

## JAVA 编码规范

### 版本历史

版本/状态	责任人	起止日期	备注
V1.0	曾东彪 仇宏磊	2017-9-11	由其它 JAVA 编码规范整理而成。

## 摘要

本文主要用于对java编程中的命名、排版、注释、空白使用、日志使用进行规范，为java编程指代指引。

**关键字：** java，规范，命名，排版，注释

## 目 录

摘要 .....	II
<b>1. 概述 .....</b>	<b>1</b>
1.1 目的 .....	1
1.2 范围 .....	1
1.3 说明 .....	1
<b>2. 命名规范 .....</b>	<b>1</b>
2.1 文件命名 .....	1
2.2 包命名 .....	1
2.3 类和接口命名 .....	1
2.4 方法命名 .....	2
2.5 变量命名 .....	3
2.6 常量命名 .....	3
<b>3. 缩进排版 .....</b>	<b>3</b>
3.1 行长度 .....	3
3.2 换行 .....	3
<b>4. 注释 .....</b>	<b>4</b>
4.1 开头注释 .....	4
4.2 文档注释 .....	4
4.3 块注释 .....	4
4.4 单行注释 .....	5
4.5 行尾注释 .....	5
4.6 其它注释规范 .....	5
<b>5. 声明 .....</b>	<b>5</b>
5.1 变量声明 .....	5
5.2 类和接口声明 .....	6
<b>6. 语句 .....</b>	<b>6</b>
6.1 包和引入语句 .....	6
6.2 简单语句 .....	6

6.3	复合语句 .....	6
6.4	返回语句 .....	7
<b>7.</b>	<b>空白 .....</b>	<b>7</b>
7.1	空白使用规范 .....	7
7.2	空格使用规范 .....	8
<b>8.</b>	<b>其它 .....</b>	<b>9</b>
8.1	序列化 .....	9
<b>9.</b>	<b>日志 .....</b>	<b>10</b>
9.1	日志级别 .....	10
9.2	编码 .....	10
<b>10.</b>	<b>IDEA 配置 .....</b>	<b>10</b>
10.1	INTELLIJ IDEA .....	10
10.1.1	设置 user name .....	10

## 1. 概述

### 1.1 目的

1. 规范 Java 编程过程中的命名和代码书写规范。
2. 改善代码可读性，提高代码质量。
3. 增强代码的可维护性，降低运维难度。

### 1.2 范围

网络研发部SMB课EAP Controller全体开发人员。

### 1.3 说明

1. 为了执行规范，每位同事必须一致遵守编码规范。
2. 本编程规范建立在标准的 JAVA 编程规范的基础上，如和标准的 JAVA 编程规范有冲突，以本编程规范为准。

## 2. 命名规范

命名规范使得代码更易理解，可读性更强。并且能够提供函数和标识符的信息。

### 2.1 文件命名

文件类型	文件后缀
Java源文件	.java
Java字节码文件	.class
配置文件	.xml、.properties

### 2.2 包命名

包名采用域后缀倒置的加上自定义的包名，采用小写字母，都应该以 `com.tp-link` 开头(不包括一些特殊原因)，接下来的部分使用项目名和模块名。

格式：`com.<company name>.<project name>.<module name>.<sub module name>;`

示例：`package com.tp-link.cloud.dispatcher.server;`

### 2.3 类和接口命名

类和接口的名称应是一个名词，采用大小写混和的方式，所有单词都应紧靠在一起，其中每个单词的首字母应大写。接口命名以 `I(Interface)`开头来标记。类名、接口名应当体现其主要职责。如果一个类完成多个职责，应考虑拆分为多个类。如果一个接口定义了多个职责，应考虑拆分为多个接口。

示例:

```
public class DeviceInfo;
public interface IDeviceService;
```

反例:

```
public class CloudUser {
    private final int OWNER = 0;
    private final int USER = 1;
    private int userRole = 0;
    private String userName = null;
    public CloudUser(int userRole, String userName) {
        super();
        this.userRole = userRole;
        this.userName = userName;
    }
}
```

应当拆为 Ower 和 User 两个类。

## 2.4 方法命名

方法应该是动词或者动名结构，采用大小写混和的方式，其中第一个单词的首字母用小写，其后单词的首字母大写，类似驼峰结构。方法名字在清楚描述本方法功能的前提下，越简短越好。一个方法只完成一件事件，不应当出现 doXAndDoY 这样的方法。

示例:

```
public String getName();
public Response getDeviceStatus(String deviceId);
```

反例 1:

```
public void getUserAndPassword();
```

建议改为

```
public UserInfo getUserInfo();
```

反例 2:

```
public void process(boolean doA) {
    if (doA) {
        // Process A;
    } else {
        // Process B;
    }
}
```

建议改为

```
public void doA();
public void doB();
```

## 2.5 变量命名

变量命名一般采用大小写混和的方式，第一个单词的首字母小写，其后单词的首字母大写，变量名的首字母不能为”\_”或者”\$”。

变量名应该尽可能的短小，但要有意义，尽可能做到见名知意。除了约定俗成的缩写（如 message 缩写为 msg，develop 缩写成 dev）之外，不要随意使用缩写。除了临时变量外（一般用于循环变量），应该避免使用只有一个字母的变量名。

示例：

```
private String deviceName=null;
```

反例：

```
private String dn = null; // private String deviceName = null;
private boolean flag = false; // private boolean getUsernameSucceed = false;
```

## 2.6 常量命名

常量名使用全大写的英文描述，英文单词之间用下划线分隔开，并且使用 static final 修饰。特别中，对于类中定义的数字常量，一定要定义一个常量，不要到处使用数字，造成魔鬼数字问题和随处修改的问题。

示例：

```
public static final int MAX_VALUE = 1000;
```

## 3. 缩进排版

4 个空格作为缩进排版的一个单位，每一级缩进都要在上一级基础上缩进一个单位。**不允许使用 TAB 键进行缩进。**可在编辑器(Eclipse, vim 等)中，设置扩展 TAB 为 4 个空格。

### 3.1 行长度

一行的长度(包括注释)不能超过 120 个字符，超过 120 个字符必须换行。

### 3.2 换行

当一个表达式无法容纳在一行时，可以依据如下规则进行断开：

1. 在一个逗号后面断开；

```
@Override
public boolean isCancelled() {
    Object value = this.value;
    return value != null && value instanceof ThrowableHolder
        && ((ThrowableHolder)value).cause instanceof CancellationException;
}
```

2. 在一个操作符前后断开；

```
Logger.info("error:[{}], " + "errorInfo:[{}], "
```

```
+ "deviceInfo:[deviceId:{}, mac:{}, deviceName:{}, alias:{}, "  
+ "fwId:{}, fwVer:{}, hwId:{}, hwType:{}, hwVer:{}, ip:{}]",  
e.getErrCode().getMessage(), e.getInfo(), deviceId, deviceMac, deviceName,  
alias, fwId, fwVer, hwId, deviceModel, deviceHwVer, ipAddress);
```

3. 在复杂的语句中，在开始下一个子表达式的开头断开，而不是在一个表达式的中间断开；

```
public static boolean isValidLocalHost(String host) {  
    return host == null || host.length() == 0 || host.equalsIgnoreCase("localhost")  
        || host.equals("0.0.0.0") || (LOCAL_IP_PATTERN.matcher(host).matches());  
}
```

4. 新的一行应该与上一行同一级别表达式的开头处对齐。

## 4. 注释

### 4.1 开头注释

所有的源文件都应该在开头有一个 C 语言风格的注释，其中列出版权信息，作者和修改日期。便于他人阅读时找到源文件出处和更新时间：

```
/**  
 * Copyright (c) 2014, TP-Link Co.,Ltd.  
 * Author: zhangsan <zhangsan@tp-link.net>  
 * Created: 2014-5-15  
 * Author : lisi <lisi@tp-link.net>  
 * Reason: code merge  
 * Updated: 2015-3-4  
 */
```

### 4.2 文档注释

文档注释描述 Java 的源文件或类提供的主要功能，有必要的话在其中添加使用示例。每个文档注释放在文档注释符/\*...\*/中。

每个类只在声明的地方之前有一个文档注释。

示例：

```
/**  
 * Service manager, router request by methodName to different serviceInvokers.  
 */  
public class DispatchServiceManager {
```

注意：文档注释第一行（/\*\*）不需缩进，其后文档注释每一行都缩进 1 格。

### 4.3 块注释

块注释主要用于描述：文件、方法、数据结构和算法，块注释应该用一个空行开头，以便于代码部分区分开来。

块注释举例：



```
/**
 * Returns <tt>true</tt> if this map contains no key-value mappings.
 * @return <tt>true</tt> if this map contains no key-value mappings
 */
public boolean isEmpty() {
```

## 4.4 单行注释

比较短的注释可以放在一行中，但必须与其后的代码有相同的缩进。单行注释举例：

```
private AtomicLong msgIdCounter = null; // Used to generate request uuid in cloud system
```

## 4.5 行尾注释

注释标记“//”能够注释一行或者该行由“//”开始直到行尾的部分。通常写在一行结尾，或者在一行代码之前单起一行进行注释。如：

```
// Init zookeeper standard url.
URL zkUrl = new URL(CommonUtils.getZkAddress(config.getZookeeperAddress()));
```

## 4.6 其它注释规范

1. 尽量在注释中不使用中文；
2. 源程序注释量必须在 20%以上；

# 5. 声明

## 5.1 变量声明

1. 每行定义的变量数目必须有且只有一个，有利于写注释；**请尽量在声明处直接完成初始化，即赋一个默认值**（构造函数中完成初始化的 final 域除外），在任意一个 checkstyle 的检查工具中，没有初始化的声明都是不被允许的。虽然 JVM 在编译阶段会为没有初始化的变量赋值，但不同版本的 JVM，其表现并不完全相同，不要依赖于 JVM。

```
int msgId = 0; // messageId;
int size = 0; // message size;
```

2. 声明局部变量的时候必须初始化。禁止局部变量与外层变量同名，导致因作用域范围引起冲突。也就是说不要在内部块中声明一个与外部块某个变量同名的变量。
3. for 循环里的循环变量应在 for 语句里面定义；

```
for (int index = 0; index < args.length; index++) {
    循环变量之外的变量，应当在循环外面定义。
```

4. 数组的定义：数据的[]应放在类型名的后面，而不是变量名的后面。

```
String[] names = new String[3];
```

5. 如非必要，不要使用静态变量及匿名对象。

## 5.2 类和接口声明

类和接口的声明应该遵循以下规范：

1. 在方法名和参数列表的圆括号以及括号后的第一个参数间都没有空格。如：

```
public static void main(String[] args) {
```

2. 开括号“{”必须与声明语句放在同一行。如：

```
public class ServerStarter {
```

3. 闭括号“}”必须与声明语句有相同的缩进格式。

4. 各个方法之间要用一个空行隔开。

5. 如非必要，不要定义静态方法。

## 6. 语句

### 6.1 包和引入语句

在 Java 源文件中，第一个非注释行是包语句，在它之后可以跟进引入语句。示例：

```
package com.tplink.cloud.dispatcher.server;
```

```
import java.io.File;
```

Import 引入的顺序可以按 jdk 自带、第三方、自己开发顺序书写。

只引入需要用到的包，不要引入未使用的包，**不要使用 `import java.util.*` 这样的引入方式。**

**不要使用静态引入。**

### 6.2 简单语句

每行最多包含一个语句。例如：

```
index++;
```

### 6.3 复合语句

复合语句指使用大括号括起来的一串语句。

1. 大括号中的语句比组合语句多一级缩进。
2. 开括号“{”应该放在组合语句前的语句末尾。闭括号“}”应该放在新的一行并与组合语句开始前的第一个语句有相同的缩进。起始开括号前要有空格。

如：

```
while (true) {  
    // do something ...  
}
```

3. 如果语句是控制语句的一部分时，所有的语句都要用大括号围起来，即使只有一个语句也要用括号，例如在 if-else 或者 for 语句中。这样避免在添加语句时忘记添加括号而导致程序产生 bug。
4. 组合语句的层次不要太深，嵌套的遍历不要超过三层，否则应对逻辑进行拆分和封装。

## 6.4 返回语句

return 语句在有返回值时不需要使用圆括号，除非使用圆括号在某些特定的情况下能够提高代码的可读性。

示例：

```
return;  
return list.size();  
return (size ? size : defaultSize);
```

## 7. 空白

### 7.1 空白使用规范

1. 在两个方法定义之间：

```
public void start() {  
}  
  
public void stop() {  
}
```

2. 包的声明、包引入和类声明之间

```
package com.tplink.cloud.dispatcher.server.dispatch;  
  
import com.tplink.cloud.rpc.direct.GenericRequest;  
  
/**  
 * Interface for dispatcher to to call AssembleService, AccountService, DeviceService,  
 * etc.  
 */  
public interface IInvoker {
```

3. 变量的声明和方法之间：

```
private ReadWriteLock rwlock = new ReentrantReadWriteLock();  
  
public AddressProxy(RpcChannelManager channelManager, Strategy strategy) {
```

4. 在一个语句块或者单行注释之前：

```
private ReadWriteLock rwlock = new ReentrantReadWriteLock();  
  
public AddressProxy(RpcChannelManager channelManager, Strategy strategy) {
```

5. 为了提高程序可读性，在一个方法的两个逻辑段之间。

## 7.2 空格使用规范

1. 各种控制语句的关键字之后，使用空格，包括但不限于 if/else、switch、for、while、do/while、try/catch/finally、synchronized、throw、带返回值的 return、assert；

```
if (size < elementData.length) {
    elementData = Arrays.copyOf(elementData, size);
}
```

2. 语句块开始之前，使用空格。包括类定义类名之后的第一个大括号之前，方法定义的方法名之后的第一个大括号之前，static 块的 static 之后的第一个大括号之前；

```
static {
    try {
        config = new PropertiesConfiguration(CONFIG_FILE_NAME);
    } catch (Exception e) {
        throw new RuntimeException("Configuration loading error: "
            + "configFileName: " + CONFIG_FILE_NAME, e);
    }
}
```

3. 类定义、方法定义、数组初始化时，逗号分隔若干个元素，逗号之后，使用空格。包括分隔类实现的多个接口的逗号之后；分隔方法的多个参数的逗号之后；分隔方法抛出的多种异常的逗号之后；数组初始化时，分隔多个数组元素的逗号之后。

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
public V put(K key, V value)
public Map<String, String> load() throws ParserConfigurationException, IOException
```

4. 分隔 for 语句的三个元素的两个分号之后，使用空格。

```
for (int index = 0; index < nodeList.getLength(); index++) {
```

5. 方法的数组参数的省略号之后，使用空格。

```
public Long del(final String... keys)
```

6. 除了成员运算符（.）以外，所有的二元运算符应该用空格和操作数分开；在一元运算符（如：++、--、负数-）和操作数之间不用空格；例如：

```
cosC = (a * a + b * b - c * c) / (2 * a * b);
while (count < 10) {
    count++;
}
print("Size is" + size + "\n");
```

## 8. 其它

### 8.1 序列化

实现了 `java.io.Serializable` 接口的类中，必须显式的指定序列化 ID（如果 IDE 支持生成随机，就使用随机数，解决类名相同可能引发的问题，如果不支持，使用 1L）。

```
private static final long serialVersionUID = 1L
```

#### ➤ 序列化的适用场景

1. 使用 RMI 进行远程调用；
2. 使用套接字在网络上传输对象；
3. 保存内存中的对象到文件或者数据库。

#### ➤ 序列化的本质是，以特定的格式存储对象数据

1. 序列化文件=魔鬼数字头+序列化格式版本+对象列表；
2. 序列化对象=类描述+数据域；
3. 类描述=类名+类序列化的版本唯一 ID（指纹）+描述序列化方法的标志集+数据域描述。

#### ➤ 类指纹的生成规则

通过对类、超类、接口、域类型和方法签名按规范方式排序，然后应用安全散列算法（SHA）得到 20 字节的数据包，再截取其前 8 个字节作为类的指纹。

#### ➤ 类指纹在反序列化中的作用

反序列化时，读入一个对象后，会拿其指纹与它所属类的当前指纹进行对比，如果不匹配，将产生异常。

#### ➤ 类指纹与版本管理

为使用老版本的数据，需要直接指定类指纹（`serialVersionUID`），而不使用默认的 SHA 生成的指纹。如果指定了类指纹，在进行反序列化时，序列化程度将尽量尝试恢复老版本数据到本地版本，由应用程序保证反序列化结果的正确使用。

#### ➤ 序列化使用注意事项

1. 只对本地访问有意义的存储文件句柄、窗口句柄，不应当被序列化（`transient`）；
2. 序列化只有数据域类型标识，不识别 `static`、`final`、可见域关键字（`public/protected/private`），这意味着所有数据域在不同对象中都可以因被修改而不相同；
3. 序列化生成的所有对象都是全新的，在使用单例及枚举时需要格外注意；
4. 序列化是深拷贝，但其拷贝对象的效率远低于直接创建对象并拷贝数据域中的值。

#### ➤ 序列化使用指导

1. 除使用 RMI 进行远程调用外，尽量少使用序列化（额外的 CPU 和内存开销）；
2. 在网络上传输对象时，应优先考虑简化数据结构和自定义网络协议而不是使用序列化；

3. 优先使用数据库支持的数据结构，当需要以二进制数据形式保存数据到数据库时，对于简单对象（不包含对其它对象的引用），应当自定义方法将其转为二进制数组，而不是序列化；
4. 严禁为了实现对象拷贝而使用序列化。

## 9. 日志

### 9.1 日志级别

error > warn > info > debug > trace

### 9.2 编码

1. 在一个对象中通常只使用一个 Logger 对象，Logger 应该是 static final 的；

如：

```
private static final Logger logger = LoggerFactory.getLogger(CodisUtils.class);
```

2. 不允许抛出异常的同时打印堆栈，因为这样会多次打印堆栈，只允许打印一次堆栈；
3. 不允许出现 System print(包括 System.out.println 和 System.error.println)语句；
4. 不允许出现 printStackTrace。

## 10. IDEA 配置

### 10.1 IntelliJ IDEA

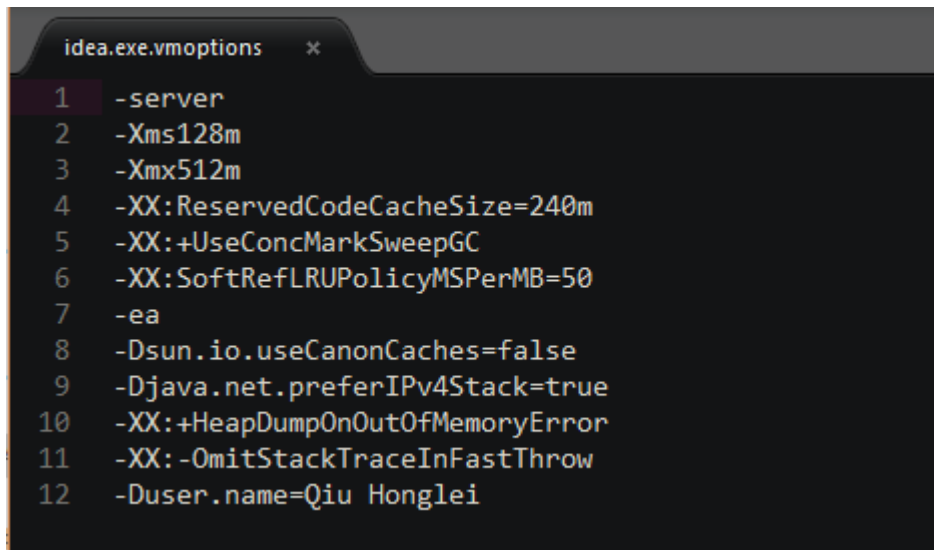
#### 10.1.1 设置 user name

为了将注释模板中的 USER 值设置位特定的代码编写人员名称，可以修改 IntelliJ 的配置文件：

修改 IntelliJ IDEA 安装目录下/bin/idea.exe.vmoptions 或/bin/idea64.exe.vmoptions（根据使用 32 位或 64 位 IntelliJ IDEA 进行修改），在文件末添加：

```
-Duser.name=*** **
```

如图：

A screenshot of a text editor window titled 'idea.exe.vmoptions'. The window contains a list of 12 JVM options, each preceded by a line number from 1 to 12. The options are: 1 -server, 2 -Xms128m, 3 -Xmx512m, 4 -XX:ReservedCodeCacheSize=240m, 5 -XX:+UseConcMarkSweepGC, 6 -XX:SoftRefLRUPolicyMSPerMB=50, 7 -ea, 8 -Dsun.io.useCanonCaches=false, 9 -Djava.net.preferIPv4Stack=true, 10 -XX:+HeapDumpOnOutOfMemoryError, 11 -XX:-OmitStackTraceInFastThrow, and 12 -Duser.name=Qiu Honglei.

```
1 -server
2 -Xms128m
3 -Xmx512m
4 -XX:ReservedCodeCacheSize=240m
5 -XX:+UseConcMarkSweepGC
6 -XX:SoftRefLRUPolicyMSPerMB=50
7 -ea
8 -Dsun.io.useCanonCaches=false
9 -Djava.net.preferIPv4Stack=true
10 -XX:+HeapDumpOnOutOfMemoryError
11 -XX:-OmitStackTraceInFastThrow
12 -Duser.name=Qiu Honglei
```

保存并重启 IntelliJ IDEA 进行测试即可。