设计与声明

条款18: 让接口容易被正确使用，不易被误用

- 好的接口很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质
- “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容
- “阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任
- shared_ptr 支持定制型删除器。这可以防范DLL问题，可被用来自动解除互斥锁等等。

设计接口的理想原则是，如果客户企图使用某个接口而没有获得他所预期的行为，这个代码就不应该通过编译；如果代码通过了编译， 它的作用就该是客户想要的。这就要求首先必须考虑客户可能做出的错误。

```
class Date {
public:
    Date(int month, int day, int year);
};
```

上面这段代码存在两个问题：

- 客户可能会按照错误的顺序传递参数
- 客户可能传递一个无效的月份或者天数

可以通过导入新类型而获得预防。

```
struct Day {
    explicit Day(int d) : val(d) {}
    int val;
};

struct Month {
    explicit Month(int m) : val(m) {}
    int val;
};

struct Year {
    explicit Year(int y) : val(y) {}
    int val;
};

class Date {
public:
    Date(const Month& month, const Day& day, const Year& year);
};

Date(Month(2), Day(15), Year(2022));
```

这样做就起到了警示作用，同时又限定了传递顺序。

限定了传递顺序后，我们可以进一步对使用的值进行限定。简单的方法是使用 enum 来限定，但是 enum 并不具备类型安全性，例如 enums 可以被用来当作一个 ints 使用。因此可以重新设计一个类。

```
class Month {
public:
    static Month Jan() { return Month(1); }
    static Month Feb() { return Month(2); }
    ...
    static Month Dec() { return Month(12); }
private:
    int val;
    explicit Month(int m) : val(m) {}
};
```

将 Month construct 设置为 private 防止产生新的月份。如果需要选择特定的月份，使用 static function 来完成。

预防客户错误的另一个方法是限制什么事可做，什么不能做。常见的方法是加入 const 限定。例如使用 const 修饰 operator* 的返回类型，可以阻止客户因 用户定义类型而犯错。

另一个一般性准则是："让 types 容易被正确使用，不容易被误用"。其表示形式是：除非有好理由，否则应该尽量令你的 types 行为与内置 types 一致。这条准则的内在理由是为了能够提供行为一致的接口。

任何要求客户必须记得做某件事情，就是有着 "不正确使用" 的倾向。例如条款 13 中所提供的 createInvestment 函数。如果 期望客户使用智能指针来接受，实际上是放纵客户产生资源泄露。

```
std::shared_ptr<Investment> createInvestment();
```

上面这种写法杜绝了客户使用过程中忘记 delete 的错误，因为客户必须将其存储在 std::shared_ptr 中。除此之外，也可以提 供特殊的 deleter 来防止客户错误的调用自行定义的 deleter。

```
std::shared_ptr<Investment> createInvestment() {
    std::shared_ptr<Investment> retVal(
        static_cast<Investment*>(0),
        getRidofInvestment()
    );
    retVal = ...; // 令 retVal 指向正确的对象
    return retVal;
}
```

上面这种写法，可以直接返回一个 "将getRidofInvestment" 绑定为删除器的 std::shared_ptr。这样做的另一个好处是它会 自动调用 deleter，因此可以消除潜在的客户错误: cross-DLL problem。这个错误发生于"在 DLL 中被

new 创建的对象，却 在另一个 DLL 内被 delete 释放”。这类行为，在众多平台上会导致运行期错误。而 std::shared_ptr 不会发生这个问题。

条款19: 设计 class 犹如设计 type

- class 的设计就是 type 的设计。在定义一个新的 type 之前，请确定你已经考虑过本条覆盖的所有讨论主题。

程序编码的大部分时间都在扩张类型系统。因此需要了解如何设计一个高效的 class (Type)。首先需要了解这需要面对哪些问题：

- 新 type 的对象应该如何被创建和销毁？
- 对象初始化和对象赋值该有什么样的差别？别混淆初始化和赋值，因为他们对应于不同的函数调用。
- 新 type 对象如果被 pass by value 意味着什么？
- 什么是新 type 的合法值？对 class 的成员变量而言，通常只有某些数值集是有效的。
- 你的新 type 需要配合某个继承图系吗？
- 你的新 type 需要什么样的转换？
- 什么样的操作符和函数对此新 type 而言是合理的？
- 什么样的标准函数应该驳回？
- 谁该取用新 type 的成员？
- 什么是新 type 的“未声明接口”？
- 你的新 type 有多么一般化？
- 你真的需要一个新的 type 吗？

条款20: 宁以 pass-by-reference-to-const 替换 pass-by-value

- 尽量以 pass-by-reference-to-const 替换 pass-by-value。前者通常比较高效，并可避免切割问题
- 以上规则并不适用于内置类型，以及 STL 的迭代器和函数对象。对他们而言，往往 pass-by-value 更合适

```
class Person {
public:
    Person();
    virtual ~Person();

private:
    std::string name;
    std::string address;
};

class Student : public Person {
public:
    Student();
    ~Student();

private:
```

```
    std::string schoolName;
    std::string schoolAddress;
};

bool validateStudent(Student s);
Student a;
bool aIsOk = validateStudent(a);
```

上述代码使用 by-value 的方式传值。在这个过程中，首先会调用 Student 的 copy constructor 来对 a 进行赋值，并且在 validateStudent 返回时调用 destructor。其中包含两个 string 的成员变量，因此也会发生对应的复制和销毁操作。同时， Student 继承自 Person，那么意味着 Person 也会发生复制和销毁，其中包含的两个成员变量也会发生复制和销毁。

可以看到这将经历 6 次构造和析构。使用 pass by reference to const 的方式传递则不会有任何进行调用。

```
bool validateStudent(const Student& s);
```

by-reference 方式也可以避免 slicing 的发生。当一个 derived class 对象被以 by-value 的方式传入到 base class 中时会发生 slicing，即原有的 derived class 对象的特化性质全被切割掉了。因为这个过程调用的时 base class copy constructor。而不会将其余内容进行复制。

```
class Window {
public:
    std::string name() const;
    virtual void display() const;
};

class WindowWithScrollBars : public Window {
public:
    virtual void display() const;
};

void printNameAndDisplay(Window w) {
    std::cout << w.name();
    w.display();
}

WindowWithScrollBars w;
printNameAndDisplay(w);
```

上述代码只会调用 Window::display 而不会调用子类的 display 方法，因为发生了切割。解决 slicing 的方法就是用 by reference to const 的方式进行传递。

```
void printNameAndDisplay(const Window& w) {
    std::cout << w.name();
```

```
    w.display();
}
```

事实上，by-reference 的底层实现是 pointer 方式，因此如果有个内置类型的对象，by-value 的方式往往比 by-reference 的方式更加高效。对于 STL 中的迭代器和函数对象，也十分适用。

注意：小型的自定义类型并不意味着和内置类型对象的成本划等号。也就意味着，即使小也不一定适用于 by-value 方式。主要原因 包括两点：

- 编译器可能不这么认为，它可能将 double 放进缓存器中，却不愿意将你用 double 实现的对象放进缓存器。
- type 会变化，随着 type 的不断维护更新可能会变得越来越大。

所以 “pass-by-value 并不昂贵” 的唯一对象就是 **内置类型** 和 **STL的迭代器和函数对象**。

条款21: 必须返回对象时，别妄想返回其 reference

- 绝对不要返回 pointer 或 reference 指向一个 local stack 对象，或返回一个 reference 指向一个 heap-allocated 对象

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1);

private:
    int n, d;
    friend const Rational operator* (const Rational& lhs, const Rational& rhs);
};
```

上述代码中，以 by value 的方式返回计算结果。而上述内容无法修改为返回 reference 因为会出现 意向不到的问题。

```
const Rational& operator* (const Rational& lhs, const Rational& rhs) {
    Rational result (lhs.n * rhs.n, lhs.d * rhs.d);
    return result;
}
```

返回 stack 对象显然是不合理的，因为当脱离当前的 scoop 后，result 被释放，于是返回的 reference 指向了被销毁的地方。

```
const Rational& operator* (const Rational& lhs, const Rational& rhs) {
    Rational* result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
    return *result;
}
```

那么返回一个 heap object 呢？这显然也是不合理的，因为你完全不知道何时去释放掉 new 出来的对 像，甚至有可能你都无法找到那个生成的对象，例如：

```
Rational w, x, y, z;
w = x * y * z; // 连续两次调用，而你无法获取第一次调用产生的指针
```

那么结果就是，当必须要返回新对象时，直接返回新对象就行了。如果成本过高，编译器会想办法进行优化。

条款22: 将成员变量声明为 private

- 切记将成员变量声明为 private。这可赋予客户访问数据的一致性、可细微划分访问权限、允许约束条件获得保障，为提供 class 作者以充分的实现弹性
- protected 并不比 public 更具封装性

首先需要了解为什么成员变量不应该是 public 和 protected 的，然后显然易见应当使用 private。

首先从语法的一致性开始。如果成员变量不是 public，客户就需要使用成员函数来访问成员变量。如此 以来就不再需要区分 成员变量 和 成员函数了，全都按照后者方式访问就行了。

其次，使用函数方式可以更加精确的控制成员变量。而如果成员变量是 public 的，这就意味着任何人都 有权限随时更改对象的成员变量。

```
class AccessLevels {
public:
    int getReadOnly() const { return readOnly; }
    void setReadWrite(int val) { readWrite = val; }
    int getReadWrite() const { return readWrite; }
    void setWriteOnly(int val) { writeOnly = val; }

private:
    int noAccess;
    int readOnly;
    int readWrite;
    int writeOnly;
};
```

通过精心设计，可以细微的划分访问控制权限。

最后就是封装，如果日后你需要改变某个值的计算方式，使用函数方式进行访问变量，用户完全不会知道 发生了什么改变。这种灵活性有时显得十分重要。

```
class SpeedDataCollection {
public:
    void addValue(int speed);
```

```
    double average() const;
};
```

上述代码可以有两个优化方向。一，在对象中维护一个平均速度，当每次调用 average 时直接返回对象。 二，每次被调用时重新计算平均值。

前者需要额外的存储，但是速度十分迅速。而后者计算缓慢却不需要额外的存储空间。因此在内存吃紧，但 很少需要平均值的机器上可以使用前者，内存宽裕但是频繁调用 average 的机器上则可以使用后者。

proteced 限定与 public 在封装上的影响基本类似，后者可以说完全没有封装性，但 protected 的 封装性也十分有限，因为这将影响所有 derived class。

从封装的角度看，其实只有两种访问权限: private(提供封装) 和 其他(不提供封装)。

条款23: 宁以 non-member、non-friend 替换 member 函数

- 宁可拿 non-member non-friend 函数替换 member 函数。这样做可以增加封装性、包裹弹性和机能扩充性。

```
class WebBrowser {
public:
    void clearCach();
    void clearHistory();
    void removeCoockies();

    void clearEverything();
};

void clearBrowser(WebBrowser& wb) {
    wb.clearCach();
    wb.clearHistory();
    wb.removeCoockies();
}
```

clearEverything 和 clearBrowser 相比之下，后者更好。这点与直观印象相反，面向对象的原则是 数据应该尽可能的被封装，然而与直观相反的是，member 函数带来的封装性要比 non-member 函数的 封装性更低。因为 non-member 函数能提供较大的"包裹弹性"。

对于封装而言，越多东西被封装，代码可改动的内容也就越大，而改动影响到的客户越少。而一个数据越多 函数可以访问，那它的封装性也就越差。

正如条款 22所言，我们要将成员变量声明为 private，而想要访问该变量就需要借助 成员函数 或者 友元函数。结合前一段所说，选择 non-member non-friend 函数，其封装性也就越好。

有两点内容需要注意：

1. 这个条款用于区分 member 和 non-member non-friend 函数，而非 non-member 函数。
2. 这里的 non-member non-firend 也可以是其他 class 的 member 函数，只要不是 friend

当随着类型的扩充，可能会提供大量的便利函数，而用户通常只需要其中的某一类。比如 WebBrowser 可 能提供了与书签相关的、与打印相关的、与cookie管理相关的内容。这时就可以使用不同的头文件进行管 理。

```
// webbrowser.h 关于 WebBrowser 的定义，以及核心机能
namespace WebBrowserStuff { // 放在一个命名空间中
class WebBrowser {};
// 核心机能函数
}

// webbrowserbookmarks.h
namespace WebBrowserStuff {
// 书签相关函数
}

// webbrowsercookies.h
namespace WebBrowserStuff {
// cookie相关函数
}
```

条款24: 若所有参数皆需要类型转换，请为此采用 non-member 函数

- 如果你需要为某个函数的所有参数进行类型转换，那么这个函数必须是个 non-member。

令 classes 支持饮食类型转换通常是一个糟糕的主意。但是也存在例外，例如设计一个 class 来表示 有理数，允许整数类型转换为有理数似乎颇为合理。

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1);
    int numerator() const;
    int denominator() const;

private:
    int _numerator;
    int _denominator;
};
```

对于上面这个类，如果希望写一个 乘法 函数应该怎样操作呢?

```
class Rational {
public:
    const Rational operator*(const Rational& rhs) const;
};
```

最简单的方式肯定是写在 class 内，这显然是能正常运行的。但是如果你想实现混合运算又怎么办，例如：

```
Rational oneHalf(1, 2);
Rational result = oneHalf * 2; // 成功
Rational result = 2 * oneHalf; // 错误
```

此时就会出现问题，因为 2.operator*() 并不存在，因为 2 是常量。紧接着编译器会尝试寻找一个可 以被调用的 non-member operator*。此时仍然有问题，因为不存在一个接收 int 和 Rational 的 乘法。

而 oneHalf * 2 为什么能成功？首先其调用的是 oneHalf.operator*(Rational)，其次由于 Rational 的 constructor 没有使用 explict 修饰，因此是可以被隐式转换得来的。

想要完全支持混合运算，可以让 operator* 成为一个 non-member 函数，允许编译器在每一个实参身 上执行隐式类型转换。

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1) : _numerator(numerator),
      _denominator(denominator) {}
    int numerator() const { return _numerator; }
    int denominator() const { return _denominator; }
private:
    int _numerator;
    int _denominator;
};

const Rational operator*(const Rational& lhs, const Rational& rhs) {
  return Rational(lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator());
}
```

这种形式可以很好的实现混合算数。需要额外注意，既然可以通过 Rational 的 public 成员函数来完 成这种运算的实现，就不要将 operator* 标记为 friend 函数。

此外，这条的实现方式是在从 Object-Oriented C++ 的角度而选择的做法，当你跨进 Template C++ 时则会有新的争议、解法需要考虑。这点会在条款46中看到。

条款25: 考虑写出一个不抛出异常的 swap 函数

- 当 std::swap 对你的类型效率不高时，提供一个 swap 成员函数，并确定这个函数不抛出异常。
- 如果你提供一个 member swap，也该提供一个 non-member swap 用来调用前者。对于 classes (而非 templates) 也请特化 std::swap
- 调用 swap 时应针对 std::swap 使用 using 声明式，然后调用 swap 并且不带任何 “命名空间资格修饰”
- 为 “用户定义类型” 进行 std templates 全特化是好的，但千万不要尝试在 std 内加入某些对 std 而言全新的东西

---

swap是个有趣的函数，它是STL的一部分，后来称为 “异常安全性编程” 的脊柱。

```
namespace std {
    template<typename T>
    void swap(T& a, T& b) {
        T temp(a);
        a = b;
        b = temp;
    }
}
```

上面这种实现方法有时候显得有些慢，因为对于某些类型而言，这些复制动作无一必要。想要更加快速，最 主要的方式就是 “以指针指向一个对象，内含真正的数据”，这种设计就是所谓的 pimpl 手法(pointer to implementation)。

```
class WidgetImpl {
public:
private:
  int a, b, c;
  std::vector<double> v;
};

class Widget {
public:
  Widget(const Widget& rhs);
  Widget& operator=(const Widget& rhs) {
    *ipml = *(rhs.ipml);
  }

private:
  WidgetImpl* ipml;
}
```

上面就是一个使用 pimpl 手法实现的 Widget class。当使用 swap 时，直接调换 ipml 指针即可。 但是我们需要告知 std::swap 需要这么做，否则仍然是默认行为。一个做法是将 std::swap 针对 Widget 进行特化。

```
namespace std {
  template <>
  void swap<Widget>(Widget& a, Widget& b) {
    swap(a.ipml, b.ipml);
  }
}
```

其中 template <> 用来表示这是 std::swap 的一个全特化版本，而后面的 表示这一特化 版本针对 Widget 设计。通常我们不能改变 std 命名空间中的任何东西，但是我们可以为标准 template 制作特化版本，使它专属于我们自己的 classes。

但是由于 ipml 是 private 的，因此需要这个特化版本声明为 friend，但是形式和以前的大不一样。

```
class Widget {
public:
  void swap(Widget& other) {
    using std::swap;
    swap(ipml, other.ipml);
  }

private:
  WidgetImpl* ipml;
};

namespace std {
  template <>
  void swap<Widget>(Widget& a, Widget& b) {
    a.swap(b);
  }
}
```

特化函数调用 Widget 的 swap 成员函数。

如果假设 Widget 和 WidgetImpl 都是 class templates 呢？也许会想到偏特化，但这是不合理的 因为 c++ 允许 class template 进行偏特化，而 function template 则不允许偏特化。如果打算 偏特化一个 function template, 那么惯用的手段实际上是重载。

```
namespace std {
  template <typename T>
  void swap(Widget<T>& a, Widget<T>& b) {
    a.swap(b);
  }
}
```

但是对于 std 这个特殊命名空间而言，客户可以对其中的 templates 进行特化，但不允许添加新的 templates。这也就意味着重载是不行的。一个好的方案是使用一个 non-member swap 来调用 member swap，但是不再将 non-member swap 声明为全特化或重载。

```
namespace WidgetStuff {

template <typename T>
class Widget {};

template <typename T>
void swap(Widget<T>& a, Widget<T>& b) {
    a.swap(b);
}
}
```

了解了所有的 swap 写法后，需要注意的一点是，最好能够同时提供 std::swap 的特化版本以及 non-member swap。这是为了允许客户有多种选择，当其中之一失效时仍然可以有正确的保证。

```
template <typename T>
void doSomething(T& obj1, T& obj2) {
    using std::swap;
    swap(obj1, obj2);
}
```

上述这个代码中，如果 T 对应的没专属 swap 存在则调用，否则会调用 std::swap。而这个过程中，编 译器仍然喜欢 T 的特化版本，因此会调用针对 T 的 std::swap 特化版本。但是千万要注意不要使用， std::swap 形式，而是使用 swap 形式，否则固定调用的就是 std::swap。

最后一点劝告，成员版的 swap 函数不要抛出异常，因为它帮助 classes 提供强烈的异常安全性保障。 这一约束仅适用于 成员版，而非适用于 non-member，因为 swap 缺省版本基于 copy，而一般情况下 两者都允许抛出异常。不抛出异常和高效置换是相等的，因为高效的基础在于置换几乎面向 内置类型 进行 操作(pimpl 手法的底层指针)，而内置类型绝对不会抛出异常。