# Java Caching with Guava

Charles Fry (fry@google.com)

# Introduction

- The Guava project is an open-source release of Google's core Java libraries
  - Stuff like collections, primitives support, concurrency libraries, string processing, & cetera
  - These are the libraries that other projects are built on
- The package `com.google.common.cache` contains our caching libraries
  - Simple, in-memory caching
  - Thread-safe implementation (internally similar to `ConcurrentHashMap`)
  - No explicit support for distributed caching

# Types of Caches

- We provide two types of caches
  - `LoadingCache`: knows how to load entries when a cache miss occurs
    - `LoadingCache.get(key)` returns the value associated with `key`, loading it first if necessary
  - `Cache`: does not automatically load entries
- We're going to focus on the loading case here; it's usually what you want

# Simple Loading Cache

```
CacheLoader<String, String> loader =
  new CacheLoader<String, String>() {
    public String load(String key) {
      return key.toUpperCase();
    }
  };

LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .build(loader);
```

# Simple Loading Cache

```
cache.size(); // returns 0

cache.getUnchecked("simple test");
// cache miss, invokes CacheLoader
// returns "SIMPLE TEST"

cache.size(); // returns 1

cache.getUnchecked("simple test");
// cache hit
// returns "SIMPLE TEST"
```

# Concurrency

Google™

- Cache instances are internally implemented very similar to `ConcurrentHashMap`
  - And are thus thread-safe
- But what happens if multiple threads simultaneously request the same key?
- `CacheLoader.load` will be invoked a single time for each key, regardless of the number of requesting threads
  - The result will be returned to all requesting threads and inserted into the cache using the equivalent of `putIfAbsent`

# Checked Exceptions

- What if loading causes a checked exception?

```
CacheLoader<String, String> checkedLoader =
  new CacheLoader<String, String>() {
    public String load(String key)
        throws IOException {
      return loadFromDisk(key);
    }
  };
```

# Checked Exceptions

Google™

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .build(checkedLoader);

try {
  cache.get(key);
} catch (ExecutionException e) {
  // ensure stack trace is for this thread
  throw new IOException(e.getCause());
}
```

# Weak Keys

- What if the cache keys are transient objects (e.g. requests), which don't belong in the cache if there are no other references elsewhere?

```
LoadingCache<Request, Metadata> cache =
  CacheBuilder.newBuilder()
    .weakKeys()
    .build(loader);
```

- Allow the garbage collector to immediately collect cache keys when other references are gone
- Causes key equality to be determined using ==
- Cost: 3 new references, adding 16 bytes per entry

# Eviction

- So far the caches we've shown you will grow without bound
- `CacheBuilder` can automatically evict elements based on various criteria

# Eviction: Maximum Size

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .maximumSize(200)
    .build(loader);
```

- Elements will be evicted in *approximate* LRU order
- Costs:
  - Every access now becomes a lightweight write (to record access order)
  - Evictions occur on write operations
  - 2 new references, in a doubly-linked access queue, adding 16 bytes per entry

# Eviction: Maximum Weight

```
Weigher<String, String> weighByLength =
  new Weigher<String, String>() {
    public int weigh(
        String key, String value) {
      return value.length();
    }
  };
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .maximumWeight(2000)
    .weigher(weighByLength)
    .build(loader);
```

# Eviction: Maximum Weight

- Eviction order is the same as `maximumSize`
  - In fact they share the same data structure (and cost)
  - However more than one entry may be evicted at a time (making room for a single large entry)
- Weight is only measured once, when an entry is added to the cache
- Weight is only used to determine whether the cache is over capacity; not for selecting what to evict

# Cache Stats

- With an automatic eviction policy in play, one starts to wonder about cache performance
  - What ratio of requests are served directly from cache?
  - How much time is spent loading entries?
- These and other questions can be answered with:

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .recordStats()
    .build(loader);
// cumulative stats since cache creation
CacheStats stats = cache.stats();
```

# Cache Stats

```
CacheStats stats = cache.stats();
stats.hitRate();
stats.missRate();
stats.loadExceptionRate();
stats.averageLoadPenalty();

CacheStats delta = cache.stats()
   .minus(stats);
delta.hitCount();
delta.missCount();
delta.loadSuccessCount();
delta.loadExceptionCount();
delta.totalLoadTime();
```

# Eviction: Time to Idle

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .expireAfterAccess(2, TimeUnit.MINUTES)
    .build(loader);
```

- Elements will expire after the specified time has elapsed since the most recent access
- Eviction order is the same as `maximumSize`
  - They share the same data structure (and cost)
  - However cache size will be dynamic instead of static
  - Evictions performed on read or write operations
- Cost: 2 new references, in a doubly-linked write queue, adding 16 bytes per entry
- Tests can advance time with `CacheBuilder.ticker`

# Eviction: Time to Live

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .expireAfterWrite(2, TimeUnit.MINUTES)
    .build(loader);
```

- Elements will expire after the specified time has elapsed since the entry's creation or update
- Useful for dropping stale data from the cache
  - Unlike other expiration strategies this is more about data correctness than resource conservation
- Cost: 2 new references, in a doubly-linked write queue, adding 16 bytes per entry

# Eviction: Soft Values

```
LoadingCache<String, String> cache =
   CacheBuilder.newBuilder()
      .softValues()
      .build(loader);
```

- Allow the garbage collector to collect cached values
  - VMs "bias against clearing recently-created or recently-used soft references"
  - But in practice "SoftReferences will always be kept for at least one GC after their last access"
- Cost: 4 new references, adding 16 bytes per entry
- Performance: O(?), large production systems can be very adversely affected by many soft references
  - Consider `maximumSize` instead (or also)

# Cache Configuration

- `CacheStats` give insight into cache performance, and open the door for optimizing the cache configuration
- Cache configuration parameters can be changed without recompiling code with `CacheBuilderSpec`

```
// from command-line flag or config file
String spec =
   "maximumSize=200,expireAfterWrite=2m";
LoadingCache<String, String> cache =
   CacheBuilder.from(spec)
      .build(loader);
```

# Removal Notifications

Google™

- Sometimes cached entries are associated with resources which need to be closed or cleaned up
- Removal notifications can be sent for each entry which is removed from the cache, containing the removed key and value (if available) and the cause of removal

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .maximumSize(200)
    .removalListener(listener)
    .build(loader);
```

# Removal Notifications

```
RemovalListener<String, String> listener =
    new RemovalListener<String, String>() {
      public void onRemoval(
          RemovalNotification<String, String> n) {
        if (n.wasEvicted()) {
          cleanupEntry(n.getKey(), n.getValue());
        }
      }
    };
```

# Removal Notifications

- Removal notifications include a `RemovalCause`, though it is generally sufficient to check `wasEvicted()`
- Removal listeners are called synchronously during user operations
  - Consider implementing `RemovalListener` asynchronously (or wrapping with `RemovalListeners.asynchronous`)
- Removal listeners shouldn't blindly re-insert removed elements back into the cache

# Refreshing Stale Entries

Google™

- We've already seen how `expireAfterWrite` can remove stale entries
- In cases where stale data should be served while fresh data is being loaded, the method `LoadingCache.refresh(K)` can be used to request a reload
  - Reload will be performed by calling `CacheLoader.reload(K, V)`, which can be implemented asynchronously
  - Reload can take the old cached value into consideration for higher efficiency
  - The stale value will continue to be returned until reload completes

# Automatic Refresh

Google™

- Alternatively, stale entries can be automatically refreshed

```
LoadingCache<String, String> cache =
  CacheBuilder.newBuilder()
    .refreshAfterWrite(2, TimeUnit.MINUTES)
    .build(loader);
```

- We call `LoadingCache.refresh` for you the first time get is called after the timeout
- Inactive entries will *not* be proactively refreshed
  - Couple with `expireAfterWrite` to purge these

# Automatic Refresh

Google™

- Benefits of automatic refresh over expiration for dealing with stale data:
  - Reload can be optimized based on the previous cached value
  - The stale value will continue to be served during reload (rather than blocking other threads)
  - Reload can be implemented asynchronously, decreasing cache latency

# Asynchronous Refresh

- Avoid blocking any user threads by providing an asynchronous `CacheLoader.reload` implementation

```
public ListenableFuture<String> reload(
    final String key, final String oldValue) {
  ListenableFutureTask<String> task =
    ListenableFutureTask.create(
      new Callable<String>() {
        public String call() {
          return load(key);
        }
      });
  executor.execute(task);
  return task;
}
```

# Bulk Operations

- Sometimes it's more efficient for a `CacheLoader` to load a set of entries simultaneously rather than one at a time
- This can be accomplished by overriding `CacheLoader.loadAll`, and then querying through `LoadingCache.getAll`
- Unlike `LoadingCache.get`, `getAll` does *not* block multiple requests for the same key
  - Doing so would dramatically increase its cost, as keys may be spread over multiple segments

# Manual Cache Writes

Google™

- So far the *only* way a value can ever get into the cache is if it comes from the `CacheLoader` you specified when creating the cache
  - Which encourages consistent cache content
- But we also support manual writes to the cache
- And reads from the cache which *don't* load missing values

```
String v = cache.getIfPresent("one");
// returns null
cache.put("one", "1");
v = cache.getIfPresent("one");
// returns "1"
```

# Get or Compute

Google™

- Alternatively a new value can be loaded from a `Callable` on cache misses

```
String v = cache.get(key,
   new Callable<String>() {
     public String call() {
       return key.toLowerCase();
     }
   });
```

- Concurrent requests for the same absent key will result in a single computation which will be returned to all threads

# Non-Loading Caches

Google™

- In fact, you don't even need a `CacheLoader` at all
  - We still recommend them for consistency
  - But sometimes it's impractical to define a `CacheLoader` at cache-creation time
- If you call `CacheBuilder.build()` (without specifying a `CacheLoader`) you get back a non-loading `Cache`
  - Which implements `put`, `getIfPresent`, and `get (K, Callable)`
  - In fact, `LoadingCache` extends `Cache`, which contains all of the non-loading methods

# Disable Caching

Google™

- Sometimes it's necessary to simply turn off caching
- The canonical way to do this is using `maximumSize(0)`
  - Can be done without recompiling using `CacheBuilderSpec`
  - Concurrent lookups of the same key will still result in a single load request, but the result will be evicted immediately

# Map View

- You can view the entries stored inside the cache as a map using `Cache.asMap()`
- Notice: `LoadingCache.get(K)` and `Map.get(Object)` have similar-looking signatures, but remember that they are very different
  - `Map.get` is really a "get if present" method, analogous to `Cache.getIfPresent`
- This can be convenient for iterating over cache content
- All `ConcurrentMap` write operations are implemented
  - However the canonical way to write to a cache is still with a `CacheLoader` or a `Callable`

# Future Work

- `AsyncLoadingCache`, where `get(K)` returns `Future<V>`
- `CacheBuilder.withBackingCache(Cache)` to facilitate cache layering (L1 + L2 cache)
- Many performance optimizations, including migrating internals to new `ConcurrentHashMap`

# FIN

# Why not Map?

Google™

- Map.get causes a type-safety hole
- Map.get is a read operation and users don't expect it to also write
- Map.equals is not symmetric on a self-populating Map
- No way to lookup without causing computation
- Either pending computations can be overwritten by explicit writes or writes must block on pending computations
- Interface fails to convey the intent (caching!)