目　录

致谢

当前文档 《Better Java(中文版)》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-03-25。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/better-java

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

Better Java

Java 虽作为最流行的编程语言之一，但是似乎并没有什么人很享受用它。好吧，Java 确实是这样的一门编程语言，从最近发布不久的 Java 8 开始，为了更好的使用 Java，我决定收集一些库，实践和工具等相关资料。“更好” 是主观的，所以推荐使用我所说的建议的某些部分，而不是一下子全部按照这些建议来做。请尽情添加其他意见并提交 PR。

这篇文章原始发布在：
我的博客.

来源

> *https://github.com/cxxr/better-java*

Style（风格）

Style

Java 传统的代码风格是被用来编写非常复杂的企业级 JavaBean。新的代码风格看起来会更加整洁，更加正确，并且更加简单。

Structs

对我们程序员来说，包装数据是最简单的事情之一。下面是传统的通过定义一个 JavaBean 的实现方式：

```
1. public class DataHolder {
2.     private String data;
3. 
4.     public DataHolder() {
5.     }
6. 
7.     public void setData(String data) {
8.         this.data = data;
9.     }
10. 
11.     public String getData() {
12.         return this.data;
13.     }
14. }
```

这种方式既繁琐又浪费代码。即使你的 IDE 可以自动生成这些代码，也是浪费。因此，[别这么干]

[dontbean].

相反，我更喜欢 C 语言保存数据的风格来写一个类：

```
1. public class DataHolder {
2.     public final String data;
3.
4.     public DataHolder(String data) {
5.         this.data = data;
6.     }
7. }
```

这样不仅减少了近一半的代码行数。并且，这个类里面保存的数据除了你去继承它，否则不会改变，由于它不可变性，我们可以认为这会更加简单。

如果你想保存很容易修改的对象数据，像 Map 或者 List，你应该使用 ImmutableMap 或者 ImmutableList，这些会在不变性那一部分讨论。

The Builder Pattern

如果你想用这种构造的方式构造更复杂的对象，请考虑构建器模式。

你可以建一个静态内部类来构建你的对象。构建器构建对象的时候，对象的状态是可变的，但是一旦你调用了 build 方法之后，构建的对象就变成了不可变的了。

想象一下我们有一个更复杂的 *DataHolder*。那么它的构建器看起来应该是这样的：

```
1. public class ComplicatedDataHolder {
2.     public final String data;
3.     public final int num;
4.     // lots more fields and a constructor
5.
6.     public static class Builder {
7.         private String data;
8.         private int num;
9.
10.        public Builder data(String data) {
11.            this.data = data;
12.            return this;
13.        }
14.
15.        public Builder num(int num) {
16.            this.num = num;
17.            return this;
18.        }
19.
```

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

然后调用它：

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

这有[关于构建器更好的例子][builderex]，他会让你感受到构建器到底是怎么回事。它没有使用许多我们尽力避免使用的样板，并且它会给你不可变的对象和非常好用的接口。

可以考虑下在众多的库中选择一个来帮你生成构建器，取代你亲手去写构建器的方式。

Immutable Object Generation

如果你要手动创建许多不可变对象，请考虑用注解处理器的方式从它们的接口自动生成。它使样板代码减少到最小化，减少产生 bug 的可能性，促进了对象的不可变性。看这 presentation 有常见的 Java 设计模式中一些问题的有趣的讨论。

一些非常棒的代码生成库如 [immutables]
(https://github.com/immutables/immutables), 谷歌的
auto-value 和
[Lombok][lombok]

Exceptions

使用[检查][checkedex]异常的时候一定要注意，或者干脆别用。它会强制你去用 try/catch 代码块包裹住可能抛出异常的部分。比较好的方式就是使你自定义的异常继承自运行时异常来取而代之。这样，可以让你的用户使用他们喜欢的方式去处理异常，而不是每次抛出异常的时候都强制它们去处理/声明，这样会污染代码。

一个比较漂亮的绝招是在你的方法异常声明中声明 RuntimeExceptions。这对编译器没有影响，但是可以通过文档告诉你的用户在这里可能会有异常抛出。

Dependency injection

在软件工程领域，而不仅是在 Java 领域，使用[依赖注入][di]是编写可测试软件最好的方法之一。

由于 Java 强烈鼓励使用面向对象的设计，所以在 Java 中为了开发可测试软件，你不得不使用依赖注入。

在 Java 中，通常使用[Spring 框架][spring]来完成依赖注入。Spring 有基于代码的和基于 XML 配置文件的两种连接方式。如果你使用基于 XML 配置文件的方式，注意不要[过度使用 Spring][springso]，正是由于它使用的基于 XML 配置文件的格式。在 XML 配置文件中绝对不应该有逻辑或者控制结构。它应该仅仅被用来做依赖注入。

使用 Google 和 Square 的 [Dagger][dagger] 或者 Google 的 [Guice][guice] 库是 Spring 比较好的替代品。它们不使用像 Spring 那样的 XML 配置文件的格式，相反它们把注入逻辑以注解的方式写到代码中。

Avoid Nulls

尽量避免使用空值。不要返回 null 的集合，你应该返回一个 empty 的集合。如果你确实准备使用 null 请考虑使用 [@Nullable][nullable] 注解。[IntelliJ IDEA][intellij] 内置支持 @Nullable 注解。

阅读[计算机科学领域最糟糕的错误][the-worst-mistake-of-computer-science]了解更多为何不使用 null。

如果你使用的是 Java 8，你可以用新出的优秀的 [Optional][optional] 类型。如果有一个值你不确定是否存在，你可以像这样在类中用 *Optional* 包裹住它们：

```
1.  public class FooWidget {
2.      private final String data;
3.      private final Optional<Bar> bar;
4.
5.      public FooWidget(String data) {
6.          this(data, Optional.empty());
7.      }
8.
9.      public FooWidget(String data, Optional<Bar> bar) {
10.         this.data = data;
11.         this.bar = bar;
12.     }
13.
14.     public Optional<Bar> getBar() {
15.         return bar;
16.     }
17. }
```

这样，现在你可以清晰地知道 *data* 肯定不为 null，但是 *bar* 不清楚是不是存在。*Optional* 有如 *isPresent* 这样的方法，可以用来检查是否为 *null*，感觉和原来的方式并没有太大区别。但是它允许你可以这样写：

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

这样比写一连串的判断是否为空的检查代码更好。使用 Optional 唯一不好的是标准库对 Optional 的支持并不是很好，所以对 null 的处理仍然是必要的。

Immutable-by-default

变量，类和集合应该设置为不可变的，除非你有很好的理由去修改他们。

变量可以用 *final* 关键字使起不可变：

```
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
```

现在你可以确定 fooWidget 对象不会意外地被重新赋值了。*final* 关键词也可以在 if/else 和 try/catch 代码块中使用。当然，如果 *fooWidget* 对象本身不是不可变的，你可以很容易去修改它。

使用集合的时候，任何可能的情况下尽量使用 Guava 的 [ImmutableMap][immutablemap], [ImmutableList][immutablelist], 或者 [ImmutableSet][immutableset] 类。这些类都有构建器，你可以很容易地动态构建集合，一旦你执行了 build 方法，集合就变成了不可变的。

类应该声明不可变的字段（通过 *final* 实现）和不可变的集合使该类不可变。或者，可以对类本身使用 *final* 关键词，这样这个类就不会被继承也不会被修改了。

Avoid lots of Util classes

如果你发现在你正在往工具类中添加很多方法，就要注意了。

```
public class MiscUtil {
```

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

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

乍一看这些工具类似乎很不错，因为里面的那些方法放在别处确实都不太合适。因此，你以可重用代码的名义全放这了。

这个想法比本身这么做还要糟糕。请把这些类放到它应该在的地方去并积极重构。不要命名一些像“MiscUtils”或者“ExtrasLibrary”这样的很普通的类，包或者库。这会鼓励产生无关代码。

Formatting

格式化代码对大多数程序员来说并没有它应有的那么重要。统一化你的代码格式对阅读你的代码的人有帮助吗？当然了。但是别在为了 if 代码块匹配添加空格上耗一天。

如果你确实需要一个代码格式风格的教程，我高度推荐 [Google’s Java Style][googlestyle] 这个教程。写的最好的部分是 [Programming Practices][googlepractices]。绝对值得一读。

Javadoc

文档对对你代码的阅读着来说也很重要。这意味着你要给出[使用示例][javadocex]，并且给出你的变量，方法和类清晰地描述。

这样做的必然结果是不要对不需要写文档的地方填写文档。如果你对一个参数的含义没什么可说的，或者它本身已经很明显是什么意思了，就不要为其写文档了。统一样板的文档比没有文档更加糟糕，这样会让读你代码的人误以为那就是文档。

Streams

[Java 8][java8] 有很棒的 [stream][javastream] and lambda 语法。你可以像这样来写代码：

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

取代这样的写法：

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

它让你可以写更多的流畅的代码，并且可读性更高。

Deploying（部署）

Deploying

Java 的部署问题确实有点棘手。现如今有两种主流的方式：使用框架或者灵活性更高的内部研发的解决方案。

Frameworks

由于 Java 的部署并不容易，所以使用框架还是很有帮助的。最好的两个框架是 [Dropwizard][dropwizard] 和 [Spring Boot][springboot]。[Play 框架][play] 也可以被看作为一种部署框架。

这些框架都是尽力地降低你部署代码的壁垒。它们对 Java 新手或者想提高效率的人尤有帮助。单独的 JAR 包部署会比复杂的 WAR 包或者 EAR 包部署更简单一点。

然而，这些框架并没有你想象的那么灵活，如果你的项目的开发者选择的框架并不合适，你不得不迁移到手动配置更多的部署方案上来。

Maven

不错的替代工具: [Gradle][gradle].

Maven 仍然是构建，打包和测试的标准。有很多不错的替代工具，如 Gradle，但是他们同样都没有像 Maven 那样的适应性。如果你是 Maven 新手，你应该从[Maven 实例][mavenexample]这里开始。

我喜欢用一个根 POM（Project Object Model，项目对象模型）来管理所有用到的外部依赖。它会像[这个样子][rootpom]。这个根 POM 仅仅包含一个外部依赖，但是如果你的产品足够大，你将会有几十个外部依赖了。你的根 POM 应该像其他 Java 项目一样采用版本控制和发布的方式，有一个自己的项目。

如果你认为你的根 POM 每添加一个外部依赖都打上一个标签很麻烦，那你肯定没有遇到过为了排查

依赖错误引起的问题，浪费一周的时间翻遍整个项目的情况。

你所有的 Maven 项目都应该包含你的根 POM，以及这些项目的所有版本信息。这样你会清除地了解到你们公司选择的每一个外部依赖的版本，以及所有正确的 Maven 插件。如果你要引入很多的外部依赖，它将会是这样子的：

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

如果你想使用内部依赖，它应该被每一个单独项目的 部分来管理。否则那将会很难保持根 POM 的版本号是正常的。

Dependency Convergence

Java 最好的一方面就是拥有大量的第三方库可以做任何事。基本上每一个 API 或者工具包都有一个 Java SDK，可以很方便的用 Maven 引入。

并且这些第三方 Java 库本身依赖特定版本的其他的库。如果你引入足够多的库，你会发现有些库的版本是冲突的，像这样：

```
Foo library depends on Bar library v1.0
Widget library depends on Bar library v0.9
```

你的项目到底要引入哪一个版本呢？

如果你的项目依赖于不同版本的同一个库，使用 [Maven 依赖趋同插件][depconverge]构建时将会报错。然后你有两个方案来解决这个冲突：

1. 在你的 *dependencyManagement* 部分明确地支出你所使用的 Bar 的版本号
2. 在 FOO 或者 Widget 中排除对 Bar 的依赖。

这两个方案到底选哪一个要看你面对的是什么情况：如果你想跟踪一个项目的版本，那么选择排除的方案是不错的。另一方面，如果你想明确地指出它，你可以选择一个版本，尽管你在需要更新其他依赖的时候也需要更新它。

Continuous Integration

很明显，你需要某种形式的持续集成服务器来帮你不断构建你的快照版本和基于 git 标签构建。

[Jenkins][jenkins] 和 [Travis-CI][travis] 就成了很自然的选择.

代码覆盖率非常有用，[Cobertura][cobertura] 就有 [一个很好的 Maven 插件][coberturamaven]
[a good Maven plugin][coberturamaven] 并且支持 CI。还有一些其他的支持 Java 的代码覆盖率工具，但是我只用过 Cobertura。

Maven repository

你需要一个地方存储你生成的 JAR 包，WAR 包或者 EAR 包，因此，你需要一个仓库。

一般选择有 [Artifactory][artifactory] 和 [Nexus][nexus] 这两个。它们都可以用，但是它们都有着各自的优缺点。

你应该有自己的 Artifactory/Nexus 设备和[镜像][artifactorymirror] 使你的依赖基于此。这样就不会由于上游的 Maven 库宕机而使你的构建崩溃了。

Configuration management

现在，你的代码已经编译完了，你的仓库也跑起来了，最终你需要把你的代码从开发环境部署到生产环境了。到了这里，千万不要吝啬，因为将来很长一段时间，你会从这些自动化方式中尝到很多的甜头。

[Chef][chef]，[Puppet][puppet]，和 [Ansible][ansible] 是很典型的选择。我曾经也写了一个叫 [Squadron][squadron] 的也可供选择，当然，我认为你应该仔细看看这个，因为它使用起来比其他的更为简单方便。

无论你选择了什么工具，不要忘了使你的部署实现自动化。

Libraries（库）

Libraries

对 Java 来说，拥有大量的扩展库也许是最大的特点了。下面这些一小部分的扩展库对大部分人来说很适用的。

Missing Features

Java 标准库曾经作出过惊人的改进，但是现在来看，它仍然缺少一些关键的特性。

Apache Commons

[Apache Commons 项目][apachecommons] 拥有大量的有用的扩展库。

Commons Codec 对 Base64 和 16 进制字符串来说有很多有用的编/解码方法。不要再浪费时间重写这些东西了。

Commons Lang 有许多关于字符串的操作和创建，字符集和许多各种各样的实用的方法。

Commons IO 拥有所有你能想到的关于文件操作的方法。有 [FileUtils.copyDirectory][copydir], [FileUtils.writeStringToFile] [writestring], [IOUtils.readLines][readlines] 和更多实用的方法。

Guava

[Guava][guava] 是谷歌优秀的对 Java 标准库缺少的特性进行补充的扩展库。虽然这很难提炼总结出我有多喜欢这个库，但是我会尽力的。

Cache 让你可以用很简单的方法，实现把网络访问，磁盘访问，缓存函数或者其他任何你想要缓存的内容，缓存到内存当中。你仅仅只需要实现 [CacheBuilder][cachebuilder] 类并且告诉 Guava 怎么样构建你的缓存，一切就搞定了！

Immutable 集合。它有许多如：[ImmutableMap][immutablemap]，[ImmutableList][immutablelist]，或者甚至 [ImmutableSortedMultiSet][immutablesorted] 等不可变集合可以使用，如果你喜欢用这种风格的话。

我也喜欢用 Guava 的方式来写一些可变的集合：

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

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

它还有一些静态类如 [Lists][lists]，[Maps][maps]和[Sets][sets] 等。使用起来它们显得更整洁，并且可读性更强。

如果你坚持使用 Java 6 或者 7 的话，你可以使用 [Collections2][collections2] 这个类，它有一些像 filter 和 transform 这样的方法。能够让你没有 Java 8 的 Stream 的支持也能写出流畅的代码。

Guava 也可以做一些很简单的事情，比如 **Joiner** 类可以用来用分隔符把字符串拼接起来，并且可以用忽略的方式[来处理打断程序][uninterrupt]的数据。

Gson

谷歌的 [Gson][gson] 库是一个简单快速的 JSON 解析库。可以这样用：

```
final Gson gson = new Gson();
final String json = gson.toJson(fooWidget);

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

这用起来真的很简单，很愉悦。[Gson 用户手册][gsonguide] 有很多的使用示例。

Java Tuples

Java 令我比较烦恼的问题之一 Java 标准库中没有内置对元组的支持。幸运的是，[Java tuples][javatuples] 项目解决了这个问题。

它使用用起来很简单，很棒：

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

Javaslang

[Javaslang][javaslang] 是一个函数式编程库，它被设计用来弥补本应该出现在 Java 8 中但缺失的一些特性。它有这样的一些特点：

- 一个全新函数式集合库
- 紧密集成的元组功能
- 模式匹配
- 通过不可变性保证线程安全
- 饥汉式和懒汉式的数据类型
- 通过 Option 实现了 null 的安全性
- 通过 Try 更好的实现异常处理

有一些 Java 库依赖于原始的 Java 集合类。它们通过以面向对象和被设计为可变的方式来保证和其他的类的兼容性。而 Javaslang 的集合的设计灵感来源于 Haskell, Clojure 和 Scala，是一个全新的飞跃。它们被设计为函数式风格并且遵循不可变性的设计风格。

像下面这样的代码就可以自动实现线程安全，并且不用 try-catch 语句处理异常：

```
// Success/Failure containing the result/exception
public static Try<User> getUser(int userId) {
    return Try.of(() -> DB.findUser(userId))
        .recover(x -> Match.of(x)
            .whenType(RemoteException.class).then(e -> ...)
            .whenType(SQLException.class).then(e -> ...));
}

// Thread-safe, reusable collections
public static List<String> sayByeBye() {
    return List.of("bye, "bye", "collect", "mania")
                .map(String::toUpperCase)
                .intersperse(" ");
}
```

Joda-Time

[Joda-Time][joda] 是我用过的最简单的时间处理库。简单，直接，并且很容易测试。夫复何求？

因为 Java 8 已经有了自己的新的 [时间处理][java8datetime]库， 所以如果你还没有用 Java 8，你需要这一个库足矣。

Lombok

[Lombok][lombok] 是一个很有意思的库。它可以让你以注解的方式减少 Java 中糟糕的样板代码。

想为你的类的变量添加 setter 和 getter 方法吗？像这样：

```
1. public class Foo {
2.     @Getter @Setter private int var;
3. }
```

现在你就可以这么用了：

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

这还有[很多][lombokguide]例子。我在之前的产品中还没有用过 Lombok，但是现在我等不急了。

Play framework

好的替代品：[Jersey][jersey] 或者 [Spark][spark]

在 Java 实现 RESTful web services 有两大主要阵营：[JAX-RS][jaxrs] 和其他。

JAX-RS 是传统的实现方式。你可以用像 [Jersey][jersey] 这样的框架，以注解的方式来实现接口及其实现的结合。这样你就可以很容易的根据接口类来开发客户端。

[Play 框架][play] 基于 JVM 的 web services 实现和其他根本框架不同：它有一个路由文件，你写的类要和路由文件中的路由信息关联起来。Play 框架其实是一个[完整的 MVC 框架][playdoc]，但是你可以很简单地仅仅使用它的 REST web services 部分的功能。

它同时支持 Java 和 Scala。虽然对重点支持的 Scala 稍有不足，但是对 Java 的支持还是很好用的。

如果你在 Python 中用过像 Flask 这样的微框架，你对 [Spark][spark] 肯定会很熟悉。它对

Java 8 的支持尤其的好。

SLF4J

有很多 Java 日志解决方案。我最喜欢的是 [SLF4J][slf4j]，因为它拥有非常棒的可插拔性，同时能够和很多的日志框架想结合。有没有做过同时使用 java.util.logging，JCL，和 log4j 的奇葩项目？SLF4J 就是为你而生。

这[两页手册][slf4jmanual]足够你可以开始入门使用 SLF4J 了。

jOOQ

我不喜欢重量级的 ORM 框架，因为我喜欢 SQL。所以我写了很多 [JDBC 模板][jdbc]，但是很难去维护它。[jOOQ][jooq] 是一个更好的解决方案。

它让你在 Java 中用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 jOOQ 和 [DAO][dao] 的模式让你的数据库访问变得轻而易举。

Testing

测试是软件的关键环节。下面这些软件包能够让你更容易地测试。

jUnit 4

好的替代品：[TestNG][testng].

[jUnit][junit] 就无需多言了。它是 Java 单元测试中的标准工具。

但是很可能你使用的 jUnit 并没有发挥它的全部潜力。jUnit 支持[参数化测试][junitparam]，[规则化][junitrules]测试，[theories][junittheories] 可以随机测试特定代码，还有 [assumptions][junitassume]，可以让你少写很多样板代码。

jMock

如果你完成了依赖注入，这是它的回报：可以 mock 出有副作用（比如和 REST 服务器交互）的代码，并且可以断言调用这段代码的行为。

[jMock][jmock] 是标准的 Java mock 工具。像这样使用：

```
1. public class FooWidgetTest {
2.     private Mockery context = new Mockery();
3. 
4.     @Test
5.     public void basicTest() {
6.         final FooWidgetDependency dep = context.mock(FooWidgetDependency.class);
7. 
8.         context.checking(new Expectations() {{
9.             oneOf(dep).call(with(any(String.class)));
10.             atLeast(0).of(dep).optionalCall();
11.         }});
12. 
13.         final FooWidget foo = new FooWidget(dep);
14. 
15.         Assert.assertTrue(foo.doThing());
16.         context.assertIsSatisfied();
17.     }
18. }
```

这段代码通过 jMock 建立了一个 *FooWidgetDependency*，然后添加你所期望结果的条件。我们期望 *dep* 的 *call* 方法会被以一个字符串为参数的形式调用，并且会被调用 0 次或者多次。

如果你想一遍又一遍地设置相同的依赖，你应该把它放到 [test fixture][junitfixture] 中，并且把*assertIsSatisfied* 放在以 *@After* 注解的 fixture 中。

AssertJ

你曾经用 jUnit 干过这个吗？

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

这是很恶心的样板代码。[AssertJ][assertj] 可以解决这个问题。你可以把相同的代码转换成这个样子：

```
1. assertThat(some.testMethod()).hasSize(4)
2.                              .contains("some result", "some other result")
```

```
3.                              .doesNotContain("shouldn't be here");
```

这样的流畅接口让你的测试更具有可读性。你还想咋地？

Tools（工具）

Tools

IntelliJ IDEA

好的替代品：[Eclipse][eclipse] 和 [Netbeans][netbeans]

Java 最好的 IDE 是 [IntelliJ IDEA][intellij]。它有大量的牛逼的特性，它是真正的能让 Java 用来像不戴套做爱那么爽的工具。自动完成功能超棒，[代码检查功能也是顶尖的][intellijexample]，重构工具那是相当有帮助。

免费的社区版对我来说已经足够好了，但是它的旗舰版加载了更多的牛逼的特性，如数据库工具，Spring 框架的支持和对 Chronon 的支持。

Chronon

我最喜欢 GDB 7 的特性之一就是调试的时候能够按照时间跟踪回来。当你拥有了旗舰版的 IntelliJ，你可以通过安装 [Chronon IntelliJ 插件][chronon]实现。

你可以获取到变量的变化历史，后退，方法的历史以及更多的信息。如果你是第一次用会觉得有点怪，但是它真的能够帮你解决很复杂的 bug，诸如海森堡类的 bug。

JRebel

好的替代品：DCEVM

持续集成往往以软件即服务为产品目标。想象一下如果你不用等待代码构建完成而能实时看到代码的变化会是怎样？

这就是 [JRebel][jrebel] 所做的。一旦你将你的服务器和你的 JReble 以 hook 方式连接，你就可以从服务器看到实时变化。当你想快速试验的时候它能为你节省大量的时间。

The Checker Framework

Java 的类型系统很差劲。它不能够区分正常的字符串和正则表达式字符串，更不用说[坏点检查][taint]了。不过 [Checker Framework][checker] 可以完成这个功能并且能够实现更多的东西。

它使用像 *@Nullable* 这样的注解来检查类型。你甚至可以使用[自定义注解][customchecker]来实现静态分析，甚至更强大的功能。

Code Quality

即使遵循着最佳实践的原则，即使是最好的开发者，也都会犯错误。这有很多工具，你可以使用它们验证你的代码从而检查代码是否有问题。下面是选出的最流行的一部分工具。很多这些工具都可以和流行的 IDE 如 Eclipse 或者 IntelliJ 集成，可以让你更快地发现代码中的错误。

- **Checkstyle**: 一个静态代码分析工具，它主要着力于保证你的代码符合代码标准。检查规则在一个 XML 文件中定义，你可以把它检入你的版本控制工具，和你的代码放在一起。
- **FindBugs**: 主要集中于发现你的代码中可能导致产生 bug 或者错误的部分。虽然作为独立的进程运行，但是对流行的 IDE 和构建工具的支持也很好。
- **PMD**: 和 FindBugs 很相似，PMD 着力于发现你代码中的错误和整理的你的代码。你可以把针对你的代码的检查规则控制在 XML 文件中，和你的代码放在一块儿提交。
- **SonarQube**: 和前面所述的工具不同，它是在本地运行的，SonarQube 启动一个服务器，你把你代码提交到这个服务器来进行分析。它提供了 web 界面，你可以看到你的代码的健康状况信息，如不好的做法，潜在的 bug，测试覆盖率百分比，和你写代码的技术水平

除了在开发工程中使用这些工具，把它们用在你的构建阶段往往也是一个不错的想法。它可以和想 Maven 或者 Gradle 这样的构建工具绑定到一起，也可以和持续集成工具绑定使用。

Eclipse Memory Analyzer

即使在 Java 中内存泄露也时有发生。幸运的是，我们有一些工具就是为此而生。[Eclipse Memory Analyzer][mat] 是我用过的最好用的解决内存泄露问题的工具。它能够获取到堆栈信息让你查阅，去发现问题所在。

有几种方法可以获取到 JVM 进程的堆栈信息，但是我用 [jmap][jmap] 工具实现：

```
$ 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
```

然后你可以用内存分析器打开 *heapdump.hprof* 文件，快看看到底是怎么回事。

Resources（资源）

Resources

这些资源能够帮你成为 Java 大牛。

Books

- Effective Java
- Java Concurrency in Practice
- Clean Code

Podcasts

- The Java Posse (*discontinued*)
- vJUG
- Les Cast Codeurs (*French*)
- Java Pub House
- Java Off Heap
- Enterprise Java Newscast

Videos

- Effective Java - Still Effective After All These Years
- InfoQ - see especially presentations and interviews
- Parleys