com.google.common.util.concurrent

The Guava Team

- com.google.common.collect
- com.google.common.util.concurrent
- java.util.concurrent
- *Effective Java* (concurrency is in chapter 10)
- *Java Concurrency in Practice*

the linked docs and books also have advice about Java concurrency (general principles, java.util.concurrent)
note: some slides refer to methods that have not yet been released (listeningDecorator, Futures.allAsList/successfulAsList). We expect to release these in Guava 10 in July

collect

- Immutable*
- ConcurrentHashMultiset
- Multimaps.synchronizedMultimap
- MapMaker

before getting to util.concurrent, touch on our other packages
we keep concurrency in mind in our other packages

collect: Immutable*

- whenever possible, for various reasons

use them. linked doc has advantages as well as some tradeoffs
one of many reason to use them: thread safety
immutable in the thread-safe sense
many other classes without "Immutable" in the name are immutable (unlike JDK SimpleDateFormat): CharMatcher, Splitter

collect: ConcurrentHashMultiset

- Multiset<K> ≈ Map<K, Integer> with extra methods and optional thread-safety

```
Map<String, Integer> map = newHashMap();
for (StatsProto proto : protos) {
  String host = proto.getHost();
  if (!map.containsKey(host)) {
    map.put(host, 1);
  } else {
    map.put(host, map.get(host) + 1);
  }
}
```

code counts number of times each host appears in a proto
map from element type to number of occurrences

collect: ConcurrentHashMultiset

- Multiset<K> ≈ Map<K, Integer> with extra methods and optional thread-safety

```
Multiset<String> multiset =
    HashMultiset.create();
for (StatsProto proto : protos) {
  multiset.add(proto.getHost());
}
```

- NumericMap (someday)
- prefer immutable

e.g., count number of queries to each shard
NumericMap to support long, double for, e.g., total nanoseconds spent in an operation
sometimes need modifiability, e.g., stats (probably from different threads, therefore concurrency)

collect: Multimaps.synchronizedMultimap

- Multimap<K, V> ≈ Map<K, Collection<V>> with extra methods and optional thread-safety

```
Map<String, List<StatsProto>> map =
    newHashMap();
for (StatsProto proto : protos) {
  String host = proto.getHost();
  if (!map.containsKey(host)) {
    map.put(host,
        new ArrayList<StatsProto>());
  }
  map.get(host).add(proto);
}
```

code indexes protos by host

collect: Multimaps.synchronizedMultimap

- Multimap<K, V> ≈ Map<K, Collection<V>> with extra methods and optional thread-safety

```
Multimap<String, StatsProto> multimap =
    ArrayListMultimap.create();
for (StatsProto proto : protos) {
  multimap.put(proto.getHost(), proto);
}
```

- synchronized performs better than our internal wait-free equivalent
- prefer immutable

this particular code could use Multimaps.index()
synchronization especially painful with "check then act" necessary for the first item (multiset and multimap both)

collect: MapMaker

- slides to come in a few months

build concurrent maps and caches

collect: MapMaker

- slides to come in a few months

```
private final ConcurrentMap<String, Feed> feedCache =
    new MapMaker()
        .expireAfterWrite(2, MINUTES)
        .maximumSize(100),
        .makeMap();
```

on-demand computation with computing maps. with a "normal" cache, you must look in the cache, perform the operation, and insert the result into cache. computing map hides cache management: register a computing Function, then just call feedCache.get

`collect:` `MapMaker`: features

- concurrent requests to a computing map share the same computation (TODO: sharing without caching)
- **prefer `Multiset` / `Multimap` / `Table`**

share computation an advantage over manual cache management: request LDAP groups for user jsmith concurrently in two threads: only one call is made. Of course, results are cached. Someday MapMaker will allow you to turn off caching so that serves only to combine in-flight queries
we see people using MapMaker.makeComputingMap() to create a Map<K, Collection<V>>, Map<K, Integer> etc. don't forget Multiset / Multimap / Table, which offer nicer interfaces
(Table is a two-keyed Map)

util.concurrent

- Future basics
- ListenableFuture
- Futures
- more about Future
- Service
- Executor

`Future` basics

"A handle to an in-progress computation."

"A promise from a service to supply us with a result."

different phrasings of the same thing

`Future` basics

```
Map<LocalDate, Long> pastSales =
    archiveService.readSales();
Map<LocalDate, Long> projectedSales =
    projectionService.projectSales();

return buildChart(pastSales, projectedSales);
```

we tell a network thread to make a request, and then we block. the network thread sends the request to a server. the server responds to a network thread, and the network thread unblocks us
repeat
~5s each; done in ~10s
our thread is doing nothing; it handed off work to another thread/machine
there's no reason not to overlap the two do-nothing periods, but this code can't

Future basics

```
inparallel {
  Map<LocalDate, Long> pastSales =
      archiveService.readSales();
  Map<LocalDate, Long> projectedSales =
      projectionService.projectSales();
}
return buildChart(pastSales, projectedSales);
```

a possible solution is to modify the language to support an inparallel keyword
instead of waiting for the sum of the two operations' durations, we now wait for the max

Future basics

```
Future<Map<LocalDate, Long>> pastSales =
    archiveService.readSales();
Future<Map<LocalDate, Long>> projectedSales =
    projectionService.projectSales();

return buildChart(pastSales.get(), projectedSales.get());
```

in Java, there's no inparallel, but we can change our methods to return a Future
this allows us to split the operation into "make request" and "wait for results" phases
the method calls now make queries but don't wait for their results; when we're done with all our other work, we call get(), which does wait

util.concurrent: ListenableFuture

- What
- Why
- When
- How

`util.concurrent`: `ListenableFuture`: the what

- `Future` with one new method: `addListener(Runnable, Executor)`
- when the future is done (success, exception, cancellation), the listener runs

if the Future is already done at the time of the addListener call, the listener is invoked when it's added. in short, if you call addListener, your listener will run as long as future doesn't run forever
ListenableFuture implementation responsible for automatically invoking when finished
works like FutureTask.done, if you're familiar with that
executor doesn't even see listener until future is done, so (1) there is no overhead or wasted threads and (2) the listener can call get() and know it won't wait
before, we blocked to get the result of a future; now we can set a callback
why would we want to do that? see next slides

util.concurrent: ListenableFuture:

the why: callbacks

```
service.process(request).addListener(new Runnable() {
  public void run() {
    try {
      Result result = Futures.makeUninterruptible(future).get();
      // do something with success |result|
    } catch (ExecutionException ee) {
      // do something with failure |ee.getCause()|
    } catch (RuntimeException e) {
      // just to be safe
    }
  }
}, executor);
```

ListenableFuture is a single, common interface for both callbacks and futures
but Runnable isn't the ideal callback interface
boilerplate: process+addListener, makeUninterruptible, two exceptions, unwrap in one case
(yes, some people throw RuntimeException from future.get(), even though perhaps they shouldn't, and if your system is callback based, you'd better catch it)

util.concurrent: ListenableFuture:

the why: callbacks

```
service.process(new AsyncCallback<Result>() {
  public void success(Result result) {
    // do something with success |result|
  }
  public void failure(Exception e) {
    // do something with failure |e|
  }
}, executor);
```

We may one day provide an adapter to support this use.

compare to GWT callback interface
not The One Thing that ListenableFuture is good for

util.concurrent: ListenableFuture: the why: "aspects"

```
future.addListener(new Runnable() {
  public void run() {
    processedCount.incrementAndGet();
    inFlight.remove(name);
    lastProcessed.set(name);
    LOGGER.infofmt("Done with %s", name);
  }
}, executorService);
```

"aspects" in the sense of aspect-oriented programming, code that runs automatically every time we do something without inserting that code into the main implementation
traditional example is that you have an RPC interface and want to log every call. you could reimplement the interface yourself such that every method does two things, log and delegate. or you could use guice or some other interceptor interface to implement one method to be automatically invoked whenever a call is made
here you're not the one who is using the output of the future (or at least you're not the primary consumer); someone else will call get() on it later to access the result
This can work, but it's not the most popular use of ListenableFuture, either.

util.concurrent: ListenableFuture: the why: building blocks

- Given several input futures, produce a future that returns the value of the first to complete successfully.
- Offload postprocessing to another thread.

the killer app: "To serve as a foundation for higher-level abstractions"
these tasks are examples of things you can't do with a plain Future (or can't do efficiently)
digression: "first to complete succesfully" is what ExecutorCompletionService does for Futures created by submission to an executor; that class insert callbacks in the same place as ListenableFuture
in the postprocessing example, we want to start postprocessing immediately, so we don't want to kick off multiple operations, wait for all of them to finish, kick off multiple postprocessings, wait for all of them to finish, etc. We want postprocessing to start automatically when a task completes
we'll look at a few libraries that use ListenableFuture later
so ListenableFuture is OK for some things, good for others. when to use...?

`util.concurrent`: `ListenableFuture`: the when

Always.

`util.concurrent`: `ListenableFuture`: the when

Always.

- (+) 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.
- (−) "`ListenableFuture`" is lengthier than "`Future`."
- (−) Classes like `ExecutorService` give you a plain `Future` by default.

cost of creating and passing around ListenableFuture is small
you might need it now, or you (or a caller) might need it later

`util.concurrent`: `ListenableFuture`: the how

Create `ListenableFuture` instead of plain `Future`:

`ExecutorService.submit(Callable)` ➜

Call `MoreExecutors.listeningDecorator` on your executor.

`MyFutureTask.set(V)` ➜

Use `SettableFuture`.

you need to decide that you want a ListenableFuture at *creation* time
we used to have method called blockAThreadInAGlobalThreadPoolForTheDurationOfTheTask to adapt to ListenableFuture. no, that's not really what it was called (really "makeListenable"), but that's how it worked
this is necessarily how any after-the-fact Future->ListenableFuture converter must work, as someone needs to invoke the listeners. by creating a listener-aware future from the beginning, you're letting the thread that sets the future's value do that
makeListenable was a pain when it appeared in tests: it's much easier if you can guarantee that listener runs right away, not when background blocking thread notices
Most futures ultimately work in one of two ways / two "kinds" of futures (executor submission and manual set()); we have utilities to create both
listeningDecorator to automatically make all submissions return ListenableFuture
if you are already using your own FutureTask subclass, subclass ListenableFutureTask instead
or use AbstractListenableFuture - *not* AbstractFuture

util.concurrent: Futures

- transform
- chain
- allAsList / successfulAsList
- others that I won't cover here

I said I'd give some examples of methods operating on futures; here they are

util.concurrent: Futures: transform

```
Future<QueryResult> queryFuture = ...;
Function<QueryResult, List<Row>> rowsFunction =
    new Function<QueryResult, List<Row>>() {
      public List<Row> apply(QueryResult queryResult) {
        return queryResult.getRows();
      }
    };
Future<List<Row>> rowsFuture =
    transform(queryFuture, rowsFunction);
```

trivial postprocessing that won't fail: e.g., proto to java object
the output value is the output of the Function, or an exception if the original future failed

util.concurrent: Futures: chain

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

chain() is transform() on steroids: the transformation can fail with a checked exception, and it can be performed asynchronously
heavy, multi-stage queries, like an index lookup + data lookup
the work done during the original listenablefuture is the first step; the function derives the second step from the result of the first
the output value is the output of the Future returned by the Function, or an exception if the original future failed
you could perform the stages in series yourself, but you might want multiple such chains of execution to occur in parallel, so you need to keep everything in terms of Future

util.concurrent: Futures: allAsList / successfulAsList

- List<Future> ➜ Future<List>
- Difference is exception policy:

  allAsList

  fails if any input fails

  successfulAsList

  succeeds, with null in place of failures

`util.concurrent`: He's still talking about `Future`

Don't implement it yourself.

- Avoid:
  - deadlocks
  - data races
  - `get()` that returns different values at different times
  - `get()` that throws `RuntimeException`
  - extra calls to listeners
  - conflating two kinds of cancellation
- Remember: `MoreExecutors.listeningDecorator`, `SettableFuture`.

I couldn't write a correct Future implementation from scratch without a lot of research. even if I did, it would be slow
you might think it would be hard to write a future that returns different values at different times, but I edited some code that had this behavior and didn't notice that it worked that way until I ran the tests after my change and found that they failed
RuntimeException breaks callbacks that call get() and don't check for it
Future cancellation is a talk unto itself; it's probably not a big deal if your cancel() implementation is broken, but why not get it right for free?
also occasionally useful are AbstractListenableFuture and ListenableFutureTask

`util.concurrent`: He's still talking about `Future`

Don't even mock it (usually).

- Don't Mock Data Objects (aka Value Objects).
- "Data object?" `Future` is more `Queue` than `List` or proto.
- Contrast to service objects:
  - Fragile:
    - accessed by `get` (timed or untimed), `addListener`, or `isDone`
    - vs. a service object with only one method per operation (usually)
  - Lightweight:
    - `immediateFuture(userData)`
    - vs. a test Bigtable with the user data

examples: a protocol buffer is a value object; an rpc stub is a service object
is Future, with all its behavior, really a data object? more or less
think of it like Queue, which is a concurrent collection but still a collection. even List has behavior. only some data classes have none whatsoever
mocking Future: you don't want to, and you don't need to
fragility: state is accessed by multiple methods: multiple get()s, or peek()/poll()/.... a change in implementation of chain() to access that state in different ways could break your mocks
by contrast, while service objects certainly *can* have multiple methods that do the same thing (e.g., sync and async RPC variants), this occurs less often
"immediateFuture: it's easier than setting up a Bigtable test cell"

util.concurrent: Executor

MoreExecutors.sameThreadExecutor

for quick tasks that can run inline

MoreExecutors.getExitingExecutorService

for "half-daemon" threads

UncaughtExceptionHandlers.systemExit

for exiting after unexpected errors

ThreadFactoryBuilder

```
new ThreadFactoryBuilder()
    .setDaemon(true)
    .setNameFormat("WebRequestHandler-%d")
    .build();
```

plus others I won't cover here
use sameThreadExecutor only when you literally don't care which thread runs the task; think of it as "anyThreadExecutor"
don't get cute, e.g. "my future's value comes from a task in thread pool X, so if I use sameThreadExecutor, my listener will run in that thread pool." that's *usually* true, but if your addListener call occurs *after* the future completes, now your listener is running in the thread that invoked addListener instead
don't let me scare you away from sameThreadExecutor entirely, but reserve it for fast tasks only
"half-daemon" solves the following problem with important background tasks: if your threads are non-daemon, the process can't exit automatically; if they're daemon, the process will exit without waiting for them to finish
getExitingExecutorService threads keep the VM alive only as long as they are doing something
one configuration option you have when setting up a thread pool is what to do with unhandled exceptions. by default, they're printed to stderr (not your logs), and the thread (not the process) dies, which might be bad if thread is important; maybe you don't know what it was in the middle of
another option in setting up thread pool is to set other properties of individual threads; to help, we provide ThreadFactoryBuilder

util.concurrent: Service

- definition
- lifecycle
- implementation

`util.concurrent`: `Service`: definition

- "An object with an operational state, plus asynchronous `start()` and `stop()` lifecycle methods to transfer into and out of this state."
- web servers, RPC servers, monitoring initialization, ...

anything you might start when starting your process and maybe take down at the end
the value of the interface is the common API for us to adapt all our services to, hiding threading, etc.; the implementation could change how it uses threads without affecting users
this frees users from thinking about these details
and it allows users to build libraries that can run atop any generic Service
this puts the burden of handling threading on the Service implementer, but we simplify things for them with helper classes. More on that in a minute

util.concurrent: Service: lifecycle

- States:
  - NEW ➜
  - STARTING ➜
  - RUNNING ➜
  - STOPPING ➜
  - TERMINATED

Note the lack of restart - it's a one way street. It's because our Service actually models a 'service invocation'.
you can trigger start/stop and optionally wait for them to complete with... a ListenableFuture

util.concurrent: Service: implementation

- AbstractExecutionThreadService
- AbstractIdleService
- AbstractService

choose an implementation based on how your service does its threading: single-threaded, multi-threaded, and arbitrarily threaded, respectively
I'll show a sample implementation using each

util.concurrent: Service: AbstractExecutionThreadService

```
protected void startUp() {
  dispatcher.listenForConnections(port, queue);
}
protected void run() {
  Connection connection;
  while ((connection = queue.take() != POISON)) {
    process(connection);
  }
}
protected void triggerShutdown() {
  dispatcher.stopListeningForConnections(queue);
  queue.put(POISON);
}
```

start() calls your startUp() method, creates a thread for you, and invokes run() in that thread. stop() calls triggerShutdown() and waits for the thread to die

util.concurrent: Service: AbstractIdleService

```
protected void startUp() {
  servlets.add(new GcStatsServlet());
}
protected void shutDown() {}
```

for when you need a thread only during startup and shutdown (here, any queries to the GcStatsServlet already have a request thread to run in)

util.concurrent: Service: AbstractService

```
protected void doStart() {
  new Thread("webserver") {
    public void run() {
      try {
        notifyStarted();
        webserver.run();
      } catch (Exception e) {
        notifyFailed(e);
      }
    }
  }.start();
}
```

```
protected void doStop() {
  new Thread("stop webserver") {
    public void run() {
      try {
        webserver.blockingShutdown();
        notifyStopped();
      } catch (Exception e) {
        notifyFailed(e);
      }
    }
  }.start();
}
```

for services that require full, manual thread management
here, we need manual thread management because our webserver doesn't have an asynchronous stopAndNotifyCallback method. stop() isn't allow to block, so our doStop() kicks off its own thread
if not for that, we could use AbstractExecutionThreadService, with the contents of doStart() moved to run()
the lack of an asynchronous shutdown method is exactly the kind of annoyance that Service exists to paper over

Questions?

- Bugs
- Usage (use the tag guava)
- Discussion