Обзор библиотеки Guava

Юрий Литвинов
yurii.litvinov@gmail.com

17.04.2019г

Guava

Guava[1] — одна из самых известных библиотек для Java

- Общего назначения
  - Делает программирование на Java приятнее в мелочах
- Новые коллекции, утилиты для ввода-вывода и многопоточности, хеширования, работы со строками и т.д. и т.п.
- Идеологически несколько отличается от JDK
  - Не гарантирует обратную совместимость
  - Не любит null

[1] https://github.com/google/guava/wiki

Preconditions

Preconditions — что-то вроде продвинутого assert
Пример:

```
checkArgument(i >= 0, "Argument was %s but expected nonnegative", i);
checkArgument(i < j, "Expected i < j, but %s > %s", i, j);
```

Вместо

```
if (i >= 0) {
    throw new IllegalArgimentException(
          "Argument was " + i + " but expected nonnegative");
}
```

Что ещё бывает

Сигнатура	Описание	Бросаемое исключение
checkArgument(boolean)	Проверить условие на аргумент	IllegalArgumentException
checkNotNull(T)	Проверить на не *null*	NullPointerException
checkState(boolean)	Проверить состояние объекта безотносительно параметров (например, состояние итератора)	IllegalStateException
checkElementIndex(int index, int size)	Проверить, что *index* от 0 до $size - 1$	IndexOutOfBoundsException
checkPositionIndex(int index, int size)	Проверить, что *index* от 0 до *size*	IndexOutOfBoundsException
checkPositionIndexes(int start, int end, int size)	Проверить, что переданный полуинтервал является валидным подынтервалом для списка заданного размера	IndexOutOfBoundsException

Objects

Класс **Objects** не так полезен с появлением Java 7, но есть ещё **MoreObjects**. Пример:

```
// Returns "ClassName{x=1}"
MoreObjects.toStringHelper(this)
    .add("x", 1)
    .toString();

// Returns "MyObject{x=1}"
MoreObjects.toStringHelper("MyObject")
    .add("x", 1)
    .toString();
```

ComparisonChain

Класс **ComparisonChain** нужен для быстрой реализации compareTo(). Пример:

```
public int compareTo(Foo that) {
  return ComparisonChain.start()
      .compare(this.aString, that.aString)
      .compare(this.anInt, that.anInt)
      .compare(this.anEnum, that.anEnum, Ordering.natural().nullsLast())
      .result();
}
```

Класс **Ordering** нужен для быстрой реализации Comparator.

Throwables

Throwables делает работу с исключениями несколько более удобной. Пример:

```
try {
  someMethodThatCouldThrowAnything();
} catch (IKnowWhatToDoWithThisException e) {
  handle(e);
} catch (Throwable t) {
  Throwables.propagateIfPossible(t, OtherException.class);
  throw new RuntimeException("unexpected", t);
}
```

Ещё есть List<Throwable> getCausalChain(Throwable throwable),

Throwable getRootCause(Throwable throwable),

void propagateIfPossible(Throwable throwable, Class<X> declaredType),

void throwIfInstanceOf(Throwable throwable, Class<X> declaredType)

Немутабельные коллекции

Для каждой коллекции из стандартной библиотеки и каждой коллекции из Guava есть Immutable-вариант, например, **ImmutableMap**, **ImmutableList** и т.д. Зачем:

- многопоточность
- эффективность
- хороший стиль

Collections.unmodifiable... из JDK не совсем немутабельны.
Immutable*-коллекция никогда не меняет своих элементов.

Как создать немутабельную коллекцию

- copyOf, например, ImmutableSet.copyOf(set)
- of, например, ImmutableSet.of("a", "b", "c") или ImmutableMap.of("a", 1, "b", 2)
- метод asList() у немутабельных коллекций, который возвращает ImmutableList.

copyOf не копирует коллекцию, если это не приведёт к проблемам

Multiset

Multiset — мутабельное мультимножество: неупорядоченный ArrayList<E> или Map<E, Integer>

Метод	Описание
count(E)	Возвращает количество вхождений элемента
elementSet()	Возвращает множество (нормальное) различных элементов (на самом деле, view)
entrySet()	Возвращает множество объектов Multiset.Entry<E>, у которых можно узнать элемент и число вхождений
add(E, int)	Добавляет указанное количество вхождений указанного элемента
remove(E, int)	Удаляет указанное количество вхождений указанного элемента
setCount(E, int)	Устанавливает количество вхождений заданного элемента в заданное число
size()	Возвращает общее количество элементов в мультимножестве

Multiset

Multiset — не просто Map:

- количество вхождений элемента не может быть отрицательным
- установление количества вхождений элемента в 0 равносильно удалению всех его вхождений
- multiset.count(elem) для элемента, не принадлежащего мультимножеству, вернёт 0

Реализации (примерно соответствуют стандартным реализациям Map): *HashMultiset*, *TreeMultiset*, *LinkedHashMultiset*, *ConcurrentHashMultiset*, *ImmutableMultiset*

Multimap

Multimap — можно понимать как Map с неуникальными ключами или как Map, отображающий ключ в список значений

- Первая интерпретация «из коробки», вторая — методом asMap(), который возвращает Map<K, Collection<V>>
- Ключ всегда отображается в хотя бы оно значение
- Интерфейсы-наследники **ListMultimap** и **SetMultimap**

Реализации: *ArrayListMultimap*, *HashMultimap*, *LinkedListMultimap*, *LinkedHashMultimap*, *TreeMultimap*, *ImmutableListMultimap*, *ImmutableSetMultimap*

Multimap, пример

```
Set<Person> aliceChildren = childrenMultimap.get(alice);
aliceChildren.clear();
aliceChildren.add(bob);
aliceChildren.add(carol);
```

Результат get() — это вид на коллекцию, так что childrenMultimap тоже изменится. Кстати, get(key) всегда возвращает коллекцию, возможно, пустую.

BiMap

BiMap — отображает ключи в значения и обратно

- И ключи, и значения должны быть уникальны
- Затирать старое значение можно методом BiMap.forcePut(key, value)
- Метод inverse() возвращает обратный BiMap

Реализации: *HashBiMap*, *ImmutableBiMap*, *EnumBiMap*, *EnumHashBiMap*

Table

Table — двумерная таблица, Map<R, Map<C, V>>

```
Table<DateOfBirth, LastName, PersonalRecord> records
        = HashBasedTable.create();
records.put(someBirthday, "Schmo", recordA);
records.put(someBirthday, "Doe", recordB);
records.put(otherBirthday, "Doe", recordC);

// Возвращает Map, отображающий "Schmo" в recordA,
// "Doe" в recordB
records.row(someBirthday);
// Возвращает Map, отображающий someBirthday в recordB,
// otherBirthday в recordC
records.column("Doe");
```

Реализации: *HashBasedTable*, *TreeBasedTable*, *ImmutableTable*, *ArrayTable*

ClassToInstanceMap

ClassToInstanceMap — отображение из класса в объект (Map<Class<? **extends** B>, B>)

```
ClassToInstanceMap<Number> numberDefaults
        = MutableClassToInstanceMap.create();
numberDefaults.putInstance(Integer.class, Integer.valueOf(0));
```

Реализации: *MutableClassToInstanceMap* и *ImmutableClassToInstanceMap*

RangeSet

RangeSet — это набор промежутков: отрезков, интервалов или полуинтервалов:

```
RangeSet<Integer> rangeSet = TreeRangeSet.create();
rangeSet.add(Range.closed(1, 10)); // {[1, 10]}
rangeSet.add(Range.closedOpen(11, 15)); // {[1, 10], [11, 15)}
rangeSet.add(Range.closedOpen(15, 20)); // {[1, 10], [11, 20)}
rangeSet.add(Range.openClosed(0, 0)); // {[1, 10], [11, 20)}
rangeSet.remove(Range.open(5, 10)); // {[1, 5], [10, 10], [11, 20)}
```

Умеет: автоматически объединять перекрывающие друг друга интервалы, делать дополнение к набору промежутков, пересечение с заданным промежутком, проверять на принадлежность точке набору промежутков, считать покрытие и т.д.

RangeMap

RangeMap отображает промежутки в некоторые значения:

```
RangeMap<Integer, String> rangeMap = TreeRangeMap.create();
rangeMap.put(Range.closed(1, 10), "foo"); // {[1, 10] => "foo"}

// {[1, 3] => "foo", (3, 6) => "bar", [6, 10] => "foo"}
rangeMap.put(Range.open(3, 6), "bar");
```

Классы-утилиты

Для всех новых коллекций и многих коллекций из JDK есть классы-утилиты (для *Multiset* есть класс *Multisets*, для *Table* — *Tables*, для *List* — *Lists* и т.д.)

- Не так полезны с выходом Java 8
- В отличие от JDK, в основном ленивы
- Могут быть короче и аккуратнее, чем стандартные

Классы-утилиты, пример 1

```
Set<String> wordsWithPrimeLength
        = ImmutableSet.of("one", "two", "three"
                        , "six", "seven", "eight");
Set<String> primes = ImmutableSet.of(
        "two", "three", "five", "seven");

SetView<String> intersection
        = Sets.intersection(primes, wordsWithPrimeLength);
return intersection.immutableCopy();
```

Классы-утилиты, пример 2

```
Set<String> animals = ImmutableSet.of("gerbil", "hamster");
Set<String> fruits = ImmutableSet.of("apple", "orange", "banana");

Set<List<String>> product = Sets.cartesianProduct(animals, fruits);
// {{"gerbil", "apple"}, {"gerbil", "orange"}, {"gerbil", "banana"},
//  {"hamster", "apple"}, {"hamster", "orange"}, {"hamster", "banana"}}

Set<Set<String>> animalSets = Sets.powerSet(animals);
// {{}, {"gerbil"}, {"hamster"}, {"gerbil", "hamster"}}
```

Классы-утилиты, пример 3

```
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
Map<String, Integer> right = ImmutableMap.of("b", 2, "c", 4, "d", 5);
MapDifference<String, Integer> diff = Maps.difference(left, right);

diff.entriesInCommon();  // {"b" => 2}
diff.entriesDiffering();  // {"c" => (3, 4)}
diff.entriesOnlyOnLeft();  // {"a" => 1}
diff.entriesOnlyOnRight();  // {"d" => 5}
```

Средства быстрого создания коллекций

Forwarding Decorators — заготовки для реализации стандартных интерфейсов, как **Abstract*** из JDK

- Вместо наследования используется композиция
- Наследуемся от Forwarding Decorator-а из Guava, переопределяем метод delegate()
- delegate() должен возвращать «декорируемую» коллекцию, которой декоратор будет перенаправлять все непереопределённые запросы
- Позволяет динамически менять декорируемые объекты и даже выстраивать цепочки «декораторов»

Пример

```
class AddLoggingList<E> extends ForwardingList<E> {
  final List<E> delegate; // декорируемый список
  @Override protected List<E> delegate() {
    return delegate;
  }
  @Override public void add(int index, E elem) {
    log(index, elem);
    super.add(index, elem);
  }
  @Override public boolean add(E elem) {
    return standardAdd(elem);  // реализуется в терминах add(int, E)
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    return standardAddAll(c); // реализуется в терминах add
  }
}
```

Средства быстрого создания итераторов

PeekingIterator — обёртка для итераторов, предоставляющая метод peek(). **AbstractIterator** и **AbstractSequentialIterator** позволяют быстро реализовать итератор:

```
Iterator<Integer> powersOfTwo
      = new AbstractSequentialIterator<Integer>(1) {
   protected Integer computeNext(Integer previous) {
      return (previous == 1 << 30) ? null : previous * 2;
   }
};
```

Графы

Графы в Guava бывают трёх видов:

- **Graph** — вершины и непомеченные рёбра
- **ValueGraph** — наследник Graph, добавляющий метки на рёбрах
- **Network** — граф, в котором и вершины, и рёбра имеют собственную идентичность
  - outEdges(node)
  - incidentNodes(edge)
  - edgesConnecting(nodeU, nodeV)
  - ...
  - asGraph()

Создание графов

Классы GraphBuilder, ValueGraphBuilder и NetworkBuilder (паттерн «Строитель»)

```
MutableGraph<Integer> graph = GraphBuilder.undirected().build();

MutableValueGraph<City, Distance> roads
    = ValueGraphBuilder.directed().build();

MutableNetwork<Webpage, Link> webSnapshot
    = NetworkBuilder.directed()
      .allowsParallelEdges(true)
      .nodeOrder(ElementOrder.natural())
      .expectedNodeCount(100000)
      .expectedEdgeCount(1000000)
      .build();
```

Замечания

- Почти все конкретные классы графов не экспортируются
- *Graph*, *ValueGraph* и *Network* не предоставляют изменяющих граф методов, но есть их наследники *MutableGraph*, *MutableValueGraph* и *MutableNetwork*
- *ImmutableGraph*, *ImmutableValueGraph* и *ImmutableNetwork* гарантируют немутабельность графа
- Метод copyOf() выполняет мелкое (shallow) копирование
- Класс **Graphs** — набор статических методов
  - hasCycle(Graph<?> graph)
  - inducedSubgraph(Graph<N> graph, Iterable<? **extends** N> nodes)
  - reachableNodes(Graph<N> graph, Object node)
  - transitiveClosure(Graph<N> graph)
- Неплохо работает на графах порядка миллионов вершин

Пример 1

```
MutableValueGraph<Integer, Double> weightedGraph
    = ValueGraphBuilder.directed().build();

weightedGraph.addNode(1);
weightedGraph.putEdgeValue(2, 3, 1.5);  // добавляет 2 и 3
weightedGraph.putEdgeValue(3, 5, 1.5);
...
weightedGraph.putEdgeValue(2, 3, 2.0);  // обновляет вес (2,3)
```

Пример 2

```
MutableNetwork<Integer, String> network
    = NetworkBuilder.directed().build();

network.addNode(1);
network.addEdge("2->3", 2, 3);  // добавляет 2 и 3

Set<Integer> successorsOfTwo = network.successors(2);
Set<String> outEdgesOfTwo = network.outEdges(2);

network.addEdge("2->3 too", 2, 3);  // исключение
network.addEdge("2->3", 2, 3);  // ничего не делает

Set<String> inEdgesOfFour = network.inEdges(4); // исключение
```

Пример работы с кешем

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .maximumSize(1000)
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) throws AnyException {
               return createExpensiveGraph(key);
             }
           });


...
try {
   return graphs.get(key);
} catch (ExecutionException e) {
   throw new OtherException(e.getCause());
}
```

Вариант без CacheLoader-a

```
Cache<Key, Value> cache = CacheBuilder.newBuilder()
    .maximumSize(1000)
    .build();
...
try {
    cache.get(key, () -> doThingsTheHardWay(key));
} catch (ExecutionException e) {
    throw new OtherException(e.getCause());
}
```

Выталкивание значений из кеша

- Чистка происходит только при записи и иногда при чтении
- CacheBuilder.expireAfterAccess(**long**, TimeUnit)
- CacheBuilder.expireAfterWrite(**long**, TimeUnit)
  - CacheBuilder.ticker(Ticker)
- CacheBuilder.maximumWeight(**long**) и интерфейс Weigher
- Cache.invalidate(key) и Cache.invalidateAll()
- Weak references
- CacheBuilder.removalListener(RemovalListener)
- CacheBuilder.recordStats() и Cache.stats()

Функциональные идиомы

С Java 8 не так актуально, но всё равно стоит посмотреть

- Function<A, B>
  - forMap(Map<A, B>), compose(Function<B, C>, Function<A, B>), constant(T), identity()
- Predicate<T>
  - instanceOf(Class), assignableFrom(Class), contains(Pattern), in(Collection), isNull(), alwaysFalse(), ...

ListenableFuture

- **ListenableFuture** — наследник обычного *Future*, который добавляет метод addListener(Runnable, Executor), что делает возможным много чего, например, связывать асинхронные операции в цепочки
- **FutureCallback<V>** — позволяет подписаться на успешное и неудачное выполнение операции
- **Futures**
  - addCallback(ListenableFuture<V>, FutureCallback<V>, Executor)
  - transformAsync(ListenableFuture<A>, AsyncFunction<A, B>, Executor)
  - allAsList(Iterable<ListenableFuture<V>>)
  - successfulAsList(Iterable<ListenableFuture<V>>)

Пример

```
ListeningExecutorService service
        = MoreExecutors.listeningDecorator(
           Executors.newFixedThreadPool(10));
ListenableFuture<Explosion> explosion
        = service.submit(() -> pushBigRedButton());
Futures.addCallback(explosion, new FutureCallback<Explosion>() {
  public void onSuccess(Explosion explosion) {
    walkAwayFrom(explosion);
  }
  public void onFailure(Throwable thrown) {
    battleArchNemesis();
  }
});
```

Service

Service — абстракция сервиса, который можно запустить и остановить, следит за своим состоянием

- Service.State.NEW
- Service.State.STARTING
- Service.State.RUNNING
- Service.State.STOPPING
- Service.State.TERMINATED

ServiceManager — класс, управляющий несколькими сервисами (startAsync(), stopAsync(), awaitHealthy(), awaitStopped(), ...)

Joiner и Splitter

Похожая функциональность есть в стандартной библиотеке, но классы из Guava немного удобнее

- Joiner

```
Joiner joiner = Joiner.on("; ").skipNulls();
return joiner.join("Harry", null, "Ron", "Hermione");
```

- Splitter

```
Splitter.on(',')
    .trimResults()
    .omitEmptyStrings()
    .split("foo,bar,,   qux");
```

CharMatcher

```
String noControl = CharMatcher.javaIsoControl().removeFrom(string);
String theDigits = CharMatcher.digit().retainFrom(string);
String spaced = CharMatcher.whitespace()
        .trimAndCollapseFrom(string, ' ');
String noDigits = CharMatcher.javaDigit().replaceFrom(string, "*");
String lowerAndDigit = CharMatcher
        .javaDigit()
        .or(CharMatcher.javaLowerCase())
        .retainFrom(string);
```

CaseFormat

```
CaseFormat.UPPER_UNDERSCORE.to(
      CaseFormat.LOWER_CAMEL, "CONSTANT_NAME"));
// "constantName"
```

Умеет:

- LOWER_CAMEL (lowerCamel)
- LOWER_HYPHEN (lower-hyphen)
- LOWER_UNDERSCORE (lower_underscore)
- UPPER_CAMEL (UpperCamel)
- UPPER_UNDERSCORE (UPPER_UNDERSCORE)

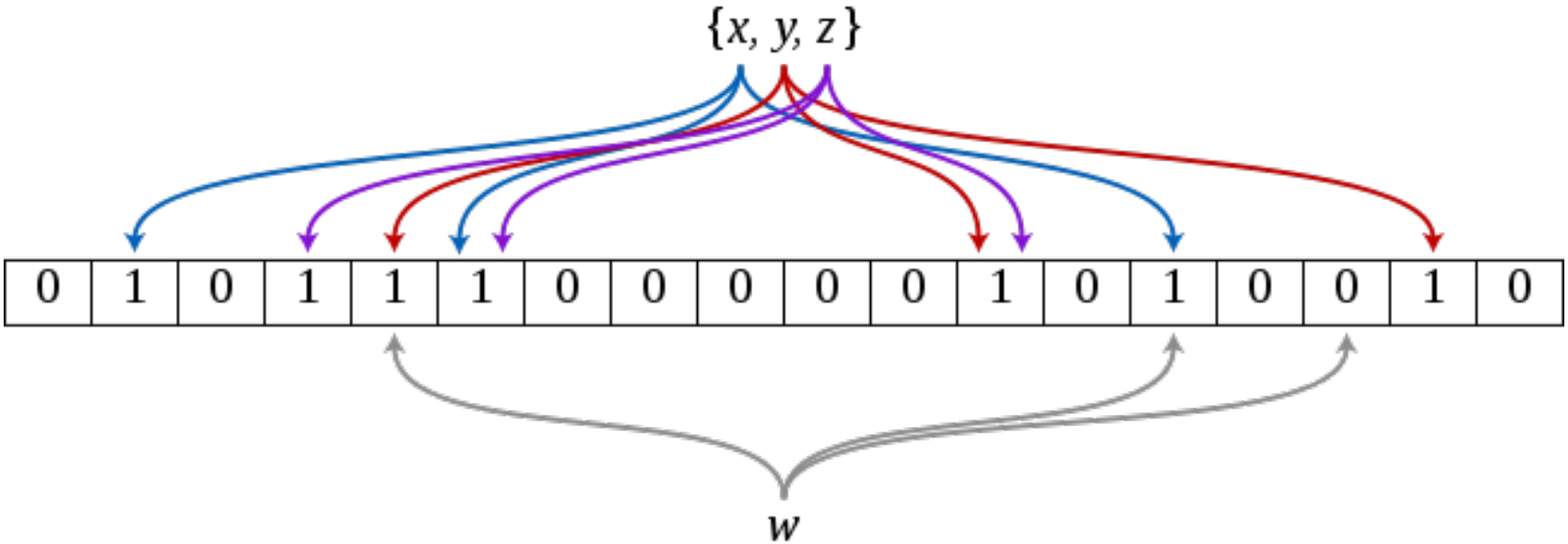
Event bus

- Подсистема для связи компонентов программы так, чтобы им не надо было даже знать друг о друге
- Publisher-Subscriber
- Для замены Listener-ов из JDK
  - Не предназначена для межпроцессного взаимодействия
  - Не может сложную маршрутизацию, фильтры и т.д.
- Использует аннотацию @Subscribe для методов-подписчиков

Пример

```
class EventBusChangeRecorder {
  @Subscribe public void recordCustomerChange(ChangeEvent e) {
    recordChange(e.getChange());
  }
}
...
eventBus.register(new EventBusChangeRecorder());
...
public void changeCustomer() {
  ChangeEvent event = getChangeEvent();
  eventBus.post(event);
}
```

Фильтр Блума



Структура данных, умеющая очень эффективно определять, что элемента в множестве точно нет или с некоторой (задаваемой) вероятностью есть.

Пример

```
BloomFilter<Person> friends
        = BloomFilter.create(personFunnel, 500, 0.01);
for(Person friend : friendsList) {
  friends.put(friend);
}
...
if (friends.mightContain(dude)) {
  // Вероятность того, что dude не друг, составляет здесь 1%
  ...
}
```

Что ещё есть

- Работа с примитивными типами, поддержка беззнаковых целых
- Удобные утилиты ввода-вывода
- Разные алгоритмы вычисления хеш-функции (SHA-1, MD5, CRC32, ...)
- Продвинутая библиотека математических функций
- Полезные классы для работы с рефлексией

https://github.com/google/guava/wiki