12 添加注释的正确姿势

更新时间：2020-07-08 13:43:51

> 机会不会上门来找人，只有人去找机会。——狄更斯

1. 前言

《手册》 21 页，第八节 注释规约部分对注释规范的要点给出了比较全面的指导 1。

> 【强制】所有类都必须添加创建者和日期。
>
> 【强制】所有的枚举类型字段都必须有注释，说明每个数据项的用途。
>
> 【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等修改。
>
> 【参考】特殊标记，请注明标记人与标记时间。

我们要思考以下几个问题：

- 你平时写注释吗？
- 你知道注释的目的是什么？
- 有哪些好的注释范例？
- 为什么会有这些规定？
- 还有哪些好的规约？

本节将为你解答上述疑问。

2 注释的目的

注释的目的是：**辅助读代码的人员更快速的理解代码**。

因此我们写注释的时候不管使用何种规约和技巧都要围绕这个目的展开。

这就要求编写注释时，要能够**准确描述函数的功能，核心逻辑，潜在风险，注意事项等**。

如果注释写地好，即使过了很久自己可以通过注释快速理解代码，也可以帮助团队其他合作的成员快速理解自己的代码，快速找到相关文档，也将方便未来接手自己工作的开发人员。这也是一个优秀程序员专业性的一种体现。

3 常见的注释类型和写法

3.1 常规注释

常规注释主要指普通的注释，比如每个接口几乎都会有的：接口的功能，接口的参数以及含义，接口异常和出现异常的原因，接口的返回值。

首先我们从 JDK 代码注释中寻找灵感。

我们可以参考 `ThreadPoolExecutor#ThreadPoolExecutor` 构造函数的注释：

```
/**
 * Creates a new {@code ThreadPoolExecutor} with the given initial
 * parameters.
 *
 * @param corePoolSize the number of threads to keep in the pool, even
 *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
 * @param maximumPoolSize the maximum number of threads to allow in the
 *        pool
 * @param keepAliveTime when the number of threads is greater than
 *        the core, this is the maximum time that excess idle threads
 *        will wait for new tasks before terminating.
 * @param unit the time unit for the {@code keepAliveTime} argument
 * @param workQueue the queue to use for holding tasks before they are
 *        executed.  This queue will hold only the {@code Runnable}
 *        tasks submitted by the {@code execute} method.
 * @param threadFactory the factory to use when the executor
 *        creates a new thread
 * @param handler the handler to use when execution is blocked
 *        because the thread bounds and queue capacities are reached
 * @throws IllegalArgumentException if one of the following holds:<br>
 *         {@code corePoolSize < 0}<br>
 *         {@code keepAliveTime < 0}<br>
 *         {@code maximumPoolSize <= 0}<br>
 *         {@code maximumPoolSize < corePoolSize}
 * @throws NullPointerException if {@code workQueue}
 *         or {@code threadFactory} or {@code handler} is null
 */
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

可以看到该注释先给出了该构造函数的**功能说明**，然后对**每个参数的含义**进行解读，然后给出了**抛出的异常以及抛出异常对应的具体原因**。

正是 JDK 的注释非常专业和详细，才为我们学习源码提供了便利。试想如果没有注释，我们学习和理解源码的速度会不会更慢呢？

3.2 工具函数注释

工具类的注释主要包含：函数的功能，函数的使用范例，函数的参数和返回值描述，该函数出现的起始版本等。

我们选取 commons-lang3 的 `StringUtils` 类的 `StringUtils#isAnyEmpty` 函数的源码来学习工具函数的注释。

```
/**
 * <p>Checks if any of the CharSequences are empty ("") or null.</p>
 *
 * <pre>
 * StringUtils.isAnyEmpty((String) null)    = true
 * StringUtils.isAnyEmpty((String[]) null)  = false
 * StringUtils.isAnyEmpty(null, "foo")      = true
 * StringUtils.isAnyEmpty("", "bar")        = true
 * StringUtils.isAnyEmpty("bob", "")        = true
 * StringUtils.isAnyEmpty("  bob  ", null)  = true
 * StringUtils.isAnyEmpty(" ", "bar")       = false
 * StringUtils.isAnyEmpty("foo", "bar")     = false
 * StringUtils.isAnyEmpty(new String[]{})   = false
 * StringUtils.isAnyEmpty(new String[]{""}) = true
 * </pre>
 *
 * @param css  the CharSequences to check, may be null or empty
 * @return {@code true} if any of the CharSequences are empty or null
 * @since 3.2
 */
public static boolean isAnyEmpty(final CharSequence... css) {
  if (ArrayUtils.isEmpty(css)) {
    return false;
  }
  for (final CharSequence cs : css){
    if (isEmpty(cs)) {
      return true;
    }
  }
  return false;
}
```

除了前面提到的功能描述，参数和返回值描述外，该注释部分还给出了**常见的使用范例和执行结果**，能够帮助读者快速理解函数的用法。

强烈建议我们在编写工具类时参考这种写法，方便使用者的同时也体现了我们的专业水准。

3.3 废弃代码的注释

正如专栏的 ” 过期类、属性和接口的正确处理方式 “ 小节所讲的：废弃的接口要给出废弃的原因和替代方案等。

我们可以参考下面代码废弃函数的注释：

com.google.common.io.LittleEndianDataOutputStream#writeBytes

```
/**
 * @deprecated The semantics of {@code writeBytes(String s)} are considered dangerous. Please use
 *     {@link #writeUTF(String s)}, {@link #writeChars(String s)} or another write method instead.
 */
@Deprecated
@Override
public void writeBytes(String s) throws IOException {
  ((DataOutputStream) out).writeBytes(s);
}
```

该函数给出了废弃的原因：该函数比较危险。

给出了两个替代方案： `{@link #writeUTF(String s)}, {@link #writeChars(String s)}` 。

从这里我们学到，除了交代废弃的原因和替代方法外，还可以使用 `{@link}` 提供跳转到替代函数的快捷方式。

3.4 警告类注释

比如有很多程序员为了方便测试会写一个测试控制器，如 `TestController` ，来提供 HTTP 接口的控制器，预留一些“测试后门”，通常会有一个比较好的做法是放到某个特定测试分支，不会带到线上。

如：

- 提供查看项目的 apollo 配置项是否生效的接口。
- 提供查看 redis 数据的接口。
- 提供修复数据的接口。
- 提供某项功能的开关接口。
- 等

那么如果有些接口操作姿势 “非常特别” 或者 “非常危险”，一定要接口上加上注释，防止其他人员误触，导致故障。

如果某个函数仅供内部使用或者仅供某个功能使用，最好可以在注释上加上警示。

这些都极大降低沟通成本，极大降低团队其他成员犯错的几率。

3.5 特殊注释

开发中特殊注释如：TODO 注释和 FIXME 注释也非常常见。

TODO 注释主要用在本该做还没做的事项。

- 待斟酌函数的命名。
- 性能不佳，待后期优化。
- 开发过程某个功能使用前需要进行权限校验，但是权限校验依赖的新接口对方还没开发好。

此时可以加上 TODO 注释可以参考下面格式：

```
// TODO:: [xxx功能] 负责人：张三，事项：待添加权限验证逻辑，添加时间：2019-08-25 预计处理时间：2019-09-01
```

包含功能名称、责任人、事项、添加时间和预处理时间等信息。

我们看看 `com.google.common.io.Resources#getResource(java.lang.String)` 源码：

```
/**
 * Returns a {@code URL} pointing to {@code resourceName} if the resource is found using the
 * {@linkplain Thread#getContextClassLoader() context class loader}. In simple environments, the
 * context class loader will find resources from the class path. In environments where different
 * threads can have different class loaders, for example app servers, the context class loader
 * will typically have been set to an appropriate loader for the current thread.
 *
 * <p>In the unusual case where the context class loader is null, the class loader that loaded
 * this class ({@code Resources}) will be used instead.
 *
 * @throws IllegalArgumentException if the resource is not found
 */
@CanIgnoreReturnValue // being used to check if a resource exists
// TODO(cgdecker): maybe add a better way to check if a resource exists
// e.g. Optional<URL> tryGetResource or boolean resourceExists
public static URL getResource(String resourceName) {
  ClassLoader loader =
      MoreObjects.firstNonNull(
          Thread.currentThread().getContextClassLoader(), Resources.class.getClassLoader());
  URL url = loader.getResource(resourceName);
  checkArgument(url != null, "resource %s not found.", resourceName);
  return url;
}
```

其中注释的最后两行用到了 TODO 注释，该注释包含了责任人和修改思路。

因此如果有未来优化的思路时，可以通过 TODO 进行注释，在未来代码迭代时实现该注释的想法。

`com.google.common.escape.Escapers#wrap` 也有一个不错的范例：

```
private static UnicodeEscaper wrap(final CharEscaper escaper) {
  // 省略
    if (hiChars != null) {
    // TODO: Is this faster than System.arraycopy() for small arrays?
      for (int n = 0; n < hiChars.length; ++n) {
        output[n] = hiChars[n];
      }
    } else {
      output[0] = surrogateChars[0];
    }
  // 省略
}
```

这里表明作者还没有将两者性能进行对比，得到最佳选项。

FIXME 注释，主要用在某些出错代码处，一般是一些不能工作需要及时纠正的错误。

如编写了一处代码，其中部分代码涉及到了计算，但是自测时发现计算结果出错。此时可以参考下面的格式添加 FIXME 注释。在代码上线前一定要修复并验证好相关错误。

示例：

```
// FIXME:: [xxx功能] 负责人：张三，错误：计算错误，添加时间：2019-08-08 预计处理时间：2019-09-01
```

4. 为什么这么规定？

不知道大家有没有思考过这个问题： **为什么《手册》会有这些规定？**

我想这么做的最主要原因是为了帮助读代码的人员快速理解代码。

下面选取几条进行解读：

【强制】方法内部单行注释，在被注释语句上方另起一行，使用 // 注释。方法内部多行注释
使用 // 注释，注意与代码对齐。

方法内部单行注释，在被注释的语句上方另起一行。主要体现了整体思维，也是为了实现 ”代码意群“效应，从视觉上让注释和下面的代码更接近。

【强制】 所有的类都必须添加创建者和创建时间。

类添加了创建者，读者就可以知道第一个创建该类的人（一般是最熟悉的人）是谁，遇到问题可以找他核实。

类添加了创建时间，有助于阅读此代码的人更方便地了解类的编写时间。

另外在这里给出一个技巧：如果我们使用的是 IDEA，并用 GIT 进行代码版本管理，可以在编辑器的左侧行数附近，右键选择 “Annotate”， 可以查看某行代码修改的人和时间。

如果你对该部分代码有疑问，可以快速定位到修改的人和修改时间，对我们协调和解决问题有极大的帮助。

5 补充

【强制】如果代码逻辑和注释不符，必须进行修改

代码逻辑和注释不符，容易让使用者误用，增加出错的概率，容易造成返工降低开发效率。

通常由于开发者理解有误，偶尔会写出了误导性注释，如果发现这类问题一定要认真核实，如果确认是误导性注释，一定要及时修改，避免团队其他成员重复趟坑。

【推荐】 TODO 注释要加上功能名称

为什么特殊注释要加上功能名称？

通常我们会有很多项目的 TODO 注释，但是最常遇到的需求是快速定位正在开发的某个功能的 TODO 注释或者其他某个想修改的功能的 TODO 进行修改。此时如果 TODO 较多且没有标注功能名称，要想找到自己要修改的 TODO 项，通常需要通过搜索自己的姓名来查找，如果 TODO 较多查找起来将非常耗时。

【推荐】方法注释中建议添加相关需求文档，接口文档地址。

很多公司都会有接口平台，开发人员可以将 Dubbo 或 HTTP 接口传到接口平台中，从而得到访问链接，方便前后端对接。

建议将上传的 Dubbo 和 HTTP 接口文档地址顺手加入到注释中，避免每次需要使用时都要手动搜索。

```
/**
 * xxx功能（功能描述）
 *
 * 需求文档：{@link <a href="http://doc.imooc.com/xxx/process/0001"/>}
 * 接口文档：{@link <a href="http://api.imooc.com/xxx/process/0001"/>}
 * 对接人员：@张三
 *
 * @param param 参数描述
 * @return 返回值描述
 */
public Object something(Object param){
    // 1. 查询xx数据

    // 2. 过滤yy条件

    // 3. 组装结果
}
```

尤其是对依赖的三方 / 二方接口的封装，大家可以将此接口的相关文档和负责人添加到注释中。

后面自己也可能经常需要找接口的文档链接，开发过程中遇到问题也可及时和对接人沟通，这将极大提供工作效率。

这是一条非常值得推荐的技巧，这种注释风格非常能够体现出一个人的专业素养。

【推荐】容易费解的地方一定要加注释。

自己某块代码的写法很诡异，一定要注明原因。

否则极有可能因为时间久远，后面自己再回头看，或者别人问你为什么这么写，自己都蒙圈了。

导致别人不敢乱改，自己也不敢改动的尴尬情况。

这将是一个非常大的隐患。

【推荐】推荐 git 提交注释的格式为： [功能名称] < 提交类型 > 修改点描述。

很多公司对 git 提交注释的格式有自己的要求， 但是很多公司没有规定，导致大家写的都很随意。

很多人提交的注释都是功能的描述，无法得知因哪个功能做的修改。

建议大家可以养成好的习惯，在提交的描述中增加功能名称，并且能够再添加修改的性质就更好了。

修改的性质包括：**新增、删除、修改、修复等**。

比如我们独立开发的一个功能，突然中间有一个提交没有带我们的功能名称或功能名称不对，我们可以及时感知到可能出现了问题。

比如我们很久之后发现之前自己对某个函数进行了修改，自己却忘记修改的目的，我们可以查看提交记录，根据注释快速了解到是由于哪个功能导致的修改。

正例：

```
[a功能] <add>  某某接口

[a功能] <delete> 删除了无用的注释

[a功能] <update>  修改函数命名

[a功能] <fix> 修复了某个错误
```

大家实践之后就会发现该规约的好处。

【参考】 利用 //----- 或 / ---- 分组 —/ 注释实现" 方法分组 "

`org.apache.commons.lang3.BooleanUtils` 工具类中就广泛应用了这种方式：

```
// Integer to Boolean methods
//-----------------------------------------------------------------------

// 各种整型转布尔类型的函数
```

再如 `java.util.HashMap` 中的方法分组注释:

```
/* ---------------- Static utilities -------------- */
```

通过方法分组的注释可以很好地实现函数的 " 分组 "，将类中功能相似的方法放在一起，并使用上述注释进行分割，是一个不错的技巧。

【参考】多写设计的目的，注意事项，不要写从代码显而易见的注释。

很多人喜欢写一些显而易见的注释，导致自己花费了时间对团队其他人却没太大帮助。

如果方法比较复杂，尽量写**设计的目的和注意的事项等更有帮助的内容**。

反例：

```
public Boolean isLegal() {
   // 如果在售或有库存或有敏感词则返回false
   if (isOnSell == null || hasStock == null || hasSensitiveWords == null) {
      return false;
   }
   return isOnSell && hasStock && (!hasSensitiveWords);
}
```

【参考】 可以将方法的核心逻辑拆分成多个步骤，关键步骤在函数内部可以加上注释并带上序号，之前空一行。

函数内的逻辑注释，将有助于我们养成任务拆解的思维，也有助于自己或团队其他成员快速理解编程的逻辑。

如果核心逻辑的关键步骤加上注释，当代码较长时可以快速帮助读代码的人理解。

这样当代码行数超过 80 行时，开发者也可以根据核心逻辑注释来拆分子函数。

即使不在核心步骤添加注释 (或提取子函数)，在核心步骤之间加上一个空格行，方便读者理解。

大家可以在每个大的步骤前加注释，也可以在核心逻辑前将注释分条列举。

可以参考 `java.util.concurrent.ThreadPoolExecutor#execute` 函数的注释：

```
/**
 * Executes the given task sometime in the future.  The task
 * may execute in a new thread or in an existing pooled thread.
 *
 * If the task cannot be submitted for execution, either because this
 * executor has been shutdown or because its capacity has been reached,
 * the task is handled by the current {@code RejectedExecutionHandler}.
 *
 * @param command the task to execute
 * @throws RejectedExecutionException at discretion of
 *         {@code RejectedExecutionHandler}, if the task
 *         cannot be accepted for execution
 * @throws NullPointerException if {@code command} is null
 */
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    else if (!addWorker(command, false))
        reject(command);
}
```

6. 总结

本节如果你只记一句话那就是：**注释的目的是让读者更快理解代码的含义** 。注释的其他规约都是围绕这一点展开的。

本节讲述了注释的目的，并结合实际的开发经验对注释相关规约进行了解读和补充。

编写恰当的注释是一个程序员专业性的体现，希望大家在编程中能够严格要求自己，能够认真实践好的注释规范，提高开发效率，少趟坑。

下一节将讲述变长参数的奥秘。

参考资料

阿里巴巴与 Java 社区开发者.《 Java 开发手册 1.5.0》华山版. 2019. 21 □□

}

11 ArrayList的subList和Arrays的asList学习

13 你真得了解可变参数吗?