

# ImportNew

- [首页](#)
- [所有文章](#)
- [资讯](#)
- [Web](#)
- [架构](#)
- [基础技术](#)
- [书籍](#)
- [教程](#)
- [Java小组](#)
- [工具资源](#)

## Java日志终极指南

2015/07/20 | 分类：[基础技术](#) | [2 条评论](#) | 标签：[日志](#)

分享到：

<sup>50</sup> 本文由 [ImportNew](#) - [Wing](#) 翻译自 [loggly](#)。欢迎加入[翻译小](#)

[组](#)。转载请见文末要求。

### Java日志基础



Java使用了一种自定义的、可扩展的方法来输出日志。虽然Java通过java.util.logging包提供了一套基本的日志处理API，但你可以很轻松的使用一种或者多种其它日志解决方案。这些解决方案尽管使用不同的方法来创建日志数据，但它们的最终目标是一样的，即将日志从你的应用程序输出到目标地址。

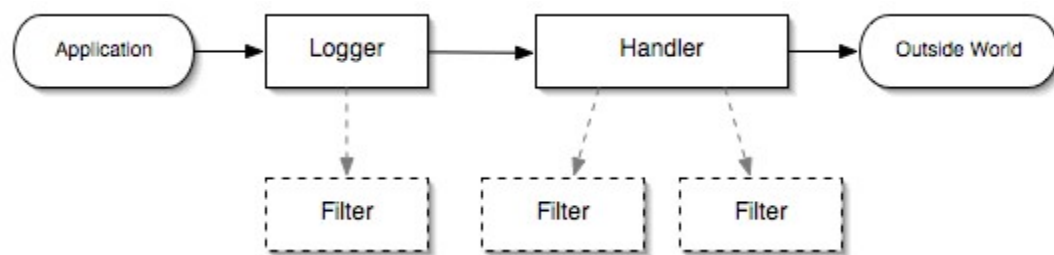
在这一节中，我们会探索Java日志背后的原理，并说明如何通过日志来让你成为一个更好的Java开发人员。

### Java日志组件

Java日志API由以下三个核心组件组成：

- [Loggers](#)：Logger负责捕捉事件并将其发送给合适的Appender。
- [Appenders](#)：也被称为Handlers，负责将日志事件记录到目标位置。在将日志事件输出之前，Appenders使用Layouts来对事件进行格式化处理。
- [Layouts](#)：也被称为Formatters，它负责对日志事件中的数据进行转换和格式化。Layouts决定了数据在一条日志记录中的最终形式。

当Logger记录一个事件时，它将事件转发给适当的Appender。然后Appender使用Layout来对日志记录进行格式化，并将其发送给控制台、文件或者其它目标位置。另外，Filters可以让你进一步指定一个Appender是否可以应用在一特定的日志记录上。在日志配置中，Filters并不是必需的，但可以让你更灵活地控制日志消息的流动。



## 日志框架

在Java中，输出日志需要使用一个或者多个日志框架，这些框架提供了必要的对象、方法和配置来传输消息。Java在java.util.logging包中提供了一个默认的框架。除此之外，还有很多其它第三方框架，包括[Log4j](#)、[Logback](#)以及[tinylog](#)。还有其它一些开发包，例如[SLF4J](#)和[Apache Commons Logging](#)，它们提供了一些抽象层，对你的代码和日志框架进行解耦，从而允许你在不同的日志框架中进行切换。

如何选择一个日志解决方案，这取决于你的日志需求的复杂度、和其它日志解决方案的兼容性、易用性以及个人喜好。Logback基于Log4j之前的版本开发（版本1），因此它们的功能集合都非常类似。然而，Log4j在最新版本（版本2）中引用了一些改进，例如支持多API，并提升了在用[Disruptor](#)库的性能。而tinylog，由于缺少了一些功能，运行特别快，非常适合小项目。

另外一个考虑因素是框架在基于Java的各种不同项目上的支持程度。例如Android程序只能使用[Log4j](#)、[Logback](#)或者第三方包来记录日志，Apache Tomcat可以使用Log4j来记录内部消息，但只能使用版本1的Log4j。

## 抽象层

诸如SLF4J这样的抽象层，会将你的应用程序从日志框架中解耦。应用程序可以在运行时选择绑定到一个特定的日志框架（例如java.util.logging、Log4j或者Logback），这通过在应用程序的类路径中添加对应的日志框架来实现。如果在类路径中配置的日志框架不可用，抽象层就会立刻取消调用日志的相应逻辑。抽象层可以让我们更加容易地改变项目现有的日志框架，或者集成那些使用了不同日志框架的项目。

## 配置

尽管所有的Java日志框架都可以通过代码进行配置，但是大部分配置还是通过外部配置文件完成的。这些文件决定了日志消息在何时通过什么方式进行处理，日志框架可以在运行时加载这些文件。在这一节中提供的大部分配置示例都使用了配置文件。

## java.util.logging

默认的Java日志框架将其配置存储到一个名为 `logging.properties` 的文件中。在这个文件中，每行是一个配置项，配置项使用点标记（dot notation）的形式。Java在其安装目录的lib文件夹下面安装了一个全局配置文件，但在启动一个Java程序时，你可以通过指定 `java.util.logging.config.file` 属性的方式来使用一个单独的日志配置文件，同样也可以在个人项目中创建和存储 `logging.properties` 文件。

下面的示例描述了如何在全局的 `logging.properties` 文件中定义一个Appender：

```
1 # default file output is in user's home directory.
2 java.util.logging.FileHandler.pattern = %h/java%u.log
3 java.util.logging.FileHandler.limit = 50000
4 java.util.logging.FileHandler.count = 1
5 java.util.logging.FileHandler.formatter = java.util.logging.XmlFormatter
```

## Log4j

Log4j[版本1](#)使用的语法和 `java.util.logging` 的语法很类似。使用了Log4j的程序会在项目目录中寻找一个名为 `log4j.properties` 的文件。默认情况下，Log4j配置会将所有日志消息输出到控制台上。Log4j同样也支持XML格式的配置文件，对应的配置信息会存储到 `log4j.xml` 文件中。

Log4j[版本2](#)支持XML、JSON和YAML格式的配置，这些配置会分别存储到 `log4j2.xml`、`log4j2.json` 和 `log4j2.yaml` 文件中。和版本1类似，版本2也会在工程目录中寻找这些文件。你可以在每个版本的文档中找到相应的配置文件示例。

## Logback

对于Logback来说，大部分配置都是在 [logback.xml](#) 文件中完成的，这个文件使用了和Log4j类似的XML语法。Logback同时也支持通过Groovy语言的方式进行配置，配置信息会存储到 [logback.groovy](#) 文件中。你可以通过每种类型配置文件的链接找到对应的配置文件示例。

## Loggers

Loggers是用来触发日志事件的对象，在我们的Java应用程序中被创建和调用，然后Loggers才会将事件传递给Appender。一个类中可以包含针对不同事件的多个独立的Loggers，你也可以在一个Loggers里面内嵌一个Loggers，从而创建一种Loggers[层次结构](#)。

## 创建新Logger

在不同的日志框架下面创建新Logger过程大同小异，尽管调用的具体方法名称可能不同。在使用 `java.util.logging` 时，你可以通过 `Logger.getLogger().getLogger()` 方法创建新Logger，这个方法接收一个string参数，用于指定Logger的名字。如果指定名字的Logger已经存在，那么只需要返回已经存在的Logger；否则，程序会创建一个新Logger。通常情况下，一种好的做法是，我们在当前类下使用 `class.getName()` 作为新Logger的名字。

```
1 Logger logger = Logger.getLogger(MyClass.class.getName());
```

## 记录日志事件

Logger提供了几种方法来触发日志事件。然而，在你记录一个事件之前，你还需要设置级别。日志级别用来确定日志的严重程度，它可以用来过滤日志事件或者将其发送给不同的Appender（想了解更多，请参考“[日志级别](#)”一节），Logger.log() 方法除了日志消息以外，还需要一个日志级别作为参数：

```
1 | logger.log(Level.WARNING, "This is a warning!");
```

大部分日志框架都针对输出特定级别日志提供了快捷方式。例如，下面语句的作用和上面语句的作用是一样的：


```
1 | logger.warning("This is a warning!");
```

你还可以阻止Logger输出低于指定日志级别的消息。在下面的示例中，Logger只能输出高于WARNING级别的日志消息，并丢弃日志级别低于WARNING的消息：

```
1 | logger.setLevel(Level.WARNING);
```

我们还有另外一些方法可以用来记录额外的信息。logp()（精确日志）可以让你指定每条日志记录的源类（source class）和方法，而 logrb()（使用资源绑定的日志）可以让你指定用于提取日志消息的资源。entering() 和 exiting() 方法可以让你记录方法调用信息，从而[追踪](#)程序的执行过程。

## Appenders

Appenders将日志消息转发给期望的输出。它负责接收日志事件，使用Layout格式化事件，然后将其发送给对应的目标。对于一个日志事件，我们可以使用多个Appenders来将事件发送到不同的目标位置。例如，我们可以在控制台上显示一个简单日志事件的同时，将其通过邮件的方式发送给指定的接收者。

请注意，在java.util.logging中，Appenders被称作Handlers。

### 增加Appender

大部分日志框架的Appender都会执行类似的功能，但在实现方面大相径庭。如果使用java.util.logging，你可以使用 Logger.addHandler() 方法将Appender添加到Logger中。例如，下面的代码添加了一个新的ConsoleHandler，它会将日志输出到控制台：

```
1 | logger.addHandler(new ConsoleHandler());
```

一种更常用的添加Appender的方式是使用配置文件。如果使用java.util.logging，Appenders会定义一个以逗号隔开的列表，下面的示例将日志事件输出到控制台和文件：

```
1 | handlers=java.util.logging.ConsoleHandler, java.util.logging.FileHandler
```

如果使用基于XML的配置文件，Appenders会被添加到<Appenders>元素下面，如果使用Log4j，我们可以很容易地添加一个新ConsoleAppender来将日志消息发送到System.out：

```
1 | <Console name="console" target="SYSTEM_OUT">
```

## Appenders类型

这一节描述了一些更通用的Appenders，以及它们在各种日志框架中是如何实现的。

### ConsoleAppender

ConsoleAppender是最常用的Appenders之一，它只是将日志消息显示到控制台上。许多日志框架都将其作为默认的Appender，并且在基本的配置中进行预配置。例如，在Log4j中ConsoleAppender的配置参数如下所示。

参数	描述
filter	用于决定是否需要使用该Appender来处理日志事件
layout	用于决定如何对日志记录进行格式化，默认情况下使用“%m%n”，它会在每一行显示一条日志记录
follow	用于决定Appender是否需要了解输出（system.out或者system.err）的变化，默认情况是不需要跟踪这种变化
name	用于设置Appender的名字
ignoreExceptions	用于决定是否需要在日志事件处理过程中出现的异常
target	用于指定输出目标位置，默认情况下使用SYSTEM_OUT，但也可以修改成SYSTEM_ERR

一个完整的Log4j2的配置文件如下所示：



```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 |   <Configuration status="warn" name="MyApp">
3 |     <Appenders>
4 |       <Console name="MyAppender" target="SYSTEM_OUT">
5 |         <PatternLayout pattern="%m%n"/>
6 |       </Console>
7 |     </Appenders>
8 |     <Loggers>
9 |       <Root level="error">
10 |        <AppenderRef ref="MyAppender"/>
11 |      </Root>
12 |    </Loggers>
13 |  </Configuration>
```

这个配置文件创建了一个名为MyAppender的ConsoleAppender，它使用PatternLayout来对日志事件进行格式化，然后再将其输出到System.out。<Loggers>元素对定义在程序代码中的Loggers进行了配置。在这里，我们只配置了一个LoggerConfig，即名为Root的Logger，它会接收哪些日志级别在ERROR以上的日志消息。如果我们使用logger.error()来记录一个消息，那么它就会出现在控制台上，就像这样：

```
1 | An unexpected error occurred.
```

你也可以使用Logback实现完全一样的效果：

```
1 <configuration>
2   <appender name="MyAppender" class="ch.qos.logback.core.ConsoleAppender">
3     <encoder>
4       <pattern>%m%n</pattern>
5     </encoder>
6   </appender>
7   <root level="error">
8     <appender-ref ref="MyAppender" />
9   </root>
10 </configuration>
```

## FileAppenders

FileAppenders将日志记录写入到文件中，它负责打开、关闭文件，向文件中追加日志记录，并对文件进行加锁，以免数据被破坏或者覆盖。

在Log4j中，如果想创建一个FileAppender，需要指定目标文件的名字，写入方式是追加还是覆盖，以及是否需要在写入日志时对文件进行加锁：

```
1 ...
2 <Appenders>
3   <File name="MyFileAppender" fileName="myLog.log" append="true" locking="true">
4     <PatternLayout pattern="%m%n"/>
5   </File>
6 </Appenders>
7 ...
```

这样我们创建了一个名为MyFileAppender的FileAppender，并且在向文件中追加日志时会对文件进行加锁操作。

如果使用Logback，你可以同时启用prudent模式来保证文件的完整性。虽然Prudent模式增加了写入文件所花费的时间，但它可以保证在多个FileAppender甚至多个Java程序向同一个文件写入日志时，文件的完整性。



```
1 ...
2 <appender name="FileAppender" class="ch.qos.logback.core.FileAppender">
3   <file>myLog.log</file>
4   <append>true</append>
5   <prudent>true</prudent>
6   <encoder>
7     <pattern>%m%n</pattern>
8   </encoder>
9 </appender>
10 ...
```

## SyslogAppender

SyslogAppenders将日志记录发送给本地或者远程系统的日志服务。[syslog](#)是一个接收日志事件服务，这些日志事件来自操作系统、进程、其它服务或者其它设备。事件的范围可以从诊断信息到用户登录硬件失败等。syslog的事件按照设备进行分类，它指定了正在记录的事件的类型。例如，auth facility表明这个事件是和安全以及认证有关。

Log4j和Logback都内置支持SyslogAppenders。在Log4j中，我们创建SyslogAppender时，需要指定syslog服务监听的主机号、端口号以及协议。下面的示例演示了如何设定装置：

```
1 ...
2 <Appenders>
```



```

3   <Syslog name="SyslogAppender" host="localhost" port="514" protocol="UDP" facility="Auth" />
4 </Appenders>
5 ...

```

在Logback中，我们可以实现同样的效果：

```

1 ...
2 <appender name="SyslogAppender" class="ch.qos.logback.classic.net.SyslogAppender">
3   <syslogHost>localhost</syslogHost>
4   <port>514</port>
5   <facility>Auth</facility>
6 </appender>
7 ...

```

## 其它Appender

我们已经介绍了一些经常用到的Appenders，还有很多其它Appender。它们添加了新功能或者在其它的一些Appender基础上实现了新功能。例如，Log4j中的RollingFileAppender扩展了FileAppender，它可以在满足特定条件时自动创建新的日志文件；SMTPAppender会将日志内容以邮件的形式发送出去；FailoverAppender会在处理日志的过程中，如果一个或者多个Appender失败，自动切换到其他Appender上。

如果想了解更多关于其他Appender的信息，可以查看[Log4j Appender参考](#)以及[Logback Appender参考](#)。

## Layouts

Layouts将日志记录的内容从一种数据形式转换成另外一种。日志框架为纯文本、HTML、syslog、XML、JSON、序列化以及其它日志提供了Layouts。

请注意：在java.util.logging中Layouts也被称为Formatters。

例如，java.util.logging提供了两种Layouts：SimpleFormatter和XMLFormatter。默认情况下，ConsoleHandlers使用SimpleFormatter，它输出的纯文本日志记录就像这样：

```

1 Mar 31, 2015 10:47:51 AM MyClass main
2 SEVERE: An exception occurred.

```

而默认情况下，FileHandlers使用XMLFormatter，它的输出就像这样：

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE log SYSTEM "logger.dtd">
3 <log>
4   <record>
5     <date>2015-03-31T10:47:51</date>
6     <millis>1427903275893</millis>
7     <sequence>0</sequence>
8     <logger>MyClass</logger>
9     <level>SEVERE</level>
10    <class>MyClass</class>
11    <method>main</method>
12    <thread>1</thread>
13    <message>An exception occurred.</message>
14  </record>
15 </log>

```

## 配置Layout

我们通常使用配置文件对Layouts进行配置。从Java 7开始，我们也可以使用[system property](#)来配置SimpleFormatter。

例如，在Log4j和Logback中最常用的Layouts是PatternLayout。它可以让你决定日志事件中的哪些部分需要输出，这是通过转换模式（[Conversion Pattern](#)）完成的，转换模式在每一条日志事件的数据中扮演了“占位符”的角色。例如，Log4j默认的PatternLayout使用了如下转换模式：

```
1 | <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
```

%d{HH:mm:ss.SSS} 将日期转换成时、分、秒和毫秒的形式，%level显示日志事件的严重程度，%C显示生成日志事件的类的名字，%t显示Logger的当前线程，%m显示时间的消息，最后，%n为下一个日志事件进行了换行。

## 改变Layouts

如果在java.util.logging中使用一个不同的Layout，需要将Appender的formatter属性设置成你想要的Layout。在代码中，你可以创建一个新的Handler，调用setFormatter方法，然后通过logger.addHandler()方法将Handler放到Logger上面。下面的示例创建了一个ConsoleAppender，它使用XMLFormatter来对日志进行格式化，而不是使用默认的SimpleFormatter：

```
1 | Handler ch = new ConsoleHandler();
2 | ch.setFormatter(new XMLFormatter());
3 | logger.addHandler(ch);
```

这样Logger会将下面的信息输出到控制台上：



```
1 | <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 | <!DOCTYPE log SYSTEM "logger.dtd">
3 | <log>
4 | <record>
5 |   <date>2015-03-31T10:47:51</date>
6 |   <millis>1427813271000</millis>
7 |   <sequence>0</sequence>
8 |   <logger>MyClass</logger>
9 |   <level>SEVERE</level>
10 |  <class>MyClass</class>
11 |  <method>main</method>
12 |  <thread>1</thread>
13 |  <message>An exception occurred.</message>
14 | </record>
```

如果想了解更多信息，你可以查看[Log4j Layouts参考](#)以及[Logback Layouts参考](#)。

## 使用自定义Layouts

自定义Layouts可以让你指定Appender应该如何输出日志记录。[从Java SE 7开始](#)，尽管你可以调整SimpleLogger的输出，但有一个限制，即只能够调整简单的纯文本消息。对于更高级的格式，例如HTML或者JSON，你需要一个自定义Layout或者一个单独的框架。

如果想了解更多使用java.util.logging创建自定义Layouts的信息，你可以查看Jakob Jenkov的Java



日志指南中的[Java Logging: Formatters](#)章节。

## 日志级别

日志级别提供了一种方式，我们可以用它来根据严重程度对日志进行分类和识别。java.util.logging按照严重程度从重到轻，提供了以下级别：

- SEVERE ( 最高级别 )
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST ( 最低级别 )

另外，还有两个日志级别：ALL和OFF。ALL会让Logger输出所有消息，而OFF则会关闭日志功能。

## 设置日志级别

在设定日志级别后，Logger会自动忽略那些低于设定级别的日志消息。例如，下面的语句会让Logger忽略那些低于WARNING级别的日志消息：

```
1 | logger.setLevel(Level.WARNING);
```

然后，Logger会记录任何WARNING或者更高级别的日志消息。我们也可以在配置文件中设置Logger的日志级别：

```
1 | ...  
2 | <Loggers>  
3 |   <Logger name="MyLogger" level="warning">  
4 |     ...
```



## 转换模式

Log4j和Logback中的PatternLayout类都支持转换模式，它决定了我们如何从每一条日志事件中提取信息以及如何对信息进行格式化。下面显示了这些模式的一个子集，对于Log4j和Logback来说，虽然这些特定的字段都是一样的，但是并不是所有的字段都会使用相同的模式。想要了解更多信息，可以查看[Log4j](#)和[Logback](#)的PatternLayout文档。

字段名称	Log4j/Logback 模式
消息	%m
级别/严重程度	%p
异常	%ex
线程	%t
Logger	%c

## 方法

%M

例如，下面的PatternLayout会在中括号内x显示日志级别，后面是线程名字和日志事件的消息：

```
1 | [%p] %t: %m
```

下面是使用了上述转换模式后的日志输出示例：

```
1 | [INFO] main: initializing worker threads
2 | [DEBUG] worker: listening on port 12222[INFO] worker: received request from 192.168.1.200[ERROR] work
```

## 记录栈跟踪信息

如果你在Java程序中使用过[异常](#)，那么很有可能已经看到过栈跟踪信息。它提供了一个程序中方法调用的快照，让你准确定位程序执行的位置。例如，下面的栈跟踪信息是程序试图打开一个不存在的文件后生成的：

```
1 | [ERROR] main: Unable to open file! java.io.FileNotFoundException: foo.file (No such file or directory
2 |   at java.io.FileInputStream.open(Native Method) ~[?:1.7.0_79]
3 |   at java.io.FileInputStream.<init>(FileInputStream.java:146) ~[?:1.7.0_79]
4 |   at java.io.FileInputStream.<init>(FileInputStream.java:101) ~[?:1.7.0_79]
5 |   at java.io.FileReader.<init>(FileReader.java:58) ~[?:1.7.0_79]
6 |   at FooClass.main(FooClass.java:47)
```

这个示例使用了一个名为FooClass的类，它包含一个main方法。在程序第47行，FileReader独享试图打开一个名为foo.file的文件，由于在程序目录下没有名字是foo.file的文件，因此Java虚拟机抛出了一个FileNotFoundException。因为这个方法调用被放到了try-catch语块中，所以我们能够捕获这个异常并记录它，或者至少可以阻止程序崩溃。

## 使用PatternLayout记录栈跟踪信息



在写本篇文章时最新版本的Log4j和Logback中，如果在Layout中没有和可抛异常相关的信息，那么都会自动将%xEx（这种栈跟踪信息包含了每次方法调用的包信息）添加到PatternLayout中。如果对于普通的日志信息的模式如下：

```
1 | [%p] %t: %m
```

它会变为：

```
1 | [%p] %t: %m%xEx
```

这样不仅仅错误信息会被记录下来，完整的栈跟踪信息也会被记录：

```
1 | [ERROR] main: Unable to open file! java.io.FileNotFoundException: foo.file (No such file or directory
2 |   at java.io.FileInputStream.open(Native Method) ~[?:1.7.0_79]
3 |   at java.io.FileInputStream.<init>(FileInputStream.java:146) ~[?:1.7.0_79]
4 |   at java.io.FileInputStream.<init>(FileInputStream.java:101) ~[?:1.7.0_79]
5 |   at java.io.FileReader.<init>(FileReader.java:58) ~[?:1.7.0_79]
6 |   at FooClass.main(FooClass.java:47)
```

%xEx中的包查询是一个代价昂贵的操作，如果你频繁的记录异常信息，那么可能会碰到性能问题，例如：

```
1 | // ...
2 | } catch (FileNotFoundException ex) {
3 |     logger.error("Unable to open file!", ex);
4 | }
```

一种解决方法是在模式中显式的包含%ex，这样就只会请求异常的栈跟踪信息：

```
1 | [%p] %t: %m%ex
```

另外一种方法是通过追加%xEx(none)的方法排除（在Log4j）中所有的异常信息：

```
1 | [%p] %t: %m%xEx{none}
```

或者在Logback中使用%nopex：

```
1 | [%p] %t: %m%nopex
```

## 使用结构化布局输出栈跟踪信息

如你在“[解析多行栈跟踪信息](#)”一节中所见，对于站跟踪信息来说，使用结构化布局来记录是最合适的方式，例如JSON和XML。这些布局会自动将栈跟踪信息按照核心组件进行分解，这样我们可以很容易将其导出到其他程序或者日志服务中。对于上述站跟踪信息，如果使用JSON格式，部分信息显示如下：

```
1 | ...
2 | "loggerName" : "FooClass",
3 | "message" : "Foo, oh no! ",
4 | "thrown" : {
5 |     "commonElementCount" : 0,
6 |     "localizedMessage" : "foo.file (No such file or directory)",
7 |     "message" : "foo.file (No such file or directory)",
8 |     "name" : "java.io.FileNotFoundException",
9 |     "extendedStackTrace" : [ {
10 |         "class" : "java.io.FileInputStream",
11 |         "method" : "open",
12 |         "file" : "FileInputStream.java",
13 |         ...
```



## 记录未捕获异常

通常情况下，我们通过捕获的方式来处理异常。如果一个异常没有被捕获，那么它可能会导致程序终止。如果能够留存任何日志，那么这是一个可以帮助我们调试为什么会发生异常的好办法，这样你就可以找到发生异常的根本原因并解决它。下面来说明我们如何建立一个默认的异常处理器来记录这些错误。

[Thread](#)类中有两个方法，我们可以用它来为未捕获的异常指定一个ExceptionHandler：

[setDefaultUncaughtExceptionHandler](#)可以让你在任何线程上处理任何异常。

setUncaughtExceptionHandler可以让你针对一个指定的线程设定一个不同的处理方法。而ThreadGroup则允许你设定一个处理方法。大部分人会使用默认的异常处理方法。

下面是一个示例，它设定了一个默认的异常处理方法，来创建一个日志事件。它要求你传入一个UncaughtExceptionHandler：

```
1 | import java.util.logging.*;
```

下面是一个未处理异常的输出示例：

```
1 | May 29, 2015 2:21:15 PM ExceptionDemo$1 uncaughtException
2 | SEVERE: Thread[Thread-1,5,main] ExceptionDemo threw an exception:
3 | java.lang.RuntimeException
4 |     at ExceptionDemo$1adminThread.run(ExceptionDemo.java:15)
5 |     at java.lang.Thread.run(Thread.java:745)
```

## JSON

JSON ( JavaScript Object Notation ) 是一种用来存储结构化数据的格式，它将数据存储成键值对的集合，类似于HashMap或者Hashtable。JSON具有的可移植性和通用性，大部分现代语言都内置支持它或者通过已经准备好的第三方类库来支持它。

JSON支持许多基本数据类型，包括字符串、数字、布尔、数组和null。例如，你可以使用下面的JSON格式来表示一个电脑：



```
1 | {
2 |   "manufacturer": "Dell",
3 |   "model": "Inspiron",
4 |   "hardware": {
5 |     "cpu": "Intel Core i7",
6 |     "ram": 16384,
7 |     "cdrom": null
8 |   },
9 |   "peripherals": [
10 |    {
11 |      "type": "monitor",
12 |      "manufacturer": "Acer",
13 |      "model": "S231HL"
14 |    }
15 |  ]
16 | }
```

JSON的可移植性使得它非常适合存储日志记录，使用JSON后，Java日志可以被任何数目的JSON解释器所读取。因为数据已经是结构化的，所以解析JSON日志要远比解析纯文本日志容易。

## Java中的JSON

对于Java来说，有大量的JSON实现，其中一个JSON.simple。JSON.simple是轻量级的、易于使用，并且全部符合JSON标准。

如果想将上面的computer对象转换成可用的Java对象，我们可以从文件中读取JSON内容，将其传递给JSON.simple，然后返回一个Object，接着我们可以将Object转换成JSONObject：

```
1 | Object computer = JSONValue.parse(new FileReader("computer.json"));
2 | JSONObject computerJSON = (JSONObject)computer;
```

另外，为了取得键值对的信息，你可以使用任何日志框架来记录一个JSONObject，JSONObject对象包含一个toString()方法，它可以将JSON转换成文本：

```
1 | 2015-05-06 14:54:32,878 INFO  JSONTest main {"peripherals":[{"model":"S231HL","manufacturer":"Acer"},"
```

虽然这样做可以很容易的打印JSONObject，但如果你使用结构化的Layouts，例如JSONLayout或者XMLLayout，可能会导致意想不到的结果：

```
1 | ...
2 | "message" : "{\"peripherals\": [{\"model\": \"S231HL\", \"manufacturer\": \"Acer\", \"type\": \"monitor\"}], \"model\": \"Insp
3 | ...
```

Log4j中的JSONLayout并没有内置支持内嵌JSON对象，但你可以通过创建自定义Layout的方式来添加一个JSONObject字段，这个Layout会继承或者替换JSONLayout。然而，如果你使用一个日志管理系统，需要记住许多日志管理系统会针对某些字段使用预定义的数据类型。如果你创建一个Layout并将JSONObject存储到message字段中，那么它可能会和日志系统中使用的String数据类型相冲突。一种解决办法是将JSON数据存储在到一个字段中，然后将字符串类型的日志消息存储在另外一个字段中。

## 其它JSON库

除了JSON.simple，Java中还有很多其它JSON库。[JSON-java](#)是由JSON创建者开发的一个参考实现，它包含了额外的一些功能，可以转换其它数据类型，包括web元素。但是目前JSON-java已经没有人来维护和提供支持了。



如果想将JSON对象转换成Java对象或者逆向转换，Google提供了一个Gson库。使用Gson时，可以很简单使用 toJson() 和 fromJson() 方法来解析JSON，这两个方法分别用来将Java对象转换成JSON字符串以及将JSON字符串转换成Java对象。Gson甚至可以应用在内存对象中，允许你映射到那些没有源代码的对象上。

## Jackson

[Jackson](#)是一个强大的、流行的、功能丰富的库，它可以在Java中管理JSON对象。有一些框架甚至使用Jackson作为它们的JSONLayouts。尽管它很大并且复杂，但Jackson对于初学者和高级用户来说，是很容易使用的。

Logback通过logback-jackson和logback-json-classic库继承了Jackson，这两个库也是[logback-contrib](#)项目的一部分。在集成了Jackson后，你可以将日志以JSON的格式导出到任何Appender中。

[Logback Wiki](#)详细解释了如何将JSON添加到logback中，在Wiki页面中的示例使用了LogglyAppender，这里的配置也可以应用到其他Appender上。下面的示例说明了如何将JSON格

式化的日志记录写入到名为myLog.json的文件中：

```
1  ...
2  <appender name="file" class="ch.qos.logback.core.FileAppender">
3    <file>myLog.json</file>
4    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
5      <layout class="ch.qos.logback.contrib.json.classic.JsonLayout">
6        <jacksonJsonFormatter class="ch.qos.logback.contrib.jackson.JacksonJsonFormatter"/>
7      </layout>
8    </encoder>
9  </appender>
10 ...
```

你也可以通过[FasterXML Wiki](#)找到更多关于Jackson的深度介绍。

## 了解更多JSON相关信息

你可以通过[JSON主页](#)学习更多JSON相关信息，或者通过[CodeAcademy](#)来通过学习一个交互式的快速上手教程（请注意这个课程是基于JavaScript的，而不是Java）。有一些在线工具例如[JSONLint](#)和[JSON在线编辑器](#)可以帮助你解析、验证以及格式化JSON代码。

## NDC、MDC以及ThreadContext

当处理多线程应用程序，特别是web服务时，跟踪事件可能会变得困难。当针对多个同时存在的多个用户生成日志记录时，你如何区分哪个行为和哪个日志事件有关呢？如何两个用户没有成功打开一个相同的文件，或者在同一时间没有成功登陆，那么怎么处理日志记录？你可能需要一种方式来将日志记录和程序中的唯一标识符关联起来，这些标识符可能是用户ID，会话ID或者设备ID。而这就是NDC、MDC以及ThreadContext的用武之地。

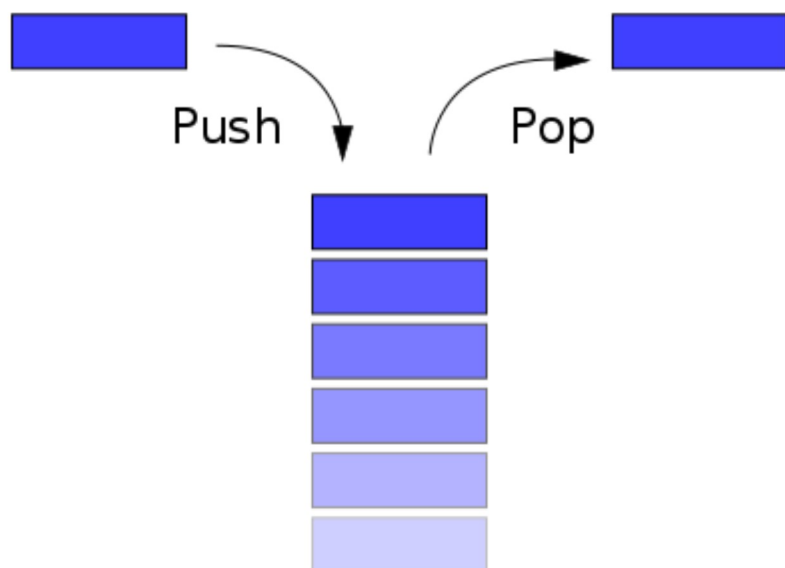
NDC、MDC和ThreadContext通过向单独的日志记录中添加独一无二的的数据戳，来创建日志足迹（log trails）。这些数据戳也被称为鱼标记（fish tagging），我们可以通过一个或者多个独一无二的值来区分日志。这些数据戳在每个线程级别上进行管理，并且一直持续到线程结束，或者直到数据戳被删掉。例如，如果你的Web应用程序为每个用户生成一个新的线程，那么你可以使用这个用户的ID来标记日志记录。当你想在一个复杂的系统中跟踪特定的请求、事务或者用户，这是一种非常有用的方法。

### 嵌套诊断上下文（NDC）

NDC或者嵌套诊断上下文（Nested Diagnostic Context）是基于栈的思想，信息可以被放到栈上或者从栈中移除。而栈中的值可以被Logger访问，并且Logger无需显示想日志方法中传入任何值。

下面的代码示例使用NDC和Log4j来将用户姓名和一条日志记录关联起来。NDC是一个静态类，因此我们可以直接访问它的方法，而无需实例化一个NDC对象。在这个示例中，NDC.push(username) 和 NDC.push(sessionID) 方法在栈中存储了当前的用户名（admin）和会话ID（1234），而NDC.pop()方法将一些项从栈中移除，NDC.remove()方法让Java回收内存，以免造成内存溢出。





```

1  import java.io.FileReader;
2  import org.apache.Log4j.Logger;
3  import org.apache.Log4j.NDC;
4  ...
5  String username = "admin";
6  String sessionID = "1234";
7  NDC.push(username);
8  NDC.push(sessionID);
9  try {
10     // tmpFile doesn't exist, causing an exception.
11     FileReader fr = new FileReader("tmpFile");
12 }
13 catch (Exception ex) {
14     logger.error("Unable to open file.");
15 }
16 finally {
17     NDC.pop();
18     NDC.pop();
19     NDC.remove();
20 }

```



Log4j的PatternLayout类通过%x转换字符从NDC中提取值。如果一个日志事件被触发，那么完整的NDC栈就被传到Log4j：

```

1 | <PatternLayout pattern="%x %-5p - %m%n" />

```

运行程序后，我们可以得出下面的输出：

```

1 | "admin 1234 ERROR - Unable to open file."

```

## 映射诊断上下文 (MDC)

MDC或者映射诊断上下文和NDC很相似，不同之处在于MDC将值存储在键值对中，而不是栈中。这样你可以很容易的在Layout中引用一个单独的键。MDC.put(key,value) 方法将一个新的键值对添加到上下文中，而 MDC.remove(key) 方法会移除指定的键值对。

如果想在日志中同样显示用户名和会话ID，我们需要使用 MDC.put() 方法将这两个变量存储成键值对：

```

1  import java.io.FileReader;
2  import org.apache.Log4j.Logger;
3  import org.apache.Log4j.MDC;
4  ...
5  MDC.put("username", "admin");
6  MDC.put("sessionID", "1234");
7  try {
8      // tmpFile doesn't exist, causing an exception.
9      FileReader fr = new FileReader("tmpFile");
10 }
11 catch (Exception ex) {
12     logger.error("Unable to open file!");
13 }
14 finally {
15     MDC.clear();
16 }

```

这里再一次强调，在不需要使用Context后，我们需要使用 `MDC.clear()` 方法将所有的键值对从MDC中移除，这样会降低内存的使用量，并阻止MDC在后面试图调用那些已经过期的数据。

在日志框架中访问MDC的值时，也稍微有些区别。对于存储在上下文中的任意键，我们可以使用 `%X(键)` 的方式来访问对应的值。这样，我们可以使用 `%X(username)` 和 `%X(sessionID)` 来获取对应的用户名和会话ID：


```

1  <PatternLayout pattern="%X{username} %X{sessionID} %-5p - %m%n" />
2  "admin 1234 ERROR - Unable to open file!"

```

如果我们没有指定任何键，那么MDC上下文就会被以 `{(key, value),(key, value)}` 的方式传递给Appender。

## Logback中的NDC和MDC

和Log4j不同，Logback内置没有实现NDC。但是`slf4j-ext`包提供了一个NDC实现，它使用MDC作为基础。在Logback内部，你可以使用 `MDC.put()`  `MDC.remove()` 和 `MDC.clear()` 方法来访问和管理MDC：

```

1  import org.slf4j.MDC;
2  ...
3  Logger logger = LoggerFactory.getLogger(MDCLogback.class);
4  ...
5  MDC.put("username", "admin");
6  MDC.put("sessionID", "1234");
7  try {
8      FileReader fr = new FileReader("tmpFile");
9  }
10 catch (Exception ex) {
11     logger.error("Unable to open file.");
12 }
13 finally {
14     MDC.clear();
15 }

```

在Logback中，你可以在Logback.xml中将如下模式应用到Appender上，它可以输出和上面Log4j相同的结果：

```

1  <Pattern>[%X{username}] %X{sessionID} %-5p - %m%n</Pattern>
2  "[admin] 1234 ERROR - Unable to open file."

```

针对MDC的访问并不仅仅限制在PatternLayout上，例如，当使用JSONFormatter时，MDC中的所

有值都会被导出：

```
1 {
2   "timestamp": "1431970324945",
3   "level": "ERROR",
4   "thread": "main",
5   "mdc": {
6     "username": "admin",
7     "sessionID": "1234"
8   },
9   "logger": "MyClass",
10  "message": "Unable to open file.",
11  "context": "default"
12 }
```

## ThreadContext

Version 2 of Log4j merged MDC and NDC into a single concept known as the Thread Context. The Thread Context is an evolution of MDC and NDC, presenting them respectively as the Thread Context Map and Thread Context Stack. The Thread Context is managed through the static ThreadContext class, which is implemented similar to Log4j 1's MDC and NDC classes.

Log4j版本2中将MDC和NDC合并到一个单独的组件中，这个组件被称为线程上下文。线程上下文是针对MDC和NDC的进化，它分别用线程上下文Map映射线程上下文栈来表示MDC和NDC。我们可以通过ThreadContext静态类来管理线程上下文，这个类在实现上类似于Log4j版本1中的MDC和NDC。

When using the Thread Context Stack, data is pushed to and popped from a stack just like with NDC:

当使用线程上下文栈时，我们可以向NDC那样向栈添加或者删除数据：

```
1 import org.apache.logging.Log4j.ThreadContext;
2 ...
3 ThreadContext.push(username);
4 ThreadContext.push(sessionID);
5 // Logging methods go here
6 ThreadContext.pop();
7 ...
```

当使用线程上下文映射时，我们可以像MDC那样将值和键结合在一起：

```
1 import org.apache.logging.Log4j.ThreadContext;
2 ...
3 ThreadContext.put("username", "admin");
4 ThreadContext.put("sessionID", "1234");
5 // Logging methods go here
6 ThreadContext.clearMap();
7 ...
```

ThreadContext类提供了一些方法，用于清除栈、清除MDC、清除存储在上下文中的所有值，对应的方法是ThreadContext.clearAll()、ThreadContext.clearMap()和ThreadContext.clearStack()。

和在MDC以及NDC中一样，我们可以使用Layouts在线程上下文中访问这些值。使用PatternLayout时，%x转换模式会从栈中获取值，%X和%X(键)会从图中获取值。

## ThreadContext过滤

一些框架允许你基于某些属性对日志进行过滤。例如，Log4j的[DynamicThresholdFilter](#) 会在键满足特定条件的情况下，自动调整日志级别。再比如，如果我们想要触发TRACE级别的日志消息，我们可以创建一个名为trace-logging-enabled的键，并向log4j配置文件中添加一个过滤器：

```
1 <Configuration name="MyApp">
2   <DynamicThresholdFilter key="trace-logging-enabled" onMatch="ACCEPT" onMismatch="NEUTRAL">
3     <KeyValuePair key="true" value="TRACE" />
4   </DynamicThresholdFilter>
5   ...
```

如果ThreadContext包含一个名为trace-logging-enabled的键，onMatch 和 onMismatch 会决定如何处理它。关于 onMatch 和 onMismatch，我们有三项选项：ACCEPT，它会处理过滤器的规则；DENY，它会忽略过滤器的规则；NEUTRAL，它会推迟到下一个过滤器。除了这些，我们还定义一个键值对，当值为true时，我们启用TRACE级别的日志。

现在，当trace-logging-enabled被设置成true时，即使根Logger设置的日志级别高于TRACE，Appender也会记录TRACE级别的消息。

你可能还想过滤一些特定的日志到特定的Appender中，Log4j中提供了ThreadContextMapFilter来实现这一点。如果我们想要限制某个特定的Appender，只记录针对某个用户的TRACE级别的消息，我们可以基于username键添加一个ThreadContextMapFilter：

```
1 <Console name="ConsoleAppender" target="SYSTEM_OUT">
2   <ThreadContextMapFilter onMatch="ACCEPT" onMismatch="DENY">
3     <KeyValuePair key="username" value="admin" />
4   </ThreadContextMapFilter>
5   ...
```

如果想了解更多信息，你可以查看[Log4j](#)和[Logback](#)文档中关于DynamicThresholdFilter部分。

## Markers

Markers允许你对单独的日志记录添加一些独一无二的信息。它可以用来对日志记录进行分组，触发一些行为，或者对日志记录进行过滤，并将过滤结果输出到指定的Appender中。你甚至可以将Markers和ThreadContext结合在一起使用，以提高搜索和过滤日志数据的能力。

例如，假设我们有一个可以连接到数据库的类，如果在打开数据库的时候发生了异常，我们需要把异常记录成fatal错误。我们可以创建一个名为DB\_ERROR的Marker，然后将其应用到日志事件中：

```
1 import org.apache.logging.Log4j.Marker;
2 import org.apache.logging.Log4j.MarkerManager;
3 ...
4 final static Marker DB_ERROR = MarkerManager.getMarker("DATABASE_ERROR");
5 ...
6 logger.fatal(DB_ERROR, "An exception occurred.");
```

为了在日志输出中显示Marker信息，我们需要在PatternLayout中添加%marker转换模式：

```
1 <PatternLayout pattern="%p %marker: %m%n" />
2 [FATAL] DATABASE_ERROR: An exception occurred.
```

或者对于JSON和XML格式的Layouts，会自动在输出中包含Marker信息：

```
1  ...
2  "thread" : "main",
3  "level" : "FATAL",
4  "loggerName" : "DBCClass",
5  "marker" : {
6    "name" : "DATABASE_ERROR"
7  },
8  "message" : "An exception occurred.",
9  ...
```

通过对Marker数据进行自动解析和排序，集中式的日志服务可以很容易对日志进行搜索处理。

## Markers过滤

Marker过滤器可以让你决定哪些Marker由哪些Logger来处理。marker字段会比较在日志事件里面的Marker名字，如果名字匹配，那么Logger会执行后续的行为。例如，在[Log4j](#)中，我们可以配置一个Appender来只显示哪些使用了DB\_ERROR Marker的消息，这可以通过log4j2.xml中的Appender添加如下信息来实现：

```
1 <MarkerFilter marker="DATABASE_ERROR" onMatch="ACCEPT" onMismatch="DENY" />
```

如果日志记录中某一条的Marker可以匹配这里的marker字段，那么onMatch会决定如何处理这条记录。如果不能够匹配，或者日志记录中没有Marker信息，那么onMismatch就会决定如何处理这条记录。对于onMatch和onMismatch来说，有3个可选项：ACCEPT，它允许记录事件；DENY，它会阻塞事件；NEUTRAL，它不会对事件进行任何处理。

在Logback中，我们需要更多一些设置。首先，想Appender中添加一个新的EvaluatorFilter，并如上所述指定onMatch和onMismatch行为。然后，添加一个OnMarkerEvaluator并将Marker的名字传递给它：



```
1 <filter class="ch.qos.Logback.core.filter.EvaluatorFilter">
2   <evaluator class="ch.qos.Logback.classic.boolex.OnMarkerEvaluator">
3     <marker>DATABASE_ERROR</marker>
4   </evaluator>
5   <onMatch>ACCEPT</onMatch>
6   <onMismatch>DENY</onMismatch>
7 </filter>
```

## 将Markers和NDC、MDC以及ThreadContext结合使用

Marker的功能和ThreadContext类似，它们都是向日志记录中添加独一无二的的数据，这些数据可以被Appender访问。如果把这两者结合使用，可以让你更容易的对日志数据进行索引和搜索。如果能够知道何时使用哪一种技术，会对我们有所帮助。

NDC、MDC和ThreadContext被用于将相关日志记录结合在一起。如果你的应用程序会处理多个同时存在的用户，ThreadContext可以让你将针对某个特定用户的一组日志记录组合在一起。因为ThreadContext针对每个线程都是不一样的，所以你可以使用同样的方法来对相关的日志记录进行自动分组。

另一方面，Marker通常用于标记或者高亮显示某些特殊事件。在上述示例中，我们使用DB\_ERROR

Marker来标明在方法中发生的SQL相关异常。我们可以使用DB\_ERROR Marker来将这些事件的处理过程和其他事件区分开来，例如我们可以使用SMTP Appender来将这些事件通过邮件发送给数据库管理员。

## 额外资源

### 指南和教程

- [Java Logging](#) ( Jakob Jenkov ) ——使用Java Logging API进行日志开发教程
- [Java Logging Overview](#) ( Oracle ) —— Oracle提供的在Java中进行日志开发的指南
- [Log4J Tutorial](#) ( Tutorials Point ) ——使用log4j 版本1进行日志开发的指南

### 日志抽象层

- [Apache Commons Logging](#) ( Apache ) ——针对Log4j、Avalon LogKit和java.util.logging的抽象层
- [SLF4J](#) ( QOS.ch ) ——一个流程的抽象层，应用在多个日志框架上，包括Log4j、Logback以及java.util.logging

### 日志框架

- [Java Logging API](#) ( Oracle ) —— Java默认的日志框架
- [Log4j](#) ( Apache ) ——开源日志框架
- [Logback](#) ( Logback Project ) ——开源项目，被设计成Log4j版本1的后续版本
- [tinylog](#) ( tinylog ) ——轻量级开源logger



原文链接：[loggly](#) 翻译：[ImportNew.com - Wing](#)

译文链接：<http://www.importnew.com/16331.html>

[ **转载请保留原文出处、译者和译文链接。** ]

### 关于作者：[Wing](#)

( 新浪微博：[@Wing011203](#) )

[查看Wing的更多文章 >>](#)





## 相关文章

- [异步打印日志的一点事](#)
- [Log4j1 升级 Log4j2 实战](#)
- [关于日志记录的一些感想](#)
- [从零开始玩转logback](#)
- [日志打印的5点建议](#)
- [Logstash实践: 分布式系统的日志监控](#)
- [简化SLF4J和通用日志工具的区别](#)
- [Java日志记录的5条规则](#)
- [在运行时开启GC日志](#)
- [Log4j 2 介绍](#)

## 发表评论



### Comment form

Name\*

姓名

邮箱\*

请填写邮箱

网站 (请以 http://开头)

请填写网站地址

评论内容\*

请填写评论内容



(\*) 表示必填项

[提交评论](#)

## 2 条评论

1. 呆子 说道：

[2017/07/19 下午 6:09](#)

学习了，简单的日志框架还有这么多功能。

 0  0

[回复](#)

2. [json formatter](#) 说道：

[2017/08/23 下午 5:35](#)

非常有用的。好文章！

👍 0 🗨️ 0

[回复](#)

« [企业级 Java 应用最重要的4个性能指标](#)  
[写Java也得了解CPU-CPU缓存](#) »

Search for:



- [本周热门文章](#)
- [本月热门](#)
- [热门标签](#)

0 [Java 面试题：百度前200页都在这里了](#)

1 [ArrayList 初始化 - Java 那...](#)

2 [Java 高并发综合](#)



3 [面试中单例模式有几种写法](#)

4 [你真的会阅读 Java 的异常信息吗](#)

5 [面试中单例模式有几种写法](#)

6 [图解 Java 多线程](#)

7 [Java 实现生产者 - 消费者模型](#)

8 [一道题看清动态规划的前世今生 \(...](#)









9 [就是让你懂 Spring 中 Mybatis...](#)

[android23days](#) [Android开发](#) [AOP](#) [ArrayList](#) [ConcurrentHashMap](#) [Eclipse](#) [GC](#) [Guava](#) [Hadoop](#) [HashMap](#) [HashSet](#) [HBase](#)  
[Hibernate](#) [IntelliJ](#) [io](#) [Java](#) [java8](#) [java 8](#) [Java9](#) [javaee](#) [Java NIO](#) [Java乱码](#) [Java编程入门](#) [JDBC](#) [JDK](#) [JMX](#) [JPA](#) [Jsoup](#) [JUnit](#) [JVM](#)  
[Lambda](#) [log4j](#) [MapReduce](#) [maven](#) [Mybatis](#) [Netty](#) [nio](#) [oracle](#) [ORM](#) [RabbitMQ](#) [redis](#) [RESTful](#) [Scala](#) [Servlet](#) [Socket](#) [solr](#) [Spring](#) [Spring4](#)  
[springboot](#) [spring boot](#) [Spring MVC](#) [SpringMVC](#) [Spring Security](#) [String](#) [synchronized](#) [ThreadLocal](#) [Tomcat](#) [volatile](#) [Web Service](#)

[Zookeeper](#) [书籍](#) [事务](#) [内存管理](#) [分布式](#) [动态代理](#) [单例](#) [参数太多怎么办](#) [反射](#) [垃圾回收](#) [基础技术](#) [多线程](#) [字符串](#) [字节码](#) [并发](#) [并发编程](#) [序列化](#) [异常](#) [异常处理](#) [性能](#) [性能优化](#) [性能调优](#) [教程](#) [数据结构](#) [日志](#) [架构](#) [泛型](#) [注解](#) [测试](#) [游戏](#) [算法](#) [线程](#) [线程池](#) [缓存](#) [自动化测试](#) [虚拟机](#) [设计模式](#) [资讯](#) [集合](#) [面试](#) [面试题](#)



## 最新评论

-   
Re: [攻破JAVA NIO技术壁垒](#)  
文章高质量好评 夜空中最亮的星
-   
Re: [轻量级分布式 RPC 框架](#)  
可以使用，对分布式系统理解和zookeeper使用很有帮助。十分感谢 领悟
-   
Re: [8张图理解Java](#)  
总结的不错 张奔奔
-   
Re: [Java 面试题：百度前200页都在...](#)   
不错，细节和深度都值得注意，感谢！ 夏天
-   
Re: [SpringBoot\(三\)：Spring boo...](#)  
你是对的 Jack
-   
Re: [Java 面试题：百度前200页都在...](#)  
人说的是百度来的java面试题前200页，又不是BAT的面试题。 苏铭枫
-   
Re: [Java 面试题：百度前200页都在...](#)  
这个收集的很好。但是没有必要打着百度的旗号。BAT的面试一般是看某个方面的深入程度。这个就是将所有的... 墨非
-   
Re: [Java 面试题：百度前200页都在...](#)  
有没有答案 taio

## 关于ImportNew

ImportNew 专注于 Java 技术分享。于2012年11月11日 11:11正式上线。是的，这是一个很特别的时刻：)

ImportNew 由两个 Java 关键字 import 和 new 组成，意指：Java 开发者学习新知识的网站。import 可认为是学习和吸收，new 则可认为是新知识、新技术圈子和新朋友.....



## 联系我们

Email : [ImportNew.com@gmail.com](mailto:ImportNew.com@gmail.com)

新浪微博 : [@ImportNew](https://weibo.com/ImportNew)

推荐微信号



ImportNew



安卓应用频道



Linux爱好者

反馈建议 : [ImportNew.com@gmail.com](mailto:ImportNew.com@gmail.com)

广告与商务合作QQ : 2302462408



## 推荐关注

[小组](#) – 好的话题、有启发的回复、值得信赖的圈子

[头条](#) – 写了文章？看干货？去头条！

[相亲](#) – 为IT单身男女服务的征婚传播平台

[资源](#) – 优秀的工具资源导航

[翻译](#) – 活跃 & 专业的翻译小组

[博客](#) – 国内外的精选博客文章

[设计](#) – UI,网页，交互和用户体验

[前端](#) – JavaScript, HTML5, CSS

[安卓](#) – 专注Android技术分享

[iOS](#) – 专注iOS技术分享

[Java](#) – 专注Java技术分享

[Python](#) – 专注Python技术分享

© 2017 ImportNew