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