> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

日志虽然表面看起来很简单，却非常重要，日志是我们排查问题的非常重要的手段。我们不仅要掌握日志的基本用法，更要懂得不该在哪里打日志，该在哪些地方打日志，并思考忘记打日志怎么办。

《手册》第 26 到 27 页，提出了很多日志规约，其中比较重要的有：

> 【强制】应用中不要直接使用日志系统的 API，而是应该依赖日志架构 SLF4J 中的 API，使用门面模式的日志架构，有利于维护各个类的日志处理方式统一。
>
> 【强制】日志至少要保留 15 天，因为有些异常具备以 "周" 为频次的特点。
>
> 【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity =false。
>
> 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。
>
> 等。

看到这些我们该思考下面几个问题：

- 我们门面模式的是什么？它的使用场景和优势是什么？
- 为什么会重复打日志？
- 不该在哪里打日志？
- 该在哪些地方打日志呢？
- 如果忘记打日志却着急排查问题怎么办？

只有主动去思考和日志规约相关的问题，这样才能知其然，才能学得更多更深入，在使用时才能更灵活。

本节将带领大家学习和分析这些问题。

2.1 日志是什么？为什么要打印日志？

日志文件是什么？

计算机领域，日志文件是记录发生在运行中的操作系统或其他软件中的事件或消息的文件。

为什么要打印日志？

打印日志的主要目的是为了监测系统状态、方便测试、方便排查问题。

当测试时遇到和预计不符的情况，看日志是解决问题的最常用手段。设想一下如果没有日志，线上出现问题排查起来是不是更困难？

很多监控系统都是通过监控日志来预警，很多线上问题通过日志来排查，很多测试人员依靠日志来辅助测试。

2.2 门面模式

很多同学可能有这样的一种体会：专门去学设计模式，看会了也容易忘。其实带着问题学知识，印象会更加深刻。

比如此日志规约章节就涉及到了门面模式，那么这就是我们学习和理解门面模式的好机会。

学习设计模式我们主要思考以下几个问题：

- 为什么会出现这种设计模式？
- 这种设计模式的核心思想是什么？
- 这种设计模式的使用场景有哪些？

- 这种设计模式的优缺点有哪些?

那么我们依次来回答这几个问题，我们可以参考《设计模式之禅》，可以参考菜鸟教程，甚至直接通过搜索博客。

2.2.1 门面设计模式是为了解决什么问题呢?

《代码整洁之道》第二章 有意义的命名，讲到：命名要名副其实。

> 我们想要强调，这件事很严肃。选一个好名字要花时间，但是生下来的时间比花费的时间多。注意命名，而且一旦发现有更好的名称，就换掉旧的。这么做，读你代码的人（包括你自己）都会更开心。

同样地，对于门面设计模式，顾名思义，是为了隐藏系统的复杂性，向使用方提供统一的可以访问系统的接口。

2.2.2 门面模式的核心思想是什么?

门面设计模式在客户端和复杂的系统之间加一层，在这一层将调用的顺序和依赖的关系调整好。

2.2.3 门面模式的主要使用场景有哪些?

为复杂的子系统提供外界访问的模块。

预防低水平的开发人员带来的风险。

2.2.4 门面模式的优缺点分别是什么?

优点：减少了系统的相互依赖；提高了灵活性和安全性

缺点：不符合开闭原则，如果接口要修改无法通过继承和覆写来解决，只能修改门面代码，影响面比较大。

2.2.5 再回到为什么应用应该依赖使用日志架构 SLF4J 中的 API，使用门面设计模式的日志架构?

SLF4J 的全称为： The Simple Logging Facade for Java， 即 Java 简单日志门面。

使用 SLF4J 编写代码，开发人员不需要关注不同的日志框架的差异，各日志框架对 SLF4J 做适配。由于没有具体依赖某个日志框架，如果系统出于安全、性能等原因想更换另外一个新的日志框架就轻而易举。

关于门面设计模式的具体编码，大家可以参考《设计模式之禅》的 23 章：门面模式。

2.3 日志级别

2.3.1 日志级别规范

常用的日志级别分为：ERROR、WARN、INFO、DEBUG、TRACE。

ERROR 日志的使用场景是：影响到程序正常运行或影响到当前请求正常运行的异常情况。比如打开配置失败、调用二方或者三方库抛出异常等。

WARN 日志 的使用场景是：不应该出现，但是不影响程序正常运行，不影响请求正常执行的情况。如找不到某个配置但是使用了默认配置，比如某些业务异常。

INFO 日志的使用场景是：需要了解的普通信息，比如接口的参数和返回值，异步任务的执行时间和任务内容等。

DEBUG 日志的使用场景是：所有调试阶段想了解的信息。比如无法进行远程 DEBUG 时，添加 DEBUG 日志在待研究的函数的某些位置打印参数和中间数据等。

如 Spring `org.springframework.boot.SpringApplication#load` 函数就用到了 DEBUG 日志：

```
protected void load(ApplicationContext context, Object[] sources) {
   if (logger.isDebugEnabled()) {
      logger.debug("Loading source " +
StringUtils.arrayToCommaDelimitedString(sources));
   }
   BeanDefinitionLoader loader =
createBeanDefinitionLoader(getBeanDefinitionRegistry(context), sources);
   if (this.beanNameGenerator != null) {
      loader.setBeanNameGenerator(this.beanNameGenerator);
   }
   if (this.resourceLoader != null) {
      loader.setResourceLoader(this.resourceLoader);
   }
   if (this.environment != null) {
      loader.setEnvironment(this.environment);
   }
   loader.load();
}
```

TRACE 日志 的使用场景是：非常详细的系统运行信息，比如某个中间件读取配置，启动完成等。

如 Spring 源码中，`org.springframework.boot.env.RandomValuePropertySource#getProperty` 就打印了 TRACE 级别的日志：

```
@Override
public Object getProperty(String name) {
   if (!name.startsWith(PREFIX)) {
      return null;
   }
   if (logger.isTraceEnabled()) {
      logger.trace("Generating random property for '" + name + "'");
   }
   return getRandomValue(name.substring(PREFIX.length()));
}
```


实际业务开发中 TRACE 级别的日志很少使用。

另外通过上面两个例子，我们看到调用打印语句前，都会先判断该级别的日志是否开启。大家可以先思考下为什么这么做？文章后半部将重点对此进行解释。

2.3.2 规范的日志级别

规范打日志可以让根据日志定位问题的同学能够抓住重点，比如优先关注错误日志，其次是警告。

【推荐】在自测或提测之后上线前一定要注意 warn 级别以上的日志，特别是 error 日志。

通常我们会将 ERROR 日志专门输出到一个 error.log 文件。调试时通过 `tail -f error.log` 随时监控出现的错误日志。

希望大家**一定**要养成这种好习惯。通过这种习惯，可以尽可能早地发现问题，避免悲剧。

> 有些报错虽然影响实际的功能，但是由于不影响主流程，很多人就没在意。
>
> 实际开发中，由于习惯查看错误日志，某次自测时发现某个刚上线的功能的某项配置错误，导致某项功能后台启动失败，及时反馈给该服务的负责人，收到了点赞。

2.3.3 不规范日志级别带来的问题

日志级别看似非常简单，但是很多人可能会打错日志级别，给调试和定位问题带来很大不便。

比如在实际开发中，发现一个**功能失败，开发人员居然打印的是 info 级别日志**！

某项功能失败，却找不到任何错误和警告级别的日志，坑队友...

还有将严重的错误打印 WARN 级别日志，导致没有及早引起重视出现故障。

日志的最基本用法比较简单，大家自行学习，这里就不再举例了。这里举两个例子来说明如何学习日志相关知识，如何分析相关问题。

3.1 叠加性如何理解?

前面提到：为了避免重复打印日志，浪费磁盘空间，务必在 `log4j.xml` 中设置 `additivity=false`。

那么为什么会重复打印日志呢?

我们分别从官方文档、源码的角度对该问题进行学习。

3.1.1 官方文档大法

这类问题我们需要从官方文档或者源码层面去寻找答案。

`MyApp` 类:

```
import com.foo.Bar;

import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

public class MyApp {

    private static final Logger logger = LogManager.getLogger(MyApp.class);

    public static void main(final String... args) {

        logger.trace("Entering application.");
        Bar bar = new Bar();
        if (!bar.doIt()) {
            logger.error("Didn't do it.");
        }
        logger.trace("Exiting application.");
    }
}
```

`Bar` 类:

```
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);
  }
}
```

如果配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console >
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger >
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="Console" />
    </Root>
  </Loggers>
</Configuration>
```


会输出这样的结果：

```
15:12:13.226 [main] TRACE com.foo.Bar - Enter
15:12:13.226 [main] TRACE com.foo.Bar - Enter
15:12:13.229 [main] ERROR com.foo.Bar - Did it again!
15:12:13.229 [main] ERROR com.foo.Bar - Did it again!
15:12:13.229 [main] TRACE com.foo.Bar - Exit with(false)
15:12:13.229 [main] TRACE com.foo.Bar - Exit with(false)
15:12:13.230 [main] ERROR MyApp - Didn't do it.
```

通过该范例和文档的描述，我们可以看到 `com.foo.Bar` 的消息都输出了两次。

这是因为第一次是使用到了 `com.foo.Bar` 这个 logger，然后又用到了它的父 logger。

日志事件被传递到 root logger 的 appender ， 被写到了 console 中，导致两次输出， 这就是所谓的叠加性。

叠加性是一个非常方便的特性，低层次的 logger 甚至都不需要配置 appender，就可以输出。但是很多情况下并不希望有这种默认的行为，那么可以通过设置 logger 的 additivity 属性为 false 来关闭叠加性。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console >
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger >
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

那么 logback 是否也提供了类似的设置呢?

> 一个 logger 可以被关联到多个 appender 上，对于 logger 的每个启用了的记录请求，都将被发送到 logger 里的全部 appender 及更高层次的 appender 中。换句话说，appender 叠加地继承了 logger 的层次等级。如果 root 日志有一个控制台 console，那么所有启动的日志至少都会输出到控制台中。如果 root 下还有一个叫 L 的 logger，它含有一个文件 appender，那么 L 和 L 的子层次 Logger 的所有日志都将会打印到控制台和文件中。
>
> 将 logger 的 additivity 属性设置为 false，则可以取消这种默认的 appender 累加行为。

另外关于 Appender 的叠加性还有如下描述：

Appender 叠加性的含义是：Logger L 的日志的输出会发送给 L 及其祖先的全部 appender。

然而，如果 Logger L 的某个祖先 Logger P 设置叠加标识为 false，那么，Logger L 的输出会发送给 Logger L 和 Logger P (含 P) 的所有 appender，但是不会发送给 Logger P 的任何祖先的 appender。

这一点和 log4j 非常相似。

3.1.2 源码模式

我们如果使用 log4j（logback 会略有不同，但是思路都是相似的），通过断点调试我们发现代码的核心源码如下：

`org.apache.logging.log4j.core.config.LoggerConfig#log`

```
protected void log(final LogEvent event, final LoggerConfigPredicate predicate)
{
    if (!isFiltered(event)) {
        processLogEvent(event, predicate);
    }
}
```

`processLogEvent` 函数源码:

```
private void processLogEvent(final LogEvent event, final LoggerConfigPredicate
predicate) {
   event.setIncludeLocation(isIncludeLocation());
   if (predicate.allow(this)) {
       callAppenders(event);
   }
  logParent(event, predicate);
}
```

其中的 `callAppenders` 函数源码:

```
protected void callAppenders(final LogEvent event) {
    final AppenderControl[] controls = appenders.get();

    for (int i = 0; i < controls.length; i++) {
        controls[i].callAppender(event);
    }
}
```


如果 additive 为 true 且 parent Logger 不为 null，则调用 parent 的 log 函数。

```
private void logParent(final LogEvent event, final LoggerConfigPredicate
predicate) {
    if (additive && parent != null) {
        parent.log(event, predicate);
    }
}
```

从这几个核心函数我们可以看出，如果 additive 为 false，则不会传递，parent Logger 不会继续记录日志。

很多人学到此可能会有些不以为意，因为很多人加入团队后一般都是维护现有的项目，没有机会去配置日志，也没注意去观察日志文件的区别。

然而在工作中的确发现有的服务 root 日志超大，很多日志被重复打印，造成了资源的浪费的情况。

3.2 为什么推荐使用占位符方式打印日志?

《手册》中规定有一条关于占位符的规定：

> 【强制】在日志输出时，字符串变量之间的拼接使用占位符的方式
>
> 说明：因为 String 字符串拼接会使用 StringBuilder 的 append () 方式，有一定的性能损耗。使用占位符可以有效提高性能。

很多人会把这一段话当做说服自己或者别人使用占位符的依据，然后就没然后了...

如果我们遇到类似的新问题，或者我们没看到《手册》中这条规定，我们如何学习和分析？

- **我们怎么知道 “String 字符串拼接会使用 StringBuilder 的 append () 方式”?**
- 俗话说：“尽信书不如无书”。规约中这种说法严谨吗？真的都是使用这一种方式拼接字符串的吗？

我们写一个简单的 DEMO:

```
@Slf4j
public class LogDemo {

    public void first() {
        log.debug("慕课" + "专栏");
    }

    public void second(String website) {
        log.debug("慕课网" + website);

    }

    public void third(String website) {
        if (log.isDebugEnabled()) {
            log.debug("慕课网" + website);
        }
    }
}
```


对源码编译然后反汇编，得到下面反汇编代码：

```
public class com.imooc.basic.log.LogDemo {
  public com.imooc.basic.log.LogDemo();
    Code:
       0: aload_0
       1: invokespecial #1
       4: return

  public void first();
    Code:
       0: getstatic     #2
       3: ldc           #3
       5: invokeinterface #4,  2
      10: return

  public void second(java.lang.String);
    Code:
       0: getstatic     #2
       3: new           #5
```

```
       6: dup
       7: invokespecial #6
      10: ldc           #7
      12: invokevirtual #8
      15: aload_1
      16: invokevirtual #8
      19: invokevirtual #9
      22: invokeinterface #4,  2
      27: return

  public void third(java.lang.String);
    Code:
       0: getstatic     #2
       3: invokeinterface #10,  1
       8: ifeq          38
      11: getstatic     #2
      14: new           #5
      17: dup
      18: invokespecial #6
      21: ldc           #7
      23: invokevirtual #8
      26: aload_1
      27: invokevirtual #8
      30: invokevirtual #9
      33: invokeinterface #4,  2
      38: return

  static {};
    Code:
       0: ldc           #11
       2: invokestatic  #12
       5: putstatic     #2
       8: return
}
```


通过上述反汇编后的代码，我们发现：

如果源码中直接将两个字符串字面量进行拼接，编译时期就会生成新的字符串，并不是通过 StringBuilder 构造的。

上述反汇编后的代码，可以逆向 “翻译” 为：

```
@Slf4j
public class LogDemo {

    public void first() {
        log.debug("慕课专栏");
    }

    public void second(String website) {
        StringBuilder builder = new StringBuilder();
        builder.append("慕课网");
        builder.append(website);
        String result = builder.toString();
```

```
        log.debug(result);
    }

    public void third(String website) {
        if (!log.isDebugEnabled()) {
            return;
        }
        StringBuilder builder = new StringBuilder();
        builder.append("慕课网");
        builder.append(website);
        String result = builder.toString();
        log.debug(result);
    }
}
```

通过反汇编后的代码和上述逆向 “还原” 的 Java 代码，我们可以清楚地知道：如果不加上 `log.isDebugEnabled()` 判断，会因为字符串拼接造成不必要的资源损耗。

但是如果每个日志打印都加上这种判断，代码非常不优雅，因此常见的实现了 `org.slf4j.Logger` 接口的日志框架提供了占位符日志打印方法。

使用占位符的方式，底层会先使用判断逻辑，再去拼接字符串（具体大家可以通过断点调试或者阅读源码来验证），从而避免了不必要的字符串拼接。

【强制】不要用 `System.out.println` 代替日志框架

很多人，尤其是新手喜欢通过打印语句而不是日志系统来打印日志排查错误。在本地调试的场景下，这种情况更加普遍。这是一个非常不专业且很 low 的行为。

以 Tomcat 为例， 使用 `System.out.println` 函数语句将信息输出到 `catalina.out` 文件中，该文件会被清理，否则影响性能。文件的 IO 操作非常耗时，而该函数底层使用了 `java.io.PrintStream#println(java.lang.Object)` 内部使用了同步代码块，非常影响性能。

由于没有输出级别的概念，很多忘记删除的调试打印语句可能都被输出到输出文件中。如果是必须要打印日志的场景，请用日志框架打印。对于暂时性的本地或者测试服务器上排错，建议使用 IDE 的本地或者远程调试功能。

在调试器中可以看到参数的值，看到调用栈，可以通过表达式查看变量的属性等，功能更加强大。

【强制】不要打印敏感信息，如果需要打印可以考虑对敏感信息脱敏处理

很多开发人员没有安全意识，将用户的密码和个人资料、商品的名称和价格等信息在未脱敏的情况下通过日志框架打印到日志文件中。这些都是很大的安全隐患，容易出现安全事故。因此不要打印敏感信息或者将敏感信息进行脱敏后再打印。

【推荐】除非业务需要，尽量不要打印大文本 (含富文本)。如果要打印可以截取前 M 个字符。

大家都知道 I/O 操作非常耗时，在高并发场景下，如果同步打印大文本日志非常影响性能。

另外，很多大文本对排查问题帮助不大，打印该信息的意义不大，因此尽量避免打印该内容或只截取一部分关键信息。

如果公司内容有封装的相关注解，可以在大文本上加上忽略的注解。如果没有尽量避免调用 JSON 转字符串的函数将整个对象打印，可以重写 toString 方法忽略或截取大文本字段的一部分。

大家写代码时，都会接触到打日志的场景。大多数同学不是不会打印日志，而是不知道在哪打日志。打印日志没有章法，很容易遇到问题才发现没有日志，再去补日志去排查问题，非常浪费时间。

那么大家是否深入思考过应该在哪里打印日志呢?

下面给出一些个人建议:

【推荐】用切面或 Filter 在 dubbo 或 Controller 层做切面来打印调用的参数、返回值和响应时间以及捕捉和打印异常日志。

通过统一的日志，将参数和返回值，以及响应时间做切面，当出现问题时，可以极大地辅助我们排查，也能够帮助我们了解接口的响应情况。

可以使用日志框架如 CAT，做统一的 traceId, 方便定位整条调用链路。

【推荐】在依赖的二方或三方接口的参数、返回值和异常处打印日志。

依赖的二方和三方接口对接最容易出错，可以将其请求的参数和接口返回值以及异常信息等通过日志打印输出，方便分析和排查问题。

一般也会使用切面统一打印，或者通过使用支持日志注解的框架来对某些接口的参数和返回值打印日志。

【推荐】 在接收消息的地方打印日志。

我们测试或者排查问题时，如果涉及到消息队列，最先要查的是有没有收到消息，因此接收到消息的地方打印日志比较有用。

【推荐】 在定时任务的开始和结束的地方。

和上面的条目很类似，在定时任务的开始和结束的地方打印日志也很重要。

如果线上某个任务没有执行或者开始但是没有结束，可以通过日志快速排查问题。

【推荐】 在异步任务的开始和结束的地方。

异步任务如果出错，很难排查。在任务开始和结束的地方打印一些关键参数对排查问题有很大的帮助。

在异步任务的异常捕捉地方打印日志也至关重要，将是分析问题的关键。

【推荐】面向测试打印日志。

所谓的**面向测试打印日志**是指：方便测试人员测试开发人员代码而提供的测试。

此日志的级别一般为 DEBUG，一般仅在测试环境可用。

有些测试人员并不会拉取你的代码或者编写单元测试，而只对你的编码进行功能性测试。

此时，测试人员构造测试用例对你的接口进行验证时只能观察到接口的返回值，如果你能提供其中的核心计算函数的日志，将极大方便测试人员去验证功能。

在开发中我们可能会遇到一种非常尴尬的问题，一个问题本地无法复现，需要去生产环境排查问题。

但是我们在关键的函数上没有打印日志，肿么办? 在线等，急...

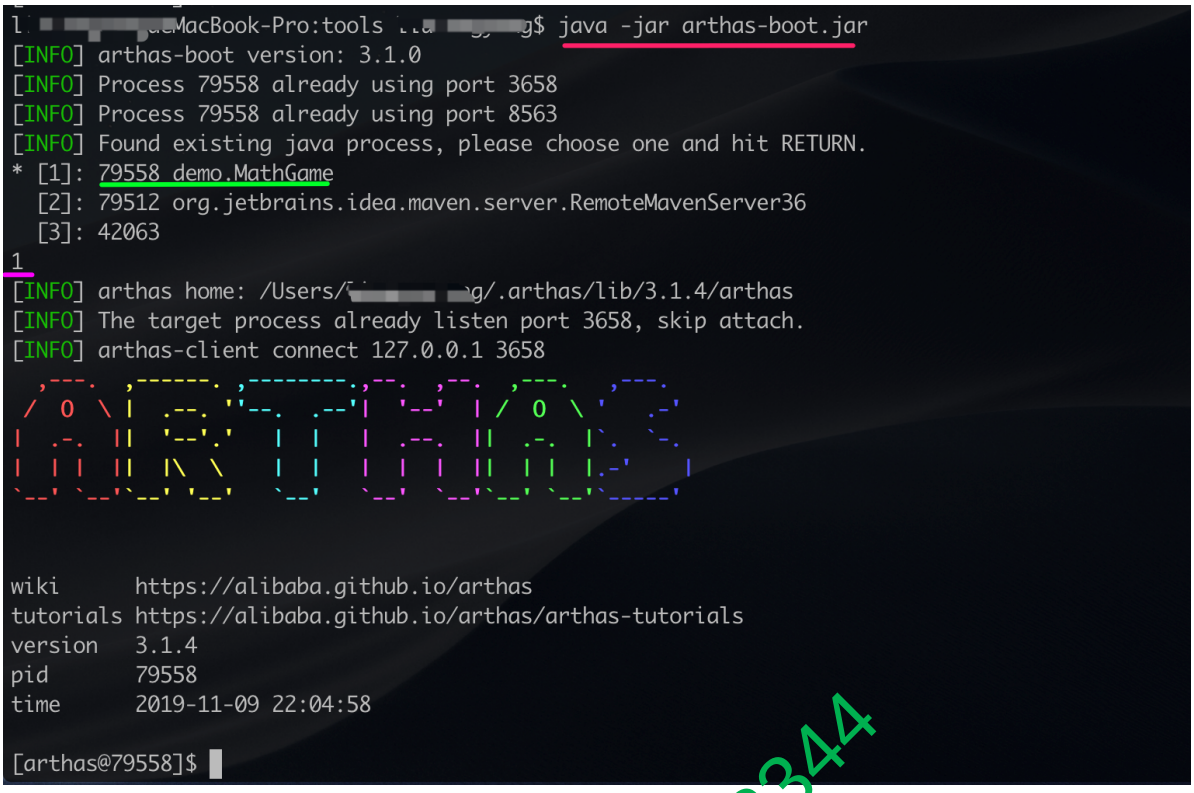
加上日志后重新发布代价比较大，而且无法快速及时地排查分析问题。

此时我们可以借助 Alibaba Java 诊断利器 Arthas 。

安装非常简单，请参考 Arthas Install, 一两行命令搞定！启动后选择所要监控的 Java 进程即可进。

如下图所示，我们可以通过 `java -jar arthas-bbot.jar` 启动 arthas。

启动后它会检测已经存在的 java 进程（下图显示有 3 个），选择我们需要研究的进程序号（此处为 1）回车即可。



自此，我们就可以使用其 watch 命令来观察返回值、抛出的异常、入参等。

查询出参和返回值

```
watch 类路径 函数名 "{params,returnObj}" -x 2
```

如输入：watch demo.MathGame primeFactors "{params,returnObj}" -x 2

```
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 68 ms.
ts=2019-11-09 22:14:05; [cost=2.149321ms] result=@ArrayList[
    @Object[][
        @Integer[70214],
    ],
    @ArrayList[
        @Integer[2],
        @Integer[35107],
    ],
]
ts=2019-11-09 22:14:06; [cost=0.200091ms] result=@ArrayList[
    @Object[][
        @Integer[119344],
    ],
    @ArrayList[
        @Integer[2],
        @Integer[2],
        @Integer[2],
        @Integer[2],
        @Integer[7459],
    ],
]
```

可以试试观察到 `MathGame` 类的 `primeFactors` 函数调用的出参和返回值。

查看异常信息

格式如下：`watch 类路径 函数名 "{params[0],throwExp}" -e -x 2`

使用 watch demo.MathGame primeFactors “{params [0],throwExp}” -e -x 2 可以抓取到官方案例的该函数调用的异常信息:

```
ts=2019-11-09 22:18:36; [cost=0.211415ms] result=@ArrayList[
    @Integer[-26077],
    java.lang.IllegalArgumentException: number is: -26077, need >= 2
    at demo.MathGame.primeFactors(MathGame.java:46)
    at demo.MathGame.run(MathGame.java:24)
    at demo.MathGame.main(MathGame.java:16)
,
]
```

watch 命令还支持条件表达式，支持根据耗进行过滤等强大功能，建议大家一定要敢于尝试，通过官方案例或自己项目代码去训练。

下面给出实际开发中的典型误用的几种情况：

用打印语句或者 `e.printStackTrace()` 打印日志

这两种打印方式非常不专业，容易造成日志丢失，污染控制日志级别，甚至还可能造成其他问题。

占位符误用

对应的函数签名为：

```
public void info(String msg, Throwable t);
```


最终导致占位符被当做普通字符串处理。

参数格式错误

```
log.info( user.getAge()+"", user.getName());
```

此时实际对应的函数为：

```
public void info(String format, Object arg);
```

由于第一个 format 参数没有格式化占位符，导致第二个参数打印不出来。

空指针异常

实际开发中，还有一些同学居然会这么写

```
if (result == null || result.getData() == null) {
    log.info("result={},resultData:{}", result, result.getData());
}
```

如果 result == null 为 true，难道不会报空指针异常吗?

还有很多花式踩坑姿势，这里就不一一列举了，也欢迎大家留言补充。

建议大家一定要注重 IDE 的警告；开发时日志的函数有较多重载方法，没把握时可以点到源码中核对；多使用 findbugs 来检查代码。

本节我们我们重点分析了如何学习日志规约，思考了为何要打印日志，并讲述了不该在哪里打印日志，该在哪里打印日志的问题。还介绍了如果没有日志还需要紧急排查问题，大家可以使用 arthas 来实现；最后给出了常见的日志形式。

希望大家在学习和开发过程中多思考，多实战，能够够举一反三。

建议大家多读读 log4j 和 logback 的官方手册和源码，更系统地学习日志的用法、理解日志的原理。

下一节我们将学习单元测试相关知识。

1、为什么 Spring 中会有下面代码这种先判断后打印日志的情况，而不直接使用占位符方式打印日志呢?

```
if (logger.isDebugEnabled()) {
   logger.debug(
         "Loading source " + StringUtils.arrayToCommaDelimitedString(sources));
}
```


2、学习 `System.out.println` 的源码

}