Guava

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。

Guava类比Apache Commons工具集。

官网：https://github.com/google/guava
Google Guava官方教程（中文版）：https://wizardforcel.gitbooks.io/guava-tutorial/content/1.html

使用之前需要引入包

XML 复制代码

```xml
 <!--guava依赖-->
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>31.0.1-jre</version>
</dependency>
```

1.字符串的处理：分割，连接，填充

JDK提供的String还不够好么?

也许还不够友好，至少让我们用起来还不够爽，还得操心！
举个栗子，比如String提供的split方法，我们得关心空字符串吧，还得考虑返回的结果中存在null元素吧，只提供了前后trim的方法（如果我想对中间元素进行trim呢）。

Joiner

Joiner是连接器,对于Joiner，常用的方法是　跳过NULL元素：skipNulls() / 对于NULL元素使用其他替代：useForNull(String)

joiner on就是将list用，连接转成字符串

Java 复制代码

```
@Test
    public void joinerListTest() {
        List<String> lists = Lists.newArrayList("a","b","g","8","9");
        String result = Joiner.on(",").join(lists);
        System.out.println(result);
    }

结果：a,b,g,8,9
```

joiner skipNulls()连接跳过null元素

Java 复制代码

```
@Test
    public void joinerListTest1() {
        List<String> lists = Lists.newArrayList("a","b","g",null,"8","9");
        String result = Joiner.on(",").join(lists);
        System.out.println(result);
    }

结果：a,b,g,null,8,9

@Test
public void joinerListTest2() {
    List<String> lists = Lists.newArrayList("a","b,"g",null,"8","9");
    String result = Joiner.on(",").skipNulls().join(lists);
    System.out.println(result);
}
结果：a,b,g,8,9
```

如果连接的时候list里面含有null值，会报空指针，

joiner useForNull(final String value)用value替换null元素值

```java
@Test
    public void useNullListTest() {
        List<String> lists = Lists.newArrayList("a", "b", "g", null, "8", "9");
        String result = Joiner.on(",").useForNull("哈哈").join(lists);
        System.out.println(result);
    }

结果：a,b,g,哈哈,8,9
```

joiner withKeyValueSeparator(String value) map连接器，keyValueSeparator为key和value之间的分隔符

```java
@Test
    public void withMapTest() {
        Map<Integer, String> maps = Maps.newHashMap();
        maps.put(1, "哈哈");
        maps.put(2, "压压");
        String result = Joiner.on(",").withKeyValueSeparator(":").join(maps);
        System.out.println(result);
        System.out.println(maps);
    }

结果：
1:哈哈,2:压压
{1=哈哈, 2=压压}
```

Splitter

Splitter是分割器,对于Splitter，常用的方法是：trimResults()/omitEmptyStrings()。注意拆分的方式，有字符串，还有正则，还有固定长度分割

splitter on 拆分

Java 复制代码

```java
@Test
    public void splitterListTest() {
        String test = "34344,34,34,哈哈";
        List<String> lists = Splitter.on(",").splitToList(test);
        System.out.println(lists);
    }

结果：[34344, 34, 34, 哈哈]
```

splitter trimResults 拆分去除前后空格

Java 复制代码

```java
@Test
    public void trimResultListTest() {
        String test = "  34344,34,34,哈哈 ";
        List<String> lists =
Splitter.on(",").trimResults().splitToList(test);
        System.out.println(lists);
    }

结果：[34344, 34, 34, 哈哈]
```

splitter omitEmptyStrings 去除拆分出来空的字符串

Java 复制代码

```java
@Test
    public void omitEmptyStringsTest() {
        String test = "  3434,434,34,,哈哈 ";
        List<String> lists =
Splitter.on(",").omitEmptyStrings().splitToList(test);
        System.out.println(lists);
    }

结果：[  3434, 434, 34, 哈哈 ]
```

splitter fixedLength(int lenght) 把字符串按固定长度分割

```java
@Test
    public void fixedLengthTest() {
        String test = "343443434哈哈";
        List<String> lists = Splitter.fixedLength(3).splitToList(test);
        System.out.println(lists);
    }

结果：[343, 443, 434, 哈哈]
```

CharMatcher

CharMatcher 提供了各种方法来处理各种 JAVA char 类型值。

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，例如：修剪[trim]、折叠[collapse]、移除[remove]、保留[retain]等等。

```java
String theDigits = CharMatcher.DIGIT.retainFrom(string); //只保留数字字符
String spaced = CharMatcher.WHITESPACE.trimAndCollapseFrom(string, ' ');
//去除两端的空格，并把中间的连续空格替换成单个空格
String noDigits = CharMatcher.JAVA_DIGIT.replaceFrom(string, "*"); //用*号替换所有数字
String lowerAndDigit = CharMatcher.JAVA_DIGIT.or(CharMatcher.JAVA_LOWER_CASE).retainFrom(string);
// 只保留数字和小写字母
```

获取字符匹配器

CharMatcher中的常量可以满足大多数字符匹配需求：

`ANY`	`NONE`	`WHITESPACE`	`BREAKING_WHITESPACE`
`INVISIBLE`	`DIGIT`	`JAVA_LETTER`	`JAVA_DIGIT`
`JAVA_LETTER_OR_DIGIT`	`JAVA_ISO_CONTROL`	`JAVA_LOWER_CASE`	`JAVA_UPPER_CASE`
`ASCII`	`SINGLE_WIDTH`		

其他获取字符匹配器的常见方法包括：

方法	描述
`anyOf(CharSequence)`	枚举匹配字符。如CharMatcher.anyOf(“aeiou”)匹配小写英语元音
`is(char)`	给定单一字符匹配。
`inRange(char, char)`	给定字符范围匹配，如CharMatcher.inRange(‘a’, ‘z’)

此外，CharMatcher还有`negate()`、`and(CharMatcher)`和`or(CharMatcher)`方法。

使用字符匹配器

CharMatcher提供了多种多样的方法操作CharSequence中的特定字符。其中最常用的罗列如下：

方法	描述
`collapseFrom(CharSequence, char)`	把每组连续的匹配字符替换为特定字符。如WHITESPACE.collapseFrom(string, ‘ ‘)把字符串中的连续空白字符替换为单个空格。
`matchesAllOf(CharSequence)`	测试是否字符序列中的所有字符都匹配。
`removeFrom(CharSequence)`	从字符序列中移除所有匹配字符。
`retainFrom(CharSequence)`	在字符序列中保留匹配字符，移除其他字符。
`trimFrom(CharSequence)`	移除字符序列的前导匹配字符和尾部匹配字符。
`replaceFrom(CharSequence, CharSequence)`	用特定字符序列替代匹配字符。

所有这些方法返回String，除了matchesAllOf返回的是boolean。

collapseFrom 配到的字符做替换

Java 复制代码

```java
 /**
     * collapseFrom 配到的字符做替换
     */
    public void testCollapseFrom() {
        String input = "     Ting Feng        ";
        String result =
CharMatcher.breakingWhitespace().collapseFrom(input, '*');
        System.out.println(result);     // *Ting*Feng*

        result = CharMatcher.is(' ').collapseFrom(input, '-');
        System.out.println(result);     // -Ting-Feng-
    }
```

trimFrom 去空格

Java 复制代码

```java
/**
     * trimFrom 去空格
     * trimLeadingFrom 左边去空格
     * trimTrailingFrom右边去空格
     */
    public void testTrim() {
        System.out.println(CharMatcher.breakingWhitespace().trimFrom("
Ting Feng     "));           // Ting Feng

 System.out.println(CharMatcher.breakingWhitespace().trimLeadingFrom("
Ting Feng     "));    // Ting Feng

 System.out.println(CharMatcher.breakingWhitespace().trimTrailingFrom("
  Ting Feng     "));   //      Ting Feng
    }
```

charMatcher is(Char char）给单一字符匹配

Java 复制代码

```java
 /**
     * is 匹配参数之内的所有字符
     * isNot 匹配参数之外的所有字符
     */
    public void testIs_isNot(){
        String input = "a, c, z, 1, 2";
        System.out.println(CharMatcher.is(',').retainFrom(input));   //
,,,,
        System.out.println(CharMatcher.is(',').removeFrom(input));   // a
c z 1 2

        System.out.println(CharMatcher.isNot(',').retainFrom(input));   //
a c z 1 2
        System.out.println(CharMatcher.isNot(',').removeFrom(input));   //
,,,,
    }
```

charMatcher retainFrom(String s）在字符序列中保留匹配字符，移除其他字符

```java
    /**
     * 匹配任意字符
     */
    public void testAny() {
        String input = "H*el.lo,}12";

        CharMatcher matcher = CharMatcher.any();
        String result = matcher.retainFrom(input);
        System.out.println(result); //  H*el.lo,}12

        matcher = CharMatcher.anyOf("Hel");
        System.out.println(matcher.retainFrom(input)); // Hell
        System.out.println(matcher.removeFrom(input)); // *.o,}12
    }
```

charMatcher matchersAllOf(Char char) 测试是否字符序列所有字符都匹配

```java
/**
     * matchesAllOf 判断sequence所有字符是否都被charMatcher匹配
     * matchesAnyOf 判断sequence中是否存在字符被charMatcher匹配
     * matchesNoneOf 判断sequence所有字符是否都没被charMatcher匹配
     */
    public void test_matchesAllOf_matchesAnyOf_matchesNoneOf(){
        String input = "**e,l.lo,}12";

        CharMatcher matcher = CharMatcher.is(',');
        System.out.println(matcher.matchesAllOf(input));    // false

        matcher = CharMatcher.is(',');
        System.out.println(matcher.matchesAnyOf(input));    // true

        matcher = CharMatcher.is('?');
        System.out.println(matcher.matchesNoneOf(input));   // true
    }
```

2.guava集合

Java 集合类虽然非常强大实用，但是提供功能还是有点薄弱。

举个例子，从输入一个文档中，统计一个关键词出现的次数。

传统的做法是这样的：

```java
String[] words = {"关注","guava集合","影子","大宗师苦荷","影子","guava集合"}
Map<String, Integer> counts = new HashMap<String, Integer>();
for (String word : words) {
    Integer count = counts.get(word);
    if (count == null) {
        counts.put(word, 1);
    } else {
        counts.put(word, count + 1);
    }
}
```

虽然这个需求使用 `Map` 可以轻松搞定，但是这种写法有点笨拙，如果没有判空，将会导致 NPE 异常

Multiset

`Multiset` 继承 JDK `Collection` 接口，我们可以多次增加相同的元素，另外 `Multiset` 最大特定将会为元素计数，我们可以将它类似等同为 `Map<E, Integer>` 。

使用Multiset操作：

```java
String[] words = {"关注","guava集合","影子","大宗师苦荷","影子","guava集合"}
Multiset<String> multiset = HashMultiset.create();
for (String word : words) {
    multiset.add(word);
}
int count = multiset.count("guava集合");
```

使用 `Multiset` 简化了代码，并且再也不用担心新 NPE 的问题。
跟 JDK 集合类一样， `Multiset` 也有许多子类。

Map	Corresponding Multiset	Supports `null` elements
`HashMap`	`HashMultiset`	Yes
`TreeMap`	`TreeMultiset`	Yes
`LinkedHashMap`	`LinkedHashMultiset`	Yes
`ConcurrentHashMap`	`ConcurrentHashMultiset`	No
`ImmutableMap`	`ImmutableMultiset`	No

虽然上面说过我们可以将 `Multiset<E>` 看做 `Map<E, Integer>`,但是 `Multiset` 可不是 `Map` 的子类，它可是 血统纯正的 `Collection` 子类。

Multimap

有时会在业务需求中使用 `Map<String,List<Integer>` 实现下面的需求。

```Java
{name=[Jack, Tom], age=[10, 12]}
2
```

使用 `Multimap` 实现代码如下:

```Java
Multimap<String, String> multimap = HashMultimap.create();
  multimap.put("name", "Jack");
  multimap.put("name", "Tom");
  multimap.put("age", "10");
  multimap.put("age", "12");
  System.out.println(multimap);
  System.out.println(multimap.get("name").size());
```

使用 `Multimap` 子类 `HashMultimap`，其行为类似为 `Map<K,Set<V>>`，也就是说 `Value` 对应的集合内部元素不能重复。如果需要保存的重复的元素我们可以使用 `ArrayListMultimap`。`Multimap` 还有其他子类，如图所示:

Implementation	Keys behave like...	Values behave like..
ArrayListMultimap	HashMap	ArrayList
HashMultimap	HashMap	HashSet
LinkedListMultimap *	LinkedHashMap``*	LinkedList``*
LinkedHashMultimap **	LinkedHashMap	LinkedHashSet
TreeMultimap	TreeMap	TreeSet
ImmutableListMultimap	ImmutableMap	ImmutableList
ImmutableSetMultimap	ImmutableMap	ImmutableSet

BiMap

`BiMap` 可以用来实现键值对的双向映射需求，这样我们就可以通过 `Key` 查找对对应的 `Value`，也可以使用 `Value` 查找对应的 `Key`。
这个需求如果使用 `Map` 实现，我们就不得不使用两个 `Map`，维护双向关系，并且任何改动还要保持同步。

Java 复制代码

```java
Map<String,Integer> nameToId = new HashMap<>();
Map<Integer,String> idToName = new HashMap<>();
nameToId.put("Jack",33);
idToName.put(33,"Jack");
//维护双向关系,每次都需要修改两个map;
```

一般的Map只提供”**键-值**“的映射，而BiMap则同时提供了”**键-值**“和”**值-键**“的映射关系。常用方法：

- put(K key, V value)：添加新的键、值。如果值和已有键重复，会抛出异常
- forcePut(K key, V value)：添加新的键、值。如果值和已有键重复，会覆盖原来的键、值
- inverse()：得到”**值-键**“的BitMap对象

用 `BiMap` 修改上面的代码：

```java
BiMap<String,String> biMap= HashBiMap.create();
  biMap.put("sina","sina.com");
  biMap.put("qq","qq.com");
  biMap.put("sina","sina.cn"); //会覆盖原来的value
  System.out.println(biMap.inverse().get("qq.com"));

  //biMap.put("tecent","qq.com"); //抛出异常
  biMap.forcePut("tecent","qq.com"); //强制替换key
  System.out.println(biMap.get("qq")); //通过value找key
  System.out.println(biMap.inverse().get("qq.com"));
  System.out.println(biMap.inverse().get("sina.com"));
  System.out.println(biMap.inverse().inverse()==biMap);
```

输出：

```
qq
null
tecent
null
true
```

同样的 `BiMap` 也有各种实现类：

Key-Value Map Impl	Value-Key Map Impl	Corresponding `BiMap`
`HashMap`	`HashMap`	`HashBiMap`
`ImmutableMap`	`ImmutableMap`	`ImmutableBiMap`
`EnumMap`	`EnumMap`	`EnumBiMap`
`EnumMap`	`HashMap`	`EnumHashBiMap`

其他扩展集合类

Guava 另外还提供其他集合类，感兴趣同学可以深入了解一下。

- Table
- ClassToInstanceMap

- RangeSet
- RangeMap

集合工具类

除了上面提到的新集合类以外，Guava 提供通用的工具类：

Interface	JDK or Guava?	Corresponding Guava utility class
`Collection`	JDK	`Collections2`
`List`	JDK	`Lists`
`Set`	JDK	`Sets`
`SortedSet`	JDK	`Sets`
`Map`	JDK	`Maps`
`SortedMap`	JDK	`Maps`
`Queue`	JDK	`Queues`
`Multiset`	Guava	`Multisets`
`Multimap`	Guava	`Multimaps`
`BiMap`	Guava	`Maps`
`Table`	Guava	`Tables`

这些工具类需对使用的方法，我们可以快速创建集合，分割集合，转化集合等。
List、Set、Map是我们开发过程中使用频次最高的三种集合类型，今天我们来看一下Guava中对这三种类型的集合提供的工具类

Lists

主要方法有

各种创建list的方法

asList()将数据组转成list

newArrayList()

newArrayListWithCapacity(10) 指定容量的创建

newArrayListWithExpectedSize（20） 初始化指定容量

newCopyOnWriteArrayList()

newLinkedList()

partition(List list, int size) 将list按指定大小分隔成多个list

cartesianProduct(List<? extends B>... lists) 获取多个list的笛卡尔集

charactersOf(String str) 将字符串转成字符集合

reverse(List list) 反转list

transform(List fromList, Function<? super F, ? extends T> function) 数据转换

各种创建list的方法

Java 复制代码

```java
    @Test
    public void ListCreateTest(){
        //将数组转成list,并在开头位置插入元素
        List<String> list = Lists.asList("a",new String[]{"b","c","d"});
        List<String> list1 = Lists.asList("a","b",new String[]{"c","d","e"});

        //直接创建ArrayList
        ArrayList<String> arrayList = Lists.newArrayList();
        //创建ArrayList,并初始化
        ArrayList<String> arrayList1 = Lists.newArrayList("a","b","c");
        //基于现有的arrayList,创建一个arrayList
        ArrayList<String> arrayList2 = Lists.newArrayList(arrayList1);
        //初始化指定容量大小的ArrayList，其中容量指ArrayList底层依赖的数组的length属性值，常用于提前知道ArrayList大小的情况的初始化
        ArrayList<String> arrayList3 = Lists.newArrayListWithCapacity(10);
        //初始化预定容量大小的ArrayList，返回的list的实际容量为5L + estimatedSize + (estimatedSize / 10)，常用于不确定ArrayList大小的情况的初始化
        ArrayList<String> arrayList4 =Lists.newArrayListWithExpectedSize(20);
        //创建CopyOnWriteArrayList
        CopyOnWriteArrayList<String> copyOnWriteArrayList = Lists.newCopyOnWriteArrayList();
        //创建linkedList
        LinkedList<String> linkedList = Lists.newLinkedList();
    }

```

按指定大小分隔list

Java 复制代码

```java
    @Test
    public void partitionTest(){
        List<String> list = Lists.newArrayList("a","b","c","d","e");
        //将list按大小为2分隔成多个list
        List<List<String>> splitList = Lists.partition(list,2);
        System.out.println(splitList);

    }

```

笛卡尔集

Java 复制代码

```java
    @Test
    public void cartesianProcustTest(){
        List<String> list1 = Lists.newArrayList("a","b","c");
        List<String> list2 = Lists.newArrayList("d","e","f");
        List<String> list3 = Lists.newArrayList("1","2","3");
        //获取多个list的笛卡尔集
        List<List<String>> list =
Lists.cartesianProduct(list1,list2,list3);
        System.out.println(list);
    }

```

字符串转成字符集合

Java 复制代码

```java
    @Test
    public void charactersOfTest(){
        //将字符串转成字符集合
        ImmutableList<Character> list = Lists.charactersOf("ababcdfb");
    }

```

list反转

```java
@Test
public void reverseTest(){
    List<String> list = Lists.newArrayList("a","b","c","1","2","3");
    //反转list
    List<String> reverseList = Lists.reverse(list);
    System.out.println(reverseList);
}
```

数据转换

```java
@Test
public void transFormTest(){
    List<String> list = Lists.newArrayList("a","b","c");
    //把list中的每个元素拼接一个1
    List<String> list1 = Lists.transform(list,str -> str + "1");
    System.out.println(list1);
}
```

Sets

主要方法有：

各种创建set的方法
newHashSet()
newLinkedHashSet()
newTreeSet()
newConcurrentHashSet()
cartesianProduct(Set<? extends B>... sets) 笛卡尔集
combinations(Set set, final int size) 按指定大小进行排列组合
difference(final Set set1, final Set<?> set2) 两个集合的差集
intersection(final Set set1, final Set<?> set2) 交集
filter(Set unfiltered, Predicate<? super E> predicate) 过滤
powerSet(Set set) 获取set可分隔成的所有子集
union(final Set<? extends E> set1, final Set<? extends E> set2) 并集

创建各种set的方法

```java
@Test
public void setsCreate(){
    HashSet<String> set = Sets.newHashSet();
    Sets.newLinkedHashSet();
    Sets.newHashSetWithExpectedSize(10);
    Sets.newTreeSet();
    Sets.newConcurrentHashSet();
}
```

笛卡尔集

```java
@Test
public void cartesianProduct(){
    Set<String> set1 = Sets.newHashSet("a","b","c");
    Set<String> set2 = Sets.newHashSet("1","2","3");
    Set<String> set3 = Sets.newHashSet("@","#","&");
    //多个Set的笛卡尔集，参数接收多个set集合
    Set<List<String>> sets = Sets.cartesianProduct(set1,set2,set3);
    System.out.println(sets);

    List<Set<String>> list = Lists.newArrayList(set1,set2,set3);
    //也可以把多个Set集合，放到一个list中，再计算笛卡尔集
    Set<List<String>> sets1 = Sets.cartesianProduct(list);
    System.out.println(sets1);
    //Sets.combinations()
    //Sets.difference()
}
```

按指定大小进行排列组合

Java 复制代码

```java
    @Test
    public void combinationsTest(){
        //将集合中的元素按指定的大小分隔，指定大小的所有组合
        Set<String> set1 = Sets.newHashSet("a","b","c","d");
        Set<Set<String>> sets = Sets.combinations(set1,3);
        for(Set<String> set : sets){
            System.out.println(set);
        }
    }

```

差集

Java 复制代码

```java
    @Test
    public void differenceTest(){
        Set<String> set1 = Sets.newHashSet("a","b","d");
        Set<String> set2 = Sets.newHashSet("d","e","f");
        //difference返回：从set1中剔除两个set公共的元素
        System.out.println(Sets.difference(set1,set2));
    }

```

交集

Java 复制代码

```java
@Test
    public void intersectionTest(){
        Set<String> set1 = Sets.newHashSet("a","b","c");
        Set<String> set2 = Sets.newHashSet("a","b","f");
        //取两个集合的交集
        System.out.println(Sets.intersection(set1,set2));
    }

```

过滤

```java
    @Test
    public void filterTest(){
        Set<String> set1 = Sets.newHashSet("a","b","c");
        //建议可以直接使用java8的过滤，比较方便
        Set<String> set2 = Sets.filter(set1,str ->
str.equalsIgnoreCase("b"));
        System.out.println(set2);
    }

```

所有的排列组合

```java
   @Test
    public void powerSetTest(){
        Set<String> set1 = Sets.newHashSet("a","b","c");
        //获取set可分隔成的所有子集
        Set<Set<String>> allSet = Sets.powerSet(set1);
        for(Set<String> set : allSet){
            System.out.println(set);
        }
    }

```

并集

```java
    @Test
    public void unionTest(){
        Set<String> set1 = Sets.newHashSet("a","b","c");
        Set<String> set2 = Sets.newHashSet("1","2","3");
        //取两个集合的并集
        System.out.println(Sets.union(set1,set2));
    }

```

Maps

主要方法有：

创建各种Map的方法

Maps.newHashMap();
Maps.newConcurrentMap();
Maps.newLinkedHashMap();
Maps.newTreeMap();
asMap(Set set, Function<? super K, V> function) set转map
difference(Map<? extends K, ? extends V> left, Map<? extends K, ? extends V> right) 计算map的差值
filterEntries(Map<K, V> unfiltered, Predicate<? super Entry<K, V>> entryPredicate) 通过Entry过滤
filterKeys(Map<K, V> unfiltered, final Predicate<? super K> keyPredicate) 通过Key过滤
filterValues(Map<K, V> unfiltered, final Predicate<? super V> valuePredicate) 通过value过滤
transformEntries(Map<K, V1> fromMap, EntryTransformer<? super K, ? super V1, V2> transformer) 转换Entry
transformValues(Map<K, V1> fromMap, Function<? super V1, V2> function) 转换value

创建各种Map的方法

Java 复制代码

```java
@Test
public void createDemo(){
    Maps.newHashMap();
    Maps.newHashMapWithExpectedSize(10);
    //Maps.newEnumMap();
    Maps.newConcurrentMap();

    Maps.newLinkedHashMap();
    Maps.newLinkedHashMapWithExpectedSize(10);

    Maps.newTreeMap();
}

```

set转map

Java 复制代码

```java
    @Test
    public void asMapTest(){
        Set<String> set = Sets.newHashSet("a","b","c");
        //将set转成Map,key为set元素,value为每个元素的长度
        Map<String,Integer> map = Maps.asMap(set,String::length);
        System.out.println(map);
    }

```

计算map的差值

Java 复制代码

```java
    @Test
    public void differenceTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> map2 = Maps.newHashMap();
        map2.put("a","1");
        map2.put("e","5");
        map2.put("f","6");
        //mapDifference是将两个map相同的部分剔除
        MapDifference<String,String> mapDifference =
Maps.difference(map1,map2);
        //两个Map相同的部分
        System.out.println(mapDifference.entriesInCommon());
        //左边集合剔除相同部分后的剩余
        System.out.println(mapDifference.entriesOnlyOnLeft());
        //右边集合剔除相同部分后的剩余
        System.out.println(mapDifference.entriesOnlyOnRight());
    }

```

通过Entry过滤

```java
    @Test
    public void filterEntriesTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> result = Maps.filterEntries(map1,item -> !item.getValue().equalsIgnoreCase("2"));
        System.out.println(result);

    }

```

通过Key过滤

```java
    @Test
    public void filterKeysTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> result = Maps.filterKeys(map1, item -> !item.equalsIgnoreCase("b"));
        System.out.println(result);
    }

```

通过value过滤

```java
    @Test
    public void filterValuesTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> result =  Maps.filterValues(map1,item -> !item.equalsIgnoreCase("3"));
        System.out.println(result);
    }

```

转换Entry

Java 复制代码

```java
    @Test
    public void transFormEntriesTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> result = Maps.transformEntries(map1,(k,v) -> k + v);
        System.out.println(result);
    }

```

转换value

Java 复制代码

```java
    @Test
    public void transformValuesTest(){
        Map<String,String> map1 = Maps.newHashMap();
        map1.put("a","1");
        map1.put("b","2");
        map1.put("c","3");
        Map<String,String> result = Maps.transformValues(map1, value -> value + 10);
        System.out.println(result);
    }

```

以上是Guava中提供的集合工具类，可以看到工具很丰富，包含了集合的各种常规操作，让我们在使用集合的时候更得心就手，熟练掌握Guava的各种集合工具类，势必能提升编码效率

不可变集合

不可变（Immutable）集合，顾名思义集合不可以被修改。初始创建不可变集合时吗，需要传入数据源，创建完成之后，集合就再也不能修改，增加，删除元素，否则将会报错。
这是一种防御性策略，防止集合在后续操作中被修改，从而引发问题。
不可变集合优点在于：

- 由于不可变集合仅仅只能读，多线程并发天然安全
- 由于不可变集合固定不变，可以将其当做常量安全，不用单线其他人修改

- 不可变集合占用更少内存空间
- 不可变集合不可以被修改，所以不用担心其他程序任意修改集合

Guava 不可变集合支持 JDK 所有集合接口：

Interface	JDK or Guava?	Immutable Version
`Collection`	JDK	`ImmutableCollection`
`List`	JDK	`ImmutableList`
`Set`	JDK	`ImmutableSet`
`SortedSet` / `NavigableSet`	JDK	`ImmutableSortedSet`
`Map`	JDK	`ImmutableMap`
`SortedMap`	JDK	`ImmutableSortedMap`
`Multiset`	Guava	`ImmutableMultiset`
`SortedMultiset`	Guava	`ImmutableSortedMultiset`
`Multimap`	Guava	`ImmutableMultimap`
`ListMultimap`	Guava	`ImmutableListMultimap`
`SetMultimap`	Guava	`ImmutableSetMultimap`
`BiMap`	Guava	`ImmutableBiMap`
`ClassToInstanceMap`	Guava	`ImmutableClassToInstanceMap`
`Table`	Guava	`ImmutableTable`

JDK也提供了Collections.unmodifiableXXX方法把集合包装为不可变形式，但我们认为不够好：

- 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；
- 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；

```java
@Test
public void test3(){
  List<Integer> list = Lists.newArrayList(1,2,3);
  List<Integer> list1 = Collections.unmodifiableList(list);

  // [1, 2, 3]
  System.out.println(list);
  // [1, 2, 3]
  System.out.println(list1);
  // list修改，list1也会被修改
  list.add(4);
  // [1, 2, 3, 4]
  System.out.println(list1);
}
```

创建不可变集合的几个方法：

- **copyOf** 方法，如ImmutableSet.copyOf(set);
- **of**方法，如ImmutableSet.of(“a”, “b”, “c”)或 ImmutableMap.of(“a”, 1, “b”, 2);
- **Builder**工具

我们可以使用如下几种方式创建不可变集合，以 `ImmutableList` 为例：

```java
List<String> fromList = Lists.newArrayList("影子","陈萍萍","庆帝");
ImmutableList<String> immutableList = ImmutableList.copyOf(fromList);
fromList.add("wowoow");
System.out.println(fromList);
System.out.println(immutableList);
//        ImmutableList.of("影子","陈萍萍");
//        ImmutableList.builder().add("关注").addAll(fromList).build();
```

3.guava缓存

缓存分为本地缓存和远端缓存。

今天说的 Guava Cache 是google guava中的一个内存缓存模块，用于将数据缓存到JVM内存中。他很好的解决了下面的几个问题：

- 很好的封装了get、put操作，能够集成数据源 ；
- 线程安全的缓存，与ConcurrentMap相似，但前者增加了更多的元素失效策略，后者只能显示的移除元素；

- Guava Cache提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。定时回收有两种：按照写入时间，最早写入的最先回收；按照访问时间，最早访问的最早回收；
- 监控缓存加载/命中情况

通常来说，`Guava Cache`适用于：

- 你愿意消耗一些内存空间来提升速度。
- 你预料到某些键会被查询一次以上。
- 缓存中存放的数据总量不会超出内存容量。（Guava Cache是单个应用运行时的本地缓存。它不把数据存放到文件或外部服务器。如果这不符合你的需求，请尝试Memcached这类工具）

如果你的场景符合上述的每一条，Guava Cache就适合你。

1.创建（加载）cache

Guava Cache提供了两种方式创建一个Cache: CacheLoader和Callable 。

CacheLoader

CacheLoader可以理解为一个固定的加载器，在创建Cache时指定，然后简单地重写V load(K key) throws Exception方法，就可以达到当检索不存在的时候，会自动的加载数据的。例子代码如下：

Java 复制代码

```java
@Test
    public void testCacheLoader() throws ExecutionException, InterruptedException {
        LoadingCache<String, String > loadingCache = CacheBuilder.newBuilder()
                //最大容量为100（基于容量进行回收）
                .maximumSize(100)
                //创建一个CacheLoader，重写load方法，以实现"当get时缓存不存在，则load，放到缓存，并返回"的效果
                .build(new CacheLoader<String, String>() {
                    //重点，自动写缓存数据的方法，必须要实现
                    @Override
                    public String load(String key) throws Exception{
                        return "value_" + key;
                    }

                });

        //测试例子，调用其get方法，cache会自动加载并返回
        String value = loadingCache.get("key1");
        //返回value_1
        System.out.println("value:"+value);



    }
```

Callable

在上面的build方法中是可以不用创建CacheLoader的，不管有没有CacheLoader，都是支持Callable的。Callable在get时可以指定，效果跟CacheLoader一样，区别就是两者定义的时间点不一样，Callable更加灵活，可以理解为Callable是对CacheLoader的扩展。例子代码如下：

Java 复制代码

```java
    public void testCallableCache() throws ExecutionException {
           Cache<String, String > cache = CacheBuilder.newBuilder()
                   //最大容量为100（基于容量进行回收）
                   .maximumSize(100)
                   .build();
           String key = "1";
           //loadingCache的定义跟上一面一样
           //get时定义一个Callable
           String value = cache.get(key, new Callable<String>() {
               @Override
               public String call() throws Exception {
                   return "call_" + key;
               }
           });
           System.out.println("value:"+value);
       }
```

2.缓存清除策略

基于容量的清除策略

通过CacheBuilder.maximumSize(long)方法可以设置Cache的最大容量数，当缓存数量达到或接近该最大值时，Cache将清除掉那些最近最少使用的缓存

Java 复制代码

```java
 /**
     * 初始化缓存
     * @param cache
     */
    public static void initCache(LoadingCache cache) throws ExecutionException {
        /* 前三条记录 */
        for (int i = 1; i <= 3; i++) {

            //get时候没数据 会 load装配进去   相当一个连接数据源，如果缓存没有 则读取数据源，加载load方法
            cache.get(String.valueOf(i));
        }
    }

    /**
     * 打印缓存中所有的内容
     * @param cache
     */
    public  void displayCache(LoadingCache cache){
        Iterator its = cache.asMap().entrySet().iterator();
        while (its.hasNext()) {
            System.out.println(its.next().toString());
        }
    }

    /**
     * 测试缓存清除策略1：基于容量清除
     */
    @Test
    public void testClear1() throws InterruptedException, ExecutionException {
        LoadingCache<String, Object> cache = CacheBuilder.newBuilder()
                ////最大缓存个数 3
                .maximumSize(3)
                //读多长时间后删除
                //.expireAfterAccess(3, TimeUnit.SECONDS)
                //写多长时间后删除
                //.expireAfterWrite(3, TimeUnit.SECONDS)
                //基于引用的删除
                //.weakValues()
                .build(new CacheLoader<String, Object>() {
                    //读数据源 加载数据到缓存
                    @Override
                    public String load(String key) throws Exception {
                        return "value_"+key;
                    }
                });
```

```
46
47          initCache(cache);
48          System.out.println("------------------------");
49          cache.get("4");//再增加一条记录
50
51
52          System.out.println("---打印缓存中所有的内容---");
53          displayCache(cache);
54
55      }
```

基于权重的清除 策略

使用CacheBuilder.weigher(Weigher)指定一个权重函数，并且用CacheBuilder.maximumWeight(long)指定最大总重。

> 如每一项缓存所占据的内存空间大小都不一样，可以看作它们有不同的“权重”（weights），作为执行清除策略时优化回收的对象

Java 复制代码

```java
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .maximumWeight(100000)
       .weigher(new Weigher<Key, Graph>() {
          public int weigh(Key k, Graph g) {
            return g.vertices().size();
          }
        })
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) { // no checked exception
               return createExpensiveGraph(key);
             }
           });

```

基于存活时间的清除

- expireAfterWrite 写缓存后多久过期
- expireAfterAccess 读写缓存后多久过期
- refreshAfterWrite 写入数据后多久过期,只阻塞当前数据加载线程,其他线程返回旧值

这几个策略时间可以单独设置,也可以组合配置。

Java 复制代码

```java
/**
     * 测试缓存清除策略2：基于存活时间的清除
     */
    @Test
    public void testClear2() throws InterruptedException,
ExecutionException {
        LoadingCache<String, Object> cache = CacheBuilder.newBuilder()
                ////最大缓存个数 3
                .maximumSize(3)
                //读写缓存后多久过期
                .expireAfterAccess(3, TimeUnit.SECONDS)
                //写多长时间后删除
                //.expireAfterWrite(3, TimeUnit.SECONDS)
                //基于引用的删除
                //.weakValues()
                .build(new CacheLoader<String, Object>() {
                    //读数据源 加载数据到缓存
                    @Override
                    public String load(String key) throws Exception {
                        return "value_"+key;
                    }
                });

        initCache(cache);
        Thread.sleep(1000);
        //访问1 1被访问
        //getIfPresent ：该方法只是简单的把Guava Cache当作Map的替代品，不执行load
方法;
        cache.getIfPresent("1");

        displayCache(cache);
        //歇了2.1秒
        Thread.sleep(2100);
        //最后缓存中会留下1
        System.out.println("----------------------------");
        displayCache(cache);

        Thread.sleep(1100);
        System.out.println("============================");
        displayCache(cache);//没有数据了

    }
```

expireAfterWrite/expireAfterAccess

使用背景

如果对缓存设置过期时间，在高并发下同时执行get操作，而此时缓存值已过期了，如果没有保护措施，则会导致大量线程同时调用生成缓存值的方法，比如从数据库读取，对数据库造成压力，这也就是我们常说的“缓存击穿”。

做法

而Guava cache则对此种情况有一定控制。当大量线程用相同的key获取缓存值时，只会有一个线程进入load方法，而其他线程则等待，直到缓存值被生成。这样也就避免了缓存击穿的危险。这两个配置的区别前者记录写入时间，后者记录写入或访问时间，内部分别用writeQueue和accessQueue维护。
PS: 但是在高并发下，这样还是会阻塞大量线程。

refreshAfterWrite

使用背景

使用 expireAfterWrite 会导致其他线程阻塞。

做法

更新线程调用load方法更新该缓存，其他请求线程返回该缓存的旧值。

示例：

Java 复制代码

```java
package com.chenj.guava;


import com.google.common.cache.CacheBuilder;
import com.google.common.cache.CacheLoader;
import com.google.common.cache.LoadingCache;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
 * google guava cache 缓存demo
 *
 */
public class DemoGuavaCache {
    public static void main(String[] args) throws Exception {
        LoadingCache<Integer, String> cache = CacheBuilder.newBuilder()
                //设置并发级别为8，并发级别是指可以同时写缓存的线程数
                .concurrencyLevel(8)
                //设置缓存容器的初始容量为10
                .initialCapacity(10)
                //设置缓存最大容量为100，超过100之后就会按照LRU最近虽少使用算法来移除缓存项
                .maximumSize(100)
                //是否需要统计缓存情况,该操作消耗一定的性能,生产环境应该去除
                .recordStats()
                //设置写缓存后n秒钟过期
                .expireAfterWrite(17, TimeUnit.SECONDS)
                //设置读写缓存后n秒钟过期,实际很少用到,类似于expireAfterWrite
                //.expireAfterAccess(17, TimeUnit.SECONDS)
                //只阻塞当前数据加载线程，其他线程返回旧值
                //.refreshAfterWrite(13, TimeUnit.SECONDS)
                //设置缓存的移除通知
                .removalListener(notification -> {
                    System.out.println(notification.getKey() + " " + notification.getValue() + " 被移除,原因:" + notification.getCause());
                })
                //build方法中可以指定CacheLoader，在缓存不存在时通过CacheLoader的实现自动加载缓存
                .build(new DemoCacheLoader());

        //模拟线程并发
        new Thread(() -> {
            //非线程安全的时间格式化工具
            SimpleDateFormat simpleDateFormat = new SimpleDateFormat("HH:mm:ss");
```

```
45              try {
46                  for (int i = 0; i < 15; i++) {
47                      String value = cache.get(1);
48                      System.out.println(Thread.currentThread().getName() +
    " " + simpleDateFormat.format(new Date()) + " " + value);
49                      TimeUnit.SECONDS.sleep(3);
50                  }
51              } catch (Exception ignored) {
52              }
53          }).start();
54
55          new Thread(() -> {
56              SimpleDateFormat simpleDateFormat = new
    SimpleDateFormat("HH:mm:ss");
57              try {
58                  for (int i = 0; i < 10; i++) {
59                      String value = cache.get(1);
60                      System.out.println(Thread.currentThread().getName() +
    " " + simpleDateFormat.format(new Date()) + " " + value);
61                      TimeUnit.SECONDS.sleep(5);
62                  }
63              } catch (Exception ignored) {
64              }
65          }).start();
66          //缓存状态查看
67          System.out.println(cache.stats().toString());
68      }
69
70      /**
71       * 随机缓存加载,实际使用时应实现业务的缓存加载逻辑,例如从数据库获取数据
72       */
73      public static class DemoCacheLoader extends CacheLoader<Integer,
    String> {
74          @Override
75          public String load(Integer key) throws Exception {
76              System.out.println(Thread.currentThread().getName() + " 加载数据
    开始");
77              TimeUnit.SECONDS.sleep(8);
78              Random random = new Random();
79              System.out.println(Thread.currentThread().getName() + " 加载数据
    结束");
80              return "value:" + random.nextInt(10000);
81          }
82      }
83  }
84
```

expireAfterWrite与refreshAfterWrite单独使用与混合使用的策略分析

```
已知配置条件:
Thread-1 每 3 秒获取一次缓存id=1的数据
Thread-2 每 5 秒获取一次缓存id=1的数据
加载一次缓存加载数据耗时 8 秒
```

1、expireAfterWrite单独使用

expireAfterWrite=17

```
Thread-2 加载数据开始
Thread-2 加载数据结束
Thread-1 01:04:07 value:6798
Thread-2 01:04:07 value:6798
Thread-1 01:04:10 value:6798
Thread-2 01:04:12 value:6798
Thread-1 01:04:13 value:6798
Thread-1 01:04:16 value:6798
Thread-2 01:04:17 value:6798
Thread-1 01:04:19 value:6798
Thread-1 01:04:22 value:6798
Thread-2 01:04:22 value:6798
1 value:6798 被移除,原因:EXPIRED
Thread-1 加载数据开始
Thread-1 加载数据结束
Thread-1 01:04:33 value:7836
Thread-2 01:04:33 value:7836
Thread-1 01:04:36 value:7836
Thread-2 01:04:38 value:7836
Thread-1 01:04:39 value:7836

```

说明:

启动时Thread-2加载数据,此时缓存中无数据,Thread-1阻塞等待Thread-2加载完成数据. 在设置的时间数据过期后Thread-1加载数据,Thread-2本应该01:04:22后的5秒加载数据,但是Thread-1等待3秒后加载,数据加载耗时8秒,所以Thread-2在01:04:33时加载数据成功.

结论:

当其他线程在加载数据的时候,当前线程会一直阻塞等待其他线程加载数据完成.

2、refreshAfterWrite单独使用

refreshAfterWrite=17

```
Thread-2 加载数据开始
Thread-2 加载数据结束
Thread-1 01:13:32 value:551
Thread-2 01:13:32 value:551
Thread-1 01:13:35 value:551
Thread-2 01:13:37 value:551
Thread-1 01:13:38 value:551
Thread-1 01:13:41 value:551
Thread-2 01:13:42 value:551
Thread-1 01:13:44 value:551
Thread-1 01:13:47 value:551
Thread-2 01:13:47 value:551
Thread-1 加载数据开始
Thread-2 01:13:52 value:551
Thread-2 01:13:57 value:551
Thread-1 加载数据结束
1 value:551 被移除,原因:REPLACED
Thread-1 01:13:58 value:827
Thread-1 01:14:01 value:827
Thread-2 01:14:02 value:827
Thread-1 01:14:04 value:827
Thread-2 01:14:07 value:827
```

说明:

启动时Thread-2加载数据,此时缓存中无数据,Thread-1阻塞等待Thread-2加载完成数据. 在设置的时间数据过期后Thread-1加载数据,Thread-2仍然按照策略获取到旧数据成功.

结论:

当没有数据的时候,其他线程在加载数据的时候,当前线程会一直阻塞等待其他线程加载数据完成;

如果有数据的情况下其他线程正在加载数据,当前线程返回旧数据.

3、expireAfterWrite与refreshAfterWrite一起使用情况一

expireAfterWrite=13

refreshAfterWrite=17

```
Thread-2 加载数据开始
Thread-2 加载数据结束
Thread-1 01:18:32 value:5901
Thread-2 01:18:32 value:5901
Thread-1 01:18:35 value:5901
Thread-2 01:18:37 value:5901
Thread-1 01:18:38 value:5901
Thread-1 01:18:41 value:5901
Thread-2 01:18:42 value:5901
Thread-1 01:18:44 value:5901
1 value:5901 被移除,原因:EXPIRED
Thread-1 加载数据开始
Thread-1 加载数据结束
Thread-2 01:18:55 value:1300
Thread-1 01:18:55 value:1300
Thread-1 01:18:58 value:1300
Thread-2 01:19:00 value:1300
Thread-1 01:19:01 value:1300

```

说明:

启动时Thread-2加载数据,此时缓存中无数据,Thread-1阻塞等待Thread-2加载完成数据. 在设置的时间数据过期后Thread-1加载数据,Thread-2本应该01:18:42后的5秒加载数据,但是Thread-1等待3秒后加载,数据加载耗时8秒,所以Thread-2在01:18:55时加载数据成功.

结论:

当其他线程在加载数据的时候,当前线程会一直阻塞等待其他线程加载数据完成,与单独使用expireAfterWrite一样的效果.

4、expireAfterWrite与refreshAfterWrite一起使用情况二

expireAfterWrite=17

refreshAfterWrite=13

Java 复制代码

```
Thread-2 加载数据开始
Thread-2 加载数据结束
Thread-1 01:20:25 value:1595
Thread-2 01:20:25 value:1595
Thread-1 01:20:28 value:1595
Thread-2 01:20:30 value:1595
Thread-1 01:20:31 value:1595
Thread-1 01:20:34 value:1595
Thread-2 01:20:35 value:1595
Thread-1 01:20:37 value:1595
Thread-2 加载数据开始
Thread-1 01:20:40 value:1595
Thread-2 加载数据结束
Thread-1 01:20:48 value:2277
1 value:1595 被移除,原因:EXPIRED
Thread-2 01:20:48 value:2277
Thread-1 01:20:51 value:2277
Thread-2 01:20:53 value:2277
Thread-1 01:20:54 value:2277
Thread-1 01:20:57 value:2277
Thread-2 01:20:58 value:2277
Thread-1 01:21:00 value:2277
Thread-1 加载数据开始
Thread-2 01:21:03 value:2277
Thread-1 加载数据结束
Thread-2 01:21:11 value:3750
1 value:2277 被移除,原因:EXPIRED
Thread-1 01:21:11 value:3750
Thread-1 01:21:14 value:3750
Thread-2 01:21:16 value:3750
Thread-1 01:21:17 value:3750
Thread-1 01:21:20 value:3750
Thread-2 01:21:21 value:3750

```

说明:

启动时Thread-2加载数据,此时缓存中无数据,Thread-1阻塞等待Thread-2加载完成数据. 在设置的时间数据过期后Thread-2加载数据,Thread-1仍然按照策略在01:20:40获取到旧数据成功,但是本应该01:20:45继续获取一次数据但是等到01:20:48才获取成功.

结论:

当没有数据的时候,其他线程在加载数据的时候,当前线程会一直阻塞等待其他线程加载数据完成; 如果有数据的情况下其他线程正在加载数据,已经超过refreshAfterWrite设置时间但是没有超过expireAfterWrite设置的时间时当前线程返回旧数据.

如果有数据的情况下其他线程正在加载数据,已经超过expireAfterWrite设置的时间时当前线程阻塞等待其他线程加载数据完成. 这种情况适合与设置一个加载缓冲区的情况,既能保证过期后加载数据,又能保证长时间没访问多个线程并发时获取到过期旧数据的情况.

异步刷新

使用背景

单个key并发下，使用refreshAfterWrite，虽然不会阻塞了，但是如果恰巧同时多个key同时过期，还是会给数据库造成压力，这就是我们所说的“**缓存雪崩**”。

做法

这时就要用到异步刷新，将刷新缓存值的任务交给后台线程，所有的用户请求线程均返回旧的缓存值。
方法是覆盖CacheLoader的reload方法，使用线程池去异步加载数据
PS：只有重写了 reload 方法才有“异步加载”的效果。默认的 reload 方法就是同步去执行 load 方法。

总结
大家都应该对各个失效/刷新机制有一定的理解，清楚在各个场景可以使用哪个配置，简单总结一下：

1. expireAfterWrite 是允许一个线程进去load方法，其他线程阻塞等待。
2. refreshAfterWrite 是允许一个线程进去load方法，其他线程返回旧的值。
3. 在上一点基础上做成异步，即回源线程不是请求线程。异步刷新是用线程异步加载数据，期间所有请求返回旧的缓存值。

显式清除

清除单个key：Cache.invalidate(key)
批量清除key：Cache.invalidateAll(keys)
清除所有缓存项：Cache.invalidateAll()

基于引用的清除策略

在构建Cache实例过程中，通过设置使用弱引用的键、或弱引用的值、或软引用的值，从而使JVM在GC时顺带实现缓存的清除
CacheBuilder.weakKeys()：使用弱引用存储键。当键没有其它（强或软）引用时，缓存项可以被垃圾回收
CacheBuilder.weakValues()：使用弱引用存储值。当值没有其它（强或软）引用时，缓存项可以被垃圾回收
CacheBuilder.softValues()：使用软引用存储值。软引用只有在响应内存需要时，才按照全局最近最少使用的顺序回收。考虑到使用软引用的性能影响，我们通常建议使用更有性能预测性的缓存大小限定

Java 复制代码

```java
 /**
     * 测试缓存清除策略3：基于引用的删除
     */
    @Test
    public void testClear3() throws InterruptedException,
ExecutionException {
        LoadingCache<String, Object> cache = CacheBuilder.newBuilder()
                ////最大缓存个数 3
                .maximumSize(3)
                //读写缓存后多久过期
                //.expireAfterAccess(3, TimeUnit.SECONDS)
                //写多长时间后删除
                //.expireAfterWrite(3, TimeUnit.SECONDS)
                //基于引用的删除
                .weakValues()
                .build(new CacheLoader<String, Object>() {
                    //读数据源 加载数据到缓存
                    @Override
                    public String load(String key) throws Exception {
                        return "value_"+key;
                    }
                });

        Object value = new Object();
        cache.put("1",value);
        value = new Object(); //原对象不再有强引用
        // 强制垃圾回收
        System.gc();
        System.out.println(cache.getIfPresent("1"));

 }
```

3.监听

可以为Cache对象添加一个移除监听器，这样当有记录被删除时可以感知到这个事件。

```java
  RemovalListener<String, String> listener = notification -> {
            if(notification.wasEvicted()){
                RemovalCause cause = notification.getCause();
                System.out.println("remove cause is "+cause.toString());
                System.out.println("[" + notification.getKey() + ":" +
notification.getValue() + "] is removed!");
            }
        };
        Cache<String,String> cache = CacheBuilder.newBuilder()
                .maximumSize(4)
                //添加同步删除监听
                //.removalListener(listener)
                //添加异步删除监听
                .removalListener(RemovalListeners.asynchronous(listener,
Executors.newSingleThreadExecutor()))
                .build();

        cache.put("guava1","guava1");
        cache.put("guava2","guava1");
        cache.put("guava3","guava1");
        cache.put("guava4","guava1");

        cache.put("guava5","guava1");//第5个时候删除


        System.out.println("-----------主程序end-------------");
```

通过`CacheBuilder.removalListener(RemovalListener)`，你可以声明一个监听器，以便缓存项被移除时做一些额外操作。缓存项被移除时，`RemovalListener`会获取移除通知`RemovalNotification`，其中包含移除原因`RemovalCause`、键和值。

警告：默认情况下，监听器方法是在移除缓存时同步调用的。因为缓存的维护和请求响应通常是同时进行的，代价高昂的监听器方法在同步模式下会拖慢正常的缓存请求。假如在同步监听模式下，**监听方法中的逻辑特别复杂，执行效率慢，那此时如果有大量的key进行清理，会使整个缓存性能变得很低下**，所以此时适合用异步监听`RemovalListeners.asynchronous(RemovalListener, Executor)`把监听器装饰为异步操作，移除key与监听key的移除分属2个线程。

Java 复制代码

```java
 @Test
    public void testListener() throws InterruptedException {
        RemovalListener<String, String> listener = notification -> {
            if(notification.wasEvicted()){
                //监听方法中特别慢
                System.out.println("------耗时操作--------begin");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("------耗时操作--------end");
                RemovalCause cause = notification.getCause();
                System.out.println("remove cause is "+cause.toString());
                System.out.println("[" + notification.getKey() + ":" +
notification.getValue() + "] is removed!");
            }
        };
        Cache<String,String> cache = CacheBuilder.newBuilder()
                .maximumSize(4)
                //添加同步删除监听
                //.removalListener(listener)
                //添加异步删除监听
                .removalListener(RemovalListeners.asynchronous(listener,
Executors.newSingleThreadExecutor()))
                .build();

        cache.put("guava1","guava1");
        cache.put("guava2","guava1");
        cache.put("guava3","guava1");
        cache.put("guava4","guava1");

        cache.put("guava5","guava1");//第5个时候删除


        System.out.println("-----------主程序end-------------");

    }
```

4.统计

guava cache还有一些其他特性，比如weight 按权重回收资源，统计等，这里列出统计。

CacheBuilder.recordStats()用来开启Guava Cache的统计功能。统计打开后Cache.stats()方法返回

如下统计信息：

- hitRate()：缓存命中率；
- hitMiss(): 缓存失误率；
- loadcount()；加载次数；
- averageLoadPenalty()：加载新值的平均时间，单位为纳秒；
- evictionCount()：缓存项被回收的总数，不包括显式清除。