C++ 11的新特性

5. 多线程

5.1 处理日期和时间的 chrono 库

C++11 中提供了日期和时间相关的库 chrono，通过 chrono 库可以很方便地处理日期和时间，为程序的开发提供了便利。chrono 库主要包含三种类型的类：时间间隔 **duration**、时钟 **clocks**、时间点 **time point**。

5.1.1 时间间隔 duration

duration表示一段时间间隔，用来记录时间长度，可以表示几秒、几分钟、几个小时的时间间隔。

duration 的原型如下：

```
// 定义于头文件 <chrono>
template<
    class Rep, // int float
    class Period = std::ratio<1>
> class duration;
```

- **Rep**：这是一个**数值类型**，表示时钟数（周期）的类型（默认为整形）。若 Rep 是浮点数，则 duration 能使用小数描述时钟周期的数目。
- **Period**：表示时钟的周期，它的原型如下：

```
// 定义于头文件 <ratio>
template<
    std::intmax_t Num,
    std::intmax_t Denom = 1
> class ratio;
```

ratio 类表示**每个时钟周期的秒数**，其中第一个模板参数 **Num 代表分子**，**Denom 代表分母**，该分母值默认为 1，因此，**ratio 代表的是一个分子除以分母的数值**，比如：

ratio<2> 代表一个时钟周期是 2 秒，

ratio<60 > 代表一分钟，

ratio<6060 > *代表一个小时，*

*ratio<60*60*24 > 代表一天。

ratio<1,1000 > 代表的是 1/1000 秒，也就是 1 毫秒，**分子为 1 分母为 1000**

ratio<1,1000000 > 代表一微秒，ratio<1,1000000000 > 代表一纳秒。

> 为了方便使用，在标准库中定义了一些常用的时间间隔，比如：时、分、秒、毫秒、微秒、纳秒，它们都位于 chrono 命名空间下，定义如下：

类型	定义
纳秒：**std::chrono::nanoseconds**	duration<Rep/ 至少 64 位的有符号整数类型 /, std::nano>
微秒：**std::chrono::microseconds**	duration<Rep/ 至少 55 位的有符号整数类型 /, std::micro>
毫秒：**std::chrono::milliseconds**	duration<Rep/ 至少 45 位的有符号整数类型 /, std::milli>
秒：**std::chrono::seconds**	duration<Rep/ 至少 35 位的有符号整数类型 />
分钟：**std::chrono::minutes**	duration<Rep/ 至少 29 位的有符号整数类型 /, std::ratio<60>>
小时：**std::chrono::hours**	duration<Rep/ 至少 23 位的有符号整数类型 /, std::ratio<3600>>

注意：到 hours 为止的每个预定义时长类型至少涵盖 ±292 年的范围。

> duration 类的构造函数原型如下：

```
// 1. 拷贝构造函数
duration( const duration& ) = default;
// 2. 通过指定时钟周期的类型来构造对象
template< class Rep2 >
constexpr explicit duration( const Rep2& r );
// 3. 通过指定时钟周期类型，和时钟周期长度来构造对象
template< class Rep2, class Period2 >
constexpr duration( const duration<Rep2,Period2>&
d );
```

> 为了更加方便的进行 duration 对象之间的操作，类内部进行了操作符重载：

操作符重载	描述
operator=	赋值内容 (公开成员函数)
operator+ operator-	实现一元 + 和一元 - (公开成员函数)
operator++ operator++(int) operator– operator–(int)	递增或递减周期计数 (公开成员函数)
operator+= operator-= operator*= operator/= operator%=	实现二个时长间的复合赋值 (公开成员函数)

> duration 类还提供了**获取时间间隔的时钟周期数**的方法 count()，函数原型如下：

```
constexpr rep count() const;
```

5.1.2 duration 类的使用

> 通过构造函数构造事件间隔对象示例代码如下：

```
#include <chrono>
#include <iostream>
using namespace std;
int main()
{
    chrono::hours h(1);
 // 一小时
    chrono::milliseconds ms{ 3 };
 // 3 毫秒
    chrono::duration<int, ratio<1000>> ks(3);
 // 3000 秒

    // chrono::duration<int, ratio<1000>>
d3(3.5);  // error
    chrono::duration<double> dd(6.6);
  // 6.6 秒

    // 使用小数表示时钟周期的次数
    chrono::duration<double, std::ratio<1, 30>>
hz(3.5);
}
```

chrono 库中根据 duration 类封装了不同长度的时钟周期（也可以自定义），基于这个时钟周期再进行周期次数的设置就可以得到总的时间间隔了（**时钟周期 * 周期次数 = 总的时间间隔**）。

6. thread 线程的使用

C++11 中提供的线程类叫做 **std::thread**，基于这个类创建一个新的线程非常的简单，只需要提供线程函数或者函数对象即可，并且可以同时指定线程函数的参数。

6.1 构造函数

```
// ①
thread() noexcept;
// ②
thread( thread&& other ) noexcept;
// ③
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
// ④
thread( const thread& ) = delete;
```

- **构造函数①**：默认构造函，构造一个线程对象，在这个线程中不执行任何处理动作
- **构造函数②**：移动构造函数，将 `other` 的线程所有权转移给新的 `thread` 对象。之后 `other` 不再表示执行线程。
- **构造函数③**：创建线程对象，并在该线程中执行函数 `f` 中的业务逻辑，`args` 是要传递给函数 `f` 的参数
  - 普通函数，类成员函数，匿名函数，仿函数（这些都是可调用对象类型）
  - 可以是可调用对象包装器类型，也可以是使用绑定器绑定之后得到的类型（仿函数）
- **构造函数④**：使用 =delete 显示删除拷贝构造，不允许线程对象之间的拷贝

6.2 get_id()

get_id网页连接

函数在哪个线程中被执行，那么函数就阻塞哪个线程。该函数的函数原型如下：

```
void join();
```

如果要阻塞主线程的执行，只需要在主线程中通过子线程对象调用这个方法即可，当调用这个方法的子线程对象中的任务函数执行完毕之后，主线程的阻塞也就随之解除了。

```
int main()
{
    cout << "主线程的线程ID: " << this_thread::get_id() << endl;
    thread t(func, 520, "i love you");
    thread t1(func1);
    cout << "线程t 的线程ID: " << t.get_id() << endl;
    cout << "线程t1的线程ID: " << t1.get_id() << endl;
    t.join();
    t1.join();
}
```

6.3 detach()

detach()网页链接

detach() 函数的作用是进行线程分离，分离主线程和创建出的子线程。在线程分离之后，主线程退出也会一并销毁创建出的所有子线程，在主线程退出之前，它可以脱离主线程继续独立的运行，任务执行完毕之后，这个子线程会自动释放自己占用的系统资源。该函数函数原型如下：

```
void detach();
```

> 线程分离函数没有参数也没有返回值，只需要在线程成功之后，通过线程对象调用该函数即可

```
int main()
{
    cout << "主线程的线程ID: " << this_thread::get_id() << endl;
    thread t(func, 520, "i love you");
    thread t1(func1);
    cout << "线程t 的线程ID: " << t.get_id() << endl;
    cout << "线程t1的线程ID: " << t1.get_id() << endl;
    t.detach();
    t1.detach();
    // 让主线程休眠, 等待子线程执行完毕
    this_thread::sleep_for(chrono::seconds(5));
}
```

6.4 joinable()

joinable()网页链接

```
bool joinable() const noexcept;
```

- 返回值为 true：主线程和子线程之间有关联（连接）关系

- 返回值为 false：主线程和子线程之间没有关联（连接）关系