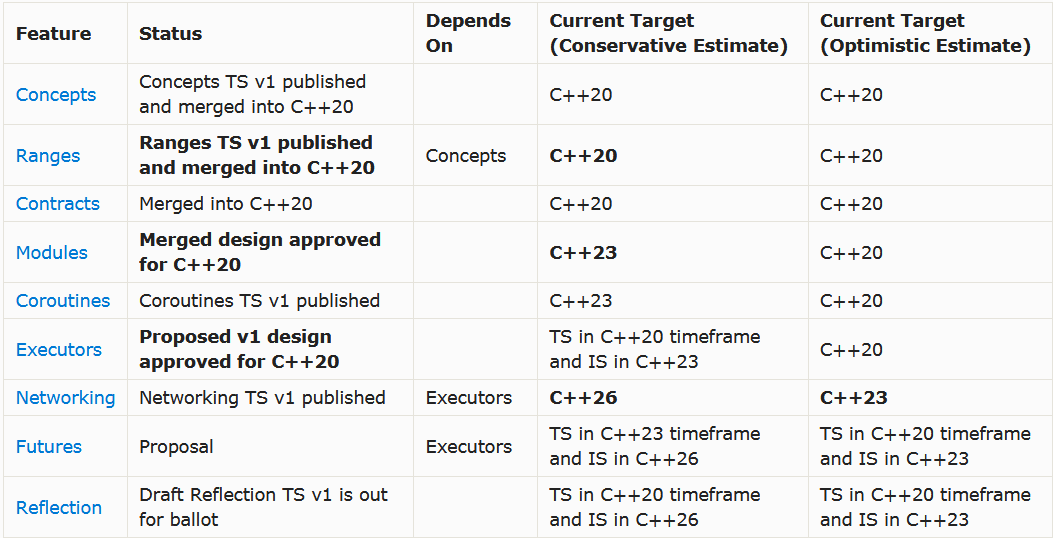
**C++20的一些重要特性**

(本文来源于祁宇(《深入应用C++11》作者), 略作修改)

C++20中可能增加哪些重要特性，下面这个图可以提供一个参考。



**01 Concepts**

模版在C++中备受推崇，大获成功，在各种C++库（如STL）中被广泛使用。

然而，模版编程还存在一些问题，比如有些模版的代码写起来比较困难，读起来比较难懂，尤其是编译出错的时候，那些糟糕的让人摸不着头脑的错误提示让人头疼。

因此，希望对模版做一些改进，**让C++的模版编程变得简单好写，错误提示更明确。**

具体思路就是给模版参数加一些约束，这些约束相比之前的写法具有更强的表达能力和可读性，会简化C++的泛型模版代码的编写。

所以Concepts的出现主要是为了简化泛型编程，一个Concept就是一个编译期判断，用于约束模版参数，Concepts则是这些编译期判断的合集。下面通过一个例子来展示Concepts是如何简化模版编程的。

template<typename T>

class B {

public:

template<typename ToString = T>

typename std::enable\_if\_t<std::is\_convertible<ToString,std::string>::value, std::string>

to\_string() const { return "Class B<>"; }

};

B<size\_t> b1; // OK

std::cout << b1.to\_string() << std::endl; // Compile ERROR!

B<std::string> b2; // OK

std::cout << b2.to\_string() << std::endl; // OK!

比如有这样一个类B，我们调用它的成员函数tostring时，对T类型进行限定，即限定T类型是std::string的可转换类型，这样做的目的是为了更安全，能在编译期就能检查错误。

这里通过C++14的std::enableif\_t来对T进行限定，但是长长的enableift看起来比较冗长繁琐，头重脚轻。来看看用Concepts怎么写这个代码的。

template<typename T>

concept CastableToString = requires(T a) {

{ a } -> std::string; };

template<typename T>

class D {

public:

std::string to\_string() const requires CastableToString<T> {

return "Class D<>"; }

};

可以看到，requires CastableToString比之前长长的enableift要简洁不少，代码可读性也更好，CastableToString就是一个Concept，一个限定T为能被转换为std::string类型的Concept，通过requires相连接，语义上也更明确了，而且这个Concept还可以复用。

**02 Ranges**

相比STL，Ranges是更高一层的抽象，Ranges对STL做了改进，它是STL的下一代。为什么说Ranges是STL的未来？虽然STL在C++中提供的容器和算法备受推崇和广泛被使用，但STL一直存在**两个问题：**

* STL强制你必须传一个begin和end迭代器用来遍历一个容器；
* STL算法不方便组合在一起。

STL必须传迭代器，这个迭代器仅仅是辅助你完成遍历序列的技术细节，和我们的函数功能无关，大部分时候我们需要的是一个range，代表的是一个比迭代器更高层的抽象。

那么Ranges到底是什么呢？Ranges是一个引用元素序列的对象，在概念上类似于一对迭代器。这意味着所有的STL容器都是Ranges。在Ranges里我们不再传迭代器了，而是传range。比如下面的代码：

STL写法：

    std::vector<int> v{1, 2};  
    std::sort(v.begin(), v.end());

Ranges写法：

    std::sort(v);

STL有时候不方便将一些算法组合在一起，来看一个例子：

    std::vector<int> v{1, 2, 3, 4, 5};  
    std::vector<int> event\_numbers;  
    std::copy\_if(v.begin(), v.end(), std::back\_inserter(event\_numbers), [](int i){ return i % 2 == 0;});  
    std::vector<int> results;  
    std::transform(event\_numbers.begin(), event\_numbers.end(), std::back\_inserter(event\_numbers), [](int i){ return i \* 2;});  
    for(int n : results){   std::cout<<n<<' ';   }    //最终输出 4 8

上面这个例子希望得到vector中的偶数乘以2的结果，需求很简单，但是用STL写起来还是有些冗长繁琐，中间还定义了两个临时变量。如果用Ranges来实现这个需求，代码就会简单得多。

    auto results = v | ranges::view::filter([](int i){ return i % 2 == 0; })  
                     | ranges::view::transform([](int i){ return i \* 2; });

用Concetps我们可以很方便地将算法组合在一起，写法更简单，语义更清晰，并且还可以实现延迟计算避免了中间的临时变量，性能也会更好。

Concepts从设计上改进了之前STL的两个问题，让我们的容器和算法变得更加简单好用，还容易组合。

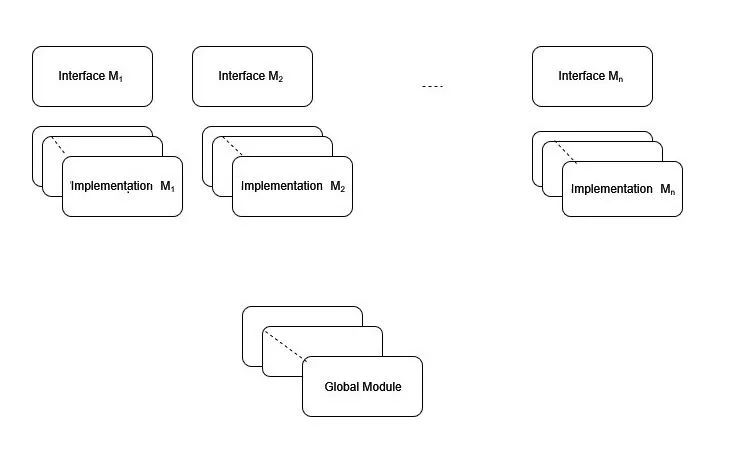
**03 Modules**

一直以来C++一直通过引用头文件方式使用库，而其他90年代以后的语言比如Java、C#、Go等语言都是通过import包的方式来使用库。现在C++决定改变这种情况了，在C++20中将引入Modules，它和Java、Go等语言的包的概念是类似的，直接通过import包来使用库，再也看不到头文件了。

为什么C++20不再希望使用#include方式了？因为使用头文件方式存在不少问题，比如有include很多模版的头文件将大大增加编译时间，代码生成物也会变大。

而且引用头文件方式不利于做一些C++库和组件的管理工具，尤其是对于一些云环境和分布式环境下不方便管理，C++一直缺一个包管理工具，这也是C++被吐槽得很多的地方，现在C++20 Modules将改变这一切。

Modules在程序中的结构如下图：



上面的图中，每个方框表示一个翻译单元，存放在一个文件里并且可以被独立编译。每个Module由Module接口和实现组成，接口只有一份，实现可以有多份。

Modules接口和实现的语法：

    export module module\_name;  
    module module\_name;

使用Modules：

    import module\_name;

Modules允许你导出类，函数，变量，常量和模版等等。

接下来看一个使用Modules的例子：

    import std.vector; // #include <vector>  
    import std.string; // #include <string>  
    import std.iostream; // #include <iostream>  
    import std.iterator; // #include <iterator >  
    int main() {  
        using namespace std;  
        vector<string> v = { "Socrates", "Plato", "Descartes", "Kant", "Bacon" };  
        copy(begin(v), end(v), ostream\_iterator<string>(cout, "\n"));  
    }

可以看到不用再include了，直接去import需要用到的Modules即可。

**04 Coroutines**

很多语言提供了Coroutine机制，因为Coroutine可以大大简化异步网络程序的编写，现在C++20中也加入协程了。

协程最明显的一个好处是允许我们书写**看起来顺序执行但是其实背后却异步执行的代码**，这样技能协调人大脑擅长顺序逻辑和计算机处理擅长异步执行的矛盾，兼顾效率和心智负担。 同时协程还可以支持惰性赋值和初始化的逻辑，进一步提高程序的运行效率（仅仅在需要的时候做运算）但是又不对程序员的大脑产生太多的额外负担。

协程是一个比进程和线程更轻量级一点的概念，具体实现上来说可以分为有栈协程和无栈协程；技术上来说前者可以通过第三方库实现就可以，但是性能开销比较大也容易出问题；而**无栈协程更加轻量级但是需要语言特性上做出改动**。

如果不用协程，写一个异步的网络程序是不那么容易的，以boost.asio的异步网络编程为例，我们需要注意的地方很多，比如异步事件完成的回调函数中需要保证调用对象仍然存在，如何构建异步回调链条等等，代码比较复杂，而且出了问题也不容易调试。而协程给我们提供了对异步编程优雅而高效的抽象，让异步编程变得简单！

C++ Courotines中增加了三个新的关键字：co\_await，co\_yield和co\_return，如果一个函数体中有这三个关键字之一就变成Coroutine了。

co\_await用来挂起和恢复一个协程，co\_return用来返回协程的结果，co\_yield返回一个值并且挂起协程。

下面来看看如何使用它们。

写一个lazy sequence：

generator<int> get\_integers( int start=0, int step=1 ) {  
      for (int current=start; current+= step)   co\_yield current;  
    }

    for(auto n : get\_integers(0, 5)){    std::cout<<n<<" ";  }  
    std::cout<<'\n';

上面的例子每次调用get\_integers，只返回一个整数，然后协程挂起，下次调用再返回一个整数，因此这个序列不是即时生成的，而是延迟生成的。

接下来再看一下co\_wait是如何简化异步网络程序的编写的：

char data[1024];  
    for (;;)  
    { std::size\_t n =

co\_await socket.async\_read\_some(boost::asio::buffer(data), token);  
       co\_await async\_write(socket, boost::asio::buffer(data, n), token);  
    }

这个例子仅仅用了四行代码就完成了异步的echo，非常简洁！co\_await会在异步读完成之前挂起协程，在异步完成之后恢复协程继续执行，执行到async\_write时又会挂起协程直到异步写完成，异步写完成之后继续异步读，如此循环。如果不用协程代码会比较繁琐，需要像这样写：

    void do\_read()  
    {      auto self(shared\_from\_this());  
      socket\_.async\_read\_some(boost::asio::buffer(data\_, max\_length),  
        [this, self](boost::system::error\_code ec, std::size\_t length)  
         {    if (!ec)    do\_write(length);   });  
      }  
    void do\_write(std::size\_t length)  
    {  auto self(shared\_from\_this());  
       boost::asio::async\_write(socket\_, boost::asio::buffer(data\_, length),  
        [this, self](boost::system::error\_code ec, std::size\_t /\*length\*/)  
        {    if (!ec)   do\_read();     });  
    }

可以看到，不使用协程来写异步代码的话，需要构建异步的回调链，需要保持异步回调的安全性等等。而使用协程可以大大简化异步网络程序的编写。

**05 Reflection**

C++中一直缺少反射功能，其他很多语言如Java、C#都具备运行期反射功能。反射可以用来做很多事情：比如做对象的序列化，把对象序列化为JSON、XML等格式，以及ORM中的实体映射，还有RPC远程过程（方法）调用等，反射是应用程序中非常需要的基础功能。

现在C++终于要提供反射功能了，C++20中可会将反射作为实验库，在C++23中正式加入到标准中。

在反射还没有进入到C++标准之前，有很多人做了一些编译期反射的库，比如purecpp社区开源的序列化引擎iguana，以及ORM库ormpp，都是基于编译期反射实现的。

然后，非语言层面支持的反射库存在种种不足之处，比如在实现上需要大量使用模版元和宏、不能访问私有成员等问题。

现在C++终于提供完备地编译期反射功能了，为什么是编译期反射而不是像其它语言一样提供运行期反射，因为C++的一个重要设计哲学就是zero-overhead，编译期反射效率远高于运行期反射。

那么，通过C++20的编译期反射我们能得到什么呢？我们可以得到很多很多关于类型和对象的元信息，主要有：

获取对象类型或枚举类型的成员变量，成员函数的类型；

获取类型和成员的名称；

获取成员变量是静态的还是constexpr；

获取方法是virtual、public、protect还是private；

获取类型定义时的源代码所在的行和列。

所以C++20的反射其实是提供了一些可以编译期向编译器查询目标类型“元数据”的API，下面来看看C++20的反射用法：

    struct person{  
        int id;  
        std::string name;  
    };  
    using MetaPerson = reflexpr(person);  
    using Members = std::reflect::get\_data\_members\_t<MetaPerson>;  
    using Metax = std::reflect::get\_data\_members\_t<Members>;  
    constexpr bool is\_public = std::reflect::is\_public\_v<Metax>;  
    using Field0 = std::reflect::get\_reflected\_type\_t<Metax>;// int

上面的例子中，C++20新增关键字reflexpr返回的是person的元数据类型，接下来我们就可以查询这个元数据类型了，std::reflect::getdatamembers\_t返回的是对象成员的元数据序列，我们可以像访问tuple一样访问这个序列，得到某一个字段的元数据之后我们就可以获取它的具体信息了，比如它的具体类型是什么，它的字段名是什么，它是公有还是私有的等等。

C++20的编译期反射实际上提供了一些编译期查询AST信息的接口，功能完备而强大。

**06 总结**

* Concepts让C++的模版程序的编写变得更简单和容易理解；
* Ranges让我们使用STL容器和算法更加简单，并且更容易组合算法及延迟计算；
* Modules帮助我们大大加快编译速度，同时弥补了C++使用库和缺乏包管理的缺陷；
* Coroutines帮助我们简化异步程序的编写；
* Reflection给我们提供强大的编译期AST元数据查询能力；
* ......