效率

条款16：谨记 80-20 法则

一个程序 80% 的资源用于 20% 的代码身上。尽可能的以作多的数据来分析写出的软件，并借助分析器来提 高软件的效率。

条款17：考虑使用 lazy evaluation (缓式评估)

使用 lazy evaluation 来撰写 classes，使他们延缓运算，直到某些运算结果刻不容缓的被迫切需要为 止。

Reference Counting (引用计数)

这种行为类似于 “写时拷贝” 的策略。

```
string s1 = "Hello"
string s2 = s1;
```

上述这个过程的实现，最直观的过程是 s2 拷贝了一份 s1 的内容。这意味着，s2 要在 heap 中申请一块 空间，并调用 strcpy 来进行字符串的复制。这个过程被称为 eager evaluation。这显然时效率过低的。

而 lazy 的做法时，让 s2 和 s1 以引用的方式共享 s1 的值。唯一需要额外做的是对共享者进行记录。 这一过程对于用户而言时透明的，用户完全不会感知两者共用。只有当其中的某个对象内容被更改时，才为其 提供一个副本，并让副本成为它的私有数据进行更改。

区分读和写

在实际的调用过程中，为了满足写时拷贝的要求，一个很显然的特点在于需要区分 读 和 写 的动作。但是一 般我们无能为力。但是结合 proxy classes 可以延缓决定 “究竟是读还是写”，直到能够确定其答案为止。

lazy fetching (缓式取出)

在使用大型对象时，可能包含许多字段。往往会将这些内容存储在一个数据库中再进行读取。在读取过程中由于体积较大，数据库的相关成本可能很高。因此可以使用 lazy 的方案来进行处理。

```
typedef long long ObjectID;

class LargeObject {
public:
  LargeObject(ObjectID id);
  const string& field1() const;
  int field2() const;
  double field3() const;
  const string& field4() const;

private:
  ObjectID oid;
```

```
  mutable string *field1Value;
  mutable int *field2Value;
  mutable double *field3Value;
  mutable string *field4Value;
};

LargeObject::LargeObject(ObjectID id) : oid(id), field1Value(0),
  field2Value(0), field3Value(0), field4Value(0) {} // 在调用构造函数时，并不读
取内容

int LargeObject::field2() const {
  if (field2Value == 0) {
    *field2Value = 1; // 这里是从数据库单独读取 field2 内容并赋值
  }

  return *field2Value;
}
```

从上面可以看到在构造过程中只创建一个对象的“外壳”，而不从磁盘中读取任何字段的数据。只有当某个对象被需要时才进行读取。

需要注意的是，即便在 const member functions 中也应该保证能修改才可以实现 lazy fetching。 这里使用 mutable 关键字来实现，除此之外也可以使用 fake this 的方案来实现。

```
{
private:
  ...
  int *field2Value;
  ...
}
int LargeObject::field2() const {
  if (field2Value == 0) {
    LargeObject * const fakeThis = const_cast<LargeObject * const>(this);
    this->field2Value = 1;
  }

  return *field2Value;
}
```

这种结果是否繁琐，但是可以用 smart pointers 来自动完成。事实上这也是一种拖延政策，因为最后在 实现 smart pointer class 时仍然需要使用 mutable 成员变量。

lazy Expression Evaluation (表达式缓式评估)

有些表达式可以使用 lazy 方式处理，比如矩阵加法。通常 operator+ 是一个 eager evaluation，但 面向大量运算时效率往往低下。

```
template <class T>
class Matrix {};
```

```
Matrix<int> m1(1000, 1000);
Matrix<int> m2(1000, 1000);
Matrix<int> m3 = m1 + m2;
```

有时，我们并不需要获得整个 m3 的内容，因此我们可以拖延时间。m3 提供两个指针和一个 enum 来实现，前者指向加法的两个部分，而后者则用来表示加减乘除运算符。

等待真正需要类似 `cout << m3[4]` 这样的语句时，再对该值进行计算。

lazy evaluation 的使用需要完成大量的工作，但是通常能够在程序执行时节省大量的时间空间，尽管对 于必要的计算 lazy evaluation 并不会节省任何工作和时间，甚至更加缓慢以及更多占用内存用量。

条款18：分期摊还预期的计算成本

前一条款强调了 lazy evaluation 的使用，而现在将要强调 over-eager evaluation 的使用。

```
template <class NumericalType>
class DataCollection {
public:
  NumericalType min() const;
  NumericalType max() const;
  NumericalType avg() const;
}
```

正如上面这个例子，当需要计算最大值时，可以一并找出最大值和平均值，并记录。当下次调用 max 和 avg 时可以直接返回。这就是 over-eager evaluation 的做法，它相较于 lazy 和 eager 的做法效率更高。

一般 over-eager evaluation 的做法包括两种。

caching

caching 技术时将已经计算好的，而且有可能再被需要的内容保存在一块内存中，而非直接写入到磁盘等存储设备中。

```
int findCubicleNumber(const string &employeeName) {
  typedef map<string, int> CubicleMap;
  static CubicleMap cubes; // 使用一个 map 对象作为局部缓存使用

  CubicleMap::iterator it = cubes.find(employeeName); // 从缓存中查找记录

  if (it == cubes.end()) { // 如果缓存中没有
    int cubicle = lookingUp(employeeName); // 从数据库中进行查找
    cubes[employeeName] = cubicle; // 放入缓存
    return cubicle;
  } else {
    return (*it).second();
  }
}
```

上述代码使用一个 map 对象作为缓存机制进行存储部分数据，而不必非要从磁盘中获取数据，进而增加效率。

prefetching

caching 是一种 “分期摊还预期计算的成本” 的一种做法，prefetching 则是另一种做法。它基于 locality of refernece 的现象完成。

```
template <class T>
T &DynArray<T>::operator[](int index) {
  if (index < 0) {
    throw exception;
  }

  if (index > size) {
    int diff = index - size;
    operator new(diff + index);
    size = diff + index;
  }

  return array[index];
}
```

这里将空间扩充缺失大小的二倍大小，这样预先取出后缀的空间，可以防止当下次使用 index + 1 时，重新 调用扩展的功能。

over-eager evaluation 的行为对于程序的性能提升非常突出，这种技巧应当贯彻始终。

条款19：了解临时对象的来源

对于 c++ 而言，局部对象时一个不可见的存在它并不会出现在源代码中。它的出现发生在，你产生了一个 non-heap object 而没有为他命名。临时对象可能会消耗大量成本，所以应该尽可能的消除它们。

为了成功调用所产生的临时对象

这个过程发生在传递某些对象给某个函数的过程中，当类型与绑定类型不一致时发生隐式类型转换的情况。

```
size_t countChar(const string &str, char ch) {
  size_t count = 0;
  for (char c : str) {
    if (c == ch) {
      ++count;
    }
  }

  return count;
}

int main() {
```

```
  char buffer[1024] = "hello world";
  char c = 'l';

  cout << countChar(buffer, c) << endl;
}
```

这里 buffer 被转换为了 string 类型，通过调用 string constructor 产生的临时对象。这种转换 只会发生在 by value 或者 by reference-to-const 的场景下。reference-to-non-const 不会 通过编译，因为存在必定失败的结果。由于该过程一定会产生临时对象，因此 reference 是指向临时对象 的，如果你尝试对其中的内容进行修改，那么临时对象的内容会发生改变，而原有的 buffer 中的内容实际 不会发生任何变化。这是未预期的行为，因此使用 const 来限制修改的发生。

返回对象时产生的临时对象

因为返回值没有名称，所以是个临时对象。大多数情况下这种行为很难进行优化，但是在优化策略中，最常见也是最有用的就是 “返回值优化”。

```
const Number operator+(const Number& lhs, const Number& rhs);
```

条款20：协助完成 “返回值优化 (RVO)”

函数返回一个对象，是严重应吸纳更效率的，这种代价无法消除。我们唯一可做的事情就是用 “特殊写法” 来 撰写函数，使它再返回对象时能够让编译器消除临时对象的成本——即 constructor arguments。

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

实际上这么做是否能被优化仍然是个未知数。只有当 c++ 编译器可以将临时对象优化，使他们不存在时，才 可以消除这些代价。

条款21：利用重载技术 (overload) 避免隐式类型转换

隐式类型转换的发生往往设计临时对象的产生，这个过程需要消耗大量的成本。使用重载技术来避免隐式类型转换十分重要。

```
class UPInt {
public:
  UPInt();
  UPInt (int value);
  const UPInt operator+(const UPInt &lhs, const UPInt &rhs);
};
```

```
UPInt upi1, upi2;
UPInt upi3 = upi1 + 10;
```

在上面的代码中，UPInt 与 int 的加法仍然可以成功，这归功于隐式类型转换。但正如前面所描述的，我 们要消除这种转换，利用 overload。

```
const UPInt UPInt::operator+(const UPInt &lhs, int rhs);
const UPInt UPInt::operator+(int lhs, const UPInt &rhs);
```

这样一来，编译器会选择重载函数，而不会进行隐式类型转换，减少了构建销毁过程。

除了 int 类型外，还可以适用于 string, char *, complex 等自变量类型。

条款22：考虑以操作符复合形式 (op=) 取代其独身形式 (op)

复合形式的操作符和独身形式的操作符不存在关系，如果需要建立一个容易维护的自然关系，好的方法是使用 **复合形式** 来实现 “独身形式”。

```
class Rational {
public:
  Rational &operator+=(const Rational &rhs);
  Rational &operator-=(const Rational &rhs);
};

const Rational operator+(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) += rhs;
}
const Rational operator-(const Rational &lhs, const Rational &rhs) {
  return Rational(lhs) -= rhs;
}
```

使用 template 来实现，甚至可以完全消除针对不同对象的独身形式的编写。

```
template <class T>
const T operator+(const T &lhs, const T &rhs) {
  return T(lhs) += rhs;
}
template <class T>
const T operator-(const T &lhs, const T &rhs) {
  return T(lhs) -= rhs; // 这里使用 T() 操作是为了取出常量性，需要编译器支持
}

Rational r1, r2;
Rational r3 = r1 + r2;
```

从效率角度思考

1. 符合操作符比独身版本效率更高

因为独身版本通常必须返回一个新对象，而这会导致构造和析构成本。

2. 同时提供两者更佳

同时提供两种方案，使用者可以决定如何使用 operator+ 是一个最佳的方案。基于前面所说，用 复合形式 来实现 独身版本，更凸显了 复合版本的重要性。

3. 独身形式的实现方式存在效率差距

对比下面两种实现方式

```
template <class T>
const T operator+(const T &lhs, const T &rhs) {
  return T(lhs) += rhs;
}

template <class T>
const T operator+(const T &lhs, const T &rhs) {
  T result(lhs);
  return result += rhs;
}
```

看起来两者似乎没有区别，但其实存在重要差异。第一个版本可以被 "返回值优化" 来消除临时变量的生成， 而或者则使用一个局部变量，促使无法使用 “返回值优化”，无形中增加了成本。

条款23：考虑使用其他程序库

不同的程序库即使提供相似的机能，也往往表现出不同的性能取舍策略。

条款24：了解 virtual function、multiple inheritance、virtual base class、runtime type identification 的成本

1. virual function

当一个虚函数被调用时，执行的代码必须对应于 “调用者的动态类型”。编译器通过 virtual tables(vtbls) 和 virtual table pointers(vptrs) 来高效的实现这种行为。

a. virtual tables

vtbl 是一个函数指针数组，每个声明（或继承）虚函数的 class 都有自己的一个 vtbl，其中的 entries 就是该 class 的各个虚函数实现体指针。

```
class C1 {
public:
  C1();
  virtual ~C1();
  virtual void f1();
```

```
  virtual int f2(char c) const;
  void f3() const;
}
```

```
C1's vtbl
  | -   | --------> implementions of C1::~C1 |
  | --- | ---------------------------------- |> implementions of C1::f1
  | -|--------> implementions of C1::f2
```

如果 C2 继承了 C1 并进行了些许的修改

```
class C2 : public C1 {
public:
  C2();
  virtual ~C2();
  virtual void f1();
  virtual void f4();
}
```

```
C2's vtbl
  | -   | --------> implementions of C2::~C2 |
  | --- | ---------------------------------- |> implementions of C2::f1
  | -   | --------> implementions of C1::f2 |
  | --- | ---------------------------------- |> implementions of C2::f4
```

每个拥有 虚函数 的 class 都会拥有一个 vtbl 空间，其大小与虚函数个数有关系。当程序中使用大量的 这种类，或者大量虚函数，那么 vtbls 会占用不少的内存。

关于 vtbls 的存储方式，编译器厂商分为两个阵营：

- 暴力式做法：每一个需要 vtbl 的目标文件内都产生一个 vtbl 副本，最后进行整合。
- 勘探式做法：vtbl 被产生于 “内含第一个 non-inline, non-pure 虚函数定义式” 的目标文件中。

> [!warning] 前提是不要提供 inline virtual function，这会在勘探式做法中导致产生大量的 vtbl 复制品。事实上，目前的编译器通常都会忽略 虚函数的 inline 指示，这其中有必要的理由。

b. virtual table pointer

vptr 的主要人物就是指明哪一个 vtbl。凡是声明有 虚函数的 class，其对象都含有一个隐藏的 data member 是 vptr。

存储成本 vptr 所带来的成本基本上不太大，但是在一个内存不是很充裕的系统中，这意味着可能所产生的对象，无法 放入同一个 cache page 或者 virtual memory page 之中，也就意味着你需要进行大量的换页操作。

运行成本 值得注意的是，虚函数的运行效率与非虚函数的效率是相当的。

执行成本 虚函数真正的运行时期成本发生在和 inline 互动的时候。inline 意味着在编译期内将被调用函数进行替 代操作。而 virtual 则意味着等待，直到运行时期才知道哪个函数被调用了。因此实际上，virtual 意味 着放弃 inline 的行为。

2. multiple inheritance 与 virtual base classes

对于 non-virtual base class，如果 derived class 在其 base class 上有多条复制路径(菱形 钻石问题)，则该 base class 会被复制多次。而使用 virtual base class 能够很好的解决这个问题， 可以消除这样的复制现象。

但是 virtual base class 会产生另一个成本，因为在实际实现过程中，继承关系使用指针实现，因此会 增加内存占用。

```
class A {};
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public c {};
```

```
        [ B data member ]
     ---『pointer to vbc』
    |   [ C data member ]
    |    『pointer to vbc』---
    |   [ D data member ]    |
     -->[ A data member ]<---
```

在此基础上再添加 vptr，内存占用会进一步增加。

```
        [ B data member ]
         『     vptr     』
     ---『pointer to vbc』
    |   [ C data member ]
    |    『     vptr     』
    |    『pointer to vbc』---
    |   [ D data member ]    |
    |    『     vptr     』    |
     -->[ A data member ]<---
```

3. Runtime Type Identification (RTTI)

RTTI 让我们可以在运行时获得 objects 和 classes 的相关信息。RTTI 的实现主要通过 typeid 来 获取某个 class 对应的 type_info 对象。每个内含虚函数的对象都需要能够获取到其专属信息。

RTTI 的设计是通过在 vtbl 中添加一个指针，放在索引 0 处。该指针指向对应的 class 的 type_info 对象。这也是 RTTI 所引入的成本。

主要成本表格

性质	对象大小增加	class 数据量增加	inlining 几率降低
虚函数	是	是	是
多重继承	是	是	否
虚拟基类	往往如此	有时候	否
RTTI	否	是	否