《现代C++编程实现》电子版：https://leanpub.com/c01

第9章 运算符重载
Operator Overloading

YouTube频道： hwdong

博客： hwdong-net.github.io

B站: hw-dong

运算符：比函数更直观

- 再复杂的计算都归结为由运算符和操作数组成的表达式计算过程。表达式是程序最基本的计算单元。
- 对于基本类型的变量，运算符的含义是固定的。例如，加法运算符 + 在 int、float、double`等类型上始终表示数值相加。
- 人们熟悉运算符，运算符比函数更直观

```
add(x, multiply( y, z))
x+y*z
```

运算符对内在类型的支持

- C++提供了一些运算符对内存类型数据进行运算，如算术运算符、输入输出运算符等。

  doble a,b;

  double c += a/c

  std::cout<<a<<“ ”<<b;

用户定义类型

- 对用户定义类型，程序员需自己定义相应的运算符。

- 如对string类型:

```
std::string s1{ "hello" }, s2{ "world" }, s3;
s3 = s1 + s2;    //用+、=对string对象进行运算符
std::cout << s3;  //用<<将s3输出到cout中
```

运算符函数

- C++中，每个运算符实际就是一个函数，称为**运算符函数**，运算符函数的完整名称是**operator**关键字加上运算符构造的。
- 如加法运算符**+**的函数是**operator+**

s1+s2　　s1.operator+(s2)

```
std::string s1{ "hello" }, s2{ "world" }, s3;
s3 = s1 + s2;     //s3.operator=(operator+(s1,s2));
std::cout << s3;  //operator<<(std::cout, s3);
```

运算符重载

- 对用户定义类型，重新定义运算符函数，称为**运算符重载。**

```
#include <string>
#include <iostream>
std::string operator+(const std::string& s1, const std::string& s2) {
    std::string s;
    /*  */
    return s;
}
std::ostream& operator<<(std::ostream& out, const std::string& s) {
    /* ... */
    return out;
}
```

运算符重载的2种方式

- 成员函数：将运算符函数定义为类的成员函数。

  a.operator@(b)

- 外部函数：将运算符函数定义为外部函数（全局函数）。

  operator@(a,b)

- 大多数运算符可以采用上述任意一种重载方式。但某些特殊运算符，只能采用其中一种方式重载。例如：赋值运算符只能作为类的成员函数重载。

```cpp
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    Point operator+(const Point& other) {
        return  Point(x + other.x, y + other.y);
    }

    friend void print(const Point & p);
};
void print(const Point& p) {
    std::cout << p.x << "," << p.y;
}
```

return Point(this->x + other.x, this->y + other.y);

```cpp
int main() {
    Point P{ 2,3 }, Q{ 4,5 };
    print(P + Q);
}
```

P.operator+(Q)

operator+也可以作为外部（全局）函数实现

```
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    friend Point operator+(Point P, const Point& other);
    friend void print(const Point & p);
};
Point operator+(Point P, const Point& other) {
    return  Point(P.x + other.x, P.y + other.y);
}

void print(const Point& p) {
    std::cout << p.x << "," << p.y;
}
```

```
int main() {
    Point P{ 2,3 }, Q{ 4,5 };
    print(P + Q);
}
```

operator+(P,Q)

二元运算符

- 作为成员函数实现时，只能包含1个参数。

```
class Point{
   //…
Public:
  Point operator+(const Point &other) const { // const 成员函数
        return  Point(x + other.x, y + other.y);
    }   //…
};
```

二元运算符

- 作为成员函数实现时，只能包含1个参数。
- 因为调用这个函数的对象就是第一个操作数（左操作数）

  P + Q 实际上是 P**.operator+** (Q)

```
class Point{
  //…
Public:
  Point operator+(const Point &other) const { // const 成员函数
       return  Point(x + other.x, y + other.y);
    }   //…
};
```

二元运算符

- 作为成员函数实现时，只能包含1个参数。
- 在成员函数中，有一个隐含的 **this**指针就指向这个调用的对象。

```
Point Point::operator+( const Point & other) const { // const成员函数
    return  Point(this->x + other.x, this->y + other.y);
}
```

二元运算符

- 作为外部函数实现时，必须包含两个参数。
- P + Q 实际上调用的是 **operator+**(Point P, const Point &other)，也就是普通的外部函数。

```
class Point{
  //…
Public:
  friend Point operator+(const Point &P, const Point &other)
   //…
};
Point operator+(const Point &P, const Point &other) {
      return  Point(P.x + other.x, P.y + other.y);
}
```

运算符函数: 参数个数

- 二元运算符

  作为成员函数，有且只有一个参数： a.operator@(b)

  作为外部函数，有且只有2个参数：operator@(a,b)

- 一元运算符

  作为成员函数，参数列表为空： a.operator++()

  作为外部函数，有且只有1个参数：operator-(a)

```cpp
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    Point operator-()const {
        return Point(-this->y, -this->x);
    }
    void print() { std::cout << x << "," << y; }
    //...
};

int main() {
    Point P(3, 4);
    Point Q = -P;
    Q.print();
}
```

```cpp
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    friend Point operator-(const Point& p);
    void print() { std::cout << x << "," << y; }
    //...
};
Point operator-(const Point& p) {
    return Point(-p.y, -p.x);
}
int main() {
    Point P(3, 4);
    Point Q = -P;
    Q.print();
}
```

成员函数和外部函数重载的主要区别

- 作为外部函数重载的运算符的**第一个操作数可以是能转化为这个类类型的变量。**

```
class Point {
    double x{}, y{};
public:
    Point(double x=0, double y=0) :x{ x }, y{ y }{}
    friend Point operator+(Point P, const Point& other);
    //…
};
Point operator+(Point P, const Point& other) {
    return  Point(P.x + other.x, P.y + other.y);
}
```

```
int main() {
    Point P(3, 4),Q(-2,6);
    P + 2;
    2 + P;
    P + Q;
}
```

成员函数和外部函数重载的主要区别

- 作为成员函数重载的运算符的第一个操作数**必须是这个类的对象**，不能是其他可转化为这个类类型的变量。

```
class Point {
    double x{}, y{};
public:
    Point(double x = 0, double y = 0) :x{ x }, y{ y }{}
    Point operator+(const Point& other) {
        return  Point(x + other.x, y + other.y);
    }
    //…
};
```

```
int main() {
    Point P(3, 4),Q(-2,6);
    P + 2;
    2 + P;
    P + Q;
}
```

赋值运算符=

- 对于Point对象，可以直接用赋值运算符 = ：

  P = Q;

- 对于Point类，编译器生成的默认赋值运算符operator = ()如下：

```
class Point {
    double x, y;
public:
    Point& operator=(const Point& other) {
        if (this != &other) {
            x = other.x; y = other.y;
        }
        return *this;
    }
}
```

赋值运算符=

- 只能以成员函数的形式重载赋值运算符operator=()，并且重载的函数最后必须返回自引用（*this）。
- 对于占用资源的类，默认的赋值运算符函数会导致“共享同一资源”问题。如string、vector。应该重新定义赋值运算符函数！

下标运算符[]

- 许多情况下，一个对象的数据并不是单一的值，而是由多个值组成的组合。例如，在 Point 类中，我们使用两个double 类型的变量分别表示点的 x 坐标和 y 坐标。

- 为了方便访问这类多值对象中的各个组成部分，可以将它们映射到一个“下标”，就像数组一样。通过重载下标运算符[]（即operator[]），我们就可以用数组的方式访问这些成员变量，例如 P[0] 表示 x，P[1] 表示 y。

下标运算符[]

- 下标运算符通常定义2个版本，
- 一个是返回可以被修改的引用，即可以作为赋值运算符的左操作数，

  T& operator[](int i) ;
- 另外一种是const成员函数，返回的是一个值，可用作赋值运算符的右操作数。

  T operator[](int i) const;

```cpp
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    double& operator[](int i) {         //返回对象的引用
        if (i == 0) return x;
        else if (i == 1) return y;
        else throw "下标非法";
    }
    double operator[](const int i) const { //返回值的const函数
        if (i == 0) return x;
        else if (i == 1) return y;
        else throw "下标非法";
    }
    friend void print(const Point & p);
    //...
};
```

```
void print(const Point & p) {
    std::cout << p.x << "," << p.y;
}
int main() {
    Point P{ 2,3 };
    P[0] = 4;         //P[0]调用的是引用版本，
    P[1] = P[0];      //P[1]调用的是引用版本，P[0]调用的是const版本
    print(P);
}
```

输入输出运算符<<和>>

- 可以为用户定义的类型重载输入输出运算符 >> 和 <<，例如：

```
class Point {
public:
    double x{}, y{};
    Point(double x=0, double y=0) : x{x}, y{y} {}

    friend std::ostream& operator<<(std::ostream& out, const Point& p);
    friend std::istream& operator>>(std::istream& in, Point& p);
};
```

输入输出运算符<<和>>

- 可以为用户定义的类型重载输入输出运算符 >> 和 <<，例如：

```
std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;  // 返回输出流引用，支持链式输出
}

std::istream& operator>>(std::istream& in, Point& p) {
    in >> p.x >> p.y;
    return in;  // 返回输入流引用，支持链式输入
}
```

输入输出运算符<<和>>

- 其中的输入流或输出流参数必须是引用

```
#include <iostream>
class Point {
    double x, y;
    //…
public:
    Point(double x=0, double y=0) :x{ x }, y{ y }{}
    friend std::ostream& operator<<(std::ostream& out, const Point& p);
    //…
};
std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "(" << p.x << "," << p.y << ")";
    return out;
}
```

```
int main() {
    Point P;
    std::cout << P;
}
```

为什么使用引用参数和返回引用?

- 若按值传递而非引用，流对象会被拷贝，导致无法操作原有流，链式输出（如 std::cout << A << B）也将失效。

- 使用引用后，函数直接操作传入的流对象，确保输出或输入的连续性。

比较运算符: <、 >、 <=、 ==、 …

- 作为成员函数的比较运算符

```
#include <iostream>
class Point {
    double x, y;
    //…
public:
    bool operator<(const Point& other);
    bool operator==(const Point& other);
    //…
};

bool Point::operator<(const Point& other) {
    if (x == other.x) return y < other.y;
    return x < other.x;
}
bool Point::operator==(const Point & other) {
    return x == other.x && y == other.y;
}
```

```
int main() {
    Point P{ 2,3 }, Q(3, 2);
    if (P < Q || P == Q) std::cout << "P<=Q";
    else std::cout << "P>Q";
}
```

比较运算符: <、 >、 <=、 ==、 …

- 作为外部函数的比较运算符

```
#include <iostream>
class Point {
    double x, y;
    //…
public:
    friend bool operator<(const Point P, const Point& Q);
    friend bool operator==(const Point P, const Point& Q);
    //…
};

bool operator<(const Point P, const Point& Q) {
    if (P.x == Q.x)return P.y < Q.y;
    return P.x < Q.x;
}
bool operator==(const Point P, const Point & Q) {
    return P.x == Q.x && P.y == Q.y;
}
```

```
int main() {
    Point P{ 2,3 }, Q(3, 2);
    if (P < Q || P == Q) std::cout << "P<=Q";
    else std::cout << "P>Q";
}
```

重载其他比较运算符

- 比较运算符并不是完全独立的，由于 > 可由 < 反转，!= 可由 == 取反，<= 和 >= 可通过 < 和 == 组合，**只要重载了 < 和 ==**，就可以用它们来实现其他比较运算符。

```
bool Point::operator<=(const Point& other) const {
    return *this < other || *this == other;
}
```

为每种类型重载所有比较运算符较为繁琐且重复。

<=> 运算符

- C++20 引入了三向比较运算符 <=>（称为太空船运算符，因形似飞船），只需重载一次，编译器即可自动生成 <、<=、>、>=、== 和 !=。

```
#include <compare>
#include <iostream>

class Point {
    double x{}, y{};
public:
    explicit Point(double x, double y) : x{ x }, y{ y } {}

    // 使用 C++20 的 <=> 运算符
    auto operator<=>(const Point& other) const = default;
};
```

- `= default` 表示按成员顺序比较（先 `x` 后 `y`），适合全序比较：
  - 如果 `x < other.x`，则 `P < Q` 为 `true`。
  - 如果 `x == other.x`，则比较 `y` 和 `other.y` 的大小。
  - 否则，返回 `false`。
- 其返回类型是`std::strong_ordering` 表示全序，但可以用auto自动推断。
- 编译器根据 <=> 自动生成其他比较运算符。

```
class Point {
    double x{}, y{};
public:
    explicit Point(double x, double y) : x{ x }, y{ y } {}

    // 使用 C++20 的 <=> 运算符
    auto operator<=>(const Point& other) const = default;
};
```

示例：比较 Point 对象

```
int main() {
    Point P{ 2, 3 }, Q{ 3, 2 };

    if (P < Q) std::cout << "P < Q\n";
    if (P <= Q) std::cout << "P <= Q\n";
    if (P > Q) std::cout << "P > Q\n";
    if (P >= Q) std::cout << "P >= Q\n";
    if (P == Q) std::cout << "P == Q\n";
    if (P != Q) std::cout << "P != Q\n";
}
```

输出结果：

```
P < Q
P <= Q
P != Q
```

```
class Point {
    double x{}, y{};
public:
    explicit Point(double x, double y) : x{ x }, y{ y } {}

    // 使用 C++20 的 <=> 运算符
    auto operator<=>(const Point& other) const = default;
};
```

9.6.5 C++20 中 `<=>` 和 `==` 的详细规则：

1. **只重载 <=>**：编译器自动生成 <、<=、>、>=、== 和 !=。
2. **重载 <=> 和 ==**：编译器根据 <=> 生成 <、<=、>、>=，根据 == 生成 !=。
3. **只重载 ==**：编译器仅生成 !=，其他运算符需手动定义。

原理：`<=>` 同时提供大小比较和相等性判断（如 `std::strong_ordering::equal` 表示相等），因此可推导所有比较运算符。

<=> 运算符的三种返回类型

- 返回的类型有三种：std::strong_ordering、std::weak_ordering 和 std::partial_ordering。这三种类型用于描述比较的结果，它们的区别在于比较的严格程度。

std::strong_ordering

用途：用于全序比较。

说明：适用于完全可比较的对象，每两个对象都有明确的顺序关系。

返回值包括 less、equal、greater。

示例：`5 <=> 10` 返回 `std::strong_ordering::less`，表示 `5 < 10`。

std::weak_ordering

用途：用于弱序比较。

说明：适用于所有对象可比较但相等对象可能仅等价的场景。

返回值包括 less、equivalent、greater。

示例：大小写无关比较中，"Apple" <=> "apple" 返回 std::weak_ordering::equivalent，表示等价。

std::partial_ordering

用途：用于偏序比较。

说明：适用于可能存在不可比较对象（如浮点数 NaN）的场景。

返回值包括 less、equivalent、greater、unordered。

示例：NaN <=> 5.0 返回 std::partial_ordering::unordered，表示不可比较。

总结：

- **std::strong_ordering**：适用于所有对象之间都能明确比较大小的情况（完全有序）。
- **std::weak_ordering**：适用于所有对象可比较但相等对象可能仅等价的情况（弱序）。
- **std::partial_ordering**：适用于某些对象可能不可比较的情况，如浮点数 `NaN`（部分序）。

通过这三种返回类型，C++20 的 `<=>` 运算符可以处理更多复杂的比较逻辑，使得代码更加灵活和强大。

函数调用运算符()

- 对一个类型，可以定义**函数调用运算符operator()**
- 根据参数n计算x^n+y^n

```
class Point {
    double x{}, y{};
public:
    double operator()(int n = 2) {
        if (n <= 1) return std::abs(x) + std::abs(y);
        return std::pow(x, n) + std::pow(y, n);
    }
    //...
};
```

- operator() 使得 Point 类的对象可以像函数一样被调用。当我们创建一个 Point 类型的对象 P后，可以通过圆括号来传递参数，计算并返回结果：

```
int main() {
    Point p{2.0, 3.0};
    std::cout << p(3) << '\n';  //P(3) 调用了函数调用运算符 operator()
    return 0;
}
```

```
class Point {
    double x{}, y{};
public:
    double operator()(int n = 2) {
        if (n <= 1) return std::abs(x) + std::abs(y);
        return std::pow(x, n) + std::pow(y, n);
    }
    //...
};
```

函数对象

- P(2)这种使用方式使对象P看起来像一个函数一样，可以通过圆括号()接受参数。因此，将这个Point类的对象称为**函数对象**。

```
int main() {
    Point p{2.0, 3.0};
    std::cout << p(3) << '\n';  //P(3) 调用了函数调用运算符 operator()
    return 0;
}
```

函数对象

- **函数对象**（Function Object，或称为 Functor）是指定义了函数调用运算符 operator() 的类的对象。这允许对象像函数一样被调用。

```
int main() {
    Point p{2.0, 3.0};
    std::cout << p(3) << '\n';  //P(3) 调用了函数调用运算符 operator()
    return 0;
}
```

函数对象

- 函数对象灵活多用，在 C++ 标准库中广泛应用于算法，如 std::sort，可通过自定义函数对象指定排序规则。

```
int arr[] = {5, 2, 9, 1, 5};
std::sort(arr, arr + 5, [](int a, int b) { return a < b; });
```

这里的 Lambda 表达式被编译器转换为一个匿名类的对象（称为闭包对象），该类实现了 operator()，定义了从小到大的排序规则。

函数对象

- C++ 编译器遇到 Lambda 表达式（如 [](int a, int b) { return a < b; } ）时，会生成一个匿名类

```
struct Anonymous {
    bool operator()(int a, int b) const { return a < b; }
};
```

函数对象

- 当然，你也可以定义非 Lambda 的函数对象并传给 sort 算法：

```
struct Compare {
    bool operator()(int a, int b) const { return a < b; }
};
std::sort(arr, arr + 5, Compare{});
```

类型转换运算符

- 带一个参数的构造函数实际上定义了一个从参数类型到类类型的隐式类型转换。

```
class X{
   public:
        X(int a) {/*…*/}
};
```

- 反过来，也可以定义一个类型转换运算符用于将类类型隐式转换为其他的类型。
- 类型转换运算符同样必须作为成员函数实现，其格式是：

```
operator type () const
```

- 类型转换函数也没有返回类型

```
class Point {
    double x{}, y{};
public:
    operator double() const {
        return x * x + y * y;
    }
    Point(double x, double y) :x{ x }, y{ y }{}
};

#include <iostream>
int main() {
    Point P(3, 4);
    double d = P;  //对象P隐式转化为double类型的值，再对d初始化
    std::cout << d << '\n';
}
```

隐含类型转换引起的歧义

- 一个参数的构造函数和类型转换运算符定义的隐式类型转换有时会引歧义

```
class A {
    int a;
public:
    A(int);          //一个参数的构造函数定义了int类型到A类的自动类型转换
    operator int() { return a; }   //类型转换运算符可以将A类型对象自动转换为int类型
    friend A operator+(const A& a1, const A& a2); //2个A类型的对象相加
};
A operator+(const A& a1, const A& a2) {
    A ret(0);
    //...
    return ret;
}
```

```
class A {
    int a;
public:
    A(int);          //一个参数的构造函数定义了int类型到A类的自动类型转换
    operator int() { return a; }   //类型转换运算符可以将A类型对象自动转换为int类型
    friend A operator+(const A& a1, const A& a2); //2个A类型的对象相加
};
A operator+(const A& a1, const A& a2) {
    A ret(0);
    //...
    return ret;
}


int main() {
    A a{ 1 };
    int i = 1, z;
    z = a + i; //错：到底是 A+A还是int+int
}
```

强制类型转换消除歧义性

```
int main() {
    A a{ 1 };
    int i = 1, z;
    z = static_cast<int>(a) + i;  //或z = (int)a + i;
    //  z = a + static_cast<A>(i);  //或z = a + A(i);
}
```

自增和自减运算符

- 自增运算符++和自减运算符--都是一元运算符，且必须作为类的成员函数实现。如：

```
int x;
++x;
x++;
```

前缀和后缀

在重载这两个运算符时，必须区分前缀形式和后缀形式：

- **前缀形式（prefix）：** 运算符出现在对象前面，例如 `++x`、`--x`。
- **后缀形式（postfix）：** 运算符出现在对象后面，例如 `x++`、`x--`。

前缀和后缀

C++ 规定，通过在函数参数中添加一个 int`类型的**占位参数**来区分前缀和后缀形式：：

- **前缀形式：** 不带参数，例如 `Point& operator++();`
- **后缀形式：** 带一个 `int` 参数，例如 `Point operator++(int);`。

  这个参数不会被使用，仅用于区分语法。

```
class Point {
    double x{}, y{};
public:
    Point(double x, double y) : x{x}, y{y} {}

    // 前缀自增运算符：返回对象自身的引用
    Point& operator++();

    // 后缀自增运算符：返回旧值的副本
    Point operator++(int);
};
```

```cpp
Point& Point::operator++() {
    ++x; ++y;
    return *this;
}

Point Point::operator++(int) {
    Point temp(*this);  // 保存当前对象的状态
    ++(*this);          // 调用前缀版本完成实际自增
    return temp;        // 返回旧值
}
```

- 前缀版本直接修改当前对象，并返回对当前对象的引用。
- 后缀版本先保存当前对象的副本，再自增自身，最后返回副本。

前缀与后缀自增的区别

- **前缀自增**（`++P`）：返回的是对象自身的引用，因此可以连续使用。例如：

```
++ ++P;   // 合法，++P 返回 P 本身，可以继续自增
(++P)++; // 合法，原因同上
```

- **后缀自增**（`P++`）：返回的是一个临时对象，而不是引用，因此不能连续使用。例如：

```
P++ ++;       // 错误，P++ 返回临时对象，无法继续自增
++(P++);      // 错误，原因同上
```

例子

```
int main() {
    Point P{ 2, 3 };
    ++ ++P;         // OK：前缀 ++ 返回对象自身的引用
    (++P)++;        // OK：前缀 ++ 返回引用，再后缀自增
    P++ ++;         // 错误：后缀 ++ 返回临时对象，无法继续自增
    ++(P++);        // 错误：原因同上
}
```

可以重载的运算符

```
+     -     *     /     %     ^     &     |     ~     !
+=    -=    *=    /=    %=    ^=    &=    |=    <<=   >>=
>     <     >=    <=    ==    !=    <<    >>
&&    ||    ++    --    ->    ->*   ,     *     T
[]    ()    new new[] delete delete[]
```

- 这里的T表示的是类型转换运算符。
- 有的运算符可能有2种含义，比如&可以作为位运算符也可以作为取地址符、-可以作为减法运算符也可以作为负号运算符。
- 有的运算符只能作为成员函数重载，如：=、[]、()、T(类型转换运算符)。

- 也有的运算符只能作为外部函数重载，如：new、 new[]、 delete、 delete[]。
- **也有一些运算符不能被重载**，如：

  :: 作用域运算符

  . 成员访问运算符

  .* 成员选择运算符

  ?:条件运算符

  sizeof 查询对象的大小

  typeid查询对象的类型

- 运算符定义不能违背约定的语法，如不能将一元定义成二元或三元，反过来也是一样。也不应该违背运算符的语义，如不能将+运算符定义成相乘的含义、赋值运算符的返回类型应返回引用而不是值。

实战：矩阵

```
#include <cassert>
class Matrix {
    double* data{ nullptr };
    int m{ 0 }, n{ 0 };
public:
    Matrix(int m = 0, int n = 0);
    explicit Matrix(int m) : Matrix(m, m) {}
    Matrix(const Matrix& M);
    Matrix& operator=(const Matrix& M);
    ~Matrix();
    double operator()(int i, int j) const;
    double& operator()(int i, int j);
    Matrix& operator+=(const Matrix& M);
    Matrix& operator-=(const Matrix& M);
    friend Matrix operator+(const Matrix& A, const Matrix& B);
    friend Matrix operator-(const Matrix& A, const Matrix& B);
    friend void swap(Matrix& A, Matrix& B) noexcept;
    int rows() const { return m; }
    int cols() const { return n; }
};
```

关注我

博客： hwdong-net.github.io

Youtube频道

hwdong

@hwdong · 5.01K subscribers · 558 videos

博客：https://hwdong-net.github.io >

youtube.com/c/4kRealSound and 4 more links