# Guava(瓜娃)学习笔记

Guava工程包含了若干被google的java项目广泛依赖的核心库，例如：集合 [collections] 、缓存 [caching] 、原生类型支持 [primitives support] 、并发库 [concurrency libraries] 、通用注解 [common annotations] 、字符串处理 [string processing] 、I/O 等等。 所有这些工具每天都在被Google的工程师应用在产品服务中。

这些高质量的API可以使你的java代码更加优雅，更加简洁，让你的工作更加轻松愉悦，下面我们来开启Java编程学习之旅。

源码包的简单说明：

com.google.common.annotations：普通注解类型。

com.google.common.base：基本工具类库和接口。

com.google.common.cache：缓存工具包，非常简单易用且功能强大的JVM内缓存。

com.google.common.collect：带泛型的集合接口扩展和实现，以及工具类，这里你会发现很多好玩的集合。

com.google.common.eventbus：发布订阅风格的事件总线。

com.google.common.hash： 哈希工具包。

com.google.common.io：I/O工具包。

com.google.common.math：原始算术类型和超大数的运算工具包。

com.google.common.net：网络工具包。

com.google.common.primitives：八种原始类型和无符号类型的静态工具包。

com.google.common.reflect：反射工具包。

com.google.common.util.concurrent：多线程工具包。

## 基本工具（Basic utilities）

### 使用和避免null（Optional）

null会引起歧义，会造成让人迷惑的错误，有时也会让人感到不爽。Guava中的许多工具遇到null时，会拒绝或者马上报错，而不是盲目的接受。

鉴于此google的guava库中提供了Optional接口来使null快速失败，即在可能为null的对象上做了一层封装，在使用Optional静态方法of时，如果传入的参数为null就抛出NullPointerException异常。

在Guava中Optional类就是用来强制提醒程序员，注意对Null的判断。

Optional的另外几个方法

Optional<T>.of(T) 为Optional赋值，当T为Null直接抛NullPointException,建议这个方法在调用的时候直接传常量，不要传变量

Optional<T>.fromNullable(T) 为Optional赋值，当T为Null则使用默认值。建议与or方法一起用，风骚无比

Optional<T>.absent() 为Optional赋值，采用默认值

T or(T) 当Optional的值为null时，使用or赋予的值返回。与fromNullable是一对好基友

T get() 当Optional的值为null时，抛出IllegalStateException，返回Optional的值

boolean isPresent() 如果Optional存在值，则返回True

T orNull() 当Optional的值为null时，则返回Null。否则返回Optional的值

Set<T> asSet() 将Optional中的值转为一个Set返回，当然只有一个值啦，或者为空，当值为null时。

使用Optional的意义？

使用Optional除了赋予null语义，增加了可读性，最大的优点在于它一种傻瓜式的防护，Optional迫使你积极思考引用缺失的情况，因为你必须显式地从Optional获取引用。直接使用null很容易让人忘掉某些情形，尽管FindBugs可以帮助查找null相关的问题，但是我们还是认为它并不能准确地定位问题根源。

### 前提条件（Preconditions）

使方法的条件检查更简单。

Guava在Preconditions类中提供了若干前置条件判断的实用方法，我们强烈建议在Eclipse中静态导入这些方法。每个方法都有三个变种：

1. 没有额外参数：抛出的异常中没有错误消息；
2. 有一个Object对象作为额外参数：抛出的异常使用Object.toString() 作为错误消息；
3. 有一个String对象作为额外参数，并且有一组任意数量的附加Object对象：这个变种处理异常消息的方式有点类似printf，但考虑GWT的兼容性和效率，只支持%s指示符。

例如：查看源代码打印帮助

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

checkArgument(i < j, "Expected i < j, but %s > %s", i, j);

|  |  |  |
| --- | --- | --- |
| 方法声明（不包括额外参数） | 描述 | 检查失败时抛出的异常 |
| checkArgument(boolean) | 检查boolean是否为true，用来检查传递给方法的参数。 | IllegalArgumentException |
| [checkNotNull(T)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Preconditions.html#checkNotNull(T)) | 检查value是否为null，该方法直接返回value，因此可以内嵌使用checkNotNull。 | NullPointerException |
| [checkState(boolean)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Preconditions.html#checkState(boolean)) | 用来检查对象的某些状态。 | IllegalStateException |
| [checkElementIndex(int index, int size)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Preconditions.html#checkElementIndex(int, int)) | 检查index作为索引值对某个列表、字符串或数组是否有效。index>=0 && index<size \* | IndexOutOfBoundsException |
| [checkPositionIndex(int index, int size)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Preconditions.html#checkPositionIndex(int, int)) | 检查index作为位置值对某个列表、字符串或数组是否有效。index>=0 && index<=size \* | IndexOutOfBoundsException |
| checkPositionIndexes(int start, int end, int size) | 检查[start, end]表示的位置范围对某个列表、字符串或数组是否有效\* | IndexOutOfBoundsException |

### 常见的对象方法（Objects）

简化Object方法实现，如hashCode()和toString()；

1. equals

当一个对象中的字段可以为null时，实现Object.equals方法会很痛苦，因为不得不分别对它们进行null检查。使用Objects.equal帮助你执行null敏感的equals判断，从而避免抛出NullPointerException。

1. hashCode

用对象的所有字段作散列[hash]运算应当更简单。Guava的Objects.hashCode(Object...)会对传入的字段序列计算出合理的、顺序敏感的散列值。你可以使用Objects.hashCode(field1, field2, …, fieldn)来代替手动计算散列值。

1. toString

好的toString方法在调试时是无价之宝，但是编写toString方法有时候却很痛苦。使用 Objects.toStringHelper可以轻松编写有用的toString方法。

### 排序

Guava强大的”流畅风格比较器”,具体到下章会介绍到。

### Throwable类

简化了异常和错误的传播与检查；

guava类库中的Throwables提供了一些异常处理的静态方法，这些方法的从功能上分为两类，一类是帮你抛出异常，另外一类是帮你处理异常。

|  |  |
| --- | --- |
| RuntimeException propagate(Throwable) | 如果Throwable是Error或RuntimeException，直接抛出；否则把Throwable包装成RuntimeException抛出。返回类型是RuntimeException，所以你可以像上面说的那样写成throw Throwables.propagate(t)，Java编译器会意识到这行代码保证抛出异常。 |
| void propagateIfInstanceOf( Throwable, Class<X extends Exception>) throws X | Throwable类型为X才抛出 |
| void propagateIfPossible( Throwable) | Throwable类型为Error或RuntimeException才抛出 |
| void propagateIfPossible( Throwable, Class<X extends Throwable>) throws X | Throwable类型为X, Error或RuntimeException才抛出 |

## 集合（Collections）

介绍guava对jdk集合类的扩展，包括不可变集合，新集合类型: multisets, multimaps, tables, bidirectional maps等，强大的集合工具类: 提供java.util.Collections中没有的集合工具，扩展工具类：让实现和扩展集合类变得更容易，比如创建Collection的装饰器，或实现迭代器

集合API的使用, 可以简化集合的创建和初始化；

guava API 提供了有用的新的集合类型，协同已经存在的java集合工作的很好。

分别是 MultiMap， MultiSet， Table， BiMap， ClassToInstanceMap

### google guava的不可变集合

不可变对象有很多优点：

* 1. 当对象被不可信的库调用时，不可变形式是安全的。
  2. 当不可变对象被对个线程调用时，不存在竞态条件问题；
  3. 不可变集合不需要考虑变化，因此可以节约时间和空间，所有不可变集合都比可变集合形式有更好的内存利用率（分析和测试细节）；
  4. 不可变对象因为有固定不变，可以用作常量来安全使用。

总结：数据不可变；不需要同步逻辑；线程安全；自由共享；容易设计和实现；内存和时间高效

创建对象的不可拷贝是一项很好的防御性编程技巧，Guava为所有JDK标准集合类型和Guava新集合类型都提供了简单易用的不可变版本。

JDK也提供了可以将集合变成不可变的方法，Collections.unmodifiableXXX，但是被认为是不好的。

* 1. 笨重而且累赘：不能舒适地用在所有想做防御性拷贝的场景；
  2. 不安全：要保证没人通过原集合的引用进行修改，返回的集合才是事实上不可变的；
  3. 低效：包装过的集合仍然保有可变集合的开销，比如并发修改的检查、散列表的额外空间，等等。

提示：guava不可变集合的实现都不接受null值，经过对google内部代码的研究发现，google内部只有不超过5%的情况下集合中允许了null值，其他情况下都不允许。如果我们相用null的不可变集合，那我们就用jdk中的集合类进行操作，然后进行集合工具类的处理Collections.unmodifiableXXX。

细节：关联可变集合和不可变集合

|  |  |  |
| --- | --- | --- |
| 可变集合接口 | 属于jdk还是guava | 不可变版本 |
| Collection | JDK | ImmutableCollection |
| List | JDK | ImmutableList |
| Set | JDK | ImmutableSet |
| SortedSet/NavigableSet | JDK | ImmutableSortedSet |
| Map | JDK | ImmutableMap |
| SortedMultiset | Guava | ImmutableSortedMultiset |
| Multimap | Guava | ImmutableMultimap |
| ListMultimap | Guava | ImmutableListMultimap |
| SetMultimap | Guava | ImmutableSetMultimap |
| BiMap | Guava | ImmutableBiMap |
| ClassToInstanceMap | Guava | ImmutableClassToInstanceMap |
| Table | Guava | ImmutableTable |

### google guava集合之Multiset

Multiset看似是一个Set，但是实质上它不是一个Set，它没有继承Set接口，它继承的是Collection<E>接口，你可以向Multiset中添加重复的元素，Multiset会对添加的元素做一个计数。

它本质上是一个Set加一个元素计数器。

显然计数不是问题，Multiset还提供了add和remove的重载方法，可以在add或这remove的同时指定计数的值。

常用实现 Multiset 接口的类有：

HashMultiset: 元素存放于 HashMap

LinkedHashMultiset: 元素存放于 LinkedHashMap，即元素的排列顺序由第一次放入的顺序决定

TreeMultiset:元素被排序存放于TreeMap

EnumMultiset: 元素必须是 enum 类型

ImmutableMultiset: 不可修改的 Mutiset

看到这里你可能已经发现 Guava Collections 都是以 create 或是 of 这样的静态方法来构造对象。这是因为这些集合类大多有多个参数的私有构造方法，由于参数数目很多，客户代码程序员使用起来就很不方便。而且以这种方式可以返回原类型的子类型对象。另外，对于创建范型对象来讲，这种方式更加简洁。

### google guava的BiMap：双向Map

我们知道Map是一种键值对映射，这个映射是键到值的映射，而BiMap首先也是一种Map，他的特别之处在于，既提供键到值的映射，也提供值到键的映射，所以它是双向Map.

BiMap的常用实现有：

HashBiMap: key 集合与 value 集合都有 HashMap 实现

EnumBiMap: key 与 value 都必须是 enum 类型

ImmutableBiMap: 不可修改的 BiMap

### google guava的Multimaps：一键多值的Map

有时候我们需要这样的数据类型Map<String,Collection<String>>，guava中的Multimap就是为了解决这类问题的。

Multimap提供了丰富的实现，所以你可以用它来替代程序里的Map<K, Collection<V>>，具体的实现如下：

|  |  |  |
| --- | --- | --- |
| 实现 | Key实现 | Value实现 |
| ArrayListMultimap | HashMap | ArrayList |
| HashMultimap | HashMap | HashSet |
| LinkedListMultimap | LinkedHashMap | LinkedList |
| LinkedHashMultimap | LinkedHashMap | LinkedHashSet |
| TreeMultimap | TreeMap | TreeSet |
| ImmutableListMultimap | ImmutableMap | ImmutableList |
| ImmutableSetMultimap | ImmutableMap | ImmutableSet |

### google guava集合之Table

在guava库中还提供了一种二维表结构：Table。使用Table可以实现二维矩阵的数据结构，可以是稀溜矩阵。

通常来说，当你想使用多个键做索引的时候，你可能会用类似Map<FirstName, Map<LastName, Person>>的实现，这种方式很丑陋，使用上也不友好。Guava为此提供了新集合类型Table，它有两个支持所有类型的键：”行”和”列”。Table提供多种视图，以便你从各种角度使用它：

rowMap()：用Map<R, Map<C, V>>表现Table<R, C, V>。同样的， rowKeySet()返回”行”的集合Set<R>。

row(r) ：用Map<C, V>返回给定”行”的所有列，对这个map进行的写操作也将写入Table中。

类似的列访问方法：columnMap()、columnKeySet()、column(c)。（基于列的访问会比基于的行访问稍微低效点）

cellSet()：用元素类型为Table.Cell<R, C, V>的Set表现Table<R, C, V>。Cell类似于Map.Entry，但它是用行和列两个键区分的。

### Guava集合：使用Iterators简化Iterator操作

Iterators是Guava中对Iterator迭代器操作的帮助类，这个类提供了很多有用的方法来简化Iterator的操作。

### ClassToInstanceMap可以实现map的key值是多个类型

有的时候，你的map的key并不是一种类型，他们是很多类型，你想通过映射他们得到这种类型，guava提供了ClassToInstanceMap满足了这个目的。

除了继承自Map接口，ClassToInstaceMap提供了方法 T getInstance(Class<T>) 和 T putInstance(Class<T>, T),消除了强制类型转换。

### Ordering犀利的比较器

Ordering是Guava类库提供的一个犀利强大的比较器工具，Guava的Ordering和JDK Comparator相比功能更强。它非常容易扩展，可以轻松构造复杂的comparator，然后用在容器的比较、排序等操作中。

本质上来说，Ordering 实例无非就是一个特殊的Comparator 实例。Ordering只是需要依赖于一个比较器（例如，Collections.max）的方法，并使其可作为实例方法。另外，Ordering提供了链式方法调用和加强现有的比较器。

常见的静态方法：

natural()：使用Comparable类型的自然顺序， 例如：整数从小到大，字符串是按字典顺序;

usingToString() ：使用toString()返回的字符串按字典顺序进行排序；

arbitrary() ：返回一个所有对象的任意顺序， 即compare(a, b) == 0 就是 a == b (identity equality)。 本身的排序是没有任何含义， 但是在VM的生命周期是一个常量。

### ComparisonChain比较

实现一个比较器[Comparator]，或者直接实现Comparable接口有时也伤不起。考虑一下这种情况：

class Student implements Comparable<Student>{

private String lastName;

private String firstName;

private int zipCode;

//jdk

public int compareTo(Student other) {

int cmp = lastName.compareTo(other.lastName);

if (cmp != 0) {

return cmp;

}

cmp = firstName.compareTo(other.firstName);

if (cmp != 0) {

return cmp;

}

return Integer.compare(zipCode, other.zipCode);

}

}

这部分代码太琐碎了，因此很容易搞乱，也很难调试。我们应该能把这种代码变得更优雅，为此，Guava提供了ComparisonChain。

ComparisonChain执行一种懒比较：它执行比较操作直至发现非零的结果，在那之后的比较输入将被忽略。

//guava比较优雅

public int compareTo(Student other) {

return ComparisonChain.start()

.compare(this.lastName, other.lastName)

.compare(this.firstName, other.firstName)

.compare(this.zipCode, other.zipCode, Ordering.natural().nullsLast())

.result();

}

这种Fluent接口风格的可读性更高，发生错误编码的几率更小，并且能避免做不必要的工作。

## 缓存（Caches）

google guava框架提供了内存缓存的功能，可以很方便的缓存对象，设置生命周期, 及缓存对象的弱引用 强应用 软引用等

### [使用google guava做内存缓存](http://outofmemory.cn/java/guava/cache/how-to-use-guava-cache)

google guava中有cache包，此包提供内存缓存功能。内存缓存需要考虑很多问题，包括并发问题，缓存失效机制，内存不够用时缓存释放，缓存的命中率，缓存的移除等等。 当然这些东西guava都考虑到了。

guava的内存缓存非常强大，可以设置各种选项，而且很轻量，使用方便。另外还提供了下面一些方法，来方便各种需要：

ImmutableMap<K, V> getAllPresent(Iterable<?> keys) 一次获得多个键的缓存值

put和putAll方法向缓存中添加一个或者多个缓存项

invalidate 和 invalidateAll方法从缓存中移除缓存项

asMap()方法获得缓存数据的ConcurrentMap<K, V>快照

cleanUp()清空缓存

refresh(Key) 刷新缓存，即重新取缓存数据，更新缓存

### google guava缓存分析

guava缓存过期时间分为两种，一种是从写入时开始计时，一种是从最后访问时间开始计时，而且guava缓存的过期时间是设置到整个一组缓存上的；这和EHCache，redis，memcached等不同，这些缓存系统设置都将缓存时间设置到了单个缓存上。

guava缓存设计成了一组对象一个缓存实例，这样做的好处是一组对象设置一组缓存策略，你可以根据不同的业务来设置不同的缓存策略，包括弱引用，软引用，过期时间，最大项数等。另外一点好处是你可以根据不同的组来统计缓存的命中率，这样更有意义一些。

这样做也是有缺点的，缺点是首先是每个缓存组都需要声明不同的缓存实例，具体到业务程序中可能就是每个业务对象一个缓存了。这样就把不同的业务缓存分散到不同的业务系统中了，不太好管理。

## 函数式风格（Functional idioms）

## 并发（Concurrency）

并发编程是一个难题，但是一个强大而简单的抽象可以显著的简化并发的编写。出于这样的考虑，Guava 定义了 ListenableFuture接口并继承了JDK concurrent包下的Future 接口。

### Guava并发：ListenableFuture使用介绍以及示例

ListenableFuture顾名思义就是可以监听的Future，它是对java原生Future的扩展增强，本文介绍ListenableFuture的用法和扩展实现

ListenableFuture顾名思义就是可以监听的Future，它是对java原生Future的扩展增强。我们知道Future表示一个异步计算任务，当任务完成时可以得到计算结果。如果我们希望一旦计算完成就拿到结果展示给用户或者做另外的计算，就必须使用另一个线程不断的查询计算状态。这样做，代码复杂，而且效率低下。使用ListenableFuture Guava帮我们检测Future是否完成了，如果完成就自动调用回调函数，这样可以减少并发程序的复杂度。

另外ListenableFuture还有其他几种内置实现：

SettableFuture：不需要实现一个方法来计算返回值，而只需要返回一个固定值来做为返回值，可以通过程序设置此Future的返回值或者异常信息

CheckedFuture： 这是一个继承自ListenableFuture接口，他提供了checkedGet()方法，此方法在Future执行发生异常时，可以抛出指定类型的异常。

### Guava并发：RateLimiter限制资源的并发访问线程数

RateLimiter类似于JDK的信号量Semphore，他用来限制对资源并发访问的线程数

RateLimiter类似于JDK的信号量Semphore，他用来限制对资源并发访问的线程数。

RateLimiter limiter = RateLimiter.create(4.0); //每秒不超过4个任务被提交

limiter.acquire();  //请求RateLimiter, 超过permits会被阻塞

executor.submit(runnable); //提交任务

也可以以非阻塞的形式来使用：

If(limiter.tryAcquire()){ //未请求到limiter则立即返回false

    doSomething();

}else{

    doSomethingElse();

}

### Guava并发：使用Monitor控制并发

Monitor就像java原生的synchronized, ReentrantLock一样，每次只允许一个线程执行代码块，且可重占用，每一次占用要对应一次退出占用。

/\*\*

 \* 通过Monitor的Guard进行条件阻塞

 \*/

public class MonitorSample {

    private List<String> list = new ArrayList<String>();

    private static final int MAX\_SIZE = 10;

    private Monitor monitor = new Monitor();

    private Monitor.Guard listBelowCapacity = new Monitor.Guard(monitor) {

        @Override

        public boolean isSatisfied() {

            return list.size() < MAX\_SIZE;

        }

    };

    public void addToList(String item) throws InterruptedException {

        monitor.enterWhen(listBelowCapacity); //Guard(形如Condition)，不满足则阻塞，而且我们并没有在Guard进行任何通知操作

        try {

            list.add(item);

        } finally {

            monitor.leave();

        }

    }

}

就如上面，我们通过if条件来判断是否可进入Monitor代码块，并再try/finally中释放：

if (monitor.enterIf(guardCondition)) {

        try {

              doWork();

    } finally {

           monitor.leave();

       }

}

其他的Monitor访问方法：

Monitor.enter //进入Monitor块，将阻塞其他线程直到Monitor.leave

Monitor.tryEnter //尝试进入Monitor块，true表示可以进入, false表示不能，并且不会一直阻塞

Monitor.tryEnterIf //根据条件尝试进入Monitor块

这几个方法都有对应的超时设置版本。

## 字符串处理（Strings）

### 连接器（Joiner）

用分隔符把字符串序列连接起来也可能会遇上不必要的麻烦。如果字符串序列中含有null，那连接操作会更难。Fluent风格的Joiner让连接字符串更简单。

警告：joiner实例总是不可变的。用来定义joiner目标语义的配置方法总会返回一个新的joiner实例。这使得joiner实例都是线程安全的，你可以将其定义为static final常量。

### 拆分器（Splitter）

JDK内建的字符串拆分工具有一些古怪的特性。比如，String.split悄悄丢弃了尾部的分隔符。例如：

”,a,,b,”.split(“,”) //””, “a”, “”, “b” 只有尾部的空字符串被忽略了

Splitter使用令人放心的、直白的流畅API模式对这些混乱的特性作了完全的掌控。

1. 拆分器工厂：

|  |  |
| --- | --- |
| 方法 | 描述 |
| Splitter.on(char) | 按单个字符拆分 |
| Splitter.on(CharMatcher) | 按字符匹配器拆分 |
| Splitter.on(String) | 按字符串拆分 |
| Splitter.on(Pattern)  Splitter.onPattern(String) | 按正则表达式拆分 |
| Splitter.fixedLength(int) | 按固定长度拆分；最后一段可能比给定长度短，但不会为空。 |

1. 拆分器修饰符：

|  |  |
| --- | --- |
| 方法 | 描述 |
| omitEmptyStrings() | 从结果中自动忽略空字符串 |
| trimResults() | 移除结果字符串的前导空白和尾部空白 |
| trimResults(CharMatcher) | 给定匹配器，移除结果字符串的前导匹配字符和尾部匹配字符 |
| limit(int) | 限制拆分出的字符串数量 |

如果你想要拆分器返回List，只要使用Lists.newArrayList(splitter.split(string))或类似方法。 警告：splitter实例总是不可变的。用来定义splitter目标语义的配置方法总会返回一个新的splitter实例。这使得splitter实例都是线程安全的，你可以将其定义为static final常量。

### 字符匹配器（CharMatcher）

然而使用CharMatcher的好处更在于它提供了一系列方法，让你对字符作特定类型的操作：修剪[trim]、折叠[collapse]、移除[remove]、保留[retain]等等。CharMatcher实例首先代表概念1：怎么才算匹配字符？然后它还提供了很多操作概念2：如何处理这些匹配字符？这样的设计使得API复杂度的线性增加可以带来灵活性和功能两方面的增长。

注：CharMatcher只处理char类型代表的字符；它不能理解0x10000到0x10FFFF的Unicode 增补字符。这些逻辑字符以代理对[surrogate pairs]的形式编码进字符串，而CharMatcher只能将这种逻辑字符看待成两个独立的字符。

### 字符集（Charsets）

Charsets针对所有Java平台都要保证支持的六种字符集提供了常量引用。尝试使用这些常量，而不是通过名称获取字符集实例。

### 大小写格式（CaseFormat）

CaseFormat被用来方便地在各种ASCII大小写规范间转换字符串——比如，编程语言的命名规范。CaseFormat支持的格式如下：

|  |  |
| --- | --- |
| 格式 | 范例 |
| LOWER\_CAMEL | lowerCamel |
| LOWER\_HYPHEN | lower-hyphen |
| LOWER\_UNDERSCORE | lower\_underscore |
| UPPER\_CAMEL | UpperCamel |
| UPPER\_UNDERSCORE | UPPER\_UNDERSCORE |

CaseFormat的用法很直接：

CaseFormat.UPPER\_UNDERSCORE.to(CaseFormat.LOWER\_CAMEL, "CONSTANT\_NAME"

)); // returns "constantName"

我们CaseFormat在某些时候尤其有用，比如编写代码生成器的时候。

## 原生类型（Primitives）

Java的原生类型也称原始类型，也是基本数据类型byte、short、int、long、float、double、char和boolean。

在从Guava查找原生类型方法之前，可以先查查Arrays类，或者对应的基础类型包装类，如Integer。

原生类型不能当作对象或泛型的类型参数使用，这意味着许多通用方法都不能应用于它们。Guava提供了若干通用工具，包括原生类型数组与集合API的交互，原生类型和字节数组的相互转换，以及对某些原生类型的无符号形式的支持。

|  |  |
| --- | --- |
| 原生类型 | Guava工具类（都在com.google.common.primitives包） |
| byte | Bytes, SignedBytes, UnsignedBytes |
| short | Shorts |
| int | Ints, UnsignedInteger, UnsignedInts |
| long | Longs, UnsignedLong, UnsignedLongs |
| float | Floats |
| double | Doubles |
| char | Chars |
| boolean | Booleans |

Bytes工具类没有定义任何区分有符号和无符号字节的方法，而是把它们都放到了SignedBytes和UnsignedBytes工具类中，因为字节类型的符号性比起其它类型要略微含糊一些。

int和long的无符号形式方法在UnsignedInts和UnsignedLongs类中，但由于这两个类型的大多数用法都是有符号的，Ints和Longs类按照有符号形式处理方法的输入参数。

此外，Guava为int和long的无符号形式提供了包装类，即UnsignedInteger和UnsignedLong，以帮助你使用类型系统，以极小的性能消耗对有符号和无符号值进行强制转换。

原生类型数组工具:

|  |  |
| --- | --- |
| 方法签名 | 描述 |
| List<Wrapper> asList(prim… backingArray) | 把数组转为相应包装类的List |
| prim[] toArray(Collection<Wrapper> collection) | 把集合拷贝为数组，和collection.toArray()一样线程安全 |
| prim[] concat(prim[]… arrays) | 串联多个原生类型数组 |
| boolean contains(prim[] array, prim target) | 判断原生类型数组是否包含给定值 |
| int indexOf(prim[] array, prim target) | 给定值在数组中首次出现处的索引，若不包含此值返回-1 |
| int lastIndexOf(prim[] array, prim target) | 给定值在数组最后出现的索引，若不包含此值返回-1 |
| prim min(prim… array) | 数组中最小的值 |
| prim max(prim… array) | 数组中最大的值 |
| String join(String separator, prim… array) | 把数组用给定分隔符连接为字符串 |
| Comparator<prim[]> lexicographicalComparator() | 按字典序比较原生类型数组的Comparator |

\*符号无关方法存在于Bytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans。而UnsignedInts, UnsignedLongs, SignedBytes, 或UnsignedBytes不存在。

\*符号相关方法存在于SignedBytes, UnsignedBytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans, UnsignedInts, UnsignedLongs。而Bytes不存在。

通用工具方法：

|  |  |
| --- | --- |
| 方法签名 | 描述 |
| int compare(prim a, prim b) | 传统的Comparator.compare方法，但针对原生类型。JDK7的原生类型包装类也提供这样的方法 |
| prim checkedCast(long value) | 把给定long值转为某一原生类型，若给定值不符合该原生类型，则抛出IllegalArgumentException |
| prim saturatedCast(long value) | 把给定long值转为某一原生类型，若给定值不符合则使用最接近的原生类型值 |

\*这里的整型包括byte, short, int, long。不包括char, boolean, float, 或double。

字节转换方法：

Guava提供了若干方法，用来把原生类型按大字节序与字节数组相互转换。所有这些方法都是符号无关的，此外Booleans没有提供任何下面的方法。

|  |  |
| --- | --- |
| 方法签名或字段签名 | 描述 |
| int BYTES | 常量：表示该原生类型需要的字节数 |
| prim fromByteArray(byte[] bytes) | 使用字节数组的前Prims.BYTES个字节，按大字节序返回原生类型值；如果bytes.length <= Prims.BYTES，抛出IAE |
| prim fromBytes(byte b1, …, byte bk) | 接受Prims.BYTES个字节参数，按大字节序返回原生类型值 |
| byte[] toByteArray(prim value) | 按大字节序返回value的字节数组 |

## 区间（Ranges）

### 简介

区间，有时也称为范围，是特定域中的凸性（非正式说法为连续的或不中断的）部分。在形式上，凸性表示对a<=b<=c, range.contains(a)且range.contains(c)意味着range.contains(b)。

区间可以延伸至无限——例如，范围”x>3″包括任意大于3的值——也可以被限制为有限，如” 2<=x<5″。Guava用更紧凑的方法表示范围，有数学背景的程序员对此是耳熟能详的：

(a..b) = {x | a < x < b}

[a..b] = {x | a <= x <= b}

[a..b) = {x | a <= x < b}

(a..b] = {x | a < x <= b}

(a..+∞) = {x | x > a}

[a..+∞) = {x | x >= a}

(-∞..b) = {x | x < b}

(-∞..b] = {x | x <= b}

(-∞..+∞) = 所有值

上面的a、b称为端点 。为了提高一致性，Guava中的Range要求上端点不能小于下端点。上下端点有可能是相等的，但要求区间是闭区间或半开半闭区间（至少有一个端点是包含在区间中的）：

[a..a]：单元素区间

[a..a); (a..a]：空区间，但它们是有效的

(a..a)：无效区间

Guava用类型Range<C>表示区间。所有区间实现都是不可变类型。

### 构建区间

区间实例可以由Range类的静态方法获取：

|  |  |
| --- | --- |
| (a..b) | open(C, C) |
| [a..b] | closed(C, C) |
| [a..b) | closedOpen(C, C) |
| (a..b] | openClosed(C, C) |
| (a..+∞) | greaterThan(C) |
| [a..+∞) | atLeast(C) |
| (-∞..b) | lessThan(C) |
| (-∞..b] | atMost(C) |
| (-∞..+∞) | all() |

此外，也可以明确地指定边界类型来构造区间：

|  |  |
| --- | --- |
| 有界区间 | range(C, BoundType, C, BoundType) |
| 无上界区间：((a..+∞) 或[a..+∞)) | downTo(C, BoundType) |
| 无下界区间：((-∞..b) 或(-∞..b]) | upTo(C, BoundType) |

这里的BoundType是一个枚举类型，包含CLOSED和OPEN两个值。

### 区间运算

Range的基本运算是它的contains(C) 方法，和你期望的一样，它用来区间判断是否包含某个值。此外，Range实例也可以当作Predicate，并且在函数式编程中使用（译者注：见第4章）。任何Range实例也都支持containsAll(Iterable<? extends C>)方法：

### 查询运算

Range类提供了以下方法来 查看区间的端点：

hasLowerBound()和hasUpperBound()：判断区间是否有特定边界，或是无限的；

lowerBoundType()和upperBoundType()：返回区间边界类型，CLOSED或OPEN；如果区间没有对应的边界，抛出IllegalStateException；

lowerEndpoint()和upperEndpoint()：返回区间的端点值；如果区间没有对应的边界，抛出IllegalStateException；

isEmpty()：判断是否为空区间。

### 关系运算

1. 包含[enclose]

区间之间的最基本关系就是包含[encloses(Range)]：如果内区间的边界没有超出外区间的边界，则外区间包含内区间。包含判断的结果完全取决于区间端点的比较

1. 相连[isConnected]

Range.isConnected(Range)判断区间是否是相连的。具体来说，isConnected测试是否有区间同时包含于这两个区间，这等同于数学上的定义”两个区间的并集是连续集合的形式”（空区间的特殊情况除外）。

1. 交集[intersection]

Range.intersection(Range)返回两个区间的交集：既包含于第一个区间，又包含于另一个区间的最大区间。当且仅当两个区间是相连的，它们才有交集。如果两个区间没有交集，该方法将抛出IllegalArgumentException。

1. 跨区间[span]

Range.span(Range)返回”同时包括两个区间的最小区间”，如果两个区间相连，那就是它们的并集。

span是可互换的[commutative] 、关联的[associative] 、闭合的[closed]运算[operation]。

### 离散域

部分（但不是全部）可比较类型是离散的，即区间的上下边界都是可枚举的。

在Guava中，用DiscreteDomain<C>实现类型C的离散形式操作。一个离散域总是代表某种类型值的全集；它不能代表类似”素数”、”长度为5的字符串”或”午夜的时间戳”这样的局部域。

DiscreteDomain提供的离散域实例包括：

|  |  |
| --- | --- |
| 类型 | 离散域 |
| Integer | integers() |
| Long | longs() |

一旦获取了DiscreteDomain实例，你就可以使用下面的Range运算方法：

ContiguousSet.create(range, domain)：用ImmutableSortedSet<C>形式表示Range<C>中符合离散域定义的元素，并增加一些额外操作——译者注：实际返回ImmutableSortedSet的子类ContiguousSet。（对无限区间不起作用，除非类型C本身是有限的，比如int就是可枚举的）

canonical(domain)：把离散域转为区间的”规范形式”。如果ContiguousSet.create(a, domain).equals(ContiguousSet.create(b, domain))并且!a.isEmpty()，则有a.canonical(domain).equals(b.canonical(domain))。（这并不意味着a.equals(b)）

你可以创建自己的离散域，但必须记住DiscreteDomain契约的几个重要方面。

一个离散域总是代表某种类型值的全集；它不能代表类似”素数”或”长度为5的字符串”这样的局部域。所以举例来说，你无法构造一个DiscreteDomain以表示精确到秒的JODA DateTime日期集合：因为那将无法包含JODA DateTime的所有值。

DiscreteDomain可能是无限的——比如BigInteger DiscreteDomain。这种情况下，你应当用minValue()和maxValue()的默认实现，它们会抛出NoSuchElementException。但Guava禁止把无限区间传入ContiguousSet.create——译者注：那明显得不到一个可枚举的集合。

## I/O

### Guava Files中的文件操作

Java的基本API对文件的操作很繁琐，为了向文件中写入一行文本，都需要写十几行的代码。guava对此作了很多改进，提供了很多方便的操作。

## 散列（Hash）

### 概述

Java内建的散列码[hash code]概念被限制为32位，并且没有分离散列算法和它们所作用的数据，因此很难用备选算法进行替换。此外，使用Java内建方法实现的散列码通常是劣质的，部分是因为它们最终都依赖于JDK类中已有的劣质散列码。

Object.hashCode往往很快，但是在预防碰撞上却很弱，也没有对分散性的预期。这使得它们很适合在散列表中运用，因为额外碰撞只会带来轻微的性能损失，同时差劲的分散性也可以容易地通过再散列来纠正（Java中所有合理的散列表都用了再散列方法）。然而，在简单散列表以外的散列运用中，Object.hashCode几乎总是达不到要求——因此，有了com.google.common.hash包。

### 散列包的组成

在这个包的Java doc中，我们可以看到很多不同的类，但是文档中没有明显地表明它们是怎样 一起配合工作的。

1. HashFunction

HashFunction是一个单纯的（引用透明的）、无状态的方法，它把任意的数据块映射到固定数目的位置，并且保证相同的输入一定产生相同的输出，不同的输入尽可能产生不同的输出。

1. Hasher

HashFunction的实例可以提供有状态的Hasher，Hasher提供了流畅的语法把数据添加到散列运算，然后获取散列值。Hasher可以接受所有原生类型、字节数组、字节数组的片段、字符序列、特定字符集的字符序列等等，或者任何给定了Funnel实现的对象。

Hasher实现了PrimitiveSink接口，这个接口为接受原生类型流的对象定义了fluent风格的API

1. Funnel

Funnel描述了如何把一个具体的对象类型分解为原生字段值，从而写入PrimitiveSink。

注：putString(“abc”, Charsets.UTF\_8).putString(“def”, Charsets.UTF\_8)完全等同于putString(“ab”, Charsets.UTF\_8).putString(“cdef”, Charsets.UTF\_8)，因为它们提供了相同的字节序列。这可能带来预料之外的散列冲突。增加某种形式的分隔符有助于消除散列冲突。

1. HashCode

一旦Hasher被赋予了所有输入，就可以通过hash()方法获取HashCode实例（多次调用hash()方法的结果是不确定的）。HashCode可以通过asInt()、asLong()、asBytes()方法来做相等性检测，此外，writeBytesTo(array, offset, maxLength)把散列值的前maxLength字节写入字节数组。

### 布鲁姆过滤器[BloomFilter]

布鲁姆过滤器是哈希运算的一项优雅运用，它可以简单地基于Object.hashCode()实现。简而言之，布鲁姆过滤器是一种概率数据结构，它允许你检测某个对象是一定不在过滤器中，还是可能已经添加到过滤器了。

Guava散列包有一个内建的布鲁姆过滤器实现，你只要提供Funnel就可以使用它。你可以使用create(Funnel funnel, int expectedInsertions, double falsePositiveProbability)方法获取BloomFilter<T>，缺省误检率[falsePositiveProbability]为3%。BloomFilter<T>提供了boolean mightContain(T) 和void put(T)，它们的含义都不言自明了。

### Hashing类

Hashing类提供了若干散列函数，以及运算HashCode对象的工具方法。

已提供的散列函数

|  |  |  |  |
| --- | --- | --- | --- |
| md5() | murmur3\_128() | murmur3\_32() | sha1() |
| sha256() | sha512() | goodFastHash(int bits) |  |

HashCode运算

|  |  |
| --- | --- |
| 方法 | 描述 |
| HashCode combineOrdered( Iterable<HashCode>) | 以有序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能是顺序相等的 |
| HashCode combineUnordered( Iterable<HashCode>) | 以无序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能在某种排序下是相等的 |
| int consistentHash( HashCode, int buckets) | 为给定的”桶”大小返回一致性哈希值。当”桶”增长时，该方法保证最小程度的一致性哈希值变化。详见一致性哈希。 |

## 事件总线（EventBus）

传统上，Java的进程内事件分发都是通过发布者和订阅者之间的显式注册实现的。设计EventBus就是为了取代这种显示注册方式，使组件间有了更好的解耦。EventBus不是通用型的发布-订阅实现，不适用于进程间通信。

## 数学运算（Math）

## 反射（Reflection）

### guava反射TypeToken解决泛型运行时类型擦除的问题

介绍guava中的TypeToken类解决java 运行时泛型类型擦除问题。

TypeToken的方法列表如下：

|  |  |
| --- | --- |
| 方法 | 描述 |
| getType() | 获得包装的java.lang.reflect.Type. |
| getRawType() | 返回大家熟知的运行时类 |
| getSubtype(Class<?>) | 返回那些有特定原始类的子类型。举个例子，如果这有一个Iterable并且参数是List.class，那么返回将是List。 |
| getSupertype(Class<?>) | 产生这个类型的超类，这个超类是指定的原始类型。举个例子，如果这是一个Set并且参数是Iterable.class，结果将会是Iterable。 |
| isAssignableFrom(type) | 如果这个类型是 assignable from 指定的类型，并且考虑泛型参数，返回true。List<? extends Number>是assignable from List，但List没有. |
| getTypes() | 返回一个Set，包含了这个所有接口，子类和类是这个类型的类。返回的Set同样提供了classes()和interfaces()方法允许你只浏览超类和接口类。 |
| isArray() | 检查某个类型是不是数组，甚至是<? extends A[]>。 |
| getComponentType() | 返回组件类型数组。 |

### guava反射之Invokable使用

Guava的Invokable是对java.lang.reflect.Method和java.lang.reflect.Constructor的流式包装。它简化了常见的反射代码的使用。

一些使用例子：

1. 方法是否是public的?

JDK:

Modifier.isPublic(method.getModifiers());

Invokable:

invokable.isPublic();

1. 方法是否是package private?

JDK:

!(Modifier.isPrivate(method.getModifiers())||Modifier.isPublic(method.getModifiers()))

Invokable:

invokable.isPackagePrivate()

1. 方法是否能够被子类重写？

JDK:

!(Modifier.isFinal(method.getModifiers())

|| Modifiers.isPrivate(method.getModifiers())

|| Modifiers.isStatic(method.getModifiers())

|| Modifiers.isFinal(method.getDeclaringClass().getModifiers()))  
Invokable:

invokable.isOverridable()

1. 方法的第一个参数是否被定义了注解@Nullable？

JDK:

for (Annotation annotation : method.getParameterAnnotations[0]) {

if (annotation instanceof Nullable) {

return true;

}

}

return false;

Invokable:

invokable.getParameters().get(0).isAnnotationPresent(Nullable.class)

1. 构造函数和工厂方法如何共享同样的代码？

你是否很想重复自己，因为你的反射代码需要以相同的方式工作在构造函数和工厂方法中？

Invokable提供了一个抽象的概念。下面的代码适合任何一种方法或构造函数：

invokable.isPublic();

invokable.getParameters();

invokable.invoke(object, args);

List的List.get(int)返回类型是什么？

Invokable提供了与众不同的类型解决方案：

Invokable<List<String>, ?> invokable = new TypeToken<List<String>>() {}.method(getMethod);

invokable.getReturnType(); // String.class

### guava反射：Reflection.newProxy方法简化动态代理

原理上Google Guava的动态代理也是使用JDK的动态代理，这是做了封装，更加简便。另外一点是能够很好的检查需要代理的对象必须拥有接口。使用Class类的isInterface()来做检查。

## 注解（Annotations）

com.google.common.annotations包下注解类的含义和用法：

### Beta

/\*\*

\* 表明一个公用API的未来版本是受不兼容变更或删除限制的

\* 拥有这个注释标志的API不受任何兼容性保证

\*

\*/

@Retention(RetentionPolicy.CLASS)

@Target({

ElementType.ANNOTATION\_TYPE,

ElementType.CONSTRUCTOR,

ElementType.FIELD,

ElementType.METHOD,

ElementType.TYPE})

@Documented

@GwtCompatible

public @interface Beta {}

### GwtCompatible

/\*\*

\* 表明一个类型可能会与 Google Web Toolkit 一起使用.

\* 如果一个方法使用这个注释,说明这个方法的返回值是 GWT 兼容的

\*

\*/

@Retention(RetentionPolicy.CLASS)

@Target({ ElementType.TYPE, ElementType.METHOD })

@Documented

@GwtCompatible

public @interface GwtCompatible {

/\*\*

\* 说明一个类型或者方法的返回值是否支持 GWT 序列化

\*

\*/

boolean serializable() default false;

/\*\*

\* 说明一个类型是否在 GWT 被模拟.

\* 被模拟的源(父源)和JVM的实现不一样

\*

\*/

boolean emulated() default false;

}

### GwtIncompatible

/\*\*

\* 说明一个方法可能无法与 GWT 一起使用

\* 他只能用于被 @GwtCompatible 标志的类的字段,方法和内部类

\*

\*/

@Retention(RetentionPolicy.CLASS)

@Target({

ElementType.TYPE, ElementType.METHOD,

ElementType.CONSTRUCTOR, ElementType.FIELD })

@Documented

@GwtCompatible

public @interface GwtIncompatible {

/\*\*

\* 用于表示不兼容 GWT 的原因

\*

\*/

String value();

}

### VisibleForTesting

/\*\*

\*注释程序元素的存在,或比其他必要广泛可见,仅用于测试代码。

\*/

@GwtCompatible

public @interface VisibleForTesting {

}

## 网络编程（Net）

guava中的net包目前提供的功能较少，而且大多类都标注了@Beta的注解，在guava中标记Beta注解表示这个类还不稳定，有可能在以后的版本中变化，或者去掉，所以不建议大量使用，这里也是只做简单的介绍。

先介绍下唯一一个没有Beta注解的类HttpHeaders，这个类中并没有实质的方法，只是定义了一些Http头名称的常量，通常如果需要我们会自己定义这些常量，如果你引用了guava包，那么就不再建议我们自己定义这些头名称的常量了，直接用它定义的即可。

这里面应该有几乎所有的Http头名称，例如：X\_FORWARDED\_FOR，UPGRADE，REFERER等等。用法也没有必要介绍了，直接引用常量就可以了。

再介绍下一个比较常用的小功能，有时候我们需要在配置文件中配置IP+端口，这时候需要自己写解析ip，端口的方法，guava为我们提供了解析类，我们看下用法实例：

HostAndPort hostAndPort = HostAndPort.fromString("127.0.0.1:8080");

System.out.println("host == " + hostAndPort.getHostText());

System.out.println("port == " + hostAndPort.getPortOrDefault(80));

HostAndPort类的静态方法fromString(String)可以解析出字符串中定义的Host和端口信息。

另外guava包中还提供了InetAddresses类，这个类是InetAddress的帮助类，通过这个类可以方便的从字符串中解析出InetAddress类。但是此类也有@Beta的注解，所以要谨慎使用。