Hashing

Overview

Java's baked-in concept of hash codes is constrained to 32 bits, and provides no separation between hash algorithms and the data they act on, so alternate hash algorithms can't be easily substituted. Also, implementations of hashCode tend to be poor-quality, in part because they end up depending on other existing poor-quality hashCode implementations, including those in many JDK classes.

Object.hashCode implementations tend to be very fast, but have weak collision prevention and no expectation of bit dispersion. This leaves them perfectly suitable for use in hash tables, because extra collisions cause only a slight performance hit, while poor bit dispersion is easily corrected using a secondary hash function (which all reasonable hash table implementations in Java use). For the many uses of hash functions beyond simple hash tables, however, `Object.hashCode` almost always falls short -- hence `com.google.common.hash`.

Organization

Looking at the package Javadoc, we see a lot of different types, but it's not obvious how they fit together.

Let's look at a sample piece of code using this library.

```
HashFunction hf = Hashing.md5();
HashCode hc = hf.newHasher()
       .putLong(id)
       .putString(name, Charsets.UTF_8)
       .putObject(person, personFunnel)
       .hash();
```

HashFunction

`HashFunction` is a pure, stateless function that maps an arbitrary block of data to a fixed number of bits, with the property that equal inputs always yield equal outputs, and unequal inputs yield unequal outputs as often as possible.

Hasher

A `HashFunction` can be asked for a stateful `Hasher`, which provides fluent syntax to add data to the hash and then retrieve the hash value. A `Hasher` can accept any primitive input, byte arrays, slices of byte arrays, character sequences, character sequences in some charset, and so on, or any other `Object`, provided with an appropriate `Funnel`.

`Hasher` implements the `PrimitiveSink` interface, which specifies a fluent API for an object that accepts a stream of primitive values.

Funnel

A `Funnel` describes how to decompose a particular object type into primitive field values. For example, if we had

```
class Person {
  final int id;
  final String firstName;
  final String lastName;
```

```
  final int birthYear;
}
```

our `Funnel` might look like

```
Funnel<Person> personFunnel = new Funnel<Person>() {
  @Override
  public void funnel(Person person, PrimitiveSink into) {
    into
        .putInt(person.id)
        .putString(person.firstName, Charsets.UTF_8)
        .putString(person.lastName, Charsets.UTF_8)
        .putInt(birthYear);
  }
};
```

Note: `putString("abc", Charsets.UTF_8).putString("def", Charsets.UTF_8)` is fully equivalent to `putString("ab", Charsets.UTF_8).putString("cdef", Charsets.UTF_8)`, because they produce the same byte sequence. This can cause unintended hash collisions. Adding separators of some kind can help eliminate unintended hash collisions.

HashCode

Once a `Hasher` has been given all its input, its `hash()` method can be used to retrieve a `HashCode`. `HashCode` supports equality testing and such, as well as `asInt()`, `asLong()`, `asBytes()` methods, and additionally, `writeBytesTo(array, offset, maxLength)`, which writes the first `maxLength` bytes of the hash into the array.

BloomFilter

Bloom filters are a lovely application of hashing that cannot be done simply using `Object.hashCode()`. Briefly, Bloom filters are a probabilistic data structure, allowing you to test if an object is *definitely* not in the filter, or was *probably* added to the Bloom filter. The Wikipedia page is fairly comprehensive, and we recommend this tutorial.

Our hashing library has a built-in Bloom filter implementation, which requires only that you implement a `Funnel` to decompose your type into primitive types. You can obtain a fresh `BloomFilter<T>` with `create(Funnel funnel, int expectedInsertions, double falsePositiveProbability)`, or just accept the default false probability of 3%. `BloomFilter<T>` offers the methods `boolean mightContain(T)` and `void put(T)`, which are self-explanatory enough.

```
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 500, 0.01);
for (Person friend : friendsList) {
  friends.put(friend);
}
// much later
if (friends.mightContain(dude)) {
  // the probability that dude reached this place if he isn't a friend is 1%
  // we might, for example, start asynchronously loading things for dude while we do
a more expensive exact check
}
```

Hashing

The `Hashing` utility class provides a number of stock hash functions and utilities to operate on `HashCode` objects.

Provided Hash Functions

- `md5()`
- `murmur3_128()`
- `murmur3_32()`
- `sha1()`
- `sha256()`
- `sha512()`
- `goodFastHash(int bits)`

For a complete list, see the `Hashing` javadocs.

HashCode Operations

Method	Description
`HashCode combineOrdered(Iterable<HashCode>)`	Combines hash codes in an ordered fashion, so that if two hashes obtained from this method are the same, then it is likely that each was computed from the same hashes in the same order.
`HashCode combineUnordered(Iterable<HashCode>)`	Combines hash codes in an unordered fashion, so that if two hashes obtained from this method are the same, then it is likely that each was computed from the same hashes in some order.
`int consistentHash(HashCode, int buckets)`	Assigns the hash code a consistent "bucket" which minimizes the need for remapping as the number of buckets grows. See Wikipedia for details.