三路比较

Three-way Comparison

计14 王博文

双路比较

- 双路比较运算符：比较两个操作数，返回一个 `bool` 值
- `==`，`!=`，`<`，`>`，`<=`，`>=`

双路比较

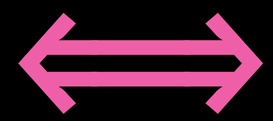
- 带来问题：
- 至少需要 2 次比较才能确定结果是小于、等于和大于中的哪一种，这在比较性能开销较大时（例如比较字符串、vector 等容器时）会带来性能浪费
- C++ 核心设计理念：Zero Overhead Abstraction 零成本抽象
- 因此有必要寻求方法规避性能浪费

双路比较

- 此外，另一个问题是在定义自己的类型时，需要重载全部 6 个运算符
- 然而其中大部分都是重复工作，反复实现同样的逻辑

第一个问题

- 为了解决第一个问题，C 字符串和 `std::string` 都提供了特化的三路比较
- C 字符串提供了 `strcmp` 函数来比较两个字符串，结果小于、等于和大于 `0` 分别对应字符串小于、等于和大于
- `std::string` 也提供了 `compare` 函数来进行类似的比较
- 但是，并没有统一的接口让类型实现类似比较

三路比较运算符 

三路比较运算符 ⟺

- C++20 引入的新运算符
- 又称宇宙飞船运算符（Spaceship Operator）
- 书写：<=>
- 优先级：低于 <<，>>，高于其他比较运算符

三路比较运算符 ⟺

- 三路比较的支持库是 `<compare>`
- 表达式 `a ⟺ b` 返回一个序对象，指示比较的结果
- 序对象有三种类型：
- 偏序关系：`std::partial_ordering`
- 弱序关系：`std::weak_ordering`
- 强序关系：`std::strong_ordering`

std::partial_ordering

- 偏序关系：存在不可比较的对象
- 区分“相等”和“等价”
- 可替换性：若 `f(a) = f(b)` 对任意函数 `f` 成立，称 `a` 和 `b` 可替换
- 等价（`equivalent`）：在某种意义下是等同的，但是不保证可替换性
- 相等（`equal`）：既等价又可替换

`std::partial_ordering`

- 例如，定义二维平面上两个点等价当且仅当它们到原点的欧氏距离相等
- 此时单位圆上的点都等价，但不相等，因为它们对“取 x 坐标”不可替换
- 复数集中存在不可比较对象，如虚数和其他数不可比，只有实数间才可比较

std::partial_ordering

- 在基础类型以及标准库类型中，只有浮点数参与比较时才会出现 `partial_ordering`
- 第一个原因：`NaN` 与任何值不可比较
- 第二个原因：存在等价但不相等的值，例如 `+0.0 == -0.0`，但它们对取符号函数（`<cmath>`）结果不同：`signbit(+0.0) ≠ signbit(-0.0)`

std::partial_ordering

- 可能取值：
- std::partial_ordering::less
- std::partial_ordering::equivalent
- std::partial_ordering::greater
- std::partial_ordering::unordered
- 取 unordered 时，除了 ≠ 比较为 true 外，其他比较都为 false

std::partial_ordering

```
1.  void print(partial_ordering order) {
2.      if (order == partial_ordering::less) {
3.          cout << "less" << endl;
4.      } else if (order == partial_ordering::equivalent) {
5.          cout << "equivalent" << endl;
6.      } else if (order == partial_ordering::greater) {
7.          cout << "greater" << endl;
8.      } else {
9.          cout << "unordered" << endl;
10.     }
11. }
```

std::partial_ordering

```
1.  double a, b;
2.  a = numbers::e, b = numbers::pi;
3.  print(a ⟺ b);
4.  a = numbers::pi, b = acos(-1);
5.  print(a ⟺ b);
6.  a = pow(numbers::e, numbers::pi), b = pow(numbers::pi, numbers::e);
7.  print(a ⟺ b);
8.  a = +0.0, b = -0.0;
9.  print(a ⟺ b);
10. cout << signbit(a) << ' ' << signbit(b) << endl;
11. a = 0.0 / 0.0, b = 42;
12. print(a ⟺ b);
13. // Output: less equivalent greater equivalent 0 1 unordered
```

`std::weak_ordering`

- 弱序关系：比偏序关系稍强，不存在不可比较的对象
- 但是仍区分相等和等价：等价不蕴涵可替换性
- 基础类型和标准库类型的比较不会返回此类型的结果

std::weak_ordering

- 可能取值：
- std::weak_ordering::less
- std::weak_ordering::equivalent
- std::weak_ordering::greater

`std::strong_ordering`

- 强序关系：比弱序关系更强，不存在不可比较的对象，相等蕴涵可替换性
- 大部分基础类型与标准库类型的三路比较返回此类型结果

std::strong_ordering

- 可能取值：
- std::strong_ordering::less
- std::strong_ordering::equivalent
- std::strong_ordering::equal
- std::strong_ordering::greater
- 其中 equivalent 和 equal 是等价的

获取比较结果

- 判断三路比较运算符的返回值和上面列出的可能取值是否相等
- 使用 `<compare>` 库提供的辅助函数 `is_eq`，`is_neq`，`is_lt`，`is_gt`，`is_lteq`，`is_gteq`，它们接受一个序对象，返回 `bool` 值
- 将序对象与 `0` 进行比较，小于、等于和大于 `0` 分别代表小于、等价（或相等）以及大于

获取比较结果

```
1.  auto ordering {value <=> 123};
2.  if (ordering == strong_ordering::less) {
3.      cout << "value < 123";
4.  } else if (is_gt(ordering)) {
5.      cout << "value > 123";
6.  } else if (ordering == 0) {
7.      cout << "value == 123";
8.  }
```

三路比较运算符的使用

- 一般情况下，如处理基础类型时，直接使用双路比较运算符即可
- 只有比较性能开销较大，而又需要确定比较结果是小于、等于和大于中具体是哪一种时，才需要先将比较结果保存下来再判断是哪一种结果

三路比较运算符的使用

```
1.  string a {"A long long string"}
2.  string b {"A long long integer”};
3.  // Only one comparison operation
4.  auto ordering {a ⟺ b};
5.  if (ordering < 0) {
6.      cout << "a < b”;
7.  } else if (ordering == 0) {
8.      cout << "a == b";
9.  } else {
10.     cout << "a > b";
11. }
```

第二个问题

- 自定义类型时，需要定义共 6 种比较运算符
- 甚至更多，例如需要和另一个类型比较时，还需考虑参数顺序可以交换，共 18 种
- 然而其中的比较逻辑都是一样的，大量重复工作
- <utility> 库中曾经提供了一个子命名空间 std::rel_ops，其中定义了 ≠，>，⩽，⩾ 四个运算符模板，只要 using 这个命名空间就可以获得这些运算符
- 缺点：依赖标准库；可能污染其他运算符，导致自动生成了不该生成的运算符

合成比较运算符

- 三路比较运算符的引入从语言层面上解决了第二个问题
- 在自定义类型中定义三路比较运算符后，编译器将自动生成 $<$，$>$，$\leqslant$，$\geqslant$ 运算符
- 如果还定义了 $=$ 运算符，编译器将生成 $\neq$ 运算符
- 因此，总共只需定义 $\iff$ 与 $=$ 运算符即可

合成比较运算符

- 例如，对于比较运算 `a @ b`，编译器会合成两种使用三路比较的表达：
- `(a ⟺ b) @ 0`
- `0 @ (b ⟺ a)`
- 这样只需为每种允许比较的类型定义一次 `⟺` 以及 `==` 即可

合成比较运算符

- 为何定义 $\iff$ 时不会合成 $=$ 与 $\neq$?
- 性能问题：判断是否相等需要的信息量少于三路比较，`"foobar"` $\neq$ `"foo"`
- 例如：三路比较两个字符串或 `vector`，必须逐位比较
- 但如果只需要知道是否相等，则可以先比较长度是否相等，如果不是直接结束
- 因此一般情况下，判断相等需要给出特化的实现，故不会自动生成

默认比较运算符

- 如果只需要按成员定义的顺序来逐个比较成员，可以将比较运算符定义为默认
- `auto operator <=> (const T&) const = default;`
- 此时编译器会生成所有六个运算符，包括 `==` 与 `!=`
- 返回成员中最弱的序关系，如有 `double` 成员结果就为 `partial_ordering`
- 若自定义了 `<=>` 的实现，但希望使用默认的 `==` 实现：
- `bool operator == (const T&) const = default;`

实例

```
1.  struct Name {
2.      string first, last;
3.      auto operator <=> (const Name &x) const {
4.          // We use tie here to avoid copy and utilize the predefined lexicographical comparison behavior
5.          return tie(last, first) <=> tie(x.last, x.first);
6.      }
7.      bool operator == (const Name &x) const {
8.          return first.size() == x.first.size() && last.size() == x.last.size()
9.              && tie(first, last) == tie(x.first, x.last);
10.     }
11. };
```

实例

```
1.  Name a {"Steve", "Jobs"};
2.  Name b {"Tim", "Cook"};
3.  cout << format("{} {} {} {} {} {}\n",
4.      a < b, a == b, a > b, a ≠ b, a ⩽ b, a ⩾ b);
5.  // Output: false false true true false true
```

总结

- C++20 引入的三路比较运算符能够：
- 以统一的接口提供对三路比较操作的访问
- 简化自定义类型对比较运算的定义

参考资料

- CPPReference: 比较运算符 https://zh.cppreference.com/w/cpp/language/operator_comparison
- CPPReference: 默认比较 https://zh.cppreference.com/w/cpp/language/default_comparisons
- CPPReference: 标准库头文件<compare> https://zh.cppreference.com/w/cpp/header/compare
- Cameron D. Simplify Your Code With Rocket Science: C++20's Spaceship Operator https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/
- The C++ Standards Committee, P0515R3, Consistent Comparison: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r3.pdf
- The C++ Standards Committee, P0905R1, Symmetry for Spaceship: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0905r1.html
- The C++ Standards Committee, P1185R2, ⟺ ≠ ═: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1185r2.html

End

```
return 0;
```