

Java 的最佳实践

Java 是在世界各地最流行的编程语言之一，但是看起来没人喜欢使用它。而 Java 事实上还算是一门不错的语言，随着 Java 8 最近的问世，我决定编制一个库，实践和工具的清单，汇集 Java 的一些最佳实践。

本文被放到了 [Github](#) 上。你可以随意地提交贡献，并加入自己的有关 Java 方面的建议和最佳实践。

- 风格
 - Javadoc
 - 构建器模式
 - 结构
 - 依赖注入
 - 避免空值
 - 默认不可变更
 - 避免大量的工具类
 - 格式化
 - 流
- 发布
 - 依赖收敛
 - 框架
 - Maven
 - 持续集成
 - Maven 资源库
 - 配置管理
- 库
 - junit 4
 - jMock

- AssertJ
- Apache Commons
- Guava
- Gson
- Java Tuples
- Joda-Time
- Lombok
- Play framework
- SLF4J
- jOOQ
- Missing Features
- Testing
- 工具
 - Chronon
 - IntelliJ IDEA
 - JRebel
 - Checker 框架
 - Eclipse 内存分析器
- 资源
 - 书籍
 - 播客

风格

通常，我们会以一种非常详细繁杂的企业级 JavaBean 的风格进行 Java 代码的编写。新的风格则更加清晰，正确，且看上去也更加的简单。

结构

作为程序员的我们要做的最简单的事情之一，就是传递数据。一般的方式就是定义一个 `JavaBean`：

```
public class DataHolder {
    private String data;

    public DataHolder() {
    }

    public void setData(String data) {
        this.data = data;
    }

    public String getData() {
        return this.data;
    }
}
```

这有点麻烦，并且也有点浪费。尽管你的 IDE 也能自动的生成这样的代码，但那也是种浪费。所以，[别这么做](#)。

相反，我更愿意选择编写类 C 的结构体风格的类，类里面只容纳数据：

```
public class DataHolder {
    public final String data;

    public DataHolder(String data) {
        this.data = data;
    }
}
```

这样就在代码行数上减少了一半。此外，这个类是不能被修改的，除非你对它进行了扩展，因此我们可以更加容易的理解它，因为我们明白它不可以被修改。

如果你要保存像 `Map` 或者 `List` 这样容易被修改的对象，就应该使用 `ImmutableMap` 和 `ImmutableList`，这一点会在不可变性质的那一节被讲到。

Builder 模式

如果你有一个相当复杂的对象想要去为其构建一个结构，可以考虑使用 `Builder` 模式。你可以在对象中创建一个能帮助你构建出这个对象的子类。它使用了可变语句，但是一旦你调用了 `build`，它就会提供给你一个

不可变的对象。想象一下我们要有一个更加复杂的 *DataHolder*。针对它的构建器看起来可能像是下面这样：

```
public class ComplicatedDataHolder {
    public final String data;
    public final int num;
    // lots more fields and a constructor

    public static class Builder {
        private String data;
        private int num;

        public Builder data(String data) {
            this.data = data;
            return this;
        }

        public Builder num(int num) {
            this.num = num;
            return this;
        }

        public ComplicatedDataHolder build() {
            return new ComplicatedDataHolder(data, num); // etc
        }
    }
}
```

然后这样去使用它：

```
final ComplicatedDataHolder cdh = new ComplicatedDataHolder.Builder()
    .data("set this")
    .num(523)
    .build();
```

还有其它[关于构建器的更好的例子](#)，而这里提供给你浅尝辄止。这样做最终会得到许多的我们努力去避免的样板式代码，不过这也让你得到了不可变的对象和一个非常流畅的接口。

依赖注入

这是更偏向软件工程而不是 Java 的一节。编写可测试软件的最佳方式之一就是使用[依赖注入](#)(DI)。因为 Java 非常鼓励 OO 设计，为了创造出可测试的软件，你需要使用 DI。

在 Java 中，一般使用 [Spring 框架](#) 的 DI 实现。它同时支持基于代码的装配和基于 XML 配置的装配。

如果你使用的是 XML 配置，因为其基于 XML 的配置，[不去过分使用 Spring](#) 这一点很重要。XML 中绝对不能有任何逻辑或者控制结构，只能用来注入依赖。

使用 Spring 的好的选择就是 Google 和 Square 的 [Dagger](#) 库以及 Google 的 [Guice](#)。他们不使用 Spring 的 XML 配置文件格式，而是将依赖逻辑放到注解和代码中。

避免空值

尽你所能避免空值。如果你可以返回一个空的集合，就不要返回一个空值。如果你要使用空值，就考虑使用 [@Nullable](#) 注解。[IntelliJ IDEA](#) 内置有对于 [@Nullable](#) 注解的支持。

如果你使用的是 [Java 8](#)，就可以利用其优秀的新的 [Optional](#) 类型。如果一个可能存在也可能不存在，那就像下面这样把它封装到一个 [Optional](#) 类中：

```
public class FooWidget {
    private final String data;
    private final Optional<Bar> bar;

    public FooWidget(String data) {
        this(data, Optional.empty());
    }

    public FooWidget(String data, Optional<Bar> bar) {
        this.data = data;
        this.bar = bar;
    }

    public Optional<Bar> getBar() {
        return bar;
    }
}
```

这样现在就能很确定数据永远都不会是空值了，不过 `bar` 可能存在也可能不存在。[Optional](#) 有一些诸如 `isPresent` 这样的方法，这使得其感觉跟只检查空值的做法小同大异。但是它能让你写出像下面这样的语句：

```
final Optional<FooWidget> fooWidget = maybeGetFooWidget();
final Baz baz = fooWidget.flatMap(FooWidget::getBar)
    .flatMap(BarWidget::getBaz)
    .orElse(defaultBaz);
```

这样就比链条时的 `if` 空值检查看起来好多了。使用 [Optional](#) 的唯一缺陷就是标准库并没有对 [Optional](#) 有很好的支持，因此针对空值的处理还是需要的。

默认不可被改变

除非你有一个好的理由要这样做，那么变量、类和集合都是不应该被修改的。

变量的引用可以用 `final` 来变成不可被修改的：

```
final FooWidget fooWidget;if (condition()) {
    fooWidget = getWidget();} else {
    try {
        fooWidget = cachedFooWidget.get();
    } catch (CachingException e) {
        log.error("Couldn't get cached value", e);
        throw e;
    }
} // fooWidget is guaranteed to be set here
```

现在你就可以确信 `fooWidget` 不会突然被重新赋值了。`final` 关键字一般同 `if/else` 块和 `try/catch` 块一起使用。当然，如果 `fooWidget` 不是不可被修改的，那你就可以很轻易了修改它了。

集合就应该无论何时都尽量使用 [Guava](#) 的 [ImmutableMap](#)，[ImmutableList](#)，或者 [ImmutableSet](#) 类。这些都拥有构建器，因此你可以动态地构建它们，并通过调用 `build` 方法来将它们标记为不可变。

类应该（通过 `final`）声明其属性域不可变和使用不可变的集合而变成不可变的。你也可以选择使得类自身为 `final`，那样它就不能被扩展和被改变了。

避免许多的工具类

在你发现自己添加了太多的方法到一个工具类中时要小心。

```
public class MiscUtil {
    public static String frobnicateString(String base, int times) {
        // ... etc
    }

    public static void throwIfCondition(boolean condition, String msg) {
        // ... etc
    }
}
```

这些类一开始看起来很吸引人，因为它们里面包含的方法并不真的属于任何一块。所以你就以代码重用的名义将它们扔到了一块儿。

治病比生病更糟糕。将这些类放到原本属于它们的地方，要不如果你必须要有像这么一些方法的话，就考虑使用 [Java 8](#) 的接口上的默认方法。然后你就可以将公共方法统统扔到接口中去。而因为他们是接口，你就可以多次实现它们。

```
public interface Thrower {
    default void throwIfCondition(boolean condition, String msg) {
        // ...
    }

    default void throwAorB(Throwable a, Throwable b, boolean throwA) {
        // ...
    }
}
```

然后每个有需要的类都可以简单的实现这个接口。

格式化

格式化比起大多数程序员所认为的更加不被重视。那么它是不是同你对于自己技术水平的在意目标一致，还有是不是能有助于其他人的对于代码的解读呢？当然是。但我们也不要浪费一整天加空格来使得 if 的括号能“匹配”。

如果你绝对需要一个代码格式手册，我强烈推荐 [Google 的 Java 代码风格](#) 指南。该指南的最佳部分就是[编程实践](#)这一节。绝对值得一读。

Javadoc

为你的用户所要面对的代码加注文档是很重要的。而这就意味着要使用[示例](#)和对于变量、方法和类的极值描述。这样做的必然结果就是对于不需要加注文档的就不要去加注文档。如果就一个参数代表的是什么你不想多费口舌，因为答案很明显，就不要为其加注文档。样板一样的文档比没有文档更糟糕，因为这对于会思考此处为何要加注的文档的用户而言这会是一种戏弄。

流

[Java 8](#) 有了一个不错的[流](#)和 lambda 语法。你可以像下面这样编写代码：

```
final List<String> filtered = list.stream()
    .filter(s -> s.startsWith("s"))
    .map(s -> s.toUpperCase());
```

而不是再像以前这样写：

```
final List<String> filtered = Lists.newArrayList();for (String str : list) {
    if (str.startsWith("s") {
        filtered.add(str.toUpperCase());
    }
}
```

这就让你能写出更加流畅的代码，更具可读性。

发布

发布 Java 通常有点棘手。如今有两种主要的 Java 发布方式：使用一套框架，或者根据灵活性的本地增量方案。

框架

因为发布 Java 并不容易，现有的框架可能会有所帮助。最好的两个就是 [Dropwizard](#) 和 [Spring Boot](#)。

[Play 框架](#) 也可以被考虑也作为这些部署框架的其中之一。

它们全都试图降低让你的代码发布出去的门槛。它们在你是名 Java 新手或者希望能快速运行起来时特别有帮助。单个的 JAR 部署比复杂的 WAR 和 EAR 部署更简单。

不过，它们可能不怎么灵活，而且简单笨拙，因此如果你的项目不适合框架开发者为你的框架所做出选择，你就得自己集成一个更加手动的配置了。

Maven

好的选择： [Gradle](#)。

Maven 仍然是构建，打包并运行你的测试的标准工具。不过还有其它可选项，比如 Gradle，但是它们并不像 Maven 那样为人们所接受。如果你是 Maven 新手，你应该通过 [示例](#) 来上手 Maven。

我喜欢能有一个根 POM，里面有你想要使用的所有的外部依赖包。它看起来像是 [这样的](#)。这个根 POM 只有一个外部依赖，而如果你自己的项目足够大，就会有很多个。你的根 POM 自身可能也是一个项目：收到版本控制中并且像其它的 Java 项目那样进行发布。

如果你想过要为你的根 POM 标记出的每一个外部依赖的变化太多了，你不必浪费一个星期去跟踪调试多个项目的依赖错误。

你的所有的 Maven 项目都将包含你的根 POM，以及他所有的版本信息。这样，你就能得到你的公司所选择的每一个外部依赖的版本，以及所有的正确的 Maven 插件。如果你需要拉入外部依赖，就会像下面这样运作：

```
<dependencies>
  <dependency>
    <groupId>org.third.party</groupId>
    <artifactId>some-artifact</artifactId>
  </dependency>
</dependencies>
```

如果你想要外部的依赖，那就应该被每一个独立的项目部分管理起来。否则就很难保持根 POM 的有序性。

依赖收敛

Java 最好的部分就是大量的第三方库能帮助你做任何事情。基本上每一个 API 或者工具包都有一个 Java SDK，并且很容易用 Maven 获取。

而那些 Java 库自身则还要依赖于其它的特定版本的库。如果你引入了够多的库，就会发生版本冲突，那会像下面这样：哪个版本会引入到你的项目中呢？

```
Foo library depends on Bar library v1.0
```

```
Widget library depends on Bar library v0.9
```

使用 [Maven 的依赖收敛插件](#)，构建就会在你的依赖没有使用相同的版本时报错。之后要解决冲突，你可以有两种选择：

1. 在你的 *dependencyManagement* 一节为 Bar 明确挑选一个版本
2. 将 Bar 从 Foo 或者 Widget 中排除出去

选择哪种方案要视你的情形而定：如果你想要跟踪一个项目的版本，那么就用排除的方案。另外一方面，如果你想要明确的指定它，你就可以挑选一个版本，虽然你将需要在更新其它依赖的同时对它进行更新。

持续集成

显然你需要一些持续集成的服务来让你连续不断地建立你的 SNAPSHOT (快照) 版本和建立基于 git 标签的 tag。

[Jenkins](#) 和 [Travis-CI](#) 是自然的选择。

代码覆盖率测试是有用的，并且 [Cobertura](#) 有一个很好的 [Maven 插件](#) 并对 CI 提供支持。还有其他 Java 的代码覆盖率工具，但我是使用 Cobertura 的。

Maven 资源库

你需要一个地方放置你的 jar 包，war 包和 ear 包，因此你需要一个资源库。

通常的选择是 [Artifactory](#) 和 [Nexus](#)。都很有用，且都有他们自己的 [优劣势](#)。

你应该有一个自己安装的 Artifactory/Nexus 并且上面有你的[依赖的镜像](#)。这样你的工作就不会因为在线的 Maven 资源库挂掉而中断。

配置管理

现在你已经把代码编译好了，资源库也设置好了，要做的就是让你开发环境的代码最后放到生产上去。在这儿不要偷懒，因为自动化一些东西将会给你带来长久的好处。

[Chef](#), [Puppet](#), 和 [Ansible](#) 是典型的选择。我也编写过一个叫做 [Squadron](#) 的可选方案，当然我认为你应该拿来看看，因为它比其他选择更容易上手。

不管你选择的哪个工具，都不要忘了对你的部署操作进行自动化。

库

可能 Java 最棒的特性就是它所拥有的大量的库。这里是可能大部分人都会用到的一些库的集合。

缺少的功能特性

Java 的标准上曾经踏出了了不起的一步，现在看起来则缺少了几个关键的功能特性。

Apache Commons

[Apache Commons](#) 项目有一堆实用的库。

Commons Codec 有许多针对 Base64 和 16 进制字符串的编码/解码方法。你就不要再浪费时间再去重新编写他们了。

Commons Lang 是针对 String 的创建和操作，字符集以及一堆实用工具方法的库。

Commons IO 拥有你可以想象得到的所有文件相关的方法。它有 [FileUtils.copyDirectory](#) , [FileUtils.writeStringToFile](#) , [IOUtils.readLine](#) 以及更多的东西。

Guava

[Guava](#) 是 Google 的优秀补充 Java 所缺的库。几乎很难提交我所喜欢的有关于这个库的每个功能，但我会试试。

Cache 是获取一个内存缓存的简单方法，可以被用来缓存网络访问，磁盘访问，记忆函数或者任何实在的数据。只要实现一个 [CacheBuilder](#) 就能告诉 Guava 如何去构建你的缓存，一切尽在你的掌握之中！

Immutable 集合。有一堆这样东西：[ImmutableMap](#),[ImmutableList](#), 或者如果那是你的风格的话，就还有 [ImmutableSortedMultiSet](#) .

我也喜欢用 Guava 的方式编写不可变的集合：

```
// Instead of
final Map<String, Widget> map = new HashMap<String, Widget>();

// You can use
final Map<String, Widget> map = Maps.newHashMap();
```

还有针对 [Lists](#), [Maps](#), [Sets](#) 以及更多集合的静态类。他们更清晰和可读。

如果你还在 Java 6 或者 7 的坑里面，你可以使用 [Collections2](#) 类，它拥有像 filter 和 transform 这样的方法。能让你在没有 [Java 8](#) 对流的支持下写出流畅的代码。

Guava 也有一些简单的东西，比如 **Joiner** 能用分隔符将字符串连接起来，以及一个通过忽略它们来 [处理中断的类](#)。

Gson

Google 的 [Gson](#) 库是一个简单快速的 JSON 转换库。像下面这样运作：

```
final Gson gson = new Gson();
```

```
final String json = gson.toJson(fooWidget);  
final FooWidget newFooWidget = gson.fromJson(json, FooWidget.class);
```

相当简单且令人愉悦。[Gson 用户手册](#) 有许多的示例。

Java Tuples

Java 经常令我头疼的一点就是他的标准库里面并没有内置元组。幸运的是，[Java tuples](#) 项目解决了这个问题。

它易于使用而且表现很棒：

```
Pair<String, Integer> func(String input) {  
    // something...  
    return Pair.with(stringResult, intResult);}
```

Joda-Time

[Joda-Time](#) 是我所使用过的最棒的时间库。简答，直接，易于测试。夫复何求？

所以你只要在如果没有使用 Java8 时使用这个库，因为 Java8 有了新的 [日期时间](#) 库。

Lombok

[Lombok](#) 是一个有趣的库。它通过注解让你减少 Java 所严重遭受的样板式代码。

想要为你的类变量加入设置器和获取器？简单：

```
public class Foo {  
    @Getter @Setter private int var;}
```

现在你可以这样做：

```
final Foo foo = new Foo();foo.setVar(5);
```

还有 [更多的东西](#)。我还没有将 Lombok 用于生产环境，但我迫不及待的想要这么做了。

Play framework

好的选择： [Jersey](#) 或者 [Spark](#)

在 Java 中实现 RESTful web 服务又两个主要的阵营：[JAX-RS](#) 和其余一切。

JAX-RS 是传统的方式。你可以使用诸如 [Jersey](#) 之类的东西来将注解结合接口和实现来组织 web 服务。这里有意思的是你可以简单的从接口类创建出客户端。

[Play framework](#) 是 JVM 的 web 服务的一个异类：你会有一个路由文件，然后你要编写在这些路由中被引用的类。它实际上是一个完整的 [MVC 框架](#)，但是你可以简单地只把他用于 REST web 服务。

它在 Java 和 Scala 上都能用。它稍有偏向 Scala 优先，不过在 Java 中也还好。

如果你用过 Python 中向 Flash 这样的微型框架，你就能熟悉 [Spark](#)。它在 Java 8 上能运行得很好。

SLF4J

有许多 Java 日志的解决方案。我喜欢的就是 [SLF4J](#) 因为它的极度的可插入性，并且可以同时结合来自不同日志框架的日志。有没有过一个使用了 `java.util.logging`, JCL, 以及 `log4j` 的古怪项目? SLF4J 为你而生。

[两页篇幅的操作手册](#) 就是你入门所需要的。

jOOQ

我不想换重量级的 ORM 框架，因为我喜欢 SQL。因此我写了许多 [JDBC 模板](#)，而这样就有点难以维护。[jOOQ](#) 是一个更好的解决方案。

他能让你用 Java 以一种更加类型安全的方式编写 SQL:

```
// Typesafely execute the SQL statement directly with jOOQ

Result<Record3<String, String, String>> result =
create.select(BOOK.TITLE, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
    .from(BOOK)
    .join(AUTHOR)
    .on(BOOK.AUTHOR_ID.equal(AUTHOR.ID))
    .where(BOOK.PUBLISHED_IN.equal(1948))
    .fetch();
```

使用这个和 [DAO](#) 模式，你可以使得访问数据库变得轻而易举。

测试

测试对于你的软件至关重要。这些包能使得测试更简单。

jUnit 4

[JUnit](#) 不需要介绍了。它是 Java 中单元测试的标准工具。

而你可能不会发挥 JUnit 的全部潜能。JUnit 支持 [参数化测试](#)，让你不用编写过多样板代码的 [规则](#)，随机测试特定代码的 [理论](#)，以及 [假设](#)。

jMock

如果依赖注入那块你弄好了的话，这时候它就能发挥点作用了：模拟那些有副作用的代码（比如同一个 REST 服务器进行交互）并且仍然可以对调用它的代码进行断言。

[jMock](#) 是标准的 Java 模拟工具。它看起来像这样：

```
private Mockery context = new Mockery();
@Test
public void basicTest() {
    final FooWidgetDependency dep = context.mock(FooWidgetDependency.class);

    context.checking(new Expectations() {{
        oneOf(dep).call(with(any(String.class)));
        atLeast(0).of(dep).optionalCall();
    }});

    final FooWidget foo = new FooWidget(dep);

    Assert.assertTrue(foo.doThing());
    context.assertIsSatisfied();
}
}
```

这里通过 jMock 设置了一个 *FooWidgetDependency*，然后加入了一个预期（*expectation*）。我们预期 *dep* 的 *call* 方法会使用某个字符串被调用一次并且 *dep* 的 *optionalCall* 方法会被调用 0 到多次。

如果你要一次又一次设置相同的依赖，你可能应该将那些放到一个 [测试夹具 \(test fixture\)](#) 中，并将 *assertIsSatisfied* 放到一个 *@After* 夹具中。

AssertJ

你曾经用 jUnit 这样做过吗？

```
final List<String> result = some.testMethod();
assertEquals(4, result.size());
assertTrue(result.contains("some result"));
assertTrue(result.contains("some other result"));
assertFalse(result.contains("shouldn't be here"));
```

这都是烦人的样板代码。[AssertJ](#) 会把这些都干掉。你可以将同样的代码转换成这样：

```
assertThat(some.testMethod()).hasSize(4)
    .contains("some result", "some other result")
    .doesNotContain("shouldn't be here");
```

这样流畅的接口让你的测试更加可读。夫复何求？

工具

IntelliJ IDEA

好的选择: [Eclipse](#) 和 [Netbeans](#)

最好的 Java IDE 是 [IntelliJ IDEA](#)。它有大量超赞的功能特性，并且真正让开发 Java 相关的所有细节都暴露无遗。自动补全很棒，[检查也是顶尖的](#)，还有重构工具真的很有帮助。

免费的社区版本对我而言已经很好的，而旗舰版本则还有许多很棒的功能，比如数据库工具，对 Spring Framework 和 Chronon 的支持。

Chronon

我最喜爱的 GDB 7 的一个功能就是调试的时候可以倒着走。这在你获取到旗舰版本并且使用了 [Chronon IntelliJ 插件](#) 时是可能的。

你可以获取到变量的历史，后退，方法的历史以及更多其它的东西。初次使用会有点怪，但它能帮助你解决一些非常复杂的问题，诸如此类的海森堡 bug。

JRebel

持续集成常常是软件即服务产品的目标。如果你不想等待编译构建结束就看到代码变化所产生的效果呢？

那正是 [JRebel](#) 所要做的。一旦你把你的服务器挂到了你的 JRebel 客户端，你就可以实时看见服务器上的变化。当你想要快速地进行试验时这能节省大量的时间。

检查框架 (Checker Framework)

Java 的类型系统很不咋地。它不能区分字符串和实际上是正则表达式的字符串，也没有做 [坏点检查](#)。不过，[Checker Framework](#) 能做到并且能做到更多。

它使用像 `@Nullable` 这样的注解来检查类型。你甚至可以 [自己定义注解](#) 来是的静态分析如虎添翼。

Eclipse 内存分析器

即使是在 Java 中，内存也会发生泄漏。幸运的是，有针对这个问题的工具。我用过的最好的解决这些问题的工具就是 [Eclipse 内存分析器](#)。它能获取到堆栈，让你可以找出问题何在。

有几种方法可以获取到一个 JVM 进程的堆栈，而我使用的是 [jmap](#)：

```
$ jmap -dump:live,format=b,file=heapdump.hprof -F 8152
Attaching to process ID 8152, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 23.25-b01
Dumping heap to heapdump.hprof ...
... snip ...
Heap dump file created
```

之后你就可以用内存分析器打开 *heapdump.hprof* 文件，并快速的看到到底发生了什么。

资源

能帮助你成为 Java 大师的资源

数据

- [Effective Java](#)
- [Java 并发实践](#)

博客

- [The Java Posse](#)