MapMaker Migration

Introduction

All caching related methods on `MapMaker` have been deprecated in favor of similar methods in `CacheBuilder`, and are scheduled for upcoming deletion. Release 11.0.0 already removed `evictionListener`, `expireAfterWrite`, and `expireAfterAccess`. Future releases will remove `makeComputingMap` and `expiration`.

Most `MapMaker` use cases should be migrated to either `CacheBuilder` or `AtomicLongMap`. Specifically, cases when `MapMaker` is used to construct maps with `AtomicLong` values should generally be migrated to `AtomicLongMap`. Other cases where `MapMaker` caching functionality is used (including all uses of `MapMaker.makeComputingMap(Function)`) should be migrated to `CacheBuilder`.

Migrating from `MapMaker.makeMap()` to `CacheBuilder.build()` is trivial, but migrating from `MapMaker.makeComputingMap(Function)` to `CacheBuilder.build(CacheLoader)` involves some subtle behavioral changes. Specifically, `MapMaker.makeComputingMap(Function)` returns a `ConcurrentMap`, while `CacheBuilder.build(CacheLoader)` returns a `LoadingCache`. While `LoadingCache` has an `asMap()` method, the map returned by that map is much different from the map created by `MapMaker`.

`MapMaker` and `ConcurrentMap`

`MapMaker.makeComputingMap(Function)` returned a magical (and ill-behaved) `ConcurrentMap`. Specifically, calls to `Map.get(Object)` on the returned map would automatically compute values for absent keys using the specified `Function`. These computations would be shared by all concurrent computations on the same key. Such maps were, in effect, autovivification maps.

At first glance this behavior is tremendously useful, but the specific implementation of this functionality behind a plain `ConcurrentMap` was riddled with issues.

Having a `Map` that auto-creates entries on `get` was simply a big mistake. It breaks type-safety (you can use it to store a key in the map that isn't of the map's key type!). Bad things will happen if that `Map` accidentally gets passed to another `Map`'s `equals()` method. Common idioms for `Map` usage (in the absence of null values) are based on the assumption of interchangeability of `containsKey(k)` and `(get(k) != null)`, and those coding patterns will break. Etc.

We studied this issue very closely, and concluded that our library will be easier to use when collections are just collections, iterators are just iterators, and things that are fancier than those have public types that convey their behavior sufficiently.

CacheBuilder and LoadingCache

And thus we introduced the `LoadingCache` interface. The primary intent of this new interface was to encapsulate a `get(K)` method which auto-created entries, while still exposing an `asMap()` view which allowed traditional map-style access to the cache internals. In other words, the magical `get` from `MapMaker.makeComputingMap(Function)` was semantically separated from the other `ConcurrentMap` methods. Note that `LoadingCache.get(K)` will automatically load absent entries, however `LoadingCache.asMap.get(Object)` will *not*.

The new `LoadingCache` interface came with a new builder, `CacheBuilder`, patterned after `MapMaker` but with an explicit focus on caching, and only capable of producing `LoadingCache` (and `Cache`) instances, instead of `ConcurrentMaps`.

Migrating from MapMaker to CacheBuilder

Now that we understand the key distinction between `MapMaker` and `LoadingCache` we turn to the subject of migrating old code from `MapMaker` to `CacheBuilder`.

Computing caches

The biggest difference is the change from using a plain `Function` to compute values to a more sophisticated `CacheLoader` type.

`CacheLoader` has a few major differences from `Function`:

- Its `load(K key)` method is permitted to throw exceptions.
- It provides a `loadAll(Iterable<? extends K>)` method to load multiple keys at once – possibly concurrently. (By default, `loadAll` just sequentially loads each key individually with the `load` method.)
- It provides a `reload(K key, V oldValue)` method for use in *refreshing* cached values asynchronously, for caches configured with `refreshAfterWrite`. By default, this synchronously calls `load`.

The simplest way to migrate a `Function`-based computing map to a `CacheLoader` is the `CacheLoader.from(Function)` adapter, which views a `Function` as a `CacheLoader`, no special effort required. That said, it's silly to call `CacheLoader.from(new Function<K, V>() {...})` when you can just write

```
new CacheLoader<K, V>() {
  public V load(K key) {
    // copy/paste code from Function.apply
  }
};
```

`asMap` view

The biggest difference between the computing maps generated by `MapMaker.makeComputingMap` and the `ConcurrentMap asMap()` view of a `Cache` is that the `asMap()` view will never compute new values on a call to `asMap().get(key)`. This is specifically deliberate to avoid the "magical" unpredictable behavior of computing maps.