D.4 future头文件

<future> 头文件提供处理异步结果(在其他线程上执行额结果)的工具。

头文件内容

```
namespace std
{
  enum class future_status {
      ready, timeout, deferred };

  enum class future_errc
  {
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
  };

  class future_error;

  const error_category& future_category();

  error_code make_error_code(future_errc e);
  error_condition make_error_condition(future_errc e);

  template<typename ResultType>
  class future;

  template<typename ResultType>
  class shared_future;

  template<typename ResultType>
  class promise;

  template<typename FunctionSignature>
  class packaged_task; // no definition provided

  template<typename ResultType,typename ... Args>
```

```
  class packaged_task<ResultType (Args...)>;

  enum class launch {
    async, deferred
  };

  template<typename FunctionType,typename ... Args>
  future<result_of<FunctionType(Args...)>::type>
  async(FunctionType&& func,Args&& ... args);

  template<typename FunctionType,typename ... Args>
  future<result_of<FunctionType(Args...)>::type>
  async(std::launch policy,FunctionType&& func,Args&& ... args);
}
```

D.4.1 std::future类型模板

`std::future` 类型模板是为了等待其他线程上的异步结果。其和 `std::promise` ，`std::packaged_task` 类型模板，还有 `std::async` 函数模板，都是为异步结果准备的工具。只有 `std::future` 实例可以在任意时间引用异步结果。

`std::future` 实例是MoveConstructible(移动构造)和MoveAssignable(移动赋值)，不过不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型声明

```
template<typename ResultType>
class future
{
public:
  future() noexcept;
  future(future&&) noexcept;
  future& operator=(future&&) noexcept;
  ~future();

  future(future const&) = delete;
  future& operator=(future const&) = delete;

  shared_future<ResultType> share();

```

```
    bool valid() const noexcept;

    see description get();

    void wait();

    template<typename Rep,typename Period>
    future_status wait_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    future_status wait_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

std::future 默认构造函数

不使用异步结果构造一个 `std::future` 对象。

声明

```
future() noexcept;
```

效果
构造一个新的 `std::future` 实例。

后置条件
valid()返回false。

抛出
无

std::future 移动构造函数

使用另外一个对象，构造一个 `std::future` 对象，将相关异步结果的所有权转移给新 `std::future` 对象。

声明

```
future(future&& other) noexcept;
```

效果
使用已有对象构造一个新的 `std::future` 对象。

后置条件
已有对象中的异步结果，将于新的对象相关联。然后，解除已有对象和异步之间的关系。 `this->valid()` 返回的结果与之前已有对象 `other.valid()` 返回的结果相同。在调用该构造函数后， `other.valid()` 将返回false。

抛出
无

std::future 移动赋值操作

将已有 `std::future` 对象中异步结果的所有权，转移到另一对象当中。

声明

```
future(future&& other) noexcept;
```

效果
在两个 `std::future` 实例中转移异步结果的状态。

后置条件
当执行完赋值操作后， `*this.other` 就与异步结果没有关系了。异步状态(如果有的话)在释放后与 `*this` 相关，并且在最后一次引用后，销毁该状态。 `this->valid()` 返回的结果与之前已有对象 `other.valid()` 返回的结果相同。在调用该构造函数后， `other.valid()` 将返回false。

抛出
无

std::future 析构函数

销毁一个 `std::future` 对象。

声明

```
~future();
```

效果
销毁 `*this` 。如果这是最后一次引用与 `*this` 相关的异步结果，之后就会将该异步结果销毁。

抛出
无

std::future::share 成员函数

构造一个新 `std::shared_future` 实例，并且将 `*this` 异步结果的所有权转移到新的 `std::shared_future` 实例中。

声明

```
shared_future<ResultType> share();
```

效果
如同 shared_future(std::move(*this))。

后置条件
当调用share()成员函数，与 `*this` 相关的异步结果将与新构造的 `std::shared_future` 实例相关。 `this->valid()` 将返回false。

抛出
无

std::future::valid 成员函数

检查 `std::future` 实例是否与一个异步结果相关联。

声明

```
bool valid() const noexcept;
```

返回
当与异步结果相关时，返回true，否则返回false。

抛出

无

std::future::wait 成员函数

如果与 *this 相关的状态包含延迟函数，将调用该函数。否则，会等待 std::future 实例中的异步结果准备就绪。

声明

```
void wait();
```

先决条件

this->valid() 将会返回true。

效果

当相关状态包含延迟函数，调用延迟函数，并保存返回的结果，或将抛出的异常保存成为异步结果。否则，会阻塞到 *this 准备就绪。

抛出

无

std::future::wait_for 成员函数

等待 std::future 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

先决条件

this->valid() 将会返回true。

效果

如果与 *this 相关的异步结果包含一个 std::async 调用的延迟函数(还未执行)，那么就不阻塞

立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或超过给定的relative_time时长。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ；当给定时间超过relative_time时，返回 `std::future_status::timeout` 。

NOTE:线程阻塞的时间可能超多给定的时长。时长尽可能由一个稳定的时钟决定。

抛出

无

std::future::wait_until 成员函数

等待 `std::future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

```
template<typename Clock,typename Duration>
future_status wait_until(
  std::chrono::time_point<Clock,Duration> const& absolute_time);
```

先决条件

this->valid()将返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或 `Clock::now()` 返回的时间大于等于absolute_time。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ； `Clock::now()` 返回的时间大于等于absolute_time，返回 `std::future_status::timeout` 。

NOTE:这里不保证调用线程会被阻塞多久，只有函数返回 `std::future_status::timeout` ，然后 `Clock::now()` 返回的时间大于等于absolute_time的时候，线程才会解除阻塞。

抛出
无

std::future::get 成员函数

当相关状态包含一个 `std::async` 调用的延迟函数，调用该延迟函数，并返回结果；否则，等待与 `std::future` 实例相关的异步结果准备就绪，之后返回存储的值或异常。

声明

```
void future<void>::get();
R& future<R&>::get();
R future<R>::get();
```

先决条件
this->valid()将返回true。

效果
如果*this相关状态包含一个延期函数，那么调用这个函数并返回结果，或将抛出的异常进行传播。

否则，线程就要被阻塞，直到与*this相关的异步结果就绪。当结果存储了一个异常，那么就就会将存储异常抛出。否则，将会返回存储值。

返回
当相关状态包含一个延期函数，那么这个延期函数的结果将被返回。否则，当ResultType为void时，就会按照常规调用返回。如果ResultType是R&(R类型的引用)，存储的引用值将会被返回。否则，存储的值将会返回。

抛出
异常由延期函数，或存储在异步结果中的异常(如果有的话)抛出。

后置条件

```
this->valid()==false
```

D.4.2 std::shared_future类型模板

`std::shared_future` 类型模板是为了等待其他线程上的异步结果。其和 `std::promise` ，`std::packaged_task` 类型模板，还有 `std::async` 函数模板，都是为异步结果准备的工具。多个 `std::shared_future` 实例可以引用同一个异步结果。

`std::shared_future` 实例是CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。你也可以同ResultType的 `std::future` 类型对象，移动构造一个 `std::shared_future` 类型对象。

访问给定 `std::shared_future` 实例是非同步的。因此，当有多个线程访问同一个 `std::shared_future` 实例，且无任何外围同步操作时，这样的访问是不安全的。不过访问关联状态时是同步的，所以多个线程访问多个独立的 `std::shared_future` 实例，且没有外围同步操作的时候，是安全的。

类型定义

```
template<typename ResultType>
class shared_future
{
public:
  shared_future() noexcept;
  shared_future(future<ResultType>&&) noexcept;

  shared_future(shared_future&&) noexcept;
  shared_future(shared_future const&);
  shared_future& operator=(shared_future const&);
  shared_future& operator=(shared_future&&) noexcept;
  ~shared_future();

  bool valid() const noexcept;

  see description get() const;

  void wait() const;

  template<typename Rep,typename Period>
  future_status wait_for(
     std::chrono::duration<Rep,Period> const& relative_time) const;

  template<typename Clock,typename Duration>
  future_status wait_until(
     std::chrono::time_point<Clock,Duration> const& absolute_time)
```

```
27      const;
28  };
```

std::shared_future 默认构造函数

不使用关联异步结果，构造一个 `std::shared_future` 对象。

声明

```
shared_future() noexcept;
```

效果
构造一个新的 `std::shared_future` 实例。

后置条件
当新实例构建完成后，调用valid()将返回false。

抛出
无

std::shared_future 移动构造函数

以一个已创建 `std::shared_future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果的所有权转移到新的实例中。

声明

```
shared_future(shared_future&& other) noexcept;
```

效果
构造一个新 `std::shared_future` 实例。

后置条件
将other对象中关联异步结果的所有权转移到新对象中，这样other对象就没有与之相关联的异步结果了。

抛出
无

std::shared_future 移动对应std::future对象的构造函数

以一个已创建 `std::future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果的所有权转移到新的实例中。

声明

```
shared_future(std::future<ResultType>&& other) noexcept;
```

效果
构造一个 `std::shared_future` 对象。

后置条件
将other对象中关联异步结果的所有权转移到新对象中，这样other对象就没有与之相关联的异步结果了。

抛出
无

std::shared_future 拷贝构造函数

以一个已创建 `std::future` 对象为准，构造 `std::shared_future` 实例，并将使用 `std::shared_future` 对象关联的异步结果(如果有的话)拷贝到新创建对象当中，两个对象共享该异步结果。

声明

```
shared_future(shared_future const& other);
```

效果
构造一个 `std::shared_future` 对象。

后置条件
将other对象中关联异步结果拷贝到新对象中，与other共享关联的异步结果。

抛出
无

std::shared_future 析构函数

销毁一个 std::shared_future 对象。

声明

```
~shared_future();
```

效果
将 *this 销毁。如果 *this 关联的异步结果与 std::promise 或 std::packaged_task 不再有关联，那么该函数将会切断 std::shared_future 实例与异步结果的联系，并销毁异步结果。

抛出
无

std::shared_future::valid 成员函数

检查 std::shared_future 实例是否与一个异步结果相关联。

声明

```
bool valid() const noexcept;
```

返回
当与异步结果相关时，返回true，否则返回false。

抛出
无

std::shared_future::wait 成员函数

当*this关联状态包含一个延期函数，那么调用这个函数。否则，等待直到与 std::shared_future 实例相关的异步结果就绪为止。

声明

```
void wait() const;
```

先决条件 this->valid()将返回true。

效果

当有多个线程调用 `std::shared_future` 实例上的get()和wait()时，实例会序列化的共享同一关联状态。如果关联状态包括一个延期函数，那么第一个调用get()或wait()时就会调用延期函数，并且存储返回值，或将抛出异常以异步结果的方式保存下来。

抛出

无

std::shared_future::wait_for 成员函数

等待 `std::shared_future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time) const;
```

先决条件

`this->valid()` 将会返回true。

效果

如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延期函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或超过给定的relative_time时长。

返回

当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ； 当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ； 当给定时间超过relative_time时，返回 `std::future_status::timeout` 。

NOTE:线程阻塞的时间可能超多给定的时长。时长尽可能由一个稳定的时钟决定。

抛出

无

std::shared_future::wait_until 成员函数

等待 `std::future` 实例上相关异步结果准备就绪，或超过某个给定的时间。

声明

```
template<typename Clock,typename Duration>
future_status wait_until(
  std::chrono::time_point<Clock,Duration> const& absolute_time) const;
```

先决条件
this->valid()将返回true。

效果
如果与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，那么就不阻塞立即返回。否则将阻塞实例，直到与 `*this` 相关异步结果准备就绪，或 `Clock::now()` 返回的时间大于等于absolute_time。

返回
当与 `*this` 相关的异步结果包含一个 `std::async` 调用的延迟函数(还未执行)，返回 `std::future_status::deferred` ；当与 `*this` 相关的异步结果准备就绪，返回 `std::future_status::ready` ； `Clock::now()` 返回的时间大于等于absolute_time，返回 `std::future_status::timeout` 。

NOTE:这里不保证调用线程会被阻塞多久，只有函数返回 `std::future_status::timeout` ，然后 `Clock::now()` 返回的时间大于等于absolute_time的时候，线程才会解除阻塞。

抛出
无

std::shared_future::get 成员函数

当相关状态包含一个 `std::async` 调用的延迟函数，调用该延迟函数，并返回结果；否则，等待与 `std::shared_future` 实例相关的异步结果准备就绪，之后返回存储的值或异常。

声明

```
void shared_future<void>::get() const;
R& shared_future<R&>::get() const;
```

```
3  R const& shared_future<R>::get() const;
```

先决条件

this->valid()将返回true。

效果

当有多个线程调用 `std::shared_future` 实例上的get()和wait()时，实例会序列化的共享同一关联状态。如果关联状态包括一个延期函数，那么第一个调用get()或wait()时就会调用延期函数，并且存储返回值，或将抛出异常以异步结果的方式保存下来。

阻塞会知道*this关联的异步结果就绪后解除。当异步结果存储了一个一行，那么就会抛出这个异常。否则，返回存储的值。

返回

当ResultType为void时，就会按照常规调用返回。如果ResultType是R&(R类型的引用)，存储的引用值将会被返回。否则，返回存储值的const引用。

抛出

抛出存储的异常(如果有的话)。

D.4.3 std::packaged_task类型模板

`std::packaged_task` 类型模板可打包一个函数或其他可调用对象，所以当函数通过 `std::packaged_task` 实例被调用时，结果将会作为异步结果。这个结果可以通过检索 `std::future` 实例来查找。

`std::packaged_task` 实例是可以MoveConstructible(移动构造)和MoveAssignable(移动赋值)，不过不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型定义

```
1  template<typename FunctionType>
2  class packaged_task; // undefined
3
4  template<typename ResultType,typename... ArgTypes>
5  class packaged_task<ResultType(ArgTypes...)>
6  {
7  public:
```

```
  packaged_task() noexcept;
  packaged_task(packaged_task&&) noexcept;
  ~packaged_task();

  packaged_task& operator=(packaged_task&&) noexcept;

  packaged_task(packaged_task const&) = delete;
  packaged_task& operator=(packaged_task const&) = delete;

  void swap(packaged_task&) noexcept;

  template<typename Callable>
  explicit packaged_task(Callable&& func);

  template<typename Callable,typename Allocator>
  packaged_task(std::allocator_arg_t, const Allocator&,Callable&&);

  bool valid() const noexcept;
  std::future<ResultType> get_future();
  void operator()(ArgTypes...);
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

std::packaged_task 默认构造函数

构造一个 `std::packaged_task` 对象。

声明

```
packaged_task() noexcept;
```

效果
不使用关联任务或异步结果来构造一个 `std::packaged_task` 对象。

抛出
无

std::packaged_task 通过可调用对象构造

使用关联任务和异步结果，构造一个 `std::packaged_task` 对象。

声明

```
template<typename Callable>
packaged_task(Callable&& func);
```

先决条件

表达式 `func(args...)` 必须是合法的，并且在 `args...` 中的args-i参数，必须是 `ArgTypes...` 中ArgTypes-i类型的一个值。且返回值必须可转换为ResultType。

效果

使用ResultType类型的关联异步结果，构造一个 `std::packaged_task` 对象，异步结果是未就绪的，并且Callable类型相关的任务是对func的一个拷贝。

抛出

当构造函数无法为异步结果分配出内存时，会抛出 `std::bad_alloc` 类型的异常。其他异常会在使用Callable类型的拷贝或移动构造过程中抛出。

std::packaged_task 通过有分配器的可调用对象构造

使用关联任务和异步结果，构造一个 `std::packaged_task` 对象。使用以提供的分配器为关联任务和异步结果分配内存。

声明

```
template<typename Allocator,typename Callable>
packaged_task(
    std::allocator_arg_t, Allocator const& alloc,Callable&& func);
```

先决条件

表达式 `func(args...)` 必须是合法的，并且在 `args...` 中的args-i参数，必须是 `ArgTypes...` 中ArgTypes-i类型的一个值。且返回值必须可转换为ResultType。

效果

使用ResultType类型的关联异步结果，构造一个 `std::packaged_task` 对象，异步结果是未就绪的，并且Callable类型相关的任务是对func的一个拷贝。异步结果和任务的内存通过内存分配器alloc进行分配，或进行拷贝。

抛出

当构造函数无法为异步结果分配出内存时，会抛出 `std::bad_alloc` 类型的异常。其他异常会在使用Callable类型的拷贝或移动构造过程中抛出。

std::packaged_task 移动构造函数

通过一个 `std::packaged_task` 对象构建另一个，将与已存在的 `std::packaged_task` 相关的异步结果和任务的所有权转移到新构建的对象当中。

声明

```
packaged_task(packaged_task&& other) noexcept;
```

效果

构建一个新的 `std::packaged_task` 实例。

后置条件

通过other构建新的 `std::packaged_task` 对象。在新对象构建完成后，other与其之前相关联的异步结果就没有任何关系了。

抛出

无

std::packaged_task 移动赋值操作

将一个 `std::packaged_task` 对象相关的异步结果的所有权转移到另外一个。

声明

```
packaged_task& operator=(packaged_task&& other) noexcept;
```

效果

将other相关异步结果和任务的所有权转移到 `*this` 中，并且切断异步结果和任务与other对象的关联，如同 `std::packaged_task(other).swap(*this)` 。

后置条件

与other相关的异步结果与任务移动转移，使*this.other无关联的异步结果。

返回

```
*this
```

抛出
无

std::packaged_task::swap 成员函数

将两个 std::packaged_task 对象所关联的异步结果的所有权进行交换。

声明

```
void swap(packaged_task& other) noexcept;
```

效果
将other和*this关联的异步结果与任务进行交换。

后置条件
将与other关联的异步结果和任务，通过调用swap的方式，与*this相交换。

抛出
无

std::packaged_task 析构函数

销毁一个 std::packaged_task 对象。

声明

```
~packaged_task();
```

效果
将 *this 销毁。如果 *this 有关联的异步结果，并且结果不是一个已存储的任务或异常，那么异步结果状态将会变为就绪，伴随就绪的是一个 std::future_error 异常和错误码 std::future_errc::broken_promise 。

抛出
无

std::packaged_task::get_future 成员函数

在*this相关异步结果中，检索一个 `std::future` 实例。

声明

```
std::future<ResultType> get_future();
```

先决条件
*this具有关联异步结果。

返回
一个与*this关联异构结果相关的一个 `std::future` 实例。

抛出
如果一个 `std::future` 已经通过get_future()获取了异步结果，在抛出 `std::future_error` 异常时，错误码是 `std::future_errc::future_already_retrieved`

std::packaged_task::reset 成员函数

将一个 `std::packaged_task` 对实例与一个新的异步结果相关联。

声明

```
void reset();
```

先决条件
*this具有关联的异步任务。

效果
如同 `*this=packaged_task(std::move(f))` ，f是*this中已存储的关联任务。

抛出
如果内存不足以分配给新的异构结果，那么将会抛出 `std::bad_alloc` 类异常。

std::packaged_task::valid 成员函数

检查*this中是都具有关联任务和异步结果。

声明

```
bool valid() const noexcept;
```

返回
当*this具有相关任务和异步结构，返回true；否则，返回false。

抛出
无

std::packaged_task::operator() 函数调用操作

调用一个 `std::packaged_task` 实例中的相关任务，并且存储返回值，或将异常存储到异常结果当中。

声明

```
void operator()(ArgTypes... args);
```

先决条件
*this具有相关任务。

效果
像 `INVOKE(func,args...)` 那要调用相关的函数func。如果返回征程，那么将会存储到*this相关的异步结果中。当返回结果是一个异常，将这个异常存储到*this相关的异步结果中。

后置条件
*this相关联的异步结果状态为就绪，并且存储了一个值或异常。所有阻塞线程，在等待到异步结果的时候被解除阻塞。

抛出
当异步结果已经存储了一个值或异常，那么将抛出一个 `std::future_error` 异常，错误码为 `std::future_errc::promise_already_satisfied` 。

同步

`std::future<ResultType>::get()` 或 `std::shared_future<ResultType>::get()` 的成功调用，代表同步操作的成功，函数将会检索异步结果中的值或异常。

std::packaged_task::make_ready_at_thread_exit 成员函数

调用一个 `std::packaged_task` 实例中的相关任务，并且存储返回值，或将异常存储到异常结果当中，直到线程退出时，将相关异步结果的状态置为就绪。

声明

```
void make_ready_at_thread_exit(ArgTypes... args);
```

先决条件

*this具有相关任务。

效果

像 `INVOKE(func,args...)` 那要调用相关的函数func。如果返回征程，那么将会存储到 `*this` 相关的异步结果中。当返回结果是一个异常，将这个异常存储到 `*this` 相关的异步结果中。当当前线程退出的时候，可调配相关异步状态为就绪。

后置条件

*this的异步结果中已经存储了一个值或一个异常，不过在当前线程退出的时候，这个结果都是非就绪的。当当前线程退出时，阻塞等待异步结果的线程将会被解除阻塞。

抛出

当异步结果已经存储了一个值或异常，那么将抛出一个 `std::future_error` 异常，错误码为 `std::future_errc::promise_already_satisfied` 。当无关联异步状态时，抛出 `std::future_error` 异常，错误码为 `std::future_errc::no_state` 。

同步

`std::future<ResultType>::get()` 或 `std::shared_future<ResultType>::get()` 在线程上的成功调用，代表同步操作的成功，函数将会检索异步结果中的值或异常。

D.4.4 std::promise类型模板

`std::promise` 类型模板提供设置异步结果的方法，这样其他线程就可以通过 `std::future` 实例来索引该结果。

ResultType模板参数，该类型可以存储异步结果。

`std::promise` 实例中的异步结果与某个 `srd::future` 实例相关联，并且可以通过调用get_future()成员函数来获取这个 `srd::future` 实例。ResultType类型的异步结果，可以通过set_value()成员函数对存储值进行设置，或者使用set_exception()将对应异常设置进异步结果中。

`std::promise` 实例是可以MoveConstructible(移动构造)和MoveAssignable(移动赋值)，但是不能CopyConstructible(拷贝构造)和CopyAssignable(拷贝赋值)。

类型定义

```
template<typename ResultType>
class promise
{
public:
  promise();
  promise(promise&&) noexcept;
  ~promise();
  promise& operator=(promise&&) noexcept;

  template<typename Allocator>
  promise(std::allocator_arg_t, Allocator const&);

  promise(promise const&) = delete;
  promise& operator=(promise const&) = delete;

  void swap(promise& ) noexcept;

  std::future<ResultType> get_future();

  void set_value(see description);
  void set_exception(std::exception_ptr p);
};
```

std::promise 默认构造函数

构造一个 `std::promise` 对象。

声明

```
promise();
```

效果

使用ResultType类型的相关异步结果来构造 `std::promise` 实例，不过异步结果并未就绪。

抛出

当没有足够内存为异步结果进行分配，那么将抛出 `std::bad_alloc` 型异常。

std::promise 带分配器的构造函数

构造一个 `std::promise` 对象，使用提供的分配器来为相关异步结果分配内存。

声明

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

效果

使用ResultType类型的相关异步结果来构造 `std::promise` 实例，不过异步结果并未就绪。异步结果的内存由alloc分配器来分配。

抛出

当分配器为异步结果分配内存时，如有抛出异常，就为该函数抛出的异常。

std::promise 移动构造函数

通过另一个已存在对象，构造一个 `std::promise` 对象。将已存在对象中的相关异步结果的所有权转移到新创建的 `std::promise` 对象当中。

声明

```
promise(promise&& other) noexcept;
```

效果

构造一个 `std::promise` 实例。

后置条件
当使用other来构造一个新的实例，那么other中相关异构结果的所有权将转移到新创建的对象上。之后，other将无关联异步结果。

抛出
无

std::promise 移动赋值操作符

在两个 `std::promise` 实例中转移异步结果的所有权。

声明

```
promise& operator=(promise&& other) noexcept;
```

效果
在other和 `*this` 之间进行异步结果所有权的转移。当 `*this` 已经有关联的异步结果，那么该异步结果的状态将会为就绪态，且伴随一个 `std::future_error` 类型异常，错误码为 `std::future_errc::broken_promise` 。

后置条件
将other中关联的异步结果转移到*this当中。other中将无关联异步结果。

返回

```
*this
```

抛出
无

std::promise::swap 成员函数

将两个 `std::promise` 实例中的关联异步结果进行交换。

声明

```
void swap(promise& other);
```

效果
交换other和*this当中的关联异步结果。

后置条件
当swap使用other时，other中的异步结果就会与*this中关联异步结果相交换。二者返回来亦然。

抛出
无

std::promise 析构函数

销毁 `std::promise` 对象。

声明

```
~promise();
```

效果
销毁 `*this` 。当 `*this` 具有关联的异步结果，并且结果中没有存储值或异常，那么结果将会置为就绪，伴随一个 `std::future_error` 异常，错误码为 `std::future_errc::broken_promise` 。

抛出
无

std::promise::get_future 成员函数

通过*this关联的异步结果，检索出所要的 `std::future` 实例。

声明

```
std::future<ResultType> get_future();
```

先决条件
*this具有关联异步结果。

返回
与*this关联异步结果关联的 `std::future` 实例。

抛出

当 `std::future` 已经通过get_future()获取过了，将会抛出一个 `std::future_error` 类型异常，伴随的错误码为 `std::future_errc::future_already_retrieved` 。

std::promise::set_value 成员函数

存储一个值到与*this关联的异步结果中。

声明

```
void promise<void>::set_value();
void promise<R&>::set_value(R& r);
void promise<R>::set_value(R const& r);
void promise<R>::set_value(R&& r);
```

先决条件

*this具有关联异步结果。

效果

当ResultType不是void型，就存储r到*this相关的异步结果当中。

后置条件

*this相关的异步结果的状态为就绪，且将值存入。任意等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出 `std::future_error` 型异常，伴随错误码为 `std::future_errc::promise_already_satisfied` 。r的拷贝构造或移动构造抛出的异常，即为本函数抛出的异常。

同步

并发调用set_value()和set_exception()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用 `std::future<Result-Type>::get()` 或 `std::shared_future<ResultType>::get()` ，这两个函数将会查找已存储的异常。

std::promise::set_value_at_thread_exit 成员函数

存储一个值到与*this关联的异步结果中，到线程退出时，异步结果的状态会被设置为就绪。

声明

```
void promise<void>::set_value_at_thread_exit();
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<R>::set_value_at_thread_exit(R const& r);
void promise<R>::set_value_at_thread_exit(R&& r);
```

先决条件
*this具有关联异步结果。

效果
当ResultType不是void型，就存储r到*this相关的异步结果当中。标记异步结果为“已存储值”。当前线程退出时，会安排相关异步结果的状态为就绪。

后置条件
将值存入*this相关的异步结果，且直到当前线程退出时，异步结果状态被置为就绪。任何等待异步结果的阻塞线程将解除阻塞。

抛出
当异步结果已经存有一个值或一个异常，那么将抛出 `std::future_error` 型异常，伴随错误码为 `std::future_errc::promise_already_satisfied` 。r的拷贝构造或移动构造抛出的异常，即为本函数抛出的异常。

同步
并发调用set_value(), set_value_at_thread_exit(), set_exception()和set_exception_at_thread_exit()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用 `std::future<Result-Type>::get()` 或 `std::shared_future<ResultType>::get()` ，这两个函数将会查找已存储的异常。

std::promise::set_exception 成员函数

存储一个异常到与*this关联的异步结果中。

声明

```
void set_exception(std::exception_ptr e);
```

先决条件
*this具有关联异步结果。(bool)e为true。

效果

将e存储到*this相关的异步结果中。

后置条件

在存储异常后，*this相关的异步结果的状态将置为继续。任何等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出 `std::future_error` 型异常，伴随错误码为 `std::future_errc::promise_already_satisfied` 。

同步

并发调用set_value()和set_exception()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用 `std::future<Result-Type>::get()` 或 `std::shared_future<ResultType>::get()` ，这两个函数将会查找已存储的异常。

std::promise::set_exception_at_thread_exit 成员函数

存储一个异常到与*this关联的异步结果中，知道当前线程退出，异步结果被置为就绪。

声明

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

先决条件

*this具有关联异步结果。(bool)e为true。

效果

将e存储到*this相关的异步结果中。标记异步结果为“已存储值”。当前线程退出时，会安排相关异步结果的状态为就绪。

后置条件

将值存入*this相关的异步结果，且直到当前线程退出时，异步结果状态被置为就绪。任何等待异步结果的阻塞线程将解除阻塞。

抛出

当异步结果已经存有一个值或一个异常，那么将抛出 `std::future_error` 型异常，伴随错误码为 `std::future_errc::promise_already_satisfied` 。

同步

并发调用set_value(), set_value_at_thread_exit(), set_exception()和set_exception_at_thread_exit()的线程将被序列化。要想成功的调用set_exception()，需要在之前调用 `std::future<Result-Type>::get()` 或 `std::shared_future<ResultType>::get()` ，这两个函数将会查找已存储的异常。

D.4.5 std::async函数模板

`std::async` 能够简单的使用可用的硬件并行来运行自身包含的异步任务。当调用 `std::async` 返回一个包含任务结果的 `std::future` 对象。根据投放策略，任务在其所在线程上是异步运行的，当有线程调用了这个future对象的wait()和get()成员函数，则该任务会同步运行。

声明

```
enum class launch
{
  async,deferred
};

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(Callable&& func,Args&& ... args);

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(launch policy,Callable&& func,Args&& ... args);
```

先决条件

表达式 `INVOKE(func,args)` 能都为func提供合法的值和args。Callable和Args的所有成员都可MoveConstructible(可移动构造)。

效果

在内部存储中拷贝构造 `func` 和 `arg...` (分别使用fff和xyz...进行表示)。

当policy是 `std::launch::async` ,运行 `INVOKE(fff,xyz...)` 在所在线程上。当这个线程完成时，返回的 `std::future` 状态将会为就绪态，并且之后会返回对应的值或异常(由调用函数抛出)。析构函数会等待返回的 `std::future` 相关异步状态为就绪时，才解除阻塞。

当policy是 `std::launch::deferred` ， fff和xyx...都会作为延期函数调用，存储在返回的 `std::future` 。首次调用future的wait()或get()成员函数，将会共享相关状态，之后执行的 `INVOKE(fff,xyz...)` 与调用wait()或get()函数的线程同步执行。

执行 `INVOKE(fff,xyz...)` 后，在调用 `std::future` 的成员函数get()时，就会有值返回或有异常抛出。

当policy是 `std::launch::async | std::launch::deferred` 或是policy参数被省略，其行为如同已指定的 `std::launch::async` 或 `std::launch::deferred` 。具体实现将会通过逐渐递增的方式(call-by-call basis)最大化利用可用的硬件并行，并避免超限分配的问题。

在所有的情况下， `std::async` 调用都会直接返回。

同步

完成函数调用的先行条件是，需要通过调用 `std::future` 和 `std::shared_future` 实例的wait(),get(),wait_for()或wait_until()，返回的对象与 `std::async` 返回的 `std::future` 对象关联的状态相同才算成功。就 `std::launch::async` 这个policy来说，在完成线程上的函数前，也需要先行对上面的函数调用后，成功的返回才行。

抛出

当内部存储无法分配所需的空间，将抛出 `std::bad_alloc` 类型异常；否则，当效果没有达到，或任何异常在构造fff和xyz...发生时，抛出 `std::future_error` 异常。