五个模块：概论，语言特性，标准库，技能进阶，总结



**Stay Hungry,Stay foolish**

01|重新认识C++：生命周期和范式编程

一个C++程序的四个阶段，编码，预处理，编译，运行：

编译：要遵守代码规范，设计模式，编程惯用法

预处理：C++独有，由预处理器完成，单纯的文字替换，如#include，#define，#if等。

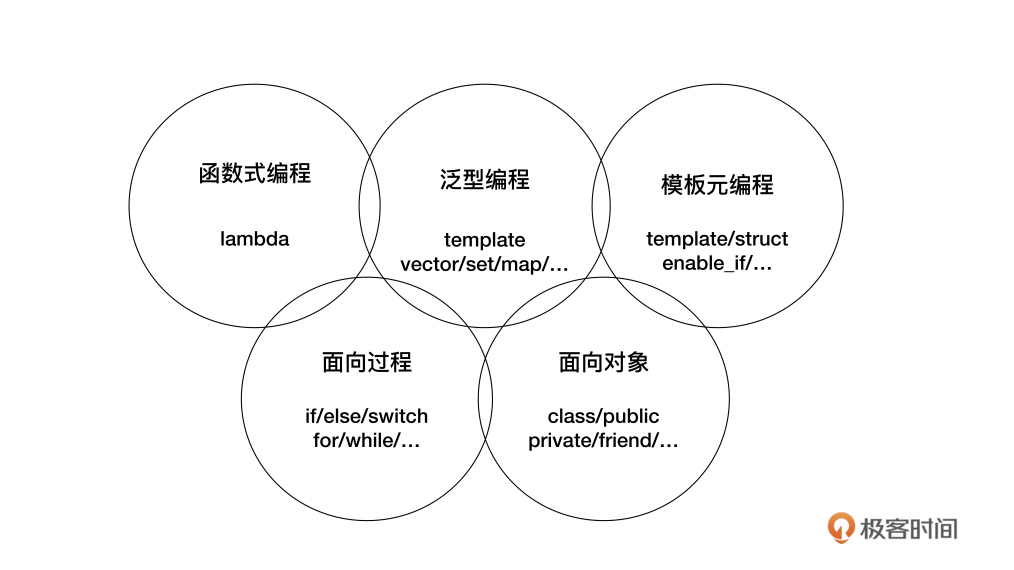
编译：实际是编译和链接，生成二进制机器码，编译器工作时会静态检查，输出编译错误

运行：载入内存，CPU逐行执行

尽量将问题在早期暴露出来，基于蝴蝶效应，后期问题解决成本高

C++语言的范式编程，就是一种方法论，是关于编码时的一些思路，规则，习惯，定式和常用语

C++语言的多范式：面向对象，面向过程，泛型，模板元和函数式。



面向过程：

面向对象：封装和抽象，高聚合低耦合，强调对象之间的关系，包括class，public，private等关键字。以及析构函数等概念。

泛型编程：一切皆为类型，基础是template，然后是STL

模板元：

函数式：核心思想是“一切皆可调用”，lambda表达式

面向对象和面向过程是基础范式，一定要掌握。

02|编码阶段能做什么：秀出好的code style

空格与空行：

起个好名字：snake\_case,加前缀，自定义的类名用驼峰式，变量名字长度跟作用域成正比。

用好注释：

03|预处理阶段可以做什么：宏定义和条件编译

预处理的操作目标是源代码，用各种指令控制预处理器，把原码改造成另一种形式。

预处理指令只有几十个，都是#开头，不受C++语法限制。永远都是定格写

示例：

# // 预处理空行

#if \_\_linux\_\_ // 预处理检查宏是否存在

# define HAS\_LINUX 1 // 宏定义，有缩进

#endif // 预处理条件语句结束

# // 预处理空行

可以用GCC的-E参数输出预处理后的源码。

包含文件（#include）：

可以包含任何文件，源码，文本文件，图片文件等。

为了防止代码被重复包含，要加上“Includ Guard”，既是#ifndef/#defien/#endif/。

也可以将一些代码片段（如比较大的静态数据）存到.inc文件中，然后用#include加载。

宏定义（#define/#undef）：

实际上就是文本替换。宏定义非常强大，可以定义常量/变量，实现函数功能，为类型起别名，减少重复代码。

使用要谨慎，以简化代码，清晰易懂为目标。

宏定义展开在预处理阶段，是源代码级别的无条件内联，适合封装一些调用频繁的小代码片段。

示例：

#define ngx\_tolower(c) ((c >= 'A' && c <= 'Z') ? (c | 0x20) : c)

#define ngx\_toupper(c) ((c >= 'a' && c <= 'z') ? (c & ~0x20) : c)

#define ngx\_memzero(buf, n) (void) memset(buf, 0, n)

宏是全局生效的，对于一些简化代码，起临时作用的宏，要及时地用“#undef”取消定义，避免冲突风险。

举例：

#define CUBE(a) (a) \* (a) \* (a) // 定义一个简单的求立方的宏

cout << CUBE(10) << endl; // 使用宏简化代码

cout << CUBE(15) << endl; // 使用宏简化代码

#undef CUBE // 使用完毕后立即取消定义

也可以定义宏之前先检查。

举例：

#ifdef AUTH\_PWD // 检查是否已经有宏定义

# undef AUTH\_PWD // 取消宏定义

#endif // 宏定义检查结束

#define AUTH\_PWD "xxx" // 重新宏定义

可以适当地用宏来表述代码中的常量。

举例：

#define MAX\_BUF\_LEN 65535

#define VERSION "1.0.18"

条件编译（#if/#else/#endif）

通常编译环境里有一些预定义宏，比如CPU支持的特殊指令集，操作系统/编译器/程序库的版*本，语*言特性等。

举例：

#ifdef \_\_cplusplus // 定义了这个宏就是在用C++编译

extern "C" { // 函数按照C的方式去处理

#endif

void a\_c\_function(int a);

#ifdef \_\_cplusplus // 检查是否是C++编译

} // extern "C" 结束

#endif

#if \_\_cplusplus >= 201402 // 检查C++标准的版本号

cout << "c++14 or later" << endl; // 201402就是C++14

#elif \_\_cplusplus >= 201103 // 检查C++标准的版本号

cout << "c++11 or before" << endl; // 201103是C++11

#else // \_\_cplusplus < 201103 // 199711是C++98

# error "c++ is too old" // 太低则预处理报错

#endif // \_\_cplusplus >= 201402 // 预处理语句结束

除了\_\_cplusplus，C++还有很多预定义的宏，可以用一条简单的gcc指令查看

可以用#if 1/#if 0 来启用或者禁用大段的代码

总结：

C++程序是混合了预处理编程和C++编程两种代码

预处理编程由预处理器执行，使用#include，#define，#if等指令来实现文件包含，文本替换，条件编译，把源代码修改成另一种形式，适当使用可以简化代码，优化性能，但过分使用会导致代码混乱，难以维护。

04|编译阶段能做什么：属性和静态断言，编译阶段编程：在编译阶段生效

属性（attribute），用“[[]]”表示，“属性”像是给编译器的一个提示和告知，举例：

[[noreturn]] // 属性标签

int func(bool flag) // 函数绝不会返回任何值

{

throw std::runtime\_error("XXX");

}

C++11中只有”noreturn”和”carries\_dependency”，C++14中新增了“deprecated”，编译器还支持gcc中的属性，用gun::来使用，举例：

deprecated：与C++14相同，但可以用在C++11

unused：用于变量，类型，函数等，表示虽然暂时不用，但以后可能会用，先保留

constructor：函数会在main函数之前执行，效果有点像全局对象的构造函数

destructor：函数会在main函数之后执行，有点像全局对象的析构函数

always\_inline：要求编译器强制内联函数，作用比inline关键字更强

hot：标记热点函数，要求编译器积极优化

静态断言（static\_assert）：

assert(i > 0 && "i must be greater than zero");

assert(p != nullptr);

assert(!str.empty());

当CPU运行到assert时，会计算表达式的指，如果是false，则调用abort终止程序并输出错误信息。

assert是在运行阶段起作用，又叫动态断言。

静态断言是static\_assert，在编译时生效，用于检查编译时期的断言。

template<int N>

struct fib

{

static\_assert(N >= 0, "N >= 0");

static const int value =

fib<N - 1>::value + fib<N - 2>::value;

};

静态断言只能看到编译时期的类型和常量，看不到运行时的变量，指针等。

错误用法：

char\* p = nullptr;

static\_assert(p == nullptr, "some error."); // 错误用法

判断类型：

// 假设T是一个模板参数，即template<typename T>

static\_assert(

is\_integral<T>::value, "int");

static\_assert(

is\_pointer<T>::value, "ptr");

static\_assert(

is\_default\_constructible<T>::value, "constructible");

小结：

1. “属性”相当于编译阶段的标签，用来标记函数，变量，类，让编译器发出或者不发出警告，还能手工指定代码的优化方式。
2. 官方的很少，可以使用非官方的属性，需要加上命名空间限定
3. Static\_assert是静态断言，在编译阶段计算常数和类型。
4. 和运行期的assert一样，static\_assert可以在编译阶段定义好各种前置条件，充分利用C++静态语言的优势，让编译器做各种检查，避免运行时期的隐患。

05|面向对象编程：怎样才能写出一个好的类

设计思想：抽象和封装

把问题中的实体抽象出来，封装成类和对象

实现原则：尽量少用继承和虚函数，如果无法避免继承，一定要控制继承的层次不要太深

设计类时，尽量让类简单，短小精悍，职责单一

避免嵌套类，新建一个命名空间，把内部类提到外面，降低原来类的耦合度和复杂度

编码准则：

标识符final，用于类定义，显式地禁用继承。

在继承时使用public，不用virtual，protected，因为会使父子类的关系变得难以捉摸，当到达继承底层时，也要及时地用final终止继承、

class Interface // 接口类定义，没有final，可以被继承

{ ... };

class Implement final : // 实现类，final禁止再被继承

public Interface // 只用public继承

{ ... };

现代C++中，一个类总会有六大基本函数：三个构造，两个赋值，一个析构。

构造函数，拷贝构造函数，转移构造函数，拷贝赋值函数，转移赋值函数，析构函数。

编译器会默认实现这个函数，但对于重要的构造函数和析构函数，应该用“=default”的形式告诉编译器和代码阅读者，使用默认的实现

class DemoClass final

{

public:

DemoClass() = default; // 明确告诉编译器，使用默认实现

~DemoClass() = default; // 明确告诉编译器，使用默认实现

};

“=detele”用于明确的禁用某个函数形式，可以用于任何函数。比如你想要禁止对象拷贝，可以显式地禁用拷贝构造和拷贝赋值函数。

class DemoClass final

{

public:

DemoClass(const DemoClass&) = delete; // 禁止拷贝构造

DemoClass& operator=(const DemoClass&) = delete; // 禁止拷贝赋值

};

因为C++中的隐式构造和隐式转型，当类中有单参数的构造函数，或者转型操作符函数，要使用“explicit”讲这些函数标记为“显式”。

class DemoClass final

{

public:

explicit DemoClass(const string\_type& str) // 显式单参构造函数

{ ... }

explicit operator bool() // 显式转型为bool

{ ... }

};

常用技巧：

1. 委托构造：当类中有多个构造函数时，将公共部分提取出来，减少重复代码

class DemoDelegating final

{

private:

int a; // 成员变量

public:

DemoDelegating(int x) : a(x) // 基本的构造函数

{}

DemoDelegating() : // 无参数的构造函数

DemoDelegating(0) // 给出默认值，委托给第一个构造函数

{}

DemoDelegating(const string& s) : // 字符串参数构造函数

DemoDelegating(stoi(s)) // 转换成整数，再委托给第一个构造函数

{}

};

1. 成员变量初始化：类中声明变量时直接初始化，这样不但简单清晰，也消除了隐患。

class DemoInit final // 有很多成员变量的类

{

private:

int a = 0; // 整数成员，赋值初始化

string s = "hello"; // 字符串成员，赋值初始化

vector<int> v{1, 2, 3}; // 容器成员，使用花括号的初始化列表

public:

DemoInit() = default; // 默认构造函数

~DemoInit() = default; // 默认析构函数

public:

DemoInit(int x) : a(x) {} // 可以单独初始化成员，其他用默认值

};

1. 类型别名：C++11中增加了using的用法，可以给类型起别名，它的格式与typedef相反。

using uint\_t = unsigned int; // using别名

typedef unsigned int uint\_t； // 等价的typedef

using的作用主要是简化名称长度和增加可读性，有的第三方库类型名字非常长。

class DemoClass final

{

public:

using this\_type = DemoClass; // 给自己也起个别名

using kafka\_conf\_type = KafkaConfig; // 外部类起别名

public:

using string\_type = std::string; // 字符串类型别名

using uint32\_type = uint32\_t; // 整数类型别名

using set\_type = std::set<int>; // 集合类型别名

using vector\_type = std::vector<std::string>;// 容器类型别名

private:

string\_type m\_name = "tom"; // 使用类型别名声明变量

uint32\_type m\_age = 23; // 使用类型别名声明变量

set\_type m\_books; // 使用类型别名声明变量

private:

kafka\_conf\_type m\_conf; // 使用类型别名声明变量

};

类型别名还有一个好处就是可以在将来维护时将现有类型换成其他类型，比如把字符串改为string\_view。这样只需要改类型别名就可以，无需动源代码。

小结：

1. “面向对象编程”是一种设计思想，要点是“封装”和抽象，“继承”和“多态”是衍生出的特性，不完全符合现实世界。
2. 在C++中要少用继承和虚函数，降低对象的成本，绕开难懂易错点坑。
3. 使用final标识符可以显式低禁用类被继承，简化类的层次。
4. 类有六大基本函数，对于重要的构造和析构函数，可用“=default”来显式地要求编译器使用默认实现。
5. “委托构造”和“成员变量初始化”可以让类的定义更简洁，使创建对象的工作更轻松。
6. 使用using和typedef来给类型起别名，既能够简化代码，又可以适应将来类型的变化。

内联函数：在函数调用处嵌入函数体的函数叫做内联函数，关键字是inline，作用是减少函数调用开销，适用于函数体简单的函数。

06|auto|decltype

自动类型推导：

int i = 0; // 整数变量，类型很容易知道

double x = 1.0; // 浮点数变量，类型很容易知道

std::string str = "hello"; // 字符串变量，有了名字空间，麻烦了一点

std::map<int, std::string> m = // 关联数组，名字空间加模板参数，很麻烦

{{1,"a"}, {2,"b"}}; // 使用初始化列表的形式

std::map<int, std::string>::const\_iterator // 内部子类型，超级麻烦

iter = m.begin();

？？？ = bind1st(std::less<int>(), 2); // 根本写不出来

除了简化代码，auto还避免了对类型的硬编码

auto和attribute一样，是编译阶段的特殊指令

认识auto：

auto只能用于初始化的场合，赋值初始化或者花括号初始化(既是初始化列表)。

但目前还不支持在类成员初始化时使用。

auto总是推导出值类型，而不是引用。

auto可以加上const，volatile，\*，&这类修饰符

auto x = 10L; // auto推导为long，x是long

auto& x1 = x; // auto推导为long，x1是long&

auto\* x2 = &x; // auto推导为long，x2是long\*

const auto& x3 = x; // auto推导为long，x3是const long&

auto x4 = &x3; // auto推导为const long\*，x4是const long\*

认识decltype：自带表达式的auto

int x = 0; // 整型变量

decltype(x) x1; // 推导为int，x1是int

decltype(x)& x2 = x; // 推导为int，x2是int&，引用必须赋值

decltype(x)\* x3; // 推导为int，x3是int\*

decltype(&x) x4; // 推导为int\*，x4是int\*

decltype(&x)\* x5; // 推导为int\*，x5是int\*\*

decltype(x2) x6 = x2; // 推导为int&，x6是int&，引用必须赋值

decltype 不仅能够推导出值类型，还能够推导出引用类型，也就是表达式的“原始类型”。

using int\_ptr = decltype(&x); // int \*

using int\_ref = decltype(x)&; // int &

Decltype的缺点是在变量初始化时，表达式要写两遍，左边的类型计算和右边的初始化，把简化代码的优势抵消了。在C++14中增加了“decltype(auto)”，即可以精确推导类型，又可以像auto一样简化代码。

使用auto/decltype：

auto写法简单，推导规则也好理解，声明变量时尽量多使用。

auto的一个最佳实践，就是range\_based for，不需要关系容器元素类型，迭代器返回值和首末位置。为了保证效率，最好用const auto&或者auto&

vector<int> v = {2,3,5,7,11}; // vector顺序容器

for(const auto& i : v) { // 常引用方式访问元素，避免拷贝代价

cout << i << ","; // 常引用不会改变元素的值

}

for(auto& i : v) { // 引用方式访问元素

i++; // 可以改变元素的值

cout << i << ",";

}

在C++14中，auto还支持返回函数的类型。

auto get\_a\_set() // auto作为函数返回值的占位符

{

std::set<int> s = {1,2,3};

return s;

}

decltype的使用：

适用于特殊类型，比如函数指针

// UNIX信号函数的原型，看着就让人晕，你能手写出函数指针吗？

void (\*signal(int signo, void (\*func)(int)))(int)

// 使用decltype可以轻松得到函数指针类型

using sig\_func\_ptr\_t = decltype(&signal) ;

再比如定义类的时候，由于auto被禁用了，可以使用decltype

class DemoClass final

{

public:

using set\_type = std::set<int>; // 集合类型别名

private:

set\_type m\_set; // 使用别名定义成员变量

// 使用decltype计算表达式的类型，定义别名

using iter\_type = decltype(m\_set.begin());

iter\_type m\_pos; // 类型别名定义成员变量

};

小结：

1. 自动类型推导是给编译器下的指令。
2. auto用于初始化时的类型推导，总是“值类型”，可以加上修饰符产生新的类型，它的推导规则比较简单，用法也简单，应该积极使用
3. Decltype使用类似函数调用的形式计算表达式的类型，能够用在任意场合。
4. Decltype可以精确返回表达式的类型，但写法较为麻烦，可以使用decltype(auto)来简化写法。
5. Auto和decltype不是硬编码，所用使用它们可以使代码更清晰，减少后期维护成本。

07|const/volatile/mutable:常量，变量，究竟是怎么回事

const和volatile

定义程序用的数字，字符串常量，代替宏定义

Const定义的常量在预处理时期并不存在，而是在运行阶段才会出现

Const定义的只读变量，使用指针获取地址，在强制写入也是可以改变值的，但绝对不提倡。

// 需要加上volatile修饰，运行时才能看到效果

const volatile int MAX\_LEN = 1024;

auto ptr = (int\*)(&MAX\_LEN);

\*ptr = 2048;

cout << MAX\_LEN << endl; // 输出2048

如果定义只读变量时不加volatile，那么修改是不会生效的，因为一般的常量是绝对不变的，编译器会做优化，在所有const常量出现的地方都替换成原始值

Volatile的作用是告诉编译器不对改const常量做优化，每次用到改常量都要到内存中去取。

最基本的const用法：

除了修饰变量，还有常量引用和常量指针

int x = 100;

const int& rx = x;

const int\* px = &x;

Const&被称为万能引用，在设计函数时，尽可能使用const&作为入参，一是保证效率，二是保证安全。

Const修饰指针时，const放在最左边，表示指针指向的只读变量，不允许修改

string name = "uncharted";

const string\* ps1 = &name; // 指向常量

\*ps1 = "spiderman"; // 错误，不允许修改

Const放在“\*”的右边时，表示指针不能被修改，既指针指向的地址不能变，但地址中的内容可以修改

string\* const ps2 = &name; // 指向变量，但指针本身不能被修改

\*ps2 = "spiderman"; // 正确，允许修改

与类相关的const用法：

class DemoClass final

{

private:

const long MAX\_SIZE = 256; // const成员变量

int m\_value; // 成员变量

public:

int get\_value() const // const成员函数

{

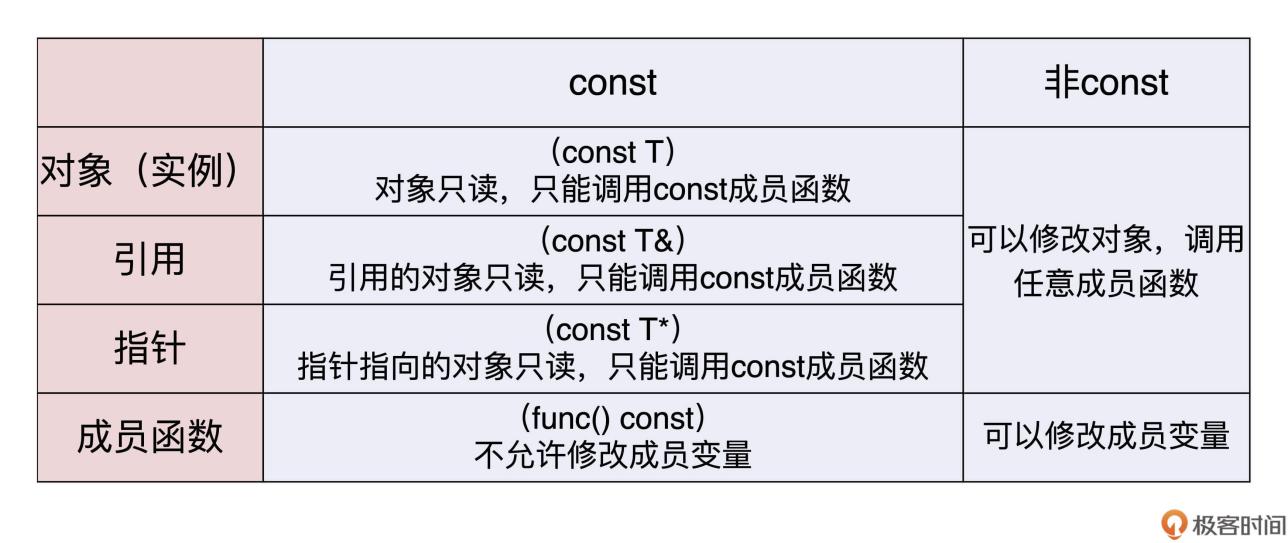
return m\_value;

}

};

Const放在函数后面，表示函数是一个只读操作，不会改变成员变量的值。

如果放在函数前面，说明返回值是const int类型的。



在标准库中，比如vector，它的empty()，size()，capacity()等查看基本属性的函数都是const的，而reserve()，clear()，erase()等都是非const的

关键字mutable：

mutable用来修饰类中的成员变量，表示变量即使在const对象里，也是可以修改的。

判断对象是否const应该是由外部可观测的特征来决定，对于外部观测不到的特征，可以允许使用mutable来修改，比如对象内部用到了一个mutex，一个缓冲区，或者一个原子变量用来做引用技术。

class DemoClass final

{

private:

mutable mutex\_type m\_mutex; // mutable成员变量

public:

void save\_data() const // const成员函数

{

// do someting with m\_mutex

}

};

Mutable和volatile一样要少用慎用。

小结：

1. const

它是一个类型修饰符，可以给任何对象附加上只读属性，保证安全

它可以修饰引用和指针，“const &”可以引用任何类型，是函数入口参数的最佳选择

它还可以修饰类成员函数，表示函数时只读的，const对象只能调用const成员函数

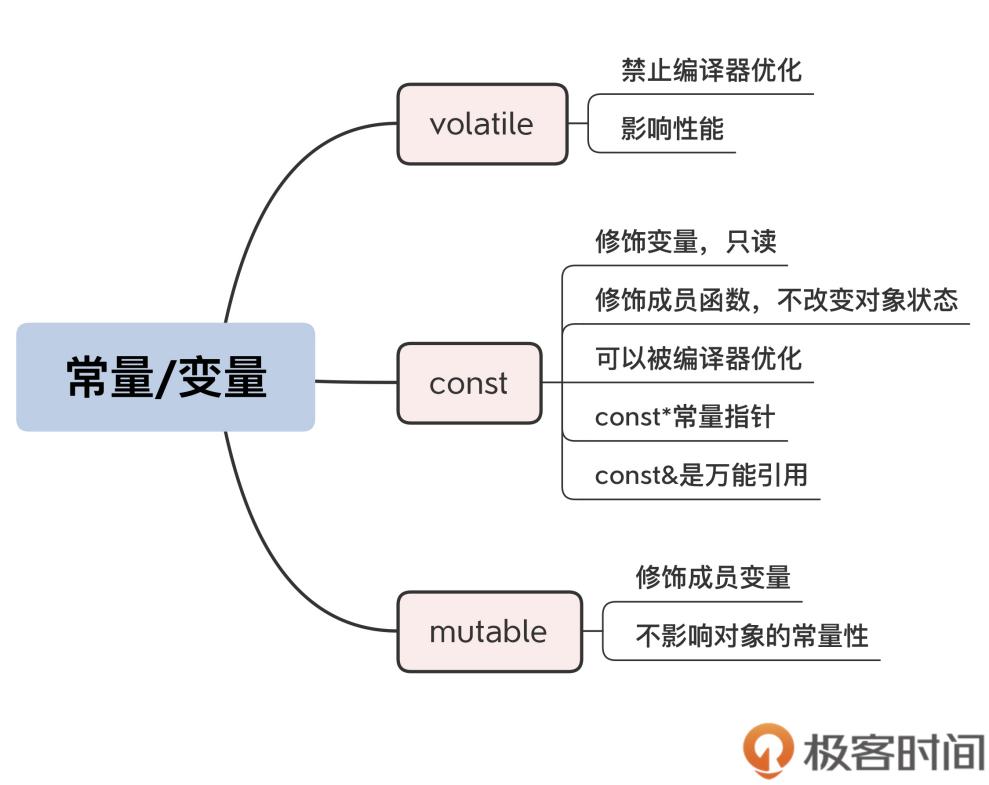
1. Volatile

它表示变量可能被不易察觉地修改，禁止编译器做优化，影响性能，要少用

1. Mutable

用来修饰类成员变量，允许const成员函数修改，mutable变量的变化不影响对象的常量性，但要小心不要误用损坏对象

要尽可能多地使用const，让代码更安全。在多线程编程时尤其有用，让编译器帮你检查所有的对象操作，把只读的属性持续传递出去，避免有害的副作用。



08|smart\_ptr：只能指针到底智能在哪里

什么是智能指针？

C++中的垃圾回收，构造/析构和RAII惯用法（对象代表资源，把管理资源的任务转化为管理对象的任务，将资源的获取和释放与对象的构造和析构对应起来）

应用到指针上，就是用类把裸指针包装起来，在构造函数里初始化，在析构函数里释放。

智能指针就是这样做的。

常用的只能指针：unique\_ptr和shared\_ptr

认识unique\_ptr：

在声明时要用模板参数指定类型

unique\_ptr<int> ptr1(new int(10)); // int智能指针

assert(\*ptr1 == 10); // 可以使用\*取内容

assert(ptr1 != nullptr); // 可以判断是否为空指针

unique\_ptr<string> ptr2(new string("hello")); // string智能指针

assert(\*ptr2 == "hello"); // 可以使用\*取内容

assert(ptr2->size() == 5); // 可以使用->调用成员函数

Unique\_ptr虽然叫指针，但实际上是一个对象，所以不能对它调用detele，他会自动管理初始化时的指针，在离开作用域时自动析构释放内存。

另外unique\_ptr也没有定义加减运算。

ptr1++; // 导致编译错误

ptr2 += 2; // 导致编译错误

Unique\_ptr必须要初始化

unique\_ptr<int> ptr3; // 未初始化智能指针

\*ptr3 = 42 ; // 错误！操作了空指针

为了避免不初始化的错误，可以调用工厂函数make\_unique()，强制创建只能指针的时候初始化。

auto ptr3 = make\_unique<int>(42); // 工厂函数创建智能指针

assert(ptr3 && \*ptr3 == 42);

auto ptr4 = make\_unique<string>("god of war"); // 工厂函数创建智能指针

assert(!ptr4->empty());

不过make\_unique要求C++14，如果使用的是C++11，可以自己实现一个简化版make\_unique()

template<class T, class... Args> // 可变参数模板

std::unique\_ptr<T> // 返回智能指针

my\_make\_unique(Args&&... args) // 可变参数模板的入口参数

{

return std::unique\_ptr<T>( // 构造智能指针

new T(std::forward<Args>(args)...)); // 完美转发

}

Unique\_ptr的所有权：

在向另一个智能指针赋值时，要用move显式的声明所有权转移，赋值之后原有的unique\_ptr变成了空指针。尽量不要对指针进行赋值操作

auto ptr1 = make\_unique<int>(42); // 工厂函数创建智能指针

assert(ptr1 && \*ptr1 == 42); // 此时智能指针有效

auto ptr2 = std::move(ptr1); // 使用move()转移所有权

assert(!ptr1 && ptr2); // ptr1变成了空指针

认识shared\_ptr:

shared\_ptr<int> ptr1(new int(10)); // int智能指针

assert(\*ptr1 == 10); // 可以使用\*取内容

shared\_ptr<string> ptr2(new string("hello")); // string智能指针

assert(\*ptr2 == "hello"); // 可以使用\*取内容

auto ptr3 = make\_shared<int>(42); // 工厂函数创建智能指针

assert(ptr3 && \*ptr3 == 42); // 可以判断是否为空指针

auto ptr4 = make\_shared<string>("zelda"); // 工厂函数创建智能指针

assert(!ptr4->empty()); // 可以使用->调用成员函数

Shared\_ptr跟unique\_ptr的最大不同点就在于：它的所有权是可以被安全共享的。

auto ptr1 = make\_shared<int>(42); // 工厂函数创建智能指针

assert(ptr1 && ptr1.unique() ); // 此时智能指针有效且唯一

auto ptr2 = ptr1; // 直接拷贝赋值，不需要使用move()

assert(ptr1 && ptr2); // 此时两个智能指针均有效

assert(ptr1 == ptr2); // shared\_ptr可以直接比较

// 两个智能指针均不唯一，且引用计数为2

assert(!ptr1.unique() && ptr1.use\_count() == 2);

assert(!ptr2.unique() && ptr2.use\_count() == 2);

Shared\_ptr支持安全共享的秘密在于使用了 引用计数，当发生拷贝赋值的时候，引用计数就+1，当发生析构销毁时，引用计数就-1，只有当引用计数为0时，才会真正的调用delete释放内存。

Shared\_ptr可以在任何场合代替原始指针，而不用担心资源回收。

Shared\_ptr的注意事项：

引用计数的存储和管理都是成本，性能略差。

Shared\_ptr的销毁动作，析构函数中不要有非常复杂，严重阻塞的操作。

class DemoShared final // 危险的类，不定时的地雷

{

public:

DemoShared() = default;

~DemoShared() // 复杂的操作会导致shared\_ptr析构时世界静止

{

// Stop The World ...

}

};

循环引用问题，这在把shared\_ptr做为类成员的时候最容易出问题，典型例子就是链表节点。

class Node final

{

public:

using this\_type = Node;

using shared\_type = std::shared\_ptr<this\_type>;

public:

shared\_type next; // 使用智能指针来指向下一个节点

};

auto n1 = make\_shared<Node>(); // 工厂函数创建智能指针

auto n2 = make\_shared<Node>(); // 工厂函数创建智能指针

assert(n1.use\_count() == 1); // 引用计数为1

assert(n2.use\_count() == 1);

n1->next = n2; // 两个节点互指，形成了循环引用

n2->next = n1;

assert(n1.use\_count() == 2); // 引用计数为2

assert(n2.use\_count() == 2); // 无法减到0，无法销毁，导致内存泄漏

可以用weak\_ptr来杜绝这种情况，Weak\_ptr没有重载\*和->，无法直接访问对象，但可以通过.expired()函数判断指向的对象是否存在。weak\_ptr不会增加引用计数，但在有需要的时候，可以调用lock()函数来获取shard\_ptr。

class Node final

{

public:

using this\_type = Node;

// 注意这里，别名改用weak\_ptr

using shared\_type = std::weak\_ptr<this\_type>;

public:

shared\_type next; // 因为用了别名，所以代码不需要改动

};

auto n1 = make\_shared<Node>(); // 工厂函数创建智能指针

auto n2 = make\_shared<Node>(); // 工厂函数创建智能指针

n1->next = n2; // 两个节点互指，形成了循环引用

n2->next = n1;

assert(n1.use\_count() == 1); // 因为使用了weak\_ptr，引用计数为1

assert(n2.use\_count() == 1); // 打破循环引用，不会导致内存泄漏

if (!n1->next.expired()) { // 检查指针是否有效

auto ptr = n1->next.lock(); // lock()获取shared\_ptr

assert(ptr == n2);

}

小结：

1. 智能指针是代理模式的具体应用，它使用的RAII技术代理了裸指针，能够自动释放内存，无需程序员干预。
2. 如果指针是独占的，那么使用unique\_ptr，它为裸指针添加了很多限制，更加安全。
3. 如果指针是共享的，那边使用shared\_ptr，它的功能非常完善，很多场景下可以代替裸指针。
4. 应当使用工厂函数make\_unique和make\_shared来创建只能指针，强制初始化，还能使用auto来简化声明。
5. Shared\_ptr有少量的管理成本，还有可能引发一些难以排查的错误，所以不要过度使用。

既然已经理解了智能指针，就尽量不要再使用裸指针，new和delete来操作内存了。

09|exception:怎样才能用好异常

为什么要有异常：

异常只是C++为了处理错误而提出的一种解决方案，当然也不会是唯一的一种

在以前，程序员处理异常的基本手段是错误码

int n = read\_data(fd, ...); // 读取数据

if (n == 0) {

... // 返回值不太对，适当处理

}

if (errno == EAGAIN) {

... // 适当处理错误

}

但这种做法导致错误处理和正常的业务处理混在一起，看起来很乱，并且错误码是可以被忽略掉的，可能导致严重的安全隐患。

异常就是针对错误码的缺陷而设计的，它有三大特点：

1. 异常的处理流程是独立的
2. 异常是不可以被忽略的
3. 可以处理错误码无法使用的场合

异常的用法和使用方式：

用try把可能出现异常的代码包起来，然后编写catch块捕获异常并处理

try

{

int n = read\_data(fd, ...); // 读取数据，可能抛出异常

... // do some right thing

}

catch(...)

{

... // 集中处理各种错误情况

}

C++允许编写多个catch块，但异常只会按照catch块的顺序依次匹配，而不会去找最佳匹配。

最好只写一个catch块代码。

Function\_try：

void some\_function()

try // 函数名之后直接写try块

{

...

}

catch(...) // catch块与函数体同级并列

{

...

}

这样可以捕获函数执行过程中所有可能产生的异常。

频繁异常的抛出和处理会对性能有一定的影响，谨慎使用异常，几个应当使用异常的场景：

1. 不允许忽略的错误
2. 极少数情况下才会发生的错误
3. 严重影响到正常流程，很难恢复到正常状态的错误
4. 无法本地处理，必须穿透调用栈，传递到上层才能被处理的错误

例子：比如构造函数如果内部初始化失败，那后面的逻辑也就进行不下去了，所以可以用异常处理；比如读写文件，通常文件系统很少会出错，总会成功，也适合用异常处理；再比如socket通信，由于网络链路的不确定因素很多，收发数据失败是常见情况，为了性能考虑吗，检查错误码重试比较好。

保证不抛出异常：

Noexcept，用来修饰函数，告诉编译器这个函数不会抛出异常（实际上是不想处理异常），好处是编译器会对改函数做优化。

void func\_noexcept() noexcept // 声明绝不会抛出异常

{

cout << "noexcept" << endl;

}

但noexcepet修饰的函数也有可能抛出异常，此时程序会直接崩溃。

小结:

1. 异常是针对错误码的缺陷而设计的，它不能被忽略，而且可以穿透调用层，被传到其他地方去
2. 使用try\_catch机制，可以分离错误处理和正常流程，使得代码更清晰
3. Throw可以排出任何类型作为异常，但最好使用C++标准里面的exception类
4. 完全用或者不用异常都是不可取的，合理分析，适度使用，降低异常的成本
5. Noexcept标记函数不抛出异常，可以让编译器做更好的优化

10|lambda：函数式编程带来了什么

C++五大编程范式之一，函数式编程

C++函数的特殊性：不存在对应类型的变量，只能用函数指针去操作

void my\_square(int x) // 定义一个函数

{

cout << x\*x << endl; // 函数的具体内容

}

auto pfunc = &my\_square; // 只能用指针去操作函数，指针不是函数

(\*pfunc)(3); // 可以用\*访问函数

pfunc(3); // 也可以直接调用函数指针

认识lambda：

auto func = [](int x) // 定义一个lambda表达式

{

cout << x\*x << endl; // lambda表达式的具体内容

};

func(3); // 调用lambda表达式

因为lambda函数是一个变量，所以可以随时随地定义它，从而限制lambda函数的生命周期和作用域，实现函数的局部化。

lambda函数可以捕捉外部的变量，在函数里直接操作

int n = 10; // 一个外部变量

auto func = [=](int x) // lambda表达式，用“=”值捕获

{

cout << x\*n << endl; // 直接操作外部变量

};

func(3); // 调用lambda表达式

这就是大名鼎鼎的闭包，一块活的代码块，因为它保存了定义时捕捉的外部变量，就可以跳离定义点，到别的地方去执行，比如在另一个lambda函数内部，而仅凭入口参数是无法做到这一点的。

auto a = [](int x) // a函数执行一个功能

{...}

auto b = [](double x) // b函数执行一个功能

{...}

auto c = [](string str) // c函数执行一个功能

{...}

auto f = [](...) // f函数执行一个功能

{...}

return f(a, b, c) // f调用a/b/c运算得到结果

使用lambda的注意事项，

1. lambda的形式

auto f1 = [](){}; // 相当于空函数，什么也不做

实际开发中的lambda函数肯定没那么简单，函数体里会有很多语句，所以一定会要注意缩进。

auto f2 = []() // 定义一个lambda表达式

{

cout << "lambda f2" << endl;

auto f3 = [](int x) // 嵌套定义lambda表达式

{

return x\*x;

};// lambda f3 // 使用注释显式说明表达式结束

cout << f3(10) << endl;

}; // lambda f2 // 使用注释显式说明表达式结束

在C++中鼓励匿名使用lambda表达式，既使用auto类型定义函数返回类型。

vector<int> v = {3, 1, 8, 5, 0}; // 标准容器

cout << \*find\_if(begin(v), end(v), // 标准库里的查找算法

[](int x) // 匿名lambda表达式，不需要auto赋值

{

return x >= 5; // 用做算法的谓词判断条件

} // lambda表达式结束

)

<< endl; // 语句执行完，lambda表达式就不存在了

1. lambda的变量捕获

int x = 33; // 一个外部变量

auto f1 = [=]() // lambda表达式，用“=”按值捕获

{

//x += 10; // x只读，不允许修改

};

auto f2 = [&]() // lambda表达式，用“&”按引用捕获

{

x += 10; // x是引用，可以修改

};

auto f3 = [=, &x]() // lambda表达式，用“&”按引用捕获x，其他的按值捕获

{

x += 20; // x是引用，可以修改

};

[=]是值捕获，[&]是引用捕获，可以显式地指定变量时按值捕获还是按引用捕获

class DemoLambda final

{

private:

int x = 0;

public:

auto print() // 返回一个lambda表达式供外部使用

{

return [this]() // 显式捕获this指针

{

cout << "member = " << x << endl;

};

}

};

1. 泛型的lambda

在C++14中，lambda表达式支持参数泛型化

auto f = [](const auto& x) // 参数使用auto声明，泛型化

{

return x + x;

};

cout << f(3) << endl; // 参数类型是int

cout << f(0.618) << endl; // 参数类型是double

string str = "matrix";

cout << f(str) << endl; // 参数类型是string

小结：

1. lambda表达式相当于闭包，能够像函数一样被调用，像变量一个被传递
2. 可以使用auto类型自动推导存储lambda表达式的值，但C++鼓励就地匿名使用，缩小作用域
3. Lambda捕获外部变量时，[=]表示值捕获，[&]表示引用捕获，空的[]则表示不捕获
4. 捕获引用时要注意外部变量的生命周期，防止变量失效
5. C++14里可以是使用泛型的lambda表达式 ，相当于简易版的模板函数

滥用lambda表达式也会导致难以阅读的代码，如调用层次过深。

11|一枝独秀的字符串：C++也能处理文本？

C++主要包括语言特性和标准库，标准库十分强大，字符串是其中最常用的。

认识字符串：

String是模板类basic\_string的特化形式，是一个typedf。

using string = std::basic\_string<char>; // string其实是一个类型别名

String对应的unicode中的UTF-8，都是单字节。

当涉及unicode，编码转化的时候，尽量不要使用C++，编码方式指的是将字符转化为二进制的方法，每个字符都在编码表中都有一个特定的索引。

用好字符串：

String是一个功能比较齐全的字符串类，可以提取子串，比较大小，检查长度，搜索字符等。

string str = "abc";

assert(str.length() == 3);

assert(str < "xyz");

assert(str.substr(0, 1) == "a");

assert(str[1] == 'b');

assert(str.find("1") == string::npos);

assert(str + "d" == "abcd");

字符串并不是一种容器，每一个字符串都是一个不可变的实体，在某些情况下，使用vector<char>更好一点。

以下是使用string的一些小技巧：

1. 字面量后缀

C++14中新增了一个字面量后缀‘s’，明确地表示它是字符串类型，这就可以使用auto来自动推导。

using namespace std::literals::string\_literals; //必须打开名字空间

auto str = "std string"s; // 后缀s，表示是标准字符串，直接类型推导

assert("time"s.size() == 4); // 标准字符串可以直接调用成员函数

1. 原始字符串

C++11增加了原始字符串的新表示形式，比原来多了大写字母R和圆括号

auto str = R"(nier:automata)"; // 原始字符串：nier:automata

这样可以不对字符串中的\进行转义，如果字符串里面本身就有引号加括号的话，可以加界定符

auto str5 = R"==(R"(xxx)")==";// 原样输出：R"(xxx)"

1. 字符串转换函数

assert(stoi("42") == 42); // 字符串转整数

assert(stol("253") == 253L); // 字符串转长整数

assert(stod("2.0") == 2.0); // 字符串转浮点数

assert(to\_string(1984) == "1984"); // 整数转字符串

1. 字符串视图类

String的成本比较大，在C++17里面新增了string\_view，内部只保留了一个指针和长度，无论是拷贝还是修改都非常廉价方便。可以在C++11中实现简单的版本

class my\_string\_view final // 简单的字符串视图类，示范实现

{

public:

using this\_type = my\_string\_view; // 各种内部类型定义

using string\_type = std::string;

using string\_ref\_type = const std::string&;

using char\_ptr\_type = const char\*;

using size\_type = size\_t;

private:

char\_ptr\_type ptr = nullptr; // 字符串指针

size\_type len = 0; // 字符串长度

public:

my\_string\_view() = default;

~my\_string\_view() = default;

my\_string\_view(string\_ref\_type str) noexcept

: ptr(str.data()), len(str.length())

{}

public:

char\_ptr\_type data() const // 常函数，返回字符串指针

{

return ptr;

}

size\_type size() const // 常函数，返回字符串长度

{

return len;

}

};

正则表达式：

String只解决了如何存储和表示文本 ，如果要做大小写转换，判断前缀，模式匹配查找等更加复杂的处理，就需要用到正则表达式。

C++11中加入了正则表达式库regex，可以任意操作文本，字符串。

C++的正则表达式主要有两个类：

Regex：表示一个正则表达式

Smatch：表示正则表达式的匹配结果

C++正则匹配有三个算法，都是只读的，不会修改原字符串

Regex\_match：完全匹配一个字符串

Regex\_search：在字符串中查找一个正则匹配

Regex\_replace：正则查找再做替换

例子：

auto make\_regex = [](const auto& txt) // 生产正则表达式

{

return std::regex(txt);

};

auto make\_match = []() // 生产正则匹配结果

{

return std::smatch();

};

auto str = "neir:automata"s; // 待匹配的字符串

auto reg =

make\_regex(R"(^(\w+)\:(\w+)$)"); // 原始字符串定义正则表达式

auto what = make\_match(); // 准备获取匹配的结果

如果匹配成功，结果存储在what里，可以像容器那样去访问，0号元素是整个匹配串，其他的是子表达式匹配串。

assert(regex\_match(str, what, reg)); // 正则匹配

for(const auto& x : what) { // for遍历匹配的子表达式

cout << x << ',';

}

Regex\_search和regex\_replace的用法差不多：

auto str = "god of war"s; // 待匹配的字符串

auto reg =

make\_regex(R"((\w+)\s(\w+))"); // 原始字符串定义正则表达式

auto what = make\_match(); // 准备获取匹配的结果

auto found = regex\_search( // 正则查找，和匹配类似

str, what, reg);

assert(found); // 断言找到匹配

assert(!what.empty()); // 断言有匹配结果

assert(what[1] == "god"); // 看第一个子表达式

assert(what[2] == "of"); // 看第二个子表达式

auto new\_str = regex\_replace( // 正则替换，返回新字符串

str, // 原字符串不改动

make\_regex(R"(\w+$)"), // 就地生成正则表达式对象

"peace" // 需要指定替换的文字

);

cout << new\_str << endl; // 输出god of peace

尽量不要反复创建正则对象，能重用的就重用。

小结：

1. C++支持多种字符类型，常用的string其实是basic\_string的特化形式。
2. C++目前对unicode的支持还不太完善，尽量避开国际化和编码转化。
3. 应该把string当成一个完整字符串来操作，不要当作容器来使用。
4. 字面量后缀’s’可以用来标记string类型，配合auto使用。
5. 原始字符串不会转义，是字符串的原始形态，适合在代码里写复杂的文本，比如正则规则。
6. 处理文本可以使用regex，它的功能非常强大，但需要花一些时间和精力。

12|容器

容器就是存放元素的一些数据结构

算法+数据结构=程序

认识容器：

容器的通用特性：

容器里存储的是元素的拷贝，副本，而非引用，容器操作元素很大的一块开销就是值的拷贝，

一个解决办法就是尽量为元素实现转移赋值和转移构造，在加入容器时用std::move()来转移。

Point p; // 一个拷贝成本很高的对象

v.push\_back(p); // 存储对象，拷贝构造，成本很高

v.push\_back(std::move(p)); // 定义转移构造后就可以转移存储，降低成本

也可以使用emplace()函数，他可以就地构造元素，免去了构造再拷贝的成本。

v.emplace\_back(...); // 直接在容器里构造元素，不需要拷贝或者转移

容器的具体特性：

按照元素的访问方式，可以分为顺序容器，有序容器，无序容器。

顺序容器就是数据结构里面的向量表，一共有5中，vector，list，array，deque，forward\_list。

按照存储结构又可以细分为连续存储的数组：array，vector，deque；以及指针结构的链表：list，forward\_list。

Array和vector对应了C的内置数组，内存布局与C完全相同，所以开销最小，速度最快。

它俩的区别在于容量是否动态增长。

array<int, 2> arr; // 初始一个array，长度是2

assert(arr.size() == 2); // 静态数组的长度总是2

vector<int> v(2); // 初始一个vector，长度是2

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

v.emplace\_back(i); // 追加多个元素

}

assert(v.size() == 12); // 长度动态增长到12

Deque也是动态增长的数组，但他可以在头尾两端高效地插入删除元素。

deque<int> d; // 初始化一个deque，长度是0

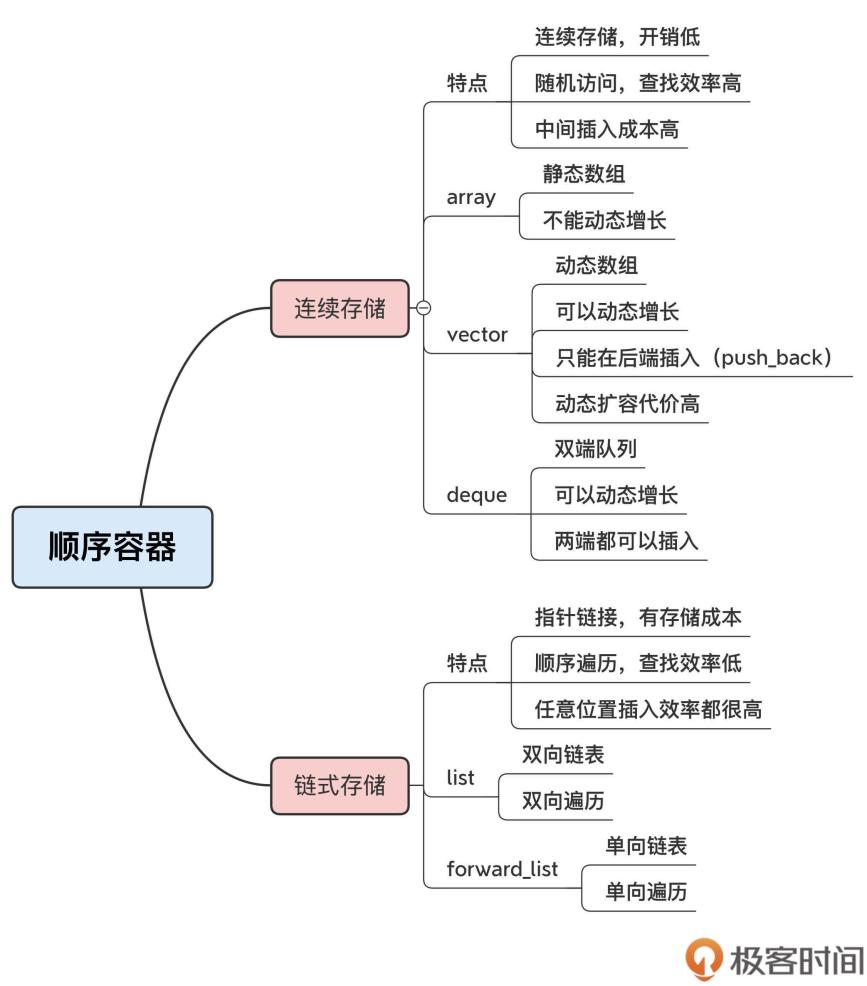
d.emplace\_back(9); // 末端添加一个元素

d.emplace\_front(1); // 前端添加一个元素

assert(d.size() == 2); // 长度动态增长到2

Vector和deque元素是连续存储的，所以在中间插入删除的效率很低，而list和forward\_list是链表结构，插入删除只需要调整指针，所以在任意位置的操作都很高效。

链表的缺点是查找效率低，并且存储成本较高，因为需要多加一个或者两个指针，list是双向链表，forward\_list是单向链表。



有序容器：

顺序容器元素的次序是插入次序排列的，而有序容器元素是按照某种规则排列的。

C++的有序容器使用的是树结构，通常是红黑树--有着最好查找性能的二叉树。

标准库里一共有四种有序容器，set/multiset和map/multimap，有multi前缀的表示可以容纳重复的key。

容器是如何判断两个元素的先后顺序。

在定义容器时，必须指定key的比较函数。

template<

class T // 模板参数只有一个元素类型

> class vector; // vector

template<

class Key, // 模板参数是key类型，即元素类型

class Compare = std::less<Key> // 比较函数

> class set; // 集合

template<

class Key, // 第一个模板参数是key类型

class T, // 第二个模板参数是元素类型

class Compare = std::less<Key> // 比较函数

> class map; // 关联数组

C++中的int，string等基本类型都可以比较大小，作为key没什么问题，但很多自定义的类型没有默认的比较函数，要作为key就比较麻烦。解决这个问题有两种办法：一个是重载“<”

bool operator<(const Point& a, const Point& b)

{

return a.x < b.x; // 自定义比较运算

}

set<Point> s; // 现在就可以正确地放入有序容器

s.emplace(7);

s.emplace(3);

另一个是自定义模板参数

set<int> s = {7, 3, 9}; // 定义集合并初始化3个元素

for(auto& x : s) { // 范围循环输出元素

cout << x << ","; // 从小到大排序，3,7,9

}

auto comp = [](auto a, auto b) // 定义一个lambda，用来比较大小

{

return a > b; // 定义大于关系

};

set<int, decltype(comp)> gs(comp) // 使用decltype得到lambda的类型

std::copy(begin(s), end(s), // 拷贝算法，拷贝数据

inserter(gs, gs.end())); // 使用插入迭代器

for(auto& x : gs) { // 范围循环输出元素

cout << x << ","; // 从大到小排序，9,7,3

}

除了比较函数这一点，其它的没啥好说的，集合关系用set（既单个元素），关联数组用map。

有序容器在插入的时候有成本，当数据量大的时候，内部的查找，树旋转的成本比较高。

如果需要实时插入排序，用set、map没什么问题，如果是非实时，最好还是用vector，插入完成后在一次性排序，效果会更好。

无序容器：

无序容器对应有序容器，也有四种：unordered\_set/unordered\_multiset、unordered\_map/unordered\_multimap

无序容器在用法上和有序容器相似，但内部数据结构是hash表。

元素的位置取决于计算的散列值，没有明显的规律可言，可以理解为乱序容器。

using map\_type = // 类型别名

unordered\_map<int, string>; // 使用无序关联数组

map\_type dict; // 定义一个无序关联数组

dict[1] = "one"; // 添加三个元素

dict.emplace(2, "two");

dict[10] = "ten";

for(auto& x : dict) { // 遍历输出

cout << x.first << "=>" // 顺序不确定

<< x.second << ","; // 既不是插入顺序，也不是大小序

}

无序容器对于key值的要求更加严格，以unordered\_map的声明为例：

template<

class Key, // 第一个模板参数是key类型

class T, // 第二个模板参数是元素类型

class Hash = std::hash<Key>, // 计算散列值的函数对象

class KeyEqual = std::equal\_to<Key> // 相等比较函数

> class unordered\_map;

它要求key具备两个条件：一是可以计算哈希值，二是可以比较大小。第一个是因为散列表的要求，只有计算哈希值才可以放入散列表，第二个则是因为hash值可能冲突，当hash值冲突的时候，需要比较key值。

与有序容器一样，要把自定义的类型作为key值，需要实现这两个函数。

“==”比较简单，可以通过重载操作符来实现：

bool operator==(const Point& a, const Point& b)

{

return a.x == b.x; // 自定义相等比较运算

}

散列函数就比较麻烦，可以使用lambda函数实现，最好只使用标准库中的std::hash函数对象，而不要自己计算，否则很容易造成hash冲突。

auto hasher = [](const auto& p) // 定义一个lambda表达式

{

return std::hash<int>()(p.x); // 调用标准hash函数对象计算

};

有了哈希函数和比较函数，自定义类型也可以放进无序容器了：

unordered\_set<Point, decltype(hasher)> s(10, hasher);

s.emplace(7);

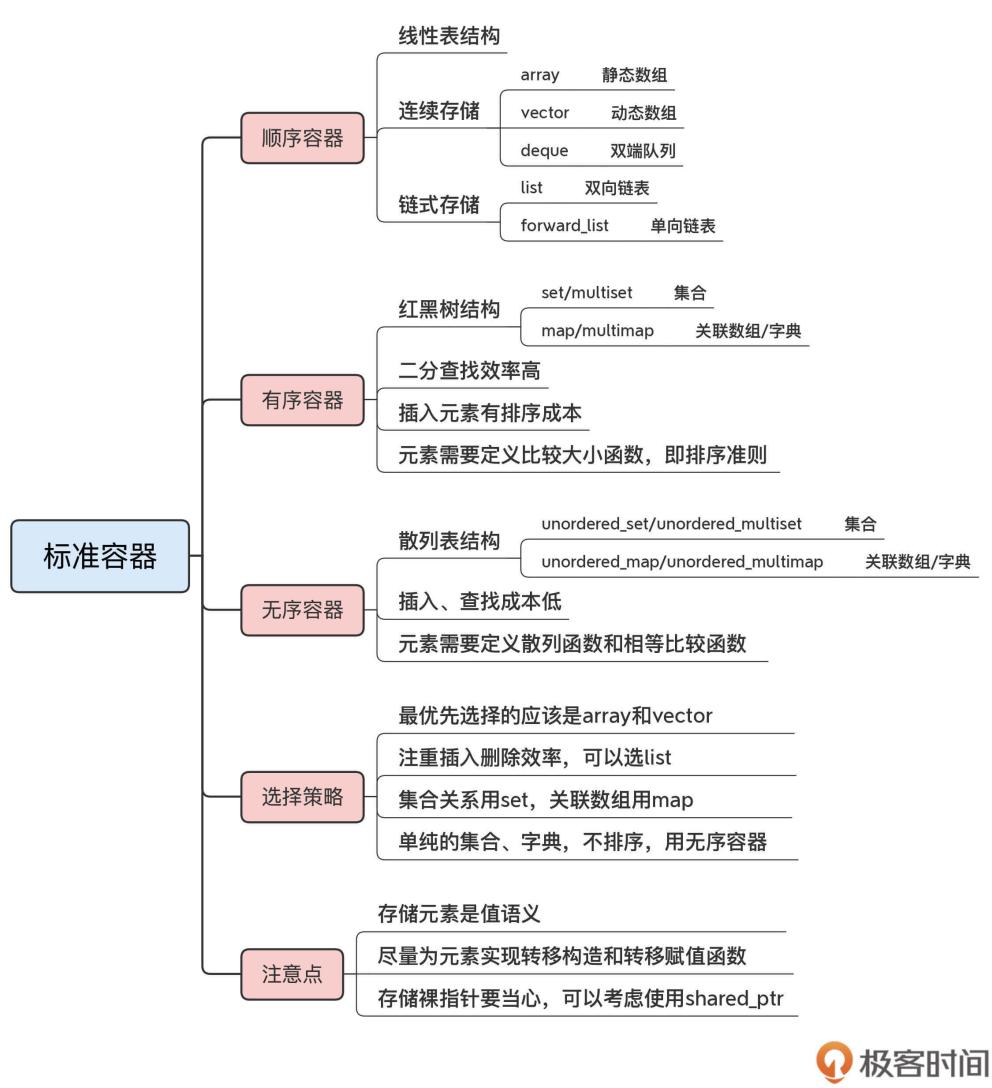
s.emplace(3);

如果没有排序需求，就应该选择无序容器，没有排序成本，它的速度非常快。

小结：

判断容器是否合适的基本依据是不要有多余的操作，比如只在末尾添加元素，就不用deque和list；如果不需要排序，只想要快读的查找元素，就应该选unoderered\_set。

1. 容器可以分为三类，顺序容器，有序容器和无序容器
2. 所有容器中优先选用vector和array，他们的速度最快，开销最低
3. List是链表结构，插入和删除快，但查找效率低
4. 有序容器是红黑树结构，查找快，但有插入成本
5. 无序容器是hash表结构，查找和插入都快
6. 有序容器和无序容器都属于关联容器，元素有key的概念，操作元素实际上都在操作key，所以要定义对key的比较函数或者散列函数



13|五花八门的算法

认识算法：从理论上说，算法是一系列定义明确的操作步骤，能在有限次的运算后得到结果。

计算机科学里的算法有很多，排序，查找，遍历，加密等，但在C++中，算法的含义要狭窄得多。

C++中的算法指的是工作在容器上的一些泛型函数，会对容器中的元素实施的各种操作。

C++标准库提供了上百中算法，比如：

Remove():移除某个特定值

Sort()：快速排除

Binary\_search:执行二分查找

Make\_heap:构造一个堆结构

不过算法并不神秘，所有的算法本质都是for或者while遍历元素来逐个处理的。

比如count算法，用来统计某个元素出现的次数

vector<int> v = {1,3,1,7,5}; // vector容器

auto n1 = std::count( // count算法计算元素的数量

begin(v), end(v), 1 // begin()、end()获取容器的范围

);

int n2 = 0;

for(auto x : v) { // 手写for循环

if (x == 1) { // 判断条件，然后统计

n2++;

}

}

算法封装了需要手写的for循环。

在以前，很多标准库算法需要传入一个函数对象，写起来很麻烦，但现在可以用lambda表达式来简化形式。

auto n = std::count\_if( // count\_if算法计算元素的数量

begin(v), end(v), // begin()、end()获取容器的范围

[](auto x) { // 定义一个lambda表达式

return x > 2; // 判断条件

}

); // 大函数里面套了三个小函数

认识迭代器：

迭代器相当于算法的手脚，算法操作容器，其实传入的是容器起始和末尾位置的迭代器，算法只能通过迭代器间接的访问容器以及元素。

这种间接的好处是，分离了数据和操作，算法不关心容器内部的数据结构，以统一的方式去操作元素，这就是泛型编程的理念。

有些通用算法对某些数据结构的效率比较低，为此容器提供了专门的替代成员函数。

迭代器可以看做是一种智能指针，但它强调的是对数据的访问，而非生命周期管理。

容器一般都会提供degin(),end()成员函数，调用它们就可以获得两端的迭代器

vector<int> v = {1,2,3,4,5}; // vector容器

auto iter1 = v.begin(); // 成员函数获取迭代器，自动类型推导

auto iter2 = v.end();

或者可以使用全局函数，另外还有cbegin(),cend()函数，返回常量迭代器

auto iter3 = std::begin(v); // 全局函数获取迭代器，自动类型推导

auto iter4 = std::end(v);

迭代器和指针类似，可以前进或者后退，但不一定支持“++”，“--”操作符，一般使用函数来操作

Distance()，计算两个迭代器之间的距离

Advance()，前进或者后退N步

Next()/perv()，计算迭代器前后的某个位置

例子：

array<int, 5> arr = {0,1,2,3,4}; // array静态数组容器

auto b = begin(arr); // 全局函数获取迭代器，首端

auto e = end(arr); // 全局函数获取迭代器，末端

assert(distance(b, e) == 5); // 迭代器的距离

auto p = next(b); // 获取“下一个”位置

assert(distance(b, p) == 1); // 迭代器的距离

assert(distance(p, b) == -1); // 反向计算迭代器的距离

advance(p, 2); // 迭代器前进两个位置，指向元素'3'

assert(\*p == 3);

assert(p == prev(e, 2)); // 是末端迭代器的前两个位置

最有用的算法：

手写循环的替代品：for\_each。

vector<int> v = {3,5,1,7,10}; // vector容器

for(const auto& x : v) { // range for循环

cout << x << ",";

}

auto print = [](const auto& x) // 定义一个lambda表达式

{

cout << x << ",";

};

for\_each(cbegin(v), cend(v), print);// for\_each算法

for\_each( // for\_each算法，内部定义lambda表达式

cbegin(v), cend(v), // 获取常量迭代器

[](const auto& x) // 匿名lambda表达式

{

cout << x << ",";

}

);

for\_each的价值体现在：将遍历元素和操作元素分成了两部分，也就是两个函数，一个遍历容器，一个操作元素，而且名字的含义也更加地准确，代码有更好的封装。

排序算法：

标准库中的sort使用的是快速排序

auto print = [](const auto& x) // lambda表达式输出元素

{

cout << x << ",";

};

std::sort(begin(v), end(v)); // 快速排序

for\_each(cbegin(v), cend(v), print); // for\_each算法

快排虽然快，但它是不稳定的，并且是全排所有元素。但我们只想要Top N，中位数，最大最小值时这样的部分元素时，全排是一种浪费。

C++为此准备了很多种算法：

1. 要求稳定排序，用stable\_sort
2. TopN用partial\_sort
3. 选出前几名，但不需要排序（BestN），用nth\_element
4. 中位数，百分位数也用nth\_element
5. 按照某种规则将元素分为两组，用partition
6. 第一名和最后一名用minmax\_element

代码示例：

// top3

std::partial\_sort(

begin(v), next(begin(v), 3), end(v)); // 取前3名

// best3

std::nth\_element(

begin(v), next(begin(v), 3), end(v)); // 最好的3个

// Median

auto mid\_iter = // 中位数的位置

next(begin(v), v.size()/2);

std::nth\_element( begin(v), mid\_iter, end(v));// 排序得到中位数

cout << "median is " << \*mid\_iter << endl;

// partition

auto pos = std::partition( // 找出所有大于9的数

begin(v), end(v),

[](const auto& x) // 定义一个lambda表达式

{

return x > 9;

}

);

for\_each(begin(v), pos, print); // 输出分组后的数据

// min/max

auto value = std::minmax\_element( //找出第一名和倒数第一

cbegin(v), cend(v)

);

这些算法通常是随机访问迭代器，所以最好在顺序容器vector/array上使用。

如果是list，应该使用sort()，她对链表做了特别优化。有序容器本身就是排好序的，直接对迭代器做运算即可，无序容器则无法调用排序算法，因为散列表元素无法交换位置。

查找算法：

排序算法是为了使元素有序，方便快速查找。

算法binary\_search只返回bool值，告知元素是否存在，而更多的时候，我们是想定位元素。

vector<int> v = {3,5,1,7,10,99,42}; // vector容器

std::sort(begin(v), end(v)); // 快速排序

auto found = binary\_search( // 二分查找，只能确定元素在不在

cbegin(v), cend(v), 7

);

想要定位元素的位置，需要用到lower\_bound，它返回第一个大于或者等于值的位置

decltype(cend(v)) pos; // 声明一个迭代器，使用decltype

pos = std::lower\_bound( // 找到第一个>=7的位置

cbegin(v), cend(v), 7

);

found = (pos != cend(v)) && (\*pos == 7); // 可能找不到，所以必须要判断

assert(found); // 7在容器里

pos = std::lower\_bound( // 找到第一个>=9的位置

cbegin(v), cend(v), 9

);

found = (pos != cend(v)) && (\*pos == 9); // 可能找不到，所以必须要判断

assert(!found); // 9不在容器里

upper\_bound返回第一个大于值的元素

pos = std::upper\_bound( // 找到第一个>9的位置

cbegin(v), cend(v), 9

);

lower\_bound和upper\_bound组成的区间可以表示为

begin < x <= lower\_bound < upper\_bound < end

实际上这个区间中最多有两个不相同的值。

对于有序容器set/map，有等价的成员函数find/lower\_bound/upper\_boun，其中find与binary\_search不同，它的返回值是迭代器而非bool。

multiset<int> s = {3,5,1,7,7,7,10,99,42}; // multiset，允许重复

auto pos = s.find(7); // 二分查找，返回迭代器

assert(pos != s.end()); // 与end()比较才能知道是否找到

auto lower\_pos = s.lower\_bound(7); // 获取区间的左端点

auto upper\_pos = s.upper\_bound(7); // 获取区间的右端点

for\_each( // for\_each算法

lower\_pos, upper\_pos, print // 输出7,7,7

);

除了这些适用于有序容器的算法，标准库中还有一些用于未排序容器的算法

vector<int> v = {1,9,11,3,5,7}; // vector容器

decltype(v.end()) pos; // 声明一个迭代器，使用decltype

pos = std::find( // 查找算法，找到第一个出现的位置

begin(v), end(v), 3

);

assert(pos != end(v)); // 与end()比较才能知道是否找到

pos = std::find\_if( // 查找算法，用lambda判断条件

begin(v), end(v),

[](auto x) { // 定义一个lambda表达式

return x % 2 == 0; // 判断是否偶数

}

);

assert(pos == end(v)); // 与end()比较才能知道是否找到

array<int, 2> arr = {3,5}; // array容器

pos = std::find\_first\_of( // 查找一个子区间

begin(v), end(v),

begin(arr), end(arr)

);

assert(pos != end(v)); // 与end()比较才能知道是否找到

小结：

C++标准库中还有很多宝藏算法，比如类似memcpy的copy/move算法，检查元素的all\_of/any\_of。

1. 算法专门操作容器的函数，是一种智能for循环，它的最佳搭档是lambda表达式
2. 算法通过迭代器来操作容器，使用两个端点作为操作范围吗，迭代器决定了算法的能力
3. for\_each是for的替代品，以函数式编程替代了面向过程编程。
4. 有多种排序算法，最基本的是sort，但应该根据实际情况选择更合适的算法，避免浪费
5. 在已排序的容器中，可以用lower\_bound查找元素
6. list/set/map提供了等价的排序，查找函数，更适应自己的数据结构
7. find/search是通用的查找算法，效率不高，但可用于未排序的容器

14|十面埋伏的并发：多线程真的很难吗

认识线程和多线程：

从语言层面来说，线程就是一个能独立运行的函数

auto f = []() // 定义一个lambda表达式

{

cout << "tid=" <<

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

};

thread t(f); // 启动一个线程，运行函数f

多线程的好处有很多，比如任务并行，避免IO阻塞，充分利用CPU等等，但多线程也对程序员的思维，能力提出了很多挑战，不夸张的说，它带来的麻烦可能要比好处多，比如死锁，同步，数据竞争等。

如果能规划好线程的工作，不与外部有过多的竞争，往往能避开很多坑，充分利用多线程跑满CPU，但现实的业务往往很复杂，很难做到完美的解耦。

在C++标准库中，为多线程提供了很多工具，可以在语言层面改善多线程应用。

一个最基本的常识是：读而不写，就不会有数据竞争。

所以在C++多线程中读取const变量总是安全的，对类调用const成员，对容器调用只读算法都是线程安全的。

一个多线程开发的原则是：最好的并发就是没有并发，最好的多线程就是没有线程

多线程开发实践：

要实现上面的原则，C++里面有四个基本工具：仅调用一次，线程局部变量，原子变量和线程对象。

仅调用一次：

比如多线程中初始化数据，防止初始化函数多次运行。为此可用“仅调用一次”来解决这个问题。

这个功能使用很简单，先声明一个once\_flag变量，最好是静态，全局的（线程可见），作为初始化的标志。

static std::once\_flag flag; // 全局的初始化标志

然后调用专门的call\_once()函数，以函数式编程的方式传递这个标志和初始化函数。这样C++就会保证，即使多个线程重入call\_once，也只有一个线程会成功运行初始化。

auto f = []() // 在线程里运行的lambda表达式

{

std::call\_once(flag, // 仅一次调用，注意要传flag

[](){ // 匿名lambda，初始化函数，只会执行一次

cout << "only once" << endl;

} // 匿名lambda结束

); // 在线程里运行的lambda表达式结束

};

thread t1(f); // 启动两个线程，运行函数f

thread t2(f);

线程局部存储：

读写全局变量是另一个比较常见的数据竞争场景。但业务场景中，全局变量不一定必须是共享的，可能仅仅是为了线程方便传入传出数据，或者本地cache。

换句话说，这应该是线程独占所有权，不应该多线程共享，术语叫做 线程局部存储。

在C++中，由关键字thread\_local实现，它是一个和static，extern同级的 变量存储说明。

有tread\_local修饰的变量在每一个线程中都有一个独立的副本，是线程独占的，所以不会有竞争的问题。

thread\_local int n = 0; // 线程局部存储变量

auto f = [&](int x) // 在线程里运行的lambda表达式，捕获引用

{

n += x; // 使用线程局部变量，互不影响

cout << n; // 输出，验证结果

};

thread t1(f, 10); // 启动两个线程，运行函数f

thread t2(f, 20);

如果把变量改成static的，由于两个线程共享，n会被累加。

static int n = 0; // 静态全局变量

... // 代码与刚才的相同

和call\_once一样，thread\_local也很容易使用，但这要求你对线程的共享数据有清楚的认识，区分出独占的那部分，消除多线程对数据的并发访问。

原子变量：

对于必须要共享的数据，要想保证多线程读写共享数据的一致性，必须要解决同步问题，不能让两个线程同时写。

互斥量可以解决这个问题，但成本太高，所以对于小数据，可以采用原子化这个方案。

但不是所有的操作都可以原子化的，目前C++只支持最基本的类型原子化，比如atomic\_int，atomic\_long等等。

using atomic\_bool = std::atomic<bool>; // 原子化的bool

using atomic\_int = std::atomic<int>; // 原子化的int

using atomic\_long = std::atomic<long>; // 原子化的long

这些原子变量是模板类atomic的特化形式，包装了原始的类型，多线程读写不会出错。

但跟原始类型不一样的点是原子变量禁用了拷贝构造函数，所以初始化不能用“=”赋值，只能用圆括号或者花括号。

atomic\_int x {0}; // 初始化，不能用=

atomic\_long y {1000L}; // 初始化，只能用圆括号或者花括号

assert(++x == 1); // 自增运算

y += 200; // 加法运算

assert(y < 2000); // 比较运算

除了模拟整数运算，原子变量还有一些特殊的原子操作，比如store，load，exchange，compare\_exchange\_weak/compare\_exchange\_strong，以及原子算数操作。

atomic\_flag和atomic<bool>都可执行TAS操作实现自旋锁。

但atomic只有test\_and\_set()以及clear()操作。

线程：

C++标准库里面有专门的线程类，使用他就可以简单地创建线程，在命名空间std::this\_thread中，还有yield()，get\_id()，sleep\_for()等几个方便管理的函数。

下面的代码同时示范了thread和atomic的用法：

static atomic\_flag flag {false}; // 原子化的标志量

static atomic\_int n; // 原子化的int

auto f = [&]() // 在线程里运行的lambda表达式，捕获引用

{

auto value = flag.test\_and\_set(); // TAS检查原子标志量

if (value) {

cout << "flag has been set." << endl;

} else {

cout << "set flag by " <<

this\_thread::get\_id() << endl; // 输出线程id

}

n += 100; // 原子变量加法运算

this\_thread::sleep\_for( // 线程睡眠

n.load() \* 10ms); // 使用时间字面量

cout << n << endl;

}; // 在线程里运行的lambda表达式结束

thread t1(f); // 启动两个线程，运行函数f

thread t2(f);

t1.join(); // 等待线程结束

t2.join();

基于看不到线程的线程才是好线程，可以用async()函数代替，它的含义是异步运行一个任务，隐含的动作是启动一个线程去执行，但不保证立即启动（此处应该是因为CPU调度此线程的优先级不高，也可以第一个参数传递std::launch::async，要求立即启动线程）。

大多数thead能做的事情都可以用async()来实现。

auto task = [](auto x) // 在线程里运行的lambda表达式

{

this\_thread::sleep\_for( x \* 1ms); // 线程睡眠

cout << "sleep for " << x << endl;

return x;

};

auto f = std::async(task, 10); // 启动一个异步任务

f.wait(); // 等待任务完成

assert(f.valid()); // 确实已经完成了任务

cout << f.get() << endl; // 获取任务的执行结果

其实这还是函数式编程的思路，在更高抽象级别上看待问题，异步并发多个任务，让底层自动管理线程，要比我们自己手动管理更好（比如内部使用线程池或者其他机制）。

async函数会返回一个future变量，可以认为是代表了执行结果的期货，如果任务有返回值，可以用get（）函数获取。要注意get（）函数只能调用一次。

这里还有一个很隐蔽的坑，就是如果不显式地获取async的返回结果，他就会同步阻塞到直至任务完成。所以即使我们不关心认为的返回值，也总要使用auto来配合async（）。

std::async(task, ...); // 没有显式获取future，被同步阻塞

auto f = std::async(task, ...); // 只有上一个任务完成后才能被执行

小结：

1. 线程是并发最常用的实现方式，好处是任务并行，避免阻塞，坏处是 开发难度高，有数据竞争，死锁等很多坑。
2. call\_once()实现了仅调用一次的功能，由它修饰的函数只会调用一次，避免多线程初始化的冲突。
3. Thread\_local实现了线程局部存储，让每个线程都独立访问数据，互不干扰。
4. Atomic实现了原子变量，可以用作线程安全的计数器，也可以实现无锁数据结构。
5. Async()启动一个异步任务，相当于开了一个线程，但内部通常会有优化，比直接使用线程更好。

好消息：C++20中正式加入了协程（关键字co\_wait，co\_yield，co\_return），它是用户态的线程，没有系统级别的线程这么多麻烦事，使用它就可以写出开销更低，性能更好的并发程序。

15|序列化：简单通用的数据交换格式有哪些

Json：注重易用性。

推荐使用https://github.com/nlohmann/json

MessagePack；

ProtoBuffer：

需要安装一个预处理器和开发库，编译时还要链接动态库 -lprotobuf

apt-get install protobuf-compiler

apt-get install libprotobuf-dev

g++ protobuf.cpp -std=c++14 -lprotobuf -o a.out

PB的一个特点是数据有模式，必须先写一个IDL文件，在里面定义好数据结构，只有定义好结构才能被序列化或者反序列化。

这样的好处是接口就是清晰的文档，沟通无歧义。但另一方面就是缺乏灵活性，改接口会导致一连串的操作，有点繁琐。

下面是一个简单的PB定义：

syntax = "proto2"; // 使用第2版

package sample; // 定义名字空间

message Vendor // 定义消息

{

required uint32 id = 1; // required表示必须字段

required string name = 2; // 有int32/string等基本类型

required bool valid = 3; // 需要指定字段的序号，序列化时用

optional string tel = 4; // optional字段可以没有

}

有了定义结构的文档，在使用protobuf的工具生成C++源码，然后把源码加入自己的项目就可以使用了。

protoc --cpp\_out=. sample.proto // 生成C++代码

PB的重要接口：

set/has()函数，字段名设置值，检查值是否存在

IsInitialized()函数，检查数据是否完整，required字段必须有值

DebugString()输出数据的可读字符描述

ByteSize函数，返回序列化数据的长度

SerializeToString从对象转化到字符串

ParseFromString从字符串反序列化到对象

SerializeToArray，ParseFromArray序列化的目标是数组

下面示范用法：

using vendor\_t = sample::Vendor; // 类型别名

vendor\_t v; // 声明一个PB对象

assert(!v.IsInitialized()); // required等字段未初始化

v.set\_id(1); // 设置每个字段的值

v.set\_name("sony");

v.set\_valid(true);

assert(v.IsInitialized()); // required等字段都设置了，数据完整

assert(v.has\_id() && v.id() == 1);

assert(v.has\_name() && v.name() == "sony");

assert(v.has\_valid() && v.valid());

cout << v.DebugString() << endl; // 输出调试字符串

string enc;

v.SerializeToString(&enc); // 序列化到字符串

vendor\_t v2;

assert(!v2.IsInitialized());

v2.ParseFromString(enc); // 反序列化

PB的IDL定义和接口都太死板生硬，并且只能用最基本的类型，不支持标准容器。

并且支持的编程语言较少。

小结：

1. json是纯文本，方便阅读，编辑，适用性最广
2. MassagePack是二进制，小巧高效，在开源届接受程度较高
3. ProtoBuf是工业级的数据格式，注重安全和性能，多用在大公司的产品里

16|网络通信：不想写原生的socket

要自己写出一个健壮可靠的网络应用程序是相当麻烦的。

所以可以使用C++中那些好用，已经写好的网络通信库：libcurl，cpr和ZMQ

Libcurl：高移植性，功能丰富的通信库。

要使用libcurl，首先要安装

apt-get install libcurl4-openssl-dev

Libcurl的接口可以大致分为两系列：easy系列和mutil系列。其中easy系列是简单的同步调用，比较简单，mutil系列是异步的多线程调用，比较复杂。

以http收发数据举例，基本步骤有四个：

1. 使用curl\_easy\_init()创建一个句柄，类型是CURL\*，直接用auto推导就行。
2. 使用curl\_easy\_setop()设置请求的各种参数，比如请求方法， URL，header/body数据，超时，回调函数等等。
3. 使用curl\_easy\_perform()发送数据,返回的数据有回调函数处理。
4. 使用curl\_easy\_clearup()清理句柄相关的资源，结束会话。

#include <curl/curl.h> // 包含头文件

auto curl = curl\_easy\_init(); // 创建CURL句柄

assert(curl);

curl\_easy\_setopt(curl, CURLOPT\_URL, "http://nginx.org"); // 设置请求URI

auto res = curl\_easy\_perform(curl); // 发送数据

if (res != CURLE\_OK) { // 检查是否执行成功

cout << curl\_easy\_strerror(res) << endl;

}

curl\_easy\_cleanup(curl); // 清理句柄相关的资源

如果不设置回调函数，libcurl会使用默认的回调函数，将http的响应数据输出到标准流，也就是打印到屏幕上。

如果想自己处理收到的数据，就需要自己实现回调函数，因为libcurl是C写的，所以回调函数必须是函数指针。不过C++11允许使用lambda表达式，这里用了一个特别规定，无捕获的lambda表达式可以显式地转换为函数指针。

// 回调函数的原型

size\_t write\_callback(char\* , size\_t , size\_t , void\* );

curl\_easy\_setopt(curl, CURLOPT\_WRITEFUNCTION, // 设置回调函数

(decltype(&write\_callback)) // decltype获取函数指针类型，显式转换

[](char \*ptr, size\_t size, size\_t nmemb, void \*userdata)// lambda

{

cout << "size = " << size \* nmemb << endl; // 简单的处理

return size \* nmemb; // 返回接收的字节数

}

);

Libcurl的用法大概就是这样，关键就是setopt这一步，必须通过查文档知道该用哪些标志宏，写一些重复单调的代码。

基于此，我们可以封装一些类，少敲一下重复的代码。而cpr就是这么做的。

Cpr：更现代，更易用的通信库

Cpr是libcurl的一个C++11封装，使用了很多现代C++的高级特性，非常简单易用。

可以从github上获取源码，在用cmake编译。

git clone https://github.com/whoshuu/cpr/

cmake . -DUSE\_SYSTEM\_CURL=ON -DBUILD\_CPR\_TESTS=OFF

make && make install

与libcurl相比，cpr的使用非常简洁，一句话就可以发送http请求

#include <cpr/cpr.h> // 包含头文件

auto res = cpr::Get( // GET请求

cpr::Url{"http://openresty.org"} // 传递URL

);

也不用写回调函数，http响应就是函数的返回值，通过成员变量就可以得到报文里的各个组成部分。

cout << res.elapsed << endl; // 请求耗费的时间

cout << res.url << endl; // 请求的URL

cout << res.status\_code << endl; // 响应的状态码

cout << res.text.length() << endl; // 响应的body数据

for(auto& x : res.header) { // 响应的头字段

cout << x.first << "=>" // 类似map的结构

<< x.second << endl;

}

在cpr中，HTTP协议的概念都被实现成响应的函数或者类，内部再转化为libcurl操作，主要有：

1. GET/POST/HEAD等请求方法，使用同名的Get/Head/Post函数
2. URL使用URL类，其实就是string的别名
3. URL使用Parameter类，kv结构，类似map
4. 请求字段使用Head类，他其实是map的别名，使用制定的函数实现了大小写无关比较
5. Cookie使用Cookie类，也是KV结构，近似map
6. 请求体使用body类
7. 超时设置使用Timeout类

这些函数用法都非常自然，并且可以使用花括号初始化，符合思维习惯

const auto url = "http://openresty.org"s; // 访问的URL

auto res1 = cpr::Head( // 发送HEAD请求

cpr::Url{url} // 传递URL

);

auto res2 = cpr::Get( // 发送GET请求

cpr::Url{url}, // 传递URL

cpr::Parameters{ // 传递URL参数

{"a", "1"}, {"b", "2"}}

);

auto res3 = cpr::Post( // 发送POST请求

cpr::Url{url}, // 传递URL

cpr::Header{ // 定制请求头字段

{"x", "xxx"},{"expect",""}},

cpr::Body{"post data"}, // 传递body数据

cpr::Timeout{200ms} // 超时时间

);

Cpr也支持异步处理，但它内部并不是用libcurl中的multi接口，而是使用了标准库中的future和async。

异步接口与同步接口用法相似，只是名字后面多了Async后缀，返回的是一个future对象，可以调用wait或者get来获取响应结果。

auto f = cpr::GetAsync( // 异步发送GET请求

cpr::Url{"http://openresty.org"}

);

auto res = f.get(); // 等待响应结果

cout << res.elapsed << endl; // 请求耗费的时间

ZMQ：高效，快速，多功能的通信库

Libcurl和cpr处理的都是http协议，协议自身有一些限制，比如必须要一来一回，必须点对点直连，在超大数据通信时就不太合适。

还有一点是，libcurl和cpr只能充当HTTP的客户端，如果想写服务端程序，这两个工具就排不上用场了。

ZMQ是一个更底层，更灵活的网络通信工具，不仅高效快速，还能同时支持客户端和服务端。

其实，ZMQ不仅仅是一个单纯的网络框架，更像是一个高级的异步并发框架。

从名字上可以看出来Zero Message Queue，零延迟的消息队列，意味着它处理收发消息，还可以用作消息中间件，解耦多个服务之间的强依赖关系，搭建高效的，有弹性的分布式系统。从而超越原生的socket。

ZMQ只需要安装开发库就可以使用了，相当于把消息队列直接嵌入到应用程序中。

apt-get install libzmq3-dev

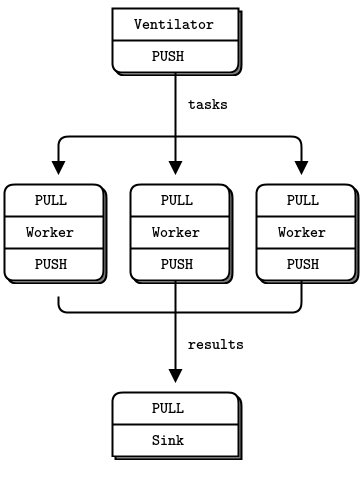
ZMQ是C++开发的，但对外提供的是纯C的接口，不过它也有很多C++的封装。

由于ZMQ自身的定位是更高层次的“异步消息队列”，所以它的用法就不想socket和http那样简单直白，而是定义了五种工作模式，来适应不同场景的网络通信。

1. 原生模式（RAW），没有消息队列功能，相当于socket的简单封装
2. 结对模式（PAIR），两个端点一对一通信
3. 请求响应模式（REQ-REP），两个端点一对一通信，但请求必须有响应
4. 发布订阅模式（PUB-SUB），一对多通信，一个端点发布消息，多个端点接收处理
5. 管道模式（PUSH-PULL），或者叫流水线，可以一对多，也可以多对一

前四种是类似HTTP协议，C-S架构。

以管道模式示范ZMQ的用法，它非常适合进程间无阻塞传送海量数据，也有点MapReduce的意思。



在ZMQ中有两个基本类。

第一个是context\_t，它是ZMQ的运行环境，使用ZMQ前必须要先创建它。

第二个是socket\_t，表示ZMQ的套接字，需要指定五种工作模式（与原生的socket无关）。

下面的代码声明了一个全局的ZMQ全局变量，并定义了一个lambda表达式，用来生产ZMQ套接字。

const auto thread\_num = 1; // 并发线程数

zmq::context\_t context(thread\_num); // ZMQ环境变量

auto make\_sock = [&](auto mode) // 定义一个lambda表达式

{

return zmq::socket\_t(context, mode); // 创建ZMQ套接字

};

和原生socket一样，ZMQ套接字也必须关联一个地址才能收发数据，但它不仅支持TCP/IP，还支持进程内和进程间通信，这在本机交换数据时更高效。

TCP通信地址的形式是”tcp://...”，指定IP地址和端口号

进程内通信地址是”inproc://...”，指定一个本地可访问的路径

进程间通信地址的形式是”ipc://...”，也是一个本地可访问到路径

用bind()/connect()这两个函数把ZMQ套接字链接起来后，就可以使用send()/recv()来收发数据了。

const auto addr = "ipc:///dev/shm/zmq.sock"s; // 通信地址

auto receiver = [=]() // lambda表达式接收数据

{

auto sock = make\_sock(ZMQ\_PULL); // 创建ZMQ套接字，拉数据

sock.bind(addr); // 绑定套接字

assert(sock.connected());

zmq::message\_t msg;

sock.recv(&msg); // 接收消息

string s = {msg.data<char>(), msg.size()};

cout << s << endl;

};

auto sender = [=]() // lambda表达式发送数据

{

auto sock = make\_sock(ZMQ\_PUSH); // 创建ZMQ套接字，推数据

sock.connect(addr); // 连接到对端

assert(sock.connected());

string s = "hello zmq";

sock.send(s.data(), s.size()); // 发送消息

};

这段代码实现了最基本的客户端和服务端，但使用ＺＭＱ完全不需要考虑底层ＴＣＰ／ＩＰ细节。它会保证消息异步，安全地到达服务器，让你关注网络通信之上更有价值的业务逻辑。

两个实际工作中比较有用的细节。

1. 一个是ＺＭＱ环境的线程数，默认值是１，可以适当增大一些提高并发能力。
2. 另一个是ＺＭＱ本地缓存的收发消息的缓存数量，术语叫ＨＩＧＨ　ＷＡＴＥＲ　ＭＡＫＥ，如果收发的数据过多，超过ＨＷＭ，ＺＭＱ要么阻塞要么丢弃消息。

sock.setsockopt(ZMQ\_RCVHWM, 1000); // 接收消息最多缓存1000条

sock.setsockopt(ZMQ\_SNDHWM, 100); // 发送消息最多缓存100条

ＨＷＭ设置上限很大，无须担心。

ＺＭＱ还有很多强大的功能，可以阅读官网教程和指南。

小结：

1. ｌｉｂｃｕｒｌ是一个功能完善。稳定可靠的应用层通信库，最常用的即使ＨＴＴＰ协议。
2. ｃｐｒ是对ｌｉｂｃｕｒｌ的Ｃ＋＋封装。
3. Ｌｉｂｃｕｒｌ和ｃｐｒ只能作为客户端使用，不能编写服务器应用
4. ＺＭＱ是一个高级的网络通信库，支持多种通信模式，可以把消息队列功能直接嵌入应用程序，搭建分布式系统。

C++23会加入networking库，networking基于boost.asio，统一了操作系统各种异步机制（epoll，kqueue，IOCP），而且支持协程。有了它，我们的网络通信工作就会更加 轻松。

17|脚本语言：搭建高性能的混合系统

使用pybind11，在python中调用C++实现的函数。

18|性能分析，找出程序的瓶颈

运行阶段可以做的事情：调试，测试，性能分析

调试：GDB

测试：单元测试，Gtest

性能分析：

1. 系统级工具：top，pstack，strace，perf。
2. 源码级工具：gperftools

小结：

最简单的性能分析工具是 top，可以快速查看进程的 CPU、内存使用情况；

pstack 和 strace 能够显示进程在用户空间和内核空间的函数调用情况；

perf 以一定的频率采样分析进程，统计各个函数的 CPU 占用百分比；

gperftools 是“侵入”式的性能分析工具，能够生成文本或者图形化的分析报告，最直观的方式是火焰图。

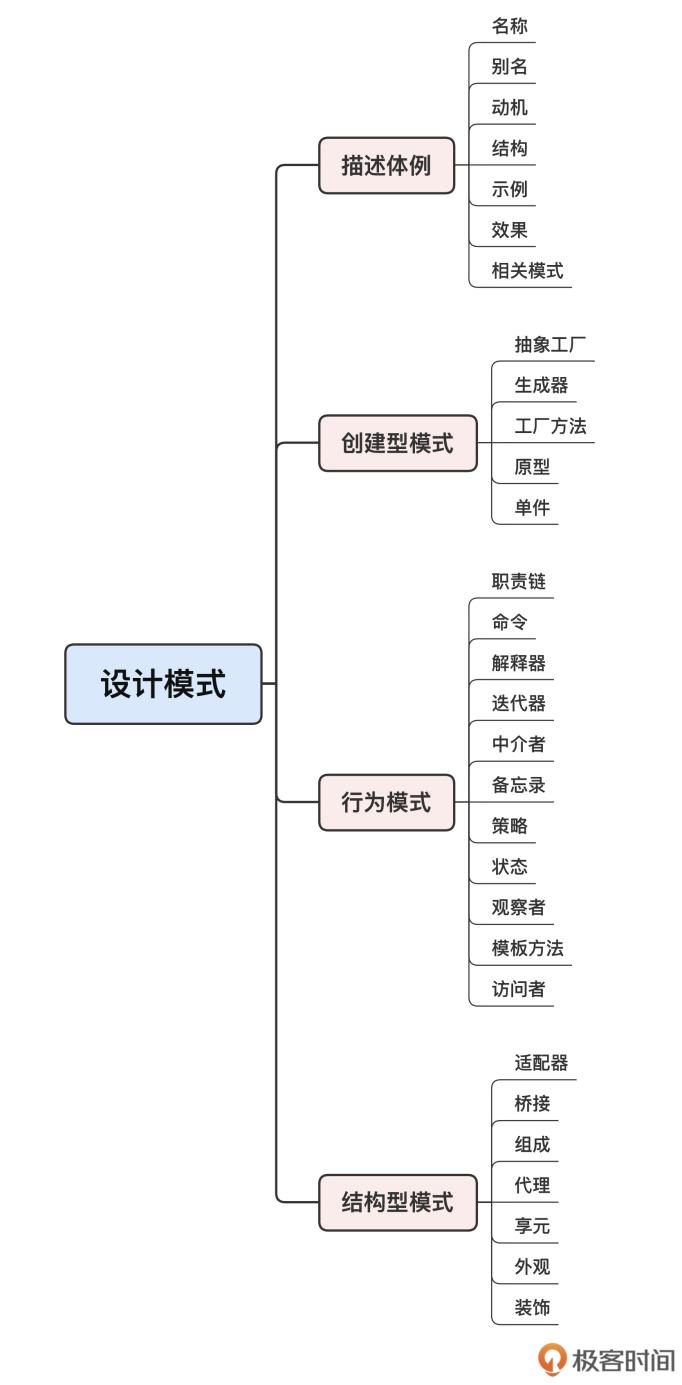
19|设计模式上：C++与设计模式有啥关系？

为什么要有设计模式：

设计模式系统地描述了一些软件开发过程中的常见问题，应用场景和对应的解决方案，给出了专家级别的设计思路和指导原则。

学习，理解设计模式，才能用好面向对线的C++

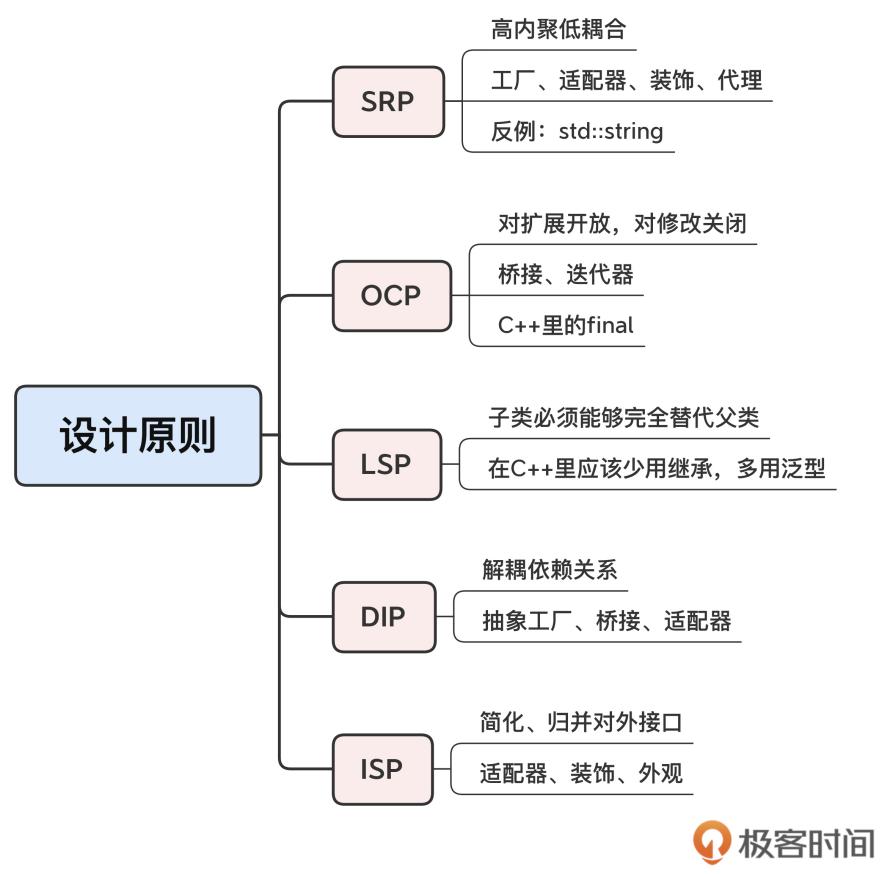
经典的设计模式有23个，分为创建型，结构型，和行为型模式。分别对应如何创建对象，如何组合对象，以及如何处理对象之间的动态通信和职责分配。



要去关注设计模式的参与者、设计意图、面对的问题、应用的场合、后续的效果等代码之外的部分，它们通常比实现代码更重要。

学习，理解设计模式，才能用好多范式的C++

1. 单一原则：不做多余的事情，高内聚低耦合。
2. 开闭原则：对扩展开放，对修改关闭，关键是做好封装。
3. 里氏替换原则：子类必须能够完全替代父类。
4. 接口隔离原则：尽量简化，归并给外部调用的接口。
5. 依赖倒置原则：上层要避免依赖下层的实现细节。
6. DRY，KISS，YAGNI，LOD原则



小结：  
1. 面向对象是主流的编程范式，使用设计模式可以比较容易的得到良好的面向对象设计。

1. 经典的设计模式有23中，分为创建型，结构型和行为型。
2. 应该从多角度，多方面去研究设计模式，关注代码之外的部分，学习解决问题的思路。
3. 设计原则是更高层面的指导思想。
4. 最常用的设计原则是“SOLID”，此外还有“DRY”和“KISS”。

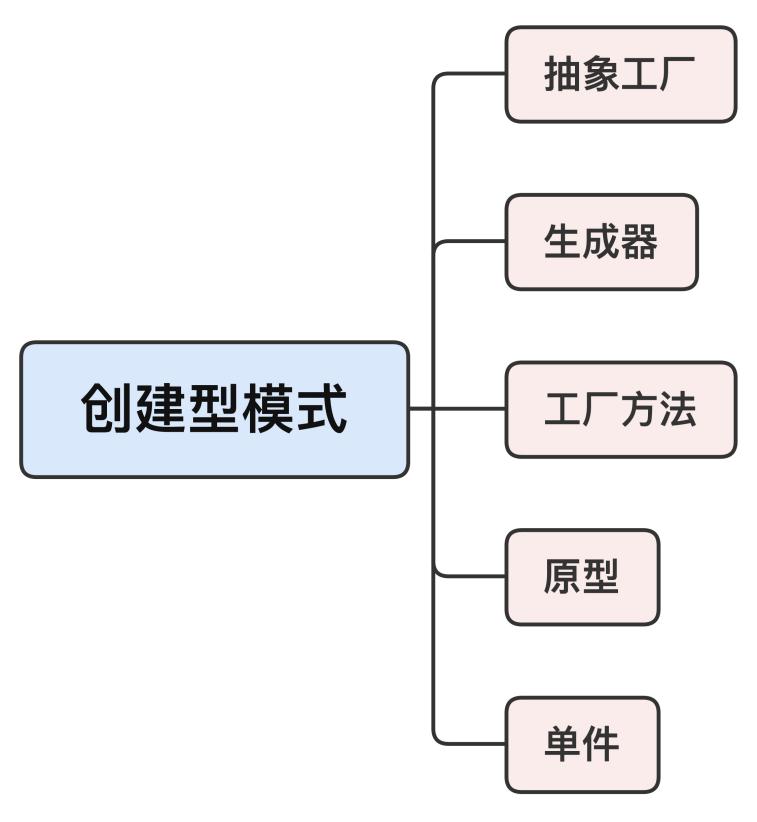
要谨防滥用设计模式。

C++是多范式的编程语言，在适当的时候使用其他范式。

20|设计模式下：C++是如何应用设计模式的？

创建型模式:隐藏了类的实例化过程和细节，让对象的创建独立于系统的其他部分。

最有用的是单例和工厂：



单例模式在多线程中的多次初始化问题，可以用call\_once解决，但更方便的方法是直接使用static静态变量。

工厂模式：

DRY原则，避免避免重复代码，简单地认为是“对new的封装”。

把new用工厂封装起来，形成了一个中间层，隔离了客户代码和创建对象。make\_unique和make\_share两个函数就是工厂模式的具体应用，直接返回智能指针，接口更简洁。

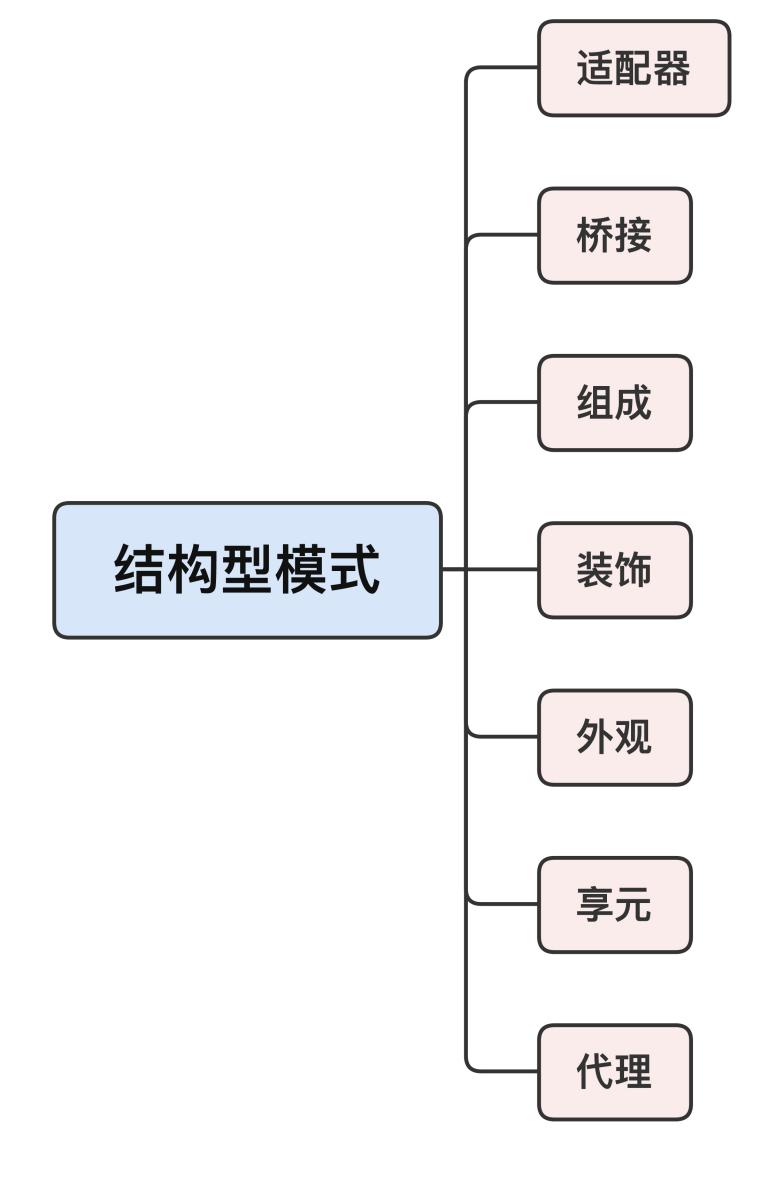
使用工厂模式的关键，就是在于要理解它面对的问题和解决问题的思路，比如说创建专属的对象，创建成套的对象。重点是“如何创建对象，创建出什么样的对象”，用函数或者类回避单纯地用new更灵活。

结构型模式：

它关注的是对象的静态联系，以灵活，可拆卸，可装配的方式组合出新的对象。

结构型模式的重要特征：虽然有多个参与者，但最终得到并且使用的是“一个对象而非多个对象”。

在C++中常用的是适配器，外观和代理模式，



1. 适配器模式

适配器的目的是接口转换，不需要修改源码，就能把一个对象转换成可以在本系统中使用的形式。

它的一个适用场景是：当C++中有第三方库或者外部接口时，通常这些接口不会恰好符合我们的系统，功能很好，但是不能直接用，改源码很难，这时候用适配器适配一下让外部工具match我们的系统，而两边都不需要动，皆大欢喜。

容器中的array就是一个适配器，包装了C++的原生数组，转换成容器的形式，让裸内存数据也可以接入标准库的泛型体系。

array<int, 5> arr = {0,1,2,3,4};

auto b = begin(arr);

auto e = end(arr);

for\_each(b, e, [](int x){...});

1. 外观模式

封装了一组对象，目的是简化这组对象的通信关系，提供一个搞层次的易用接口，让外部用户更容易使用。

外观模式的特点是内部封装了很多对线，然后对外表现成一个对象，一个大管家对象。

Async函数就是一个外观模式例子，它封装了线程创建，调度等细节，用起来很简单，但也不排斥你直接使用thread，mutex等线程工具。

auto f = std::async([](){...});

f.wait();

1. 代理模式

和适配器有点类似，都是封装一个对象，但它的目的是为了控制一个对象，不允许外部与内部对象直接通信。

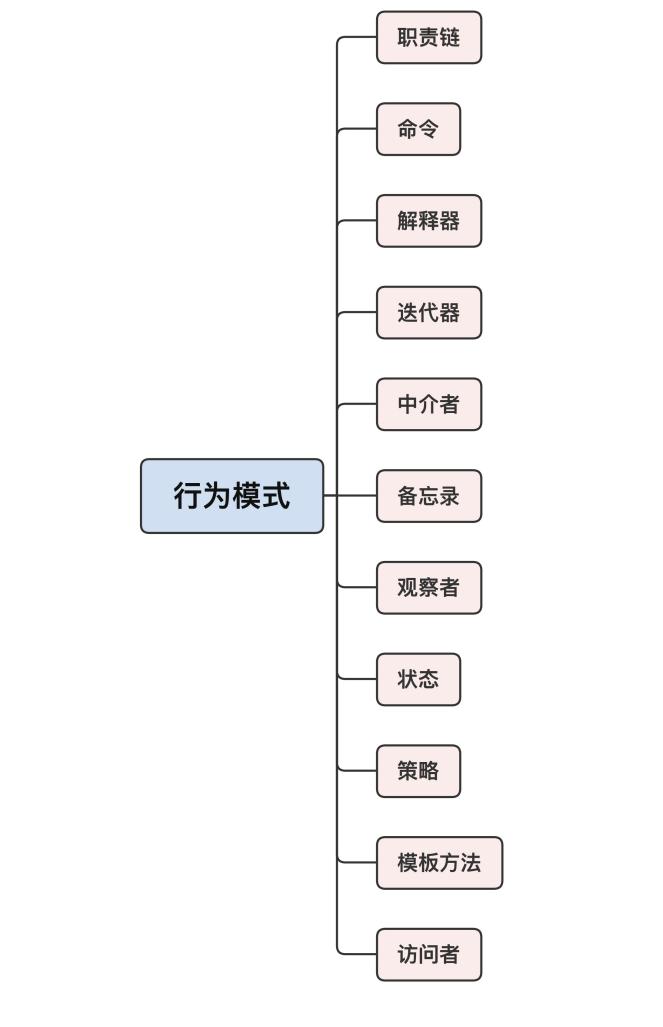
代理模式的应用非常广泛，如果你想屏蔽，隐藏，增强或者优化一个类，就可以使用代理。

客户代码只能看到代理对象，这样就实现了控制的目的。

C++中的智能指针就是一个代理模式的典型应用。

行为模式：

行为模式描述了对象之间动态的消息传递，对就是对象的行为，工作的方式。



比较容易理解，容易使用的有职责链，命令和策略模式。

职责链模式和命令模式经常一起使用，职责链把多个对象串成一个链条，让链条上的每个对象都有机会处理请求，而请求通常就是使用命令模式，把相关的数据打包成一个对象，解耦请求方和接收方。

C++中的异常处理就是职责链+命令模式的一个实际应用。

在异常的处理过程中，异常类exception就是一个命令对象，throw抛出异常就是发起了一个请求处理流程，而一系列的try-catch就是处理异常的职责链。

策略模式的要点是策略二字，它封装了不同的算法，可以在运行的时候灵活替换，从而在外部“非侵入”地改变系统的行为内核。

容器和算法中用到的比较函数，散列函数和for\_each的lambda表达式都是策略模式的具体应用。

小结：

1. 创建型模式里常用的是工厂和单例，封装了对线的创建，隔离了对线的生产和使用。
2. 结构型模式里面常用的是适配器，外观，代理模式，目的是为了适配，简化或者控制，隔离客户代码和原对象的接口。
3. 行为型模式中常用的是职责链， 命令和策略模式，只有在运行时才会建立联系，封装，隔离了程序中动态变化的部分。



21|知识串讲上：开发一个书店应用

22|知识串讲下：开发一个书店应用