日志代码编写规范

深圳微品致远信息科技有限公司

所属项目:	所有 Java 项目
文件类别:	日志代码编写规范
版 本 号:	V1.0.0
编 写 者:	邓冰寒
审 核 者:	
批 准 者:	

文档历史记录

版本	更新要点	日期	作者
V1.0.0	1.　初版	2017-01-03	邓冰寒

目录

目标

一般来说日志分为两种：业务日志和异常日志，使用日志我们希望能达到以下目标：

1. 对程序运行情况的记录和监控；
2. 在必要时可详细了解程序内部的运行状态；
3. 对系统性能的影响尽量小；
4. 没有副作用，不会影响程序本身的使用
5. 使用简单，不会比 print 复杂

日志框架

Logback

Logback 是由 log4j 创始人设计的又一个开源日记组件。logback 当前分成三个模块：logback-core,logback- classic 和 logback-access。logback-core 是其它两个模块的基础模块。logback-classic 是 log4j 的一个改良版本。此外 logback-classic 完整实现 SLF4J API 使你可以很方便地更换成其它日记系统如 log4j

日志分级

Java 的日志框架一般会提供以下日志级别，缺省打开 info 级别，也就是 debug，trace 级别的日志在生产环境不会输出，在开发和测试环境可以通过不同的日志配置文件打开 debug 级别。

1. error - 其他错误运行期错误；
2. warn - 警告信息，如程序调用了一个即将作废的接口，接口的不当使用，运行状态不是期望的但仍可继续处理等；
3. info - 有意义的事件信息，如程序启动，关闭事件，收到请求事件等；
4. debug - 调试信息，可记录详细的业务处理到哪一步了，以及当前的变量状态；
5. trace - 更详细的跟踪信息；

在程序里要合理使用日志分级

日志编码规范

1. 在每 Java 实现类上添加注解`@Slf4j` 即可以打印日志，无需实例化变量 `log`。要支持@Slf4j 注解，先安装 Intellij Idea 插件 lombok，可以打开 File->Settings ->Plugins->Browse Repositories,搜索 lombok 并安装。如果由于网络问题安装失败，也肯于直接打开浏览器，进入 https://plugins.jetbrains.com/ 搜索关键字 lombok，下载 Idea 对应的版本，进入 File->Settings->Plugins->Install Plugin from disk，找到你下载的插件包（如：lombok-plugin-0.13.16.zip）即可。

```
@Slf4j
public class MyServiceImpl implements MyService {
        public void foo() {
                log.info("foo method");
        }
}
```

2. 输出 Exceptions 的全部 Throwable 信息，因为 log.error(msg)和 log.error(msg, e.getMessage())这样的日志输出方法会丢失掉最重要的 StackTrace 信息。

```
try {
        // do something ...
} catch (Exception e) {
        log.error("Error: ", e);
}
```

3. 不允许记录日志后又抛出异常，因为这样会多次记录日志，只允许记录一次日志。

```
public void foo() throws LogException {
        try {
                // do something ...
        } catch (Exception e) {
                log.error("Bad things ... ", e);
                throw new LogException("Bad things ...", e);
        }
}
```

4. 不允许出现 System print(包括 System.out.println 和 System.error.println)语句。

```

System.out.println("print something ..."); //  错误，不允许
```

```
log.debug("print something ..."); //  正确
```

5. printStackTrace 仅作临时调试用，但不允许提交到 Gitlab 上。

```
try {
        // do something ...
} catch (Exception e) {
        e.printStackTrace();        // 错误，不允许
        log.error("Error: ", e);        // 正确
}
```

6. 日志性能的考虑，如果代码为核心代码，执行频率非常高，则输出日志建议增加判断，尤其是低级别的输出<debug、info、warn>。

debug 日志太多后可能会影响性能，有一种改进方法是：

```
if (log.isDebugEnabled()) {
        log.debug("print something ...");
}
```

Slf4j 提供了 parameterized logging：

```
log.debug("input value: {}, output value: {}", inVal, outVal);
```

一方面可以减少参数构造的开销，另一方面也不用多写多行代码。

7. 有意义的日志

通常情况下在程序日志里记录一些比较有意义的状态数据：程序启动，退出的时间点；程序运行消耗时间；耗时程序的执行进度；重要变量的状态变化。

附录：parameterized logging 最佳实践

parameterized logging

SLF4J supports an advanced feature called parameterized logging which can significantly boost logging performance for *disabled* logging statement.

For some Log `log`, writing,

```
log.debug("Entry number: " + i + " is " +
String.valueOf(entry[i]));
```

修改成：

```
if(log.isDebugEnabled()) {
  log.debug("Entry number: " + i + " is " +
String.valueOf(entry[i]));
}
```

通过 log4j 参数化

```
Object entry = new SomeObject();
log.debug("The entry is {}.", entry);
log.debug("The new entry is {}.", entry);
```

A two argument variant is also available. For example, you can write:

```
log.debug("The new entry is {}. It replaces {}.", entry,
oldEntry);
```

If three or more arguments need to be passed, you can make use of the `Object...` variant of the printing methods. For example, you can write:

```
log.debug("Value {} was inserted between {} and {}.",
newVal, below, above);
```

This form incurs the hidden cost of construction of an Object[] (object array) which is usually very small. The one and two argument variants do not incur this hidden cost and exist solely for

this reason (efficiency). The slf4j-api would be smaller/cleaner with only the Object... variant.
Array type arguments, including multi-dimensional arrays, are also supported.
SLF4J uses its own message formatting implementation which differs from that of the Java platform. This is justified by the fact that SLF4J's implementation performs about 10 times faster but at the cost of being non-standard and less flexible.

Escaping the "{}" pair

The "{}" pair is called the *formatting anchor*. It serves to designate the location where arguments need to be substituted within the message pattern.
SLF4J only cares about the *formatting anchor*, that is the '{' character immediately followed by '}'. Thus, in case your message contains the '{' or the '}' character, you do not have to do anything special unless the '}' character immediately follows '}'. For example,

```
log.debug("Set {1,2} differs from {}", "3");
```

which will print as "Set {1,2} differs from 3".
You could have even written,

```
log.debug("Set {1,2} differs from {{}}", "3");
```

which would have printed as "Set {1,2} differs from {3}".
In the extremely rare case where the the "{}" pair occurs naturally within your text and you wish to disable the special meaning of the formatting anchor, then you need to escape the '{' character with '\', that is the backslash character. Only the '{' character should be escaped. There is no need to escape the '}' character. For example,

```
log.debug("Set \\{} differs from {}", "3");
```

will print as "Set {} differs from 3". Note that within Java code, the backslash character needs to be written as '\\'.
In the rare case where the "\{}" occurs naturally in the message, you can double escape the formatting anchor so that it retains its original meaning. For example,

```
log.debug("File name is C:\\\\{}.", "file.zip");
```

will print as "File name is C:\file.zip".

参照：

http://www.slf4j.org/faq.html#logging_performance