案例：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_060_util_guava>

代码结构：

com.zx.guava.base 基础工具

com.zx.guava.collection 集合

com.zx.guava.cache 缓存

com.zx.guava.callback 函数式编程

com.zx.guava.concurrent 并发

com.zx.guava.string 字符串处理

com.zx.guava.io IO

com.zx.guava.bus 事件

com.zx.guava.limit 限流--令牌桶限流算法

# 引言介绍

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。

查阅Javadoc并不一定是学习这些库最有效的方式。在此，我们希望通过此文档为Guava中最流行和最强大的功能，提供更具可读性和解释性的说明。

# 基本工具 [Basic utilities] 让使用Java语言变得更舒适

## 使用和避免null

null是模棱两可的，会引起令人困惑的错误，有些时候它让人很不舒服。很多Guava工具类用快速失败拒绝null值，而不是盲目地接受

详细看[ObjectsDemo](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_060_util_guava/src/main/java/com/zx/guava/base/ObjectsDemo.java)

使用Optional除了赋予null语义，增加了可读性，最大的优点在于它是一种傻瓜式的防护。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。

如同输入参数，方法的返回值也可能是null。和其他人一样，你绝对很可能会忘记别人写的方法method(a,b)会返回一个null，就好像当你实现

method(a,b)时，也很可能忘记输入参数a可以为null。将方法的返回类型指定为Optional，也可以迫使调用者思考返回的引用缺失的情形。

其他处理null的便利方法

当你需要用一个默认值来替换可能的null，请使用Objects.firstNonNull(T, T) 方法。如果两个值都是null，该方法会抛出NullPointerException。

Optional也是一个比较好的替代方案，例如：Optional.of(first).or(second)。还有其它一些方法专门处理null或空字符串：emptyToNull(String)，

nullToEmpty(String)，isNullOrEmpty(String)。我们想要强调的是，这些方法主要用来与混淆null/空的API进行交互。当每次你写下混淆null/空的代码时，

Guava团队都泪流满面。（好的做法是积极地把null和空区分开，以表示不同的含义，在代码中把null和空同等对待是一种令人不安的坏味道。

## 前置条件

让方法中的条件检查更简单

详细看[PreconditionsTest](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_060_util_guava/src/main/java/com/zx/guava/base/PreconditionsTest.java)

静态导入方法进行检查

import static com.google.common.base.Preconditions.\*;

索引值常用来查找列表、字符串或数组中的元素，如List.get(int), String.charAt(int)

位置值和位置范围常用来截取列表、字符串或数组，如List.subList(int，int), String.substring(int)

相比Apache Commons提供的类似方法，我们把Guava中的Preconditions作为首选。Piotr Jagielski在他的博客中简要地列举了一些理由：

在静态导入后，Guava方法非常清楚明晰。checkNotNull清楚地描述做了什么，会抛出什么异常；

checkNotNull直接返回检查的参数，让你可以在构造函数中保持字段的单行赋值风格：this.field = checkNotNull(field)

简单的、参数可变的printf风格异常信息。鉴于这个优点，在JDK7已经引入Objects.requireNonNull的情况下，我们仍然建议你使用checkNotNull。

在编码时，如果某个值有多重的前置条件，我们建议你把它们放到不同的行，这样有助于在调试时定位。此外，把每个前置条件放到不同的行，也可以帮助你编写清晰和有用的错误消息。

## 常见Object方法

简化Object方法实现，如hashCode()和toString()

详细看[ObjectsDemo](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_060_util_guava/src/main/java/com/zx/guava/base/ObjectsDemo.java)

## 排序

Guava强大的”流畅风格比较器”

## Throwables

简化了异常和错误的传播与检查

# 集合[Collections]

# 缓存[Caches]

# 函数式风格[Functional idioms]

# 并发[Concurrency]

# 字符串处理[Strings]

# 原生类型[Primitives]

# 区间[Ranges]

# I/O

# 散列[Hash]

# 事件总线[EventBus]

EventBus是Guava的事件处理机制，是设计模式中的观察者模式（生产/消费者编程模型）的优雅实现。对于事件监听和发布订阅模式，EventBus是一个非常优雅和简单解决方案，我们不用创建复杂的类和接口层次结构。

Observer模式是比较常用的设计模式之一，虽然有时候在具体代码里，它不一定叫这个名字，比如改头换面叫个Listener，但模式就是这个模式。手工实现一个Observer也不是多复杂的一件事，只是因为这个设计模式实在太常用了，Java就把它放到了JDK里面：Observable和Observer，从JDK 1.0里，它们就一直在那里。从某种程度上说，它简化了Observer模式的开发，至少我们不用再手工维护自己的Observer列表了。不过，如前所述，JDK里的Observer从1.0就在那里了，直到Java 7，它都没有什么改变，就连通知的参数还是Object类型。要知道，Java 5就已经泛型了。Java 5是一次大规模的语法调整，许多程序库从那开始重新设计了API，使其更简洁易用。当然，那些不做应对的程序库，多半也就过时了。这也就是这里要讨论知识更新的原因所在。今天，对于普通的应用，如果要使用Observer模式该如何做呢？答案是Guava的EventBus。

eventbus主要有以下几部分组成：

1、eventbus、asyncEventBus：事件发送器。

2、event：事件承载单元。

3、SubscriberRegistry：订阅者注册器，将订阅者注册到event上，即将有注解Subscribe的方法和event绑定起来。

4、Dispatcher：事件分发器，将事件的订阅者调用来执行。

5、Subscriber、SynchronizedSubscriber：订阅者，并发订阅还是同步订阅。

## EventBus基本用法

使用Guava之后, 如果要订阅消息, 就不用再继承指定的接口, 只需要在指定的方法上加上@Subscribe注解即可。代码如下：

消息封装类：

public class TestEvent {

private final int message;

public TestEvent(int message) {

this.message = message;

System.out.println("event message:"+message);

}

public int getMessage() {

return message;

}

}

消息接受类：

public class EventListener {

public int lastMessage = 0;

@Subscribe

public void listen(TestEvent event) {

lastMessage = event.getMessage();

System.out.println("Message:"+lastMessage);

}

public int getLastMessage() {

return lastMessage;

}

}

测试类及输出结果：

public class TestEventBus {

@Test

public void testReceiveEvent() throws Exception {

EventBus eventBus = new EventBus("test");

EventListener listener = new EventListener();

eventBus.register(listener);

eventBus.post(new TestEvent(200));

eventBus.post(new TestEvent(300));

eventBus.post(new TestEvent(400));

System.out.println("LastMessage:"+listener.getLastMessage());

;

}

}

//输出信息

event message:200

Message:200

event message:300

Message:300

event message:400

Message:400

LastMessage:400

## MultiListener的使用

只需要在要订阅消息的方法上加上@Subscribe注解即可实现对多个消息的订阅，代码如下：

public class MultipleListener {

public Integer lastInteger;

public Long lastLong;

@Subscribe

public void listenInteger(Integer event) {

lastInteger = event;

System.out.println("event Integer:"+lastInteger);

}

@Subscribe

public void listenLong(Long event) {

lastLong = event;

System.out.println("event Long:"+lastLong);

}

public Integer getLastInteger() {

return lastInteger;

}

public Long getLastLong() {

return lastLong;

}

}

测试类：

public class TestMultipleEvents {

@Test

public void testMultipleEvents() throws Exception {

EventBus eventBus = new EventBus("test");

MultipleListener multiListener = new MultipleListener();

eventBus.register(multiListener);

eventBus.post(new Integer(100));

eventBus.post(new Integer(200));

eventBus.post(new Integer(300));

eventBus.post(new Long(800));

eventBus.post(new Long(800990));

eventBus.post(new Long(800882934));

System.out.println("LastInteger:"+multiListener.getLastInteger());

System.out.println("LastLong:"+multiListener.getLastLong());

}

}

//输出信息

event Integer:100

event Integer:200

event Integer:300

event Long:800

event Long:800990

event Long:800882934

LastInteger:300

LastLong:800882934

## Dead Event

如果EventBus发送的消息都不是订阅者关心的称之为Dead Event。实例如下：

public class DeadEventListener {

boolean notDelivered = false;

@Subscribe

public void listen(DeadEvent event) {

notDelivered = true;

}

public boolean isNotDelivered() {

return notDelivered;

}

}

测试类：

public class TestDeadEventListeners {

@Test

public void testDeadEventListeners() throws Exception {

EventBus eventBus = new EventBus("test");

DeadEventListener deadEventListener = new DeadEventListener();

eventBus.register(deadEventListener);

eventBus.post(new TestEvent(200));

eventBus.post(new TestEvent(300));

System.out.println("deadEvent:"+deadEventListener.isNotDelivered());

}

}

//输出信息

event message:200

event message:300

deadEvent:true

说明：如果没有消息订阅者监听消息， EventBus将发送DeadEvent消息，这时我们可以通过log的方式来记录这种状态。

## Event的继承

如果Listener A监听Event A, 而Event A有一个子类Event B, 此时Listener A将同时接收Event A和B消息，实例如下：

Listener 类：

public class NumberListener {

private Number lastMessage;

@Subscribe

public void listen(Number integer) {

lastMessage = integer;

System.out.println("Message:"+lastMessage);

}

public Number getLastMessage() {

return lastMessage;

}

}

public class IntegerListener {

private Integer lastMessage;

@Subscribe

public void listen(Integer integer) {

lastMessage = integer;

System.out.println("Message:"+lastMessage);

}

public Integer getLastMessage() {

return lastMessage;

}

}

测试类：

public class TestEventsFromSubclass {

@Test

public void testEventsFromSubclass() throws Exception {

EventBus eventBus = new EventBus("test");

IntegerListener integerListener = new IntegerListener();

NumberListener numberListener = new NumberListener();

eventBus.register(integerListener);

eventBus.register(numberListener);

eventBus.post(new Integer(100));

System.out.println("integerListener message:"+integerListener.getLastMessage());

System.out.println("numberListener message:"+numberListener.getLastMessage());

eventBus.post(new Long(200L));

System.out.println("integerListener message:"+integerListener.getLastMessage());

System.out.println("numberListener message:"+numberListener.getLastMessage());

}

}

//输出类

Message:100

Message:100

integerListener message:100

numberListener message:100

Message:200

integerListener message:100

numberListener message:200

说明：在这个方法中,我们看到第一个事件(新的整数(100))是收到两个听众,但第二个(新长(200 l))只能到达NumberListener作为整数一不是创建这种类型的事件。可以使用此功能来创建更通用的监听器监听一个广泛的事件和更详细的具体的特殊的事件。

## 一个综合实例

public class UserThread extends Thread {

private Socket connection;

private EventBus channel;

private BufferedReader in;

private PrintWriter out;

public UserThread(Socket connection, EventBus channel) {

this.connection = connection;

this.channel = channel;

try {

in = new BufferedReader(new InputStreamReader(connection.getInputStream()));

out = new PrintWriter(connection.getOutputStream(), true);

} catch (IOException e) {

e.printStackTrace();

System.exit(1);

}

}

@Subscribe

public void recieveMessage(String message) {

if (out != null) {

out.println(message);

System.out.println("recieveMessage:"+message);

}

}

@Override

public void run() {

try {

String input;

while ((input = in.readLine()) != null) {

channel.post(input);

}

} catch (IOException e) {

e.printStackTrace();

}

//reached eof

channel.unregister(this);

try {

connection.close();

} catch (IOException e) {

e.printStackTrace();

}

in = null;

out = null;

}

}

mport java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import com.google.common.eventbus.EventBus;

public class EventBusChat {

public static void main(String[] args) {

EventBus channel = new EventBus();

ServerSocket socket;

try {

socket = new ServerSocket(4444);

while (true) {

Socket connection = socket.accept();

UserThread newUser = new UserThread(connection, channel);

channel.register(newUser);

newUser.start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

说明：用telnet命令登录：telnet 127.0.0.1 4444 ，如果你连接多个实例你会看到任何消息发送被传送到其他实例。

# 数学运算[Math]

# 反射[Reflection]