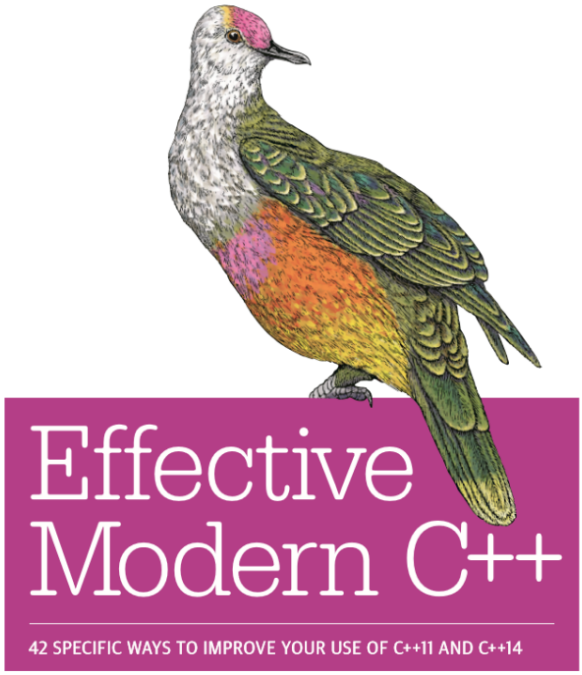
《Effective Modern C++ 》翻译



> ! 2017.10开始更新
>
> ! 标注“已修订”的章节表示已经没有大致的错误
> ! 我没有版权，我没有版权，我没有版权
>
> ！ 本书要求读者具有C++基础
>
> ！ 未翻译的条款名称现在直译，翻译时可能适当修改
>
> ! PDf版本下载,仅供翻译参考

目录

2. Item 8:优先考虑nullptr而非0和NULL
3. Item 9:优先考虑别名声明而非typedefs
4. Item 10:优先考虑限域枚举而非未限域枚举 **已修订**
5. Item 11:优先考虑使用deleted函数而非使用未定义的私有声明
6. Item 12:使用override声明重载函数
7. Item 13:优先考虑const_iterator而非iterator
8. Item 14:如果函数不抛出异常请使用noexcept
9. Item 15:尽可能的使用constexpr
10. Item 16:让const成员函数线程安全 **由 @windski贡献**
11. Item 17:理解特殊成员函数函数的生成

4. 智能指针

1. Item 18:对于独占资源使用std::unique_ptr **@wendajiang更新完成**
2. Item 19:对于共享资源使用std::shared_ptr **已修订**
3. Item 20:像std::shared_ptr一样使用std::weak_ptr可能造成dangle **更新完成**
4. Item 21:优先考虑使用std::make_unique和std::make_shared而非new **由 @pusidun贡献**
5. Item 22:当使用Pimpl惯用法，请在实现文件中定义特殊成员函数 **由 @BlurryLight贡献**

5. 右值引用，移动语意，完美转发

1. Item 23:理解std::move和std::forward **由 @BlurryLight贡献**
2. Item 24:区别通用引用和右值引用 **由 @BlurryLight贡献**
3. Item 25:对于右值引用使用std::move，对于通用引用使用std::forward**由 @wendajiang贡献**
4. Item 26:避免重载通用引用**由 @wendajiang贡献**
5. Item 27:熟悉重载通用引用的替代品**由 @wendajiang贡献**
6. Item 28:理解引用折叠**由 @wendajiang贡献**
7. Item 29:认识移动操作的缺点 **由 @wendajiang贡献**
8. Item 30:熟悉完美转发失败的情况**由 @wendajiang贡献**

6. Lambda表达式

1. Item 31:避免使用默认捕获模式 **由 @LucienXian贡献**
2. Item 32:使用初始化捕获来移动对象到闭包中 **由 @LucienXian贡献**
3. Item 33:对于std::forward的auto&&形参使用decltype **由 @LucienXian贡献**
4. Item 34:优先考虑lambda表达式而非std::bind **由 @LucienXian贡献**

7. 并发API

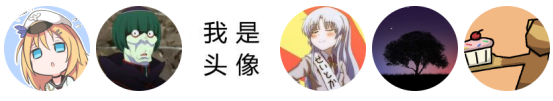
1. Item 35:优先考虑基于任务的编程而非基于线程的编程 **由 @wendajiang贡献**
2. Item 36:如果有异步的必要请指定std::launch::threads **由 @wendajiang贡献**
3. Item 37:从各个方面使得std::threads unjoinable **由 @wendajiang贡献**
4. Item 38:关注不同线程句柄析构行为 **由 @wendajiang贡献**
5. Item 39:考虑对于单次事件通信使用void **由 @wendajiang贡献**
6. Item 40:对于并发使用std::atomic，volatile用于特殊内存区 **由 @wendajiang贡献**

8. 微调

1. Item 41:对于那些可移动总是被拷贝的形参使用传值方式 **由 @wendajiang贡献**
2. Item 42:考虑就地创建而非插入 **由 @wendajiang贡献**

贡献者

感谢所有参与翻译/勘误/建议的贡献者们~



赞助翻译

[Become a backer]

Become a
Backer

CHAPTER 1 Deducing Types

C++98有一套用于模板类型推导的规则，C++11修改了其中的一些规则并为**auto**和**decltype**添加了新的规则。类型推导的广泛应用让我们不必再输入那些明显多余的类型，它让C++程序更具适应性，因为在源代码某处修改类型会通过类型推导自动传播到其它地方。但是类型推导也会让代码更复杂，因为由编译器进行的类型推导并不总是如我们期望的那样进行。

如果对于类型推导操作没有一个扎实的理解，要想写出有现代感的C++程序是不可能的。类型推导随处可见：在函数模板调用中，在**auto**出现的地方，在**decltype**表达式出现的地方，以及C++14的**decltype(auto)** 中。

这一章的内容是每个C++程序员都应该掌握的知识。它解释了模板类型推导是如何工作的，**auto**是如何依赖模板类型推导的，以及**decltype**是如何按照它自己那套独特的规则工作的。它甚至解释了你该如何强制编译器产生他进行类型推导的结果，这能让你确认编译器的类型推导是否按照你期望的那样进行。

Item1 :Understand template type deduction

条款一:理解模板类型推导

对于一个复杂系统的用户来说很多时候他们最关心的是它做了什么而不是它怎么做的。在这一点上C++中的模板类型推导表现得非常出色。数百万的程序员只需要向模板函数传递实参就能通过编译器的类型推导获得令人满意的结果，尽管他们中的大多数对于传递给函数的那些实参是如何引导编译器进行类型推导的只能给出非常模糊的描述，而且还是在被逼无奈的情况。

如果那些人中包括你，我有一个好消息和一个坏消息。好消息是现在C++最重要最吸引人的特性**auto**是建立在模板类型推导的基础上的，如果你熟悉C++98的模板类型推导，那么你不必害怕C++11的**auto**。坏消息是虽然**auto**是建立在模板类型推导的基础上，但是在某些情况下**auto**不如模板类型推导那么直观容易理解。这个条款便包含了你需要知道的关于模板类型推导的全部内容：

如果你不介意浏览少许伪代码，考虑这样一个函数模板：

```
template<typename T>
void f(ParamType param);
```

它的调用看起来像这样

```
f(expr);    //使用表达式调用f
```

在编译期间，编译器使用expr进行两个类型推导：一个是针对T的，另一个是针对ParamType的。这两个类型通常是不同的，因为ParamType包括了const和引用的修饰。举个例子，如果模板这样声明：

```
template<typename T>
void f(const T& param);
```

然后这样进行调用

```
int x = 0;
f(x);   //用一个int类型的变量调用f
```

T被推导为**int**，ParamType却被推导为**const int&**

我们可能很自然的期望T和传递进函数的参数是相同的类型，在上面的例子中，事实就是那样，**x**是**int**，T是**expr**的类型即**int**。但有时情况并非总是如此，T的推导不仅取决于**expr**的类型，也取决于**ParamType**的类型。这里有三种情况：

- ParamType是一个指针或引用，但不是通用引用（关于通用引用请参见Item24。在这里你只需要知道它存在，而且不同于左值引用和右值引用）
- ParamType一个通用引用
- ParamType既不是指针也不是引用

我们下面将分成三个情景来讨论这三种情况，每个情景的都基于我们之前给出的模板：

```
template<typename T>
void f(ParamType param);

f(expr);    //从expr中推导T和ParamType
```

情景一：ParamType是一个指针或引用但不是通用引用

最简单的情况是**ParamType**是一个指针或者引用但非通用引用，也就是我们这个情景讨论的内容。在这种情况下，类型推导会这样进行：

1. 如果expr的类型是一个引用，忽略引用部分
2. 然后剩下的部分决定T，然后T与形参匹配得出最终ParamType

举个例子，如果这是我们的模板

```
template<typename T>
void f(T & param);  //param是一个引用
```

我们声明这些变量：

```
int x=27;       //x是int
const int cx=x;     //cx是const int
const int & rx=cx;  //rx是指向const int的引用
```

当把这些变量传递给f时类型推导会这样

```
f(x);       //T是int，param的类型是int&
f(cx);      //T是const int，param的类型是const int &
f(rx);      //T是const int，param的类型是const int &
```

在第二个和第三个调用中，注意因为cx和rx被指定为**const**值，所以T被推导为**const int**，从而产生了**const int&**类型的**param**。这对于调用者来说很重要，当他们传递一个**const**对象给一个引用类型的参数时，他们传递的对象保留了常量性。这也是为什么向**T&**类型的参数传递**const**对象是安全的：对象T的常量性会被保留为T的一部分。

在第三个例子中，注意即使rx的类型是一个引用，T也会被推导为一个非引用，这是因为如上面提到的如果**expr**的类型是一个引用，将忽略引用部分。

这些例子只展示了左值引用，但是类型推导会如左值引用一样对待右值引用。通常，右值只能传递给右值引用，但是在模板类型推导中这种限制将不复存在。

情景二：ParamType一个通用引用

如果**ParamType**是一个通用引用那事情就比情景一更复杂了。如果**ParamType**被声明为通用引用（在函数模板中假设有一个模板参数**T**,那么通用引用就是**T&&**),它们的行为和**T&**大不相同，完整的叙述请参见Item24，在这有些最必要的你还是需要知道：

- 如果**expr**是左值，T和ParamType都会被推导为左值引用。这非常不寻常，第一，这是模板类型推导中唯一一种T和ParamType都被推导为引用的情况。第二，虽然ParamType被声明为右值引用类型，但是最后推导的结果它是左值引用。
- 如果**expr**是右值，就使用**情景**一的推导规则

举个例子：

```
template<typename T>
void f(T&& param);       //param现在是一个通用引用类型

int x=27;                //如之前一样
const int cx=x;          //如之前一样
const int & rx=cx;       //如之前一样

f(x);                //x是左值，所以T是int&
                     //param类型也是int&

f(cx);               //cx是左值，所以T是const int &
                     //param类型也是const int&

f(rx);               //rx是左值，所以T是const int &
                     //param类型也是const int&

f(27);               //27是右值，所以T是int
                     //param类型就是int&&
```

Item24详细解释了为什么这些例子要这样做。这里关键在于类型推导对于通用引用是不同于普通的左值或者右值引用。比如，当通用引用被使用时，类型推导会区分左值实参和右值实参，但是**情况**一就不会。

情景三：ParamType既不是指针也不是引用

当**ParamType**既不是指针也不是引用时，我们通过传值（pass-by-value）的方式处理：

```
template<typename T>
void f(T param);             //以传值的方式处理param
```

这意味着无论传递什么param都会成为它的一份拷贝——一个完整的新对象。事实上param成为一个新对象这一行为会影响T如何从expr中推导出结果。

1. 和之前一样，如果expr的类型是一个引用，忽略这个引用部分
2. 如果忽略引用之后expr是一个const，那就再忽略const。如果它是volatile，也会被忽略（volatile不常见，它通常用于驱动程序的开发中。关于volatile的细节请参见Item40)

因此

```
int x=27;               //如之前一样
const int cx=x;         //如之前一样
const int & rx=cx;      //如之前一样

f(x);               //T和param都是int
f(cx);              //T和param都是int
f(rx);              //T和param都是int
```

注意即使**cx**和**rx**表示const值，param也不是**const**。这是有意义的。param是一个拷贝自**cx**和**rx**且现在独立的完整对象。具有常量性的**cx**和**rx**不可修改并不代表param也是一样。这就是为什么expr的常量性或易变性（volatileness)在类型推导时会被忽略：因为**expr**不可修改并不意味着他的拷贝也不能被修改。

认识到只有在传值给形参时才会忽略常量性和易变性这一点很重要，正如我们看到的，对于形参来说指向const的指针或者指向const的引用在类型推导时const都会被保留。但是考虑这样的情况，expr是一个const指针，指向const对象，expr通过传值传递给**param**：

```
template<typename T>
void f(T param);        //传值

const char* const ptr = //ptr是一个常量指针，指向常量对象
" Fun with pointers";
```

在这里，解引用符号（*）的右边的const表示ptr本身是一个const：ptr不能被修改为指向其它地址，也不能被设置为null（解引用符号左边的const表示ptr指向一个字符串，这个字符串是const，因此字符串不能被修改）。当ptr作为实参传给f，像这种情况，ptr自身会传值给形参，根据类型推导的第三条规则，**ptr**自身的常量性将会被省略，所以param是**const char*** 。也就是说一个常量指针指向const字符串，在类型推导中这个指针指向的数据的常量性将会被保留，但是指针自身的常量性将会被忽略。

数组实参

上面的内容几乎覆盖了模板类型推导的大部分内容，但这里还有一些小细节值得注意，比如在模板类型推导中指针不同于数组，虽然它们两个有时候是完全等价的。关于这个等价最常见的例子是在很多上下文中数组会退化为指向它的第一个元素的指针，比如下面就是允许的做法：

```
const char name[] = "J. P. Briggs";     //name的类型是const char[13]

const char * ptrToName = name;      //数组退化为指针
```

在这里**const char*** 指针**ptrToName**会由name初始化，而name的类型为**const char[13]**，这两种类型(**const char *** 和**const char[13]**)是不一样的，但是由于数组退化为指针的规则，编译器允许这样的代码。

但要是一个数组传值给一个模板会怎样？会发生什么？

```
template<typename T>
void f(T param);

f(name);        //对于T和param会产生什么样的类型
```

我们从一个简单的例子开始，这里有一个函数的形参是数组，是的，这样的语法是合法的：

```
void myFunc(int param[]);
```

但是数组声明会被视作指针声明，这意味着myFunc的声明和下面声明是等价的：

```
void myFunc(int *param);    //同上
```

这样的等价是C语言的产物，C++又是建立在C语言的基础上，它让人产生了一种数组和指针是等价的的错觉。

因为数组形参会视作指针形参，所以传递给模板的一个数组类型会被推导为一个指针类型。这意味着在模板函数f的调用中，它的模板类型参数T会被推导为**const char***：

```
f(name);    //name是一个数组，但是T被推导为const char *
```

但是现在难题来了，虽然函数不能接受真正的数组，但是可以接受指向数组的引用！所以我们修改f为传引用：

```
template<typename T>
void f(T& param);
```

我们这样进行调用

```
f(name);    //传数组
```

T被推导为了真正的数组！这个类型包括了数组的大小，在这个例子中T被推导为**const char[13]**，param则被推导为**const char(&)[13]**。是的，这种语法看起来简直有毒，但是知道它将会让你在关心这些问题的人的提问中获得大神的称号。

有趣的是，对模板函数声明为一个指向数组的引用使得我们可以在模板函数中推导出数组的大小：

```
//在编译期间返回一个数组大小的常量值（
//数组形参没有名字，因为我们只关心数组
//的大小）
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept
{
    return N;
}
```

在Item15提到将一个函数声明为constexpr使得结果在编译期间可用。这使得我们可以用一个花括号声明一个数组，然后第二个数组可以使用第一个数组的大小作为它的大小，就像这样：

```
int keyVals[] = {1,3,5,7,9,11,22,25};   //keyVals有七个元素

int mappedVals[arraySize(keyVals)]; //mappedVals也有七个
```

当然作为一个现代C++程序员，你自然应该想到使用**std::array**而不是内置的数组：

```
std::array<int,arraySize(keyVals)> mappedVals;      //mappedVals的size为7
```

至于**arraySize**被声明为**noexcept**，会使得编译器生成更好的代码，具体的细节请参见Item14。

函数实参

在C++中不止是数组会退化为指针，函数类型也会退化为一个函数指针，我们对于数组的全部讨论都可以应用到函数来：

```
void someFunc(int, double); //someFunc是一个函数，类型是void(int,double)

template<typename T>
void f1(T param);        //传值

template<typename T>
void f2(T & param);      //传引用

f1(someFunc);            //param被推导为指向函数的指针，类型是void(*)(int, double)
f2(someFunc);            //param被推导为指向函数的引用，类型为void(&)(int, bouel)
```

这个实际上没有什么不同，但是如果你知道数组退化为指针，你也会知道函数退化为指针。

这里你需要知道：**auto**依赖于模板类型推导，正如我在开始谈论的，在大多数情况下它们的行为很直接。在通用引用中对于左值的特殊处理使得本来很直接的行为变得有些污点，然而，数组和函数退化为指针把这团水搅得更浑浊。有时你只需要编译器告诉你推导出的类型是什么。这种情况下，翻到**item4**，它会告诉你如何让编译器这么做。

记住：

- 在模板类型推导时，有引用的实参会被视为无引用，他们的引用会被忽略
- 对于通用引用的推导，左值实参会被特殊对待
- 对于传值类型推导，实参如果具有常量性和易变性会被忽略
- 在模板类型推导时，数组或者函数实参会退化为指针，除非它们被用于初始化引用

Item 2:Understand auto type deduction

条款二:理解auto类型推导

如果你已经读过Item1的模板类型推导，那么你几乎已经知道了auto类型推导的大部分内容，至于为什么不是全部是因为这里有一个auto不同于模板类型推导的例外。但这怎么可能，模板类型推导包括模板，函数，形参，但是auto不处理这些东西啊。

你是对的，但没关系。auto类型推导和模板类型推导有一个直接的映射关系。它们之间可以通过一个非常规范非常系统化的转换流程来转换彼此。

在Item1中，模板类型推导使用下面这个函数模板来解释：

```
template<typename T>
void f(ParmaType param);     //使用一些表达式调用f
```

在f的调用中，编译器使用expr推导T和ParamType。当一个变量使用auto进行声明时，auto扮演了模板的角色，变量的类型说明符扮演了ParamType的角色。废话少说，这里便是更直观的代码描述，考虑这个例子：

```
auto x = 27;
```

这里x的类型说明符是auto，另一方面，在这个声明中：

```
const auto cx = x;
```

类型说明符是**const auto**。另一个：

```
const auto & rx=cx;
```

类型说明符是**const auto&**。在这里例子中要推导x rx cx的类型，编译器的行为看起来就像是认为这里每个声明都有一个模板，然后使用合适的初始化表达式进行处理：

```
template<typename T>     //理想化的模板用来推导x的类型
void func_for_x(T param);

func_for_x(27);

template<typename T>     //理想化的模板用来推导cx 的类型
void func_for_cx(const T param);

func_for_cx(x);

template<typename T>     //理想化的模板用来推导rx的类型
void func_for_rx(const T & param);

func_for_rx(x);
```

正如我说的，auto类型推导除了一个例外（我们很快就会讨论），其他情况都和模板类型推导一样。

Item1把模板类型推导分成三个部分来讨论ParamType在不同情况下的类型。在使用**auto**作为类型说明符的变量声明中，类型说明符代替了ParamType，因此Item1描述的三个情景稍作修改就能适用于auto：

- 类型说明符是一个指针或引用但不是通用引用
- 类型说明符一个通用引用
- 类型说明符既不是指针也不是引用

我们早已看过情景一和情景三的例子：

```
auto x = 27;            //情景三
const auto cx = x;      //情景三
const auto & rx=cx;     //情景一
```

Item1讨论并总结了数组和函数如何退化为指针，那些内容也同样适用于auto类型推导

```
const char name[] =     //name的类型是const char[13]
 "R. N. Briggs";

auto arr1 = name;       //arr1的类型是const char*
auto& arr2 = name;      //arr2的类型是const char(&)[13]

void someFunc(int,double);

auto func1=someFunc;    //func1的类型是void(int,double)
auto& func2 = someFunc; //func2的类型是void(&)(int,double)
```

就像你看到的那样auto类型推断和模板类型推导一样几乎一样的工作，它们就像一个硬币的两面。

讨论完相同点接下来就是不同点，前面我们已经说到auto类型推导和模板类型推导有一个例外使得它们的工作方式不同，接下来我们要讨论的就是那个例外。
我们从一个简单的例子开始，如果你想用一个int值27来声明一个变量，C++98提供两种选择：

```
int x1=27;
int x2(27);
```

C++11由于也添加了用于支持统一初始化（**uniform initialization**）的语法：

```
int x3={27};
int x47{27};
```

总之，这四种不同的语法只会产生一个相同的结果：变量类型为int值为27

但是Item5解释了使用auto说明符代替指定类型说明符的好处，所以我们应该很乐意把上面声明中的**int**替换为**auto**，我们会得到这样的代码：

```
auto x1=27;
auto x2(27);
auto x3={27};
auto x4{27};
```

这些声明都能通过编译，但是他们不像替换之前那样有相同的意义。前面两个语句确实声明了一个类型为int值为27的变量，但是后面两个声明了一个存储一个元素27的 **std::initializer_list<int>** 类型的变量。

```
auto x1=27;          //类型是int，值是27
auto x2(27);         //同上
auto x3={27};        //类型是std::initializer_list<int>,值是{27}
auto x4{27};         //同上
```

这就造成了auto类型推导不同于模板类型推导的特殊情况。当用auto声明的变量使用花括号进行初始化，auto类型推导会推导出auto的类型为 **std::initializer_list**。如果这样的一个类型不能被成功推导（比如花括号里面包含的是不同类型的变量），编译器会拒绝这样的代码！

```
auto x5={1,2,3.0};       //错误！auto类型推导不能工作
```

就像注释说的那样，在这种情况下类型推导将会失败，但是对我们来说认识到这里确实发生了两种类型推导是很重要的。一种是由于auto的使用：x5的类型不得不被推导，因为x5使用花括号的方式进行初始化，x5必须被推导为 **std::initializer_list**,但是 **std::initializer_list**是一个模板。**std::initializer_list**会被实例化，所以这里T也会被推导。 另一种推导也就是模板类型推导被归入第二种推导。在这个例子中推导之所以出错是因为在花括号中的值并不是同一种类型。

对于花括号的处理是auto类型推导和模板类型推导唯一不同的地方。当使用auto的变量使用花括号的语法进行初始化的时候，会推导出**std::initializer_list**的实例化，但是对于模板类型推导这样就行不通：

```
auto x={11,23,9};   //x的类型是std::initializer_list<int>

template<typename T>
void  f(T param);

f({11,23,9});       //错误！不能推导出T
```

然而如果指定T是**std::initializer**而留下未知T,模板类型推导就能正常工作：

```
template<typename T>
void f(std::initializer_list<T> initList);

f({11,23,9});       //T被推导为int，initList的类型被推导为std::initializer_list<int>
```

因此auto类型推导和模板类型推导的真正区别在于auto类型推导假定花括号表示**std::initializer_list**而模板类型推导不会这样（确切的说是不知道怎么办）。

你可能想知道为什么auto类型推导对于花括号和模板类型推导有不同的处理方式。我也想知道。哎，我至今没找到一个令人信服的解释。但是规则就是规则，这意味着你必须记住如果你使用auto声明一个变量，并用花括号进行初始化，auto类型推导总会得出std::initializer_list的结果。如果你使用**uniform initialization（花括号的方式进行初始化）**用得很爽你就得记住这个例外以免犯错，在C++11编程中一个典型的错误就是偶然使用了**std::initializer_list**类型的变量,这个陷阱也导致了很多C++程序员抛弃花括号初始化，只有不得不使用的时候再做考虑。

对于C++11故事已经说完了。但是对于C++14故事还在继续，C++14允许auto用于函数返回值并会被推导（参见Item3），而且C++14的lambda函数也允许在形参中使用auto。但是在这些情况下虽然表面上使用的是auto但是实际上是模板类型推导的那一套规则在工作，所以说下面这样的代码不会通过编译：

```
auto createInitList()
{
    return {1,2,3};     //错误！推导失败
}
```

同样在C++14的lambda函数中这样使用auto也不能通过编译：

```
std::vector<int> v;

auto resetV = [&v](const auto & newValue){v=newValue;}; //C++14
...
reset({1,2,3});          //错误！推导失败
```

记住：

- auto类型推导通常和模板类型推导相同，但是auto类型推导假定花括号初始化代表**std::initializer_list**而模板类型推导不这样做
- 在C++14中auto允许出现在函数返回值或者lambda函数形参中，但是它的工作机制是模板类型推导那一套方案。

Item 3: Understand decltype

条款三:理解decltype

decltype是一个奇怪的东西。给它一个名字或者表达式**decltype**就会告诉你名字或者表达式的类型。通常，它会精确的告诉你你想要的结果。但有时候它得出的结果也会让你挠头半天最后只能网上问答求助寻求解释。

我们将从一个简单的情况开始，没有任何令人惊讶的情况。相比模板类型推导和auto类型推导，decltype只是简单的返回名字或者表达式的类型：

```
const int i=0;                          //decltype(i)是const int

bool f(const Widget& w);      //decltype(w)是const Widget&
                                                //decltype(f)是bool(const
Widget&)

struct Point{
    int x;                                      //decltype(Point::x)是int
    int y;                                      //decltype(Point::y)是int
};

template<typename T>
class Vector{
    ...
    T& operator[](std::size_t index);
    ...
}
vector<int> v;                      //decltype(v)是vector<int>
...
if(v[0]==0)                             //decltype(v[0])是int&
```

看见了吧？没有任何奇怪的东西。

在C++11中，decltype最主要的用途就是用于函数模板返回类型，而这个返回类型依赖形参。举个例子，假定我们写一个函数，一个参数为容器，一个参数为索引值，这个函数支持使用方括号的方式访问容器中指定索引值的数据，然后在返回索引操作的结果前执行认证用户操作。函数的返回类型应该和索引操作返回的类型相同。

对一个T类型的容器使用**operator[]** 通常会返回一个T&对象，比如**std::deque**就是这样，但是**std::vector**有一个例外，对于**std::vector**，**operator[]**不会返回**bool&**，它会返回一个有名字的对象类型（译注：MSVC的STL实现中返回的是**std::_Vb_reference<std::Wrap_alloc<std::allocator>>**）。关于这个问题的详细讨论请参见Item6，这里重要的是我们可以看到对一个容器进行**operator[]**操作返回的类型取决于容器本身。

使用**decltype**使得我们很容易去实现它，这是我们写的第一个版本，使用**decltype**计算返回类型，这个模板需要改良，我们把这个推迟到后面：

```
template<typename Container,typename Index>
auto authAndAccess(Container& c,Index i)
->decltype(c[i])
{
    authenticateUser();
    return c[i];
}
```

函数名称前面的**auto**不会做任何的类型推导工作。相反的，他只是暗示使用了C++11的尾置返回类型语法，即在函数形参列表后面使用一个**->** 符号指出函数的返回类型，尾置返回类型的好处是我们可以在函数返回类型中使用函数参数相关的信息。在**authAndAccess**函数中，我们指定返回类型使用c和i。如果我们按照传统语法把函数返回类型放在函数名称之前， c和i就未被声明所以不能使用。

在这种声明中，**authAndAccess**函数返回**operator[]** 应用到容器中返回的对象的类型，这也正是我们期望的结果。

C++11允许自动推导单一语句的lambda表达式的返回类型， C++14扩展到允许自动推导所有的lambda表达式和函数，甚至它们内含多条语句。对于**authAndAccess**来说这意味着在C++14标准下我们可以忽略尾置返回类型，只留下一个**auto**。在这种形式下**auto**不再进行auto类型推导，取而代之的是它意味着编译器将会从函数实现中推导出函数的返回类型。

```
template<typename Container,typename Index> //C++ 14版本
auto authAndAccess(Container& c,Index i)
{
    authenticateUser();
    return c[i];
}
```

Item2解释了函数返回类型中使用**auto**编译器实际上是使用的模板类型推导的那套规则。如果那样的话就会这里就会有一些问题，正如我们之前讨论的，**operator[]** 对于大多数T类型的容器会返回一个**T&**，但是Item1解释了在模板类型推导期间，如果表达式是一个引用那么引用会被忽略。基于这样的规则，考虑它会对下面用户的代码有哪些影响：

```
std::deque<int> d;
...
authAndAccess(d,5)=10;          //认证用户，返回d[5],
                                                    //然后把10赋值给它
                                                     //无法通过编译器!
```

在这里**d[5]**本该返回一个**int&**，但是模板类型推导会剥去引用的部分，因此产生了**int**返回类型。函数返回的值是一个右值，上面的代码尝试把10赋值给右值，C++11禁止这样做，所以代码无法编译。

要想让**authAndAccess**像我们期待的那样工作，我们需要使用**decltype**类型推导来推导它的返回值，比如指定**authAndAccess**应该返回一个和**c[i]** 表达式类型一样的类型。C++期望在某些情况下当类型被暗示时需要使用**decltype**类型推导的规则，C++14通过使用**decltype(auto)** 说明符使得这成为可能。我们第一次看见**decltype(auto)** 可能觉得非常的矛盾，（到底是decltype还是auto？），实际上我们可以这样解释它的意义：**auto**说明符表示这个类型将会被推导，**decltype**说明**decltype**的规则将会引用到这个推导过程中。因此我们可以这样写**authAndAccess**：

```
template<typename Container,typename Index>
decltype(auto)
authAndAccess(Container& c,Index i)
{
    authenticateUser();
    return c[i];
}
```

现在authAndAccess将会真正的返回c[i]的类型。现在事情解决了，一般情况下c[i]返回**T&**，authAndAccess也会返回
T&,特殊情况下c[i]返回一个对象，authAndAccess也会返回一个对象。

decltype(auto) 的使用不仅仅局限于函数返回类型，当你想对初始化表达式使用decltype推导的规则，你也可以使用：

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;             //auto类型推导
                                 //myWidget1的类型为Widget
decltype(auto) myWidget2 = cw;   //decltype类型推导
                                 //myWidget2的类型是const Widget&
```

但是这里有两个问题困惑着你。一个是我之前提到的authAndAccess的改良至今都没有描述。让我们现在加上它。

再看看C++14版本的authAndAccess：

```
template<typename Container,typename Index>
decltype(auto) authAndAccess(Container& c,Index i);
```

容器通过传引用的方式传递非常量左值引用，因为返回一个引用允许用户可以修改容器。但是这意味着在这个函数里面不能传值调用，右值不能被绑定到左值引用上（除非这个左值引用是一个const，但是这里明显不是）。

公认的向**authAndAccess**传递一个右值是一个edge case。一个右值容器，是一个临时对象，通常会在authAndAccess调用结束被销毁，这意味着authAndAccess返回的引用将会成为一个悬置的(dangle)引用。但是使用向authAndAccess传递一个临时变量也并不是没有意义，有时候用户可能只是想简单的获得临时容器中的一个元素的拷贝，比如这样：

```
std::deque<std::string> makeStringDeque();       //工厂函数

//从makeStringDeque中或得第五个元素的拷贝并返回
auto s = authAndAccess(makeStringDeque(),5);
```

要想支持这样使用authAndAccess我们就得修改一下当前的声明使得它支持左值和右值。重载是一个不错的选择（一个函数重载声明为左值引用，另一个声明为右值引用），但是我们就不得不维护两个重载函数。另一个方法是使authAndAccess的引用可以绑定左值和右值，Item24解释了那正是通用引用能做的，所以我们这里可以使用通用引用进行声明：

```
template<typename Containter,typename Index>
decltype(auto) authAndAccess(Container&& c,Index i);
```

在这个模板中，我们不知道我们操纵的容器的类型是什么，那意味着我们相当于忽略了索引对象的可能，对一个未知类型的对象使用传值是通常对程序的性能有极大的影响在这个例子中还会造成不必要的拷贝，还会造成对象切片行为，以及给同事落下笑柄。但是就容器索引来说，我们遵照标准模板库对于对于索引的处理是有理由的，所以我们坚持传值调用。

然而，我们还需要更新一下模板的实现让它能听从Item25的告诫应用**std::forward**实现通用引用：

```
template<typename Container,typename Index>     //最终的C++14版本
decltype(auto)
authAndAccess(Container&& c,Index i){
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

这样就能对我们的期望交上一份满意的答卷，但是这要求编译器支持C++14。如果你没有这样的编译器，你还需要使用C++11版本的模板，它看起来和C++14版本的极为相似，除了你不得不指定函数返回类型之外：

```
template<typename Container,typename Index>     //最终的C++11版本
auto
authAndAccess(Container&& c,Index i)
->decltype(std::forward<Container>(c)[i])
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

另一个问题是就像我在条款的开始唠叨的那样，decltype通常会产生你期望的结果，但并不总是这样。在极少数情况下它产生的结果可能让你很惊讶。老实说如果你不是一个大型库的实现者你不太可能会遇到这些异常情况。

为了完全理解decltype的行为，你需要熟悉一些特殊情况。它们大多数都太过晦涩以至于几乎没有书进行有过权威的讨论，这本书也不例外，但是其中的一个会让我们更加理解decltype的使用。

对一个名字使用decltype将会产生这个名字被声明的类型。名字是左值表达式，但那不影响decltype的行为，decltype确保产生的类型总是左值引用。换句话说，如果一个左值表达式除了名字外还有类型，那么decltype将会产生**T&**LEIX .这几乎没有什么太大影响，因为大多数左值表达式的类型天生具备一个左值引用修饰符。举个例子，函数返回左值，几乎也返回了左值引用。

这个行为暗含的意义值得我们注意，在：

```
int x =0;
```

中，x是一个变量的名字，所以**decltype(x)** 是**int**。但是如果用一个小括号包覆这个名字，比如这样**(x)**，就会产生一个比名字更复杂的表达式。对于名字来说，**x**是一个左值，C++11定义了表达式**(x)** 也是一个左值。因此**decltype((x))** 是**int&** 。用小括号覆盖一个名字可以改变decltype对于名字产生的结果。

在C++11中这稍微有点奇怪，但是由于C++14允许了**decltype(auto)** 的使用，这意味着你在函数返回语句中细微的改变就可以影响类型的推导：

```
decltype(auto) f1()
{
    int x = 0;
    ...
    return x;           //decltype(x) 是int，所以f1返回int
}

decltype(auto) f2()
{
    int x =0l;
    return (x);         //decltype((x))是int&，所以f2返回int&
}
```

注意不仅f2的返回类型不同于f1，而且它还引用了一个局部变量！这样的代码将会把你送上未定义行为的特快列车，一辆你绝对不想上第二次的车。

当使用**decltype(auto)** 的时候一定要加倍的小心，在表达式中看起来无足轻重的细节将会影响到类型的推导。为了确认类型推导是否产出了你想要的结果，请参见Item4描述的那些技术。

同时你也不应该忽略decltype这块大蛋糕。没错，decltype可能会偶尔产生一些令人惊讶的结果，但那毕竟是少数情况。通常，decltype都会产生你想要的结果，尤其是当你对一个名字使用decltype时，因为在这种情况下，decltype只是做一件本分之事：它产出名字的声明类型。

记住

- **decltype**总是不加修改的产生变量或者表达式的类型。
- 对于T类型的左值表达式，**decltype**总是产出T的引用即**T&**。
- C++14支持**decltype(auto)**，就像auto一样，推导出类型，但是它使用自己的独特规则进行推导。

Item 4:Know how to view deduced types

条款四:学会查看类型推导结果

选择使用工具查看类型推导取决于软件开发过程中你想在哪个阶段显示类型推导信息，我们探究三种方案：在你编辑代码的时候获得类型推导的结果，在编译期间获得结果，在运行时获得结果

IDE编辑器

在IDE中的代码编辑器通常可以显示程序代码中变量，函数，参数的类型，你只需要简单的把鼠标移到它们的上面，举个例子，有这样的代码中：

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

一个IDE编辑器可以直接显示x推导的结果为**int**，y推导的结果为**const int***

为此，你的代码必须或多或少的处于可编译状态，因为IDE之所以能提供这些信息是因为一个C++编译器（或者至少是前端中的一个部分）运行于IDE中。如果这个编译器对你的代码不能做出有意义的分析或者推导，它就不会显示推导的结果。

对于像int这样简单的推导，IDE产生的信息通常令人很满意。正如我们将看到的，如果更复杂的类型出现时，IDE提供的信息就几乎没有什么用了。

编译器诊断

另一个获得推导结果的方法是使用编译器出错时提供的错误消息。这些错误消息无形的提到了造成我们编译错误的类型是什么。
举个例子，假如我们想看到之前那段代码中x和y的类型，我们可以首先声明一个类模板但不定义。就像这样：

```
template<typename T>     //只对TD进行声明
class TD;            //TD == "Type Displayer"
```

如果尝试实例化这个类模板就会引出一个错误消息，因为这里没有用来实例化的类模板定义。为了查看x和y的类型，只需要使用它们的类型去实例化TD：

```
TD<decltype(x)> xType;  //引出错误消息
TD<decltype(y)> yType;  //x和y的类型
```

我使用**variableNameType**的结构来命名变量，因为这样它们产生的错误消息可以有助于我们查找。对于上面的代码，我的编译器产生了这样的错误信息，我取一部分贴到下面：

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type and
    cannot be defined
```

另一个编译器也产生了一样的错误，只是格式稍微改变了一下：

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

除了格式不同外，几乎所有我测试过的编译器都产生了这样有用的错误消息。

运行时输出

使用**printf**的方法使类型信息只有在运行时才会显示出来（尽管我不是非常建议你使用printf），但是它提供了一种格式化输出的方法。现在唯一的问题是只需对于你关心的变量使用一种优雅的文本表示。“这有什么难的“，你这样想”这正是typeid和std::type_info::name的价值所在”。为了实现我们我们想要查看x和y的类型的需求，你可能会这样写：

```
std::cout<<typeid(x).name()<<"\n";  //显示x和y的类型
std::cout<<typeid(y).name()<<"\n";
```

这种方法对一个对象如x或y调用**typeid**产生一个**std::type_info**的对象，然后**std::type_info**里面的成员函数**name()**来产生一个C风格的字符串表示变量的名字。

调用**std::type_info::name**不保证返回任何有意义的东西，但是库的实现者尝试尽量使它们返回的结果有用。实现者们对于“有用”有不同的理解。举个例子，GNU和Clang环境下x会显示为**i**，y会显示为**PKi**，这样的输出你必须要问问编译器实现者们才能知道他们的意义：i表示int，PK表示**const to konst**（**const**）。Microsoft的编译器输出得更直白一些：对于x输出“int“对于y输出”int const*“

因为对于x和y来说这样的结果是正确的，你可能认为问题已经接近了，别急，考虑一个更复杂的例子：

```
template<typename T>
void f(const T& param);

std::vector<Widget> createVec();

const auto vw = createVec();

if(!vw.empty()){
    f(&vw[0]);
    ...
}
```

在这段代码中包含了一个用户定义的类型Widget，一个STL容器和一个auto变量vw，这个更现实的情况是你可能在会遇到的并且想获得他们类型推导的结果，比如模板类型参数T，比如函数参数param。

从这里中我们不难看出typeid的问题所在。我们添加一些代码来显示类型：

```
template<typename T>
void f(const T& param){
    using std::cout;
    cout<<"T=       "<<typeid(T).name()<<"\n";
    cout<<"param =  "<<typeid(param).name()<<"\n";
    ...
}
```

GNU和Clang执行这段代码将会输出这样的结果

```
T=      PK6Widget
param=  PK6Widget
```

我们早就知道在这些编译器中PK表示“指向常量”，所以只有数字6对我们来说是神奇的。其实数字6是类名称的字符串长度，所以这些编译器高数我们T和param都是**const Widget***。

Microsoft的编译器也同意上述言论：

```
T=      class Widget const *
param=  class Widget const *
```

这三个独立的编译器产生了相同的信息而且非常准确，当然看起来不是那么准确。在模板f中，param的类型是**const T&**。难道你们不觉得T和param相同类型很奇怪吗？比如T是int，param的类型应该是**const int&**而不是相同类型才对吧。

遗憾的是，事实就是这样，**std::type_info::name**的结果并不总是可信的，就像上面一样三个编译器都犯了相同的错误。因为**std::type_info::name**被批准犯这样的错。正如Item1提到的如果传递的是一个引用，那么引用部分将被忽略，如果忽略后还具有常量性或者易变性，那么常量性或者易变性也会被忽略。那就是为什么**const Widget *const &**类型会输出**const Widget ***，首先引用被忽略，然后这个指针自身的常量性被忽略，剩下的就是指针指向一个常量对象。

同样遗憾的是，IDE编辑器显示的类型信息也不总是可靠的，或者说不总是有用的。还是一样的例子，一个IDE编辑器可能会把T的类型显示为

```
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget>>::_Alloc>::value_type>::value_type *
```

同样把param的类型显示为

```
const std::_Simple_types<...>::value_type *const&
```

这个比起T来说要简单一些，但是如果你不知道<...>表示编译器忽略T的类型那么可能你还是会产生困惑。如果你运气好点你的IDE可能表现得比这个要好一些。

比起运气如果你更倾向于依赖库，那么你乐意被告知**std::type_info::name**不怎么好，Boost TypeIndex Library（通常写作Boost.TypeIndex）是更好的选择。这个库不是标准C++的一部分，也不时IDE或者TD这样的模板。Boost TypeIndex是跨平台，开源，有良好的开源协议的库，这意味着使用Boost和STL一样具有高度可移植性。

这里是如何使用Boost.TypeIndex得到f的类型的代码

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param){
    using std::cout;
    using boost::type_index::type_id_with_cvr;

    //显示T
    cout<<"T=    "
            <<type_id_with_cvr<T>().pretty_name()
            <<"\n";
    //显示param类型
    cout<<"param=    "
            <<type_id_with_cvr<decltype(param)>().pretty_name()
            <<"\n";
}
```

boost::type_index::type_id_with_cvr获取一个类型实参，它不消除实参的常量性，易变性和引用修饰符，然后**pretty_name**成员函数输出一个我们能看懂的友好内容。
基于这个f的实现版本，再次考虑那个产生错误类型信息的调用：

```
std::vetor<Widget> createVec();
const auto vw = createVec();
if(!vw.empty()){
    f(&vw[0]);
    ...
}
```

在GNU和Clang的编译器环境下，使用Boost.TypeIndex版本的f最后会产生下面的输出：

```
T=      Widget const *
param=  Widget const * const&
```

在Microsoft的编译器环境下，结果也是极其相似：

```
T=      class Widget const *
param=  class Widget const * const&
```

这样近乎一致的结果是很不错的，但是请记住IDE，编译器错误诊断或者Boost.TypeIndex只是用来帮助你理解编译器推导的类型是什么。它们是有用的，但是作为本章结束语我想说它们根本不能让你不用理解Item1-3提到的。

记住

- 类型推断可以从IDE看出，从编译器报错看出，从一些库的使用看出
- 这些工具可能既不准确也无帮助，所以理解C++类型推导规则才是最重要的

CHAPTER 2 auto

从概念上来说，auto要多简单有多简单，但是它比看起来要微妙一些。使用它可以存储类型，当然，它也会犯一些错误，而且比之手动声明一些复杂类型也会存在一些性能问题。此外，从程序员的角度来说，如果按照符合规定的流程走，那auto类型推导的一些结果是错误的。当这些情况发生时，对我们来说引导auto产生正确的结果是很重要的，因为严格按照说明书上面的类型写声明虽然可行但是最好避免。

本章简单的覆盖了auto的里里外外。

Item 5:Prefer auto to explicit type declarations

条款五:优先考虑auto而非显式类型声明

哈，开心一下：

```
int x;
```

等等，该死！我忘记了初始化x，所以x的值是不确定的。它可能会被初始化为0，这得取决于工作环境。哎。

别介意，让我们转换一个话题， 对一个局部变量使用解引用迭代器的方式初始化：

```
template<typename It>
void dwim(It b, It e)
{
    while(b!=e){
        typename std::iterator_traits<It>::value_type
        currValue = *b;
    }
}
```

嘿！`typename std::iterator_traits<It>::value_type`是想表达迭代器指向的元素的值的类型吗？我无论如何都说不出它是多么有趣这样的话，该死！等等，我早就说过了吗？

好吧，声明一个局部变量，变量的类型只有编译后知道，这里必须使用'typename'指定，该死!

该死该死该死，C++编程不应该是这样不愉快的体验。

别担心，它只在过去是这样，到了C++11所有的这些问题都消失了，这都多亏了auto。auto变量从初始化表达式中推导出类型，所以我们必须初始化。这意味着当你在现代化C++的高速公路上飞奔的同时你不得不对只声明不初始化变量的老旧方法说拜拜：

```
int x1;             //潜在的未初始化的变量

auto x2;            //错误！必须要初始化

auto x3=0;          //没问题，x已经定义了
```

而且即使初始化表达式使用解引用迭代器也不会对你的高速驾驶有任何影响

```
template<typename It>
void dwim(It b,It e)
{
    while(b!=e){
        auto currValue = *b;
        ...
    }
}
```

因为auto使用Item2所述的auto类型推导技术，它甚至能表示一些只有编译器才知道的类型：

```
auto derefUPLess = [](const std::unique_ptr<Widget> &p1,    //专用于Widget类型的比较函数
const std::unique_ptr<Widget> &p2){return *p1<*p2;};
```

很酷对吧，如果使用C++14，将会变得更酷，因为lambda表达式中的形参也可以使用auto：

```
auto derefUPLess = [](const auto& p1,const auto& p2){return *p1<*p2;};
```

尽管这很酷，但是你可能会想我们完全不需要使用auto声明局部变量来保存一个闭包，因为我们可以使用`std::function`对象。
没错，我们的确可以那么做，但是事情可能不是完全如你想的那样。当然现在你可能会问：
`std::function`对象到底是什么，让我来给你解释一下：

`std::function`是一个C++11标准模板库中的一个模板，它泛化了函数指针的概念。与函数指针只能指向函数不同，`std::function`可以指向任何可调用对象，也就是那些像函数一样能进行调用的东西。当你声明函数指针时你必须指定函数类型（即函数签名），同样当你创建`std::function`对象时你也需要提供函数签名，由于它是一个模板所以你需要在它的模板参数里面提供。举个例子，假设你想声明一个`std::function`对象func使他指向一个可调用对象，比如一个具有这样函数签名的函数：

```
bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2);
```

你就得这么写：

```
std::function<bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2)> func;
```

因为lambda表达式能产生一个可调用对象，所以我们现在可以把闭包存放到`std::function`对象中。这意味着我们可以不使用auto写出C++11版的**dereUPLess**：

```
std::function<bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2)>
dereUPLess = [](const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2){return *p1<*p2;};
```

语法冗长不说，还需要重复写很多形参类型，使用`std::function`还不如使用auto。用auto声明的变量保存一个闭包这个变量将会得到和闭包一样的类型。

实例化`std::function`并声明一个对象这个对象将会有固定的大小。当使用这个对象保存一个闭包时它可能大小不足不能存储，这个时候`std::function`的构造函数将会在堆上面分配内存来存储，这就造成了使用`std::function`比auto会消耗更多的内存。并且通过具体实现我们得知通过std::function调用一个闭包几乎无疑比auto声明的对象调用要慢。
换句话说，std::function方法比auto方法要更耗空间且更慢，并且比起写一大堆类型使用auto要方便得多。在这场存储闭包的比赛中，auto无疑取得了胜利（也可以使用std::bind来生成一个闭包，但在Item34我会尽我最大努力说服你使用lambda表达式代替std::bind)

使用auto除了使用未初始化的无效变量，省略冗长的声明类型，直接保存闭包外，它还有一个好处是可以避免一个问题，我称之为依赖类型快捷方式的问题。你将看到这样的代码——甚至你会这么写：

```
std::vector<int> v;
unsigned sz = v.size();
```

`v.size()`的标准返回类型是`std::vector<int>::size_type`,但是很多程序猿都知道`std::vector<int>::size_type`实际上被指定为无符号整型，所以很多人都认为用`unsigned`比写那一长串的标准返回类型方便。这会造成一些有趣的结果。
举个例子，在**Windows 32-bit**上`std::vector<int>::size_type`和`unsigned int`都是一样的类型，但是在**Windows 64-bit**上`std::vector<int>::size_type`是64位，`unsigned int`是32位。这意味着这段代码在Windows 32-bit上正常工作，但是当把应用程序移植到Windows 64-bit上时就可能会出现一些问题。
谁愿意花时间处理这些细枝末节的问题呢?

所以使用auto可以确保你的不需要浪费时间：

```
auto sz =v.size();
```

你还是不相信使用auto是多么明智的选择？考虑下面的代码：

```
std::unordered_map<std::string,int> m;
...
for(const std::pair<std::string,int>& p : m)
{
    ...
}
```

看起来好像很合情合理的表达，但是这里有一个问题，你看到了吗?

要想看到错误你就得知道`std::unordered_map`的key是一个常量，所以`std::pair`的类型不是`std::pair<std::string,int>`而是`std::pair<const std::string,int>`。编译器会努力的找到一种方法把前者转换为后者。它会成功的，因为它会创建一个临时对象，这个临时对象的类型是p想绑定到的对象的类型，即m中元素的类型，然后把p的引用绑定到这个临时对象上。在每个循环迭代结束时，临时对象将会销毁，如果你写了这样的一个循环，你可能会对它的一些行为感到非常惊讶，因为你确信你只是让成为p指向m中各个元素的引用而已。

使用auto可以避免这些很难被意识到的类型不匹配的错误：

```
for(const auto & p : m)
{
    ...
}
```

这样无疑更具效率，且更容易书写。而且，这个代码有一个非常吸引人的特性，如果你把p换成是指向m中各个元素的指针，在没有auto的版本中p会指向一个临时变量，这个临时变量在每次迭代完成时会被销毁!

后面这两个例子说明了显式的指定类型可能会导致你不像看到的类型转换。如果你使用auto声明目标变量你就不必担心这个问题。

基于这些原因我建议你优先考虑auto而非显式类型声明。然而auto也不是完美的。每个auto变量都从初始化表达式中推导类型，有一些表达式的类型和我们期望的大相径庭。关于在哪些情况下会发生这些问题，以及你可以怎么解决这些问题我们在Item2和6讨论，所以这里我不再赘述。我想把注意力放到你可能关心的另一点：使用auto代替传统类型声明对源码可读性的影响。

首先，深呼吸，放松，auto是**可选项**，不是**命令**，在某些情况下如果你的专业判断告诉你使用显式类型声明比auto要更清晰更易维护，那你就不必再坚持使用auto。牢记C++没有在其他众所周知的语言所拥有的类型接口上开辟新土地。

其他静态类型的过程式语言（如C#,D,Sacla,Visual Basic等)或多或少的都有那些非静态类型的函数式语言（如ML,Haskell,OCaml.F#等）的特性。在某种程度上，几乎没有显式类型使得动态类型语言Perl,Python,Ruby等取得了成功，软件开发社区对于类型接口有丰富的经验，他们展示了在维护大型工业强度的代码上使用这种技术没有任何争议。

一些开发者也担心使用auto就不能瞥一眼源代码便知道对象的类型，然而，IDE扛起了部分担子，在很多情况下，少量显示一个对象的类型对于知道对象的确切类型是有帮助的，这通常已经足够了。举个例子，要想知道一个对象是容器还是计数器还是智能指针，不需要知道它的确切类型，一个适当的变量名称就能告诉我们大量的抽象类型信息。

真正的问题是显式指定类型可以避免一些微妙的错误，以及更具效率和正确性，而且，如果初始化表达式改变变量的类型也会改变，这意味着使用auto可以帮助我们完成一些重构工作。举个例子，如果一个函数返回类型被声明为int，但是后来你认为将它声明为long会更好，调用它作为初始化表达式的变量会自动改变类型，但是如果你不使用auto你就不得不在源代码中挨个找到调用地点然后修改它们。

记住

- auto变量必须初始化，通常它可以避免一些移植性和效率性的问题，也使得重构更方便，还能让你少打几个字。
- 正如Item2和6讨论的，auto类型的变量可能会踩到一些陷阱。

Item 6:Use the explicitly typed initializer idiom when auto deduces undesired types.

条款六:auto推导若非己愿，使用显式类型初始化惯用法

在Item5中解释了比起显式指定类型使用auto声明变量有若干技术优势，但是有时当你想向左转auto却向右转。举个例子，假如我有一个函数，参数为Widget，返回一个`std::vector<bool>`，这里的bool表示Widget是否提供一个独有的特性。

```
std::vector<bool> features(const Widget& w);
```

更进一步假设5表示是否Widget具有高优先级，我们可以写这样的代码:

```
bool highPriority = features(w)[5];
...
processWidget(w,highPriority);
```

这个代码没有任何问题。它会正常工作，但是如果我们使用auto代替显式指定类型做一些看起来很无害的改变:

```
auto highPriority = features(w)[5];
...
processWidget(w,highPriority);      //未定义行为!
```

就像注释说的，这个processWidget是一个未定义行为。为什么呢？答案有可能让你很惊讶，使用auto后highPriority不再是bool类型。虽然从概念上来说`std::vector<bool>`意味着存放bool，但是`std::vector<bool>`的`operator[]`不会返回容器中元素的引用，取而代之它返回一个`std::vector<bool>::reference`的对象（一个嵌套于`std::vector<bool>`中的类）。`std::vector<bool>::reference`之所以存在是因为`std::vector<bool>`指定了它作为代理类。`operator[]`返回一个代理类来扮演bool&。要想成功扮演这个角色，bool&适用的上下文`std::vector<bool>::reference`也必须一样能适用。基于这个特性`std::vector<bool>::reference`可以隐式的转化为bool（不是bool&，是bool！要想完整的解释`std::vector<bool>::reference`能模拟bool&的行为所使用的一堆技术可能扯得太远了，所以这里简单地说隐式类型转换只是这个大型马赛克的一小块）

有了这些信息，我们再来看看原始代码的一部分:

```
bool highPriority = features(w)[5];     //显式的声明highPriority的类型
```

这里，feature返回一个`std::vector<bool>`对象后再调用`operator[]`，`operator[]`将会返回一个`std::vector<bool>::reference`对象，然后再通过隐式转换赋值给bool变量highPriority。highPriority因此表示的是features返回的vector中的第五个bit，这也正如我们所期待的那样。
然后再对照一下当使用auto时发生了什么:

```
auto highPriority = features(w)[5];     //推导highPriority的类型
```

同样的，feature返回一个`std::vector<bool>`对象，再调用`operator[]`，`operator[]`将会返回一个`std::vector<bool>::reference`对象，但是现在这里有一点变化了，auto推导highPriority的类型为`std::vector<bool>::reference`，但是highPriority对象没有第五bit的值。

这个值取决于`std::vector<bool>::reference`的具体实现。其中的一种实现是这样的（`std::vector<bool>::reference`）对象包含一个指向word的指针，然后加上方括号中的偏移实现被引用bit这样的行为。然后再来考虑highPriority初始化表达的意思，注意这里假设`std::vector<bool>::reference`就是刚提到的实现方式。

调用feature将返回一个std::vector，这个对象没有名字，为了方便我们的讨论，我这里叫他temp，`operator[]`被temp调用，然后然后的`std::vector<bool>::reference`包含一个指针，这个指针指向一个temp里面的word，加上相应的偏移,。highPriority是一个`std::vector<bool>::reference`的拷贝，所以highPriority也包含一个指针，指向temp中的一个word，加上合适的偏移，这里是5.在这个语句解释的时候temp将会被销毁，因为它是一个临时变量。因此highPriority包含一个悬置的指针，如果用于processWidget调用中将会造成未定义行为：

```
processWidget(w,highPriority);      //未定义行为!
                                    //highPriority包含一个悬置指针
```

`std::vector<bool>::reference`是一个代理类的例子：所谓代理类就是以模仿和增强一些类型的行为为目的而存在的类。很多情况下都会使用代理类，`std::vector<bool>::reference`展示了对`std::vector<bool>`使用`operator[]`来实现引用bit这样的行为。另外，C++标准模板库中的智能指针也是用代理类实现了对原始指针的资源管理行为。代理类的功能已被大家广泛接受。事实上，“Proxy”设计模式是软件设计这座万神庙中一直都存在的高级会员。

一些代理类被设计于用以对客户可见。比如`std::shared_ptr`和`std::unique_ptr`。其他的代理类则与之相反，比如`std::vector<bool>::reference`和`std::bitset::reference`。

在后者的阵营里一些C++库也是用了表达式模板的黑科技。这些库通常被用于提高数值运算的效率。给出一个矩阵类Matrix和矩阵对象m1，m2，m3，m4，举个例子，这个表达式

```
Matrix sum = m1 + m2 + m3 + m4;
```

可以使计算更加高效，只需要使让`operator+`返回一个代理类代理结果而不是返回结果本身。也就是说，对两个Matrix对象使用`operator+`将会返回如`Sum<Matrix,Matrix>`这样的代理类作为结果而不是直接返回一个Matrix对象。在`std::vector<bool>::reference`和bool中存在一个隐式转换，同样对于Matrix来说也可以存在一个隐式转换允许Matrix的代理类转换为Matrix，这让表达式等号右边能产生代理对象来初始化Sum。客户应该避免看到实际的类型。

作为一个通则，不可见的代理类通常不适用于auto。这样类型的对象的生命期通常不会设计为能活过一条语句，所以创建那样的对象你基本上就走向了违反程序库设计基本假设的道路.`std::vector<bool>::reference`就是这种情况，我们看到违反这个基本假设将导致未定义行为。

因此你想避开这种形式的代码：

```
auto someVar = expression of "invisible" proxy class type;
```

但是你怎么能意识到你正在使用代理类？它们被设计为不可见，至少概念上说是这样！每当你发现它们，你真的应该舍弃Item5演示的auto所具有的诸多好处吗？

让我们首先回到如何找到它们的问题上。虽然代理类都在程序员日常使用的雷达下方飞行，但是很多库都证明它们可以上方飞行。当你越熟悉你使用的库的基本设计理念，你的思维就会越活跃，不至于思维僵化认为代理类只能在这些库中使用。

当缺少文档的时候，可以去看看头文件。很少会出现源代码全都用代理对象，它们通常用于一些函数的返回类型，所以通常能从函数签名中看出它们的存在。这里有一份来自C++ STANDARD的说明书：

```
namespace std{
    template<class Allocator>
    class vector<bool,Allocator>{
        public:
        class reference{...};

        reference operator[](size_type n);
    };
}
```

假设你知道对std::vector使用`operator[]`通常会返回一个**T&**,在这里operator[]不寻常的返回类型提示你它使用了代理类。多关注你使用的接口可以暴露代理类的存在。

实际上，很多开发者都是在跟踪一些令人困惑的复杂问题或在单元测试出错进行调试时才看到代理类的使用。不管你怎么发现它们的，当你不知道这个类型有没有被代理还想使用auto时你就不能单单只用一个auto。auto本身没什么问题，问题是auto不会推导出你想要的类型。解决方案是强制使用一个不同的类型推导形式，这种方法我通常称之为显式类型初始器惯用法（*the explicitly typed initialized idiom*）

显式类型初始器惯用法使用auto声明一个变量，然后对表达式强制类型转换得出你期望的推导结果。举个例子，我们该怎么将这个惯用法施加到highPriority上？

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

这里，feature(w)[5]还是返回一个`std::vector<bool>::reference`对象，就像之前那样，但是这个转型使得表达式类型为bool，然后auto才被用于推导highPriority。在运行时，对std::vector使用operator[]将返回一个std::vector::reference，然后强制类型转换使得它执行向bool的转型，在这个过程中指向`std::vector<bool>`的指针已经被解引用。这就避开了我们之前的未定义行为。然后5将被用于指向bit的指针，bool值被用于初始化highPriority。

对于Matrix来说，显式类型初始器惯用法是这样的：

```
auto sum = static_cast<Matrix>(m1+m2+m3+m4);
```

应用这个惯用法不限制初始化表达式产生一个代理类。它也可以用于强调你声明了一个变量类型，它的类型不同于初始化表达式的类型。举个例子，假设你有这样一个表达式计算公差值：

```
double calEpsilon();
```

calEpsilon清楚的表明它返回一个double，但是假设你知道对于这个程序来说使用float的精度已经足够了，而且你很关心double和float的大小。你可以声明一个float变量储存**calEpsilon**的计算结果。

```
float ep = calEpsilon();
```

但是这几乎没有表明“我确实要减少函数返回值的精度”。使用显式类型初始器惯用法我们可以这样：

```
auto ep = static_cast<float>(calEpsilon());
```

处于同样的原因，如果你故意想用int类型存储一个表达式返回的float类型的结果，你也可以使用这个方法。假如你需要计算一个随机访问迭代器（比如std::vector,std::deque,std::array）中某元素的下标，你给它一个0.0到1.0的值表明这个元素离容器的头部有多远（0.5意味着位于容器中间）。进一步假设你很自信结果下标是int。如果容器是c，d是double类型变量，你可以用这样的方法计算容器下标：

```
int index = d * c.size();
```

但是这种写法并没有明确表明你想将右侧的double类型转换成int类型，显式类型初始器可以帮助你正确表意：

```
auto index = static_cast<int>(d * size());
```

记住

- 不可见的代理类可能会使auto从表达式中推导出“错误的”类型
- 显式类型初始器惯用法强制auto推导出你想要的结果

CHAPTER 3 Moving to Modern C++

说起知名的特性，C++11/14有一大堆可以吹的东西，auto，智能指针，移动语意，lambda，并发——每个都是如此的重要，这章将覆盖这些内容。
精通这些特性是必要的，但是成为高效率的现代C++程序员也要求一系列小步骤。
从C++98移步C++11/14遇到的每个细节问题都会在本章得到答复。
应该在创建对象时用{}而不是()吗？为什么alias声明比typedef好？constexpr和const有什么不同？常量成员函数和线程安全有什么关系？这个列表越列越多。
这章将会逐个回答这些问题。

Item 7:Distinguish between () and {} when creating objects

条款七:区别使用()和{}创建对象

从不同的角度看，C++11初始化对象的语法选择既丰富得让人尴尬又混乱得让人糊涂。一般来说，初始化值要用()或者{}括起来或者放到"="的右边：

```
int x(0);               //使用小括号初始化

int y = 0;              //使用"="初始化

int z{0};               //使用花括号初始化
```

在很多情况下，你可以使用"="和花括号的组合：

```
int z = {0};            //使用"="和花括号
```

在这个条款的剩下部分，我通常会忽略"="和花括号组合初始化的语法，因为C++通常把它视作和只有花括号一样。
"混乱得令人糊涂"指出在初始化中使用"="可能会误导C++新手，使他们以为这里是赋值运算符。
对于像int这样的内置类型，研究两者区别是没有多大意义的，但是对于用户定义的类型而言，区别赋值运算符和初始化就非常重要了，因为这可能包含不同的函数调用：

```
Widget w1;              //调用默认构造函数

Widget w2 = w1;         //不是赋值运算符，调用拷贝构造函数

w1 = w2;                //是一个赋值运算符，调用operator=函数
```

甚至对于一些初始化语法，在一些情况下C++98没有办法去表达初始化。举个例子，要想直接表示一些存放一个特殊值的STL容器是不可能的（比如Item1,3,5）

C++11使用统一初始化(uniform initialization)来整合这些混乱且繁多的初始化语法，所谓统一初始化是指使用单一初始化语法在任何地方*[0]*表达任何东西。
它基于花括号，出于这个原因我更喜欢称之为括号初始化*[1]*。统一初始化是一个概念上的东西，而括号初始化是一个具体语法构型。
括号初始化让你可以表达以前表达不出的东西。使用花括号，指定一个容器的元素变得很容易：

```
std::vector<int> v{1,3,5};      //v包含1,3,5
```

括号初始化也能被用于为非静态数据成员指定默认初始值。C++11允许"="初始化也拥有这种能力：

```
class Widget{
    ...
private:
    int x{0};        //没问题，x初始值为0
    int y = 0;       //同上
    int z(0);        //错误!
}
```

另一方面，不可拷贝的对象可以使用花括号初始化或者小括号初始化，但是不能使用"="初始化：

```
std::vector<int> ai1{0};    //没问题，x初始值为0
std::atomic<int> ai2(0);    //没问题
std::atomic<int> ai3 = 0;   //错误!
```

因此我们很容易理解为什么括号初始化又叫统一初始化，在C++中这三种方式都被指派为初始化表达式，但是只有括号任何地方都能被使用。

括号表达式有一个异常的特性，它不允许内置类型隐式的变窄转换（narrowing conversion）。如果一个使用了括号初始化的表达式的值无法用于初始化某个类型的对象，代码就不会通过编译：

```
double x,y,z;

int sum1{x+y+z};            //错误！三个double的和不能用来初始化int类型的变量
```

使用小括号和"="的初始化不检查是否转换为变窄转换，因为由于历史遗留问题它们必须要兼容老旧代码

```
int sum2(x + y +z);         //可以（表达式的值被截为int）

int sum3 = x + y + z;       //同上
```

另一个值得注意的特性是括号表达式对于C++最令人头疼的解析问题*[2]*有天生的免疫性。
C++规定任何能被决议为一个声明的东西必须被决议为声明。这个规则的副作用是让很多程序员备受折磨：当他们想创建一个使用默认构造函数构造的对象，却不小心变成了函数声明。
问题的根源是如果你想使用一个实参调用一个构造函数，你可以这样做：

```
Widget w1(10);              //使用实参10调用Widget的一个构造函数
```

但是如果你尝试使用一个没有参数的构造函数构造对象，它就会变成函数声明：

```
Widget w2();                //最令人头疼的解析！声明一个函数w2，返回Widget
```

由于函数声明中形参列表不能使用花括号，所以使用花括号初始化表明你想调用默认构造函数构造对象就没有问题：

```
Widget w3{};                    //调用没有参数的构造函数构造对象
```

关于括号初始化还有很多要说的。它的语法能用过各种不同的上下文，它防止了隐式的变窄转换，而且对于C++最令人头疼的解析也天生免疫。
既然好到这个程度那为什么这个条款不叫“Prefer braced initialization syntax”呢?

括号初始化的缺点是有时它有一些令人惊讶的行为。
这些行为使得括号初始化和std::initializer_list和构造函数重载决议本来就不清不楚的暧昧关系进一步混乱。
把它们放到一起会让看起来应该左转的代码右转。
举个例子，Item2解释了当auto声明的变量使用花括号初始化，变量就会被推导为std::initializer_list，尽管使用相同内容的其他初始化方式会产生正常的结果。
所以，你越喜欢用atuo，你就越不能用括号初始化。

在构造函数调用中，只要不包含std::initializer_list参数，那么花括号初始化和小括号初始化都会产生一样的结果：

```
class Widget {
public:
    Widget(int i, bool b);       //未声明默认构造函数
    Widget(int i, double d);     // std::initializer_list参数
    …
};
Widget w1(10, true);           // 调用构造函数
Widget w2{10, true};           // 同上
Widget w3(10, 5.0);            // 调用第二个构造函数
Widget w4{10, 5.0};            // 同上
```

然而，如果有一个或者多个构造函数的参数是std::initializer_list，
使用括号初始化语法绝对比传递一个std::initializer_list实参要好。
而且只要某个调用能使用括号表达式编译器就会使用它。
如果上面的Widget的构造函数有一个**std::initializer_list**实参，就像这样：

```
class Widget {
public:
    Widget(int i, bool b);     // 同上
    Widget(int i, double d);   // 同上
    Widget(std::initializer_list<long double> il);       //新添加的
    …
};
```

w2和**w4**将会使用新添加的构造函数构造，即使另一个非std::initializer_list构造函数对于实参是更好的选择：

```
Widget w1(10, true);      // 使用小括号初始化
                          //调用第一个构造函数

Widget w2{10, true};      // 使用花括号初始化
                          // 调用第二个构造函数
                          // (10 和 true 转化为long double)

Widget w3(10, 5.0);       // 使用小括号初始化
                          // 调用第二个构造函数

Widget w4{10, 5.0};       // 使用花括号初始化
                          // 调用第二个构造函数
                          // (10 和 5.0 转化为long double)
```

甚至普通的构造函数和移动构造函数都会被std::initializer_list构造函数劫持：

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<long double> il);
    operator float() const;          // convert to float (译者注：高亮)

};
Widget w5(w4);                  // 使用小括号，调用拷贝构造函数

Widget w6{w4};                  // 使用花括号，调用std::initializer_list构造函数

Widget w7(std::move(w4));       // 使用小括号，调用移动构造函数

Widget w8{std::move(w4)};       // 使用花括号，调用std::initializer_list构造函数
```

编译器热衷于把括号初始化与使std::initializer_list构造函数匹配了，热衷程度甚至超过了最佳匹配。比如：

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<bool> il);    // element type is now bool

  …                                    // no implicit conversion funcs
};
Widget w{10, 5.0};      //错误！要求变窄转换
```

这里，编译器会直接忽略前面两个构造函数，然后尝试调用第三个构造函数，也即是std::initializer_list构造函数。
调用这个函数将会把`int(10)`和double(5.0)`转换为bool，由于括号初始化拒绝变窄转换，所以这个调用无效，代码无法通过编译。

只有当没办法把括号初始化中实参的类型转化为std::initializer_list时，编译器才会回到正常的函数决议流程中。
比如我们在构造函数中用`std::initializer_list<std::string`代替`std::initializer_list<bool>`，这时非std::initializer_list构造函数将再次成为函数决议的候选者，因为没有办法把int和bool转换为std::string:

```
class Widget {
public:
    Widget(int i, bool b);
    Widget(int i, double d);
    Widget(std::initializer_list<std::string> il);
     …
};
Widget w1(10, true);      // 使用小括号初始化，调用第一个构造函数
Widget w2{10, true};      // 使用花括号初始化，调用第一个构造函数
Widget w3(10, 5.0);       // 使用小括号初始化，调用第二个构造函数
Widget w4{10, 5.0};       // 使用花括号初始化，调用第二个构造函数
```

代码的行为和我们刚刚的论述如出一辙。这里还有一个有趣的边缘情况*[3]*。
假如你使用的花括号初始化是空集，并且你欲构建的对象有默认构造函数，也有std::initializer_list构造函数。
你的空的花括号意味着什么？如果它们意味着没有实参，就该使用默认构造函数，
但如果它意味着一个空的std::initializer_list，就该调用std::initializer_list构造函数。

最终会调用默认构造函数。空的花括号意味着没有实参，不是一个空的std::initializer_list：

```
class Widget {
public:
    Widget();
    Widget(std::initializer_list<int> il);

    ...
};
Widget w1;              // 调用默认构造函数
Widget w2{};            // 同上
Widget w3();            // 最令人头疼的解析！声明一个函数
```

如果你想调用std::initializer_list构造，你就得创建一个空花括号的实参来表明你想调用一个std::initializer_list构造函数，它的实参是一个空值。

```
Widget w4({});          // 调用std::initializer_list
Widget w5{{}};          // 同上
```

此时，括号初始化的晦涩规则，std::initializer_list和构造函数重载就会一下子涌进你的脑袋，你可能会想研究了半天这些东西在你的日常编程中到底占多大比例。
可能比你想象的要多。因为std::vector也会受到影响。
std::vector有一个非std::initializer_list构造函数允许你去指定容器的初始大小，以及使用一个值填满你的容器。
但它也有一个std::initializer_list构造函数允许你使用花括号里面的值初始化容器。如果你创建一个数值类型的vector，然后你传递两个实参。把这两个实参放到小括号和放到花括号中是不同：

```
std::vector<int> v1(10, 20);    //使用非std::initializer_list
                                //构造函数创建一个包含10个元素的std::vector
                                //所有的元素的值都是20
std::vector<int> v2{10, 20};    //使用std::initializer_list
                                //构造函数创建包含两个元素的std::vector
                                //元素的值为10和20
```

让我们退回之前的讨论。从这个讨论中我有两个重要结论。
第一，作为一个类库作者，你需要意识到如果你的一堆构造函数中重载过一个或者多个std::initializer_list，
用户代码如果使用了括号初始化，可能只会看到你重载的std::initializer_list这一个版本的构造函数。
因此，你最好把你的构造函数设计为不管用户是小括号还是使用花括号进行初始化都不会有什么影响。
换句话说，现在看到std::vector设计的缺点以后你设计的时候避免它。

这里的暗语是如果一个类没有std::initializer_list构造函数，然后你添加一个，
用户代码中如果使用括号初始化可能会发现过去被决议为非std::initializer_list构造函数现在被决议为新的函数。
当然，这种事情也可能发生在你添加一堆重载函数的时候，std::initializer_list重载不会和其他重载函数比较，
它直接盖过了其它重载函数，其它重载函数几乎不会被考虑。所以如果你要使用std::initializer_list构造函数，请三思而后行。

第二个，作为一个类库使用者，你必须认真的在花括号和小括号之间选择一个来创建对象。
大多数开发者都使用其中一种作为默认情况，只有当他们不能使用这种的时候才会考虑另一种。
如果使用默认使用花括号初始化，会得到大范围适用面的好处，它禁止变窄转换，免疫C++最令人头疼的解析。
他们知道在一些情况下（比如给一个容器大小和一个值创建std::vector)要使用小括号。
如果默认使用小括号初始化，它们能和C++98语法保持一致，它避开了auto自动推导std::initializer_list的问题，
也不会不经意间就调用了std::initializer_list构造函数。
他们承认有时候只能使用花括号（比如创建一个包含特殊值的容器）。
关于花括号和小括号的使用没有一个一致的观点，所以我的建议是用一个，并坚持使用。

如果你是一个模板的作者，花括号和小括号创建对象就更麻烦了。
通常不能知晓哪个会被使用。
举个例子，假如你想创建一个接受任意数量的参数，然后用它们创建一个对象。使用可变参数模板(variadic template)可以非常简单的解决：

```
template<typename T,
         typename... Ts>
void doSomeWork(Ts&&... params) {
  create local T object from params...  …
}
```

在现实中我们有两种方式使用这个伪代码（关于**std::forward**请参见Item25）：

```
T localObject(std::forward<Ts>(params)...);     // 使用小括号
T localObject{std::forward<Ts>(params)...};     // 使用花括号
```

考虑这样的调用代码：

```
std::vector<int> v;
…
 doSomeWork<std::vector<int>>(10, 20);
```

如果doSomeWork创建localObject时使用的是小括号，std::vector就会包含10个元素。
如果doSomeWork创建localObject时使用的是花括号，std::vector就会包含2个元素。
哪个是正确的？**doSomeWork**的作者不知道，只有调用者知道。

这正是标准库函数std::make_unique和std::make_shared（参见Item21）面对的问题。
它们的解决方案是使用小括号，并被记录在文档中作为接口的一部分。

记住

- 括号初始化是最广泛使用的初始化语法，它防止变窄转换，并且对于C++最令人头疼的解析有天生的免疫性
- 在构造函数重载决议中，括号初始化尽最大可能与std::initializer_list参数匹配，即便其他构造函数看起来是更好的选择
- 对于数值类型的std::vector来说使用花括号初始化和小括号初始化会造成巨大的不同
- 在模板类选择使用小括号初始化或使用花括号初始化创建对象是一个挑战。

译注

[0] 结合上下文得知这里的"任何地方"指的是初始化表达式存在的地方而不是广义上源代码的各处。

[1] 注意，这里的括号初始化指的是花括号初始化，在没有歧义的情况下下文的括号初始化指的都是用花括号进行初始化；当与小括号初始化同时存在并可能产生歧义时我会直接指出

[2] 所谓最令人头疼的解析即*most vexing parse*，更多信息请参见https://en.wikipedia.org/wiki/Most_vexing_parse

[3] 参见https://en.wikipedia.org/wiki/Edge_case

Item 8: Prefer nullptr to 0 and NULL.

条款八:优先考虑nullptr而非0和NULL

你看这样对不对：字面值0是一个int不是指针。
如果C++发现在当前上下文只能使用指针，它会很不情愿的把0解释为指针，但是那是最后的退路。
一般来说C++的解析策略是把0看做int而不是指针。

实际上，NULL也是这样的。但在NULL的实现细节有些不确定因素，
因为实现被允许给NULL一个除了**int**之外的整型类型（比如**long**）。
这不常见，但也算不上问题所在。这里的问题不是NULL没有一个确定的类型，而是0和NULL都不是指针类型。

在C++98中，对指针类型和整型进行重载意味着可能导致奇怪的事情。
如果给下面的重载函数传递0或NULL，它们绝不会调用指针版本的重载函数：

```
void f(int);        //三个f的重载函数
void f(bool);
void f(void*);

f(0);               //调用f(int)而不是f(void*)

f(NULL);            //可能不会被编译，一般来说调用f(int),绝对不会调用f(void*)
```

而f(NULL)的不确定行为是由NULL的实现不同造成的。
如果NULL被定义为**0L**（指的是**0**为**long**类型），这个调用就具有二义性，因为从**long**到**int**的转换或从**long**到**bool**的转换或**0L**到**void*** 的转换都会被考虑。
有趣的是源代码表现出的意思（我指的是使用NULL调用f）和实际想表达的意思（我指的是用整型数据调用f）是相矛盾的。
这种违反直觉的行为导致C++98程序员都将避开同时重载指针和整型作为编程准则*[0]*。
在C++11中这个编程准则也有效，因为尽管我这个条款建议使用**nullptr**，可能很多程序员还是会继续使用**0**或**NULL**，哪怕**nullptr**是更好的选择。

nullptr的优点是它不是整型。
老实说它也不是一个指针类型，但是你可以把它认为是通用类型的指针。
nullptr的真正类型是**std::nullptr_t**，在一个完美的循环定义以后，**std::nullptr_t**又被定义为**nullptr**。
std::nullptr_t可以转换为指向任何内置类型的指针，这也是为什么我把它叫做通用类型的指针。

使用**nullptr**调用f将会调用**void***版本的重载函数，因为**nullptr**不能被视作任何整型：

```
f(nullptr);         //调用重载函数f的f(void*)版本
```

使用**nullptr***代替**0**和**NULL**可以避开了那些令人奇怪的函数重载决议，这不是它的唯一优势。
它也可以使代码表意明确，尤其是当和**auto**一起使用时。
举个例子，假如你在一个代码库中遇到了这样的代码：

```
auto result = findRecord( /* arguments */ );
if (result == 0) {
    …
}
```

如果你不知道findRecord返回了什么（或者不能轻易的找出），那么你就不太清楚到底result是一个指针类型还是一个整型。
毕竟，**0**也可以像我们之前讨论的那样被解析。
但是换一种假设如果你看到这样的代码：

```
auto result = findRecord( /* arguments */ );
if (result == nullptr) {
    …
}
```

这就没有任何歧义：**result**的结果一定是指针类型。

当模板出现时**nullptr**就更有用了。
假如你有一些函数只能被合适的已锁互斥量调用。
每个函数都有一个不同类型的指针：

```
int    f1(std::shared_ptr<Widget> spw);  // 只能被合适的
double f2(std::unique_ptr<Widget> upw);  // 已锁互斥量调
bool   f3(Widget* pw);                   // 用
```

如果这样传递空指针：

```
std::mutex f1m, f2m, f3m;          // 互斥量f1m，f2m，f3m，各种用于f1，f2，f3函数
using MuxGuard =                   // C++11的typedef，参见Item9
    std::lock_guard<std::mutex>;
     …
{
    MuxGuard g(f1m);               // 为f1m上锁
    auto result = f1(0);           // 向f1传递控制空指针
}                                  // 解锁
…
{
    MuxGuard g(f2m);               // 为f2m上锁
    auto result = f2(NULL);        // 向f2传递控制空指针
}                                  // 解锁
…
{
  MuxGuard g(f3m);                 // 为f3m上锁
  auto result = f3(nullptr);       // 向f3传递控制空指针
}                                  // 解锁
```

令人遗憾前两个调用没有使用**nullptr**，但是代码可以正常运行，这也许对一些东西有用。
但是重复的调用代码——为互斥量上锁，调用函数，解锁互斥量——更令人遗憾。它让人很烦。
模板就是被设计于减少重复代码，所以让我们模板化这个调用流程：

```
template<typename FuncType,
            typename MuxType,
            typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr)) {
    MuxGuard g(mutex);
    return func(ptr);
}
```

如果你对函数返回类型** (auto ... -> decltype(func(ptr)) **感到困惑不解，Item3可以帮助你。**
在C++14中代码的返回类型还可以被简化为decltype(auto)**:

```
template<typename FuncType,
             typename MuxType,
             typename PtrType>
decltype(auto) lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) {
    MuxGuard g(mutex);
    return func(ptr);
}
```

可以写这样的代码调用**lockAndCall**模板（两个都算）：

```
auto result1 = lockAndCall(f1, f1m, 0);          // 错误!
…
auto result2 = lockAndCall(f2, f2m, NULL);       // 错误!
…
auto result3 = lockAndCall(f3, f3m, nullptr);    // 没问题
```

代码虽然可以这样写，但是就像注释中说的，前两个情况不能通过编译。
在第一个调用中存在的问题是当0被传递给**lockAndCall**模板，模板类型推导会尝试去推导实参类型，
0的类型总是**int**，所以**int**版本的实例化中的func会被int类型的实参调用。
这与**f1**期待的参数**std::shared_ptr**不符。
传递**0**本来想表示空指针，结果**f1**得到的是和它相差十万八千里的**int**。
把**int**类型看做**std::shared_ptr**类型自然是一个类型错误。
在模板**lockAndCall**中使用**0**之所以失败是因为得到的是**int**但实际上模板期待的是一个
std::shared_ptr

第二个使用**NULL**调用的分析也是一样的。当**NULL**被传递给**lockAndCall**，形参ptr被推导为整型*[1]*，
然后当ptr——一个int或者类似int的类型——传递给f2的时候就会出现类型错误。当ptr被传递给f3的时候，
隐式转换使**std::nullptr_t**转换为**Widget***，因为**std::nullptr_t**可以隐式转换为任何指针类型。

模板类型推导将**0**和**NULL**推导为一个错误的类型，这就导致它们的替代品**nullptr**很吸引人。
使用**nullptr**，模板不会有什么特殊的转换。
另外，使用**nullptr**不会让你受到同重载决议特殊对待**0**和**NULL**一样的待遇。
当你想用一个空指针，使用**nullptr**，不用**0**或者**NULL**。

记住

- 优先考虑nullptr而非0和NULL
- 避免重载指针和整型

译注

[0] 请务必注意结合上下文使用这条规则

[1] 由于依赖于具体实现所以不一定是整数类型，所以用整型泛指int,long等类型

Item 9:Prefer alias declarations to typedefs

条款九:优先考虑别名声明而非typedefs

我相信每个人都同意使用STL容器是个好主意，并且我希望Item18能说服你让你觉得使用**std:unique_ptr**也是个好主意，但我猜没有人喜欢写上几次`std::unique_ptr<std::unordered_map<std::string,std::string>>`这样的类型，它可能会让你患上腕管综合征的风险大大增加。

避免上述医疗悲剧也很简单，引入**typedef**即可:

```
typedef  std::unique_ptr<std::unordered_map<std::string, std::string>>
 UPtrMapSS;
```

但**typedef**是C++98的东西。虽然它可以在C++11中工作，但是C++11也提供了一个别名声明（alias declaration）:

```
using UPtrMapSS =  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

由于这里给出的**typedef**和别名声明做的都是完全一样的事情，我们有理由想知道会不会出于一些技术上的原因两者有一个更好。

这里，在说它们之前我想提醒一下很多人都发现当声明一个函数指针时别名声明更容易理解:

```
// FP是一个指向函数的指针的同义词，它指向的函数带有int和const std::string&形参，不返回任何东西
typedef void (*FP)(int, const std::string&);      // typedef

//同上
using FP = void (*)(int, const std::string&);     // 别名声明
```

当然，两个结构都不是非常让人满意，没有人喜欢花大量的时间处理函数指针类型的别名*[0]*，所以至少在这里，没有一个吸引人的理由让你觉得别名声明比**typedef**好。

不过有一个地方使用别名声明吸引人的理由是存在的：模板。特别的，别名声明可以被模板化但是**typedef**不能。
这使得C++11程序员可以很直接的表达一些C++98程序员只能把**typedef**嵌套进模板化的**struct**才能表达的东西，
考虑一个链表的别名，链表使用自定义的内存分配器，**MyAlloc**。

使用别名模板，这真是太容易了:

```
template<typename T>
using MyAllocList = std::list<T,MyAlloc<T>>;

MyAllocList<Widget> lw;
```

使用**typedef**，你就只能从头开始:

```
template<typename T>
struct MyAllocList {
    typedef std::list<T, MyAlloc<T>> type;
};
MyAllocList<Widget>::type lw;
```

更糟糕的是，如果你想使用在一个模板内使用**typedef**声明一个持有链表的对象，而这个对象又使用了模板参数，你就不得不在在**typedef**前面加上**typename**

```
template<typename T>
class Widget {
private:
    typename MyAllocList<T>::type list;
    …
};
```

这里**MyAllocList::type**使用了一个类型，这个类型依赖于模板参数**T**。
因此**MyAllocList::type**是一个依赖类型，在C++很多讨人喜欢的规则中的一个提到必须要在依赖类型名前加上**typename**。
如果使用别名声明定义一个**MyAllocList**，就不需要使用**typename**（同时省略麻烦的**::type**后缀），

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;  // as before
template<typename T>
class Widget {
private:
    MyAllocList<T> list;
     …
};
```

对你来说，**MyAllocList**（使用了模板别名声明的版本）可能看起来和**MyAllocList::type**（使用**typedef**的版本）一样都应该依赖模板参数**T**，但是你不是编译器。
当编译器处理**Widget**模板时遇到**MyAllocList**（使用模板别名声明的版本），它们知道**MyAllocList**是一个类型名，
因为**MyAllocList**是一个别名模板。它一定是一个类型名。因此**MyAllocList**就是一个非依赖类型，就不要求必须使用**typename**。

当编译器在**Widget**的模板中看到**MyAllocList::type**（使用**typedef**的版本），它不能确定那是一个类型的名称。
因为可能存在**MyAllocList**的一个特化版本没有**MyAllocList::type**。
那听起来很不可思议，但不要责备编译器穷尽考虑所有可能。
举个例子，一个误入歧途的人可能写出这样的代码：

```
class Wine { … };
template<>                          // 当T是Wine
class MyAllocList<Wine> {           // 特化MyAllocList
private:
    enum class WineType             // 参见Item10了解
    { White, Red, Rose };           // "enum class"
    WineType type;                  // 在这个类中，type是
    …                               // 一个数据成员!
};
```

就像你看到的，**MyAllocList::type**不是一个类型。
如果**Widget**使用**Wine**实例化，在**Widget**模板中的**MyAllocList::type**将会是一个数据成员，不是一个类型。
在**Widget**模板内，如果**MyAllocList::type**表示的类型依赖于**T**，编译器就会坚持要求你在前面加上**typename**。

如果你尝试过模板元编程（TMP），你一定会碰到取模板类型参数然后基于它创建另一种类型的情况。
举个例子，给一个类型**T**，如果你想去掉**T**的常量修饰和引用修饰，比如你想把**const std::string&**变成**const std::string**。
又或者你想给一个类型加上**const**或左值引用，比如把**Widget**变成**const Widget**或**Widget&**。
（如果你没有用过玩过模板元编程，太遗憾了，因为如果你真的想成为一个高效C++程序员*[1]*，至少你需要熟悉C++的基础。你可以看看我在Item23，27提到的类型转换）。
C++11在*type traits*中给了你一系列工具去实现类型转换，如果要使用这些模板请包含头文件**<type_traits>**。
里面不全是类型转换的工具，也包含一些`predictable`接口的工具。给一个类型**T**，你想将它应用于转换中，结果类型就是**std::transformation<T>::type**，比如：

```
std::remove_const<T>::type           // 从const T中产出T
std::remove_reference<T>::type       // 从T&和T&&中产出T
std::add_lvalue_reference<T>::type   // 从T中产出T&
```

注释仅仅简单的总结了类型转换做了什么，所以不要太随便的使用。
在你的项目使用它们之前，你最好看看它们的详细说明书。
尽管写了一些，但我这里不是想给你一个关于type traits使用的教程。注意类型转换尾部的**::type**。
如果你在一个模板内部使用类型参数，你也需要在它们前面加上**typename**。
至于为什么要这么做是因为这些type traits是通过在**struct**内嵌套**typedef**来实现的。
是的，它们使用类型别名*[2]*技术实现，而正如我之前所说这比别名声明要差。

关于为什么这么实现是有历史原因的，但是我们跳过它（我认为太无聊了），因为标准委员会没有及时认识到别名声明是更好的选择，所以直到C++14它们才提供了使用别名声明的版本。
这些别名声明有一个通用形式：对于C++11的类型转换**std::transformation::type**在C++14中变成了**std::transformation_t**。
举个例子或许更容易理解：

```
std::remove_const<T>::type           // C++11: const T → T
std::remove_const_t<T>               // C++14 等价形式

std::remove_reference<T>::type       // C++11: T&/T&& → T
std::remove_reference_t<T>           // C++14 等价形式

std::add_lvalue_reference<T>::type   // C++11: T → T&
std::add_lvalue_reference_t<T>       // C++14 等价形式
```

C++11的的形式在C++14中也有效，但是我不能理解为什么你要去用它们。
就算你没有使用C++14，使用别名模板也是小儿科
只需要C++11，甚至每个小孩都能仿写。
对吧？如果你有一份C++14标准，就更简单了，只需要复制粘贴：

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t =  typename add_lvalue_reference<T>::type;
```

看见了吧？不能再简单了。

记住

- typedef不支持模板化，但是别名声明支持。
- 别名模板避免了使用"**::type**"后缀，而且在模板中使用**typedef**还需要在前面加上**typename**
- C++14提供了C++11所有类型转换的别名声明版本

译注

[0] 即FP

[1] 哈，这大概是作为《Modern C++ Design -Generic Programming and Design Pattern Applied》的作者的Scott Meyes才能说出的话。

[2] 作者所言的类型别名是泛指typedef和using语法进行的别名操作，根据上下文这里的类型别名指的是使用typedef

Item 10:优先考虑限域枚举而非未限域枚举

条款10:优先考虑限域枚举而非未限域枚举

通常来说，在花括号中声明一个名字会限制它的作用域在花括号之内。但这对于C++98风格的enum中声明的枚举名是不成立的。这些在enum作用域中声明的枚举名所在的作用域也包括enum本身，也就是说这些枚举名和enum所在的作用域中声明的相同名字没有什么不同

```
enum Color { black, white, red };   // black, white, red 和
                                    // Color一样都在相同作用域
auto white = false;                 // 错误! white早已在这个作用
                                    // 域中存在
```

事实上这些枚举名泄漏进和它们所被定义的enum域一样的作用域。有一个官方的术语：未限域枚举(unscoped enum)在C++11中它们有一个相似物，限域枚举(scoped enum)，它不会导致枚举名泄漏：

```
enum class Color { black, white, red }; // black, white, red
                                        // 限制在Color域内
auto white = false;                     // 没问题，同样域内没有这个名字

Color c = white;                        //错误，这个域中没有white

Color c = Color::white;                 // 没问题
auto c = Color::white;                  // 也没问题（也符合条款5的建议）
```

因为限域枚举是通过**enum class**声明，所以它们有时候也被称为枚举类(enum classes)。

使用限域枚举减少命名空间污染是一个足够合理使用它而不是它的同胞未限域枚举的理由，其实限域枚举还有第二个吸引人的优点：在它的作用域中,枚举名是强类型。未限域枚举中的枚举名会隐式转换为整型（现在，也可以转换为浮点类型）。因此下面这种歪曲语义的做法也是完全有效的：

```
enum Color { black, white, red };       // 未限域枚举
std::vector<std::size_t>                // func返回x的质因子
primeFactors(std::size_t x);
Color c = red;
…
if (c < 14.5) {                         // Color与double比较 (!)
    auto factors =                      // 计算一个Color的质因子(!)
    primeFactors(c);
…
}
```

在**enum**后面写一个**class**就可以将非限域枚举转换为限域枚举，接下来就是完全不同的故事展开了。现在不存在任何隐式转换可以将限域枚举中的枚举名转化为任何其他类型。

```
enum class Color { black, white, red }; // Color现在是限域枚举
Color c = Color::red;                   // 和之前一样，只是
…                                       // 多了一个域修饰符
if (c < 14.5) {                         // 错误！不能比较
                                        // Color和double
    auto factors =                      // 错误！不能向参数为std::size_t的函数
        primeFactors(c);                // 传递Color参数
    …
}
```

如果你真的很想执行Color到其他类型的转换，和平常一样，使用正确的类型转换运算符扭曲类型系统：

```
if (static_cast<double>(c) < 14.5) { // 奇怪的代码，但是
                                     // 有效
    auto factors = // suspect, but
        primeFactors(static_cast<std::size_t>(c)); // 能通过编译
    …
}
```

似乎比起非限域枚举而言限域枚举有第三个好处，因为限域枚举可以前置声明。比如，它们可以不指定枚举名直接前向声明：

```
enum Color;         // 错误!
enum class Color;   // 没问题
```

其实这是一个误导。在C++11中，非限域枚举也可以被前置声明，但是只有在做一些其他工作后才能实现。这些工作来源于一个事实：
在C++中所有的枚举都有一个由编译器决定的整型的基础类型。对于非限域枚举比如`Color`，

```
enum Color { black, white, red };
```

编译器可能选择`char`作为基础类型，因为这里只需要表示三个值。然而，有些枚举中的枚举值范围可能会大些，比如：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
              };
```

这里值的范围从**0**到**0xFFFFFFFF**。除了在不寻常的机器上（比如一个**char**至少有32bits的那种），编译器都会选择一个比**char**大的整型类型来表示**Status**。

为了高效使用内存，编译器通常在确保能包含所有枚举值的前提下为枚举选择一个最小的基础类型。在一些情况下，编译器
将会优化速度，舍弃大小，这种情况下它可能不会选择最小的基础类型，而是选择对优化大小有帮助的类型。为此，C++98
只支持枚举定义（所有枚举名全部列出来）；枚举声明是不被允许的。这使得编译器能为之前使用的每一个枚举选择一个基础类型。

但是不能前置声明枚举也是有缺点的。最大的缺点莫过于它可能增加编译依赖。再次考虑**Status**枚举：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
              };
```

这种**enum**很有可能用于整个系统，因此系统中每个包含这个头文件的组件都会依赖它。如果引入一个新状态值，

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
              };
```

那么可能整个系统都得重新编译，即使只有一个子系统——或者一个函数使用了新添加的枚举名。这是大家都不希望看到的。C++11中的前置声明可以解决这个问题。
比如这里有一个完全有效的限域枚举声明和一个以该限域枚举作为形参的函数声明：

```
enum class Status; // forward declaration
void continueProcessing(Status s); // use of fwd-declared enum
```

即使**Status**的定义发生改变，包含这些声明的头文件也不会重新编译。而且如果**Status**添加一个枚举名（比如添加一个*audited*），**continueProcessing**的行为不受影响（因为**continueProcessing**没有使用这个新添加的*audited*），**continueProcessing**也不需要重新编译。
但是如果编译器在使用它之前需要知晓该枚举的大小，该怎么声明才能让C++11做到C++98不能做到的事情呢？
答案很简单：限域枚举的基础类型总是已知的，而对于非限域枚举，你可以指定它。默认情况下，限域枚举的基础类型是`int`：

```
enum class Status; // 基础类型是int
```

如果默认的*int*不适用，你可以重写它：

```
enum class Status: std::uint32_t;   // Status的基础类型
                                    // 是std::uint32_t
                                    // (需要包含 <cstdint>)
```

不管怎样，编译器都知道限域枚举中的枚举名占用多少字节。要为非限域枚举指定基础类型，你可以同上，然后前向声明一下：

```
enum Color: std::uint8_t;   // 为非限域枚举Color指定
                            // 基础为
                            // std::uint8_t
```

基础类型说明也可以放到枚举定义处：

```
enum class Status: std::uint32_t { good = 0,
                                   failed = 1,
                                   incomplete = 100,
                                   corrupt = 200,
                                   audited = 500,
                                   indeterminate = 0xFFFFFFFF
                                   };
```

限域枚举避免命名空间污染而且不接受隐式类型转换，但它并非万事皆宜，你可能会很惊讶听到至少有一种情况下非限域枚举是很有用的。
那就是获取C++11 tuples中的字段的时候。比如在社交网站中，假设我们有一个`tuple`保存了用户的名字，email地址，声望点：

```
using UserInfo = // 类型别名，参见Item 9
    std::tuple<std::string, // 名字
    std::string, // email地址
    std::size_t> ; // 声望
```

虽然注释说明了tuple各个字段对应的意思，但当你在另文件遇到下面的代码那之前的注释就不是那么有用了：

```
UserInfo uInfo; // tuple对象
…
auto val = std::get<1>(uInfo); // 获取第一个字段
```

作为一个程序员，你有很多工作要持续跟进。你应该记住第一个字段代表用户的email地址吗？我认为不。
可以使用非限域枚举将名字和字段编号关联起来以避免上述需求：

```
enum UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo;
…
auto val = std::get<uiEmail>(uInfo); // , 获取用户email
```

之所以它能正常工作是因为`UserInfoFields`中的枚举名隐式转换成**std::size_t**了,其中**std::size_t**是**std::get**模板实参所需的。
对应的限域枚举版本就很啰嗦了：

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo; // as before
…
auto val =
std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
(uInfo);
```

为避免这种冗长的表示，我们可以写一个函数传入枚举名并返回对应的**std::size_t**值，但这有一点技巧性。
std::get是一个模板（函数），需要你给出一个**std::size_t**值的模板实参（注意使用`<>`而不是`()`），因此将枚举名变换为**std::size_t**值会发生在编译期。
如Item 15提到的，那必须是一个**constexpr**模板函数。
事实上，它也的确该是一个**constexpr**函数，因为它应该能用于任何`enum`。
如果我们想让它更一般化，我们还要泛化它的返回类型。较之于返回**std::size_t**，我们更应该泛化枚举

的基础类型。
这可以通过**std::underlying_type**这个`type trait`获得。（参见Item 9关于type trait的内容）。
最终我们还要再加上**noexcept**修饰（参见Item 14），因为我们知道它肯定不会产生异常。
根据上述分析最终得到的**toUType**模板函数在编译期接受任意枚举名并返回它的值：

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
      std::underlying_type<E>::type>(enumerator);
}
```

在C++14中，**toUType**还可以进一步用`std::underlying_type_t`（参见Item 9）代替`typename std::underly ing_type<E>::type`打磨：

```
template<typename E> // C++14
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

还可以再用C++14 auto（参见Item 3）打磨一下代码：

```
template<typename E> // C++14
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

不管它怎么写，**toUType**现在允许这样访问tuple的字段了：

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

比起使用非限域枚举，限域有很多可圈可点的地方，它避免命名空间污染，防止不经意间使用隐式转换。
（下面这句我没看懂，保留原文。。（是什么典故吗。。。））
In many cases, you
may decide that typing a few extra characters is a reasonable price to pay for the ability
to avoid the pitfalls of an enum technology that dates to a time when the state of
the art in digital telecommunications was the 2400-baud modem.

记住

- C++98的枚举即非限域枚举
- 限域枚举的枚举名仅在enum内可见。要转换为其它类型只能使用cast。
- 非限域/限域枚举都支持基础类型说明语法，限域枚举基础类型默认是`int`。非限域枚举没有默认基础类型。
- 限域枚举总是可以前置声明。非限域枚举仅当指定它们的基础类型时才能前置。

Item 11:优先考虑使用deleted函数而非使用未定义的私有声明

条款11:优先考虑使用deleted函数而非使用未定义的私有声明

如果你写的代码要被其他人使用，你不想让他们调用某个特殊的函数，你通常不会声明这个函数。无声明，不函数。简简单单！但有时C++会给你自动声明一些函数，如果你想防止客户调用这些函数，事情就不那么简单了。

上述场景见于特殊的成员函数，即当有必要时C++自动生成的那些函数。Item 17 详细讨论了这些函数，但是现在，我们只关心拷贝构造函数和拷贝赋值运算符重载。This chapter is largely devoted to common practices in
C++98 that have been superseded by better practices in C++11, and in C++98, if you want to suppress use of a member function, it's almost always the copy constructor, the assignment operator, or both.

在C++98中防止调用这些函数的方法是将它们声明为私有成员函数。举个例子，在C++ 标准库*iostream*继承链的顶部是模板类`basic_ios`。所有`istream`和`ostream`类都继承此类(直接或者间接)。拷贝`istream`和`ostream`是不合适的，因为要进行哪些操作是模棱两可的。比如一个`istream`对象，代表一个输入值的流，流中有一些已经被读取，有一些可能马上要被读取。如果一个`istream`被拷贝，需要像拷贝将要被读取的值那样也拷贝已经被读取的值吗？解决这个问题最好的方法是不定义这个操作。直接禁止拷贝流。

要使`istream`和`ostream`类不可拷贝，`basic_ios`在C++98中是这样声明的(包括注释):

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
    …
private:
    basic_ios(const basic_ios& ); // not defined
    basic_ios& operator=(const basic_ios&); // not defined
};
```

将它们声明为私有成员可以防止客户端调用这些函数。故意不定义它们意味着假如还是有代码用它们就会在链接时引发缺少函数定义(missing function definitions)错误。

在C++11中有一种更好的方式，只需要使用相同的结尾：用`= delete`将拷贝构造函数和拷贝赋值运算符标记为`deleted`函数。上面相同的代码在C++11中是这样声明的:

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
    …
    basic_ios(const basic_ios& ) = delete;
    basic_ios& operator=(const basic_ios&) = delete;
    …
};
```

删除这些函数(译注：添加"= delete")和声明为私有成员可能看起来只是方式不同，别无其他区别。其实还有一些实质性意义。`deleted`函数不能以任何方式被调用，即使你在成员函数或者友元函数里面调用`deleted`函数也不能通过编译。这是较之C++98行为的一个改进，后者不正确的使用这些函数在链接时才被诊断出来。

通常，`deleted`函数被声明为*public*而不是*private*.这也是有原因的。当客户端代码试图调用成员函数，C++会在检查`deleted`状态前检查它的访问性。当客户端代码调用一个私有的`deleted`函数，一些编译器只会给出该函数是*private*的错误(译注：而没有诸如该函数被`deleted`修饰的错误)，即使函数的访问性不影响它的使用。所以值得牢记，如果要将老代码的"私有且未定义"函数替换为`deleted`函数时请一并修改它的访问性为*public*，这样可以让编译器产生更好的错误信息。

`deleted`函数还有一个重要的优势是任何函数都可以标记为`deleted`，而只有*private*只能修饰成员函数。假如我们有一个非成员函数，它接受一个整型参数，检查它是否为幸运数：

```
bool isLucky(int number);
```

C++有沉重的C包袱，使得含糊的、能被视作数值的任何类型都能隐式转换为`int`，但是有一些调用可能是没有意义的：

```
if (isLucky('a')) … // 字符'a'是幸运数?
if (isLucky(true)) … // "true"是?
if (isLucky(3.5)) … // 难道判断它的幸运之前还要先截尾成3?
```

如果幸运数必须真的是整数，我们该禁止这些调用通过编译。
其中一种方法就是创建`deleted`重载函数，其参数就是我们想要过滤的类型：

```
bool isLucky(int number); // 原始版本
bool isLucky(char) = delete; // 拒绝char
bool isLucky(bool) = delete; // 拒绝bool
bool isLucky(double) = delete; // 拒绝float和double
```

(上面double重载版本的注释说拒绝float和double可能会让你惊讶，但是请回想一下：将`float`转换为`int`和`double`，C++更喜欢转换为`double`。使用`float`调用`isLucky`因此会调用`double`重载版本，而不是`int`版本。好吧，它也会那么去尝试。事实是调用被删除的`double`重载版本不能通过编译。不再惊讶了吧。)

虽然`deleted`寒暑假不能被使用，它它们还是存在于你的程序中。也即是说，重载决议会考虑它们。这也是为什么上面的函数声明导致编译器拒绝一些不合适的函数调用。

```
if (isLucky('a')) … //错误! 调用deleted函数
if (isLucky(true)) … // 错误!
if (isLucky(3.5f)) … // 错误!
```

另一个`deleted`函数用武之地（private成员函数做不到的地方）是禁止一些模板的实例化。
假如你要求一个模板仅支持原生指针（尽管第四章建议使用智能指针代替原生指针）

```
template<typename T>
void processPointer(T* ptr);
```

在指针的世界里有两种特殊情况。一是`void*`指针，因为没办法对它们进行解引用，或者加加减减等。另一种指针是`char*`，因为它们通常代表C风格的字符串，而不是正常意义下指向单个字符的指针。这两种情况要特殊处理，在`processPointer`模板里面，我们假设正确的函数应该拒绝这些类型。
也即是说，`processPointer`不能被`void*`和`char*`调用。
要想确保这个很容易，使用`delete`标注模板实例：

```
template<>
void processPointer<void>(void*) = delete;
template<>
void processPointer<char>(char*) = delete;
```

现在如果使用`void*`和`char*`调用`processPointer`就是无效的，按常理说`const void*`和`const void*`也应该无效，所以这些实例也应该标注`delete`：

```
template<>
void processPointer<const void>(const void*) = delete;
template<>
void processPointer<const char>(const char*) = delete;
```

如果你想做得更彻底一些，你还要删除`const volatile void*`和`const volatile char*`重载版本，另外还需要一并删除其他标准字符类型的重载版本：`std::wchar_t`,`std::char16_t`和`std::char32_t`。

有趣的是，如果的类里面有一个函数模板，你可能想用`private`（经典的C++98惯例）来禁止这些函数模板实例化，但是不能这样做，因为不能给特化的模板函数指定一个不同（于函数模板）的访问级别。如果`processPointer`是类`Widget`里面的模板函数， 你想禁止它接受`void*`参数，那么通过下面这样C++98的方法就不能通过编译：
compile:

```
class Widget {
public:
    …
    template<typename T>
    void processPointer(T* ptr)
    { … }
private:
    template<> // 错误!
    void processPointer<void>(void*);
};
```

问题是模板特例化必须位于一个命名空间作用域，而不是类作用域。`delete`不会出现这个问题，因为它不需要一个不同的访问级别，且他们可以在类外被删除（因此位于命名空间作用域）：

```
class Widget {
public:
    …
    template<typename T>
    void processPointer(T* ptr)
    { … }
    …
};
template<>
void Widget::processPointer<void>(void*) = delete; // 还是public，但是已经被删除了
```

事实上C++98的最佳实践即声明函数为*private*但不定义是在做C++11 delete函数要做的事情。作为模仿者，C++98的方法不是十全十美。它不能在类外正常工作，不能总是在类中正常工作，它的罢工可能直到链接时才会表现出来。所以请坚定不移的使用`delete`函数。

记住：

- 比起声明函数为private但不定义，使用delete函数更好
- 任何函数都能`delete`，包括非成员函数和模板实例

译注：
+本条款`delete`，`deleted`,删除视情况使用，都表示一个意思.`删除函数`和`delete函数`也是如此

- 函数模板意指未特化前的源码，模板函数则倾向于模板实例化后的函数

Item 12:使用override声明重载函数

条款12:使用override声明重载函数

在C++面向对象的世界里，涉及的概念有类，继承，虚函数。这个世界最基本的概念是派生类的虚函数重写基类同名函数。令人遗憾的是虚函数重写可能一不小心就错了。给人感觉语言的这一部分设计观点是墨菲定律不是用来遵守的，只是值得尊敬的。

鉴于"重写"听起来像"重载"，尽管两者完全不相关，下面就通过一个派生类和基类来说明什么是虚函数重写:

```
class Base {
public:
    virtual void doWork(); // 基类虚函数
    …
};
class Derived: public Base {
    public:
    virtual void doWork(); // 重写Base::doWork(这里"virtual"是可以省略的)
    …
};
std::unique_ptr<Base> upb =          // 创建基类指针
    std::make_unique<Derived>();     // 指向派生类对象
                                     // 关于std: : make_unique请
…                                    // 参见Item1
upb->doWork();  // 通过基类指针调用doWork
                // 实际上是派生类的doWork
                // 函数被调用
```

要想重写一个函数，必须满足下列要求:

- 基类函数必须是`virtual`
- 基类和派生类函数名必须完全一样（除非是析构函数
- 基类和派生类函数参数必须完全一样
- 基类和派生类函数常量性(constness)必须完全一样
- 基类和派生类函数的返回值和异常说明(exception specifications)必须兼容
  除了这些C++98就存在的约束外，C++11又添加了一个:
- 函数的引用限定符（reference qualifiers）必须完全一样。成员函数的引用限定符是C++11很少抛头露脸的特性，所以如果你从没听过它无需惊讶。它可以限定成员函数只能用于左值或者右值。成员函数不需要`virtual`也能使用它们:

```
class Widget {
public:
    …
    void doWork() &; //只有*this为左值的时候才能被调用
    void doWork() &&; //只有*this为右值的时候才能被调用
};
…
Widget makeWidget();     // 工厂函数（返回右值）
Widget w;                // 普通对象（左值）
…
w.doWork(); // 调用被左值引用限定修饰的Widget::doWork版本
            // (即Widget::doWork &)
```

```
makeWidget().doWork(); // 调用被右值引用限定修饰的Widget::doWork版本
                       // (即Widget::doWork &&)
```

后面我还会提到引用限定符修饰成员函数，但是现在，只需要记住如果基类的虚函数有引用限定符，派生类的重写就必须具有相同的引用限定符。如果没有，那么新声明的函数还是属于派生类，但是不会重写父类的任何函数。

这么多的重写需求意味着哪怕一个小小的错误也会造成巨大的不同。
代码中包含重写错误通常是有效的，但它的意图不是你想要的。因此你不能指望当你犯错时编译器能通知你。比如，下面的代码是完全合法的，咋一看，还很有道理，但是它包含了非虚函数重写。你能识别每个case的错误吗，换句话说，为什么派生类函数没有重写同名基类函数?

```
class Base {
public:
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    void mf4() const;
};
class Derived: public Base {
public:
    virtual void mf1();
    virtual void mf2(unsigned int x);
    virtual void mf3() &&;
    void mf4() const;
};
```

需要一点帮助吗?

- `mf1`在基类声明为`const`,但是派生类没有这个常量限定符
- `mf2`在基类声明为接受一个`int`参数，但是在派生类声明为接受`unsigned int`参数
- `mf3`在基类声明为左值引用限定，但是在派生类声明为右值引用限定
- `mf4`在基类没有声明为虚函数
  你可能会想，“哎呀，实际操作的时候，这些warnings都能被编译器探测到，所以我不需要担心。”可能你说的对，也可能不对。就我目前检查的两款编译器来说，这些代码编译时没有任何warnings，即使我开启了输出所有warnings（其他编译器可能会为这些问题的部分输出warnings，但不是全部）

由于正确声明派生类的重写函数很重要，但很容易出错，C++11提供一个方法让你可以显式的将派生类函数指定为应该是基类重写版本：将它声明为`override`。还是上面那个例子，我们可以这样做:

```
class Derived: public Base {
    public:
    virtual void mf1() override;
    virtual void mf2(unsigned int x) override;
    virtual void mf3() && override;
    virtual void mf4() const override;
};
```

代码不能编译，当然了，因为这样写的时候，编译器会抱怨所有与重写有关的问题。这也是你想要的，以及为什么要在所有重写函数后面加上`override`。使用`override`的代码编译时看起来就像这样（假设我们的目的是重写基类的所有函数）：

```
class Base {
public:
```

```
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    virtual void mf4() const;
};
class Derived: public Base {
public:
    virtual void mf1() const override;
    virtual void mf2(int x) override;
    virtual void mf3() & override;
    void mf4() const override; // 可以添加virtual，但不是必要
};
```

注意在这个例子中`mf4`有别于之前，它在`Base`中的声明有`virtual`修饰，所以能正常工作。
大多数和重写有关的错误都是在派生类引发的，但也可能是基类的不正确导致。

比起让编译器（译注：通过warnings）告诉你"将要"重写实际不会重写，不如给你的派生类成员函数全都加上`override`。如果你考虑修改修改基类虚函数的函数签名，`override`还可以帮你评估后果。
如果派生类全都用上`override`，你可以只改变基类函数签名，重编译系统，再看看你造成了多大的问题（即，多少派生类不能通过编译），然后决定是否值得如此麻烦更改函数签名。没有重写，你只能寄希望于完善的单元测试，因为，正如我们所见，派生类虚函数本想重写基类，但是没有，编译器也没有探测并发出诊断信息。

C++既有很多关键字，C++11引入了两个上下文关键字(contextual keywords),`override`和`final`（向虚函数添加`final`可以防止派生类重写。`final`也能用于类，这时这个类不能用作基类）。
这两个关键字的特点是它们是保留的，它们只是位于特定上下文才被视为关键字。对于`override`，它只在成员函数声明结尾处才被视为关键字。这意味着如果你以前写的代码里面已经用过**override**这个名字，那么换到C++11标准你也无需修改代码：

```
class Warning { // potential legacy class from C++98
public:
    …
void override(); // C++98和C++11都合法
};
```

关于`override`想说的就这么多，但对于成员函数引用限定(reference qualifiers)还有一些内容。我之前承诺我会在后面提供更多的关于它们的资料，现在就是"后面"了。
如果我们想写一个函数只接受左值实参，我们的声明可以包含一个左值引用形参：

```
void doSomething(Widget& w); // 只接受左值Widget对象
```

如果我们想写一个函数只接受右值实参，我们的声明可以包含一个右值引用形参：

```
void doSomething(Widget&& w); // 只接受右值Widget对象
```

成员函数的引用限定可以很容易的区分哪个成员函数被对象调用（即`*this`）。它和在成员函数声明尾部添加一个`const`暗示该函数的调用者（即`*this`）是`const`很相似。
对成员函数添加引用限定不常见，但是可以见。
举个例子，假设我们的`Widget`类有一个`std::vector`数据成员，我们提供一个范围函数让客户端可以直接访问它：

```
class Widget {
public:
    using DataType = std::vector<double>; // 参见Item
    …
    DataType& data() { return values; }
    …
private:
    DataType values;
};
```

这是最具封装性的设计，只给外界保留一线光。但先把这个放一边，思考一下下面的客户端代码：

```
Widget w;
…
auto vals1 = w.data(); // 拷贝w.values到vals1
```

Widget::data函数的返回值是一个左值引用（准确的说是`std::vector<double>&`），
因为左值引用是左值，`vals1`从左值初始化，因此它由`w.values`拷贝构造而得，就像注释说的那样。
现在假设我们有一个创建`Widgets`的工厂函数，

```
Widget makeWidget();
```

我们想用`makeWidget`返回的`std::vector`初始化一个变量：

```
auto vals2 = makeWidget().data(); // 拷贝Widget里面的值到vals2
```

再说一次，`Widgets::data`返回的是左值引用，还有，左值引用是左值。所以，我们的对象(vals2)又得从Widget里的values拷贝构造。这一次，`Widget`是`makeWidget`返回的临时对象（即右值），所以将其中的`std::vector`进行拷贝纯属浪费。最好是移动，但是因为`data`返回左值引用，C++的规则要求编译器不得不生成一个拷贝。
我们需要的是指明当`data`被右值`Widget`对象调用的时候结果也应该是一个右值。
现在就可以使用引用限定写一个重载函数来达成这一目的：

```
class Widget {
public:
    using DataType = std::vector<double>;
    …
    DataType& data() &              // 对于左值Widgets,
    { return values; }              // 返回左值
    DataType data() &&              // 对于右值Widgets,
    { return std::move(values); }   // 返回右值
    …
private:
    DataType values;
};
```

注意`data`重载的返回类型是不同的，左值引用重载版本返回一个左值引用，右值引用重载返回一个临时对象。这意味着现在客户端的行为和我们的期望相符了：

```
auto vals1 = w.data();              //调用左值重载版本的Widget::data，拷贝构造vals1
auto vals2 = makeWidget().data();   //调用右值重载版本的Widget::data，移动构造vals2
```

这真的很nice，但别被这结尾的暖光照耀分心以致忘记了该条款的中心。这个条款的中心是只要你在派生类声明想要重写基类虚函数的函数，就加上`override`。

记住：

- 为重载函数加上`override`
- 成员函数限定让我们可以区别对待左值对象和右值对象（即`*this`）

Item 13:优先考虑const_iterator而非iterator

条款 13:优先考虑const_iterator而非iterator

STL **const_iterator**等价于指向常量的指针。它们都指向不能被修改的值。标准实践是能加上**const**就加上，这也指示我们对待**const_iterator**应该如出一辙。

上面的说法对C++11和C++98都是正确的，但是在C++98中，标准库对**const_iterator**的支持不是很完整。首先不容易创建它们，其次就算你有了它，它的使用也是受限的。
假如你想在`std::vector<int>`中查找第一次出现**1983**(C++代替C with classes的那一年)的位置，然后插入**1998**（第一个ISO C++标准被接纳的那一年）。如果vector中没有1983，那么就在vector尾部插入。在C++98中使用**iterator**可以很容易做到：

```
std::vector<int> values;
…
std::vector<int>::iterator it =
std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

但是这里**iterator**真的不是一个好的选择，因为这段代码不修改**iterator**指向的内容。用**const_iterator**重写这段代码是很平常的，但是在C++98中就不是了。下面是一种概念上可行但是不正确的方法：

```
typedef std::vector<int>::iterator IterT;   // typetypedef
std::vector<int>::const_iterator ConstIterT; // defs
std::vector<int> values;
…
ConstIterT ci =
    std::find(static_cast<ConstIterT>(values.begin()), // cast
        static_cast<ConstIterT>(values.end()), // cast
        1983);
values.insert(static_cast<IterT>(ci), 1998); // 可能无法通过编译，原因见下
```

typedef不是强制的，但是可以让类型转换更好写。（你可能想知道为什么我使用**typedef**而不是Item 9提到的别名声明，因为这段代码在演示C++98做法，别名声明是C++11加入的特性）

之所以`std::find`的调用会出现类型转换是因为在C++98中**values**是非常量容器，没办法简简单单的从非常量容器中获取**const_iterator**。严格来说类型转换不是必须的，因为用其他方法获取**const_iterator**也是可以的（比如你可以把values绑定到常量引用上，然后再用这个变量代替values），但不管怎么说，从非常量容器中获取**const_iterator**的做法都有点别扭。

当你费劲地获得了**const_iterator**，事情可能会变得更糟，因为C++98中，插入操作的位置只能由**iterator**指定，**const_iterator**是不被接受的。这也是我在上面的代码中，将**const_iterator**转换为**iterat**的原因，因为向**insert**传入**const_iterator**不能通过编译。

老实说，上面的代码也可能无法编译，因为没有一个可移植的从**const_iterator**到**iterator**的方法，即使使用`static_cast`也不行。甚至传说中的牛刀**reinterpret_cast**也杀不了这条鸡。（它C++98的限制，也不是C++11的限制，只是**const_iterator**就是不能转换为**iterator**，不管看起来对它们施以转换是有多么合理。）不过有办法生成一个**iterator**，使其指向和**const_iterator**指向相同，但是看起来不明显，也没有广泛应用，在这本书也不值得讨论。除此之外，我希望目前我陈述的观点是清晰的：**const_iterator**在C++98中会有很多问题。这一天结束时，开发者们不再相信能加**const**就加它的教条，而是只在实用的地方加它，C++98的**const_iterator**不是那么实用。

所有的这些都在C++11中改变了，现在**const_iterator**即容易获取又容易使用。容器的成员函数`cbegin`和`cend`产出**const_iterator**，甚至对于非常量容器，那些之前只使用**iterator**指示位置的STL成员函数也可以使用**const_iterator**了。使用C++11 **const_iterator**重写C++98使用**iterator**的代码也稀松平常：

```
std::vector<int> values; // 和之前一样
…
auto it =                                           // 使用cbegin
    std::find(values.cbegin(),values.cend(), 1983); // 和cend
values.insert(it, 1998);
```

现在使用**const_iterator**的代码就很实用了！

唯一一个C++11对于**const_iterator**支持不足（译注：C++14支持但是C++11的时候还没）的情况是：当你想写最大程度通用的库，并且这些库代码为一些容器和类似容器的数据结构提供非成员函数**begin**、**end**（以及**cbegin**，**cend**，**rbegin**，**rend**）而不是成员函数（其中一种情况就是原生数组）。最大程度通用的库会考虑使用非成员函数而不是假设成员函数版本存在。

举个例子，我们可以泛化下面的`findAndInsert`：

```
template<typename C, typename V>
void findAndInsert(C& container,     // 在容器中查找第一次
const V& targetVal,                  // 出现targetVal的位置，
const V& insertVal)                  // 然后插入insertVal
{
    using std::cbegin; // there
    using std::cend;
    auto it = std::find(cbegin(container),  // 非成员函数cbegin
                        cend(container),    // 非成员函数cend
                        targetVal);
    container.insert(it, insertVal);
}
```

它可以在C++14工作良好，但是很遗憾，C++11不在良好之列。由于标准化的疏漏，C++11只添加了非成员函数**begin**和**end**，但是没有添加**cbegin**，**cend**，**rbegin**，**rend**，**crbegin**，**crend**。C++14修订了这个疏漏，如果你使用C++11，并且想写一个最大程度通用的代码，而你使用的STL没有提供缺失的非成员函数**cbegin**和它的朋友们，你可以简单的抛出你自己的实现。比如，下面就是非成员函数**cbegin**的实现：

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
    return std::begin(container); // 解释见下
}
```

你可能很惊讶非成员函数**cbegin**没有调用成员函数**cbegin**吧？但是请跟逻辑走。这个**cbegin**模板接受任何容器或者类似容器的数据结构`C`，并且通过`const`引用访问第一个实参**container**。如果`C`是一个普通的容器类型（如`std::vector<int>`），**container**将会引用一个常量版本的容器（即`const std::vector<int>&`）。对const容器调用非成员函数**begin**（由C++11提供)将产出**const_iterator**，这个迭代器也是模板要返回的。用这种方法实现的好处是就算容器只提供**begin**不提供**cbegin**也没问题。那么现在你可以将这个非成员函数**cbegin**施于只支持**begin**的容器。

如果**C**是原生数组，这个模板也能工作。这时，**container**成为一个const数组。C++11为数组提供特化版本的非成员函数**begin**，它返回指向数组第一个元素的指针。一个const数组的元素也是const，所以对于const数组，非成员函数**begin**返回指向const的指针。在数组的上下文中，所谓指向const的指针，也就是**const_iterator**了。

回到最开始，本条款的中心是鼓励你只要能就使用**const_iterator**。最原始的动机是——只要它有意义就加上const——C++98就有的思想。但是在C++98，它（译注：const_iterator）只是一般有用，到了C++11,它就是极其有用了，C++14在其基础上做了些修补工作。

记住

- 优先考虑const_iterator而非iterator
- 在最大程度通用的代码中，优先考虑非成员函数版本的**begin**，**end**，**rbegin**等，而非同名成员函数

Item 14:如果函数不抛出异常请使用noexcept

条款 14:如果函数不抛出异常请使用noexcept

在C++98中，异常说明（exception specifications）是喜怒无常的野兽。你不得不写出函数可能抛出的异常类型，如果函数实现有所改变，异常说明也可能需要修改。改变异常说明会影响客户端代码，因为调用者可能依赖原版本的异常说明。编译器不会为函数实现，异常说明和客户端代码中提供一致性保障。大多数程序员最终都认为不值得为C++98的异常说明如此麻烦。

在C++11标准化过程中，大家一致认为异常说明真正有用的信息是一个函数是否会抛出异常。非黑即白，一个函数可能抛异常，或者不会。这种"可能-绝不"的二元论构成了C++11异常说的基础，从根本上改变了C++98的异常说明。（C++98风格的异常说明也有效，但是已经标记为deprecated（废弃））。在C++11中，无条件的**noexcept**保证函数不会抛出任何异常。

关于一个函数是否已经声明为**noexcept**是接口设计的事。函数的异常抛出行为是客户端代码最关心的。调用者可以查看函数是否声明为**noexcept**，这个可以影响到调用代码的异常安全性和效率。

就其本身而言，函数是否为**noexcept**和成员函数是否**const**一样重要。如果知道这个函数不会抛异常就加上**noexcept**是简单天真的接口说明。

不过这里还有给不抛异常的函数加上**noexcept**的动机：它允许编译器生成更好的目标代码。
要想知道为什么，了解C++98和C++11指明一个函数不抛异常的方式是很有用了。考虑一个函数**f**，它允许调用者永远不会受到一个异常。两种表达方式如下：

```
int f(int x) throw(); // C++98风格
int f(int x) noexcept; // C++11风格
```

如果在运行时，**f**出现一个异常，那么就和**f**的异常说明冲突了。在C++98的异常说明中，调用栈会展开至**f**的调用者，一些不合适的动作比如程序终止也会发生。C++11异常说明的运行时行为明显不同：调用栈只是*可能*在程序终止前展开。
展开调用栈和*可能*展开调用栈两者对于代码生成（code generation）有非常大的影响。在一个**noexcept**函数中，当异常传播到函数外，优化器不需要保证运行时栈的可展开状态，也不需要保证**noexcept**函数中的对象按照构造的反序析构。而"**throw()**"标注的异常声明缺少这样的优化灵活性，它和没加一样。可以总结一下：

```
RetType function(params) noexcept;   // 极尽所能优化
RetType function(params) throw();    // 较少优化
RetType function(params);            // 较少优化
```

这是一个充分的理由使得你当知道它不抛异常时加上**noexcept**。

还有一些函数让这个案例更充分。移动操作是绝佳的例子。假如你有一份C++98代码，里面用到了`std::vector<Widget>`。**Widget**通过**push_back**一次又一次的添加进`std::vector`：

```
std::vector<Widget> vw;
…
Widget w;
…                    // work with w
vw.push_back(w);     // add w to vw
```

假设这个代码能正常工作，你也无意修改为C++11风格。但是你确实想要C++11移动语义带来的性能优势，毕竟这里的类型是可以移动的(move-enabled types)。因此你需要确保**Widget**有移动操作，可以手写代码也可以让编译器自动生成，当然前提是自动生成的条件能满足（参见Item 17）。

当新元素添加到`std::vector`，`std::vector`可能没地方放它，换句话说，`std::vector`的大小(size)等于它的容量(capacity)。这时候，`std::vector`会分配一片的新的大块内存用于存放，然后将元素从已经存在的内存移动到新内存。在C++98中，移动是通过复制老内存区的每一个元素到新内存区完成的，然后老内存区的每个元素发生析构。
这种方法使得`push_back`可以提供很强的异常安全保证：如果在复制元素期间抛出异常，`std::vector`状态保持不变，因为老内存元素析构必须建立在它们已经成功复制到新内存的前提下。

在C++11中，一个很自然的优化就是将上述复制操作替换为移动操作。但是很不幸运，这回破坏`push_back`的异常安全。如果**n**个元素已经从老内存移动到了新内存区，但异常在移动第**n+1**个元素时抛出，那么`push_back`操作就不能完成。但是原始的`std::vector`已经被修改：有**n**个元素已经移动走了。恢复`std::vector`至原始状态也不太可能，因为从新内存移动到老内存本身又可能引发异常。

这是个很严重的问题，因为老代码可能依赖于`push_back`提供的强烈的异常安全保证。因此，C++11版本的实现不能简单的将`push_back`里面的复制操作替换为移动操作，除非知晓移动操作绝不抛异常，这时复制替换为移动就是安全的，唯一的副作用就是性能得到提升。

`std::vector::push_back`受益于"如果可以就移动，如果必要则复制"策略，并且它不是标准库中唯一采取该策略的函数。C++98中还有一些函数如`std::vector::reverse`,`std:;deque::insert`等也受益于这种强异常保证。对于这个函数只有在知晓移动不抛异常的情况下用C++11的move替换C++98的copy才是安全的。但是如何知道一个函数中的移动操作是否产生异常？答案很明显：它检查是否声明**noexcept**。

swap函数是**noexcept**的绝佳用地。**swap**是STL算法实现的一个关键组件，它也常用于拷贝运算符重载中。它的广泛使用意味着对其施加不抛异常的优化是非常有价值的。有趣的是，标准库的**swap**是否**noexcept**有时依赖于用户定义的**swap**是否**noexcept**。比如，数组和`std::pair`的**swap**声明如下：

```
template <class T, size_t N>
void swap(T (&a)[N], // see
          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // below

template <class T1, class T2>
struct pair {
    …
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
    noexcept(swap(second, p.second)));
    …
};
```

这些函数视情况**noexcept**：它们是否**noexcept**依赖于`noexcept`声明中的表达式是否**noexcept**。假设有两个**Widget**数组，不抛异常的交换数组前提是数组中的元素交换不抛异常。对于**Widget**的交换是否**noexcept**决定了对于`Widget`数组的交换是否**noexcept**，反之亦然。类似的，交换两个存放**Widget**的`std::pair`是否**noexcept**依赖于**Widget**的交换是否**noexcept**。事实上交换高层次数据结构是否**noexcept**取决于它的构成部分的那些低层次数据结构是否异常，这激励你只要可以就提供**noexcept swap**函数（译注：因为如果你的函数不提供**noexcept**保证，其它依赖你的高层次**swap**就不能保证**noexcept**）。

现在，我希望你能为**noexcept**提供的优化机会感到高兴，同时我还得让你缓一缓别太高兴了。优化很重要，但是正确性更重要。我在这个条款的开头提到**noexcept**是函数接口的一部分，所以仅当你保证一个函数实现在长时间内不会抛出异常时才声明**noexcept**。如果你声明一个函数为**noexcept**，但随即又后悔了，你没有选择。你只能从函数声明中移除**noexcept**（即改变它的接口），这理所当然会影响客户端代码。你可以改变实现使得这个异常可以避免，再保留原版本（不正确的）异常说明。如果你这

么做，程序将会在异常离开这个函数时终止。或者你可以重新设计既有实现，改变实现后再考虑你希望它是什么样子。这些选择都不尽人意。

这个问题的本质是实际上大多数函数都是异常中立（**exception neutral**）的。这些函数自己不抛异常，但是它们内部的调用可能抛出。此时，异常中立函数允许那些抛出异常的函数在调用链上更进一步直到遇到异常处理程序，而不是就地终止。异常中立函数决不应该声明为**noexcept**，因为它们可能抛出那种"让它们过吧"的异常（译注：也就是说在当前这个函数内不处理异常，但是又不立即终止程序，而是让调用这个函数的函数处理）异常。因此大多数函数都不应该被指定为**noexcept**。

然而，一些函数很自然的不应该抛异常，更进一步值得注意的是移动操作和**swap**——使其不抛异常有重大意义，只要可能就应该将它们声明为**noexcept**。老实说，当你确保函数决不抛异常的时候，一定要将它们声明为**noexcept**。

请注意我说的那些很自然不应该抛异常的函数实现。为了**noexcept**而扭曲函数实现达成目的是本末倒置。是把马放到马车前，是一叶障目不见泰山。是...选择你喜欢的比喻吧。如果一个简单的函数实现可能引发异常（即调用它可能抛出异常），而你为了讨好调用者隐藏了这个（即捕获所有异常，然后替换为状态码或者特殊返回值），这不仅会使你的函数实现变得复杂，还会让所有调用点的代码变得复杂。调用者可能不得不检查状态码或特殊返回值。而这些**复杂**的运行时开销（额外的分支，大的函数放入指令缓存）可以超出**noexcept**带来的性能提升，再加上你会悲哀的发现这些代码又难读又难维护。那是糟糕的软件工程化。

对于一些函数，使其成为**noexcept**是很重要的，它们应当默认如是。在C++98构造函数和析构函数抛出异常是糟糕的代码设计——不管是用户定义的还是编译器生成的构造析构都是**noexcept**。因此它们不需要声明**noexcept**。（这么做也不会有问题，只是不合常规）。析构函数非隐式**noexcept**的情况仅当类的数据成员明确声明它的析构函数可能抛出异常（即，声明`noexcept(false)`）。这种析构函数不常见，标准库里面没有。如果一个对象的析构函数可能被标准库使用，析构函数又可能抛异常，那么程序的行为是未定义的。

值得注意的是一些库接口设计者会区分有宽泛契约(**wild contracts**)和严格契约(**narrow contracts**)的函数。有宽泛契约的函数没有前置条件。这种函数不管程序状态如何都能调用，它对调用者传来的实参不设约束。宽泛契约的函数决不表现出未定义行为。

反之，没有宽泛契约的函数就有严格契约。对于这些函数，如果违反前置条件，结果将会是未定义的。

如果你写了一个有宽泛契约的函数并且你知道它不会抛异常，那么遵循这个条款给它声明一个**noexcept** 是很容易的。
对于严格契约的函数，情况就有点微妙了。举个例子，假如你在写一个参数为**std::string**的函数**f**，并且这个函
数f很自然的决不引发异常。这就在建议我们**f**应该被声明为**noexcept** 。

现在假如**f**有一个前置条件：类型为**std::string**的参数的长度不能超过32个字符。如果现在调用f并传给它一个
大于32字符的参数，函数行为将是未定义的，因为违反了 *（口头/文档）定义的* 前置条件，导致了未定义行为。**f**没有
义务去检查前置条件，它假设这些前置条件都是满足的。（调用者有责任确保参数字符不超过32字符等这些假设有效。）。
即使有前置条件，将**f**声明为**noexcept**似乎也是合适的：

```
void f(const std::string& s) noexcept;  // 前置条件:
                                        // s.length() <= 32
```

f的实现者决定在函数里面检查前置条件冲突。虽然检查是没有必要的，但是也没禁止这么做。另外在系统测试时，检查
前置条件可能就是有用的了。debug一个抛出的异常一般都比跟踪未定义行为起因更容易。那么怎么报告前置条件冲突使得

测试工具或客户端错误处理程序能检测到它呢？简单直接的做法是抛出`"precondition was violated"`异常，但是如果
f声明了**noexcept**，这就行不通了；抛出一个异常会导致程序终止。因为这个原因，区分严格/宽泛契约库设计者一般
会将**noexcept**留给宽泛契约函数。

作为结束语，让我详细说明一下之前的观察，即编译器不会为函数实现和异常规范提供一致性保障。考虑下面的代码，它是
完全正确的：

```
void setup(); // 函数定义另在一处
void cleanup();
void doWork() noexcept
{
 setup();   // 前置设置
 …          // 真实工作
 cleanup(); // 执行后置清理
}
```

这里，**doWork**声明为**noexcept**，即使它调用了非**noexcept**函数`setup`和`cleanup`。看起来有点矛盾，
其实可以猜想`setup`和`cleanup`在文档上写明了它们决不抛出异常，即使它们没有写上**noexcept**。至于为什么
明明不抛异常却不写**noexcept**也是有合理原因的。比如，它们可能是用C写的库函数的一部分。（即使一些函数从
C标准库移动到了**std**命名空间，也可能缺少异常规范，**std::strlen**就是一个例子，它没有声明**noexcept**）。
或者它们可能是C++98库的一部分，它们不使用C++98异常规范的函数的一部分，到了C++11还没有修订。

因为有很多合理原因解释为什么**noexcept**依赖于缺少**noexcept**保证的函数，所以C++允许这些代码，编译器
一般也不会给出warnigns。

记住：

- **noexcept**是函数接口的一部分，这意味着调用者会依赖它、
- **noexcept**函数较之于非**noexcept**函数更容易优化
- **noexcept**对于移动语义,**swap**，内存释放函数和析构函数非常有用
- 大多数函数是异常中立的(译注：可能抛也可能不抛异常）而不是**noexcept**

Item 15:尽可能的使用constexpr

条款 15:尽可能的使用constexpr

如果要给C++11颁一个“最令人困惑新词”奖，**constexpr**十有八九会折桂。当用于对象上面，它本质上就是**const**的加强形式，但是当它用于函数上，意思就大不相同了。有必要消除困惑，因为你绝对会用它的，特别是当你发现**constexpr** “正合吾意”的时候。

从概念上来说，**constexpr**表明一个值不仅仅是常量，还是编译期可知的。这个表述并不全面，因为当**constexpr**被用于函数的时候，事情就有一些细微差别了。
为了避免我毁了结局带来的surprise，我现在只想说，你不能假设**constexpr**函数是**const**，也不能保证它们的（译注：返回）值是在编译期可知的。最有意思的是，这些是特性。关于**constexpr**函数返回的结果不需要是**const**，也不需要编译期可知这一点是良好的行为。

不过我们还是先从**constexpr**对象开始说起。这些对象，实际上，和**const**一样，它们是编译期可知的。（技术上来讲，它们的值在翻译期（translation）决议，所谓翻译不仅仅包含是编译（compilation）也包含链接（linking），除非你准备写C++的编译器和链接器，否则这些对你不会造成影响，所以你编程时无需担心，把这些**constexpr**对象值看做编译期决议也无妨的。）

编译期可知的值“享有特权”，它们可能被存放到只读存储空间中。对于那些嵌入式系统的开发者，这个特性是相当重要的。更广泛的应用是“其值编译期可知”的常量整数会出现在需要“整型常量表达式（**integral constant expression**）的`context`中，这类`context`包括数组大小，整数模板参数（包括`std::array`对象的长度），枚举量，对齐修饰符（译注：`alignas(val)`），等等。如果你想在这些`context`中使用变量，你一定会希望将它们声明为**constexpr**，因为编译器会确保它们是编译期可知的：

```
int sz;                                // 非constexpr变量
…
constexpr auto arraySize1 = sz;        // 错误！ sz的值在
                                       // 编译期不可知
std::array<int, sz> data1;             // 错误!一样的问题
constexpr auto arraySize2 = 10;        // 没问题，10是编译
                                       // 期可知常量
std::array<int, arraySize2> data2;     // 没问题，arraySize2是constexpr
```

注意**const**不提供**constexpr**所能保证之事，因为**const**对象不需要在编译期初始化它的值。

```
int sz;                                // 和之前一样
const auto arraySize = sz;             // 没问题，arraySize是sz的常量复制
std::array<int, arraySize> data;       // 错误，arraySize值在编译期不可知
```

简而言之，所有**constexpr**对象都是**const**，但不是所有**const**对象都是**constexpr**。如果你想编译器保证一个变量有一个可以放到那些需要编译期常量的上下文的值，你需要的工具是**constexpr**而不是**const**。

如果使用场景涉及函数，那 **constexpr**就更有趣了。如果实参是编译期常量，它们将产出编译期值；如果是运行时值，它们就将产出运行时值。这听起来就像你不知道它们要做什么一样，那么想是错误的，请这么看：

- **constexpr**函数可以用于需求编译期常量的上下文。如果你传给**constexpr**函数的实参在编译期可知，那么结果将在编译期计算。如果实参的值在编译期不知道，你的代码就会被拒绝。
- 当一个**constexpr**函数被一个或者多个编译期不可知值调用时，它就像普通函数一样，运行时计算它的结果。这意味着你不需要两个函数，一个用于编译期计算，一个用于运行时计算。**constexpr**

全做了。

假设我们需要一个数据结构来存储一个实验的结果，而这个实验可能以各种方式进行。实验期间风扇转速，温度等等都可能导致亮度值改变，亮度值可以是高，低，或者无。如果有n个实验相关的环境条件。它们每一个都有三个状态，最终可以得到的组合$$3^n$$个。储存所有实验结果的所有组合需要这个数据结构足够大。假设每个结果都是**int**并且**n**是编译期已知的（或者可以被计算出的），一个`std::array`是一个合理的选择。我们需要一个方法在编译期计算`3^n`。C++标准库提供了`std::pow`，它的数学意义正是我们所需要的，但是，对我们来说，这里还有两个问题。第一，`std::pow`是为浮点类型设计的 我们需要整型结果。第二，`std::pow`不是**constexpr**（即，使用编译期可知值调用得到的可能不是编译期可知的结果），所以我们不能用它作为`std::array`的大小。

幸运的是，我们可以应需写个`pow`。我将展示怎么快速完成它，不过现在让我们先看看它应该怎么被声明和使用：

```
constexpr                                   // pow是constexpr函数
int pow(int base, int exp) noexcept         // 绝不抛异常
{
 …                                          // 实现在这里
}
constexpr auto numConds = 5;                //条件个数
std::array<int, pow(3, numConds)> results; // 结果有3^numConds个元素
```

回忆下`pow`前面的**constexpr**没有告诉我们`pow`返回一个**const**值，它只说了如果**base**和**exp**是编译期常量，`pow`返回值可能是编译期常量。如果**base** 和/或 **exp**不是编译期常量，`pow`结果将会在运行时计算。这意味着**pow**不知可以用于像`std::array`的大小这种需要编译期常量的地方，它也可以用于运行时环境：

```
auto base = readFromDB("base");     // 运行时获取三个值
auto exp = readFromDB("exponent");
auto baseToExp = pow(base, exp);    // 运行时调用pow
```

因为**constexpr**函数必须能在编译期值调用的时候返回编译器结果，就必须对它的实现施加一些限制。这些限制在C++11和C++14标准间有所出入。

C++11中，**constexpr**函数的代码不超过一行语句：一个**return**。听起来很受限，但实际上有两个技巧可以扩展**constexpr**函数的表达能力。第一，使用三元运算符"?:"来代替**if-else**语句，第二，使用递归代替循环。因此**pow**可以像这样实现：

```
constexpr int pow(int base, int exp) noexcept
{
 return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

这样没问题，但是很难想象除了使用函数式语言的程序员外会觉得这样硬核的编程方式更好。在C++14中，**constexpr**函数的限制变得非常宽松了，所以下面的函数实现成为了可能；

```
constexpr int pow(int base, int exp) noexcept  // C++14
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;
  return result;
}
```

constexpr函数限制为只能获取和返回字面值类型，这基本上意味着具有那些类型的值能在编译期决定。在C++11中，除了void外的所有内置类型外还包括一些用户定义的字面值，因为构造函数和其他成员函数可以是**constexpr**：

```
class Point {
  public:
    constexpr Point(double xVal = 0, double yVal = 0) noexcept : x(xVal), y(yVal)
    {}
    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }

    void setX(double newX) noexcept { x = newX; }
    void setY(double newY) noexcept { y = newY; }
  private:
    double x, y;
};
```

Point的构造函数被声明为**constexpr**，因为如果传入的参数在编译期可知，**Point**的数据成员也能在编译器可知。因此**Point**就能被初始化为**constexpr**：

```
constexpr Point p1(9.4, 27.7); // 没问题，构造函数会在编译期“运行”
constexpr Point p2(28.8, 5.3); // 也没问题
```

类似的，xValue和yValue的getter函数也能是**constexpr**，因为如果对一个编译期已知的**Point**对象调用getter，数据成员x和y的值也能在编译期知道。这使得我们可以写一个**constexpr**函数里面调用**Point**的getter并初始化**constexpr**的对象：

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,
           (p1.yValue() + p2.yValue()) / 2 };
}
constexpr auto mid = midpoint(p1, p2);
```

这太令人激动了。它意味着mid对象通过调用构造函数，getter和成员函数就能在只读内存中创建！它也意味着你可以在模板或者需要枚举量的表达式里面使用像`mid.xValue()*10`的表达式！它也意味着以前相对严格的某一行代码只能用于编译期，某一行代码只能用于运行时的界限变得模糊，一些运行时的普通计算能并入编译时。越多这样的代码并入，你的程序就越快。（当然，编译会花费更长时间）

在C++11中，有两个限制使得**Point**的成员函数`setX`和`setY`不能声明为**constexpr**。第一，它们修改它们操作的对象的状态，并且在C++11中，**constexpr**成员函数是隐式的const。第二，它们只能有void返回类型，void类型不是C++11中的字面值类型。这两个限制在C++14中放开了，所以C++14中Point的setter也能声明为**constexpr**：

```
class Point {
  public:
    ...
    constexpr void setX(double newX) noexcept { x = newX; }
    constexpr void setY(double newY) noexcept { y = newY; }
    ...
};
```

现在也能写这样的函数：

```
constexpr Point reflection(const Point& p) noexcept
{
    Point result;
    result.setX(-p.xValue());
    result.setY(-p.yValue());
    return result;
}
```

客户端代码可以这样写：

```
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid =          // reflectedMid的值
     reflection(mid);                  // 在编译期可知
```

本章的建议是尽可能的使用**constexpr**，现在我希望大家已经明白缘由：**constexopr**对象和**constexpr**函数可以用于很多非**constexpr**不能使用的场景。使用**constexpr**关键字可以最大化你的对象和函数可以使用的场景。

还有个重要的需要注意的是**constexpr**是对象和函数接口的一部分。加上**constexpr**相当于宣称“我能在C++要求常量表达式的地方使用它”。如果你声明一个对象或者函数是constexpr，客户端程序员就会在那些场景中使用它。如果你后面认为使用constexpr是一个错误并想移除它，你可能造成大量客户端代码不能编译。**尽可能**的使用**constexpr**表示你需要长期坚持对某个对象或者函数施加这种限制。

记住

- **constexpr**对象是**cosnt**，它的值在编译期可知
- 当传递编译期可知的值时，**cosntexpr**函数可以产出编译期可知的结果

Item16:让const成员函数线程安全

条款16: 让const成员函数线程安全

如果我们在数学领域中工作，我们就会发现用一个类表示多项式是很方便的。在这个类中，使用一个函数来计算多项式的根是很有用的。也就是多项式的值为零的时候。这样的一个函数它不会更改多项式。所以，它自然被声明为const函数。

```
class Polynomial {
public:
    using RootsType =           // 数据结构保存多项式为零的值
      std::vector<double>;      //  (“using” 的信息查看条款9)

    RootsType roots() const;

};
```

计算多项式的根是很复杂的，因此如果不需要的话，我们就不做。如果必须做，我们肯定不会只做一次。所以，如果必须计算它们，就缓存多项式的根，然后实现`roots`来返回缓存的值。下面是最基本的实现:

```
class Polynomial {
public:
    using RootsType = std::vector<double>;

    RootsType roots() const
    {
        if (!rootsAreVaild) {                // 如果缓存不可用
                                             // 计算根
            rootsAreVaild = true;            // 用`rootVals`存储它们
        }

        return rootVals;
    }

private:
    mutable bool rootsAreVaild{ false };    // initializers 的更多信息
    mutable RootsType rootVals{};           // 请查看条款7
};
```

从概念上讲，`roots`并不改变它所操作的多项式对象。但是作为缓存的一部分，它也许会改变`rootVals`和`rootsAreVaild`的值。这就是`mutable`的经典使用样例，这也是为什么它是数据成员声明的一部分。

假设现在有两个线程同时调用`Polynomial`对象的`roots`方法:

```
Polynomial p;


/*------ Thread 1 ------*/      /*-------- Thread 2 --------*/
auto rootsOfp = p.roots();      auto valsGivingZero = p.roots();
```

这些用户代码是非常合理的。`roots`是const 成员函数，那就表示着它是一个读操作。在没有同步的情况下，让多个线程执行读操作是安全的。它最起码应该做到这点。在本例中却没有做到线程安全。因为在`roots`中，这些线程中的一个或两个可能尝试修改成员变量`rootsAreVaild`和`rootVals`。这就意味着在没有同步的情况下，这些代码会有不同的线程读写相同的内存，这就是`data race`的定义。这段代码的行为是未定义的。

问题就是`roots`被声明为const，但不是线程安全的。const声明在c++11和c++98 中都是正确的（检索多项式的根并不会更改多项式的值），因此需要纠正的是线程安全的缺乏。

解决这个问题最普遍简单的方法就是-------使用互斥锁:

```
class Polynomial {
public:
    using RootsType = std::vector<double>;

    RootsType roots() const
    {
        std::lock_guard<std::mutex> g(m);         // lock mutex

        if (!rootsAreVaild) {                     // 如果缓存无效
                                                  // 计算/存储roots
            rootsAreVaild = true;
        }

        return rootsVals;
    }                                             // unlock mutex

private:
    mutable std::mutex m;
    mutable bool rootsAreVaild { false };
    mutable RootsType rootsVals {};
};
```

`std::mutex m`被声明为`mutable`，因为锁定和解锁它的都是non-const函数。在`roots`（const成员函数）中，`m`将被视为const对象。
值得注意的是，因为`std::mutex`是一种`move-only`的类型（一种可以移动但不能复制的类型），所以将`m`添加进多项式中的副作用是使它失去了被复制的能力。不过，它仍然可以移动。

在某些情况下，互斥量是过度的（？）。例如，你所做的只是计算成员函数被调用了多少次。使用`std::atomic`修饰的counter（保证其他线程视这个操作为不可分割的发生，参见item40）。（然而它是否轻量取决于你使用的硬件和标准库中互斥量的实现。）以下是如何使用`std::atomic`来统计调用次数。

```
class Point {                                     // 2D point
public:
    // noexcept的使用参考Item 14
    double distanceFromOrigin() const noexcept
    {
        ++callCount;                              // 原子的递增

        return std::sqrt((x * x) + (y * y));
    }

private:
    mutable std::atomic<unsigned> callCount{ 0 };
    double x, y;
```

```
};
```

与`std::mutex`一样，`std::atomic`是`move-only`类型，所以在`Point`中调用`Count`的意思就是`Point`也是`move-only`的。

因为对`std::atomic`变量的操作通常比互斥量的获取和释放的消耗更小，所以你可能更倾向与依赖`std::atomic`。例如，在一个类中，缓存一个开销昂贵的`int`，你就会尝试使用一对`std::atomic`变量而不是互斥锁。

```
class Widget {
public:

    int magicValue() const
    {
        if (cacheVaild) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;              // 第一步
            cacheVaild = true;                      // 第二步
            return cachedVaild;
        }
    }

private:
    mutable std::atomic<bool> cacheVaild{ false };
    mutable std::atomic<int> cachedValue;
};
```

这是可行的，但有时运行会比它做到更加困难。考虑：

- 一个线程调用`Widget::magicValue`，将`cacheValid`视为`false`，执行这两个昂贵的计算，并将它们的和分配给`cachedValue`。
- 此时，第二个线程调用`Widget::magicValue`，也将`cacheValid`视为`false`，因此执行刚才完成的第一个线程相同的计算。（这里的“第二个线程”实际上可能是其他几个线程。）

这种行为与使用缓存的目的背道而驰。将`cachedValue`和`CacheValid`的顺序交换可以解决这个问题，但结果会更糟：

```
class Widget {
public:

    int magicValue() const
    {
        if (cacheVaild) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cacheVaild = true;                      // 第一步
            return cachedValue = val1 + val2;       // 第二步
        }
    }

}
```

假设`cacheVaild`是false，那么：

- 一个线程调用`Widget::magicValue`，在`cacheVaild`被设置成true时执行到它。
- 在这时，第二个线程调用`Widget::magicValue`随后检查缓存值。看到它是true，就返回`cacheValue`，即使第一个线程还没有给它赋值。因此返回的值是不正确的。

这里有一个坑。对于需要同步的是单个的变量或者内存位置，使用`std::atomic`就足够了。
不过，一旦你需要对两个以上的变量或内存位置作为一个单元来操作的话，就应该使用互斥锁。对于`Widget::magicValue`是这样的。

```
class Widget {
public:

    int magicValue() const
    {
        std::lock_guard<std::mutex> guard(m);   // lock m
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;
            cacheValid = true;
            return cachedValue;
        }
    }                                           // unlock m

private:
    mutable std::mutex m;
    mutable int cachedValue;                    // no longer atomic
    mutable bool cacheValid{ false };           // no longer atomic
};
```

现在，这个条款是基于，多个线程可以同时在一个对象上执行一个const成员函数这个假设的。如果你不是在这种情况下编写一个const成员函数。也就是你可以保证在对象上永远不会有多个线程执行该成员函数。再换句话说，该函数的线程安全是无关紧要的。比如，为单线程使用而设计类的成员函数的线程安全是不重要的。在这种情况下你可以避免，因使用`mutex`和`std::atomics`所消耗的资源，以及包含它们的类只能使用移动语义带来的副作用。然而，这种单线程的场景越来越少见，而且很可能会越来越少。可以肯定的是，const成员函数应支持并发执行，这就是为什么你应该确保const成员函数是线程安全的。

> 应该注意的事情
>
> - 确保const成员函数线程安全，除非你确定它们永远不会在临界区（concurrent context）中使用。
> - `std::atomic`可能比互斥锁提供更好的性能，但是它只适合操作单个变量或内存位置。

Ite 17:理解特殊成员函数的生成

条款 17:理解特殊成员函数函数的生成

在C++术语中，特殊成员函数是指C++自己生成的函数。C++98有四个：默认构造函数函数，析构函数，拷贝构造函数，拷贝赋值运算符。这些函数仅在需要的时候才生成，比如某个代码使用它们但是它们没有在类中声明。默认构造函数仅在类完全没有构造函数的时候才生成。（防止编译器为某个类生成构造函数，但是你希望那个构造函数有参数）生成的特殊成员函数是隐式public且inline，除非该类是继承自某个具有虚函数的类，否则生成的析构函数是非虚的。

但是你早就知道这些了。好吧好吧，都说古老的历史：美索不达米亚，商朝，FORTRAN,C++98。但是时代改变了，C++生成特殊成员的规则也改变了。要留意这些新规则，因为用C++高效编程方面很少有像它们一样重要的东西需要知道。

C++11特殊成员函数俱乐部迎来了两位新会员：移动构造函数和移动赋值运算符。它们的签名是：

```
class Widget {
public:
    ...
    Widget(Widget&& rhs);
    Widget& operator=(Widget&& rhs);
    ...
};
```

掌控它们生成和行为的规则类似于拷贝系列。移动操作仅在需要的时候生成，如果生成了，就会对非static数据执行逐成员的移动。那意味着移动构造函数根据`rhs`参数里面对应的成员移动构造出新部分，移动赋值运算符根据参数里面对应的非static成员移动赋值。移动构造函数也移动构造基类部分（如果有的话），移动赋值运算符也是移动赋值基类部分。

现在，当我对一个数据成员或者基类使用移动构造或者移动赋值时，没有任何保证移动一定会真的发生。逐成员移动，实际上，更像是逐成员移动**请求**，因为对不可移动类型使用移动操作实际上执行的是拷贝操作。逐成员移动的核心是对对象使用**std::move**，然后函数决议时会选择执行移动还是拷贝操作。**Item 23**包括了这个操作的细节。本章中，简单记住如果支持移动就会逐成员移动类成员和基类成员，如果不支持移动就执行拷贝操作就好了。

两个拷贝操作是独立的：声明一个不会限制编译器声明另一个。所以如果你声明一个拷贝构造函数，但是没有声明拷贝赋值运算符，如果写的代码用到了拷贝赋值，编译器会帮助你生成拷贝赋值运算符重载。同样的，如果你声明拷贝赋值运算符但是没有拷贝构造，代码用到拷贝构造编译器就会生成它。上述规则在C++98和C++11中都成立。

如果你声明了某个移动函数，编译器就不再生成另一个移动函数。这与复制函数的生成规则不太一样：两个复制函数是独立的，声明一个不会影响另一个的默认生成。这条规则的背后原因是，如果你声明了某个移动函数，就表明这个类型的移动操作不再是“逐一移动成员变量”的语义，即你不需要编译器默认生成的移动函数的语义，因此编译器也不会为你生成另一个移动函数。

再进一步，如果一个类显式声明了拷贝操作，编译器就不会生成移动操作。这种限制的解释是如果声明拷贝操作就暗示着默认逐成员拷贝操作不适用于该类，编译器会明白如果默认拷贝不适用于该类，移动操作也可能是不适用的。

这是另一个方向。声明移动操作使得编译器不会生成拷贝操作。（编译器通过给这些函数加上delete来保证，参见Item11）。比较，如果逐成员移动对该类来说不合适，也没有理由指望逐成员考吧操作是合适的。听起来会破坏C++98的某些代码，因为C++11中拷贝操作可用的条件比C++98更受限，但事实并非如此。C++98的代码没有移动操作，因为C++98中没有移动对象这种概念。只有一种方法能让老代码

使用用户声明的移动操作，那就是使用C++11标准然后添加这些操作， 并在享受这些操作带来的好处同时接受C++11特殊成员函数生成规则的限制。

也许你早已听过*Rule of Three*规则。这个规则告诉我们如果你声明了拷贝构造函数，拷贝赋值运算符，或者析构函数三者之一，你应该也声明其余两个。它来源于长期的观察，即用户接管拷贝操作的需求几乎都是因为该类会做其他资源的管理，这也几乎意味着1）无论哪种资源管理如果能在一个拷贝操作内完成，也应该在另一个拷贝操作内完成2）类析构函数也需要参与资源的管理（通常是释放）。通常意义的资源管理指的是内存（如STL容器会动态管理内存），这也是为什么标准库里面那些管理内存的类都声明了“the big three”：拷贝构造，拷贝赋值和析构。

Rule of Three带来的后果就是只要出现用户定义的析构函数就意味着简单的逐成员拷贝操作不适用于该类。接着，如果一个类声明了析构也意味着拷贝操作可能不应该自定生成，因为它们做的事情可能是错误的。在C++98提出的时候，上述推理没有得倒足够的重视，所以C++98用户声明析构不会左右编译器生成拷贝操作的意愿。C++11中情况仍然如此，但仅仅是因为限制拷贝操作生成的条件会破坏老代码。

Rule of Three规则背后的解释依然有效，再加上对声明拷贝操作阻止移动操作隐式生成的观察，使得C++11不会为那些有用户定义的析构函数的类生成移动操作。所以仅当下面条件成立时才会生成移动操作：

- 类中没有拷贝操作
- 类中没有移动操作
- 类中没有用户定义的析构

有时，类似的规则也会扩展至移动操作上面，因为现在类声明了拷贝操作，C++11不会为它们自动生成其他拷贝操作。这意味着如果你的某个声明了析构或者拷贝的类依赖自动生成的拷贝操作，你应该考虑升级这些类，消除依赖。假设编译器生成的函数行为是正确的（即逐成员拷贝类数据是你期望的行为），你的工作很简单，C++11的`=default`就可以表达你想做的：

```
class Widget {
   public:
    ...
    ~Widget();
    ...
    Widget(const Widget&) = default;
    Widget&
    operator=(const Widget&) = default; // behavior is OK
    ...
};
```

这种方法通常在多态基类中很有用，即根据继承自哪个类来定义接口。多态基类通常有一个虚析构函数，因为如果它们非虚，一些操作（比如对一个基类指针或者引用使用delete或者typeid）会产生未定义或错误结果。除非类继承自一个已经是virtual的析构函数，否则要想析构为虚函数的唯一方法就是加上virtual关键字。通常，默认实现是对的，`=default`是一个不错的方式表达默认实现。然而用户声明的析构函数会抑制编译器生成移动操作，所以如果该类需要具有移动性，就为移动操作加上`=default`。声明移动会抑制拷贝生成，所以如果拷贝性也需要支持，再为拷贝操作加上`=default`：

```
class Base {
public:
  virtual ~Base() = default;
  Base(Base&&) = default;
  Base& operator=(Base&&) = default;
  Base(const Base&) = default;
  Base& operator=(const Base&) = default;
  ...
};
```

实际上，就算编译器乐于为你的类生成拷贝和移动操作，生成的函数也如你所愿，你也应该手动声明它们然后加上`=default`。这看起来比较多余，但是它让你的意图更明确，也能帮助你避免一些微妙的bug。比如，你有一个字符串哈希表，即键为整数id，值为字符串，支持快速查找的数据结构：

```
 class StringTable {
   public:
     StringTable() {}
     ...
   private:
     std::map<int, std::string> values;
};
```

假设这个类没有声明拷贝操作，没有移动操作，也没有析构，如果它们被用到编译器会自动生成。没错，很方便。

后来需要在对象构造和析构中打日志，增加这种功能很简单：

```
class StringTable {
public:
  StringTable()
  { makeLogEntry("Creating StringTable object"); }

  ~StringTable()
  { makeLogEntry("Destroying StringTable object"); }
  ...
 Item 17 | 113
private:
     std::map<int, std::string> values;    // as before
};
```

看起来合情合理，但是声明析构有潜在的副作用：它阻止了移动操作的生成。然而，拷贝操作的生成是不受影响的。因此代码能通过编译，运行，也能通过功能（译注：即打日志的功能）测试。功能测试也包括移动功能，因为即使该类不支持移动操作，对该类的移动请求也能通过编译和运行。这个请求正如之前提到的，会转而由拷贝操作完成。它因为着对**StringTable**对象的移动实际上是对对象的拷贝，即拷贝里面的`std::map<int, std::string>`对象。拷贝`std::map<int, std::string>`对象很可能比移动慢几个数量级。简单的加个析构就引入了极大的性能问题！对拷贝和移动操作显式加个`=default`，问题将不再出现。

受够了我喋喋不休的讲述C++11拷贝移动规则了吧，你可能想知道什么时候我才会把注意力转入到剩下两个特殊成员函数，默认构造和析构。现在就是时候了，但是只有一句话，因为它们几乎没有改变：它们在C++98中是什么样，在C++11中就是什么样。

C++11对于特殊成员函数处理的规则如下：

- 默认构造函数：和C++98规则相同。仅当类不存在用户声明的构造函数时才自动生成。
- 析构函数：基本上和C++98相同；稍微不同的是现在析构默认**noexcept**（参见Item14）。和C++98一样，仅当基类析构为虚函数时该类析构才为虚函数。
- 拷贝构造函数：和C++98运行时行为一样：逐成员拷贝非static数据。仅当类没有用户定义的拷贝构造时才生成。如果类声明了移动操作它就是**delete**。当用户声明了拷贝赋值或者析构，该函数不再自动生成。
- 拷贝赋值运算符：和C++98运行时行为一样：逐成员拷贝赋值非static数据。仅当类没有用户定义的拷贝赋值时才生成。如果类声明了移动操作它就是**delete**。当用户声明了拷贝构造或者析构，该函数不再自动生成。
- 移动构造函数和移动赋值运算符：都对非static数据执行逐成员移动。仅当类没有用户定义的拷贝操作，移动操作或析构时才自动生成。

注意没有成员函数模版阻止编译器生成特殊成员函数的规则。这意味着如果**Widget**是这样：

```
class Widget {
  ...
  template<typename T>
  Widget(const T& rhs);

  template<typename T>
  Widget& operator=(const T& rhs); ...
};
```

编译器仍会生成移动和拷贝操作（假设正常生成它们的条件满足），即使可以模板实例化产出拷贝构造和拷贝赋值运算符的函数签名。（当T为Widget时）。很可能你会决定这是一个不值得承认的边缘情况，但是我提到它是有道理的，Item16将会详细讨论它可能带来的后果。

记住：

- 特殊成员函数是编译器可能自动生成的函数：默认构造，析构，拷贝操作，移动操作。
- 移动操作仅当类没有显式声明移动操作，拷贝操作，析构时才自动生成。
- 拷贝构造仅当类没有显式声明拷贝构造时才自动生成，并且如果用户声明了移动操作，拷贝构造就是delete。拷贝赋值运算符仅当类没有显式声明拷贝赋值运算符时才自动生成，并且如果用户声明了移动操作，拷贝赋值运算符就是delete。当用户声明了析构函数，拷贝操作不再自动生成。

CHAPTER 4 Smart Pointers

诗人和歌曲作家喜欢爱。有时候喜欢计数。很少情况下两者兼有。受伊丽莎白·巴雷特·勃朗宁（Elizabeth Barrett Browning）对爱和数的不同看法的启发（“我怎么爱你？”让我数一数。”）和保罗·西蒙（Paul Simon）（“离开你的爱人必须有50种方法。”），我们可以试着枚举一些为什么原始指针很难被爱的原因：

1. 它的声明不能指示所指到底是单个对象还是数组。
2. 它的声明没有告诉你用完后是否应该销毁它，即指针是否拥有所指之物。
3. 如果你决定你应该销毁对象所指，没人告诉你该用delete还是其他析构机制（比如将指针传给专门的销毁函数）。
4. 如果你发现该用delete。 原因1说了不知道是delete单个对象还是delete数组。如果用错了结果是未定义的。
5. 假设你确定了指针所指，知道销毁机制，也很难确定你在所有执行路径上都执行了销毁操作（包括异常产生后的路径）。少一条路径就会产生资源泄漏，销毁多次还会导致未定义行为。
6. 一般来说没有办法告诉你指针是否变成了悬空指针（dangling pointers），即内存中不再存在指针所指之物。悬空指针会在对象销毁后仍然指向它们。

原始指针是强大的工具，当然，另一方面几十年的经验证明，只要注意力稍有疏忽，这个强大的工具就会攻击它的主人。

智能指针是解决这些问题的一种办法。智能指针包裹原始指针，它们的行为看起来像被包裹的原始指针，但避免了原始指针的很多陷阱。你应该更倾向于智能指针而不是原始指针。几乎原始指针能做的所有事情智能指针都能做，而且出错的机会更少。

在C++11中存在四种智能指针：`std::auto_ptr`，`std::unique_ptr`，`std::shared_ptr`，`std::weak_ptr`。都是被设计用来帮助管理动态对象的生命周期，在适当的时间通过适当的方式来销毁对象，以避免出现资源泄露或者异常行为。

`std::auto_ptr`是C++98的遗留物，它是一次标准化的尝试，后来变成了C++11的`std::unique_ptr`。要正确的模拟原生制作需要移动语义，但是C++98没有这个东西。取而代之，`std::auto_ptr`拉拢拷贝操作来达到自己的移动意图。这导致了令人奇怪的代码（拷贝一个`std::auto_ptr`会将它本身设置为null！）和令人沮丧的使用限制（比如不能将`std::auto_ptr`放入容器）。

`std::unique_ptr`能做`std::auto_ptr`可以做的所有事情以及更多。它能高效完成任务，而且不会扭曲拷贝语义。在所有方面它都比`std::unique_ptr`好。现在`std::auto_ptr`唯一合法的使用场景就是代码使用C++98编译器编译。除非你有上述限制，否则你就该把`std::auto_ptr`替换为`std::unique_ptr`而且绝不回头。

各种智能指针的API有极大的不同。唯一功能性相似的可能就是默认构造函数。因为有很多关于这些API的详细手册，所以我将只关注那些API概览没有提及的内容，比如值得注意的使用场景，运行时性能分析等，掌握这些信息可以更高效的使用智能指针。

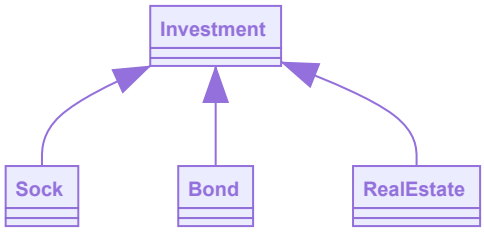
Item 18:对于独占资源使用std::unique_ptr

当你需要一个智能指针时，`std::unique_ptr`通常是最合适的。可以合理假设，默认情况下，`std::unique_ptr`等同于原始指针，而且对于大多数操作（包括取消引用），他们执行的指令完全相同。这意味着你甚至可以在内存和时间都比较紧张的情况下使用它。如果原始指针够小够快，那么`std::unique_ptr`一样可以。

`std::unique_ptr`体现了专有所有权语义。一个`non-null std::unique_ptr`始终有其指向的内容。移动操作将所有权从源指针转移到目的指针，拷贝操作是不允许的，因为如果你能拷贝一个`std::unique_ptr`，你会得到指向相同内容的两个`std::unique_ptr`，每个都认为自己拥有资源，销毁时就会出现重复销毁。因此，`std::unique_ptr`只支持移动操作。当`std::unique_ptr`销毁时，其指向的资源也执行析构函数。而原始指针需要显示调用delete来销毁指针指向的资源。

`std::unique_ptr`的常见用法是作为继承层次结构中对象的工厂函数返回类型。假设我们有一个基类Investment（比如 stocks,bonds,real estate等）的继承结构。

```
class Investment { ... };
class Sock: public Investment {...};
class Bond: public Investment {...};
class RealEstate: public Investment {...};
```



这种继承关系的工厂函数在堆上分配一个对象然后返回指针，调用方在不需要的时候，销毁对象。这使用场景完美匹配`std::unique_ptr`，因为调用者对工厂返回的资源负责（即对该资源的专有所有权），并且`std::unique_ptr`会自动销毁指向的内容。可以这样声明：

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&&... params);
```

调用者应该在单独的作用域中使用返回的`std::unique_ptr`智能指针：

```
{
  ...
  auto pInvestment = makeInvestment(arguments);
  ...
} //destroy *pInvestment
```

但是也可以在所有权转移的场景中使用它，比如将工厂返回的`std::unique_ptr`移入容器中，然后将容器元素移入对象的数据成员中，然后对象随即被销毁。发生这种情况时，并且销毁该对象将导致销毁从工厂返回的资源，对象`std::unique_ptr`的数据成员也被销毁。如果所有权链由于异常或者其他非典型控制流出现中断（比如提前return函数或者循环中的break），则拥有托管资源的`std::unique_ptr`将保证指向内容的析构函数被调用，销毁对应资源。

默认情况下，销毁将通过delete进行，但是在构造过程中，可以自定义`std::unique_ptr`指向对象的析构函数：任意函数（或者函数对象，包括lambda）。如果通过`makeInvestment`创建的对象不能直接被删除，应该首先写一条日志，可以实现如下：

```
auto delInvmt = [](Investment* pInvestment)
{
```

```
  makeLogEntry(pInvestment);
  delete pInvestment;
};
template<typename... Ts>
std::unique_ptr<Investment, decltype(delInvmt)>
makeInvestment(Ts&& params)
{
  std::unique_ptr<Investment, decltype(delInvmt)> pInv(nullptr, delInvmt);
  if (/*a Stock object should be created*/)
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* a Bond object should be created */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* a RealEstate object should be created */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;
}
```

稍后，我将解释其工作原理，但首先请考虑如果你是调用者，情况如何。假设你存储`makeInvestment`调用结果在auto变量中，那么你将在愉快中忽略在删除过程中需要特殊处理的事实，当然，你确实幸福，因为使用了`unique_ptr`意味着你不需要考虑在资源释放时的路径，以及确保只释放一次，`std::unique_ptr`自动解决了这些问题。从使用者角度，`makeInvestment`接口很棒。

这个实现确实相当棒，如果你理解了：

- `delInvmt`是自定义的从`makeInvestment`返回的析构函数。所有的自定义的析构行为接受要销毁对象的原始指针，然后执行销毁操作。如上例子。使用lambda创建`delInvmt`是方便的，而且，正如稍后看到的，比编写常规的函数更有效
- 当使用自定义删除器时，必须将其作为第二个参数传给`std::unique_ptr`。对于decltype，更多信息查看Item3
- `makeInvestment`的基本策略是创建一个空的`std::unique_ptr`，然后指向一个合适类型的对象，然后返回。为了与pInv关联自定义删除器，作为构造函数的第二个参数
- 尝试将原始指针（比如new创建）赋值给`std::unique_ptr`通不过编译，因为不存在从原始指针到智能指针的隐式转换。这种隐式转换会出问题，所以禁止。这就是为什么通过`reset`来传递new指针的原因
- 使用new时，要使用`std::forward`作为参数来完美转发给`makeInvestment`（查看Item 25）。这使调用者提供的所有信息可用于正在创建的对象的构造函数
- 自定义删除器的参数类型是`Investment*`，尽管真实的对象类型是在`makeInvestment`内部创建的，它最终通过在lambda表达式中，作为`Investment*`对象被删除。这意味着我们通过基类指针删除派生类实例，为此，基类必须是虚函数析构：

```
class Investment {
  public:
    ...
    virtual ~Investment();
    ...
};
```

在C++14中，函数的返回类型推导存在（参阅Item 3），意味着`makeInvestment`可以更简单，封装的方式实现：

```
template<typename... Ts>
makeInvestment(Ts&& params)
{
  auto delInvmt = [](Investment* pInvestment)
    {
    makeLogEntry(pInvestment);
    delete pInvestment;
    };
  std::unique_ptr<Investment, decltype(delInvmt)> pInv(nullptr, delInvmt);
  if (/*a Stock object should be created*/)
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* a Bond object should be created */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* a RealEstate object should be created */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;
}
```

我之前说过，当使用默认删除器时，你可以合理假设`std::unique_ptr`和原始指针大小相同。当自定义删除器时，情况可能不再如此。删除器是个函数指针，通常会使`std::unique_ptr`的字节从一个增加到两个。对于删除器的函数对象来说，大小取决于函数对象中存储的状态多少，无状态函数对象（比如没有捕获的lambda表达式）对大小没有影响，这意味当自定义删除器可以被lambda实现时，尽量使用lambda

```
auto delInvmt = [](Investment* pInvestment)
{
  makeLogEntry(pInvestment);
  delete pInvestment;
};
template<typename... Ts>
std::unique_ptr<Investment, decltype(delInvmt)>
makeInvestment(Ts&& params); //返回Investment*的大小

void delInvmt2(Investment* pInvestment)
{
  makeLogEntry(pInvestment);
  delete pInvestment;
}
template<typename... Ts>
std::unique_ptr<Investment, void(*)(Investment*)>
makeInvestment(Ts&&... params); //返回Investment*的指针加至少一个函数指针的大小
```

具有很多状态的自定义删除器会产生大尺寸`std::unique_ptr`对象。如果你发现自定义删除器使得你的`std::unique_ptr`变得过大，你需要审视修改你的设计。

工厂函数不是`std::unique_ptr`的唯一常见用法。作为实现**Pimpl Idiom**的一种机制，它更为流行。代码并不复杂，但是在某些情况下并不直观，所以这安排在Item22的专门主题中。

`std::unique_ptr`有两种形式，一种用于单个对象（`std::unique_ptr<T>`），一种用于数组（`std::unique_ptr<T[]>`）。结果就是，指向哪种形式没有歧义。`std::unique_ptr`的API设计会自动匹配你的用法，比如[]操作符就是数组对象，*和->就是单个对象专有。

数组的`std::unique_ptr`的存在应该不被使用，因为`std::array`，`std::vector`，`std::string`这些更好用的数据容器应该取代原始数组。原始数组的使用唯一情况是你使用类似C的API返回一个指向堆数组的原始指针。

`std::unique_ptr`是C++11中表示专有所有权的方法，但是其最吸引人的功能之一是它可以轻松高效的转换为`std::shared_ptr`：

```
std::shared_ptr<Investment> sp = makeInvestment(arguments);
```

这就是为什么`std::unique_ptr`非常适合用作工厂函数返回类型的关键部分。 工厂函数无法知道调用者是否要对它们返回的对象使用专有所有权语义，或者共享所有权（即`std::shared_ptr`）是否更合适。 通过返回`std::unique_ptr`，工厂为调用者提供了最有效的智能指针，但它们并不妨碍调用者用其更灵活的兄弟替换它。（有关`std::shared_ptr`的信息，请转到Item 19。

小结

- `std::unique_ptr`是轻量级、快速的、只能move的管理专有所有权语义资源的智能指针
- 默认情况，资源销毁通过delete，但是支持自定义delete函数。有状态的删除器和函数指针会增加`std::unique_ptr`的大小
- 将`std::unique_ptr`转化为`std::shared_ptr`是简单的

Item 19:对于共享资源使用std::shared_ptr

条款十九:对于共享资源使用std::shared_ptr

程序员使用带垃圾回收的语言指着C++笑看他们如何防止资源泄露。“真是原始啊！”他们嘲笑着说。“你们没有从1960年的Lisp那里得到启发吗，机器应该自己管理资源的生命周期而不应该依赖人类。”C++程序员翻白眼。“你得到的启发就是只有内存算资源，其他资源释放都是非确定性的你知道吗？我们更喜欢通用，可预料的销毁，谢谢你。”但我们的虚张声势可能底气不足。因为垃圾回收真的很方便，而且手动管理生命周期真的就像是使用石头小刀和兽皮制作RAM电路。为什么我们不能同时有两个完美的世界：一个自动工作的世界（垃圾回收），一个销毁可预测的世界（析构）？

C++11中的`std::shared_ptr`将两者组合了起来。一个通过`std::shared_ptr`访问的对象其生命周期由指向它的指针们共享所有权（shared ownership）。没有特定的`std::shared_ptr`拥有该对象。相反，所有指向它的`std::shared_ptr`都能相互合作确保在它不再使用的那个点进行析构。当最后一个`std::shared_ptr`到达那个点，`std::shared_ptr`会销毁它所指向的对象。就垃圾回收来说，客户端不需要关心指向对象的生命周期，而对象的析构是确定性的。

`std::shared_ptr`通过引用计数来确保它是否是最后一个指向某种资源的指针，引用计数关联资源并跟踪有多少`std::shared_ptr`指向该资源。`std::shared_ptr`构造函数递增引用计数值（注意是通常——原因参见下面），析构函数递减值，拷贝赋值运算符可能递增也可能递减值。（如果sp1和sp2是`std::shared_ptr`并且指向不同对象，赋值运算符`sp1=sp2`会使sp1指向sp2指向的对象。直接效果就是sp1引用计数减一，sp2引用计数加一。）如果`std::shared_ptr`发现引用计数值为零，没有其他`std::shared_ptr`指向该资源，它就会销毁资源。

引用计数暗示着性能问题：

- **`std::shared_ptr`大小是原始指针的两倍**，因为它内部包含一个指向资源的原始指针，还包含一个资源的引用计数值。
- **引用计数必须动态分配**。 理论上，引用计数与所指对象关联起来，但是被指向的对象不知道这件事情（译注：不知道有指向自己的指针）。因此它们没有办法存放一个引用计数值。Item21会解释使用`std::make_shared`创建`std::shared_ptr`可以避免引用计数的动态分配，但是还存在一些`std::make_shared`不能使用的场景，这时候引用计数就会动态分配。
- **递增递减引用计数必须是原子性的**，因为多个reader、writer可能在不同的线程。比如，指向某种资源的`std::shared_ptr`可能在一个线程执行析构，在另一个不同的线程，`std::shared_ptr`指向相同的对象，但是执行的确是拷贝操作。原子操作通常比非原子操作要慢，所以即使是引用计数，你也应该假定读写它们是存在开销的。

我写道`std::shared_ptr`构造函数只是“通常”递增指向对象的引用计数会不会让你有点好奇？创建一个指向对象的`std::shared_ptr`至少产生了一个指向对象的智能指针，为什么我没说**总是**增加引用计数值？

原因是移动构造函数的存在。从另一个`std::shared_ptr`移动构造新`std::shared_ptr`会将原来的`std::shared_ptr`设置为null，那意味着老的`std::shared_ptr`不再指向资源，同时新的`std::shared_ptr`指向资源。这样的结果就是不需要修改引用计数值。因此移动`std::shared_ptr`会比拷贝它要快：拷贝要求递增引用计数值，移动不需要。移动赋值运算符同理，所以移动赋值运算符也比拷贝赋值运算符快。

类似`std::unique_ptr`（参加Item18），`std::shared_ptr`使用**delete**作为资源的默认销毁器，但是它也支持自定义的销毁器。这种支持有别于`std::unique_ptr`。对于`std::unique_ptr`来说，销毁器类型是智能指针类型的一部分。对于`std::shared_ptr`则不是：

```
auto loggingDel = [](Widget *pw)     //自定义销毁器
                  {                  // (和Item 18一样)
                      makeLogEntry(pw);
                      delete pw;
                  };

std::unique_ptr<                     // 销毁器类型是
    Widget, decltype(loggingDel)     // ptr类型的一部分
    > upw(new Widget, loggingDel);
std::shared_ptr<Widget>              // 销毁器类型不是
spw(new Widget, loggingDel);         // ptr类型的一部分
```

`std::shared_ptr`的设计更为灵活。考虑有两个`std::shared_ptr`，每个自带不同的销毁器（比如通过lambda表达式自定义销毁器）：

```
auto customDeleter1 = [](Widget *pw) { … };
auto customDeleter2 = [](Widget *pw) { … };
std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

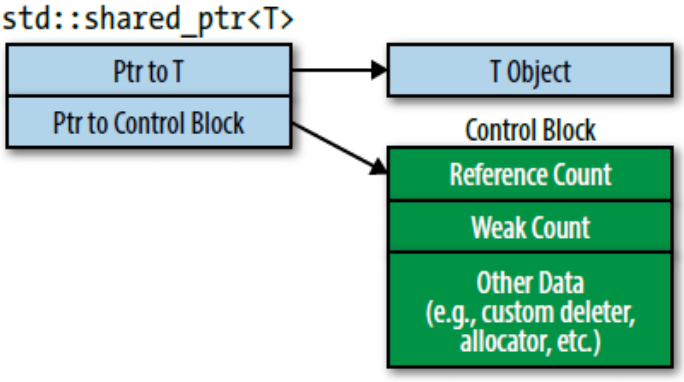
因为**pw1**和**pw2**有相同的类型，所以它们都可以放到存放那个类型的对象的容器中：

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

它们也能相互赋值，也可以传入形参为`std::shared_ptr<Widget>`的函数。但是`std::unique_ptr`就不行，因为`std::unique_ptr`把销毁器视作类型的一部分。

另一个不同于`std::unique_ptr`的地方是，指定自定义销毁器不会改变`std::shared_ptr`对象的大小。不管销毁器是什么，一个`std::shared_ptr`对象都是两个指针大小。这是个好消息，但是它应该让你隐隐约约不安。自定义销毁器可以是函数对象，函数对象可以包含任意多的数据。它意味着函数对象是任意大的。`std::shared_ptr`怎么能引用一个任意大的销毁器而不使用更多的内存？

它不能。它必须使用更多的内存。然而，那部分内存不是`std::shared_ptr`对象的一部分。那部分在堆上面，只要`std::shared_ptr`自定义了分配器，那部分内存随便在哪都行。我前面提到了`std::shared_ptr`对象包含了所指对象的引用计数。没错，但是有点误导人。因为引用计数是另一个更大的数据结构的一部分，那个数据结构通常叫做**控制块**（control block）。控制块包含除了引用计数值外的一个自定义销毁器的拷贝，当然前提是存在自定义销毁器。如果用户还指定了自定义分配器，控制器也会包含一个分配器的拷贝。控制块可能还包含一些额外的数据，正如Item21提到的，一个次级引用计数weak count，但是目前我们先忽略它。我们可以想象`std::shared_ptr`对象在内存中是这样：



当`std::shared_ptr`对象一创建，对象控制块就建立了。至少我们期望是如此。通常，对于一个创建指向对象的`std::shared_ptr`的函数来说不可能知道是否有其他`std::shared_ptr`早已指向那个对象，所以控制块的创建会遵循下面几条规则：

- **`std::make_shared`总是创建一个控制块**(参见Item21)。它创建一个指向新对象的指针，所以可以肯定`std::make_shared`调用时对象不存在其他控制块。
- **当从独占指针上构造出`std::shared_ptr`时会创建控制块（即`std::unique_ptr`或者`std::auto_ptr`）**。独占指针没有使用控制块，所以指针指向的对象没有关联其他控制块。（作为构造的一部分，`std::shared_ptr`侵占独占指针所指向的对象的独占权，所以`std::unique_ptr`被设置为null）
- **当从原始指针上构造出`std::shared_ptr`时会创建控制块**。如果你想从一个早已存在控制块的对象上创建`std::shared_ptr`，你将假定传递一个`std::shared_ptr`或者`std::weak_ptr`作为构造函数实参，而不是原始指针。用`std::shared_ptr`或者`std::weak_ptr`作为构造函数实参创建`std::shared_ptr`不会创建新控制块，因为它可以依赖传递来的智能指针指向控制块。

这些规则造成的后果就是从原始指针上构造超过一个`std::shared_ptr`就会让你走上未定义行为的快车道，因为指向的对象有多个控制块关联。多个控制块意味着多个引用计数值，多个引用计数值意味着对象将会被销毁多次（每个引用计数一次）。那意味着下面的代码是有问题的，很有问题，问题很大：

```
auto pw = new Widget; // pw是原始指针
…
std::shared_ptr<Widget> spw1(pw, loggingDel); // 为*pw创建控制块
…
std::shared_ptr<Widget> spw2(pw, loggingDel); // 为*pw创建第二个控制块
```

创建原始指针指向动态分配的对象很糟糕，因为它完全背离了这章的建议：对于共享资源使用`std::shared_ptr`而不是原始指针。（如果你忘记了该建议的动机，请翻到115页）。撇开那个不说，创建**pw**那一行代码虽然让人厌恶，但是至少不会造成未定义程序行为。

现在，传给**spw1**的构造函数一个原始指针，它会为指向的对象创建一个控制块（引用计数值在里面）。这种情况下，指向的对象是`*pw`。就其本身而言没什么问题，但是将同样的原始指针传递给**spw2**的构造函数会再次为`*pw`创建一个控制块。因此`*pw`有两个引用计数值，每一个最后都会变成零，然后最终导致`*pw`销毁两次。第二个销毁会产生未定义行为。

`std::shared_ptr`给我们上了两堂课。第一，避免传给`std::shared_ptr`构造函数原始指针。通常替代方案是使用`std::make_shared`(参见Item21)，不过上面例子中，我们使用了自定义销毁器，用`std::make_shared`就没办法做到。第二，如果你必须传给`std::shared_ptr`构造函数原始指针，直接传new出来的结果，不要传指针变量。如果上面代码第一部分这样重写：

```
std::shared_ptr<Widget> spw1(new Widget, // 直接使用new的结果
                             loggingDel);
```

会少了很多创建第二个从原始指针上构造`std::shared_ptr`的诱惑。相应的，创建spw2也会很自然的用spw1作为初始化参数（即用`std::shared_ptr`拷贝构造），那就没什么问题了：

```
std::shared_ptr<Widget> spw2(spw1);  // spw2使用spw1一样的控制块
```

一个尤其令人意外的地方是使用**this**原始指针作为`std::shared_ptr`构造函数实参的时候可能导致创建多个控制块。假设我们的程序使用`std::shared_ptr`管理**Widget**对象，我们有一个数据结构用于跟踪已经处理过的**Widget**对象：

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

继续，假设**Widget**有一个用于处理的成员函数：

```
class Widget {
public:
    …
    void process();
    …
};
```

对于**Widget::process**看起来合理的代码如下：

```
void Widget::process()
{
    …                                           // 处理Widget
    processedWidgets.emplace_back(this); // 然后将他加到已处理过的Widget的列表中
                                                // 这是错的
}
```

评论已经说了这是错的——或者至少大部分是错的。（错误的部分是传递this，而不是使用了**emplace_back**。如果你不熟悉**emplace_back**，参见Item42）。上面的代码可以通过编译，但是向容器传递一个原始指针（this），`std::shared_ptr`会由此为指向的对象（`*this`）创建一个控制块。那看起来没什么问题，直到你意识到如果成员函数外面早已存在指向**Widget**对象的指针，它是未定义行为的Game, Set, and Match（译注：一部电影，但是译者没看过。。。）。

`std::shared_ptr`API已有处理这种情况的设施。它的名字可能是C++标准库中最奇怪的一个：`std::enable_shared_from_this`。它是一个用做基类的模板类，模板类型参数是某个想被`std::shared_ptr`管理且能从该类型的**this**对象上安全创建`std::shared_ptr`指针的存在。在我们的例子中，**Widget**将会继承自`std::enable_shared_from_this`：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    …
    void process();
    …
};
```

正如我所说，`std::enable_shared_from_this`是一个用作基类的模板类。它的模板参数总是某个继承自它的类，所以**Widget**继承自`std::enable_shared_from_this<Widget>`。如果某类型继承自一个由该类型（译注：作为模板类型参数）进行模板化得到的基类这个东西让你心脏有点遭不住，别去想它就好了。代码完全合法，而且它背后的设计模式也是没问题的，并且这种设计模式还有个标准名字，尽管该名字和`std::enable_shared_from_this`一样怪异。这个标准名字就是奇异递归模板模式（The Curiously Recurring Template Pattern(CRTP)）。如果你想学更多关于它的内容，请搜索引擎一展身手，现在我们要回到`std::enable_shared_from_this`上。

`std::enable_shared_from_this`定义了一个成员函数，成员函数会创建指向当前对象的`std::shared_ptr`却不创建多余控制块。这个成员函数就是`shared_from_this`，无论在哪当你想使用`std::shared_ptr`指向this所指对象时都请使用它。这里有个`Widget::process`的安全实现：

```
void Widget::process()
{
    // 和之前一样，处理Widget
    …
    // 把指向当前对象的shared_ptr加入processedWidgets
    processedWidgets.emplace_back(shared_from_this());
}
```

从内部来说，`shared_from_this`查找当前对象控制块，然后创建一个新的`std::shared_ptr`指向这个控制块。设计的依据是当前对象已经存在一个关联的控制块。要想符合设计依据的情况，必须已经存在一个指向当前对象的`std::shared_ptr`(即调用shared_from_this的成员函数外面已经存在一个`std::shared_ptr`)。如果没有`std::shared_ptr`指向当前对象（即当前对象没有关联控制块），行为是未定义的，shared_from_this通常抛出一个异常。

要想防止客户端在调用`std::shared_ptr`前先调用`shared_from_this`，继承自`std::enable_shared_from_this`的类通常将它们的构造函数声明为private，并且让客户端通过工厂方法创建`std::shared_ptr`。以**Widget**为例，代码可以是这样：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    // 完美转发参数的工厂方法
    template<typename... Ts>
    static std::shared_ptr<Widget> create(Ts&&... params);
    …
    void process(); // 和前面一样
    …
private:
    …
};
```

现在，你可能隐约记得我们讨论控制块的动机是想了解`std::shared_ptr`关联一个控制块的成本。既然我们已经知道了怎么避免创建过多控制块，就让我们回到原来的主题。

控制块通常只占几个word大小，自定义销毁器和分配器可能会让它变大一点。通常控制块的实现比你想的更复杂一些。它使用继承，甚至里面还有一个虚函数（用来确保指向的对象被正确销毁）。这意味着使用`std::shared_ptr`还会招致控制块使用虚函数带来的成本。

了解了动态分配控制块，任意大小的销毁器和分配器，虚函数机制，原子引用计数修改，你对于`std::shared_ptr`的热情可能有点消退。可以理解，对每个资源管理问题来说都没有最佳的解决方案。但就它提供的功能来说，`std::shared_ptr`的开销是非常合理的。在通常情况下，`std::shared_ptr`创建控制块会使用默认销毁器和默认分配器，控制块只需三个word大小。它的分配基本上是无开销的。（开销被并入了指向的对象的分配成本里。细节参见Item21）。对`std::shared_ptr`解引用的开销不会比原始指针高。执行原子引用计数修改操作需要承担一两个原子操作开销，这些操作通常都会一一映射到机器指令上，所以即使对比非原子指令来说，原子指令开销较大，但是它们仍然只是单个指令。对于每个被`std::shared_ptr`指向的对象来说，控制块中的虚函数机制产生的开销通常只需要承受一次，即对象销毁的时候。

作为这些轻微开销的交换，你得到了动态分配的资源的生命周期自动管理的好处。大多数时候，比起手动管理，使用`std::shared_ptr`管理共享性资源都是非常合适的。如果你还在犹豫是否能承受`std::shared_ptr`带来的开销，那就再想想你是否需要共享资源。如果独占资源可行或者可能可行，用`std::unique_ptr`是一个更好的选择。它的性能profile更接近于原始指针，并且从`std::unique_ptr`升级到`std::shared_ptr`也很容易，因为`std::shared_ptr`可以从`std::unique_ptr`上创建。

反之不行。当你的资源由`std::shared_ptr`管理，现在又想修改资源生命周期管理方式是没有办法的。即使引用计数为一，你也不能重新修改资源所有权，改用`std::unique_ptr`管理它。所有权和`std::shared_ptr`指向的资源之前签订的协议是"除非死亡否则永不分离"。不能离婚，不能废除，没有特许。

`std::shared_ptr`不能处理的另一个东西是数组。和`std::unique_ptr`不同的是，`std::shared_ptr`的API设计之初就是针对单个对象的，没有办法`std::shared_ptr<T[]>`。一次又一次，"聪明"的程序员踌躇于是否该使用`std::shared_ptr<T>`指向数组，然后传入自定义数组销毁器。（即`delete []`）。这可以通过编译，但是是一个糟糕的注意。一方面，`std::shared_ptr`没有提供`operator[]`重载，所以数组索引操作需要借助怪异的指针算术。另一方面，`std::shared_ptr`支持转换为指向基类的指针，这对于单个对象来说有效，但是当用于数组类型时相当于在类型系统上开洞。（出于这个原因，`std::unique_ptr`禁止这种转换。）。更重要的是，C++11已经提供了很多内置数组的候选方案（比如`std::array`,`std::vector`,`std::string`）。声明一个指向傻瓜数组的智能指针几乎总是标识着糟糕的设计。

记住:

- `std::shared_ptr`为任意共享所有权的资源一种自动垃圾回收的便捷方式。
- 较之于`std::unique_ptr`，`std::shared_ptr`对象通常大两倍，控制块会产生开销，需要原子引用计数修改操作。
- 默认资源销毁是通过**delete**，但是也支持自定义销毁器。销毁器的类型是什么对于`std::shared_ptr`的类型没有影响。
- 避免从原始指针变量上创建`std::shared_ptr`。

Item 20: 当std::shard_ptr可能悬空时使用std::weak_ptr

自相矛盾的是，如果有一个像`std::shared_ptr`的指针但是不参与资源所有权共享的指针是很方便的。换句话说，是一个类似`std::shared_ptr`但不影响对象引用计数的指针。这种类型的智能指针必须要解决一个`std::shared_ptr`不存在的问题：可能指向已经销毁的对象。一个真正的智能指针应该跟踪所值对象，在悬空时知晓，悬空(*dangle*)就是指针指向的对象不再存在。这就是对`std::weak_ptr`最精确的描述。

你可能想知道什么时候该用`std::weak_ptr`。你可能想知道关于`std::weak_ptr`API的更多。它什么都好除了不太智能。`std::weak_ptr`不能解引用，也不能测试是否为空值。因为`std::weak_ptr`不是一个独立的智能指针。它是`std::shared_ptr`的增强。

这种关系在它创建之时就建立了。`std::weak_ptr`通常从`std::shared_ptr`上创建。当从`std::shared_ptr`上创建`std::weak_ptr`时两者指向相同的对象，但是`std::weak_ptr`不会影响所指对象的引用计数：

```
auto spw =                    // after spw is constructed
std::make_shared<Widget>(); // the pointed-to Widget's
                              // ref count(RC) is 1
                              // See Item 21 for in on std::make_shared
…
std::weak_ptr<Widget> wpw(spw); // wpw points to same Widget as spw. RC remains 1
…
spw = nullptr;                    // RC goes to 0, and the
                                  // Widget is destroyed.
                                  // wpw now dangles
```

`std::weak_ptr`用**expired**来表示已经dangle。你可以用它直接做测试：

```
if (wpw.expired()) … // if wpw doesn't point to an object
```

但是通常你期望的是检查`std::weak_ptr`是否已经失效，如果没有失效则访问其指向的对象。这做起来比较容易。因为缺少解引用操作，没有办法写这样的代码。即使有，将检查和解引用分开会引入竞态条件：在调用**expired**和解引用操作之间，另一个线程可能对指向的对象重新赋值或者析构，并由此造成对象已析构。这种情况下，你的解引用将会产生未定义行为。

你需要的是一个原子操作实现检查是否过期，如果没有过期就访问所指对象。这可以通过从`std::weak_ptr`创建`std::shared_ptr`来实现，具体有两种形式可以从`std::weak_ptr`上创建`std::shared_ptr`，具体用哪种取决于`std::weak_ptr`过期时你希望`std::shared_ptr`表现出什么行为。一种形式是`std::weak_ptr::lock`，它返回一个`std::shared_ptr`，如果`std::weak_ptr`过期这个`std::shared_ptr`为空：

```
std::shared_ptr<Widget> spw1 = wpw.lock();  // if wpw's expired, spw1 is null

auto spw2 = wpw.lock();                     // same as above, but uses auto
```

另一种形式是以`std::weak_ptr`为实参构造`std::shared_ptr`。这种情况中，如果`std::weak_ptr`过期，会抛出一个异常：

```
std::shared_ptr<Widget> spw3(wpw);          // if wpw's expired, throw
std::bad_weak_ptr
```

但是你可能还想知道为什么`std::weak_ptr`就有用了。考虑一个工厂函数，它基于一个UID从只读对象上产出智能指针。根据Item18的描述，工厂函数会返回一个该对象类型的`std::unique_ptr`：

```
    std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

如果调用`loadWidget`是一个昂贵的操作（比如它操作文件或者数据库I/O）并且对于ID来重复使用很常见，一个合理的优化是再写一个函数除了完成`loadWidget`做的事情之外再缓存它的结果。当请求获取一个Widget时阻塞在缓存操作上这本身也会导致性能问题，所以另一个合理的优化可以是当Widget不再使用的时候销毁它的缓存。

对于可缓存的工厂函数，返回`std::unique_ptr`不是好的选择。调用者应该接收缓存对象的智能指针，调用者也应该确定这些对象的生命周期，但是缓存本身也需要一个指针指向它所缓的对象。缓存对象的指针需要知道它是否已经悬空，因为当工厂客户端使用完工厂产生的对象后，对象将被销毁，关联的缓存条目会悬空。所以缓存应该使用`std::weak_ptr`，这可以知道是否已经悬空。这意味着工厂函数返回值类型应该是`std::shared_ptr`，因为只有当对象的生命周期由`std::shared_ptr`管理时，`std::weak_ptr`才能检测到悬空。

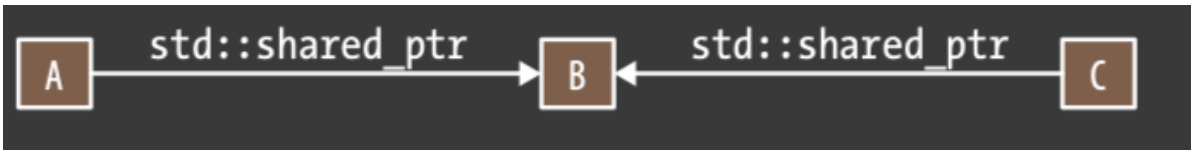
下面是一个临时凑合的`loadWidget`的缓存版本的实现：

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
    static std::unordered_map<WidgetID,
                              std::weak_ptr<const Widget>> cache; // 译者注：这里是
高亮
    auto objPtr = cache[id].lock();     // objPtr is std::shared_ptr
                                        // to cached object
                                        // (or null if object's not in cache)
    if (!objPtr) {                      // if not in cache
        objPtr = loadWidget(id);        // load it
        cache[id] = objPtr;             // cache it
    }
    return objPtr;
}
```

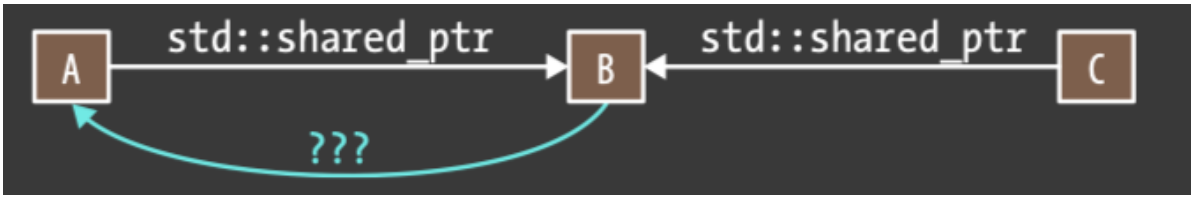
这个实现使用了C++11的hash表容器`std::unordered_map`，但是需要的`WidgetID`哈希和相等性比较函数在这里没有展示。

`fastLoadWidget`的实现忽略了以下事实：cache可能会累积过期的`std::weak_ptr`(对应已经销毁的`Widget`)。可以改进实现方式，但不要花时间在不会引起对`std :: weak_ptr`的深入了解的问题上，让我们考虑第二个用例：观察者设计模式。此模式的主要组件是subjects（状态可能会更改的对象）和observers（状态发生更改时要通知的对象）。在大多数实现中，每个subject都包含一个数据成员，该成员持有指向其observer的指针。这使subject很容易发布状态更改通知。subject对控制observers的生命周期（例如，当它们被销毁时）没有兴趣，但是subject对确保observers被销毁时，不会访问它具有极大的兴趣 。一个合理的设计是每个subject持有其observers的`std::weak_ptr`，因此可以在使用前检查是否已经悬空。

作为最后一个使用`std::weak_ptr`的例子，考虑一个持有三个对象A、B、C的数据结构，A和C共享B的所有权，因此持有`std::shared_ptr`：



假定从B指向A的指针也很有用。应该使用哪种指针？



有三种选择：

- **原始指针**。使用这种方法，如果A被销毁，但是C继续指向B，B就会有一个指向A的悬空指针。而且B不知道指针已经悬空，所以B可能会继续访问，就会导致未定义行为。
- **`std::shared_ptr`**。这种设计，A和B都互相持有对方的`std::shared_ptr`，导致`std::shared_ptr`在销毁时出现循环。即使A和B无法从其他数据结构被访问（比如，C不再指向B），每个的引用计数都是1。如果发生了这种情况，A和B都被泄露：程序无法访问它们，但是资源并没有被回收。
- **`std::weak_ptr`**。这避免了上述两个问题。如果A被销毁，B还是有悬空指针，但是B可以检查。尤其是尽管A和B互相指向，B的指针不会影响A的引用计数，因此不会导致无法销毁。

使用`std::weak_ptr`显然是这些选择中最好的。但是，需要注意使用`std::weak_ptr`打破`std::shared_ptr`循环并不常见。在严格分层的数据结构比如树，子节点只被父节点持有。当父节点被销毁时，子节点就被销毁。从父到子的链接关系可以使用`std::unique_ptr`很好的表征。从子到父的反向连接可以使用原始指针安全实现，因此子节点的生命周期肯定短于父节点。因此子节点解引用一个悬垂的父节点指针是没有问题的。

当然，不是所有的使用指针的数据结构都是严格分层的，所以当发生这种情况时，比如上面所述cache和观察者情况，知道`std::weak_ptr`随时待命也是不错的。

从效率角度来看，`std::weak_ptr`与`std::shared_ptr`基本相同。两者的大小是相同的，使用相同的控制块（参见Item 19），构造、析构、赋值操作涉及引用计数的原子操作。这可能让你感到惊讶，因为本Item开篇就提到`std::weak_ptr`不影响引用计数。我写的是`std::weak_ptr`不参与对象的*共享所有权*，因此不影响指向对象的引用计数。实际上在控制块中还是有第二个引用计数，`std::weak_ptr`操作的是第二个引用计数。想了解细节的话，继续看Item 21吧。

记住

- 像`std::shared_ptr`使用`std::weak_ptr`可能会悬空。
- `std::weak_ptr`的潜在使用场景包括：caching、observer lists、打破`std::shared_ptr`指向循环。

Item 21:优先考虑使用std::make_unique和std::make_shared而非new

让我们先对`std::make_unique`和`std::make_shared`做个铺垫。`std::make_shared`是C++11标准的一部分，但很可惜的是，`std::make_unique`不是。它从C++14开始加入标准库。如果你在使用C++11，不用担心，一个基础版本的`std::make_unique`是很容易自己写出的，如下：

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

正如你看到的，`make_unique`只是将它的参数完美转发到所要创建的对象的构造函数，从新产生的原始指针里面构造出`std::unique_ptr`，并返回这个`std::unique_ptr`。这种形式的函数不支持数组和自定义析构，但它给出了一个示范：只需一点努力就能写出你想要的`make_uniqe`函数。需要记住的是，不要把它放到std命名空间中，因为你可能并不希望在升级厂家编译器到符合C++14标准的时候产生冲突。

`std::make_unique`和`std::make_shared`有三个make functions中的两个：接收抽象参数，完美转发到构造函数去动态分配一个对象，然后返回这个指向这个对象的指针。第三个make function 是`std::allocate_shared.`它和`std::make_shared`一样，除了第一个参数是用来动态分配内存的对象。

即使是对使用和不使用make函数创建智能指针的最简单比较，也揭示了为什么最好使用这些函数的第一个原因。例如：

```
auto upw1(std::make_unique<Widget>());  // with make func
std::unique_ptr<Widget> upw2(new Widget);  // without make func
auto spw1(std::make_shared<Widget>());  // with make func
std::shared_ptr<Widget> spw2(new Widget);  // without make func
```

我高亮了区别：使用new的版本重复了类型，但是make function的版本没有。(译者注：这里高亮的是Widget，用new的声明语句需要写2遍Widget，make function只需要写一次) 重复写类型和软件工程里面一个关键原则相冲突：应该避免重复代码。源代码中的重复增加了编译的时间，会导致目标代码冗余，并且通常会让代码库使用更加困难。它经常演变成不一致的代码，而代码库中的不一致常常导致bug。此外，打两次字比一次更费力，而且谁不喜欢减少打字负担？

第二个使用make function的原因和异常安全有段。假设我们有个函数按照某种优先级处理Widget：

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

根据值传递`std::shared_ptr`可能看起来很可疑，但是Item 41解释了，如果processWidget总是复制`std::shared_ptr`(例如，通过将其存储在已处理的Widget的数据结构中)，那么这可能是一个可复用的设计选择。

现在假设我们有一个函数来计算相关的优先级

`int computePriority();`

并且我们在调用processWidget时使用了new而不是std:: make_shared

```
processWidget(std::shared_ptr<Widget>(new Widget), computePriority()); //
potential resource leak!
```

如注释所说，这段代码可能在new Widget时发生泄露。为何？调用的代码和被调用的函数都用`std::shared_ptr`s,且`std::shared_ptr`s就是设计出来防止泄露的。它们会在最后一个`std::shared_ptr`销毁时自动释放所指向的内存。如果每个人在每个地方都用`std::shared_ptr`s,这段代码怎么会泄露呢?

答案和编译器将源码转换为目标代码有关。在运行时，一个函数的参数必须先被计算，才能被调用，所以在调用processWidget之前，必须执行以下操作，processWidget才开始执行:

- 表达式'new Widget'必须计算，例如,一个Widget对象必须在堆上被创建
- 负责管理new出来指针的`std::shared_ptr<Widget>`构造函数必须被执行
- computePriority()必须运行

编译器不需要按照执行顺序生成代码。“new Widget"必须在`std::shared_ptr`的构造函数被调用前执行，因为new出来的结果作为构造函数的参数，但compute Priority可能在这之前，之后，或者之间执行。也就是说，编译器可能按照这个执行顺序生成代码:

1. 执行new Widget
2. 执行computePriority
3. 运行`std::shared_ptr`构造函数

如果按照这样生成代码，并且在运行是computePriority产生了异常，那么第一步动态分配的Widget就会泄露。因为它永远都不会被第三步的`std::shared_ptr`所管理了。

使用`std::make_shared`可以防止这种问题。调用代码看起来像是这样:

```
processWidget(std::make_shared<Widget>(), computePriority());
```

在运行时，`std::make_shared`和computePriority会先被调用。如果是`std::make_shared`，在computePriority调用前，动态分配Widget的原始指针会安全的保存在作为返回值的`std::shared_ptr`中。如果compu tePriority生成一个异常，那么`std::shared_ptr`析构函数将确保管理的Widget被销毁。如果首先调用computePriority并产生一个异常，那么`std::make_shared`将不会被调用，因此也就不需要担心new Widget(会泄露)。

如果我们将`std::shared_ptr`,`std::make_shared`替换成std::unique_ptr,std::make_unique,同样的道理也适用。因此，在编写异常安全代码时，使用std::make_unique而不是new与使用`std::make_shared`同样重要。

`std::make_shared`的一个特性(与直接使用new相比)得到了效率提升。使用`std::make_shared`允许编译器生成更小，更快的代码，并使用更简洁的数据结构。考虑以下对new的直接使用:

```
std::shared_ptr<Widget> spw(new Widget);
```

显然，这段代码需要进行内存分配，但它实际上执行了两次。Item 19解释了每个`std::shared_ptr`指向一个控制块，其中包含被指向对象的引用计数。这个控制块的内存在`std::shared_ptr`构造函数中分配。因此，直接使用new需要为Widget分配一次内存，为控制块分配再分配一次内存。

如果使用`std::make_shared`代替: `auto spw = std::make_shared_ptr<Widget>();`一次分配足矣。这是因为`std::make_shared`分配一块内存，同时容纳了Widget对象和控制块。这种优化减少了程序的静态大小，因为代码只包含一个内存分配调用，并且它提高了可执行代码的速度，因为内存只分配一次。此外，使用`std::make_shared`避免了对控制块中的某些簿记信息的需要，潜在地减少了程序的总内存占用。

对于`std::make_shared`的效率分析同样适用于`std::allocate_shared`，因此`std::make_shared`的性能优势也扩展到了该函数。

更倾向于使用函数而不是直接使用new的争论非常激烈。尽管它们在软件工程、异常安全和效率方面具有优势，但本item的意见是，更倾向于使用make函数，而不是完全依赖于它们。这是因为有些情况下它们不能或不应该被使用。

例如，没有make函数允许指定定制的析构(见item18和19)，但是`std::unique_ptr`和`std::shared_ptr`有构造函数这么做。给Widget自定义一个析构:

```
auto widgetDeleter = [](Widget*){...};
```

使用new创建智能指针非常简单:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
 upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

对于make函数，没有办法做同样的事情。

make函数第二个限制来自于其单一概念的句法细节。Item7解释了，当构造函数重载，有`std::initializer_list`作为参数和不用其作为参数时，用大括号创建对象更倾向于使用`std::initializer_list`作为参数的构造函数，而用圆括号创建对象倾向于不用`std::initializer_list`作为参数的构造函数。make函数会将它们的参数完美转发给对象构造函数，但是它们是使用圆括号还是大括号？对某些类型，问题的答案会很不相同。例如，在这些调用中，

```
auto upv = std::make_unique<std::vector<int>>(10, 20);
auto spv = std::make_shared<std::vector<int>>(10, 20);
```

生成的智能指针是否指向带有10个元素的`std::vector`，每个元素值为20，或指向带有两个元素的`std::vector`，其中一个元素值10，另一个为20 ?或者结果是不确定的?

好消息是这并非不确定：两种调用都创建了10个元素，每个值为20.这意味着在make函数中，完美转发使用圆括号，而不是大括号。坏消息是如果你想用大括号初始化指向的对象，你必须直接使用new。使用make函数需要能够完美转发大括号初始化，但是，正如item31所说，大括号初始化无法完美转发。但是，item30介绍了一个变通的方法：使用auto类型推导从大括号初始化创建std::initializer_list对象(见Item 2)，然后将auto创建的对象传递给make函数。

```
// create std::initializer_list
auto initList = { 10, 20 };
// create std::vector using std::initializer_list ctor
auto spv = std::make_shared<std::vector<int>>(initList);
```

对于std::unique_ptr,只有这两种情景（定制删除和大括号初始化）使用make函数有点问题。对于`std::shared_ptr`和它的make函数，还有至少2个问题。都属于边界问题，但是一些开发者常碰到，你也可能是其中之一。

一些类重载了operator new和operator delete。这些函数的存在意味着对这些类型的对象的全局内存分配和释放是不合常规的。设计这种定制类往往只会精确的分配、释放对象的大小。例如，Widget类的operator new和operator delete只会处理sizeof(Widget)大小的内存块的分配和释放。这种常识不太适用于`std::shared_ptr`对定制化分配(通过std::allocate_shared)和释放(通过定制化deleters)，因为std::allocate_shared需要的内存总大小不等于动态分配的对象大小，还需要再加上控制块大小。因此，适用make函数去创建重载了operator new 和 operator delete类的对象是个典型的糟糕想法。

与直接使用new相比，`std::make_shared`在大小和速度上的优势源于`std::shared_ptr`的控制块与指向的对象放在同一块内存中。当对象的引用计数降为0，对象被销毁(析构函数被调用).但是，因为控制块和对象被放在同一块分配的内存块中，直到控制块的内存也被销毁，它占用的内存是不会被释放的。

正如我说，控制块除了引用计数，还包含簿记信息。引用计数追踪有多少`std::shared_ptr`s指向控制块，但控制块还有第二个计数，记录多少个std::weak_ptrs指向控制块。第二个引用计数就是weak count。当一个std::weak_ptr检测对象是否过期时(见item 19),它会检测指向的控制块中的引用计数(而不是weak count)。如果引用计数是0(即对象没有`std::shared_ptr`再指向它，已经被销毁了)，std::weak_ptr已经过期。否则就没过期。

只要std::weak_ptrs引用一个控制块(即weak count大于零)，该控制块必须继续存在。只要控制块存在，包含它的内存就必须保持分配。通过`std::shared_ptr` make函数分配的内存，直到最后一个`std::shared_ptr`和最后一个指向它的std::weak_ptr已被销毁，才会释放。

如果对象类型非常大，而且销毁最后一个`std::shared_ptr`和销毁最后一个std::weak_ptr之间的时间很长，那么在销毁对象和释放它所占用的内存之间可能会出现延迟。

```
class ReallyBigType { … };

// 通过std::make_shared创建一个大对象
auto pBigObj = std::make_shared<ReallyBigType>();

…           // 创建 std::shared_ptrs 和 std::weak_ptrs
            // 指向这个对象，使用它们

…           // 最后一个 std::shared_ptr 在这销毁，
            // 但 std::weak_ptrs 还在

…           // 在这个阶段，原来分配给大对象的内存还分配着

…           // 最后一个std::weak_ptr在这里销毁；
            // 控制块和对象的内存被释放
```

直接只用new，一旦最后一个`std::shared_ptr`被销毁，ReallyBigType对象的内存就会被释放：

```
class ReallyBigType { … };

//通过new创建特大对象
std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);

…           // 像之前一样，创建 std::shared_ptrs 和 std::weak_ptrs
            // 指向这个对象，使用它们

…           // 最后一个 std::shared_ptr 在这销毁，
            // 但 std::weak_ptrs 还在

            // memory for object is deallocated

…           // 在这阶段，只有控制块的内存仍然保持分配

…           // 最后一个std::weak_ptr在这里销毁；
            // 控制块内存被释放
```

如果你发现自己处于不可能或不合适使用`std::make_shared`的情况下，你将想要保证自己不受我们之前看到的异常安全问题的影响。最好的方法是确保在直接使用new时，在一个不做其他事情的语句中，立即将结果传递到智能指针构造函数。这可以防止编译器生成的代码在使用new和调用管理新对象的智能指针的构造函数之间发生异常。

例如，考虑我们前面讨论过的processWidget函数，对其非异常安全调用的一个小修改。这一次，我们将指定一个自定义删除器:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
void cusDel(Widget *ptr); // 自定义删除器
```

这是非异常安全调用:

```
//和之前一样，潜在的内存泄露
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel),
    computePriority()
);
```

回想一下:如果computePriority在"new Widget"之后，而在`std::shared_ptr`构造函数之前调用，并且如果computePriority产生一个异常，那么动态分配的Widget将会泄漏。

这里使用自定义删除排除了对`std::make_shared`的使用，因此避免这个问题的方法是将Widget的分配和`std::shared_ptr`的构造放入它们自己的语句中，然后使用得到的`std::shared_ptr`调用processWidget。这是该技术的本质，不过，正如我们稍后将看到的，我们可以对其进行调整以提高其性能:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority());  // 正确，但是没优化，见下
```

这是可行的，因为`std::shared_ptr`假定了传递给它的构造函数的原始指针的所有权，即使构造函数产生了一个异常。此例中，如果spw的构造函数抛出异常(即无法为控制块动态分配内存)，仍然能够保证cusDel会在new Widget产生的指针上调用。

一个小小的性能问题是，在异常不安全调用中，我们将一个右值传递给processWidget

```
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel), // arg is rvalue
    computePriority()
);
```

但是在异常安全调用中，我们传递了左值

```
processWidget(spw, computePriority());  //spw是左值
```

因为processWidget的`std::shared_ptr`参数是传值，传右值给构造函数只需要move，而传递左值需要拷贝。对`std::shared_ptr`而言，这种区别是有意义的，因为拷贝`std::shared_ptr`需要对引用计数原子加，move则不需要对引用计数有操作。为了使异常安全代码达到异常不安全代码的性能水平，我们需要用std::move将spw转换为右值.

```
processWidget(std::move(spw), computePriority());
```

这很有趣，也值得了解，但通常是无关紧要的，因为您很少有理由不使用make函数。除非你有令人信服的理由这样做，否则你应该使用make函数。

记住：

- 和直接使用new相比，make函数消除了代码重复，提高了异常安全性。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用大括号初始化
- 对于`std::shared_ptr`s, make函数可能不被建议的其他情况包括
  (1)有自定义内存管理的类和
  (2)特别关注内存的系统，非常大的对象，以及`std::weak_ptr`s比对应的`std::shared_ptr`s活得更久

当使用Pimpl惯用法，请在实现文件中定义特殊成员函数

如果你曾经与过多的编译次数斗争过，你会对`Pimpl`(Pointer to implementation)惯用法很熟悉。 凭借这样一种技巧，你可以将一个**类数据成员**替换成一个指向包含具体实现的类或结构体的指针, 并将放在主类(primary class)的数据成员们移动到实现类去(implementation class), 而这些数据成员的访问将通过指针间接访问。 举个例子，假如有一个类`Widget`看起来如下:

```
class Widget()      //定义在头文件`widget.h`
{
public:
    Widget();
    ...
private:
    std::string name;
    std::vector<double> data;
    Gadget g1, g2, g3;  //Gadget是用户自定义的类型
}
```

因为类`Widget`的数据成员包含有类型`std::string`， `std::vector`和`Gadget`， 定义有这些类型的头文件在类`Widget`编译的时候，必须被包含进来，这意味着类`Widget`的使用者必须要`#include <string>,<vector>`以及`gadget.h`。 这些头文件将会增加类`Widget`使用者的编译时间，并且让这些使用者依赖于这些头文件。 如果一个头文件的内容变了，类`Widget`使用者也必须要重新编译。 标准库文件`<string>`和`<vector>`不是很常变，但是`gadget.h`可能会经常修订。

在C++98中使用`Pimpl`惯用法，可以把`Widget`的数据成员替换成一个原始指针(raw pointer)，指向一个已经被声明过却还未被定义的类，如下:

```
class Widget        //仍然在"Widget.h"中
{
public:
    Widget();
    ~Widget();       //析构函数在后面会分析
    ...

    private:
    struct Impl;     //声明一个 实现结构体
    Impl *pImpl;     //以及指向它的指针
}
```

因为类`Widget`不再提到类型`std:::string`,`std::vector`以及`Gadget`,`Widget`的使用者不再需要为了这些类型而引入头文件。 这可以加速编译，并且意味着，如果这些头文件中有所变动，`Widget`的使用者不会受到影响。

一个已经被声明，却还未被实现的类型，被称为**未完成类型**(incomplete type)。 `Widget::Impl`就是这种类型。 你能对一个未完成类型做的事很少，但是声明一个指向它指针是可以的。 `Pimpl`手法利用了这一点。

`Pimpl`惯用法的第一步，是声明一个数据成员，它是个指针，指向一个未完成类型。 第二步是动态分配(dynamic allocation)和回收一个对象，该对象包含那些以前在原来的类中的数据成员。 内存分配和回收的代码都写在**实现文件**(implementation file)里，比如，对于类`Widget`而言，写在`Widget.cpp`里:

```
#include "widget.h"      //以下代码均在实现文件 widget.cpp里
```

```
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl      //之前在Widget中声明的Widget::Impl类型的定义
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()         //为此Widget对象分配数据成员
: pImpl(new Impl)
{}

Widget::~Widget()
{delete pImpl;}          //销毁数据成员
```

在这里我把`#include`命令写出来是为了明确一点，对于头文件`std::string`,`std::vector`和`Gadget`的整体依赖依然存在。 然而，这些依赖从头文件`widget.h`(它被所有`Widget`类的使用者包含，并且对他们可见）移动到了`widget.cpp`（该文件只被`Widget`类的实现者包含，并只对它可见）。 我高亮了其中动态分配和回收`Impl`对象的部分(markdown高亮不了，实际是`new`和`delete`两部分——译者注)。这就是为什么我们需要`Widget`的析构函数——我们需要回收该对象。

但是，我展示给你们看的是一段C++98的代码，散发着一股已经过去了几千年的腐朽气息。 它使用了原始指针，原始的`new`和原始的`delete`，一切都让它如此的...原始。这一章建立在"智能指针比原始指针更好"的主题上，并且，如果我们想要的只是在类`Widget`的构造函数动态分配`Widget::impl`对象，在`Widget`对象销毁时一并销毁它， `std::unique_ptr`（见Item 18)是最合适的工具。 在头文件中用`std::unique_ptr`替代原始指针，就有了如下代码:

```
class Widget       //在"Widget.h"中
{
public:
    Widget();
    ...

    private:
    struct Impl;    //声明一个 实现结构体
    std::unique_ptr<Impl> pImpl;   //使用智能指针而不是原始指针
}
```

实现文件也可以改成如下:

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()        //根据Item 21， 通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
```

```
{}
```

你会注意到，`Widget`的析构函数不存在了。这是因为我们没有代码加在里面了。`std::unique_ptr`在自身析构时，会自动销毁它所指向的对象，所以我们自己无需手动销毁任何东西。这就是智能指针的众多优点之一：它使我们从手动资源释放中解放出来。

以上的代码能编译，但是，最普通的`Widget`用法却会导致编译出错：

```
#include "widget.h"

Wdiget w;       //编译出错
```

你所看到的错误信息根据编译器不同会有所不同，但是其文本一般会提到一些**有关于把`sizeof`和`delete`应用到未完成类型`incomplete type`上**的信息。对于未完成类型，使用以上操作是禁止的。

在`Pimpl`惯用法中使用`std::unique_ptr`会抛出错误，有点惊悚，因为第一`std::unique_ptr`宣称它支持未完成类型，第二`Pimpl`惯用法是`std::unique_ptr`的最常见的用法。 幸运的是，让这段代码能正常运行很简单。 只需要对是什么导致以上代码编译出错有一个基础的认识就可以了。

在对象`w`被析构时，例如离开了作用域(scope)，问题出现了。在这个时候，它的析构函数被调用。我们在类的定义里使用了`std::unique_ptr`，所以我们没有声明一个析构函数，因为我们并没有任何代码需要写在里面。根据编译器自动生成的*特殊成员函数*的规则(见 Item 17)，编译器会自动为我们生成一个析构函数。 在这个析构函数里，编译器会插入一些代码来调用类`Widget`的数据成员`Pimpl`的析构函数。`Pimpl`是一个`std::unique_ptr<Widget::Impl>`,也就是说，一个带有默认销毁器(default deleter)的`std::unique_ptr`。 默认销毁器(default deleter)是一个函数，它使用`delete`来销毁内置于`std::unique_ptr`的原始指针。然而，在使用`delete`之前，通常会使默认销毁器使用C++11的特性`static_assert`来确保原始指针指向的类型不是一个未完成类型。 当编译器为`Widget w`的析构生成代码时，它会遇到`static_assert`检查并且失败，这通常是错误信息的来源。 这些错误信息只在对象`w`销毁的地方出现，因为类`Widget`的析构函数，正如其他的编译器生成的特殊成员函数一样，是暗含`inline`属性的。 错误信息自身往往指向对象`w`被创建的那行，因为这行代码明确地构造了这个对象，导致了后面潜在的析构。

为了解决这个问题，你只需要确保在编译器生成销毁`std::unique_ptr<Widget::Imple>`的代码之前， `Widget::Impl`已经是一个完成类型(complete type)。 当编译器"看到"它的定义的时候，该类型就成为*完成类型*了。 但是`Widget::Impl`的定义在`wideget.cpp`里。成功编译的关键，就是，在`widget.cpp`文件内，让编译器在"看到" `Widget`的析构函数实现之前（也即编译器自动插入销毁`std::unique_ptr`的数据成员的位置），先定义`Wdiget::Impl`。

做出这样的调整很容易。只需要在先在`widget.h`里，只**声明**(declare)类`Widget`的析构函数，却不要在这里**定义**(define)它:

```
class Widget {      // as before, in "widget.h"
public:
  Widget();
  ~Widget();     // declaration only
  ...

private:     // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

在`widget.cpp`文件中，在结构体`Widget::Impl`被定义之后，再定义析构函数:

```
#include "widget.h"      //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl      //跟之前一样,定义Widget::Impl
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()         //根据Item 21,  通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget()        //析构函数的定义(译者注: 这里高亮)
{}
```

这样就可以了，并且这样增加的代码也最少，但是，如果你想要强调编译器自动生成的析构函数会工作的很好——你声明`Widget`的析构函数的唯一原因，是确保它会在`Widget`的实现文件内(指`widget.cpp`，译者注)被自动生成，你可以把析构函数体直接定义为`=default`:

```
    Widget::~Widget() = default;        //同上述代码效果一致
```

使用了`Pimpl`惯用法的类自然适合支持移动操作，因为编译器自动生成的移动操作正合我们所意： 对隐藏的`std::unique_ptr`进行移动。 正如`Item 17`所解释的那样，声明一个类`Widget`的析构函数会阻止编译器生成移动操作，所以如果你想要支持移动操作，你必须自己声明相关的函数。考虑到编译器自动生成的版本能够正常功能，你可能会被诱使着来这样实现:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(Widget&& rhs) = default;              //思路正确，但代码错误
    Widget& operator=(Widget&& rhs) = default;


    private:
    struct Impl;         //如上
    std::unique_ptr<Impl> pImpl;
}
```

这样的做法会导致同样的错误，和之前的声明一个不带析构函数的类的错误一样，并且是因为同样的原因。 编译器生成的移动赋值操作符(move assignment operator)，在重新赋值之前，需要先销毁指针`pImpl`指向的对象。然而在`Widget`的头文件里，`pImpl`指针指向的是一个未完成类型。情况和移动构造函数(move constructor)有所不同。 移动构造函数的问题是编译器自动生成的代码里，包含有抛出异常的事件，在这个事件里会生成销毁`pImpl`的代码。然而，销毁`pImpl`需要`Impl`是一个完成类型。

因为这个问题同上面一致，所以解决方案也一样——把移动操作的定义移动到实现文件里:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(Widget&& rhs);   //仅声明
    Widget& operator=(Widget&& rhs);


    private:
    struct Impl;        //如上
    std::unique_ptr<Impl> pImpl;
}
```

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样,定义Widget::Impl
{
    std::string name;
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()        //根据Item 21,  通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget() = default;

Widget(Widget&& rhs) = default;             //在这里定义
Widget& operator=(Widget&& rhs) = default;
```

`pImpl`惯用法是用来减少类实现者和类使用者之间的编译依赖的一种方法，但是，从概念而言，使用这种惯用法并不改变这个类的表现。 原来的类`Widget`包含有`std::string`,`std::vector`和`Gadget`数据成员，并且，假设类型`Gadget`，如同`std::string`和`std::vector`一样,允许复制操作，所以类`Widget`支持复制操作也很合理。 我们必须要自己来写这些函数，因为第一，对包含有**只可移动(move-only)**类型，如`std::unique_ptr`的类，编译器不会生成复制操作;第二，即使编译器帮我们生成了，生成的复制操作也只会复制`std::unique_ptr`(也即浅复制(shallow copy)),而实际上我们需要复制指针所指向的对象(也即深复制(deep copy))。

使用我们已经熟悉的方法，我们在头文件里声明函数，而在实现文件里去实现他们:

```
class Widget        //在"Widget.h"中
{
public:
    Widget();
    ~Widget();
    ...

    Widget(const Widget& rhs);   //仅声明
    Widget& operator=(const Widget& rhs);
```

```
    private:
    struct Impl;        //如上
    std::unique_ptr<Impl> pImpl;
}
```

```
#include "widget.h"     //以下代码均在实现文件 widget.cpp里
#include "gadget.h"
#include <string>
#include <vector>
struct Widget::Impl     //跟之前一样,定义Widget::Impl
{
    ...
}

Widget::Widget()        //根据Item 21, 通过std::make_shared来创建std::unique_ptr
: pImpl(std::make_unique<Imple>())
{}

Widget::~Widget() = default;
    ...
Widget::Widget(const Widget& rhs)
:pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs)
{
    *pImpl = *rhs.pImpl;
    return *this;
}
```

两个函数的实现都比较中规中矩。 在每个情况中，我们都只从源对象(rhs)中，复制了结构体`Impl`的内容到目标对象中(*this)。我们利用了编译器会为我们自动生成结构体`Impl`的复制操作函数的机制，而不是逐一复制结构体`Impl`的成员，自动生成的复制操作能自动复制每一个成员。 因此我们通过调用`Widget::Impl`的编译器生成的复制操作函数来实现了类`Widget`的复制操作。 在复制构造函数中，注意，我们仍然遵从了Item 21的建议，使用`std::make_unique`而非直接使用`new`。

为了实现`Pimpl`惯用法，`std::unique_ptr`是我们使用的智能指针，因为位于对象内部的`pImpl`指针(例如，在类`Widget`内部)，对所指向的对应实现的对象的享有独占所有权(exclusive ownership)。然而，有趣的是，如果我们使用`std::shared_ptr`而不是`std::unique_ptr`来做`pImpl`指针， 我们会发现本节的建议不再适用。 我们不需要在类`Widget`里声明析构函数，也不用用户定义析构函数，编译器将会愉快地生成移动操作，并且将会如我们所期望般工作。代码如下:

```
//在Widget.h中
class Widget{
public:
    Widget();
    ...         //没有对移动操作和析构函数的声明
    private:
    struct Impl;
    std::shared_ptr<Impl> pImpl;    //使用std::shared_ptr而非std::unique_ptr
}
```

而类`Widget`的使用者，使用`#include widget.h`，可以使用如下代码

```
Widget w1;
auto w2(std::move(w1));  //移动构造w2
w1 = std::move(w2);      //移动赋值w1
```

这些都能编译，并且工作地如我们所望: `w1`将会被默认构造，它的值会被移动进`w2`，随后值将会被移动回`w1`，然后两者都会被销毁(因此导致指向的`Widget::Impl`对象一并也被销毁)。

`std::unique_ptr`和`std::shared_ptr`在`pImpl`指针上的表现上的区别的深层原因在于，他们支持自定义销毁器(custom deleter)的方式不同。 对`std::unique_ptr`而言，销毁器的类型是`unique_ptr`的一部分，这让编译器有可能生成更小的运行时数据结构和更快的运行代码。 这种更高效率的后果之一就是`unique_ptr`指向的类型，在编译器的生成特殊成员函数被调用时(如析构函数，移动操作)时，必须已经是一个完成类型。 而对`std::shared_ptr`而言，销毁器的类型不是该智能指针的一部分，这让它会生成更大的运行时数据结构和稍微慢点的代码，但是当编译器生成的特殊成员函数被使用的时候，指向的对象不必是一个完成类型。(译者注: 知道`unique_ptr`和`shared_ptr`的实现，这一段才比较容易理解。)

对于`pImpl`惯用法而言，在`std::unique_ptr`和`std::shared_ptr`的特性之间，没有一个比较好的折中。 因为对于类`Widget`以及`Widget::Impl`而言，他们是独享占有权关系，这让`std::unique_ptr`使用起来很合适。 然而，有必要知道，在其他情况中，当共享所有权(shared ownership)存在时，`std::shared_ptr`是很适用的选择的时候，没有必要使用`std::unique_ptr`所必需的**声明——定义**(function-definition)这样的麻烦事了。

记住

- `pImpl`惯用法通过减少在类实现和类使用者之间的编译依赖来减少编译时间。
- 对于`std::unique_ptr`类型的`pImpl`指针，需要在头文件的类里声明特殊的成员函数，但是在实现文件里面来实现他们。即使是编译器自动生成的代码可以工作，也要这么做。
- 以上的建议只适用于`std::unique_ptr`，不适用于`std::shared_ptr`。

CHAPTER 5 RValue References, Move Semantics and Perfect Forwarding

当你第一次了解到**移动语义**和**完美转发**的时候，它们看起来非常直观:

- **移动语义**使编译器有可能用廉价的移动操作来代替昂贵的复制操作。正如复制构造函数和复制赋值操作符给了你赋值对象的权利一样，移动构造函数和移动赋值操作符也给了控制移动语义的权利。移动语义也允许创建**只可移动**(move-only)的类型，例如`std::unique_ptr`, `std::future` 和 `std::thread`。
- **完美转发**使接收任意数量参数的函数模板成为可能，它可以将参数转发到其他的函数，使目标函数接收到的参数与被传递给转发函数的参数保持一致。

右值引用是连接这两个截然不同的概念的胶合剂。它隐藏在语言机制之下，使移动语义和完美转发变得可能。

你对这些特点(features)越熟悉，你就越会发现，你的初印象只不过是冰山一角。移动语义、完美转发和右值引用的世界比它所呈现的更加微妙。
举个例子，`std::move`并不移动任何东西，完美转发也并不完美。移动操作并不永远比复制操作更廉价；即便如此，它也并不总是像你期望的那么廉价。而且，它也并不总是被调用，即使在当移动操作可用的时候。构造`type&&`也并非总是代表一个右值引用。

无论你挖掘这些特性有多深，它们看起来总是还有更多隐藏起来的部分。幸运的是，它们的深度总是有限的。本章将会带你到最基础的部分。一旦到达，`C++11`的这部分特性将会具有非常大的意义。比如，你会掌握`std::move`和`sd::forward`的惯用法。你能够对`type&&`的歧义性质感到舒服。你会理解移动操作的令人惊奇的不同代价的背后真相。这些片段都会豁然开朗。在这一点上，你会重新回到一开始的状态，因为移动语义、完美转发和右值引用都会又一次显得直截了当。但是这一次，它们不再使人困惑。

在本章的这些小节中，非常重要的一点是要牢记**参数**(parameter)永远是**左值**(lValue)，即使它的类型是一个右值引用。比如，假设

```
void f(Widget&& w);
```

参数`w`是一个左值，即使它的类型是一个**Widget**的右值引用(如果这里震惊到你了，请重新回顾从本书第二页开始的关于左值和右值的总览。)

Item 23: 理解std::move和std::forward

为了了解`std::move`和`std::forward`，一种有用的方式是从*它们不做什么*这个角度来了解它们。`std::move`不移动(move)任何东西，`std::forward`也不转发(forward)任何东西。在运行期间(run-time)，它们不做任何事情。它们不产生任何可执行代码，一字节也没有。

`std::move`和`std::forward`仅仅是执行转换(cast)的函数（事实上是函数模板）。`std::move`无条件的将它的参数转换为右值，而`std::forward`只在特定情况满足时下进行转换。
它们就是如此。这样的解释带来了一些新的问题，但是从根本上而言，这就是全部内容。

为了使这个故事更加的具体，这里是一个C++11的`std::move`的示例实现。它并不完全满足标准细则，但是它已经非常接近了。

```
template <typename T>                           //in namespace std
typename remove_reference<T>::type&&
move(T&& param)
{
    using ReturnType =                          // alias declaration;
    typename remove_reference<T>::type&&;       // 见 Item 9

    return static_cast<ReturnType>(param);
}
```

我为你们高亮了这段代码的两部分（译者注：markdown不支持代码段内高亮。高亮的部分为`move`和`static_cast`）。一个是函数名字，因为函数的返回值非常具有干扰性。而且我不想你们被它搞得晕头转向。另外一个高亮的部分是包含这段函数的本质的转换。正如你所见，`std::move`接受一个对象的引用（准确的说，一个通用引用(universal reference)，后见Item 24），返回一个指向同对象的引用。

该函数返回类型的`&&`部分表明`std::move`函数返回的是一个右值引用，但是，正如Item 28所解释的那样，如果类型`T`恰好是一个左值引用，那么`T&&`将会成为一个左值引用。为了避免如此，类型萃取器（type trait，见Item 9）`std::remove_reference`应用到了类型`T`上，因此确保了`&&`被正确的应用到了一个不是引用的类型上。这保证了`std::move`返回的真的是右值引用，这很重要，因为函数返回的右值引用是右值（rvalues）。因此，`std::move`将它的参数转换为一个右值，这就是它的全部作用。

此外，`std::move`在C++14中可以被更简单地实现。多亏了函数返回值类型推导（见Item 3)和标准库的模板别名`std::remove_reference_t`（见Item 9），`std::move`可以这样写：

```
template <typename T>
decltype(auto) move(T&& param)          //C++14;still in namesapce std
{
    using ReturnType = remove_referece_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

看起来更简单，不是吗？

因为`std::move`除了转换它的参数到右值以外什么也不做，有一些提议说它的名字叫`rvalue_cast`可能会更好。虽然可能确实是这样，但是它的名字已经是`std::move`，所以记住`std::move`做什么和不做什么很重要。它其实并不移动任何东西。

当然，右值本来就是移动操作的侯选者，所以对一个对象使用`std::move`就是告诉编译器，这个对象很适合被移动。所以这就是为什么`std::move`叫现在的名字：更容易指定可以被移动的对象。

事实上，右值只不过**经常**是移动操作的候选者。假设你有一个类，它用来表示一段注解。这个类的构造函数接受一个包含有注解的`std::string`作为参数，然后它复制该参数到类的数据成员（data member)。假设你了解Item 41,你声明一个值传递(by value)的参数：

```
class Annotation {
public:
    explicit Annotation(std::string text);  //将会被复制的参数
    ...                                     //如同 Item 41,
};                                          //值传递
```

但是`Annotation`类的构造函数仅仅是需要读取参数`text`的值，它并不需要修改它。为了和历史悠久的传统：能使用`const`就使用`const`保持一致，你修订了你的声明以使`text`变成`const`，

```
class Annotation {
public:
    explicit Annotation(const std::string text);
    ...
};
```

当复制参数`text`到一个数据成员的时候，为了避免一次复制操作的代价，你仍然记得来自Item 41的建议，把`std::move`应用到参数`text`上，因此产生一个右值，

```
class Annotation {
public:
    explicit Annotation(const std::string text)
    : value(std::move(text))    //"move" text到value上；这段代码执行起来
                                //并不如看起来那样
    {...}
    ...

    private:
        std::string value;
};
```

这段代码可以编译，可以链接，可以运行。这段代码将数据成员`value`设置为`text`的值。这段代码与你期望中的完美实现的唯一区别，是`text`并不是被移动到`value`，而是被**复制**。诚然，`text`通过`std::move`被转换到右值，但是`text`被声明为`const std::string`，所以在转换之前，`text`是一个左值的`const std::string`，而转换的结果是一个右值的`const std::string`，但是纵观全程，`const`属性一直保留。

当编译器决定哪一个`std::string`的构造函数被构造时，考虑它的作用，将会有两种可能性。

```
class string {                  //std::string事实上是
    public:                     //std::basic_string<char>的类型别名
    ...
    string(const string& rhs); //复制构造函数
    string(string&& rhs);       //移动构造函数
}
```

在类`Annotation`的构造函数的成员初始化列表(member initialization list)中，`std::move(text)`的结构是一个`const std::string`的右值。这个右值不能被传递给`std::string`的移动构造函数，因为移动构造函数只接受一个指向非常量(non-const)`std::string`的右值引用。然而，该右值却可以被传递给`std::string`的复制构造函数，因为指向常量的左值引用允许被绑定到一个常量右值上。因此，`std::string`在成员初始化的过程中调用了**复制构造函数**，即使`text`已经被转换成了右值。这样是为了确保维持常量属性的正确性。从一个对象中移动（Moving)出某个值通常代表着修改该对象，所以语言不允许常量对象被传递给可以修改他们的函数（例如移动构造函数）。

从这个例子中，可以总结出两点。第一，不要在你希望能移动对象的时候，声明他们为常量。对常量对象的移动请求会悄无声息的被转化为复制操作。第二点，`std::move`不仅不移动任何东西，而且它也不保证它执行转换的对象可以被移动。关于`std::move`，你能确保的唯一一件事就是将它应用到一个对象上，你能够得到一个右值。

关于`std::forward`的故事与`std::move`是相似的，但是与`std::move`总是**无条件**的将它的参数转换为右值不同，`std::forward`只有在满足一定条件的情况下才执行转换。`std::forward`是**有条件**的转换。要明白什么时候它执行转换，什么时候不，想想`std::forward`的典型用法。
最常见的情景是一个模板函数，接收一个通用引用参数(universal reference parameter)，并将它传递

给另外的函数:

```
void process(const Widget& lvalArg);  //左值处理
void process(Widget&& rvalArg);         //右值处理

template <typename T>                  //用以转发参数到process的模板
void logAndProcess(T&& param)
{
    auto now =                         //获取现在时间
        std::chrono::system_clock::now();
    makeLogEntry("calling 'process',now);
    process(std::forward<T>(param));
}
```

考虑两次对`logAndProcess`的调用，一次左值为参数，一次右值为参数，

```
Widget w;

logAndProcess(w);               //call with lvalue
logAndProcess(std::move(w));    //call with rvalue
```

在`logAndProcess`函数的内部，参数`param`被传递给函数`process`。函数`process`分别对左值和右值参数做了重载。当我们使用左值来调用`logAndProcess`时，自然我们期望该左值被当作左值转发给`process`函数，而当我们使用右值来调用`logAndProcess`函数时，我们期望`process`函数的右值重载版本被调用。

但是参数`param`，正如所有的其他函数参数一样，是一个左值。每次在函数`logAndProcess`内部对函数`process`的调用，都会因此调用函数`process`的左值重载版本。为防如此，我们需要一种机制(mechanism)：当且仅当传递给函数`logAndProcess`的用以初始化参数`param`的值是一个右值时，参数`param`会被转换为有一个右值。这就是为什么`std::forward`是一个**有条件**的转换：它只把由右值初始化的参数，转换为右值。

你也许会想知道`std::forward`是怎么知道它的参数是否是被一个右值初始化的。举个例子，在上述代码中，`std::forward`是怎么分辨参数`param`是被一个左值还是右值初始化的？ 简短的说，该信息藏在函数`logAndProcess`的模板参数`T`中。该参数被传递给了函数`std::forward`，它解开了含在其中的信息。该机制工作的细节可以查询 Item 28.

考虑到`std::move`和`std::forward`都可以归结于**转换**，他们唯一的区别就是`std::move`总是执行转换，而`std::forward`偶尔为之。你可能会问是否我们可以免于使用`std::move`而在任何地方只使用`std::forward`。 从纯技术的角度，答案是yes: `std::forward`是可以完全胜任，`std::move`并非必须。当然，其实两者中没有哪一个函数是**真的必须**的，因为我们可以到处直接写转换代码，但是我希望我们能同意：这将相当的，嗯，让人恶心。

`std::move`的吸引力在于它的便利性： 减少了出错的可能性，增加了代码的清晰程度。考虑一个类，我们希望统计有多少次移动构造函数被调用了。我们只需要一个静态的计数器(static counter)，它会在移动构造的时候自增。假设在这个类中，唯一一个非静态的数据成员是`std::string`，一种经典的移动构造函数（例如，使用std::move)可以被实现如下:

```
class Widget{
public:
    Widget(Widget&& rhs)
    : s(std::move(rhs.s))
    {
        ++moveCtorCalls;
    }
private:
    static std::size_t moveCtorCalls;
    std::string s;
}
```

如果要用`std::forward`来达成同样的效果，代码可能会看起来像

```
class Widget{
public:
    Widget(Widget&& rhs)                    //不自然，不合理的实现
    : s(std::forward<std::string>(rhs.s))
    {
        ++moveCtorCalls;
    }
    ...
}
```

注意，第一，`std::move`只需要一个函数参数(rhs.s)，而`std::forward`不但需要一个函数参数(rhs.s)，还需要一个模板类型参数`std::string`。其次，我们转发给`std::forward`的参数类型应当是一个**非引用**(non-reference)，因为传递的参数应该是一个右值（见 Item 28)。 同样，这意味着`std::move`比起`std::forward`来说需要打更少的字，并且免去了传递一个表示我们正在传递一个右值的类型参数。同样，它根绝了我们传递错误类型的可能性，（例如，`std::string&`可能导致数据成员`s`被复制而不是被移动构造）。

更重要的是，`std::move`的使用代表着无条件向右值的转换，而使用`std::forward`只对绑定了右值的引用进行到右值转换。这是两种完全不同的动作。前者是典型地为了移动操作，而后者只是传递（亦作转发）一个对象到另外一个函数，保留它原有的左值属性或右值属性。因为这些动作实在是差异太大，所以我们拥有两个不同的函数（以及函数名）来区分这些动作。

记住:

- `std::move`执行到右值的无条件的转换，但就自身而言，它不移动任何东西。
- `std::forward`只有当它的参数被绑定到一个右值时，才将参数转换为右值。
- `std::move`和`std::forward`在运行期什么也不做。

区分通用引用与右值引用

据说，真相使人自由，然而在特定的环境下，一个精心挑选的谎言也同样使人解放。这一节就是这样一个谎言。因为我们在和软件打交道，然而，让我们避开“谎言(lie)”这个词，不妨说，本节包含了一种“**抽象**(abstraction)”。

为了声明一个指向某个类型T的右值引用(Rvalue Reference), 你写下了`T&&`。由此，一个合理的假设是，当你看到一个`T&&`出现在源码中，你看到的是一个右值引用。唉，事情并不如此简单:

```
void f(Widget&& param);      //右值引用
Widget&& var1 = Widget();    //右值引用
auto&& var2 = var1;          //不是右值引用

template <typename T>
void f(std::vector<T>&& param); //右值引用

template <typename T>
void f(T&& param);           //不是右值引用
```

事实上，`T&&`有两种不同的意思。第一种，当然是右值引用。这种引用表现得正如你所期待的那样: 它们只绑定到右值上，并且它们主要的存在原因就是为了声明某个对象可以被移动。

`T&&`的第二层意思，是它既可以是一个右值引用，也可以是一个左值引用。这种引用在源码里看起来像右值引用（也即`T&&`),但是它们可以表现得它们**像是**左值引用(也即`T&`)。它们的二重性(dual nature)使它们既可以绑定到右值上(就像右值引用)，也可以绑定到左值上(就像左值引用)。 此外，它们还可以绑定到常量(const)和非常量(non-const)的对象上，也可以绑定到`volatile`和`non-volatile`的对象上，甚至可以绑定到即`const`又`volatile`的对象上。它们可以绑定到几乎任何东西。这种空前灵活的引用值得拥有自己的名字。我把它叫做**通用引用**(universal references)。（注: Item 25解释了`std::forward`几乎总是可以应用到通用引用上，并且在这本书即将出版之际，一些C++社区的成员已经开始将这种通用引用称之为**转发引用**(forwarding references))。

在两种情况下会出现通用引用。最常见的一种是函数模板参数，正如在之前的示例代码中所出现的例子:

```
    template <typename T>
    void f(T&& param);       //param是一个通用引用
```

第二种情况是`auto`声明符，包含从以上示例中取得的这个例子:

```
    auto&& val2 = var1;      //var2是一个通用引用
```

这两种情况的共同之处就是都存在**类型推导(type deduction)**。在模板`f`的内部，参数`param`的类型需要被推导，而在变量`var2`的声明中，`var2`的类型也需要被推导。同以下的例子相比较(同样来自于上面的示例代码)，下面的例子不带有类型推导。如果你看见`T&&`不带有类型推导，那么你看到的就是一个右值引用。

```
void f(Widget&& param);          //没有类型推导
                                 //param是一个右值引用
Widget&& var1 = Widget();        //没有类型推导
                                 //var1是一个右值引用
```

因为通用引用是引用，所以他们必须被初始化。一个通用引用的初始值决定了它是代表了右值引用还是左值引用。如果初始值是一个右值，那么通用引用就会是对应的右值引用，如果初始值是一个左值，那么通用引用就会是一个左值引用。对那些是函数参数的通用引用来说，初始值在调用函数的时候被提供:

```
template <typename T>
void f(T&& param);              //param是一个通用引用

Widget w;
f(w);                           //传递给函数f一个左值;参数param的类型
                                //将会是Widget&,也即左值引用

f(std::move(w));                //传递给f一个右值;参数param的类型会是
                                //Widget&&,即右值引用
```

对一个通用引用而言，类型推导是必要的,但是它还不够。声明引用的格式必须正确，并且这种格式是被限制的。它必须是准确的`T&&`。再看看之前我们已经看过的代码示例:

```
template <typename T>
void f(std::vector<T>&& param);     //param是一个右值引用
```

当函数`f`被调用的时候，类型`T`会被推导（除非调用者显式地指定它，这种边缘情况我们不考虑）。但是参数`param`的类型声明并不是`T&&`，而是一个`std::vector<T>&&`。这排除了参数`param`是一个通用引用的可能性。`param`因此是一个右值引用——当你向函数`f`传递一个左值时，你的编译器将会开心地帮你确认这一点:

```
std::vector<int> v;
f(v);                   //错误！不能将左值绑定到右值引用
```

即使是出现一个简单的`const`修饰符，也足以使一个引用失去成为通用引用的资格:

```
    template <typename T>
    void f(const T&& param);    //param是一个右值引用
```

如果你在一个模板里面看见了一个函数参数类型为`T&&`,你也许觉得你可以假定它是一个通用引用。错！这是由于在模板内部并不保证一定会发生类型推导。考虑如下`push_back`成员函数，来自`std::vector`:

```
template <class T,class Allocator = allocator<T>> //来自C++标准
    class vector
    {
        public:
        void push_back(T&& x);
        ...
    }
```

`push_back`函数的参数当然有资格成为一个通用引用，然而，在这里并没有发生类型推导。因为`push_back`在一个特有(particular)的`vector`实例化(instantiation)之前不可能存在，而实例化`vector`时的类型已经决定了`push_back`的声明。也就是说，

```
    std::vector<Widget> v;
```

将会导致`std::vector`模板被实例化为以下代码:

```
    class vector<Widget , allocagor<Widget>>
    {
        public:
        void push_back(Widget&& x);              // 右值引用
    }
```

现在你可以清楚地看到，函数`push_back`不包含任何类型推导。`push_back`对于`vector<T>`而言(有两个函数——它被重载了)总是声明了一个类型为指向`T`的右值引用的参数。

相反，`std::vector`内部的概念上相似的成员函数`emplace_back`，却确实包含类型推导:

```
template <class T,class Allocator = allocator<T>> //依旧来自C++标准
    class vector
    {
        public:
        template <class... Args>
        void emplace_back(Args&&... args);
        ...
    }
```

这儿，类型参数(type parameter)`Args`是独立于`vector`的类型参数之外的，所以`Args`会在每次`emplace_back`被调用的时候被推导(Okay,`Args`实际上是一个参数包(parameter pack),而不是一个类型参数，但是为了讨论之利，我们可以把它**当作**是一个类型参数)。

虽然函数`emplace_back`的类型参数被命名为`Args`，但是它仍然是一个通用引用，这补充了我之前所说的，通用引用的格式必须是`T&&`。 没有任何规定必须使用名字`T`。举个例子，如下模板接受一个通用引用，但是格式(`type&&`)是正确的，并且参数`param`的类型将会被推导（重复一次，不考虑边缘情况，也即当调用者明确给定参数类型的时候)。

```
template <typename MyTemplateType>          //param是通用引用
void someFunc(MyTemplateType&& param);
```

我之前提到，类型为`auto`的变量可以是通用引用。更准确地说，类型声明为`auto&&`的变量是通用引用，因为会发生类型推导，并且它们满足正确的格式要求(`T&&`)。`auto`类型的通用引用不如模板函数参数中的通用引用常见，但是它们在`C++11`中常常突然出现。而它们在`C++14`中出现地更多，因为`C++14`的匿名函数表达式(lambda expressions)可以声明`auto&&`类型的参数。举个例子，如果你想写一个`C++14`标准的匿名函数，来记录任意函数调用花费的时间，你可以这样:

```
    auto timeFuncInvocation =
    [](auto&& func, auto&&... params)           //C++14标准
    {
        start timer;
        std::forward<decltype(func)>(func)(     //对参数params调用func
            std::forward<delctype(params)>(params)...
        );
        stop timer and record elapsed time;
    };
```

如果你对位于匿名函数里的`std::forward<decltype(blah blah blah)>`反应是"What the ....!", 这只代表着你可能还没有读 Item 33。别担心。在本节，重要的事是匿名函数声明的`auto&&`类型的参数。`func`是一个通用引用，可以被绑定到任何可被调用的对象，无论左值还是右值。`args`是**0个**或者多个通用引用（也就是说，它是个通用引用参数包（a universal reference parameter pack)），它可以绑定到任意数目、任意类型的对象上。
多亏了`auto`类型的通用引用，函数`timeFuncInvocation`可以对**近乎任意**(pretty-much any)函数进行计时。(如果你想知道*任意(any)*和*近乎任意(pretty-much any*的区别，往后翻到 Item 30)。

牢记整个本小节——通用引用的基础——是一个谎言,uhh,一个“抽象”。隐藏在其底下的真相被称为"**引用折叠(reference collapsing)**",小节Item 28致力于讨论它。但是这个真相并不降低该抽象的有用程度。区分右值引用和通用引用将会帮助你更准确地阅读代码（"究竟我眼前的这个`T&&`是只绑定到右值还是可以绑定任意对象呢？"），并且，当你在和你的合作者交流时，它会帮助你避免歧义（"在这里我在用一个通用引用，而非右值引用")。它也可以帮助你弄懂Item 25和26,它们依赖于右值引用和通用引用的区别。所以，拥抱这份抽象，陶醉于它吧。就像牛顿的力学定律（本质上不正确），比起爱因斯坦的相对论（这是真相）而言，往往更简单，更易用。所以这份通用引用的概念，相较于穷究引用折叠的细节而言，是更合意之选。

记住：

- 如果一个函数模板参数的类型为`T&&`，并且`T`需要被推导得知，或者如果一个对象被声明为`auto&&`，这个参数或者对象就是一个通用引用。
- 如果类型声明的形式不是标准的`type&&`，或者如果类型推导没有发生，那么`type&&`代表一个右值引用。
- 通用引用，如果它被右值初始化，就会对应地成为右值引用;如果它被左值初始化，就会成为左值引用。

Item25: 对右值引用使用`std::move`，对通用引用使用`std::forward`

右值引用仅绑定可以移动的对象。如果你有一个右值引用参数，你就知道这个对象可能会被移动：

```
class Widget {
  Widget(Widget&& rhs); //rhs definitely refers to an object eligible for moving
  ...
};
```

这是个例子，你将希望通过可以利用该对象右值性的方式传递给其他使用对象的函数。这样做的方法是将绑定次类对象的参数转换为右值。如Item23中所述，这不仅是`std::move`所做，而且是为它创建：

```
class Widget {
public:
  Widget(Widget&& rhs) :name(std::move(rhs.name)), p(std::move(rhs.p)) {...}
  ...
private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

另一方面（查看Item24），通用引用可能绑定到有资格移动的对象上。通用引用使用右值初始化时，才将其强制转换为右值。Item23阐释了这正是`std::forward`所做的：

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName) {      //newName is universal reference
    name = std::forward<T>(newName);
  }
  ...
}
```

总而言之，当传递给函数时右值引用应该无条件转换为右值（通过`std::move`），通用引用应该有条件转换为右值（通过`std::forward`）。

Item23 解释说，可以在右值引用上使用`std::forward`表现出适当的行为，但是代码较长，容易出错，所以应该避免在右值引用上使用`std::forward`。更糟的是在通用引用上使用`std::move`，这可能会意外改变左值。

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName) {
    name = std::move(newName);  //universal reference compiles, but is bad ! bad ! bad !
  }
  ...

private:
```

```
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName(); // factory function

Widget w;
auto n = getWidgetName(); // n is local variiable
w.setName(n); // move n into w! n's value now unkown
```

上面的例子，局部变量n被传递给`w.setName`，可以调用方对n只有只读操作。但是因为`setName`内部使用`std::move`无条件将传递的参数转换为右值，`n`的值被移动给w，n最终变为未定义的值。这种行为使得调用者蒙圈了。

你可能争辩说`setName`不应该将其参数声明为通用引用。此类引用不能使用`const`（Item 24），但是`setName`肯定不应该修改其参数。你可能会指出，如果const左值和右值分别进行重载可以避免整个问题，比如这样：

```
class Widget {
public:
  void setName(const std::string& newName) { // set from const lvalue
    name = newName;
  }
  void setName(std::string&& newName) { // set from rvalue
    name = std::move(newName);
  }
};
```

这样的话，当然可以工作，但是有缺点。首先编写和维护的代码更多；其次，效率下降。比如，考虑如下场景：

```
w.setName("Adela Novak");
```

使用通用引用的版本，字面字符串"Adela Novak"可以被传递给setName，在w内部使用了`std::string`的赋值运算符。w的name的数据成员直接通过字面字符串直接赋值，没有中间对象被创建。但是，重载版本，会有一个中间对象被创建。一次setName的调用会包括`std::string`的构造器调用（中间对象），`std::string`的赋值运算调用，`std::string`的析构调用（中间对象）。这比直接通过const char*赋值给`std::string`开销昂贵许多。实际的开销可能因为库的实现而有所不同，但是事实上，将通用引用模板替换成多个函数重载在某些情况下会导致运行时的开销。如果例子中的`Widget`数据成员是任意类型（不一定是`std::string`），性能差距可能会变得更大，因为不是所有类型的移动操作都像`std::string`开销较小（参看Item29）。

但是，关于重载函数最重要的问题不是源代码的数量，也不是代码的运行时性能。而是设计的可扩展性差。`Widget::setName`接受一个参数，可以是左值或者右值，因此需要两种重载实现，`n`个参数的话，就要实现2^n种重载。这还不是最坏的。有的函数---函数模板----接受无限制参数，每个参数都可以是左值或者右值。此类函数的例子比如`std::make_unique`或者`std::make_shared`。查看他们的的重载声明：

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);

template<class T, class... Args>
unique_ptr<T> make_unique(Args&&... args);
```

对于这种函数，对于左值和右值分别重载就不能考虑了：通用引用是仅有的实现方案。对这种函数，我向你保证，肯定使用`std::forward`传递通用引用给其他函数。

好吧，通常，最终。但是不一定最开始就是如此。在某些情况，你可能需要在一个函数中多次使用绑定到右值引用或者通用引用的对象，并且确保在完成其他操作前，这个对象不会被移动。这时，你只想在最后一次使用时，使用`std::move`或者`std::forward`。比如：

```
template<typename T>
void setSignText(T&& text)
{
  sign.setText(text);

  auto now = std::chrono::system_clock::now();

  signHistory.add(now, std::forward<T>(text));
}
```

这里，我们想要确保`text`的值不会被`sign.setText`改变，因为我们想要在`signHistory.add`中继续使用。因此`std::forward`只在最后使用。

对于`std::move`，同样的思路，但是需要注意，在有些稀少的情况下，你需要调用`std::move_if_noexcept`代替`std::move`。要了解何时以及为什么，参考Item 14。

如果你使用的按值返回的函数，并且返回值绑定到右值引用或者通用引用上，需要对返回的引用使用`std::move`或者`std::forward`。要了解原因，考虑`+`操作两个矩阵的函数，左侧的矩阵参数为右值（可以被用来保存求值之后的和）

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs){
  lhs += rhs;
  return std::move(lhs); // move lhs into return value
}
```

通过在返回语句中将lhs转换为右值，lhs可以移动到返回值的内存位置。如果`std::move`省略了

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs){
  lhs += rhs;
  return lhs; // copy lhs into return value
}
```

事实上，lhs作为左值，会被编译器拷贝到返回值的内存空间。假定Matrix支持移动操作，并且比拷贝操作效率更高，使用`std::move`的代码效率更高。

如果Matrix不支持移动操作，将其转换为左值不会变差，因为右值可以直接被Matrix的拷贝构造器使用。如果Matrix随后支持了移动操作，`+`操作符的定义将在下一次编译时受益。就是这种情况，通过将`std::move`应用到返回语句中，不会损失什么，还可能获得收益。

使用通用引用和`std::forward`的情况类似。考虑函数模板`reduceAndCopy`收到一个未规约对象`Fraction`，将其规约，并返回一个副本。如果原始对象是右值，可以将其移动到返回值中，避免拷贝开销，但是如果原始对象是左值，必须创建副本，因此如下代码：

```
template<typename T>
Fraction reduceAndCopy(T&& frac) {
  frac.reduce();
  return std::forward<T>(frac); // move rvalue into return value, copy lvalue
}
```

如果`std::forward`被忽略，frac就是无条件复制到返回值内存空间。

有些开发者获取到上面的知识后，并尝试将其扩展到不适用的情况。

```
Widget makeWidget() {
  Widget w; //local variable
  ...   // configure w
  return w; // "copy" w into return value
}
```

想要优化copy的动作为如下代码:

```
Widget makeWidget() {
  Widget w; //local variable
  ...   // configure w
  return std::move(w); // move w into return value(don't do this!)
}
```

这种用法是有问题的，但是问题在哪?

在进行优化时，标准化委员会远领先于开发者，第一个版本的makeWidget可以在分配给函数返回值的内存中构造局部变量w来避免复制局部变量w的需要。这就是所谓的返回值优化（RVO），这在C++标准中已经实现了。

所以"copy"版本的makeWidget在编译时都避免了拷贝局部变量w，进行了返回值优化。（返回值优化的条件：1. 局部变量与返回值的类型相同；2. 局部变量就是返回值）。

移动版本的makeWidget行为与其名称一样，将w的内容移动到makeWidget的返回值位置。但是为什么编译器不使用RVO消除这种移动，而是在分配给函数返回值的内存中再次构造w呢？条件2中规定，仅当返回值为局部对象时，才进行RVO，但是move版本不满足这条件，再次看一下返回语句:

```
return std::move(w);
```

返回的已经不是局部对象w，而是局部对象w的引用。返回局部对象的引用不满足RVO的第二个条件，所以编译器必须移动w到函数返回值的位置。开发者试图帮助编译器优化反而限制了编译器的优化选项。

（译者注：本段即绕又长，大意为即使开发者非常熟悉编译器，坚持要在局部变量上使用`std::move`返回）

这仍然是一个坏主意。C++标准关于RVO的部分表明，如果满足RVO的条件，但是编译器选择不执行复制忽略，则必须将返回的对象视为右值。实际上，标准要求RVO，忽略复制或者将`sdt::move`隐式应用于返回的本地对象。因此，在makeWidget的"copy"版本中，编译器要不执行复制忽略的优化，要不自动将`std::move`隐式执行。

按值传递参数的情形与此类似。他们没有资格进行RVO，但是如果作为返回值的话编译器会将其视作右值。结果就是，如果代码如下:

```
Widget makeWidget(Widget w) {
  ...
  return w;
}
```

实际上，编译器的代码如下:

```
Widget makeWidget(Widget w){
  ...
  return std::move(w);
}
```

这意味着，如果对从按值返回局部对象的函数使用`std::move`，你并不能帮助编译器，而是阻碍其执行优化选项。在某些情况下，将`std::move`应用于局部变量可能是一件合理的事，但是不要阻碍编译器RVO。

需要记住的点

- 在右值引用上使用`std::move`，在通用引用上使用`std::forward`
- 对按值返回的函数返回值，无论返回右值引用还是通用引用，执行相同的操作
- 当局部变量就是返回值是，不要使用`std::move`或者`std::forward`

Item 26: Avoid overloading on universal references

Item 26: 避免在通用引用上重载

假定你需要写一个函数，它使用name这样一个参数，打印当前日期和具体时间到日志中，然后将name加入到一个全局数据结构中。你可能写出来这样的代码：

```
std::multiset<std::string> names; // global data structure
void logAndAdd(const std::string& name)
{
  auto now = std::chrono::system_lock::now(); // get current time
  log(now, "logAndAdd"); // make log entry
  names.emplace(name); // add name to global data structure; see Item 42 for info
on emplace
}
```

这份代码没有问题，但是同样的也没有效率。考虑这三个调用：

```
std::string petName("Darla");
logAndAdd(petName); // pass lvalue std::string
logAndAdd(std::string("Persephone")); // pass rvalue std::string
logAndAdd("Patty Dog"); // pass string literal
```

在第一个调用中，`logAndAdd`使用变量作为参数。在`logAndAdd`中`name`最终也是通过`emplace`传递给`names`。因为`name`是左值，会拷贝到`names`中。没有方法避免拷贝，因为是左值传递的。

在第三个调用中，参数`name`绑定一个右值，但是这次是通过"Patty Dog"隐式创建的临时`std::string`变量。在第二个调用总，`name`被拷贝到`names`，但是这里，传递的是一个字符串字面量。直接将字符串字面量传递给`emplace`，不会创建`std::string`的临时变量，而是直接在`std::multiset`中通过字面量构建`std::string`。在第三个调用中，我们会消耗`std::string`的拷贝开销，但是连移动开销都不想有，更别说拷贝的。

我们可以通过使用通用引用（参见Item 24）重写第二个和第三个调用来使效率提升，按照Item 25的说法，`std::forward`转发引用到`emplace`。代码如下：

```
template<typename T>
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_lock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
std::string petName("Darla"); // as before
logAndAdd(petName); // as before , copy
logAndAdd(std::string("Persephone")); // move rvalue instead of copying it
logAndAdd("Patty Dog"); // create std::string in multiset instead of copying a
temporary std::string
```

非常好，效率优化了！

在故事的最后，我们可以骄傲的交付这个代码，但是我没有告诉你client不总是有访问`logAndAdd`要求的`names`的权限。有些clients只有`names`的下标。为了支持这种client，`logAndAdd`需要重载为：

```
std::string nameFromIdx(int idx); // return name corresponding to idx
void logAndAdd(int idx)
{
  auto now = std::chrono::system_lock::now();
  log(now, "logAndAdd");
  names.emplace(nameFromIdx(idx));
}
```

之后的两个调用按照预期工作：

```
std::string petName("Darla");
logAndAdd(petName);
logAndAdd(std::string("Persephone"));
logAndAdd("Patty Dog"); // these calls all invoke the T&& overload

logAndAdd(22); // calls int overload
```

事实上，这只能基本按照预期工作，假定一个client将`short`类型当做下标传递给`logAndAdd`：

```
short nameIdx;
...
logAndAdd(nameIdx); // error!
```

之后一行的error说明并不清楚，下面让我来说明发生了什么。

有两个重载的`logAndAdd`。一个使用通用应用推导出T的类型是`short`，因此可以精确匹配。对于`int`参数类型的重载`logAndAdd`也可以`short`类型提升后匹配成功。根据正常的重载解决规则，精确匹配优先于类型提升的匹配，所以被调用的是通用引用的重载。

在通用引用中的实现中，将`short`类型`emplace`到`std::string`的容器中，发生了错误。所有这一切的原因就是对于`short`类型通用引用重载优先于`int`类型的重载。

使用通用引用类型的函数在C++中是贪婪函数。他们机会可以精确匹配任何类型的参数（极少不适用的类型在Item 30中介绍）。这也是组合重载和通用引用使用是糟糕主意的原因：通用引用的实现会匹配比开发者预期要多得多的参数类型。

一个更容易调入这种陷阱的例子是完美转发构造函数。简单对`logAndAdd`例子进行改造就可以说明这个问题。将使用`std::string`类型改为自定义`Person`类型即可：

```
class Person
{
public:
  template<typename T>
  explicit Person(T&& n) :name(std::forward<T>(n)) {} // perfect forwarding ctor;
initializes data member
  explicit Person(int idx): name(nameFromIdx(idx)) {}
  ...
private:
  std::string name;
};
```

在`logAndAdd`的例子中，传递一个不是int的整型变量（比如`std::size_t, short, long`等）会调用通用引用的构造函数而不是int的构造函数，这会导致编译错误。这里这个问题甚至更糟糕，因为`Person`中存在的重载比肉眼看到的更多。在Item 17中说明，在适当的条件下，C++会生成拷贝和移动构造函数，即使类包含了模板构造也在合适的条件范围内。如果拷贝和移动构造被生成，Person类看起

来就像这样：

```
class Person
{
public:
  template<typename T>
  explicit Person(T&& n) :name(std::forward<T>(n)) {} // perfect forwarding ctor
  explicit Person(int idx); // int ctor

  Person(const Person& rhs); // copy ctor(complier-generated)
  Person(Person&& rhs); // move ctor (compiler-generated)
  ...
};
```

只有你在花了很多时间在编译器领域时，下面的行为才变得直观（译者注：这里意思就是这种实现会导致不符合人类直觉的结果，下面就解释了这种现象的原因）

```
Person p("Nancy");
auto cloneOfP(p); // create new Person from p; this won't compile!
```

这里我们视图通过一个`Person`实例创建另一个`Person`，显然应该调用拷贝构造即可（p是左值，我们可以思考通过移动操作来消除拷贝的开销）。但是这份代码不是调用拷贝构造，而是调用完美转发构造。然后，该函数将尝试使用Person对象p初始化`Person`的`std::string`的数据成员，编译器就会报错。

“为什么？”你可能会疑问，“为什么拷贝构造会被完美转发构造替代？我们显然想拷贝Person到另一个Person”。确实我们是这样想的，但是编译器严格遵循C++的规则，这里的相关规则就是控制对重载函数调用的解析规则。

编译器的理由如下：`cloneOfP`被`non-const`左值p初始化，这意味着可以实例化模板构造函数为采用`Person`的`non-const`左值。实例化之后，`Person`类看起来是这样的：

```
class Person {
public:
  explicit Person(Person& n)             // instantiated from
    : name(std::forward<Person&>(n)) {}  // perfect-forwarding

// template
  explicit Person(int idx);              // as before
  Person(const Person& rhs); // copy ctor (compiler-generated)
  ...
};
```

在`auto cloneOfP(p);`语句中，p被传递给拷贝构造或者完美转发构造。调用拷贝构造要求在p前加上const的约束，而调用完美转发构造不需要任何条件，所以编译器按照规则：采用最佳匹配，这里调用了完美转发的实例化的构造函数。

如果我们将本例中的传递的参数改为const的，会得到完全不同的结果：

```
const Person cp("Nancy");
auto cloneOfP(cp); // call copy constructor!
```

因为被拷贝的对象是const，是拷贝构造函数的精确匹配。虽然模板参数可以实例化为完全一样的函数签名：

```
class Person {
public:
  explicit Person(const Person& n); // instantiated from template

  Person(const Person& rhs); // copy ctor(compiler-generated)
  ...
};
```

但是无所谓，因为重载规则规定当模板实例化函数和非模板函数（或者称为“正常”函数）匹配优先级相当时，优先使用“正常”函数。拷贝构造函数（正常函数）因此胜过具有相同签名的模板实例化函数。

（如果你想知道为什么编译器在生成一个拷贝构造函数时还会模板实例化一个相同签名的函数，参考Item17）

当继承纳入考虑范围时，完美转发的构造函数与编译器生成的拷贝、移动操作之间的交互会更加复杂。尤其是，派生类的拷贝和移动操作会表现的非常奇怪。来看一下：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs) :Person(rhs)
  {...} // copy ctor; calls base class forwarding ctor!
  SpecialPerson(SpecialPerson&& rhs): Person(std::move(rhs))
  {...} // move ctor; calls base class forwarding ctor!
};
```

如同注释表示的，派生类的拷贝和移动构造函数没有调用基类的拷贝和移动构造函数，而是调用了基类的完美转发构造函数！为了理解原因，要知道派生类使用`SpecialPerson`作为参数传递给其基类，然后通过模板实例化和重载解析规则作用于基类。最终，代码无法编译，因为`std::string`没有`SpecialPerson`的构造函数。

我希望到目前为止，已经说服了你，如果可能的话，避免对通用引用的函数进行重载。但是，如果在通用引用上重载是糟糕的主意，那么如果需要可转发大多数类型的参数，但是对于某些类型又要特殊处理应该怎么办？存在多种办法。实际上，下一个Item，Item27专门来讨论这个问题，敬请阅读。

需要记住的事

- 对通用引用参数的函数进行重载，调用机会会比你期望的多得多
- 完美转发构造函数是糟糕的实现，因为对于`non-const`左值不会调用拷贝构造而是完美转发构造，而且会劫持派生类对于基类的拷贝和移动构造

Item 27: Familiarize yourself with alternatives to overloading on universal references

Item27:熟悉通用引用重载的替代方法

Item 26中说明了对使用通用引用参数的函数，无论是独立函数还是成员函数（尤其是构造函数），进行重载都会导致一系列问题。但是也提供了一些示例，如果能够按照我们期望的方式运行，重载可能也是有用的。这个Item探讨了几种通过避免在通用引用上重载的设计或者通过限制通用引用可以匹配的参数类型的方式来实现所需行为的方案。

讨论基于Item 26中的示例，如果你还没有阅读Item 26，请先阅读在继续本Item的阅读。

Abandon overloading

在Item 26中的第一个例子中，`logAndAdd`代表了许多函数，这些函数可以使用不同的名字来避免在通用引用上的重载的弊端。例如两个重载的`logAndAdd`函数，可以分别改名为`logAndAddName`和`logAndAddNameIdx`。但是，这种方式不能用在第二个例子，Person构造函数中，因为构造函数的名字本类名固定了。此外谁愿意放弃重载呢?

Pass by const T&

一种替代方案是退回到C++98，然后将通用引用替换为const的左值引用。事实上，这是Item 26中首先考虑的方法。缺点是效率不高，会有拷贝的开销。现在我们知道了通用引用和重载的组合会导致问题，所以放弃一些效率来确保行为正确简单可能也是一种不错的折中。

Pass by value

通常在不增加复杂性的情况下提高性能的一种方法是，将按引用传递参数替换为按值传递，这是违反直觉的。该设计遵循Item 41中给出的建议，即在你知道要拷贝时就按值传递，因此会参考Item 41来详细讨论如何设计与工作，效率如何。这里，在Person的例子中展示:

```
class Person {
public:
  explicit Person(std::string p) // replace T&& ctor; see
  : name(std::move(n)) {} // Item 41 for use of std::move

  explicit Person(int idx)
  : name(nameFromIdx(idx)) {}
  ...
private:
  std::string name;
};
```

因为没有`std::string`构造器可以接受整型参数，所有`int`或者其他整型变量（比如`std::size_t`、`short`、`long`等）都会使用`int`类型重载的构造函数。相似的，所有`std::string`类似的参数（字面量等）都会使用`std::string`类型的重载构造函数。没有意外情况。我想你可能会说有些人想要使用0或者NULL会调用`int`重载的构造函数，但是这些人应该参考Item 8反复阅读指导使用0或者NULL作为空指针让他们恶心。

Use Tag dispatch

传递`const`左值引用参数以及按值传递都不支持完美转发。如果使用通用引用的动机是完美转发，我们就只能使用通用引用了，没有其他选择。但是又不想放弃重载。所以如果不放弃重载又不放弃通用引用，如何避免咋通用引用上重载呢?

实际上并不难。通过查看重载的所有参数以及调用的传入参数，然后选择最优匹配的函数----计算所有参数和变量的组合。通用引用通常提供了最优匹配，但是如果通用引用是包含其他非通用引用参数列表的一部分，则不是通用引用的部分会影响整体。这基本就是tag dispatch 方法，下面的示例会使这段话更容易理解。

我们将tag dispatch应用于`logAndAdd`例子，下面是原来的代码，以免你找不到Item 26的代码位置:

```
std::multiset<std::string> names; // global data structure
template<typename T> // make log entry and add
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_clokc::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

就其本身而言，功能执行没有问题，但是如果引入一个`int`类型的重载，就会重新陷入Item 26中描述的麻烦。这个Item的目标是避免它。不通过重载，我们重新实现`logAndAdd`函数分拆为两个函数，一个针对整型值，一个针对其他。`logAndAdd`本身接受所有的类型。

这两个真正执行逻辑的函数命名为`logAndAddImpl`使用重载。一个函数接受通用引用参数。所以我们同时使用了重载和通用引用。但是每个函数接受第二个参数，表征传入的参数是否为整型。这第二个参数可以帮助我们避免陷入到Item 26中提到的麻烦中，因为我们将其安排为第二个参数决定选择哪个重载函数。

是的，我知道，“不要在啰嗦了，赶紧亮出代码”。没有问题，代码如下，这是最接近正确版本的:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_integral<T>()); // not quite correct
}
```

这个函数转发它的参数给`logAndAddImpl`函数，但是多传递了一个表示是否T为整型的变量。至少，这就是应该做的。对于右值的整型参数来说，这也是正确的。但是如同Item 28中说明，如果左值参数传递给通用引用`name`，类型推断会使左值引用。所以如果左值int被传入`logAndAdd`，T将被推断为`int&`。这不是一个整型类型，因为引用不是整型类型。这意味着`std::is_integral<T>`对于左值参数返回false，即使确实传入了整型值。

意识到这个问题基本相当于解决了它，因为C++标准库有一个类型trait（参见Item 9），`std::remove_reference`，函数名字就说明做了我们希望的：移除引用。所以正确实现的代码应该是这样:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_instegral<typename std::remove_reference<T>::type>());
}
```

这个代码很巧妙。（在C++14中，你可以通过`std::remove_reference_t<T>`来简化写法，参看Item 9）

处理完之后，我们可以将注意力转移到名为`logAndAddImpl`的函数上了。有两个重载函数，第一个仅用于非整型类型（即`std::is_instegral<typename std::remove_reference<T>::type>()`是`false`）：

```
template<typename T>
void logAndAddImpl(T&& name, std::false_type) // 高亮为std::false_type
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

一旦你理解了高亮参数的含义代码就很直观。概念上，`logAndAdd`传递一个布尔值给`logAndAddImpl`表明是否传入了一个整型类型，但是`true`和`false`是运行时值，我们需要使用编译时决策来选择正确的`logAndAddImpl`重载。这意味着我们需要一个类型对应`true`，`false`同理。这个需要是经常出现的，所以标准库提供了这样两个命名`std::true_type and std::false_type`。`logAndAdd`传递给`logAndAddImpl`的参数类型取决于T是否整型，如果T是整型，它的类型就继承自`std::true_type`，反之继承自`std::false_type`。最终的结果就是，当T不是整型类型时，这个`logAndAddImpl`重载会被调用。

第二个重载覆盖了相反的场景：当T是整型类型。在这个场景中，`logAndAddImpl`简单找到下标处的`name`，然后传递给`logAndAdd`：

```
std::string nameFromIdx(int idx); // as in item 26
void logAndAddImpl(int idx, std::true_type) // 高亮: std::true_type
{
  logAndAdd(nameFromIdx(idx));
}
```

通过下标找到对应的`name`，然后让`logAndAddImpl`传递给`logAndAdd`，我们避免了将日志代码放入这个`logAndAddImpl`重载中。

在这个设计中，类型`std::true_type`和`std::false_type`是“标签”，其唯一目的就是强制重载解析按照我们的想法来执行。注意到我们甚至没有对这些参数进行命名。他们在运行时毫无用处，事实上我们希望编译器可以意识到这些tag参数是无用的然后在程序执行时优化掉它们（至少某些时候有些编译器会这样做）。这种在`logAndAdd`内部的通过tag来实现重载实现函数的“分发”，因此这个设计名称为：**tag dispatch**。这是模板元编程的标准构建模块，你对现代C++库中的代码了解越多，你就会越多遇到这种设计。

就我们的目的而言，tag dispatch的重要之处在于它可以允许我们组合重载和通用引用使用，而没有Item 26中提到的问题。分发函数---`logAndAdd`----接受一个没有约束的通用引用参数，但是这个函数没有重载。实现函数---`logAndAddImpl`----是重载的，一个接受通用引用参数，但是重载规则不仅依赖通用引用参数，还依赖新引入的tag参数。结果是tag来决定采用哪个重载函数。通用引用参数可以生成精

确匹配的事实在这里并不重要。（译者注：这里确实比较啰嗦，如果理解了上面的内容，这段完全可以没有。）

Constraining templates that take universal references（约束使用通用引用的模板）

tag dispatch的关键是存在单独一个函数（没有重载）给客户端API。这个单独的函数分发给具体的实现函数。创建一个没有重载的分发函数通常是容易的，但是Item 26中所述第二个问题案例是`Person`类的完美转发构造函数，是个例外。编译器可能会自行生成拷贝和移动构造函数，所以即使你只写了一个构造函数并在其中使用tag dispatch，编译器生成的构造函数也打破了你的期望。

实际上，真正的问题不是编译器生成的函数会绕过tag diapatch设计，而是不总会绕过tag dispatch。你希望类的拷贝构造总是处理该类型的`non-const`左值构造请求，但是如同Item 26中所述，提供具有通用引用的构造函数会使通用引用构造函数被调用而不是拷贝构造函数。还说明了当一个基类声明了完美转发构造函数，派生类实现自己的拷贝和移动构造函数时会发生错误的调用（调用基类的完美转发构造函数而不是基类的拷贝或者移动构造）

这种情况，采用通用引用的重载函数通常比期望的更加贪心，但是有不满足使用tag dispatch的条件。你需要不同的技术，可以让你确定允许使用通用引用模板的条件。朋友你需要的就是`std::enable_if`。

`std::enable_if`可以给你提供一种强制编译器执行行为的方法，即使特定模板不存在。这种模板也会被禁止。默认情况下，所有模板是启用的，但是使用`std::enable_if`可以使得仅在条件满足时模板才启用。在这个例子中，我们只在传递的参数类型不是`Person`使用`Person`的完美转发构造函数。如果传递的参数是`Person`，我们要禁止完美转发构造函数（即让编译器忽略它），因此就是拷贝或者移动构造函数处理，这就是我们想要使用`Person`初始化另一个`Person`的初衷。

这个主意听起来并不难，但是语法比较繁杂，尤其是之前没有接触过的话，让我慢慢引导你。有一些使用`std::enbale_if`的样板，让我们从这里开始。下面的代码是`Person`完美转发构造函数的声明，我仅展示声明，因为实现部分跟Item 26中没有区别。

```
class Person {
public:
  template<typename T,
                typename = typename std::enable_if<condition>::type> // 本行高亮
  explicit Person(T&& n);
  ...
};
```

为了理解高亮部分发生了什么，我很遗憾的表示你要自行查询语法含义，因为详细解释需要花费一定空间和时间，而本书并没有足够的空间（在你自行学习过程中，请研究"SFINAE"以及`std::enable_if`，因为“SFINAE”就是使`std::enable_if`起作用的技术）。这里我想要集中讨论条件的表示，该条件表示此构造函数是否启用。

这里我们想表示的条件是确认T不是`Person`类型，即模板构造函数应该在T不是`Person`类型的时候启用。因为type trait可以确定两个对象类型是否相同（`std::is_same`），看起来我们需要的就是`!std::is_same<Person, T>::value`（注意语句开始的！，我们想要的是不相同）。这很接近我们想要的了，但是不完全正确，因为如同Item 28中所述，对于通用引用的类型推导，如果是左值的话会推导成左值引用，比如这个代码：

```
Person p("Nancy");
auto cloneOfP(p); // initialize from lvalue
```

T的类型在通用引用的构造函数中被推导为`Person&`。`Person`和`Person&`类型是不同的，`std::is_same`对比`std::is_same<Person, Person&>::value`会是`false`。

如果我们更精细考虑仅当T不是`Person`类型才启用模板构造函数，我们会意识到当我们查看T时，应该忽略：

- **是否引用**。对于决定是否通用引用构造器启用的目的来说，`Person, Person&, Person&&`都是跟`Person`一样的。
- **是不是`const`或者`volatile`**。如上所述，`const Person , volatile Person , const volatile Person`也是跟`Person`一样的。

这意味着我们需要一种方法消除对于`T`的`引用, const, volatile`修饰。再次，标准库提供了这样的功能type trait，就是`std::decay`。`std::decay<T>::value`与`T`是相同的，只不过会移除`引用, const, volatile`的修饰。（这里我没有说出另外的真相，`std::decay`如同其名一样，可以将array或者function退化成指针，参考Item 1，但是在这里讨论的问题中，它刚好合适）。我们想要控制构造器是否启用的条件可以写成：

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

表示`Person`与`T`的类型不同。

将其带回整体代码中，`Person`的完美转发构造函数的声明如下：

```
class Person {
public:
  template<typename T,
                typename = typename std::enable_if<
             !std::is_same<Person, typename std::decay<T>::type>::value
             >::type> // 本行高亮
  explicit Person(T&& n);
  ...
};
```

如果你之前从没有看到过这种类型的代码，那你可太幸福了。最后是这种设计是有原因的。当你使用其他机制来避免同时使用重载和通用引用时（你总会这样做），确实应该那样做。不过，一旦你习惯了使用函数语法和尖括号的使用，也不坏。此外，这可以提供你一直想要的行为表现。在上面的声明中，使用`Person`初始化一个`Person`----无论是左值还是右值，`const`还是`volatile`都不会调用到通用引用构造函数。

成功了，对吗？确实！

当然没有。等会再庆祝。Item 26还有一个情景需要解决，我们需要继续探讨下去。

假定从`Person`派生的类以常规方式实现拷贝和移动操作：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs): Person(rhs)
  {...} // copy ctor; calls base class forwarding ctor!
  SpecialPerson(SpecialPerson&& rhs): Person(std::move(rhs))
  {...} // move ctor; calls base class forwarding ctor!
};
```

这和Item 26中的代码是一样的，包括注释也是一样。当我们拷贝或者移动一个`SpecialPerson`对象时，我们希望调用基类对应的拷贝和移动构造函数，但是这里，我们将`SpecialPerson`传递给基类的构造器，因为`SpecialPerson`和`Person`类型不同，所以完美转发构造函数是启用的，会实例化为精确匹配的构造函数。生成的精确匹配的构造函数之于重载规则比基类的拷贝或者移动构造函数更优，所以这里的代码，拷贝或者移动`SpecialPerson`对象就会调用`Person`类的完美转发构造函数来执行基类的部分。跟Item 26的困境一样。

派生类仅仅是按照常规的规则生成了自己的移动和拷贝构造函数，所以这个问题的解决还要落实在在基类，尤其是控制是否使用`Person`通用引用构造函数启用的条件。现在我们意识到不只是禁止`Person`类型启用模板构造器，而是禁止`Person`以及任何派生自`Person`的类型启用模板构造器。讨厌的继承！

你应该不意外在这里看到标准库中也有type trait判断一个类型是否继承自另一个类型，就是`std::is_base_of`。如果`std::is_base_of<T1, T2>`是`true`表示`T2`派生自`T1`。类型系统是自派生的，表示`std::is_base_of<T, T>::value`总是`true`。这就很方便了，我们想要修正关于我们控制`Person`完美转发构造器的启用条件，只有当`T`在消除引用，`const`，`volatile`修饰之后，并且既不是`Person`又不是`Person`的派生类，才满足条件。所以使用`std::is_base_of`代替`std::is_same`就可以了：

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_base_if<Person,
                                                                  typename
std::decay<T>::type
                                  >::value
                >::type
  >
  explicit Person(T&& n);
  ...
};
```

现在我们终于完成了最终版本。这是C++11版本的代码，如果我们使用C++14，这份代码也可以工作，但是有更简洁一些的写法如下：

```
class Person  { // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<   // less code here
                        !std::is_base_of<Person,
                                 std::decay_t<T> // and here
                                >::value
                > // and here
  >
  explicit Person(T&& n);
  ...
};
```

好了，我承认，我又撒谎了。我们还没有完成，但是越发接近最终版本了。非常接近，我保证。

我们已经知道如何使用`std::enable_if`来选择性禁止`Person`通用引用构造器来使得一些参数确保使用到拷贝或者移动构造器，但是我们还是不知道将其应用于区分整型参数和非整型参数。毕竟，我们的原始目标是解决构造函数模糊性问题。

我们需要的工具都介绍过了，我保证都介绍了，
（1）加入一个`Person`构造函数重载来处理整型参数
（2）约束模板构造器使其对于某些参数禁止
使用这些我们讨论过的技术组合起来，就能解决这个问题了：

```
class Person { // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<
     !std::is_base_of<Person, std::decay_t<T>::value
       &&
       !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n): name(std::forward<T>(n))
  {...} // ctor for std::strings and args convertible to strings

  explicit Person(int idx): name(nameFromIdx(idx))
  {...} // ctor for integral args

  ... // copy and move ctors, etc
private:
  std::string name;
};
```

看！多么优美！好吧，优美之处只是对于那些迷信模板元编程之人，但是事实却是提出了不仅能工作的方法，而且极具技巧。因为使用了完美转发，所以具有最大效率，因为控制了使用通用引用的范围，可以避免对于大多数参数能实例化精确匹配的滥用问题。

Trade-offs （权衡，折中）

本Item提到的前三个技术---abandoning overloading, passing by const T&, passing by value---在函数调用中指定每个参数的类型。后两个技术----tag dispatch和 constraing template eligibility----使用完美转发，因此不需要指定参数类型。这一基本决定（是否指定类型）有一定后果。

通常，完美转发更有效率，因为它避免了仅处于符合参数类型而创建临时对象。在`Person`构造函数的例子中，完美转发允许将`Nancy`这种字符串字面量转发到容器内部的`std::string`构造器，不使用完美转发的技术则会创建一个临时对象来满足传入的参数类型。

但是完美转发也有缺点。·即使某些类型的参数可以传递给特定类型的参数的函数，也无法完美转发。Item 30中探索了这方面的例子。

第二个问题是当client传递无效参数时错误消息的可理解性。例如假如创建一个`Person`对象的client传递了一个由`char16_t`（一种C++11引入的类型表示16位字符）而不是`char`（`std::string`包含的）：

```
Person p(u"Konrad Zuse"); // "Konrad Zuse" consists of characters of type const
char16_t
```

使用本Item中讨论的前三种方法，编译器将看到可用的采用`int`或者`std::string`的构造函数，并且它们或多或少会产生错误消息，表示没有可以从`const char16_t`转换为`int`或者`std::string`的方法。

但是，基于完美转发的方法，`const char16_t`不受约束地绑定到构造函数的参数。从那里将转发到`Person`的`std::string`的构造函数，在这里，调用者传入的内容(`const char16_t`数组)与所需内容(`std::string`构造器可接受的类型)发生的不匹配会被发现。由此产生的错误消息会让人更容易理解，在我使用的编译器上，会产生超过160行错误信息。

在这个例子中，通用引用仅被转发一次（从`Person`构造器到`std::string`构造器），但是更复杂的系统中，在最终通用引用到达最终判断是否可接受的函数之前会有多层函数调用。通用引用被转发的次数越多，产生的错误消息偏差就越大。许多开发者发现仅此问题就是在性能优先的接口使用通用引用的障碍。（译者注：最后一句话可能翻译有误，待确认）

在`Person`这个例子中，我们知道转发函数的通用引用参数要支持`std::string`的初始化，所以我们可以用`static_assert`来确认是不是支持。`std::is_constructible` type trait执行编译时测试一个类型的对象是否可以构造另一个不同类型的对象，所以代码可以这样：

```
class Person {
public:
  template<typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
        !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n) :name(std::forward<T>(n))
  {
    //assert that a std::string can be created from a T object(这里到...高亮)
    static_assert(
        std::is_constructible<std::string, T>::value,
      "Parameter n can't be used to construct a std::string"
    );
    ... // the usual ctor work goes here
  }
  ... // remainder of Person class (as before)
};
```

如果client代码尝试使用无法构造`std::string`的类型创建`Person`，会导致指定的错误消息。不幸的是，在这个例子中，`static_assert`在构造函数体中，但是作为成员初始化列表的部分在检查之前。所以我使用的编译器，结果是由`static_assert`产生的清晰的错误消息在常规错误消息（最多160行以上那个）后出现。

需要记住的事

- 通用引用和重载的组合替代方案包括使用不同的函数名，通过const左值引用传参，按值传递参数，使用tag dispatch
- 通过`std::enable_if`约束模板，允许组合通用引用和重载使用，`std::enable_if`可以控制编译器哪种条件才使用通用引用的实例
- 通用引用参数通常具有高效率的优势，但是可用性就值得斟酌

Item28：理解引用折叠

Item23中指出，当参数传递给模板函数时，模板参数的类型是左值还是右值被推导出来。但是并没有提到只有当参数被声明为通用引用时，上述推导才会发生，但是有充分的理由忽略这一点：因为通用引用是Item24中才提到。回过头来看，通用引用和左值/右值编码意味着：

```
template<typename T>
void func(T&& param);
```

被推导的模板参数T将根据被传入参数类型被编码为左值或者右值。

编码机制是简单的。当左值被传入时，T被推导为左值。当右值被传入时，T被推导为非引用（请注意不对称性：左值被编码为左值引用，右值被编码为非引用），因此：

```
Widget widgetFactory(); // function returning rvalue
Widget w; // a variable(an lvalue)
func(w);  // call func with lvalue; T deduced to be Widget&
func(widgetFactory()); // call func with rvalue; T deduced to be Widget
```

上面的两种调用中，Widget被传入，因为一个是左值，一个是右值，模板参数T被推导为不同的类型。正如我们很快看到的，这决定了通用引用成为左值还是右值，也是`std::forward`的工作基础。

在我们更加深入`std::forward`和通用引用之前，必须明确在C++中引用的引用是非法的。不知道你是否尝试过下面的写法，编译器会报错：

```
int x;
...
auto& & rx = x;  //error! can't declare reference to reference
```

考虑下，如果一个左值传给模板函数的通用引用会发生什么：

```
template<typename T>
void func(T&& param);

func(w); // invoke func with lvalue; T deduced as Widget&
```

如果我们把推导出来的类型带入回代码中看起来就像是这样：

```
void func(Widget& && param);
```

引用的引用！但是编译器没有报错。我们从Item24中了解到因为通用引用param被传入一个左值，所以param的类型被推导为左值引用，但是编译器如何采用T的推导类型的结果，这是最终的函数签名？

```
void func(Widget& param);
```

答案是引用折叠。是的，禁止你声明引用的引用，但是编译器会在特定的上下文中使用，包括模板实例的例子。当编译器生成引用的引用时，引用折叠指导下一步发生什么。

存在两种类型的引用（左值和右值），所以有四种可能的引用组合（左值的左值，左值的右值，右值的右值，右值的左值）。如果一个上下文中允许引用的引用存在（比如，模板函数的实例化），引用根据规则折叠为单个引用：

> 如果任一引用为左值引用，则结果为左值引用。否则（即，如果引用都是右值引用），结果为右值引用

在我们上面的例子中，将推导类型Widget&替换模板func会产生对左值引用的右值引用，然后引用折叠规则告诉我们结果就是左值引用。

引用折叠是`std::forward`工作的一种关键机制。就像Item25中解释的一样，`std::forward`应用在通用引用参数上，所以经常能看到这样使用：

```
template<typename T>
void f(T&& fParam)
{
  ...  // do some work
  someFunc(std::forward<T>(fParam)); // forward fParam to someFunc
}
```

因为fParam是通用引用，我们知道参数T的类型将在传入具体参数时被编码。`std::forward`的作用是当传入参数为右值时，即T为非引用类型，才将fParam（左值）转化为一个右值。

`std::forward`可以这样实现：

```
template<typename T>
T&& forward(typename remove_reference<T>::type& param)
{
  return static_cast<T&&>(param);
}
```

这不是标准库版本的实现（忽略了一些接口描述），但是为了理解`std::forward`的行为，这些差异无关紧要。

假设传入到f的Widget的左值类型。T被推导为Widget&，然后调用`std::forward`将初始化为`std::forward<Widget&>`。带入到上面的`std::forward`的实现中：

```
Widget& && forward(typename remove_reference<Widget&>::type& param)
{
  return static_cast<Widget& &&>(param);
}
```

`std::remove_reference<Widget&>::type`表示Widget（查看Item9），所以`std::forward`成为：

```
Widget& && forward(Widget& param)
{
  return static_cast<Widget& &&>(param);
}
```

根据引用折叠规则，返回值和static_cast可以化简，最终版本的`std::forward`就是

```
Widget& forward(Widget& param)
{
  return static_cast<Widget&>(param);
}
```

正如你所看到的，当左值被传入到函数模板f时，`std::forward`转发和返回的都是左值引用。内部的转换不做任何事，因为param的类型已经是`Widget&`，所以转换没有影响。左值传入会返回左值引用。通过定义，左值引用就是左值，因此将左值传递给`std::forward`会返回左值，就像说的那样，完美转发。

现在假设一下，传递给f的是一个`Widget`的右值。在这个例子中，T的类型推导就是Widget。内部的`std::forward`因此转发`std::forward<Widget>`，带入回`std::forward`实现中：

```
Widget&& forward(typename remove_reference<Widget>::type& param)
{
  return static_cast<Widget&&>(param);
}
```

将`remove_reference`引用到非引用的类型上还是相同的类型，所以化简如下

```
Widget&& forward(Widget& param)
{
  return static_cast<Widget&&>(param);
}
```

这里没有引用的引用，所以不需要引用折叠，这就是最终版本。

从函数返回的右值引用被定义为右值，因此在这种情况下，`std::forward`会将f的参数fParam（左值）转换为右值。最终结果是，传递给f的右值参数将作为右值转发给someFunc，完美转发。

在C++14中，`std::remove_reference_t`的存在使得实现变得更简单：

```
template<typename T>  // C++ 14; still in namepsace std
T&& forward(remove_reference_t<T>& param)
{
  return static_cast<T&&>(param);
}
```

引用折叠发生在四种情况下。**第一**，也是最常见的就是模板实例化。**第二**，是auto变量的类型生成，具体细节类似模板实例化的分析，因为类型推导基本与模板实例化雷同（参见Item2）。考虑下面的例子：

```
template<typename T>
void func(T&& param);
Widget widgetFactory(); // function returning rvalue
Widget w; // a variable(an lvalue)
func(w); // call func with lvalue; T deduced to be Widget&
func(widgetFactory()); // call func with rvalue; T deduced to be Widget
```

在auto的写法中，规则是类似的：`auto&& w1 = w;`初始化`w1`为一个左值，因此为auto推导出类型`Widget&`。带回去就是`Widget& && w1 = w`，应用引用折叠规则，就是`Widget& w1 = w`,结果就是`w1`是一个左值引用。

另一方面，`auto&& w2 = widgetFactory();`使用右值初始化`w2`，非引用带回`Widget&& w2 = widgetFactory()`。没有引用的引用，这就是最终结果。

现在我们真正理解了Item24中引入的通用引用。通用引用不是一种新的引用，它实际上是满足两个条件下的右值引用：

- **通过类型推导将左值和右值区分**。T类型的左值被推导为&类型，T类型的右值被推导为T

- **引用折叠的发生**

通用引用的概念是有用的，因为它使你不必一定意识到引用折叠的存在，从直觉上判断左值和右值的推导即可。

我说了有四种情况会发生引用折叠，但是只讨论了两种：模板实例化和auto的类型生成。**第**三，是使用typedef和别名声明（参见Item9），如果，在创建或者定义typedef过程中出现了引用的引用，则引用折叠就会起作用。举例子来说，假设我们有一个Widget的类模板，该模板具有右值引用类型的嵌入式typedef：

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
  ...
};
```

假设我们使用左值引用实例化Widget：

```
Widget<int&> w;
```

就会出现

```
typedef int& && RvalueRefToT;
```

引用折叠就会发挥作用：

```
typedef int& RvalueRefToT;
```

这清楚表明我们为typedef选择的name可能不是我们希望的那样：RvalueRefToT是左值引用的typedef，当使用Widget被左值引用实例化时。

最后，**也是第四**种情况是，decltype使用的情况，如果在分析decltype期间，出现了引用的引用，引用折叠规则就会起作用（关于decltype，参见Item3）

需要记住的事

- 引用折叠发生在四种情况：模板实例化；auto类型推导；typedef的创建和别名声明；decltype
- 当编译器生成了引用的引用时，结果通过引用折叠就是单个引用。有左值引用就是左值引用，否则就是右值引用
- 通用引用就是通过类型推导区分左值还是右值，并且引用折叠出现的右值引用

Item29: Assume that move operations are not present, not cheap, and not used

移动语义可以说是C++11最主要的特性。你可能会见过这些类似的描述"移动容器和拷贝指针一样开销小"，"拷贝临时对象现在如此高效，编码避免这种情况简直就是过早优化"这种情绪很容易理解。移动语义确实是这样重要的特性。它不仅允许编译器使用开销小的移动操作代替大开销的复制操作，而且默认这么做。以C++98的代码为基础，使用C++11重新编译你的代码，然后，哇，你的软件运行的更快了。

移动语义确实令人振奋，但是有很多夸大的说法，这个Item的目的就是给你泼一瓢冷水，保持理智看待移动语义。

让我们从已知很多类型不支持移动操作开始这个过程。为了升级到C++11，C++98的很多标准库做了大修改，为很多类型提供了移动的能力，这些类型的移动实现比复制操作更快，并且对库的组件实现修改以利用移动操作。但是很有可能你工作中的代码没有完整地利用C++11。对于你的应用中（或者代码库中），没有适配C++11的部分，编译器即使支持移动语义也是无能为力的。的确，C++11倾向于为缺少移动操作定义的类默认生成，但是只有在没有声明复制操作，移动操作，或析构函数的类中才会生成移动操作（参考Item17）。禁止移动操作的类中（通过delete move operation 参考Item11），编译器不生成移动操作的支持。对于没有明确支持移动操作的类型，并且不符合编译器默认生成的条件的类，没有理由期望C++11会比C++98进行任何性能上的提升。

即使显式支持了移动操作，结果可能也没有你希望的那么好。比如，所有C++11的标准库都支持了移动操作，但是认为移动所有容器的开销都非常小是个错误。对于某些容器来说，压根就不存在开销小的方式来移动它所包含的内容。对另一些容器来说，开销真正小的移动操作却使得容器元素移动含义事与愿违。

考虑一下`std::array`，这是C++11中的新容器。`std::array`本质上是具有STL接口的内置数组。这与其他标准容器将内容存储在堆内存不同。存储具体数据在堆内存的容器，本身只保存了只想堆内存数据的指针（真正实现当然更复杂一些，但是基本逻辑就是这样）。这种实现使得在常数时间移动整个容器成为可能的，只需要拷贝容器中保存的指针到目标容器，然后将原容器的指针置为空指针就可以了。

```
std::vector<Widget> vm1;

auto vm2 = std::move(vm1); // move vm1 into vm2. Runs in constant time. Only ptrs
in vm1 and vm2 are modified
```

`std::array`没有这种指针实现，数据就保存在`std::array`容器中

```
std::array<Widget, 10000> aw1;

auto aw2 = std::move(aw1); // move aw1 into aw2. Runs in linear time. All
elements in aw1 are moved into aw2.
```

注意`aw1`中的元素被移动到了`aw2`中，这里假定`Widget`类的移动操作比复制操作快。但是使用`std::array`的移动操作还是复制操作都将花费线性时间的开销，因为每个容器中的元素终归需要拷贝一次，这与"移动一个容器就像操作几个指针一样方便"的含义想去甚远。

另一方面，`std::strnig`提供了常数时间的移动操作和线性时间的复制操作。这听起来移动比复制快多了，但是可能不一定。许多字符串的实现采用了*small string optimization(SSO)*。"small"字符串（比如长度小于15个字符的）存储在了`std::string`的缓冲区中，并没有存储在堆内存，移动这种存储的字符串并不必复制操作更快。

SSO的动机是大量证据表明，短字符串是大量应用使用的习惯。使用内存缓冲区存储而不分配堆内存空间，是为了更好的效率。然而这种内存管理的效率导致移动的效率并不必复制操作高。

即使对于支持快速移动操作的类型，某些看似可靠的移动操作最终也会导致复制。Item14解释了原因，标准库中的某些容器操作提供了强大的异常安全保证，确保C++98的代码直接升级C++11编译器不会不可运行，仅仅确保移动操作不会抛出异常，才会替换为移动操作。结果就是，即使类提供了更具效率的移动操作，编译器仍可能被迫使用复制操作来避免移动操作导致的异常。

因此，存在几种情况，C++11的移动语义并无优势：

- **No move operations**：类没有提供移动操作，所以移动的写法也会变成复制操作
- **Move not faster**：类提供的移动操作并不必复制效率更高
- **Move not usable**：进行移动的上下文要求移动操作不会抛出异常，但是该操作没有被声明为`noexcept`

值得一提的是，还有另一个场景，会使得移动并没有那么有效率：

- **Source object is lvalue**：除了极少数的情况外（例如 Item25），只有右值可以作为移动操作的来源

但是该Item的标题是假定不存在移动操作，或者开销不小，不使用移动操作。存在典型的场景，就是编写模板代码，因为你不清楚你处理的具体类型是什么。在这种情况下，你必须像出现移动语义之前那样，保守地考虑复制操作。不稳定的代码也是如此，类的特性经常被修改导致可能移动操作会有问题。

但是，通常，你了解你代码里使用的类，并且知道是否支持快速移动操作。这种情况，你无需这个Item的假设，只需要查找所用类的移动操作详细信息，并且调用移动操作的上下文中，可以安全的使用快速移动操作替换复制操作。

需要记住的事

- Assume that move operations are not present, not cheap, and not used.
- 完全了解的代码可以忽略本Item

Item30：熟悉完美转发的失败case

C++11最显眼的功能之一就是完美转发功能。完美转发，太棒了！哎，开始使用，你就发现“完美”，理想与现实还是有差距。C++11的完美转发是非常好用，但是只有当你愿意忽略一些失败情况，这个Item就是使你熟悉这些情形。

在我们开始epsilon探索之前，有必要回顾一下“完美转发”的含义。“转发”仅表示将一个函数的参数传递给另一个函数。对于被传递的第二个函数目标是收到与第一个函数完全相同的对象。这就排除了按值传递参数，因为它们是原始调用者传入内容的副本。我们希望被转发的函数能够可以与原始函数一起使用对象。指着参数也被排除在外，因为我们不想强迫调用者传入指针。关于通用转发，我们将处理引用参数。

完美转发意味着我们不仅转发对象，我们还转发显著的特征：它们的类型，是左值还是右值，是const还是volatile。结合到我们会处理引用参数，这意味着我们将使用通用引用（参见Item24），因为通用引用参数被传入参数时才确定是左值还是右值。

假定我们有一些函数f，然后想编写一个转发给它的函数（就使用一个函数模板）。我们需要的核心看起来像是这样：

```
template<typename T>
void fwd(T&& param)        // accept any argument
{
  f(std::forward<T>(param));  // forward it to f
}
```

从本质上说，转发功能是通用的。例如fwd模板，接受任何类型的采纳数，并转发得到的任何参数。这种通用性的逻辑扩展是转发函数不仅是模板，而且是可变模板，因此可以接受任何数量的参数。fwd的可变个是如下：

```
template<typename... Ts>
void fwd(Ts&&... params)    // accept any arguments
{
  f(std::forward<Ts>(params)...); // forward them to f
}
```

这种形式你会在标准化容器emplace中（参见Item42）和只能容器的工厂函数`std::make_unique`和`std::make_shared`中（参见Item21）看到。

给定我们的目标函数f和被转发的函数fwd，如果f使用特定参数做一件事，但是fwd使用相同的参数做另一件事，完美转发就会失败：

```
f(expression); // if this does one thing
fwd(expression); // but this does something else, fwd fails to perfectly forward
expression to f
```

导致这种失败的原因有很多。知道它们是什么以及如何解决它们很重要，因此让我们来看看那种参数无法做到完美转发。

Braced initializers（支撑初始化器）

假定f这样声明：

```
void f(const std::vector<int>& v);
```

在这个例子中，通过列表初始化器，

```
f({1,2,3});  // fine "{1,2,3}" implicitly converted to std::vector<int>
```

但是传递相同的列表初始化器给fwd不能编译

```
fwd({1,2,3}); // error! doesn't compile
```

这是因为这是完美转发失效的一种情况。

所有这种错误有相同的原因。在对f的直接调用（例如f({1,2,3})），编译器看到传入的参数是声明中的类型。如果类型不匹配，就会执行隐式转换操作使得调用成功。在上面的例子中，从`{1,2,3}`生成了临时变量`std::vector<int>`对象，因此f的参数会绑定到`std::vector<int>`对象上。

当通过调用函数模板fwd调用f时，编译器不再比较传入给fwd的参数和f的声明中参数的类型。代替的是，推导传入给fwd的参数类型，然后比较推导后的参数类型和f的声明类型。当下面情况任何一个发生时，完美转发就会失败：

- **编译器不能推导出一个或者多个fwd的参数类型**，编译器就会报错
- **编译器将一个或者多个fwd的参数类型推导错误**。在这里，“错误”可能意味着fwd将无法使用推导出的类型进行编译，但是也可能意味着调用者f使用fwd的推导类型对比直接传入参数类型表现出不一致的行为。这种不同行为的原因可能是因为f的函数重载定义，并且由于是“不正确的”类型推导，在fwd内部调用f和直接调用f将重载不同的函数。

在上面的`f({1,2,3})`例子中，问题在于，如标准所言，将括号初始化器传递给未声明为`std::initializer_list`的函数模板参数，该标准规定为“非推导上下文”。简单来讲，这意味着编译器在对fwd的调用中推导表达式`{1,2,3}`的类型，因为fwd的参数没有声明为`std::initializer_list`。对于fwd参数的推导类型被阻止，编译器只能拒绝该调用。

有趣的是，Item2 说明了使用**braced initializer**的auto的变量初始化的类型推导是成功的。这种变量被视为`std::initializer_list`对象，在转发函数应推导为`std::initializer_list`类型的情况，这提供了一种简单的解决方法----使用auto声明一个局部变量，然后将局部变量转发：

```
auto il = {1,2,3}; // il's type deduced to be std::initializer_list<int>
fwd(il); // fine, perfect-forwards il to f
```

0或者NULL作为空指针

Item8说明当你试图传递0或者NULL作为空指针给模板时，类型推导会出错，推导为一个整数类型而不是指针类型。结果就是不管是0还是NULL都不能被完美转发为空指针。解决方法非常简单，使用nullptr就可以了，具体的细节，参考Item 8.

仅声明的整数静态const数据成员

通常，无需在类中定义整数静态const数据成员；声明就可以了。这是因为编译器会对此类成员

```
class Widget {
public:
  static const std::size_t MinVals = 28; // MinVal's declaration
  ...
};
...      // no defn. for MinVals
std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals); // use of MinVals
```

这里，我们使用`Widget::MinVals`（或者简单点MinVals）来确定`widgetData`的初始容量，即使`MinVals`缺少定义。编译器通过将值28放入所有位置来补充缺少的定义。没有为`MinVals`的值留存储空间是没有问题的。如果要使用`MinVals`的地址（例如，有人创建了`MinVals`的指针），则`MinVals`需要存储（因为指针总要有一个地址），尽管上面的代码仍然可以编译，但是链接时就会报错，直到为`MinVals`提供定义。

按照这个思路，想象下f（转发参数给fwd的函数）这样声明：

```
void f(std::size_t val);
```

使用`MinVals`调用f是可以的，因为编译器直接将值28代替`MinVals`：

```
f(Widget::MinVals); // fine, treated as "28"
```

同样的，如果尝试通过fwd来调用f

```
fwd(Widget::MinVals); // error! shouldn't link
```

代码可以编译，但是不能链接。就像使用`MinVals`地址表现一样，确实，底层的问题是一样的。

尽管代码中没有使用`MinVals`的地址，但是fwd的参数是通用引用，而引用，在编译器生成的代码中，通常被视作指针。在程序的二进制底层代码中指针和引用是一样的。在这个水平下，引用只是可以自动取消引用的指针。在这种情况下，通过引用传递`MinVals`实际上与通过指针传递`MinVals`是一样的，因此，必须有内存使得指针可以指向。通过引用传递整型static const数据成员，必须定义它们，这个要求可能会造成完美转发失败，即使等效不使用完美转发的代码成功。（译者注：这里意思应该是没有定义，完美转发就会失败）

可能你也注意到了在上述讨论中我使用了一些模棱两可的词。代码“不应该”链接，引用“通常”被看做指针。传递整型static const数据成员“通常”要求定义。看起来就像有些事情我没有告诉你.....

确实，根据标准，通过引用传递`MinVals`要求有定义。但不是所有的实现都强制要求这一点。所以，取决于你的编译器和链接器，你可能发现你可以在未定义的情况使用完美转发，恭喜你，但是这不是那样做的理由。为了具有可移植性，只要给整型static const提供一个定义，比如这样：

```
const std::size_t Widget::MinVals; // in Widget's .cpp file
```

注意定义中不要重复初始化（这个例子中就是赋值28）。不要忽略这个细节，否则，编译器就会报错，提醒你只初始化一次。

重载的函数名称和模板名称

假定我们的函数f（通过fwd完美转发参数给f）可以通过向其传递执行某些功能的函数来定义其行为。假设这个函数参数和返回值都是整数，f声明就像这样：

```
void f(int (*pf)(int)); // pf = "process function"
```

值得注意的是，也可以使用更简单的非指针语法声明。这种声明就像这样，含义与上面是一样的：

```
void f(int pf(int)); // declares same f as above
```

无论哪种写法，我们都有了一个重载函数，processVal:

```
int processVal(int value);
int processVal(int value, int priority);
```

我们可以传递processVal给f

```
f(processVal); // fine
```

但是有一点要注意，f要求一个函数指针，但是`processVal`不是一个函数指针或者一个函数，它是两个同名的函数。但是，编译器可以知道它需要哪个：通过参数类型和数量来匹配。因此选择了一个int参数的`processVal`地址传递给f

工作的基本机制是让编译器帮选择f的声明选择一个需要的`processVal`。但是，fwd是一个函数模板，没有需要的类型信息，使得编译器不可能帮助自动匹配一个合适的函数：

```
fwd(processVal); // error! which processVal?
```

`processVal`没有类型信息，就不能类型推导，完美转发失败。

同样的问题会发生在如果我们试图使用函数模板代替重载的函数名。一个函数模板是未实例化的函数，表示一个函数族：

```
template<typename T>
T workOnVal(T param) { ... } // template for processing values
fwd(workOnVal); // error! which workOnVal instantiation ?
```

获得像fwd的完美转发接受一个重载函数名或者模板函数名的方式是指定转发的类型。比如，你可以创造与f相同参数类型的函数指针，通过processVal或者workOnVal实例化这个函数指针（可以引导生成代码时正确选择函数实例），然后传递指针给f:

```
using ProcessFuncType = int (*)(int); // make typedef; see Item 9
PorcessFuncType processValPtr = processVal; // specify needed signature for
processVal
fwd(processValPtr); // fine
fwd(static_cast<ProcessFuncType>(workOnVal)); // alse fine
```

当然，这要求你知道fwd转发的函数指针的类型。对于完美转发来说这一点并不合理，毕竟，完美转发被设计为转发任何内容，如果没有文档告诉你转发的类型，你如何知道？（译者注：这里应该想表达，这是解决重载函数名或者函数模板的解决方案，但是这是完美转发本身的问题）

位域

完美转发最后一种失败的情况是函数参数使用位域这种类型。为了更直观的解释，IPv4的头部可以如下定义：

```
struct IPv4Header {
  std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
  ...
};
```

如果声明我们的函数f（转发函数fwd的目标）为接收一个`std::size_t`的参数，则使用IPv4Header对象的totalLength字段进行调用没有问题：

```
void f(std::size_t sz);
IPv4Header h;
...
f(h.totalLength);// fine
```

如果通过fwd转发h.totalLength给f呢，那就是一个不同的情况了：

```
fwd(h.totalLength); // error!
```

问题在于fwd的参数是引用，而h.totalLength是非常量位域。听起来并不是那么糟糕，但是C++标准非常清楚地谴责了这种组合：非常量引用不应该绑定到位域。禁止的理由很充分。位域可能包含了机器字节的任意部分（比如32位int的3-5位），但是无法直接定位。我之前提到了在硬件层面引用和指针时一样的，所以没有办法创建一个指向任意bit的指针（C++规定你可以指向的最小单位是char），所以就没有办法绑定引用到任意bit上。

一旦意识到接收位域作为参数的函数都将接收位域的副本，就可以轻松解决位域不能完美转发的问题。毕竟，没有函数可以绑定引用到位域，也没有函数可以接受指向位域的指针（不存在这种指针）。这种位域类型的参数只能按值传递，或者有趣的事，常量引用也可以。在按值传递时，被调用的函数接受了一个位域的副本，而且事实表明，位域的常量引用也是将其“复制”到普通对象再传递。

传递位域给完美转发的关键就是利用接收参数函数接受的是一个副本的事实。你可以自己创建副本然后利用副本调用完美转发。在IPv4Header的例子中，可以如下写法：

```
// copy bitfield value; see Item6 for info on init. form
auto length = static_cast<std::uint16_t>(h.totalLength);
fwd(length); // forward the copy
```

总结

在大多数情况下，完美转发工作的很好。你基本不用考虑其他问题。但是当其不工作时，当看起来合理的代码无法编译，或者更糟的是，无法按照预期运行时，了解完美转发的缺陷就很重要了。同样重要的是如何解决它们。在大多数情况下，都很简单

需要记住的事

- 完美转发会失败当模板类型推导失败或者推导类型错误
- 导致完美转发失败的类型有braced initializers，作为空指针的0或者NULL，只声明的整型static const数据成员，模板和重载的函数名，位域

Lambda表达式是C++编程中的游戏规则改变者。这有点令人惊讶，因为它没有给语言带来新的表达能力。Lambda可以做的所有事情都可以通过其他方式完成。但是lambda是创建函数对象相当便捷的一种方法，对于日常的C++开发影响是巨大的。没有lambda时，标准库中的`_if`算法（比如，`std::find_if, std::remove_if, std::count_if`等）通常需要繁琐的谓词，但是当有lambda可用时，这些算法使用起来就变得相当方便。比较函数（比如，`std::sort, std::nth_element, std::lower_bound`等）与算法函数也是相同的。在标准库外，lambda可以快速创建`std::unique_ptr`和`std::shared_ptr`的自定义`deleter`，并且使线程API中条件变量的条件设置变得同样简单（参见Item 39）。除了标准库，lambda有利于即时的回调函数，接口适配函数和特定上下文中的一次性函数。Lambda确实使C++成为更令人愉快的编程语言。

与Lambda相关的词汇可能会令人疑惑，这里做一下简单的回顾：

- *lambda表达式就是一个表达式*。在代码的高亮部分就是lambda

```
std::find_if(container.begin(), container.end(),
             [](int val){ return 0 < val && val < 10; }); // 本行高亮
```

- *闭包*是lambda创建的运行时对象。依赖捕获模式，闭包持有捕获数据的副本或者引用。在上面的`std::find_if`调用中，闭包是运行时传递给`std::find_if`第三个参数。
- *闭包类（closure class）*是从中实例化闭包的类。每个lambda都会使编译器生成唯一的闭包类。Lambda中的语句成为其闭包类的成员函数中的可执行指令。

Lambda通常被用来创建闭包，该闭包仅用作函数的参数。上面对`std::find_if`的调用就是这种情况。然而，闭包通常可以拷贝，所以可能有多个闭包对应于一个lambda。比如下面的代码：

```
{
  int x; // x is local variable
  ...
  auto c1 = [x](int y) { return x * y > 55; }; // c1 is copy of the closure
produced by the lambda

  auto c2 = c1; // c2 is copy of c1
  auto c3 = c2; // c3 is copy of c2
  ...
}
```

c1， c2，c3都是lambda产生的闭包的副本。

非正式的讲，模糊lambda，闭包和闭包类之间的界限是可以接受的。但是，在随后的Item中，区分编译期（lambdas 和 closure classes）还是运行时（closures）以及它们之间的相互关系是重要的。

避免使用默认捕获模式

C++11中有两种默认的捕获模式：按引用捕获和按值捕获。但按引用捕获可能会带来悬空引用的问题，而按值引用可能会诱骗你让你以为能解决悬空引用的问题（实际上并没有），还会让你以为你的闭包是独立的（事实上也不是独立的）。

这就是本条目的一个总结。如果你是一个工程师，渴望了解更多内容，就让我们从按引用捕获的危害谈起把。

按引用捕获会导致闭包中包含了对局部变量或者某个形参（位于定义lambda的作用域）的引用，如果该lambda创建的闭包生命周期超过了局部变量或者参数的生命周期，那么闭包中的引用将会变成悬空引用。举个例子，假如我们有一个元素是过滤函数的容器，该函数接受一个int作为参数，并返回一个布尔值，该布尔值的结果表示传入的值是否满足过滤条件。

```
using FilterContainer =                 // see Item 9 for
    std::vector<std::function<bool(int)>>; // "using", Item 2
                                        // for std::function
FilterContainer filters;                // filtering funcs
```

我们可以添加一个过滤器，用来过滤掉5的倍数。

```
filters.emplace_back(                     // see Item 42 for
    [](int value) { return value % 5 == 0; } // info on
);
```

然而我们可能需要的是能够在运行期获得被除数，而不是将5硬编码到lambda中。因此添加的过滤器逻辑将会是如下这样：

```
void addDivisorFilter()
{
    auto calc1 = computeSomeValue1();
    auto calc2 = computeSomeValue2();
    auto divisor = computeDivisor(calc1, calc2);
    filters.emplace_back(                                          // danger!
        [&](int value) { return value % divisor == 0; }   // ref to
    );                                                                    //
divisor
}
 // will

 // dangle!
```

这个代码实现是一个定时炸弹。lambda对局部变量divisor进行了引用，但该变量的生命周期会在addDivisorFilter返回时结束，刚好就是在语句filters.emplace_back返回之后，因此该函数的本质就是容器添加完，该函数就死亡了。使用这个filter会导致未定义行为，这是由它被创建那一刻起就决定了的。

现在，同样的问题也会出现在divisor的显式按引用捕获。

```
filters.emplace_back(
    [&divisor](int value)           // danger! ref to
    { return value % divisor == 0; } // divisor will
);
```

但通过显式的捕获，能更容易看到lambda的可行性依赖于变量divisor的生命周期。另外，写成这种形式能够提醒我们要注意确保divisor的生命周期至少跟lambda闭包一样长。比起"[&]"传达的意思，显式捕获能让人更容易想起“确保没有悬空变量”。

如果你知道一个闭包将会被马上使用（例如被传入到一个stl算法中）并且不会被拷贝，那么在lambda环境中使用引用捕获将不会有风险。在这种情况下，你可能会争论说，没有悬空引用的危险，就不需要避免使用默认的引用捕获模式。例如，我们的过滤lambda只会用做C++11中std::all_of的一个参数，返回满足条件的所有元素：

```
template<typename C>
void workWithContainer(const C& container)
{
    auto calc1 = computeSomeValue1();           // as above
    auto calc2 = computeSomeValue2();           // as above
    auto divisor = computeDivisor(calc1, calc2); // as above
    using ContElemT = typename C::value_type;    // type of
                                                 // elements in
                                                 // container
    using std::begin;                           // for
    using std::end;                            // genericity;
                                              // see Item 13
    if (std::all_of(                           // if all values
        begin(container), end(container),       // in container
        [&](const ContElemT& value)             // are multiples
        { return value % divisor == 0; })        // of divisor...
    ) {
        …                                      // they are...
    } else {
        …                                      // at least one
    }                                          // isn't...
}
```

的确如此，这是安全的做法，但这种安全是不确定的。如果发现lambda在其它上下文中很有用（例如作为一个函数被添加在filters容器中），然后拷贝粘贴到一个divisor变量已经死亡的，但闭包生命周期还没结束的上下文中，你又回到了悬空的使用上了。同时，在该捕获语句中，也没有特别提醒了你注意分析divisor的生命周期。

从长期来看，使用显式的局部变量和参数引用捕获方式，是更加符合软件工程规范的做法。

额外提一下，C++14支持了在lambda中使用auto来声明变量，上面的代码在C++14中可以进一步简化，ContElemT的别名可以去掉，if条件可以修改为：

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value) // C++14
                { return value % divisor == 0; }))
```

一个解决问题的方法是，divisor按值捕获进去，也就是说可以按照以下方式来添加lambda：

```
filters.emplace_back(                           // now
    [=](int value) { return value % divisor == 0; } // divisor
);                                             // can't
                                               // dangle
```

这足以满足本实例的要求，但在通常情况下，按值捕获并不能完全解决悬空引用的问题。这里的问题是如果你按值捕获的是一个指针，你将该指针拷贝到lambda对应的闭包里，但这样并不能避免lambda外删除指针的行为，从而导致你的指针变成悬空指针。

也许你要抗议说：“这不可能发生。看过了第四章，我对智能指针的使用非常热衷。只有那些失败的C++98的程序员才会用裸指针和delete语句。”这也许是正确的，但却是不相关的，因为事实上你的确会使用裸指针，也的确存在被你删除的可能性。只不过在现代的C++编程风格中，不容易在源代码中显露出来而已。

假设在一个Widget类，可以实现向过滤容器添加条目：

```
class Widget {
public:
    …                           // ctors, etc.
    void addFilter() const; // add an entry to filters
private:
    int divisor;            // used in Widget's filter
};
```

这是Widget::addFilter的定义：

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=](int value) { return value % divisor == 0; }
    );
}
```

这个做法看起来是安全的代码，lambda依赖于变量divisor，但默认的按值捕获被拷贝进了lambda对应的所有比保重，这真的正确吗？

错误，完全错误。

闭包只会对lambda被创建时所在作用域里的非静态局部变量生效。在Widget::addFilter()的视线里，divisor并不是一个局部变量，而是Widget类的一个成员变量。它不能被捕获。如果默认捕获模式被删除，代码就不能编译了：

```
void Widget::addFilter() const
{
    filters.emplace_back(                                // error!
        [](int value) { return value % divisor == 0; }  // divisor
    );                                                   // not
}                                                        // available
```

另外，如果尝试去显式地按引用或者按值捕获divisor变量，也一样会编译失败，因为divisor不是这里的一个局部变量或者参数。

```
void Widget::addFilter() const
{
    filters.emplace_back(
      [divisor](int value)            // error! no local
      { return value % divisor == 0; } // divisor to capture
    );
}
```

因此这里的默认按值捕获并不是不会变量divisor，但它的确能够编译通过，这是怎么一回事呢？

解释就是这里隐式捕获了this指针。每一个非静态成员函数都有一个this指针，每次你使用一个类内的成员时都会使用到这个指针。例如，编译器会在内部将divisor替换成this->divisor。这里Widget::addFilter()的版本就是按值捕获了this。

```
void Widget::addFilter() const
{
    filters.emplace_back(
      [=](int value) { return value % divisor == 0; }
    );
}
```

真正被捕获的是Widget的this指针。编译器会将上面的代码看成以下的写法：

```
void Widget::addFilter() const
{
    auto currentObjectPtr = this;

    filters.emplace_back(
      [currentObjectPtr](int value)
      { return value % currentObject->divisor == 0; }
    );
}
```

明白了这个就相当于明白了lambda闭包的生命周期与Widget对象的关系，闭包内含有Widget的this指针的拷贝。特别是考虑以下的代码，再参考一下第四章的内容，只使用智能指针：

```
using FilterContainer =                     // as before
    std::vector<std::function<bool(int)>>;
FilterContainer filters;                    // as before
void doSomeWork()
{
    auto pw =                               // create Widget; see
        std::make_unique<Widget>();         // Item 21 for
                                            // std::make_unique
    pw->addFilter();                        // add filter that uses
                                            // Widget::divisor
    …
}                                           // destroy Widget; filters
                                            // now holds dangling pointer!
```

当调用doSomeWork时，就会创建一个过滤器，其生命周期依赖于由std::make_unique管理的Widget对象。即一个含有Widget this指针的过滤器。这个过滤器被添加到filters中，但当doSomeWork结束时，Widget会由std::unique_ptr去结束其生命。从这时起，filter会含有一个悬空指针。

这个特定的问题可以通过做一个局部拷贝去解决：

```
void Widget::addFilter() const
{
    auto divisorCopy = divisor;             // copy data member
    filters.emplace_back(
        [divisorCopy](int value)            // capture the copy
        { return value % divisorCopy == 0; } // use the copy
    );
}
```

事实上如果采用这种方法，默认的按值捕获也是可行的。

```
void Widget::addFilter() const
{
    auto divisorCopy = divisor;             // copy data member
    filters.emplace_back(
        [=](int value)                      // capture the copy
        { return value % divisorCopy == 0; } // use the copy
    );
}
```

但为什么要冒险呢？当你一开始捕获divisor的时候，默认的捕获模式就会自动将this指针捕获进来了。

在C++14中，一个更好的捕获成员变量的方式时使用通用的lambda捕获：

```
void Widget::addFilter() const
{
    filters.emplace_back(               // C++14:
        [divisor = divisor](int value)   // copy divisor to closure
        { return value % divisor == 0; } // use the copy
    );
}
```

这种通用的lambda捕获并没有默认的捕获模式，因此在C++14中，避免使用默认捕获模式的建议仍然时成立的。

使用默认的按值捕获还有另外的一个缺点，它们预示了相关的闭包是独立的并且不受外部数据变化的影响。一般来说，这是不对的。lambda并不会独立于局部变量和参数，但也没有不受静态存储生命周期的影响。一个定义在全局空间或者指定命名空间的全局变量，或者是一个声明为static的类内或文件内的成员。这些对象也能在lambda里使用，但它们不能被捕获。但按值引用可能会因此误导你，让你以为捕获了这些变量。参考下面版本的addDivisorFilter()函数：

```
void addDivisorFilter()
{
    static auto calc1 = computeSomeValue1(); // now static
    static auto calc2 = computeSomeValue2(); // now static
    static auto divisor =                    // now static
    computeDivisor(calc1, calc2);
    filters.emplace_back(
        [=](int value)                       // captures nothing!
        { return value % divisor == 0; }      // refers to above static
    );
    ++divisor;                               // modify divisor
}
```

随意地看了这份代码的读者可能看到"[=]"，就会认为“好的，lambda拷贝了所有使用的对象，因此这是独立的”。但上面的例子就表现了不独立闭包的一种情况。它没有使用任何的非static局部变量和形参，所以它没有捕获任何东西。然而lambda的代码引用了静态变量divisor，任何lambda被添加到filters之后，divisor都会递增。通过这个函数，会把许多lambda都添加到filiters里，但每一个lambda的行为都是新的（分别对应新的divisor值）。这个lambda是通过引用捕获divisor，这和默认的按值捕获表示的含义有着直接的矛盾。如果你一开始就避免使用默认的按值捕获模式，你就能解除代码的风险。

建议

- 默认的按引用捕获可能会导致悬空引用；
- 默认的按值引用对于悬空指针很敏感（尤其是this指针），并且它会误导人产生lambda是独立的想法；

使用初始化捕获来移动对象到闭包中

在某些场景下，按值捕获和按引用捕获都不是你所想要的。如果你有一个只能被移动的对象（例如**std::unique_ptr**或**std::future**）要进入到闭包里，使用C++11是无法实现的。如果你要复制的对象复制开销非常高，但移动的成本却不高（例如标准库中的大多数容器），并且你希望的是宁愿移动该对象到闭包而不是复制它。然而C++11却无法实现这一目标。

但如果你的编译器支持C++14，那又是另一回事了，它能支持将对象移动道闭包中。如果你的兼容支持C++14，那么请愉快地阅读下去。如果你仍然在使用仅支持C++11的编译器，也请愉快阅读，因为在C++11中有很多方法可以实现近似的移动捕获。

缺少移动捕获被认为是C++11的一个缺点，直接的补救措施是将该特性添加到C++14中，但标准化委员会选择了另一种方法。他们引入了一种新的捕获机制，该机制非常灵活，移动捕获是它执行的技术之一。新功能被称作初始化捕获，它几乎可以完成C++11捕获形式的所有工作，甚至能完成更多功能。默认的捕获模式使得你无法使用初始化捕获表示，但第31项说明提醒了你无论如何都应该远离这些捕获模式。（在C++11捕获模式所能覆盖的场景里，初始化捕获的语法有点不大方便。因此在C++11的捕获模式能完成所需功能的情况下，使用它是完全合理的）。

使用初始化捕获可以让你指定：

1. 从lambda生成的闭包类中的数据成员名称；
2. 初始化该成员的表达式；

这是使用初始化捕获将**std::unique_ptr**移动到闭包中的方法：

```
class Widget { // some useful type
public:
...
  bool isValidated() const;
  bool isProcessed() const;
  bool isArchived() const;
private: ...
};

auto pw = std::make_unique<Widget>(); // create Widget; see Item 21 for info on
std::make_unique configure *pw

auto func = [pw = std::move(pw)] // init data mbr in closure w/ std::move(pw)
                        { return pw->isValidated()
                                             && pw->isArchived(); };
```

上面的文本包含了初始化捕获的使用，"="的左侧是指定的闭包类中数据成员的名称，右侧则是初始化表达式。有趣的是，"="左侧的作用范围不同于右侧的作用范围。在上面的示例中，'='左侧的名称`pw`表示闭包类中的数据成员，而右侧的名称`pw`表示在lambda上方声明的对象，即由调用初始化的变量到调用`std::make_unique`。因此，`pw = std :: move(pw)`的意思是“在闭包中创建一个数据成员pw，并通过将`std::move`应用于局部变量pw的方法来初始化该数据成员。

一般中，lambda主体中的代码在闭包类的作用范围内，因此pw的使用指的是闭包类的数据成员。

在此示例中，注释`configure * pw`表示在由`std::make_unique`创建窗口小部件之后，再由lambda捕获到该窗口小部件的`std::unique_ptr`之前，该窗口小部件即pw对象以某种方式进行了修改。如果不需要这样的配置，即如果`std::make_unique`创建的`Widget`处于适合被lambda捕获的状态，则不需要局部变量`pw`，因为闭包类的数据成员可以通过直接初始化`std::make_unique`来实现：

```
auto func = [pw = std::make_unique<Widget>()] // init data mbr
                        { return pw->isValidated()              // in closure w/
                                            && pw->isArchived(); };    // result of
call // to make_unique
```

这清楚地表明了，这个C ++ 14的捕获概念是从C ++11发展出来的的，在C ++11中，无法捕获表达式的结果。 因此，初始化捕获的另一个名称是广义lambda捕获。
但是，如果您使用的一个或多个编译器不支持C ++ 14的初始捕获怎么办？ 如何使用不支持移动捕获的语言完成移动捕获？

请记住，lambda表达式只是生成类和创建该类型对象的一种方式而已。如果对于lambda，你觉得无能为力。 那么我们刚刚看到的C++ 14的示例代码可以用C ++11重新编写，如下所示：

```
class IsValAndArch {
public:
    using DataType = std::unique_ptr<Widget>; // "is validated and archived"
  explicit IsValAndArch(DataType&& ptr) // Item 25 explains
                    : pw(std::move(ptr)) {}         // use of std::move
  bool operator()() const
  { return pw->isValidated() && pw->isArchived(); }
private:
  DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

这个代码量比lambda表达式要多，但这并不难改变这样一个事实，即如果你希望使用一个C++11的类来支持其数据成员的移动初始化，那么你唯一要做的就是在键盘上多花点时间。

如果你坚持要使用lambda（并且考虑到它们的便利性，你可能会这样做），可以在C++11中这样使用：

1. 将要捕获的对象移动到由`std::bind`；
2. 将被捕获的对象赋予一个引用给lambda；

如果你熟悉std::bind，那么代码其实非常简单。如果你不熟悉std::bind，那可能需要花费一些时间来习惯改代码，但这无疑是值得的。

假设你要创建一个本地的`std::vector`，在其中放入一组适当的值，然后将其移动到闭包中。在C ++14中，这很容易实现：

```
std::vector<double> data; // object to be moved
                          // into closure
                          // populate data
auto func = [data = std::move(data)] { /* uses of data */ }; // C++14 init
capture
```

我已经对该代码的关键部分进行了高亮：要移动的对象的类型（`std::vector\<double>`），该对象的名称（数据）以及用于初始化捕获的初始化表达式（`std::move(data)`）。C++11的等效代码如下，其中我强调了相同的关键事项：

```
std::vector<double> data; // as above
auto func =
  std::bind(
// C++11 emulation
        [](const std::vector<double>& data) { /* uses of data */ }, // of init
capture
    std::move(data)
);
```

如lambda表达式一样，`std::bind`生产了函数对象。我将它称呼为由std::bind所绑定对象返回的函数对象。`std::bind`的第一个参数是可调用对象，后续参数表示要传递给该对象的值。

一个绑定的对象包含了传递给`std::bind`的所有参数副本。对于每个左值参数，绑定对象中的对应对象都是复制构造的。对于每个右值，它都是移动构造的。在此示例中，第二个参数是一个右值（`std::move`的结果，请参见第23项），因此将数据移动到绑定对象中。这种移动构造是模仿移动捕获的关键，因为将右值移动到绑定对象是我们解决无法将右值移动到C++11闭包中的方法。

当“调用”绑定对象（即调用其函数调用运算符）时，其存储的参数将传递到最初传递给`std::bind`的可调用对象。在此示例中，这意味着当调用func（绑定对象）时，func中所移动构造的数据副本将作为参数传递给传递给`std::bind`中的lambda。

该lambda与我们在C++14中使用的lambda相同，只是添加了一个参数data来对应我们的伪移动捕获对象。此参数是对绑定对象中数据副本的左值引用。（这不是右值引用，因尽管用于初始化数据副本的表达式（`std::move(data)`)为右值，但数据副本本身为左值。）因此，lambda将对绑定在对象内部的移动构造数据副本进行操作。

默认情况下，从lambda生成的闭包类中的`operator()`成员函数为`const`的。这具有在lambda主体内呈现闭包中的所有数据成员为`const`的效果。但是，绑定对象内部的移动构造数据副本不一定是`const`的，因此，为了防止在lambda内修改该数据副本，lambda的参数应声明为`const`引用。 如果将`lambda`声明为可变的，则不会在其闭包类中将`operator()`声明为const，并且在lambda的参数声明中省略`const`也是合适的：

```
auto func =
        std::bind(                                  // C++11 emulation
        [](std::vector<double>& data) mutable       // of init capture
        { /* uses of data */ },                                 // for
mutable lambda std::move(data)
);
```

因为该绑定对象存储着传递给`std::bind`的所有参数副本，所以在我们的示例中，绑定对象包含由lambda生成的闭包副本，这是它的第一个参数。 因此闭包的生命周期与绑定对象的生命周期相同。 这很重要，因为这意味着只要存在闭包，包含伪移动捕获对象的绑定对象也将存在。

如果这是您第一次接触`std::bind`，则可能需要先阅读您最喜欢的C ++11参考资料，然后再进行讨论所有详细信息。 即使是这样，这些基本要点也应该清楚：

- 无法将移动构造一个对象到C ++11闭包，但是可以将对象移动构造为C++11的绑定对象。
- 在C++11中模拟移动捕获包括将对象移动构造为绑定对象，然后通过引用将对象移动构造传递给lambda。
- 由于绑定对象的生命周期与闭包对象的生命周期相同，因此可以将绑定对象中的对象视为闭包中的对象。

作为使用`std::bind`模仿移动捕获的第二个示例，这是我们之前看到的在闭包中创建`std::unique_ptr`的C++14代码：

```
auto func = [pw = std::make_unique<Widget>()]   // as before,
                        { return pw->isValidated()                  // create pw
                        && pw->isArchived(); };                         // in
closure
```

这是C++11的模拟实现：

```
auto func = std::bind(
              [](const std::unique_ptr<Widget>& pw)
              { return pw->isValidated()
                              && pw->isArchived(); },
              std::make_unique<Widget>()
);
```

具备讽刺意味的是，这里我展示了如何使用`std::bind`解决C++11 lambda中的限制，但在条款34中，我却主张在`std::bind`上使用lambda。

但是，该条目解释的是在C++11中有些情况下`std::bind`可能有用，这就是其中一种。（在C++14中，初始化捕获和自动参数等功能使得这些情况不再存在。）

要谨记的是：

- 使用C ++14的初始化捕获将对象移动到闭包中。
- 在C ++11中，通过手写类或`std::bind`的方式来模拟初始化捕获。

对于std::forward的auto&&形参使用decltype

泛型lambda(generic lambdas)是C++14中最值得期待的特性之一——因为在lambda的参数中可以使用auto关键字。这个特性的实现是非常直截了当的：即在闭包类中的operator()函数是一个函数模版。例如存在这么一个lambda：

```
auto f = [](auto x){ return func(normalize(x)); };
```

对应的闭包类中的函数调用操作符看来就变成这样：

```
class SomeCompilerGeneratedClassName { public:
  template<typename T>
  auto operator()(T x) const
  { return func(normalize(x)); }
  ...
};
```

在这个样例中，lambda对变量`x`做的唯一一件事就是把它转发给函数`normalize`。如果函数`normalize`对待左值右值的方式不一样，这个lambda的实现方式就不大合适了，因为即使传递到lambda的实参是一个右值，lambda传递进去的形参总是一个左值。

实现这个lambda的正确方式是把`x`完美转发给函数`normalize`。这样做需要对代码做两处修改。首先，x需要改成通用引用，其次，需要使用`std::forward`将`x`转发到函数`normalize`。实际上的修改如下：

```
auto f = [](auto&& x)
    { return func(normalize(std::forward<???>(x))); };
```

在理论和实际之间存在一个问题：你传递给`std::forward`的参数是什么类型，就决定了上面的`???`该怎么修改。

一般来说，当你在使用完美转发时，你是在一个接受类型参数为`T`的模版函数里，所以你可以写`std::forward<T>`。但在泛型lambda中，没有可用的类型参数`T`。在lambda生成的闭包里，模版化的`operator()`函数中的确有一个`T`，但在lambda里却无法直接使用它。

前面item28解释过在传递给通用引用的是一个左值，那么它会变成左值引用。传递的是右值就会变成右值引用。这意味着在这个lambda中，可以通过检查`x`的类型来检查传递进来的实参是一个左值还是右值，decltype就可以实现这样的效果。传递给lambda的是一个左值，`decltype(x)`就能产生一个左值引用；如果传递的是一个右值，`decltype(x)`就会产生右值引用。

Item28也解释过在调用`std::forward`，传递给它的类型类型参数是一个左值引用时会返回一个左值；传递的是一个非引用类型时，返回的是一个右值引用，而不是常规的非引用。在前面的lambda中，如果x绑定的是一个左值引用，`decltype(x)`就能产生一个左值引用；如果绑定的是一个右值，`decltype(x)`就会产生右值引用，而不是常规的非引用。

在看一下Item28中关于`std::forward`的C++14实现：

```
template<typename T> // in namespace
T&& forward(remove_reference_t<T>& param) // std
{
    return static_cast<T&&>(param);
}
```

如果用户想要完美转发一个Widget类型的右值时，它会使用Widget类型（非引用类型）来示例化`std::forward`，然后产生以下的函数：

```
Widget&& forward(Widget& param)
{                                                                    //
instantiation of
    return static_cast<Widget&&>(param); // std::forward when
}                                                                    // T is
Widget
```

思考一下如果用户代码想要完美转发一个Widget类型的右值，但没有遵守规则将T指定为非引用类型，而是将T指定为右值引用，这回发生什么？思考将T换成`Widget`如何，在`std::forward`实例化、应用了`remove_reference_t`后，音乐折叠之前，这是产生的代码：

```
Widget&& && forward(Widget& param)        // instantiation of
{                                                                  //
std::forward when
    return static_cast<Widget&& &&>(param); // T is Widget&&
}                                                                        //
(before reference-collapsing)
```

应用了引用折叠之后，代码会变成：

```
Widget&& forward(Widget& param)        // instantiation of
{                                         // std::forward when
    return static_cast<Widget&&>(param); // T is Widget&&
}                                         // (before reference-collapsing)
```

对比发现，用一个右值引用去实例化`std::forward`和用非引用类型去实例化产生的结果是一样的。

那是一个很好的消息，引用当传递给lambda形参x的是一个右值实参时，`decltype(x)`可以产生一个右值引用。前面已经确认过，把一个左值传给lambda时，`decltype(x)`会产生一个可以传给`std::forward`的常规类型。而现在也验证了对于右值，把`decltype(x)`产生的类型传递给`std::forward`的类型参数是非传统的，不过它产生的实例化结果与传统类型相同。所以无论是左值还是右值，把`decltype(x)`传递给`std::forward`都能得到我们想要的结果，因此lambda的完美转发可以写成：

```
auto f =
    [](auto&& param)
    {
        return
          func(normalize(std::forward<decltype(pram)>(param)));
    };
```

再加上6个点，就可以让我们的lambda完美转发接受多个参数了，因为C++14中的lambda参数是可变的：

```
auto f =
    [](auto&&... params)
    {
        return
          func(normalized(std::forward<decltype(params)>(params)...));
    };
```

要谨记的是:

- 对`auto&&`参数使用`decltype`来（`std::forward`）转发参数;

考虑lambda表达式而非std::bind

C++11中的`std::bind`是C++98的`std::bind1st`和`std::bind2nd`的后续，但在2005年已经成为了标准库的一部分。那时标准化委员采用了TR1的文档，其中包含了bind的规范。（在TR1中，`bind`位于不同的命名空间，因此它是`std::tr1::bind`，而不是`std::bind`，接口细节也有所不同）。这段历史意味着一些程序员有十年或更长时间的使用`std::bind`经验。如果您是其中之一，可能会不愿意放弃一个对您有用的工具。这是可以理解的，但是在这种情况下，改变是更好的，因为在C ++11中，`lambda`几乎是比`std :: bind`更好的选择。 从C++14开始，`lambda`的作用不仅强大，而且是完全值得使用的。

这个条目假设您熟悉`std::bind`。 如果不是这样，您将需要获得基本的了解，然后再继续。 无论如何，这样的理解都是值得的，因为您永远不知道何时会在必须阅读或维护的代码库中遇到`std::bind`的使用。

与第32项中一样，我们将从`std::bind`返回的函数对象称为绑定对象。

优先lambda而不是`std::bind`的最重要原因是lambda更易读。 例如，假设我们有一个设置闹钟的函数：

```
// typedef for a point in time (see Item 9 for syntax)
using Time = std::chrono::steady_clock::time_point;

// see Item 10 for "enum class"
enum class Sound { Beep, Siren, Whistle };

// typedef for a length of time
using Duration = std::chrono::steady_clock::duration;
// at time t, make sound s for duration d void setAlarm(Time t, Sound s, Duration
d);
```

进一步假设，在程序的某个时刻，我们已经确定需要设置一个小时后响30秒的闹钟。 但是，具体声音仍未确定。我们可以编写一个lambda来修改`setAlarm`的界面，以便仅需要指定声音：

```
// setSoundL ("L" for "lambda") is a function object allowing a // sound to be
specified for a 30-sec alarm to go off an hour // after it's set
auto setSoundL =
  [](Sound s)
    {
    // make std::chrono components available w/o qualification
    using namespace std::chrono;
    setAlarm(steady_clock::now() + hours(1), // alarm to go off
             s, // in an hour for
             seconds(30)); // 30 seconds
    };
```

我们在lambda中突出了对`setAlarm`的调用。这看来起是一个很正常的函数调用，即使是几乎没有lambda经验的读者也可以看到：传递给lambda的参数被传递给了`setAlarm`。

通过使用基于C++11对用户自定义常量的支持而建立的标准后缀，如秒(s)，毫秒(ms)和小时(h)等，我们可以简化C++14中的代码。这些后缀在`std::literals`命名空间中实现，因此上述代码可以按照以下方式重写：

```
auto setSoundL =
  [](Sound s)
  {
    using namespace std::chrono;
    using namespace std::literals; // for C++14 suffixes
        setAlarm(steady_clock::now() + 1h, // C++14, but
             s, // same meaning
             30s); // as above
    };
```

下面是我们第一次编写对应的`std::bind`调用。这里存在一个我们后续会修复的错误，但正确的代码会更加复杂，即使是此简化版本也会带来一些重要问题：

```
using namespace std::chrono; // as above
using namespace std::literals;
using namespace std::placeholders; // needed for use of "_1"
auto setSoundB = std::bind(setAlarm, // "B" for "bind"
                           steady_clock::now() + 1h, // incorrect! see below
                           _1,
                           30s);
```

我想像在lambda中一样突出显示对`setAlarm`的调用，但是没有这么做。这段代码的读者只需知道，调用`setSoundB`会使用在对`std :: bind`的调用中所指定的时间和持续时间来调用`setAlarm`。对于初学者来说，占位符**“ _1”**本质上是一个魔术，但即使是普通读者也必须从思维上将占位符中的数字映射到其在`std::bind`参数列表中的位置，以便明白调用`setSoundB`时的第一个参数会被传递进`setAlarm`，作为调用时的第二个参数。在对`std::bind`的调用中未标识此参数的类型，因此读者必须查阅`setAlarm`声明以确定将哪种参数传递给`setSoundB`。

但正如我所说，代码并不完全正确。在lambda中，表达式`steady_clock::now() + 1h`显然是是`setAlarm`的参数。调用`setAlarm`时将对其进行计算。这是合理的：我们希望在调用`setAlarm`后一小时发出警报。但是，在`std::bind`调用中，将`steady_clock::now() + 1h`作为参数传递给了`std::bind`，而不是setAlarm。这意味着将在调用std::bind时对表达式进行求值，并且该表达式产生的时间将存储在结果绑定对象中。结果，闹钟将被设置为在调用std::bind后一小时发出声音，而不是在调用setAlarm`一小时后发出。

要解决此问题，需要告诉`std::bind`推迟对表达式的求值，直到调用`setAlarm`为止，而这样做的方法是将对`std::bind`的第二个调用嵌套在第一个调用中：

```
auto setSoundB =
     std::bind(setAlarm,
     std::bind(std::plus<>(), steady_clock::now(), 1h), _1,
     30s);
```

如果您熟悉C++98的`std::plus`模板，您可能会惊讶地发现在此代码中，尖括号之间未指定任何类型，即该代码包含`std::plus<>`，而不是`std::plus<type>`。 在C ++14中，通常可以省略标准运算符模板的模板类型参数，因此无需在此处提供。 C++11没有提供此类功能，因此等效于lambda的C ++11`std::bind`使用为：

```
using namespace std::chrono;                    // as above
using namespace std::placeholders;
auto setSoundB =
     std::bind(setAlarm,
              std::bind(std::plus<steady_clock::time_point>(),
steady_clock::now(), hours(1)),
               seconds(30));
```

如果此时Lambda看起来不够吸引，那么应该检查一下视力了。

当setAlarm重载时，会出现一个新问题。 假设有一个重载函数，其中第四个参数指定了音量：

```
enum class Volume { Normal, Loud, LoudPlusPlus };
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

lambda能继续像以前一样使用，因为根据重载规则选择了`setAlarm`的三参数版本：

```
auto setSoundL =
    [](Sound s)
    {
      using namespace std::chrono;
        setAlarm(steady_clock::now() + 1h, s,
        30s);
        };
```

然而，`std::bind`的调用将会编译失败：

```
auto setSoundB =                                // error! which
    std::bind(setAlarm,                         // setAlarm?
                           std::bind(std::plus<>(),
              steady_clock::now(),
                1h),
    _1,
        30s);
```

这里的问题是，编译器无法确定应将两个setAlarm函数中的哪一个传递给`std::bind`。 它们仅有的是一个函数名称，而这个函数名称是不确定的。
要获得对`std::bind`的调用能进行编译，必须将`setAlarm`强制转换为适当的函数指针类型：

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);
auto setSoundB = // now
  std::bind(static_cast<SetAlarm3ParamType>(setAlarm), // okay
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

但这在`lambda`和`std::bind`的使用上带来了另一个区别。 在`setSoundL`的函数调用操作符（即lambda的闭包类对应的函数调用操作符）内部，对`setAlarm`的调用是正常的函数调用，编译器可以按常规方式进行内联：

```
setSoundL(Sound::Siren);    // body of setAlarm may
                                                // well be inlined here
```

但是，对`std::bind`的调用是将函数指针传递给`setAlarm`，这意味着在`setSoundB`的函数调用操作符（即绑定对象的函数调用操作符）内部，对`setAlarm`的调用是通过一个函数指针。 编译器不太可能通过函数指针内联函数，这意味着与通过`setSoundL`进行调用相比，通过`setSoundB`对`setAlarm`的调用，其函数不大可能被内联：

```
setSoundB(Sound::Siren);    // body of setAlarm is less
                                                // likely to be inlined here
```

因此，使用`lambda`可能会比使用`std::bind`能生成更快的代码。
`setAlarm`示例仅涉及一个简单的函数调用。如果您想做更复杂的事情，使用lambda会更有利。 例如，考虑以下C++14的lambda使用，它返回其参数是否在最小值（`lowVal`）和最大值（`highVal`）之间的结果，其中`lowVal`和`highVal` 是局部变量：

```
auto betweenL =
     [lowVal, highVal]
     (const auto& val)                          // C++14
     { return lowVal <= val && val <= highVal; };
```

使用`std::bind`可以表达相同的内容，但是该构造是一个通过晦涩难懂的代码来保证工作安全性的示例：

```
using namespace std::placeholders;          // as above
auto betweenB =
    std::bind(std::logical_and<>(), // C++14
                        std::bind(std::less_equal<>(), lowVal, _1),
                    std::bind(std::less_equal<>(), _1, highVal));
```

在C++11中，我们必须指定要比较的类型，然后`std::bind`调用将如下所示：

```
auto betweenB = // C++11 version
  std::bind(std::logical_and<bool>(),
                        std::bind(std::less_equal<int>(), lowVal, _1),
            std::bind(std::less_equal<int>(), _1, highVal));
```

当然，在C++11中，lambda也不能采用`auto`参数，因此它也必须指定一个类型：

```
auto betweenL = // C++11 version
  [lowVal, highVal]
    (int val)
    { return lowVal <= val && val <= highVal; };
```

无论哪种方式，我希望我们都能同意，lambda版本不仅更短，而且更易于理解和维护。
之前我就说过，对于那些没有`std::bind`使用经验的人，其占位符（例如_1，_2等）本质上都是magic。 但是，不仅仅占位符的行为是不透明的。 假设我们有一个函数可以创建Widget的压缩副本，

```
enum class CompLevel { Low, Normal, High };  // compression
                                             // level
Widget compress(const Widget& w,             // make compressed
                CompLevel lev);              // copy of w
```

并且我们想创建一个函数对象，该函数对象允许我们指定应将特定`w`的压缩级别。这种使用`std::bind`的话将创建一个这样的对象：

```
Widget w;
using namespace std::placeholders;
auto compressRateB = std::bind(compress, w, _1);
```

现在，当我们将`w`传递给`std::bind`时，必须将其存储起来，以便以后进行压缩。它存储在对象compressRateB中，但是这是如何存储的呢（是通过值还是引用）。之所以会有所不同，是因为如果在对`std::bind`的调用与对`compressRateB`的调用之间修改了`w`，则按引用捕获的`w`将反映其更改，而按值捕获则不会。

答案是它是按值捕获的，但唯一知道的方法是记住`std::bind`的工作方式；在对`std::bind`的调用中没有任何迹象。与lambda方法相反，其中`w`是通过值还是通过引用捕获是显式的：

```
auto compressRateL =            // w is captured by
  [w](CompLevel lev)                        // value; lev is
    { return compress(w, lev); };   // passed by value
```

同样明确的是如何将参数传递给lambda。 在这里，很明显参数`lev`是通过值传递的。 因此：

```
compressRateL(CompLevel::High); // arg is passed
                                                  // by value
```

但是在对由`std::bind`生成的对象调用中，参数如何传递？

```
compressRateB(CompLevel::High); // how is arg
                                                  // passed?
```

同样，唯一的方法是记住`std::bind`的工作方式。（答案是传递给绑定对象的所有参数都是通过引用传递的，因为此类对象的函数调用运算符使用完美转发。）
与lambda相比，使用`std::bind`进行编码的代码可读性较低，表达能力较低，并且效率可能较低。 在C++14中，没有`std::bind`的合理用例。 但是，在C ++11中，可以在两个受约束的情况下证明使用`std::bind`是合理的：

- 移动捕获。 C++11的lambda不提供移动捕获，但是可以通过结合lambda和`std::bind`来模拟。有关详细信息，请参阅条款32，该条款还解释了在C ++ 14中，lambda对初始化捕获的支持将少了模拟的需求。
- 多态函数对象。 因为绑定对象上的函数调用运算符使用完全转发，所以它可以接受任何类型的参数（以条款30中描述的完全转发的限制为例子）。当您要使用模板化函数调用运算符来绑定对象时，此功能很有用。 例如这个类，

```
class PolyWidget {
 public:
    template<typename T>
    void operator()(const T& param); ...
};
```

`std::bind`可以如下绑定一个`PolyWidget`对象：

```
PolyWidget pw;
auto boundPW = std::bind(pw, _1);
```

`boundPW`可以接受任意类型的对象了：

```
boundPW(1930);  // pass int to
                                  // PolyWidget::operator()
boundPW(nullptr); // pass nullptr to
                                    // PolyWidget::operator()
boundPW("Rosebud"); // pass string literal to
                                       // PolyWidget::operator()
```

这一点无法使用C++11的lambda做到。 但是，在C++14中，可以通过带有`auto`参数的lambda轻松实现：

```
auto boundPW = [pw](const auto& param) // C++14
                               { pw(param); };
```

当然，这些是特殊情况，并且是暂时的特殊情况，因为支持C++14 lambda的编译器越来越普遍了。 当`bind`在2005年被非正式地添加到C ++中时，与1998年的前身相比有了很大的改进。 在C ++11中增加了lambda支持，这使得`std::bind`几乎已经过时了，从C ++ 14开始，更是没有很好的用例了。

要谨记的是：

- 与使用`std::bind`相比，Lambda更易读，更具表达力并且可能更高效。
- 只有在C++11中，`std::bind`可能对实现移动捕获或使用模板化函数调用运算符来绑定对象时会很有用。

C++11的伟大标志之一是将并发整合到语言和库中。熟悉其他线程API（比如pthreads或者Windows threads）的开发者有时可能会对C++提供的斯巴达式（译者注：应该是简陋和严谨的意思）功能集感到惊讶，这是因为C++对于并发的大量支持是在编译器的约束层面。由此产生的语言保证意味着在C++的历史中，开发者首次通过标准库可以写出跨平台的多线程程序。这位构建表达库奠定了坚实的基础，并发标准库（tasks, futures, threads, mutexes, condition variables, atomic objects等）仅仅是成为并发软件开发者丰富工具集的基础。

在接下来的Item中，记住标准库有两个futures的模板：`std::future`和`std::shared_future`。在许多情况下，区别不重要，所以我们经常简单的混于一谈为*futures*。

优先基于任务编程而不是基于线程

如果开发者想要异步执行 `doAsyncWork` 函数，通常有两种方式。其一是通过创建 `std::thread` 执行 `doAsyncWork`， 比如

```
int doAsyncWork();
std::thread t(doAsyncWork);
```

其二是将 `doAsyncWork` 传递给 `std::async`， 一种基于任务的策略：

```
auto fut = std::async(doAsyncWork); // "fut" for "future"
```

这种方式中，函数对象作为一个任务传递给 `std::async`。

基于任务的方法通常比基于线程的方法更优，原因之一上面的代码已经表明，基于任务的方法代码量更少。我们假设唤醒`doAsyncWork`的代码对于其提供的返回值是有需求的。基于线程的方法对此无能为力，而基于任务的方法可以简单地获取`std::async`返回的`future`提供的`get`函数获取这个返回值。如果`doAsycnWork`发生了异常，`get`函数就显得更为重要，因为`get`函数可以提供抛出异常的访问，而基于线程的方法，如果`doAsyncWork`抛出了异常，线程会直接终止（通过调用`std::terminate`）。

基于线程与基于任务最根本的区别在于抽象层次的高低。基于任务的方式使得开发者从线程管理的细节中解放出来，对此在C++并发软件中总结了'thread'的三种含义：

- 硬件线程（Hardware threads）是真实执行计算的线程。现代计算机体系结构为每个CPU核心提供一个或者多个硬件线程。
- 软件线程（Software threads）（也被称为系统线程）是操作系统管理的在硬件线程上执行的线程。通常可以存在比硬件线程更多数量的软件线程，因为当软件线程被比如 I/O、同步锁或者条件变量阻塞的时候，操作系统可以调度其他未阻塞的软件线程执行提供吞吐量。
- `std::threads`是C++执行过程的对象，并作为软件线程的handle(句柄)。`std::threads`存在多种状态，1. `null`表示空句柄，因为处于默认构造状态（即没有函数来执行），因此不对应任何软件线程。 2. moved from (moved-to的`std::thread`就对应软件进程开始执行) 3. `joined`（连接唤醒与被唤醒的两个线程） 4. `detached`（将两个连接的线程分离）

软件线程是有限的资源。如果开发者试图创建大于系统支持的硬件线程数量，会抛出`std::system_error`异常。即使你编写了不抛出异常的代码，这仍然会发生，比如下面的代码，即使`doAsyncWork`是`noexcept`

```
int doAsyncWork() noexcept; // see Item 14 for noexcept
```

这段代码仍然会抛出异常。

```
std::thread t(doAsyncWork); // throw if no more
                            // threads are available
```

设计良好的软件必须有效地处理这种可能性（软件线程资源耗尽），一种有效的方法是在当前线程执行`doAsyncWork`，但是这可能会导致负载不均，而且如果当前线程是GUI线程，可能会导致响应时间过长的问题；另一种方法是等待当前运行的线程结束之后再创建新的线程，但是仍然有可能当前运行的线程在等待`doAsyncWork`的结果（例如操作得到的变量或者条件变量的通知）。

即使没有超出软件线程的限额，仍然可能会遇到资源超额的麻烦。如果当前准备运行的软件线程大于硬件线程的数量，系统的线程调度程序会将硬件核心的时间切片，当一个软件线程的时间片执行结束，会让给另一个软件线程，即发生上下文切换。软件线程的上下文切换会增加系统的软件线程管理开销，并且如果发生了硬件核心漂移，这个开销会更高，具体来说，如果发生了硬件核心漂移，（1）CPU cache中关于上次执行线程的数据很少，需要重新加载指令；（2）新线程的cache数据会覆盖老线程的数据，如果将来会再次覆盖老线程的数据，显然频繁覆盖增加很多切换开销。

避免资源超额是困难的，因为软件线程之于硬件线程的最佳比例取决于软件线程的执行频率，（比如一个程序从IO密集型变成计算密集型，执行频率是会改变的），而且比例还依赖上下文切换的开销以及软件线程对于CPU cache的使用效率。此外，硬件线程的数量和CPU cache的速度取决于机器的体系结构，即使经过调校，软件比例在某一种机器平台取得较好效果，换一个其他类型的机器这个调校并不能提供较好效果的保证。

而使用`std::async`可以将调校最优比例这件事隐藏于标准库中，在应用层面不需过多考虑

```
auto fut = std::async(doAsyncWork);  // onus of thread mgmt is
                                                                // on
implement of
                                                                // the
Standard Library
```

这种调用方式将线程管理的职责转交给C++标准库的开发者。举个例子，这种调用方式会减少抛出资源超额的异常，为何这么说调用`std::async`并不保证开启一个新的线程，只是提供了执行函数的保证，具体是否创建新的线程来运行此函数，取决于具体实现，比如可以通过调度程序来将`AsyncWork`运行在等待此函数结果的线程上，调度程序的合理性决定了系统是否会抛出资源超额的异常，但是这是库开发者需要考虑的事情了。

如果考虑自己实现在等待结果的线程上运行输出结果的函数，之前提到了可能引出负载不均衡的问题，`std::async`运行时的调度程序显然比开发者更清楚调度策略的制定，因为运行时调度程序管理的是所有执行过程，而不仅仅个别开发者运行的代码。

如果在GUI程序中使用`std::async`会引起响应变慢的问题，还可以通过`std::launch::async`向`std::async`传递调度策略来保证运行函数在不同的线程上执行。

最前沿的线程调度算法使用线程池来避免资源超额的问题，并且通过窃取算法来提升了跨硬件核心的负载均衡。C++标准实际上并不要求使用线程池或者`work-stealing`算法，而且这些技术的实现难度可能比你想象中更有挑战。不过，库开发者在标准库实现中采用了这些前沿的技术，这使得采用基于任务的方式编程的开发者在这些技术发展中持续获得回报，相反如果开发者直接使用`std::thread`编程，处理资源耗竭，负责均衡问题的责任就压在了应用开发者身上，更不说如何使得开发方案跨平台使用。

对比基于线程的开发方式，基于任务的设计为开发者避免了线程管理的痛苦，并且自然提供了一种获取异步执行的结果的方式。当然，仍然存在一些场景直接使用`std::thread`会更有优势：

- **需要访问非常基础的线程API**。C++并发API通常是通过操作系统提供的系统级API(pthreads 或者 windows threads)来实现的，系统级API通常会提供更加灵活的操作方式，举个例子，C++并发API没有线程优先级和affinities的概念。为了提供对底层系统级线程API的访问，`std::thread`对象提供了`native_handle`的成员函数，而在高层抽象的比如`std::futures`没有这种能力。

- **需要优化应用的线程使用**。举个例子，只在特定系统平台运行的软件，可以调教地比使用C++并行API更好的程序性能。
- **需要实现C++并发API之外的线程技术**。举例来说，自行实现线程池技术。

这些都是在应用开发中并不常见的例子，大多数情况，开发者应该优先采用基于任务的编程方式。

记住

- `std::thread`API不能直接访问异步执行的结果，如果执行函数有异常抛出，代码会终止执行
- 基于线程的编程方式关于解决资源超限，负载均衡的方案移植性不佳
- 基于任务的编程方式`std::async`会默认解决上面两条问题

Item 36:Specify std::launch::async if asynchronicity is essential

Item36： 确保在异步为必须时，才指定 `std::launch::async`

当你调用`std::async`执行函数时（或者其他可调用对象），你通常希望异步执行函数。但是这并不一定是你想要`std::async`执行的操作。你确实通过`std::async`launch policy（译者注：这里没有翻译）要求执行函数，有两种标准policy，都通过`std::launch`域的枚举类型表示（参见Item10关于枚举的更多细节）。假定一个函数**f**传给`std::async`来执行：

- **`std::launch::async`的launch policy**意味着f必须异步执行，即在不同的线程
- **`std::launch::deferred`的launch policy**意味着f仅仅在当调用`get`或者`wait`要求`std::async`的返回值时才执行。这表示f推迟到被求值才延迟执行（译者注：异步与并发是两个不同概念，这里侧重于惰性求值）。当`get`或`wait`被调用，f会同步执行，即调用方停止直到f运行结束。如果`get`和`wait`都没有被调用，f将不会被执行

有趣的是，`std::async`的默认launch policy是以上两种都不是。相反，是求或在一起的。下面的两种调用含义相同

```
auto fut1 = std::async(f); // run f using default launch policy
auto fut2 = std::async(std::launch::async | std::launch::deferred, f); // run f
either async or defered
```

因此默认策略允许f异步或者同步执行。如同Item 35中指出，这种灵活性允许`std::async`和标准库的线程管理组件（负责线程的创建或销毁）避免超载。这就是使用`std::async`并发编程如此方便的原因。

但是，使用默认启动策略的`std::async`也有一些有趣的影响。给定一个线程t执行此语句：

```
auto fut = std::async(f); // run f using default launch policy
```

- 无法预测f是否会与t同时运行，因为f可能被安排延迟运行
- 无法预测f是否会在调用`get`或`wait`的线程上执行。如果那个线程是t，含义就是无法预测f是否也在线程t上执行
- 无法预测f是否执行，因为不能确保`get`或者`wait`会被调用

默认启动策略的调度灵活性导致使用线程本地变量比较麻烦，因为这意味着如果f读写了线程本地存储（thread-local storage, TLS），不可能预测到哪个线程的本地变量被访问：

```
auto fut = std::async(f); // TLS for f possibly for independent thread, but
possibly for thread invoking get or wait on fut
```

还会影响到基于超时机制的wait循环，因为在task的`wait_for`或者`wait_until`调用中（参见Item 35）会产生延迟求值（`std::launch::deferred`）。意味着，以下循环看似应该终止，但是实际上永远运行：

```
using namespace std::literals; // for C++14 duration suffixes; see Item 34
void f()
{
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);
while (fut.wait_for(100ms) != std::future_status::ready)
{ // loop until f has finished running... which may never happen!
  ...
}
```

如果f与调用`std::async`的线程同时运行（即，如果为f选择的启动策略是`std::launch::async`），这里没有问题（假定f最终执行完毕），但是如果f是延迟执行，`fut.wait_for`将总是返回`std::future_status::deferred`。这表示循环会永远执行下去。

这种错误很容易在开发和单元测试中忽略，因为它可能在负载过高时才能显现出来。当机器负载过重时，任务推迟执行才最有可能发生。毕竟，如果硬件没有超载，没有理由不安排任务并发执行。

修复也是很简单的：只需要检查与`std::async`的future是否被延迟执行即可，那样就会避免进入无限循环。不幸的是，没有直接的方法来查看future是否被延迟执行。相反，你必须调用一个超时函数----比如`wait_for`这种函数。在这个逻辑中，你不想等待任何事，只想查看返回值是否`std::future_status::deferred`，如果是就使用0调用`wait_for`来终止循环。

```
auto fut = std::async(f);
if (fut.wait_for(0s) == std::future_status::deferred) { // if task is deferred
  ... // use wait or get on fut to call f synchronously
}
else { // task isn't deferred
  while(fut.wait_for(100ms) != std::future_status::ready) { // infinite loop not
possible(assuming f finished)
    ... // task is neither deferred nor ready, so do concurrent word until it's
ready
  }
}
```

这些各种考虑的结果就是，只要满足以下条件，`std::async`的默认启动策略就可以使用：

- task不需要和执行`get or wait`的线程并行执行
- 不会读写线程的线程本地变量
- 可以保证在`std::async`返回的将来会调用`get or wait`，或者该任务可能永远不会执行是可以接受的
- 使用`wait_for or wait_until`编码时考虑deferred状态

如果上述条件任何一个都满足不了，你可能想要保证`std::async`的任务真正的异步执行。进行此操作的方法是调用时，将`std::launch::async`作为第一个参数传递：

```
auto fut = std::async(std::launch::async, f); // launch f asynchronously
```

事实上，具有类似`std::async`行为的函数，但是会自动使用`std::launch::async`作为启动策略的工具也是很容易编写的，C++11版本如下：

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async, std::forward<F>(f), std::forward<Ts>
(params)...);
}
```

这个函数接受一个可调用对象和0或多个参数params然后完美转发（参见Item25）给`std::async`，使用`std::launch::async`作为启动参数。就像`std::async`一样，返回`std::future`类型。确定结果的类型很容易，因为类型特征`std::result_of`可以提供（参见Item 9 关于类型特征的详细表述）。

`reallyAsync`就像`std::async`一样使用：

```
auto fut = reallyAsync(f);
```

在C++14中，返回类型的推导能力可以简化函数的定义：

```
template<typename f, typename... Ts>
inline
auto
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async, std::forward<T>(f), std::forward<Ts>
(params)...);
}
```

这个版本清楚表明，`reallyAsync`除了使用`std::launch::async`启动策略之外什么也没有做。

需要记住的事

- `std::async`的默认启动策略是异步或者同步的
- 灵活性导致访问**thread_locals**的不确定性，隐含了task可能不会被执行的意思，会影响程序基于`wait`的超时逻辑
- 只有确实异步时才指定`std::launch::async`

Item 37：Make `std::threads` unjoinable on all paths

每个`std::thread`对象处于两个状态之一：*joinable or unjoinable*。*joinable*状态的`std::thread`对应于正在运行或者可能正在运行的异步执行线程。比如，一个blocked或者等待调度的`std::thread`是*joinable*，已运行结束的`std::thread`也可以认为是*joinable*

*unjoinable*的`std::thread`对象比如：

- **Default-constructed std::threads**。这种`std::thread`没有函数执行，因此无法绑定到具体的线程上
- **已经被moved的`std::thread`对象**。move的结果就是将`std::thread`对应的线程所有权转移给另一个`std::thread`
- **已经joined的`std::thread`**。在join之后，`std::thread`执行结束，不再对应于具体的线程
- **已经detached的`std::thread`**。detach断开了`std::thread`与线程之间的连接

（译者注：`std::thread`可以视作状态保存的对象，保存的状态可能也包括可调用对象，有没有具体的线程承载就是有没有连接）

`std::thread`的可连接性如此重要的原因之一就是当连接状态的析构函数被调用，执行逻辑被终止。比如，假定有一个函数`doWork`，执行过滤函数`filter`，接收一个参数`maxVal`。`doWork`检查是否满足计算所需的条件，然后通过使用0到maxVal之间的所有值过滤计算。如果进行过滤非常耗时，并且确定doWork条件是否满足也很耗时，则将两件事并发计算是很合理的。

我们希望为此采用基于任务的设计（参与Item 35），但是假设我们希望设置做过滤线程的优先级。Item 35阐释了需要线程的基本句柄，只能通过`std::thread`的API来完成；基于任务的API（比如futures）做不到。所以最终采用基于`std::thread`而不是基于任务

代码如下：

```
constexpr auto tenMillion = 10000000; // see Item 15 for constexpr
bool doWork(std::function<bool(int)> filter, int maxVal = tenMillion) // return
whether computation was performed; see Item2 for std::function
{
  std::vector<int> goodVals;
  std::thread t([&filter, maxVal, &goodVals]
                {
                  for (auto i = 0; i <= maxVal; ++i)
                  {
                    if (filter(i)) goodVals.push_back(i);
                  }
                });
  auto nh = t.native_handle(); // use t's native handle to set t's priority
  ...
  if (conditionsAreStatisfied()) {
    t.join(); // let t finish
    performComputation(goodVals); // computation was performed
    return true;
  }

  return false; // computation was not performed
}
```

在解释这份代码为什么有问题之前，看一下tenMillion的初始化可以在C++14中更加易读，通过单引号分隔数字：

```
constexpr auto tenMillion = 10'000'000; // C++14
```

还要指出，在开始运行之后设置t的优先级就像把马放出去之后再关上马厩门一样（译者注：太晚了）。更好的设计是在t为挂起状态时设置优先级（这样可以在执行任何计算前调整优先级），但是我不想你为这份代码考虑这个而分心。如果你感兴趣代码中忽略的部分，可以转到Item 39，那个Item告诉你如何以挂起状态开始线程。

返回`doWork`。如果`conditionsAreSatisfied()`返回真，没什么问题，但是如果返回假或者抛出异常，`std::thread`类型的`t`在`doWork`结束时会调用`t`的析构器。这造成程序执行中止。

你可能会想，为什么`std::thread`析构的行为是这样的，那是因为另外两种显而易见的方式更糟：

- **隐式join**。这种情况下，`std::thread`的析构函数将等待其底层的异步执行线程完成。这听起来是合理的，但是可能会导致性能异常，而且难以追踪。比如，如果`conditonAreStatisfied()`已经返回了假，`doWork`继续等待过滤器应用于所有值就很违反直觉。
- **隐式detach**。这种情况下，`std::thread`析构函数会分离其底层的线程。线程继续运行。听起来比join的方式好，但是可能导致更严重的调试问题。比如，在`doWork`中，`goodVals`是通过引用捕获的局部变量。可能会被lambda修改。假定，lambda的执行时异步的，`conditionsAreStatisfied()`返回假。这时，`doWork`返回，同时局部变量`goodVals`被销毁。堆栈被弹出，并在`doWork`的调用点继续执行线程

  某个调用点之后的语句有时会进行其他函数调用，并且至少一个这样的调用可能会占用曾经被`doWork`使用的堆栈位置。我们称为`f`，当`f`运行时，`doWork`启动的lambda仍在继续运行。该lambda可以在堆栈内存中调用`push_back`，该内存曾是`goodVals`，位于`doWork`曾经的堆栈位置。这意味着对`f`来说，内存被修改了，想象一下调试的时候痛苦

标准委员会认为，销毁连接中的线程如此可怕以至于实际上禁止了它（通过指定销毁连接中的线程导致程序终止）

这使你有责任确保使用`std::thread`对象时，在所有的路径上最终都是unjoinable的。但是覆盖每条路径可能很复杂，可能包括`return, continue, break, goto or exception`，有太多可能的路径。

每当你想每条路径的块之外执行某种操作，最通用的方式就是将该操作放入本地对象的析构函数中。这些对象称为RAII对象，通过RAII类来实例化。（RAII全称为 Resource Acquisition Is Initialization）。RAII类在标准库中很常见。比如STL容器，智能指针，`std::fstream`类等。但是标准库没有RAII的`std::thread`类，可能是因为标准委员会拒绝将`join`和`detach`作为默认选项，不知道应该怎么样完成RAII。

幸运的是，完成自行实现的类并不难。比如，下面的类实现允许调用者指定析构函数`join`或者`detach`：

```
class ThreadRAII {
public:
  enum class DtorAction{ join, detach }; // see Item 10 for enum class info
  ThreadRAII(std::thread&& t, DtorAction a): action(a), t(std::move(t)) {} // in
dtor, take action a on t
  ~ThreadRAII()
  {
    if (t.joinable()) {
      if (action == DtorAction::join) {
        t.join();
      } else {
```

```
        t.detach();
      }
    }
  }
  std::thread& get() { return t; } // see below
private:
  DtorAction action;
  std::thread t;
};
```

我希望这段代码是不言自明的，但是下面几点说明可能会有所帮助：

- 构造器只接受`std::thread`右值，因为我们想要move`std::thread`对象给`ThreadRAII`（再次强调，`std::thread`不可以复制）
- 构造器的参数顺序设计的符合调用者直觉（首先传递`std::thread`，然后选择析构执行的动作），但是成员初始化列表设计的匹配成员声明的顺序。将`std::thread`成员放在声明最后。在这个类中，这个顺序没什么特别之处，调整为其他顺序也没有问题，但是通常，可能一个成员的初始化依赖于另一个，因为`std::thread`对象可能会在初始化结束后就立即执行了，所以在最后声明是一个好习惯。这样就能保证一旦构造结束，所有数据成员都初始化完毕可以安全的异步绑定线程执行
- `ThreadRAII`提供了`get`函数访问内部的`std::thread`对象。这类似于标准智能指针提供的`get`函数，可以提供访问原始指针的入口。提供`get`函数避免了`ThreadRAII`复制完整`std::thread`接口的需要，因为着`ThreadRAII`可以在需要`std::thread`上下文的环境中使用
- 在`ThreadRAII`析构函数调用`std::thread`对象t的成员函数之前，检查t是否joinable。这是必须的，因为在unjoinbale的`std::thread`上调用`join or detach`会导致未定义行为。客户端可能会构造一个`std::thread`t，然后通过t构造一个`ThreadRAII`，使用`get`获取t，然后移动t，或者调用`join or detach`，每一个操作都使得t变为unjoinable
  如果你担心下面这段代码

  ```
  if (t.joinable()) {
    if (action == DtorAction::join) {
      t.join();
    } else {
      t.detach();
    }
  }
  ```

  存在竞争，因为在`t.joinable()`和`t.join or t.detach`执行中间，可能有其他线程改变了t为unjoinable，你的态度很好，但是这个担心不必要。`std::thread`只有自己可以改变`joinable or unjoinable`的状态。在`ThreadRAII`的析构函数中被调用时，其他线程不可能做成员函数的调用。如果同时进行调用，那肯定是有竞争的，但是不在析构函数中，是在客户端代码中试图同时在一个对象上调用两个成员函数（析构函数和其他函数）。通常，仅当所有都为const成员函数时，在一个对象同时调用两个成员函数才是安全的。

在`doWork`的例子上使用`ThreadRAII`的代码如下：

```
bool doWork(std::function<bool(int)> filter, int maxVal = tenMillion)
{
  std::vector<int> goodVals;
  ThreadRAII t(std::thread([&filter, maxVal, &goodVals] {
      for (auto i = 0; i <= maxVal; ++i) {
        if (filter(i)) goodVals.push_back(i);
      }
```

```
    }),
    ThreadRAII::DtorAction::join
  );
  auto nh = t.get().native_handle();
  ...
  if (conditonsAreStatisfied()) {
    t.get().join();
    performComputation(goodVals);
    return true;
  }
  return false;
}
```

这份代码中，我们选择在`ThreadRAII`的析构函数中异步执行`join`的动作，因为我们先前分析中，`detach`可能导致非常难缠的bug。我们之前也分析了`join`可能会导致性能异常（坦率说，也可能调试困难），但是在未定义行为（`detach`导致），程序终止（`std::thread`默认导致），或者性能异常之间选择一个后果，可能性能异常是最好的那个。

哎，Item 39表明了使用`ThreadRAII`来保证在`std::thread`的析构时执行`join`有时可能不仅导致程序性能异常，还可能导致程序挂起。“适当”的解决方案是此类程序应该和异步执行的lambda通信，告诉它不需要执行了，可以直接返回，但是C++11中不支持可中断线程。可以自行实现，但是这不是本书讨论的主题。（译者注：关于这一点，C++ Concurrency in Action 的section 9.2 中有详细讨论，也有中文版出版）

Item 17说明因为`ThreadRAII`声明了一个析构函数，因此不会有编译器生成移动操作，但是没有理由`ThreadRAII`对象不能移动。所以需要我们显式声明来告诉编译器自动生成：

```
class ThreadRAII {
public:
  enum class DtorAction{ join, detach }; // see Item 10 for enum class info
  ThreadRAII(std::thread&& t, DtorAction a): action(a), t(std::move(t)) {} // in
dtor, take action a on t
  ~ThreadRAII()
  {
    if (t.joinable()) {
      if (action == DtorAction::join) {
        t.join();
      } else {
        t.detach();
      }
    }
  }

  ThreadRAII(ThreadRAII&&) = default;
  ThreadRAII& operator=(ThreadRAII&&) = default;
  std::thread& get() { return t; } // see below
private:
  DtorAction action;
  std::thread t;
};
```

需要记住的事

- 在所有路径上保证`thread`最终是unjoinable
- 析构时`join`会导致难以调试的性能异常问题
- 析构时`detach`会导致难以调试的未定义行为
- 声明类数据成员时，最后声明`std::thread`类型成员

Item 38：Be aware of varying thread handle destructor behavior.

Item 38：关注不同线程句柄的析构行为

Item 37中说明了joinable的`std::thread`对应于可执行的系统线程。non-defered任务的`future`（参见Item 36）与系统线程有相似的关系。因此，可以将`std::thread`对象和`future`对象都视作系统线程的句柄。

从这个角度来说，有趣的是`std::thread`和`futures`在析构时有相当不同的行为。在Item 37中说明，joinable的`std::thread`析构会终止你的程序，因为两个其他的替代选择--隐式`join`或者隐式`detach`都是更加糟糕的。但是，`futures`的析构表现有时就像执行了隐式`join`，有时又是隐式执行了`detach`，有时又没有执行这两个选择。永远不会造成程序终止。这个线程句柄多种表现值得研究一下。

我们可以观察到实际上`future`是通信信道的一端（被调用者通过该信道将结果发送给调用者）。被调用者（通常是异步执行）将计算结果写入通信信道中（通过`std::promise`对象），调用者使用`future`读取结果。你可以想象成下面的图示，虚线表示信息的流动方向：

但是被调用者的结果存储在哪里？被调用者会在调用者`get`相关的`future`之前执行完成，所以结果不能存储在被调用者的`std::promise`。这个对象是局部的，当被调用者执行结束后，会被销毁。

结果同样不能存储在调用者的`future`，因为`std::future`可能会被用来创建`std::shared_future`(这会将被调用者的结果所有权从`std::future`转移给`std::shared_future`)，而`std::shared_future`在`std::future`被销毁之后被复制很多次。鉴于不是所有的结果都可以被拷贝（有些只能移动）和结果的声明周期与最后一个引用它的`future`一样长，哪个才是被调用者用来存储结果的？这两个问题。

因为与被调用者关联的对象和调用者关联的对象都不适合存储这个结果，必须存储在两者之外的位置。此位置称为共享状态（*shared state*）。共享状态通常是基于堆的对象，但是标准并未指定其类型、接口和实现。标准库的作者可以通过任何他们喜欢的方式来实现共享状态。

我们可以想象调用者，被调用者，共享状态之间关系如下图，虚线还是表示信息的流控方向：

共享状态的存在非常重要，因为`future`的析构行为--这个Item的话题---取决于关联`future`的共享状态。

- Non-defered任务（启动参数为`std::launch::async`）的最后一个关联共享状态的`future`析构函数会在任务完成之前block住。本质上，这种`future`的析构对执行异步任务的线程做了隐式的`join`。
- `future`其他对象的析构简单的销毁。对于异步执行的任务，就像对底层的线程执行`detach`。对于defered任务的最后一种`future`，意味着这个defered任务永远不会执行了。

这些规则听起来好复杂。我们真正要处理的是一个简单的“正常”行为以及一个单独的例外。正常行为是`future`析构函数销毁`future`。那意味着不`join`也不`detach`，只销毁`future`的数据成员（当然，还做了另一件事，就是对于多引用的共享状态引用计数减一。）

正常行为的例外情况仅在同时满足下列所有情况下才会执行：

- 关联`future`的共享状态是被调用了`std::async`创建的
- 任务的启动策略是`std::launch::async`（参见Item 36），原因是运行时系统选择了该策略，或者在对`std::async`的调用中指定了该策略。
- `future`是关联共享状态的最后一个引用。对于`std::future`，情况总是如此，对于`std::shared_future`，如果还有其他的`std::shared_future`引用相同的共享状态没有销毁，就不是。

只有当上面的三个条件都满足时，`future`的析构函数才会表现“异常”行为，就是在异步任务执行完之前block住。实际上，这相当于运行`std::async`创建的任务的线程隐式`join`。

通常会听到将这种异常的析构函数行为称为"Futures from std::async block in their destructors"。作为近似描述没有问题，但是忽略了原因和细节，现在你已经知道了其中三味。

你可能想要了解更加深入。比如“为什么会有这样的规则”（译者注：这里的问题是意译，原文重复了问题本身），这很合理。据我所知，标准委员会希望避免这个问题与隐式`detach`（参见Item 37）相关联，但是不想采取强制程序终止这种激进的方案（因此搞了`join`，同样参见Item 37），所以妥协使用隐式`join`。这个决定并非没有争议，并且认真讨论过在C++14中放弃这种行为。最后，决定先不改变，所以C++11和C++14中这里的行为是一致的。

没有API来提供`future`是否指向`std::async`调用产生的共享状态，因此给定一个`std::future`对象，无法判断是不是会在析构函数block等待异步任务的完成。这就产生了有意思的事情：

```
// this container might block in its dtor, because one or more contained futures
could refer to a shared state for a non-deferred task launched via std::async
std::vector<std::future<void>> futs; // see Item 39 for info on std::future<void>
class Widget // Widget objects might block in their dtors
{
public:
  ...
private:
  std::shared_future<double> fut;
};
```

当然，如果你有办法知道给定的`future`不满足上面条件的任意一条，你就可以确定析构函数不会执行“异常”行为。比如，只有通过`std::async`创建的共享状态才有资格执行“异常”行为，但是有其他创建共享状态的方式。一种是使用`std::packaged_task`，一个`std::packaged_task`对象准备一个函数（或者其他可调用对象）来异步执行，然后将其结果放入共享状态中。然后通过`std::packaged_task`的`get_future`函数获取有关该共享状态的信息：

```
int calcValue(); // func to run
std::packaged_task<int()> pt(calcValue); // wrap calcValue so it can run
asynchrously
auto fut = pt.get_future(); // get future for pt
```

此时，我们知道`future`没有关联`std::async`创建的共享状态，所以析构函数肯定正常方式执行。

一旦被创建，`std::packaged_task`类型的pt可能会在线程上执行。（译者注：后面有些啰嗦的话这里不完整翻译。。大意就是可以再次使用`std::async`来执行，但是那就不用`std::packaged_task`了）

`std::packaged_task`不能拷贝，所以当`pt`被传递给`std::thread`时是右值传递（通过move，参见Item 23）：

```
std::thread t(std::move(pt)); // run pt on t
```

这个例子是你对于`future`的析构函数的正常行为有一些了解，但是将这些语句放在一个作用域的语句块里更容易：

```
{ // begin block
  std::packaged_task<int()> pt(calcValue);
    auto fut = pt.get_future();
  std::thread t(std::move(pt));
  ...
} // end block
```

此处最有趣的代码是在创建`std::thread`对象t之后的"..."。"..."有三种可能性：

- 对t不做什么。这种情况，t会在语句块结束joinable，这会使得程序终止（参见Item 37）
- 对t调用`join`。这种情况，不需要fut的析构函数block，因为`join`被显式调用了
- 对t调用`detach`。这种情况，不需要在fut的析构函数执行`detach`，因为显式调用了

换句话说，当你有一个关联了`std::packaged_task`创建的共享状态的`future`时，不需要采取特殊的销毁策略，通常你会代码中做这些。

需要记住的事

- `future`的正常析构行为就是销毁`future`本身的成员数据
- 最后一个引用`std::async`创建共享状态的`future`析构函数会在任务结束前block

Item 39: Consider void futures for one-shot event communication

Item 39:对于一次性事件通信考虑使用无返回futures

有时，一个任务通知另一个异步执行的任务发生了特定的事件很有用，因为第二个任务要等到特定事件发生之后才能继续执行。事件也许是数据已经初始化，也许是计算阶段已经完成，或者检测到重要的传感器值。这种情况，什么是线程间通信的最佳方案?

一个明显的方案就是使用条件变量（condvar）。如果我们将检测条件的任务称为检测任务，对条件作出反应的任务称为反应任务，策略很简单：反应任务等待一个条件变量，检测任务在事件发生时改变条件变量。代码如下：

```
std::condition_variable cv; // condvar for event
std::mutex m; // mutex for use with cv
```

检测任务中的代码不能再简单了：

```
... // detect event
cv.notify_one(); // tell reacting task
```

如果有多个反应任务需要被通知，使用`notify_all()`代替`notify_one()`，但是这里，我们假定只有一个反应任务需要通知。

反应任务对的代码稍微复杂一点，因为在调用`wait`条件变量之前，必须通过`std::unique_lock`对象使用互斥锁`mutex`来同步（lock a mutex是等待条件变量的经典实现。`std::unique_lock`是C++11的易用API），代码如下：

```
... // propare to react
{ // open critical section
  std::unique_lock<std::mutex> lk(m); // lock mutex
  cv.wait(lk); // wati for notify; this isn't correct!
  ... // react to event(m is blocked)
} // close crit. section; unlock m via lk's dtor
... // continue reacting (m now unblocked)
```

这份代码的第一个问题就是有时被称为*code smell*：即使代码正常工作，但是有些事情也不是很正确。这种问题源自于使用互斥锁。互斥锁被用于保护共享数据的访问，但是可能检测任务和反应任务不会同时访问共享数据，比如说，检测任务会初始化一个全局数据结构，然后给反应任务用，如果检测任务在初始化之后不会再访问这个数据，而反应任务在初始化之前不会访问这个数据，就不存在数据竞争，也就没有必要使用互斥锁。但是条件变量必须使用互斥锁，这就留下了令人不适的设计。

即使你忽略了这个问题，还有两个问题需要注意：

- **如果检测任务在反应任务`wait`之前通知条件变量，反应任务会挂起**。为了能使条件变量唤醒另一个任务，任务必须等待在条件变量上。如果检测任务在反应任务`wait`之前就通知了条件变量，反应任务就会丢失这次通知，永远不被唤醒
- **`wait`语句虚假唤醒**。线程API的存在一个事实（不只是C++）即使条件变量没有被通知，也可能被虚假唤醒，这种唤醒被称为*spurious wakeups*。正确的代码通过确认条件变量进行处理，并将其作为唤醒后的第一个操作。C++条件变量的API使得这种问题很容易解决，因为允许lambda（或者其他函数对象）来测试等待条件。因此，可以将反应任务这样写：

```
cv.wait(lk,
       [] { return whether the evet has occurred; });
```

要利用这个能力需要反应任务能够确定其等待的条件为真。但是我们考虑的情况下，它正在等待的条件是检测线程负责识别的事件。反应线程可能无法确定等待的事件是否已发生。这就是为什么需要一个条件变量的原因

在很多情况下，使用条件变量进行任务通信非常合适，但是也有不那么合适的情况。

对于很多开发者来说，他们的下一个诀窍是共享的boolean标志。flag被初始化为false。当检测线程识别到发生的事件，将flag设置为true；

```
std::atomic<bool> flag(false); // shared flag; see Item 40 for std::atomic
... // detect event
flag = true; // tell reacting task
```

就其本身而言，反应线程轮询该flag。当发现flag被设置为true，它就知道等待的事件已经发生了：

```
... // prepare
while(!flag); // wait for event
... // react to event
```

这种方法不存在基于条件变量的缺点。不需要互斥锁，在反应变量设置flag为true之前轮询不会出现问题，并且不会出现虚假唤醒。好，好，好。

不好的一点是反应任务中轮询的开销。在等待flag为设置为true的时候，任务基本被锁住了，但是一直占用cpu。这样，反应线程占用了可能给另一个任务使用的硬件线程，每次启动或者完成的时间片都会产生上下文切换的开销，并且保持CPU核心运行（否则可能会停下来省电）。一个真正blocked的任务不会这样，这也是基于条件变量的优点，因为等待调用中的任务真的blocked。

将条件变量和flag的设计组合起来很常用。一个flag表示是否发生了感兴趣的事件，但是通过互斥锁同步了对该flag的访问。因为互斥锁阻止并发该flag，所以如Item 40所述，不需要将flag设置为`std::atomic`。一个简单的bool类型就可以，检测任务代码如下：

```
std::conditon_variable cv;
std::mutex m;
bool flag(false); // not std::atomic
... // detect event
{
  std::lock_guard<std::mutex> g(m); // lock m via g's ctor
  flag = true; // tell reacting task(part 1)
} // unlock m via g's dtor
cv.notify_one(); // tell reacting task (part 2)
```

反应任务代码如下：

```
... // prepare to react
{
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk, []{ return flag; }); // use lambda to avoid spurious wakeups
  ... // react to event (m is blocked)
}
... // continue reacting (m now unblocked)
```

这份代码解决了我们一直讨论的问题。无论是否反应线程在调用`wait`之前还是之后检测线程对条件变量发出通知都可以工作，即使出现了虚假唤醒也可以工作，而且不需要轮询。但是仍然有些古怪，因为检测任务通过奇怪的方式与反应线程通信。（译者注：下面的话挺绕的，可以参考原文）检测任务通知条件变量告诉反应线程等待的事件可能发生了，反应线程必须通过检查flag来确保事件发生了。检测线程设置flag来告诉反应线程事件确实发生了，但是检测线程首先需要通知条件变量唤醒反应线程来检查flag。这种方案是可以工作的，但是不太优雅。

一个替代方案是让反应任务通过在检测任务设置的future上`wait`来避免使用条件变量，互斥锁和flag。这可能听起来也是个古怪的方案。毕竟，Item 38中说明了future代表了从被调用方（通常是异步的）到调用方的通信的接收端，这里的检测任务和反应任务没有调用-被调用的关系。然而，Item 38中也说说明了通信新到发送端是`std::promise`，接收端是`future`不只能用在调用-被调用场景。这样的通信信道可以被在任何你需要从程序一个地方传递到另一个地方的场景。这里，我们用来在检测任务和反应任务之间传递信息，传递的信息就是感兴趣的事件是否已发生。

方案很简单。检测任务有一个`std::promise`对象（通信信道的写入），反应任务有对应的`std::future`（通信信道的读取）。当反应任务看到事件已经发生，设置`std::promise`对象（写入到通信信道）。同时，反应任务在`std::future`上等待。`wait`会锁住反应任务直到`std::promise`被设置。

现在，`std::promise和futures(std::future and std::shared_future)`都是需要参数类型的模板。参数表明被传递的信息类型。在这里，没有数据被传递，只需要让反应任务知道`future`已经被设置了。我们需要的类型是表明在`std::promise`和`futures`之间没有数据被传递。所以选择`void`。检测任务使用`std::promise<void>`，反应任务使用`std::future<void> or std::shared_future<void>`。当感兴趣的事件发生时，检测任务设置`std::promise<void>`，反应任务在`futures`上等待。即使反应任务不接收任何数据，通信信道可以让反应任务知道检测任务是否设置了`void`数据（通过对`std::promise<void>`调用`set_value`）。

所以，代码如下：

```
std::promise<void> p; // promise for communications channel
```

检测任务代码如下：

```
... // detect event
p.set_value(); // tell reacting task
```

反应任务代码如下：

```
... // prepare to react
p.get_future().wait(); // wait on future corresponding to p
... //react to event
```

像使用flag的方法一样，此设计不需要互斥锁，无论检测任务是否在反应任务等待之前设置`std::promise`都可以工作，并且不受虚假唤醒的影响（只有条件变量才容易受到此影响）。与基于条件变量的方法一样，反应任务真是被blocked，不会一直占用系统资源。是不是很完美？

当然不是，基于`future`的方法没有了上述问题，但是有其他新的问题。比如，Item 38中说明，`std::promise 和 future`之间有共享状态，并且共享状态是动态分配的。因此你应该假定此设计会产生基于堆的分配和释放开销。

也许更重要的是，`std::promise`只能设置一次。`std::promise 与 future`之间的通信是一次性的：不能重复使用。这是与基于条件变量或者flag的明显差异，条件变量可以被重复通知，flag也可以重复清除和设置。

一次通信可能没有你想象中那么大的限制。假定你想创建一个挂起的线程以避免想要使用一个线程执行程序的时候的线程创建的开销。或者你想在线程运行前对其进行设置，包括优先级和core affinity。C++并发API没有提供这种设置能力，但是提供了`native_handle()`获取原始线程的接口（通常获取的是POXIC或者Windows的线程），这些低层次的API使你可以对线程设置优先级和 core affinity。

假设你仅仅想要挂起一次线程（在创建后，运行前），使用`void future`就是一个方案。代码如下：

```
std::promise<void> p;
void react(); // func for reacting task
void detect() // func for detecting task
{
  st::thread t([] // create thread
               {
                 p.get_future().wait(); // suspend t until future is set
                 react();
               });
  ... // here, t is suspended prior to call to react
  p.set_value(); // unsuspend t (and thus call react)
  ... // do additional work
  t.join(); // make t unjoinable(see Item 37)
}
```

因为根据Item 37说明，对于检测任务所有路径`thread`t都要是unjoinable的，所以使用建议的`ThreadRAII`。代码如下：

```
void detect()
{
    ThreadRAII tr(
    std::thread([]
                {
                  p.get_future().wait();
                  react();
                }),
    ThreadRAII::DtorAction::join  // risky ! (see below)
  );
  ... // thread inside tr is suspended here
  p.set_value(); // unsuspend thread inside tr
  ... // do additional work
}
```

这样看起来安全多了。问题在于第一个"..."区域（注释了thread inside tr is suspended here），如果异常发生，`p.set_value()`永远不会调用，这意味着`lambda`中的`wait`永远不会返回，即lambda不会结束，问题就是，因为RAII对象tr再析构函数中join。换句话说，如果在第一个"..."中发生了异常，函数挂起，因为tr的析构不会被调用。

有很多方案解决这个问题，但是我把这个经验留给读者（译者注：http://scottmeyers.blogspot.com/2013/12/threadraii-thread-suspension-trouble.html 中这个问题的讨论）。这里，我只想展示如何扩展原始代码（不使用RAII类）使其挂起然后取消挂起，这不仅是个例，是个通用场景。简单概括，关键就是在反应任务的代码中使用`std::shared_future`代替`std::future`。一旦你知道`std::future`的`share`成员函数将共享状态所有权转移到`std::shared_future`中，代码自然就写出来了。唯一需要注意的是，每个反应线程需要处理自己的`std::shared_future`副本，该副本引用共享状态，因此通过`share`获得的`shared_future`要被lambda按值捕获：

```
std::promise<void> p; // as before
void detect() // now for multiple reacting tasks
```

```
{
  auto sf = g.get_future().share(); // sf's type is std::shared_future<void>
  std::vector<std::thread> vt; // container for reacting threads
  for (int i = 0;  i < threadsToRun; ++i)
  {
    vt.emplace_back([sf]{
      sf.wait();
      react();
    }); // wait on local copy of sf; see Item 43 for info on emplace_back
  }
  ... // detect hangs if this "..." code throws !
  p.set_value(); // unsuspend all threads
  ...
  for (auto& t : vt) {
    t.join(); // make all threads unjoinable: see Item2 for info on "auto&"
  }
}
```

这样`future`就可以达到预期效果了，这就是你应该将其应用于一次通信的原因。

需要记住的事

- 对于简单的事件通信，条件变量需要一个多余的互斥锁，对检测和反应任务的相对进度有约束，并且需要反应任务来验证事件是否已发生
- 基于flag的设计避免的上一条的问题，但是不是真正的挂起反应任务
- 组合条件变量和flag使用，上面的问题都解决了，但是逻辑不让人愉快
- 使用`std::promise`和`future`的方案，要考虑堆内存的分配和销毁开销，同时有只能使用一次通信的限制

Item 40：Use std::atomic for councurrency, volatile for special memory

Item 40: 当需要并发时使用`std::atomic`，特定内存才使用`volatile`

可伶的`volatile`。如此令人迷惑。本不应该出现在本章节，因为它没有关于并发的能力。但是在其他编程语言中（比如，Java和C#），`volatile`是有并发含义的，即使在C++中，有些编译器在实现时也将并发的某种含义加入到了`volatile`关键字中。因此在此值得讨论下关于`volatile`关键字的含义以消除异议。

开发者有时会混淆`volatile`的特性是`std::atomic`（这确实本节的内容）的模板。这种模板的实例化（比如，`std::atomic<int> , std::atomic<bool>, std::atomic<Widget*>`等）给其他线程提供了原子操作的保证。一旦`std::atomic`对象被构建，在其上的操作使用特定的机器指令实现，这比锁的实现更高效。

分析如下使用`std::atmoic`的代码：

```
std::atomic<int> ai(0); // initialize ai to 0
ai = 10; // atomically set ai to 10
std::cout << ai; // atomically read ai's value
++ai; //atomically increment ai to 11
--ai; // atomically decrement ai to 10
```

在这些语句执行过程中，其他线程读取`ai`，只能读取到0，10，11三个值其中一个。在没有其他线程修改`ai`情况下，没有其他可能。

这个例子中有两点值得注意。**首先**，在`std::cout << ai;`中，`std::atomic`只保证了对`ai`的读取时原子的。没有保证语句的整个执行是原子的，这意味着在读取`ai`与将其通过`<<`操作符写入到标准输出之间，另一个线程可能会修改`ai`的值。这对于这个语句没有影响，因为`<<`操作符是按值传递参数的（所以输出就是读取到的`ai`的值），但是重要的是要理解原子性的范围只保证了读取是原子的。

第二点值得注意的是最后两条语句---关于`ai`的加减。他们都是 read-modify-write（RMW）操作，各自原子执行。这是`std::atomic`类型的最优的特性之一：一旦`std::atomic`对象被构建，所有成员函数，包括RMW操作，对于其他线程来说保证原子执行。

相反，使用`volatile`在多线程中不保证任何事情：

```
volatile int vi(0); // initalize vi to 0
vi = 10; // set vi to 10
std::cout << vi;  // read vi's value
++vi; // increment vi to 11
--vi; // decrement vi to 10
```

代码的执行过程中，如果其他线程读取`vi`，可能读到任何值，比如-12，68，4090727。这份代码就是未定义的，因为这里的语句修改`vi`，同时其他线程读取，这就是有没有`std::atomic`或者互斥锁保护的对于内存的同时读写，这就是数据竞争的定义。

为了举一个关于在多线程程序中`std::atomic`和`volatile`表现不同的恰当例子，考虑这样一个加单的计数器，同时初始化为0：

```
std::atomic<int> ac(0);
volatile int vc(0);
```

然后我们在两个同时运行的线程中对两个计数器计数：

```
/*--------- Thread1 ---------*/      /*---------- Thread2 -----------*/
          ++ac;                                  ++ac;
                    ++vc;                                  ++vc;
```

当两个线程执行结束时，`ac`的值肯定是2，以为每个自增操作都是原子的。另一方面，`vc`的值，不一定是2，因为自增不是原子的。每个自增操作包括了读取`vc`的值，增加读取的值，然后将结果写回到`vc`。这三个操作对于`volatile`修饰的整形变量不能保证原子执行，所有可能是下面的执行顺序：

1. Thread1 读取`vc`的值，是0
2. Thread2读取`vc`的值，还是0
3. Thread1 将0加1，然后写回到`vc`
4. Thread2将0加1，然后写回到vc

`vc`的最后结果是1，即使看起来自增了两次。

不仅只有这一种执行顺序的可能，`vc`的最终结果是不可预测的，因为`vc`会发生数据竞争，标准规定数据竞争的造成的未定义行为表示编译器生成的代码可能是任何逻辑，当然，编译器不会利用这种行为来作恶。但是只有在没有数据竞争的程序中编译器的优化才有效，这些优化在存在数据竞争的程序中会造成异常和不可预测的行为。

RMW操作不是仅有的`std::atomic`在并发中有效而`volatile`无效的例子。假定一个任务计算第二个任务需要的重要值。当第一个任务完成计算，必须传递给第二个任务。Item 39表明一种使用`std::atomic<bool>`的方法来使第一个任务通知第二个任务计算完成。代码如下：

```
std::atomic<bool> valVailable(false);
auto imptValue = coputeImportantValue(); // compute  value
valAvailable = true; // tell other task it's vailable
```

人类读这份代码，能看到在`valAvailable`赋值true之前对`imptValue`赋值是重要的顺序，但是所有编译器看到的是一对没有依赖关系的赋值操作。通常来说，编译器会被允许重排这对没有关联的操作。这意味着，给定如下顺序的赋值操作：

```
a = b;
x = y;
```

编译器可能重排为如下顺序：

```
x = y;
a = b;
```

即使编译器没有重排顺序，底层硬件也可能重排，因为有时这样代码执行更快。

然而，`std::atomic`会限制这种重排序，并且这样的限制之一是，在源代码中，对`std::atomic`变量写之前不会有任何操作。这意味对我们的代码

```
auto impatValue = computeImportantValue();
valVailable = true;
```

编译器不仅要保证赋值顺序，还要保证生成的硬件代码不会改变这个顺序。结果就是，将`valAvaliable`声明为`std::atomic`确保了必要的顺序---- 其他线程看到`imptValue`值保证`valVailable`设为true之后。

声明为`volatile`不能保证上述顺序：

```
volatile bool valAvaliable(false);
auto imptValue = computeImportantValue();
valAvailable = true;
```

这份代码编译器可能将赋值顺序对调，也可能在生成机器代码时，其他核心看到`valVailable`更改在`imptValue`之前。

“正常”内存应该有这个特性，在写入值之后，这个值会一直保证直到被覆盖。假设有这样一个正常的int

```
int x;
```

编译器看到下列的操作序列：

```
auto y = x; // read x
y = x; // read x again
```

编译器可通过忽略对y的一次赋值来优化代码，因为初始化和赋值是冗余的。

正常内存还有一个特征，就是如果你写入内存没就不会读，再次吸入，第一次写就可以被忽略，因为肯定会被覆盖。给出下面的代码：

```
x = 10; // write x
x = 20; // write x again
```

编译器可以忽略第一次写入。这意味着如果写在一起：

```
auto y = x;
y = x;
x = 10;
x = 20;
```

编译器生成的代码是这样的：

```
auto y = x;
x = 20;
```

可能你会想睡会写这种重复读写的代码（技术上称为redundant loads 和 dead stores），答案是开发者不会直接写，至少我们不希望开发者这样写。但是在编译器执行了模板实例化，内联和一系列重排序优化之后，结果会出现多余的操作和无效存储，所以编译器需要摆脱这样的情况并不少见。

这种有话讲仅仅在内存表现正常时有效。“特殊”的内存不行。最常见的“特殊”内存是用来memory-mapped I/O的内存。这种内存实际上是与外围设备（比如外部传感器或者显示器，打印机，网络端口）通信，而不是读写（比如RAM）。这种情况下，再次考虑多余的代码：

```
auto y = x; // read x
y = x; // read x again
```

如果x的值是一个温度传感器上报的，第二次对于x的读取就不是多余的，因为温度可能在第一次和第二次读取之间变化。

类似的，写也是一样：

```
x = 10;
x = 20;
```

如果x与无线电发射器的控制端口关联，则代码时控制无线电，10和20意味着不同的指令。优化会更改第一条无线电指令。

`volatile`是告诉编译器我们正在处理"特殊"内存。意味着告诉编译器"不要对这块内存执行任何优化"。所以如果x对应于特殊内存，应该声明为`volatile`：

```
volatile int x;
```

带回我们原始代码：

```
auto y = x;
y = x; // can't be optimized away

x = 10; // can't be optimized away
x = 20;
```

如果x是内存映射（或者已经映射到跨进程共享的内存位置等），这正是我们想要的。

那么，在最后一段代码中，y是什么类型：int还是volatile int?

在处理特殊内存时，必须保留看似多余的读取或者无效存储的事实，顺便说明了为什么`std::atomic`不适合这种场景。`std::atomic`类型允许编译器消除此类冗余操作。代码的编写方式与使用`volatile`的方式完全不同，但是如果我们暂时忽略它，只关注编译器执行的操作，则可以说，

```
std::atomic<int> x;
auto y = x; // conceptually read x (see below)
y = x; // conceptually read x again(see below)

x = 10; // write x
y = 20; // write x again
```

原则上，编译器可能会优化为：

```
auto y = x; // conceptually read x
x = 20; // write x
```

对于特殊内存，显然这是不可接受的。

现在，就当他没有优化了，但是对于x是`std::atomic<int>`类型来说，下面的两条语句都编译不通过。

```
auto y = x; // error
y = x; // error
```

这是因为`std::atomic`类型的拷贝操作时被删除的（参见Item 11）。想象一下如果y使用x来初始化会发生什么。因为x是`std::atomic`类型，y的类型被推导为`std::atomic`（参见Item 2）。我之前说了`std::atomic`最好的特性之一就是所有成员函数都是原子的，但是为了执行从x到y的拷贝初始化是原子的，编译器不得不生成读取x和写入x为原子的代码。硬件通常无法做到这一点，因此`std::atomic`不支持拷贝构造。处于同样的原因，拷贝赋值也被delete了，这也是为什么从x赋值给y也编译失败。（移动操作在`std::atomic`没有显式声明，因此对于Item 17中描述的规则来看，`std::atomic`既不提移动构造器也不提供移动赋值能力）。

可以将x的值传递给y，但是需要使用`std::atomic`的`load`和`store`成员函数。`load`函数原子读取，`store`原子写入。要使用x初始化y，然后将x的值放入y，代码应该这样写：

```
std::atomic<int> y(x.load());
y.store(x.load());
```

这可以编译，但是可以清楚看到不是整条语句原子，而是读取写入分别原子化执行。

给出的代码，编译器可以通过存储x的值到寄存器代替读取两次来“优化”：

```
register = x.load(); // read x into register
std::atomic<int> y(register); // init y with register value
y.store(register); // store register value into y
```

结果如你所见，仅读取x一次，这是对于特殊内存必须避免的优化（这种优化不允许对`volatile`类型值执行）。

事情越辩越明：

- `std::atomic`用在并发程序中
- `volatile`用于特殊内存场景

因为`std::atomic`和`volatile`用于不同的目的，所以可以结合起来使用：

```
volatile std::atomic<int> vai; // operations on vai are atomic and can't be
optimized away
```

这可以用在比如`vai`变量关联了memory-mapped I/O内存并且用于并发程序的场景。

最后一点，一些开发者尤其喜欢使用`std::atomic`的`load`和`store`函数即使不必要时，因为这在代码中显式表明了这个变量不“正常”。强调这一事实并非没有道理。因为访问`std::atomic`确实会更慢一些，我们也看到了`std::atomic`会阻止编译器对代码执行顺序重排。调用`load`和`store`可以帮助识别潜在的可扩展性瓶颈。从正确性的角度来看，没有看到在一个变量上调用`store`来与其他线程进行通信（比如flag表示数据的可用性）可能意味着该变量在声明时没有使用`std::atomic`。这更多是习惯问题，但是，一定要知道`atomic`和`volatile`的巨大不同。

必须记住的事

- `std::atomic`是用在不使用锁，来使变量被多个线程访问。是用来编写并发程序的
- `volatile`是用在特殊内存的场景中，避免被编译器优化内存。

CHAPTER8 Tweaks

对于C++中的通用技术，总是存在适用场景。除了本章覆盖的两个例外，描述什么场景使用哪种通用技术通常来说很容易。这两个例外是传值（pass by value）和 emplacement。决定何时使用这两种技术受到多种因素的影响，本书提供的最佳建议是在使用它们的同时仔细考虑清楚，尽管它们都是高效的现代C++编程的重要角色。接下来的Items提供了是否使用它们来编写软件的所需信息。

Item41.Consider pass by value for copyable parameters that are cheap to move and always copied 如果参数可拷贝并且移动操作开销很低，总是考虑直接按值传递

有些函数的参数是可复制的。比如说，`addName`成员函数可以拷贝自己的参数到一个私有容器。为了提高效率，应该拷贝左值，移动右值。

```
class Widget {
public:
  void addName(const std::string& newName) {
    names.push_back(newName);
  }
  void addName(std::string&& newName) {
    names.push_back(std::move(newName));
  }
  ...
private:
  std::vector<std::string> names;
};
```

这是可行的，但是需要编写两个同名异参函数，这有点让人难受：两个函数声明，两个函数实现，两个函数文档，两个函数的维护。唉。

此外，你可能会担心程序的目标代码的空间占用，当函数都内联（inlined）的时候，会避免同时两个函数同时存在导致的代码膨胀问题，但是一旦存在没有被内联（inlined），目标代码就是出现两个函数。

另一种方法是使`addName`函数成为具有通用引用的函数模板:（参考Item24）

```
class Widget {
public:
  template<typename T>
  void addName(T&& newName) {
    names.push_back(std::forward<T>(newName));
  }
  ...
};
```

这减少了源代码的维护工作，但是通用引用会导致其他复杂性。作为模板，`addName`的实现必须放置在头文件中。在编译器展开的时候，可能会不止为左值和右值实例化为多个函数，也可能为`std::string`和可转换为`std::string`的类型分别实例化为多个函数（参考Item25）。同时有些参数类型不能通过通用引用传递（参考Item30），而且如果传递了不合法的参数类型，编译器错误会令人生畏。（参考Item27）

是否存在一种编写`addName`的方法（左值拷贝，右值移动），而且源代码和目标代码中都只有一个函数，避免使用通用模板这种特性？答案是是的。你要做的就是放弃你学习C++编程的第一条规则，就是用户定义的对象避免传值。像是`addName`函数中的`newName`参数，按值传递可能是一种完全合理的策略。

在我们讨论为什么对于`addName`中的`newName`参数按值传递非常合理之前，让我们来考虑如下实现：

```
class Widget {
public:
  void addName(std::string newName) {
    names.push_back(std::move(newName));
  }
  ...
}
```

该代码唯一可能令人困惑的部分就是`std::move`这里。`std::move`典型的应用场景是用在右值引用，但是在这里，我们了解到的信息：（1）`newName`是完全复制的传递进来的对象，换句话说，改变不会影响原值；（2）`newName`的最终用途就在这个函数里，不会再做他用，所以移动它不会影响其他代码。

事实就是我们只编写了一个`addName`函数，避免了源代码和目标代码的重复。我们没有使用通用引用的特性，不会导致头文件膨胀，odd failure cases(这里不知道咋翻译)，或者令人困惑的错误问题（编译）。但是这种设计的效率如何呢？按值传值会不会开销很大？

在C++98中，可以肯定的是，无论调用者如何调用，参数`newName`都是拷贝传递。但是在C++11中，`addName`就是左值拷贝，右值移动，来看如下例子：

```
Widget w;
...
std::string name("Bart");
w.addName(name); // call addName with lvalue
...
w.addName(name + "Jenne"); // call addName with rvalue
```

第一处调用，`addName`的参数是左值，因此是拷贝构造参数，就像在C++98中一样。第二处调用，参数是一个临时值，是一个右值，因此`newName`的参数是移动构造的。

就像我们想要的那样，左值拷贝，右值移动，优雅吧？

优雅，但是要牢记一些警示，回顾一下我们考虑过的三个版本的`addName`：

```
class Widget {  // Approach 1
public:
  void addName(const std::string& newName) {
    names.push_back(newName);
  }
  void addName(std::string&& newName) {
    names.push_back(std::move(newName));
  }
```

```
  ...
private:
  std::vector<std::string> names;
};

class Widget {  // Approach 2
public:
  template<typename T>
  void addName(T&& newName) {
    names.push_back(std::forward<T>(newName));
  }
  ...
};

class Widget { // Approach 3
public:
  void addName(std::string newName) {
    names.push_back(std::move(newName));
  }
  ...
};
```

本书将前两个版本称为“按引用方法”，因为都是通过引用传递参数，仍然考虑这两种调用方式：

```
Widget w;
...
std::string name("Bart");
w.addName(name); // call addName with lvalue
...
w.addName(name + "Jenne"); // call addName with rvalue
```

现在分别考虑三种实现中，两种调用方式，拷贝和移动操作的开销。会忽略编译器对于移动和拷贝操作的优化。

- **Overloading（重载）**：无论传递左值还是传递右值，调用都会绑定到一种`newName`的引用实现方式上。拷贝和复制零开销。左值重载中，`newName`拷贝到`Widget::names`中，右值重载中，移动进去。开销总结：左值一次拷贝，右值一次移动。
- **Using a universal reference（通用模板方式）**：同重载一样，调用也绑定到`addName`的引用实现上，没有开销。由于使用了`std::forward`，左值参数会复制到`Widget::names`，右值参数移动进去。开销总结同重载方式。
  Item25 解释了如果调用者传递的参数不是`std::string`类型，将会转发到`std::string`的构造函数（几乎是零开销的拷贝或者移动操作）。因此通用引用的方式同样有同样效率，所以者不影响本次分析，简单分析`std::string`参数类型即可。
- **Passing by value（按值传递）**：无论传递左值还是右值，都必须构造`newName`参数。如果传递的是左值，需要拷贝的开销，如果传递的是右值，需要移动的开销。在函数的实现中，`newName`总是采用移动的方式到`Widget::names`。开销总结：左值参数，一次拷贝一次移动，右值参数两次移动。对比按引动传递的方法，对于左值或者右值，均多出一次移动操作。

再次回顾本Item的内容：

> 总是考虑直接按值传递，如果参数可拷贝并且移动操作开销很低

这样措辞是有原因的：

1. 应该仅*consider using pass by value*。是的，因为只需要编写一个函数，同时只会在目标代码中生成一个函数。避免了通用引用方式的种种问题。但是毕竟开销会更高，而且下面还会讨论，还会存在一些目前我们并未讨论到的开销。

2. 仅考虑对于*copable parameters*按值传递。不符合此条件的的参数必须只有移动构造函数。回忆一下"重载"方案的问题，就是必须编写两个函数来分别处理左值和右值，如果参数没有拷贝构造函数，那么只需要编写右值参数的函数，重载方案就搞定了。
   考虑一下`std::unique_ptr<std::string>`的数据成员和其`set`函数。因为`std::unique_ptr`是仅可移动的类型，所以考虑使用"重载"方式编写即可：

   ```
   class Widget {
   public:
     ...
     void setPtr(std::unique_ptr<std::string>&& ptr) {
       p = std::move(ptr);
     }
   private:
     std::unique_ptr<std::string> p;
   };
   ```

   调用者可能会这样写：

   ```
   Widget w;
   ...
   w.setPtr(std::make_unique<std::string>("Modern C++"));
   ```

   这样，传递给`setPtr`的参数就是右值，整体开销就是一次移动。如果使用传值方式编写：

   ```
   class Widget {
   public:
     ...
     void setPtr(std::unique_ptr<std::string> ptr) {
       p = std::move(ptr);
     }
   private:
     std::unique_ptr<std::string> p;
   };
   ```

   同样的调用就会先使用移动构造函数移动到参数`ptr`,然后再移动到`p`，整体开销就是两次移动。

3. 按值传递应该仅应用于哪些*cheap to move*的参数。当移动的开销较低，额外的一次移动才能被开发者接受，但是当移动的开销很大，执行不必要的移动类似不必要的复制时，这个规则就不适用了。

4. 你应该只对*always copied（肯定复制）*的参数考虑按值传递。为了看清楚为什么这很重要，假定在复制参数到`names`容器前，`addName`需要检查参数的长度是否过长或者过短，如果是，就忽略增加`name`的操作：

```
class Widget { // Approach 3
public:
  void addName(std::string newName) {
    if ((newName.length() >= minLen) && (newName.length() <= maxLen)) {
          names.push_back(std::move(newName));
    }
  }
  ...
private:
  std::vector<std::string> names;
};
```

即使这个函数没有在`names`添加任何内容，也增加了构造和销毁`newName`的开销，而按引用传递会避免这笔开销。

即使你编写的函数是移动开销小的参数而且无条件复制，有时也可能不适合按值传递。这是因为函数复制参数存在两种方式：一种是通过构造函数（拷贝构造或者移动构造），还有一种是赋值（拷贝赋值或者移动赋值）。`addName`使用构造函数，它的参数传递给`vector::push_back`，在这个函数内部，`newName`是通过构造函数在`std::vector`创建一个新元素。对于使用构造函数拷贝参数的函数，上述分析已经可以给出最终结论：按值传递对于左值和右值均增加了一次移动操作的开销。

当参数通过赋值操作进行拷贝时，分析起来更加复杂。比如，我们有一个表征密码的类，因为密码可能会被修改，我们提供了`setter`函数`changeTo`。用按值传递的策略，我们实现一个密码类如下：

```
class Password {
public:
  explicit Password(std::string pwd) : text(std::move(pwd)) {}
  void changeTo(std::string newPwd) {
    text = std::move(newPwd);
  }
  ...
private:
  std::string text;
};
```

将密码存储为纯文本格式恐怕将使你的软件安全团队抓狂，但是先忽略这点考虑这段代码：

```
std::string initPwd("Supercalifragilisticexpialidocious");
Password p(initPwd);
```

`p.text`被给定的密码构造，用按值传递的方式增加了一次移动操作的开销相对于重载或者通用引用，但是这无关紧要，一切看起来如此美好。

但是，该程序的用户可能对初始密码不太满意，因为这段密码`"Supercalifragilisticexpialidocious"`在许多字典中可以被发现。他或者她因此修改密码：

```
std::string newPassword = "Beware the Jabberwock";
p.changeTo(newPassword);
```

不用关心新密码是不是比就密码更好，那是用户关心的问题。我们对于`changeTo`函数的按值传递实现方案会导致开销大大增加。

传递给`changeTo`的参数是一个左值（`newPassword`），所以`newPwd`参数需要被构造，`std::string`的拷贝构造函数会被调用，这个函数会分配新的存储空间给新密码。`newPwd`会移动赋值到`text`，这会导致释放旧密码的内存。所以`changeTo`存在两次动态内存管理的操作：一次是为新密码创建内存，一次是销毁旧密码的内存。

但是在这个例子中，旧密码比新密码长度更长，所以本来不需要分配新内存，销毁就内存的操作。如果使用重载的方式，两次动态内存管理操作可以避免：

```
class Password {
public:
  ...
  void changeTo(std::string& newPwd) {
    text = newPwd;
  }
  ...
private:
  std::string text;
};
```

这种情况下，按值传递的开销（包括了内存分配和内存销毁）可能会比`std::string`的`move`操作高出几个数量级。

有趣的是，如果旧密码短于新密码，在赋值过程中就不可避免要重新分配内存，这种情况，按值传递跟按引用传递的效率是一样的。因此，参数的赋值操作开销取决于具体的参数的值，这种分析适用于动态分配内存的参数类型。

这种潜在的开销增加仅在传递左值参数时才适用，因为执行内存分配和释放通常发生在复制操作中。

结论是，使用按值传递的函数通过赋值复制一个参数的额外开销取决于传递的类型中左值和右值的比例，即这个值是否需要动态分配内存，以及赋值操作符的具体实现中对于内存的使用。对于`std::string`来说，取决于实现是否使用了小字符串优化(SSO 参考Item 29)，如果是，值是否匹配SSO缓冲区。

所以，正如我所说，当参数通过赋值进行拷贝时，分析按值传递的开销是复杂的。通常，最有效的经验就是“在证明没问题之前假设有问题”，就是除非已证明按值传递会为你需要的参数产生可接受开销的执行效率，否则使用重载或者通用引用的实现方式。

到此为止，对于需要运行尽可能快的软件来说，按值传递可能不是一个好策略，因为毕竟多了一次移动操作。此外，有时并不能知道是不是还多了其他开销。在`Widget::addName`例子中，按值传递仅多了一次移动操作，但是如果加入值的一些校验，可能按值传递就多了创建和销毁类型的开销相对于重载和通用引用的实现方式。

可以看到导致的方向，在调用链中，每次调用多了一次移动的开销，那么当调用链较长，总体就会产生无法忍受的开销，通过引用传递，调用链不会增加任何开销。

跟性能无关，总是需要考虑的是，按值传递不像按引用传递那样，会收到切片问题的影响。这是C++98的问题，在此不在详述，但是如果要设计一个函数，来处理这样的参数：基类或者其派生类，如果不想声明为按值传递，因为你就是要分割派生类型

```
class Widget{...};
class SpecialWidget: public Widget{...};
void processWidget(Widget w);
...
SecialWidget sw;
...
processWidget(sw);
```

如果不熟悉**slicing problem**，可以先通过搜索引擎了解一下。这样你就知道切片问题是另一个C++98中默认按值传递名声不好的原因。有充分的理由来说明为什么你学习C++编程的第一件事就是避免用户自定义类型进行按值传递。

C++11没有从根本上改变C++98按值传递的基本盘，通常，按值传递仍然会带来你希望避免的性能下降，而且按值传递会导致切片问题。C++11中新的功能是区分了左值和右值，实现了可移动类型的移动语义，尽管重载和通用引用都有其缺陷。对于特殊的场景，复制参数，总是会被拷贝，而且移动开销小的函数，可以按值传递，这种场景通常也不会有切片问题，这时，按值传递就提供了一种简单的实现方式，同时实现了接近引用传递的开销的效率。

需要记住的事

- 对于可复制，移动开销低，而且无条件复制的参数，按值传递效率基本与按引用传递效率一致，而且易于实现，生成更少的目标代码
- 通过构造函数拷贝参数可能比通过赋值拷贝开销大的多
- 按值传递会引起切片问题，所说不适合基类类型的参数

Item42: 考虑使用emplacement代替insertion

如果你拥有一个容器，例如`std::string`，那么当你通过插入函数（例如`insert, push_front, push_back`，或者对于`std::forward_list`， `insert_after`）添加新元素时，你传入的元素类型应该是`std::string`。毕竟，这就是容器里的内容。

逻辑上看来如此，但是并非总是如此。考虑如下代码：

```
std::vector<std::string> vs; // container of std::string
vs.push_back("xyzzy"); // add string literal
```

这里，容量里内容是`std::string`，但是你试图通过`push_back`加入字符串字面量，即引号内的字符序列。字符转字面量并不是`std::string`，这意味着你传递给`push_back`的参数并不是容器里的内容类型。

`std::vector`的`push_back`被按左值和右值分别重载：

```
template<class T, class Allocator = allocator<T>>
class vector {
public:
  ...
  void push_back(const &T x); // insert lvalue
  void push_back(T&& x); // insert rvalue
  ...
};
```

在`vs.push_back("xyzzy")`这个调用中，编译器看到参数类型（const char[6]）和`push_back`采用的参数类型（`std::string`的引用）之间不匹配。它们通过从字符串字面量创建一个`std::string`类型的临时变量来消除不匹配，然后传递临时变量给`push_back`。换句话说，编译器处理的这个调用应该像这样：

```
vs.push_back(std::string("xyzzy")); // create temp std::string and pass it to
push_back
```

代码编译并运行，皆大欢喜。除了对于性能执着的人意识到了这份代码不如预期的执行效率高。

为了创建`std::string`类型的临时变量，调用了`std::string`的构造器，但是这份代码并不仅调用了一次构造器，调用了两次，而且还调用了析构器。这发生在`push_back`运行时：

1. 一个`std::string`的临时对象从字面量"xyzzy"被创建。这个对象没有名字，我们可以称为*temp*，*temp*通过`std::string`构造器生成，因为是临时变量，所以*temp*是右值。
2. *temp*被传递给`push_back`的右值x重载函数。在`std::vector`的内存中一个x的副本被创建。这次构造器是第二次调用，在`std::vector`内部重新创建一个对象。（将x副本复制到`std::vector`内部的构造器是移动构造器，因为x传入的是右值，有关将右值引用强制转换为右值的信息，请参见Item25）。
3. 在`push_back`返回之后，*temp*被销毁，调用了一次`std::string`的析构器。

性能执着者（译者注：直译性能怪人）不禁注意到是否存在一种方法可以获取字符串字面量并将其直接传入到步骤2中的`std::string`内部构造，可以避免临时对象*temp*的创建与销毁。这样的效率最好，性能执着者也不会有什么意见了。

因为你是一个C++开发者，所以你会有高于平均水平的要求。如果你不是C++开发者，你可能也会同意这个观点（如果你根本不考虑性能，为什么你没在用python？）。所以让我来告诉你如何使得`push_back`达到最高的效率。就是不使用`push_back`，你需要的是`emplace_back`。

`emplace_back`就是像我们想要的那样做的：直接把传递的参数（无论是不是`std::string`）直接传递到`std::vector`内部的构造器。没有临时变量会生成：

```
vs.emplace_back("xyzzy"); // construct std::string inside vs directly from
"xyzzy"
```

`emplace_back`使用完美转发，因此只要你没有遇到完美转发的限制（参见Item30），就可以传递任何参数以及组合到`emplace_back`。比如，如果你在vs传递一个字符和一个数量给`std::string`构造器创建`std::string`，代码如下：

```
vs.emplace_back(50, 'x'); // insert std::string consisting of 50 'x' characters
```

`emplace_back`可以用于每个支持`push_back`的容器。类似的，每个支持`push_front`的标准容器支持`emplace_front`。每个支持`insert`（除了`std::forward_list`和`std::array`）的标准容器支持`emplace`。关联容器提供`emplace_hint`来补充带有“hint”迭代器的插入函数，`std::forward_list`有`emplace_after`来匹配`insert_after`。

使得emplacement函数功能优于insertion函数的原因是它们灵活的接口。insertion函数接受对象来插入，而emplacement函数接受构造器接受的参数插入。这种差异允许emplacement函数避免临时对象的创建和销毁。

因为可以传递容器内类型给emplacement函数（该参数使函数执行复制或者移动构造器），所以即使insertion函数不会构造临时对象，也可以使用emplacement函数。在这种情况下，insertion和emplacement函数做的是同一件事，比如：

```
std::string queenOfDisco("Donna Summer");
```

下面的调用都是可行的，效率也一样：

```
vs.push_back(queenOfDisco); // copy-construct queenOfDisco
vs.emplace_back(queenOfDisco); // ditto
```

因此，emplacement函数可以完成insertion函数的所有功能。并且有时效率更高，至上在理论上，不会更低效。那为什么不在所有场合使用它们?

因为，就像说的那样，理论上，在理论和实际上没有什么区别，但是实际，区别还是有的。在当前标准库的实现下，有些场景，就像预期的那样，emplacement执行性能优于insertion，但是，有些场景反而insertion更快。这种场景不容易描述，因为依赖于传递的参数类型、容器类型、emplacement或insertion的容器位置、容器类型构造器的异常安全性和对于禁止重复值的容器（即`std::set,std::map,std::unorder_set,set::unorder_map`）要添加的值是否已经在容器中。因此，大致的调用建议是：通过benchmakr测试来确定emplacment和insertion哪种更快。

当然这个结论不是很令人满意，所以还有一种启发式的方法来帮助你确定是否应该使用emplacement。如果下列条件都能满足，emplacement会优于insertion：

- **值是通过构造器添加到容器，而不是直接赋值**。例子就像本Item刚开始的那样（添加"xyzzy"到`std::string`的`std::vector`中）。新值必须通过`std::string`的构造器添加到`std::vector`。如果我们回看这个例子，新值放到已经存在对象的位置，那情况就完全不一样了。考虑下：

```
std::vector<std::string> vs; // as before
... // add elements to vs
vs.emplace(vs.begin(), "xyzzy"); // add "xyzzy" to beginning of vs
```

对于这份代码，没有实现会在已经存在对象的位置`vs[0]`构造添加的`std::string`。而是，通过移动赋值的方式添加到需要的位置。但是移动赋值需要一个源对象，所以这意味着一个临时对象要被创建，而emplacement优于insertion的原因就是没有临时对象的创建和销毁，所以当通过赋值操作添加元素时，emplacement的优势消失殆尽。

而且，向容器添加元素是通过构造还是赋值通常取决于实现者。但是，启发式仍然是有帮助的。基于节点的容器实际上总是使用构造器添加新元素，大多数标准库容器都是基于节点的。例外的容器只有`std::vector, std::deque, std::string`（`std::array`也不是基于节点的，但是它不支持emplacement和insertion）。在不是基于节点的容器中，你可以依靠`emplace_back`来使用构造向容器添加元素，对于`std::deque`，`emplace_front`也是一样的。

- **传递的参数类型与容器的初始化类型不同**。再次强调，emplacement优于insertion通常基于以下事实：当传递的参数不是容器保存的类型时，接口不需要创建和销毁临时对象。当将类型为T的对象添加到container时，没有理由期望emplacement比insertion运行的更快，因为不需要创建临时对象来满足insertion接口。
- **容器不拒绝重复项作为新值**。这意味着容器要么允许添加重复值，要么你添加的元素都是不重复的。这样要求的原因是为了判断一个元素是否已经存在于容器中，emplacement实现通常会创建一个具有新值的节点，以便可以将该节点的值与现有容器中节点的值进行比较。如果要添加的值不在容器中，则链接该节点。然后，如果值已经存在，emplacement创建的节点就会被销毁，意味着构造和析构时浪费的开销。这样的创建就不会在insertion函数中出现。

本Item开始的例子中下面的调用满足上面的条件。所以调用比`push_back`运行更快。

```
vs.emplace_back("xyzzy"); // construct new value at end of container; don't pass
the type in container; don't use container rejecting duplicates
vs.emplace_back(50, 'x'); // ditto
```

在决定是否使用emplacement函数时，需要注意另外两个问题。**首先**是资源管理。假定你有一个`std::shared_ptr<Widget>s`的容器，

```
std::list<std::shared_ptr<Widget>> ptrs;
```

然后你想添加一个通过自定义deleted释放的`std::shared_ptr`（参见Item 19）。Item 21说明你应该使用`std::make_shared`来创建`std::shared_ptr`，但是它也承认有时你无法做到这一点。比如当你要指定一个自定义deleter时。这时，你必须直接创建一个原始指针，然后通过`std::shared_ptr`来管理。

如果自定义deleter是这个函数，

```
void killWidget(Widget* pWidget);
```

使用insertion函数的代码如下:

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

也可以像这样

```
ptrs.push_back({new Widget, killWidget});
```

不管哪种写法，在调用`push_back`中会生成一个临时`std::shared_ptr`对象。`push_back`的参数是`std::shared_ptr`的引用，因此必须有一个`std::shared_ptr`。

`std::shared_ptr`的临时对象创建应该可以避免，但是在这个场景下，临时对象值得被创建。考虑如下可能的时间序列：

1. 在上述的调用中，一个`std::shared_ptr<Widget>`的临时对象被创建来持有`new Widget`对象。称这个对象为*temp*。
2. `push_back`接受*temp*的引用。在节点的分配一个副本来复制*temp*的过程中，OOM异常被抛出
3. 随着异常从`push_back`的传播，*temp*被销毁。作为唯一管理Widget的弱指针`std::shared_ptr`对象，会自动销毁`Widget`，在这里就是调用`killWidget`。

这样的话，即使发生了异常，没有资源泄露：在调用`push_back`中通过`new Widget`创建的`Widget`在`std::shared_ptr`管理下自动销毁。生命周期良好。

考虑使用`emplace_back`代替`push_back`

```
ptrs.emplace_back(new Widget, killWidget);
```

1. 通过`new Widget`的原始指针完美转发给`emplace_back`的内部构造器。如果分配失败，还是抛出OOM异常
2. 当异常从`emplace_back`传播，原始指针是仅有的访问途径，但是因为异常丢失了，这就发生了资源泄露

在这个场景中，生命周期不良好，这个失误不能赖`std::shared_ptr`。`std::unique_ptr`使用自定义deleter也会有同样的问题。根本上讲，像`std::shared_ptr`和`std::unique_ptr`这样的资源管理类的有效性取决于资源被**立即**传递给资源管理对象的构造函数。实际上，这就是`std::make_shared`和`std::make_unique`这样的函数如此重要的原因。

在对存储资源管理类的容器调用insertion函数时（比如`std::list<std::shared_ptr<Widget>>`），函数的参数类型通常确保在资源的获取和管理资源对象的创建之间没有其他操作。在emplacement函数中，完美转发推迟了资源管理对象的创建，直到可以在容器的内存中构造它们为止，这给异常导致资源泄露提供了可能。所有的标准库容器都容易受到这个问题的影响。在使用资源管理对象的容器时，比如注意确保使用emplacement函数不会为提高效率带来降低异常安全性的后果。

坦白说，无论如何，你不应该将`new Widget`传递给`emplace_back`或者`push_back`或者大多数这种函数，因为，就像Item 21中解释的那样，这可能导致我们刚刚讨论的异常安全性问题。使用独立语句将从`new Widget`获取指针然后传递给资源管理类，然后传递这个对象的右值引用给你想传递`new Widget`的函数（Item 21 有这个观点的详细讨论）。代码应该如下：

```
std::shared_ptr<Widget> spw(new Widget, killWidget); // create Widget and have
spw manage it
ptrs.push_back(std::move(spw)); // add spw as rvalue
```

emplace_back的版本如下：

```
std::shared_ptr<Widget> spw(new Widget, killWidget); // create Widget and have
spw manage it
ptrs.emplace_back(std::move(spw));
```

无论哪种方式，都会产生spw的创建和销毁成本。给出选择emplacement函数优于insertion函数的动机是避免临时对象的开销，但是对于swp的概念来讲，当根据正确的方式确保获取资源和连接到资源管理对象上之间无其他操作，添加资源管理类型对象到容器中，emplacement函数不太可能胜过insertion函数。

emplacement函数的**第二个**值得注意的方面是它们与显式构造函数的交互。对于C++11正则表达式的支持，假设你创建了一个正则表达式的容器：

```
std::vector<std::regex> regexes;
```

由于你同事的打扰，你写出了如下看似毫无意义的代码：

```
regexes.emplace_back(nullptr); // add nullptr to container of regexes?
```

你没有注意到错误，编译器也没有提示你，所以你浪费了大量时间来调试。突然，你发现你插入了空指针到正则表达式的容器中。但是这怎么可能？指针不是正则表达式，如果你试图下面这样写

```
std::regex r = nullptr; // error! won't compile
```

编译器就会报错。有趣的是，如果你调用`push_back`而不是`emplace_back`，编译器就会报错

```
regexes.push_back(nullptr); // error! won't compile
```

当前你遇到的奇怪行为由于可能用字符串构造`std::regex`的对象，这就意味着下面代码合法：

```
std::regex upperCaseWorld("[A-Z]+");
```

通过字符串创建`std::regex`要求相对较长的运行时开销，所以为了最小程度减少无意中产生此类开销的可能性，采用`const char*`指针的`std::regex`构造函数是显式的。这就是为什么下面代码无法编译的原因：

```
std::regex r = nullptr; // error! won't compile
regexes.push_back(nullptr); // error
```

在上面的代码中，我们要求从指针到`std::regex`的隐式转换，但是显式构造的要求拒绝了此类转换。

但是在`emplace_back`的调用中，我们没有声明传递一个`std::regex`对象。代替的是，我们传递了一个`std::regex`构造器参数。那不是隐式转换，而是显式的：

```
std::regex r(nullptr); // compiles
```

如果简洁的注释“compiles”表明缺乏直观理解，好的，因为这个代码可以编译，但是行为不确定。使用`const char*`指针的`std::regex`构造器要求字符串是一个有效的正则表达式，nullptr不是有效的。如果你写出并编译了这样的代码，最好的希望就是运行时crash掉。如果你不幸运，就会花费大量的时间调试。

先把`push_back, emplace_back`放在一边，注意到相似的初始化语句导致了多么不一样的结果：

```
std::regex r1 = nullptr; // error ! won't compile
std::regex r2(nullptr); // compiles
```

在标准的官方术语中，用于初始化r1的语法是所谓的复制初始化。相反，用于初始化r2的语法是（也被称为braces）被称为直接初始化。复制初始化不是显式调用构造器的，直接初始化是。这就是r2可以编译的原因。

然后回到`push_back`和 `emplace_back`，更一般来说，insertion函数对比emplacment函数。emplacement函数使用直接初始化，这意味着使用显式构造器。insertion函数使用复制初始化。因此：

```
regexes.emplace_back(nullptr); // compiles. Direct init permits use of explicit
std::regex ctor taking a pointer
regexes.push_back(nullptr); // error! copy init forbids use of that ctor
```

要汲取的是，当你使用emplacement函数时，请特别小心确保传递了正确的参数，因为即使是显式构造函数，编译器可以尝试解释你的代码称为有效的（译者注：这里意思是即使你写的代码逻辑上不对，显式构造器时编译器可能能解释通过即编译成功）

需要记住的事

- 原则上，emplacement函数有时会比insertion函数高效，并且不会更差
- 实际上，当执行如下操作时，emplacement函数更快
    1. 值被构造到容器中，而不是直接赋值
    2. 传入的类型与容器类型不一致
    3. 容器不拒绝已经存在的重复值
- emplacement函数可能执行insertion函数拒绝的显示构造