书籍地址：

https://github.com/kelthuzadx/EffectiveModernCppChinese

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C++中右值指的的临时值或常量，更准确的说法是保存在CPU寄存器中的值为右值，而保存在内存中的值为左值。

比如说一个常数5，我们在使用它时不会在内存中为其分配一个空间，而是直接把它放到寄存器中，所以它在C++中就是一个右值。

\*通过std::move可以将一个左值转成右值。

\*当编译器遇到这类形式的时候它会使用引用折叠技术，将它们变成我们熟悉的格式。其规则如下：

int&& 折叠为 int&

int&&& 折叠为 int&

int&&& 折叠为 int&

int&&&& 折叠为 int&&

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. 类型推导

\*Item 1: 理解模板类型推导

\*Item 2: 理解auto类型推导

+auto x3 = { 27 }; //类型是std::initializer\_list<int>,值是{27}

+initializer\_list对象中的元素永远是常量值

\*Item 3: 理解decltype

+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.

+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.

\*Item 4: 学会查看类型推导结果

+类型推导工具给不了很多帮助， 需要真正理解类型推导。

1. Auto

\*Item 5: 优先考虑auto而非显示类型声明

+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。

+auto sz = v.size();

+for (const auto& p : m)

{

...

}

\*Item 6: auto推导若非己愿，使用显示类型初始化惯用法

+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

auto highPriority = static\_cast<bool>(features(w)[5]);

+显式类型初始器惯用法强制auto推导出你想要的结果

auto index = static\_cast<int>(d \* size());

1. 移步现代C++

\*Item 7: 区别使用()和{}创建对象

+如果你要使用std::initializer\_list构造函数，请三思而后行

+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

\*Item 8:优先考虑nullptr 而非0 和NULL

+

\*Item 9: 优先考虑别名声明而非typedefs

+typedef std::unique\_ptr<std::unordered\_map<std::string, std::string>> UPtrMapSS;

+using UPtrMapSS = std::unique\_ptr<std::unordered\_map<std::string, std::string>>;

\*Item 10: 优先考虑限域枚举而非未限域枚举

+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)。

\*Item 11: 优先考虑使用delete函数而非使用未定义的私有声明

+

\*Item 12: 使用override声明重载函数

+

\*Item 13: 优先考虑const\_iterator而非iterator

\*Item 14: 如果函数不抛出异常请使用noexcept

+noexcept对于移动语义, swap，内存释放函数和析构函数非常有用

\*Item 15: 尽可能的使用constexpr

+所有constexpr对象都是const，但不是所有const对象都是constexpr。

\*Item16: 让const成员函数线程安全

+

\*Item 17: 理解特殊成员函数的生成

+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。

+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

+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。

+Rule of Three: 拷贝构造，赋值构造，析构

+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

+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。

+一般基类需要virtual 析构函数， 所以一般这样定义类。

class Base {

public:

virtual ~Base() = default;

Base(Base&&) = default;

Base& operator=(Base&&) = default;

Base(const Base&) = default;

Base& operator=(const Base&) = default;

...

};

1. 智能指针

\*Item 18:对于独占资源使用std::unique\_ptr

+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

\*Item 19:对于共享资源使用std::shared\_ptr

+

class Widget : public std::enable\_shared\_from\_this<Widget> {

public:

void process()

{

processedWidgets.emplace\_back(shared\_from\_this());

}

};

+std::shared\_ptr不能处理的另一个东西是数组

\*Item 20: 当std::shard\_ptr可能悬空时使用std::weak\_ptr

+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

\*Item 21: 优先考虑使用std::make\_unique和std::make\_shared而非new

+

\*Item 22: 当使用Pimpl惯用法，请在实现文件中定义特殊成员函数

+struct Impl 来实现，避免Widget头文件有变化，需要重新编译

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

}

#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>())

{}

1. 右值引用，移动语意，完美转发

\*Item 23: 理解std::move和std::forward

+std::move总是无条件的将它的参数转换

std::forward是有条件的转换:它只把由右值初始化的参数，转换为右值

\*Item 24: 区分通用引用与右值引用

+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

\*Item25: 对右值引用使用std::move，对通用引用使用std::forward

+

坚持要在局部变量上使用std::move返回

Widget makeWidget(Widget w) {

...

return std::move(w);

}

+

Widget makeWidget() {

Widget w; //local variable

... // configure w

return std::move(w); // move w into return value(don't do this!)

}

+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: 避免在通用引用上重载

\*Item27: 熟悉通用引用重载的替代方法

\*Item28：理解引用折叠

+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

\*Item30: 熟悉完美转发的失败case

1. Lambda表达式

\*Item31: 避免使用默认捕获模式

+闭包实现： 1.重载operator(); 2.lambda表达式; 3.std::bind

+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；

\*Item 32:使用初始化捕获来移动对象到闭包中

+

auto func = [pw = std::move(pw)] // init data mbr in closure w/ std::move(pw)

{ return pw->isValidated() && pw->isArchived(); };

\*Item 33:对于std::forward的auto && 形参使用decltype

+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。

+

auto f1 = [](auto&& param)

{

return func(normalize(std::forward<decltype(param)>(param)));

};

\*Item 34:优先考虑lambda表达式而非std::bind

+

1. 并发API

\*Item 35:优先考虑基于任务的编程而非基于线程的编程

+auto fut = std::async(doAsyncWork);

+std::threadAPI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

\*Item36: 确保在异步为必须时，才指定std::launch::async

\*Item 37: Item 37:从各个方面使得std::threads unjoinable

\*Item 38:关注不同线程句柄析构行为

+

{ // begin block

std::packaged\_task<int()> pt(calcValue);

auto fut = pt.get\_future();

std::thread t(std::move(pt));

...

} // end block

\*Item 39:考虑对于单次事件通信使用void

\*Item 40:对于并发使用std::atomic，volatile用于特殊内存区

+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”

+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。

1. 微调

\*Item 41:对于那些可移动总是被拷贝的形参使用传值方式

+

Widget w;

std::string name("Bart");

w.addName(name); // call addName with lvalue 左值拷贝

w.addName(name + "Jenne"); // call addName with rvalue 右值移动

\*Item 42:考虑就地创建而非插入

+