ListenableFuture

Concurrency is a *hard* problem, but it is significantly simplified by working with powerful and simple abstractions. To simplify matters, Guava extends the `Future` interface of the JDK with `ListenableFuture`.

We strongly advise that you always use `ListenableFuture` instead of `Future` in all of your code, because:

- Most `Futures` methods require it.
- It's easier than changing to `ListenableFuture` later.
- Providers of utility methods won't need to provide `Future` and `ListenableFuture` variants of their methods.

Interface

A traditional `Future` represents the result of an asynchronous computation: a computation that may or may not have finished producing a result yet. A `Future` can be a handle to an in-progress computation, a promise from a service to supply us with a result.

A `ListenableFuture` allows you to register callbacks to be executed once the computation is complete, or if the computation is already complete, immediately. This simple addition makes it possible to efficiently support many operations that the basic `Future` interface cannot support.

The basic operation added by `ListenableFuture` is `addListener(Runnable, Executor)`, which specifies that when the computation represented by this `Future` is done, the specified `Runnable` will be run on the specified `Executor`.

Adding Callbacks

Most users will prefer to use `Futures.addCallback(ListenableFuture<V>, FutureCallback<V>, Executor)`. A `FutureCallback<V>` implements two methods:

- `onSuccess(V)`, the action to perform if the future succeeds, based on its result
- `onFailure(Throwable)`, the action to perform if the future fails, based on the failure

Creation

Corresponding to the JDK `ExecutorService.submit(Callable)` approach to initiating an asynchronous computation, Guava provides the `ListeningExecutorService` interface, which returns a `ListenableFuture` wherever `ExecutorService` would return a normal `Future`. To convert an `ExecutorService` to a `ListeningExecutorService`, just use `MoreExecutors.listeningDecorator(ExecutorService)`.

```
ListeningExecutorService service = MoreExecutors.listeningDecorator(Executors.newFixedThread
ListenableFuture<Explosion> explosion = service.submit(
    new Callable<Explosion>() {
      public Explosion call() {
        return pushBigRedButton();
      }
    });
Futures.addCallback(
    explosion,
    new FutureCallback<Explosion>() {
      // we want this handler to run immediately after we push the big red button!
      public void onSuccess(Explosion explosion) {
        walkAwayFrom(explosion);
      }
      public void onFailure(Throwable thrown) {
        battleArchNemesis(); // escaped the explosion!
      }
    },
    service);
```

Alternatively, if you're converting from an API based on `FutureTask`, Guava offers `ListenableFutureTask.create(Callable<V>)` and `ListenableFutureTask.create(Runnable, V)`. Unlike the JDK, `ListenableFutureTask` is not meant to be extended directly.

If you prefer an abstraction in which you set the value of the future rather than implementing a method to compute the value, consider extending `AbstractFuture<V>` or using `SettableFuture` directly.

If you must convert a `Future` provided by another API to an `ListenableFuture`, you may have no choice but to use the heavyweight `JdkFutureAdapters.listenInPoolThread(Future)` to convert a `Future` to a `ListenableFuture`. Whenever possible, it is preferred to modify the original code to return a `ListenableFuture`.

Application

The most important reason to use `ListenableFuture` is that it becomes possible to have complex chains of asynchronous operations.

```
ListenableFuture<RowKey> rowKeyFuture = indexService.lookUp(query);
AsyncFunction<RowKey, QueryResult> queryFunction =
  new AsyncFunction<RowKey, QueryResult>() {
    public ListenableFuture<QueryResult> apply(RowKey rowKey) {
      return dataService.read(rowKey);
    }
  };
ListenableFuture<QueryResult> queryFuture =
    Futures.transformAsync(rowKeyFuture, queryFunction, queryExecutor);
```

Many other operations can be supported efficiently with a `ListenableFuture` that cannot be supported with a `Future` alone. Different operations may be executed by different executors, and a single `ListenableFuture` can have multiple actions waiting upon it.

When several operations should begin as soon as another operation starts – "fan-out" – `ListenableFuture` just works: it triggers all of the requested callbacks. With slightly more work, we can "fan-in," or trigger a `ListenableFuture` to get computed as soon as several other futures have *all* finished: see the implementation of `Futures.allAsList` for an example.

Method	Description	See also
`transformAsync(ListenableFuture<A>, AsyncFunction<A, B>, Executor)*`	Returns a `ListenableFuture` whose result is the product of applying the given `AsyncFunction` to the result of the given `ListenableFuture`.	`transformAsync(ListenableFuture<A>, AsyncFunction<A, B>)`
`transform(ListenableFuture<A>, Function<A, B>, Executor)`	Returns a `ListenableFuture` whose result is the product of applying the given `Function` to the result of the given `ListenableFuture`.	`transform(ListenableFuture<A>, Function<A, B>)`
`allAsList(Iterable<ListenableFuture<V>>)`	Returns a `ListenableFuture` whose value is a list containing the values of each of the input futures, in order. If any of the input futures fails or is cancelled, this future fails or is cancelled.	`allAsList(ListenableFuture<V>...)`
`successfulAsList(Iterable<ListenableFuture<V>>)`	Returns a `ListenableFuture` whose value is a list containing the values of each of the successful input futures, in order. The values corresponding to failed or cancelled futures are replaced with `null`.	`successfulAsList(ListenableFuture<V>...)`

`*` An `AsyncFunction<A, B>` provides one method, `ListenableFuture<B> apply(A input)`. It can be used to asynchronously transform a value.

```
List<ListenableFuture<QueryResult>> queries;
// The queries go to all different data centers, but we want to wait until they're all done
```

```
ListenableFuture<List<QueryResult>> successfulQueries = Futures.successfulAsList(queries);

Futures.addCallback(successfulQueries, callbackOnSuccessfulQueries);
```

Avoid nested Futures

In cases where code calls a generic interface and returns a Future, it's possible to end up with nested `Futures`. For example:

```
executorService.submit(new Callable<ListenableFuture<Foo>() {
  @Override
  public ListenableFuture<Foo> call() {
    return otherExecutorService.submit(otherCallable);
  }
});
```

would return a `ListenableFuture<ListenableFuture<Foo>>`. This code is incorrect, because if a `cancel` on the outer future races with the completion of the outer future, that cancellation will not be propagated to the inner future. It's also a common error to check for failure of the other future using `get()` or a listener, but unless special care is taken an exception thrown from `otherCallable` would be suppressed. To avoid this, all of Guava's future-handling methods (and some from the JDK) have *Async versions that safely unwrap this nesting - `transform(ListenableFuture<A>, Function<A, B>, Executor)` and `transformAsync(ListenableFuture<A>, AsyncFunction<A, B>, Executor)`, or `ExecutorService.submit(Callable)` and `submitAsync(AsyncCallable<A>, Executor)`, etc.