判断一个表达式是否是左值的一个有用的启发式方法是询问你是否可以接受它的地址。如果你能，它通常是。如果你不能，它通常是一个右值。

当我提到函数对象时，我通常指的是支持operator（）成员函数的类型的对象。

C++中的许多东西都可以声明和定义。声明引入名称和类型，但不提供细节，例如存储位于何处或如何实现：

定义提供了存储位置或实施详细信息：

第1项：了解模板类型演绎。



编译器首先根据expr的类型推导出T，然后根据ParamType的形式确定最终的参数类型

我们所说的模版，通常包含两个部分模版类型T和模版参数paramtype。其中T推导的是参数的类型，paramtype推导这个T有无const，引用，或者指针。

分三种情况讨论，

1，T只带有一个&，传入变量x会被添加引用，而如果x自带引用了，则会被忽略。Const一直保留。

2. T带有两个&&，（通用引用）。如果 expr 是左值，那么 T 和 ParamType 都会被推导为左值引用。这有两点不同寻常之处。首先，这是模板类型推导中唯一一种 T 被推导为引用的情况。

 T 带有两个 &，传入左值被推导为左值引用，而传入右值被推导为右值引用

3 T不带有&时，引入和const都会被忽略，指针除外，指针的左边的const不会被忽略

**数组作为按值传递的模板参数**

当数组作为按值传递的模板参数时，模板类型 T 会被推导为指针类型。因为在函数参数声明中，数组会被当作指针处理。

**数组作为按引用传递的模板参数**

当数组作为按引用传递的模板参数时，模板类型 T 会被推导为数组的实际类型，包含数组的大小信息。

·在模板类型推导期间，作为引用的参数被视为非引用，即，其参考性被忽略。

·在推导通用引用参数的类型时，左值参数得到特殊处理。

·当推导按值参数的类型时，const和/或volatile参数被视为非const和非volatile。

·在模板类型演绎过程中，数组或函数名的参数衰减为指针，除非它们用于初始化引用。

第二项：理解自动类型推导。

将auto看作是T。

在 C++ 中，统一初始化提倡使用花括号进行初始化。但由于 auto 对花括号初始化器的特殊处理，可能会导致开发者不小心声明了 std::initializer\_list 类型的变量，而不是原本想要的类型。这是 C++11 编程中容易犯的错误，所以有些开发者只在必要时才使用花括号初始化。

·自动类型演绎通常与模板类型演绎相同，但自动类型演绎假设带括号的初始化器表示std：：initializer\_list，而模板类型演绎则不是。

·函数返回类型或lambda参数中的auto意味着模板类型推导，而不是auto类型推导。

第3条：理解decltype。

decltype几乎总是产生变量或表达式的类型，而无需任何修改。

·对于除名称之外的T类型的左值表达式，decltype始终报告T&类型。

C++14支持decltype（auto），它和auto一样，从类型的初始化器推导出类型，但它使用decltype规则执行类型推导。

第4项：知道如何查看推导类型。

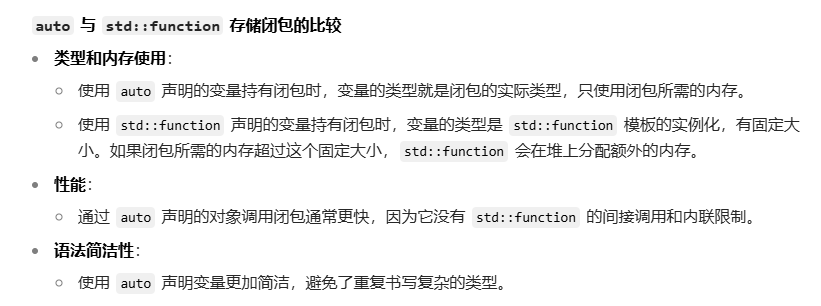
* **取地址操作**：当对 theAnswer 取地址时，得到的是一个指向 const int 类型对象的指针。因为 theAnswer 的值不能被修改，所以指向它的指针必须保证不能通过该指针去修改 theAnswer 的值，因此这个指针的类型是 const int\*。

演绎类型通常可以使用IDE编辑器、编译器错误消息和Boost TypeIndex库来查看。

·某些工具的结果可能既没有帮助也不准确，因此了解C++的类型推导规则仍然是必不可少的。

第5条：与显式类型声明相比，更喜欢auto。

std::function 是 C++11 标准库中的一个模板，它是函数指针概念的泛化。不过，函数指针只能指向函数，而 std::function 对象可以引用任何可调用对象，即任何可以像函数一样被调用的东西。



闭包是一个可调用对象，它可以捕获其所在作用域中的变量，并且在其生命周期内保持对这些变量的访问权，即使该作用域已经结束。在 C++ 里，lambda 表达式就是创建闭包的主要方式。

自动变量必须初始化，通常不受可能导致可移植性或效率问题的类型不匹配的影响，可以简化重构过程，并且通常需要比显式指定类型的变量更少的类型。

·自动类型化的变量会遇到第2项和第6项中描述的陷阱。

第6条：当自动推导出不需要的类型时，使用显式类型的初始化器。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

·“不可见”代理类型会导致auto为初始化表达式推导出“错误”类型。

·显式类型初始化器习惯用法强制auto推断出您希望它具有的类型。

图片包含 背景图案

描述已自动生成

文本, 信件

描述已自动生成

“隐形” 代理类是为了实现某些特定的功能或优化而设计的类，它们通常会伪装成被代理类型的对象，让程序员可以像使用被代理类型一样使用它们，但在内部实现上有一些特殊处理。

第7条：创建对象时区分（）和{}。

初始化值可以用括号、等号或花括号来指定：

C++ 通常会把它和仅使用花括号的版本视为相同。

文本

描述已自动生成

花括号初始化被称为 “统一” 初始化。在 C++ 指定初始化表达式的三种方式中，只有花括号可以在任何地方使用。

花括号初始化的一个新特性是，它禁止内置类型之间的隐式缩窄转换。如果花括号初始化器中的表达式的值不能保证可以用被初始化对象的类型表示，编译器必须报错

**花括号初始化的优点回顾**

花括号初始化具有诸多优点：

* **通用性**：可以在各种不同的场景下使用，比如容器初始化、非静态数据成员默认初始化以及不可拷贝对象的初始化等。
* **防止隐式缩窄转换**：对于内置类型，花括号初始化会检查是否存在缩窄转换，如果可能会导致数据丢失，编译器会报错。
* **避免最令人头疼的解析问题**：在默认构造对象时，使用花括号不会被误解析为函数声明。

**花括号初始化的缺点**

花括号初始化的主要问题在于它与 std::initializer\_list 和构造函数重载解析之间的复杂关系，可能导致代码行为不符合预期。

**不涉及 std::initializer\_list 时的构造函数调用**

当构造函数不涉及 std::initializer\_list 参数时，使用括号和花括号进行构造函数调用的效果是一样的。示例代码如下：

·带括号的初始化是最广泛使用的初始化语法，它可以防止收缩转换，并且不受C++最令人烦恼的解析的影响。

·在构造函数重载解析过程中，如果可能的话，花括号初始化器会匹配到std：：initializer\_list参数，即使其他构造函数提供了看似更好的匹配。

圆括号和大括号之间的选择可以产生显著差异的一个例子是创建一个带有两个参数的std：：vector<numeric type>。

·在模板内创建对象时，在括号和大括号之间进行选择可能具有挑战性。

首先，作为类的设计者，你需要意识到，如果你的重载构造函数集合中包含一个或多个接受 std::initializer\_list 的函数，使用花括号初始化的客户端代码可能只会看到 std::initializer\_list 重载版本

第8条：nullptr优于0和NULL。

字面量 0 是一个 int 类型，而不是指针。如果 C++ 在只能使用指针的上下文中遇到 0，它会勉强将 0 解释为空指针，但这只是一种后备策略。C++ 的主要策略是将 0 视为 int 类型，而非指针。

实际上，NULL 的情况也是如此。在 NULL 的具体细节上存在一些不确定性，因为实现允许为 NULL 指定除 int 之外的整数类型（例如 long）。这种情况并不常见，但这其实无关紧要，因为这里的问题不在于 NULL 的具体类型，而在于 0 和 NULL 都没有指针类型。

而 nullptr 是 C++11 引入的一个新关键字，它的类型是 std::nullptr\_t。std::nullptr\_t 可以隐式转换为所有原始指针类型，

首选nullptr而不是0和NULL。

·避免整型和指针类型的重载。

第9条：与typedef相比，更喜欢别名声明。

·typedef不支持模板化，但别名声明支持。

·模板避免使用“：：type”后缀，在模板中，引用typedef时通常需要使用“typeName”前缀。

C++14为所有C++11类型traits转换提供了别名模板。

第10条：与无作用域枚举相比，更喜欢有作用域枚举。

由于有作用域枚举是通过 enum class 声明的，所以它们有时也被称为 “枚举类（enum classes）”。图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

* C++98 风格的枚举现在被称为无作用域枚举（unscoped enums）。
* 有作用域枚举（scoped enums）的枚举项仅在该枚举内部可见。它们只有通过强制类型转换才能转换为其他类型。
* 有作用域枚举和无作用域枚举都支持指定底层类型。有作用域枚举的默认底层类型是 int。无作用域枚举没有默认底层类型。
* 有作用域枚举总是可以进行前向声明。无作用域枚举只有在其声明中指定了底层类型时才可以进行前向声明。

第11条：优先使用已删除的函数，而不是私有的未定义函数。

**已删除函数用于过滤函数调用类型**

**已删除函数用于阻止特定模板实例化**

偏好删除的函数，而不是私有的未定义的函数。

·任何函数都可以被删除，包括非成员函数和模板实例化

**条款 12：将重写函数声明为 override**

**虚函数重写的条件**

1. **基类函数为虚函数**：只有基类中的函数被声明为 virtual，派生类中的同名函数才有可能重写它。
2. **函数名称相同**：基类和派生类中函数的名称必须一致（析构函数是个例外，派生类的析构函数会自动重写基类的虚析构函数）。
3. **参数类型相同**：基类和派生类函数的参数类型必须完全相同，否则不会构成重写，而是函数重载。
4. **常量性相同**：如果基类函数是 const 成员函数，派生类中要重写该函数，也必须是 const 成员函数。
5. **返回类型和异常规范兼容**：返回类型通常需要相同，但在某些情况下（协变返回类型）可以不同；异常规范也需要兼容。
6. **引用限定符相同（C++11 新增）**：成员函数引用限定符可以将函数的调用限制为左值或右值。如果基类虚函数有引用限定符，派生类重写时必须使用相同的引用限定符。

·Decorative overriding functions override。·

成员函数引用限定符使得有可能以不同的方式对待左值和右值对象（\*this）。

图形用户界面, 文本, 应用程序

描述已自动生成

第13条：与迭代器相比，更喜欢const\_iterators。

**C++11 对常量迭代器支持的改进**

* **易于获取**：C++11 引入了容器的成员函数 cbegin 和 cend，即使对于非常量容器，也能方便地获取常量迭代器。例如 values.cbegin() 和 values.cend() 可以直接得到指向 std::vector<int> values 容器的常量迭代器。
* **易于使用**：STL 中使用迭代器指定位置的成员函数（如 insert 和 erase）开始支持常量迭代器，使得代码编写更加方便。如修改后的代码可以直接将 std::find 返回的常量迭代器用于 insert 操作。

·更喜欢const\_iterators而不是iterators。

·在最大通用代码中，优选开始、结束、r开始等的非成员版本，相对于其成员函数。

第14条：声明函数不执行，除非它们不会发出异常。.

声明为 noexcept 的函数应该确保在任何情况下都不会抛出异常，否则程序会突然终止，可能导致资源泄漏等问题。因此，在使用 noexcept 时需要谨慎，只有当你确定函数不会抛出异常时，才应该使用该声明。

如何确保一个函数在任何情况下都不会抛出异常？

在 C++ 中，是否有办法捕捉 `std::terminate` 引发的异常？

除了 `noexcept`，C++ 中还有哪些方式可以保证函数的异常安全性？

采用了 “能移动就移动，必须复制就复制” 的策略，

只有当知道移动操作不会抛出异常时，这些函数才会在 C++11 中用移动操作调用替代 C++98 中的复制操作调用。

类似地，交换两个包含 Widget 的 std::pair 对象是否为 noexcept 取决于 Widget 的 swap 操作是否为 noexcept。一般来说，只有当较低层级组件的交换操作是 noexcept 时，较高层级数据结构的交换操作才能是 noexcept，这一事实应该促使你尽可能提供 noexcept 的 swap 函数。

· noexcept是函数接口的一部分，这意味着调用者可能依赖于它。· noexcept函数比非noexcept函数更容易优化。noexcept对于移动操作、交换、内存释放函数和析构函数特别有价值。·大多数函数都是异常中性的，而不是noexcept。

第15条：尽可能使用constexpr。

constexpr 函数不需要产生常量结果或者在编译时就已知的结果，这其实是件好事！

但我们先从 constexpr 对象说起。实际上，这类对象是常量，而且它们的值确实在编译时就已知。（从技术上讲，它们的值是在翻译阶段确定的，而翻译不仅包括编译，还包括链接。不过，除非你是在编写 C++ 编译器或链接器，否则这对你没有影响，所以你可以放心地编程，就好像 constexpr 对象的值是在编译时确定的一样。）

文本, 信件

描述已自动生成

**constexpr 函数的特性**

* **编译时和运行时计算**：constexpr 函数根据传入参数是否为编译时常量，决定是在编译时还是运行时计算结果。如果参数是编译时常量，结果在编译时计算；如果参数在运行时才确定，函数像普通函数一样在运行时计算结果。这使得一个函数可以同时适用于编译时和运行时的场景，无需编写两个功能相同但适用场景不同的函数。

constexpr对象是const，并且使用编译期间已知的值进行初始化。

· constexpr函数在使用其值在编译期间已知的参数调用时可以产生编译时结果。·

constexpr对象和函数可以在比非constexpr对象和函数更宽范围的上下文中使用。constexpr是对象或函数接口的一部分。

条款16：使const成员函数线程安全。

使const成员函数线程安全，除非你确定它们永远不会在并发上下文中使用。使用std：：原子变量可能比互斥体提供更好的性能，但它们只适合于操作单个变量或内存位置。

第17项：理解特殊成员函数的生成。

 “三 / 五法则”（这里原文 Rule of Three 通常翻译为三法则，结合后续 C++11 内容准确说应是三 / 五法则）的准则。三法则指出，如果你声明了拷贝构造函数、拷贝赋值运算符或析构函数中的任何一个，你应该声明全部三个。

* 特殊成员函数是指编译器可能自行生成的那些函数：默认构造函数、析构函数、复制操作（复制构造函数和复制赋值运算符）以及移动操作（移动构造函数和移动赋值运算符）。
* 仅当类中没有显式声明移动操作、复制操作和析构函数时，编译器才会生成移动操作（移动构造函数和移动赋值运算符）。
* 仅当类中没有显式声明复制构造函数时，编译器才会生成复制构造函数；如果类声明了移动操作，则复制构造函数会被删除。仅当类中没有显式声明复制赋值运算符时，编译器才会生成复制赋值运算符；如果类声明了移动操作，则复制赋值运算符会被删除。在显式声明了复制操作或析构函数的类中生成复制操作已被弃用。
* 成员函数模板绝不会抑制特殊成员函数的生成。

第4章聪明的指针

Item 18：使用std：：unique\_ptr进行独占所有权资源管理。

std：：unique\_ptr是一个小型、快速、只移动的智能指针，用于管理具有独占所有权语义的资源

。·默认情况下，资源销毁通过删除进行，但可以指定自定义删除器。有状态删除器和作为删除器的函数指针增加了std：：unique\_ptr对象的大小。

·将std：：unique\_ptr转换为std：：shared\_ptr很容易。

自定义删除器是一个可调用对象（函数、函数对象或 lambda 表达式），它接受一个指向要删除对象的指针作为参数，并负责执行必要的清理操作。当 std::unique\_ptr 的生命周期结束时，它会调用自定义删除器来释放所管理的资源。

Item 19：使用std：：shared\_ptr进行共享所有权资源管理。

std::shared\_ptr 的大小是原始指针的两倍，

引用计数的内存必须动态分配。在堆中，方便所以指针访问。、

引用计数的递增和递减操作必须是原子的

· std：：shared\_ptrs为任意资源的共享生存期管理提供了接近垃圾收集的便利。

·与std：：unique\_ptr相比，std：：shared\_ptr对象通常是std：：unique\_ptr对象的两倍大，会产生控制块开销，并且需要原子引用计数操作。

·默认的资源销毁是通过删除，但支持自定义删除器。

删除器的类型对std：：shared\_ptr的类型没有影响。·避免从原始指针类型的变量创建std：：shared\_ptrs。

Item 20：使用std：：weak\_ptr作为std：：shared\_ptr类型的指针。

在一个需要缓存对象的工厂函数中，std::unique\_ptr 作为返回类型不合适，因为缓存需要能够检测自身的指针何时变成悬空指针。std::weak\_ptr 可以满足这个需求，因为它能够检测所指向的对象是否已被销毁。

·使用std：：weak\_ptr作为std：：shared\_ptr类型的指针。

std：：weak\_ptr的潜在用例包括缓存、观察者列表和防止std：：shared\_ptr循环。

第21条：首选std：：make\_unique和std：：make\_shared，而不是直接使用new。

·与直接使用new相比，make函数消除了源代码重复，提高了异常安全性，并且对于std：：make\_shared和std：：allocate\_shared，生成的代码更小更快。

·不适合使用make函数的情况包括需要指定自定义删除器和希望传递花括号初始化器。

·对于std：：shared\_ptrs，不建议使用make函数的其他情况包括（1）具有自定义内存管理的类和（2）具有内存问题的系统，非常大的对象，以及std：：weak\_ptrs比相应的std：：shared\_ptrs寿命长。

第22条：当使用Pimpl习语时，在实现文件中定义特殊的成员函数。

Pimpl Idiom通过减少类客户端和类实现之间的编译依赖性来减少构建时间。

·对于std：：unique\_ptr pImpl指针，在类头中声明特殊成员函数，但在实现文件中实现它们。

即使默认函数实现是可接受的，也要这样做。以上建议适用于std：：unique\_ptr，但不适用于std：：shared\_ptr。

右值引用、移动语义和完美转发

完美转发使得编写函数模板成为可能，这些函数模板接受任意参数并将其转发给其他函数，以便目标函数接收与传递给转发函数的参数完全相同的参数

std：：move不会移动任何东西，而完美的转发是不完美的。移动操作并不总是比复制更便宜;即使是这样，也不总是像您期望的那样便宜;并且它们并不总是在移动有效的上下文中调用。构造“type&&”并不总是表示右值引用。

在函数 void f(Widget&& w); 中，w 的类型是右值引用 Widget&&。右值引用的初衷是用来绑定右值，以实现移动语义等特性。然而，一旦右值被传递给函数参数 w，w 本身就变成了一个左值。这是因为 w 有了一个名字，并且有确定的内存地址，满足左值的定义。

Item 23：理解std：：move和std：：forward。

std::move 并不移动任何东西，std::forward 也不转发任何东西

std::forward 是一个条件转换，常用于函数模板中，将通用引用参数转发给其他函数。在 logAndProcess 函数模板中：

* 当调用 logAndProcess(w) 传递左值时，std::forward<T>(param) 会将 param 作为左值转发给 process 函数，从而调用 process(const Widget& lvalArg)。
* 当调用 logAndProcess(std::move(w)) 传递右值时，std::forward<T>(param) 会将 param 作为右值转发给 process 函数，从而调用 process(Widget&& rvalArg)。

· std：：move执行一个无条件转换为右值。

就其本身而言，它不会移动任何东西。

· std：：forward仅当参数绑定到右值时才将参数转换为右值。

·std：：move和std：：forward在运行时都不做任何事情。·const对象上的移动请求被视为复制请求。

项目24：区分通用引用和右值引用。

·如果一个函数模板参数具有T&&类型，用于推导类型T，或者如果一个对象使用auto&&声明，则该参数或对象是通用引用。

·如果类型声明的形式不是精确的type&&，或者如果类型推导没有发生，type&&表示右值引用。

·如果使用右值初始化，则通用引用对应于右值引用。它们对应于左值引用，如果它们是用左值初始化的。

Item 25：在右值引用上使用std：：move，在通用引用上使用std：：forward。

·在最后一次使用时将std：：move应用于右值引用，将std：：forward应用于通用引用。

·对从按值返回的函数返回的右值引用和通用引用做同样的事情。

永远不要将std：：move或std：：forward应用于本地对象，如果它们有资格进行返回值优化。

项目26：避免在通用引用上过载。

接受通用引用的函数可以对几乎任何类型的参数进行精确匹配，这种特性使得它在重载场景中会匹配比开发者预期更多的参数类型。这就导致了在结合重载和通用引用时，很容易出现意外的匹配结果，从而引发编译错误或不符合预期的行为。所以，在实际编程中，一般应尽量避免将重载和通用引用结合使用。

·在通用引用上重载几乎总是会导致通用引用重载比预期更频繁地被调用。

·完美转发构造函数特别有问题，因为它们通常比非const左值的复制构造函数更好地匹配，并且它们可以劫持派生类对基类复制和移动构造函数的调用。

项目27：熟悉通用引用重载的替代方案。

**标签分发的原理**

重载函数的调用解析是综合考虑所有参数和实参的匹配情况。通用引用虽然通常能精确匹配传入的参数，但如果参数列表中还有其他非通用引用参数，这些非通用引用参数的匹配情况可以影响最终的重载选择。标签分发就是利用这一点，通过引入一个额外的参数（标签）来控制重载选择。



图形用户界面, 文本, 应用程序

描述已自动生成

用标签分发，判断是否为具体的类型，如果为具体的类型，int这种，则调用重载的int版本，如果不是，则调用通用引用版本

**std::enable\_if 的作用**

std::enable\_if 是一种可以控制模板是否启用的工具。默认情况下，所有模板都是启用的，但使用 std::enable\_if 的模板只有在满足其指定的条件时才会被启用。如果条件不满足，编译器会表现得好像这个模板不存在一样，这一机制基于 SFINAE（Substitution Failure Is Not An Error，替换失败不是错误）原则。

标签分发和约束模板适用性

通用引用和重载组合的替代方案包括使用不同的函数名，通过lvaluereference-to-const传递参数，通过值传递参数，以及使用标记分派。

通过std：：enable\_if约束模板允许同时使用通用引用和重载，但它控制编译器可以使用通用引用重载的条件。

·通用参考参数通常具有效率优势，但通常具有可用性劣势。

第28条：理解引用折叠。

例如，如果 T 被推导为左值引用，std::forward 会将参数作为左值转发；如果 T 被推导为非引用类型（对应右值传入），std::forward 会将参数作为右值转发。

引用会根据以下规则折叠成单个引用：

* 如果其中一个引用是左值引用，结果是一个左值引用。
* 否则（即如果两个都是右值引用），结果是一个右值引用。
* 引用折叠发生在四种上下文中：模板实例化、auto 类型生成、typedef 和别名声明的创建与使用，以及 decltype。
* 当编译器在引用折叠的上下文中生成引用的引用时，结果会变成单个引用。如果原始引用中有一个是左值引用，那么结果就是左值引用；否则就是右值引用。
* 通用引用是在类型推导能够区分左值和右值，并且会发生引用折叠的上下文中的右值引用。

条款29：假设移动操作不存在，不便宜，也不使用。

因此，有几种情况 C++11 的移动语义对你没有帮助：

* **没有移动操作**：要进行移动的对象不提供移动操作。因此，移动请求就变成了复制请求。
* **移动不快**：要进行移动的对象的移动操作并不比其复制操作快。
* **移动不可用**：进行移动操作的上下文要求移动操作不抛出异常，但该操作没有声明为 noexcept。

还有一种情况也值得一提，在这种情况下移动语义不会带来效率提升：

* **源对象是左值**：极少数情况除外（见条款 25），只有右值才能作为移动操作的源对象。

·假设移动操作不存在，不便宜，也不使用。·

在具有已知类型或支持移动语义的代码中，不需要假设。

在处理字符串时，通常有两种常见的存储方式：一是将字符串数据存储在堆上，通过指针来管理；二是直接将字符串数据存储在对象内部。堆存储适合存储较长的字符串，因为它可以动态分配足够的内存。但对于短字符串，如果每次都在堆上分配内存，会带来额外的开销，包括内存分配和释放的系统调用、内存碎片等问题。

SSO 的核心思想是，对于较短的字符串，不使用堆存储，而是直接将其存储在 std::string 对象内部的一个固定大小的缓冲区中。这样可以避免堆内存分配和释放的开销，提高性能。当字符串长度超过这个固定大小的缓冲区时，再使用堆存储。

第30条：熟悉完美转发失败案例。

* **无法推导类型**：在 fwd({ 1, 2, 3 }) 中，由于 fwd 的参数没有声明为 std::initializer\_list，根据 C++ 标准，这是一个 “非推导上下文”，编译器不能为 { 1, 2, 3 } 推导类型，所以编译失败。
* **推导错误类型**：当 f 是重载函数时，如果 fwd 对参数类型推导不正确，可能会导致 fwd 内部调用的 f 的重载版本与直接调用 f 时不同，从而产生不同的行为。

·模板类型推演失败或推演错误类型时，完美转发失败。·导致完美转发失败的参数类型有花括号初始化器、表示为0或NULL的空指针、仅声明整型常量静态数据成员、模板和重载函数名以及位字段。

条款31：避免默认捕获模式。

·默认的按引用捕获可能会导致悬空引用。·

默认的按值捕获容易受到悬空指针的影响（尤其是this），它误导性地暗示了指针是自包含的。

Item 32：使用init capture将对象移动到闭包中。这里需要注意，“=” 左右两边的作用域不同，左边是闭包类的作用域，右边是 Lambda 表达式定义的作用域。

使用C++14的init capture将对象移动到闭包中。在C++11中，通过手写类或std：：bind模拟初始化捕获。

Item 33：在auto&& parameters上使用decltype来std：：forward它们。

泛型 lambda 表达式 —— 即在参数声明中使用 auto 的 lambda 表达式。



Item 34: Prefer lambdas to std::bind.

Lambdas比std：：bind更易读，更有表现力，可能更有效。

仅在C++11中，std：：bind可能对实现移动捕获或使用模板化函数调用操作符绑定对象有用。

并发API

并发 C++ 软件中 “线程” 的三种含义：

* **硬件线程**：是实际执行计算的线程。当代计算机体系结构每个 CPU 核心提供一个或多个硬件线程。
* **软件线程**（也称为操作系统线程或系统线程）：是操作系统管理的跨所有进程的线程，并将它们调度到硬件线程上执行。通常可以创建比硬件线程更多的软件线程，因为当一个软件线程被阻塞时（例如，在进行 I/O 操作或等待互斥锁或条件变量时），通过执行其他未被阻塞的线程可以提高吞吐量。
* **std::thread 对象**：是 C++ 进程中的对象，作为底层软件线程的句柄。一些 std::thread 对象表示 “空” 句柄，即不对应任何软件线程，因为它们处于默认构造状态（因此没有要执行的函数）、被移动过（被移动到的 std::thread 对象随后作为底层软件线程的句柄）、已经被合并（它们要运行的函数已经完成）或者已经被分离（它们与底层软件线程之间的连接已经被切断）。

·std：：thread API没有提供从异步运行的函数获取返回值的直接方法，如果这些函数抛出，程序将被终止。

·基于线程的编程需要手动管理线程占用、超额订阅、负载平衡和对新平台的适应。

·通过std：：bloc使用默认启动策略进行基于任务的编程，可以为您处理大部分问题。

条款 36：如果必须异步执行，指定 std::launch::async

* std::async 的默认启动策略允许任务以异步和同步两种方式执行。
* 这种灵活性在访问线程局部变量（thread\_local）时会导致不确定性，意味着任务可能永远不会执行，并且会影响基于超时的等待调用的程序逻辑。
* 如果必须要异步执行任务，请指定 std::launch::async。

Item 37：使std：：threads在所有路径上都不可连接。.

·使std：：threads在所有路径上不可连接。·销毁时加入可能导致难以调试的性能异常。·销毁时调试可能导致难以调试的未定义行为。Deconstructstd：：thread对象在数据成员列表中的最后一个。

* 通常情况下，std::future 的析构函数仅销毁 std::future 对象的数据成员。
* 对于通过 std::async 启动的非延迟任务，指向其共享状态的最后一个 std::future 对象在析构时会阻塞，直到该任务完成。

条款39：考虑一次性事件通信的无效未来。

对于简单的事件通信，基于condvar的设计需要一个超级互斥体，对检测和反应任务的相对进度施加约束，并需要反应任务来验证事件是否发生。

使用标志的设计避免了这些问题，但基于轮询，而不是阻塞。

·condvar和flag可以一起使用，但由此产生的通信机制有点僵硬。

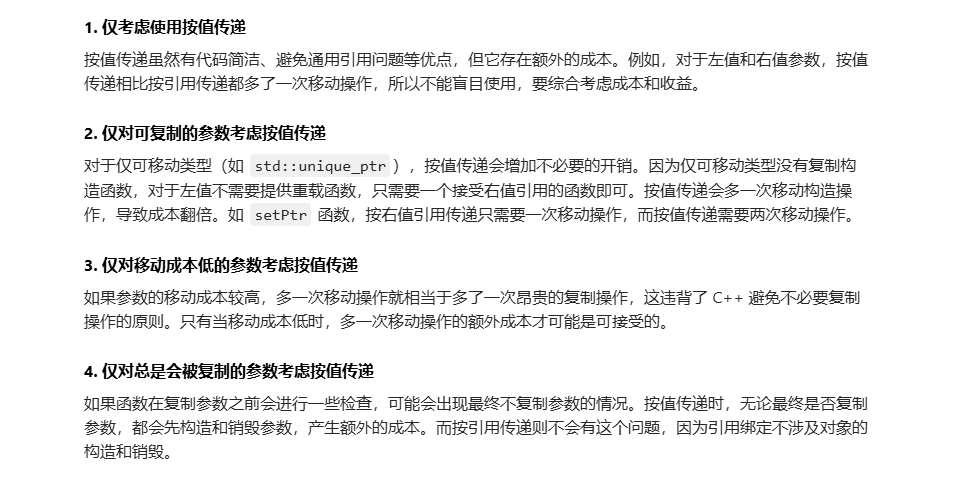
使用std：：promise和futures可以避免这些问题，但是这种方法使用堆内存来共享状态，并且仅限于一次性通信。

std：：atomic用于从多个线程访问数据，而不使用互斥锁。它是一个编写并发软件的工具。

· volatile是针对内存的，其中读取和写入不应该被优化掉。它是一种处理特殊记忆的工具。

这里所说的通用技术是按值传递，通用特性是就地构造（emplace）。

条款41：考虑通过值传递可复制的参数，这些参数移动起来很便宜，并且总是被复制。



·对于可复制的、易于移动的参数，总是复制，按值传递可能与按引用传递一样有效，更容易实现，并且可以生成更少的目标代码。

·对于左值参数，按值传递（即，复制构造）之后进行移动赋值可能比通过引用传递之后进行复制赋值要昂贵得多。

·按值传递受到切片问题的影响，因此它通常不适合基类参数类型。

项目42：考虑安置而不是插入。

·原则上，安置职能有时应比插入职能更有效，而且绝不应效率更低。·在实践中，当（1）被添加的值被构造到容器中，而不是赋值时，它们最有可能更快;

（2）传递的参数类型与容器所持有的类型不同;（

3）容器不会拒绝被添加的值，因为它是重复的。

·安置函数可以执行将被插入函数拒绝的类型转换。

通常情况下 push\_back 可能会涉及临时对象的复制或移动，而 emplace\_back 可以直接在容器中构造元素，避免临时对象的产生

**原则上置入函数的效率优势**

从理论角度来看，置入函数利用完美转发机制，直接使用传入的参数在容器内部构造对象。这意味着在合适的场景下，它可以避免插入函数可能需要的临时对象的创建和销毁过程，从而节省时间和资源。因此，置入函数有时能展现出比插入函数更高的效率，而且在任何情况下，其效率都不应该低于插入函数。

**实际中置入函数更快的情况**

1. **构造而非赋值**：当新元素是在容器中原本没有对象的位置进行构造时，置入函数能直接利用传入的参数完成构造。例如，向 std::vector 的末尾添加元素，emplace\_back 可以直接在末尾位置构造对象。但如果是在已有对象的位置进行赋值操作，可能需要创建临时对象作为赋值的源，这样置入函数的优势就会减弱。像基于节点的容器大多使用构造方式添加新值，非基于节点的容器如 std::vector 的 emplace\_back 和 std::deque 的 emplace\_front 也倾向于构造操作，此时置入函数效率更高。
2. **参数类型不同**：若传递的参数类型与容器元素类型不一致，插入函数通常需要创建临时对象来匹配容器元素的类型，这会带来额外的开销。而置入函数可以直接使用这些参数在容器内构造对象，避免了临时对象的创建和销毁。例如，向 std::vector<std::string> 中添加字符串字面量时，emplace\_back 可以直接用字符串字面量构造 std::string 对象，而 push\_back 可能需要先创建一个临时的 std::string 对象。
3. **无重复值拒绝**：对于禁止重复值的容器，置入函数为了检查重复值，通常会创建一个新节点。若要添加的值已存在，该节点会被销毁，造成额外开销。当容器允许重复值或添加的大多是唯一值时，置入函数就不会有这种额外的节点创建和销毁成本，性能更优。

**置入函数的类型转换特性**

置入函数使用直接初始化，允许调用显式构造函数。这使得它在某些情况下可以执行插入函数会拒绝的类型转换。例如，对于 std::vector<std::regex> 容器，使用 push\_back 尝试添加 nullptr 会编译失败，因为 std::regex 接受 const char\* 指针的构造函数是显式的，push\_back 使用复制初始化，不允许这种隐式转换。但使用 emplace\_back 时，由于它使用直接初始化，代码可以编译通过，不过传入 nullptr 会导致未定义行为。这表明在使用置入函数时，需要特别注意传递的参数，确保不会因为这种特性引入潜在的错误。