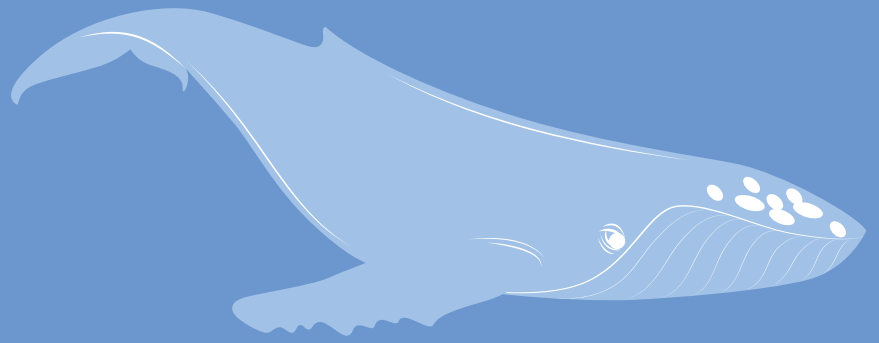
Effective
Java

Effective
Java

Java

学习效率手册



提炼 Effective Java 中的 90 条编程法则

Effective
Java

2

第2章

创建和销毁对象

条目 1　用静态工厂方法代替构造器

在设计类时，可以将静态工厂方法作为公有的构造器的替代或补充。

静态工厂方法和公有的构造器各有所长，重要的是了解其相对优势。通常应该首选静态工厂，切忌在没有考虑静态工厂的情况下本能地提供公有的构造器。

条目 2　当构造器参数较多时考虑使用生成器

静态工厂和构造器有一个共同的缺点：当可选参数非常多时，不能很好地扩展。

当我们要设计的类的构造器或静态工厂具有多个参数，特别是其中的许多参数是可选的或具有相同的类型时，生成器模式是个不错的选择。与重叠构造器模式相比，使用生成器的客户代码更容易阅读和编写，生成器也比 JavaBeans 更安全。

条目 3　利用私有构造器或枚举类型强化 Singleton 属性

有两种常见的实现 Singleton 的方式。其原理都是将构造器设置为私有的，通过导出一个公有的静态成员来提供对唯一实例的访问。第一种方式是用一个 final 的字段

作为公有的静态成员，第二种方式是提供一个公有的静态工厂方法。

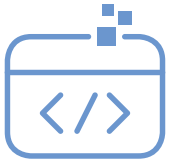
现在有了第三种方式，即声明一个只包含单个元素的枚举类型。单元素的枚举类型往往是实现 Singleton 的最佳方式。

条目 4　利用私有构造器防止类被实例化

有时需要编写仅包含静态方法和静态字段的类，这样的工具类并不是为了被实例化而设计的，可以让类包含一个私有的构造器来防止它被实例化。

条目 5　优先考虑通过依赖注入来连接资源

对于依赖于一个或多个底层资源，而且资源的行为会对其行为造成影响的类，不要使用 Singleton 或静态工具类来实现，也不要让该类直接创建这些资源。相反，应该将资源或创建资源的工厂传递给构造器（或静态工厂，或生成器）。



这种做法，也就是所谓的依赖注入，将极大地提升类的灵活性、可复用性和可测试性。

条目 6　避免创建不必要的对象

复用对象，而不是每次需要时都创建一个新的功能相同的对象，往往是更好的选择。不可变对象总是可以复用的。

对于既提供了静态工厂方法，又提供了构造器的不可变类，通常首选前者，以避免创建不必要的对象。

应该优先使用基本类型而不是其封装类，并提防无意中的自动装箱。

除非创建对象的开销极为高昂，否则通过维护自己的对象池来避免创建对象并不是好的选择。一个有正当理由使用对象池的典型例子是数据库连接。建立数据库连接的开销高到值得复用这些对象。

条目 7　清除过期的对象引用

一般来说，每当出现类自己管理内存的情形时，程序员都应该警惕内存泄漏。每当释放一个元素时，其中包含的任何对象引用都应该清除。

因为内存泄漏通常不会表现为明显的故障，所以可能会在系统中存在很多年。通常只有通过仔细地检查代码或借助堆剖析器(heap profiler)这样的调试工具才能发现。学会如何预测这样的问题，防患于未然，非常重要。

条目 8　避免使用终结方法和清理方法

终结方法（finalizer）是不可预测的，往往存在危险，而且一般来说并不必要。使用终结方法有可能导致行为不稳定、性能降低，以及可移植性问题。Java 9 引入的清理方法（cleaner），其危险性比终结方法要小，但仍然是不可预测的，而且运行很慢，一般来说也是不必要的。

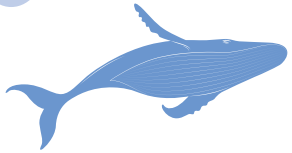
总之，不要使用清理方法或终结方法，除非作为安全网，或用于终止非关键的本地资源。

条目 9　与 try-finally 相比，首选 try-with-resources

Java 类库中有很多需要通过调用 close 方法手动关闭的资源。这样的例子包括以下：

在处理必须关闭的资源时，应该总是选择 try-with-resources，而不是 try-finally。这样得到的代码更简短、更清晰，生成的异常也更有价值。try-with-resources 语句使得编写正确的代码更容易，而使用 try-finally 是几乎无法实现的。

Effective Java



3

第 3 章

对所有对象都通用的方法

条目 10　在重写 equals 方法时要遵守通用约定

除非迫不得已，否则不要重写 equals 方法：在很多情况下，从 Object 继承的 equals 实现就能满足需要。如果确实要重写，请确保比较了所有的重要字段，并确保比较过程没有违反 equals 约定的 5 个方面：

条目 11　重写 equals 方法时应该总是重写 hashCode 方法

每当重写 equals 时都必须重写 hashCode，否则程序将不能正常运行。hashCode 方法必须遵守 Object 指定的通用约定，而且必须提供合理的功能，为不相等的实例分配不相等的哈希码。AutoValue 之类的框架为手动编写 equals 和 hashCode 方法提供了一个很好的替代方案，很多 IDE 也提供了部分类似功能。

条目 12　总是重写 toString 方法

虽然 Object 类提供了 toString 方法的一个实现，但它所返回的字符串通常不是类的用户所希望看到的。它由类名、@ 符号以及哈希码的无符号十六进制形式组成，如 PhoneNumber@adbbd。toString 的通用约定指出，所返回的字符串应该是“一个简洁但信息丰富，而且适合人阅读的表达形式”。

在我们编写的每个可实例化的类中都要重写 Object 的 toString 实现，除非有超类已经这样做了。这样会使类用起来更舒服，而且有助于调试。toString 方法应该以一种美观的格式返回对这个对象的简洁、有用的描述。

条目 13　谨慎重写 clone 方法

考虑到与 Cloneable 相关的所有问题，新的接口不应该扩展该接口，新的可扩展的类也不应该实现该接口。尽管 final 类实现 Cloneable 的危害要小得多，但应该将其视作一种性能优化，除非是有充足理由的少数情况，否则也不建议使用。一般来说，复制功能最好通过构造器或工厂来提供。

不过数组是个例外，它们最好用 clone 方法来复制。

条目 14　考虑实现 Comparable 接口

每当要实现一个可以合理地进行排序的值类时，都应该让这个类实现 Comparable 接口，这样它的实例就可以轻松地被排序、查找和用在基于比较的集合中。在 compareTo 方法的实现中，当比较字段的值时，应避免使用 < 和 > 运算符。相反，请使用基本类型的封装类中的静态 compare 方法，或使用 Comparator 接口中的比较器构造方法。

Effective Java

第4章 类和接口

条目 15 最小化类和成员的可访问性

区分设计良好的组件和设计不良的组件最重要的因素是，这个组件能在多大程度上将其内部数据和其他实现细节对别的组件隐藏起来。设计良好的组件会隐藏其所有的实现细节，并将 API 与实现清晰地隔离。然后，组件之间仅通过它们的 API 进行通信，而对彼此的内部工作一无所知。

应该尽可能（合理地）降低程序元素的可访问性。在精心设计了一个最小的公有 API 之后，应该防止任何游离的类、接口或成员成为这个 API 的一部分。除了作为常量的公有静态 final 字段外，公有类不应该有任何公有的字段。请确保公有的静态 final 字段所引用的对象是不可变的。

条目 16 在公有类中，使用访问器方法，而不使用公有的字段

公有类永远不应该暴露可变字段。公有类暴露不可变的字段的危害会小一些，但仍然值得怀疑。然而，对于包私有的类或私有的嵌套类而言，无论字段是可变的还是不可变的，有时暴露它们是可取的。

条目 17 使可变性最小化

不可变类是指其实例无法修改的类。不要轻易为每个 getter 方法编写一个对应的 setter 方法。在设计类时，除非有充分的理由将其设计为可变的，否则就应该将其设计为不可变的。如果类无法被设计为不可变的，就应该尽可能限制其可变性。减少对象可能存在的状态的数量，推断其状态就会更容易，也减少了出错的可能性。用 private final 来声明类中的每个字段，除非有充分的理由不这样做。

条目 18 组合优先于继承

继承非常强大，但也存在问题，因为它会破坏封装。只有当子类和超类之间存在真正的子类型关系时，才适合使用继承。即便如此，如果子类与超类在不同的包中，并且超类不是为继承而设计的，那么继承有可能导致脆弱性。

为了避免这种脆弱性，应该使用“组合并转发”方式而不是继承，特别是在存在一个合适的接口来实现包装器类的情况下。包装器类不仅比子类更健壮，而且也更强大。

条目 19 要么为继承而设计并提供文档说明，要么就禁止继承

设计用于继承的类是一项艰巨的任务。必须将可重写方法的所有自身使用情况写到文档中，而一旦写到文档中，就必须在类的整个生命周期内坚守承诺。子类会依赖超类的实现细节，如果我们未能遵守承诺，修改了超类，子类有可能遭到破坏。

为了让他人编写高效的子类，我们可能需要导出一个或多个受保护的方法。除非知道确实需要子类，否则最好通过将类声明为 final 的或确保没有可访问的构造器来禁止继承。

条目 20 与抽象类相比，优先选择接口

要定义支持多种实现的类型，接口通常是最佳选择。如果导出了一个不是很简单的接口，请务必考虑配合提供一个骨架实现。在可能的情况下，应该通过接口上的默认方法来提供骨架实现，以便该接口的所有实现者都可以使用。即便如此，接口上的限制通常会使得抽象类形式成为骨架实现的不二之选。

条目 21 为传诸后世而设计接口

尽管默认方法现在已经成为 Java 平台的一部分，但谨慎设计接口仍然是极其重要

的。虽然默认方法使得向现有的接口中添加方法成为可能，但这样做存在很大的风险。如果接口包含一个小缺陷，则可能会永远困扰其用户；如果接口存在严重的缺陷，则可能会毁掉包含它的 API。虽然在接口发布之后再修正一些缺陷也是有可能的，但千万不要寄希望于此。

条目 22　接口仅用于定义类型

接口应仅用于定义类型。如果只是要导出常量，不应该使用接口。

条目 23　优先使用类层次结构而不是标记类

标记类只是对类层次结构的比较蹩脚的模仿，通常不适合使用。如果想编写一个带有显式标记字段的类，应该认真考虑一下是否可以去除这个标记，并用一个层次结构来代替这个类。当我们遇到一个带有标记字段的现有类时，也应该考虑将其重构为一个层次结构。

条目 24　与非静态成员类相比，优先选择静态成员类

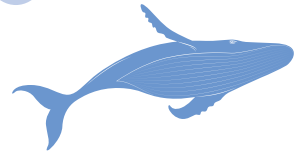
有四种不同的嵌套类，各有其应用场景。如果一个嵌套类需要在单个方法之外仍然可见，或者因为太长而不适合放在方法内部，那么就使用成员类。如果成员类的每个实例都需要一个指向其包围实例的引用，那么就把它设计为非静态的；否则就设计为静态的。

假设这个类属于一个方法的内部，如果只需要在一个地方创建其实例，并且存在一个可以描述这个类的预先存在的类型，那么就把它设计为匿名类；否则就把它设计为局部类。

条目 25　将源文件限制为单个顶层类

永远不要将多个顶层类或接口放在一个源文件中。遵循这个规则可以确保在编译时不会出现单个类的重复定义问题。这反过来又保证了编译生成的类文件和所得到的程序的行为都不会受到源文件传递给编译器时的顺序的影响。

Effective Java



第 5 章 泛型

条目 26　不要使用原始类型

使用原始类型有可能导致运行时出现异常，所以不要使用它们。提供它们只是为了保证与引入泛型之前的遗留代码的兼容性和互操作性。快速回顾一下，Set<Object> 是参数化类型，表示可以包含任何类型对象的集合；Set<?> 是通配符类型，表示只能包含某些未知类型对象的集合；而 Set 是原始类型，它脱离了泛型类型系统。

前两种是安全的，而最后一种则不安全。

条目 27　消除 unchecked 类型的警告

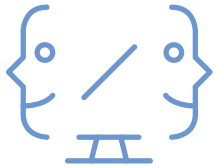
unchecked 警告非常重要。不要忽略它们。每个 unchecked 警告都代表着一个在运行时可能发生 ClassCastException 的潜在风险点。应该尽力消除这些警告。如果无法消除某个 unchecked 警告，并且可以证明引发它的代码是类型安全的，请在尽可能小的作用范围内使用 @SuppressWarnings("unchecked") 注解来抑制此警告，并将这么决定的理由写在注释中。

条目 28　列表优先于数组

数组和泛型的类型规则有很大的不同。数组是协变的和具体化的，而泛型是不变的，而且会被擦除。因此，数组提供了运行时的类型安全，但没有提供编译时的类型安全，而泛型正好相反。一般来说，数组和泛型不能很好地混用。如果混用时遇到了编译错误或警告，首先应该考虑的是将数组替换为列表。

条目 29　首选泛型类型

与需要在客户端代码中进行强制类型转换的类型相比，泛型类型更安全，而且更容易使用。当设计新的类型时，应该确保它们可以在不进行此类转换的情况下使用。这通常意味着要将其设计为泛型类型。如果有任何一个现有的类型本应该是泛型类型，但实际却不是，那就将其泛型化。



这将使这些类型的新用户使用起来更容易，同时不会破坏已有的客户端。

条目 30　首选泛型方法

泛型方法与泛型类型一样，与需要客户端对输入参数和返回值进行显式强制类型转换的做法相比，它们更安全，也更容易使用。就像类型一样，我们应该确保自己的方法不需要强制类型转换就能使用，这通常意味着使其成为泛型方法。而且像类型一样，我们应该将需要强制类型转换才能使用的现有方法泛型化。这使得新用户用起来更方便，同时不会破坏现有的客户端。

条目 31　使用有限制的通配符增加 API 的灵活性

在 API 中使用通配符类型尽管比较棘手，但能使 API 更为灵活。如果我们编写的类库将被广泛使用，一定要合理使用通配符类型。请记住基本规则，也就是 PECS 口诀“producer-extends,consumer-super”。还要记住，所有 Comparable 和 Comparator 都是消费者。

条目 32　谨慎混用泛型和可变参数

可变参数和泛型不能很好地配合，因为可变参数机制是在数组之上构建起来的存在漏洞的抽象，而数组与泛型有不同的类型规则。虽然泛型可变参数不是类型安全的，但它们是合法的。如果选择编写带有泛型（或参数化）可变参数的方法，首先要确保该方法是类型安全的，然后使用 @SafeVarargs 注解，这样使用起来就不会出现令人不快的情况了。

条目 33　考虑类型安全的异构容器

以集合 API 为例，泛型的正常使用方式会限制我们针对每个容器使用数量固定的类型参数。通过将类型参数放到键上而不是容器上，可以绕过这个限制。可以使用 Class 对象作为这种类型安全的异构容器的键。以这种方式使用的 Class 对象称为类型令牌。我们还可以使用自定义的键类型。

例如，可以用一个 DatabaseRow 类型表示数据库的一行（容器），以泛型类型 Column<T> 作为它的键。

Effective Java

第6章 枚举和注解

条目 34 使用 enum 代替 int 常量

与 int 常量相比，枚举类型的优势非常明显。枚举类型的可读性更好，也更安全，功能更强大。许多枚举不需要显式的构造器或成员，但其他一些枚举可以受益于将数据与每个常量关联起来并提供根据这些数据来执行计算的方法。还有少量枚举可以受益于将多个行为和单个方法关联起来。

在这种相对罕见的情况下，应该优先考虑特定于常量的方法，而不是在枚举上使用 switch 语句。如果一些（但不是所有的）枚举常量有共同的行为，可以考虑策略枚举模式。

条目 35 使用实例字段代替序号

Enum 的文档中对 ordinal 方法有以下说明：“大多数程序员用不到这个方法。它是设计用于基于枚举的通用数据结构的，如 EnumSet 和 EnumMap。”除非正在编写这种数据结构，否则最好完全避免使用 ordinal 方法。

条目 36 使用 EnumSet 代替位域

不能仅仅因为一个可枚举类型要放到集合中，就使用位域来表示它。EnumSet 类集位域的简洁性和性能与枚举类型的诸多优点于一身。

EnumSet 有个实际的缺点，一直到 Java 9，还无法创建不可变的 EnumSet，但这可能会在未来的版本中得到解决。在这个问题解决之前，可以使用 Collections.unmodifiableSet 将 EnumSet 封装起来，但简洁性和性能会受到影响。

条目 37 不要以序号作为索引，使用 EnumMap 代替

序号不太适合用来索引数组，应该使用 EnumMap 代替。如果要表示的关系是多维的，可以使用 EnumMap<...,EnumMap<...>>。

条目 38 使用接口模拟可扩展的枚举

虽然无法编写可扩展的枚举类型，但可以通过编写一个接口并配合一个实现了该接口的基本枚举类型来模拟。这允许客户端编写自己的实现该接口的枚举（或其他类型）。这样，只要 API 是基于同样的接口编写的，任何可以使用基本枚举类型的实例的地方，就都可以使用客户端定义的这些类型的实例。

条目 39 与命名模式相比首选注解

如果我们编写的工具需要程序员向源代码中添加信息，就可以定义适当的注解类型。在可以使用注解的情况下，没有理由再使用命名模式。即便如此，除了开发工具的程序员，大多数程序员不需要定义注解类型。但是，所有程序员都应该使用 Java 提供的预定义的注解类型。此外，考虑使用 IDE 或静态分析工具提供的注解。

这类注解可以提高这些工具所提供的诊断信息的质量。

条目 40　始终使用 Override 注解

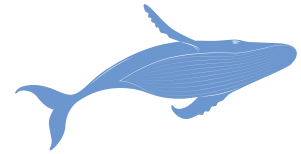
如果对我们认为要重写超类方法的任何方法声明都加上 Override 注解，那么编译器可以帮我们避免很多错误。但有一种例外情况，在具体类中，如果我们认为一个方法是在重写抽象方法声明，这时候并不需要使用 Override 注解（尽管这样做也没坏处）。

条目 41　使用标记接口来定义类型

标记接口和标记注解各有用处。如果想定义一个不带任何新方法的类型，那么应该选择标记接口。如果想标记除了类和接口之外的程序元素，或者将这个标记与已经大量使用注解类型的框架配合使用，则应该选择标记注解。

如果发现自己正在编写一个目标类型为 ElementType.TYPE 的标记注解类型，则需要花点时间考虑清楚，它是否真的应该是注解类型，是否标记接口更为合适。

Effective Java



第 7 章

Lambda 表达式和流

条目 42　与匿名类相比，优先选择 Lambda 表达式

从 Java 8 开始，Lambda 表达式已经是表示小型函数对象的最佳方式。除非我们要为其创建实例的类型不是函数式接口，否则不要使用匿名类来表示函数对象。另外，还请记住，Lambda 使得表示小型函数变得非常容易，这就打开了通往函数式编程的大门，而在此之前这是不现实的。

条目 43　与 Lambda 表达式相比，优先选择方法引用

方法引用通常可以提供比 Lambda 表达式更简洁的替代方案。如果方法引用更简短、更清晰，就使用方法引用；如果并非如此，就坚持使用 Lambda 表达式。

条目 44　首选标准的函数式接口

现在 Java 有了 Lambda 表达式，我们在设计 API 时就要考虑使用 Lambda。应该考虑接受函数式接口作为输入，以及将函数式接口作为输出返回。一般来说，最好使用 java.util.function 中提供的标准接口，但也要注意在极少数情况下，最好编写自己的函数式接口。

条目 45　谨慎使用流

有些任务最好用流来完成，有些则最好用迭代。还有许多任务最好结合这两种方式来完成。对于选择哪种方式来完成某项任务，并没有硬性的规定。如果不确定某项任务是使用流更好，还是使用迭代更好，那就两种都试试，看看哪种效果更好。

条目 46　在流中首选没有副作用的函数

编程实现流管道的精髓在于无副作用的函数对象。这适用于所有传递给流和相关对象的诸多函数对象。终结操作 forEach 应该只用来报告流执行的计算结果，而不是用来执行计算。为了正确使用流，我们必须了解收集器。最重要的收集器工厂是 toList、toSet、toMap、groupingBy 和 joining。

条目 47　作为返回类型时，首选 Collection 而不是 Stream

在编写返回元素序列的方法时，请记住，有些用户可能希望将其作为流来处理，而其他用户可能希望迭代处理。要尽量满足两个群体的需求。如果可以返回集合，就返回集合。如果这些元素已经在一个集合中，或序列中元素的数量小到有理由创建一个新集合，就返回一个标准的集合（如 ArrayList）。否则，可以考虑像我们为幂集所做的那样实现一个自定义的集合。

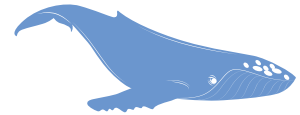
如果无法返回集合，则返回 Stream 或 Iterable，哪个看上去更自然就选哪个。如果在未来的 Java 版本中，Stream 接口声明被修改为扩展 Iterable 接口，则可以放心地返回流，因为它们将同时支持流处理和迭代。

条目 48　将流并行化时要谨慎

除非有充分的理由相信对流管道进行并行化可以保持计算的正确性并能提高速度，否则甚至不要尝试。不适当地将流并行化，其代价可能是程序运行失败，或造成性能灾难。如果你认为并行化可能是合理的，请确保你的代码在并行运行时能保持其正确性，并在真实环境下进行仔细的性能测量。

如果代码仍然是正确的，而且这些实验证明了确实可以提高性能，也只有在这时，才应该在生产代码中对这个流进行并行化。

Effective Java



8 第8章 方法

条目 49　检查参数的有效性

每次编写方法或构造器时，都需要考虑其参数应该存在哪些限制。我们应该将这些限制写到文档中，并在方法体的开头进行显式的检查，以强制实施这些限制。养成这样做的习惯非常重要。这样做并不需要很大的工作量，但是在第一次有效性检查失败时，我们就得到了连本带利的回报。

条目 50　必要时进行保护性复制

如果一个类包含会从其使用者得到或者会返回给其使用者的可变组件，则这个类必须对这些组件进行保护性复制。如果复制的成本高到难以接受，并且这个类确信其使用者不会不恰当地修改这些组件，那么可以不使用保护性复制，而是在文档中说明，不修改受影响的组件是使用者的责任。

条目 51　仔细设计方法签名

如果综合使用下面这些技巧，我们设计出的 API 会更容易学习和使用，并且不容易出错。

仔细选择方法的名称。方法的名称应始终遵循标准的命名约定。

不要以方便用户使用的名义提供过多方法。每个方法都应该发挥自己的作用。过多的方法会使类难以学习、使用、文档化、测试和维护。

避免过长的参数列表。参数最好不超过4个。再长的话，大多数程序员都记不住。

对于参数类型，应该优先使用接口而不是类。如果有合适的接口来定义参数，就使用这个接口，而不使用实现了这个接口的类。

除非从方法名称中能够明确看出要表达布尔值的含义，否则对于存在两个选项的参数，应该首选包含两个元素的枚举类型，而不是 boolean 类型。枚举使我们的代码更容易阅读和编写，以后要添加更多选项也很容易。

条目 52　谨慎使用重载

可以重载并不意味着就应该重载。一般来说，最好不要重载参数数量相同的多个方法。在某些情况下，尤其是涉及构造器时，可能无法遵循这一建议。在这些情况下，至少应该避免这样的情形：同一组参数可以通过增加强制类型转换而传递给不同的重载版本。

如果无法避免，比如我们正在改造一个现有的类来实现一个新的接口，那么应该确保所有的重载版本在面对相同的参数时表现一致。如果未能做到这一点，程序员将很难有效地使用这样的重载方法或构造器，他们也无法理解它为什么不能正常工作。

条目 53　谨慎使用可变参数

如果需要定义参数数量不固定的方法，可变参数的价值不可估量。应该将任何必需的参数加在可变参数之前，还要注意使用可变参数对性能的影响。

条目 54　返回空的集合或数组，而不是 null

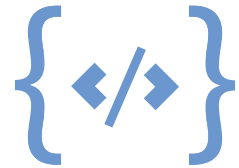
永远不要返回 null，而要返回空的数组或集合。返回 null 会使我们的 API 更难使用，更容易出错，而且在性能上并没有优势。

条目 55　谨慎返回 Optional

如果发现自己编写的方法未必总能返回一个值，而且我们认为用户在每次调用该方法时都要考虑这种可能性，或许就应该返回一个 Optional 实例。然而，我们应该意识到返回 Optional 实例会对性能造成真正的影响；对于性能关键的方法来说，返回 null 或抛出异常可能更好。最后，除了作为返回值之外，几乎不应该在其他情况下使用 Optional。

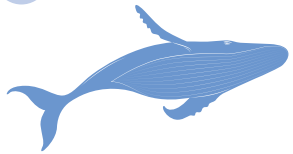
条目 56　为所有导出的 API 元素编写文档注释

文档注释是为 API 编写文档的最好、最有效的方式。对于所有导出的 API 元素，必须使用文档注释。应该采用遵循惯例的一致的风格。



记住，在文档注释中可以使用任何 HTML，但是所使用的 HTML 中的元字符必须进行转义。

Effective Java



第9章 通用编程

条目 57 最小化局部变量的作用域

将局部变量的作用域最小化，可以提高代码的可读性和可维护性，并降低出错的可能性。要最小化局部变量的作用域，最好的办法是在第一次使用它的地方进行声明。几乎每个局部变量声明都应该包含一个初始化器。

如果循环变量的内容在循环终止后不再需要，应该优先选择 for 循环而不是 while 循环。

最后一种将局部变量的作用域最小化的技术是使方法保持小且聚焦。如果把两个操作放在了同一个方法中，那么与其中一个操作相关的局部变量可能会出现在执行另一个操作的代码的作用域内。为了防止这种情况发生，只需将这个方法分成两个，每个操作对应一个方法。

条目 58 与传统的 for 循环相比，首选 for-each 循环

与传统的 for 循环相比，for-each 循环在清晰、灵活性和预防故障等方面优势明显，而且没有性能损失。应该尽可能使用 for-each 循环。

条目 59 了解并使用类库

不要重复发明轮子。如果需要执行某些看起来应该很常见的操作，可能已经有某个类库提供了这样的功能。如果有，就使用它；如果我们不知道，那就去查。

一般说来，类库代码很可能比你自己编写的代码更好，并且会随着时间的推移而不断改进。这并不是怀疑你作为程序员的能力。规模经济决定了类库代码会得到极大的关注，这是开发同样功能的大部分开发者所不具备的。

条目 60 如果需要精确的答案，避免使用 float 和 double

对于任何需要精确答案的计算，都不要使用 float 或 double。如果想让系统记录小数点的位置，并且不介意不使用基本类型所带来的不便和开销，可以使用 BigDecimal。

条目 61 首选基本类型，而不是其封装类

应该尽可能使用基本类型而不是其封装类。基本类型更简单，速度更快。如果必须使用其封装类，务必小心！自动装箱减少了使用封装类的烦琐，但并没有减少风险。当程序使用 == 运算符比较两个封装类的实例时，执行的是同一性比较，几乎可以肯定，这并不是我们想要的。

当程序执行涉及基本类型与其封装类的混合类型计算时，它会自动拆箱；而在执行拆箱时，有可能抛出 NullPointerException。最后，当程序对基本类型值进行装箱时，有可能导致开销较大而且并不必要的对象创建。

条目 62 如果其他类型更适合，就不要使用字符串

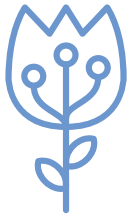
当存在更好的数据类型或可以编写更好的数据类型时，要避免将对象表示为字符串的自然倾向。如果使用不当，字符串会比其他类型更加麻烦、更不灵活、速度更慢，也更容易出错。经常被错误地用字符串来代替的类型包括基本类型、枚举类型和聚合类型等。

条目 63　注意字符串拼接操作的性能

除非性能不重要，否则不要使用字符串拼接运算符来组合多个字符串。应该使用 StringBuilder 的 append 方法来代替。或者使用字符数组，或者逐个处理每个字符串，而不是将其组合在一起。

条目 64　通过接口来引用对象

应该首选使用接口而不是类来引用对象。如果存在适合的接口类型，那么参数、返回值、变量和字段都应该使用接口类型来声明。唯一真正需要用到对象的类的时候，是使用构造器创建这个对象时。



如果养成了使用接口作为类型的习惯，程序将变得更加灵活。

条目 65　与反射相比，首选接口

反射是一种强大的机制，对于某些复杂的系统编程任务是必要的，但它也有很多缺点。如果正在编写的程序必须用到一些在编译时未知的类，可能的话，应该仅使用反射来实例化对象，并使用在编译时已知的某个接口或超类来访问这些对象。

条目 66　谨慎使用本地方法

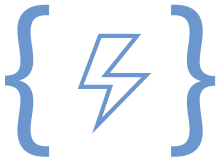
在使用本地方法之前请三思。我们很少需要使用它们来提高性能。如果必须使用本地方法来访问底层资源或本地库，应该尽可能少地使用本地代码，并对其进行彻底的测试。本地代码中的一个错误，可能会破坏整个应用程序。

条目 67　谨慎进行优化

不要努力编写快的程序，而要努力编写好的程序；程序编写得好，速度会随之而来。

但在设计系统时，尤其是在设计 API、线路层协议和持久化数据格式时，一定要考虑到性能。在完成了系统的构建之后，要测量其性能。

如果速度足够快，那就完成了。如果不够快，应该借助剖析器定位到问题的源头，并着手优化系统的相关部分。首先要检查所选择的算法：



算法选择不当，再多的底层优化都无济于事。必要时重复这个过程，每次修改后都要测量性能，直到满意为止。

条目 68　遵循普遍接受的命名惯例

Java 平台有一套完善的命名惯例（naming convention），其中许多都包含在《Java 语言规范：基于 Java SE 8》中。粗略地讲，命名惯例可以分为两类，分别是排版（typographical）惯例和语法（grammatical）惯例。

我们应该将标准的命名惯例内化在自己心里，并学着将其当作第二天性来使用。排版惯例简单明了，而且基本上没有歧义；语法惯例则更为复杂，而且比较松散。

Effective Java

10 第10章 异常

条目 69 异常机制应该仅用于异常的情况

异常机制是为异常的情况而设计的。不要将其用于普通的控制流，也不要编写强制其他人这么做的 API。

条目 70 对于可恢复的条件，使用检查型异常；对于编程错误，使用运行时异常

对于可恢复的条件，要抛出检查型异常；对于编程错误，要抛出非检查型异常。如果拿不定主意，则抛出非检查型异常。不要定义任何既不是检查型异常也不是运行时异常的可抛出类。要在检查型异常上提供帮助恢复的方法。

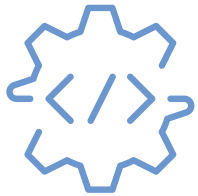
条目 71 避免不必要地使用检查型异常

适度使用检查型异常可以提高程序的可靠性；过度使用则会使 API 难以使用。如果调用者无法从失败中恢复，就抛出非检查型异常。如果有可能恢复，并且我们想强制调用者处理异常条件，首先应该考虑返回一个 Optional。

如果在失败的情况下，返回 Optional 无法提供足够的信息，才应该抛出检查型异常。

条目 72 优先使用标准异常

Java 类库提供了一组异常，可以满足大多数 API 抛出异常的需求。复用标准异常有几个好处。最主要的是，它使我们的 API 更容易学习和使用，因为它符合程序员已经熟悉的既定惯例。其次是使用我们的 API 的程序也更容易阅读，因为没有混杂不常见的异常。



最后，异常类越少，意味着内存占用越少，加载类所需要的时间也越少。

条目 73 抛出适合于当前抽象的异常

如果无法防止或处理来自底层的异常，除非底层方法所抛出的所有异常恰好适合于上层抽象，否则应该使用异常转译。异常链提供了两全其美的方法：它允许抛出一个适合于上层的异常，同时又能获得底层的 cause 进行失败原因分析。

条目 74 将每个方法抛出的所有异常都写在文档中

对于我们编写的每一个方法，应该将它可能抛出的每个异常都写在文档中。对于非检查型异常和检查型异常都是如此，对于抽象方法和具体方法也都是如此。这个文档应该采用文档注释中的 @throws 标签的形式。

应该在方法的 throws 子句中单独声明每个检查型异常，但不要声明非检查型异常。如果我们没有将方法可能抛出的异常都写在文档中，那么其他人很难（甚至根本不可能）有效地使用这些类和接口。

条目 75　将故障记录信息包含在详细信息中

当程序由于未被捕获的异常而失败时，系统会自动打印该异常的栈轨迹信息。栈轨迹信息包含该异常的字符串表示，即调用其 toString 方法的结果。通常，它由异常的类名后跟其详细消息组成。为了记录故障信息，异常的详细消息应该包含导致该异常的所有参数和字段的值。

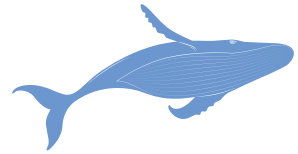
条目 76　努力保持故障的原子性

对于作为方法规格说明组成部分的异常，在抛出它们时通常应该让对象处于和调用之前相同的状态。如果违反了这一规则，API 文档应该明确指出对象将处于什么状态。遗憾的是，很多现有的 API 文档都没有做到这一点。

条目 77　不要忽略异常

虽然这条建议看起来是显而易见的，但它经常被违反，所以值得再强调一遍。当 API 的设计者声明方法会抛出一个异常时，他们是想告诉你一些重要信息。所以不要忽略它！如果选择忽略一个异常，catch 块应该包含一条注释，解释为什么这样做是合适的，而且异常变量应该被命名为 ignored。

Effective Java



第 11 章 并发

条目 78　同步对共享可变数据的访问

当多个线程共享可变数据时，每个读取或写入这些数据的线程都必须执行同步。如果没有同步，就无法保证一个线程的更改对另一个线程可见。其后果就是活性失败和安全性失败。

这些问题是最难调试的。它们可能会间歇性发作，也可能与时序有关，而且程序行为可能会因虚拟机的不同而大相径庭。如果只需要线程间通信而不需要互斥，volatile 修饰符是一种可接受的同步形式，但要正确使用它会有一定的难度。

条目 79　避免过度同步

为了避免死锁和数据损坏，永远不要在同步区域内调用外来方法。更一般地说，应该尽量减少在同步区域内执行的工作量。在设计一个可变类时，要考虑它是否应该实现自己的同步。

在多核时代，不要过度同步这一点比以往任何时候都重要。只有在理由充分的情况下才应该在类的内部进行同步，同时还要将自己的决定清清楚楚地写在文档中。

条目 80　与线程相比，首选执行器、任务和流

我们不仅应该避免编写自己的工作队列，而且通常应该避免直接使用线程。在直接使用线程时，Thread 既是工作单元，又是执行机制。而在执行器框架中，工作单元和执行机制是分开的。关键抽象是工作单元，即任务（task）。任务有两种类型：

Runnable

Callable

是 Runnable 的近亲，和它类似，
但它会返回一个值，而且有可能抛出任意异常

执行任务的一般机制是执行器服务。如果以任务的方式思考，并让执行器服务来执行这些任务，就可以灵活选择适合自己需求的执行策略，并在需求改变时更改策略。

条目 81　与 wait 和 notify 相比，首选高级并发工具

与使用 java.util.concurrent 提供的高级语言相比，直接使用 wait 和 notify 就像使用“并发汇编语言”进行编程。在新代码中很少或根本没有理由使用 wait 和 notify。如果维护的代码中使用了 wait 和 notify，请确保总是在一个 while 循环中使用标准的习惯用法来调用 wait。

通常应该优先使用 notifyAll 方法而不是 notify 方法。如果使用 notify 方法，必须非常小心地确保程序的活性。

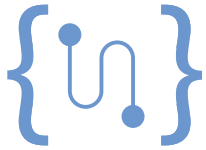
条目 82　将线程安全性写在文档中

每个类都应该使用措辞严谨的描述或线程安全注解，在文档中将其线程安全性表达清楚。synchronized 修饰符在文档中不起作用。对于有条件的线程安全的类，必须在文档中写清楚，什么样的方法调用序列需要外部同步，以及在执行这些序列时需要获取哪个锁。

如果我们要编写的是无条件的线程安全的类，应该考虑使用私有锁对象来代替同步方法。这可以使我们免受客户端程序和子类对同步的干扰，并为在以后的版本中采用更高级的并发控制方式提供更大的灵活性。

条目 83　谨慎使用延迟初始化

大多数字段应该使用正常初始化，而不是延迟初始化。如果为了实现性能目标，或为了打破不良的初始化循环依赖，而必须使用延迟初始化，那么应该使用适当的延迟初始化技术。对于实例字段，使用双重检查习惯用法；对于静态字段，使用延迟初始化 Holder 类习惯用法。

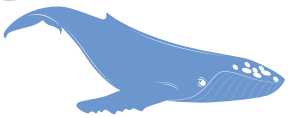


对于可以容忍重复初始化的实例字段，
还可以考虑单次检查习惯用法。

条目 84　不要依赖线程调度器

不要依赖线程调度器来保证程序的正确性。否则，得到的程序既不健壮，也不可移植。不难推出，也不要依赖 Thread.yield 或线程优先级。这些机制只是用于提示调度器。线程优先级可以适度使用，来提高一个已经在运行的程序的服务质量，但绝不能用于“修复”一个几乎无法正常工作的程序。

Effective Java



12 第12章 序列化

条目 85 优先选择其他序列化替代方案

序列化非常危险，应该尽量避免。如果正在重新设计一个系统，应该使用跨平台结构化数据表示，例如 JSON 或 protobuf。不要反序列化不可信数据。如果不得不这样做，则应该使用对象反序列化过滤器，但要知道，它不能阻挡所有攻击。应该避免编写可序列化的类。如果不得不编写，一定要非常谨慎。

条目 86 在实现 Serializable 接口时要特别谨慎

实现 Serializable 说起来容易，其实不然。除非一个类只会在受保护的环境中使用——版本之间不需要进行互操作，服务器也不会暴露给不可信的数据——否则实现 Serializable 是一个非常严肃的承诺，需要谨慎对待。如果类允许继承，更需要格外小心。

条目 87 考虑使用自定义的序列化形式

如果已经决定让类支持序列化，则应该认真考虑其序列化形式应该是什么样的。只有当默认的序列化形式是对其对象的逻辑状态的合理描述时，才使用它；否则就应该设计一个自定义的序列化形式，来恰当地描述其对象。在设计导出方法上花了多少时间，就应该为设计类的序列化形式分配多少时间。

就像不能在未来的版本中去掉导出方法一样，也不能去掉序列化形式中的字段；它们必须被永远保留，以确保序列化的兼容性。如果选错了序列化形式，会对类的复杂性和性能产生永久性的负面影响。

条目 88 保护性地编写 readObject 方法

每当编写 readObject 时，都要抱着这样的心态：我们正在编写一个公有的构造器，无论给定的字节流是什么样的，它都必须生成一个有效的实例。不要假设字节流代表的就是一个实际的序列化的实例。虽然本条目中的示例涉及的都是使用默认的序列化形式的类，但出现的所有问题，同样适用于使用自定义的序列化形式的类。

概括一下编写 readObject 方法的准则。

如果类中的对象引用字段必须保持私有，对这类字段中的每个对象执行保护性复制。不可变类的可变组件就属于这一类。

检查任何不变式，如果检查失败，抛出 InvalidObjectException。这些检查应该放在任何保护性复制之后进行。

如果在反序列化之后必须验证整个对象图，请使用 ObjectInputValidation 接口。

不要直接或间接地调用类中的任何可重写的方法。

条目 89　对于实例受控的类，首选枚举类型而不是 readResolve

应该尽可能使用枚举类型来强制实施实例控制这类不变式。

如果无法做到，而且我们需要一个实例受控的类同时支持序列化，则必须提供一个 readResolve 方法，并确保该类的所有实例字段要么是基本类型的，要么是用 transient 修饰的。

条目 90　考虑使用序列化代理代替序列化实例

对于一个不能由客户端扩展的类，如果必须为其编写一个 readObject 或 writeObject 方法，这时可以考虑序列化代理模式。而对于其不变式非常复杂的对象而言，要保证序列化的健壮性，这可能是最简单的方式。