目录

JavaLib

Introduction 1.1
Dozer 使用指南 1.2
Freemark 使用指南 1.3
细说 Java 主流 JSON 库 1.4
细说 Java 主流日志工具库 1.5
细说 Java 主流工具包 1.6
JavaMail 使用指南 1.7
Jsoup 使用指南 1.8
JUnit5 使用指南 1.9
Lombok 使用指南 1.10
Mockito 使用指南 1.11
Reflections 使用指南 1.12
Thumbnailator 使用指南 1.13
ZXing 使用指南 1.14

JavaTool

构建工具 2.1
- Maven 教程 2.1.1
  - Maven 快速指南 2.1.1.1
  - Maven 教程之 pom.xml 详解 2.1.1.2
  - Maven 教程之 settings.xml 详解 2.1.1.3
  - Maven 实战问题和最佳实践 2.1.1.4
  - Maven 教程之发布 jar 到私服或中央仓库 2.1.1.5
  - Maven 插件之代码检查 2.1.1.6
- Ant 简易教程 2.1.2

Java IDE 2.2
- Intellij IDEA 使用指南 2.2.1
- Eclipse 使用指南 2.2.2

JavaCore 教程

JavaWeb 教程

Spring 教程

Spring Boot 教程

- Java 教程
  - JavaLib
  - JavaTool
  - JavaCore 教程
  - JavaWeb 教程
  - Spring 教程
  - Spring Boot 教程
  - 相关技术栈
  - 学习资源

Java 教程

> Java Tutorial 是本人在 Java 技术领域的十年积累。
>
> - 项目同步维护：Github | Gitee
> - 电子书阅读：Github Pages | Gitee Pages

JavaLib

> 各种主流 Java 库的应用。

- Dozer 使用指南
- Freemark 使用指南
- Java 与 JSON
- 细说 Java 主流日志工具库
- 细说 Java 主流工具包
- JavaMail 使用指南
- Jsoup 使用指南
- JUnit5 使用指南
- Lombok 使用指南
- Mockito 使用指南
- Reflections 使用指南
- Thumbnailator 使用指南
- ZXing 使用指南

JavaTool

> 各种主流 Java 工具的应用。

- 构建工具
  - Maven 教程
    - Maven 快速指南
    - Maven 教程之 pom.xml 详解
    - Maven 教程之 settings.xml 详解

        - Maven 实战问题和最佳实践
        - Maven 教程之发布 jar 到私服或中央仓库
        - Maven 插件之代码检查
    - Ant 简易教程
- Java IDE
    - Intellij IDEA 使用指南
    - Eclipse 使用指南
- 测试
    - Jmeter 应用指南

JavaCore 教程

> Java 核心技术教程。内容包含：Java 基础特性、Java 高级特性、Java 并发、JVM、Java IO 等。

JavaWeb 教程

> Java Web 技术教程。内容包含：JavaEE、分布式应用技术等。

Spring 教程

> Spring 教程。

Spring Boot 教程

> Spring Boot 教程

相关技术栈

- 🔢 db-tutorial - 是对数据库领域开发经验的总结。内容包含：关系型数据库和 Nosql 理论、Mysql、Redis 等。
- 🎯 algorithm-tutorial - 是对数据结构和算法的总结。内容包含：一些基本的数据结构、算法。
- 🐧 linux-tutorial - 是对 Linux 操作系统的经验总结。内容包含：Linux 常用命令；各种常见软件的 Linux 环境安装配置；运维、部署脚本；Shell、Python 语法教程；Git、Docker 教程。
- 🎨 frontend-tutorial - 前端教程

学习资源

- Java 经典书籍
  - 《Effective Java 中文版》 - 本书介绍了在 Java 编程中 78 条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。同推荐《重构 : 改善既有代码的设计》、《代码整洁之道》、《代码大全》，有一定的内容重叠。
  - 《Java 并发编程实战》 - 本书深入浅出地介绍了 Java 线程和并发，是一本完美的 Java 并发参考手册。
  - 《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》 - 不去了解 JVM 的工程师，和咸鱼有什么区
  - 《Maven 实战》 - 国内最权威的 Maven 专家的力作，唯一一本哦！
- 其他领域书籍
  - 《Redis 设计与实现》 - 系统而全面地描述了 Redis 内部运行机制。图示丰富，描述清晰，并给出大量参考信息，是 NoSQL 数据库开发人员案头必备。
  - 《鸟哥的 Linux 私房菜 （基础学习篇）》 - 本书是最具知名度的 Linux 入门书《鸟哥的 Linux 私房菜基础学习篇》的最新版，全面而详细地介绍了 Linux 操作系统。内容非常全面，建议挑选和自己实际工作相关度较高的，其他部分有需要再阅读。
  - 《Head First 设计模式》 - 《Head First 设计模式》(中文版)共有 14 章，每章都介绍了几个设计模式，完整地涵盖了四人组版本全部 23 个设计模式。
  - 《HTTP 权威指南》 - 本书尝试着将 HTTP 中一些互相关联且常被误解的规则梳理清楚，并编写了一系列基于各种主题的章节，对 HTTP 各方面的特性进行了介绍。纵观全书，对 HTTP“为什么”这样做进行了详细的解释，而不仅仅停留在它是“怎么做”的。
  - 《TCP/IP 详解 系列》 - 完整而详细的 TCP/IP 协议指南。针对任何希望理解 TCP/IP 协议是如何实现的读者设计。
  - 《剑指 Offer：名企面试官精讲典型编程题》 - 剖析了 80 个典型的编程面试题，系统整理基础知识、代码质量、解题思路、优化效率和综合能力这 5 个面试要点。

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 18:32:29

- Dozer 使用指南
  - 简介
  - 安装
    - 引入 jar 包
    - Eclipse 插件
  - 使用
    - 准备
    - Dozer 的配置
    - 与 Spring 整合
  - Dozer 支持的数据类型转换
  - Dozer 的映射配置
    - 用注解来配置映射
    - 用 API 来配置映射
    - 用 XML 来配置映射
  - 参考

Dozer 使用指南

这篇文章是本人在阅读 Dozer 官方文档（5.5.1 版本，官网已经一年多没更新了）的过程中，整理下来我认为比较基础的应用场景。

本文中提到的例子应该能覆盖 JavaBean 映射的大部分场景，希望对你有所帮助。

- 简介
- 安装
  - 引入 jar 包
  - Eclipse 插件
- 使用
  - 准备
  - Dozer 的配置
  - 与 Spring 整合
- Dozer 支持的数据类型转换
- Dozer 的映射配置
  - 用注解来配置映射
  - 用 API 来配置映射
  - 用 XML 来配置映射
- 参考

简介

Dozer 是什么?

Dozer 是一个 JavaBean 映射工具库。

它支持简单的属性映射，复杂类型映射，双向映射，隐式显式的映射，以及递归映射。

它支持三种映射方式：注解、API、XML。

它是开源的，遵从Apache 2.0 协议

安装

引入 jar 包

maven 方式

如果你的项目使用 maven，添加以下依赖到你的 pom.xml 即可：

```
<dependency>
    <groupId>net.sf.dozer</groupId>
    <artifactId>dozer</artifactId>
    <version>5.4.0</version>
</dependency>
```

非 maven 方式

如果你的项目不使用 maven，那就只能发扬不怕苦不怕累的精神了。

使用 Dozer 需要引入 Dozer 的 jar 包以及其依赖的第三方 jar 包。

- Dozer
- Dozer 依赖的第三方 jar 包

Eclipse 插件

Dozer 有插件可以在 Eclipse 中使用(不知道是否好用，反正我没用过)

插件地址: http://dozer.sourceforge.net/eclipse-plugin

使用

将 Dozer 引入到工程中后，我们就可以来小试一番了。

实践出真知，先以一个最简单的例子来展示 Dozer 映射的处理过程。

准备

我们先准备两个要互相映射的类

NotSameAttributeA.java

```
public class NotSameAttributeA {
    private long id;
    private String name;
    private Date date;

    // 省略getter/setter
}
```

NotSameAttributeB.java

```
public class NotSameAttributeB {
    private long id;
    private String value;
    private Date date;

    // 省略getter/setter
}
```

这两个类存在属性名不完全相同的情况：name 和 value。

Dozer 的配置

为什么要有映射配置?

如果要映射的两个对象有完全相同的属性名，那么一切都很简单。

只需要直接使用 Dozer 的 API 即可：

```
Mapper mapper = new DozerBeanMapper();
DestinationObject destObject =
    mapper.map(sourceObject, DestinationObject.class);
```

但实际映射时，往往存在属性名不同的情况。

所以，你需要一些配置来告诉 Dozer 应该转换什么，怎么转换。

注：官网着重建议：在现实应用中，最好不要每次映射对象时都创建一个 `Mapper` 实例来工作，这样会产生不必要的开销。如果你不使用 IoC 容器（如：spring）来管理你的项目，那么，最好将 `Mapper` 定义为单例模式。

映射配置文件

在 `src/test/resources` 目录下添加 `dozer/dozer-mapping.xml` 文件。 `<mapping>` 标签中允许你定义 `<class-a>` 和 `<class-b>` ，对应着相互映射的类。 `<field>` 标签里定义要映射的特殊属性。需要注

意 `<a>` 和 `<class-a>` 对应， `<b>` 和 `<class-b>` 对应，聪明的你，猜也猜出来了吧。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http:/
          xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <mapping date-format="yyyy-MM-dd">
    <class-a>org.zp.notes.spring.common.dozer.vo.NotSameAttribut
    <class-b>org.zp.notes.spring.common.dozer.vo.NotSameAttribut
    <field>
      <a>name</a>
      <b>value</b>
    </field>
  </mapping>
</mappings>
```

与 Spring 整合

配置 DozerBeanMapperFactoryBean

在 `src/test/resources` 目录下添加 `spring/spring-dozer.xml` 文件。

Dozer 与 Spring 的整合很便利，你只需要声明一个 `DozerBeanMapperFactoryBean` ， 将所有的 dozer 映射配置文件作为属性注入到 `mappingFiles` ， `DozerBeanMapperFactoryBean` 会加载这些规则。

spring-dozer.xml 文件范例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema
       http://www.springframework.org/schema/beans/spring-beans-
       default-autowire="byName" default-lazy-init="false">

  <bean id="mapper" class="org.dozer.spring.DozerBeanMapperFacto
    <property name="mappingFiles">
      <list>
        <value>classpath*:dozer/dozer-mapping.xml</value>
      </list>
    </property>
  </bean>
</beans>
```

自动装配

至此，万事具备，你只需要自动装配 `mapper` 。

```
RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:spring/spring-doze
@TransactionConfiguration(defaultRollback = false)
public class DozerTest extends TestCase {
    @Autowired
    Mapper mapper;

    @Test
    public void testNotSameAttributeMapping() {
        NotSameAttributeA src = new NotSameAttributeA();
        src.setId(007);
        src.setName("邦德");
        src.setDate(new Date());

        NotSameAttributeB desc = mapper.map(src, NotSameAttribut
        Assert.assertNotNull(desc);
    }
}
```

运行一下单元测试，绿灯通过。

Dozer 支持的数据类型转换

Dozer 可以自动做数据类型转换。当前，Dozer 支持以下数据类型转换（都是双向的）

- **Primitive to Primitive Wrapper**

  原型(int、long 等)和原型包装类(Integer、Long)

- **Primitive to Custom Wrapper**

  原型和定制的包装

- **Primitive Wrapper to Primitive Wrapper**

  原型包装类和包装类

- **Primitive to Primitive**

  原型和原型

- **Complex Type to Complex Type**

  复杂类型和复杂类型

- **String to Primitive**

  字符串和原型

- **String to Primitive Wrapper**

  字符串和原型包装类

- **String to Complex Type if the Complex Type contains a String constructor**

  字符串和有字符串构造器的复杂类型（类）

- **String to Map**

  字符串和 Map

- **Collection to Collection**

  集合和集合

- **Collection to Array**

  集合和数组

- **Map to Complex Type**

  Map 和复杂类型

- **Map to Custom Map Type**

  Map 和定制 Map 类型

- **Enum to Enum**

  枚举和枚举

- **Each of these can be mapped to one another: java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar, java.util.GregorianCalendar**

  这些时间相关的常见类可以互换：java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar, java.util.GregorianCalendar

- **String to any of the supported Date/Calendar Objects.**

  字符串和支持 Date/Calendar 的对象

- **Objects containing a toString() method that produces a long representing time in (ms) to any supported Date/Calendar object.**

  如果一个对象的 toString()方法返回的是一个代表 long 型的时间数值（单位：ms），就可以和任何支持 Date/Calendar 的对象转换。

Dozer 的映射配置

在前面的简单例子中，我们体验了一把 Dozer 的映射流程。但是两个类进行映射，有很多复杂的情况，相应的，你也需要一些更复杂的配置。

Dozer 有三种映射配置方式：

- 注解方式
- **API** 方式
- **XML** 方式

用注解来配置映射

Dozer 5.3.2版本开始支持注解方式配置映射（只有一个注解：`@Mapping`）。可以应对一些简单的映射处理，复杂的就玩不转了。

看一下`@Mapping`的声明就可以知道，这个注解只能用于元素和方法。

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface Mapping {
  String value() default "";
}
```

让我们来试试吧：

TargetBean.java

```
public class SourceBean {

    private Long id;

    private String name;

    @Mapping("binaryData")
    private String data;

    @Mapping("pk")
    public Long getId() {
        return this.id;
    }

    //其余getter/setter方法略
}
```

TargetBean.java

```
public class TargetBean {

    private String pk;

    private String name;

    private String binaryData;

    //getter/setter方法略
}
```

定义了两个相互映射的 Java 类，只需要在源类中用 `@Mapping` 标记和目标类中对应的属性就可以了。

```
@Test
public void testAnnotationMapping() {
    SourceBean src = new SourceBean();
    src.setId(7L);
    src.setName("邦德");
    src.setData("00000111");

    TargetBean desc = mapper.map(src, TargetBean.class);
    Assert.assertNotNull(desc);
}
```

测试一下，绿灯通过。

官方文档说，虽然当前版本（文档的版本对应 Dozer 5.5.1）仅支持 `@Mapping` ，但是在未来的发布版本会提供其他的注解功能，那就敬请期待吧（再次吐槽一下：一年多没更新了）。

用 API 来配置映射

个人觉得这种方式比较麻烦，不推荐，也不想多做介绍，就是这么任性。

用 XML 来配置映射

需要**强调**的是：如果两个类的所有属性都能很好的互转，可以你中有我，我中有你，不分彼此，那么就不要画蛇添足的在 xml 中去声明映射规则了。

属性名不同时的映射(Basic Property Mapping)

Dozer 会自动映射属性名相同的属性，所以不必添加在 xml 文件中。

```
<field>
  <a>one</a>
  <b>onePrime</b>
</field>
```

字符串和日期映射(String to Date Mapping)

字符串在和日期进行映射时，允许用户指定日期的格式。

格式的设置分为三个作用域级别：

属性级别

对当前属性有效（这个属性必须是日期字符串）

```
<field>
  <a date-format="MM/dd/yyyy HH:mm:ss:SS">dateString</a>
  <b>dateObject</b>
</field>
```

类级别

对这个类中的所有日期相关的属性有效

```
<mapping date-format="MM-dd-yyyy HH:mm:ss">
  <class-a>org.dozer.vo.TestObject</class-a>
  <class-b>org.dozer.vo.TestObjectPrime</class-b>
  <field>
    <a>dateString</a>
    <b>dateObject</b>
  </field>
</mapping>
```

全局级别

对整个文件中的所有日期相关的属性有效。

```
<mappings>
  <configuration>
    <date-format>MM/dd/yyyy HH:mm</date-format>
  </configuration>

  <mapping wildcard="true">
    <class-a>org.dozer.vo.TestObject</class-a>
    <class-b>org.dozer.vo.TestObjectPrime</class-b>
    <field>
      <a>dateString</a>
      <b>dateObject</b>
    </field>
  </mapping>
</mappings>
```

集合和数组映射(Collection and Array Mapping)

Dozer 可以自动处理以下类型的双向转换。

- List to List
- List to Array
- Array to Array
- Set to Set
- Set to Array
- Set to List

使用 hint

如果使用泛型或数组，没有必要使用 hint。

如果不使用泛型或数组。在处理集合或数组之间的转换时，你需要用 `hint` 指定目标列表的数据类型。

若你不指定 `hint` ，Dozer 将认为目标集合和源集合的类型是一致的。

使用 Hints 的范例：

```
<field>
  <a>hintList</a>
  <b>hintList</b>
  <b-hint>org.dozer.vo.TheFirstSubClassPrime</b-hint>
</field>
```

累计映射和非累计映射（Cumulative vs. Non-Cumulative List Mapping）

如果你要转换的目标类已经初始化，你可以选择让 Dozer 添加或更新对象到你的集合中。

而这取决于 `relationship-type` 配置，默认是累计。

它的设置有作用域级别：

- 全局级

```
<mappings>
  <configuration>
     <relationship-type>non-cumulative</relationship-type>
  </configuration>
</mappings>
```

- 类级别

```
<mappings>
  <mapping relationship-type="non-cumulative">
    <!-- 省略 -->
  </mapping>
</mappings>
```

- 属性级别

```
<field relationship-type="cumulative">
  <a>hintList</a>
  <b>hintList</b>

  <a-hint>org.dozer.vo.TheFirstSubClass</a-hint>
  <b-hint>org.dozer.vo.TheFirstSubClassPrime</b-hint>
</field>
```

移动孤儿(Removing Orphans)

这里的孤儿是指目标集合中存在，但是源集合中不存在的元素。

你可以使用 `remove-orphans` 开关来选择是否移除这样的元素。

```
<field remove-orphans="true">
  <a>srcList</a>
  <b>destList</b>
</field>
```

深度映射(Deep Mapping)

所谓深度映射，是指允许你指定属性的属性（比如一个类的属性本身也是一个类）。举例来说

Source.java

```
public class Source {
    private long id;
    private String info;
}
```

Dest.java

```
public class Dest {
    private long id;
    private Info info;
}
```

```
public class Info {
    private String content;
}
```

映射规则

```
<mapping>
  <class-a>org.zp.notes.spring.common.dozer.vo.Source</class-a>
  <class-b>org.zp.notes.spring.common.dozer.vo.Dest</class-b>
  <field>
    <a>info</a>
    <b>info.content</b>
  </field>
</mapping>
```

排除属性(Excluding Fields)

就像任何团体都有捣乱分子，类之间转换时也有想要排除的因子。

如何在做类型转换时，自动排除一些属性，Dozer 提供了几种方法，这里只介绍一种比较通用的方法。

更多详情参考官网。

field-exclude 可以排除不需要映射的属性。

```
<field-exclude>
  <a>fieldToExclude</a>
  <b>fieldToExclude</b>
</field-exclude>
```

单向映射(One-Way Mapping)

注：本文的映射方式，无特殊说明，都是双向映射的。

有的场景可能希望转换过程不可逆，即单向转换。

单向转换可以通过使用 `one-way` 来开启

类级别

```
<mapping type="one-way">
  <class-a>org.dozer.vo.TestObjectFoo</class-a>
  <class-b>org.dozer.vo.TestObjectFooPrime</class-b>
    <field>
      <a>oneFoo</a>
      <b>oneFooPrime</b>
    </field>
</mapping>
```

属性级别

```
<mapping>
  <class-a>org.dozer.vo.TestObjectFoo2</class-a>
  <class-b>org.dozer.vo.TestObjectFooPrime2</class-b>
  <field type="one-way">
    <a>oneFoo2</a>
    <b>oneFooPrime2</b>
  </field>

  <field type="one-way">
    <a>oneFoo3.prime</a>


    <b>oneFooPrime3</b>
  </field>
```

全局配置(Global Configuration)

全局配置用来设置全局的配置信息。此外，任何定制转换都是在这里定义的。

全局配置都是可选的。

- `<date-format>` 表示日期格式
- `<stop-on-errors>` 错误处理开关
- `<wildcard>` 通配符
- `<trim-strings>` 裁剪字符串开关

```
<configuration >

  <date-format>MM/dd/yyyy HH:mm</date-format>
  <stop-on-errors>true</stop-on-errors>
  <wildcard>true</wildcard>
  <trim-strings>false</trim-strings>

  <custom-converters> <!-- these are always bi-directional -->
    <converter type="org.dozer.converters.TestCustomConverter" >
      <class-a>org.dozer.vo.TestCustomConverterObject</class-a>
      <class-b>another.type.to.Associate</class-b>
    </converter>

  </custom-converters>
</configuration>
```

全局配置的作用是帮助你少配置一些参数，如果个别类的映射规则需要变更，你可以 mapping 中覆盖它。

覆盖的范例如下

```
<mapping date-format="MM-dd-yyyy HH:mm:ss">
  <!-- 省略 -->
</mapping>


<mapping wildcard="false">
  <!-- 省略 -->
</mapping>

<mapping stop-on-errors="false">
  <!-- 省略 -->
</mapping>

<mapping trim-strings="true">
  <!-- 省略 -->
</mapping>
```

定制转换(Custom Converters)

如果 Dozer 默认的转换规则不能满足实际需要，你可以选择定制转换。

定制转换通过配置 XML 来告诉 Dozer 如何去转换两个指定的类。当 Dozer 转换这两个指定类的时候，会调用你的映射规则去替换标准映射规则。

为了让 Dozer 识别，你必须实现 `org.dozer.CustomConverter` 接口。否则，Dozer 会抛异常。

具体做法：

(1) 创建一个类实现 `org.dozer.CustomConverter` 接口。

```
public class TestCustomConverter implements CustomConverter {

  public Object convert(Object destination, Object source,
      Class destClass, Class sourceClass) {
    if (source == null) {
      return null;
    }
    CustomDoubleObject dest = null;
    if (source instanceof Double) {
      // check to see if the object already exists
      if (destination == null) {
        dest = new CustomDoubleObject();
      } else {
        dest = (CustomDoubleObject) destination;
      }
      dest.setTheDouble(((Double) source).doubleValue());
      return dest;
    } else if (source instanceof CustomDoubleObject) {
      double sourceObj =
        ((CustomDoubleObject) source).getTheDouble();
      return new Double(sourceObj);
    } else {
      throw new MappingException("Converter TestCustomConverter
          + "used incorrectly. Arguments passed in were:"
          + destination + " and " + source);
    }
  }
```

(2) 在 xml 中引用定制的映射规则

引用定制的映射规则也是分级的，你可以酌情使用。

- 全局级

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <configuration>
    <!-- 总是双向转换的 -->
    <custom-converters>
      <converter type="org.dozer.converters.TestCustomConverter"
        <class-a>org.dozer.vo.CustomDoubleObject</class-a>
        <class-b>java.lang.Double</class-b>
      </converter>

      <!-- You are responsible for mapping everything between
           ClassA and ClassB -->
      <converter
        type="org.dozer.converters.TestCustomHashMapConverter" >
        <class-a>org.dozer.vo.TestCustomConverterHashMapObject</
        <class-b>org.dozer.vo.TestCustomConverterHashMapPrimeObj
      </converter>
    </custom-converters>
  </configuration>
</mappings>
```

- 属性级

```
<mapping>
  <class-a>org.dozer.vo.SimpleObj</class-a>
  <class-b>org.dozer.vo.SimpleObjPrime2</class-b>
  <field custom-converter=
    "org.dozer.converters.TestCustomConverter">
    <a>field1</a>
    <b>field1Prime</b>
  </field>
</mapping>
```

映射的继承(Inheritance Mapping)

Dozer 支持映射规则的继承机制。

属性如果有着相同的名字则不需要在 xml 中配置，除非使用了 `hint`

我们来看一个例子

```
<mapping>
  <class-a>org.dozer.vo.SuperClass</class-a>
  <class-b>org.dozer.vo.SuperClassPrime</class-b>

  <field>
    <a>superAttribute</a>
    <b>superAttr</b>
  </field>
</mapping>

<mapping>
  <class-a>org.dozer.vo.SubClass</class-a>
  <class-b>org.dozer.vo.SubClassPrime</class-b>

  <field>
    <a>attribute</a>
    <b>attributePrime</b>
  </field>
</mapping>

<mapping>
  <class-a>org.dozer.vo.SubClass2</class-a>
  <class-b>org.dozer.vo.SubClassPrime2</class-b>

  <field>
    <a>attribute2</a>
    <b>attributePrime2</b>
  </field>
</mapping>
```

在上面的例子中 SubClass、SubClass2 是 SuperClass 的子类；

SubClassPrime 和 SubClassPrime2 是 SuperClassPrime 的子类。

superAttribute 和 superAttr 的映射规则会被子类所继承，所以不必再重复的在子类中去声明。

参考

Dozer 官方文档 | Dozer 源码地址

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 15:50:03

- Freemark 使用指南
  - 简介
    - 什么是 Freemark？
  - 入门
    - 创建 Configuration 实例
    - 创建数据模型
    - 获取模板
    - 合并模板和数据模型
    - 完整示例
  - 基础
    - 数值
    - 类型
    - 模板
  - 参考资料

Freemark 使用指南

> FreeMarker 是一款 模板引擎： 即一种基于模板和要改变的数据，并用来生成输出文本(HTML 网页，电子邮件，配置文件，源代码等)的通用工具。 它不是面向最终用户的，而是一个 Java 类库，是一款程序员可以嵌入他们所开发产品的组件。

- 简介
  - 什么是 Freemark？
- 入门
  - 创建 Configuration 实例
  - 创建数据模型
  - 获取模板
  - 合并模板和数据模型
  - 完整示例
- 基础
  - 数值
  - 类型
  - 模板
- 参考资料

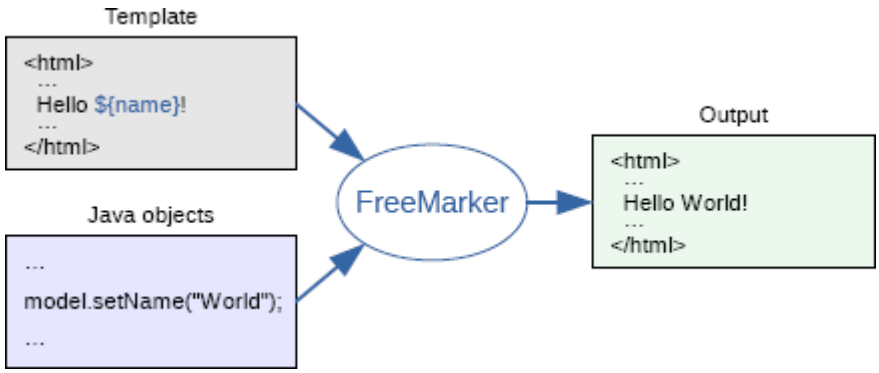
简介

什么是 Freemark？

FreeMarker 是一款 模板引擎： 即一种基于模板和要改变的数据， 并用来生成输出文本(HTML 网页，电子邮件，配置文件，源代码等)的通用工具。 它不是面向最终用户的，而是一个 Java 类库，是一款程序员可以嵌

入他们所开发产品的组件。

简言之：模板 + 数据 = 输出



入门

创建 Configuration 实例

首先，你应该创建一个 `freemarker.template.Configuration` 实例， 然后调整它的设置。Configuration 实例是存储 FreeMarker 应用级设置的核心部分。同时，它也处理创建和缓存预解析模板(比如 Template 对象)的工作。

```
// Create your Configuration instance, and specify if up to what
// version (here 2.3.22) do you want to apply the fixes that are
// backward-compatible. See the Configuration JavaDoc for detail
Configuration cfg = new Configuration(Configuration.VERSION_2_3_

// Specify the source where the template files come from. Here I
// plain directory for it, but non-file-system sources are possi
cfg.setDirectoryForTemplateLoading(new File("/where/you/store/te

// Set the preferred charset template files are stored in. UTF-8
// a good choice in most applications:
cfg.setDefaultEncoding("UTF-8");

// Sets how errors will appear.
// During web page *development* TemplateExceptionHandler.HTML_D
cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW
```

> 注：不需要重复创建 Configuration 实例；它的代价很高，尤其是会丢失缓存。Configuration 实例就是应用级别的单例。

创建数据模型

在简单的示例中你可以使用 `java.lang` 和 `java.util` 包中的类， 还有用户自定义的 Java Bean 来构建数据对象：

- 使用 `java.lang.String` 来构建字符串。
- 使用 `java.lang.Number` 来派生数字类型。
- 使用 `java.lang.Boolean` 来构建布尔值。
- 使用 `java.util.List` 或 Java 数组来构建序列。
- 使用 `java.util.Map` 来构建哈希表。
- 使用自定义的 bean 类来构建哈希表，bean 中的项和 bean 的属性对应。比如， `product` 的 `price` 属性 (`getProperty()`)可以通过 `product.price` 获取。(bean 的 action 也可以通过这种方式拿到；要了解更多可以参看 这里)

示例：

```
// Create the root hash
Map<String, Object> root = new HashMap<>();
// Put string ``user'' into the root
root.put("user", "Big Joe");
// Create the hash for ``latestProduct''
Map<String, Object> latest = new HashMap<>();
// and put it into the root
root.put("latestProduct", latest);
// put ``url'' and ``name'' into latest
latest.put("url", "products/greenmouse.html");
latest.put("name", "green mouse");
```

获取模板

模板代表了 `freemarker.template.Template` 实例。典型的做法是从 `Configuration` 实例中获取一个 `Template` 实例。无论什么时候你需要一个模板实例， 都可以使用它的 `getTemplate` 方法来获取。在 之前 设置的目录中的 `test.ftl` 文件中存储 示例模板，那么就可以这样来做：

```
Template temp = cfg.getTemplate("test.ftl");
```

当调用这个方法的时候，将会创建一个 `test.ftl` 的 `Template` 实例，通过读取 `*/where/you/store/templates/*test.ftl` 文件，之后解析(编译)它。 `Template` 实例以解析后的形式存储模板， 而不是以源文件的文本形式。

`Configuration` 缓存 `Template` 实例，当再次获得 `test.ftl` 的时候，它可能再读取和解析模板文件了， 而只是返回第一次的 `Template` 实例。

合并模板和数据模型

我们已经知道，数据模型+模板=输出，我们有了一个数据模型 (`root`) 和一个模板 (`temp`)， 为了得到输出就需要合并它们。这是由模板的 `process` 方法完成的。它用数据模型 root 和 `Writer` 对象作为参数，然后向 `Writer` 对象写入产生的内容。 为简单起见，这里我们只做标准的输出：

```
Writer out = new OutputStreamWriter(System.out);
temp.process(root, out);
```

这会向你的终端输出你在模板开发指南部分的 第一个示例 中看到的内容。

Java I/O 相关注意事项：基于 `out` 对象，必须保证 `out.close()` 最后被调用。当 `out` 对象被打开并将模板的输出写入文件时，这是很电影的做法。其它时候， 比如典型的 Web 应用程序，那就 *不能* 关闭 `out` 对象。FreeMarker 会在模板执行成功后 (也可以在 `Configuration` 中禁用) 调用 `out.flush()` ，所以不必为此担心。

请注意，一旦获得了 `Template` 实例， 就能将它和不同的数据模型进行不限次数 (`Template` 实例是无状态的)的合并。此外， 当 `Template` 实例创建之后 `test.ftl` 文件才能访问，而不是在调用处理方法时。

完整示例

源码

基础

数值

注意观察每个数据模型的例子你也许能发现：被"(root)"所标识的内容就是哈希表类型的值。 当编写如 `user` 这样的代码时，那就意味着要把"user"变量存储在哈希表的根上。 就像编写 `root.user` 一样，这里但并没有名"root"为的变量， 那么这就起不到任何作用了。

某些人也许会被这种数据模型的例子所困惑，也就是说，根哈希表包含更多的哈希表或序列 (`lotteryNumbers` and `cargo`)。其它就没有更特殊的内容了。 哈希表包含其他变量，那些变量包含其它值，这些数值可以是字符串，数字等变量， 当然也可以是哈希表或序列变量。最初我们解释过的，就像字符串和数字， 序列或哈希表也是一种值的表示形式。

类型

Freemark 支持的类型有：

- 标量：
  - 字符串
  - 数字
  - 布尔值
  - 日期/时间 (日期，时间或日期时间)
- 容器：
  - 哈希表
  - 序列
  - 集合
- 子程序：
  - 方法和函数
  - 用户自定义指令
- 其它/很少使用：
  - 结点

模板

总体结构

模板(FTL 编程)是由如下部分混合而成的：

- **文本**：文本会照着原样来输出。
- **插值**：这部分的输出会被计算的值来替换。插值由 `${` and `}` 所分隔。
- **FTL 标签**：FTL 标签和 HTML 标签很相似，但是它们却是给 FreeMarker 的指示， 而且不会打印在输出内容中。
- **注释**：注释和 HTML 的注释也很相似，但它们是由 `<#--` 和 `-->` 来分隔的。注释会被 FreeMarker 直接忽略， 更不会在输出内容中显示。

我们来看一个具体的模板。其中的内容已经用颜色来标记了：文本，插值，FTL 标签，注释。为了看到可见的换行符，这里使用了 [BR]。

TEMPLATE

```
<html>[BR]
<head>[BR]
  <title>Welcome!</title>[BR]
</head>[BR]
<body>[BR]
  <#-- Greet the user with his/her name -->[BR]
  <h1>Welcome ${user}!</h1>[BR]
  <p>We have these animals:[BR]
  <ul>[BR]
  <#list animals as animal>[BR]
    <li>${animal.name} for ${animal.price} Euros[BR]
  </#list>[BR]
  </ul>[BR]
</body>[BR]
</html>
```

> 注意：
>
> - FTL 是区分大小写的。
> - `插值` 仅仅可以在 `文本` 中使用。
> - `FTL 标签` 不可以在其他 `FTL 标签` 和 `插值` 中使用。
> - `注释` 可以放在 `FTL 标签` 和 `插值` 中。

指令

使用 FTL 标签来调用 指令。

FTL 标签分为两种：

- 开始标签： `<#*directivename* *parameters*>`
- 结束标签： `</#*directivename*>`

除了标签以 `#` 开头外，其他都和 HTML，XML 的语法很相似。 如果标签没有嵌套内容(在开始标签和结束标签之间的内容)，那么可以只使用开始标签。 例如 `<#if *something*>*...*</#if>` ， 而 FreeMarker 知道 `<#include *something*>` 中的 `include` 指令没有可嵌套的内容。

`*parameters*` 的格式由 `*directivename*` 来决定。

事实上，指令有两种类型： 预定义指令 和 用户自定义指令。 对于用户自定义的指令使用 `@` 来代替 `#` 。

> 注意：
>
> - FreeMarker 仅仅关心 FTL 标签的嵌套而不关心 HTML 标签的嵌套。 它只会把 HTML 看做是文本，不会来解释 HTML。
> - 如果你尝试使用一个不存在的指令(比如，输错了指令的名称)，FreeMarker 就会拒绝执行模板，同时抛出错误信息。
> - FreeMarker 会忽略 FTL 标签中多余的 空白标记。

表达式

以下为快速浏览清单，如果需要了解更多细节，请参考这里。

- 直接指定值
  - 字符串： `"Foo"` 或者 `'Foo'` 或者 `"It's \"quoted\""` 或者 `'It\'s "quoted"'` 或者 `r"C:\raw\string"`
  - 数字： `123.45`
  - 布尔值： `true` ， `false`
  - 序列： `["foo", "bar", 123.45]` ； 值域： `0..9` ， `0..<10` (或 `0..!10`)， `0..`
  - 哈希表： `{"name":"green mouse", "price":150}`
- 检索变量
  - 顶层变量： `user`
  - 从哈希表中检索数据： `user.name` ， `user["name"]`

    - 从序列中检索数据： `products[5]`
    - 特殊变量： `.main`
- 字符串操作
    - 插值(或连接)： `"Hello ${user}!"` (或 `"Hello " + user + "!"`)
    - 获取一个字符： `name[0]`
    - 字符串切分： 包含结尾： `name[0..4]` ，不包含结尾： `name[0..<5]` ，基于长度(宽容处理)： `name[0..*5]` ，去除开头： `name[5..]`
- 序列操作
    - 连接： `users + ["guest"]`
    - 序列切分：包含结尾： `products[20..29]` ， 不包含结尾： `products[20..<30]` ，基于长度(宽容处理)： `products[20..*10]` ，去除开头： `products[20..]`
- 哈希表操作
    - 连接： `passwords + { "joe": "secret42" }`
- 算术运算： `(x * 1.5 + 10) / 2 - y % 100`
- 比较运算： `x == y` ， `x != y` ， `x < y` ， `x > y` ， `x >= y` ， `x <= y` ， `x lt y` ， `x lte y` ， `x gt y` ， `x gte y` ， 等等。。。。。。
- 逻辑操作： `!registered && (firstVisit || fromEurope)`
- 内建函数： `name?upper_case` ， `path?ensure_starts_with('/')`
- 方法调用： `repeat("What", 3)`
- 处理不存在的值
    - 默认值： `name!"unknown"` 或者 `(user.name)!"unknown"` 或者 `name!` 或者 `(user.name)!`
    - 检测不存在的值： `name??` 或者 `(user.name)??`
- 赋值操作： `=` ， `+=` ， `-=` ， `*=` ， `/=` ， `%=` ， `++` ， `--`

插值

插值的使用格式是： `${*expression*}` ，这里的 `*expression*` 可以是所有种类的表达式(比如 `${100 + x}`)。

插值是用来给 `*表达式*` 插入具体值然后转换为文本(字符串)。插值仅仅可以在两种位置使用：在 文本 区 (比如 `<h1>Hello ${name}!</h1>`) 和 字符串表达式 (比如 `<#include "/footer/${company}.html">`)中。

表达式的结果必须是字符串，数字或者日期/时间/日期-时间值， 因为(默认是这样)仅仅这些值可以被插值自动转换为字符串。其它类型的值 (比如布尔值，序列)必须 "手动地" 转换成字符串(后续会有一些建议)， 否则就会发生错误，中止模板执行。

注意：插值 *仅仅* 在 文本区 (比如 `<h1>Hello ${name}!</h1>`) 和 字符串中起作用。

- ⭕ `<#include "/footer/${company}.html">`
- ⭕ `<#if big>...</#if>`
- ❌ `<#if ${big}>...</#if>`
- ❌ `<#if "${big}">...</#if>`

参考资料

- http://freemarker.foofun.cn/

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-10-21 18:06:24

- Java 与 JSON
  - 1. JSON 简介
    - JSON 是什么
    - 优缺点、标准与 schema
    - 相关技术以及与 XML 的关系
    - 工具
    - 1.2. Java JSON 库
  - JSON 编码指南
    - 4.1 Google JSON 风格指南
    - 4.2 使用 JSON 实现 API
  - 2. Fastjson 应用
    - 2.1. 添加 maven 依赖
    - 2.2. Fastjson API
    - 2.3. Fastjson 注解
  - 3. Jackson 应用
    - 3.1. 添加 maven 依赖
    - 3.2. Jackson API
    - 3.3. Jackson 注解
  - 4. Gson 应用
    - 4.1. 添加 maven 依赖
    - 4.2. Gson API
    - 4.3. Gson 注解
  - 5. 示例源码
  - 6. 参考资料

Java 与 JSON

> JSON（JavaScript Object Notation）是一种基于文本的数据交换格式。几乎所有的编程语言都有很好的库或第三方工具来提供基于 JSON 的 API 支持，因此你可以非常方便地使用任何自己喜欢的编程语言来处理 JSON 数据。
>
> 本文主要从 Java 语言的角度来讲解 JSON 的应用。

- 1. JSON 简介
  - JSON 是什么
  - 优缺点、标准与 schema
  - 相关技术以及与 XML 的关系
  - 工具
  - 1.2. Java JSON 库
- JSON 编码指南
  - 4.1 Google JSON 风格指南
  - 4.2 使用 JSON 实现 API
- 2. Fastjson 应用

    - 2.1. 添加 maven 依赖
    - 2.2. Fastjson API
    - 2.3. Fastjson 注解
- 3. Jackson 应用
    - 3.1. 添加 maven 依赖
    - 3.2. Jackson API
    - 3.3. Jackson 注解
- 4. Gson 应用
    - 4.1. 添加 maven 依赖
    - 4.2. Gson API
    - 4.3. Gson 注解
- 5. 示例源码
- 6. 参考资料

1. JSON 简介

JSON 是什么

JSON 起源于 1999 年的 JS 语言规范 ECMA262 的一个子集（即 15.12 章节描述了格式与解析），后来 2003 年作为一个数据格式 ECMA404（很囧的序号有不有？）发布。 2006 年，作为 rfc4627 发布，这时规范增加到 18 页，去掉没用的部分，十页不到。

JSON 的应用很广泛，这里有超过 100 种语言下的 JSON 库：json.org。

更多的可以参考这里，关于 json 的一切。

优缺点、标准与 schema

结构与类型

这估计是最简单标准规范之一：

- 只有两种结构：对象内的键值对集合结构和数组，对象用 `{}` 表示、内部是"key":"value"，数组用 `[]` 表示，不同值用逗号分开
- 基本数值有 7 个： false / null / true / object / array / number / string
- 再加上结构可以嵌套，进而可以用来表达复杂的数据
- 一个简单实例：

```
{
    "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url": "http://www.example.com/image/481989943",
            "Height": 125,
            "Width": "100"
        },
        "IDs": [116, 943, 234, 38793]
    }
}
```

> 扩展阅读：
>
> - http://www.json.org/json-zh.html - 图文并茂介绍 json 数据形式
> - json 的 RFC 文档

2.2 优点

- 基于纯文本，所以对于人类阅读是很友好的。
- 规范简单，所以容易处理，开箱即用，特别是 JS 类的 ECMA 脚本里是内建支持的，可以直接作为对象使用。
- 平台无关性，因为类型和结构都是平台无关的，而且好处理，容易实现不同语言的处理类库，可以作为多个不同异构系统之间的数据传输格式协议，特别是在 HTTP/REST 下的数据格式。

2.3 缺点

缺点也很明显：

- 性能一般，文本表示的数据一般来说比二进制大得多，在数据传输上和解析处理上都要更影响性能。
- 缺乏 schema，跟同是文本数据格式的 XML 比，在类型的严格性和丰富性上要差很多。XML 可以借由 XSD 或 DTD 来定义复杂的格式，并由此来验证 XML 文档是否符合格式要求，甚至进一步的，可以基于 XSD 来生成具体语言的操作代码，例如 apache xmlbeans。并且这些工具组合到一起，形成一套庞大的生态，例如基于 XML 可以实现 SOAP 和 WSDL，一系列的 ws-*规范。但是我们也可以看到 JSON 在缺乏规范的情况下，实际上有更大一些的灵活性，特别是近年来 REST 的快速发展，已经有一些 schema 相关的发展(例如理解 JSON Schema，使用 JSON Schema， 在线 schema 测试)，也有类似于 WSDL 的WADL出现。

相关技术以及与 XML 的关系

- 使用 JSON 实现 RPC（类似 XML-RPC）：JSON-RPC
- 使用 JSON 实现 path 查询操作（类似 XML-PATH）：JsonPATH
- 在线查询工具：JsonPATH

工具

- 格式化工具：jsbeautifier
- chrome 插件：5 个 Json View 插件
- 在线 Mock: 在线 mock
- 其他 Mock：SoapUI可以支持，SwaggerUI 也可以，RestMock也可以。

1.2. Java JSON 库

Java 中比较流行的 JSON 库有：

- Fastjson - 阿里巴巴开发的 JSON 库，性能十分优秀。
- Jackson - 社区十分活跃且更新速度很快。Spring 框架默认 JSON 库。
- Gson - 谷歌开发的 JSON 库，目前功能最全的 JSON 库 。

从性能上来看，一般情况下：Fastjson > Jackson > Gson

JSON 编码指南

4.1 Google JSON 风格指南

遵循好的设计与编码风格，能提前解决 80%的问题:

- 英文版Google JSON Style Guide：https://google.github.io/styleguide/jsoncstyleguide.xml
- 中文版Google JSON 风格指南：https://github.com/darcyliu/google-styleguide/blob/master/JSONStyleGuide.md

简单摘录如下：

- 属性名和值都是用双引号，不要把注释写到对象里面，对象数据要简洁
- 不要随意结构化分组对象，推荐是用扁平化方式，层次不要太复杂
- 命名方式要有意义，比如单复数表示
- 驼峰式命名，遵循 Bean 规范
- 使用版本来控制变更冲突
- 对于一些关键字，不要拿来做 key

- 如果一个属性是可选的或者包含空值或 null 值，考虑从 JSON 中去掉该属性，除非它的存在有很强的语义原因
- 序列化枚举类型时，使用 name 而不是 value
- 日期要用标准格式处理
- 设计好通用的分页参数
- 设计好异常处理

4.2 使用 JSON 实现 API

JSON API与 Google JSON 风格指南有很多可以相互参照之处。

JSON API是数据交互规范，用以定义客户端如何获取与修改资源，以及服务器如何响应对应请求。

JSON API 设计用来最小化请求的数量，以及客户端与服务器间传输的数据量。在高效实现的同时，无需牺牲可读性、灵活性和可发现性。

2. Fastjson 应用

> 扩展阅读：更多 API 使用细节可以参考：JSONField 用法

2.1. 添加 maven 依赖

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>x.x.x</version>
</dependency>
```

2.2. Fastjson API

定义 Bean

Group.java

```java
public class Group {

    private Long       id;
    private String     name;
    private List<User> users = new ArrayList<User>();

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<User> getUsers() {
        return users;
    }

    public void setUsers(List<User> users) {
        this.users = users;
    }

    public void addUser(User user) {
        users.add(user);
    }
}
```

User.java

```java
public class User {

    private Long   id;
    private String name;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

初始化 **Bean**

```java
Group group = new Group();
group.setId(0L);
group.setName("admin");

User guestUser = new User();
guestUser.setId(2L);
guestUser.setName("guest");

User rootUser = new User();
rootUser.setId(3L);
rootUser.setName("root");

group.addUser(guestUser);
group.addUser(rootUser);
```

序列化

```java
String jsonString = JSON.toJSONString(group);
System.out.println(jsonString);
```

反序列化

```
Group bean = JSON.parseObject(jsonString, Group.class);
```

2.3. Fastjson 注解

`@JSONField`

> 扩展阅读：更多 API 使用细节可以参考：JSONField 用法，这里介绍基本用法。

可以配置在属性（setter、getter）和字段（必须是 public field）上。

```
@JSONField(name="ID")
public int getId() {return id;}

// 配置date序列化和反序列使用yyyyMMdd日期格式
@JSONField(format="yyyyMMdd")
public Date date1;

// 不序列化
@JSONField(serialize=false)
public Date date2;

// 不反序列化
@JSONField(deserialize=false)
public Date date3;

// 按ordinal排序
@JSONField(ordinal = 2)
private int f1;

@JSONField(ordinal = 1)
private int f2;
```

`@JSONType`

- 自定义序列化：ObjectSerializer
- 子类型处理：SeeAlso

JSONType.alphabetic 属性: fastjson 缺省时会使用字母序序列化，如果你是希望按照 java fields/getters 的自然顺序序列化，可以配置 JSONType.alphabetic，使用方法如下：

```
@JSONType(alphabetic = false)
public static class B {
    public int f2;
    public int f1;
    public int f0;
}
```

3. Jackson 应用

> 扩展阅读：更多 API 使用细节可以参考 jackson-databind 官方说明

3.1. 添加 maven 依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.8</version>
</dependency>
```

3.2. Jackson API

序列化

```
ObjectMapper mapper = new ObjectMapper();

mapper.writeValue(new File("result.json"), myResultObject);
// or:
byte[] jsonBytes = mapper.writeValueAsBytes(myResultObject);
// or:
String jsonString = mapper.writeValueAsString(myResultObject);
```

反序列化

```
ObjectMapper mapper = new ObjectMapper();

MyValue value = mapper.readValue(new File("data.json"), MyValue.
// or:
value = mapper.readValue(new URL("http://some.com/api/entry.json
// or:
value = mapper.readValue("{\"name\":\"Bob\", \"age\":13}", MyVal
```

容器的序列化和反序列化

```
Person p = new Person("Tom", 20);
Person p2 = new Person("Jack", 22);
Person p3 = new Person("Mary", 18);

List<Person> persons = new LinkedList<>();
persons.add(p);
persons.add(p2);
persons.add(p3);

Map<String, List> map = new HashMap<>();
map.put("persons", persons);

String json = null;
try {
    json = mapper.writeValueAsString(map);
} catch (JsonProcessingException e) {
    e.printStackTrace();
}
```

3.3. Jackson 注解

> 扩展阅读：更多注解使用细节可以参考 jackson-annotations 官方说明

`@JsonProperty`

```
public class MyBean {
   private String _name;

   // without annotation, we'd get "theName", but we want "name"
   @JsonProperty("name")
   public String getTheName() { return _name; }

   // note: it is enough to add annotation on just getter OR set
   // so we can omit it here
   public void setTheName(String n) { _name = n; }
}
```

`@JsonIgnoreProperties` 和 `@JsonIgnore`

```
// means that if we see "foo" or "bar" in JSON, they will be qui
// regardless of whether POJO has such properties
@JsonIgnoreProperties({ "foo", "bar" })
public class MyBean {
   // will not be written as JSON; nor assigned from JSON:
   @JsonIgnore
   public String internal;

   // no annotation, public field is read/written normally
   public String external;

   @JsonIgnore
   public void setCode(int c) { _code = c; }

   // note: will also be ignored because setter has annotation!
   public int getCode() { return _code; }
}
```

`@JsonCreator`

```
public class CtorBean {
  public final String name;
  public final int age;


  @JsonCreator // constructor can be public, private, whatever
  private CtorBean(@JsonProperty("name") String name,
    @JsonProperty("age") int age)
  {
      this.name = name;
      this.age = age;
  }
}
```

`@JsonPropertyOrder`

alphabetic 设为 true 表示，json 字段按自然顺序排列，默认为 false。

```
@JsonPropertyOrder(alphabetic = true)
public class JacksonAnnotationBean {}
```

4. Gson 应用

> 详细内容可以参考官方文档：Gson 用户指南

4.1. 添加 maven 依赖

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
</dependency>
```

4.2. Gson API

序列化

```
Gson gson = new Gson();
gson.toJson(1);             // ==> 1
gson.toJson("abcd");        // ==> "abcd"
gson.toJson(10L); // ==> 10
int[] values = { 1 };
gson.toJson(values);        // ==> [1]
```

反序列化

```
int i1 = gson.fromJson("1", int.class);
Integer i2 = gson.fromJson("1", Integer.class);
Long l1 = gson.fromJson("1", Long.class);
Boolean b1 = gson.fromJson("false", Boolean.class);
String str = gson.fromJson("\"abc\"", String.class);
String[] anotherStr = gson.fromJson("[\"abc\"]", String[].class)
```

GsonBuilder

`Gson` 实例可以通过 `GsonBuilder` 来定制实例化，以控制其序列化、反序列化行为。

```
Gson gson = new GsonBuilder()
        .setPrettyPrinting()
        .setDateFormat("yyyy-MM-dd HH:mm:ss")
        .excludeFieldsWithModifiers(Modifier.STATIC, Modifier.TR
        .create();
```

4.3. Gson 注解

`@Since`

`@Since` 用于控制对象的序列化版本。示例：

```
public class VersionedClass {
  @Since(1.1) private final String newerField;
  @Since(1.0) private final String newField;
  private final String field;

  public VersionedClass() {
    this.newerField = "newer";
    this.newField = "new";
    this.field = "old";
  }
}

VersionedClass versionedObject = new VersionedClass();
Gson gson = new GsonBuilder().setVersion(1.0).create();
String jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
System.out.println();

gson = new Gson();
jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
```

`@SerializedName`

`@SerializedName` 用于将类成员按照指定名称序列化、反序列化。示例：

```
private class SomeObject {
  @SerializedName("custom_naming") private final String someFiel
  private final String someOtherField;

  public SomeObject(String a, String b) {
    this.someField = a;
    this.someOtherField = b;
  }
}
```

5. 示例源码

> 示例源码：javalib-io-json

6. 参考资料

- 官方
  - Fastjson Github
  - Gson Github
  - jackson 官方文档
  - jackson-databind
- 文章
  - http://www.json.org/json-zh.html
  - json 的 RFC 文档
  - JSON 最佳实践
  - 【简明教程】JSON

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-24 21:08:11

- 细说 Java 主流日志工具库
  - 日志框架
    - java.util.logging (JUL)
    - Log4j
    - Logback
    - Log4j2
    - Log4j vs Logback vs Log4j2
  - 日志门面
    - common-logging
    - slf4j
    - common-logging vs slf4j
    - 总结
  - 实施日志解决方案
    - 引入 jar 包
    - 使用 API
  - log4j2 配置
  - logback 配置
    -
    -
    -
    -
    - 完整的 logback.xml 参考示例
  - log4j 配置
    - 完整的 log4j.xml 参考示例
  - 参考

细说 Java 主流日志工具库

> 在项目开发中，为了跟踪代码的运行情况，常常要使用日志来记录信息。
>
> 在 Java 世界，有很多的日志工具库来实现日志功能，避免了我们重复造轮子。
>
> 我们先来逐一了解一下主流日志工具。
>
> 📓 本文已归档到：「blog」

- 日志框架
  - java.util.logging (JUL)
  - Log4j
  - Logback
  - Log4j2
  - Log4j vs Logback vs Log4j2
- 日志门面

    - common-logging
    - slf4j
    - common-logging vs slf4j
    - 总结
- 实施日志解决方案
    - 引入 jar 包
    - 使用 API
- log4j2 配置
- logback 配置
    - `<configuration>`
    - `<appender>`
    - `<logger>`
    - `<root>`
    - 完整的 logback.xml 参考示例
- log4j 配置
    - 完整的 log4j.xml 参考示例
- 参考

日志框架

java.util.logging (JUL)

JDK1.4 开始，通过 `java.util.logging` 提供日志功能。

它能满足基本的日志需要，但是功能没有 Log4j 强大，而且使用范围也没有 Log4j 广泛。

Log4j

Log4j 是 apache 的一个开源项目，创始人 Ceki Gulcu。

Log4j 应该说是 Java 领域资格最老，应用最广的日志工具。从诞生之日到现在一直广受业界欢迎。

Log4j 是高度可配置的，并可通过在运行时的外部文件配置。它根据记录的优先级别，并提供机制，以指示记录信息到许多的目的地，诸如：数据库，文件，控制台，UNIX 系统日志等。

Log4j 中有三个主要组成部分：

- **loggers** - 负责捕获记录信息。
- **appenders** - 负责发布日志信息，以不同的首选目的地。
- **layouts** - 负责格式化不同风格的日志信息。

官网地址

Logback

Logback 是由 log4j 创始人 Ceki Gulcu 设计的又一个开源日记组件，目标是替代 log4j。

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 或 JDK14 Logging。
- `logback-access` - 访问模块与 Servlet 容器集成提供通过 Http 来访问日记的功能。

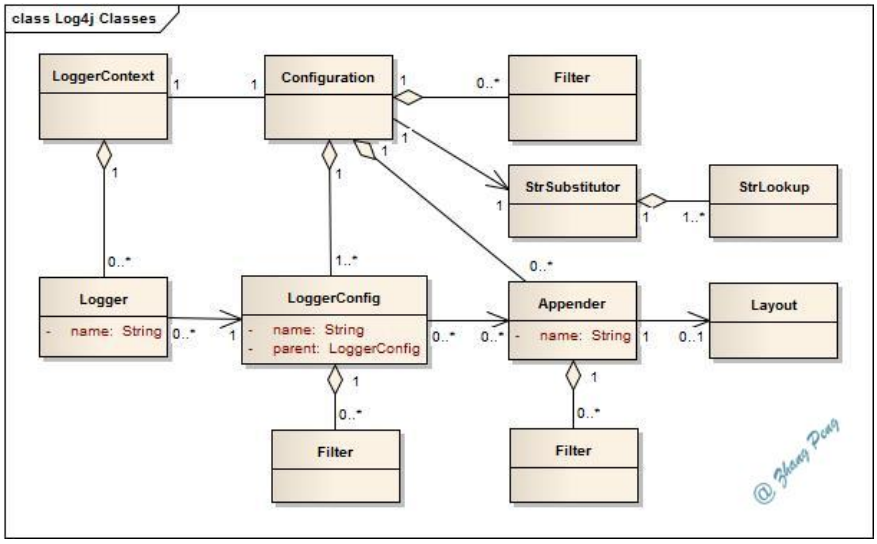
官网地址

Log4j2

官网地址

按照官方的说法，Log4j2 是 Log4j 和 Logback 的替代。

Log4j2 架构：



Log4j vs Logback vs Log4j2

按照官方的说法，**Log4j2 大大优于 Log4j 和 Logback**。

那么，Log4j2 相比于先问世的 Log4j 和 Logback，它具有哪些优势呢？

1. Log4j2 旨在用作审计日志记录框架。 Log4j 1.x 和 Logback 都会在重新配置时丢失事件。 Log4j 2 不会。在 Logback 中，Appender 中的

异常永远不会对应用程序可见。在 Log4j 中，可以将 Appender 配置为允许异常渗透到应用程序。
2. Log4j2 在多线程场景中，异步 Loggers 的吞吐量比 Log4j 1.x 和 Logback 高 10 倍，延迟低几个数量级。
3. Log4j2 对于独立应用程序是无垃圾的，对于稳定状态日志记录期间的 Web 应用程序来说是低垃圾。这减少了垃圾收集器的压力，并且可以提供更好的响应时间性能。
4. Log4j2 使用插件系统，通过添加新的 Appender、Filter、Layout、Lookup 和 Pattern Converter，可以非常轻松地扩展框架，而无需对 Log4j 进行任何更改。
5. 由于插件系统配置更简单。配置中的条目不需要指定类名。
6. 支持自定义日志等级。
7. 支持 lambda 表达式。
8. 支持消息对象。
9. Log4j 和 Logback 的 Layout 返回的是字符串，而 Log4j2 返回的是二进制数组，这使得它能被各种 Appender 使用。
10. Syslog Appender 支持 TCP 和 UDP 并且支持 BSD 系统日志。
11. Log4j2 利用 Java5 并发特性，尽量小粒度的使用锁，减少锁的开销。

日志门面

何谓日志门面？

日志门面是对不同日志框架提供的一个 API 封装，可以在部署的时候不修改任何配置即可接入一种日志实现方案。

common-logging

common-logging 是 apache 的一个开源项目。也称**Jakarta Commons Logging**，缩写 **JCL**。

common-logging 的功能是提供日志功能的 API 接口，本身并不提供日志的具体实现（当然，common-logging 内部有一个 Simple logger 的简单实现，但是功能很弱，直接忽略），而是在运行时动态的绑定日志实现组件来工作（如 log4j、java.util.loggin）。

官网地址

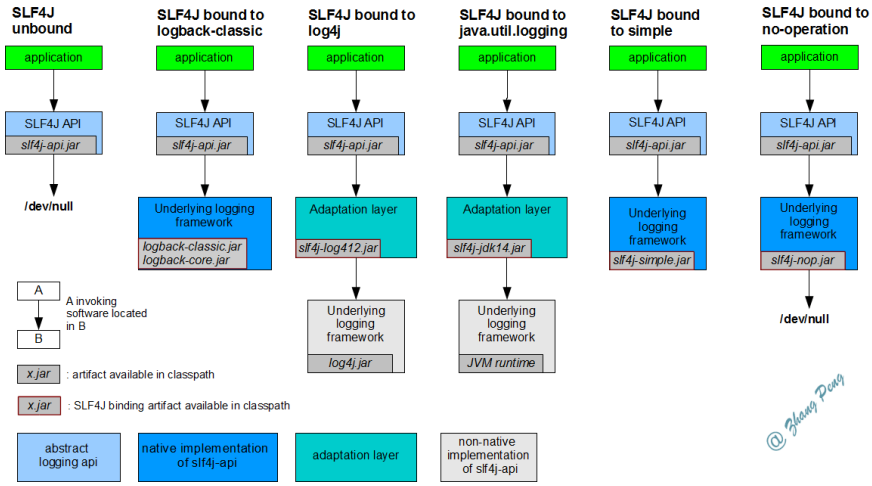
slf4j

全称为 Simple Logging Facade for Java，即 java 简单日志门面。

什么，作者又是 Ceki Gulcu！这位大神写了 Log4j、Logback 和 slf4j，专注日志组件开发五百年，一直只能超越自己。

类似于 Common-Logging，slf4j 是对不同日志框架提供的一个 API 封装，可以在部署的时候不修改任何配置即可接入一种日志实现方案。但是，slf4j 在**编译时**静态绑定真正的 Log 库。使用 SLF4J 时，如果你需要使用某一种日志实现，那么你必须选择正确的 SLF4J 的 jar 包的集合（各种桥接包）。

官网地址



common-logging vs slf4j

slf4j 库类似于 Apache Common-Logging。但是，他在编译时静态绑定真正的日志库。这点似乎很麻烦，其实也不过是导入桥接 jar 包而已。

slf4j 一大亮点是提供了更方便的日志记录方式：

不需要使用 `logger.isDebugEnabled()` 来解决日志因为字符拼接产生的性能问题。slf4j 的方式是使用 `{}` 作为字符串替换符，形式如下：

```
logger.debug("id: {}, name: {} ", id, name);
```

总结

综上所述，使用 slf4j + Logback 可谓是目前最理想的日志解决方案了。

接下来，就是如何在项目中实施了。

实施日志解决方案

使用日志解决方案基本可分为三步：

1. 引入 jar 包
2. 配置
3. 使用 API

常见的各种日志解决方案的第 2 步和第 3 步基本一样，实施上的差别主要在第 1 步，也就是使用不同的库。

引入 jar 包

这里首选推荐使用 slf4j + logback 的组合。

如果你习惯了 common-logging，可以选择 common-logging+log4j。

强烈建议不要直接使用日志实现组件(logback、log4j、java.util.logging)，理由前面也说过，就是无法灵活替换日志库。

还有一种情况：你的老项目使用了 common-logging，或是直接使用日志实现组件。如果修改老的代码，工作量太大，需要兼容处理。在下文，都将看到各种应对方法。

注：据我所知，当前仍没有方法可以将 slf4j 桥接到 common-logging。如果我孤陋寡闻了，请不吝赐教。

slf4j 直接绑定日志组件

slf4j + logback

添加依赖到 pom.xml 中即可。

logback-classic-1.0.13.jar 会自动将 *slf4j-api-1.7.21.jar* 和 *logback-core-1.0.13.jar* 也添加到你的项目中。

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.13</version>
</dependency>
```

slf4j + log4j

添加依赖到 pom.xml 中即可。

slf4j-log4j12-1.7.21.jar 会自动将 *slf4j-api-1.7.21.jar* 和 *log4j-1.2.17.jar* 也添加到你的项目中。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.21</version>
</dependency>
```

slf4j + java.util.logging

添加依赖到 pom.xml 中即可。

slf4j-jdk14-1.7.21.jar 会自动将 *slf4j-api-1.7.21.jar* 也添加到你的项目中。

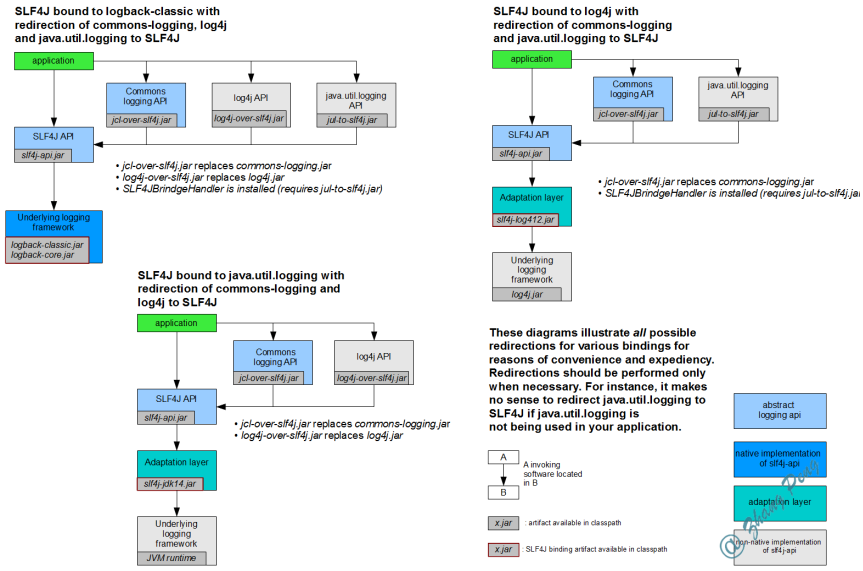
```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>1.7.21</version>
</dependency>
```

slf4j 兼容非 slf4j 日志组件

在介绍解决方案前，先提一个概念——桥接

什么是桥接呢

假如你正在开发应用程序所调用的组件当中已经使用了 common-logging，这时你需要 jcl-over-slf4j.jar 把日志信息输出重定向到 slf4j-api，slf4j-api 再去调用 slf4j 实际依赖的日志组件。这个过程称为桥接。下图是官方的 slf4j 桥接策略图：



从图中应该可以看出，无论你的老项目中使用的是 common-logging 或是直接使用 log4j、java.util.logging，都可以使用对应的桥接 jar 包来解决兼容问题。

slf4j 兼容 common-logging

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>1.7.12</version>
</dependency>
```

slf4j 兼容 log4j

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
    <version>1.7.12</version>
</dependency>
```

slf4j 兼容 java.util.logging

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <version>1.7.12</version>
</dependency>
```

spring 集成 slf4j

做 java web 开发，基本离不开 spring 框架。很遗憾，spring 使用的日志解决方案是 common-logging + log4j。

所以，你需要一个桥接 jar 包：*logback-ext-spring*。

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.1.3</version>
</dependency>
<dependency>
  <groupId>org.logback-extensions</groupId>
  <artifactId>logback-ext-spring</artifactId>
  <version>0.1.2</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>1.7.12</version>
</dependency>
```

common-logging 绑定日志组件

common-logging + log4j

添加依赖到 pom.xml 中即可。

```
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

使用 API

slf4j 用法

使用 slf4j 的 API 很简单。使用 `LoggerFactory` 初始化一个 `Logger` 实例，然后调用 Logger 对应的打印等级函数就行了。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class App {
    private static final Logger log = LoggerFactory.getLogger(Ap
    public static void main(String[] args) {
        String msg = "print log, current level: {}";
        log.trace(msg, "trace");
        log.debug(msg, "debug");
        log.info(msg, "info");
        log.warn(msg, "warn");
        log.error(msg, "error");
    }
}
```

common-logging 用法

common-logging 用法和 slf4j 几乎一样，但是支持的打印等级多了一个更高级别的：**fatal**。

此外，common-logging 不支持 `{}` 替换参数，你只能选择拼接字符串这种方式了。

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class JclTest {
    private static final Log log = LogFactory.getLog(JclTest.cla

    public static void main(String[] args) {
        String msg = "print log, current level: ";
        log.trace(msg + "trace");
        log.debug(msg + "debug");
        log.info(msg + "info");
        log.warn(msg + "warn");
        log.error(msg + "error");
        log.fatal(msg + "fatal");
    }
}
```

log4j2 配置

log4j2 基本配置形式如下：

```xml
<?xml version="1.0" encoding="UTF-8"?>;
<Configuration>
  <Properties>
    <Property name="name1">value</property>
    <Property name="name2" value="value2"/>
  </Properties>
  <Filter type="type" ... />
  <Appenders>
    <Appender type="type" name="name">
      <Filter type="type" ... />
    </Appender>
    ...
  </Appenders>
  <Loggers>
    <Logger name="name1">
      <Filter type="type" ... />
    </Logger>
    ...
    <Root level="level">
      <AppenderRef ref="name"/>
    </Root>
```

```
  </Loggers>
</Configuration>
```

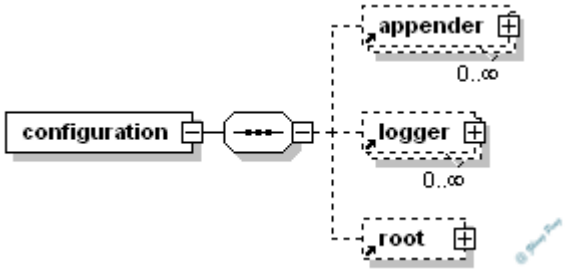
配置示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" strict="true" name="XMLConfigTest"
               packages="org.apache.logging.log4j.test">
  <Properties>
    <Property name="filename">target/test.log</Property>
  </Properties>
  <Filter type="ThresholdFilter" level="trace"/>

  <Appenders>
    <Appender type="Console" name="STDOUT">
      <Layout type="PatternLayout" pattern="%m MDC%X%n"/>
      <Filters>
        <Filter type="MarkerFilter" marker="FLOW" onMatch="DENY"
        <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="
      </Filters>
    </Appender>
    <Appender type="Console" name="FLOW">
      <Layout type="PatternLayout" pattern="%C{1}.%M %m %ex%n"/>
      <Filters>
        <Filter type="MarkerFilter" marker="FLOW" onMatch="ACCEP
        <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="
      </Filters>
    </Appender>
    <Appender type="File" name="File" fileName="${filename}">
      <Layout type="PatternLayout">
        <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
      </Layout>
    </Appender>
  </Appenders>

  <Loggers>
    <Logger name="org.apache.logging.log4j.test1" level="debug"
      <Filter type="ThreadContextMapFilter">
        <KeyValuePair key="test" value="123"/>
      </Filter>
      <AppenderRef ref="STDOUT"/>
    </Logger>

    <Logger name="org.apache.logging.log4j.test2" level="debug"
      <AppenderRef ref="File"/>
    </Logger>

    <Root level="trace">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>

</Configuration>
```

logback 配置

`<configuration>`

- 作用：`<configuration>` 是 logback 配置文件的根元素。
- 要点
  - 它有 `<appender>` 、 `<logger>` 、 `<root>` 三个子元素。



`<appender>`

- 作用：将记录日志的任务委托给名为 appender 的组件。
- 要点
  - 可以配置零个或多个。
  - 它有 `<file>` 、 `<filter>` 、 `<layout>` 、 `<encoder>` 四个子元素。
- 属性
  - name：设置 appender 名称。
  - class：设置具体的实例化类。

`<file>`

- 作用：设置日志文件路径。

`<filter>`

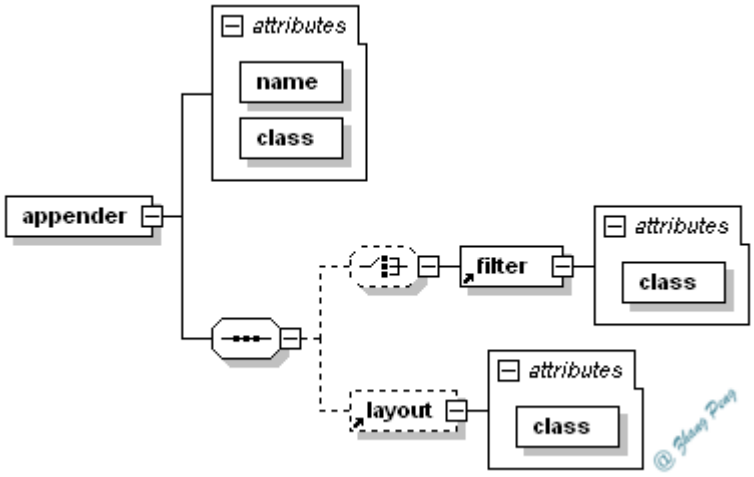
- 作用：设置过滤器。
- 要点
  - 可以配置零个或多个。

`<layout>`

- 作用：设置 appender。
- 要点
  - 可以配置零个或一个。
- 属性
  - class：设置具体的实例化类。

`<encoder>`

- 作用：设置编码。
- 要点
  - 可以配置零个或多个。
- 属性
  - class：设置具体的实例化类。



`<logger>`

- 作用：设置 logger。
- 要点
  - 可以配置零个或多个。
- 属性
  - name
  - level：设置日志级别。不区分大小写。可选值：TRACE、DEBUG、INFO、WARN、ERROR、ALL、OFF。
  - additivity：可选值：true 或 false。

`<appender-ref>`

- 作用：appender 引用。
- 要点
  - 可以配置零个或多个。

`<root>`

- 作用：设置根 logger。
- 要点
  - 只能配置一个。
  - 除了 level，不支持任何属性。level 属性和 `<logger>` 中的相同。
  - 有一个子元素 `<appender-ref>`，与 `<logger>` 中的相同。

完整的 logback.xml 参考示例

在下面的配置文件中，我为自己的项目代码（根目录：org.zp.notes.spring）设置了五种等级：

TRACE、DEBUG、INFO、WARN、ERROR，优先级依次从低到高。

因为关注 spring 框架本身的一些信息，我增加了专门打印 spring WARN 及以上等级的日志。

```
<?xml version="1.0" encoding="UTF-8" ?>

<!-- logback中一共有5种有效级别，分别是TRACE、DEBUG、INFO、WARN、ER
<configuration scan="true" scanPeriod="60 seconds" debug="false"

  <property name="DIR_NAME" value="spring-helloworld"/>

  <!-- 将记录日志打印到控制台 -->
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppe
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <!-- RollingFileAppender begin -->
  <appender name="ALL" class="ch.qos.logback.core.rolling.Rollin
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/all.%d{yyyy-
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>30MB</maxFileSize>
    </triggeringPolicy>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <appender name="ERROR" class="ch.qos.logback.core.rolling.Roll
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/error.%d{yyy
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>ERROR</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
```

```
    </filter>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <appender name="WARN" class="ch.qos.logback.core.rolling.Rolli
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/warn.%d{yyyy
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>WARN</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
    </filter>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <appender name="INFO" class="ch.qos.logback.core.rolling.Rolli
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/info.%d{yyyy
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>INFO</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
    </filter>
```

```
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <appender name="DEBUG" class="ch.qos.logback.core.rolling.Roll
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/debug.%d{yyy
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>DEBUG</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
    </filter>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>

  <appender name="TRACE" class="ch.qos.logback.core.rolling.Roll
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/trace.%d{yyy
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <filter class="ch.qos.logback.classic.filter.LevelFilter">
      <level>TRACE</level>
      <onMatch>ACCEPT</onMatch>
      <onMismatch>DENY</onMismatch>
    </filter>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
```

```
    </encoder>
  </appender>

  <appender name="SPRING" class="ch.qos.logback.core.rolling.Rol
    <!-- 根据时间来制定滚动策略 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedR
      <fileNamePattern>${user.dir}/logs/${DIR_NAME}/springframew
      </fileNamePattern>
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <!-- 根据文件大小来制定滚动策略 -->
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBas
      <maxFileSize>10MB</maxFileSize>
    </triggeringPolicy>

    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%
    </encoder>
  </appender>
  <!-- RollingFileAppender end -->

  <!-- logger begin -->
  <!-- 本项目的日志记录，分级打印 -->
  <logger name="org.zp.notes.spring" level="TRACE" additivity="f
    <appender-ref ref="STDOUT"/>
    <appender-ref ref="ERROR"/>
    <appender-ref ref="WARN"/>
    <appender-ref ref="INFO"/>
    <appender-ref ref="DEBUG"/>
    <appender-ref ref="TRACE"/>
  </logger>

  <!-- SPRING框架日志 -->
  <logger name="org.springframework" level="WARN" additivity="fa
    <appender-ref ref="SPRING"/>
  </logger>

  <root level="TRACE">
    <appender-ref ref="ALL"/>
  </root>
  <!-- logger end -->

</configuration>
```

log4j 配置

完整的 log4j.xml 参考示例

log4j 的配置文件一般有 xml 格式或 properties 格式。这里为了和 logback.xml 做个对比，就不介绍 properties 了，其实也没太大差别。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4

  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppende
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
             value="%d{yyyy-MM-dd HH:mm:ss,SSS\} [%-5p] [%t] %c{
    </layout>

    <!--过滤器设置输出的级别-->
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
      <param name="levelMin" value="debug"/>
      <param name="levelMax" value="fatal"/>
      <param name="AcceptOnMatch" value="true"/>
    </filter>
  </appender>


  <appender name="ALL" class="org.apache.log4j.DailyRollingFileA
    <param name="File" value="${user.dir}/logs/spring-common/jcl
    <param name="Append" value="true"/>
    <!-- 每天重新生成日志文件 -->
    <param name="DatePattern" value="'-'yyyy-MM-dd'.log'"/>
    <!-- 每小时重新生成日志文件 -->
    <!--<param name="DatePattern" value="'-'yyyy-MM-dd-HH'.log'"
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
             value="%d{yyyy-MM-dd HH:mm:ss,SSS\} [%-5p] [%t] %c{
    </layout>
  </appender>

  <!-- 指定logger的设置，additivity指示是否遵循缺省的继承机制-->
  <logger name="org.zp.notes.spring" additivity="false">
    <level value="error"/>
    <appender-ref ref="STDOUT"/>
    <appender-ref ref="ALL"/>
  </logger>

  <!-- 根logger的设置-->
  <root>
    <level value="warn"/>
    <appender-ref ref="STDOUT"/>
  </root>
</log4j:configuration>
```

参考

- slf4 官方文档
- logback 官方文档
- log4j 官方文档
- commons-logging 官方文档
- http://blog.csdn.net/yycdaizi/article/details/8276265

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 15:29:01

- 细说 Java 主流工具包

细说 Java 主流工具包

- apache.commons
  - commons-lang
  - commons-collections
  - common-io
- guava

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 15:50:03

- JavaMail 使用指南
  - 简介
    - 邮件相关的标准
    - JavaMail 简介
    - 邮件传输过程
    - Message 结构
  - JavaMail 的核心类
    - java.util.Properties 类（属性对象）
    - javax.mail.Session 类（会话对象）
    - javax.mail.Transport 类（邮件传输）
    - javax.mail.Store 类（邮件存储 ）
    - javax.mail.Message 类（消息对象）
    - javax.mail.Address 类（地址）
    - Authenticator 类（认证者）
  - 实例
    - 发送文本邮件
    - 发送 HTML 格式的邮件
    - 发送带附件的邮件
    - 获取邮箱中的邮件
    - 转发邮件

JavaMail 使用指南

- 简介
  - 邮件相关的标准
  - JavaMail 简介
  - 邮件传输过程
  - Message 结构
- JavaMail 的核心类
  - java.util.Properties 类（属性对象）
  - javax.mail.Session 类（会话对象）
  - javax.mail.Transport 类（邮件传输）
  - javax.mail.Store 类（邮件存储 ）
  - javax.mail.Message 类（消息对象）
  - javax.mail.Address 类（地址）
  - Authenticator 类（认证者）
- 实例
  - 发送文本邮件
  - 发送 HTML 格式的邮件
  - 发送带附件的邮件
  - 获取邮箱中的邮件
  - 转发邮件

简介

邮件相关的标准

厂商所提供的 JavaMail 服务程序可以有选择地实现某些邮件协议，常见的邮件协议包括：

- `SMTP(Simple Mail Transfer Protocol)` ：即简单邮件传输协议，它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。
- `POP3(Post Office Protocol - Version 3)` ：即邮局协议版本 3 ，用于接收电子邮件的标准协议。
- `IMAP(Internet Mail Access Protocol)` ：即 Internet 邮件访问协议。是 POP3 的替代协议。

这三种协议都有对应 SSL 加密传输的协议，分别是 **SMTPS** ， **POP3S** 和 **IMAPS** 。

`MIME(Multipurpose Internet Mail Extensions)` ：即多用途因特网邮件扩展标准。它不是邮件传输协议。但对传输内容的消息、附件及其它的内容定义了格式。

JavaMail 简介

JavaMail 是由 Sun 发布的用来处理 email 的 API 。它并没有包含在 Java SE 中，而是作为 Java EE 的一部分。

- `mail.jar` ：此 JAR 文件包含 JavaMail API 和 Sun 提供的 SMTP 、 IMAP 和 POP3 服务提供程序；
- `activation.jar` ：此 JAR 文件包含 JAF API 和 Sun 的实现。

JavaMail 包中用于处理电子邮件的核心类是： `Properties` 、 `Session` 、 `Message` 、 `Address` 、 `Authenticator` 、 `Transport` 、 `Store` 等。

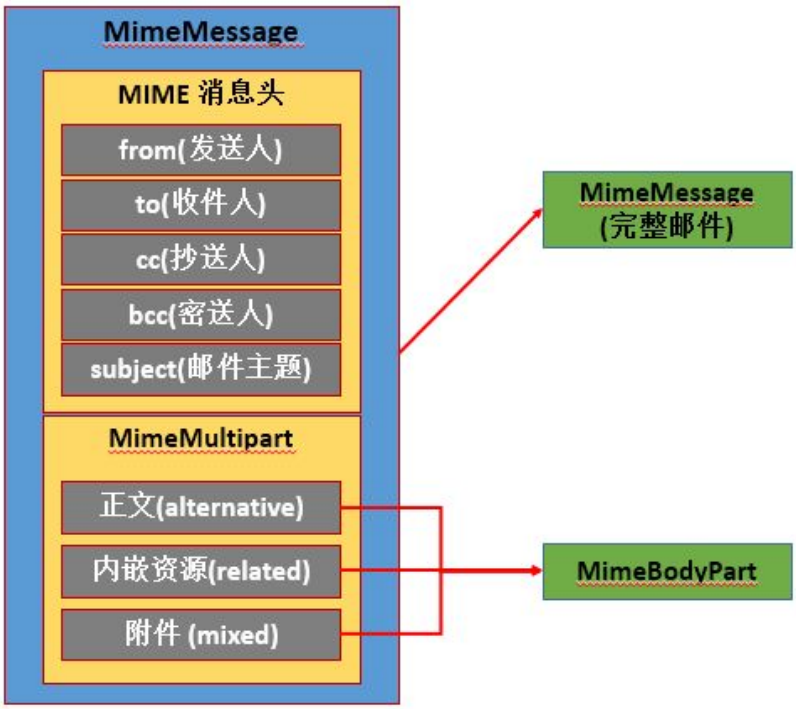
邮件传输过程

如上图，电子邮件的处理步骤如下：

1. 创建一个 Session 对象。
2. Session 对象创建一个 Transport 对象 /Store 对象，用来发送 / 保存邮件。
3. Transport 对象 /Store 对象连接邮件服务器。
4. Transport 对象 /Store 对象创建一个 Message 对象 (也就是邮件内容) 。
5. Transport 对象发送邮件； Store 对象获取邮箱的邮件。

Message 结构

- `MimeMessage` 类：代表整封邮件。
- `MimeBodyPart` 类：代表邮件的一个 MIME 信息。
- `MimeMultipart` 类：代表一个由多个 MIME 信息组合成的组合 MIME 信息。



JavaMail 的核心类

JavaMail 对收发邮件进行了高级的抽象，形成了一些关键的的接口和类，它们构成了程序的基础，下面我们分别来了解一下这些最常见的对象。

java.util.Properties 类（属性对象）

java.util.Properties 类代表一组属性集合。

它的每一个键和值都是 String 类型。

由于 JavaMail 需要和邮件服务器进行通信，这就要求程序提供许多诸如服务器地址、端口、用户名、密码等信息， JavaMail 通过 Properties 对象封装这些属性信息。

例： 如下面的代码封装了几个属性信息：

```
Properties prop = new Properties();
prop.setProperty("mail.debug", "true");
prop.setProperty("mail.host", "[email protected]");
prop.setProperty("mail.transport.protocol", "smtp");
prop.setProperty("mail.smtp.auth", "true");
```

针对不同的的邮件协议， JavaMail 规定了服务提供者必须支持一系列属性，

下表是一些常见属性（属性值都以 String 类型进行设置，属性类型栏仅表示属性是如何被解析的）：

关键词	类型	描述
mail.debug	boolean	debug 开关。
mail.host	String	指定发送、接收邮件的默认邮箱服务器。
mail.store.protocol	String	指定接收邮件的协议。
mail.transport.protocol	String	指定发送邮件的协议。
mail.debug.auth	boolean	debug 输出中是否包含认证命令。默认是 false 。

详情请参考官方 API 文档：

https://javamail.java.net/nonav/docs/api/ 。

javax.mail.Session 类（会话对象）

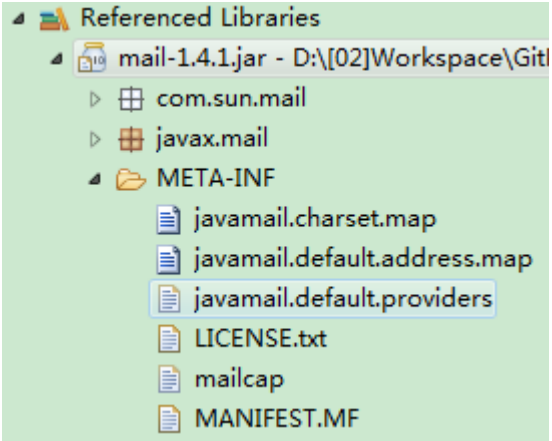
`Session` 表示一个邮件会话。

Session 的主要作用包括两个方面：

- 接收各种配置属性信息：通过 Properties 对象设置的属性信息；
- 初始化 JavaMail 环境：根据 JavaMail 的配置文件，初始化 JavaMail 环境，以便通过 Session 对象创建其他重要类的实例。

JavaMail 在 Jar 包的 META-INF 目录下，通过以下文件提供了基本配置信息，以便 session 能够根据这个配置文件加载提供者的实现类：

- javamail.default.providers
- javamail.default.address.map



例：

```
Properties props = new Properties();
props.setProperty("mail.transport.protocol", "smtp");
Session session = Session.getInstance(props);
```

javax.mail.Transport 类（邮件传输）

邮件操作只有发送或接收两种处理方式。

JavaMail 将这两种不同操作描述为传输（ javax.mail.Transport ）和存储（ javax.mail.Store ），传输对应邮件的发送，而存储对应邮件的接收。

- `getTransport` - Session 类中的 getTransport () 有多个重载方法，可以用来创建 Transport 对象。
- `connect` - 如果设置了认证命令—— mail.smtp.auth ，那么使用 Transport 类的 connect 方法连接服务器时，则必须加上用户名和密码。
- `sendMessage` - Transport 类的 sendMessage 方法用来发送邮件消息。
- `close` - Transport 类的 close 方法用来关闭和邮件服务器的连接。

javax.mail.Store 类（邮件存储 ）

- `getStore` - Session 类中的 getStore () 有多个重载方法，可以用来创建 Store 对象。
- `connect` - 如果设置了认证命令—— mail.smtp.auth ，那么使用 Store 类的 connect 方法连接服务器时，则必须加上用户名和密码。
- `getFolder` - Store 类的 getFolder 方法可以 获取邮箱内的邮件夹 Folder 对象
- `close` - Store 类的 close 方法用来关闭和邮件服务器的连接。

javax.mail.Message 类（消息对象）

- `javax.mail.Message` - 是个抽象类，只能用子类去实例化，多数情况下为 `javax.mail.internet.MimeMessage` 。
- `MimeMessage` - 代表 MIME 类型的电子邮件消息。

要创建一个 Message ，需要将 Session 对象传递给 `MimeMessage` 构造器：

```
MimeMessage message = new MimeMessage(session);
```

注意：还存在其它构造器，如用按 RFC822 格式的输入流来创建消息。

- setFrom - 设置邮件的发件人
- setRecipient - 设置邮件的发送人、抄送人、密送人

三种预定义的地址类型是：

- `Message.RecipientType.TO` - 收件人
- `Message.RecipientType.CC` - 抄送人
- `Message.RecipientType.BCC` - 密送人
- `setSubject` - 设置邮件的主题
- `setContent` - 设置邮件内容
- `setText` - 如果邮件内容是纯文本，可以使用此接口设置文本内容。

javax.mail.Address 类（地址）

一旦您创建了 Session 和 Message ，并将内容填入消息后，就可以用 Address 确定信件地址了。和 Message 一样， Address 也是个抽象类。您用的是 javax.mail.internet.InternetAddress 类。

若创建的地址只包含电子邮件地址，只要传递电子邮件地址到构造器就行了。

例：

```
Address address = new InternetAddress("[email protected]");
```

Authenticator 类（认证者）

与 java.net 类一样， JavaMail API 也可以利用 `Authenticator` 通过用户名和密码访问受保护的资源。对于 JavaMail API 来说，这些资源就是邮件服务器。 `Authenticator` 在 javax.mail 包中，而且它和 java.net 中同名的类 Authenticator 不同。两者并不共享同一个 Authenticator ，因为 JavaMail API 用于 Java 1.1 ，它没有 java.net 类别。

要使用 Authenticator ，先创建一个抽象类的子类，并从 `getPasswordAuthentication()` 方法中返回 `PasswordAuthentication` 实例。创建完成后，您必需向 session 注册 `Authenticator` 。然后，在需要认证的时候，就会通知 `Authenticator` 。您可以弹出窗口，也可以从配置文件中（虽然没有加密是不安全的）读取用户名和密码，将它们作为 `PasswordAuthentication` 对象返回给调用程序。

例：

```
Properties props = new Properties();
Authenticator auth = new MyAuthenticator();
Session session = Session.getDefaultInstance(props, auth);
```

实例

发送文本邮件

```
public static void main(String[] args) throws Exception {
    Properties prop = new Properties();
    prop.setProperty("mail.debug", "true");
    prop.setProperty("mail.host", MAIL_SERVER_HOST);
    prop.setProperty("mail.transport.protocol", "smtp");
    prop.setProperty("mail.smtp.auth", "true");

    // 1、创建session
    Session session = Session.getInstance(prop);
    Transport ts = null;

    // 2、通过session得到transport对象
    ts = session.getTransport();

    // 3、连上邮件服务器
    ts.connect(MAIL_SERVER_HOST, USER, PASSWORD);

    // 4、创建邮件
    MimeMessage message = new MimeMessage(session);

    // 邮件消息头
    message.setFrom(new InternetAddress(MAIL_FROM)); // 邮件的发
    message.setRecipient(Message.RecipientType.TO, new InternetA
    message.setRecipient(Message.RecipientType.CC, new InternetA
    message.setRecipient(Message.RecipientType.BCC, new Internet
    message.setSubject("测试文本邮件"); // 邮件的标题

    // 邮件消息体
    message.setText("天下无双。");

    // 5、发送邮件
    ts.sendMessage(message, message.getAllRecipients());
    ts.close();
}
```

发送 HTML 格式的邮件

```java
public static void main(String[] args) throws Exception {
    Properties prop = new Properties();
    prop.setProperty("mail.debug", "true");
    prop.setProperty("mail.host", MAIL_SERVER_HOST);
    prop.setProperty("mail.transport.protocol", "smtp");
    prop.setProperty("mail.smtp.auth", "true");

    // 1、创建session
    Session session = Session.getInstance(prop);
    Transport ts = null;

    // 2、通过session得到transport对象
    ts = session.getTransport();

    // 3、连上邮件服务器
    ts.connect(MAIL_SERVER_HOST, USER, PASSWORD);

    // 4、创建邮件
    MimeMessage message = new MimeMessage(session);

    // 邮件消息头
    message.setFrom(new InternetAddress(MAIL_FROM)); // 邮件的发
    message.setRecipient(Message.RecipientType.TO, new InternetA
    message.setRecipient(Message.RecipientType.CC, new InternetA
    message.setRecipient(Message.RecipientType.BCC, new Internet
    message.setSubject("测试HTML邮件"); // 邮件的标题

    String htmlContent = "<h1>Hello</h1>" + "<p>显示图片<img src=
    MimeBodyPart text = new MimeBodyPart();
    text.setContent(htmlContent, "text/html;charset=UTF-8");
    MimeBodyPart image = new MimeBodyPart();
    DataHandler dh = new DataHandler(new FileDataSource("D:\\05_
    image.setDataHandler(dh);
    image.setContentID("abc.jpg");

    // 描述数据关系
    MimeMultipart mm = new MimeMultipart();
    mm.addBodyPart(text);
    mm.addBodyPart(image);
    mm.setSubType("related");
    message.setContent(mm);
    message.saveChanges();

    // 5、发送邮件
    ts.sendMessage(message, message.getAllRecipients());
    ts.close();
}
```

发送带附件的邮件

```java
public static void main(String[] args) throws Exception {
    Properties prop = new Properties();
    prop.setProperty("mail.debug", "true");
    prop.setProperty("mail.host", MAIL_SERVER_HOST);
    prop.setProperty("mail.transport.protocol", "smtp");
    prop.setProperty("mail.smtp.auth", "true");

    // 1、创建session
    Session session = Session.getInstance(prop);

    // 2、通过session得到transport对象
    Transport ts = session.getTransport();

    // 3、连上邮件服务器
    ts.connect(MAIL_SERVER_HOST, USER, PASSWORD);

    // 4、创建邮件
    MimeMessage message = new MimeMessage(session);

    // 邮件消息头
    message.setFrom(new InternetAddress(MAIL_FROM)); // 邮件的发
    message.setRecipient(Message.RecipientType.TO, new InternetA
    message.setRecipient(Message.RecipientType.CC, new InternetA
    message.setRecipient(Message.RecipientType.BCC, new Internet
    message.setSubject("测试带附件邮件"); // 邮件的标题

    MimeBodyPart text = new MimeBodyPart();
    text.setContent("邮件中有两个附件。", "text/html;charset=UTF-8

    // 描述数据关系
    MimeMultipart mm = new MimeMultipart();
    mm.setSubType("related");
    mm.addBodyPart(text);
    String[] files = {
            "D:\\00_Temp\\temp\\1.jpg", "D:\\00_Temp\\temp\\2.pn
    };

    // 添加邮件附件
    for (String filename : files) {
        MimeBodyPart attachPart = new MimeBodyPart();
        attachPart.attachFile(filename);
        mm.addBodyPart(attachPart);
    }

    message.setContent(mm);
    message.saveChanges();

    // 5、发送邮件
```

```
    ts.sendMessage(message, message.getAllRecipients());
    ts.close();
}
```

获取邮箱中的邮件

```
public static void main(String[] args) throws Exception {

    // 创建一个有具体连接信息的Properties对象
    Properties prop = new Properties();
    prop.setProperty("mail.debug", "true");
    prop.setProperty("mail.store.protocol", "pop3");
    prop.setProperty("mail.pop3.host", MAIL_SERVER_HOST);

    // 1、创建session
    Session session = Session.getInstance(prop);

    // 2、通过session得到Store对象
    Store store = session.getStore();

    // 3、连上邮件服务器
    store.connect(MAIL_SERVER_HOST, USER, PASSWORD);

    // 4、获得邮箱内的邮件夹
    Folder folder = store.getFolder("inbox");
    folder.open(Folder.READ_ONLY);

    // 获得邮件夹Folder内的所有邮件Message对象
    Message[] messages = folder.getMessages();
    for (int i = 0; i < messages.length; i++) {
        String subject = messages[i].getSubject();
        String from = (messages[i].getFrom()[0]).toString();
        System.out.println("第 " + (i + 1) + "封邮件的主题：" + su
        System.out.println("第 " + (i + 1) + "封邮件的发件人地址：
    }

    // 5、关闭
    folder.close(false);
    store.close();
}
```

转发邮件

例：获取指定邮件夹下的第一封邮件并转发

```java
public static void main(String[] args) throws Exception {
    Properties prop = new Properties();
    prop.put("mail.store.protocol", "pop3");
    prop.put("mail.pop3.host", MAIL_SERVER_POP3);
    prop.put("mail.pop3.starttls.enable", "true");
    prop.put("mail.smtp.auth", "true");
    prop.put("mail.smtp.host", MAIL_SERVER_SMTP);

    // 1、创建session
    Session session = Session.getDefaultInstance(prop);

    // 2、读取邮件夹
    Store store = session.getStore("pop3");
    store.connect(MAIL_SERVER_POP3, USER, PASSWORD);
    Folder folder = store.getFolder("inbox");
    folder.open(Folder.READ_ONLY);

    // 获取邮件夹中第1封邮件信息
    Message[] messages = folder.getMessages();
    if (messages.length <= 0) {
        return;
    }
    Message message = messages[0];

    // 打印邮件关键信息
    String from = InternetAddress.toString(message.getFrom());
    if (from != null) {
        System.out.println("From: " + from);
    }

    String replyTo = InternetAddress.toString(message.getReplyTo
    if (replyTo != null) {
        System.out.println("Reply-to: " + replyTo);
    }

    String to = InternetAddress.toString(message.getRecipients(M
    if (to != null) {
        System.out.println("To: " + to);
    }

    String subject = message.getSubject();
    if (subject != null) {
        System.out.println("Subject: " + subject);
    }

    Date sent = message.getSentDate();
    if (sent != null) {
        System.out.println("Sent: " + sent);
```

```
    }

    // 设置转发邮件信息头
    Message forward = new MimeMessage(session);
    forward.setFrom(new InternetAddress(MAIL_FROM));
    forward.setRecipient(Message.RecipientType.TO, new InternetA
    forward.setSubject("Fwd: " + message.getSubject());

    // 设置转发邮件内容
    MimeBodyPart bodyPart = new MimeBodyPart();
    bodyPart.setContent(message, "message/rfc822");

    Multipart multipart = new MimeMultipart();
    multipart.addBodyPart(bodyPart);
    forward.setContent(multipart);
    forward.saveChanges();

    Transport ts = session.getTransport("smtp");
    ts.connect(USER, PASSWORD);
    ts.sendMessage(forward, forward.getAllRecipients());

    folder.close(false);
    store.close();
    ts.close();
    System.out.println("message forwarded successfully....");
}
```

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 17:21:39

- Jsoup 使用指南
  - 简介
  - 加载
    - 从HTML字符串加载一个文档
    - 解析一个body片断
    - 从URL加载一个文档
    - 从一个文件加载一个文档
  - 解析
    - 使用DOM方法来遍历一个文档
    - 使用选择器语法来查找元素
    - 从元素抽取属性，文本和HTML
    - 处理URLs
  - 数据修改
    - 设置属性的值
    - 设置一个元素的HTML内容
    - 设置元素的文本内容
  - HTML清理
    - 消除不受信任的HTML (来防止XSS攻击)
  - 参考

Jsoup 使用指南

- 简介
- 加载
  - 从HTML字符串加载一个文档
  - 解析一个body片断
  - 从URL加载一个文档
  - 从一个文件加载一个文档
- 解析
  - 使用DOM方法来遍历一个文档
  - 使用选择器语法来查找元素
  - 从元素抽取属性，文本和HTML
  - 处理URLs
- 数据修改
  - 设置属性的值
  - 设置一个元素的HTML内容
  - 设置元素的文本内容
- HTML清理
  - 消除不受信任的HTML (来防止XSS攻击)
- 参考

简介

jsoup 是一款 Java 的 HTML 解析器，可直接解析某个 URL 地址、HTML 文本内容。它提供了一套非常省力的 API，可通过 DOM，CSS 以及类似于 JQuery 的操作方法来取出和操作数据。

jsoup 工作的流程主要如下：

1. 从一个 URL，文件或字符串中解析 HTML，并加载为一个 `Document` 对象。
2. 使用 DOM 或 CSS 选择器来取出数据；
3. 可操作 HTML 元素、属性、文本。

jsoup 是基于 MIT 协议发布的，可放心使用于商业项目。

加载

从HTML字符串加载一个文档

使用静态 `Jsoup.parse(String html)` 方法或 `Jsoup.parse(String html, String baseUri)` 示例代码：

```
String html = "<html><head><title>First parse</title></head>"
  + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

> 说明
>
> `parse(String html, String baseUri)` 这方法能够将输入的HTML解析为一个新的文档 (Document），参数 baseUri 是用来将相对 URL 转成绝对URL，并指定从哪个网站获取文档。如这个方法不适用，你可以使用 `parse(String html)` 方法来解析成HTML字符串如上面的示例。
>
> 只要解析的不是空字符串，就能返回一个结构合理的文档，其中包含(至少) 一个head和一个body元素。
>
> 一旦拥有了一个Document，你就可以使用Document中适当的方法或它父类 `Element` 和 `Node` 中的方法来取得相关数据。

解析一个body片断

问题

假如你有一个HTML片断 (比如. 一个 `div` 包含一对 `p` 标签; 一个不完整的HTML文档) 想对它进行解析。这个HTML片断可以是用户提交的一条评论或在一个CMS页面中编辑body部分。

办法

使用 `Jsoup.parseBodyFragment(String html)` 方法.

```
String html = "<div><p>Lorem ipsum.</p>";
Document doc = Jsoup.parseBodyFragment(html);
Element body = doc.body();
```

> 说明
>
> `parseBodyFragment` 方法创建一个空壳的文档，并插入解析过的HTML到 `body` 元素中。假如你使用正常的 `Jsoup.parse(String html)` 方法，通常你也可以得到相同的结果，但是明确将用户输入作为 body片段处理，以确保用户所提供的任何糟糕的HTML都将被解析成body元素。
>
> `Document.body()` 方法能够取得文档body元素的所有子元素，与 `doc.getElementsByTag("body")` 相同。

保证安全Stay safe

假如你可以让用户输入HTML内容，那么要小心避免跨站脚本攻击。利用基于 `Whitelist` 的清除器和 `clean(String bodyHtml, Whitelist whitelist)` 方法来清除用户输入的恶意内容。

从URL加载一个文档

使用 `Jsoup.connect(String url)` 方法

```
Document doc = Jsoup.connect("http://example.com/").get();
```

> 说明
>
> `connect(String url)` 方法创建一个新的 `Connection` , 和 `get()` 取得和解析一个HTML文件。如果从该URL获取HTML时发生错误，便会抛出 IOException，应适当处理。

`Connection` 接口还提供一个方法链来解决特殊请求，具体如下：

```
Document doc = Jsoup.connect("http://example.com")
  .data("query", "Java")
  .userAgent("Mozilla")
  .cookie("auth", "token")
  .timeout(3000)
  .post();
```

从一个文件加载一个文档

可以使用静态 `Jsoup.parse(File in, String charsetName, String baseUri)` 方法

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/"
```

> 说明
>
> `parse(File in, String charsetName, String baseUri)` 这个方法用来加载和解析一个HTML文件。如在加载文件的时候发生错误，将抛出IOException，应作适当处理。
>
> `baseUri` 参数用于解决文件中URLs是相对路径的问题。如果不需要可以传入一个空的字符串。
>
> 另外还有一个方法 `parse(File in, String charsetName)` ，它使用文件的路径做为 `baseUri` 。 这个方法适用于如果被解析文件位于网站的本地文件系统，且相关链接也指向该文件系统。

解析

使用DOM方法来遍历一个文档

问题

你有一个HTML文档要从中提取数据，并了解这个HTML文档的结构。

方法

将HTML解析成一个 `Document` 之后，就可以使用类似于DOM的方法进行操作。示例代码：

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/"

Element content = doc.getElementById("content");
Elements links = content.getElementsByTag("a");
for (Element link : links) {
  String linkHref = link.attr("href");
  String linkText = link.text();
}
```

说明

`Elements` 这个对象提供了一系列类似于 DOM 的方法来查找元素，抽取并处理其中的数据。

具体如下：

查找元素

- `getElementById(String id)`
- `getElementsByTag(String tag)`
- `getElementsByClass(String className)`
- `getElementsByAttribute(String key)` (and related methods)
- Element siblings: `siblingElements()`，`firstElementSibling()`，`lastElementSibling()`；`nextElementSibling()`，`previousElementSibling()`
- Graph: `parent()`，`children()`，`child(int index)`

元素数据

- `attr(String key)` 获取属性 `attr(String key, String value)` 设置属性
- `attributes()` 获取所有属性
- `id()`，`className()` and `classNames()`
- `text()` 获取文本内容 `text(String value)` 设置文本内容
- `html()` 获取元素内HTML `html(String value)` 设置元素内的HTML内容
- `outerHtml()` 获取元素外HTML内容
- `data()` 获取数据内容（例如：script和style标签)
- `tag()` and `tagName()`

操作**HTML**和文本

- `append(String html)`，`prepend(String html)`
- `appendText(String text)`，`prependText(String text)`
- `appendElement(String tagName)`，`prependElement(String tagName)`
- `html(String value)`

使用选择器语法来查找元素

问题

你想使用类似于CSS或jQuery的语法来查找和操作元素。

方法

可以使用 `Element.select(String selector)` 和 `Elements.select(String selector)` 方法实现：

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/"

Elements links = doc.select("a[href]"); //带有href属性的a元素
Elements pngs = doc.select("img[src$=.png]");
  //扩展名为.png的图片

Element masthead = doc.select("div.masthead").first();
  //class等于masthead的div标签

Elements resultLinks = doc.select("h3.r > a"); //在h3元素之后的a元
```

> 说明
>
> jsoup elements对象支持类似于CSS (或jquery)的选择器语法，来实现非常强大和灵活的查找功能。.
>
> 这个 `select` 方法在 `Document` , `Element` ,或 `Elements` 对象中都可以使用。且是上下文相关的，因此可实现指定元素的过滤，或者链式选择访问。
>
> Select方法将返回一个 `Elements` 集合，并提供一组方法来抽取和处理结果。

Selector选择器概述

- `tagname` : 通过标签查找元素，比如： `a`
- `ns|tag` : 通过标签在命名空间查找元素，比如：可以用 `fb|name` 语法来查找 `` 元素
- `#id` : 通过ID查找元素，比如： `#logo`
- `.class` : 通过class名称查找元素，比如： `.masthead`
- `[attribute]` : 利用属性查找元素，比如： `[href]`
- `[^attr]` : 利用属性名前缀来查找元素，比如：可以用 `[^data-]` 来查找带有HTML5 Dataset属性的元素
- `[attr=value]` : 利用属性值来查找元素，比如： `[width=500]`
- `[attr^=value]` , `[attr$=value]` , `[attr*=value]` : 利用匹配属性值开头、结尾或包含属性值来查找元素，比如： `[href*=/path/]`
- `[attr\~=regex]` : 利用属性值匹配正则表达式来查找元素，比如： `img[src\~=(?i)\.(png|jpe?g)]`
- `*` : 这个符号将匹配所有元素

Selector选择器组合使用

- `el##id` : 元素+ID，比如： `div##logo`
- `el.class` : 元素+class，比如： `div.masthead`

- `el[attr]`：元素+class，比如：`a[href]`
- 任意组合，比如：`a[href].highlight`
- `ancestor child`：查找某个元素下子元素，比如：可以用`.body p`查找在"body"元素下的所有`p`元素
- `parent > child`：查找某个父元素下的直接子元素，比如：可以用`div.content > p`查找`p`元素，也可以用`body > *`查找body标签下所有直接子元素
- `siblingA + siblingB`：查找在A元素之前第一个同级元素B，比如：`div.head + div`
- `siblingA \~ siblingX`：查找A元素之前的同级X元素，比如：`h1 \~ p`
- `el, el, el`:多个选择器组合，查找匹配任一选择器的唯一元素，例如：`div.masthead, div.logo`

伪选择器selectors

- `:lt(n)`：查找哪些元素的同级索引值（它的位置在DOM树中是相对于它的父节点）小于n，比如：`td:lt(3)`表示小于三列的元素
- `:gt(n)`:查找哪些元素的同级索引值大于`n``，比如：`div p:gt(2)`表示哪些div中有包含2个以上的p元素
- `:eq(n)`：查找哪些元素的同级索引值与`n`相等，比如：`form input:eq(1)`表示包含一个input标签的Form元素
- `:has(seletor)`：查找匹配选择器包含元素的元素，比如：`div:has(p)`表示哪些div包含了p元素
- `:not(selector)`：查找与选择器不匹配的元素，比如：`div:not(.logo)`表示不包含 class=logo 元素的所有 div 列表
- `:contains(text)`：查找包含给定文本的元素，搜索不区分大不写，比如：`p:contains(jsoup)`
- `:containsOwn(text)`：查找直接包含给定文本的元素
- `:matches(regex)`：查找哪些元素的文本匹配指定的正则表达式，比如：`div:matches((?i)login)`
- `:matchesOwn(regex)`：查找自身包含文本匹配指定正则表达式的元素
- 注意：上述伪选择器索引是从0开始的，也就是说第一个元素索引值为0，第二个元素index为1等

可以查看`Selector` API参考来了解更详细的内容

从元素抽取属性，文本和HTML

问题

在解析获得一个Document实例对象，并查找到一些元素之后，你希望取得在这些元素中的数据。

方法

- 要取得一个属性的值，可以使用 `Node.attr(String key)` 方法
- 对于一个元素中的文本，可以使用 `Element.text()` 方法
- 对于要取得元素或属性中的HTML内容，可以使用 `Element.html()` ，或 `Node.outerHtml()` 方法

示例：

```
String html = "<p>An <a href='http://example.com/'><b>example</b
Document doc = Jsoup.parse(html);//解析HTML字符串返回一个Document
Element link = doc.select("a").first();//查找第一个a元素

String text = doc.body().text(); // "An example link"//取得字符串
String linkHref = link.attr("href"); // "http://example.com/"//取
String linkText = link.text(); // "example""//取得链接地址中的文本

String linkOuterH = link.outerHtml();
    // "<a href="http://example.com"><b>example</b></a>"
String linkInnerH = link.html(); // "<b>example</b>"//取得链接内的
```

> 说明
>
> 上述方法是元素数据访问的核心办法。此外还其它一些方法可以使用：
>
> - `Element.id()`
> - `Element.tagName()`
> - `Element.className()` and `Element.hasClass(String className)`
>
> 这些访问器方法都有相应的setter方法来更改数据

参见

- `Element` 和 `Elements` 集合类的参考文档
- URLs处理
- 使用CSS选择器语法来查找元素

处理URLs

问题

你有一个包含相对URLs路径的HTML文档，需要将这些相对路径转换成绝对路径的URLs。

方法

1. 在你解析文档时确保有指定 `base URI` ，然后

2. 使用 `abs:` 属性前缀来取得包含 `base URI` 的绝对路径。代码如下：

```
Document doc = Jsoup.connect("http://www.open-open.com").get();

Element link = doc.select("a").first();
String relHref = link.attr("href"); // == "/"
String absHref = link.attr("abs:href"); // "http://www.open-open
```

> 说明
>
> 在HTML元素中，URLs 经常写成相对于文档位置的相对路径： `<a href="/download">...</a>` . 当你使用 `Node.attr(String key)` 方法来取得a元素的href属性时，它将直接返回在HTML源码中指定定的值。
>
> 假如你需要取得一个绝对路径，需要在属性名前加 `abs:` 前缀。这样就可以返回包含根路径的URL地址 `attr("abs:href")`
>
> 因此，在解析HTML文档时，定义base URI非常重要。
>
> 如果你不想使用 `abs:` 前缀，还有一个方法能够实现同样的功能 `Node.absUrl(String key)` 。

数据修改

设置属性的值

问题

在你解析一个 `Document` 之后可能想修改其中的某些属性值，然后再保存到磁盘或都输出到前台页面。

方法

可以使用属性设置方法 `Element.attr(String key, String value)` , 和 `Elements.attr(String key, String value)` .

假如你需要修改一个元素的 `class` 属性，可以使用 `Element.addClass(String className)` 和 `Element.removeClass(String className)` 方法。

`Elements` 提供了批量操作元素属性和class的方法，比如：要为div中的每一个a元素都添加一个 `rel="nofollow"` 可以使用如下方法：

```
doc.select("div.comments a").attr("rel", "nofollow");
```

> 说明
>
> 与 `Element` 中的其它方法一样， `attr` 方法也是返回当 `Element` (或在使用选择器是返回 `Elements` 集合)。这样能够很方便使用方法连用的书写方式。比如：
>
> ```
> doc.select("div.masthead").attr("title", "jsoup").addClass(
> ```

设置一个元素的HTML内容

问题

你需要一个元素中的HTML内容

方法

可以使用 `Element` 中的HTML设置方法具体如下：

```
Element div = doc.select("div").first(); // <div></div>
div.html("<p>lorem ipsum</p>"); // <div><p>lorem ipsum</p></div>
div.prepend("<p>First</p>");//在div前添加html内容
div.append("<p>Last</p>");//在div之后添加html内容
// 添完后的结果: <div><p>First</p><p>lorem ipsum</p><p>Last</p></

Element span = doc.select("span").first(); // <span>One</span>
span.wrap("<li><a href='http://example.com/'></a></li>");
// 添完后的结果: <li><a href="http://example.com"><span>One</span
```

> 说明
>
> - `Element.html(String html)` 这个方法将先清除元素中的HTML内容，然后用传入的HTML代替。
> - `Element.prepend(String first)` 和 `Element.append(String last)` 方法用于在分别在元素内部HTML的前面和后面添加HTML内容
> - `Element.wrap(String around)` 对元素包裹一个外部HTML内容。
>
> 参见
>
> 可以查看API参考文档中 `Element.prependElement(String tag)` 和 `Element.appendElement(String tag)` 方法来创建新的元素并作为文档的子元素插入其中。

设置元素的文本内容

问题

你需要修改一个HTML文档中的文本内容

方法

可以使用 `Element` 的设置方法：：

```
Element div = doc.select("div").first(); // <div></div>
div.text("five > four"); // <div>five &gt; four</div>
div.prepend("First ");
div.append(" Last");
// now: <div>First five &gt; four Last</div>
```

> 说明
>
> 文本设置方法与 HTML setter 方法一样：
>
> - `Element.text(String text)` 将清除一个元素中的内部HTML内容，然后提供的文本进行代替
> - `Element.prepend(String first)` 和 `Element.append(String last)` 将分别在元素的内部html前后添加文本节点。
>
> 对于传入的文本如果含有像 `<` , `>` 等这样的字符，将以文本处理，而非HTML。

HTML清理

消除不受信任的HTML (来防止XSS攻击)

问题

在做网站的时候，经常会提供用户评论的功能。有些不怀好意的用户，会搞一些脚本到评论内容中，而这些脚本可能会破坏整个页面的行为，更严重的是获取一些机要信息，此时需要清理该HTML，以避免跨站脚本 cross-site scripting攻击（XSS）。

方法

使用jsoup HTML `Cleaner` 方法进行清除，但需要指定一个可配置的 `Whitelist` 。

```
String unsafe =
  "<p><a href='http://example.com/' onclick='stealCookies()'>Lin
String safe = Jsoup.clean(unsafe, Whitelist.basic());
// now: <p><a href="http://example.com/" rel="nofollow">Link</a>
```

说明

XSS 又叫 CSS (Cross Site Script)，跨站脚本攻击。它指的是恶意攻击者往 Web 页面里插入恶意 html 代码，当用户浏览该页之时，嵌入其中 Web 里面的 html 代码会被执行，从而达到恶意攻击用户的特殊目的。XSS 属于被动式的攻击，因为其被动且不好利用，所以许多人常忽略其危害性。所以我们经常只让用户输入纯文本的内容，但这样用户体验就比较差了。

一个更好的解决方法就是使用一个富文本编辑器 WYSIWYG 如 CKEditor 和 TinyMCE。这些可以输出 HTML 并能够让用户可视化编辑。虽然他们可以在客户端进行校验，但是这样还不够安全，需要在服务器端进行校验并清除有害的HTML代码，这样才能确保输入到你网站的HTML是安全的。否则，攻击者能够绕过客户端的 Javascript 验证，并注入不安全的 HMTL 直接进入您的网站。

jsoup 的 whitelist 清理器能够在服务器端对用户输入的 HTML 进行过滤，只输出一些安全的标签和属性。

jsoup 提供了一系列的 `Whitelist` 基本配置，能够满足大多数要求；但如有必要，也可以进行修改，不过要小心。

这个 cleaner 非常好用不仅可以避免 XSS 攻击，还可以限制用户可以输入的标签范围。

参见

- 参阅XSS cheat sheet，有一个例子可以了解为什么不能使用正则表达式，而采用安全的whitelist parser-based清理器才是正确的选择。
- 参阅 `Cleaner`，了解如何返回一个 `Document` 对象，而不是字符串
- 参阅 `Whitelist`，了解如何创建一个自定义的whitelist
- nofollow 链接属性了解

参考

- jsoup github托管代码
- jsoup Cookbook
- jsoup Cookbook(中文版)
- 不错的jsoup学习笔记

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 15:50:03

- JUnit5 使用指南
  - 1. 安装
  - 2. JUnit 注解
  - 3. 编写单元测试
    - 3.1. 基本的单元测试类和方法
    - 3.2. 定制测试类和方法的显示名称
    - 3.3. 断言（Assertions）
    - 3.4. 假想（Assumptions）
    - 3.5. 禁用
    - 3.6. 测试条件
    - 3.7. 嵌套测试
    - 3.8. 重复测试
    - 3.9. 参数化测试
  - 4. 引用和引申

JUnit5 使用指南

> version: junit5

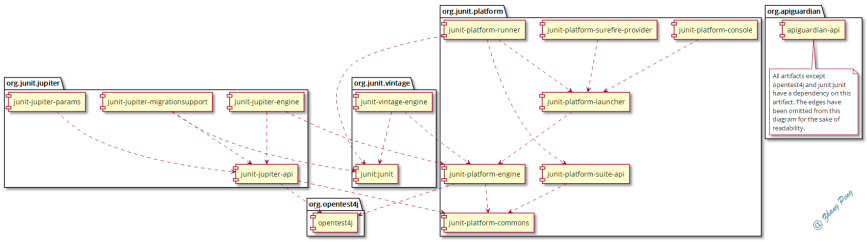
- 1. 安装
- 2. JUnit 注解
- 3. 编写单元测试
  - 3.1. 基本的单元测试类和方法
  - 3.2. 定制测试类和方法的显示名称
  - 3.3. 断言（Assertions）
  - 3.4. 假想（Assumptions）
  - 3.5. 禁用
  - 3.6. 测试条件
  - 3.7. 嵌套测试
  - 3.8. 重复测试
  - 3.9. 参数化测试
- 4. 引用和引申

1. 安装

在 pom 中添加依赖

```
<properties>
  <junit.jupiter.version>5.3.2</junit.jupiter.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

组件间依赖关系：



2. JUnit 注解

Annotation	Description
`@Test`	Denotes that a method is a test method. Unlike JUnit 4's `@Test` annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are *inherited* unless they are *overridden*.
`@ParameterizedTest`	Denotes that a method is a parameterized test. Such methods are *inherited* unless they are *overridden*.
`@RepeatedTest`	Denotes that a method is a test template for a repeated test. Such methods are *inherited* unless they are *overridden*.
`@TestFactory`	Denotes that a method is a test factory for dynamic tests. Such methods are *inherited* unless they are *overridden*.
`@TestInstance`	Used to configure the test instance lifecycle for the annotated test class. Such annotations are *inherited*.
`@TestTemplate`	Denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such methods are *inherited* unless they are *overridden*.
`@DisplayName`	Declares a custom display name for the test class or test method. Such annotations are not *inherited*.
`@BeforeEach`	Denotes that the annotated method should be executed *before* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@Before`. Such methods are *inherited* unless they are *overridden*.
`@AfterEach`	Denotes that the annotated method should be executed *after* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, or `@TestFactory` method in the current class; analogous to JUnit 4's `@After`. Such methods are *inherited* unless they are *overridden*.

Annotation	Description
`@BeforeAll`	Denotes that the annotated method should be executed *before* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@BeforeClass`. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be `static` (unless the "per-class" test instance lifecycle is used).
`@AfterAll`	Denotes that the annotated method should be executed *after* **all** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and `@TestFactory` methods in the current class; analogous to JUnit 4's `@AfterClass`. Such methods are *inherited* (unless they are *hidden* or *overridden*) and must be `static` (unless the "per-class" test instance lifecycle is used).
`@Nested`	Denotes that the annotated class is a nested, non-static test class. `@BeforeAll` and `@AfterAll` methods cannot be used directly in a `@Nested` test class unless the "per-class" test instance lifecycle is used. Such annotations are not *inherited*.
`@Tag`	Used to declare *tags* for filtering tests, either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are *inherited* at the class level but not at the method level.
`@Disabled`	Used to *disable* a test class or test method; analogous to JUnit 4's `@Ignore`. Such annotations are not *inherited*.
`@ExtendWith`	Used to register custom extensions. Such annotations are *inherited*.

3. 编写单元测试

3.1. 基本的单元测试类和方法

```java
import org.junit.jupiter.api.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class Junit5StandardTests {

    private static final Logger LOGGER = LoggerFactory.getLogger

    @BeforeAll
    static void beforeAll() {
        LOGGER.info("call beforeAll()");
    }

    @BeforeEach
    void beforeEach() {
        LOGGER.info("call beforeEach()");
    }

    @Test
    void succeedingTest() {
        LOGGER.info("call succeedingTest()");
    }

    @Test
    void failingTest() {
        LOGGER.info("call failingTest()");
        // fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        LOGGER.info("call skippedTest()");
        // not executed
    }

    @AfterEach
    void afterEach() {
        LOGGER.info("call afterEach()");
    }

    @AfterAll
    static void afterAll() {
        LOGGER.info("call afterAll()");
    }
}
```

3.2. 定制测试类和方法的显示名称

支持普通字符、特殊符号、emoji

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class JunitDisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() { }

    @Test
    @DisplayName("╯°□°）╯")
    void testWithDisplayNameContainingSpecialCharacters() { }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() { }
}
```

3.3. 断言（Assertions）

```java
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.*;

class AssertionsDemo {

    private static Person person;

    @BeforeAll
    public static void beforeAll() {
        person = new Person("John", "Doe");
    }

    @Test
    void standardAssertions() {
        assertEquals(2, 2);
        assertEquals(4, 4, "The optional assertion message is no
        assertTrue('a' < 'b', () -> "Assertion messages can be l
            + "to avoid constructing complex messages unnecessar
    }

    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, a
        // failures will be reported together.
        assertAll("person", () -> assertEquals("John", person.ge
            () -> assertEquals("Doe", person.getLastName()));
    }

    @Test
    void dependentAssertions() {
        // Within a code block, if an assertion fails the
        // subsequent code in the same block will be skipped.
        assertAll("properties", () -> {
            String firstName = person.getFirstName();
            assertNotNull(firstName);

            // Executed only if the previous assertion is valid.
            assertAll("first name", () -> assertTrue(firstName.s
                () -> assertTrue(firstName.endsWith("n")));
        }, () -> {
            // Grouped assertion, so processed independently
            // of results of first name assertions.
            String lastName = person.getLastName();
            assertNotNull(lastName);
```

```
        // Executed only if the previous assertion is valid.
        assertAll("last name", () -> assertTrue(lastName.sta
            () -> assertTrue(lastName.endsWith("e")));
    });
}

@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentExcept
        throw new IllegalArgumentException("a message");
    });
    assertEquals("a message", exception.getMessage());
}

@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}

@Test
void timeoutNotExceededWithResult() {
    // The following assertion succeeds, and returns the sup
    String actualResult = assertTimeout(ofMinutes(2), () ->
        return "a result";
    });
    assertEquals("a result", actualResult);
}

@Test
void timeoutNotExceededWithMethod() {
    // The following assertion invokes a method reference an
    String actualGreeting = assertTimeout(ofMinutes(2), Asse
    assertEquals("Hello, World!", actualGreeting);
}

@Test
void timeoutExceeded() {
    // The following assertion fails with an error message s
    // execution exceeded timeout of 10 ms by 91 ms
    assertTimeout(ofMillis(10), () -> {
        // Simulate task that takes more than 10 ms.
        Thread.sleep(100);
    });
}
```

```
    @Test
    void timeoutExceededWithPreemptiveTermination() {
        // The following assertion fails with an error message s
        // execution timed out after 10 ms
        assertTimeoutPreemptively(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

    private static String greeting() {
        return "Hello, World!";
    }

}
```

3.4. 假想（Assumptions）

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    @Test
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation")
        // remainder of test
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI serve
                assertEquals(2, 2);
            });

        // perform these assertions in all environments
        assertEquals("a string", "a string");
    }

}
```

3.5. 禁用

禁用单元测试类示例：

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled
class DisabledClassDemo {
    @Test
    void testWillBeSkipped() {
    }
}
```

禁用单元测试方法示例：

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTestsDemo {

    @Disabled
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }
}
```

3.6. 测试条件

操作系统条件

```java
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
    // ...
}

@TestOnMac
void testOnMac() {
    // ...
}

@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Test
@EnabledOnOs(MAC)
@interface TestOnMac {
}
```

Java 运行时版本条件

```
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}

@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}

@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {
    // ...
}
```

系统属性条件

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test
@DisabledIfSystemProperty(named = "ci-server", matches = "true")
void notOnCiServer() {
    // ...
}
```

3.7. 嵌套测试

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.EmptyStackException;
import java.util.Stack;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, () -> stack.
        }

        @Test
        @DisplayName("throws EmptyStackException when peeked")
        void throwsExceptionWhenPeeked() {
```

```
            assertThrows(EmptyStackException.class, () -> stack.
        }

        @Nested
        @DisplayName("after pushing an element")
        class AfterPushing {

            String anElement = "an element";

            @BeforeEach
            void pushAnElement() {
                stack.push(anElement);
            }

            @Test
            @DisplayName("it is no longer empty")
            void isNotEmpty() {
                assertFalse(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when popped and is
            void returnElementWhenPopped() {
                assertEquals(anElement, stack.pop());
                assertTrue(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when peeked but re
            void returnElementWhenPeeked() {
                assertEquals(anElement, stack.peek());
                assertFalse(stack.isEmpty());
            }
        }
    }
}
```

3.8. 重复测试

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.logging.Logger;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.TestInfo;

class RepeatedTestsDemo {

    private Logger logger = // ...

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetition
        int currentRepetition = repetitionInfo.getCurrentRepetit
        int totalRepetitions = repetitionInfo.getTotalRepetition
        String methodName = testInfo.getTestMethod().get().getNa
        logger.info(String.format("About to execute repetition %
            currentRepetition, totalRepetitions, methodName));
    }

    @RepeatedTest(10)
    void repeatedTest() {
        // ...
    }

    @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitio
        assertEquals(5, repetitionInfo.getTotalRepetitions());
    }

    @RepeatedTest(value = 1, name = "{displayName} {currentRepet
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals(testInfo.getDisplayName(), "Repeat! 1/1");
    }

    @RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NA
    @DisplayName("Details...")
    void customDisplayNameWithLongPattern(TestInfo testInfo) {
        assertEquals(testInfo.getDisplayName(), "Details... :: r
    }

    @RepeatedTest(value = 5, name = "Wiederholung {currentRepeti
    void repeatedTestInGerman() {
        // ...
```

```
    }

}
```

3.9. 参数化测试

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I s
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```

4. 引用和引申

- Github
- 官方用户手册
- Javadoc
- 版本声明
- 官方示例

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-21 13:20:28

- Lombok 使用指南
  - 1. Lombok 简介
  - 2. Lombok 安装
  - 3. Lombok 使用
    - 3.1. @Getter and @Setter
    - 3.2. @NonNull
    - 3.3. @ToString
    - 3.4. @EqualsAndHashCode
    - 3.5. @Data
    - 3.6. @Cleanup
    - 3.7. @Synchronized
    - 3.8. @SneakyThrows
    - 3.9. 示例源码
  - 4. 参考资料

Lombok 使用指南

- 1. Lombok 简介
- 2. Lombok 安装
- 3. Lombok 使用
  - 3.1. @Getter and @Setter
  - 3.2. @NonNull
  - 3.3. @ToString
  - 3.4. @EqualsAndHashCode
  - 3.5. @Data
  - 3.6. @Cleanup
  - 3.7. @Synchronized
  - 3.8. @SneakyThrows
  - 3.9. 示例源码
- 4. 参考资料

1. Lombok 简介

Lombok 是一种 Java 实用工具，可用来帮助开发人员消除 Java 的冗长，尤其是对于简单的 Java 对象（POJO）。它通过注释实现这一目的。通过在开发环境中实现 Lombok，开发人员可以节省构建诸如 `hashCode()` 和 `equals()` 、 `getter / setter` 这样的方法以及以往用来分类各种 accessor 和 mutator 的大量时间。

2. Lombok 安装

使 IntelliJ IDEA 支持 Lombok 方式如下：

- **Intellij** 设置支持注解处理
  - 点击 File > Settings > Build > Annotation Processors
  - 勾选 Enable annotation processing
- 安装插件
  - 点击 Settings > Plugins > Browse repositories
  - 查找 Lombok Plugin 并进行安装
  - 重启 IntelliJ IDEA
- 将 **lombok** 添加到 **pom** 文件

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
</dependency>
```

3. Lombok 使用

Lombok 提供注解 API 来修饰指定的类：

3.1. @Getter and @Setter

@Getter and @Setter Lombok 代码：

```
@Getter @Setter private boolean employed = true;
@Setter(AccessLevel.PROTECTED) private String name;
```

等价于 Java 源码：

```
private boolean employed = true;
private String name;

public boolean isEmployed() {
    return employed;
}

public void setEmployed(final boolean employed) {
    this.employed = employed;
}

protected void setName(final String name) {
    this.name = name;
}
```

3.2. @NonNull

@NonNull Lombok 代码：

```
@Getter @Setter @NonNull
private List<Person> members;
```

等价于 Java 源码：

```
@NonNull
private List<Person> members;

public Family(@NonNull final List<Person> members) {
    if (members == null) throw new java.lang.NullPointerExceptio
    this.members = members;
}

@NonNull
public List<Person> getMembers() {
    return members;
}

public void setMembers(@NonNull final List<Person> members) {
    if (members == null) throw new java.lang.NullPointerExceptio
    this.members = members;
}
```

3.3. @ToString

@ToString Lombok 代码：

```
@ToString(callSuper=true,exclude="someExcludedField")
public class Foo extends Bar {
    private boolean someBoolean = true;
    private String someStringField;
    private float someExcludedField;
}
```

等价于 Java 源码：

```
public class Foo extends Bar {
    private boolean someBoolean = true;
    private String someStringField;
    private float someExcludedField;

    @java.lang.Override
    public java.lang.String toString() {
        return "Foo(super=" + super.toString() +
            ", someBoolean=" + someBoolean +
            ", someStringField=" + someStringField + ")";
    }
}
```

3.4. @EqualsAndHashCode

@EqualsAndHashCode Lombok 代码：

```
@EqualsAndHashCode(callSuper=true,exclude={"address","city","sta
public class Person extends SentientBeing {
    enum Gender { Male, Female }

    @NonNull private String name;
    @NonNull private Gender gender;

    private String ssn;
    private String address;
    private String city;
    private String state;
    private String zip;
}
```

等价于 Java 源码：

```
public class Person extends SentientBeing {

    enum Gender {
        /*public static final*/ Male /* = new Gender() */,
        /*public static final*/ Female /* = new Gender() */;
    }
    @NonNull
    private String name;
    @NonNull
    private Gender gender;
    private String ssn;
    private String address;
    private String city;
    private String state;
    private String zip;

    @java.lang.Override
    public boolean equals(final java.lang.Object o) {
        if (o == this) return true;
        if (o == null) return false;
        if (o.getClass() != this.getClass()) return false;
        if (!super.equals(o)) return false;
        final Person other = (Person)o;
        if (this.name == null ? other.name != null : !this.name.
        if (this.gender == null ? other.gender != null : !this.g
        if (this.ssn == null ? other.ssn != null : !this.ssn.equ
        return true;
    }

    @java.lang.Override
    public int hashCode() {
        final int PRIME = 31;
        int result = 1;
        result = result * PRIME + super.hashCode();
        result = result * PRIME + (this.name == null ? 0 : this.
        result = result * PRIME + (this.gender == null ? 0 : thi
        result = result * PRIME + (this.ssn == null ? 0 : this.s
        return result;
    }
}
```

3.5. @Data

@Data Lombok 代码：

```java
@Data(staticConstructor="of")
public class Company {
    private final Person founder;
    private String name;
    private List<Person> employees;
}
```

等价于 Java 源码：

```
public class Company {
    private final Person founder;
    private String name;
    private List<Person> employees;

    private Company(final Person founder) {
        this.founder = founder;
    }

    public static Company of(final Person founder) {
        return new Company(founder);
    }

    public Person getFounder() {
        return founder;
    }

    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }

    public List<Person> getEmployees() {
        return employees;
    }

    public void setEmployees(final List<Person> employees) {
        this.employees = employees;
    }

    @java.lang.Override
    public boolean equals(final java.lang.Object o) {
        if (o == this) return true;
        if (o == null) return false;
        if (o.getClass() != this.getClass()) return false;
        final Company other = (Company)o;
        if (this.founder == null ? other.founder != null : !this
        if (this.name == null ? other.name != null : !this.name.
        if (this.employees == null ? other.employees != null : !
        return true;
    }

    @java.lang.Override
    public int hashCode() {
        final int PRIME = 31;
```

```
        int result = 1;
        result = result * PRIME + (this.founder == null ? 0 : th
        result = result * PRIME + (this.name == null ? 0 : this.
        result = result * PRIME + (this.employees == null ? 0 :
        return result;
    }

    @java.lang.Override
    public java.lang.String toString() {
        return "Company(founder=" + founder + ", name=" + name +
    }
}
```

3.6. @Cleanup

@Cleanup Lombok 代码：

```
public void testCleanUp() {
    try {
        @Cleanup ByteArrayOutputStream baos = new ByteArrayOutpu
        baos.write(new byte[] {'Y','e','s'});
        System.out.println(baos.toString());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

等价于 Java 源码：

```
public void testCleanUp() {
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream()
        try {
            baos.write(new byte[]{'Y', 'e', 's'});
            System.out.println(baos.toString());
        } finally {
            baos.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

3.7. @Synchronized

@Synchronized Lombok 代码：

```
private DateFormat format = new SimpleDateFormat("MM-dd-YYYY");

@Synchronized
public String synchronizedFormat(Date date) {
    return format.format(date);
}
```

等价于 Java 源码：

```
private final java.lang.Object $lock = new java.lang.Object[0];
private DateFormat format = new SimpleDateFormat("MM-dd-YYYY");

public String synchronizedFormat(Date date) {
    synchronized ($lock) {
        return format.format(date);
    }
}
```

3.8. @SneakyThrows

@SneakyThrows Lombok 代码：

```
@SneakyThrows
public void testSneakyThrows() {
    throw new IllegalAccessException();
}
```

等价于 Java 源码：

```
public void testSneakyThrows() {
    try {
        throw new IllegalAccessException();
    } catch (java.lang.Throwable $ex) {
        throw lombok.Lombok.sneakyThrow($ex);
    }
}
```

3.9. 示例源码

> 示例源码：javalib-bean

4. 参考资料

- Lombok 官网
- Lombok Github
- IntelliJ IDEA - Lombok Plugin

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-22 17:07:35

- Mockito 使用指南
  - 预备知识
  - 使用 mock 对象来进行测试
    - 单元测试的目标和挑战
    - 测试类的分类
    - Mock 对象的产生
    - 使用 Mockito 生成 Mock 对象
  - 为自己的项目添加 Mockito 依赖
    - 在 Gradle 添加 Mockito 依赖
    - 在 Maven 添加 Mockito 依赖
    - 在 Eclipse IDE 使用 Mockito
    - 以 OSGi 或者 Eclipse 插件形式添加 Mockito 依赖
  - 使用 Mockito API
    - 静态引用
    - 使用 Mockito 创建和配置 mock 对象
    - 配置 mock
    - 验证 mock 对象方法是否被调用
    - 使用 Spy 封装 java 对象
    - 使用 @InjectMocks 在 Mockito 中进行依赖注入
    - 捕捉参数
    - Mockito 的限制
  - 在 Android 中使用 Mockito
  - 实例：使用 Mockito 写一个 Instrumented Unit Test
    - 创建一个测试的 Android 应用
    - 在 app/build.gradle 文件中添加 Mockito 依赖
    - 创建测试
  - 实例：使用 Mockito 创建一个 mock 对象
    - 目标
    - 创建一个 Twitter API 的例子
    - 模拟 ITweet 的实例
    - 验证方法调用
    - 验证
  - 模拟静态方法
    - 使用 Powermock 来模拟静态方法
    - 用封装的方法代替 Powermock
  - 引用和引申

Mockito 使用指南

> Mockito 是一个针对 Java 的 mock 框架。

预备知识

如果需要往下学习，你需要先理解 Junit 框架中的单元测试。

如果你不熟悉 JUnit，请看 Junit 教程

使用 mock 对象来进行测试

单元测试的目标和挑战

单元测试的思路是在不涉及依赖关系的情况下测试代码（隔离性），所以测试代码与其他类或者系统的关系应该尽量被消除。一个可行的消除方法是替换掉依赖类（测试替换），也就是说我们可以使用替身来替换掉真正的依赖对象。

测试类的分类

- **dummy object** 做为参数传递给方法但是绝对不会被使用。譬如说，这种测试类内部的方法不会被调用，或者是用来填充某个方法的参数。
- **Fake** 是真正接口或抽象类的实现体，但给对象内部实现很简单。譬如说，它存在内存中而不是真正的数据库中。（译者注：**Fake** 实现了真正的逻辑，但它的存在只是为了测试，而不适合于用在产品中。）
- **stub** 类是依赖类的部分方法实现，而这些方法在你测试类和接口的时候会被用到，也就是说 **stub** 类在测试中会被实例化。**stub** 类会回应任何外部测试的调用。**stub** 类有时候还会记录调用的一些信息。
- **mock object** 是指类或者接口的模拟实现，你可以自定义这个对象中某个方法的输出结果。

测试替代技术能够在测试中模拟测试类以外对象。因此你可以验证测试类是否响应正常。譬如说，你可以验证在 Mock 对象的某一个方法是否被调用。这可以确保隔离了外部依赖的干扰只测试测试类。

我们选择 Mock 对象的原因是因为 Mock 对象只需要少量代码的配置。

Mock 对象的产生

你可以手动创建一个 Mock 对象或者使用 Mock 框架来模拟这些类，Mock 框架允许你在运行时创建 Mock 对象并且定义它的行为。

一个典型的例子是把 Mock 对象模拟成数据的提供者。在正式的生产环境中它会被实现用来连接数据源。但是我们在测试的时候 Mock 对象将会模拟成数据提供者来确保我们的测试环境始终是相同的。

Mock 对象可以被提供来进行测试。因此，我们测试的类应该避免任何外部数据的强依赖。

通过 Mock 对象或者 Mock 框架，我们可以测试代码中期望的行为。譬如说，验证只有某个存在 Mock 对象的方法是否被调用了。

使用 Mockito 生成 Mock 对象

Mockito 是一个流行 mock 框架，可以和 JUnit 结合起来使用。Mockito 允许你创建和配置 mock 对象。使用 Mockito 可以明显的简化对外部依赖的测试类的开发。

一般使用 Mockito 需要执行下面三步

1. 模拟并替换测试代码中外部依赖
2. 执行测试代码
3. 验证测试代码是否被正确的执行

为自己的项目添加 Mockito 依赖

在 Gradle 添加 Mockito 依赖

如果你的项目使用 Gradle 构建，将下面代码加入 Gradle 的构建文件中为自己项目添加 Mockito 依赖

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:2.0.57-beta
```

在 Maven 添加 Mockito 依赖

需要在 Maven 声明依赖，您可以在 http://search.maven.org 网站中搜索 `g:"org.mockito", a:"mockito-core"` 来得到具体的声明方式。

在 Eclipse IDE 使用 Mockito

Eclipse IDE 支持 Gradle 和 Maven 两种构建工具，所以在 Eclipse IDE 添加依赖取决你使用的是哪一个构建工具。

以 OSGi 或者 Eclipse 插件形式添加 Mockito 依赖

在 Eclipse RCP 应用依赖通常可以在 p2 update 上得到。Orbit 是一个很好的第三方仓库，我们可以在里面寻找能在 Eclipse 上使用的应用和插件。

Orbit 仓库地址：http://download.eclipse.org/tools/orbit/downloads

使用 Mockito API

静态引用

如果在代码中静态引用了 `org.mockito.Mockito.*;` ，那你你就可以直接调用静态方法和静态变量而不用创建对象，譬如直接调用 mock() 方法。

使用 Mockito 创建和配置 mock 对象

除了上面所说的使用 mock() 静态方法外，Mockito 还支持通过 `@Mock` 注解的方式来创建 mock 对象。

如果你使用注解，那么必须要实例化 mock 对象。Mockito 在遇到使用注解的字段的时候，会调用 `MockitoAnnotations.initMocks(this)` 来初始化该 mock 对象。另外也可以通过使用 `@RunWith(MockitoJUnitRunner.class)` 来达到相同的效果。

通过下面的例子我们可以了解到使用 `@Mock` 的方法和 `MockitoRule` 规则。

```
import static org.mockito.Mockito.*;

public class MockitoTest  {

        @Mock
        MyDatabase databaseMock; (1)

        @Rule public MockitoRule mockitoRule = MockitoJUnit.rule

        @Test
        public void testQuery()  {
                ClassToTest t  = new ClassToTest(databaseMock);
                boolean check = t.query("* from t"); (4)
                assertTrue(check); (5)
                verify(databaseMock).query("* from t"); (6)
        }
}
```

1. 告诉 Mockito 模拟 databaseMock 实例
2. Mockito 通过 @mock 注解创建 mock 对象
3. 使用已经创建的 mock 初始化这个类
4. 在测试环境下，执行测试类中的代码
5. 使用断言确保调用的方法返回值为 true
6. 验证 query 方法是否被 `MyDatabase` 的 mock 对象调用

配置 mock

当我们需要配置某个方法的返回值的时候，Mockito 提供了链式的 API 供我们方便的调用

`when(….).thenReturn(….)` 可以被用来定义当条件满足时函数的返回值，如果你需要定义多个返回值，可以多次定义。当你多次调用函数的时候，Mockito 会根据你定义的先后顺序来返回返回值。Mocks 还可以根据传入参数的不同来定义不同的返回值。譬如说你的函数可以将 `anyString` 或者 `anyInt` 作为输入参数，然后定义其特定的放回值。

```java
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

@Test
public void test1()  {
        //  创建 mock
        MyClass test = Mockito.mock(MyClass.class);

        // 自定义 getUniqueId() 的返回值
        when(test.getUniqueId()).thenReturn(43);

        // 在测试中使用mock对象
        assertEquals(test.getUniqueId(), 43);
}

// 返回多个值
@Test
public void testMoreThanOneReturnValue()  {
        Iterator i= mock(Iterator.class);
        when(i.next()).thenReturn("Mockito").thenReturn("rocks")
        String result=i.next()+" "+i.next();
        // 断言
        assertEquals("Mockito rocks", result);
}

// 如何根据输入来返回值
@Test
public void testReturnValueDependentOnMethodParameter()  {
        Comparable c= mock(Comparable.class);
        when(c.compareTo("Mockito")).thenReturn(1);
        when(c.compareTo("Eclipse")).thenReturn(2);
        // 断言
        assertEquals(1,c.compareTo("Mockito"));
}

// 如何让返回值不依赖于输入
@Test
public void testReturnValueInDependentOnMethodParameter()  {
        Comparable c= mock(Comparable.class);
        when(c.compareTo(anyInt())).thenReturn(-1);
        // 断言
        assertEquals(-1 ,c.compareTo(9));
}

// 根据参数类型来返回值
@Test
public void testReturnValueInDependentOnMethodParameter()  {
        Comparable c= mock(Comparable.class);
```

```
        when(c.compareTo(isA(Todo.class))).thenReturn(0);
        // 断言
        Todo todo = new Todo(5);
        assertEquals(todo ,c.compareTo(new Todo(1)));
}
```

对于无返回值的函数，我们可以使用 `doReturn(…).when(…).methodCall` 来获得类似的效果。例如我们想在调用某些无返回值函数的时候抛出异常，那么可以使用 `doThrow` 方法。如下面代码片段所示

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

// 下面测试用例描述了如何使用doThrow()方法

@Test(expected=IOException.class)
public void testForIOException() {
        // 创建并配置 mock 对象
        OutputStream mockStream = mock(OutputStream.class);
        doThrow(new IOException()).when(mockStream).close();

        // 使用 mock
        OutputStreamWriter streamWriter= new OutputStreamWriter(
        streamWriter.close();
}
```

验证 mock 对象方法是否被调用

Mockito 会跟踪 mock 对象里面所有的方法和变量。所以我们可以用来验证函数在传入特定参数的时候是否被调用。这种方式的测试称行为测试，行为测试并不会检查函数的返回值，而是检查在传入正确参数时候函数是否被调用。

```
import static org.mockito.Mockito.*;

@Test
public void testVerify()  {
        // 创建并配置 mock 对象
        MyClass test = Mockito.mock(MyClass.class);
        when(test.getUniqueId()).thenReturn(43);

        // 调用mock对象里面的方法并传入参数为12
        test.testing(12);
        test.getUniqueId();
        test.getUniqueId();

        // 查看在传入参数为12的时候方法是否被调用
        verify(test).testing(Matchers.eq(12));

        // 方法是否被调用两次
        verify(test, times(2)).getUniqueId();

        // 其他用来验证函数是否被调用的方法
        verify(mock, never()).someMethod("never called");
        verify(mock, atLeastOnce()).someMethod("called at least
        verify(mock, atLeast(2)).someMethod("called at least twi
        verify(mock, times(5)).someMethod("called five times");
        verify(mock, atMost(3)).someMethod("called at most 3 tim
}
```

使用 Spy 封装 java 对象

@Spy 或者 `spy()` 方法可以被用来封装 java 对象。被封装后，除非特殊声明（打桩 *stub*），否则都会真正的调用对象里面的每一个方法

```
import static org.mockito.Mockito.*;

// Lets mock a LinkedList
List list = new LinkedList();
List spy = spy(list);

// 可用 doReturn() 来打桩
doReturn("foo").when(spy).get(0);

// 下面代码不生效
// 真正的方法会被调用
// 将会抛出 IndexOutOfBoundsException 的异常，因为 List 为空
when(spy.get(0)).thenReturn("foo");
```

方法 `verifyNoMoreInteractions()` 允许你检查没有其他的方法被调用了。

使用 @InjectMocks 在 Mockito 中进行依赖注入

我们也可以使用 `@InjectMocks` 注解来创建对象，它会根据类型来注入对象里面的成员方法和变量。假定我们有 ArticleManager 类

```
public class ArticleManager {
    private User user;
    private ArticleDatabase database;

    ArticleManager(User user) {
     this.user = user;
    }

    void setDatabase(ArticleDatabase database) { }
}
```

这个类会被 Mockito 构造，而类的成员方法和变量都会被 mock 对象所代替，正如下面的代码片段所示：

```
@RunWith(MockitoJUnitRunner.class)
public class ArticleManagerTest  {

       @Mock ArticleCalculator calculator;
       @Mock ArticleDatabase database;
       @Most User user;

       @Spy private UserProvider userProvider = new ConsumerUser

       @InjectMocks private ArticleManager manager; (1)

       @Test public void shouldDoSomething() {
               // 假定 ArticleManager 有一个叫 initialize() 的方法
               // 使用 ArticleListener 来调用 addListener 方法
               manager.initialize();

               // 验证 addListener 方法被调用
               verify(database).addListener(any(ArticleListener.
       }
}
```

1. 创建 ArticleManager 实例并注入 Mock 对象

更多的详情可以查看
http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html

捕捉参数

`ArgumentCaptor` 类允许我们在 verification 期间访问方法的参数。得到方法的参数后我们可以使用它进行测试。

```
import static org.hamcrest.Matchers.hasItem;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

import java.util.Arrays;
import java.util.List;

import org.junit.Rule;
import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.junit.MockitoJUnit;
import org.mockito.junit.MockitoRule;

public class MockitoTests {
    @Rule
    public MockitoRule rule = MockitoJUnit.rule();

    @Captor
    private ArgumentCaptor<List<String>> captor;

    @Test
    public final void shouldContainCertainListItem() {
        List<String> asList = Arrays.asList("someElement_test",
        final List<String> mockedList = mock(List.class);
        mockedList.addAll(asList);

        verify(mockedList).addAll(captor.capture());
        final List<String> capturedArgument = captor.getValue();
        assertThat(capturedArgument, hasItem("someElement"));
    }
}
```

Mockito 的限制

Mockito 当然也有一定的限制。而下面三种数据类型则不能够被测试

- final classes
- anonymous classes
- primitive types

在 Android 中使用 Mockito

在 Android 中的 Gradle 构建文件中加入 Mockito 依赖后就可以直接使用 Mockito 了。若想使用 Android Instrumented tests 的话，还需要添加 dexmaker 和 dexmaker-mockito 依赖到 Gradle 的构建文件中。（需要 Mockito 1.9.5 版本以上）

```
dependencies {
    testCompile 'junit:junit:4.12'
    // Mockito unit test 的依赖
    testCompile 'org.mockito:mockito-core:1.+'
    // Mockito Android instrumentation tests 的依赖
    androidTestCompile 'org.mockito:mockito-core:1.+'
    androidTestCompile "com.google.dexmaker:dexmaker:1.2"
    androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2
}
```

实例：使用 Mockito 写一个 Instrumented Unit Test

创建一个测试的 Android 应用

创建一个包名为 `com.vogella.android.testing.mockito.contextmock` 的 Android 应用，添加一个静态方法，方法里面创建一个包含参数的 Intent，如下代码所示：

```
public static Intent createQuery(Context context, String query,
    // 简单起见，重用MainActivity
    Intent i = new Intent(context, MainActivity.class);
    i.putExtra("QUERY", query);
    i.putExtra("VALUE", value);
    return i;
}
```

在 app/build.gradle 文件中添加 Mockito 依赖

```
dependencies {
    // Mockito 和 JUnit 的依赖
    // instrumentation unit tests on the JVM
    androidTestCompile 'junit:junit:4.12'
    androidTestCompile 'org.mockito:mockito-core:2.0.57-beta'
    androidTestCompile 'com.android.support.test:runner:0.3'
    androidTestCompile "com.google.dexmaker:dexmaker:1.2"
    androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2

    // Mockito 和 JUnit 的依赖
    // tests on the JVM
    testCompile 'junit:junit:4.12'
    testCompile 'org.mockito:mockito-core:1.+'

}
```

创建测试

使用 Mockito 创建一个单元测试来验证在传递正确 extra data 的情况下，intent 是否被触发。

因此我们需要使用 Mockito 来 mock 一个 `Context` 对象，如下代码所示：

```java
package com.vogella.android.testing.mockitocontextmock;

import android.content.Context;
import android.content.Intent;
import android.os.Bundle;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class TextIntentCreation {

    @Test
    public void testIntentShouldBeCreated() {
        Context context = Mockito.mock(Context.class);
        Intent intent = MainActivity.createQuery(context, "query
        assertNotNull(intent);
        Bundle extras = intent.getExtras();
        assertNotNull(extras);
        assertEquals("query", extras.getString("QUERY"));
        assertEquals("value", extras.getString("VALUE"));
    }
}
```

实例：使用 Mockito 创建一个 mock 对象

目标

创建一个 Api，它可以被 Mockito 来模拟并做一些工作

创建一个 Twitter API 的例子

实现 `TwitterClient` 类，它内部使用到了 `ITweet` 的实现。但是 `ITweet` 实例很难得到，譬如说他需要启动一个很复杂的服务来得到。

```
public interface ITweet {

        String getMessage();
}


public class TwitterClient {

        public void sendTweet(ITweet tweet) {
                String message = tweet.getMessage();

                // send the message to Twitter
        }
}
```

模拟 ITweet 的实例

为了能够不启动复杂的服务来得到 `ITweet` ，我们可以使用 Mockito 来模拟得到该实例。

```
@Test
public void testSendingTweet() {
        TwitterClient twitterClient = new TwitterClient();

        ITweet iTweet = mock(ITweet.class);

        when(iTweet.getMessage()).thenReturn("Using mockito is g

        twitterClient.sendTweet(iTweet);
}
```

现在 `TwitterClient` 可以使用 `ITweet` 接口的实现，当调用 `getMessage()` 方法的时候将会打印 "Using Mockito is great" 信息。

验证方法调用

确保 getMessage() 方法至少调用一次。

```
@Test
public void testSendingTweet() {
        TwitterClient twitterClient = new TwitterClient();

        ITweet iTweet = mock(ITweet.class);

        when(iTweet.getMessage()).thenReturn("Using mockito is g

        twitterClient.sendTweet(iTweet);

        verify(iTweet, atLeastOnce()).getMessage();
}
```

验证

运行测试，查看代码是否测试通过。

模拟静态方法

使用 Powermock 来模拟静态方法

因为 Mockito 不能够 mock 静态方法，因此我们可以使用 `Powermock` 。

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public final class NetworkReader {
    public static String getLocalHostname() {
        String hostname = "";
        try {
            InetAddress addr = InetAddress.getLocalHost();
            // Get hostname
            hostname = addr.getHostName();
        } catch ( UnknownHostException e ) {
        }
        return hostname;
    }
}
```

我们模拟了 NetworkReader 的依赖，如下代码所示：

```
import org.junit.runner.RunWith;
import org.powermock.core.classloader.annotations.PrepareForTest

@RunWith( PowerMockRunner.class )
@PrepareForTest( NetworkReader.class )
public class MyTest {

// 测试代码

 @Test
public void testSomething() {
    mockStatic( NetworkUtil.class );
    when( NetworkReader.getLocalHostname() ).andReturn( "localho

    // 与 NetworkReader 协作的测试
}
```

用封装的方法代替 Powermock

有时候我们可以在静态方法周围包含非静态的方法来达到和 Powermock 同样的效果。

```
class FooWraper {
      void someMethod() {
           Foo.someStaticMethod()
       }
}
```

引用和引申

- 官网
- Github
- 使用强大的 Mockito 测试框架来测试你的代码

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-29 13:39:46

- Reflections 使用指南
  - 使用
  - ReflectionUtils

Reflections 使用指南

- 使用
- ReflectionUtils

引入

```
<dependency>
    <groupId>org.reflections</groupId>
    <artifactId>reflections</artifactId>
    <version>0.9.11</version>
</dependency>
```

典型应用

```
Reflections reflections = new Reflections("my.project");
Set<Class<? extends SomeType>> subTypes = reflections.getSubType
Set<Class<?>> annotated = reflections.getTypesAnnotatedWith(Some
```

使用

基本上，使用 Reflections 首先使用 urls 和 scanners 对其进行实例化

```
//scan urls that contain 'my.package', include inputs starting w
Reflections reflections = new Reflections("my.package");

//or using ConfigurationBuilder
new Reflections(new ConfigurationBuilder()
     .setUrls(ClasspathHelper.forPackage("my.project.prefix"))
     .setScanners(new SubTypesScanner(),
                  new TypeAnnotationsScanner().filterResultsBy(o
     .filterInputsBy(new FilterBuilder().includePackage("my.proj
     ...);
```

然后，使用方便的查询方法

```
// 子类型扫描
Set<Class<? extends Module>> modules =
    reflections.getSubTypesOf(com.google.inject.Module.class);
// 类型注解扫描
Set<Class<?>> singletons =
    reflections.getTypesAnnotatedWith(javax.inject.Singleton.cla
// 资源扫描
Set<String> properties =
    reflections.getResources(Pattern.compile(".*\\.properties"))
// 方法注解扫描
Set<Method> resources =
    reflections.getMethodsAnnotatedWith(javax.ws.rs.Path.class);
Set<Constructor> injectables =
    reflections.getConstructorsAnnotatedWith(javax.inject.Inject
// 字段注解扫描
Set<Field> ids =
    reflections.getFieldsAnnotatedWith(javax.persistence.Id.clas
// 方法参数扫描
Set<Method> someMethods =
    reflections.getMethodsMatchParams(long.class, int.class);
Set<Method> voidMethods =
    reflections.getMethodsReturn(void.class);
Set<Method> pathParamMethods =
    reflections.getMethodsWithAnyParamAnnotated(PathParam.class)
// 方法参数名扫描
List<String> parameterNames =
    reflections.getMethodParamNames(Method.class)
// 方法使用扫描
Set<Member> usages =
    reflections.getMethodUsages(Method.class)
```

说明：

- 如果未配置扫描程序，则将使用默认值 - SubTypesScanner 和 TypeAnnotationsScanner。
- 还可以配置类加载器，它将用于从名称中解析运行时类。
- Reflection 默认情况下会扩展超类型。 这解决了传输 URL 不被扫描的一些问题。

ReflectionUtils

```
import static org.reflections.ReflectionUtils.*;

Set<Method> getters = getAllMethods(someClass,
  withModifier(Modifier.PUBLIC), withPrefix("get"), withParamete

//or
Set<Method> listMethodsFromCollectionToBoolean =
  getAllMethods(List.class,
    withParametersAssignableTo(Collection.class), withReturnType

Set<Field> fields = getAllFields(SomeClass.class, withAnnotation
```

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 15:50:03

- Thumbnailator 使用指南
  - 简介
  - 核心 API
    - Thumbnails
    - Thumbnails.Builder
    - 工作流
  - 实战
    - 安装
    - 图片缩放
    - 图片旋转
    - 加水印
    - 批量处理图片
  - 参考

Thumbnailator 使用指南

- 简介
- 核心 API
  - Thumbnails
  - Thumbnails.Builder
  - 工作流
- 实战
  - 安装
  - 图片缩放
  - 图片旋转
  - 加水印
  - 批量处理图片
- 参考

简介

`Thumbnailator` 是一个开源的 **Java** 项目，它提供了非常简单的 API 来对图片进行缩放、旋转以及加水印的处理。

有多简单呢？简单到一行代码就可以完成图片处理。形式如下：

```
Thumbnails.of(new File("path/to/directory").listFiles())
    .size(640, 480)
    .outputFormat("jpg")
    .toFiles(Rename.PREFIX_DOT_THUMBNAIL);
```

当然，Thumbnailator 还有一些使用细节，下面我会一一道来。

核心 API

Thumbnails

`Thumbnails` 是使用 Thumbnailator 创建缩略图的主入口。

它提供了一组初始化 `Thumbnails.Builder` 的接口。

先看下这组接口的声明：

```
// 可变长度参数列表
public static Builder<File> of(String... files) {...}
public static Builder<File> of(File... files) {...}
public static Builder<URL> of(URL... urls) {...}
public static Builder<? extends InputStream> of(InputStream... i
public static Builder<BufferedImage> of(BufferedImage... images)
// 迭代器（所有实现 Iterable 接口的 Java 对象都可以，当然也包括 List
public static Builder<File> fromFilenames(Iterable<String> files
public static Builder<File> fromFiles(Iterable<File> files) {...
public static Builder<URL> fromURLs(Iterable<URL> urls) {...}
public static Builder<InputStream> fromInputStreams(Iterable<? e
public static Builder<BufferedImage> fromImages(Iterable<Buffere
```

很显然，**Thumbnails** 允许通过传入文件名、文件、网络图的**URL**、图片流、图片缓存多种方式来初始化构造器。

因此，你可以根据实际需求来灵活的选择图片的输入方式。

需要注意一点：如果输入是多个对象（无论你是直接输入容器对象或使用可变参数方式传入多个对象），则输出也必须选用输出多个对象的方式，否则会报异常。

Thumbnails.Builder

`Thumbnails.Builder` 是 `Thumbnails` 的内部静态类。它用于设置生成缩略图任务的相关参数。

注：`Thumbnails.Builder` 的构造函数是私有函数。所以，它只允许通过 `Thumbnails` 的实例化函数来进行初始化。

设置参数的函数

`Thumbnails.Builder` 提供了一组函数链形式的接口来设置缩放图参数。

以设置大小函数为例：

```
public Builder<T> size(int width, int height)
{
    updateStatus(Properties.SIZE, Status.ALREADY_SET);
    updateStatus(Properties.SCALE, Status.CANNOT_SET);

    validateDimensions(width, height);
    this.width = width;
    this.height = height;

    return this;
}
```

通过返回this指针，使得设置参数函数可以以链式调用的方式来使用，形式如下：

```
Thumbnails.of(new File("original.jpg"))
        .size(160, 160)
        .rotate(90)
        .watermark(Positions.BOTTOM_RIGHT, ImageIO.read(new File
        .outputQuality(0.8)
        .toFile(new File("image-with-watermark.jpg"));
```

好处，不言自明：那就是大大简化了代码。

输出函数

`Thumbnails.Builder` 提供了一组重载函数来输出生成的缩放图。

函数声明如下：

```
// 返回图片缓存
public List<BufferedImage> asBufferedImages() throws IOException
public BufferedImage asBufferedImage() throws IOException {...}
// 返回文件列表
public List<File> asFiles(Iterable<File> iterable) throws IOExce
public List<File> asFiles(Rename rename) throws IOException {...
public List<File> asFiles(File destinationDir, Rename rename) th
// 创建文件
public void toFile(File outFile) throws IOException {...}
public void toFile(String outFilepath) throws IOException {...}
public void toFiles(Iterable<File> iterable) throws IOException
public void toFiles(Rename rename) throws IOException {...}
public void toFiles(File destinationDir, Rename rename) throws I
// 创建输出流
public void toOutputStream(OutputStream os) throws IOException {
public void toOutputStreams(Iterable<? extends OutputStream> ite
```

工作流

Thumbnailator 的工作步骤十分简单，可分为三步：

1. 输入： `Thumbnails` 根据输入初始化构造器—— `Thumbnails.Builder` 。
2. 设置： `Thumbnails.Builder` 设置缩放图片的参数。
3. 输出： `Thumbnails.Builder` 输出图片文件或图片流。

> 更多详情可以参考： [Thumbnailator 官网javadoc]

实战

前文介绍了 Thumbnailator 的核心 API，接下来我们就可以通过实战来看看 Thumbnailator 究竟可以做些什么。

Thumbnailator 生成什么样的图片，是根据设置参数来决定的。

安装

maven项目中引入依赖：

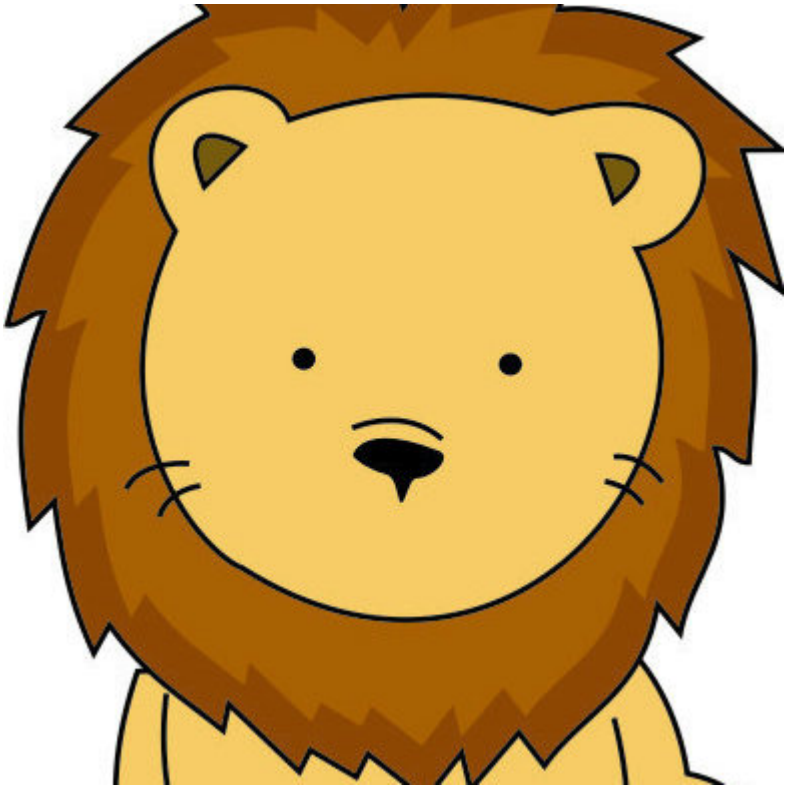
```
<dependency>
  <groupId>net.coobird</groupId>
  <artifactId>thumbnailator</artifactId>
  <version>[0.4, 0.5)</version>
</dependency>
```

图片缩放

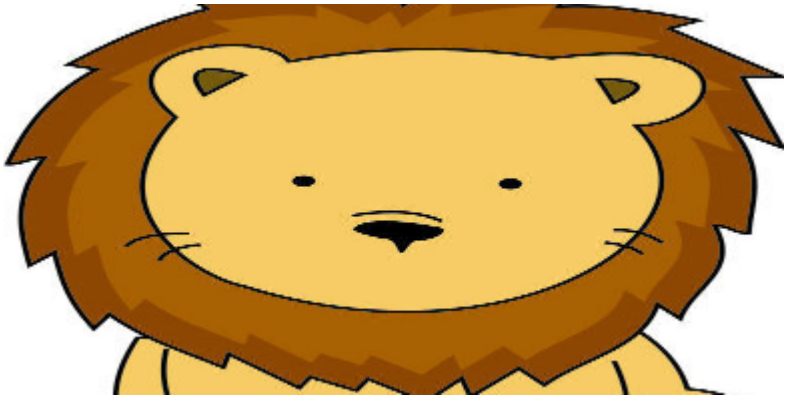
`Thumbnails.Builder` 的 `size` 函数可以设置新图片精确的宽度和高度，也可以用 `scale` 函数设置缩放比例。

```
Thumbnails.of("oldFile.png")
        .size(16, 16)
        .toFile("newFile_16_16.png");

Thumbnails.of("oldFile.png")
        .scale(2.0)
        .toFile("newFile_scale_2.0.png");

Thumbnails.of("oldFile.png")
        .scale(1.0, 0.5)
        .toFile("newFile_scale_1.0_0.5.png");
```

oldFile.png



newFile_scale_1.0_0.5.png

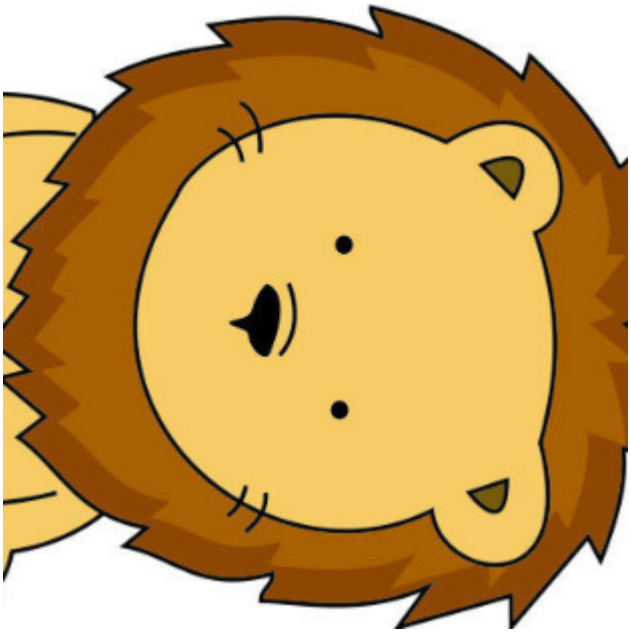


图片旋转

`Thumbnails.Builder` 的 `size` 函数可以设置新图片的旋转角度。

```
Thumbnails.of("oldFile.png")
        .scale(0.8)
        .rotate(90)
        .toFile("newFile_rotate_90.png");

Thumbnails.of("oldFile.png")
        .scale(0.8)
        .rotate(180)
        .toFile("newFile_rotate_180.png");
```

newFile_rotate_90.png



加水印

`Thumbnails.Builder` 的 `watermark` 函数可以为图片添加水印图片。第一个参数是水印的位置；第二个参数是水印图片的缓存数据；第三个参数是透明度。

```
BufferedImage watermarkImage = ImageIO.read(new File("wartermark
Thumbnails.of("oldFile.png")
        .scale(0.8)
        .watermark(Positions.BOTTOM_LEFT, watermarkImage, 0.5f)
        .toFile("newFile_watermark.png");
```

wartermarkFile.png

newFile_watermark.png

批量处理图片

下面以批量给图片加水印来展示一下如何处理多个图片文件。

```
BufferedImage watermarkImage = ImageIO.read(new File("wartermark

File destinationDir = new File("D:\\watermark\\");
Thumbnails.of("oldFile.png", "oldFile2.png")
        .scale(0.8)
        .watermark(Positions.BOTTOM_LEFT, watermarkImage, 0.5f)
        .toFiles(destinationDir, Rename.PREFIX_DOT_THUMBNAIL);
```

> 需要参考完整测试例代码请 点击这里

参考

Thumbnailator 官方示例文档

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 17:21:39

- ZXing 使用指南
  - 简介
  - 实战
    - 安装
    - 生成二维码图片
    - 解析二维码图片
  - 参考

ZXing 使用指南

- 简介
- 实战
  - 安装
  - 生成二维码图片
  - 解析二维码图片
- 参考

简介

`ZXing` 是一个开源 Java 类库用于解析多种格式的 1D/2D 条形码。目标是能够对QR编码、Data Matrix、UPC的1D条形码进行解码。 其提供了多种平台下的客户端包括：J2ME、J2SE和Android。

官网：ZXing github仓库

实战

*本例演示如何在一个非 **android** 的 **Java** 项目中使用 **ZXing** 来生成、解析二维码图片。*

安装

maven项目只需引入依赖：

```
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>core</artifactId>
  <version>3.3.0</version>
</dependency>
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>javase</artifactId>
  <version>3.3.0</version>
</dependency>
```

如果非maven项目，就去官网下载发布版本：[下载地址]

生成二维码图片

ZXing 生成二维码图片有以下步骤：

1. `com.google.zxing.MultiFormatWriter` 根据内容以及图像编码参数生成图像2D矩阵。
2. `com.google.zxing.client.j2se.MatrixToImageWriter` 根据图像矩阵生成图片文件或图片缓存 `BufferedImage` 。

```
public void encode(String content, String filepath) throws Write
    int width = 100;
    int height = 100;
    Map<EncodeHintType, Object> encodeHints = new HashMap<Encode
    encodeHints.put(EncodeHintType.CHARACTER_SET, "UTF-8");
    BitMatrix bitMatrix = new MultiFormatWriter().encode(content
    Path path = FileSystems.getDefault().getPath(filepath);
    MatrixToImageWriter.writeToPath(bitMatrix, "png", path);
}
```

解析二维码图片

ZXing 解析二维码图片有以下步骤：

1. 使用 `javax.imageio.ImageIO` 读取图片文件，并存为一个 `java.awt.image.BufferedImage` 对象。
2. 将 `java.awt.image.BufferedImage` 转换为 ZXing 能识别的 `com.google.zxing.BinaryBitmap` 对象。
3. `com.google.zxing.MultiFormatReader` 根据图像解码参数来解析 `com.google.zxing.BinaryBitmap` 。

```
public String decode(String filepath) throws IOException, NotFou
    BufferedImage bufferedImage = ImageIO.read(new FileInputStre
    LuminanceSource source = new BufferedImageLuminanceSource(bu
    Binarizer binarizer = new HybridBinarizer(source);
    BinaryBitmap bitmap = new BinaryBitmap(binarizer);
    HashMap<DecodeHintType, Object> decodeHints = new HashMap<De
    decodeHints.put(DecodeHintType.CHARACTER_SET, "UTF-8");
    Result result = new MultiFormatReader().decode(bitmap, decod
    return result.getText();
}
```

完整参考示例：测试例代码

以下是一个生成的二维码图片示例：

参考

ZXing github仓库

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 17:21:39

- 构建工具
  - 目录
    - Maven
    - Ant

构建工具

目录

Maven

- Maven 快速指南
- Maven 之 pom.xml 详解
- Maven 之 settings.xml 详解
- Maven 常见问题和最佳实践
- 发布项目到 Maven 中央仓库
- 部署并使用 Nexus 作为 Maven 私服

Ant

- Ant 简易教程

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 15:50:04

- Maven 教程之入门指南
  - 简介
    - Maven 是什么
    - Maven 的生命周期
    - Maven 的标准工程结构
    - Maven 的"约定优于配置"
    - Maven 的版本规范
  - 安装
    - 本地仓储配置
  - 第一个 Maven 工程
    - 在 Intellij 中创建 Maven 工程
    - 在 Eclipse 中创建 Maven 工程
  - 使用指导
    - 如何添加依赖
    - 如何寻找 jar 包
    - 如何使用 Maven 插件(Plugin)
    - 如何一次编译多个工程
    - 常用 Maven 插件
    - 常用 Maven 命令
  - 引用和引申

Maven 教程之入门指南

> 📓 本文已归档到：「blog」

- 简介
  - Maven 是什么
  - Maven 的生命周期
  - Maven 的标准工程结构
  - Maven 的"约定优于配置"
  - Maven 的版本规范
- 安装
  - 本地仓储配置
- 第一个 Maven 工程
  - 在 Intellij 中创建 Maven 工程
  - 在 Eclipse 中创建 Maven 工程
- 使用指导
  - 如何添加依赖
  - 如何寻找 jar 包
  - 如何使用 Maven 插件(Plugin)
  - 如何一次编译多个工程
  - 常用 Maven 插件
  - 常用 Maven 命令
- 引用和引申

简介

Maven 是什么

Maven 是一个项目管理工具。它负责管理项目开发过程中的几乎所有的东西。

- 版本 - maven 有自己的版本定义和规则。
- 构建 - maven 支持许多种的应用程序类型，对于每一种支持的应用程序类型都定义好了一组构建规则和工具集。
- 输出物管理 - maven 可以管理项目构建的产物，并将其加入到用户库中。这个功能可以用于项目组和其他部门之间的交付行为。
- 依赖关系 - maven 对依赖关系的特性进行细致的分析和划分，避免开发过程中的依赖混乱和相互污染行为
- 文档和构建结果 - maven 的 site 命令支持各种文档信息的发布，包括构建过程的各种输出，javadoc，产品文档等。
- 项目关系 - 一个大型的项目通常有几个小项目或者模块组成，用 maven 可以很方便地管理。
- 移植性管理 - maven 可以针对不同的开发场景，输出不同种类的输出结果。

Maven 的生命周期

maven 把项目的构建划分为不同的生命周期(lifecycle)。粗略一点的话，它这个过程(phase)包括：编译、测试、打包、集成测试、验证、部署。maven 中所有的执行动作(goal)都需要指明自己在这个过程中的执行位置，然后 maven 执行的时候，就依照过程的发展依次调用这些 goal 进行各种处理。

这个也是 maven 的一个基本调度机制。一般来说，位置稍后的过程都会依赖于之前的过程。当然，maven 同样提供了配置文件，可以依照用户要求，跳过某些阶段。

Maven 的标准工程结构

Maven 的标准工程结构如下：

```
|-- pom.xml(maven的核心配置文件)
|-- src
|-- main
    |-- java(java源代码目录)
    |-- resources(资源文件目录)
|-- test
    |-- java(单元测试代码目录)
|-- target(输出目录，所有的输出物都存放在这个目录下)
    |-- classes(编译后的class文件存放处)
```

Maven 的"约定优于配置"

所谓的"约定优于配置"，在 maven 中并不是完全不可以修改的，他们只是一些配置的默认值而已。但是除非必要，并不需要去修改那些约定内容。maven 默认的文件存放结构如下：

每一个阶段的任务都知道怎么正确完成自己的工作，比如 compile 任务就知道从 src/main/java 下编译所有的 java 文件，并把它的输出 class 文件存放到 target/classes 中。

对 maven 来说，采用"约定优于配置"的策略可以减少修改配置的工作量，也可以降低学习成本，更重要的是，给项目引入了统一的规范。

Maven 的版本规范

maven 使用如下几个要素来唯一定位某一个输出物：

- **groupId** - 团体、组织的标识符。团体标识的约定是，它以创建这个项目的组织名称的逆向域名(reverse domain name)开头。一般对应着 JAVA 的包的结构。例如 org.apache
- **artifactId** - 单独项目的唯一标识符。比如我们的 tomcat, commons 等。不要在 artifactId 中包含点号(.)。
- **version** - 一个项目的特定版本。
- **packaging** - 项目的类型，默认是 jar，描述了项目打包后的输出。类型为 jar 的项目产生一个 JAR 文件，类型为 war 的项目产生一个 web 应用。

maven 有自己的版本规范，一般是如下定义 `<major version>` 、 `<minor version>` 、 `<incremental version>-<qualifier>` ，比如 1.2.3-beta-01。要说明的是，maven 自己判断版本的算法是 major,minor,incremental 部分用数字比 较，qualifier 部分用字符串比较，所以要小心 alpha-2 和 alpha-15 的比较关系，最好用 alpha-02 的格式。

maven 在版本管理时候可以使用几个特殊的字符串 SNAPSHOT，LATEST，RELEASE。比如"1.0-SNAPSHOT"。各个部分的含义和处理逻辑如下说明：

- **SNAPSHOT** - 这个版本一般用于开发过程中，表示不稳定的版本。
- **LATEST** - 指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个 snapshot 版，具体看哪个时间最后。
- **RELEASE** - 指最后一个发布版。

安装

> 官网下载地址
>
> Linux 环境安装可以使用我写一键安装脚本：https://github.com/dunwu/linux-tutorial/tree/master/codes/linux/ops/service/maven

安装步骤如下：

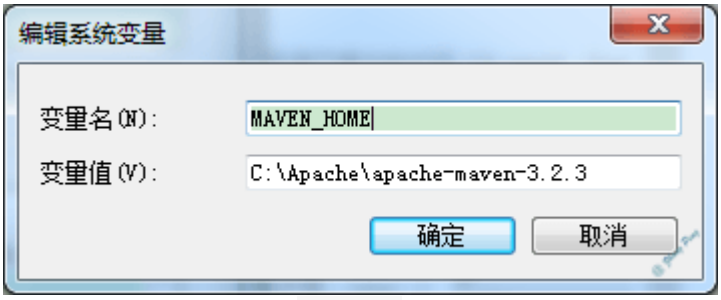
（1）进入官网下载地址：https://maven.apache.org/download.cgi ，选择合适的版本下载并解压到本地。

（2）添加环境变量 MAVEN_HOME，值为 maven 的安装路径

输入 `vi /etc/profile` ，添加环境变量如下：

```
# MAVEN 的根路径
export MAVEN_HOME=/opt/maven/apache-maven-3.5.2
export PATH=\$MAVEN_HOME/bin:\$PATH
```

执行 `source /etc/profile` ，立即生效



（3）检验是否安装成功，执行 `mvn -v` 命令，如果出现 maven 的版本信息，说明配置成功。

```
D:\>mvn --version
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-12T04:58:1
0+08:00)
Maven home: C:\Apache\apache-maven-3.2.3\bin\..
Java version: 1.8.0_25, vendor: Oracle Corporation
Java home: C:\Program Files (x86)\Java\jdk1.8.0_25\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
```

本地仓储配置

从中央仓库下载的 jar 包，都会统一存放到本地仓库中。我们需要配置本地仓库的位置。

打开 maven 安装目录，打开 conf 目录下的 setting.xml 文件。

可以参照下图配置本地仓储位置。

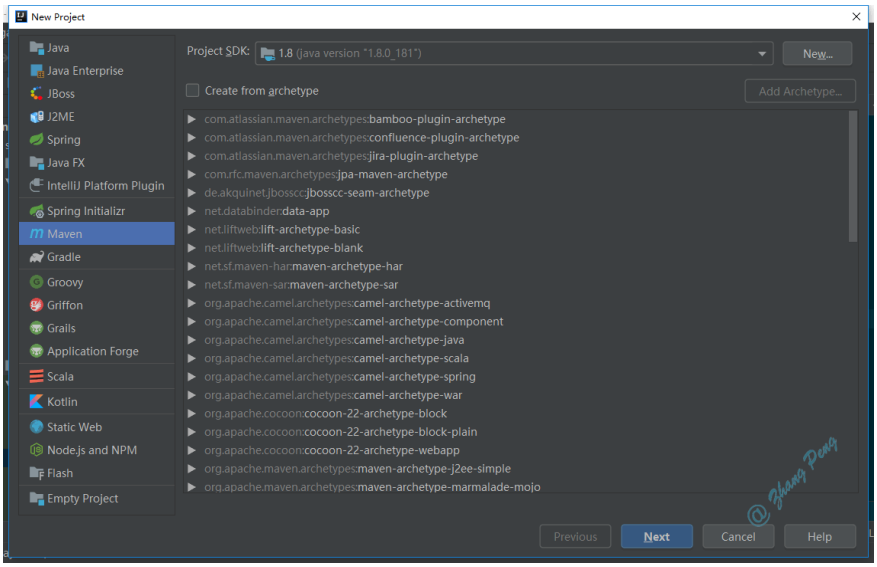
```
 |-->
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
   | The path to the local repository maven will use to store artifacts.
   |
   | Default: ~/.m2/repository
  -->
  <localRepository>D:\soft\maven_repo</localRepository>
```

第一个 Maven 工程

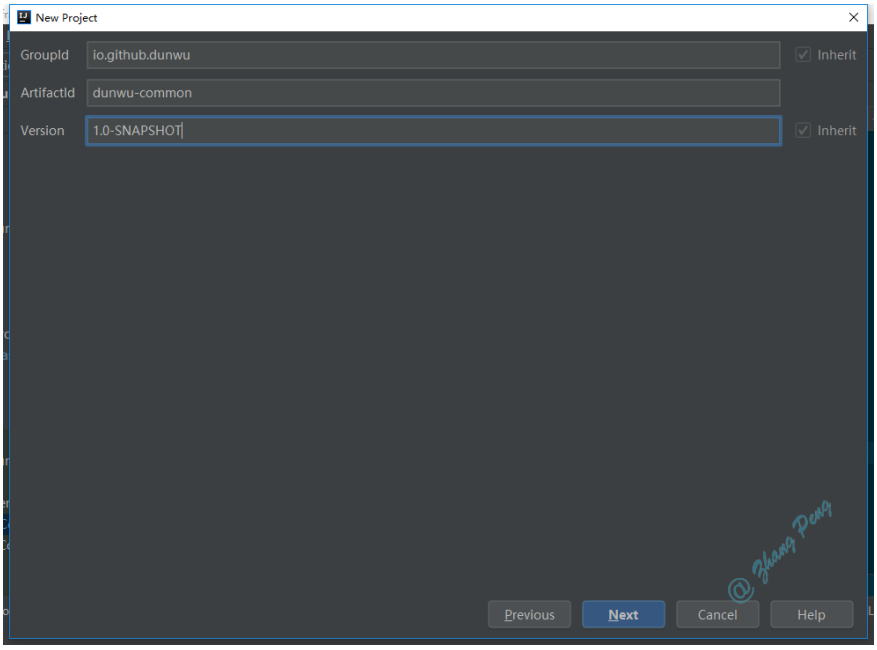
在 Intellij 中创建 Maven 工程

（1）创建 Maven 工程

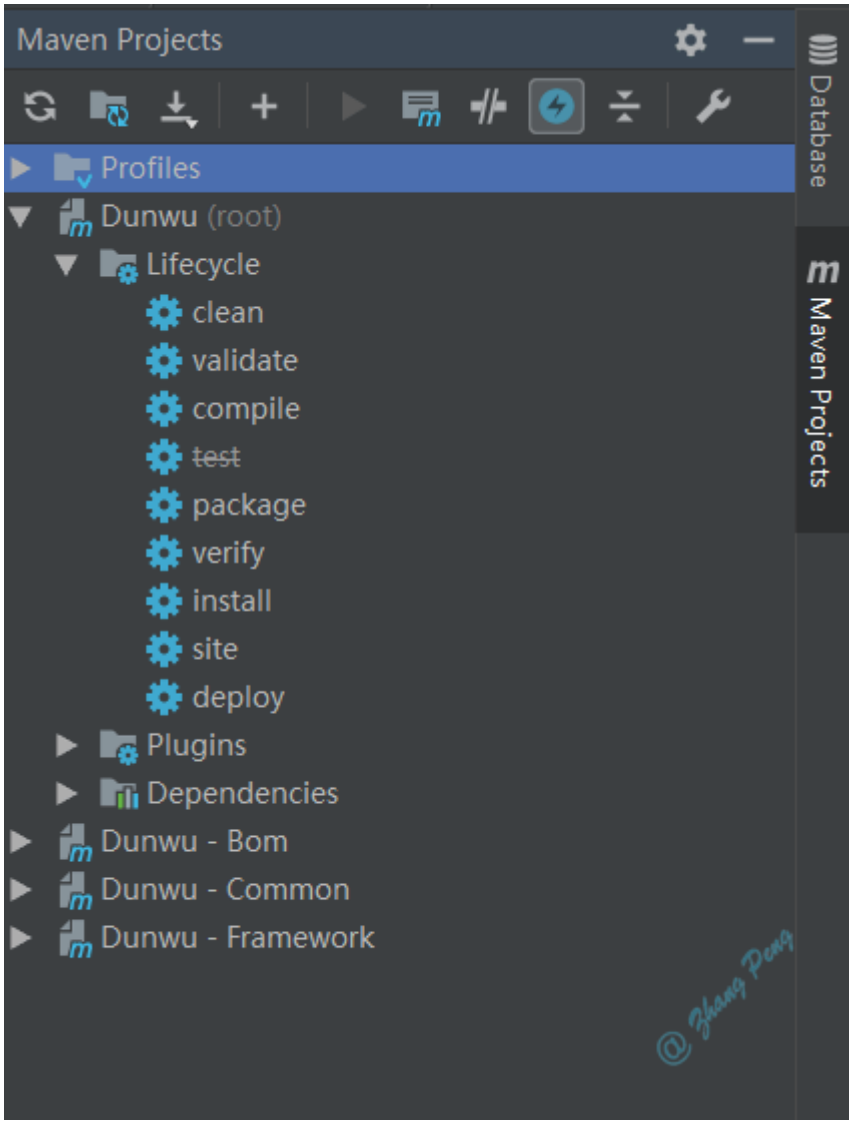
依次点击 File -> New -> Project 打开创建工程对话框，选择 Maven 工程。



（2）输入项目信息



（3）点击 Intellij 侧边栏中的 Maven 工具界面，有几个可以直接使用的 maven 命令，可以帮助你进行构建。



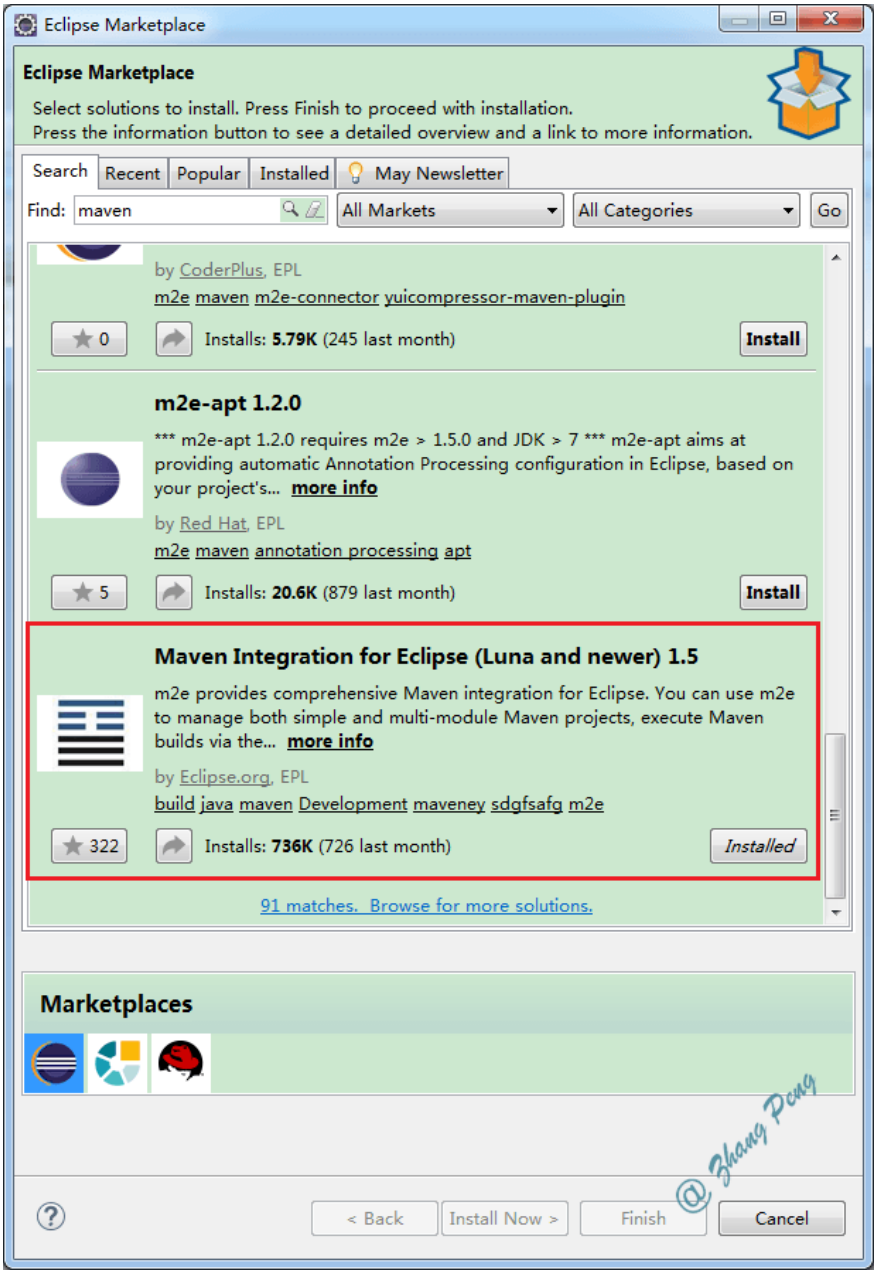
在 Eclipse 中创建 Maven 工程

（1）Maven 插件

在 Eclipse 中创建 Maven 工程，需要安装 Maven 插件。

一般较新版本的 Eclipse 都会带有 Maven 插件，如果你的 Eclipse 中已经有 Maven 插件，可以跳过这一步骤。

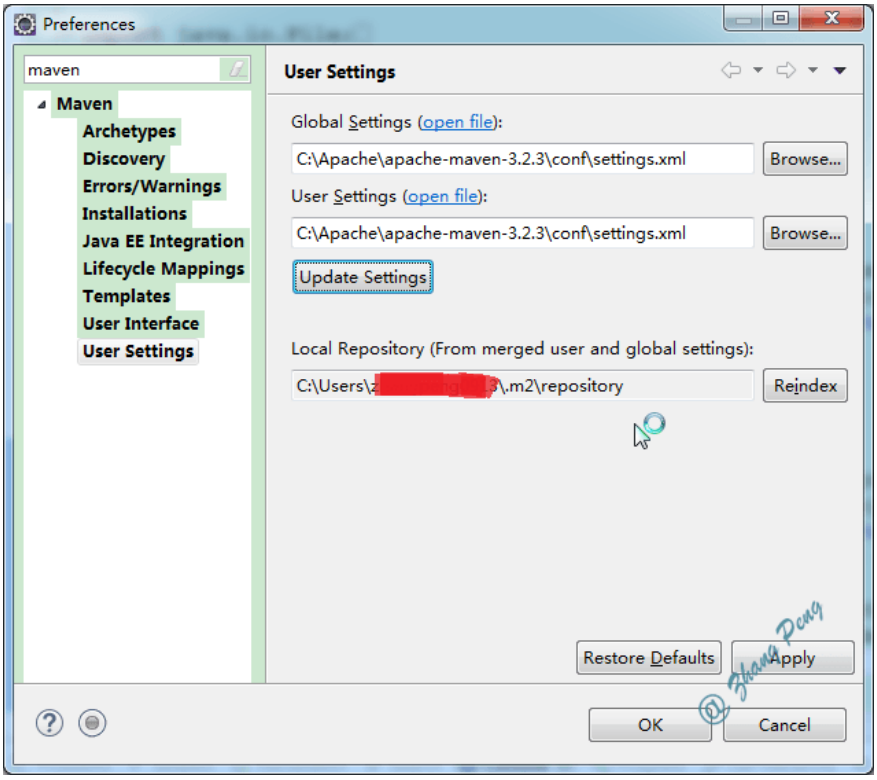
点击 Help -> Eclipse Marketplace，搜索 maven 关键字，选择安装红框对应的 Maven 插件。



（2）Maven 环境配置

点击 Window -> Preferences

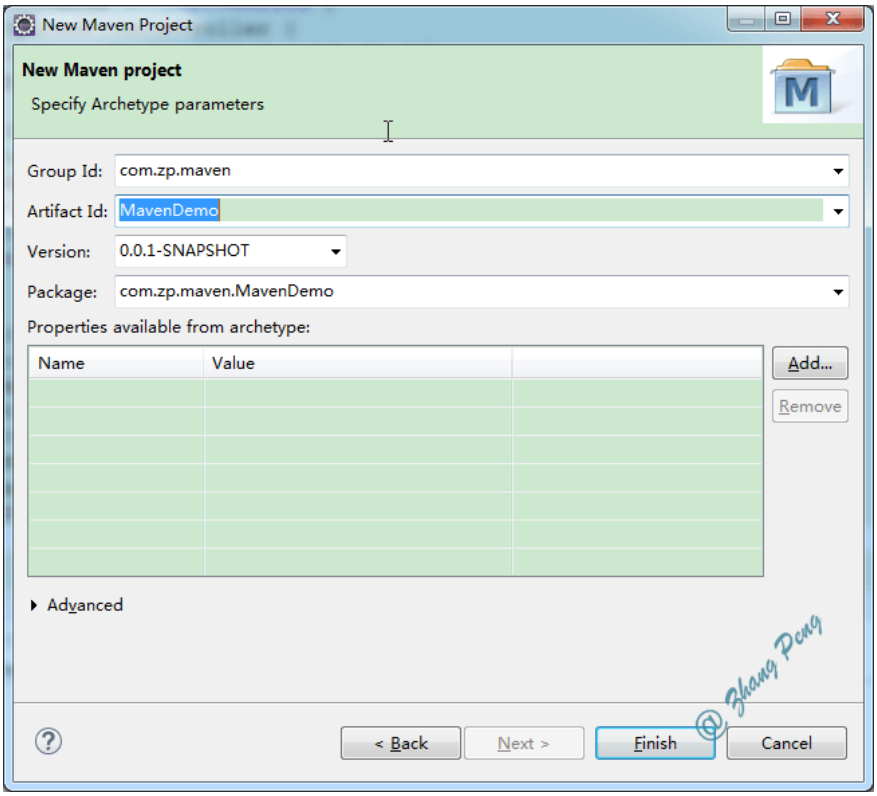
如下图所示，配置 settings.xml 文件的位置



（3）创建 Maven 工程

File -> New -> Maven Project -> Next，在接下来的窗口中会看到一大堆的项目模板，选择合适的模板。

接下来设置项目的参数，如下：



groupId是项目组织唯一的标识符，实际对应 JAVA 的包的结构，是 main 目录里 java 的目录结构。

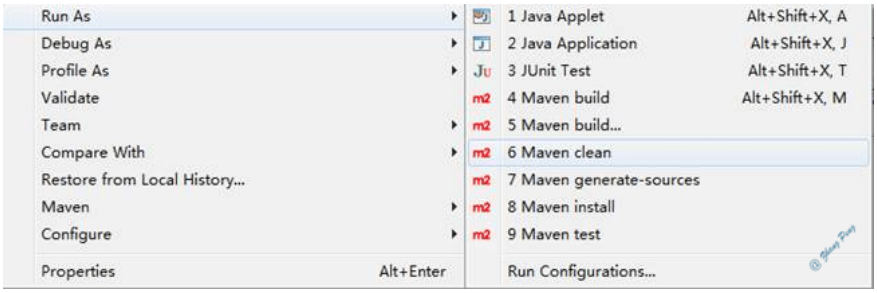
artifactId就是项目的唯一的标识符，实际对应项目的名称，就是项目根目录的名称。

点击 Finish，Eclipse 会创建一个 Maven 工程。

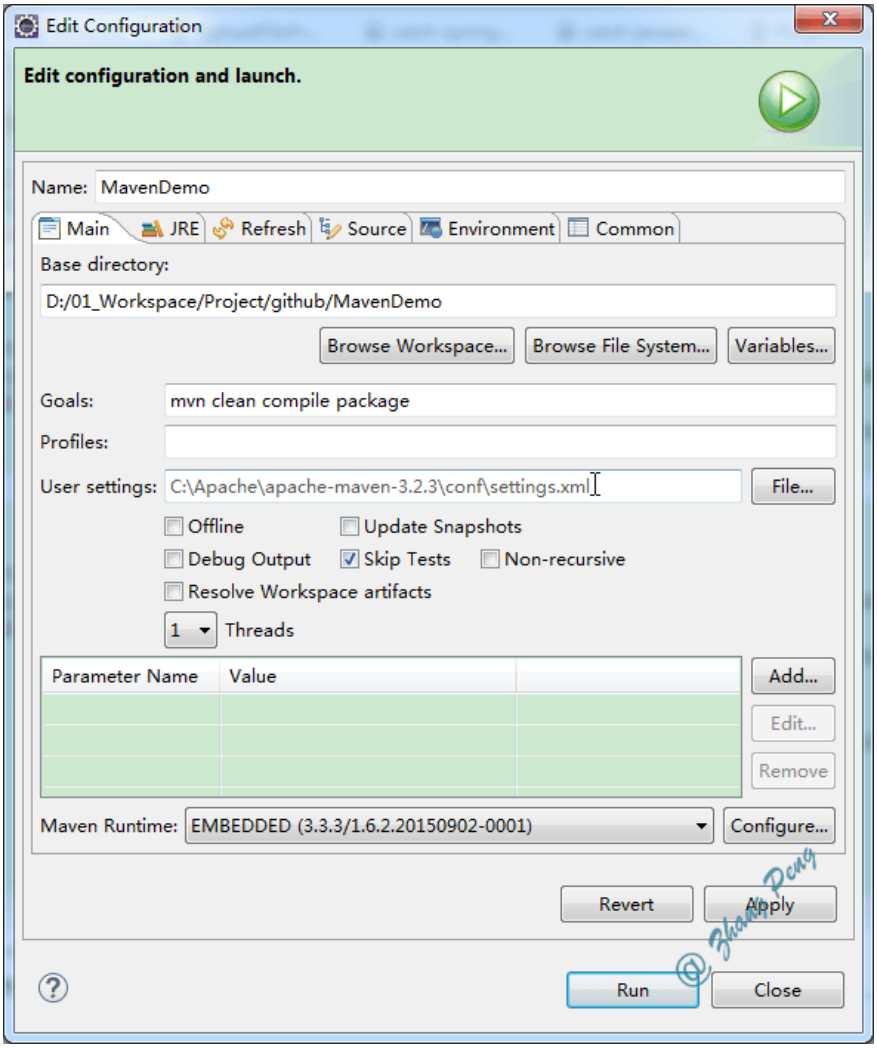
（4）使用 Maven 进行构建

Eclipse 中构建方式

在 Elipse 项目上右击 -> Run As 就能看到很多 Maven 操作。这些操作和 maven 命令是等效的。例如 Maven clean，等同于 mvn clean 命令。



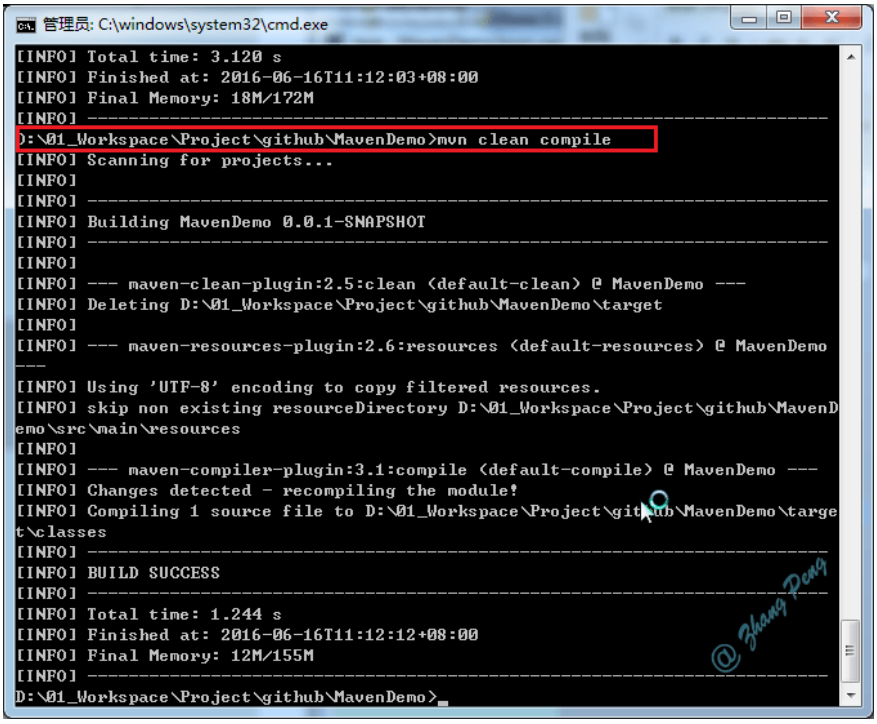
你也可以点击 Maven build，输入组合命令，并保存下来。如下图：



Maven 命令构建方式

当然，你也可以直接使用 maven 命令进行构建。

进入工程所在目录，输入 maven 命令就可以了。



使用指导

如何添加依赖

在 Maven 工程中添加依赖 jar 包，很简单，只要在 POM 文件中引入对应的 `<dependency>` 标签即可。

参考下例：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="htt
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.zp.maven</groupId>
  <artifactId>MavenDemo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>MavenDemo</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEnc
    <junit.version>3.8.1</junit.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

`<dependency>` 标签最常用的四个属性标签：

- `<groupId>` - 项目组织唯一的标识符，实际对应 JAVA 的包的结构。
- `<artifactId>` - 项目唯一的标识符，实际对应项目的名称，就是项目根目录的名称。
- `<version>` - jar 包的版本号。可以直接填版本数字，也可以在 properties 标签中设置属性值。
- `<scope>` - jar 包的作用范围。可以填写 compile、runtime、test、system 和 provided。用来在编译、测试等场景下选择对应的 classpath。

如何寻找 jar 包

可以在 http://mvnrepository.com/ 站点搜寻你想要的 jar 包版本

例如，想要使用 log4j，可以找到需要的版本号，然后拷贝对应的 maven 标签信息，将其添加到 pom .xml 文件中。

如何使用 Maven 插件(Plugin)

要添加 Maven 插件，可以在 pom.xml 文件中添加 `<plugin>` 标签。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

`<configuration>` 标签用来配置插件的一些使用参数。

如何一次编译多个工程

假设要创建一个父 maven 工程，它有两个子工程：my-app 和 my-webapp：

```
+- pom.xml
+- my-app
| +- pom.xml
| +- src
|   +- main
|     +- java
+- my-webapp
| +- pom.xml
| +- src
|   +- main
|     +- webapp
```

app 工程的 pom.xml 如下，重点在于在 modules 中引入两个子 module：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xs
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>my-app</module>
    <module>my-webapp</module>
  </modules>
</project>
```

选择编译 XXX 时，会依次对它的所有 Module 执行相同操作。

常用 Maven 插件

> 更多详情请参考：https://maven.apache.org/plugins/

maven-antrun-plugin

maven-antrun-plugin 能让用户在 Maven 项目中运行 Ant 任务。用户可以直接在该插件的配置以 Ant 的方式编写 Target， 然后交给该插件的 run 目标去执行。在一些由 Ant 往 Maven 迁移的项目中，该插件尤其有用。此外当你发现需要编写一些自定义程度很高的任务，同时又觉 得 Maven 不够灵活时，也可以以 Ant 的方式实现之。maven-antrun-plugin 的 run 目标通常与生命周期绑定运行。

maven-archetype-plugin

Archtype 指项目的骨架，Maven 初学者最开始执行的 Maven 命令可能就是**mvn archetype:generate**，这实际上就是让 maven-archetype-plugin 生成一个很简单的项目骨架，帮助开发者快速上手。可能也有人看到一些文档写了**mvn archetype:create**， 但实际上 create 目标已经被弃用了，取而代之的是 generate 目标，该目标使用交互式的方式提示用户输入必要的信息以创建项目，体验更好。 maven-archetype-plugin 还有一些其他目标帮助用户自己定义项目原型，例如你由一个产品需要交付给很多客户进行二次开发，你就可以为 他们提供一个 Archtype，帮助他们快速上手。

maven-assembly-plugin

maven-assembly-plugin 的用途是将项目打包，该包可能包含了项目的可执行文件、源代码、readme、平台脚本等等。 maven-assembly-plugin 支持各种主流的格式如 zip、tar.gz、jar 和 war 等，具体打包哪些文件是高度可控的，例如用户可以 按文件级别的粒度、文件集级别的粒度、模块级别的粒度、以及依赖级别的粒度控制打包，此外，包含和排除配置也是支持的。maven-assembly- plugin 要求用户使用一个名为 `assembly.xml` 的元数据文件来表述打包，它的 single 目标可以直接在命令行调用，也可以被绑定至生命周期。

maven-dependency-plugin

maven-dependency-plugin 最大的用途是帮助分析项目依赖，**dependency:list**能够列出项目最终解析到的依赖列表，**dependency:tree**能进一步的描绘项目依赖树，**dependency:analyze**可以告诉你项目依赖潜在的问题，如果你有直接使用到的却未声明的依赖，该目标就会发出警告。maven-dependency-plugin 还有很多目标帮助你操作依赖文件，例如**dependency:copy-dependencies**能将项目依赖从本地 Maven 仓库复制到某个特定的文件夹下面。

maven-enforcer-plugin

在一个稍大一点的组织或团队中，你无法保证所有成员都熟悉 Maven，那他们做一些比较愚蠢的事情就会变得很正常，例如给项目引入了外部的 SNAPSHOT 依赖而导致构建不稳定，使用了一个与大家不一致的 Maven 版本而经常抱怨构建出现诡异问题。maven-enforcer- plugin 能够帮助你避免之类问题，它允许你创建一系列规则强制大家遵守，包括设定 Java 版本、设定 Maven 版本、禁止某些依赖、禁止 SNAPSHOT 依赖。只要在一个父 POM 配置规则，然后让大家继承，当规则遭到破坏的时候，Maven 就会报错。除了标准的规则之外，你还可以扩展该插 件，编写自己的规则。maven-enforcer-plugin 的 enforce 目标负责检查规则，它默认绑定到生命周期的 validate 阶段。

maven-help-plugin

maven-help-plugin 是一个小巧的辅助工具，最简单的**help:system**可以打印所有可用的环境变量和 Java 系统属性。**help:effective-pom**和**help:effective-settings**最 为有用，它们分别打印项目的有效 POM 和有效 settings，有效 POM 是指合并了所有父 POM（包括 Super POM）后的 XML，当你不确定 POM 的某些信息从何而来时，就可以查看有效 POM。有效 settings 同理，特别是当你发现自己配置的 settings.xml 没有生效时，就可以用**help:effective-settings**来验证。此外，maven-help-plugin 的 describe 目标可以帮助你描述任何一个 Maven 插件的信息，还有 all-profiles 目标和 active-profiles 目标帮助查看项目的 Profile。

maven-release-plugin

maven-release-plugin 的用途是帮助自动化项目版本发布，它依赖于 POM 中的 SCM 信息。**release:prepare**用来准备版本发布，具体的工作包括检查是否有未提交代码、检查是否有 SNAPSHOT 依赖、升级项目的 SNAPSHOT 版本至 RELEASE 版本、为项目打标签等等。**release:perform**则 是签出标签中的 RELEASE 源码，构建并发布。版本发布是非常琐碎的工作，它涉及了各种检查，而且由于该工作仅仅是偶尔需要，因此手动操作很容易遗漏一 些细节，maven-release-plugin 让该工作变得非常快速简便，不易出错。maven-release-plugin 的各种目标通常直接在 命令行调用，因为版本发布显然不是日常构建生命周期的一部分。

maven-resources-plugin

为了使项目结构更为清晰，Maven 区别对待 Java 代码文件和资源文件，maven-compiler-plugin 用来编译 Java 代码，maven-resources-plugin 则用来处理资源文件。默认的主资源文件目录是 `src/main/resources` ，很多用户会需要添加额外的资源文件目录，这个时候就可以通过配置 maven-resources-plugin 来实现。此外，资源文件过滤也是 Maven 的一大特性，你可以在资源文件中使用*${propertyName}*形式的 Maven 属性，然后配置 maven-resources-plugin 开启对资源文件的过滤，之后就可以针对不同环境通过命令行或者 Profile 传入属性的值，以实现更为灵活的构建。

maven-surefire-plugin

可能是由于历史的原因，Maven 2.3 中用于执行测试的插件不是 maven-test-plugin，而是 maven-surefire-plugin。其实大部分时间内，只要你的测试 类遵循通用的命令约定（以 Test 结尾、以 TestCase 结尾、或者以 Test 开头），就几乎不用知晓该插件的存在。然而在当你想要跳过测试、排除某些 测试类、或者使用一些 TestNG 特性的时候，了解 maven-surefire-plugin 的一些配置选项就很有用了。例如 **mvn test -Dtest=FooTest** 这样一条命令的效果是仅运行 FooTest 测试类，这是通过控制 maven-surefire-plugin 的 test 参数实现的。

build-helper-maven-plugin

Maven 默认只允许指定一个主 Java 代码目录和一个测试 Java 代码目录，虽然这其实是个应当尽量遵守的约定，但偶尔你还是会希望能够指定多个 源码目录（例如为了应对遗留项目），build-helper-maven-plugin 的 add-source 目标就是服务于这个目的，通常它被绑定到 默认生命周期的 generate-sources 阶段以添加额外的源码目录。需要强调的是，这种做法还是不推荐的，因为它破坏了 Maven 的约定，而且可能会遇到其他严格遵守约定的插件工具无法正确识别额外的源码目录。

build-helper-maven-plugin 的另一个非常有用的目标是 attach-artifact，使用该目标你可以以 classifier 的形式选取部分项目文件生成附属构件，并同时 install 到本地仓库，也可以 deploy 到远程仓库。

exec-maven-plugin

exec-maven-plugin 很好理解，顾名思义，它能让你运行任何本地的系统程序，在某些特定情况下，运行一个 Maven 外部的程序可能就是最简单的问题解决方案，这就是**exec:exec**的 用途，当然，该插件还允许你配置相关的程序运行参数。除了 exec 目标之外，exec-maven-plugin 还提供了一个 java 目标，该目标要求你 提供一个 mainClass 参数，然后它能够利用当前项目的依赖作为 classpath，在同一个 JVM 中运行该 mainClass。有时候，为了简单的 演示一个命令行 Java 程序，你可以在 POM 中配置好 exec-maven-plugin 的相关运行参数，然后直接在命令运行**mvn exec:java** 以查看运行效果。

jetty-maven-plugin

在进行 Web 开发的时候，打开浏览器对应用进行手动的测试几乎是无法避免的，这种测试方法通常就是将项目打包成 war 文件，然后部署到 Web 容器 中，再启动容器进行验证，这显然十分耗时。为了帮助开发者节省时间，jetty-maven-plugin 应运而生，它完全兼容 Maven 项目的目录结构，能够周期性地检查源文件，一旦发现变更后自动更新到内置的 Jetty Web 容器中。做一些基本配置后（例如 Web 应用的 contextPath 和自动扫描变更的时间间隔），你只要执行 **mvn jetty:run** ，然后在 IDE 中修改代码，代码经 IDE 自动编译后产生变更，再由 jetty-maven-plugin 侦测到后更新至 Jetty 容器，这时你就可以直接 测试 Web 页面了。需要注意的是，jetty-maven-plugin 并不是宿主于 Apache 或 Codehaus 的官方插件，因此使用的时候需要额外 的配置 `settings.xml` 的 pluginGroups 元素，将 org.mortbay.jetty 这个 pluginGroup 加入。

versions-maven-plugin

很多 Maven 用户遇到过这样一个问题，当项目包含大量模块的时候，为他们集体更新版本就变成一件烦人的事情，到底有没有自动化工具能帮助完成这件 事情呢？（当然你可以使用 sed 之类的文本操作工具，不过不在本文讨论范围）答案是肯定的，versions-maven- plugin 提供了很多目标帮助你管理 Maven 项目的各种版本信息。例如最常用的，命令 **mvn versions:set -DnewVersion=1.1-SNAPSHOT** 就能帮助你把所有模块的版本更新到 1.1-SNAPSHOT。该插件还提供了其他一些很有用的目标，display-dependency- updates 能告诉你项目依赖有哪些可用的更新；类似的 display-plugin-updates 能告诉你可用的插件更新；然后 use- latest-versions 能自动帮你将所有依赖升级到最新版本。最后，如果你对所做的更改满意，则可以使用 **mvn versions:commit** 提交，不满意的话也可以使用 **mvn versions:revert** 进行撤销。

常用 Maven 命令

> 更详细命令说明请参考：
> https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

生命周期	阶段描述
mvn validate	验证项目是否正确，以及所有为了完整构建必要的信息是否可用
mvn generate-sources	生成所有需要包含在编译过程中的源代码
mvn process-sources	处理源代码，比如过滤一些值
mvn generate-resources	生成所有需要包含在打包过程中的资源文件
mvn process-resources	复制并处理资源文件至目标目录，准备打包
mvn compile	编译项目的源代码
mvn process-classes	后处理编译生成的文件，例如对 Java 类进行字节码增强（bytecode enhancement）
mvn generate-test-sources	生成所有包含在测试编译过程中的测试源码
mvn process-test-sources	处理测试源码，比如过滤一些值
mvn generate-test-resources	生成测试需要的资源文件
mvn process-test-resources	复制并处理测试资源文件至测试目标目录
mvn test-compile	编译测试源码至测试目标目录
mvn test	使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布
mvn prepare-package	在真正的打包之前，执行一些准备打包必要的操作。这通常会产生一个包的展开的处理过的版本（将会在 Maven 2.1+中实现）
mvn package	将编译好的代码打包成可分发的格式，如 JAR，WAR，或者 EAR

生命周期	阶段描述
mvn pre-integration-test	执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境
mvn integration-test	如果有必要的话，处理包并发布至集成测试可以运行的环境
mvn post-integration-test	执行一些在集成测试运行之后需要的动作。如清理集成测试环境。
mvn verify	执行所有检查，验证包是有效的，符合质量规范
mvn install	安装包至本地仓库，以备本地的其它项目作为依赖使用
mvn deploy	复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关）

使用参数

`-Dmaven.test.skip=true` : 跳过单元测试(eg: mvn clean package -Dmaven.test.skip=true)

引用和引申

- 官方文档
- http://www.oschina.net/question/158170_29368
- http://www.cnblogs.com/crazy-fox/archive/2012/02/09/2343722.html

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 15:29:01

- Maven 教程之 pom.xml 详解
  - 简介
    - 什么是 pom？
    - pom 配置一览
  - 基本配置
    - maven 坐标
  - 依赖配置
    - dependencies
    - parent
    - dependencyManagement
    - modules
    - properties
  - 构建配置
    - build
    - reporting
  - 项目信息
  - 环境配置
    - issueManagement
    - ciManagement
    - mailingLists
    - scm
    - prerequisites
    - repositories
    - pluginRepositories
    - distributionManagement
    - profiles
  - 参考资料

Maven 教程之 pom.xml 详解

> 📓 本文已归档到：「blog」

- 简介
  - 什么是 pom？
  - pom 配置一览
- 基本配置
  - maven 坐标
- 依赖配置
  - dependencies
  - parent
  - dependencyManagement
  - modules
  - properties
- 构建配置

    - build
    - reporting
- 项目信息
- 环境配置
    - issueManagement
    - ciManagement
    - mailingLists
    - scm
    - prerequisites
    - repositories
    - pluginRepositories
    - distributionManagement
    - profiles
- 参考资料

简介

什么是 pom？

POM 是 Project Object Model 的缩写，即项目对象模型。

pom.xml 就是 maven 的配置文件，用以描述项目的各种信息。

pom 配置一览

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xs
  <modelVersion>4.0.0</modelVersion>

  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>

  <!-- More Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <contributors>...</contributors>

  <!-- Environment Settings -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

基本配置

- **project -** `project` 是 pom.xml 中描述符的根。

- **modelVersion** - `modelVersion` 指定 pom.xml 符合哪个版本的描述符。maven 2 和 3 只能为 4.0.0。

一般 jar 包被识别为： `groupId:artifactId:version` 的形式。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xs
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
</project>
```

maven 坐标

在 **maven** 中，根据 `groupId` 、 `artifactId` 、 `version` 组合成 `groupId:artifactId:version` 来唯一识别一个 **jar** 包。

- **groupId** - 团体、组织的标识符。团体标识的约定是，它以创建这个项目的组织名称的逆向域名(reverse domain name)开头。一般对应着 java 的包结构。
- **artifactId** - 单独项目的唯一标识符。比如我们的 tomcat、commons 等。不要在 artifactId 中包含点号(.)。
- **version** - 一个项目的特定版本。
  - maven 有自己的版本规范，一般是如下定义 major version、minor version、incremental version-qualifier ，比如 1.2.3-beta-01。要说明的是，maven 自己判断版本的算法是 major、minor、incremental 部分用数字比较，qualifier 部分用字符串比较，所以要小心 alpha-2 和 alpha-15 的比较关系，最好用 alpha-02 的格式。
  - maven 在版本管理时候可以使用几个特殊的字符串 SNAPSHOT、LATEST、RELEASE。比如 `1.0-SNAPSHOT` 。各个部分的含义和处理逻辑如下说明：
    - **SNAPSHOT** - 这个版本一般用于开发过程中，表示不稳定的版本。
    - **LATEST** - 指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个 snapshot 版，具体看哪个时间最后。
    - **RELEASE** ：指最后一个发布版。

- **packaging** - 项目的类型，描述了项目打包后的输出，默认是 jar。常见的输出类型为：pom, jar, maven-plugin, ejb, war, ear, rar, par。

依赖配置

dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <dependencies>
    <dependency>
     <groupId>org.apache.maven</groupId>
      <artifactId>maven-embedder</artifactId>
      <version>2.0</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
      <exclusions>
        <exclusion>
          <groupId>org.apache.maven</groupId>
          <artifactId>maven-core</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

- **groupId**, **artifactId**, **version** - 和基本配置中的 `groupId` 、 `artifactId` 、 `version` 意义相同。
- **type** - 对应 `packaging` 的类型，如果不使用 `type` 标签，maven 默认为 jar。
- **scope** - 此元素指的是任务的类路径（编译和运行时，测试等）以及如何限制依赖关系的传递性。有 5 种可用的限定范围：
  - **compile** - 如果没有指定 `scope` 标签，maven 默认为这个范围。编译依赖关系在所有 classpath 中都可用。此外，这些依赖关系被传播到依赖项目。
  - **provided** - 与 compile 类似，但是表示您希望 jdk 或容器在运行时提供它。它只适用于编译和测试 classpath，不可传递。

    - **runtime** - 此范围表示编译不需要依赖关系，而是用于执行。它是在运行时和测试 classpath，但不是编译 classpath。
    - **test** - 此范围表示正常使用应用程序不需要依赖关系，仅适用于测试编译和执行阶段。它不是传递的。
    - **system** - 此范围与 provided 类似，除了您必须提供明确包含它的 jar。该 artifact 始终可用，并且不是在仓库中查找。
- **systemPath** - 仅当依赖范围是系统时才使用。否则，如果设置此元素，构建将失败。该路径必须是绝对路径，因此建议使用 `propertie` 来指定特定的路径，如$ {java.home} / lib。由于假定先前安装了系统范围依赖关系，maven 将不会检查项目的仓库，而是检查库文件是否存在。如果没有，maven 将会失败，并建议您手动下载安装。
- **optional** - `optional` 让其他项目知道，当您使用此项目时，您不需要这种依赖性才能正常工作。
- **exclusions** - 包含一个或多个排除元素，每个排除元素都包含一个表示要排除的依赖关系的 `groupId` 和 `artifactId` 。与可选项不同，可能或可能不会安装和使用，排除主动从依赖关系树中删除自己。

parent

maven 支持继承功能。子 POM 可以使用 `parent` 指定父 POM ，然后继承其配置。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>my-parent</artifactId>
    <version>2.0</version>
    <relativePath>../my-parent</relativePath>
  </parent>

  <artifactId>my-project</artifactId>
</project>
```

- **relativePath** - 注意 `relativePath` 元素。在搜索本地和远程存储库之前，它不是必需的，但可以用作 maven 的指示符，以首先搜索给定该项目父级的路径。

dependencyManagement

`dependencyManagement` 是表示依赖 jar 包的声明。即你在项目中的 `dependencyManagement` 下声明了依赖，maven 不会加载该依赖，`dependencyManagement` 声明可以被子 POM 继承。

`dependencyManagement` 的一个使用案例是当有父子项目的时候，父项目中可以利用 `dependencyManagement` 声明子项目中需要用到的依赖 jar 包，之后，当某个或者某几个子项目需要加载该依赖的时候，就可以在子项目中 `dependencies` 节点只配置 `groupId` 和 `artifactId` 就可以完成依赖的引用。

`dependencyManagement` 主要是为了统一管理依赖包的版本，确保所有子项目使用的版本一致，类似的还有 `plugins` 和 `pluginManagement` 。

modules

子模块列表。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <packaging>pom</packaging>

  <modules>
    <module>my-project</module>
    <module>another-project</module>
    <module>third-project/pom-example.xml</module>
  </modules>
</project>
```

properties

属性列表。定义的属性可以在 pom.xml 文件中任意处使用。使用方式为 `${propertie}` 。

```
<project>
  ...
  <properties>
    <maven.compiler.source>1.7<maven.compiler.source>
    <maven.compiler.target>1.7<maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEnc
    <project.reporting.outputEncoding>UTF-8</project.reporting.o
  </properties>
  ...
</project>
```

构建配置

build

build 可以分为 "project build" 和 "profile build"。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <!-- "Project Build" contains more elements than just the Base
  <build>...</build>

  <profiles>
    <profile>
      <!-- "Profile Build" contains a subset of "Project Build"s
      <build>...</build>
    </profile>
  </profiles>
</project>
```

基本构建配置：

```
<build>
  <defaultGoal>install</defaultGoal>
  <directory>${basedir}/target</directory>
  <finalName>${artifactId}-${version}</finalName>
  <filters>
    <filter>filters/filter1.properties</filter>
  </filters>
  ...
</build>
```

defaultGoal : 默认执行目标或阶段。如果给出了一个目标，它应该被定义为它在命令行中（如 jar：jar）。如果定义了一个阶段（如安装），也是如此。

directory ：构建时的输出路径。默认为： `${basedir}/target` 。

finalName ：这是项目的最终构建名称（不包括文件扩展名，例如：my-project-1.0.jar）

filter ：定义 `* .properties` 文件，其中包含适用于接受其设置的资源的属性列表（如下所述）。换句话说，过滤器文件中定义的“name = value”对在代码中替换$ {name}字符串。

resources

资源的配置。资源文件通常不是代码，不需要编译，而是在项目需要捆绑使用的内容。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  <build>
    ...
    <resources>
      <resource>
        <targetPath>META-INF/plexus</targetPath>
        <filtering>false</filtering>
        <directory>${basedir}/src/main/plexus</directory>
        <includes>
          <include>configuration.xml</include>
        </includes>
        <excludes>
          <exclude>**/*.properties</exclude>
        </excludes>
      </resource>
    </resources>
    <testResources>
      ...
    </testResources>
    ...
  </build>
</project>
```

- **resources**: 资源元素的列表，每个资源元素描述与此项目关联的文件和何处包含文件。
- **targetPath**: 指定从构建中放置资源集的目录结构。目标路径默认为基本目录。将要包装在 jar 中的资源的通常指定的目标路径是 META-INF。
- **filtering**: 值为 true 或 false。表示是否要为此资源启用过滤。请注意，该过滤器 `* .properties` 文件不必定义为进行过滤 - 资源还可以使用默认情况下在 POM 中定义的属性（例如${project.version}），并将其传递到命令行中“-D”标志（例如，“-Dname = value”）或由 properties 元素显式定义。过滤文件覆盖上面。
- **directory**: 值定义了资源的路径。构建的默认目录是 `${basedir}/src/main/resources` 。
- **includes**: 一组文件匹配模式，指定目录中要包括的文件，使用*作为通配符。
- **excludes**: 与 `includes` 类似，指定目录中要排除的文件，使用*作为通配符。注意：如果 `include` 和 `exclude` 发生冲突，maven 会以 `exclude` 作为有效项。

- **testResources**: `testResources` 与 `resources` 功能类似，区别仅在于： `testResources` 指定的资源仅用于 test 阶段，并且其默认资源目录为： `${basedir}/src/test/resources` 。

plugins

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <extensions>false</extensions>
        <inherited>true</inherited>
        <configuration>
          <classifier>test</classifier>
        </configuration>
        <dependencies>...</dependencies>
        <executions>...</executions>
      </plugin>
    </plugins>
  </build>
</project>
```

- **groupId**, **artifactId**, **version** ： 和基本配置中的 `groupId` 、 `artifactId` 、 `version` 意义相同。
- **extensions** ： 值为 true 或 false。是否加载此插件的扩展名。默认为 false。
- **inherited** ： 值为 true 或 false。这个插件配置是否应该适用于继承自这个插件的 POM。默认值为 true。
- **configuration** - 这是针对个人插件的配置，这里不扩散讲解。
- **dependencies** ： 这里的 `dependencies` 是插件本身所需要的依赖。
- **executions** ： 需要记住的是，插件可能有多个目标。每个目标可能有一个单独的配置，甚至可能将插件的目标完全绑定到不同的阶段。执行配置插件的目标的执行。
    - **id**: 执行目标的标识。

- **goals**: 像所有多元化的 POM 元素一样，它包含单个元素的列表。在这种情况下，这个执行块指定的插件目标列表。
- **phase**: 这是执行目标列表的阶段。这是一个非常强大的选项，允许将任何目标绑定到构建生命周期中的任何阶段，从而改变 maven 的默认行为。
- **inherited**: 像上面的继承元素一样，设置这个 false 会阻止 maven 将这个执行传递给它的子代。此元素仅对父 POM 有意义。
- **configuration**: 与上述相同，但将配置限制在此特定目标列表中，而不是插件下的所有目标。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <id>echodir</id>
            <goals>
              <goal>run</goal>
            </goals>
            <phase>verify</phase>
            <inherited>false</inherited>
            <configuration>
              <tasks>
                <echo>Build Dir: ${project.build.directory}</ech
              </tasks>
            </configuration>
          </execution>
        </executions>

      </plugin>
    </plugins>
  </build>
</project>
```

pluginManagement

与 `dependencyManagement` 很相似，在当前 POM 中仅声明插件，而不是实际引入插件。子 POM 中只配置 `groupId` 和 `artifactId` 就可以完成插件的引用，且子 POM 有权重写 pluginManagement 定义。

它的目的在于统一所有子 POM 的插件版本。

directories

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <build>
    <sourceDirectory>${basedir}/src/main/java</sourceDirectory>
    <scriptSourceDirectory>${basedir}/src/main/scripts</scriptSo
    <testSourceDirectory>${basedir}/src/test/java</testSourceDir
    <outputDirectory>${basedir}/target/classes</outputDirectory>
    <testOutputDirectory>${basedir}/target/test-classes</testOut
    ...
  </build>
</project>
```

目录元素集合存在于 `build` 元素中，它为整个 POM 设置了各种目录结构。由于它们在配置文件构建中不存在，所以这些不能由配置文件更改。

如果上述目录元素的值设置为绝对路径（扩展属性时），则使用该目录。否则，它是相对于基础构建目录：`${basedir}` 。

extensions

扩展是在此构建中使用的 artifacts 的列表。它们将被包含在运行构建的 classpath 中。它们可以启用对构建过程的扩展（例如为 Wagon 传输机制添加一个 ftp 提供程序），并使活动的插件能够对构建生命周期进行更改。简而言之，扩展是在构建期间激活的 artifacts。扩展不需要实际执行任何操作，也不包含 Mojo。因此，扩展对于指定普通插件接口的多个实现中的一个是非常好的。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <build>
    ...
    <extensions>
      <extension>
        <groupId>org.apache.maven.wagon</groupId>
        <artifactId>wagon-ftp</artifactId>
        <version>1.0-alpha-3</version>
      </extension>
    </extensions>
    ...
  </build>
</project>
```

reporting

报告包含特定针对 `site` 生成阶段的元素。某些 maven 插件可以生成 `reporting` 元素下配置的报告，例如：生成 javadoc 报告。 `reporting` 与 `build` 元素配置插件的能力相似。明显的区别在于：在执行块中插件目标的控制不是细粒度的，报表通过配置 `reportSet` 元素来精细控制。而微妙的区别在于 `reporting` 元素下的 `configuration` 元素可以用作 `build` 下的 `configuration` ，尽管相反的情况并非如此（ `build` 下的 `configuration` 不影响 `reporting` 元素下的 `configuration` ）。

另一个区别就是 `plugin` 下的 `outputDirectory` 元素。在报告的情况下，默认输出目录为 `${basedir}/target/site` 。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <reporting>
    <plugins>
      <plugin>
        ...
        <reportSets>
          <reportSet>
            <id>sunlink</id>
            <reports>
              <report>javadoc</report>
            </reports>
            <inherited>true</inherited>
            <configuration>
              <links>
                <link>http://java.sun.com/j2se/1.5.0/docs/api/</
              </links>
            </configuration>
          </reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

项目信息

项目信息相关的这部分标签**都不是必要的**，也就是说完全可以不填写。

它的作用仅限于描述项目的详细信息。

下面的示例是项目信息相关标签的清单：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...

  <!-- 项目信息 begin -->

  <!--项目名-->
  <name>maven-notes</name>

  <!--项目描述-->
  <description>maven 学习笔记</description>

  <!--项目url-->
  <url>https://github.com/dunwu/maven-notes</url>

  <!--项目开发年份-->
  <inceptionYear>2017</inceptionYear>

  <!--开源协议-->
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
      <comments>A business-friendly OSS license</comments>
    </license>
  </licenses>

  <!--组织信息(如公司、开源组织等)-->
  <organization>
    <name>...</name>
    <url>...</url>
  </organization>

  <!--开发者列表-->
  <developers>
    <developer>
      <id>victor</id>
      <name>Zhang Peng</name>
      <email>forbreak at 163.com</email>
      <url>https://github.com/dunwu</url>
      <organization>...</organization>
      <organizationUrl>...</organizationUrl>
      <roles>
        <role>architect</role>
        <role>developer</role>
```

```
      </roles>
      <timezone>+8</timezone>
      <properties>...</properties>
    </developer>
  </developers>

  <!--代码贡献者列表-->
   <contributors>
    <contributor>
      <!--标签内容和<developer>相同-->
    </contributor>
  </contributors>

  <!-- 项目信息 end -->

  ...
</project>
```

这部分标签都非常简单，基本都能做到顾名思义，且都属于可有可无的标签，所以这里仅简单介绍一下：

- **name** - 项目完整名称
- **description** - 项目描述
- **url** - 一般为项目仓库的 host
- **inceptionYear** - 开发年份
- **licenses** - 开源协议
- **organization** - 项目所属组织信息
- **developers** - 项目开发者列表
- **contributors** - 项目贡献者列表，`<contributor>` 的子标签和 `<developer>` 的完全相同。

环境配置

issueManagement

这定义了所使用的缺陷跟踪系统（Bugzilla，TestTrack，ClearQuest 等）。虽然没有什么可以阻止插件使用这些信息的东西，但它主要用于生成项目文档。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <issueManagement>
    <system>Bugzilla</system>
    <url>http://127.0.0.1/bugzilla/</url>
  </issueManagement>
  ...
</project>
```

ciManagement

CI 构建系统配置，主要是指定通知机制以及被通知的邮箱。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <ciManagement>
    <system>continuum</system>
    <url>http://127.0.0.1:8080/continuum</url>
    <notifiers>
      <notifier>
        <type>mail</type>
        <sendOnError>true</sendOnError>
        <sendOnFailure>true</sendOnFailure>
        <sendOnSuccess>false</sendOnSuccess>
        <sendOnWarning>false</sendOnWarning>
        <configuration><address>continuum@127.0.0.1</address></c
      </notifier>
    </notifiers>
  </ciManagement>
  ...
</project>
```

mailingLists

邮件列表

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <mailingLists>
    <mailingList>
      <name>User List</name>
      <subscribe>user-subscribe@127.0.0.1</subscribe>
      <unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>
      <post>user@127.0.0.1</post>
      <archive>http://127.0.0.1/user/</archive>
      <otherArchives>
        <otherArchive>http://base.google.com/base/1/127.0.0.1</o
      </otherArchives>
    </mailingList>
  </mailingLists>
  ...
</project>
```

scm

SCM（软件配置管理，也称为源代码/控制管理或简洁的版本控制）。常见的 scm 有 svn 和 git 。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <scm>
    <connection>scm:svn:http://127.0.0.1/svn/my-project</connect
    <developerConnection>scm:svn:https://127.0.0.1/svn/my-projec
    <tag>HEAD</tag>
    <url>http://127.0.0.1/websvn/my-project</url>
  </scm>
  ...
</project>
```

prerequisites

POM 执行的预设条件。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <prerequisites>
    <maven>2.0.6</maven>
  </prerequisites>
  ...
</project>
```

repositories

`repositories` 是遵循 Maven 存储库目录布局的 artifacts 集合。默认的 Maven 中央存储库位于https://repo.maven.apache.org/maven2/上。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <repositories>
    <repository>
      <releases>
        <enabled>false</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <id>codehausSnapshots</id>
      <name>Codehaus Snapshots</name>
      <url>http://snapshots.maven.codehaus.org/maven2</url>
      <layout>default</layout>
    </repository>
  </repositories>
  <pluginRepositories>
    ...
  </pluginRepositories>
  ...
</project>
```

pluginRepositories

与 `repositories` 差不多。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <distributionManagement>
    ...
    <downloadUrl>http://mojo.codehaus.org/my-project</downloadUr
    <status>deployed</status>
  </distributionManagement>
  ...
</project>
```

distributionManagement

它管理在整个构建过程中生成的 artifact 和支持文件的分布。从最后的元素开始：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <distributionManagement>
    ...
    <downloadUrl>http://mojo.codehaus.org/my-project</downloadUr
    <status>deployed</status>
  </distributionManagement>
  ...
</project>
```

- **repository** - 与 `repositories` 相似
- **site** - 站点信息
- **relocation** - 项目迁移位置

profiles

`activation` 是一个 `profile` 的关键。配置文件的功能来自于在某些情况下仅修改基本 POM 的功能。这些情况通过 `activation` 元素指定。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.x
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>sparrow-type</name>
          <value>African</value>
        </property>
        <file>
          <exists>${basedir}/file2.properties</exists>
          <missing>${basedir}/file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
</project>
```

参考资料

- maven 官方文档之 pom

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 15:29:01

- Maven 教程之 settings.xml 详解
  - 简介
    - settings.xml 有什么用？
    - settings.xml 文件位置
    - 配置优先级
  - settings.xml 元素详解
    - 顶级元素概览
    - LocalRepository
    - InteractiveMode
    - UsePluginRegistry
    - Offline
    - PluginGroups
    - Servers
    - Mirrors
    - Proxies
    - Profiles
    - ActiveProfiles
  - 参考资料

Maven 教程之 settings.xml 详解

> 本文已归档到：「blog」

- 简介
  - settings.xml 有什么用？
  - settings.xml 文件位置
  - 配置优先级
- settings.xml 元素详解
  - 顶级元素概览
  - LocalRepository
  - InteractiveMode
  - UsePluginRegistry
  - Offline
  - PluginGroups
  - Servers
  - Mirrors
  - Proxies
  - Profiles
  - ActiveProfiles
- 参考资料

简介

settings.xml 有什么用？

从 settings.xml 的文件名就可以看出，它是用来设置 maven 参数的配置文件。settings.xml 中包含类似本地仓储位置、修改远程仓储服务器、认证信息等配置。

- settings.xml 是 maven 的全局配置文件。
- pom.xml 文件是本地项目配置文件。

settings.xml 文件位置

settings.xml 文件一般存在于两个位置：

- 全局配置 - `${maven.home}/conf/settings.xml`
- 用户配置 - `${user.home}/.m2/settings.xml`

> 注意：用户配置优先于全局配置。`${user.home}` 和和所有其他系统属性只能在 3.0+版本上使用。请注意 windows 和 Linux 使用变量的区别。

配置优先级

> 重要：局部配置优先于全局配置。

配置优先级从高到低：pom.xml > user settings > global settings

如果这些文件同时存在，在应用配置时，会合并它们的内容，如果有重复的配置，优先级高的配置会覆盖优先级低的。

settings.xml 元素详解

顶级元素概览

下面列举了 `settings.xml` 中的顶级元素

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                          https://maven.apache.org/xsd/settings-
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

LocalRepository

作用：该值表示构建系统本地仓库的路径。

其默认值：~/.m2/repository。

```
<localRepository>${user.home}/.m2/repository</localRepository>
```

InteractiveMode

作用：表示 maven 是否需要和用户交互以获得输入。

如果 maven 需要和用户交互以获得输入，则设置成 true，反之则应为 false。默认为 true。

```
<interactiveMode>true</interactiveMode>
```

UsePluginRegistry

作用：maven 是否需要使用 plugin-registry.xml 文件来管理插件版本。

如果需要让 maven 使用文件~/.m2/plugin-registry.xml 来管理插件版本，则设为 true。默认为 false。

```
<usePluginRegistry>false</usePluginRegistry>
```

Offline

作用：表示 maven 是否需要在离线模式下运行。

如果构建系统需要在离线模式下运行，则为 true，默认为 false。

当由于网络设置原因或者安全因素，构建服务器不能连接远程仓库的时候，该配置就十分有用。

```
<offline>false</offline>
```

PluginGroups

作用：当插件的组织 id（groupId）没有显式提供时，供搜寻插件组织 Id（groupId）的列表。

该元素包含一个 pluginGroup 元素列表，每个子元素包含了一个组织 Id（groupId）。

当我们使用某个插件，并且没有在命令行为其提供组织 Id（groupId）的时候，Maven 就会使用该列表。默认情况下该列表包含了 `org.apache.maven.plugins` 和 `org.codehaus.mojo` 。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <pluginGroups>
    <!--plugin的组织Id（groupId） -->
    <pluginGroup>org.codehaus.mojo</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

Servers

作用：一般，仓库的下载和部署是在 pom.xml 文件中的 `repositories` 和 `distributionManagement` 元素中定义的。然而，一般类似用户名、密码（**有些仓库访问是需要安全认证的**）等信息不应该在 pom.xml 文件中配置，这些信息可以配置在 `settings.xml` 中。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <!--配置服务端的一些设置。一些设置如安全证书不应该和pom.xml一起分发
  <servers>
    <!--服务器元素包含配置服务器时需要的信息 -->
    <server>
      <!--这是server的id（注意不是用户登陆的id），该id与distributio
      <id>server001</id>
      <!--鉴权用户名。鉴权用户名和鉴权密码表示服务器认证所需要的登录名
      <username>my_login</username>
      <!--鉴权密码 。鉴权用户名和鉴权密码表示服务器认证所需要的登录名
      <password>my_password</password>
      <!--鉴权时使用的私钥位置。和前两个元素类似，私钥位置和私钥密码指
      <privateKey>${usr.home}/.ssh/id_dsa</privateKey>
      <!--鉴权时使用的私钥密码。 -->
      <passphrase>some_passphrase</passphrase>
      <!--文件被创建时的权限。如果在部署的时候会创建一个仓库文件或者目
      <filePermissions>664</filePermissions>
      <!--目录被创建时的权限。 -->
      <directoryPermissions>775</directoryPermissions>
    </server>
  </servers>
  ...
</settings>
```

Mirrors

作用：为仓库列表配置的下载镜像列表。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <mirrors>
    <!-- 给定仓库的下载镜像。 -->
    <mirror>
      <!-- 该镜像的唯一标识符。id用来区分不同的mirror元素。 -->
      <id>planetmirror.com</id>
      <!-- 镜像名称 -->
      <name>PlanetMirror Australia</name>
      <!-- 该镜像的URL。构建系统会优先考虑使用该URL，而非使用默认的服
      <url>http://downloads.planetmirror.com/pub/maven2</url>
      <!-- 被镜像的服务器的id。例如，如果我们要设置了一个Maven中央仓
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

Proxies

作用：用来配置不同的代理。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <proxies>
    <!--代理元素包含配置代理时需要的信息 -->
    <proxy>
      <!--代理的唯一定义符，用来区分不同的代理元素。 -->
      <id>myproxy</id>
      <!--该代理是否是激活的那个。true则激活代理。当我们声明了一组代理
      <active>true</active>
      <!--代理的协议。 协议://主机名:端口，分隔成离散的元素以方便配置
      <protocol>http</protocol>
      <!--代理的主机名。协议://主机名:端口，分隔成离散的元素以方便配置
      <host>proxy.somewhere.com</host>
      <!--代理的端口。协议://主机名:端口，分隔成离散的元素以方便配置。
      <port>8080</port>
      <!--代理的用户名，用户名和密码表示代理服务器认证的登录名和密码。
      <username>proxyuser</username>
      <!--代理的密码，用户名和密码表示代理服务器认证的登录名和密码。
      <password>somepassword</password>
      <!--不该被代理的主机名列表。该列表的分隔符由代理服务器指定；例子
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

Profiles

作用：根据环境参数来调整构建配置的列表。

`settings.xml` 中的 `profile` 元素是 `pom.xml` 中 `profile` 元素的裁剪版本。

它包含了 `id` 、 `activation` 、 `repositories` 、 `pluginRepositories` 和 `properties` 元素。这里的 profile 元素只包含这五个子元素是因为这里只关心构建系统这个整体（这正是 settings.xml 文件的角色定位），而非单独的项目对象模型设置。如果一个 `settings.xml` 中的 `profile` 被激活，它的值会覆盖任何其它定义在 `pom.xml` 中带有相同 id 的 `profile` 。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <profiles>
    <profile>
      <!-- profile的唯一标识 -->
      <id>test</id>
      <!-- 自动触发profile的条件逻辑 -->
      <activation />
      <!-- 扩展属性列表 -->
      <properties />
      <!-- 远程仓库列表 -->
      <repositories />
      <!-- 插件仓库列表 -->
      <pluginRepositories />
    </profile>
  </profiles>
  ...
</settings>
```

Activation

作用：自动触发 `profile` 的条件逻辑。

如 `pom.xml` 中的 `profile` 一样， `profile` 的作用在于它能够在某些特定的环境中自动使用某些特定的值；这些环境通过 `activation` 元素指定。 `activation` 元素并不是激活 `profile` 的唯一方式。 `settings.xml` 文件中的 `activeProfile` 元素可以包含 `profile` 的 `id` 。 `profile` 也可以通过在命令行，使用 `-P` 标记和逗号分隔的列表来显式的激活（如， `-P test` ）。

```
<activation>
  <!--profile默认是否激活的标识 -->
  <activeByDefault>false</activeByDefault>
  <!--当匹配的jdk被检测到，profile被激活。例如，1.4激活JDK1.4，1.4.0
  <jdk>1.5</jdk>
  <!--当匹配的操作系统属性被检测到，profile被激活。os元素可以定义一些
  <os>
    <!--激活profile的操作系统的名字 -->
    <name>Windows XP</name>
    <!--激活profile的操作系统所属家族(如 'windows') -->
    <family>Windows</family>
    <!--激活profile的操作系统体系结构 -->
    <arch>x86</arch>
    <!--激活profile的操作系统版本 -->
    <version>5.1.2600</version>
  </os>
  <!--如果Maven检测到某一个属性（其值可以在POM中通过${name}引用），其
  <property>
    <!--激活profile的属性的名称 -->
    <name>mavenVersion</name>
    <!--激活profile的属性的值 -->
    <value>2.0.3</value>
  </property>
  <!--提供一个文件名，通过检测该文件的存在或不存在来激活profile。miss
  <file>
    <!--如果指定的文件存在，则激活profile。 -->
    <exists>${basedir}/file2.properties</exists>
    <!--如果指定的文件不存在，则激活profile。 -->
    <missing>${basedir}/file1.properties</missing>
  </file>
</activation>
```

> 注：在 maven 工程的 pom.xml 所在目录下执行 `mvn help:active-profiles` 命令可以查看中央仓储的 profile 是否在工程中生效。

properties

作用：对应 `profile` 的扩展属性列表。

maven 属性和 ant 中的属性一样，可以用来存放一些值。这些值可以在 `pom.xml` 中的任何地方使用标记${X}来使用，这里 X 是指属性的名称。属性有五种不同的形式，并且都能在 settings.xml 文件中访问。

```
<!--
  1. env.X: 在一个变量前加上"env."的前缀，会返回一个shell环境变量。例
  2. project.x: 指代了POM中对应的元素值。例如: <project><version>1.
  3. settings.x: 指代了settings.xml中对应元素的值。例如: <settings>
  4. Java System Properties: 所有可通过java.lang.System.getProper
  5. x: 在<properties/>元素中，或者外部文件中设置，以${someVar}的形
 -->
<properties>
  <user.install>${user.home}/our-project</user.install>
</properties>
```

> 注：如果该 profile 被激活，则可以在 `pom.xml` 中使用 ${user.install}。

Repositories

作用：远程仓库列表，它是 maven 用来填充构建系统本地仓库所使用的一组远程仓库。

```
<repositories>
  <!--包含需要连接到远程仓库的信息 -->
  <repository>
    <!--远程仓库唯一标识 -->
    <id>codehausSnapshots</id>
    <!--远程仓库名称 -->
    <name>Codehaus Snapshots</name>
    <!--如何处理远程仓库里发布版本的下载 -->
    <releases>
      <!--true或者false表示该仓库是否为下载某种类型构件（发布版，快照
      <enabled>false</enabled>
      <!--该元素指定更新发生的频率。Maven会比较本地POM和远程POM的时间
      <updatePolicy>always</updatePolicy>
      <!--当Maven验证构件校验文件失败时该怎么做-ignore（忽略），fail
      <checksumPolicy>warn</checksumPolicy>
    </releases>
    <!--如何处理远程仓库里快照版本的下载。有了releases和snapshots这两
    <snapshots>
      <enabled />
      <updatePolicy />
      <checksumPolicy />
    </snapshots>
    <!--远程仓库URL，按protocol://hostname/path形式 -->
    <url>http://snapshots.maven.codehaus.org/maven2</url>
    <!--用于定位和排序构件的仓库布局类型-可以是default（默认）或者leg
    <layout>default</layout>
  </repository>
</repositories>
```

pluginRepositories

作用：发现插件的远程仓库列表。

和 `repository` 类似，只是 `repository` 是管理 jar 包依赖的仓库， `pluginRepositories` 则是管理插件的仓库。 maven 插件是一种特殊类型的构件。由于这个原因，插件仓库独立于其它仓库。 `pluginRepositories` 元素的结构和 `repositories` 元素的结构类似。每个 `pluginRepository` 元素指定一个 Maven 可以用来寻找新插件的远程地址。

```
<pluginRepositories>
  <!-- 包含需要连接到远程插件仓库的信息.参见profiles/profile/reposi
  <pluginRepository>
    <releases>
      <enabled />
      <updatePolicy />
      <checksumPolicy />
    </releases>
    <snapshots>
      <enabled />
      <updatePolicy />
      <checksumPolicy />
    </snapshots>
    <id />
    <name />
    <url />
    <layout />
  </pluginRepository>
</pluginRepositories>
```

ActiveProfiles

作用：手动激活 profiles 的列表，按照 `profile` 被应用的顺序定义 `activeProfile` 。

该元素包含了一组 `activeProfile` 元素，每个 `activeProfile` 都含有一个 profile id。任何在 `activeProfile` 中定义的 profile id，不论环境设置如何，其对应的 `profile` 都会被激活。如果没有匹配的 `profile` ，则什么都不会发生。

例如，env-test 是一个 activeProfile，则在 pom.xml（或者 profile.xml）中对应 id 的 profile 会被激活。如果运行过程中找不到这样一个 profile，Maven 则会像往常一样运行。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:x
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      https://maven.apache.org/xsd/settings-1.0.
  ...
  <activeProfiles>
    <!-- 要激活的profile id -->
    <activeProfile>env-test</activeProfile>
  </activeProfiles>
  ...
</settings>
```

至此，maven settings.xml 中的标签都讲解完毕，希望对大家有所帮助。

参考资料

- maven 官方文档之 settings

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 15:29:01

- Maven 实战问题和最佳实践
  - 1. 常见问题
    - 1.1. dependencies 和 dependencyManagement，plugins 和 pluginManagement 有什么区别？
    - 1.2. IDEA 修改 JDK 版本后编译报错
    - 1.3. 重复引入依赖
    - 1.4. 如何打包一个可以直接运行的 Spring Boot jar 包
    - 1.5. 去哪儿找 maven dependency ？
    - 1.6. 如何指定编码？
    - 1.7. 如何指定 JDK 版本？
    - 1.8. 如何避免将 dependency 打包到构件中？
    - 1.9. 如何跳过单元测试
    - 1.10. IDEA 修改 JDK 版本后编译报错
    - 1.11. 重复引入依赖
    - 1.12. 如何引入本地 jar
    - 1.13. 如何排除依赖
  - 2. 最佳实践
    - 2.1. 通过 bom 统一管理版本

Maven 实战问题和最佳实践

- 1. 常见问题
  - 1.1. dependencies 和 dependencyManagement，plugins 和 pluginManagement 有什么区别？
  - 1.2. IDEA 修改 JDK 版本后编译报错
  - 1.3. 重复引入依赖
  - 1.4. 如何打包一个可以直接运行的 Spring Boot jar 包
  - 1.5. 去哪儿找 maven dependency ？
  - 1.6. 如何指定编码？
  - 1.7. 如何指定 JDK 版本？
  - 1.8. 如何避免将 dependency 打包到构件中？
  - 1.9. 如何跳过单元测试
  - 1.10. IDEA 修改 JDK 版本后编译报错
  - 1.11. 重复引入依赖
  - 1.12. 如何引入本地 jar
  - 1.13. 如何排除依赖
- 2. 最佳实践
  - 2.1. 通过 bom 统一管理版本

1. 常见问题

1.1. dependencies 和 dependencyManagement，plugins 和 pluginManagement 有什么区别？

dependencyManagement 是表示依赖 jar 包的声明，即你在项目中的 dependencyManagement 下声明了依赖，maven 不会加载该依赖，dependencyManagement 声明可以被继承。

dependencyManagement 的一个使用案例是当有父子项目的时候，父项目中可以利用 dependencyManagement 声明子项目中需要用到的依赖 jar 包，之后，当某个或者某几个子项目需要加载该插件的时候，就可以在子项目中 dependencies 节点只配置 groupId 和 artifactId 就可以完成插件的引用。

dependencyManagement 主要是为了统一管理插件，确保所有子项目使用的插件版本保持一致，类似的还有 plugins 和 pluginManagement。

1.2. IDEA 修改 JDK 版本后编译报错

错误现象

修改 JDK 版本，指定 maven-compiler-plugin 的 source 和 target 为 1.8 。

然后，在 Intellij IDEA 中执行 maven 指令，报错：

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-co
```

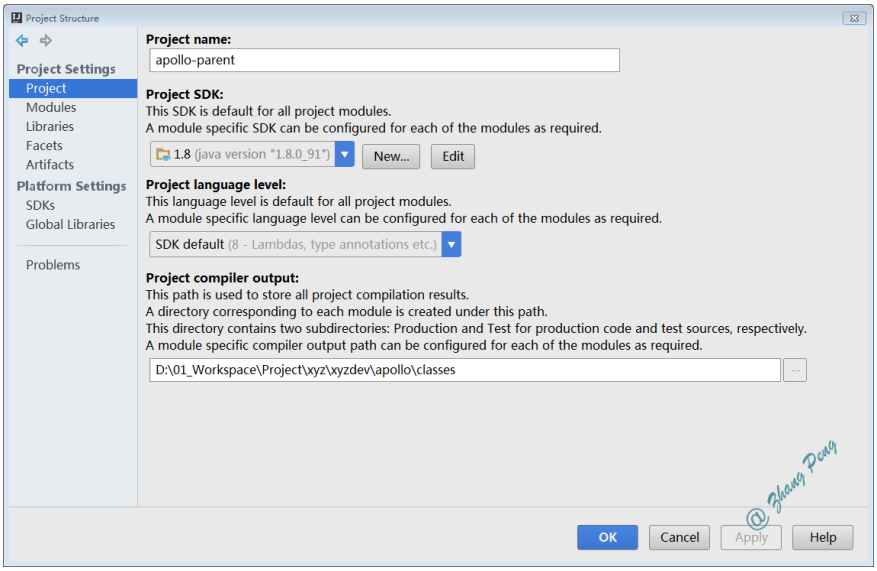
错误原因

maven 的 JDK 源与指定的 JDK 编译版本不符。

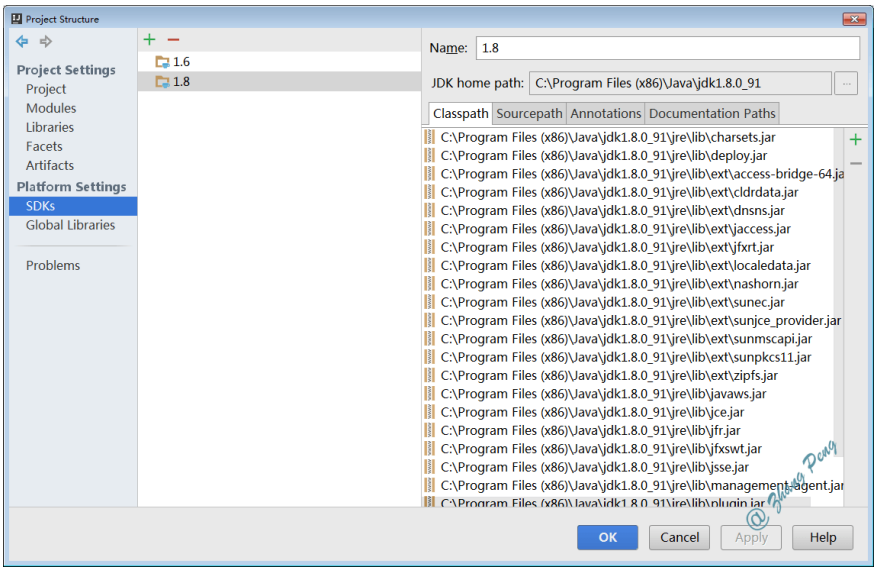
排错手段

- 查看 **Project Settings**

Project SDK 是否正确

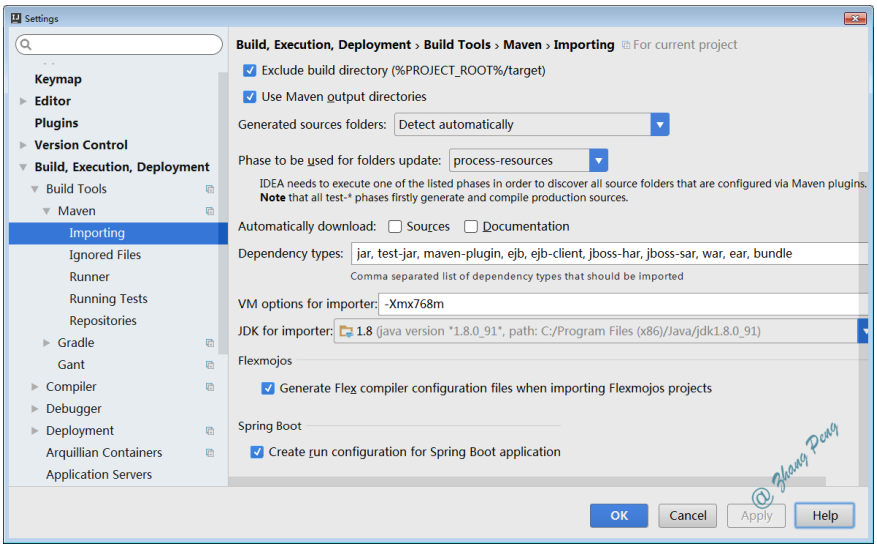


SDK 路径是否正确

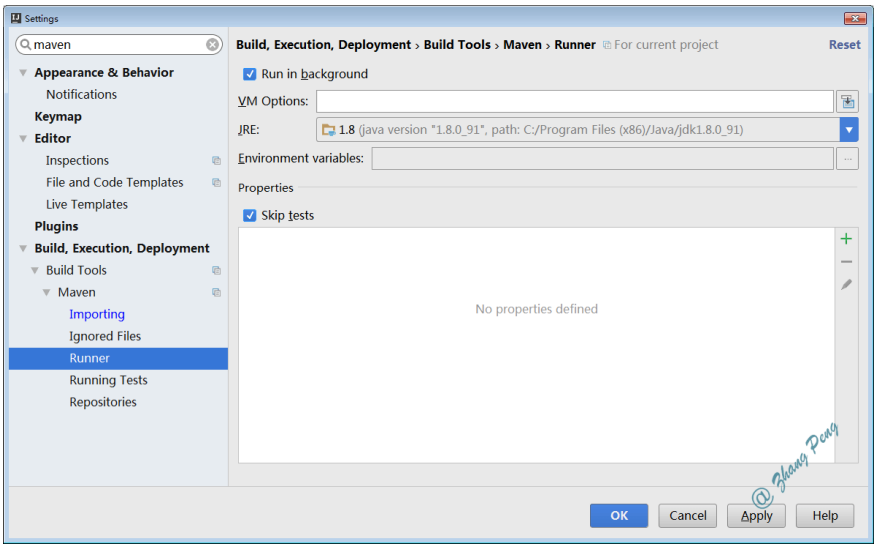


- 查看 **Settings > Maven** 的配置

JDK for importer 是否正确



Runner 是否正确



1.3. 重复引入依赖

在 Idea 中，选中 Module，使用 `Ctrl+Alt+Shift+U`，打开依赖图，检索是否存在重复引用的情况。如果存在重复引用，可以将多余的引用删除。

1.4. 如何打包一个可以直接运行的 Spring Boot jar 包

可以使用 spring-boot-maven-plugin 插件

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

如果引入了第三方 jar 包，如何打包？

首先，要添加依赖

```
<dependency>
  <groupId>io.github.dunwu</groupId>
  <artifactId>dunwu-common</artifactId>
  <version>1.0.0</version>
  <scope>system</scope>
  <systemPath>${project.basedir}/src/main/resources/lib/dunwu-co
</dependency>
```

接着，需要配置 spring-boot-maven-plugin 插件：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <includeSystemScope>true</includeSystemScope>
      </configuration>
    </plugin>
  </plugins>
</build>
```

1.5. 去哪儿找 maven dependency ？

问：刚接触 maven 的新手，往往会有这样的疑问，我该去哪儿找 jar？

答：官方推荐的搜索 maven dependency 网址：

- https://search.maven.org
- https://repository.apache.org
- https://mvnrepository.com

1.6. 如何指定编码？

问：众所周知，不同编码格式常常会产生意想不到的诡异问题，那么 maven 构建时如何指定 maven 构建时的编码？

答：在 properties 中指定 `project.build.sourceEncoding`

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncod
</properties>
```

1.7. 如何指定 JDK 版本？

问：如何指定 maven 构建时的 JDK 版本？

答：有两种方法：

（1）properties 方式

```
<project>
  ...
  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  ...
</project>
```

（2）使用 maven-compiler-plugin 插件，并指定 source 和 target 版本

```
<build>
...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
...
</build>
```

1.8. 如何避免将 dependency 打包到构件中？

答：指定 maven dependency 的 scope 为 `provided` ，这意味着：依赖关系将在运行时由其容器或 JDK 提供。 具有此范围的依赖关系不会传递，也不会捆绑在诸如 WAR 之类的包中，也不会包含在运行时类路径

中。

1.9. 如何跳过单元测试

问：执行 mvn package 或 mvn install 时，会自动编译所有单元测试 (src/test/java 目录下的代码)，如何跳过这一步？

答：在执行命令的后面，添加命令行参数 `-Dmaven.test.skip=true` 或者 `-DskipTests=true`

1.10. IDEA 修改 JDK 版本后编译报错

错误现象

修改 JDK 版本，指定 maven-compiler-plugin 的 source 和 target 为 1.8 。

然后，在 Intellij IDEA 中执行 maven 指令，报错：

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-co
```

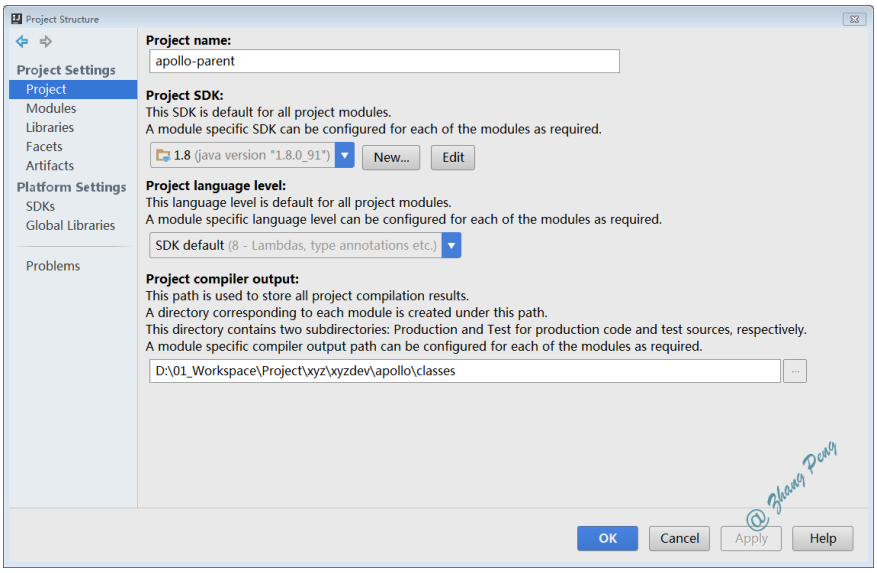
错误原因

maven 的 JDK 源与指定的 JDK 编译版本不符。

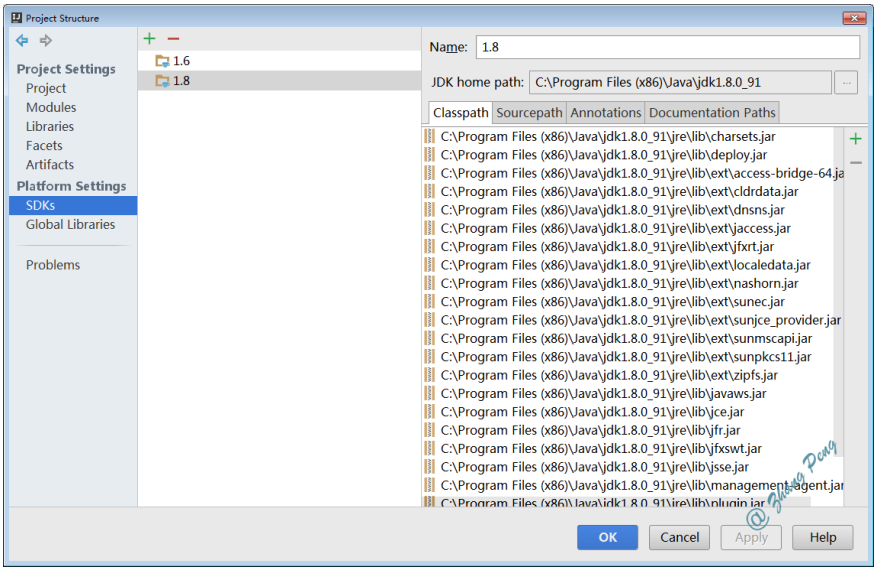
排错手段

- 查看 **Project Settings**

Project SDK 是否正确

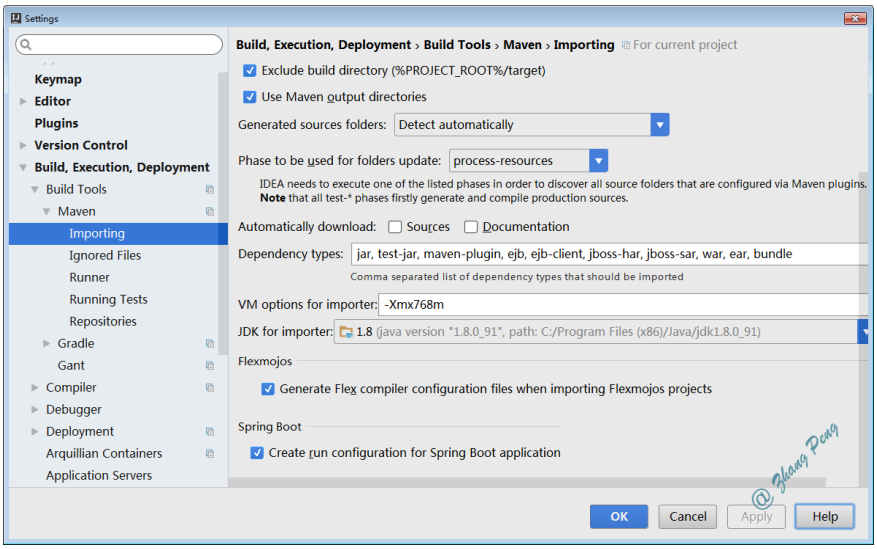


SDK 路径是否正确

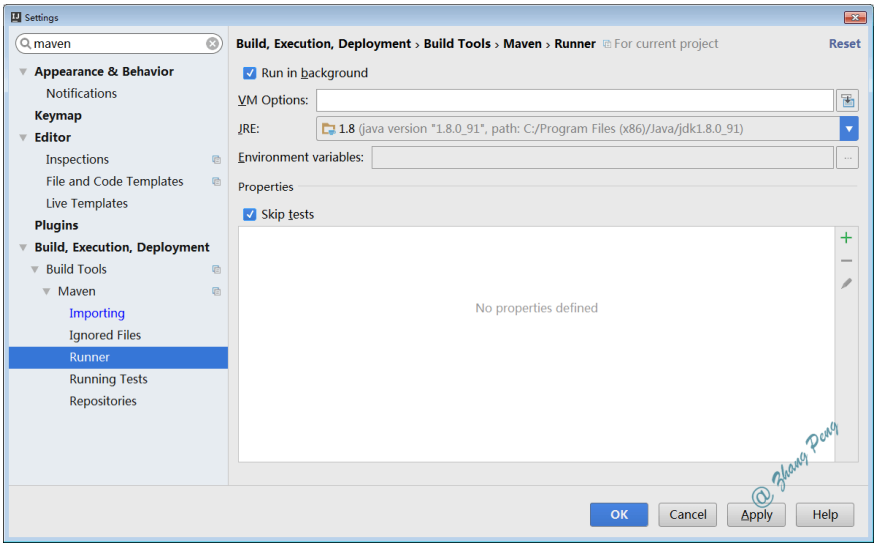


- 查看 **Settings > Maven** 的配置

JDK for importer 是否正确



Runner 是否正确



1.11. 重复引入依赖

在 Idea 中，选中 Module，使用 `Ctrl+Alt+Shift+U`，打开依赖图，检索是否存在重复引用的情况。

1.12. 如何引入本地 jar

问：有时候，需要引入在中央仓库找不到的 jar，但又想通过 maven 进行管理，那么应该如何做到呢？ 答：可以通过设置 dependency 的 scope 为 system 来引入本地 jar。 例：

- 将私有 jar 放置在 resouces/lib 下，然后以如下方式添加依赖：
- groupId 和 artifactId 可以按照 jar 包中的 package 设置，只要和其他 jar 不冲突即可。

```
<dependency>
    <groupId>xxx</groupId>
    <artifactId>xxx</artifactId>
    <version>1.0.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/src/main/resources/lib/xxx-6.
</dependency>
```

1.13. 如何排除依赖

问：如何排除依赖一个依赖关系？比方项目中使用的 libA 依赖某个库的 1.0 版。libB 以来某个库的 2.0 版，如今想统一使用 2.0 版，怎样去掉 1.0 版的依赖？

答：通过 exclusion 排除指定依赖即可。

例：

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.12</version>
    <optional>true</optional>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

2. 最佳实践

2.1. 通过 bom 统一管理版本

采用类似 `spring-boot-dependencies` 的方式统一管理依赖版本。

spring-boot-dependencies 的 pom.xml 形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLoc
<modelVersion>4.0.0</modelVersion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.1.9.RELEASE</version>
<packaging>pom</packaging>

<!-- 省略 -->

<!-- 依赖包版本管理 -->
<dependencyManagement>
    <dependencies>
    <!-- 省略 -->
    </dependencies>
</dependencyManagement>

<build>
<!-- 插件版本管理 -->
<pluginManagement>
    <plugins>
    <!-- 省略 -->
    </plugins>
</pluginManagement>
</build>
</project>
```

其他项目引入 spring-boot-dependencies 来管理依赖版本的方式：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-10-11 14:56:32

- Maven 教程之发布 jar 到私服或中央仓库
  - 发布 jar 包到中央仓库
    - 在 Sonatype 创建 Issue
    - 使用 GPG 生成公私钥对
    - Maven 配置
    - 部署和发布
  - 部署 maven 私服
    - 下载安装 Nexus
    - 启动停止 Nexus
    - 使用 Nexus
  - 参考资料

Maven 教程之发布 jar 到私服或中央仓库

> 📓 本文已归档到：「blog」

- 发布 jar 包到中央仓库
  - 在 Sonatype 创建 Issue
  - 使用 GPG 生成公私钥对
  - Maven 配置
  - 部署和发布
- 部署 maven 私服
  - 下载安装 Nexus
  - 启动停止 Nexus
  - 使用 Nexus
- 参考资料

发布 jar 包到中央仓库

> 为了避免重复造轮子，相信每个 Java 程序员都想打造自己的脚手架或工具包（自己定制的往往才是最适合自己的）。那么如何将自己的脚手架发布到中央仓库呢？下面我们将一步步来实现。

在 Sonatype 创建 Issue

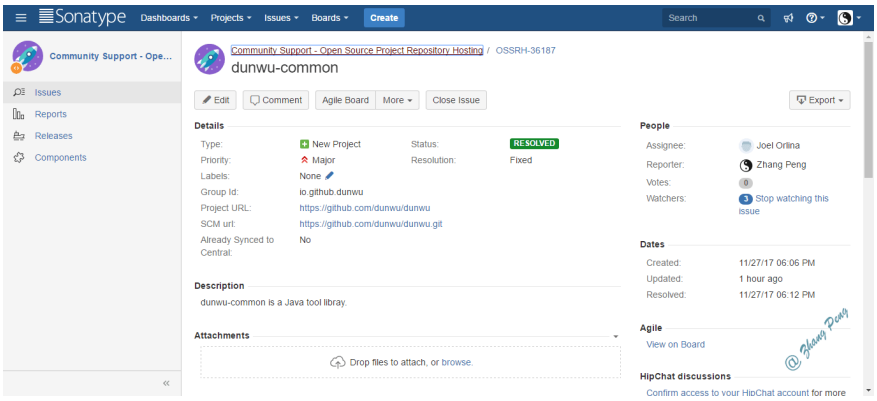
（1）注册 Sonatype 账号

发布 Java 包到 Maven 中央仓库，首先需要在 Sonatype 网站创建一个工单(Issues)。

第一次使用这个网站的时候需要注册自己的帐号（这个帐号和密码需要记住，后面会用到）。

（2）创建 Issue

注册账号成功后，根据你 Java 包的功能分别写上 `Summary` 、 `Description` 、 `Group Id` 、 `SCM url` 以及 `Project URL` 等必要信息，可以参见我之前创建的 Issue：OSSRH-36187。



创建完之后需要等待 Sonatype 的工作人员审核处理，审核时间还是很快的，我的审核差不多等待了两小时。当 Issue 的 Status 变为 `RESOLVED` 后，就可以进行下一步操作了。

> 说明：如果你的 Group Id 填写的是自己的网站（我的就是这种情况），Sonatype 的工作人员会询问你那个 Group Id 是不是你的域名，你只需要在上面回答是就行，然后就会通过审核。

使用 GPG 生成公私钥对

（1）安装 Gpg4win

Windows 系统，可以下载 Gpg4win 软件来生成密钥对。

Gpg4win 下载地址

安装后，执行命令 gpg --version 检查是否安装成功。

```
C:\Program Files (x86)\GnuPG\bin>gpg --version
gpg (GnuPG) 2.2.10
libgcrypt 1.8.3
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/lice
This is free software: you are free to change and redistribute i
There is NO WARRANTY, to the exdunwu permitted by law.

Home: C:/Users/Administrator/AppData/Roaming/gnupg
Supported algorithms:
Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFIS
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

（2）生成密钥对

执行命令 `gpg --gen-key`

```
C:\Program Files (x86)\GnuPG\bin>gpg --gen-key
gpg (GnuPG) 2.2.10; Copyright (C) 2018 Free Software Foundation,
This is free software: you are free to change and redistribute i
There is NO WARRANTY, to the exdunwu permitted by law.

Note: Use "gpg --full-generate-key" for a full featured key gene

GnuPG needs to construct a user ID to identify your key.

Real name: Zhang Peng
Email address: forbreak@163.com
You selected this USER-ID:
    "Zhang Peng <forbreak@163.com>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit? O
```

说明：按照提示，依次输入用户名、邮箱。

（3）查看公钥

```
C:\Program Files (x86)\GnuPG\bin>gpg --list-keys

gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0
gpg: next trustdb check due at 2020-11-05
C:/Users/Administrator/AppData/Roaming/gnupg/pubring.kbx
-------------------------------------------------------
pub   rsa2048 2018-11-06 [SC] [expires: 2020-11-06]
      E4CE537A3803D49C35332221A306519BAFF57F60
uid           [ultimate] forbreak <forbreak@163.com>
sub   rsa2048 2018-11-06 [E] [expires: 2020-11-06]
```

> 说明：其中，E4CE537A3803D49C35332221A306519BAFF57F60 就是公钥

（4）将公钥发布到 PGP 密钥服务器

执行 `gpg --keyserver hkp://pool.sks-keyservers.net --send-keys` 发布公钥：

```
C:\Program Files (x86)\GnuPG\bin>gpg --keyserver hkp://pool.sks-
gpg: sending key A306519BAFF57F60 to hkp://pool.sks-keyservers.n
```

> 注意：有可能出现 gpg: keyserver receive failed: No dat 错误，等大约 30 分钟后再执行就不会报错了。

（5）查看公钥是否发布成功

执行 `gpg --keyserver hkp://pool.sks-keyservers.net --recv-keys` 查看公钥是否发布成功。

```
C:\Program Files (x86)\GnuPG\bin>gpg --keyserver hkp://pool.sks-
gpg: key A306519BAFF57F60: "forbreak <forbreak@163.com>" not cha
gpg: Total number processed: 1
gpg:              unchanged: 1
```

Maven 配置

完成了前两个章节的准备工作，就可以将 jar 包上传到中央仓库了。当然了，我们还要对 maven 做一些配置。

settings.xml 配置

一份完整的 settings.xml 配置示例如下：

```xml
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sche

  <pluginGroups>
    <pluginGroup>org.sonatype.plugins</pluginGroup>
  </pluginGroups>

  <!-- 用户名、密码就是 Sonatype 账号、密码 -->
  <servers>
    <server>
      <id>sonatype-snapshots</id>
      <username>xxxxxx</username>
      <password>xxxxxx</password>
    </server>
    <server>
      <id>sonatype-staging</id>
      <username>xxxxxx</username>
      <password>xxxxxx</password>
    </server>
  </servers>

  <!-- 使用 aliyun maven 仓库加速下载 -->
  <mirrors>
    <mirror>
      <id>nexus-aliyun</id>
      <mirrorOf>*</mirrorOf>
      <name>Aliyun</name>
      <url>http://maven.aliyun.com/nexus/condunwu/groups/public<
    </mirror>
  </mirrors>

  <!-- gpg 的密码，注意，这里不是指公钥 -->
  <profiles>
    <profile>
      <id>sonatype</id>
      <properties>
        <gpg.executable>C:/Program Files (x86)/GnuPG/bin/gpg.exe
        <gpg.passphrase>xxxxxx</gpg.passphrase>
      </properties>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>sonatype</activeProfile>
  </activeProfiles>
</settings>
```

pom.xml 配置

（1）添加 licenses、scm、developers 配置：

```
<licenses>
  <license>
    <name>The Apache Software License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
  </license>
</licenses>

<developers>
  <developer>
    <name>xxxxxx</name>
    <email>forbreak@163.com</email>
    <url>https://github.com/dunwu</url>
  </developer>
</developers>

<scm>
  <url>https://github.com/dunwu/dunwu</url>
  <connection>git@github.com:dunwu/dunwu.git</connection>
  <developerConnection>https://github.com/dunwu</developerConnec
</scm>
```

（2）添加 distributionManagement 配置

```
<distributionManagement>
  <snapshotRepository>
    <id>sonatype-snapshots</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots
  </snapshotRepository>
  <repository>
    <id>sonatype-staging</id>
    <url>https://oss.sonatype.org/service/local/staging/deploy/m
  </repository>
</distributionManagement>
```

> 说明：`<snapshotRepository>` 指定的是 snapshot 仓库地址；`<repository>` 指定的是 staging （正式版）仓库地址。需要留意的是，这里的 id 需要和 settings.xml 中的 `<server>` 的 id 保持一致。

（3）添加 profiles 配置

```
<profiles>
 <profile>
   <id>sonatype</id>
   <build>
     <plugins>
       <plugin>
         <groupId>org.sonatype.plugins</groupId>
         <artifactId>nexus-staging-maven-plugin</artifactId>
         <version>1.6.7</version>
         <extensions>true</extensions>
         <configuration>
           <serverId>sonatype-snapshots</serverId>
           <nexusUrl>https://oss.sonatype.org/</nexusUrl>
           <autoReleaseAfterClose>true</autoReleaseAfterClose>
         </configuration>
       </plugin>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-javadoc-plugin</artifactId>
         <version>3.0.1</version>
         <configuration>
           <failOnError>false</failOnError>
           <quiet>true</quiet>
         </configuration>
         <executions>
           <execution>
             <id>attach-javadocs</id>
             <goals>
               <goal>jar</goal>
             </goals>
           </execution>
         </executions>
       </plugin>

       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-gpg-plugin</artifactId>
         <version>1.6</version>
         <executions>
           <execution>
             <id>sign-artifacts</id>
             <phase>verify</phase>
             <goals>
               <goal>sign</goal>
             </goals>
           </execution>
         </executions>
       </plugin>
```

```
      </plugins>
    </build>
  </profile>
</profiles>
```

部署和发布

按照上面的步骤配置完后，一切都已经 OK。

此时，使用 `mvn clean deploy -P sonatype` 命令就可以发布 jar 包到中央仓库了：

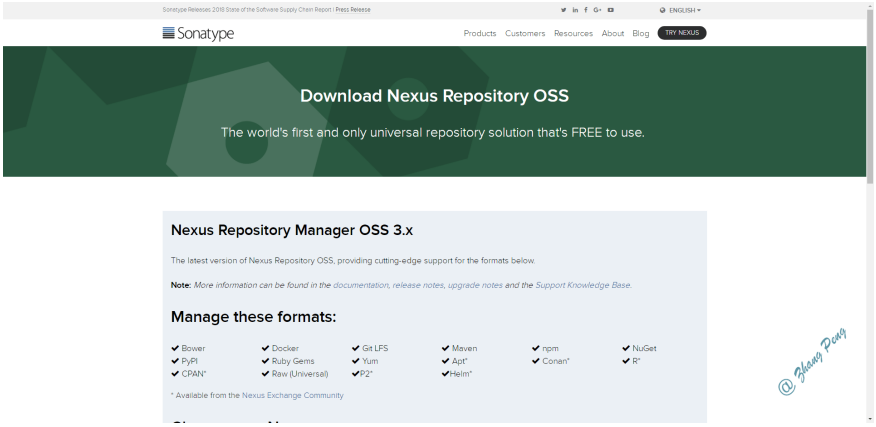
> 说明：-P 参数后面的 sonatype 需要和 pom.xml 中 `<profile>` 的 id 保持一致，才能激活 profile。

部署 maven 私服

> 工作中，Java 程序员开发的商用 Java 项目，一般不想发布到中央仓库，使得人人尽知。这时，我们就需要搭建私服，将 maven 服务器部署在公司内部网络，从而避免 jar 包流传出去。怎么做呢，让我们来一步步学习吧。

下载安装 Nexus

进入官方下载地址，选择合适版本下载。



本人希望将 Nexus 部署在 Linux 机器，所以选用的是 Unix 版本。

这里，如果想通过命令方式直接下载（比如用脚本安装），可以在官方历史发布版本页面中找到合适版本，然后执行以下命令：

```
wget -O /opt/maven/nexus-unix.tar.gz http://download.sonatype.co
tar -zxf nexus-unix.tar.gz
```

解压后，有两个目录：

- nexus-3.13.0-01 - 包含了 Nexus 运行所需要的文件。是 Nexus 运行必须的。
- sonatype-work - 包含了 Nexus 生成的配置文件、日志文件、仓库文件等。当我们需要备份 Nexus 的时候默认备份此目录即可。

启动停止 Nexus

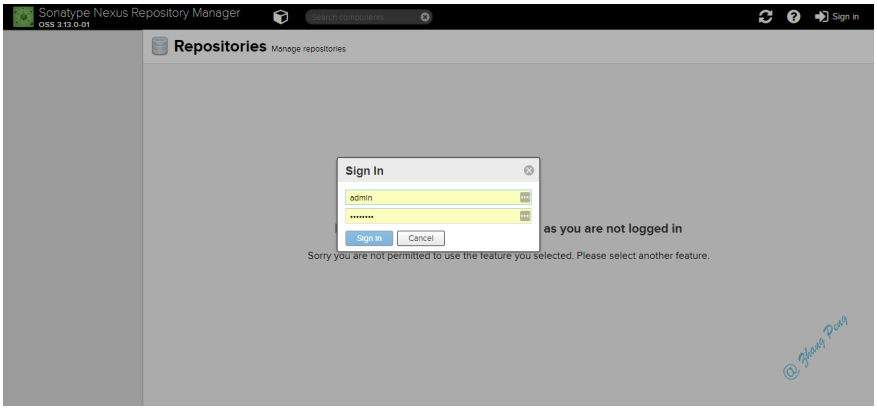
进入 nexus-3.13.0-01/bin 目录，有一个可执行脚本 nexus。

执行 `./nexus` ，可以查看允许执行的参数，如下所示，含义可谓一目了然：

```
$ ./nexus
Usage: ./nexus {start|stop|run|run-redirect|status|restart|force
```

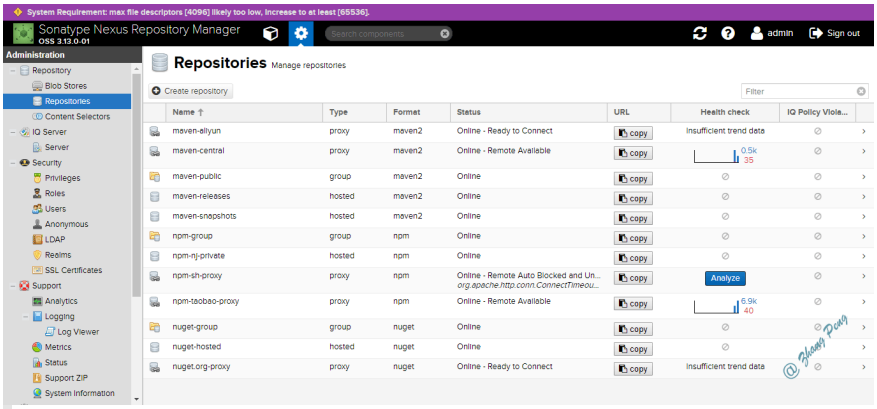
- 启动 nexus - `./nexus start`
- 停止 nexus -

启动成功后，在浏览器中访问 `http://<ip>:8081` ，欢迎页面如下图所示：



点击右上角 Sign in 登录，默认用户名/密码为：admin/admin123。

有必要提一下的是，在 Nexus 的 Repositories 管理页面，展示了可用的 maven 仓库，如下图所示：



说明：

- maven-central - maven 中央库（如果没有配置 mirror，默认就从这里下载 jar 包），从 https://repo1.maven.org/maven2/ 获取资源
- maven-releases - 存储私有仓库的发行版 jar 包
- maven-snapshots - 存储私有仓库的快照版（调试版本） jar 包
- maven-public - 私有仓库的公共空间，把上面三个仓库组合在一起对外提供服务，在本地 maven 基础配置 settings.xml 中使用。

使用 Nexus

如果要使用 Nexus，还必须在 settings.xml 和 pom.xml 中配置认证信息。

配置 settings.xml

一份完整的 `settings.xml`：

```xml
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sche
  <pluginGroups>
    <pluginGroup>org.sonatype.plugins</pluginGroup>
  </pluginGroups>

  <!-- Maven 私服账号信息 -->
  <servers>
    <server>
      <id>releases</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
    <server>
      <id>snapshots</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
  </servers>

  <!-- jar 包下载地址 -->
  <mirrors>
    <mirror>
      <id>public</id>
      <mirrorOf>*</mirrorOf>
      <url>http://10.255.255.224:8081/repository/maven-public/</
    </mirror>
  </mirrors>

  <profiles>
    <profile>
      <id>zp</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://central</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
```

```
          <id>central</id>
          <url>http://central</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>zp</activeProfile>
  </activeProfiles>
</settings>
```

配置 pom.xml

在 pom.xml 中添加如下配置：

```
  <distributionManagement>
    <repository>
      <id>releases</id>
      <name>Releases</name>
      <url>http://10.255.255.224:8081/repository/maven-releases<
    </repository>
    <snapshotRepository>
      <id>snapshots</id>
      <name>Snapshot</name>
      <url>http://10.255.255.224:8081/repository/maven-snapshots
    </snapshotRepository>
  </distributionManagement>
```

> 注意：
>
> - `<repository>` 和 `<snapshotRepository>` 的 id 必须和 `settings.xml` 配置文件中的 `<server>` 标签中的 id 匹配。
> - `<url>` 标签的地址需要和 maven 私服的地址匹配。

执行 maven 构建

如果要使用 settings.xml 中的私服配置，必须通过指定 `-P zp` 来激活 profile。

示例：

```
## 编译并打包 maven 项目
$ mvn clean package -Dmaven.skip.test=true -P zp

## 编译并上传 maven 交付件（jar 包）
$ mvn clean deploy -Dmaven.skip.test=true -P zp
```

参考资料

- https://www.jianshu.com/p/8c3d7fb09bce
- http://www.ruanyifeng.com/blog/2013/07/gpg.html
- https://www.cnblogs.com/hoobey/p/6102382.html
- https://blog.csdn.net/wzygis/article/details/49276779
- https://blog.csdn.net/clj198606061111/article/details/52200928

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-27 15:29:01

- Maven 插件之代码检查
  - maven-checkstyle-plugin
    - 定义 checkstyle.xml
    - 配置 pom.xml
  - maven-pmd-plugin
    - 配置 pom.xml
  - 参考资料

Maven 插件之代码检查

- maven-checkstyle-plugin
  - 定义 checkstyle.xml
  - 配置 pom.xml
- maven-pmd-plugin
  - 配置 pom.xml
- 参考资料

maven-checkstyle-plugin

> **maven-checkstyle-plugin**，用于检测代码中不符合规范的地方。

定义 checkstyle.xml

```
<!DOCTYPE module PUBLIC
  "-//Checkstyle//DTD Checkstyle Configuration 1.3//EN"
  "https://checkstyle.org/dtds/configuration_1_3.dtd">

<!-- Generated by RHY @will_awoke -->

<module name="Checker">

  <property name="charset" value="UTF-8"/>
  <property name="severity" value="warning"/>

  <!-- Checks for Size Violations.  -->
  <!-- 检查文件的长度（行） default max=2000 -->
  <module name="FileLength">
    <property name="max" value="2500"/>
  </module>

  <!-- Checks that property files contain the same keys. -->
  <!-- 检查**.properties配置文件 是否有相同的key
  <module name="Translation">
  </module>
  -->

  <module name="TreeWalker">

    <!-- Checks for imports    -->
    <!-- 必须导入类的完整路径，即不能使用*导入所需的类 -->
    <module name="AvoidStarImport"/>

    <!-- 检查是否从非法的包中导入了类 illegalPkgs: 定义非法的包名称-
    <module name="IllegalImport"/> <!-- defaults to sun.* packag

    <!-- 检查是否导入了不必显示导入的类-->
    <module name="RedundantImport"/>

    <!-- 检查是否导入的包没有使用-->
    <module name="UnusedImports"/>

    <!-- Checks for whitespace
    <module name="EmptyForIteratorPad"/>
    <module name="MethodParamPad"/>
    <module name="NoWhitespaceAfter"/>
    <module name="NoWhitespaceBefore"/>
    <module name="OperatorWrap"/>
    <module name="ParenPad"/>
    <module name="TypecastParenPad"/>
    <module name="WhitespaceAfter"/>
    <module name="WhitespaceAround"/>
```

```
-->

<!-- 检查类和接口的javadoc 默认不检查author 和version tags
  authorFormat: 检查author标签的格式
        versionFormat: 检查version标签的格式
        scope: 可以检查的类的范围，例如：public只能检查public修
        excludeScope: 不能检查的类的范围，例如：public，public
        tokens: 该属性适用的类型，例如：CLASS_DEF,INTERFACE_DE
<module name="JavadocType">
  <property name="authorFormat" value="\S"/>
  <property name="scope" value="protected"/>
  <property name="tokens" value="CLASS_DEF,INTERFACE_DEF"/>
</module>

<!-- 检查方法的javadoc的注释
        scope: 可以检查的方法的范围，例如：public只能检查public
        allowMissingParamTags: 是否忽略对参数注释的检查
        allowMissingThrowsTags: 是否忽略对throws注释的检查
        allowMissingReturnTag: 是否忽略对return注释的检查 -->
<module name="JavadocMethod">
  <property name="scope" value="private"/>
  <property name="allowMissingParamTags" value="false"/>
  <property name="allowMissingThrowsTags" value="false"/>
  <property name="allowMissingReturnTag" value="false"/>
  <property name="tokens" value="METHOD_DEF"/>
  <property name="allowUndeclaredRTE" value="true"/>
  <property name="allowThrowsTagsForSubclasses" value="true"
  <!--允许get set 方法没有注释-->
  <property name="allowMissingPropertyJavadoc" value="true"/
</module>

<!-- 检查类变量的注释
        scope: 检查变量的范围，例如：public只能检查public修饰的
<module name="JavadocVariable">
  <property name="scope" value="private"/>
</module>

<!--option: 定义左大括号'{'显示位置，eol在同一行显示，nl在下一行
  maxLineLength: 大括号'{'所在行行最多容纳的字符数
  tokens: 该属性适用的类型，例：CLASS_DEF,INTERFACE_DEF,METHOD
<module name="LeftCurly">
  <property name="option" value="nl"/>
</module>

<!-- NeedBraces 检查是否应该使用括号的地方没有加括号
  tokens: 定义检查的类型 -->
<module name="NeedBraces"/>
```

```
<!-- Checks the placement of right curly braces ('}') for  e
  option: 右大括号是否单独一行显示
  tokens: 定义检查的类型  -->
<module name="RightCurly">
  <property name="option" value="alone"/>
</module>

<!-- 检查在重写了equals方法后是否重写了hashCode方法 -->
<module name="EqualsHashCode"/>

<!--  Checks for illegal instantiations where a factory meth
  Rationale: Depending on the project, for some classes it m
  A simple example is the java.lang.Boolean class. In order
  Some extremely performance sensitive projects may require
<module name="IllegalInstantiation">
  <property name="classes" value="java.lang.Boolean"/>
</module>

<!-- Checks for Naming Conventions.   命名规范   -->
<!-- local, final variables, including catch parameters -->
<module name="LocalFinalVariableName"/>

<!-- local, non-final variables, including catch parameters-
<module name="LocalVariableName"/>

<!-- static, non-final fields -->
<module name="StaticVariableName">
  <property name="format" value="(^[A-Z0-9_]{0,19}$)"/>
</module>

<!-- packages -->
<module name="PackageName">
  <property name="format" value="^[a-z]+(\.[a-z][a-z0-9]*)*$
</module>

<!-- classes and interfaces -->
<module name="TypeName">
  <property name="format" value="(^[A-Z][a-zA-Z0-9]{0,19}$)"
</module>

<!-- methods -->
<module name="MethodName">
  <property name="format" value="(^[a-z][a-zA-Z0-9]{0,19}$)"
</module>

<!-- non-static fields -->
<module name="MemberName">
  <property name="format" value="(^[a-z][a-z0-9][a-zA-Z0-9]{
```

```
    </module>

    <!-- parameters -->
    <module name="ParameterName">
      <property name="format" value="(^[a-z][a-zA-Z0-9_]{0,19}$)
    </module>

    <!-- constants (static,  final fields) -->
    <module name="ConstantName">
      <property name="format" value="(^[A-Z0-9_]{0,19}$)"/>
    </module>

    <!-- 代码缩进   -->
    <module name="Indentation">
    </module>

    <!-- Checks for redundant exceptions declared in throws clau
      检查是否抛出了多余的异常
    <module name="RedundantThrows">
        <property name="logLoadErrors" value="true"/>
        <property name="suppressLoadErrors" value="true"/>
    </module>
    -->

    <!--  Checks for overly complicated boolean expressions. Cur
      检查boolean值是否冗余的地方
      Rationale: Complex boolean logic makes code hard to unders
    <module name="SimplifyBooleanExpression"/>

    <!--  Checks for overly complicated boolean return statement
       检查是否存在过度复杂的boolean返回值
       if (valid())
          return false;
       else
          return true;
       could be written as
          return !valid();
       The Idea for this Check has been shamelessly stolen from
    <module name="SimplifyBooleanReturn"/>

    <!-- Checks that a class which has only private constructors
    <module name="FinalClass"/>

    <!--  Make sure that utility classes (classes that contain o
      确保Utils类（只提供static方法和属性的类）没有public构造器。
      Rationale: Instantiating utility classes does not make sen
      If you make the constructor protected you may want to cons
      public class StringUtils // not final to allow subclassing
```

```
    {
        protected StringUtils() {
            throw new UnsupportedOperationException(); // prev
        }
        public static int count(char c, String s) {
            // ...
        }
    }
<module name="HideUtilityClassConstructor"/>
-->

 <!--  Checks visibility of class members. Only static final
   检查class成员属性可见性。只有static final 修饰的成员是可以publ
    Public members are not flagged if the name matches the pu
    Rationale: Enforce encapsulation. 强制封装 -->
 <module name="VisibilityModifier"/>

 <!-- 每一行只能定义一个变量 -->
 <module name="MultipleVariableDeclarations">
 </module>

 <!-- Checks the style of array type definitions. Some like J
   检查再定义数组时，采用java风格还是c风格，例如：int[] num是java
 <module name="ArrayTypeStyle">
 </module>

 <!-- Checks that there are no "magic numbers", where a magic
 <module name="MagicNumber">
 </module>
 -->

 <!-- A check for TODO: comments. Actually it is a generic re
    检查是否存在TODO（待处理） TODO是javaIDE自动生成的。一般代码写
  -->
 <module name="TodoComment"/>

 <!--  Checks that long constants are defined with an upper e
   检查是否在long类型是否定义了大写的L.字母小写l和数字1（一）很相似
   looks a lot like 1. -->
 <module name="UpperEll"/>

 <!--  Checks that switch statement has "default" clause. 检查
    Rationale: It's usually a good idea to introduce a defaul
    Even if the developer is sure that all currently possible
     e.g. by using an assertion. This way the code is protect
 <module name="MissingSwitchDefault"/>

 <!--检查switch中case后是否加入了跳出语句，例如：return、break、
```

```
    <module name="FallThrough"/>

    <!-- Checks the number of parameters of a method or construc
    <module name="ParameterNumber">
      <property name="max" value="5"/>
    </module>

    <!-- 每行字符数 -->
    <module name="LineLength">
      <property name="max" value="200"/>
    </module>

    <!-- Checks for long methods and constructors. max default 1
    <module name="MethodLength">
      <property name="max" value="300"/>
    </module>

    <!-- ModifierOrder 检查修饰符的顺序，默认是 public,protected,p
    <module name="ModifierOrder">
    </module>

    <!-- 检查是否有多余的修饰符，例如：接口中的方法不必使用public、ab
    <module name="RedundantModifier">
    </module>

    <!--- 字符串比较必须使用 equals() -->
    <module name="StringLiteralEquality">
    </module>

    <!-- if-else嵌套语句个数 最多4层 -->
    <module name="NestedIfDepth">
      <property name="max" value="3"/>
    </module>

    <!-- try-catch 嵌套语句个数 最多2层 -->
    <module name="NestedTryDepth">
      <property name="max" value="2"/>
    </module>

    <!-- 返回个数 -->
    <module name="ReturnCount">
      <property name="max" value="5"/>
      <property name="format" value="^$"/>
    </module>

  </module>
</module>
```

配置 pom.xml

```
<project>
    ...
    <properties>
        <checkstyle.config.location>config/maven_checks.xml</che
    </properties>
    ...
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>3.0</version>
                <executions>
                    <execution>
                        <!-- 绑定pmd:pmd到validate生命周期，在vali
                        <id>validate</id>
                        <phase>validate</phase>
                        <configuration>
                            <!-- 配置文件的路径，在style文件夹下 --
                            <configLocation>style/checkstyle.xml
                            <encoding>UTF-8</encoding>
                            <consoleOutput>true</consoleOutput>
                            <failsOnError>true</failsOnError>
                            <includeTestSourceDirectory>false</i
                        </configuration>
                        <goals>
                            <goal>check</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jxr-plugin</artifactId>
                <version>2.3</version>
            </plugin>
        </plugins>
    </reporting>
    ...
</project>
```

其中可以修改使用的检查规则文件路径，插件默认提供了四个规则文件可以直接使用，无需手动下载：

- config/sun_checks.xml - Sun Microsystems Definition (default).
- config/maven_checks.xml - Maven Development Definitions.
- config/turbine_checks.xml - Turbine Development Definitions.
- config/avalon_checks.xml - Avalon Development Definitions.

配置好后，可以执行 `mvn clean checkstyle:check` 检查代码。

maven-pmd-plugin

> maven-pmd-plugin 是阿里编程规范检查插件。

配置 pom.xml

参考 https://github.com/alibaba/p3c/blob/master/p3c-pmd/pom.xml 配置

```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <sourceEncoding>${project.build.sourceEncoding}</sourc
        <targetJdk>${maven.compiler.target}</targetJdk>
        <printFailingErrors>true</printFailingErrors>
        <rulesets>
          <ruleset>rulesets/java/ali-comment.xml</ruleset>
          <ruleset>rulesets/java/ali-concurrent.xml</ruleset>
          <ruleset>rulesets/java/ali-constant.xml</ruleset>
          <ruleset>rulesets/java/ali-exception.xml</ruleset>
          <ruleset>rulesets/java/ali-flowcontrol.xml</ruleset>
          <ruleset>rulesets/java/ali-naming.xml</ruleset>
          <ruleset>rulesets/java/ali-oop.xml</ruleset>
          <ruleset>rulesets/java/ali-orm.xml</ruleset>
          <ruleset>rulesets/java/ali-other.xml</ruleset>
          <ruleset>rulesets/java/ali-set.xml</ruleset>
        </rulesets>
        <printFailingErrors>true</printFailingErrors>
      </configuration>
      <executions>
        <execution>
          <phase>verify</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
      <dependencies>
        <dependency>
          <groupId>com.alibaba.p3c</groupId>
          <artifactId>p3c-pmd</artifactId>
          <version>2.0.0</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
```

配置好后，可以执行 `mvn clean pmd:check` 检查代码。

参考资料

- https://maven.apache.org/plugins/maven-checkstyle-plugin/

- https://maven.apache.org/jxr/maven-jxr-plugin/
- https://www.jianshu.com/p/557b975ae40d
- 阿里巴巴编程规范
  - https://github.com/alibaba/p3c
  - https://github.com/alibaba/p3c/blob/master/p3c-pmd/pom.xml

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-27 16:25:48

- Ant 简易教程
  - 简介
  - 下载和安装
    - 下载
    - 配置环境变量
    - 验证
  - 例子
  - 关键元素
    - Project 元素
    - Target 元素
    - Task 元素
    - Property 元素
    - extension-point 元素
  - 参考资料

Ant 简易教程

- 简介
- 下载和安装
  - 下载
  - 配置环境变量
  - 验证
- 例子
- 关键元素
  - Project 元素
  - Target 元素
  - Task 元素
  - Property 元素
  - extension-point 元素
- 参考资料

简介

`Apache Ant` 是一个将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，大多用于 Java 环境中的软件开发。由 Apache 软件基金会所提供。

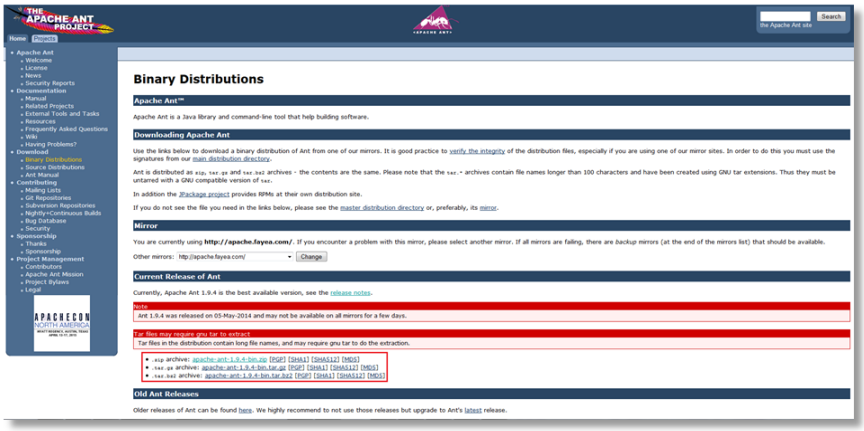
Ant 是纯 Java 语言编写的，所以具有很好的跨平台性。

下载和安装

下载

ant 的官方下载地址：http://ant.apache.org/bindownload.cgi

进入页面后，在下图的红色方框中可以下载最新版本。笔者下载的版本是 **apache-ant-1.9.4**。

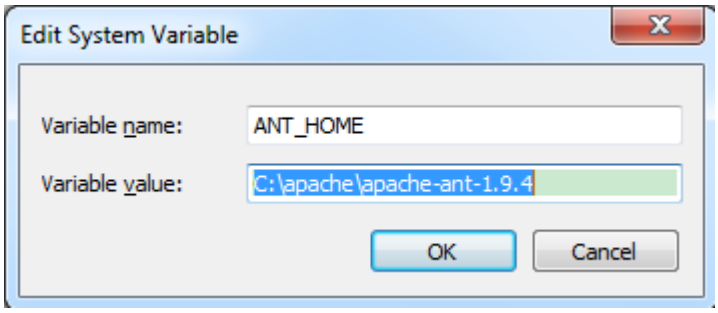


配置环境变量

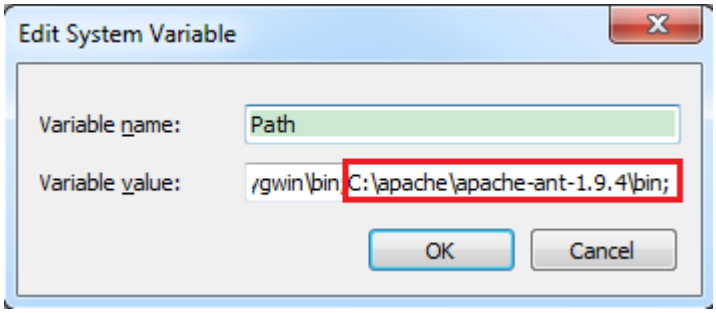
配置环境变量（我的电脑 -> 属性 -> 高级 -> 环境变量）。

设置 ant 环境变量：

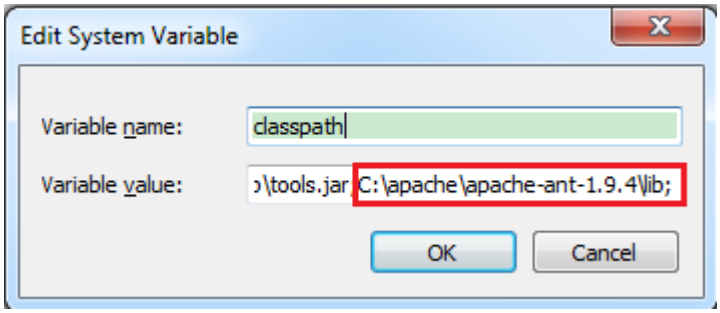
ANT_HOME C:/ apache-ant-1.9.4



path C:/ apache-ant-1.9.4/bin



classpath C:/apache-ant-1.9.4/lib



验证

点击 开始 -> 运行 -> 输入 cmd

执行构建文件

输入如下命令：**ant**

如果出现如下内容，说明安装成功：

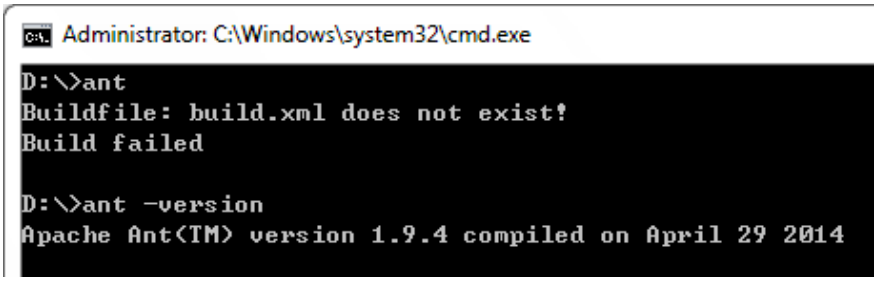
> Buildfile: build.xml does not exist! Build failed

注意：因为 ant 默认运行 build.xml 文件，这个文件需要我们创建。

如果不想命名为 build.xml，运行时可以使用 **ant -buildfile test.xml** 命令指明要运行的构建文件。

查看版本信息

输入 **ant -version**，可以查看版本信息。



但如果出现 'ant' 不是内部或外部命令，也不是可运行的程序或批处理文件，说明安装失败：（可以重复前述步骤，直至安装成功。）

例子

在安装和配置成功后，我们就可以使用 ant 了。

为了让读者对 ant 有一个直观的认识，首先以 Ant 官方手册上的一个简单例子做一个说明。

以下是一个 build.xml 文件的内容：

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile --
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

在这个 xml 文件中，有几个 target 标签，每个 target 对应一个执行目标。

我们将这个 build.xml 放在 D:\Temp\ant_test 路径下，然后在 dos 界面下进行测试。

ant init

```
D:\Temp\ant_test>ant init
Buildfile: D:\Temp\ant_test\build.xml

init:
    [mkdir] Created dir: D:\Temp\ant_test\build

BUILD SUCCESSFUL
Total time: 0 seconds
```

在 D:\Temp\ant_test 路径下创建了一个 build 目录，执行成功。

ant compile

```
D:\Temp\ant_test>ant compile
Buildfile: D:\Temp\ant_test\build.xml

init:

compile:
    [javac] D:\Temp\ant_test\build.xml:20: warning: 'includeantruntime' was not
set, defaulting to build.sysclasspath=last; set to false for repeatable builds

BUILD FAILED
D:\Temp\ant_test\build.xml:20: srcdir "D:\Temp\ant_test\src" does not exist!

Total time: 1 second
```

提示错误，原来是在 build.xml 的所在目录下找不到 src 目录。好的，我们直接创建一个 src 目录，然后再次尝试。这次，执行成功。

```
D:\Temp\ant_test>ant compile
Buildfile: D:\Temp\ant_test\build.xml

init:

compile:
    [javac] D:\Temp\ant_test\build.xml:20: warning: 'includeantruntime' was not
set, defaulting to build.sysclasspath=last; set to false for repeatable builds

BUILD SUCCESSFUL
Total time: 0 seconds
```

ant dist

```
D:\Temp\ant_test>ant dist
Buildfile: D:\Temp\ant_test\build.xml

init:

compile:
    [javac] D:\Temp\ant_test\build.xml:20: warning: 'includeantruntime' was not
set, defaulting to build.sysclasspath=last; set to false for repeatable builds

dist:
    [mkdir] Created dir: D:\Temp\ant_test\dist\lib
      [jar] Building MANIFEST-only jar: D:\Temp\ant_test\dist\lib\MyProject-2015
0417.jar

BUILD SUCCESSFUL
Total time: 0 seconds
```

在 D:\Temp\ant_test 路径下创建了一个 dist 目录，执行成功。

ant clean

```
D:\Temp\ant_test>ant clean
Buildfile: D:\Temp\ant_test\build.xml

clean:
   [delete] Deleting directory D:\Temp\ant_test\build
   [delete] Deleting directory D:\Temp\ant_test\dist

BUILD SUCCESSFUL
Total time: 0 seconds
```

清除创建的 build 和 dist 目录，执行成功。

一个细节

细心的读者，想必已经发现一个问题——在执行 ant compile 和 ant dist 命令的时候把前面的命令也执行了。这是为什么呢？

请留意一下 build.xml 中的内容。有部分 **target** 标签中含有 **depends** 关键字。

```
<target name="dist" depends="compile"
```

这表明，当前的 target 在执行时需要依赖其他的 target，必须先执行依赖的 target，然后再执行。

关键元素

Ant 的构件文件都是 XML 格式的。每个构件文件包含一个 project 元素和至少一个 target。

target 元素可以包含多个 task 元素。

Project 元素

project 元素是构建文件的根元素。

一个 project 元素可以有多个 target 元素，一个 target 元素可以有多个 task。

在上节的例子中，project 标签里有三个属性。

```
<project name="MyProject" default="dist" basedir=".">
```

name 属性，指示 project 元素的名字。例子中的名字就是 MyProject。

default 属性，指示这个 project 默认执行的 target。在本文的例子中，默认执行的 target 为 dist。

如果我们输入命令 ant 时，不指定 target 参数，默认会执行 dist 这个 target。

basedir 属性，指定根路径的位置。该属性没有指定时，使用 Ant 的构件文件的所在目录作为根目录。

Target 元素

target 元素是 task 的容器，也就是 Ant 的一个基本执行单元。

以上节例子中的 compile 来举例。

```
<target name="compile" depends="init" description="compile the s
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
</target>
```

这个 target 中出现了几个属性。

name 属性，指示 target 元素的名称。

这个属性在一个 project 元素中必须是唯一的。这很好理解，如果出现重复，Ant 就不知道具体该执行哪个 target 了。

depends 属性，指示依赖的 target，当前的 target 必须在依赖的 target 之后执行。

description 属性，是关于 target 的简短说明。

此外，还有其他几个未出现在构建文件中的属性。

if 属性，验证指定的属性是否存在，若不存在，所在 target 将不会被执行。

unless 属性，**正好和 if 属性相反**，验证指定的属性是否存在，若存在，所在 target 将不会被执行。****

extensionOf 属性，添加当前 target 到 **extension-point** 依赖列表。**——Ant1.8.0 新特性。**

> **extension-point** 元素和 target 元素十分类似，都可以指定依赖的 target。但是不同的是，extension-point 中不能包含任何 task。

请看以下实例：

```
<target name="create-directory-layout">
   ...
</target>
<extension-point name="ready-to-compile" depends="create-directo
<target name="compile" depends="ready-to-compile">
   ...
</target>
```

调用 **target** 顺序: create-directory-layout --> 'empty slot' --> compile

```
<target name="generate-sources" extensionOf="ready-to-compile">
   ...
</target>
```

调用 **target** 顺序: create-directory-layout --> generate-sources --> compile

onMissingExtensionPoint 属性：当无法找到一个 extension-point 时，target 尝试去做的动作("fail", "warn", "ignore")。——*Ant1.8.2 新特性*

Task 元素

task 是一段可以被执行的代码。

一个 task 可以有多个属性， 一个属性可以包含对一个 **property** 的引用。

task 的通常结构为

```
<name attribute1="value1" attribute2="value2" ... />
```

其中，name 是 task 的名字， attributeN 是属性名， valueN 是这个属性的值。

还是以 compile 做为例子：

```
<target name="compile" depends="init" description="compile the s
    <!-- Compile the java code from srcintosrcinto{build} -->
    <javac srcdir="${src}" destdir="${build}"/>
</target>
```

在 compile 这个 target 标签中包含了一个任务。

这个任务的动作是：执行 JAVA 编译，编译 src 下的代码，并把编译生成的文件放在 build 目录中。

常用 **task**

javac：用于编译一个或者多个 Java 源文件，通常需要 srcdir 和 destdir 两个属性，用于指定 Java 源文件的位置和编译后 class 文件的保存位置。

```
<javac srcdir="${src}" destdir="${build}" classpath="abc.jar" de
```

java：用于运行某个 Java 类，通常需要 classname 属性，用于指定需要运行哪个类。

```
<java classname="test.Main">
    <arg value="-h" />
    <classpath>
        <pathelement location="dist/test.jar" />
    </classpath>
</java>
```

jar：用于生成 JAR 包，通常需要指定 destfile 属性，用于指定所创建 JAR 包的文件名。除此之外，通常还应指定一个文件集，表明需要将哪些文件打包到 JAR 包里。

```
<jar jarfile="dist/lib/MyProject-dist/lib/MyProject-{DSTAMP}.jar
```

echo：输出某个字符串。

```
<echo message="Building to ${builddir}"/>
<echo>You are using version ${java.version} of Java! This messag
```

copy：用于复制文件或路径。

```
<copy todir="${builddir}/srccopy">
    <fileset dir="${srcdir}">
        <include name="**/*.java"/>
    </fileset>
    <filterset>
        <filter token="VERSION" value="${app.version}"/>
    </filterset>
</copy>
```

delete：用于删除文件或路径。

```
<copy todir="${builddir}/srccopy">
    <fileset dir="${srcdir}">
        <include name="**/*.java"/>
    </fileset>
    <filterset>
        <filter token="VERSION" value="${app.version}"/>
    </filterset>
</copy>
```

mkdir：用于创建文件夹。

```
<mkdir dir="${dist}/lib" />
```

move：用户移动文件和路径。

```
<move todir="some/new/dir">
    <fileset dir="my/src/dir">
        <include name="**/*.jar" />
        <exclude name="**/ant.jar" />
    </fileset>
</move>
```

Property 元素

Property 是对参数的定义。

project 的属性可以通过 property 元素来设定，也可在 Ant 之外设定。若要在外部引入某文件，例如 build.properties 文件，可以通过如下内容将其引入：。

property 元素可用作 task 的属性值。在 task 中是通过将属性名放在“${”和“}”之间，并放在 task 属性值的位置来实现的。

例如 complile 例子中，使用了前面定义的 src 作为源目录。

```
<javac srcdir="${src}" destdir="${build}"/>
```

Ant 提供了一些内置的属性，它能得到的系统属性的列表与 Java 文档中 System.getPropertis()方法得到的属性一致，这些系统属性可参考 sun 网站的说明。

extension-point 元素

和 target 元素十分类似，都可以指定依赖的 target。但是不同的是，extension-point 中不能包含任何 task。

——*Ant1.8.0 新增特性。*

在 target 元素中的例子里已提到过，不再赘述。

参考资料

- ant 官方手册
- http://www.blogjava.net/amigoxie/archive/2007/11/09/159413.html

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 17:21:39

- Intellij IDEA 使用指南
  - 快捷键
    - 核心快捷键
    - 快捷键分类
    - Intellij IDEA 官方快捷键表
  - 插件
  - 个性化
    - 颜色主题
  - 破解
  - 参考资料

Intellij IDEA 使用指南

快捷键

核心快捷键

IntelliJ IDEA 作为一个以快捷键为中心的 IDE，为大多数操作建议了键盘快捷键。在这个主题中，您可以找到最不可缺少的列表，使 IntelliJ IDEA 轻松实现第一步。

核心快捷键表：

操作	快捷键
根据名称查找操作	`Ctrl+Shift+A`
显示可用 意图操作 列表	`Alt+Enter`
切换视图 (Project,Structure, etc.).	`Alt+F1`
切换工具窗口和在编辑器中打开的文件	`Ctrl+Tab`
显示 导航栏.	`Alt+Home`
插入代码模板.	`Ctrl+J`
在周围插入代码模板.	`Ctrl+Alt+J`
Edit an item from the Project or another tree view.	`F4`
注释	`Ctrl+/` `Ctrl+Shift+/`
根据名称查找类或文件.	`Ctrl+N` `Ctrl+Shift+N`
拷贝当前行或指定的行.	`Ctrl+D`
增加或减少选中的表达式.	`Ctrl+W` and `Ctrl+Shift+W`
在当前文件查找或替换.	`Ctrl+F` `Ctrl+R`
在项目中或指定的目录中查找或替换	`Ctrl+Shift+F` `Ctrl+Shift+R`
全局搜索	双击 `Shift`
快速查看选中对象的引用.	`Ctrl+Shift+F7`
展开或折叠编辑器中的代码块.	`Ctrl+NumPad Plus` `Ctrl+NumPad -`
调用代码完成.	`Ctrl+Space`
智能声明完成.	`Ctrl+Shift+Enter`
智能补全代码	`Ctrl+Shift+Space`
显示可用的重构方法列表	`Ctrl+Shift+Alt+T`

快捷键分类

Tradition

快捷键	介绍
Ctrl + Z	撤销
Ctrl + Shift + Z	取消撤销
Ctrl + X	剪切
Ctrl + C	复制
Ctrl + S	保存
Tab	缩进
Shift + Tab	取消缩进
Shift + Home/End	选中光标到当前行头位置/行尾位置
Ctrl + Home/End	跳到文件头/文件尾

Editing

快捷键	介绍
Ctrl + Space	基础代码补全，默认在 Windows 系统上被输入法占用，需要进行修改，建议修改为 Ctrl + 逗号 （必备）
Ctrl + Alt + Space	类名自动完成
Ctrl + Shift + Enter	自动结束代码，行末自动添加分号 （必备）
Ctrl + P	方法参数提示显示
Ctrl + Q	光标所在的变量/类名/方法名等上面（也可以在提示补充的时候按），显示文档内容
Shift + F1	如果有外部文档可以连接外部文档
Ctrl + F1	在光标所在的错误代码处显示错误信息 （必备）
Alt + Insert	代码自动生成，如生成对象的 set/get 方法，构造函数，toString() 等 （必备）
Ctrl + O	选择可重写的方法
Ctrl + I	选择可继承的方法
Ctrl + Alt + T	对选中的代码弹出环绕选项弹出层 （必备）
Ctrl + /	注释光标所在行代码，会根据当前不同文件类型使用不同的注释符号 （必备）
Ctrl + Shift + /	代码块注释 （必备）
Ctrl + W	递进式选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展选中范围 （必备）
Ctrl + Shift + W	递进式取消选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展取消选中范围 （必备）
Alt + Q	弹出一个提示，显示当前类的声明/上下文信息
Alt + Enter	IntelliJ IDEA 根据光标所在问题，提供快速修复选择，光标放在的位置不同提示的结果也不同 （必备）
Ctrl + Alt + L	格式化代码，可以对当前文件和整个包目录使用 （必备）
Ctrl + Alt + O	优化导入的类，可以对当前文件和整个包目录使用 （必备）
Ctrl + Alt + I	光标所在行 或 选中部分进行自动代码缩进，有点类似格式化
Ctrl + Shift + C	复制当前文件磁盘路径到剪贴板 （必备）

快捷键	介绍
Ctrl + Shift + V	弹出缓存的最近拷贝的内容管理器弹出层
Ctrl + Alt + Shift + C	复制参考信息
Ctrl + Alt + Shift + V	无格式黏贴 （必备）
Ctrl + D	复制光标所在行 或 复制选择内容，并把复制内容插入光标位置下面 （必备）
Ctrl + Y	删除光标所在行 或 删除选中的行 （必备）
Ctrl + Shift + J	自动将下一行合并到当前行末尾 （必备）
Shift + Enter	开始新一行。光标所在行下空出一行，光标定位到新行位置 （必备）
Ctrl + Shift + U	对选中的代码进行大/小写轮流转换 （必备）
Ctrl + Shift +]/[	选中从光标所在位置到它的底部/顶部的中括号位置 （必备）
Ctrl + Delete	删除光标后面的单词或是中文句 （必备）
Ctrl + BackSpace	删除光标前面的单词或是中文句 （必备）
Ctrl + +/-	展开/折叠代码块
Ctrl + Shift + +/-	展开/折叠所有代码 （必备）
Ctrl + F4	关闭当前编辑文件
Ctrl + Shift + Up/Down	光标放在方法名上，将方法移动到上一个/下一个方法前面，调整方法排序 （必备）
Alt + Shift + Up/Down	移动光标所在行向上移动/向下移动 （必备）
Ctrl + Shift + 左键单击	把光标放在某个类变量上，按此快捷键可以直接定位到该类中 （必备）
Alt + Shift + 左键双击	选择被双击的单词/中文句，按住不放，可以同时选择其他单词/中文句 （必备）
Ctrl + Shift + T	对当前类生成单元测试类，如果已经存在的单元测试类则可以进行选择 （必备）

Search/Replace

快捷键	介绍
Double Shift	弹出 Search Everywhere 弹出层
F3	在查找模式下，定位到下一个匹配处
Shift + F3	在查找模式下，查找匹配上一个
Ctrl + F	在当前文件进行文本查找 （必备）
Ctrl + R	在当前文件进行文本替换 （必备）
Ctrl + Shift + F	根据输入内容查找整个项目 或 指定目录内文件 （必备）
Ctrl + Shift + R	根据输入内容替换对应内容，范围为整个项目 或 指定目录内文件 （必备）

Usage Search

快捷键	介绍
Alt + F7	查找光标所在的方法/变量/类被调用的地方
Ctrl + Alt + F7	显示使用的地方。寻找被该类或是变量被调用的地方，用弹出框的方式找出来
Ctrl + Shift + F7	高亮显示所有该选中文本，按 Esc 高亮消失 （必备）

Compile and Run

快捷键	介绍
Ctrl + F9	执行 Make Project 操作
Ctrl + Shift + F9	编译选中的文件/包/Module
Shift + F9	Debug
Shift + F10	Run
Alt + Shift + F9	弹出 Debug 的可选择菜单
Alt + Shift + F10	弹出 Run 的可选择菜单

Debugging

快捷键	介绍
F7	在 Debug 模式下，进入下一步，如果当前行断点是一个方法，则进入当前方法体内，如果该方法体还有方法，则不会进入该内嵌的方法中
F8	在 Debug 模式下，进入下一步，如果当前行断点是一个方法，则不进入当前方法体内
Shift + F7	在 Debug 模式下，智能步入。断点所在行上有多个方法调用，会弹出进入哪个方法
Shift + F8	在 Debug 模式下，跳出，表现出来的效果跟 F9 一样
Alt + F8	在 Debug 模式下，选中对象，弹出可输入计算表达式调试框，查看该输入内容的调试结果
Alt + F9	在 Debug 模式下，执行到光标处
F9	在 Debug 模式下，恢复程序运行，但是如果该断点下面代码还有断点则停在下一个断点上
Ctrl + F8	在 Debug 模式下，设置光标当前行为断点，如果当前已经是断点则去掉断点
Ctrl + Shift + F8	在 Debug 模式下，指定断点进入条件

Navigation

快捷键	介绍
Ctrl + N	跳转到类 （必备）
Ctrl + Shift + N	跳转到文件 （必备）
Ctrl + Alt + Shift + N	跳转到符号 （必备）
Alt + Left/Right	切换当前已打开的窗口中的子视图，比如 Debug 窗口中有 Output、Debugger 等子视图，用此快捷键就可以在子视图中切换 （必备）
F12	回到前一个工具窗口 （必备）
ESC	从工具窗口进入代码文件窗口 （必备）
Shift + ESC	隐藏当前 或 最后一个激活的工具窗口
Ctrl + G	跳转到当前文件的指定行处
Ctrl + E	显示最近打开的文件记录列表 （必备）
Ctrl + Shift + E	显示最近编辑的文件记录列表 （必备）
Ctrl + Alt + Left/Right	跳转到上一个/下一个操作的地方 （必备）
Ctrl + Shift + Backspace	退回到上次修改的地方 （必备）
Alt + F1	显示当前文件选择目标弹出层，弹出层中有很多目标可以进行选择 （必备）
Ctrl + B/Ctrl + 左键单击	跳转到声明处
Ctrl + Alt + B	在某个调用的方法名上使用会跳到具体的实现处，可以跳过接口
Ctrl + Shift + B	跳转到类型声明处 （必备）
Ctrl + Shift + I	快速查看光标所在的方法 或 类的定义
Ctrl + U	前往当前光标所在的方法的父类的方法/接口定义 （必备）
Alt + Up/Down	跳转到当前文件的前一个/后一个方法 （必备）
Ctrl +]/[	跳转到当前所在代码的花括号结束位置/开始位置
Ctrl + F12	弹出当前文件结构层，可以在弹出的层上直接输入，进行筛选

快捷键	介绍
Ctrl + H	显示当前类的层次结构
Ctrl + Shift + H	显示方法层次结构
Ctrl + Alt + H	调用层次
F2/Shift + F2	跳转到下一个/上一个高亮错误 或 警告位置 （必备）
F4	编辑源 （必备）
Alt + Home	定位/显示到当前文件的 Navigation Bar
F11	添加书签 （必备）
Ctrl + F11	选中文件/文件夹，使用助记符设定/取消书签 （必备）
Shift + F11	弹出书签显示层 （必备）
Alt + 1,2,3...9	显示对应数值的选项卡，其中 1 是 Project 用得最多 （必备）
Ctrl + 1,2,3...9	定位到对应数值的书签位置 （必备）

Refactoring

快捷键	介绍
Shift + F6	对文件/文件夹 重命名 （必备）
Ctrl + Alt + Shift + T	打开重构菜单 （必备）

VCS/Local History

快捷键	介绍	
Ctrl + K	版本控制提交项目，需要此项目有加入到版本控制才可用	
Ctrl + T	版本控制更新项目，需要此项目有加入到版本控制才可用	
`Alt +	`	显示版本控制常用操作菜单弹出层 （必备）
Alt + Shift + C	查看最近操作项目的变化情况列表	
Alt + Shift + N	选择/添加 task （必备）	

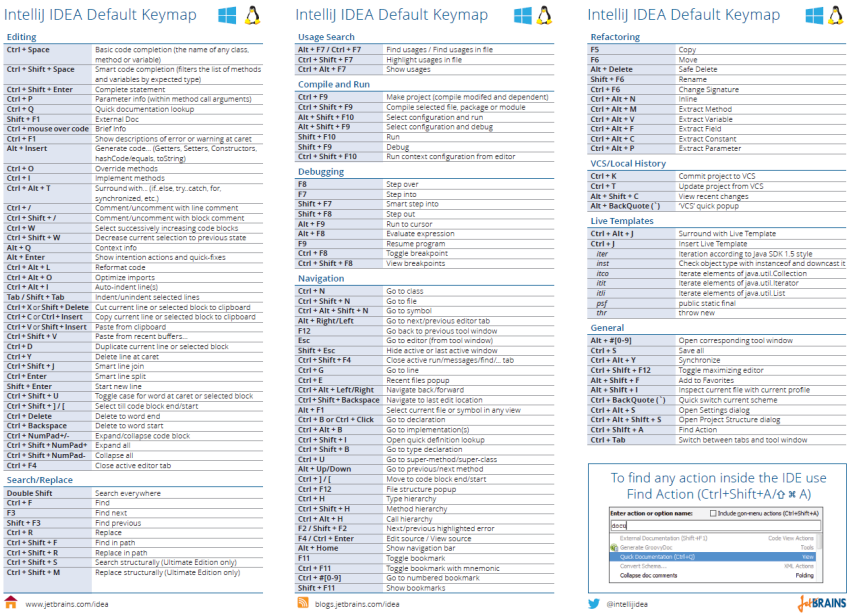
Live Templates

快捷键	介绍
Ctrl + J	插入自定义动态代码模板 （必备）
Ctrl + Alt + J	弹出模板选择窗口，将选定的代码加入动态模板中

General

快捷键	介绍
Ctrl + Tab	编辑窗口切换，如果在切换的过程又加按上 delete，则是关闭对应选中的窗口
Ctrl + Alt + Y	同步、刷新
Ctrl + Alt + S	打开 IntelliJ IDEA 系统设置 （必备）
Ctrl + Alt + Shift + S	打开当前项目设置 （必备）
Ctrl + Shift + A	查找动作/设置 （必备）
Ctrl + Shift + F12	编辑器最大化 （必备）
Alt + Shift + F	显示添加到收藏夹弹出层/添加到收藏夹
Alt + Shift + I	查看项目当前文件

Intellij IDEA 官方快捷键表



IntelliJ IDEA Default Keymap

Editing	
Ctrl + Space	Basic code completion (the name of any class, method or variable)
Ctrl + Shift + Space	Smart code completion (filters the list of methods and variables by expected type)
Ctrl + Shift + Enter	Complete statement
Ctrl + P	Parameter info (within method call arguments)
Ctrl + Q	Quick documentation lookup
Shift + F1	External Doc
Ctrl + mouse over code	Brief info
Ctrl + F1	Show descriptions of error or warning at caret
Alt + Insert	Generate code... (Getters, Setters, Constructors, hashCode/equals, toString)
Ctrl + O	Override methods
Ctrl + I	Implement methods
Ctrl + Alt + T	Surround with... (if..else, try..catch, for, synchronized, etc.)
Ctrl + /	Comment/uncomment with line comment
Ctrl + Shift + /	Comment/uncomment with block comment
Ctrl + W	Select successively increasing code blocks
Ctrl + Shift + W	Decrease current selection to previous state
Alt + Q	Context info
Alt + Enter	Show intention actions and quick-fixes
Ctrl + Alt + L	Reformat code
Ctrl + Alt + O	Optimize imports
Ctrl + Alt + I	Auto-indent line(s)
Tab / Shift + Tab	Indent/unindent selected lines
Ctrl + X or Shift + Delete	Cut current line or selected block to clipboard
Ctrl + C or Ctrl+ Insert	Copy current line or selected block to clipboard
Ctrl + V or Shift + Insert	Paste from clipboard
Ctrl + Shift + V	Paste from recent buffers...
Ctrl + D	Duplicate current line or selected block
Ctrl + Y	Delete line at caret
Ctrl + Shift + J	Smart line join
Ctrl + Enter	Smart line split
Shift + Enter	Start new line
Ctrl + Shift + U	Toggle case for word at caret or selected block
Ctrl + Shift +] / [	Select till code block end/start
Ctrl + Delete	Delete to word end
Ctrl + Backspace	Delete to word start
Ctrl + NumPad+/-	Expand/collapse code block
Ctrl + Shift + NumPad+	Expand all
Ctrl + Shift + NumPad-	Collapse all
Ctrl + F4	Close active editor tab

Search/Replace	
Double Shift	Search everywhere
Ctrl + F	Find
F3	Find next
Shift + F3	Find previous
Ctrl + R	Replace
Ctrl + Shift + F	Find in path
Ctrl + Shift + R	Replace in path
Ctrl + Shift + S	Search structurally (Ultimate Edition only)
Ctrl + Shift + M	Replace structurally (Ultimate Edition only)

www.jetbrains.com/idea

IntelliJ IDEA Default Keymap

Usage Search	
Alt + F7 / Ctrl + F7	Find usages / Find usages in file
Ctrl + Shift + F7	Highlight usages in file
Ctrl + Alt + F7	Show usages

Compile and Run	
Ctrl + F9	Make project (compile modified and dependent)
Ctrl + Shift + F9	Compile selected file, package or module
Alt + Shift + F10	Select configuration and run
Alt + Shift + F9	Select configuration and debug
Shift + F10	Run
Shift + F9	Debug
Ctrl + Shift + F10	Run context configuration from editor

Debugging	
F8	Step over
F7	Step into
Shift + F7	Smart step into
Shift + F8	Step out
Alt + F9	Run to cursor
Alt + F8	Evaluate expression
F9	Resume program
Ctrl + F8	Toggle breakpoint
Ctrl + Shift + F8	View breakpoints

Navigation	
Ctrl + N	Go to class
Ctrl + Shift + N	Go to file
Ctrl + Alt + Shift + N	Go to symbol
Alt + Right/Left	Go to next/previous editor tab
F12	Go back to previous tool window
Esc	Go to editor (from tool window)
Shift + Esc	Hide active or last active window
Ctrl + Shift + F4	Close active run/messages/find/... tab
Ctrl + G	Go to line
Ctrl + E	Recent files popup
Ctrl + Alt + Left/Right	Navigate back/forward
Ctrl + Shift + Backspace	Navigate to last edit location
Alt + F1	Select current file or symbol in any view
Ctrl + B or Ctrl + Click	Go to declaration
Ctrl + Alt + B	Go to implementation(s)
Ctrl + Shift + I	Open quick definition lookup
Ctrl + Shift + B	Go to type declaration
Ctrl + U	Go to super-method/super-class
Alt + Up/Down	Go to previous/next method
Ctrl +] / [	Move to code block end/start
Ctrl + F12	File structure popup
Ctrl + H	Type hierarchy
Ctrl + Shift + H	Method hierarchy
Ctrl + Alt + H	Call hierarchy
F2 / Shift + F2	Next/previous highlighted error
F4 / Ctrl + Enter	Edit source / View source
Alt + Home	Show navigation bar
F11	Toggle bookmark
Ctrl + F11	Toggle bookmark with mnemonic
Ctrl + #[0-9]	Go to numbered bookmark
Shift + F11	Show bookmarks

blogs.jetbrains.com/idea

IntelliJ IDEA Default Keymap

Refactoring	
F5	Copy
F6	Move
Alt + Delete	Safe Delete
Shift + F6	Rename
Ctrl + F6	Change Signature
Ctrl + Alt + N	Inline
Ctrl + Alt + M	Extract Method
Ctrl + Alt + V	Extract Variable
Ctrl + Alt + F	Extract Field
Ctrl + Alt + C	Extract Constant
Ctrl + Alt + P	Extract Parameter

VCS/Local History	
Ctrl + K	Commit project to VCS
Ctrl + T	Update project from VCS
Alt + Shift + C	View recent changes
Alt + BackQuote (`)	'VCS' quick popup

Live Templates	
Ctrl + Alt + J	Surround with Live Template
Ctrl + J	Insert Live Template
iter	Iteration according to Java SDK 1.5 style
inst	Check object type with instanceof and downcast it
itco	Iterate elements of java.util.Collection
itit	Iterate elements of java.util.Iterator
itli	Iterate elements of java.util.List
psf	public static final
thr	throw new

General	
Alt + #[0-9]	Open corresponding tool window
Ctrl + S	Save all
Ctrl + Alt + Y	Synchronize
Ctrl + Shift + F12	Toggle maximizing editor
Alt + Shift + F	Add to Favorites
Alt + Shift + I	Inspect current file with current profile
Ctrl + BackQuote (`)	Quick switch current scheme
Ctrl + Alt + S	Open Settings dialog
Ctrl + Alt + Shift + S	Open Project Structure dialog
Ctrl + Shift + A	Find Action
Ctrl + Tab	Switch between tabs and tool window

To find any action inside the IDE use Find Action (Ctrl+Shift+A/⇧⌘A)

@intellijidea

JetBRAINS

插件

推荐几个比较好用的插件

- Key promoter 快捷键提示
- CamelCase 驼峰式命名和下划线命名交替变化
- CheckStyle-IDEA 代码规范检查
- FindBugs-IDEA潜在 Bug 检查
- MetricsReloaded 代码复杂度检查
- Statistic 代码统计
- JRebel Plugin 热部署
- GsonFormat 把 JSON 字符串直接实例化成类
- Eclipse Code Formatter 如果你以前用的是 IDE，并有自己的一套代码风格配置，可以通过此插件导入到 IDEA
- Alibaba Java Coding Guidelines 阿里 Java 开发规范的静态检查工具
- IDE Features Trainer 官方的新手训练插件
- Markdown Navigator Markdown 插件，适用于喜欢用 markdown 写文档的人

个性化

颜色主题

intellij-colors-solarized 个人觉得这种色彩搭配十分优雅

下载地址

破解

Intellij 是一个收费的 IDE，坦白说有点小贵，买不起。

所以，很惭愧，只好用下破解方法了。网上有很多使用注册码的网文，但是注册码不稳定，随时可能被封。还是自行搭建一个注册服务器比较稳定。我使用了 ilanyu 博文 IntelliJ IDEA License Server 本地搭建教程 的方法，亲测十分有效。

我的备用地址：百度云盘

下载并解压文中的压缩包到本地，选择适合操作系统的版本运行。

如果是在 Linux 上运行，推荐创建一个脚本，代码如下：

```
# 使用 nohup 创建守护进程，运行 IntelliJIDEALicenseServer_linux_am
# 如果运行在其他 Linux 发行版本，替换执行的脚本即可
nohup sh IntelliJIDEALicenseServer_linux_amd64 2>&1
```

这样做是因为：大部分人使用 linux 是使用仿真器连接虚拟机，如果断开连接，进程也会被 kill，每次启动这个注册服务器很麻烦不是吗？而启动了守护进程，则不会出现这种情况，只有你主动 kill 进程才能将其干掉。

Windows 版本是 exe 程序，将其设为开机自动启动即可，别告诉我你不知道怎么设置开机自动启动。

参考资料

- IntelliJ-IDEA-Tutorial
- 极客学院 - Intellij IDEA 使用教程

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-11-29 13:32:51

- Eclipse 使用指南
  - 代码智能提示
    - Java 智能提示
    - JavaScript 智能提示
    - HTML 智能提示
  - 插件安装
  - 基本设置
  - 设置文本文件及 JSP 文件编码
  - 设置 JDK 本地 JavaDOC API 路径及源码路径
  - 设置 Servlet 源码或其它 Jar 包源码
  - 反编译插件 JD-Eclipse
  - Validate 优化
  - 常用快捷键

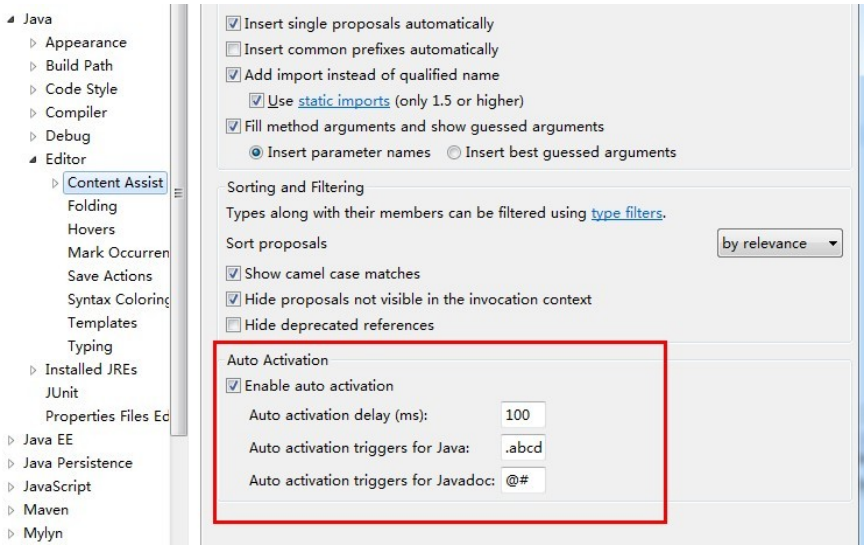
Eclipse 使用指南

- 代码智能提示
- 插件安装
- 基本设置
- 设置文本文件及 JSP 文件编码
- 设置 JDK 本地 JavaDOC API 路径及源码路径
- 设置 Servlet 源码或其它 Jar 包源码
- 反编译插件 JD-Eclipse
- Validate 优化
- 常用快捷键

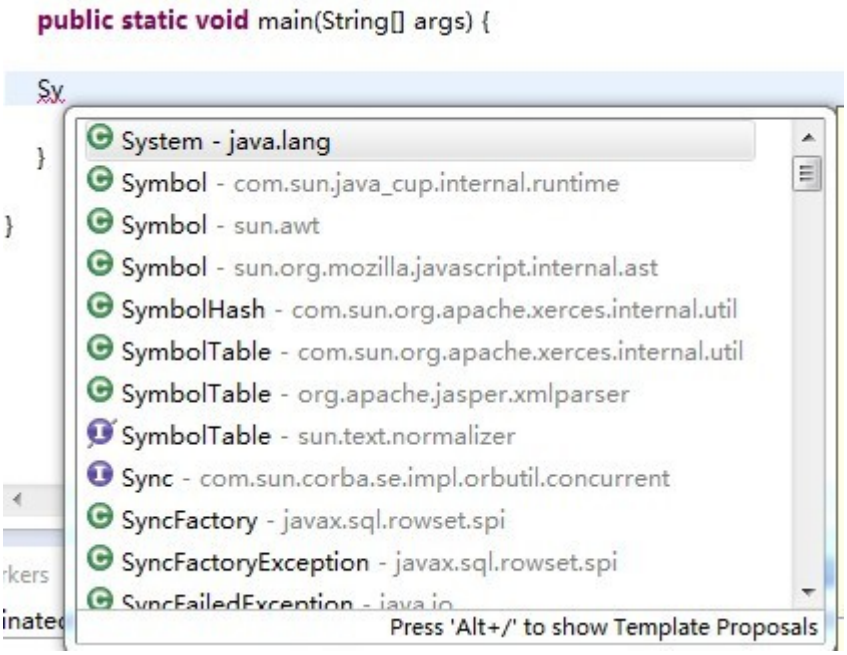
代码智能提示

Java 智能提示

Window -> Preferences -> Java -> Editor -> Content Assist -> Auto Activation



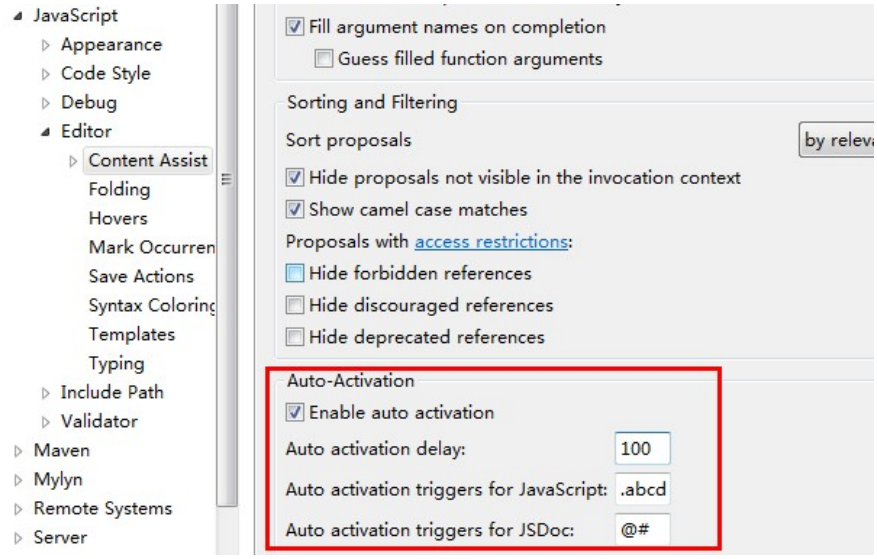
delay 是自动弹出提示框的延时时间，我们可以修改成 100 毫秒；triggers 这里默认是"."，只要加上"abcdefghijklmnopqrstuvwxyz"或者"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"，嘿嘿！这下就能做到和 VS 一样的输入每个字母都能提示啦：



其它类型的文件比如 HTML、JavaScript、JSP 如果也能提供提示那不是更爽了？有了第二点设置的基础，其实这些设置都是一样的。

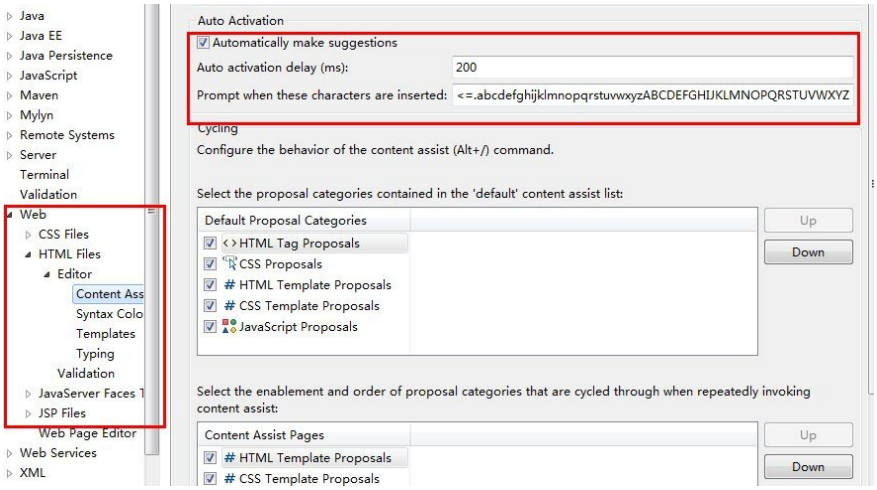
JavaScript 智能提示

Window -> Preferences -> JavaScript-> Editor -> Content Assist -> Auto-Activation

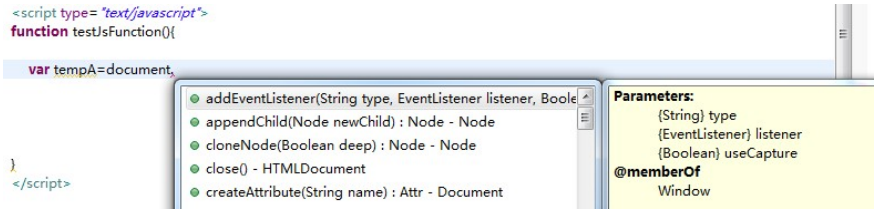


HTML 智能提示

Window -> Preferences -> Web -> HTML Files -> Editor -> Content Assist -> Auto-Activation



保存后，我们再来输入看看，感觉真是不错呀：

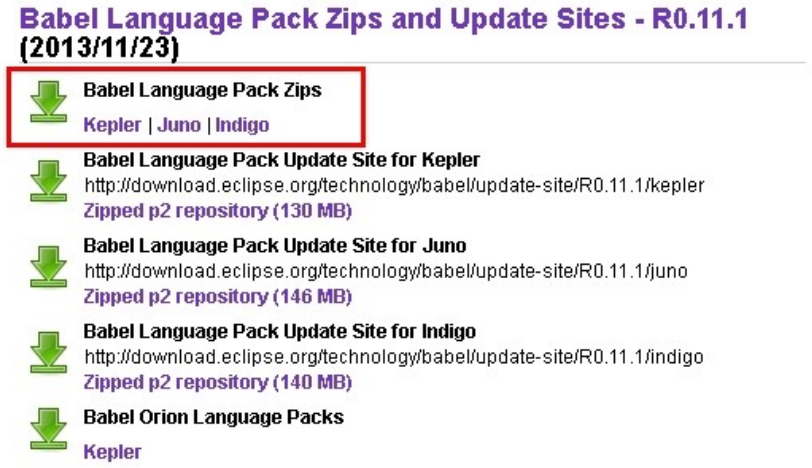


插件安装

很多教科书上说到 Eclipse 的插件安装都是通过 Help -> Install New SoftWare 这种自动检索的方式，操作起来固然是方便，不过当我们不需要某种插件时不太容易找到要删除哪些内容，而且以后 Eclipse 版本升级

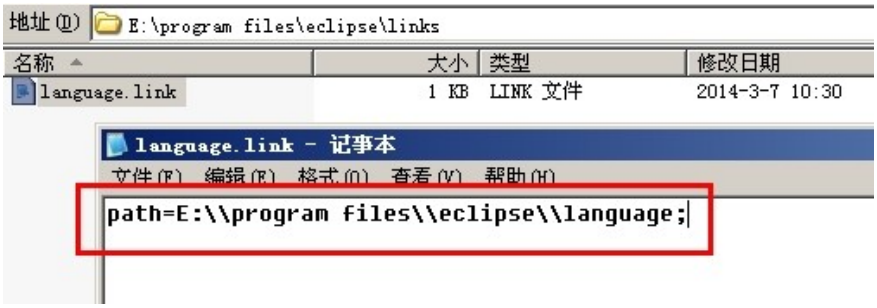
的时候，通过这种方式安装过的插件都得再重新装一次。另外一种通过 Link 链接方式，就可以解决这些问题。

我们以 Eclipse 的中文汉化包插件为例，先到官方提供的汉化包地址下载一个：http://www.eclipse.org/babel/downloads.php，注意选好自己的 Eclipse 版本：



我的版本是 Kepler，然后进入下载页面，单击红框框中的链接，即可下载汉化包了：

下载完解压缩后，会有个包含 features 和 plugin 目录的 eclipse 文件夹，把这个 eclipse 放在我们的 Eclipse 安装根目录，也就是和 eclipse.exe 同一级目录下。然后仍然在这一级目录下，新建一个 links 文件夹，并在该文件夹内，建一个 language.link 的文本文件。该文本文件的名字是可以任取的，后缀名是.link，而不是.txt 哟。好了，最后一步，编辑该文件，在里面写入刚才放入的语言包的地址，并用“\”表示路径，一定要有 path= 这个前缀。

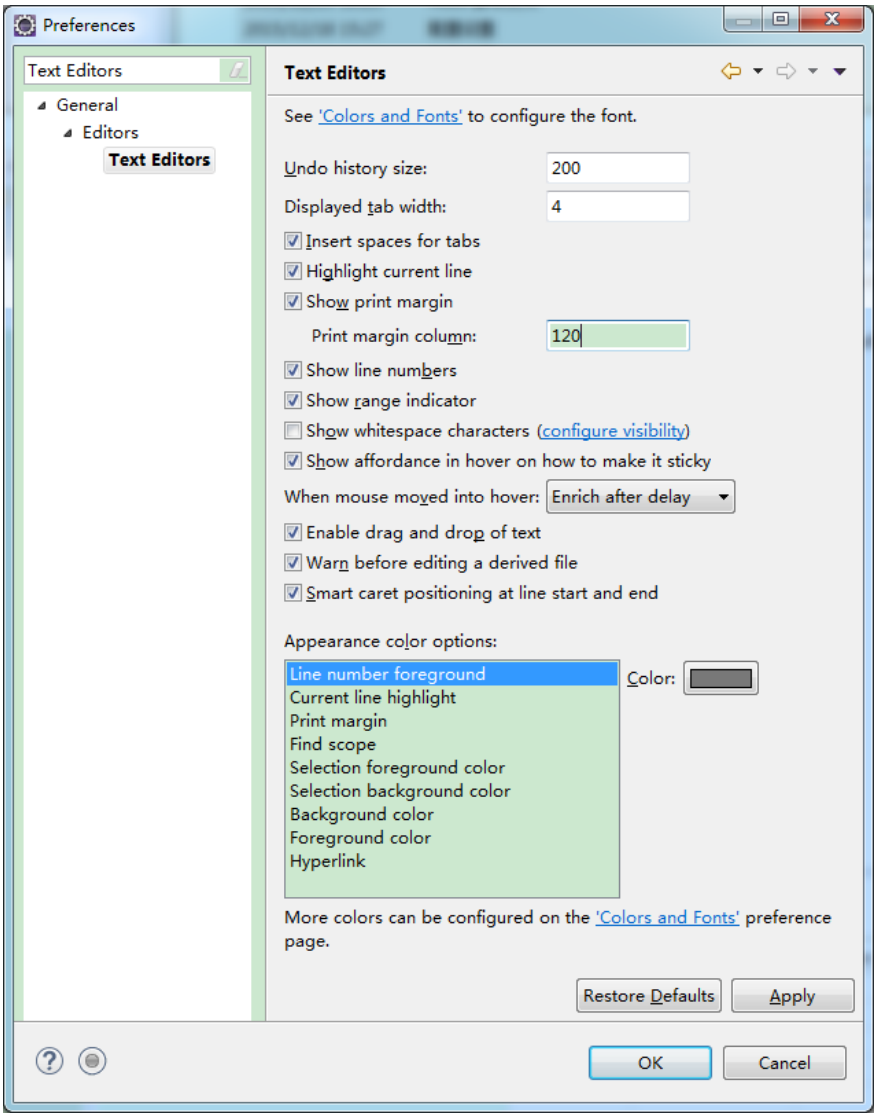


保存文件后，重新打开 Eclipse，熟悉的中文界面终于看到了。虽然汉化不完全，不过也够用了不是么。如果仍然出现的是英文，说明汉化失败，重新检查下 language.link 文件中配置的信息是否和汉化包的目录一

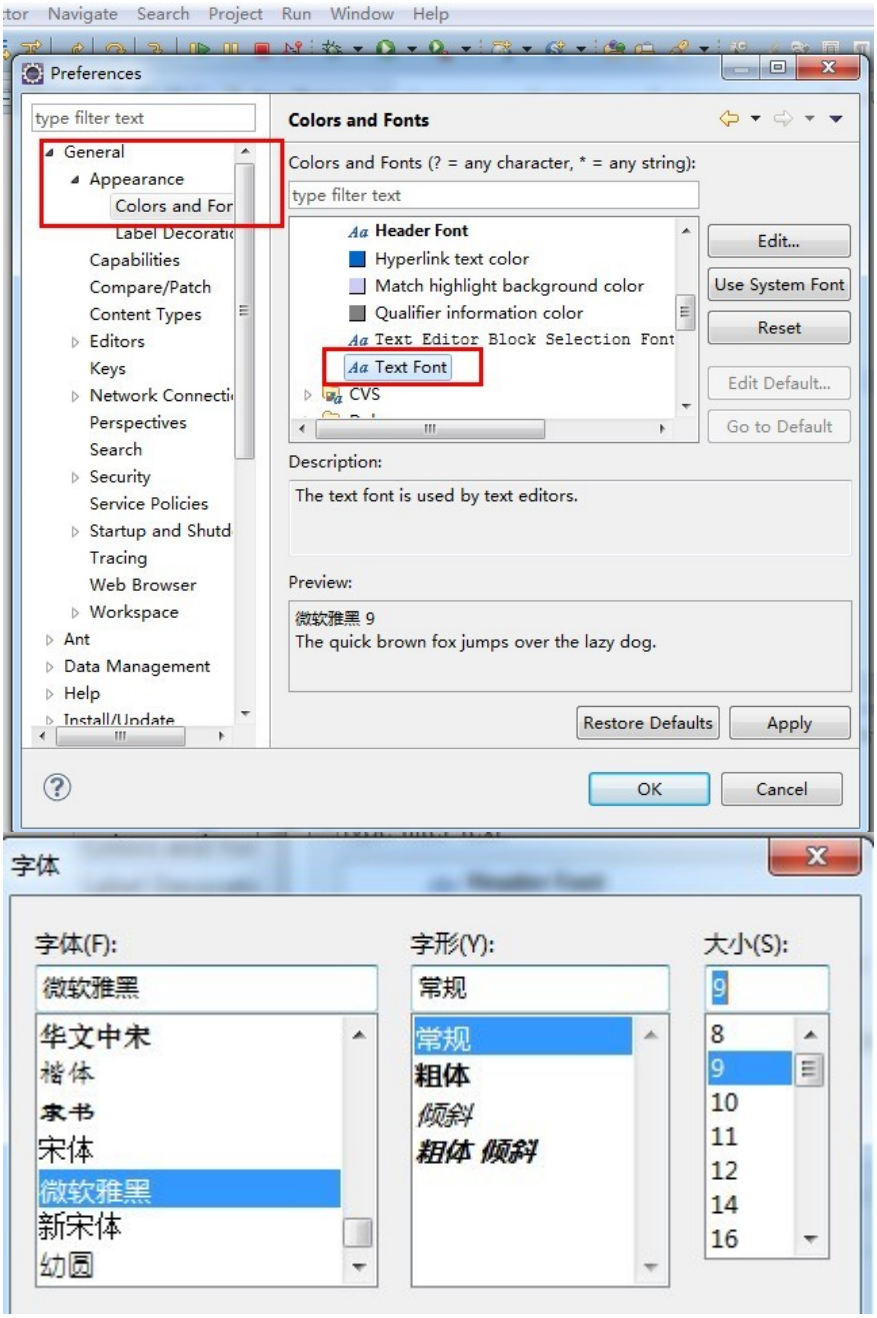
致。　其它的插件安装方法也是如此，当不需要某个插件时，只需删除存放插件的目录和 links 目录下相应的 link 文件，或者改变下 link 文件里面的路径变成无效路径即可；对 Eclipse 做高版本升级时，也只需把老版存放插件的目录和 links 目录复制过去就行了。

基本设置

在 Preference 的搜索项中搜索 Text Editors。 可以参考我的设置： Show line numbers Show print margin Insert spaces for tabs

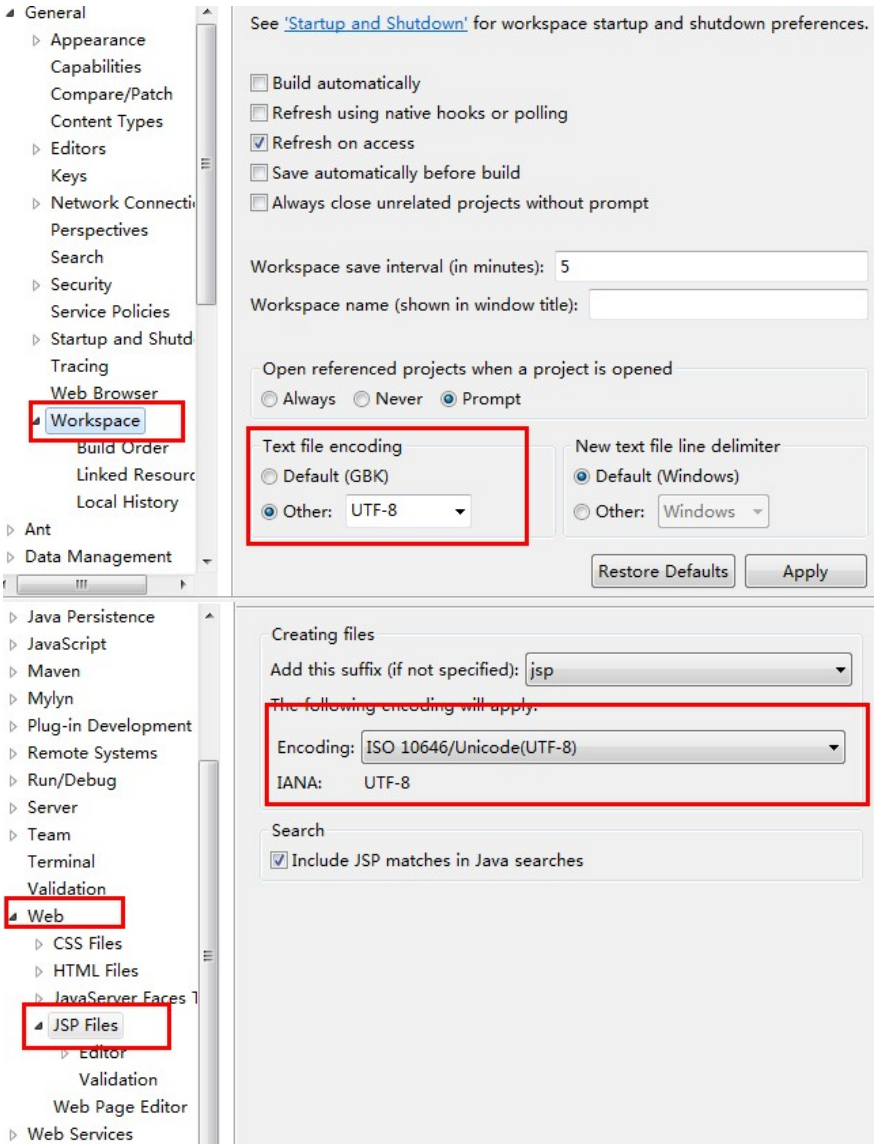


设置代码的字体类型和大小： Window -> Preferences -> General -> Appearance -> Content Assist -> Colors and Fornts，只需修改 Basic 里面的 Text Font 就可以了。 推荐 Courier New。

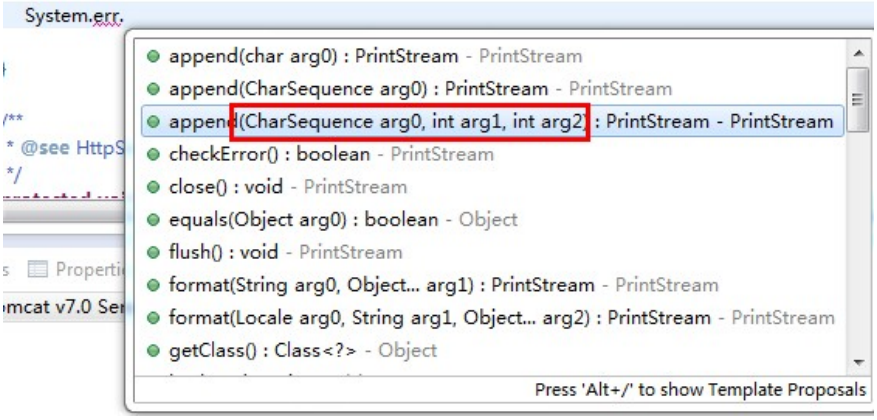


设置文本文件及 JSP 文件编码

Window -> Preferences -> General -> Workspace -> Text file encoding -> Other：



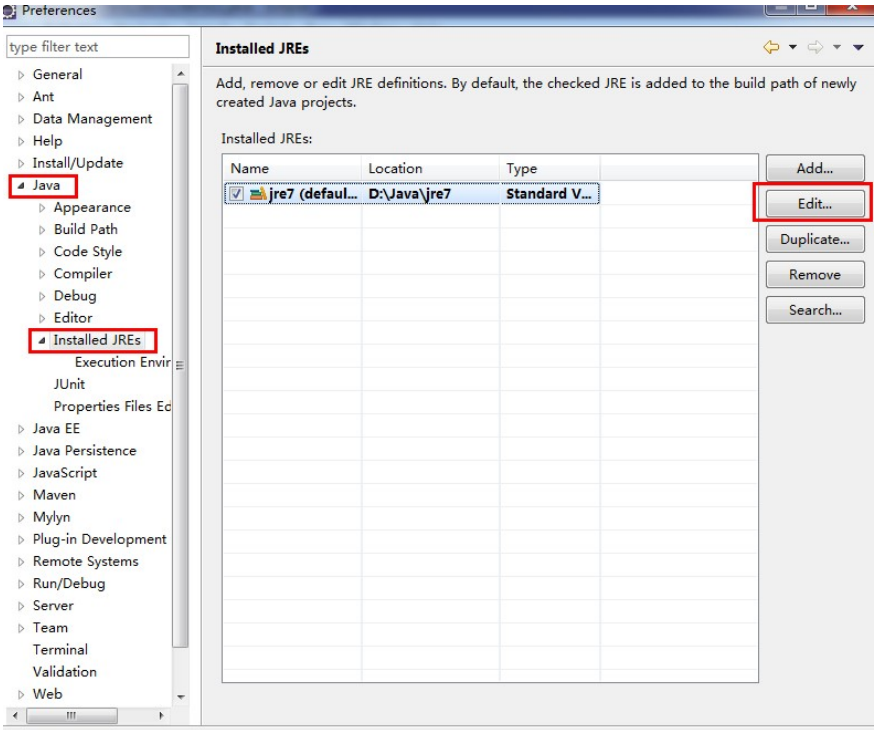
设置 JDK 本地 JavaDOC API 路径及源码路径



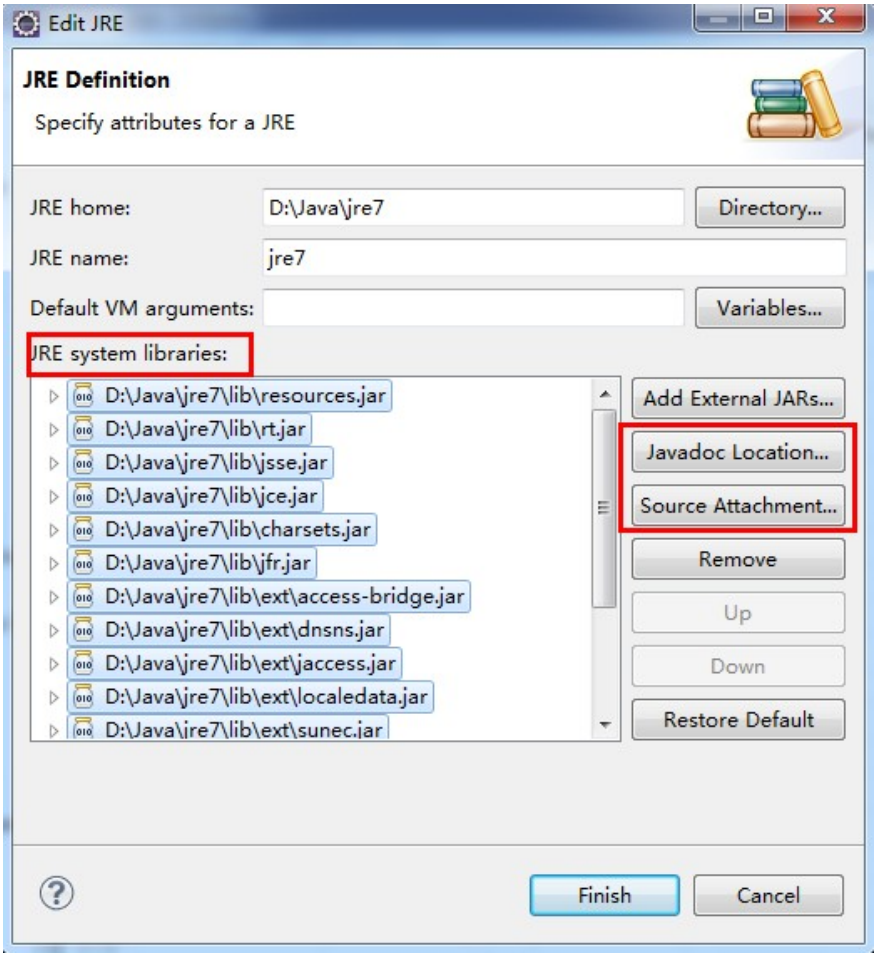
还都生成的是无意义的变量名，这样可能会对含有相同类型的变量参数的调用顺序造成干扰；

这种问题，我们把 JDK 或者相应 Jar 包的源码导入进去就能避免了：

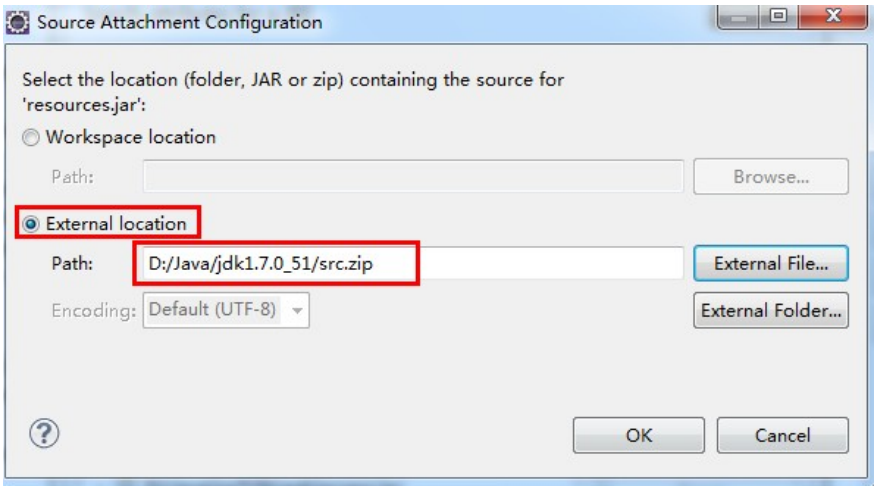
Window -> Preferences -> Java -> Installed JREs -> Edit：



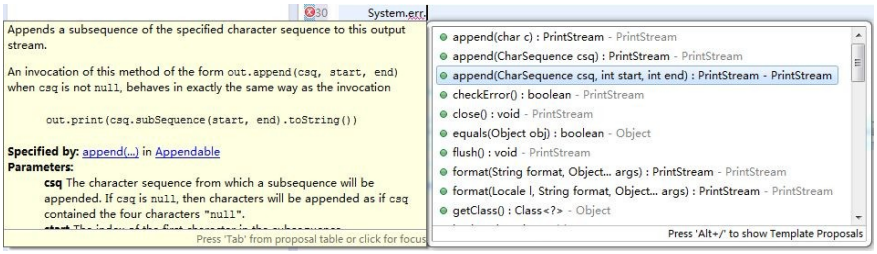
选中设置好的 JRE 目录，编辑，然后全选 JRE system libraries 下的所有 Jar 包，点击右边的 Source Attachment；



External location 下，选中 JDK 安装目录下的 src.zip 文件，一路 OK 下来。



设置完，我们再来看看，幸福来的好突然有木有！



设置 Servlet 源码或其它 Jar 包源码

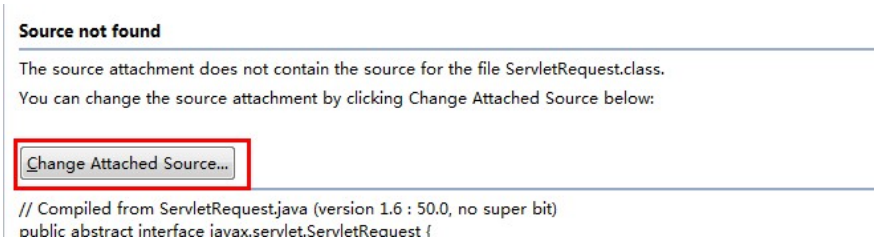
上一步已经设置过了 JDK 的源码或 JavaDoc 路径，为啥现在又出来了呢？其实这个不难理解，因为我们使用到的类的源码并不在 JDK 的源码包中。

仔细看，我们会发现这些 Jar 包其实都在 Tomcat 根目录下的 lib 文件夹中，但是翻遍了 Tomcat 目录也没有相应的 jar 或 zip 文件呀。既然本地没有，那就去官网上找找：

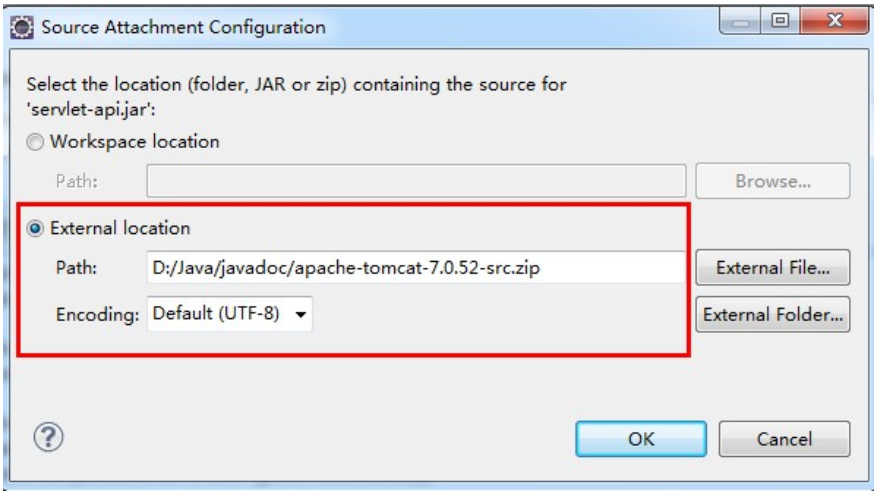
http://tomcat.apache.org/download-70.cgi这里有Tomcat的安装包和源码包；

可以自定义一个专门用于存放 JavaSource 和 JavaDoc 的文件夹，把下载文件放到该目录下，

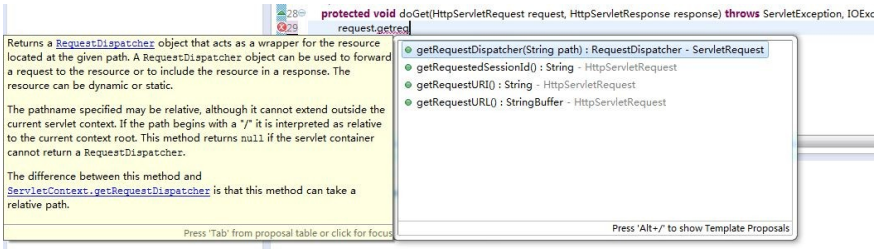
然后再切换到 Eclipse 下，选中没有代码提示的类或者函数， 按下 F3，点击 Change Attached Source：



选择我们刚才下载好的 tomcat 源码文件，一路 OK。



然后再回过头看看我们的代码提示，友好多了：

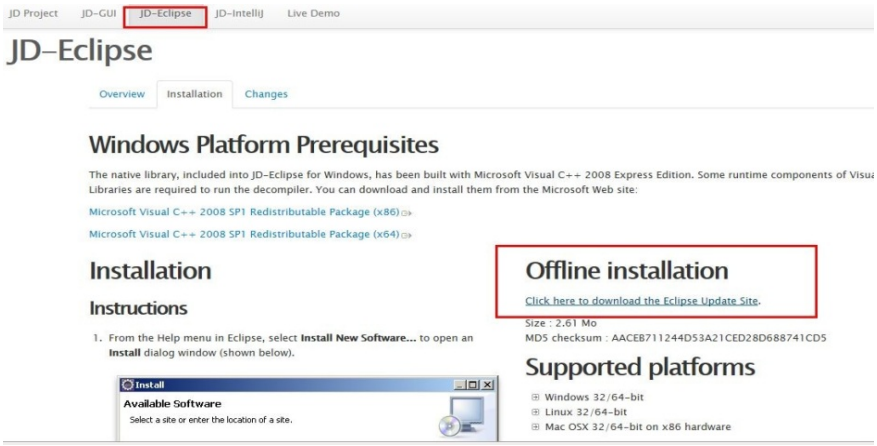


其它 Jar 包源码的设置方式也一样。

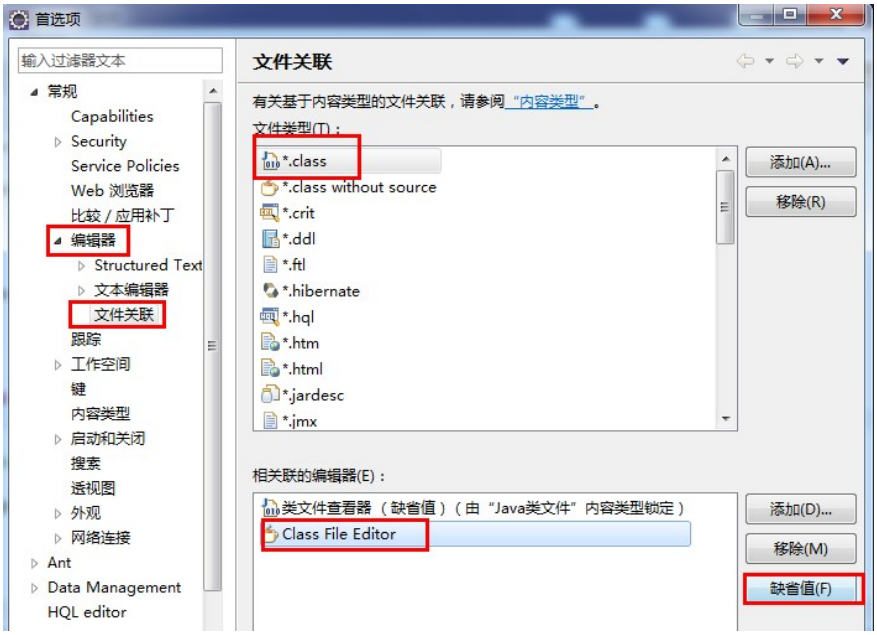
反编译插件 JD-Eclipse

无论是开发还是调试，反编译必不可少，每次都用 jd-gui 打开去看，多麻烦，干脆配置下 JD 插件，自动关联.class：

先从 http://jd.benow.ca/ 上下载离线安装包 jdeclipse_update_site.zip，解压缩后把 features、plugins 这 2 个文件夹复制到 新建文件夹 jdeclipse，然后把 jdeclipse 文件夹整个复制到 Eclipse 根目录的 dropins 文件夹下，重启 Eclipse 即可。这种方式是不是比建 link 文件更方便了？

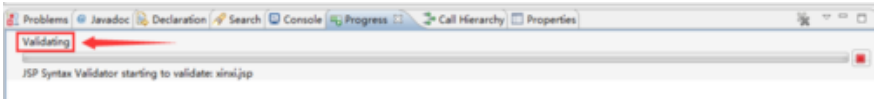


打开 Eclipse，Window -> Preferences -> General - > Editors ，把 .class 文件设置关联成 jd 插件的 editor

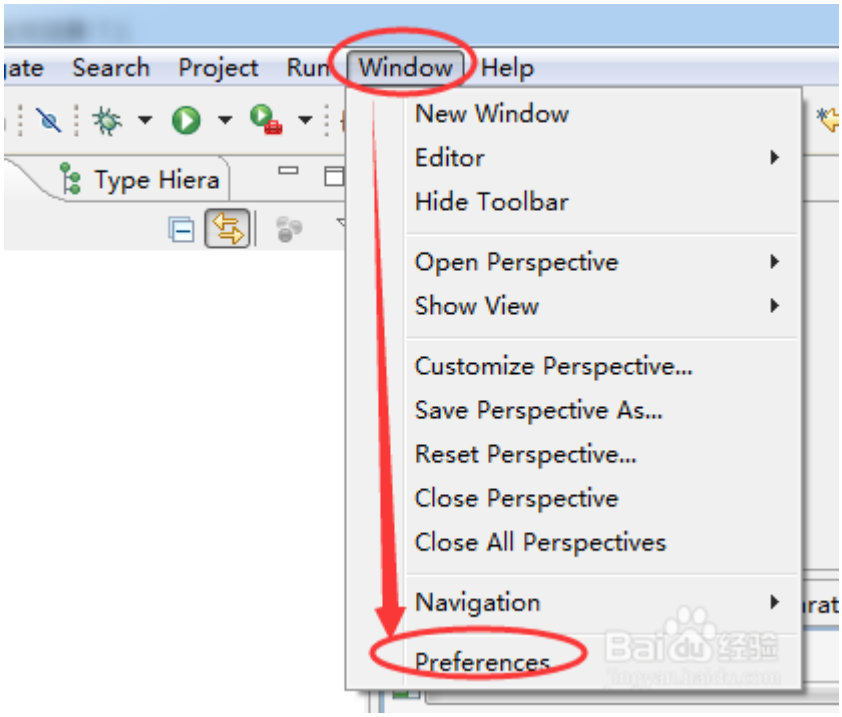


Validate 优化

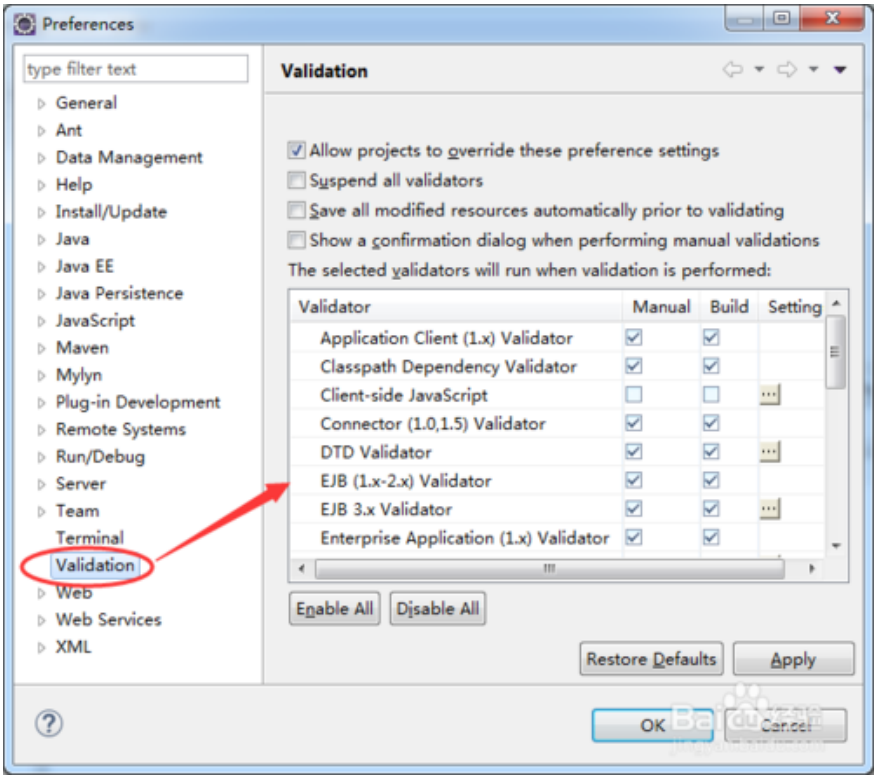
我们在 eclipse 里经常看到这个进程，validating... 逐个的检查每一个文件。那么如何关闭一些 validate 操作呢？



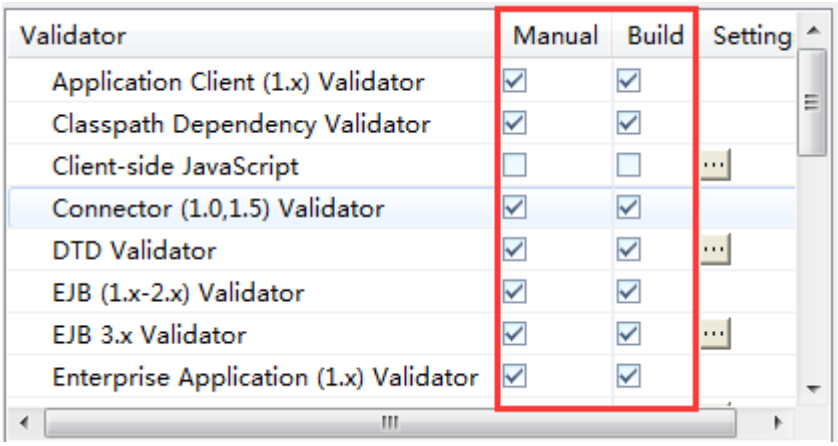
打开 eclipse，点击【window】菜单，选择【preferences】选项。



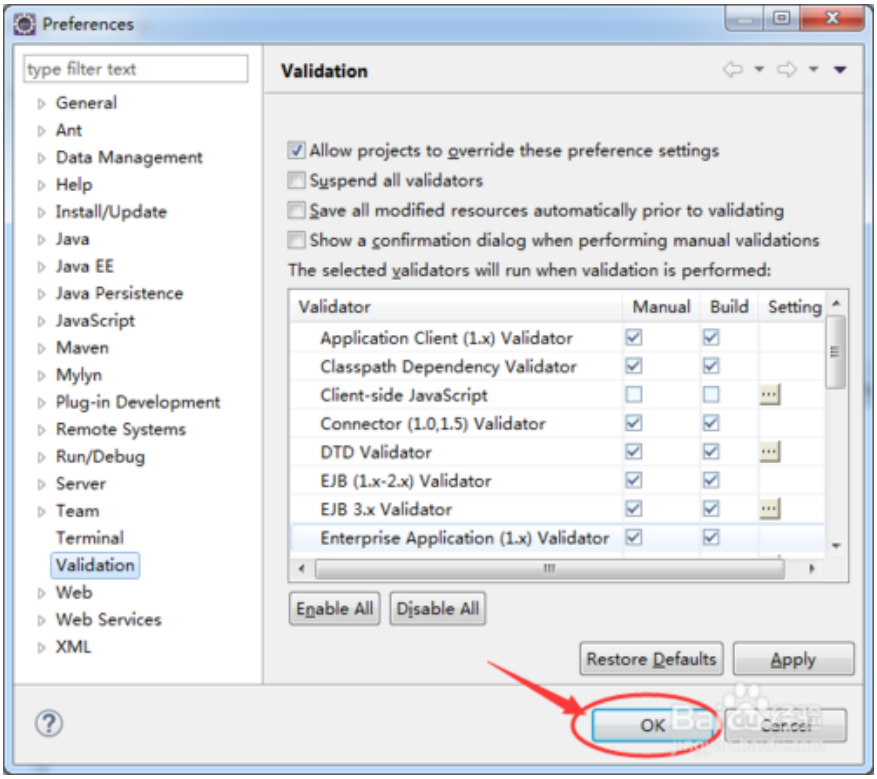
在左侧点击【validation】选项，在右侧可以看到 eclipse 进行的自动检查都有哪些内容。



将 Manual（手动）保持不动，将 build 里面只留下 classpath dependency Validator，其他的全部去掉。



最后点击【OK】按钮，保存设置。



以后如果需要对文件进行校验检查的时候，在文件上点击右键，点击【Validate】进行检查。

常用快捷键

快捷键	描述
Ctrl+1	快速修复（最经典的快捷键,就不用多说了，可以解决很多问题，比如 import 类、try catch 包围等）
Ctrl+Shift+F	格式化当前代码
Ctrl+Shift+M	添加类的 import 导入
Ctrl+Shift+O	组织类的 import 导入（既有 Ctrl+Shift+M 的作用，又可以帮你去除没用的导入，很有用）
Ctrl+Y	重做（与撤销 Ctrl+Z 相反）
Alt+/	内容辅助（帮你省了多少次键盘敲打，太常用了）
Ctrl+D	删除当前行或者多行
Alt+↓	当前行和下面一行交互位置（特别实用,可以省去先剪切,再粘贴了）
Alt+↑	当前行和上面一行交互位置（同上）
Ctrl+Alt+↓	复制当前行到下一行（复制增加）
Ctrl+Alt+↑	复制当前行到上一行（复制增加）
Shift+Enter	在当前行的下一行插入空行（这时鼠标可以在当前行的任一位置,不一定是最后）
Ctrl+/	注释当前行,再按则取消注释
Alt+Shift+↑	选择封装元素
Alt+Shift+←	选择上一个元素
Alt+Shift+→	选择下一个元素
Shift+←	从光标处开始往左选择字符
Shift+→	从光标处开始往右选择字符
Ctrl+Shift+←	选中光标左边的单词
Ctrl+Shift+→	选中光标又边的单词
Ctrl+←	光标移到左边单词的开头，相当于 vim 的 b
Ctrl+→	光标移到右边单词的末尾，相当于 vim 的 e
Ctrl+K	参照选中的 Word 快速定位到下一个（如果没有选中 word，则搜索上一次使用搜索的 word）

快捷键	描述
Ctrl+Shift+K	参照选中的 Word 快速定位到上一个
Ctrl+J	正向增量查找（按下 Ctrl+J 后,你所输入的每个字母编辑器都提供快速匹配定位到某个单词,如果没有,则在状态栏中显示没有找到了,查一个单词时,特别实用,要退出这个模式，按 escape 建）
Ctrl+Shift+J	反向增量查找（和上条相同,只不过是从后往前查）
Ctrl+Shift+U	列出所有包含字符串的行
Ctrl+H	打开搜索对话框
Ctrl+G	工作区中的声明
Ctrl+Shift+G	工作区中的引用
Ctrl+Shift+T	搜索类（包括工程和关联的第三 jar 包）
Ctrl+Shift+R	搜索工程中的文件
Ctrl+E	快速显示当前 Editer 的下拉列表（如果当前页面没有显示的用黑体表示）
F4	打开类型层次结构
F3	跳转到声明处
Alt+←	前一个编辑的页面
Alt+→	下一个编辑的页面（当然是针对上面那条来说了）
Ctrl+PageUp/PageDown	在编辑器中，切换已经打开的文件
F5	单步跳入
F6	单步跳过
F7	单步返回
F8	继续
Ctrl+Shift+D	显示变量的值
Ctrl+Shift+B	在当前行设置或者去掉断点
Ctrl+R	运行至行(超好用，可以节省好多的断点)
Alt+Shift+R	重命名方法名、属性或者变量名 （是我自己最爱用的一个了,尤其是变量和类的 Rename,比手工方法能节省很多劳动力）

快捷键	描述
Alt+Shift+M	把一段函数内的代码抽取成方法 （这是重构里面最常用的方法之一了,尤其是对一大堆泥团代码有用）
Alt+Shift+C	修改函数结构（比较实用,有 N 个函数调用了这个方法,修改一次搞定）
Alt+Shift+L	抽取本地变量（ 可以直接把一些魔法数字和字符串抽取成一个变量,尤其是多处调用的时候）
Alt+Shift+F	把 Class 中的 local 变量变为 field 变量（比较实用的功能）
Alt+Shift+I	合并变量（可能这样说有点不妥 Inline）
Alt+Shift+V	移动函数和变量（不怎么常用）
Alt+Shift+Z	重构的后悔药（Undo）
Alt+Enter	显示当前选择资源的属性，windows 下的查看文件的属性就是这个快捷键，通常用来查看文件在 windows 中的实际路径
Ctrl+↑	文本编辑器 上滚行
Ctrl+↓	文本编辑器 下滚行
Ctrl+M	最大化当前的 Edit 或 View （再按则反之）
Ctrl+O	快速显示 OutLine
Ctrl+T	快速显示当前类的继承结构
Ctrl+W	关闭当前 Editer
Ctrl+L	文本编辑器 转至行
F2	显示工具提示描述

Copyright © Zhang Peng 2017 all right reserved，powered by Gitbook该文件上次修订时间： 2019-08-20 17:21:39