# 日志技术——Log4j2.8使用

# 1日志的由来

现状：服务器24小时都在运行，但是，不可能找一个技术人员，24小时监控服务器运行。

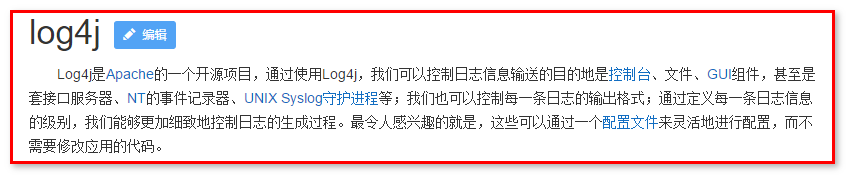
需求：让程序自己监控自己，如果发生错误，记录下来，保存成一个文件，方便程序员后期，查阅文件，排查系统中错误。

解决方案：日志技术

总结：日志记录服务器的运行状态，将数据保存成文件，方便程序员后期排查错误。

日志技术有很多种我们今天学习的是Log4j！

# 2什么是log4j？

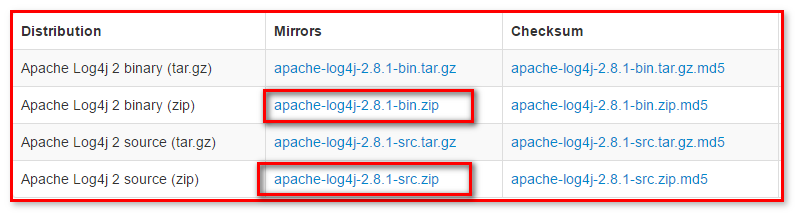


使用准备:

官网：http://logging.apache.org/log4j/2.x/

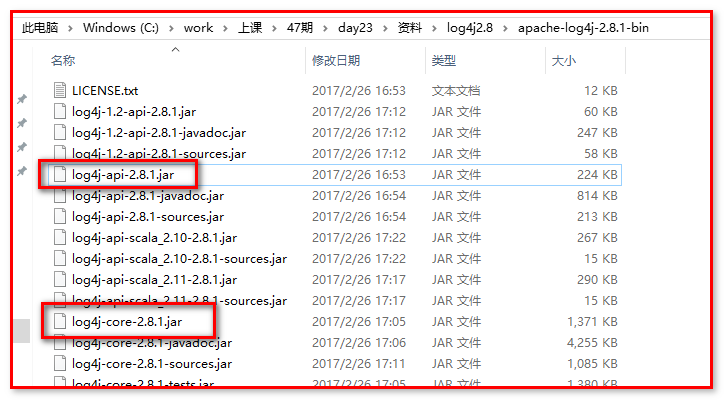
下载log4j：  


选择版本：

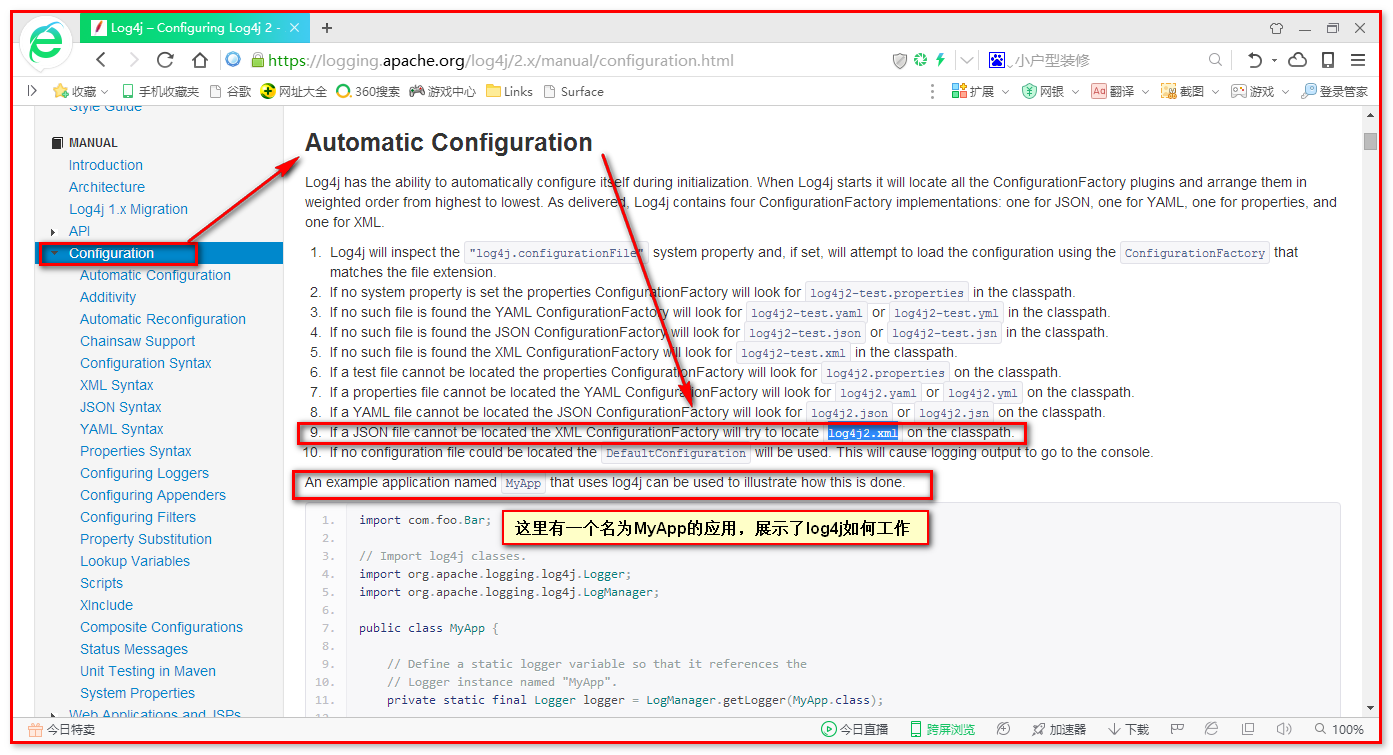




文件目录：



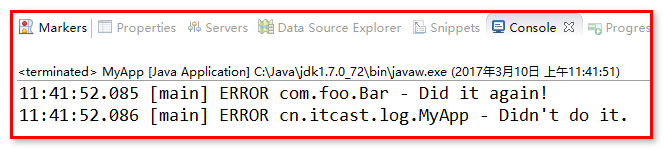
# 3 Log4j快速入门：



测试代码：

|  |
| --- |
| **package** cn.itcast.log;  **import** com.foo.Bar;    // Import log4j classes.  **import** org.apache.logging.log4j.Logger;  **import** org.apache.logging.log4j.LogManager;    **public** **class** MyApp {    // Define a static logger variable so that it references the  // Logger instance named "MyApp".  **private** **static** **final** Logger ***logger*** = LogManager.*getLogger*(MyApp.**class**);    **public** **static** **void** main(**final** String... args) {    // Set up a simple configuration that logs on the console.    ***logger***.trace("Entering application.");  Bar bar = **new** Bar();  **if** (!bar.doIt()) {  ***logger***.error("Didn't do it.");  }  ***logger***.trace("Exiting application.");  }  } |
| **package** com.foo;  **import** org.apache.logging.log4j.Logger;  **import** org.apache.logging.log4j.LogManager;    **public** **class** Bar {  **static** **final** Logger ***logger*** = LogManager.*getLogger*(Bar.**class**.getName());    **public** **boolean** doIt() {  ***logger***.entry();  ***logger***.error("Did it again!");  **return** ***logger***.exit(**false**);  }  } |
| 配置文件：log4j2.xml  <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <Configuration status=*"WARN"*>  <Appenders>  <Console name=*"Console"* target=*"SYSTEM\_OUT"*>  <PatternLayout pattern=*"%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"*/>  </Console>  </Appenders>  <Loggers>  <Root level=*"error"*>  <AppenderRef ref=*"Console"*/>  </Root>  </Loggers>  </Configuration> |

效果：



1 导入jar包

2 需要配置文件log4j2.xml

3 核心对象必然是logger

# 4 Log4j核心对象：

日志需求：将程序运行的过程记录下来，方便程序员后期查看，排查错误。

日志记录器：logger ——IO 输入流

输出源：Appender——IO输出流

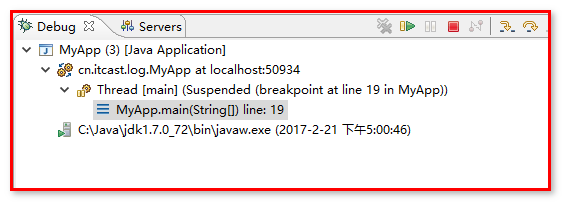
布局：Layout——按照什么样的格式输出日志的内容

## 4.1 日志记录器：Logger

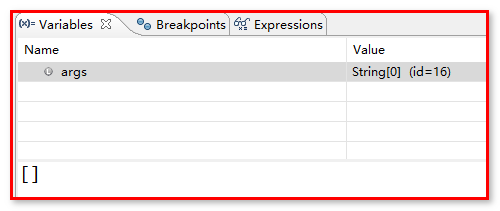
日志级别：

现状：日志的内容是不同

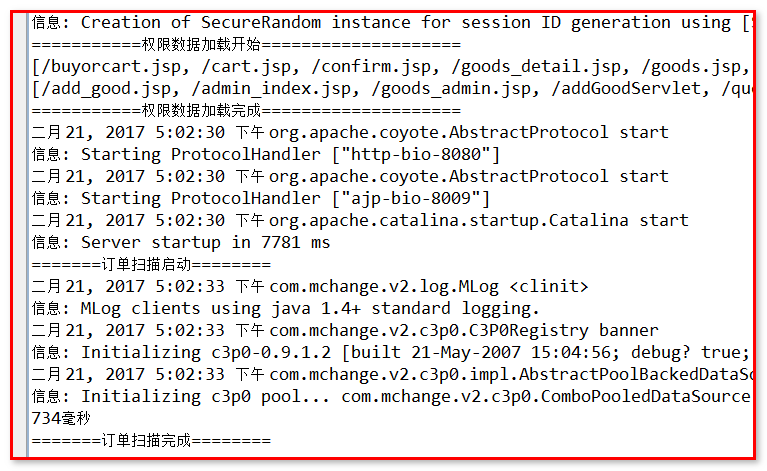
1 程序的堆栈跟踪信息：



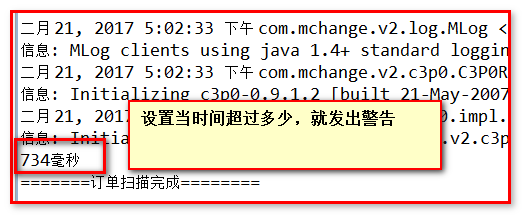
2 调试信息：



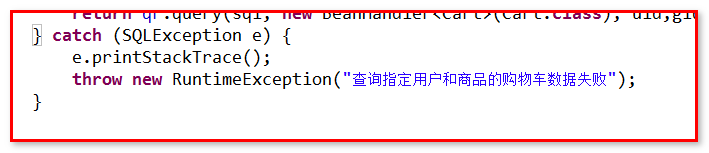
3 服务器运行过程的信息：



4 程序的警告信息：



5 程序异常信息:



6 程序的致命错误：（应用程序一打开就报错）

需求：根据不同的需求，需要的日志内容，也是不同。所以，将日志的内容，进行分级。

分级之后的好处：

可以，根据等级控制哪些日志可以输出，哪些日志不需要输出。

trace： 是追踪，就是程序推进以下，你就可以写个trace输出，所以trace应该会特别多，不过没关系，我们可以设置最低日志级别不让他输出。

debug： 调试么，一般就只用这个作为最低级别，trace压根不用。是在没办法就用eclipse或者idea的debug功能就好了么。

info： 输出一下你感兴趣的或者重要的信息，这个用的最多了。

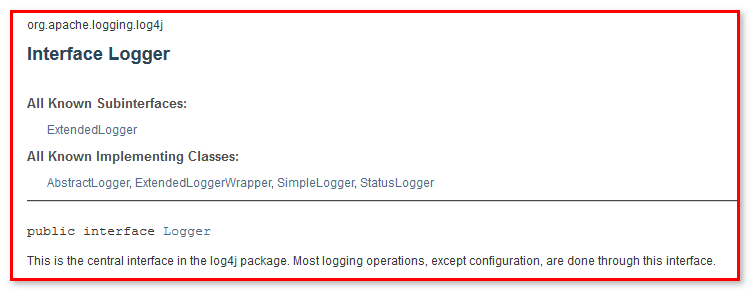
warn： 有些信息不是错误信息，但是也要给程序员的一些提示，类似于eclipse中代码的验证不是有error 和warn（不算错误但是也请注意，比如以下depressed的方法）。

error： 错误信息。用的也比较多。

fatal： 级别比较高了。重大错误，这种级别会直接停止程序！

级别 trace<debug<info<warn<error<fatal

API文档：



记录日志的方法：

***logger***.trace("打印跟踪信息");

***logger***.debug("打印调试信息");

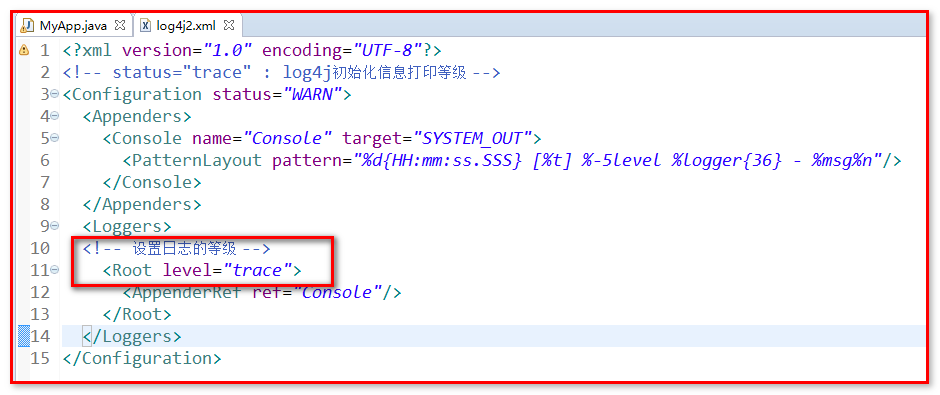
***logger***.info("打印运行信息");

***logger***.warn("打印警告信息");

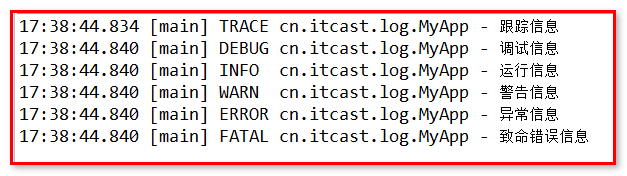
***logger***.error("打印异常信息");

***logger***.fatal("打印致命错误信息");

代码演示设置日志级别控制日志输出：



效果：



## 4.2 什么是输出源？

现状：日志运行，但是，向控制台输出，但我们目的，向文件输出

解决：指定输出的目的地——输出源。

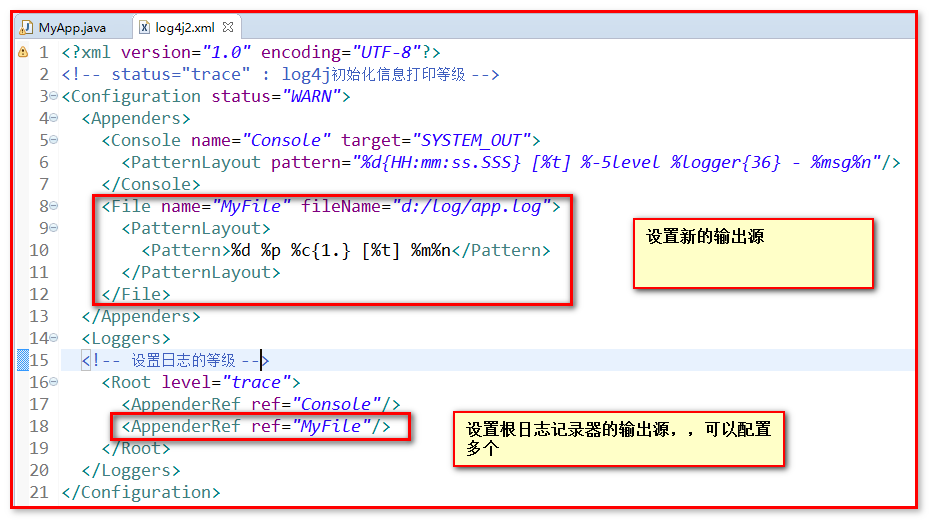
ConsoleAppender：日志输出控制台



FileAppender：日志输出到文件



代码演示：



新需求：让日志记录器中的数据，可以有多个输出源输出

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 配置 status：初始化信息打印等级 -->  <Configuration status=*"warn"* name=*"Mylog4j2"*>  <!-- 设置输出源 -->  <Appenders>  <!-- 输出到文件的输出源 -->  <File name=*"MyFile"* fileName=*"d:/log/app.log"*>  <PatternLayout>  <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>  </PatternLayout>  </File>  <!-- 输出到控制台的输出源 -->  <Console name=*"STDOUT"* target=*"SYSTEM\_OUT"*>  <PatternLayout pattern=*"%m%n"*/>  </Console>  </Appenders>  <!-- 配置日志记录器 -->  <Loggers>  <!-- 默认日志记录器 -->  <Root level=*"trace"*>  <AppenderRef ref=*"MyFile"*/>  <AppenderRef ref=*"STDOUT"*/>  </Root>  </Loggers>  </Configuration> |

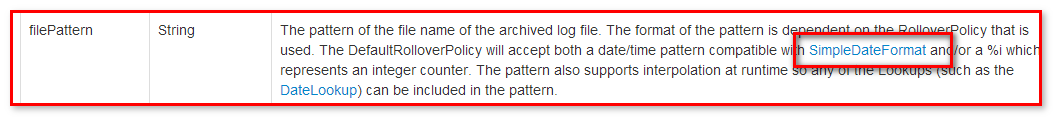
新需求：随着服务器运行，日志的内容，越来越多，文件越来越大（问题：打开慢内存占用多，查询错误发生的位置）。

解决方案：尽量让日志文件，小一点。只能将日志内容分开存放

当一个日志文件到达一定的大小的时候或经过一定时间后——创建一个新的日志文件：

RollingFileAppender：







测试代码：

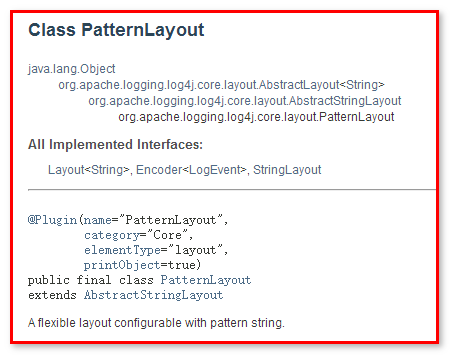
|  |
| --- |
| **package** cn.itcast.test;  **import** org.apache.logging.log4j.LogManager;  **import** org.apache.logging.log4j.Logger;  **public** **class** MyApp {    **private** **static** **final** Logger ***logger*** = LogManager.*getLogger*(MyApp.**class**);    **public** **static** **void** main(**final** String... args) {    **for** (**int** i = 0; i < 100; i++) {    **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  ***logger***.trace("打印跟踪信息");  ***logger***.debug("打印调试信息");  ***logger***.info("打印运行信息");  ***logger***.warn("打印警告信息");  ***logger***.error("打印异常信息");  ***logger***.fatal("打印致命错误信息");  }    }  } |

Xml配置：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <Configuration status=*"WARN"*>  <Appenders>  <Console name=*"Console"* target=*"SYSTEM\_OUT"*>  <PatternLayout pattern=*"%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"*/>  </Console>  <!-- <File name="MyFile" fileName="d:/log/app.log">  <PatternLayout>  <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>  </PatternLayout>  </File> -->  <RollingFile name=*"RollingFile"* fileName=*"d:/log/app.log"*  filePattern=*"d:/log/$${date:yyyy-MM}/app-%d{yyyy-MM-dd-HH-mm}-%i.log.gz"*>  <PatternLayout>  <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>  </PatternLayout>  <Policies>  <TimeBasedTriggeringPolicy />  <SizeBasedTriggeringPolicy size=*"10 KB"*/>  </Policies>  </RollingFile>  </Appenders>  <Loggers>  <Root level=*"trace"*>  <AppenderRef ref=*"Console"*/>  <!-- <AppenderRef ref="MyFile"/> -->  <AppenderRef ref=*"RollingFile"*/>  </Root>  </Loggers>  </Configuration> |

## 4.3 什么是布局（layout）？

我们使用的是自定义布局：PatternLayout



这个类的参数配置:如下，有兴趣可以自己玩。

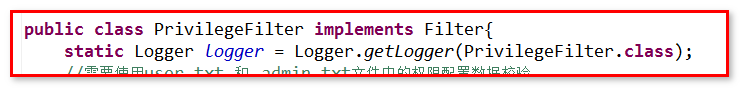
|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **说明** | **例子** | |
| %c | 列出logger名字空间的全称，如果加上{<层数>}表示列出从最内层算起的指定层数的名字空间 | log4j配置文件参数举例 | 输出显示媒介 |
| 假设当前logger名字空间是“a.b.c” | |
| %c | a.b.c |
| %c{2} | b.c |
| %20c | （若名字空间长度小于20，则左边用空格填充） |
| %-20c | （若名字空间长度小于20，则右边用空格填充） |
| %.30c | （若名字空间长度超过30，截去多余字符） |
| %20.30c | （若名字空间长度小于20，则左边用空格填充；若名字空间长度超过30，截去多余字符） |
| %-20.30c | （若名字空间长度小于20，则右边用空格填充；若名字空间长度超过30，截去多余字符） |
| %C | 列出调用logger的类的全名（包含包路径） | 假设当前类是“org.apache.xyz.SomeClass” | |
| %C | org.apache.xyz.SomeClass |
| %C{2} | xyz.SomeClass |
| %d | 显示日志记录时间，{<日期格式>}使用ISO8601定义的日期格式 | %d{yyyy/MM/dd HH:mm:ss,SSS} | 2005/10/12 22:23:30,117 |
| %d{ABSOLUTE} | 22:23:30,117 |
| %d{DATE} | 12 Oct 2005 22:23:30,117 |
| %d{ISO8601} | 2005-10-12 22:23:30,117 |
| %F | 显示调用logger的源文件名 | %F | MyClass.java |
| %l | 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数 | %l | MyClass.main(MyClass.java:129) |
| %L | 显示调用logger的代码行 | %L | 129 |
| %m | 显示输出消息 | %m | This is a message for debug. |
| %M | 显示调用logger的方法名 | %M | main |
| %n | 当前平台下的换行符 | %n | Windows平台下表示rn UNIX平台下表示n |
| %p | 显示该条日志的优先级 | %p | INFO |
| %r | 显示从程序启动时到记录该条日志时已经经过的毫秒数 | %r | 1215 |
| %t | 输出产生该日志事件的线程名 | %t | MyClass |
| %x | 按NDC（Nested Diagnostic Context，线程堆栈）顺序输出日志 | 假设某程序调用顺序是MyApp调用com.foo.Bar | |
| %c %x - %m%n | MyApp - Call com.foo.Bar. com.foo.Bar - Log in Bar MyApp - Return to MyApp. |
| %X | 按MDC（Mapped Diagnostic Context，线程映射表）输出日志。通常用于多个客户端连接同一台服务器，方便服务器区分是那个客户端访问留下来的日志。 | %X{5} | （记录代号为5的客户端的日志） |
| %% | 显示一个百分号 | %% | % |

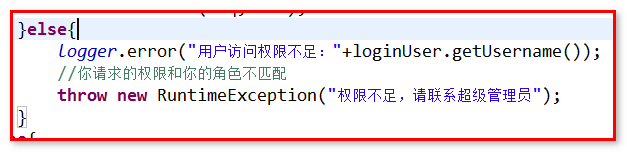
# 5 estore日志应用：

我们需要记录什么日志？

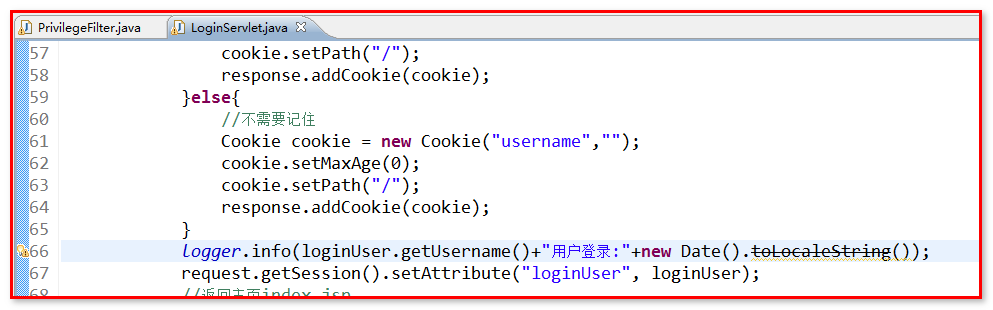
答：我们的系统开发常常会涉及到系统错误日志，用户操作日志和系统运行日志这三大类日志的记录。

系统错误日志——在所有的try catch 中加日志。





用户操作日志——收集用户的偏好和习惯（登陆时间，使用浏览器）



系统运行日志——项目启动关闭过程

