**attension:本文严禁转载。**

**一、前言**

      promise/future是一个非常重要的异步编程模型，它可以让我们摆脱传统的回调陷阱，从而使用更加优雅、清晰的方式进行异步编程。c++11中已经开始支持std::future/std::promise，那么为什么folly还要提供自己的一套实现呢？原因是c++标准提供的future过于简单，而folly的实现中最大的改进就是可以为future添加回调函数（比如then），这样可以方便的链式调用，从而写出更加优雅、间接的代码，然后，改进还不仅仅如此。

**二、入门实例**

     让我们先来看一个入门实例，代码如下所示：

[复制代码](javascript:void(0);)

1 #include <folly/futures/Future.h>

2 using namespace folly;

3 using namespace std;

4

5 void foo(int x) {

6 // do something with x

7 cout << "foo(" << x << ")" << endl;

8 }

9

10 // ...

11

12 cout << "making Promise" << endl;

13 Promise<int> p;

14 Future<int> f = p.getFuture();

15 f.then(foo);

16 cout << "Future chain made" << endl;

17

18 // ... now perhaps in another event callback

19

20 cout << "fulfilling Promise" << endl;

21 p.setValue(42);

22 cout << "Promise fulfilled" << endl;

[复制代码](javascript:void(0);)

      代码非常简洁，首先定义一个Promise，然后从这个Promise获取它相关联的Future(通过getFuture接口)，之后通过then为这个Future设置了一个回调函数foo，最后当为Promise赋值填充时（setValue），相关的Future就会变为ready状态（或者是completed状态），那么它相关的回调（这里为foo）会被执行。这段代码的打印结果如下：

making Promise

Future chain made

fulfilling Promise

foo(42)

Promise fulfilled

**三、基本概念**

**1、Promise**

     如果你需要包装一个异步操作、或者向用户提供一个异步编程接口，那么你就可能会用到promise。每一个Future都有一个与之相关的Promise（除了使用makeFuture()产生的处于completed状态的Future）,Promise的使用是很简单的：首先是创建Promise，然后从它“提取”出一个Future,最后在适当的时候向Promise填充一个值或者是异常。

     例如使用setValue填充一个值：

[复制代码](javascript:void(0);)

1 Promise<int> p;

2 Future<int> f = p.getFuture();

3

4 f.isReady() == false

5

6 p.setValue(42);

7

8 f.isReady() == true

9 f.value() == 42

[复制代码](javascript:void(0);)

     下面一个例子是使用setException填充一个异常：

[复制代码](javascript:void(0);)

1 Promise<int> p;

2 Future<int> f = p.getFuture();

3

4 f.isReady() == false

5

6 p.setException(std::runtime\_error("Fail"));

7

8 f.isReady() == true

9 f.value() // throws the exception

[复制代码](javascript:void(0);)

     但是其实更优雅的使用Promise的方式是使用setWith方法，它接收一个函数而且可以自动捕获函数抛出的异常，示例如下：

[复制代码](javascript:void(0);)

1 Promise<int> p;

2 p.setWith([]{

3 try {

4 // do stuff that may throw

5 return 42;

6 } catch (MySpecialException const& e) {

7 // handle it

8 return 7;

9 }

10 // Any exceptions that we didn't catch, will be caught for us

11 });

[复制代码](javascript:void(0);)

      注意：通常来说，在基于Future的编程模型中，多数情况下应该都是单独使用Future而不是Promise（调用返回Future的接口、为Future添加回调函数最终返回另一个Future），Promise在编写底层的异步操作接口时会变得非常有用，比如：

[复制代码](javascript:void(0);)

1 void fooOldFashioned(int arg, std::function<int(int)> callback);

2

3 Future<int> foo(int arg) {

4 auto promise = std::make\_shared<Promise<int>>();

5

6 fooOldFashioned(arg, [promise](int result) {

7 promise->setValue(result);

8 });

9

10 return promise->getFuture();

11 }

[复制代码](javascript:void(0);)

**2、使用then方法为Future设置回调函数**

      前面的例子中，我们都是使用Future的value方法获取值的，除此之外，我们还可以使用回调的方式获取值或者异常，也就是当Promise被填充时，与之相关的Future的回调函数就会被触发执行，例如：

[复制代码](javascript:void(0);)

1 Promise<int> p;

2 Future<int> f = p.getFuture();

3

4 f.then([](int i){

5 cout << i;

6 });

7

8 p.setValue(42);

[复制代码](javascript:void(0);)

      注意：上面的例子中，设置回调的动作和填充Promise的动作之前没有顺序要求，也就是可以先填充Promise再使用then设置回调函数，如果是这样的话，那么回调函数会被立刻执行。

      那么如何获取一个异常呢？上面的例子中，lambda表达式的参数类型为int，这个显然不能传递一个异常，此时你可以把你的回调函数的参数类型设置为Try，该类型既可以捕获正常值又可以捕获一个异常。例如：

1 f.then([](Try<int> const& t){

2 cout << t.value();

3 });

      注意：不推荐在回调函数中使用Try，回调函数中应该只用来捕获值，对于异常的处理和捕获，后文还会讲到更好的方式。同时，当通过then设置回调函数时，这个回调函数的一个副本会被存储在Future中直到它被执行，比如你传递了一个lambda表示式到then中，这个lambda表达式的captures中捕获了一个shared\_ptr，那么Future将会一直持有这个引用直到回调函数被执行。

      then方法的真正威力在于，它会返回一个新的Future，因此可以进行链式嵌套调用，比如：

1 Future<string> f2 = f.then([](int i){

2 return folly::to<string>(i);

3 });

4

5 f2.then([](string s){ / ... / });

       这里，我们在回调函数中改变了Future的值类型（int变为string），因此为f2设置回调函数的参数类型自然就为string，其实我们更推荐以下写法：

1 auto finalFuture = getSomeFuture()

2 .then(...)

3 .then(...)

4 .then(...);

      需要注意的是，上面的代码仍然是同步的，这在组织、编排异步操作的时候是非常有用的。现在假设有一个远程服务（service）负责将int转为string，而你拥有一个返回Future的客户端接口，那么事实上回调函数允许你可以返回一个Future<T>而不仅仅是一个T，例如：

1 Future<string> f2 = f.then([](int i){

2 return getClient()->future\_intToString(i); // returns Future<string>

3 });

4

5 f2.then([](Try<string> const& s){ ... });

      注意：通常情况下，回调函数都是以返回T的形式，除非必须返回Future<T>，这样会使代码变得简洁。

**3、Promise/Future的move语义**

     Promise/Future都支持move语义、但是禁止拷贝的，这可以保证Promise和Future之间的一对一的关系。

**4、同步的创建处于completed状态的Future**

1、可以通过makeFuture<T>()函数创建一个处于completed状态的Future，该函数接收一个T&&类型参数（或者是一个异常类型）。如果T类型是需要被自动类型推断的，那么你可以不用指定它。

2、获取Future的T类型的value值可以通过Future<T>::get()方法，该方法是阻塞的，所以一定要确保该Future已经处于completed状态或者是其他线程将设置该Future的completed状态。当然，get()方法可以接受一个超时时间。

3、可以使用Future<T>::wait()进行同步的阻塞等待，这点和get()很像，唯一不同的是wait()不会提取Future内的值或者异常，wait会返回一个新的Futute，该Future持有input Future的结果。同样，wait也可以设置一个超时时间。

4、getVia()和waitVia()类似于get()和wait()，不同之处在于，它们会在Future处于completed之前一直驱动执行一个Executor。

**5、then的其它重载版本**

     上面关于then的演示中可以看到回调函数的特点：

* **返回值类型：**Future<T> 或 T
* **参数类型：**T const& 或 Try<T> const& (也可能是 T, Try<T>, T&&, 和 Try<T>&&)

    then的灵活性不止于此，then其它重载版本还允许你绑定全局函数、成员函数和静态成员函数，例如：

[复制代码](javascript:void(0);)

1 void globalFunction(Try<int> const& t);

2

3 struct Foo {

4 void memberMethod(Try<int> const& t);

5 static void staticMemberMethod(Try<int> const& t);

6 };

7 Foo foo;

8

9 // bind global function

10 makeFuture<int>(1).then(globalFunction);

11 // bind member method

12 makeFuture<int>(2).then(&Foo::memberMethod, &foo);

13 // bind static member method

14 makeFuture<int>(3).then(&Foo::staticMemberMethod);

[复制代码](javascript:void(0);)

**6、SharedPromise**

      SharedPromise提供了和Promise相同的接口，唯一的不同在于SharedPromise的getFuture()方法可以被多次调用。当SharedPromise被填充时，所有的与之相关的Future都会被回调。在一个已经被填充的SharedPromise上调用getFuture()将返回一个处于completed状态的Future。如果你发现你需要构造一个Promise集合并同时为他们填充相同的值，那么可以考虑使用SharedPromise。

**四、错误处理**

     众所周知，try/catch机制在异步代码中不再是那么通用，因此Future必须提供了一种自然、简洁的错误处理能力。

**1、抛异常**

     有很多种方式可以给Future设置一个异常，比如makeFuture<T>() 和 Promise<T>::setException()可以创建一个 failed  Future，这些异常类型可以是

std::exception、folly::exception\_wrapper、std::exception\_ptr 其中的任何一种。例如：

[复制代码](javascript:void(0);)

1 makeFuture<int>(std::runtime\_error("oh no!"));

2 makeFuture<int>(folly::make\_exception\_wrapper<std::runtime\_error>("oh no!"));

3 makeFuture<int>(std::current\_exception());

4

5 Promise<int> p1, p2, p3;

6 p1.setException(std::runtime\_error("oh no!"));

7 p2.setException(folly::make\_exception\_wrapper<std::runtime\_error>("oh no!"));

8 p3.setException(std::current\_exception());

[复制代码](javascript:void(0);)

     通常情况下，任何时候当你向Future方法传递一个返回Future的函数或者填充一个Promise，你可以放心的是，函数中抛出的任何异常都会被捕获和存储，比如：

1 auto f = makeFuture().then([]{

2 throw std::runtime\_error("ugh");

3 });

      上面的代码是完全正确的，异常会被捕获并被存放在返回的结果Future中，类似的方法还有以下几种：

* Future<T>::then() 和它虽有的变体
* Future<T>::onError(): 后文会提到
* makeFutureTry(): 拿到一个函数并执行它，然后用这个函数的执行结果（或者异常）创建一个Future
* Promise<T>::setWith(): 拿到一个函数并执行它，并用执行结果(或异常)来填充这个Promise

**2、捕获异常**

     同样，在Future编程模型中有很多种方式可以捕获异常。

**1）使用Try**

      Try是一个抽象概念，既可以代表一个值又可以代表一个异常，所以很适合用在then的回调函数中，例如：

[复制代码](javascript:void(0);)

1 makeFuture<int>(std::runtime\_error("ugh")).then([](Try<int> t){

2 try {

3 auto i = t.value(); // will rethrow

4 // handle success

5 } catch (const std::exception& e) {

6 // handle failure

7 }

8 });

9

10 // Try is also integrated with exception\_wrapper

11 makeFuture<int>(std::runtime\_error("ugh")).then([](Try<int> t){

12 if (t.hasException<std::exception>()) {

13 // this is enough if we only care whether the given exception is present

14 }

15 });

16

17 makeFuture<int>(std::runtime\_error("ugh")).then([](Try<int> t){

18 // we can also extract and handle the exception object

19 // TODO(jsedgwick) infer exception type from the type of the function

20 bool caught = t.withException<std::exception>([](const std::exception& e){

21 // do something with e

22 });

23 });

[复制代码](javascript:void(0);)

     但是很不幸的是，上面的代码逻辑导致成功的处理逻辑和错误的处理逻辑相互交织，导致代码不够简洁，同时，上述代码还存在异常过度rethrow的问题。

**2）使用onError()**

      Future<T>::onError() 允许你单独设置一个异常处理器作为回调函数，回调函数的参数类型就是你要捕获处理的异常类型，如果future没有异常，那么这个异常处理回调函数会被直接跳过（忽略），否则，它将会被执行，同时它返回的T或者Future<T>将会变为新的结果Future。这里需要注意的是，多次调用onError和多次catch块的效果是不一样的，也就是说，如果你在一个onError抛出了一个异常，那么下一个onError将会捕获它。

[复制代码](javascript:void(0);)

1 intGenerator() // returns a Future<int>, which might contain an exception

2 // This is a good opportunity to use the plain value (no Try)

3 // variant of then()

4 .then([](int i) {

5 return 10 \* i; // maybe we throw here instead

6 })

7 .onError([](const std::runtime\_error& e) {

8 // ... runtime\_error handling ...

9 return -1;

10 })

11 .onError([](const std::exception& e) {

12 // ... all other exception handling ...

13 return -2;

14 });

[复制代码](javascript:void(0);)

     你也可以直接使用onError直接处理exception\_wrapper，比如当你想处理一个非std::exception异常时，例如：

1 makeFuture().then([]{

2 throw 42;

3 })

4 .onError([](exception\_wrapper ew){

5 // ...

6 });

**3）ensure()**

     Future<T>::ensure(F func)作用非常类型java语言中的finally块，也就是说，它只有一个void类型的函数并最终执行它而不管Future是否包含异常。结果Future将包含前一个Future的值或异常，除非提供给ensure的函数抛出了新的异常，这种情况下该异常会被捕获并传播，例如：

[复制代码](javascript:void(0);)

1 auto fd = open(...);

2 auto f = makeFuture().then([fd]{

3 // do some stuff with the file descriptor

4 // maybe we throw, maybe we don't

5 })

6 .ensure([fd]{

7 // either way, let's release that fd

8 close(fd);

9 });

10

11 // f now contains the result of the then() callback, unless the ensure()

12 // callback threw, in which case f will contain that exception

[复制代码](javascript:void(0);)

**3）异常处理的性能**

     在内部实现中，Future使用folly::exception\_wrapper存储异常以求将rethrow最小化，然而这个机制的有效性取决于我们所使用的库（和exception\_wrapper）是否能够维持异常的类型信息，实际上，这意味着直接构造异常Future而不是使用throw，比如：

[复制代码](javascript:void(0);)

1 // This version will throw the exception twice

2 makeFuture()

3 .then([]{

4 throw std::runtime\_error("ugh");

5 })

6 .onError([](const std::runtime\_error& e){

7 // ...

8 });

9 // This version won't throw at all!

10 makeFuture()

11 .then([]{

12 // This will properly wrap the exception

13 return makeFuture<Unit>(std::runtime\_error("ugh"));

14 })

15 .onError([](const std::runtime\_error& e){

16 // ...

17 });

[复制代码](javascript:void(0);)

      也就是说，直接使用onError而不是通过Try的throwing可以减少rethrow的次数。如果真的想使用Try，那么可以考虑使用

Try<T>::hasException() 和 Try<T>::withException() 来检查和处理异常而不用将他们rethrow。

**五、高阶语义**

      某些时候链式、嵌套使用then还不足够解决所有问题，下面将介绍一些工具便于组装、构建future。

**1、collectAll()**

      collectAll持有一个元素类型为Future<T>的可迭代集合类型，返回一个Future<std::vector<Try<T>>> ，这个返回的Future将在所有的input futures都变为completed状态时变为completed状态。结果(resultant)Future中的vector将按照Future被添加的顺序包含input futures的值（或者异常）。任何组件Future的错误都不会导致这个过程提前终止，input futures都是被move而变得无效，例如：

[复制代码](javascript:void(0);)

1 Future<T> someRPC(int i);

2

3 std::vector<Future<T>> fs;

4 for (int i = 0; i < 10; i++) {

5 fs.push\_back(someRPC(i));

6 }

7

8 collectAll(fs).then([](const std::vector<Try<T>>& tries){

9 for (const auto& t : tries) {

10 // handle each response

11 }

12 });

[复制代码](javascript:void(0);)

      注意：和任何then回调一样，你也可以使用只带一个Try参数的回调，这样可以通过编译，但是你最好不要这么做，因为外部future失败的唯一原因可能是库有一个错误，这个建议同样使用下面的组合操作。

**2、collectAll() variadic**

     这是collectAll的可变长模板版本，它允许你混合、匹配不同类型的Future，它返回Future<std::tuple<Try<T1>, Try<T2>, ...>>类型，例如：

[复制代码](javascript:void(0);)

1 Future<int> f1 = ...;

2 Future<string> f2 = ...;

3 collectAll(f1, f2).then([](const std::tuple<Try<int>, Try<string>>& tup) {

4 int i = std::get<0>(tup).value();

5 string s = std::get<1>(tup).value();

6 // ...

7 });

[复制代码](javascript:void(0);)

**3、collect()**

      collect()有点类似collectAll()，唯一不同就是，如果input Futures中任何一个抛出了异常，那么这个Future将会被提前终止，所以collect()的返回类型为

std::vector<T>。和collectAll()一样，input futures都是被move而变得无效，并且结果(resultant)Future中的vector将按照Future被添加的顺序包含input futures的值（如果全部成功）。例如：

[复制代码](javascript:void(0);)

1 collect(fs).then([](const std::vector<T>& vals) {

2 for (const auto& val : vals) {

3 // handle each response

4 }

5 })

6 .onError([](const std::exception& e) {

7 // drat, one of them failed

8 });

9

10 // Or using a Try:

11 collect(fs).then([](const Try<std::vector<T>>& t) {

12 // ...

13 });

[复制代码](javascript:void(0);)

 4**、collect() variadic**

      这是 collect()的变长模板参数版本，它允许你混合、匹配不同类型的Future，它的返回类型为Future<std::tuple<T1, T2, ...>>。

 5**、collectN()**

      collectN()类似于collectAll()，都持有一个future集合，但是除此之外，它还持有一个size\_t类的N，只要input futures中有N个处于completed状态，那么这个Future就处于completed状态。它的返回类型为Future<std::vector<std::pair<size\_t, Try<T>>>>，每一个pair都持有相关的Future在原始集合中的索引和结果，但是这些pair本身是随机顺序的。同样，input futures都是被move而变得无效。如果input futures中同时有多个Future处于completed状态，获胜者将被选中，但是选择是未定义的。

[复制代码](javascript:void(0);)

1 // Wait for 5 of the input futures to complete

2 collectN(fs, 5,

3 [](const std::vector<std::pair<size\_t, Try<int>>>& tries){

4 // there will be 5 pairs

5 for (const auto& pair : tries) {

6 size\_t index = pair.first;

7 int result = pair.second.value();

8 // ...

9 }

10 });

[复制代码](javascript:void(0);)

 6**、collectAny()**

      collectAny()同样持有一个Future的集合，但是它会在input Futures中的任何一个处于completed状态时变为completed状态，它的返回类型为

Future<std::pair<size\_t, Try<T>>>，其中pair对中持有第一个变为completed状态的Future在原始集合中的索引和结果，input futures都是被move而变得无效。input futures都是被move而变得无效。

1 collectAny(fs, [](const std::pair<size\_t, Try<int>>& p){

2 size\_t index = p.first;

3 int result = p.second.value();

4 // ...

5 });

 7**、map()**

      map()属于Future的高阶函数应用，它持有一个元素类型为Future<A>的集合和一个可以被传递给Future<A>::then()的函数，然后用这些函数作为参数反过来调用集合中每一个Future的then，然后返回一个结果(resultant )future的vector集合（顺序和原始集合一致）。这个过程好比以下代码的语法糖：

1 std::vector<Future<A>> fs;

2 std::vector<Future<B>> fs2;

3 for (auto it = fs.begin(); it < fs.end(); it++) {

4 fs2.push\_back(it->then(func));

5 }

**8、reduce**

      reduce()是Future的另一个高阶函数，它持有一个元素类型为Future<A>的集合，一个类型为B的初始值以及一个拥有两个参数的函数（reducing function，参数类型分别为类型为B的reduced值，来自集合中Future<A>的下一个结果值），该函数的返回值只能为B或者Future<B>，reduce()函数本身返回Future<B>，开始时，初始值和第一个Future的结果值会被应用在该函数上，然后本次应用的结果和第二个Future的结果值会被继续应用在该函数上，以此来推，直到集合中的所有Future都被reduced或者出现了一个未处理的异常。

     reducing function的第二个参数可以为A或者Try<A>，这依赖于你是否想处理input Futures中的异常。如果 input Future中有一个异常并且你没有去Try，那么reduce操作将会被短路，同样，reducing function中抛出的所有异常同样会短路整个reduce操作。

     例如，有一个Future<int> 类型的集合，现在想得到一个Future<bool>用来标识是否集合中所有的Future的值为0，那么可以这样写：

[复制代码](javascript:void(0);)

1 reduce(fs, true, [](bool b, int i){

2 // You could also return a Future<bool> if you needed to

3 return b && (i == 0);

4 })

5 .then([](bool result){

6 // result is true if all inputs were zero

7 });

8 // You could use onError or Try here in case one of your input Futures

9 // contained an exception or if your reducing function threw an exception

[复制代码](javascript:void(0);)

     为了演示异常处理，假设有一个Future<T>类型的集合，现在想获取一个Future<bool>用于标识集合中所有的Future都没有异常，那么可以这么写：

1 reduce(fs, true, [](bool b, Try<T> t){

2 return b && t.hasValue();

3 })

4 .then([](bool result){

5 // result is true if all inputs were non-exceptional

6 });

      最后一个例子来看一下求和的应用：

1 reduce(fs, 0, [](int a, int b){

2 return a + b;

3 })

4 .then([](int sum){

5 // ...

6 });

**六、多线程*via()***

      Promise/Future的核心操作都是线程安全的，如果被误用就会抛异常（比如有些方法重复调用了两次，包括在不同线程中同时调用），比如then()、onError()以及其他设置回调函数的函数，只要被重复调用就会抛出异常。同样，Promise中的setValue()和setException()同样不能调用两次。

      下面先来看一段代码：

[复制代码](javascript:void(0);)

1 // Thread A

2 Promise<Unit> p;

3 auto f = p.getFuture();

4

5 // Thread B

6 f.then(x).then(y);

7

8 // Thread A

9 p.setValue();

[复制代码](javascript:void(0);)

      上面的代码中，x和y分别会在哪个线程执行？不幸的是，这个是不确定的。这里Promise的填充操作和设置回调函数的操作是存在竞态的，如果设置回调函数的动作先发生，那么x和y就会在Promise被填充的线程执行（也就是线程A）。如果Promise的填充操作先发生，那么x和y会在设置回调函数的线程中执行（也就是线程B），而且是立即执行。如果恰好setValue发生在两个then之间，那么x将在线程A中执行，而y会在线程B中执行。可以想象，这种不确定性会带来很多的问题。幸运的是，我们有另一种方法可以解决这个问题。

      Future拥有一个via()函数，该函数需要一个Executor类型的参数。Executor是一个非常简单的接口，它只存在一个线程安全的add(std::function<void()> func) 方法，它会在某个时候执行这个func，尽管不是立即执行。而via()可以确保被设置的回调函数在指定的Executor上执行。例如：

1 makeFutureWith(x)

2 .via(exe1).then(y)

3 .via(exe2).then(z);

      在上面的例子中，y将在exe1中执行，z将在exe2中执行，这是一个相当大的抽象，它不但解决了上文提到的竞态现象，还给我们提供了一个清晰、简洁可控的线程执行模型。比如可以使用不同类型的Executor来执行不同类型的工作（io密集型和cpu密集型）。

      为了便于使用，还存在一个static类型的via版本，它创建并返回一个处于completed状态的Future<Unit> ，同时这个Future的回调被指定在Executor上执行，例如：

1 via(exe).then(a);

2 via(exe, a).then(b);

      via()的一个另类的用法是，把Executor作为第一个参数传递给then，也能保证回调函数在指定的Executor上执行，与via不同的是，使用then设置的Executor不具备粘滞性，也就是只对then本身设置的回调函数有效。

      那么folly都提供了哪些Executor实现呢？

* [ThreadPoolExecutor](https://github.com/facebook/wangle/blob/master/wangle/concurrent/ThreadPoolExecutor.h) ：是一个抽象的线程池实现，支持调整大小、自定义线程工厂、池和每个任务的统计信息、支持NUMA、用户自定义的任务终结。它和它的子类正在积极的开发之中，当前它有两个实现。CPUThreadPoolExecutor（是一个通用线程池，除了上述功能之外，它还支持任务优先级）、IOThreadPoolExecutor (类似CPUThreadPoolExecutor，但是每一个线程都在一个EventBase 事件循环上旋转)。
* EventBase ：是一个Executor，把任务作为一个回调在事件循环上执行。
* ManualExecutor ： 仅在手动起动时执行工作。 这对测试非常有用。
* InlineExecutor ：以内联的方式立刻执行。
* QueuedImmediateExecutor ：类似于InlineExecutor，但在其它回调执行期间添加的工作将被放入等待队列，而不是立即执行。
* ScheduledExecutor：是Executor接口的子接口，支持延迟执行。
* FutureExecutor：包装了其他Executor，并提供了Future<T> addFuture(F func)函数返回一个Future用于异步获取函数的执行结果。这个和futures::async(executor, func) 是等价的。

**七、超时处理**

**1、时间分辨率**

      后面要提到的接收时间的函数和方法时间精度都为Duration类型（std::chrono::milliseconds的别名），但是不要直接使用Duration类型，相反的，应该适当的使std::chrono::duration,例如std::chrono::seconds 或 std::chrono::milliseconds。

**2、TimeKeeper**

      大多数时间相关的方法都有一个可选的TimeKeeper参数。如果你想自己控制Future底层的时间运行那么可以实现TimeKeeper接口，如果没有提供，那么一个默认的单例TimeKeeper将被使用懒汉式创建出来，默认的实现使用folly::HHWheelTimer在一专门的EventBase线程管理超时。

**3、within()**

      Future<T>::within()将返回一个新的Future，如果这个Future没有在指定的时间内变为completed状态，那么将会以一个异常（默认为TimedOut异常）变为completed状态。例如：

[复制代码](javascript:void(0);)

1 using std::chrono::milliseconds;

2 Future<int> foo();

3

4 // f will complete with a TimedOut exception if the Future returned by foo()

5 // does not complete within 500 ms

6 f = foo().within(milliseconds(500));

7

8 // Same deal, but a timeout will trigger the provided exception instead

9 f2 = foo().within(milliseconds(500), std::runtime\_error("you took too long!"));

[复制代码](javascript:void(0);)

**4、onTimeout()**

       Future<T>::onTimeout() 允许你同时设置一个超时时间和超时处理函数，例如：

[复制代码](javascript:void(0);)

1 Future<int> foo();

2 foo()

3 .onTimeout(milliseconds(500), []{

4 // You must maintain the resultant future's type

5 // ... handle timeout ...

6 return -1;

7 })

8 .then(...);

[复制代码](javascript:void(0);)

        细心的你可能会发现上述代码只是下面的一个语法糖。

[复制代码](javascript:void(0);)

1 foo()

2 .within(milliseconds(500))

3 .onError([](const TimedOut& e) {

4 // handle timeout

5 return -1;

6 })

7 .then(...);

[复制代码](javascript:void(0);)

**5、get() and wait() with timeouts**

      可以为get()和wait()设置超时参数，例如：

[复制代码](javascript:void(0);)

1 Future<int> foo();

2 // Will throw TimedOut if the Future doesn't complete within one second of

3 // the get() call

4 int result = foo().get(milliseconds(1000));

5

6 // If the Future doesn't complete within one second, f will remain

7 // incomplete. That is, if a timeout occurs, it's as if wait() was

8 // never called.

9 Future<int> f = foo().wait(milliseconds(1000));

[复制代码](javascript:void(0);)

**6、delayed()**

       Future<T>::delayed()返回一个新的Future，该Future将会被延迟一定时间变为completed状态。例如：

1 makeFuture()

2 .delayed(milliseconds(1000))

3 .then([]{

4 // This will be executed when the original Future has completed or when

5 // 1000ms has elapsed, whichever comes last.

6 });

**7、futures::sleep()**

        sleep() 返回一个Future<Unit>，该Future将会在指定时间间隔之后变为completed状态。

1 futures::sleep(milliseconds(1000)).then([]{

2 // This will be executed after 1000ms

3 });

**八、中断机制**

  中断是一种future持有者向Promose发送信号的机制，假设你的Future代码在另外一个线程中执行了一个耗时很长的操作，一段时间之后你可能不需要这个操作的结果了，那么此时就可以使用中断机制。

      中断机制允许Future机制以异常的形式向Promise发送信号，Promise可以自由的选择异常的处理方式（甚至可以不处理）。例如：

[复制代码](javascript:void(0);)

1 auto p = std::make\_shared<Promise<int>>();

2 p->setInterruptHandler([weakPromise = folly::to\_weak\_ptr(p)](

3 const exception\_wrapper& e) {

4 auto promise = weakPromise.lock();

5 // Handle the interrupt. For instance, we could just fulfill the Promise

6 // with the given exception:

7 if (promise) {

8 promise->setException(e);

9 }

10

11 // Or maybe we want the Future to complete with some special value

12 if (promise) {

13 promise->setValue(42);

14 }

15

16 // Or maybe we don't want to do anything at all! Including not setting

17 // this handler in the first place.

18 });

19

20 auto f = p->getFuture();

21 // The Future holder can now send an interrupt whenever it wants via raise().

22 // If the interrupt beats out the fulfillment of the Promise and there is

23 // an interrupt handler set on the Promise, that handler will be called with

24 // the provided exception

25 f.raise(std::runtime\_error("Something went awry! Abort!"));

26

27 // cancel() is syntactic sugar for raise(FutureCancellation())

28 f.cancel();

[复制代码](javascript:void(0);)