链滴

Kotlin 实战 | 2 = 12？泛型、类委托、重载运算符综合应用

作者：lzlyy

原文链接：https://ld246.com/article/1597027088418

来源网站：链滴

这是该系列的第八篇，系列文章目录如下：

1. Kotlin基础 | 白话文转文言文般的Kotlin常识
2. Kotlin基础 | 望文生义的Kotlin集合操作
3. Kotlin实战 | 用实战代码更深入地理解预定义扩展函数
4. Kotlin实战 | 使用DSL构建结构化API去掉冗余的接口方法
5. Kotlin基础 | 抽象属性的应用场景
6. Kotlin进阶 | 动画代码太丑，用DSL动画库拯救
7. Kotlin基础 | 用约定简化相亲
8. Kotlin基础 | 2 = 12？泛型、类委托、重载运算符综合应用

项目应用中通常会对 SharedPreference 再封装一层，使用 Kotlin 语法糖可以极大简化这层封装。文将使用委托、泛型、重载运算符来达到化简的目的。

SharedPreference 在项目中的封装类通常如下所示：

```
public final class SPUtils {
    //'持有SharedPreferences实例'
    private SharedPreferences sp;

    //'在构造函数中构建SharedPreferences实例'
    private SPUtils(final String spName) {
        sp = Utils.getApp().getSharedPreferences(spName, Context.MODE_PRIVATE);
    }

    //'写函数'
    public void put(@NonNull final String key, final int value, final boolean isCommit) {
        if (isCommit) {
            sp.edit().putInt(key, value).commit();
        } else {
            sp.edit().putInt(key, value).apply();
        }
    }

    //'读函数'
    public int getInt(@NonNull final String key, final int defaultValue) {
        return sp.getInt(key, defaultValue);
    }
    ...
}
```

SharedPreferences 共支持 6 种数据类型的读写，加起来一共 12 个读写函数。

类委托

用 Kotlin类委托语法，可以将 SharedPreferences 工具类声明如下：

```
//'将类委托给 SharedPreferences 实例'
class Preference(private val sp: SharedPreferences) : SharedPreferences by sp {
    ...
```

```
}
```

Preferen继承接口 SharedPreferences，这样做的目的是对业务层保留 SharedPreferences原有接。并把构建 SharedPreferences实例移出工具类，交给业务层处理。

class Class2(private val c1: Class1) : Class1 by c1这个语法叫**类委托**，它的意思是：Class2是 Class1子类型并且 Class2将所有接口的实现委托为 c1实例，这样 Kotlin 会自动生成所有接口并将其实现委给超类型的实例。这就是为啥 Preferen并未实现接口中的任何一个函数，编译器却不报错，打开 Kotln 字节码验证一下：

```
public final class Preference implements SharedPreferences {
  private final SharedPreferences sp;

  public Preference(@NotNull SharedPreferences sp) {
    Intrinsics.checkParameterIsNotNull(sp,  'sp' );
    super();
    //'依赖注入'
    this.sp = sp;
  }

  public boolean getBoolean(String key, boolean defValue) {
    //'将实现委托为sp实例'
    return this.sp.getBoolean(key, defValue);
  }

  public float getFloat(String key, float defValue) {
    //'将实现委托为sp实例'
    return this.sp.getFloat(key, defValue);
  }
  ...
}
```

Kotlin 自动将所有接口的实现委托给了 SharedPreferences 实例。by关键词使我们可以省去这些模代码。

重载运算符

为了简化读写函数的调用，重新定义了两个函数：

```
class Preference(private val sp: SharedPreferences) : SharedPreferences by sp {
   //'写函数'
   operator fun set(key: String, isCommit: Boolean = false , value: Int) {
       val edit = sp.edit()
       edit.putInt(key,value)
       if (isCommit) {
           edit.commit()
       } else {
           edit.apply()
       }
   }

   //'读函数'
   operator fun get(key: String, default: Int): Int = sp.getInt(key,default)
}
```

这两个函数都以保留词 operator开头，它表示**重载运算符**，即重新定义运算符的语义。Kotlin 中预义了一些函数名和运算符的对应关系，称为**约定**：

函数名	运算符
plus	a+b
times	a*b
div	a/b
mod	a%b
minus	a-b
unaryPlus	+a
unaryMinus	-a
not	!a
inc	++a,a++
dec	--a,a--
get	a[b]
set	a=b
rangeTo	..

这次用到的是约定是 get和 set，声明他们时可以定义任意长度的参数，分别以 2 参数和 3 参数为例：

java	kotlin
p.get(key, default)	p[key, default]
p.set(key, true, value)	p[key, true] = value

其中 set函数中最后一个参数是 =右边的值，其余参数是 =左边的值。

然后就可以像这样简洁地读写了：

```
class Activity1 : AppCompatActivity() {

    private lateinit var pre: Preference

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //'构建Preferences实例'
        pre = Preference(getSharedPreferences(" test ", Context.MODE_PRIVATE))
        //'写'
        pre[" a "] = 1
        //'读'
        pre[" a ",0]
    }
}
```

但只定义了 Int 值的读写，难道其余的类型都要新起一对读写函数?

泛型

泛型是类型的类型，使用泛型就可以让读写函数与具体类型解耦：

```
class Preference(private val sp: SharedPreferences) : SharedPreferences by sp {
    //'将写函数的数据类型抽象为T'
    operator fun <T> set(key: String, isCommit: Boolean = false , value: T) {
        with(sp.edit()) {
            //'将泛型具体化，并委托给sp实例'
            when (value) {
                is Long -> putLong(key, value)
                is Int -> putInt(key, value)
                is Boolean -> putBoolean(key, value)
                is Float -> putFloat(key, value)
                is String -> putString(key, value)
                is Set<*> -> (value as? Set<String>)?.let { putStringSet(key, it) }
                else -> throw IllegalArgumentException("  unsupported type of value  ")
            }
            if (isCommit) {
                commit()
            } else {
                apply()
            }
        }
    }

    //'将读函数的数据类型抽象为T'
    operator fun <T> get(key: String, default: T): T = with(sp) {
        //'将泛型具体化，并委托给sp实例'
        when (default) {
            is Long -> getLong(key, default)
            is Int -> getInt(key, default)
            is Boolean -> getBoolean(key, default)
            is Float -> getFloat(key, default)
            is String -> getString(key, default)
            is Set<*> -> getStringSet(key, mutableSetOf())
            else -> throw IllegalArgumentException("  unsupported type of value  ")
        } as T
    }
}
```

其中的 with、as、when、is的详细解释可以翻阅该系列前面的文章。

然后就可以像这样无视类型地使用读写函数了：

```
class DelegateActivity : AppCompatActivity() {

    private lateinit var pre: Preference

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        pre = Preference(getSharedPreferences("  test  ", Context.MODE_PRIVATE))
        //'写int'
        pre["  a  "] = 1
        //'使用commit方式写String'
```

```
        pre[" b ",true] = " 2 "
        //'写Set<String>'
        pre[" c "] = mutableSetOf(" cc ", "dd" )
        //'读String'
        pre[" b "]
    }
}
```