一、Concepts的定义

下面是concept的定义的形式。

```
template < template-parameter-list >
concept  concept-name = constraint-expression;
```

其中， `constraint-expression` 是一个可以被eval为bool的表达式或者编译期函数。 在使用定义好的concept时， `constraint-expression` 会根据上面 `template-parameter-list` 传入的类型，执行编译期计算，判断使用该concept的模板定义是否满足。 如果不满足，则编译期会给定一个具有明确语义的错误，即 这个concept没有匹配成功啦啦这种。 注意到，上述匹配的行为都是在编译期完成的，因此concept其实是zero-cost的。 举个例子来描述一下，最基本的concept的定义。

```
// 一个永远都能匹配成功的concept
template <typename T>
concept always_satisfied = true;

// 一个约束T只能是整数类型的concept，整数类型包括 char, unsigned char, short, ushort, int,
template <typename T>
concept integral = std::is_integral_v<T>;

// 一个约束T只能是整数类型，并且是有符号的concept
template <typename T>
concept signed_integral = integral<T> && std::is_signed_v<T>;
```

接下来，我们再简单示例一下如何使用一个concept

```
// 任意类型都能匹配成功的约束，因此mul只要支持乘法运算符的类型都可以匹配成功。
template <always_satisfied T>
T mul(T a, T b) {
    return a * b;
}

// 整型才能匹配add函数的T
template <integral T>
T add(T a, T b) {
    return a + b;
}

// 有符号整型才能匹配subtract函数的T
template <signed_integral T>
T subtract(T a, T b) {
    return a - b;
}

int main() {
    mul(1, 2); // 匹配成功, T => int
    mul(1.0f, 2.0f);  // 匹配成功，T => float
```

```
    add(1, -2);  // 匹配成功, T => int
    add(1.0f, 2.0f); // 匹配失败, T => float, 而T必须是整型
    subtract(1U, 2U); // 匹配失败, T => unsigned int,而T必须是有符号整型
    subtract(1, 2); // 匹配成功, T => int
}
```

二、 Concept的本质的基本理解

Concept其实是一个语法糖，它的本质可以认为是一个模板类型的bool变量。定义一个concept本质上是在定义一个bool类型的编译期的变量。使用一个concept本质上是利用SFINAE机制来约束模板类型。

约束模板类型，在之前的C++也可以做。但是，有了concept之后，做类型约束后，代码不仅清晰了很多，而且脑细胞也节省了不少。

为了加深对concept的本质的理解，不如试试，上面 `add` 函数，用传统的C++11应该怎么写。如下面所示：

```
template <typename T, std::enable_if_t<std::is_integral_v<T>, T> = 0 >
T add_original(T a, T b) {
    return a + b;
}
```

对于C++大佬，上面这段拗口的模板代码非常简单，但是我第一次学习的时候，觉得很绕。这个是如何匹配只能是整数的呢？ 这个其实是利用模板特化和SFINAE机制来做的。下面是 `std::enable_if_t` 在VS中的STL的实现。

```
// STRUCT TEMPLATE enable_if
template <bool _Test, class _Ty = void>
struct enable_if {}; // no member "type" when !_Test

template <class _Ty>
struct enable_if<true, _Ty> { // type is _Ty for _Test
    using type = _Ty;
};

template <bool _Test, class _Ty = void>
using enable_if_t = typename enable_if<_Test, _Ty>::type;
```

虽然旧的实现方法代码量更少，但是从语言设计来说，这样写我认为实在是丑陋而且难懂。我觉得主要缺点有以下：

1. `add_original` 函数的模板中引入了一个多余的模板参数，而这个模板参数其实是没有作用的。这点我觉得比较丑陋。

2.模板类型T的约束的逻辑，写的跟模板声明一样，这样导致逻辑耦合了。

三、 Concept的花式使用方法

据说C++委员会为了满足各种不同喜好的使用的提案，直接支持了好多种Concept的使用方法。对此，我觉得真的脑壳疼，C++真的越走越脱离群众了。

具体有以下几种，简单总结就是 有3种方式，另外再加上与auto关键字的一些结合方式。

```
// 约束函数模板方法1
template <my_concept T>
void f(T v);

// 约束函数模板方法2
template <typename T>
requires my_concept<T>
void f(T v);

// 约束函数模板方法3
template <typename T>
void f(T v) requires my_concept<T>;

// 直接约束C++14的auto的函数参数
void f(my_concept auto v);

// 约束模板的auto参数
template <my_concept auto v>
void g();

// 约束auto变量
my_concept auto foo = ...;
```

Concept当然也可以用在lambda函数上，使用方法跟上面一样，也有同样数量的花式用法

```
// 约束lambda函数的方法1
auto f = []<my_concept T> (T v) {
  // ...
};
// 约束lambda函数的方法2
auto f = []<typename T> requires my_concept<T> (T v) {
  // ...
};
// 约束lambda函数的方法3
auto f = []<typename T> (T v) requires my_concept<T> {
  // ...
};
// auto函数参数约束
auto f = [](my_concept auto v) {
  // ...
};
// auto模板参数约束
auto g = []<my_concept auto v> () {
  // ...
};
```

四、concept的组合(与或非)

concept的本质是一个模板的编译期的bool变量，因此它可以使用C++的与或非三个操作符。当然，理解上也就跟我们常见的bool变量一样啦。 例如，我们可以在定义concept的时候，使用其他concept或者表达式，进行逻辑操作。

```
template <typename T>
concept Integral = std::is_integral<T>::value;
template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
template <typename T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

当然，我们也可以在使用concept的时候使用 逻辑操作符。

```
template <typename T>
requires Integral<T> && std::is_signed_v<T>
T add(T a, T b);
```

五、 requires关键字的其他用法

requires关键字不仅能用在concept的使用上，也可以用在定义中。 例如

```
// requires用在使用concept时
template <typename T>
  requires my_concept<T>
void f(T);

// requires用在concept的定义，它表达了类型T的参数f，必须符合大括号内的模式，也就是能被调用。
// 也就是它是一个函数或者一个重载了operator()的类型
template <typename T>
concept callable = requires (T f) { f(); };

template <typename T>
  requires requires (T x) { x + x; } // `requires` 同时使用在concept的定义和使用上
T add(T a, T b) {
  return a + b;
}
```

requires的语法看起来很复杂，初看会觉得很乱，没有啥规律或者总结性的东西。我这边个人理解大概是这样：

```
// 这种也就是，requires后面接的是一个正在被eval的concept，这也就是用在上面的concept的使用中。
requires evaled-concept

// 本质上，concept在evaluate时，也就会是一个编译期返回结果为bool的表达式。这种其实等价于上面那
requires expression

// 例如 下面这种就是requires后直接接个bool表达式了
```

```
template <typename T>
requires std::is_integral_v<T>
T add(T a, T b) {
    return a + b;
}
```

我这边认为，requires后接的东西本质上是一个表达式。当然，

六、 使用requires关键字进行约束嵌套或组合

为了提高concept定义的能力，requires支持用大括号的语法，进行多个约束分开表达，这些约束之间的关系是 与 的关系。

requires的这种方式的语法形式是

```
requires { requirement-seq }
requires ( parameter-list(optional) ) { requirement-seq }
```

这里每个 `requirement-seq` 是可以由多行约束组成，每一行之间以分号分隔。 这些约束的形式有以下几种

简单约束(Simple Requirements)
类型约束(Type Requirements)
复合约束(Compound Requirements)
嵌套约束(Nested Requirements)

6.1 简单约束

简单约束就是一个任意的表达式，编译器对这个约束的检查就是检查这个表达式是否是合法的。注意，不是说这个表达式在编译期运行返回true或者false。而是这个表达式是否合法。 例如

```
template<typename T>
concept bool Addable =
requires (T a, T b) {
    a + b; // "the expression a+b is a valid expression that will compile"
};

// example constraint from the standard library (ranges TS)
template <class T, class U = T>
concept bool Swappable = requires(T&& t, U&& u) {
    swap(std::forward<T>(t), std::forward<U>(u));
    swap(std::forward<U>(u), std::forward<T>(t));
};
```

6.2 类型约束

类型的约束是类似模板里面的参数一样，在 typename 后接一个类型。这个约束表达的含义是该类型在该concept进行evaluate时，必须是存在的。 如下面的例子：

```
struct foo {
    int foo;
};

struct bar {
    using value = int;
    value data;
};

struct baz {
    using value = int;
    value data;
};

// Using SFINAE, enable if `T` is a `baz`.
template <typename T, typename = std::enable_if_t<std::is_same_v<T, baz>>>
struct S {};

template <typename T>
using Ref = T&;

template <typename T>
concept C = requires {
    // Requirements on type `T`:
    typename T::value;  // A) has an inner member named `value`
    typename S<T>;      // B) must have a valid class template specialization for `S`
    typename Ref<T>;    // C) must be a valid alias template substitution
};

template <C T>
void g(T a);

g(foo{}); // ERROR: Fails requirement A.
g(bar{}); // ERROR: Fails requirement B.
g(baz{}); // PASS.
```

我估摸着这种复杂的类型约束，可以用于做一些高层模式的约束，例如迭代器什么的。

6.3 复合约束

复合约束用于约束表达式的返回值的类型。它的写法形式为：

```
// 这里 ->和type-constraint是可选的.
{expression} noexcept(optional) -> type-constraint;
```

这里的约束的行为主要有三点,并且约束进行evaluate的顺序按照以下顺序

模板类型代换到表达式中是否使得表达式合法
如果用了noexcept,表达式必须不能可能抛出异常.
如果用了->后的类型约束, 则按照以下步骤进行evaluate

代换模板类型到 type-constraint中,
并且 `decltype((expression))` 的类型必须满足type-constraint的约束.

上述步骤任何一个失败,则evaluate的结果是false.

```
template <typename T>
concept C = requires(T x) {
  {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::i
  {x + 1} -> std::same_as<int>; // the expression `x + 1` satisfies `std::same_as<declt
  {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
};
```

6.4 嵌套约束

requires内部还可以嵌套requires. 这种方式被称为嵌套的约束.它的形式为

```
requires constraint-expression ;
```

例如

```
template <class T>
concept Semiregular = DefaultConstructible<T> &&
    CopyConstructible<T> && Destructible<T> && CopyAssignable<T> &&
requires(T a, size_t n) {
    requires Same<T*, decltype(&a)>;  // nested: "Same<...> evaluates to true"
    { a.~T() } noexcept;  // compound: "a.~T()" is a valid expression that doesn't thro
    requires Same<T*, decltype(new T)>; // nested: "Same<...> evaluates to true"
    requires Same<T*, decltype(new T[n])>; // nested
    { delete new T };  // compound
    { delete new T[n] }; // compound
};
```

七、 concept的约束的深层次理解以及暗坑

7.1 原子约束

前面给出了concept定义的格式:

```
template < template-parameter-list >
concept  concept-name = constraint-expression;
```

其中concept中最核心的就是约束的表达式。编译器是怎么理解constraint-expression的呢，编译器认为constraint-expression由三种类型的约束组成

Conjunctions(与)
Disjunctions(或)
Atomic Constraints(原子约束)

与或是非常好理解的，原子约束就有点难以理解了。 在编译器看来，原子约束由一个表达式E，以及E的参数映射(parameter mappings)。E的参数映射的意思是 约束的实体的跟表达式E相关的模板参数和模板类型。

原子约束是在编译器进行 `constraint normalization` 时生成的。表达式E中必然不包含 AND 或者OR的逻辑，否则它就会被分割为2个原子约束。

那么，哪些东西是原子约束呢，我这边认为是不带有AND或者OR的约束，我理解中比如下面的例子

```
template<class T> constexpr bool is_a = true;
template<class T> constexpr bool is_b = true;

template<class T>
concept concept_a_or_b = is_a<T> || is_b<T>;
```

concept_a_or_b含有2个原子约束，然后通过disjunctions组合而成。

7.2 constraint normalization

约束单元化指的是将一个复杂的constraint转换成原子约束以及他们的与和或。这个转换过程按照以下逻辑来执行：

表达式 `(E)` 就是E本身 - 表达式 `E1 && E2` 是由E1和E2组成的与
表达式 `E1 || E2` 是由E1和E2组成的或
表达式 `C<A1, A2, ..., AN>` ,其中C是一个concept，`A1, A2...` 是C的模板参数的传入参数。在这里，如果传入的这些模板参数，在代换时是不合法的类型或者表达式，则表达式就是不合法的。

例如下面这个例子, B是合法的，因为A的T模板参数，可以传入U*这种指针类型，虽然U*没有value对象，但是模板参数T可以是指针类型的。C是不合法的，因为它代换为B的时候，再由B代换为A时，模板参数T是 `V&\*` ，C++不存在引用的指针类型。所以模板参数T这次类型代换是不合法。

```
template<typename T> concept A = T::value || true;
template<typename U>
concept B = A<U*>; // OK: normalized to the disjunction of
                   // - T::value (with mapping T -> U*) and
                   // - true (with an empty mapping).
                   // No invalid type in mapping even though
                   // T::value is ill-formed for all pointer types

template<typename V>
concept C = B<V&>; // Normalizes to the disjunction of
                   // - T::value (with mapping T-> V&*) and
                   // - true (with an empty mapping).
                   // Invalid type V&* formed in mapping => ill-formed NDR
```

任何其他的类型的表达式，都是原子的约束。包括 Flod Expression, 即使它内部包含了 && 或者 || 操作符。

用户重载的&&和||操作符对 constraint normalization 无效。

7.3 坑1：原子约束的相等判断与normalization的关系

两个原子约束相等 仅当 他们是在源码层面相同的表达式，并且他们的parameter mappings相同。例如下面这个例子：

```
template<class T> constexpr bool is_meowable = true;
template<class T> constexpr bool is_cat = true;

template<class T>
concept Meowable = is_meowable<T>;

template<class T>
concept BadMeowableCat = is_meowable<T> && is_cat<T>;

template<class T>
concept GoodMeowableCat = Meowable<T> && is_cat<T>;

template<Meowable T>
void f1(T); // #1

template<BadMeowableCat T>
void f1(T); // #2

template<Meowable T>
void f2(T); // #3

template<GoodMeowableCat T>
void f2(T); // #4

void g(){
    f1(0); // error, ambiguous:
           // the is_meowable<T> in Meowable and BadMeowableCat forms distinct
           // atomic constraints that are not identical (and so do not subsume each oth

    f2(0); // OK, calls #4, more constrained than #3
           // GoodMeowableCat got its is_meowable<T> from Meowable
}
```

上面f1之所以二义性，是因为 在 Meowable和BadMeowableCat这两个concept在normalzation时，他们的原子约束 `is_mewable<T>` 在源码层面不是同一个表达式。 也就是来自源码文件中不同的行。。。

7.4 坑2：constraint normalization的返回类型必须是bool，不能进行类型转换

例如下面这个例子， `S<T>{}` 对象是可以转换为bool类型的struct对象。但是 `(S<T>{})` 这个约束的表达式还是非法的。因为它被代换后，它的类型不是bool，即使它能转换为bool。

```
template<typename T>
struct S {
    constexpr operator bool() const { return true; }
};

template<typename T>
    requires (S<T>{})
void f(T); // #1

void f(int); // #2

void g() {
    f(0); // error: S<int>{} does not have type bool when checking #1,
          // even though #2 is a better match
}
```

把它改成下面这样就可以

```
template<typename T>
struct S {
    constexpr operator bool() const { return true; }
};

template<typename T>
    requires (bool(S<T>{}))
void f(T); // #1
```

或者这样

```
template<typename T>
struct S {
    constexpr operator bool() const { return true; }
};

template<typename T>
    requires (S<T>{}())
void f(T); // #1
```