D.6 ratio头文件

<ratio> 头文件提供在编译时进行的计算。

头文件内容

```
namespace std
{
  template<intmax_t N,intmax_t D=1>
  class ratio;

  // ratio arithmetic
  template <class R1, class R2>
  using ratio_add = see description;

  template <class R1, class R2>
  using ratio_subtract = see description;

  template <class R1, class R2>
  using ratio_multiply = see description;

  template <class R1, class R2>
  using ratio_divide = see description;

  // ratio comparison
  template <class R1, class R2>
  struct ratio_equal;

  template <class R1, class R2>
  struct ratio_not_equal;

  template <class R1, class R2>
  struct ratio_less;

  template <class R1, class R2>
  struct ratio_less_equal;

  template <class R1, class R2>
  struct ratio_greater;
```

```
34
35    template <class R1, class R2>
36    struct ratio_greater_equal;
37
38    typedef ratio<1, 1000000000000000000> atto;
39    typedef ratio<1, 1000000000000000> femto;
40    typedef ratio<1, 1000000000000> pico;
41    typedef ratio<1, 1000000000> nano;
42    typedef ratio<1, 1000000> micro;
43    typedef ratio<1, 1000> milli;
44    typedef ratio<1, 100> centi;
45    typedef ratio<1, 10> deci;
46    typedef ratio<10, 1> deca;
47    typedef ratio<100, 1> hecto;
48    typedef ratio<1000, 1> kilo;
49    typedef ratio<1000000, 1> mega;
50    typedef ratio<1000000000, 1> giga;
51    typedef ratio<1000000000000, 1> tera;
52    typedef ratio<1000000000000000, 1> peta;
53    typedef ratio<1000000000000000000, 1> exa;
54  }
```

D.6.1 std::ratio类型模板

`std::ratio` 类型模板提供了一种对在编译时进行计算的机制，通过调用合理的数，例如：半(`std::ratio<1,2>`),2/3(std::ratio)或15/43(std::ratio)。其使用在C++标准库内部，用于初始化 `std::chrono::duration` 类型模板。

类型定义

```
1  template <intmax_t N, intmax_t D = 1>
2  class ratio
3  {
4  public:
5    typedef ratio<num, den> type;
6    static constexpr intmax_t num= see below;
7    static constexpr intmax_t den= see below;
8  };
```

要求

D不能为0。

描述

num和den分别为分子和分母，构造分数N/D。den总是正数。当N和D的符号相同，那么num为正数；否则num为负数。

例子

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

D.6.2 std::ratio_add模板别名

`std::ratio_add` 模板别名提供了两个 `std::ratio` 在编译时相加的机制(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_add<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>` 相同。

例子

```
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::num == 11
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::den == 15

```

```
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::num == 3
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::den == 2
```

D.6.3 std::ratio_subtract模板别名

`std::ratio_subtract` 模板别名提供两个 `std::ratio` 数在编译时进行相减(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

先决条件
R1和R2必须使用 `std::ratio` 进行初始化。

效果
ratio_add被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_subtract<R1, R2>` 应该应该与
`std::ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>` 相同。

例子

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::num == 2
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::den == 15

std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::num == -5
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::den == 6
```

D.6.4 std::ratio_multiply模板别名

`std::ratio_multiply` 模板别名提供两个 `std::ratio` 数在编译时进行相乘(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况下， `std::ratio_multiply<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::num, R1::den * R2::den>` 相同。

例子

```
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::num == 2
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::den == 15

std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::num == 5
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::den == 7
```

D.6.5 std::ratio_divide模板别名

`std::ratio_divide` 模板别名提供两个 `std::ratio` 数在编译时进行相除(使用有理计算)。

定义

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

效果

ratio_add被定义为一个别名，如果两数可以计算，且无溢出，该类型可以表示两个 `std::ratio` 对象R1和R2的和。如果计算出来的结果溢出了，那么程序里面就有问题了。在算术溢出的情况

下， `std::ratio_multiply<R1, R2>` 应该应该与 `std::ratio<R1::num * R2::num * R2::den, R1::den * R2::den>` 相同。

例子

```
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::num == 5
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::den == 6

std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::num == 7
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::den == 45
```

D.6.6 std::ratio_equal类型模板

`std::ratio_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_equal:
  public std::integral_constant<
    bool,(R1::num == R2::num) && (R1::den == R2::den)>
{};
```

先决条件
R1和R2必须使用 `std::ratio` 进行初始化。

例子

```
std::ratio_equal<std::ratio<1,3>, std::ratio<2,6> >::value == true
std::ratio_equal<std::ratio<1,3>, std::ratio<1,6> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<2,3> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<1,3> >::value == true
```

D.6.7 std::ratio_not_equal类型模板

`std::ratio_not_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_not_equal:
  public std::integral_constant<bool,!ratio_equal<R1,R2>::value>
{};
```

先决条件
R1和R2必须使用 `std::ratio` 进行初始化。

例子

```
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,6> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,3> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,3> >::value == false
```

D.6.8 std::ratio_less类型模板

`std::ratio_less` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_less:
  public std::integral_constant<bool,see below>
{};
```

先决条件
R1和R2必须使用 `std::ratio` 进行初始化。

效果
std::ratio_less可通过 `std::integral_constant<bool, value >` 导出，这里value为 `(R1::num*R2::den) < (R2::num*R1::den)` 。如果有可能，需要实现使用一种机制来避免计算结果已出。当溢出发生，那么程序中就肯定有错误。

例子

```
std::ratio_less<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_less<std::ratio<1,6>, std::ratio<1,3> >::value == true
std::ratio_less<
  std::ratio<999999999,1000000000>,
  std::ratio<1000000001,1000000000> >::value == true
std::ratio_less<
  std::ratio<1000000001,1000000000>,
  std::ratio<999999999,1000000000> >::value == false
```

D.6.9 std::ratio_greater类型模板

`std::ratio_greater` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_greater:
  public std::integral_constant<bool,ratio_less<R2,R1>::value>
{};
```

先决条件
R1和R2必须使用 `std::ratio` 进行初始化。

D.6.10 std::ratio_less_equal类型模板

`std::ratio_less_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_less_equal:
  public std::integral_constant<bool,!ratio_less<R2,R1>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。

D.6.11 std::ratio_greater_equal类型模板

`std::ratio_greater_equal` 类型模板提供在编译时比较两个 `std::ratio` 数(使用有理计算)。

类型定义

```
template <class R1, class R2>
class ratio_greater_equal:
  public std::integral_constant<bool,!ratio_less<R1,R2>::value>
{};
```

先决条件

R1和R2必须使用 `std::ratio` 进行初始化。