Guava YYDS

并发工具

ListenableFuture是jdk的Future接口的一个扩展，guava团队建议在所有使用Future的地方，使用ListenableFuture进行代替。ListenableFuture增加了Future执行成功或失败的监听器，可以在计算完成时即时进行回调，使得并发编程的代码更加高效。

```
// 使用listeningDecorator方法包装线程池
ListeningExecutorService service =
MoreExecutors.listeningDecorator(Executors.newFixedThreadPool(10));

// 调用submit方法，即可返回ListenableFuture
ListenableFuture<Explosion> explosion = service.submit(
        new Callable<Explosion>() {
            public Explosion call() {
                return pushBigRedButton();
            }
        });

// 通过Futures工具类的addCallback方法，为ListenableFuture添加执行成功和失败的回调
Futures.addCallback(
        explosion,
        new FutureCallback<Explosion>() {

            // 执行成功
            public void onSuccess(Explosion explosion) {
                walkAwayFrom(explosion);
            }

            // 执行失败
            public void onFailure(Throwable thrown) {
                battleArchNemesis();
            }
        },
        service);
```

Service接口代表了对象的运行状态，它提供了开始和停止的方法。计时器、RPC服务器等服务均可以实现此接口，以便管理开启和结束等状态。

Service提供了五个正常的状态和一个失败的状态，即：

- Service.State.NEW 新建
- Service.State.STARTING 开启
- Service.State.RUNNING 运行中
- Service.State.STOPPING 正在停止
- Service.State.TERMINATED 中止
- Service.State.FAILED 失败

正常服务的生命周期应该为

新建->开启->运行中->停止->中止

缓存

本地缓存（内存级别的缓存），支持各种方式的过期行为，不会将数据存到硬盘。

```
// 创建缓存
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .build(new CacheLoader<>() {
            @Override
            public Graph load(Key key) throws Exception {
                // 通过key来生成值，这个过程消耗资源比较大，因此需要缓存
                return createExpensiveGraph(key);
            }
        });

// 读取缓存中的值
try {
    // 尝试获取缓存中的值，如果没有值，通过CacheLoader执行load方法添加值
    graphs.get(key);
} catch (ExecutionException e) {
    // 异常处理
}
```

反射工具

提供更方便的反射工具。

```
// 获取类的包名
final String packageName = Reflection.getPackageName(Test.class);

// 生成一个接口的动态代理对象
Foo foo = Reflection.newProxy(Foo.class, invocationHandler);
```

数学库

提供了一些JDK没有提供的数学工具，这些工具已经经过了彻底的测试。数学库中的函数很多，下面是一些简单的例子。

```
// 保证相加的结果不溢出，如果溢出则抛出ArithmeticException
int sum = IntMath.checkedAdd(a, b);

// 获取两个整数的最大公约数
int gcd = IntMath.gcd(a, b);

// 比较两个浮点数是否在容差范围内相等
final boolean equals = DoubleMath.fuzzyEquals(1.0 / 3, 0.3333333, 0.00001);

// 保证幂运算的结果不溢出，如果溢出则抛出ArithmeticException
final long checkedPow = LongMath.checkedPow(5, 20);

// 将BigDecimal数字近似为double，如果数字过大，则返回Double.MAX_VALUE。
final double v = BigDecimalMath.roundToDouble(BigDecimal.valueOf(2).pow(2000), HALF_UP);
```

I/O

封装了一些更方便的I/O工具。

```
// 将整个输入流读为byte数组，不会关闭流
byte[] bytes = ByteStreams.toByteArray(inputStream);

// 将整个输入流中的字节复制到输出流中，不会关闭流或清空缓存区
long length = ByteStreams.copy(inputStream, outputStream);

// 将整个输入流/可读对象中的数据按行读出，不包含换行符
List<String> lines = CharStreams.readLines(new InputStreamReader(inputStream));

// 将整个输入流/可读对象中的数据读成一个字符串，不关闭流
String s = CharStreams.toString(new InputStreamReader(inputStream));

// 读取一个文本文件所有行
ImmutableList<String> textLines = Files.asCharSource(file, Charsets.UTF_8)
        .readLines();

// 计算一个文件里出现的不同单词的次数
Multiset<String> wordOccurrences = HashMultiset.create(
        Splitter.on(CharMatcher.whitespace())
                .trimResults()
                .omitEmptyStrings()
                .split(Files.asCharSource(file, Charsets.UTF_8).read()));
```

集合工具类

```java
// 创建一个ArrayList，可以传不定长参数
List<Integer> arrayList = Lists.newArrayList(1, 2, 3);

// 创建一个LinkedList，传一个iterable（可迭代）变量
List<Integer> linkedList = Lists.newLinkedList(arrayList);

// 翻转一个集合，原集合不变
List<Integer> reversed = Lists.reverse(arrayList);

// 集合变换，传一个function对象，相当于stream操作
List<Integer> transformed = Lists.transform(arrayList, i -> i * 2);

// 创建一个HashSet
Set<Integer> hashSet = Sets.newHashSet(1, 2, 3);

// 创建一个TreeSet
Set<Integer> treeSet = Sets.newTreeSet(Sets.newHashSet(4, 2, 3));

// 求两个Set的并集，返回的是一个只读的视图
Sets.SetView<Integer> union = Sets.union(hashSet, treeSet);

// 求两个Set的交集，返回的是一个只读的视图
Sets.SetView<Integer> intersection = Sets.intersection(hashSet, treeSet);

// 求两个Set的差集，返回的是一个只读的视图
Sets.SetView<Integer> difference = Sets.difference(hashSet, treeSet);

// jdk9创建不可变map的api
Map<String, Object> map0 = Map.of("a", 1, "b", 2, "c", 3);
Map<String, Object> map1 = Map.of("a", 1, "b", 2, "d", 4);

// 求两个map的差异
MapDifference<String, Object> mapDifference = Maps.difference(map0, map1);
```

扩展集合

```java
// Multiset是支持重复元素的集合，即背包
// 创建背包
Multiset<String> multiset = HashMultiset.create();

// 向背包中添加一个元素
multiset.add("one");

// 向背包中添加多个相同元素
multiset.add("two", 4);

// 查询元素个数
int oneCount = multiset.count("one");

// 删除一个元素若干次
multiset.remove("two", 2);

// Multimap是一个key对应多个value的map
// 创建一个multimap
Multimap<String, Object> multimap = HashMultimap.create();

// 可以在相同的key下put值，put后map元素为{a=[1, 2]}
multimap.put("a", 1);
multimap.put("a", 2);

// 可以通过putAll为一个key一次性放置多个value，value不保证有序
multimap.putAll("b", Sets.newHashSet(3, 4, 5));
```

哈希

提供更复杂的哈希算法，包括Bloom filter数据结构，该结构通过允许少量的错误来节省大量的存储空间。

```
// 要计算hashCode的实体类
class Person {
    int id;
    String firstName;
    String lastName;
    int birthYear;
}

// 将对象分解为基础属性值的数据漏斗
Funnel<Person> personFunnel = (person, into) -> into
        .putInt(person.id)
        .putString(person.firstName, Charsets.UTF_8)
        .putString(person.lastName, Charsets.UTF_8)
        .putInt(person.birthYear);

// 初始化哈希函数
HashFunction hf = Hashing.sha256();

// 计算哈希值
HashCode hc = hf.newHasher()
        .putLong(id)
        .putString(name, Charsets.UTF_8)
        .putObject(person, personFunnel)
        .hash();

// 创建BloomFilter
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 500, 0.01);

// 向Bloom Filter对象中添加元素
for (Person friend : friendsList) {
    friends.put(friend);
}

// 判断Bloom Filter里是否可能包含某个元素，这里牺牲一定的准确度来换取空间和执行效率
if (friends.mightContain(dude)) {
    // 如果不包含元素并且代码执行到这里的概率是1%。
}
```

注意事项

1. 部分功能已加入到 JDK 中，可以优先使用 JDK 中的相同功能，这些功能包括：
   - 用于空值处理 Optional 的类
   - 基础的函数式接口，如 Function、Predicate、Supplier
   - 流式计算，FluentIterable 的操作相当于 JDK8 中的 Stream 操作。
   - I/O 操作，Files 工具类
   - 不可变集合，jdk 中有 ImmutableCollections 工具类
2. 所有的类都有自己的使用场景，没有银弹。