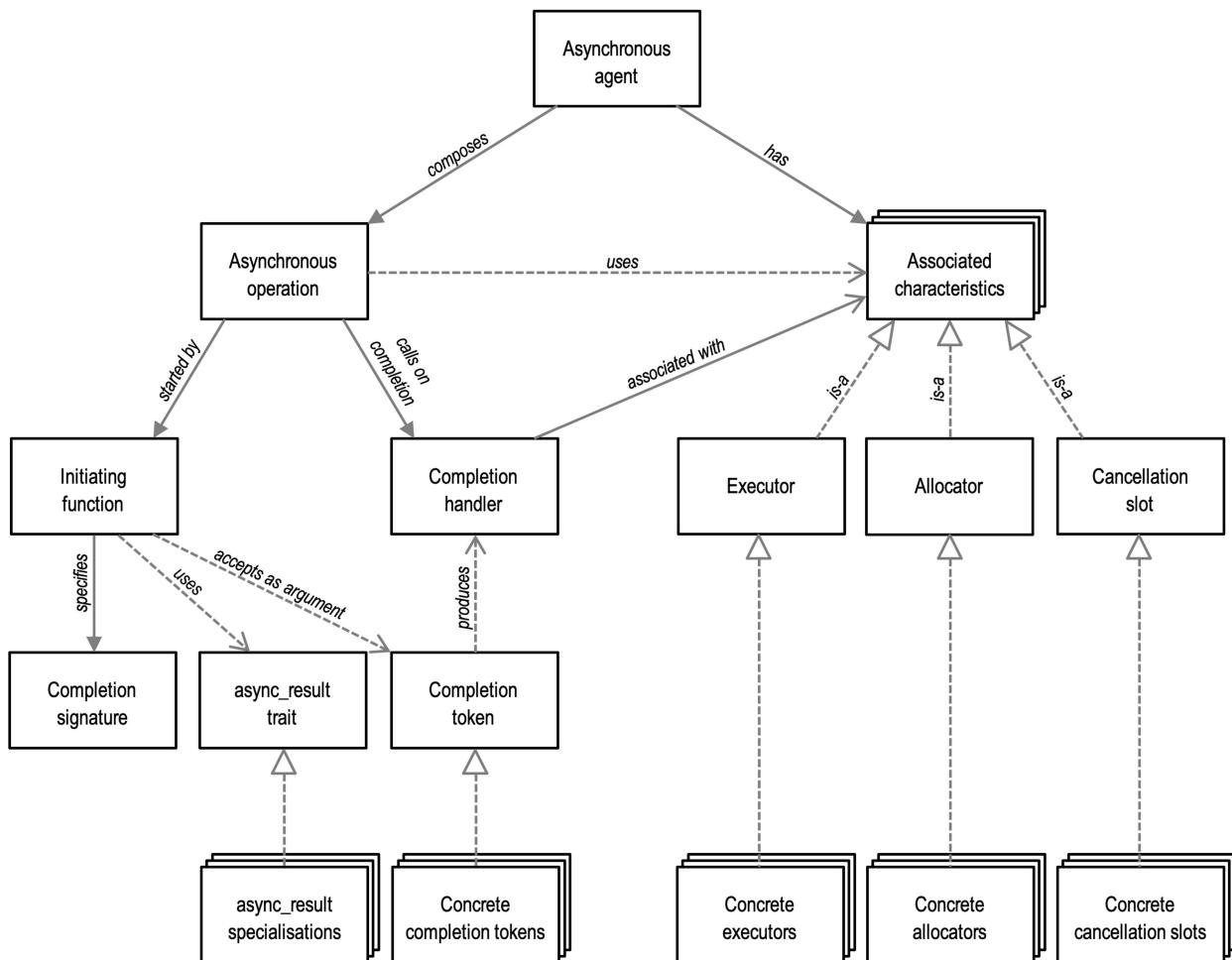
Supporting Library Elements

Boost.Asio 异步模型 被 下面列举的 库元素 启用。

library element	description
<code>completion_signature</code>	定义有效的 completion signature 格式
<code>completion_handler_for</code>	确定 completion handler 是否可以被 一个给定的 completion signature 的集合 调用。
<code>async_result</code>	将 completion signature 和 completion token 转换为 一个 具体的 completion handler，并 启动 operation
<code>async_initiate</code> 函数	帮助函数 来简化 <code>async_result</code> 的 使用。
<code>completion_token_for</code>	确定 completion token 是否 为指定的 completion signature 集合 生成 completion handler

associator	通过抽象层 自动传播所有 associator
associated_executor trait, associated_executor_t type alias, and get_associated_executor function	定义 异步agent 的 关联的 executor
associated_allocator trait, associated_allocator_t type alias, and get_associated_allocator function	定义 异步agent 的 关联的 allocator
associated_cancellation_slot trait, associated_cancellation_slot_t type alias, and get_associated_cancellation_slot function	定义 异步agent 的 关联的 cancellation slot

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/model/higher_levels.html
Higher Level Abstractions



这个异步模型 为 高层抽象 提供了一个 基础。

Boost.Asio 基于这个核心模型来提供其他的设施，例如：

1. I/O 对象 比如 socket 和 timer，它们在此模型之上 公开异步操作。
2. 具体的 executor，如 io_context, thread_pool, 和 标准适配器(保证 completion handler 的 非并发执行)
3. 不同组合机制的 completion token，如 C++ 协程， stackful 协程， future， deferred operation。
4. 对C++ 协程的 高级支持，它结合了 executor 和 cancel slot，来允许 并发异步代理的 easy 协程。

为了让用户 更容易地 编写 他们自己的 遵从这个模型的 异步操作，Asio 提供了 帮助函数 `async_compose`。

Core Concepts and Functionality

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/async.html

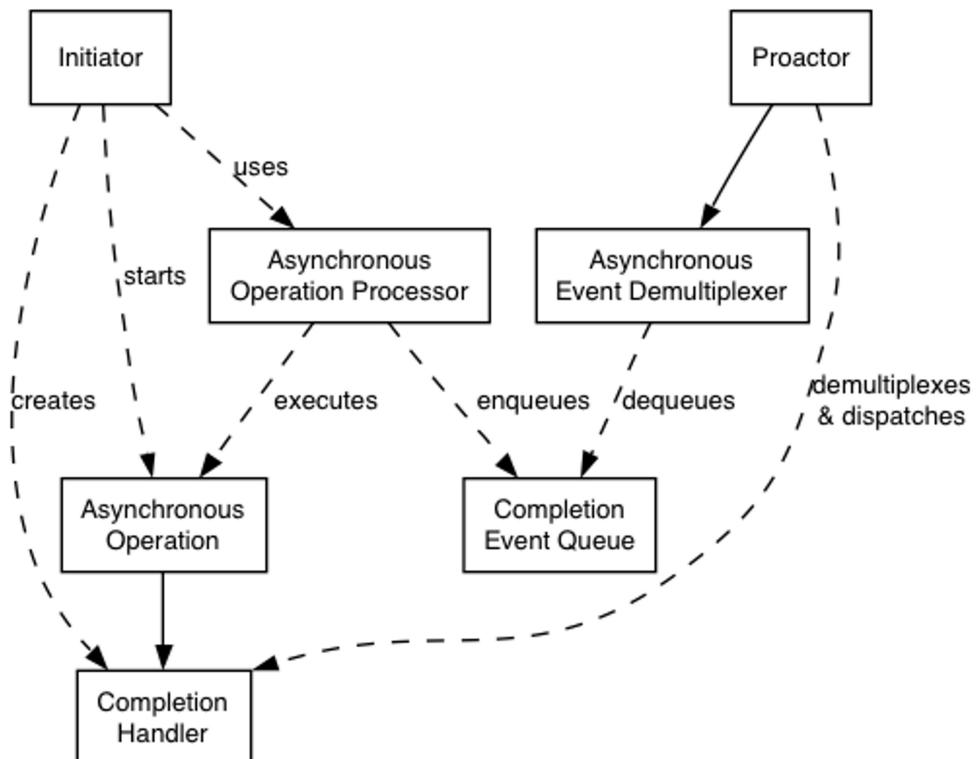
The Proactor Design Pattern: Concurrency Without Threads

Boost.Asio 库 同时支持 同步和异步 操作。

异步支持 是基于 Proactor 设计模式。这种实现 与 同步或Reactor 的 优劣，下面会列举。

Proactor and Boost.Asio

让我们检查 Proactor设计模式 在 Boost.Asio中是怎么 实现的，(不考虑平台特定的 细节)



Proactor design pattern (adapted from [POSA2])

Asynchronous Operation

定义 异步执行的 操作，比如 socket 上的 异步 读或 写。

Asynchronous Operation Processor

执行 异步操作， 操作完成后， 将 event 入队到 一个 completion event queue。

从一个高级别的视点来看，因特网服务 比如 reactive_socket_service 是 异步操作处理器

Completion Event Queue

缓冲completion event，直到 它们被 异步event 多路分配器(demultiplexer) 出队。

Completion Handler

处理 异步操作的结果。这些是 function object，通常使用 boost::bind 来创建。

Asynchronous Event Demultiplexer

阻塞等待 event 在 completion event queue 中发生，返回一个 completed event 到 它的caller。

Proactor

调用 异步event 多路分配器 来 将 event 出队，分发 event 给 completion handler (也就是，调用function object)。这个 抽象 通过 io_context 类表示。

Initiator

APP特定的代码 来启动 异步操作。initiator 通过一个 高层接口(如， basic_stream_socket) 来和 异步操作处理器 交互。

Implementation Using Reactor

在许多平台，Boost.Aasio 以 Reactor 的思想 来实现 Proactor 设计模式，比如 select , epoll, kqueue。

该实现方式对应于 Proactor 设计模式，如下所示：

Asynchronous Operation Processor

使用 select, epoll 或 kqueue 实现的 reactor。当 reactor 指示 资源可以用于执行 操作时， processor 执行 异步操作 且 将 对应的 completion handler 入队到 completion event queue。

Completion Event Queue

由 completion handler(即 function object) 组成的 linked list

Asynchronous Event Demultiplexer

通过以下方式实现： 等待 event 或 condition variable 直到 completion handler 在 completion event queue 中 可用。

Implementation Using Windows Overlapped I/O

在 Window 上， Boost.Asio 使用了 overlapped IO 的优势 来 提供 高效的 Proactor 设计模式的 实现。 这个实现对应于 Proactor 设计模式，如下所示：

Asynchronous Operation Processor

这个被 OS 实现。 通过调用一个 overlapped function (比如 AcceptEx) 来 初始化 操作。

Completion Event Queue

这个被 OS 实现， 被关联到 一个 IO completion port。每个 io_context 实例都 有一个 IO completion port。

Asynchronous Event Demultiplexer

Boost.Asio 调用， 来 将event 和 它关联的completion handler 出队，

Advantages

Portability(可移植性)

许多OS 提供一个 native 异步 IO API (比如 window上的 overlapped IO) 作为 开发 高性能互联网APP 的 首选项。 库 可以根据 native 异步IO 来实现。 但是， 如果native 支持 不可用， 库 也可以 使用 代表Reactor模式的 同步event demultiplexor (比如 POSIX select()) 来实现

Decoupling threading from concurrency. (从并发中 解耦线程)

持续时间长 的操作 由 APP的实现 异步执行。 因此，APP不需要 创建很多 线程 来 增加 并发性。

Performance and scalability.

一些实现策略，如 thread-per-connection 会降低系统性能。因为 增加了 上下文切换，synchronisation 和 CPU间数据移动。 通过异步操作，可以避免 上下玩切换 的 cost 通过 最小化 OS线程(这是一个有限的资源) 的 数量， 和 仅激活 要处理事件 的 控制逻辑线程。

Simplified application synchronisation.

异步操作 completion handler 可以 被 编码， 就像 它们存在于 单线程环境中， APP逻辑 可以在 很少 或不考虑 同步问题的 情况下 开发。

Function composition.

函数组合 是指 实现函数 以提供更高级别的 操作，例如 以特定格式 发送消息。每 个函数 都是通过 多次 调用 较低级别的 读 或 写 操作 来实现的。

例如，考虑一个 协议：每条消息 包含 固定长度的 header + 可变长度的body， body 的长度在 header中 指定。 一个 read_message 操作 可以 被实现为：使用 2个低级 读，第一个接受 header，一旦 长度知道后，第二个接收 body。

要在 异步模型中 组合函数，异步操作 可以被 chain 起来。即，一个操作的 completion handler 可以 启动 下一个 操作。 启动chain 的第一个call 可以被封装，这样 caller 就不必知道 高级操作 是 通过 异步操作chain 来实现的。

这种 组合新操作的 能力 简化了 网络库之上 的 更高级别抽象的 开发，例如 支持 特定协议的 功能。

Disadvantages

Program complexity.

由于 operation 启动 和 完成 之间的 时间空间的 割裂，使得 使用异步机制 开发 APP 更困难。 由于控制流的 invert(反转)，更难debug

Memory usage.

在 读或写 操作期间 必须提交 缓冲区空间，这个可能会无限持续，并且 每个 并发操作 都有一个 独立的 buffer。

Reactor模式，不需要 buffer space 直到 socket 准备好 读 写

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/thread.html
Threads and Boost.Aasio

Thread Safety

通常，并发地 使用不同对象 是安全的，但是 对一个对象的 并发使用 是 不安全的。然而，一些类型(比如 io_context) 提供了 保证：同时使用单个对象是安全的。

Thread Pools

多线程 可以调用 `io_context::run()` 来 设置一个 线程池，可以从中调用 completion handler。这个方法也可以和 `post()` 一起使用，作为一种手段 来 在线程池中 执行任意的计算task。

注意，所有加入到 `io_context` 的 池 中的 线程 被认为是等价的，`io_context` 可以以任意方式 在它们之间 分配工作。

Internal Threads

正对某些平台的 此库的实现 可能使用了 一个或多个 内部线程 来 模拟 异步。这些线程 必须尽可能对库用户不可见，特别是，这些 线程 必须不会 直接调用 用户的代码，必须阻断所有信号。

该approach由以下保证来实现：异步 completion handler 只会被 当前正在调用 `io_context::run()` 的线程 调用。

因此，这是库的用户的责任 来 创建和管理 将向其发送通知的 所有线程。

这个approach的原因包括：

1. 通过 从一个单独线程 只调用`io_context::run()`，用户的代码 可以 避免 sync的开发的复杂性。例如，一个库user 可以实现 (在用户看来是) 单线程的 可伸缩的 server。
2. 库user 可能需要 在线程启动后，其他代码执行前， 立刻 执行一些初始化。例如，微软的 COM 的 用户 必须 调用 `CoInitializeEx`，在任何其他 COM操作 被 那个线程调用前。
3. 库接口 和 用于线程创建和管理的 接口 分离，并 允许 在线程不可用的 平台上 实现。

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/thread.html

Strands: Use Threads Without Explicit Locking

strand 被定义为: event handler 的严格的调用顺序 (即, 没有并发调用)。使用strand 允许 多线程程序 中的 代码的 执行 而不需要 显式lock。

strand 可能是 显式 或 隐式的, 如如下 替代方法所示:

1. 只从 一个线程中 调用 `io_context::run()`, 意味着 所有 event handler 是 隐式 strand 执行的, 由于 `io_context` 的保证: handler只会在 `run()` 中被调用。
2. 当有一个 异步操作的 `single` 链 关联到 一个 `connection`, handler 就不可能 并发执行, 这是一个 隐式strand。
3. 显式 strand 是 `strand<>` 或 `io_context::strand` 的 实例。所有的 event handler 函数对象 需要 通过 `boost::asio::bind_executor()` 绑定到 strand, 或 通过 strand对象来被 posted/dispatched 。

组合的异步操作 的情况下, 如 `async_read()` 或 `async_read_until()`, 如果 `completion handler` 通过 strand 运行, 那么 所有的 中间 handler 都 应该 通过 相同的 strand 运行。这里需要确保 调用方 和 组合操作 之间 共享的任何对象 是线程安全的 (比如, `async_read()` 是socket, 调用者可以 `close()` 来 取消操作)。

要达成这个, 所有的 异步操作 获得 handler 关联的 executor, 通过 使用 `get_associated_executor` 函数:

```
boost::asio::associated_executor_t<Handler> a =
boost::asio::get_associated_executor(h);
```

关联的 executor 必须 满足 Executor要求。它 会被 异步操作 使用 来 submit 中间 和 final handler

executor 可以被 定制 对于特定的handler type, 通过 指定一个 嵌套的类型 `executor_type` 和 成员函数 `get_executor()`。

```
class my_handler
{
public:
    // Custom implementation of Executor type requirements.
    typedef my_executor executor_type;

    // Return a custom executor implementation.
    executor_type get_executor() const noexcept
    {
        return my_executor();
    }

    void operator()() { ... }
};
```

更复杂的情况, `associated_executor` 模板 可以被 直接 部分特殊化

```
struct my_handler
{
```

```

    void operator() () { ... }

};

namespace boost { namespace asio {

    template <class Executor>
    struct associated_executor<my_handler, Executor>
    {
        // Custom implementation of Executor type requirements.
        typedef my_executor type;

        // Return a custom executor implementation.
        static type get(const my_handler&,
            const Executor& = Executor()) noexcept
        {
            return my_executor();
        }
    };
}

} } // namespace boost::asio

```

`boost::asio::bind_executor()` 函数 是一个 helper 来 绑定 具体的executor 对象 (比如 一个strand) 到 一个completion handler。

如之前所示，这个绑定 自动关联 一个 executor。例如， 要绑定 一个 strand 到 一个 completion handler ， 我们 只需要写：

```

my_socket.async_read_some(my_buffer,
    boost::asio::bind_executor(my_strand,
        [] (error_code ec, size_t length)
    {
        // ...
    }));

```

Buffers

https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/buffers.html

从根本上讲，I/O 涉及 转换数据 from/to 连续的内存(被称为 buffer)。

这些 buffer 可以被 简单地表达为 包含 一个指针 和 byte大小 的 元组。

然而，为了 允许 开发 高效的 互联网应用， Boost.Asio 包含了 对 scatter-gather 操作的支持。 这些操作 调用 一个 或 多个 buffer:

- 一个 scatter-read 接收 数据 到 多个buffer
- 一个 gather-write 发送 多个buffer。

。 。 。 scatter: 分散, gather: 聚集

因为，我们需要 一个抽象 来代表 buffer的 集合。 Boost.Asio使用的 方法 是 定义 一

个 类型(实际上是2个类型) 来 表达 一个buffer。 它们能被 保存到 容器中，然后 被传递到 scatter-gather 操作。

除了 定义 buffer 为 一个 指针 + byte size 外， Boost.Asio 在 可修改的内存 和 不可修改的内存 间 做了 区分。 因为 有了 如下的 2种类型

```
typedef std::pair<void*, std::size_t> mutable_buffer;
typedef std::pair<const void*, std::size_t> const_buffer;
```

`mutable_buffer` 可以转为 `const_buffer`， 反之不行。

但是， Boost.Asio 不是 直接使用 上面的 定义， 而是 定义了 2个类： `mutable_buffer` 和 `const_buffer`。 这样做的目的是 提供一个 连续内存的 不透明的 表示， 其中：

类型 可以被转换， 即， `mutable_buffer` 可以转为 `const_buffer`， 但是 反之不行。

防止buffer 超支， 给定一个 buffer 实例， 用户可以 创建 一个新的 buffer， 但是 代表了相同的 范围， 或 有重叠。 为了进一步提供安全， 库 也包含了 机制 来 自动 确定 来自 array, std::string, POD元素的 boost::array 或 std::vector, 的 buffer 的size。

通过 `data()` 成员函数 可以 显式访问底层内存。 一般情况下， APP永远不需要这样 做， 但是 有些情况下 是需要的： 库 传递 原生 内存 到 操作系统的 函数。

最后，多buffer 可以被 传递到 scatter-gather 操作 (比如 `read()` , `write()`) 通过 将 buffer 对象 放到 一个 容器中。

`MutableBufferSequence` 和 `ConstBufferSequence` 的概念被 定义， 这样 容器(如 `std::vector`, `list`, `array`, `boost::array`) 可以被 使用。

Streambuf for Integration with Iostreams

类 `boost::asio::basic_streambuf`， 继承自 `std::basic_streambuf`，

=====

=====

=====

=====