知暖知热的基础工具

规避`null`

大意粗心的使用`null`可会造成各种各样的bug. 研读Google的基础代码库后，我们发现95%的集合库都不支持存入`null`值，而且它们在面临`null`时，不会逆来顺受，而是掷地有声的快速作出失败响应。而这种策略对开发者无疑是益处极大的。

此外，`null`的意义还是模棱两可的。很难明确一个`null`返回值是代表何意。例如，`Map.get(key)`返回`null`既可以表示在map中的值是`null`，也可以表示不存在map中。`null`或表示失败，或表示成功，或表示任何含义。因此，使用非`null`的值可使含义表达得更清晰明朗。

即使如此，但正确的使用`null`却能带来内存和时间上的节省。而且在对象数组里，`null`是不可避免的。与类库比之，在应用代码里，它却是狡黠而迷惑的，诡异而难以细查的bug源头。最值一提的是，一个`null`值不具任何意义。

之于这些原因，许多Guava工具在设计时，都能对`null`作出快速失败响应，即直接拒绝处理`null`。然而，也有适用于`null`场景的解决方案。
当你必需要应对`null`时，Guava提供有很多工具来辅助你简易便利的使用`null`。

特定场景(Specific Cases)

如果你想把`null`作为`Set`的值或作为`Map`的key – 那么不要那么做。
通常你在作检索操作时，明确的指定使用`null`场景，效果会更一目了然。
如果你想把`null`作为`Map`的值 – 那么就省去那个`Entry`吧。
在`Set`中保持单一的非`null`值吧。

我们很容易混淆这种情况：`Key`在`Map`中的值为`null` 与`Key`在`Map`中固已没有值。
此时，分离这些特有的`key`出来会稍显好处，并需要思考与这些`key`关联的`null`在应用中具体是何意。

如果你想在一个稀疏的`List`里使用null值， 那为什么不考虑使用`Map<Integer, E>`呢。这种实际上还可能更高效一点，还能潜在的更准确的满足你得应用需要。

虑及到空对象这种情况，虽然不一定常常用到，但也时有偶为。例如，为一个枚举类型添加一个表示空含义的常量，就像`java.math.RoundingMode`用`UNNECESSARY`来表示不需作舍入值，如果设定为该值，则抛一例异常而出。

如果真的需要`null`值，但又用到了不能处理`null`的集合实现，可尝试使用多种不同的实现，亦如使用`Collections.unmodifiableList(Lists.newArrayList())`来代替`ImmutableList`。

可选之(Optional)

在许多场景中，开发者用`null`来表示某种不存在的含义： 或许是，本应该有值，但却为空。 亦或许是，本应有一个值，但却无其容值之处(not be found)。似如针对key调用`Map.get`方法返回`null`这种情况。

`Optional<T>`可用一个非空值来替换一个可为空的T类型引用。一个`Optional`或包含一个非空的`T`引用(这种场景称之为present，即表示有值)， 或不包含任何值(这种场景称之为absent，表示缺乏值，即不存在值)。我们不会表述为Optional包含null值。

```
Optional<Integer> possible = Optional.of(5);
possible.isPresent(); // returns true
possible.get(); // returns 5
```

即使`Optional`与其他程序环境中的`option(选项)`，`maybe(或许)`有些类似，但它却不具有直接的相似性。

我们再次列举一些`Optional`的最常用的操作。

创建`Optional`

`Optional`的这类方法都是静态的。

方法	描述
`Optional.of(T)`	创建一个包含非null值的Optional对象，如果给定的值是null,则抛出异常
`Optional.absent()`	返回某类型的不存在的Optional对象
`Optional.fromNullable(T)`	根据可空的引用，转换成一个Optional对象。如果值为非空，则作为present, 反之作

	为absent

查询方法

针对特定值的`Optional`，这类方法都是非静态的。

方法	描述
`boolean isPresent()`	如果当前Optional包含非空实例，则返回`true`
`T get()`	返回Optional包含的T类型非空实例，如果为空，则抛出`IllegalStateException`
`T or(T)`	返回Optional包含的T类型实例，如果为空，则返回默认值
`T orNull()`	返回Optional包含的T类型实例, 如果为空，则返回`null`。对应的方法为`fromNullable`
`Set<T> asSet()`	返回包含有Optional值的不可变的单例`Set`对象，如果没有值，则返回空`Set`

除了这些方法，`Optional`还提供有很多便利的工具方法，详细请参见Javadoc。

Optional的意义

Optional除了赋予`null`更好的可读性以外，最大的益处是表意更加简明。因为它可以迫使你思考有`null`出现的情况，然后从Optional中取出相应的值，并处理它。
我们常常会忽略甚至遗忘对null值得处理，虽然可以借助FindBugs来帮助审查，但是我们认为这却不是处理null问题最好的方式。

这种情况在返回值可能为空的场景中显得尤为相关，在实现`other.method(a, b)`方法时, 你(或其他人)可能更会遗忘返回值为空的情况，相较之下，不会很容易忽略`a`为`null`。

返回`Optional`就可以避免这种情况，因为你会手动解开Optional，并取出其中包含的值，然后才可以继续你的代码。

便捷方法

当你想用一些默认值来替换`null`时， 可使用`Objects.firstNonNull(T, T)`，如该方法名之意，是不允许两个参数都为`null`， 否则会抛出`NullPointerException`. 如果使用了Optional， 就有可选的方案，如`first.or(second)`.
在`Strings`这个类中，我们提供了很多用以处理空字符串的方法，具体来说，我们赋予的方法名称形如如下：

方法
`emptyToNull(String)`
`isNullOrEmpty(String)`
`nullToEmpty(String)`

我们想要强调的是，这些方法主要是应对忽视空字符串与null字符串的API接口。如果你写代码时，每次都把`null`字符串和空字符串(`empty`)混为一谈，Guava团队对此将是黯然挥泪，因为我们也无可奈你何。(`null`与`empty`字符如果不作含义上的区别，这种无疑是一种令人惴惴不安的代码坏味道)