Item 27: Familiarize yourself with alternatives to overloading on universal references

Item27:熟悉通用引用重载的替代方法

Item 26中说明了对使用通用引用参数的函数，无论是独立函数还是成员函数（尤其是构造函数），进行重载都会导致一系列问题。但是也提供了一些示例，如果能够按照我们期望的方式运行，重载可能也是有用的。这个Item探讨了几种通过避免在通用引用上重载的设计或者通过限制通用引用可以匹配的参数类型的方式来实现所需行为的方案。

讨论基于Item 26中的示例，如果你还没有阅读Item 26，请先阅读在继续本Item的阅读。

Abandon overloading

在Item 26中的第一个例子中，`logAndAdd`代表了许多函数，这些函数可以使用不同的名字来避免在通用引用上的重载的弊端。例如两个重载的`logAndAdd`函数，可以分别改名为`logAndAddName`和`logAndAddNameIdx`。但是，这种方式不能用在第二个例子，Person构造函数中，因为构造函数的名字本类名固定了。此外谁愿意放弃重载呢?

Pass by const T&

一种替代方案是退回到C++98，然后将通用引用替换为const的左值引用。事实上，这是Item 26中首先考虑的方法。缺点是效率不高，会有拷贝的开销。现在我们知道了通用引用和重载的组合会导致问题，所以放弃一些效率来确保行为正确简单可能也是一种不错的折中。

Pass by value

通常在不增加复杂性的情况下提高性能的一种方法是，将按引用传递参数替换为按值传递，这是违反直觉的。该设计遵循Item 41中给出的建议，即在你知道要拷贝时就按值传递，因此会参考Item 41来详细讨论如何设计与工作，效率如何。这里，在Person的例子中展示:

```
class Person {
public:
  explicit Person(std::string p) // replace T&& ctor; see
  : name(std::move(n)) {} // Item 41 for use of std::move

  explicit Person(int idx)
  : name(nameFromIdx(idx)) {}
  ...
private:
  std::string name;
};
```

因为没有`std::string`构造器可以接受整型参数，所有`int`或者其他整型变量（比如`std::size_t`、`short`、`long`等）都会使用`int`类型重载的构造函数。相似的，所有`std::string`类似的参数（字面量等）都会使用`std::string`类型的重载构造函数。没有意外情况。我想你可能会说有些人想要使用0或者NULL会调用`int`重载的构造函数，但是这些人应该参考Item 8反复阅读指导使用0或者NULL作为空指针让他们恶心。

Use Tag dispatch

传递`const`左值引用参数以及按值传递都不支持完美转发。如果使用通用引用的动机是完美转发，我们就只能使用通用引用了，没有其他选择。但是又不想放弃重载。所以如果不放弃重载又不放弃通用引用，如何避免咋通用引用上重载呢?

实际上并不难。通过查看重载的所有参数以及调用的传入参数，然后选择最优匹配的函数----计算所有参数和变量的组合。通用引用通常提供了最优匹配，但是如果通用引用是包含其他非通用引用参数列表的一部分，则不是通用引用的部分会影响整体。这基本就是tag dispatch 方法，下面的示例会使这段话更容易理解。

我们将tag dispatch应用于`logAndAdd`例子，下面是原来的代码，以免你找不到Item 26的代码位置:

```
std::multiset<std::string> names; // global data structure
template<typename T> // make log entry and add
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_clokc::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

就其本身而言，功能执行没有问题，但是如果引入一个`int`类型的重载，就会重新陷入Item 26中描述的麻烦。这个Item的目标是避免它。不通过重载，我们重新实现`logAndAdd`函数分拆为两个函数，一个针对整型值，一个针对其他。`logAndAdd`本身接受所有的类型。

这两个真正执行逻辑的函数命名为`logAndAddImpl`使用重载。一个函数接受通用引用参数。所以我们同时使用了重载和通用引用。但是每个函数接受第二个参数，表征传入的参数是否为整型。这第二个参数可以帮助我们避免陷入到Item 26中提到的麻烦中，因为我们将其安排为第二个参数决定选择哪个重载函数。

是的，我知道，“不要在啰嗦了，赶紧亮出代码”。没有问题，代码如下，这是最接近正确版本的:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_integral<T>()); // not quite correct
}
```

这个函数转发它的参数给`logAndAddImpl`函数，但是多传递了一个表示是否T为整型的变量。至少，这就是应该做的。对于右值的整型参数来说，这也是正确的。但是如同Item 28中说明，如果左值参数传递给通用引用`name`，类型推断会使左值引用。所以如果左值int被传入`logAndAdd`，T将被推断为`int&`。这不是一个整型类型，因为引用不是整型类型。这意味着`std::is_integral<T>`对于左值参数返回false，即使确实传入了整型值。

意识到这个问题基本相当于解决了它，因为C++标准库有一个类型trait（参见Item 9），`std::remove_reference`，函数名字就说明做了我们希望的：移除引用。所以正确实现的代码应该是这样:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_instegral<typename std::remove_reference<T>::type>());
}
```

这个代码很巧妙。（在C++14中，你可以通过`std::remove_reference_t<T>`来简化写法，参看Item 9）

处理完之后，我们可以将注意力转移到名为`logAndAddImpl`的函数上了。有两个重载函数，第一个仅用于非整型类型（即`std::is_instegral<typename std::remove_reference<T>::type>()`是`false`）：

```
template<typename T>
void logAndAddImpl(T&& name, std::false_type) // 高亮为std::false_type
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

一旦你理解了高亮参数的含义代码就很直观。概念上，`logAndAdd`传递一个布尔值给`logAndAddImpl`表明是否传入了一个整型类型，但是`true`和`false`是运行时值，我们需要使用编译时决策来选择正确的`logAndAddImpl`重载。这意味着我们需要一个类型对应`true`，`false`同理。这个需要是经常出现的，所以标准库提供了这样两个命名`std::true_type and std::false_type`。`logAndAdd`传递给`logAndAddImpl`的参数类型取决于T是否整型，如果T是整型，它的类型就继承自`std::true_type`，反之继承自`std::false_type`。最终的结果就是，当T不是整型类型时，这个`logAndAddImpl`重载会被调用。

第二个重载覆盖了相反的场景：当T是整型类型。在这个场景中，`logAndAddImpl`简单找到下标处的`name`，然后传递给`logAndAdd`：

```
std::string nameFromIdx(int idx); // as in item 26
void logAndAddImpl(int idx, std::true_type) // 高亮: std::true_type
{
  logAndAdd(nameFromIdx(idx));
}
```

通过下标找到对应的`name`，然后让`logAndAddImpl`传递给`logAndAdd`，我们避免了将日志代码放入这个`logAndAddImpl`重载中。

在这个设计中，类型`std::true_type`和`std::false_type`是“标签”，其唯一目的就是强制重载解析按照我们的想法来执行。注意到我们甚至没有对这些参数进行命名。他们在运行时毫无用处，事实上我们希望编译器可以意识到这些tag参数是无用的然后在程序执行时优化掉它们（至少某些时候有些编译器会这样做）。这种在`logAndAdd`内部的通过tag来实现重载实现函数的“分发”，因此这个设计名称为：**tag dispatch**。这是模板元编程的标准构建模块，你对现代C++库中的代码了解越多，你就会越多遇到这种设计。

就我们的目的而言，tag dispatch的重要之处在于它可以允许我们组合重载和通用引用使用，而没有Item 26中提到的问题。分发函数---`logAndAdd`----接受一个没有约束的通用引用参数，但是这个函数没有重载。实现函数---`logAndAddImpl`----是重载的，一个接受通用引用参数，但是重载规则不仅依赖通用引用参数，还依赖新引入的tag参数。结果是tag来决定采用哪个重载函数。通用引用参数可以生成精

确匹配的事实在这里并不重要。（译者注：这里确实比较啰嗦，如果理解了上面的内容，这段完全可以没有。）

Constraining templates that take universal references（约束使用通用引用的模板）

tag dispatch的关键是存在单独一个函数（没有重载）给客户端API。这个单独的函数分发给具体的实现函数。创建一个没有重载的分发函数通常是容易的，但是Item 26中所述第二个问题案例是`Person`类的完美转发构造函数，是个例外。编译器可能会自行生成拷贝和移动构造函数，所以即使你只写了一个构造函数并在其中使用tag dispatch，编译器生成的构造函数也打破了你的期望。

实际上，真正的问题不是编译器生成的函数会绕过tag diapatch设计，而是不总会绕过tag dispatch。你希望类的拷贝构造总是处理该类型的`non-const`左值构造请求，但是如同Item 26中所述，提供具有通用引用的构造函数会使通用引用构造函数被调用而不是拷贝构造函数。还说明了当一个基类声明了完美转发构造函数，派生类实现自己的拷贝和移动构造函数时会发生错误的调用（调用基类的完美转发构造函数而不是基类的拷贝或者移动构造）

这种情况，采用通用引用的重载函数通常比期望的更加贪心，但是有不满足使用tag dispatch的条件。你需要不同的技术，可以让你确定允许使用通用引用模板的条件。朋友你需要的就是`std::enable_if`。

`std::enable_if`可以给你提供一种强制编译器执行行为的方法，即使特定模板不存在。这种模板也会被禁止。默认情况下，所有模板是启用的，但是使用`std::enable_if`可以使得仅在条件满足时模板才启用。在这个例子中，我们只在传递的参数类型不是`Person`使用`Person`的完美转发构造函数。如果传递的参数是`Person`，我们要禁止完美转发构造函数（即让编译器忽略它），因此就是拷贝或者移动构造函数处理，这就是我们想要使用`Person`初始化另一个`Person`的初衷。

这个主意听起来并不难，但是语法比较繁杂，尤其是之前没有接触过的话，让我慢慢引导你。有一些使用`std::enbale_if`的样板，让我们从这里开始。下面的代码是`Person`完美转发构造函数的声明，我仅展示声明，因为实现部分跟Item 26中没有区别。

```
class Person {
public:
  template<typename T,
                typename = typename std::enable_if<condition>::type> // 本行高亮
  explicit Person(T&& n);
  ...
};
```

为了理解高亮部分发生了什么，我很遗憾的表示你要自行查询语法含义，因为详细解释需要花费一定空间和时间，而本书并没有足够的空间（在你自行学习过程中，请研究"SFINAE"以及`std::enable_if`，因为“SFINAE”就是使`std::enable_if`起作用的技术）。这里我想要集中讨论条件的表示，该条件表示此构造函数是否启用。

这里我们想表示的条件是确认T不是`Person`类型，即模板构造函数应该在T不是`Person`类型的时候启用。因为type trait可以确定两个对象类型是否相同（`std::is_same`），看起来我们需要的就是`!std::is_same<Person, T>::value`（注意语句开始的！，我们想要的是不相同）。这很接近我们想要的了，但是不完全正确，因为如同Item 28中所述，对于通用引用的类型推导，如果是左值的话会推导成左值引用，比如这个代码：

```
Person p("Nancy");
auto cloneOfP(p); // initialize from lvalue
```

T的类型在通用引用的构造函数中被推导为`Person&`。`Person`和`Person&`类型是不同的，`std::is_same`对比`std::is_same<Person, Person&>::value`会是`false`。

如果我们更精细考虑仅当T不是`Person`类型才启用模板构造函数，我们会意识到当我们查看T时，应该忽略：

- **是否引用**。对于决定是否通用引用构造器启用的目的来说，`Person, Person&, Person&&`都是跟`Person`一样的。
- **是不是`const`或者`volatile`**。如上所述，`const Person , volatile Person , const volatile Person`也是跟`Person`一样的。

这意味着我们需要一种方法消除对于`T`的`引用, const, volatile`修饰。再次，标准库提供了这样的功能type trait，就是`std::decay`。`std::decay<T>::value`与`T`是相同的，只不过会移除`引用, const, volatile`的修饰。（这里我没有说出另外的真相，`std::decay`如同其名一样，可以将array或者function退化成指针，参考Item 1，但是在这里讨论的问题中，它刚好合适）。我们想要控制构造器是否启用的条件可以写成：

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

表示`Person`与`T`的类型不同。

将其带回整体代码中，`Person`的完美转发构造函数的声明如下：

```
class Person {
public:
  template<typename T,
                typename = typename std::enable_if<
             !std::is_same<Person, typename std::decay<T>::type>::value
             >::type> // 本行高亮
  explicit Person(T&& n);
  ...
};
```

如果你之前从没有看到过这种类型的代码，那你可太幸福了。最后是这种设计是有原因的。当你使用其他机制来避免同时使用重载和通用引用时（你总会这样做），确实应该那样做。不过，一旦你习惯了使用函数语法和尖括号的使用，也不坏。此外，这可以提供你一直想要的行为表现。在上面的声明中，使用`Person`初始化一个`Person`----无论是左值还是右值，`const`还是`volatile`都不会调用到通用引用构造函数。

成功了，对吗？确实！

当然没有。等会再庆祝。Item 26还有一个情景需要解决，我们需要继续探讨下去。

假定从`Person`派生的类以常规方式实现拷贝和移动操作：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs): Person(rhs)
  {...} // copy ctor; calls base class forwarding ctor!
  SpecialPerson(SpecialPerson&& rhs): Person(std::move(rhs))
  {...} // move ctor; calls base class forwarding ctor!
};
```

这和Item 26中的代码是一样的，包括注释也是一样。当我们拷贝或者移动一个`SpecialPerson`对象时，我们希望调用基类对应的拷贝和移动构造函数，但是这里，我们将`SpecialPerson`传递给基类的构造器，因为`SpecialPerson`和`Person`类型不同，所以完美转发构造函数是启用的，会实例化为精确匹配的构造函数。生成的精确匹配的构造函数之于重载规则比基类的拷贝或者移动构造函数更优，所以这里的代码，拷贝或者移动`SpecialPerson`对象就会调用`Person`类的完美转发构造函数来执行基类的部分。跟Item 26的困境一样。

派生类仅仅是按照常规的规则生成了自己的移动和拷贝构造函数，所以这个问题的解决还要落实在在基类，尤其是控制是否使用`Person`通用引用构造函数启用的条件。现在我们意识到不只是禁止`Person`类型启用模板构造器，而是禁止`Person`以及任何派生自`Person`的类型启用模板构造器。讨厌的继承！

你应该不意外在这里看到标准库中也有type trait判断一个类型是否继承自另一个类型，就是`std::is_base_of`。如果`std::is_base_of<T1, T2>`是`true`表示`T2`派生自`T1`。类型系统是自派生的，表示`std::is_base_of<T, T>::value`总是`true`。这就很方便了，我们想要修正关于我们控制`Person`完美转发构造器的启用条件，只有当`T`在消除引用，`const`，`volatile`修饰之后，并且既不是`Person`又不是`Person`的派生类，才满足条件。所以使用`std::is_base_of`代替`std::is_same`就可以了：

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_base_if<Person,
                                                                  typename
std::decay<T>::type
                                  >::value
                >::type
  >
  explicit Person(T&& n);
  ...
};
```

现在我们终于完成了最终版本。这是C++11版本的代码，如果我们使用C++14，这份代码也可以工作，但是有更简洁一些的写法如下：

```
class Person  { // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<   // less code here
                        !std::is_base_of<Person,
                                 std::decay_t<T> // and here
                                >::value
                > // and here
  >
  explicit Person(T&& n);
  ...
};
```

好了，我承认，我又撒谎了。我们还没有完成，但是越发接近最终版本了。非常接近，我保证。

我们已经知道如何使用`std::enable_if`来选择性禁止`Person`通用引用构造器来使得一些参数确保使用到拷贝或者移动构造器，但是我们还是不知道将其应用于区分整型参数和非整型参数。毕竟，我们的原始目标是解决构造函数模糊性问题。

我们需要的工具都介绍过了，我保证都介绍了，
（1）加入一个`Person`构造函数重载来处理整型参数
（2）约束模板构造器使其对于某些参数禁止
使用这些我们讨论过的技术组合起来，就能解决这个问题了：

```
class Person { // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<
     !std::is_base_of<Person, std::decay_t<T>::value
       &&
       !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n): name(std::forward<T>(n))
  {...} // ctor for std::strings and args convertible to strings

  explicit Person(int idx): name(nameFromIdx(idx))
  {...} // ctor for integral args

  ... // copy and move ctors, etc
private:
  std::string name;
};
```

看！多么优美！好吧，优美之处只是对于那些迷信模板元编程之人，但是事实却是提出了不仅能工作的方法，而且极具技巧。因为使用了完美转发，所以具有最大效率，因为控制了使用通用引用的范围，可以避免对于大多数参数能实例化精确匹配的滥用问题。

Trade-offs （权衡，折中）

本Item提到的前三个技术---abandoning overloading, passing by const T&, passing by value---在函数调用中指定每个参数的类型。后两个技术----tag dispatch和 constraing template eligibility----使用完美转发，因此不需要指定参数类型。这一基本决定（是否指定类型）有一定后果。

通常，完美转发更有效率，因为它避免了仅处于符合参数类型而创建临时对象。在`Person`构造函数的例子中，完美转发允许将`Nancy`这种字符串字面量转发到容器内部的`std::string`构造器，不使用完美转发的技术则会创建一个临时对象来满足传入的参数类型。

但是完美转发也有缺点。·即使某些类型的参数可以传递给特定类型的参数的函数，也无法完美转发。Item 30中探索了这方面的例子。

第二个问题是当client传递无效参数时错误消息的可理解性。例如假如创建一个`Person`对象的client传递了一个由`char16_t`（一种C++11引入的类型表示16位字符）而不是`char`（`std::string`包含的）：

```
Person p(u"Konrad Zuse"); // "Konrad Zuse" consists of characters of type const
char16_t
```

使用本Item中讨论的前三种方法，编译器将看到可用的采用`int`或者`std::string`的构造函数，并且它们或多或少会产生错误消息，表示没有可以从`const char16_t`转换为`int`或者`std::string`的方法。

但是，基于完美转发的方法，`const char16_t`不受约束地绑定到构造函数的参数。从那里将转发到`Person`的`std::string`的构造函数，在这里，调用者传入的内容(`const char16_t`数组)与所需内容(`std::string`构造器可接受的类型)发生的不匹配会被发现。由此产生的错误消息会让人更容易理解，在我使用的编译器上，会产生超过160行错误信息。

在这个例子中，通用引用仅被转发一次（从`Person`构造器到`std::string`构造器），但是更复杂的系统中，在最终通用引用到达最终判断是否可接受的函数之前会有多层函数调用。通用引用被转发的次数越多，产生的错误消息偏差就越大。许多开发者发现仅此问题就是在性能优先的接口使用通用引用的障碍。（译者注：最后一句话可能翻译有误，待确认）

在`Person`这个例子中，我们知道转发函数的通用引用参数要支持`std::string`的初始化，所以我们可以用`static_assert`来确认是不是支持。`std::is_constructible` type trait执行编译时测试一个类型的对象是否可以构造另一个不同类型的对象，所以代码可以这样：

```
class Person {
public:
  template<typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
        !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n) :name(std::forward<T>(n))
  {
    //assert that a std::string can be created from a T object(这里到...高亮)
    static_assert(
        std::is_constructible<std::string, T>::value,
      "Parameter n can't be used to construct a std::string"
    );
    ... // the usual ctor work goes here
  }
  ... // remainder of Person class (as before)
};
```

如果client代码尝试使用无法构造`std::string`的类型创建`Person`，会导致指定的错误消息。不幸的是，在这个例子中，`static_assert`在构造函数体中，但是作为成员初始化列表的部分在检查之前。所以我使用的编译器，结果是由`static_assert`产生的清晰的错误消息在常规错误消息（最多160行以上那个）后出现。

需要记住的事

- 通用引用和重载的组合替代方案包括使用不同的函数名，通过const左值引用传参，按值传递参数，使用tag dispatch
- 通过`std::enable_if`约束模板，允许组合通用引用和重载使用，`std::enable_if`可以控制编译器哪种条件才使用通用引用的实例
- 通用引用参数通常具有高效率的优势，但是可用性就值得斟酌